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

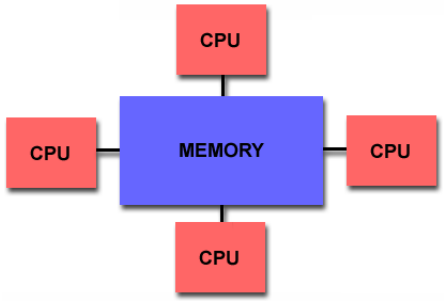
- In this the operating system picks up and begins to execute one of the jobs from memory.
- Once this job needs an I/O operation operating system switches to another job (CPU and OS always busy).
- Jobs in the memory are always less than the number of jobs on disk(Job Pool).
- If several jobs are ready 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..

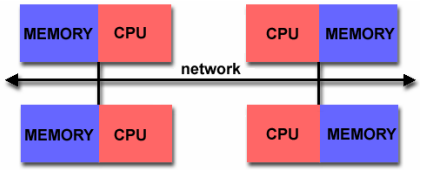
- Distribute the computation among several physical processors.
- Due to low data transfer rate and high message delay, these systems are also called loosely coupled systems
- *Loosely coupled system* – each processor has its own local memory; processors communicate with one another through various communications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

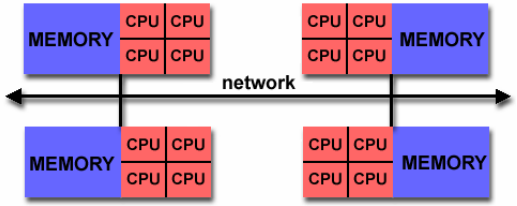
Distributed Systems

  - Scalability
  - Reliability
  - Fail-Safe
  - Communications
- OS has to cater for resource sharing.

Hybrid Systems

- Also called as Multi-Computing systems





contd..

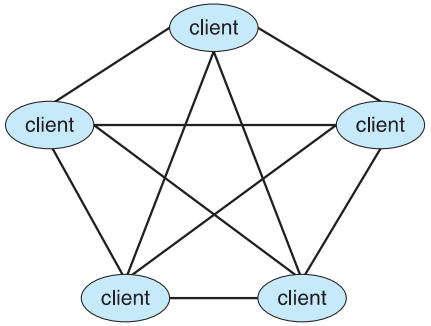
Client-Server Systems

- **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 assurity 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.

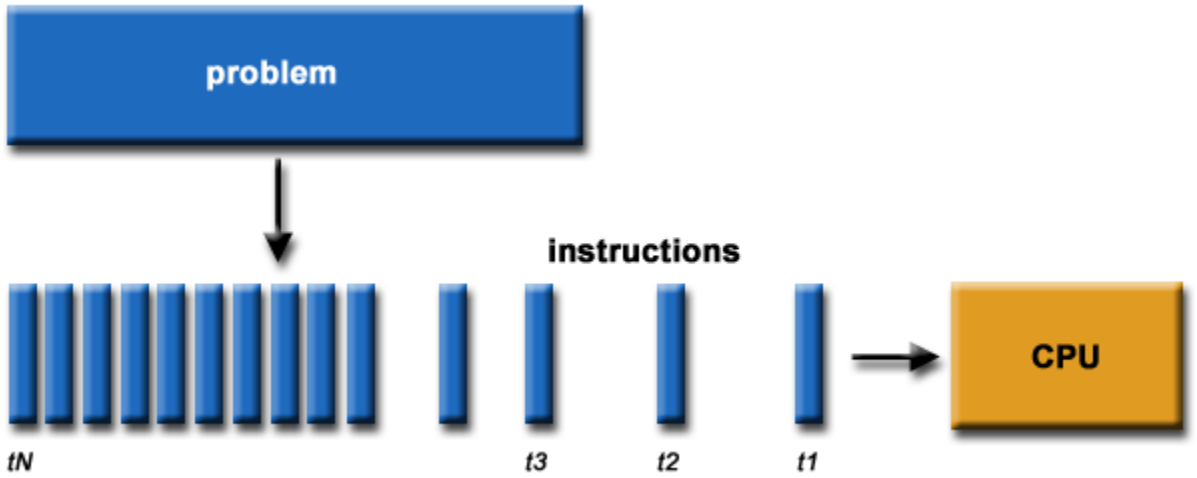
Handheld Systems

- 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

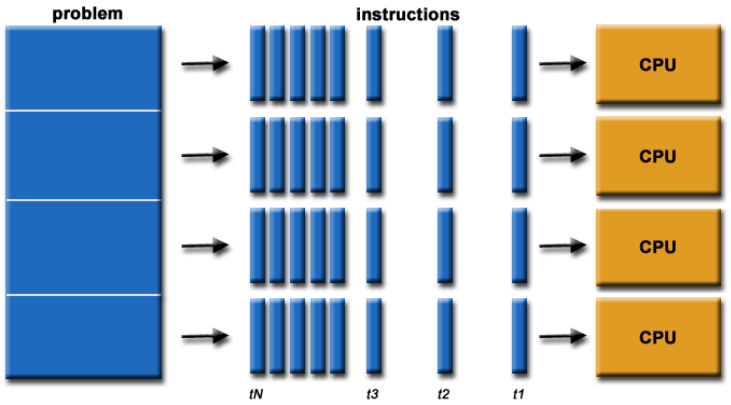
Serial Computation

- 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

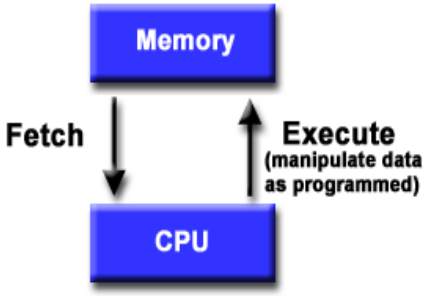
- Von Newmann Architecture
- Flynn's Classical Taxonomy

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

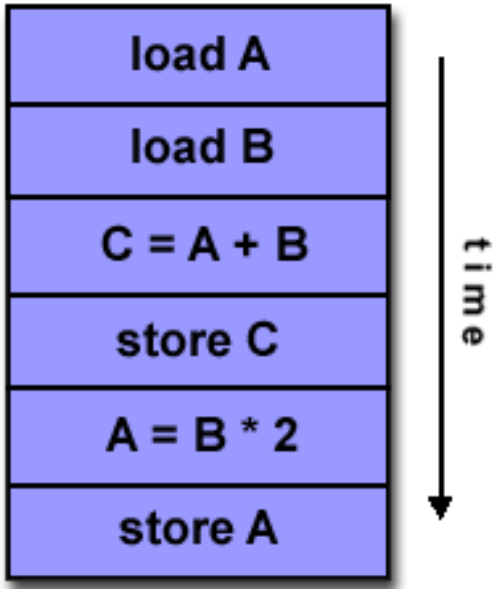
- There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.

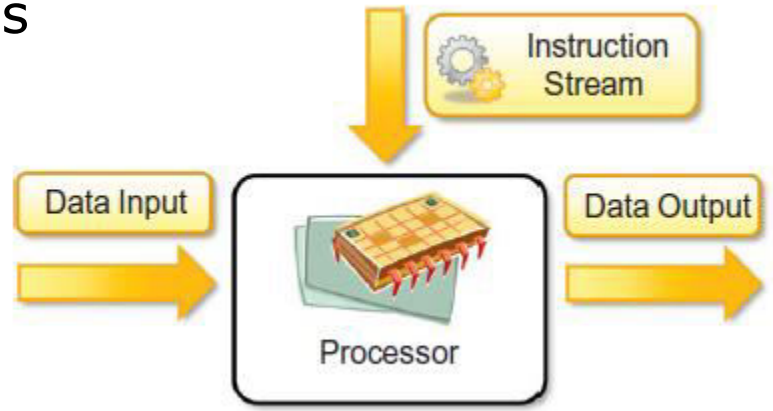
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction*** and ***Data***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

- 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

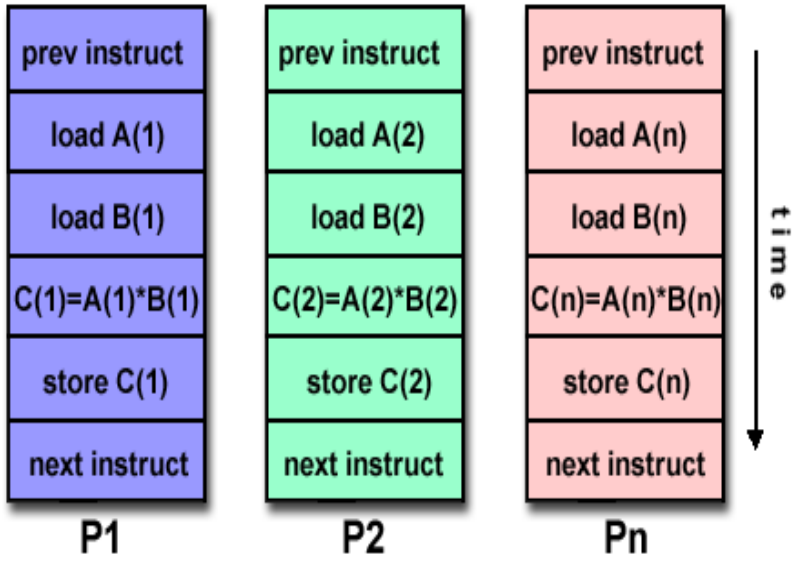
- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most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



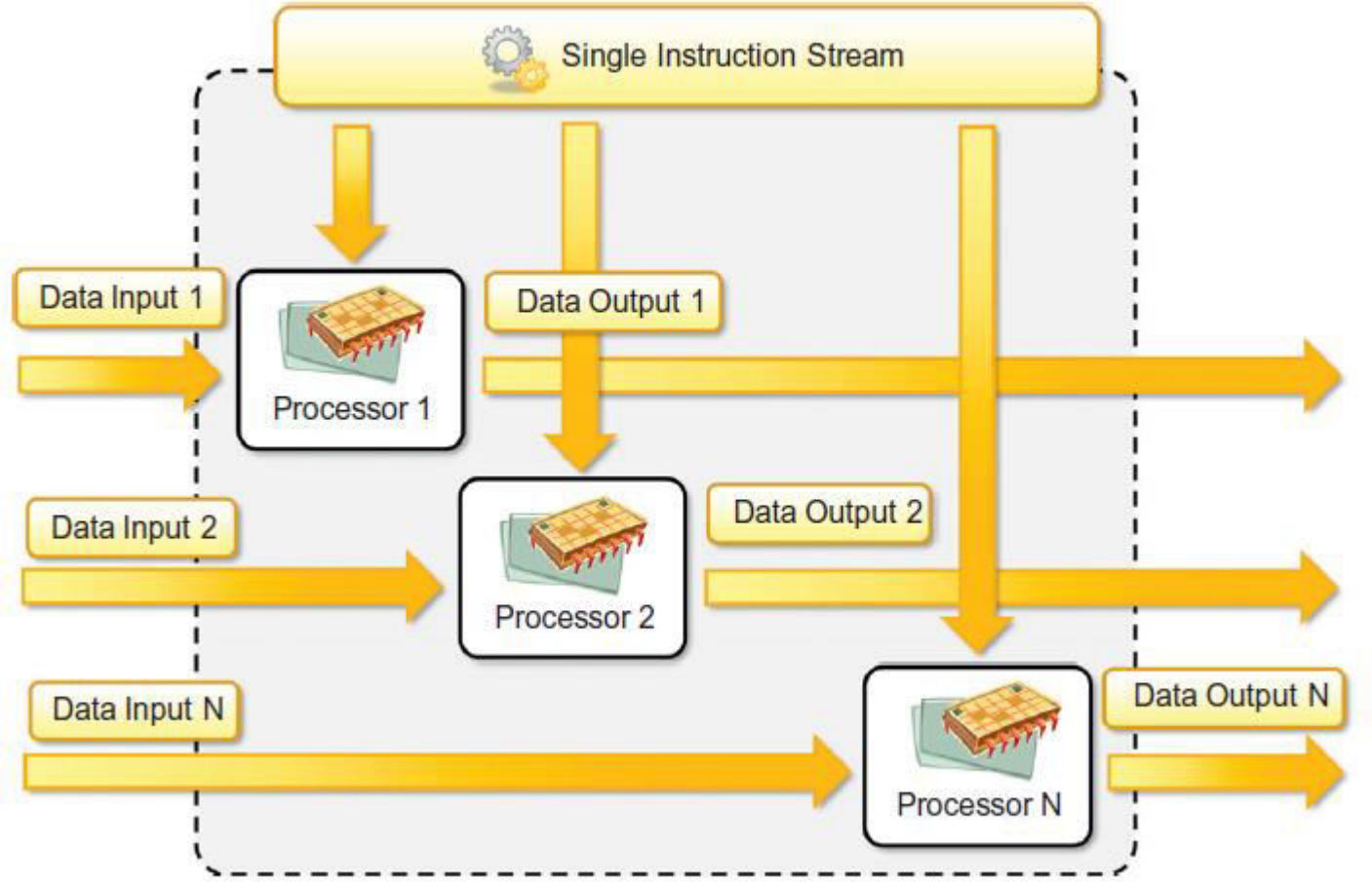


Single Instruction Multiple Data

- 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

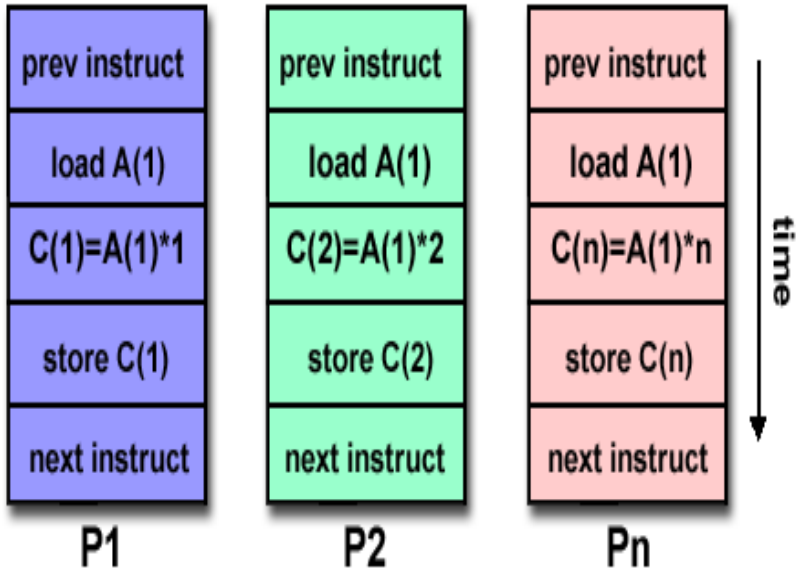


SIMD (contd..)

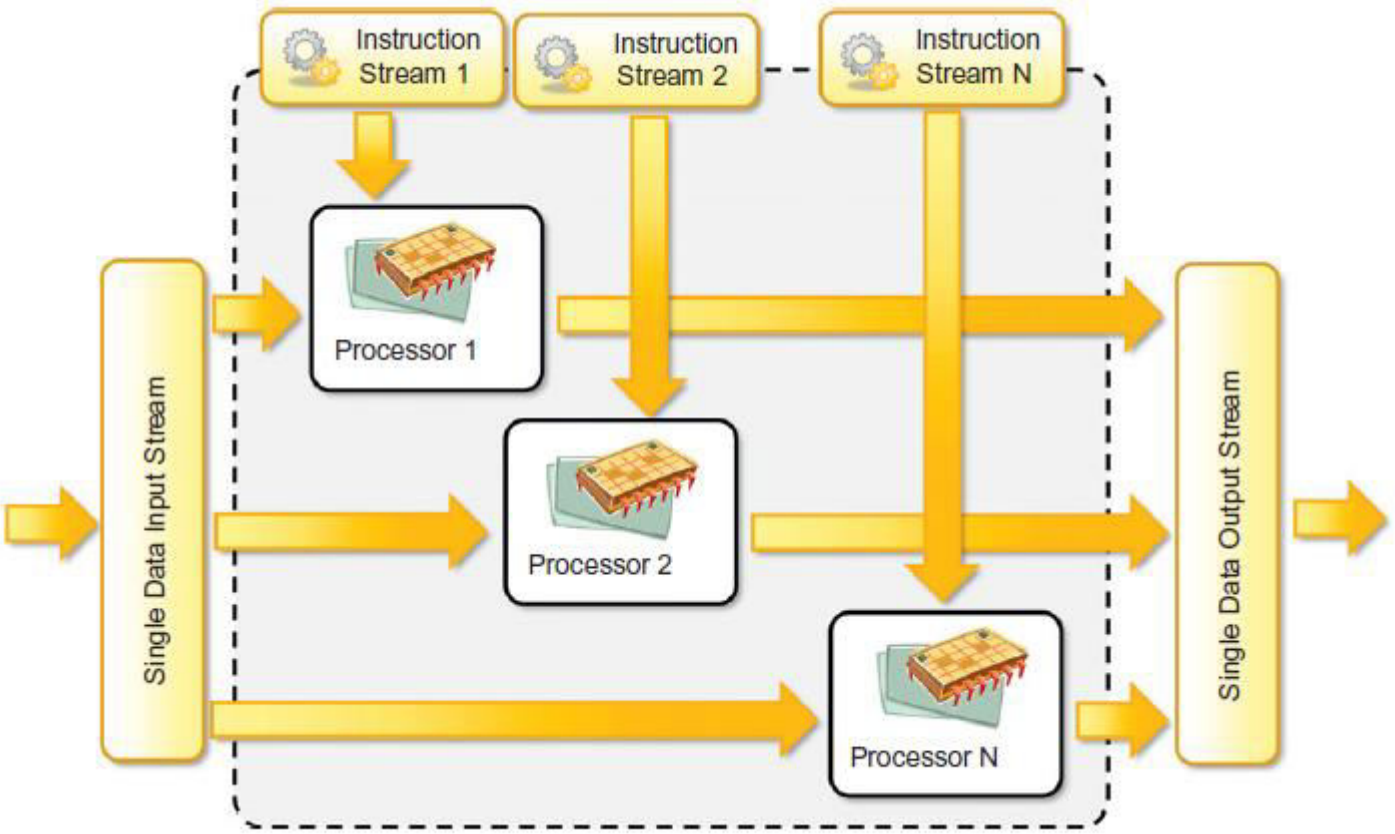

Single Instruction Stream
Data Input 1
Processor 1
Data Output 1
Data Input 2
Processor 2
Data Output 2
Data Input N
Processor N
Data Output N

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.

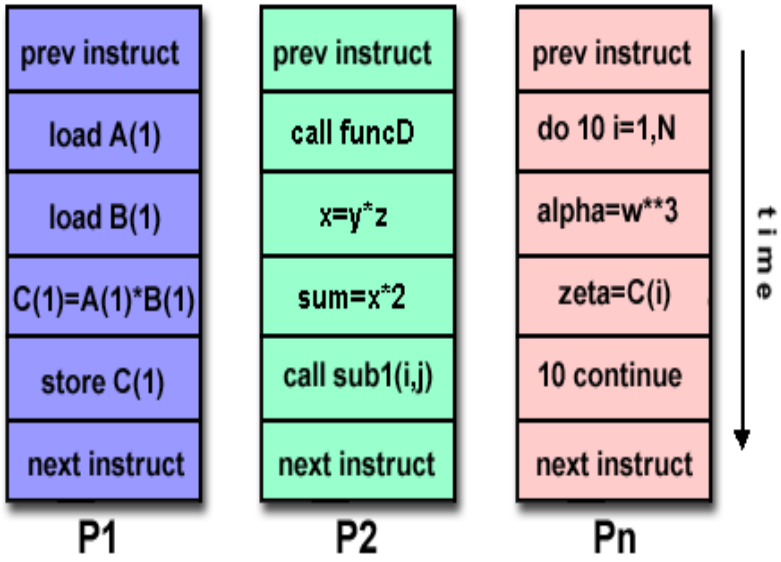


MISD (contd..)

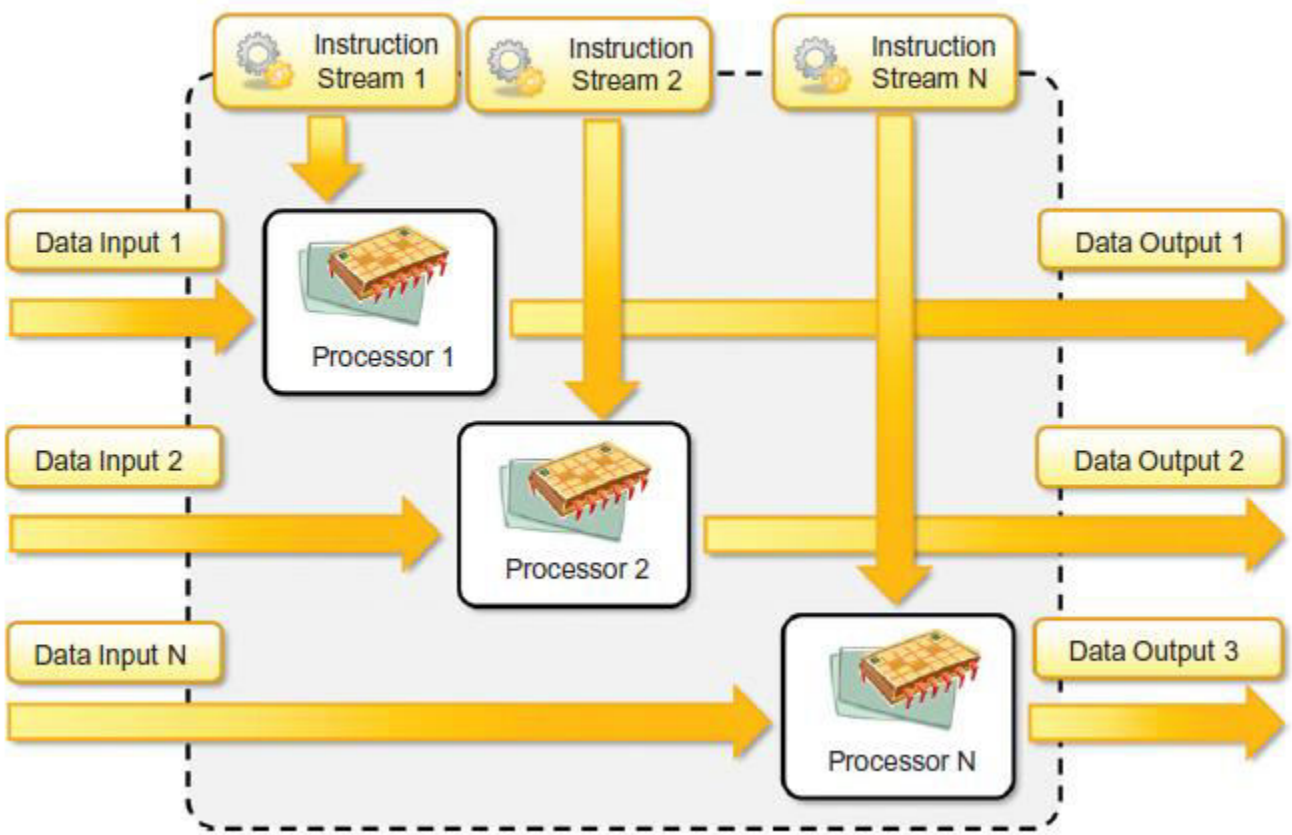

Instruction Stream 1
Instruction Stream 2
Instruction Stream N
Processor 1
Processor 2
Processor N
Single Data Input Stream
Single Data Output Stream

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..)

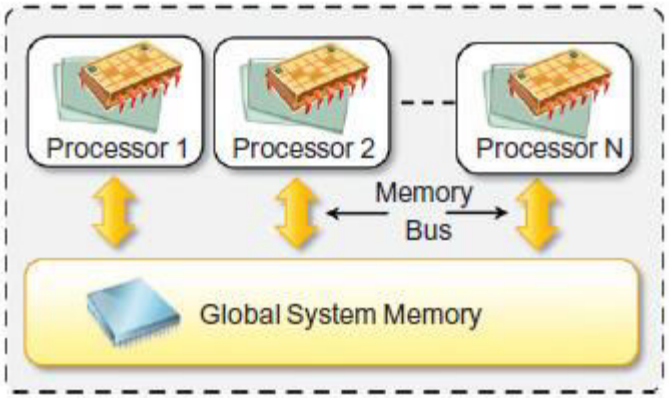

Instruction Stream 1
Instruction Stream 2
Instruction Stream N
Data Input 1
Data Input 2
Data Input N
Processor 1
Processor 2
Processor N
Data Output 1
Data Output 2
Data Output 3

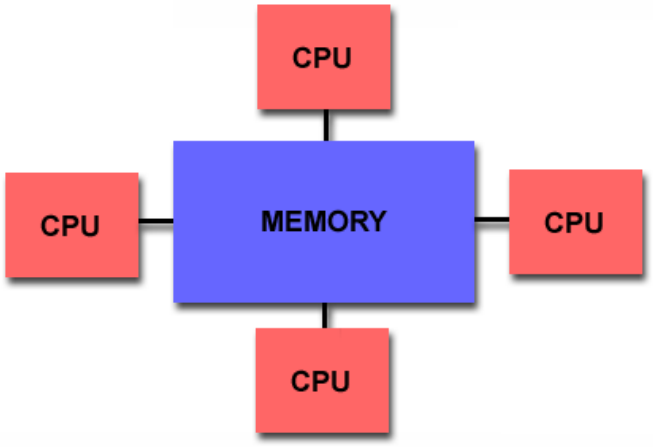
Parallel Computer Memory Architectures

- 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***.



