FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Advanced Computer Architecture

17CS72

MODULE-1

Theory of Parallelism

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

https://hemanthrajhemu.github.io

MODULE-1

Syllabus:

MODULE	Theory of Parallelism	
1	Parallel Computer Models	1
	The State of Computing, Multiprocessors and Multicomputer	1
	Multivector and SIMD Computers	1
	PRAM and VLSI Models, Program and Network Properties	1
	Conditions of Parallelism	1
	Program Partitioning and Scheduling	1
	Program Flow Mechanisms, System Interconnect Architectures	1
	Principles of Scalable Performance	1
	Performance Metrics and Measures, Parallel Processing Applications	1
	Speedup Performance Laws, Scalability Analysis and Approaches	1

https://hemanthrajhemu.github.io

Theory of Parallelism

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Computer Development Milestones

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

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Computer Generations

- **First Generation (1945 – 54)**
  - **Technology & Architecture:**
    - Vacuum Tubes
    - Relay Memories (Switch kind latch)
    - 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

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Computer Generations

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

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Computer Generations

- **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

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Computer Generations

- **Fourth Generation (1975 – 90)**
  - o **Technology & Architecture:**
    - LSI/VLSI
    - Semiconductor memories
    - Multiprocessors
    - Multi-computers
    - Vector supercomputers
  - o **Software and Applications:**
    - Multiprocessor OS
    - Languages, Compilers and environment for parallel processing
  - o **Representative Systems:** VAX 9000, Cray X-MP, IBM 3090, BBN TC 2000

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Computer Generations

- **Fifth Generation (1991 onwards)**
  - **Technology & Architecture:**
    - Advanced VLSI Processors
    - Memory and Switches
    - Scalable Architectures
    - High Density Packaging
  - **Software and Applications:**
    - Superscalar Processors
    - 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

- The hardware, software, and programming elements of modern computer systems can be characterized by looking at a variety of factors, including:
  - Computing problems
  - Algorithms and data structures
  - Hardware resources
  - Operating systems
  - System software support
  - Compiler support

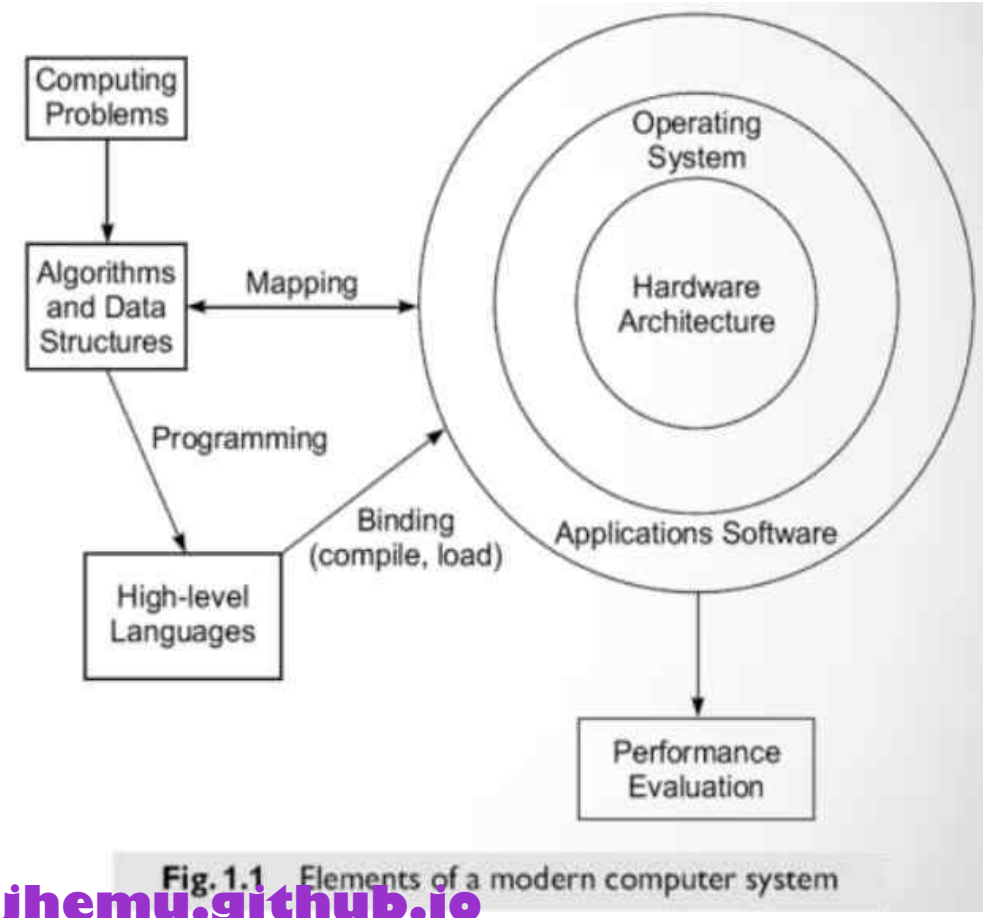


Fig. 1.1 Elements of a modern computer system

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Elements of Modern Computers

Computing problems

- A modern computer is an integrated system consisting of :
  - Machine hardware
  - An instruction set
  - System software
  - Application programs
  - User interfaces.
- The use of a computer is driven by real-life problems demanding cost effective solutions.
- For numerical problems in science and technology, the solutions demand complex mathematical formulations and intensive integer or floating-point computations.
- For artificial intelligence [Al] problems, the solutions demand logic inferences and symbolic manipulations.

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Elements of Modern Computers

Algorithms and Data Structures

- Traditional algorithms and data structures are designed
  - For sequential machines.
- New, specialized algorithms and data structures are needed
  - To exploit the capabilities of parallel architectures.
- These often require interdisciplinary interactions
  - Among theoreticians, experimentalists and programmers.

THE STATE OF COMPUTING: Elements of Modern Computers

Hardware Resources

- Processors, memory, and peripheral devices form the hardware core of a computer system.
- A modem computer system demonstrates its power through coordinated efforts by :
  - Hardware recourses
  - An operating system
  - Application software.
- Special hardware interfaces are often built into I-O devices such as display terminals, workstations.
- In addition, software interface programs are needed. These software interfaces include :
  - File transfer systems, editors, word processors, device drivers, interrupt handlers, network communication programs, etc.

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Elements of Modern Computers

Operating System

- Operating systems manage the allocation and deallocation of resources during user program execution.
- An OS plays a significant role in mapping hardware resources to algorithmic and data structures.
  - Implementation of Mapping:
    - Relies on efficient Compiler and Operating system support.
- Parallelism can be exploited at Algorithm :
  - Design Time
  - Program Time
  - Compile Time
  - Run Time

THE STATE OF COMPUTING: Elements of Modern Computers

System Software Support

- System software is needed for the development of efficient programs in HLL.
- Few system software's are: Compliers, Loaders, Linkers, OS Kernel
- Parallel software can be developed:
  - Using entirely new languages designed specifically with parallel support
  - Using extensions to existing sequential languages.
- New languages have obvious advantages:
  - Like new constructs specifically for parallelism
  - But require additional programmer education and system software.
- The most common approach is to extend an existing language.

THE STATE OF COMPUTING: Elements of Modern Computers

Complier Support

There Complier upgrade approaches:

1. Pre-processors
   - Use existing sequential compilers and specialized libraries to implement parallel constructs
2. Pre-compilers
   - Perform some program flow analysis, dependence checking, and limited parallel optimizations
3. Parallelizing Compilers
   - Requires full detection of parallelism in source code, and transformation of sequential code into parallel constructs

- Compiler directives are often inserted into source code to aid compiler parallelizing efforts

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Evolution of Computer Architecture

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

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Evolution of Computer Architecture

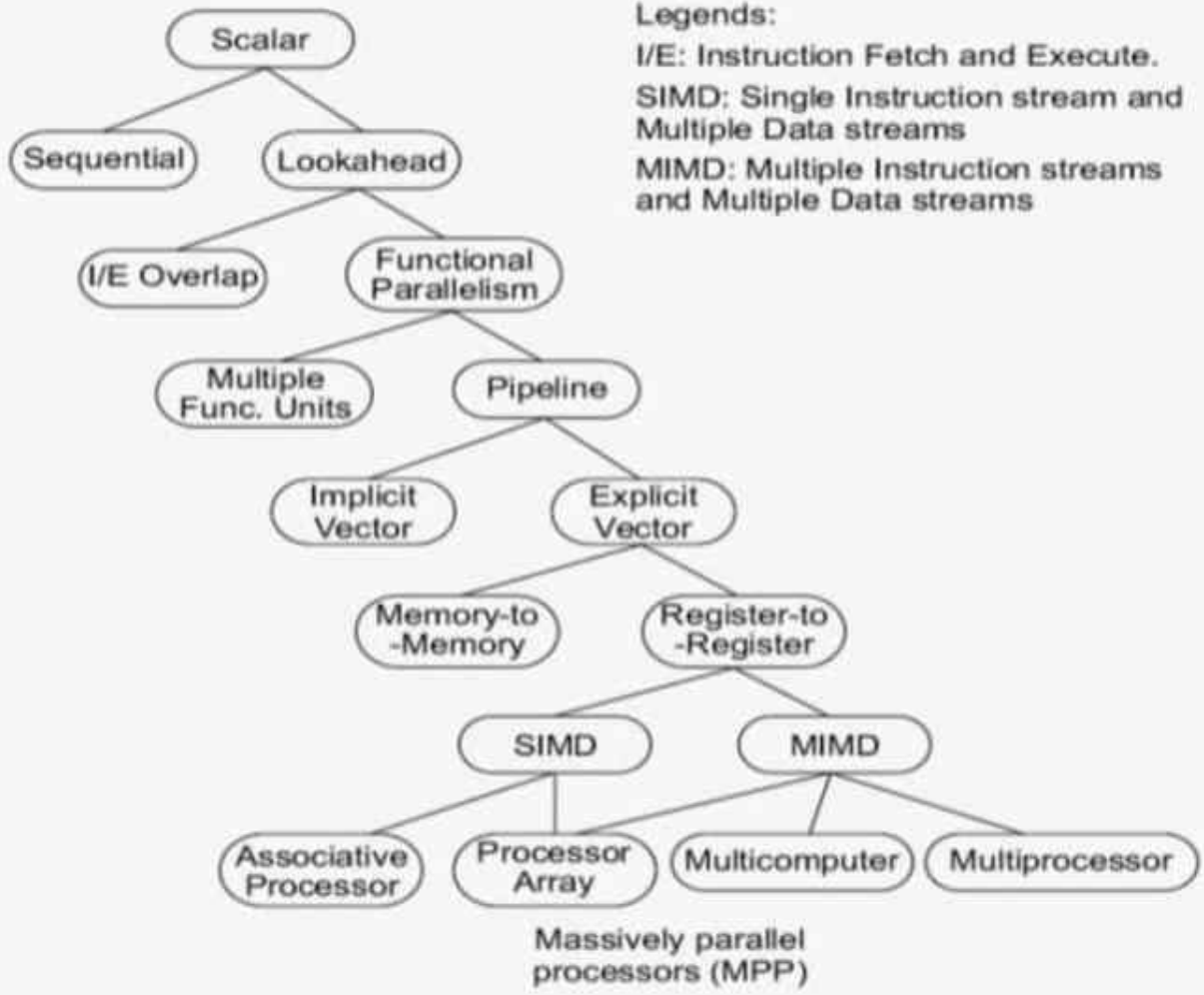


Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel [illegible]

THE STATE OF COMPUTING: Evolution of Computer Architecture

Look-ahead, parallelism and pipelining

- Look-ahead
  - Introduced to pre-fetch Instruction to overlap Instruction Fetch & Execute
  - They enable function parallelism
- Parallelism
  - Supported by two approaches:
    - Use of multiple functional units simultaneously.
    - Practicing pipelining at various processing levels➔ Includes
      - Pipelined Instruction execution
      - Pipelined arithmetic computations
      - Memory access operations
- Pipelining
  - Performs identical operations repeatedly on vector data strings
  - Uses Scalar pipeline processors

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Evolution of Computer Architecture

Flynn's classification

- This taxonomy distinguishes multi-processor computer architectures according to two independent dimensions :
- Instruction stream
  - Sequence of instructions executed by machine
- Data stream.
  - Sequence of data including input, partial or temporary results used by instruction stream
- Each of these dimensions can have only one of two possible states:
  - Single or Multiple
- Flynn's classification depends on the distinction between :
  - The performance of control unit
  - The data processing unit rather than its operational and structural interconnections.
- Following are the four category of Flynn classification:
  - SISD(**Single instruction stream, single data stream**)
  - SIMD(**Single instruction stream, multiple data stream**)
  - MIMD(**Multiple instruction stream, multiple data stream**)
  - MISD(**Multiple instruction stream, single data stream**)

THE STATE OF COMPUTING: Evolution of Computer Architecture

Flynn's classification → SISD(Single instruction stream, single data stream)

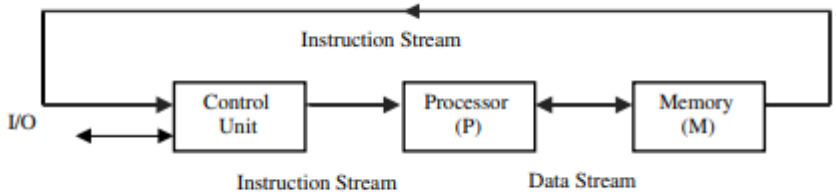
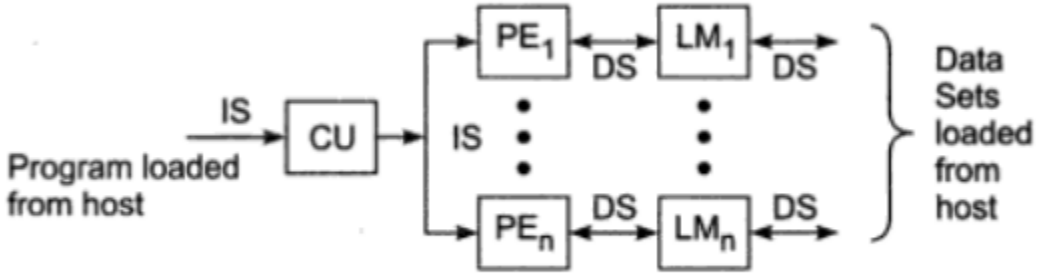


Figure 1.1 SISD architecture.

Flynn's classification → SIMD(Single instruction stream, multiple data stream)



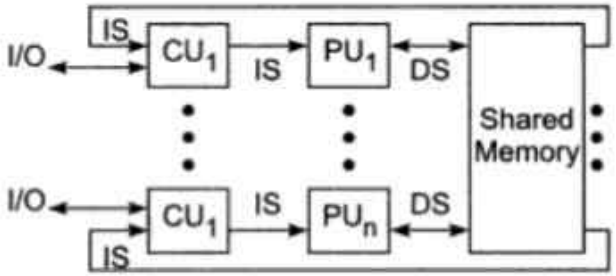
(b) SIMD architecture (with distributed memory)

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

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Evolution of Computer Architecture

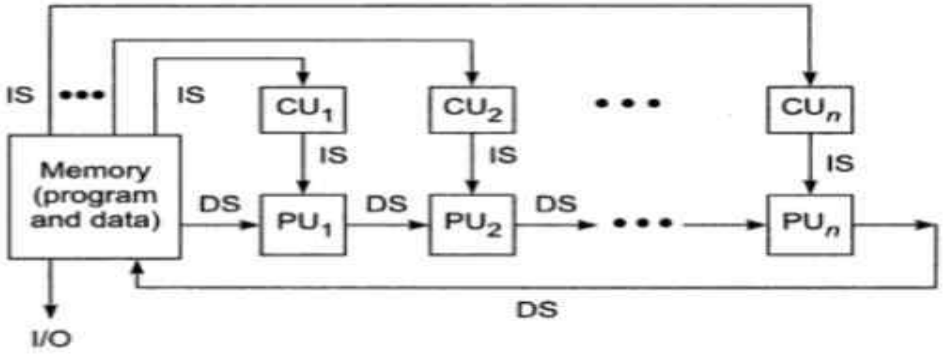
Flynn's classification → MIMD(Multiple instruction stream, multiple data stream)



(c) MIMD architecture (with shared memory)

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 → MISD(Multiple instruction stream, single data stream)



(d) MISD architecture (the systolic array)

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Evolution of Computer Architecture

Parallel/Vector Computers

Parallel computers are those that execute programs in➔ MIMD mode.

- There are two major classes of parallel computers
  - Shared-memory multiprocessors
  - message-passing multicomputers.
- The major distinction between multiprocessors and multicomputers lies :
  - Memory sharing
    - Multiprocessors: Processors communicate through shared variable in a common memory
    - Multicomputer: Each computer has a local memory unshared with other computers.
  - Inter-processor communication
    - Through message passing for a multi computer

Vector Processors:

- Equipped with multiple vector pipelines that can be concurrently used under hardware or firmware control
- Two families :
  - *Memory-to-memory* architecture
  - *Register-to-register* architecture

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: Evolution of Computer Architecture

Development Layers

Six layers of computer system development:

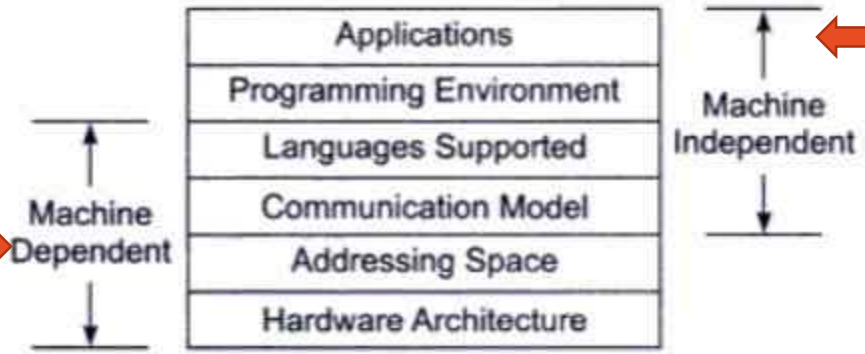


Fig. 1.4 Six layers for computer system development (Courtesy of Lionel Ni, 1990)

Programmers View Point→ these 2 layers should be architecture transparent

This features are up to the designer → should match the target application domain

As a good computer architect ➔One has to approach problem from both the ends

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: System Attributes Affecting Performance

- The ideal performance of a computer system demands a perfect match between machine capability and program behavior.
- Performance depends on:
  - Hardware technology
  - Architectural features
  - Efficient resource management
  - Algorithm design
  - Data structures
  - Language efficiency
  - Programmer skill
  - Compiler technology

THE STATE OF COMPUTING: System Attributes Affecting Performance

Performance Indicators:

- The simplest measure of program performance is the turnaround time , which includes :
  - Disk and memory accesses
  - Input and output
  - Compilation time
  - Operating system overhead
  - CPU time
- Since I/O and system overhead frequently ➔overlaps processing by other programs, it is fair to consider only :➔
  - The CPU time used by a program.
  - The user CPU time is the most important factor.

THE STATE OF COMPUTING: System Attributes Affecting Performance

Clock Rate and CPI:

- CPU is driven by a clock with a constant cycle time τ (usually measured in nanoseconds).
- The inverse of the cycle time is the clock rate ($f = 1/\tau$, measured in megahertz).
- The size of a program is determined by its instruction count, Ic , the number of machine instructions to be executed by the program.
- Different machine instructions require different numbers of clock cycles to execute. CPI (cycles per instruction) is thus an important parameter.

Average CPI:

- It is easy to determine the average number of cycles per instruction for a particular processor if we know the frequency of occurrence of each instruction type.
- Of course, any estimate is valid only for a specific set of programs (which defines the instruction mix), and then only if there are sufficiently large number of instructions.
- In general, the term CPI is used with respect to a particular instruction set and a given program mix.

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: System Attributes Affecting Performance

Performance Factors (1):

- The time required to execute a program containing Ic instructions is just

$$T = Ic * CPI * \tau.$$

- Each instruction must be ➔ fetched from memory → decoded → then operands fetched from memory → the instruction executed → the results stored.
- The time required to access memory is called the memory cycle time,
  - **Which is usually k times the processor cycle time t.**
  - **The value of k depends on the memory technology and the processor-memory interconnection scheme.**

Performance Factors (2):

- The processor cycles required for each instruction (CPI)➔is the total of
  - cycles needed for instruction decode & execution **(p)**
  - cycles needed for memory references **(m * k)**
- The total time needed to execute a program can then be rewritten as

T = Ic * (p + m * k) * τ

THE STATE OF COMPUTING: System Attributes Affecting Performance

System Attributes:

The five performance factors (Ic , p, m, k, τ) are influenced by four system attributes:

- **Instruction-set architecture (affects Ic and p)**
- **Compiler technology (affects Ic and p and m)**
- **CPU implementation and control (affects p × τ)**
- **Cache and memory hierarchy (affects memory access latency, k × τ)**

Total CPU time can be used as a basis in estimating the execution rate of a processor.

THE STATE OF COMPUTING: System Attributes Affecting Performance

System Attributes:

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

https://hemanthrajhemu.github.io

MIPS Rate:

If C is the total number of clock cycles needed to execute a given program =>

- Then total CPU time can be estimated as:
  - $T = C \times \tau = C / f.$
- Other relationships are easily observed:
  - $CPI = C / Ic$
  - $T = Ic \times CPI \times \tau$
  - $T = Ic \times CPI / f$

Processor speed is often measured in terms of millions of instructions per second, frequently called the MIPS rate of the processor

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: System Attributes Affecting Performance

MIPS Rate:

- Then total CPU time can be estimated as:
  - $T = C \times \tau = C / f.$
- Other relationships are easily observed:
  - $CPI = C / Ic$

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

The MIPS rate is:

- Directly proportional ==> To the clock rate
- Inversely proportion to the CPI.

Note:

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

https://hemanthrajhemu.github.io

Throughput Rate:

The number of programs a system can execute per unit time, Ws, in programs per second.

CPU throughput, Wp, is defined as :

$$Wp = f/(Ic * CPI)$$

In a multiprogrammed system, the system throughput is often less than the CPU throughput.

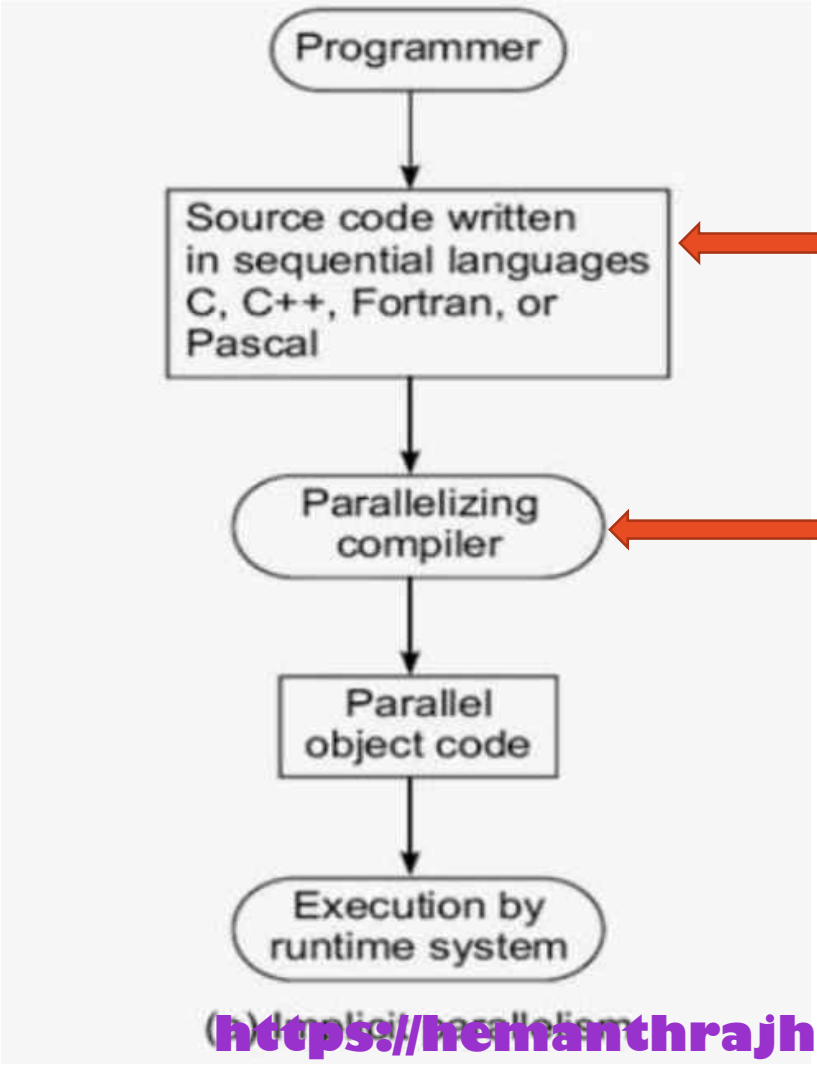
Programming Enviroments:

- **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**
  - **Operating systems that can perform parallel scheduling of:**
    - **Concurrent events**
    - **Shared memory allocation**
    - **Shared peripheral**
    - **Communication links.**

https://hemanthrajhemu.github.io

THE STATE OF COMPUTING: System Attributes Affecting Performance

Approach of Parallelism

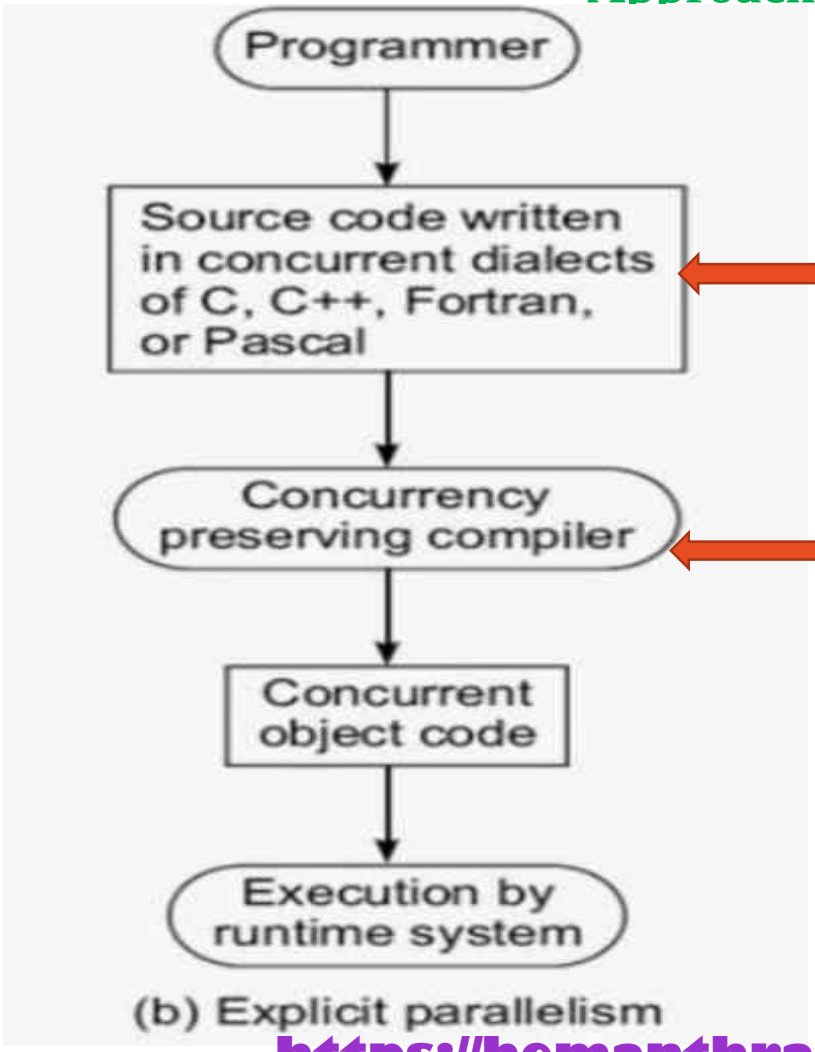


Source Program ➔ Sequential coded

- Complier has to detect parallelism ➔assign machine resources
- Used in shared memory multiprocessors
- Success relies➔ on the intelligence of the parallelizing complier

