

# Introduction to Parallel/Distributed Computing

# Types of Computing Systems

- ▣ There are several architectures which require a different OS:
  - Desktop PCs
  - Simple Batch System
  - Multiprogramming Batch System
  - Time sharing Systems
  - Parallel Systems
  - Distributed Systems
  - Clustered Systems
  - Real-time Systems
  - Embedded Systems
  - Handheld System

# Desktop Systems

- ❑ Earlier, CPUs and PCs lacked the features needed to protect an operating system from user programs.
- ❑ PC operating systems therefore were neither **multiuser** nor **multitasking**.
- ❑ Instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness. These systems are called **Desktop Systems**
  - Example: PCs running Microsoft Windows and Apple Macintosh.
- ❑ Operating systems for these computers have benefited in several ways from the development of operating systems for **mainframes**.
- ❑ **Microcomputers** were immediately able to adopt some of the technology developed for larger operating systems.
- ❑ The hardware costs for microcomputers are sufficiently **low** that individuals have sole use of the computer, and CPU utilization is no longer a prime concern.

# Simple Batch Systems

- In this type of system, there is no direct interaction between user and the computer.
- The user has to submit a job (written on cards or tape) to a computer operator.
- Then computer operator places a batch of several jobs on an input device.
- Jobs are batched together by type of languages and requirement.
- Then a special program, the monitor, manages the execution of each program in the batch.
- The monitor is always in the main memory and available for execution.
- Following are some disadvantages of this type of system:
  - No interaction between user and computer.
  - No mechanism to prioritize the processes.

# Multiprogramming Batch Systems

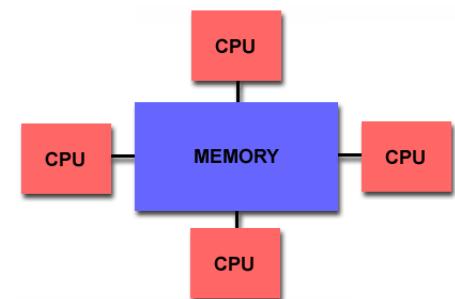
- In this the operating system picks up and begins to execute one of the jobs from memory.
- Once this job needs an I/O operation operating system switches to another job (CPU and OS always busy).
- Jobs in the memory are always less than the number of jobs on disk(Job Pool).
- If several jobs are ready to run at the same time, then the system chooses which one to run through the process of **CPU Scheduling**.
- In non-multiprogrammed system, there are moments when CPU sits idle and does not do any work.
- In Multiprogramming system, CPU will never be idle and keeps on processing.

# Time Sharing Systems

- ❑ **Time-Sharing Systems** are very similar to Multiprogramming batch systems. In fact time sharing systems are an extension of multiprogramming systems.
- ❑ In time sharing systems the prime focus is on minimizing the response time, while in multiprogramming the prime focus is to maximize the CPU usage.

# Multiprocessor Systems

- A multiprocessor system consists of several processors that share a common physical memory.
- Multiprocessor system provides higher computing power and speed.
- Here, all processors operate under single operating system. Multiplicity of the processors and how they do act together are transparent to the others.
- Following are some advantages of this type of system.
  - Enhanced performance
  - Increased the system's throughput
  - System can divide task into many subtasks which can be executed in parallel in different processors. Thereby speeding up the execution of single tasks.



## contd..

- Also called as Parallel Systems
- Due to their low message delay and high data transfer rate, they are also called tightly coupled systems
- *Tightly coupled system* – processors share memory and the internal clock; communication usually takes place through the shared memory.
- Advantages of parallel system:
  - Increased *throughput*
  - Economical
  - Increased reliability

# Distributed Systems

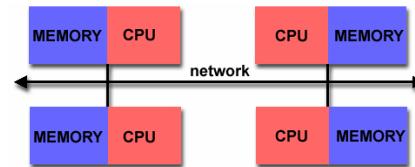
- The motivation behind developing distributed operating systems is the availability of powerful and inexpensive microprocessors and advances in communication technology.
- Distributed systems comprises of many computers that are inter connected by communication networks. The main benefit of distributed systems is its low price/performance ratio.
- Following are some advantages of this type of system.
  - As there are multiple systems involved, user at one site can utilize the resources of systems at other sites for resource-intensive tasks.
  - Fast processing
  - Less load on the Host Machine
- The two types of Distributed Operating Systems are:
  - Client /Server Systems
  - Peer to Peer Systems

# contd..

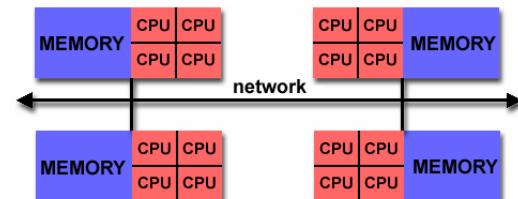
- Distribute the computation among several physical processors.
- Due to low data transfer rate and high message delay, these systems are also called loosely coupled systems
- *Loosely coupled system* – each processor has its own local memory; processors communicate with one another through various communications lines, such as high-speed buses or network communication.
- Advantages of distributed systems.
  - Resources Sharing
  - Computation speed up – load balancing

## Distributed Systems

- Scalability
- Reliability
- Fail-Safe
- Communications



- OS has to cater for resource sharing.
- Hybrid Systems
- Also called as Multi-Computing systems



contd..

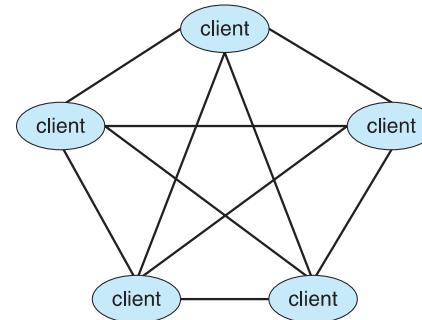
## **Client-Server Systems**

- **Centralized systems** today act as **server systems** to satisfy requests generated by **client systems**.
- Server Systems can be broadly categorized as **compute servers** and **file servers**.
- **Compute-server systems** provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client (Ex: Remote Procedure Calls)
- **File-server systems** provide a file-system interface where clients can create, update, read, and delete files.

contd..

## Peer-to-Peer Systems

- Systems with no central management, self organizing
- Peers in P2P are all equal and distributed
- Resource sharing
- Preferred when there are large number of peers in a network
- Heterogenous



# Clustered Systems

- Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work.
- Clustered systems differ from parallel systems, however, in that they are composed of two or more individual systems coupled together.
- The definition of the term clustered is **not concrete**; the general accepted definition is that clustered computers share storage and are closely linked via LAN networking.
- Clustering is usually performed to provide **high availability**.

# Real-time Systems

- It is defined as an operating system known to give maximum time for each of the critical operations that it performs, like OS calls and interrupt handling.
- The Real-Time Operating system which guarantees the maximum time for critical operations and complete them on time are referred to as **Hard Real-Time Operating Systems**.
- While the real-time operating systems that can only guarantee a maximum of the time, i.e. the critical task will get priority over other tasks, but no assurance of completing it in a defined time. These systems are referred to as **Soft Real-Time Operating Systems**.

# Embedded Systems

- ❑ One of the most important and widely used categories of operating systems
- ❑ Hardware and software designed to perform a dedicated function
- ❑ Tightly coupled to their environment
- ❑ Issues:
  - Limited memory
  - Slow processors
  - Small display screens.
- ❑ Often, embedded systems are part of a larger system or product,
  - E.G. antilock braking system in a car.

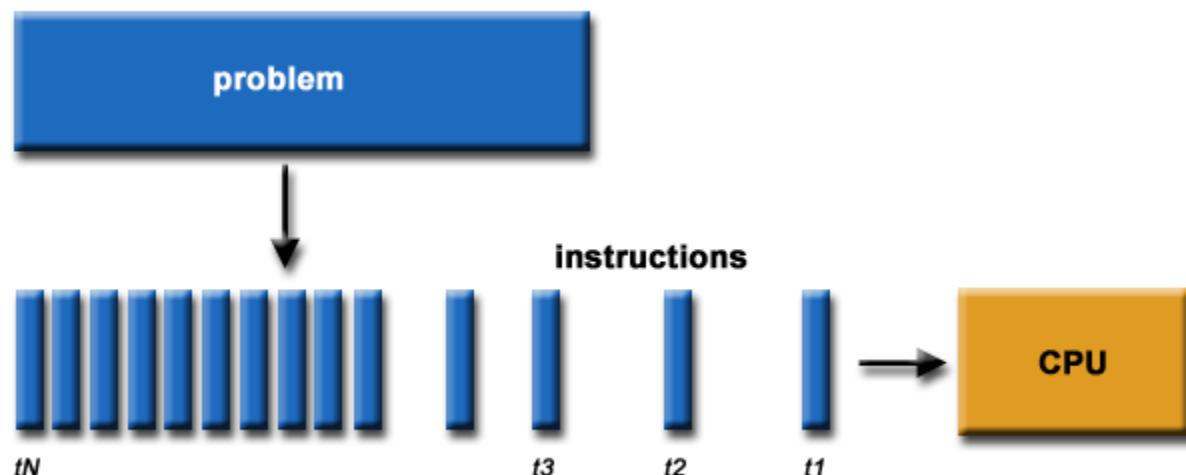
# Handheld Systems

- Handheld systems include **Personal Digital Assistants(PDAs)**, such as Palmtops, Cellphones with connectivity to a network such as the Internet. They are usually of limited size due to which most handheld devices have a small amount of memory, include slow processors, and feature small display screens.
- Many handheld devices have between **512 KB** and **8 MB** of memory. As a result, the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager once the memory is no longer being used.
- Processors for most handheld devices often run at a fraction of the speed of a processor in a PC. Faster processors require **more power**. To include a faster processor in a handheld device would require a **larger battery** that would have to be replaced more frequently.
- The last issue confronting program designers for handheld devices is the small display screens typically available.
- Some handheld devices may use wireless technology such as **BlueTooth**, allowing remote access to e-mail and web browsing. **Cellular telephones** with connectivity to the Internet fall into this category.

# Principles of Parallel/Distributed Computing

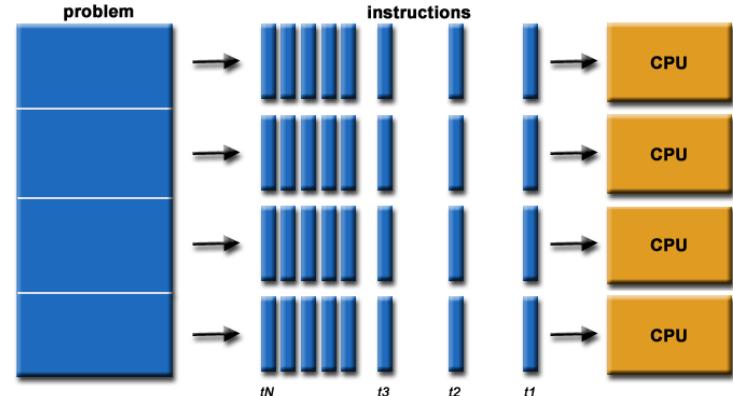
# Serial Computation

- ❑ Traditionally, software has been written for ***serial*** computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.



# Parallel Computing

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
  - Instructions from each part execute simultaneously on different CPUs



# Resource and Problem

- The compute resources can include:
  - A single computer with multiple processors;
  - An arbitrary number of computers connected by a network;
  - A combination of both.
- The computational problem usually demonstrates characteristics such as the ability to be:
  - Broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Solved in less time with multiple compute resources than with a single compute resource.

# Grand Challenge Problems

- Traditionally, parallel computing has been considered to be "the high end of computing" and has been motivated by numerical simulations of complex systems and "Grand Challenge Problems" such as:
  - weather and climate
  - chemical and nuclear reactions
  - biological, human genome
  - geological, seismic activity
  - mechanical devices - from prosthetics to spacecraft
  - electronic circuits
  - manufacturing processes

# Applications

- Today, commercial applications are providing an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. Example applications include:
  - parallel databases, data mining
  - oil exploration
  - web search engines, web based business services
  - computer-aided diagnosis in medicine
  - management of national and multi-national corporations
  - advanced graphics and virtual reality, particularly in the entertainment industry
  - networked video and multi-media technologies
  - collaborative work environments
- Ultimately, parallel computing is an attempt to maximize the infinite but seemingly scarce commodity called **time**.

# Why use parallel computing?

- The primary reasons for using parallel computing:
  - Save time - wall clock time
  - Solve larger problems
  - Provide concurrency (do multiple things at the same time)
- Other reasons might include:
  - Taking advantage of non-local resources - using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
  - Cost savings - using multiple "cheap" computing resources instead of paying for time on a supercomputer.
  - Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

# Why use parallel computing?

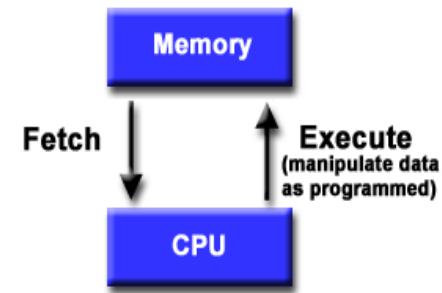
- Limits to serial computing - both physical and practical reasons pose significant constraints to simply building ever faster serial computers:
  - Transmission speeds - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
  - Limits to miniaturization - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
  - Economic limitations - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.
- The future: during the past 10 years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) suggest that **parallelism is the future of computing**.

# Concept and Terminology

- ❑ Von Newmann Architecture
- ❑ Flynn's Classical Taxonomy

# Von Neumann Architecture

- ❑ For over 40 years, virtually all computers have followed a common machine model known as the von Neumann computer. Named after the Hungarian mathematician John von Neumann.
- ❑ A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.
- ❑ Basic design:
  - Memory is used to store both program and data instructions
  - Program instructions are coded data which tell the computer to do something
  - Data is simply information to be used by the program
  - A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then **sequentially** performs them.

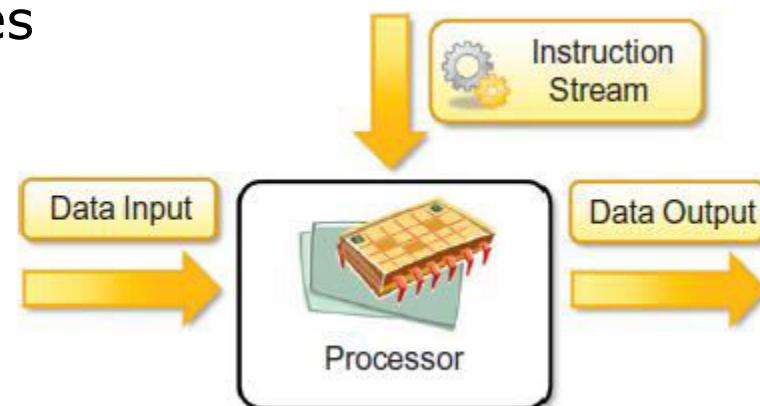
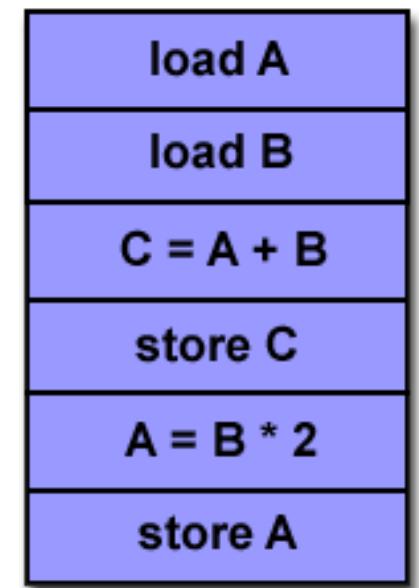


# Flynn's Classical Taxonomy

- There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of **Instruction** and **Data**. Each of these dimensions can have only one of two possible states: **Single** or **Multiple**.
- There are 4 possible classifications according to Flynn.
  - **Single Instruction, Single Data (SISD)**
  - **Single Instruction, Multiple Data (SIMD)**
  - **Multiple Instruction, Single Data (MISD)**
  - **Multiple Instruction, Multiple Data (MIMD)**

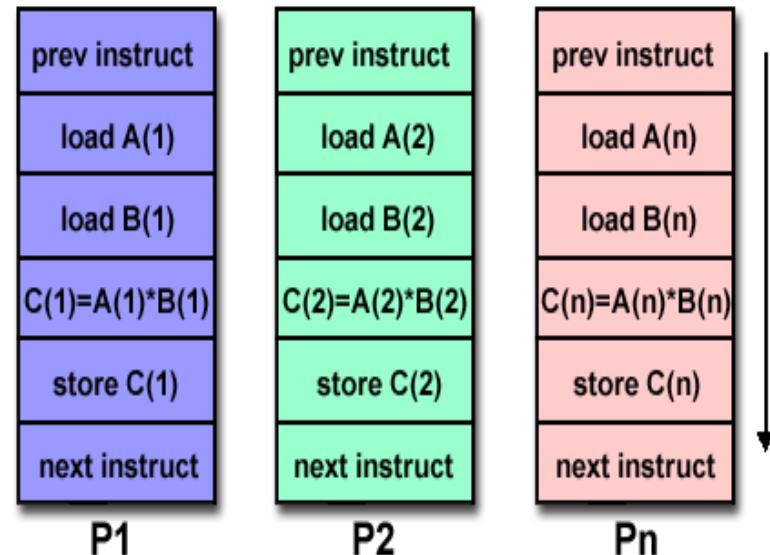
# Single Instruction Single Data

- ❑ A serial (non-parallel) computer
- ❑ Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- ❑ Single data: only one data stream is being used as input during any one clock cycle
- ❑ Deterministic execution
- ❑ This is the oldest and until recently, the most prevalent form of computer
- ❑ Examples: most PCs, single CPU workstations and mainframes

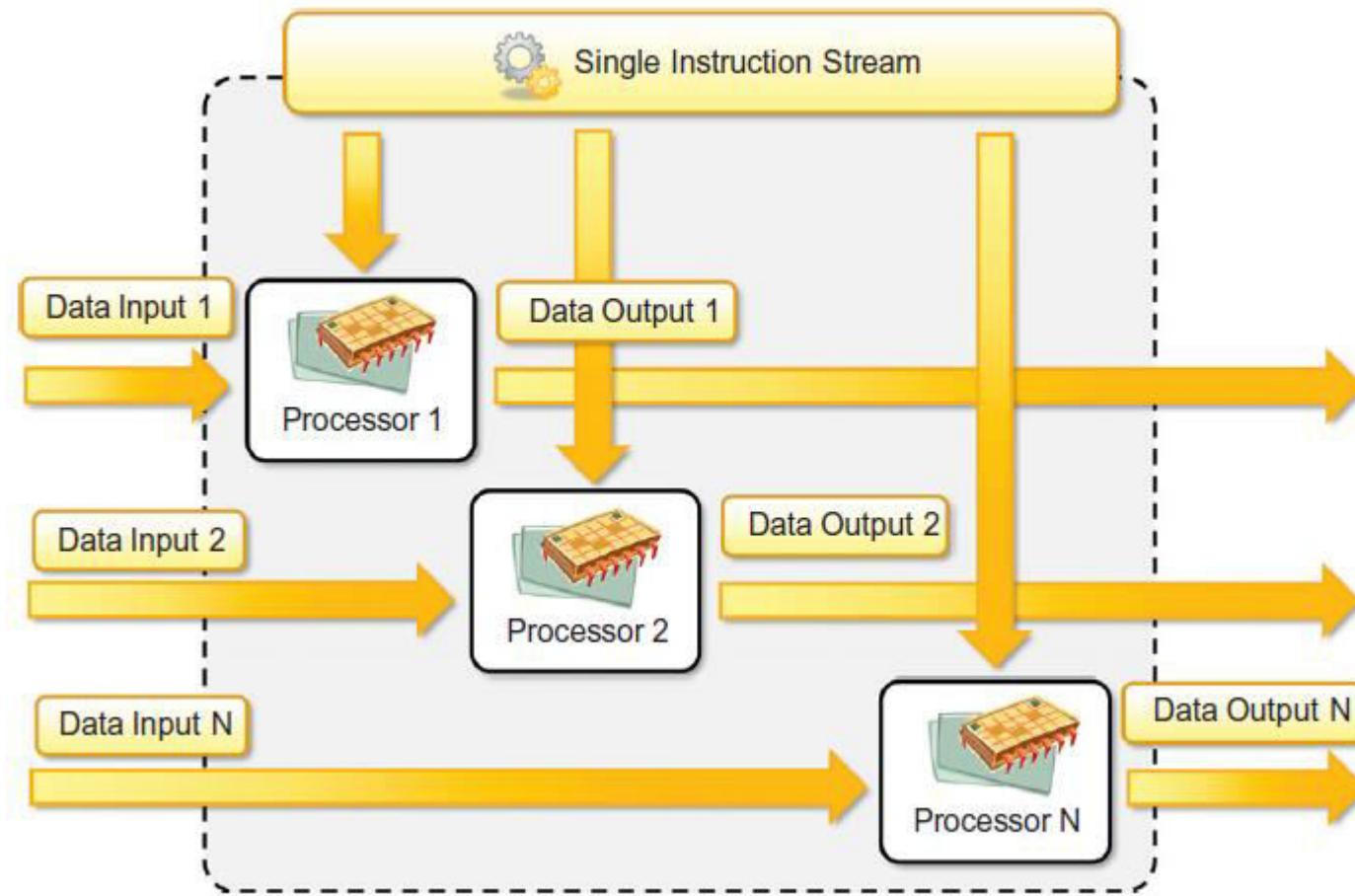


# Single Instruction Multiple Data

- ❑ A type of parallel computer
- ❑ Single instruction: All processing units execute the same instruction at any given clock cycle
- ❑ Multiple data: Each processing unit can operate on a different data element
- ❑ This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- ❑ Best suited for specialized problems characterized by a high degree of regularity, such as image processing.
- ❑ Synchronous (lockstep) and deterministic execution
- ❑ Two varieties: Processor Arrays and Vector Pipelines
- ❑ Examples:
  - Processor Arrays: Connection Machine CM-2, Maspar MP-1, MP-2
  - Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820

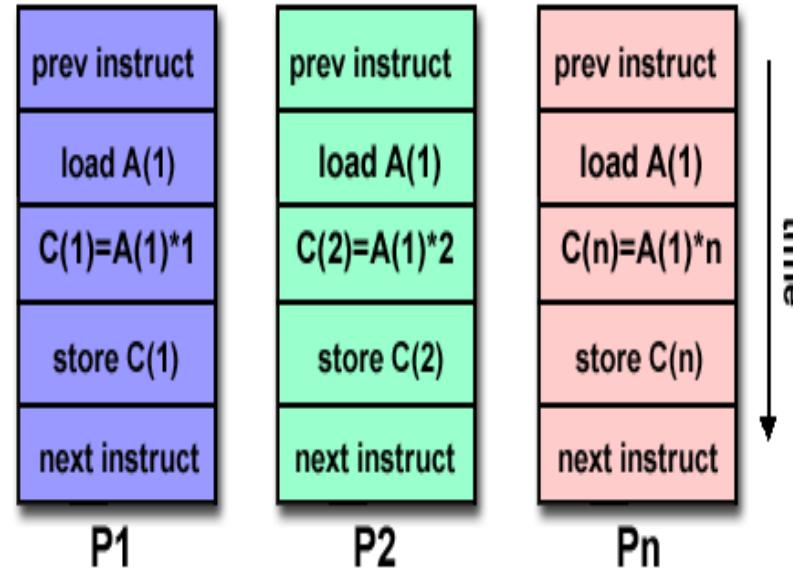


# SIMD (contd..)

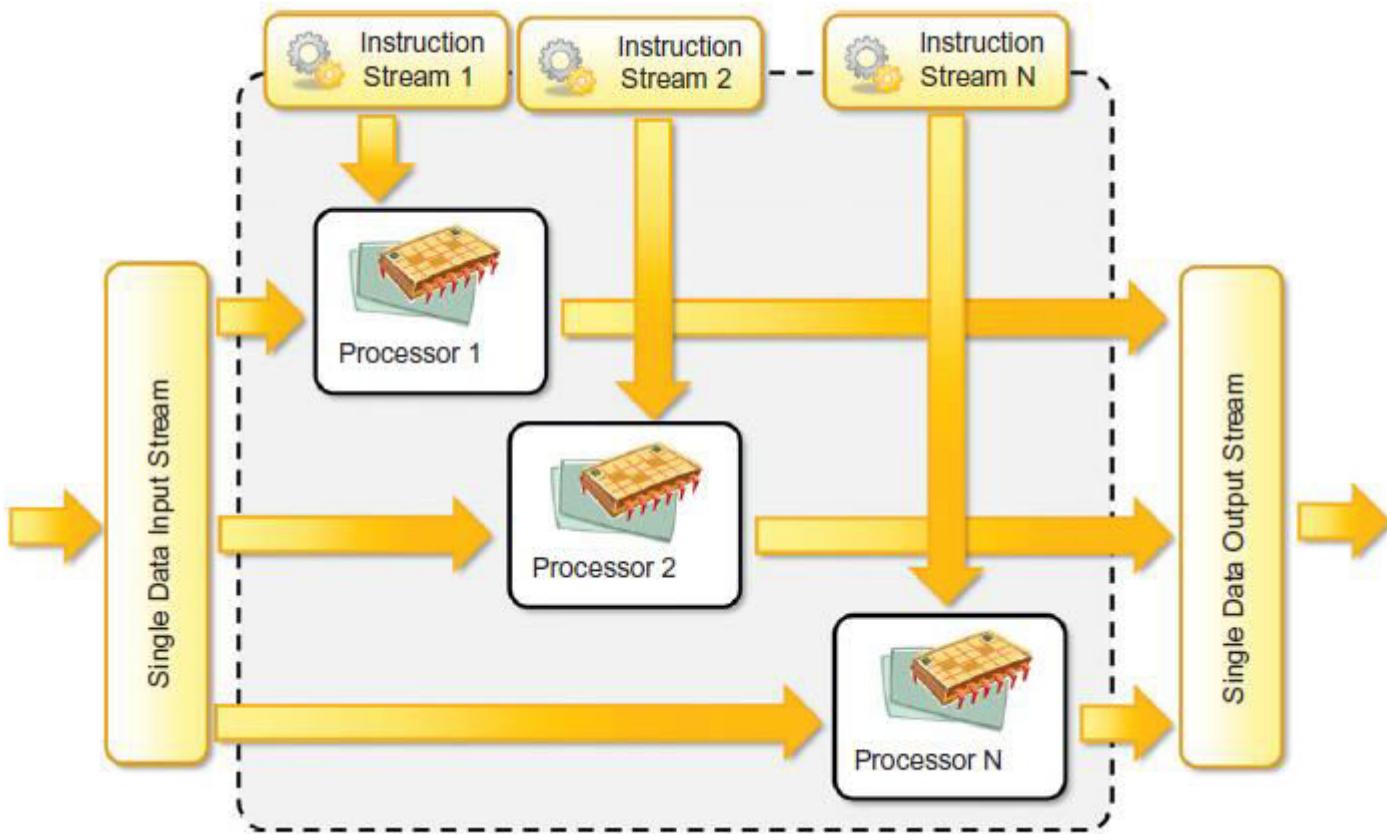


# Multiple Instruction Single Data

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.

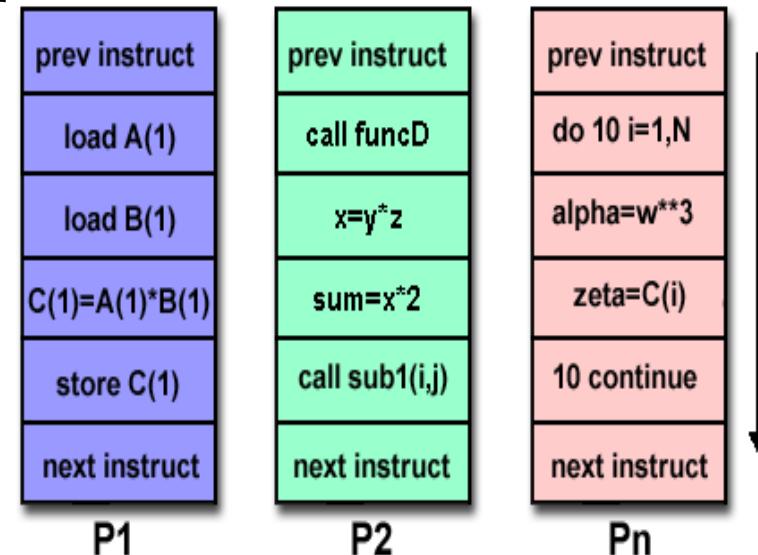


# MISD (contd..)

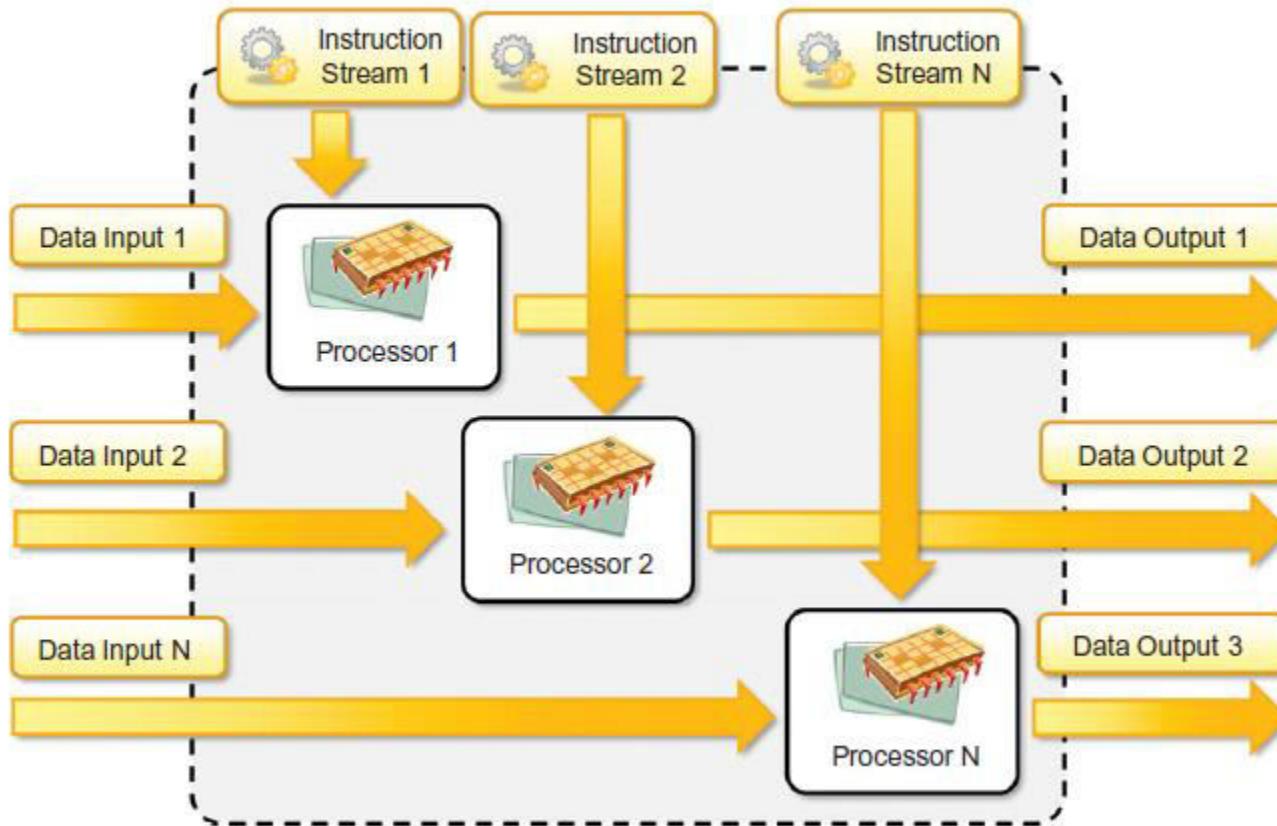


# Multiple Instruction Multiple Data

- Currently, the most common type of parallel computer. Most modern computers fall into this category.
- Multiple Instruction: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.



# MIMD (contd..)

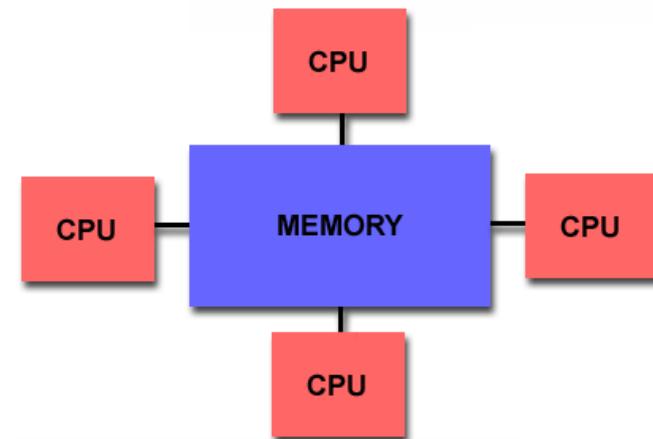
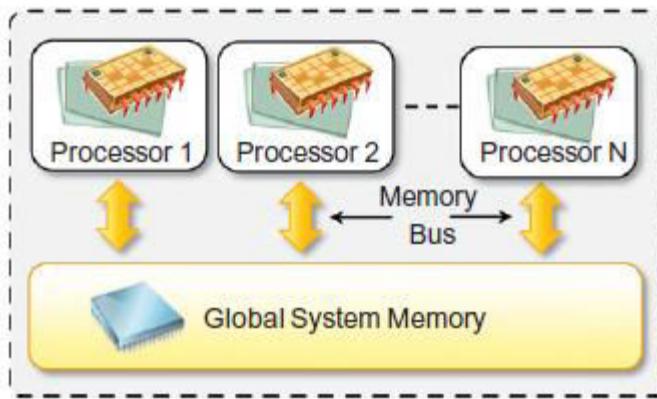


# Parallel Computer Memory Architectures

- ❑ Shared Memory
- ❑ Distributed Memory
- ❑ Hybrid Distributed Shared Memory

# Shared Memory

- ❑ Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- ❑ Multiple processors can operate independently but share the same memory resources.
- ❑ Changes in a memory location effected by one processor are visible to all other processors.
- ❑ Shared memory machines can be divided into two main classes based upon memory access times: **UMA** and **NUMA**.

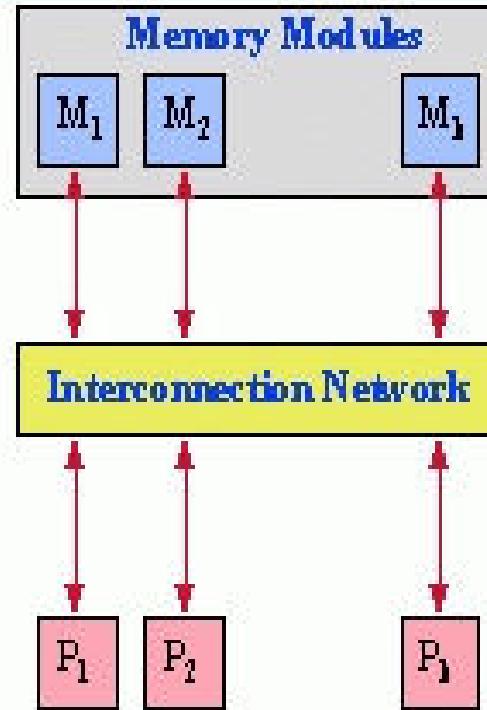


# Shared Memory

- Uniform Memory Access (UMA):
  - Most commonly represented today by Symmetric Multiprocessor (SMP) machines
  - Identical processors
  - Equal access and access times to memory
  - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.
- Non-Uniform Memory Access (NUMA):
  - Often made by physically linking two or more SMPs
  - One SMP can directly access memory of another SMP
  - Not all processors have equal access time to all memories
  - Memory access across link is slower
  - If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

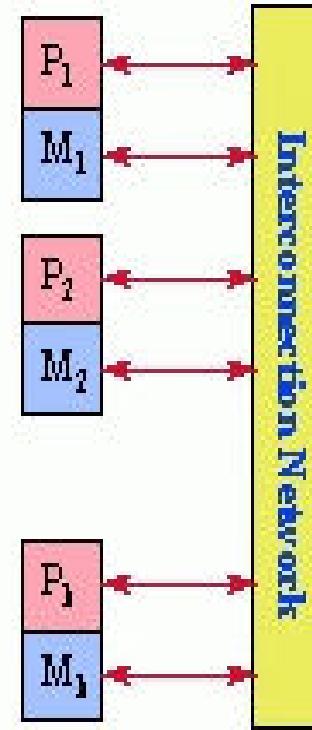
# UMA and NUMA

## UMA vs. NUMA



Ram Miller

Since 1997



IS

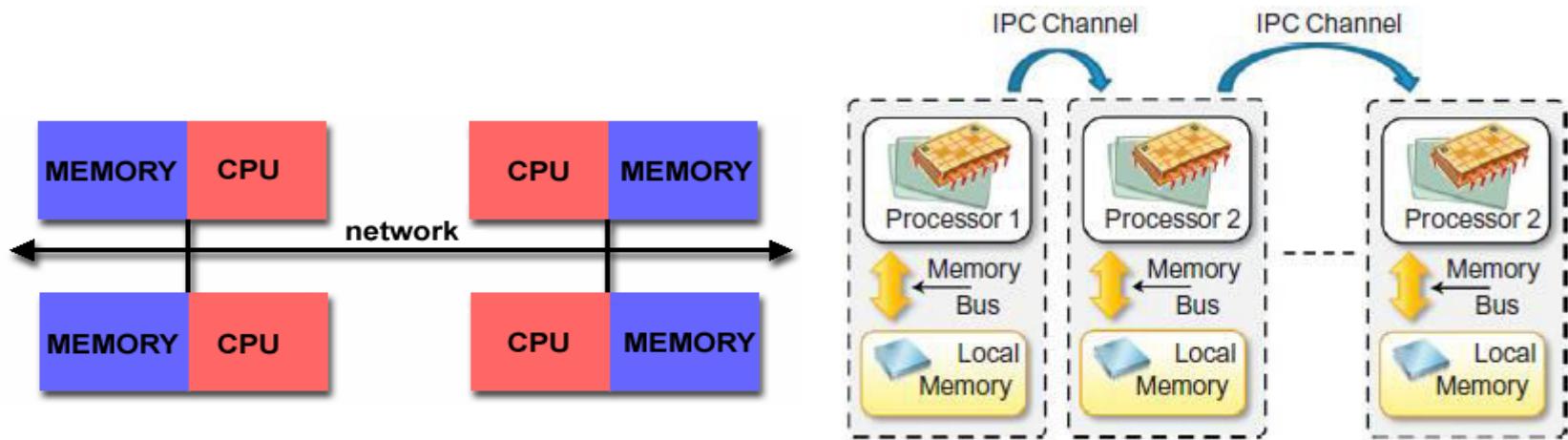
# Shared Memory

- Advantages:
  - Global address space provides a user-friendly programming perspective to memory
  - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs
- Disadvantages:
  - Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
  - Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
  - Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

# Distributed Memory

- ▣ Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.
- ▣ Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- ▣ Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- ▣ When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- ▣ The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

contd..

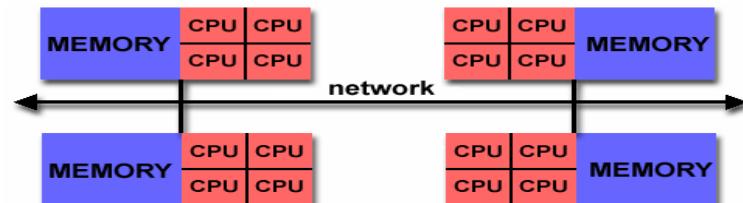


# Distributed Memory

- Advantages:
  - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- Disadvantages:
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Non-uniform memory access (NUMA) times

# Distributed Shared Memory

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.



# Approaches of Parallel/Distributed Programming

- ❑ A sequential program is one that runs on a single processor and has a single line of control.
- ❑ To make many processors collectively work on a single program, the program must be divided into smaller independent chunks so that each processor can work on separate chunks of the problem.
- ❑ The program decomposed in this way is a parallel program.

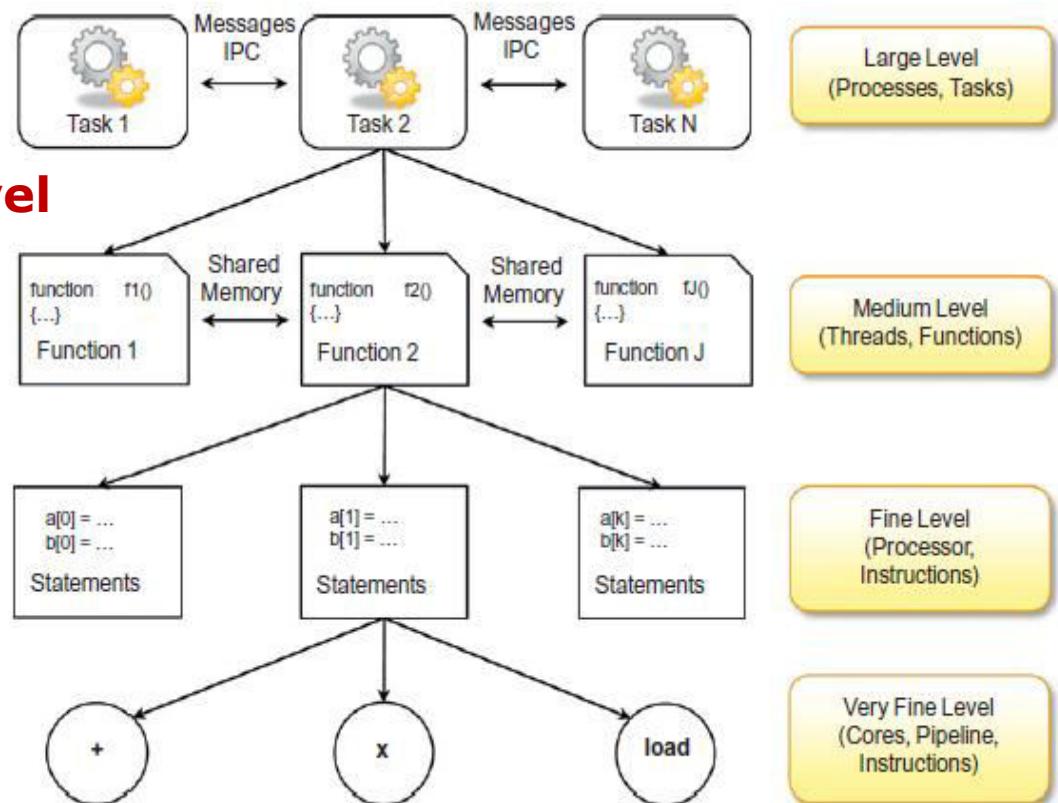
## contd..

- A wide variety of parallel programming approaches are available.
- **Data parallelism** - the divide-and-conquer technique is used to split data into multiple sets, and each data set is processed on different PEs using the same instruction. This approach is highly suitable to processing on machines based on the SIMD model.
- **Process parallelism** - a given operation has multiple (but distinct) activities that can be processed on multiple processors.
- **Farmer-and-worker model** - a job distribution approach is used: one processor is configured as master and all other remaining PEs are designated as slaves; the master assigns jobs to slave PEs and, on completion, they inform the master, which in turn collects results.

# Levels of Parallelism or Granularity

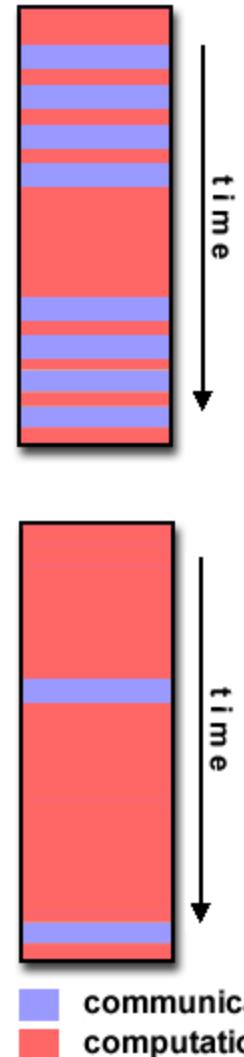
- Levels of parallelism are decided based on the lumps of code (grain size) that can be a potential candidate for parallelism.

- **Large or task level**
- **Medium or control level**
- **Fine or data level**
- **Very fine level**



# Granularity contd..

- In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
- Fine-grain Parallelism:
  - Relatively small amounts of computational work are done between communication events
  - Low computation to communication ratio
  - Facilitates load balancing
  - Implies high communication overhead and less opportunity for performance enhancement
  - If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.



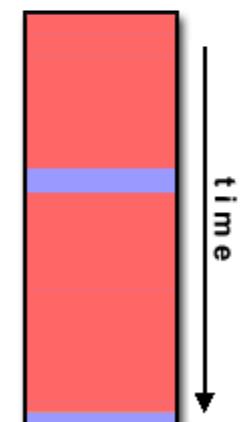
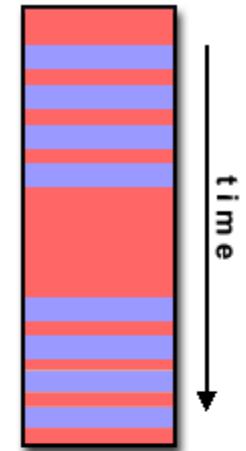
# Granularity contd..

## □ Coarse-grain Parallelism:

- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently

## □ Which is Best?

- The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance.



Legend:  
■ communication  
■ computation

# Speedup Factor

- ❑ How much faster the multiprocessor solves the problem?
- ❑ We defined the speedup factor  $S(p)$  which is a measure of relative performance

$$S(p) = \frac{\text{Execution time using single processor system}}{\text{Execution time using a multiprocessor with } p \text{ processors}} = \frac{t_s}{t_p}$$

- ❑ Maximum speedup (linear speedup)

$$S(p) \leq \frac{t_s}{t_s/p} = p$$

- ❑ Superlinear speedup  $S(p) > p$

# Efficiency

- If we want to know how long processors are being used on the computation. The efficiency  $E$  is defined as

$$E = \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor} \times \text{number of processors}}$$

$$= \frac{t_s}{t_p \times p}$$

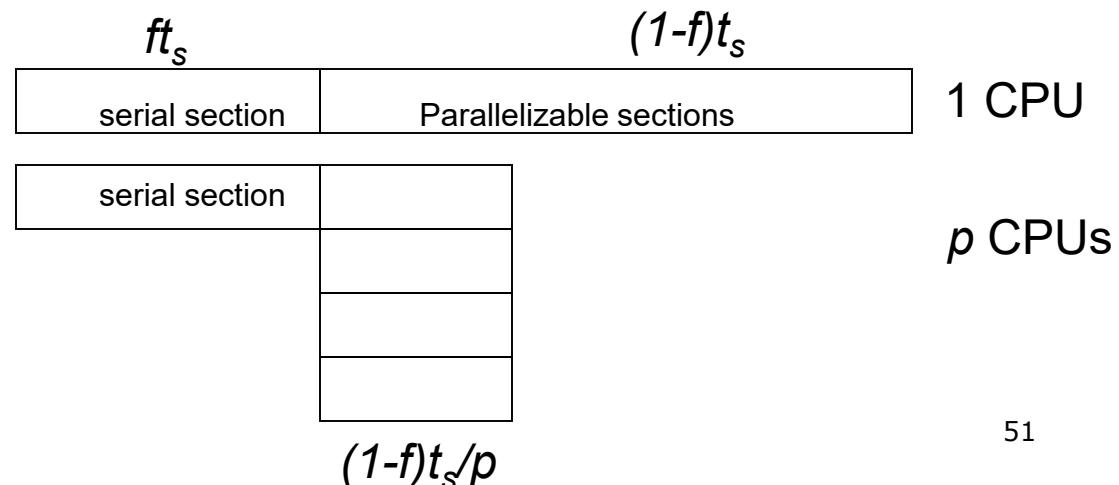
$$E = \frac{S(p)}{p} \times 100\%$$

while  $E$  is given as a percentage. If  $E$  is 50%, the processors are being used half the time on the actual computation, on average. If efficiency is 100% then the speedup is  $p$ .

# Overheads

- Several factors will appear as overhead in the parallel computation
  - Periods when not all the processors can be performing useful work
  - Extra computations in the parallel version
  - Communication time between processors
- Assume the fraction of the computation that cannot be divided into concurrent tasks is  $f$ .
- The time used to perform computation with  $p$  processors is

$$t_p = ft_s + (1-f)t_s / p$$

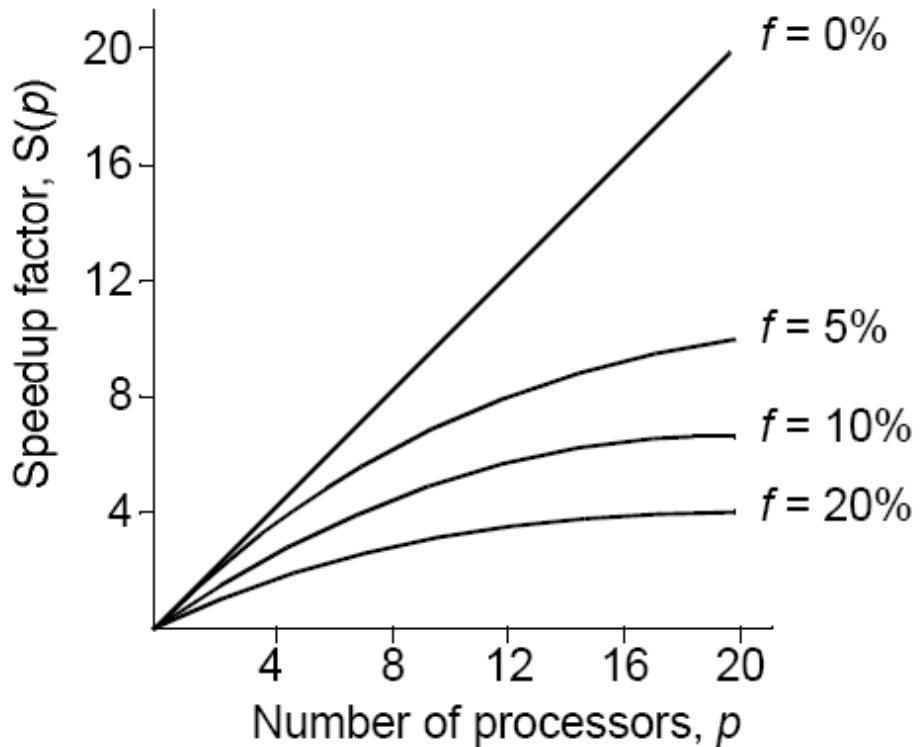


# Amdahl's Law

- The speedup factor is given as

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s / p} = \frac{p}{1+(p-1)f}$$

$$S(p) = \frac{1}{f} \quad p \rightarrow \infty$$



# Complexity

- ❑ In general, parallel applications are much more complex than corresponding serial applications, perhaps an order of magnitude. Not only do you have multiple instruction streams executing at the same time, but you also have data flowing between them.
- ❑ The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle:
  - Design
  - Coding
  - Debugging
  - Tuning
  - Maintenance
- ❑ Adhering to "good" software development practices is essential when working with parallel applications - especially if somebody besides you will have to work with the software.

# Parallel Terminology

## □ Task

- A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

## □ Parallel Task

- A task that can be executed by multiple processors safely (yields correct results)

## □ Serial Execution

- Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.

## □ Parallel Execution

- Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.

# Parallel Terminology

## □ Shared Memory

- From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

## □ Distributed Memory

- In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

## □ Communications

- Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

# Parallel Terminology

## □ Synchronization

- The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point. Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

## □ Granularity

- In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
  - **Coarse:** relatively large amounts of computational work are done between communication events
  - **Fine:** relatively small amounts of computational work are done between communication events

## □ Observed Speedup

- Observed speedup of a code which has been parallelized, defined as: wall-clock time of serial execution / wall-clock time of parallel execution
- One of the simplest and most widely used indicators for a parallel program's performance.

# Parallel Terminology

## □ Parallel Overhead

- The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:
  - Task start-up time
  - Synchronizations
  - Data communications
  - Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
  - Task termination time

## □ Massively Parallel

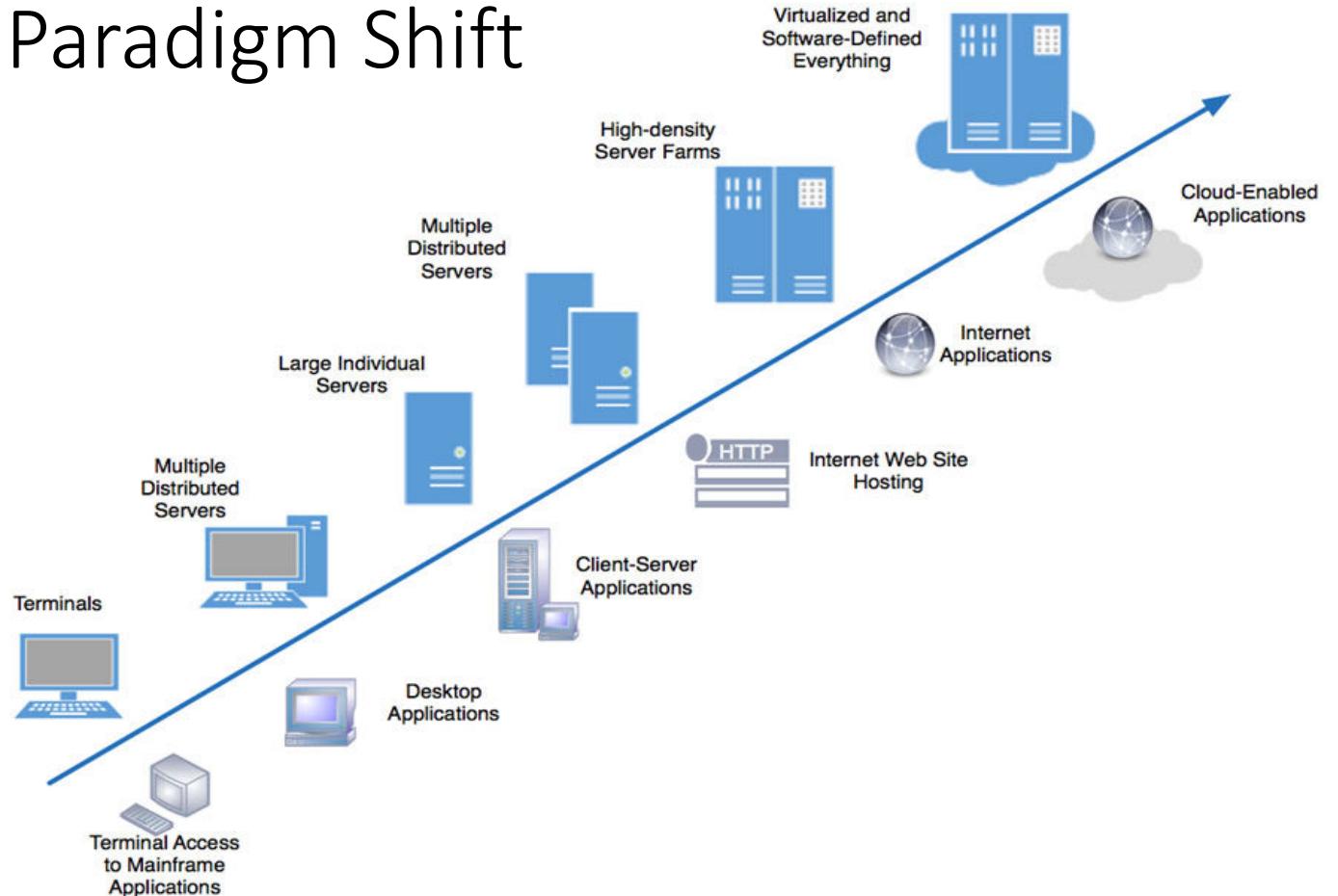
- Refers to the hardware that comprises a given parallel system - having many processors.

## □ Scalability

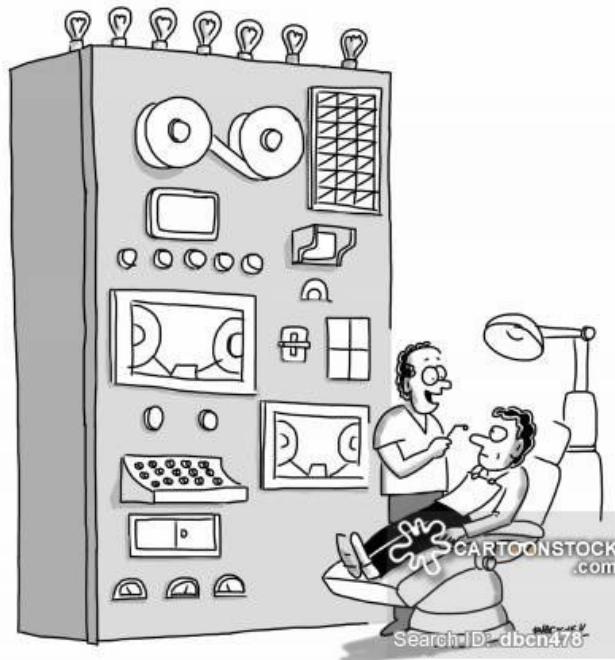
- Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:
  - Hardware - particularly memory-cpu bandwidths and network communications
  - Application algorithm
  - Parallel overhead related
  - Characteristics of your specific application and coding

# Distributed Computing Models

# Computing Paradigm Shift



# Mainframe Computing



"YES, I BELIEVE I WAS ONE OF THE FIRST DENTISTS TO USE COMPUTERS!"

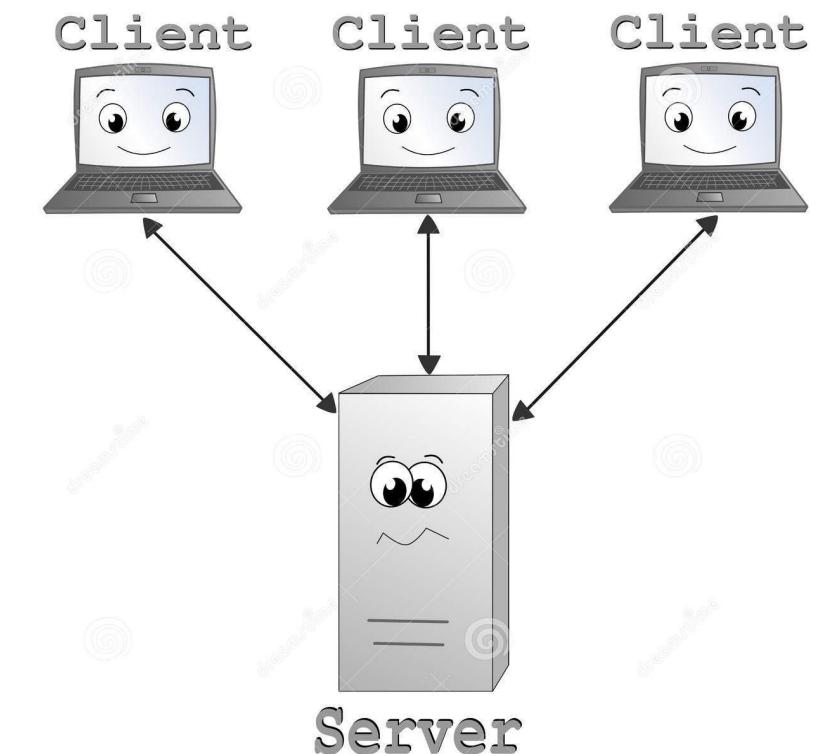
- Jobs
- Batches
- Processing
- To carry out some mundane and routine jobs such as payroll, accounts, inventory thus sparing employees from tedious jobs.
- It was available in one location, and anyone who needs it must go to computer center for availing it.

# Personal Computing



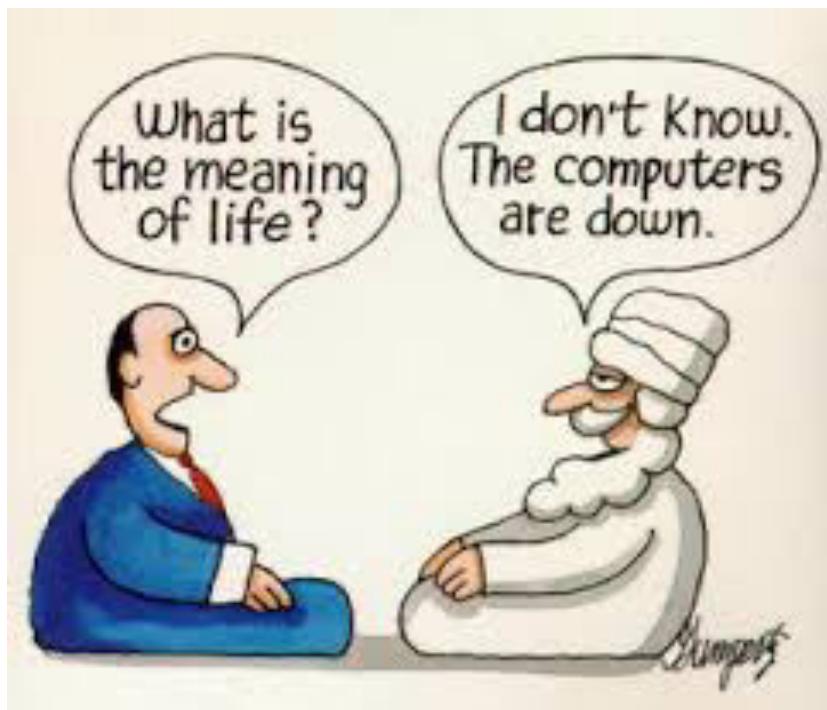
- Desktop computing - personal computer small enough to fit conveniently in an individual workspace.
- Providing computers to each employee on their desktop or workspace.
- Decentralized computing.
- Less expensive, easy to upgrade and less accessories needed.
- Information sharing with other users is a tedious process.

# Network Computing



- Networked computers - Local Area network (LAN) achieved this.
- In the networked computing model- a relatively powerful computer- **server** is loaded with all software needed
- Each user is provided with a connected- **terminal** to access and work

# Internet Computing



- Network computing such as LAN connected users within an office or institutions.
- Internet computing - connect organizations located in **different geographical locations**.

# Utility Computing

- Conventional Internet hosting services have the capability to quickly arrange for the rental of individual servers, for example to provision a bank of web servers to accommodate a sudden surge in traffic to a web site.
- “Utility computing” usually envisions some form of virtualization so that the amount of storage or computing power available is considerably larger than that of a single time-sharing computer. Multiple servers are used on the “back end” to make this possible.
- These might be a dedicated computer cluster specifically built for the purpose of being rented out, or even an under-utilized supercomputer.
- The technique of running a single calculation on multiple computers is known as distributed computing.

# Cluster Computing

- A computer cluster is a group of linked computers, working together closely thus in many respects forming a **single computer**.
- The components of a cluster are connected to each other through fast local area networks.
- Clusters are mainly used for load balancing and providing high availability.
- Requirements for such a computing increasing fast.
  - More data to process.
  - More compute intensive algorithms available.

# Cluster Computing

## Benefits

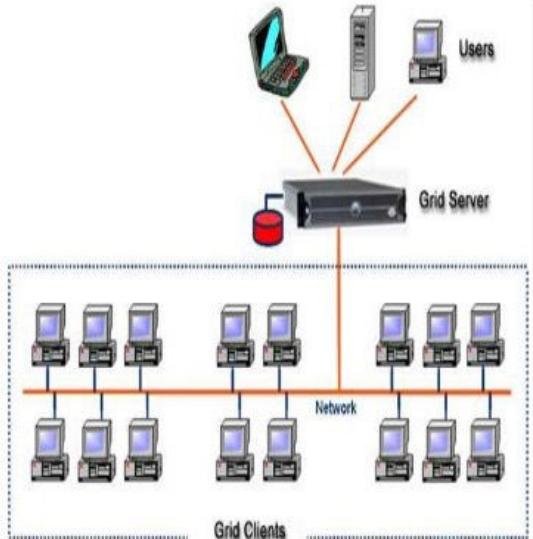
- High Availability.
- Reducing cost.
- Manageability.

## Drawbacks

- Problem in Finding Fault.
- The machines in a cluster are dedicated to work as a single unit.
- The computers in the cluster are normally contained in a single location.

# Grid Computing

## How Grid computing works ?



In general, a grid computing system requires:

- At least one computer, usually a server, which handles all the administrative duties for the System
- A network of computers running special grid computing network software.
- A collection of computer software called middleware

- Computing power available within an enterprise is not sufficient to carry out the computing task.
- Data required for the processing is generated at various geographical locations.
- GC requires the use of software that can divide and farm out pieces of a program as one large system image to several thousand computers.

# Grid Computing

## Benefits

- Enables applications to be easily scaled
- Better utilization of underused resources
- Parallelization of processing

## Drawbacks

- Proprietary approach should be eliminated
- There is a single point of failure if one unit on the grid degrades
- No pay as you go

# Grid Computing vs Cluster Computing

- Cluster is homogenous.
  - The cluster computers all have the same hardware and OS.
- The computers in the cluster are normally contained in a single location
- Grids are heterogeneous.
  - Run different operating systems and have different hardware.
- Grids are inherently distributed by its nature over a LAN, metropolitan or WAN.

# Cloud Computing

- Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services.
- The datacenter hardware and software is what we will call a Cloud.

# Cloud Computing

## Benefits

- Disaster recovery
- Increased Scalability
- Faster Deployment
- Metered Service
- Highly Automated

## Drawbacks

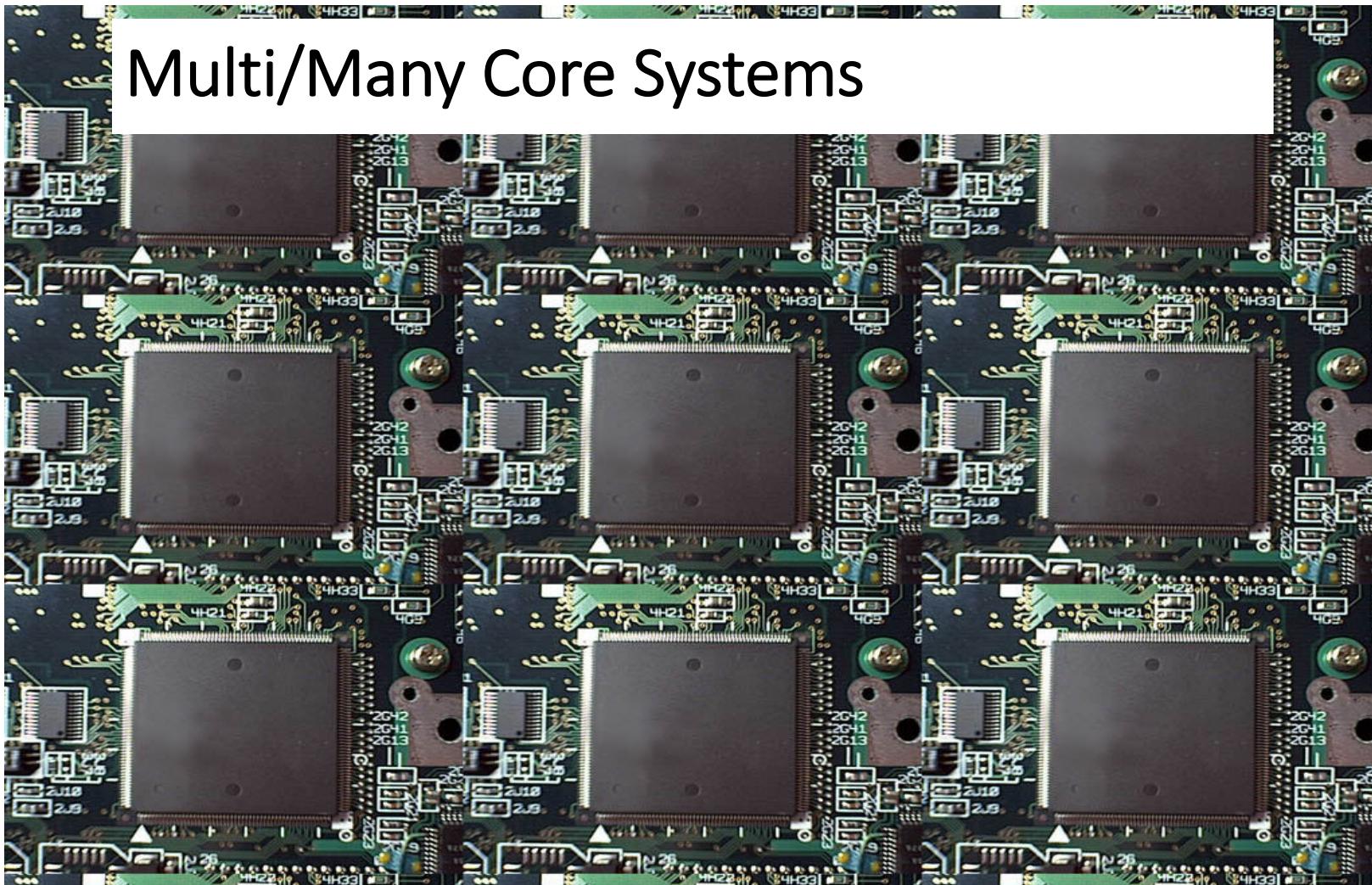
- Constant Internet Connection
- High Speed Internet Required
- Data Stored is not secure



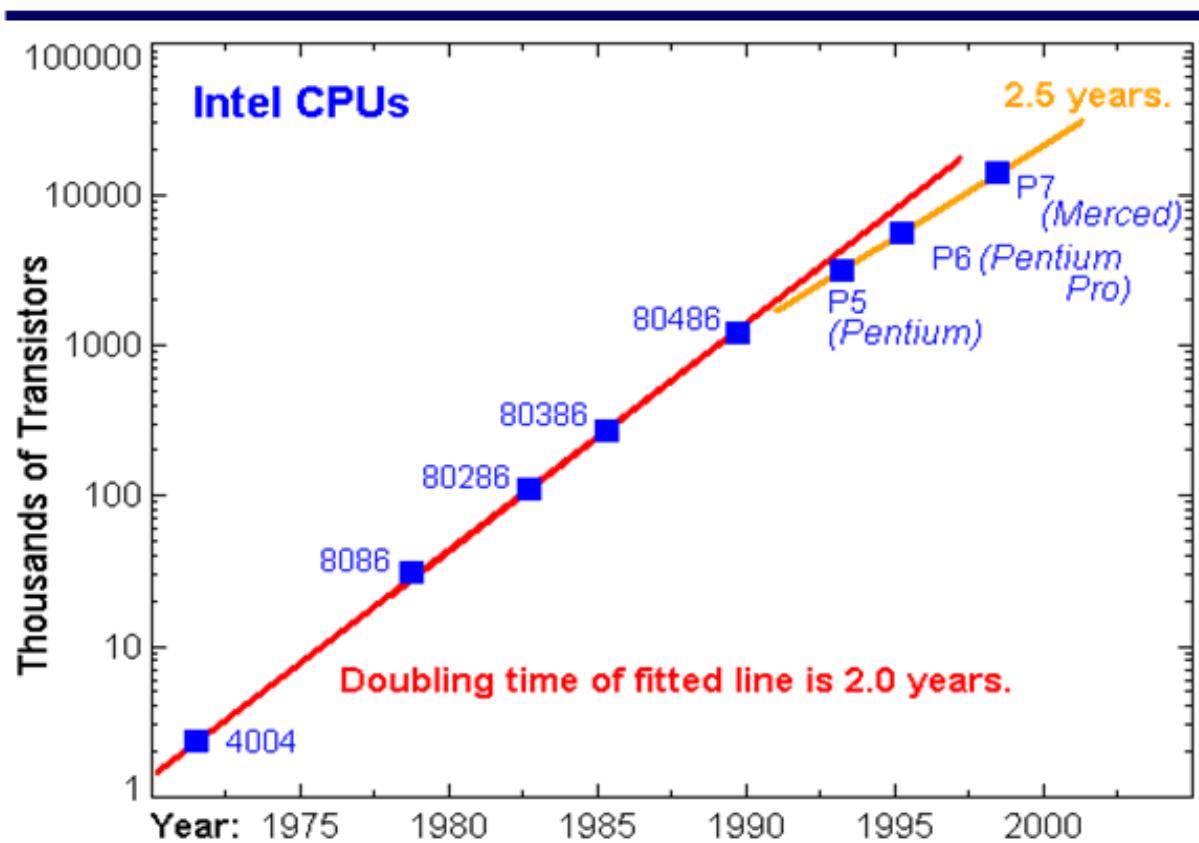
# Grid Vs Cluster Vs Cloud Computing

Properties	Cloud	Cluster	Grid
On-demand self-Service	Yes	No	No
Broad network access	Yes	Yes	Yes
Resource pooling	Yes	Yes	Yes
Rapid elasticity	Yes	No	No
Measured service	Yes	No	Yes

# Multi/Many Core Systems



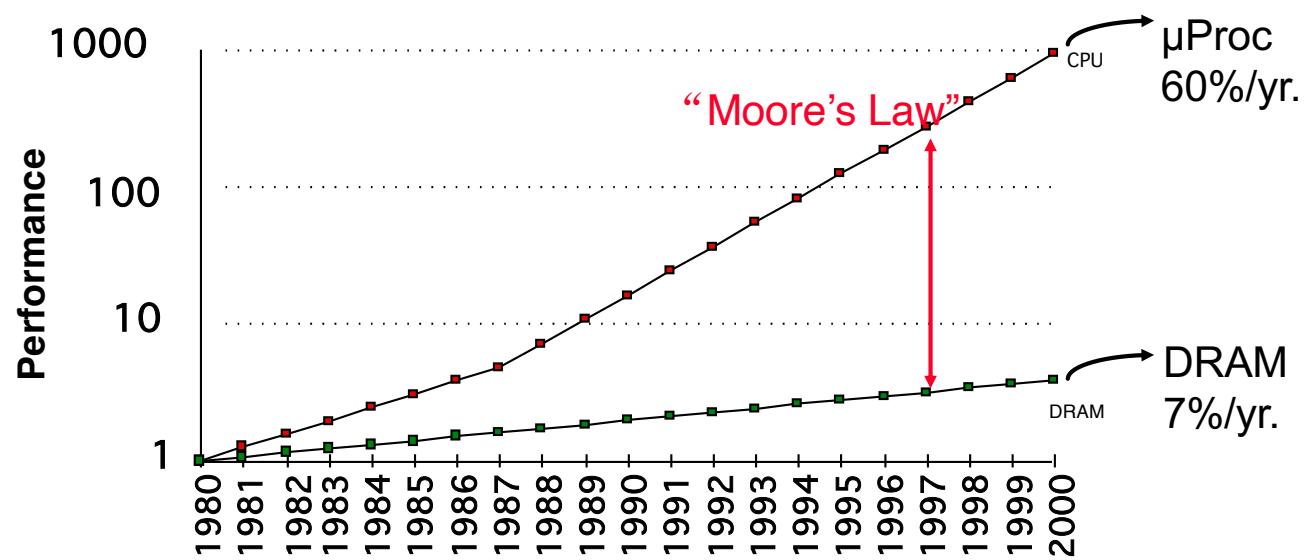
## Moore's Law



# Processor Trends So Far..

- Smarter Brain
  - (e.g. x386 → x486 → Pentium → P2 → P3 → P4)
- Larger Memory
  - Larger caches, DRAM, Disk
- Smaller Head
  - Fewer chips (integrate more things onto a chip)
- More Power Consumption
  - few Watts → 120+ Watts!
- More Complex
  - 1Billion Transistors; design + verification complexity

# Processor-Memory Performance Gap



From D. Patterson, CS252, Spring 1998 ©UCB

## Major Problem Today: End of the Road!

- Can't increase Power Consumption (~100W!)
- Can't increase Design Complexity
- Can't increase Verification Requirements

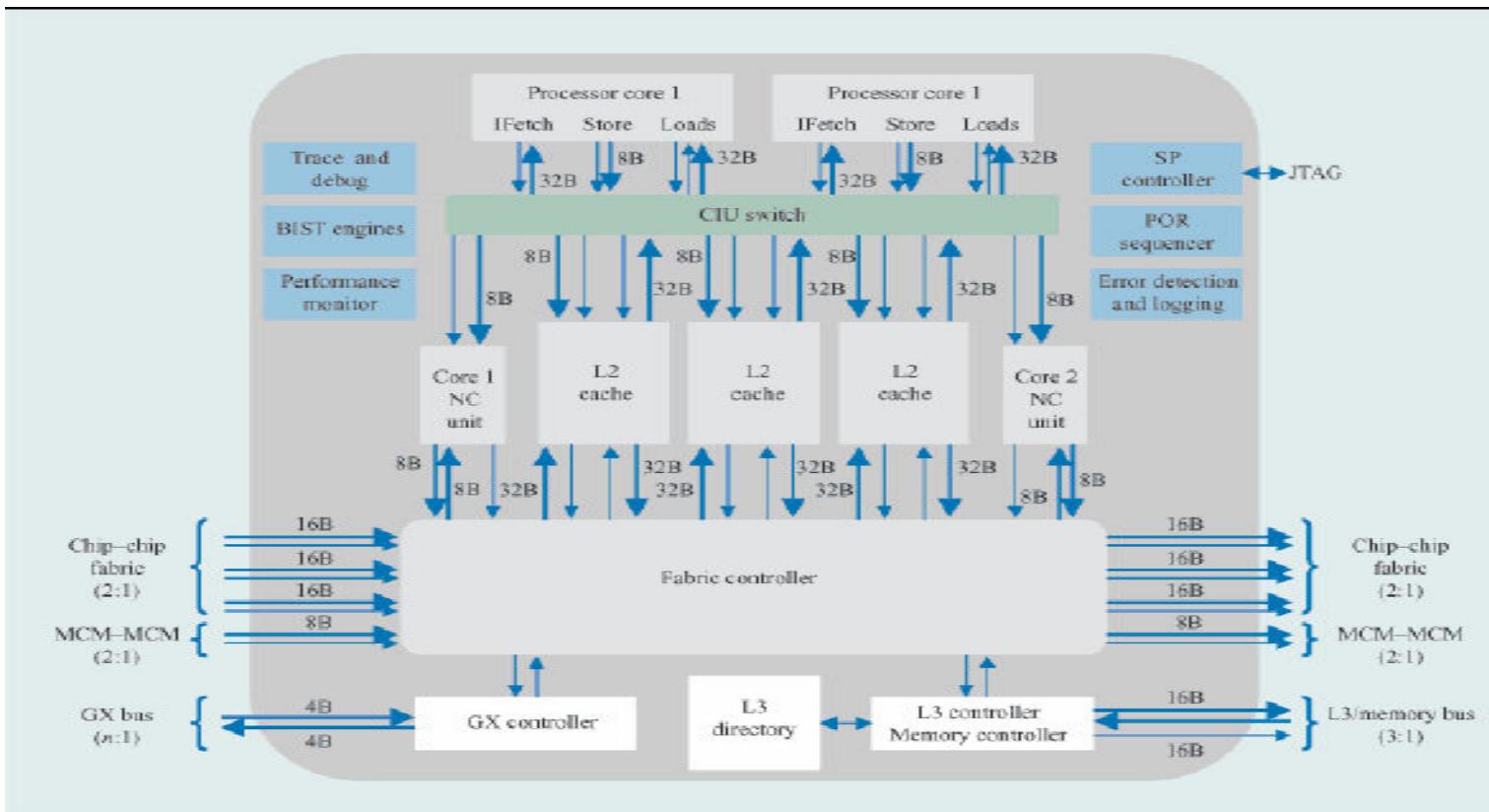
No further improvements to a Processor?!

## Obvious Answer: Use Multiple Brains

- If single brain can't be improved, use multiple brains!
- Put multiple simple CPUs on a single chip

**Multi-Core!**

# The First Multi-Core: IBM Power 4 Processor



**Figure 1**

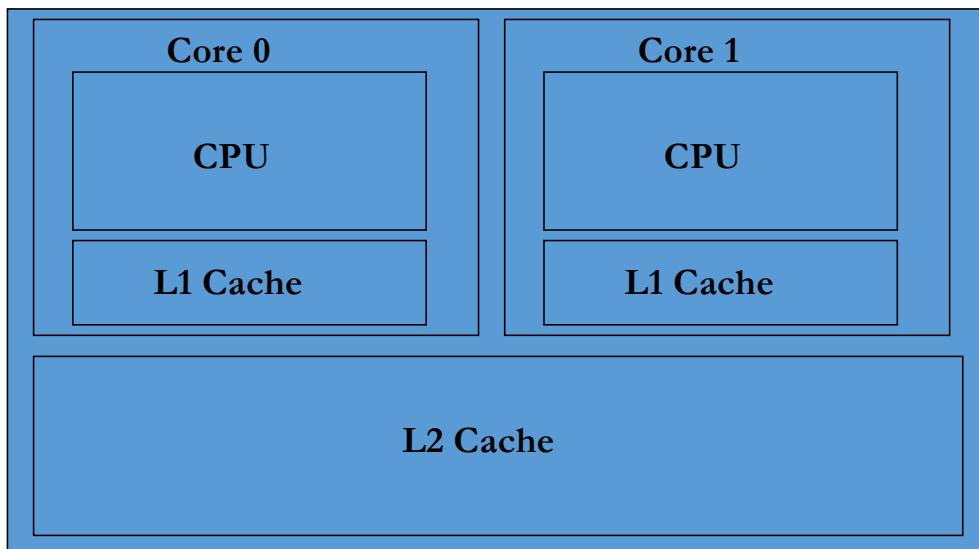
## POWER4 chip logical view.

# Definitions

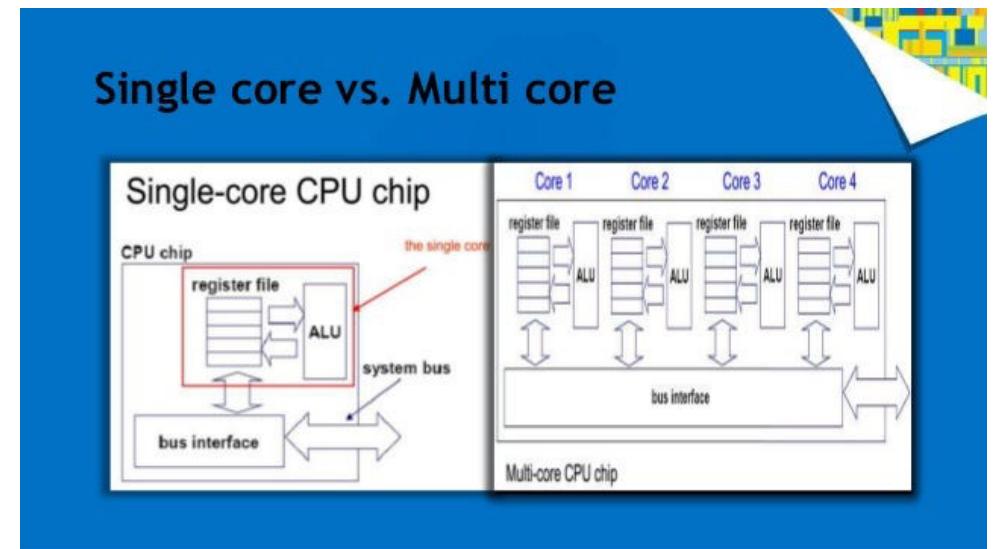
- A **Multi-Core** processor combines two or more independent **cores** into a single package composed of a single integrated circuit (IC), called a die, or more dies packaged together.  
A **dual-core** processor contains two cores, and a **quad-core** processor contains four cores.
- Typically the term **Many-Core** is sometimes used to describe multi-core architectures with an especially high number of cores (tens or hundreds).

# Multi core

- Single Chip
- Multiple distinct processing Engine
- E.g.) Shared-cache Dual Core Architecture



Multicore..



# Multi core

- Cores in a multi-core device may share a single coherent cache at the highest on-device cache level (e.g. L2 for the Intel Core 2) or may have separate caches (e.g. current AMD dual-core processors).
- The processors also share the same interconnect to the rest of the system.
- Each "core" independently implements optimizations such as superscalar execution, pipelining, and multithreading.
- A system with n cores is effective when it is presented with n or more threads concurrently.

# Advantages of Multi-core

- The proximity of multiple CPU cores on the same die allows the cache coherency circuitry to operate at a much higher clock rate than is possible if the signals have to travel off-chip.
- Combining equivalent CPUs on a single die significantly improves the performance of cache snoop (alternative: Bus snooping) operations.
- This means that signals between different CPUs travel shorter distances, and therefore those signals degrade less. These higher quality signals allow more data to be sent in a given time period since individual signals can be shorter and do not need to be repeated as often.

## Advantages of Multi-core(cont..)

- The largest boost in performance will likely be noticed in improved response time while running CPU-intensive processes, like antivirus scans, ripping/ burning media (requiring file conversion), or searching for folders.
- For example, if the automatic virus scan initiates while a movie is being watched, the application running the movie is far less likely to be starved of processor power, as the antivirus program will be assigned to a different processor core than the one running the movie playback.