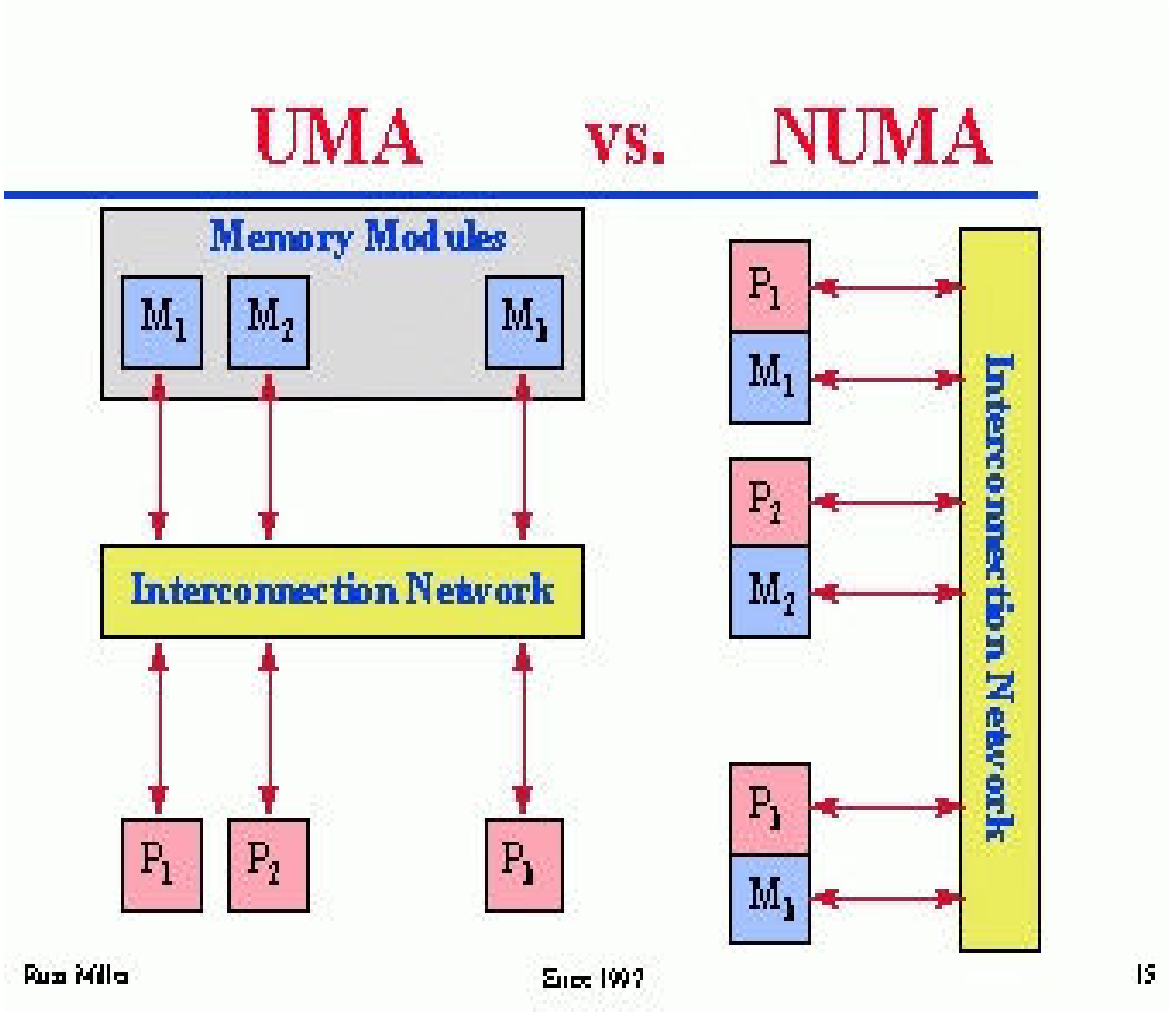


Shared Memory

- 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
Memory Modules
M1
M2
M3
Interconnection Network
P1
P2
P3
P1
M1
P2
M2
P3
M3
Interconnection Network

Shared Memory

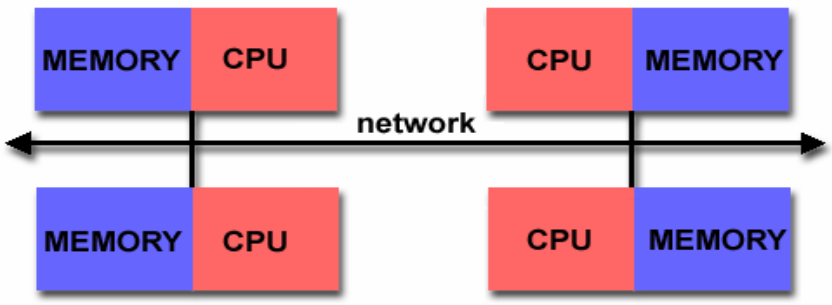
- Advantages:
  - Global address space provides a user-friendly programming perspective to memory
  - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

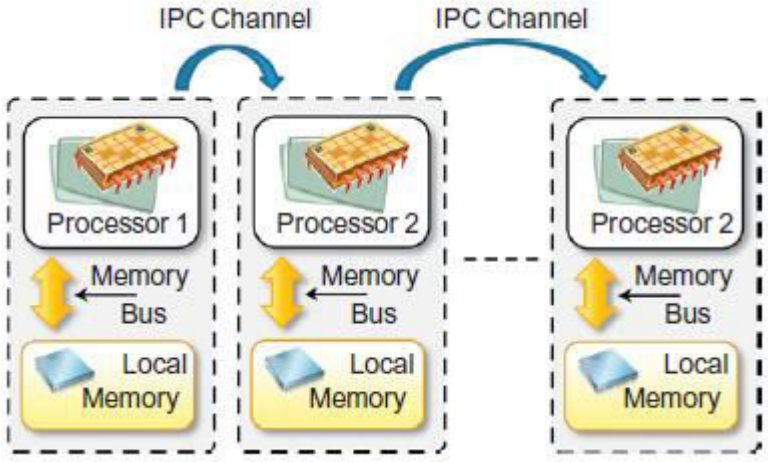
- Disadvantages:
  - Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
  - Programmer responsibility 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 can be as simple as Ethernet.

contd..


MEMORY
CPU
CPU
MEMORY
network
MEMORY
CPU
CPU
MEMORY


IPC Channel
IPC Channel
Processor 1
Processor 2
Processor 2
Memory Bus
Memory Bus
Memory Bus
Local Memory
Local Memory
Local Memory

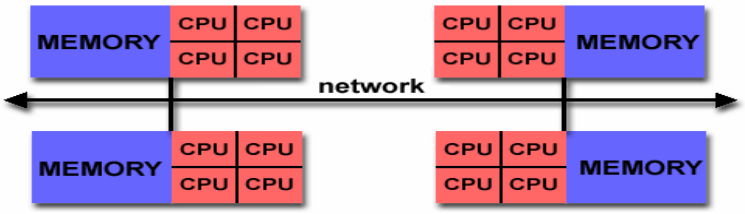
Distributed Memory

- 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.

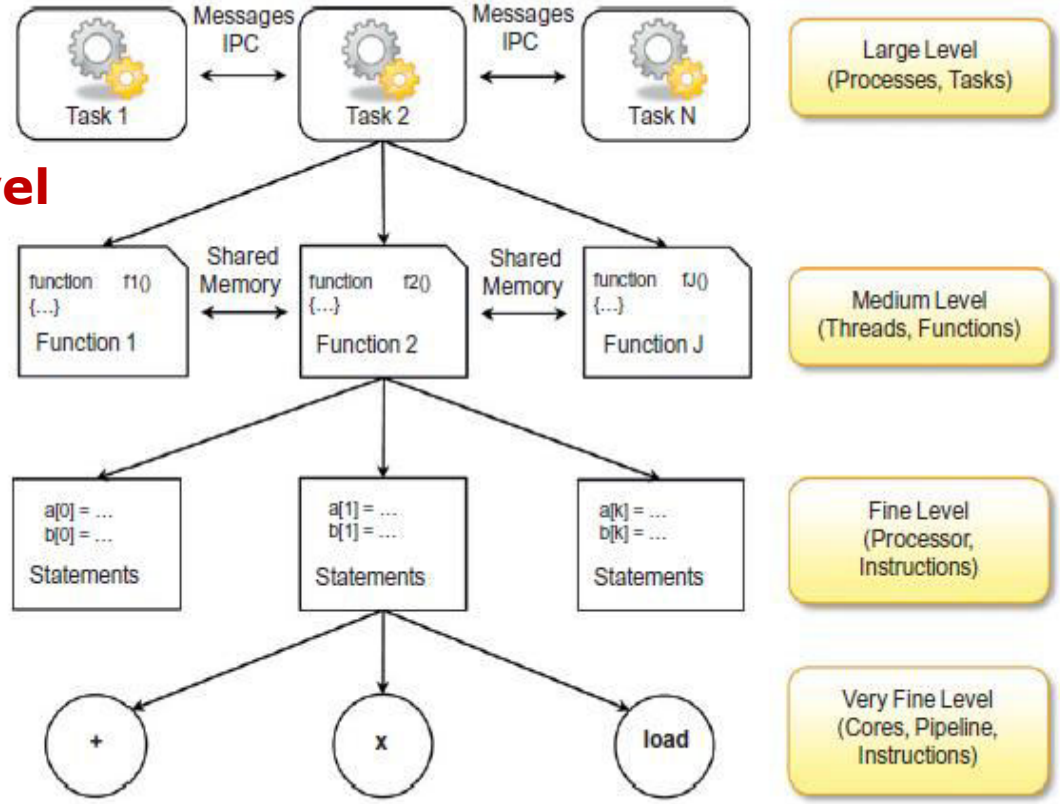
contd..

- 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

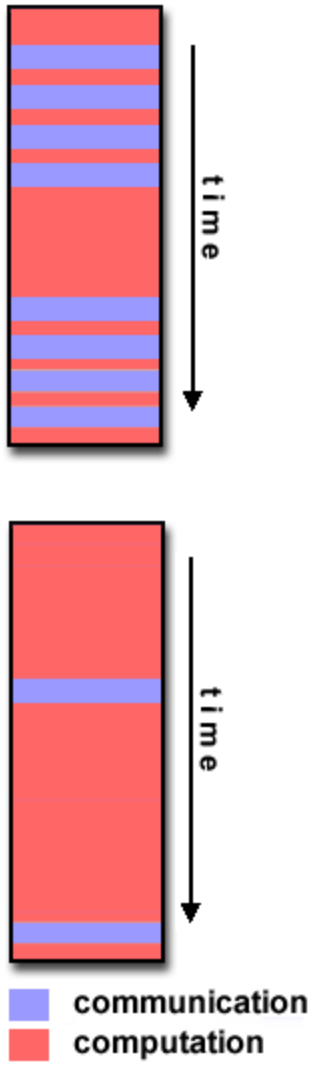
- Levels of parallelism are decided based on the lumps of code (grain size) that can be a potential candidate for parallelism.

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



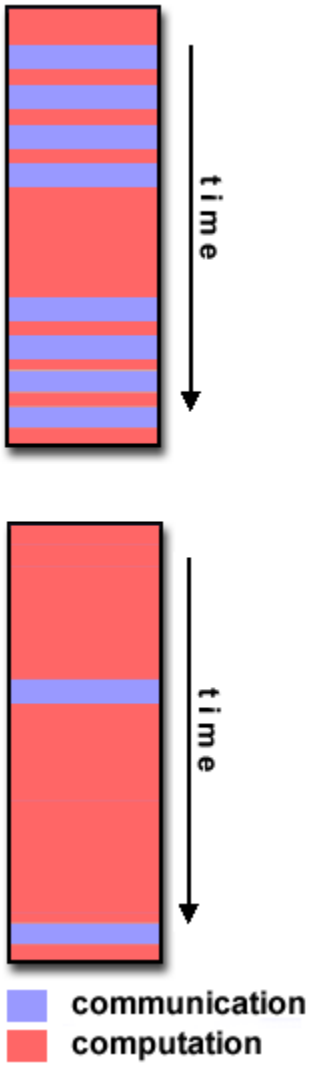
Granularity contd..

- 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.



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

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

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

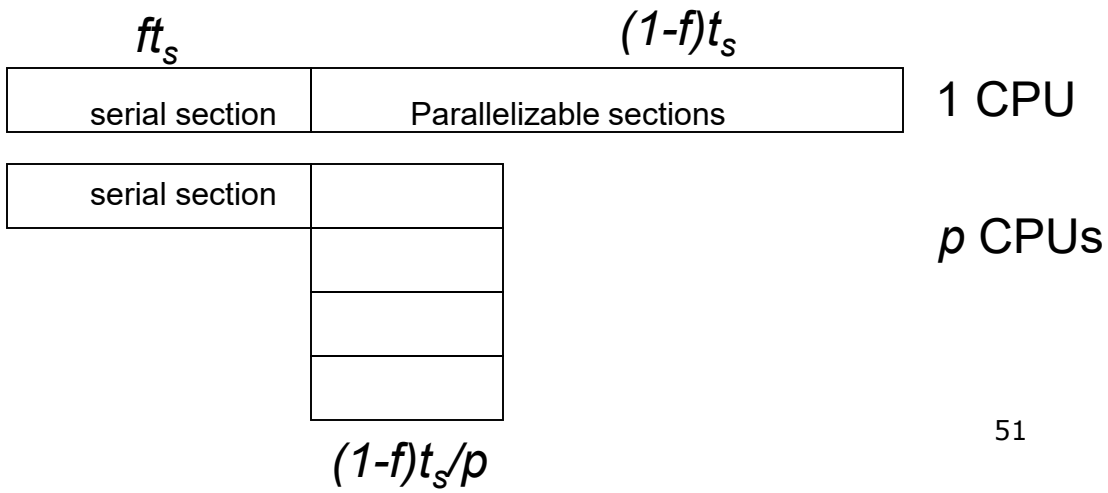
$$E = \frac{S(p)}{p} \times 100\%$$

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 in to concurrent tasks is *f*.
- The time used to perform computation with *p* processors is

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

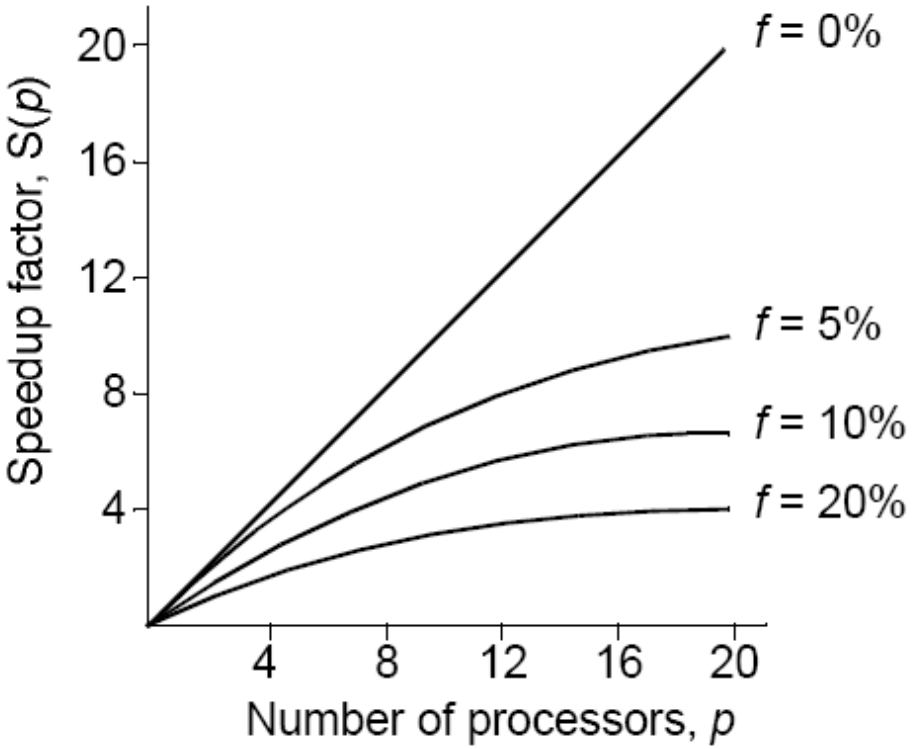


Amdahl's Law

- The speedup factor is given as

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

$$\lim_{p \to \infty} S(p) = \frac{1}{f}$$



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.

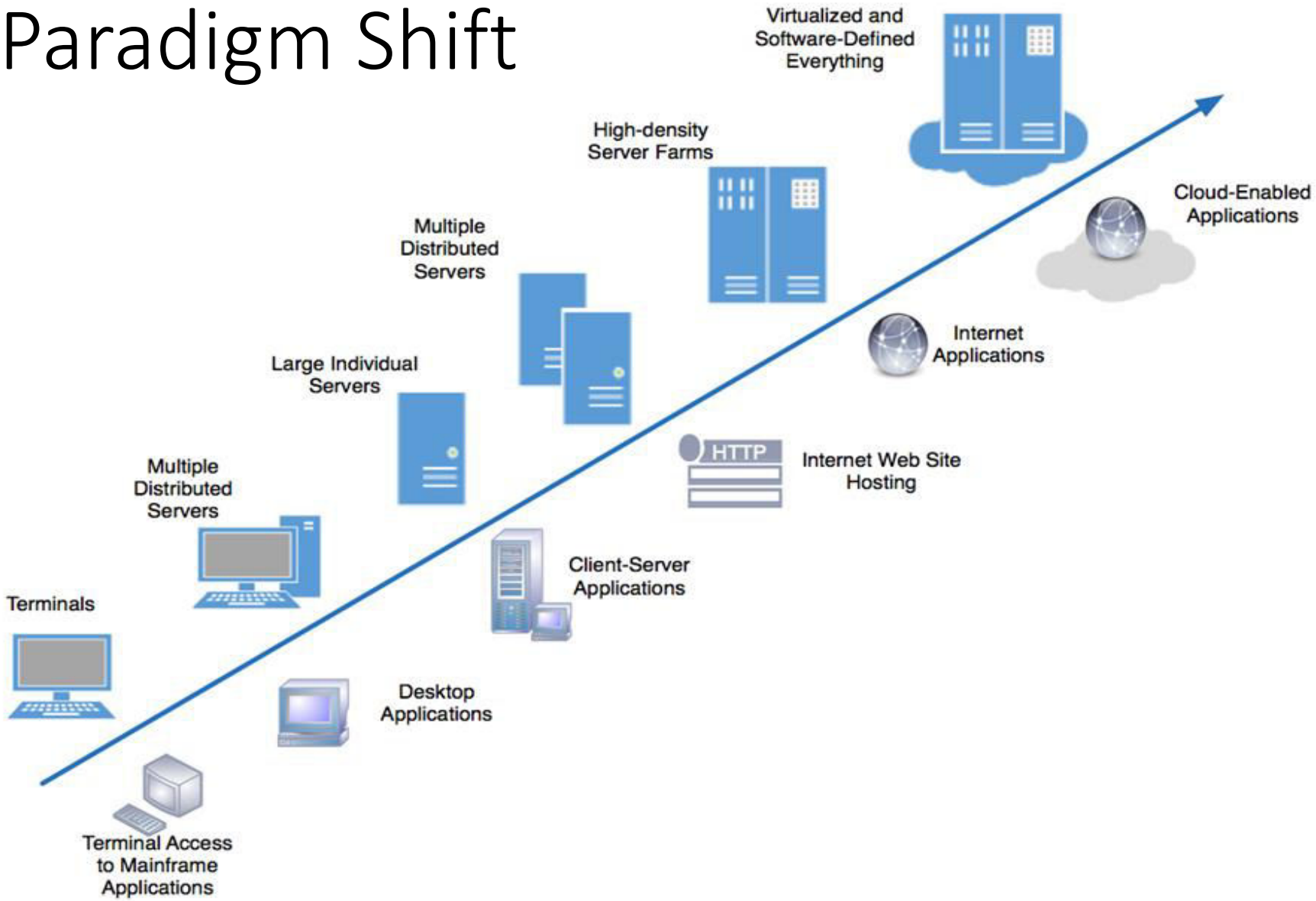
Parallel Terminology

- **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.

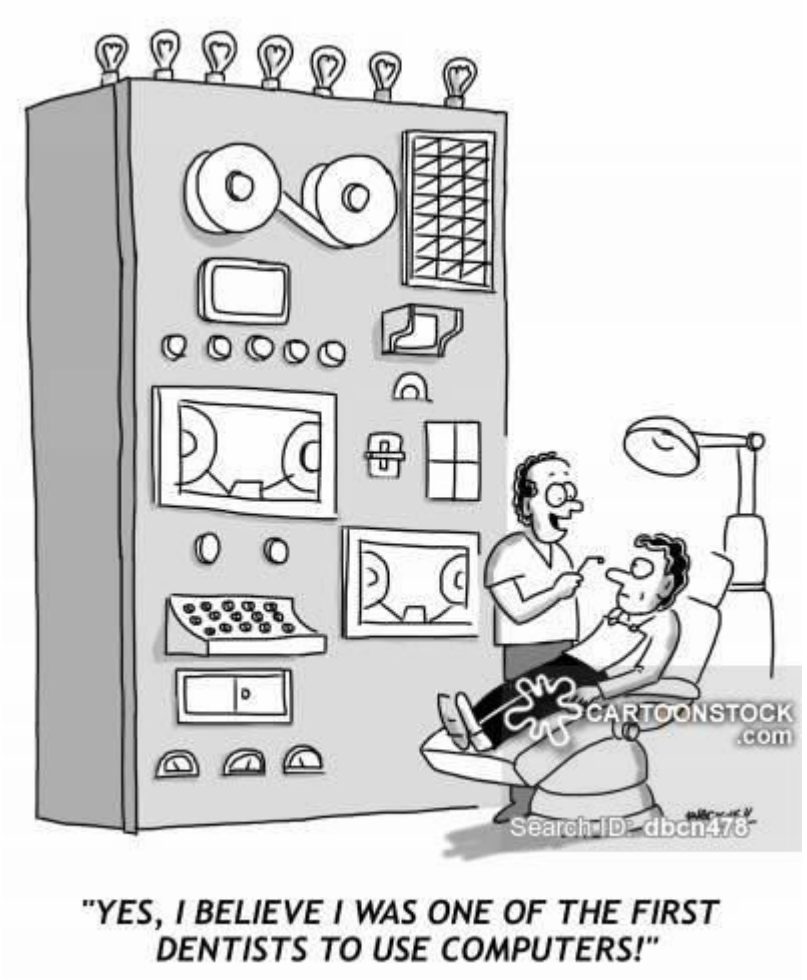
- **Scalability**
  - Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:
    - Hardware - particularly memory-cpu bandwidths and network communications
    - 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!"

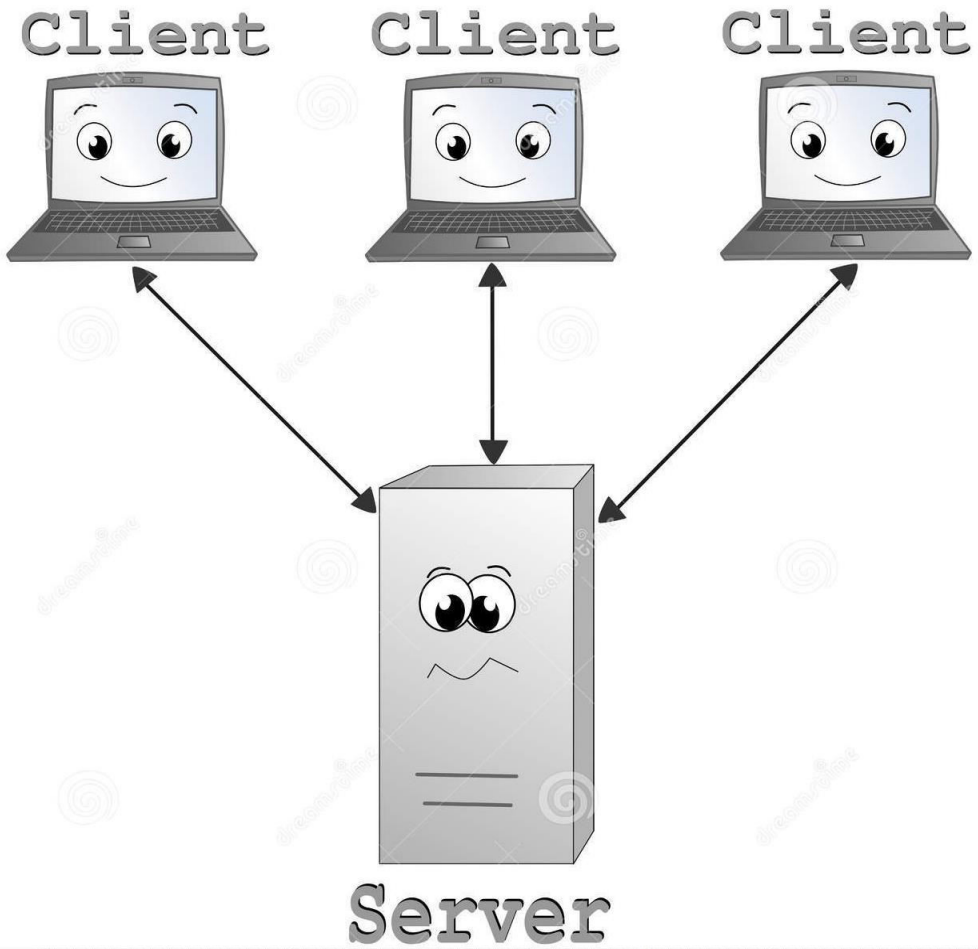
- Jobs
- Batches
- Processing
- To carry out some mundane and routine jobs such as payroll, accounts, inventory thus sparing employees from tedious jobs.
- It was available in one location, and anyone who needs it must go to computer center for availing it.

Personal Computing

"Tech support says the problem is located somewhere between the keyboard and my chair."

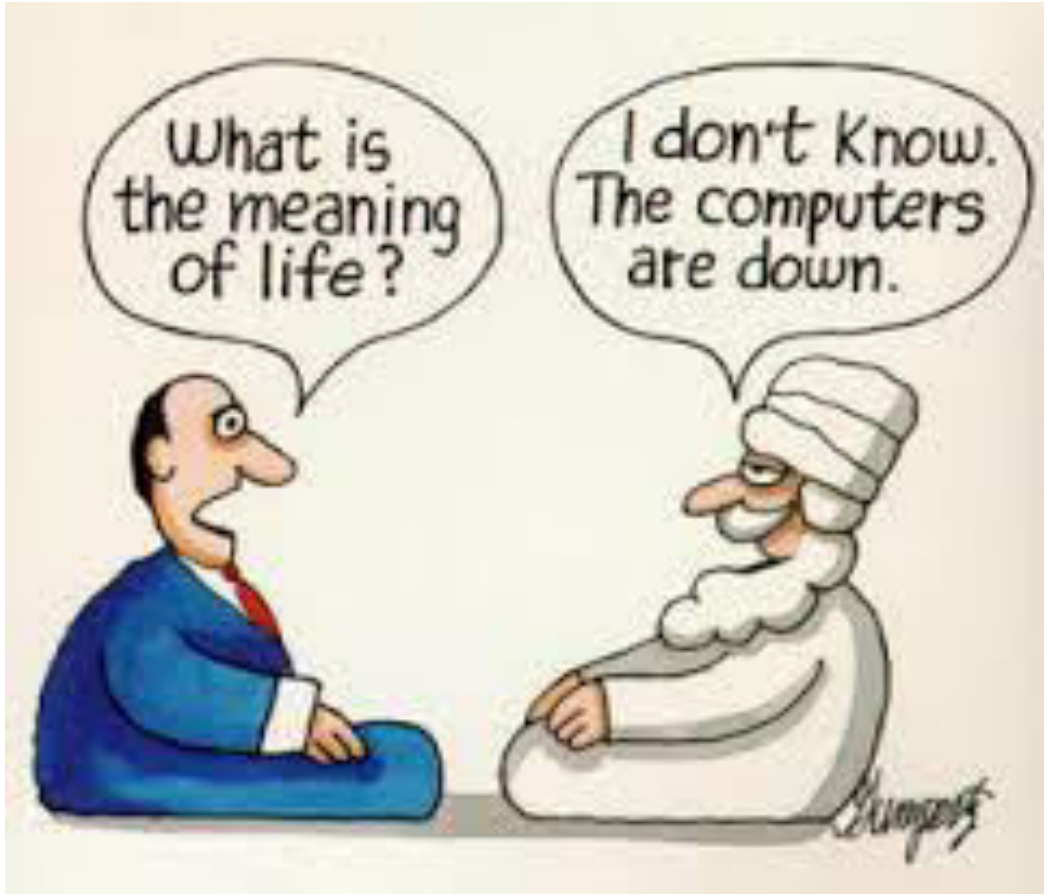
- Desktop computing - personal computer small enough to fit conveniently in an individual workspace.
- Providing computers to each employee on their desktop or workspace.
- Decentralized computing.
- Less expensive, easy to upgrade and less accessories needed.
- Information sharing with other users is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