THE STATE OF COMPUTING: System Attributes Affecting Performance

Approach of Parallelism



(b) Explicit parallelism

Source Program ➔ Parallelism is explicitly specified in the user program

- Complier needs to preserve Parallelism ➔assign machine resources

https://hemanthrajhemu.github.io

Multiprocessors and Multicomputers

Categories of Parallel Computers:

- **Considering their architecture only, there are two main categories of parallel computers:**
  - **Systems with shared common memories(Shared Memory Multiprocessors)**
    - **The UMA Model**
    - **The NUMA Model**
    - **The COMA Model**
    - **The CC-NUMA Model**
  - **Systems with unshared distributed memories(Distributed-Memory Multicomputers)**
    - **The NORMA Machines**
    - **Message Passing multicomputers**
  - **Taxonomy of MIMD Computers**
  - **Representative Systems**
    - **Multiprocessors: BBN TC-200, MPP, S-81, IBM ES/9000 Model 900/VF,**
    - **Multicomputers: Intel Paragon XP/S, nCUBE/2[illegible] SuperNode 1000, CM5, KSR-1**

Multiprocessors and Multicomputers:

Shared Memory Multiprocessors(Uniform Memory Access Model ➔UMA Model)

- **Physical memory uniformly shared by all processors**
  - **Has equal access time to all words.**
- **Processors may have local cache memories.**
- **Peripherals also shared in some fashion.**
- **Tightly coupled systems use a :**
  - **common bus, crossbar, or multistage network to connect processors, peripherals, and memories.**
- **Many manufacturers have multiprocessor (MP) extensions of uniprocessor (UP) product lines.**

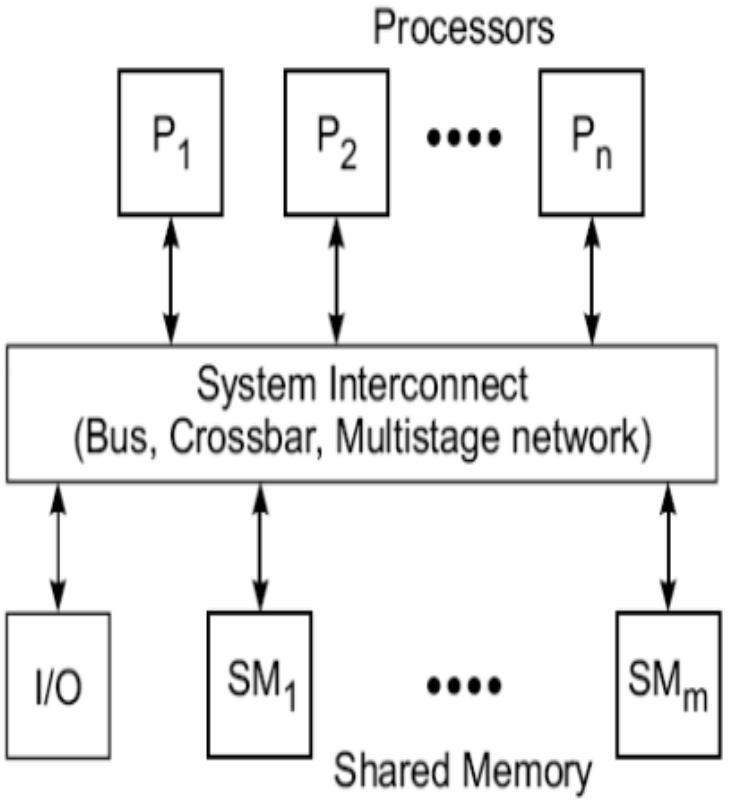


Fig. 1.6 The UMA multiprocessor model

https://hemanthrajhemu.github.io

Multiprocessors and Multicomputers:

Shared Memory Multiprocessors(UMA Model)

- **Synchronization and communication among processors achieved through shared variables in common memory.**

- **Two types of Multiprocessors:**

- **Symmetric MP systems :**
  - **All processors have access to all peripherals**
  - **Any processor can run the OS and I/O device drivers.**
- **Asymmetric MP systems :**
  - **Not all peripherals accessible by all processors**
  - **Kernel runs only on selected processors (master)**
  - **Others are called attached processors (AP)**

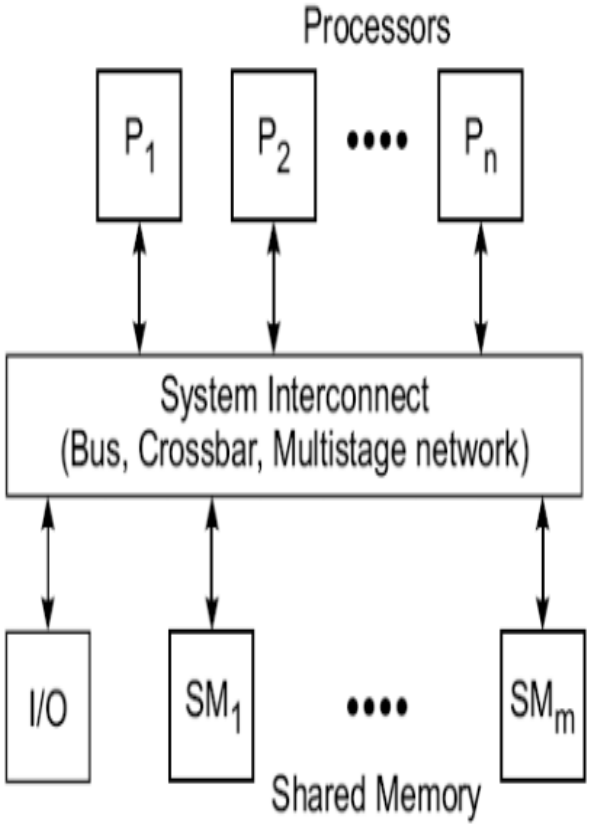


Fig. 1.6 The UMA multiprocessor model

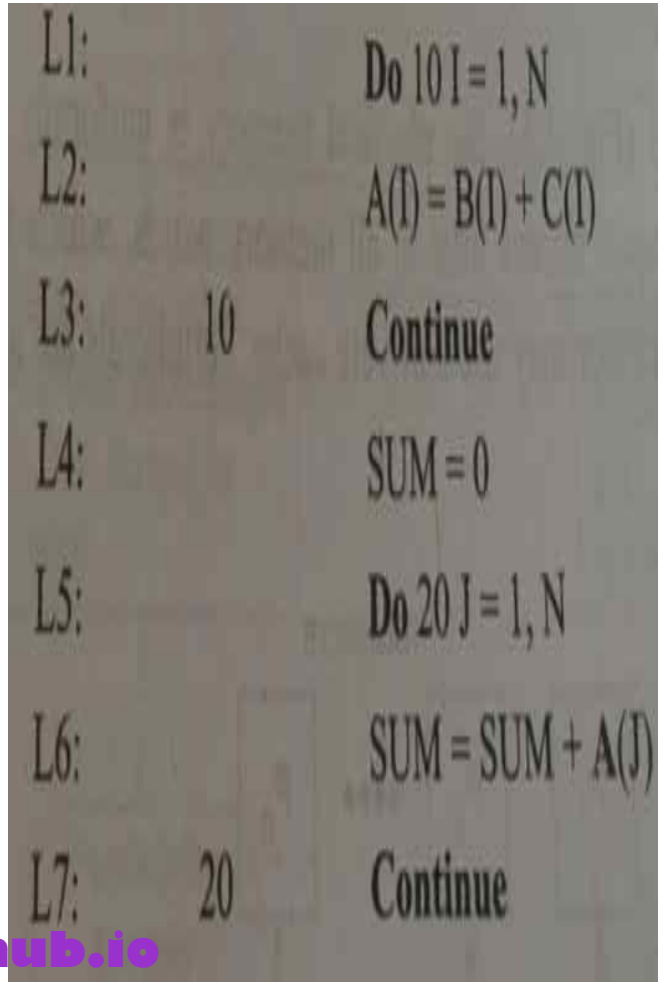
Multiprocessors and Multicomputers:

Shared Memory Multiprocessors(UMA Model)

Example: Performance Calculation:

Consider two loops:(Sequential Processor)

- **The first loop ➔ adds corresponding elements of two N-element vectors to yield a third vector.**
- **The second loop ➔ sums elements of the third vector.**
- **Assume each add/assign operation**
  - **Takes 1 cycle,**
  - **And ignore time spent on other actions (e.g. loop counter incrementing/testing, instruction fetch, etc.).**
- **Assume interprocessor communication**
  - **Requires k cycles.**
- **On a sequential system, each loop(I/J) will require N cycles**
  - **Hence a total of 2N cycles of processor time is required**



Multiprocessors and Multicomputers:

Shared Memory Multiprocessors(UMA Model)

Example: Performance Calculation::(Parallel Processor)

- To execute a program ➔On an M-processor system,
- We can partition each loop into M parts,
  - Each having L = N / M (for each Loop section).
- As we have 2 loops ➔The total time required is thus 2L.
  - This leaves us with M partial sums that must be totalled.(Through Recursion Concept)
- Computing the final sum from the M partial sums requires➔
  - To merge all partial sums of l level binary tree takes $l = \log_2(M)$ additions
  - The addition of each pair of partial sum requires k cycles through shared memory.
  - 1 cycle (for the add/assign),
  - Hence Total cycles ➔ l × (k+1) cycles.
- The parallel computation thus requires

$$2N / M + (k + 1) \log2(M) \text{ cycles}$$

https://hemanthrajhemu.github.io

Multiprocessors and Multicomputers:

Shared Memory Multiprocessors(UMA Model)

Example: Performance Calculation:

- Assume $N = 2^{20}$.
- Sequential execution requires $2N = 2^{21}$ cycles.
- If processor synchronization requires $k = 200$ cycles, and we have $M = 256$ processors, parallel execution requires

$$2N / M + (k + 1) \log_2(M)$$

$$= 2^{21} / 2^8 + 201 \times 8$$

$$= 2^{13} + 1608 = 9800 \text{ cycles}$$

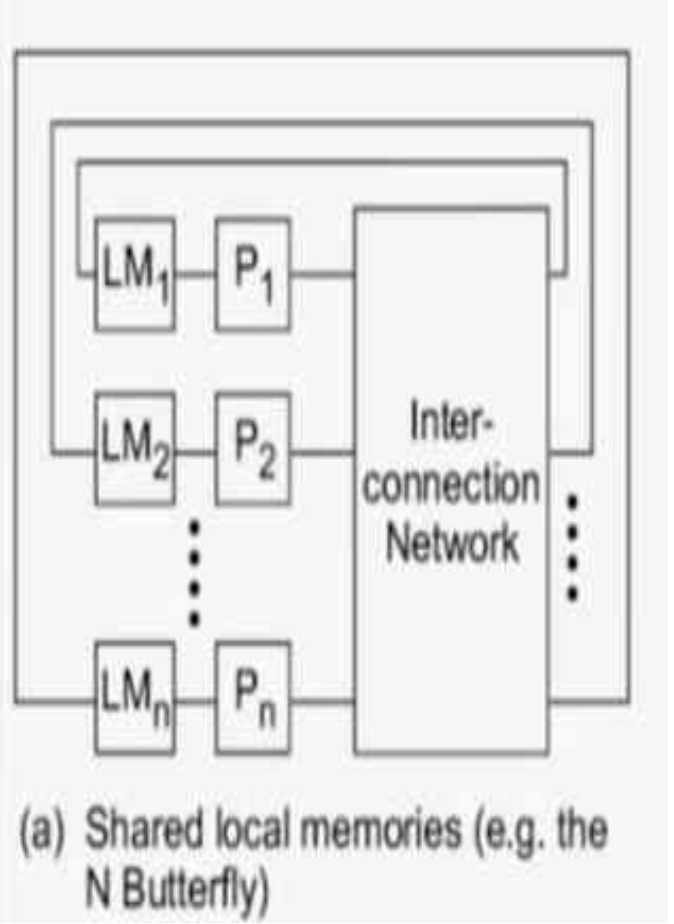
- Comparing results, the parallel solution is 214 times faster than the sequential, with the best theoretical speedup being 256 (since there are 256 processors).
- Thus the efficiency of the parallel solution is 214 / 256 = 83.6 %

https://hemanthrajhemu.github.io

Multiprocessors and Multicomputers:

Shared Memory Multiprocessors(Non-Uniform Memory Access(NUMA) Model)

- Shared memories, but access time depends on the location of the data item.
- The shared memory is distributed among the :
  - As local memories, but each of these is still accessible by all processors (with varying access times).
- Memory access is fastest from the :
  - Locally-connected processor, with the interconnection network adding delays for other processor accesses.
- Additionally, there may be global memory in a multiprocessor system:
  - With two separate interconnection networks, one for clusters of processors and their cluster memories, and another for the global shared memories

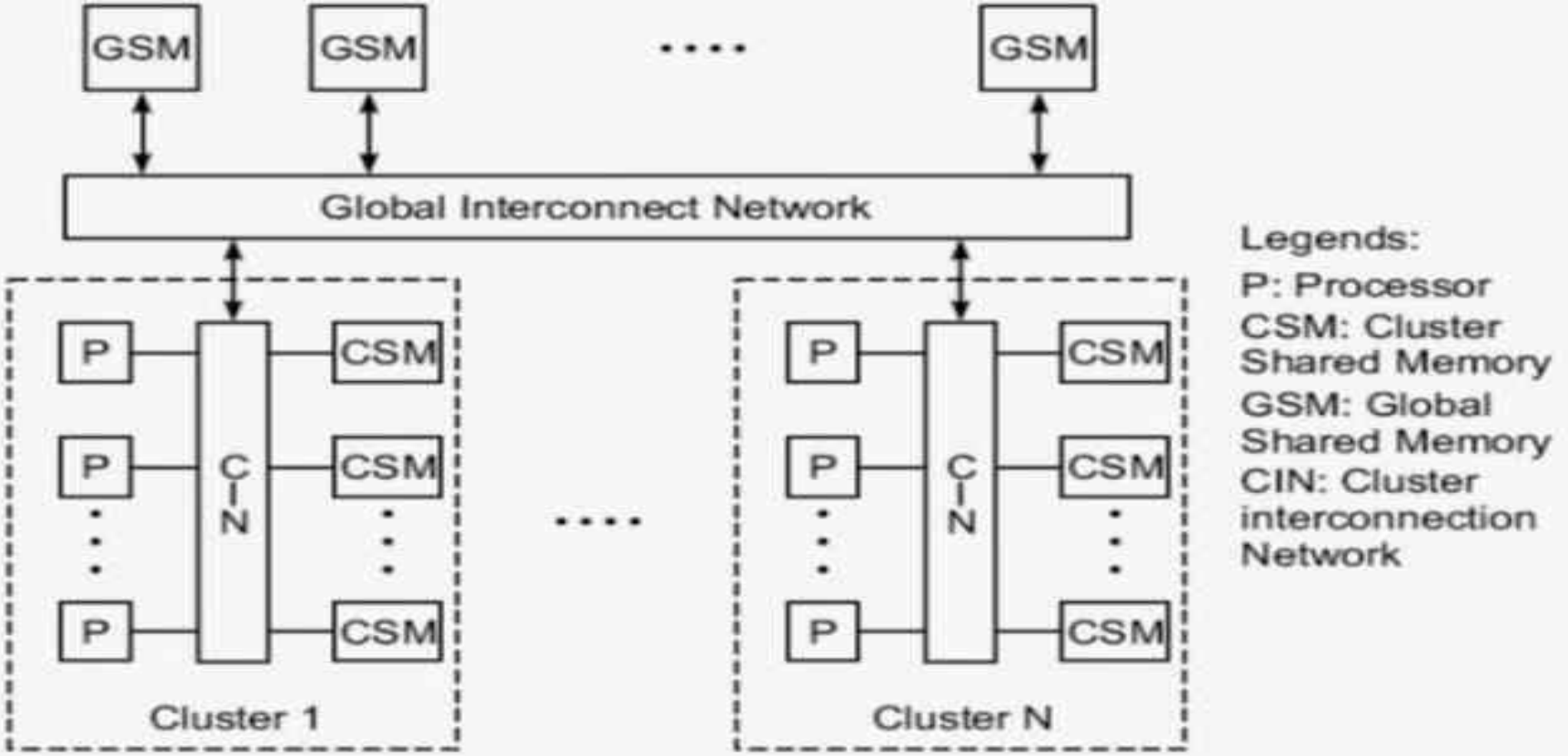


(a) Shared local memories (e.g. the N Butterfly)

https://hemanthrajhemu.github.io

Multiprocessors and Multicomputers:

Shared Memory Multiprocessors(NUMA Model:Hierarchical Cluster Model)



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

https://hemanthrajhemu.github.io

Multiprocessors and Multicomputers:

Shared Memory Multiprocessors(Cache Only Memory Access Model -COMA)

- **In the COMA model,**
  - **Special kind of NUMA model**
  - **Distributed memories of NUMA have converted to cache memories;**
  - **The caches, taken together, form a global address space.**
- **Each cache has an associated directory➔**
  - **Remote machines in their lookups;**
  - **Hierarchical directories may exist in machines based on this model.**
- **Initial data placement is not critical, as cache blocks will eventually migrate to where they are needed**

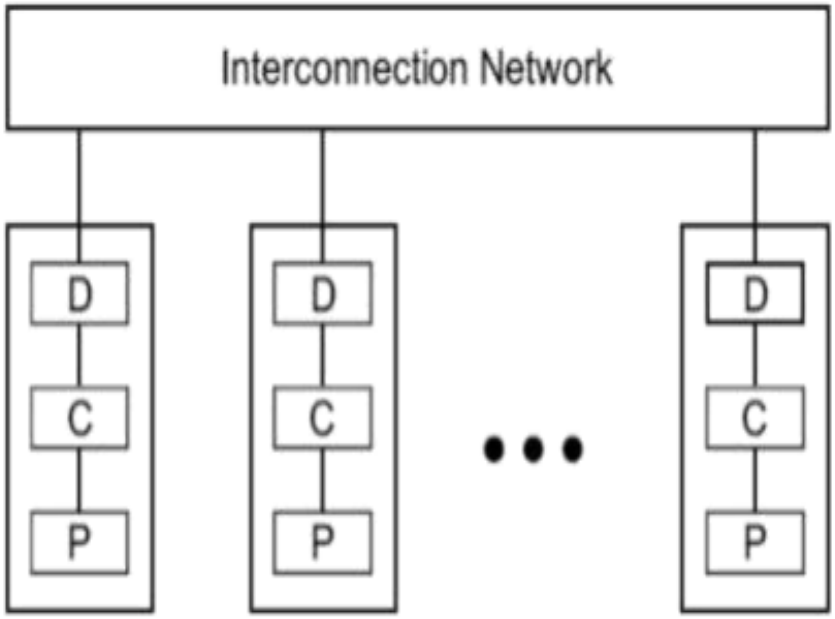


Fig. 1.8 The COMA model of a multiprocessor

(P: Processor, C: Cache, D: Directory)

https://hemanthrajhemu.github.io

Multiprocessors and Multicomputers:

Shared Memory Multiprocessors

(Cache Coherent –Non Uniform Memory Access ➔CC-NUMA)

It uses the concept of both:

1. COMA
2. NUMA

Multiprocessors and Multicomputers:

Distributed Memory Multicomputers

- Multicomputer consist of multiple computers/nodes, interconnected by a message-passing network.
- Each node is autonomous, with its own processor and local memory, and sometimes local peripherals.
- The message-passing network provides point-to-point static connections among the nodes.
- Local memories are not shared,
  - So traditional multicomputer are called no-remote-memory access (or NORMA) machines.
- Inter-node communication is achieved by passing messages through the static connection network.

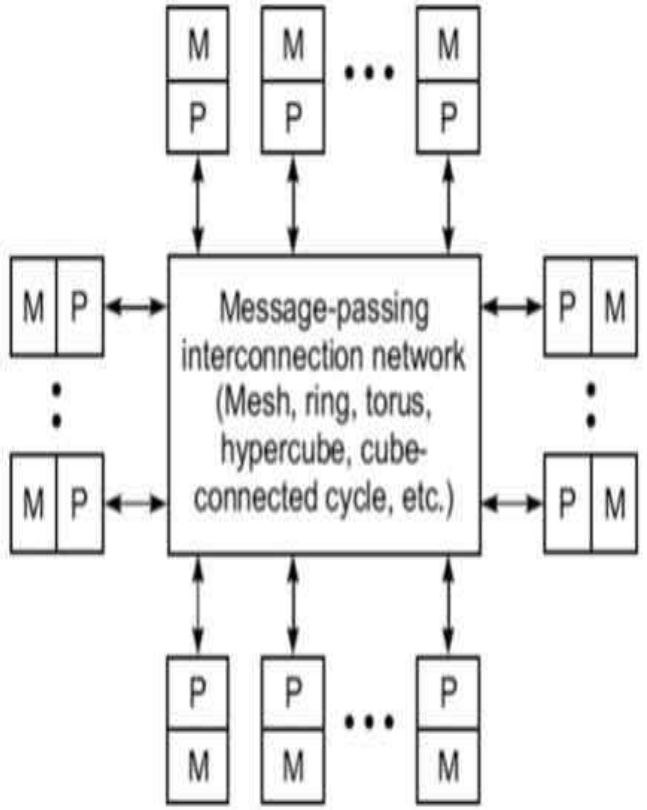


Fig. 1.9 Generic model of a message-passing multicomputer

https://hemanthrajhemu.github.io

Multiprocessors and Multicomputers:

Multicomputer Generations

- **Each multicomputer uses:**
  - Routers and channels in its interconnection network,
  - Heterogeneous systems may involved mixed node types
  - Uniform data representation and communication protocols.
- **First generation:**
  - Hypercube architecture
  - Software controlled message switching
  - Processor boards
- **Second generation:**
  - Mesh-connected architecture
  - Hardware message switching
  - Software for medium-grain distributed computing
- **Third generation:**
  - Fine-grained distributed computing, with each VLSI chip containing the processor and communication resources

https://hemanthrajhemu.github.io

Multiprocessors and Multicomputer

Taxonomy of MIMD Computers

Parallel Computers

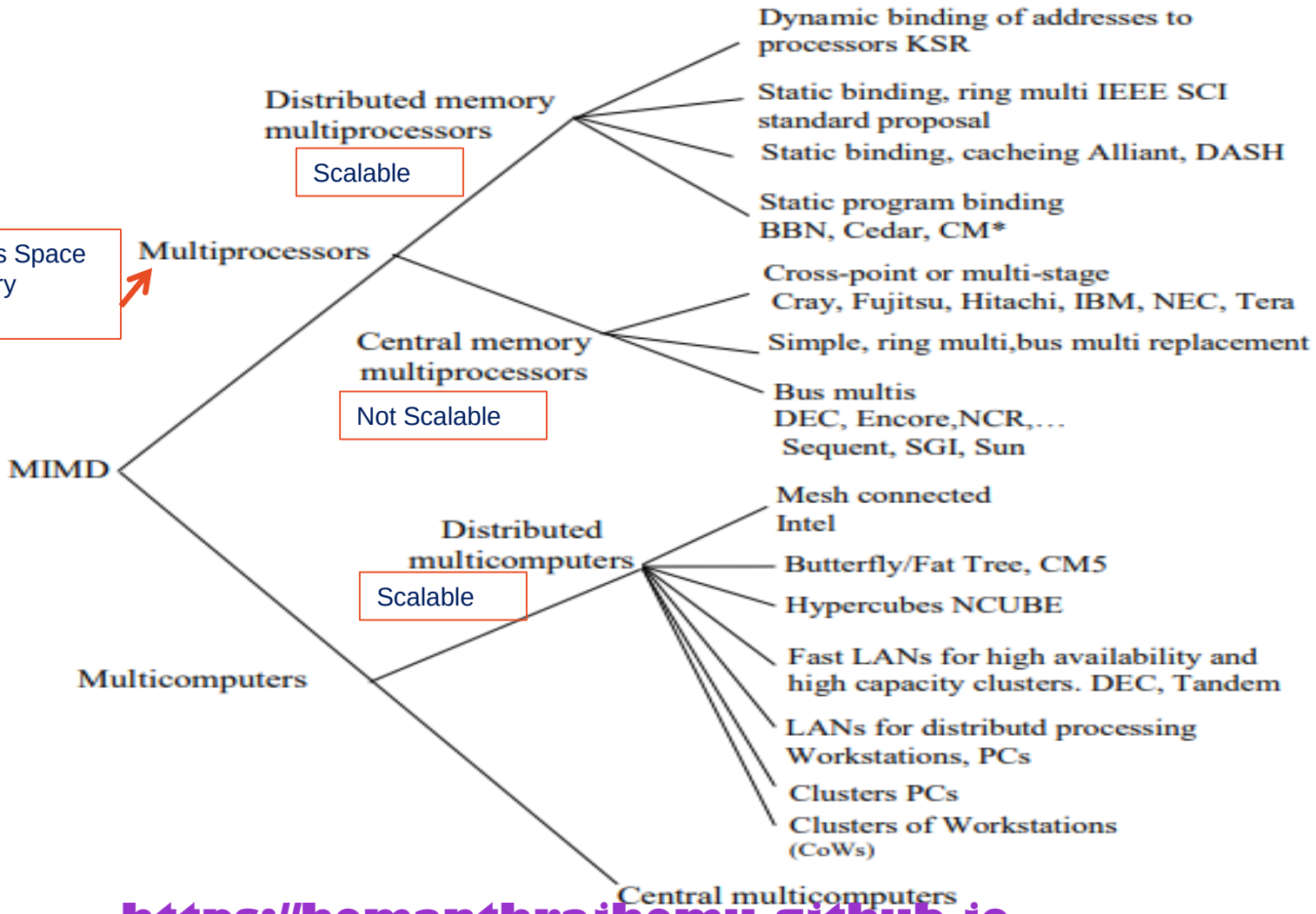
Parallel Computers : Exists

- **SIMD or MIMD configuration**
- **SIMD configuration**
  - For special purpose applications
  - CM - 2 (Connection Machine) on SIMD architecture
- **MIMD configuration**
  - CM - 5 on MIMD architecture
  - Having globally shared virtual address space
- **Scalable multiprocessors or multicomputer**
  - Use distributed shared memory
- **Un-scalable multiprocessors:**
  - Use centrally shared memory

https://hemanthrajhemu.github.io

Taxonomy of MIMD Computers

- Single Address Space
- Shared Memory
- Computation



https://hemanthrajhemu.github.io

Multivector and SIMD Computers

Supercomputer (Have highest operational rate) Classification:

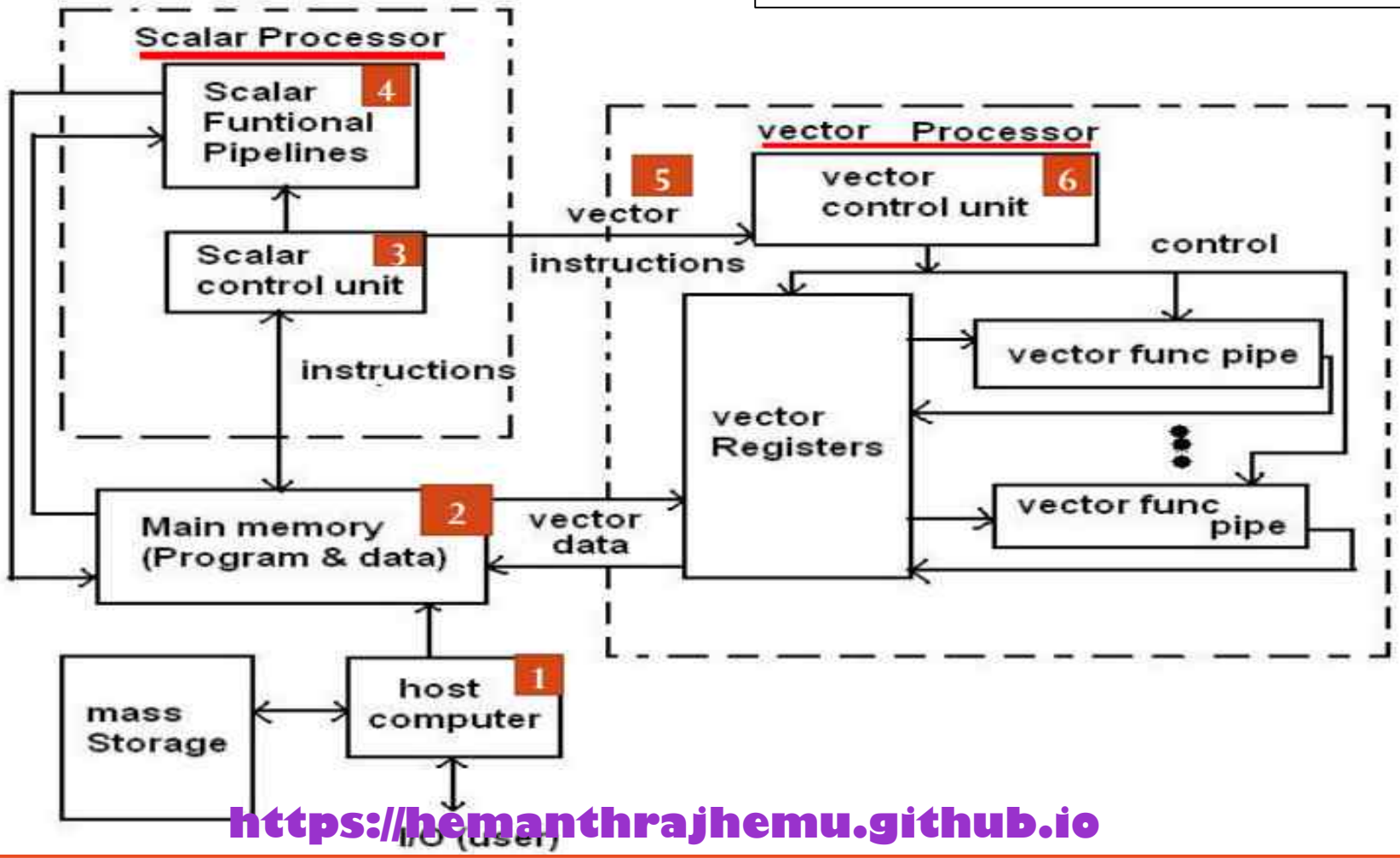
- **Pipelined Vector machine/ Vector Supercomputers:**
  - **Using a few powerful processors equipped with vector hardware**
  - **Vector Processing**

- **SIMD Computers / Parallel Processors:**
  - **Emphasizing massive data parallelism**

Multivector and SIMD Computers

Vector Supercomputers

Vector computer ➔ Is built on top of Scalar Processor.



https://hemanthrajhemu.github.io

Multivector and SIMD Computers

Vector Supercomputers

Step 1-2: Program & data are first loaded into the Main Memory through a Host computer.

Step 3: All instructions are first decoded by the Scalar Control Unit.

Step 4: 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.

Step 5: If the instructions are decoded as a Vector operation, it will be sent to the vector control unit.

Step 6: Vector control unit will supervise the flow of vector data between the main memory and vector functional pipelines.

Note: A number of vector functional pipelines may be built into a Vector Processor.

https://hemanthrajhemu.github.io

Multivector and SIMD Computers

Vector Supercomputers - 2 Models

Model 1:Register-to-register architecture➔

- A fixed number of possibly reconfigurable registers
  - **Which are used to hold all vector operands, intermediate, and final vector results.**
- All registers are accessible in user instructions
- All vector registers length will be fixed(e.g.➔64 bits)

Model 2: Memory-to-memory architecture➔

- Primary memory holds operands and results
- A vector stream unit accesses memory for fetches
- Stores into the main memory in large units➔ superwords (e.g.➔512 bits)

https://hemanthrajhemu.github.io

Multivector and SIMD Computers

SIMD Supercomputers

Single Instruction Stream and over multiple data stream

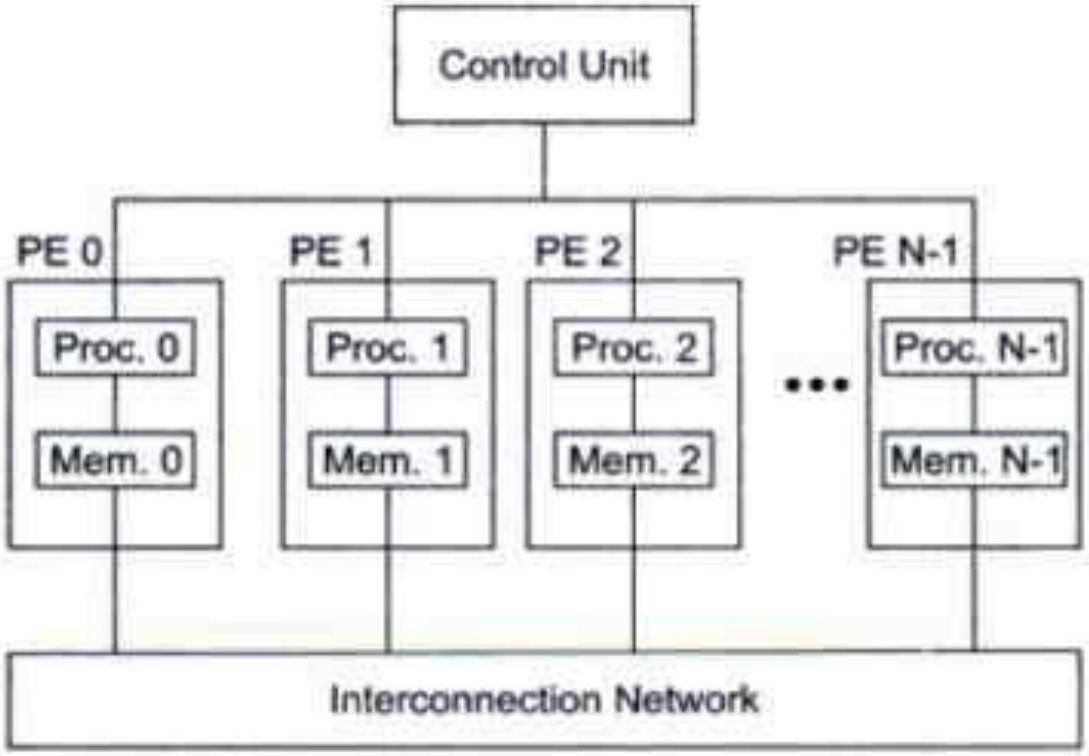


Fig. 1.12 Operational model of SIMD computers

Multivector and SIMD Computers

SIMD Supercomputers

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

(1) N = No. of Processing Elements (PE) in the machine.

(2) C =Set of instructions directly executed by the control unit (CU).

- **Scalar & Program Flow Control Instructions.**

(3) I = Set of instructions broadcast by the CU to all PEs for parallel execution.

- **Include: Arithmetic, logic, data routing, masking,**
- **And other local operations executed by each active PE over data within that PE.**

(4) M = Set of Masking Schemes

- **Each mask partitions the set of PEs into enabled and disabled subsets for a given cycle.**

(5) R = Set of data-routing functions

- **Specifying various patterns to be set up in the interconnection network for inter-PE communications.**

PRAM and VLSI Models

Parallel Random Access Machines :

o Time and Space Complexities

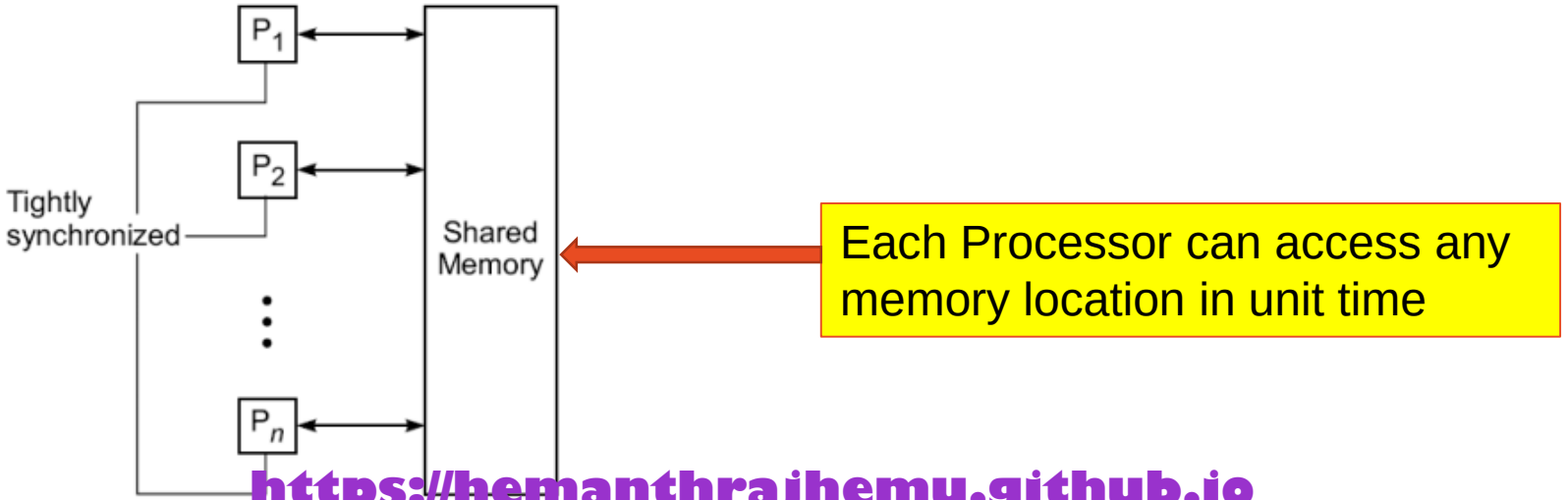
- Time complexity
  - » It is a function of problem size
  - » Asymptotic Time complexity of an algorithm like Big-O etc.
  - » Best case ➔ Worst case
- Space complexity
  - » It is a function of problem size S
  - » Asymptotic Space complexity refers to data storage of large problems
  - » Program and input data storage is not considered.
- Serial and Parallel complexity ➔
  - » Time complexity of an algorithm
  - » Parallel complexity should be lower serial complexity.
- Deterministic and Non-deterministic algorithm
  - » In deterministic ➔ **every operational step is uniquely defined.**
  - » Operations resulting in one outcome➔ **set of possible outcomes.**

https://hemanthrajhemu.github.io

PRAM and VLSI Models

Parallel Random Access Machines :

- Developed by Fortune and Wyllie (1978)
- Objective:
  - » Modeling idealized parallel computers with **zero synchronization or memory access overhead**
- An **n-processor** PRAM has a **globally** addressable Memory



https://hemanthrajhemu.github.io

PRAM and VLSI Models

Parallel Random Access Machines :

o NP Completeness:

- » An algorithm has polynomial complexity ➔ **if there exits polynomial p(s) and its time complexity O(p(s)) for a problem size → s.**
- » Set of problems having **polynomial complexity algorithm by deterministic way** is called **P-class (for polynomial class)**
- » Set of problems solvable by **nondeterministic algorithms in polynomial time** is called **NP-class (for nondeterministic polynomial class)**
- » **P ⊂ NP**
- » **P is solvable where as NP is not solvable**
- » Note: **P= NP/ P≠ NP**➔Is still an open problem

https://hemanthrajhemu.github.io

PRAM and VLSI Models

PRAM Models

Conventional Uniprocessor computers➔

- Modelled as random access machines (RAM)

Parallel random access machines➔

- Modelled to idealize parallel computers with zero synchronization / memory access overhead.
- Used ➔ parallel algorithm development, scalability & complexity analysis.
- Globally addressable shared memory➔ distributed among n Processor Elements (PEs) / centralized.
- PEs➔ operate on synchronized read memory, compute, write memory cycle.

Four memory-update options in PRAMs:

Exclusive Read (ER): Read takes ➔ 1 Cycle atmost 1 Processor

Exclusive Write (EW): Write takes ➔ 1 Cycle atmost 1 Processor

Concurrent read (CR):Read of a memory location takes ➔ Same Cycle for multiple Processor

Concurrent Write (CW):Simultaneously Writes to a same memory location ➔ conflict need to be avoided

https://hemanthrajhemu.github.io

PRAM and VLSI Models

PRAM Variants:- Classified based on Memory read/write : Four memory-update

1. **EREW PRAM model:**
   - ❖ Prohibits more than 1 processor from reading or writing simultaneously to same cell.
2. **CREW PRAM model:**
   - ❖ Mutual exclusion used to avoid write conflicts.
   - ❖ Allows concurrent reads from same memory.
3. **ERCW PRAM model:**
   - ❖ Allows exclusive reads/ concurrent writes from same memory.
4. **CRCW PRAM model:**
   - ❖ Allows either concurrent reads/ concurrent writes from same memory.

PRAM and VLSI Models

Discrepancy with Physical Models of PRAM Variants:-

- Most popular variants: **EREW and CRCW**
- **SIMD** machine with shared architecture is closest architecture **modelled by PRAM**
- PRAM Allows **different** instructions to be executed on **different** processors simultaneously. Thus, PRAM really operates in **synchronized MIMD mode** with shared memory

PRAM and VLSI Models

VLSI Complexity Model:-

- Parallel computers rely on the use of VLSI chip ➔
  - To fabricate components such as Processor arrays, Memory arrays, Switching networks etc.
- Nowadays, VLSI technologies are 2-dimensional.
  - The size of a VLSI chip is proportional to the amount of storage (memory) space available in that chip.

We deal with AT^2 model (2-dimension)

- We can calculate the space complexity of an algorithm by :
  - The chip area (A) of the VLSI chip implementation of that algorithm.
  - If T is the time (latency) needed to execute the algorithm
- Then A.T gives an upper bound on the total number of bits processed through the chip (or I/O).
- For certain computing, there exists a lower bound, f(s), such that

$$A.T^2 >= O(f(s))$$

Where A=chip area and T=time

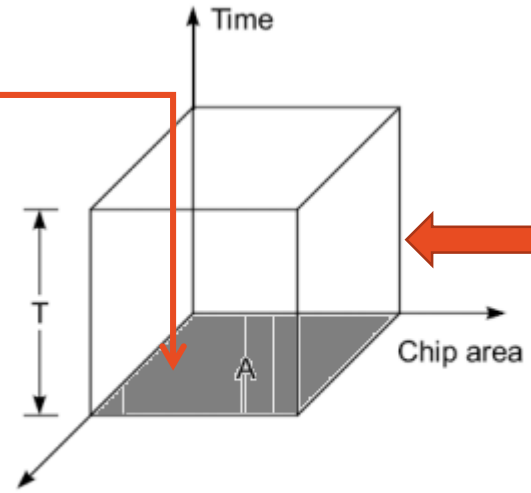
https://hemanthrajhemu.github.io

PRAM and VLSI Models

VLSI Complexity Model:-

Memory Bound on Chip Area:

- Many computations are memory bound➔ has there will be a need in processing large data sets
- Memory requirement of computation sets a lower bound on the chip area (A)
- Amount of information processed by the chip ➔ can be visualized as information flow upward across the chip area



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

Volume of rectangular AT:➔ Product of AT

Note : As the information flows over the time :-
The number of input bits <= Volume AT

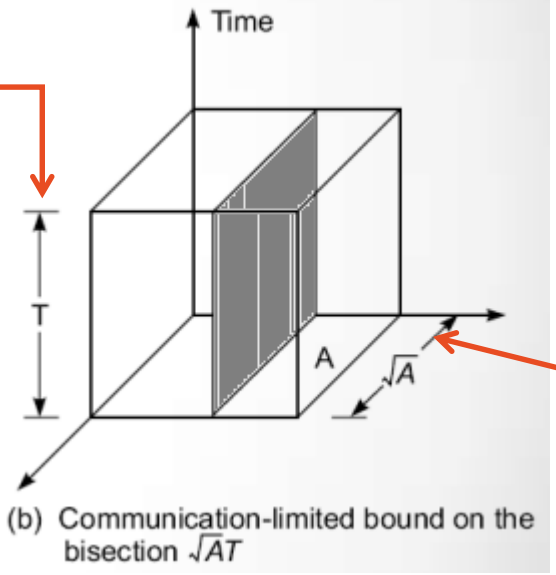
PRAM and VLSI Models

VLSI Complexity Model:-

Bisection Communication Bound, $\sqrt{A}\,T$

- The bisection is represented by the vertical slice across the chip area A.

The height of the cross section is



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

The bisection area represent ➔ the maximum amount of information exchange between the two halves of the chip ➔ during the time period T

The distance of this dimension is

https://hemanthrajhemu.github.io

Chapter-2
Program and Network Properties

In this chapter we will concentrate on following topics:

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

Conditions of Parallelism

Three key areas for significant progress in parallel processing:

- ❖ Computational models for parallel computing
- ❖ Inter-processor communication in parallel architectures
- ❖ System integration for incorporating parallel system

Condition of parallelism Includes:

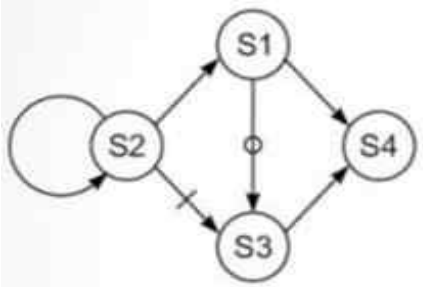
- o Data dependence and resource dependence
- o Hardware and software dependence
- o The role of compiler

Conditions of Parallelism

Data dependence 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)**
- **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.**



(a) Dependence graph

https://hemanthrajhemu.github.io

Conditions of Parallelism

Data dependence and resource dependence

Data Dependence : The ordering relationship between statements is indicated by the data dependence.

Five Types:

1. Flow dependence
2. Antidependence
3. Output dependence
4. I/O dependence
5. Unknown dependence

Conditions of Parallelism

Data dependence and resource dependence

1. Flow dependence : A statement S2 is flow dependent on S1➔ RAW

$$S_1 \rightarrow S_2$$

S1: ADD R1, R2, R3

S2 : SUB R4, R1, R2

ADD R1➔SUB R1

2. Antidependence : Statement S2 is antidependent on the statement S1➔WAR

$$S_1 \mapsto S_2$$

S1: ADD R1, R2, R3

S2 : SUB R4, R1, R2

S3: ADD R1, R2, R3

S3 depends on S2 ➔
i.e S3 should be executed after S2

https://hemanthrajhemu.github.io

Conditions of Parallelism

Data dependence and resource dependence

3. Output dependence : Two statements are output dependent if they produce (write) the same output variable. Also called WAW hazard and denoted as

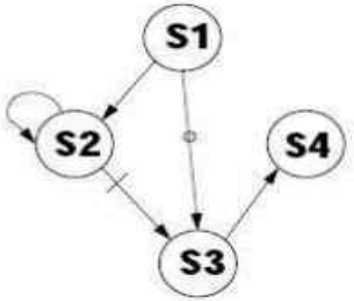
$$S_1 \mapsto S_2$$

S1: ADD R1, R2, R3

S2 : SUB R4, R1, R2

S3: ADD R1, R2, R3

S3 depends on S1 ➔
i.e S3 and S1 writes to R1



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.

https://hemanthrajhemu.github.io

Conditions of Parallelism

Data dependence and resource dependence

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
   - 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.

Conditions of Parallelism

Data dependence and resource dependence

Control Dependence:➔ Dependency is know at Run time

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 Dependency ➔ Prohibits Parallelism

https://hemanthrajhemu.github.io

Conditions of Parallelism

Data dependence and resource dependence

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.

Conditions of Parallelism

Data dependence and resource dependence

Bernstein's Conditions - 1 : Bernstein's conditions are a set of conditions which must exist if two processes can execute in parallel.

– Notation

- I_i is the set of all input variables for a process P_i .
- O_i is the set of all output variables for a process P_i .
- If P1 and P2 can execute in parallel (which is written as P1 || P2), then:

$$I_1 \cap O_2 = \varnothing$$

$$I_2 \cap O_1 = \varnothing$$

$$O_1 \cap O_2 = \varnothing$$

Conditions of Parallelism

Data dependence and resource dependence

Bernstein's Conditions - 2 : 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.

https://hemanthrajhemu.github.io

Conditions of Parallelism

Hardware and Software Parallelism

Hardware Parallelism : Hardware parallelism is defined by machine architecture and hardware multiplicity.

- 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.
- A machine with n processors ➔ with k-issue should be able to handle a maximum of nk threads simultaneously.
- 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)

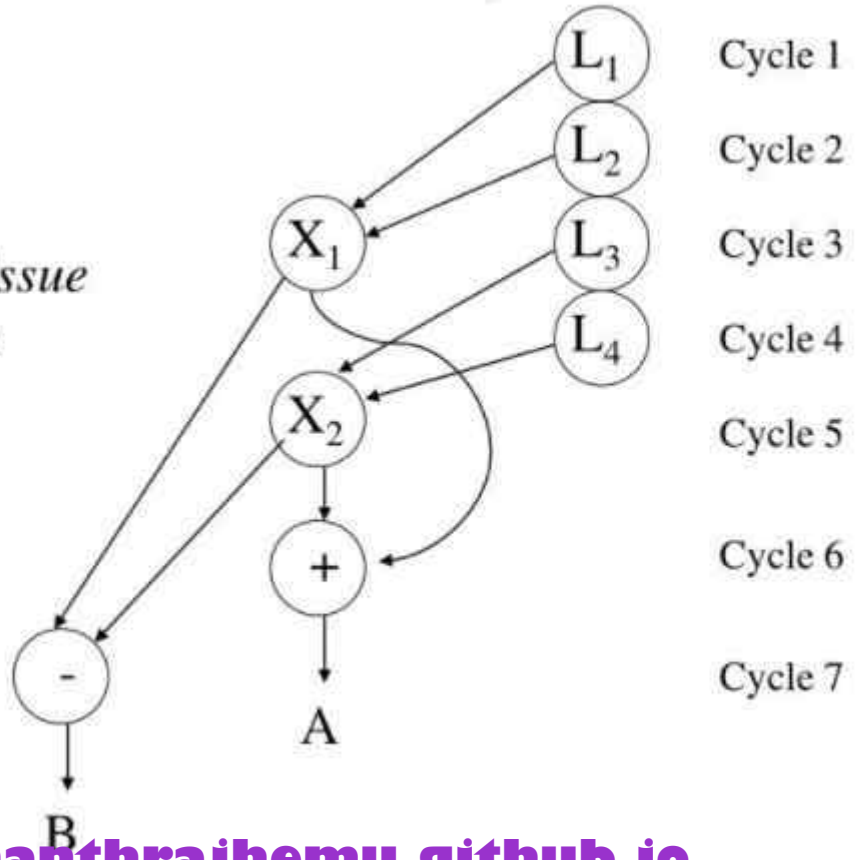
https://hemanthrajhemu.github.io

Conditions of Parallelism

Hardware and Software Parallelism

Hardware Parallelism :

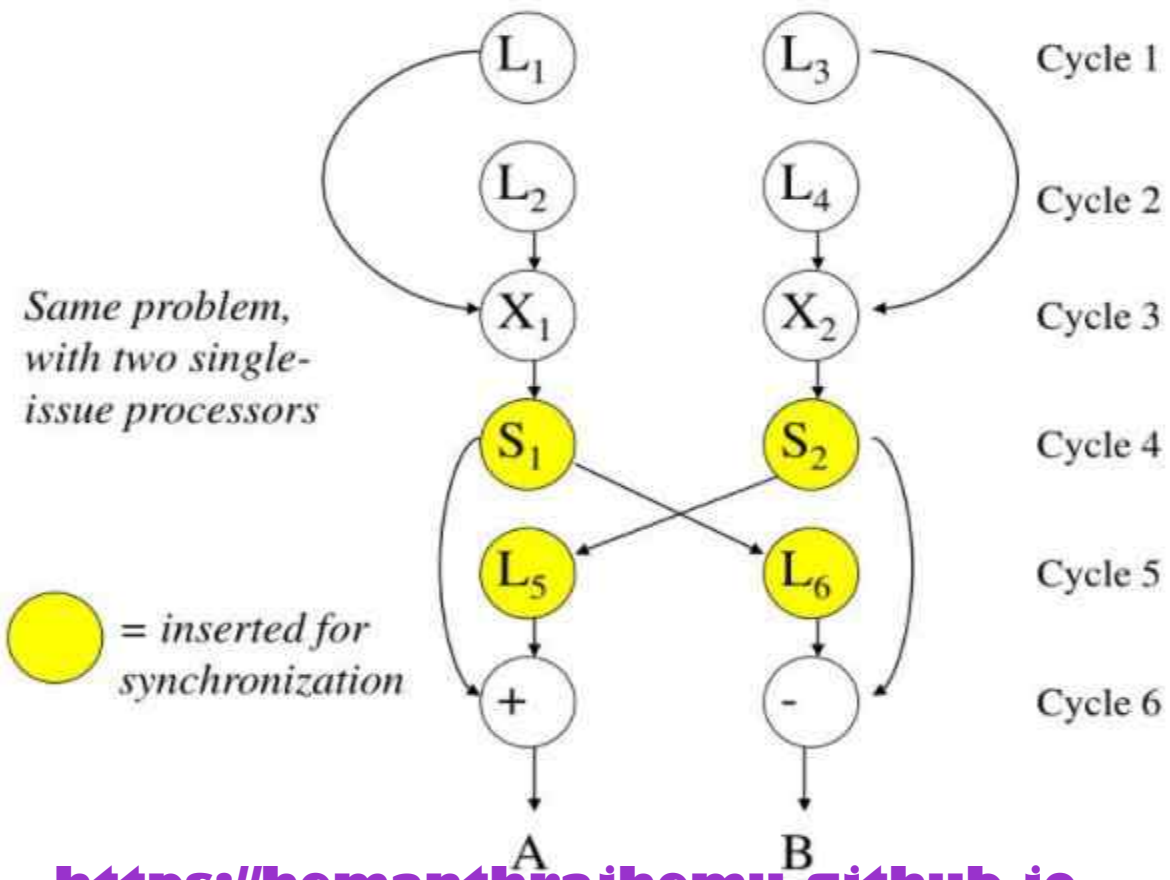
Same problem, but considering the parallelism on a two-issue superscalar processor.