# Speed up and Amdahl's Law

# How to calculate parallel time ( $tp$ )?

$$t_p = t_{\text{comm}} + t_{\text{comp}}$$

$$t_{\text{comm}} = t_{\text{comm1}} + t_{\text{comm2}} + \dots$$

$$t_{\text{temp}} = t_{\text{temp}_1} + t_{\text{temp}_2} + \dots$$

$$t_{\text{comm}} = t_{\text{stop}} + w t_{\text{data}}$$

$w$  = amount  
of data

$t_p$  = Parallel time       $t_{comp}$  = Computation time  
 $t_{comm}$  = Communication time  
 $t_{start}$  = Start up time  
 $t_{data}$  = transmission time

# Example 1

## Example

Suppose we were to add  $n$  numbers on two computers, where each computer adds  $n/2$  numbers together, and the numbers are initially all held by the first computer. The second computer submits its result to the first computer for adding the two partial sums together. This problem has several phases:

1. Computer 1 sends  $n/2$  numbers to computer 2.
2. Both computers add  $n/2$  numbers simultaneously.
3. Computer 2 sends its partial result back to computer 1.
4. Computer 1 adds the partial sums to produce the final result.

# Example 1

*Computation* (for steps 2 and 4):

$$t_{\text{comp}} = n/2 + 1$$

*Communication* (for steps 1 and 3):

$$t_{\text{comm}} = (t_{\text{startup}} + n/2 t_{\text{data}}) + (t_{\text{startup}} + t_{\text{data}}) = 2t_{\text{startup}} + (n/2 + 1)t_{\text{data}}$$

## Example 2

- Suppose we are to add ‘n’ numbers on 3 processors where each processor adds  $n/3$  numbers. The numbers are initially held by the first processor. The second and the third processors submit the results to the first for adding the partial sums. Assuming the  $t_{\text{start-up}}$  and  $t_{\text{data}}$  as 25 units and 32 units and ‘n’ as 6000, find the ***computation/communication ratio***. Is this system setup, efficient? Why or why not?

## Example 3

- Assume 0.1% of the runtime of a program is not parallelizable. This program is supposed to run on the Tianhe-2 supercomputer, which consists of 3,120,000 cores. Under the assumption that the program runs at the same speed on all of those cores, and there are no additional overheads, what is the parallel speedup on 30,000 and 3,000,000 cores using Amdahl's law?

**Input: f=0.1%, .001**

- For 30000, speed-up is 968
- For 3000000, speed-up is 1000

## Example 4

- 95% of a program's execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?

$$S \leq \frac{1}{0.05 + (1 - 0.05)/8} \cong 5.9$$

## Example 5

- 5% of a parallel program's execution time is spent within inherently sequential code. The maximum speedup achievable by this program, regardless of how many PEs are used, is

$$\lim_{p \rightarrow \infty} \frac{1}{0.05 + (1 - 0.05)/p} = \frac{1}{0.05} = 20$$

## Example 6

- An oceanographer gives you a serial program and asks you how much faster it might run on 8 processors. You can only find one function amenable to a parallel solution. Benchmarking on a single processor reveals 80% of the execution time is spent inside this function. What is the best speedup a parallel version is likely to achieve on 8 processors?

Soln: Show that the answer is about 3.3

## Example 7

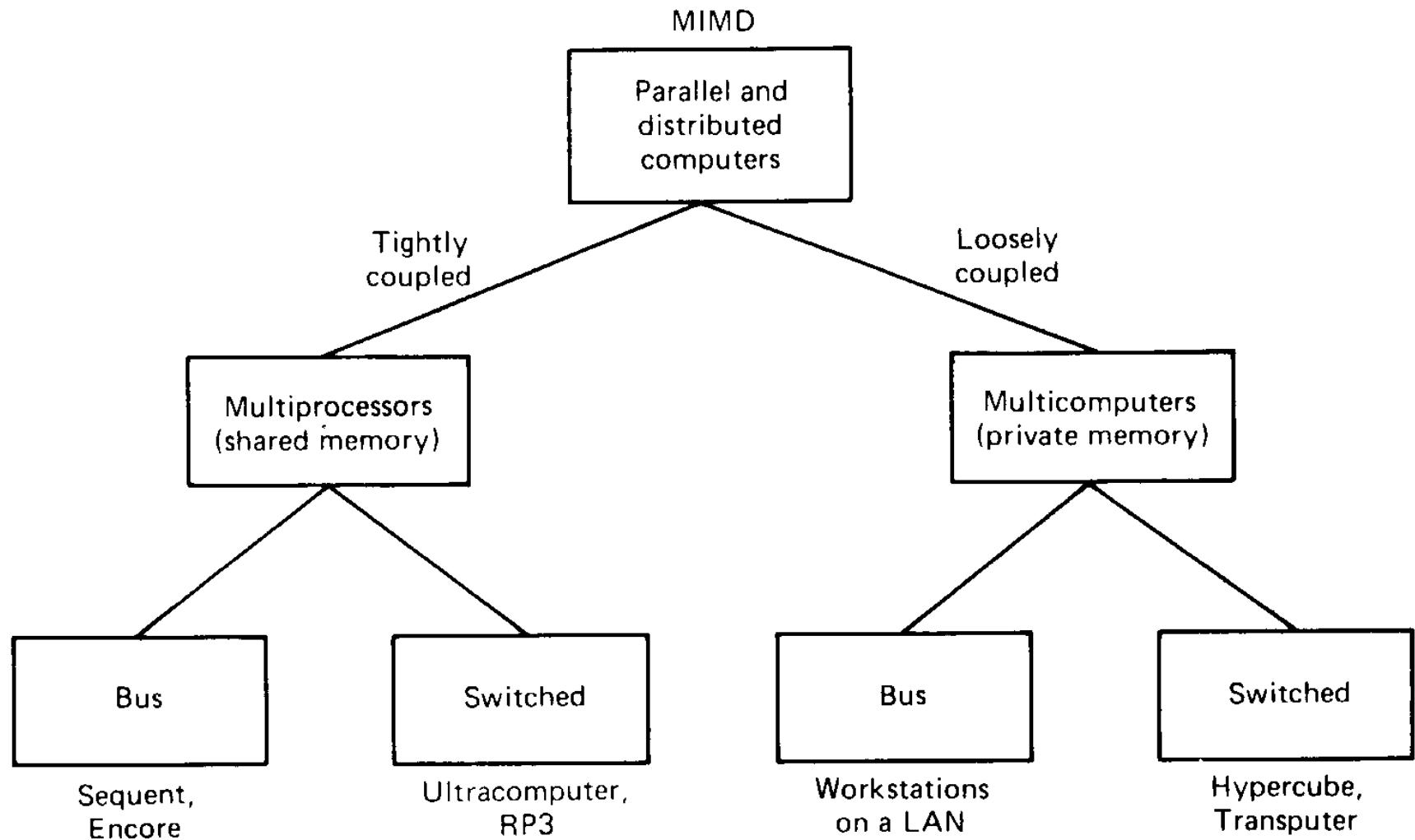
**Example 3 (p.40):** Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

Answer

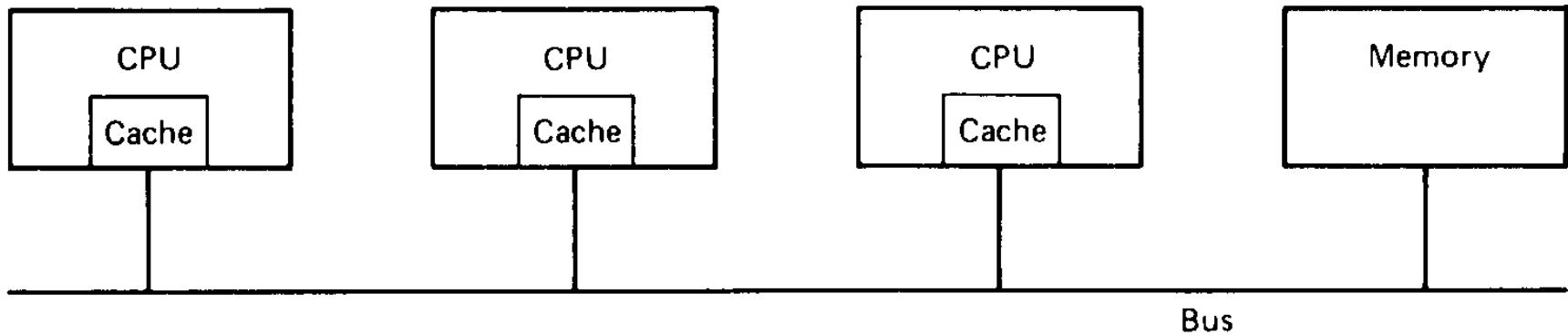
$$\text{Fraction}_{\text{enhanced}} = 0.4, \text{Speedup}_{\text{enhanced}} = 10$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1-0.4)+\frac{0.4}{10}} = \frac{1}{0.6+0.04} = \frac{1}{0.64} \approx 1.56$$

# Communication / Interconnection Networks

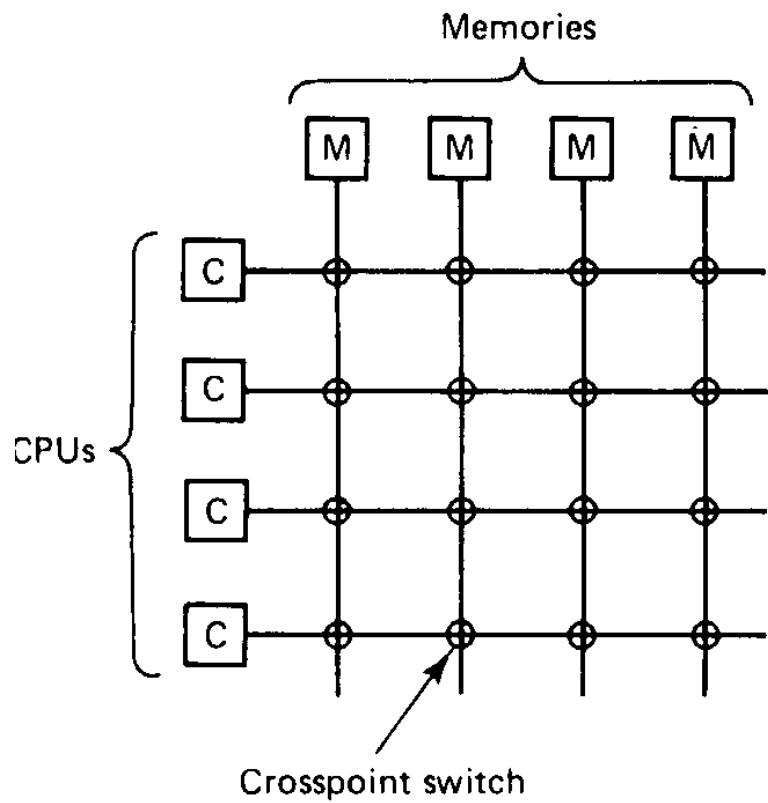


# Bus based Multiprocessors

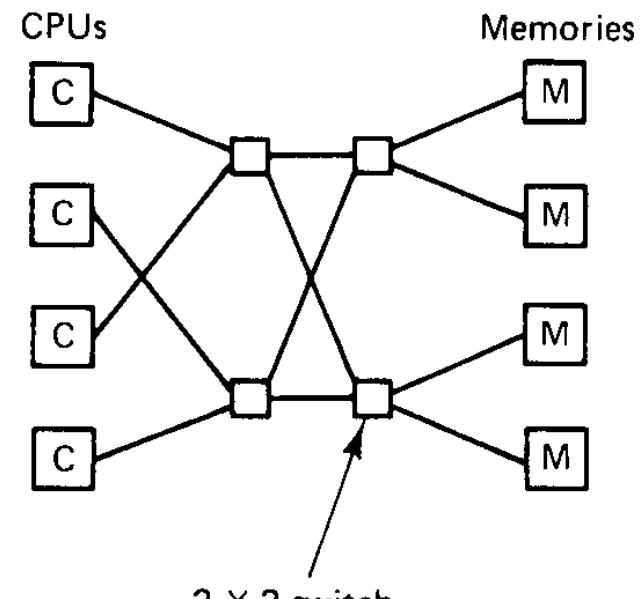


- Data transmitted in the form of packets
- Cache coherency

# Switched Multiprocessors



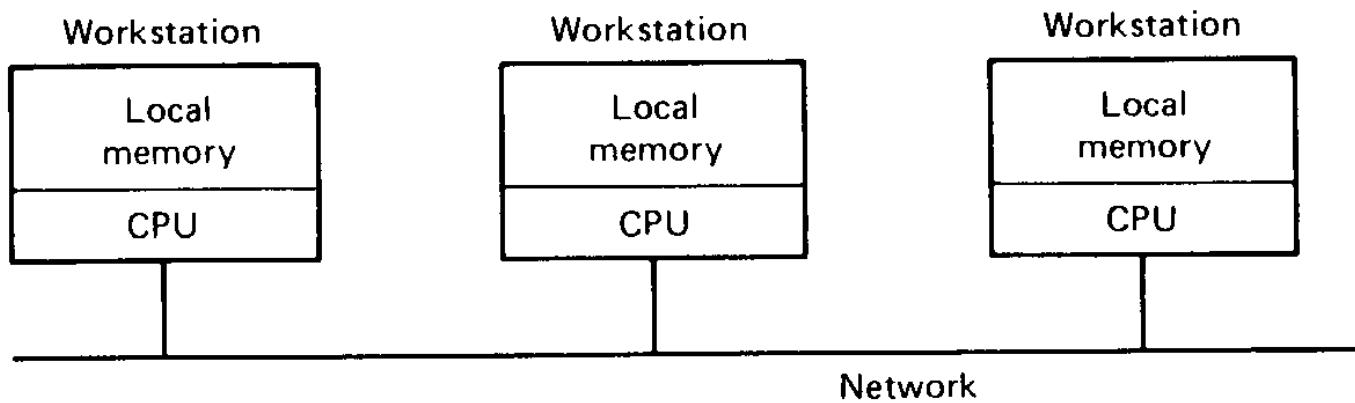
(a)



(b)

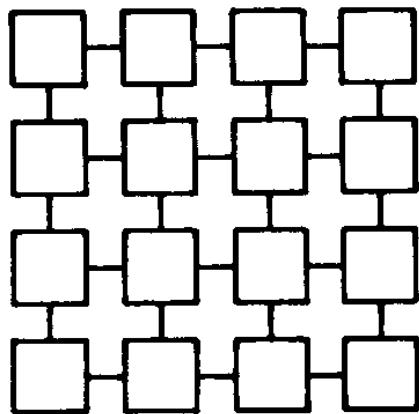
(a) A crossbar switch. (b) An omega switching network.

# Bus based Multicomputers

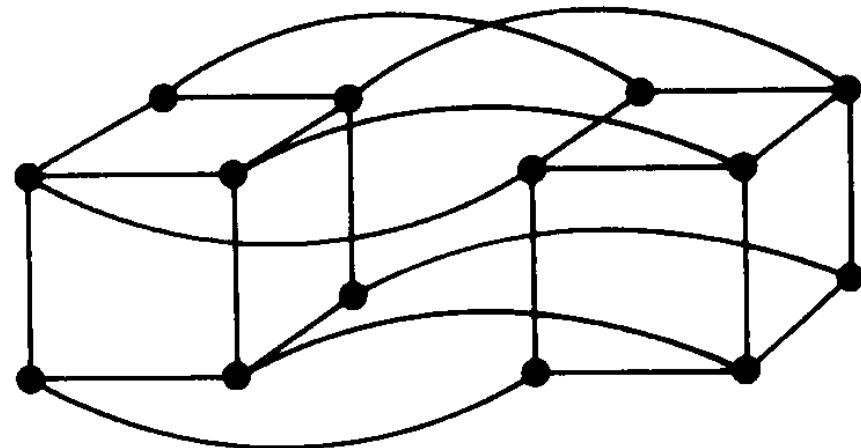


- Limitation – slow as no. of computers increase

# Switched Multicomputers



(a)



(b)

(a) Grid. (b) Hypercube.

# Example 1

- A multicomputer with 256 CPUs is organized as a  $16 \times 16$  grid. What is the worst-case delay (in hops) that a message might have to take?
- **A:** Assuming that routing is optimal, the longest optimal route is from one corner of the grid to the opposite corner. The length of this route is 30 hops. If the end processors in a single row or column are connected to each other, the length becomes 15.

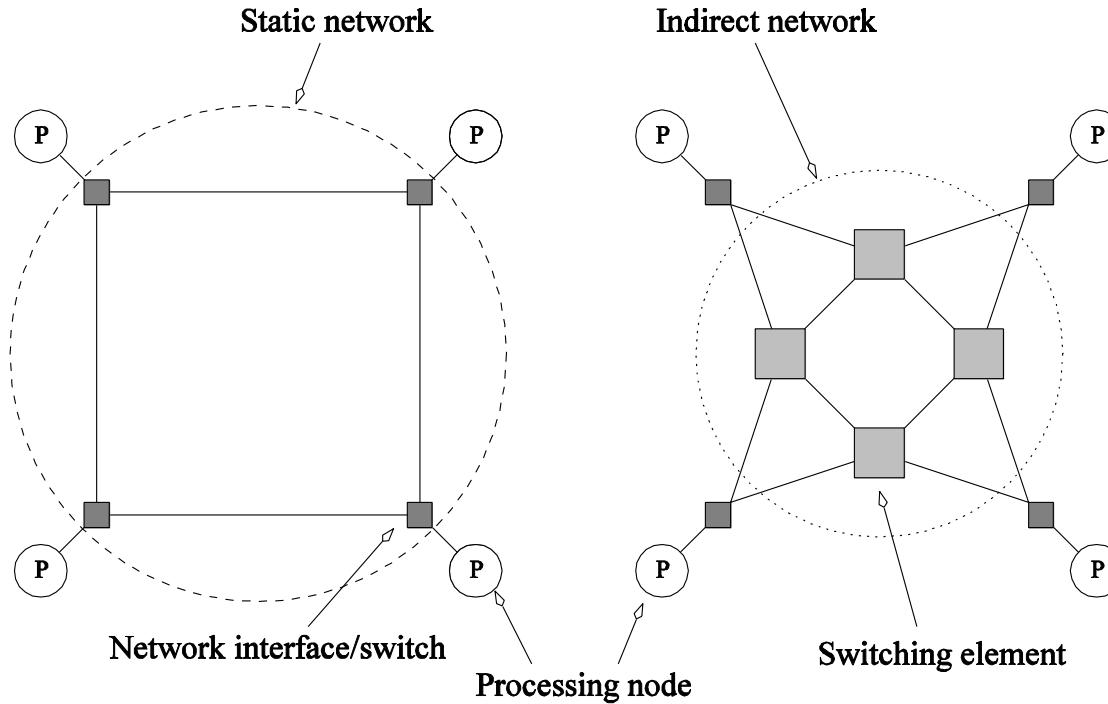
## Example 2

- Now consider a 256-CPU hypercube. What is the worst-case delay here, again in hops?
- **A:** With a 256-CPU hypercube, each node has a binary address, from 00000000 to 11111111. A hop from one machine to another always involves changing a single bit in the address. Thus from 00000000 to 00000001 is one hop. From there to 00000011 is another hop. In all, eight hops are needed.

# Interconnection Networks for Parallel Computers

- Interconnection networks carry data between processors and to memory.
- Interconnects are made of switches and links (wires, fiber).
- Interconnects are classified as static or dynamic.
- Static networks consist of point-to-point communication links among processing nodes and are also referred to as *direct* networks.
- Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as *indirect* networks.

# Static and Dynamic Interconnection Networks



Classification of interconnection networks: (a) a static network; and (b) a dynamic network.

# Properties of a Topology/Network

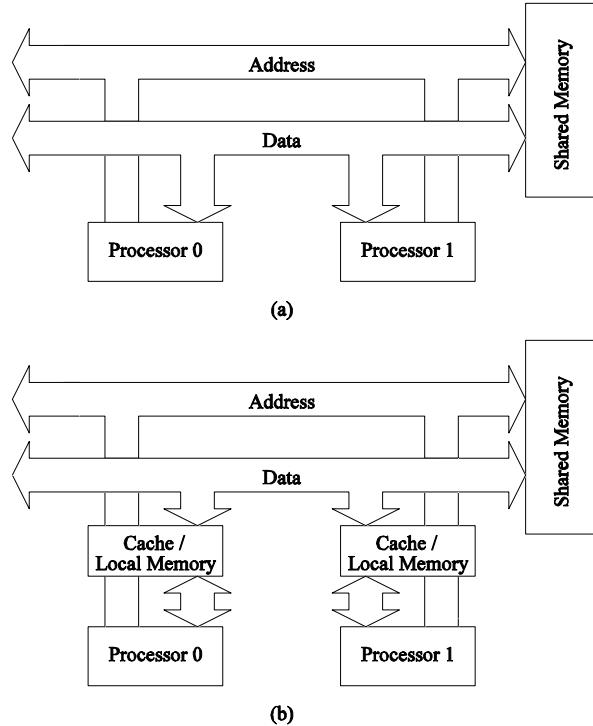
## Bisection Bandwidth

- Often used to describe network performance
- Cut network in half and sum bandwidth of links severed
- $(\text{Min } \# \text{ channels spanning two halves}) * (\text{BW of each channel})$
- Meaningful only for recursive topologies
- Can be misleading, because does not account for switch and routing efficiency

## **Many Topology Examples**

- **Bus**
- **Crossbar**
- **Ring**
- **Tree**
- **Omega**
- **Hypercube**
- **Mesh**
- **Torus**
- **Butterfly**
- ...

# Buses

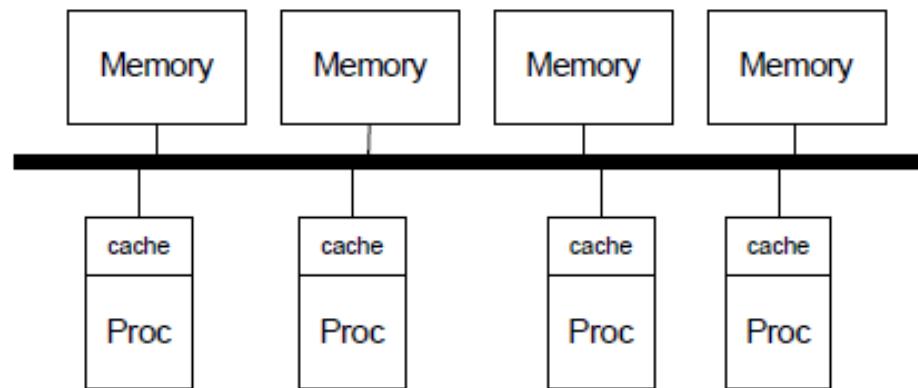


Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

Since much of the data accessed by processors is local to the processor, a local memory can improve the performance of bus-based machines.

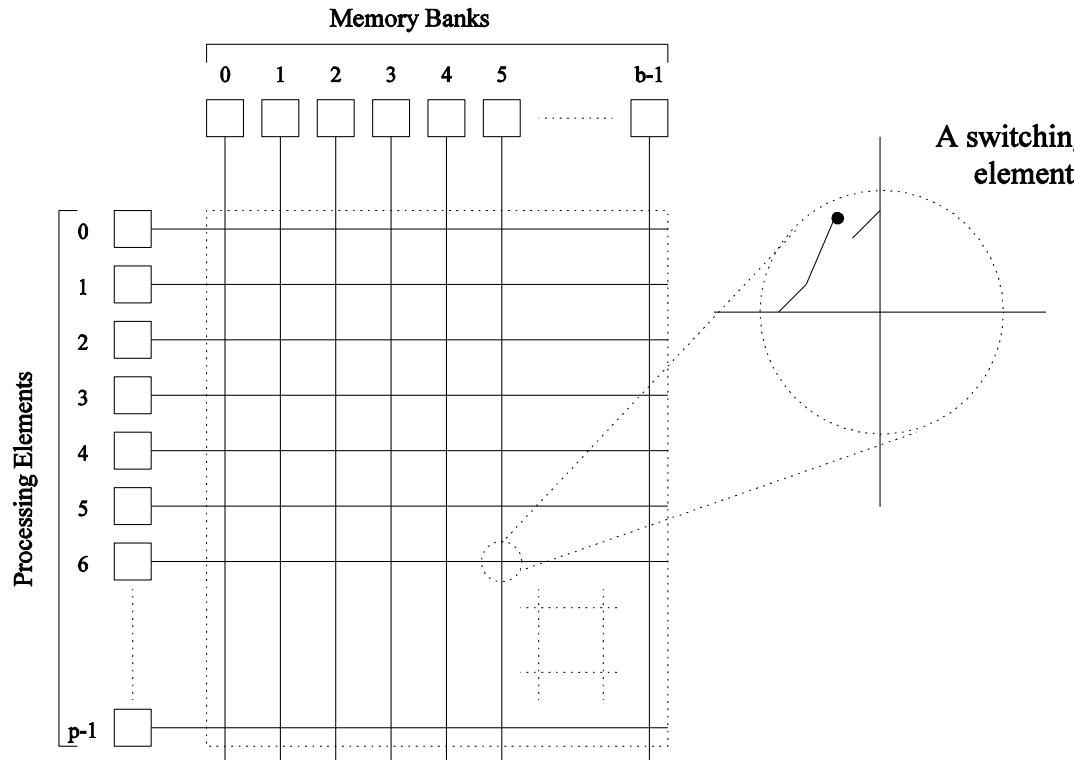
# Bus

- + Simple
- + Cost effective for a small number of nodes
- + Easy to implement coherence (snooping)
- Not scalable to large number of nodes
  - (limited bandwidth, electrical loading → reduced frequency)
- High contention

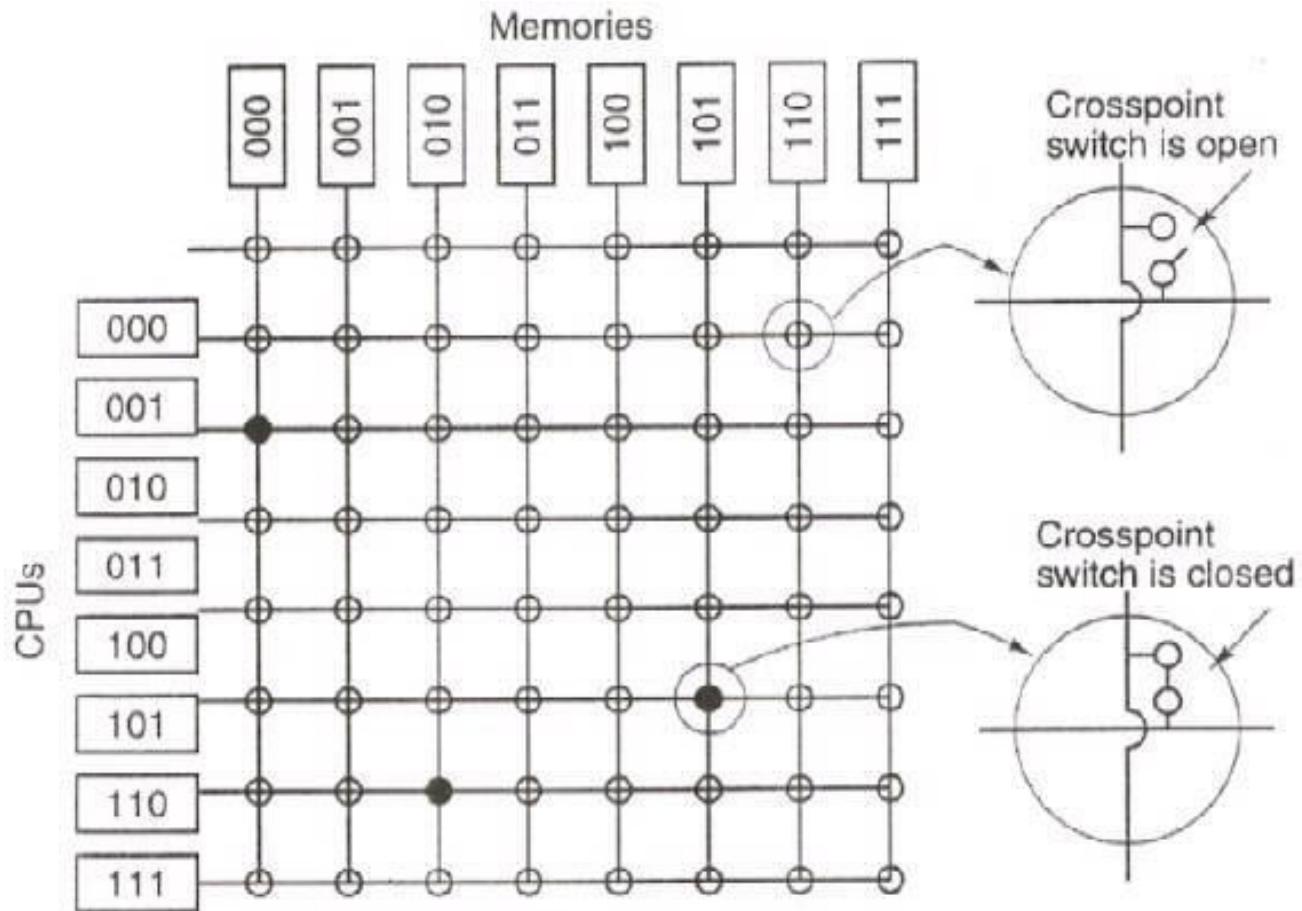


# Crossbars

A crossbar network uses an  $p \times m$  grid of switches to connect  $p$  inputs to  $m$  outputs in a non-blocking manner.



A completely non-blocking crossbar network connecting  $p$  processors to  $b$  memory banks.

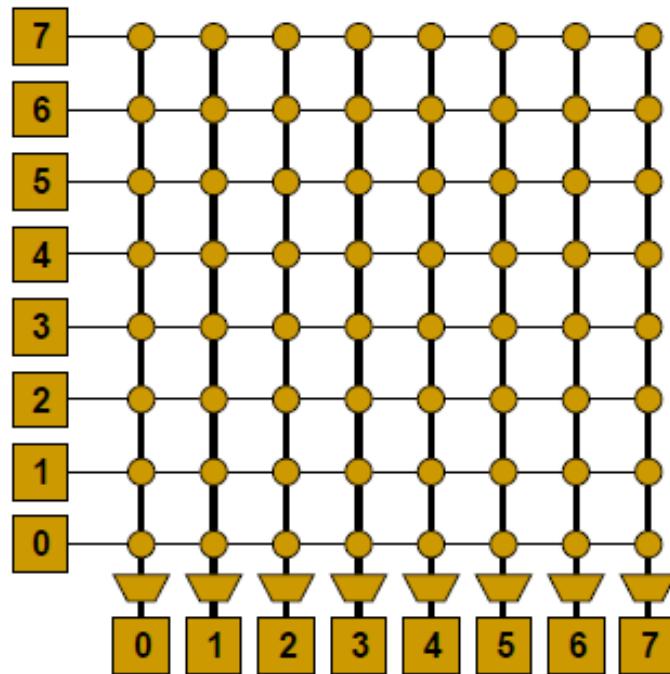


# Crossbar

- Every node connected to all others (non-blocking)
- Good for small number of nodes
- + Low latency and high throughput
- Expensive
- Not scalable  $\rightarrow O(N^2)$  cost
- Difficult to arbitrate

Core-to-cache-bank networks:

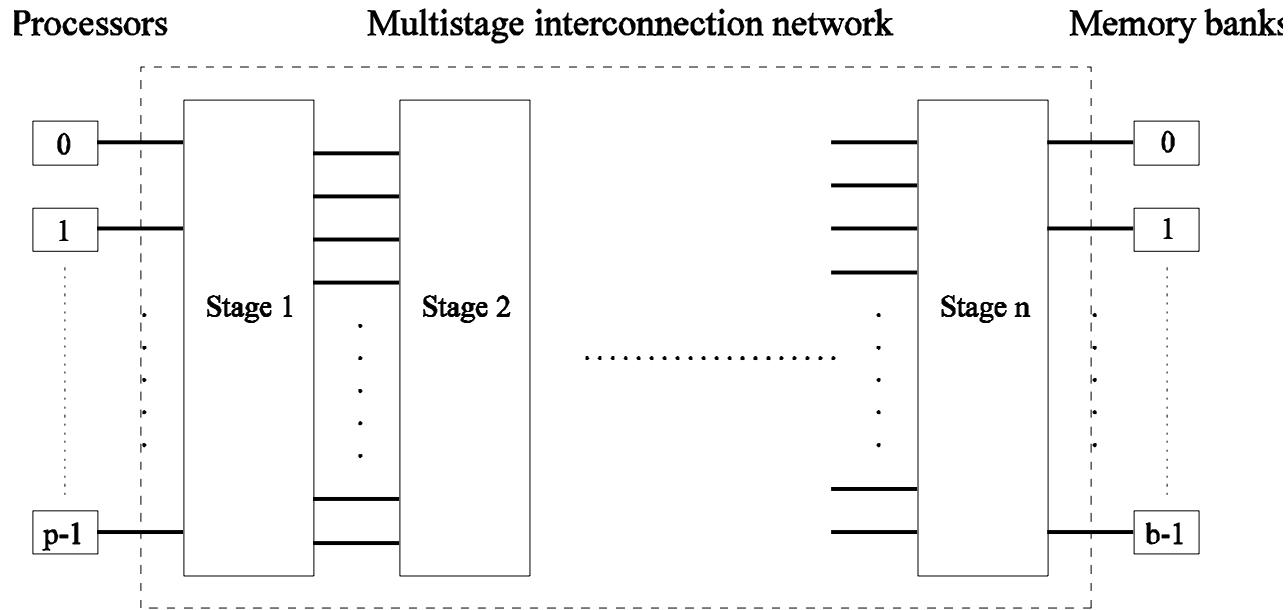
- IBM POWER5
- Sun Niagara I/II



# Multistage Networks

- Crossbars have excellent performance scalability but poor cost scalability.
- Buses have excellent cost scalability, but poor performance scalability.
- Multistage interconnects strike a compromise between these extremes.

# Multistage Networks



The schematic of a typical multistage interconnection network.

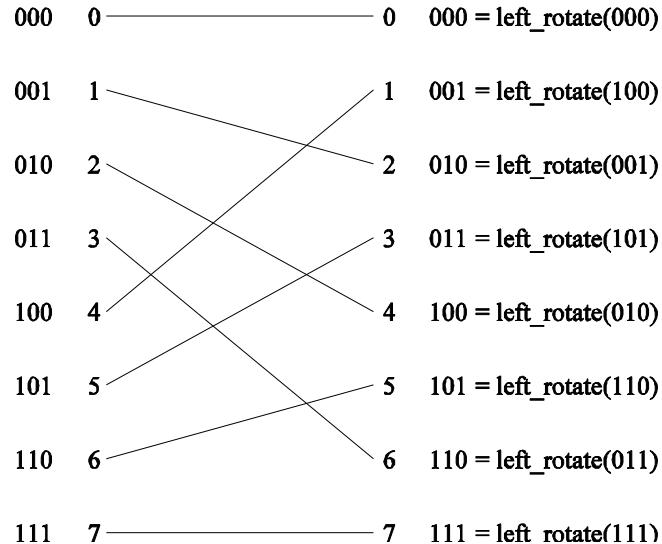
# Multistage Networks

- One of the most commonly used multistage interconnects is the Omega network.
- This network consists of  $\log p$  stages, where  $p$  is the number of inputs/outputs.
- At each stage, input  $i$  is connected to output  $j$  if:

$$j = \begin{cases} 2i, & 0 \leq i \leq p/2 - 1 \\ 2i + 1 - p, & p/2 \leq i \leq p - 1 \end{cases}$$

# Multistage Omega Networks

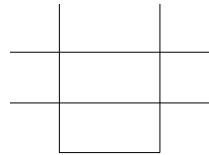
Each stage of the Omega network implements a perfect shuffle as follows:



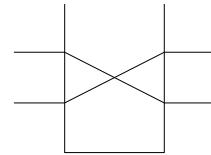
A perfect shuffle interconnection for eight inputs and outputs.

# Multistage Omega Network

- The perfect shuffle patterns are connected using  $2 \times 2$  switches.
- The switches operate in two modes – crossover or passthrough.



(a)



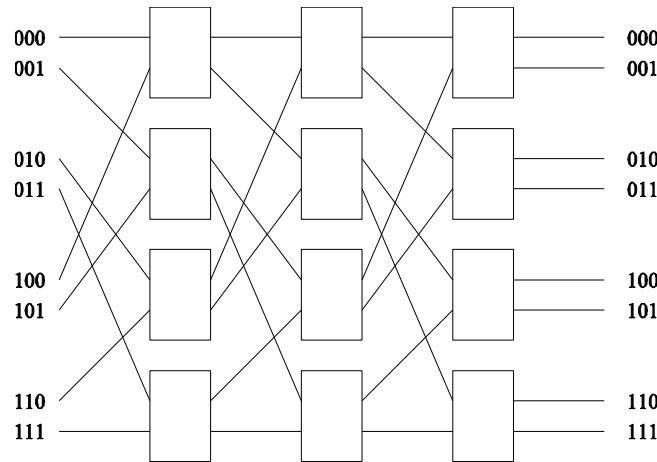
(b)

Two switching configurations of the  $2 \times 2$  switch:

(a) Pass-through; (b) Cross-over.

# Multistage Omega Network

A complete Omega network with the perfect shuffle interconnects and switches can now be illustrated:



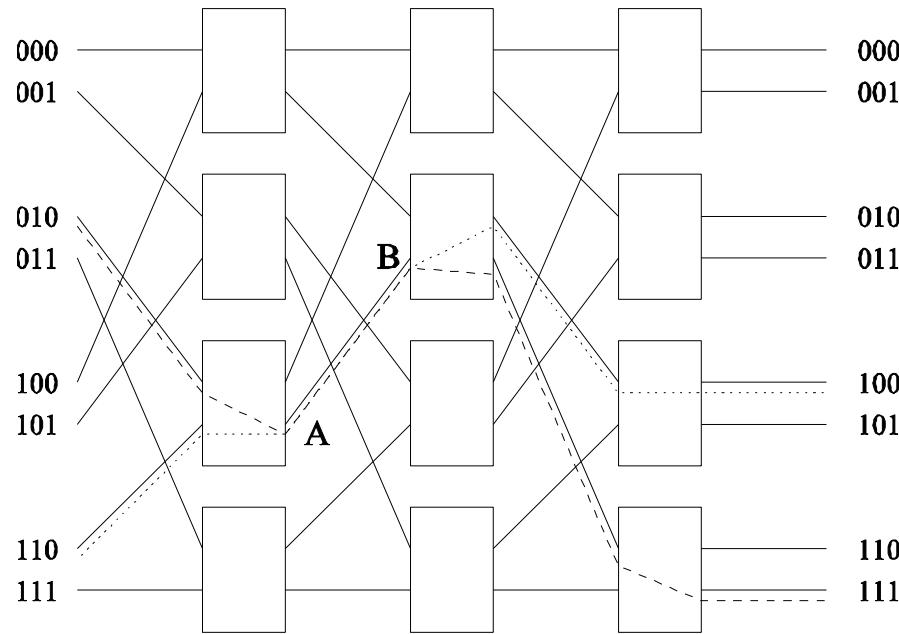
A complete omega network connecting eight inputs and eight outputs.

An omega network has  $p/2 \times \log p$  switching nodes, and the cost of such a network grows as  $(p \log p)$ .

# Multistage Omega Network – Routing

- Let  $s$  be the binary representation of the source and  $d$  be that of the destination processor.
- The data traverses the link to the first switching node. If the most significant bits of  $s$  and  $d$  are the same, then the data is routed in pass-through mode by the switch else, it switches to crossover.
- This process is repeated for each of the  $\log p$  switching stages.
- Note that this is not a non-blocking switch.

# Multistage Omega Network – Routing



An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.

# Example 3

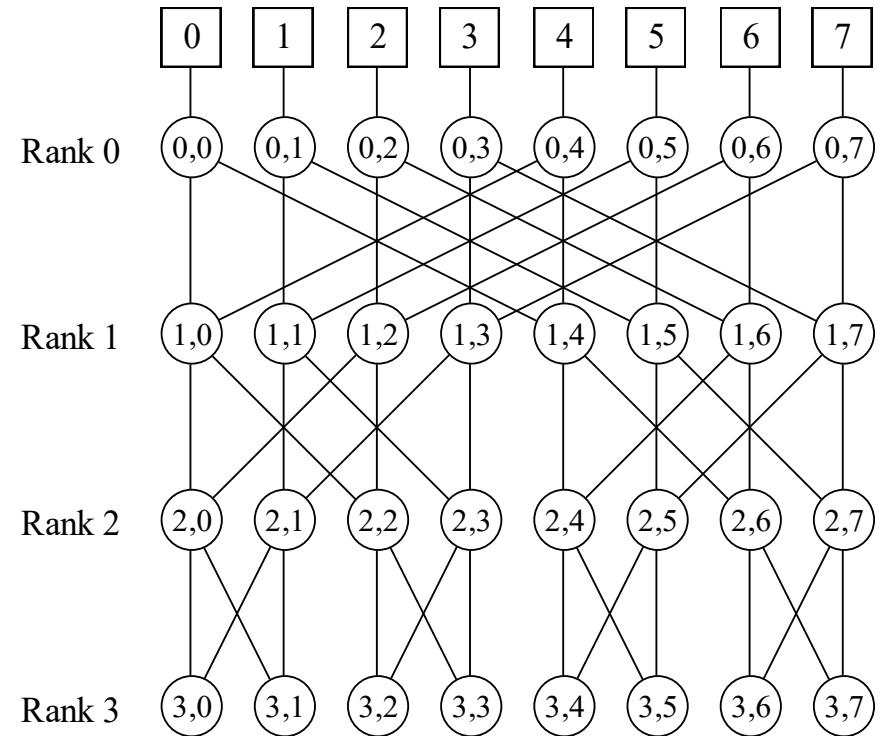
- A multiprocessor has 1024 100-MIPS CPUs connected to memory by an omega network. How fast do the switches have to be to allow a request to go to memory and back in one instruction time?
- Solution: 5120 0.5 nano sec switches

# Example 4

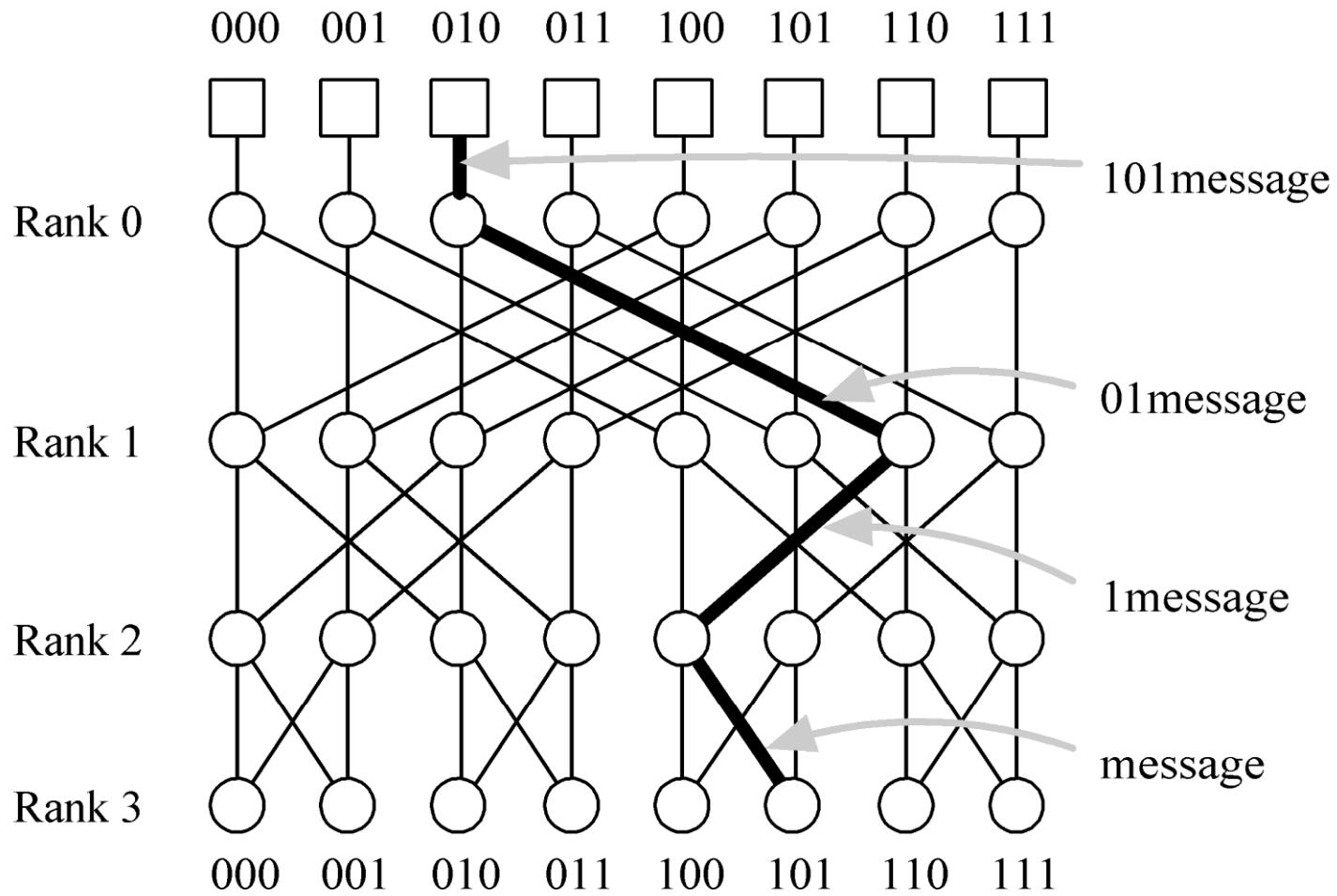
- A multiprocessor has 4096 50-MIPS CPUs connected to memory by an omega network. How fast do the switches have to be to allow a request to go to memory and back in one instruction time?
- Solution: 1 instruction= ~~20 nano sec~~  
1 switching stage=~~0.833 nano sec~~  
~~2048x12 0.833 nano second switches~~

# Butterfly Network

- Indirect topology
- $n = 2^d$  processor nodes connected by  $n(\log n + 1)$  switching nodes
- Rows are labeled 0 ... n. Each processor has four connections to other processors (except processors in top and bottom row).
- Processor  $P(r, j)$ , i.e. processor number j in row r is connected to  $P(r-1, j)$  and  $P(r-1, m)$  where m is obtained by inverting the  $r^{\text{th}}$  significant bit in the binary representation of j.



# Butterfly Network Routing



# Evaluating Butterfly Network

- Diameter:  $\log n$
- Bisection width:  $n / 2$
- Edges per node: 4
- Constant edge length? No

# Completely Connected Network

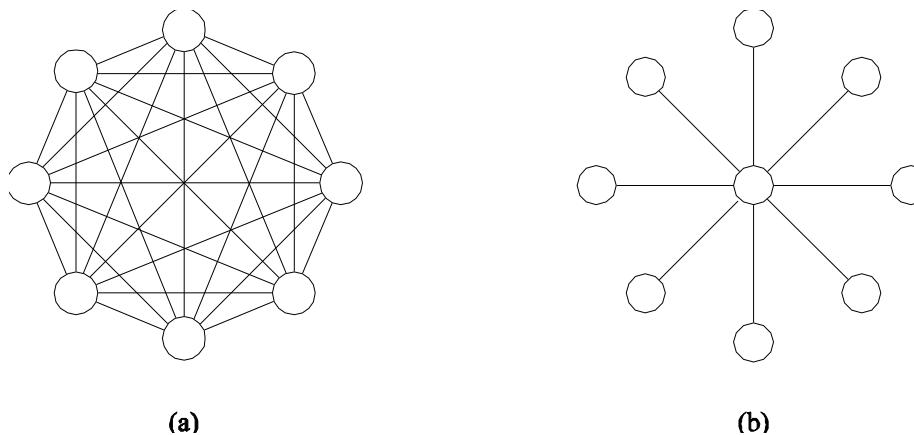
- Each processor is connected to every other processor.
- The number of links in the network scales as  $O(p^2)$ .
- While the performance scales very well, the hardware complexity is not realizable for large values of  $p$ .
- In this sense, these networks are static counterparts of crossbars.

# Star Connected Network

- Every node is connected only to a common node at the center.
- Distance between any pair of nodes is  $O(1)$ . However, the central node becomes a bottleneck.
- In this sense, star connected networks are static counterparts of buses.

# Completely Connected and Star Connected Networks

Example of an 8-node completely connected network.

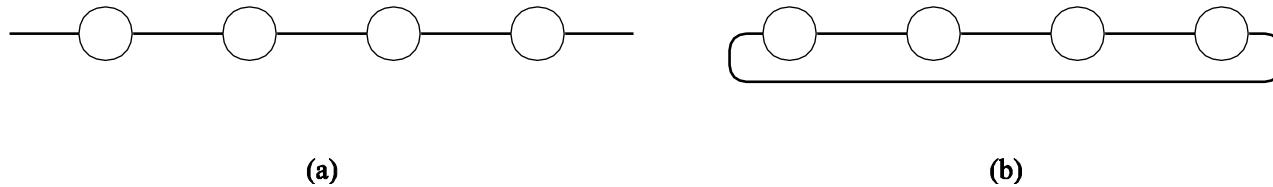


- (a) A completely-connected network of eight nodes;
- (b) a star connected network of nine nodes.

# Linear Arrays, Meshes, and $k$ - $d$ Meshes

- In a linear array, each node has two neighbors, one to its left and one to its right. If the nodes at either end are connected, we refer to it as a 1-D torus or a ring.
- A generalization to 2 dimensions has nodes with 4 neighbors, to the north, south, east, and west.
- A further generalization to  $d$  dimensions has nodes with  $2d$  neighbors.
- A special case of a  $d$ -dimensional mesh is a hypercube. Here,  $d = \log p$ , where  $p$  is the total number of nodes.

# Linear Arrays



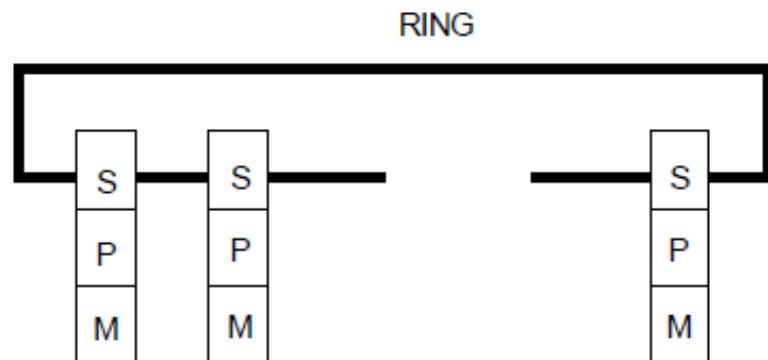
Linear arrays: (a) with no wraparound links; (b) with wraparound link.

# Ring

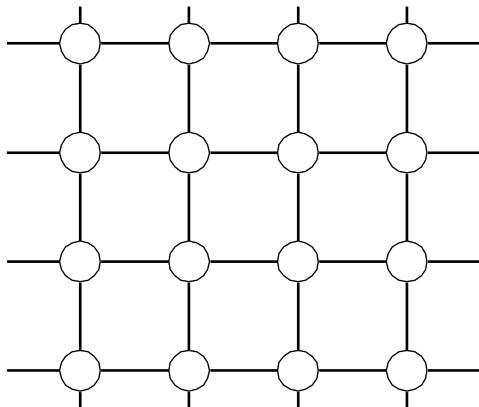
- + Cheap:  $O(N)$  cost
- High latency:  $O(N)$
- Not easy to scale
- Bisection bandwidth remains constant

Used in:

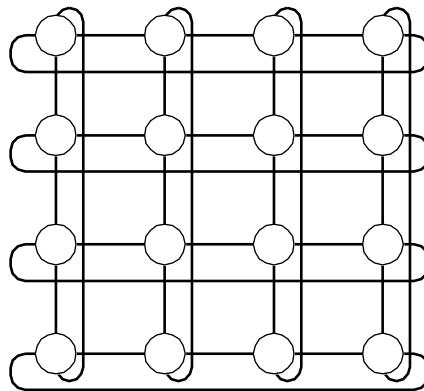
- Intel Larrabee/Core i7
- IBM Cell



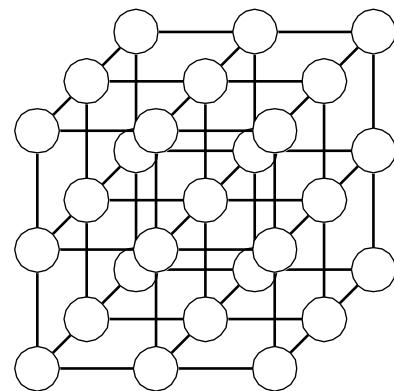
# Two- and Three Dimensional Meshes



(a)



(b)

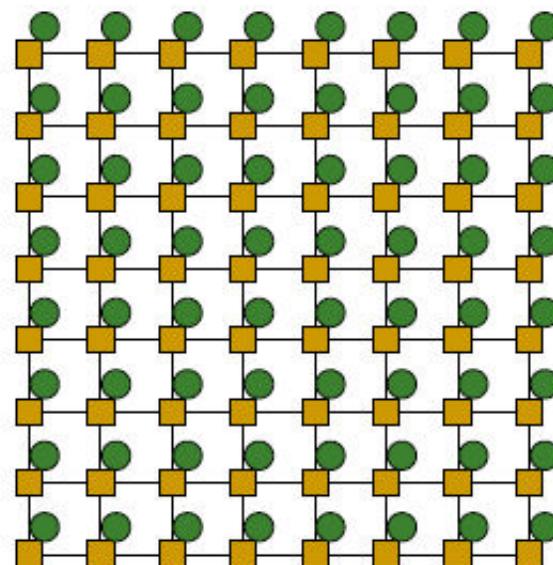


(c)

Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

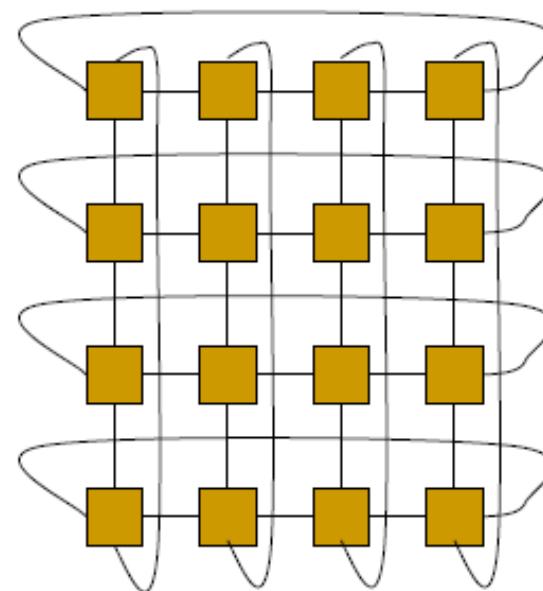
# Mesh

- $O(N)$  cost
- Average latency:  $O(\sqrt{N})$
- Easy to layout on-chip: regular & equal-length links
- Path diversity: many ways to get from one node to another
- Used in:
  - Tilera 100-core CMP
  - On-chip network prototypes

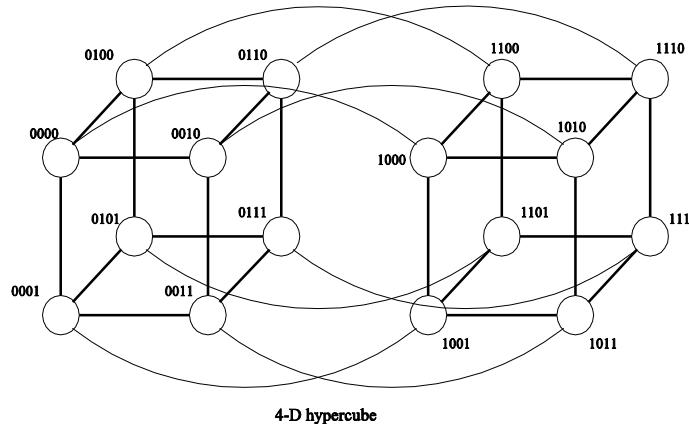
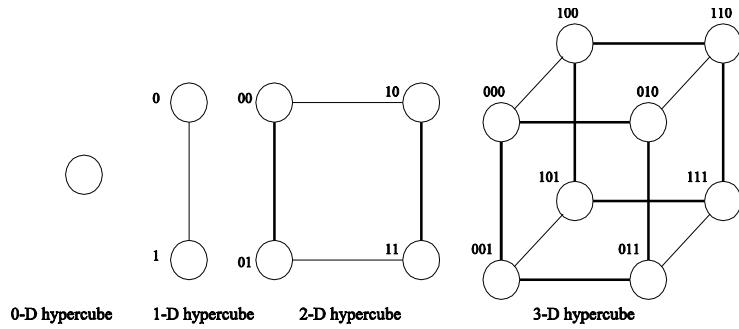


# Torus

- Mesh is not symmetric on edges: performance very sensitive to placement of task on edge vs. middle
- Torus avoids this problem
  - + Higher path diversity (& bisection bandwidth) than mesh
  - Higher cost
  - Harder to lay out on-chip
  - Unequal link lengths



# Hypercubes and their Construction



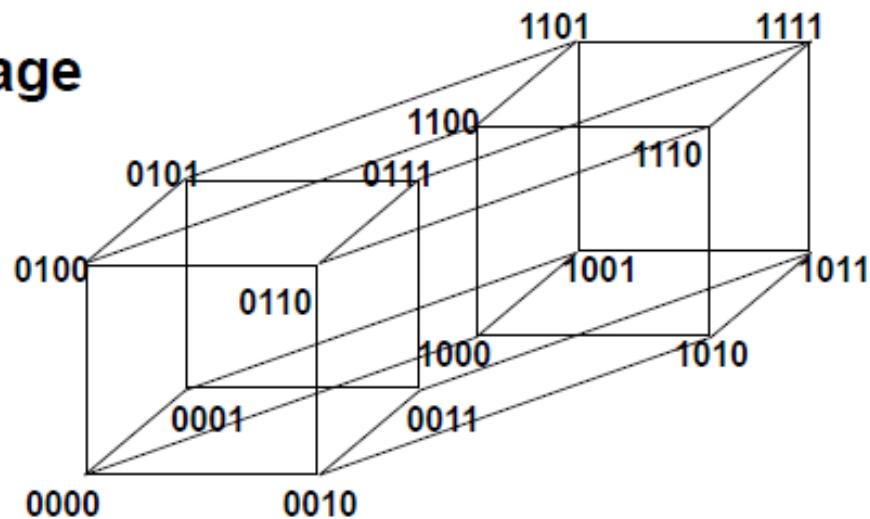
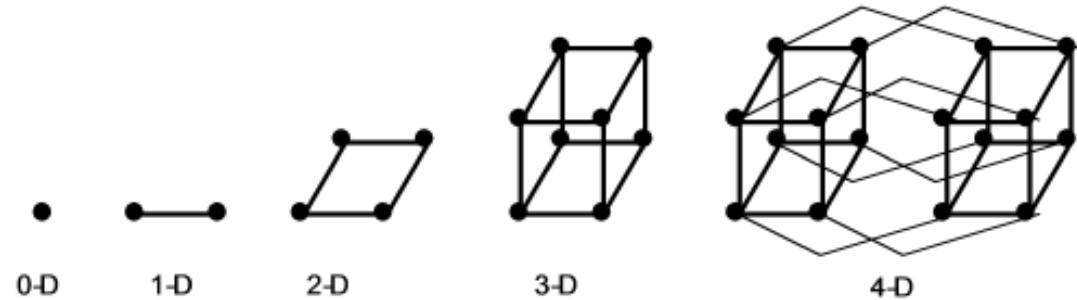
Construction of hypercubes from hypercubes of lower dimension.

# Properties of Hypercubes

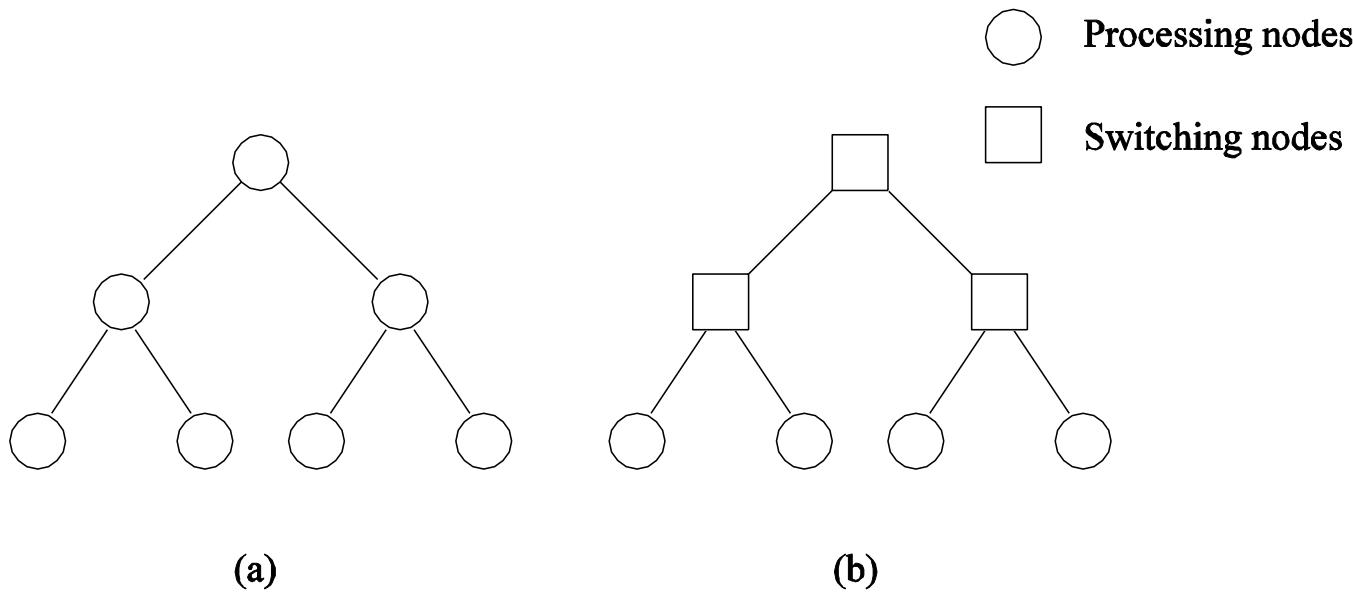
- The distance between any two nodes is at most  $\log p$ .
- Each node has  $\log p$  neighbors.
- The distance between two nodes is given by the number of bit positions at which the two nodes differ.

# Hypercube

- Latency:  $O(\log N)$
  - Radix:  $O(\log N)$
  - #links:  $O(N \log N)$
- + Low latency
- Hard to lay out in 2D/3D
- Used in some early message passing machines, e.g.:
    - Intel iPSC
    - nCube



# Tree-Based Networks

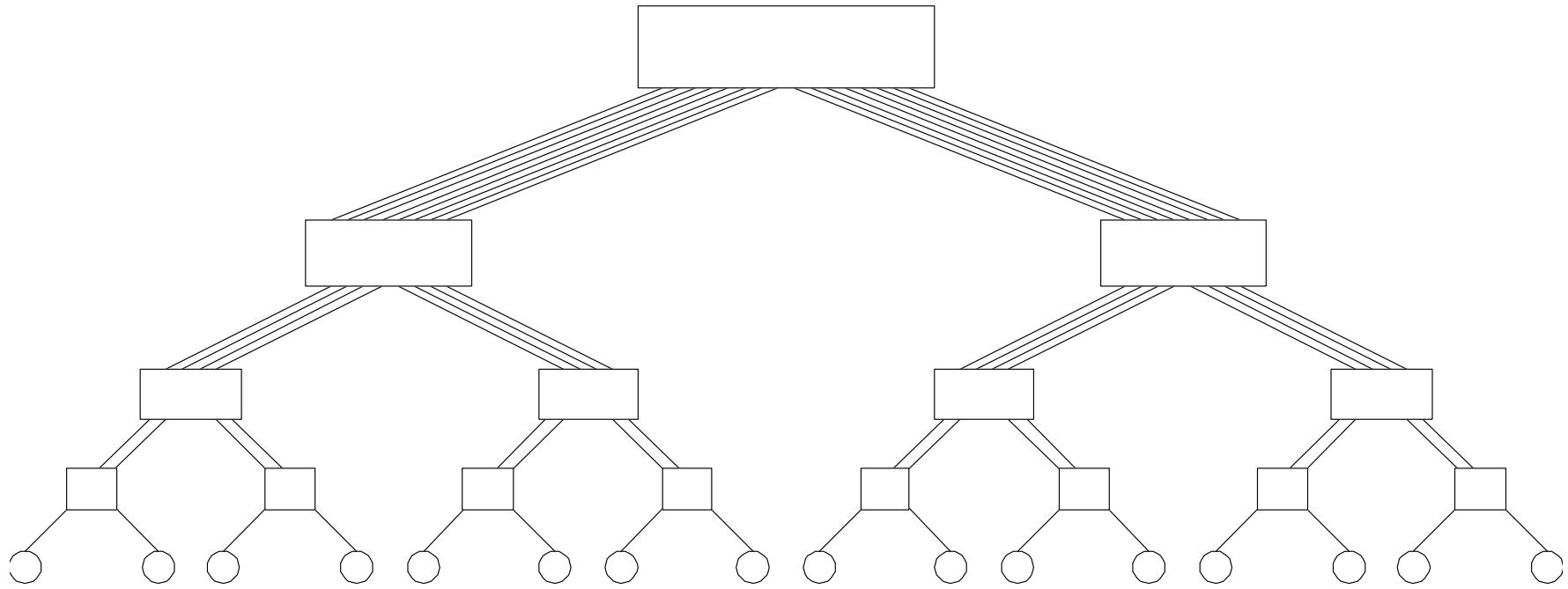


Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

# Tree Properties

- The distance between any two nodes is no more than  $2\log p$ .
- Links higher up the tree potentially carry more traffic than those at the lower levels.
- For this reason, a variant called a fat-tree, fattens the links as we go up the tree.
- Trees can be laid out in 2D with no wire crossings. This is an attractive property of trees.

# Fat Trees



A fat tree network of 16 processing nodes.

# Trees

Planar, hierarchical topology

Latency:  $O(\log N)$

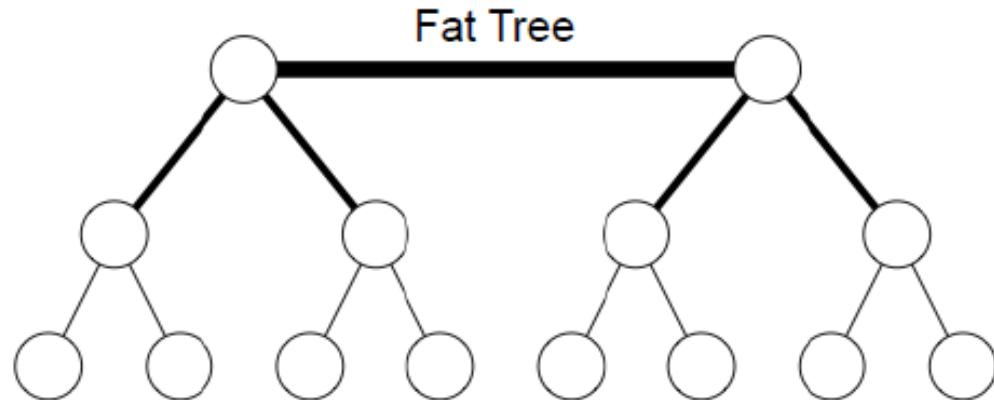
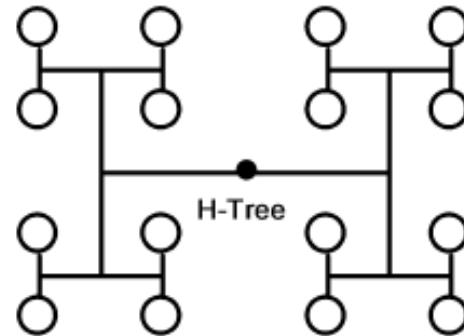
Good for local traffic

+ Cheap:  $O(N)$  cost

+ Easy to Layout

- Root can become a bottleneck

Fat trees avoid this problem (CM-5)



# Evaluating Static Interconnection Networks

- *Diameter*: The distance between the farthest two nodes in the network. The diameter of a linear array is  $p - 1$ , that of a mesh is  $2\sqrt{p} - 1$ , that of a tree and hypercube is  $\log p$ , and that of a completely connected network is  $O(1)$ .
- *Bisection Width*: The minimum number of wires you must cut to divide the network into two equal parts. The bisection width of a linear array and tree is 1, that of a mesh is  $\sqrt{p}$ , that of a hypercube is  $p/2$  and that of a completely connected network is  $p^2/4$ .
- *Cost*: The number of links or switches (whichever is asymptotically higher) is a meaningful measure of the cost. However, a number of other factors, such as the ability to layout the network, the length of wires, etc., also factor in to the cost.

# Evaluating Static Interconnection Networks

- The number of bits that can be communicated simultaneously over a link connecting two nodes is called the ***channel width***. Channel width is equal to the number of physical wires in each communication link.
- The peak rate at which a single physical wire can deliver bits is called the ***channel rate***.
- The peak rate at which data can be communicated between the ends of a communication link is called ***channel bandwidth***.
- Channel bandwidth is the product of channel rate and channel width.
- The ***bisection bandwidth*** of a network is defined as the minimum volume of communication allowed between any two halves of the network. It is the product of the bisection width and the channel bandwidth. Bisection bandwidth of a network is also sometimes referred to as ***crosssection bandwidth***.

# Evaluating Static Interconnection Networks

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p - 1$	$p(p - 1)/2$
Star	2	1	1	$p - 1$
Complete binary tree	$2 \log((p + 1)/2)$	1	1	$p - 1$
Linear array	$p - 1$	1	1	$p - 1$
2-D mesh, no wraparound	$2(\sqrt{p} - 1)$	$\sqrt{p}$	2	$2(p - \sqrt{p})$
2-D wraparound mesh	$2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p \log p)/2$
Wraparound $k$ -ary $d$ -cube	$d\lfloor k/2 \rfloor$	$2k^{d-1}$	$2d$	$dp$

# Evaluating Dynamic Interconnection Networks

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Crossbar	1	$p$	1	$p^2$
Omega Network	$\log p$	$p/2$	2	$p/2$
Dynamic Tree	$2 \log p$	1	2	$p - 1$

# Communication Primitives

# Process Communication

- The single most important difference between a distributed system and a uniprocessor system is the **interprocess communication**.
- In a uniprocessor system, interprocess communication assumes the existence of shared memory.
- A typical example is the producer-consumer problem.
- One process   **writes to → buffer → reads from**   another process
- The most basic form of synchronization, the semaphore requires **one word** (the semaphore variable) to be shared.

# Process Communication

- In a distributed system, there's no shared memory, so the entire nature of interprocess communication must be completely rethought from scratch.
- All communication in distributed system is based on **message passing**.

E.g. Proc. A wants to communicate with Proc. B

1. It first builds a message in its own address space
2. It executes a system call
3. The OS fetches the message and sends it through network to B.

# Process Communication

- A and B have to agree on the meaning of the bits being sent. For example,
  - How many volts should be used to signal a 0-bit? 1-bit?
  - How does the receiver know which is the last bit of the message?
  - How can it detect if a message has been damaged or lost?
  - What should it do if it finds out?
  - How long are numbers, strings, and other data items? And how are they represented?

# Types of Communication Primitives

Widely used communication primitives in Distributed Operating Systems

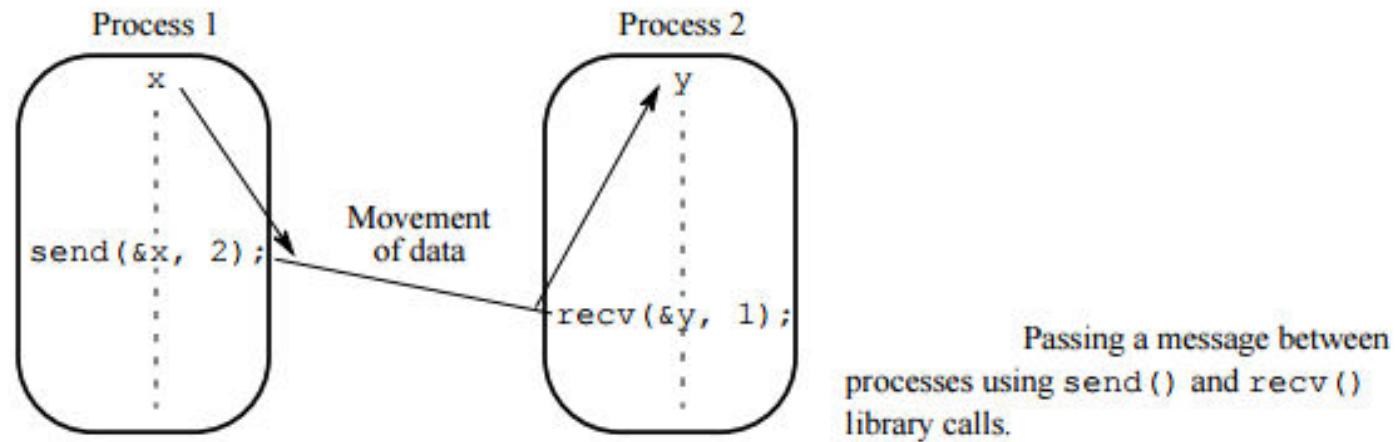
- Message Passing
  - Using Send() and Receive() Primitives
- Remote Procedure Calls
  - An extension of conventional procedure call (used for transfer of control and data within a single process)

# Message Passing Model

- Two basic communication primitives
  - Send() and Receive()
  - Example:  
`send(&x, destination_id)`  
`receive(&y, source_id)`
- Found in Client/Server computing models

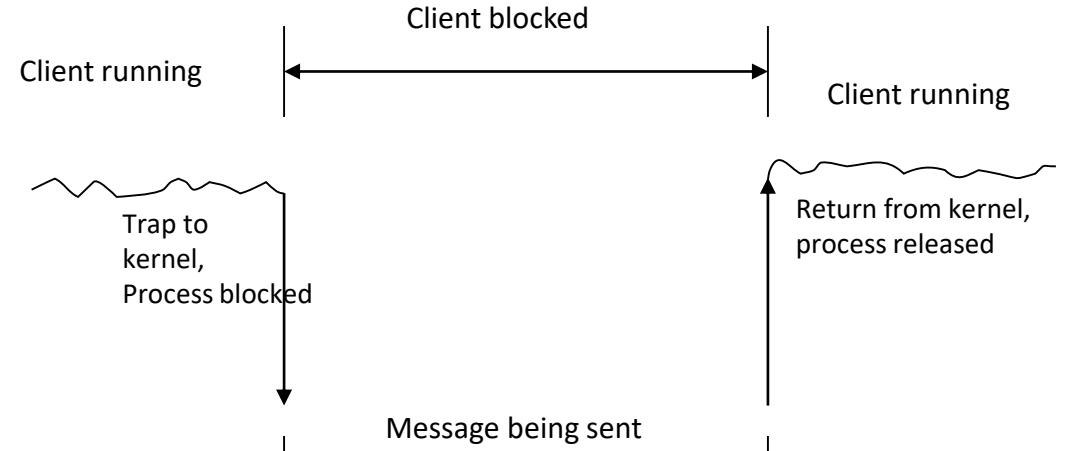
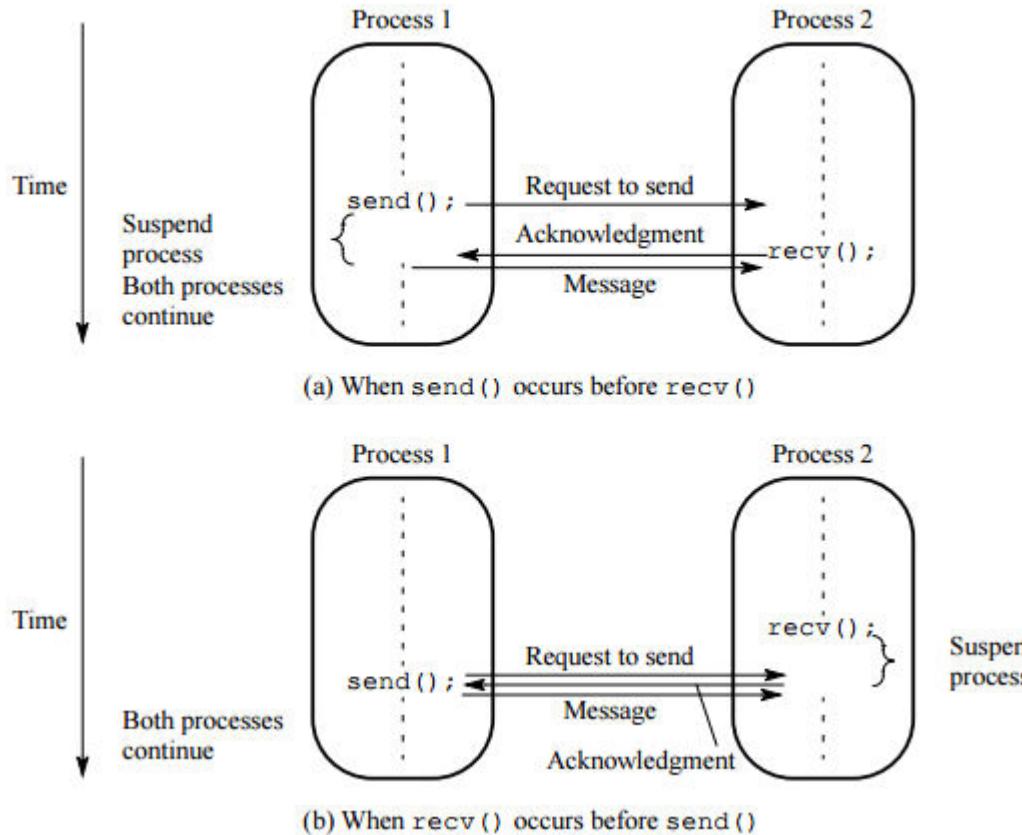
# Message passing

- Synchronous and asynchronous send routines

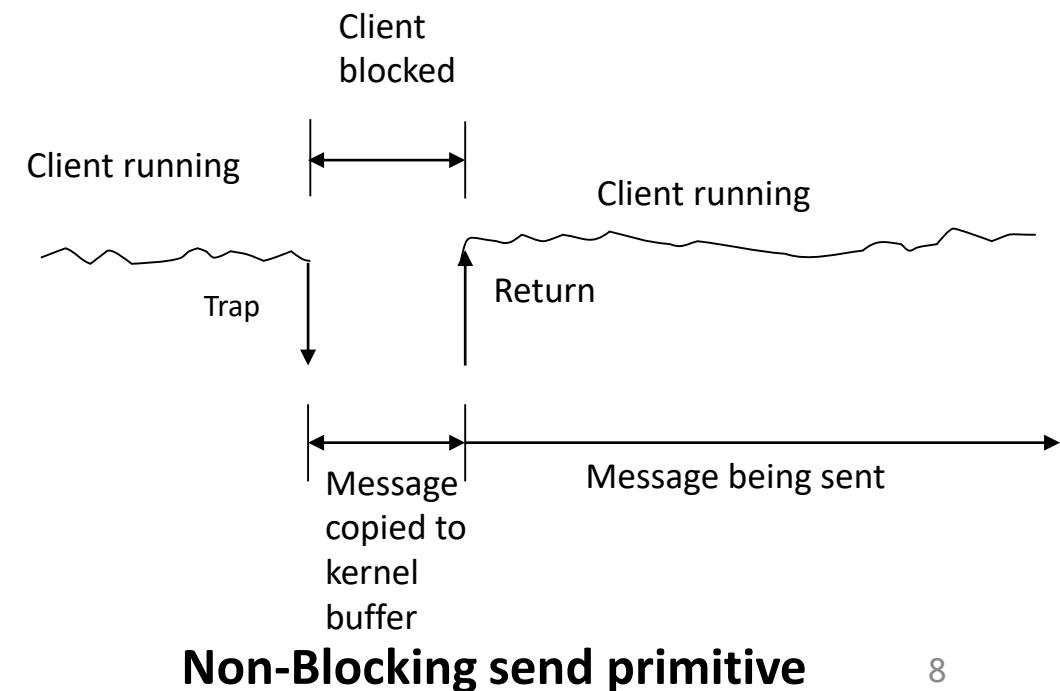


# Message passing

- Blocking and non-blocking routines



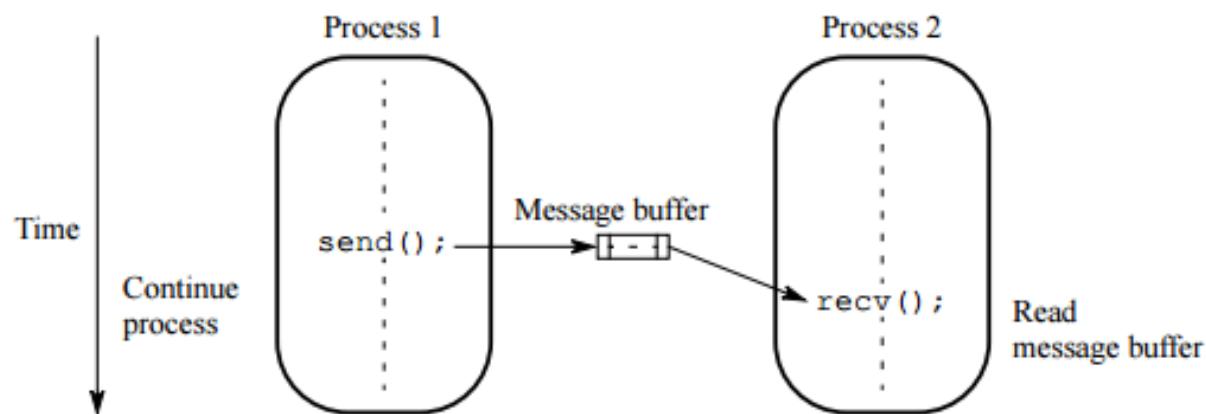
**Blocking send primitive**



**Non-Blocking send primitive**

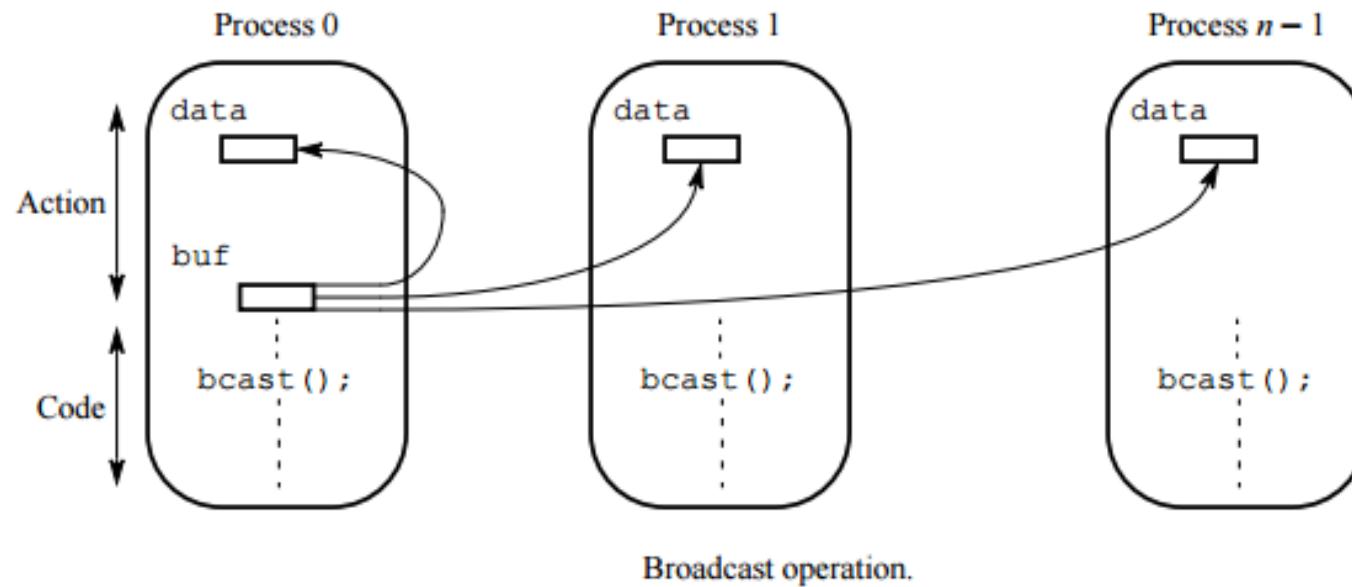
# Using a message buffer

## Buffered versus Unbuffered Primitives

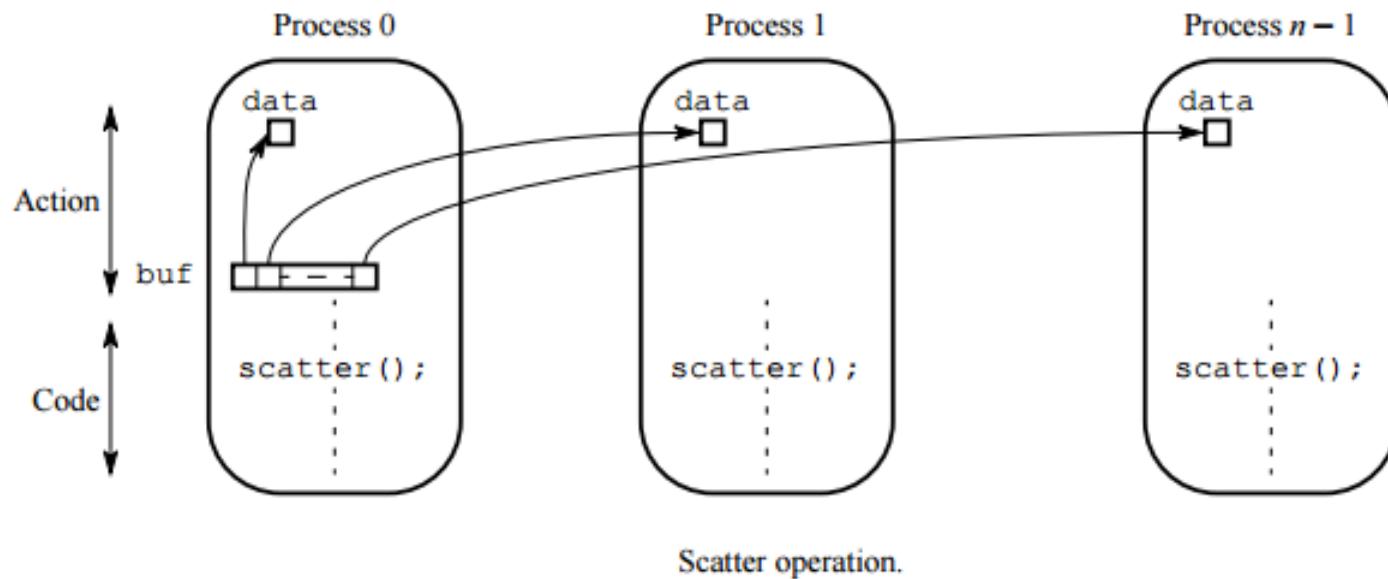


- No buffer allocated. Fine if `receive()` is called before `send()`.
- Buffers allocated, freed, and managed to store the incoming message. Usually a mailbox created.

