CS621 Handouts Complete

Parallel and Distributed Computing (Virtual University of Pakistan)

CS621 Parallel and Distributed Computing

About The Course

Topic

1) Introduction of Parallel and
   Distributed Computing
2) Classification for Computer Organization
3) Interconnection Networks/Topologies
4) Concurrency Control & Memory Hierarchies
5) Memory Consistency Model
6) Fault Tolerance & Load Balancing
7) Heterogeneity / GPU Architecture
   and Programming
8) Message Passing Interface (MPI)
9) Multithreaded Programming
10) Parallel Algorithms & Architectures
11) Parallel I/O, Performance Analysis and Tuning Power
12) Scalability And Performance Studies
13) Programming Models, Scheduling and
    Storage Systems
14) Synchronization and OpenMP
15) OpenMP, Types of Distributed Systems

Why study Parallel Computing?

1. Today, serial computers do not exists.
2. Even, your mobile phone is a parallel machine.
3. CPU's Clock frequencies are getting lower.
4. Number of processors are increasing.
5. Application will get slower, if we don't learn the skill to develop parallel programs

Week 1

What is Computing?

Objectives

Introduction of Computing.

History of Computing.

What is Computing

"Computing is the process to complete a given goal-oriented task by using computer technology."

Computing may include the design and development of software and hardware systems for broad range of purposes.

Used for structuring, processing and managing any kind of information - to aid in the pursuit of scientific studies and making intelligent systems.

History of Computing

Batch Era

Time Sharing Era

Desktop Era

Network Era

Batch Era: Execution of series of programs on a computer without manual intervention.

Time Sharing: Sharing of computing resource among many users by means of multiprogramming and multi-tasking.

Desktop Era: A personal computer provides computing power to one user.

Network Era: Systems with shared memory and distributed memory.

Serial Vs. Parallel Computing

Objectives

Serial Computing.

Parallel Computing.

Difference between serial and parallel computing

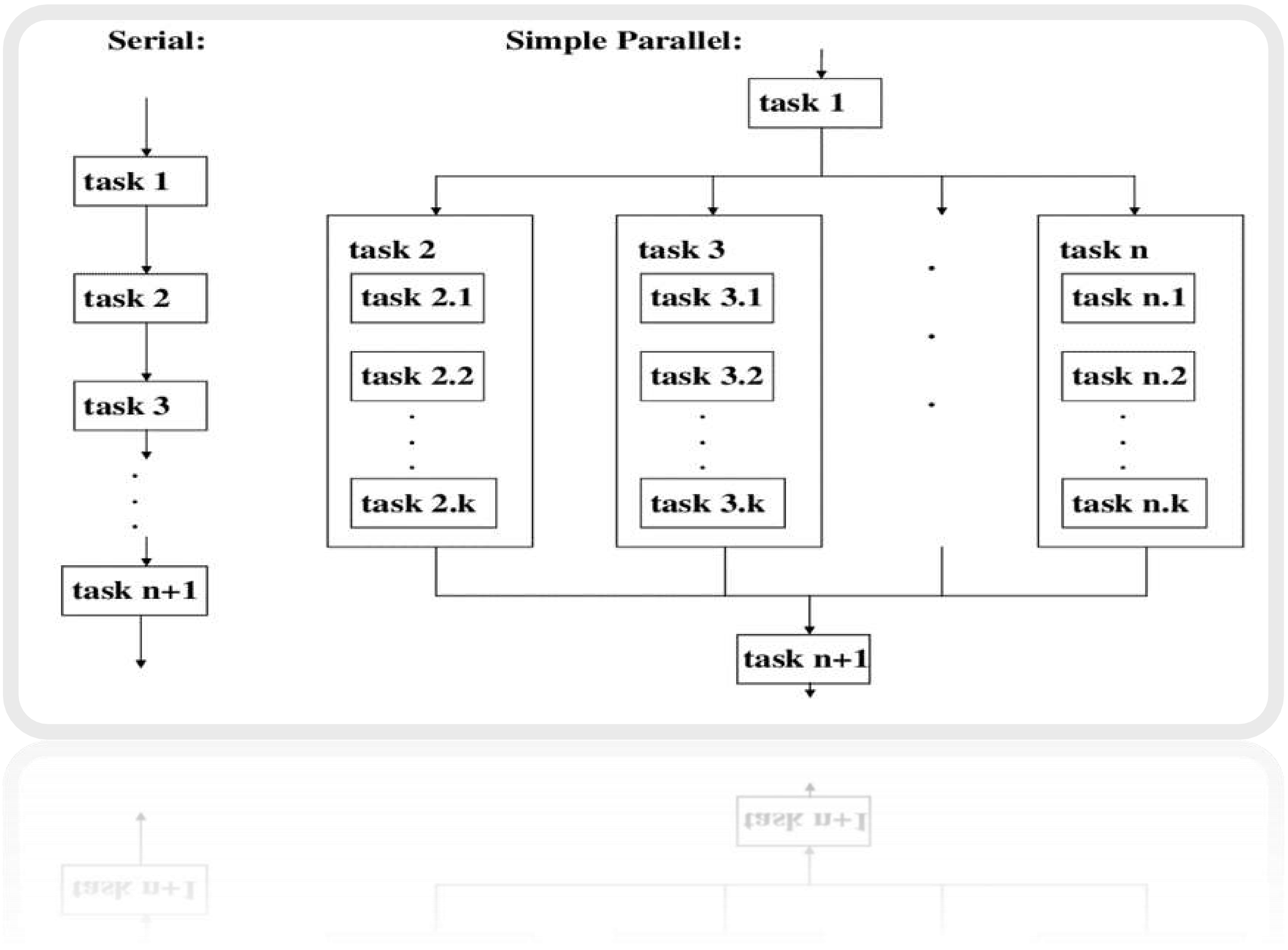
Serial Computing

“Serial computing is a type of processing in which <u>one task is completed at a time</u> and all the tasks are executed by the processor in a sequence.”

Parallel Computing

“Parallel computing is a type of computing architecture in which several processors simultaneously execute multiple, smaller calculations broken down from an overall larger, complex problem.”

Serial vs parallel Computing



Difference between Serial and parallel Computing

Serial Computing	**Parallel Computing**
Are uniprocessor systems.	Are multiprocessor systems.
Can execute one instruction at a time	Can execute multiple instructions At a time
Speed is limited.	No limitation on speed.
Lower performance.	Higher performance.
Examples: EDVAC, BINAC, and LGP-30.	Example: Window 7, 8 and 10.

Introduction to Parallel Computing

Objectives

Parallel Computing

Multi-Processer

Multi-Core

Introduction to Parallel Computing

“Parallel Computing is the simultaneous execution of the same task (split up and adapted) on multiple processors in order to obtain faster results.”

It is a kind of computing architecture where the large problems break into independent, smaller, usually similar parts that can be processed in one go. It is done by multiple CPUs communicating via shared memory, which combines results upon completion. It helps in performing large computations as it divides the large problem between more than one processor.

HPC: High Performance/Productivity Computing

Technical Computing

Cluster computing

The term parallel computing architecture sometimes used for a computer with more than one processor available for processing.

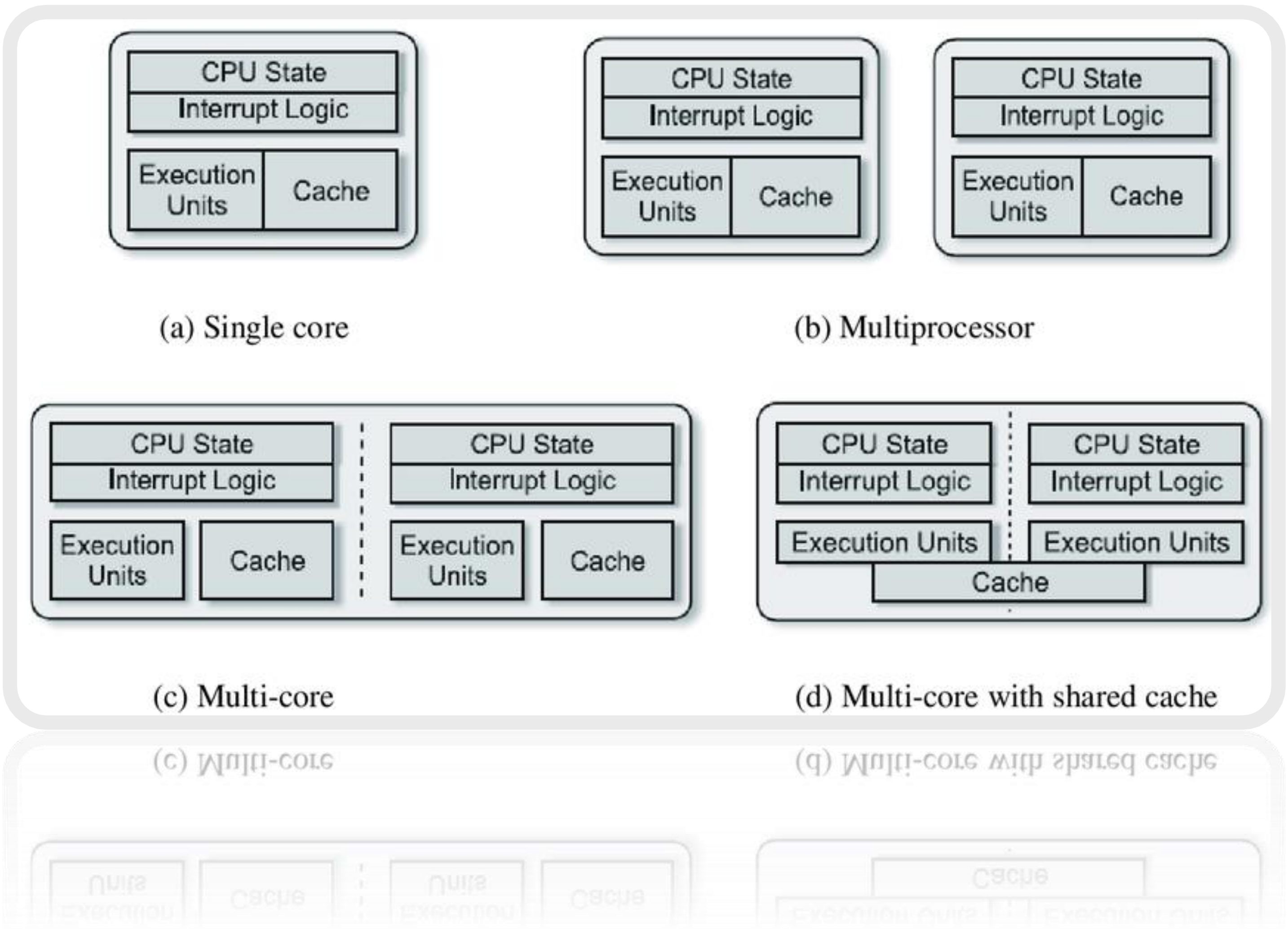
The recent multicore processor (chips with more than one processor core) are some commercial examples which bring parallel computing to the desktop.

Multi-processor

More than one CPU works together to carry out computer instructions or programs**.**

Multi-core

Is a microprocessor on a single integrated circuit with two or more separate processing units, called cores, each of which reads and executes program instructions.



Principles of Parallel Computing

Objectives

Finding enough parallelism

Scale

Locality

Load balance

Coordination and Synchronization

Performance modeling

Finding Enough Parallelism

Conventional architectures coarsely comprise of a processor, memory system, and the data-path. Each of these components' present significant performance bottlenecks. Parallelism addresses each of these components in significant ways.

Scale

Parallelism overhead includes cost of starting a head, accessing data, communicating shared data, synchronization and extra computation. Algorithms needs sufficiently large units of work to run fast in parallel

Locality

Parallel processors collectively have large and fast cache. The memory addresses are distributed across the processors, a processor may have faster access to memory locations mapped locally than to memory locations mapped to other processors

Load Balance

Determines the workload, divide up evenly before staring in case of static load balancing but in dynamic load balance workload changes dynamically, need to rebalance dynamically.

Coordination and Synchronization

Several kind of synchronization is needed by processes cooperating to perform computation

Performance Modeling

More efficient programming models and tools formulated for massively parallel supercomputers.

Why Use Parallel Computing?

Objectives

Identifying the aspects that make parallel computing more useful.

Computing power

Modern consumer grade computing hardware comes equipped with multiple central processing units (CPUs) and/or graphics processing units (GPUs) that can process many sets of instructions simultaneously

Performance

Theoretical performance steadily increased, due to the fact that performance is proportional to the product of the clock frequency and the number of cores.

Scalability

Problem can be scaled up from size to sizes that were out of reach with a serial application. The larger problem sizes are enabled by the larger amounts of main memory, disk storage, bandwidth over networks and to disk, and CPUs

Solve large problems

Solve large problems by breaking down larger problems into smaller, independent, often similar parts that can be executed simultaneously by multiple processors communicating via shared memory. Can solve large problems like Web search engines, processing millions of transaction per second, etc.

Cost

the cost of computation would reduce if we are to deploy parallel computation than sequential computation.

Provide concurrency

Parallelism leads naturally to Concurrency. For example, Several processes trying to print a file on a single printer

Week 2

Introduction to Distributed Computing

Detailed explanation of distributed computing.

Introduction to Distributed Computing

"A distributed system is a collection of independent computers that appears to its users as a single coherent system."

In distributed computing

- We have multiple autonomous computers which seems to the user as single system.
- There is no shared memory and computers communicate with each other through message passing.
- A single task is divided among different computers.

Uses or coordinates physically separate computing resources:

- Grid computing
- Cloud Computing

Why Use Distributed Computing?

Objective

Identifying the aspects that makes distributed computing more useful.

Heterogeneity

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks.

Distribution Transparency

A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent.

Openness

An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services.

Resiliency

With multiple computers, redundancies are implemented to ensure that a single failure doesn't equate to systems-wide failure.

Scalability

Scalability of a distributed system can be measured along at least three different dimensions: a system can be scalable with respect to its size, a geographically scalable system, a system can be administratively scalable.

Resources accessibility

The main goal of a distributed system is to make it easy for the users to access remote resources, and to share them in a controlled and efficient way.

Concurrency

Concurrency between programs sharing data is generally kept under control through synchronization mechanisms for mutual exclusion and transaction.

Fault tolerance, Error recovery

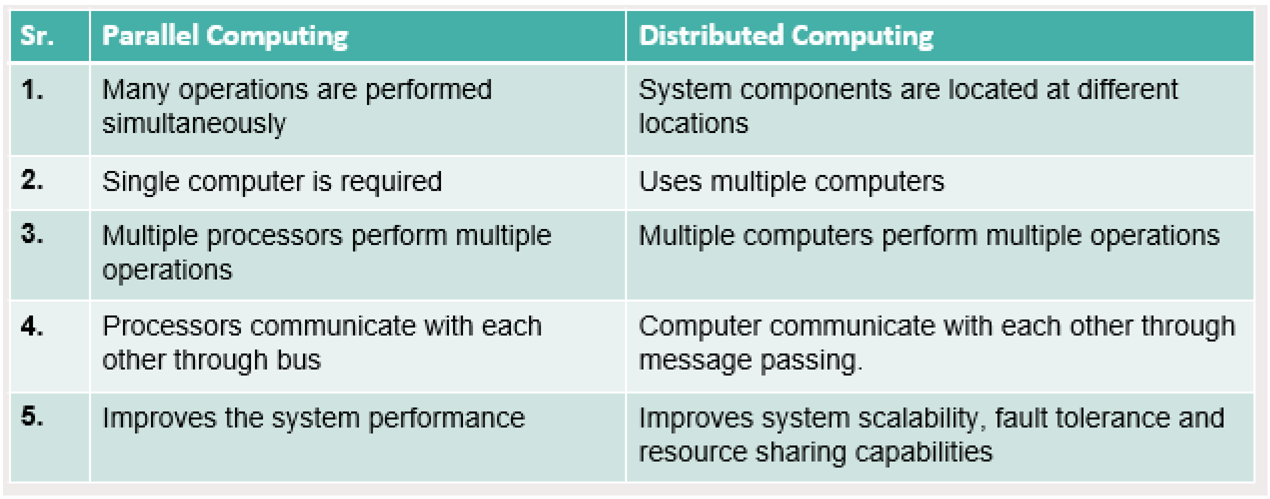
Fault tolerance is an important aspect in distributed systems design. A system is fault tolerant if it can continue to operate in the presence of failures.

Difference between Parallel and Distributed Computing

Objectives

Identifying the key differences between parallel and distributed computing

Distributed computing is often used in tandem with parallel computing. Parallel computing on a single computer uses multiple processors to process tasks in parallel, whereas distributed parallel computing uses multiple computing devices to process those tasks.



Sr.	Parallel Computing	Distributed Computing
1.	Many operations are performed simultaneously	System components are located at different locations
2.	Single computer is required	Uses multiple computers
3.	Multiple processors perform multiple operations	Multiple computers perform multiple operations
4.	Processors communicate with each other through bus	Computer communicate with each other through message passing.
5.	Improves the system performance	Improves system scalability, fault tolerance and resource sharing capabilities

Applications of Parallel and Distributed Computing

Objectives

Scope of parallel and distributed computing.

Science

- Global climate modeling
- Biology: genomics; protein folding; drug design
- Astrophysical modeling
- Computational Chemistry
- Computational Material Sciences and Nano sciences

Engineering

- Semiconductor design
- Earthquake and structural modeling
- Computation fluid dynamics (airplane design)
- Combustion (engine design)
- Crash simulation

Business

- Financial and economic modeling
- Transaction processing, web services and search engines

Defense

- Nuclear weapons -- test by simulations
- Cryptography

P2P Network

- eDonkey, BitTorrent, Skype, …

Google

- 1500+ Linux machines behind Google search engine

Issues in Parallel and Distributed Computing

Failure Handling

Failures, like in any program, are a major problem. With so many processes and users, the consequences of failures are exacerbated.

Scalability

As a distributed system is scaled, several factors need to be taken into account: size, geography, and administration; and their associated problems like overloading, control and reliability

Security

As connectivity and sharing increase, security is becoming increasingly important. Increased connectivity can also lead to unwanted communication, such as electronic junk mail, often called spam.

Process synchronization

One of the most important issues that engineers of distributed systems are facing is synchronizing computations consisting of thousands of components. Current methods of synchronization like semaphores, monitors, barriers, remote procedure call, object method invocation, and message passing, do not scale well.

Resource management

In distributed systems, objects consisting of resources are located on different places. Routing is an issue at the network layer of the distributed system and at the application layer. Resource management in a distributed system will interact with its heterogeneous nature

Communication and Latency

Distributed Systems have become more effective with the advent of Internet but there are certain requirements for performance, reliability etc. Effective approaches to communication should be used.

Parallel and Distributed Computing Efforts

Before a program is written or a piece of software is developed, it must first go through a design process.

For parallel and distributed programs, the design process will include three issues:

Decomposition

Decomposition is the process of dividing up the problem and the solution into parts: logical areas and logical resources.

One of the primary issues of concurrent programming is identifying a natural WBS(Work breakdown structure) for the software solution at hand.

Communication

Following issues must be considered when designing parallel or distributed systems:

Important Questions:

- How will this communication be performed if the parts are in different processes or different computers?
- Do the different parts need to share any memory?
- How will one part of the software know when the other part is done?
- Which part starts first?
- How will one component know if another component has failed?

Synchronization

The WBS designates who does what.

When multiple components of software are working on the same problem, they must be coordinated.

The components' order of execution must be coordinated. . Do all the parts start at the same time or does some work while others wait?

What two or more components need access to the same resource?

Who gets it first? If some of the parts finish their work long before the other parts, should they be assigned new work?

Who assigns the new work in such cases?

Week 3

Shared Memory

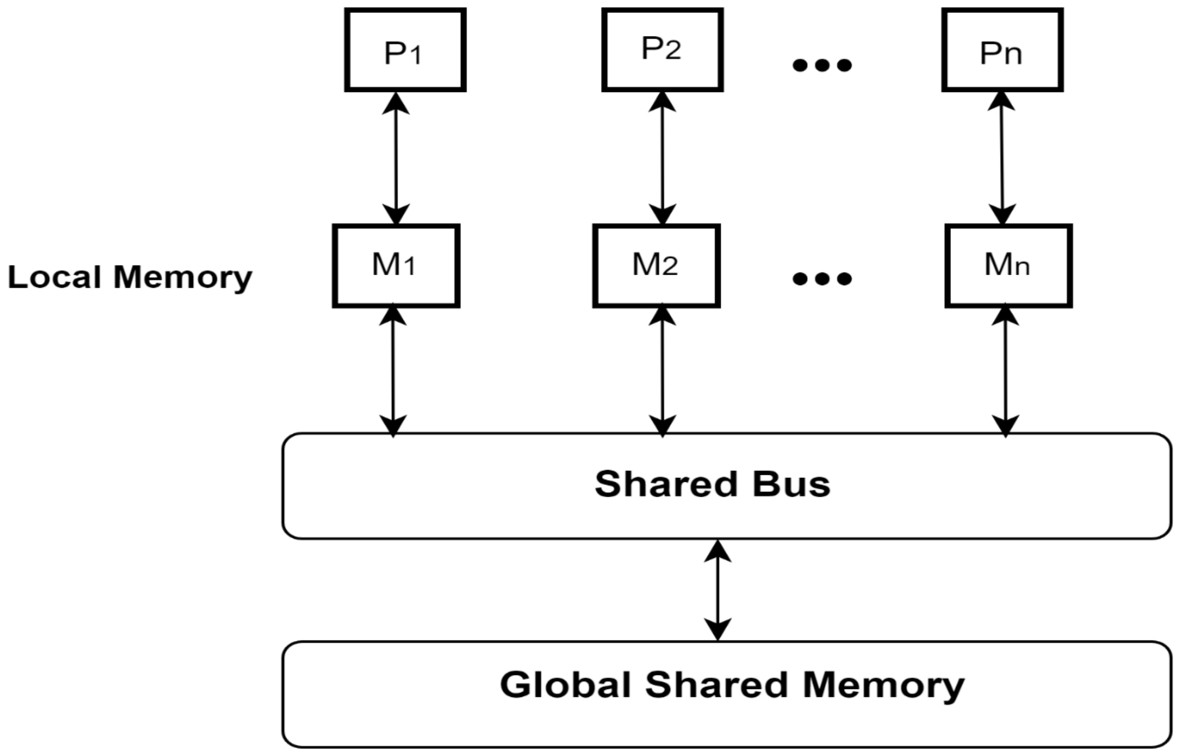
Objectives

- Introduction of Shared Memory.
- Architecture of Shared Memory.

“Shared memory is a type of memory architecture that allows multiple processors or threads to access the same memory space. In the context of distributed and parallel computing, shared memory can be used to facilitate communication and synchronization between different processes or threads.”

- Processors have direct access to global memory and I/O
  through bus or fast switching network
- Cache Coherency Protocol guarantees consistency
  of memory and I/O accesses
- Each processor also has its own memory (cache)
- Data structures are shared in global address space
- Concurrent access to shared memory must be coordinated
- Programming Models
  - Multithreading (Thread Libraries)
  - OpenMP

Architecture of Shared Memory:

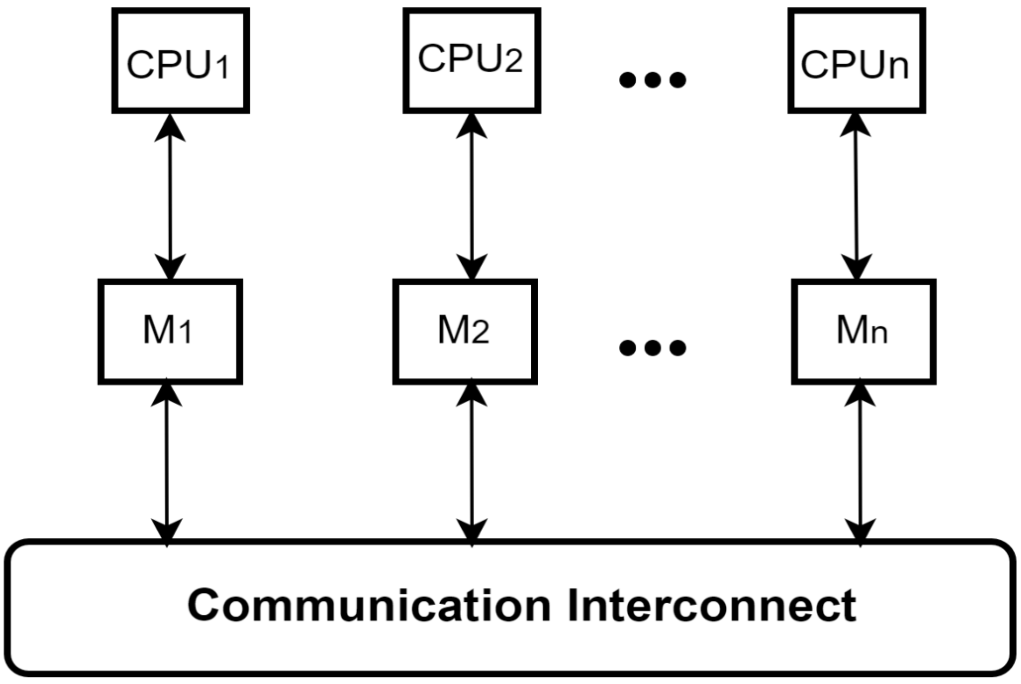


Distributed Memory

"Distributed memory refers to a type of parallel computing architecture where each processor has its own private memory, and communication between processors happens through message passing. In this architecture, the memory of one processor is not directly accessible by other processors, and communication between processors occurs explicitly through messages that are sent and received."

- Each Processor has direct access only to its local memory
- Processors are connected via high-speed interconnect
- Data structures must be distributed
- Data exchange is done via explicit processor-to-processor communication: send/receive messages
- Programming Models
  - Widely used standard: MPI
  - Others: PVM, PARMACS, Express, P4, Chameleon, etc.

Architecture of Distributed Memory:



Flynn’s classification of computer architectures

Objectives

- What is Flynn’s classification of computer architectures?
- Basis for Flynn’s classification.
- Types of Flynn’s classification.

Flynn’s classification of computer architectures (1966):

“Michael J Flynn classified computers on the basis of multiplicity of instruction stream and data streams in a computer system.”

- **Instruction stream**
- **Data stream**
- **Single vs. multiple**

The four classifications defined by Flynn are based upon the number of concurrent instruction (or control) and data streams available in the architecture.

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Multiple Data (MIMD)
- Multiple Instruction Single Data (MISD)

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

SISD (Single-Instruction Single-Data)

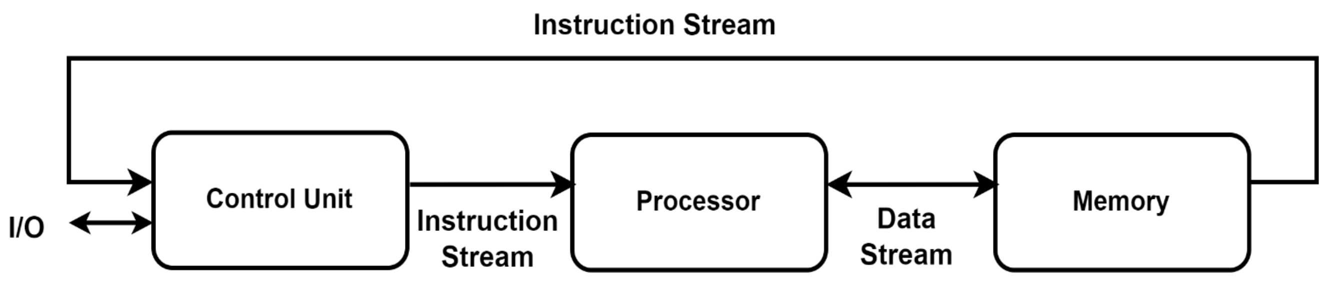
Objectives

- Introduction of SISD.
- Architecture of SISD.

SISD (Single-Instruction Single-Data)

"Refers to the traditional von Neumann architecture where a single sequential processing element (PE) operates on a single stream of data."

SISD (Single-Instruction Single-Data) Architecture



- Conventional single-processor von Neumann computers are classified as SISD systems.
- A typical non-pipelined architecture with general purpose registers as well as some dedicated special registers like:
  - Program Counter (PC),
  - Memory data registers (MDR),
  - Memory address registers (MAR) and

    - Instruction registers (IR).
- Perform the same operation on multiple data operands concurrently.
- Concurrency of processing rather than concurrency of execution.
- Serial computer
- Example: A personal computer processing instructions and data on single processor.

SIMD (Single-Instruction Multi-Data)

Objectives

- Introduction of SIMD.
- SIMD Architecture.
- SIMD Schemes.

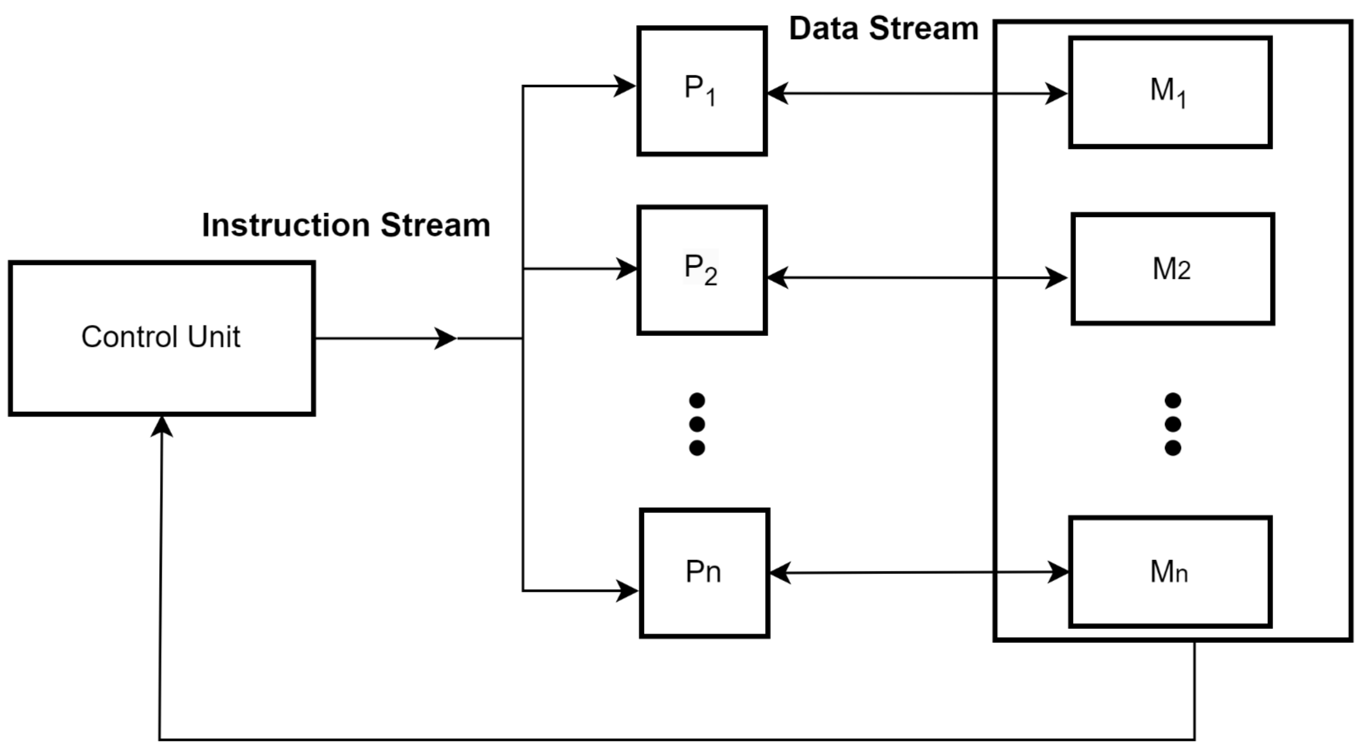
SIMD (Single-Instruction Multi-Data)

"SIMD is a multiple-processing system that performs one operation simultaneously on more than one piece of data."

- All processors in a parallel computer execute the same instructions but operate on different data at the same time.
- Processors run in synchronous, lockstep function.
- Shared or distributed memory
- SIMD instructions give very speedups in things like linear algebra, or image/video manipulation/encoding/decoding, etc.
- Examples: Wireless MMX unit, CM-1, CM-2, DAP, MasPar MP-1

SIMD (Single-Instruction Multi-Data) Architecture

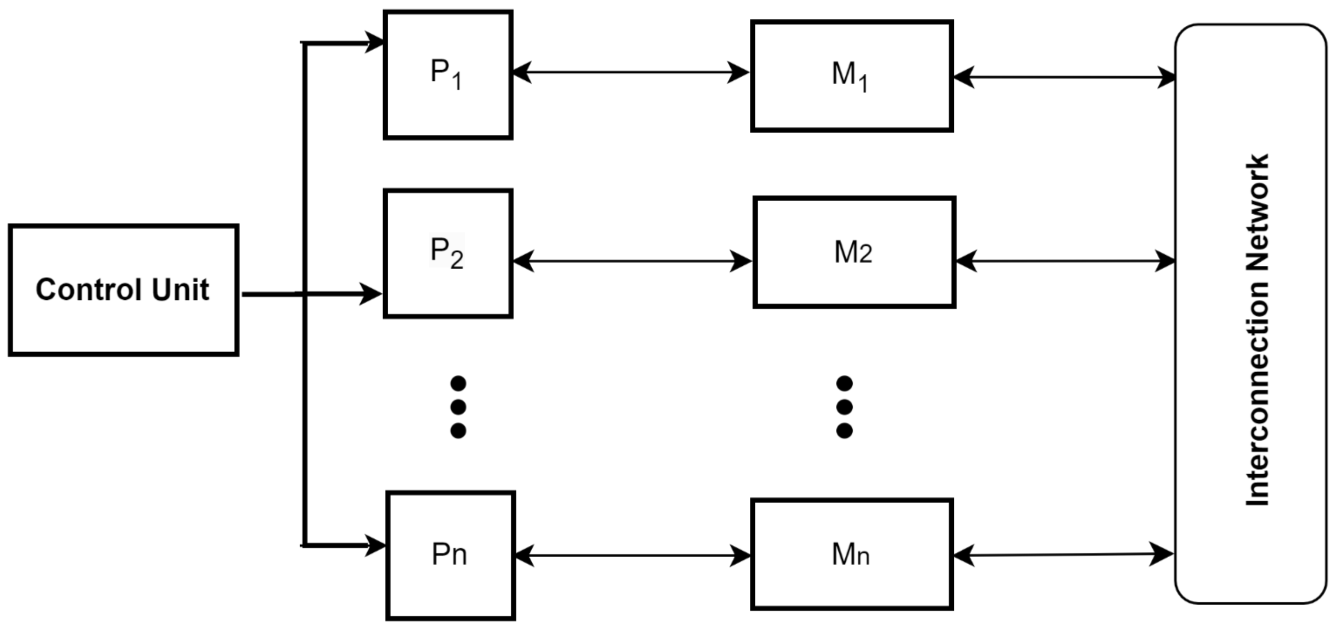
- Consists of 2 parts:
    - A front-end Von Neumann computer.
    - A processor array: connected to the memory bus of the front end.



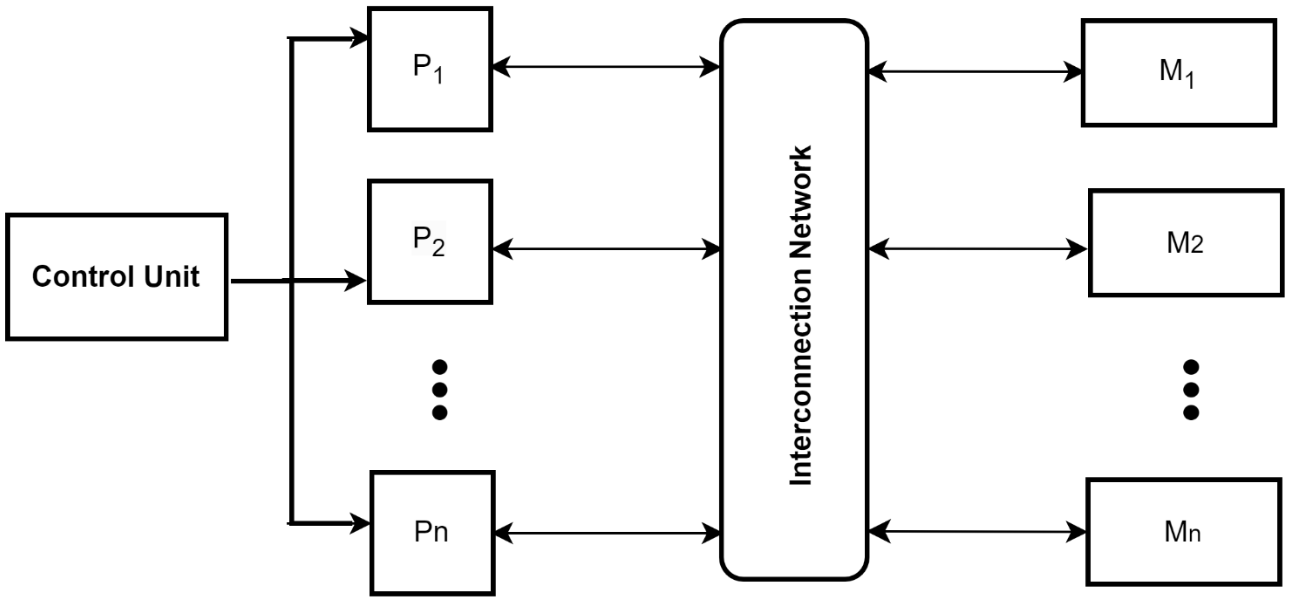
SIMD (Single-Instruction Multi-Data) Schemes

- Classified into two configuration schemes
    - Scheme 1 – Each processor has its own local memory.
    - Scheme 2 – Processors and memory modules communicate with each other via interconnection network.

Scheme 1



- **SIMD Scheme 2**



MISD (Multiple-Instruction Single-Data)

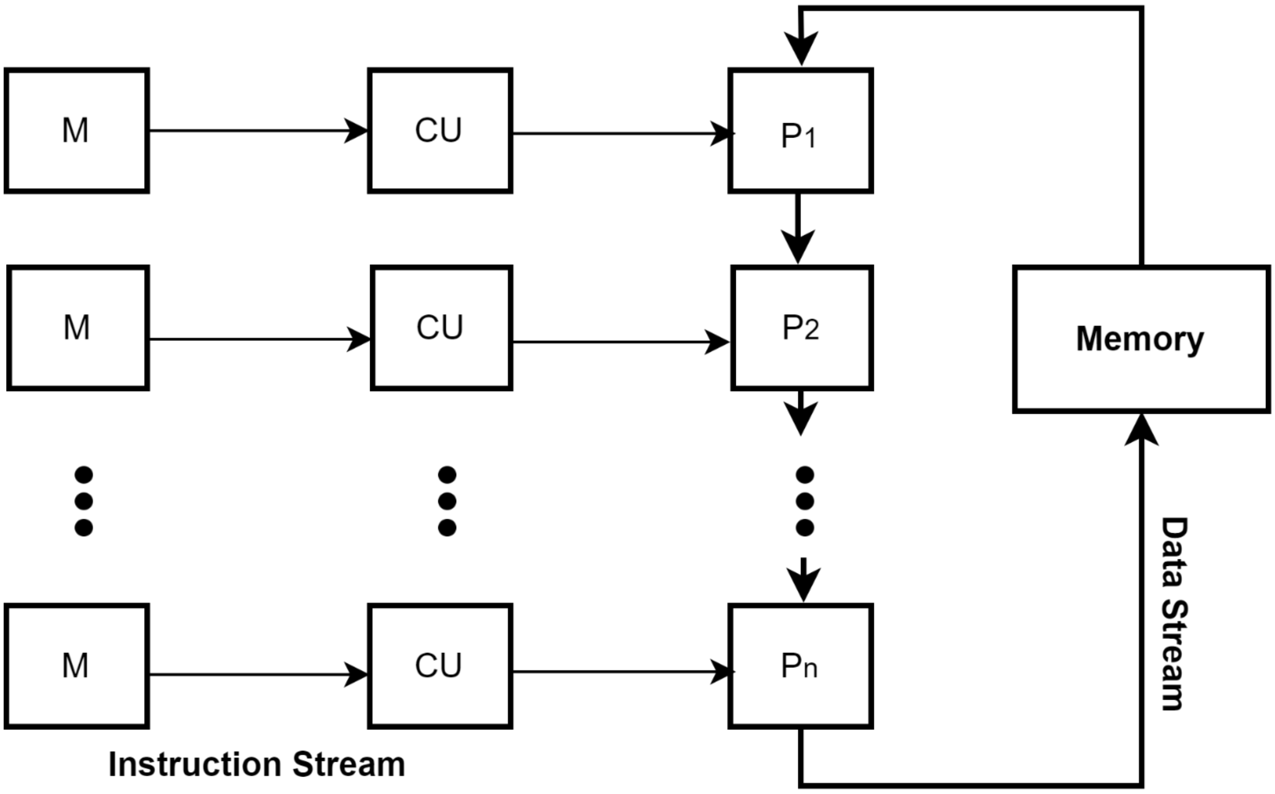
Objectives

- **Introduction of MISD.**
- **MISD Architecture.**

MISD (Multiple-Instruction Single-Data)

"A pipeline of multiple independently executing functional units operating on a single stream of data, forwarding results from one functional unit to the next."

MISD (Multiple-Instruction Single-Data) Architecture



- Special purpose computer
- Excellent for situation where fault tolerance is critical
- Heterogeneous systems operate on the same data stream and must agree on the result
- Rarely used; some specific use systems (space flight control computers)
- Example: Systolic array

MIMD (Multi-Instruction Multi-data)

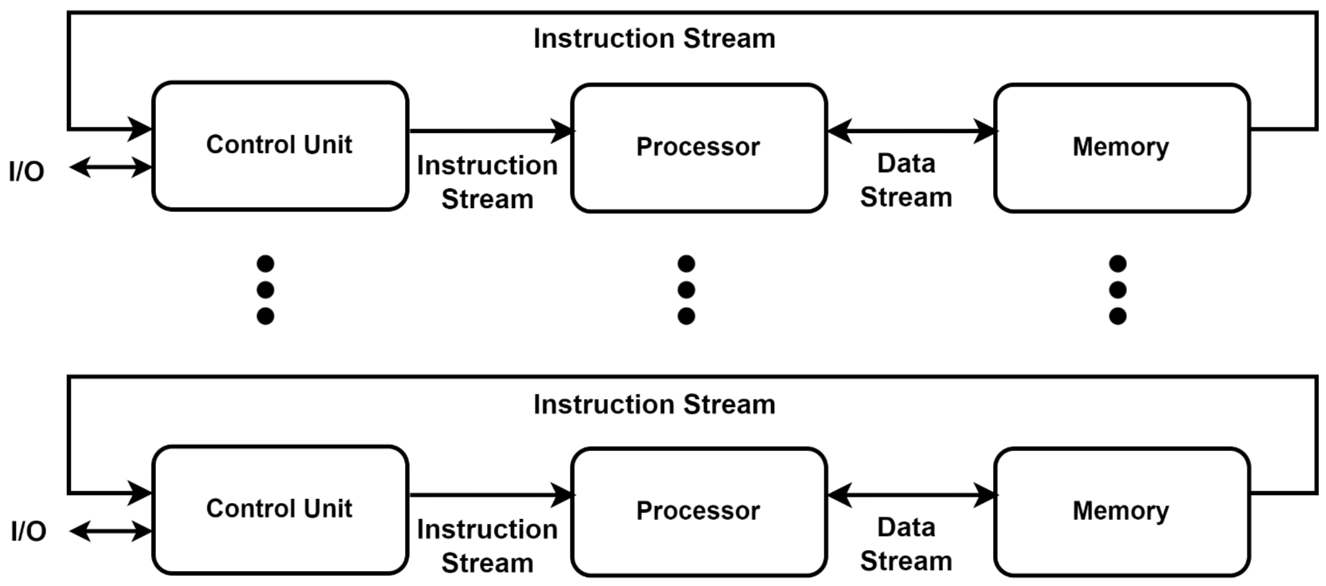
Objectives

- Introduction of MIMD.
- MIMD Architecture.
- MIMD Shared Memory & Message Passing Architectures.

