

CS405 Computer System Architecture

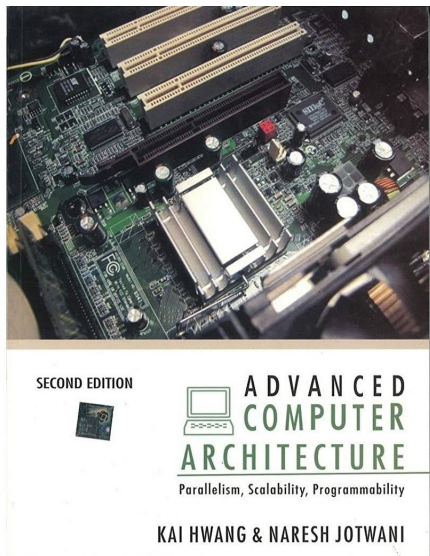
MODULE 1

Prepared by Sharika T R, Assistant Professor,
SNGCE

Syllabus

- Parallel computer models
 - Evolution of Computer Architecture,
 - System Attributes to performance,
- Amdahl's law for a fixed workload.
- Multiprocessors and Multicomputers,
- Multivector and SIMD computers,
- Architectural development tracks,
- Conditions of parallelism.

Textbook



- *Kai Hwang & Naresh Jotwani, 'Advanced Computer System Architecture, Parallelism, Scalability, Programability'*




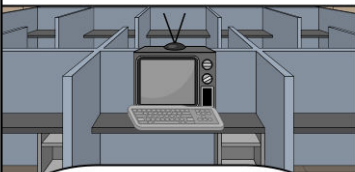
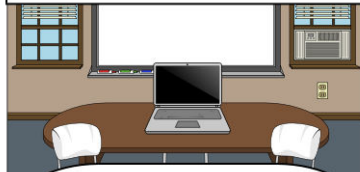

3

Computer Architecture vs Computer Organization

- **Computer Organization**
 - how operational parts of computer system are linked together
 - implements the computer architecture
- **Computer Architecture**
 - Blueprint for design and implementation of a computer system
 - provides **functional details** and behaviour of a computer system and comes before computer organization
 - **explain what a computer should do**

4

Computer Development Milestones

 <p>Long ago before all computers were invented people trusted doves to send their letters.</p>	<p>1st Generation Computers (1940-1955)</p>  <p>Charles Babbage invented the first computer. The things they used to create this, were vacuum tubes and to store data magnetic drums.</p>	<p>2nd generation of computers 1956</p>  <p>In this generation they no longer used vacuum tubes. They used something called a transistor to store data.</p>
<p>3rd generation computers (1964-1971)</p>  <p>They no longer used transistors they used integrated circuits which was more efficient.</p>	<p>4th generation computers (1971-1980)</p>  <p>Computers have become way easier to transport. They have gotten less clunky and more faster.</p>	 <p>In the 6th generation of computers, they fly and there are robots?</p>

5

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

1st
2nd
3rd
4th
5th

 - 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

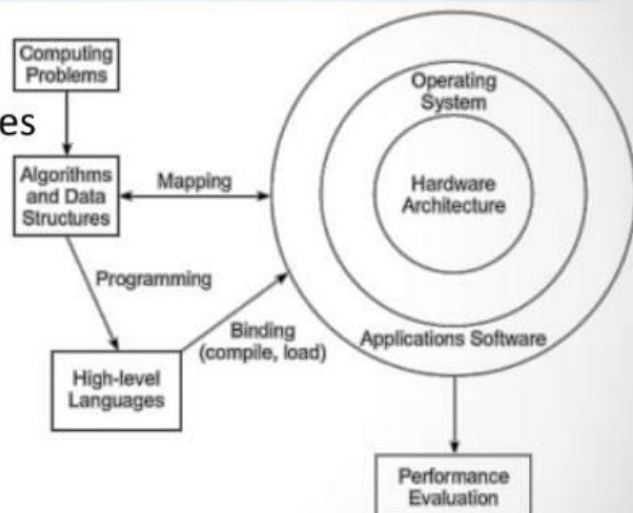


Fig. 1.1 Elements of a modern computer system

Computing problems

- **Numerical computing** - For numerical problems in science and technology, the solutions demand complex mathematical formulations and intensive integer or floating-point computations.
- **Transaction processing** - For alphanumerical 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.

13

Algorithms and Data Structures

- Special algorithms and data structures are needed to specify the computations and communications involved in computing problems.
- Most numerical algorithms are deterministic, using regularly structured data.

14

Hardware Resources

- Coordinated efforts by hardware resources, an operating system, and application software.
- Hardware core of a computer system -
 - 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.

15

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.

16

Mapping

- Mapping of algorithmic and data structures onto the machine architecture is a **bidirectional**
- process matching algorithmic structure with hardware architecture, and vice versa.
- Efficient mapping will benefit the programmer and produce better source codes.
- It includes processor scheduling, memory maps, interprocessor communications, etc

17

System Software Support

- 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.

18

17-8-2020

19

Evolution of Computer Architecture

- The study of computer architecture involves both **hardware organization** and **programing/software requirements**
- Evolution of computer architecture started with the **von Neumann architecture**
 - Built as a **sequential machine** executing **scalar data**
 - **slow due** to sequential execution of instructions in programs.
- The sequential computer was improved from **hit-serial** to **word-parallel** operations, and from **fixed-point** to **floating point operations**

20

- Major advancements came due to following techniques
 - Look ahead Parallelism and pipelining
 - Flynn's Classification
 - Parallel/Vector computers

21

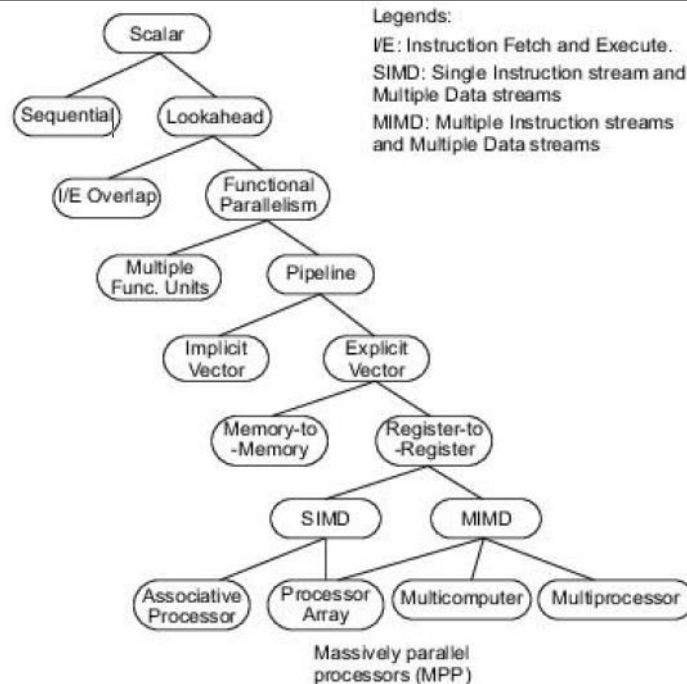


Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers

22

Look Ahead Technique

- Introduced to **prefetch** instructions in order to **overlap** I/E
 - Instruction fetch and execution operations
- and to enable **functional parallelism**
 - use multiple functional units simultaneously
 - practice pipelining at various processing levels

Lookahead is a concept where processor looks at what resources might not be needed by a process(or by processes of higher priority) in near future and allocates those to other processes(or low priority processes).

parallelism is simply multiple operations being done at the same time.

This can be achieved using mutliprocessors , coding in a way that at a time different parts of program could be executed simultaneously ..
for example threading etc.

23

Flynn's Classification

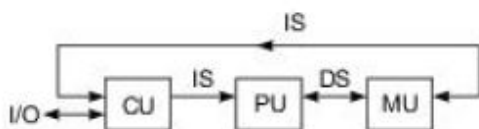
- Michael Flynn introduced classification of computer architecture based on
 - instruction stream
 - data streams

24

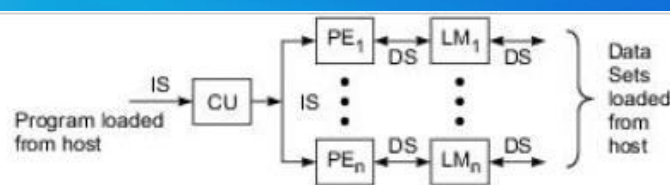
Cont..

