NITTE EDUCATION TRUST | NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

Scalable Computing

18IS71

7th Semester

Department of Information Science and Engineering

NMIT

Course Outcomes:
Students will be able to:

Cos	Course Outcome Description	Blooms Level
1	Explain the fundamentals of parallel computing including models of computation, multiprocessor architectures, communication costs, and laws/theorems of parallel computation.	L2
2	Develop and evaluate MPI-based distributed memory programs using message-passing techniques for process-to-process and collective communications.	L4
3	Implement and analyze OpenMP-based shared memory programs by parallelizing loops, handling reductions, and managing task-level parallelism.	L4
4	Design and execute GPU-based applications using CUDA by leveraging thread hierarchies, memory management, and SIMT programming for computational acceleration.	L3
5	Apply Spark for large-scale in-memory data processing using RDDs and distributed transformations to solve real-world data-intensive problems efficiently.	L3

Unit -I Unit -I
Foundations

What is Scalable Computing?

Scalable computing is a computing paradigm that allows a system to increase (scale up/out) or decrease (scale down/in) its computing resources (CPU, memory, storage, network) in response to workload demand, while maintaining performance and cost efficiency.

Key aspects:

- **Elasticity:** Ability to add/remove resources dynamically.
- **Efficiency:** Avoid over-provisioning or under-utilization.
- **Transparency:** Applications should work seamlessly when the underlying resources change.

Types of Scalability

Vertical Scaling (Scale Up/Down)

Add more power (CPU, RAM) to a single machine.

Example: Upgrading a server from 8 cores to 32 cores.

Horizontal Scaling (Scale Out/In)

Add more nodes/machines to distribute workload.

Example: Adding more servers to a cloud cluster.

Diagonal / Hybrid Scaling

Combine vertical and horizontal approaches for flexibility.

ENABLING TECHNOLOGIES

CLOUD COMPUTING PLATFORMS (AWS, AZURE, GCP)

CLUSTER COMPUTING (HADOOP, SPARK)

CONTAINERIZATION & ORCHESTRATION (DOCKER, KUBERNETES)

LOAD BALANCERS FOR WORKLOAD DISTRIBUTION

Real-World Applications

1. Big Data Analytics → Spark clusters scale horizontally for processing petabytes of data.

2. Web Applications → Auto-scaling web servers during peak traffic.

3. Scientific Simulations → HPC clusters scale resources for computational physics or bioinformatics.

4. AI/ML Training → GPU clusters that scale for deep learning models.

Difference between Scalable, Parallel, and Distributed Computing

Aspect	Parallel Computing	Distributed Computing	Scalable Computing
Focus	Multiple processors execute tasks simultaneously	Tasks run on multiple networked systems	Adapts resources to workload demand
Goal	Reduce execution time	Share resources & tasks	Maintain performance as demand grows
Example	CUDA/OpenMP	Hadoop, MPI	AWS Auto Scaling, Kubernetes clusters

Why do we need parallel programming

We need **parallel programming** to **increase speed, process large data efficiently, utilize modern hardware fully, and meet real-time performance demands**.

1. Performance and Speed

- Sequential execution is slow for large-scale computations.
- Parallel programming divides tasks across multiple cores/processors, reducing execution time.
- Example: A task that takes 10 hours sequentially could complete in 1 hour on 10 parallel cores.

2. Handling Large-Scale Data

- Applications in Big Data, AI/ML, and scientific computing generate terabytes or petabytes of data.
- Parallel programming allows simultaneous processing of different data chunks.
- Example: Training deep neural networks on GPUs with thousands of parallel threads.

3. Efficient Resource Utilization

- Modern CPUs and GPUs have multiple cores that remain underutilized in sequential programs.
- Parallel programming maximizes hardware usage, improving efficiency.

4. Real-Time Applications

- Applications like autonomous driving, weather forecasting, video rendering, and gaming require real-time responses.
- Parallelism allows tasks like image recognition, sensor fusion, and simulation to run concurrently.

5. Scalability and Future-Proofing

- Moore's Law is slowing down; instead of faster single cores, we get more cores.

The number of transistors on an integrated circuit (IC) doubles approximately every two years, leading to exponential growth in computing power

Parallelism is found on all levels of a modern computer's architecture

- Parallelism exists at every level of a modern computer's architecture, from tiny CPU components to large-scale distributed clusters.

1. Instruction-Level Parallelism (ILP)

- Found in: CPU microarchitecture
- How it works: Executes multiple instructions per clock cycle.
- Examples:
  - Pipelining (fetch → decode → execute → write-back stages)
  - Superscalar processors (execute multiple instructions in parallel)

2. Data-Level Parallelism (DLP)

- Found in: Vector units (SIMD) and GPUs
- How it works: The same operation is performed on multiple data elements simultaneously.
- Examples:
  - AVX/NEON vector instructions
  - GPU threads processing matrix multiplications

3. Thread-Level Parallelism (TLP)

- Found in: Multi-core processors
- How it works: Multiple threads execute concurrently on different CPU cores.
- Examples:
  - Multi-core CPUs (Intel i9, AMD Ryzen)
  - Hyper-Threading / SMT (Simultaneous Multithreading)

4. Memory/Storage-Level Parallelism

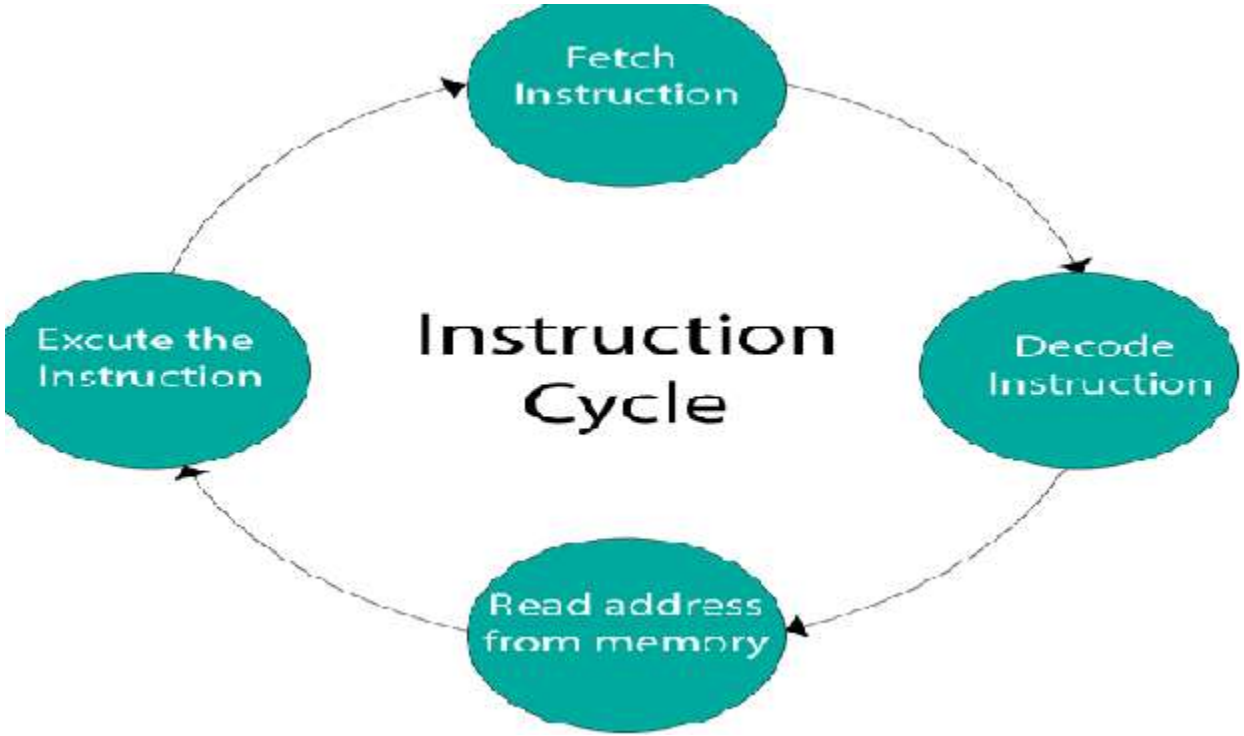
- Found in: Memory hierarchy and I/O subsystems
- How it works: Multiple memory accesses and I/O operations happen in parallel.
- Examples:
  - Multi-channel DRAM
  - SSDs with multiple NAND flash controllers

5. Task-Level / Process-Level Parallelism

- Found in: Operating systems and distributed applications
- How it works: Independent processes or tasks execute concurrently.
- Examples:
  - Running multiple applications simultaneously
  - Microservices in cloud computing

6. Cluster-Level / Distributed Parallelism

- Found in: Supercomputers, cloud, HPC clusters
- How it works: Multiple machines (nodes) work together on a large computation.
- Examples:
  - Apache Spark clusters for Big Data
  - MPI-based scientific simulations
  - Google Cloud TPU pods for AI


Fetch Instruction
Decode Instruction
Read address from memory
Excute the Instruction
Instruction Cycle

1st Level: Instruction-Level Parallelism (ILP)

- **Registers and ALUs**
  - The Arithmetic Logic Units (ALUs) perform the core operations like add, subtract, and logic functions.
  - Multiple ALUs can work in parallel if instructions are independent.

- **Pipelining**
  - Instructions are divided into stages (fetch, decode, execute, write-back).
  - Multiple instructions can be in different stages at the same time.

- **Superscalar Execution**
  - Modern CPUs can issue multiple instructions per clock cycle using multiple functional units.

- **Out-of-Order Execution (OoOE)**
  - CPU executes instructions out of order for better utilization of ALUs and pipelines.

- **Branch Prediction & Speculative Execution**
  - Predicts upcoming instructions to keep the pipeline full and reduce stalls

2. Data-Level Parallelism (DLP)

Flynn's taxonomy

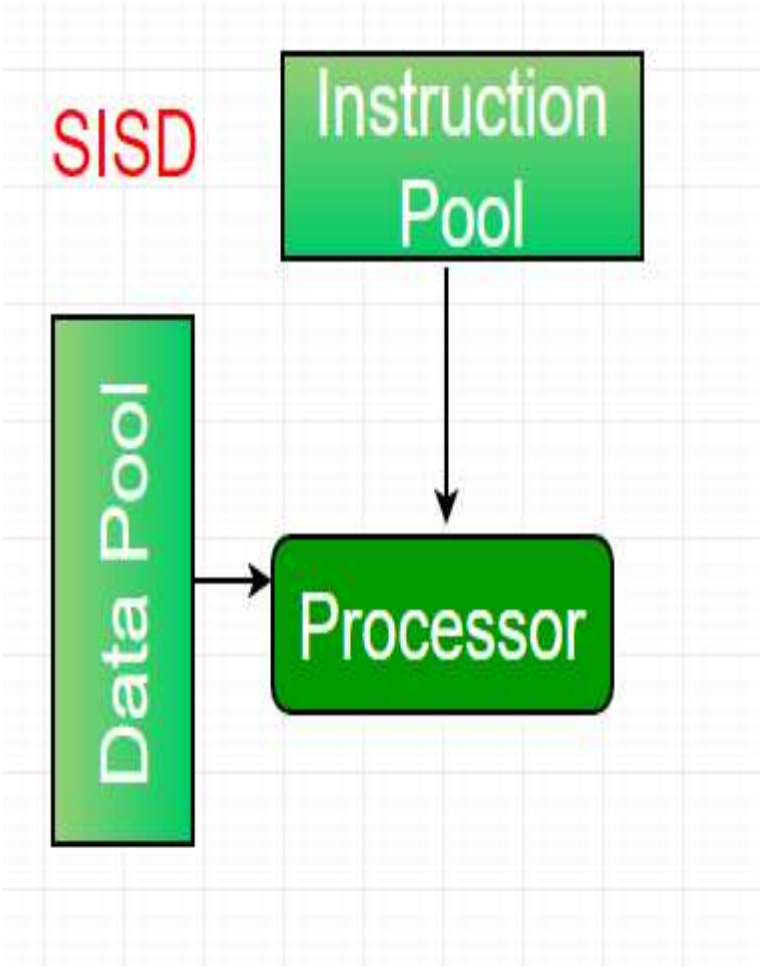
➢**Single-instruction, single-data (SISD) systems**

An SISD computing system is a uniprocessor machine that is capable of executing a single instruction, operating on a single data stream.

In SISD, machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential computers.

Most conventional computers have SISD architecture.

All the instructions and data to be processed have to be stored in primary memory.

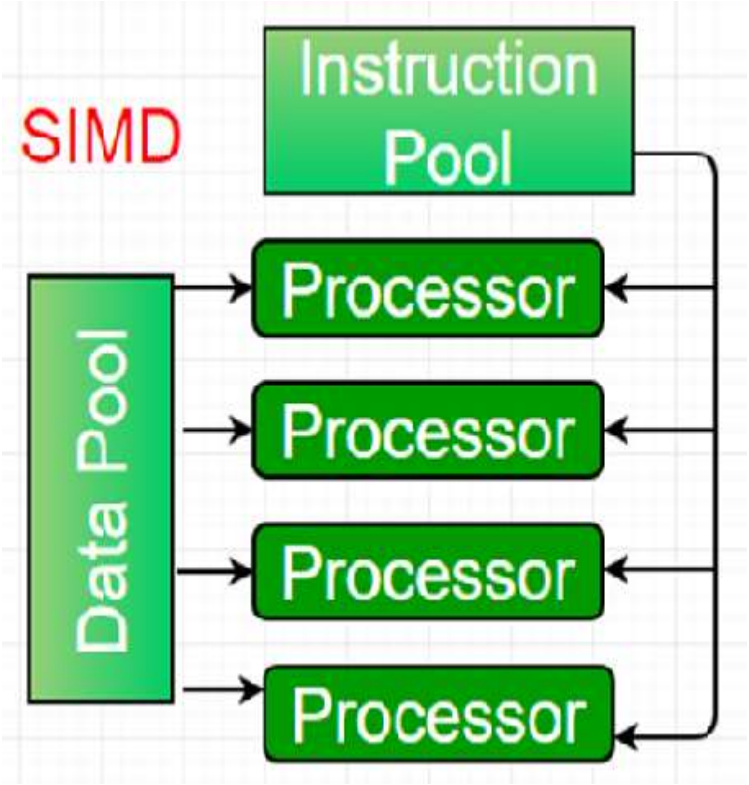


- **Single-instruction, multiple-data (SIMD) systems**

- An SIMD system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams.

- Machines based on a SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations.

- So that the information can be passed to all the processing elements (PEs) organized data elements of vectors can be divided into multiple sets(N-sets for N PE systems) and each PE can process one data set.

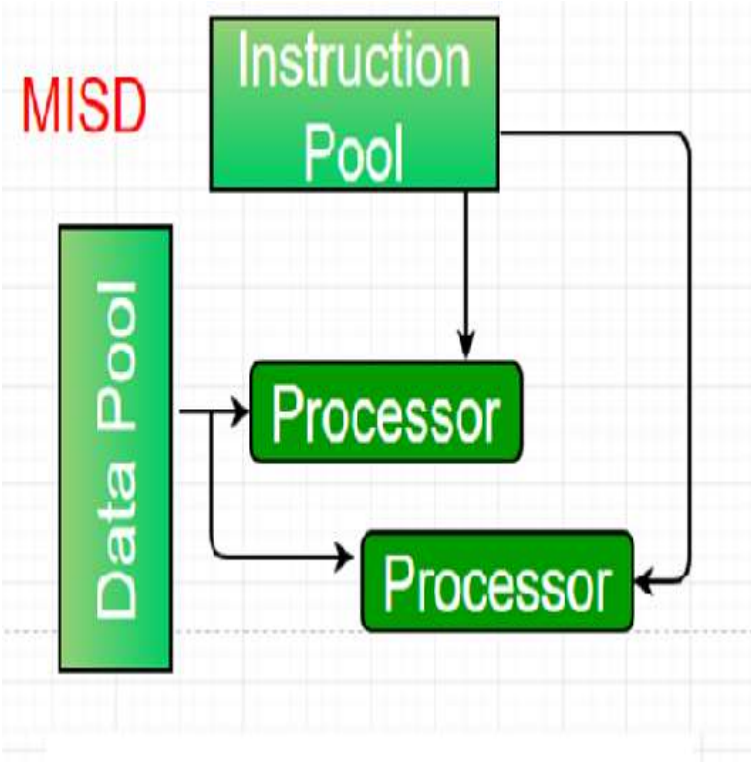


- **Multiple-instruction, single-data (MISD) systems**

- An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operate on the same dataset.

- Example Z = sin(x)+cos(x)+tan(x) The system performs different operations on the same data set.

- Machines built using the MISD model are not useful in most applications, a few machines are built, but none of them are available commercially.