MIMD (Multi-Instruction Multi-data)

"In MIMD all processors in a parallel computer can execute different instructions and operate on various data at the same time."

MIMD (Multi-Instruction Multi-data) Architecture



- MIMD processors can execute different programs on different processors.
- Each processor has a separate program, and an instruction stream is generated from each program.
- Parallelism achieved by connecting multiple processors together.
- Different programs can be run simultaneously.
- Each processor can perform any operation regardless of what other processors are doing.
- Examples: S-1, Cray-3, Cray T90, Cray T3E, Multiprocessor PCs and 370/168 MP

Special purpose computer

Shared or distributed memory

Made of multiple processors and multiple memory modules connected via some interconnection network.

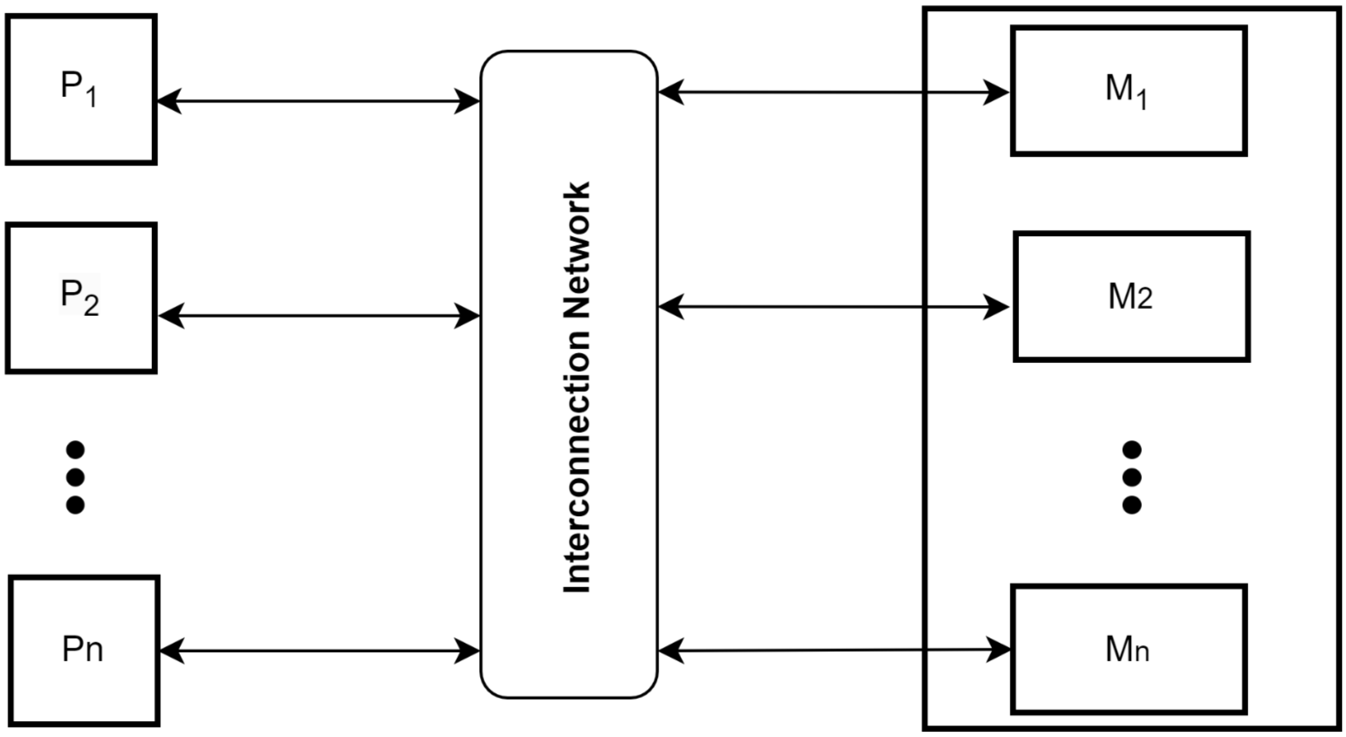
Classified into two configuration schemes:

- Shared memory
- Message passing

Shared Memory Organization

- Inter-processor coordination is accomplished by reading and writing in a global memory shared by all processors.
- Any processor can access then local memory of any other processor.
- Typically consists of servers that communicate through a bus and cache memory controller.

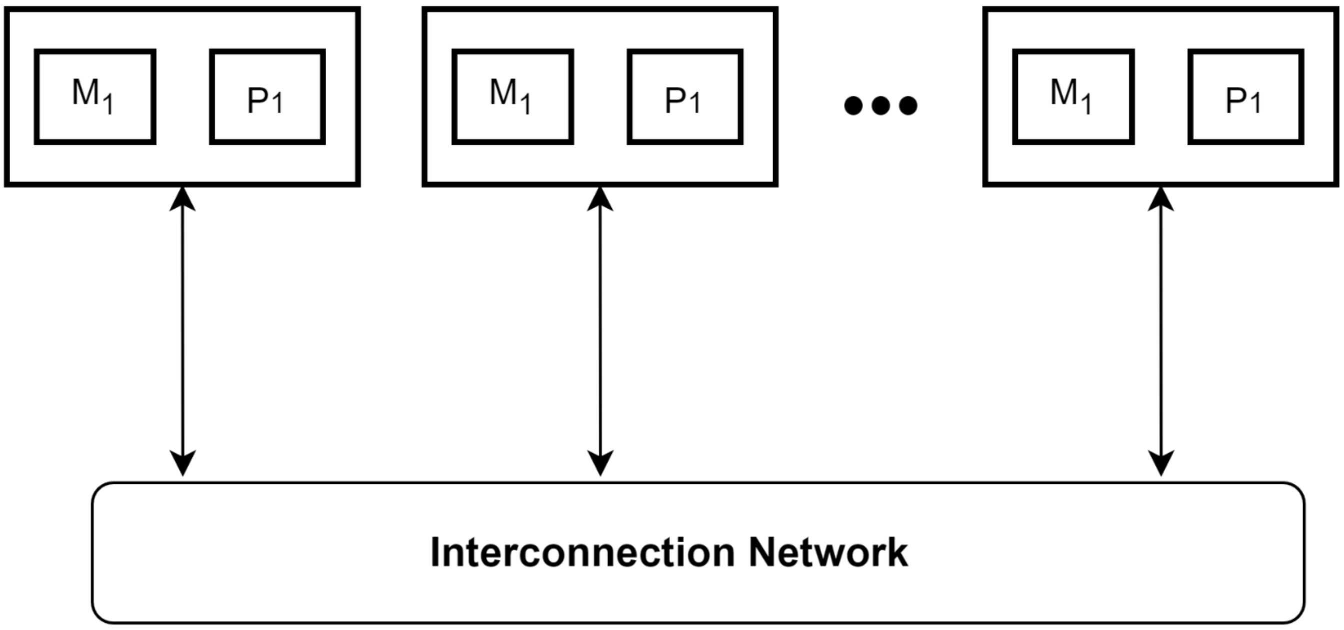
Shared Memory Architecture



Message Passing Organization

- Point-to-Point
- Each processor has access to its own local memory.
- Communications are performed via send and receive operations.
- Message passing multiprocessors employ a variety of static networks in local communications.
- Must be synchronized among different processors.

Message Passing Architecture



SIMD-MIMD Comparison

Objectives

- **Comparison of SIMD-MIMD.**

SIMD-MIMD Comparison

- SIMD computers require less hardware than MIMD computers (single control unit).
- SIMD is less expensive as compared to MIMD.
- SIMD follows synchronous processing where as, MIMD incorporates an asynchronous processing.
- In MIMD each processing elements stores its individual copy of the program which increases the memory requirements. Conversely, SIMD requires less memory as it stores only one copy of the program.
- MIMD is more complex and efficient as compared to SIMD.

Architecture	Single Instruction Multiple Data (SIMD)	Multiple Instruction Multiple Data (MIMD)
Type of processing	Same instruction on multiple data sets	Different instructions on multiple data sets
Flexibility	Limited	High
Memory access	Shared	Separate
Best suited for	Applications with regular data parallelism	Applications with irregular data dependencies or complex computations
Performance	High efficiency for parallelizable tasks	Flexible, but may be less efficient for parallelizable tasks

Week 4

Introduction to Fault Tolerance

Objectives

- Introduction of Fault Tolerance.
- Fault Classification.
- Failure Classification.
- Failure Masking.

Fault Tolerance

"A fault-tolerance system is one that continues to provide the required functionality in the presence of fault/failure."

- A characteristic feature of distributed systems is the notion of partial failure:
    - A partial failure may happen when one component in a distributed system fails.
    - This failure may affect the proper operation of other components, while at the same time leaving yet other components totally unaffected.
- An important goal in distributed systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting the overall performance.

Fault Classification

Faults are generally classified as transient, intermittent, or permanent:

- **Transient fault:** Occurs once and then disappear. If the operation is repeated, the fault goes away.
- **Intermittent fault:** Occurs, then vanishes of its own accord, then reappears, and so on. A loose contact on a connector will often cause an intermittent fault.
- **Permanent fault:** Is one that continues to exist until the faulty component is replaced. Burnt-out chips, software bugs, and disk head crashes are examples of permanent faults.

Failure Classification

Faults are generally classified into five categories

- **Crash failure:** A server halts but working correctly until it halts.
- **Omission failure:** A server fails to respond to incoming requests.
- **Timing failure:** A server's response lies outside the specified time interval.
- **Response failure:** A server's response is incorrect.
- **Arbitrary failure:** A server may produce the arbitrary responses at arbitrary times.

Failure Masking

"Failure masking is a fault tolerance technique that hides occurrence of failures from other processes."

- The most common approach to failure masking is redundancy which is categorized into three types
    - **Information redundancy:** Add extra bits to allow recovery from garbled bits.
    - **Time redundancy:** Repeat an action if needed.
    - **Physical redundancy:** Add extra equipment or processes so that the system can tolerate the loss or malfunctioning of some components.

Process Resilience

Objectives

- Introduction of Process Resilience
- Flat Groups versus Hierarchical Groups.
- Failure Masking and Replications
- Approaches for Replications

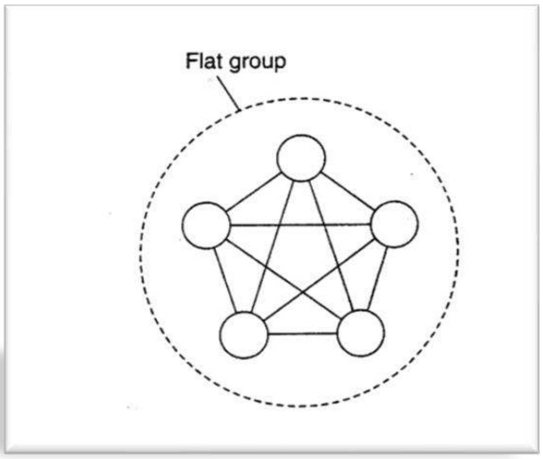
Process Resilience

“Process resilience incorporates techniques by which one or more processes can fail without seriously disturbing the rest of the system.”

- Related to this issue is reliable multicasting, by which message transmission to a collection of processes is guaranteed to succeed.
- Reliable multicasting is often necessary to keep processes synchronized.
- Protection against process failures can be achieved by process replication, organizing several identical processes into a group.
- Groups are categorized into two categories: Flat Group and Hierarchy Group.

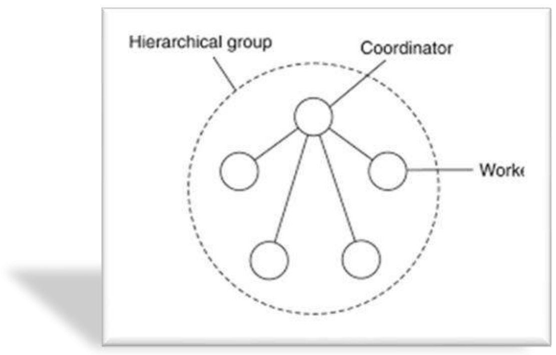
Flat Group

- All processes are equal.
- The processes make decisions collectively.
- No single point of failure, but decision making is more complicated as consensus is required.



Hierarchical Group

- A single coordinator makes all decisions.
- Single point-of failure, however: decisions are easily and quickly made by the coordinator without first having to get consensus.
- Group is transparent to its users; the whole group is dealt with as a single process.



Failure Masking and Replication

- By organizing a fault tolerant group of processes , we can protect a single vulnerable process.
- Two approaches to arranging the replication of the group are:
    - Primary-base protocols and Replicated-write protocols

Primary-Base Protocols

- Appears in the form of a primary-backup protocol
- A group of processes is organized in a hierarchical fashion in which a primary coordinates all write operations.
- When the primary crashes, the backups execute some election algorithm to choose a new primary.

Replicated-Write Protocols

- Replicated-write protocols are used in the form of active replication, as well as by means of quorum-based protocols.
- Solutions correspond to organizing a collection of identical processes into a flat group.
- These groups have no single point of failure, at the cost of distributed coordination.

Reliable Client-Server Communication

Objectives

- Understanding of Reliable Client-Server Communication
- RPC Semantics in the Presence of Failures.

Reliable Client-Server Communication

- Fault tolerance in distributed systems concentrates on faulty processes.
- However, communication failures should also be considered .
- A communication channel may exhibit crash, omission, timing, and arbitrary failures.

Peer to Peer Communication

- Reliable point-to-point communication is established by making use of a reliable transport protocol, such as TCP.
- TCP masks omission failures, which occur in the form of lost messages by using acknowledgments and retransmissions.
- Crash failures of connections are not masked. The only way to mask such failures is to let the distributed system attempt to automatically set up a new connection.

RPC Semantics in the Presence of Failures

- Remote Procedure Call (RPC) mechanism works well as long as both the client and server function perfectly.
- Five classes of RPC failure can be identified:
    - The client is unable to locate the server.
    - The request message from the client to the server is lost
    - The server crashes after receiving a request.
    - The reply message from the server to the client is lost.
    - The client crashes after sending a request.

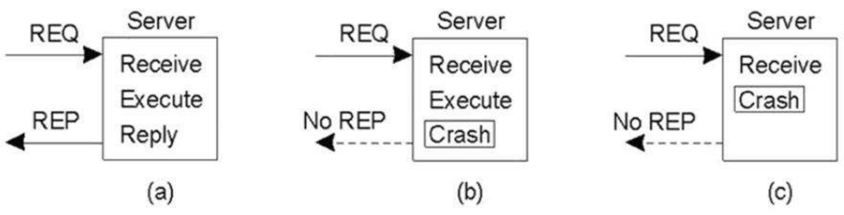
Server in Client-Server Communication

The sequence of events at a server is shown in Fig.

(a) A request arrives, is carried out, and a reply is sent.

(b) A request arrives and is carried out, just as before, but the server crashes before it can send the reply.

(c) Again, a request arrives, but this time the server crashes before it can even be carried out and no reply is sent back.



A server in client-server communication
a) Normal case
b) Crash after execution
c) Crash before execution

Server crashes are dealt with by implementing one of three possible implementation philosophies:

- **At least once semantics:** A guarantee is given that the RPC occurred at least once, but (also) possibly more that once.
- **At most once semantics:** A guarantee is given that the RPC occurred at most once, but possibly not at all.
- **No semantics:** Nothing is guaranteed, and client and servers take their chances

Client in Client-Server Communication

When a client sends a request to a server and crashes before the server replies.At this point a computation is active and no parent is waiting for the result. Such an unwanted computation is called an orphan. Four orphan solutions have been proposed:

- **Extermination:** The orphan is simply killed-off.
- **Reincarnation:** Each client session has an epoch associated with it, making orphans easy to spot.
- **Gentle reincarnation:** When a new epoch is identified, an attempt is made to locate a requests owner, otherwise the orphan is killed.
- **Expiration:** If the RPC cannot be completed within a standard amount of time, it is assumed to have expired.

Reliable Group Communication

Objectives

- Understanding of Reliable Group Communication.
- Reliable-Multicasting Schemes.

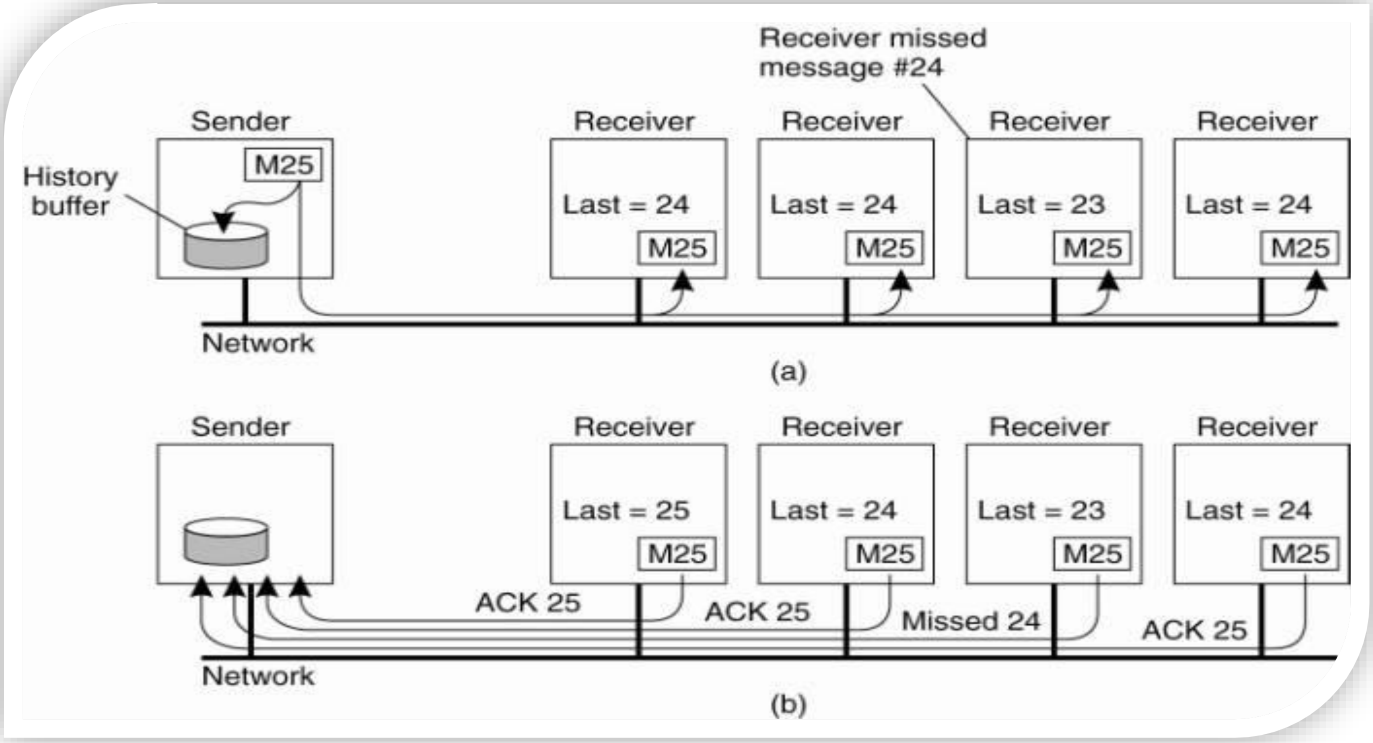
Reliable Group Communication

"Reliable multicast services guarantee that all messages are delivered to all members of a process group."

Basic Reliable-Multicasting Schemes

A simple solution to reliable multicasting when all receivers are known and are assumed not to fail.

- The sending process assigns a sequence number to each message it multicasts.
- Assume that messages are received in the order they are sent.
- Each multicast message is stored locally in a history buffer at the sender.
- Assuming the receivers are known to the sender, the sender simply keeps the message in its history buffer until each receiver has returned an acknowledgment.
- If a receiver detects it is missing a message, it may return a negative acknowledgment, requesting the sender for a retransmission.



(a) Message transmission – note that the third receiver is expecting 24.

(b) Reporting feedback – the third receiver informs the sender.

<u>Distributed Commit</u>

Objectives

- Introduction of Distributed Commit
- Distributed Commit Protocol Phases.

Distributed Commit

"The distributed commit problem involves having an operation being performed by each member of a process group, or none at all."

- In the case of reliable multicasting, the operation is the delivery of a message
- With distributed transactions, the operation may be the commit of a transaction at a single site that takes part in the transaction.
- Other examples of distributed commit, and how it can be solved are discussed in Tanisch (2000).
- Commit protocol is distributed into three types:
    - Single-phase commit
    - Two-phase commit
    - Three-phase commit.

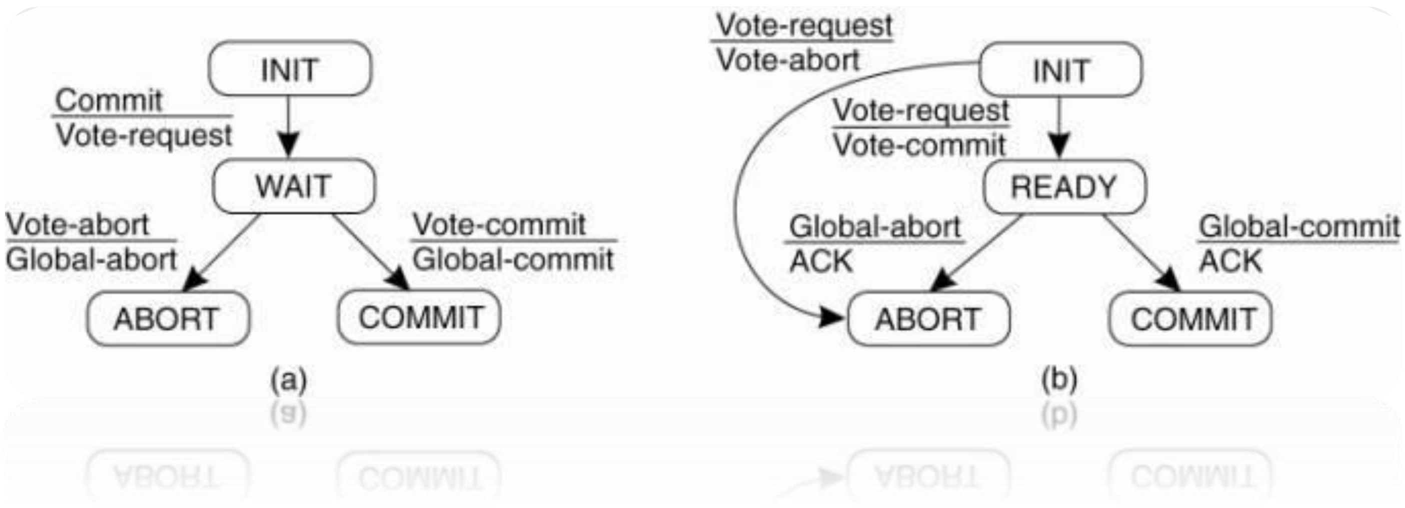
One-Phase Commit Protocol:

- Coordinator tells all other processes that are also involved, called participants, whether to (locally) perform the operation in question
- If one of the participants cannot perform the operation, there is no way to tell the coordinator
- It cannot efficiently handle the failure of the coordinator.
    - The solutions:
      Two-Phase and Three-Phase Commit Protocols

Two-Phase Commit Protocol

"Assuming that no failures occur, the protocol consists of the following two phases, each consisting of two steps: The first phase is the voting phase, and the second phase is the decision phase."

- The coordinator sends a VOTE_REQUEST message to all participants.
- A group member returns VOTE_COMMIT if it can commit locally, otherwise VOTE_ABORT message.
- All votes are collected by the coordinator.
    - A GLOBAL_COMMIT is sent if all the group members voted to commit.
    - If one group member voted to abort, a GLOBAL_ABORT is sent.
- Group members then COMMIT or ABORT based on the last message received from the coordinator



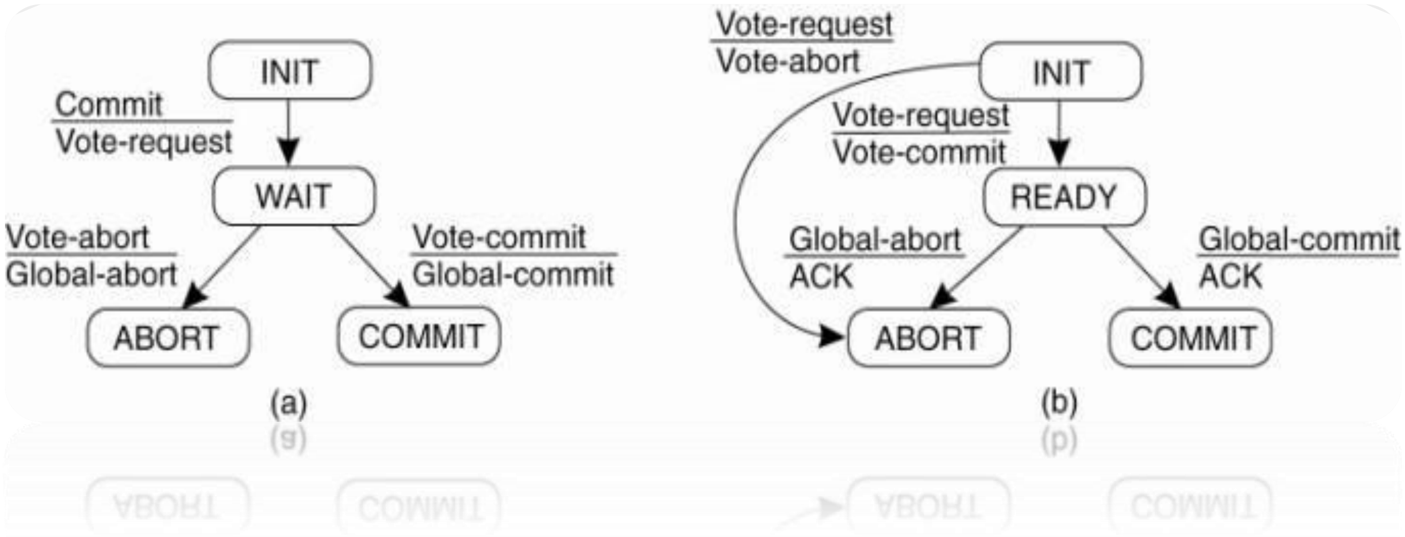
(a) The finite state machine for the coordinator in 2PC.

(b) The finite state machine for a participant.

- It can lead to both the coordinator and the participants blocking, which may lead to the dreaded deadlock.
- If the coordinator crashes, the participants may not be able to reach a final decision, and they may, therefore, block until the coordinator recovers.
- Two-Phase Commit is known as a blocking-commit protocol for this reason.
- The solution: Three-Phase Commit Protocol

Three-Phase Commit Protocol

- Skeen (1981) developed a variant of 2PC, called the three-phase commit protocol (3PC), that avoids blocking processes in the presence of fail-stop crashes.
- The states of the coordinator and each participant satisfy the following two conditions:
    - There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
    - There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made.


INIT
Commit / Vote-request
WAIT
Vote-abort / Global-abort
Vote-commit / Global-commit
ABORT
COMMIT
(a)
Vote-request / Vote-abort
INIT
Vote-request / Vote-commit
READY
Global-abort / ACK
Global-commit / ACK
ABORT
COMMIT
(b)

(a) The finite state machine for the coordinator in 3PC.

(b) The finite state machine for a participant.

Recovery

Objectives

- Basic Concept of Recovery
- Types of Recovery

Recovery

“The whole idea of error recovery is to replace an erroneous state with an error-free state. Once a failure has occurred, it is essential that the process where the failure happened recovers to a correct state.”

- Recovery from an error is fundamental to fault tolerance. Two main forms of recovery are:
    - **Backward Recovery:** Return the system to some previous correct state (using checkpoints), then continue executing.
    - **Forward Recovery:** When the system has entered an erroneous state, instead of moving back to a previous, checkpointed state, an attempt is made to bring the system in a correct new state from which it can continue to execute

Backward Recovery

- Advantages:
    - Generally applicable method independent of any specific system or process.

    - It can be integrated into (the middleware layer) of a distributed system as a general-purpose service.
- Disadvantages:
    - Restoring a system or process to a previous state is generally a relatively costly operation in terms of performance.
    - Backward error recovery mechanisms are independent of the distributed application for which they are actually used, no guarantees can be given that once recovery has taken place, the same or similar failure will not happen again.

Forward Recovery

- Advantages:
    - Generally, have low overhead.
- Disadvantages:
    - It has to be known in advance which errors may occur. Only in that case is it possible to correct those errors and move to a new state.
    - When an error occurs, the recovery mechanism then knows what to do to bring the system forward to a correct state.

Week 5

Introduction to Load Balancing

Objectives

- Introduction of Load Balancing.
- Issues in Load Balancing

Introduction to Load Balancing

"The goal of partitioning is to distribute the computation load to each processing element such that all processing elements become neither overloaded nor idle and all processors can finish their computation at about the same time."

- Distributed computing system provides high performance environment that are able to provide high processing power
- Deals with distribution of processes among processors connected by a network
- For homogeneous parallel systems, the computation load is distributed as evenly as possible in a parallel computer.
- For heterogeneous parallel system, the computation load is distributed according to the computing power of each processor.

Week 5

Introduction to Load Balancing

Objectives

- Introduction of Load Balancing.
- Issues in Load Balancing

Introduction to Load Balancing

"The goal of partitioning is to distribute the computation load to each processing element such that all processing elements become neither overloaded nor idle and all processors can finish their computation at about the same time."

- Distributed computing system provides high performance environment that are able to provide high processing power.
- Deals with distribution of processes among processors connected by a network.
- For homogeneous parallel systems, the computation load is distributed as evenly as possible in a parallel computer.
- For heterogeneous parallel system, the computation load is distributed according to the computing power of each processor.

Issues in Load Balancing

A load balancing strategy needs to resolve following issues:

- What load information (measurements) can be used.
- When to invoke balancing, i.e., conditions to balance.
- Which nodes make the load balancing decision.
- How should old data be handled.
- How load migrations are to be managed.

Mapping Techniques for Load Balancing

Objectives

- Understanding of Mapping Techniques for Load Balancing.
- The need of Mapping Techniques for Load Balancing

Mapping Techniques for Load Balancing

- The computation domain is partitioned into several subdomains and then mapped onto processors of a parallel system with the objective that all tasks complete in the shortest amount of elapsed time.
- In general, the number of subdomains equals to the number of processors in a parallel system.
- In order to achieve a small execution time, the overheads of executing the tasks in parallel must be minimized.

Quality of Load Balancing Algorithm

- The quality of load balancing algorithms can be measured by two factor:
  - Number of steps: Needed to get the balanced state.
  - Extent of loads: Moves over the link to which nodes are connected.

Mapping Techniques for Load Balancing

- There are two key sources of overhead. The time spent in inter-process interaction is one source of overhead. Another important source of overhead is the time that some processes may spend being idle.
- A good mapping of tasks onto processes must strive to achieve the twin objectives:
  - Reducing the amount of time processes spend in interacting with each other
  - Reducing the total amount of time some processes are idle while the others are engaged in performing some tasks.
- Mapping techniques used in parallel algorithms can be broadly classified into two categories
  - Static Mapping
  - Dynamic Mapping.
- The parallel programming paradigm and the characteristics of tasks and the interactions among them determine whether a static or a dynamic mapping is more suitable.

Static Mapping for Load Balancing

Objectives

- Understanding of Static Mapping for Load Balancing.
- Advantages
- Disadvantages

Static Mapping for Load Balancing

"Static mapping techniques distribute the tasks among processes prior to the execution of the algorithm."

- Processes are assigned to the processors at compile time.
- Once the process are assigned, no change or reassignment is possible at run time.
- Need a good estimate of task sizes.
- Number of jobs on each node is fix.
- Scheduling decisions are made probabilistically.

Advantages of Static Mapping

- Algorithms that make use of static mapping are in general easier to design and program
- Since the mapping is fixed, there is no need for communication between processing nodes to determine task allocation.

  This reduces communication overhead and can improve performance.
- Less network traffic due to load balancing related messages

Disadvantages of Static Mapping

- It is very difficult to compute a-priori execution time.
- Lack of Fault Tolerance: Static mapping does not account for node failures, which can result in system downtime if a node fails.
- The process allocation cannot be changed during execution.

Schemes for Static Mapping

Objectives

- Mappings Based on Data Partitioning.
- Mappings Based on Task Partitioning
- Hierarchical Mappings

Scheme for Static Mapping

- Static mapping is often used in conjunction with a decomposition based on data partitioning.
- Static mapping is also used for mapping certain problems that are expressed naturally by a static task-dependency graph. Mapping schemes based on data partitioning and task partitioning are:
    - Mappings Based on Data Partitioning
    - Mappings Based on Task Partitioning
    - Hierarchical Partitioning

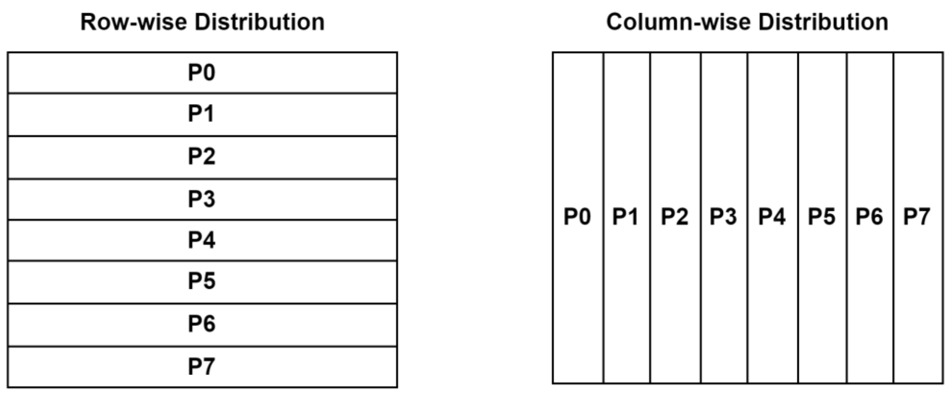
Mappings Based on Data Partitioning

Mappings based on data partitioning's two of the most common ways of representing data in algorithms, namely, arrays and graphs.

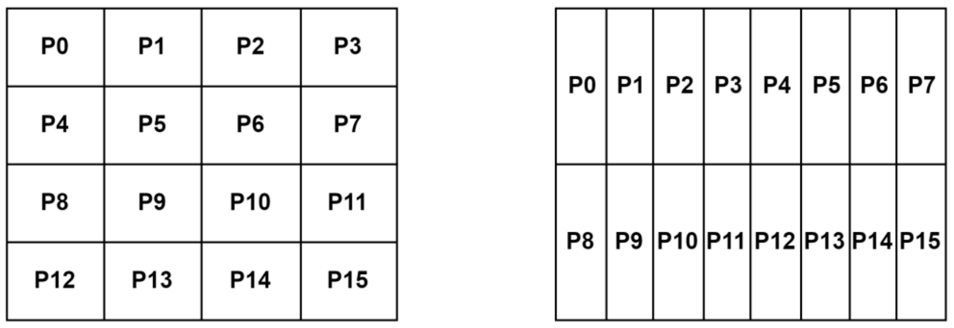
- Block Distribution Schemes
    - Block Array Distributions
    - Cyclic and Block-Cyclic Distributions
    - Randomized Block Distributions

Mappings Based on Data Partitioning: Block Array Distribution

The data partitioning can be combined with the "owner-computes" rule to partition the computation into subtasks.
The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.
In general, higher dimension decomposition allows the use of higher number of processes.



Examples of one-dimensional partitioning of an array among eight processes.



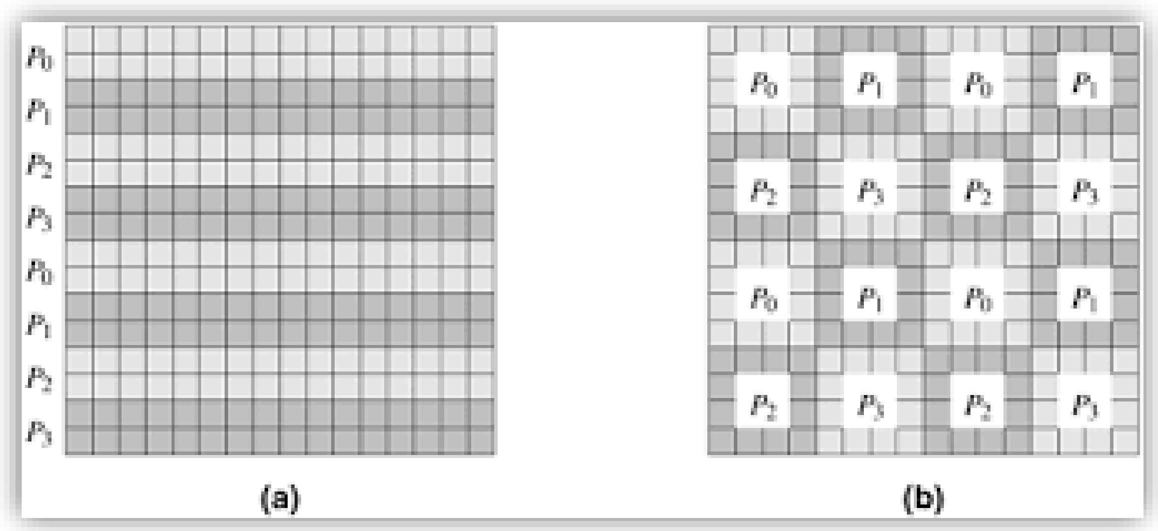
Examples of two-dimensional distributions of an array, (a) on a 4 x 4 process grid, and (b) on a 2 x 8 process grid.

Mappings Based on Data Partitioning: Cyclic and Block-Cyclic Distributions

The central idea behind a block-cyclic distribution is to partition an array into many more blocks than the number of available processes.

Then we assign the partitions (and the associated tasks) to processes in a round-robin manner so that each process gets several non-adjacent blocks.

More precisely, in a one-dimensional block-cyclic distribution of a matrix among p processes, the rows (columns) of an n x n matrix are divided into ap groups of n/(ap).



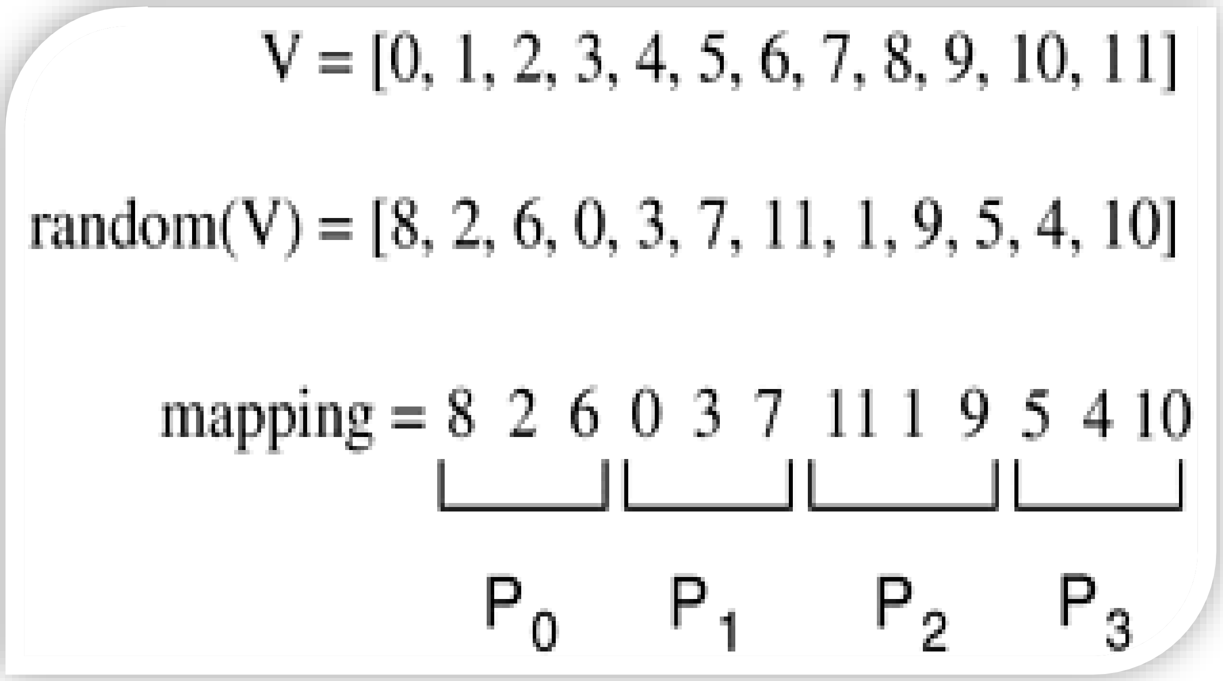
Examples of one- and two-dimensional block-cyclic distributions among four processes. (a) The rows of the array are grouped into blocks each consisting of two rows, resulting in eight blocks of rows. These blocks are distributed to four processes in a wraparound fashion. (b) The matrix is blocked into 16 blocks each of size 4 x 4, and it is mapped onto a 2 x 2 grid of processes in a wraparound fashion.

Mappings Based on Data Partitioning: Randomized Block Distribution

Just like a block-cyclic distribution, load balance is sought by partitioning the array into many more blocks than the number of available processes.

However, the blocks are uniformly and randomly distributed among the processes.

A one-dimensional randomized block mapping of 12 blocks onto four process (i.e., a = 3) is shown figure.