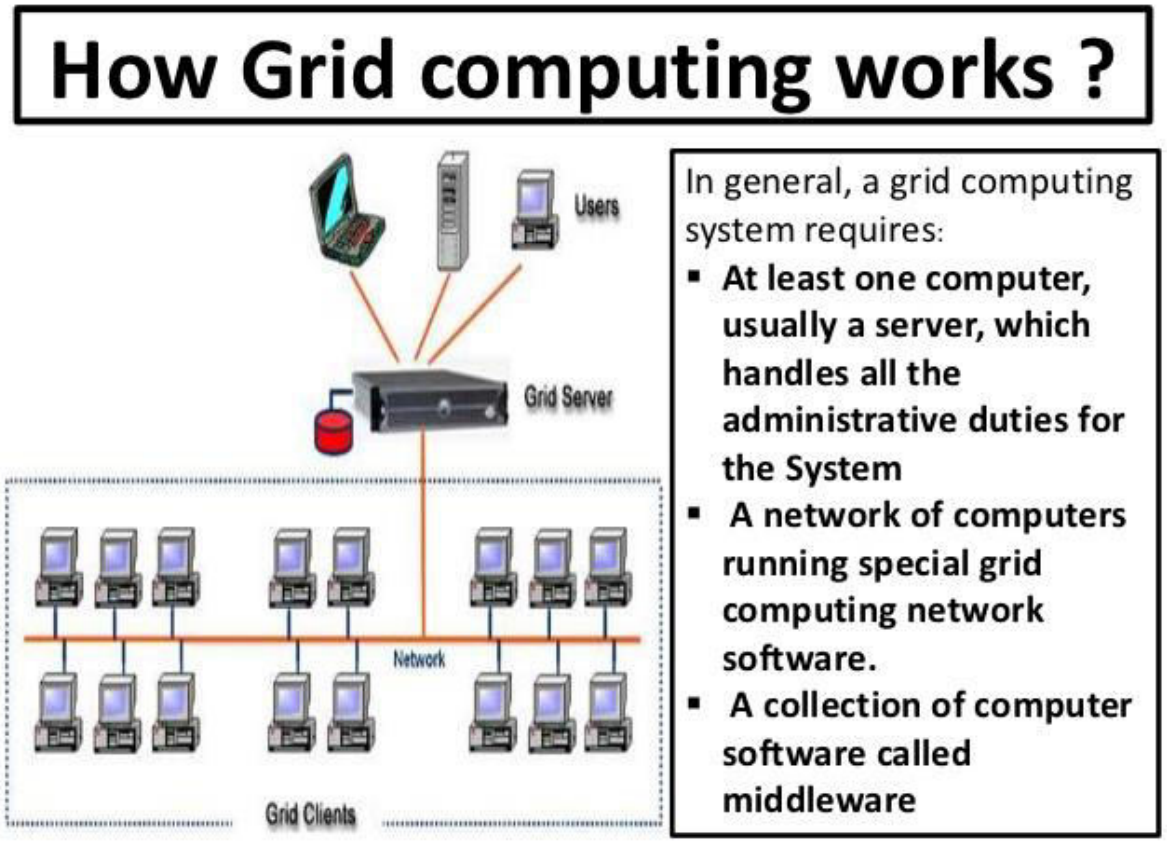
Benefits

- 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



- 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

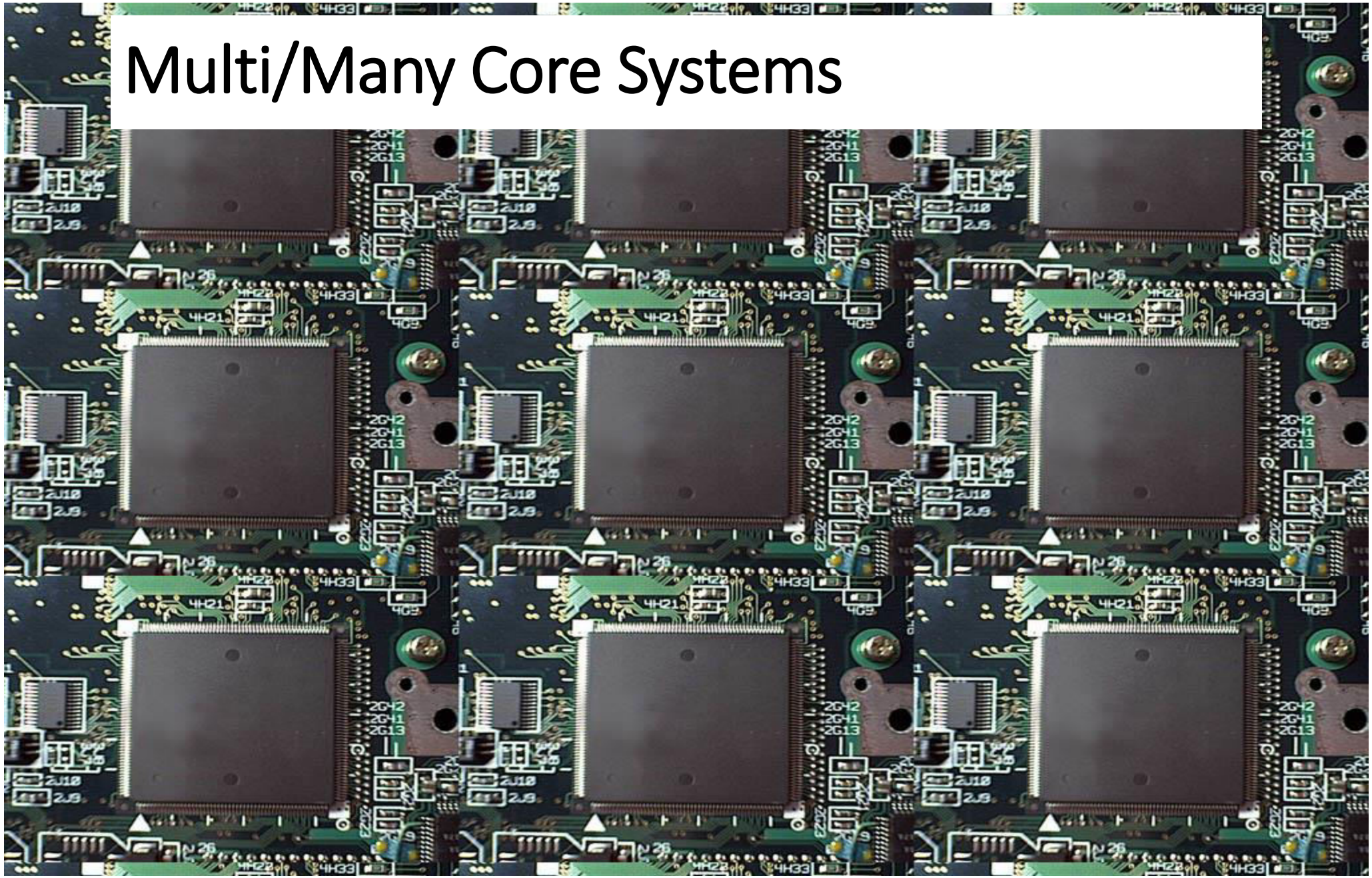
Drawbacks

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

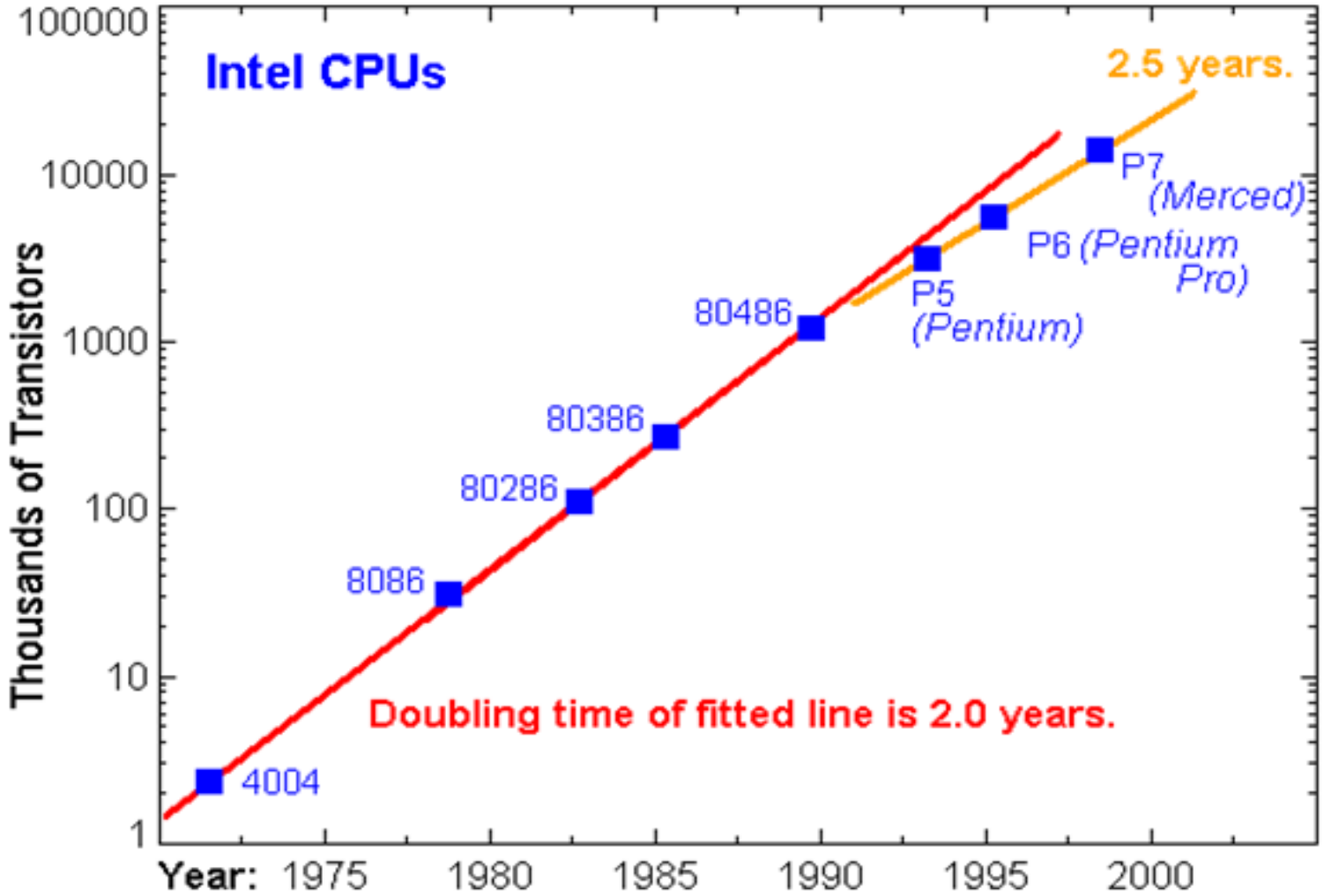
Grid Vs Cluster Vs Cloud Computing

Properties	Cloud	Cluster	Grid
On-demand self-Service			
Broad network access			
Resource pooling			
Rapid elasticity			
Measured service			

Multi/Many Core Systems



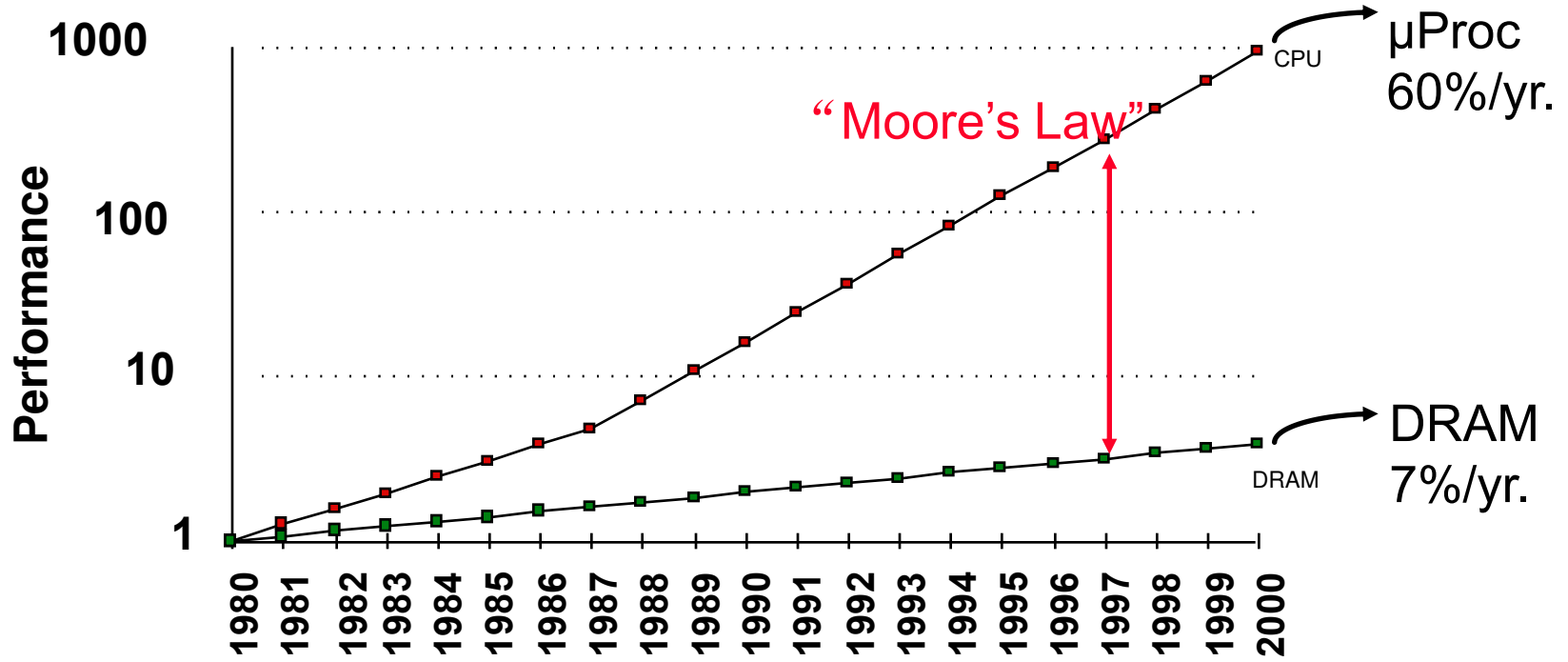
Moore's Law


Intel CPUs
2.5 years.
Thousands of Transistors
100000
10000
1000
100
10
1
Year: 1975
1980
1985
1990
1995
2000
4004
8086
80286
80386
80486
P5
(Pentium)
P6 (Pentium
Pro)
P7
(Merced)
Doubling time of fitted line is 2.0 years.

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

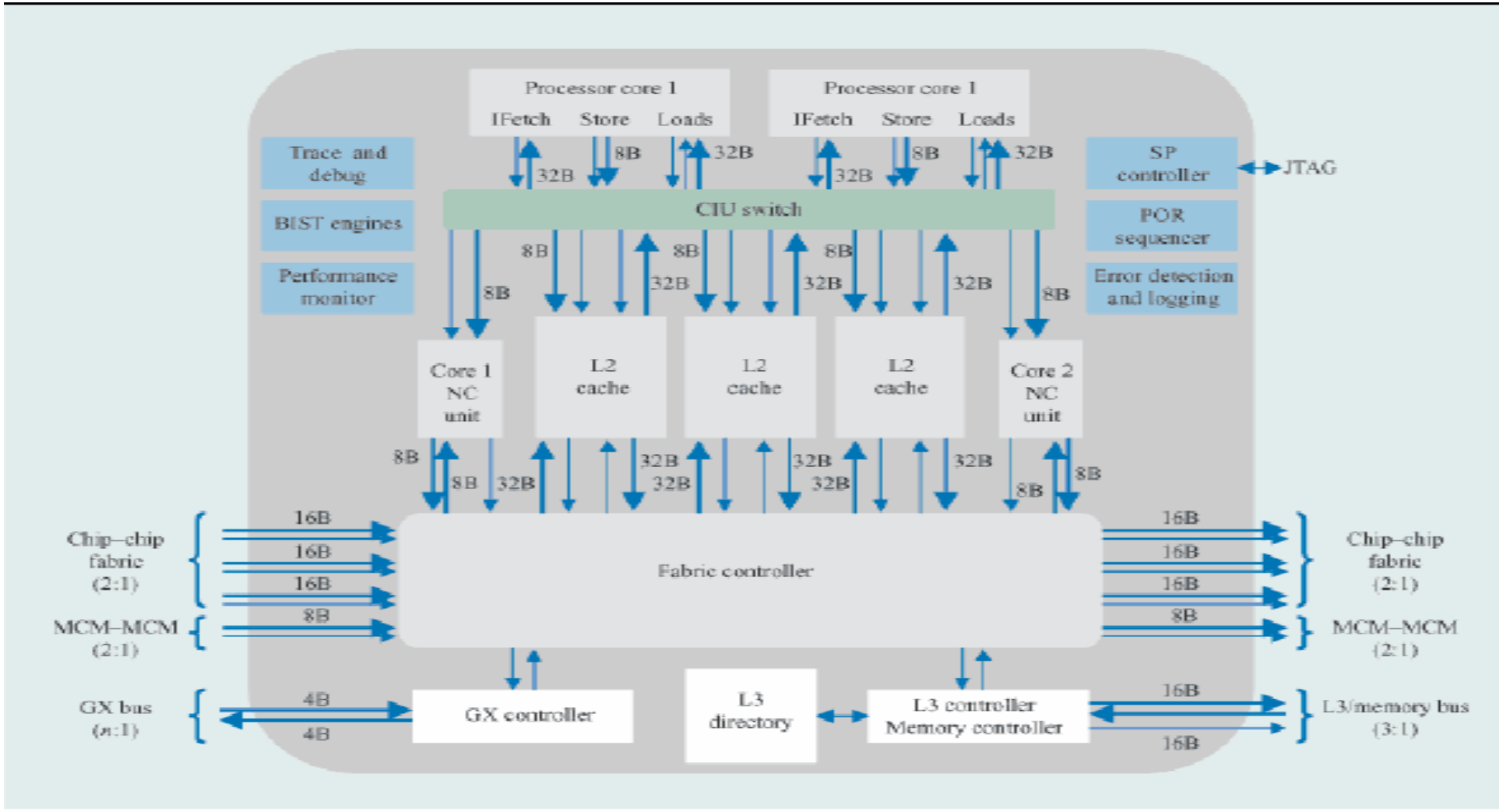


Figure 1

POWER4 chip logical view.

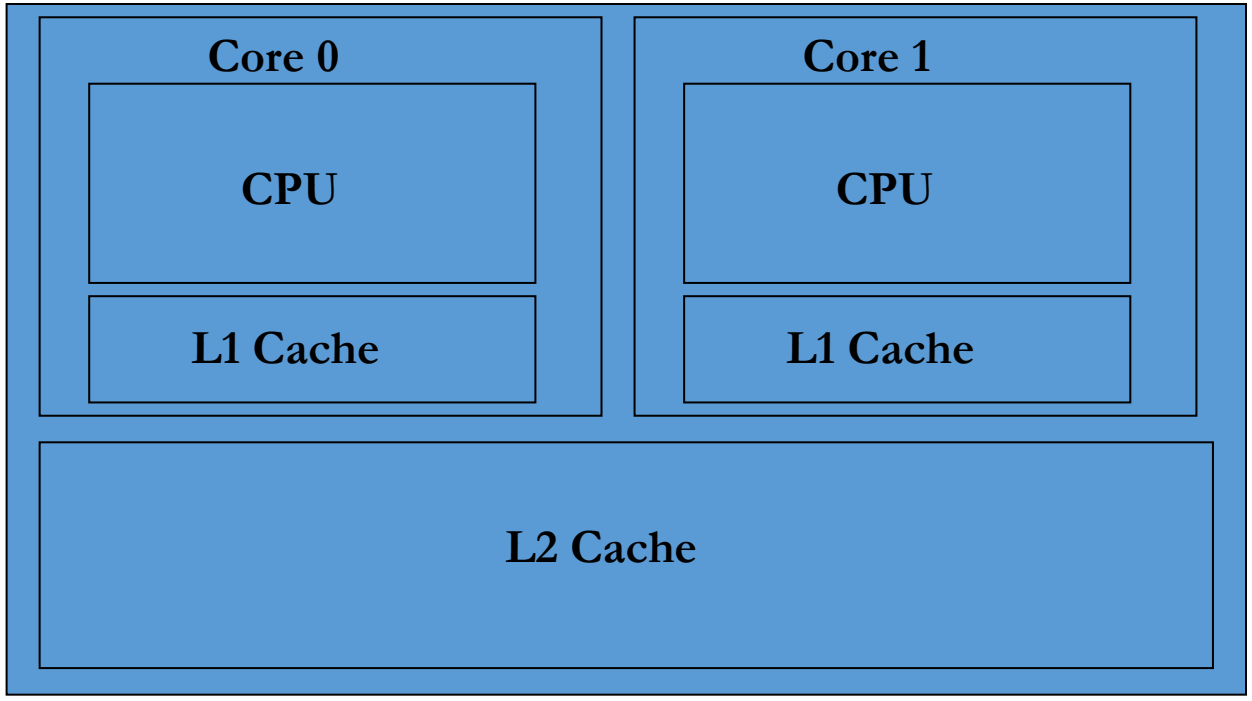
Definitions

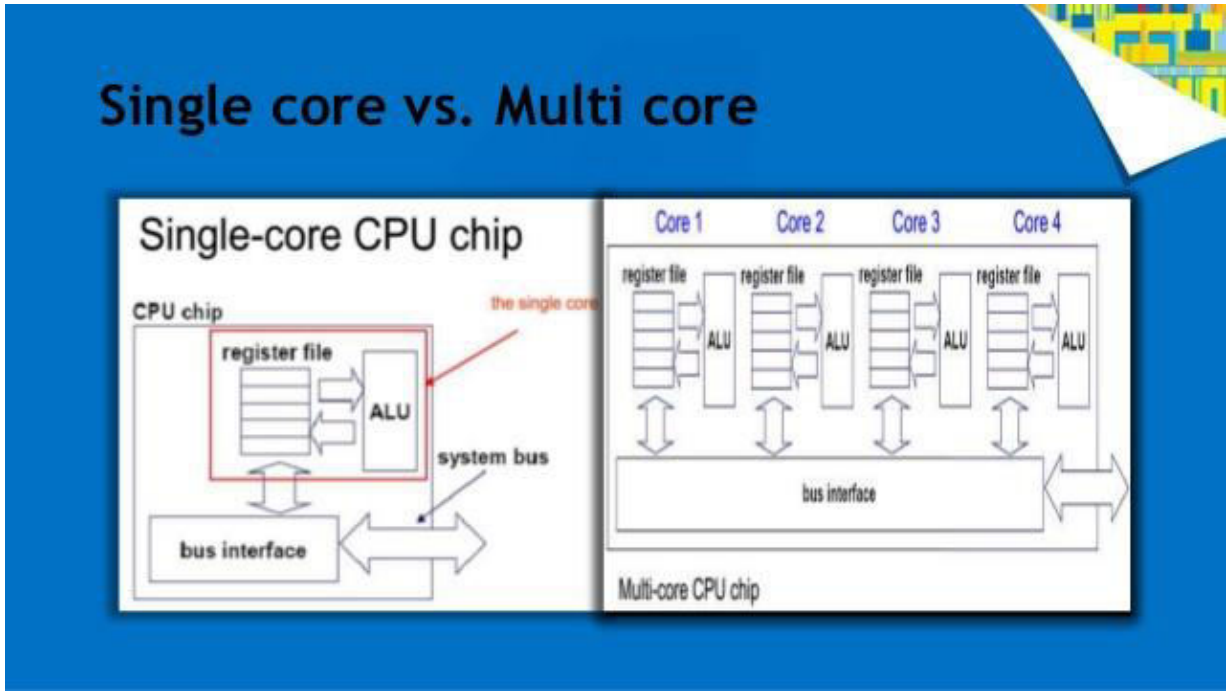
- A **Multi-Core** processor combines two or more independent **cores** into a single package composed of a single integrated circuit (IC), called a die, or more dies packaged together.

  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)?

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):

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

Communication (for steps 1 and 3):

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

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 $\mathbf{t_{start\text{-}up}}$ and $\mathbf{t_{data}}$ 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 is1000**

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

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

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

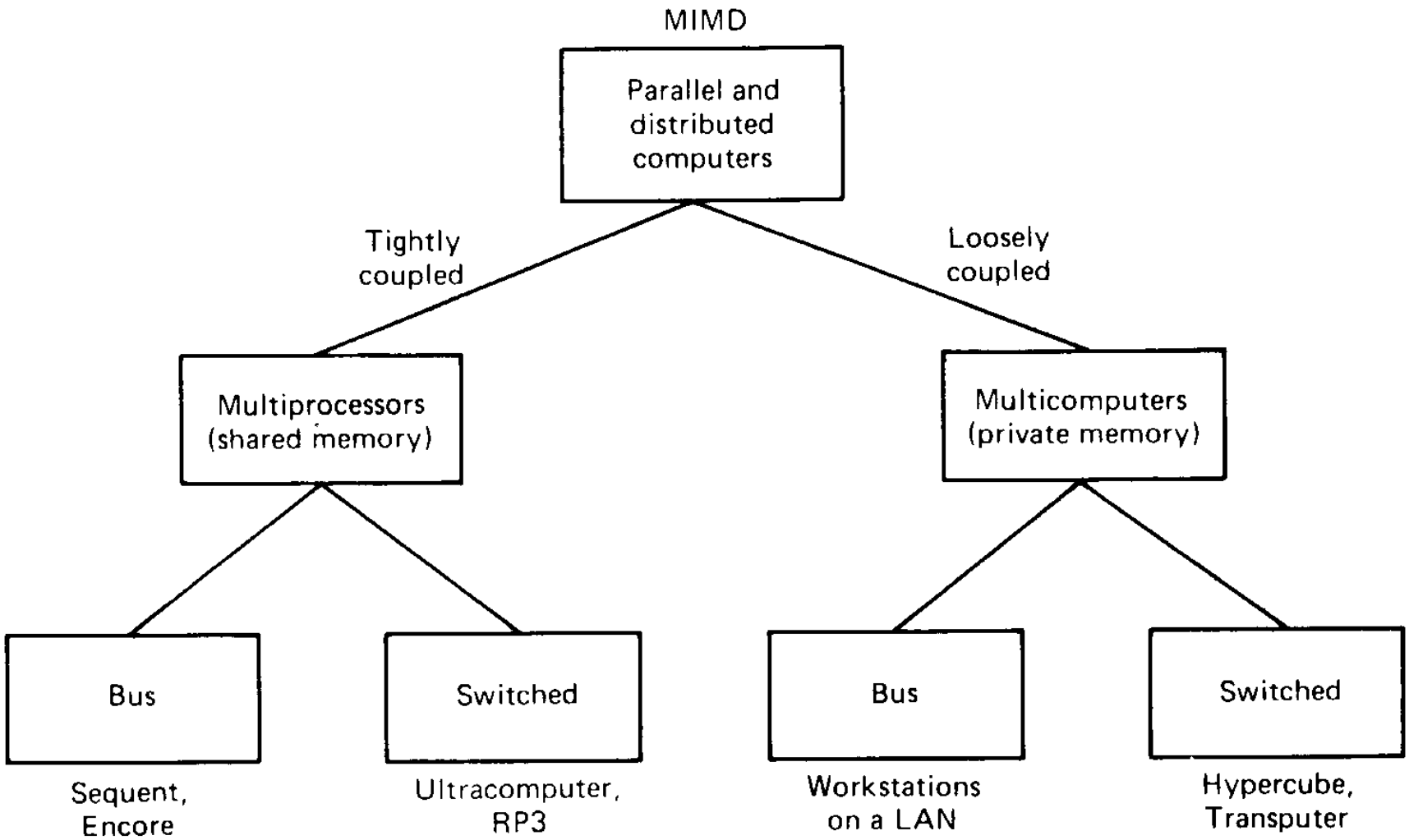
Example 3 (p.40): Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60%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