- Major classifications are
 - **SISD** (Single Instruction Stream over a Single Data stream)
 - conventional sequential machines
 - **SIMD** (Single Instruction Stream over a Multiple Data stream)
 - vector computers equipped with scalar or vector hardware
 - **MIMD** (Multiple Instruction Stream over a Multiple Data stream)
 - Parallel Computers
 - **MISD** (Multiple Instruction Stream over a Single Data stream)
 - special purpose computer

25



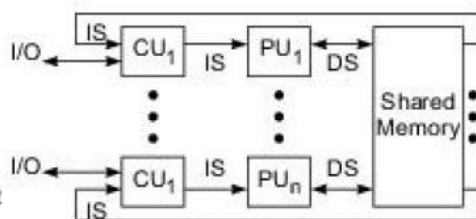
(a) SISD uniprocessor architecture



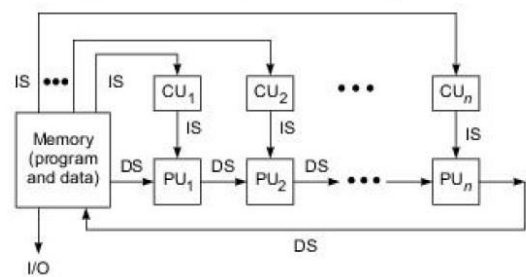
(b) SIMD architecture (with distributed memory)

Captions:

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



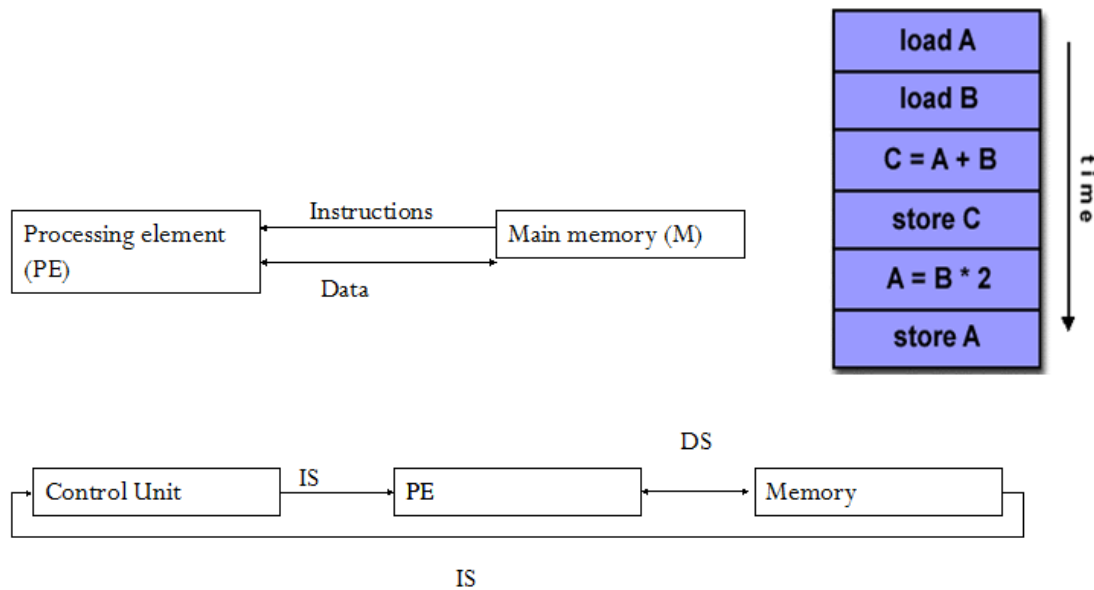
(c) MIMD architecture (with shared memory)



(d) MISD architecture (the systolic array)

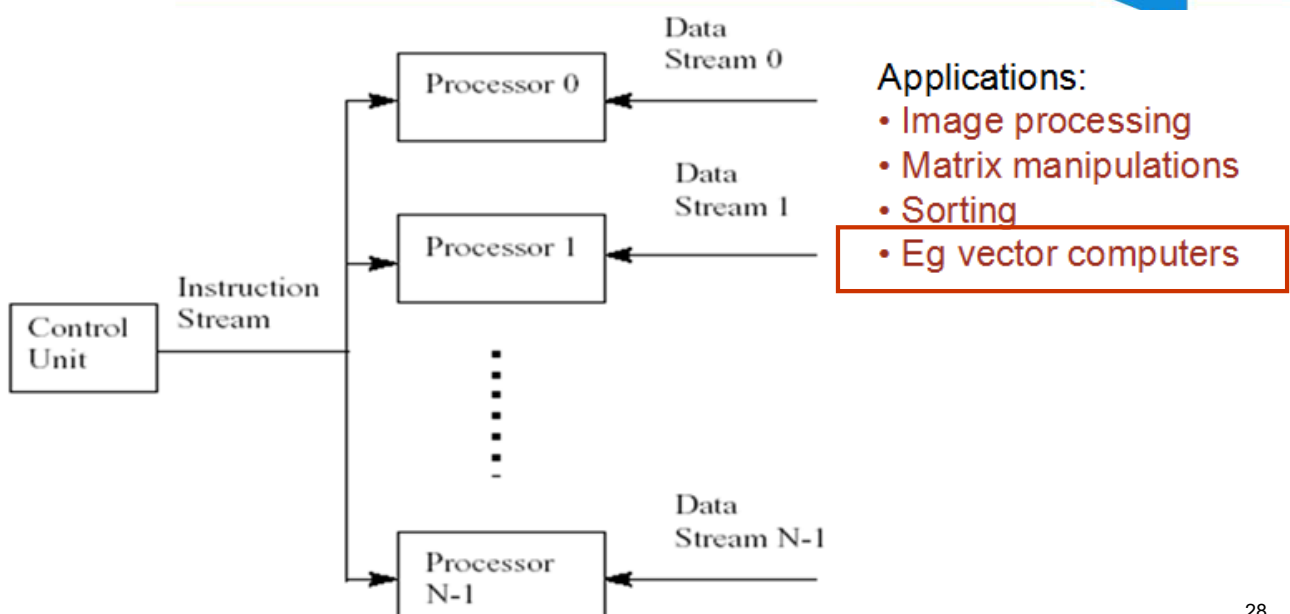
26

SISD (Single Instruction Stream over a Single Data stream)



27

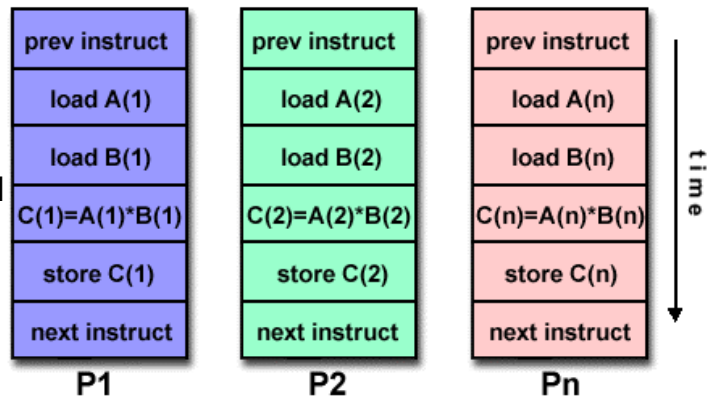
SIMD - Single Instruction stream, Multiple Data streams



28

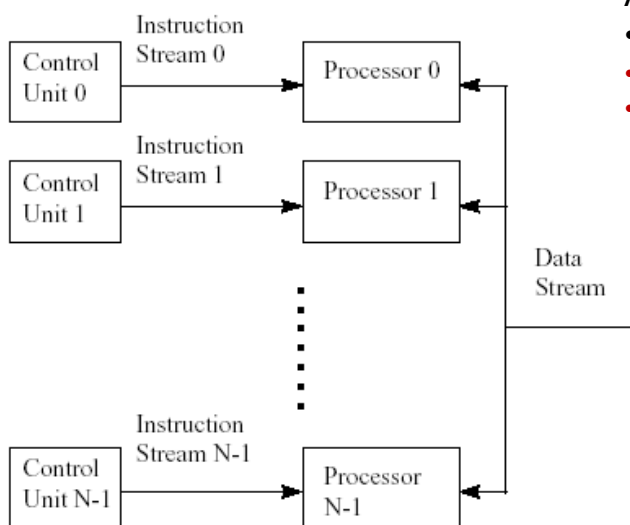
Cont..

- Fine-grained
 - Image processing application
 - Large number of PEs
 - Minimum complexity PEs
 - Programming language is a simple extension of a sequential language
- Coarse-grained
 - Each PE is of higher complexity and it is usually built with commercial devices
 - Each PE has local memory



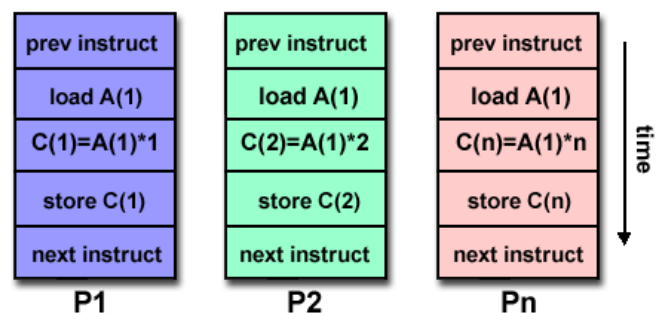
29

MISD -Multiple Instruction streams, Single Data stream



Applications:

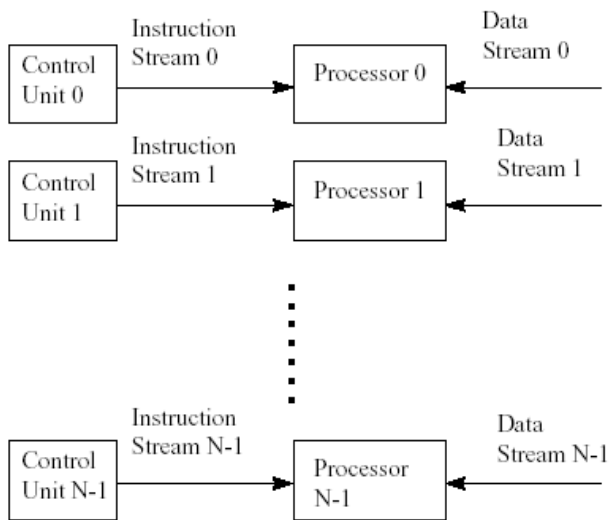
- Classification
- Robot vision
- Systolic array for pipelined execution of specific algorithms



MISD system architecture of [Fly66]

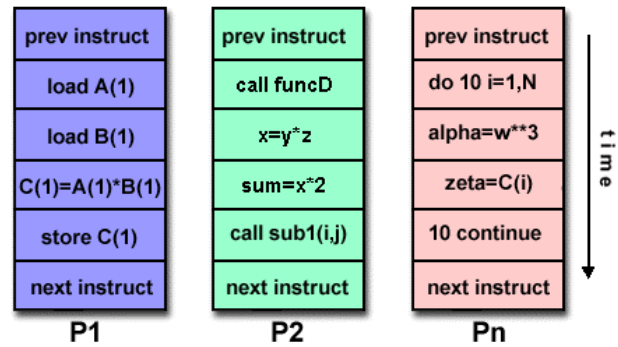
30

MIMD - Multiple Instruction streams, Multiple Data streams



Applications:

- Parallel computers
- Shared Memory



MIMD system architecture of [Fly66]

31

Advantages and Disadvantages

- Advantages of Flynn
 - » Universally accepted
 - » Compact Notation
 - » Easy to classify a system
- Disadvantages of Flynn
 - » Very coarse-grain differentiation among machine systems
 - » Comparison of different systems is limited
 - » Interconnections, I/O, memory not considered in the scheme

32

Parallel/ Vector Computer

- execute programs in MIMD mode(simultaneously)
- two major classes
 - shared memory multiprocessor
 - message passing multicomputer
- The major distinction between multiprocessors and multicomputers lies in
 - memory sharing and
 - the mechanisms used for interprocessor communication.