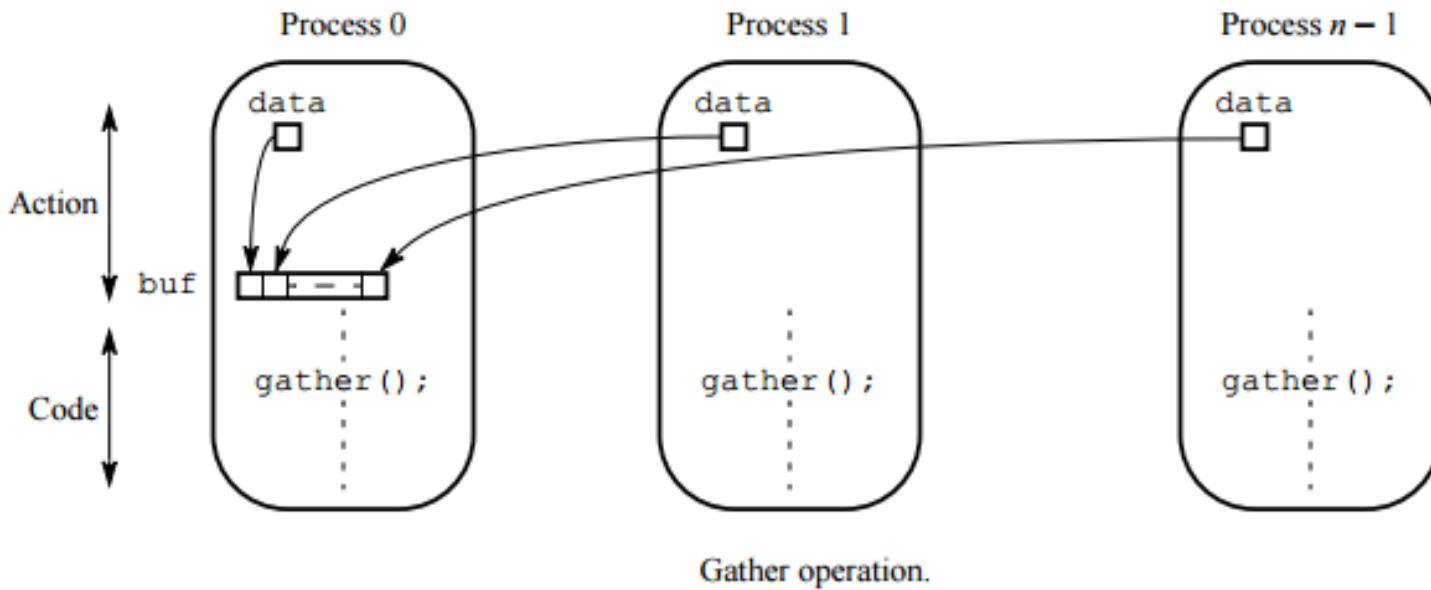
# Group communication routines



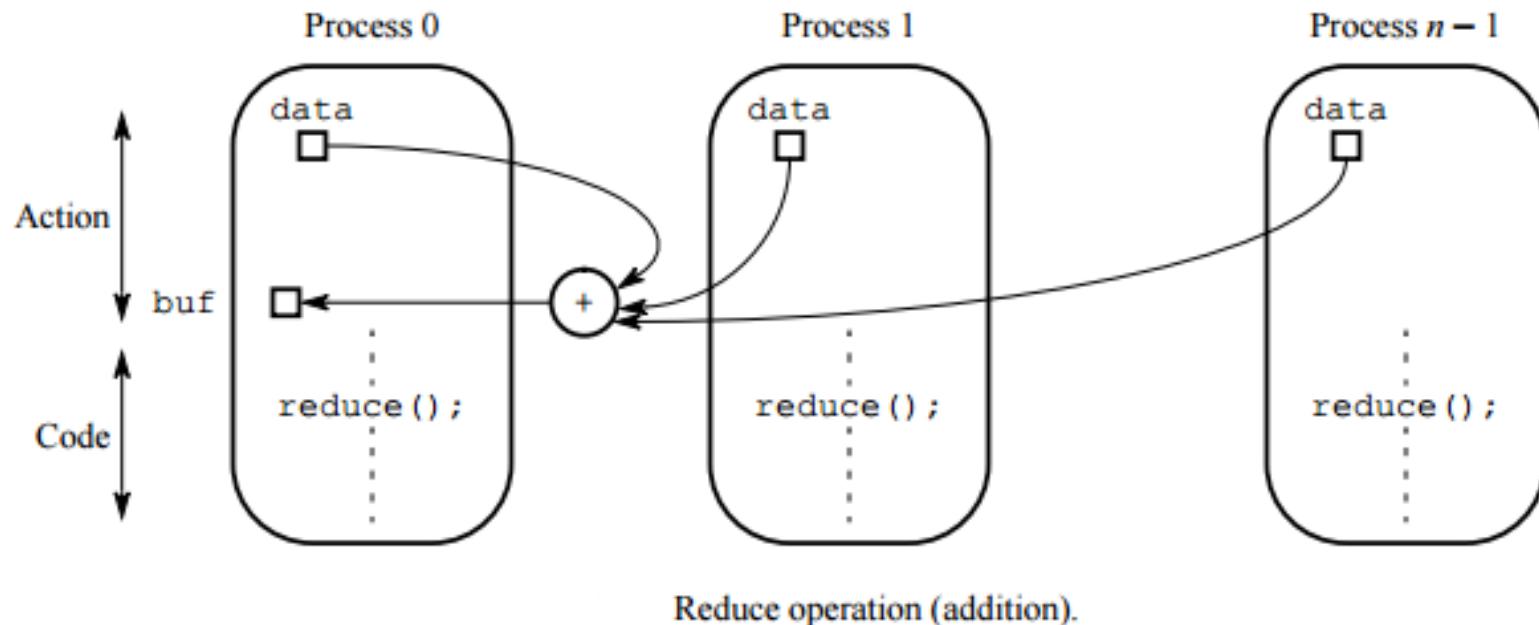
# Group communication routines



# Group communication routines



# Group communication routines



# Example

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000

void main(int argc, char *argv)
{
    int myid, numprocs;
    int data[MAXSIZE], i, x, low, high, myresult, result;
    char fn[255];
    char *fp;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    if (myid == 0) { /* Open input file and initialize data */
        strcpy(fn,getenv("HOME"));
        strcat(fn,"/MPI/rand_data.txt");
        if ((fp = fopen(fn,"r")) == NULL) {
            printf("Can't open the input file: %s\n\n", fn);
            exit(1);
        }
        for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d", &data[i]);
    }

    /* broadcast data */
    MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);
```

# Example

```
/* Add my portion Of data */
x = n/nproc;
low = myid * x;
high = low + x;
for(i = low; i < high; i++)
    myresult += data[i];
printf("I got %d from %d\n", myresult, myid);

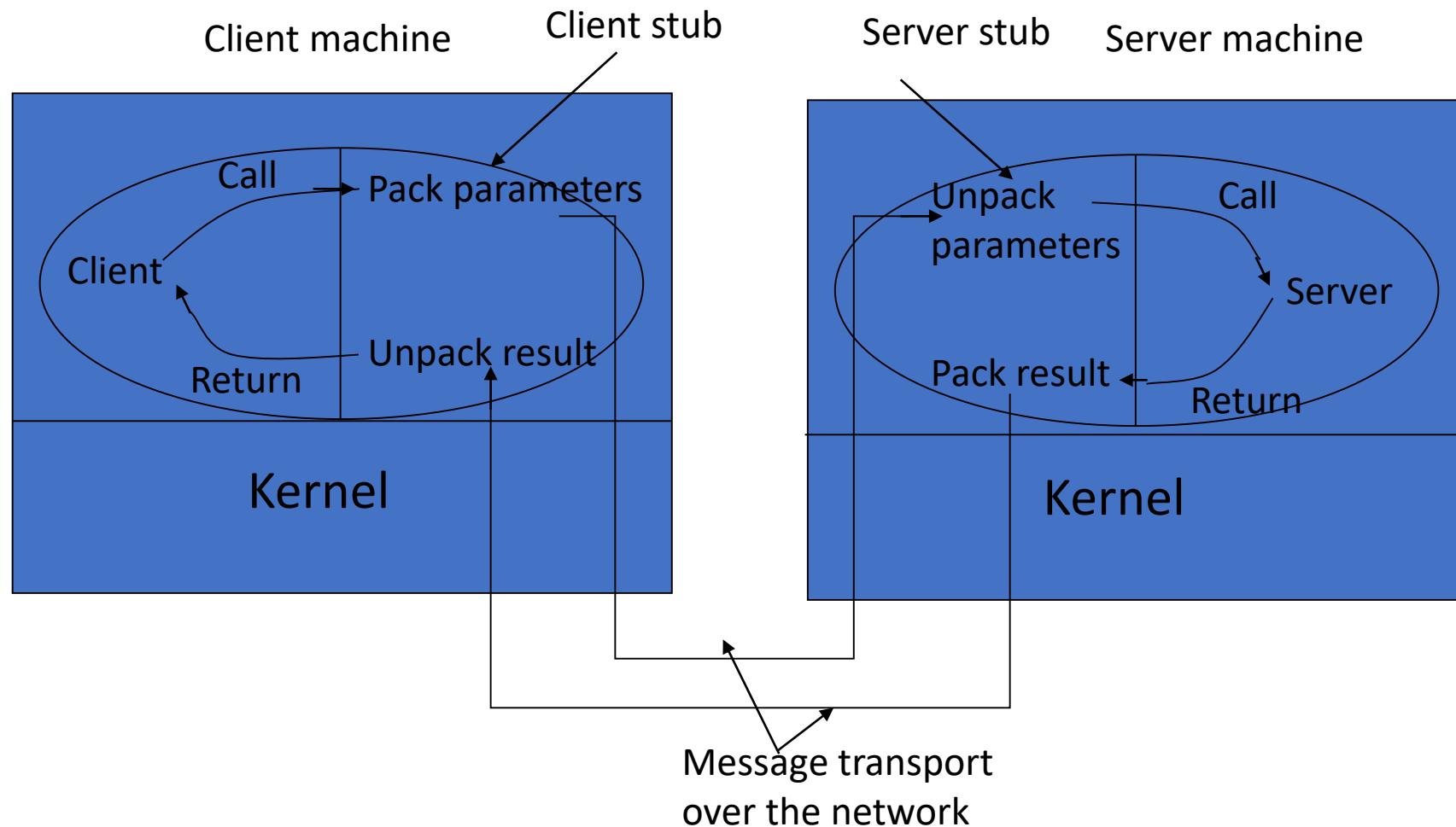
/* Compute global sum */
MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0) printf("The sum is %d.\n", result);

MPI_Finalize();
}
```

# Remote Procedure Call

- The idea behind RPC is to make a remote procedure call look as much as possible like a local one.
- A remote procedure call occurs in the following steps:
  - The client procedure calls the client stub in the normal way.
  - The client stub builds a message and traps to the kernel.
  - The kernel sends the message to the remote kernel.
  - The remote kernel gives the message to the server stub.
  - The server stub unpacks the parameters and calls the server.
  - The server does the work and returns the result to the stub.
  - The server stub packs it in a message and traps to the kernel.
  - The remote kernel sends the message to the client's kernel.
  - The client's kernel gives the message to the client stub.
  - The stub unpacks the result and returns to the client.

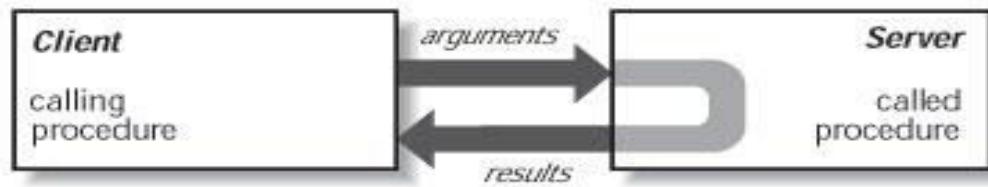
# Remote Procedure Call



# Remote Procedure Call

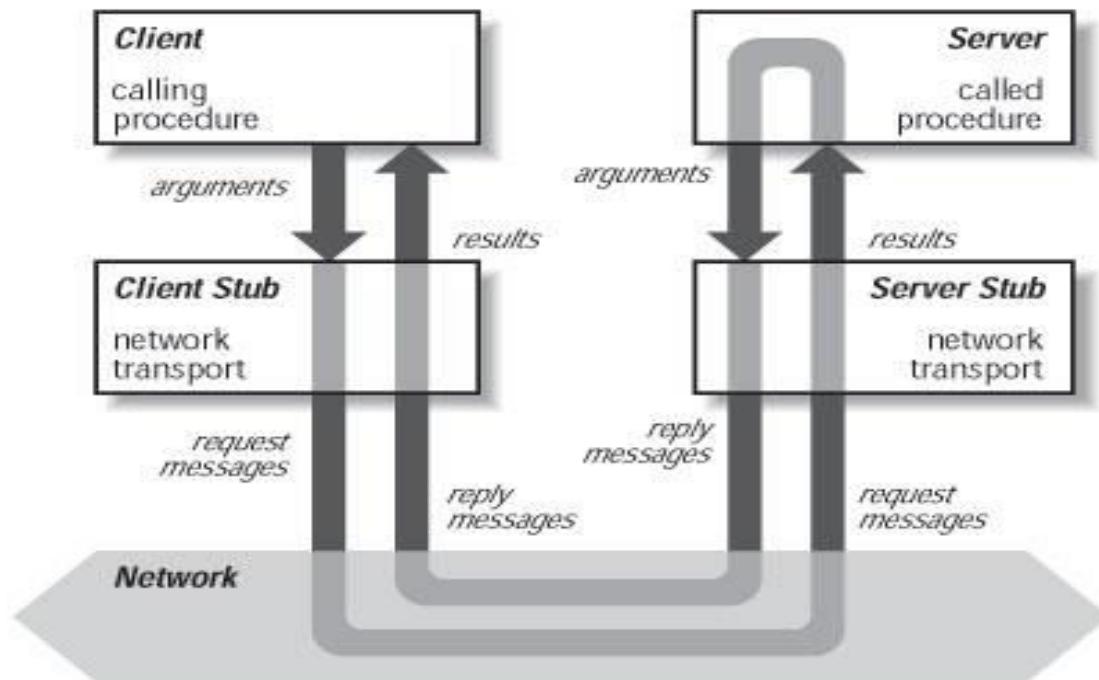
- An extension of conventional procedure call (used for transfer of control and data within a single process)
- allows a client application to call procedures in a different address space in the same or remote machine
- ideal for the client-server modeled applications
- primary goal is to make distributed programming easy, which is achieved by making the semantics of RPC as close as possible to conventional local procedure call
  - what is the semantics of local procedure call?

# Local vs. Remote Procedure Calls



In a local procedure call, a calling process executes a procedure in its own address space.

**Local Procedure Call**

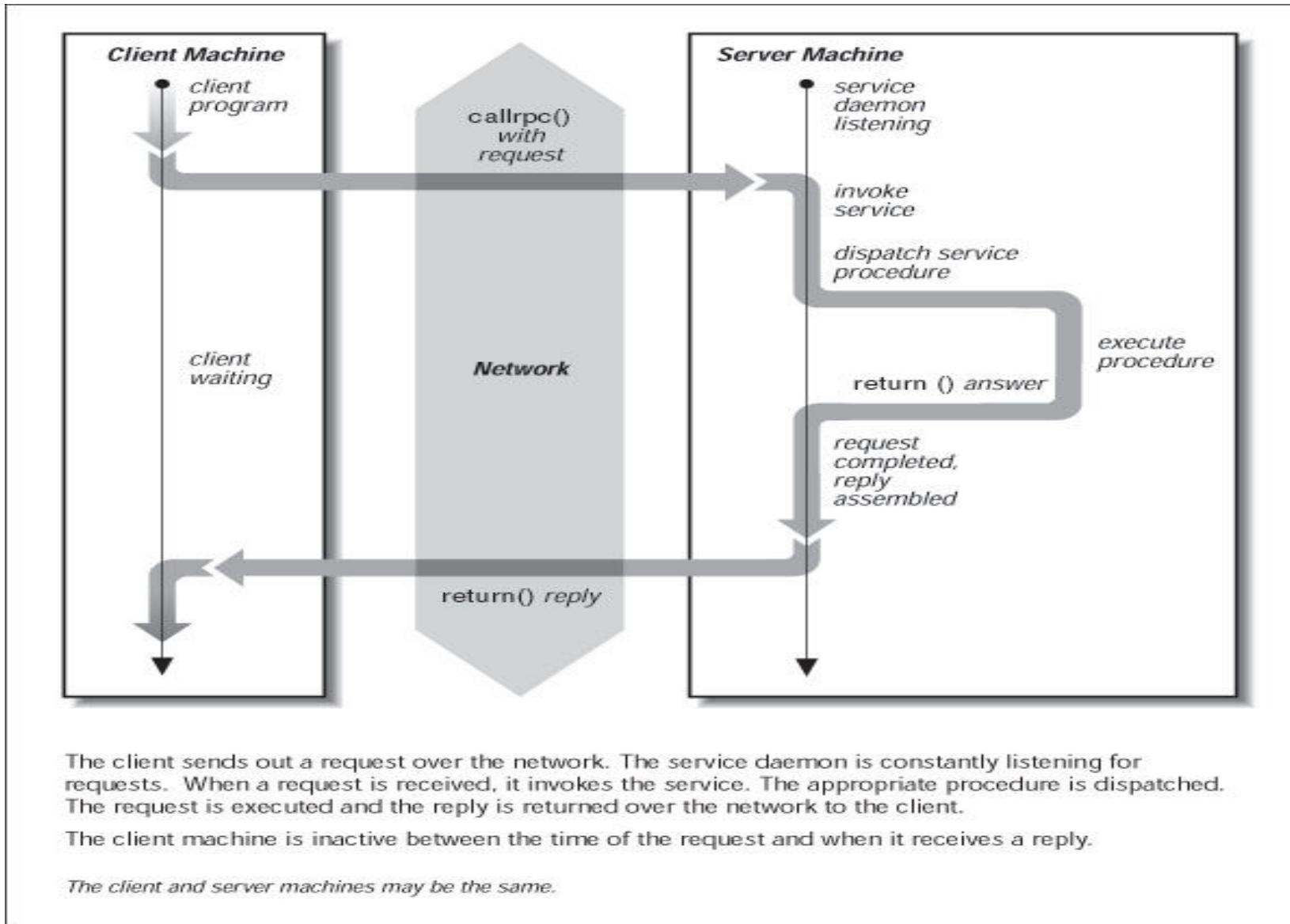


In a remote procedure call, the client and server run as two separate processes. It is not necessary for them to run on the same machine.

The two processes communicate through stubs, one each for the client and server. These stubs are pieces of code that contain functions to map local procedure calls into a series of network RPC function calls.

**Remote Procedure Call**

# RPC Communication



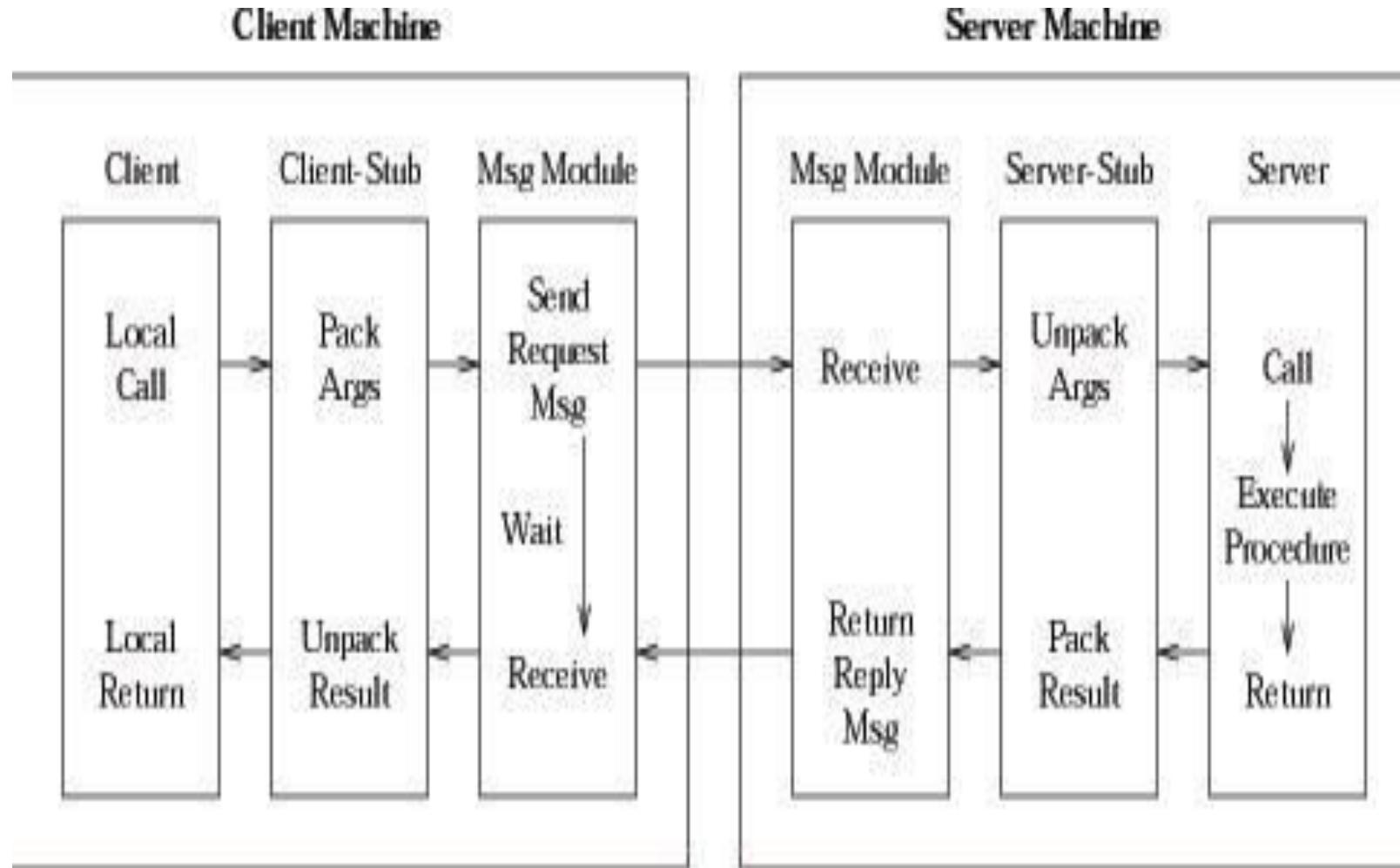
# RPC System Components

- **Message module**
  - IPC module of Send/Receive/Reply
  - responsible for exchanging messages
- **Stub procedures** (client and server stubs)
  - a stub is a communications interface that implements the RPC protocol and specifies how messages are constructed and exchanged
  - responsible for packing and unpacking of arguments and results (this is also referred to as “marshaling”)
  - these procedures are automatically generated by “stub generators” or “protocol compilers”

# RPC System Components

- **Client stub**
  - packs the arguments with the procedure name or ID into a message
  - sends the msg to the server and then awaits a reply msg
  - unpacks the results and returns them to the client
- **Server stub**
  - receives a request msg
  - unpacks the arguments and calls the appropriate server procedure
  - when it returns, packs the result and sends a reply msg back to the client

# RPC System Components and Call Flows



# Parameter Passing

- **little endian:** bytes are numbered from right to left

0 3	0 2	0 1	5 0
L 7	L 6	I 5	J 4

- **big endian:** bytes are numbered from left to right

5 0	0 1	0 2	0 3
J 4	I 5	L 6	L 7

# How to let two kinds of machines talk to each other?

- a standard should be agreed upon for representing each of the basic data types, given a parameter list (n parameters) and a message.
- devise a network standard or canonical form for integers, characters, Booleans, floating-point numbers, and so on.
- Convert to either little endian/big endian. But inefficient.
- use native format and indicate in the first byte of the message which format this is.

# How are pointers passed?

- not to use pointers. Highly undesirable.
- copy the array into the message and send it to the server. When the server finishes, the array can be copied back to the client.
- distinguish input array or output array. If input, no need to be copied back. If output, no need to be sent over to the server.
- still cannot handle the most general case of a pointer to an arbitrary data structure such as a complex graph.

# How can a client locate the server?

- hardwire the server network address into the client.

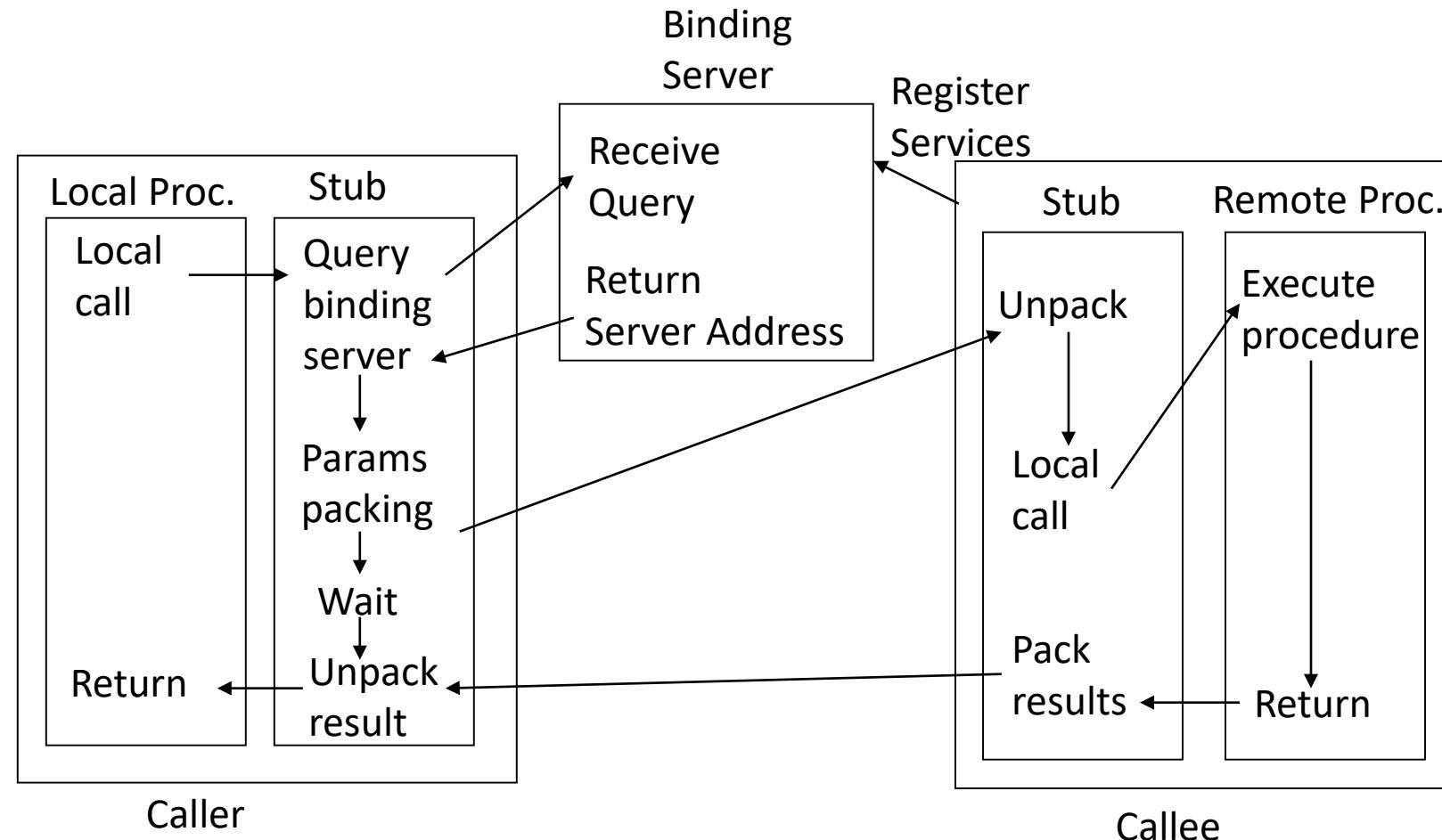
Disadvantage: inflexible.

- use **dynamic binding** to match up clients and servers.

# Dynamic Binding

- Server: **exports** the server interface.
- The server **registers** with a **binder** (a program), that is, give the binder *its name, its version number, a unique identifier, and a handle*.
- The server can also deregister when it is no longer prepared to offer service.

# Remote Procedure Call



# How the client locates the server?

- When the client calls one of the remote procedure “read” for the first time, the client stub sees that it is not yet bound to a server.
- The client stub sends message to the binder asking to **import** version 3.1 of the file-server interface.
- The binder checks to see if one or more servers have already **exported** an interface with this name and version number.
- If no server is willing to support this interface, the “read” call fails; else if a suitable server exists, the binder gives its handle and unique identifier to the client stub.
- The client stub uses the handle as the address to send the request message to.

# Advantages

- It can handle multiple servers that support the same interface
- The binder can spread the clients randomly over the servers to even the load
- It can also poll the servers periodically, automatically deregistering any server that fails to respond, to achieve a degree of fault tolerance
- It can also assist in authentication. Because a server could specify it only wished to be used by a specific list of users

# Disadvantages

- the extra overhead of exporting and importing interfaces cost time.

# Server Crashes

- The server can crash before the execution or after the execution
- The client cannot distinguish these two.
- The client can:
  - Wait until the server reboots and try the operation again (**at least once semantics**).
  - Zero or one execution can take place, if the remote procedure succeeds, exactly one computation has taken place otherwise none (**Exactly once semantics**)
  - Gives up immediately and reports back failure (**at most once semantics**).
  - Guarantee nothing.

# Client Crashes

- If a client sends a request to a server and crashes before the server replies, then a computation is active and no parent is waiting for the result. Such an unwanted computation is called an **orphan**.

# Problems with orphans

- They waste CPU cycles
- They can lock files or tie up valuable resources
- If the client reboots and does the RPC again, but the reply from the orphan comes back immediately afterward, confusion can result

# What to do with orphans?

- **Extermination:** Before a client stub sends an RPC message, it makes a log entry telling what it is about to do. After a reboot, the log is checked and the orphan is explicitly killed off.
  - Disadvantage: the expense of writing a disk record for every RPC; it may not even work, since orphans themselves may do RPCs, thus creating **grandorphans** or further descendants that are impossible to locate.
- **Reincarnation:** Divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message to all machines declaring the start of a new epoch. When such a broadcast comes in, all remote computations are killed.

# What to do with orphans?

- **Gentle reincarnation:** when an epoch broadcast comes in, each machine checks to see if it has any remote computations, and if so, tries to locate their owner. Only if the owner cannot be found is the computation killed.
- **Expiration:** Each RPC is given a standard amount of time,  $T$ , to do the job. If it cannot finish, it must explicitly ask for another quantum. On the other hand, if after a crash the server waits a time  $T$  before rebooting, all orphans are sure to be gone.
- None of the above methods are desirable.

# Implementation Issues

- the choice of the RPC protocol: connection-oriented or connectionless protocol?
- general-purpose protocol or specifically designed protocol for RPC?
- packet and message length
- Acknowledgements
- Flow control

**overrun error:** with some designs, a chip cannot accept two back-to-back packets because after receiving the first one, the chip is temporarily disabled during the packet-arrived interrupt, so it misses the start of the second one.

# How to deal with overrun error?

- If the problem is caused by the chip being disabled temporarily while it is processing an interrupt, a smart sender can insert a delay between packets to give the receiver just enough time.
- If the problem is caused by the finite buffer capacity of the network chip, say  $n$  packets, the sender can send  $n$  packets, followed by a substantial gap.

# Chapter 2: A Model of Distributed Computations

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

## A Distributed Program

- A distributed program is composed of a set of  $n$  asynchronous processes,  $p_1, p_2, \dots, p_i, \dots, p_n$ .
- The processes do not share a global memory and communicate solely by passing messages.
- The processes do not share a global clock that is instantaneously accessible to these processes.
- Process execution and message transfer are asynchronous.
- Without loss of generality, we assume that each process is running on a different processor.
- Let  $C_{ij}$  denote the channel from process  $p_i$  to process  $p_j$  and let  $m_{ij}$  denote a message sent by  $p_i$  to  $p_j$ .
- The message transmission delay is finite and unpredictable.

## A Model of Distributed Executions

- The execution of a process consists of a sequential execution of its actions.
- The actions are atomic and the actions of a process are modeled as three types of events, namely, internal events, message send events, and message receive events.
- Let  $e_i^x$  denote the  $x$ th event at process  $p_i$ .
- For a message  $m$ , let  $send(m)$  and  $rec(m)$  denote its send and receive events, respectively.
- The occurrence of events changes the states of respective processes and channels.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that receives the message and the state of the channel on which the message is received.

## A Model of Distributed Executions

- The events at a process are linearly ordered by their order of occurrence.
- The execution of process  $p_i$  produces a sequence of events  $e_i^1, e_i^2, \dots, e_i^x, e_i^{x+1}, \dots$  and is denoted by  $\mathcal{H}_i$  where

$$\mathcal{H}_i = (h_i, \rightarrow_i)$$

$h_i$  is the set of events produced by  $p_i$  and  
binary relation  $\rightarrow_i$  defines a linear order on these events.

- Relation  $\rightarrow_i$  expresses causal dependencies among the events of  $p_i$ .

## A Model of Distributed Executions

- The send and the receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process.
- A relation  $\rightarrow_{msg}$  that captures the causal dependency due to message exchange, is defined as follows. For every message  $m$  that is exchanged between two processes, we have

$$send(m) \rightarrow_{msg} rec(m).$$

- Relation  $\rightarrow_{msg}$  defines causal dependencies between the pairs of corresponding send and receive events.

## A Model of Distributed Executions

- The evolution of a distributed execution is depicted by a space-time diagram.
- A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer.
- Since we assume that an event execution is atomic (hence, indivisible and instantaneous), it is justified to denote it as a dot on a process line.
- In the Figure 2.1, for process  $p_1$ , the second event is a message send event, the third event is an internal event, and the fourth event is a message receive event.

## A Model of Distributed Executions

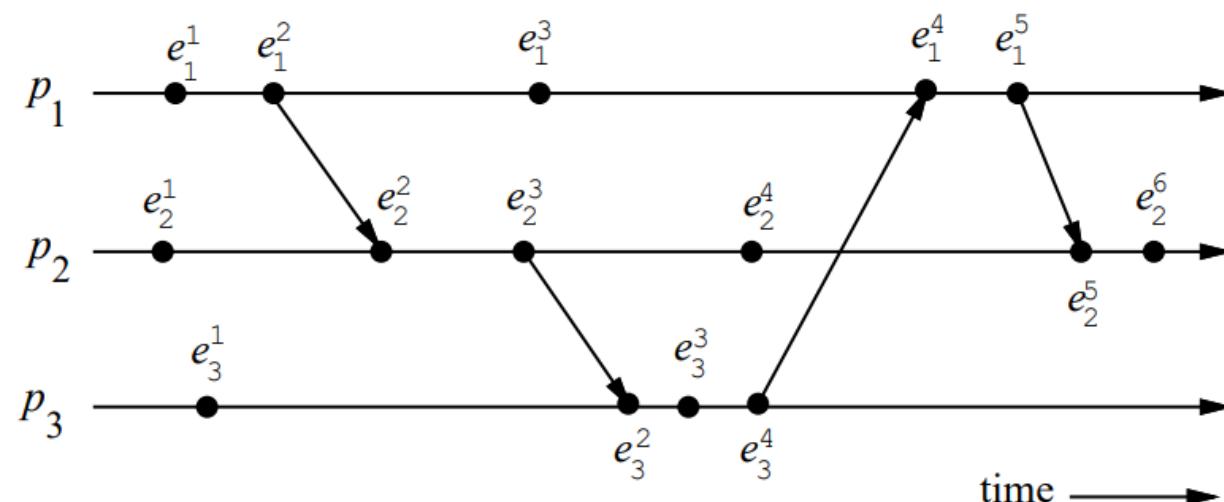


Figure 2.1: The space-time diagram of a distributed execution.

# A Model of Distributed Executions

## Causal Precedence Relation

- The execution of a distributed application results in a set of distributed events produced by the processes.
- Let  $H = \cup_i h_i$  denote the set of events executed in a distributed computation.
- Define a binary relation  $\rightarrow$  on the set  $H$  as follows that expresses causal dependencies between events in the distributed execution.

$$\forall e_i^x, \forall e_j^y \in H, e_i^x \rightarrow e_j^y \Leftrightarrow \begin{cases} e_i^x \rightarrow_i e_j^y \text{ i.e., } (i = j) \wedge (x < y) \\ \text{or} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{or} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \end{cases}$$

- The causal precedence relation induces an irreflexive partial order on the events of a distributed computation that is denoted as  $\mathcal{H} = (H, \rightarrow)$ .

# A Model of Distributed Executions

## ... Causal Precedence Relation

- Note that the relation  $\rightarrow$  is nothing but Lamport's "happens before" relation.
- For any two events  $e_i$  and  $e_j$ , if  $e_i \rightarrow e_j$ , then event  $e_j$  is directly or transitively dependent on event  $e_i$ . (Graphically, it means that there exists a path consisting of message arrows and process-line segments (along increasing time) in the space-time diagram that starts at  $e_i$  and ends at  $e_j$ .)
- For example, in Figure 2.1,  $e_1^1 \rightarrow e_3^3$  and  $e_3^3 \rightarrow e_2^6$ .
- The relation  $\rightarrow$  denotes flow of information in a distributed computation and  $e_i \rightarrow e_j$  dictates that all the information available at  $e_i$  is potentially accessible at  $e_j$ .
- For example, in Figure 2.1, event  $e_2^6$  has the knowledge of all other events shown in the figure.

## A Model of Distributed Executions

### ... Causal Precedence Relation

- For any two events  $e_i$  and  $e_j$ ,  $e_i \not\rightarrow e_j$  denotes the fact that event  $e_j$  does not directly or transitively depend on event  $e_i$ . That is, event  $e_i$  does not causally affect event  $e_j$ .
- In this case, event  $e_j$  is not aware of the execution of  $e_i$  or any event executed after  $e_i$  on the same process.
- For example, in Figure 2.1,  $e_1^3 \not\rightarrow e_3^3$  and  $e_2^4 \not\rightarrow e_3^1$ .

Note the following two rules:

- For any two events  $e_i$  and  $e_j$ ,  $e_i \not\rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$ .
- For any two events  $e_i$  and  $e_j$ ,  $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$ .

# A Model of Distributed Executions

## Concurrent events

- For any two events  $e_i$  and  $e_j$ , if  $e_i \not\rightarrow e_j$  and  $e_j \not\rightarrow e_i$ , then events  $e_i$  and  $e_j$  are said to be concurrent (denoted as  $e_i \parallel e_j$ ).
- In the execution of Figure 2.1,  $e_1^3 \parallel e_3^3$  and  $e_2^4 \parallel e_3^1$ .
- The relation  $\parallel$  is not transitive; that is,  $(e_i \parallel e_j) \wedge (e_j \parallel e_k) \not\Rightarrow e_i \parallel e_k$ .
- For example, in Figure 2.1,  $e_3^3 \parallel e_2^4$  and  $e_2^4 \parallel e_1^5$ , however,  $e_3^3 \not\parallel e_1^5$ .
- For any two events  $e_i$  and  $e_j$  in a distributed execution,  
 $e_i \rightarrow e_j$  or  $e_j \rightarrow e_i$ , or  $e_i \parallel e_j$ .

# A Model of Distributed Executions

## Logical vs. Physical Concurrency

- In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other.
- Physical concurrency, on the other hand, has a connotation that the events occur at the same instant in physical time.
- Two or more events may be logically concurrent even though they do not occur at the same instant in physical time.
- However, if processor speed and message delays would have been different, the execution of these events could have very well coincided in physical time.
- Whether a set of logically concurrent events coincide in the physical time or not, does not change the outcome of the computation.
- Therefore, even though a set of logically concurrent events may not have occurred at the same instant in physical time, we can assume that these events occurred at the same instant in physical time.

## Models of Communication Networks

- There are several models of the service provided by communication networks, namely, FIFO, Non-FIFO, and causal ordering.
- In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.

## Models of Communication Networks

- The “causal ordering” model is based on Lamport’s “happens before” relation.
- A system that supports the causal ordering model satisfies the following property:

CO: For any two messages  $m_{ij}$  and  $m_{kj}$ , if  $\text{send}(m_{ij}) \rightarrow \text{send}(m_{kj})$ , then  $\text{rec}(m_{ij}) \rightarrow \text{rec}(m_{kj})$ .

- This property ensures that causally related messages destined to the same destination are delivered in an order that is consistent with their causality relation.
- Causally ordered delivery of messages implies FIFO message delivery. (Note that CO  $\subset$  FIFO  $\subset$  Non-FIFO.)
- Causal ordering model considerably simplifies the design of distributed algorithms because it provides a built-in synchronization.

## Global State of a Distributed System

"A collection of the local states of its components, namely, the processes and the communication channels."

- The state of a process is defined by the contents of processor registers, stacks, local memory, etc. and depends on the local context of the distributed application.
- The state of channel is given by the set of messages in transit in the channel.
- The occurrence of events changes the states of respective processes and channels.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that receives the message and the state of the channel on which the message is received.

## ... Global State of a Distributed System

### Notations

- $LS_i^x$  denotes the state of process  $p_i$  after the occurrence of event  $e_i^x$  and before the event  $e_i^{x+1}$ .
- $LS_i^0$  denotes the initial state of process  $p_i$ .
- $LS_i^x$  is a result of the execution of all the events executed by process  $p_i$  till  $e_i^x$ .
- Let  $send(m) \leq LS_i^x$  denote the fact that  $\exists y: 1 \leq y \leq x :: e_i^y = send(m)$ .
- Let  $rec(m) \not\leq LS_i^x$  denote the fact that  $\forall y: 1 \leq y \leq x :: e_i^y \neq rec(m)$ .

## ... Global State of a Distributed System

### A Channel State

- The state of a channel depends upon the states of the processes it connects.
- Let  $SC_{ij}^{x,y}$  denote the state of a channel  $C_{ij}$ .

The state of a channel is defined as follows:

$$SC_{ij}^{x,y} = \{m_{ij} \mid send(m_{ij}) \leq e_i^x \wedge rec(m_{ij}) \not\leq e_j^y\}$$

Thus, channel state  $SC_{ij}^{x,y}$  denotes all messages that  $p_i$  sent upto event  $e_i^x$  and which process  $p_j$  had not received until event  $e_j^y$ .

## ... Global State of a Distributed System

### Global State

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state  $GS$  is defined as,

$$GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j, z_k}\}$$

- For a global state to be meaningful, the states of all the components of the distributed system must be recorded at the same instant.
- This will be possible if the local clocks at processes were perfectly synchronized or if there were a global system clock that can be instantaneously read by the processes. (However, both are impossible.)

## ... Global State of a Distributed System

### A Consistent Global State

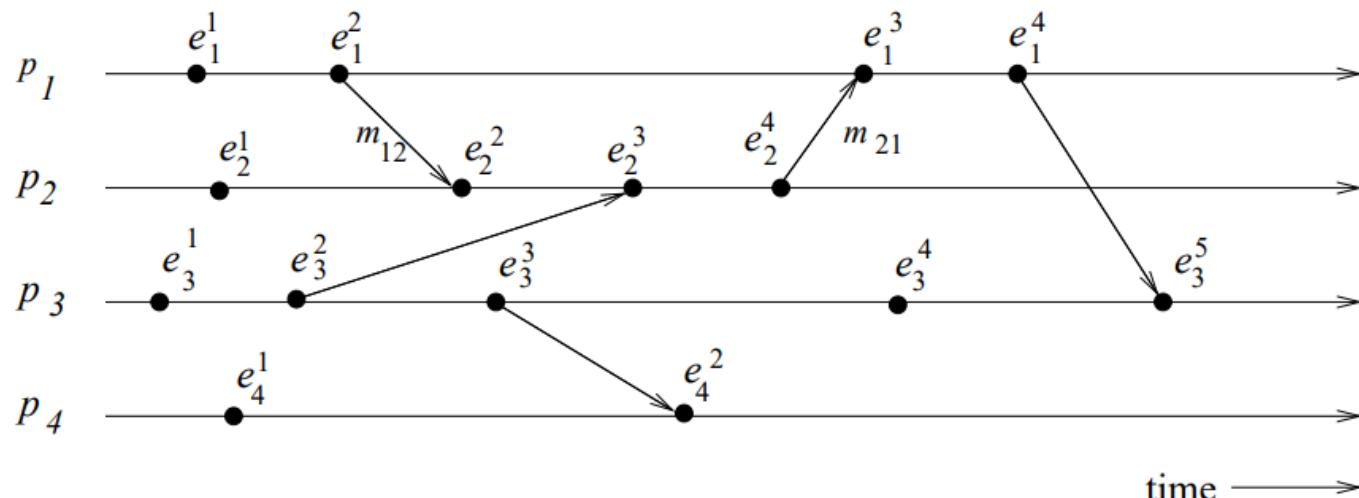
- Even if the state of all the components is not recorded at the same instant, such a state will be meaningful provided every message that is recorded as received is also recorded as sent.
- Basic idea is that a state should not violate causality – an effect should not be present without its cause. A message cannot be received if it was not sent.
- Such states are called *consistent global states* and are meaningful global states.
- Inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.
- A global state  $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j, z_k}\}$  is a *consistent global state* iff
$$\forall m_{ij} : send(m_{ij}) \not\leq LS_i^{x_i} \Leftrightarrow m_{ij} \notin SC_{ij}^{x_i, y_j} \wedge rec(m_{ij}) \not\leq LS_j^{y_j}$$
- That is, channel state  $SC_{ij}^{y_j, z_k}$  and process state  $LS_j^{z_k}$  must not include any message that process  $p_i$  sent after executing event  $e_i^{x_i}$ .

## ... Global State of a Distributed System

### An Example

Consider the distributed execution of Figure 2.2.

Figure 2.2: The space-time diagram of a distributed execution.



## ... Global State of a Distributed System

In Figure 2.2:

- A global state  $GS_1 = \{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$  is inconsistent because the state of  $p_2$  has recorded the receipt of message  $m_{12}$ , however, the state of  $p_1$  has not recorded its send.
- A global state  $GS_2$  consisting of local states  $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$  is consistent; all the channels are empty except  $C_{21}$  that contains message  $m_{21}$ .

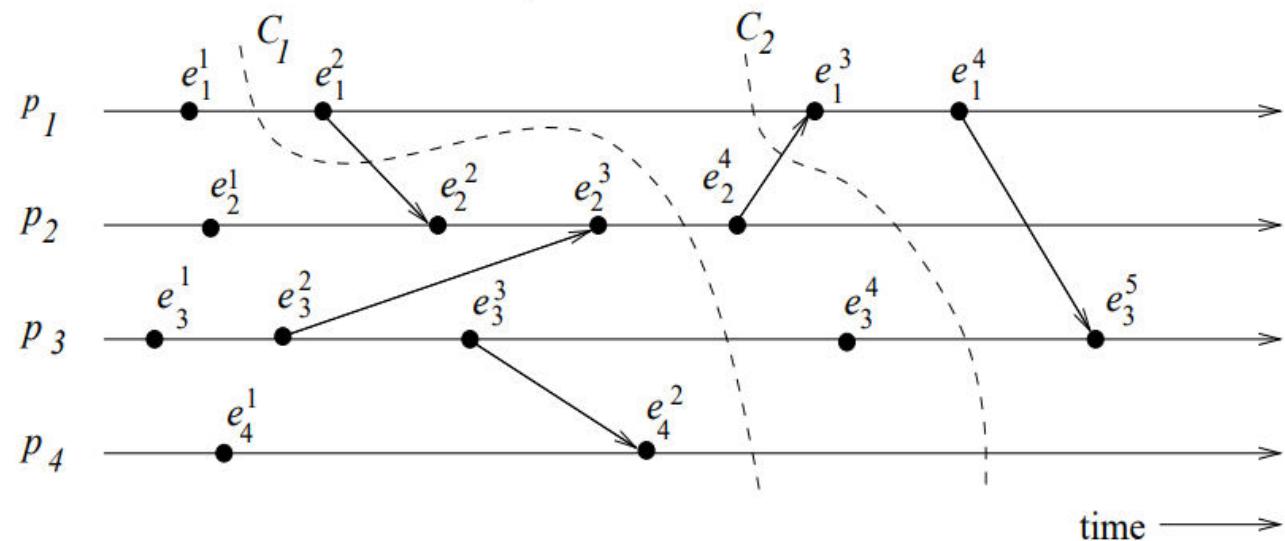
## Cuts of a Distributed Computation

"In the space-time diagram of a distributed computation, a *cut* is a zigzag line joining one arbitrary point on each process line."

- A cut slices the space-time diagram, and thus the set of events in the distributed computation, into a PAST and a FUTURE.
- The PAST contains all the events to the left of the cut and the FUTURE contains all the events to the right of the cut.
- For a cut  $C$ , let  $PAST(C)$  and  $FUTURE(C)$  denote the set of events in the PAST and FUTURE of  $C$ , respectively.
- Every cut corresponds to a global state and every global state can be graphically represented as a cut in the computation's space-time diagram.
- Cuts in a space-time diagram provide a powerful graphical aid in representing and reasoning about global states of a computation.

## ... Cuts of a Distributed Computation

Figure 2.3: Illustration of cuts in a distributed execution.



## ... Cuts of a Distributed Computation

- In a consistent cut, every message received in the PAST of the cut was sent in the PAST of that cut. (In Figure 2.3, cut  $C_2$  is a consistent cut.)
- All messages that cross the cut from the PAST to the FUTURE are in transit in the corresponding consistent global state.
- A cut is *inconsistent* if a message crosses the cut from the FUTURE to the PAST. (In Figure 2.3, cut  $C_1$  is an inconsistent cut.)

# Chapter 3: Logical Time

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

# Logical clock

1. A logical clock is a mechanism for capturing chronological and causal relationships in a distributed system. Distributed systems may have no physically synchronous global clock, so **a logical clock allows global ordering on events from different processes in distributed systems.**
2. It refers to implementing a protocol on all machines so that the **M/Cs are able to maintain a consistent ordering of events within some virtual time span.**
3. Logical clocks are useful in computation analysis, distributed algorithm design, individual event tracking, and exploring computational progress
4. The logical time in distributed systems is used to maintain the consistent ordering of events. The concept of causality, i.e. the causal precedence relationship, is fundamental for distributed systems.

# Logical time in Distributed Systems

**The logical time in distributed systems is used to maintain the consistent ordering of events. (causality- happens before relationship, assigning time stamps to events)**

**It is used to determine the order of events in a distributed computer system**

The concept of causality, i.e. the causal precedence relationship, is fundamental for distributed systems. Usually, it is tracked using physical time, but physical clocks are hard to maintain in a distributed system, so logical clocks are used instead.

The idea that makes logical clocks different is that these are designed to maintain the information about the order of events rather than pertaining to the same notion of time as the physical clocks.

- The concept of causality between events is fundamental to the design and analysis of parallel and distributed computing and operating systems.
- Usually causality is tracked using physical time.
- In distributed systems, it is not possible to have a global physical time.
- As asynchronous distributed computations make progress in spurts, the logical time is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems.

## Introduction

- This chapter discusses three ways to implement logical time - scalar time, vector time, and matrix time.
- Causality among events in a distributed system is a powerful concept in reasoning, analyzing, and drawing inferences about a computation.
- The knowledge of the causal precedence relation among the events of processes helps solve a variety of problems in distributed systems, such as distributed algorithms design, tracking of dependent events, knowledge about the progress of a computation, and concurrency measures.

# A Framework for a System of Logical Clocks

## Definition

- A system of logical clocks consists of a time domain  $T$  and a logical clock  $C$ . Elements of  $T$  form a partially ordered set over a relation  $<$ .
- Relation  $<$  is called the *happened before* or *causal precedence*. Intuitively, this relation is analogous to the *earlier than* relation provided by the physical time.
- The logical clock  $C$  is a function that maps an event  $e$  in a distributed system to an element in the time domain  $T$ , denoted as  $C(e)$  and called the timestamp of  $e$ , and is defined as follows:

$$C : H \mapsto T$$

such that the following property is satisfied:

$$\text{for two events } e_i \text{ and } e_j, e_i \rightarrow e_j \implies C(e_i) < C(e_j).$$

$C$  is a function which will take the events out of the distributed events which is denoted by  $H$  and basically maps on to a Time domain and the function will form a Time stamp i.e  $C(e)$

# A Framework for a System of Logical Clocks

- This monotonicity property is called the *clock consistency condition*.
- When  $T$  and  $C$  satisfy the following condition,

for two events  $e_i$  and  $e_j$ ,  $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$

the system of clocks is said to be *strongly consistent*.

## Implementing Logical Clocks

- Implementation of logical clocks requires addressing two issues: data structures local to every process to represent logical time and a protocol to update the data structures to ensure the consistency condition.
- Each process  $p_i$  maintains data structures that allow it the following two capabilities:
  - ▶ A *local logical clock*, denoted by  $lc_i$ , that helps process  $p_i$  measure its own progress.  
i.e ... ensures the progress of internal events

## Implementing Logical Clocks

- ▶ A *logical global clock*, denoted by  $gc_i$ , that is a representation of process  $p_i$ 's local view of the logical global time. Typically,  $lc_i$  is a part of  $gc_i$ .

The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently. The protocol consists of the following two rules:

- $R1$ : This rule governs how the local logical clock is updated by a process when it executes an event.
- $R2$ : This rule governs how a process updates its global logical clock to update its view of the global time and global progress.
- Systems of logical clocks differ in their representation of logical time and also in the protocol to update the logical clocks.

# Three ways to implement Logical Time

1. Scalar time
2. Vector time
3. Matrix time

Causality among events in a DS is a powerful concept in reasoning, analysing and drawing inferences about a computation.

# Scalar time

Scalar time is designed by Lamport to synchronize all the events in distributed systems.

A Lamport logical clock is an incrementing counter maintained in each process. This logical clock has meaning only in relation to messages moving between processes

Logical local time is used by the process to mark its own events, and logical global time is the local information about global time. A special protocol is used to update logical local time after each local event, and logical global time when processes exchange data.

# Implementing Lamport clocks

Lamport clocks tag events in a distributed system and order them accordingly. We seek a clock time  $C(a)$  for every event  $a$ . The clock condition is defined as follows:

If  $a \rightarrow b$ , then  $C(a) < C(b)$ .

Each process maintains an event counter. This event counter is the local Lamport clock.

# Lamport clock algorithm

Before the execution of an event, the local clock is updated. This can be explained by the equation  $C_i = C_i + 1$ , where  $i$  is the process identifier.

When a message is sent to another process, the message contains the process' local clock,  $C_m$ .

When a process receives a message  $m$ , it sets its local clock to  $1 + \max(C_i, C_m)$ .

## Scalar Time

- Proposed by Lamport in 1978 as an attempt to totally order events in a distributed system.
- Time domain is the set of non-negative integers.
- The logical local clock of a process  $p_i$  and its local view of the global time are squashed into one integer variable  $C_i$ .
- Rules  $R1$  and  $R2$  to update the clocks are as follows:
- $R1$ : Before executing an event (send, receive, or internal), process  $p_i$  executes the following:

$$C_i := C_i + d \quad (d > 0)$$

In general, every time  $R1$  is executed,  $d$  can have a different value; however, typically  $d$  is kept at 1.

## Scalar Time

- $R2$ : Each message piggybacks the clock value of its sender at sending time. When a process  $p_i$  receives a message with timestamp  $C_{msg}$ , it executes the following actions:
  - ▶  $C_i := \max(C_i, C_{msg})$
  - ▶ Execute  $R1$ .
  - ▶ Deliver the message.
- Figure 3.1 shows evolution of scalar time.

# Scalar Time

**Evolution of scalar time:**

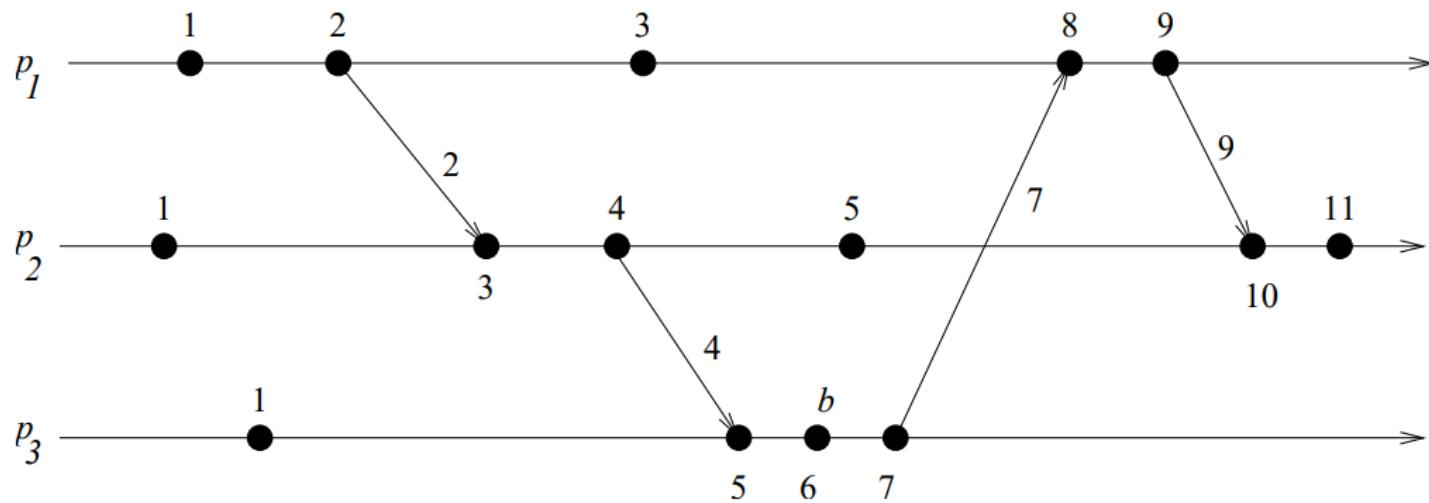


Figure 3.1: The space-time diagram of a distributed execution.

# Basic Properties

## Consistency Property

- Scalar clocks satisfy the monotonicity and hence the consistency property:  
for two events  $e_i$  and  $e_j$ ,  $e_i \rightarrow e_j \implies C(e_i) < C(e_j)$ .

## Total Ordering

- Scalar clocks can be used to totally order events in a distributed system.
- The main problem in totally ordering events is that two or more events at different processes may have identical timestamp.
- For example in Figure 3.1, the third event of process  $P_1$  and the second event of process  $P_2$  have identical scalar timestamp.

## Scalar Time

Evolution of scalar time:

How to order events ?

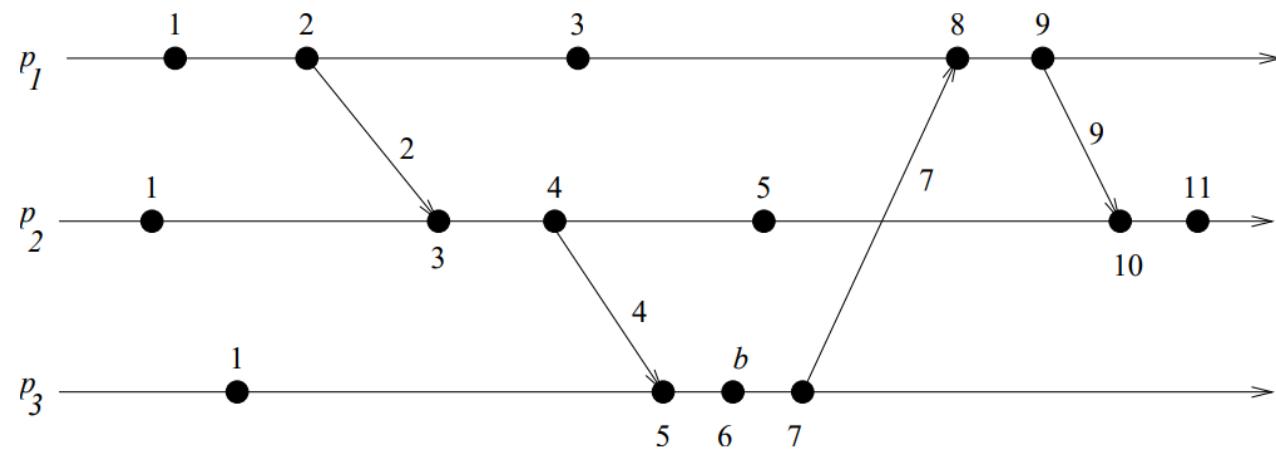


Figure 3.1: The space-time diagram of a distributed execution.

# Total Ordering

(ts x, idx) and (ts y, idy), mssg IDs are used to order the events

A tie-breaking mechanism is needed to order such events. A tie is broken as follows:

- Process identifiers are linearly ordered and tie among events with identical scalar timestamp is broken on the basis of their process identifiers.
- The lower the process identifier in the ranking, the higher the priority.
- The timestamp of an event is denoted by a tuple  $(t, i)$  where  $t$  is its time of occurrence and  $i$  is the identity of the process where it occurred.
- The total order relation  $\prec$  on two events  $x$  and  $y$  with timestamps  $(h,i)$  and  $(k,j)$ , respectively, is defined as follows:

$$x \prec y \Leftrightarrow (h < k \text{ or } (h = k \text{ and } i < j))$$

## Properties...

Scalar time for event counting

### Event counting

- If the increment value  $d$  is always 1, the scalar time has the following interesting property: if event  $e$  has a timestamp  $h$ , then  $h-1$  represents the minimum logical duration, counted in units of events, required before producing the event  $e$ ;
- We call it the height of the event  $e$ .
- In other words,  $h-1$  events have been produced sequentially before the event  $e$  regardless of the processes that produced these events.
- For example, in Figure 3.1, five events precede event  $b$  on the longest causal path ending at  $b$ .

## Scalar Time

Evolution of scalar time:

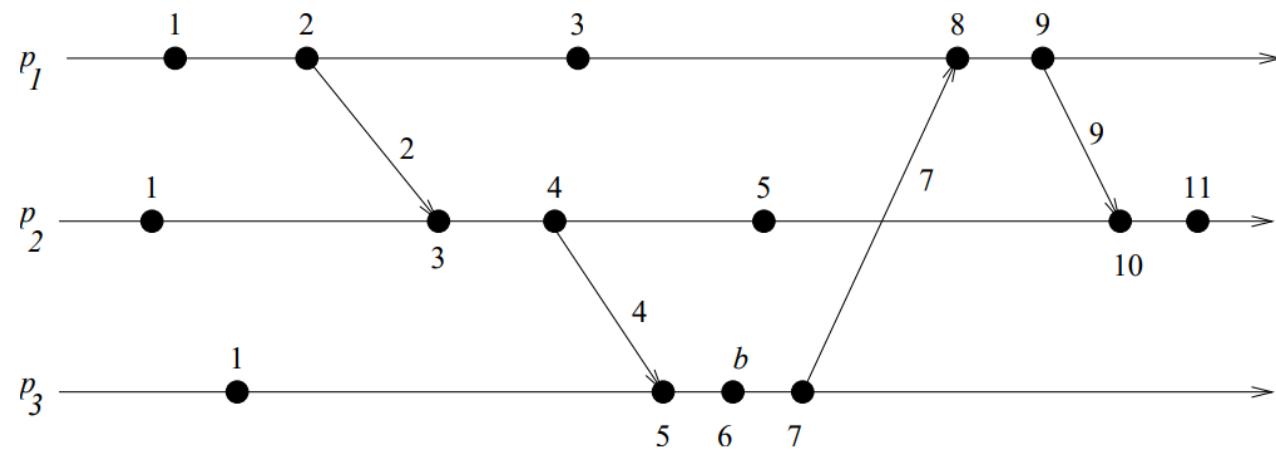


Figure 3.1: The space-time diagram of a distributed execution.

## Properties...

### No Strong Consistency

- The system of scalar clocks is not strongly consistent; that is, for two events  $e_i$  and  $e_j$ ,  $C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j$ .
- For example, in Figure 3.1, the third event of process  $P_1$  has smaller scalar timestamp than the third event of process  $P_2$ . However, the former did not happen before the latter.
- The reason that scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss causal dependency information among events at different processes.
- For example, in Figure 3.1, when process  $P_2$  receives the first message from process  $P_1$ , it updates its clock to 3, forgetting that the timestamp of the latest event at  $P_1$  on which it depends is 2.

## Scalar Time

Evolution of scalar time:

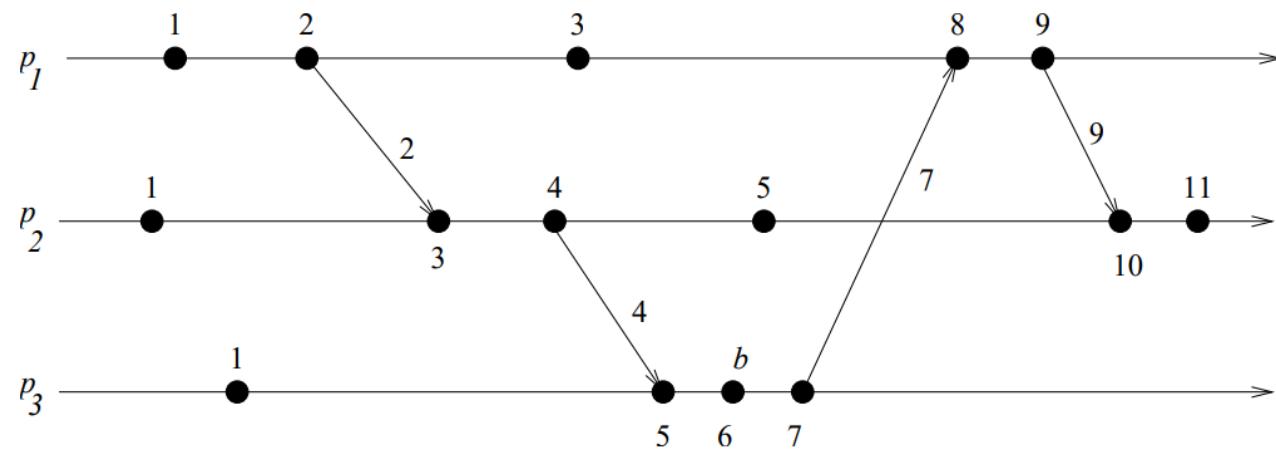


Figure 3.1: The space-time diagram of a distributed execution.

## Vector Time

- The system of vector clocks was developed independently by Fidge, Mattern and Schmuck.
- In the system of vector clocks, the time domain is represented by a set of  $n$ -dimensional non-negative integer vectors.
- Each process  $p_i$  maintains a vector  $vt_i[1..n]$ , where  $vt_i[i]$  is the local logical clock of  $p_i$  and describes the logical time progress at process  $p_i$ .
- $vt_i[j]$  represents process  $p_i$ 's latest knowledge of process  $p_j$  local time.
- If  $vt_i[j]=x$ , then process  $p_i$  knows that local time at process  $p_j$  has progressed till  $x$ .
- The entire vector  $vt_i$  constitutes  $p_i$ 's view of the global logical time and is used to timestamp events.

## Vector Time

Process  $p_i$  uses the following two rules  $R1$  and  $R2$  to update its clock:

- $R1$ : Before executing an event, process  $p_i$  updates its local logical time as follows:

$$vt_i[i] := vt_i[i] + d \quad (d > 0)$$

- $R2$ : Each message  $m$  is piggybacked with the vector clock  $vt$  of the sender process at sending time. On the receipt of such a message  $(m, vt)$ , process  $p_i$  executes the following sequence of actions:
  - ▶ Update its global logical time as follows:

$$1 \leq k \leq n : vt_i[k] := \max(vt_i[k], vt[k])$$

- ▶ Execute  $R1$ .
- ▶ Deliver the message  $m$ .

## Vector Time

- The timestamp of an event is the value of the vector clock of its process when the event is executed.
- Figure 3.2 shows an example of vector clocks progress with the increment value  $d=1$ .
- Initially, a vector clock is  $[0, 0, 0, \dots, 0]$ .

# Vector Time

## An Example of Vector Clocks

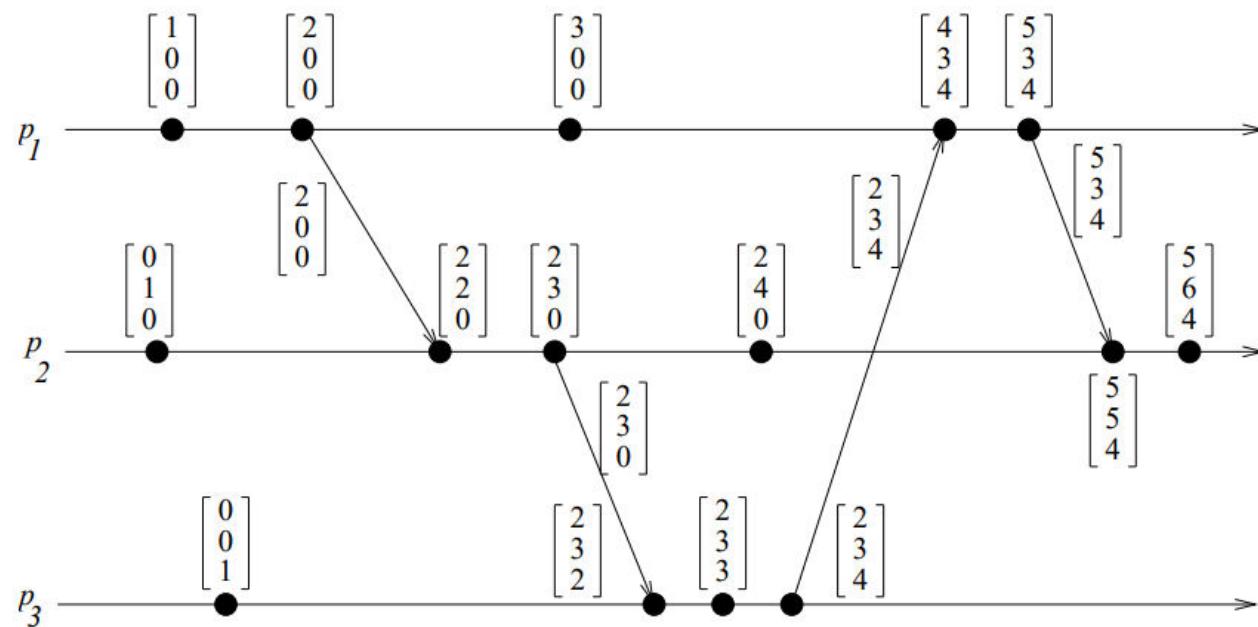


Figure 3.2: Evolution of vector time.

# Vector Time

## Comparing Vector Timestamps

- The following relations are defined to compare two vector timestamps,  $vh$  and  $vk$ :

$$vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$$

$$vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$$

$$vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$$

$$vh \parallel vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh)$$

- If the process at which an event occurred is known, the test to compare two timestamps can be simplified as follows: If events  $x$  and  $y$  respectively occurred at processes  $p_i$  and  $p_j$  and are assigned timestamps  $vh$  and  $vk$ , respectively, then

$$x \rightarrow y \Leftrightarrow vh[i] \leq vk[i]$$

$$x \parallel y \Leftrightarrow vh[i] > vk[i] \wedge vh[j] < vk[j]$$

# Vector Time

## Properties of Vector Time

### Isomorphism

- If events in a distributed system are timestamped using a system of vector clocks, we have the following property.  
If two events  $x$  and  $y$  have timestamps  $vh$  and  $vk$ , respectively, then

$$\begin{aligned}x \rightarrow y &\Leftrightarrow vh < vk \\x \parallel y &\Leftrightarrow vh \parallel vk.\end{aligned}$$

- Thus, there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps.

# Vector Time

## Strong Consistency

- The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related.
- However, Charron-Bost showed that the dimension of vector clocks cannot be less than  $n$ , the total number of processes in the distributed computation, for this property to hold.

## Event Counting

- If  $d=1$  (in rule  $R1$ ), then the  $i^{th}$  component of vector clock at process  $p_i$ ,  $vt_i[i]$ , denotes the number of events that have occurred at  $p_i$  until that instant.
- So, if an event  $e$  has timestamp  $vh$ ,  $vh[j]$  denotes the number of events executed by process  $p_j$  that causally precede  $e$ . Clearly,  $\sum vh[j] - 1$  represents the total number of events that causally precede  $e$  in the distributed computation.

## Efficient Implementations of Vector Clocks

- If the number of processes in a distributed computation is large, then vector clocks will require piggybacking of huge amount of information in messages.
- The message overhead grows linearly with the number of processors in the system and when there are thousands of processors in the system, the message size becomes huge even if there are only a few events occurring in few processors.
- We discuss an efficient way to maintain vector clocks.
- Charron-Bost showed that if vector clocks have to satisfy the strong consistency property, then in general vector timestamps must be at least of size  $n$ , the total number of processes.
- However, optimizations are possible and next, and we discuss a technique to implement vector clocks efficiently.

## Singhal-Kshemkalyani's Differential Technique

- *Singhal-Kshemkalyani's differential technique* is based on the observation that between successive message sends to the same process, only a few entries of the vector clock at the sender process are likely to change.
- When a process  $p_i$  sends a message to a process  $p_j$ , it piggybacks only those entries of its vector clock that differ since the last message sent to  $p_j$ .
- If entries  $i_1, i_2, \dots, i_{n_1}$  of the vector clock at  $p_i$  have changed to  $v_1, v_2, \dots, v_{n_1}$ , respectively, since the last message sent to  $p_j$ , then process  $p_i$  piggybacks a compressed timestamp of the form:

$$\{(i_1, v_1), (i_2, v_2), \dots, (i_{n_1}, v_{n_1})\}$$

to the next message to  $p_j$ .

## Singhal-Kshemkalyani's Differential Technique

When  $p_j$  receives this message, it updates its vector clock as follows:

$$vt_i[i_k] = \max(vt_i[i_k], v_k) \text{ for } k = 1, 2, \dots, n_1.$$

- Thus this technique cuts down the message size, communication bandwidth and buffer (to store messages) requirements.
- In the worst of case, every element of the vector clock has been updated at  $p_i$  since the last message to process  $p_j$ , and the next message from  $p_i$  to  $p_j$  will need to carry the entire vector timestamp of size  $n$ .
- However, on the average the size of the timestamp on a message will be less than  $n$ .

## Singhal-Kshemkalyani's Differential Technique

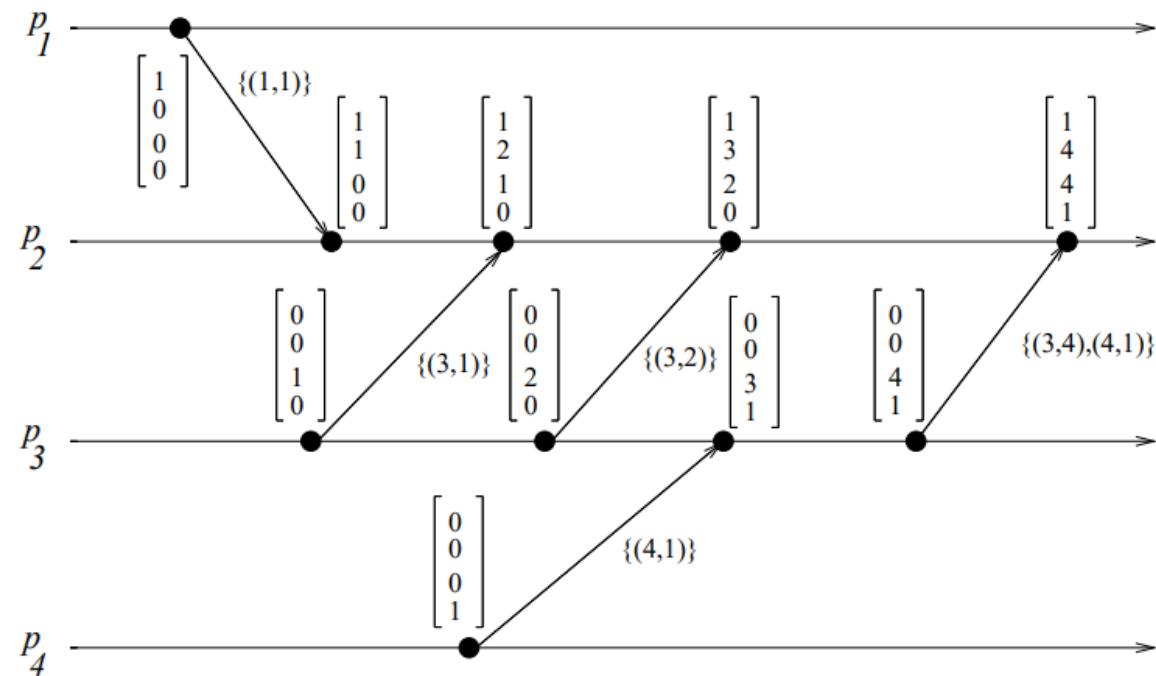


Figure 3.3: Vector clocks progress in Singhal-Kshemkalyani technique. 33

## Singhal-Kshemkalyani's Differential Technique

- Implementation of this technique requires each process to remember the vector timestamp in the message last sent to every other process.
- Direct implementation of this will result in  $O(n^2)$  storage overhead at each process.
- Singhal and Kshemkalyani developed a clever technique that cuts down this storage overhead at each process to  $O(n)$ . The technique works in the following manner:
- Process  $p_i$  maintains the following two additional vectors:
  - ▶  $LS_i[1..n]$  ('Last Sent'):  $LS_i[j]$  indicates the value of  $vt_i[i]$  when process  $p_i$  last sent a message to process  $p_j$ .
  - ▶  $LU_i[1..n]$  ('Last Update'):  $LU_i[j]$  indicates the value of  $vt_i[i]$  when process  $p_i$  last updated the entry  $vt_i[j]$ .
- Clearly,  $LU_i[i] = vt_i[i]$  at all times and  $LU_i[j]$  needs to be updated only when the receipt of a message causes  $p_i$  to update entry  $vt_i[j]$ . Also,  $LS_i[j]$  needs to be updated only when  $p_i$  sends a message to  $p_j$ .

## Singhal-Kshemkalyani's Differential Technique

- Since the last communication from  $p_i$  to  $p_j$ , only those elements of vector clock  $vt_i[k]$  have changed for which  $LS_i[j] < LU_i[k]$  holds.
- Hence, only these elements need to be sent in a message from  $p_i$  to  $p_j$ . When  $p_i$  sends a message to  $p_j$ , it sends only a set of tuples

$$\{(x, vt_i[x]) | LS_i[j] < LU_i[x]\}$$

as the vector timestamp to  $p_j$ , instead of sending a vector of  $n$  entries in a message.

- Thus the entire vector of size  $n$  is not sent along with a message. Instead, only the elements in the vector clock that have changed since the last message send to that process are sent in the format  $\{(p_1, \text{latest\_value}), (p_2, \text{latest\_value}), \dots\}$ , where  $p_i$  indicates that the  $p_i$ th component of the vector clock has changed.
- This technique requires that the communication channels follow FIFO discipline for message delivery.

## Singhal-Kshemkalyani's Differential Technique

- This method is illustrated in Figure 3.3. For instance, the second message from  $p_3$  to  $p_2$  (which contains a timestamp  $\{(3, 2)\}$ ) informs  $p_2$  that the third component of the vector clock has been modified and the new value is 2.
- This is because the process  $p_3$  (indicated by the third component of the vector) has advanced its clock value from 1 to 2 since the last message sent to  $p_2$ .
- This technique substantially reduces the cost of maintaining vector clocks in large systems, especially if the process interactions exhibit temporal or spatial localities.

## Fowler-Zwaenepoel's direct-dependency technique

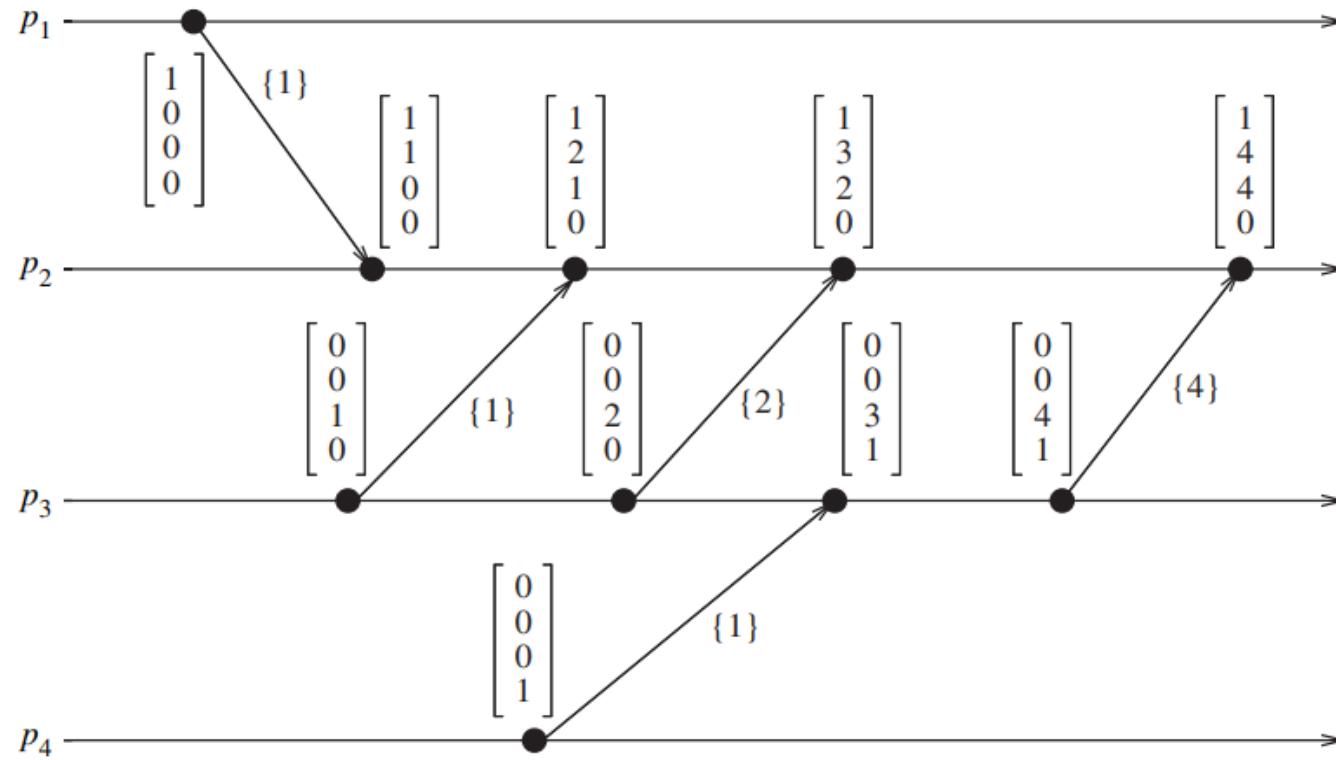
This technique further reduces the message size by only sending the single clock value of the sending process with a message. But it will be difficult to track the event dependencies.

Each process  $p_i$  maintains a dependency vector  $D_i$ . Initially,

$$D_i[j] = 0 \text{ for } j = 1, \dots, n.$$

$D_i$  is updated as follows:

1. Whenever an event occurs at  $p_i$ ,  $D_i[i] := D_i[i] + 1$ . That is, the vector component corresponding to its own local time is incremented by one.
2. When a process  $p_i$  sends a message to process  $p_j$ , it piggybacks the updated value of  $D_i[i]$  in the message.
3. When  $p_i$  receives a message from  $p_j$  with piggybacked value  $d$ ,  $p_i$  updates its dependency vector as follows:  $D_i[j] := \max\{D_i[j], d\}$ .



---