Answer

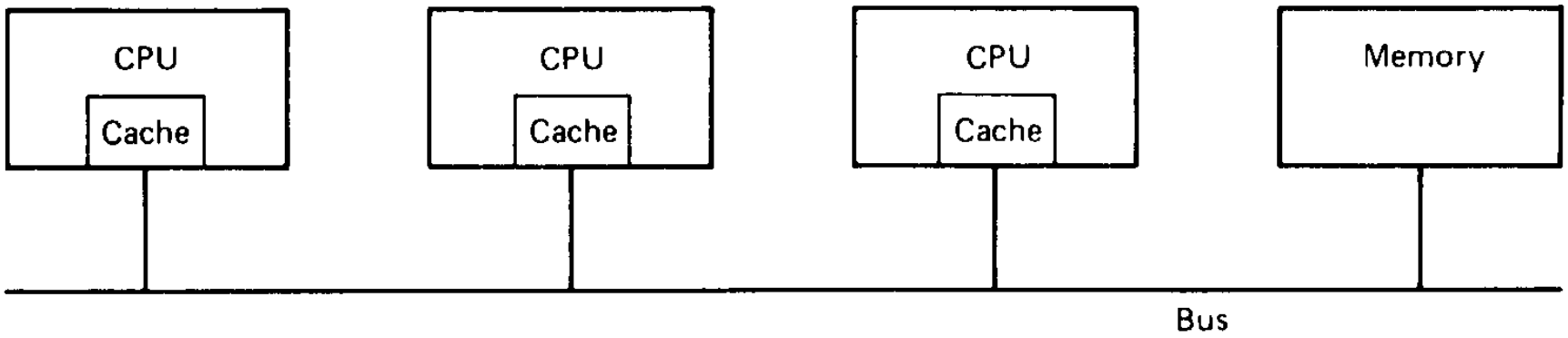
$$\text{Fraction}_{\text{enhanced}} = 0.4, \text{Speedup}_{\text{enhanced}} = 10$$

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

Communication / Interconnection Networks

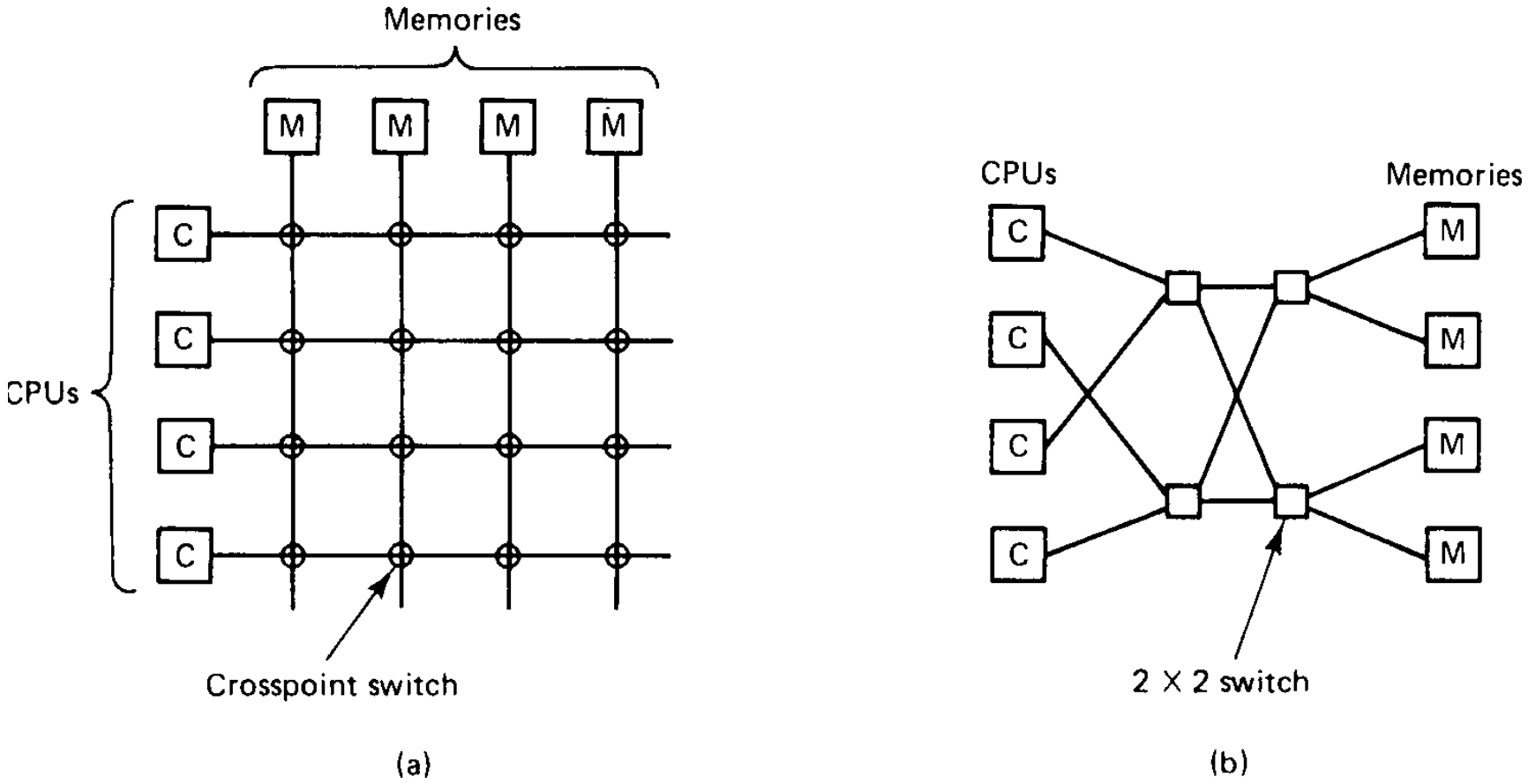

MIMD
Parallel and distributed computers
Tightly coupled
Loosely coupled
Multiprocessors (shared memory)
Multicomputers (private memory)
Bus
Switched
Bus
Switched
Sequent, Encore
Ultracomputer, RP3
Workstations on a LAN
Hypercube, Transputer

Bus based Multiprocessors



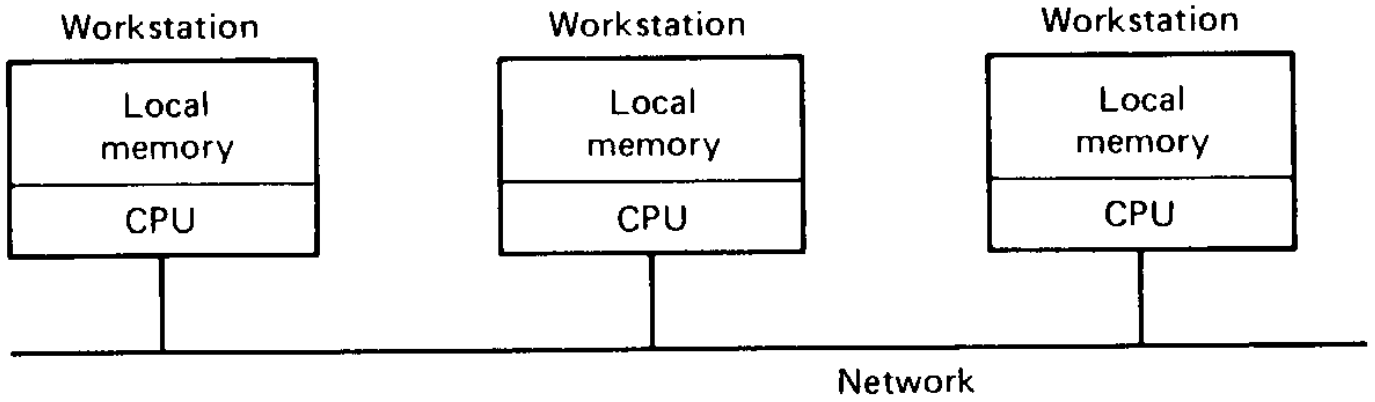
- Data transmitted in the form of packets
- Cache coherency

Switched Multiprocessors



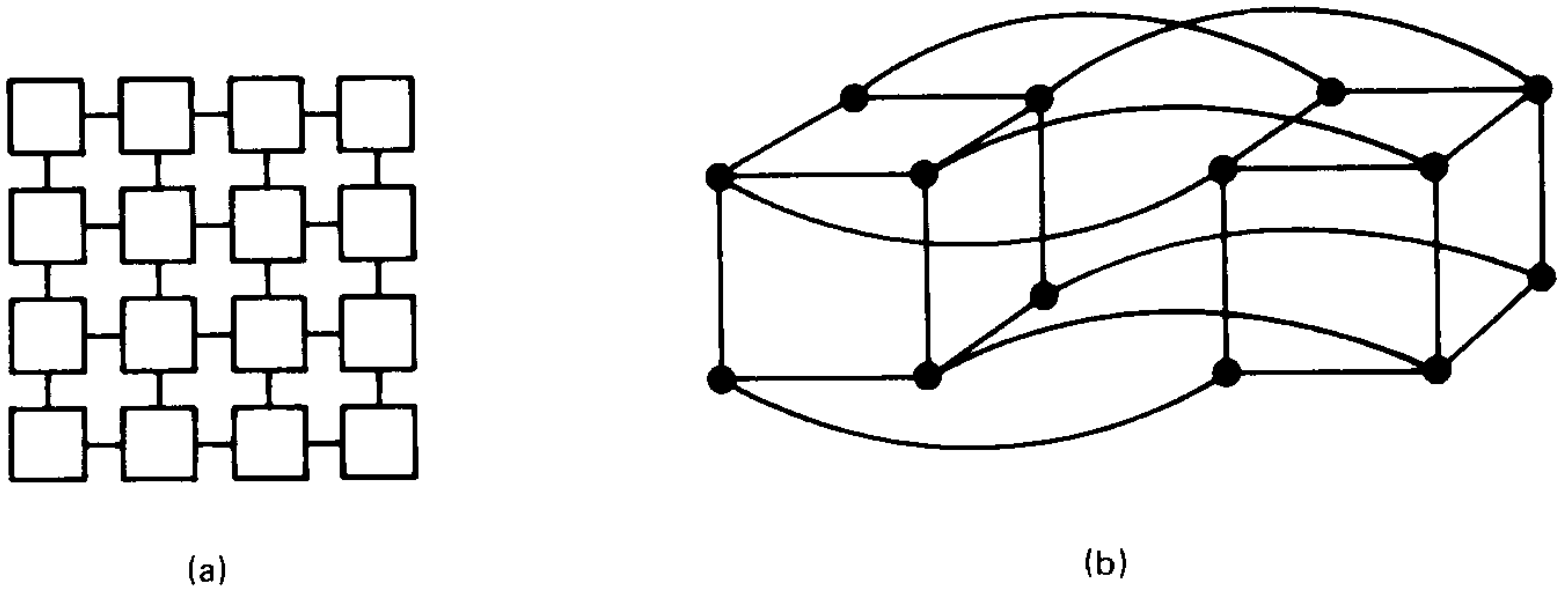
(a) A crossbar switch. (b) An omega switching network.

Bus based Multicomputers



- Limitation – slow as no. of computers increase

Switched Multicomputers



(a) Grid. (b) Hypercube.

Example 1

- A multicomputer with 256 CPUs is organized as a 16 × 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.

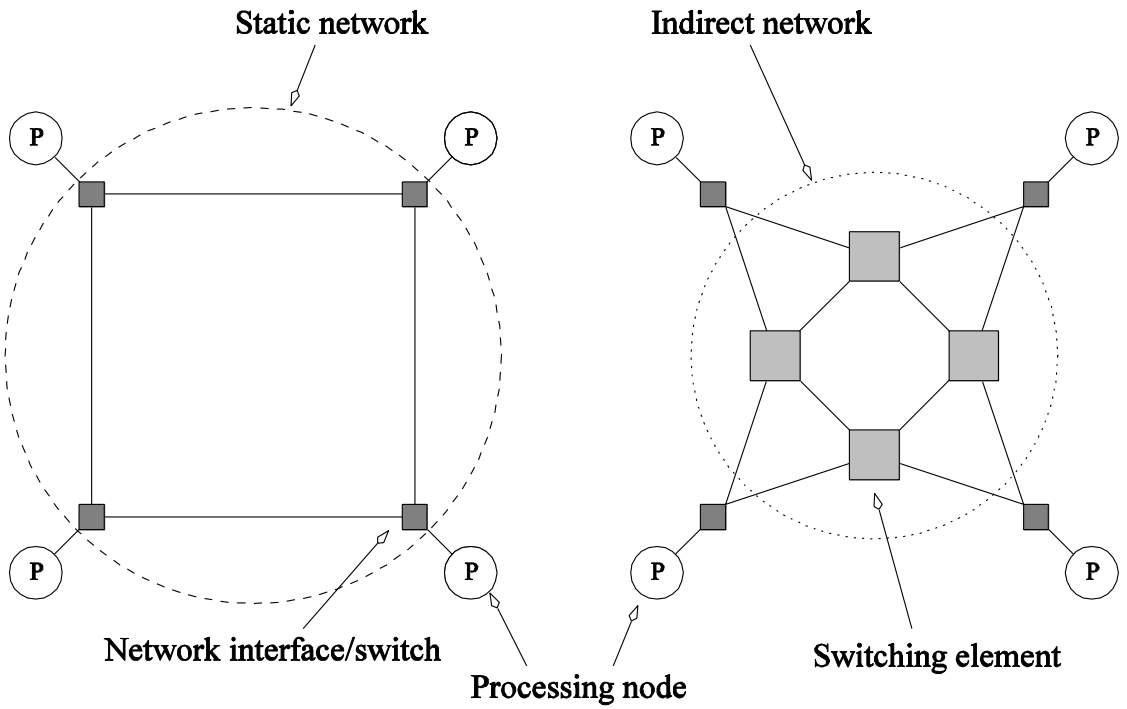
Example 2

- 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

-(Min # channels spanning two halves) * (BW of each channel)

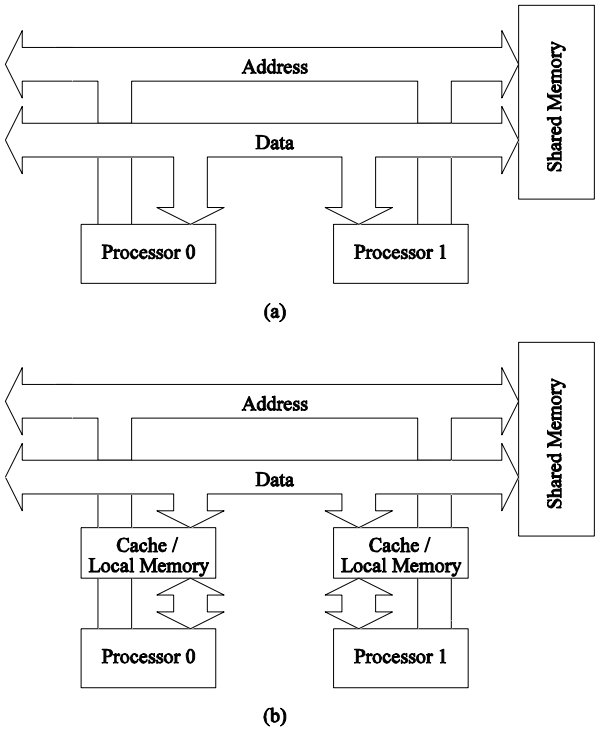
-Meaningful only for recursive topologies

-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

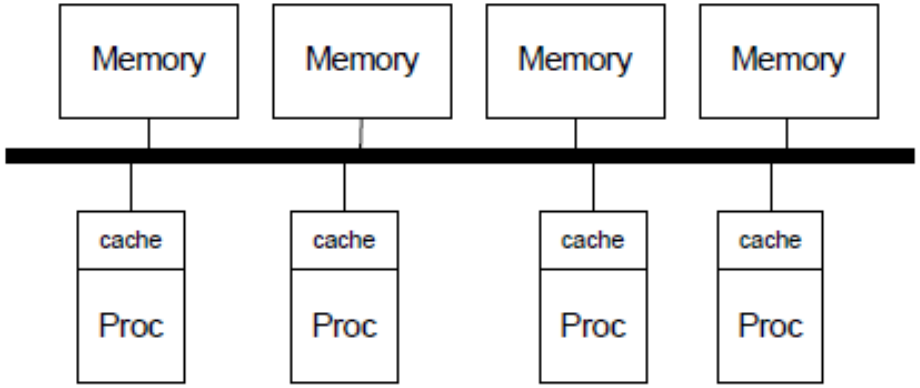


Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

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

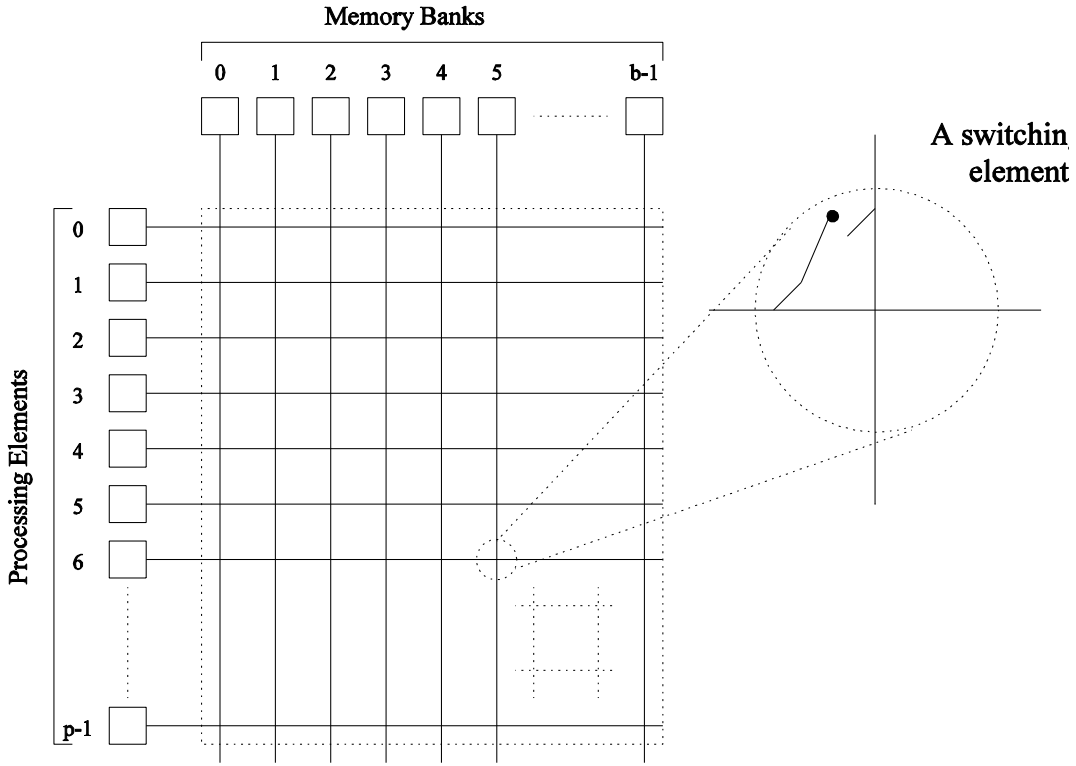
Bus

+ 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

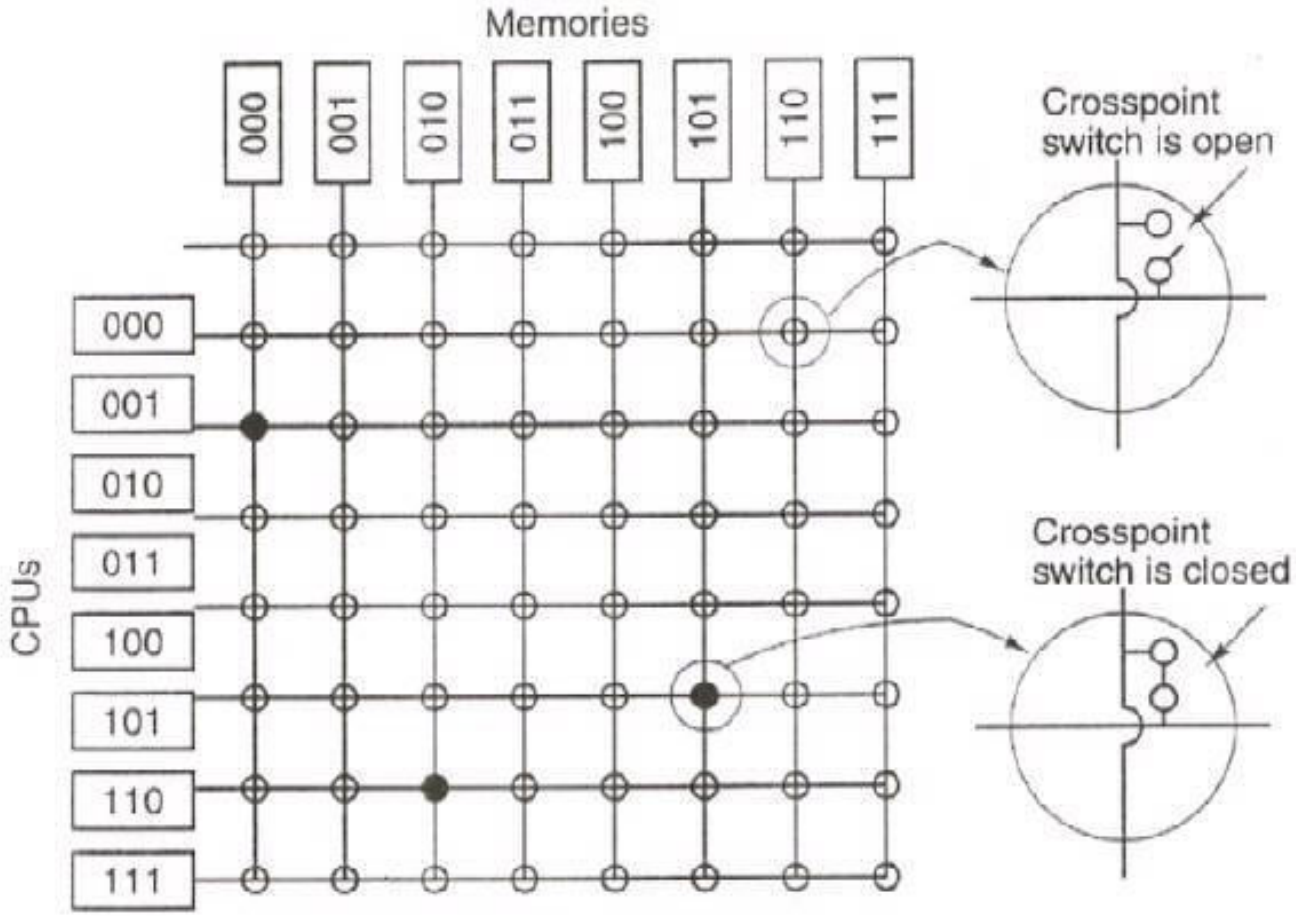


Crossbars

A crossbar network uses an *p*×*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.


Memories
000
001
010
011
100
101
110
111
CPUs
000
001
010
011
100
101
110
111
Crosspoint switch is open
Crosspoint switch is closed

Crossbar

- **Every node connected to all others** (non-blocking)
- **Good for small number of nodes**

+ Low latency and high throughput

- Expensive

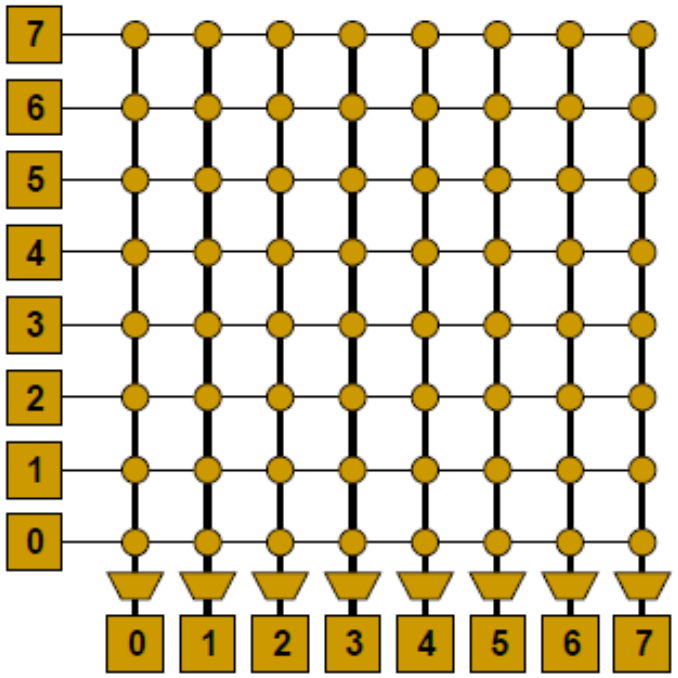
- Not scalable → O(N^2) cost

- Difficult to arbitrate

Core-to-cache-bank networks:

- IBM POWER5

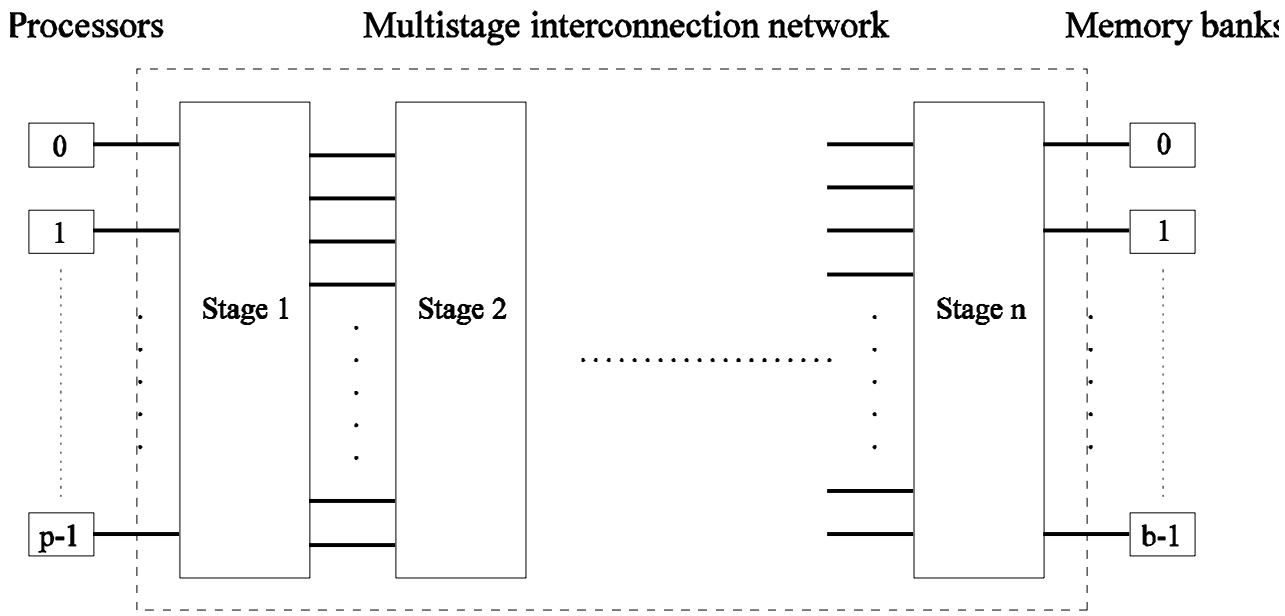
- Sun Niagara I/II



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.

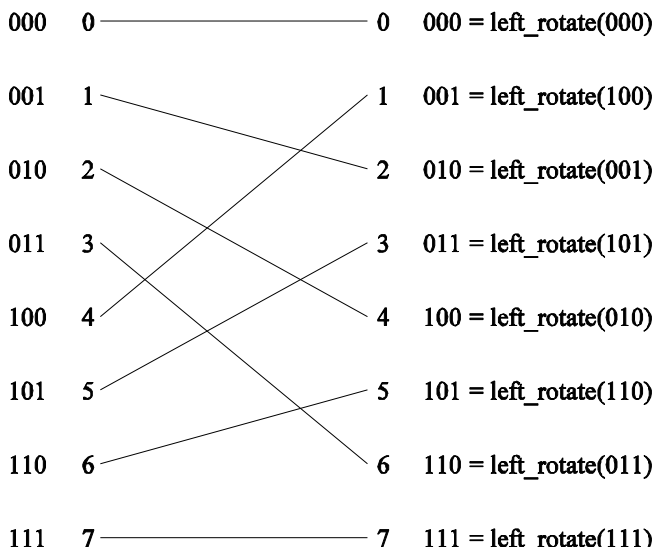
Multistage Networks

- 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.