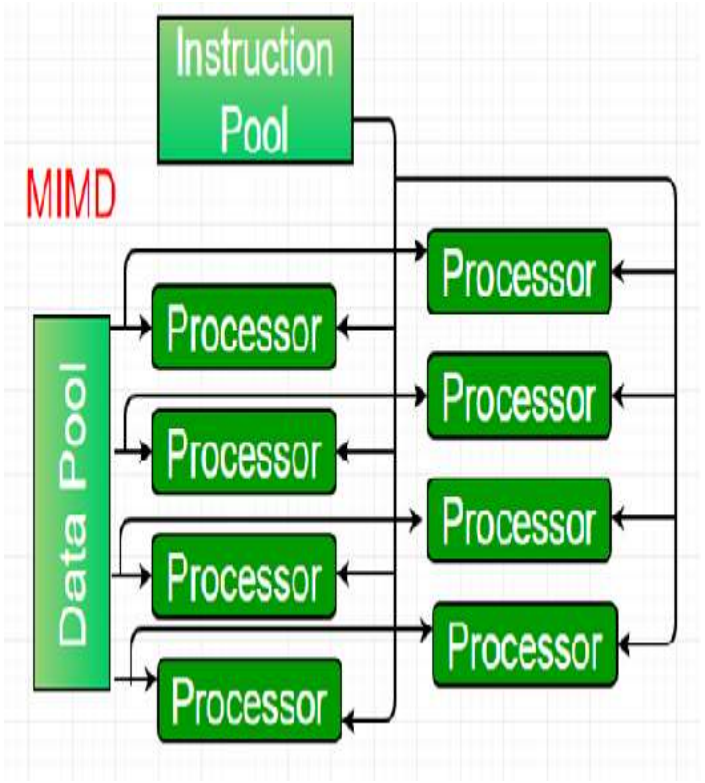


- **Multiple-instruction, multiple-data (MIMD) systems**

- An MIMD system is a multiprocessor machine that is capable of executing multiple instructions on multiple data sets.

- Each PE in the MIMD model has separate instruction and data streams; therefore machines built using this model are capable of any application.

- Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.



1st levels of a modern computer's architecture

- First, parallelism is present deep in the processor microarchitecture.

- In the past, processors ran programs by repeating the so-called instruction cycle, a sequence of four steps:

  (i) reading and decoding an instruction;

  (ii) finding data needed to process the instruction;

  (iii) processing the instruction; and

  (iv) writing the result out.

- Since step (ii) introduced lengthy delays which were due to the arriving data, much of research focused on designs that reduced these delays and in this way increased the effective execution speed of programs.

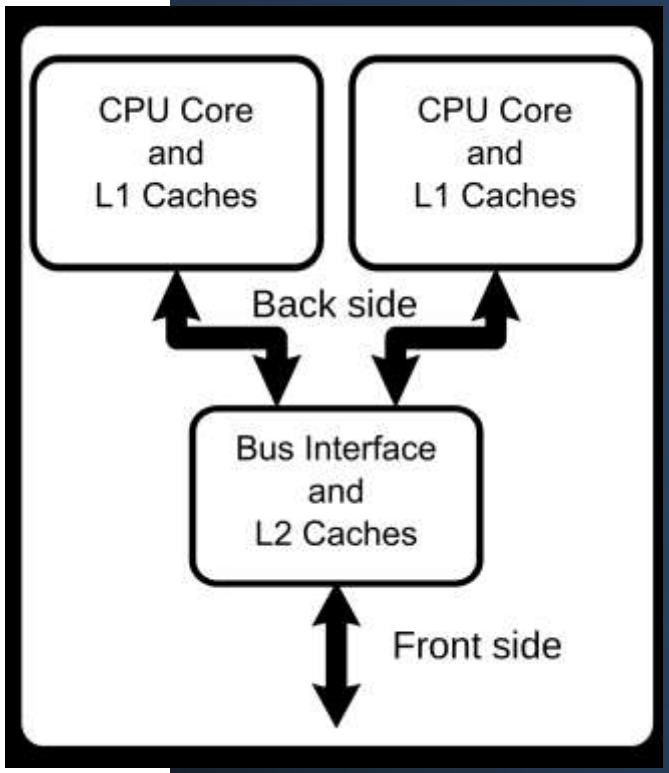
- Over the years, however, the main goal has become the design of a processor capable of execution of several instructions simultaneously.

- The workings of such a processor enabled detection and exploitation of parallelism inherent in instruction execution.

- These processors allowed even higher execution speeds of programs, regardless of the processor and memory frequency.

2nd levels of a modern computer's architecture

- Commercial computer, tablet and smartphone contains a processor with multiple cores
  - each of which is capable of running its own instruction stream.
- Cores collaborate in running an application,
  - Application run in parallel and may be considerably speed up.

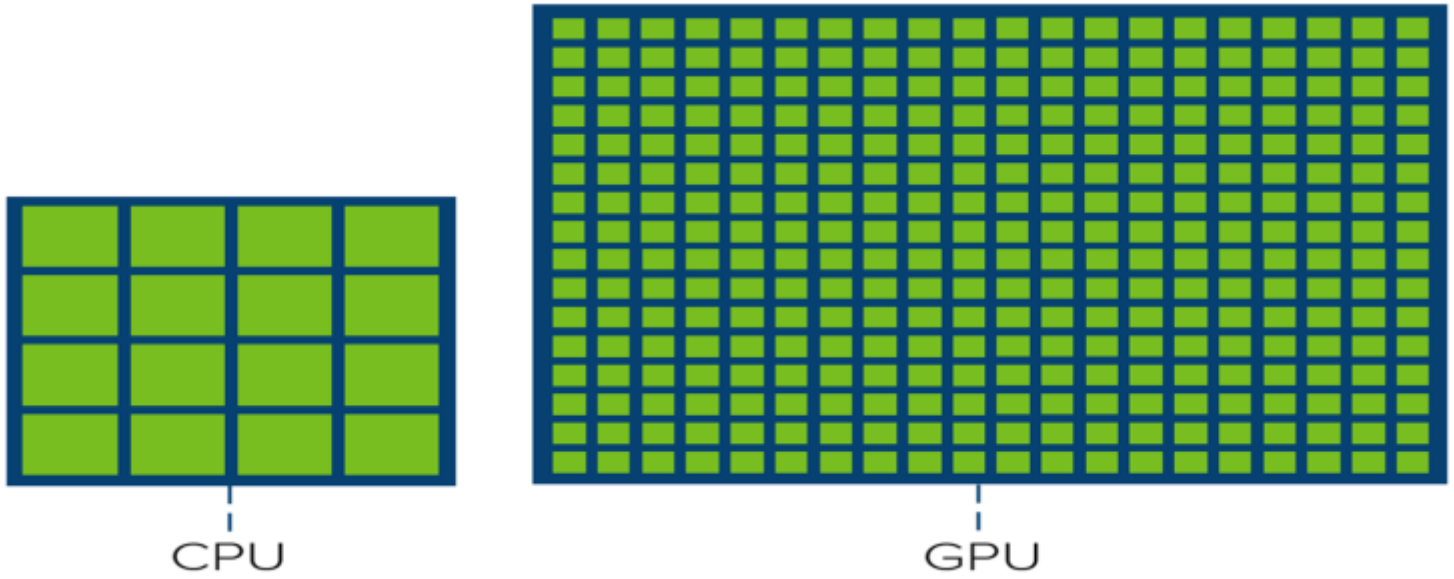


3rd levels of a modern computer's architecture

- Servers contain a several multicore processors.
- Server is capable of running a service in parallel, and also several services in parallel.

4th level of a modern computer's architecture

- Consumer level computers contain graphic processors capable of running hundreds or even thousands of threads in parallel.
- Processors capable of coping with such a large parallelism are necessary to support graphic animation.

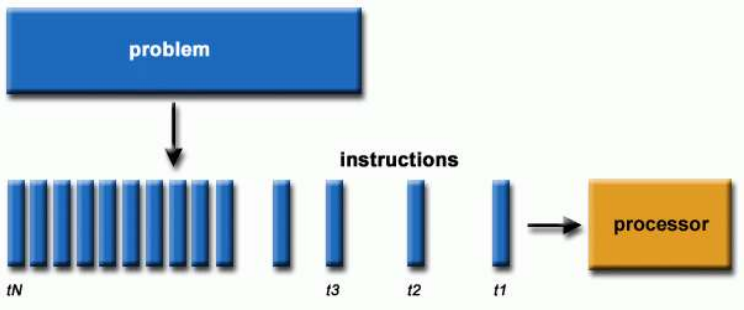


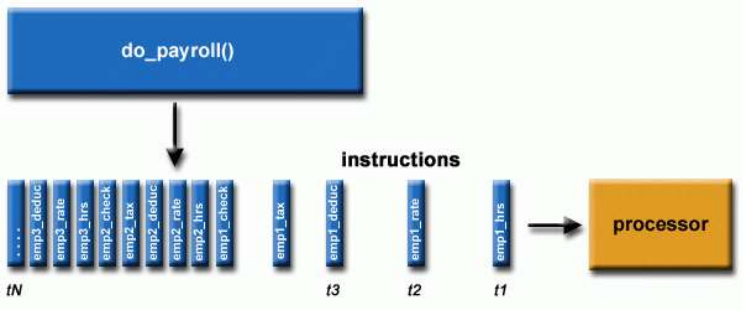
Reasons for making modern computers parallel

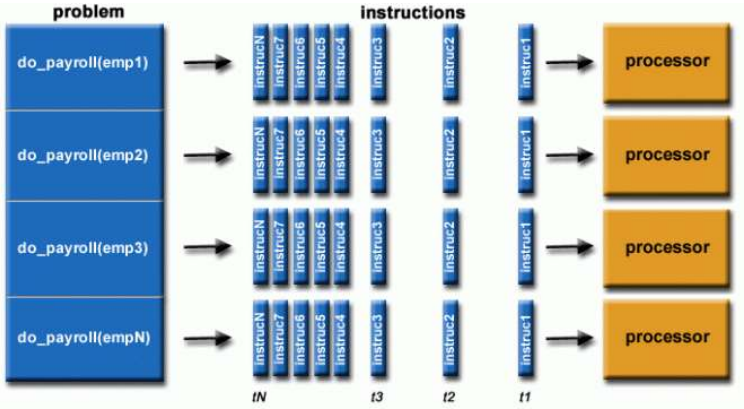
- First, it is not possible to increase processor and memory frequencies indefinitely, at least not with the current silicon-based technology. Therefore, to increase computational power of computers, new architectural and organizational concepts are needed.

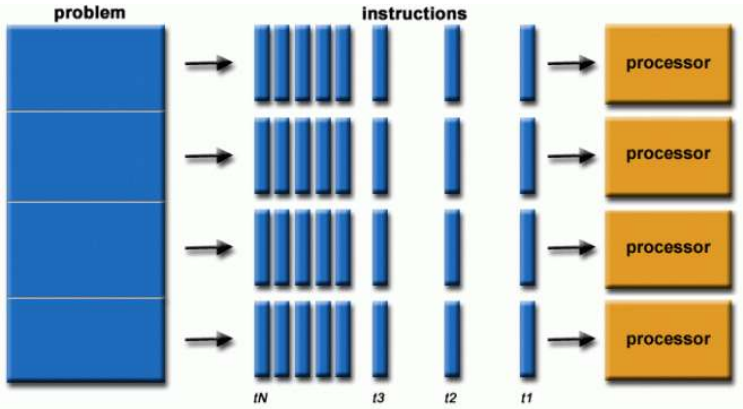
- Second, power consumption rises with processor frequency while the energy efficiency decreases. However, if the computation is performed in parallel at lower processor speed, the undesirable implications of frequency increase can be avoided.

- Finally, parallelism has become a part of any computer and this is likely to remain unchanged due to simple inertia: parallelism can be done and it sells well.


problem
instructions
processor
tN
t3
t2
t1


do_payroll()
instructions
emp3_deduc
emp3_rate
emp3_hrs
emp2_check
emp2_tax
emp2_deduc
emp2_rate
emp2_hrs
emp1_check
emp1_tax
emp1_deduc
emp1_rate
emp1_hrs
processor
tN
t3
t2
t1


problem
instructions
do_payroll(emp1)
do_payroll(emp2)
do_payroll(emp3)
do_payroll(empN)
instrucN
instruc7
instruc6
instruc5
instruc4
instruc3
instruc2
instruc1
processor
tN
t3
t2
t1


problem
instructions
processor
tN
t3
t2
t1

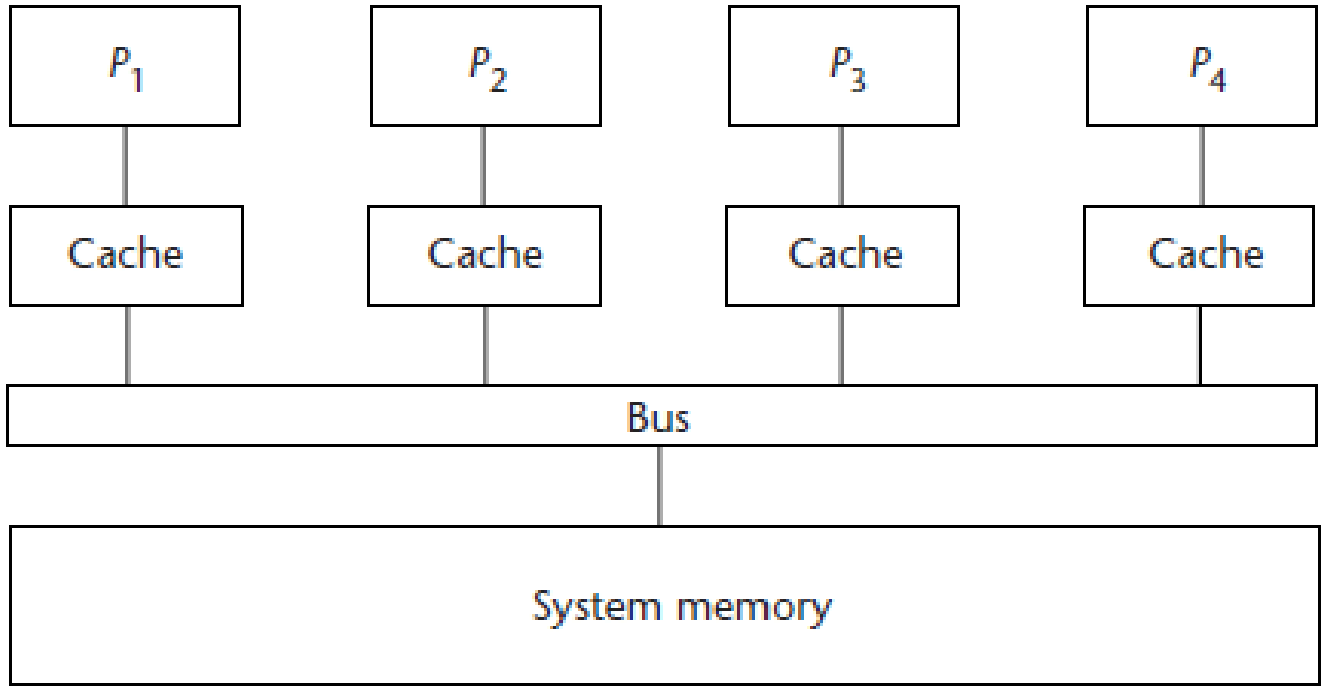
Three prevailing types of parallelism

- Starting from last decade, many different parallel computing systems appeared on the market.
  - Supercomputers dedicated for solving specific scientific problems.
  - Parallelism has spread all the way down into the consumer market and all kinds of handheld devices.

Types of parallelism

1. Shared memory systems

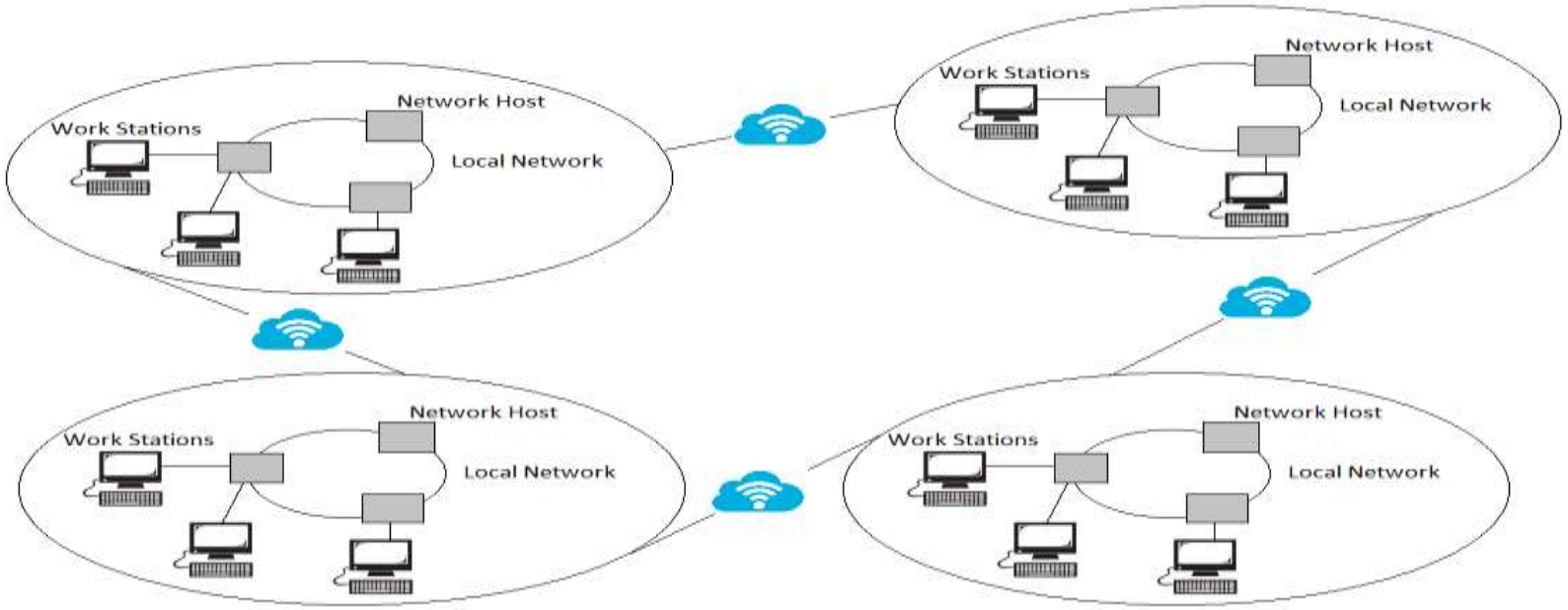
- Systems with multiple processing units attached to a single memory.