```

DependencyTrack( $i$  : process,  $\sigma$  : event index)
  /* Casual distributed breakpoint for  $\sigma_i$  */
  /* DTV holds the result */
  for all  $k \neq i$  do
     $DTV[k] = 0$ 
  end for
   $DTV[i] = \sigma$ 
  end DependencyTrack

VisitEvent( $j$  : process,  $e$  : event index)
  /* Place dependencies of  $\tau$  into DTV */
  for all  $k \neq j$  do
     $\alpha = D_j^e[k]$ 
    if  $\alpha > DTV[k]$  then
       $DTV[k] = \alpha$ 
      VisitEvent( $k, \alpha$ )
    end if
  end for
  end VisitEvent

```

---

To track the event dependencies, this approach uses the Dependency Track algorithm.

The Visit Event proposed as part of the algorithm works in a recursive manner to track the list of causally related events for a specific event running in a particular process.

**Algorithm 3.1** Recursive dependency trace algorithm

### **DependencyTrack(2 4):**

- DTV is initially set to < 0400 > by DependencyTrack.
- It then calls VisitEvent(2 4).
- The values held by  $D^4_2$  are < 144 0 >.
- So, DTV is now updated to < 1400 > and VisitEvent(1 1) is called.
- The values held by  $D^1_1$  are < 1000 >.
- Since none of the entries are greater than those in DTV, the algorithm returns.
- Again the values held by  $D^4_2$  are checked and this time entry 3 is found to be greater in  $D^4_2$  than DTV.
- So, DTV is updated as < 1440 > and VisitEvent(3 4) is called.
- The values held by  $D^4_3$  are < 0041 >.
- Since entry 4 of  $D^4_3$  is greater than that of DTV, it is updated as < 1441 > and VisitEvent(4 1) is called.
- Since none of the entries in  $D^1_4$ : < 0001 > are greater than those of DTV, the algorithm returns to VisitEvent(2 4).
- Since all the entries have been checked, VisitEvent(2 4) is exited and so is DependencyTrack.
- At this point, DTV holds < 1441 >, meaning event 4 of process p2 is dependent upon event 1 of process p1, event 4 of process p3 and event 1 in process p4.

## Matrix Time

In a system of matrix clocks, the time is represented by a set of  $n \times n$  matrices of non-negative integers.

A process  $p_i$  maintains a matrix  $mt_i[1..n, 1..n]$  where,

- $mt_i[i, i]$  denotes the local logical clock of  $p_i$  and tracks the progress of the computation at process  $p_i$ .
- $mt_i[i, j]$  denotes the latest knowledge that process  $p_i$  has about the local logical clock,  $mt_j[j, j]$ , of process  $p_j$ .
- $mt_i[j, k]$  represents the knowledge that process  $p_i$  has about the latest knowledge that  $p_j$  has about the local logical clock,  $mt_k[k, k]$ , of  $p_k$ .
- The entire matrix  $mt_i$  denotes  $p_i$ 's local view of the global logical time.

## Matrix Time

Process  $p_i$  uses the following rules  $R1$  and  $R2$  to update its clock:

- $R1$  : Before executing an event, process  $p_i$  updates its local logical time as follows:

$$mt_i[i, i] := mt_i[i, i] + d \quad (d > 0)$$

- $R2$ : Each message  $m$  is piggybacked with matrix time  $mt$ . When  $p_i$  receives such a message  $(m, mt)$  from a process  $p_j$ ,  $p_i$  executes the following sequence of actions:

- ▶ Update its global logical time as follows:

$$(a) \ 1 \leq k \leq n : mt_i[i, k] := \max(mt_i[i, k], mt[j, k])$$

(That is, update its row  $mt_i[i, *]$  with the  $p_j$ 's row in the received timestamp,  $mt$ .)

$$(b) \ 1 \leq k, l \leq n : mt_i[k, l] := \max(mt_i[k, l], mt[k, l])$$

- ▶ Execute  $R1$ .
  - ▶ Deliver message  $m$ .

## Matrix Time

- Figure 3.4 gives an example to illustrate how matrix clocks progress in a distributed computation. We assume  $d=1$ .
- Let us consider the following events:  $e$  which is the  $x_i$ -th event at process  $p_i$ ,  $e_k^1$  and  $e_k^2$  which are the  $x_k^1$ -th and  $x_k^2$ -th event at process  $p_k$ , and  $e_j^1$  and  $e_j^2$  which are the  $x_j^1$ -th and  $x_j^2$ -th events at  $p_j$ .
- Let  $mt_e$  denote the matrix timestamp associated with event  $e$ . Due to message  $m_4$ ,  $e_k^2$  is the last event of  $p_k$  that causally precedes  $e$ , therefore, we have  $mt_e[i, k] = mt_e[k, k] = x_k^2$ .
- Likewise,  $mt_e[i, j] = mt_e[j, j] = x_j^2$ . The last event of  $p_k$  known by  $p_j$ , to the knowledge of  $p_i$  when it executed event  $e$ , is  $e_k^1$ ; therefore,  $mt_e[j, k] = x_k^1$ . Likewise, we have  $mt_e[k, j] = x_j^1$ .

## Matrix Time

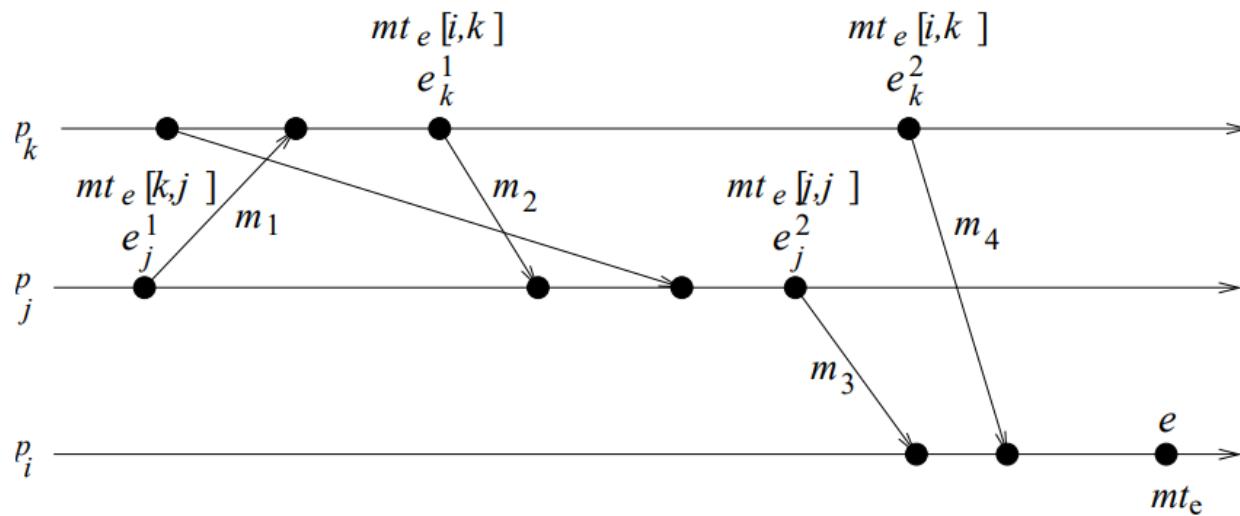
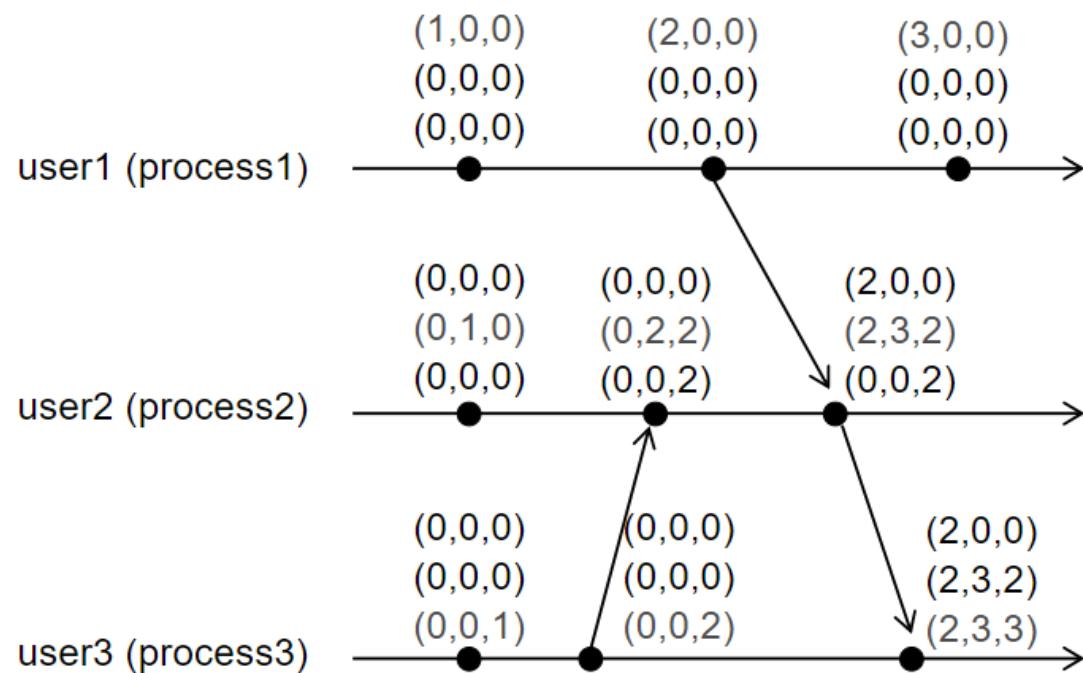


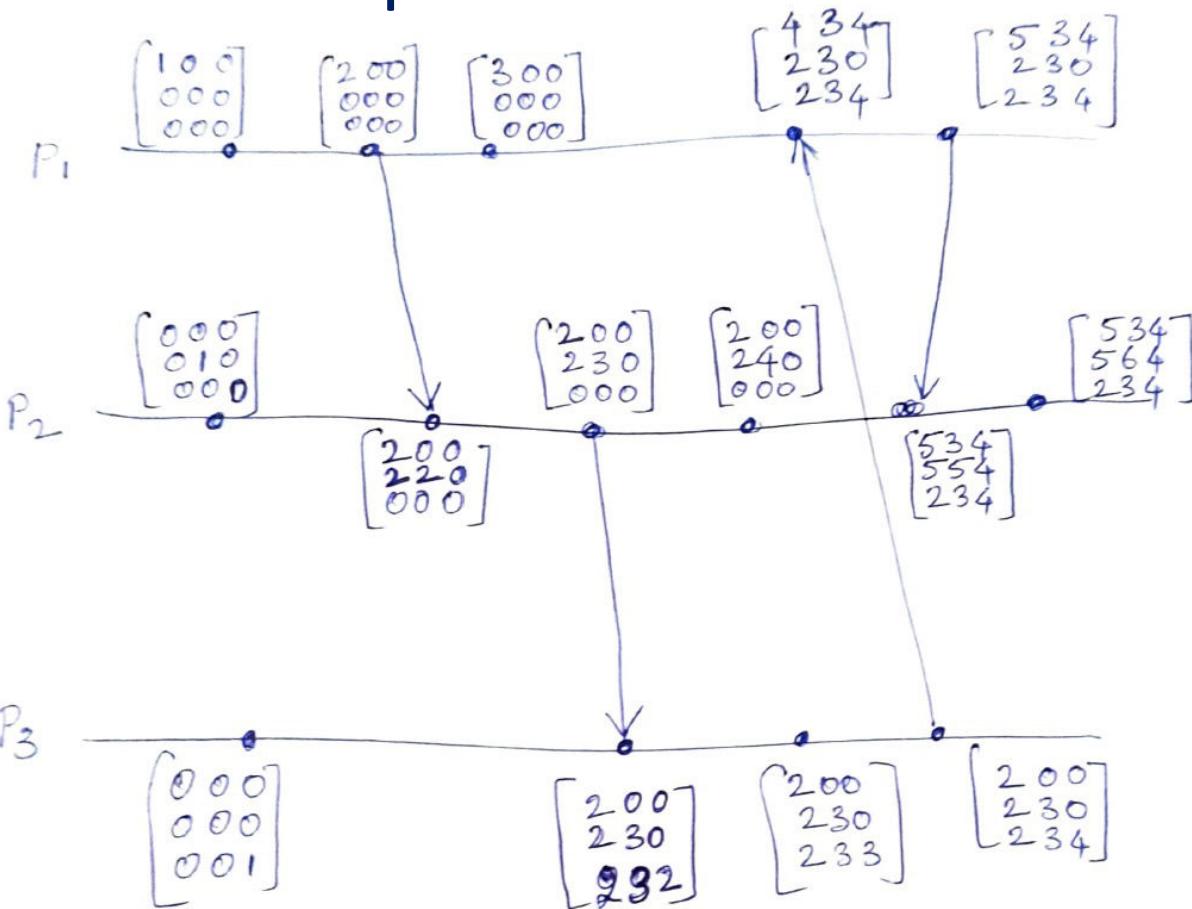
Figure 3.4: Evolution of matrix time.

# Matrix Time – Example 1

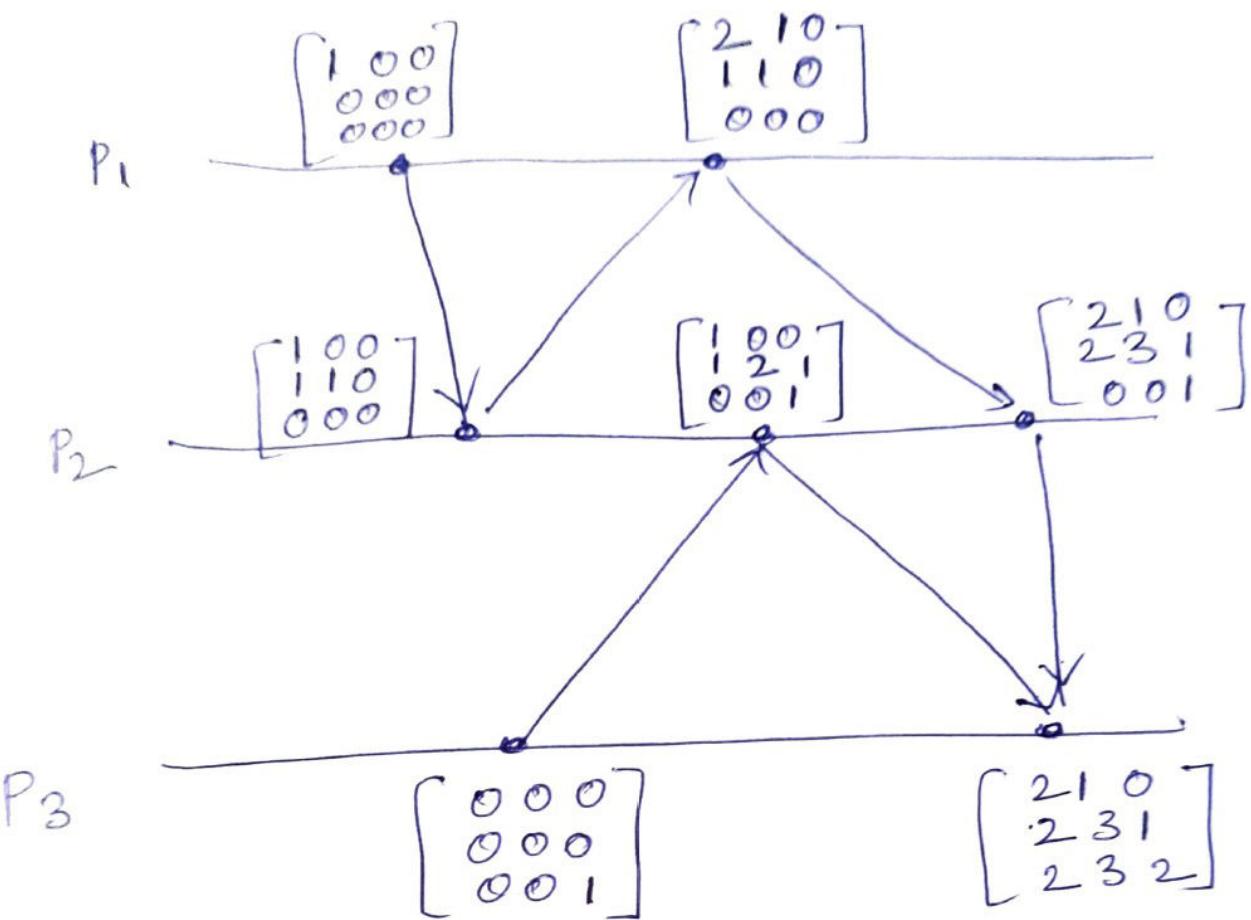
Matrix Clock Example



## Matrix Time – Example 2



## Matrix Time – Example 3



# Matrix Time

## Basic Properties

- Vector  $mt_i[i, .]$  contains all the properties of vector clocks.
- In addition, matrix clocks have the following property:  
 $\min_k(mt_i[k, l]) \geq t \Rightarrow$  process  $p_i$  knows that every other process  $p_k$  knows that  $p_l$ 's local time has progressed till  $t$ .
  - ▶ If this is true, it is clear that process  $p_i$  knows that all other processes know that  $p_l$  will never send information with a local time  $\leq t$ .
  - ▶ In many applications, this implies that processes will no longer require from  $p_l$  certain information and can use this fact to discard obsolete information.
- If  $d$  is always 1 in the rule  $R1$ , then  $mt_i[k, l]$  denotes the number of events occurred at  $p_l$  and known by  $p_k$  as far as  $p_i$ 's knowledge is concerned.

# Types of Parallel/Distributed Computations

# 1. Embarrassingly Parallel Computation

- also called naturally parallel
- Dividing the problem into completely independent parts that can be executed simultaneously
- There is no communication between the separate processes
- Simply distribute the data and start the process
- Appropriate for SPMD model

# contd..

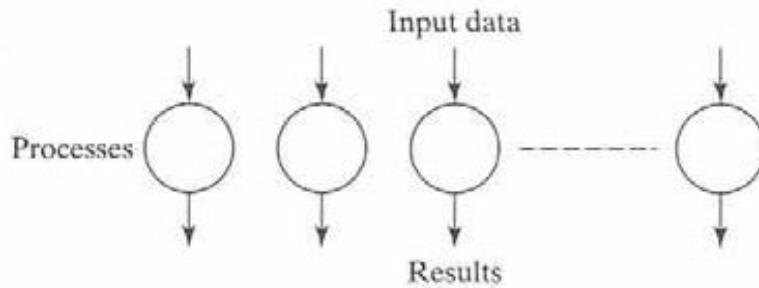


Figure 3.1 Disconnected computational graph (embarrassingly parallel problem).

- A master process is started that spawns identical slaves

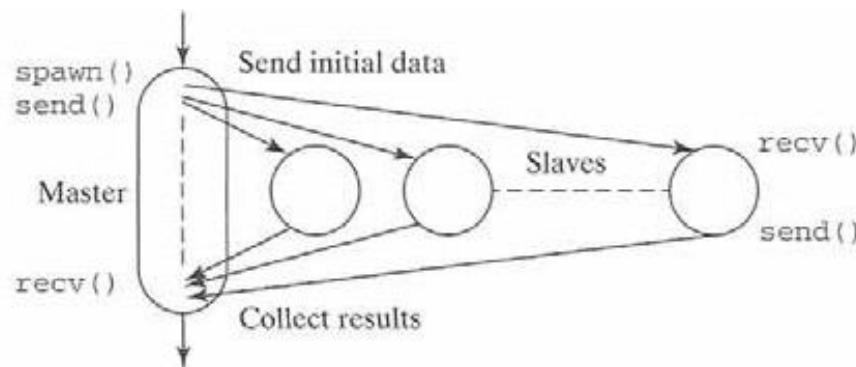


Figure 3.2 Practical embarrassingly parallel computational graph with dynamic process creation and the master-slave approach.

# Example

- Geometrical transformation of images

- (a) Shifting

The coordinates of a two-dimensional object shifted by  $\Delta x$  in the  $x$ -dimension and  $\Delta y$  in the  $y$ -dimension are given by

$$x' = x + \Delta x$$

$$y' = y + \Delta y$$

where  $x$  and  $y$  are the original and  $x'$  and  $y'$  are the new coordinates.

- (b) Scaling

The coordinates of an object scaled by a factor  $S_x$  in the  $x$ -direction and  $S_y$  in the  $y$ -direction are given by

$$x' = xS_x$$

$$y' = yS_y$$

- (c) Rotation

The coordinates of an object rotated through an angle  $\theta$  about the origin of the coordinate system are given by

$$x' = x \cos \theta + y \sin \theta$$

$$y' = -x \sin \theta + y \cos \theta$$

# Partitioning for n=48 processes

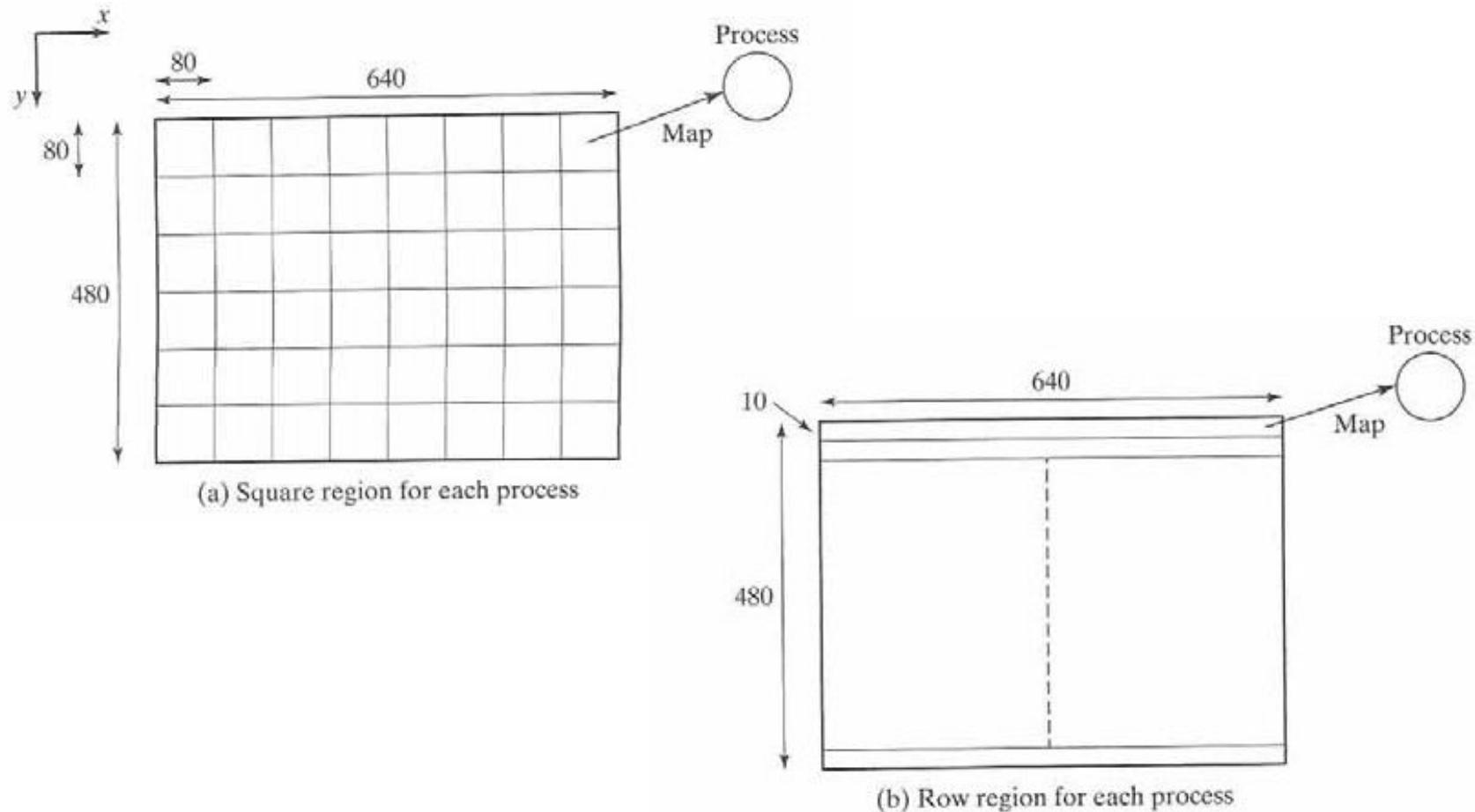


Figure 3.3 Partitioning into regions for individual processes.

# Parallel code

Master

```
for (i = 0, row = 0; i < 48; i++, row = row + 10) /* for each process*/
    send(row, P1);                                /* send row no.*/
for (i = 0; i < 480; i++)                         /* initialize temp */
    for (j = 0; j < 640; j++)
        temp_map[i][j] = 0;
for (i = 0; i < (640 * 480); i++) {                /* for each pixel */
    recv(oldrow,oldcol,newrow,newcol, PANY);        /* accept new coords */
    if !((newrow < 0) || (newrow >= 480) || (newcol < 0) || (newcol >= 640))
        temp_map[newrow][newcol]=map[oldrow][oldcol];
}
for (i = 0; i < 480; i++)                          /* update bitmap */
    for (j = 0; j < 640; j++)
        map[i][j] = temp_map[i][j];
```

# contd..

## Slave

```
recv(row, Pmaster);                                /* receive row no. */
for (oldrow = row; oldrow < (row + 10); oldrow++)
    for (oldcol = 0; oldcol < 640; oldcol++) {      /* transform coords */
        newrow = oldrow + delta_x;                  /* shift in x direction */
        newcol = oldcol + delta_y;                  /* shift in y direction */
        send(oldrow,oldcol,newrow,newcol, Pmaster);  /* coords to master */
    }
```

# Analysis

- Sequential complexity

$$t_S = 2n^2$$

- Parallel complexity

$$t_{\text{comm}} = t_{\text{startup}} + mt_{\text{data}}$$

$$t_{\text{comm}} = p(t_{\text{startup}} + t_{\text{data}}) + n^2(t_{\text{startup}} + 4t_{\text{data}}) = O(p + n^2)$$

$$t_{\text{comp}} = 2 \left( \frac{n^2}{p} \right) = O(n^2/p)$$

# Overall speedup

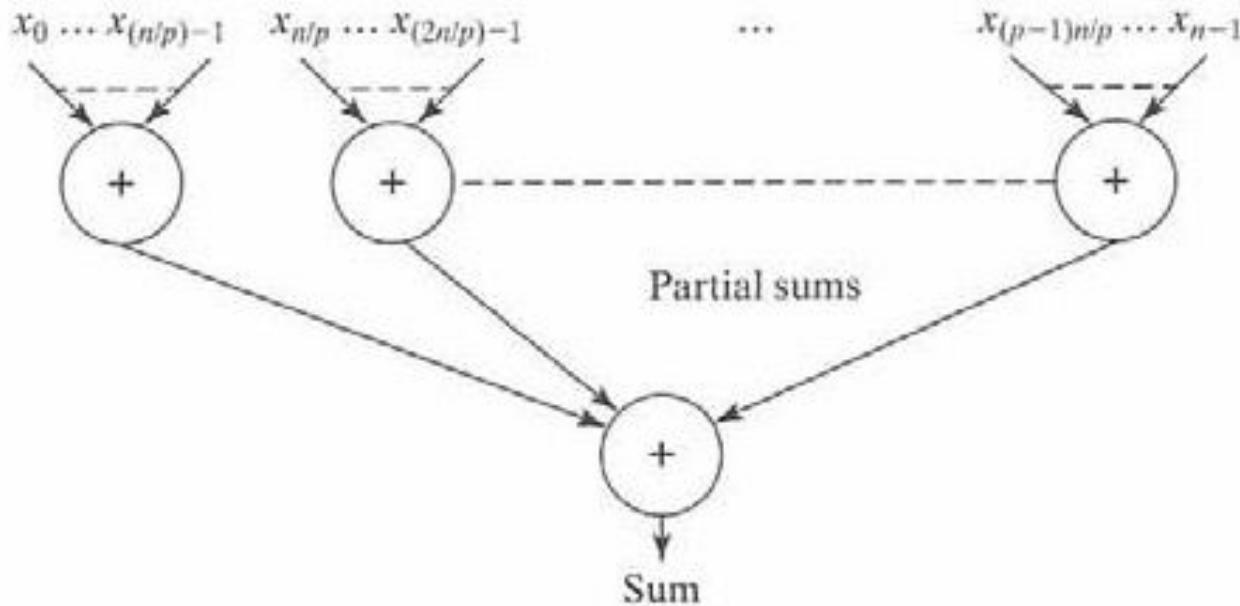
$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{2n^2}{2\left(\frac{n^2}{p}\right) + p(t_{\text{startup}} + t_{\text{data}}) + n^2(t_{\text{startup}} + 4t_{\text{data}})}$$

$$\begin{aligned}\text{Computation/communication ratio} &= \frac{2(n^2/p)}{p(t_{\text{startup}} + 2t_{\text{data}}) + 4n^2(t_{\text{startup}} + t_{\text{data}})} \\ &= O\left(\frac{n^2/p}{p + n^2}\right)\end{aligned}$$

## 2. Partitioning and Divide and Conquer

- In partitioning, the problem is divided into separate parts and each part is computed separately
- In divide and conquer, the partitioning is applied recursively, solving the smaller parts and combining the results

# Partitioning example



**Figure 4.1** Partitioning a sequence of numbers into parts and adding them.

# Example-sum of n numbers

## Master

```
s = n/p;                                /* number of numbers for slaves*/
for (i = 0, x = 0; i < p; i++, x = x + s)
    send(&numbers[x], s, Pi);           /* send s numbers to slave */

sum = 0;
for (i = 0; i < p; i++) {                  /* wait for results from slaves */
    recv(&part_sum, PANY);
    sum = sum + part_sum;                 /* accumulate partial sums */
}
```

## Slave

```
recv(numbers, s, Pmaster);            /* receive s numbers from master */
part_sum = 0;
for (i = 0; i < s; i++)                 /* add numbers */
    part_sum = part_sum + numbers[i];
send(&part_sum, Pmaster);             /* send sum to master */
```

# Using broadcast routine

## Master

```
s = n/p;                                /* number of numbers for slaves */
bcast(numbers, s, Pslave_group);          /* send all numbers to slaves */
sum = 0;
for (i = 0; i < p; i++) {                  /* wait for results from slaves */
    recv(&part_sum, PANY);
    sum = sum + part_sum;                  /* accumulate partial sums */
}
```

## Slave

```
bcast(numbers, s, Pmaster);              /* receive all numbers from master*/
start = slave_number * s;                 /* slave number obtained earlier */
end = start + s;
part_sum = 0;
for (i = start; i < end; i++)           /* add numbers */
    part_sum = part_sum + numbers[i];
send(&part_sum, Pmaster);                /* send sum to master */
```

# Analysis

$$t_{\text{comm1}} = p(t_{\text{startup}} + (n/p)t_{\text{data}})$$

$$t_{\text{compl1}} = n/p - 1$$

$$t_{\text{comm2}} = p(t_{\text{startup}} + t_{\text{data}})$$

$$t_{\text{comp2}} = p - 1$$

# Overall execution time

$$\begin{aligned}t_p &= (t_{\text{comm1}} + t_{\text{comm2}}) + (t_{\text{comp1}} + t_{\text{comp2}}) \\&= p(t_{\text{startup}} + (n/p)t_{\text{data}}) + p(t_{\text{startup}} + t_{\text{data}}) + (n/p - 1 + p - 1) \\&= 2pt_{\text{startup}} + (n + p)t_{\text{data}} + p + n/p - 2\end{aligned}$$

$$t_p = O(n)$$

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n - 1}{2pt_{\text{startup}} + (n + p)t_{\text{data}} + p + n/p - 2}$$

$$\text{Computation/communication ratio} = \frac{t_{\text{comp}}}{t_{\text{comm}}} = \frac{p + n/p - 2}{2pt_{\text{startup}} + (n + p)t_{\text{data}}}$$

# 3. Divide and Conquer

```
int add(int *s)                                /* add list of numbers, s */
{
    if (number(s) <= 2) return (n1 + n2);      /* see explanation */
    else {
        Divide (s, s1, s2);                  /* divide s into two parts, s1 and s2 */
        part_sum1 = add(s1);                 /*recursive calls to add sub lists */
        part_sum2 = add(s2);
        return (part_sum1 + part_sum2);
    }
}
```

# Parallel version

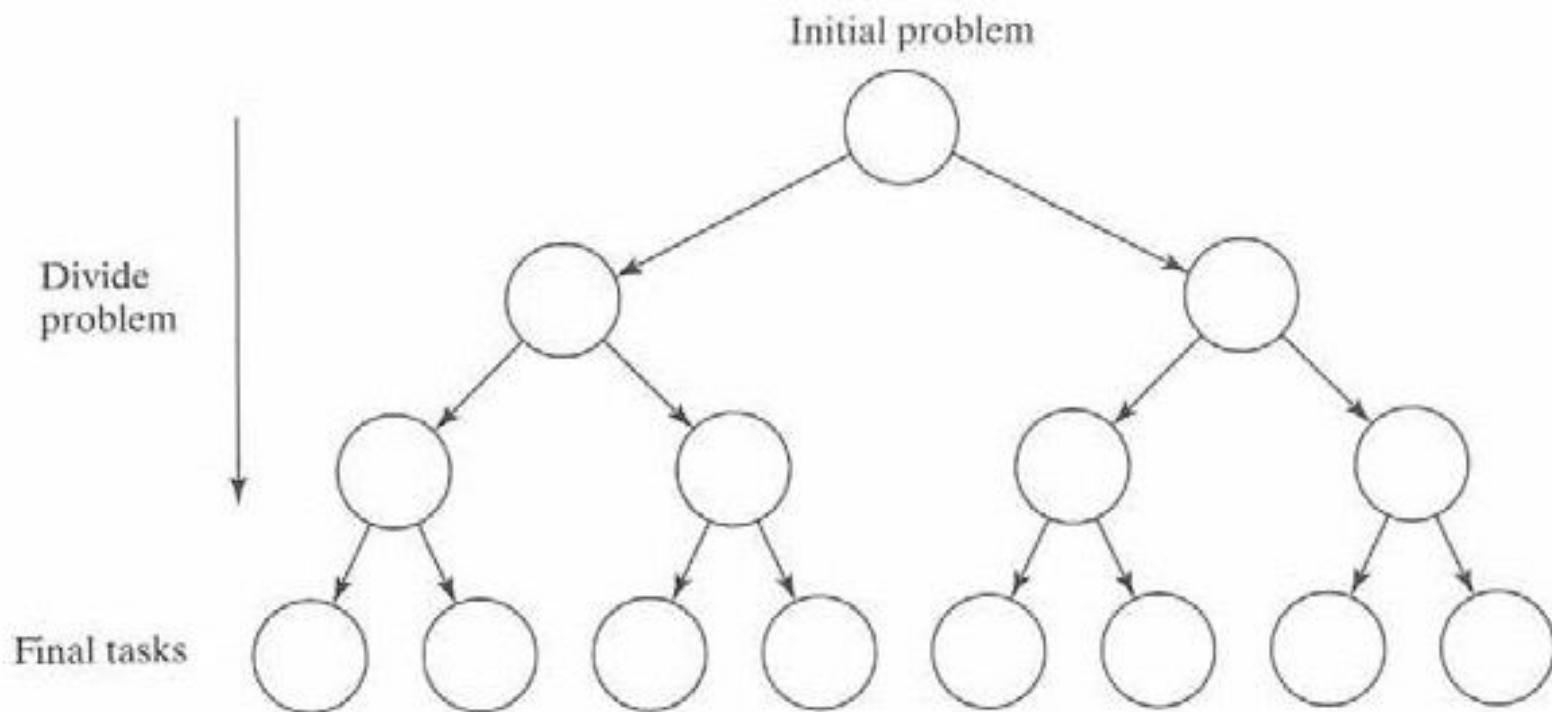
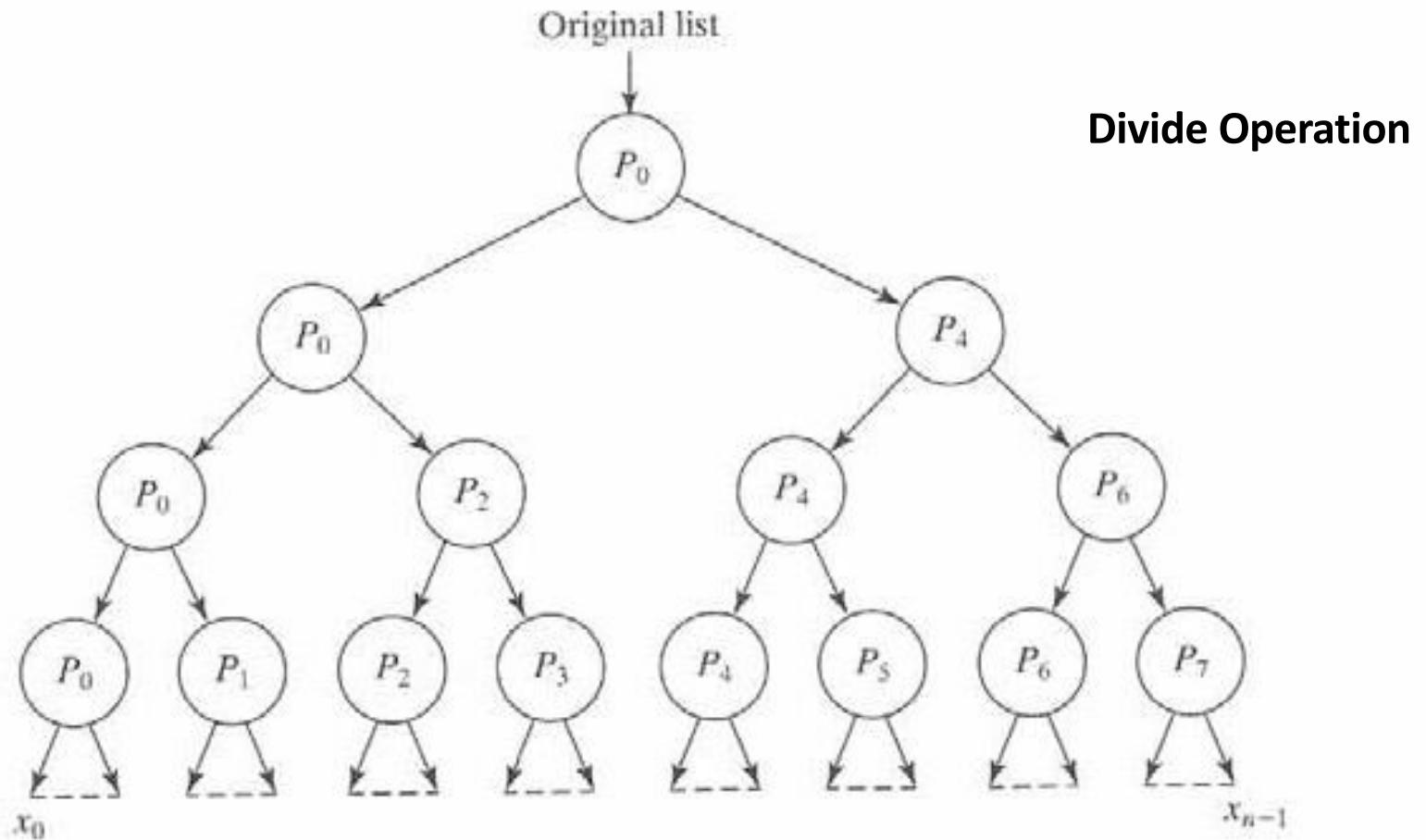


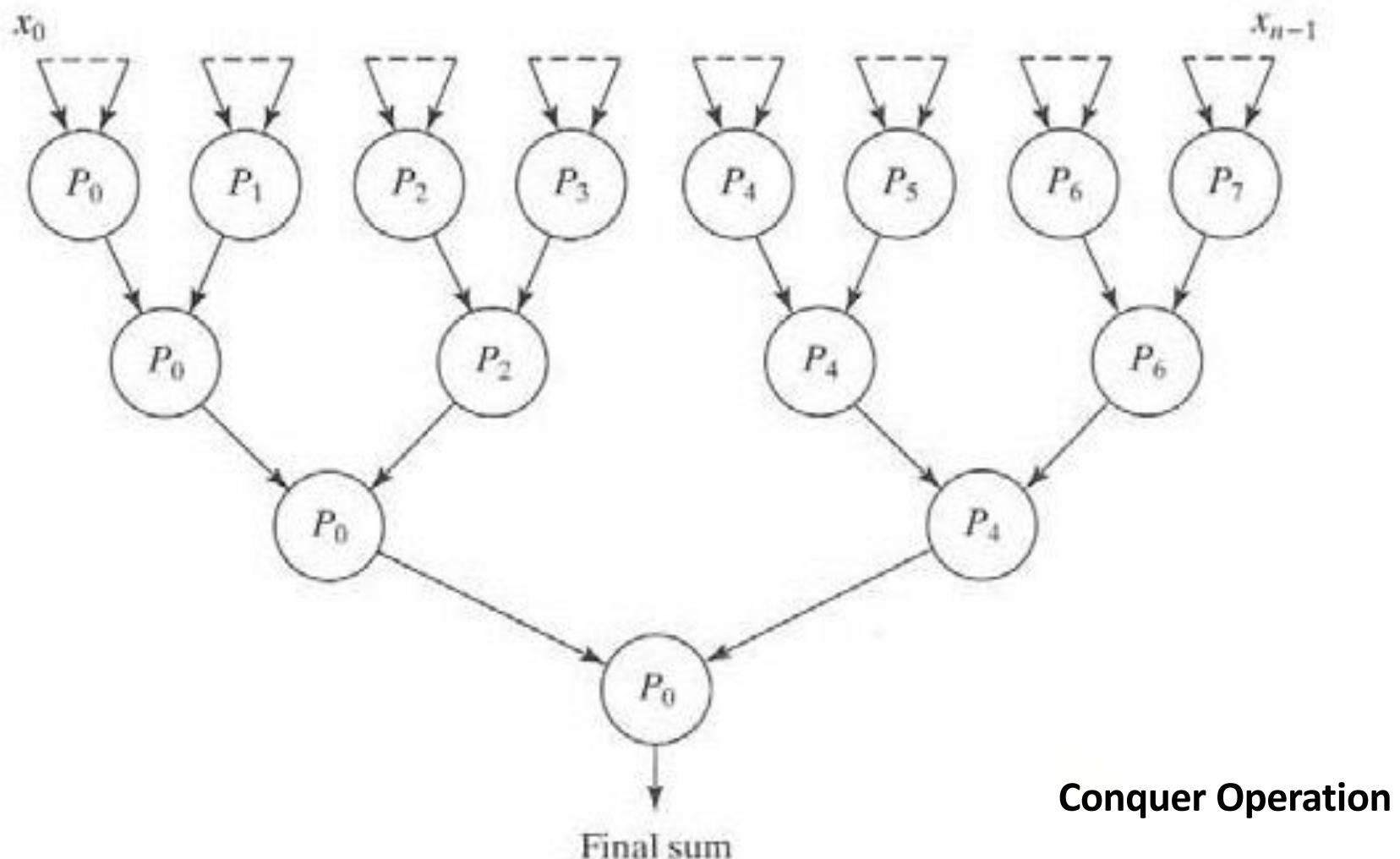
Figure 4.2 Tree construction.

Needs  $2^{m+1}-1$  processors

# Parallel version (with processor reuse)



contd..



# Code for Process 0

Process  $P_0$

```
divide(s1, s1, s2);           /* division phase */
send(s2, P4);              /* divide s1 into two, s1 and s2 */
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P1);
part_sum = *s1;                /* combining phase */
recv(&part_sum1, P1);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
part_sum = part_sum + part_sum1;
```

# Code for Process 4

Process  $P_4$

```
recv(sl, P0);                                /* division phase */  
divide(sl, s1, s2);  
send(s2, P6);  
divide(s1, s1, s2);  
send(s2, P5);  
part_sum = *s1;                                /* combining phase */  
recv(&part_sum1, P5);  
part_sum = part_sum + part_sum1;  
recv(&part_sum1, P6);  
part_sum = part_sum + part_sum1;  
send(&part_sum, P0);
```

# Analysis

$$t_{\text{comm1}} = \frac{n}{2}t_{\text{data}} + \frac{n}{4}t_{\text{data}} + \frac{n}{8}t_{\text{data}} + \dots + \frac{n}{p}t_{\text{data}} = \frac{n(p-1)}{p}t_{\text{data}}$$

$$t_{\text{comm2}} = (\log p)t_{\text{data}}$$

$$t_{\text{comm}} = t_{\text{comm1}} + t_{\text{comm2}} = \frac{n(p-1)}{p}t_{\text{data}} + (\log p)t_{\text{data}}$$

$$t_{\text{comp}} = \frac{n}{p} + \log p$$

## contd..

**Overall Execution Time.** The total parallel execution time becomes

$$t_p = \left( \frac{n(p-1)}{p} + \log p \right) t_{\text{data}} + \frac{n}{p} + \log p$$

**Speedup Factor.** The speedup factor is

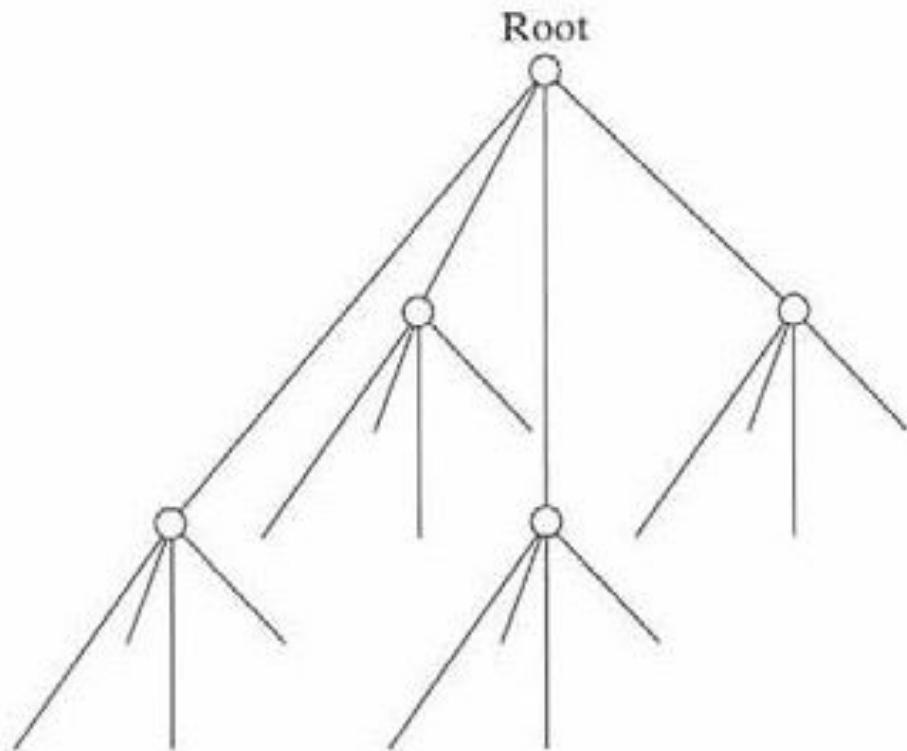
$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n-1}{((n/p)(p-1) + \log p)t_{\text{data}} + n/p + \log p}$$

$$\text{Computation/communication ratio} = \frac{t_{\text{comp}}}{t_{\text{comm}}} = \frac{n/p + \log p}{((n/p)(p-1) + \log p)t_{\text{data}}}$$

# M-ary divide and conquer

```
int add(int *s)                      /* add list of numbers, s */
{
    if (number(s) <= 4) return(n1 + n2 + n3 + n4);
    else {
        Divide (s,s1,s2,s3,s4);      /* divide s into s1,s2,s3,s4*/
        part_sum1 = add(s1);          /*recursive calls to add sublists */
        part_sum2 = add(s2);
        part_sum3 = add(s3);
        part_sum4 = add(s4);
        return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
    }
}
```

contd..



**Figure 4.6** Quadtree.

# 4. Partitioning and Divide-and-Conquer Approach

- Sorting using bucket sort

$$t_s = n + m((n/m)\log(n/m)) = n + n \log(n/m) = O(n \log(n/m))$$

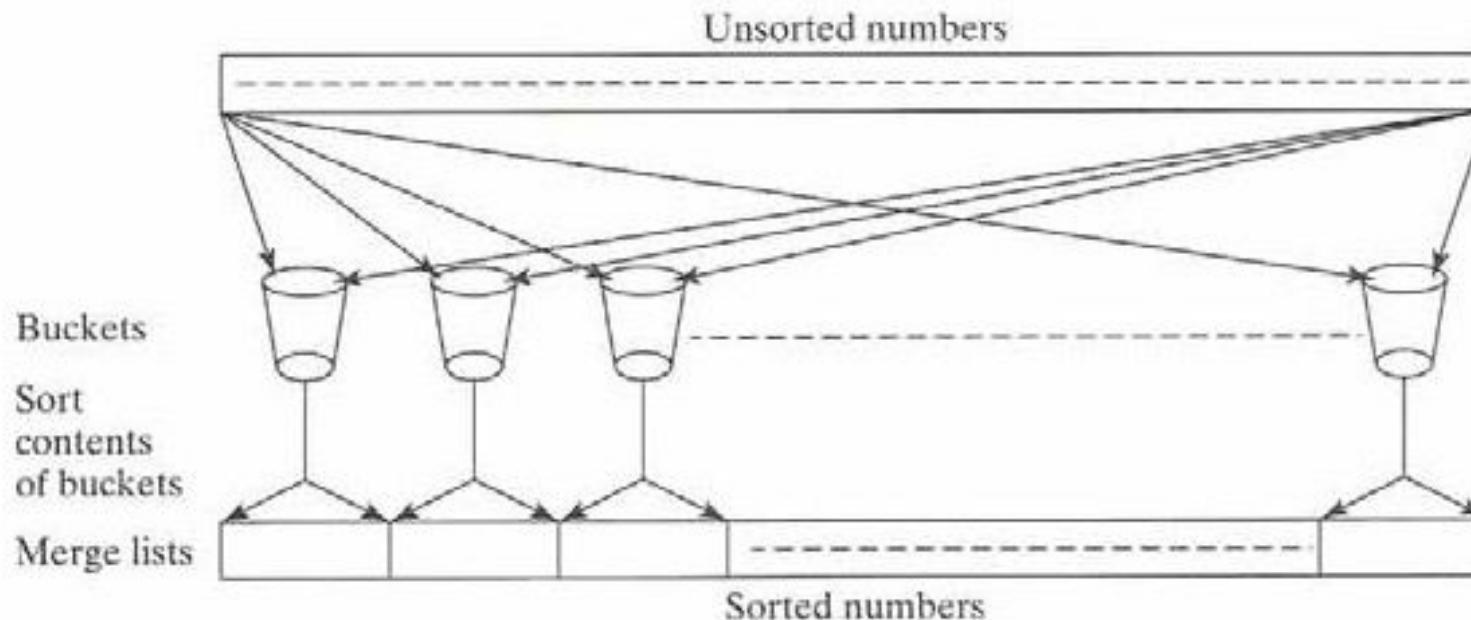


Figure 4.8 Bucket sort.

# Parallel Version

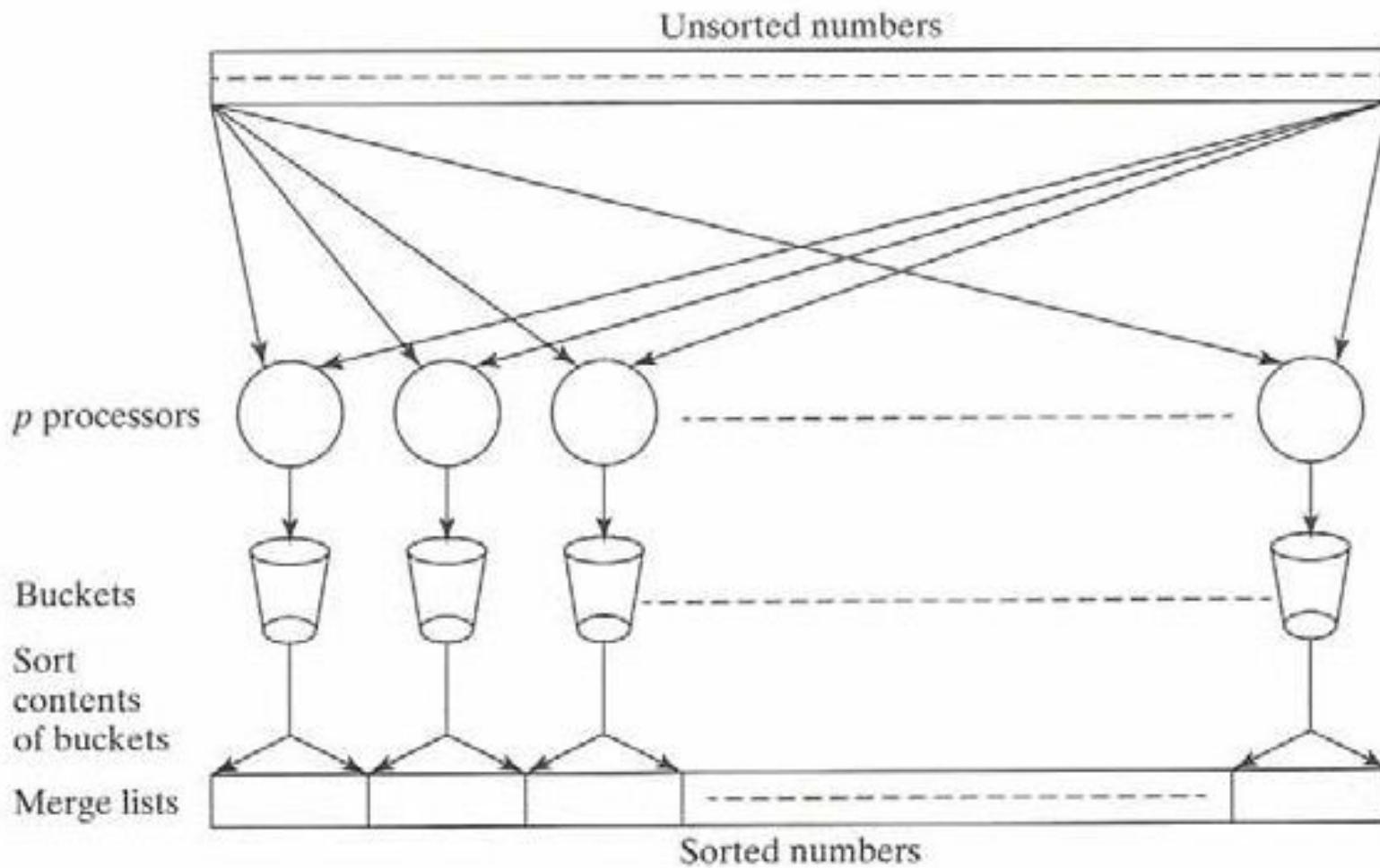


Figure 4.9 One parallel version of bucket sort.

# contd..

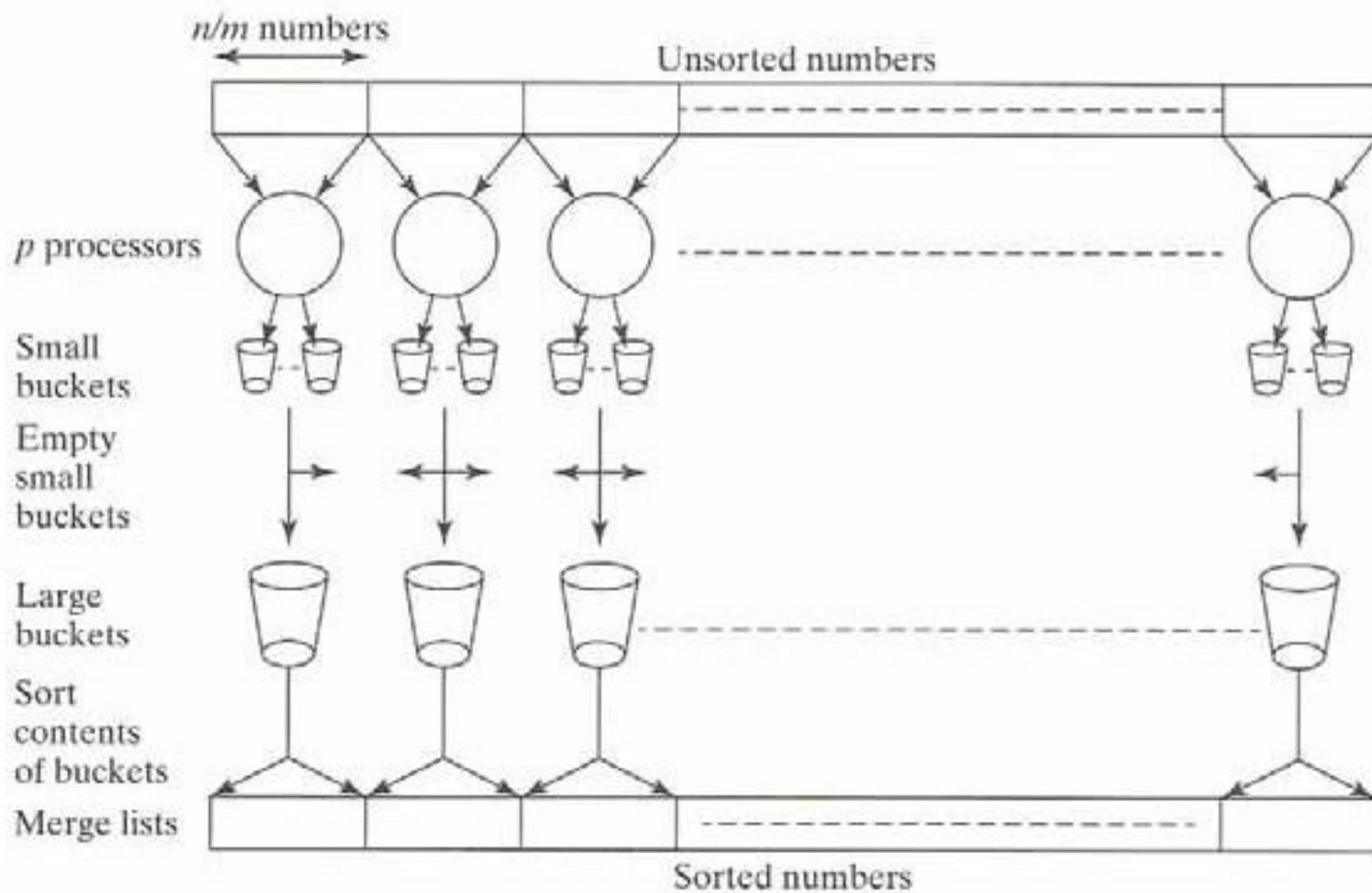


Figure 4.10 Parallel version of bucket sort.

# Analysis

$$t_{\text{comm1}} = t_{\text{startup}} + nt_{\text{data}}$$

$$t_{\text{comp2}} = n/p$$

$$t_{\text{comm3}} = p(p - 1)(t_{\text{startup}} + (n/p^2)t_{\text{data}})$$

or

$$t_{\text{comm3}} = (p - 1)(t_{\text{startup}} + (n/p^2)t_{\text{data}})$$

$$t_{\text{comp4}} = (n/p)\log(n/p)$$

# contd..

*Overall Execution Time.* The overall run time, including communication, is

$$\begin{aligned}t_p &= t_{\text{comm1}} + t_{\text{comp2}} + t_{\text{comm3}} + t_{\text{comp4}} \\t_p &= t_{\text{startup}} + nt_{\text{data}} + n/p + (p-1)(t_{\text{startup}} + (n/p^2)t_{\text{data}}) + (n/p)\log(n/p) \\&= (n/p)(1 + \log(n/p)) + pt_{\text{startup}} + (n + (p-1)(n/p^2))t_{\text{data}}\end{aligned}$$

*Speedup Factor.* The speedup factor, when compared to sequential bucket sort, is

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n + n\log(n/m)}{(n/p)(1 + \log(n/p)) + pt_{\text{startup}} + (n + (p-1)(n/p^2))t_{\text{data}}}$$

$$\text{Computation/communication ratio} = \frac{t_{\text{comp}}}{t_{\text{comm}}} = \frac{(n/p)(1 + \log(n/p))}{pt_{\text{startup}} + (n + (p-1))(n/p^2)t_{\text{data}}}$$

# All-to-all broadcast

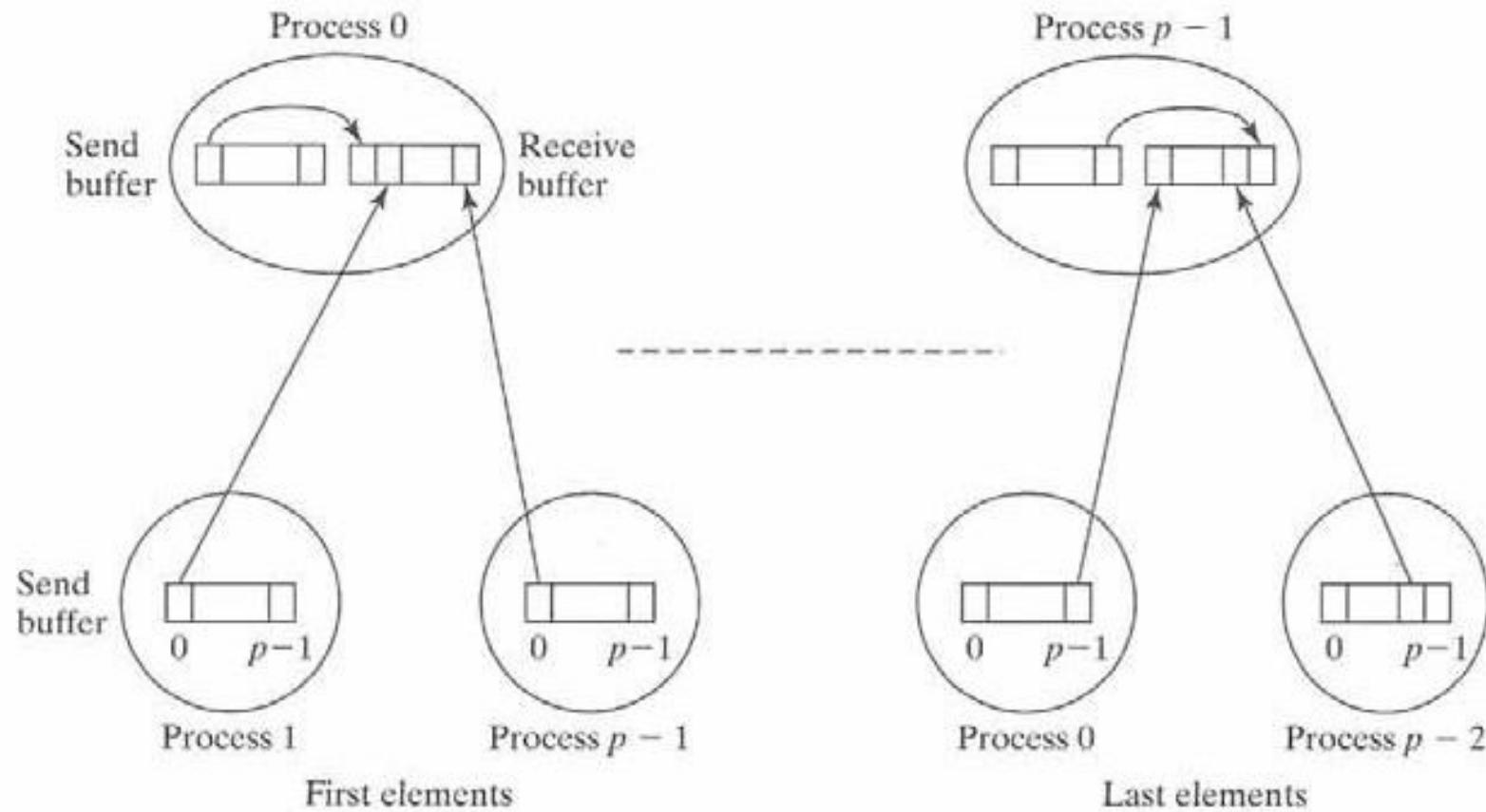


Figure 4.11 All-to-all broadcast.

# contd..

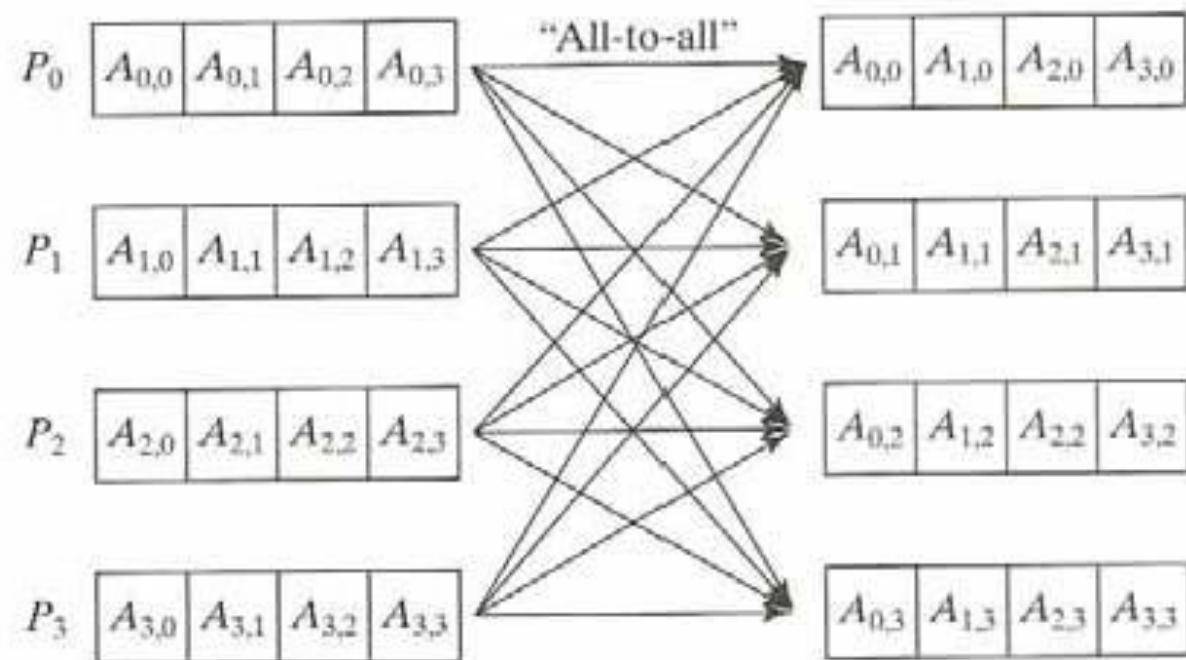


Figure 4.12 Effect of all-to-all on an array.

# Exercises

Write parallel programs to compute  $n!$  in each of the following ways and assess their performance. The number,  $n$ , may be odd or even but is a positive constant.

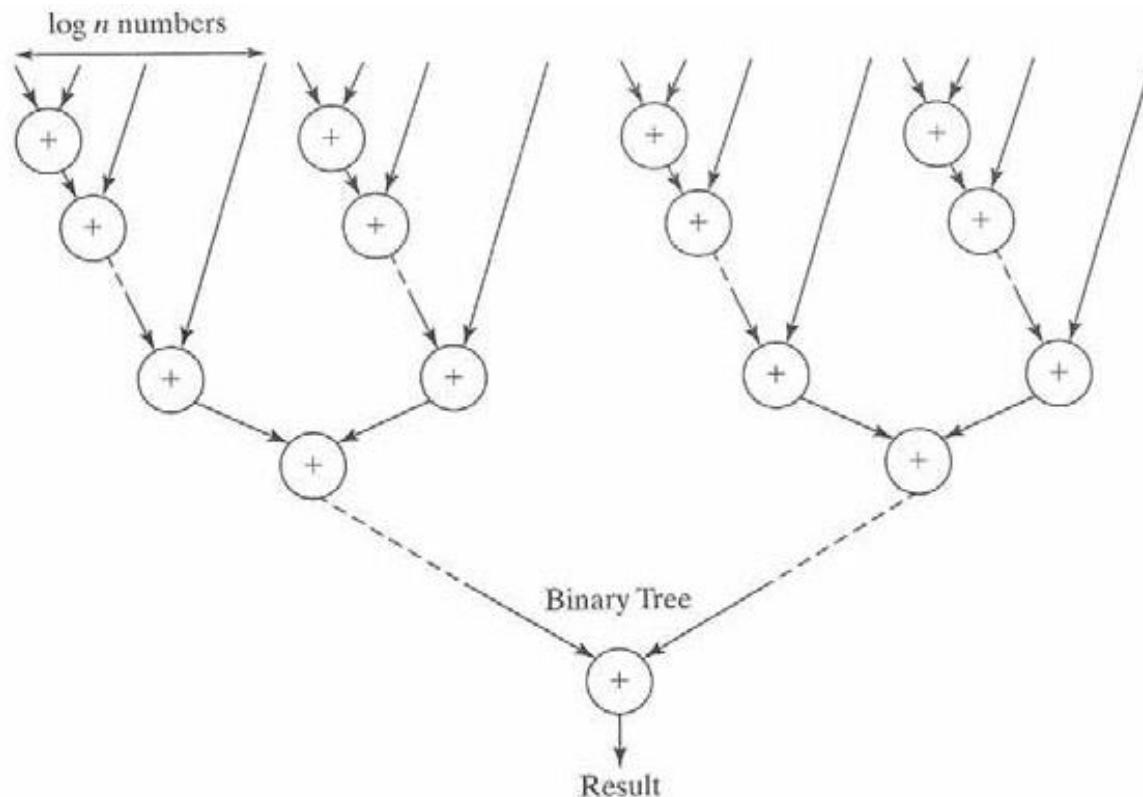
- (a) Compute  $n!$  using two concurrent processes, each computing approximately half of the complete sequence. A master process then combines the two partial results.
- (b) Compute  $n!$  using a producer process and a consumer process connected together. The producer produces the numbers  $1, 2, 3, \dots, n$  in sequence. The consumer accepts the sequence of numbers from the producer and accumulates the result; i.e.,  $1 \times 2 \times 3 \dots$ .

Write parallel programs to compute the summation of  $n$  integers in each of the following ways and assess their performance. Assume that  $n$  is a power of 2.

- (a) Partition the  $n$  integers into  $n/2$  pairs. Use  $n/2$  processes to add together each pair of integers resulting in  $n/2$  integers. Repeat the method on the  $n/2$  integers to obtain  $n/4$  integers and continue until the final result is obtained. (This is a binary tree algorithm.)

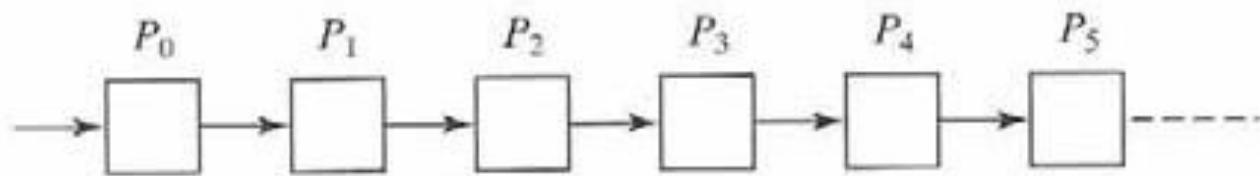
# contd..

- (b) Divide the  $n$  integers into  $n/\log n$  groups of  $\log n$  numbers each. Use  $n/\log n$  processes, each adding the numbers in one group sequentially. Then add the  $n/\log n$  results using method (a).



# 5. Pipelining

- It is a form of functional decomposition



- Consider a simple loop

```
for (i = 0; i < n; i++)  
    sum = sum + a[i];
```

## contd..

- The loop can be unrolled as

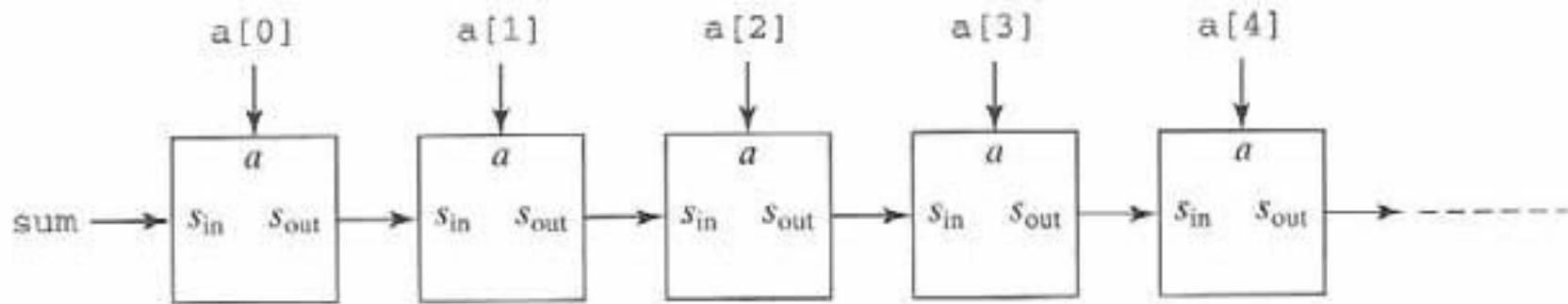
```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
:
```

- Pipelining solution is to separate as

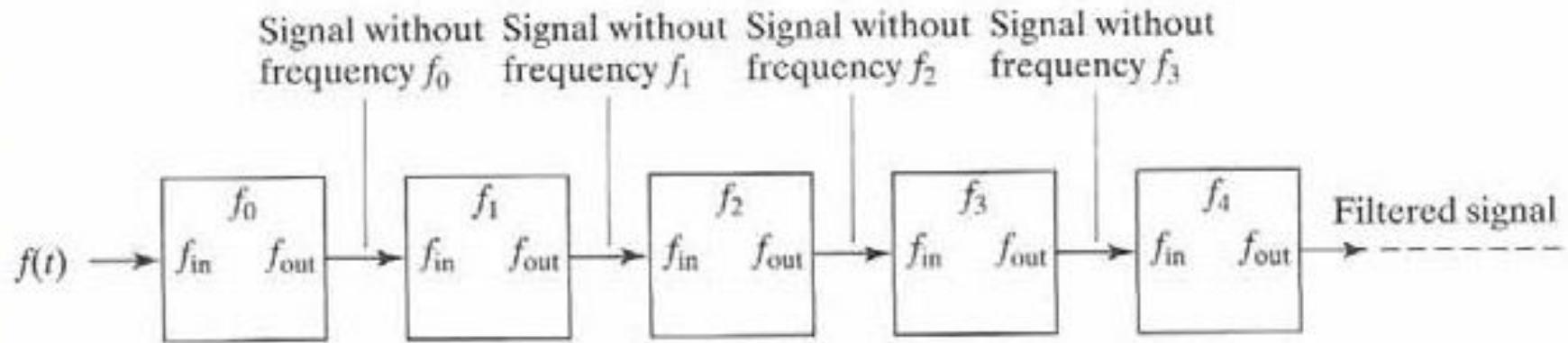
```
sout = sin + a[i];
```

where  $i^{\text{th}}$  stage (processor) performs adding of  $i^{\text{th}}$  array element to the previous accumulated sum

# contd..



## Example: Frequency Filter

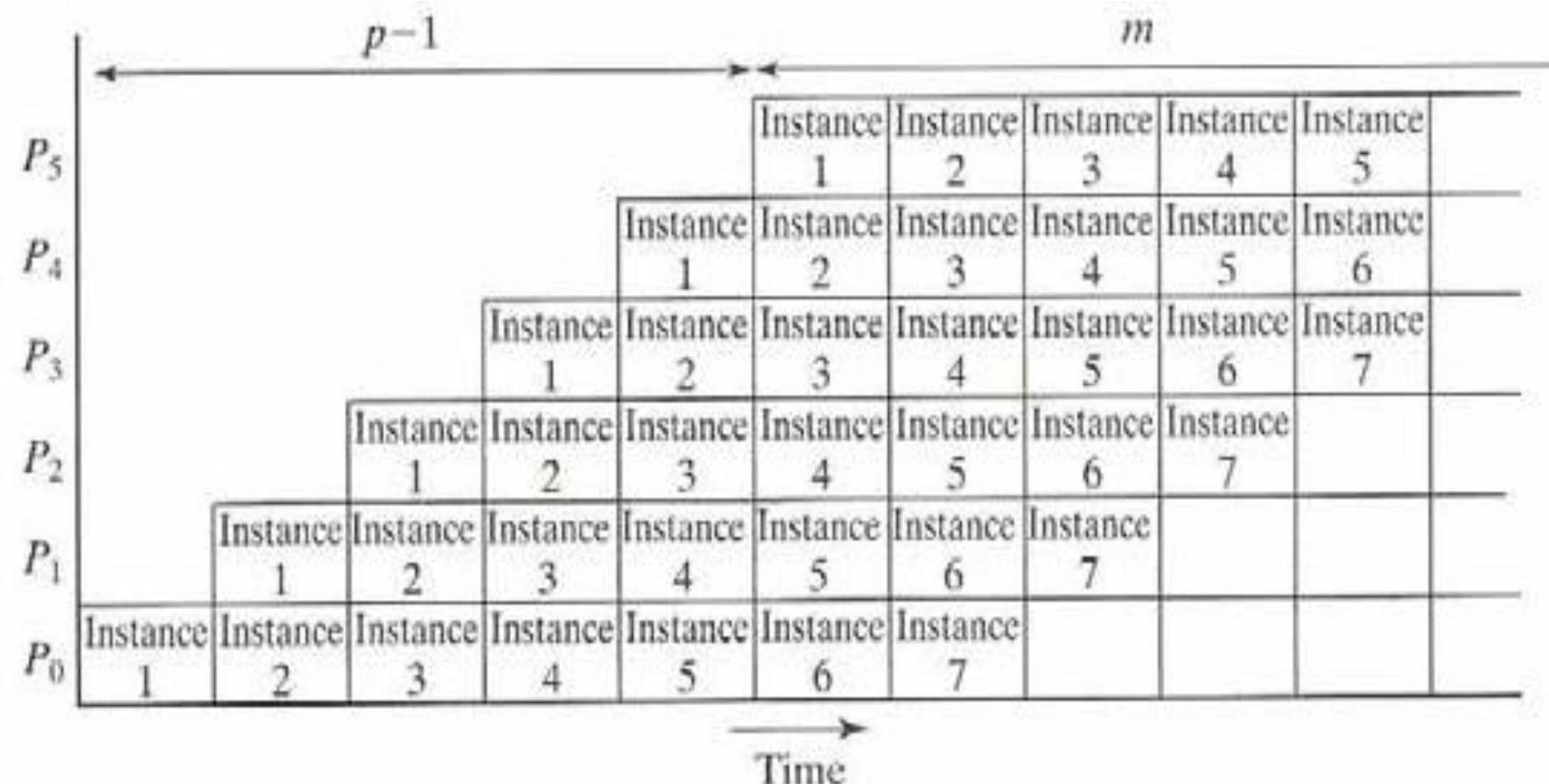


# Types of Pipelining

- If more than one instance of the complete problem is to be executed – TYPE 1
- If a series of data items must be processed, each requiring multiple operations – TYPE 2
- If the information to start the next process can be passed forward before the process has completed all its internal operations – TYPE 3

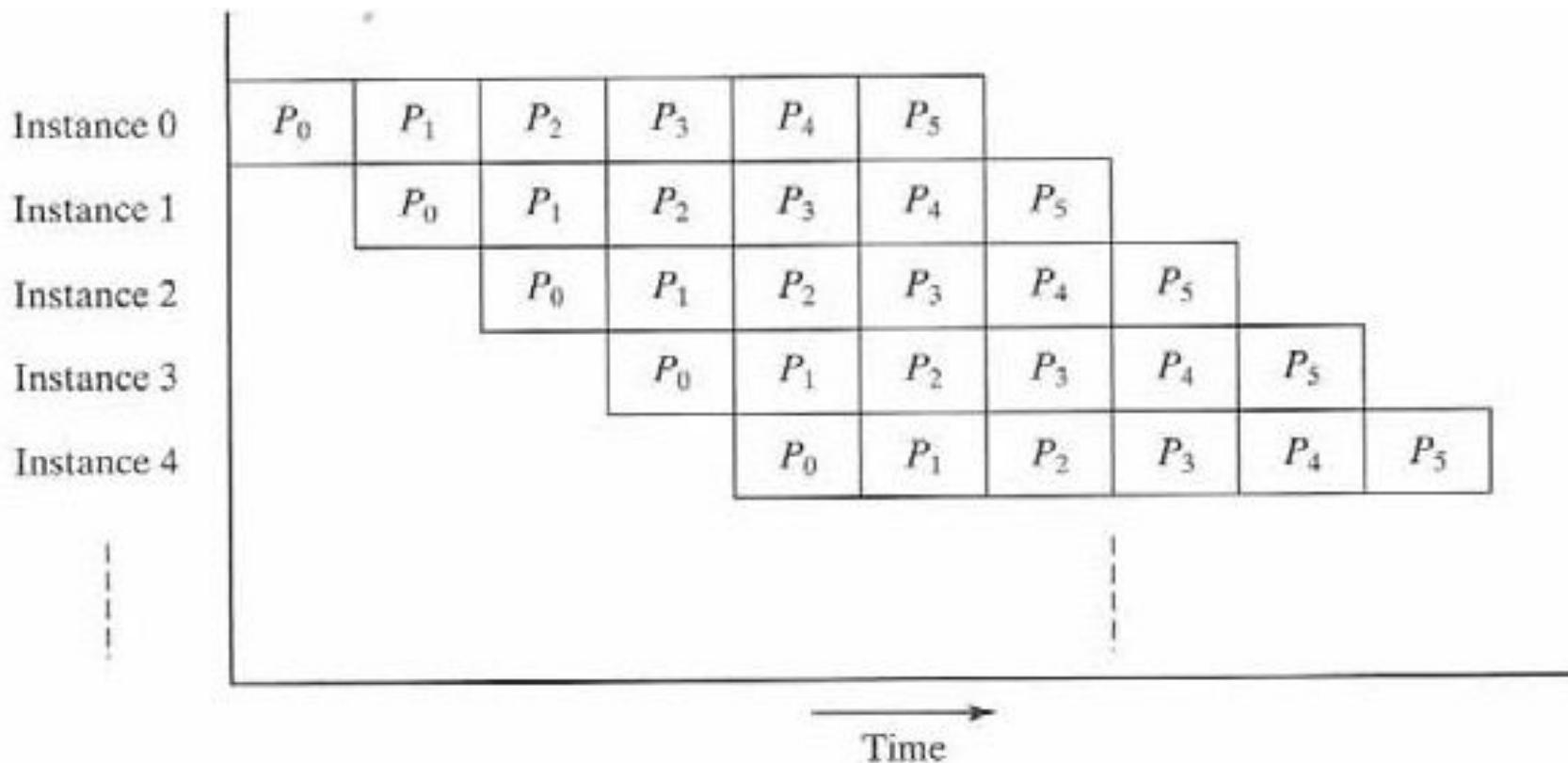
# Type 1

- Each time period is one pipeline cycle
- Each instance of the problem requires six sequential processes. Space time diagram is shown below:



# contd..

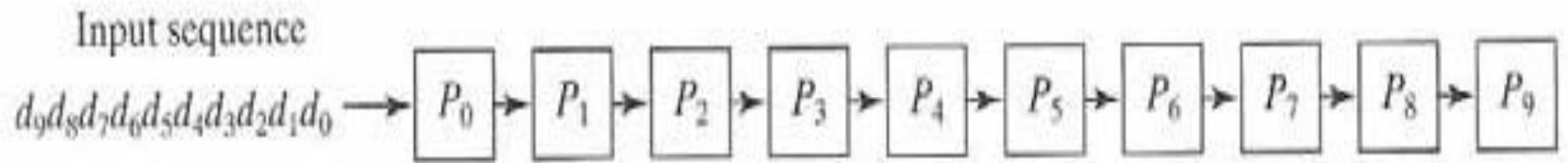
- Alternate space time diagram



## contd..

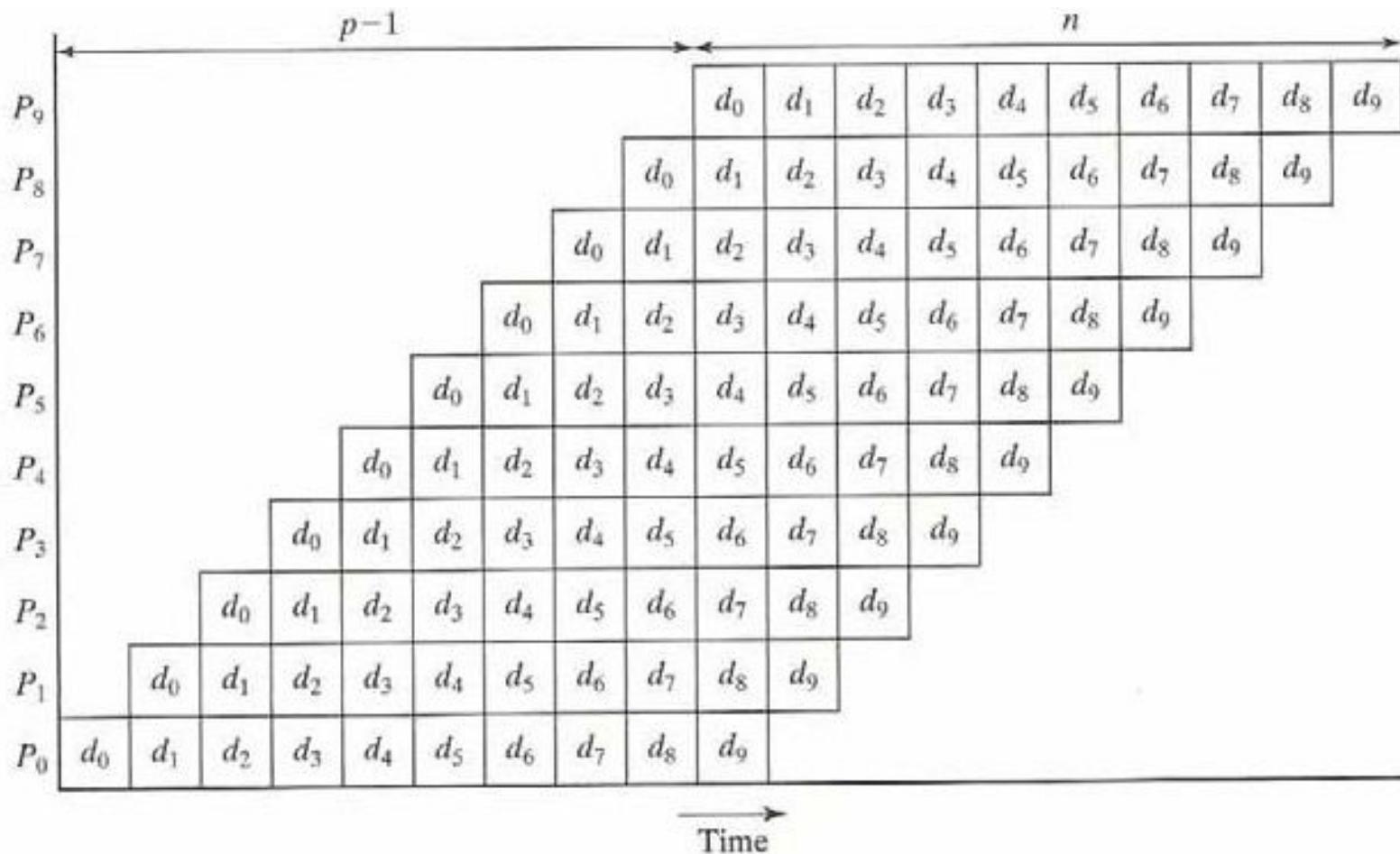
- The number of pipelining cycles to execute all  $m$  instances is  $(m+p-1)$
- Average number of cycles is  $(m+p-1)/m$  (i.e) one cycle per problem instance for large  $m$
- One instance of the problem will be completed in each pipeline cycle after  $p-1$  cycles (pipeline latency)

# Type 2

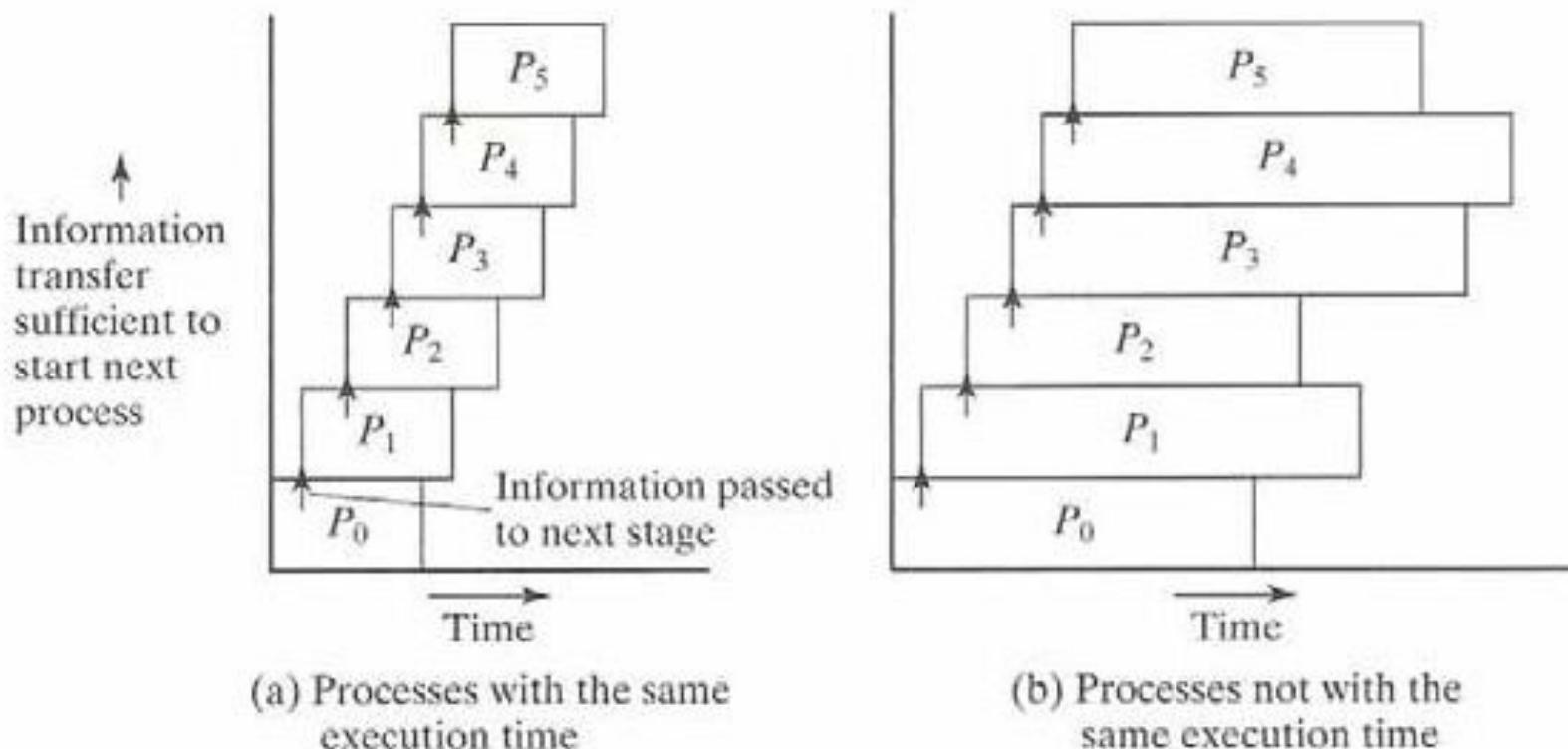


- Here, series of data items are processed in a sequence
- With  $p$  processors, and  $n$  data items, the overall execution time is  $(p-1)+n$  pipeline cycles

contd..

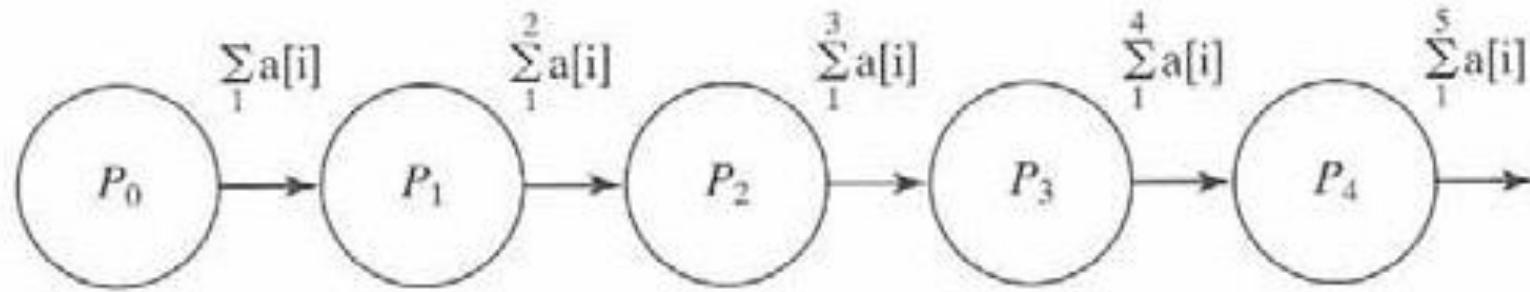


# Type 3



# Type 1 - Example

- Adding of list of numbers



- Code for process  $P_i$

```
recv(&accumulation, Pi-1);  
accumulation = accumulation + number;  
send(&accumulation, Pi+1);
```

## contd..

- Code for process  $P_0$

```
send(&number, P1);
```

- Code for process  $P_{p-1}$

```
recv(&number, Pp-2);  
accumulation = accumulation + number;
```

## contd..