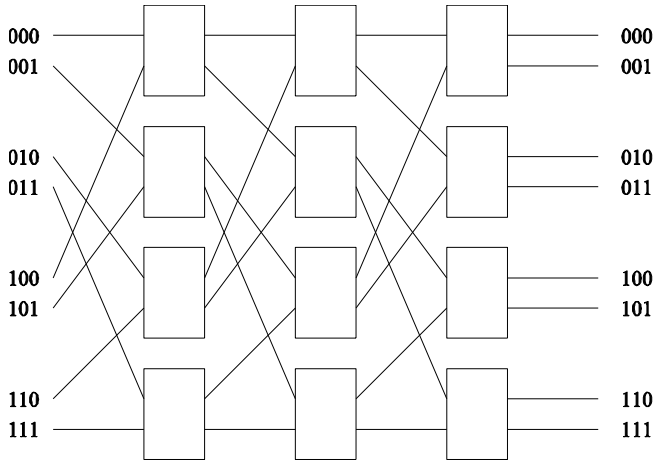
Multistage Omega Network

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

Two switching configurations of the 2 × 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:



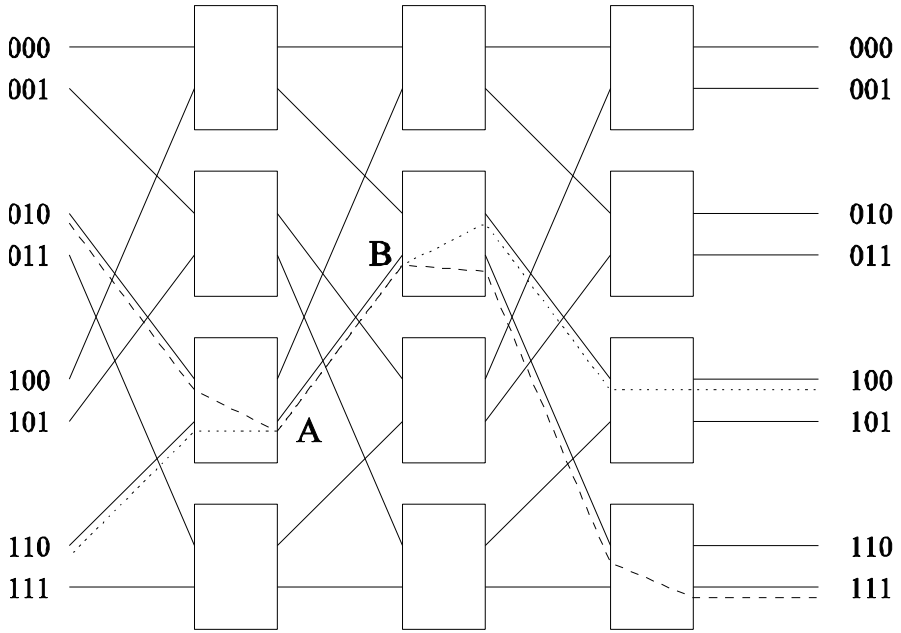
A complete omega network connecting eight inputs and eight outputs.

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

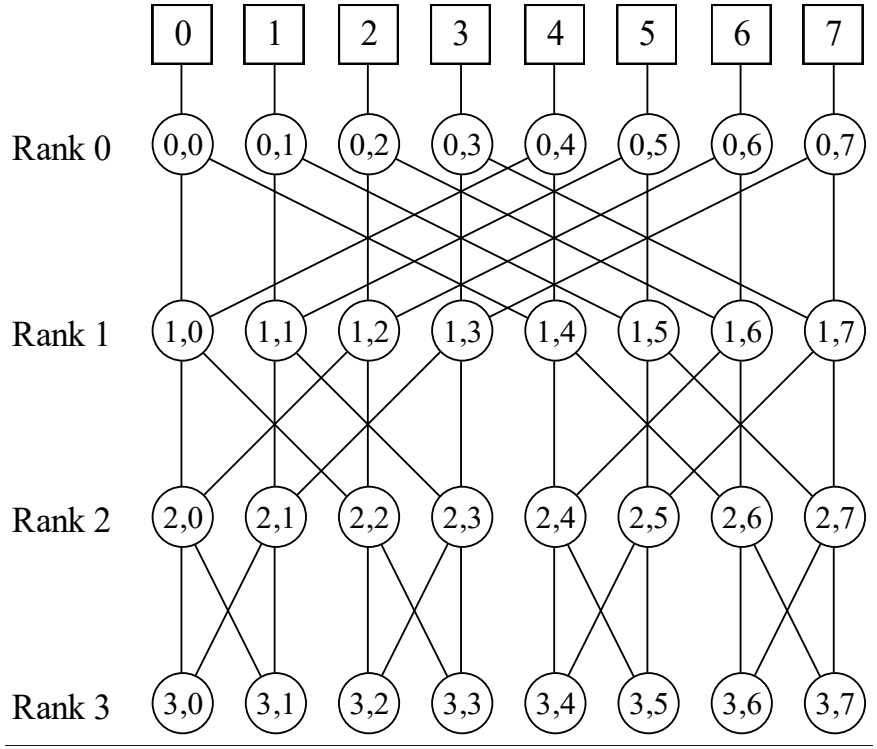
- A multiprocessor has 4096 50-MIPS CPUs connected to memory by an omega network. How fast do the switches have to be to allow a request to go to memory and back in one instruction time?
- Solution: 1 instruction= 20 nano sec

  1 switching stage=0.833 nano sec

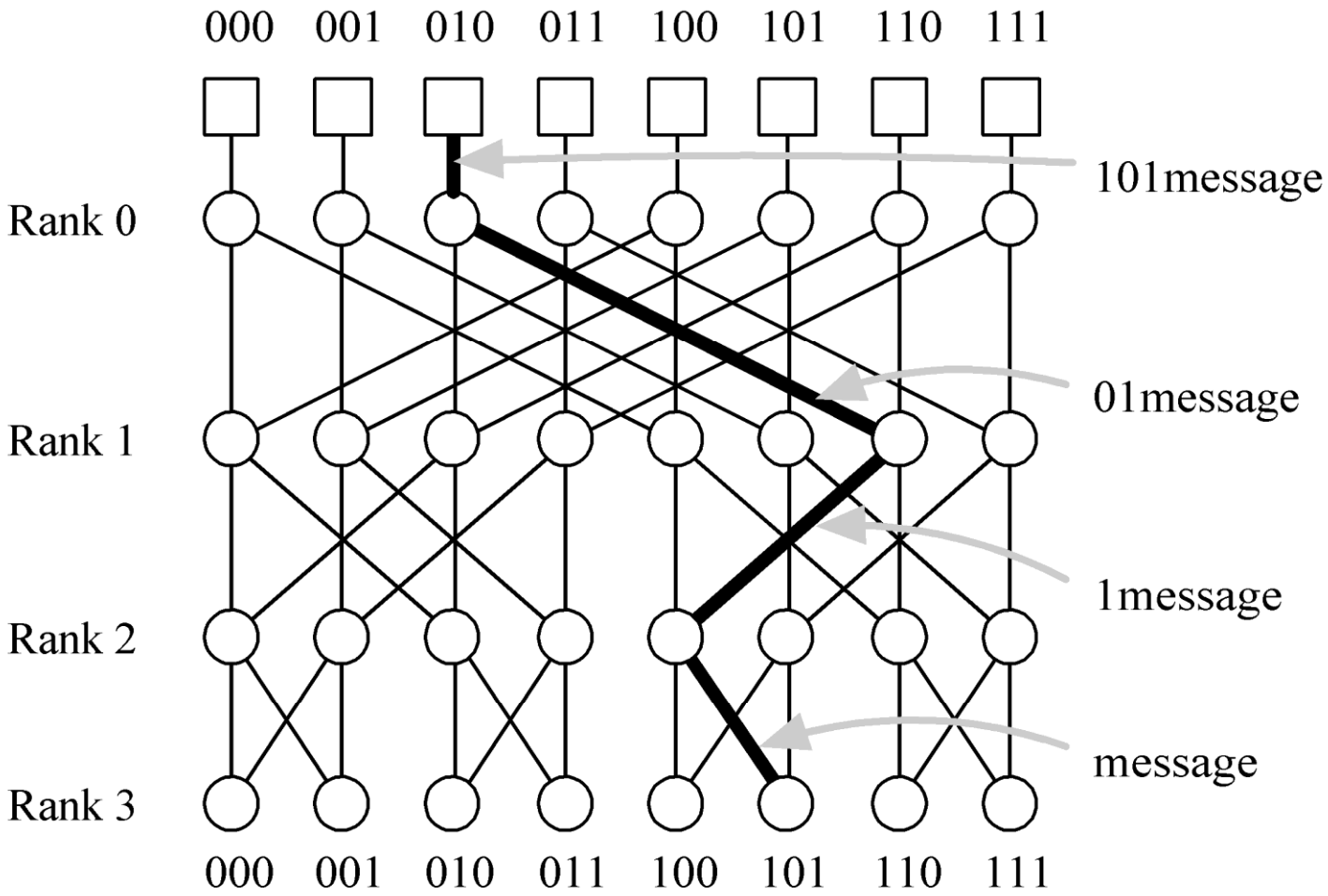
  2048x12 0.833 nano second switches

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^{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

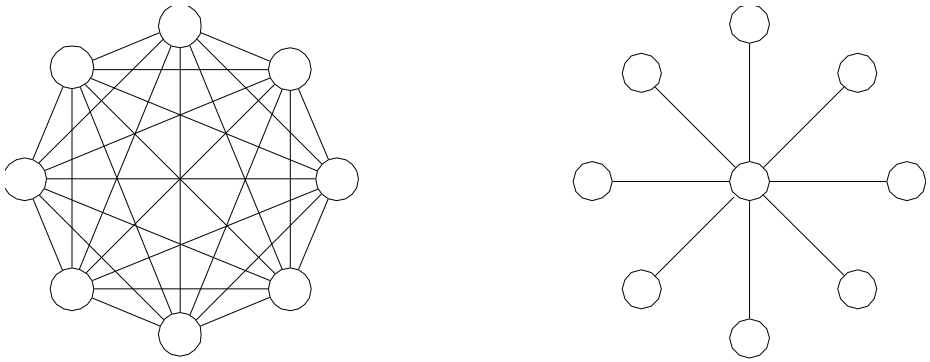
- Each processor is connected to every other processor.
- The number of links in the network scales as $O(p^2)$.
- While the performance scales very well, the hardware complexity is not realizable for large values of p.
- In this sense, these networks are static counterparts of crossbars.

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.

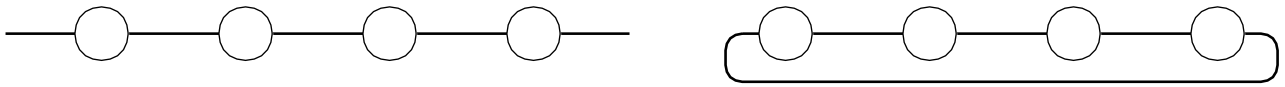


(a) A completely-connected network of eight nodes;
(b) a star connected network of nine nodes.

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



(a) (b)

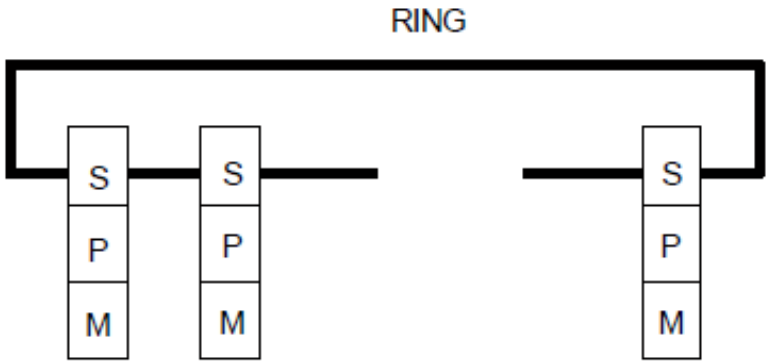
Linear arrays: (a) with no wraparound links; (b) with wraparound link.

Ring

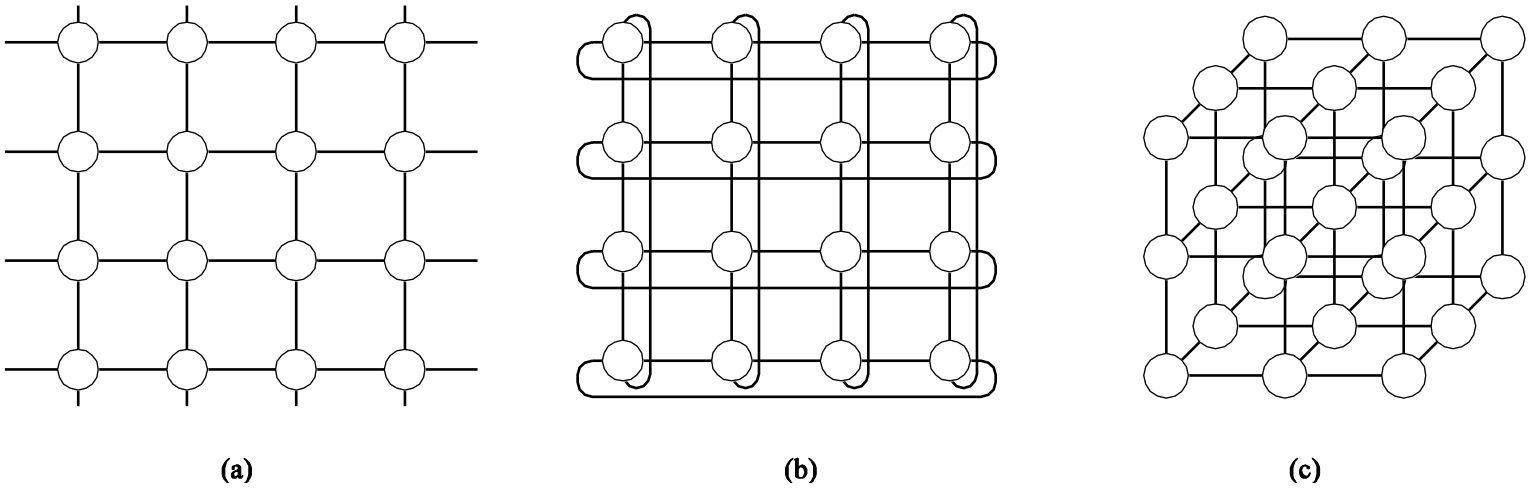
+ Cheap: O(N) cost
- High latency: O(N)
- Not easy to scale
  - Bisection bandwidth remains constant

Used in:

- Intel Larrabee/Core i7
- IBM Cell



Two- and Three Dimensional Meshes

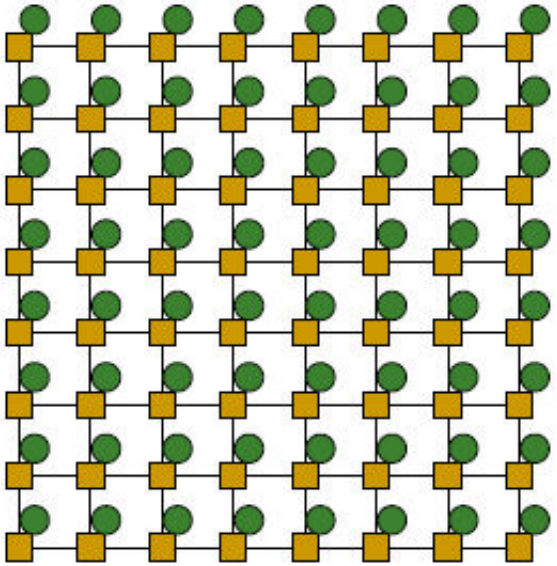


Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

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

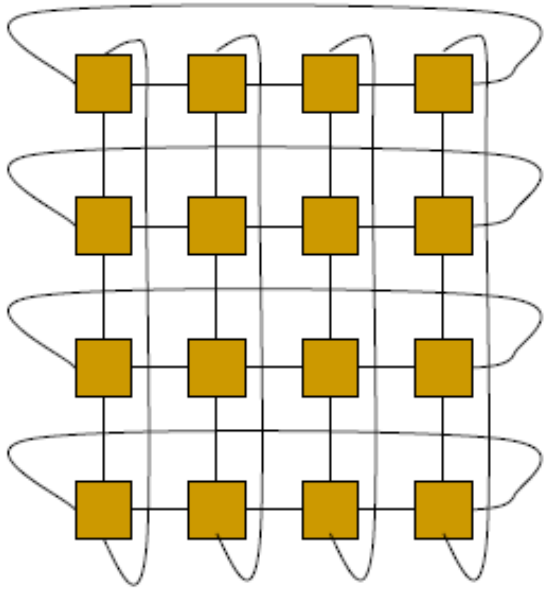
- Mesh is not symmetric on edges: performance very sensitive to placement of task on edge vs. middle
- Torus avoids this problem

+ Higher path diversity (& bisection bandwidth) than mesh

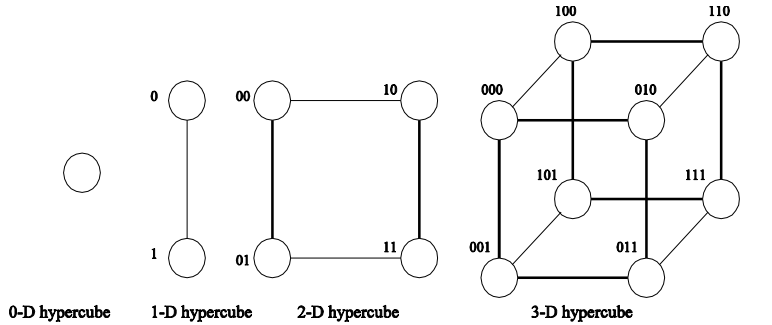
- Higher cost

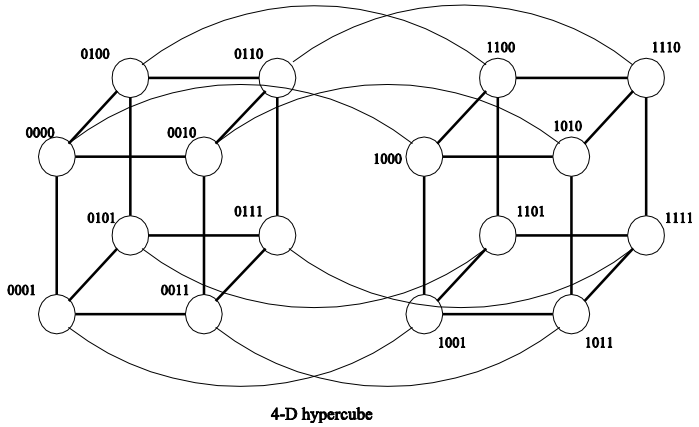
- Harder to lay out on-chip

  - Unequal link lengths



Hypercubes and their Construction





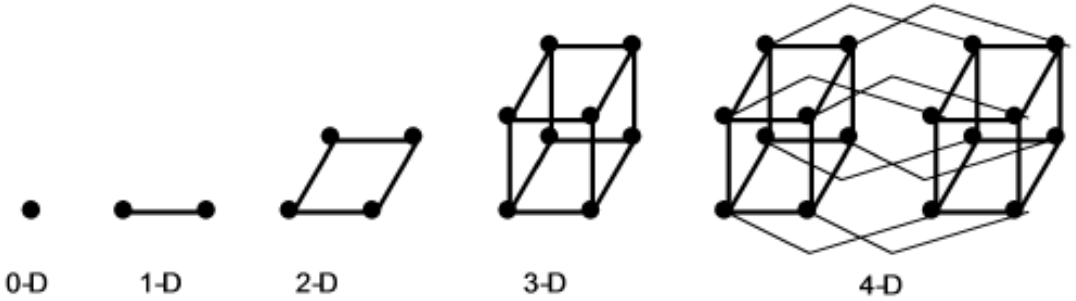
Construction of hypercubes from hypercubes of lower dimension.

Properties of Hypercubes

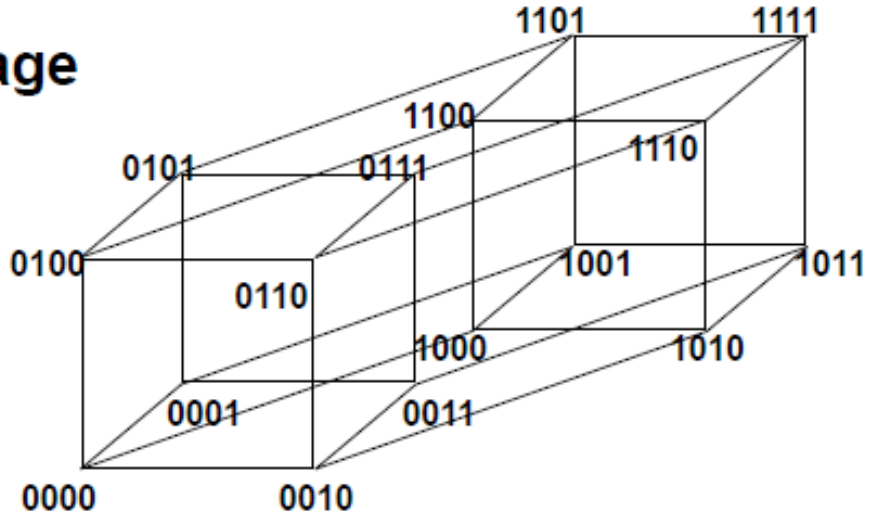
- The distance between any two nodes is at most *log p*.
- Each node has *log p* neighbors.
- The distance between two nodes is given by the number of bit positions at which the two nodes differ.

Hypercube

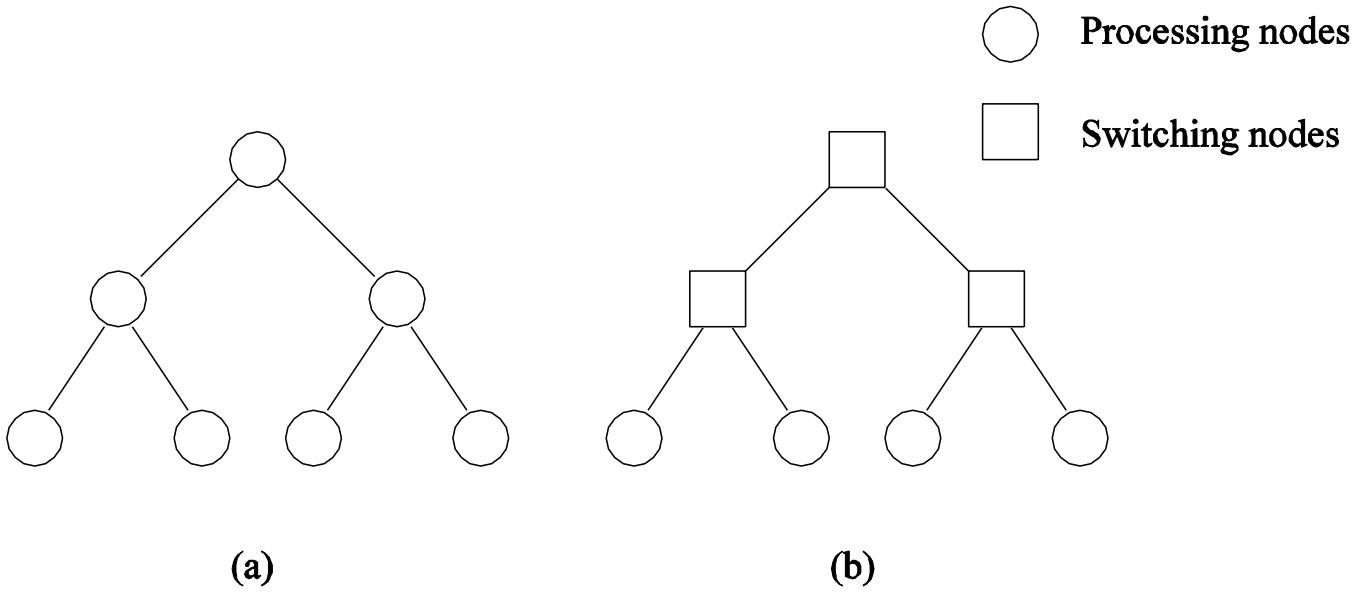
- Latency: O(logN)
- Radix: O(logN)
- #links: O(NlogN)



+ Low latency
- Hard to lay out in 2D/3D
- Used in some early message passing machines, e.g.:
  - Intel iPSC
  - nCube



Tree-Based Networks

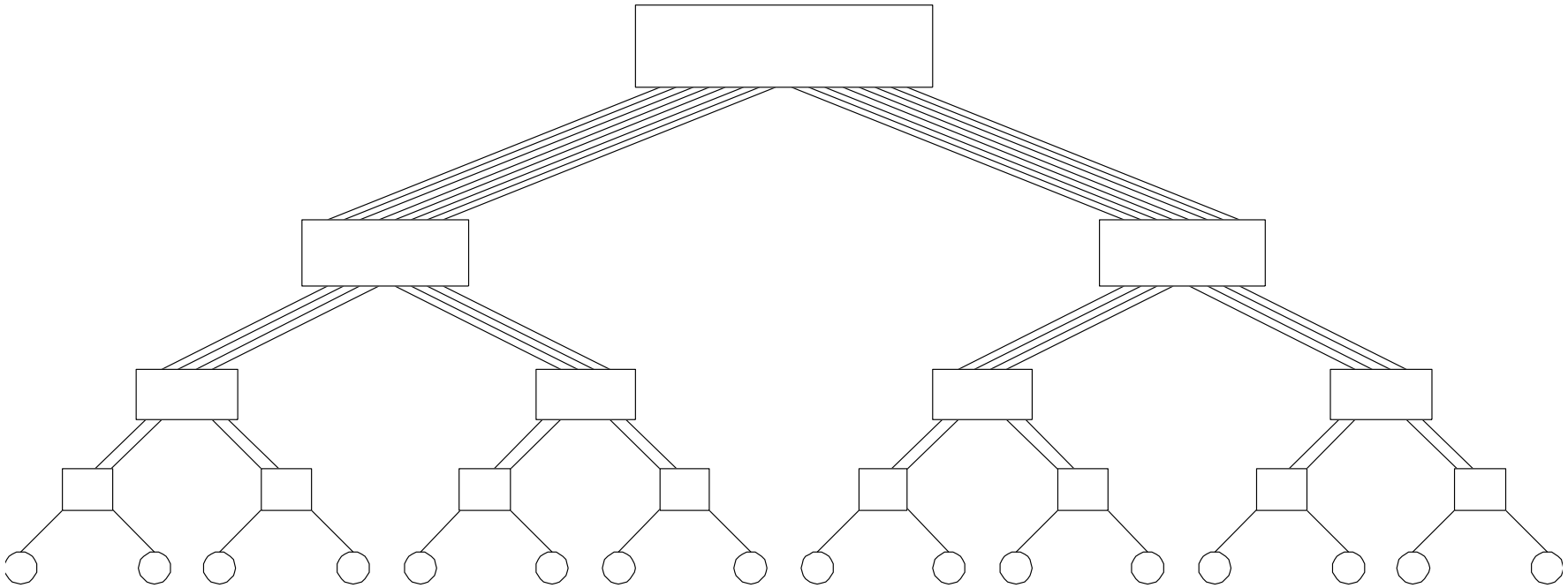


Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

Tree Properties

- The distance between any two nodes is no more than *2logp*.
- 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(logN)