33

Shared Memory Multiprocessor

- processors in a multiprocessor system communicate with each other through
 - shared variables
 - in a common memory

34

Message Passing Multicomputer

- Processors in a multiprocessor system communicate with each other through
 - message passing among the nodes
 - with local memory, unshared with other nodes

35

Vector Processor

- Processors whose instructions operate on vector data
- equipped with multiple vector pipelines that can be concurrently used under hardware or firmware control
- two families of pipelined vector processors
 - Memory-to-memory architecture
 - supports the pipelined flow of vector operands directly from the memory to pipelines and then back to the memory.
 - Register-to-register architecture
 - uses vector registers to interface between the memory and functional pipelines.

36

19-8-2020

37

System Attributes to Performance

- The ideal performance of a computer system demands a perfect match between **machine capability and program behavior**
- Machine capability can be enhanced with
 - better hardware technology,
 - innovative architectural features, and
 - efficient resources management

38

- factors affecting program behavior
 - algorithm design,
 - data structures,
 - language efficiency,
 - programmer skill, and
 - compiler technology
- performance should be described as a range or a distribution

39

Performance Factors

- cycle time, τ
- clock rate f , $f = 1/\tau$
- CPI(Cycle per Instructions)
- Instruction count(I_C)
- Processor cycle p
- Memory cycle m
- Ratio between memory cycle and process cycle k

40

- Clock Rate and CPI

- cycle time τ

- time taken to complete one clock cycles

- clock rate f , $f = 1/\tau$

- inverse of clock time

- Instruction count(I_c)

- number of machine instructions to be executed in the program

- Cycles Per Instruction(CPI)

- for measuring the time needed to execute each instruction

41

Performance Factors

- CPU time (T in seconds/program)

- time needed to execute the program is estimated by finding the product of three contributing factors:

- $$T = I_c \times CPI \times \tau \quad (1.1)$$

- execution of an instruction requires going through a cycle of events involving the instruction fetch, decode, operand fetch, execution, and store results

Instruction fetch	→ memory access
Decode	→ carried out by CPU
Operand fetch	→ memory access
Execution	→ carried out by CPU
Store result	→ memory access

42

- memory cycle m
 - time needed to complete one memory reference
 k times the processor cycle τ .
 - value of k depends on
 - the speed of the cache and
 - memory technology and
 - processor-memory interconnection scheme used

43

Cycles per Instruction(CPI)

- C = total number of clock cycles needed to execute a program(n instructions)
- CPI=no of clock cycles needed to execute a **single instruction**

$$CPI = \frac{C}{I_c} \quad (3)$$

- The eqn 1.1 can be rewritten as

$$T = I_c * CPI * \tau$$

$$\rightarrow T = \frac{I_c * C * \tau}{I_c}$$

$$\rightarrow T = C * \tau \dots\dots\dots(4)$$

$$\rightarrow T = \frac{C}{f} \dots\dots\dots(5)$$

44

- CPI can be divided into two component terms
 - total processor cycles and
 - memory cycles needed to complete the execution of the instruction
- Depending on the instruction type, the complete instruction cycle may involve one to as many as four memory references
 - we can rewrite Eq. 1.1 as

$$T = I_c \times (p + m \times k) \times \tau \quad (1.2)$$

$p \rightarrow$ no: of processor cycles needed for inst decode & execution

$m \rightarrow$ no: of memory reference needed

$k \rightarrow$ ratio between memory and processor cycles

$\tau \rightarrow$ processor cycle time

45

System attributes

- five performance factors (I_c , p , m , k , τ) are influenced by four system attributes:
 - instruction-set architecture
 - affects the program length (I_c) and processor cycles needed (p)
 - compiler technology
 - affects the values of I_c , p , and the memory reference count (m).
 - CPU implementation and control, and
 - determine the total processor time ($p \cdot \tau$) needed
 - cache and memory hierarchy
 - affect the memory access latency ($k \cdot \tau$).

46

Table 1.2 Performance Factors versus System Attributes

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

MIPS(Million Instructions Per Second) Rate

- MIPS rate is based on following factors
 - clock rate f
 - Instruction count I_c
 - CPI of given machine

$$MIPS = \frac{I_c}{T \times 10^6} \quad (6)$$

- Let C be the total number of clock cycles needed to execute a given program

$$T = I_c \times (P + M \times K) \times Z \quad (1.2)$$

\downarrow
 C

$$T = C \times Z$$

$$CPI = \frac{C}{I_c} \quad (3)$$

$$\begin{aligned} T &= I_c \times CPI \times Z \quad (1.1) \\ &= I_c \times \frac{C}{I_c} \times Z \\ &= C \times Z \end{aligned}$$

$$T = I_c \times CPI \times Z \quad (1.1)$$

$$= I_c \times CPI \times \frac{1}{f}$$

$$\begin{aligned} \text{MIPS rate} &= \frac{I_c}{T \times 10^6} \\ &= \frac{I_c}{I_c \times CPI \times 10^6} \times f \end{aligned}$$

$$\begin{aligned} &= \frac{f}{CPI \times 10^6} \\ &= \frac{f}{C \times 10^6} \times I_c \quad (1.3) \end{aligned}$$

Based on 1.3 the CPU time in eqn 1.2 can be written as:

$$T = \frac{I_c \times 10^{-6}}{\text{MIPS}}$$

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

$$\text{MIPS} = \frac{f}{CPI \times 10^6}$$

Steps

$$T = C \times Z = C \times \frac{1}{f}$$

$$\text{MIPS} = \frac{f \times I_c}{C \times 10^6}$$

$$C = \frac{f \times I_c}{\text{MIPS} \times 10^6}$$

$$T = C \times \frac{1}{f}$$

$$= \frac{f \times I_c}{\text{MIPS} \times 10^6} \times \frac{1}{f}$$

$$= \frac{I_c}{\text{MIPS} \times 10^6}$$

$$= \frac{I_c \times 10^{-6}}{\text{MIPS}}$$

Throughput Rate

- System throughput W_s
 - how many programs a system can execute per unit time
- CPU throughput W_p
 - measure of how many programs can be executed per second
 - multiprogrammed system system throughput is often lower than the CPU throughput

$$\begin{aligned}
 W_p &= \frac{P}{I_c \times CPI} \quad (1.4) \\
 &= \frac{P}{CPI} \times \frac{1}{I_c} \\
 &= MIPS \times 10^6 \times \frac{1}{I_c}
 \end{aligned}$$

51

Performance

- ❖ For some program running on machine A,
Performance of A, $Perf(A) = 1 / ExecTime(A)$
- ❖ "A is n times faster than B" iff
 $Perf(A) / Perf(B) = ExecTime(B) / ExecTime(A) = n$
- ❖ "A is X% faster than B" iff
 $Perf(A) / Perf(B) = ExecTime(B) / ExecTime(A) = 1 + X/100$

52

- **Problem:**

- Machine A runs a program in 20 seconds. Machine B runs the same program in 25 seconds. How many times faster is machine A?

$$n = \text{ExecTime(B)} / \text{ExecTime(A)}$$

$$= \frac{25 \text{ sec}}{20 \text{ sec}} = 1.25$$

Machine A is 1.25 times faster than Machine B

53

20/8

54

Problem

A 40 MHz processor was used to execute a benchmark program with the following instruction mix and clock cycle counts:

Instruction Type	Instruction count	Clock cycle count
Integer Arithmetic	35000	1
Data Transfer	20000	2
Floating point	15000	2
Control Transfer	6000	2

Determine the effective CPI, MIPS rate and execution time for this program.

2018 regular question paper

55

- Clock speed=40Mhz

Instruction Type	Instruction count	Clock cycle count	Cycles
Integer Arithmetic	35000	1	$35000 \times 1 = 35000$
Data Transfer	20000	2	$20000 \times 2 = 40000$
Floating point	15000	2	$15000 \times 2 = 30000$
Control Transfer	6000	2	$6000 \times 2 = 12000$
Instruction Count	76000		

Total number of cycles required to execute complete program $C = 35000 + 40000 + 30000 + 12000 = 117000$ cycles

56

$$CPI = \frac{C}{I_c} = \frac{117000}{76000} = 1.53$$

$$MIPS = \frac{f}{CPI \times 10^6} = \frac{40 \times 10^6}{1.53 \times 10^6} = 26.14$$

$$T = I_c \times CPI \times \tau = I_c \times CPI \times \frac{1}{f} = 76000 \times 1.53 \times \frac{1}{40} = 2907 = 2.907ms$$

57

Consider the execution of a task with 100000 instructions on 500 MHz processor. The program consists of **FOUR** major types of instructions:

Instruction Type	CPI	Instruction%
Integer arithmetic	1	60%
Floating point arithmetic	2	20%
Load/Store	4	10%
Memory Reference	6	10%

When the task is executed on a uniprocessor;

- Calculate the average CPI?
- Determine the corresponding **MIPS** rate?