```
if (process > 0) {  
    recv(&accumulation, Pi-1);  
    accumulation = accumulation + number;  
}  
if (process < p-1) send(&accumulation, Pi+1);
```

# Analysis for Type 1

$t_{\text{total}} = (\text{time for one pipeline cycle})(\text{number of cycles})$

$$t_{\text{total}} = (t_{\text{comp}} + t_{\text{comm}})(m + p - 1)$$

where there are  $m$  instances of a problem, and  $p$  pipeline stages.

Average time for computation is

$$t_a = \frac{t_{\text{total}}}{m}$$

## contd..

- Single instance of a problem

$$t_{\text{comp}} = 1$$

$$t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}})$$

$$t_{\text{cycle}} = 2(t_{\text{startup}} + t_{\text{data}}) + 1$$

$$t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + 1)n$$

# contd..

- Multiple instance of a problem  
we have  $m$  groups of  $n$  numbers to add,  
resulting in a separate result

$$t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + 1)(m + n - 1)$$

For large  $m$ , the average execution time is approximately

$$t_a = \frac{t_{\text{total}}}{m} \approx 2(t_{\text{startup}} + t_{\text{data}}) + 1$$

## contd..

- Data partitioning with multiple instances of a problem

$$t_{\text{comp}} = d$$

$$t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}})$$

$$t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + d)(m + n/d - 1)$$

## Type 2 - Example

- Prime Number Generation (sieve of Erathosthenes)

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

# contd..

- Sequential code

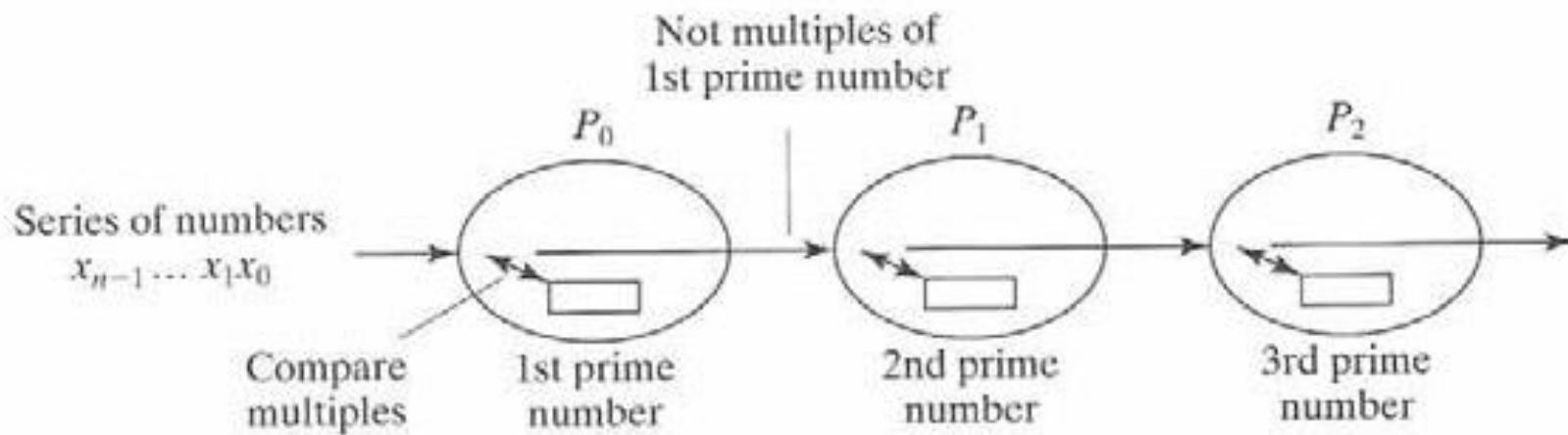
```
for (i = 2; i <= n; i++)
    prime[i] = 1;                                /* Initialize array */
for (i = 2; i <= sqrt_n; i++)                  /* for each number */
    if (prime[i] == 1)                          /* identified as prime */
        for (j = i + i; j <= n; j = j + i)    /* strike out all multiples */
            prime[j] = 0;                      /* includes already done */
```

$$t_s = \left\lfloor \frac{n}{2} - 1 \right\rfloor + \left\lfloor \frac{n}{3} - 1 \right\rfloor + \left\lfloor \frac{n}{5} - 1 \right\rfloor + \dots + \left\lfloor \frac{n}{\sqrt{n}} - 1 \right\rfloor$$

$$O(n^2).$$

# contd..

- Parallel version



# contd..

- Code for process Pi

```
recv(&x, Pi-1);
/* repeat following for each number */
recv(&number, Pi-1);
if ((number % x) != 0) send(&number, Pi+1);
```

## contd..

- Rewritten code (using terminator)

```
recv(&x, Pi-1);
for (i = 0; i < n; i++) {
    recv(&number, Pi-1);
    if (number == terminator) break;
    if (number % x) != 0) send(&number, Pi+1);
}
```

# Analysis

$$t_{\text{comp}} = 1$$

$$t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}})$$

$$t_{\text{total}} = t_{\text{cycle}} * \text{no.of cycles}$$

$$t_{\text{cycle}} = 2(t_{\text{startup}} + t_{\text{data}}) + 1$$

$$\text{no. of cycles} = (m+n-1) = n + \sqrt{n} - 1$$

$$t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + 1)(n + \sqrt{n} - 1)$$

# Type 3 - Example

- Solving system of linear equations

Consider the upper triangular form of the linear equations:

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

⋮

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 = b_1$$

$$a_{0,0}x_0 = b_0$$

a's and b's are constants and x is the unknown

contd..

- First find  $x_0$

$$x_0 = \frac{b_0}{a_{0,0}}$$

- Substitute  $x_0$  and find  $x_1$

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

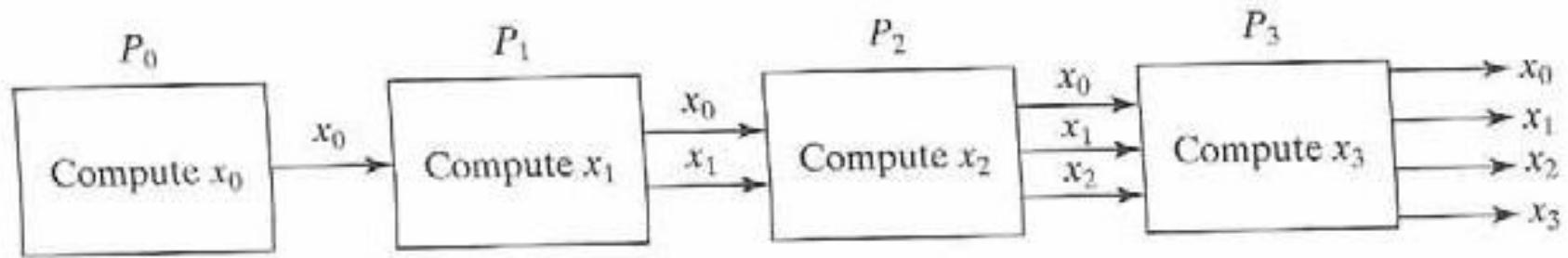
## contd..

- Now, substitute  $x_0$  and  $x_1$  to obtain  $x_2$  and so on..

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

- The first stage of pipelining computes  $x_0$  and passes on to second stage
- Second stage computes  $x_1$  from  $x_0$  and passes both  $x_1$  and  $x_0$  to next level and so on..

# contd..



- Each stage is implemented with one processor
- Assume there are  $n$  processors for  $n$  equations ( $p=n$ )

## contd..

- The  $i^{\text{th}}$  process ( $0 < i < p$ ) receives the values  $x_0, x_1, x_2, \dots, x_{i-1}$  and computes  $x_i$  from the equation

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j}x_j}{a_{i,i}}$$

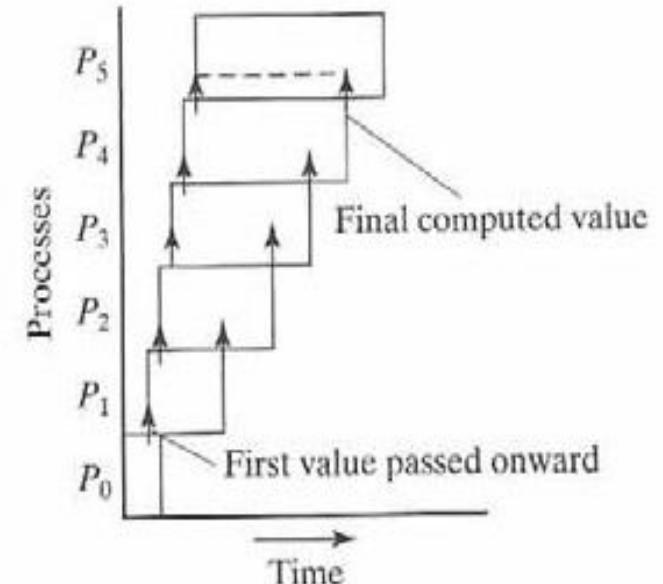
# Sequential code

```
x[0] = b[0]/a[0][0];                      /* x[0] computed separately */  
for (i = 1; i < n; i++) {                  /* for remaining unknowns */  
    sum = 0;  
    for (j = 0; j < i; j++)  
        sum = sum + a[i][j]*x[j];  
    x[i] = (b[i] - sum)/a[i][i];  
}
```

# Parallel code

- Algorithm for  $P_i$  ( $1 < i < p$ ) is given as

```
for (j = 0; j < i; j++) {  
    recv(&x[j], P_{i-1});  
    send(&x[j], P_{i+1});  
}  
sum = 0;  
for (j = 0; j < i; j++)  
    sum = sum + a[i][j]*x[j];  
x[i] = (b[i] - sum) / a[i][i];  
send(&x[i], P_{i+1});
```



## contd..

- The modified code is

```
sum = 0;  
for (j = 0; j < i; j++) {  
    recv(&x[j], Pi-1);  
    send(&x[j], Pj+1);  
    sum = sum + a[i][j]*x[j];  
}  
x[i] = (b[i] - sum)/a[i][i];  
send(&x[i], Pi+1);
```

# Analysis

	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$
divide					
$\text{send}(x_0) \Rightarrow$	$\text{recv}(x_0)$				
end	$\text{send}(x_0) \Rightarrow$	$\text{recv}(x_0)$			
	multiply/add	$\text{send}(x_0) \Rightarrow$	$\text{recv}(x_0)$		
	divide/subtract	$\text{multiply/add}$	$\text{send}(x_0) \Rightarrow$	$\text{recv}(x_0)$	
	$\text{send}(x_1) \Rightarrow$	$\text{recv}(x_1)$	$\text{multiply/add}$	$\text{send}(x_1) \Rightarrow$	$\text{recv}(x_0)$
Time	end	$\text{send}(x_1) \Rightarrow$	$\text{recv}(x_1)$	$\text{multiply/add}$	$\text{send}(x_1) \Rightarrow$
		$\text{multiply/add}$	$\text{send}(x_1) \Rightarrow$	$\text{recv}(x_1)$	$\text{multiply/add}$
		divide/subtract	$\text{recv}(x_1)$	$\text{send}(x_1) \Rightarrow$	$\text{recv}(x_1)$
		$\text{send}(x_2) \Rightarrow$	$\text{multiply/add}$	$\text{recv}(x_1)$	$\text{send}(x_1) \Rightarrow$
		end	$\text{send}(x_2) \Rightarrow$	$\text{recv}(x_2)$	$\text{multiply/add}$
			$\text{multiply/add}$	$\text{send}(x_2) \Rightarrow$	$\text{recv}(x_2)$
			divide/subtract	$\text{recv}(x_2)$	$\text{send}(x_2) \Rightarrow$
			$\text{send}(x_3) \Rightarrow$	$\text{multiply/add}$	$\text{multiply/add}$
			end	$\text{recv}(x_3)$	$\text{recv}(x_3)$
				$\text{send}(x_3) \Rightarrow$	$\text{send}(x_3) \Rightarrow$
				$\text{multiply/add}$	$\text{multiply/add}$
				divide/subtract	$\text{divide/subtract}$
				$\text{send}(x_4) \Rightarrow$	$\text{send}(x_4) \Rightarrow$
				end	end

## contd..

- First process performs one divide and one send
- Each of the  $i^{\text{th}}$  process performs  $i$  `recvs()` and  $i$  `sends()`,  $i$  multiply/add, one divide /subtract and final one `send()` , Total of  $2i+1$  communication times and  $2i+2$  computation steps
- Last process performs,  $p-1$  `recvs()`,  $p-1$  multiply/add, one divide/subtract, Total of  $2p$  computation steps and  $p-1$  communication times

## contd..

- Final  $t_{cycle}$  includes: time taken for the execution of the statements by the last process, plus,  $p-1$  send()s plus one divide.
- Overall complexity is  $O(n)$   
where  $n=p$

# Exercises

Write a parallel program to compute  $x^{16}$  using a pipeline approach. Repeat by applying a divide-and-conquer approach. Compare the two methods analytically and experimentally.

Develop a pipeline solution to compute  $\sin\theta$  according to

$$\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \frac{\theta^9}{9!} - \dots$$

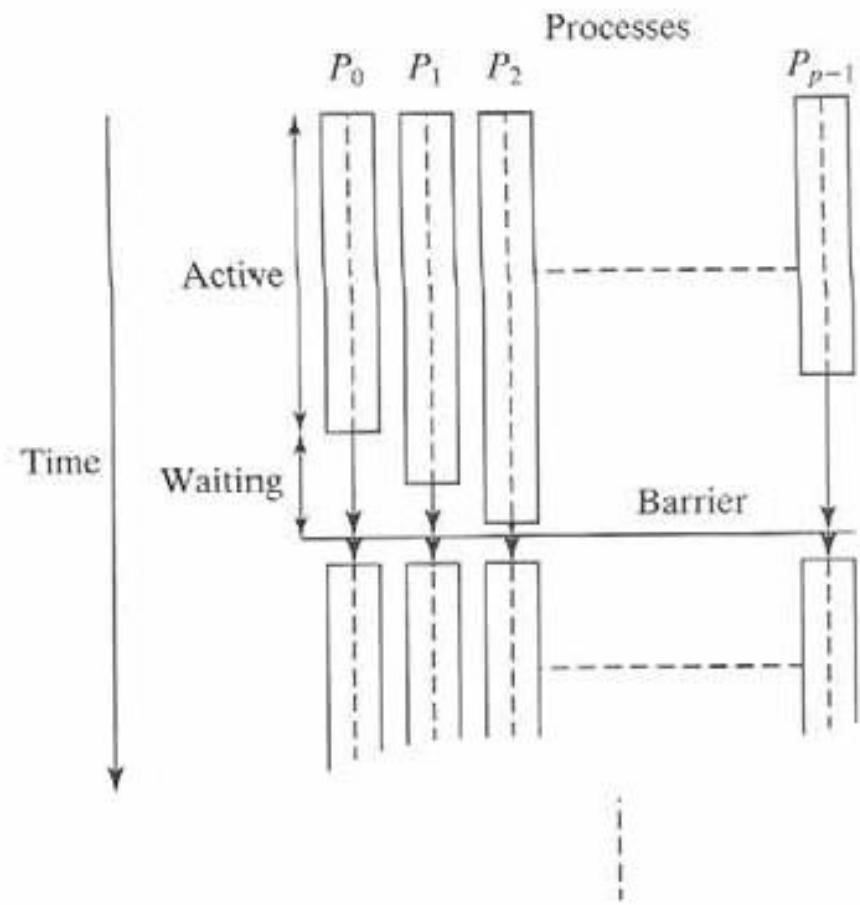
A series of values are input,  $\theta_0, \theta_1, \theta_2, \theta_3, \dots$ .

# 6. Synchronous Computations

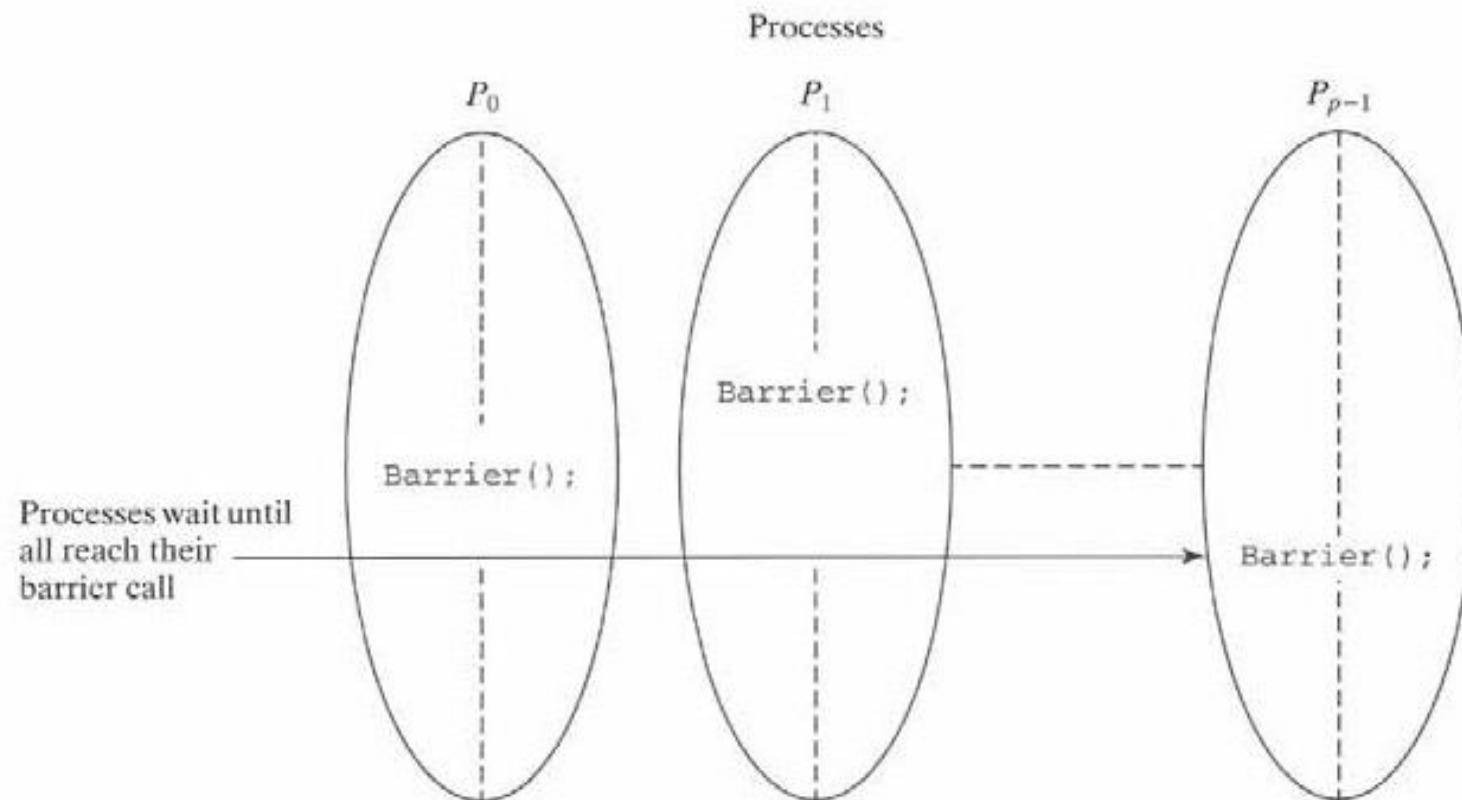
## Barrier

- It is a mechanism that prevents any process from continuing past a specified point until all the processes are ready.
- It helps in synchronizing the processes
- It is applied in both shared memory and message passing systems

# contd..

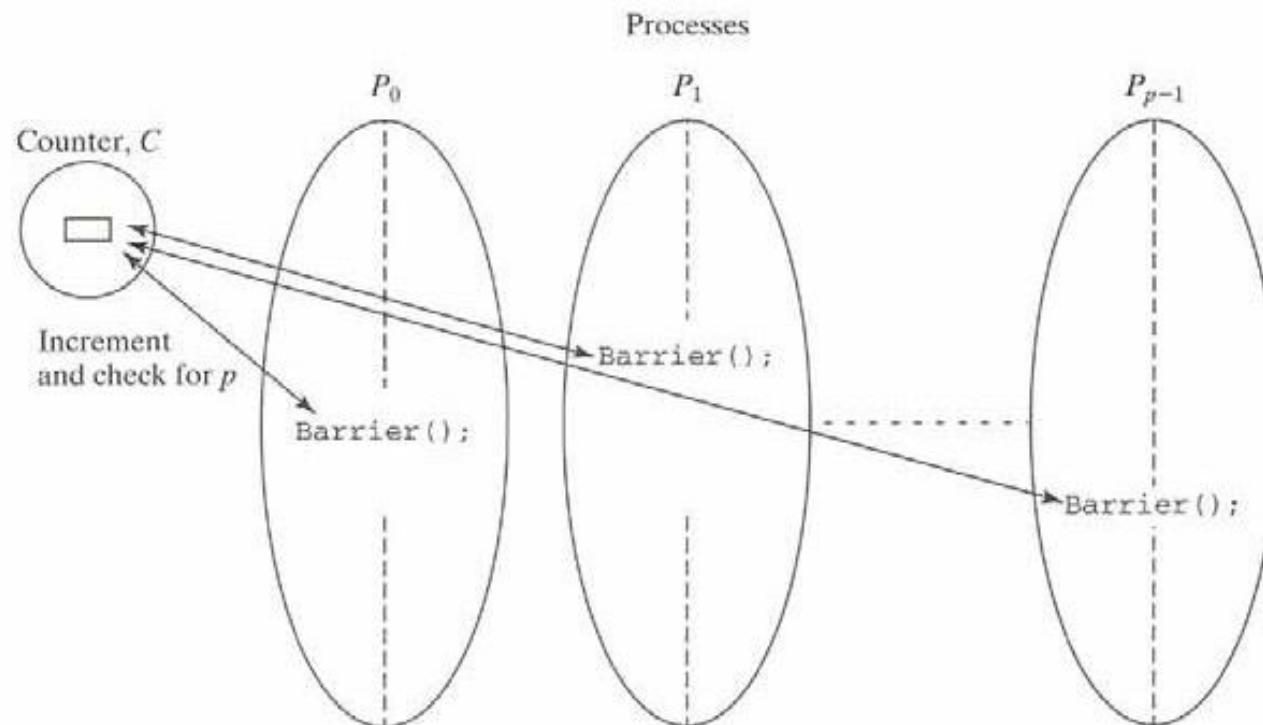


# contd..



# Barrier Implementations

- Counter or Linear implementation
  - Centralized implementation



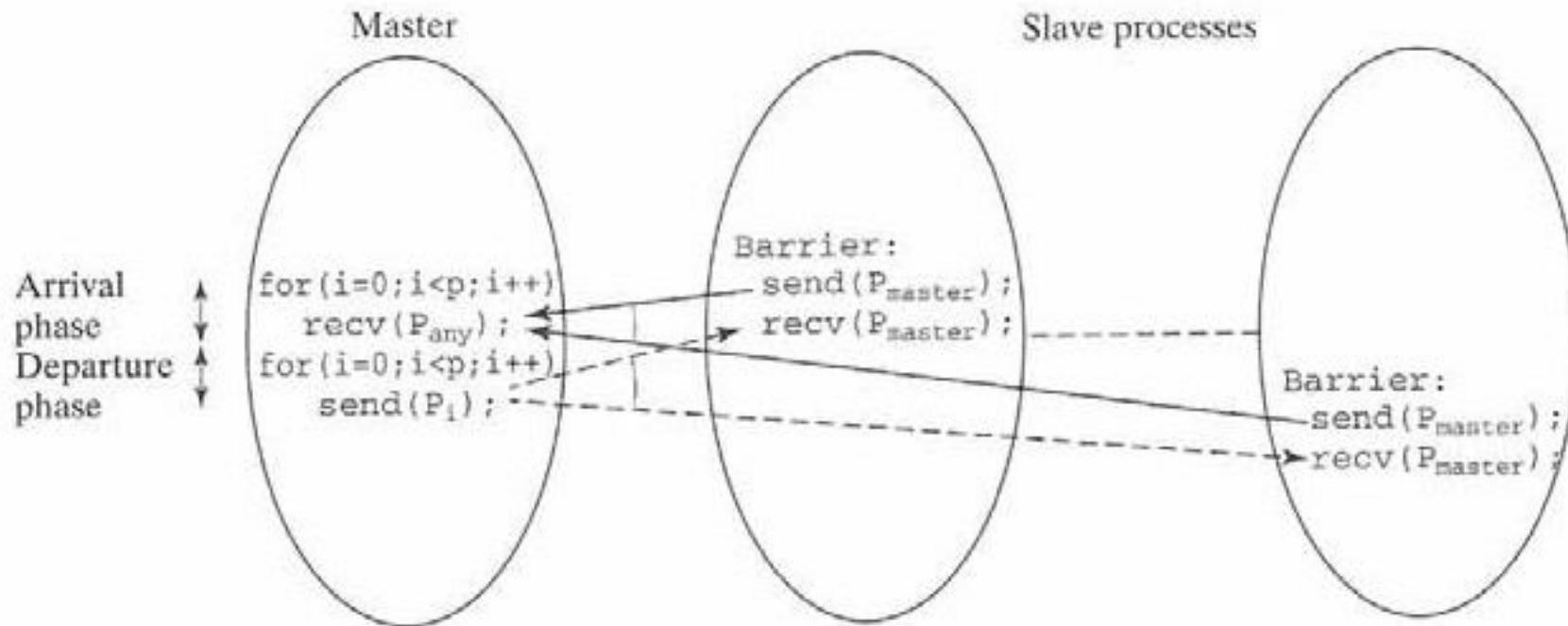
# contd..

- Counter implementation

```
for (i = 0; i < p; i++)      /* count slaves as they reach their barrier */
    recv(Pany);
for (i = 0; i < p; i++)      /* release slaves */
    send(Pi);
```

# contd..

- Implementation – arrival phase and departure phase



- Arrival phase can also be implemented using a **gather** routine and departure phase by a **broadcast** routine
- Complexity –  $O(P)$

# contd..

- Tree implementation
  - Efficient method of barrier implementation
  - Complexity -  $O(2\log P) \approx O(\log P)$

First stage:  $P_1$  sends message to  $P_0$  (when  $P_1$  reaches its barrier)

$P_3$  sends message to  $P_2$  (when  $P_3$  reaches its barrier)

$P_5$  sends message to  $P_4$  (when  $P_5$  reaches its barrier)

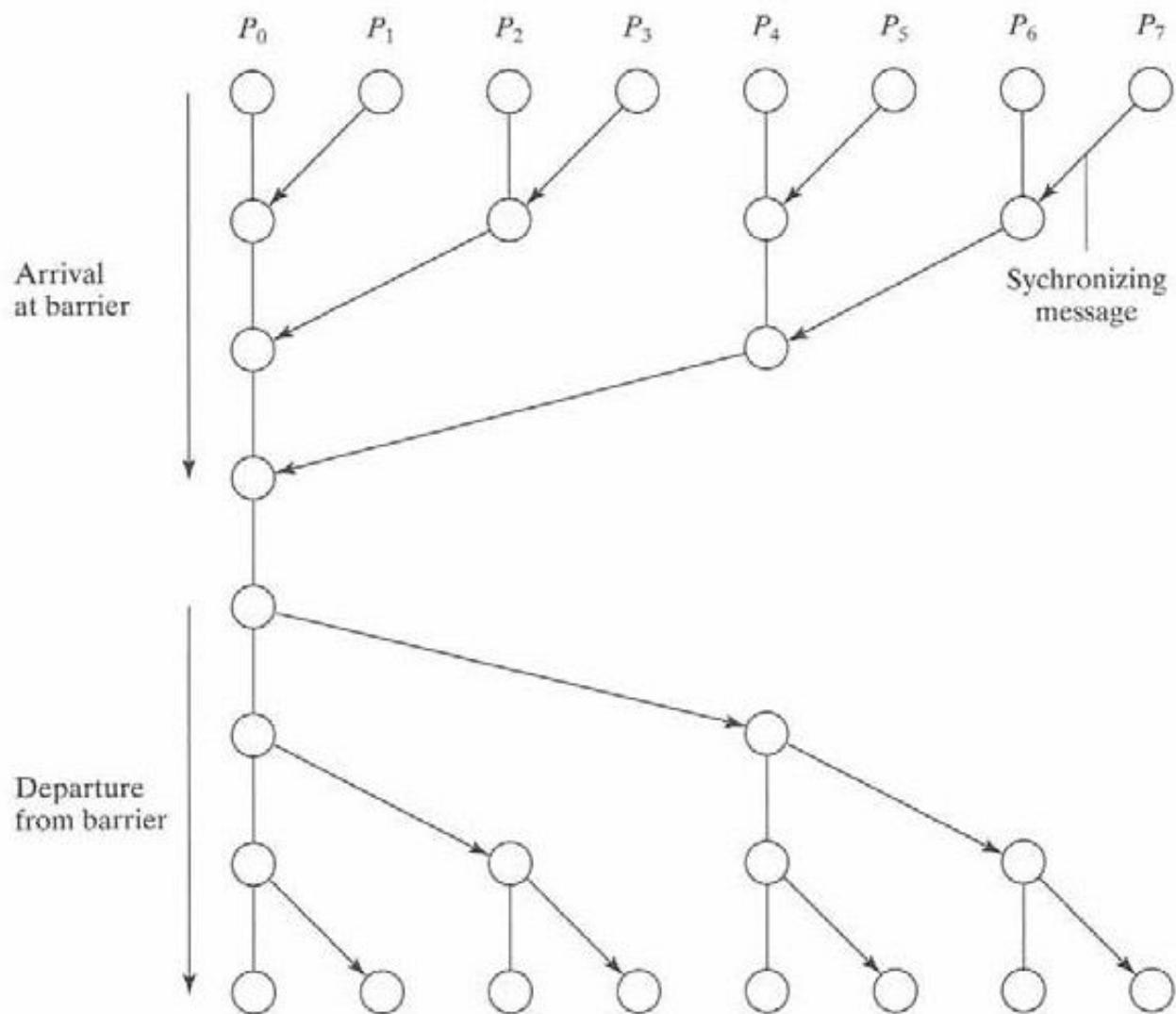
$P_7$  sends message to  $P_6$  (when  $P_7$  reaches its barrier)

Second stage:  $P_2$  sends message to  $P_0$  ( $P_2$  and  $P_3$  have reached their barrier)

$P_6$  sends message to  $P_4$  ( $P_6$  and  $P_7$  have reached their barrier)

Third stage:  $P_4$  sends message to  $P_0$  ( $P_4, P_5, P_6$ , and  $P_7$  have reached their barrier)  
 $P_0$  terminates arrival phase (when  $P_0$  reaches barrier and has received message from  $P_4$ )

# contd..



## contd..

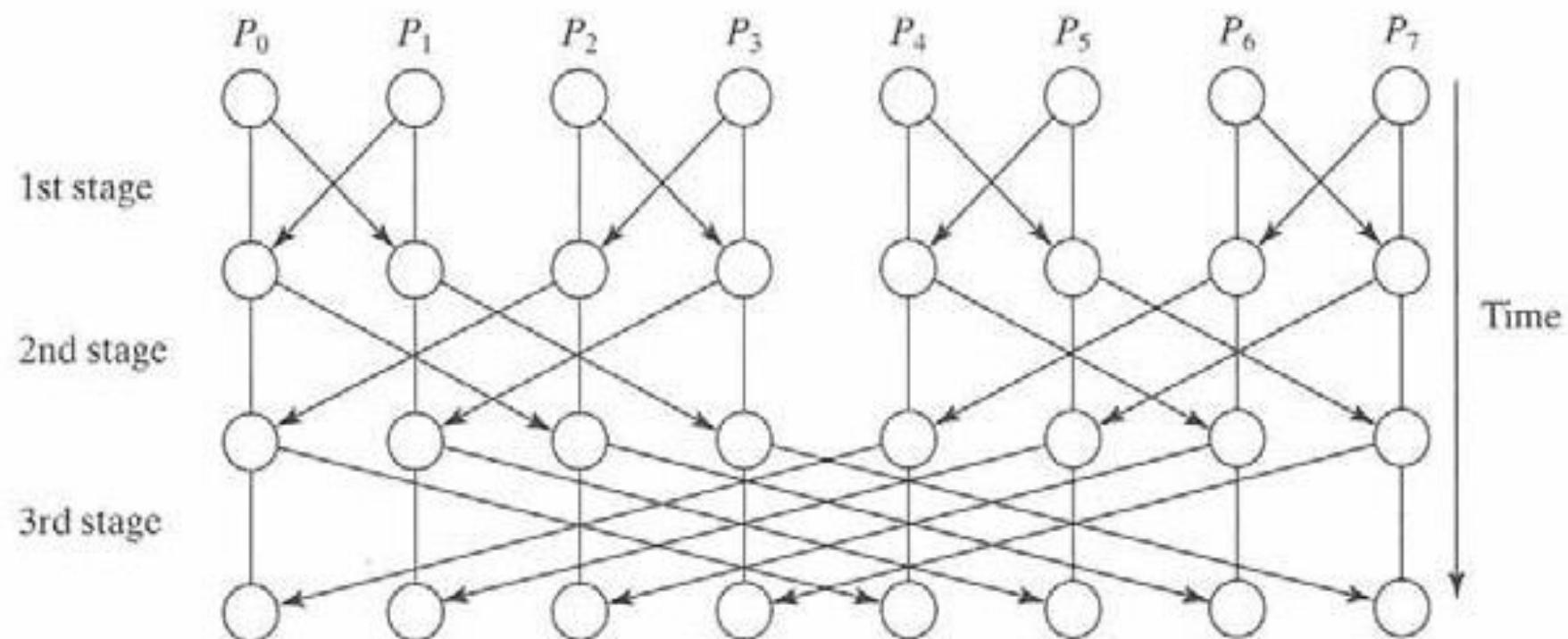
- Butterfly barrier
  - Variation of tree implementation
  - Complexity –  $O(\log P)$

First stage       $P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$

Second stage     $P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$

Third stage      $P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$

contd..



# Example: Data Parallel Computations

- Data Parallel computation: same operation is performed on different data elements simultaneously (SIMD model)

```
for (i = 0; i < n; i++)
    a[i] = a[i] + k;
```

# contd..

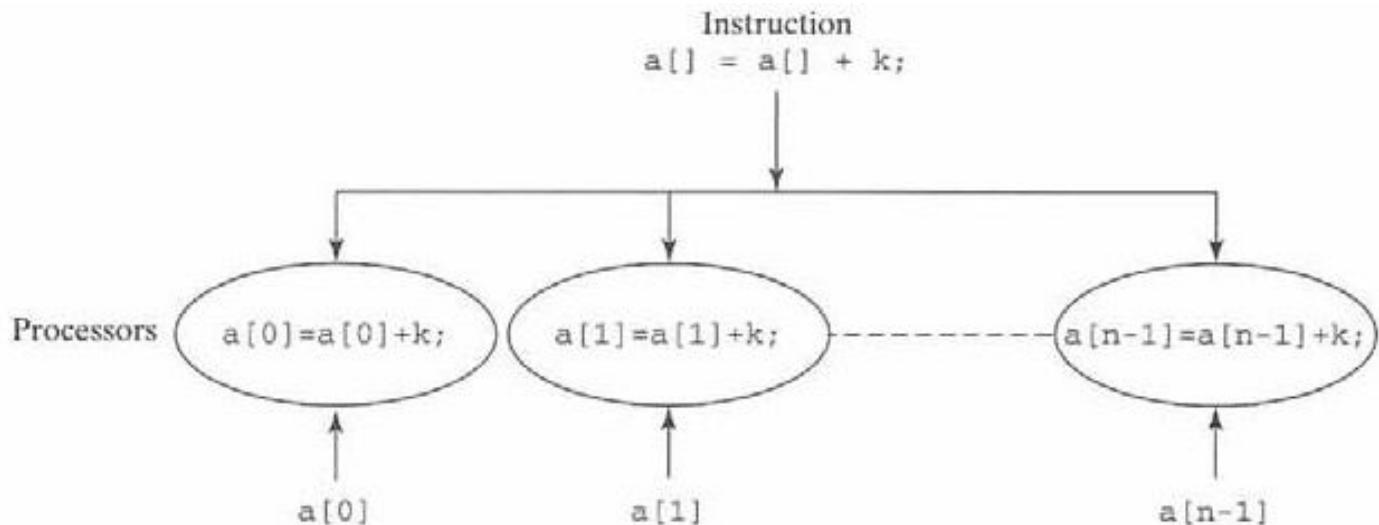


Figure 6.7 Data parallel computation.

- A special construct in parallel programming helps to perform this (**forall**)

```
forall (i = 0; i < n; i++)
    a[i] = a[i] + k;
```

# Example: Prefix Sum problem

- Given a list of numbers,  $x_0, x_1, \dots x_{n-1}$ , all the partial summations are computed in the prefix sum problem.

(i.e)  $x_0, x_0+x_1, x_0+x_1+x_2, x_0+x_1+x_2+x_3, \dots$ )

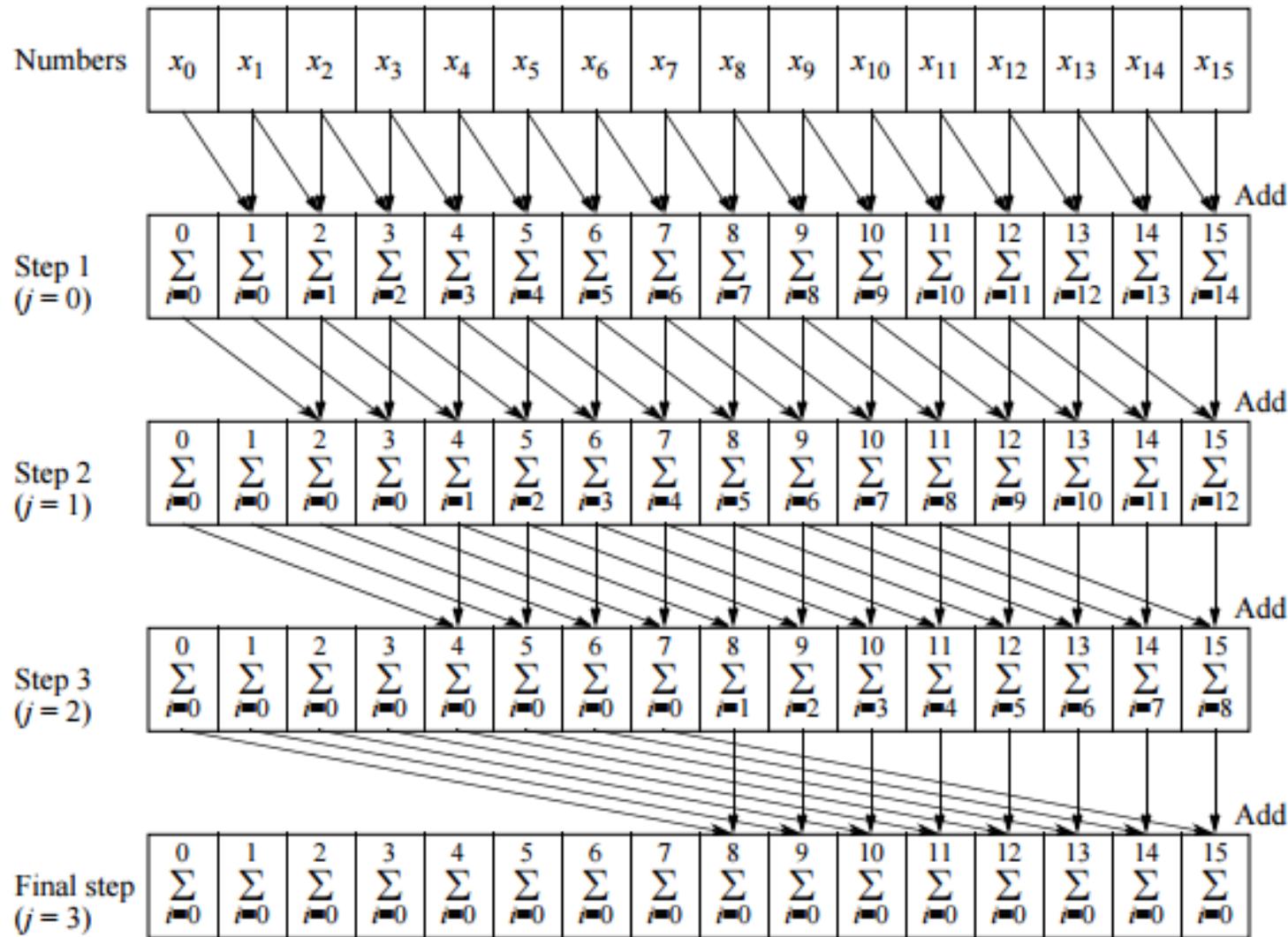
- The sequential code is

```
sum[0] = x[0];
for (i = 1; i < n; i++)
    sum[i] = sum[i-1] + x[i];
```

- Complexity –  $O(n)$

contd..

- Data Parallel Method



## contd..

- Sequential version (using data parallel approach)

```
for (j = 0; j < log(n); j++)          /* at each step */  
    for (i = 2j; i < n; i++)        /* add to accumulating sum */  
        x[i] = x[i] + x[i - 2j];
```

- Parallel version (for SIMD computers)