Conditions of Parallelism

Hardware and Software Parallelism

Hardware Parallelism :

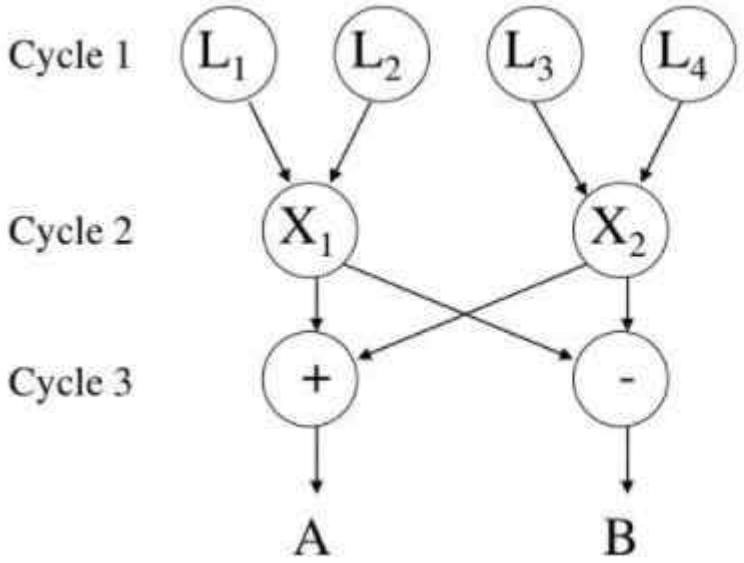


Conditions of Parallelism

Hardware and Software Parallelism

Software Parallelism :

- Software parallelism is defined by the control and data dependence of programs,
- Is revealed in the program's flow graph.
- It is a function of algorithm, programming style, and compiler optimization.



Maximum software parallelism (L=load, X/+/- = arithmetic).

https://hemanthrajhemu.github.io

Conditions of Parallelism

Hardware and Software Parallelism

Types of Software Parallelism :

1. Control Parallelism:
   - Two or more operations can be performed simultaneously.
   - This can be detected by a compiler, or a programmer can explicitly indicate control parallelism
     - By using special language constructs or dividing a program into multiple processes.
2. Data parallelism:
   - Multiple data elements have the same operations applied to them at the same time.
   - This offers the highest potential for concurrency (in SIMD and MIMD modes).
   - Synchronization in SIMD machines handled by hardware.

Conditions of Parallelism

Hardware and Software Parallelism

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.

Conditions of Parallelism

Program Partitioning & Scheduling

Program Partitioning & Scheduling :

Mainly deals with:-

1. **Grain Sizes and Latency:**
2. **Grain Packing and Scheduling**
3. **Grain determination and scheduling optimization**

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Sizes and Latency)

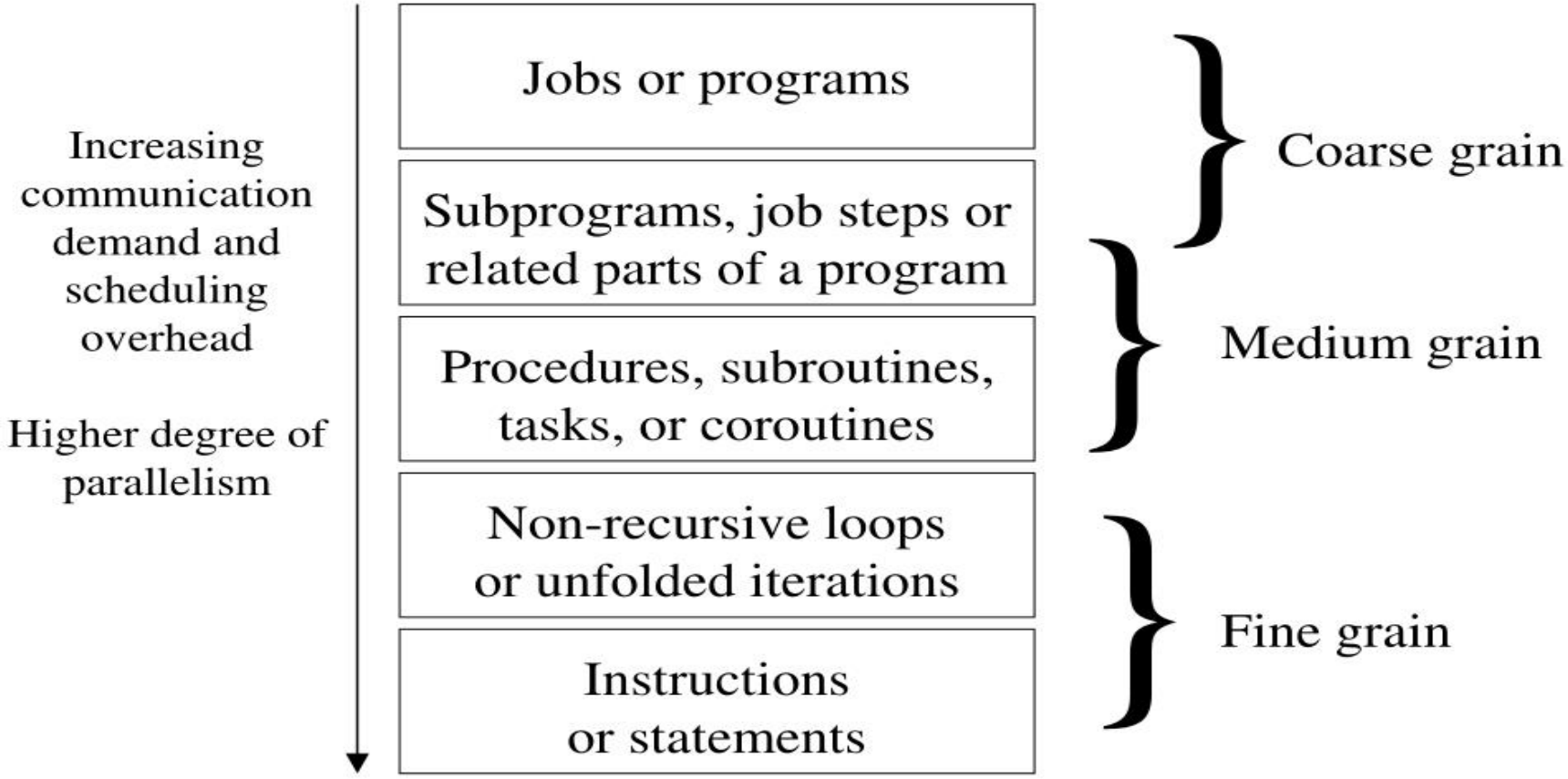
Grain Sizes and Latency:

- Size of a program 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

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Sizes and Latency)

Parallelism has exploited at various Processing levels:



https://hemanthrajhemu.github.io

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Sizes and Latency)

Instruction Level Parallelism:

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

https://hemanthrajhemu.github.io

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Sizes and Latency)

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 loop operations are self scheduled to execute on a parallel or vector machine (MIMD Mode)
- Some loops (e.g. recursive) are difficult to handle.
- Loop-level parallelism is still considered fine grain computation.

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Sizes and Latency)

Procedure Level Parallelism:

- It is 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.

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Sizes and Latency)

Subprogram Level Parallelism:

- Job step level and related subprograms
- 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.

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Sizes and Latency)

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).

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Sizes and Latency)

Final Summary :

- Fine-grain parallelism is ➔exploited at instruction or loop levels
  - Assisted by the compiler.
- Medium-grain parallelism is ➔At task or job step level
  - Requires programmer and compiler support.
- Coarse-grain parallelism is ➔ At Program Level
  - Relies heavily on effective OS support.
- Shared-variable communication used at fine- and medium-grain levels.
- Message passing can be used for medium- and coarse-grain communication
  - But fine-grain really need better technique because of heavier communication requirements.

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Sizes and Latency)

Communication Latency:

- Balancing granularity and latency can yield better performance.
- Various latencies attributes ➔ To machine architecture, technology, and communication patterns used.
- Latency imposes a limiting factor➔ On scalability of machine size.
- Ex. Memory latency increases as memory capacity increases ➔
  - Limiting the amount of memory that can be used with a given tolerance for communication latency.

https://hemanthrajhemu.github.io

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Sizes and Latency)

Inter-process Communication Latency:

- Needs to be minimized by system designer
- Affected by signal delays and communication patterns
- Example:
  - n communicating tasks may require ➔
  - n (n - 1)/2 communication links
  - The complexity grows quadratically , effectively limiting the number of processors in the system.

https://hemanthrajhemu.github.io

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Packing and Scheduling)

Grain Packing and Scheduling:

Two questions exists in parallel programming:

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

There is an obvious tradeoff between the:

- Time spent for scheduling and synchronizing parallel grains and the speedup obtained by parallel execution.
- One approach to the problem is called "grain packing."

https://hemanthrajhemu.github.io

Conditions of Parallelism

Program Partitioning & Scheduling(Grain Packing and Scheduling)

Grain Packing and Scheduling:

- The grain size problem requires determination of both➔
  - The number of partitions
  - The size of grains in a parallel problem
- Solution is through problem dependent and machine dependent in parallelism.
- A short schedule is required for fast execution of subdivided program modules.

https://hemanthrajhemu.github.io

Conditions of Parallelism

Program Partitioning & Scheduling

(Grain Determination and Scheduling Optimization)

Grain determination and scheduling optimization:

Includes 4 steps:

Step 1: Construct a fine-grain program graph

Step 2: Schedule the fine-grain computation

Step 3: Grain packing to produce coarse grains

Step 4: Generate a parallel schedule based on the packed graph

Conditions of Parallelism

Program Partitioning & Scheduling(Program Graphs and Packing)

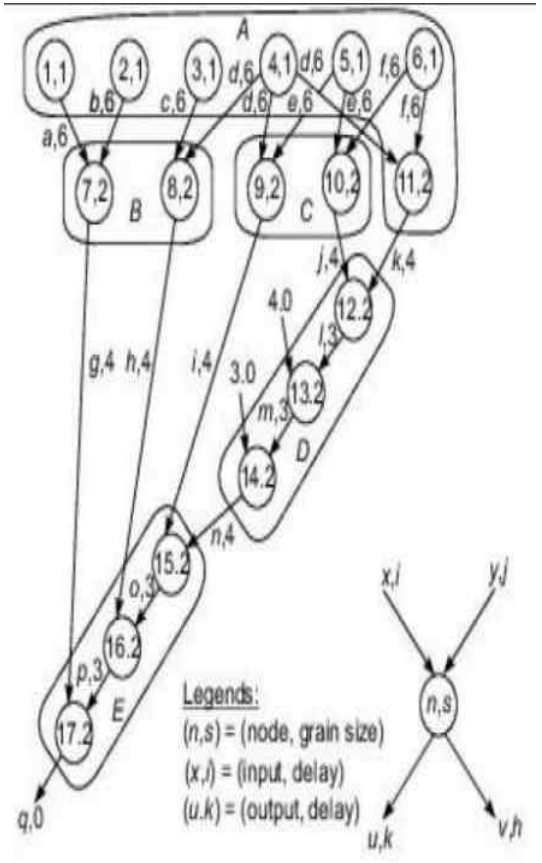
A program graph is like a dependence graph

- Nodes= {(n, s)} where n= node name and s= size
  - Larger the s = larger grain size

– Edges= {(v,d)} where v= communicated variable and d=communication delay

– Packing 2 or more nodes produces a larger grain size node and possibly more edges to other nodes.

– Packing helps in eliminating unnecessary communication delay or reduce overall scheduling overhead.



(a) Fine-grain program graph before packing

https://hemanthrajhemu.github.io

Conditions of Parallelism

Program Partitioning & Scheduling(Program Graphs and Packing)

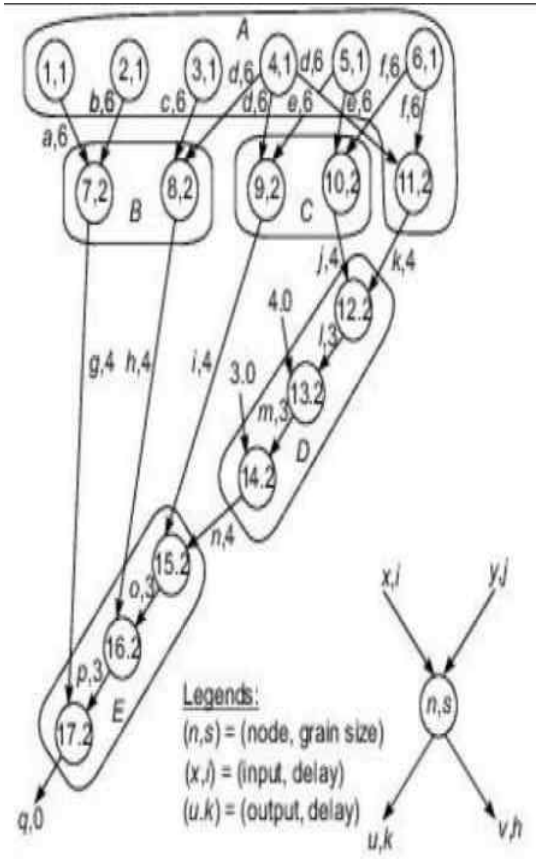
- Nodes l, 2, 3, 4, 5, and 6 are memory reference (data fetch} operations.
- Each takes one cycle to address and six cycles to fetch from memory.

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



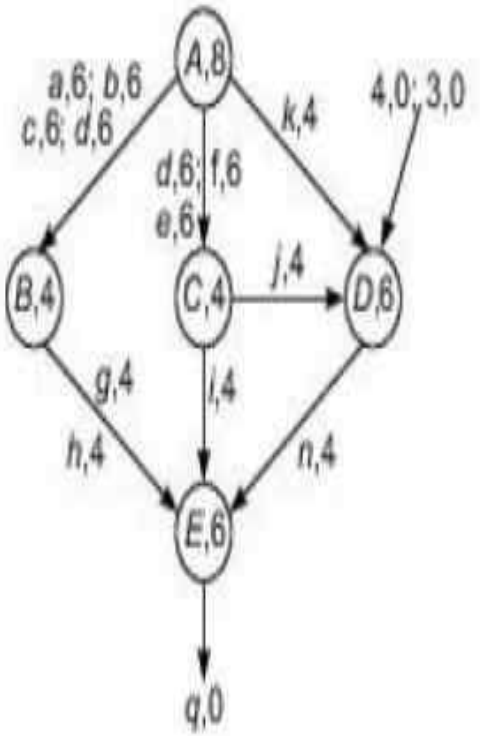
(a) Fine-grain program graph before packing

https://hemanthrajhemu.github.io

Conditions of Parallelism

Program Partitioning & Scheduling(Program Graphs and Packing)

- Each node in the program graph corresponds to a computational unit in the program.
- The grain size is measured by the number of basic machine cycle (including both processor and memory cycles) needed to execute all the operations within the node.
- Grain size reflects the number of computations involved in a program segment.
- Fine-grain nodes have a smaller grain size. and coarse-grain nodes have a larger grain size.



(b) Coarse-grain program graph after packing

Program Flow Mechanisms

There are three flow mechanism :

- Control flow mechanism(used by conventional computers)
  - Order of program execution ➔ **Stated in the user program**
- Data-driven mechanism(used by data-flow computers)
  - Order of program execution ➔ **Is driven by data(operand) availability**
- Demand-driven mechanism(used by reduction computers)
  - Order of program execution ➔ **Based on the demand for its result by other computations**

Program Flow Mechanisms

Control flow mechanism

Control flow mechanism(used by conventional computers):

- Conventional von Neumann computers use a ➔
  - **Program counter to sequence the execution of instruction in a program.**
  - **This sequential execution style is called control-driven.**
- Control flow computers use **shared** memory to hold program instructions and data objects.
- A uniprocessor computers is inherently sequential ➔ Use control-driven mechanism.
- Control flow can be made parallel by using parallel language constructs or parallel compiler.
- Control flow machines give complete control, but are less efficient than other approaches.

Program Flow Mechanisms

Data flow mechanism

Data-driven mechanism(used by data-flow computers):

- Dataflow computers emphasize a high degree of parallelism at the fine-grain instructional level.
- Data flow machines have ➔
  - **High potential** for parallelism and throughput and freedom from side effects,
  - **But have high** control overhead, lose time waiting for unneeded arguments, and difficulty in manipulating data structures.
- **No need for ➔ shared memory**, program counter, control sequencer
- **Special mechanisms are required to➔ detect data availability**, match data tokens with instructions needing them, enable chain reaction of asynchronous instruction execution

https://hemanthrajhemu.github.io

Program Flow Mechanisms

Demand flow mechanism

Demand-driven mechanism(used by reduction computers):

- Data-driven machines select instructions for execution based on the availability of their operands ➔ this is essentially a bottom-up approach.
  - **Eager Evaluation**
- 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.**
  - **Lazy Evaluation**

Program Flow Mechanisms

Demand flow mechanism

Demand-driven mechanism(used by reduction computers): 2 Reduction Models:

1. String-reduction model:

– Each demander gets a separate copy of the expression string ➔

- To evaluate each reduction step has an operator
- Embedded reference to demand the corresponding input operands.
- Each operator is suspended while arguments are evaluated

2. Graph-reduction model:

– Expression graph 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.

https://hemanthrajhemu.github.io

Table 2.1 *Control-Flow, Dataflow, and Reduction Computers*

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 Architectures

- 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

https://hemanthrajhemu.github.io

System Interconnect Architectures

- Static and dynamic networks are used
  - For interconnecting computer subsystems
  - For constructing multiprocessors or multicomputers
- Ideal: Construct a low-latency network with a high data transfer rate

System Interconnect Architectures
Network Properties and Routing

- Static Networks:
  - Point-to-Point direct connections which **will not change** during program execution
- Dynamic Networks:
  - Implemented with **switched channels.**
  - They are **dynamically** configured to match the communication demand in user programs

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Goals and Analysis:

- 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**

System Interconnect Architectures
Network Properties and Routing

Terminologies used in Network Properties and Routing:

- Network usually represented by a graph :
  - With a finite number of nodes
  - Linked by directed or undirected edges.
- Number of nodes in graph = **network size** .
- Number of edges (links or channels) incident on a node ➔ **node degree d** (also note in and out degrees when edges are directed).
  - Node degree reflects number of I/O ports associated with a node, and should ideally be small and constant.

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Terminologies used in Network Properties and Routing:

- **Diameter D of a network**
  - Is the maximum shortest path between any two nodes
  - Measured by the number of links traversed
  - This should be as small as possible (from a communication point of view)
- **Channel bisection width b** ➔ minimum number of edges cut to split a network into two parts each having the same number of nodes.
  - Since each channel has **w bit wires**, the wire bisection width **B = bw**.
  - Bisection width provides **good indication** of **maximum communication bandwidth** along the bisection of a network,
  - And all other cross sections should be **bounded** by the bisection width.

System Interconnect Architectures
Network Properties and Routing

Terminologies used in Network Properties and Routing:

- **Wire (or channel) length** ➔ length (e.g. weight) of edges between nodes.
- **Network is symmetric** ➔ if the topology is the same looking from any node; these are easier to implement or to program.

System Interconnect Architectures
Network Properties and Routing

Data Routing Functions:➔ Is used for inter-PE data exchange

- Shifting
- Permutation (one to one)
- Broadcast (one to all)
- Multicast (many to many)
- Personalized broadcast (one to many)
- Shuffle
- Exchange Etc.

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Data Routing Functions:➔Permutation (one to one)

- Given n objects, there are n ! ways in which they can be reordered (one of which is no reordering).
- For example: π = (a , b , c) (d , e)
  - Possible mapping are: a→b, b→c, c→a, d→e, e→d in a circular fashion.
  - The cycle (a , b , c) has 3 periods and (d , e)has 2 periods. Hence permutation π has a period=3 * 2 = 6
  - If one has to be permuted then it has identity mapping

I = (a) (b) (c) (d) (e)

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Data Routing Functions:➔Permutation (one to one)

- A permutation can be specified by giving the rule for reordering a group of objects.
- Permutations can be implemented using crossbar switches, multistage networks, shifting, and broadcast operations.
- The time required to perform permutations of the connections between nodes often dominates the network performance when n is large.

System Interconnect Architectures
Network Properties and Routing

Data Routing Functions:➔Perfect Shuffle and Exchange

- Perfect Shuffle is a special permutation function by Harold Stone
- The entries according to the mapping of the k-bit binary number

a b … k to b c … k a

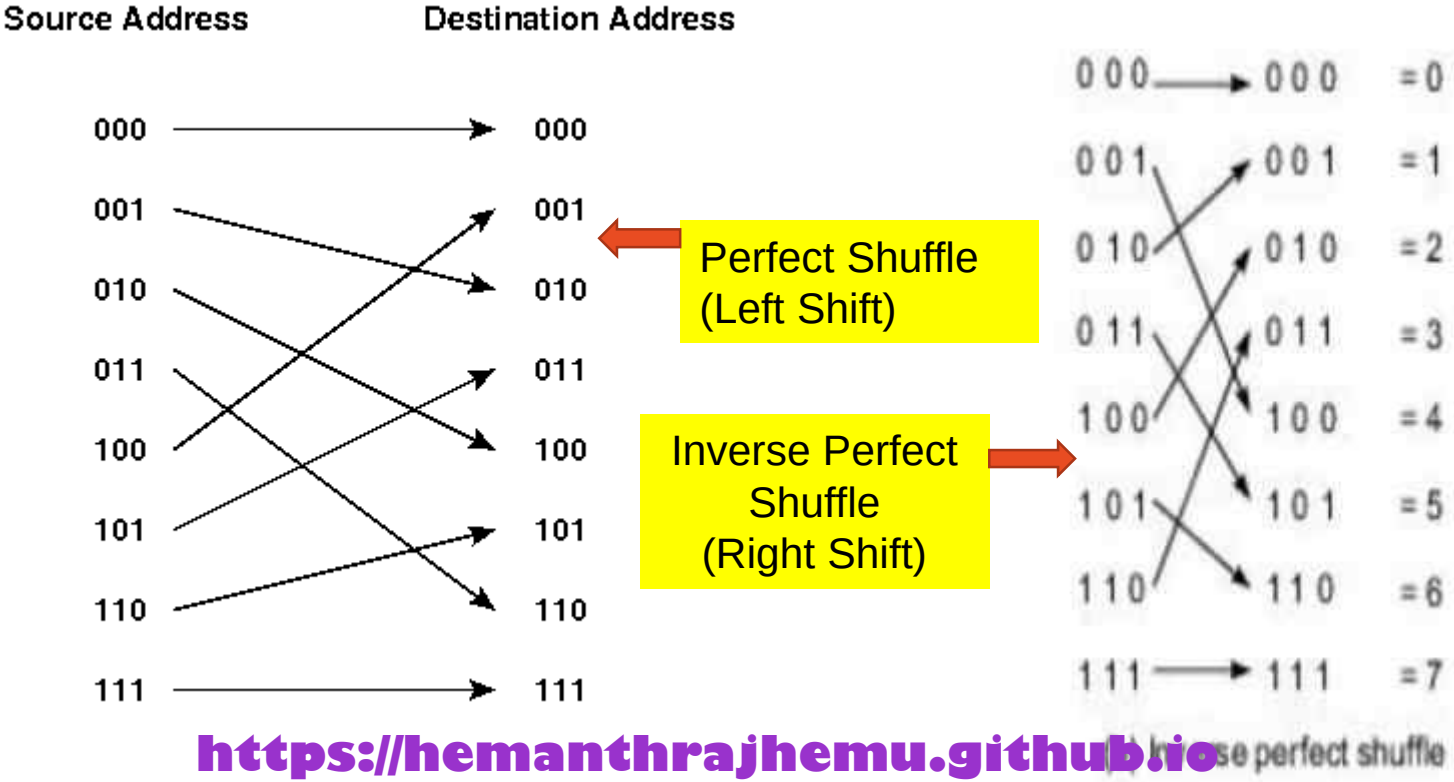
- That is, shifting 1 bit to the left and wrapping it around to the least significant bit position.
- The inverse perfect shuffle reverses the effect of the perfect shuffle.