58

Solution:

Average CPI = $1 * 0.6 + 2 * 0.2 + 4 * 0.1 + 6 * 0.1 = 2$ cycles/instruction

$$MIPS = \frac{f}{CPI \times 10^6} = \frac{500 MHz}{2 \text{ cycles/instr}} = 250$$

59

Floating Point Operations per Second

- Performance measured in FLOPs
- FLOPS: floating point operations per second

60

AMDAHL'S LAW FOR FIXED WORKLOAD

61

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.

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

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.

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]*

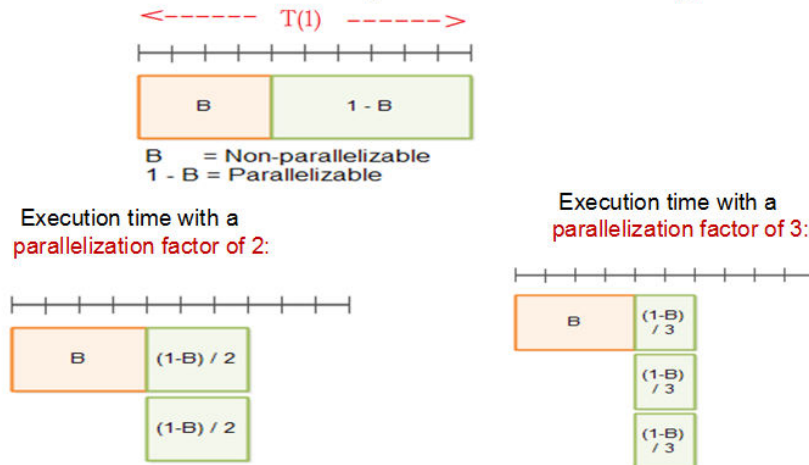
Sun & Ni's Law

[applied to scaled problems bounded by memory capacity]

Amdahl's Law

- A program (or algorithm) which can be parallelized can be split up into two parts:
 - A part which cannot be parallelized and
 - A part which can be parallelized
- Eg:
- Imagine a program that processes files from disk. A small part of that program may scan the directory and create a list of files internally in memory. After that, each file is passed to a separate thread for processing. The part that scans the directory and creates the file list cannot be parallelized, but processing the files can be done in parallel.
- Total time taken to execute the program only serially is called T.
- The time T includes the time of both the non-parallelizable and parallelizable parts.
- T = Total time of serial execution
- B = Total time of non- parallelizable part
- T - B = Total time of parallelizable part (when executed serially, not in parallel)

- First of all, a program can be broken up into a non-parallelizable part B , and
- a parallelizable part $1-B$, as illustrated by this diagram:
- The line with the delimiters on at the top is the total execution time $T(1)$.



67

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 T_s taken to perform sequential operations be a fraction α ($0 < \alpha \leq 1$) of the total execution time $T(1)$ of the program, then the time T_p to perform parallel operations shall be $(1-\alpha)$ of $T(1)$

Amdahl's Law

Thus, $T_s = \alpha.T(1)$ and $T_p = (1-\alpha).T(1)$

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

$$T(n) = T_s + T_p/n = \alpha.T(1) + \frac{(1-\alpha).T(1)}{n}$$

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.

$$\text{As } n \rightarrow \infty, S(n) \rightarrow 1/\alpha$$

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

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!***

Example: 1

Suppose that a calculation has a 4% serial portion,

- a) What is the limit of speedup on 16 processors?
- b) What is the maximum speedup?

Ans:

- a) Limit of speedup on 16 processors
 $= 16 / (1 + (16 - 1) * 0.04) = 10$

- b) The maximum speedup $= 1 / \alpha$
 $= 1 / 0.04 = 25$

$$S_n = \frac{n}{1 + (n - 1)\alpha}$$

Example: 2

If 90% of a calculation can be parallelized, then what is the maximum speed-up which can be achieved on 5 processors?

Ans: $S(n) = n / (1 + (n - 1) * \alpha)$ ($\alpha = 1 - 0.9 = 0.1$) (sequential fraction)
 $= 5 / (1 + (5 - 1) * 0.1) = 3.57$

(the program can theoretically run 3.57 times faster on five processors than on one)

22-8-2020

73

Amdahl's Law (A different perspective)

- The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's law.
- Amdahl's law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Law defines the term 'speedup'.

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

Alternatively,

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

74

- Amdahl's law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

- Fraction enhancement

- The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement.

- Speed-up-enhanced

- how much faster the task would run if the enhanced mode were used for the entire program

75

Fraction enhancement

- Example:
- If 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement, the fraction enhanced is $20/60$.
- This value, $\text{Fraction}_{\text{enhancement}}$, is always less than or equal to 1.

76

Speedup enhancement

- The speedup or improvement gained by the enhanced execution mode, that is, **how much faster the task would run if the enhanced mode were used for the entire program** – this value is the **time of the original mode over the time of the enhanced mode**.
- Example, if the enhanced mode takes 2 sec for a portion of the program, while it is 5 sec in the original mode, the speedup enhanced is $5/2$.
- This is called **Speed-up-enhanced** which is **always greater than 1**.

77

Amdhal's law can serve as a guide to how much an enhancement will improve performance and how to distribute resources to improve cost/performance

78

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

79

- If three different enhancements use fractions of time (f_1 , f_2 and f_3) respectively, and have individual speedups as S_1 , S_2 and S_3 respectively, the overall speedup is

$$\text{OverallSpeedup} = \frac{1}{\left(1 - (f_1 + f_2 + f_3) \right) + \left(\frac{f_1}{s_1} + \frac{f_2}{s_2} + \frac{f_3}{s_3} \right)}$$

80

Applications of Amdahl's Law

- Amdahl's law gives us a quick way to find the speedup from some enhancement.
- Amdahl's law is particularly useful for comparing the overall system performance of two different systems
- It can also be applied to compare two processor design alternatives, based on enhancement on the same system

81

Exercise and Solution on Amdahl's Law

1. What is the overall speedup if you make 10% of a program 90 times faster?
2. What is the overall speedup if you make 90% of a program 10 times faster?

• Amdahl's law $OverallSpeedup = \frac{1}{(1-f) + \frac{f}{s}}$

- What is the overall speedup if you make 10% of a program 90 times faster?

$$\frac{1}{(1-0.1) + \frac{0.1}{90}} \approx \frac{1}{0.9011} \approx 1.11$$

- What is the overall speedup if you make 90% of a program 10 times faster?

$$\frac{1}{(1-0.9) + \frac{0.9}{10}} = \frac{1}{0.19} \approx 5.26$$

3. We are considering an enhancement to the processor of a web server. The new CPU is 20 times faster on search queries than the old processor. The old processor is busy with search queries 70% of the time, what is the speedup gained by integrating the enhanced CPU?

$$Speedup = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$$Fraction_{enhanced} = 70\% = 0.70$$

$$Speedup_{enhanced} = 20$$

$$Speedup = \frac{1}{(1 - 0.70) + \frac{0.70}{20}} = \frac{1}{0.335} = 2.985$$

4. We are considering an enhancement to the processor of a server. The new CPU 10X faster. I/O bound server, so 60% time waiting for I/O.

- New CPU 10X faster
- I/O bound server, so 60% time waiting for I/O

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$$= \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56$$

Three enhancements with the following speedup are proposed for a new architecture. $S_1=30$, $S_2=20$, $S_3 = 15$. If enhancements 1 and 2 are each usable for 25% of the time, what fraction of the time must enhancement 3 be used to achieve an overall speed up of 10?

$$\text{OverallSpeedup} = \frac{1}{\left(1 - (f_1 + f_2 + f_3)\right) + \left(\frac{f_1}{s_1} + \frac{f_2}{s_2} + \frac{f_3}{s_3}\right)}$$

$$10 = 1 / [1 - (0.25 + 0.25 + f_3) + [(0.25/30) + (0.25/20) + (f_3/10)]]$$

$$10 = 1 / [0.5 - f_3 + [(0.5 + 0.75 + 4 f_3) / 60]]$$

$$10 = 60 / [30 - 60 f_3 + 1.25 + 4 f_3]$$

$$-56 f_3 = 6 - 31.25$$

$$f_3 = -25.25 / -56 = 0.45 = 45\%$$

85

Programming Environments

- programmability of a computer depends on the programming environment provided to the users.
- uniprocessor computers are programmed in a sequential environment
 - Successive system calls must be serialized
- When using a parallel computer,
 - one desires a parallel environment where parallelism is automatically exploited.
 - Language extensions or new constructs must be developed to specify parallelism or
 - to facilitate easy detection of parallelism at various granularity levels by more intelligent compilers.