```
for (j = 0; j < log(n); j++)          /* at each step */  
    forall (i = 0; i < n; i++)        /* add to accumulating sum */  
        if (i >= 2j) x[i] = x[i] + x[i - 2j];
```

- Complexity – O(log n) – both computations and communications

# Chapter 9: Distributed Mutual Exclusion Algorithms

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

# Introduction

- Mutual exclusion: Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner.
- Only one process is allowed to execute the critical section (CS) at any given time.
- In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion.

# Introduction

- Distributed mutual exclusion algorithms must deal with unpredictable message delays and incomplete knowledge of the system state.
- Three basic approaches for distributed mutual exclusion:
  - ① Token based approach
  - ② Non-token based approach
  - ③ Quorum based approach
- Token-based approach:
  - ▶ A unique token is shared among the sites.
  - ▶ A site is allowed to enter its CS if it possesses the token.
  - ▶ Mutual exclusion is ensured because the token is unique.

# Introduction

- Non-token based approach:
  - ▶ Two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next.
- Quorum based approach:
  - ▶ Each site requests permission to execute the CS from a subset of sites (called a quorum).
  - ▶ Any two quorums contain a common site.
  - ▶ This common site is responsible to make sure that only one request executes the CS at any time.

# Preliminaries

## System Model

- The system consists of  $N$  sites,  $S_1, S_2, \dots, S_N$ .
- We assume that a single process is running on each site. The process at site  $S_i$  is denoted by  $p_i$ .
- A site can be in one of the following three states: requesting the CS, executing the CS, or neither requesting nor executing the CS (i.e., idle).
- In the 'requesting the CS' state, the site is blocked and can not make further requests for the CS. In the 'idle' state, the site is executing outside the CS.
- In token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS (called the *idle token* state).
- At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.

# Requirements

## Requirements of Mutual Exclusion Algorithms

- ① **Safety Property:** At any instant, only one process can execute the critical section.
- ② **Liveness Property:** This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.
- ③ **Fairness:** Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.

# Performance Metrics

The performance is generally measured by the following four metrics:

- **Message complexity:** The number of messages required per CS execution by a site.
- **Synchronization delay:** After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 1).

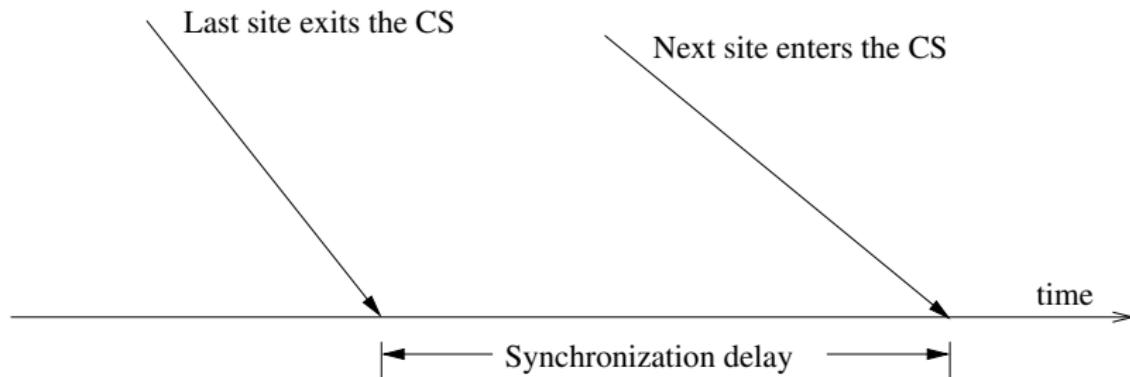


Figure 1: Synchronization Delay.

# Performance Metrics

- **Response time:** The time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 2).

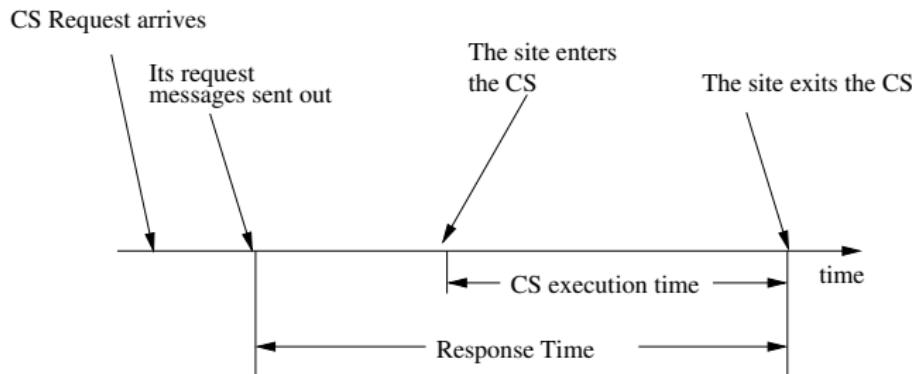


Figure 2: Response Time.

- **System throughput:** The rate at which the system executes requests for the CS.

$$\text{system throughput} = 1/(SD+E)$$

where  $SD$  is the synchronization delay and  $E$  is the average critical section execution time.

# Performance Metrics

## Low and High Load Performance:

- We often study the performance of mutual exclusion algorithms under two special loading conditions, viz., “low load” and “high load”.
- The load is determined by the arrival rate of CS execution requests.
- Under *low load* conditions, there is seldom more than one request for the critical section present in the system simultaneously.
- Under *heavy load* conditions, there is always a pending request for critical section at a site.

# Lamport's Algorithm

- Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
- Every site  $S_i$  keeps a queue,  $request\_queue_i$ , which contains mutual exclusion requests ordered by their timestamps.
- This algorithm requires communication channels to deliver messages the FIFO order.

# The Algorithm

## Requesting the critical section:

- When a site  $S_i$  wants to enter the CS, it broadcasts a  $\text{REQUEST}(ts_i, i)$  message to all other sites and places the request on  $request\_queue_i$ . ( $(ts_i, i)$  denotes the timestamp of the request.)
- When a site  $S_j$  receives the  $\text{REQUEST}(ts_i, i)$  message from site  $S_i$ , places site  $S_i$ 's request on  $request\_queue_j$  and it returns a timestamped REPLY message to  $S_i$ .

**Executing the critical section:** Site  $S_i$  enters the CS when the following two conditions hold:

- L1:  $S_i$  has received a message with timestamp larger than  $(ts_i, i)$  from all other sites.
- L2:  $S_i$ 's request is at the top of  $request\_queue_i$ .

# The Algorithm

## Releasing the critical section:

- Site  $S_i$ , upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
- When a site  $S_j$  receives a RELEASE message from site  $S_i$ , it removes  $S_i$ 's request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS.

## correctness

**Theorem: Lamport's algorithm achieves mutual exclusion.**

**Proof:**

- Proof is by contradiction. Suppose two sites  $S_i$  and  $S_j$  are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites *concurrently*.
- This implies that at some instant in time, say  $t$ , both  $S_i$  and  $S_j$  have their own requests at the top of their *request\_queues* and condition L1 holds at them. Without loss of generality, assume that  $S_i$ 's request has smaller timestamp than the request of  $S_j$ .
- From condition L1 and FIFO property of the communication channels, it is clear that at instant  $t$  the request of  $S_i$  must be present in *request\_queue<sub>j</sub>* when  $S_j$  was executing its CS. This implies that  $S_j$ 's own request is at the top of its own *request\_queue* when a smaller timestamp request,  $S_i$ 's request, is present in the *request\_queue<sub>j</sub>* – a contradiction!

# correctness

**Theorem: Lamport's algorithm is fair.**

**Proof:**

- The proof is by contradiction. Suppose a site  $S_i$ 's request has a smaller timestamp than the request of another site  $S_j$  and  $S_j$  is able to execute the CS before  $S_i$ .
- For  $S_j$  to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say  $t$ ,  $S_j$  has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.
- But *request\\_queue* at a site is ordered by timestamp, and according to our assumption  $S_i$  has lower timestamp. So  $S_i$ 's request must be placed ahead of the  $S_j$ 's request in the *request\\_queue<sub>j</sub>*. This is a contradiction!

# Performance

- For each CS execution, Lamport's algorithm requires  $(N - 1)$  REQUEST messages,  $(N - 1)$  REPLY messages, and  $(N - 1)$  RELEASE messages.
- Thus, Lamport's algorithm requires  $3(N - 1)$  messages per CS invocation.
- Synchronization delay in the algorithm is  $T$ .

# An optimization

- In Lamport's algorithm, REPLY messages can be omitted in certain situations. For example, if site  $S_j$  receives a REQUEST message from site  $S_i$  after it has sent its own REQUEST message with timestamp higher than the timestamp of site  $S_i$ 's request, then site  $S_j$  need not send a REPLY message to site  $S_i$ .
- This is because when site  $S_i$  receives site  $S_j$ 's request with timestamp higher than its own, it can conclude that site  $S_j$  does not have any smaller timestamp request which is still pending.
- With this optimization, Lamport's algorithm requires between  $3(N - 1)$  and  $2(N - 1)$  messages per CS execution.

# Ricart-Agrawala Algorithm

- The Ricart-Agrawala algorithm assumes the communication channels are FIFO. The algorithm uses two types of messages: REQUEST and REPLY.
- A process sends a REQUEST message to all other processes to request their permission to enter the critical section. A process sends a REPLY message to a process to give its permission to that process.
- Processes use Lamport-style logical clocks to assign a timestamp to critical section requests and timestamps are used to decide the priority of requests.
- Each process  $p_i$  maintains the Request-Deferred array,  $RD_i$ , the size of which is the same as the number of processes in the system.
- Initially,  $\forall i \forall j: RD_i[j]=0$ . Whenever  $p_i$  defer the request sent by  $p_j$ , it sets  $RD_i[j]=1$  and after it has sent a REPLY message to  $p_j$ , it sets  $RD_i[j]=0$ .

# Description of the Algorithm

## Requesting the critical section:

- (a) When a site  $S_i$  wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites.
- (b) When site  $S_j$  receives a REQUEST message from site  $S_i$ , it sends a REPLY message to site  $S_i$  if site  $S_j$  is neither requesting nor executing the CS, or if the site  $S_j$  is requesting and  $S_i$ 's request's timestamp is smaller than site  $S_j$ 's own request's timestamp. Otherwise, the reply is deferred and  $S_j$  sets  $RD_j[i]=1$

## Executing the critical section:

- (c) Site  $S_i$  enters the CS after it has received a REPLY message from every site it sent a REQUEST message to.

# Algorithm

## Releasing the critical section:

- (d) When site  $S_i$  exits the CS, it sends all the deferred REPLY messages:  $\forall j$  if  $RD_i[j]=1$ , then send a REPLY message to  $S_j$  and set  $RD_i[j]=0$ .

## Notes:

- When a site receives a message, it updates its clock using the timestamp in the message.
- When a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request.

# Correctness

**Theorem: Ricart-Agrawala algorithm achieves mutual exclusion.**

**Proof:**

- Proof is by contradiction. Suppose two sites  $S_i$  and  $S_j$  are executing the CS concurrently and  $S_i$ 's request has higher priority than the request of  $S_j$ . Clearly,  $S_i$  received  $S_j$ 's request after it has made its own request.
- Thus,  $S_j$  can concurrently execute the CS with  $S_i$  only if  $S_i$  returns a REPLY to  $S_j$  (in response to  $S_j$ 's request) before  $S_i$  exits the CS.
- However, this is impossible because  $S_j$ 's request has lower priority. Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.

# Performance

- For each CS execution, Ricart-Agrawala algorithm requires  $(N - 1)$  REQUEST messages and  $(N - 1)$  REPLY messages.
- Thus, it requires  $2(N - 1)$  messages per CS execution.
- Synchronization delay in the algorithm is  $T$ .

# Quorum-Based Mutual Exclusion Algorithms

Quorum-based mutual exclusion algorithms are different in the following two ways:

- ➊ A site does not request permission from all other sites, but only from a subset of the sites. The request set of sites are chosen such that  $\forall i \forall j : 1 \leq i, j \leq N :: R_i \cap R_j \neq \emptyset$ . Consequently, every pair of sites has a site which mediates conflicts between that pair.
- ➋ A site can send out only one REPLY message at any time. A site can send a REPLY message only after it has received a RELEASE message for the previous REPLY message.

## continuation..

Since these algorithms are based on the notion of ‘Coteries’ and ‘Quorums’, we next describe the idea of coteries and quorums.

A coterie  $C$  is defined as a set of sets, where each set  $g \in C$  is called a quorum. The following properties hold for quorums in a coterie:

- **Intersection property:** For every quorum  $g, h \in C$ ,  $g \cap h \neq \emptyset$ .  
For example, sets  $\{1,2,3\}$ ,  $\{2,5,7\}$  and  $\{5,7,9\}$  cannot be quorums in a coterie because the first and third sets do not have a common element.
- **Minimality property:** There should be no quorums  $g, h$  in coterie  $C$  such that  $g \supseteq h$ . For example, sets  $\{1,2,3\}$  and  $\{1,3\}$  cannot be quorums in a coterie because the first set is a superset of the second.

## continuation..

Coteries and quorums can be used to develop algorithms to ensure mutual exclusion in a distributed environment. A simple protocol works as follows:

- Let 'a' is a site in quorum 'A'. If 'a' wants to invoke mutual exclusion, it requests permission from all sites in its quorum 'A'.
- Every site does the same to invoke mutual exclusion. Due to the Intersection Property, quorum 'A' contains at least one site that is common to the quorum of every other site.
- These common sites send permission to only one site at any time. Thus, mutual exclusion is guaranteed.

Note that the Minimality property ensures efficiency rather than correctness.

# Maekawa's Algorithm

Maekawa's algorithm was the first quorum-based mutual exclusion algorithm. The request sets for sites (i.e., quorums) in Maekawa's algorithm are constructed to satisfy the following conditions:

- M1:  $(\forall i \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \emptyset)$
- M2:  $(\forall i : 1 \leq i \leq N :: S_i \in R_i)$
- M3:  $(\forall i : 1 \leq i \leq N :: |R_i| = K)$
- M4: Any site  $S_j$  is contained in  $K$  number of  $R_i$ s,  $1 \leq i, j \leq N$ .

Maekawa used the theory of projective planes and showed that  $N = K(K - 1) + 1$ . This relation gives  $|R_i| = \sqrt{N}$ .

## continuation..

- Conditions M1 and M2 are necessary for correctness; whereas conditions M3 and M4 provide other desirable features to the algorithm.
- Condition M3 states that the size of the requests sets of all sites must be equal implying that all sites should have to do equal amount of work to invoke mutual exclusion.
- Condition M4 enforces that exactly the same number of sites should request permission from any site implying that all sites have “equal responsibility” in granting permission to other sites.

# The Algorithm

A site  $S_i$  executes the following steps to execute the CS.

## Requesting the critical section

- (a) A site  $S_i$  requests access to the CS by sending REQUEST( $i$ ) messages to all sites in its request set  $R_i$ .
- (b) When a site  $S_j$  receives the REQUEST( $i$ ) message, it sends a REPLY( $j$ ) message to  $S_i$  provided it hasn't sent a REPLY message to a site since its receipt of the last RELEASE message. Otherwise, it queues up the REQUEST( $i$ ) for later consideration.

## Executing the critical section

- (c) Site  $S_i$  executes the CS only after it has received a REPLY message from every site in  $R_i$ .

# The Algorithm

## Releasing the critical section

- (d) After the execution of the CS is over, site  $S_i$  sends a RELEASE( $i$ ) message to every site in  $R_i$ .
- (e) When a site  $S_j$  receives a RELEASE( $i$ ) message from site  $S_i$ , it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

# Correctness

**Theorem:** Maekawa's algorithm achieves mutual exclusion.

**Proof:**

- Proof is by contradiction. Suppose two sites  $S_i$  and  $S_j$  are concurrently executing the CS.
- This means site  $S_i$  received a REPLY message from all sites in  $R_i$  and concurrently site  $S_j$  was able to receive a REPLY message from all sites in  $R_j$ .
- If  $R_i \cap R_j = \{S_k\}$ , then site  $S_k$  must have sent REPLY messages to both  $S_i$  and  $S_j$  concurrently, which is a contradiction.  $\square$

# Performance

- Since the size of a request set is  $\sqrt{N}$ , an execution of the CS requires  $\sqrt{N}$  REQUEST,  $\sqrt{N}$  REPLY, and  $\sqrt{N}$  RELEASE messages, resulting in  $3\sqrt{N}$  messages per CS execution.
- Synchronization delay in this algorithm is  $2T$ . This is because after a site  $S_i$  exits the CS, it first releases all the sites in  $R_i$  and then one of those sites sends a REPLY message to the next site that executes the CS.

# Problem of Deadlocks

- Maekawa's algorithm can deadlock because a site is exclusively locked by other sites and requests are not prioritized by their timestamps.
- Assume three sites  $S_i$ ,  $S_j$ , and  $S_k$  simultaneously invoke mutual exclusion.
- Suppose  $R_i \cap R_j = \{S_{ij}\}$ ,  $R_j \cap R_k = \{S_{jk}\}$ , and  $R_k \cap R_i = \{S_{ki}\}$ .
- Consider the following scenario:
  - ▶  $S_{ij}$  has been locked by  $S_i$  (forcing  $S_j$  to wait at  $S_{ij}$ ).
  - ▶  $S_{jk}$  has been locked by  $S_j$  (forcing  $S_k$  to wait at  $S_{jk}$ ).
  - ▶  $S_{ki}$  has been locked by  $S_k$  (forcing  $S_i$  to wait at  $S_{ki}$ ).
- This state represents a deadlock involving sites  $S_i$ ,  $S_j$ , and  $S_k$ .

# Handling Deadlocks

- Maekawa's algorithm handles deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock.
- A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request arrives and waits at a site because the site has sent a REPLY message to a lower priority request.

Deadlock handling requires three types of messages:

**FAILED:** A FAILED message from site  $S_i$  to site  $S_j$  indicates that  $S_i$  can not grant  $S_j$ 's request because it has currently granted permission to a site with a higher priority request.

**INQUIRE:** An INQUIRE message from  $S_i$  to  $S_j$  indicates that  $S_i$  would like to find out from  $S_j$  if it has succeeded in locking all the sites in its request set.

**YIELD:** A YIELD message from site  $S_i$  to  $S_j$  indicates that  $S_i$  is returning the permission to  $S_j$  (to yield to a higher priority request at  $S_j$ ).

# Handling Deadlocks

Maekawa's algorithm handles deadlocks as follows:

- When a  $\text{REQUEST}(ts, i)$  from site  $S_j$  blocks at site  $S_j$  because  $S_j$  has currently granted permission to site  $S_k$ , then  $S_j$  sends a  $\text{FAILED}(j)$  message to  $S_i$  if  $S_i$ 's request has lower priority. Otherwise,  $S_j$  sends an  $\text{INQUIRE}(j)$  message to site  $S_k$ .
- In response to an  $\text{INQUIRE}(j)$  message from site  $S_j$ , site  $S_k$  sends a  $\text{YIELD}(k)$  message to  $S_j$  provided  $S_k$  has received a  $\text{FAILED}$  message from a site in its request set or if it sent a  $\text{YIELD}$  to any of these sites, but has not received a new  $\text{GRANT}$  from it.
- In response to a  $\text{YIELD}(k)$  message from site  $S_k$ , site  $S_j$  assumes as if it has been released by  $S_k$ , places the request of  $S_k$  at appropriate location in the request queue, and sends a  $\text{GRANT}(j)$  to the top request's site in the queue. Maekawa's algorithm requires extra messages to handle deadlocks
- Maximum number of messages required per CS execution in this case is  $5\sqrt{N}$ .

# Agarwal-EI Abbadi Quorum-Based Algorithm

Agarwal-EI Abbadi quorum-based algorithm uses 'tree-structured quorums'.

- All the sites in the system are logically organized into a complete binary tree.
- For a complete binary tree with level ' $k$ ', we have  $2^{k+1} - 1$  sites with its root at level  $k$  and leaves at level 0.
- The number of sites in a path from the root to a leaf is equal to the level of the tree  $k+1$  which is equal to  $O(\log n)$ .
- A path in a binary tree is the sequence  $a_1, a_2 \dots a_i, a_{i+1}, \dots, a_k$  such that  $a_i$  is the parent of  $a_{i+1}$ .

# Algorithm for constructing a tree-structured quorum

- The algorithm tries to construct quorums in a way that each quorum represents any path from the root to a leaf.
- If it fails to find such a path (say, because node 'x' has failed), the control goes to the ELSE block which specifies that the failed node 'x' is substituted by two paths both of which start with the left and right children of 'x' and end at leaf nodes.
- If the leaf site is down or inaccessible due to any reason, then the quorum cannot be formed and the algorithm terminates with an error condition.
- The sets that are constructed using this algorithm are termed as *tree quorums*.

## Examples of Tree-Structured Quorums

When there is no node failure, the number of quorums formed is equal to the number of leaf sites.

- Consider the tree of height 3 shown in Figure 3, constructed from 15 ( $=2^{3+1}-1$ ) sites.
- In this case 8 quorums are formed from 8 possible root-leaf paths: 1-2-4-8, 1-2-4-9, 1-2-5-10, 1-2-5-11, 1-3-6-12, 1-3-6-13, 1-3-7-14 and 1-3-7-15.
- If any site fails, the algorithm substitutes for that site two possible paths starting from the site's two children and ending in leaf nodes.
- For example, when node 3 fails, we consider possible paths starting from children 6 and 7 and ending at leaf nodes. The possible paths starting from child 6 are 6-12 and 6-13, and from child 7 are 7-14 and 7-15.
- So, when node 3 fails, the following eight quorums can be formed:  
 $\{1,6,12,7,14\}$ ,  $\{1,6,12,7,15\}$ ,  $\{1,6,13,7,14\}$ ,  $\{1,6,13,7,15\}$ ,  $\{1,2,4,8\}$ ,  
 $\{1,2,4,9\}$ ,  $\{1,2,5,10\}$ ,  $\{1,2,5,11\}$ .

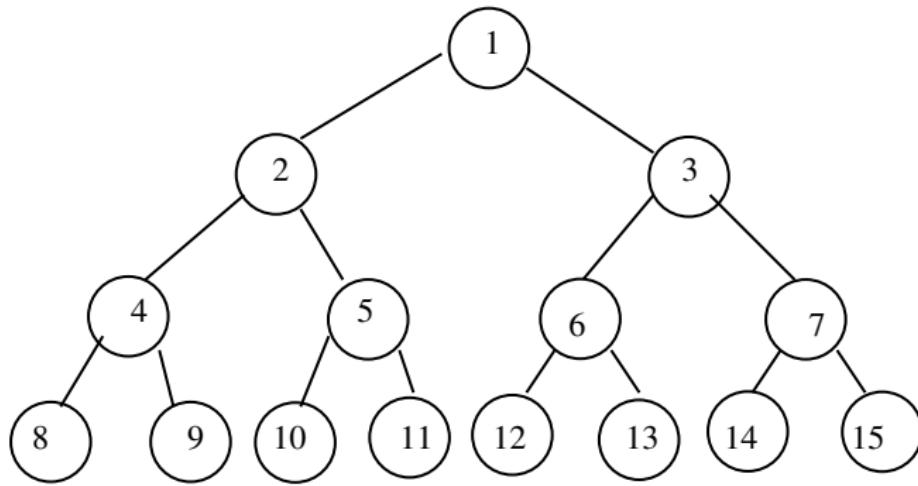


Figure 3: A tree of 15 sites.

# Examples of Tree-Structured Quorums

- Since the number of nodes from root to leaf in an ' $n$ ' node complete tree is  $\log n$ , the best case for quorum formation, i.e, the least number of nodes needed for a quorum is  $\log n$ .
- When the number of node failures is greater than or equal to  $\log n$ , the algorithm may not be able to form tree-structured quorum.
- So, as long as the number of site failures is less than  $\log n$ , the tree quorum algorithm guarantees the formation of a quorum and it exhibits the property of 'graceful degradation'.

# Mutual Exclusion Algorithm

A site  $s$  enters the critical section (CS) as follows:

- Site  $s$  sends a ‘Request’ message to all other sites in the structured quorum it belongs to.
- Each site in the quorum stores incoming requests in a *request queue*, ordered by their timestamps.
- A site sends a ‘Reply’ message, indicating its consent to enter CS, only to the request at the head of its *request queue*, having the lowest timestamp.
- If the site  $s$  gets a ‘Reply’ message from all sites in the structured quorum it belongs to, it enters the CS.
- After exiting the CS,  $s$  sends a ‘Relinquish’ message to all sites in the structured quorum. On the receipt of the ‘Relinquish’ message, each site removes  $s$ ’s request from the head of its *request queue*.

# Mutual Exclusion Algorithm

- If a new request arrives with a timestamp smaller than the request at the head of the queue, an 'Inquire' message is sent to the process whose request is at the head of the queue and waits for a 'Yield' or 'Relinquish' message.
- When a site  $s$  receives an 'Inquire' message, it acts as follows:  
If  $s$  has acquired all of its necessary replies to access the CS, then it simply ignores the 'Inquire' message and proceeds normally and sends a 'Relinquish' message after exiting the CS.  
If  $s$  has not yet collected enough replies from its quorum, then it sends a 'Yield' message to the inquiring site.
- When a site gets the 'Yield' message, it puts the pending request (on behalf of which the 'Inquire' message was sent) at the head of the queue and sends a 'Reply' message to the requestor.

## Correctness proof

Mutual exclusion is guaranteed because the set of quorums satisfy the Intersection property.

- Consider a coterie C which consists of quorums  $\{1,2,3\}$ ,  $\{2,4,5\}$  and  $\{4,1,6\}$ .
- Suppose nodes 3, 5 and 6 want to enter CS, and they send requests to sites  $(1, 2)$ ,  $(2, 4)$  and  $(1, 4)$ , respectively.
- Suppose site 3's request arrives at site 2 before site 5's request. In this case, site 2 will grant permission to site 3's request and reject site 5's request.
- Similarly, suppose site 3's request arrives at site 1 before site 6's request. So site 1 will grant permission to site 3's request and reject site 6's request.
- Since sites 5 and 6 did not get consent from all sites in their quorums, they do not enter the CS.
- Since site 3 alone gets consent from all sites in its quorum, it enters the CS and mutual exclusion is achieved.

# Token-Based Algorithms

- In token-based algorithms, a unique token is shared among the sites.
- A site is allowed to enter its CS if it possesses the token.
- Token-based algorithms use sequence numbers instead of timestamps. (Used to distinguish between old and current requests.)

# Suzuki-Kasami's Broadcast Algorithm

- If a site wants to enter the CS and it does not have the token, it broadcasts a REQUEST message for the token to all other sites.
- A site which possesses the token sends it to the requesting site upon the receipt of its REQUEST message.
- If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.

## continuation..

This algorithm must efficiently address the following two design issues:

**(1) How to distinguish an outdated REQUEST message from a current REQUEST message:**

- Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied.
- If a site can not determine if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it.
- This will not violate the correctness, however, this may seriously degrade the performance.

**(2) How to determine which site has an outstanding request for the CS:**

- After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them.

## continuation..

The first issue is addressed in the following manner:

- A REQUEST message of site  $S_j$  has the form REQUEST( $j, n$ ) where  $n$  ( $n=1, 2, \dots$ ) is a sequence number which indicates that site  $S_j$  is requesting its  $n^{th}$  CS execution.
- A site  $S_i$  keeps an array of integers  $RN_i[1..N]$  where  $RN_i[j]$  denotes the largest sequence number received in a REQUEST message so far from site  $S_j$ .
- When site  $S_i$  receives a REQUEST( $j, n$ ) message, it sets  $RN_i[j] := \max(RN_i[j], n)$ .
- When a site  $S_i$  receives a REQUEST( $j, n$ ) message, the request is outdated if  $RN_i[j] > n$ .

## continuation..

The second issue is addressed in the following manner:

- The token consists of a queue of requesting sites,  $Q$ , and an array of integers  $LN[1..N]$ , where  $LN[j]$  is the sequence number of the request which site  $S_j$  executed most recently.
- After executing its CS, a site  $S_i$  updates  $LN[i]:=RN_i[i]$  to indicate that its request corresponding to sequence number  $RN_i[i]$  has been executed.
- At site  $S_i$  if  $RN_i[j]=LN[j]+1$ , then site  $S_j$  is currently requesting token.

# The Algorithm

## Requesting the critical section

- (a) If requesting site  $S_i$  does not have the token, then it increments its sequence number,  $RN_i[i]$ , and sends a  $\text{REQUEST}(i, sn)$  message to all other sites. (' $sn$ ' is the updated value of  $RN_i[i]$ .)
- (b) When a site  $S_j$  receives this message, it sets  $RN_j[i]$  to  $\max(RN_j[i], sn)$ . If  $S_j$  has the idle token, then it sends the token to  $S_i$  if  $RN_j[i] = LN[i] + 1$ .

## Executing the critical section

- (c) Site  $S_i$  executes the CS after it has received the token.

# The Algorithm

**Releasing the critical section** Having finished the execution of the CS, site  $S_i$  takes the following actions:

- (d) It sets  $LN[i]$  element of the token array equal to  $RN_i[i]$ .
- (e) For every site  $S_j$  whose id is not in the token queue, it appends its id to the token queue if  $RN_i[j] = LN[j] + 1$ .
- (f) If the token queue is nonempty after the above update,  $S_i$  deletes the top site id from the token queue and sends the token to the site indicated by the id.

# Correctness

Mutual exclusion is guaranteed because there is only one token in the system and a site holds the token during the CS execution.

**Theorem:** A requesting site enters the CS in finite time.

**Proof:**

- Token request messages of a site  $S_i$  reach other sites in finite time.
- Since one of these sites will have token in finite time, site  $S_i$ 's request will be placed in the token queue in finite time.
- Since there can be at most  $N - 1$  requests in front of this request in the token queue, site  $S_i$  will get the token and execute the CS in finite time.

# Performance

- No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request.
- If a site does not hold the token when it makes a request, the algorithm requires  $N$  messages to obtain the token. Synchronization delay in this algorithm is 0 or  $T$ .

# Raymond's Tree-Based Algorithm

- This algorithm uses a spanning tree to reduce the number of messages exchanged per critical section execution.
- The network is viewed as a graph, a spanning tree of a network is a tree that contains all the  $N$  nodes.
- The algorithm assumes that the underlying network guarantees message delivery. All nodes of the network are 'completely reliable'.

# Raymond's Tree-Based Algorithm

- The algorithm operates on a minimal spanning tree of the network topology or a logical structure imposed on the network.
- The algorithm assumes the network nodes to be arranged in an unrooted tree structure.
- Figure 4 shows a spanning tree of seven nodes A, B, C, D, E, F, and G.
- Messages between nodes traverse along the undirected edges of the tree.

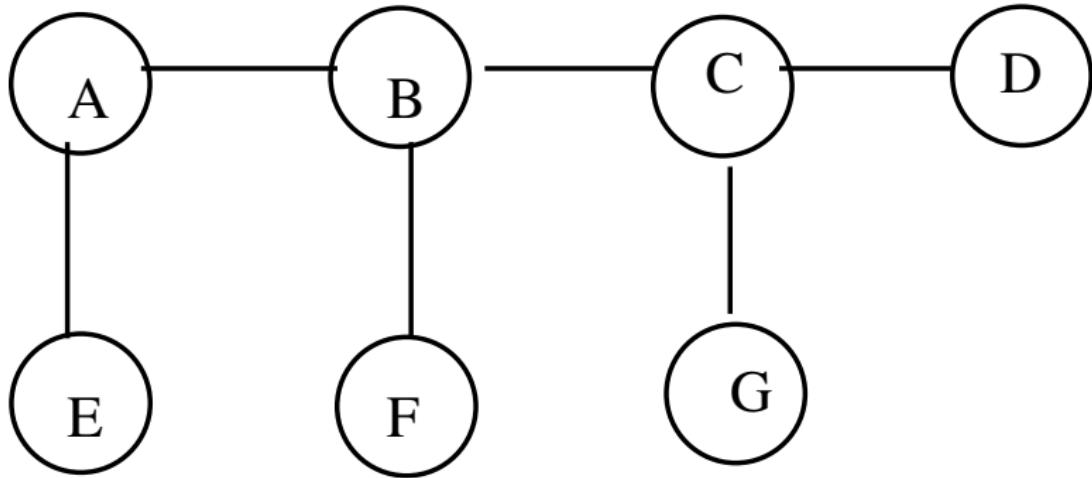


Figure 4: A tree of 15 sites.

# Raymond's Tree-Based Algorithm

- A node needs to hold information about and communicate only to its immediate-neighboring nodes.
- Similar to the concept of tokens used in token-based algorithms, this algorithm uses a concept of privilege.
- Only one node can be in possession of the privilege (called the privileged node) at any time, except when the privilege is in transit from one node to another in the form of a PRIVILEGE message.
- When there are no nodes requesting for the privilege, it remains in possession of the node that last used it.

# The HOLDER Variables

- Each node maintains a HOLDER variable that provides information about the placement of the privilege in relation to the node itself.
- A node stores in its HOLDER variable the identity of a node that it thinks has the privilege or leads to the node having the privilege.
- For two nodes X and Y, if  $\text{HOLDER}_X = Y$ , we could redraw the undirected edge between the nodes X and Y as a directed edge from X to Y.
- For instance, if node G holds the privilege, Figure 4 can be redrawn with logically directed edges as shown in the Figure 5.

# The HOLDER Variables

- The shaded node in Figure 5 represents the privileged node.
- The following will be the values of the HOLDER variables of various nodes:

$\text{HOLDER}_A = \text{B}$

$\text{HOLDER}_B = \text{C}$

$\text{HOLDER}_C = \text{G}$

$\text{HOLDER}_D = \text{C}$

$\text{HOLDER}_E = \text{A}$

$\text{HOLDER}_F = \text{B}$

$\text{HOLDER}_G = \text{self}$

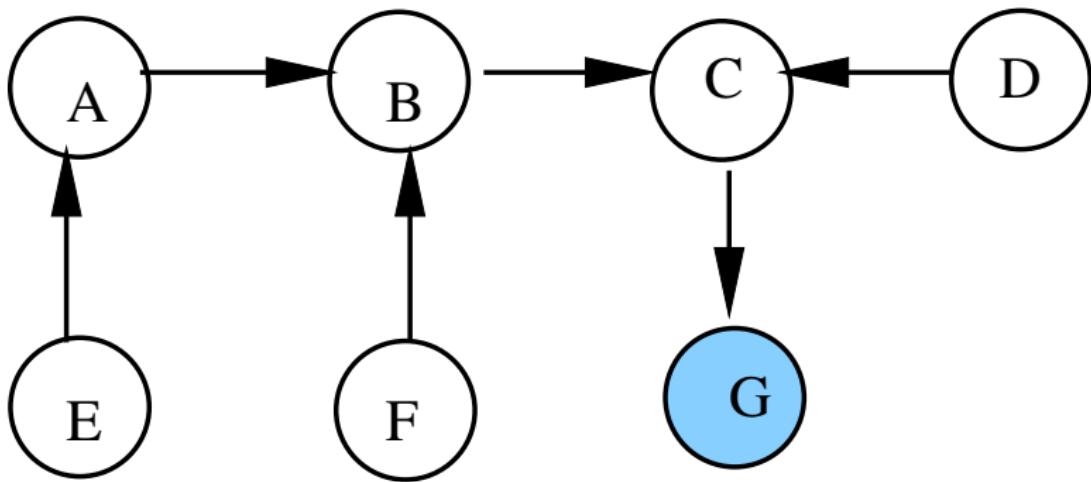


Figure 5: Tree with logically directed edges, all pointing in a direction towards node G - the privileged node.

# The HOLDER Variables

- Now suppose node B that does not hold the privilege wants to execute the critical section.
- B sends a REQUEST message to  $\text{HOLDER}_B$ , i.e., C, which in turn forwards the REQUEST message to  $\text{HOLDER}_C$ , i.e., G.
- The privileged node G, if it no longer needs the privilege, sends the PRIVILEGE message to its neighbor C, which made a request for the privilege, and resets  $\text{HOLDER}_G$  to C.
- Node C, in turn, forwards the PRIVILEGE to node B, since it had requested the privilege on behalf of B. Node C also resets  $\text{HOLDER}_C$  to B.
- The tree in Figure 5 will now look as in Figure 6.

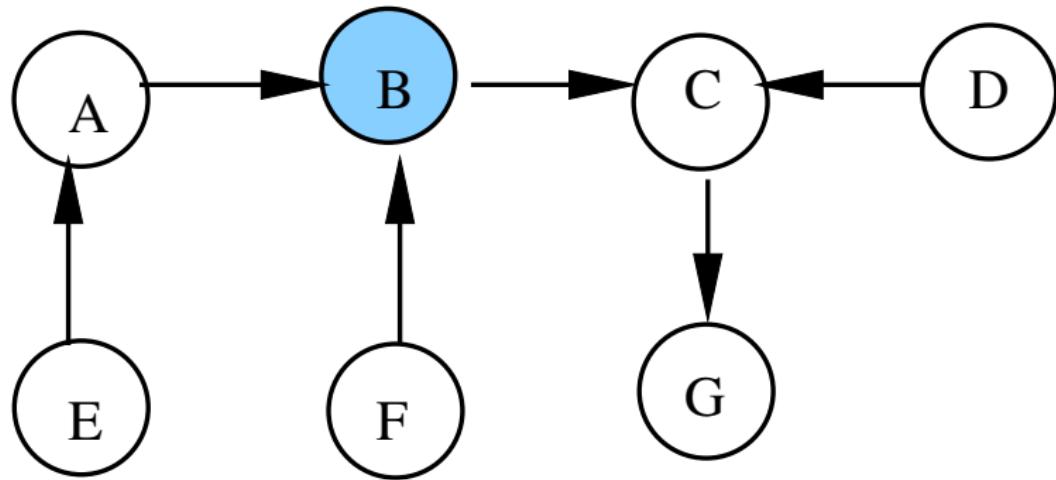


Figure 6: Tree with logically directed edges, all pointing in a direction towards node G - the privileged node.

# Data Structures

Each node maintains the following variables:

Variable Name	Possible Values	Comments
HOLDER	“self” or the identity of one of the immediate neighbours.	Indicates the location of the privileged node in relation to the current node.
USING	True or false.	Indicates if the current node is executing the critical section.
REQUEST_Q	A FIFO queue that could contain “self” or the identities of immediate neighbors as elements.	The REQUEST_Q of a node consists of the identities of those immediate neighbors that have requested for privilege but have not yet been sent the privilege .
ASKED	True or false.	Indicates if node has sent a request for the privilege.

## Data Structures

- The value “self” is placed in REQUEST\_Q if the node makes a request for the privilege for its own use.
- The maximum size of REQUEST\_Q of a node is the number of immediate neighbors + 1 (for “self”).
- ASKED prevents the sending of duplicate requests for privilege, and also makes sure that the REQUEST\_Qs of the various nodes do not contain any duplicate elements.

# The Algorithm

The algorithm consists of the following routines:

- ASSIGN\_PRIVILEGE
- MAKE\_REQUEST

## ASSIGN\_PRIVILEGE:

This is a routine sends a PRIVILEGE message. A privileged node sends a PRIVILEGE message if

- it holds the privilege but is not using it,
- its REQUEST\_Q is not empty, and
- the element at the head of its REQUEST\_Q is not “self.”

# ASSIGN\_PRIVILEGE

- A situation where “self” is at the head of REQUEST\_Q may occur immediately after a node receives a PRIVILEGE message.
- The node will enter into the critical section after removing “self” from the head of REQUEST\_Q. If the id of another node is at the head of REQUEST\_Q, then it is removed from the queue and a PRIVILEGE message is sent to that node.
- Also, the variable ASKED is set to false since the currently privileged node will not have sent a request for the PRIVILEGE message.

# MAKE\_REQUEST

This is a routine sends a REQUEST message. An unprivileged node sends a REQUEST message if

- it does not hold the privilege,
- its REQUEST\_Q is not empty, i.e., it requires the privilege for itself, or on behalf of one of its immediate neighboring nodes, and
- it has not sent a REQUEST message already.

# MAKE\_REQUEST

- The variable ASKED is set to true to reflect the sending of the REQUEST message. The MAKE\_REQUEST routine makes no change to any other variables.
- The variable ASKED will be true at a node when it has sent REQUEST message to an immediate neighbor and has not received a response.
- A node does not send any REQUEST messages, if ASKED is true at that node. Thus the variable ASKED makes sure that unnecessary REQUEST messages are not sent from the unprivileged node.
- This makes the REQUEST\_Q of any node bounded, even when operating under heavy load.

# Events

Below we show four events that constitute the algorithm.

---

Event	Algorithm Functionality	
A node wishes to execute critical section.	Enqueue(REQUEST_Q, self); SIGN_PRIVILEGE; MAKE_REQUEST	AS-
A node receives a REQUEST message from one of its immediate neighbors X.	Enqueue(REQUEST_Q, X); SIGN_PRIVILEGE; MAKE_REQUEST	AS-
A node receives a PRIVILEGE message.	HOLDER := self; ASSIGN_PRIVILEGE; MAKE_REQUEST	
A node exits the critical section.	USING := false; ASSIGN_PRIVILEGE; MAKE_REQUEST	

---

# Events..

## **A node wishes critical section entry:**

If it is the privileged node, the node could enter the critical section using the ASSIGN\_PRIVILEGE routine. Otherwise, it sends a REQUEST message using the MAKE\_REQUEST routine.

## **A node receives a REQUEST message from one of its immediate neighbors:**

If this node is the current HOLDER, it may send the PRIVILEGE to a requesting node using the ASSIGN\_PRIVILEGE routine. Otherwise, it forwards the request using the MAKE\_REQUEST routine.

# Events..

## **A node receives a PRIVILEGE message:**

The ASSIGN\_PRIVILEGE routine could result in the execution of the critical section at the node, or may forward the privilege to another node. After the privilege is forwarded, the MAKE\_REQUEST routine could send a REQUEST message to reacquire the privilege, for a pending request at this node.

## **A node exits the critical section:**

On exit from the critical section, this node may pass the privilege on to a requesting node using the ASSIGN\_PRIVILEGE routine. It may then use the MAKE\_REQUEST routine to get back the privilege, for a pending request at this node.

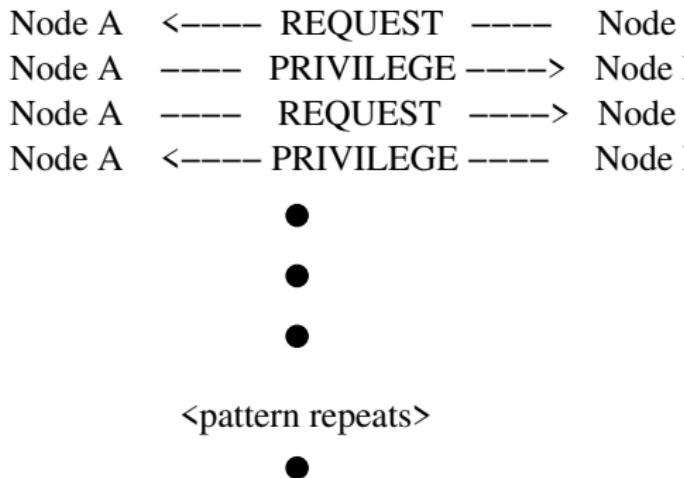


Figure 7: Logical pattern of message flow between neighboring nodes A and B.

# Message Overtaking

- This algorithm does away with the use of sequence numbers. The algorithm works such that message flow between any two neighboring nodes sticks to a logical pattern as shown in the Figure 7.
- If at all message overtaking occurs between the nodes A and B, it can occur when a PRIVILEGE message is sent from node A to node B, which is then very closely followed by a REQUEST message from node A to node B.
- Such a message overtaking will not affect the operation of the algorithm.
- If node B receives the REQUEST message from node A before receiving the PRIVILEGE message from node A, A's request will be queued in  $\text{REQUEST\_Q}_B$ . Since B is not a privileged node, it will not be able to send a privilege to node A in reply.
- When node B receives the PRIVILEGE message from A after receiving the REQUEST message, it could enter the critical section or could send a PRIVILEGE message to an immediate neighbor at the head of  $\text{REQUEST\_Q}_B$ , which need not be node A. So message overtaking does not affect the algorithm.

# Correctness

The algorithm provides the following guarantees:

- Mutual exclusion is guaranteed
- Deadlock is impossible
- Starvation is impossible

## Mutual Exclusion

- The algorithm ensures that at any instant of time, not more than one node holds the privilege.
- Whenever a node receives a PRIVILEGE message, it becomes privileged. Similarly, whenever a node sends a PRIVILEGE message, it becomes unprivileged.
- Between the instants one node becomes unprivileged and another node becomes privileged, there is no privileged node. Thus, there is at most one privileged node at any point of time in the network.

# Deadlock is Impossible

When the critical section is free, and one or more nodes want to enter the critical section but are not able to do so, a deadlock may occur. This could happen due to any of the following scenarios:

- ① The privilege cannot be transferred to a node because no node holds the privilege.
- ② The node in possession of the privilege is unaware that there are other nodes requiring the privilege.
- ③ The PRIVILEGE message does not reach the requesting unprivileged node.

# Deadlock is Impossible

- The scenario 1 can never occur in this algorithm because nodes do not fail and messages are not lost.
- The logical pattern established using HOLDER variables ensures that a node that needs the privilege sends a REQUEST message either to a node holding the privilege or to a node that has a path to a node holding the privilege. Thus scenario 2 can never occur.
- The series of REQUEST messages are enqueued in the REQUEST\_Qs of various nodes such that the REQUEST\_Qs of those nodes collectively provide a logical path for the transfer of the PRIVILEGE message from the privileged node to the requesting unprivileged nodes. So scenario 3 can never occur.

# Starvation is Impossible

- When a node A holds the privilege, and another node B requests for the privilege, the identity of B or the id's of proxy nodes for node B will be present in the REQUEST\_Qs of various nodes in the path connecting the requesting node to the currently privileged node.
- So depending upon the position of the id of node B in those REQUEST\_Qs, node B will sooner or later receive the privilege.
- Thus once node B's REQUEST message reaches the privileged node A, node B, is sure to receive the privilege.

# Cost and Performance Analysis

- In the worst-case, the algorithm requires ( $2 * \text{longest path length of the tree}$ ) messages per critical section entry.
- This happens when the privilege is to be passed between nodes at either ends of the longest path of the minimal spanning tree.
- The worst possible network topology for this algorithm is where all nodes are arranged in a straight line and the longest path length will be  $N - 1$ , and thus the algorithm will exchange  $2 * (N - 1)$  messages per CS execution.
- However, if all nodes generate equal number of REQUEST messages for the privilege, the average number of messages needed per critical section entry will be approximately  $2N/3$  because the average distance between a requesting node and a privileged node is  $(N + 1)/3$ .

# Cost and Performance Analysis

- The best topology for the algorithm is the radiating star topology. The worst case cost of this algorithm for this topology is  $O(\log_{K-1} N)$ .
- Trees with higher fan-outs are preferred over radiating star topologies. The longest path length of such trees is typically  $O(\log N)$ . Thus, on an average, this algorithm involves the exchange of  $O(\log N)$  messages per critical section execution.
- Under heavy load, the algorithm exhibits an interesting property: “As the number of nodes requesting for the privilege increases, the number of messages exchanged per critical section entry decreases.”
- In heavy load, the algorithm requires exchange of only four messages per CS execution.

# Distributed Deadlock Detection

# Distributed Deadlock Detection

- Assumptions:
  - System has only reusable resources
  - Only exclusive access to resources
  - Only one copy of each resource
- States of a process: running or blocked
- Running state: process has all the resources
- Blocked state: waiting on one or more resource

# Deadlocks

- Resource Deadlocks
  - A process needs multiple resources for an activity.
  - Processes can simultaneously wait for several resources and cannot proceed until they have acquired all those resources
  - Deadlock occurs if each process in a set request resources held by another process in the same set, and it must receive all the requested resources to move further.
- Communication Deadlocks
  - Processes wait to communicate with other processes in a set.
  - Each process in the set is waiting on another process's message, and no process in the set initiates a message until it receives a message for which it is waiting.

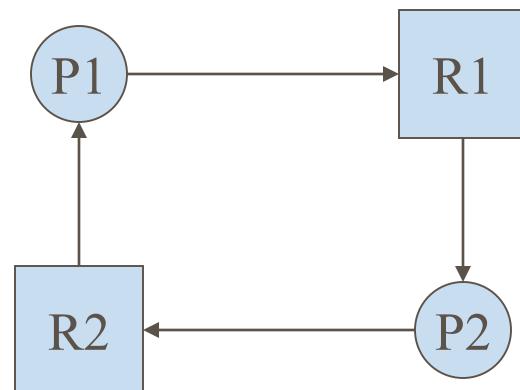
# Wait For Graphs (WFG)

- Nodes of a graph are processes. Edges of a graph the pending requests or assignment of resources.
- Wait-for Graphs (WFG):  $P_1 \rightarrow P_2$  implies  $P_1$  is waiting for a resource from  $P_2$ .
- Transaction-wait-for Graphs (TWG): WFG in databases.
- Deadlock: directed cycle in the graph.
- Cycle example:



# Wait For Graphs (WFG)

- Wait-for Graphs (WFG):  $P_1 \rightarrow P_2$  implies  $P_1$  is waiting for a resource from  $P_2$ .



# Models of Deadlocks

## - Single Resource Model

Distributed systems allow several kinds of resource requests.

### The Single Resource Model

- In the single resource model, a process can have at most one outstanding request for only one unit of a resource.
- Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock.

# Models of Deadlocks

## -AND Model

- In the AND model, a process can request for more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process.
- The out degree of a node in the WFG for AND model can be more than 1.
- The presence of a cycle in the WFG indicates a deadlock in the AND model.
- Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

# Models of Deadlocks

## -OR Model

- In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.
- Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model.
- Consider example in Figure 1: If all nodes are OR nodes, then process  $P_{11}$  is not deadlocked because once process  $P_{33}$  releases its resources,  $P_{32}$  shall become active as one of its requests is satisfied.