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Data Routing Functions:➔Perfect Shuffle and Exchange

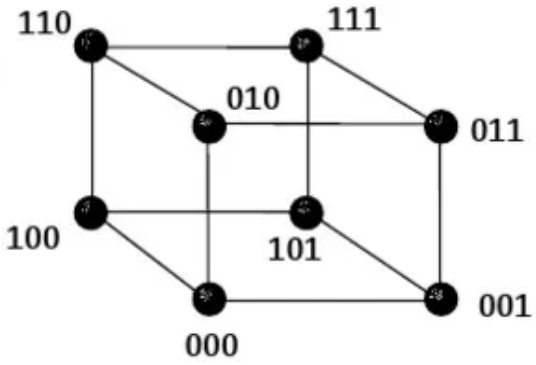
– Perfect Shuffle is a special permutation function by Harold Stone



https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

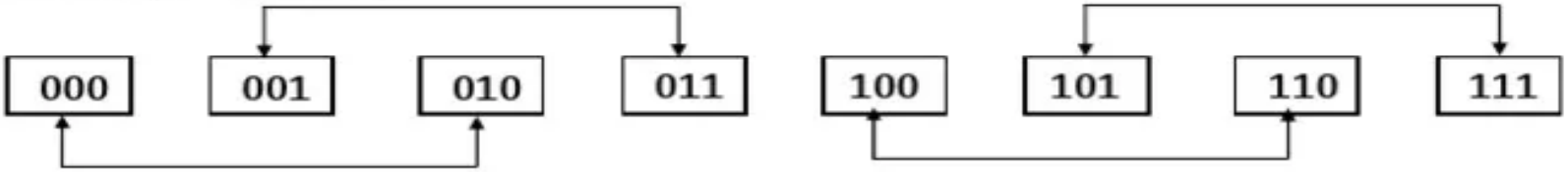
Data Routing Functions:➔Hypercube Routing Functions



A 3-cube with nodes $C_2C_1C_0$

Routing by least significant bit C_0

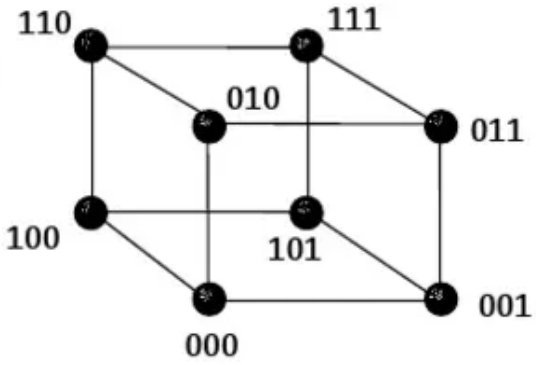
Routing by middle bit C_1



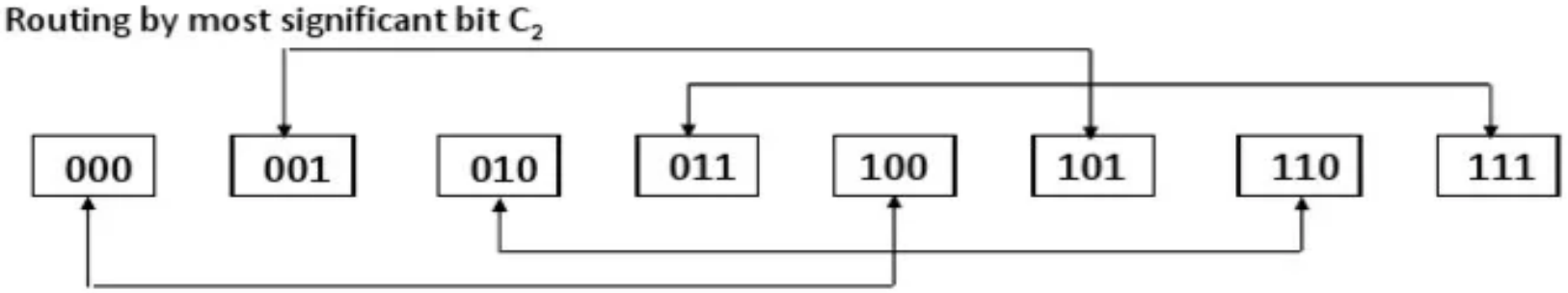
https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Data Routing Functions:➔Hypercube Routing Functions



A 3-cube with nodes $C_2C_1C_0$



https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Data Routing Functions:➔Broadcast and Multicast

- Broadcast is one-to-all mapping
  - This can be achieved in SIMD computers.
  - Message passing multicomputers also has mechanisms to broadcast.
- Multicast is one-to-many mapping
  - Is implemented by matching the destination nodes in the network.

System Interconnect Architectures
Network Properties and Routing

Data Routing Functions:➔Network Performance

- Functionality:-
  - **How the network supports data routing, interrupt handling, synchronization, request/message combining, and coherence.**
- Network latency:-
  - **Worst-case time for a unit message to be transferred.**
- Bandwidth
  - **Maximum data rate**
- Hardware complexity
  - **Implementation costs for wire, logic, switches, connectors, etc.**
- Scalability
  - **How easily does the scheme adapt to an increasing number of processors, memories, etc.?**

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ **Static networks use direct links which are fixed once built.**

Few topologies are:

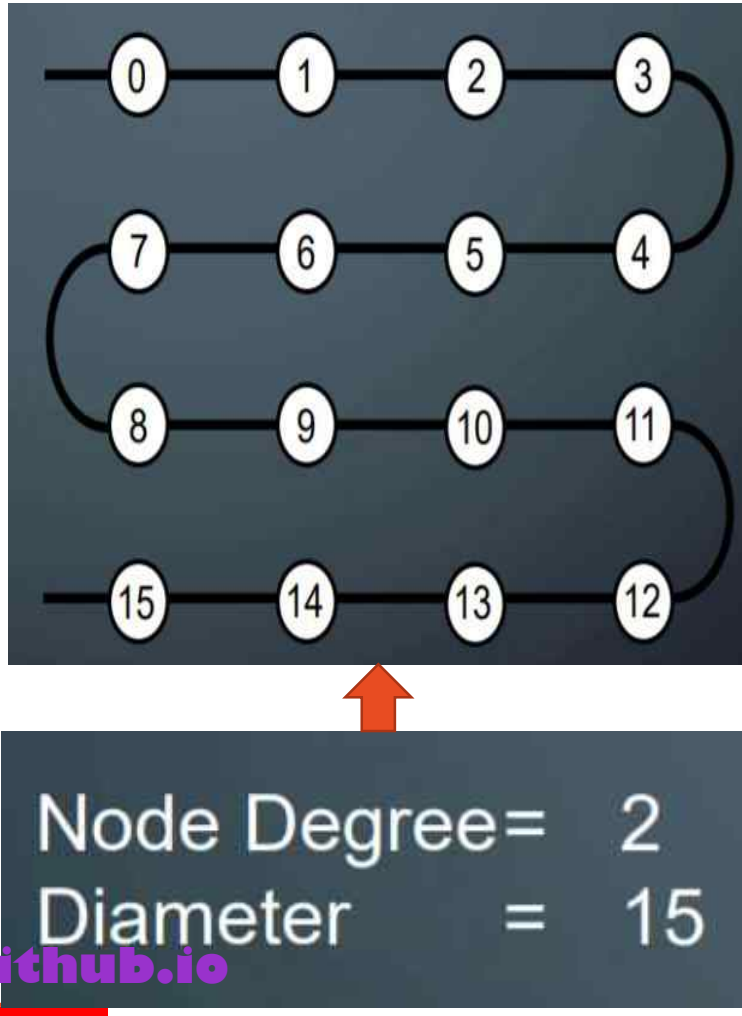
- Linear Array
- Ring and Chordal Ring
- Barrel Shifter
- Tree and Star
- Fat Tree
- Mesh and Torus

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ Linear Array

- It is one-dimensional network where ➔
  - N nodes connected by n-1 links in a line.
- Internal nodes have degree 2;
- End nodes have degree 1.
- Diameter = n-1
- Bisection = 1
- For small n, this is economical, but for large n, it is obviously inappropriate.

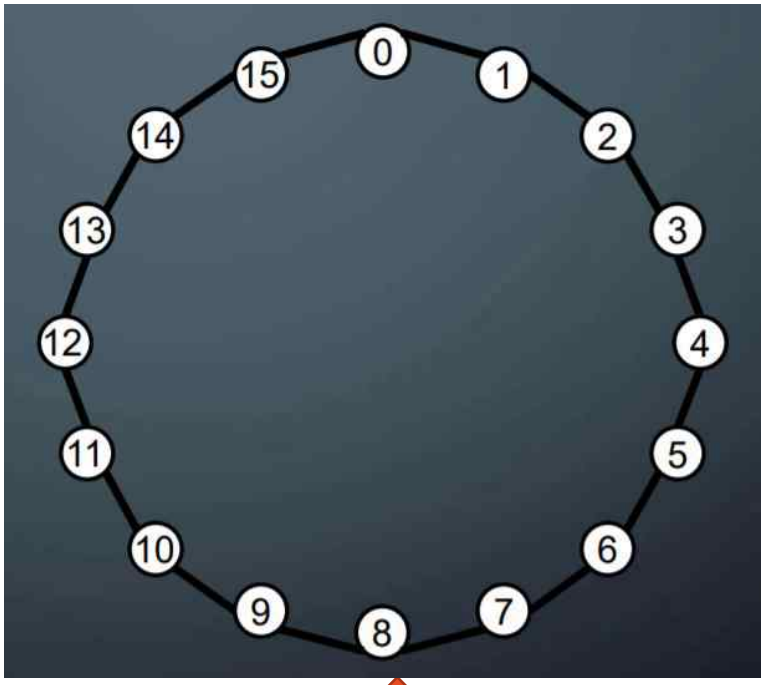


https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ Ring, Chordal Ring

- Like a linear array, but the two end nodes are connected by an n th link;
- The ring can be uni-directional or bidirectional.
- Diameter is n/2 for a bidirectional ring, or n for a unidirectional ring.



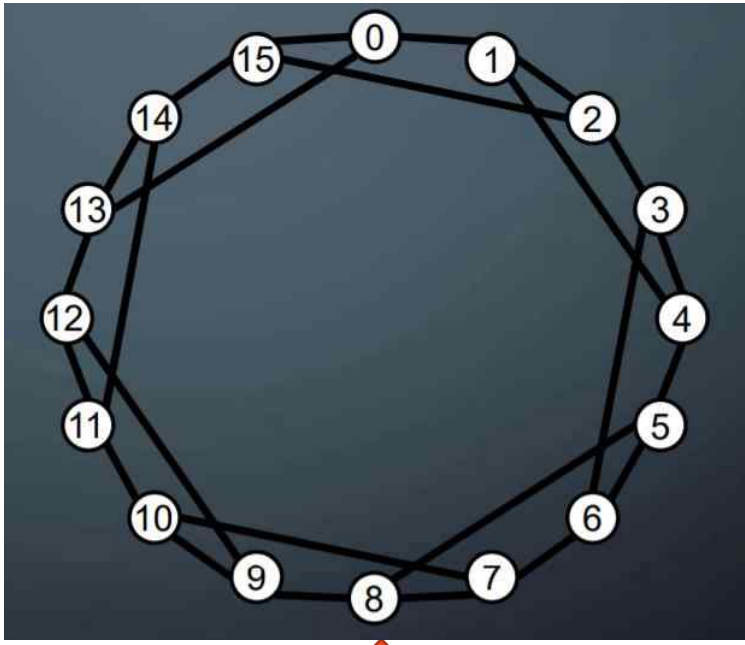
Node Degree = 2
Diameter = N/2 = 8

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ Ring, Chordal Ring

- By adding additional links (e.g. “chords” in a circle)
- The node degree is increased, and we obtain a chordal ring.
- This reduces the network diameter.
- In the limit, we obtain a fully-connected network, with a node degree of n -1 and a diameter of 1.



Node Degree= 3
Diameter = 5

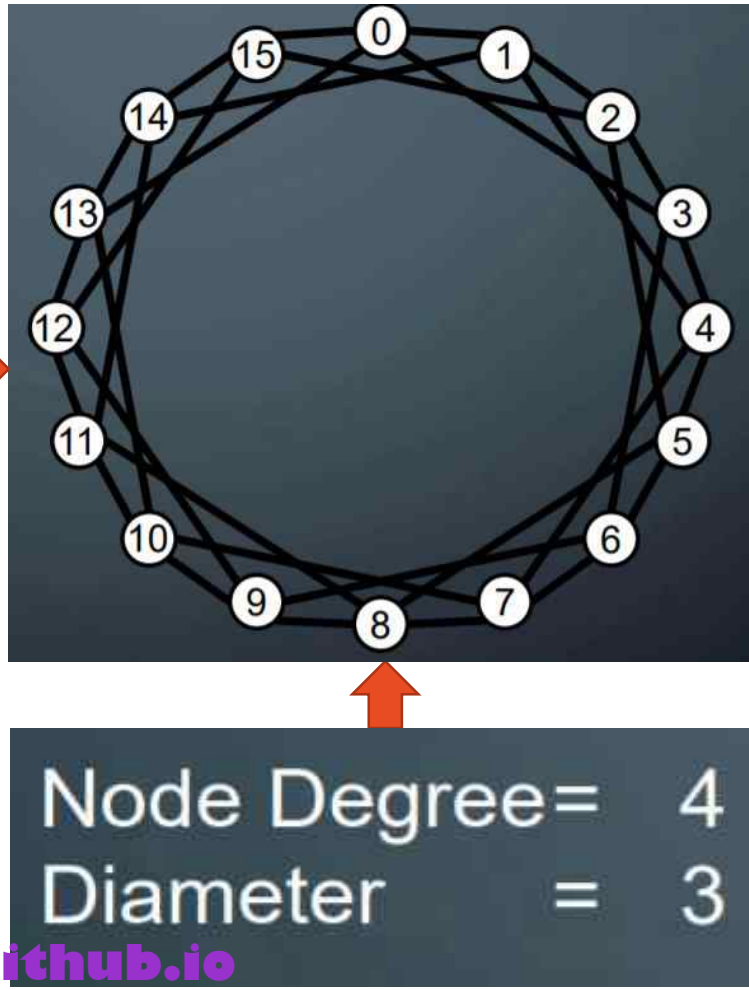
System Interconnect Architectures

Network Properties and Routing

Static Connection Networks ➔ Ring, Chordal

Ring

Other example for Chordal Ring



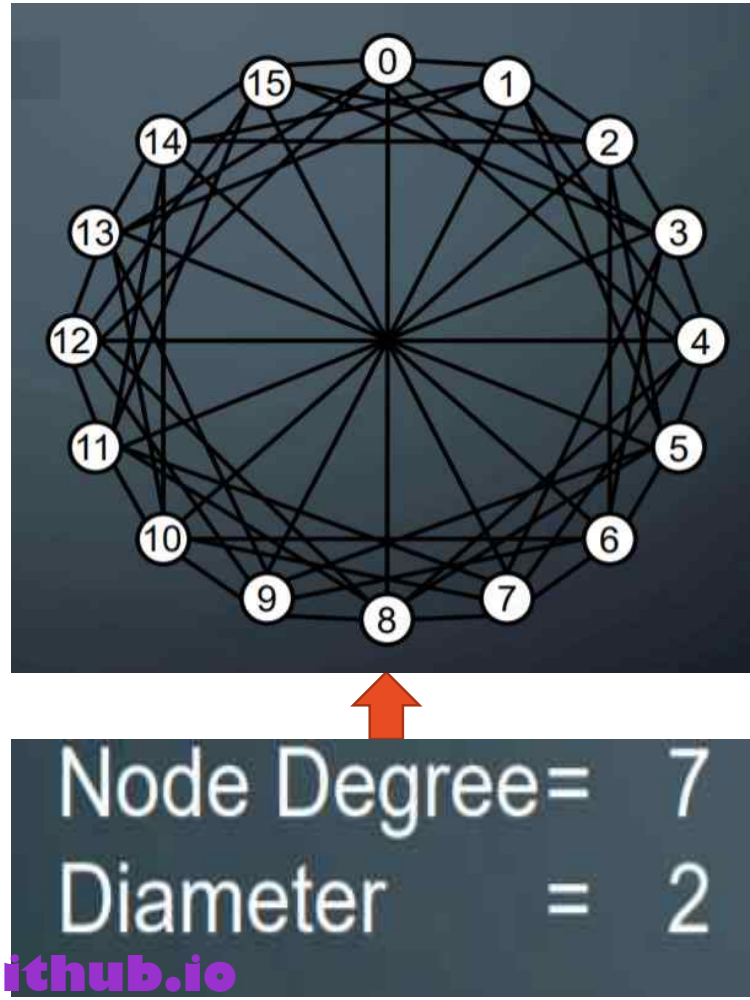
https://hemanthrajhemu.github.io

System Interconnect Architectures

Network Properties and Routing

Static Connection Networks ➔ Barrel Shifter

Barrel shifter is obtained from Ring ➔ By adding extra lines from each node to those nodes having a distance equal to an integer powers of 2



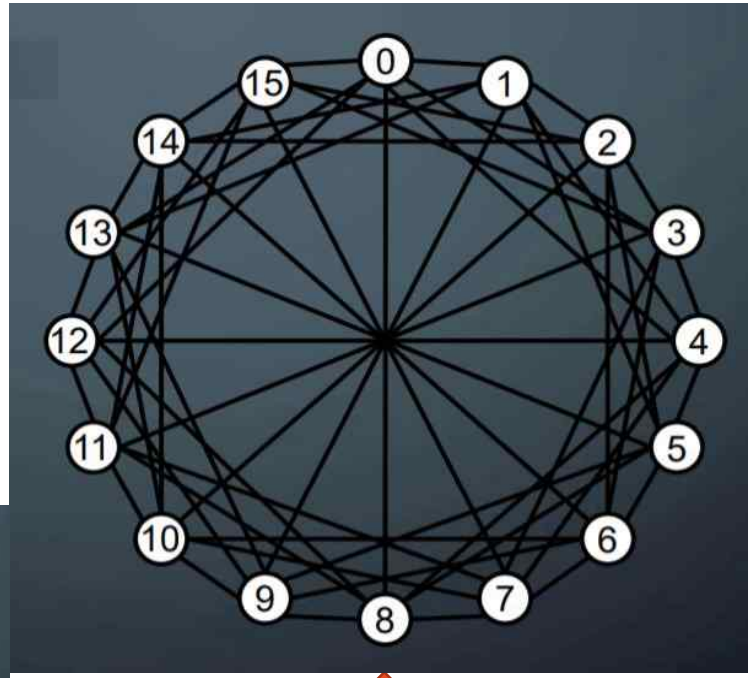
https://hemanthrajhemu.github.io

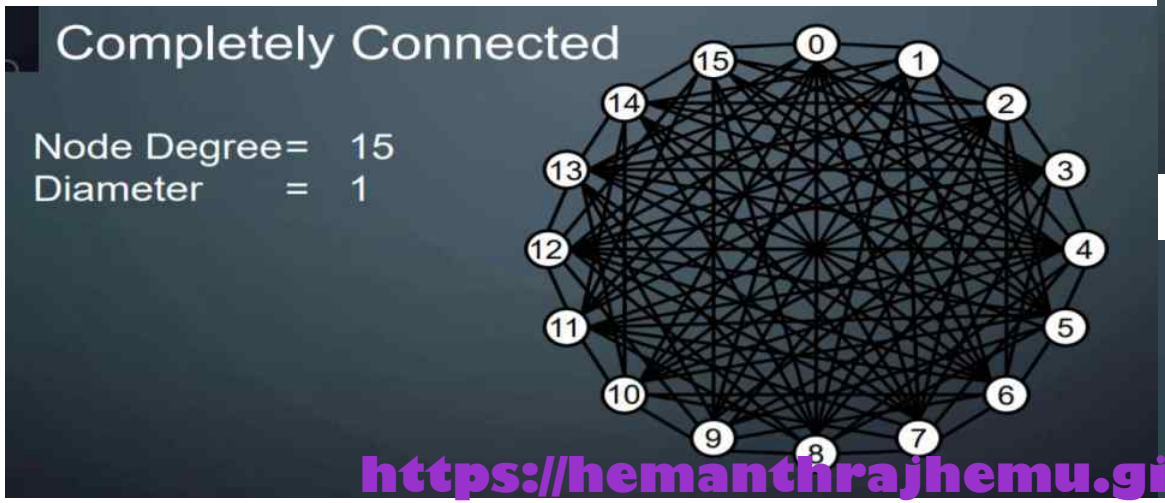
System Interconnect Architectures

Network Properties and Routing

Static Connection Networks ➔ Barrel Shifter

Barrel shifter is obtained from Ring ➔ By adding extra lines from each node to those nodes having a distance equal to an integer powers of 2





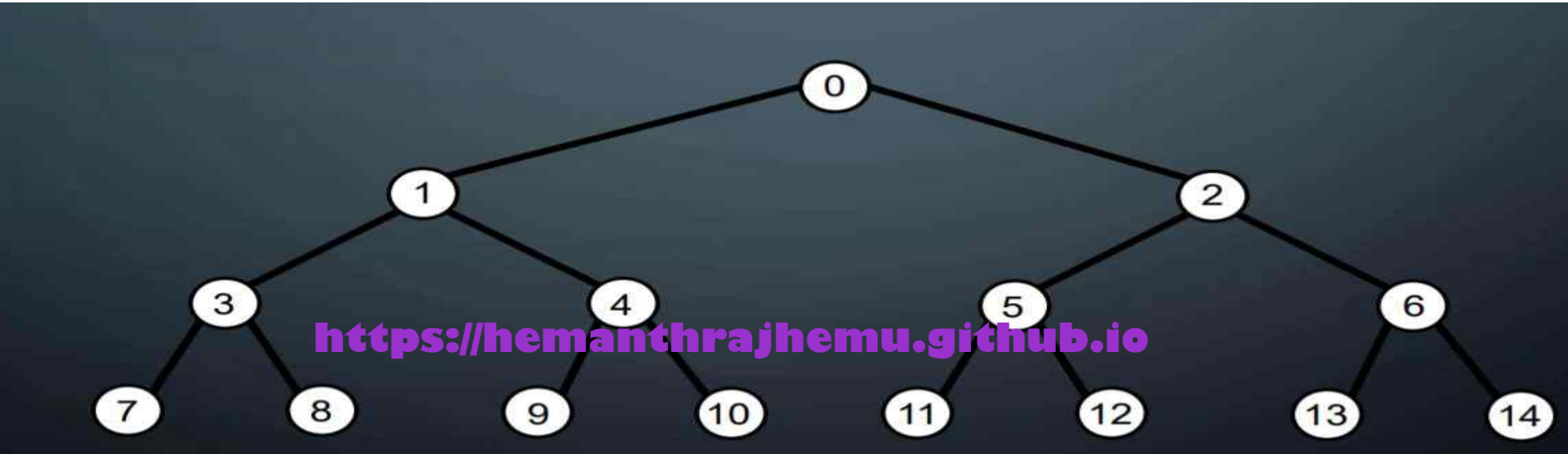
Node Degree= 7
Diameter = 2

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ Tree and Star

- A k-level completely balanced binary tree will have N = 2k – 1 nodes,
- With maximum node degree of 3 and network diameter is 2(k – 1).
- The balanced binary tree is scalable, since it has a constant maximum node degree.

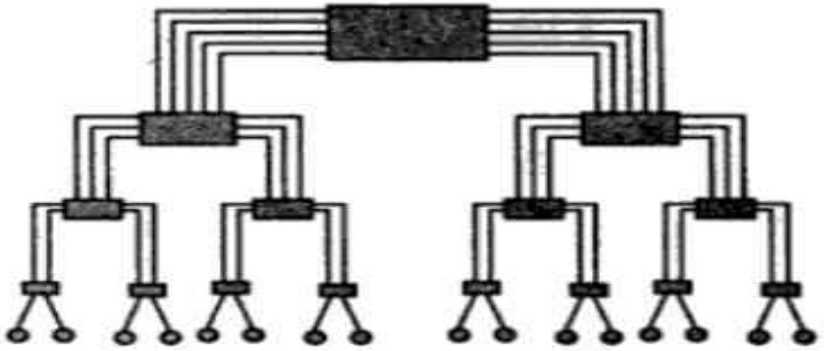


System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ Tree ,Star and Fat Tree

- A star is a two-level tree with a node degree d = N – 1 and a constant diameter of 2.
- A fat tree is a tree in which the number of edges between nodes increases closer to the root
- The edges represent communication channels (“wires”).

(b) Star



(c) Binary fat tree

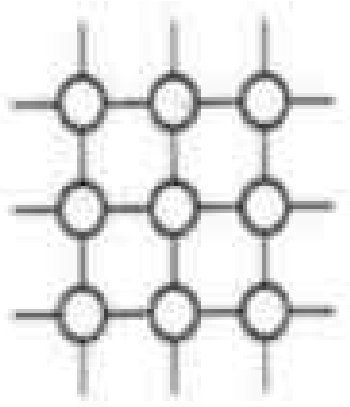
https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ Mesh and Torus

– Pure mesh:

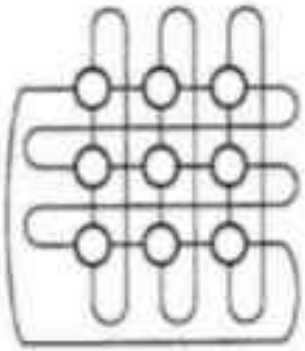
- $N = n^k$ nodes with links between each adjacent pair of nodes in a row or column (or higher degree).
- Interior node degree d = 2k, diameter = k (n – 1).
- This is not a symmetric network➔
  - degree and diameter doesn't exist as per the above requirement



https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

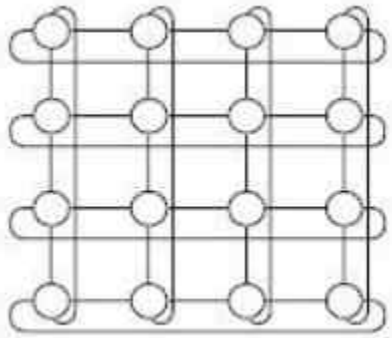
Static Connection Networks ➔ Mesh and Torus



– Illiac mesh (used in Illiac IV computer):

- Wraparound is allowed ➔ thus reducing the network diameter to about half that of the equivalent pure mesh.

– A torus mesh :



- Has ring connections in each dimension, and is symmetric.
- An n × n binary torus has node degree of 4 and a diameter of ➔

$$2 \times \lfloor n / 2 \rfloor$$

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ Systolic Array

- A systolic array is an arrangement of processing elements and communication links designed specifically to match the computation and communication requirements of a specific algorithm (or class of algorithms).
- This specialized character may yield better performance than more generalized structures, but also makes them more expensive, and more difficult to program.

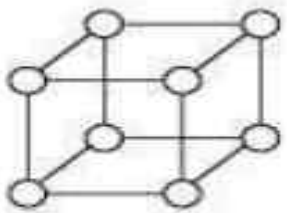
(d) Systolic array

https://hemanthrajhemu.github.io

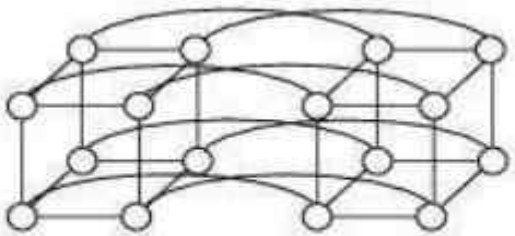
System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ Hypercubes

- A binary n-cube architecture with $N = 2^n$ nodes spanning along n dimensions, with two nodes per dimension.
- The hypercube scalability is poor, and packaging is difficult for higher-dimensional Hypercubes



(a) 3-cube



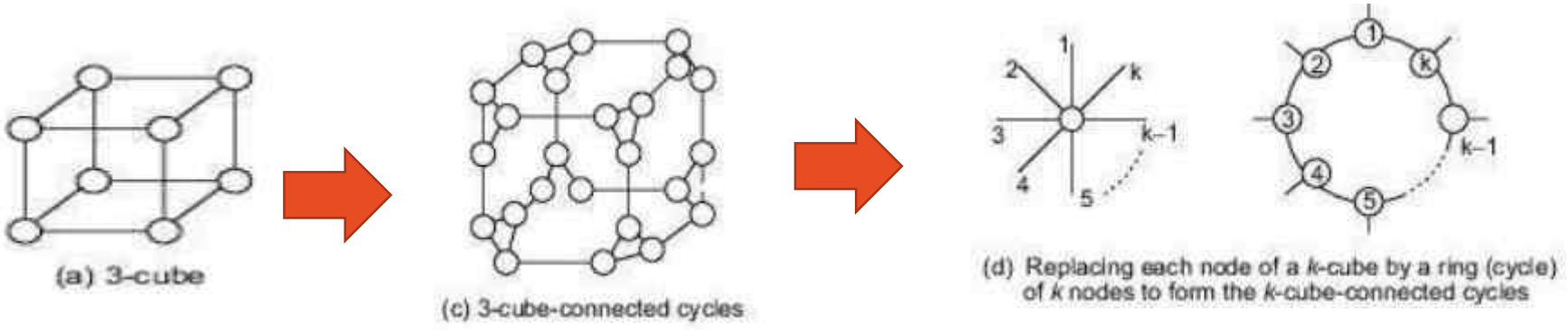
(b) A 4-cube formed by interconnecting two 3-cubes

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ Cube-connected Cycles

– 3-cube connected cycles (CCC) ➔ can be created from a 3-cube by replacing each corner nodes (vertex) of the 3 cube by a ring of 3 nodes.