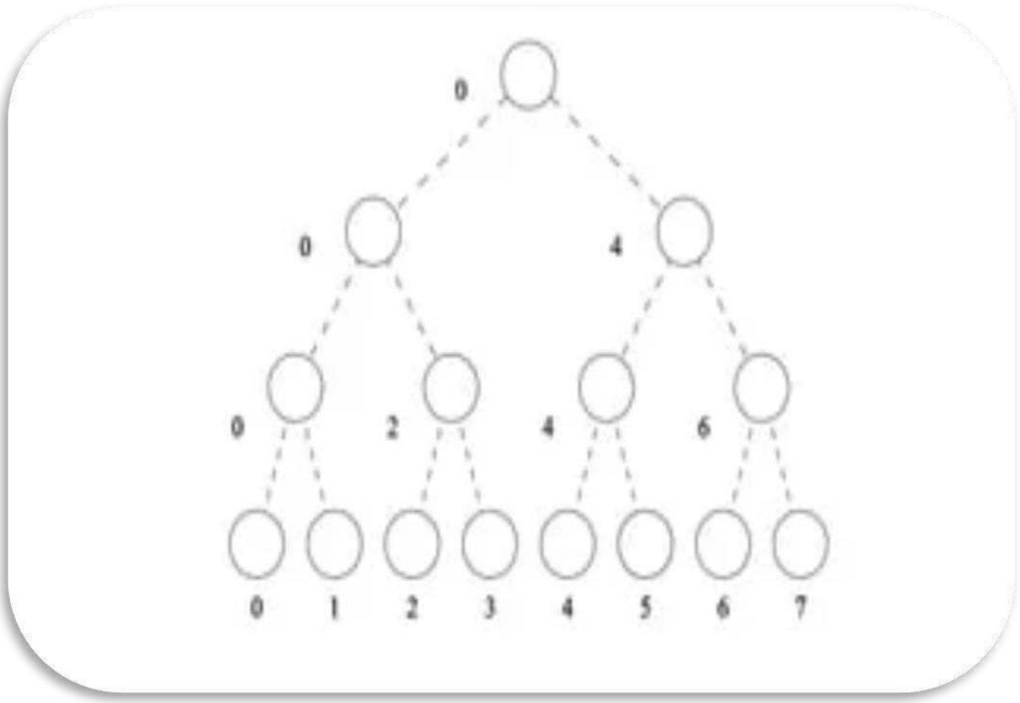

V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
random(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]
mapping = 8 2 6 0 3 7 11 1 9 5 4 10
P0
P1
P2
P3

Mappings Based on Task Partitioning

- Partitioning a given task-dependency graph across processes.
- Determining an optimal mapping for a general task-dependency graph is an NP-complete problem.
- Excellent heuristics exist for structured graphs

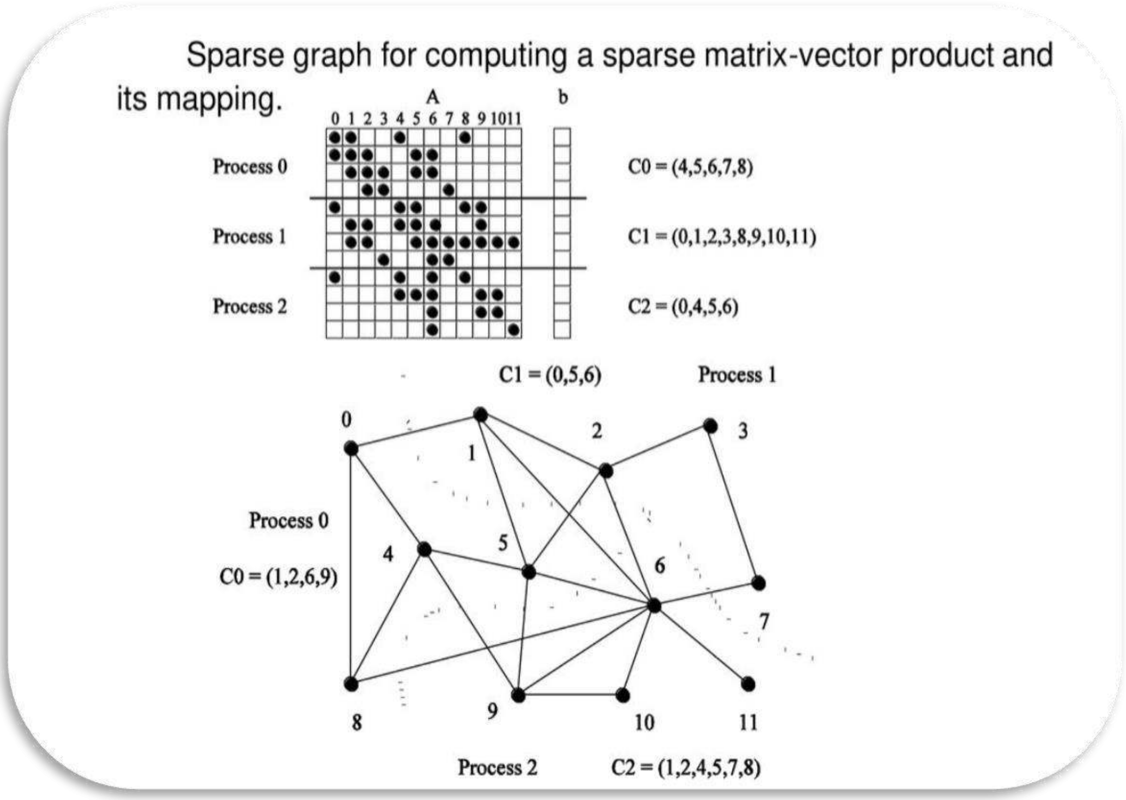
Mappings Based on Task Partitioning: Mapping a Binary Tree Dependency Graph

Example illustrates the dependency graph of one view of quick-sort and how it can be assigned to processes in a hypercube.



Mappings Based on Task Partitioning: Mapping a Sparse Graph

Reducing interaction overhead in sparse matrix-vector multiplication by partitioning the task-interaction graph.

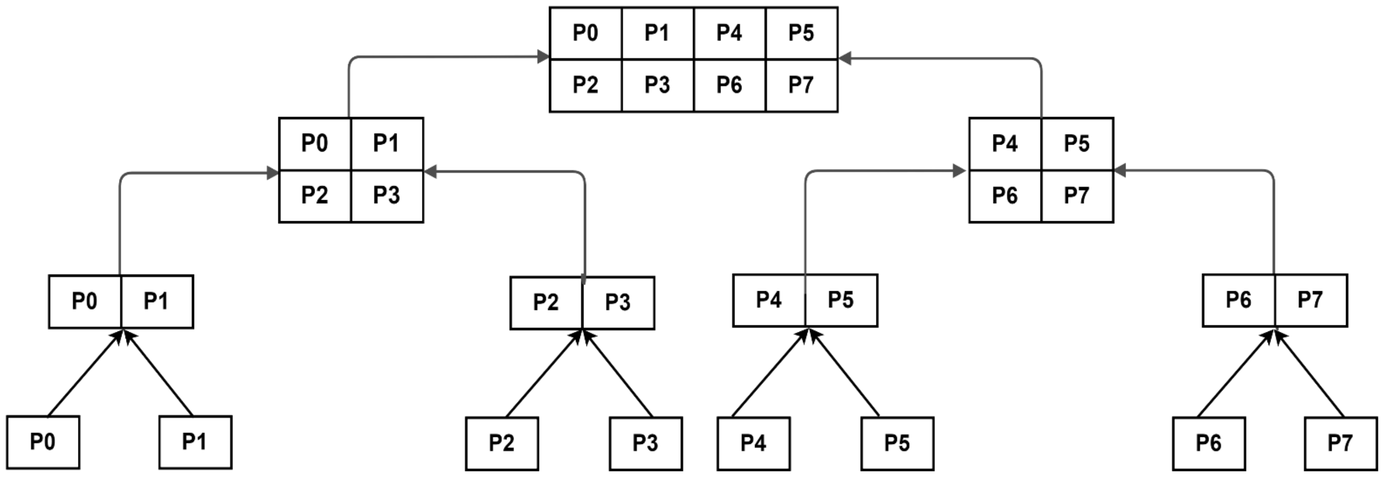


Hierarchical Mappings

A hierarchical mapping can have many layers and different decomposition and mapping techniques may be suitable for different layers."

- If the tasks are large enough, then a better mapping can be obtained by a further decomposition of the tasks into smaller subtasks.

- A single mapping is inadequate.
- Task mapping at the top layer and Data partitioning within each level
- Static mapping based on hierarchical mappings can be used in a variety of applications, including scientific simulations, machine learning, and image processing.



An example of hierarchical mapping of a task-dependency graph. Each node represented by an array is a supertask. The partitioning of the arrays represents subtasks, which are mapped onto eight processes.

Dynamic Mapping for Load Balancing

Objectives

- Introduction of Dynamic Mapping
- Advantages of Dynamic Mapping
- Disadvantages of Dynamic Mapping

Dynamic Mapping for Load Balancing

"Dynamic mapping techniques distribute the work among processes
during the execution of the algorithm."

- If tasks are generated dynamically, then they must be mapped dynamically too
- If task sizes are unknown, then a static mapping can potentially lead to serious load-imbalances and dynamic mappings are usually more effective.
- If the amount of data associated with tasks is large

Advantages of Dynamic Mapping

- Greater Resource utilization.
- Enhanced Load Balancing
- The process allocation can be modified during execution if required.
- Scalability: Dynamic mapping can scale the system up or down based on the workload.

Disadvantages of Dynamic Mapping

- Algorithms that require dynamic mapping are usually more complicated, particularly in the message-passing programming paradigm.
- Greater overhead due to process redistribution.

- Lack of determinism: Dynamic mapping introduces non-determinism into the system

Schemes for Dynamic Mapping

Objectives

- Schemes for Dynamic Mapping
- Centralized Dynamic Mapping Scheme
- Distributed Dynamic Mapping Scheme.

Schemes for Dynamic Mapping

- Schemes for Dynamic Mapping
    - Centralized Schemes
        - Master Process
        - Slave Processes
    - Distributed Schemes

Centralized Dynamic Load Balancing Scheme

- All executable tasks are maintained in a common central data structure, or they are maintained by a special process. If a special process is designated to manage the pool of available tasks, then it is often referred to as the master and the other processes that depend on the master to obtain work are referred to as slaves.
- Whenever a process has no work, it takes a portion of available work from the central data structure or the master process.
- When a new task is generated, it is added to this centralized data structure or reported to the master process.
- Centralized load balancing schemes are usually easier to implement than distributed schemes.
- If number of processes increases, master may become the bottleneck.
- Chunk scheduling: A process picks up multiple tasks at once.
    - Large chunk sizes may lead to significant load imbalances as well.
    - Schemes to gradually decrease chunk size as the computation progresses.

Distributed Dynamic Load Balancing Scheme

“In a distributed dynamic load balancing scheme, the set of executable tasks are distributed
among processes which <u>exchange tasks at run time to balance work</u>. Each process can send
work to or receive work from any other process.”

- Alleviates the bottleneck in centralized schemes. Critical parameters of a distributed load balancing scheme are as follows:
    - How are the sending and receiving processes paired together?
    - Is the work transfer initiated by the sender or the receiver?
    - How much work is transferred in each exchange?
    - When is the work transfer performed?

Week 6

Concurrency Control

Objectives

- What is Concurrency?
- Mechanisms for Concurrency Control

Concurrency Control

"Concurrency is the task of running two or more computations over the same time interval. Two events are said to be concurrent if they occur within the same time interval."

What is Concurrency?

- Concurrent doesn't necessarily mean at the same exact instant. For example, two tasks may occur concurrently within the same second but with each task executing within different fractions of the second.
- Concurrent tasks can execute in a single or multiprocessing environment
    - In a single processing environment, concurrent tasks exist at the same time and execute within the same time period by context switching.
    - In a multiprocessor environment, if enough processors are free, concurrent tasks may execute at the same instant over the same time period. The determining factor for what makes an acceptable time period for concurrency is relative to the application.

Concurrency Control

- There must be an implicit or explicit control over concurrency. It is both hazardous and unsafe when multiple flows of executions simultaneously operate in the same address space without any kind of agreement on ordered access. Two or more activities might access the same data and thus induce data corruption as well as inconsistent or invalid application state.
- Multiple activities that work jointly on a problem need an agreement on their common progress. Both issues represent fundamental challenges of concurrency and concurrent programming.

Synchronization and Coordination are two basic approaches to tackle this challenge:

- Synchronization is a mechanism that controls access on shared resources between multiple activities. It enforces exclusiveness and ordered access on the resource by different activities.
- Coordination aims at the orchestration of collaborating activities.

Benefits of Concurrency

- The overall time to perform the series of tasks is reduced.
- Concurrent processes can reduce duplication in code.
- The overall runtime of the algorithm can be significantly reduced.
- Concurrency control can also increase the scalability of parallel and distributed computing systems
- Redundancy can make systems more reliable.
- More real-world problems can be solved than with sequential algorithms alone

Concurrency Control: Basic Approaches to Achieving Concurrency

Objectives

- Understanding of Concurrency Control.
- Parallel Programming Technique
- Distributed Programming Technique

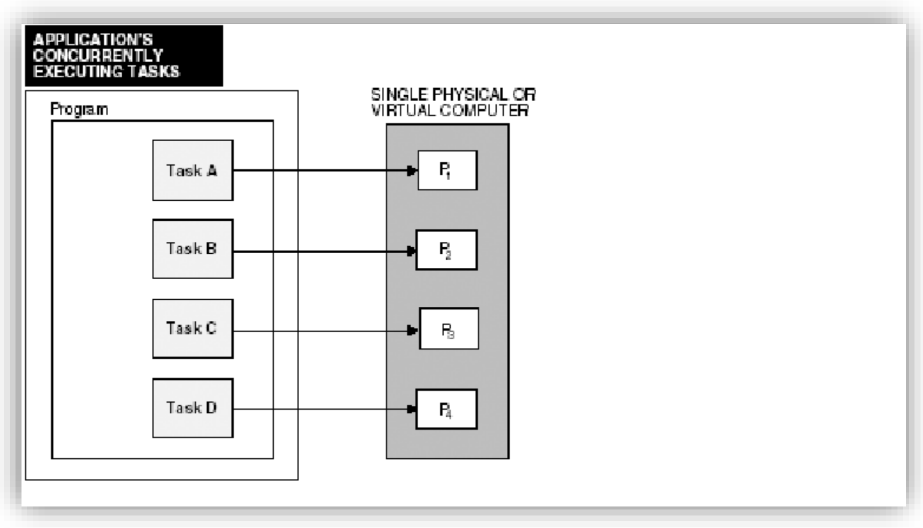
Achieving Concurrency

Parallel programming and distributed programming are two basic approaches for achieving concurrency:

- Parallel programming techniques assign the work a program has do to two or more processors within a single physical or a single virtual computer.
- Distributed programming techniques assign the work a program has to do to two or more processes where the processes may or may not exist on the same computer.

Achieving Concurrency: Parallel Programming Technique

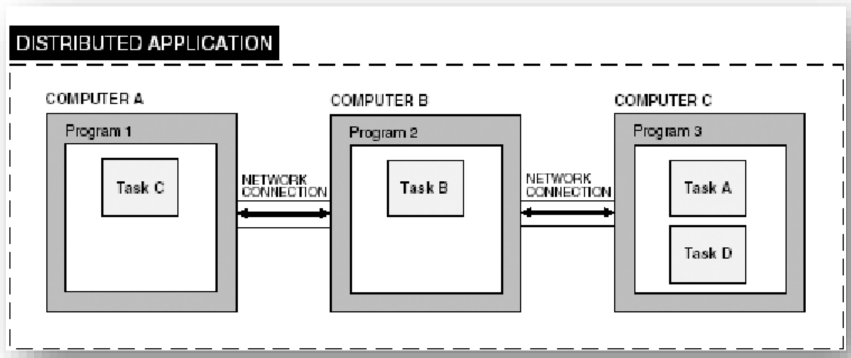
- The parallel application consists of one program divided into four tasks. Each task executes on a separate processor; therefore, each task may execute simultaneously. The tasks can be implemented by either a process or a thread.


APPLICATION'S CONCURRENTLY EXECUTING TASKS
Program
Task A
Task B
Task C
Task D
SINGLE PHYSICAL OR VIRTUAL COMPUTER
P_1
P_2
P_3
P_4

Typical architecture for a parallel program.

Achieving Concurrency: Distributed Programming Technique

- The distributed application consists of three separate programs with each program executing on a separate computer. Program 3 consists of two separate parts that execute on the same computer. Although task A and D of Program 3 are on the same computer, they are distributed because they are implemented by two separate processes.



Typical architecture for a parallel and distributed program.

Concurrency Control: Models for Programming Concurrency

Objectives

- Models of Programming Concurrency.
- Van Roy Approaches for Programming Concurrency.

Concurrency Control: Models for Programming Concurrency

Van Roy introduces four main approaches for programming concurrency:

- Sequential Programming.
- Declarative Concurrency.
- Message-passing Concurrency.
- Shared-state Concurrency.

Sequential Programming

In this deterministic programming model, no concurrency is used at all. In its strongest form, there is a total order of all operations of the program. Weaker forms still keep the deterministic behavior. However, they either make no guarantees on the exact execution order to the programmer a priori. Or they provide mechanisms for explicit preemption of the task currently active, as co-routines do, for instance.

Declarative Concurrency

Declarative programming is a programming model that favors implicit control flow of computations. Control flow is not described directly, it is rather a result of computational logic of the program. The declarative concurrency model extends the declarative programming model by allowing multiple flows of executions. It adds implicit concurrency that is based on a data-driven or a demand-driven approach. While this introduces some form of nondeterminism at runtime, the nondeterminism is generally not observable from the outside.

Message-passing Concurrency

This model is a programming style that allows concurrent activities to communicate via messages. Generally, this is the only allowed form of interaction between activities which are otherwise completely isolated. Message passing can be either synchronous or asynchronous resulting in different mechanisms and patterns for synchronization and coordination

Shared-state Concurrency

Shared-state concurrency is an extended programming model where multiple activities are allowed to access contended resources and states. Sharing the exact same resources and states among different activities requires dedicated mechanisms for synchronization of access and coordination between activities. The general nondeterminism and missing invariants of this model would otherwise directly cause problems regarding consistency and state validity.

Memory Hierarchies

Objectives

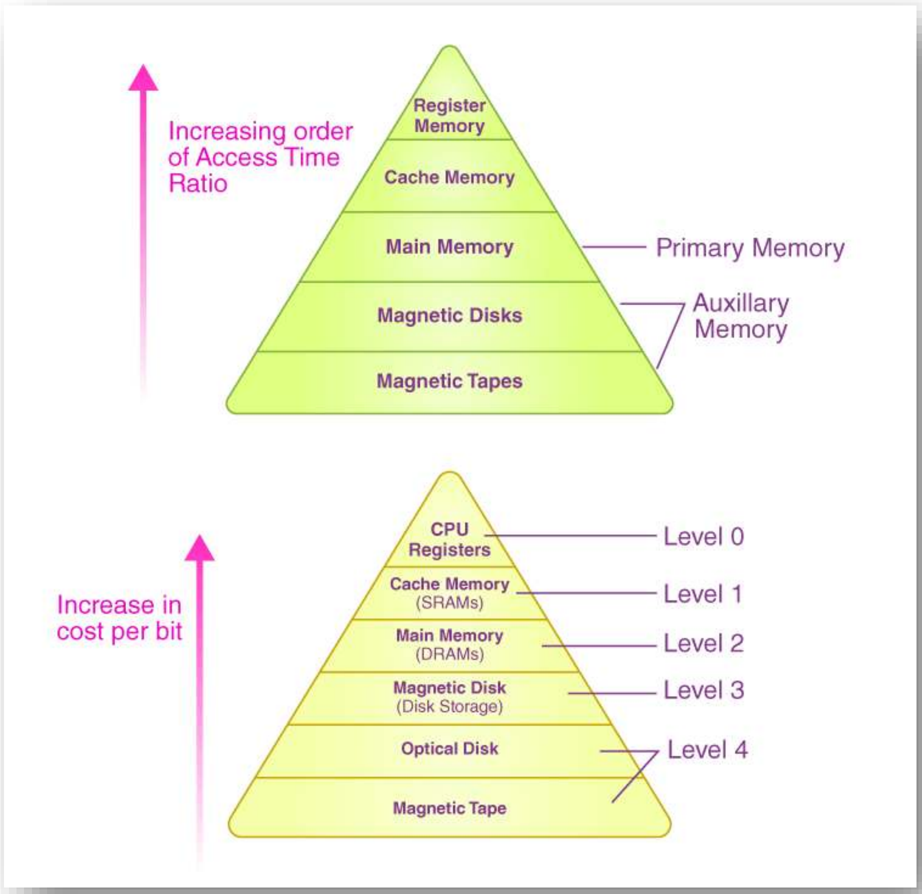
- Introduction of Memory Hierarchy
- Characteristics of Memory Hierarchy

Memory Hierarchies

"Memory Hierarchy, in Computer System Design, is an enhancement that helps in organizing the memory so that it can minimize the access time. The development of the Memory Hierarchy occurred on a behavior of a program known as locality of references."

We are concerned with five types of memory:

- **Registers:** are the fastest type of memory, which are located internal to a processor. These elements are primarily used for temporary storage of operands, small partitions of memory, etc., and are assumed to be one word (32 bits) in length in the MIPS architecture.
- **Cache:** is a very fast type of memory that can be external or internal to a given processor. Cache is used for temporary storage of blocks of data obtained from main memory (read operation) or created by the processor and eventually written to main memory (write operation).
- **Main Memory:** is modelled as a large, linear array of storage elements that is partitioned into static and dynamic storage. Main memory is used primarily for storage of data that a program produces during its execution, as well as for instruction storage.
- **Disk Storage:** is much slower than main memory, but also has much higher capacity than the preceding three types of memory.
- **Archival Storage:** is offline storage such as a CD-ROM jukebox or (in former years) rooms filled with racks containing magnetic tapes. This type of storage has a very long access time, in comparison with disk storage, and is also designed to be much less volatile than disk data.



Characteristics of Memory Hierarchy

Characteristics of a Memory Hierarchy can be inferred from the previous figure:

- **Capacity:** It refers to the total volume of data that a system's memory can store. The capacity increases moving from the top to the bottom in the Memory Hierarchy.
- **Access Time:** It refers to the time interval present between the request for read/write and the data availability. The access time increases as we move from the top to the bottom in the Memory Hierarchy.

Characteristics of a Memory Hierarchy can be inferred from the previous figure:

- **Performance:** When a computer system was designed earlier without the Memory Hierarchy Design, the gap in speed increased between the given CPU registers and the Main Memory due to a large difference in the system's access time. It ultimately resulted in the system's lower performance, and thus, enhancement was required. Such a kind of enhancement was introduced in the form of Memory Hierarchy Design, and because of this, the system's performance increased. One of the primary ways to increase the performance of a system is minimizing how much a memory hierarchy has to be done to manipulate data.
- **Cost per bit:** The cost per bit increases as one moves from the bottom to the top in the Memory Hierarchy, i.e. External Memory is cheaper than Internal Memory

Limitations of Memory System Performance

Objectives

- Understanding of Limitations of Memory System Performance.
- Memory Latency Example.

Limitations of Memory System Performance

- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters, latency and bandwidth.
    - **Latency:** Is the time from the issue of a memory request to the time the data is available at the processor.
    - **Bandwidth:** Is the rate at which data can be pumped to the processor by the memory system.
- It is very important to understand the difference between latency and bandwidth.
- Consider the example of a fire-hose. If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.
    - Once the water starts flowing, if the hydrant delivers water at the rate of 5 gallons/second, the bandwidth of the system is 5 gallons/second.
    - If you want immediate response from the hydrant, it is important to reduce latency.
    - If you want to fight big fires, you want high bandwidth.

Memory Latency Example

Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The following observations follow:

- The peak processor rating is 4 GFLOPS.
- Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data.

Improving Effective Memory

Latency Using Caches

Objectives

- Effect of Cache.
- Effect of Cache with Example

Improving Effective Memory Latency Using Caches

- Caches are small and fast memory elements between the processor and DRAM.
- This memory acts as a low-latency high-bandwidth storage.
- If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.
- The fraction of data references satisfied by the cache is called the cache hit ratio of the computation on the system.
- Cache hit ratio achieved by a code on a memory system often determines its performance.

Effect of Cache

- Repeated references to the same data item correspond to temporal locality.
- In our example, we had O(n2) data accesses and O(n3) computation. This asymptotic difference makes the above example particularly desirable for caches.
- Reduce network congestion and improve overall performance.

Effect of Cache Example

Continue the previous example of memory latency, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle. We use this setup to multiply two matrices A and B of dimensions 32 × 32(8KB or 1K words for each matrix). We have carefully chosen these numbers so that the cache is large enough to store matrices A and B, as well as the result matrix C.

- 1GHz processor, 4GFLOPS theoretical peak, 100ns memory   Latency.
- Assume 1ns cache latency (full-speed cache)

The following observations can be made about the problem:

- Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 µs.
- Multiplying two n × n matrices takes 2n3 operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16 µs) at four instructions per cycle.
- The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., 200 + 16 µs.
- This corresponds to a peak computation rate of 64K/216 or 303 MFLOPS.

Effect of Memory Bandwidth

Objectives

- Effect of Memory Bandwidth.
- Effect of Memory Bandwidth Example.

Effect of Memory Bandwidth

- Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.
- Memory bandwidth can be improved by increasing the size of memory blocks.
- The performance of the CPU or GPU can also impact memory bandwidth.

Effect of Memory Bandwidth Example

Consider the same setup as in previous topic, except in this case, the block size is 4 words instead of 1 word. We repeat the dot-product computation in this scenario:

- Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles.
- This is because a single memory access fetches four consecutive words in the vector.
- Therefore, two accesses can fetch four elements of each of the vectors. This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS.

- It is important to note that increasing block size does not change latency of the system.

- Physically, the scenario illustrated here can be viewed as a wide data bus (4 words or 128 bits) connected to multiple memory banks.
- In practice, such wide buses are expensive to construct.
- In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.
- The above examples clearly illustrate how increased bandwidth results in higher peak computation rates
- The data layouts were assumed to be such that consecutive data words in memory were used by successive instructions (spatial locality of reference).
- If we take a data-layout centric view, computations must be reordered to enhance spatial locality of reference.

Week 7

Interconnection Networks

Objectives

- Introduction of Interconnection Networks
- Introduction of Static Network
- Introduction of Dynamic Network

Interconnection Networks:

"Interconnection networks provide mechanisms for data transfer between processing nodes or between processors and memory modules."

Interconnects are made of switches and links.

Provide efficient, correct, robust message passing between two separate nodes.

- Local area network (LAN) – connects nodes in single building, fast & reliable.
    - Media: twisted-pair, coax, fiber
    - Bandwidth: 10-100MB/s
- Wide area network (WAN) – connects nodes across large geographic area.
    - Media: fiber, microwave links, satellite channels
    - Bandwidth: 1.544MB/s (T1), 45 MB/s (T3)

Interconnection networks are classified into two categories:

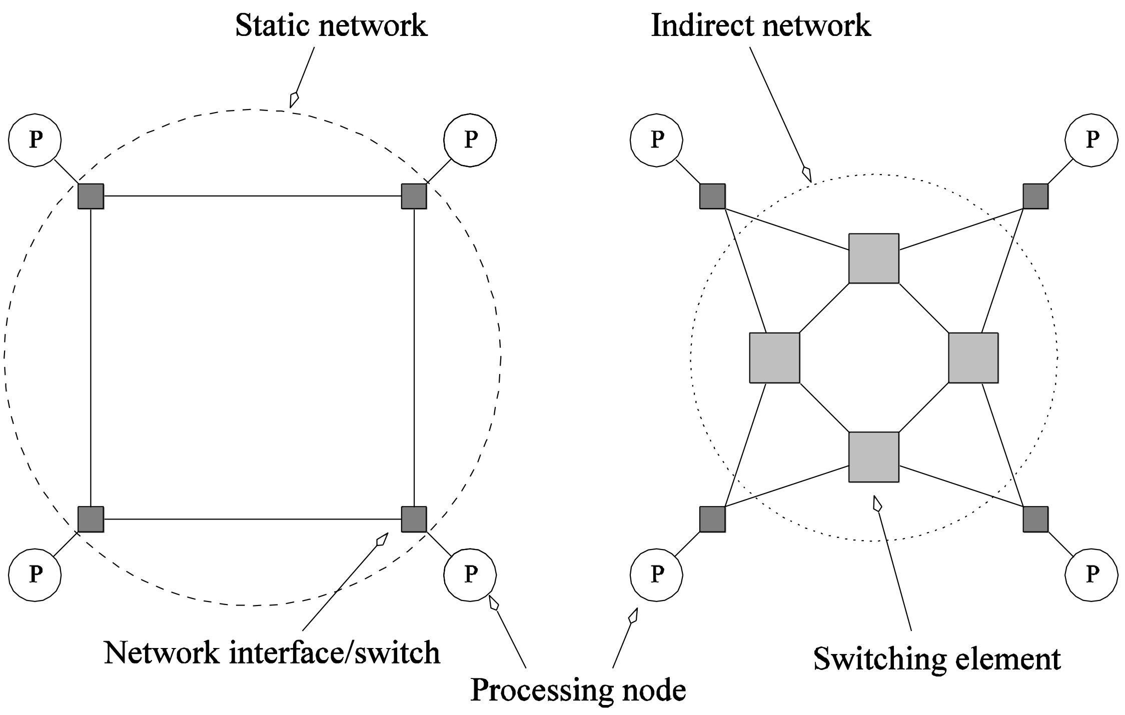
- Static network
- Dynamic network

Interconnection Networks: Static Network

- Static networks consist of point-to-point communication links among processing nodes and are also referred to as direct networks.
- Direct fixed links are established among nodes to form a fixed network.

Interconnection Networks: Dynamic Network

- Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as indirect networks.
- Connections are established when needed
- Communication links are connected to one another dynamically by the switches to establish paths among processing nodes and memory banks.



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

Control Strategy

- Objectives Understanding of Control Strategies.
- Centralized Strategy.
- Decentralized Strategy

Control Strategy

- Depending on where the decisions are made as well as on the number of measurements that are utilized to make the control decisions, these control strategies are classified into two categories:
    - Centralized
    - Decentralized

Control Strategy: Centralized

- One central control unit is used to control the operations of the components of the system
- Non autonomous components.
- Usually homogeneous technology
- Multiple users share the same resources at all time
- Single point of control
- Single point of failure

Control Strategy: Decentralized

- Set of tightly coupled programs executing on one or more computers which are interconnected through a network and coordinating their actions.
- The control function is distributed among different components in the system.
- Autonomous components
- Mostly build using heterogeneous technology
- System components may be used exclusively
- Concurrent processes can execute
- Multiple point of failure

Switching Techniques

Objectives

- Understanding of Switching Techniques.
- Types of Switching Techniques

Switching Techniques

"Switching is process to forward packets coming in from one port to a port leading towards the destination."

- Switches map a fixed number of inputs to outputs.
- The total number of ports on a switch is the degree of the switch.
- Switches may also provide support for internal buffering, routing and multicast
- The cost of a switch grows as the square of the degree of the switch, the peripheral hardware linearly as the degree, and the packaging costs linearly as the number of pins.
- Nodes may connect to other nodes only, or to stations and other nodes.
- End devices are stations:
    - Computer, terminal, phone, etc.
- Two different switching technologies are:
    - Circuit switching
    - Packet switching

Switching Techniques: Circuit switching

- A complete path has to be established prior to the start of communication between a source and a destination.
- Dedicated communication path between two stations.
- Must have switching capacity and channel capacity to establish connection
- Must have intelligence to work out routing
- Three phases are:
    - Establish
    - Transfer
    - Disconnect

Switching Techniques: Packet switching

- Communication between a source and a destination takes place via messages divided into smaller entities, called packets.
- Data transmitted in small packets.
    - Typically, 1000 octets.
    - Longer messages split into series of packets.
    - Each packet contains a portion of user data plus some control information
- Control info
    - Routing (addressing) information
- Packets are received, stored briefly (buffered) and past on to the next node.
    - Store and forward

Network Topologies

Objectives

- Introduction of Network Topologies
- Types of Network Topologies

Network Topologies:

"A network topology is the physical and logical arrangement of nodes and connections in a network. Describes how to connect processors and memories to other processors and memories"

- A variety of network topologies have been proposed and implemented.
- These topologies tradeoff performance for cost
- Commercial machines often implement hybrids of multiple topologies for reasons of packaging, cost, and available components
- Connection of nodes impacts
    - Maximum & average communication time
    - Fault tolerance
    - Expense
- Two basic types of topologies are:
    - Static topology.
    - Dynamic topology.
- Static Connection Networks:
    - Ring (Loop) networks
    - Mesh
    - Torus
    - Tree networks
    - Hypercube network
- Dynamic Connection Networks
    - Bus-based
    - Switch-based

Network Properties

- **Diameter**: The diameter of a network with n nodes is the length of the maximum shortest path between any two nodes in the network.
- **Degree of a node**: The number of connections for that node.
- **Latency:** Total time to send a message
- **Bandwidth:** Number of bits transmitted in a unit of time.
- **Bisection:** The number of connections that need to be cut to partition the network in 2

Static Topologies: Star

Objectives

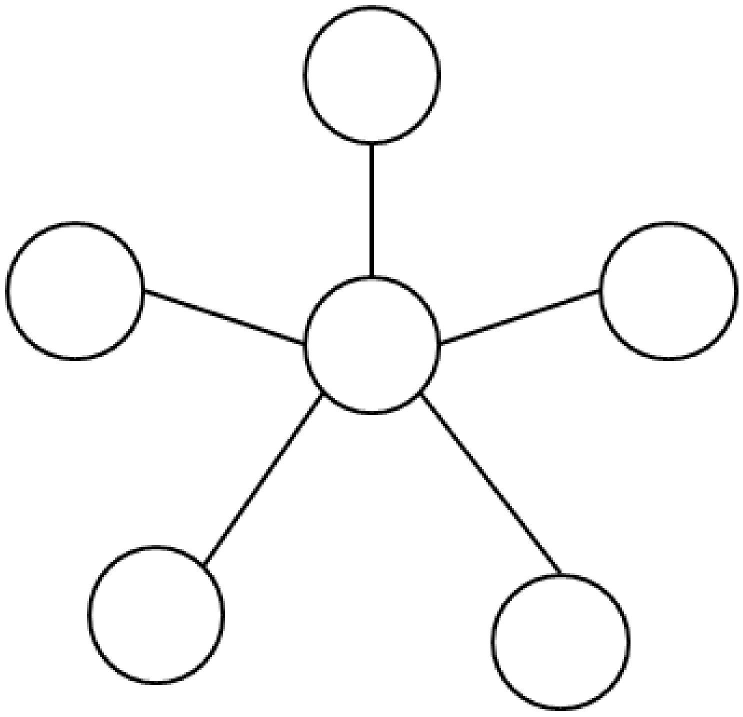
- Introduction Star Topology
- Properties of Star Topology

Static Topologies: Star

"All devices are connected to a central switch, which makes it easy to add new nodes without rebooting all currently connected devices. Logical: master-slave model."

- 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.

Inexpensive – sometimes used for LANs



A star connected network of nine nodes.

Static Topologies: Mesh

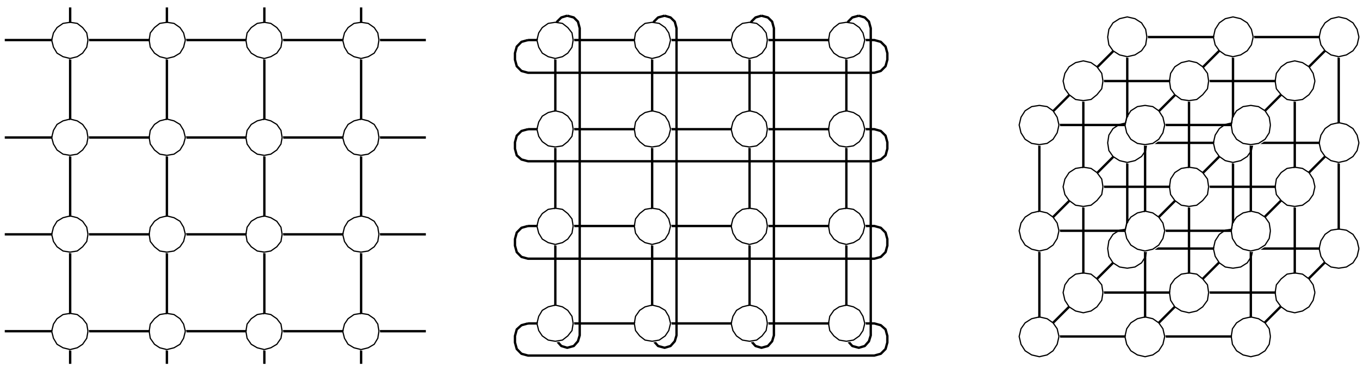
Objectives

- Introduction of Mesh Topology.
- Properties of Mesh Topology.

Static Topologies: Mesh

"Each node is connected to every other mode with a direct link. This topology creates a very reliable network but requires a large amount of cable and is difficult to administer."

- Two-dimensional mesh is an extension of the linear array to two dimensions.
- 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 2-D mesh has the property that it can be laid out in 2-D space, making
- It attractive from a wiring standpoint.



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.

Static Topologies: Hypercube

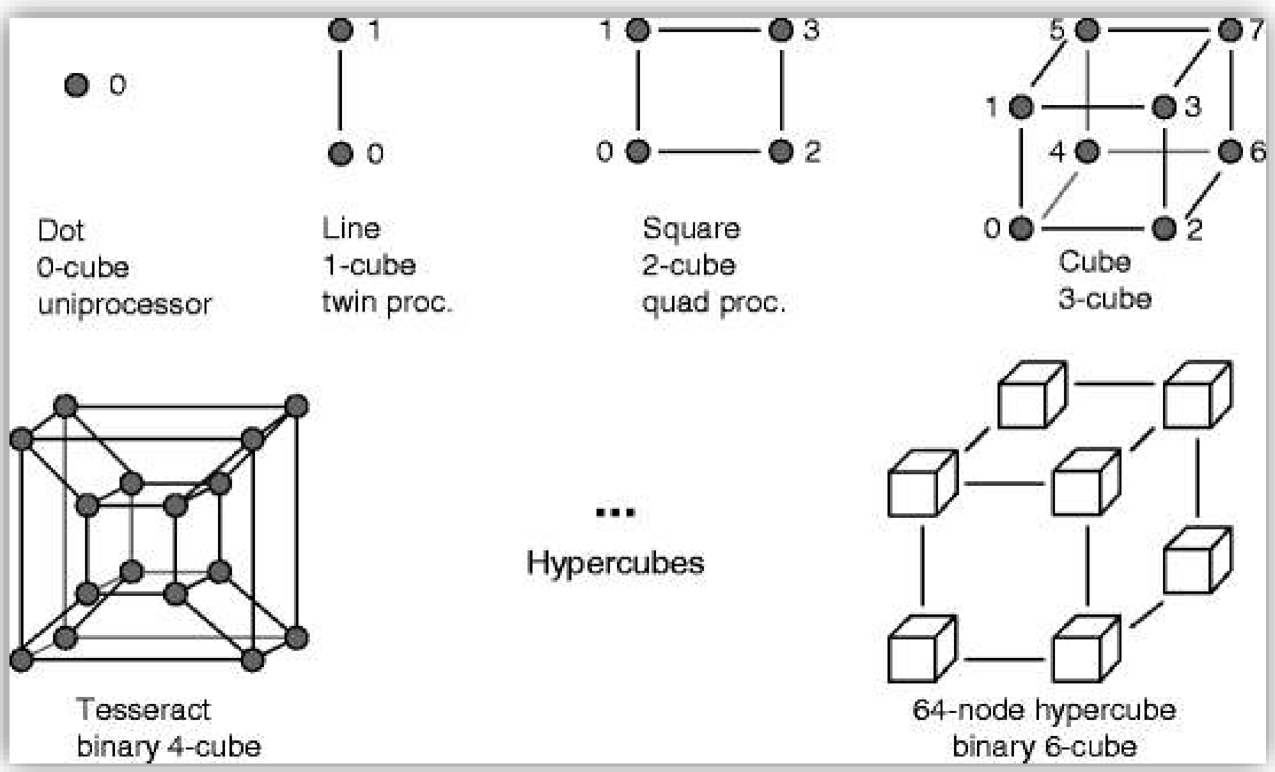
Objectives

- Introduction of Hypercube Topology.
- Properties of Hypercube Topology

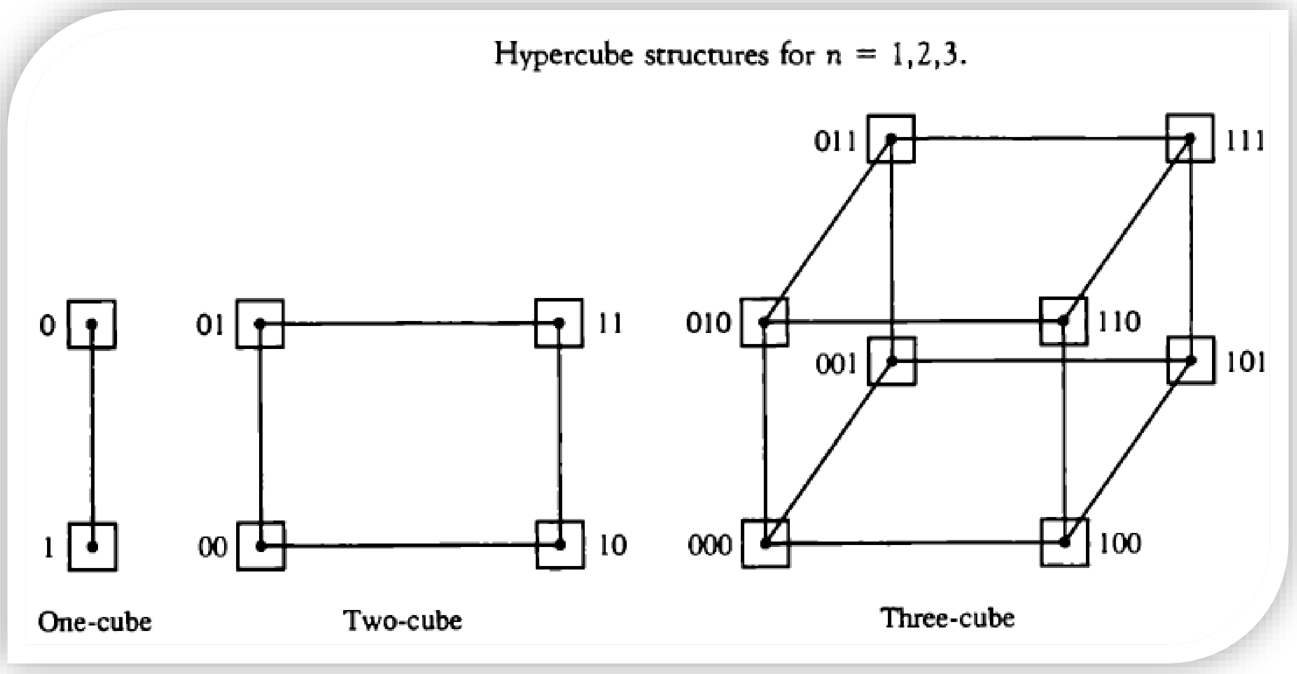
Static Topologies: Hypercube

"Hypercube (or Binary n-cube multiprocessor) structure represents a loosely coupled system made up of N=2n processors interconnected in an n-dimensional binary cube."

- A special case of a d-dimensional mesh is a hypercube. Here, d = log p, where p is the total number of nodes.
- 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
- Each node is assigned a binary address in such a manner, that the addresses of two neighbors differ in exactly one bit position.



Construction of hypercubes from hypercubes of lower dimension.



Construction of hypercubes from hypercubes of lower dimension

Static Topologies: Tree

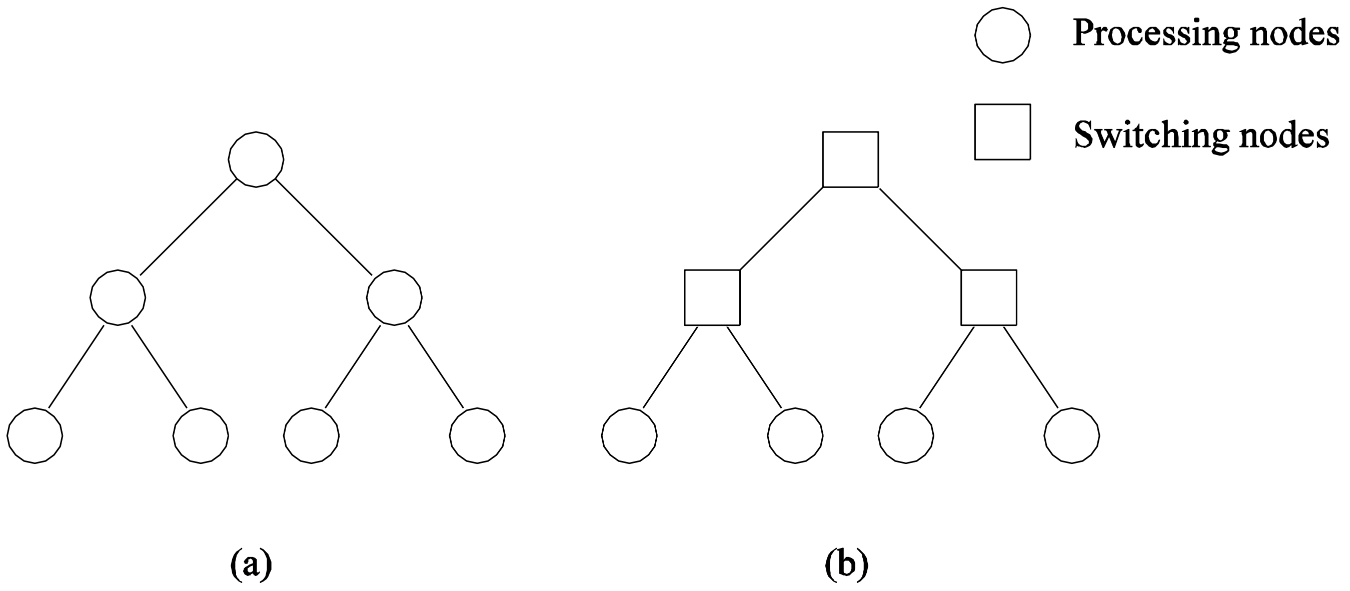
Objectives

- Introduction of Tree Topology
- Types of Tree Topology.