Types of parallelism

2. Distributed systems

- Systems consisting of many computer units
- Each with its own processing unit and its physical memory
- That are connected with fast interconnection networks.



Types of parallelism

3. Graphic processor units

- Co-proccessors for solving general purpose numerically intensive problems.

Types of parallelism

- Design of parallel algorithms and parallel programming are still considered to be an order of magnitude harder than the design of sequential algorithms and sequential-program development.

- Three different approaches to parallel programming exist:
  - Threads model for shared memory systems,
  - Message passing model for distributed systems
  - Stream based model for GPUs.

MIMD machines are broadly categorized into

1.S**hared-memory MIMD** and

2. Distributed-memory MIMD based on the way PEs are coupled to the main memory.

- In the **shared memory MIMD** model (tightly coupled multiprocessor systems), all the PEs are connected to a single global memory, and they all have access to it.

- The communication between PEs in this model takes place through the shared memory, modification of the data stored in the global memory by one PE is visible to all other PEs.

- The dominant representative shared memory MIMD systems are Silicon Graphics machines and Sun/IBM's SMP (Symmetric Multi-Processing).

- **Distributed memory MIMD** machines (loosely coupled multiprocessor systems) all PEs have a local memory.

- The communication between PEs in this model takes place through the interconnection network (the inter-process communication channel, or IPC).

- The network connecting PEs can be configured to tree, mesh, or in accordance with the requirement.

- The shared-memory MIMD architecture is easier to program but is less tolerant to failures and harder to extend with respect to the distributed memory MIMD model.

Application Classes

- **FEA – Finite Element Analysis**
  - Simulation of hard physical materials, e.g. metal, plastic Crash test, product design, suitability for purpose
  - Examples: MSC Nastran, Ansys, LS-Dyna, Abaqus, ESI PAMCrash, Radioss
- **CFD – Computational Fluid Dynamics**
  - Simulation of soft physical materials, gases and fluids Engine design, airflow, oil reservoir modelling
  - Examples: Fluent, Star-CD, CFX
- **Geophysical Sciences**
  - Seismic Imaging – taking echo traces and building a picture of the sub-earth geology
  - Reservoir Simulation – CFD specific to oil asset management
  - Examples: Omega, Landmark VIP and Pro/Max, Geoquest Eclipse

Application Classes

- **Life Sciences**
  - Understanding the living world – genome matching, protein folding, drug design, bio-informatics, organic chemistry
  - Examples: BLAST, Gaussian, other
- **High Energy Physics**
  - Understanding the atomic and sub-atomic world
  - Software from Fermi-Lab or CERN, or home-grown
- **Financial Modelling**
  - Meeting internal and external financial targets particularly regarding investment positions
  - VaR – Value at Risk – assessing the impact of economic and political factors on the bank's investment portfolio
  - Trader Risk Analysis – what is the risk on a trader's position, a group of traders

Dependencies

- Understanding data dependencies is fundamental in implementing parallel algorithms.
- No program can run more quickly than the longest chain of dependent calculations.
- However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.
- Let P_i and P_j be two program segments. Bernstein's conditions describe when the two are independent and can be executed in parallel.
- For P_i, let I_i be all of the input variables and O_i the output variables, and likewise for P_j. P_i and P_j are independent if they satisfy

Let's assume we have **two processes**:

- **P_1** — first process
- **P_2** — second process
- **I(P)** — set of input variables to process P
- **O(P)** — set of output variables from process P

2. Bernstein's Conditions

For **P_1** and **P_2** to execute in parallel **safely**, the following must hold:

- **No Read-After-Write (RAW) hazard**
  $O(P1) \cap I(P2) = \emptyset$

→ P_2 should not read data that P_1 writes (no dependency).

- **No Write-After-Read (WAR) hazard**
  $I(P1) \cap O(P2) = \emptyset$

→ P_2 should not write to data that P_1 reads.

- **No Write-After-Write (WAW) hazard**
  $O(P1) \cap O(P2) = \emptyset$

→ P_1 and P_2 should not write to the same variable.

If **all three conditions** are satisfied, **P_1** and **P_2** can execute in parallel.

Example

Let:

P_1: A = B + C

P_2: D = E + F

P_1: X = Y + Z

P_2: Y = W + V

Here:

1. I(P1)={B,C}, O(P1)={A}

2. I(P2)={E,F}, O(P2)={D}

Check conditions:

- O(P1) ∩ I(P2)={}
- I(P1) ∩ O(P2)={}
- O(P1) ∩ O(P2)={}
- **All clear — can run in parallel**.

Consider the following functions, which demonstrate several kinds of dependencies:

```
1: function Dep(a, b)
2: c := a * b
3: d := 3 * c
4: end function
```

In this example, instruction 3 cannot be executed before (or even in parallel with) instruction 2, because instruction 3 uses a result from instruction 2.
It violates condition 1 and thus introduces a flow dependency.

```
1: function NoDep(a, b)
2: c := a * b
3: d := 3 * b
4: e := a + b
 5: end function
```

In this example, there are no dependencies between the instructions, so they can all be run in parallel.
Bernstein's conditions do not allow memory to be shared between different processes. For that, some means of enforcing an ordering between accesses is necessary, such as semaphores, barriers or some other synchronization method.

- I_1 : x = (a + b) / (a * b)
- I_2 : y = (b + c) * d
- I_3 : z = x2 + (a * e)

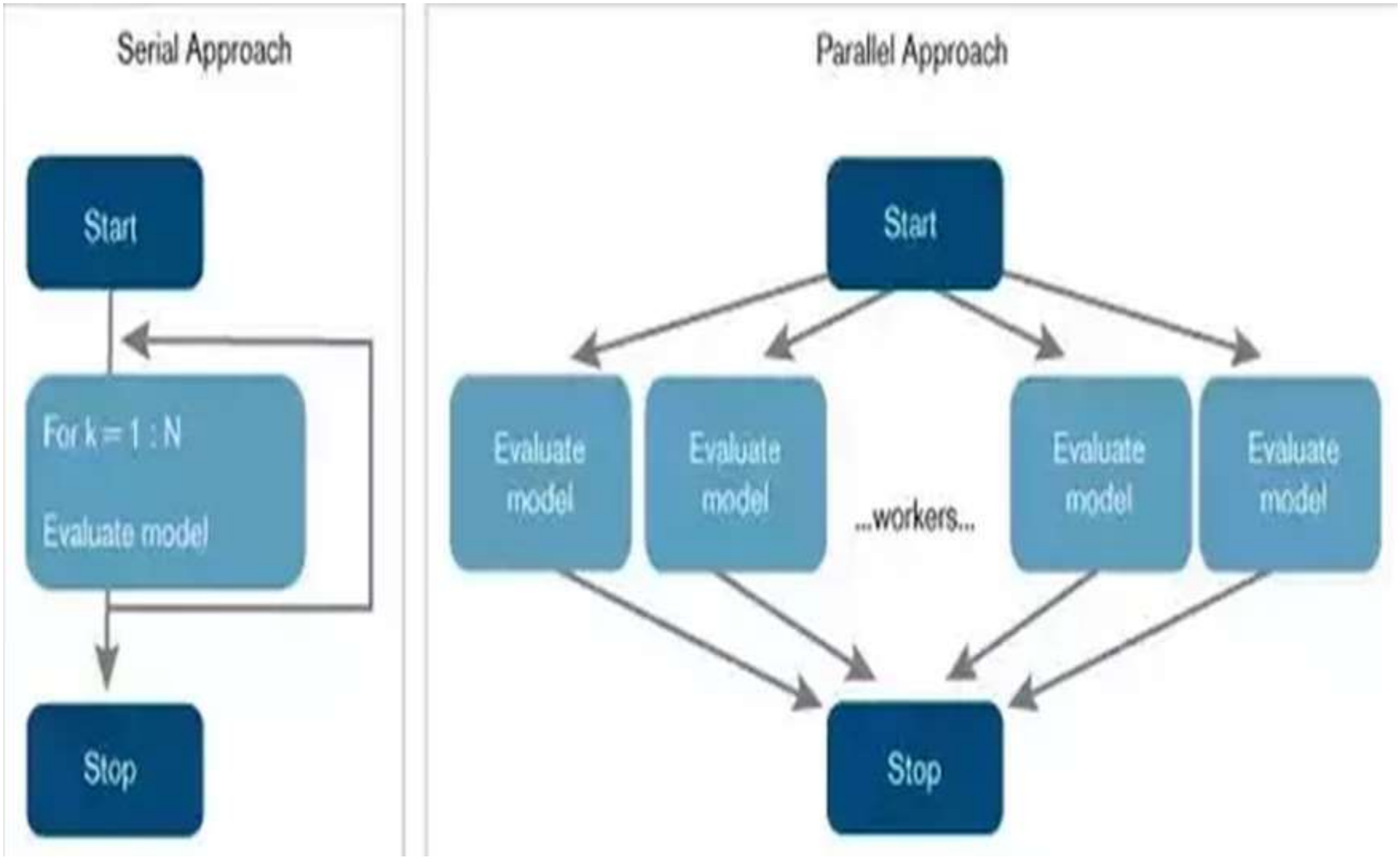
Now, the read set and write set of I1, I2 and I3 are as given:

- I_1 = {a,b} O1 = {x}
- I_2 = {b,c,d} O2 = {y}
- I_3 = {x,a,e} O3 = {z}

Overview of parallel systems

parallel computing

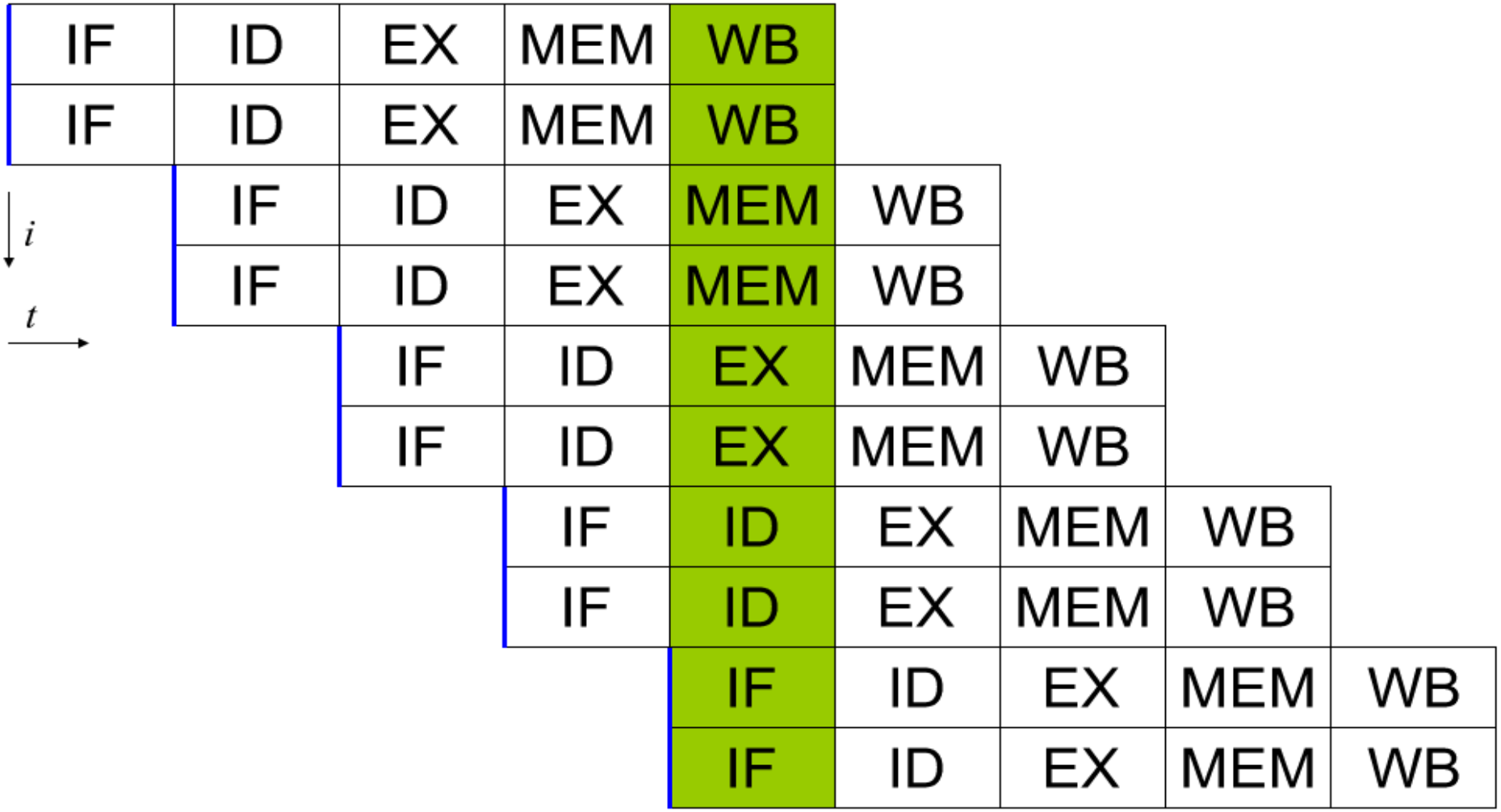
- Let P be an arbitrary computational problem which is to be solved by a computer.
- First objective is to design an algorithm for solving P.
- Class of all algorithms is infinite, but we can partition it into two subclasses
  - class of all sequential algorithms
  - class of all parallel algorithms.
- Sequential algorithm performs one operation in each step
- Parallel algorithm may perform multiple operations in a single step.


Serial Approach
Start
For k = 1 : N
Evaluate model
Stop
Parallel Approach
Start
Evaluate model
Evaluate model
...workers...
Evaluate model
Evaluate model
Stop

parallel computing

- Let P be an arbitrary parallel algorithm
  - There is parallelism in P.
- Parallelism in P can be exploited by various kinds of parallel computers.
  - Multiple operations of P may be executed simultaneously by multiple processing units of a parallel computer C1
  - Multiple operations of P may be executed by multiple pipelined functional units of a single-processor computer C2.
  - P can always be sequentially executed on a single-processor computer C3
    - simply by executing P's potentially parallel operations one by one in succession.

Multiple pipelined functional units



parallel computing

- Let C(p) be a parallel computer of the kind C which contains p processing units.
- Performance of P on C(p) depend both on C and p.
  - Potential parallelism in P
  - Capability of C(p) to execute in parallel,
  - Multiple operations of P
- So the performance of the algorithm P on the parallel computer C(p) depends on C(p)'s capability to exploit P's potential parallelism.

parallel computing

- “performance" →time required to execute P on C(p);
  - Parallel execution time (or, parallel runtime) of P on C(p),
  - Denote by Tpar

parallel computing

- “performance” -How many times parallel execution of P on C(p) faster than the sequential execution of P;
  - Speedup of P on C(p),
  - S = Tseq / Tpar
  - Parallel execution of P on C(p) is S-times faster than sequential execution of P.

parallel computing

- We might be interested in how much of the speedup S is, on average, due to each of the processing units.
- "performance" -average contribution of each of the p processing units of C(p) to the speedup;
  - Efficiency of P on C(p),
  - E = S / p

parallel computing

- Tpar <= Tseq <=p .Tpar
- Speedup and efficiency are bounded by
  - E <= 1:
  - For any C and p, the parallel execution of P on C(p) can be at most p times faster than the execution of P on a single processor.
  - Efficiency of the parallel execution of P on C(p) can be at most 1.

parallel computing

- Both speedup and efficiency depend on Tpar, the parallel execution time of P on C(p).