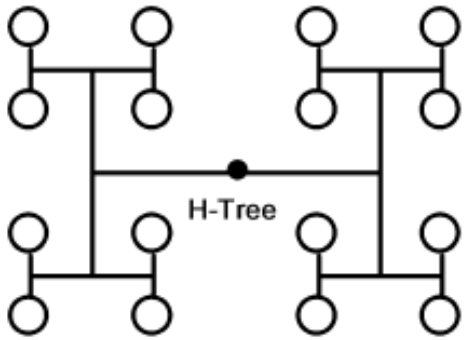
Good for local traffic

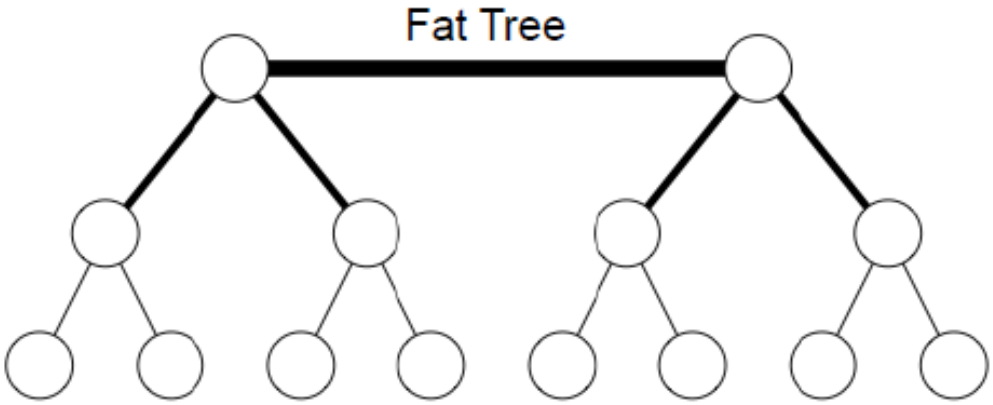
+ Cheap: O(N) cost

+ 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

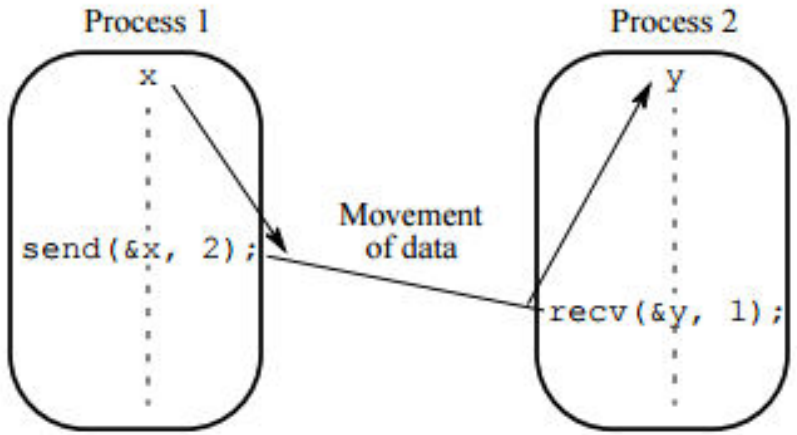
- Two basic communication primitives
  - Send() and Receive()
  - Example:

    send(&x, destination_id)

    receive(&y, source_id)
- Found in Client/Server computing models

Message passing

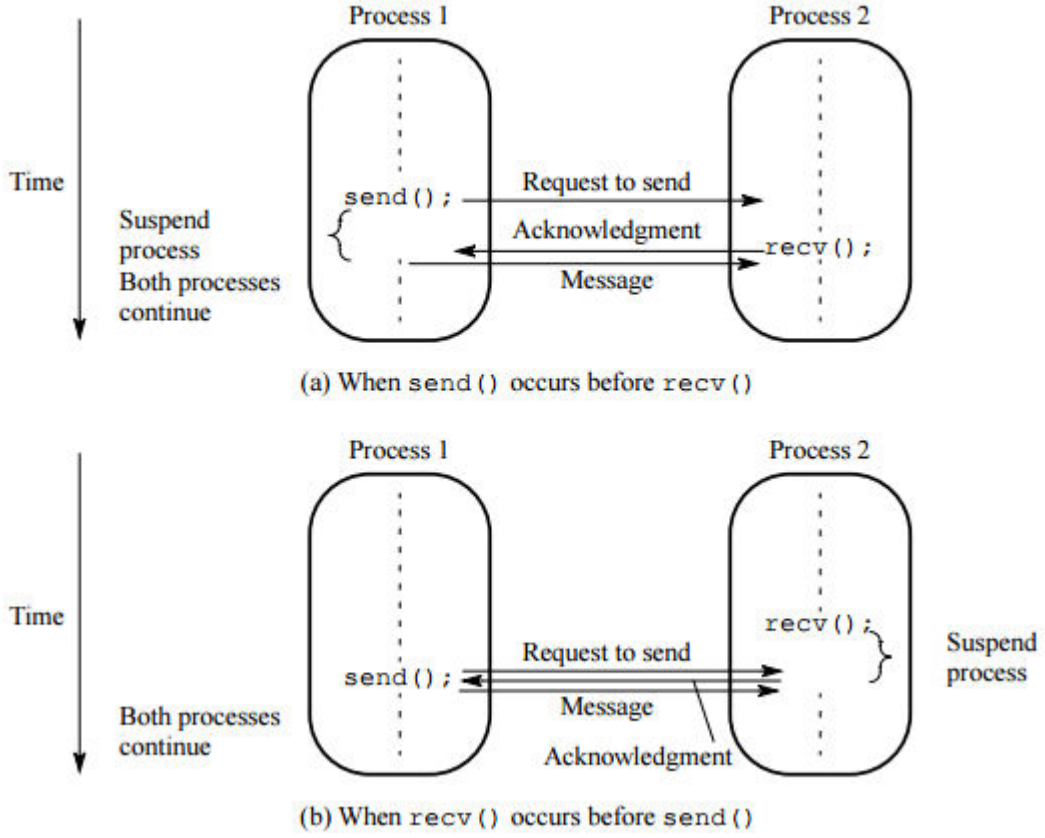
- Synchronous and asynchronous send routines

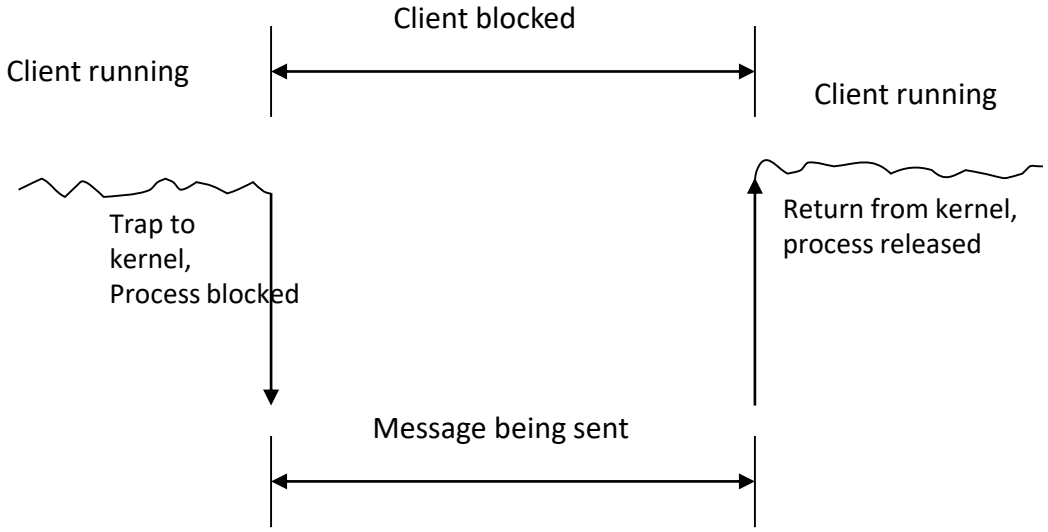


Passing a message between processes using `send()` and `recv()` library calls.

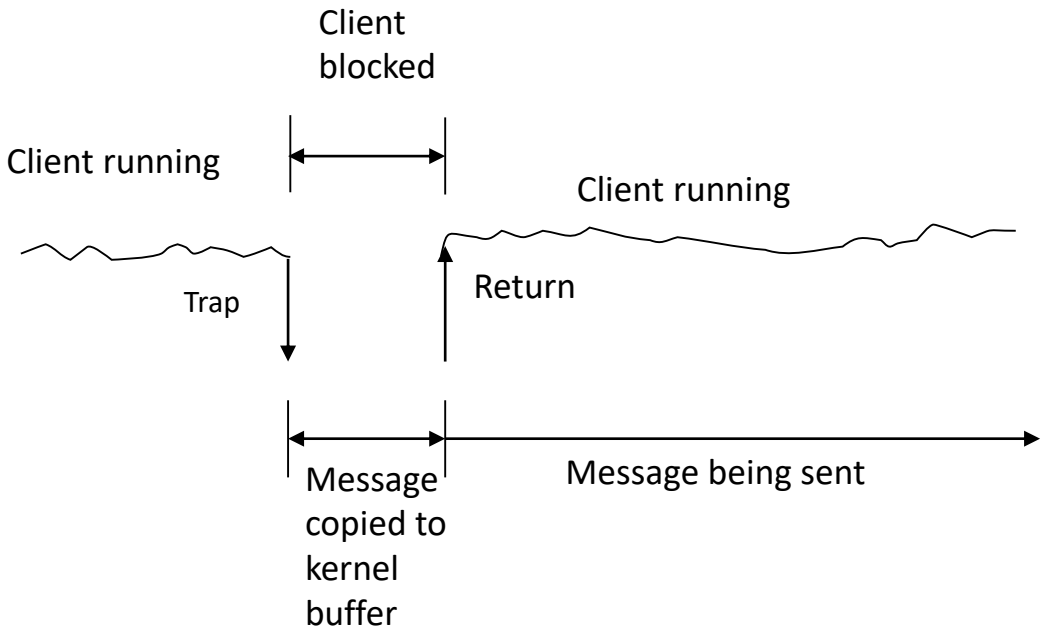
Message passing

- Blocking and non-blocking routines



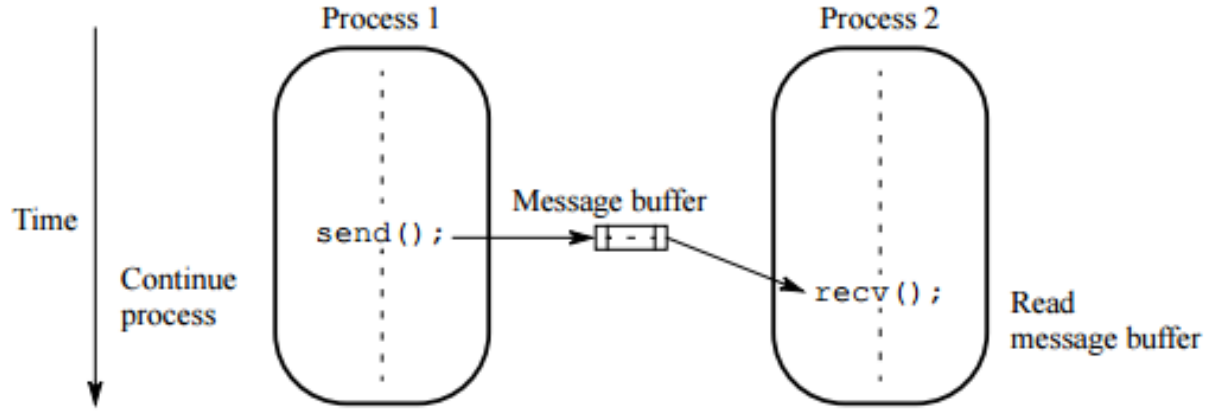


Blocking send primitive



Non-Blocking send primitive

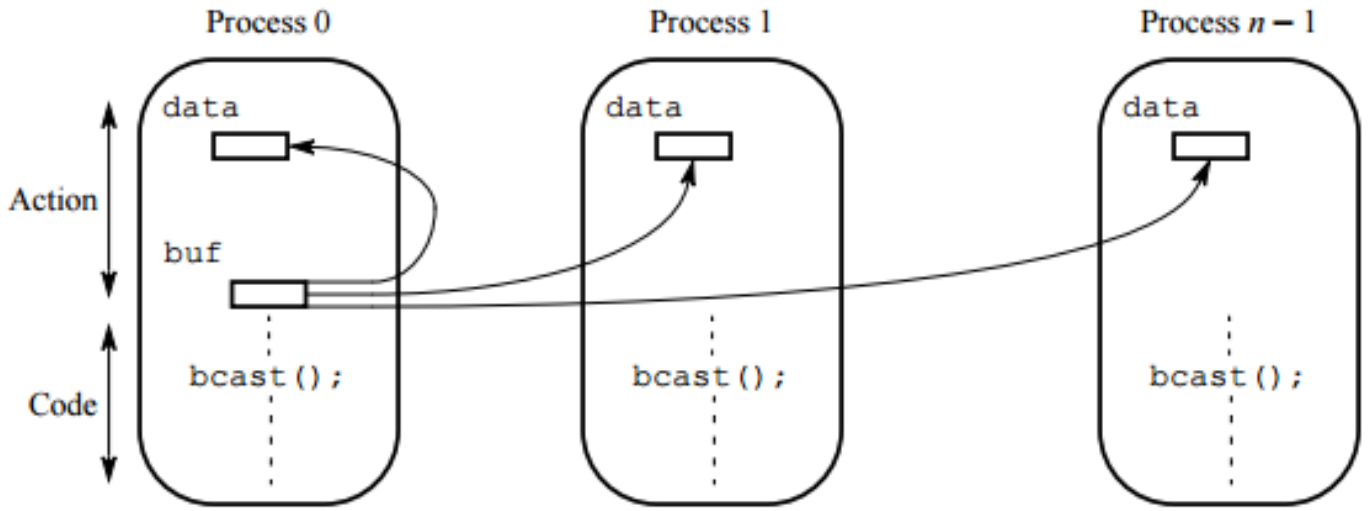
Using a message buffer



Buffered versus Unbuffered Primitives

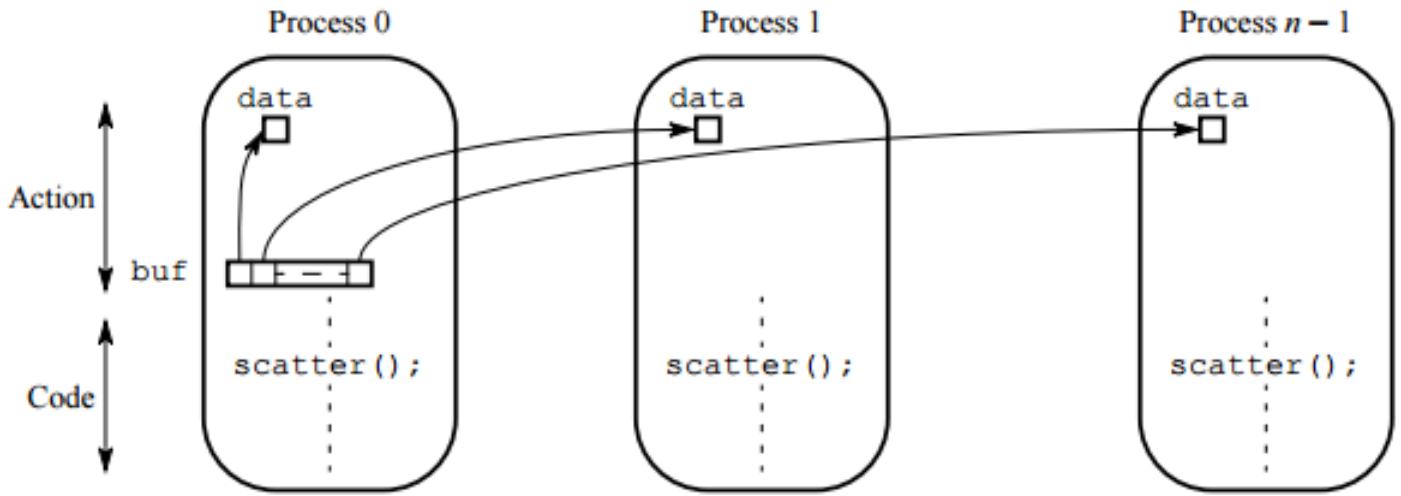
- No buffer allocated. Fine if receive() is called before send().
- Buffers allocated, freed, and managed to store the incoming message. Usually a mailbox created.

Group communication routines



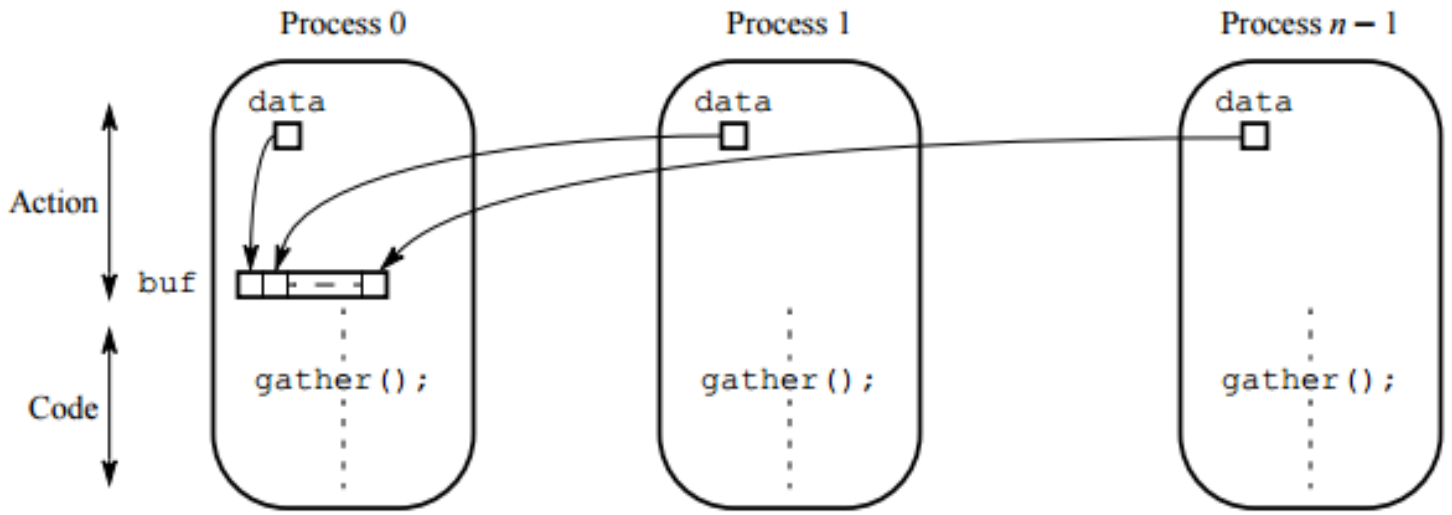
Broadcast operation.

Group communication routines



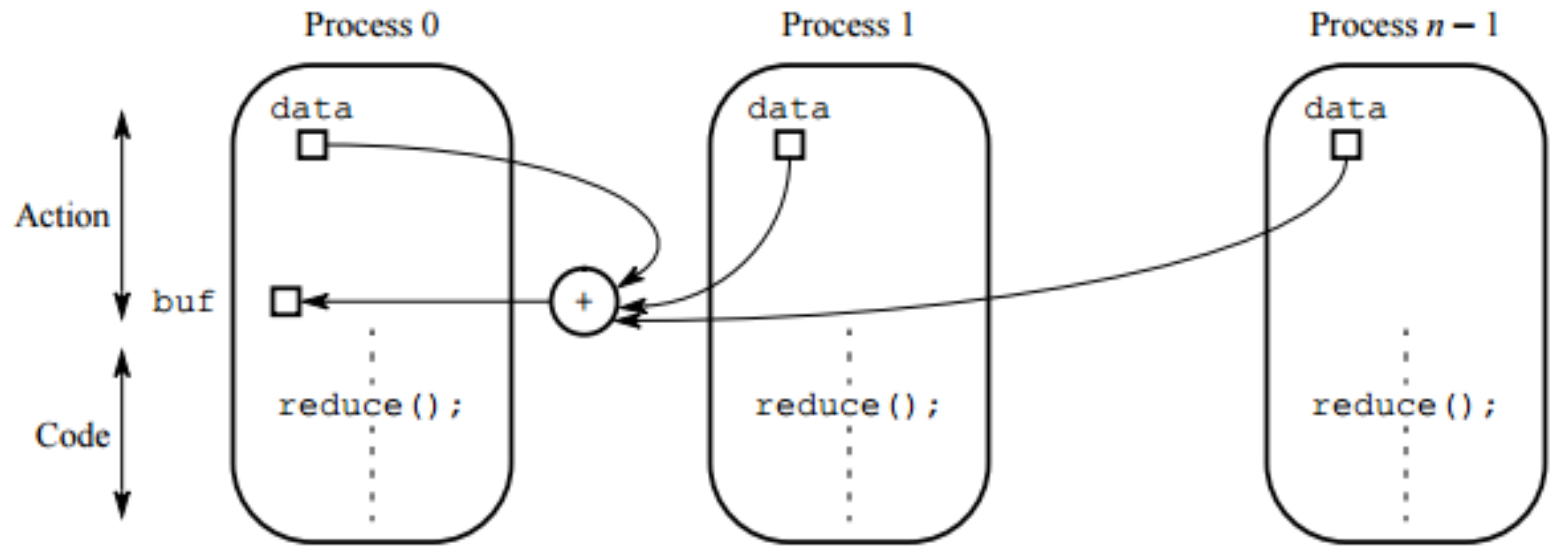
Scatter operation.

Group communication routines



Gather operation.

Group communication routines



Reduce operation (addition).