- After  $P_{32}$  finishes execution and releases its resources, process  $P_{11}$  can continue with its processing.
- In the OR model, the presence of a knot indicates a deadlock.

# Models of Deadlocks

## -OR Model

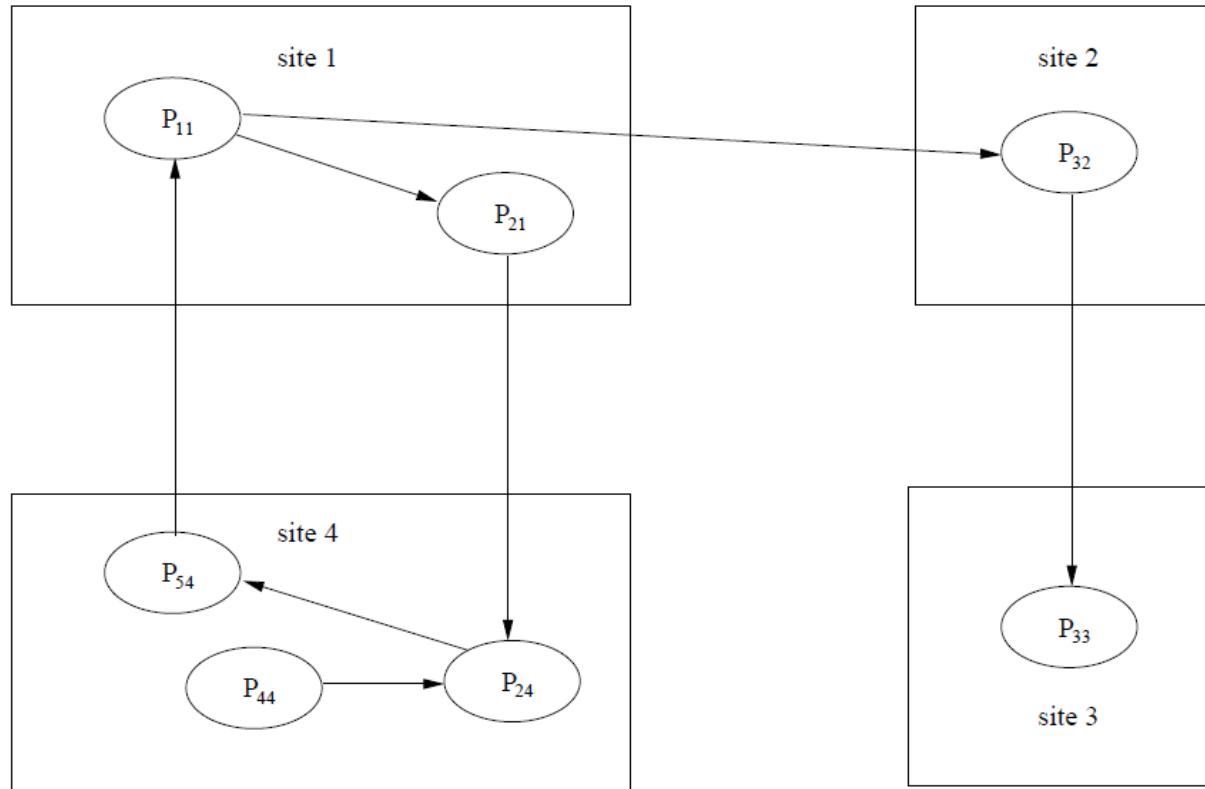


Figure 1: An Example of a WFG

# Models of Deadlocks

## -AND/OR Model

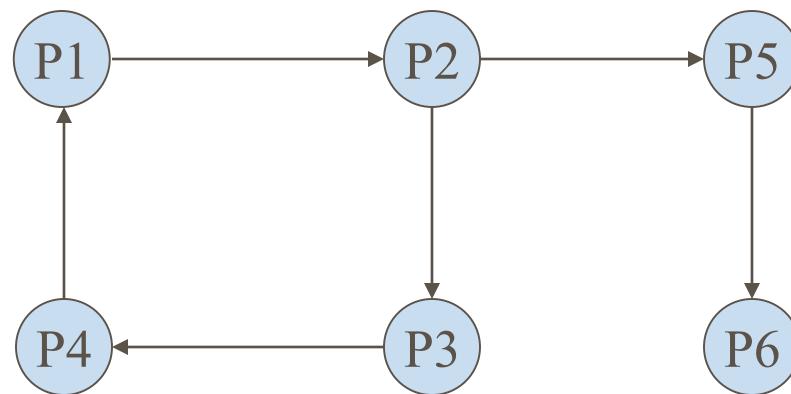
- A generalization of the previous two models (OR model and AND model) is the AND-OR model.
- In the AND-OR model, a request may specify any combination of *and* and *or* in the resource request.
- For example, in the AND-OR model, a request for multiple resources can be of the form  $x \text{ and } (y \text{ or } z)$ .
- To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG.
- Since a deadlock is a stable property, a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock.

# AND, OR Models

- AND Model
  - A process/transaction can simultaneously request for multiple resources.
  - Remains blocked until it is granted *all* of the requested resources.
  
- OR Model
  - A process/transaction can simultaneously request for multiple resources.
  - Remains blocked till *any one* of the requested resource is granted.

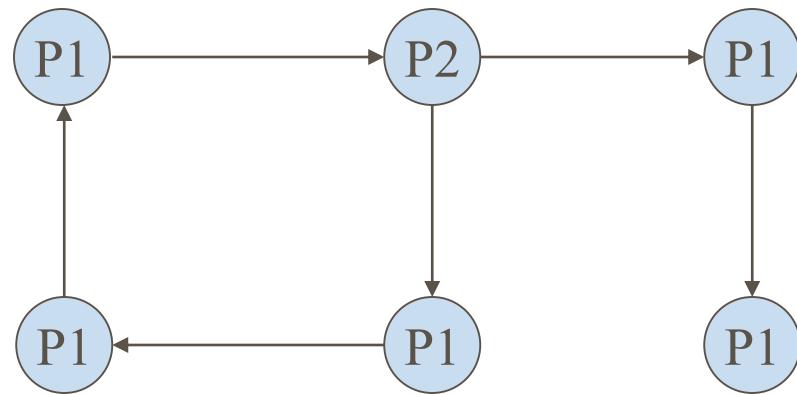
# Sufficient Condition

Deadlock ??



# AND, OR Models

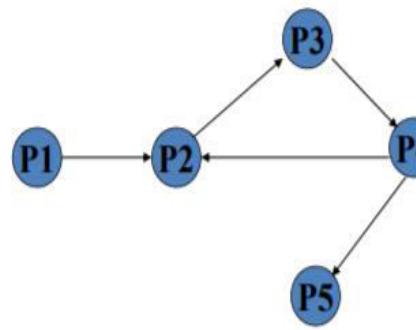
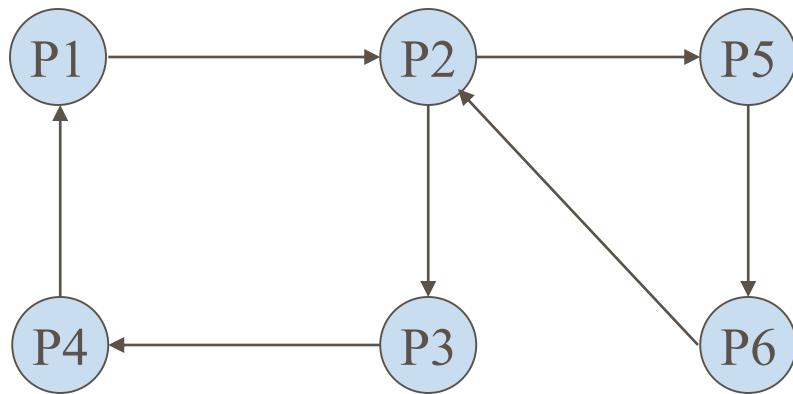
- AND Model
  - Presence of a cycle.



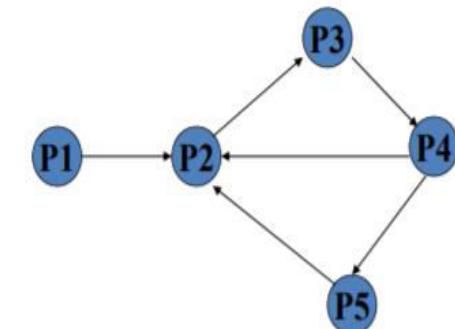
# AND, OR Models

## ■ OR Model

- Presence of a knot.
- Knot: Subset of a graph such that starting from any node in the subset, it is impossible to leave the knot by following the edges of the graph.



No deadlock



Deadlock

# AND, OR Models

- The AND model of requests requires all resources currently being requested to be granted to unblock a computation
  - A **cycle is sufficient** to declare a deadlock with this model
- The OR model of requests allows a computation making multiple different resource requests to unblock as soon as any are granted
  - A **cycle is a necessary condition**
  - A **knot** is a **sufficient** condition

# Models of Deadlocks

## -P out of Q $\binom{P}{Q}$ Model

- The  $\binom{p}{q}$  model (called the P-out-of-Q model) allows a request to obtain any k available resources from a pool of n resources.
- It has the same expressive power as the AND-OR model.
- However,  $\binom{p}{q}$  model lends itself to a much more compact formation of a request.
- Every request in the  $\binom{p}{q}$  model can be expressed in the AND-OR model and vice-versa.
- Note that AND requests for p resources can be stated as  $\binom{p}{p}$  and OR requests for p resources can be stated as  $\binom{1}{p}$ .

# Deadlock Handling Strategies

- Deadlock Prevention: difficult
- Deadlock Avoidance: before allocation, check for possible deadlocks.
  - Difficult as it needs global state info in each site (that handles resources).
- Deadlock Detection: Find cycles. Focus of discussion.
- Deadlock detection algorithms must satisfy 2 conditions:
  - No undetected deadlocks - **Progress**
  - No false deadlocks - **Safety**

# Distributed Deadlocks

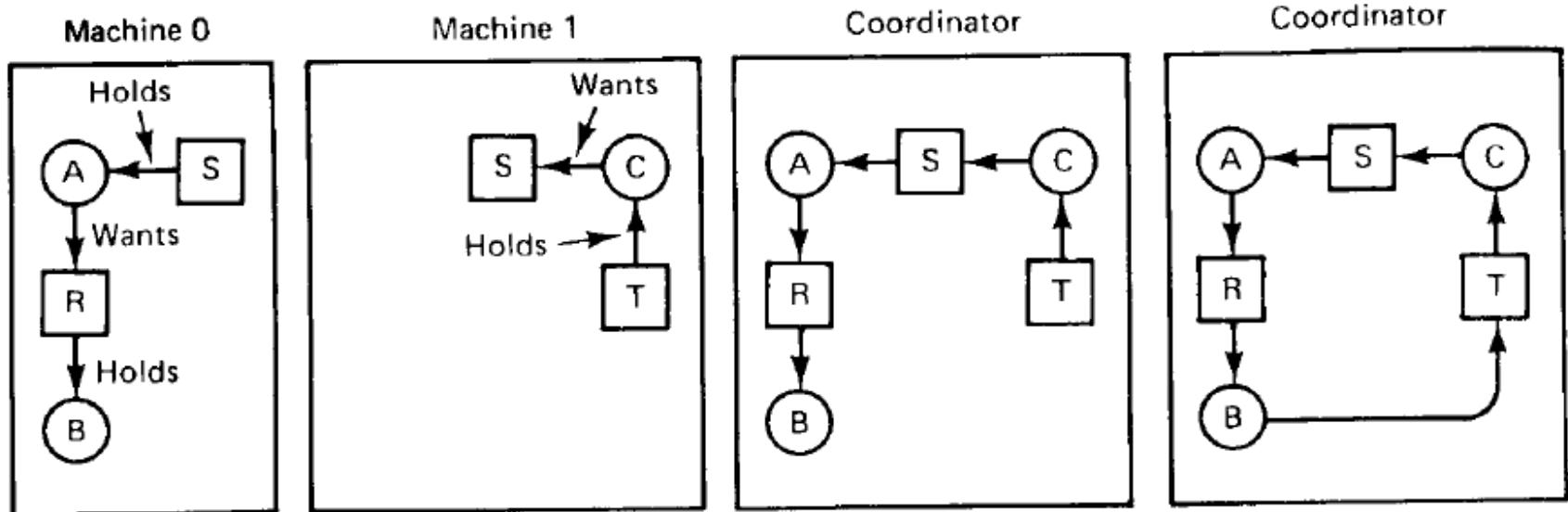
- Centralized Control
  - A *control site* constructs wait-for graphs (WFGs) and checks for directed cycles.
  - WFG can be maintained continuously (or) built on-demand by requesting WFGs from individual sites.
  - **Single point failure**
- Distributed Control
  - WFG is spread over different sites. Any site can initiate the deadlock detection process.
  - **Difficult to design**
  - **Several sites may involve in same deadlock detection**
- Hierarchical Control
  - Sites are arranged in a hierarchy.
  - A site checks for cycles only in descendants.
  - **Hierarchy should be carefully formed**
  - **Difficult if most deadlocks span several clusters**

# Centralized Algorithms

- Completely centralized algorithm
  - Simplest algorithm
  - A designated site called *control site* maintains the WFG for the entire system
  - All the sites send *request resource* and *release resource* messages to the *control site*
  - *Control site* updates the WFG to check if any deadlock
  - **Simple but inefficient**
  - **Control site may be overloaded**

# Centralized Algorithms

- Completely centralized algorithm



- System may detect **false deadlocks**

# Centralized Algorithms

If R1 and R2 are stored at sites S1 and S2 and two transactions T1 and T2 are started simultaneously at sites S3 and S4...

T1

lock R1

unlock R1

lock R2

unlock R2

T2

lock R1

unlock R1

lock R2

unlock R2

**False deadlock T1 -> T2 -> T1 (due to inconsistent view and lack of perfectly synchronized clocks (i.e) lock R2 of T2 arrives before lock R2 of T1)**

# Centralized Algorithms

- Ho-Ramamoorthy 2-phase Algorithm
  - Each site maintains a status table of all processes initiated at that site: includes all resources locked & all resources being waited on.
  - Controller requests (periodically) the status table from each site.
  - Controller then constructs WFG from these tables, searches for cycle(s).
  - If no cycles, no deadlocks.
  - Otherwise, (cycle exists): Request for state tables again.
  - Construct WFG based *only* on common transactions in the 2 tables.
  - If the same cycle is detected again, system is in deadlock.
  - Later proved: cycles in 2 consecutive reports *need not* result in a deadlock. Hence, this algorithm detects false deadlocks.

# Centralized Algorithms...

- Ho-Ramamoorthy 1-phase Algorithm
  - Each site maintains 2 status tables: *resource status* table and *process status* table.
  - **Resource table:** transactions that have locked or are waiting for resources.
  - **Process table:** resources locked by or waited on by transactions.
  - Controller periodically collects these tables from each site.
  - Constructs a WFG from transactions common to both the tables.
  - No cycle, no deadlocks.
  - A cycle means a deadlock.
  - No false deadlock detected
  - Faster and requires only few messages
  - But requires more storage because of presence of two tables

# Distributed Algorithms

- **Path-pushing:** resource dependency information disseminated through designated paths (in the graph).
- **Edge-chasing:** special messages or probes circulated along edges of WFG. Deadlock exists if the probe is received back by the initiator.
- **Diffusion computation:** queries on status sent to process in WFG.
- **Global state detection:** get a snapshot of the distributed system.

## Path-Pushing Algorithms

- In path-pushing algorithms, distributed deadlocks are detected by maintaining an explicit global WFG.
- The basic idea is to build a global WFG for each site of the distributed system.
- In this class of algorithms, at each site whenever deadlock computation is performed, it sends its local WFG to all the neighboring sites.
- After the local data structure of each site is updated, this updated WFG is then passed along to other sites, and the procedure is repeated until some site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present.
- This feature of sending around the paths of global WFG has led to the term path-pushing algorithms.

## Edge-Chasing Algorithms

- In an edge-chasing algorithm, the presence of a cycle in a distributed graph structure is verified by propagating special messages called probes, along the edges of the graph.
- These probe messages are different than the request and reply messages.
- The formation of cycle can be deleted by a site if it receives the matching probe sent by it previously.
- Whenever a process that is executing receives a probe message, it discards this message and continues.
- Only blocked processes propagate probe messages along their outgoing edges.
- Main advantage of edge-chasing algorithms is that probes are fixed size messages which is normally very short.

## Diffusing Computations Based Algorithms

- In *diffusion computation* based distributed deadlock detection algorithms, deadlock detection computation is diffused through the WFG of the system.
- These algorithms make use of echo algorithms to detect deadlocks.
- This computation is superimposed on the underlying distributed computation. If this computation terminates, the initiator declares a deadlock.
- To detect a deadlock, a process sends out query messages along all the outgoing edges in the WFG.
- These queries are successively propagated (i.e., diffused) through the edges of the WFG.

## Diffusing Computations Based Algorithms

- When a blocked process receives first query message for a particular deadlock detection initiation, it does not send a reply message until it has received a reply message for every query it sent.
- For all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message.
- The initiator of a deadlock detection detects a deadlock when it receives reply for every query it had sent out.

## Global state detection based algorithms

- Global state detection based deadlock detection algorithms exploit the following facts:
  - ① A consistent snapshot of a distributed system can be obtained without freezing the underlying computation and
  - ② If a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.
- Therefore, distributed deadlocks can be detected by taking a snapshot of the system and examining it for the condition of a deadlock.

# Obermarck's Path Pushing algorithm

- The algorithm works as follows iteratively:
  - The site waits for deadlock-related information from other sites (fourth step in the algorithm)
  - The site combines the received information with its local WFG or TWF graph to build an updated TWF graph
  - It then detects all cycles and breaks only those that do not contain the node 'Ex' or external node.
  - For all cycles 'Ex -> T1 -> T2 -> Ex' the site transmits them in string form 'Ex, T1, T2, Ex' to all other sites

# Path-pushing

- Obermarck's algorithm for path propagation
  - based on a database model using transaction processing
  - sites which detect a cycle in their partial WFG views convey the paths discovered to members of the (totally ordered) transaction
  - the highest priority transaction detects the deadlock “ $\text{Ex} \Rightarrow T1 \Rightarrow T2 \Rightarrow \text{Ex}$ ”
  - Algorithm can detect *phantoms* due to its asynchronous snapshot method

# Mitchell and Merritt's algorithm for Single Resource Model

- Belongs to the class of edge-chasing algorithms where probes are sent in opposite direction of the edges of WFG.
- When a probe initiated by a process comes back to it, the process declares deadlock.
- Only one process in a cycle detects the deadlock. This simplifies the deadlock resolution – this process can abort itself to resolve the deadlock.

# Mitchell and Merritt's algorithm for Single Resource Model

- Each node of the WFG has two local variables, called labels:
  - ① a private label, which is unique to the node at all times, though it is not constant, and
  - ② a public label, which can be read by other processes and which may not be unique.
- Each process is represented as  $u/v$  where  $u$  and  $v$  are the public and private labels, respectively.
- Initially, private and public labels are equal for each process.
- A global WFG is maintained and it defines the entire state of the system.

# Mitchell and Merritt's algorithm for Single Resource Model

- The algorithm is defined by the four state transitions shown in Figure 2, where  $z = \text{inc}(u, v)$ , and  $\text{inc}(u, v)$  yields a unique label greater than both  $u$  and  $v$  labels that are not shown do not change.
- Block creates an edge in the WFG.
- Two messages are needed, one resource request and one message back to the blocked process to inform it of the public label of the process it is waiting for.
- Activate denotes that a process has acquired the resource from the process it was waiting for.
- Transmit propagates larger labels in the opposite direction of the edges by sending a probe message.

# Mitchell and Merritt's algorithm for Single Resource Model

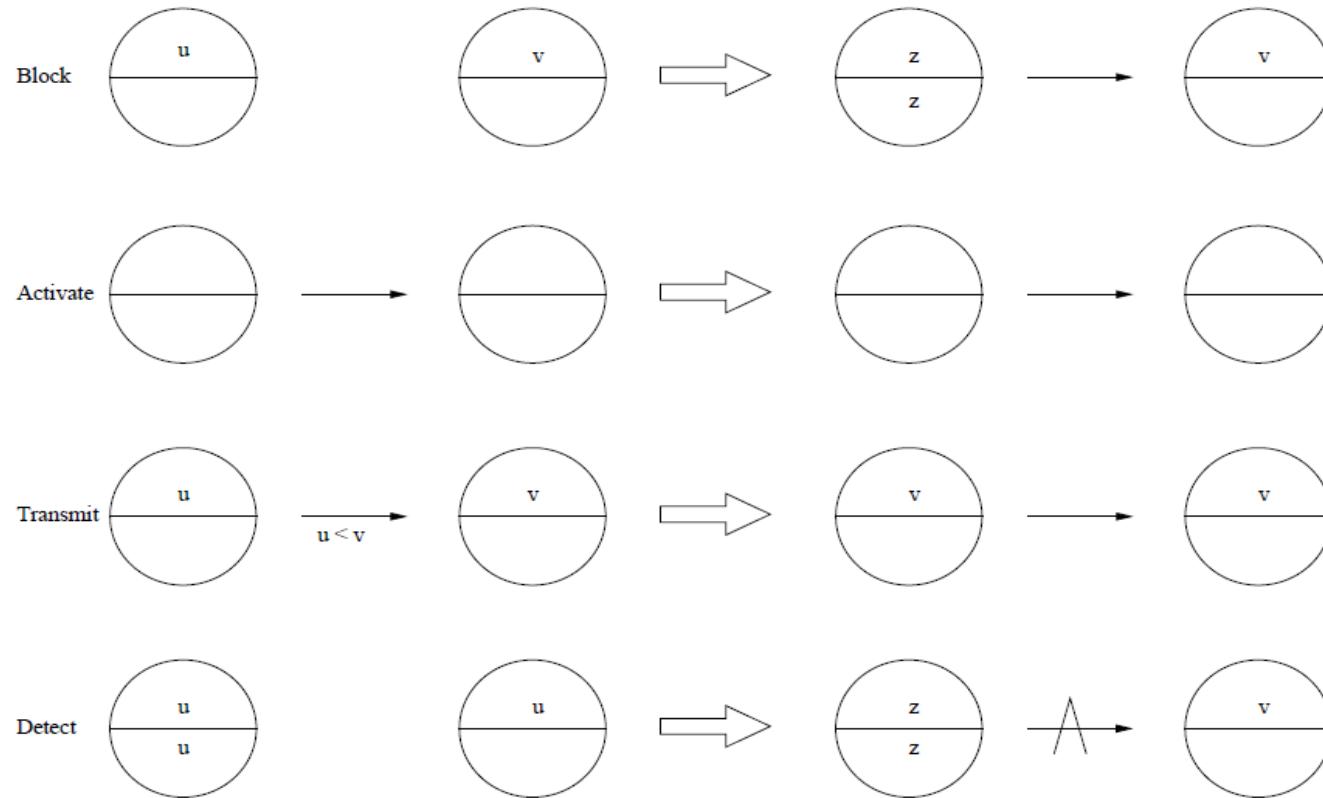


Figure 2: The four possible state transitions

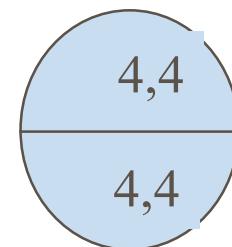
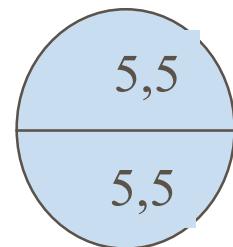
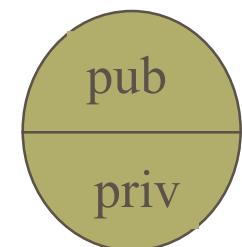
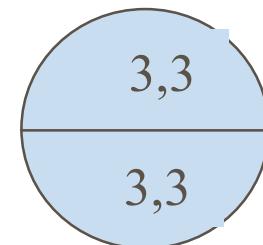
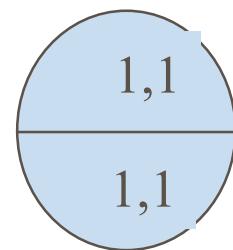
# Mitchell and Merritt's algorithm for Single Resource Model

- Whenever a process receives a probe which is less than its public label, then it simply ignores that probe.
- Detect means that the probe with the private label of some process has returned to it, indicating a deadlock.
- The above algorithm can be easily extended to include priorities where whenever a deadlock occurs, the lowest priority process gets aborted.

## Message Complexity:

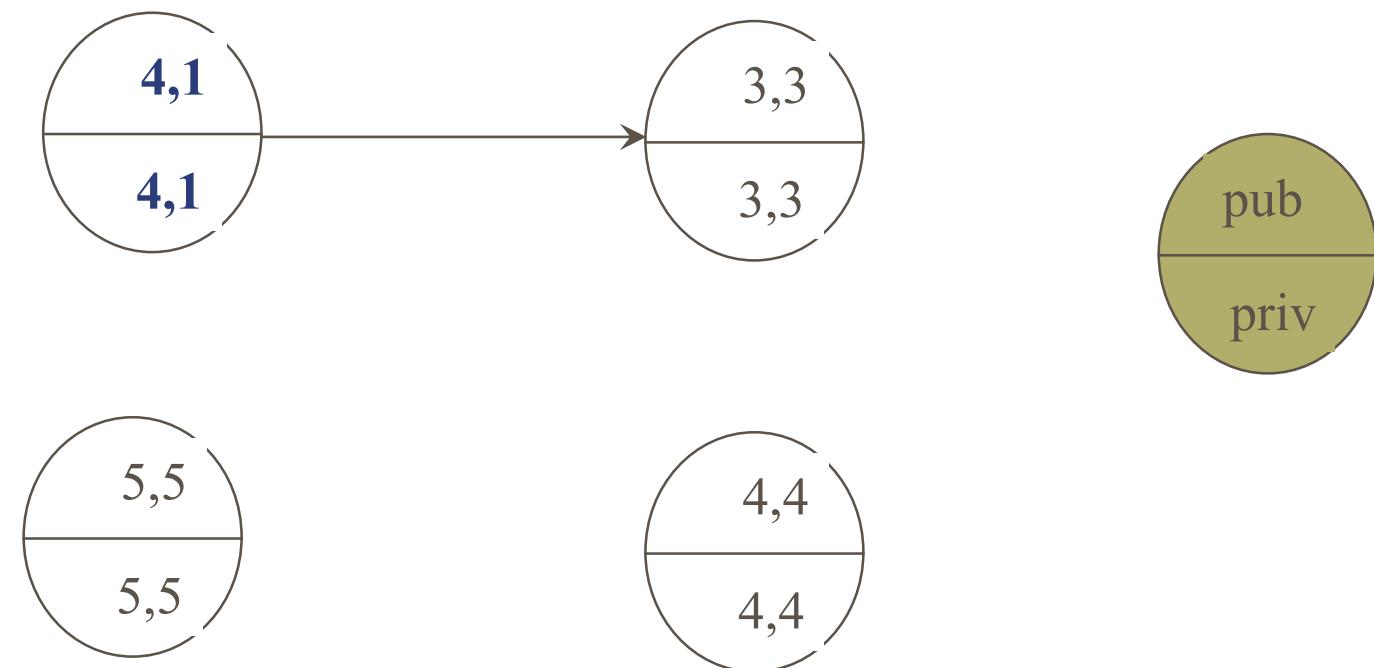
If we assume that a deadlock persists long enough to be detected, the worst-case complexity of the algorithm is  $s(s - 1)/2$  Transmit steps, where  $s$  is the number of processes in the cycle.

# Mitchell-Merritt Example



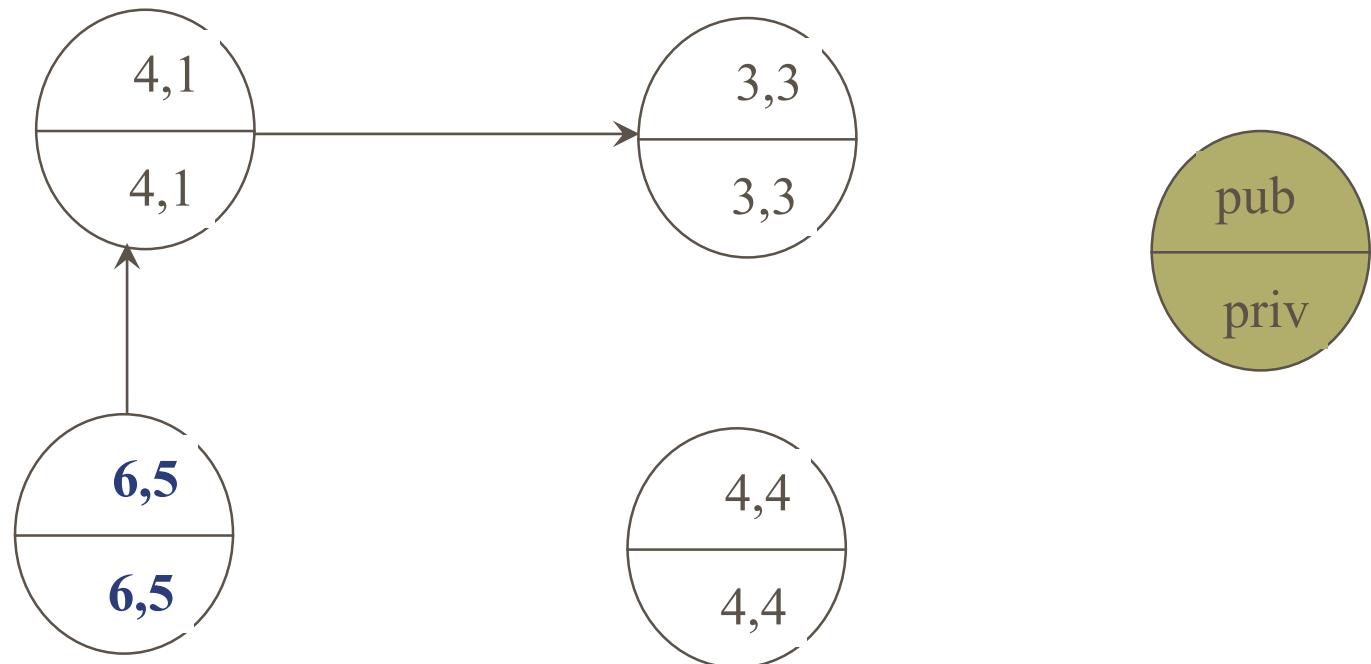
# Mitchell-Merritt Example

BLOCK



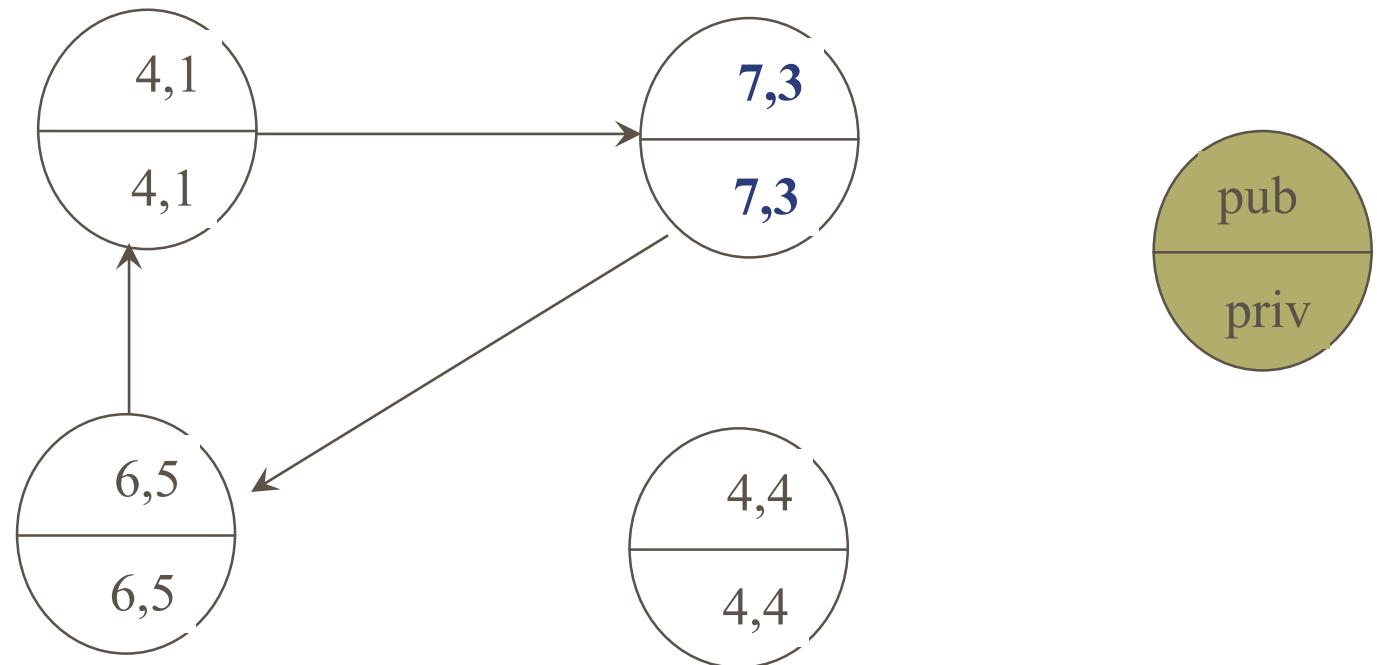
# Mitchell-Merritt Example

BLOCK

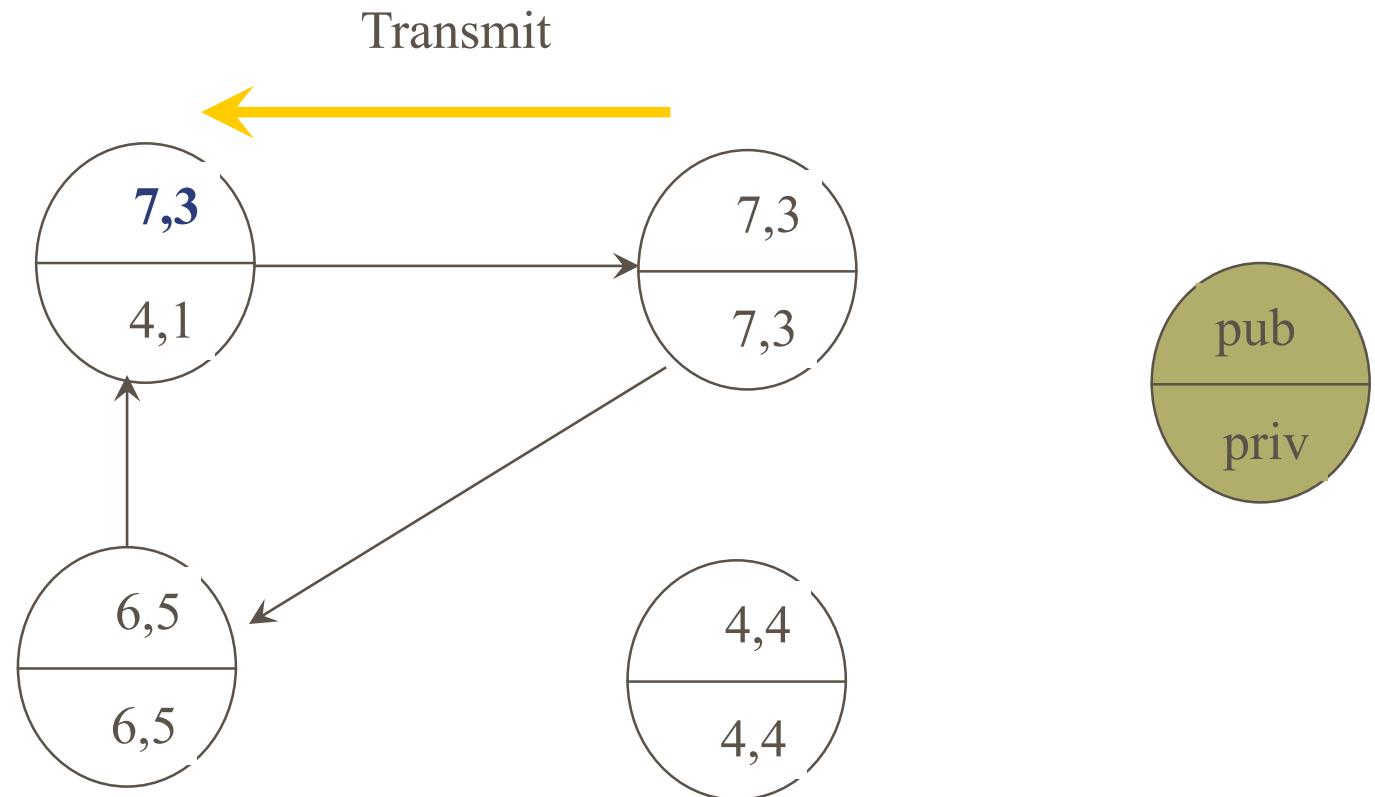


# Mitchell-Merritt Example

BLOCK

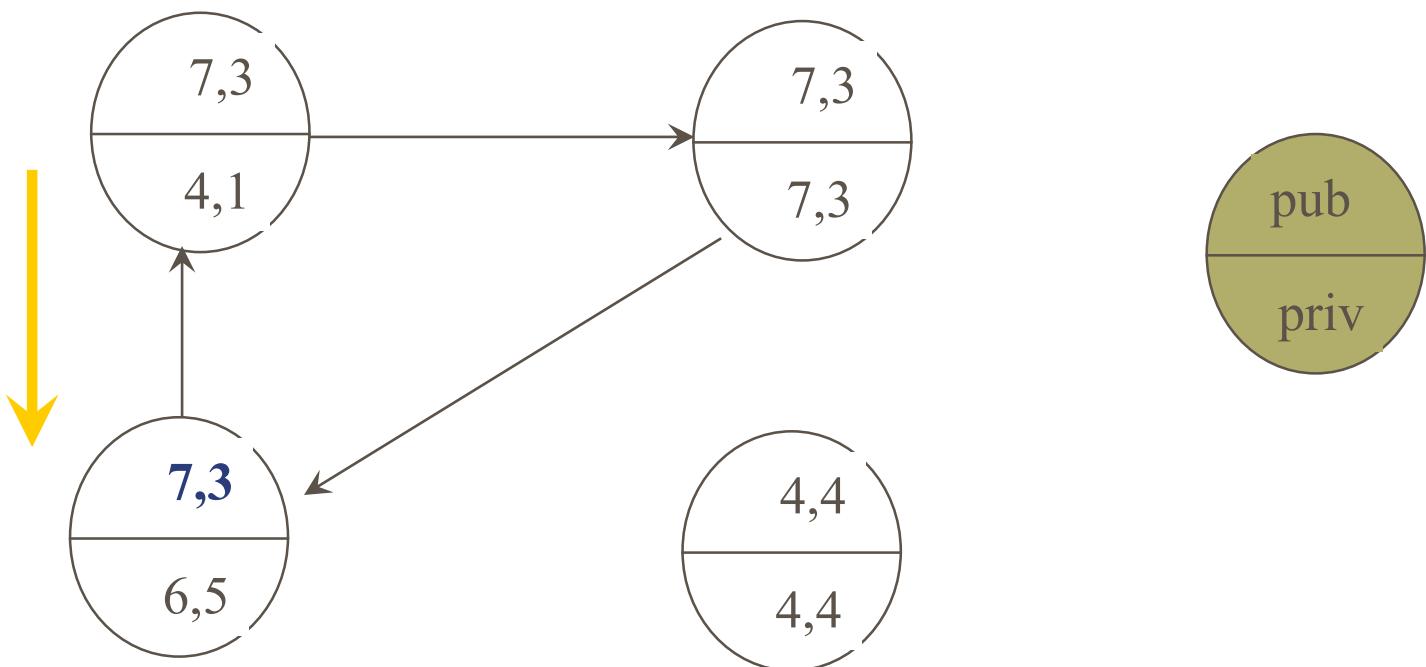


# Mitchell-Merritt Example



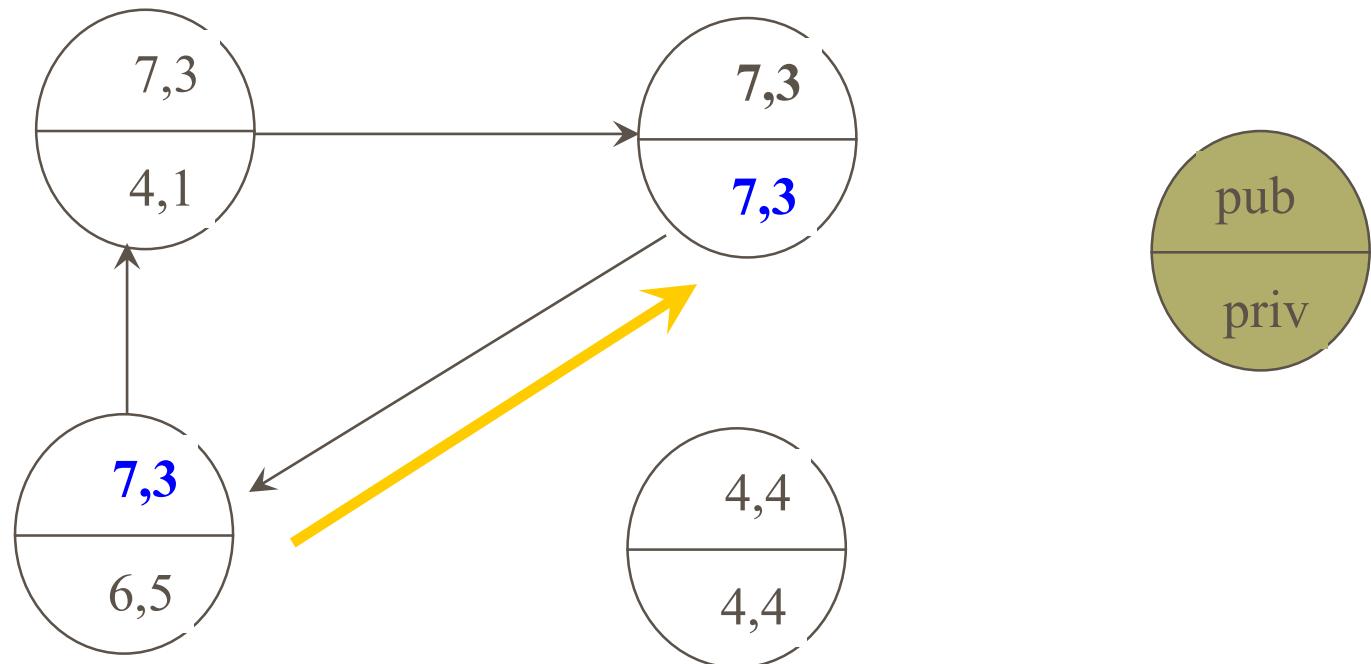
# Mitchell-Merritt Example

Transmit



# Mitchell-Merritt Example

DEADLOCK

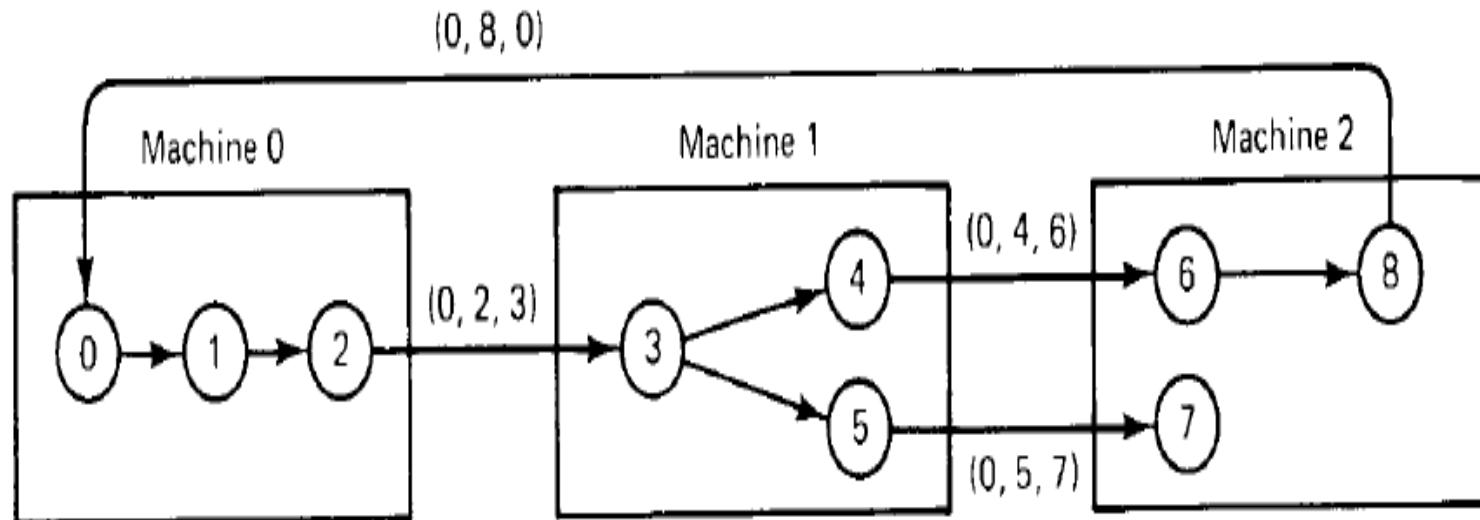


# Edge-Chasing Algorithm

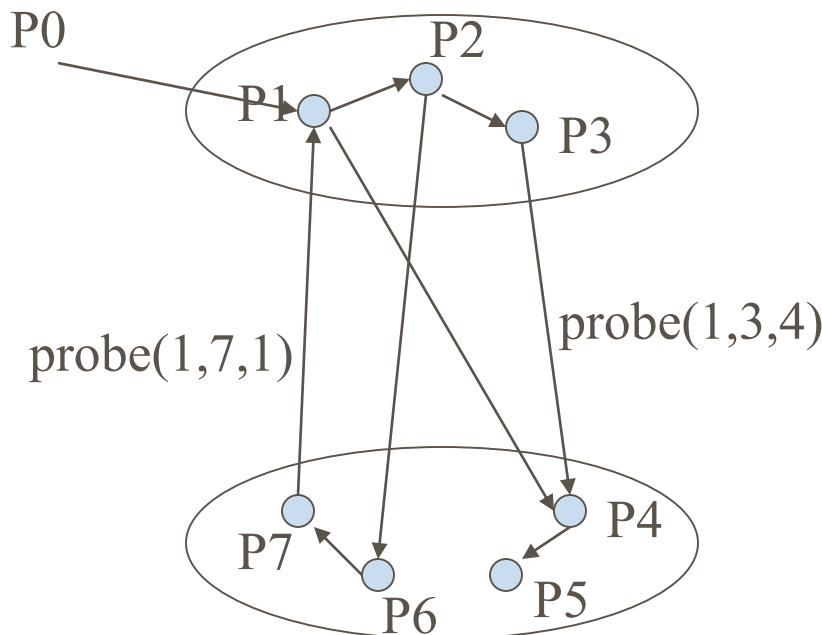
- Chandy-Misra-Haas's distributed deadlock detection algorithm for AND model is based on edge-chasing.
- The algorithm uses a special message called *probe*, which is a triplet  $(i, j, k)$ , denoting that it belongs to a deadlock detection initiated for process  $P_i$  and it is being sent by the home site of process  $P_j$  to the home site of process  $P_k$ .
- A probe message travels along the edges of the global WFG graph, and a deadlock is detected when a probe message returns to the process that initiated it.

# Edge-Chasing Algorithm

- Chandy-Misra-Haas's Algorithm:
  - A probe( $i, j, k$ ) is used by a deadlock detection process  $P_i$ . This probe is sent by the home site of  $P_j$  to  $P_k$ .
  - This probe message is circulated via the edges of the graph. Probe returning to  $P_i$  implies deadlock detection.



# C-M-H Algorithm: Example



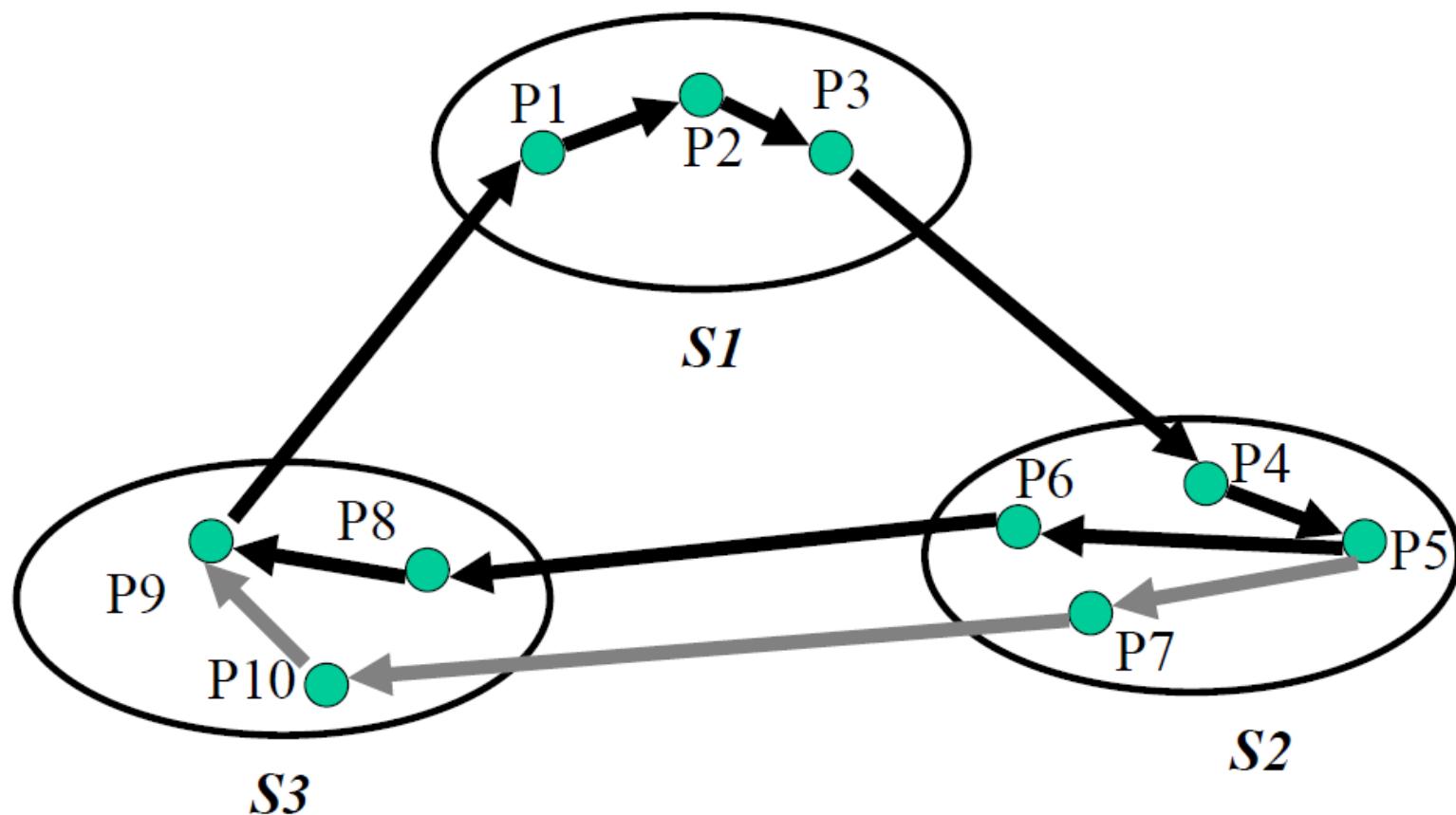
# Diffusion-based Algorithm

- Deadlock detection computations are diffused through the WFG of the system
  - **Query messages** are sent from a computation (process or thread) on a node and diffused across the edges of the WFG
  - When a **query** reaches an active (non-blocked) computation the query is discarded, but when a query reaches a blocked computation the query is echoed back to the originator when (and if) **all outstanding queries** of the blocked computation are returned to it
  - If all **queries** sent are echoed back to an initiator, there is deadlock

# Diffusion-based Algorithm

- A waiting computation on node x periodically sends a **query** to all computations it is waiting for (**the dependent set**), marked with the originator ID and target ID
- Each of these computations in turn will **query** their dependent set members (only if they are blocked themselves) marking each query with the **originator ID**, their own ID and a new target ID they are waiting on
- A computation cannot echo a reply to its requestor **until it has received replies from its entire dependent set**, at which time it sends a reply marked with the **originator ID**, its own ID and the **dependent (target) ID**
- When (and if) the original requestor receives echo replies from all members of its dependent set, it can **declare a deadlock** when an echo reply's **originator ID and target ID** are its own

# Diffusion Computation of Chandy et al



# Diffusion Computation of Chandy et al

P1 => P2 message at P2 from P1 (P1, P1, P2)

P2 => P3 message at P3 from P2 (P1, P2, P3)

P3 => P4 message at P4 from P3 (P1, P3, P4)

P4 => P5                          ETC.

P5 => P6

P5 => P7

P6 => P8

P7 => P10

*end condition*

P8 => P9 (P1, P8, P9), **now reply** (P1, P9, P1)

P10 => P9 (P1, P10, P9), **now reply** (P1, P9, P1)

---

P8 <= P9 **reply** (P1, P9, P8)

P10 <= P9 **reply** (P1, P9, P10)

P6 <= P8 **reply** (P1, P8, P6)

P7 <= P10 **reply** (P1, P10, P7)

P5 <= P6                          ETC.

P5 <= P7

P4 <= P5 ←

P3 <= P4

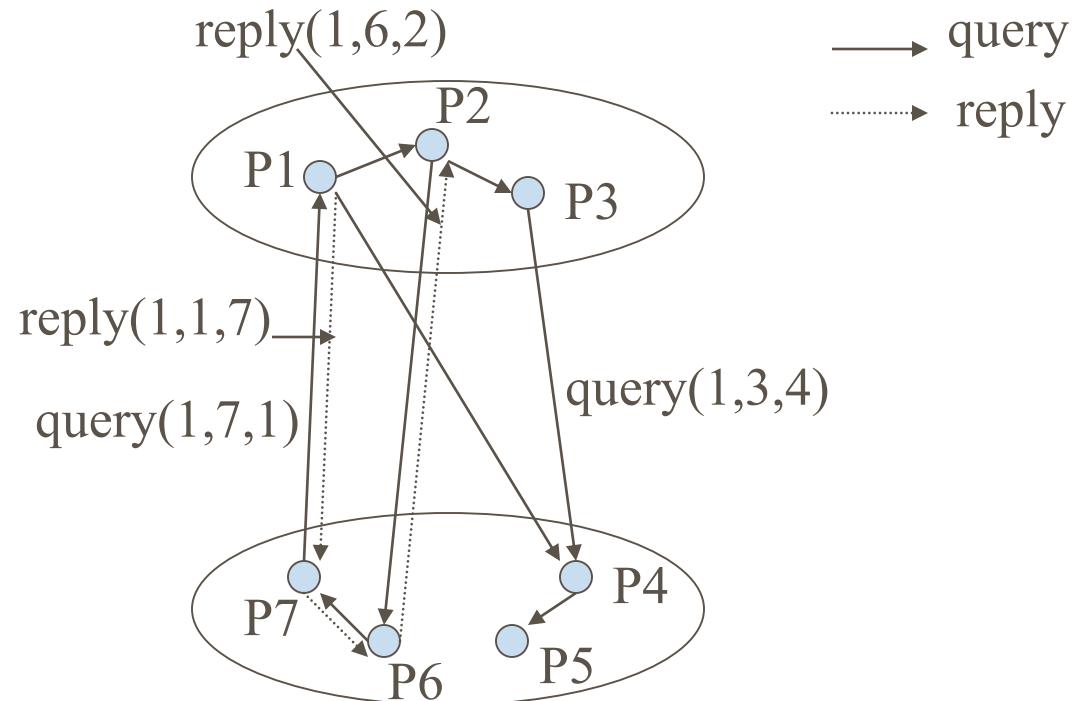
P2 <= P3

P1 <= P2 **reply** (P1, P2, P1)

*deadlock condition*

***P5 cannot reply until both P6 and P7 replies arrive !***

# Diffusion Algorithm: Example



# Engaging Query

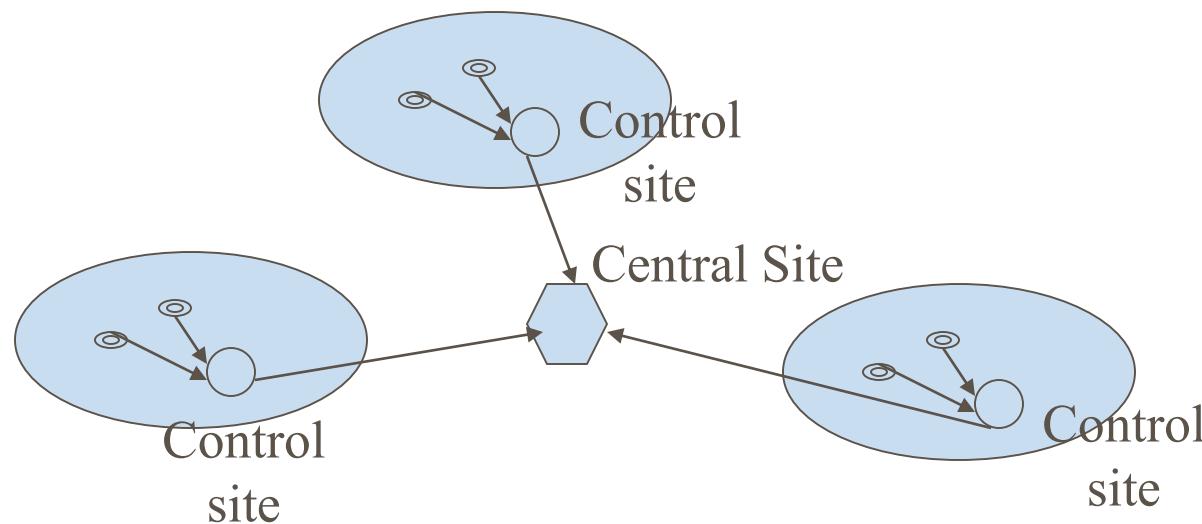
- How to distinguish an engaging query?
  - $\text{query}(i,j,k)$  from the initiator contains a unique sequence number for the query apart from the tuple  $(i,j,k)$ .
  - This sequence number is used to identify subsequent queries.
  - (e.g.,) when  $\text{query}(1,7,1)$  is received by P1 from P7, P1 checks the sequence number along with the tuple.
  - P1 understands that the query was initiated by itself and it is not an engaging query.
  - Hence, P1 sends a reply back to P7 instead of forwarding the query on all its outgoing links.

# AND, OR Models

- AND Model
  - A process/transaction can simultaneously request for multiple resources.
  - Remains blocked until it is granted *all* of the requested resources.
  - Edge-chasing algorithm can be applied here.
- OR Model
  - A process/transaction can simultaneously request for multiple resources.
  - Remains blocked till *any one* of the requested resource is granted.
  - Diffusion based algorithm can be applied here.

# Hierarchical Deadlock Detection

- Follows Ho-Ramamoorthy's 1-phase algorithm. More than 1 control site organized in hierarchical manner.
- Each control site applies 1-phase algorithm to detect (intracluster) deadlocks.
- Central site collects info from control sites, applies 1-phase algorithm to detect intracluster deadlocks.



# Persistence & Resolution

- Deadlock persistence:
  - Average time a deadlock exists before it is resolved.
- Implication of persistence:
  - Resources unavailable for this period: affects utilization
  - Processes wait for this period unproductively: affects response time.
- Deadlock resolution:
  - Aborting at least one process/request involved in the deadlock.
  - Efficient resolution of deadlock requires knowledge of all processes and resources.
  - If every process detects a deadlock and tries to resolve it independently -> highly inefficient ! Several processes might be aborted.

# Deadlock Resolution

- Priorities for processes/transactions can be useful for resolution.
  - Consider priorities
  - Highest priority process initiates and detects deadlock (initiations by lower priority ones are suppressed).
  - When deadlock is detected, lowest priority process(es) can be aborted to resolve the deadlock.
- After identifying the processes/requests to be aborted,
  - All resources held by the victims must be released. State of released resources restored to previous states. Released resources granted to deadlocked processes.
  - All deadlock detection information concerning the victims must be removed at all the sites.

# Load Balancing & Termination Detection

# Load Balancing

- The process of spreading the tasks evenly across the processors that would lead in the minimum execution time is called load balancing
  - Static load balancing (also called as mapping or scheduling problem)
    - Attempted before the execution of any process
  - Dynamic load balancing
    - Attempted dynamically during the execution of the process

# Issues in Load Distribution

- Load
  - CPU queue length as a load indicator
  - CPU utilization as a load indicator

# Classification of Load Distribution Algorithms

- Goal of Load distributing algorithm
  - To transfer load from heavily loaded computers to idle or lightly loaded computers
- broadly characterized as
  - Static
    - Decision is hard wired in the algorithm using a priory knowledge of the system
  - Dynamic
    - Make use of system state information to make load distributing decisions

# **Classification of Load Distribution Algorithms contd..**

- Adaptive
  - Special class of dynamic algorithm
  - they adapt their activities by dynamically changing the parameters of the algorithm to suit the changing system state

# Preemptive vs Nonpreemptive transfers

- Preemptive:
  - Transfer of a task that is partially executed
- Non preemptive:
  - Transfer of a task that has not yet started execution

# Components of Load Distribution Algorithms

- Four components
  - Transfer policy
    - Determines whether a node is in a suitable state to participate in a task transfer
  - Selection policy
    - determines which task should be transferred
  - Location policy
    - determines to which node a task selected for transfer should be sent
  - Information policy
    - responsible for triggering the collection of system state information

# Transfer Policy

- Threshold policy
  - Thresholds are expressed in terms of units of load
  - Decided upon the origination of new task
  - Concept of sender and receiver
- On detecting imbalance in load amongst nodes in system

# Selection Policy

- Selects a task for transfer
- Simplest approach: to select the newly originated task
- Overhead incurred in task transfer should be compensated by the reduction in the response time realized by the task
- Factors to consider:
  - Overhead incurred by transfer should be minimal
  - Number of location dependent system calls made by the selected task should be minimal

# Location Policy

- To find suitable nodes to share load(sender or receiver)
- Widely used method : polling
  - Either serially or in parallel
  - Either randomly or on a nearest-neighbor basis
- Alternative to polling
  - Broadcast a query to find out if any node is available for load sharing

# Information Policy

- To decide when, where and what information about the states of other nodes on the system should be collected
- One of three types:
  - Demand driven
    - Node collects the state of the other nodes only when it becomes either a sender or a receiver
    - dynamic policy
    - can be sender-initiated, receiver-initiated or symmetrically initiated

# Information Policy

- Periodic
  - Nodes exchange load information periodically
  - Do not adapt their activity to the system state
  - Benefits are minimal at high system loads ???
- State change driven
  - Nodes disseminate state information whenever their state changes by a certain degree
  - Centralized and decentralized policy

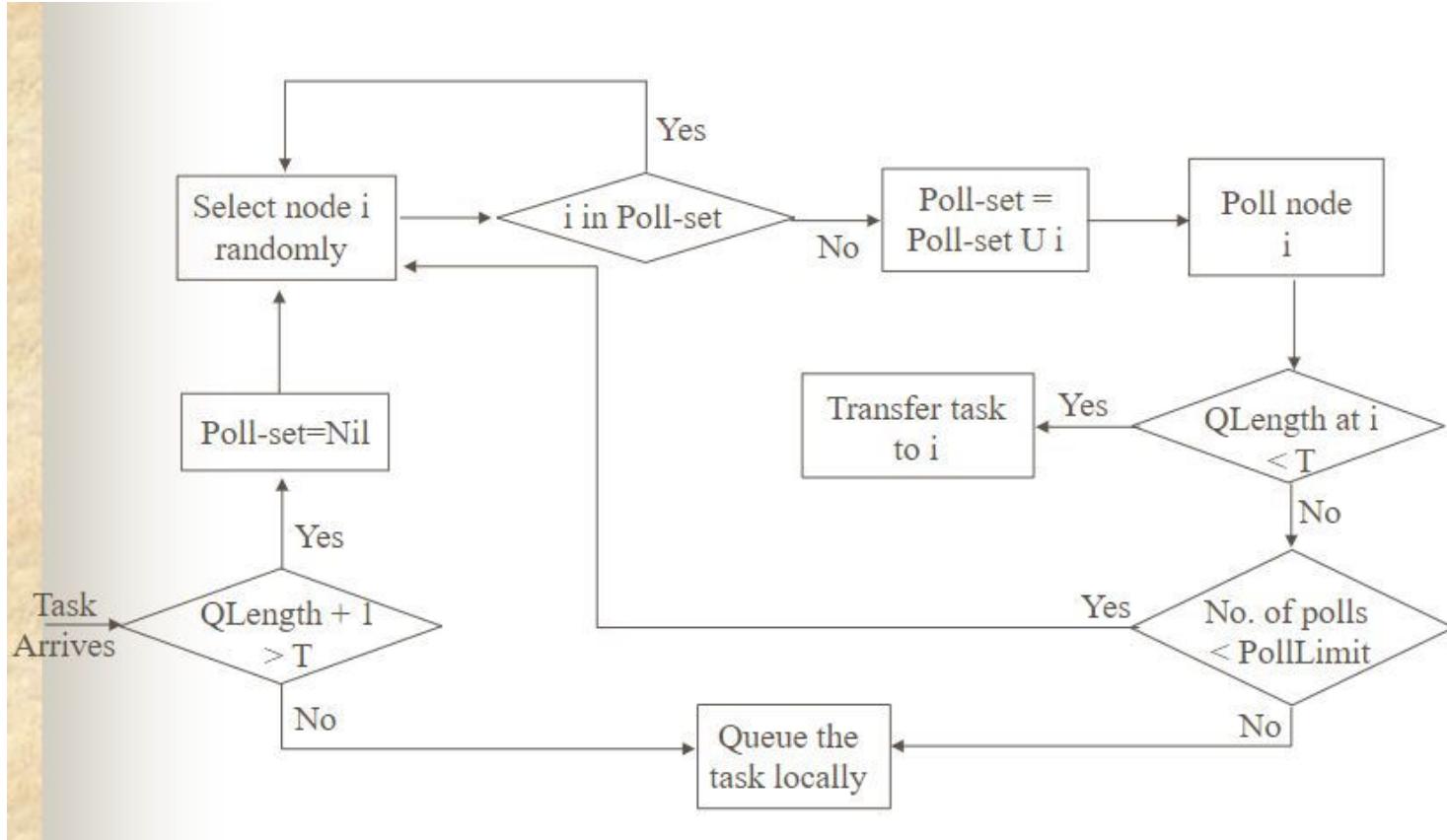
# Load distributing algorithms

- Sender initiated algorithms
- Receiver initiated algorithms
- Symmetrically initiated algorithms

# Sender Initiated algorithms

- Initiative by overloaded node (sender) to send a task to an underloaded node(receiver)
- Transfer policy :
  - Threshold policy based on CPU queue length
- Selection Policy:
  - Consider only newly arrived tasks for transfer
- Location policy:
  - Random :
    - no remote state information
    - task is transferred to a node selected at random

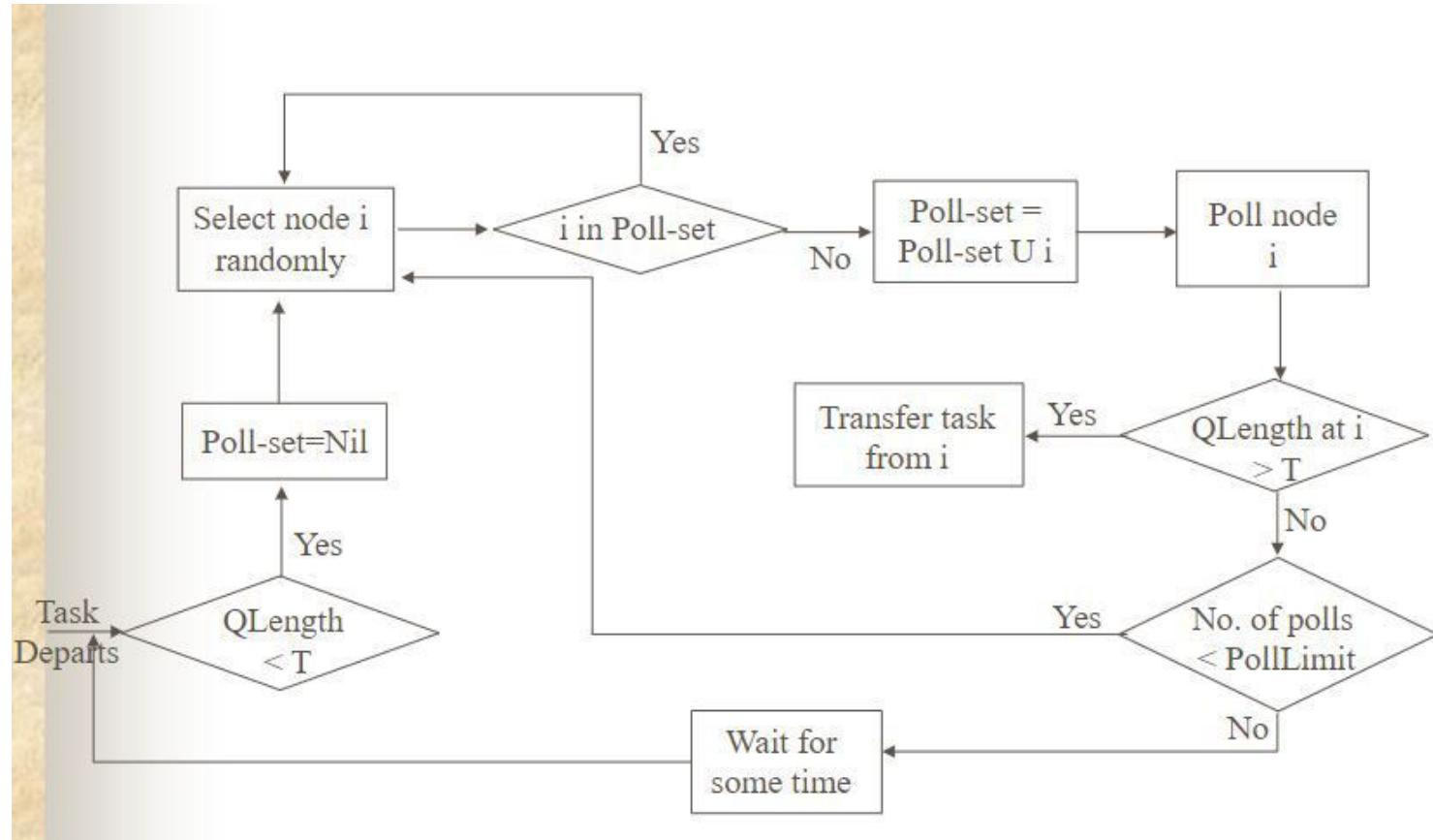
# Sender Initiated algorithms



# Receiver Initiated algorithms

- Initiation by an underloaded node (receiver)
- Transfer policy
  - Threshold policy based on CPU queue length.
  - Triggered when the task departs
- Selection policy
  - Any
- Location policy
  - Threshold policy

# Receiver Initiated algorithms



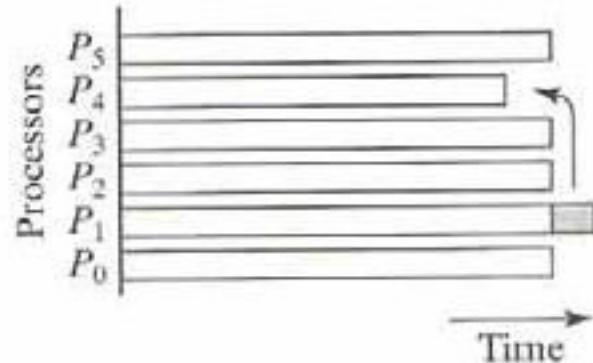
# Symmetrically Initiated algorithms

- Both senders and receivers search for receivers and senders respectively
- Advantages and disadvantages of both sender and receiver initiated algorithms.
- Above average algorithm.

# Static load balancing

- Some techniques are
  - Round robin algorithm : execution of the tasks in a round fashion
  - Randomized algorithms : selects processes at random to take tasks
  - Recursive bisection : recursively divides the problem into subproblems of equal computational effort while minimizing the message passing
  - Simulated annealing & Genetic algorithm : optimization techniques

# contd..



(a) Imperfect load balancing leading to increased execution time



(b) Perfect load balancing

# Problems with static load balancing

- Difficult to estimate the exact execution time of various parts of the program without actually executing the parts
- Communication delays encountered in the actual execution cannot be estimated in advance
- Example: searching in a large data space cannot be worked using static load balancing algorithms, needs dynamic load balancing

# Dynamic load balancing

- Most effective than static load balancing
- Division of load depends on execution of parts
- Incurs additional overhead but effective for parallel/distributed computations
- But **termination detection** algorithms are needed for the detection of end of computations

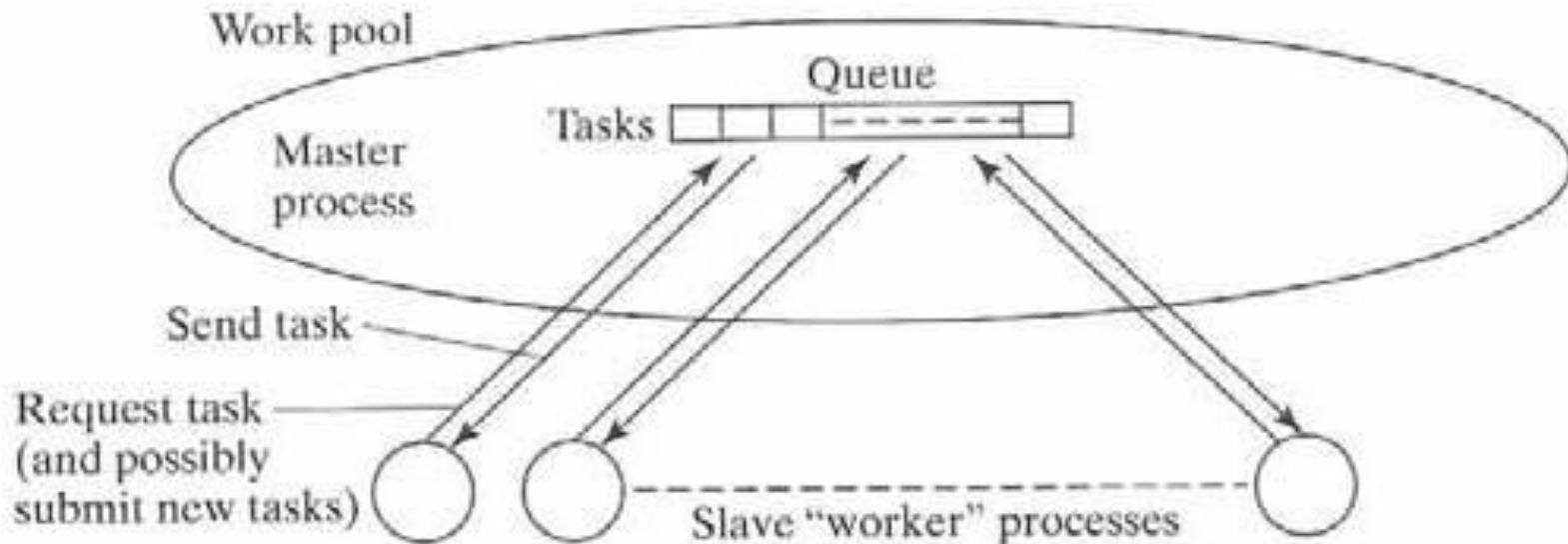
# contd..

- Dynamic load balancing types
  - Centralized
  - Decentralized
  - Distributed

# Centralized dynamic load balancing

- Called as work-pool approach
- Master-slave approach
- Can be applied to divide and conquer problems, or even if the tasks are different and of different sizes
- Larger tasks can be sent for execution first to balance the load
- Uses a queue to hold the waiting tasks (FIFO or priority based if needed)
- Even dynamic arrival of tasks can be handled by this approach

# contd..



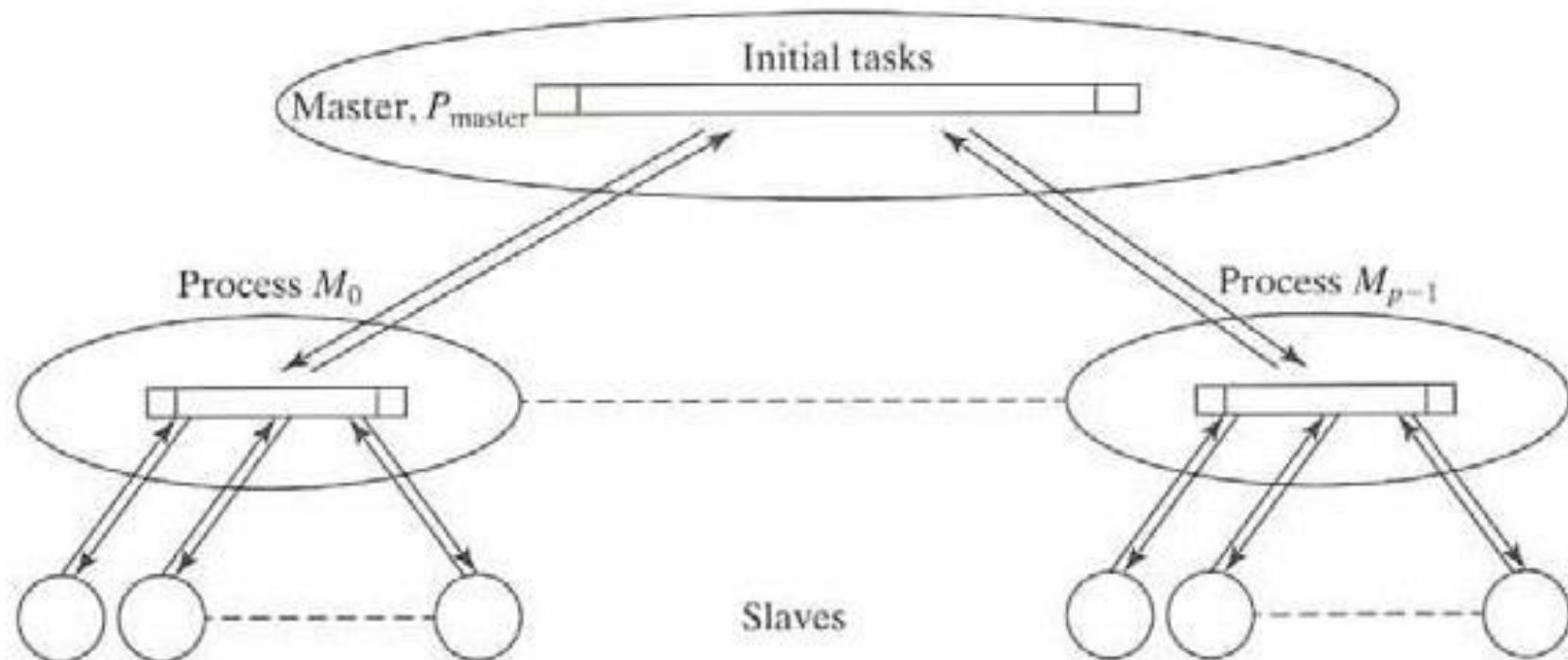
# Decentralized dynamic load balancing

- Problems with centralized approach
  - Master can issue the tasks only one at a time
  - Can respond to the requests only one after other
  - If the slaves are more in number, would incur overhead on the master process
  - Single point failure

## contd..

- Solution: distribute the work pool
- Each master has mini-masters to control the slaves
- To terminate, the slaves would reach the local termination, then the mini-masters and finally the master would terminate
- Works like divide and conquer approach

# contd..

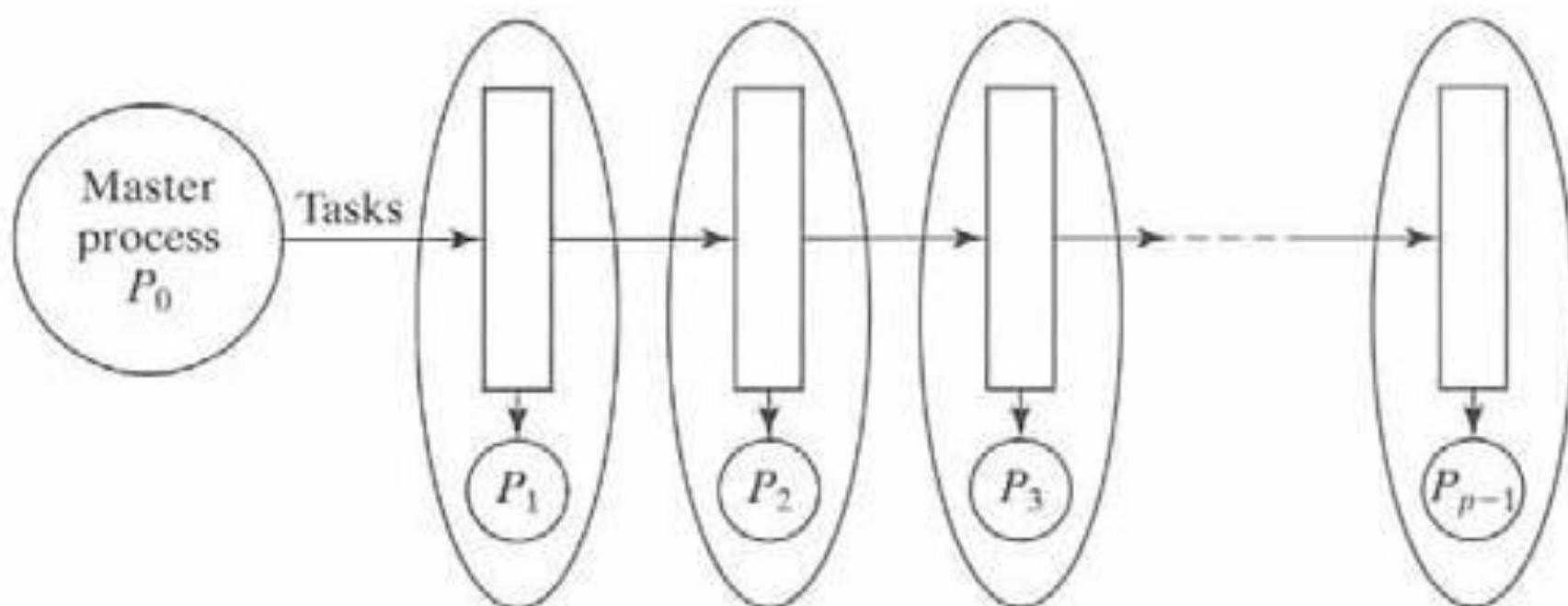


# Distributed load balancing

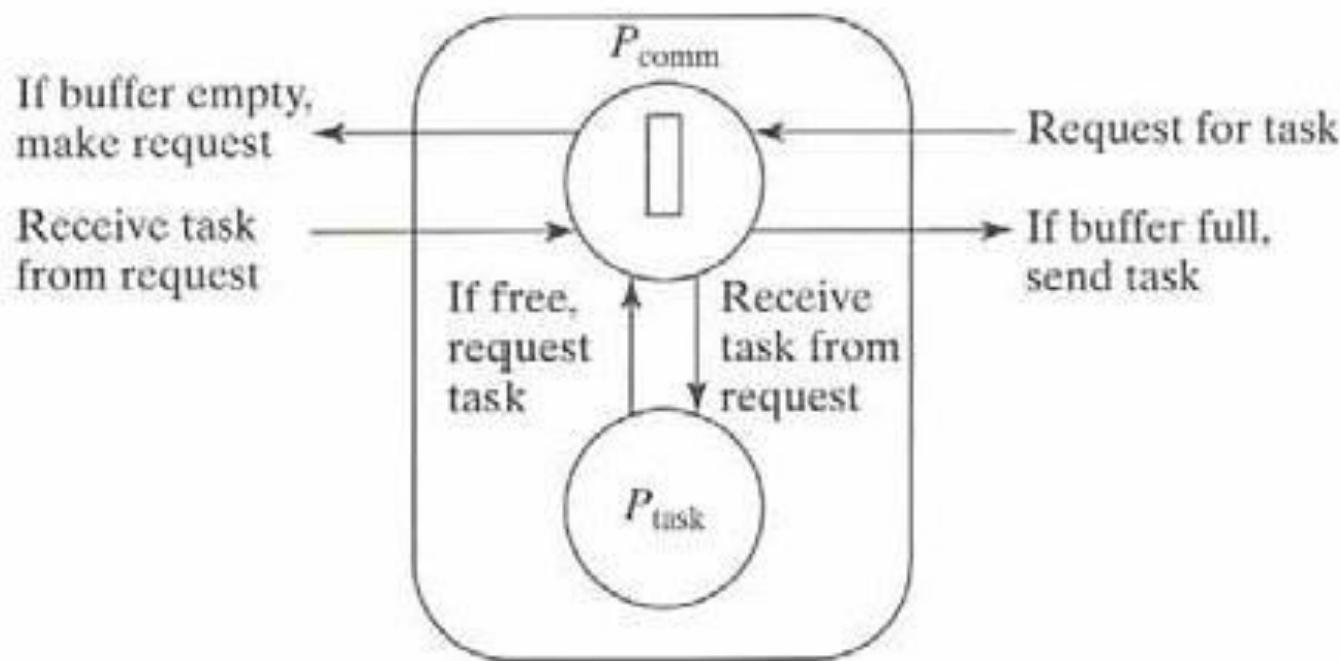
- Receiver-initiated
  - Process requests tasks from other processes when it has few or no tasks to execute
  - It probes the neighboring processes for certain numbers and either gets the task for execution or becomes idle for certain intervals and again starts probing
- Sender-initiated
  - Here the probing is done by the heavily loaded process
- Randomized polling algorithm
  - Selects a random number between 0 to  $p-1$  and requests or sends the probe

# Load balancing using line structure

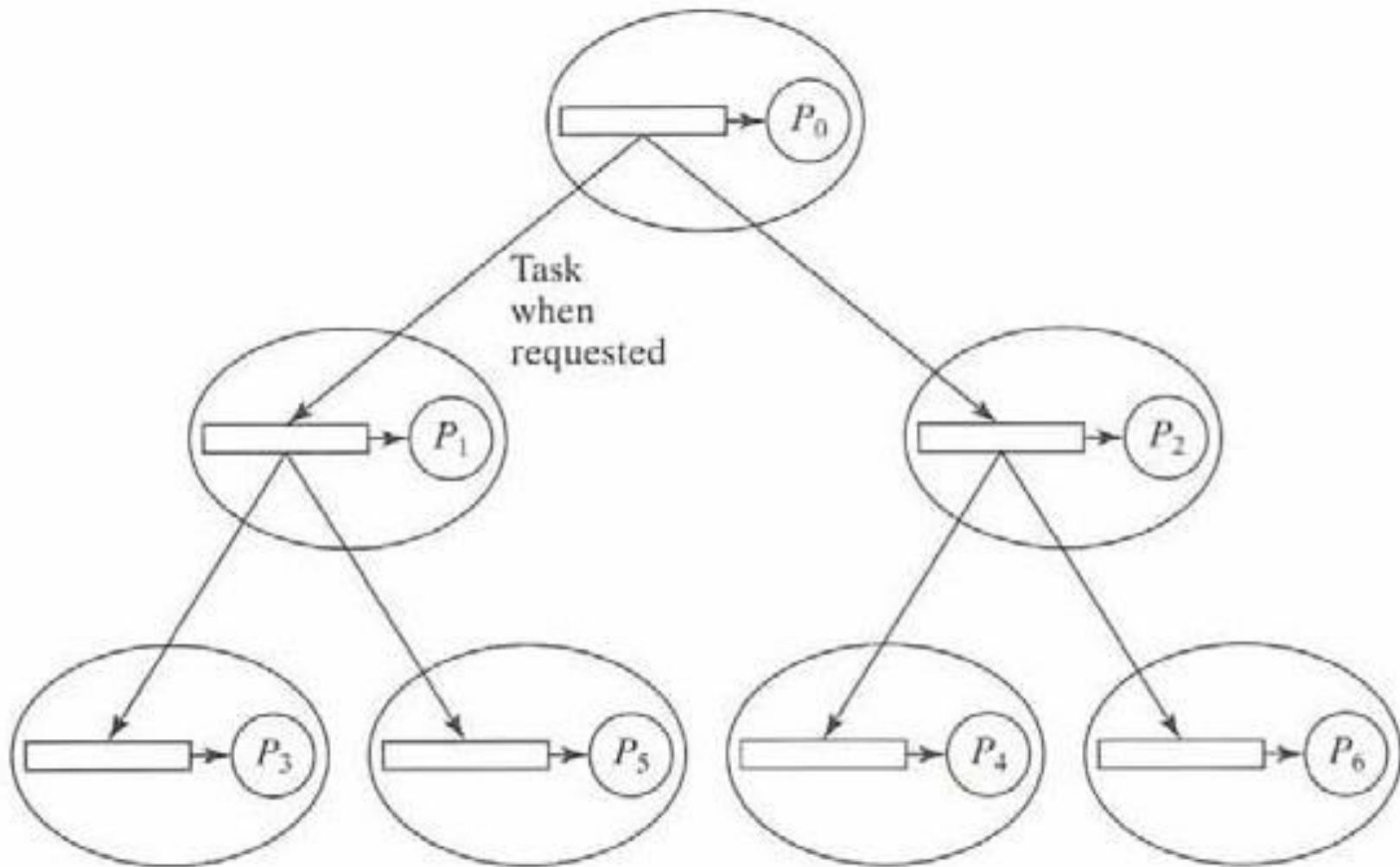
- Pipelining technique



# contd..



# Load balancing using tree structure



# Distributed termination detection algorithms

- Termination conditions
  - Helps in recognizing the end of the computations when the task scheduling is dynamic
  - The following conditions are to be satisfied for the termination detection at time  $t$ 
    - The process should attain the local termination at  $t$
    - There are no messages in transit between the processes at time  $t$ .

# contd..

- Termination detection
  - Computation terminated when both the following conditions are satisfied
    - The task queue is empty
    - Every slave process is idle and has made a request for another task without any new tasks being generated
  - In some applications like searching, a slave process on reaching the local termination would send a termination message to the master, and the master in turn would forcefully quit all the slave processes

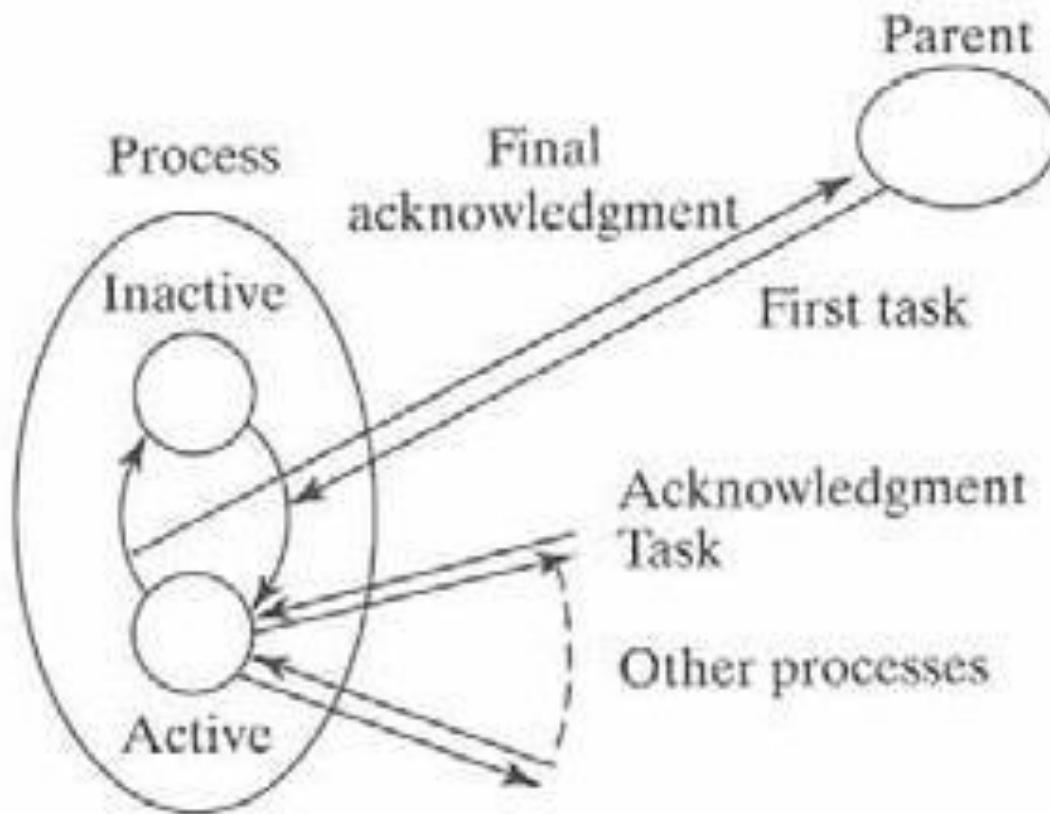
# Termination detection - types

- Using acknowledgement messages
  - Each process is in two states
    - Inactive (initially, without any task to perform)
    - Active (upon receiving a task from a process)
      - If a process passes on a task to an inactive process, it becomes the parent of the process and so on
      - In similar manner, creates a tree of processes, each with a unique parent
      - If a process sends a message to every process, it needs an acknowledgement message
      - If a process receives a task from its parent, it holds the acknowledgement message till the completion of the task and sends an acknowledgement message only when it is ready to become inactive

## contd..

- A process becomes inactive on the following conditions
  - Its local termination condition exists (all tasks are completed)
  - It has transmitted all the acknowledgements for tasks it has been received
  - It has received all its acknowledgements for tasks it has sent out
- Drawback: more messages sent/received

# contd..

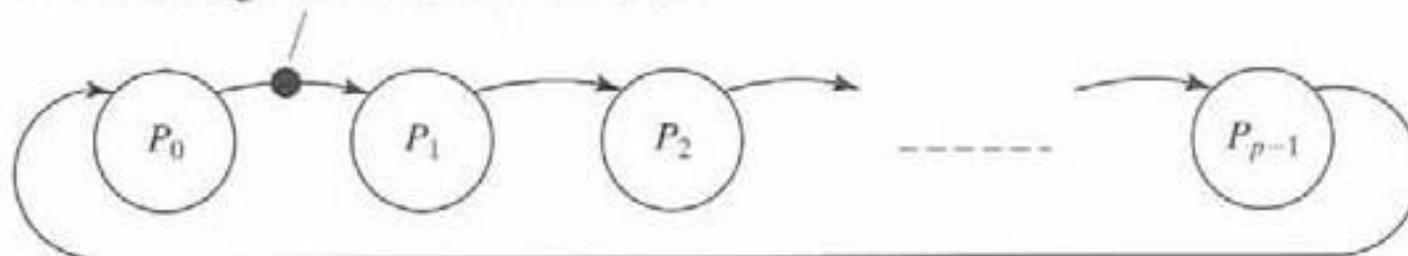


# Ring termination algorithms

- Single pass ring termination detection algorithm

1. When  $P_0$  has terminated, it generates a token that is passed to  $P_1$ .
2. When  $P_i$  ( $1 \leq i < p$ ) receives the token and has already terminated, it passes the token onward to  $P_{i+1}$ . Otherwise, it waits for its local termination condition and then passes the token onward.  $P_{p-1}$  passes the token to  $P_0$ .
3. When  $P_0$  receives a token, it knows that all the processes in the ring have terminated. A message can then be sent to all the processes informing them of the global termination, if necessary.

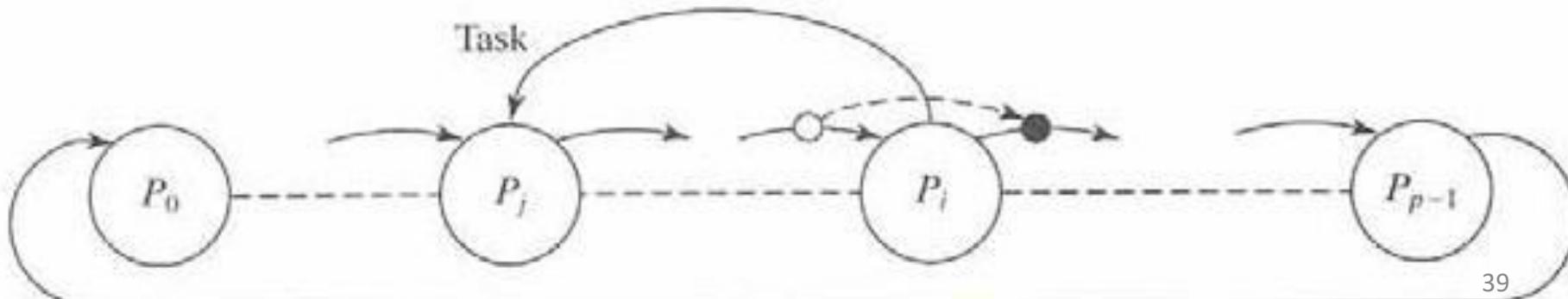
Token passed to next processor  
after reaching local termination condition



# contd..

- Dual pass ring algorithm

1.  $P_0$  becomes white when it has terminated and generates a white token to  $P_1$ .
2. The token is passed through the ring from one process to the next when each process has terminated, but the color of the token may be changed. If  $P_i$  passes a task to  $P_j$  where  $j < i$  (i.e., before this process in the ring), it becomes a *black process*; otherwise it is a *white process*. A black process will color a token black and pass it on. A white process will pass on the token in its original color (either black or white). After  $P_i$  has passed on a token, it becomes a white process.  $P_{p-1}$  passes the token to  $P_0$ .
3. When  $P_0$  receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.



# contd..

- Fixed energy distributed termination algorithm
  - Fixed energy is maintained by the master process
  - The master process passes out energy along with the tasks to processes making requests for tasks
  - The energy is further divided on every request and passed to the processes
  - When the process completes the tasks and ready to become idle, it transfers the energy to its parent or master
  - Upon receiving the energy, the master aggregates and terminates if the received energy is equal to the initial energy it had
- Drawback: floating point arithmetic may create problems at times, integer arithmetic can overcome this.

## Consistency Models for Distributed Shared Memory

### Strict Consistency

#### Strict consistency

- ① A Read should return the most recent value written, per a global time axis.  
For operations that overlap per the global time axis, the following must hold.
- ② All operations appear to be atomic and sequentially executed.
- ③ All processors see the same order of events, equivalent to the global time ordering of non-overlapping events.

$P_1: \frac{W(x)1}{}$   
 $P_2: \quad \quad \quad R(x)1$

(a)

$P_1: \frac{W(x)1}{}$   
 $P_2: \quad \quad \quad R(x)0 \quad R(x)1$

(b)

Behavior of two processes. The horizontal axis is time. (a) Strictly consistent memory. (b) Memory that is not strictly consistent.

### Sequential Consistency

#### Sequential Consistency.

- The result of any execution is the same as if all operations of the processors were executed in *some* sequential order.
- The operations of each individual processor appear in this sequence in the local program order.

$P_1: \frac{W(x)1}{}$   
 $P_2: \quad \quad \quad R(x)0 \quad R(x)1$

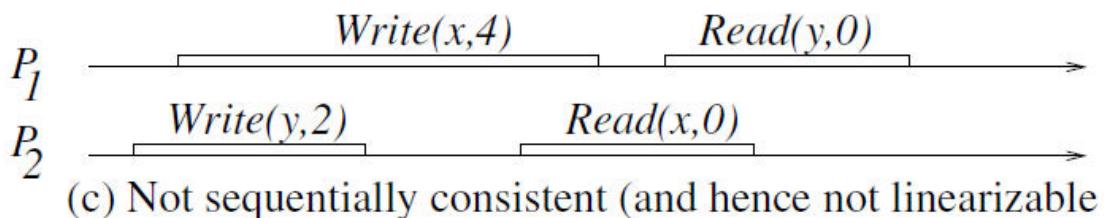
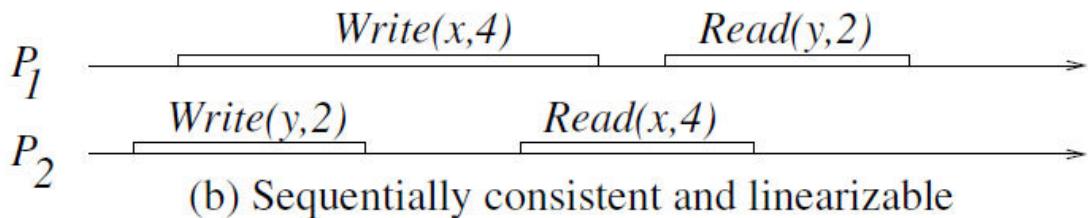
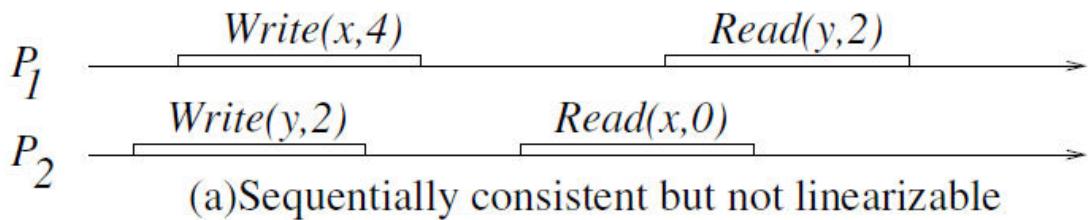
(a)

$P_1: \frac{W(x)1}{}$   
 $P_2: \quad \quad \quad R(x)1 \quad R(x)1$

(b)

Two possible results of running the same program.

## Strict Consistency / Linearizability: Examples



### Causal Consistency

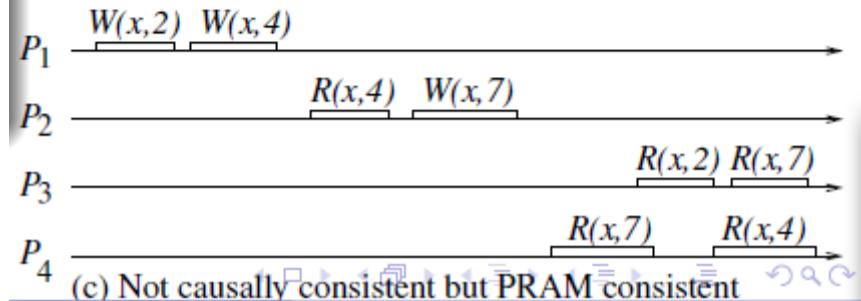
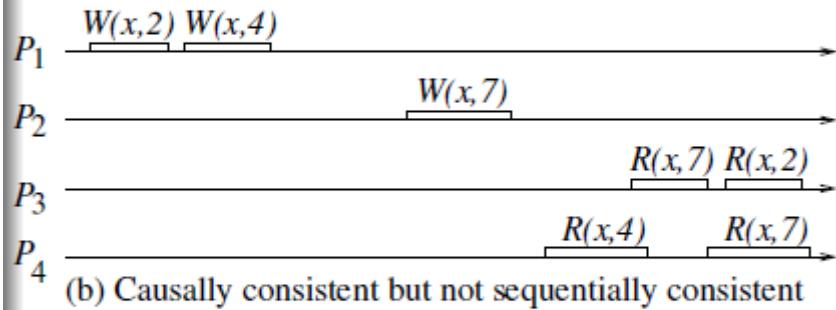
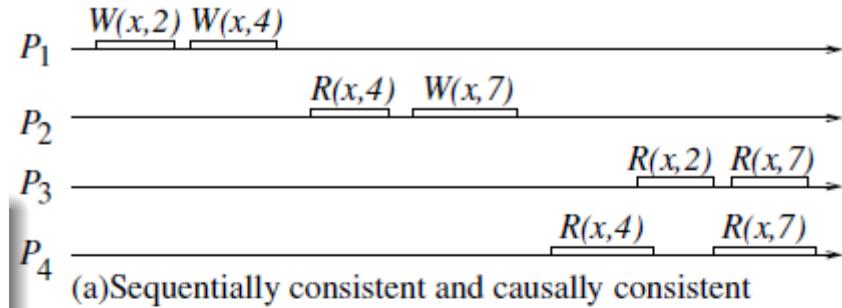
For *causal consistency*, only causally related Writes should be seen in common order.

#### Causal relation for shared memory systems

- At a processor, local order of events is the causal order
- A Write causally precedes Read issued by another processor if the Read returns the value written by the Write.
- The transitive closure of the above two orders is the causal order

P <sub>1</sub> :	<u>W(x)1</u>	<u>W(x)3</u>
P <sub>2</sub> :	<u>R(x)1</u>	<u>W(x)2</u>
P <sub>3</sub> :	<u>R(x)1</u>	<u>R(x)3</u> <u>R(x)2</u>
P <sub>4</sub> :	<u>R(x)1</u>	<u>R(x)2</u> <u>R(x)3</u>

This sequence is allowed with causally consistent memory, but not with sequentially consistent memory or strictly consistent memory.



### PRAM Consistency (Pipelined RAM)

#### PRAM memory

Only Write ops issued by the same processor are seen by others in the order they were issued, but Writes from different processors may be seen by other processors in different orders.

P <sub>1</sub> :	W(x)1
P <sub>2</sub> :	R(x)1 W(x)2
P <sub>3</sub> :	R(x)2 R(x)1
P <sub>4</sub> :	R(x)1 R(x)2

(a)

P <sub>1</sub> :	W(x)1
P <sub>2</sub> :	W(x)2
P <sub>3</sub> :	R(x)2 R(x)1
P <sub>4</sub> :	R(x)1 R(x)2

(b)

(a) A violation of causal memory. (b) A correct sequence of events in causal memory.

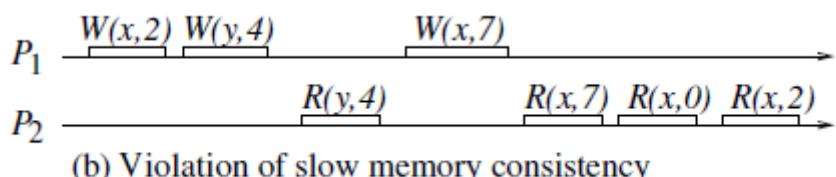
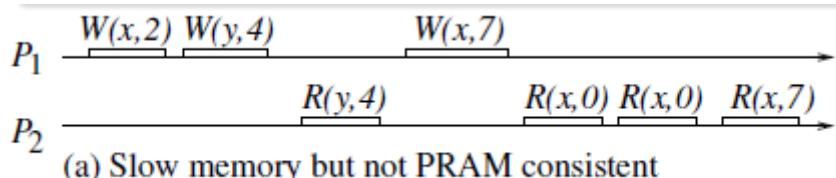
P <sub>1</sub> :	W(x)1
P <sub>2</sub> :	R(x)1 W(x)2
P <sub>3</sub> :	R(x)1 R(x)2
P <sub>4</sub> :	R(x)2 R(x)1

A valid sequence of events for PRAM consistency.

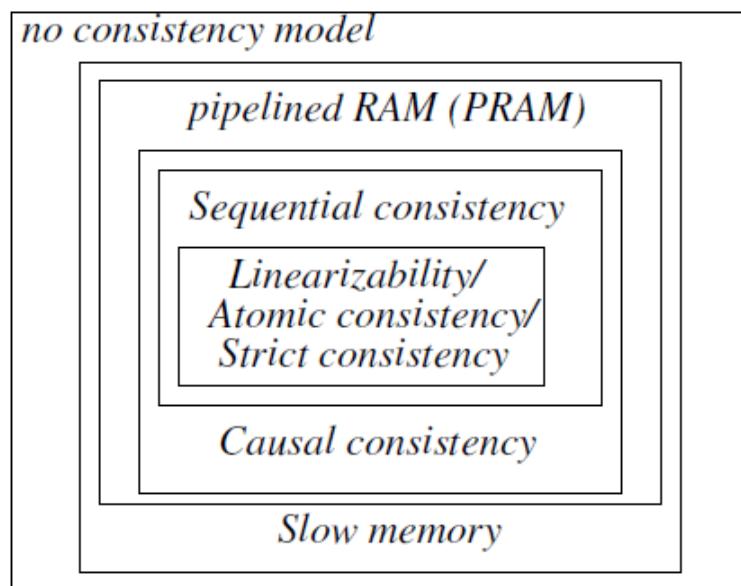
### Processor Consistency (Slow Memory)

#### Slow Memory

Only Write operations issued by the same processor and to the same memory location must be seen by others in that order.



# Hierarchy of Consistency Models



## Synchronization based Consistency Models

- Consistency conditions apply only to special "synchronization" instructions, e.g., barrier synchronization
- Non-sync statements may be executed in any order by various processors.
- E.g., weak consistency, release consistency, entry consistency

### Weak Consistency

P <sub>1</sub> :	W(x)1	W(x)2	S
P <sub>2</sub> :		R(x)1	R(x)2 S
P <sub>3</sub> :		R(x)2	R(x)1 S

(a)

P <sub>1</sub> :	W(x)1	W(x)2	S
P <sub>2</sub> :			S R(x)1

(b)

(a) A valid sequence of events for weak consistency. (b) An invalid sequence for weak consistency.

## Weak consistency:

All Writes are propagated to other processes, and all Writes done elsewhere are brought locally, at a sync instruction.

## Release Consistency

### Release Consistency

- *Acquire* indicates CS is to be entered. Hence all *Writes* from other processors should be locally reflected at this instruction
- *Release* indicates access to CS is being completed. Hence, all *Updates* made locally should be propagated to the replicas at other processors.
- *Acquire* and *Release* can be defined on a subset of the variables.
- If no CS semantics are used, then *Acquire* and *Release* act as barrier synchronization variables.
- Lazy release consistency: propagate updates on-demand, not the PRAM way.

P <sub>1</sub> :	Acq(L) W(x)1 W(x)2 Rel(L)
P <sub>2</sub> :	Acq(L) R(x)2 Rel(L)
P <sub>3</sub> :	R(x)1

A valid event sequence for release consistency.

## Entry Consistency

Same as Release Consistency, except that the lock is separate for every shared variable

### Entry Consistency

- Each ordinary shared variable is associated with a synchronization variable (e.g., lock, barrier)
- For Acquire /Release on a synchronization variable, access to only those ordinary variables guarded by the synchronization variables is performed.

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters
Sequential	All processes see all shared accesses in the same order
Causal	All processes see all causally-related shared accesses in the same order
Processor	PRAM consistency + memory coherence
PRAM	All processes see writes from each processor in the order they were issued. Writes from different processors may not always be seen in the same order

(a)

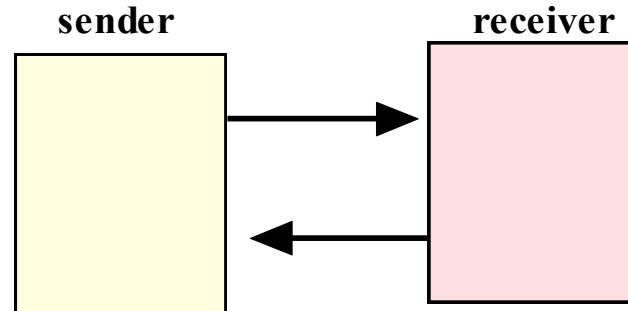
Weak	Shared data can only be counted on to be consistent after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered

(b)

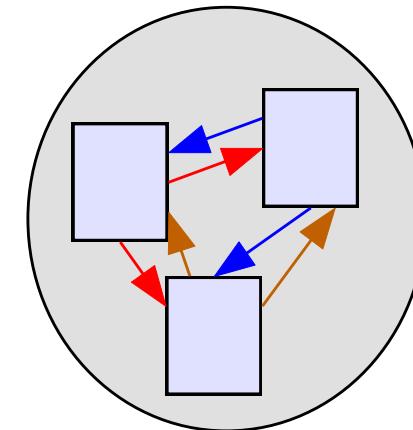
Consistency models not using synchronization operations. (b)  
Models with synchronization operations.

# Group Communication

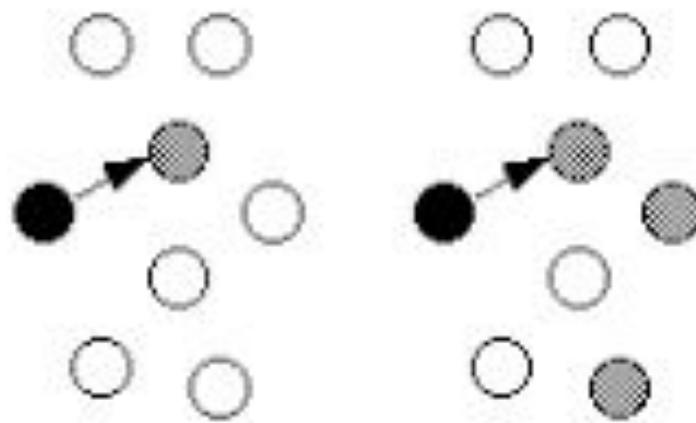
# Unicast vs. Multicast



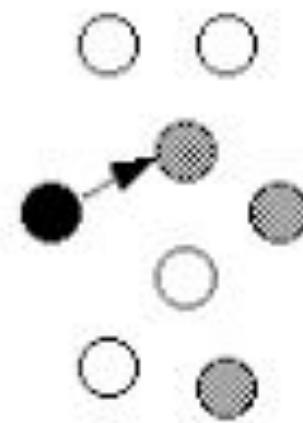
One-to-one communication or unicast



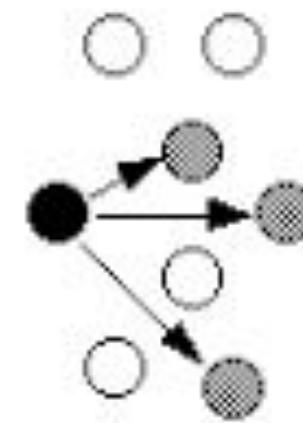
Group communication or multicast



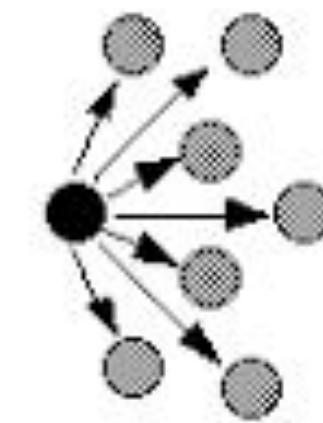
Unicast



Anycast



Multicast



Broadcast

# Multicast

- Whereas the majority of network services and network applications use unicast for IPC, multicasting is useful for applications such as:

online conferences, interactive distance learning, and can be used for applications such as online auction. It can also be used in replication of services for fault tolerance.

# Mutlicast group

- In an application or network service which makes use of multicasting, a set of processes form a group, called a ***multicast group***. Each process in a group can send and receive message. A message sent by any process in the group can be received by each participating process in the group. A process may also choose to leave a multicast group.
- In an application such as online conferencing, a group of processes interoperate using multicasting to exchange audio, video, and/or text data.

# Multicast API

Primitive operations:

- **Join** – This operation allows a process to join a specific multicast group. A process that has joined a multicast group is a member of the group and is entitled to receive all multicast addressed to the group. A process should be able to be a member of multiple multicast groups at any one time.

# Multicast API Operations - continued

- **Leave** – This operation allows a process to stop participating in a multicast group. A process that has left a multicast group is no longer a member of the group and is thereafter not entitled to receive any multicast addressed to the group, although the process may remain a member of other multicast groups.
- **Send** – This operation allows a process to send a message to all processes currently participating in a multicast group.
- **Receive** – This operation allows a member process to receive messages sent to a multicast group.

# Reliable Multicast vs. Unreliable Multicast

- When a multicast message is sent by a process, the runtime support of the multicast mechanism is responsible for delivering the message to each process currently in the multicast group.
- Due to factors such as failures of network links and/or network hosts, routing delays, and differences in software and hardware, the time between when a unicast message is sent and when it is received may vary among the recipient processes.

# Reliable Multicast vs. Unreliable Multicast

- Moreover, a message may not be received by one or more of the processes at all, due to errors and/or failures in the network, the machines, or the runtime support.
- Whereas some applications, such as video conferencing, can tolerate an occasional miss or misordering of messages, there are applications – such as database applications – for which such anomalies are unacceptable.
- Therefore, when employing a multicasting mechanism for an application, it is important that you choose one with the characteristics appropriate for your application. Otherwise, measures will need to be provided in the coding of the application in order to handle the anomalies which may occur in message delivery.

# Classification of multicasting mechanisms in terms of message delivery -1

**Unreliable multicast:** At its most basic, a multicast system will make a good-faith attempt to deliver messages to each participating process, but the arrival of the correct message at each process is not guaranteed.

**In the best case, the correct message is received by all processes.**

- **In some cases, the message may be received by some but not all,**
- **The messages may be received by some processes in a corrupted form.** Such a system is said to provide **unreliable multicast**.

## Classification of multicasting mechanisms in terms of message delivery - 2

**Reliable multicast:** A multicast system which guarantees that each message is eventually delivered to each process in the group in uncorrupted form is said to provide **reliable multicast**.

## Classification of multicasting mechanisms in terms of message delivery - 3

- The definition of reliable multicast requires that each participating process receives exactly one copy of each message sent. However, the definition places **no restriction on the order** that the messages are delivered to each process: each process may receive the messages in any permutation of those messages.
- For applications where the order of message delivery is significant, it is helpful to further classify reliable multicast systems based on the order of the delivery of messages.

# Classification of reliable multicast

## Unordered

An unordered reliable multicast system provides no guarantee on the delivery order of the messages.

**Example:** Processes  $P_1$ ,  $P_2$ , and  $P_3$  have formed a multicast group. Further suppose that three messages,  $m_1$ ,  $m_2$ ,  $m_3$  have been sent to the group. Then an unordered reliable multicast system may deliver the messages to each of the three processes in any of the  $3! = 6$  permutations ( $m_1-m_2-m_3$ ,  $m_1-m_3-m_2$ ,  $m_2-m_1-m_3$ ,  $m_2-m_3-m_1$ ,  $m_3-m_1-m_2$ ,  $m_3-m_2-m_1$ ). Note that it is possible for each participant to receive the messages in an order different from the orders of messages delivered to other participants.

## Classification of reliable multicast - 2

### FIFO multicast

A system which guarantees that the delivery of the messages adhere to the following condition is said to provide FIFO (first-in-first-out) or send-order multicast:

If process  $P$  sent messages  $m_i$  and  $m_j$ , in that order, then each process in the multicast group will be delivered the messages  $m_i$  and  $m_j$ , in that order.

Suppose  $P_1$  sends messages  $m_1$ ,  $m_2$ , and  $m_3$  in order, then each process in the group is guaranteed to have those messages delivered in that same order:  $m_1$ ,  $m_2$ , then  $m_3$ .

## Classification of reliable multicast - 2

Note that FIFO multicast places no restriction on the delivery order among messages sent by different processes.

To illustrate the point, let us use a simplified example of a multicast group of two processes:  $P_1$  and  $P_2$ . Suppose  $P_1$  sends messages  $m_{11}$  then  $m_{12}$ , while  $P_2$  sends messages  $m_{21}$  then  $m_{22}$ . Then a FIFO multicast system can deliver the messages to each of the two processes in any of the following orders:

$m_{11}-m_{12}-m_{21}-m_{22}$ ,

$m_{11}-m_{21}-m_{12}-m_{22}$ ,

$m_{11}-m_{21}-m_{22}-m_{12}$ ,

$m_{21}-m_{11}-m_{12}-m_{22}$

$m_{21}-m_{11}-m_{22}-m_{12}$

$m_{21}-m_{22}-m_{11}-m_{12}$ .

# Classification of reliable multicast - 3

## Causal Order Multicast

A multicast system is said to provide **causal** multicast if its message delivery satisfies the following criterion:

If message  $m_i$  causes (results in) the occurrence of message  $m_j$ , then  $m_i$  will be delivered to each process prior to  $m_j$ . Messages  $m_i$  and  $m_j$  are said to have a causal or happen-before relationship, denoted  $m_i \rightarrow m_j$ . The happen-before relationship is **transitory**: if  $m_i \rightarrow m_j$  and  $m_j \rightarrow m_k$ , then  $m_i \rightarrow m_j \rightarrow m_k$ . In this case, a causal-order multicast system guarantees that these three messages will be delivered to each process in the order of  $m_i$ ,  $m_j$ , then  $m_k$ .

## Causal Order Multicast – example1

Suppose three processes  $P_1$ ,  $P_2$ , and  $P_3$  are in a multicast group.  $P_1$  sends a message  $m_1$ , to which  $P_2$  replies with a multicast message  $m_2$ . Since  $m_2$  is triggered by  $m_1$ , the two messages share a causal relationship of  $m_1 \rightarrow m_2$ . Suppose the receiving of  $m_2$  in turn triggers a multicast message  $m_3$  sent by  $P_3$ , that is,  $m_2 \rightarrow m_3$ . The three messages share the causal relationship of  $m_1 \rightarrow m_2 \rightarrow m_3$ . A causal-order multicast message system ensures that these three messages will be delivered to each of the three processes in the order of  $m_1$ -  $m_2$ -  $m_3$ .

## Causal Order Multicast – example2

As a variation of the above example, suppose  $P_1$  multicasts message  $m_1$ , to which  $P_2$  replies with a multicast message  $m_2$ , and independently  $P_3$  replies to  $m_1$  with a multicast message  $m_3$ . The three messages now share these causal relationships:  $m_1 \rightarrow m_2$  and  $m_1 \rightarrow m_3$ . A causal-order multicast system can delivery these message in either of the following orders:

$\textcolor{red}{m_1 - m_2 - m_3}$

$\textcolor{red}{m_1 - m_3 - m_2}$

since the causal relations are preserved in either of the two sequences. In such a system, it is not possible for the messages to be delivered to any of the processes in any other permutation of the three messages, such as  $m_2 - m_1 - m_3$  or  $m_3 - m_1 - m_2$ , the first of these violates the causal relationship  $m_1 \rightarrow m_2$ , while the second permutation violates the causal relationship  $m_1 \rightarrow m_3$ .

# Classification of reliable multicast - 4

## Atomic order multicast

In an atomic-order multicast system, all messages are guaranteed to be delivered to each participant in the exact same order. Note that the delivery order does not have to be FIFO or causal, but must be identical for each process.

Example:

$P_1$  sends  $m_1$ ,  $P_2$  sends  $m_2$ , and  $P_3$  sends  $m_3$ .

An atomic system will guarantee that the messages will be delivered to each process in only one of the six orders:

$m_1-m_2-m_3$ ,     $m_1-m_3-m_2$ ,     $m_2-m_1-m_3$ ,

$m_2-m_3-m_1$ ,     $m_3-m_1-m_2$ ,     $m_3-m_2-m_1$ .

# Atomic Multicast Example

$P_1$  sends  $m_1$  then  $m_2$ .

$P_2$  replies to  $m_1$  by sending  $m_3$ .

$P_3$  replies to  $m_3$  by sending  $m_4$ .

Although atomic multicast imposes no ordering on these messages, the sequence of the events dictates that  $P_1$  must deliver  $m_1$  before sending  $m_2$ . Likewise,  $P_2$  must receive  $m_1$  then  $m_3$ , while  $P_3$  must receive  $m_3$  before  $m_4$ . Hence any atomic delivery order must preserve the order  $m_1$ -  $m_3$ -  $m_4$ . The remaining message  $m_2$  can, however, be interleaved with these messages in any manner. Thus an atomic multicast will result in the messages being delivered to each of the processes in one of the following orders:  $m_1$ -  $m_2$ -  $m_3$ -  $m_4$ ,  $m_1$ -  $m_3$ -  $m_2$ -  $m_4$ , or  $m_1$ -  $m_3$ -  $m_4$ -  $m_2$ . For example, each process may be delivered the messages in this order  $m_1$ -  $m_3$ -  $m_2$ -  $m_4$ .

# Fault Tolerance

# Fault Tolerance

- A DS should be fault-tolerant
  - Should be able to continue functioning in the presence of faults
- Fault tolerance is related to **dependability**

# Dependability

## Dependability Includes

- Availability
- Reliability
- Safety
- Maintainability

# Availability & Reliability (1)

- **Availability:** A measurement of whether a system is *ready to be used immediately*
  - System is up and running at any given moment
- **Reliability:** A measurement of whether a system can *run continuously without failure*
  - System continues to function for a long period of time

# Availability & Reliability (2)

- A system goes down 1ms/hr has an availability of more than 99.99%, but is unreliable
- A system that never crashes but is shut down for a week once every year is 100% reliable but only 98% available

# Safety & Maintainability

- **Safety:** A measurement of *how safe failures are*
  - System fails, nothing serious happens
  - For instance, high degree of safety is required for systems controlling nuclear power plants
- **Maintainability:** A measurement of *how easy it is to repair a system*
  - A highly maintainable system may also show a high degree of availability
  - Failures can be detected and repaired automatically?  
Self-healing systems?

# Faults

- A system **fails** when it cannot meet its promises (specifications)
- An **error** is part of a system state that may lead to a failure
- A **fault** is the cause of the error
- **Fault-Tolerance:** the system can provide services even in the presence of faults
- Faults can be:
  - Transient (appear once and disappear)
  - Intermittent (appear-disappear-reappear behavior)
    - A loose contact on a connector → intermittent fault
  - Permanent (appear and persist until repaired)

# Failure Models

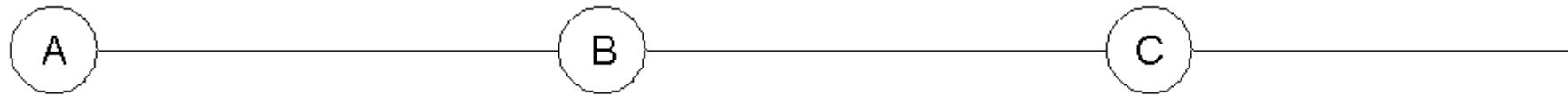
Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure <i>Receive omission</i> <i>Send omission</i>	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure <i>Value failure</i> <i>State transition failure</i>	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure (Byzantine failure)	A server may produce arbitrary responses at arbitrary times

# Failure Masking

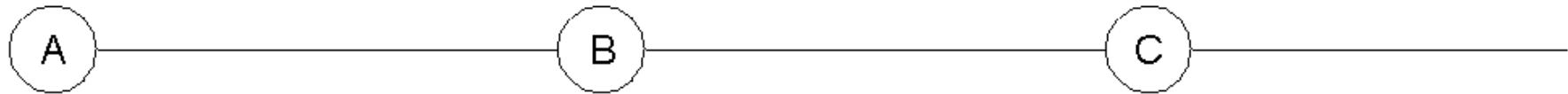


- Redundancy is key technique for hiding failures
- Redundancy types:
  1. Information: add extra (control) information
    - Error-correction codes in messages
  2. Time: perform an action persistently until it succeeds:
    - Transactions
  3. Physical: add extra components (S/W & H/W)
    - Process replication, electronic circuits

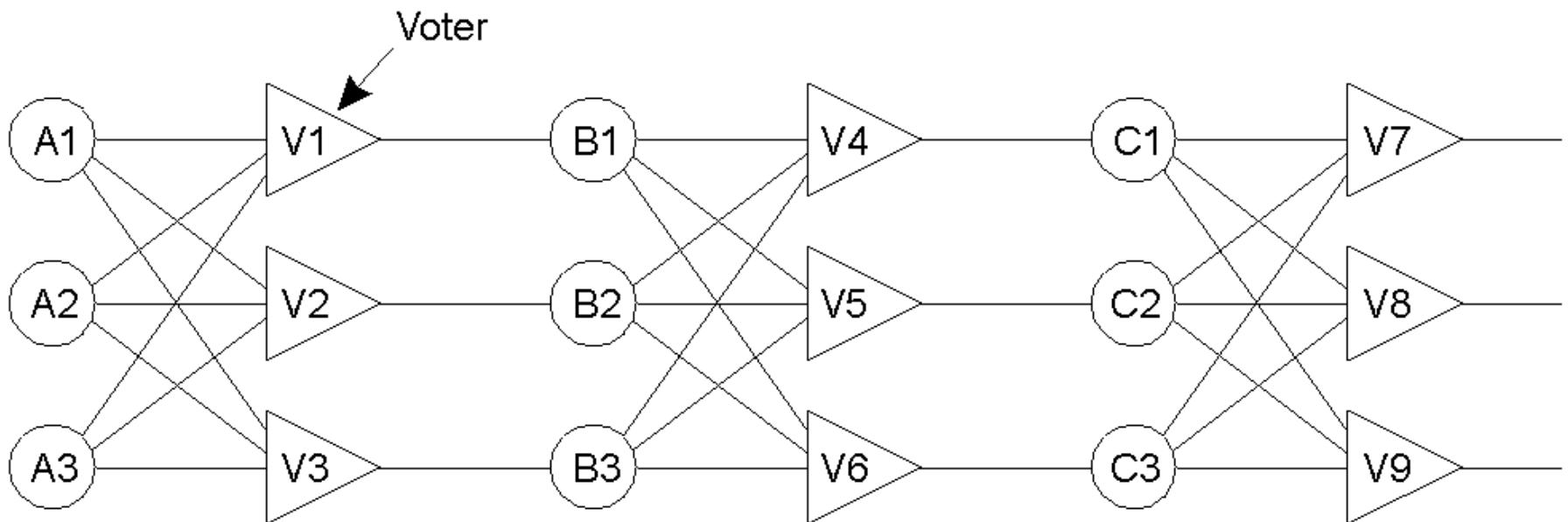
# Example – Redundancy in Circuits (1)



# Example – Redundancy in Circuits (2)



(a)

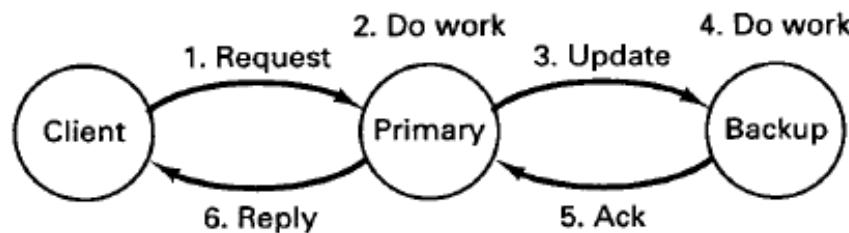


(b)

Triple modular redundancy.

# Process Resilience

- Mask process failures by replication
- Organize processes into groups, a message sent to a group is delivered to all members
- If a member fails, another should fill in



# Process Replication

- Replicate a process and group replicas in one group
- How many replicas do we create?
- A system is  $k$  fault-tolerant if it can survive and function even if it has  $k$  faulty processes
  - For crash failures (**a faulty process halts, but is working correctly until it halts**)
    - $k+1$  replicas
  - For Byzantine failures (**a faulty process may produce arbitrary responses at arbitrary times**)
    - $2k+1$  replicas

# Agreement Protocols

- When distributed systems engage in cooperative efforts like enforcing distributed mutual exclusion algorithms, processor failure can become a critical factor
- Processors may fail in various ways, and their failure modes and communication interfaces are central to the ability of healthy processors to detect and respond to such failures

# The System Model

- There are  $n$  processors in the system and at most  $m$  of them can be faulty
- The processors can directly communicate with other processors via messages (fully connected system)
- A receiver computation always knows the identity of a sending computation
- The communication system is pipelined and reliable

# Faulty Processors

- May fail in various ways
  - Drop out of sight completely
  - Start sending spurious messages
  - Start to lie in its messages (behave maliciously)
  - Send only occasional messages (fail to reply when expected to)
- May believe themselves to be healthy
- Are not known to be faulty initially by non-faulty processors

# Communication Requirements

- Synchronous model communication is assumed in this section:
  - Healthy processors receive, process and reply to messages in a lockstep manner
  - The receive, process, reply sequence is called a *round*
  - In the synch-comm model, processes know what messages they expect to receive in a round
- The synch model is critical to agreement protocols, and the agreement problem is not solvable in an asynchronous system

# Processor Failures

- Crash fault
  - Abrupt halt, never resumes operation
- Omission fault
  - Processor “omits” to send required messages to some other processors
- Malicious fault
  - Processor behaves randomly and arbitrarily
  - Known as *Byzantine faults*

## Authenticated vs. Non-Authenticated Messages

- Authenticated messages (also called *signed* messages)
  - assure the receiver of correct identification of the sender
  - assure the receiver the message content was not modified in transit
- Non-authenticated messages (also called *oral* messages)
  - are subject to intermediate manipulation
  - may lie about their origin

## Authenticated vs. Non-Authenticated Messages (cont'd)

- To be generally useful, agreement protocols must be able to handle non-authenticated messages
- The classification of agreement problems include:
  - The Byzantine agreement problem
  - The consensus problem
  - the interactive consistency problem

# Agreement Problems

<b>Problem</b>	<b>Who initiates value</b>	<b>Final Agreement</b>
<b>Byzantine Agreement</b>	One Processor	Single Value
<b>Consensus</b>	All Processors	Single Value
<b>Interactive Consistency</b>	All Processors	A Vector of Values

# Agreement Problems (cont'd)

- **Byzantine Agreement**
  - One processor broadcasts a value to all other processors
  - All non-faulty processors agree on this value, faulty processors may agree on any (or no) value
- **Consensus**
  - Each processor broadcasts a value to all other processors
  - All non-faulty processors agree on one common value from among those sent out. Faulty processors may agree on any (or no) value
- **Interactive Consistency**
  - Each processor broadcasts a value to all other processors
  - All non-faulty processors agree on the same vector of values such that  $v_i$  is the initial broadcast value of non-faulty *processor<sub>i</sub>* . Faulty processors may agree on any (or no) value

## Agreement Problems (cont'd)

- The **Byzantine Agreement** problem is a primitive to the other 2 problems
- The focus here is thus the **Byzantine Agreement** problem
- Lamport showed the first solutions to the problem
  - An initial broadcast of a value to all processors
  - A following set of messages exchanged among all (healthy) processors within a set of message rounds

# The Byzantine Agreement problem

- The upper bound on number of faulty processors:
  - It is impossible to reach a consensus (in a fully connected network) if the number of faulty processors  $m$  exceeds  $\lfloor (n - 1) / 3 \rfloor$  (from Pease et al)
  - Lamport et al were the first to provide a protocol to reach Byzantine agreement which requires  $m + 1$  rounds of message exchanges
  - Fischer et al showed that  $m + 1$  rounds is the lower bound to reach agreement in a fully connected network where only processors are faulty
  - Thus, in a three processor system with one faulty processor, agreement cannot be reached

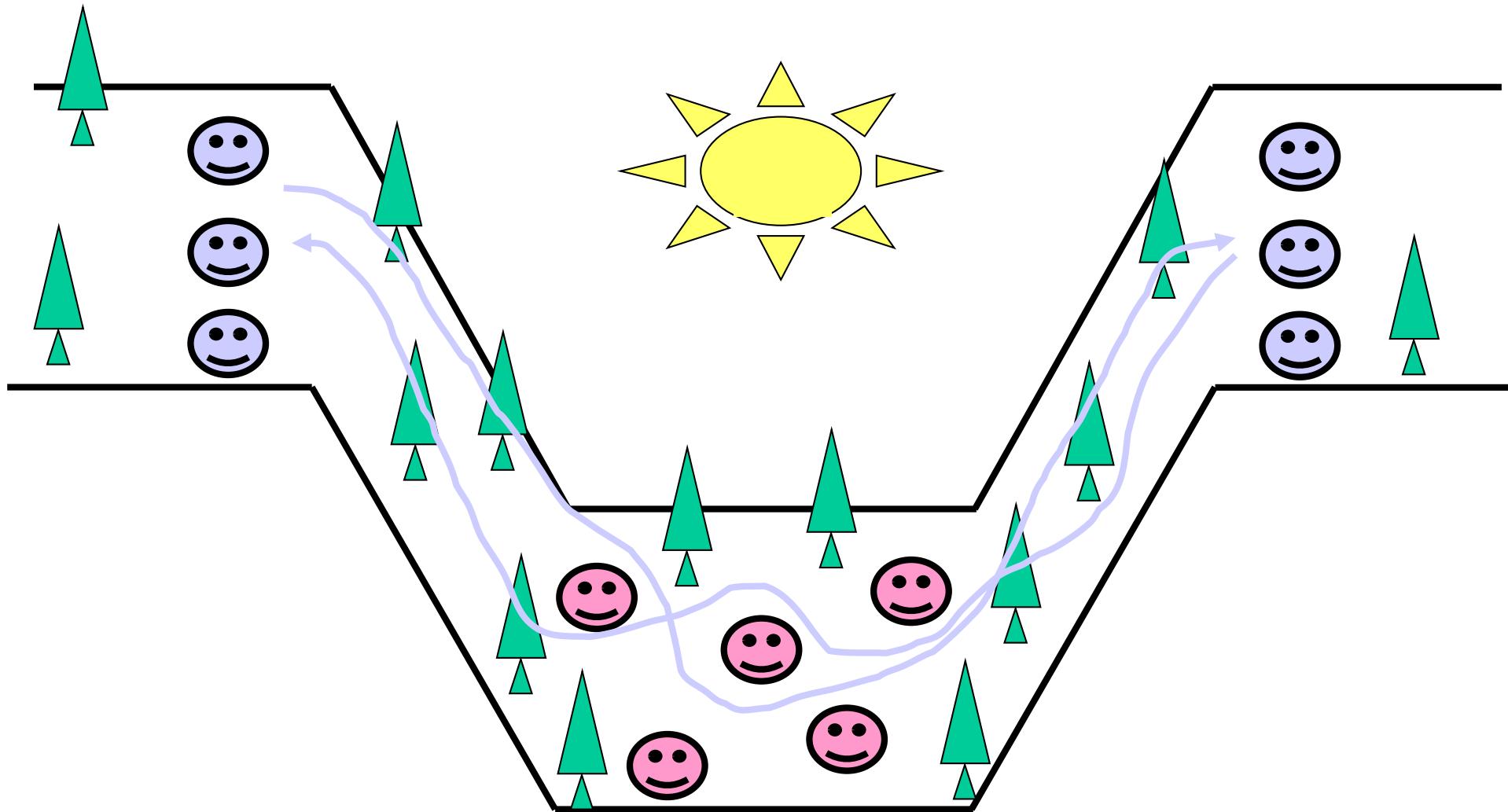
# Lamport - Shostak - Pease Algorithm

- The Oral Message ( $\text{OM}(m)$ ) algorithm with  $m > 0$  (some faulty processor(s)) solves the Byzantine agreement problem for  $3m + 1$  processors with at most  $m$  faulty processors
  - The initiator sends  $n - 1$  messages to everyone else to start the algorithm
  - Everyone else begins  $\text{OM}(m - 1)$  activity, sending messages to  $n - 2$  processors
  - Each of these messages causes  $\text{OM}(m - 2)$  activity, etc., until  $\text{OM}(0)$  is reached when the algorithm stops
  - When the algorithm stops each processor has input from all others and chooses the majority value as its value

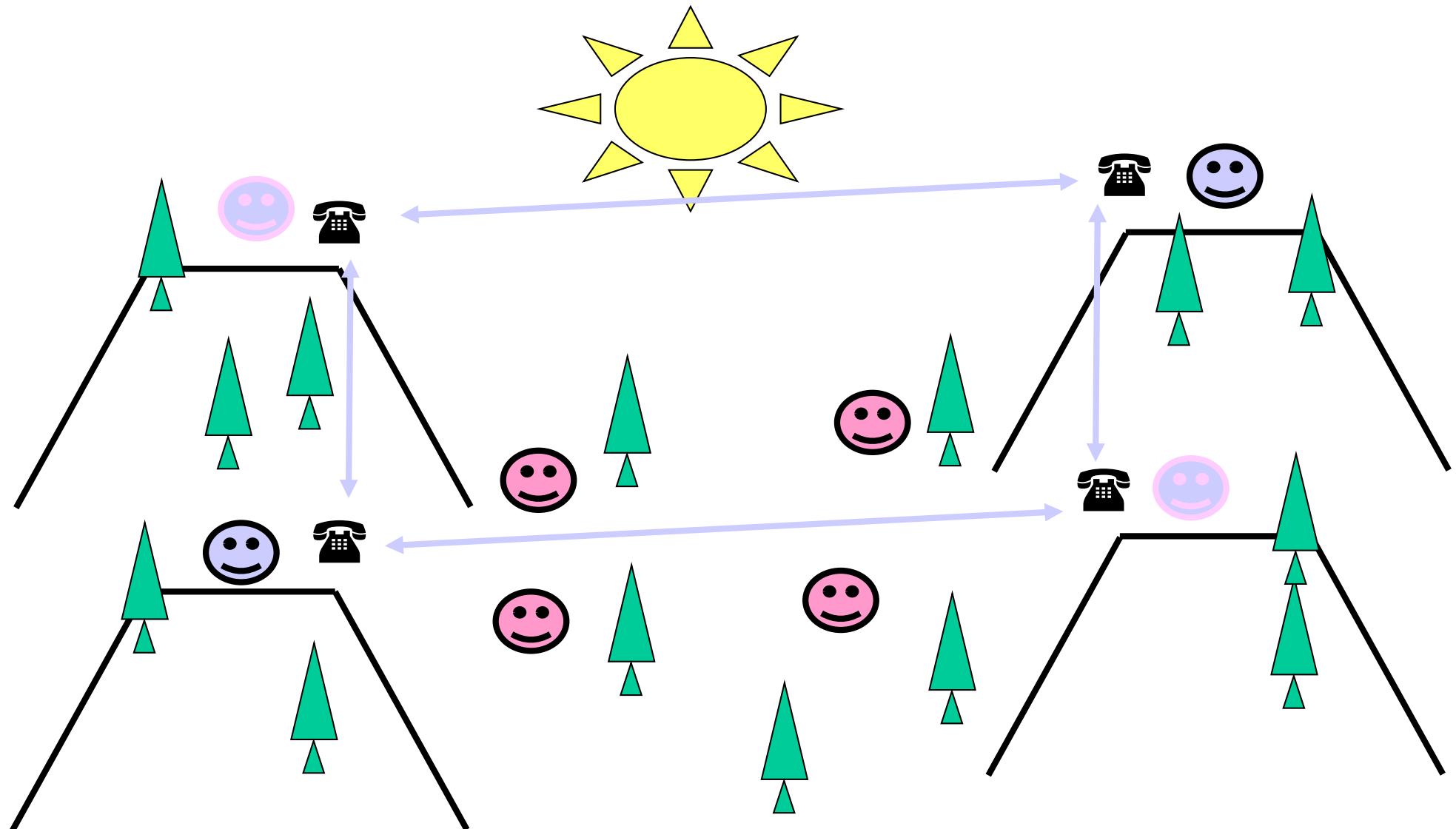
# Agreement

- Need agreement in DS:
  - Leader, commit, synchronize
- **Distributed Agreement algorithm:** all non-faulty processes achieve consensus in a finite number of steps
- Perfect processes, faulty channels: two-army
- Faulty processes, perfect channels: Byzantine generals

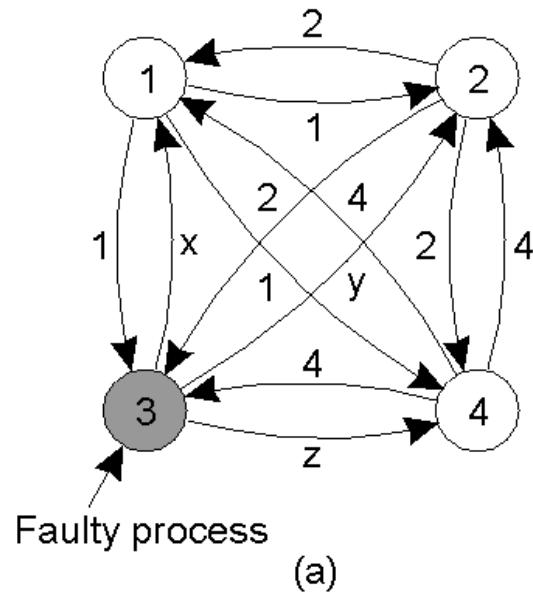
# Two-Army Problem



# Byzantine Generals Problem



# Byzantine Generals -Example (1)



1 Got(1, 2, x, 4)  
 2 Got(1, 2, y, 4)  
 3 Got(1, 2, 3, 4)  
 4 Got(1, 2, z, 4)

(b)

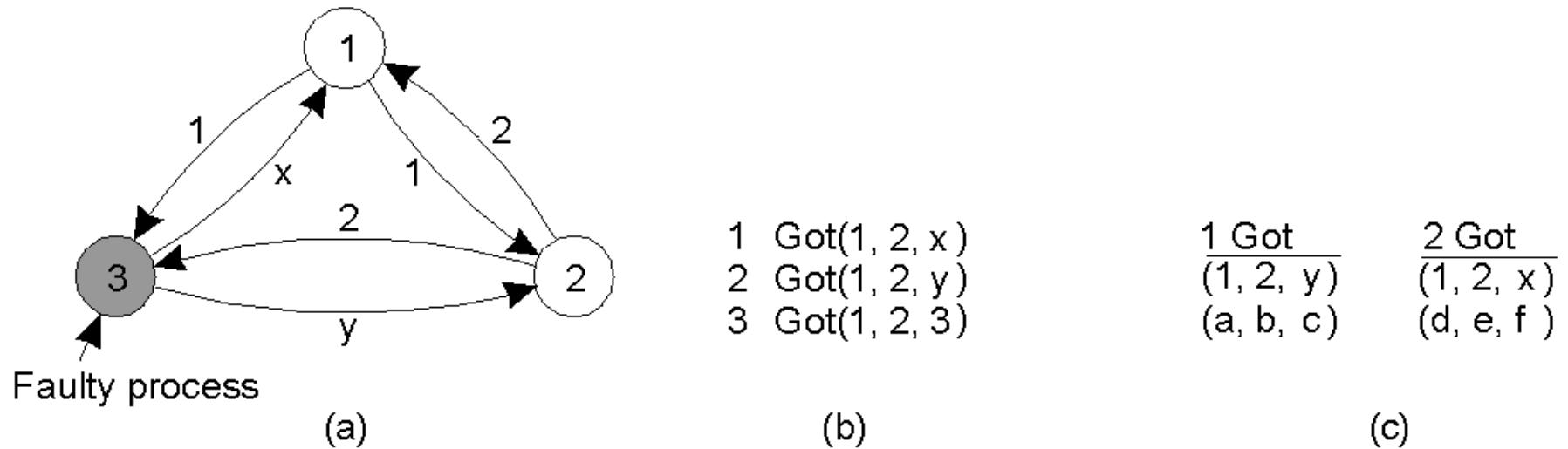
$\frac{1 \text{ Got}}{(1, 2, y, 4)}$ $(a, b, c, d)$	$\frac{2 \text{ Got}}{(1, 2, x, 4)}$ $(e, f, g, h)$	$\frac{4 \text{ Got}}{(1, 2, y, 4)}$ $(i, j, k, l)$
--	--	--

(c)

The Byzantine generals problem for 3 loyal generals and 1 traitor.

- a) The generals announce the time to launch the attack (by messages marked by their ids).
- b) The vectors that each general assembles based on (a)
- c) The vectors that each general receives in step 3, where every general passes his vector from (b) to every other general.

# Byzantine Generals –Example (2)



The same as in previous slide, except now  
with 2 loyal generals and one traitor.

# Byzantine Generals

- Given three processes, if one fails, consensus is impossible
- Given  $N$  processes, if  $F$  processes fail, consensus is impossible if  $N \leq 3F$

# Fault Tolerance

- Recovery: bringing back the failed node in step with other nodes in the system.
- Fault Tolerance: Increase the availability of a service or the system in the event of failures. Two ways of achieving it:
  - *Masking failures*: Continue to perform its specified function in the event of failure.
  - *Well defined failure behavior*: System may or may not function in the event of failure, but can facilitate actions suitable for recovery.
    - (e.g.,) effect of database transactions visible only if committed to by all sites. Otherwise, transaction is undone without any effect to other transactions.
- Key approach to fault tolerance: *redundancy*. e.g., multiple copies of data, multiple processes providing same service.
- Fault Tolerance achieved by:
  - commit protocols, voting protocols.

# Atomic Actions

- Example: Processes P1 & P2 share a data named X.
  - P1: ... lock(X); X:= X + Z; unlock(X); ...
  - P2: ... lock(X); X := X + Y; unlock(X); ...
- Updating of X by P1 or P2 should be done *atomically* i.e., without any interruption.

# Committing

- A set of actions is grouped as a transaction and the group is treated as an atomic action.
- The transaction, during the course of its execution, decides to commit or abort.
- *Commit*: guarantee that the transaction will be completed.
- *Abort*: guarantee *not* to do the transaction and erase any part of the transaction done so far.
- *Global atomicity*: (e.g.,) A distributed database transaction that must be processed at every or none of the sites.
- *Commit protocols*: are ones that enforce global atomicity.

# 2-phase Commit Protocol

- Distributed transaction carried out by a coordinator + a set of cohorts executing at different sites.
- Phase 1:
  - At the coordinator:
    - Coordinator sends a COMMIT-REQUEST message to every cohort requesting them to commit.
    - Coordinator waits for reply from all others.
  - At the cohorts:
    - On receiving the request: if the transaction execution is successful, the cohort writes UNDO and REDO log on stable storage. Sends AGREED message to coordinator.
    - Otherwise, sends an ABORT message.
- Phase 2:
  - At the coordinator:
    - All cohorts agreed? : write a COMMIT record on log, send COMMIT request to all cohorts.

# 2-phase Commit Protocol ...

- Phase 2...:
  - At the coordinator...:
    - Otherwise, send an ABORT message
    - Coordinator waits for acknowledgement from each cohort.
    - No acknowledgement within a timeout period? : resend the commit/abort message to that cohort.
    - All acknowledgements received? : write a COMPLETE record to the log.
  - At the cohorts:
    - On COMMIT message: resources & locks for the transaction released. Send Acknowledgement to the coordinator.
    - On ABORT message: undo the transaction using UNDO log, release resources & locks held by the transaction, send Acknowledgement.

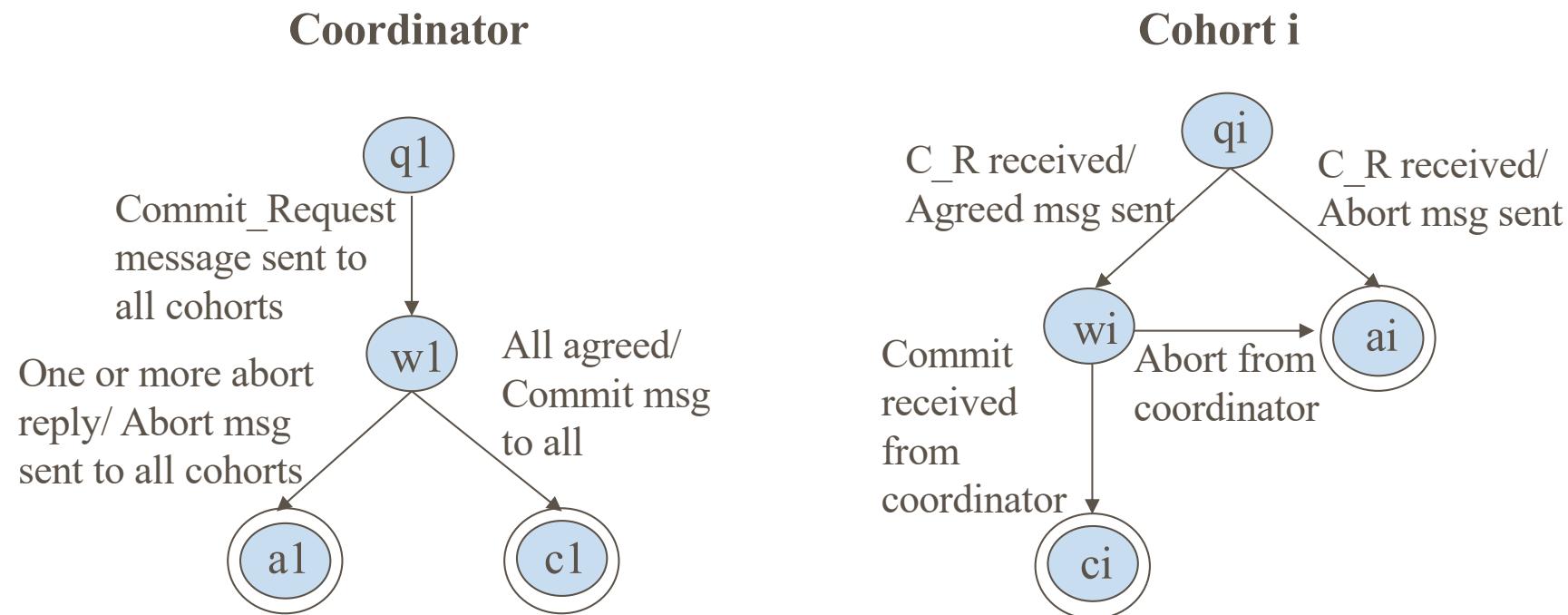
# Handling failures

- 2-phase commit protocol handles failures as below:
  - If coordinator crashes before writing the COMMIT record:
    - on recovery, it will send ABORT message to all others.
    - Cohorts who agreed to commit, will simply undo the transaction using the UNDO log and abort.
    - Other cohorts will simply abort.
    - All cohorts are blocked till coordinator recovers.
  - Coordinator crashes after COMMIT before writing COMPLETE
    - On recovery, broadcast a COMMIT and wait for ack
  - Cohort crashes in phase 1? : coordinator aborts the transaction.
  - Cohort crashes in phase 2? : on recovery, it will check with the coordinator whether to abort or commit.
- Drawback: blocking protocol. Cohorts blocked if coordinator fails.
  - Resources and locks held unnecessarily.

# 2-phase commit: State Machine

- Synchronous protocol: all sites proceed in rounds, i.e., a site never leads another by more than 1 state transition.
- A state transition occurs in a process participating in the 2-phase commit protocol whenever it receives/sends messages.
- States: q (idle or querying state), w (wait), a (abort), c (commit).
- When coordinator is in state q, cohorts are in q or a.
- Coordinator in w -> cohort can be in q, w, or a.
- Coordinator in a/c -> cohort is in w or a/c.
- A cohort in a/c: other cohorts may be in a/c or w.
- A site is never in c when another site is in q as the protocol is synchronous.

# 2-phase commit: State Machine...



# Drawback

- Drawback: blocking protocol. Cohorts blocked if coordinator fails.
  - Resources and locks held unnecessarily.
- Conditions that cause blocking:
  - Assume that only one site is operational. This site cannot decide to abort a transaction as some other site may be in commit state.
  - It cannot commit as some other site can be in abort state.
  - Hence, the site is blocked until all failed sites recover.

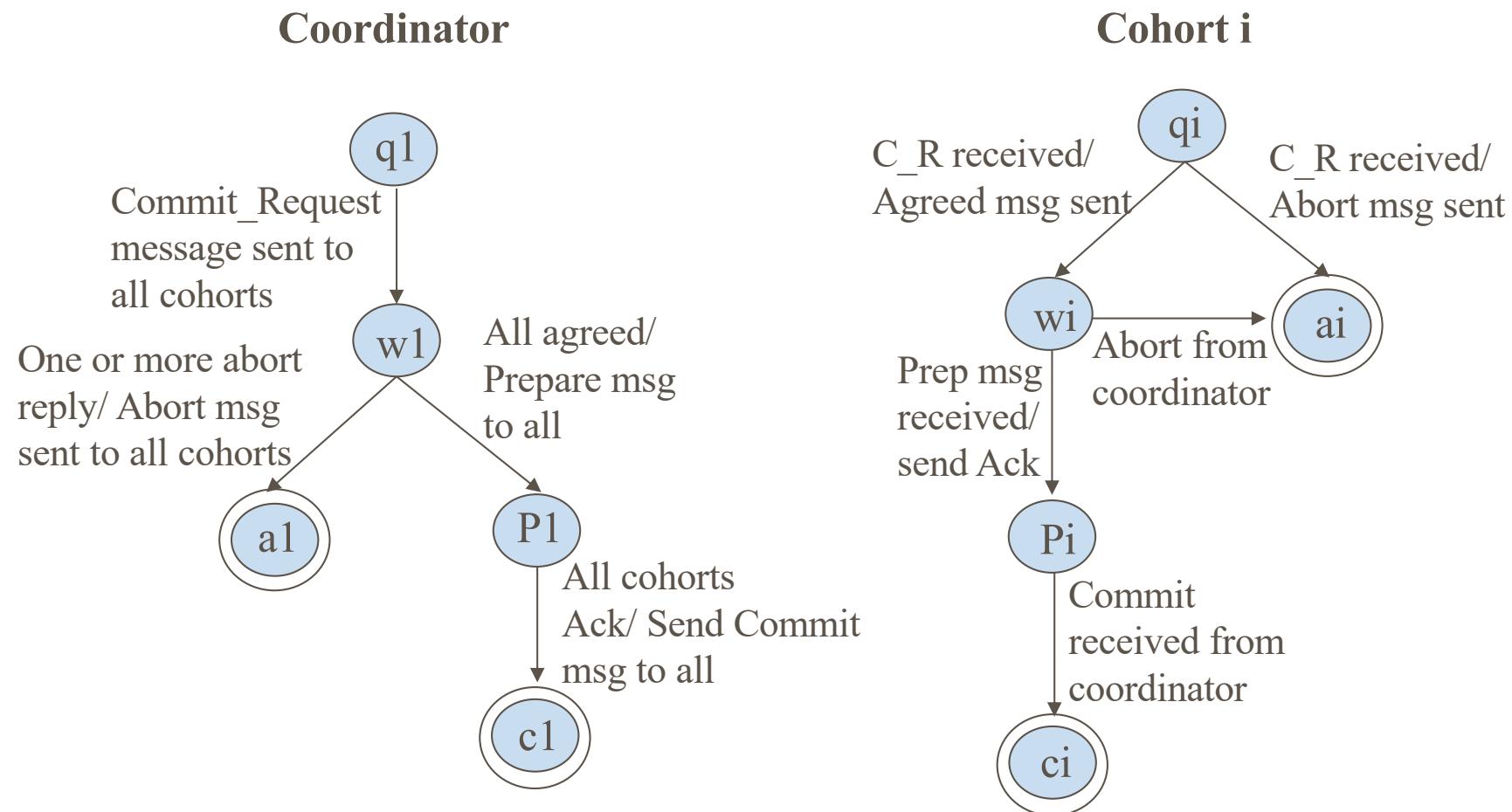
# Nonblocking Commit

- Nonblocking commit? :
  - Sites should agree on the outcome by examining their local states.
  - A failed site, upon recovery, should reach the same conclusion regarding the outcome. Consistent with other working sites.
  - *Independent recovery*: if a recovering site can decide on the final outcome based solely on its local state.
  - A nonblocking commit protocol can support independent recovery.
- Notations:
  - *Concurrency set*: Let  $S_i$  denote the state of the site  $i$ . The set of all the states that may be concurrent with it is concurrency set ( $C(s_i)$ ).
    - (e.g.,) Consider a system having 2 sites. If site 2's state is  $w_2$ , then  $C(w_2) = \{c_1, a_1, w_1\}$ .  $C(q_2) = \{q_1, w_1\}$ .  $a_1, c_1$  not in  $C(q_2)$  as 2-phase commit protocol is synchronous within 1 state transaction.
  - *Sender set*: Let  $s$  be any state,  $M$  be the set of all messages received in  $s$ . Sender set,  $S(s) = \{i \mid \text{site } i \text{ sends } m \text{ and } m \in M\}$

# 3-phase Commit

- *Lemma:* If a protocol contains a local state of a site with both abort and commit states in its concurrency set, then under independent recovery conditions it is not resilient to an arbitrary single failure.
- In previous figure,  $C(W2)$  can have both abort and commit states in the concurrency set.
- To make it a non-blocking protocol: introduce a buffer state at both coordinator and cohorts.
- Now,  $C(W1) = \{q2, w2, a2\}$  and  $C(w2) = \{a1, p1, w1\}$ .

# 3-phase commit: State Machine



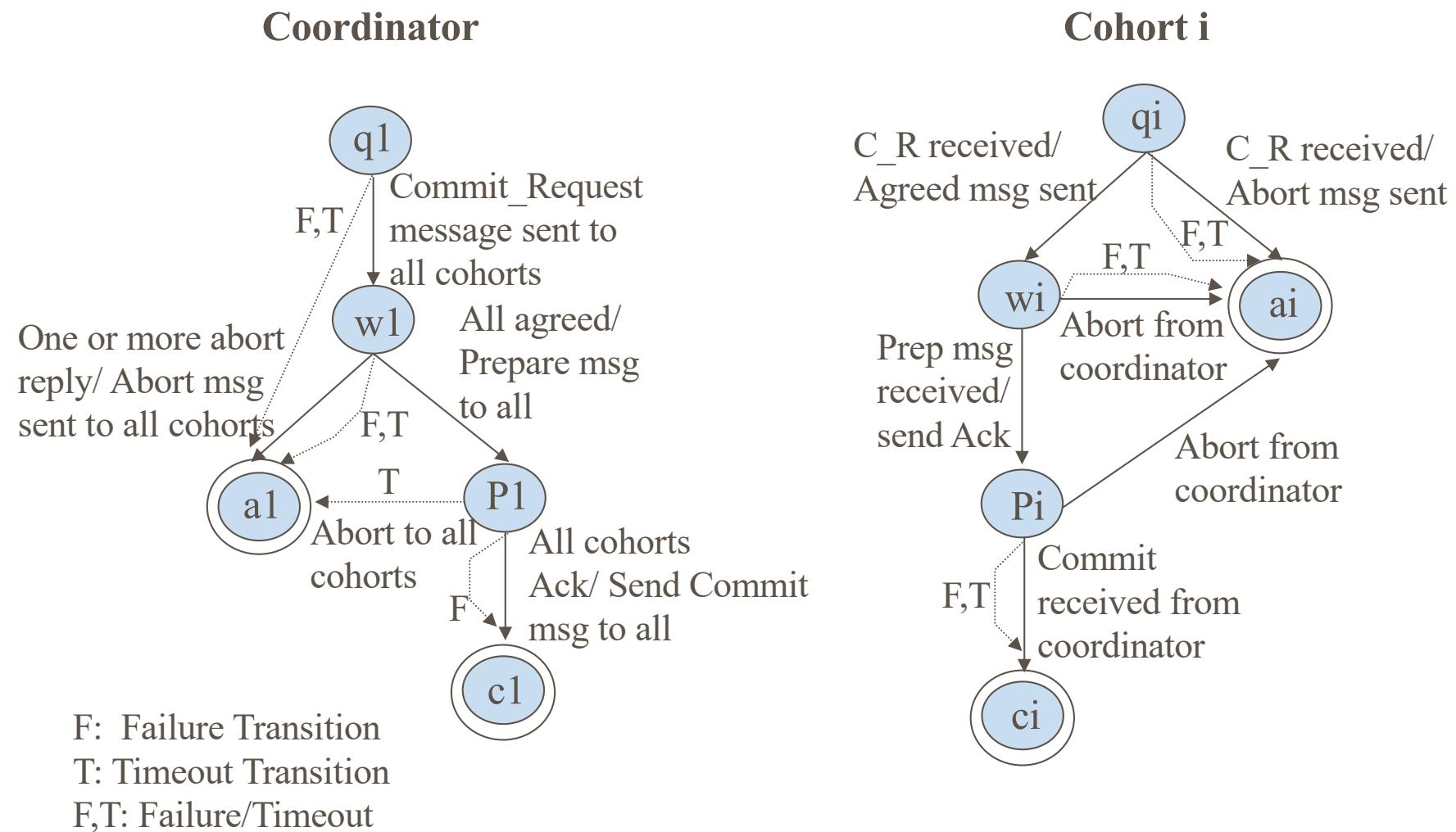
# Failure, Timeout Transitions

- A *failure transition* occurs at a failed site at the instant it fails or immediately after it recovers from the failure.
  - *Rule for failure transition:* For every non-final state  $s$  (i.e.,  $qi$ ,  $wi$ ,  $pi$ ) in the protocol, if  $C(s)$  contains a commit, then assign a failure transition from  $s$  to a commit state in its FSA. Otherwise, assign a failure transition from  $s$  to an abort state.
  - *Reason:*  $pi$  is the only state with a commit state in its concurrency set. If a site fails at  $pi$ , then it can commit on recovery. Any other state failure, safer to abort.
- If site  $i$  is waiting on a message from  $j$ ,  $i$  can time out.  $i$  can determine the state of  $j$  based on the expected message.
- Based on  $j$ 's state, the final state of  $j$  can be determined using failure transition at  $j$ .

# Failure, Timeout Transitions

- This can be used for incorporating *Timeout transitions* at  $i$ .
  - *Rule for timeout transition:* For each nonfinal state  $s$ , if site  $j$  in  $S(s)$ , and site  $j$  has a failure transition from  $s$  to a commit (abort) state, then assign a timeout transition from  $s$  to a commit (abort) state.
  - *Reason:*
    - Failed site makes a transition to a commit (abort) state using failure transition rule.
    - So, the operational site must make the same transition to ensure that the final outcome is the same at all sites.

# 3-phase commit + Failure Trans.



# Nonblocking Commit Protocol

- Phase 1:
  - First phase identical to that of 2-phase commit, except for failures.
  - Here, coordinator is in w1 and each cohort is in a or w or q, depending on whether it has received the `commit_request` message or not.
- Phase 2:
  - Coordinator sends a `Prepare` message to all the cohorts (if all of them sent `Agreed` message in phase 1).
  - Otherwise, it will send an `Abort` message to them.
  - On receiving a `Prepare` message, a cohort sends an acknowledgement to the coordinator.
    - If the coordinator fails before sending a `Prepare` message, it aborts the transaction on recovery.
    - Cohorts, on timing out on a `Prepare` message, also aborts the transaction.

# Nonblocking Commit Protocol

- Phase 3:
  - On receiving acknowledgements to Prepare messages, the coordinator sends a Commit message to all cohorts.
  - Cohort commits on receiving this message.
    - Coordinator fails before sending commit? : commits upon recovery.
    - So cohorts on Commit message timeout, commit to the transaction.
    - Cohort failed before sending an acknowledgement? : coordinator times out and sends an abort message to all others.
    - Failed cohort aborts the transaction upon recovery.
- Use of buffer state:
  - (e.g.,) Suppose state  $p_i$  (in cohort) is not present. Let coordinator wait in state  $p_1$  waiting for ack. Let cohort 2 (in  $w_2$ ) acknowledge and commit.
  - Suppose cohort 3 fails in  $w_3$ . Coordinator will time out and abort. Cohort 3 will abort on recovery. Inconsistent with cohort 2.

# Commit Protocols Disadvantages

- No protocol using the above independent recovery technique can detect the simultaneous failure of more than 1 site.
- The above protocol is also not resilient to network partitioning.

# Failure Resilience

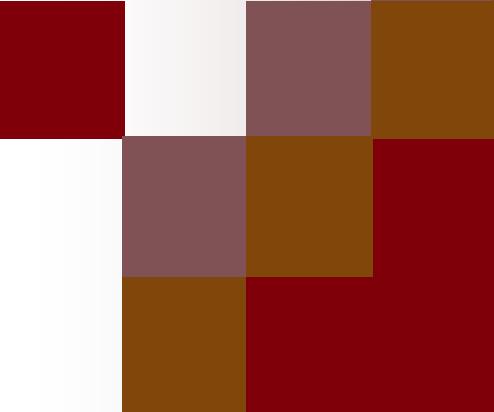
- Resilient process: is one that masks failures and guarantees progress despite a certain number of failures.
- Approaches:
  - Backup process:
    - A primary process + 1 or more backup processes
    - Primary process executes while backups are inactive
    - If primary fails, one of the backup processes take over the functions of the primary process.
    - To facilitate this takeover, state of the primary process is checkpointed at appropriate intervals.
    - Checkpointed state stored in a place that will be available even in the case of primary process failure.

# Failure Resilience

- Resilient Approaches:
  - Backup process...:
    - Plus:
      - Little system resources are consumed by backup processes as they are inactive.
    - Minus...:
      - Delay in takeover as (1) backups need to detect primary's failure (mostly by timeouts of "heartbeat" messages)  
(2) backup needs to recompute the system state based on checkpoint.
      - Backup should not reissue IOs and resend messages already sent by the primary process. Also, messages processed by the primary after the checkpoint must be available for back up process during the recomputation.
      - Which backup to act as primary needs to be solved.

# Failure Resilience

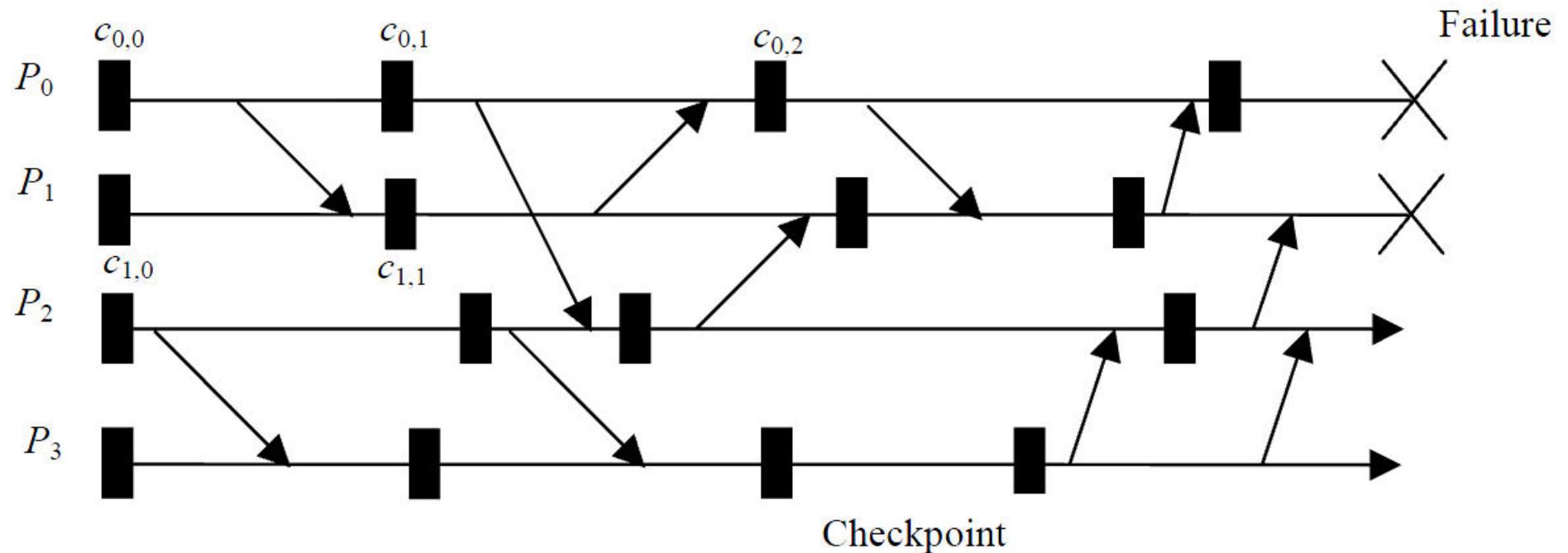
- Resilient Approaches...:
  - Replicated execution...:
    - Several processes execute simultaneously. Service continues as long as one of them is available.
    - Plus:
      - Provides increased reliability and availability
    - Minus:
      - More system resources needed for all processes
      - Concurrent updates need to be handled



# **Rollback-Recovery Protocols**

Message Passing Systems

# Checkpoints

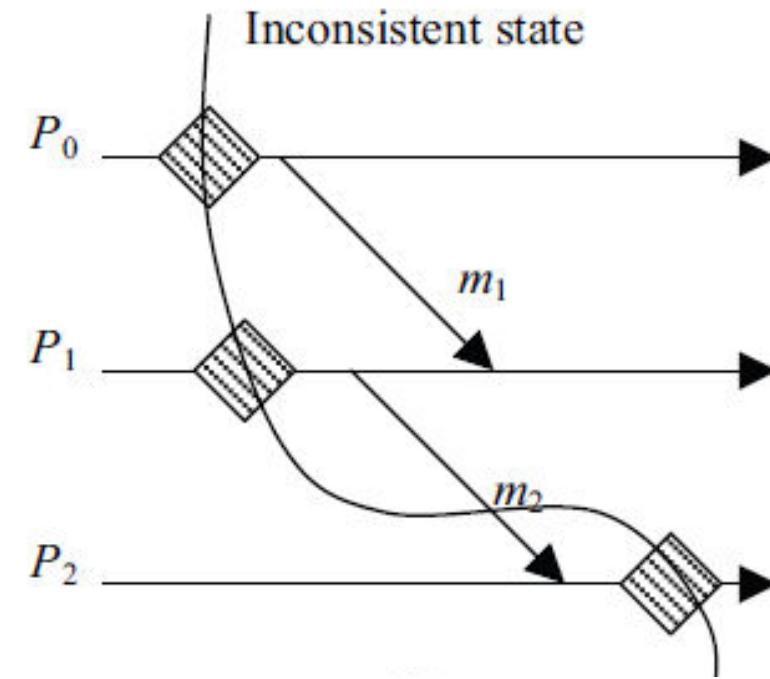
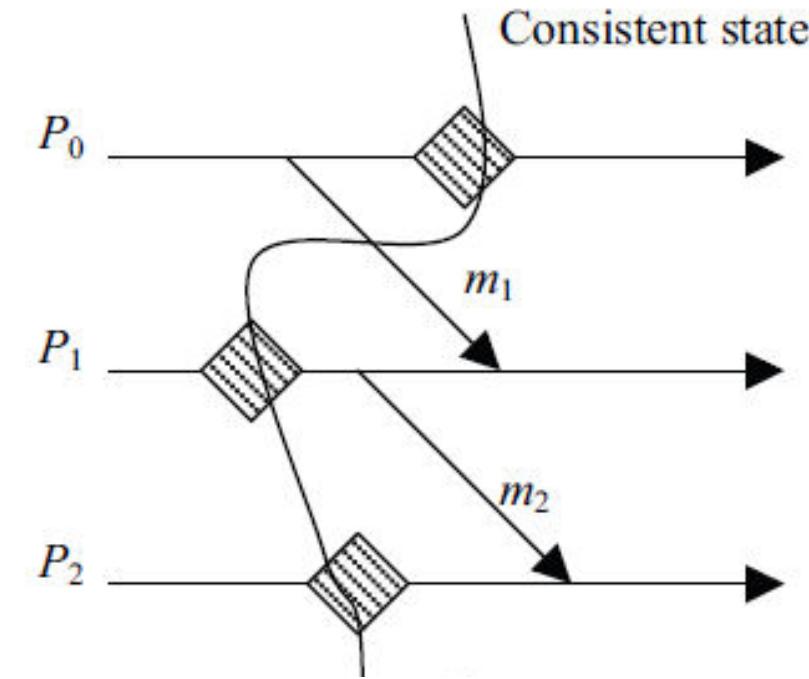


## Stable Storage

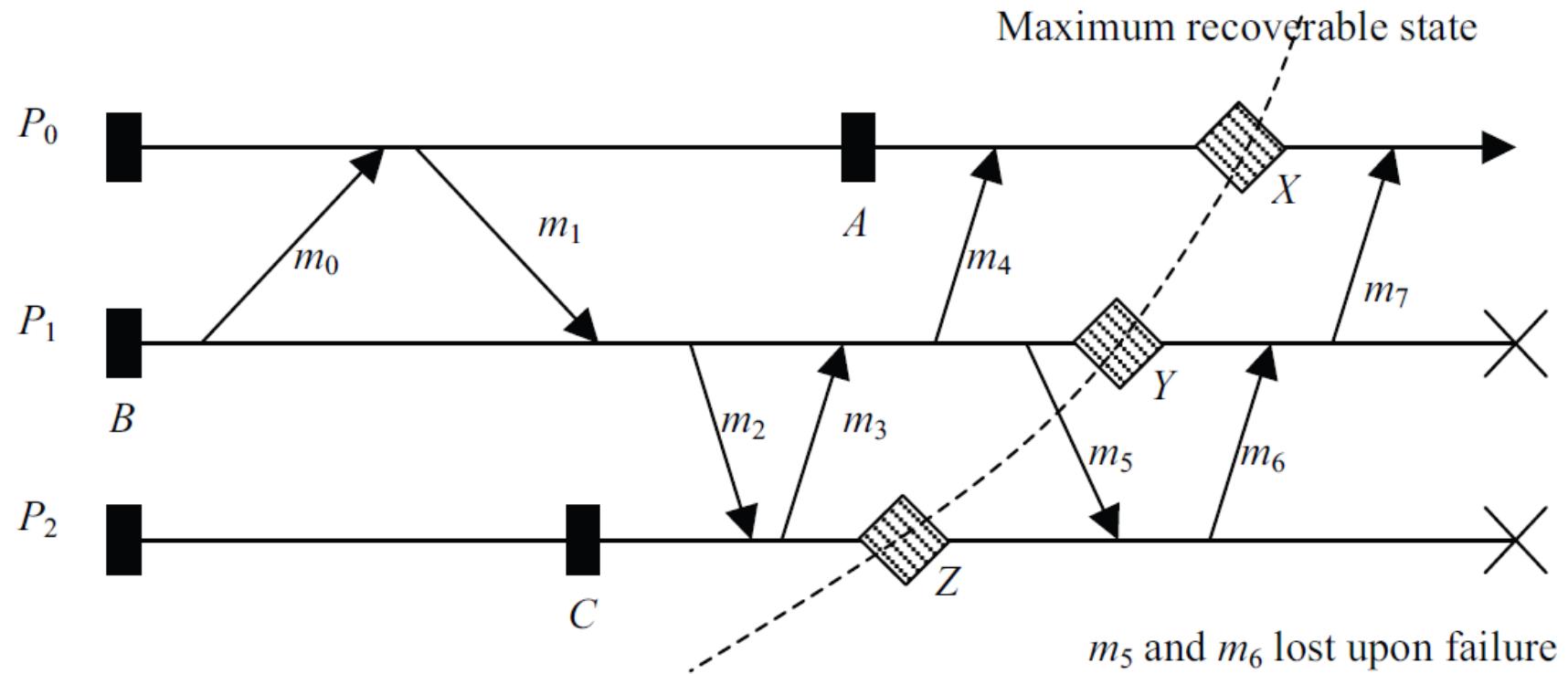
- must store recovery data through failures
  - Checkpoints, event logs, other recovery info
  - Implementation options
    - A system that tolerates only a single failure
      - Volatile memory
    - A system that tolerates transient failures
      - Local disk in each host
    - A system that tolerates non-transient failures
      - A replicated file system

## Consistent System States

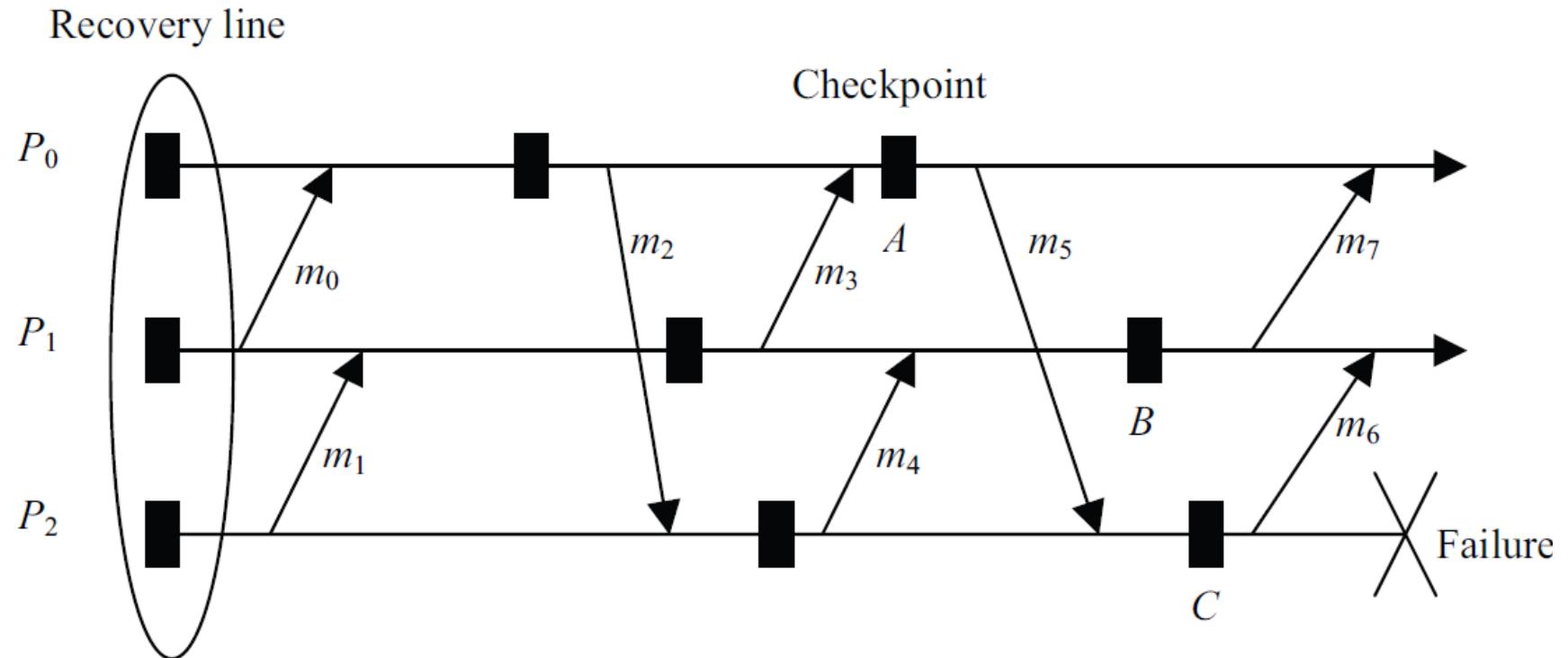
- Lost Messages
  - **Sent but never received - OK**
- "Orphan Messages"
  - **Received but never sent - bad**



# Maximum Recoverable State



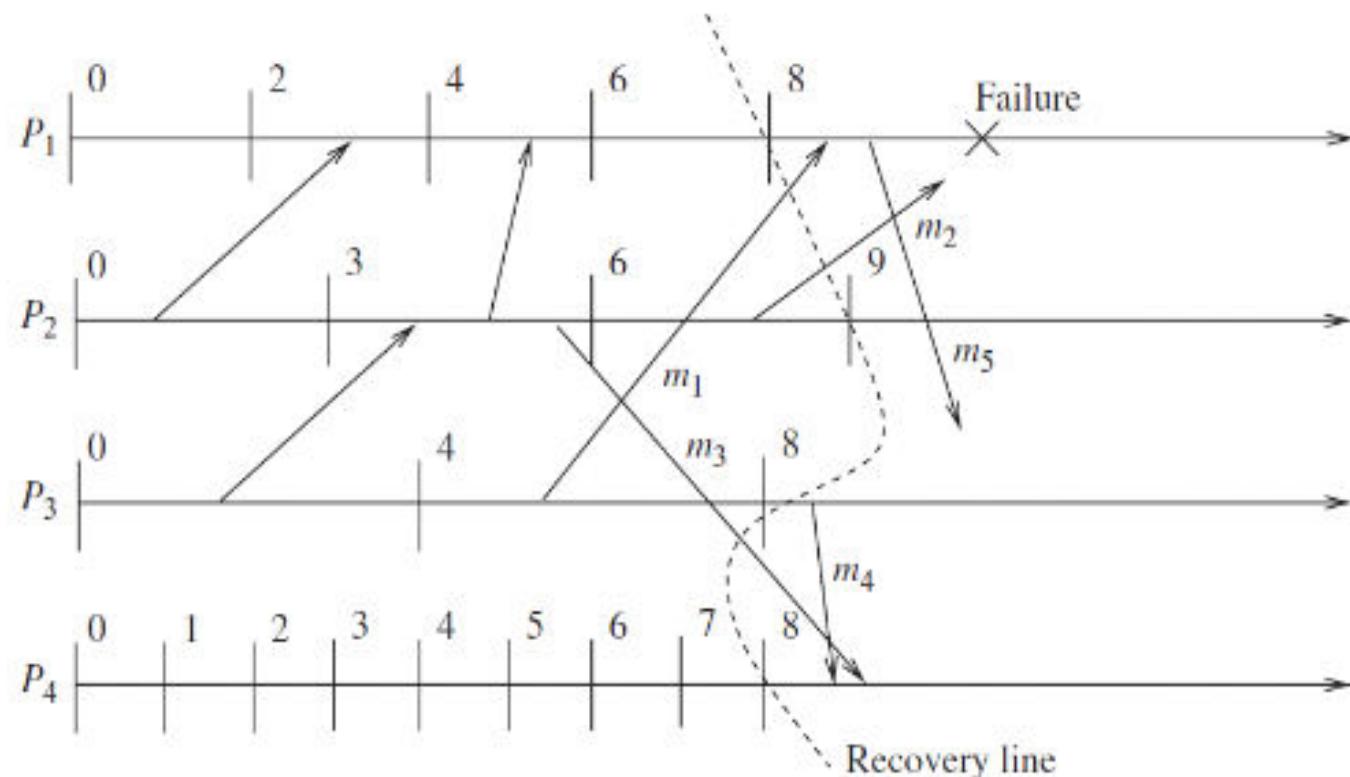
# The Domino Effect



# Messages

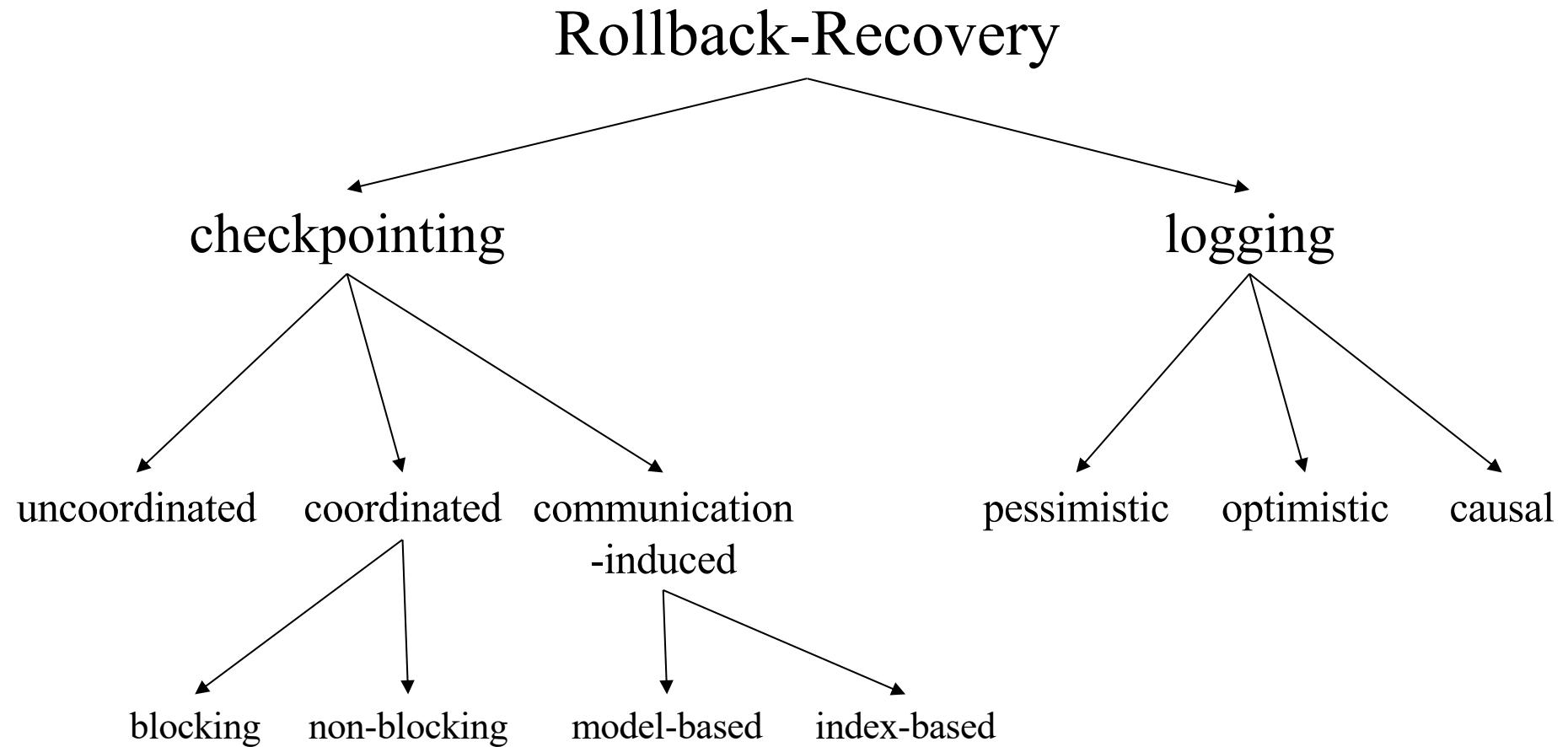
- In-transit message
  - messages that have been sent but not yet received
- Lost messages
  - messages whose ‘send’ is done but ‘receive’ is undone due to rollback
- Delayed messages
  - messages whose ‘receive’ is not recorded because the receiving process was either down or the message arrived after rollback
- Orphan messages
  - messages with ‘receive’ recorded but message ‘send’ not recorded
  - do not arise if processes roll back to a consistent global state
- Duplicate messages
  - arise due to message logging and replaying during process recovery

# Messages – example



- In-transit
  - $m_1, m_2$
- Lost
  - $m_1$
- Delayed
  - $m_1, m_5$
- Orphan
  - none
- Duplicated
  - $m_4, m_5$

## Taxonomy



## Checkpoint-Based Rollback Recovery

- restores the system state to the recovery line
- less restrictive and simpler to implement
- Does not guarantee that prefailure execution can be deterministically regenerated after a rollback
- Not suited for interactions with the outside world
- Categories
  - ✓ Uncoordinated checkpointing
  - ✓ Coordinated checkpointing
  - ✓ Communication-induced checkpointing

## Uncoordinated Checkpointing

- Each process takes checkpoints independently
- Recovery line must be calculated after failure
- Disadvantages
  - ✓ susceptible to domino effect
  - ✓ can generate useless checkpoints
  - ✓ complicates storage
  - ✓ not suitable for frequent output commits

## Coordinated Checkpointing

- Checkpoints are orchestrated between processes
- Triggered by application decision
- Simplifies recovery
- Not susceptible to the domino effect
- Only one checkpoint per process on stable storage
- Garbage collection not necessary
- Large latency

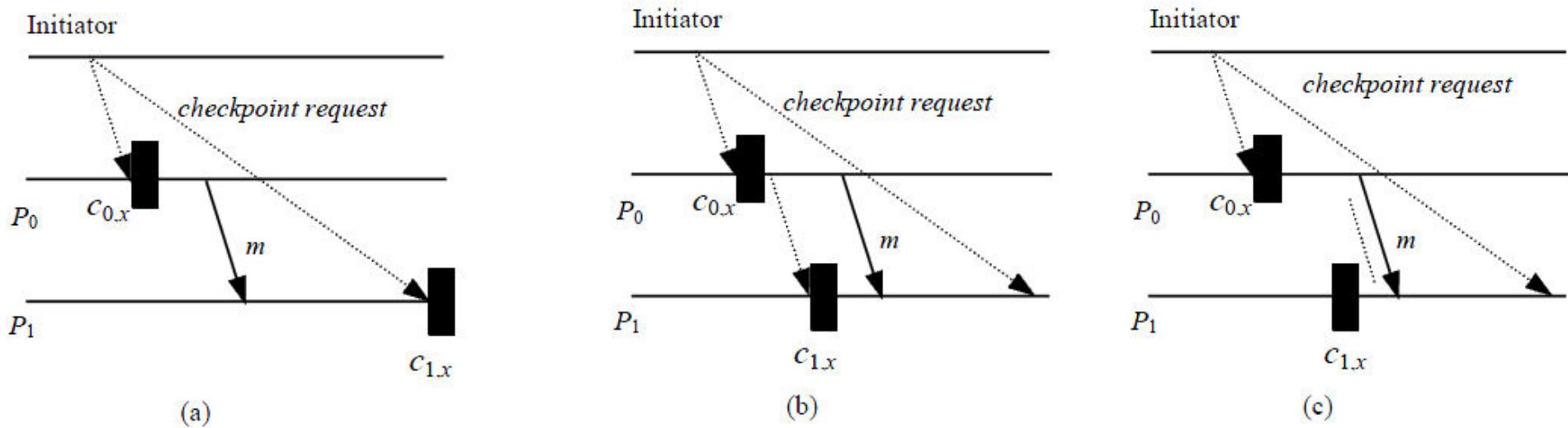
## Coordinated Checkpointing / Blocking

- No messages can be in transit during checkpointing
- Large overhead

## Two-Phase Checkpointing Protocol

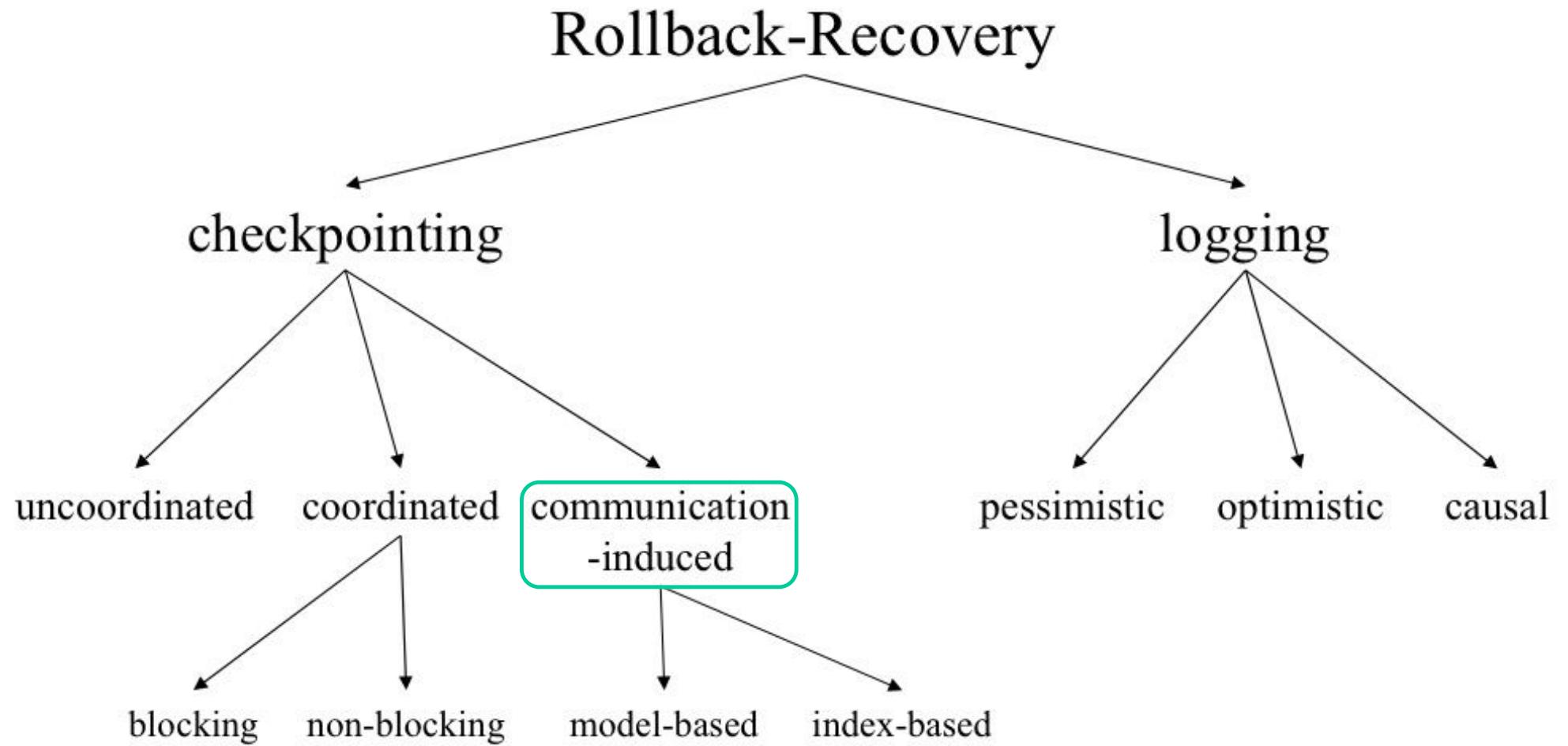
- A coordinator takes a checkpoint
- Broadcasts a checkpoint request to all processes
- When a process receives this message, it stops its execution, takes a tentative checkpoint
- Send an acknowledgment back to coordinator
- Coordinator broadcasts a commit message
- Each process removes the old checkpoint and makes the tentative checkpoint permanent

# Coordinated Checkpointing / Non-Blocking



**Figure 8.** Non-blocking coordinated checkpointing: (a) checkpoint inconsistency; (b) with FIFO channels; (c) non-FIFO channels (short dashed line represents piggybacked *checkpoint request*).

# Taxonomy

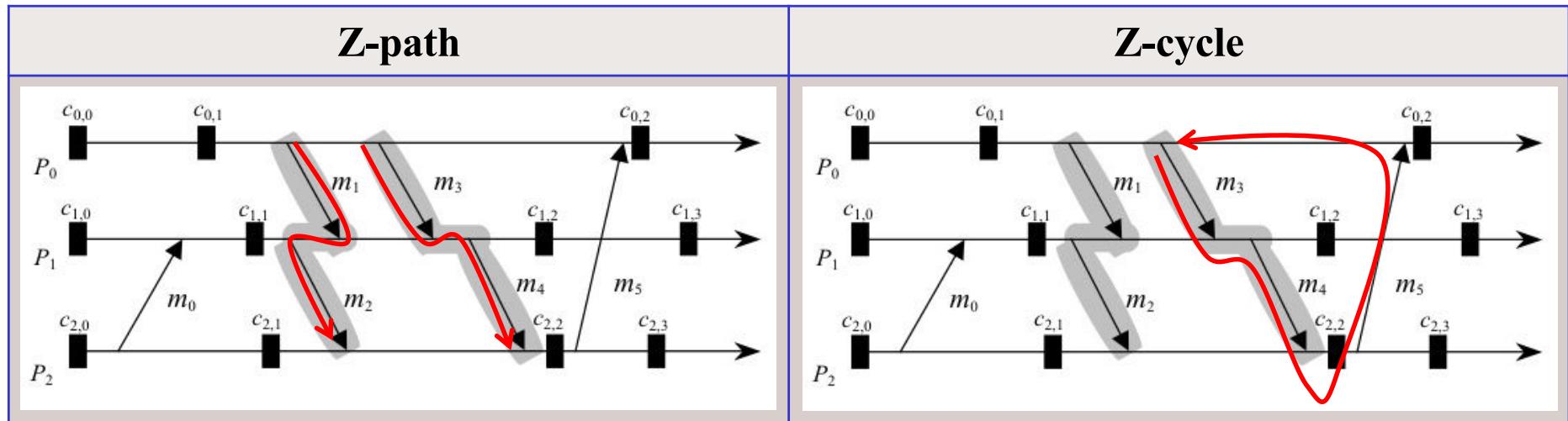


## Communication-Induced Checkpointing

- Avoid the domino effect without requiring all checkpoints to be coordinated.
- Processes take two kinds of checkpoints: *local* and *forced*.
- Local checkpoints can be taken independently.
- Forced checkpoints must be taken to guarantee the eventual progress of the recovery line.
- No special coordination messages are exchanged to determine when forced checkpoints should be taken.
- Protocol-specific information is piggybacked on each application message.
- The receiver uses this information to decide if it should take a forced checkpoint.

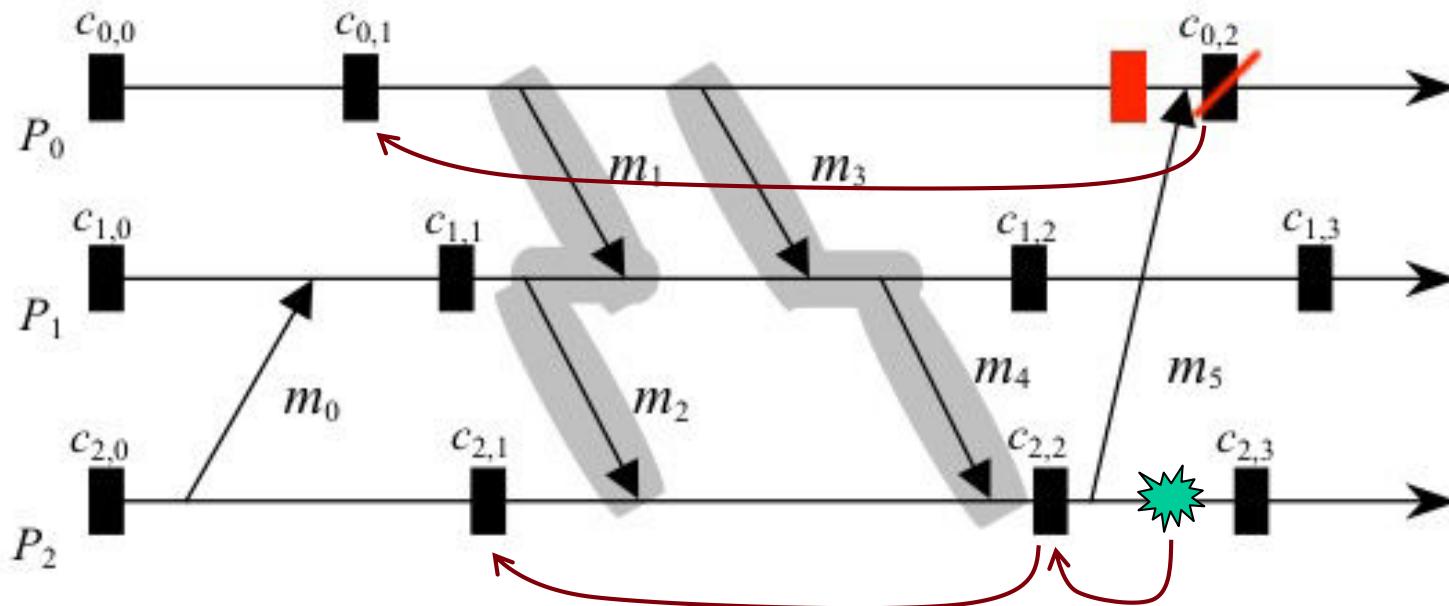
## Communication-Induced Checkpointing Notation

How does a receiver decide when to take a forced checkpoint?



- A checkpoint is useless if and only if it is part of a Z-cycle.
- The receiver should determine if past communication and checkpoint patterns can lead to the creation of useless checkpoints.

## Communication-Induced Checkpointing Notation

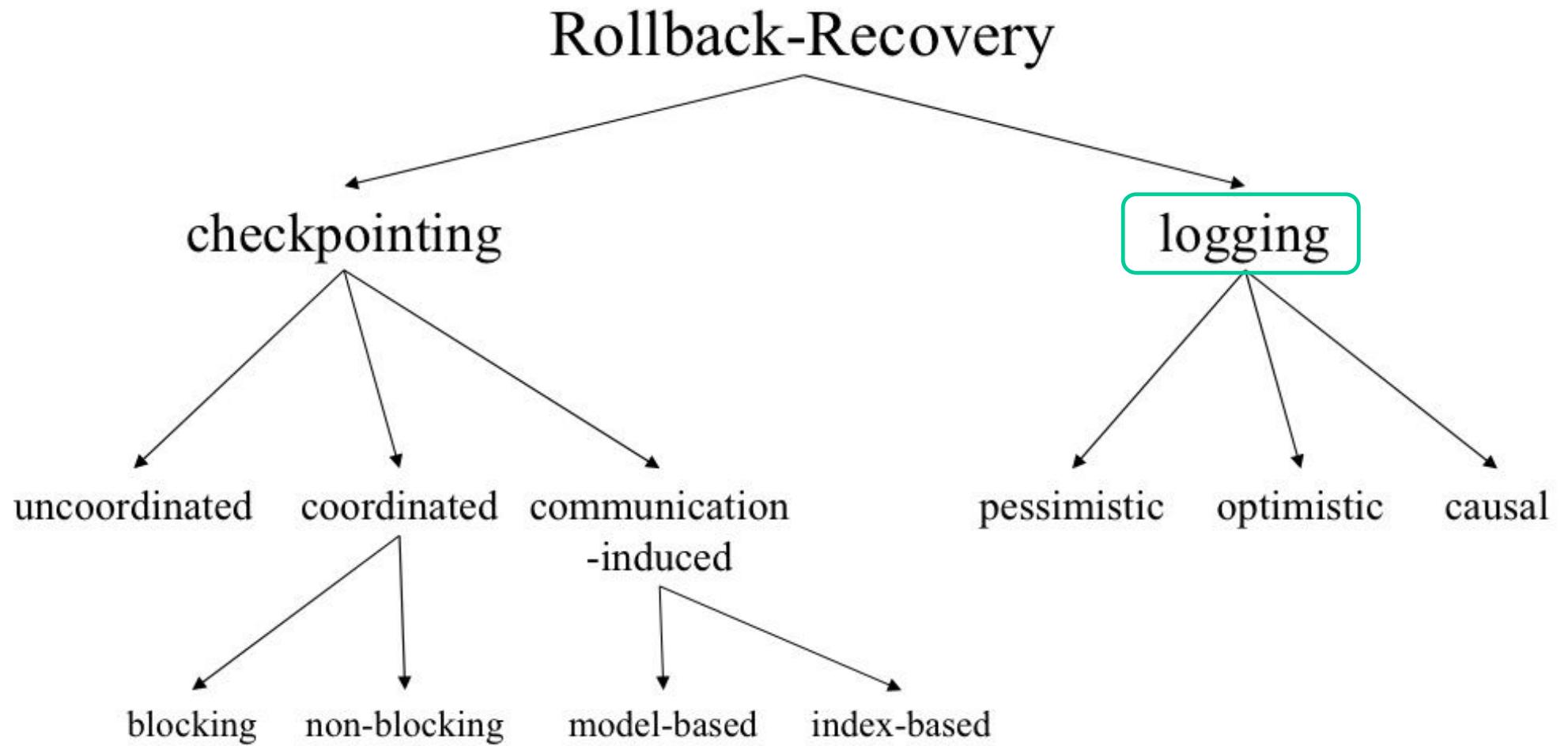


Checkpoint  $c_{2,2}$  is useless under any failure scenario.  $P_0$  must create a forced checkpoint before delivering  $m_5$  to break the  $m_3$ - $m_4$ - $m_5$  Z-cycle.

## Communication-Induced Checkpointing

- CIC protocols have been classified in two types:
  - ***Model-based Protocols:*** Take more forced checkpoints than is probably necessary, because without explicit coordination, no process has complete information about the global system state.
  - ***Index-based protocols:*** Guarantee that checkpoints having the same index at different processes form a consistent state.

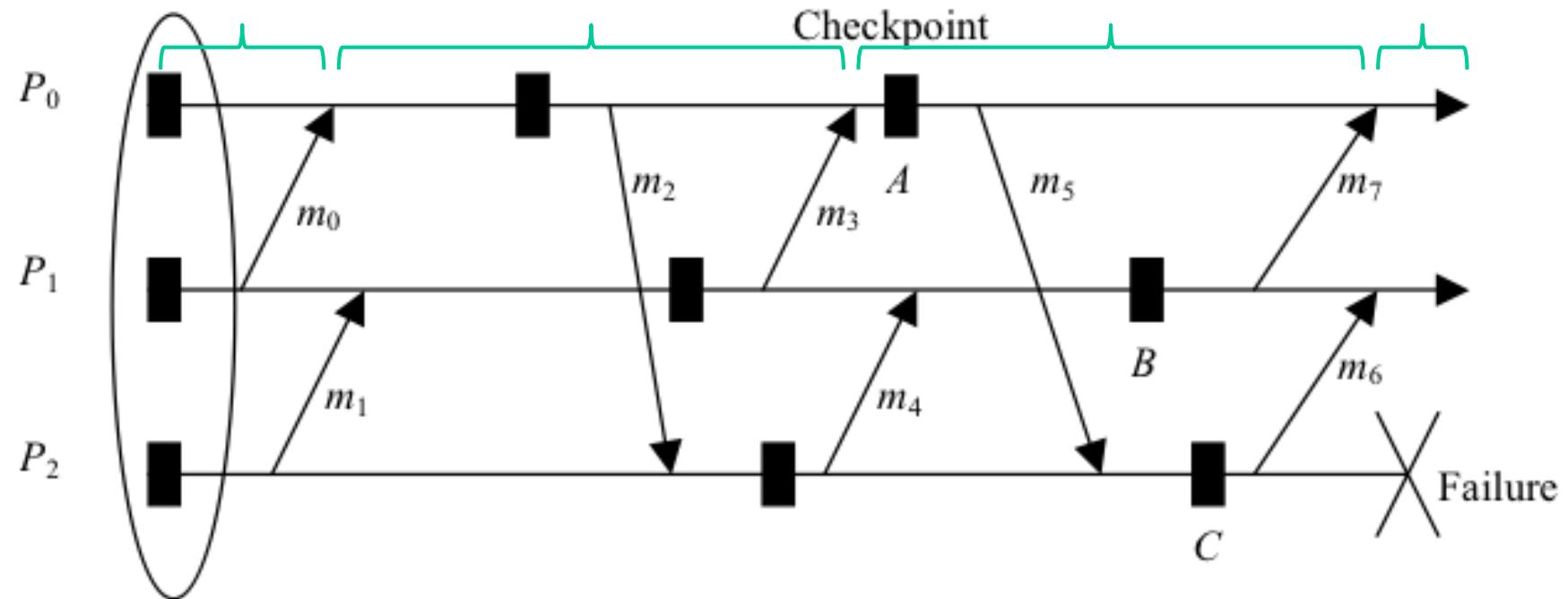
# Taxonomy



## Log-Based Rollback Recovery

- Process execution is modeled as a sequence of deterministic state intervals, each starting with the execution of a nondeterministic event.
- *Non-deterministic event*: the receipt of a message or an internal event (something that affects the process).
- *Deterministic event*: sending a message (an effect caused by the process).

# Log-Based Rollback Recovery



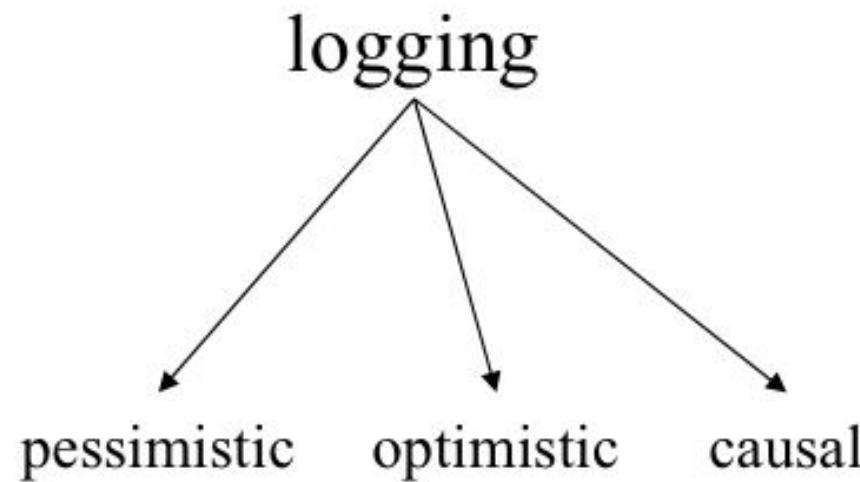
## Log-Based Rollback Recovery

- All non-deterministic events can be identified and their determinants are logged to stable storage.
  - **Determinant:** the information need to “replay” the occurrence of a non-deterministic event.
- During failure-free operation, each process logs the determinants of all the non-deterministic events it observes onto stable storage.
- Each process also takes checkpoints to reduce the extent of rollback during recovery.
- After a failure occurs, the failed processes recover by using the checkpoints and logged determinants to replay the corresponding nondeterministic events precisely as they occurred during the pre-failure execution.

## Log-Based Rollback Recovery

Key parameters:

- Failure-free performance overhead.
- Output-commit latency.
- Simplicity of recovery and garbage collection.
- Potential for rolling back correct processes.



## Log-Based Rollback Recovery / Pessimistic Logging

- Assumes that a failure can occur after *any* nondeterministic.
- The determinant of each nondeterministic event is logged to stable storage before the event is allowed to affect the computation.
- Employs *synchronous* logging (a strengthening of the *always-no-orphans* condition):

## Log-Based Rollback Recovery / Optimistic Logging

- Determinants of non-deterministic events are logged *asynchronously*: determinants are kept in a volatile log which is periodically flushed to stable storage.
- Assumes that logging will complete before a failure occurs.
- Allows the temporary creation of orphan processes, but none should exist by the time recovery is complete.

## Log-Based Rollback Recovery / Causal Logging

- Has the failure-free performance advantages of optimistic logging while retaining most of the advantages of optimistic logging.
- Avoids synchronous access to stable storage except during output commit.
- Similar to pessimistic logging in:
  - **Allows each process to commit output independently.**
  - **Never creates orphan processes.**
  - **Limits the rollback of any failed process to the most recent checkpoint.**
- Cost: a more complex recovery protocol.

## Log-Based Rollback Recovery / Causal Logging

- Ensures the *always-no-orphans* property by ensuring that the determinant of each non-deterministic event that causally precedes the state of a process is either stable or it is available locally to that process.
- Processes piggyback the non-stable determinants in their volatile log on the messages they send to other processes.

# *Election Algorithms*

# *Why Election?*

- ❖ Example 1: Your Bank maintains multiple servers in their cloud, but for each customer, one of the servers is responsible, i.e., is the **leader**
  - ❖ What if there are two leaders per customer?
    - ❖ Inconsistency
  - ❖ What if servers disagree about who the leader is?
    - ❖ Inconsistency
  - ❖ What if the leader crashes?
    - ❖ Unavailability

- ❖ Example 2: Group of cloud servers replicating a file need to elect one among them as the primary replica that will communicate with the client machines

# ***What is Election?***

- ❖ In a group of processes, elect a *Leader* to undertake special tasks.
- ❖ What happens when a leader fails (crashes)
  - ❖ Some (at least one) process detects this (how?)
  - ❖ Then what?
- ❖ Focus of this lecture: **Election algorithm**
  1. Elect one leader only among the non-faulty processes
  2. All non-faulty processes agree on who is the leader

## ***System Model/Assumptions***

- ❖ Any process can **call** for an **election**.
- ❖ A process can call for **at most one election at a time**.
- ❖ Multiple processes can call an election simultaneously.
  - ❖ All of them together must yield a single leader only
- ❖ The result of an election should not depend on which process calls for it.
- ❖ Messages are eventually delivered.

# **Problem Specification**

- ❖ At the end of the election protocol, the non-faulty process with the best (highest) election attribute value is elected.
  - ❖ Attribute examples: leader has highest id or address. Fastest cpu. Most disk space. Most number of files, etc.
- ❖ Protocol may be initiated anytime or after leader failure
- ❖ A *run* (execution) of the election algorithm must always guarantee at the end:
  - Safety:  $\forall$  non-faulty p: (p's elected = (q: a particular non-faulty process with the best attribute value) or  $\perp$ )
  - Liveness:  $\forall$  election: (election terminates)  
&  $\forall$  p: non-faulty process, p's elected is not  $\perp$

A special value  $\perp$  is used to mark when this variable is undefined, i.e. when there is an election in progress.

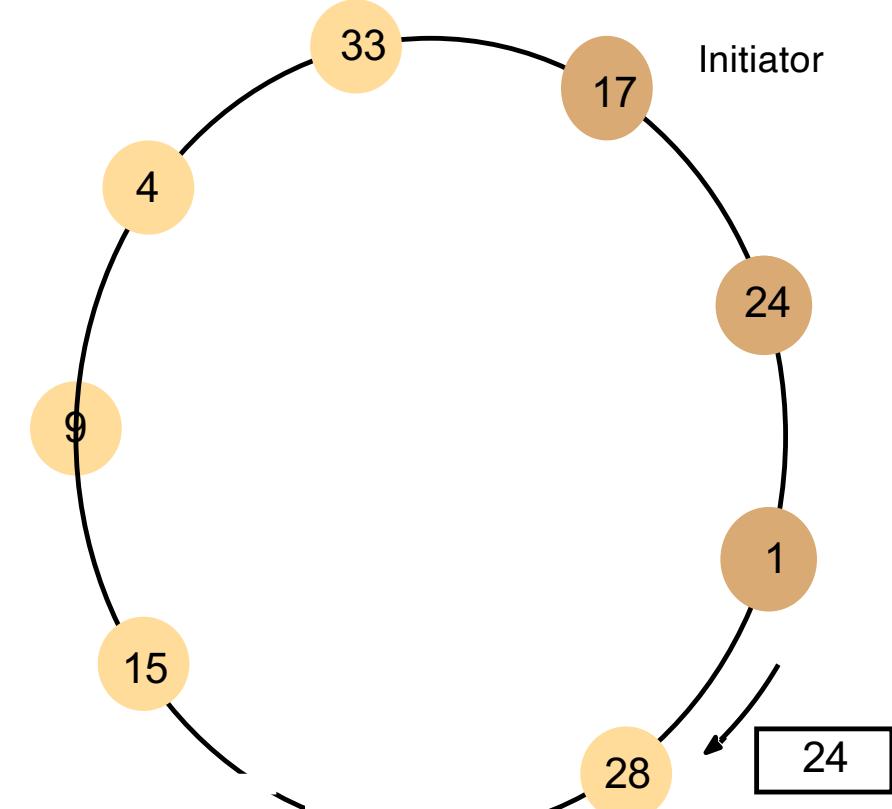
# Algorithm 1: Ring Election

- ❖ N Processes are organized in a logical ring
  - ❖  $p_i$  has a communication channel to  $p_{(i+1) \bmod N}$
  - ❖ All messages are sent clockwise around the ring.
- ❖ Any process  $p_i$  that discovers the old coordinator has failed initiates an “election” message that contains  $p_i$ ’s own id:attr. This is the *initiator* of the election.
- ❖ When a process  $p_i$  receives an *election* message, it compares the attr in the message with its own attr.
  - ❖ If the arrived attr is greater,  $p_i$  forwards the message.
  - ❖ If the arrived attr is smaller and  $p_i$  has not yet forwarded an election message, it overwrites the message with its own id:attr, and forwards it.
  - ❖ If the arrived id:attr matches that of  $p_i$ , then  $p_i$ ’s attr must be the greatest (why?), and it becomes the new coordinator. This process then sends an “elected” message to its neighbor with its id, announcing the election result.
- ❖ When a process  $p_i$  receives an *elected* message, it
  - ❖ sets its variable *elected*  $\leftarrow$  id of the message.
  - ❖ forwards the message, unless it is the new coordinator.

# **Ring-Based Election: Example**

(In this example, attr:=id)

- In the example: The election was started by process 17. The highest process identifier encountered so far is 24. (final leader will be 33)
- The worst-case scenario occurs when the counter-clockwise neighbor (@ the initiator) has the highest attr.

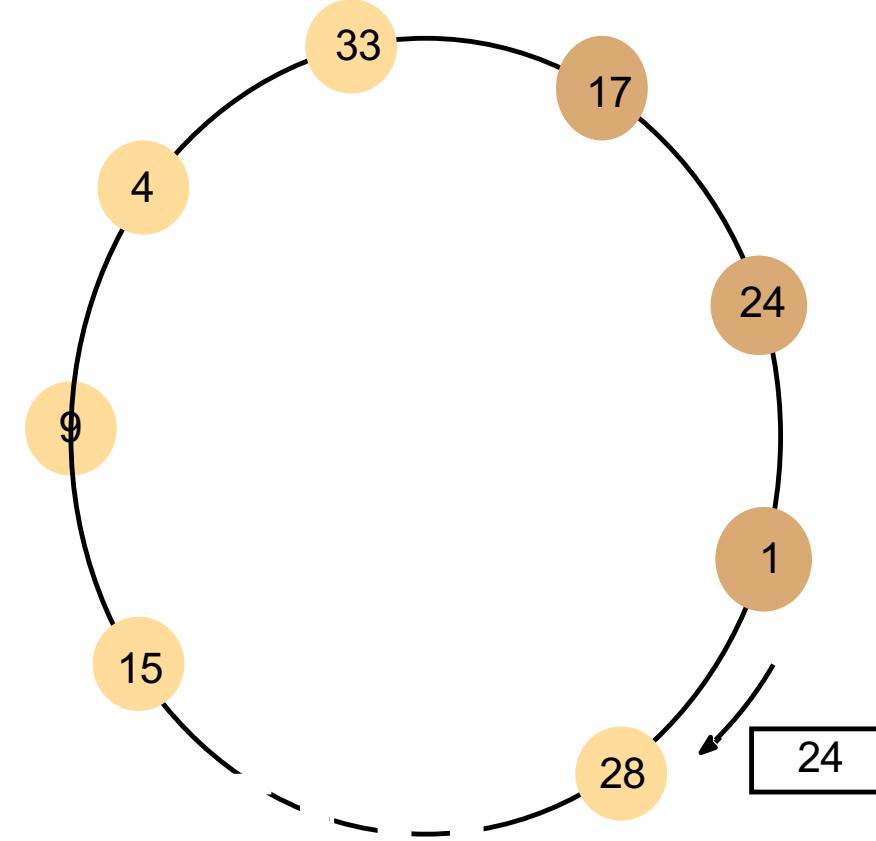


# **Ring-Based Election: Analysis**

- ❖ The worst-case scenario occurs when the counter-clockwise neighbor has the highest attr.

In a ring of N processes, in the worst case:

- ❖ A total of  $N-1$  messages are required to reach the new coordinator-to-be (election messages).
- ❖ Another  $N$  messages are required until the new coordinator-to-be ensures it is the new coordinator (election messages – no changes).
- ❖ Another  $N$  messages are required to circulate the elected messages.
- ❖ Total Message Complexity =  $3N-1$
- ❖ Turnaround time =  $3N-1$



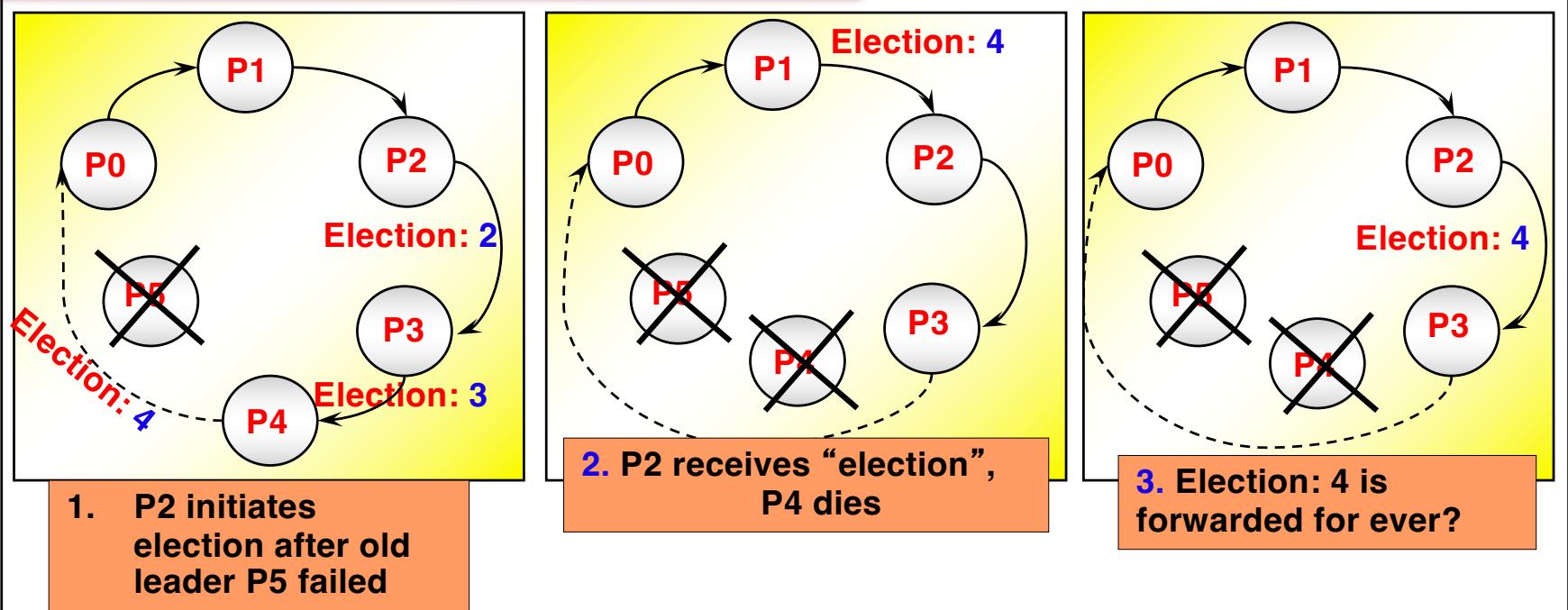
# **Correctness?**

**Assume – no failures happen during the run of the election algorithm**

- **Safety and Liveness are satisfied.**

**What happens if there are failures during the election run?**

# Example: Ring Election



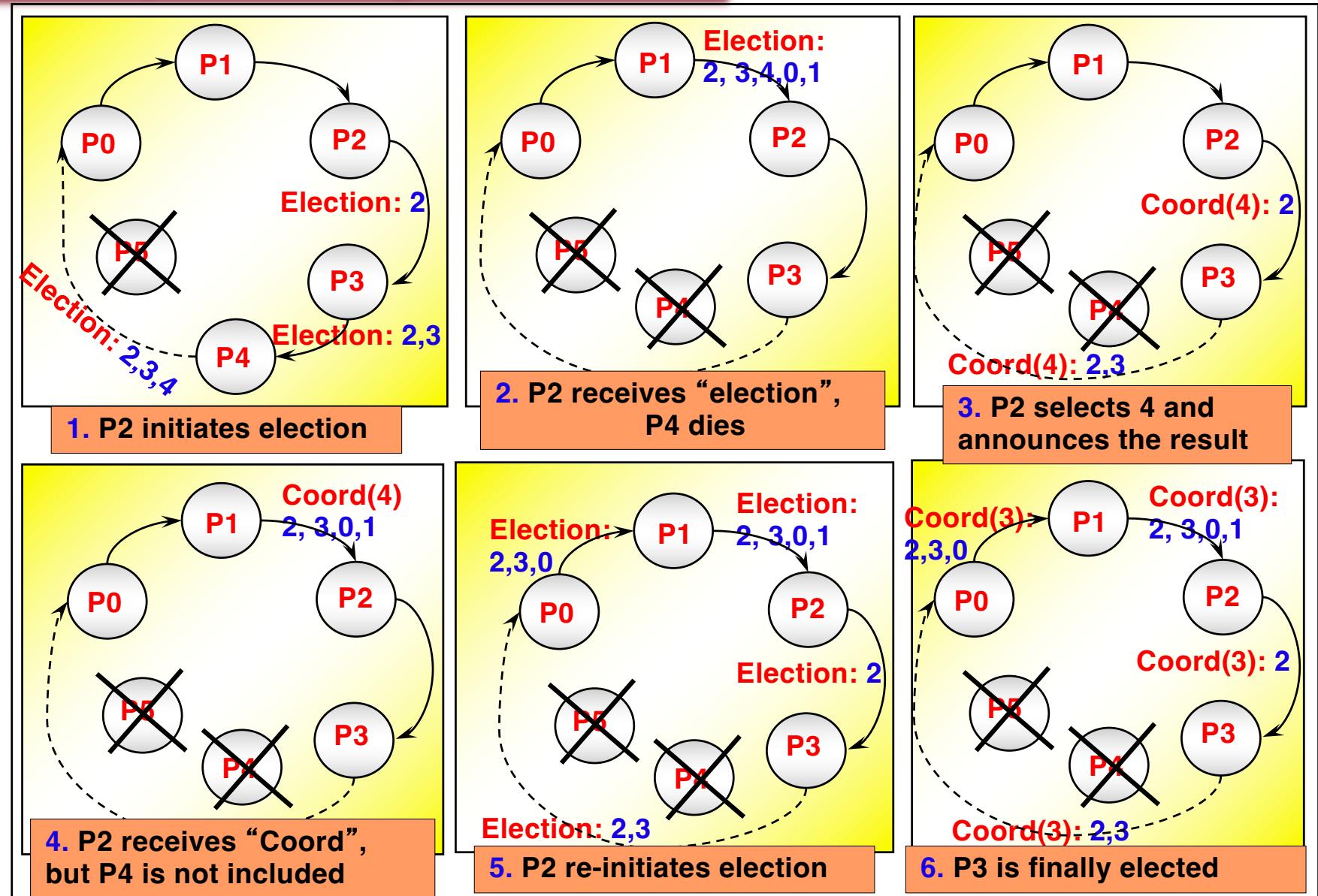
May not terminate when process failure occurs during the election!  
Consider above example where attr == id

Does not satisfy liveness

## **Algorithm 2: Modified Ring Election**

- ❖ Processes are organized in a logical ring.
- ❖ Any process that discovers the coordinator (leader) has failed initiates an “election” message.
- ❖ The message is circulated around the ring, bypassing failed processes.
- ❖ Each process appends (adds) its id:attr to the message as it passes it to the next process (without overwriting what is already in the message)
- ❖ Once the message gets back to the initiator, it elects the process with the best election attribute value.
- ❖ It then sends a “coordinator” message with the id of the newly-elected coordinator. Again, each process adds its id to the end of the message, and records the coordinator id locally.
- ❖ Once “coordinator” message gets back to initiator,
  - ❖ election is over if would-be-coordinator’s id is in id-list.
  - ❖ else the algorithm is repeated (handles election failure).

# Example: Ring Election



# **Modified Ring Election**

- Supports concurrent elections – an initiator with a lower id blocks other initiators’ election messages
- Reconfiguration of ring upon failures
  - Can be done if all processes “know” about all other processes in the system
  - If initiator non-faulty ...
  - How many messages?  $2N$
  - What is the turnaround time?  $2N$
  - Size of messages?  $O(N)$
- How would you redesign the algorithm to be fault-tolerant to an initiator’s failure?
  - One idea: Have the initiator’s successor wait a while, timeout, then re-initiate a new election. Do the same for this successor’s successor, and so on...
  - What if timeouts are too short... starts to get messy

# **Leader Election Is Hard**

- The Election problem is related to the **consensus problem**
- Consensus is impossible to solve with 100% guarantee in an asynchronous system with no bounds on message delays and arbitrarily slow processes
- So is leader election in fully asynchronous system model
- Where does the modified Ring election start to give problems with the above asynchronous system assumptions?
  - $p_i$  may just be very slow, but not faulty (yet it is not elected as leader!)
  - Also slow initiator, ring reorganization

## ***Algorithm 3: Bully Algorithm***

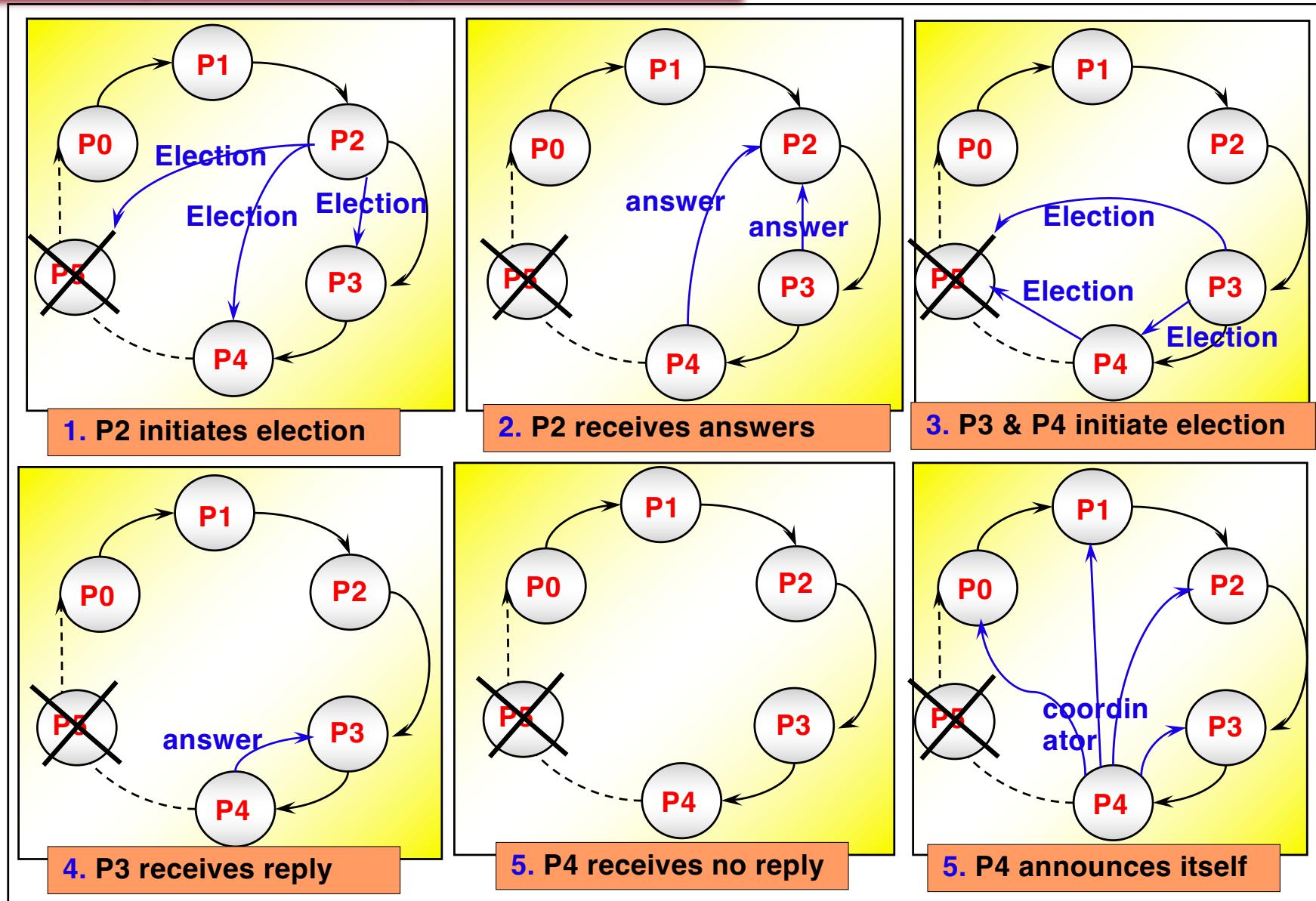
- ❖ Assumptions:
  - ❖ Synchronous system
    - ❖ All messages arrive within  $T_{trans}$  units of time.
    - ❖ A reply is dispatched within  $T_{process}$  units of time after the receipt of a message.
    - ❖ if no response is received in  $2T_{trans} + T_{process}$ , the process is assumed to be faulty (crashed).
  - ❖ attr==id
  - ❖ Each process knows all the other processes in the system (and thus their id's)

## ***Algorithm 3: Bully Algorithm***

- ❖ When a process finds that the coordinator has failed, if it knows its id is the highest, it elects itself as coordinator, then sends a *coordinator* message to all processes with lower identifiers than itself
- ❖ A process initiates election by sending an *election* message to only processes that have a higher id than itself.
  - If no answer within timeout, send coordinator message to lower id processes → Done.
  - if any answer received, then there is some non-faulty higher process → so, wait for coordinator message. If none received after another timeout, start a new election.
- ❖ A process that receives an “election” message replies with *answer* message, & starts its own election protocol (unless it has already done so)

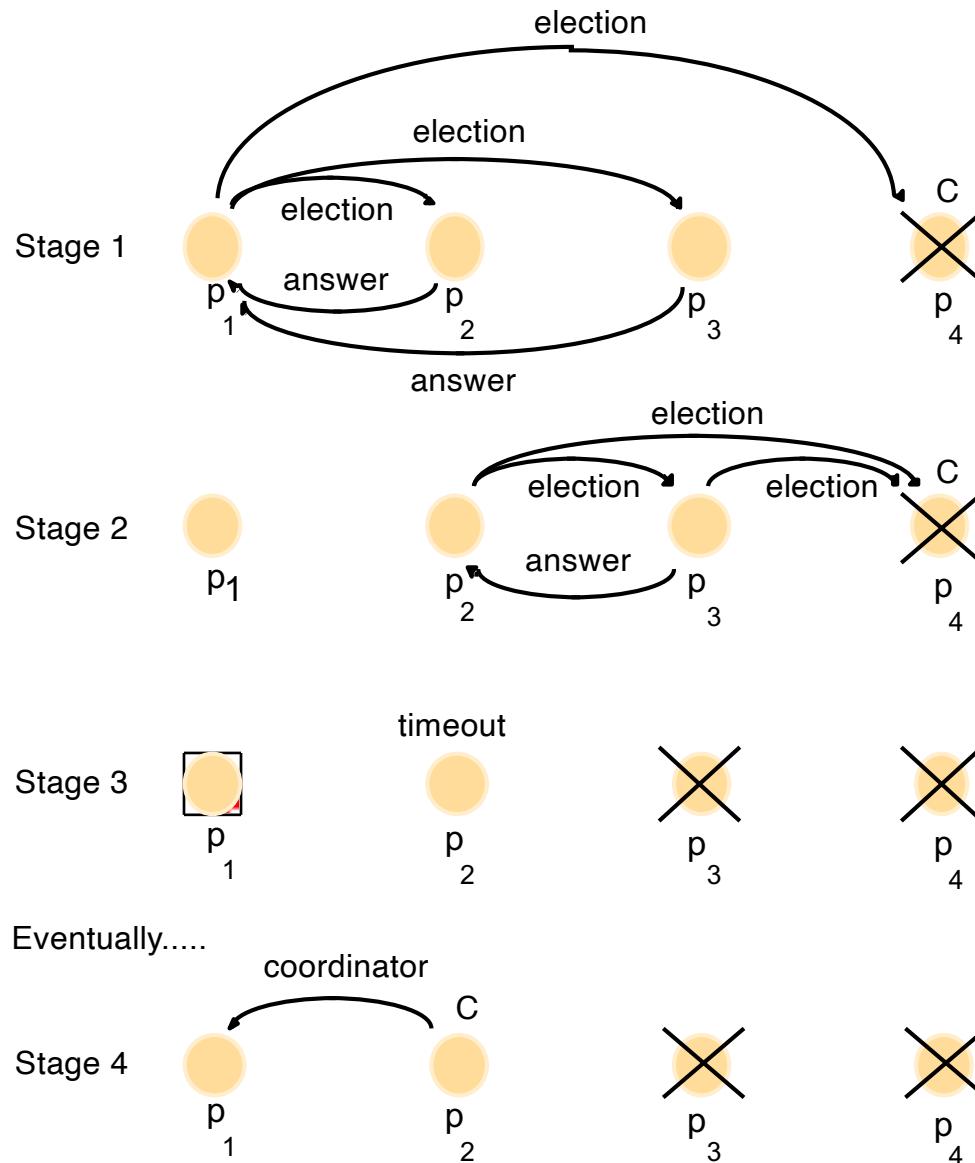
# Example: Bully Election

answer=OK



# The Bully Algorithm with Failures

The coordinator  $p_4$  fails and  $p_1$  detects this



# *Analysis of The Bully Algorithm*

- **Best case scenario:** The process with the second highest id notices the failure of the coordinator and elects itself.
  - $(N-2)$  coordinator messages are sent.
  - Turnaround time is one message transmission time.

# *Analysis of The Bully Algorithm*

- **Worst case scenario: When the process with the lowest id in the system detects the failure.**
  - N-1 processes altogether begin elections, each sending messages to processes with higher ids.
    - » i-th highest id process sends i-1 election messages
  - The message overhead is  $O(N^2)$ .
  - Turnaround time is approximately 5 message transmission times if there are no failures during the run:
    1. Election message from lowest id process
    2. Answer to lowest id process from 2<sup>nd</sup> highest id process
    3. Election from 2nd highest id process
    4. Timeout for answers @ 2nd highest id process
    5. Coordinator message from 2<sup>nd</sup> highest id process

# ***Summary***

- Coordination in distributed systems requires a leader process
- Leader process might fail
- Need to (re-) elect leader process
- Three Algorithms
  - Ring algorithm
  - Modified Ring algorithm
  - Bully Algorithm