86

Implicit Parallelism

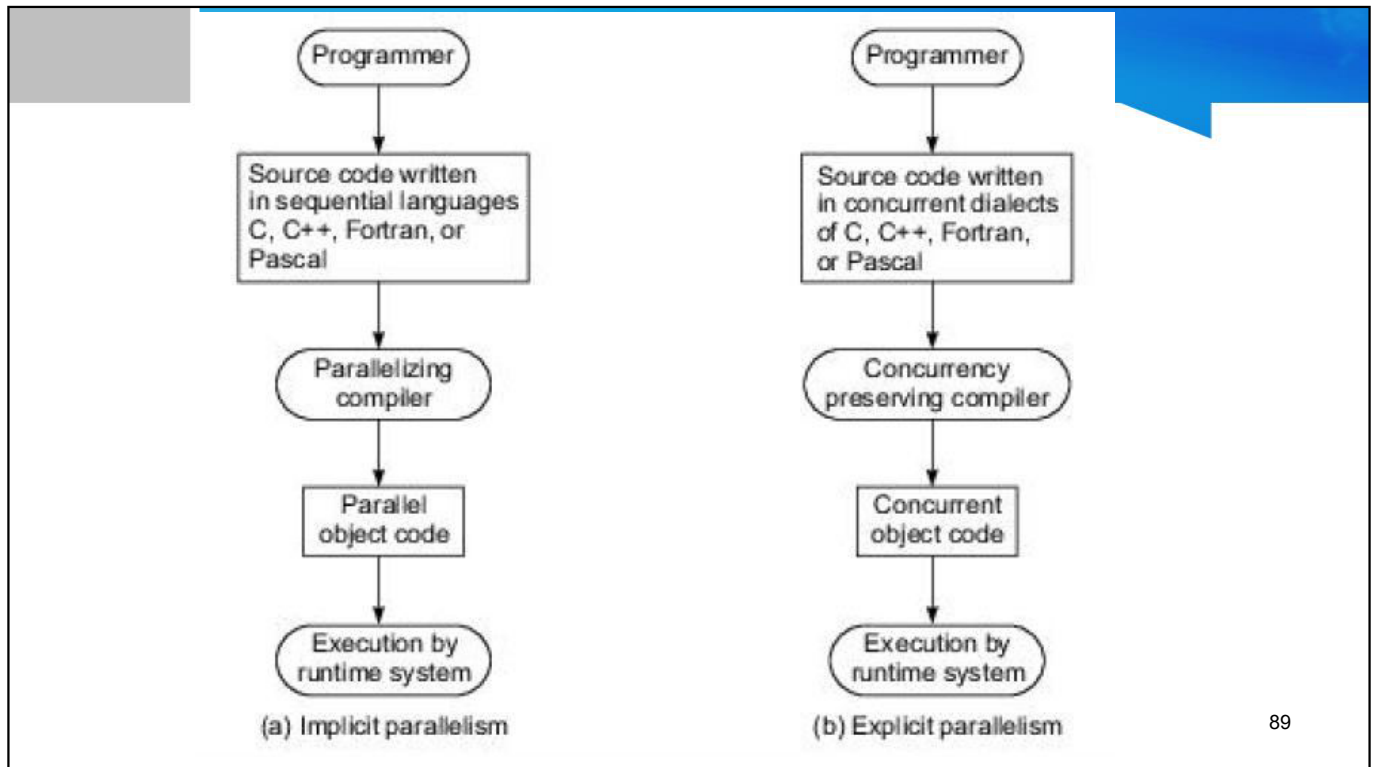
- uses a conventional language, such as C, C++, Fortran, or Pascal, to write the source program.
- The sequentially coded source program is translated into parallel object code by a parallelizing compiler.
- this **compiler must be able to detect parallelism** and assign target machine resources.
- With parallelism being implicit, success relies heavily on the “intelligence” of a parallelizing compiler.
- This approach requires less effort on the part of the programmer

87

Explicit Parallelism

- requires more effort by the programmer to develop a source program using parallel dialects of C, C++, Fortran, or Pascal.
- Parallelism is explicitly specified in the user programs.
- This **reduces the burden on the compiler to detect parallelism.**
- Instead, the compiler needs to preserve parallelism and, where possible, assigns target machine resources.

88



MULTIPROCESSORS AND MULTICOMPUTERS

- Two categories of parallel computers are
 - Shared Memory Multiprocessors
 - Message Passing Multicomputers
- Their difference is based on
 - One has shared common memory
 - Other has unshared distributed memory

Shared-Memory Multiprocessors

- three shared- memory multiprocessor models
 - Uniform Memory Access(UMA model)
 - Non Uniform Memory Access(NUMA model), and
 - Cache Only Memory Architecture(COMA) model.
- These models differ in how the memory and peripheral resources are shared or distributed.

91

The UMA Modal

- Physical memory is **uniformly shared** by all the processors
- 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

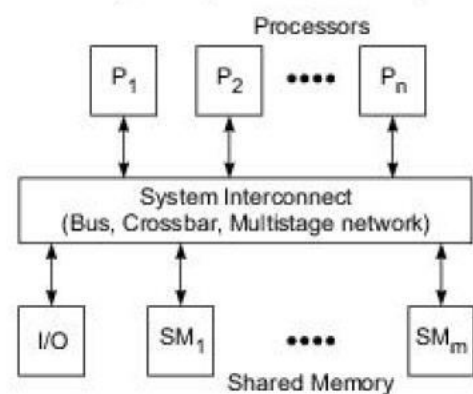


Fig. 1.6 The UMA multiprocessor model

92

- 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
- UMA model is suitable for
 - general-purpose and times sharing 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.

93

24-08

94

Symmetric and Asymmetric Multiprocessor

- Symmetric Multiprocessor
 - When all processors have equal access to all peripheral devices,
 - all the processors are equally capable of running the executive programs, such as the OS kernel and I/O service routines.

95

- Asymmetric Multiprocessor
 - only one or a sub set of processors are executive-capable.
 - An executive or a master processor(MP) can execute the operating system and handle I/O.
 - The remaining processors have no I/O capability and thus are called attached processors(APs).
 - Attached processors execute user codes under the supervision of the master processor.
 - In both MP and AP configurations, memory sharing among master and attached processors is still in place.

96

Cache coherent UMA

- If one processor update location in shared memory, all other processors know the update

97

Advantages of UMA

- Suitable for general purpose and timesharing applications by multiple users
- Speed up execution of a single large program in time critical applications

98

Non Uniform Memory Acces, NUMA Model

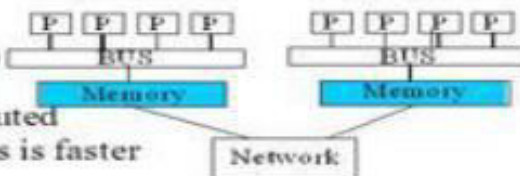
- It is shared-memory system in which the access time varies with the location of the memory word
- Two NUMA machine models are
 - Shared local memories model
 - A hierarchical cluster model

99



Uniform memory access (UMA)
Each processor has uniform access time to memory - also known as symmetric multiprocessors (SMPs) (example: SUN ES1000)

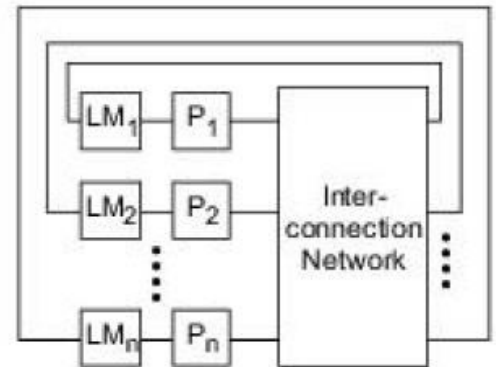
Non-uniform memory access (NUMA)
Time for memory access depends on location of data; also known as Distributed Shared memory machines. Local access is faster than non-local access. Easier to scale than SMPs (example: SGI Origin 2000)



100

Shared local memories model

- 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
- Access
 - 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.



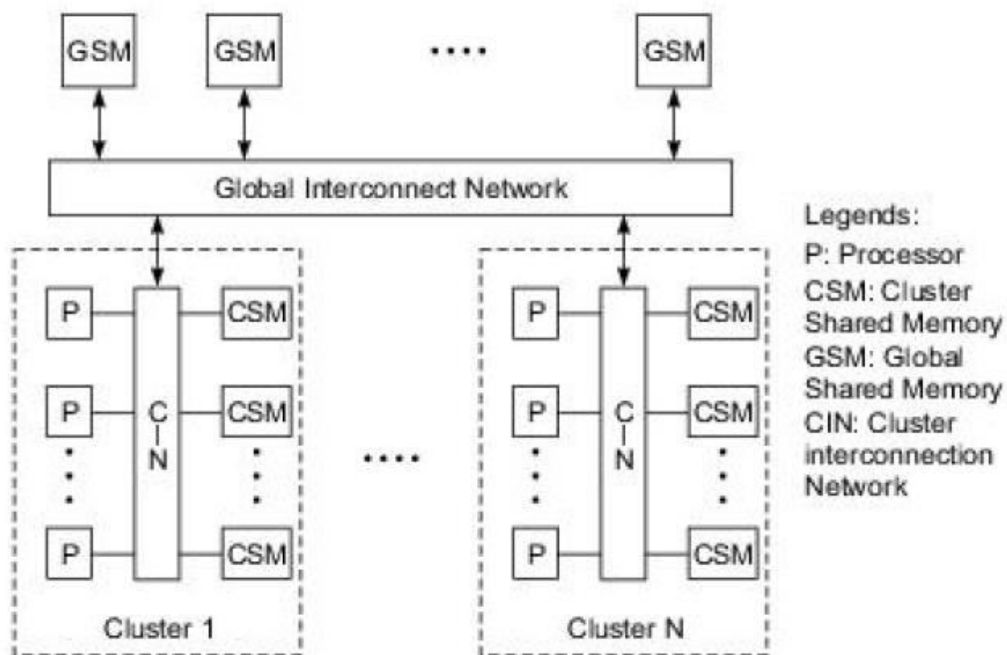
(a) Shared local memories (e.g. the N Butterfly)

101

A hierarchical cluster model

- Globally shared memory
- The processors are divided into several clusters.
- Each cluster is itself an UMA or a NUMA multiprocessor.
- The clusters are connected to global shared-memory modules.
- The entire system is considered a NUMA multiprocessor.
- All processors belonging to the same cluster are allowed to uniformly access the cluster shared-memory modules.
- All clusters have equal access to the global memory.
- However, the access time to the cluster memory is shorter than that to the global memory.