How do we determine T_{par}?
How does T_{par} depend on C (the kind of a parallel computer) ?
Which properties of C must we take into account in order to determine T_{par}?

- Answer -to appropriately model parallel computation.

Modeling parallel computation

- Parallel computers vary greatly in their organization.
  - Processing units may or may not be directly connected one to another;
    - some of the processing units may share a common memory
    - while the others may only own local (private) memories;
  - Operation of the processing units may be synchronized
    - by a common clock
    - Or they may run each at its own pace.

Modeling parallel computation

- Architectural details and hardware specifics of the components
  - Design and use of a computer.
- Technological differences
  - different clock rates, memory access times etc.
- Hence, the following question arises:
  - Which properties of parallel computers must be considered and which may be ignored in the design and analysis of parallel algorithms?

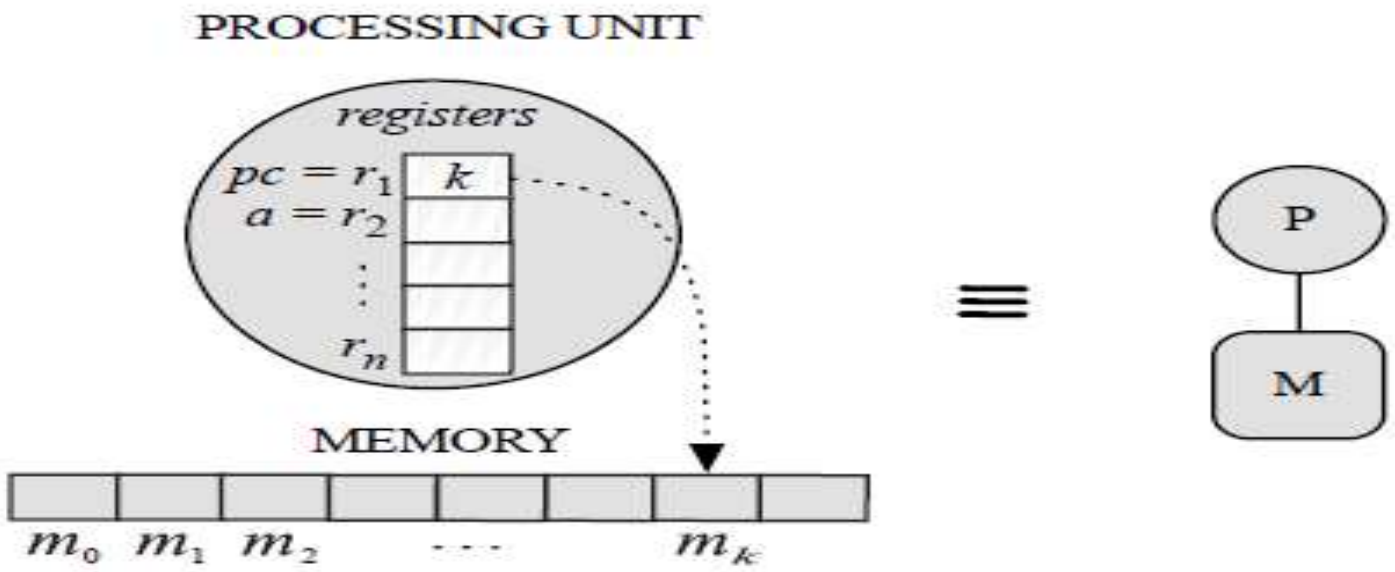
Modeling parallel computation

- To answer the question, we apply ideas similar to those discovered in the case of sequential computation.
  - Various models of computation were discovered
  - Intention of each of these models was to abstract the relevant properties of the (sequential) computation from the irrelevant ones.

Modeling parallel computation

Model called the Random Access Machine (RAM) is particularly attractive.

The RAM model of computation has a memory M (containing program instructions and data) and a processing unit P (executing instructions on data).



Modeling parallel computation

- RAM consists of a processing unit and a memory.
- Memory is a potentially infinite sequence of equally sized locations m0,m1, …..
- Index i is called the address of mi.
- Each location is directly accessible by the processing unit:
- Given an arbitrary i, reading from mi or writing to mi is accomplished in constant time.
- Registers are a sequence r1 ….. rn of locations in the processing unit.
- Registers are directly accessible.

Modeling parallel computation

- Program counter pc (= r1)
  - contains the address of the location in the memory which contains the instruction to be executed next.
- Accumulator a (= r2)
  - involved in the execution of each instruction.
- Other registers are given roles as needed.
- Program is a finite sequence of instructions
- Before the RAM is started, the following is done:
  - Program is loaded into successive locations of the memory starting with, say, m0
  - Input data are written into empty memory locations

What is the appropriate model of parallel computation?

- Substantially more challenging
- Why?
  - Many ways to organize parallel computers,
  - Many ways to model them
  - Selecting a single model that will be appropriate for all parallel computers.
- As a result, in the last decades, researchers proposed several models of parallel computation.
  - However, no common agreement has been reached about which is the right one.

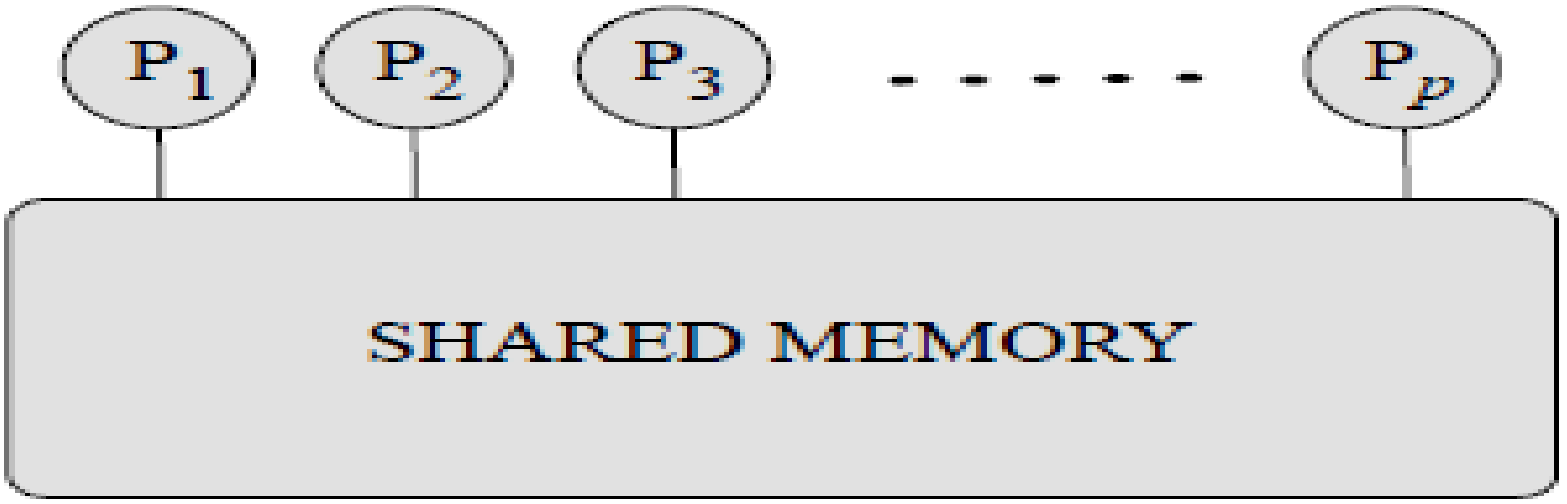
Multiprocessor models

- Multiprocessor model is a model of parallel computation that builds on the RAM model of computation;
- Three different multiprocessor models.
  - Each of the three models has some number p(> 2) of processing units
  - Models differ in the organization of their memories and in the way the processing units access the memories.
    - Parallel Random Access Machine (PRAM)
    - Local Memory Machine (LMM)
    - Modular Memory Machine (MMM).

Parallel Random Access Machine

- Parallel Random Access Machine(PRA) model, has p processing units that are all connected to a common unbounded shared memory
- Each processing unit can access any location (word) in the shared memory by issuing a memory request directly to the shared memory.

Parallel Random Access Machine

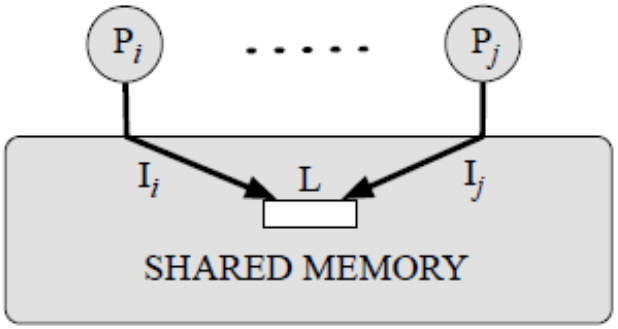


PRAM model of parallel computation is idealized in several respects.

- No limit on the number p of processing units, except that p is finite.
- Processing unit can access any location in the shared memory in one single step.
- Words in the shared memory are of the same size;
  - otherwise they can be of arbitrary finite size

Parallel Random Access Machine

- No interconnection network for transferring memory requests and data back and forth between processing units and shared memory.
- Assumption -Any processing unit can access any memory location in one step.
  - Suppose that processing units Pi and Pj simultaneously issue instructions I_i and I_j where both instructions intend to access (for reading from or writing to) the same memory location L



Parallel Random Access Machine

- Even if a truly simultaneous physical access to L had been possible,
  - such an access could have resulted in unpredictable contents of L.
  - Imagine what would be the contents of L after simultaneously writing 3 and 5 into it.
- It is reasonable to assume
  - Actual accesses of Ii and I j to L are somehow, on the fly serialized (sequentialized) by hardware so that Ii and I j physically access L one after the other.

Parallel Random Access Machine

- Does such an implicit serialization neutralize all hazards of simultaneous access to the same location?
  - Unfortunately not so.
  - Order of physical accesses of Ii and Ij to L is unpredictable:
  - We cannot know whether Ii will physically access L before or after Ij.

Parallel Random Access Machine

- Consequently, also the effects of instructions Ii and Ij are unpredictable Why?
- If both Pi and Pj want to read simultaneously from L,
  - Instructions Ii and Ij will both read the same contents of L, regardless of their serialization,
  - Both processing units will receive the same contents of L—as expected.

Parallel Random Access Machine

- if one of the processing units wants to read from L and the other simultaneously wants to write to L
  - Data received by the reading processing unit will depend on whether the reading instruction has been serialized before or after the writing instruction.
- if both Pi and Pj simultaneously attempt to write to L,
  - Resulting contents of L will depend on how Ii and I j have been serialized, i.e., which of Ii and I j was the last to physically write to L.

The variants of PRAM

PRAM (Parallel Random Access Machine) Models

- EREW PRAM – Exclusive Read, Exclusive Write (no concurrent memory access).
- CREW PRAM – Concurrent Read, Exclusive Write.
- ERCW PRAM – Exclusive Read, Concurrent Write.
- CRCW PRAM – Concurrent Read, Concurrent Write

Model	Multiple Read?	Multiple Write?	Rule for Writes
EREW (Exclusive Read, Exclusive Write)	✗	✗	No two processors can read/write the same location at the same time.
CREW (Concurrent Read, Exclusive Write)	✓	✗	Many can read simultaneously, but only one can write at a time.
CRCW (Concurrent Read, Concurrent Write)	✓	✓	Needs a tie-breaking rule for writes.
CRCW Subtypes:			
- **Common**	✓	✓	All writing processors must write the same value.
- **Arbitrary**	✓	✓	One processor's write is chosen arbitrarily.
- **Priority**	✓	✓	Processor with highest priority writes.

EREW-PRAM.

- EREW-PRAM model does not support simultaneous accessing to the same memory location;
  - if such an attempt is made, the model stops executing its program.
- Implicit assumption is that programs running on EREW-PRAM never issue instructions that would simultaneously access the same location
  - Any access to any memory location must be exclusive.
  - Construction of such programs is the responsibility of algorithm designers

CREW-PRAM

- Model supports simultaneous reads from the same memory location but requires exclusive writes to it.
  - Burden of constructing such programs is on the algorithm designer

CRCW-PRAM

- Least realistic of the three versions of the PRAM model.
- CRCW-PRAM model allows
  - simultaneous reads from the same memory location,
  - simultaneous writes to the same memory location
  - Simultaneous reads from and writes to the same memory location.
- To avoid unpredictable effects, different additional restrictions are imposed on simultaneous writes.
  - This yields to versions of the model CRCW-PRAM:

Relevance of the PRAM model

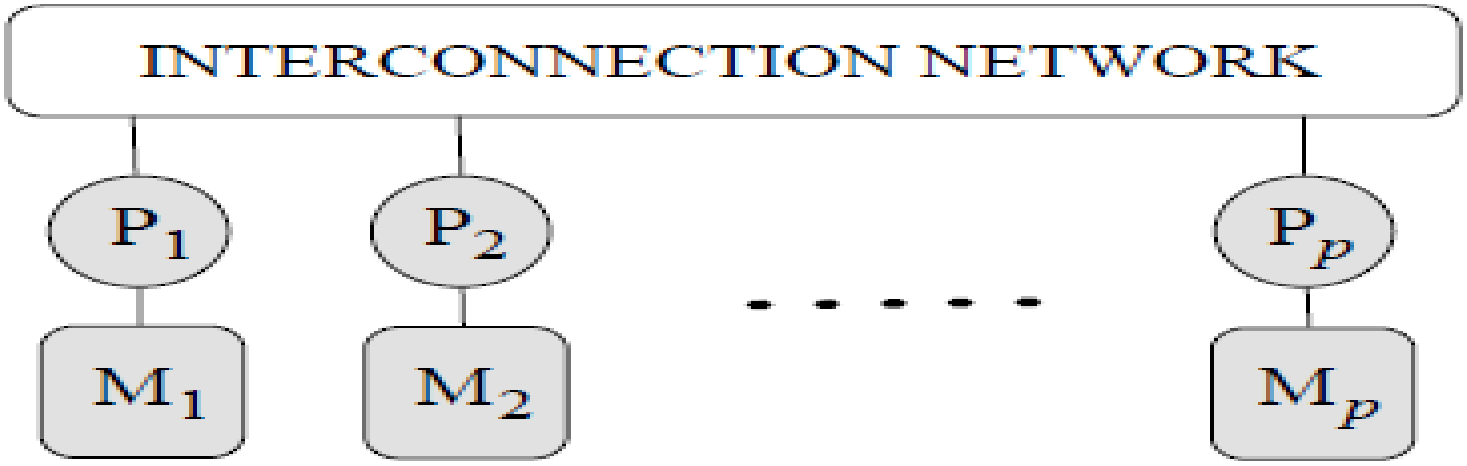
- PRAM model is unrealistic
  - Does PRAM model is irrelevant for the purposes of practical implementation of parallel computation?
    - Answer depends on what we expect from the PRAM model
- Design an algorithm for solving a problem P on PRAM
  - Efforts may not end up with a practical algorithm, ready for solving P.
  - Design may reveal something inherent to P, namely, that P is parallelizable.
  - Design may detect in P subproblems some of which could, at least in principle, be solved in parallel.

Relevance of PRAM is reflected in the following method:

- Design a program P for solving P on the model CRCW-PRAM(p), where p may depend on the problem P.
  - Design of P for CRCW-PRAM is expected to be easier than the design for EREW-PRAM,
    - CRCW-PRAM has no simultaneous-access restrictions to be taken into account.

Local-Memory Machine

- LMM model has p processing units, each with its own local memory .
- Processing units are connected to a common interconnection network.
- Each processing unit can access its own local memory directly.
- Processing units can access a non-local memory only by sending a memory request through the interconnection network.

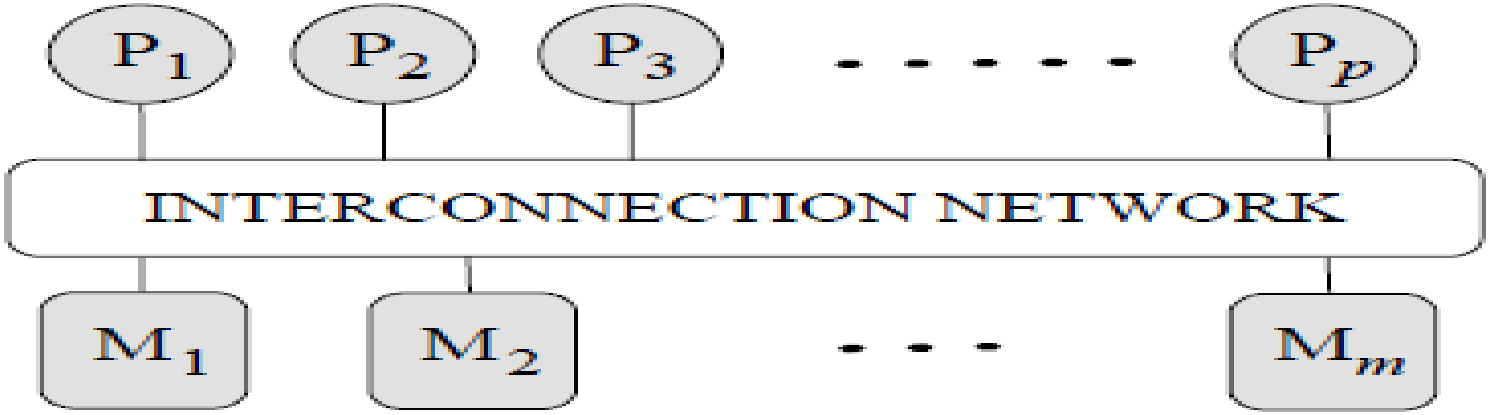


Local-Memory Machine

- All local operations, including accessing the local memory, take unit time.
- Time required to access a non-local memory depends on
  - Capability of the interconnection network
  - Pattern of coincident non-local memory accesses of other processing units as the accesses may congest the interconnection network.