(a) 3-cube

(c) 3-cube-connected cycles

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

– k-cube connected cycles (CCC) ➔ can be created from a k-cube by replacing each vertex of the k dimensional hypercube by a ring of k nodes.

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Static Connection Networks ➔ Network Throughput

- Network throughput:- number of messages a network can handle in a unit time interval.
- One way to estimate is to calculate the maximum number of messages that can be present in a network at any instant (its capacity) ➔ Throughput usually is some fraction of its capacity.
- A hot spot is a pair of nodes that accounts for a disproportionately large portion of the total network traffic (**possibly causing congestion**).
- **Hot spot throughput** is maximum rate at which messages can be sent between two specific nodes.

System Interconnect Architectures
Network Properties and Routing

Dynamic Connection Networks ➔

- Dynamic connection networks can implement all communication patterns based on program demands.
- In increasing order of cost and performance, these include
  - **Bus systems**
  - **Multistage interconnection networks**
  - **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.**

System Interconnect Architectures
Network Properties and Routing

Dynamic Connection Networks ➔ Bus Systems

- A bus system (contention bus, time-sharing bus):➔
  - Has a collection of wires and connectors
  - Multiple modules (processors, memories, peripherals, etc.) which connect to the wires
  - Data transactions between pairs of modules
- 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.

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Dynamic Connection Networks ➔ Bus Systems

- A bus is the simplest type of dynamic interconnection networks.
- It constitutes a common data transfer path for many devices.
- Depending on the type of implemented transmissions we have serial busses and parallel busses.
- The devices connected to a bus can be processors, memories, I/O units, as shown in the figure below.

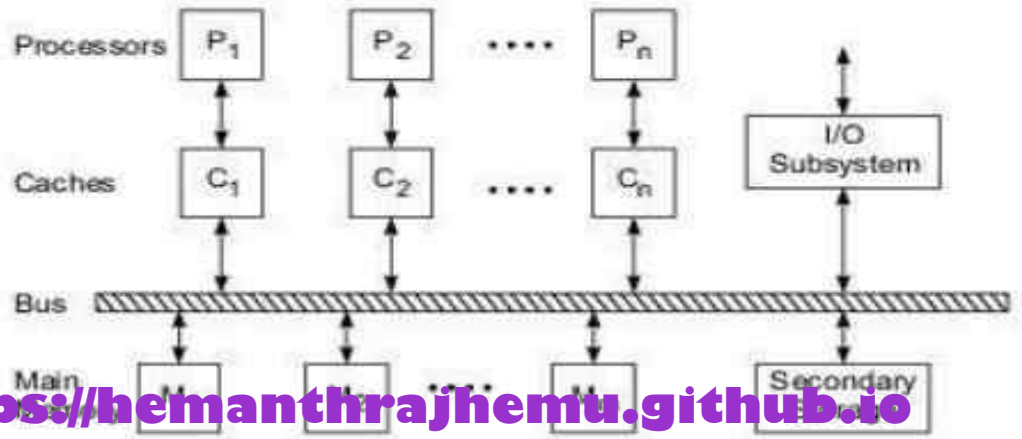
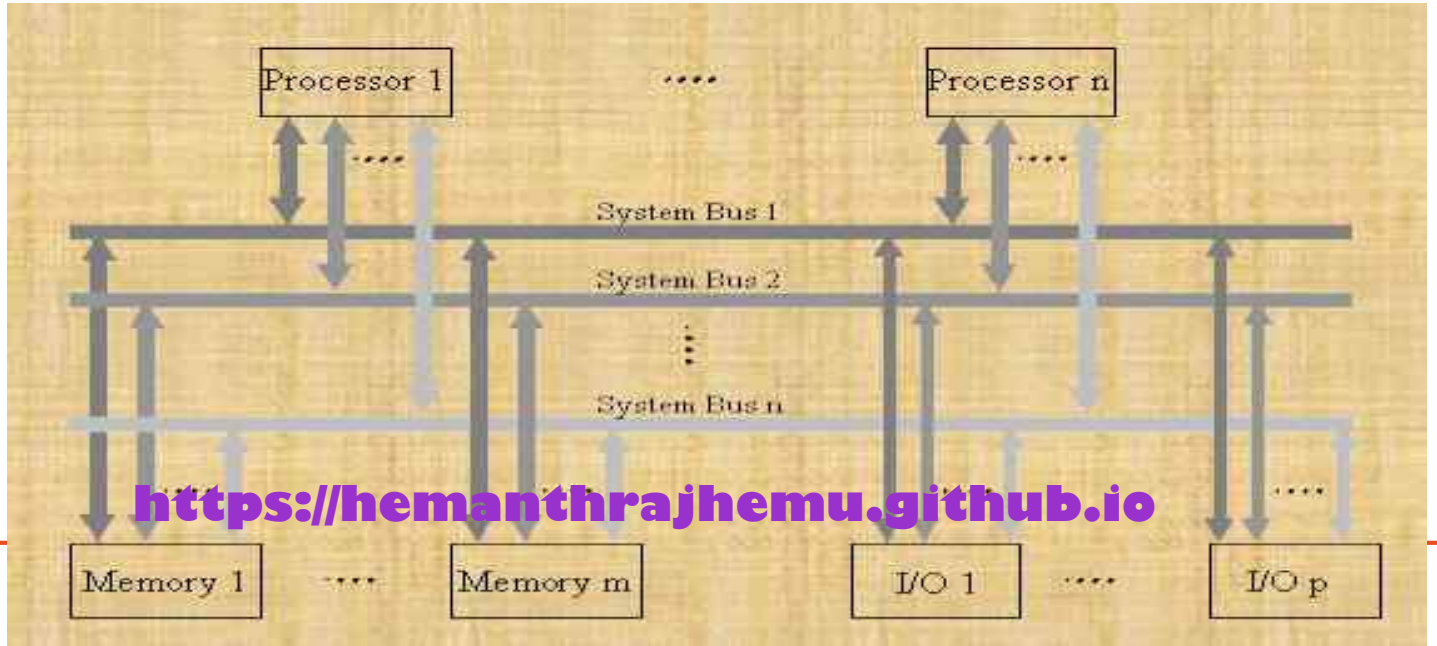


Fig. 2.22 A bus-connected multiprocessor system.

System Interconnect Architectures
Network Properties and Routing

Dynamic Connection Networks ➔ Bus Systems

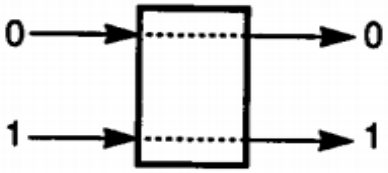
- The throughput of the network based on a bus ➔ can be increased by the use of a multibus network shown in the figure below.
- In this network, processors connected to the busses can transmit data in parallel (**one for each bus**) and **many processors can read data from many busses at a time.**



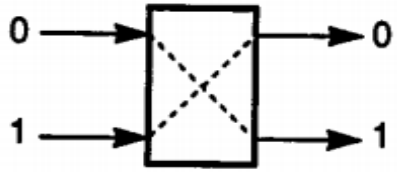
System Interconnect Architectures
Network Properties and Routing

Dynamic Connection Networks ➔ Switch Modules

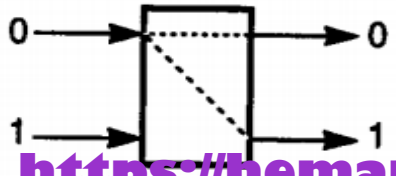
- An a × b switch module has a inputs and b outputs.
- A binary switch has a = b = 2.
  - **It is not necessary for a = b, but usually a = b = 2^k , for some integer k.**



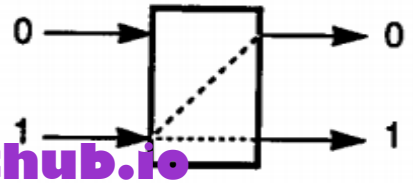
(a) Straight



(b) Crossover



(c) Upper broadcast

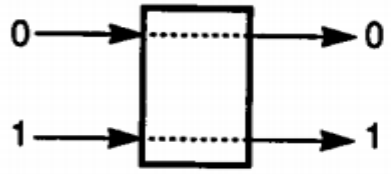


(d) Lower broadcast

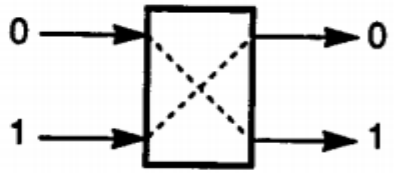
System Interconnect Architectures
Network Properties and Routing

Dynamic Connection Networks ➔ Switch Modules

- In general, any input can be connected to one or more of the outputs.
- However, multiple inputs may not be connected to the same output. When only one-to-one mappings are allowed, the switch is called a crossbar switch.

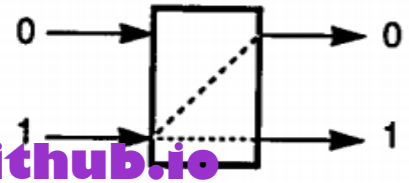


(a) Straight



(b) Crossover

(c) Upper broadcast



(d) Lower broadcast

System Interconnect Architectures
Network Properties and Routing

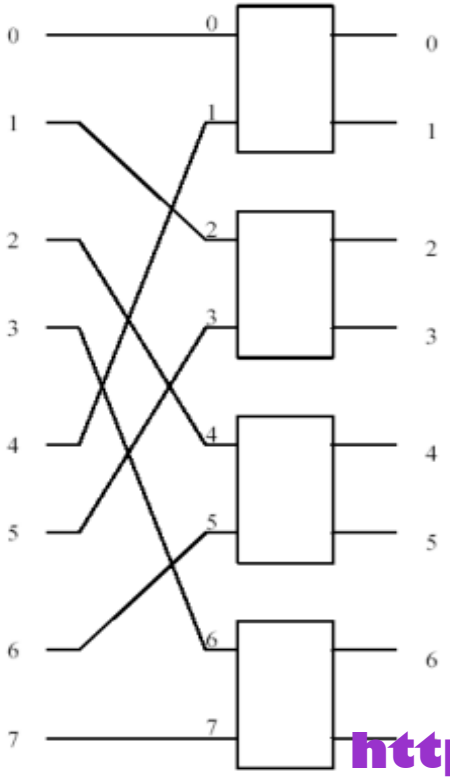
Dynamic Connection Networks ➔ Multistage Interconnection Networks

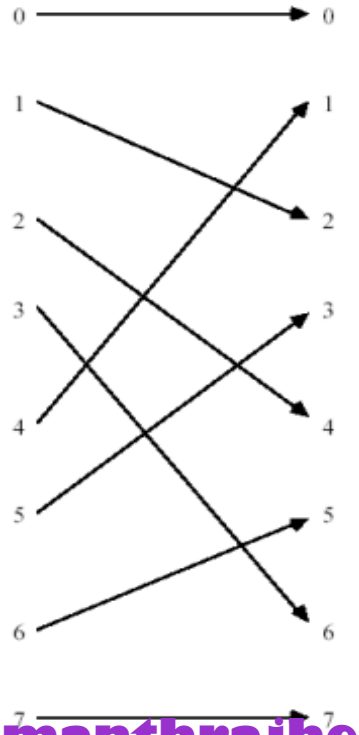
- In general, any multistage network is comprised of a ➔
  - **Collection of a × b switch modules and fixed network modules.**
  - The a × b switch modules are used to provide **variable permutation or other reordering of the inputs, which are then further reordered by the fixed network modules.**
- A generic multistage network consists of a sequence **alternating dynamic switches (with relatively small values for a and b) with static networks (with larger numbers of inputs and outputs). The static networks are used to implement interstage connections (ISC).**

System Interconnect Architectures
Network Properties and Routing

Dynamic Connection Networks ➔ Multistage Interconnection Networks

Single-stage networks:➔





- Perfect shuffle operation:➔ cyclic shift 1 place left, eg 101 --> 011

https://hemanthrajhemu.github.io

System Interconnect Architectures
Network Properties and Routing

Dynamic Connection Networks ➔ Multistage Interconnection Networks

- Single-stage networks:➔Since a single-stage network was not sufficient to get the desired output, a multistage is built.
- **Omega Network** : An example of a multistage network is the **omega network**.
- The capability of single stage networks are limited ➔
  - **But if we cascade enough of them together, they form a completely connected MIN (Multistage Interconnection Network).**
- Switches can perform their own routing or can be controlled by a central router

https://hemanthrajhemu.github.io

System Interconnect Architectures

Network Properties and Routing

Dynamic Connection Networks ➔ Multistage Interconnection Networks

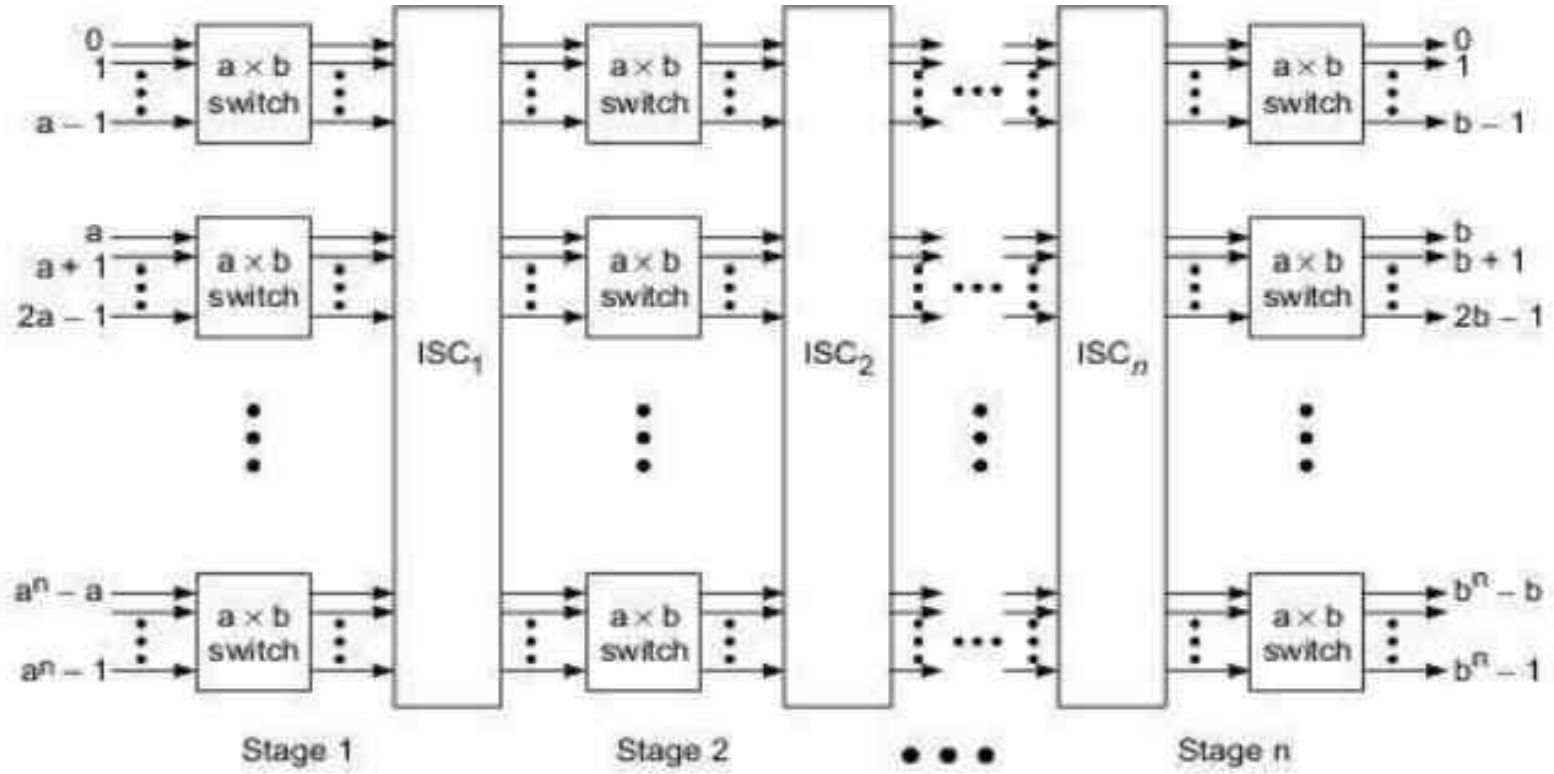


Fig. 2.23 A generalized structure of a multistage interconnection network (MIN) built with a × b switch modules and interstage connection patterns $ISC_1, ISC_2, \ldots, ISC_n$

System Interconnect Architectures

Network Properties and Routing

Dynamic Connection Networks ➔ Multistage Interconnection Networks

This type of networks can be classified into the following four categories:

- **Nonblocking :** A network is called strictly nonblocking if it can connect any idle input to any idle output regardless of what other connections are currently in process
- **Rearrangeable nonblocking :** In this case a network should be able to establish all possible connections between inputs and outputs by rearranging its existing connections.
- **Blocking interconnection:** A network is said to be blocking if it can perform many, but not all, possible connections between terminals. Example: the Omega network

System Interconnect Architectures

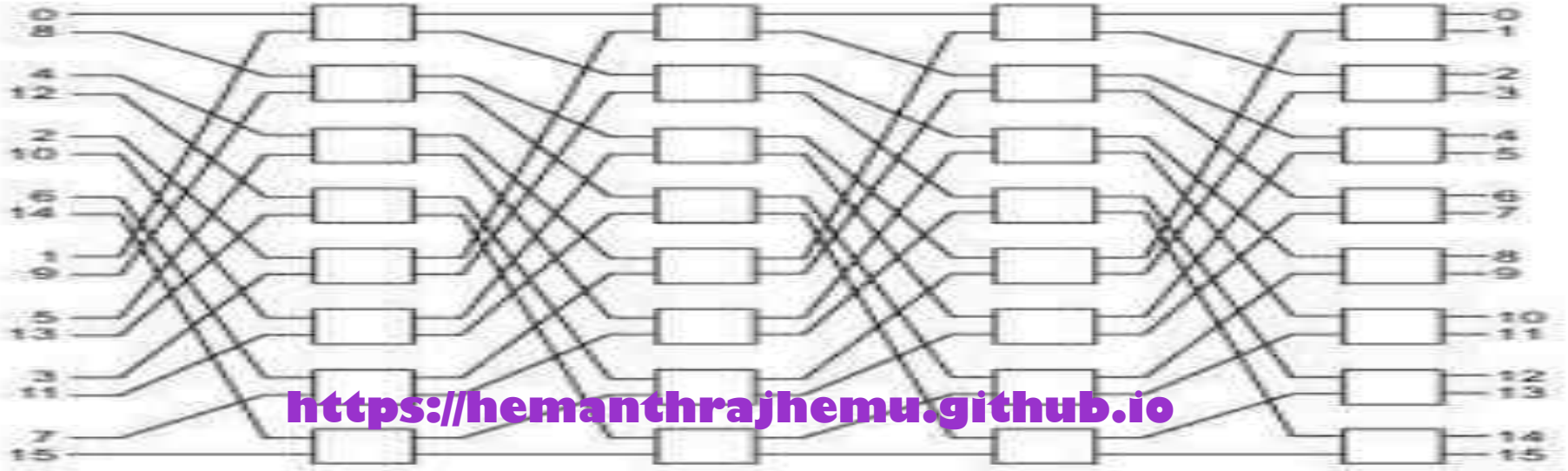
Network Properties and Routing

Dynamic Connection Networks ➔ Multistage Interconnection Networks

Omega Networks:

A 2 × 2 switch can be configured for Straight-through, Crossover, Upper broadcast (upper input to both outputs), Lower broadcast (lower input to both outputs)

– With four stages of eight 2 × 2 switches, and a static perfect shuffle for each of the four ISCs➔A 16 by 16 Omega network can be constructed (but not all permutations are possible).



(e) 16 × 16 Omega network

System Interconnect Architectures

Network Properties and Routing

Dynamic Connection Networks ➔ Multistage Interconnection Networks

Baseline Networks:

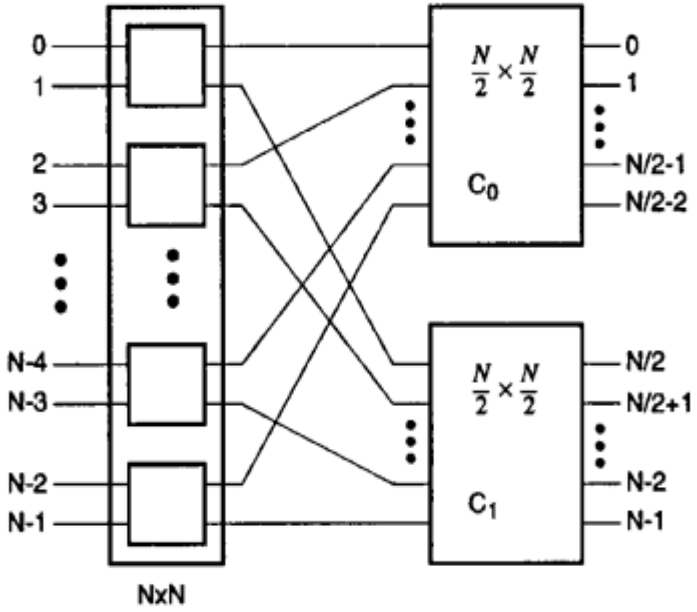
- A baseline network can be shown to be topologically equivalent to other networks (including Omega), and has a simple recursive generation procedure.
- Stage k (k = 0, 1, …) is an m × m switch block (where m = N / 2k) composed entirely of 2 × 2 switch blocks, each having two configurations: straight through and crossover.
- **The network can be generated recursively**
- **The first stage N × N, the second (N/2) × (N/2).**
- Networks are topologically equivalent if one network can be easily reproduced from the other networks by simply rearranging nodes at each stage

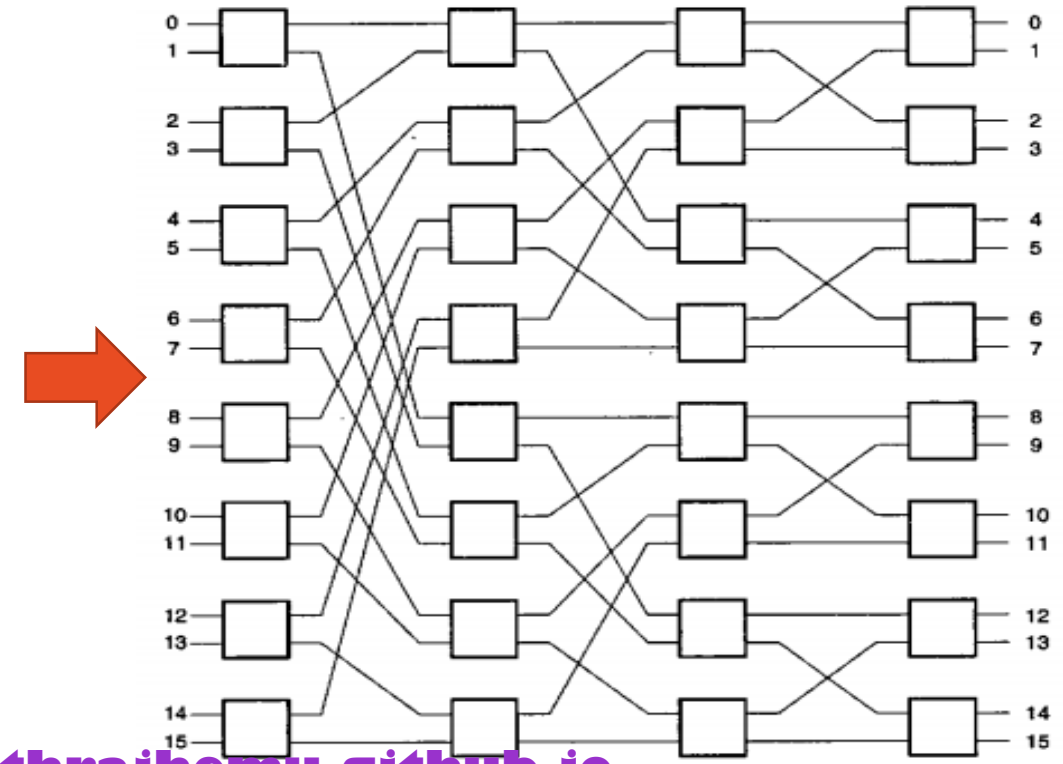
System Interconnect Architectures

Network Properties and Routing

Dynamic Connection Networks ➔ Multistage Interconnection Networks

Baseline Networks:





https://hemanthrajhemu.github.io

System Interconnect Architectures

Network Properties and Routing

Dynamic Connection Networks ➔ Multistage Interconnection Networks

Crossbar Networks:

- A crossbar switch has a number of input and output data pins and a number of control pins.
- In response to control instructions set to its control input,➔ the crossbar switch implements a stable connection of a determined input with a determined output.
- A m × n crossbar network➔
  - **With m processors and n memories, one processor may be able to generate requests for multiple memories in sequence; thus several switches might be set in the same row.**
  - **For m × m interprocessor communication, each PE is connected to both an input and an output of the crossbar; only one switch in each row and column can be turned on simultaneously. Additional control processors are used to manage the crossbar itself.**

https://hemanthrajhemu.github.io

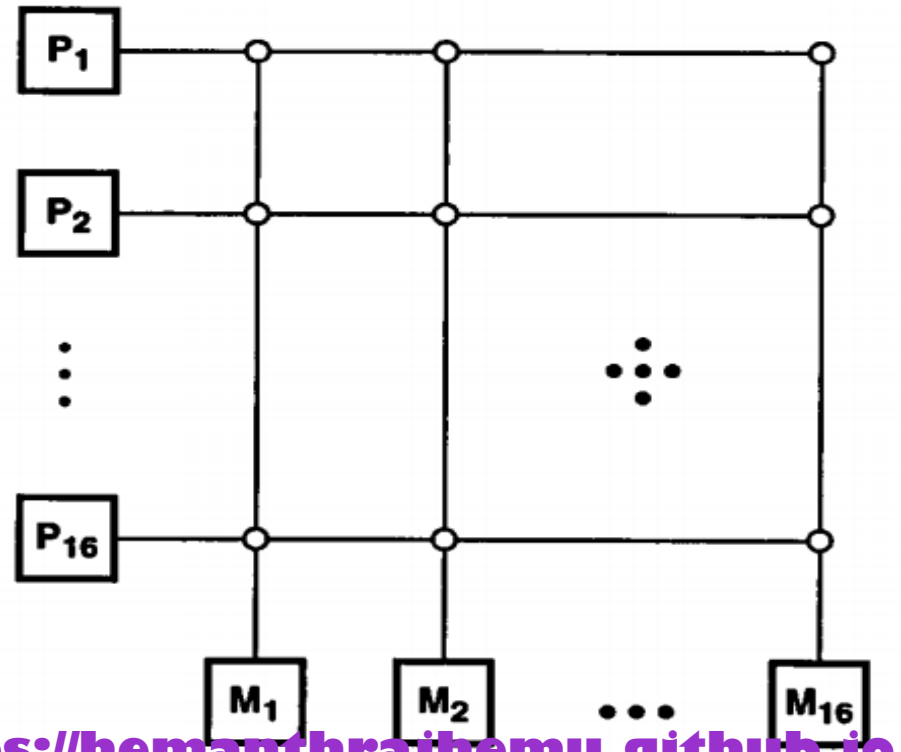
System Interconnect Architectures

Network Properties and Routing

Dynamic Connection Networks ➔ Multistage Interconnection Networks

Crossbar Networks:

– Each junction is a switching component – connecting the row to the column.