Static Topologies: Tree

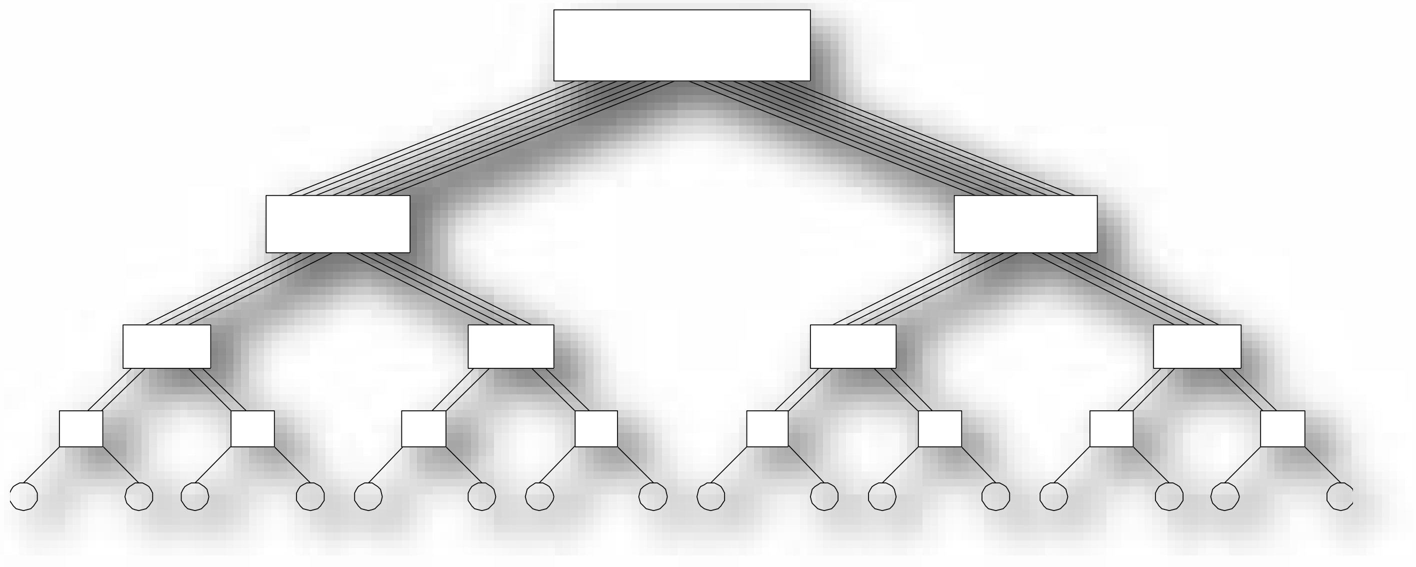
"A tree network is one in which there is only one path between any pair of nodes."

- 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
- Both linear arrays and star-connected networks are special cases of tree networks
- Trees can be laid out in 2D with no wire crossings. This is an attractive property of trees
- To route a message in a tree, the source node sends the message up the tree until it reaches the node at the root of the smallest subtree containing both the source and destination nodes.



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

Static Topologies: Fat Tree



A fat tree network of 16 processing nodes

Dynamic Topologies: Buses

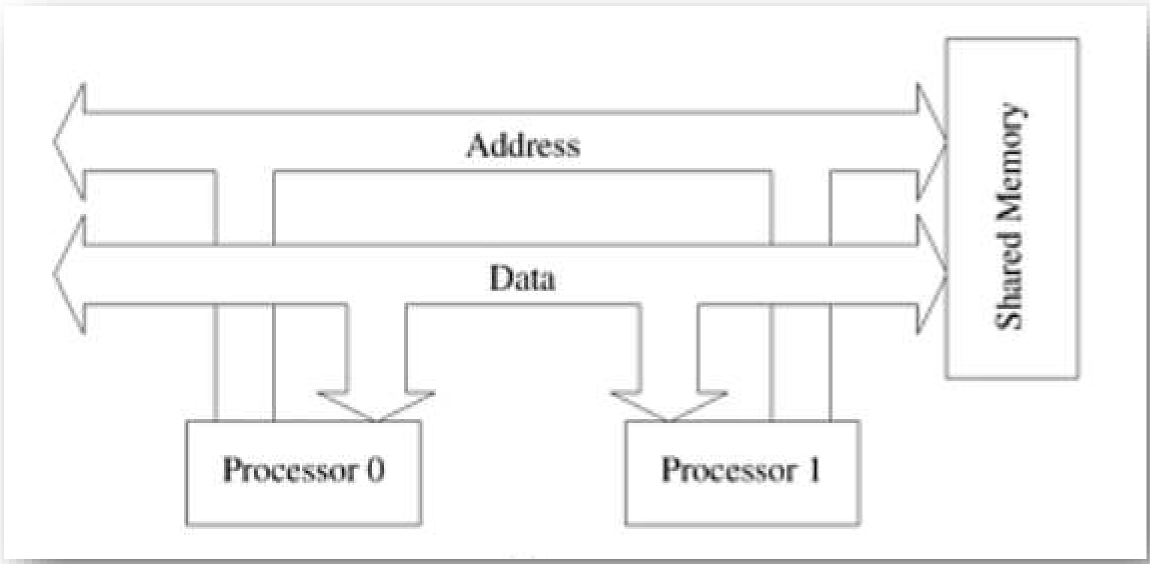
Objectives

- Introduction of Bus Topology
- Introduction of Distributed Memory.

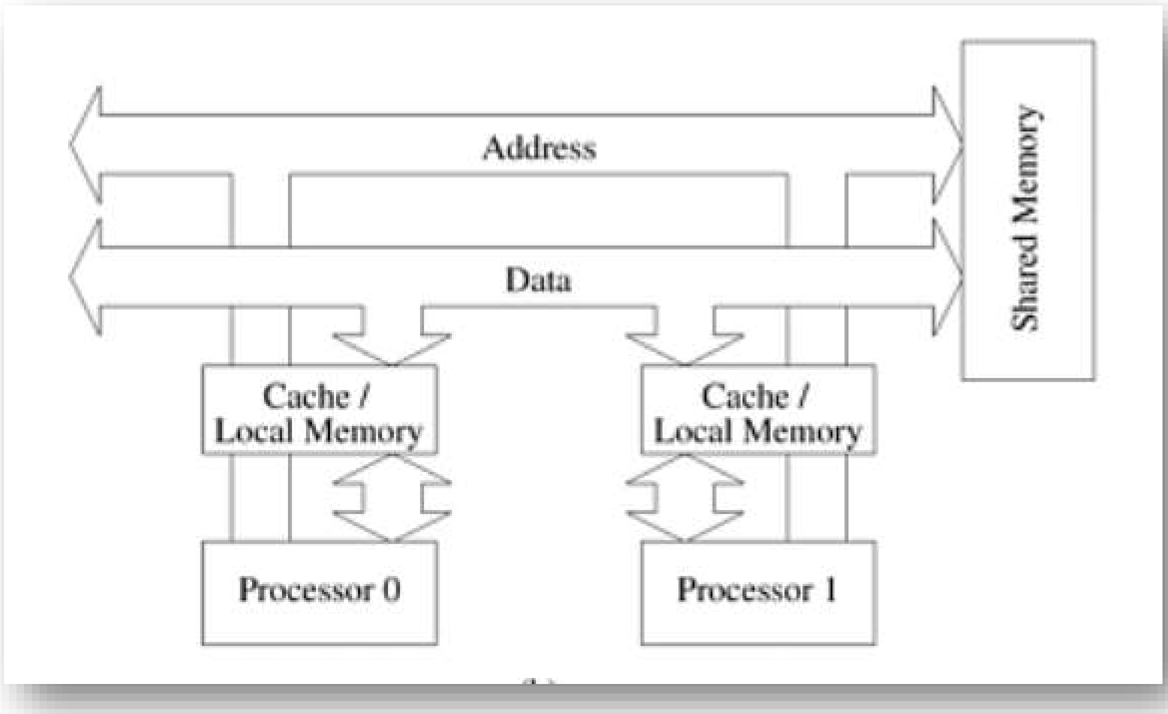
Dynamic Topologies: Bus

“Bus topology, also known as line topology, is a type of network topology in which all devices in the network are connected by one central network cable. The single cable, where all data is transmitted between devices, is referred to as the bus, backbone, or trunk.”

- Some of the simplest and earliest parallel machines used buses. All processors access a common bus for exchanging data.
- A bus has the desirable property that the cost of the network scales linearly as the number of nodes, p. This cost is typically associated with bus interfaces.
- The distance between any two nodes is *O(1)* in a bus. The bus also provides a convenient broadcast media. However, the bandwidth of the shared bus is a major bottleneck.
- Typical bus based machines are limited to dozens of nodes. Sun Enterprise servers and Intel Pentium based shared-bus multiprocessors are examples of such architectures.



Bus-based interconnects with no local caches.



Bus-based interconnects with local memory/caches.

Dynamic Topologies: Bus Cont…

Objectives

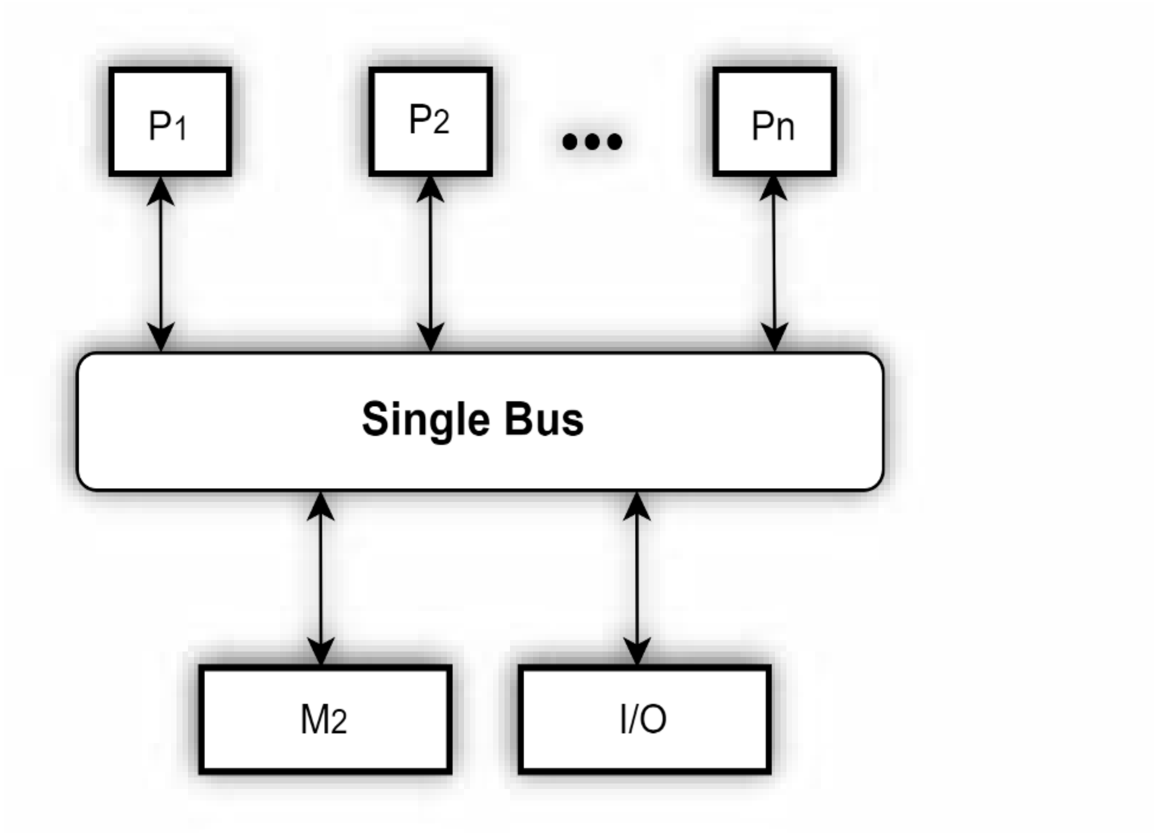
- Understanding of Single Bus System.
- Understanding of Multi-Bus System.

Dynamic Topologies: Bus

- Bus-based dynamic topologies are broadly classified into two categories:
    - Sigle bus
    - Multi bus

Dynamic Topologies: Sigle BUS

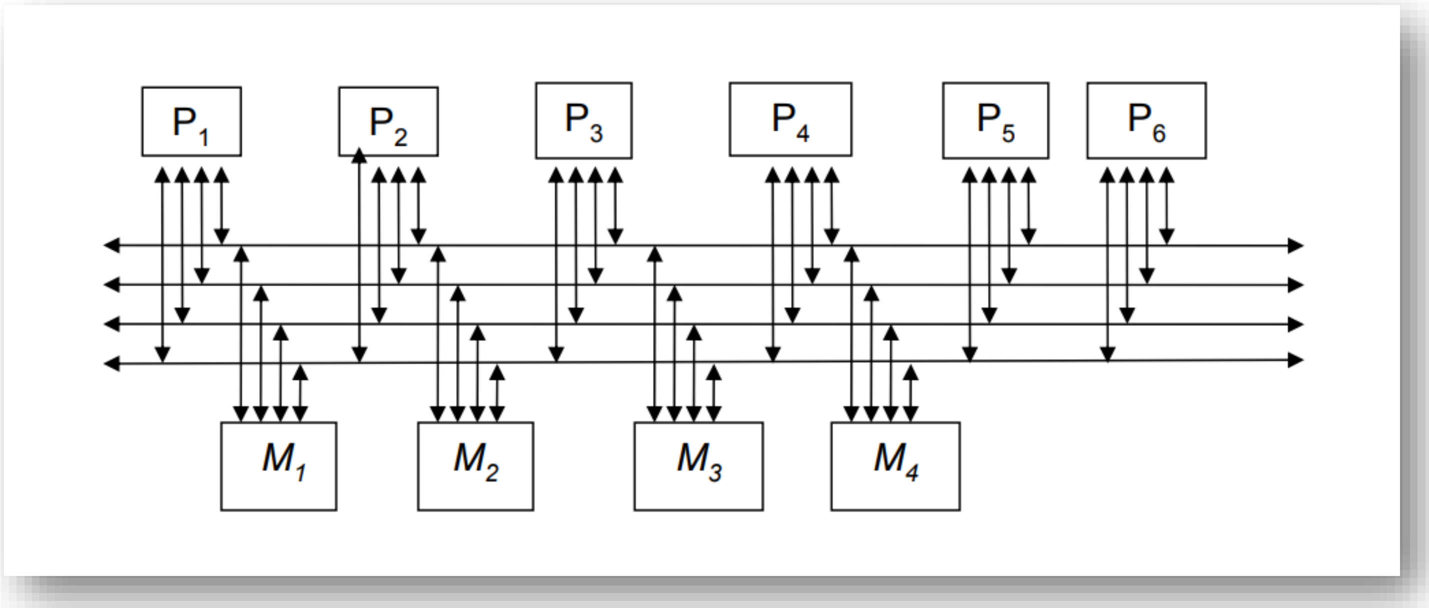
- Simplest way to connect multiprocessor systems.
- The use of local caches reduces the processor memory traffic.
- Size of such system varies between 2 and 50 processors.
- Single bus multiprocessors are inherently limited by:
    - Bandwidth of bus.
    - 1 processor can access the bus.
    - 1 memory access can take place at any given time.



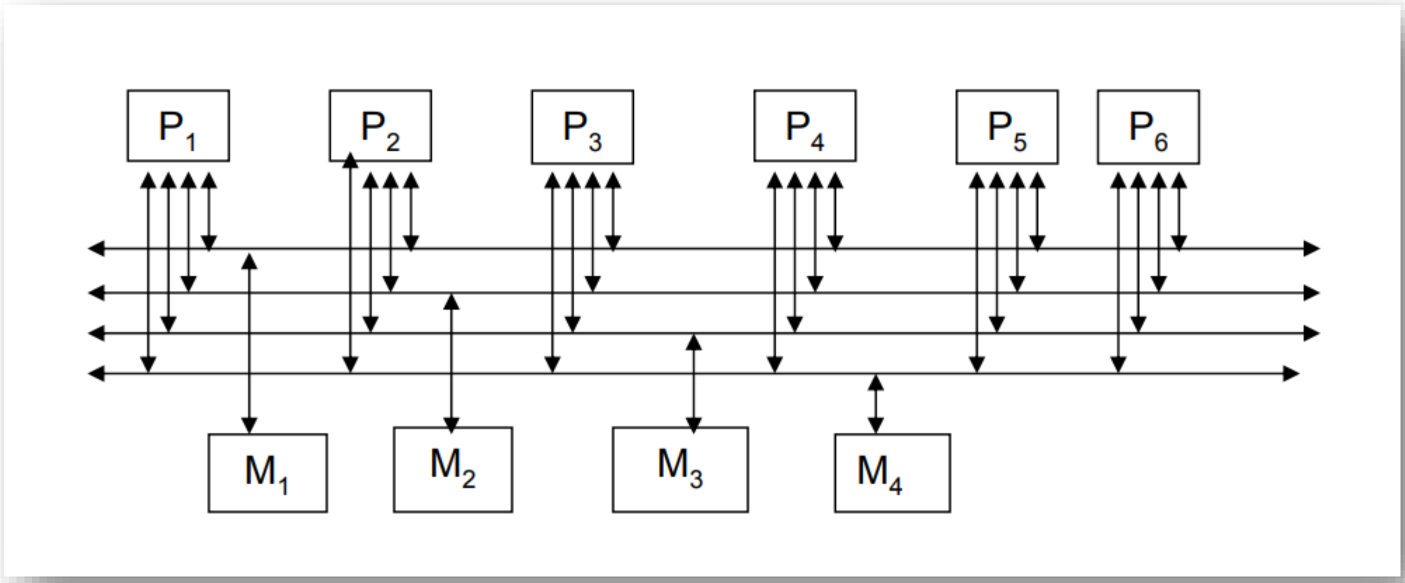
Dynamic Topologies: Multi Bus

- Several parallel buses to interconnect multiple processors and multiple memory modules.
- Many connection schemes are possible:
  - Multiple Bus with Full Bus – Memory Connection (MBFBMC).
  - Multiple Bus with Single Bus – Memory Connection (MBSBMC).
  - Multiple Bus with Partial Bus – Memory Connection (MBPBMC).
  - Multiple Bus with Class-based Bus – Memory Connection (MBCBMC).

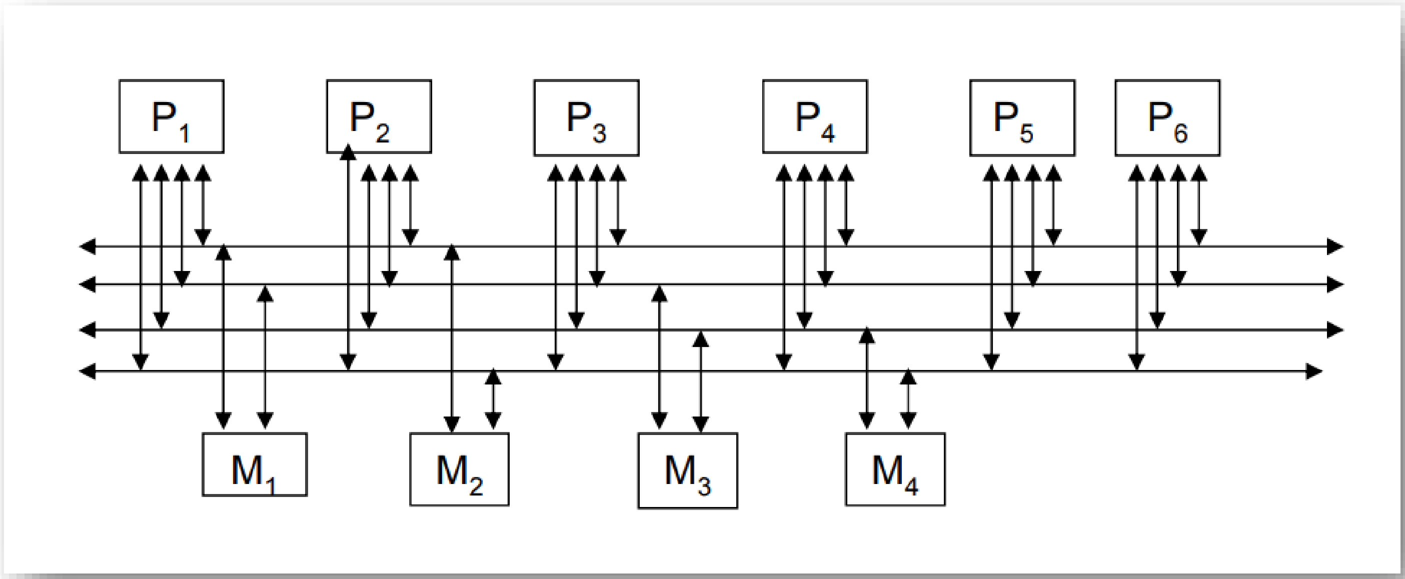
Multiple Bus with Full Bus – Memory Connection (MBFBMC)



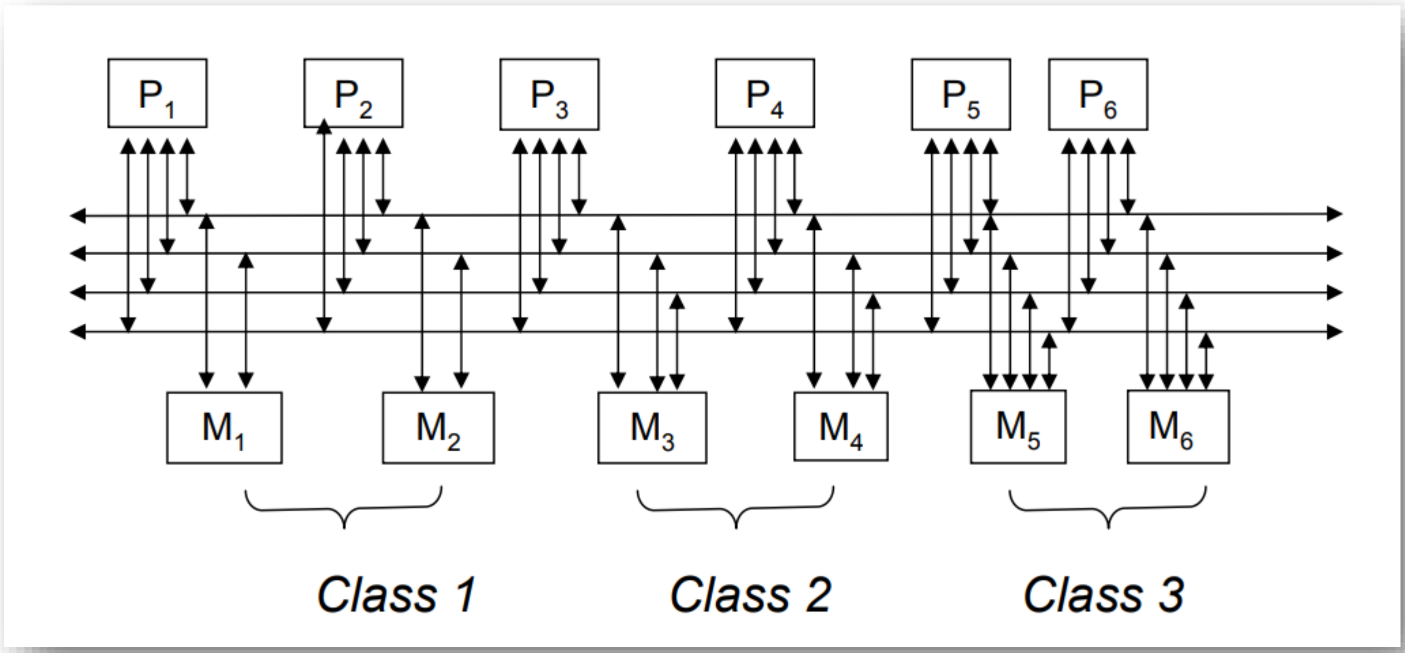
Multiple Bus with Single Bus – Memory Connection (MBSBMC)



Multiple Bus with Partial Bus – Memory Connection (MBPBMC)



Multiple Bus with Class-based Bus – Memory Connection (MBCBMC).



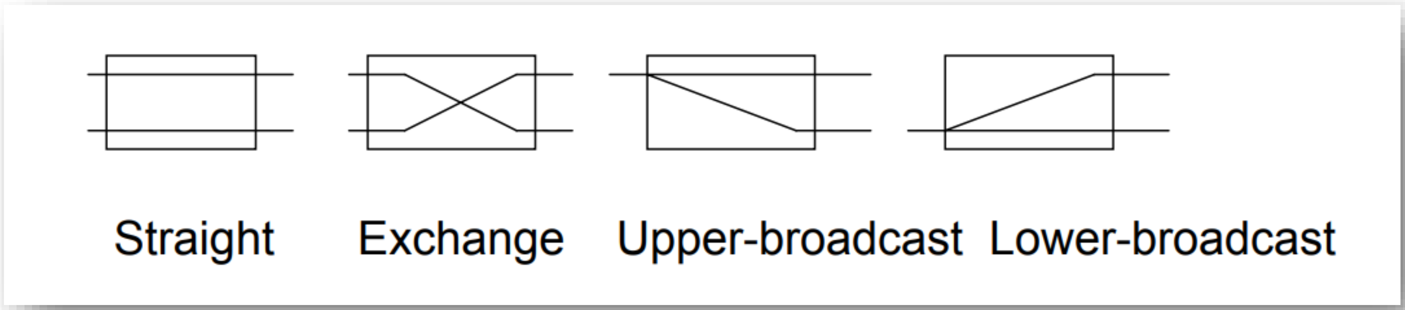
Dynamic Topologies: Switch Based Networks

Objectives

- Understanding of Single-Staged Switch.
- Understanding of Multi-Staged Switch.

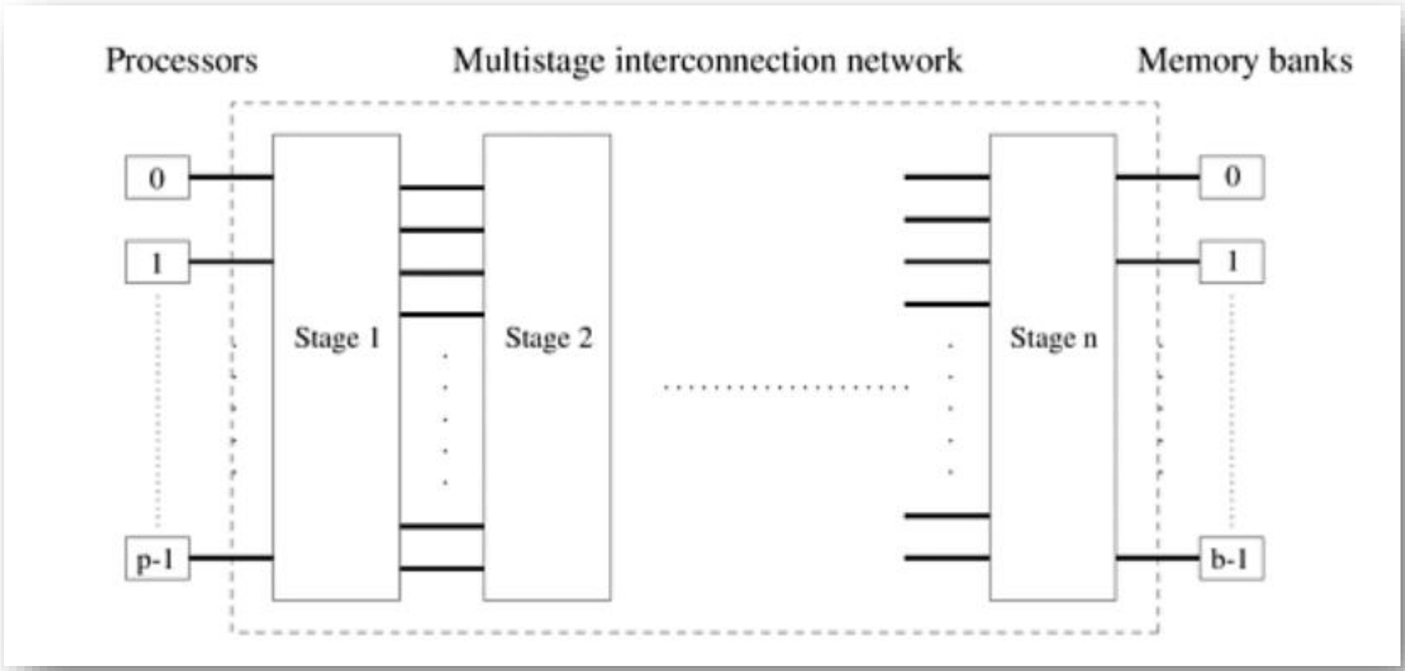
Switch-based Dynamic Topologies: Sigle-Stage Switch.

A single stage of SE exists between the inputs and outputs of the network. Possible settings of a 2x2 SE are:



Switch-based Dynamic Topologies: Multi-Stage Switch

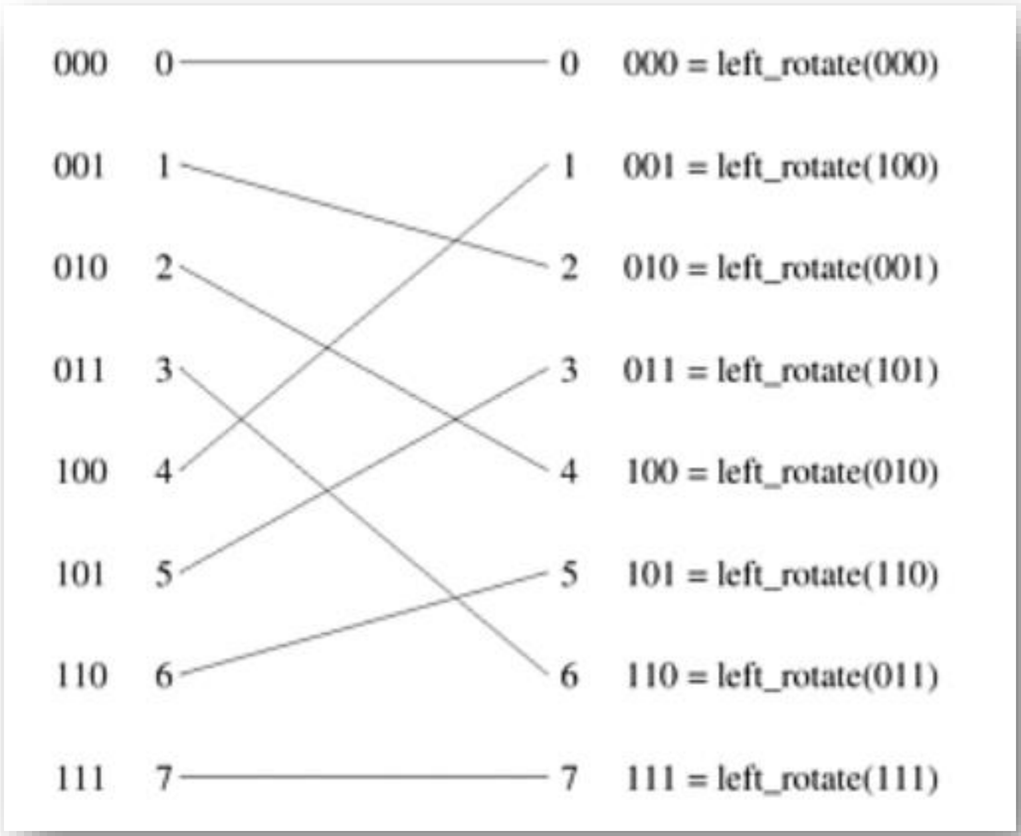
- Crossbars have excellent performance scalability but poor cost scalability.
- Buses have excellent cost scalability, but poor performance scalability.
- Multistage interconnects strike a compromise between these extremes.
- A Multistage switch network consists of a number of stages each consisting of a set of 2x2 SEs.
- Stages are connected to each other using Inter-Stage Connection (ISC) pattern.
- In MINs the routing of a message from a given source to a given destination is based on the destination address (self-routing).



The schematic of a typical multistage interconnection network

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

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



A perfect shuffle interconnection for eight inputs and outputs.

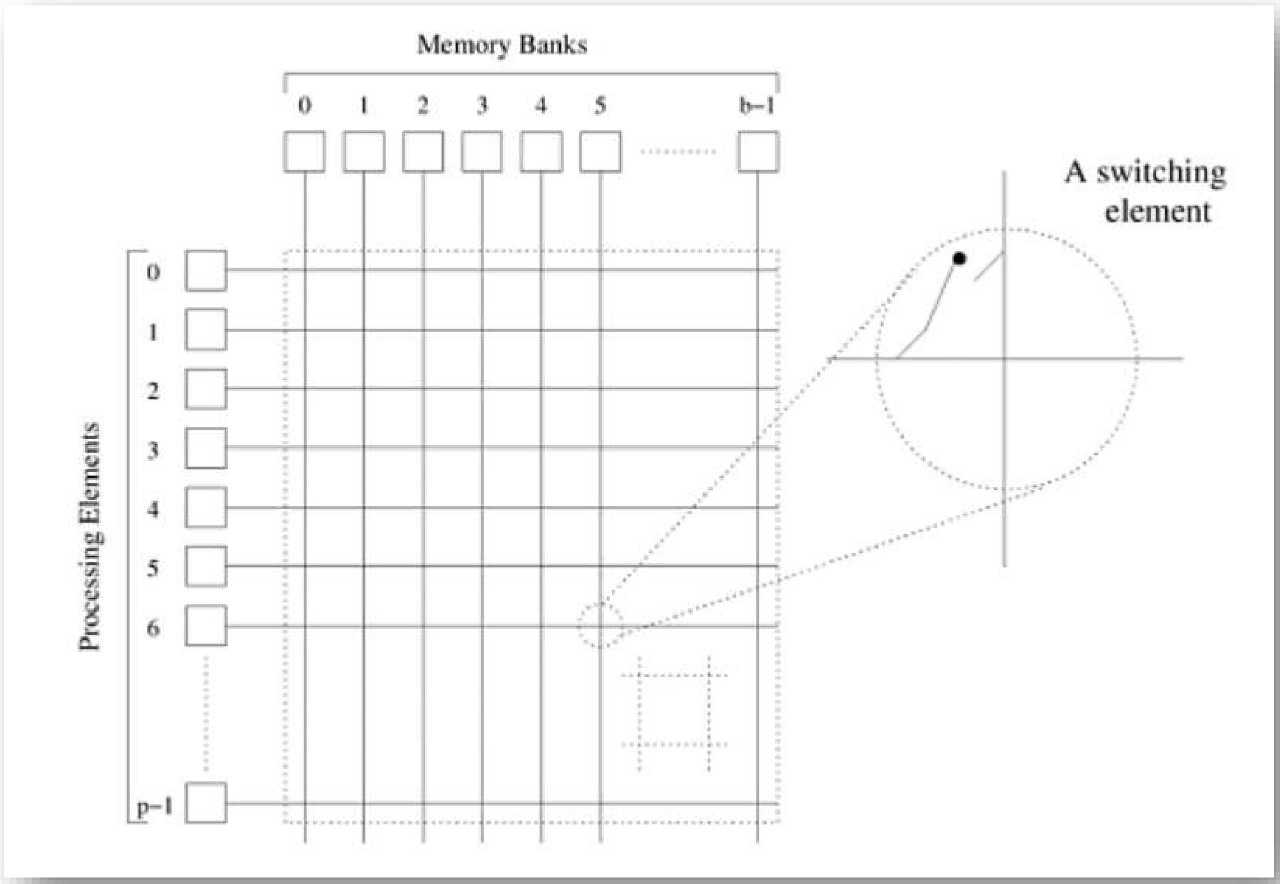
Dynamic Topologies: Cross Bar

Objectives

- Understanding of Cross Bar Topology
- Architecture of Cross Bar Topology.

Dynamic Topologies: Cross Bar

- A simple way to connect p processors to b memory banks is to use a crossbar network.
- A crossbar network uses an *p×m* grid of switches to connect *p* inputs to m outputs in a non-blocking manner.
- The cost of a crossbar of *p* processors grows as $O(p^2)$.
- This is generally difficult to scale for large values of *p*.
- Provide simultaneous connections among all its inputs and all its outputs.
- A Switching Element (SE) is at the intersection of any 2 lines extended horizontally or vertically inside the switch.
- It is a non-blocking network allowing multiple input output connection pattern to be achieved simultaneously
- Examples of machines that employ crossbars include the Sun Ultra HPC 10000 and the Fujitsu VPP500



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

Analysis and Performance Metrics

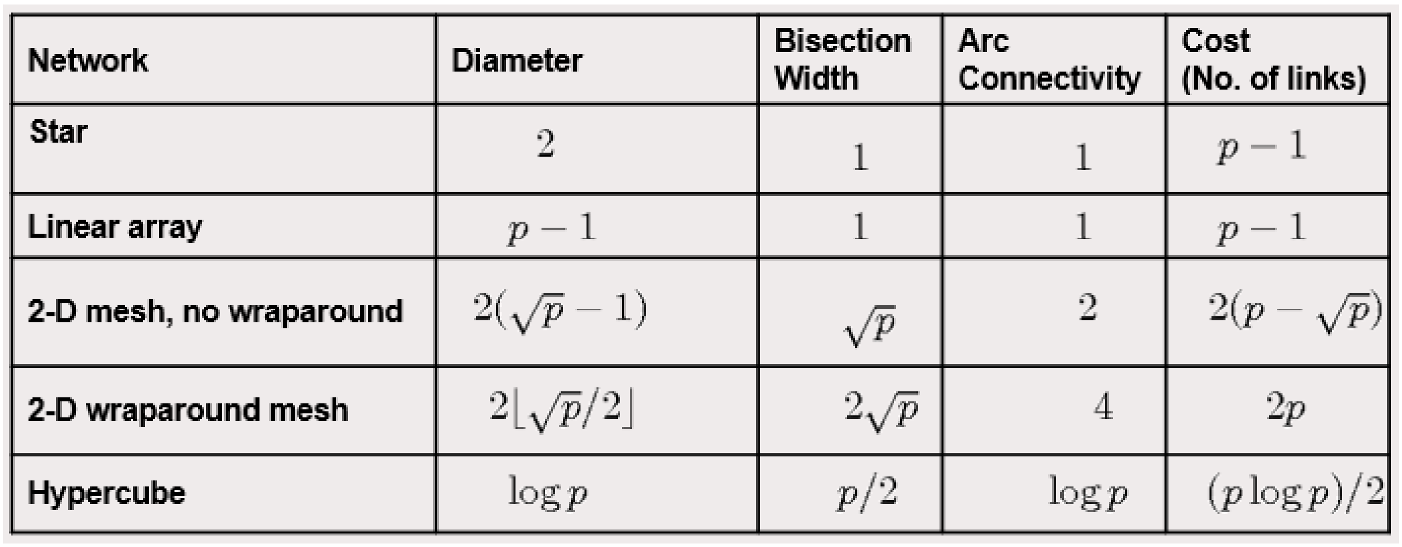
Objectives

- Evaluating of Static Interconnection Networks
- Evaluating of Dynamic Interconnection Networks

Evaluating Interconnection Networks

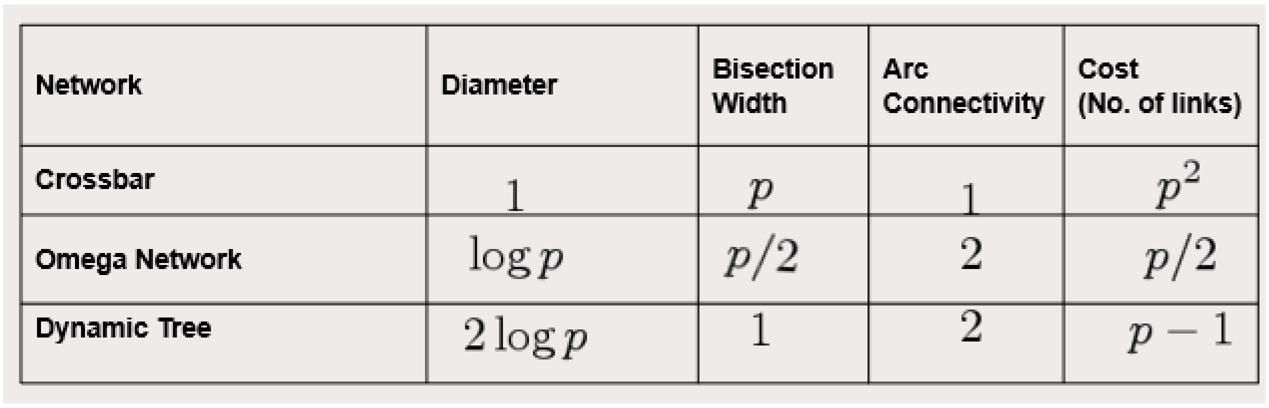
- ***Diameter:*** The distance between the farthest two nodes in the network. The diameter of a linear array is *p − 1*, that of a mesh is *2(− 1),* that of a tree and hypercube is *log p*, and that of a completely connected network is *O(1).*
- ***Bisection Width:*** The minimum number of wires you 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   , 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 into the cost

Analysis and Performance Metrics: Static Networks



Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Star	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	$\log p$	$p/2$	$\log p$	$(p\log p)/2$

Analysis and Performance Metrics: Dynamics 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	$p-1$

Week 8

What is replication of data?

Objectives

- Concept of Replication of Data
- Replication as Scaling Technique

What is replication of data?

“Data replication is the process by which data residing on a physical/virtual servers or cloud instance is continuously replicated or copied to a secondary server(s) or cloud instance. Organizations replicate data to support high availability, backup, or disaster recovery.”

Reasons for Replication

- Data are replicated to increase the reliability of a system.
- Replication for performance:
  - Scaling in numbers.
  - Scaling in geographical area.
- Replicas allows remote sites to continue working in the event of local failures
- It is also possible to protect against data corruption.

- Replicas allow data to reside close to where it is used.

Replication as Scaling Technique

Replication and caching for performance are widely applied as scaling techniques

Replicating the data and moving it closer to where it is needed helps to solve the scalability problem

When systems scale:

  - The first problems to surface are those associated with performance as the systems get bigger, they get often slower.
  - Another problem is how to efficiently synchronize all of the replicas created to solve the scalability issue?

Replication and Consistency

Objectives

- Understanding of Replication and Consistency.
- Data Consistency Models

Replication and Consistency

- Adding replicas improves scalability but provoke the overhead of keeping the replicas up-to-date. The solution often results in a relaxation of any consistency constraints.
- If there are many replicas of the same thing, it is not easy to keep all those replicas consistent. Two principal keys are:
  - How do we keep all of them up-to-date?
  - How do we keep the replicas consistent?

Replication and Consistency: Data Consistency Models

Consistency can be achieved in a number of ways. We will study a number of consistency models, as well as protocols for implementing the models

The consistency models are classified into two broader categories

  - Data-centric Consistency Models
  - Client-Centric Consistency Models

Data Consistency Models

- Data-centric Consistency Models
  - Continuous Consistency
  - Consistent Ordering of Operations
    - Causal Consistency
    - Grouping Operations
- Client-Centric Consistency Models
  - Eventual Consistency
  - Monotonic Writes
  - Read Your Writes
  - Writes Follow Reads

Data-Centric Consistency Models: Continuous Consistency

Objectives

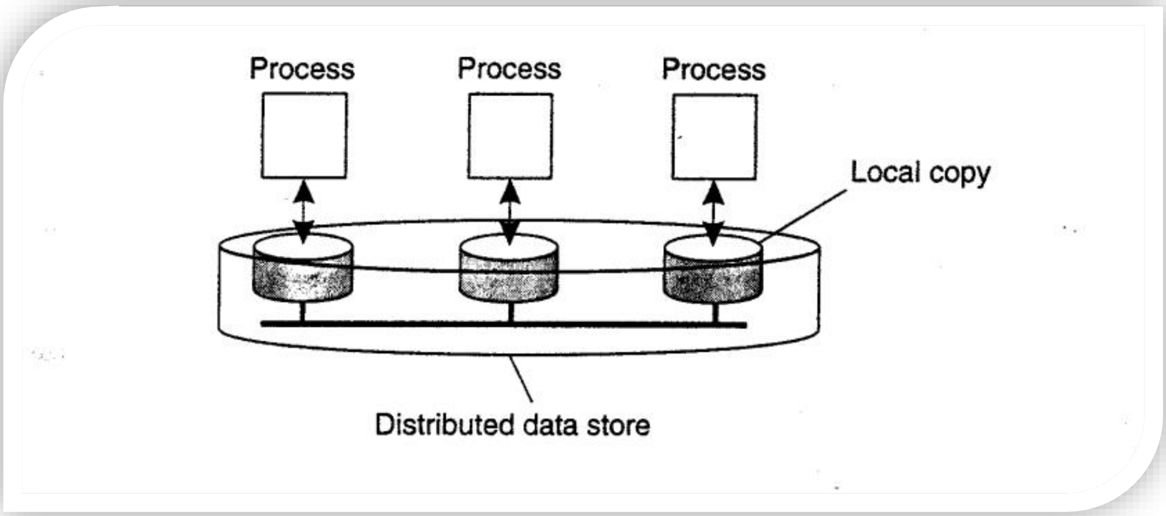
- Understanding of Data-Centric Consistency Models.
- Understanding of Continuous Consistency
- Understanding of Conit with example

Data Consistency Model

"A contract between processes and the data store that says that if processes agree to obey certain rules, the store promises to work correctly."