Memory-Module Machine

- MMM model consists of p processing units and m memory modules
- Each of which can be accessed by any processing unit via a common interconnection network.
- No local memories to processing units.
- Processing unit can access the memory module by sending a memory request through the interconnection network.



Memory-Module Machine

- Processing units and memory modules are arranged in such a way that
  - No coincident accesses
  - Time for any processing unit to access any memory module is roughly uniform.
- When there are coincident accesses, the access time depends on
  - Capability of the interconnection network and
  - Pattern of coincident memory accesses.

Multiprocessor Architectures

- Parallel computer architectures differ in how processors and memory are organized.

Shared Memory Multiprocessors

- UMA (Uniform Memory Access) – Equal memory access time for all processors.
- NUMA (Non-Uniform Memory Access) – Access time depends on memory location.
- Example: Modern multi-core CPUs.

Distributed Memory Multiprocessors

- Processors have private memory, communicate via message passing.
- Example: HPC clusters using MPI.

Hybrid Architectures

- Nodes are shared-memory multicore machines connected via a distributed network.
- Example: Supercomputers like Summit or Fugaku

Impact of communication

- LMM model and MMM model explicitly use interconnection networks to convey memory requests to the non-local memories
- Focus
  - Role of an interconnection network in a multiprocessor model and its impact on the parallel time complexity of parallel algorithms.

Interconnection networks

- Experiments have shown that execution times of most real-world parallel applications are becoming increasingly dependent on the communication time rather than on the calculation time.
- As the number of cooperating processing units increases
  - performance of interconnection networks is becoming more important than the performance of the processing unit.
- Interconnection network has great impact on the efficiency and scalability of a parallel computer.
- High performance of an interconnection network may ultimately reflect in higher speedups,
  - interconnection network
    - can shorten the overall parallel execution time
    - increase the number of processing units that can be efficiently exploited.

Interconnection networks

Performance of an interconnection network depends on following factors.

- Routing
  - Routing is the process of selecting a path for traffic in an interconnection network;
- Flow-control algorithms
  - Flow control is the process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver;
- Network topology.
  - Network topology is the arrangement of the various elements, such as communication nodes and channels, of an interconnection network.
  - Performance increase can be expected to come from the improvements in the topology of interconnection networks

Basic properties of interconnection networks

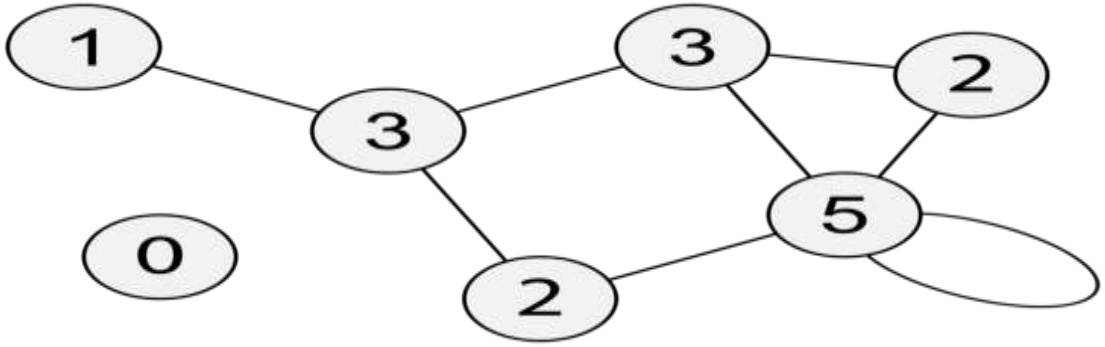
- Classify interconnection networks in many ways and characterize them by various parameters.
- Interconnection network can be modeled as a graph G(N,C)
  - N is a set of communication nodes
  - C is a set of communication links (or, channels) between the communication nodes.
- Graph-theoretical view of interconnection networks ,we can define parameters that represent
  - Topological properties
  - Performance properties

Topological properties of interconnection networks

- Node degree
- Regularity
- Symmetry
- Diameter
- Path diversity
- Expansion scalability.

Node Degree

- Node degree is the number d of channels through which a communication node is connected to other communication nodes.
- Node degree includes only the ports for the network communication and ports for service or maintenance channels.

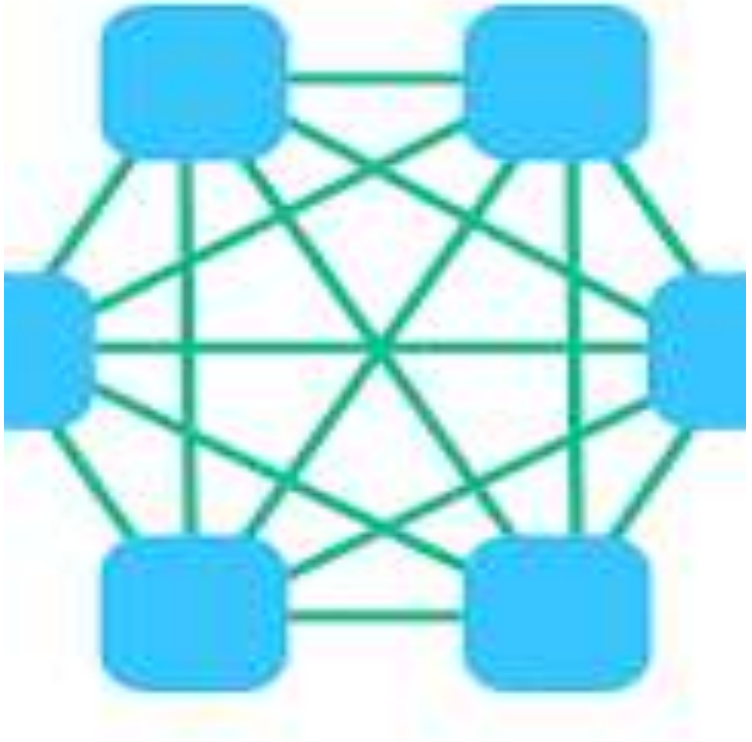


Regularity

- Interconnection network is said to be regular if all communication nodes have the same node degree;
  - d >0 such that every communication node has node degree d.

Symmetry

- Interconnection network is said to be symmetric if all communication nodes possess the "same view" of the network;
  - Homomorphism that maps any communication node to any other communication node.
- In a symmetric interconnection network, the load can be evenly distributed through all communication nodes, thus reducing congestion problems.
- Many real implementations of interconnection networks are based on symmetric regular graphs
  - Because of their fruitful topological properties that lead to a simple routing and fair load balancing under the uniform traffic.



Diameter

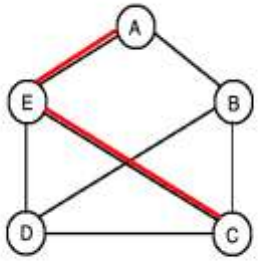
- In order to move from a source node to a destination node
  - Packet must traverse through a series of elements, such as routers or switches
  - together comprise a path between the source and the destination node.
- Hop count.
  - The number of communication nodes traversed by the packet

In the best case, two nodes communicate through the path which has the minimum hop count, l

l may vary with the source and destination nodes,

we also use the average distance, l_{avg},

Average taken over all possible pairs of nodes.



Important characteristic of any topology is the diameter, l_{max},

Maximum of all the minimum hop counts, taken over all pairs of source and destination nodes

Path diversity

- In an interconnection network, there may exist multiple paths between two nodes.
  - Nodes can be connected in many ways.
- Packet starting at source node will have at its disposal multiple routes to reach the destination node.
- The packet can take different routes depending on the current situation in the network.
- Interconnection network that has high **Path diversity** offers more alternatives when packets need to seek their destinations and/or avoid obstacles.

Scalability

- Scalability
  - capability of a system to handle a growing amount of work,
  - Potential of the system to be enlarged to accommodate that growth.
- For example,
  - Basic building block must be easily connected to other blocks in a uniform way.
  - Same building block must be used to build interconnection networks of different sizes, with only a small performance degradation
- Interconnection networks have important impact on scalability of parallel computers that are based on the LMM or MMM multiprocessor model.

Performance properties of interconnection networks

- Channel bandwidth,
- Latency.

Channel bandwidth

- Channel bandwidth is the amount of data communicated through a channel in a given time.
- Channel bandwidth can be adequately determined by using a simple model of communication
  - communication time t_{comm}, needed to communicate given data through the channel,
  - $t_s + t_d$
    - start-up time t_s, needed to set-up the channel's software and hardware,
    - data transfer time td,
      - $t_d = m * t_w$,
      - Product of the number of words making up the data, m

      And time per one word, t_w.
    - Channel bandwidth is $1/t_w$.

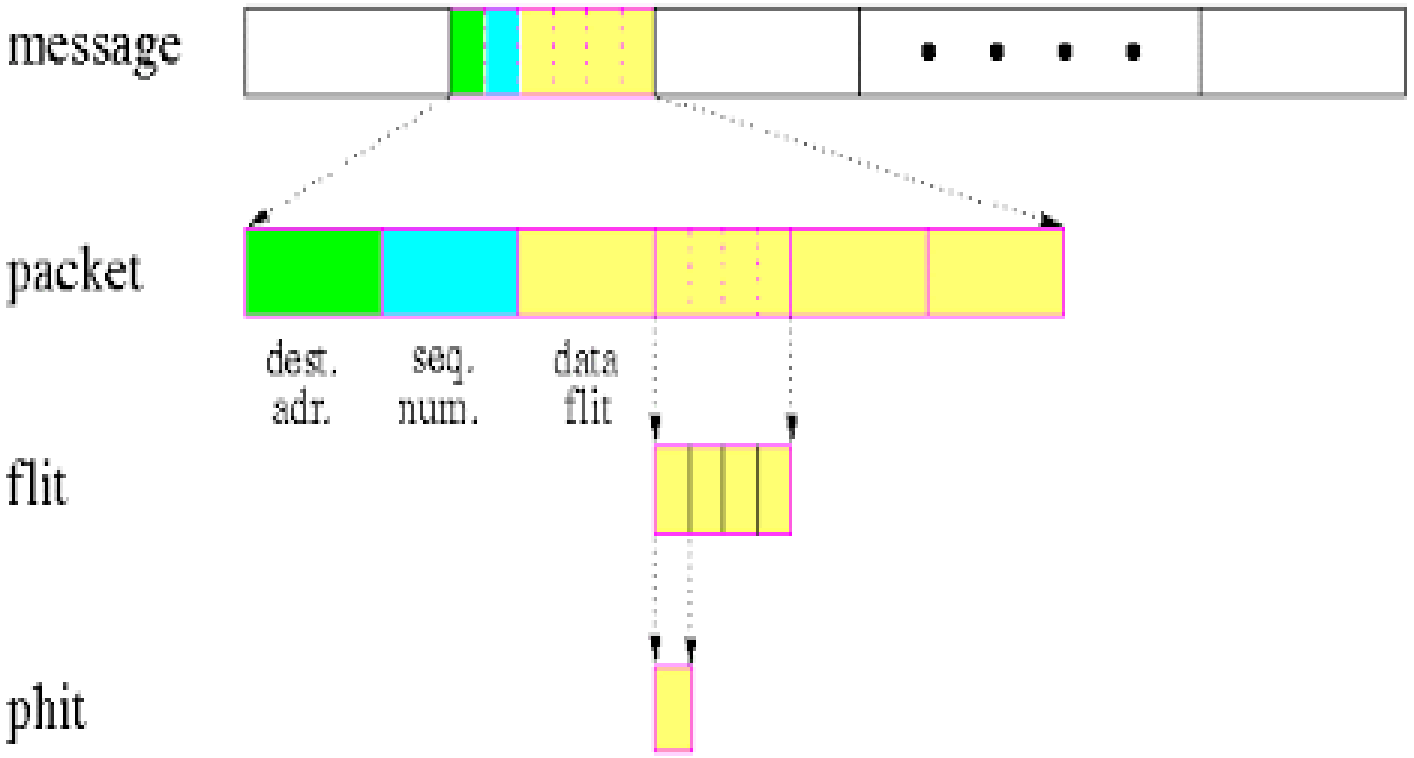
Latency

- Latency is the time required for a packet to travel from the source node to the destination node.
- Many applications, especially those using short messages, are latency sensitive
  - For such applications, their software overhead may become a major factor that influences the latency.
- latency is bounded by the time in which light traverses the physical distance between two nodes.

Performance properties of interconnection networks

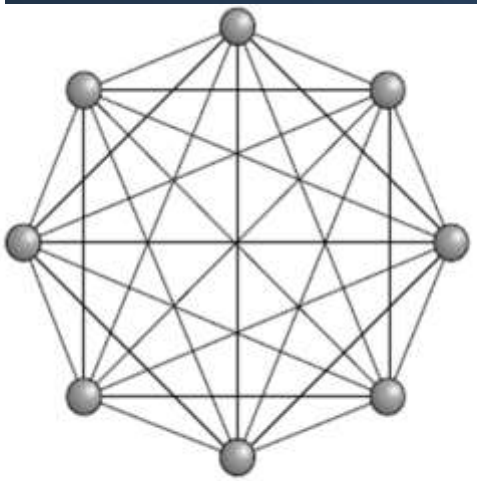
- Transfer of data from a source node to a destination node is measured in terms of various units
  - Packet, the smallest amount of data that can be transferred by hardware
  - FLIT (flow control digit), the amount of data used to allocate the buffer space in some flow-control techniques;
  - PHIT (physical digit), the amount of data that can be transferred in a single cycle.
- These units are closely related to the bandwidth and to the latency of the network.

Performance properties of interconnection networks



Classification of interconnection networks

- Direct networks
  - Network is said to be direct when each node is directly connected to its neighbors.
  - In a fully connected network, each of the n = |N| nodes is directly connected to all the other nodes,
    - so each node has n-1 neighbors.



A fully connected network with $n = 8$ nodes.

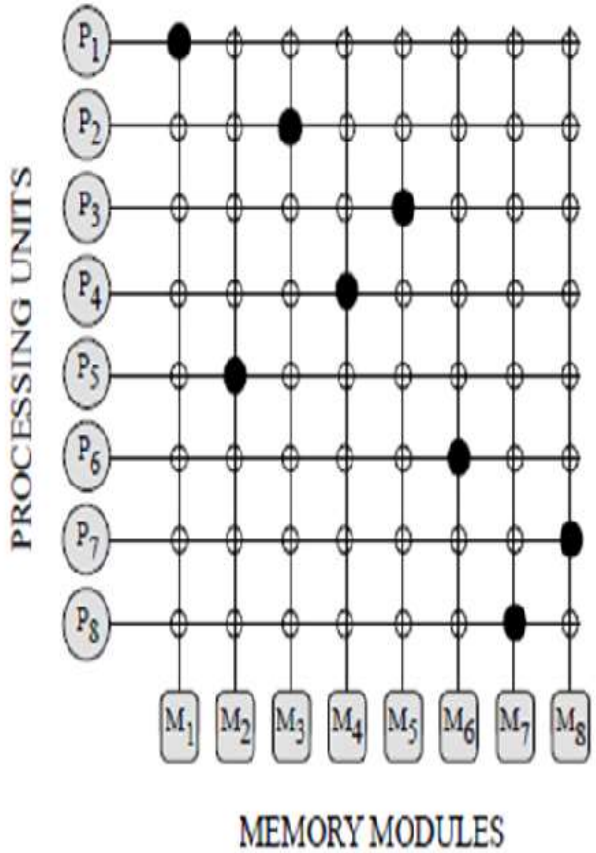
When n is large, each node is directly connected to a proper subset of other nodes.

While the communication to the remaining nodes is achieved by routing messages through intermediate nodes.

An example of such a direct interconnection network is the hypercube

Indirect networks

- Indirect network connects the nodes through switches.
  - Connects processing units on one end of the network
  - Memory modules on the other end of the network.
- Simplest circuit for connecting processing units to memory modules is the fully connected **crossbar switch**
  - Advantage is that it can establish a connection between processing units and memory modules in an arbitrary way.



A fully connected crossbar switch connecting 8 nodes to 8 nodes.