Principles of Scalable Performance

Chapter-3

This chapter includes:

- Performance measures
- Speedup laws
- Scalability principles
- Scaling up vs. scaling down

Principles of Scalable Performance
Chapter-3

Performance metrics and measures:-

- Parallelism profiles
- Asymptotic speedup factor
- System efficiency, utilization and quality
- Standard performance measures

Principles of Scalable Performance

Parallelism Profile in Programs

Parallelism Profile in Programs:

- Degree of parallelism ➔ Reflects the matching of software and hardware parallelism
- Under this topic we study:
  - **Degree of parallelism**
  - **Average parallelism**
  - **Available parallelism**
  - **Asymptotic speedup factor**

https://hemanthrajhemu.github.io

Principles of Scalable Performance
Parallelism Profile in Programs

Degree of parallelism(DOP): ➔ Reflects the matching of software and hardware parallelism

- **DOP is defined as:➔**For each period, the number of processors used to execute a program in parallelism profile
  - **This is a discrete time function only for non-negative integer values**
- **Parallelism profile of a given program ➔** is a plot of the DOP as a function of time Ideally have unlimited resources

https://hemanthrajhemu.github.io

Principles of Scalable Performance
Parallelism Profile in Programs

Degree of parallelism(DOP): **Factors affecting parallelism profiles:**

- Algorithm structure
- Program optimization
- Resource utilization
- Run-time conditions
- Realistically limited by ➔ **# of available processors, memory, and other non-processor resources**

Principles of Scalable Performance
Parallelism Profile in Programs

Average parallelism:➔Its variables on which it depends-

- **n** ➔ Homogeneous Processors
- **m** ➔ Maximum Parallelism in a Profile
  - In ideal case, n >> m
- **Δ** ➔ Computing Capacity of a Single Processor
  - Depends on → execution rate such as MIPS
  - no overhead
- **DOP = i** ➔ i processors busy during an observation period

https://hemanthrajhemu.github.io

Principles of Scalable Performance

Parallelism Profile in Programs

Average parallelism:➔ Total amount of work **W (instructions or computations)** performed is proportional to the area under the profile curve

$$W = \Delta \int_{t1}^{t2} DOP(t)dt$$

This integral is often computed by the following discrete summation

$$W = \Delta \sum_{i=1}^{m} i \cdot t_i$$

t_i → Is the total amount of time that DOP=i

$$\sum_{i=1}^{m} t_i = t_2 - t_1$$

https://hemanthrajhemu.github.io

Principles of Scalable Performance

Parallelism Profile in Programs

Average parallelism:➔Is computed by

$$A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOP(t)dt$$

In discrete form, we get

$$A = \left(\sum_{i=1}^{m} i \cdot t_i \right) / \left(\sum_{i=1}^{m} t_i \right)$$

https://hemanthrajhemu.github.io

Principles of Scalable Performance

Parallelism Profile in Programs

Example: →Parallelism profile and average parallelism

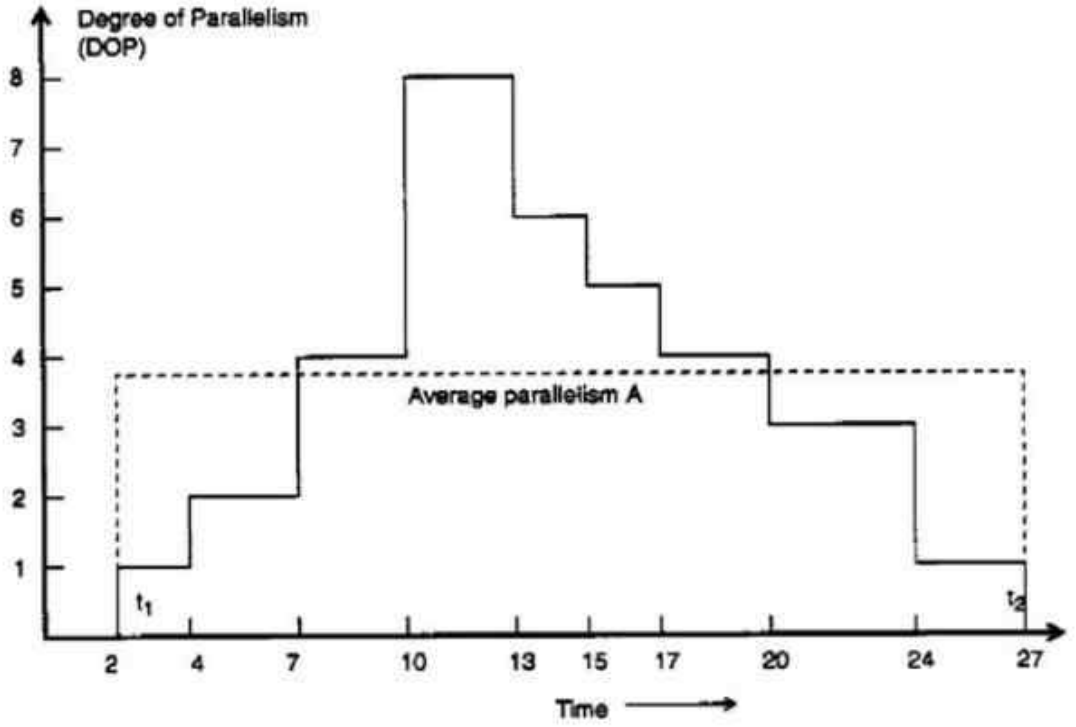


Figure 3.1 Parallelism profile of a divide-and-conquer algorithm.

Principles of Scalable Performance

Parallelism Profile in Programs

Asymptotic speedup ➔ the amount of work executed with DOP=i

For i=1(Single processor-sequential)

$$T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta}$$

W.R.T
Response Time

For i=∞(Infinite number of processors)

$$T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i\Delta}$$

https://hemanthrajhemu.github.io

Principles of Scalable Performance

Parallelism Profile in Programs

Asymptotic speedup ➔ S_∞ ➔ is defined as the ratio of T(1) to T(∞)

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} W_i / i}$$

$$= A \text{ in the ideal case}$$

In ideal case,
n=∞
n >> m

https://hemanthrajhemu.github.io

Principles of Scalable Performance

Mean Performance

Performance measures:

- Consider n processors executing m programs in various modes
- Want to define the mean performance of these multimode computers:
  - **Arithmetic mean performance**
  - **Harmonic mean performance**

Principles of Scalable Performance

Mean Performance

Arithmetic mean performance:

Let $\{R_i\}$ be the execution rate of programs i=1,2,3,….,m.

Arithmetic mean execution rate is defined as:➔

$$R_a = \sum_{i=1}^{m} R_i / m$$

Arithmetic mean execution rate (assumes equal weighting)

Ex1:→ Average time to complete a program ➔(10 secs, 20 secs, 15 secs) ➔ 15 secs •

Ex2:→Consider the scores (1, 2, 2, 2, 3, 9). The arithmetic mean is 3.17, but five out of six scores are below this.

https://hemanthrajhemu.github.io

Principles of Scalable Performance

Mean Performance

Arithmetic mean performance:

Let $\{R_i\}$ be the execution rate of programs i =1,2,3,….,m

Note: If the programs are weighted with a distribution

$\pi=\{f_i \mid i=1,2,.....m\}$

Weighted Arithmetic mean execution rate is defined as:➔

By replacing 1/m with f_i

$$R_a^* = \sum_{i=1}^{m} (f_i R_i)$$ Weighted arithmetic mean execution rate

-proportional to the sum of the inverses of execution times

Principles of Scalable Performance

Mean Performance

Harmonic mean performance:

$$T_i = 1/R_i$$

Mean execution time per instruction
For program i

$$T_a = \frac{1}{m}\sum_{i=1}^{m} T_i = \frac{1}{m}\sum_{i=1}^{m} \frac{1}{R_i}$$

Arithmetic mean execution time per instruction

https://hemanthrajhemu.github.io

Principles of Scalable Performance

Mean Performance

Harmonic mean performance:

$$R_h = 1/T_a = \frac{m}{\sum_{i=1}^{m}(1/R_i)}$$ Harmonic mean execution rate

$$R_h^* = \frac{1}{\sum_{i=1}^{m}(f_i/R_i)}$$ Weighted harmonic mean execution rate

-corresponds to total # of operations divided by the total time (closest to the real performance)

Principles of Scalable Performance

Mean Performance

Harmonic mean Speedup:

- Ties the various modes of a program to the number of processors used
- Program is in *mode i* if *i* processors used
- Sequential execution time $T_1 = 1/R_1 = 1$

$$S = T_1 / T^* = \frac{1}{\sum_{i=1}^{n} f_i / R_i}$$

Principles of Scalable Performance
Mean Performance-Amdahl's Law

Amdahl's Law

- Assume $R_i = i$, $w = (\alpha, 0, 0, \ldots, 1-\alpha)$
- System is either sequential, with probability α, or fully parallel with prob. $1-\alpha$

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

- Implies $S \to 1/\alpha$ as $n \to \infty$

https://hemanthrajhemu.github.io

Principles of Scalable Performance

Mean Performance - Amdahl's Law

Speedup Performance

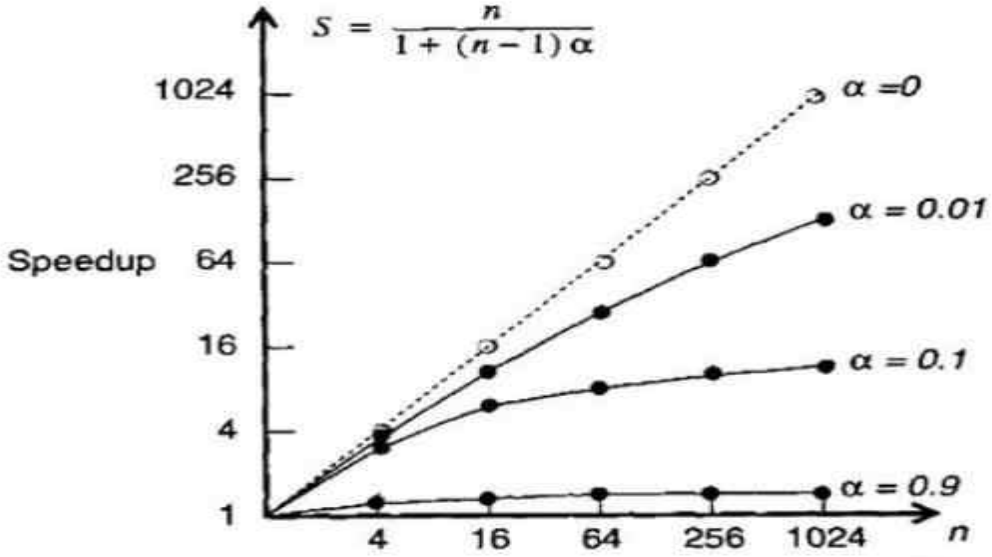


Figure 3.3 Speedup performance with respect to the probability distribution $\pi = (\alpha, 0, ..., 0, 1-\alpha)$ where α is the fraction of sequential bottleneck

https://hemanthrajhemu.github.io

Principles of Scalable Performance

Efficiency, Utilization and Quality

System Efficiency

- $O(n)$ is the total # of unit operations
- $T(n)$ is execution time in unit time steps
- $T(n) < O(n)$ and $T(1) = O(1)$

$$S(N) = T(1) / T(n)$$

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

https://hemanthrajhemu.github.io

Principles of Scalable Performance
Efficiency, Utilization and Quality

Redundancy and Utilization

- Redundancy signifies the extent of matching software and hardware parallelism

$$R(n) = O(n) / O(1)$$

- Utilization indicates the percentage of resources kept busy during execution

$$U(n) = R(n)E(n) = \frac{O(n)}{nT(n)}$$

https://hemanthrajhemu.github.io

Principles of Scalable Performance
Efficiency, Utilization and Quality

Quality of Parallelism

- Directly proportional to the speedup and efficiency and inversely related to the redundancy
- Upper-bounded by the speedup $S(n)$

$$Q(n) = \frac{S(n)E(n)}{R(n)} = \frac{T^3(1)}{nT^2(n)O(n)}$$

https://hemanthrajhemu.github.io

Principles of Scalable Performance
Efficiency, Utilization and Quality

Example of Performance

- Given $O(1) = T(1) = n^3$, $O(n) = n^3 + n^2 \log n$, and $T(n) = 4\ n^3/(n+3)$

$$\text{Speedup } S(n) = (n+3)/4$$
$$\text{Efficiency } E(n) = (n+3)/(4n)$$
$$\text{Redundancy } R(n) = (n + \log n)/n$$
$$\text{Utilization } U(n) = (n+3)(n + \log n)/(4n^2)$$
$$\text{Quality } Q(n) = (n+3)^2 / (16(n + \log n))$$

https://hemanthrajhemu.github.io

Principles of Scalable Performance
Efficiency, Utilization and Quality

Standard Performance Measures

- MIPS and Mflops
  - Depends on instruction set and program used
- Dhrystone results
  - Measure of integer performance
- Whestone results
  - Measure of floating-point performance
- TPS and KLIPS ratings
  - Transaction performance and reasoning power

Mflops → mega floating-point operations per second

KLIPS – kilo logical inferences per second → used to indicate the reasoning power of an AI machine

Principles of Scalable Performance

Parallel Processing Applications

Few applications listed:-

- Drug design
- High-speed civil transport
- Ocean modelling
- Ozone depletion research
- Air pollution
- Digital anatomy

https://hemanthrajhemu.github.io

Parallel Processing Applications

Massively parallel processing has become one of the challenges in supercomputer applications.

Massive Parallelism for Grand Challenges:➔Which varies with time.

- **Any machine having hundreds and thousands of processors is a Massively Parallel Processing (MPP) System.**
- **Here we will discuss the grand challenges identified in the U.S. High Performance Computer and Communication (HPCC) program.**

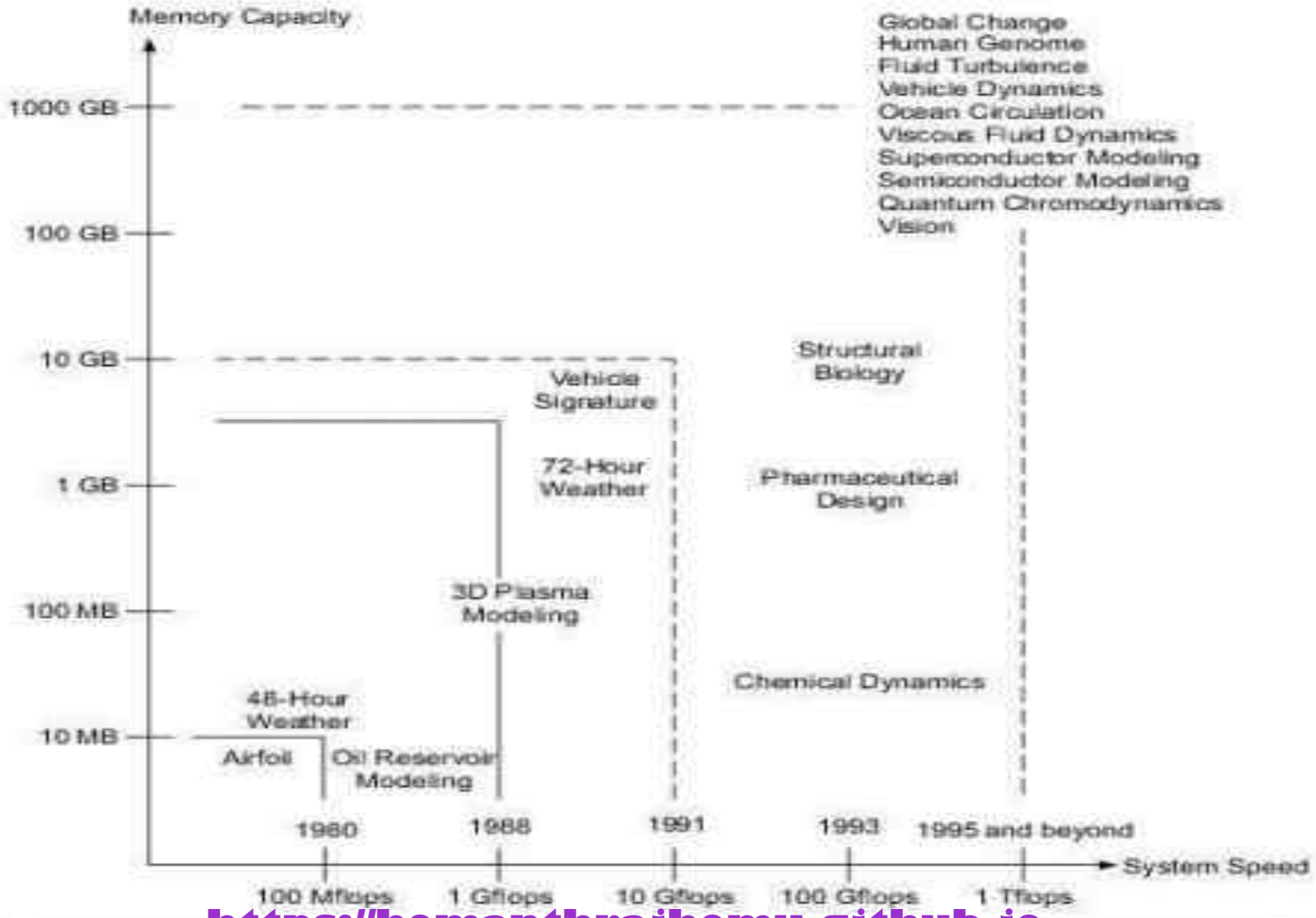


Fig. 3.5 Grand challenge requirements in computing and communications (Courtesy of U.S. High-Performance Computing and Communication Program, 1992)

Parallel Processing Applications

Exploiting Massive Parallelism:

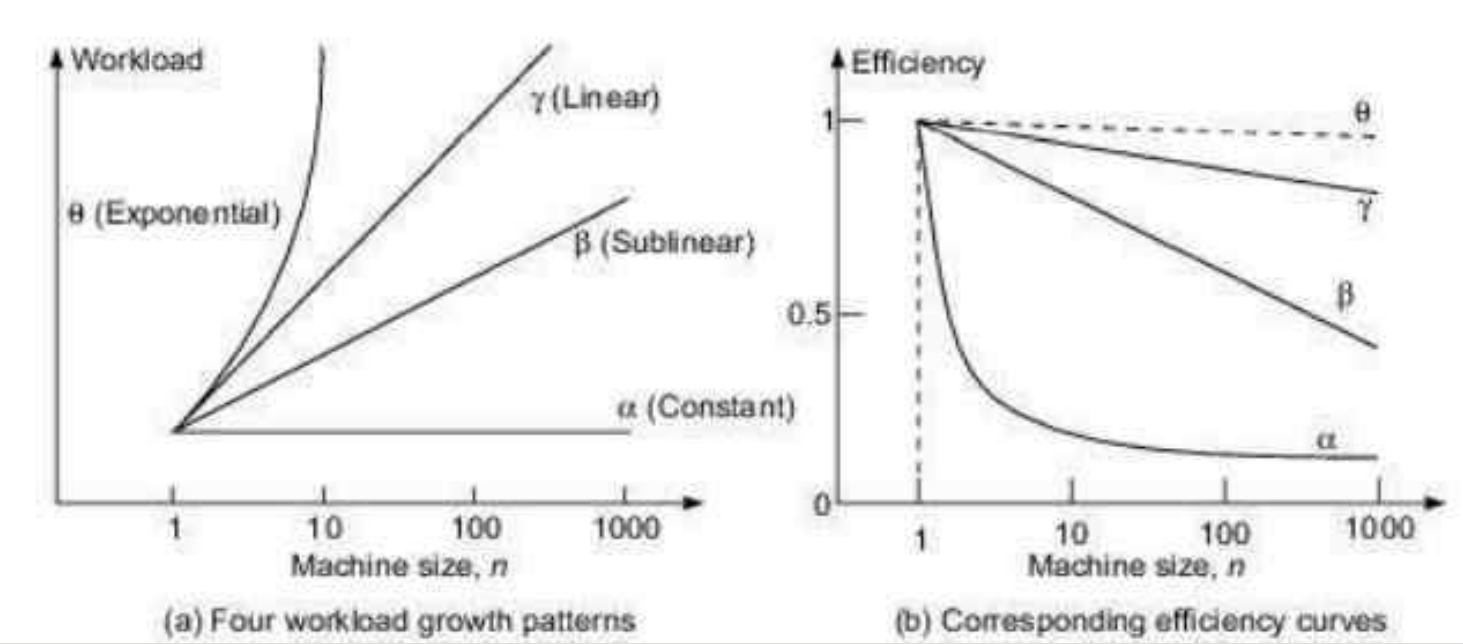
- **Instruction level parallelism:**
  - Very limited
  - Very few processors exists →which can execute 2 instructions in one cycle.
  - Depends upon complier, OS incapability's and program flow built in modern computers.
- **Data parallelism**
  - Higher than Instruction level parallelism.
  - Deals with large array/operands
- **On Message passing multicomputer → Parallelism is more scalable than shared multiprocessor**

https://hemanthrajhemu.github.io

Parallel Processing Applications (Application Models)

Application Models for Parallel Processing:

– Keeping the workload as constant **(α)** ➔ Efficiency E decreases rapidly as Machine Size n increases

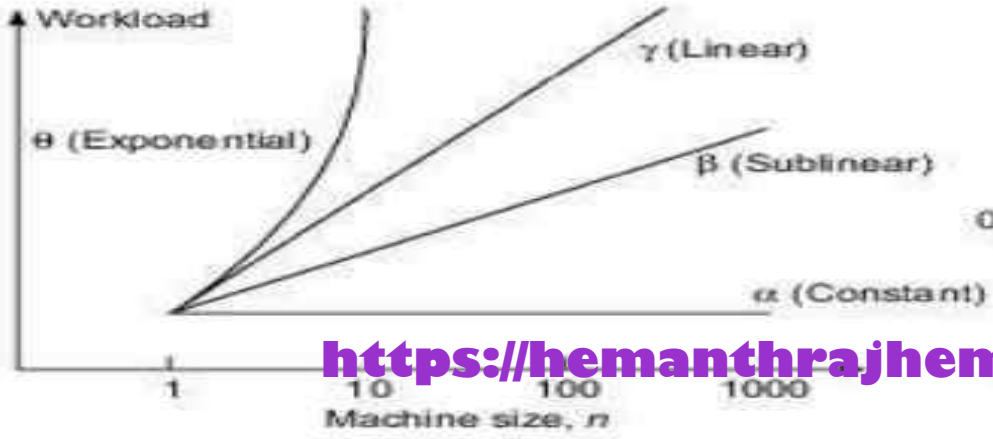


– Because: **Overhead h increases faster than machine size**

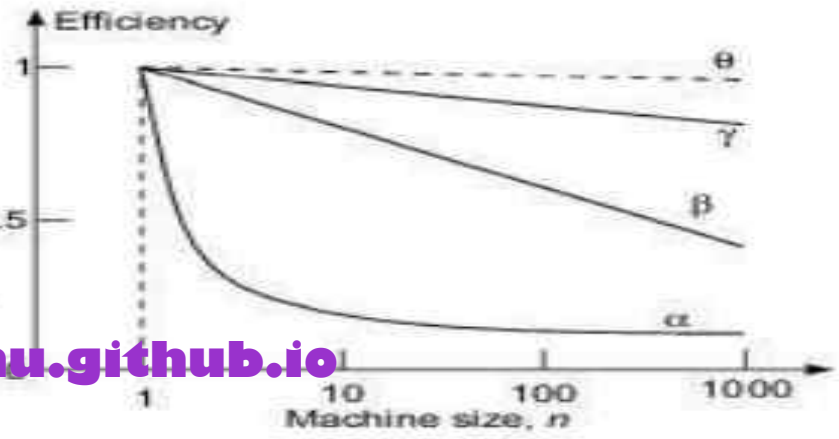
Parallel Processing Applications (Application Models)

Application Models for Parallel Processing:

- To maintain the efficiency at desired level ➔ **we has to increase the machine size and program size proportionally** ➔ Know as Scalable computers/Scaled problems
- Workload with linear function of n (**represented as γ**)-linear scalability in problem size.
- If linear not possible we can go to Sublinear **(β)** by keeping some constant value
- If there is enormous growth in workload **(θ)** ➔ **system is poorly scalable** ➔**To keep constant efficiency** ➔ **Exceeds Memory and I/O limits**



(a) Four workload growth patterns



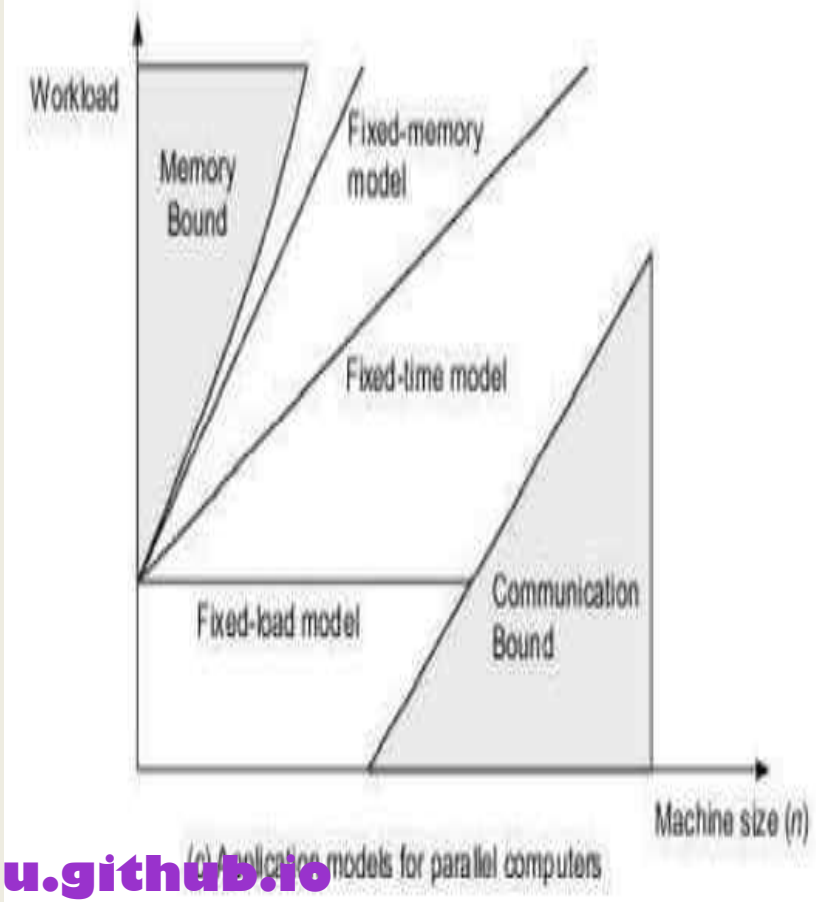
(b) Corresponding efficiency curves

Parallel Processing Applications (Application Models)

Application Models for Parallel Processing: 3 Models → Depends on limited memory, limited latency tolerance , limited I/O bandwidth

The models are:

1. **Fixed load model:**
   - Corresponds to constant workload (α)
   - The use is limited by the communication bound.
2. **Fixed time model:**
   - Demands constant execution time
   - Linear workload growth refers to this model (γ)
3. **Fixed memory model:**
   - Is limited by memory bound
   - Corresponds to workload curve between γ and θ



(c) Application models for parallel computers

https://hemanthrajhemu.github.io

Parallel Processing Applications
(Scalability of Parallel Algorithms)

Algorithmic characteristics:-

- Parallel algorithms are specially designed for parallel computers
- Important characteristics of parallel algorithms which are machine implementable are:
  - Deterministic versus Non-Deterministic
  - Computational granularity
  - Parallelism Profile
  - Communicational patterns and synchronization requirements
  - Uniformity of the operations
  - Memory requirement and data structures.