Data-Centric Consistency Models

- A data-store can be read from or written to by any process in a distributed system.
- A local copy of the data-store (replica) can support "fast reads".
- A write to a local replica needs to be propagated to all remote replicas.



The general organization of a logical data store, physically distributed and replicated across multiple processes.

Data-Centric Consistency Models: Continuous Consistency

Degree of consistency: Yu and Vahdat (2002) take a general approach by distinguishing three independent axes for defining inconsistencies:

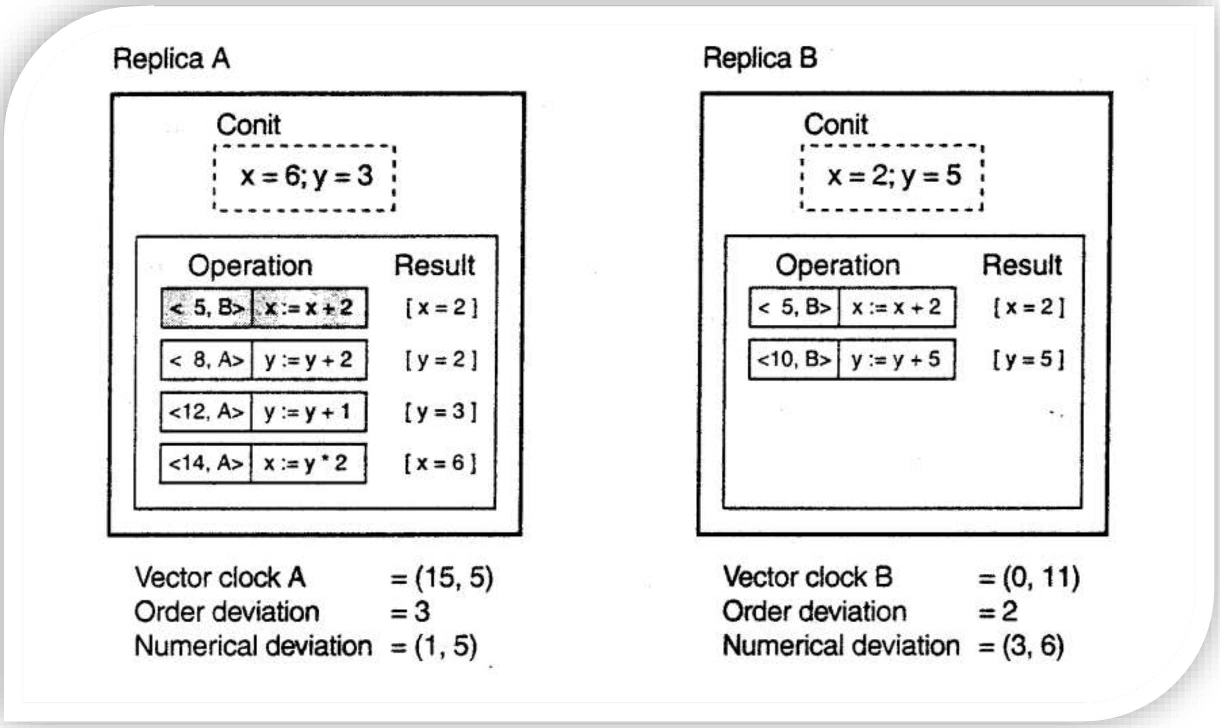
  - Replicas may differ in their numerical value.
  - Replicas may differ in their relative staleness.
  - There may differences with respect to order of performed update operations.

- A conit specifies the unit over which consistency is to be measured.
- Example: numerical and ordering deviations.

contains the variables x and y:

  - Each replica maintains a vector clock
  - B sends A operation [h5,Bi: x := x + 2];
  - A has made this operation permanent (cannot be rolled back)

- A has three pending operations; order deviation = 3
- A has missed one operation from B, yielding a max diff of 5 units) (1,5)



An example of keeping track of consistency deviations [adapted from (Yu and Vahdat, 2002)].

Data-Centric Consistency Models: Sequential Consistency

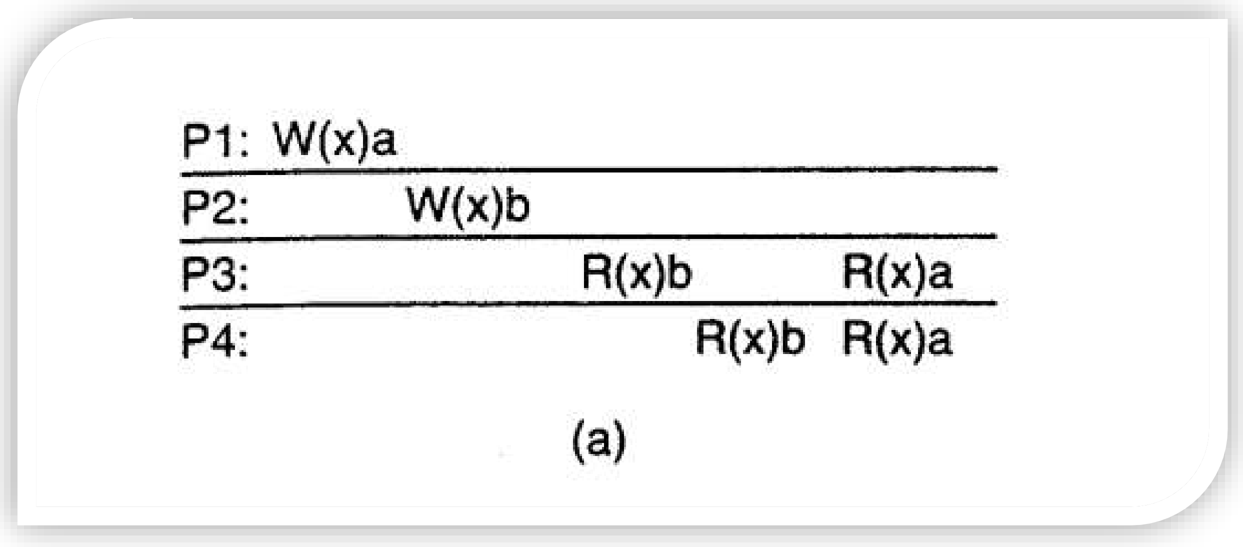
Objectives

- Introduction of Sequential Consistency.
- Example of Sequential Consistency.

Data-Centric Consistency Models: Sequential Consistency

"The result of any execution is the same as if the (read and write) operations by all processes on the data-store were executed in the same sequential order and the operations of each individual process appear in this sequence in the order specified by its program."

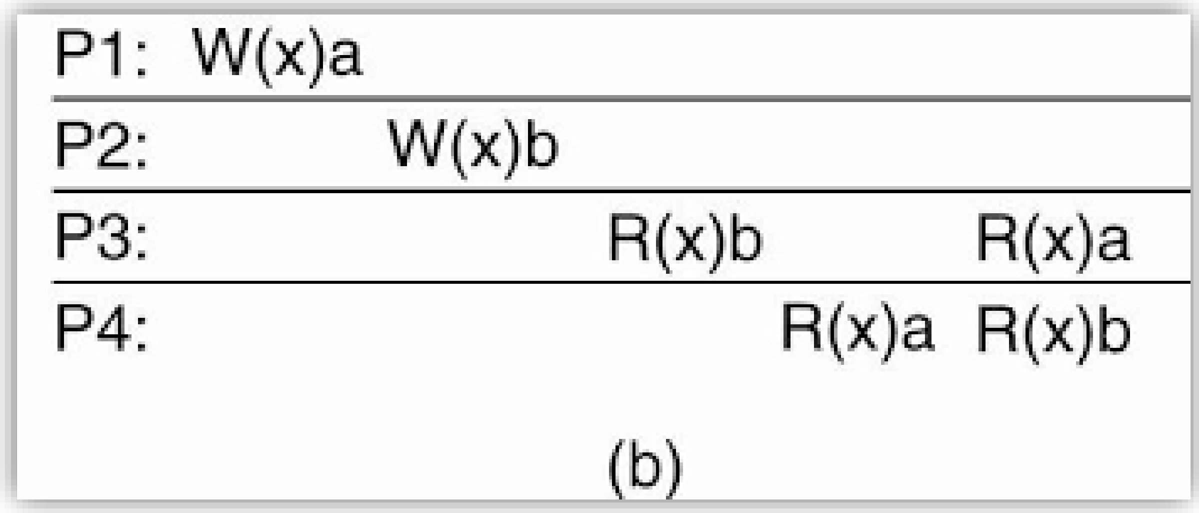
- A weaker consistency model, which represents a relaxation of the rules.
- It is also must easier to implement.
- Example: Time independent process. Four processes operating on the same data item x.
  - Process P1 first performs W(x)a to x.
  - Later (in absolute time), process P2 performs a write operation, by setting the value of x to b.
  - Both P3 and P4 first read value b, and later value a.
  - Write operation of process P2 appears to have taken place before that of P1.



(a) A sequentially consistent data store.

Example: Time independent process. Four processes operating on the same data item x.

- Violates sequential consistency - not all processes see the same interleaving of write operations.
- To process P3, it appears as if the data item has first been changed to b, and later to a.
- But, P4 will conclude that the final value is b.



(b) A data store that is not sequentially consistent

Data-Centric Consistency Models: Causal Consistency

Objectives

- Introduction of Casual Consistency.
- Example of Casual Consistency.

Data-Centric Consistency Models: Causal Consistency

For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

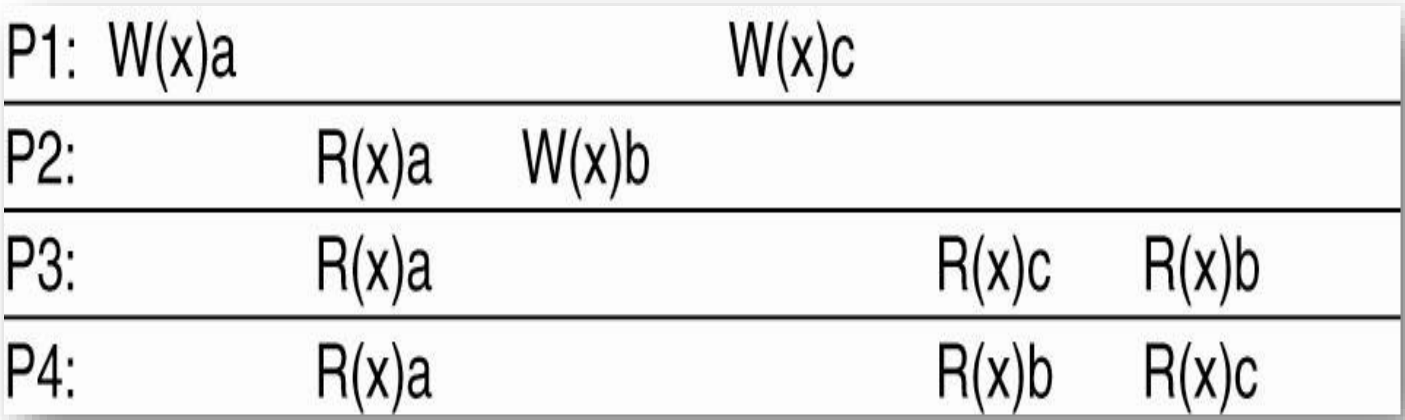
- Writes that are potentially causally related must be seen by all processes in the same order.
- Concurrent writes may be seen in a different order on different machines

Causal Consistency Example

Interaction through a distributed shared database.

- Process P1 writes data item x.
- Then P2 reads x and writes y.
- Reading of x and writing of y are potentially causally related because the computation of y may have depended on the value of x as read by P2 (i.e., the value written by P1).
- Conversely, if two processes spontaneously and simultaneously write two different data items, these are not causally related.
- Operations that are not causally related are said to be concurrent.
- For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:
    - Writes that are potentially causally related must be seen by all processes in the same order.
    - Concurrent writes may be seen in a different order on different machines.

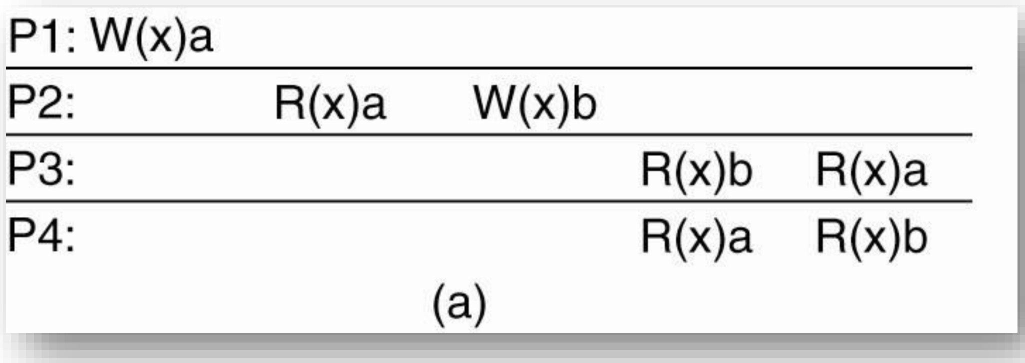
Example 1: This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.



P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

Example 2:

- W2(x)b potentially depending on W1(x)a because b may result from a computation involving the value read by R2(x)a.
- The two writes are causally related, so all processes must see them in the same order.
- It is incorrect.

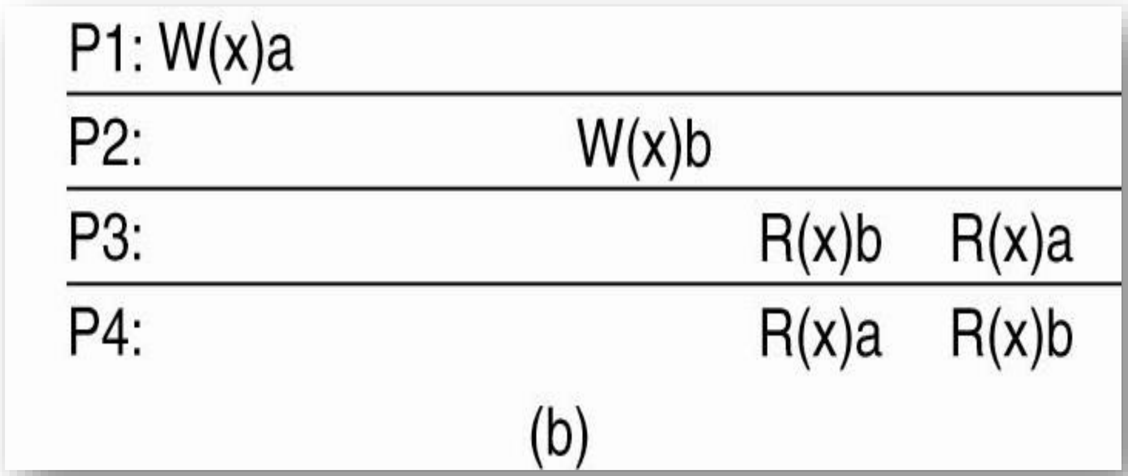


(a) A violation of a causally-consistent store.

Example 2:

- Read has been removed, so W1(x)a and W2(x)b are now concurrent writes.
- A causally-consistent store does not require concurrent writes to be globally ordered,
- It is correct.

Note: situation that would not be acceptable for a sequentially consistent store.



(a) A violation of a causally-consistent store.

Client-Centric Consistency Models: Eventual Consistency

Objectives

- Understanding of Client-Centric Consistency Models
- Understanding of Eventual Consistency.

Client-Centric Consistency Models

“A special class of distributed data-store which is characterized by the lack of simultaneous updates. Here, the emphasis is more on maintaining a consistent view of things for the individual client process that is currently operating on the data-store."

Client-centric consistency models are described using the following notations

    - X: data item
    - Xi: ith version of x
    - WS xi[t] is the set of write operations at Li that lead to version xi of x (at time t);
    - If operations in WS xi[t1] have also been performed at local copy Lj at a later time t2, we write WS (xi[t1] , xj[t2]).
    - If the ordering of operations or the timing is clear from the context, the time index will be omitted.

Client-Centric Consistency Models: Eventual Consistency

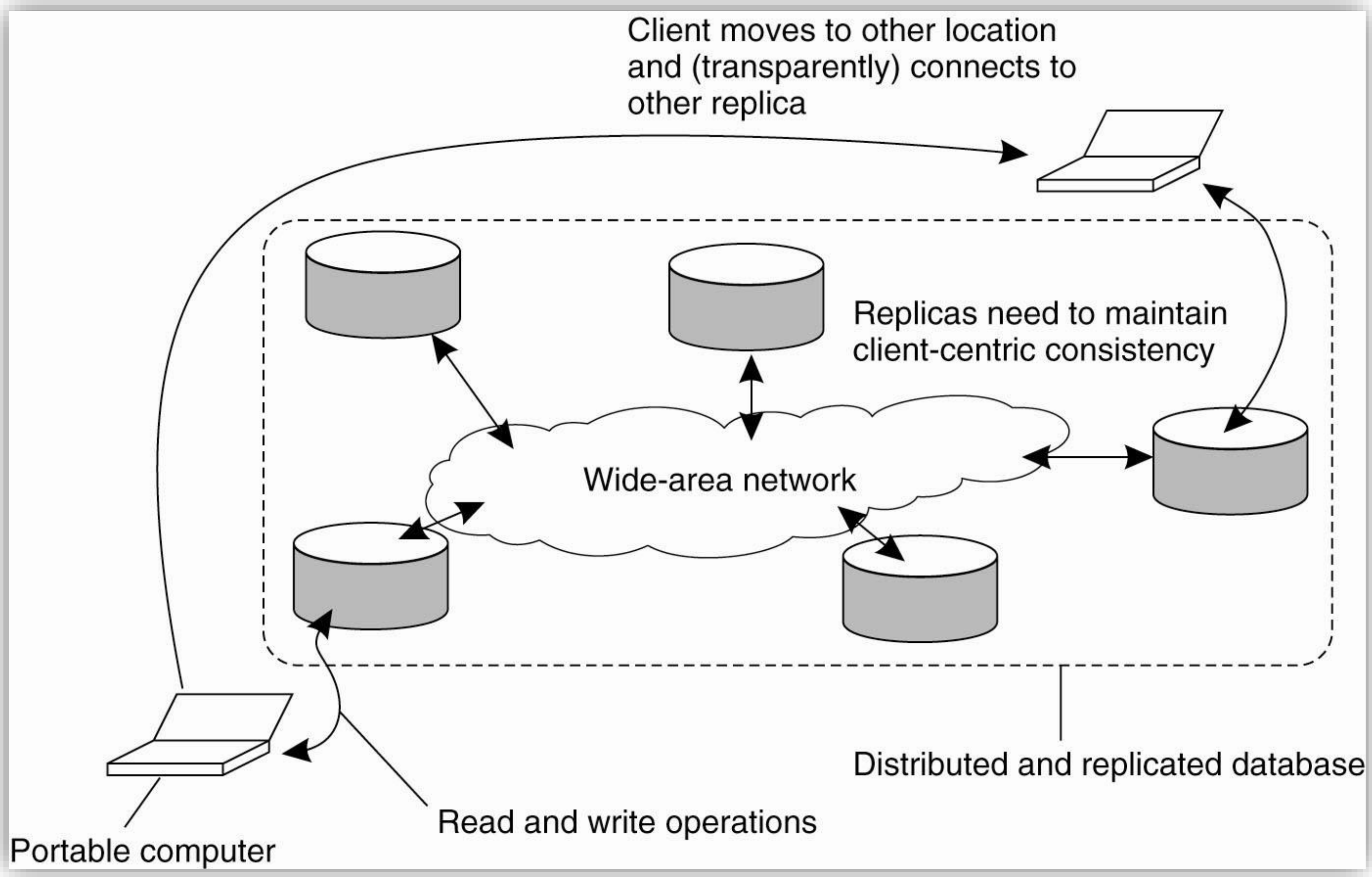
"The eventual consistency model states that, when no updates occur for a long period of time, eventually all updates will propagate through the system and all the replicas will be consistent."

- Client-centric consistency models originate from the work on Bayou
- Bayou is a database system developed for mobile computing, where it is assumed that network connectivity is unreliable and subject to various performance problems.
- Wireless networks and networks that span large areas, such as the Internet, fall into this category.
- Eventual consistency essentially requires only updates are guaranteed to propagate to all replicas.

Eventual Consistency Example

Example: Consistency for Mobile Users

- Consider a distributed database to which you have access through your notebook.
- Assume your notebook acts as a front end to the database.
- At location A you access the database doing reads and updates.
- At location B you continue your work, but unless you access the same server as the one at location A, you may detect inconsistencies:
    - Your updates at A may not have yet been propagated to B
    - You may be reading newer entries than the ones available at A
    - Your updates at B may eventually conflict with those at A


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

The principle of a mobile user accessing different replicas of a distributed database.

Client-Centric Consistency Models: Monotonic Reads

Objectives

- Understanding of Monotonic Reads.
- Monotonic Reads Example.

Client-Centric Consistency Models: Monotonic Reads

- If a process reads the value of a data item x, any successive read operation on x by that process will always return that same or a more recent value.
- Monotonic-read consistency guarantees that if a process has seen a value of x at time t, it will never see an older version of x at a later time.

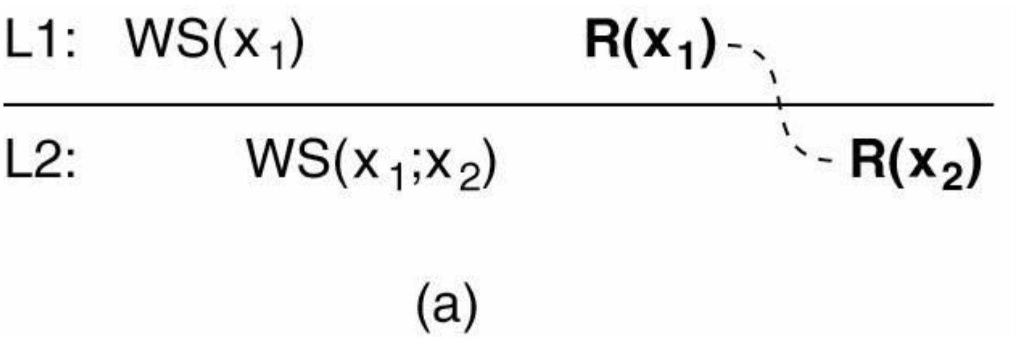
Monotonic Reads Example

Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited

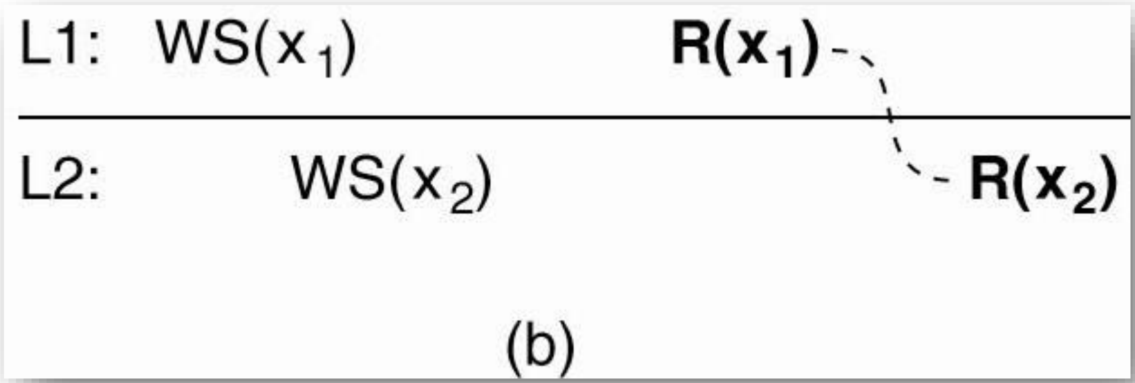
- Example: The read operations performed by a single process P at two different local copies of the same data store.

- Vertical axis - two different local copies of the data store are shown - L1 and L2.
- Time is shown along the horizontal axis.
- Operations carried out by a single process P in boldface are connected by a dashed line representing the order in which they are carried out.
- Process P first performs a read operation on x at L1, returning the value of x1 (at that time).
- This value results from the write operations in WS (x1) performed at L1.
- Later, P performs a read operation on x at L2, shown as R (x2).
- To guarantee monotonic-read consistency, all operations in WS (x1) should have been propagated to L2 before the second read operation takes place.



(a) A monotonic-read consistent data store.

- Situation in which monotonic-read consistency is not guaranteed.
- After process P has read x1 at L1, it later performs the operation R (x2) at L2 .
- But, only the write operations in WS (x2) have been performed at L2 .
- No guarantees are given that this set also contains all operations contained in WS (x1).



(b) A data store that does not provide monotonic reads

Client-Centric Consistency Models: Monotonic Writes

Objectives

- Understanding of Monotonic Writes
- Monotonic Writes Example.

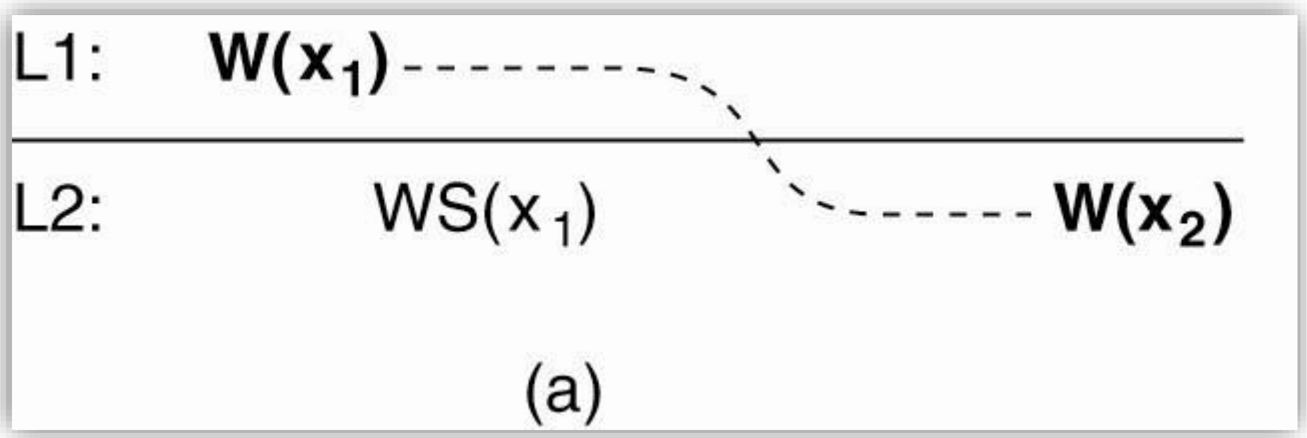
Client-Centric Consistency Models: Monotonic Writes

- A write operation by a process on a data item x is completed before any successive write operation on X by the same process.
- A write operation on a copy of item x is performed only if that copy has been brought up to date by means of any preceding write operation, which may have taken place on other copies of x. If need be, the new write must wait for old ones to finish.

Monotonic Writes Example

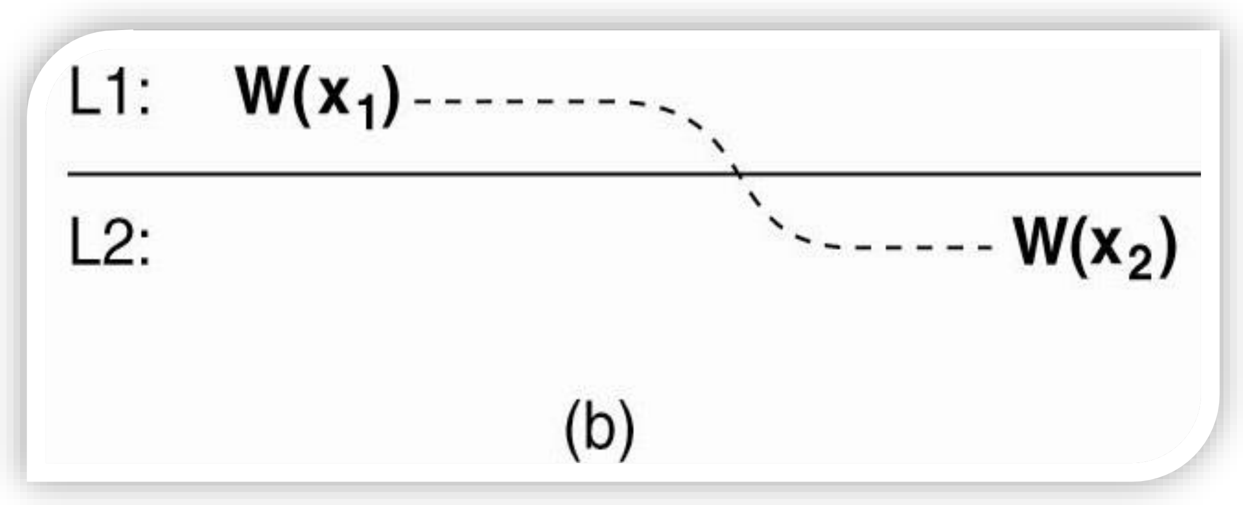
- Updating
    - Updating a program at server S2, and ensuring that all components on which compilation and linking depends, are also placed at S2.
- Maintaining
    - Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

- Process P performs a write operation on x at local copy L1, presented as the operation W(x1).
- Later, P performs another write operation on x, but this time at L2, shown as W (x2).
- To ensure monotonic-write consistency, the previous write operation at L1 must have been propagated to L2.
- This explains operation W (x1) at L2, and why it takes place before W (x2).



(a) A monotonic-write consistent data store.

- Situation in which monotonic-write consistency is not guaranteed.
- Missing is the propagation of W(x1) to copy L2.
- No guarantees can be given that the copy of x on which the second write is being performed has the same or more recent value at the time W(x1) completed at L1.



(b) A data store that does not provide monotonic-write consistency.

Client-Centric Consistency Models: Read Your Writes

Objectives

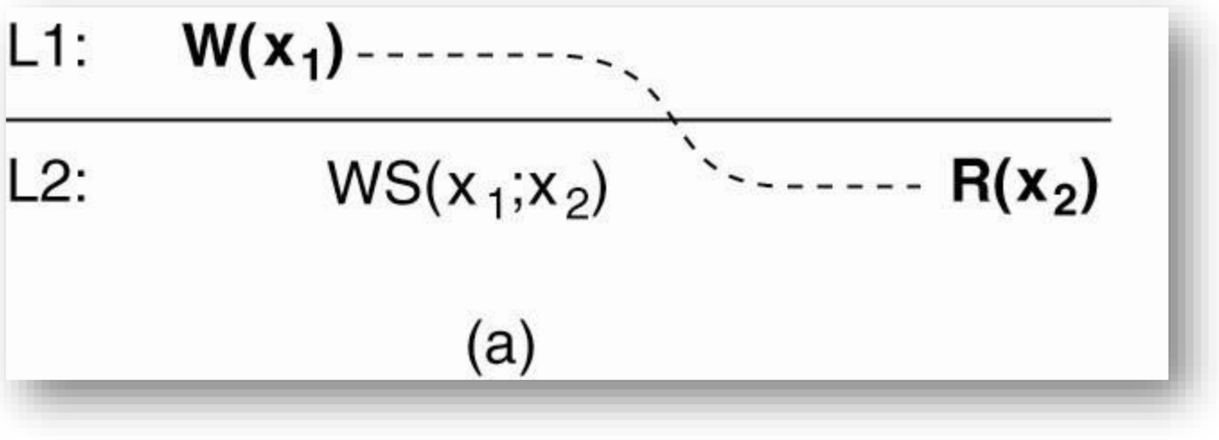
- Understanding of Read Your Writes.
- Read Your Writes Example

"Client-Centric Consistency Models: Read Your Writes

- The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.
- A write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.
- Example: Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.

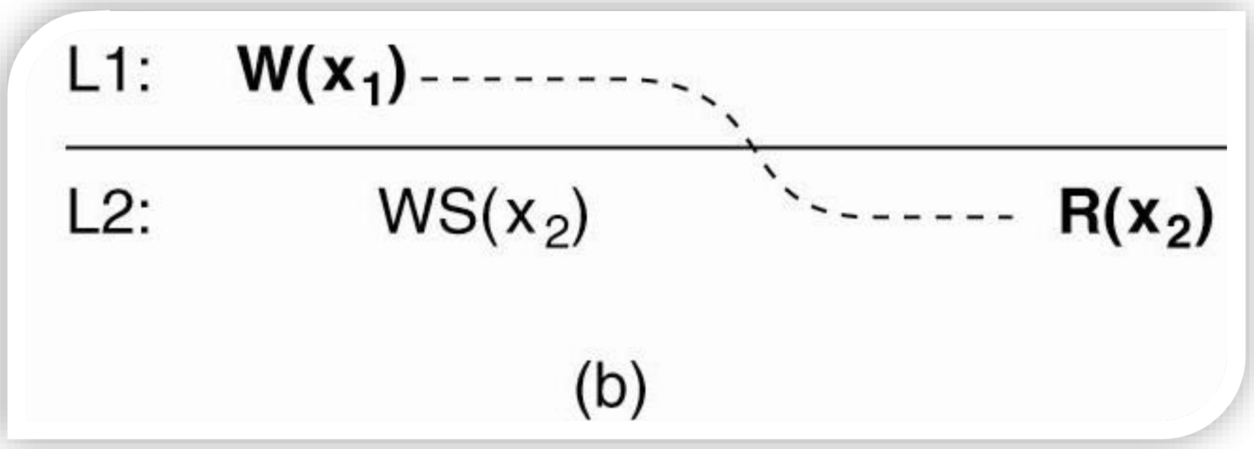
Read Your Writes Example

- Process P performed a write operation W(x1) and later a read operation at a different local copy.
- Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation.
- This is expressed by WS (x1;x2), which states that W (x1) is part of WS (x2).



(a) A data store that provides read-your-writes consistency.

- W (x1) has been left out of WS (x2), meaning that the effects of the previous write operation by process P have not been propagated to L2.



.

(b) A data store that does not.

Client-Centric Consistency Models: Writes Follow Reads

Objectives

- Understanding Writes Follow Reads Model.
- Example of Writes Follow Reads Model.

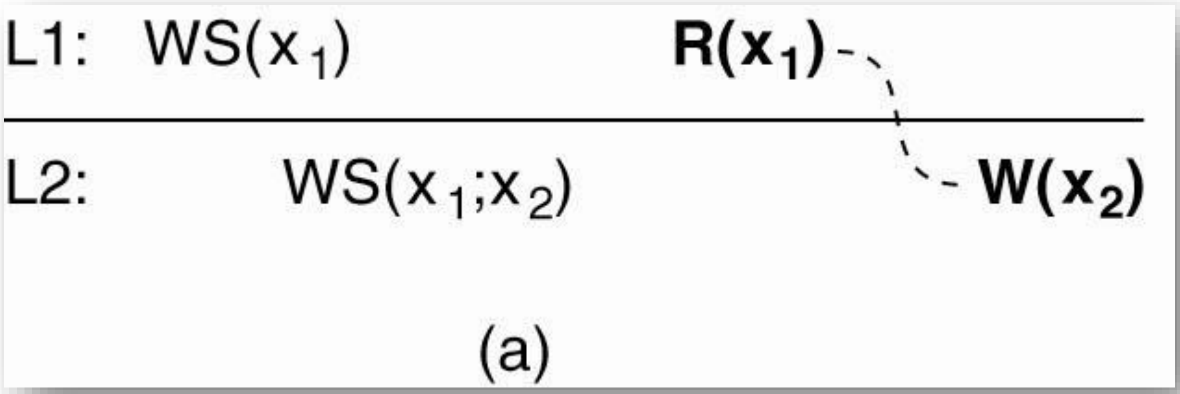
Client-Centric Consistency Models: Writes Follow Reads

- A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.
- Any successive write operation by a process on a data item x will be performed on a copy of x that is up to date with the value most recently read by that process
- Example: See reactions to posted articles only if you have the original posting (a read .pulls in. the corresponding write operation).

Writes Follow Reads Example

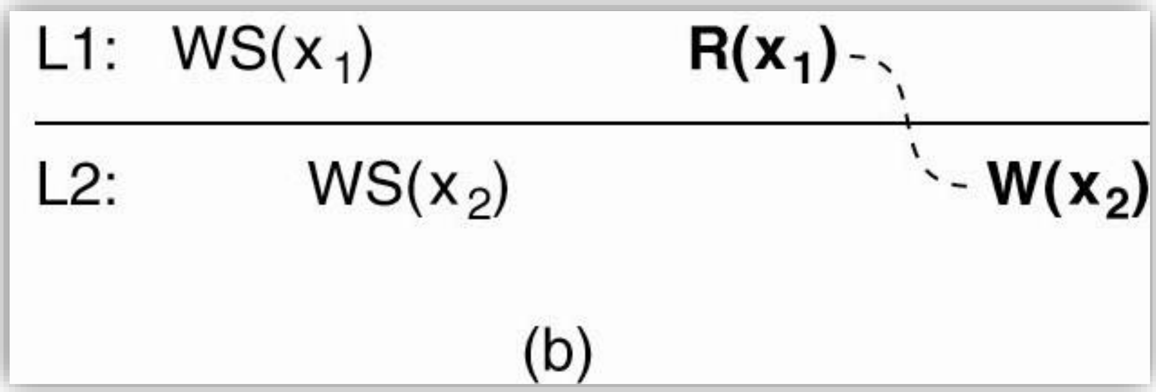
- A process reads x at local copy L1.

- The write operations that led to the value just read, also appear in the write set at L2, where the same process later performs a write operation.
- (Note that other processes at L2 see those write operations as well.)



(a) A writes-follow-reads consistent data store

- No guarantees are given that the operation performed at L2,
- They are performed on a copy that is consistent with the one just read at L1.



(b) A data store that does not provide writes-follow-reads consistency.

Week 9

Introduction to GPU

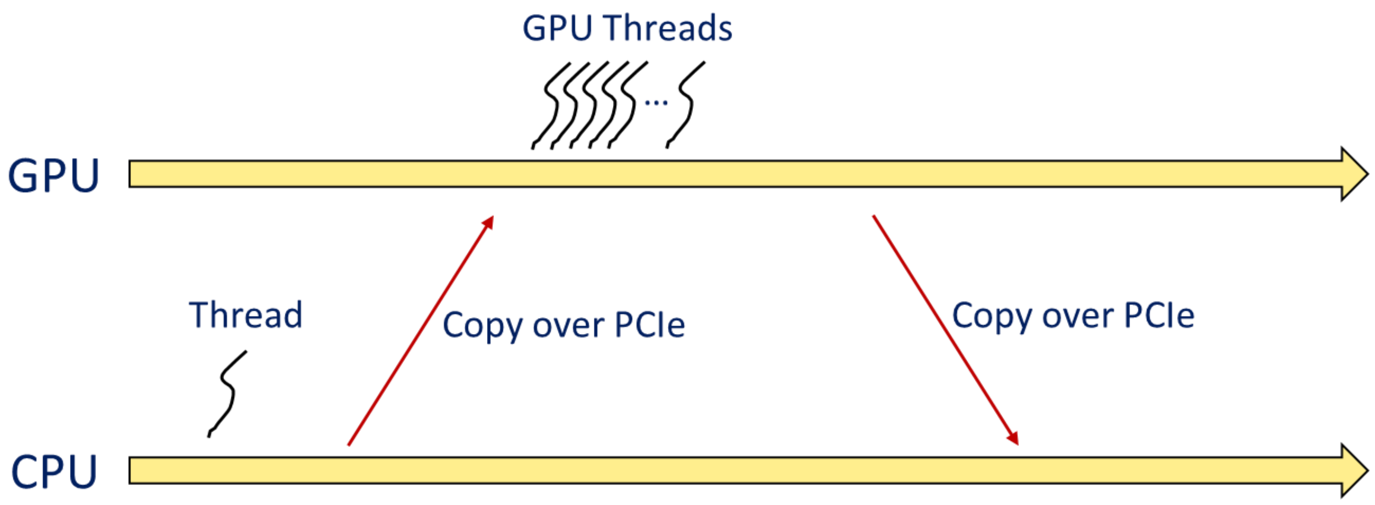
Objectives

- Introduction of GPU
- Applications of GPU

Introduction to GPU

"Graphics Processing Unit (GPU) is a chip or electronic circuit capable of rendering graphics for display on an electronic device. GPUs work by using a method called parallel processing, where multiple processors handle separate parts of the same task."

- The world's first GPU, the GeForce 256, was marketed by NVIDIA in 1999. These GPU chips can process a minimum of 10 million polygons per second and are used in nearly every computer on the market today.
- Traditional CPUs are structured with only a few cores. However, modern GPU chip can be built with hundreds of processing cores. GPU parallelism is similar to multicore parallelism.
- GPUs have a throughput architecture that exploits massive parallelism by executing many concurrent threads slowly, instead of executing a single long thread in a conventional microprocessor very quickly.
- GPUs have evolved to the point where many real-world applications are easily implemented on them and run significantly faster than on multi-core systems. Future computing architectures will be hybrid systems with parallel-core GPUs working in tandem with multi-core CPUs



NVIDIA CUDA (Compute Uniform Device Architecture) – 2007

A way to run custom programs on the massively parallel architecture.

GPU vs. CPU

- GPU is designed for highly parallel operations while a CPU execute the programs serially.
- GPUs have many parallel executing units while CPUs has a few execution units.
- GPUs have significantly faster and more advance memory interfaces as they need to shift around a lot more data than CPUs
- GPUs have much deeper pipelines several thousand stages vs 10 to 20 for CPUs

Applications of GPU

- Many applications have been developed to use GPUs for supercomputing in various fields:
    - **Scientific Computing**
        - CFD, Molecular Dynamics, Physical modeling, computational engineering, Genome Sequencing, Mechanical Simulation, Quantum Electrodynamics, Game effects (FX) physics.
    - **Image Processing**
        - Registration, interpolation, feature detection, recognition, filtering.
    - **Data Analysis**
        - Databases, matrix algebra, sorting and searching, data mining.