Indirect networks

- Intersection of a horizontal and vertical line is a crosspoint.
- Crosspoint is a small switch that can be electrically opened (o) or closed (•), depending on whether the horizontal and vertical lines are to be connected or not.
  - Ex- Eight crosspoints closed simultaneously, allowing connections between the pairs (P1, M1), (P2, M3), (P3, M5), (P4, M4), (P5, M2), (P6, M6), (P7, M8) and (P8, M7) at the same time.

Fully connected crossbar has too large complexity to be used for connecting large numbers of input and output ports.

Number of cross points grows as pm

where p and m are the numbers of processing units and memory modules, respectively.

For p = m = 1000 this amounts to a million crosspoints which is not feasible.

Indirect networks

- Indirect networks can be further classified as follows:
  - **Non-blocking** network can connect any idle source to any idle destination
    - This is due to the network topology which ensures the existence of multiple paths between the source and destination.
  - **Blocking rearrangeable** networks can rearrange the connections that have already been established across the network
    - Such a network can establish all possible connections between inputs and outputs.

Indirect networks

- In a **blocking** network
  - Connection that has been established across the network may block the establishment of a new connection between a source and destination
  - Such a network cannot always provide a connection between a source and an arbitrary free destination

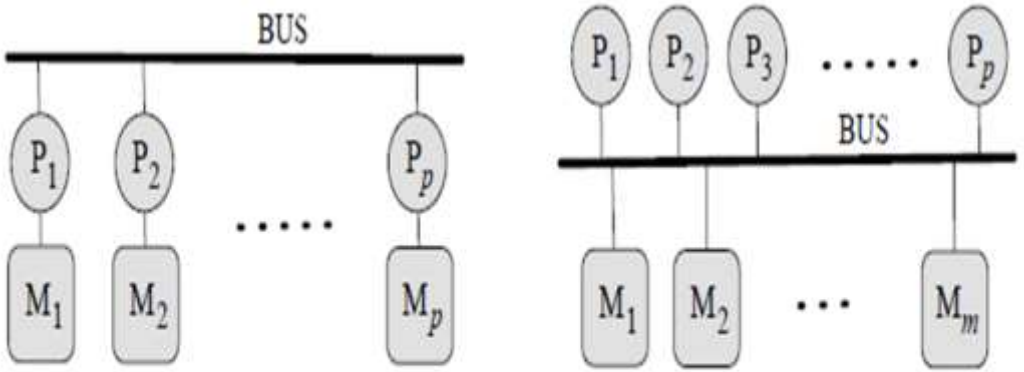
Topologies of interconnection networks

- Not every network topology is capable of conveying memory requests quickly enough to efficiently back up parallel computation.
- Network topology has a large influence on the performance of the interconnection network and, consequently, of parallel computation.
- Network topology may incur considerable difficulties in the actual construction of the network and its cost.

Various network topologies

- Bus
- Mesh
- 3D-mesh
- Torus
- Hypercube
- Multistage network
- Fat tree

The bus



The bus is the simplest network topology.

The bus

- Memory-module machine a processing unit wants to read a memory word,
  - it must first check to see if the bus is busy.
- If the bus is idle
  - processing unit puts the address of the desired word on the bus
  - issues the necessary control signals
  - waits until the memory puts the desired word on the bus.
- If the bus is busy, the processing unit must wait until the bus becomes idle.
- If there is a small number of processing units, say two or three, the contention for the bus is manageable;
- For larger numbers of processing units, say 32, the contention becomes unbearable
  - because most of the processing units will wait most of the time.
- To solve this problem we add a local cache to each processing unit.

The Bus

- Advantage
  - Simple to build,
  - Relatively easy to develop protocols that allow processing units to cache memory values locally
- Disadvantage of using a bus is that the processing units must take turns accessing the bus.
  - Average time to perform a memory access grows proportionately with the number of processing units.

The ring

Ring is among the simplest and the oldest interconnection networks.

Given n nodes, arranged in linear fashion so that each node has a distinct label i,

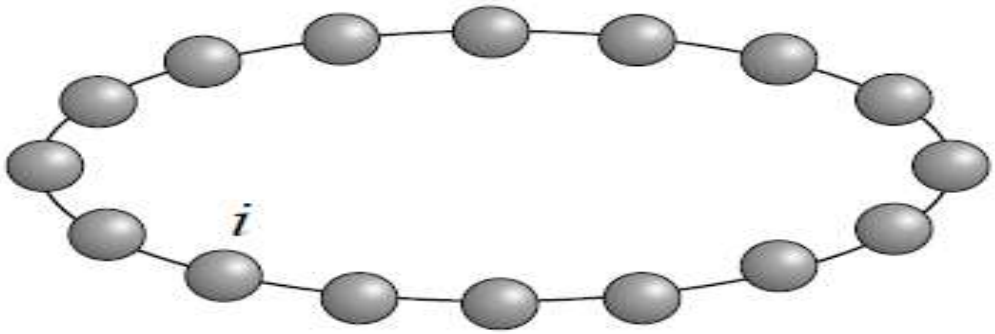
where 0 <= i <= n-1.

Every node is connected to two neighbors, one to the left and one to the right.

node labeled i is connected to the nodes labeled i+1 mod n and

i-1mod n

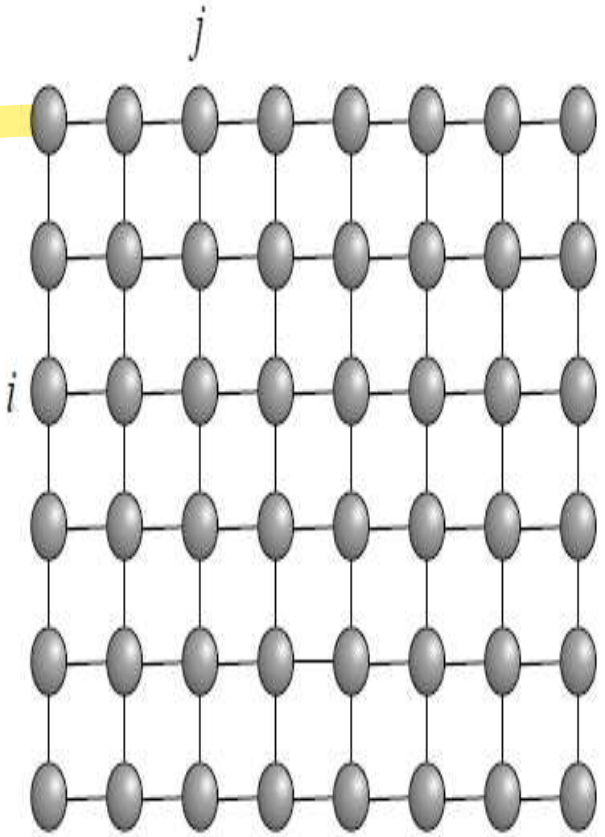
The ring is used in local-memory machines (LMMs).



: A ring. Each node represents a processor unit with local memory

2D-mesh

- Two-dimensional mesh is an interconnection network that can be arranged in rectangular fashion
  - so that each switch in the mesh has a distinct label (I, j)
  - 0 <= i <= X-1
  - 0 <= j <= Y-1
  - values X and Y determine the lengths of the sides of the mesh.
  - Number of switches in a mesh is XY.

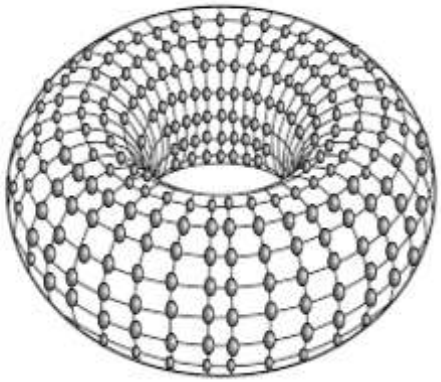


A 2D-mesh. Each node represents a processor unit with local memory

Meshes typically appear in local-memory machines (LMMs)
Processing unit (along with its local memory) is connected to each switch
Remote memory accesses are made by routing messages through the mesh.

2D-torus (toroidal 2D-mesh)

- Switches on the sides have no connections to the switches on the opposite sides.
  - Interconnection network that compensates for this is called the toroidal mesh, or just torus when d = 2.
- In torus every switch located at (I , j) is connected to four other switches, which are located at (I, j+1 mod Y), (I , j-1 mod Y), (i+1 mod X , j) and (i-1 mod X , j).



1: A 2D-torus. Each node represents a processor unit with local memory

Toruses appear in local-memory machines (LMMs):

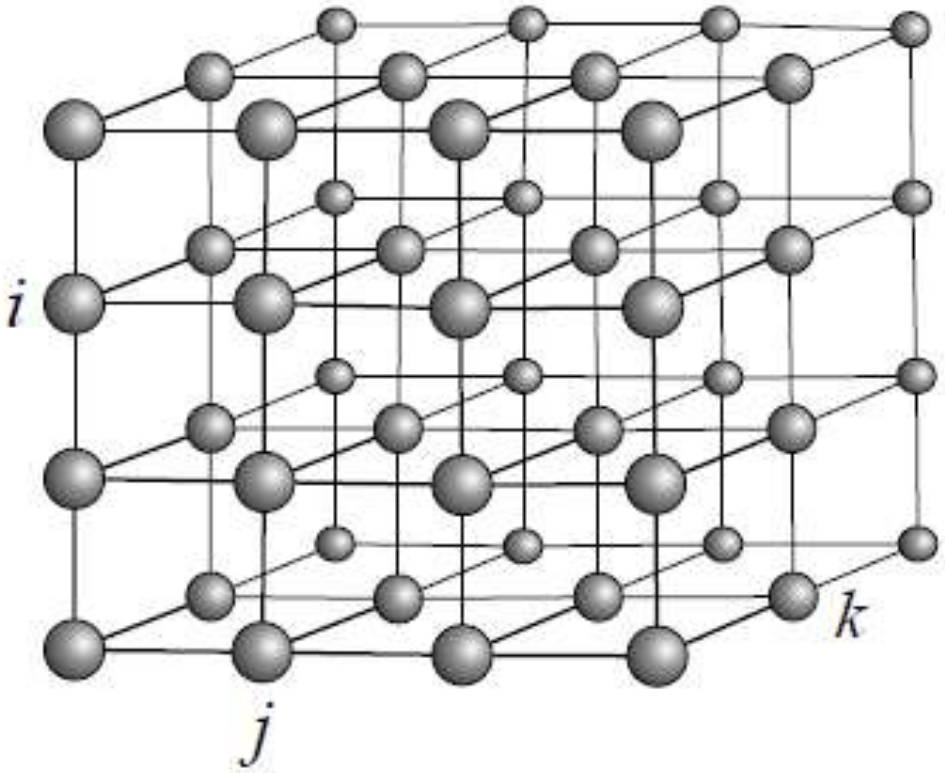
Each switch is connected a processing unit with its local memory.

Each processing unit can access any remote memory by routing messages through the torus.

3D-mesh and 3D-torus

- Three-dimensional mesh is similar to two-dimensional.
- Now each switch in a mesh has a distinct label (I , j , k),
  - where 0 <= i <= X - 1, 0 <= j <= Y - 1, and 0 <= k <= Z -1.
  - X, Y and Z determine the lengths of the sides of the mesh, so the number of switches in it is XYZ.
- Every switch, except those on the sides of the mesh, is now connected to six neighbors:
  - one to the north, one to the south, one to the east, one to the west, one up, and one down.
  - Switch labeled (i; j; k), where 0 < i < X -1, 0 < j <Y -1 and 0 < k < Z -1, is connected to the switches (I , j+1,k), (I, j-1, k), (i+1, j, k), (i-1, j, k), (I ,j, k+1) and (I, j ,k-1).
- Typically appear in LMMs.

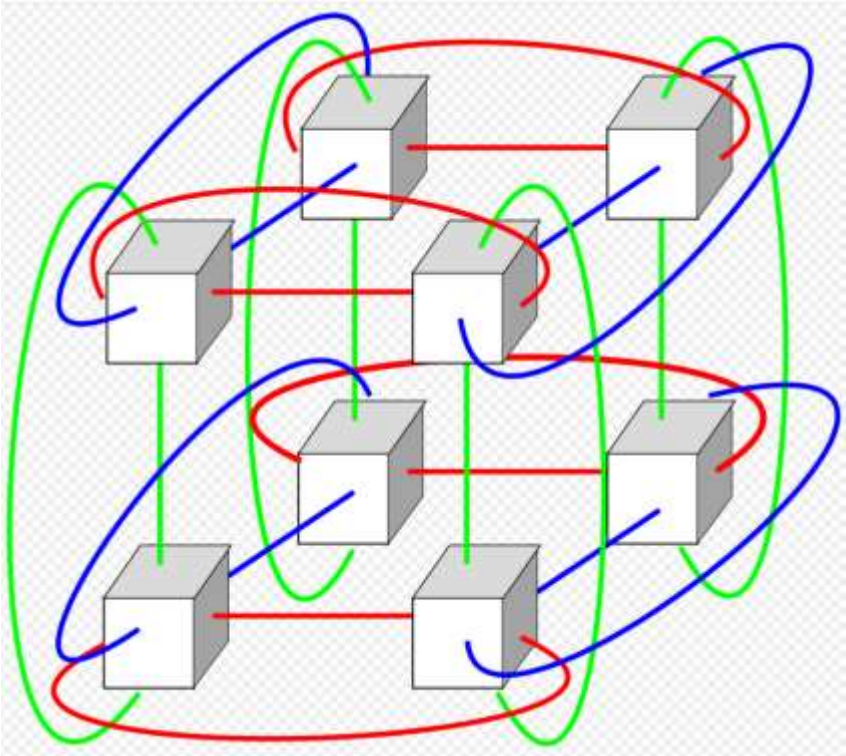
3D-mesh and 3D-torus



: A 3D-mesh. Each node represents a processor unit with local memory

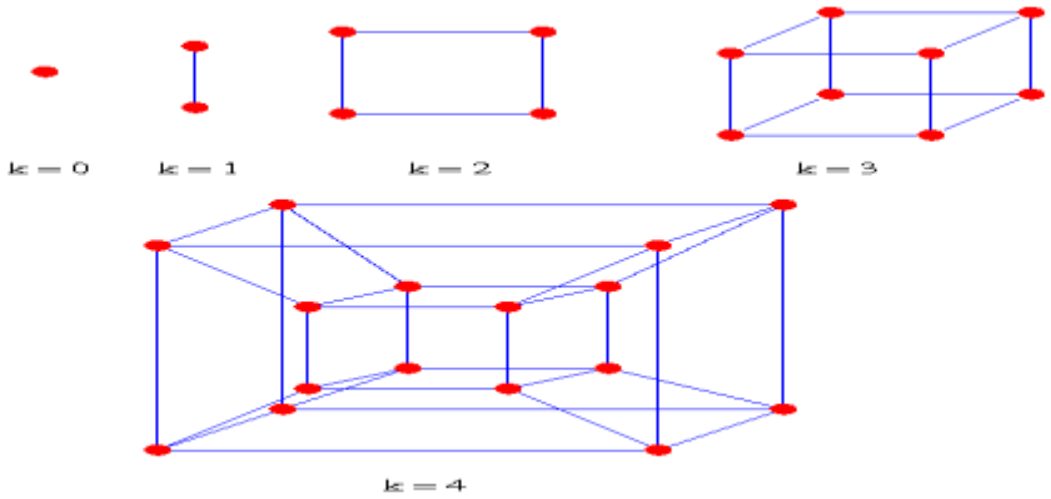
3D-mesh and 3D-torus

- Expand a 3D-mesh into a toroidal 3D-mesh by adding edges that connect nodes located at the opposite sides of the 3D-mesh.
- Switch labeled (i, j, k) is connected to the switches (i+1modX, j ,k), (i-1modX, j, k), (i, j+1modY, k), (i, j-1modY, k), (i, j ,k+1modZ) and (i , j ,k -1modZ).

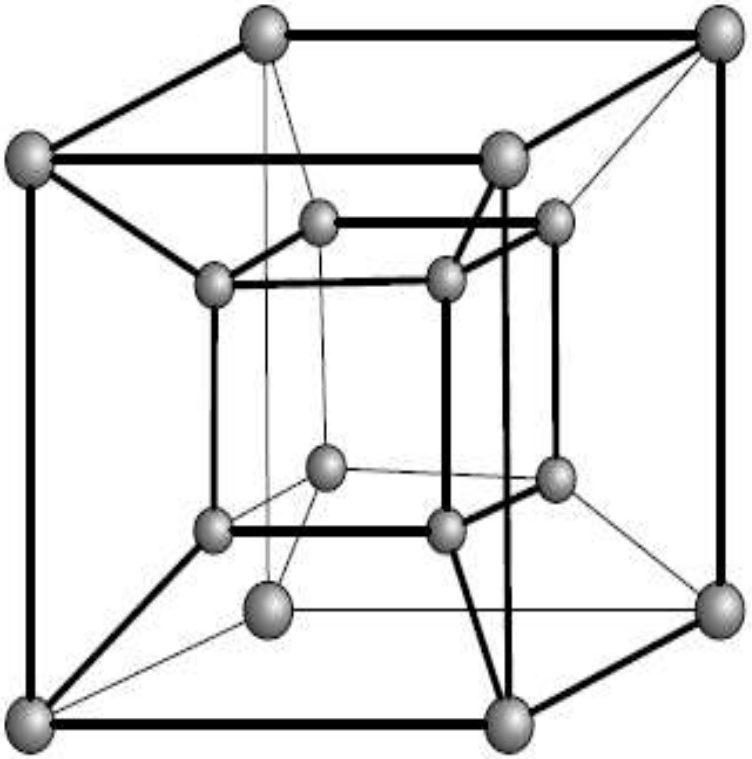


Hypercube

- Hypercube is an interconnection network that has n = 2 ^b nodes, for some b > 0.
- Each node has a distinct label consisting of b bits.
- Two nodes are connected by a communication link if and only if their labels differ in precisely one bit location.
- Each node of a hypercube has $b = \log_2 n$ neighbors.
- Hypercubes are used in local-memory machines (LMMs).