102



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

103

The COMA Model

- A multiprocessor using cache-only memory assumes the COMA model.
- Special case of NUMA
- 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
- Depending on the interconnection networks sometimes hierarchical directories may be used to help locate copies of each blocks.
- Initial dataplacement is not critical because **data will eventually migrate to where it will be used**

104

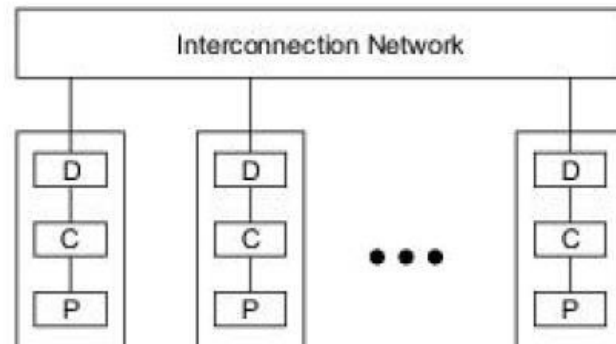


Fig. 1.8 The COMA model of a multiprocessor (P: Processor, C: Cache, D: Directory; e.g. the KSR-1)

105

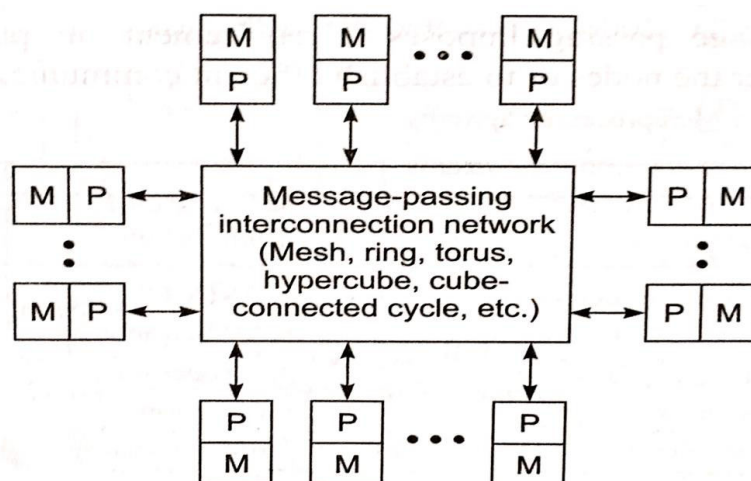
- Remote cache access is assisted by the distributed cache directories
- Depending on the interconnection network used.
- sometimes hierarchical directories may be used to help locate copies of each blocks.
- Initial dataplacement is not critical because data will eventually migrate to where it will

106

Distributed-Memory/Message Passing Multicomputers

- system 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
- Nodes interconnected by a message passing network
 - can be Mesh, Ring, Torus, Hypercube etc (discussed later)
- All interconnection provide **point to point static connection** among nodes

107



Generic model of a message-passing multicomputer

108

- Local memories are private and accessible only by processor – Thus Multicomputer are also called No-remote-Memory –Access(NORMA)(difference with UMA and NUMA)
- Communication between nodes if required is carried out by passing messages through static connection network.

109

- Advantages over Shared Memory
 - Scalable and Flexible : we can add CPU's
 - Reliable and accessible : since with Shared memory a failure can bring the whole system down
- Disadvantage
 - Considered harder to program because we are used to programming on common memory systems.

110

MULTIVECTOR and SIMD COMPUTERS

- We can classify supercomputers into 2
 - **Pipelined Vector machines** using powerful processors equipped with vector hardware
 - **SIMD computers** emphasizing on massive data parallelism

111

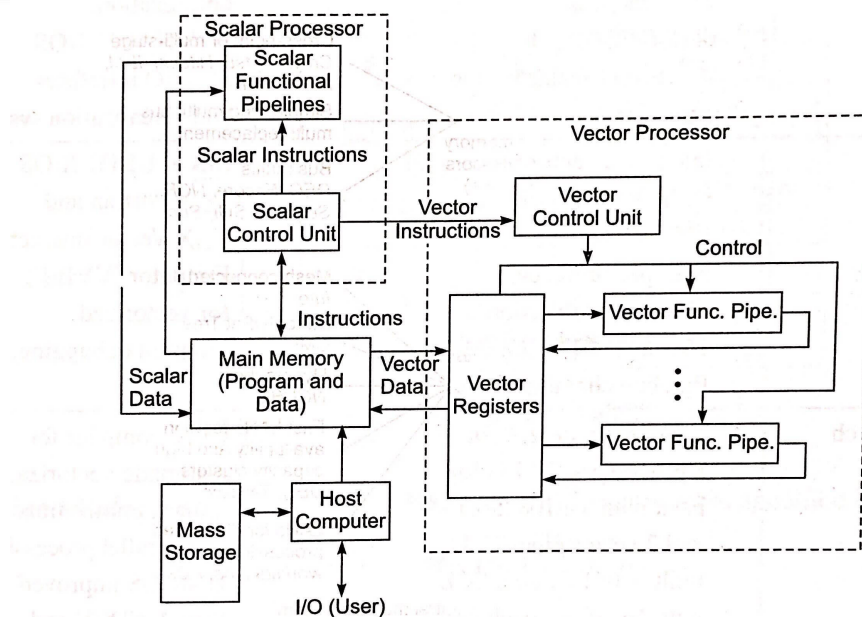
VECTOR SUPERCOMPUTERS

- A vector computer is usually built on top of scalar processor
 - its attached to scalar processor as an optional feature
- **host computer**: load program and data into main memory
- **scalar Control Unit**: decode instructions
- if it's a **scalar operation** or program control operation,
 - it will be directly executed using scalar functional pipelines.

112

- If its **vector operation** it will be send to the vector control unit.
 - The control unit supervises the flow of vector data between main memory and vector functional pipelines.
 - Vector data flow is coordinated by the control unit.
 - A number of **vector functional pipelines** may be built into a vector processor

113



114

VECTOR PROCESSOR MODELS

- 2 types :
 - Register to Register architecture
 - Memory to memory architecture

115

REGISTER to REGISTER architecture

- **Vector registers** are used to hold
 - 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.

116

- Length of vector register is usually fixed
- Some machines use reconfigurable vector registers to dynamically match register length
- Generally there are fixed no: of vector registers and functional pipelines in vector processors
 - hence they must be reserved in advance to avoid conflicts

117

MEMORY-to-MEMORY architecture

- differs from a register-to-register architecture in the use of a **vector stream unit** to replace the vector registers.
- Vector operands and results are directly retrieved from and stored into the main memory in superwords

118

SIMD Supercomputers

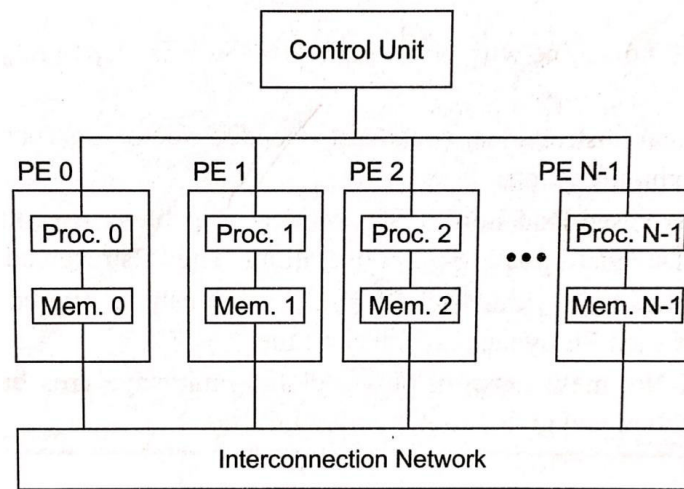
- Computers with Multiple Processing elements
- They perform same operation on multiple data points simultaneously
- An operational model of an SIMD computer is specified by a 5-tuple:

- $$M = (N, C, I, M, R)$$

119

- **N** is the number of Processing Element's(PE's) in m/c (ex: Illiac IV has 64 PE's)
- **C** is the set of instructions directly executed by Control Unit(CU)- including scalar and program flow control instructions.
- **I** is the set of instruction broadcast by CU to all PE's for parallel execution(Ex: arithmetic, logic , data routing, masking etc)
- **M** is the set of masking schemes which sets each PE's into enabled and disabled mode
- **R** is the data-routing function, specifies the pattern to be set up in the interconnection n/w for inter-PE communications.