Architecture of GPU

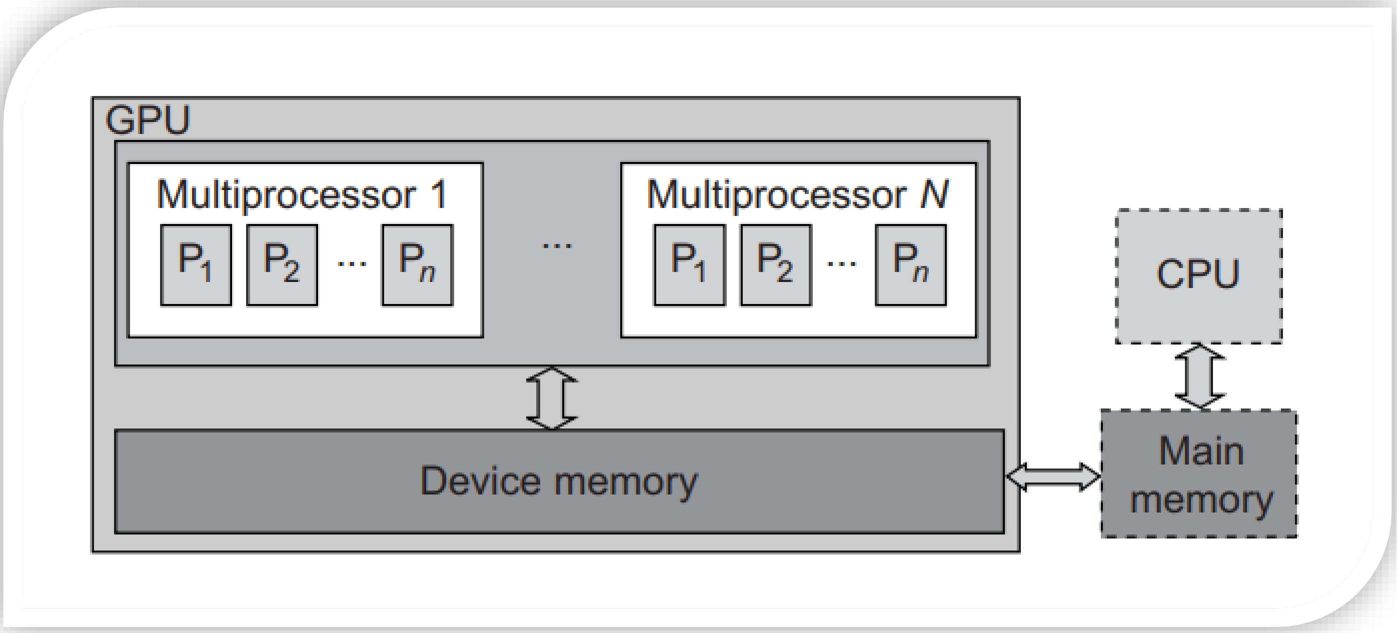
Objective

- Understanding of Architecture of GPU.
- CUDA NVIDIA's General Purpose Parallel Computing Architecture.

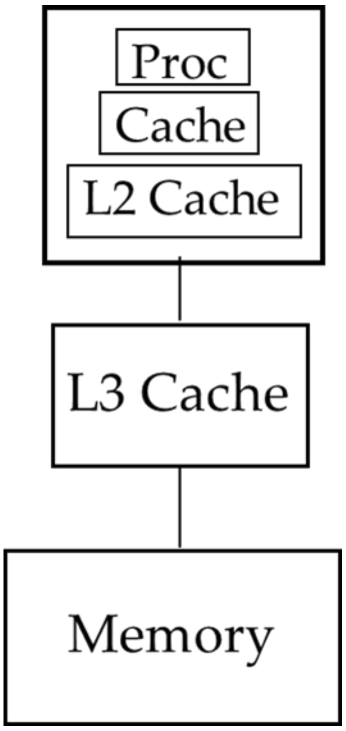
Architecture of GPU

- Parallel Coprocessor to conventional CPUs
  - Implement a SIMD structure, multiple threads running the same code.
- Grid of Blocks of Threads
  - Thread local registers
  - Block local memory and control
  - Global memory
- The CPU is the conventional multicore processor with limited parallelism to exploit.
- The GPU has a many-core architecture that has hundreds of simple processing cores organized as multiprocessors. Each core can have one or more threads.
- Essentially, the CPU's floating-point kernel computation role is largely offloaded to the many-core GPU. The CPU instructs the GPU to perform massive data processing.

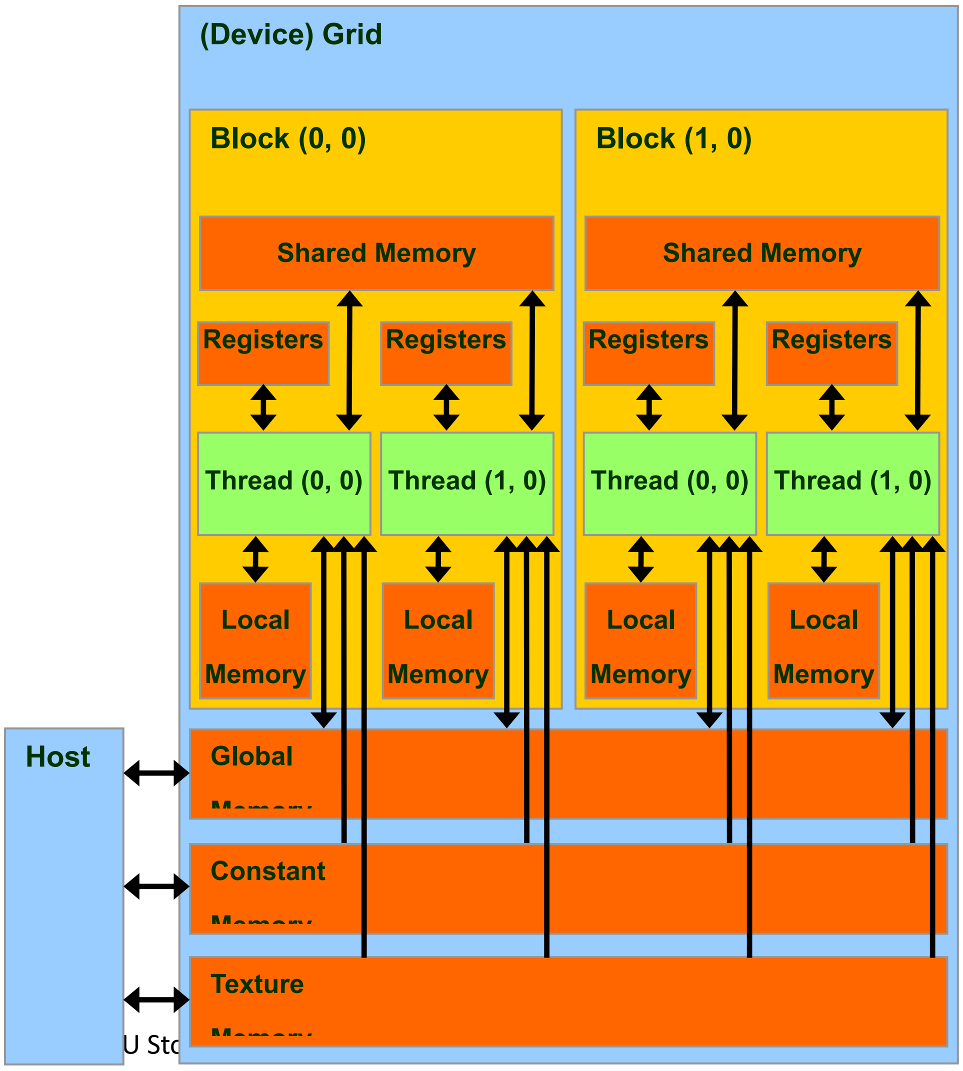
- Scale code to hundreds of cores running thousands of threads.
- The task runs on the GPU independently from the CPU.



The use of a GPU along with a CPU for massively parallel execution in hundreds or thousands of processing cores.

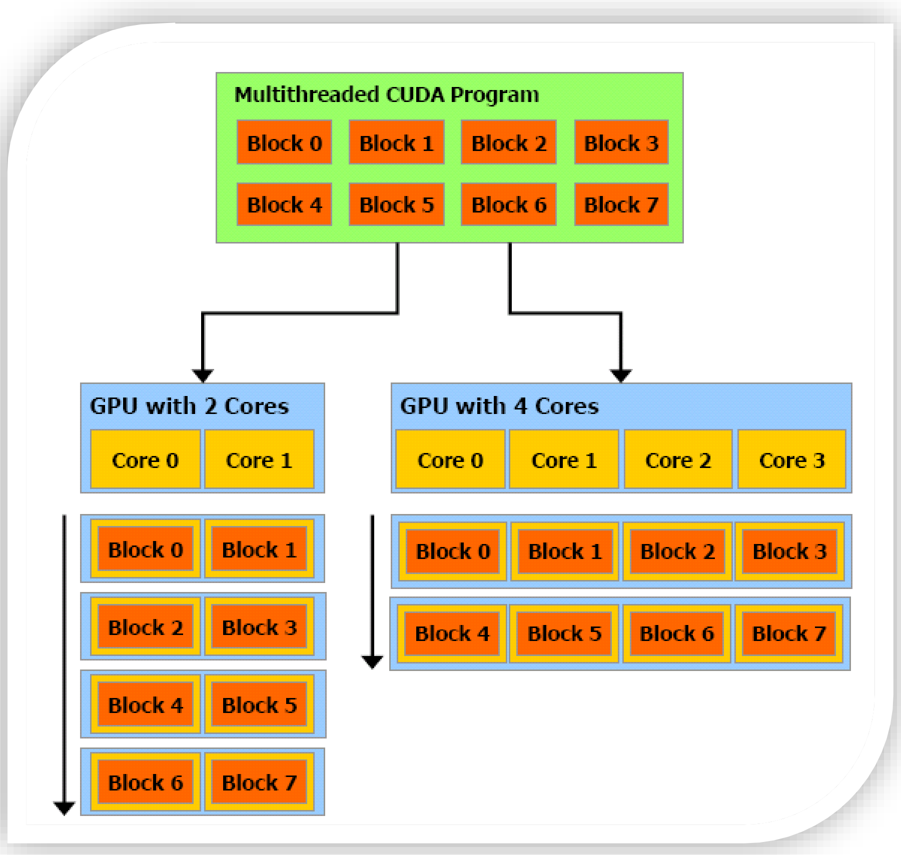


Conventional Storage Hierarchy



- Blocks map to cores on the GPU.

- Allows for portability when changing hardware.



CUDA is NVIDIA's general purpose parallel computing architecture .

- 8-series GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
    - Available in laptops, desktops, and clusters.
- GPU parallelism is doubling every year.
- Programming model scales transparently.
- Programmable in C with CUDA tools.
- Multithreaded SPMD model uses application data parallelism and thread parallelism.

GPU Programming Models

Objectives

- Understanding of GPU Programming Models
- GPU Accelerated Libraries

GPU Programming Models

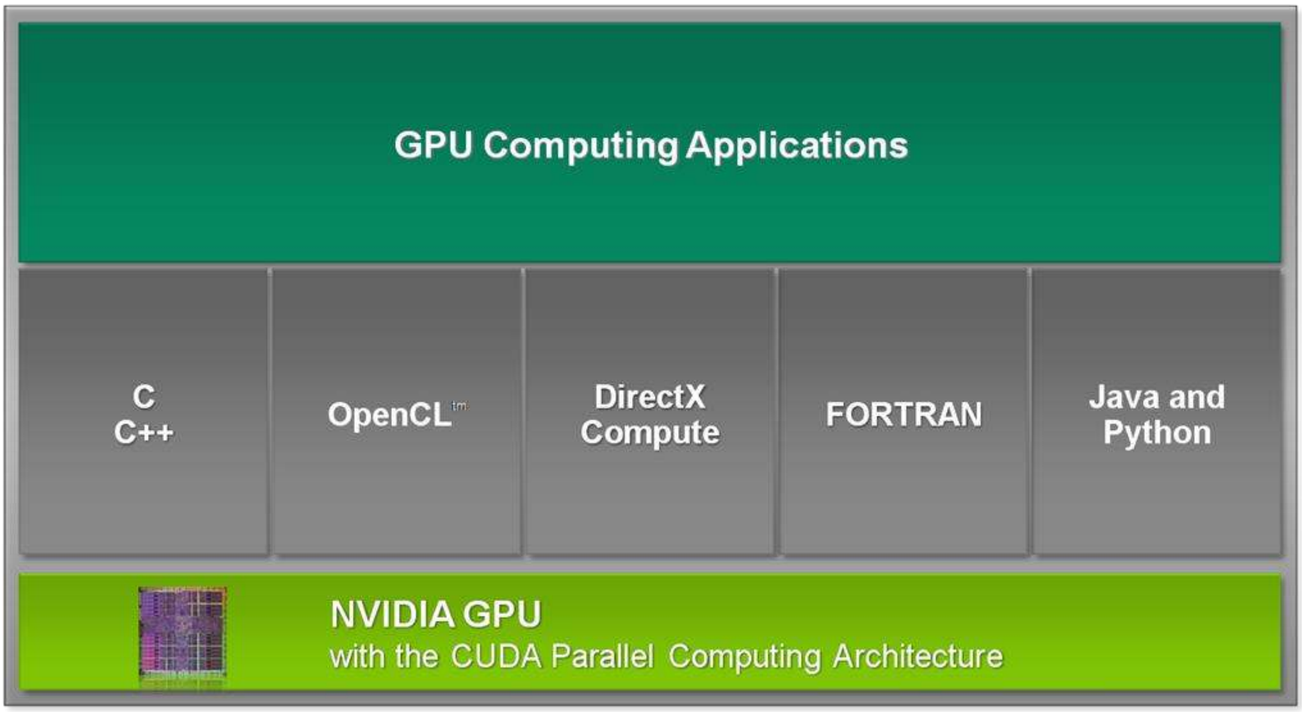
"GPU Programming is a method of running highly parallel general-purpose computations on GPU accelerators. While the past GPUs were designed exclusively for computer graphics, today they are being used extensively for general-purpose computing (GPGPU computing) as well."

GPU Programming Models: CUDA

CUDA is NVIDIA's general purpose parallel computing architecture .

- Designed for calculation-intensive computation on GPU hardware.

CUDA is not a language, it is an API

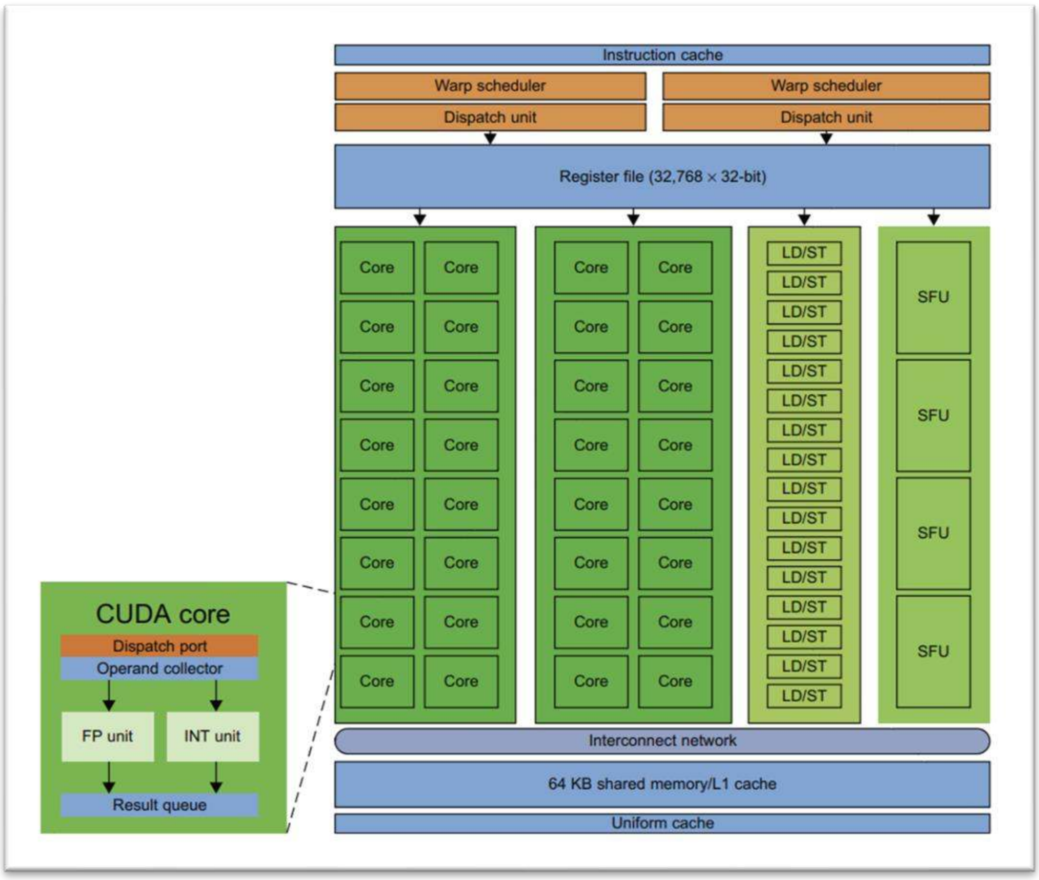

GPU Computing Applications
C
C++
OpenCL™
DirectX Compute
FORTRAN
Java and Python
NVIDIA GPU
with the CUDA Parallel Computing Architecture

GPU Programming Models: CUDA

- General purpose programming model:
  - User kicks off batches of threads on the GPU.
  - GPU = dedicated super-threaded, massively data parallel co-processor.
- CUDA compute device:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device

GPU Programming Models: CUDA

Figure shows the architecture of the Fermi GPU, a next-generation GPU from NVIDIA. This is a streaming multiprocessor (SM) module. Multiple SMs can be built on a single GPU chip. The Fermi chip has 16 SMs implemented with 3 billion transistors. Each SM comprises up to 512 streaming processors (SPs), known as CUDA cores. The Tesla GPUs used in the Tianhe-1a have a similar architecture, with 448 CUDA cores.



NVIDIA Fermi GPU built with 16 streaming multiprocessors (SMs) of 32 CUDA cores each; only one SM Is shown.

GPU Accelerated Libraries

NVIDIA cuBLAS
NVIDIA cuRAND
NVIDIA NPP
OpenCV
NVIDIA cuSPARSE
CUSP
Sparse Linear Algebra
Thrust
C++ STL Features for CUDA
NVIDIA cuFFT

Power Efficiency of GPU

Objectives

- Performance of GPU

- Power Efficiency of GPU

Performance of GPU

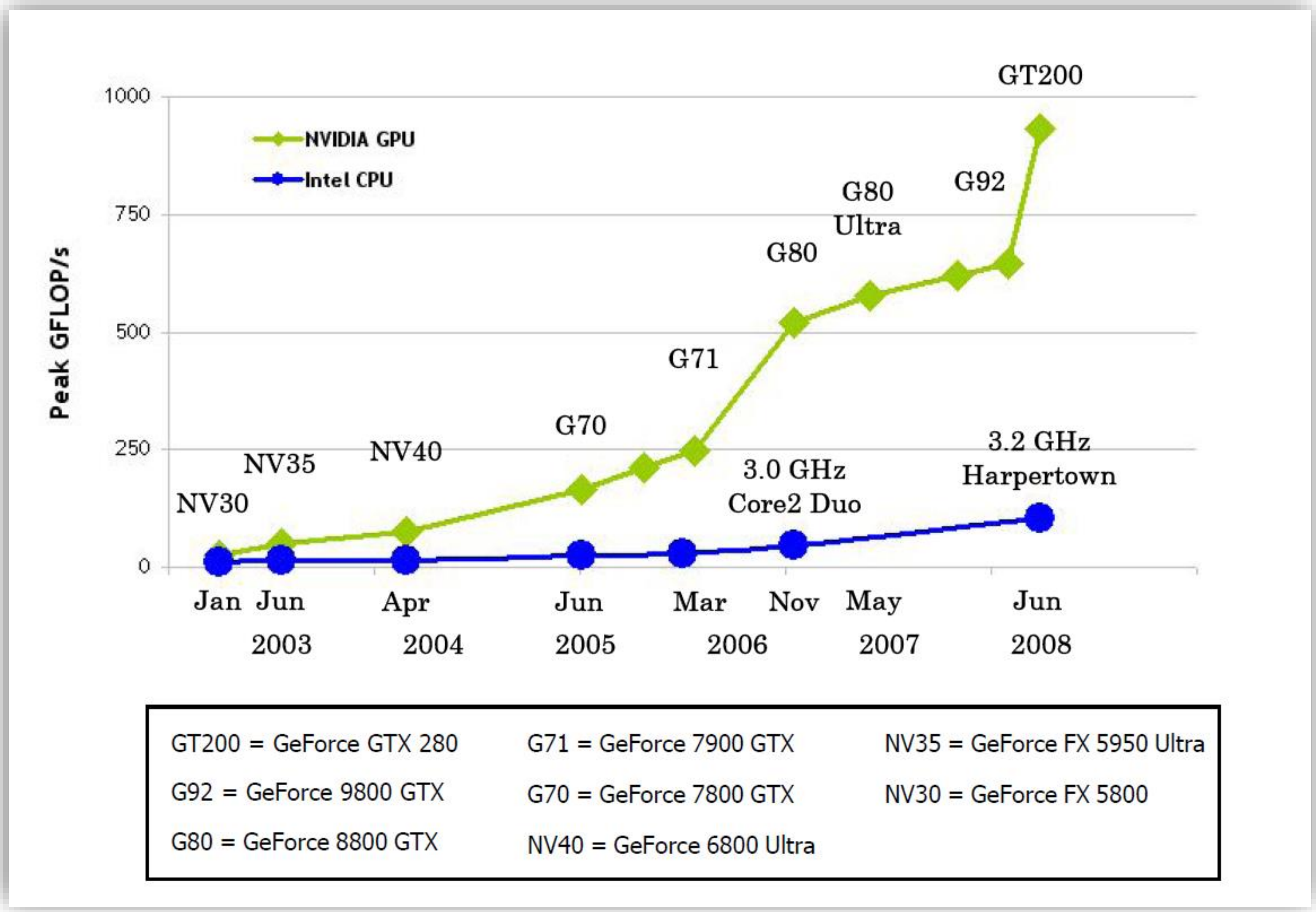
Bill Dally of Stanford University considers power and massive parallelism as the major benefits of GPUs over CPUs for the future. Two Aspects:

  - Data Access Rate Capability
    - Bandwidth
  - Data Processing Capability
    - How many ops per sec

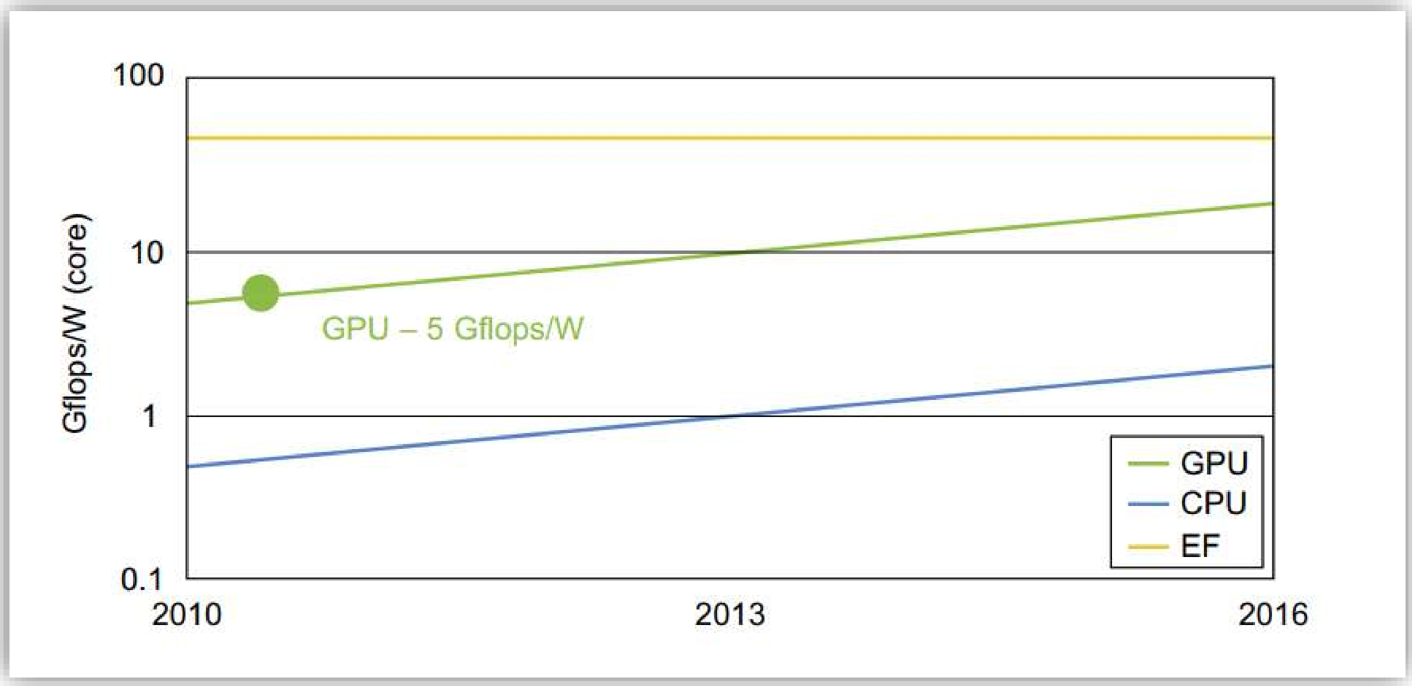
Performance of GPU: Data Access Capability

- **High-End CPU Today**
  - 31.92 GB/sec (nehalem) - 12.8 GB/sec (hapertown)
  - Bus width 64-bit
- **GPU / GTX280**
  - 141.7 GB/sec
  - Bus width 512-bit
  - 4.39x – 11x
- **GFLOPS**
  - Billion Floating-Point Operations per Second
  - **Caveat: FOPs can be different**
    - **But today things are not as bad as before**
- **High-End CPU today**
  - 3.4Ghz x 8 FOPS/cycle = 27 GFLOPS
  - Assumes SSE
- **High-End GPU today / GTX280**
  - 933.1 GFLOPS or 34x capability

Power Efficiency of GPU: GPU vs. CPU



Power Efficiency of GPU: GPU vs. CPU



The GPU performance (middle line, measured 5 Gflops/W/core in 2011), compared with the lower CPU performance (lower line measured 0.8 Gflops/W/core in 2011) and the estimated 60 Gflops/W/core performance in 2011 for the Exascale (EF in upper curve) in the future.

What is Heterogeneity?

Objectives

- Definition of Heterogeneity.
- Heterogeneity and Mobile Code

What is Heterogeneity?

“Heterogeneity in distributed computing refers to the presence of diverse types of hardware, software, and networking technologies in a distributed system. The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks.”

Heterogeneity (mobile code and mobile agent)

- Networks
- Hardware
- Operating systems and middleware
- Program languages

It calls for integration of components written using different programming languages, running on different operating systems, executing on different hardware platforms. In a distributed system, heterogeneity is almost unavoidable, as different components may require different implementation technologies.

Heterogeneity and Mobile Code

- The term mobile code is used to refer to program code that can be transferred from one computer to another and run at the destination -Java applets are example.
- Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system
- The virtual machine approach provides a way of making code executable on a variety of host computers
- Today, the most commonly used form of mobile code is the inclusion Javascript programs in some web pages loaded into client browsers.

What is Code Migration?

Objectives

- Introduction of Code Migration.
- Dynamically Configuring a Client

What is Code Migration?

“Code migration in distributed computing refers to the process of transferring software code from one computer or node in a network to another. This allows the code to be executed on the destination computer, which may have better processing power, network connectivity, or other resources that are needed to perform a particular task. ”

- Traditionally, code migration in distributed systems took place in the form of process migration in which an entire process was moved from one machine to another.
- Code migration is a form of mobile code, which is a general term that refers to any code that can be transferred from one system to another for execution. Code migration can be used in a variety of distributed computing scenarios, such as distributed processing, load balancing, fault tolerance, and resource optimization.

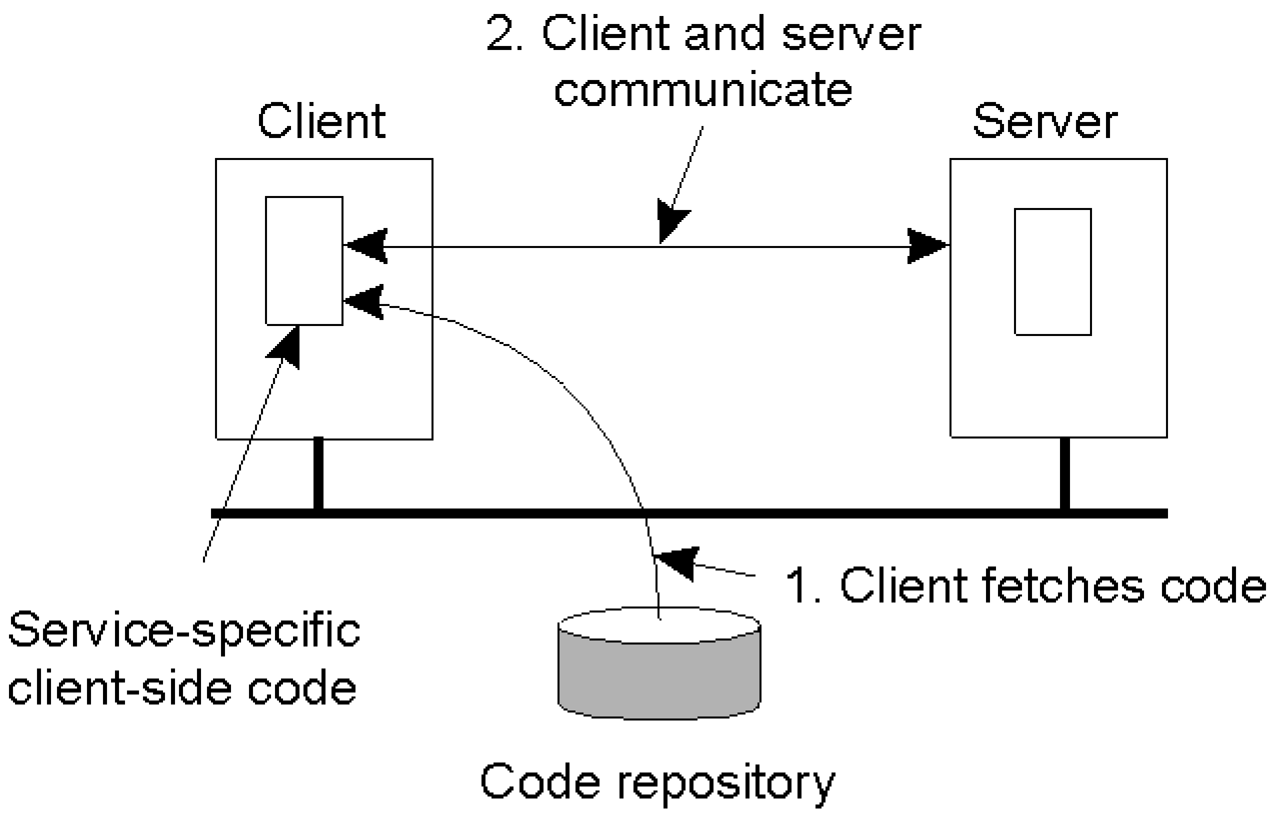
Code Migration: Motivation

- Performance
    - Move code on a faster machine.
    - Move code closer to data.
- Flexibility
    - Allow to dynamically configure a distributed system.

Dynamically Configuring a Client

- The model of dynamically moving code from a remote site does require the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine. To allow remote clients to access the file system, the server makes use of a proprietary protocol
- The server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server. At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. This principle is shown in Figure.

Dynamically Configuring a Client


2. Client and server communicate
Client
Server
1. Client fetches code
Service-specific client-side code
Code repository

The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

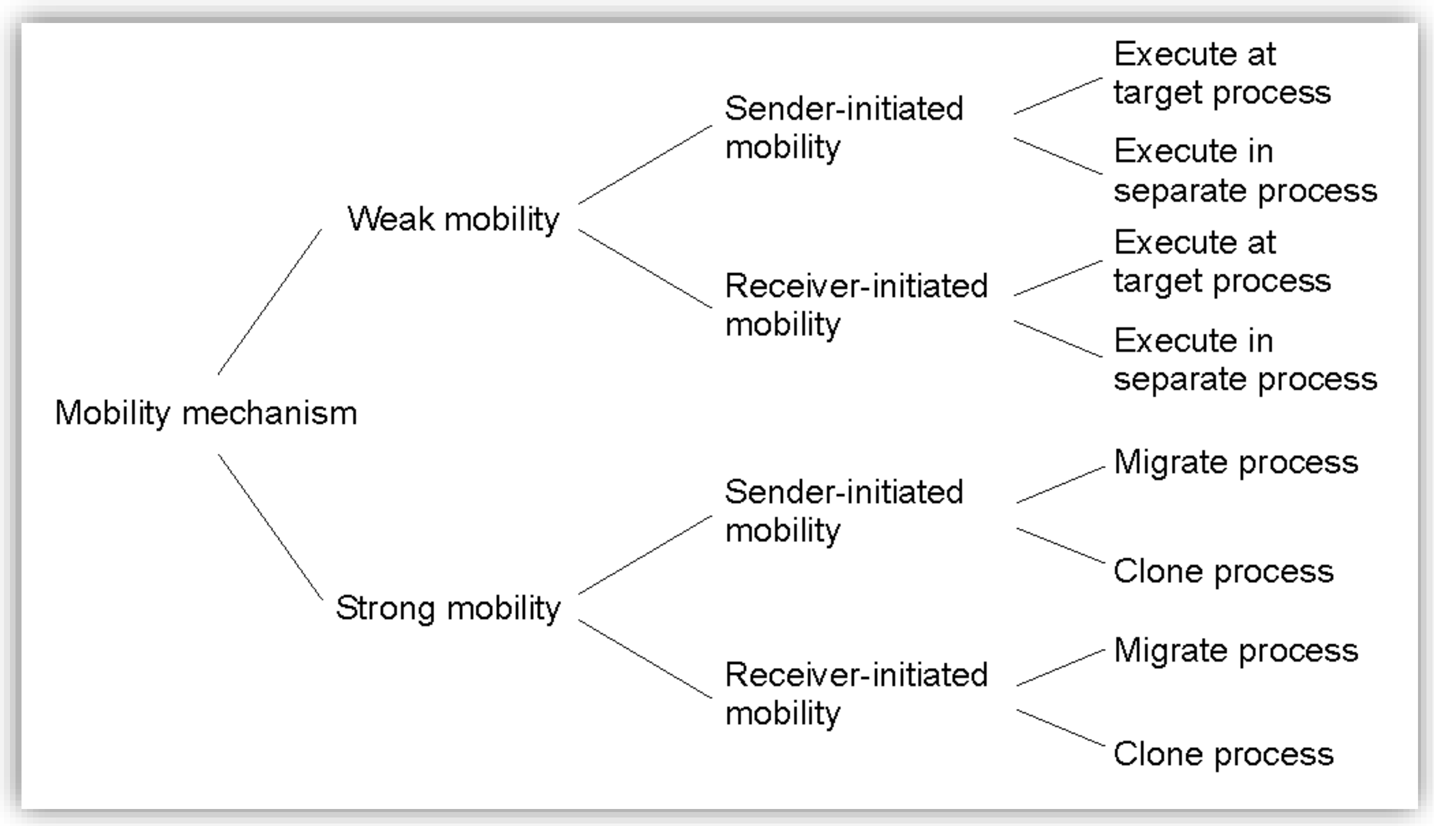
Models for Code Migration

Objectives

- What are the Models for Code Migration?
- Code Migration and Local Resources.

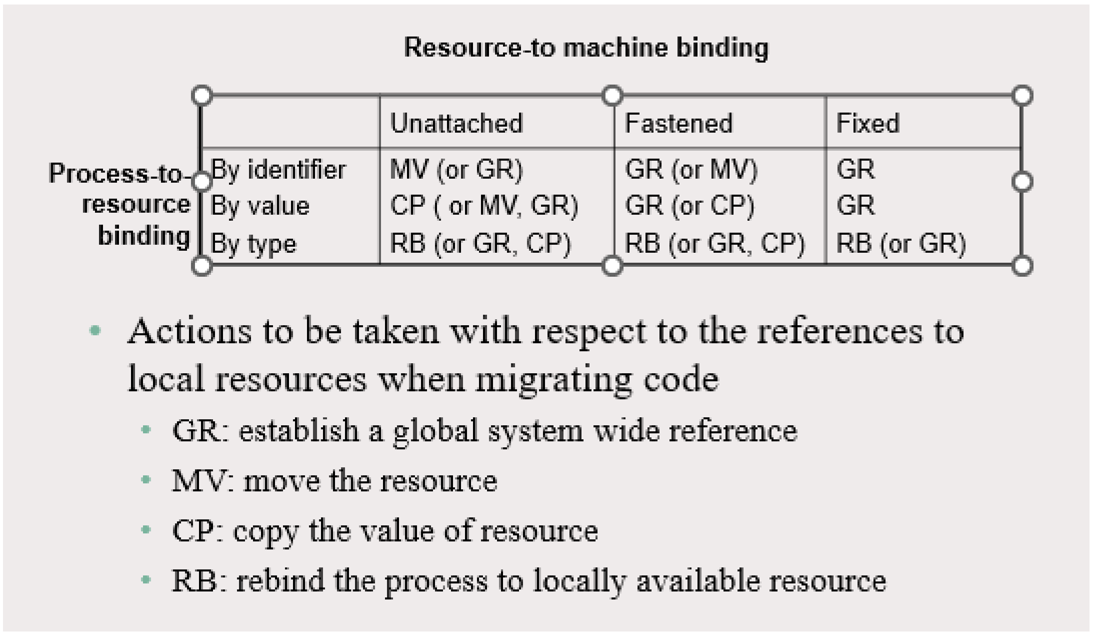
Models for Code Migration

- Process model for code migration (Fugetta et al., 98)
    - Code segment: set of instructions that make up the program
    - Resource segment: references to external resources
    - Execution segment: store current execution state
- Type of mobility
    - Weak mobility: migrate only code segment
    - Strong mobility: migrate execution segment and resource segment


Mobility mechanism
Weak mobility
Strong mobility
Sender-initiated mobility
Receiver-initiated mobility
Sender-initiated mobility
Receiver-initiated mobility
Execute at target process
Execute in separate process
Execute at target process
Execute in separate process
Migrate process
Clone process
Migrate process
Clone process

Migration and Local Resources

- Types of process-to-resource binding
    - Binding by identifier (e.g., URL, (IPaddr:Port))
    - Binding by value (e.g., standard libraries)
    - Binding by type (e.g., monitor, printer)
- Type of resources
    - Unattached resources: can be easily moved (e.g., data files)
    - Fastened resources: can be used but at a high cost (e.g., local databases, web sites)
    - Fixed resources: cannot be moved (e.g., local devices



Resource-to machine binding

Process-to-resource binding		Unattached	Fastened	Fixed
	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV, GR)	GR (or CP)	GR
	By type	RB (or GR, CP)	RB (or GR, CP)	RB (or GR)

- Actions to be taken with respect to the references to local resources when migrating code
  - GR: establish a global system wide reference
  - MV: move the resource
  - CP: copy the value of resource
  - RB: rebind the process to locally available resource

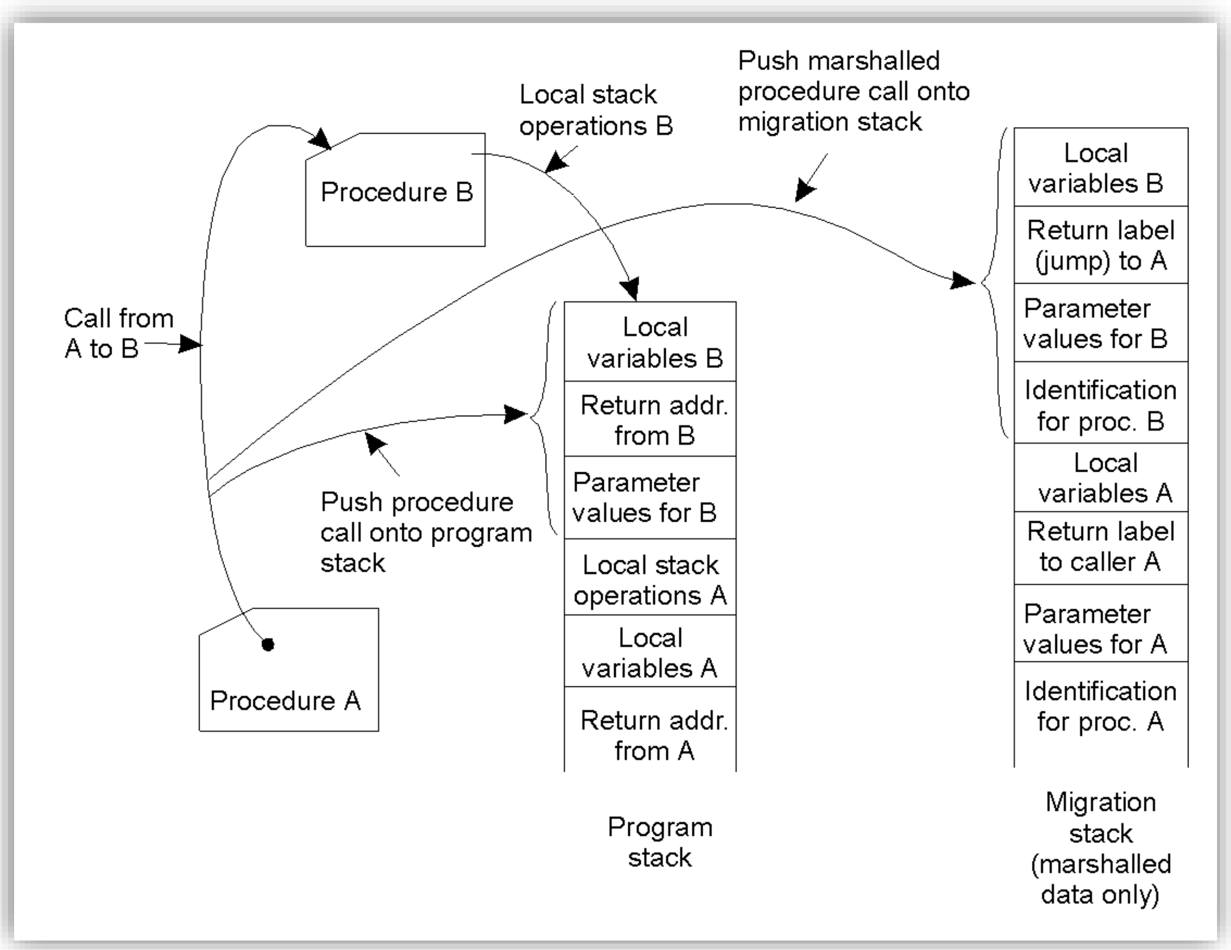
Code Migration in Heterogeneous Systems

Objectives

- Code Migration in Heterogenous System.
- Weak Mobility in D'Agents
- Strong Mobility in D'Agents.

Code Migration in Heterogeneous Systems

- Maintain a migration stack in an independent format
- Migrate only at certain points in the program (e.g., before/after calling a procedure)



Weak Mobility in D'Agents

A Tel agent in D'Agents submitting a script to a remote machine (adapted from [Gray '95])

```
proc factorial n {
   if ($n ≤ 1) { return 1; }                # fac(1) = 1
  expr $n * [ factorial [expr $n – 1] ]           # fac(n) = n * fac(n – 1)
}
set number …  # tells which factorial to compute
set machine …             # identify the target machine
agent_submit $machine –procs factorial –vars number –script {factorial $number }
agent_receive …          # receive the results (left unspecified for simplicity)
```

Strong Mobility in D'Agents

A Tel agent in D'Agents migrating to different machines where it executes the UNIX *who* command (adapted from [Gray 95])

all_users $machines

```
proc all_users machines {
   set list ""                      # Create an initially empty list
   foreach m $machines {       # Consider all hosts in the set of given machines
      agent_jump $m              # Jump to each host
      set users [exec who]       # Execute the who command
      append list $users         # Append the results to the list
   }
   return $list              # Return the complete list when done
}
set machines …                      # Initialize the set of machines to jump to
set this_machine                    # Set to the host that starts the agent
# Create a migrating agent by submitting the script to this machine, from where
# it will jump to all the others in  $machines.
agent_submit  $this_machine –procs all_users
                            -vars  machines
                            -script { all_users $machines }
agent_receive …                     #receive the results  (left unspecified for simplicity)
```

Use of Virtual Machines to Handle the Heterogeneity

Objectives

- Virtual Machines to Handle the Heterogeneity.
- Migrating the Entire Memory Image
- Migrating Bindings to Local Resources

Use of Virtual Machines to Handle the Heterogeneity

“Virtual machines (VMs) can be a useful tool for handling heterogeneity in computing environments. VMs allow for the creation of multiple virtualized instances of operating systems and applications, which can be run on a single physical machine. This allows for the consolidation of multiple computing environments onto a single machine, reducing the need for multiple physical machines with different configurations.”