Hypercube



A hypercube. Each node represents a processor unit with local memory

The k-ary d-cube family of network topologies

- Ring, 2D-torus, 3D-torus, hypercube, etc - all belong to one larger family of k-ary d-cube topologies
- Given k > 1 and d > 1, the k-ary d-cube topology is a family of certain "gridlike" topologies that share the fashion in which they are constructed.
  - k-ary d-cube topology is a generalization of certain topologies.

The k-ary d-cube family of network topologies

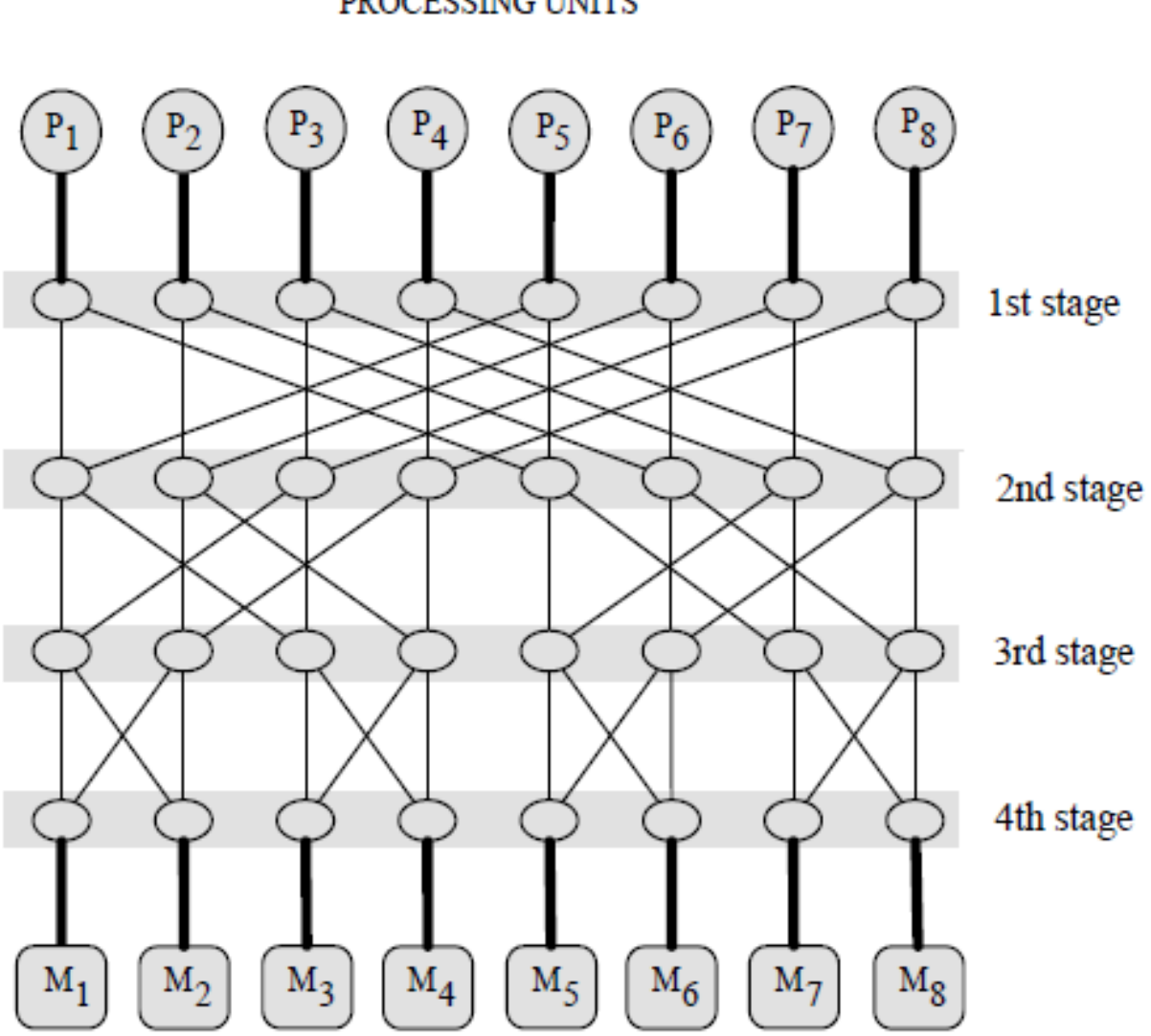
- Parameter d is called the dimension of these topologies
- k is their side length, the number of nodes along each of the d directions.
- k-ary d-cube is constructed from k other k-ary (d-1)-cubes by connecting the nodes with identical positions into rings.

$$n = k^d \text{ communication nodes}$$

$$c = dn = dk^d \text{ communication links.}$$

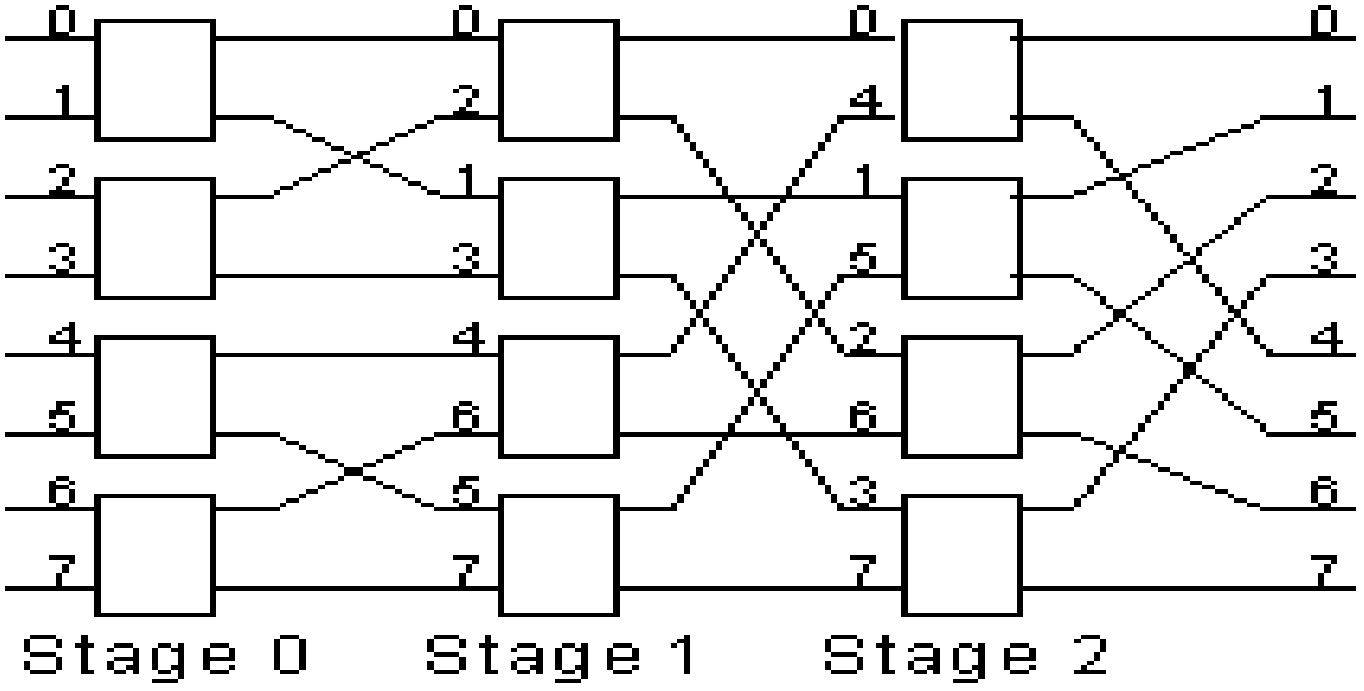
Multistage network

- Multistage network connects one set of switches, called the input switches, to another set, called the output switches.
- Network achieves this through a sequence of stages, where each stage consists of switches.
- Input switches form the first stage, and the output switches form the last stage.
- Number d of stages is called the depth of the multistage network.
- Multistage network allows to send a piece of data from any input switch to any output switch.
  - This is done along a path that traverses all the stages of the network in order from 1 to d.


PROCESSING UNITS
P1
P2
P3
P4
P5
P6
P7
P8
1st stage
2nd stage
3rd stage
4th stage
M1
M2
M3
M4
M5
M6
M7
M8
MEMORY MODULES

Multistage network

- Multistage networks are frequently used in memory-module machines (MMMs);
- Processing units are attached to input switches
- Memory modules are attached to output switches.



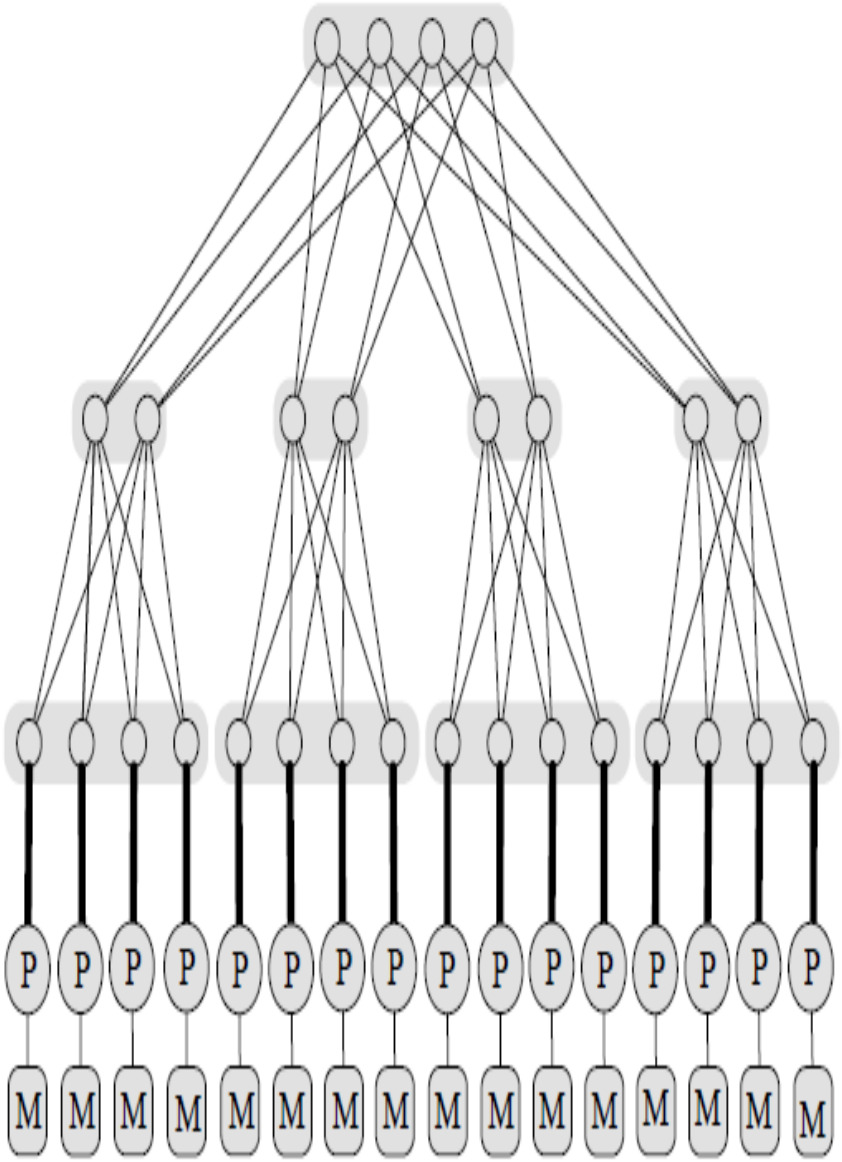
Fat tree

- Fat tree is a network whose structure is based on that of a tree.
  - Edges that are nearer the root of the tree are "fatter" than edges that are further down the tree.
- Each node of a fat tree may represent many network switches
- Each edge may represent many communication channels.
- More channels an edge represents, the larger is its capacity and the fatter is the edge.
  - Capacities of the edges near the root of the fat tree are much larger than the capacities of the edges near the leaves.

Fat trees can be used to construct local-memory machines (LMMs):

Processing units along with their local memories are connected to the leaves of the fat tree

Message from one processing unit to another first travels up the tree to the least common ancestor of the two processing units and then down the tree to the destination processing unit.



PROCESSING UNITS WITH LOCAL MEMORIES

Parallel computational complexity- Problem instances and their sizes

- Let P be a computational problem.
- Instance is obtained from P by replacing the variables in the definition of P with actual data.
  - Instance of a problem is all the inputs needed to compute a solution to the problem.
- problem P can be viewed as a set of all the possible instances of P.

Problem instances and their sizes

- To each instance pi of P
  - size of the instance pi and denote by size(pi):
- size(pi) is roughly the amount of space needed to represent pi in some way accessible to a computer. and, in practice, depends on the problem P.
- For example
  - if we choose P "sort a given finite sequence of numbers," pi "sort 0 9 2 7 4 5 6 3" is an instance of P and size(p) = 8
  - If P is a problem about graphs, then the size of an instance of P is often defined as the number of nodes in the actual graph.

Problem instances and their sizes

- Why do we need sizes of instances?
  - how fast an algorithm A for a problem P is?
  - how A's execution time depends on the size of instances of P that are input to A.?
- T(n)
  - Represent the execution time of A on instances of size n.
- Rate of growth of T(n),
  - how quickly T(n) grows when n grows.

Number of processing units vs. size of problem instances

- P
- C(p)
- Tpar
- Tseq
- n

$$S(n) \stackrel{\text{def}}{=} \frac{T_{\text{seq}}(n)}{T_{\text{par}}(n)},$$

$$E(n) \stackrel{\text{def}}{=} \frac{S(n)}{p}.$$

Number of processing units vs. size of problem instances

- solving instances of P whose size is n.
  - too few processing units in C(p), i.e., p is too small,
    - potential parallelism in the program P will not be fully exploited during the execution of P on C(p),
    - reflect in low speedup S(n) of P.
  - C(p) has too many processing units, i.e., p is too large,
    - some of the processing units will be idling during the execution of the program P,
    - reflect in low speedup of P.
- This raises the following question
  - How many processing units p should have C(p), so that, for all instances of P of size n, the speedup of P will be maximal?

Number of processing units vs. size of problem instances

- if we let n grow then p must grow too;
  - otherwise, p would eventually become too small relative to n,
  - thus making C(p) incapable of fully exploiting the potential parallelism of P.
- Consequently, we may view p, the number of processing units that are needed to maximize speedup,
  - to be some function of n
  - p = f (n);

where $f : \mathbb{N} \to \mathbb{N}$ is some *nondecreasing* function, i.e., $f(n) \leqslant f(n+1)$, for all n.

Number of processing units vs. size of problem instances

- If f (n) grows exponentially.
  - if there are exponentially many processing units in a parallel computer
    - incurs long communication paths between some of them.
  - Some communicating processing units become exponentially distant from each other
  - Communication times between them increase correspondingly
  - eventually, blemish the theoretically achievable speedup.
- Exponential number of processing units is impractical and leads to theoretically tricky situations.

Number of processing units vs. size of problem instances

- poly(n) and exp(n) are a polynomial and an exponential function, respectively,
- poly(n) < exp(n)
  - poly(n) is eventually dominated by exp(n)
- polynomial function poly(n) asymptotically grows slower than an exponential function exp(n).
- f (n) = poly(n)
  - p = poly(n);

The class NC(Nick's Class) of efficiently parallelizable problems

- Let P be an algorithm for solving a problem P on CRCW-PRAM(p).
- The execution of P on EREW-PRAM(p) will be at most O(log p) times slower than on CRCW-PRAM(p).
- Observations from previous section p = poly(n).
  - log p = log poly(n) = O(log n)
  - This means that for p = poly(n) the execution of P on EREW-PRAM(p) will be at most O(logn)-times slower than on CRCW-PRAM(p).
- Execution time of a program does not vary too much as we choose the variant of PRAM that will execute it.