120



Operational model of SIMD computers

121

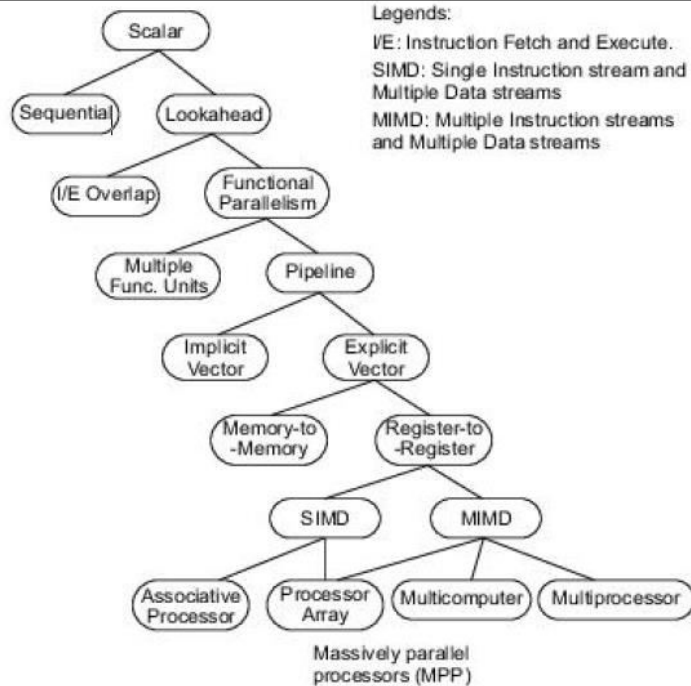


Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers

122

ARCHITECTURAL DEVELOPMENT TRACKS

- The architectures of most existing computers follow certain development tracks.
- Understanding features of various tracks provides insights for new architectural development
 1. Multiple-Processor Tracks
 1. Multiprocessor track
 2. Multicomputer track
 2. Multiple data track
 1. Vector track
 2. SIMD track
 3. Multiple threads track
 1. Multithreaded track
 2. Dataflow track

123

Assignment 1

Make a study on different architectural development tracks

124

26-08-2020

125

CONDITIONS of PARALLELISM

- 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 relation between statements
 - 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.

126

- Program segments cannot be executed in parallel unless they are independent.
- Independence comes in several forms
 - I. Data dependence
 - II. Control dependence
 - III. Resource Dependence

127



I. Data Dependence

- The ordering relationship between statements is indicated by the data dependence.
- Five types of data dependence are defined below:
 - a) Flow dependence
 - b) Anti-dependence
 - c) Output dependence
 - d) I/O dependence
 - e) Unknown dependence

128

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 of S1 feeds in as input to S2. Flow dependence is
 - denoted as $S1 \rightarrow S2$
 - Example
 - S1: LOAD R1,A
 - S2: ADD R2,R1
 - S2 is flow dependent on S1
 - Output of S1 fed as input of S2
 - ie A is passed to R1

129

b) Anti-dependence

- Statement S2 is antidependent on statement S1
 - if S2 follows S1 in program order and
 - if the output of S2 overlaps the input to S1.
 - A direct arrow crossed with a bar denotes antidependence from S1 to S2
 - $S1 \nrightarrow S2$
 - Example
 - S2: ADD R2,R1
 - S3: MOV R1,R3
 - S3 is antidependent on S2
 - S3 is overlapping the input to S2

130

c) Output dependence

- Two statements are output-dependent if they produce the same output variable.
- $S1 \rightarrow S2$
- Example
 - S1: LOAD R1, A
 - S3: MOV R1, R3
 - S3 is output-dependent on S1
 - Both modify same register R1

131

d) I/O dependence

- Read and write are IO statements.
- IO dependence occurs not because the same variable is involved but because the same file is referenced by both IO statements.
 - S1: READ(4), A(i)
 - S2: PROCESS
 - S3: WRITE(4), B(i)
 - S4: CLOSE(4)
- S1 and S3 are I/O dependent
 - as they access same file

132

e) Unknown dependence

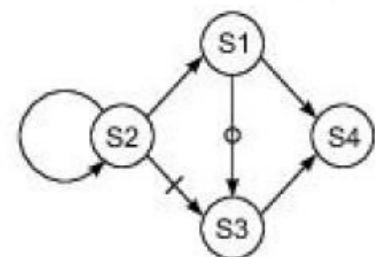
- The dependence relation between two statements cannot be determined in the following situations:
 - The subscript of a variable is itself subscribed (eg: $a(I(J))$)
 - The subscript does not contain the loop index variable. (eg: $a[]$)
 - A variable appears more than once with subscripts having different coefficients of the loop variable.
 - The subscript is non linear in the loop index variable.
- When one or more of these conditions exist, a conservative assumption is to claim unknown dependence among the statements involved.

133

Example

- Consider the following code
- S2 is flow dependent on S1
 - because the variable A is passed via the register R1
- S3 is anti-dependent on S2
 - because of potential conflicts in register content in R1
- S3 is output-dependent on S1
 - because they both modify the same register R1
- S2 and S4 are totally independent

S1:	Load R1, A	$/R1 \leftarrow \text{Memory}(A)/$
S2:	Add R2, R1	$/R2 \leftarrow (R1) + (R2)/$
S3:	Move R1, R3	$/R1 \leftarrow (R3)/$
S4:	Store B, R1	$/\text{Memory}(B) \leftarrow (R1)/$



(a) Dependence graph

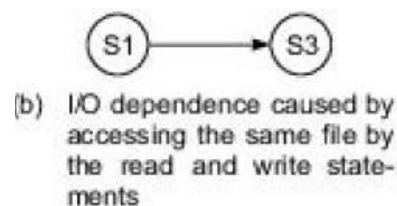
134

Example 2

- Consider following code

S1:	Read (4), A(I)	/Read array A from file 4/
S2:	Process	/Process data/
S3:	Write (4), B(I)	/Write array B into file 4/
S4:	Close (4)	/Close file 4/

- S1 and S3 are I/O dependent on each other
 - because they both access the same file



135

II. Control Dependence

- Implies the order of execution of statements cannot be determined before run time
- For example,
 - conditional statements will not be resolved until run time.
- Different paths taken after a conditional branch may introduce or eliminate data dependence among instructions.
- Dependence may also exist between operations performed in successive iterations of a looping procedure.

136

Example

```
if (a < 0)
  b = 1;
  c = 2;
```

- While the assignment $b = 1$ is executed only if $a < 0$, the assignment $c = 2$ is always executed regardless of the value of a .
- We say that $b = 1$ is *control dependent* on the condition $a < 0$ and that $c = 2$ is *control independent*.
- We refer to the branch on which an instruction is control dependent as its *control dependence branch*.

137

- Control dependence also avoids parallelism to being exploited.
- Compiler techniques or hardware branch *prediction techniques* are needed to get around the control dependence in order to exploit more parallelism.

138

Resource dependence

- Data and control dependencies are based on the **independence** of the work to be done.
- Even if several segments are independent in other ways, they cannot be executed in parallel if there aren't sufficient processing resources.
- Resource dependence is concerned with **conflicts** in using **shared resources** among parallel events
- ALU conflicts are called **ALU dependence**
- Memory conflicts are called **storage dependence**.

139

Bernstein's Conditions

- A set of conditions which must exist if two processes can execute in parallel
- Notation
 - Let P1 and P2 be two processes.
 - Input set I_i is the set of all input variables for a process P_i .
 - I_i is also called the **read set** or **domain of P_i** .
 - We define the input set I_i of a process P_i as the set of all input variables needed to execute the process.
 - Output set O_i is the set of all output variables generated after execution for a process P_i .
 - O_i is also called **write set**.

140

- Input variables are essentially operands which can be fetched from the memory or registers and output variables are the results to be stored in working registers or memory locations.
- If **P1 and P2 can execute in parallel** (which is written as $P1 \parallel P2$), then:

$$I_1 \cap O_2 = \emptyset$$

$$I_2 \cap O_1 = \emptyset$$

$$O_1 \cap O_2 = \emptyset$$

141

Bernstein's condition -2

- In terms of data dependencies, Bernstein's conditions imply that two processes can execute in parallel
 - if they **are flow-independent**,
 - **ant independent**, and
 - **output-independent**.
- a set of processes $P1, P2, \dots, Pk$, can execute in parallel
 - if Bernstein's conditions are satisfied on a **pairwise basis**.
 - That is $P1 \parallel P2 \parallel P3 \dots \parallel Pk$ if and only if $Pi \parallel Pj$ for all $i \neq j$.

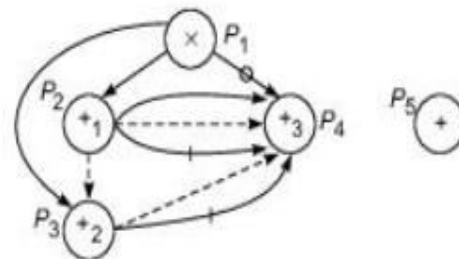
142

- The parallelism relation \parallel is **commutative** ie
 - $(P_i \parallel P_j \text{ implies } P_j \parallel P_i)$,
 - but not transitive ($P_i \parallel P_j$ and $P_j \parallel P_k$ does not guarantee $P_i \parallel P_k$).
 - Therefore, \parallel is not an equivalence relation.
 - $P_i \parallel P_j \parallel P_k$ implies associativity. ie $(P_i \parallel P_j) \parallel P_k = P_i \parallel (P_j \parallel P_k)$
- Since the order in which parallel executable processes are executed should not make any difference in the output sets.