Virtual machines (VMs) can be used to handle heterogeneity in distributed computing environments. In distributed computing, different nodes in the network may have different hardware configurations, operating systems, and software environments, which can make it challenging to develop and deploy applications that work consistently across all nodes.

By using VMs, distributed computing systems can create virtualized instances of a consistent operating system and application environment that can be deployed on any node in the network. This allows applications to be developed and tested on a single virtualized environment and then deployed across multiple nodes in the network, without needing to worry about the heterogeneity of the underlying hardware and software configurations.

In addition, VMs can be used to facilitate the migration of applications across different nodes in the network. For example, if a node fails or needs to be replaced, the VM can be easily migrated to a new node with minimal disruption to the application.

Let us consider one specific example of migrating virtual machines, as discussed in Clark et al. (2005). In this case, the authors concentrated on real-time migration of a virtualized operating system, typically something that would be convenient in a cluster of servers where a tight coupling is achieved through a single, shared local-area network. Under these circumstances migration involves two major problems:

- Migrating the entire memory image.
- Migrating bindings to local resources

Migrating the Entire Memory Image

There are, in principle, three ways to handle migration:

- Pushing memory pages to the new machine and resending the ones that are later modified during the migration process.
- Stopping the current virtual machine; migrate memory and start the new virtual machine.
- Letting the new virtual machine pull in new pages as needed, that is, let processes start on the new virtual machine immediately and copy memory pages on demand.

Migrating Bindings to Local Resources

- Concerning local resources, matters are simplified when dealing only with a cluster server. First, because there is a single network, the only thing that needs to be done is to announce the new network-to-MAC address binding, so that clients can contact the migrated processes at the correct network interface.
- Finally, if it can be assumed that storage is provided as a separate tier, then migrating binding to files is similarly simple. The overall effect is that, instead of migrating processes, we now actually see that an entire operating system can be moved between machines.

Week 10

Introduction To Message Passing

Objectives

- Introduction to Message Passing.
- Message Passing Model

Introduction to Message Passing

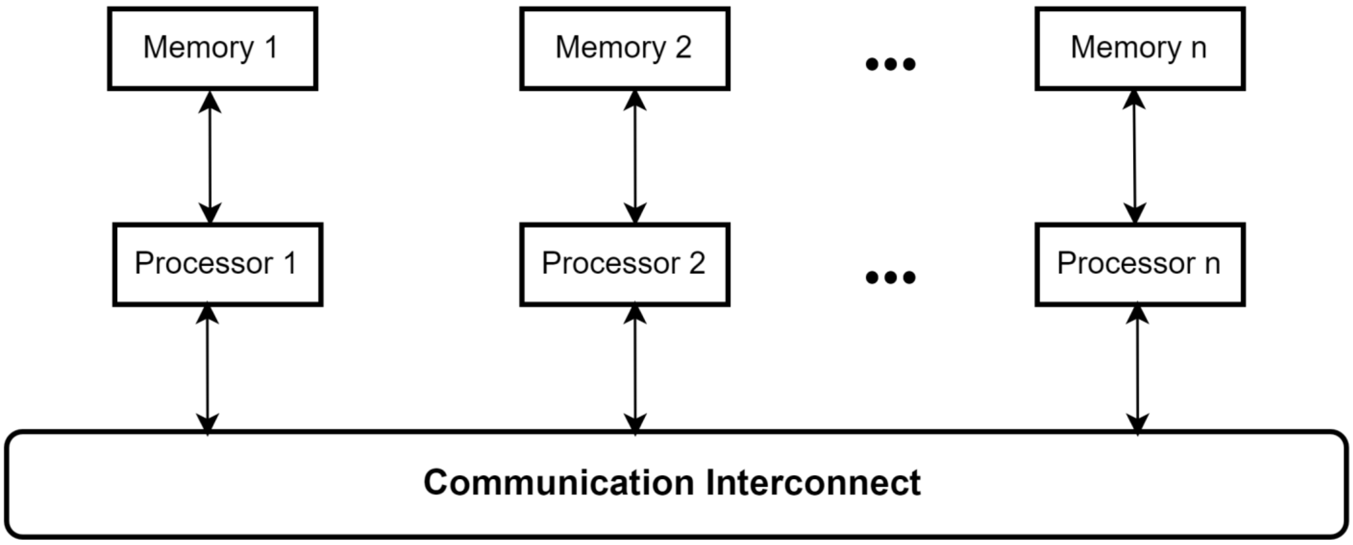
Message passing is a method of communication in distributed systems where processes or objects exchange messages with each other to share information or coordinate activities."

Message Passing Model

- Message passing is the most commonly used parallel programming approach in distributed memory systems. Here, the programmer has to determine the parallelism. In this model, all the processors have their own local memory unit and they exchange data through a communication network.
- There are two main types of message passing: synchronous and asynchronous.

Processors use message-passing libraries for communication among themselves. Along with the data being sent, the message contains the following components:

- The address of the processor from which the message is being sent.
- Starting address of the memory location of the data in the sending processor.
- Data type of the sending data.
- Data size of the sending data.
- The address of the processor to which the message is being sent.
- Starting address of the memory location for the data in the receiving processor.



The message passing model demonstrates the following characteristics:

- A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

What is Message Passing Interface (MPI)?

Objectives

- Understanding of Message Passing Interface (MPI).
- History and Versions of MPI.

Message Passing Interface

- A process is (traditionally) a program counter and address space

- Processes may have multiple threads (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Inter-process communication consists of:
    - Synchronization
    - Movement of data from one process's address space to another's.

Types of Parallel Computing Models:

- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
- Task Parallel - different instructions on different data (MIMD)
- SPMD (single program, multiple data) not synchronized at individual operation level
- SPMD is equivalent to MIMD since each MIMD program can be made SPMD (similarly for SIMD, but not in practical sense.)

Message passing (and MPI) is for MIMD/SPMD parallelism. HPF is an example of the SIMD interface

- Standardized message passing library specification (IEEE):
    - For parallel computers, clusters and heterogeneous networks.
    - Not a specific product, compiler specification etc.
    - Many implementations, MPICH, LAM, OpenMPI …
- Portable, with Fortran and C/C++ interfaces.
- Many functions.
- Real parallel programming.
- Notoriously difficult to debug.

History and Versions of MPI

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded in source code. The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
- Part 1 of the Message Passing Interface (MPI) was released in 1994. Part 2 (MPI-2) was released in 1996. Both MPI specifications are available on the web at www.mcs.anl.gov/Projects/mpi/standard.html.

MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI. A few offer a full implementation of MPI-2.

For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons.

Library Interface

Objectives

- Library Message-Passing
- MPI Basics Program

Library Interface

MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.

A message-passing library specifications

    - Extended message-passing model.
    - Not a language or compiler specification.
    - Not a specific implementation or product.

- MPI provides a powerful, efficient, and portable way to express parallel programs.
- MPI was explicitly designed to enable libraries which may eliminate the need for many users to learn (much of) MPI.
- Portable.
- Good way to learn about subtle issues in parallel computing.

  MPI provides point-to-point communication.

  Collective operations

    - Barrier synchronization
    - Gather/scatter operations
    - Broadcast, reductions

  Predefined and derived datatypes

  Virtual topologies

  C/C++ and Fortran bindings.

How big is the MPI library?

- Huge (125 Functions).
- Basic (6 Functions).

Where to get MPI library?

- Standard message-passing library includes best of several previous libraries.
- MPICH (WINDOWS / UNICES)
    - http://www-unix.mcs.anl.gov/mpi/mpich/
- Open MPI (UNICES)
    - http://www.open-mpi.org/

MPI Basics

Many parallel programs can be written using just these six functions, only two of which are non-trivial;

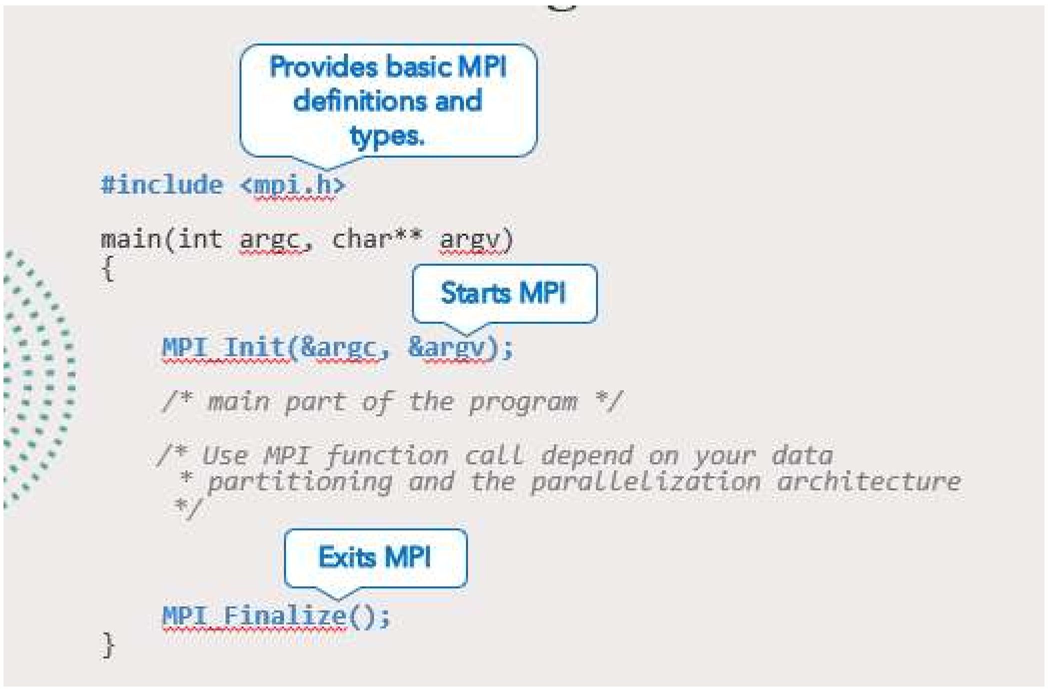
– MPI_INIT

– MPI_FINALIZE

– MPI_COMM_SIZE

- MPI_COMM_RANK
- **MPI_SEND**
- **MPI_RECV**

Skeleton MPI Program



```
#include <mpi.h>

main(int argc, char** argv)
{

    MPI_Init(&argc, &argv);

    /* main part of the program */

    /* Use MPI function call depend on your data
     * partitioning and the parallelization architecture
     */

    MPI_Finalize();
}
```

MPI Program Example

#include “mpi.h”

#include <stdio.h>

int main(int argc, char *argv[])

{

MPI_Init(&argc, &argv);

printf(“Hello, world!\n”);

MPI_Finalize();

return 0;

}

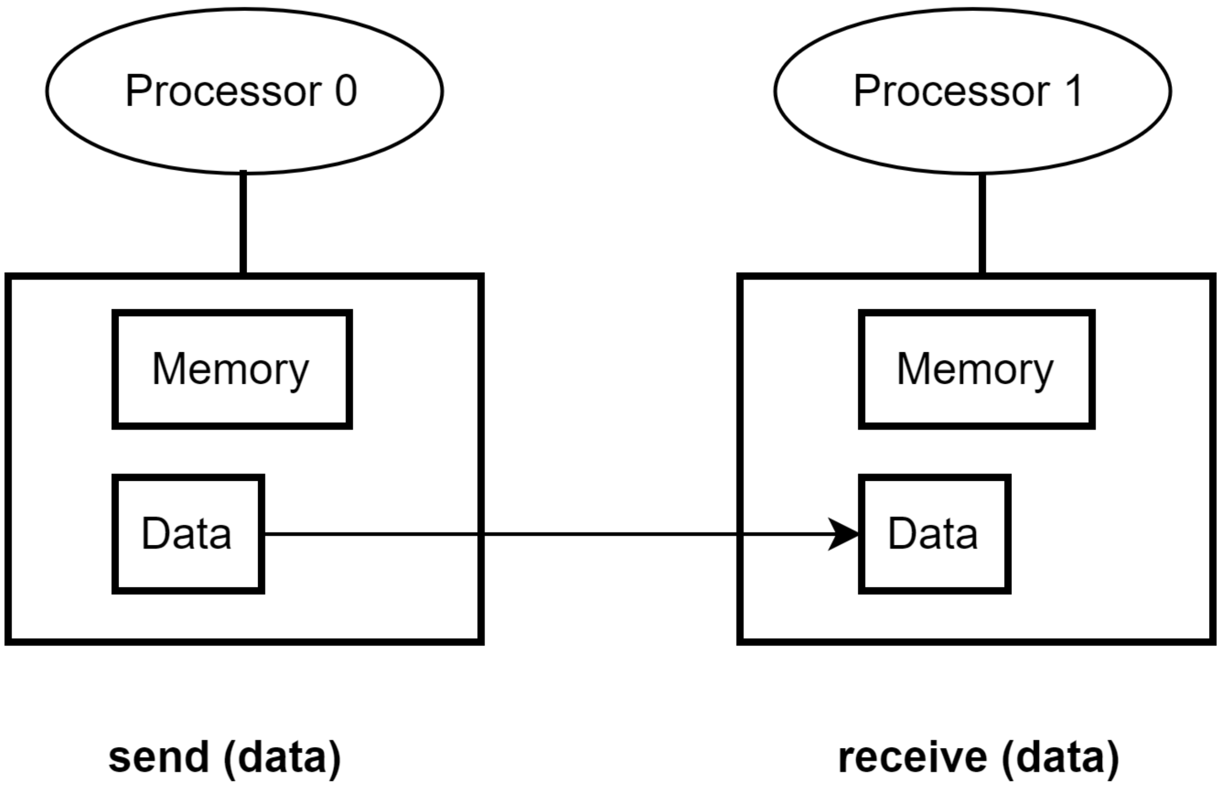
Message Passing Programming Modes

Objectives

- What is Message Passing Programming?
- Message Passing Programming Modes.

Message Passing Programming

- Distributed memory processes have access only to local data. The sender process issues a send call, and the receiver process issues a matching receive call.
- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
- All interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data
- These two constraints, while onerous, make underlying costs very explicit to the programmer.



Message Passing Programming Modes

- Different communication modes that can be used for message passing programming are:
    - Synchronous/asynchronous
    - Blocking/non-blocking
    - Buffered/unbuffered

Non-blocking: A routine is non-blocking if it is guaranteed to complete regardless of external events (e.g., the other processors). Example: A send is non-blocking if it is guaranteed to return whether or not there is a matching receive.

Blocking: A routine is blocking if its completion (return of control to the calling routine) may depend on an external event (an event that is outside the control of the routine itself). Example: A send is blocking if it does not return until there is a matching receive.

- **Asynchronous**: A routine is asynchronous if it initiates an operation that happens logically outside the flow of control of the calling process. The important practical distinction is whether the program may be required to check for completion of the operation before proceeding.
- **Synchronous:** A routine is synchronous if its operation happens within the flow of control of the calling process.
- **Buffered:** A routine that uses buffered message passing is a program that sends and receives messages using a buffer. In this mode, messages are queued in the buffer until they are ready to be sent or received
- **Unbuffered:** A routine may be used to perform a specific message-passing task, such as sending or receiving a message between two processes. Does not use a buffer. Messages are sent and received immediately without any buffering. This mode is useful when low latency is required.

Asynchronous/Synchronous Message Passing

Objectives

- Understanding of Synchronous Message Passing.
- Understanding of Asynchronous Message Passing.

Asynchronous/Synchronous Message Passing

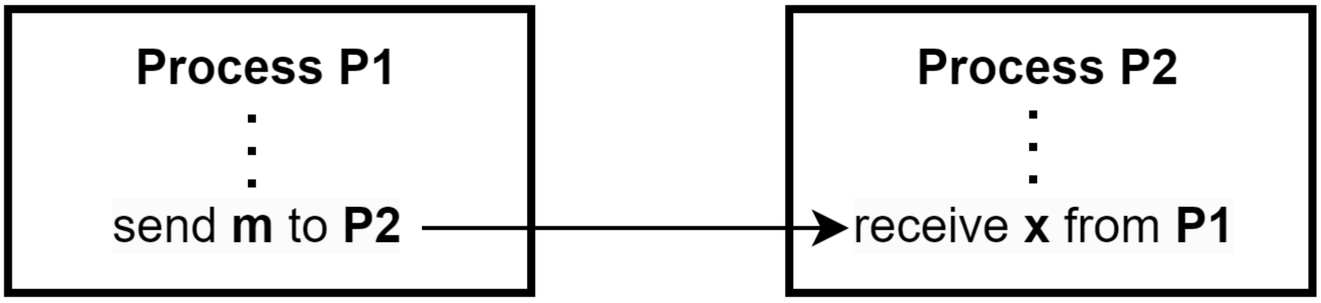
- Message-passing programs are often written using the asynchronous or loosely synchronous paradigms.
    - In the asynchronous paradigm, all concurrent tasks execute asynchronously.

    - In the loosely synchronous model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.

Synchronous Message Passing

- A synchronous communication is not complete until the message has been received.
    - Completes once ack is received by sender.
- Communication upon synchronization
- Hoare's Communicating Sequential Processes (1978).
- BLOCKING send and receive operations.
    - Unbuffered communication.
    - Several steps in protocol- synchronization, data movement, completion.
    - Delays participating processes.

Synchronous Message Passing



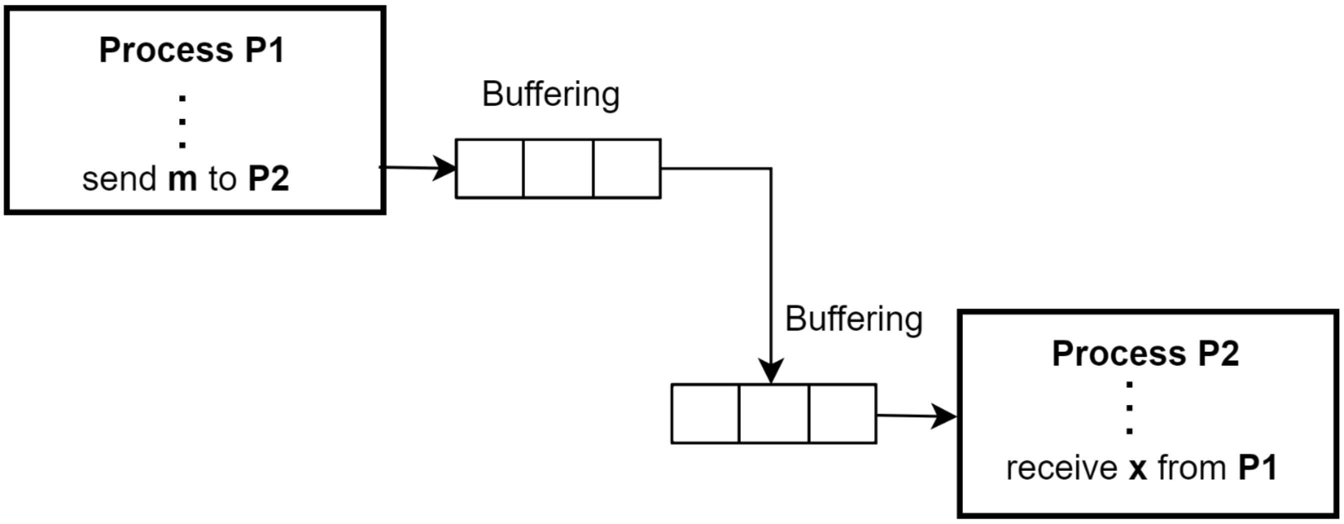
Asynchronous Message Passing

- **Asynchronous Communication:** An asynchronous communication completes before the message is received. It has three modes:
- Standard send: completes once the message has been sent, which may or may not imply that the message has arrived at its destination.
- Buffered send: completes immediately, if receiver not ready, MPI buffers the message locally.
- Ready send: completes immediately, if the receiver is ready for the message it will get it, otherwise the message is dropped silently.

Asynchronous Message Passing

- Buffered communication
  - May increase concurrency (e.g. producer/consumer).
  - May increase transit time.
- Send operation
  - Send operation completes when message is completely copied to buffer.
  - Generally non-blocking but will block if buffer is full
- Receive operation – two flavors
  - BLOCKING
    - Receive operation completes when message has been delivered.
  - NON-BLOCKING
    - Receive operation provides location for message.
    - Notified when receive complete (via flag or interrupt).

Asynchronous Message Passing



Benefits of the Message Passing Interface

Objectives

- Understanding of Benefits of MPI.

Benefits of the Message Passing Interface

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.

- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.
- **Functionality** - Over 115 routines are defined in MPI-1 alone.
- **Scalability-** MPI is designed to scale to large numbers of processors, which makes it well-suited for high-performance computing.
- **Availability** - A variety of implementations are available, both vendor and public domain.

Week 11

What is Thread?

Objectives

- Introduction of Threading
- Creating and Terminating Threads using POSIX API

What is Thread?

- In general
- "A long, thin strand of cotton, fibers etc.,"
- In Computing
- "A sequence of linked instructions"
- movl $14, %eax

```
 movl    $10,    %ebx
   add    %eax,    %ebx
```

Threads in context of CPU Utilization:

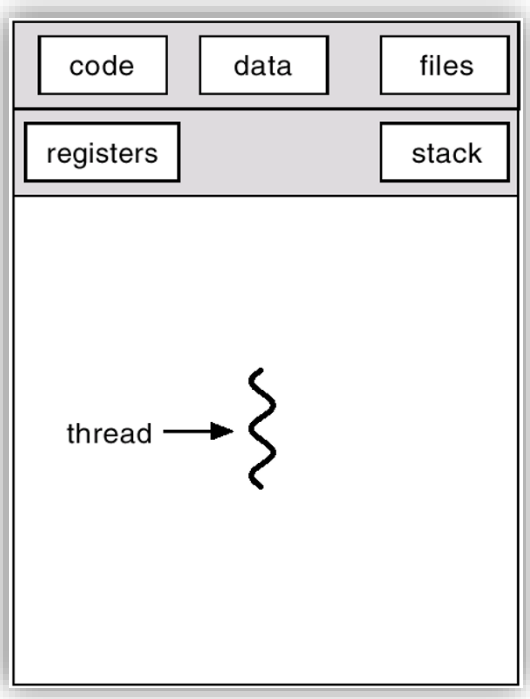
"A thread is a basic unit of CPU utilization having…"

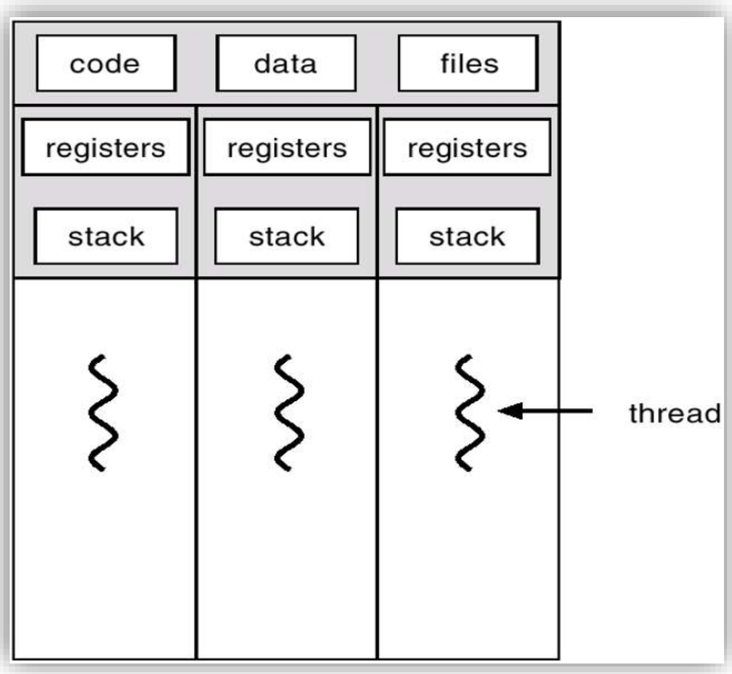
- Thread ID
- CPU Context
- Stack
- Priority
- Errno.

Threads in context of process

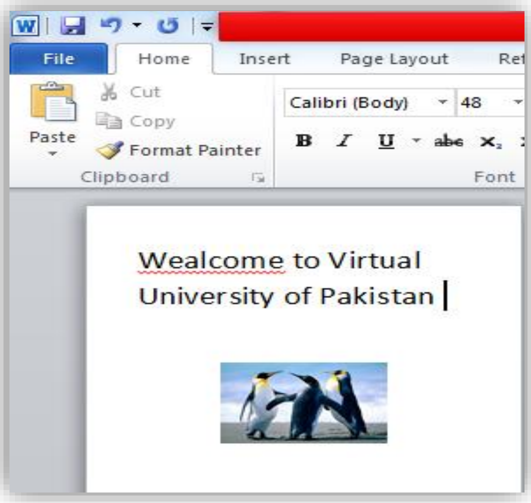
- A thread is a piece of code within the process
- Executes within the address space of a process
- Lightweight process
- Can be scheduled to run on a CPU as an independent unit and terminate
- Multiple threads can run simultaneously .

Single thread vs Multiple threads





Threads Example



Wealcome to Virtual University of Pakistan

Word Processor

Background thread may check spelling and grammar

Another thread does periodic automatic backups of the file being edited

A third thread loads images from the hard drive, and

Foreground thread processes user input (keystrokes

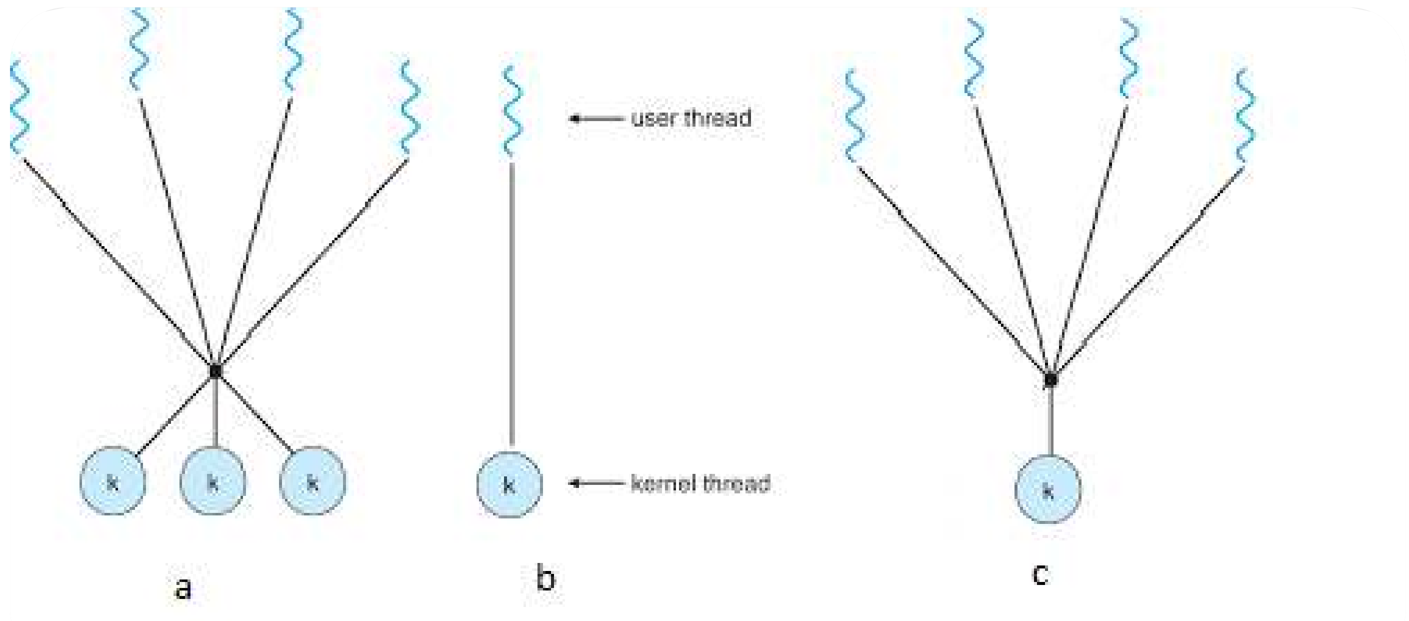
Thread Model?

Objectives

- Introduction of Thread Model..
- Logical Machine Model of Threads

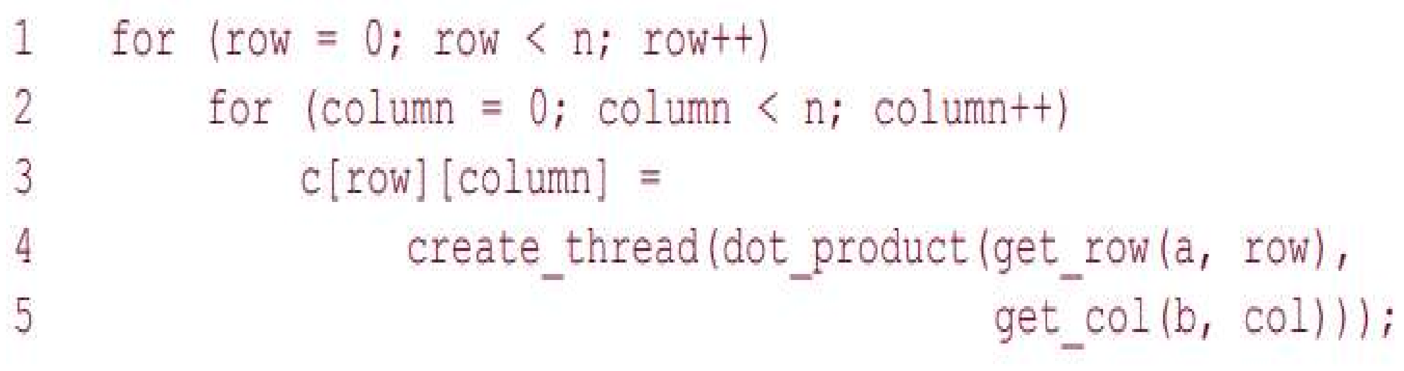
Thread Models

- **Many to Many**
- **One to One**
- **Many to One**



What is Thread?

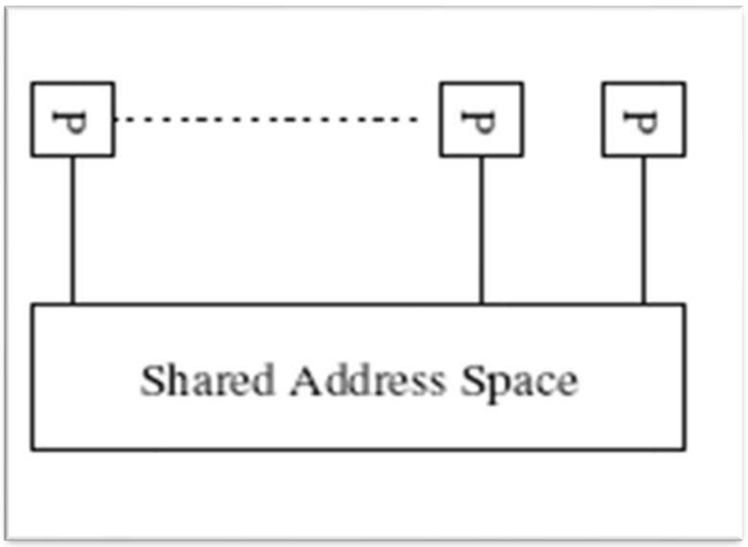
- **Thread Example**



```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(dot_product(get_row(a, row),
                                      get_col(b, col)));
```

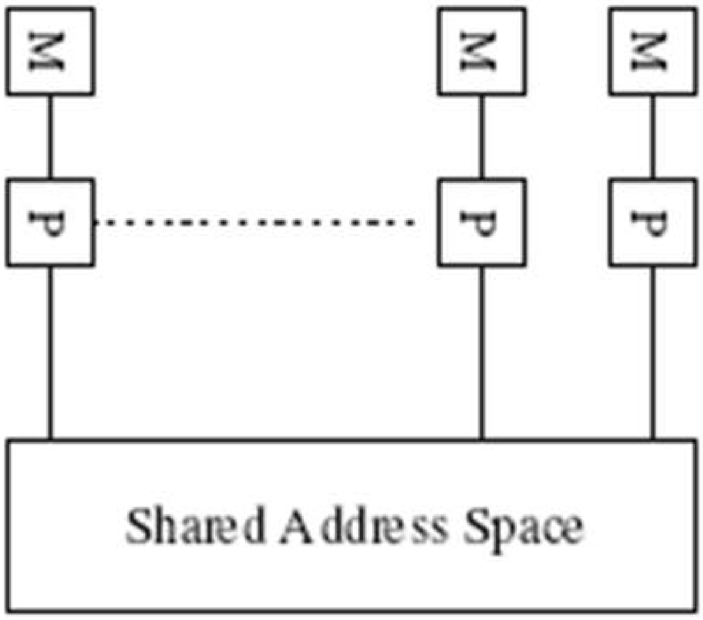
What is Thread?

All memory in the logical machine model of a thread is globally accessible to every thread



Logical Machine Model of Threads

Threads are invoked as function calls



Thread API?

Objectives

- What Is Thread API?
- Thread Creation

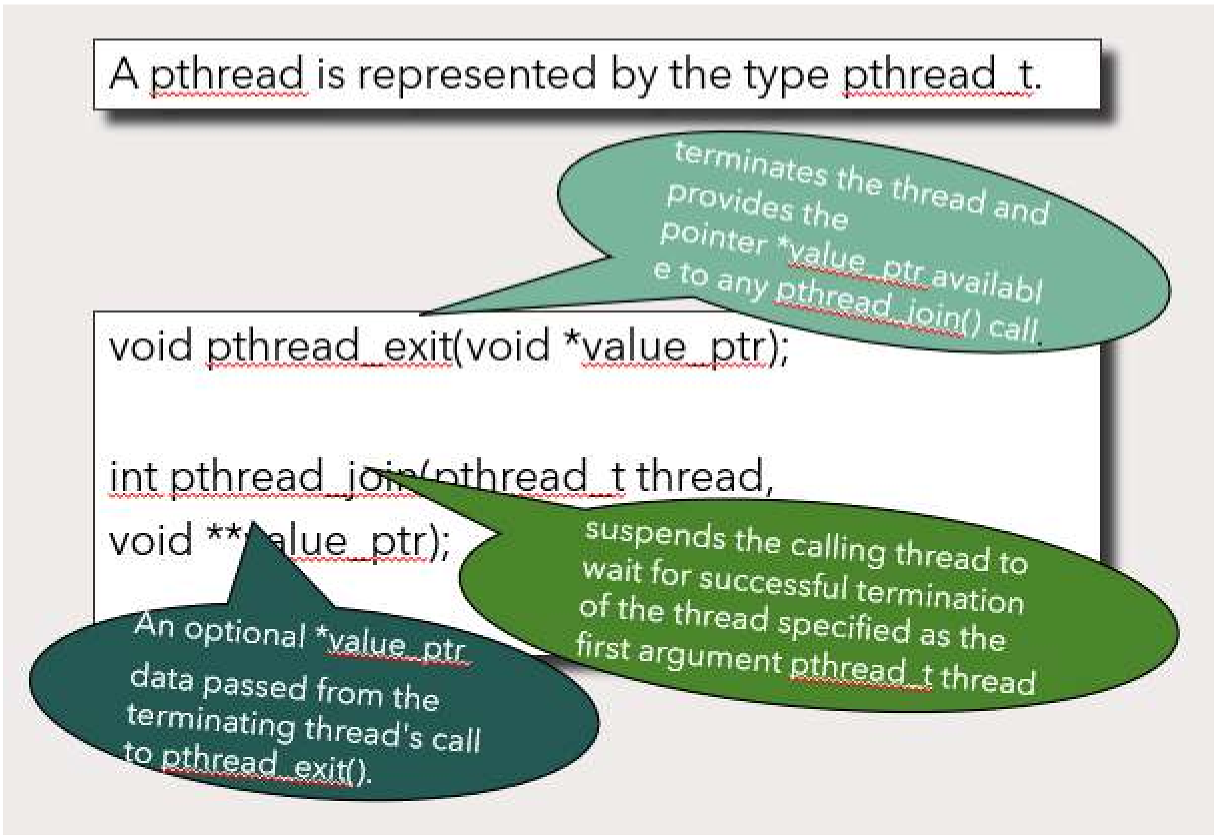
The POSIX Thread API

A number of vendors provide vendor-specific thread APIs (NT threads, Solaris threads, Java threads, etc.)

The IEEE specifies a standard 1003.1c-1995, POSIX API, also referred to as Pthreads

POSIX has emerged as the standard threads API, supported by most vendors.

Thread Termination



A pthread is represented by the type pthread_t.

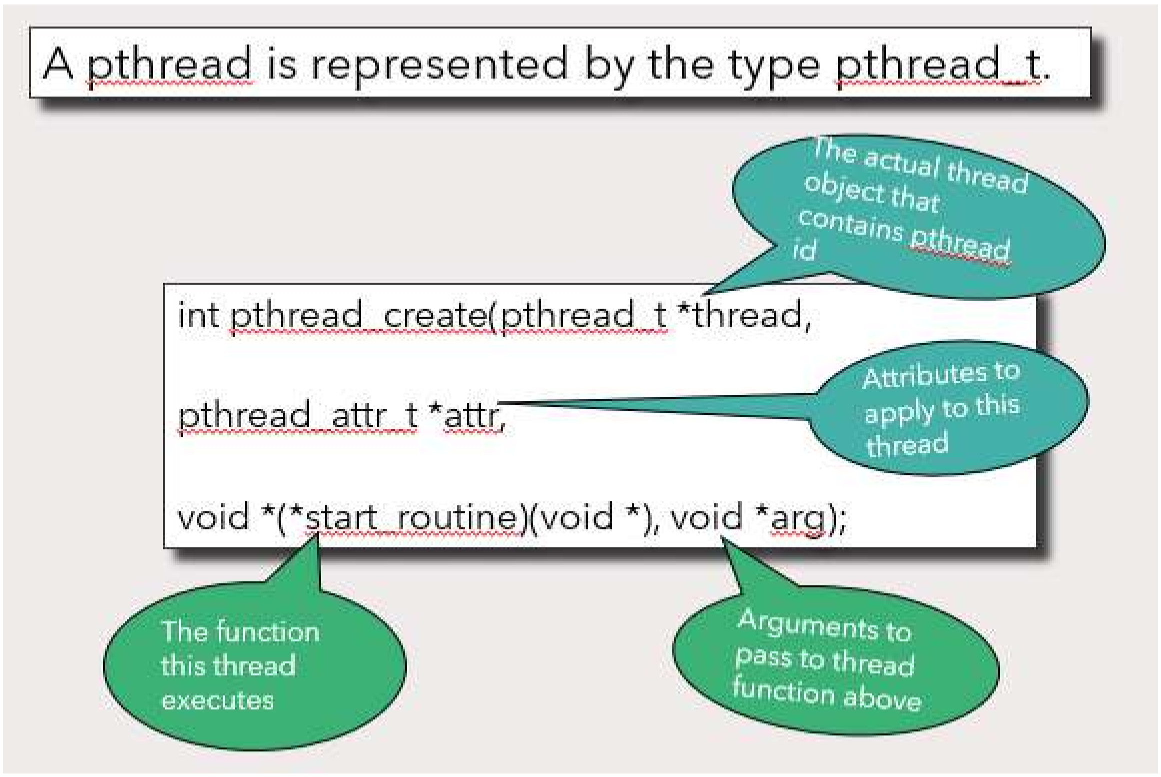
```
void pthread_exit(void *value_ptr);

int pthread_join(pthread_t thread,
void **value_ptr);
```

terminates the thread and provides the pointer *value_ptr available to any pthread_join() call.

suspends the calling thread to wait for successful termination of the thread specified as the first argument pthread_t thread

An optional *value_ptr data passed from the terminating thread's call to pthread_exit().

Thread Creation



A pthread is represented by the type pthread_t.

```
int pthread_create(pthread_t *thread,

pthread_attr_t *attr,

void *(*start_routine)(void *), void *arg);
```

The actual thread object that contains pthread id

Attributes to apply to this thread

The function this thread executes

Arguments to pass to thread function above

Why Thread?

Objectives

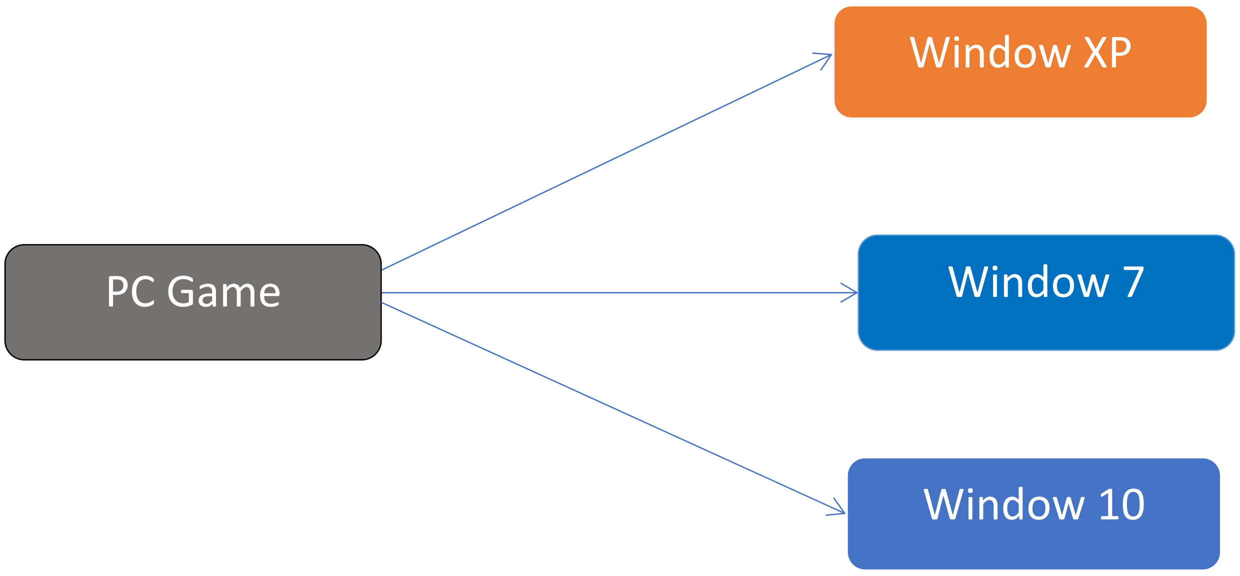
- Why Threads?

- Characteristics of Threads

Why Thread?