Parallel Processing Applications
(Scalability of Parallel Algorithms)

Isoefficiency:- It is the concept ➔

- **Relating workload (w = w(s) // s-Problem size) to machine size n ➔ needed to maintain a fixed efficiency ➔For a parallel algorithm on a parallel computer**
- **Let h ➔ Communication overhead of the algorithm implemented**

  ➔ h=h(s , n)
- **Efficiency is given by : -**

$$E = \frac{w(s)}{w(s) + h(s,n)}$$

- **With Fixed problem size ➔ Efficiency decreases as n increases**
- **With Fixed machine size ➔ Efficiency increases as s increases with n fixed**

https://hemanthrajhemu.github.io

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]

https://hemanthrajhemu.github.io

Speedup Performance Laws

Amdahl's Law (1967)

For a given problem size, the speedup does not increase linearly as the number of processors increases. In fact, the speedup tends to become saturated.

This is a consequence of Amdahl's Law.

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

Completely sequential

Completely parallel

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

https://hemanthrajhemu.github.io

Speedup Performance Laws

Amdahl's Law

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

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

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

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

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

Speedup Performance Laws

Amdahl's Law

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

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

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.*

https://hemanthrajhemu.github.io

Speedup Performance Laws

Amdahl's Law

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

The total work load or the problem size is fixed

Thus, execution time decreases with increasing no. of processors

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

Speedup Performance Laws

Amdahl's Law

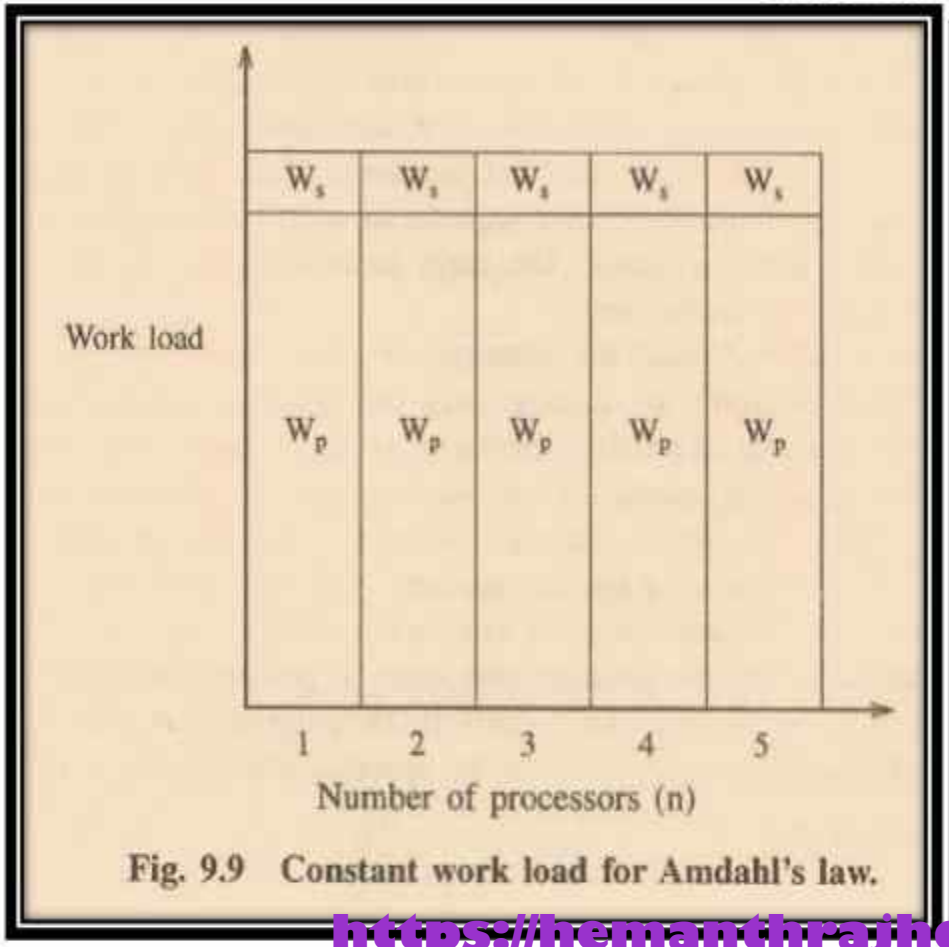
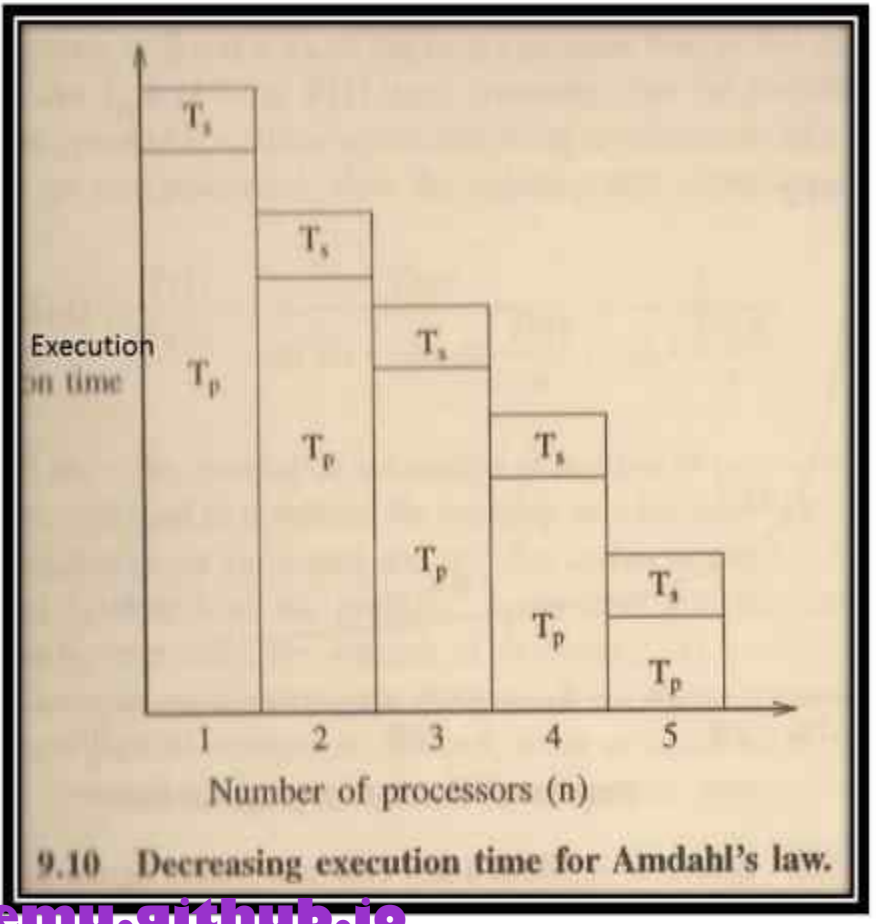


Fig. 9.9 Constant work load for Amdahl's law.



9.10 Decreasing execution time for Amdahl's law.

Speedup Performance Laws

Gustafson's Law (1988)

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

According to Gustafson's Law,

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

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

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

https://hemanthrajhemu.github.io

Speedup Performance Laws

Gustafson's Law

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

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

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

Then the speedup with n processors is:

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

https://hemanthrajhemu.github.io

Speedup Performance Laws

Gustafson's Law

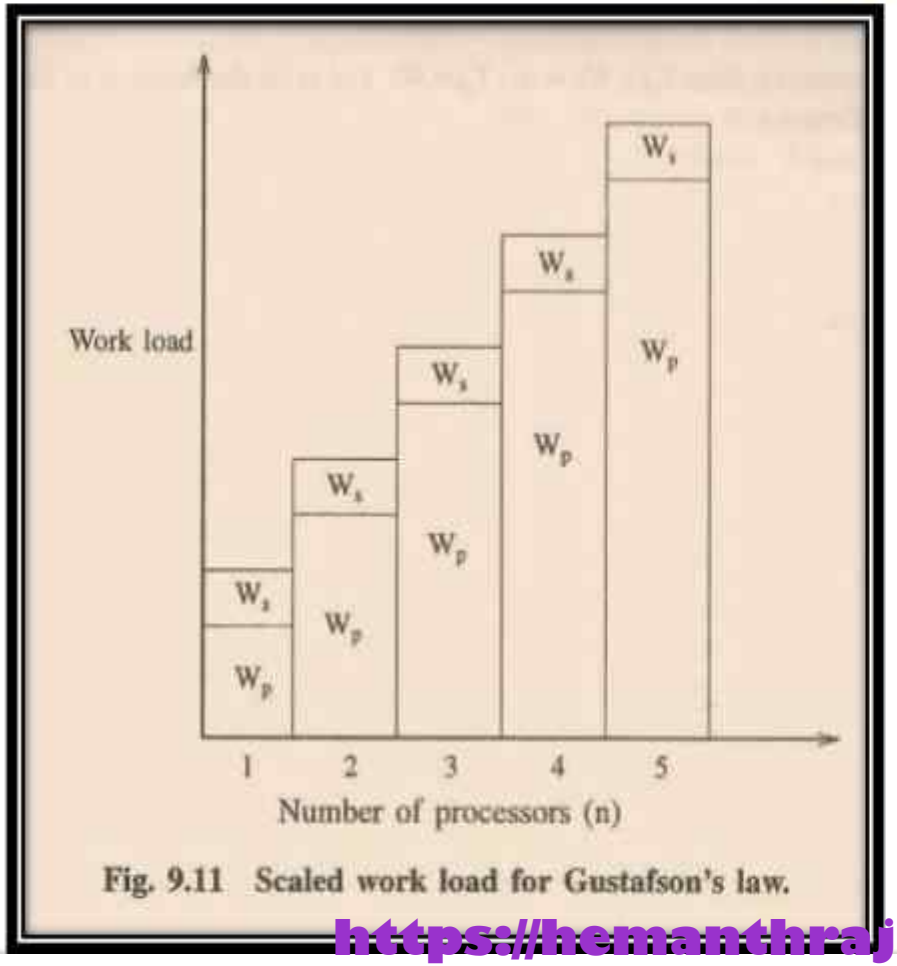


Fig. 9.11 Scaled work load for Gustafson's law.

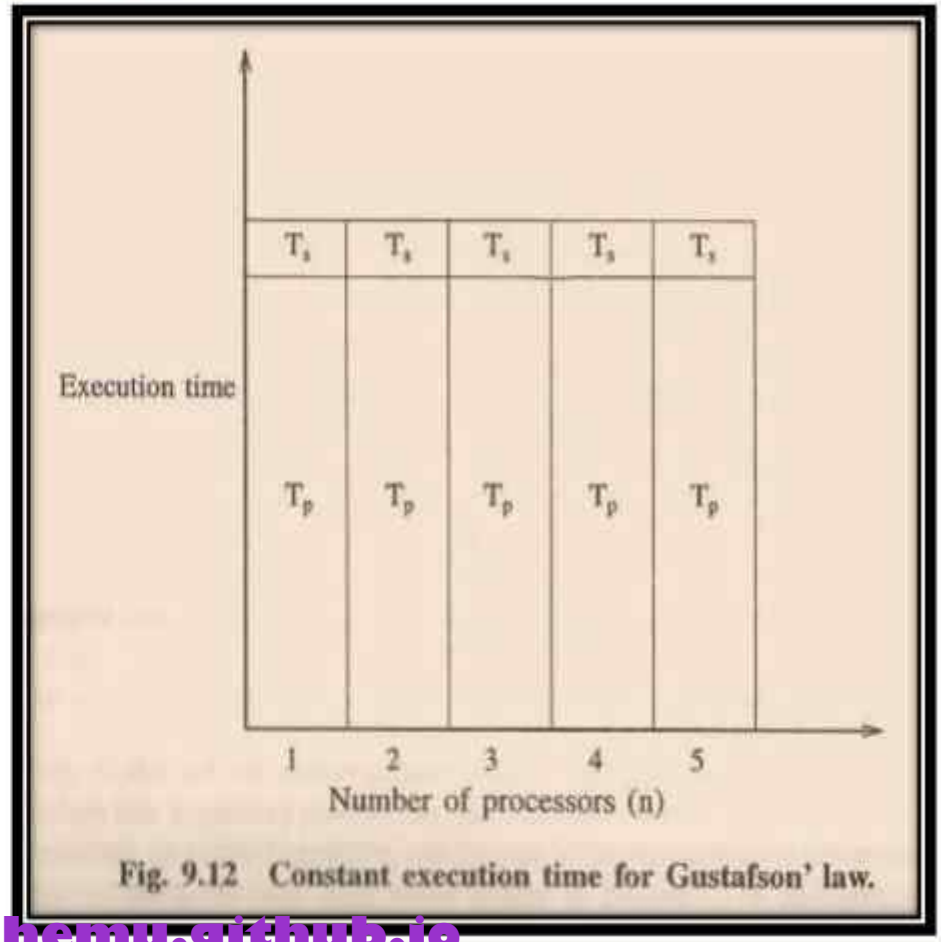


Fig. 9.12 Constant execution time for Gustafson' law.

Speedup Performance Laws

Gustafson's Law

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

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

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

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

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

https://hemanthrajhemu.github.io

Speedup Performance Laws

Sun & Ni's Law (1993)

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

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

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

providing *higher speedup,*

Higher efficiency, and

Better resource (processor & memory) utilization

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

https://hemanthrajhemu.github.io

Speedup Performance Laws

Sun & Ni's Law

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

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

G(n)→Reflects the increase in workload has memory increases n times.

Assumptions made while deriving the above expression:

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

https://hemanthrajhemu.github.io

Speedup Performance Laws

Sun & Ni's Law

Special Cases:

- **G(n) = 1**

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

- **G(n) = n**

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

- **G(n) ≥ n**

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

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

https://hemanthrajhemu.github.io

Speedup Performance Laws

Sun & Ni's Law

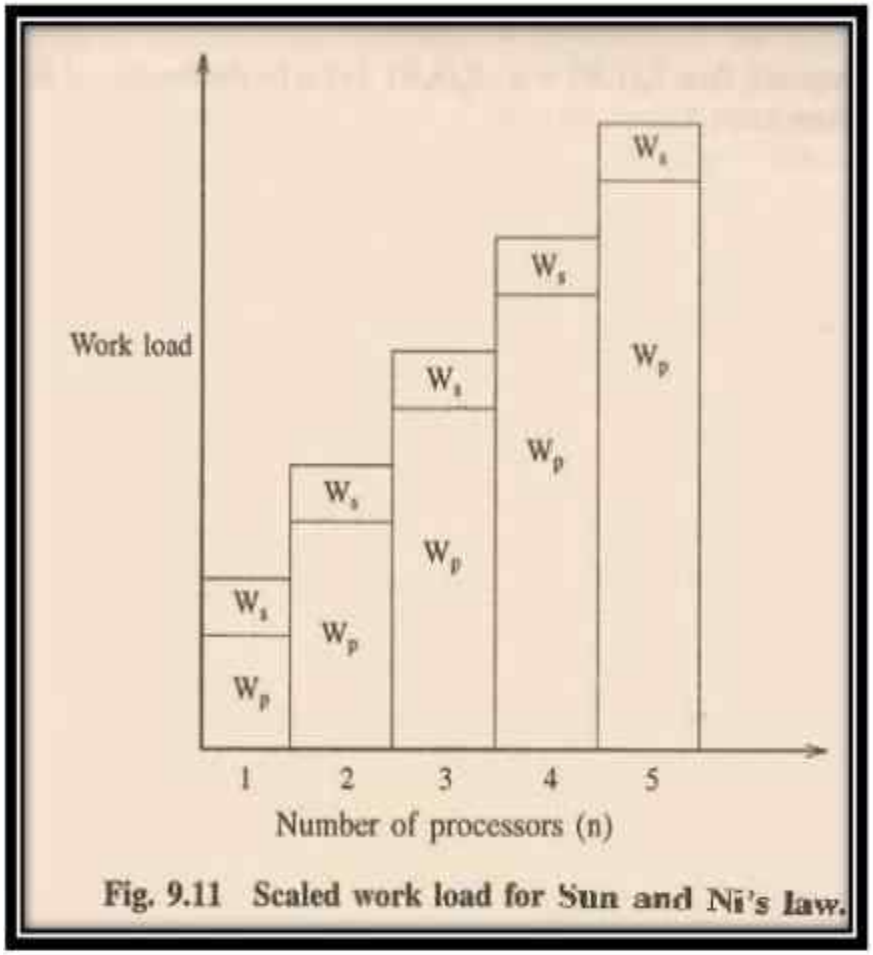


Fig. 9.11 Scaled work load for Sun and Ni's law.

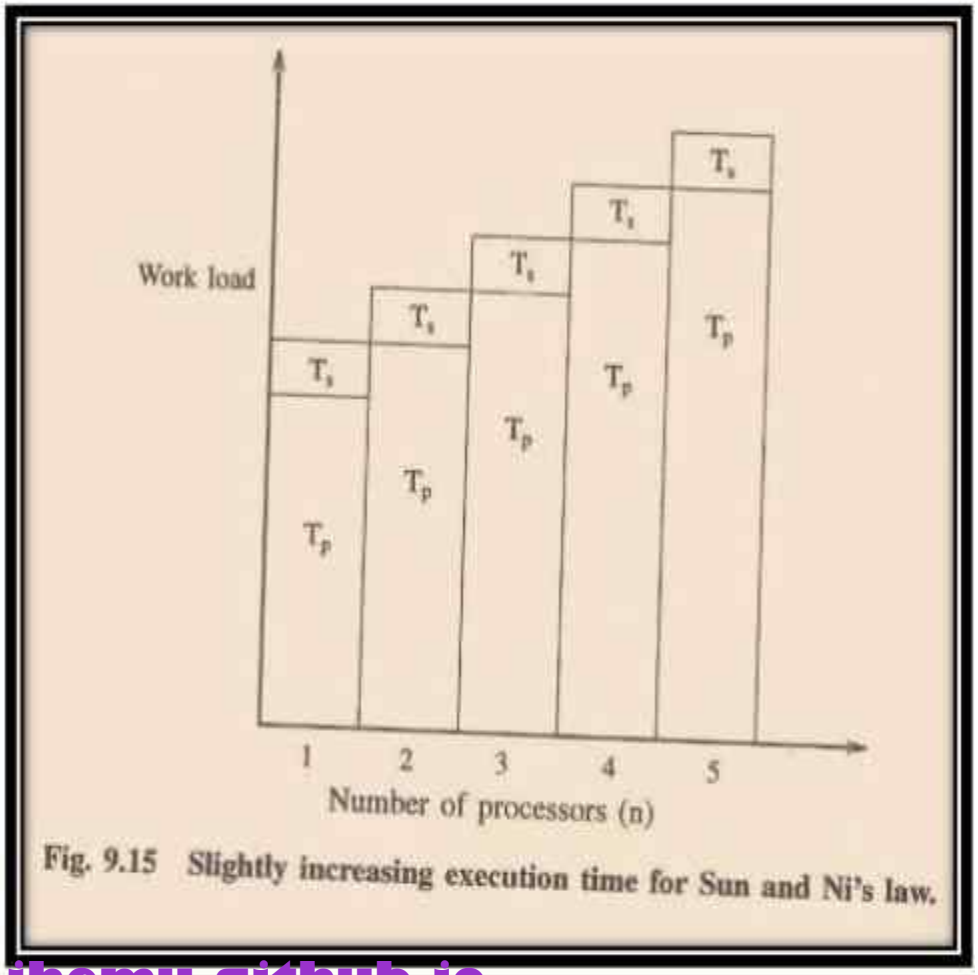


Fig. 9.15 Slightly increasing execution time for Sun and Ni's law.

https://hemanthrajhemu.github.io

Scalability Analysis and Approaches

Scalability

– Increasing the no. of processors decreases the efficiency!

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

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

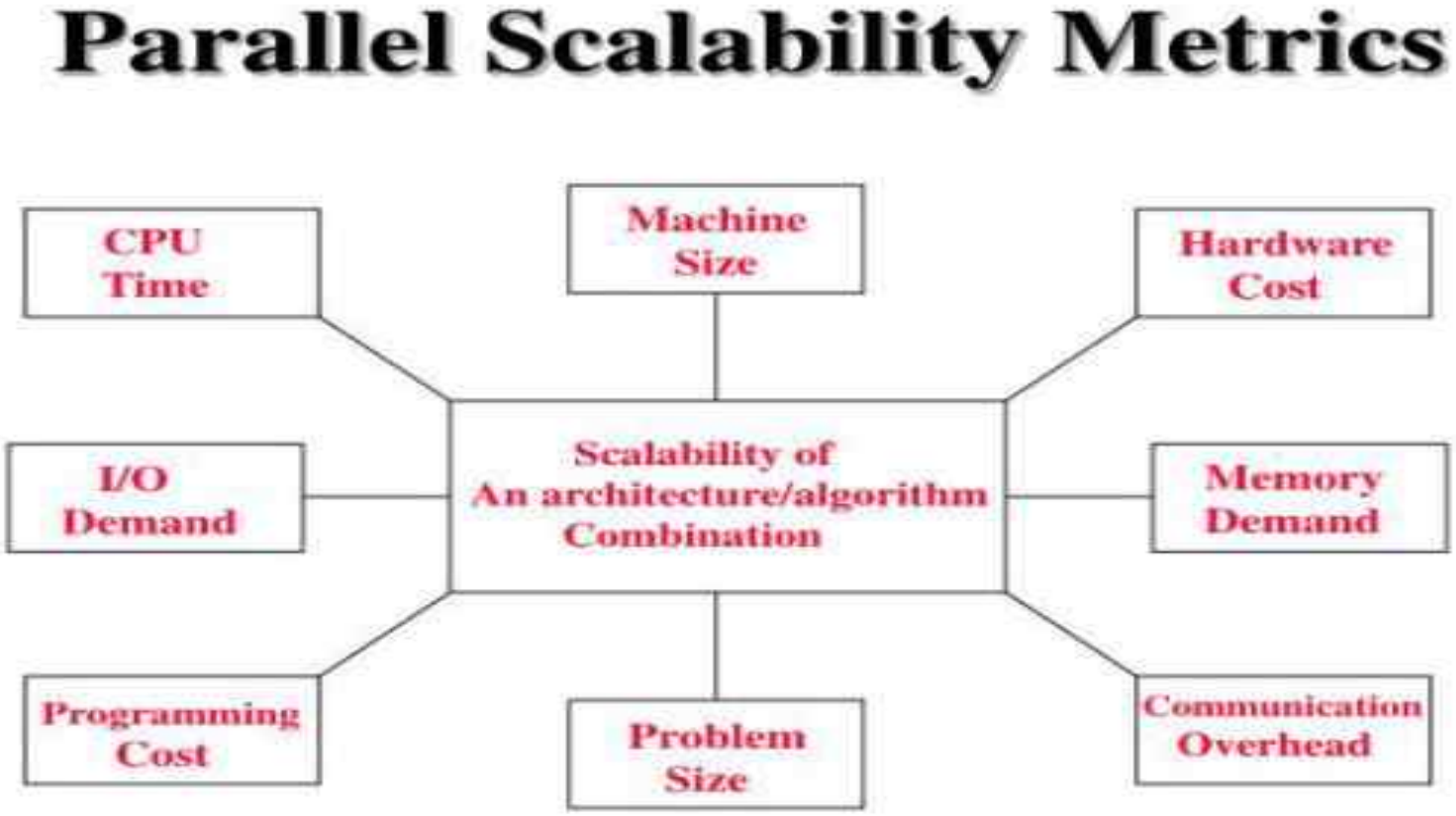
A parallel computing system is said to be scalable if its efficiency can be fixed by simultaneously increasing the machine size and the problem size.

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

https://hemanthrajhemu.github.io

Scalability Analysis and Approaches

Scalability Metrics and Goals:



https://hemanthrajhemu.github.io

Scalability Analysis and Approaches

Revised Asymptotic Speedup, Efficiency

- **Revised Asymptotic Speedup:**

$$S(s,n) = \frac{T(s,1)}{T(s,n) + h(s,n)}$$

Attained by varying only the number of processor(n)

  - s problem size.
  - T(s, 1) minimal sequential execution time on a uniprocessor.
  - T(s, n) minimal parallel execution time on an n-processor system.
  - h(s, n) lump sum of all communication and other overheads.

- **Revised Asymptotic Efficiency:**

$$E(s,n) = \frac{S(s,n)}{n}$$

https://hemanthrajhemu.github.io

Scalability Analysis and Approaches

- **Scalability** (informal very restrictive definition):

A system architecture is scalable if the system efficiency $E(s, n) = 1$ for all algorithms with any number of processors and any size problem s

- **Another Scalability Definition** (more formal):

The scalability $\Phi(s, n)$ of a machine for a given algorithm is defined as the ratio of the asymptotic speedup $S(s,n)$ on the real machine to the asymptotic speedup $S_I(s, n)$ on the ideal realization of an EREW PRAM

$$S_I(s,n) = \frac{T(s,1)}{T_I(s,n)}$$

$$\Phi(s,n) = \frac{S(s,n)}{S_I(s,n)} = \frac{T_I(s,n)}{T(s,n)}$$

Scalability Analysis and Approaches

Scalability Issues and Possible solutions:

- Problems:
  - Memory-access latency.
  - Interprocess communication complexity or synchronization overhead.
  - Multi-cache inconsistency.
  - Message-passing overheads.
  - Low processor utilization and poor system performance for very large system sizes.
- Possible Solutions:
  - Low-latency fast synchronization techniques.
  - Weaker memory consistency models.
  - Scalable cache coherence protocols.
  - To relize shared virtual memory.
  - Improved software portability; standard parallel and distributed operating system support.

https://hemanthrajhemu.github.io