143

Detection of parallelism in a program using Bernstein's conditions

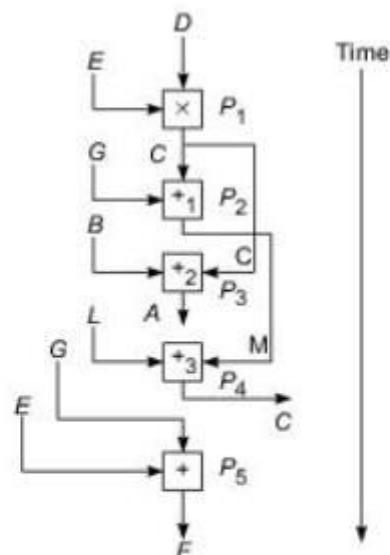
- Consider the simple case in which each process is a single High Level Language(HLL) statement.
- We want to detect the parallelism embedded in the following five statements labeled P_1 , P_2 , P_3 , P_4 , and P_5 , in program order.

$$\left. \begin{array}{l} P_1: C = D \times E \\ P_2: M = G + C \\ P_3: A = B + C \\ P_4: C = L + M \\ P_5: F = G + E \end{array} \right\}$$


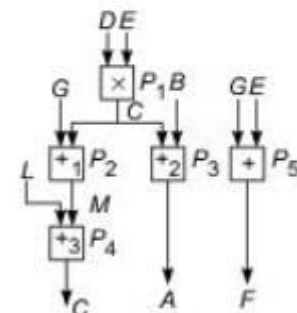
(a) A dependence graph showing both data dependence (solid arrows) and resource dependence (dashed arrows)

- Assume that each statement requires one step to execute. No pipelining is considered here.
- The dependence graph demonstrate flow dependence as well as resource dependence
- In sequential execution five steps are needed

145



(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

146

- If two adders are available simultaneously,
 - the parallel execution requires only three steps as shown in Fig. 2.2c.
 - Pairwise, there are 10 pairs of statements to check against Bernstein's conditions.
 - Only 5 pairs,
 - $P1||P5$, $P2||P3$, $P2||P5$, $P5||P3$, and $P4||P5$, can execute in parallel if there are no resource conflicts.
 - only $P2||P3||P5$, is possible because $P2||P3$, $P3||P5$, and $P5||P2$ are all possible.
- Violations of any one or more of the three conditions prohibits parallelism between two processes.
- data dependence, control dependence, and resource dependence all prevent parallelism from being exploitable.

147

Hardware Parallelism

e.g **Number of processors**

- Defined by machine architecture, hardware multiplicity (number of processors available) and connectivity.
- **Displays Resource Utilization Patterns** of simultaneously executable operations indicate **Peak performance of processor** resources.
- Often a function of **cost/performance tradeoffs**.
- Characterized in a single processor by the number of instructions k issued in a single cycle (k -issue processor).
- A multiprocessor system with n k -issue processor can handle a maximum limit of nk parallel instructions (at ILP level) or n parallel threads at thread-level parallelism (TLP) level.

148

Software Parallelism

e.g **Degree of Parallelism (DOP)**

- Defined by the control and data dependence of programs.
- Revealed in program profiling or program dependency (data flow) graph.
- A function of algorithm, parallel task grain size, programming style and compiler optimization.
- Program Flow Graphs - Patterns of simultaneously executable operations
- Parallelism in a program varies during execution period
- Types
 - Control Parallelism and
 - Data Parallelism

149

Hardware and Software Parallelism

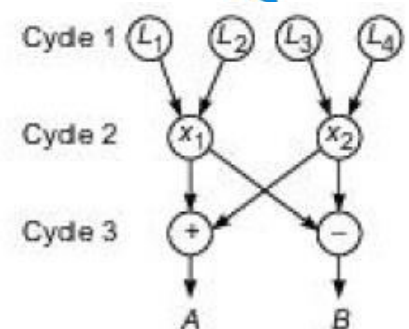
- For implementation of parallelism special hardware and software support is needed
- Compilation support is also needed to close the gap between hardware and software

150

Hardware Parallelism	Software Parallelism
Its build into machines architecture and hardware multiplicity. Also known as machine parallelism	Its exploited by the concurrent execution of machine language instructions in a program
It's a function of cost and performance trade off	It's a function of algorithm, programming style and compiler optimization.
It displays resource utilization patterns of simultaneously executable operations. It also indicates the peak performance of processor resources	It displays patterns of simultaneously executable operations
Its characterized by no: of instruction issues per machine cycle	<p>The program flow graph displays the patterns of simultaneously executable operations</p> <p>2 types –</p> <p>Control parallelism – allows 2 or more operations to be performed simultaneously.</p> <p>Data parallelism – atleast same operation is performed over many data elements by many processors simultaneously</p>

Mismatch between software parallelism and hardware parallelism

- 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.



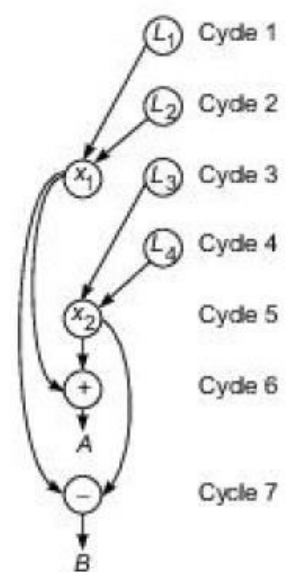
L_i : Load operation
 X_j : Multiply operation

(a) Software parallelism

- 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**

153

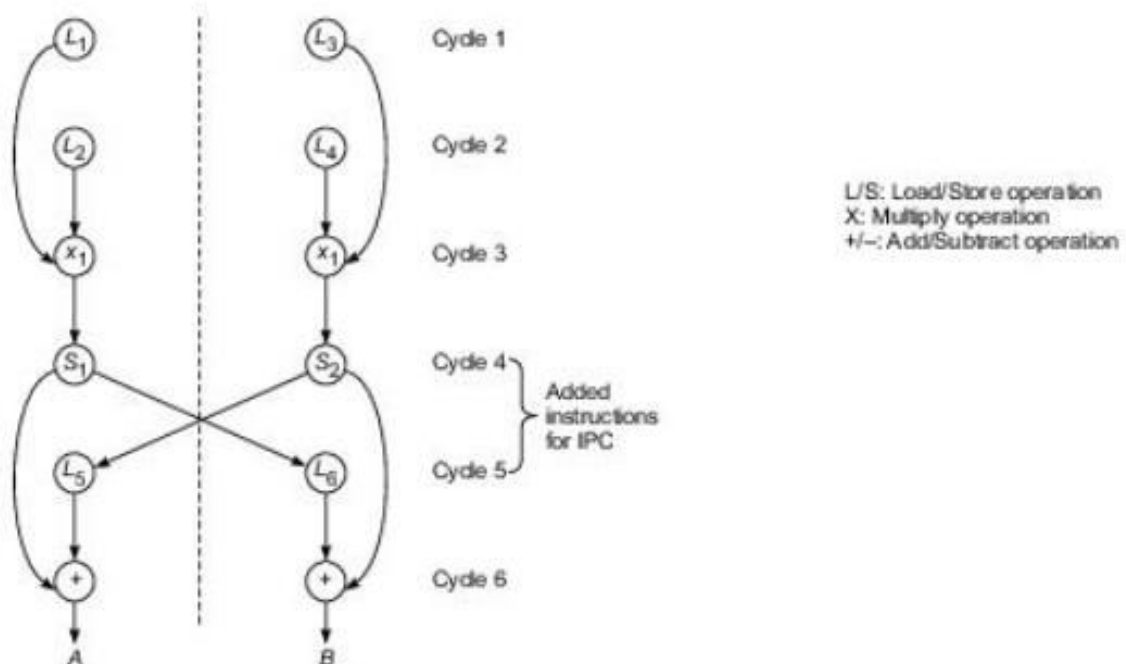
- With this hardware restriction, the program must execute in seven machine cycles
- Therefore, the hardware parallelism displays an average value of $8/7 = 1.14$ instructions executed per cycle.
- This demonstrates a mismatch between the software parallelism and the hardware parallelism.



(b) Hardware parallelism

- Let us try to match the software parallelism in a hardware platform of a dual processor system,
- where single-issue processors are used.
- L/S stands for load/store operations.
- six processor cycles are needed to execute the I2 instructions by two processors. .
- S1 and S2 are two inserted store operations, and
- I5 and I6 are two inserted load operations.
- These added instructions are needed for interprocessor communication through the shared memory.

155



Types of Software Parallelism

1. control parallelism

- which allows two or more operations to be performed simultaneously
- appearing in the form of pipelining or multiple functional units, is limited by the pipeline length and by the multiplicity of functional units.
- Both pipelining and functional parallelism are handled by the hardware; programmers need take no special actions to invoke them.

157

2. data parallelism

- in which almost the same operation is performed over many data elements by many processors simultaneously.
- offers the highest potential for concurrency.
- It is practiced in both SIMD and MIMD modes on MPP systems.
- Data parallel code is easier to write and to debug than control parallel code.
- Synchronization in SIMD data parallelism is handled by the hardware.
- Data parallelism exploits parallelism in proportion to the quantity of data involved.

158

Solving Mismatch Problem

- To solve the mismatch problem between software parallelism and hardware parallelism,
 - one approach is to develop compilation support, and the
 - other is through hardware redesign for more efficient exploitation of parallelism.
- These two approaches must cooperate with each other to produce the best result.

159

ROLE OF COMPILERS

- Hardware processors can be better designed to exploit parallelism by an optimizing compiler
- That is compiler techniques are used to exploit hardware features to improve performance.
- Such processors use large register file and sustained instruction pipelining to execute nearly one instruction per cycle.
- The large register file supports fast access to temporary values generated by an optimizing compiler.
- The registers are exploited by code optimizer and global register allocator in such a compiler.

160

Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

$$\text{Fraction}_{\text{enhanced}} = 0.4, \text{Speedup}_{\text{enhanced}} = 10, \text{Speedup}_{\text{overall}} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} \approx 1.56$$

161

A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design alternatives.

We can compare these two alternatives by comparing the speedups:

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1 - 0.2) + \frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1 - 0.5) + \frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

162