Class NC of efficiently parallelizable problems

Definition 2.1. A function is **polylogarithmic** in n if it is polynomial in $\log n$, i.e., if it is $a_k(\log n)^k + a_{k-1}(\log n)^{k-1} + \cdots + a_1(\log n)^1 + a_0$, for some $k \geqslant 1$.

Definition 2.2. Let **NC** be the class of computational problems solvable in polylogarithmic time on PRAM with polynomial number of processing units.

The class NC of efficiently parallelizable problems

- If a problem P is in the class NC, then it is solvable in polylogarithmic parallel time with polynomially many processing units regardless of the variant of PRAM used to solve P.
- In other words, the class NC is robust, insensitive to the variations of PRAM.
- If we replace one variant of PRAM with another,

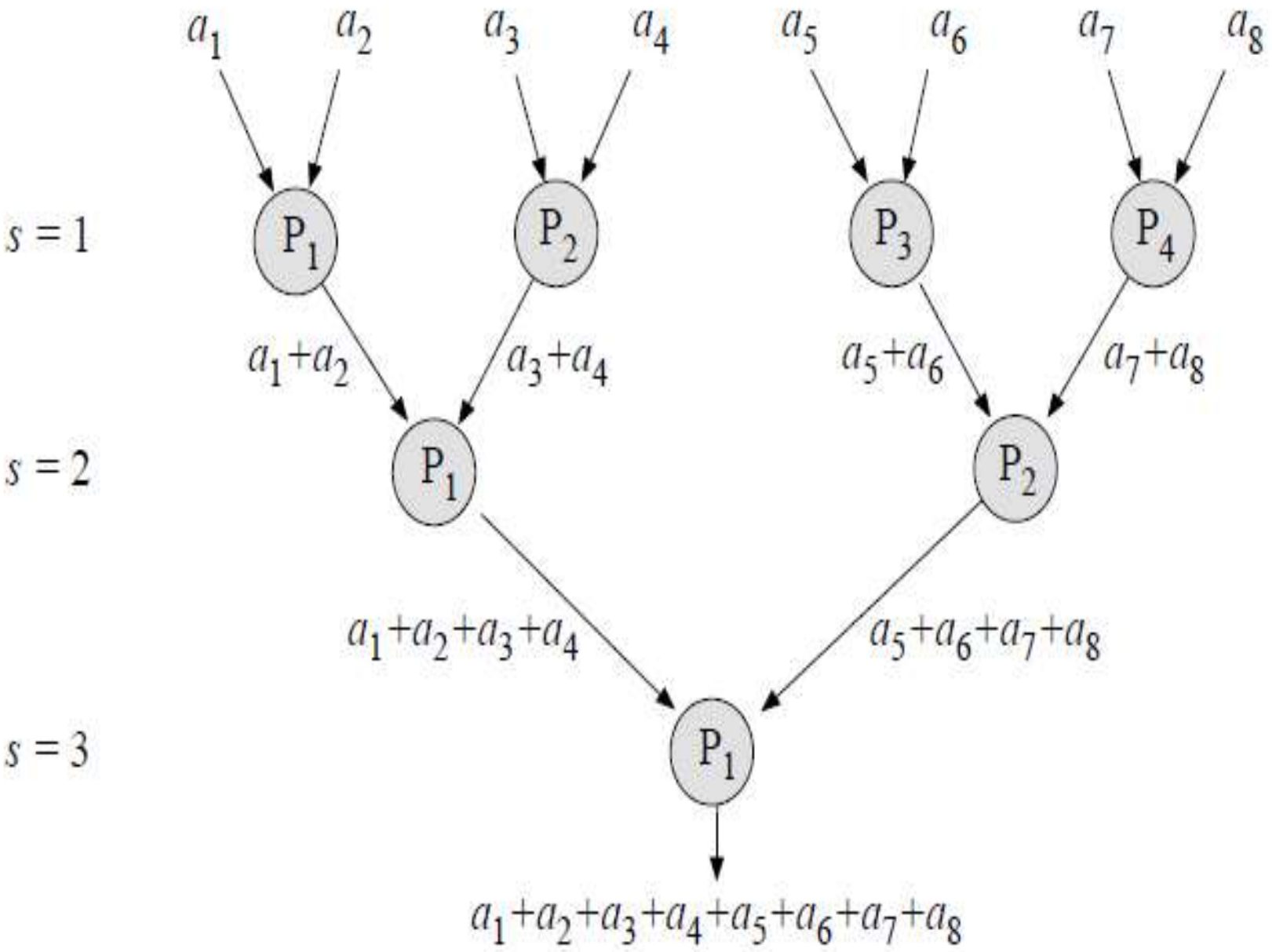
by Theorem 2.1 Π's parallel execution time $O(\log^k n)$ can only increase by a factor $O(\log n)$ to $O(\log^{k+1} n)$ which is still polylogarithmic.

The class NC of efficiently parallelizable problems

- In sum, NC is the class of efficiently parallelizable computational problems.
- Example
  - Given the problem P - “add n given numbers.”
  - pi -“add numbers 10, 20, 30, 40, 50, 60, 70, 80” is an instance of size(p) = 8 of the problem P.
  - Let us now focus on all instances of size 8, that is, instances of the form p - “add numbers a1,a2,a3,a4,a5,a6,a7.a8.”

Example

- The fastest sequential algorithm for computing the sum a1+a2+a3+a4+a5+ a6+a7+a8 requires Tseq(8) = 7 steps
- In parallel, however, the numbers a1,a2,a3,a4,a5,a6,a7,a8 can be summed in just Tpar(8) = 3 parallel steps using 8/2 = 4 processing units which communicate in a tree-like pattern


a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
$s = 1$
P1 P2 P3 P4
a_1+a_2 a_3+a_4 a_5+a_6 a_7+a_8
$s = 2$
P1 P2
$a_1+a_2+a_3+a_4$ $a_5+a_6+a_7+a_8$
$s = 3$
P1
$a_1+a_2+a_3+a_4+a_5+a_6+a_7+a_8$

The class NC of efficiently parallelizable problems

In general, instances $\pi(n)$ of Π can be solved in parallel time $T_{par} = \lceil \log n \rceil = O(\log n)$ with $\lceil \frac{n}{2} \rceil = O(n)$ processing units communicating in similar tree-like patterns. Hence, $\Pi \in$ NC and the associated speedup is $S(n) = \frac{T_{seq}(n)}{T_{par}(n)} = O(\frac{n}{\log n})$. □

$$E(n) = O(\frac{1}{\log n}).$$

Laws and theorems of parallel computation - Brent's theorem

- Brent's theorem enables us to quantify the performance of a parallel program when the number of processing units is reduced.
- Let M be a PRAM of an arbitrary type and containing unspecified number of processing units.
- When a parallel program P is run on M, different numbers of operations of P are performed, at each step, by different processing units of M.

Brent's Theorem (Work-Time Scheduling)

- Brent's theorem helps us understand how much parallel time an algorithm will take if we know:
  - Work (W): The total number of operations (as if run sequentially).
  - Depth (T_∞): The length of the critical path (minimum time if unlimited processors are available).
  - p: The number of processors available.

Theorem Statement

- The running time T_p of a parallel algorithm on p processors is:

$$T_p \leq \frac{W}{p} + T_\infty$$

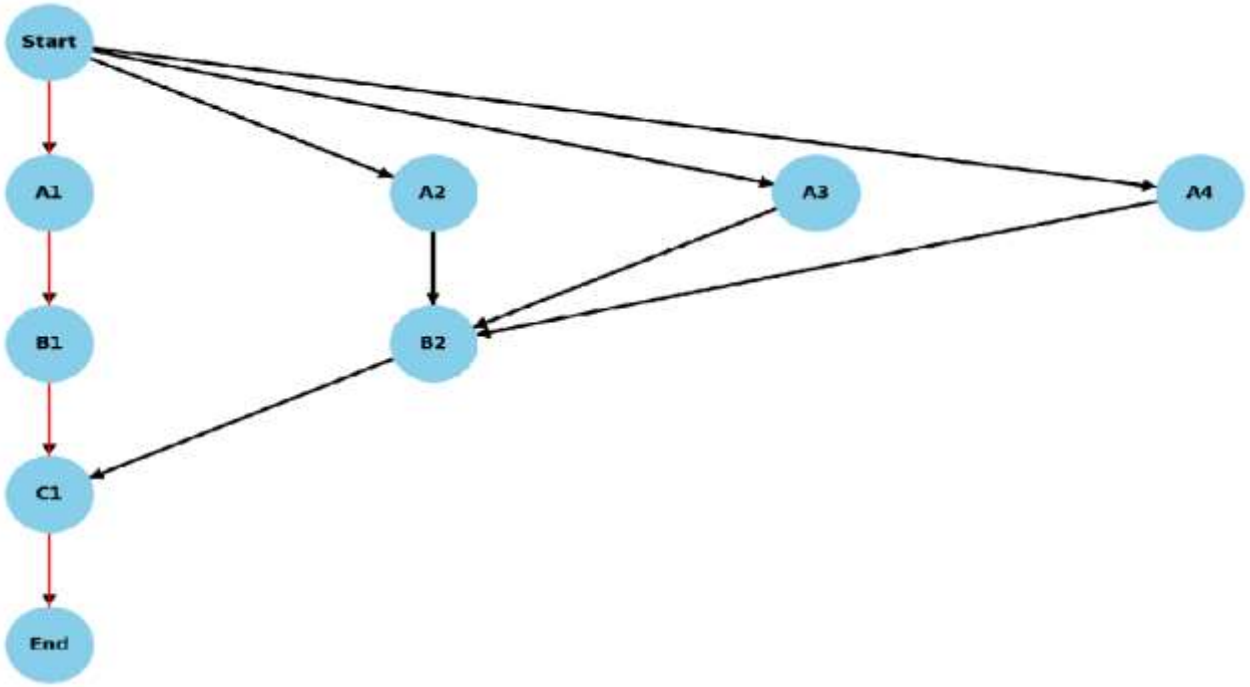
1. $\frac{W}{p}$ = **work per processor** if load is evenly balanced.
2. T_∞ = **inherent sequential bottleneck** (cannot be parallelized).
3. The runtime is bounded by these two factors.

Here's a **DAG (Directed Acyclic Graph) illustration for Brent's theorem**:

- Each node represents a computational task.
- Black edges show task dependencies.
- The **red path** is the *critical path* (the longest sequential chain of dependent tasks).

Brent's Theorem states that:

If a computation requires **W work** (total operations) and has **D depth** (longest path length), then on **P processors** it can be executed in time



Example 1 — Work-dominated (bound is tight-ish)

DAG idea: 100 independent leaf tasks, then 1 final combine task.

- Work $W = 100 + 1 = 101$ unit operations
- Depth $D = 2$ (leaves → combine)
- Processors $P = 8$

Brent's bound

$$T_P \le \frac{W}{P} + D = \frac{101}{8} + 2 \approx 14.625$$

A feasible schedule

- Distribute 100 leaves across 8 processors: $\lceil 100/8 \rceil = 13$ steps
- Final combine: $+1$ step
- **Actual $T_P \approx 14$ steps**, which satisfies the bound.

Speedup & efficiency

$$S = \frac{W}{T_P} \approx \frac{101}{14} \approx 7.21, \quad E = \frac{S}{P} \approx \frac{7.21}{8} \approx 0.90$$

(High efficiency because W/P dominates and D is small.)

Example 2 — Depth-dominated (parallelism limited by critical path)

DAG idea: 45 independent leaf tasks, then a **chain of 10 dependent tasks.**

- Work $W = 45 + 10 = 55$
- Depth $D = 10$
- Processors $P = 8$

Brent's bound

$$T_P \le \frac{55}{8} + 10 \approx 6.875 + 10 = 16.875$$

A feasible schedule

- Do the 45 leaves in $\lceil 45/8 \rceil = 6$ steps
- Then the 10-step chain must run sequentially: $+10$
- **Actual $T_P \approx 16$**, which is below the bound (bound isn't tight).

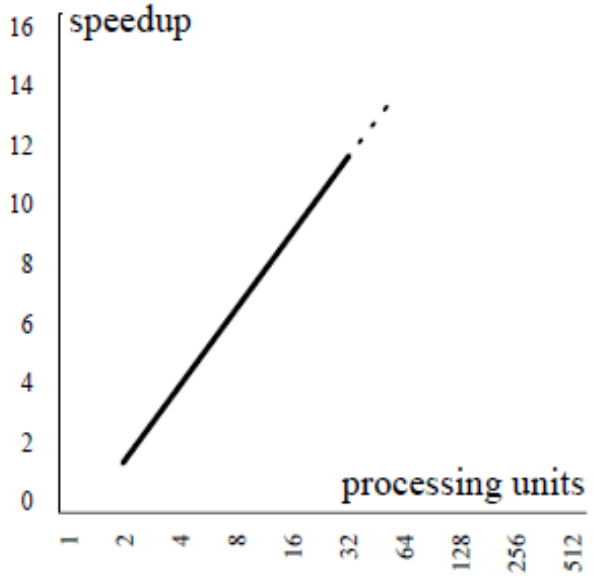
Speedup & efficiency

$$S = \frac{55}{16} \approx 3.44, \quad E = \frac{3.44}{8} \approx 0.43$$

(Lower efficiency because the critical path D dominates.)

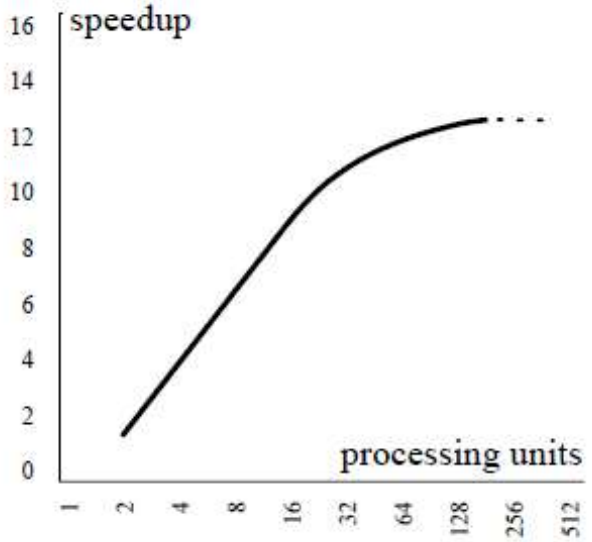
Amdahl's law

- We would expect that
  - doubling the number of processing units should halve the parallel execution time.
  - doubling the number of processing units again should halve the parallel execution time once more.
- In other words, we would expect that the speedup from parallelization is a linear function of the number of processing units



Amdahl's law

- Linear speedup from parallelization is just a desirable optimum which is not very likely to become a reality.
  - In reality very few parallel algorithms achieve it.
- Most of parallel programs have a speedup which is near-linear for small numbers of processing elements
  - then flattens out into a constant value for large numbers of processing elements



Amdahl's Law gives the **theoretical maximum speedup** of a program when part of it is parallelized.

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

Where:

- f = fraction of program that is **parallelizable**
- 1−f = fraction that is **sequential**
- P = number of processors
- S(P) = speedup with PPP processors

Amdahl's Law

- Speedup in the Amdahl's Law is a function of three variables, P, b and s, so it would be more appropriately denoted by S(P, b, s).
  - b is the fraction of the time during which the sequential execution of P can benefit from multiple processing units.
- If multiple processing units are actually available and exploited by P, the part of P that exploits them is sped up by the factor s > 1.
- Since s is only the speedup of a part of the program P, the speedup of the whole P cannot be larger than s; specifically, it is given by S(P) of the Amdahl's Law.

Amdahl's Law

$$S < \frac{1}{1 - b},$$

- small part of the program which cannot be parallelized will limit the overall speedup available from parallelization

Amdahl's law at work

- Suppose that 70% of a program execution can be sped up if the program is parallelized and run on 16 processing units instead of one.

$$S(P) = \frac{1}{(1-f)+\frac{f}{P}} = \frac{1}{0.3+\frac{0.7}{s}} \leqslant \frac{1}{0.3+\frac{0.7}{16}} = 2.91.$$

Amdahl's law at work

- If we double the number of processing units to 32

$$S = \frac{1}{1 - b + \frac{b}{s}} = \frac{1}{0.3 + \frac{0.7}{s}} \leqslant \frac{1}{0.3 + \frac{0.7}{32}} = 3.11\,,$$

Amdahl's law at work

- if we double it once again to 64 processing units

$$S(P) = \frac{1}{(1-f) + \frac{f}{P}} = \frac{1}{0.3 + \frac{0.7}{s}} \leqslant \frac{1}{0.3 + \frac{0.7}{64}} = 3.22.$$

Amdahl's law at work

- if we double the number of processing units even to 128

$$S(P) = \frac{1}{(1-f)+\frac{f}{P}} = \frac{1}{0.3+\frac{0.7}{s}} \leqslant \frac{1}{0.3+\frac{0.7}{128}} = 3.27.$$

- In this case doubling the processing power only slightly improves the speedup.
- Therefore, using more processing units is not necessarily the optimal approach.

Thank you!!