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

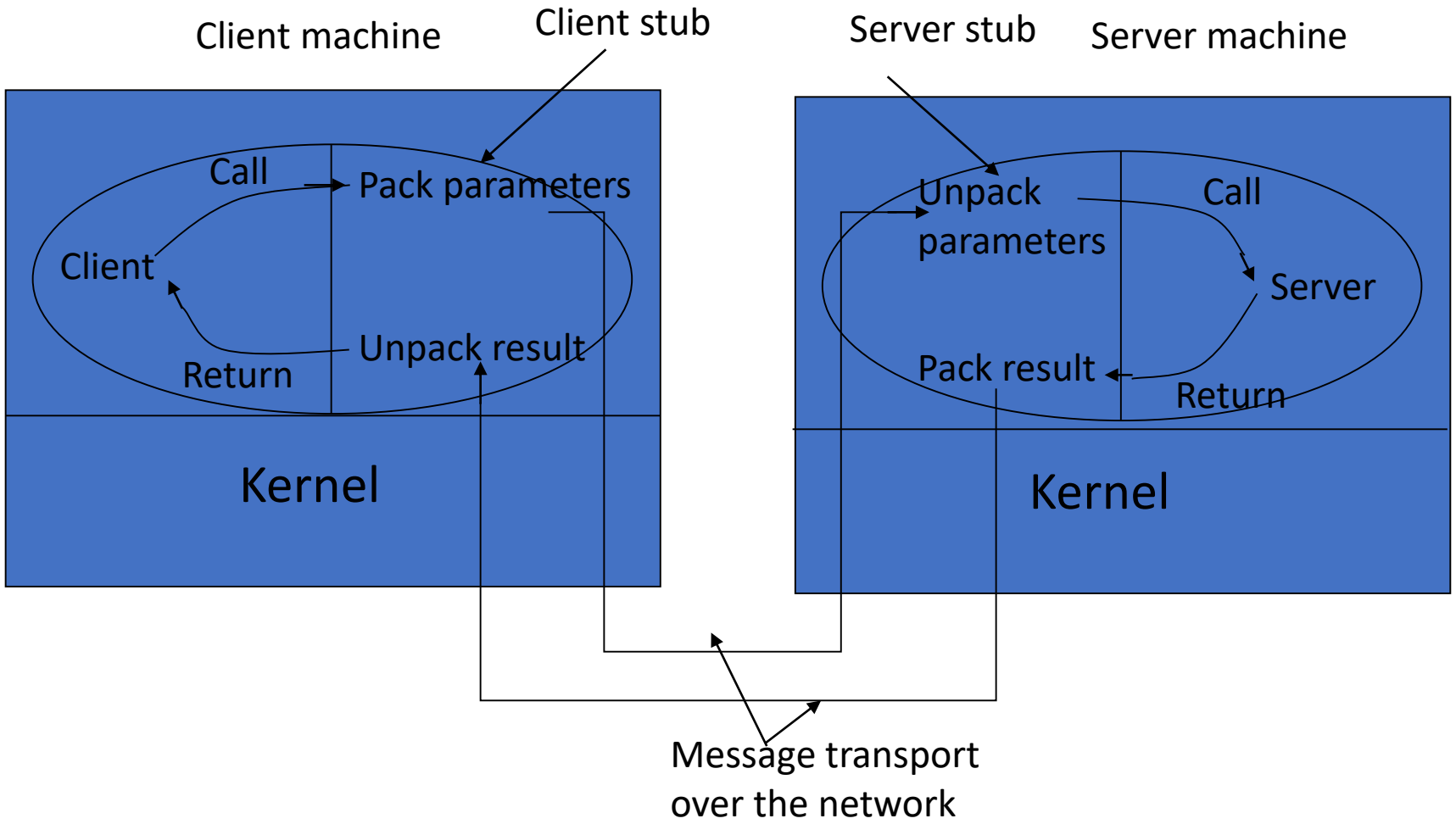
```
/* Add my portion Of data */
   x = n/nproc;
   low = myid * x;
   high = low + x;
   for(i = low; i < high; i++)
      myresult += data[i];
   printf("I got %d from %d\n", myresult, myid);

/* 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.

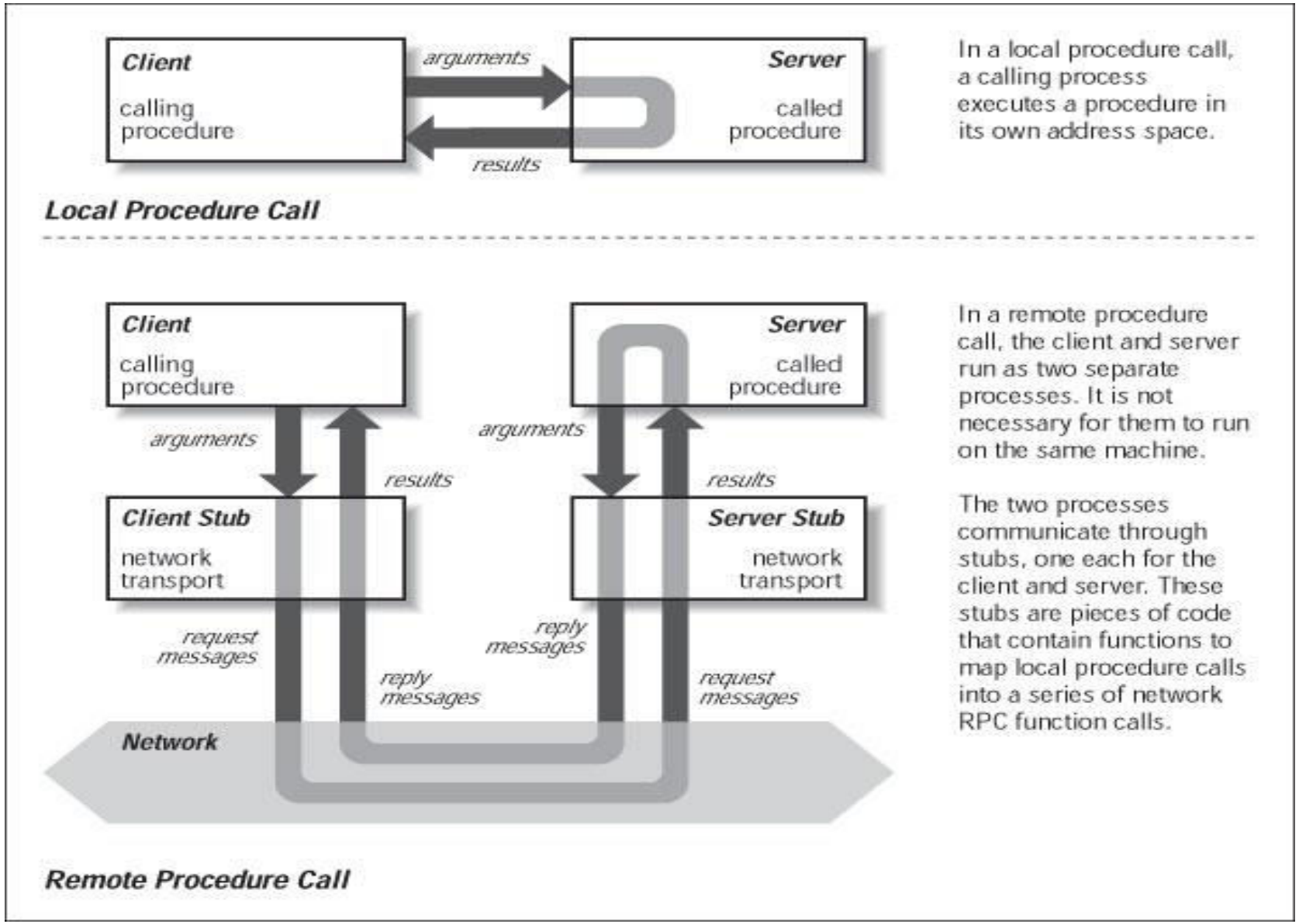
Remote Procedure Call



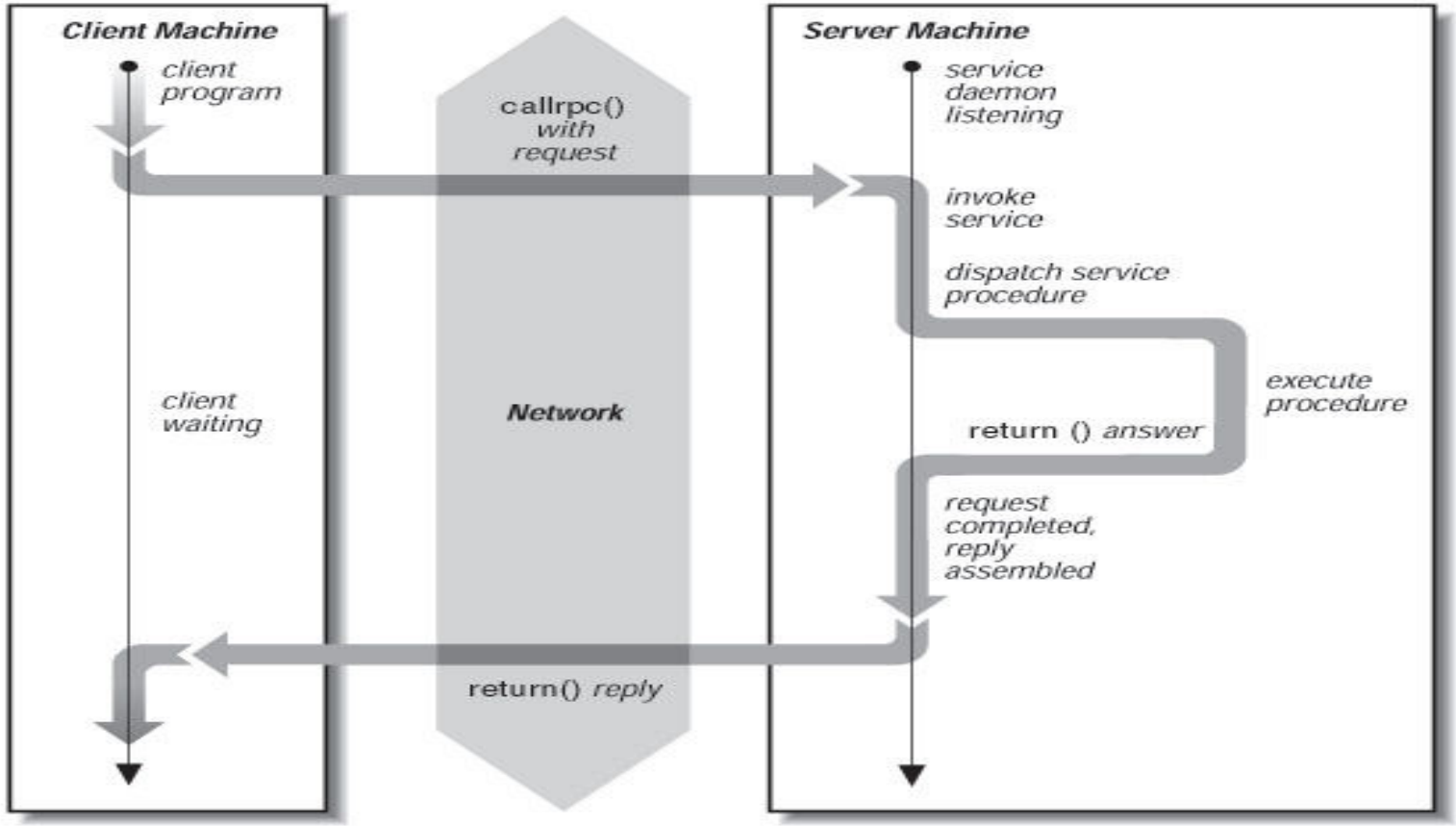
Remote Procedure Call

- 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



RPC Communication



The client sends out a request over the network. The service daemon is constantly listening for requests. When a request is received, it invokes the service. The appropriate procedure is dispatched. The request is executed and the reply is returned over the network to the client.

The client machine is inactive between the time of the request and when it receives a reply.

The client and server machines may be the same.

RPC System Components

- **Message module**
  - IPC module of Send/Receive/Reply
  - responsible for exchanging messages

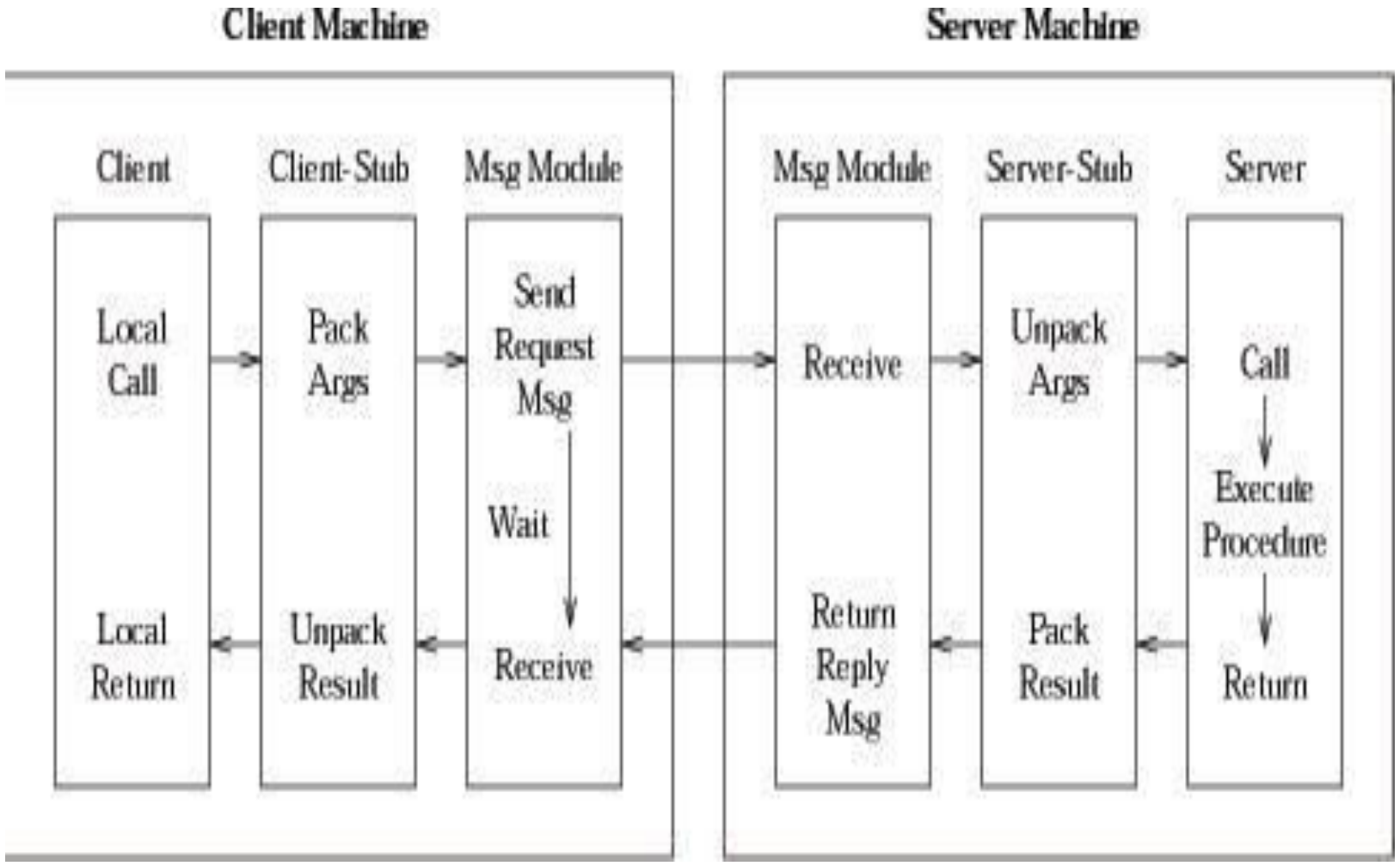
- **Stub procedures** (client and server stubs)
  - a stub is a communications interface that implements the RPC protocol and specifies how messages are constructed and exchanged
  - responsible for packing and unpacking of arguments and results (this is also referred to as “marshaling”)
  - these procedures 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.

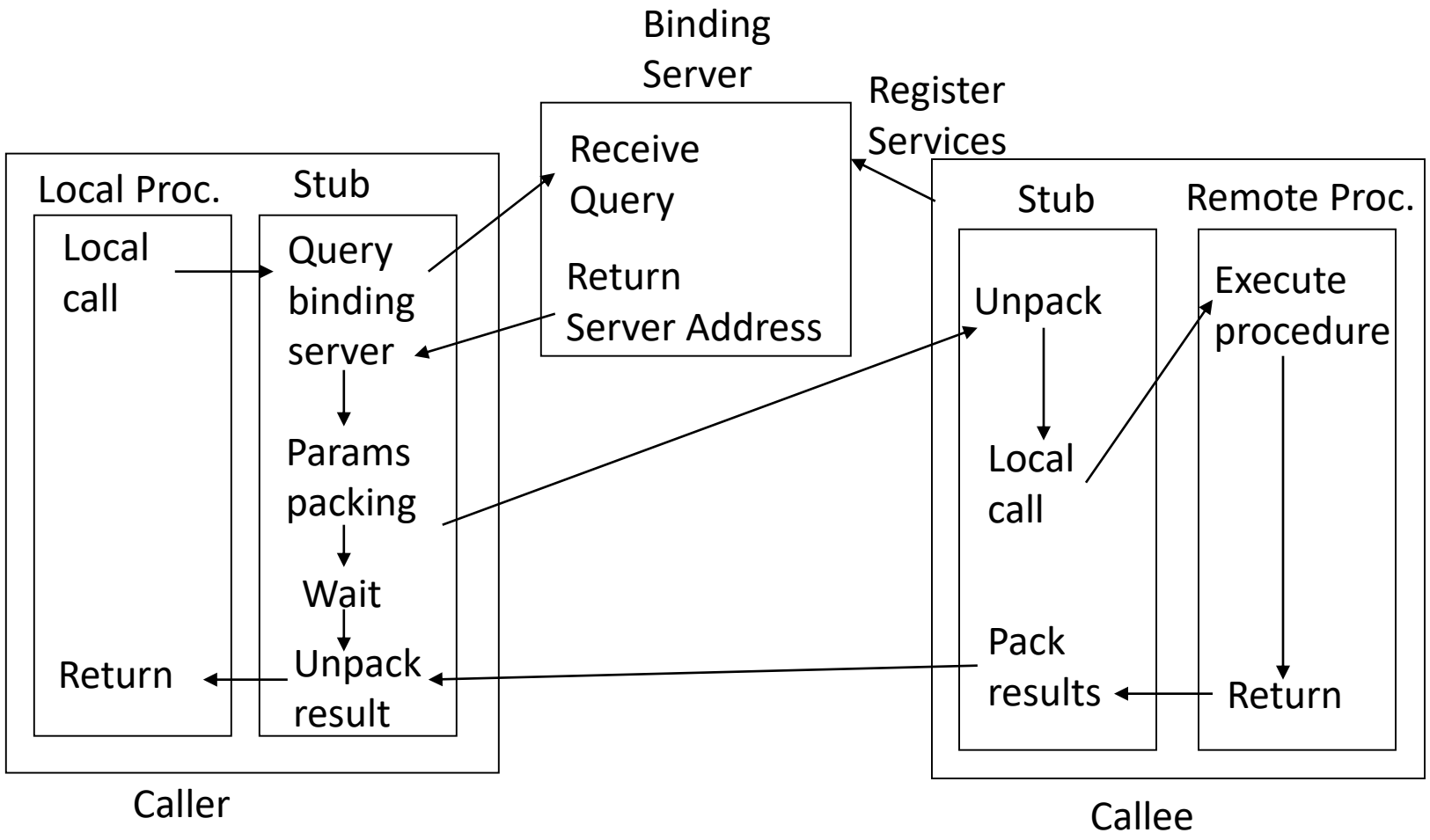
  Disadvantage: inflexible.

- 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 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, ..., p_i, ..., 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 xth 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, ..., e_i^x, e_i^{x+1}, ... 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

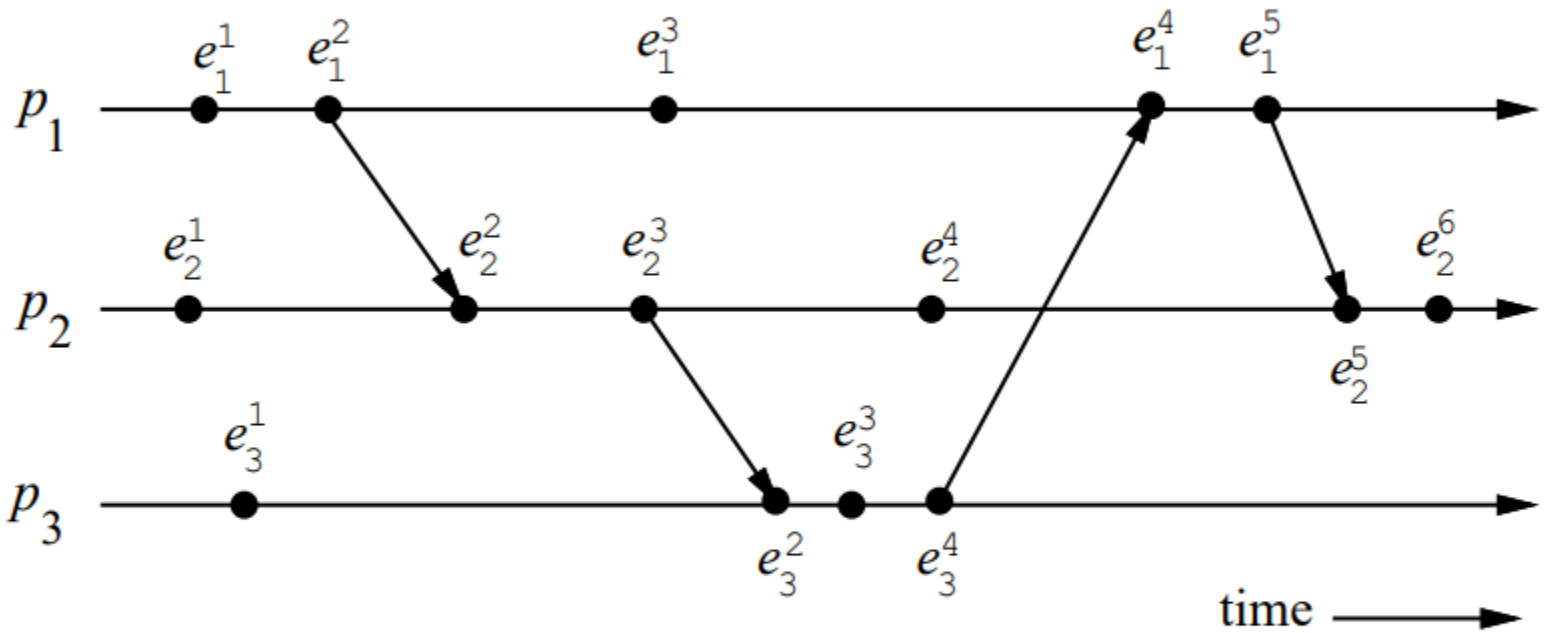


Figure 2.1: The space-time diagram of a distributed execution.

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.

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

- 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 dependent 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, $e_i \not\rightarrow e_j \not\Rightarrow e_j \not\rightarrow e_i$.
- 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 occured 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 $send(m_{ij}) \longrightarrow send(m_{kj})$, then $rec(m_{ij}) \longrightarrow rec(m_{kj})$.

- 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 or 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:

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

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

$$\forall m_{ij}: \ send(m_{ij}) \not\leq LS_i^{x_i} \ \Leftrightarrow \ m_{ij} \notin SC_{ij}^{x_i,y_j} \bigwedge rec(m_{ij}) \not\leq LS_j^{y_j}$$

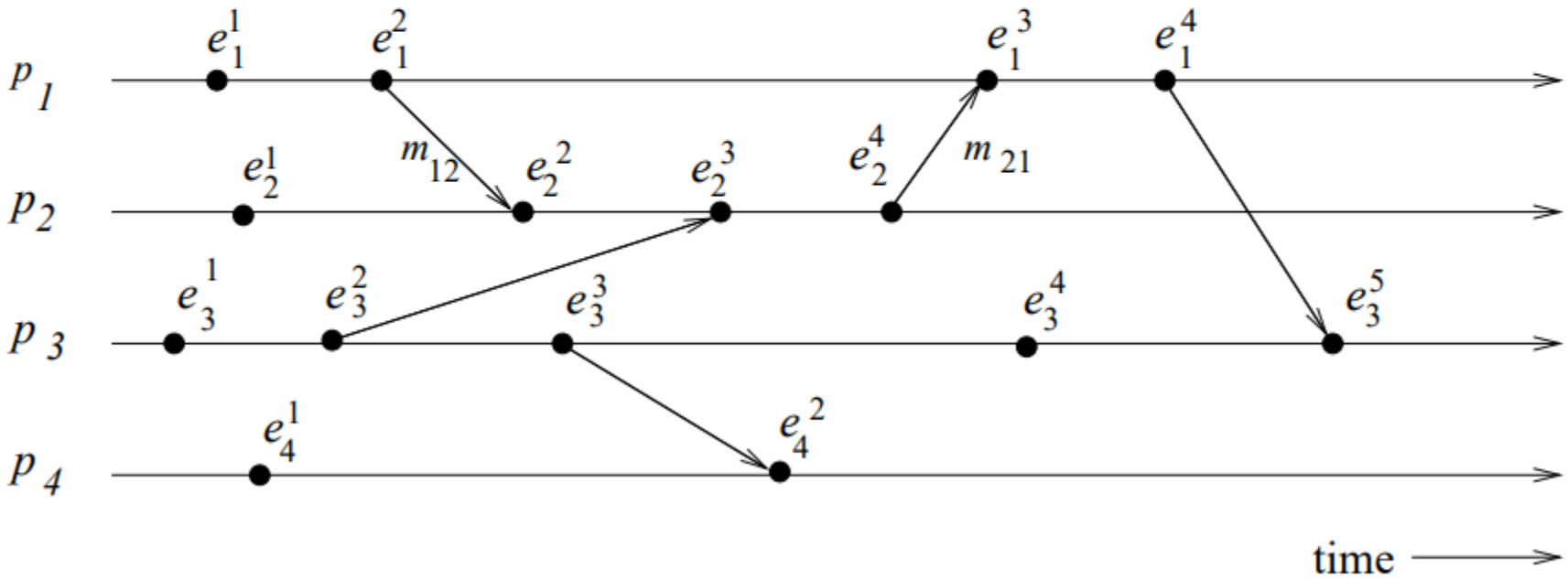
- That is, channel state $SC_{ij}^{y_i,z_k}$ and process state $LS_j^{z_k}$ must not include any message that process p_i sent after executing event $e_i^{x_i}$.

. . . 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}.

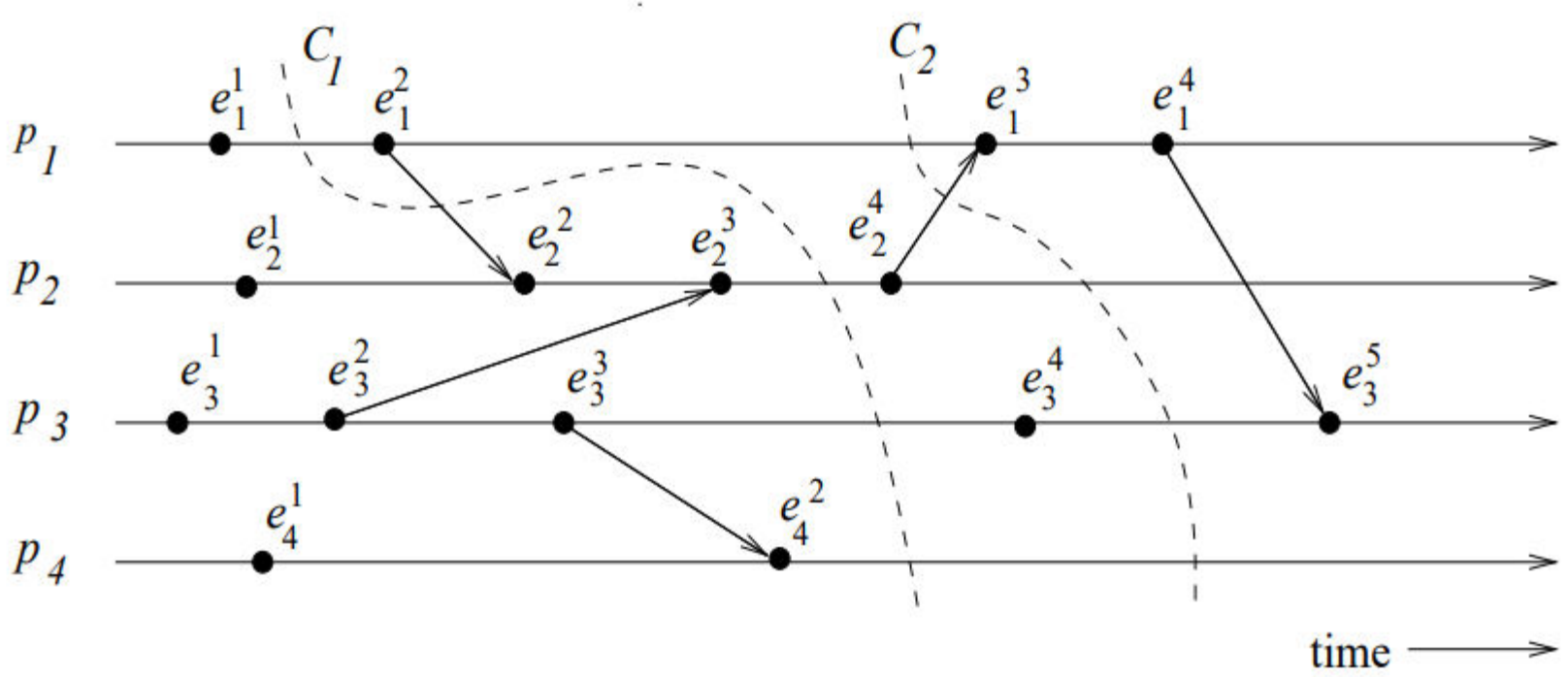
Cuts of a Distributed Computation

"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

Introduction

- 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 $\mathrm{C}(e)$ and called the timestamp of e, and is defined as follows:

$$C : H \mapsto T$$

such that the following property is satisfied:

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

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,

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

  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.

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.

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:

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

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:

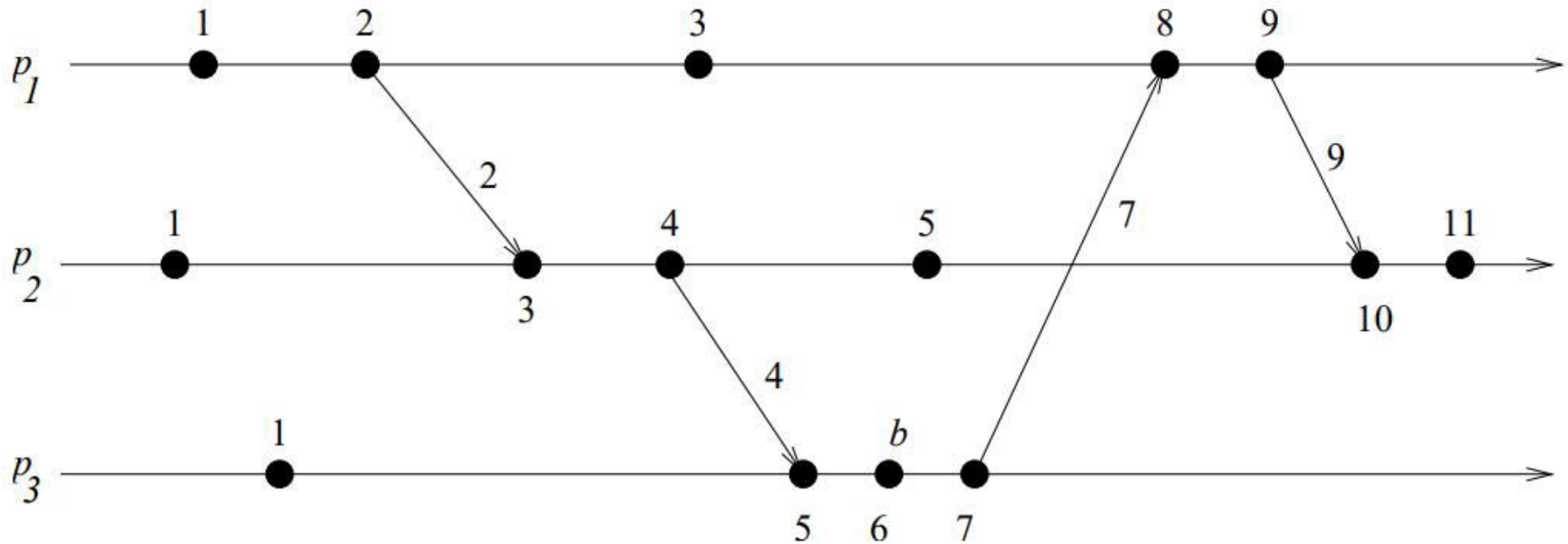


Figure 3.1: The space-time diagram of a distributed execution.

Basic Properties

Consistency Property

- Scalar clocks satisfy the monotonicity and hence the consistency property:

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

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.

Total Ordering

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:

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

Properties...

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.

Properties...

No Strong Consistency

- The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j, $C(e_i) < C(e_j) \not\Longrightarrow e_i \rightarrow e_j$.
- 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.

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:

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

- *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 $[0, 0, 0, ..., 0]$.

Vector Time

An Example of Vector Clocks

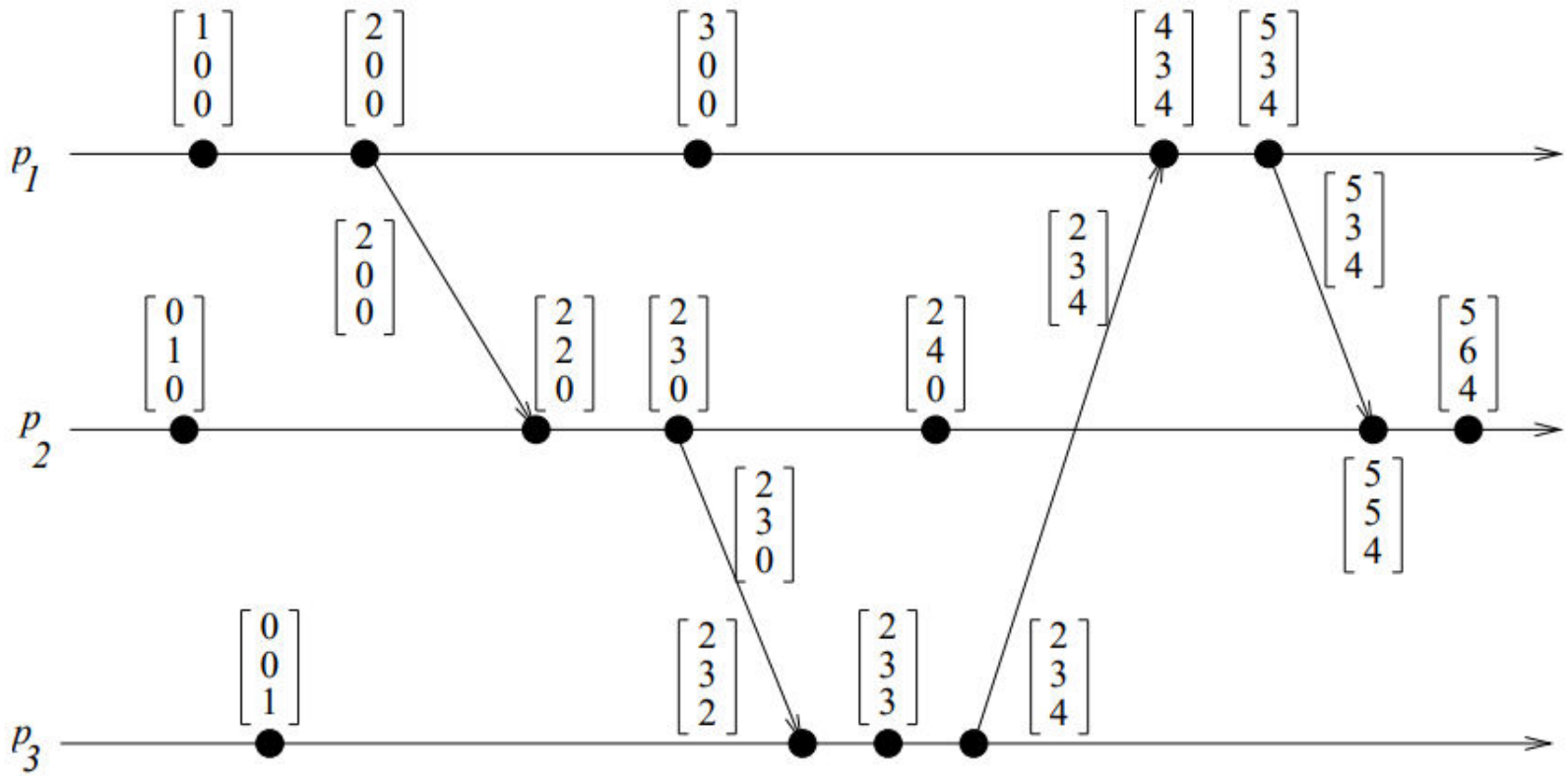


Figure 3.2: Evolution of vector time.

Vector Time

Comparing Vector Timestamps

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

$$
\begin{aligned}
vh = vk &\Leftrightarrow \forall x : vh[x] = vk[x] \\
vh \leq vk &\Leftrightarrow \forall x : vh[x] \leq vk[x] \\
vh < vk &\Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x] \\
vh \parallel vk &\Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh)
\end{aligned}
$$

- 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

$$
\begin{aligned}
x \rightarrow y &\Leftrightarrow vh[i] \leq vk[i] \\
x \parallel y &\Leftrightarrow vh[i] > vk[i] \wedge vh[j] < vk[j]
\end{aligned}
$$

Vector Time

Properties of Vectot 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

$$x \rightarrow y \quad \Leftrightarrow \quad vh < vk$$
$$x \parallel y \quad \Leftrightarrow \quad vh \parallel vk.$$

- 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 $i_1, i_2, \ldots, i_{n_1}$ of the vector clock at p_i have changed to $v_1, v_2, \ldots, v_{n_1}$, respectively, since the last message sent to p_j, then process p_i piggybacks a compressed timestamp of the form:

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

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:

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

- 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

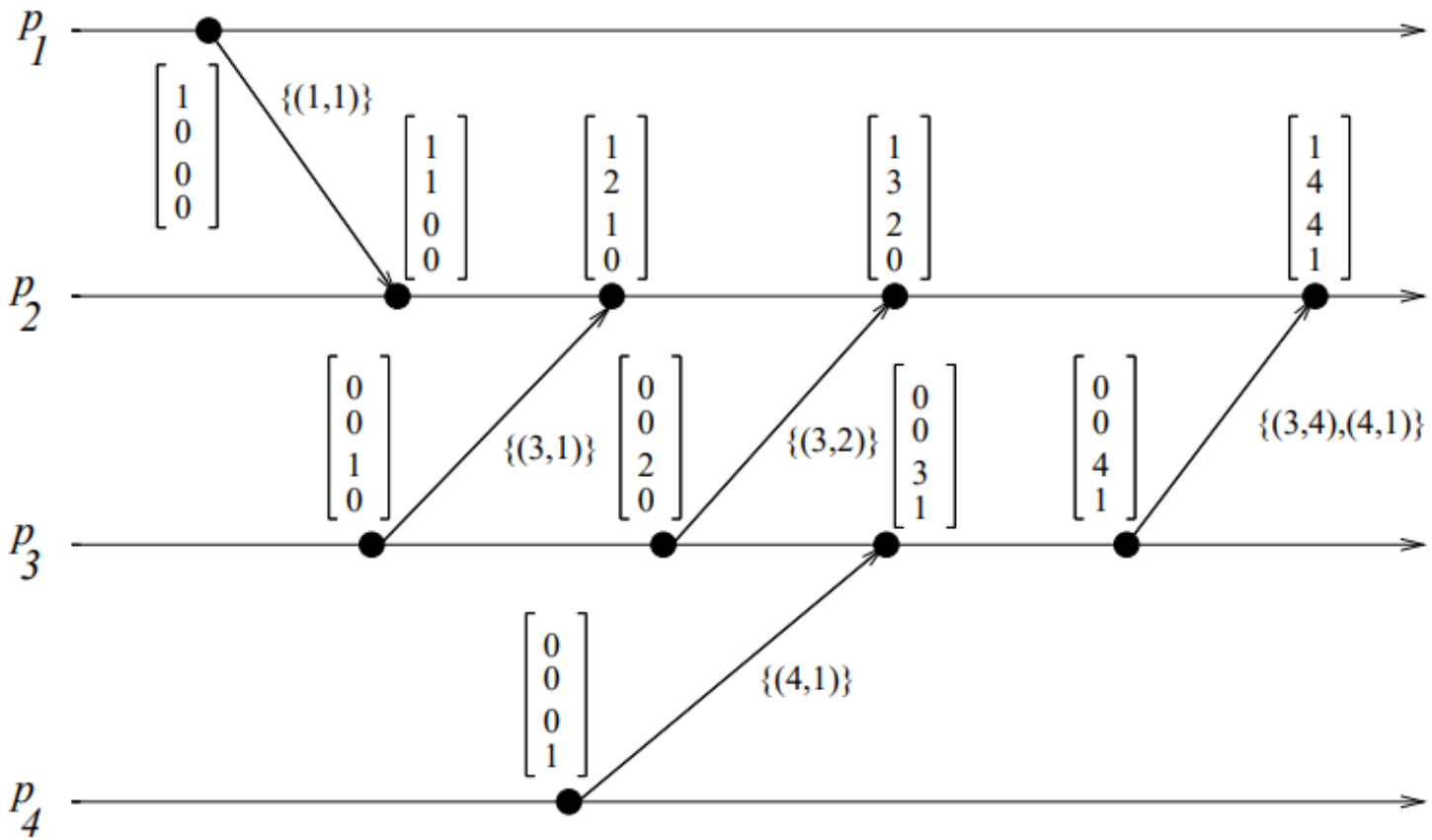


Figure 3.3: Vector clocks progress in Singhal-Kshemkalyani technique.

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 $\{(p_1, latest\_value), (p_2, latest\_value), \ldots\}$, where p_i indicates that the p_ith 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

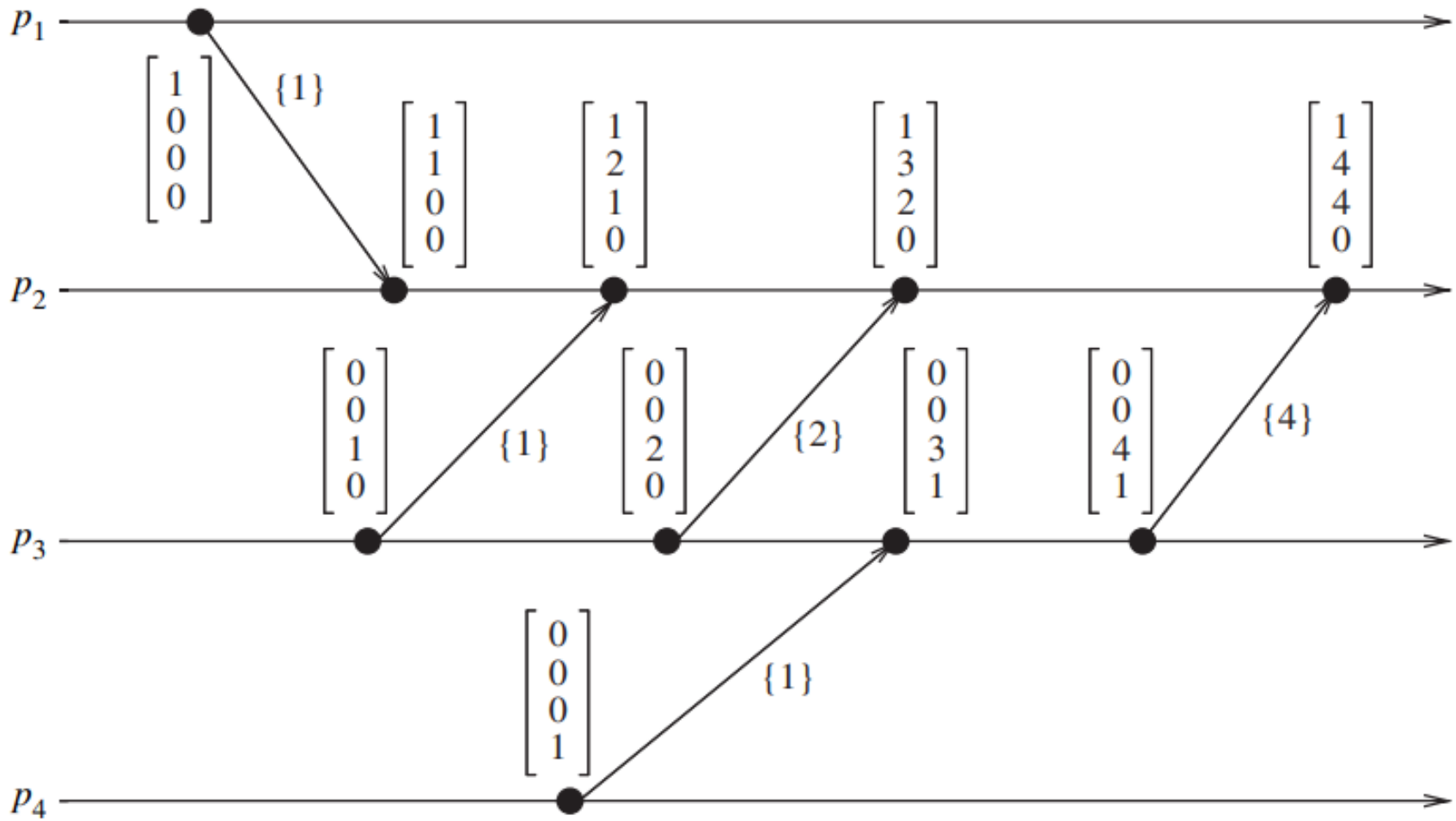
This technique further reduces the message size by only sending the single clock value of the sending process with a message. But it will be difficult to track the event dependencies.

Each process p_i maintains a dependency vector D_i. Initially,

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

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: $D_i[j] := max\{D_i[j], d\}$.


p_1
p_2
p_3
p_4
1 0 0 0
{1}
1 1 0 0
1 2 1 0
1 3 2 0
1 4 4 0
0 0 1 0
{1}
0 0 2 0
{2}
0 0 3 1
0 0 4 1
{4}
0 0 0 1
{1}

```
DependencyTrack(i : process, σ : event index)
\* Casual distributed breakpoint for σ_i *\
\* DTV holds the result *\
for all k ≠ i do
    DTV[k]=0
end for
DTV[i]=σ
end DependencyTrack

VisitEvent(j : process, e : event index)
\* Place dependencies of τ into DTV *\
for all k ≠ j do
    α = D_j^e[k]
    if α >DTV[k] then
        DTV[k]=α
        VisitEvent(k, α)
    end if
end for
end VisitEvent
```

Algorithm 3.1 Recursive dependency trace algorithm

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.

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:

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

- *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

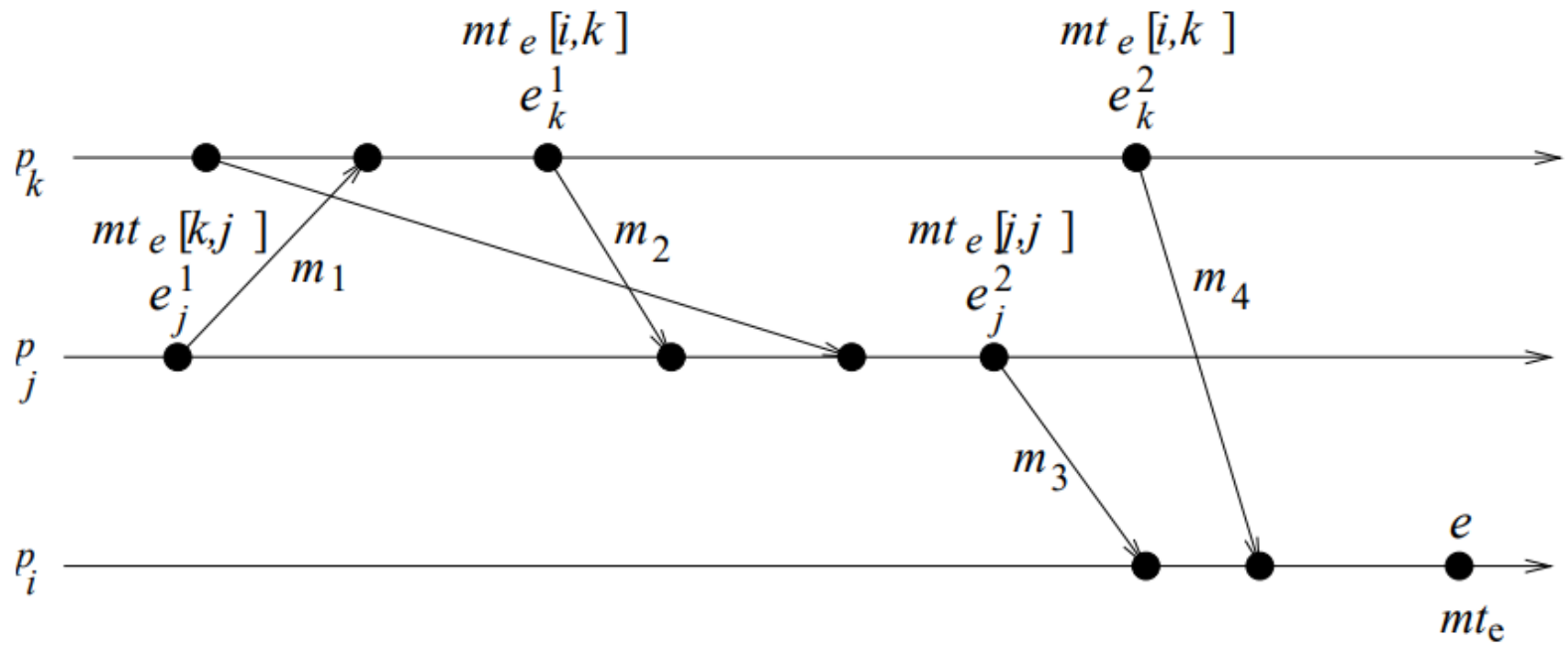
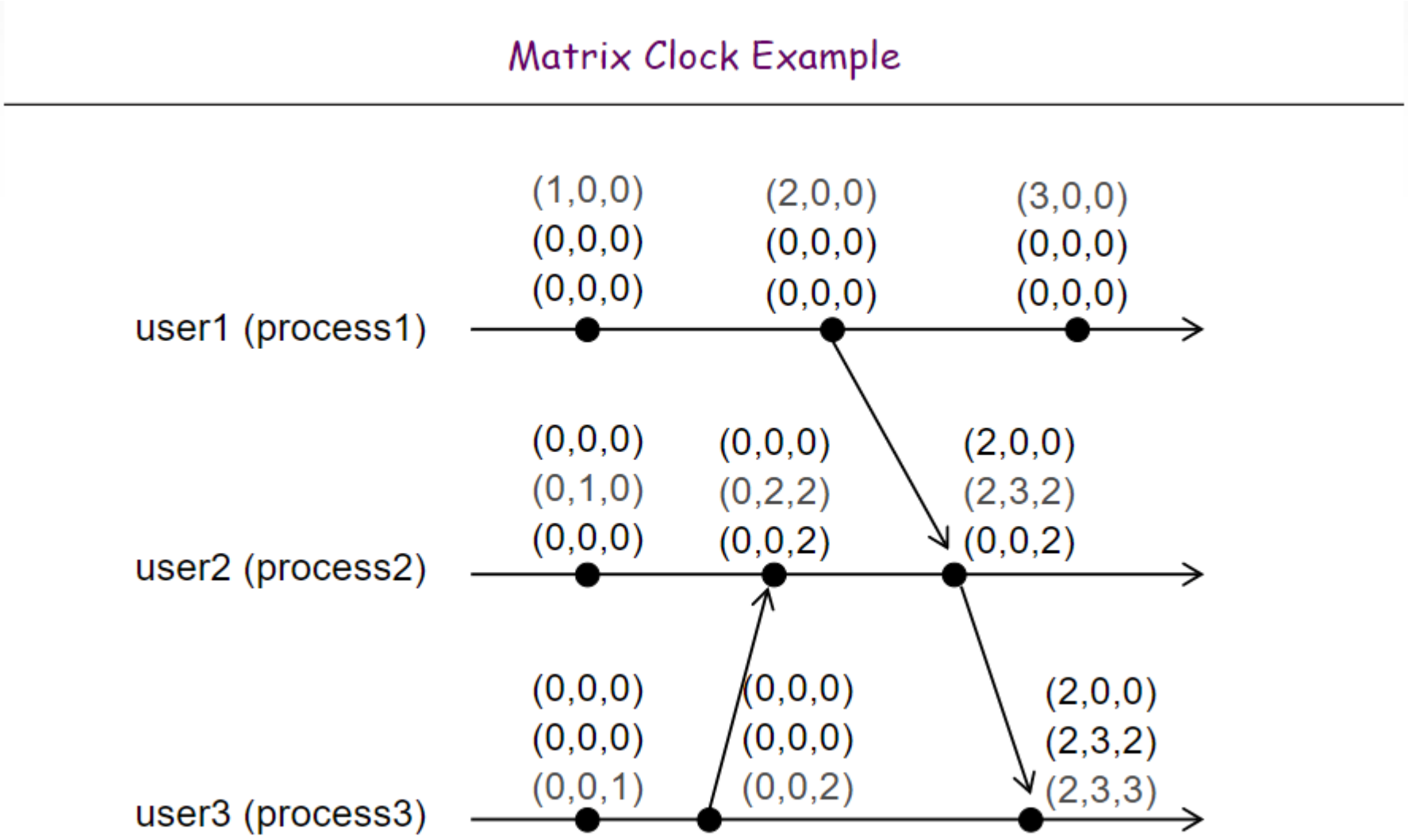
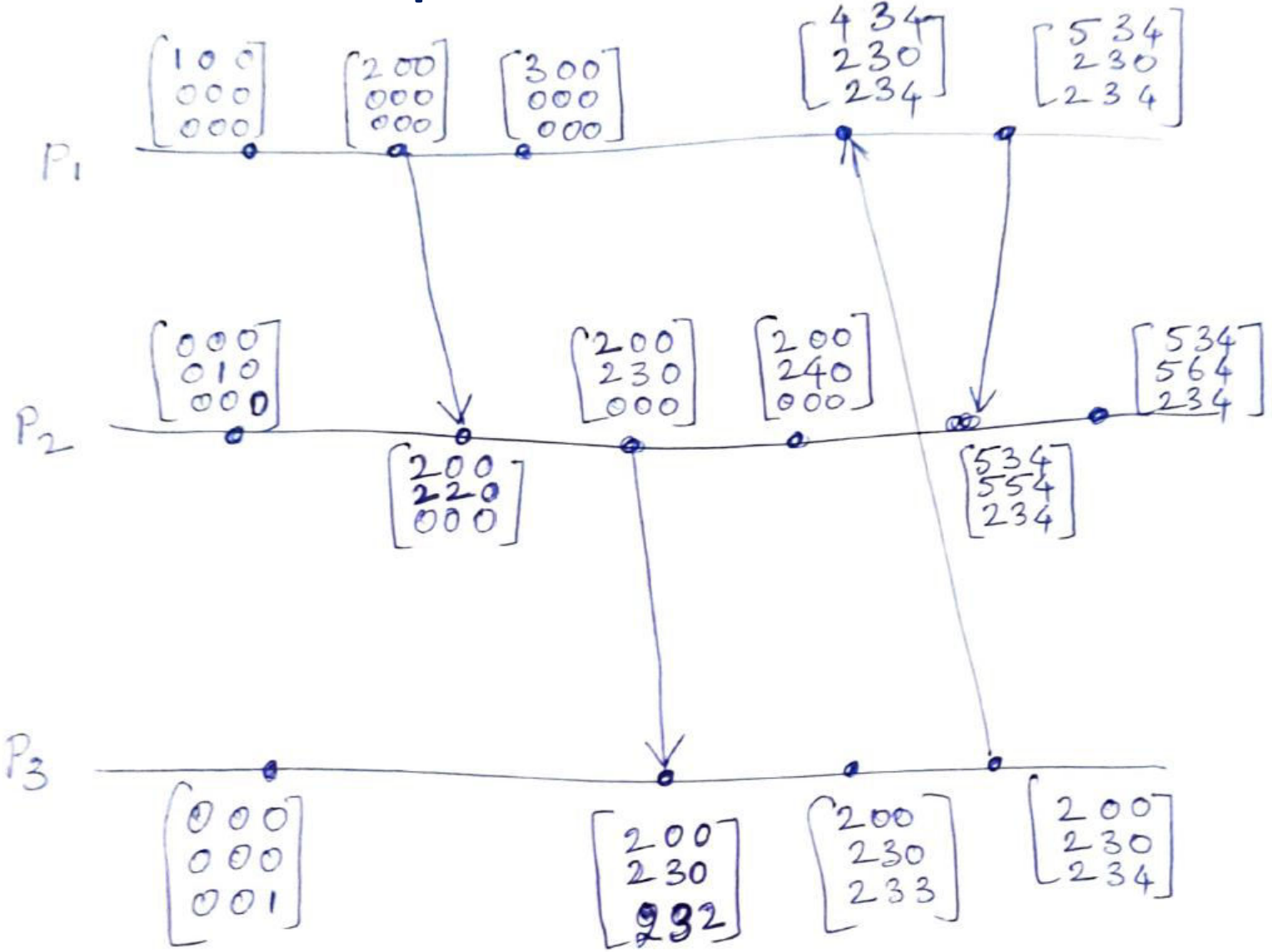


Figure 3.4: Evolution of matrix time.

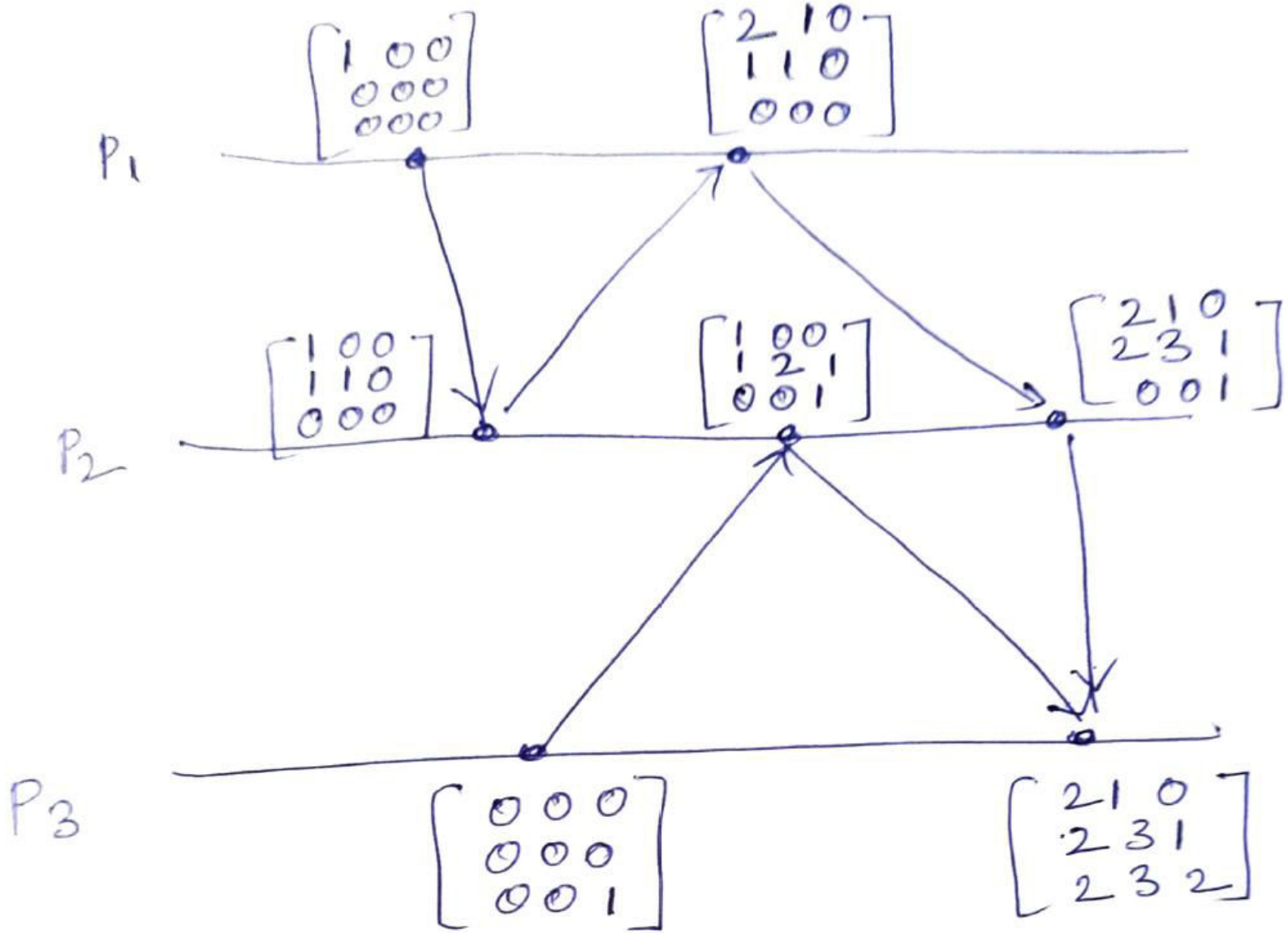
Matrix Time – Example 1



Matrix 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.