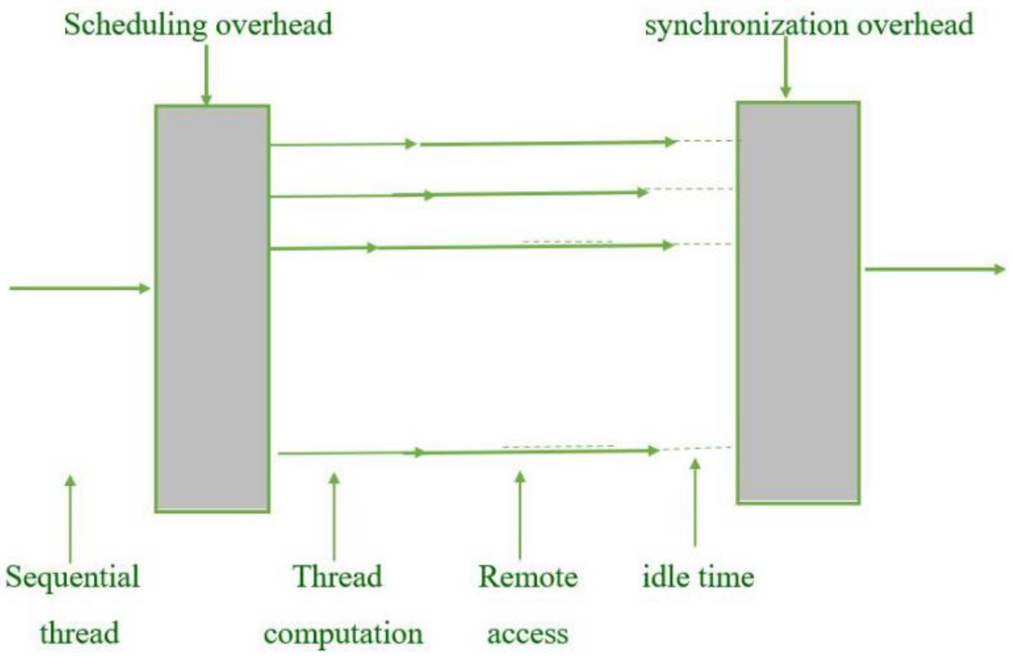
Portability:

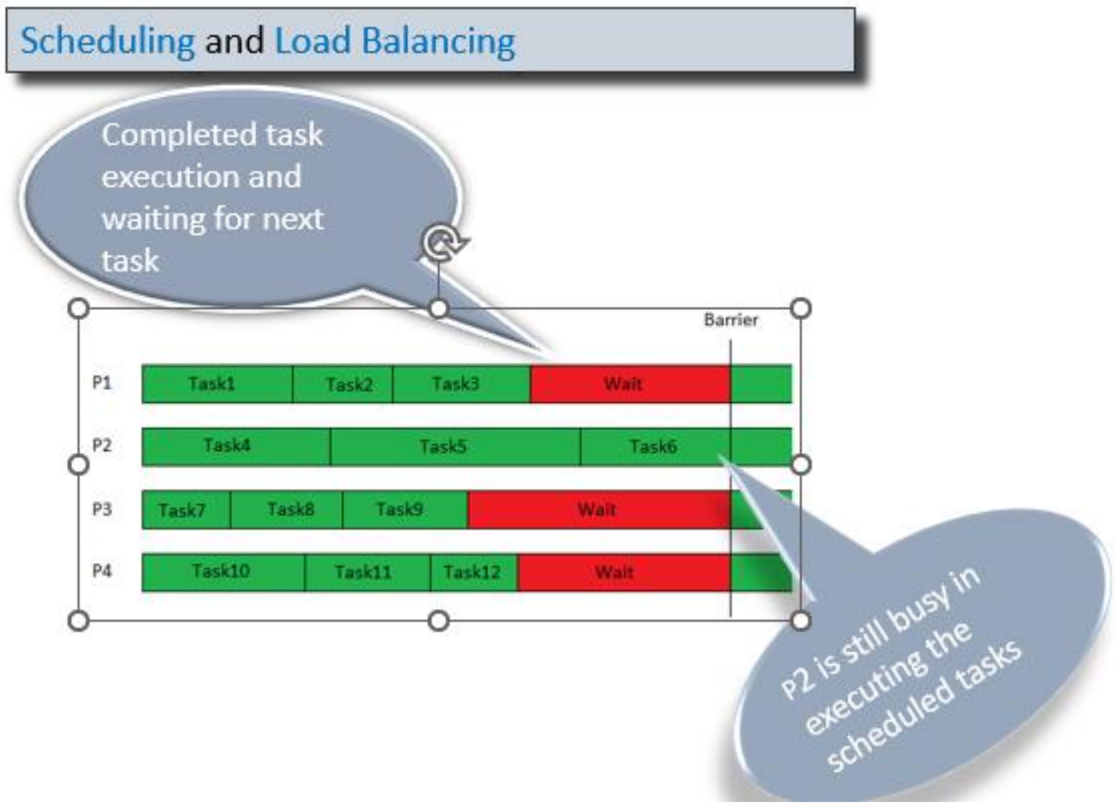
The possibility to use the same software in different environments



Latency hiding

Multithreading enables latency reduction/hiding




Scheduling and Load Balancing
Completed task execution and waiting for next task
Barrier
P1 Task1 Task2 Task3 Wait
P2 Task4 Task5 Task6
P3 Task7 Task8 Task9 Wait
P4 Task10 Task11 Task12 Wait
P2 is still busy in executing the scheduled tasks

Ease of Programming

Threaded programs are significantly easier to write than corresponding programs using message passing APIs.

Widespread Use

The widespread acceptance of the POSIX thread API, development tools for POSIX threads are more widely available and stable.

Thread Synchronization

Objectives

- Introduction of Thread Synchronization
- Logical Machine Model of Threads.

What is the biggest challenge with using thread?

Communication is implicit in shared-address-space programming.

Much of the effort associated with writing correct threaded programs is spent on synchronizing concurrent threads with respect to their data accesses or scheduling.

Synchronization

What will happen when multiple threads attempt to manipulate the same data item, if proper care is not taken to synchronize?

Can results incoherent

Synchronization Primitives of threads

Consider:

if (my_cost < best_cost)

best_cost = my_cost;

best_cost = 100

t1 = 50

t2 = 75

After time interval T, the best_cost =?

Mutual Exclusion

The code in the previous example corresponds to a critical segment; i.e.,

“A segment that must be executed by only one thread at any time”.

- Critical segments in Pthreads are implemented using mutex locks

Mutex-locks have two states:.

- locked and
- Unlocked

At any point of time, only one thread can lock a mutex lock.

Synchronization Primitives of threads

- A lock is an atomic operation
- A thread entering a critical segment first tries to get a lock
- It goes ahead when the lock is granted

Mutex Locks

Objectives

- What are Mutex Locks?
- Uses and types of mutex

Mutual Exclusion

The Pthreads API functions for handling mutex-locks:

- int pthread_mutex_lock (pthread_mutex_t *mutex_lock);
- int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);
- int pthread_mutex_init (pthread_mutex_t *mutex_lock, const pthread_mutexattr_t *lock_attr);

Producer-Consumer Using Mutex Locks

The producer-consumer scenario imposes the following constraints:

The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread

The consumer threads must not pick up tasks until there is something present in the shared data structure

Individual consumer threads should pick up tasks one at a time.

Types of mutex

- A normal mutex
- A recursive mutex
- Error check mutex

Conditional Variable

Objectives

- Problems with mutex
- Introduction to conditional Variable

Locking Overhead

- Performance issue
- Serialization issue

Locking overhead: Idling overheads

- It is often possible to reduce the idling overhead associated with locks using an alternate function, pthread_mutex_trylock
- Int pthread_mutex_trylock (pthread_mutex_t *mutex_lock);
- pthread_mutex_trylock is typically much faster than pthread_mutex_lock on typical systems

Conditional Variable in Synchronization

Problem with trylock:

trylock introduces the overhead of polling for availability of locks.

Solution:

Condition Variable

What is conditional variable?

"A condition variable is a data object used for synchronizing threads. This variable allows a thread to block itself until specified data reaches a predefined state",

Conditional Variable in Synchronization

A condition variable is associated with a predicate.

- When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition

A single condition variable may be associated with more than one predicate

A condition variable always has a mutex associated with it.

A thread locks this mutex and tests the predicate defined on the shared variable.

If the predicate is not true, the thread waits on the condition variable associated with the predicate using the function:

pthread_cond_wait.

Pthreads functions for condition variables

- int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
- int pthread_cond_signal(pthread_cond_t *cond);
- int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

- int pthread_cond_destroy(pthread_cond_t *cond);
- int pthread_cond_broadcast(pthread_cond_t *cond);

Week 12

Principles of Parallel Algorithm Design

Objectives

- Introduction to Parallel Algorithms.
- Key steps and units in the design of parallel algorithms

Parallel Algorithms

Algorithm: "A prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps"

Algorithms in which several operations may be executed simultaneously are referred to as parallel algorithms. In general, a parallel algorithm can be defined as a set of processes or tasks that may be executed simultaneously and may communicate with each other in order to solve a given problem.

What are the key steps in design of Parallel Algorithms?

- Assigning them to different processors for parallel execution
- Dividing a computation into smaller computations

What is decomposition?

"The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel, is called decomposition."

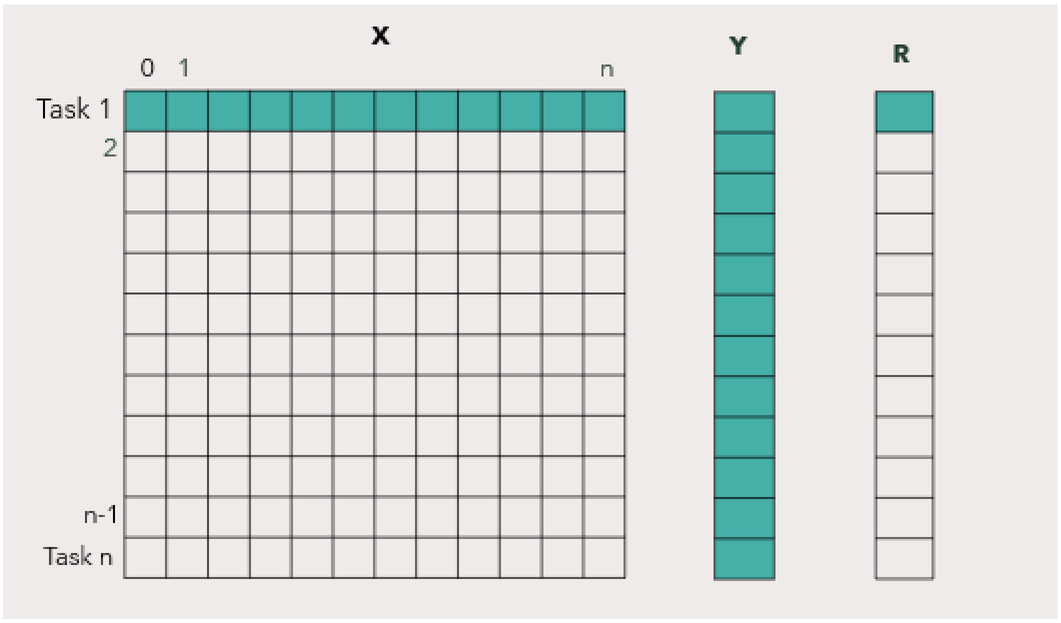
What are task?

"Tasks are programmer-defined units of computation into which the main computation is subdivided by means of decomposition."

Types of tasks in term of interdependency?

- Dependent tasks
- Independent tasks

Example decomposition: Dense matrix-vector Multiplication



Task dependency graph

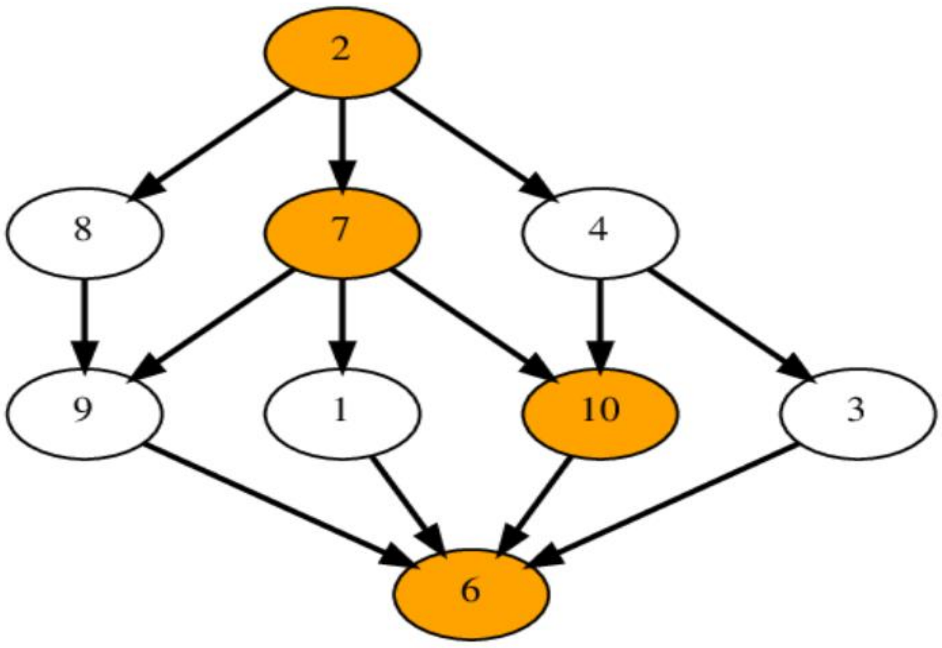
Objectives

- Introduction task dependency graph.
- Examples of task dependency graph

What is a task dependency graph?

- Decomposition can be illustrated in the form of a directed graph with:
    - Nodes corresponding to tasks and
    - Edges indicating that the result of one task is required for processing the next.
    - Such a graph is called a task dependency graph.

Example of task dependency graph?



What is critical path length?

- A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other
- The longest such path determines the shortest time in which the program can be executed in parallel.
- The length of the longest path in a task dependency graph is called the critical path length.

Example TDG: Database Query Processing:

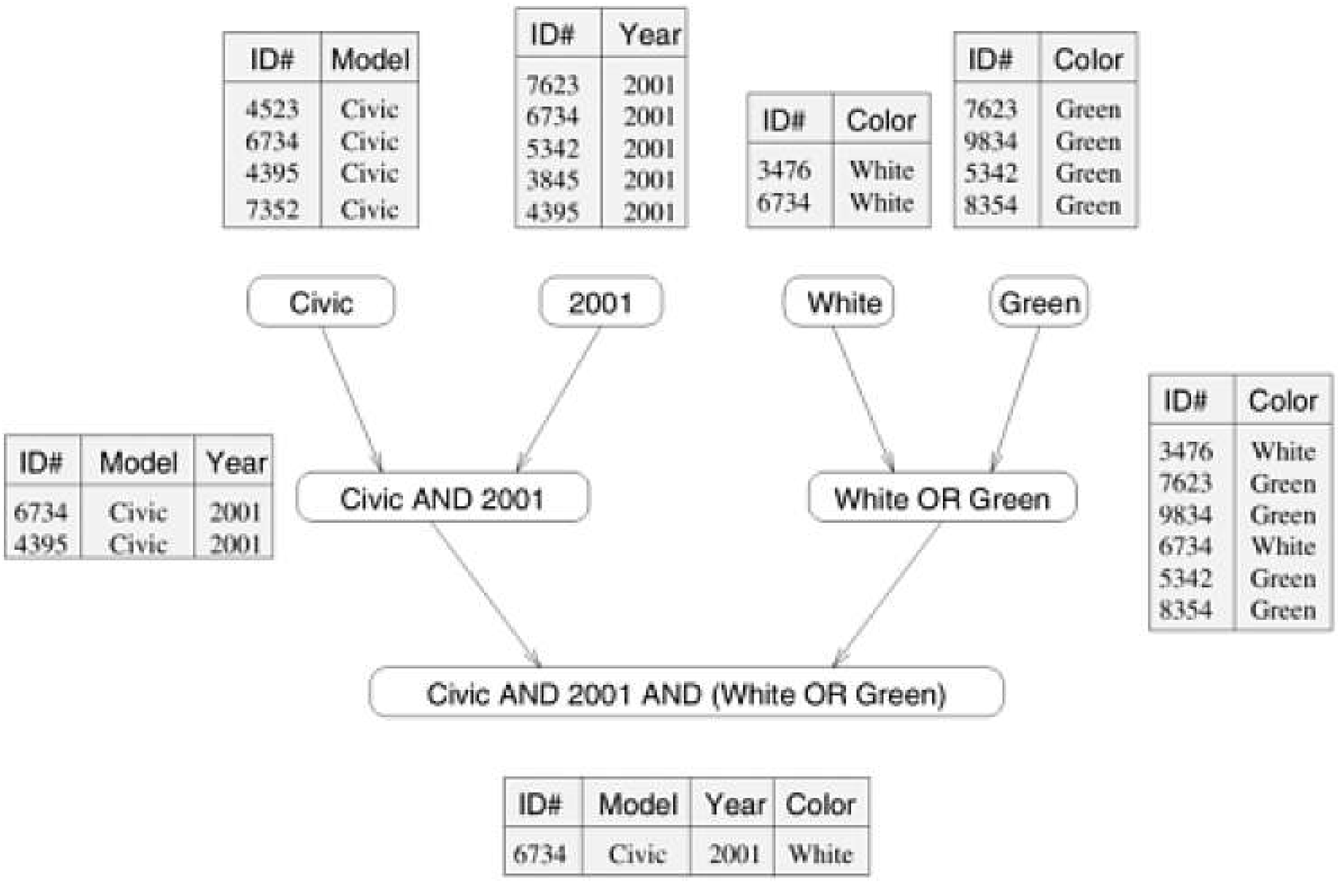
Consider the execution of the query:

“MODEL = ``CIVIC'' AND YEAR = 2001 AND (COLOR = ``GREEN'' OR COLOR = ``WHITE)”

on the given database:

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Example: Database Query Processing



Granularity of Task Decompositions

Objectives

- Introduction granularity of task decomposition.
- Examples of granularity of task decomposition

Granularity of Task Decompositions (Task size)

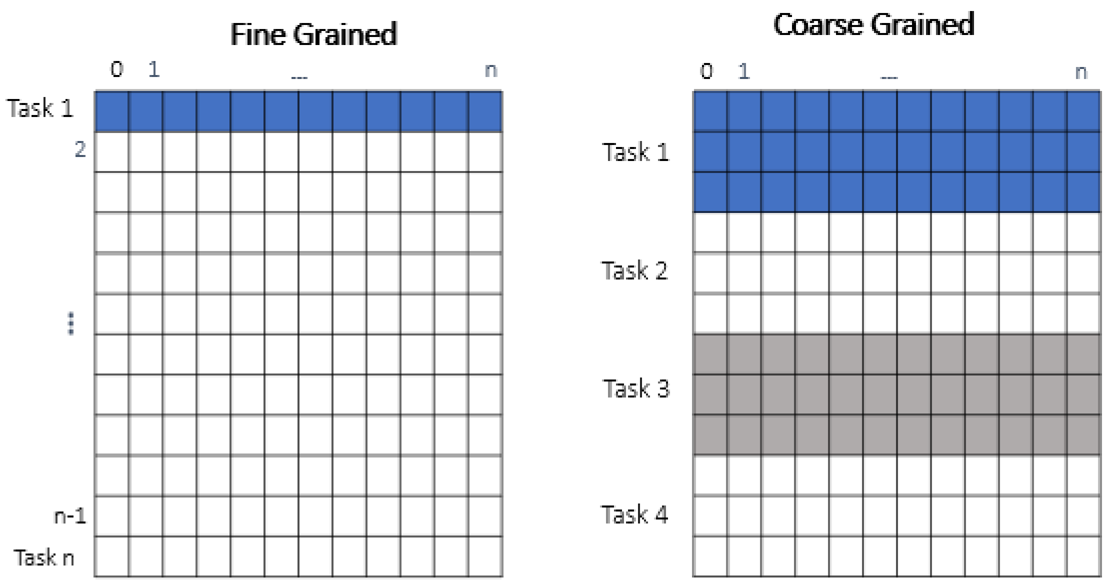
Fined-grained

- Decomposition of computation into a large number of tasks results in fine-grained decomposition.

Coarse Grained

- Decomposition of computation into a small number of tasks results in a coarse grained decomposition

Example: Granularity of Tasks



Granularity of Task Decompositions (Concurrency)

- The number of tasks that can be executed in parallel is the degree of concurrency of a decomposition
- The maximum degree of concurrency is the maximum number of such tasks at any point or time during execution.
- The average degree of concurrency is the average number of tasks that can be processed in parallel over the execution of the program

Is there any limit on parallel performance?

- Do you think that finer decomposition of tasks always results in small time
- There is an inherent bound on how fine the granularity of a computation can be
- Concurrent tasks may also have to exchange data with other tasks.
- This results in communication overhead.

Decomposition Techniques

Objectives

- Classification of decomposition techniques.
- When to use what?
- What is recursive decomposition technique?

What are the decomposition techniques?

“There is no single recipe that works for all problems, we present a set of commonly used techniques that apply to broad classes of
problems.”

Decomposition Techniques

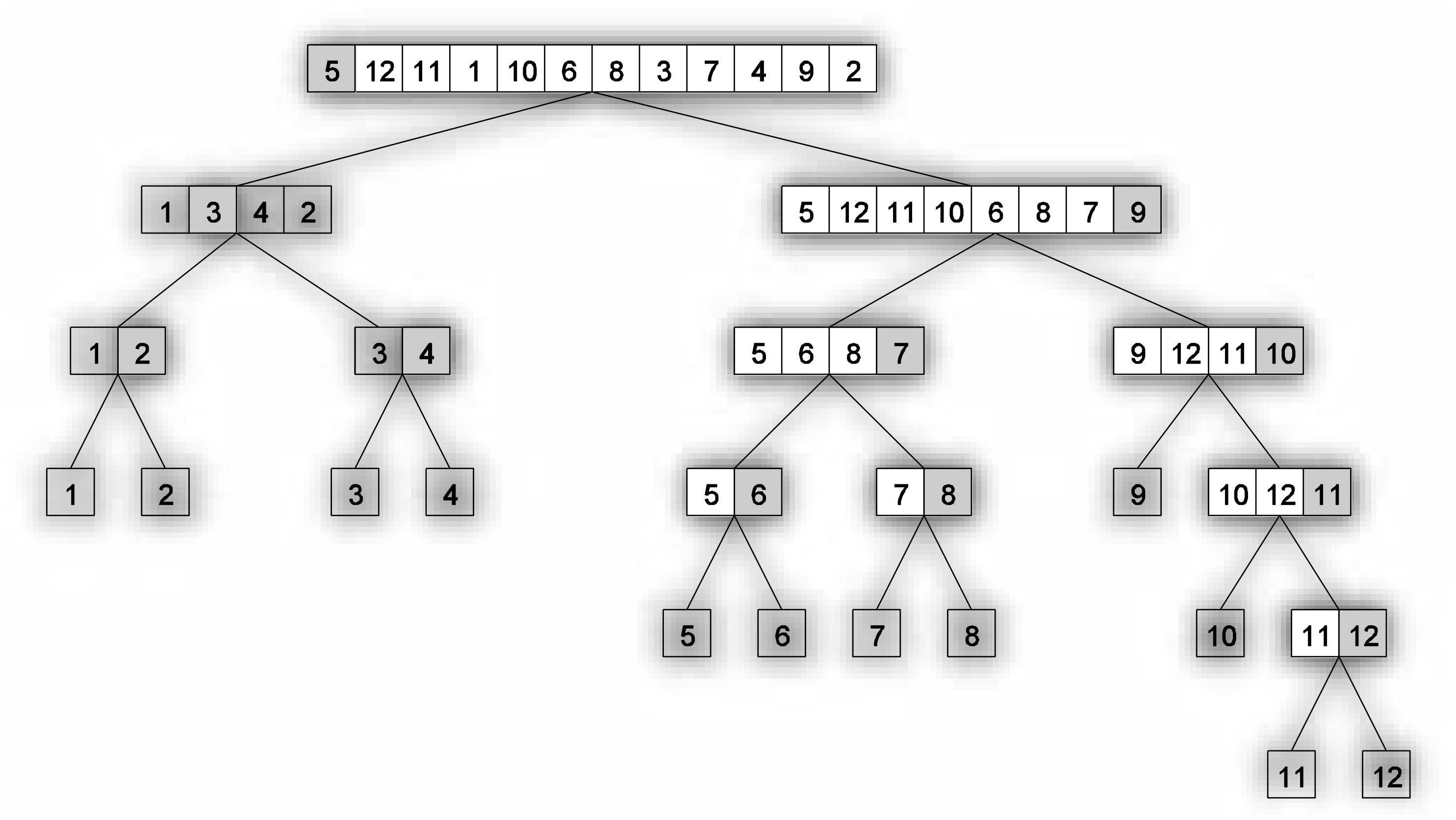
- Recursive Decomposition
- Data Decomposition

- Exploratory Decomposition
- Speculative Decomposition

Recursive Decomposition

- Generally suited to problems that are solved using the divide-and-conquer strategy.
- A given problem is first decomposed into a set of sub-problems.
- These sub-problems are recursively decomposed further until a desired granularity is reached.
- A classic example of a divide-and-conquer algorithm on which we can apply recursive decomposition is Quicksort.

Recursive Decomposition Example:Quicksort

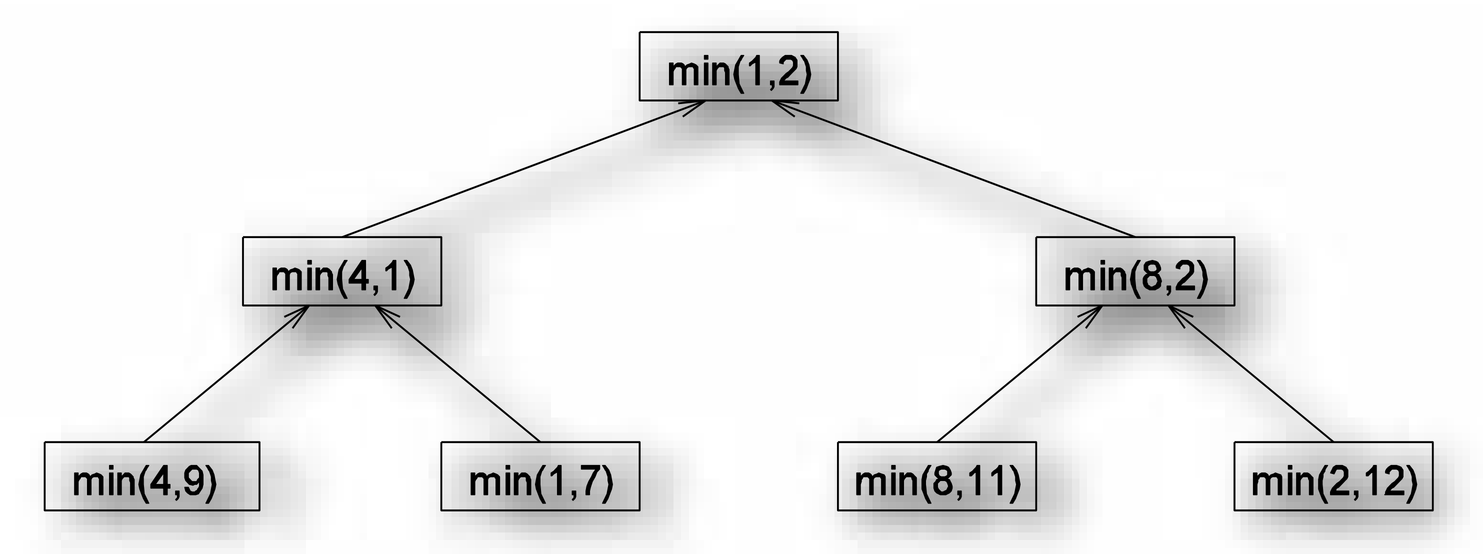


Recursive Decomposition Example: Finding Minimum Number

```
1. procedure RECURSIVE_MIN (A, n)
2. begin
3. if ( n = 1 ) then
4.         min := A [0]  ;
5. else
6.         lmin := RECURSIVE_MIN ( A, n/2 );
7.         rmin := RECURSIVE_MIN (  &(A[n/2]), n - n/2 );
8.         if (lmin  < rmin) then
9.                 min := lmin;
10.        else
11.                min := rmin;
```

```
12.        endelse;
13. endelse;
14. return min;
15. end RECURSIVE_MIN
```

Recursive Decomposition Example: Finding Minimum Number



Example Set: {4,9,1,7,8,11,2,12}

Data Decomposition Technique

Objectives

- What is data decomposition technique
- What are different types of data decomposition?

What is the Data Decomposition Technique?

"Data decomposition is a powerful and commonly used method for deriving concurrency in algorithms that operate on large data structures. In this method, the decomposition of computations is done in two steps."

Data Decomposition Steps?

- In step2, this data partitioning is used to induce a partitioning of the computations into tasks
- In step1, the data on which the computations are performed is partitioned.

How to Decompose Data?

- Identify the data on which computations are performed.
- Partition this data across various tasks.
- This partitioning induces a decomposition of the problem.
- Data can be partitioned in various ways - this critically impacts performance of a parallel algorithm.

Decomposition based on Data Output

- Often, each element of the output can be computed independently of others (but simply as a function of the input).
- A partition of the output across tasks decomposes the problem naturally.

Output Data Decomposition: Example :

“Consider the problem of multiplying two n x n matrices A and B to yield matrix C. The output matrix C can be partitioned into four tasks ”

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

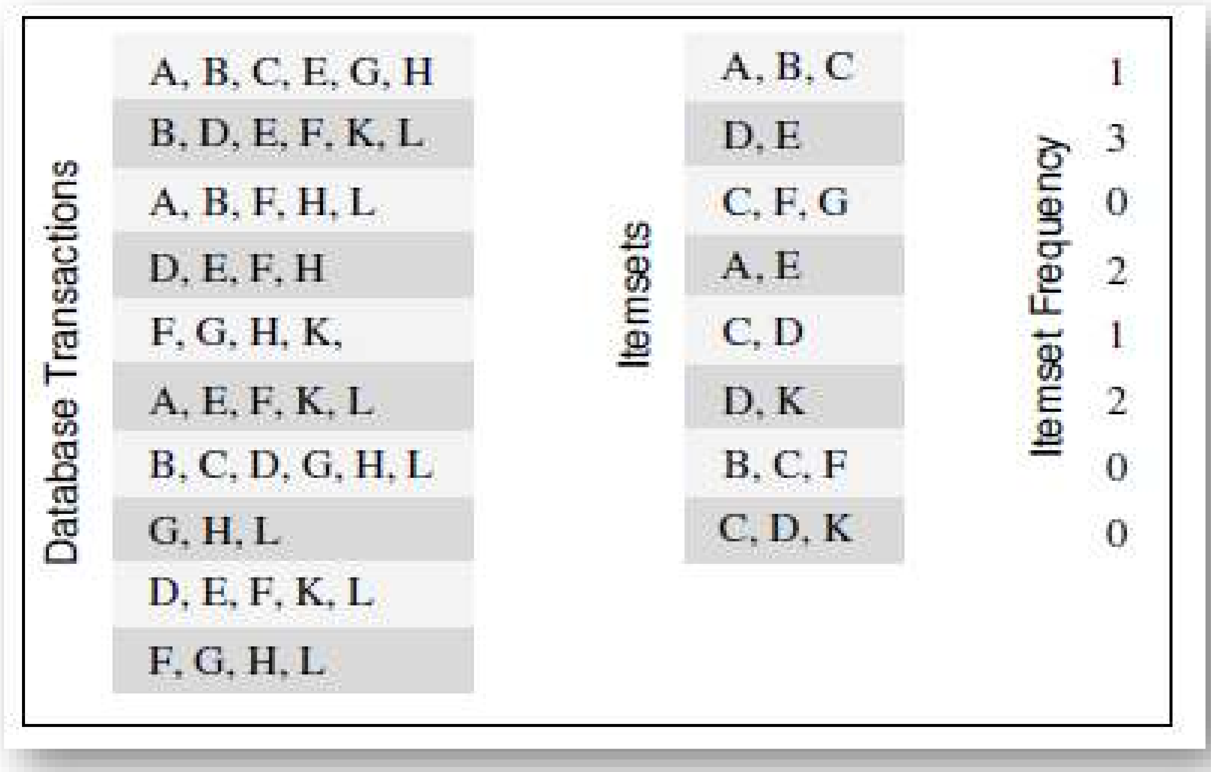
$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Decomposition based on Input Data

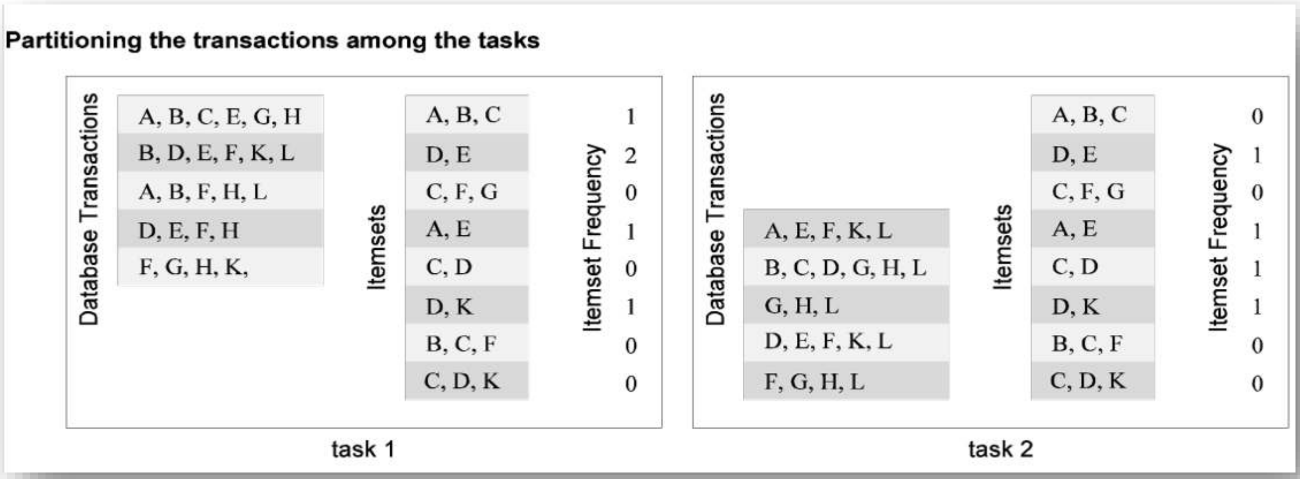
- The Generally applicable if each output can be naturally computed as a function of the input
- In many cases, this is the only natural decomposition because the output is not clearly known a-priori (e.g., the problem of finding the minimum in a list, sorting a given list, etc.).
- A task is associated with each input data partition. The task performs as much of the computation with its part of the data. Subsequent processing combines these partial results.

Transactions (input), itemsets (input), and frequencies (output)



Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,	C, D	1
A, E, F, K, L	D, K	2
B, C, D, G, H, L	B, C, F	0
G, H, L	C, D, K	0
D, E, F, K, L		
F, G, H, L		

Input Data Partitioning: Example



Partitioning the transactions among the tasks

task 1

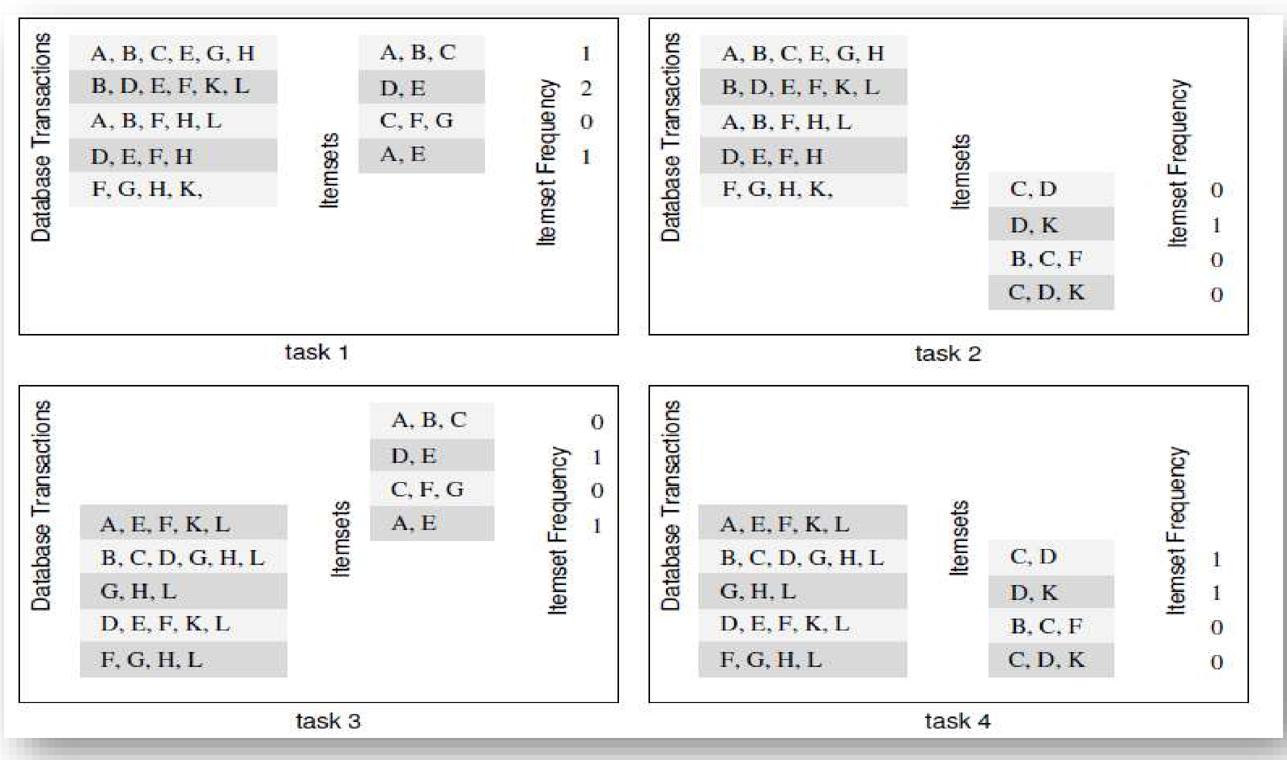
Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	2
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	1
F, G, H, K,	C, D	0
	D, K	1
	B, C, F	0
	C, D, K	0

task 2

Database Transactions	Itemsets	Itemset Frequency
	A, B, C	0
	D, E	1
	C, F, G	0
A, E, F, K, L	A, E	1
B, C, D, G, H, L	C, D	1
G, H, L	D, K	1
D, E, F, K, L	B, C, F	0
F, G, H, L	C, D, K	0

Input and Output Data Partitioning: Example

Partitioning both transactions and frequencies among the



Decomposition based on Intermediate Data

- Computation can often be viewed as a sequence of transformation from the input to the output data.
- In these cases, it is often beneficial to use one of the intermediate stages as a basis for decomposition.

The Owner Computes Rule

- The *Owner Computes Rule* generally states that the process assigned a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule implies that all computations that use the input data are performed by the process.
- In the case of output data decomposition, the owner computes rule implies that the output is computed by the process to which the output data is assigned.

Exploratory and Speculative Decomposition Techniques

Objectives

- What is exploratory decomposition technique
- What is speculative decomposition?

Exploratory Decomposition

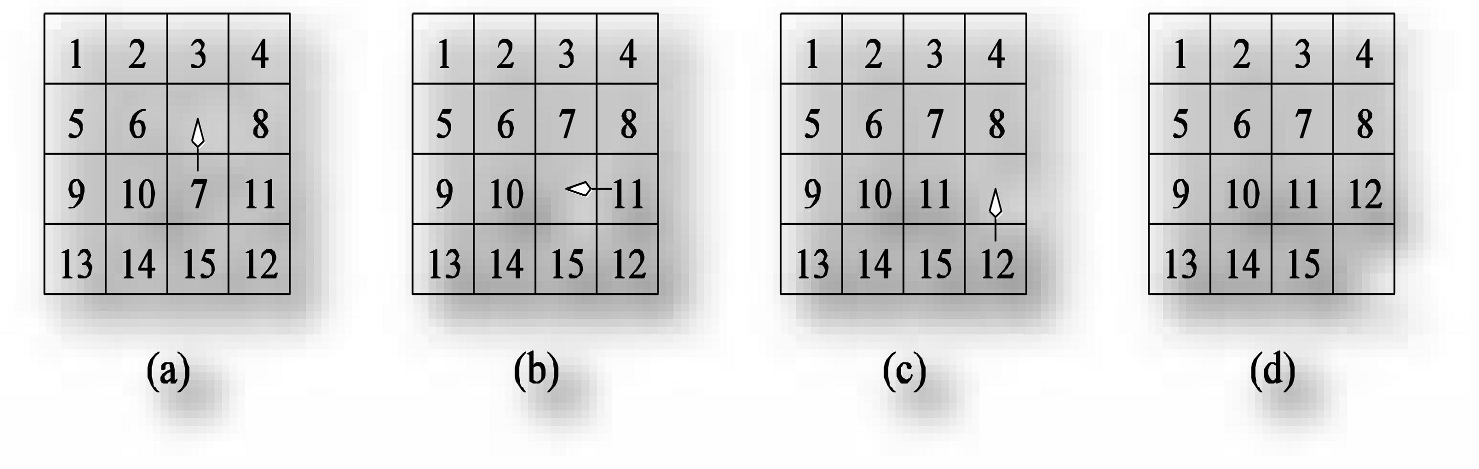
“Exploratory decomposition is used to decompose problems whose underlying computations correspond to a search of a space for solutions.”

Exploratory Decomposition

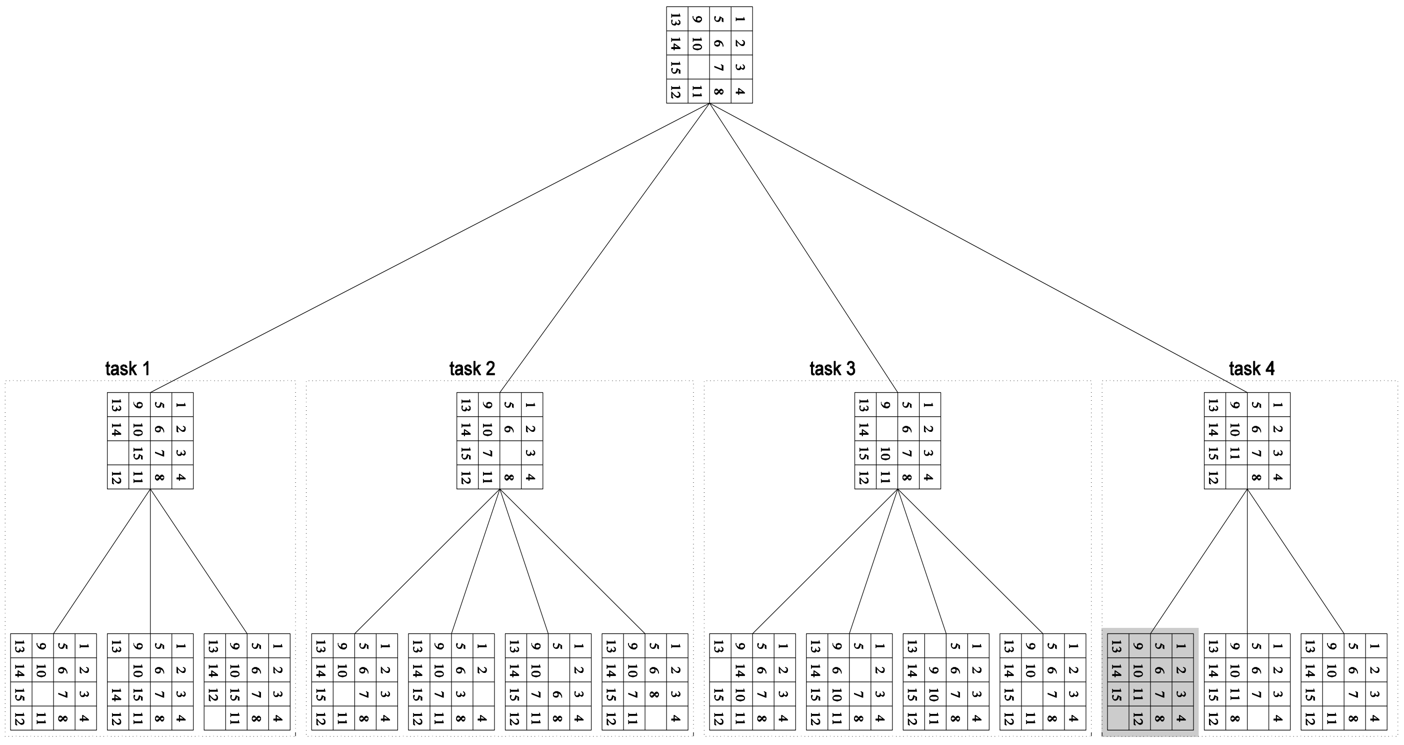
- In many cases, the decomposition of the problem goes hand-in-hand with its execution.
- These problems typically involve the exploration (search) of a state space of solutions.

- Problems in this class include a variety of discrete optimization problems (0/1 integer programming, QAP, etc.), theorem proving, game playing, etc.

Exploratory Decomposition Example:



Exploratory Decomposition Example:



Speculative Decomposition

"Speculative decomposition is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it."

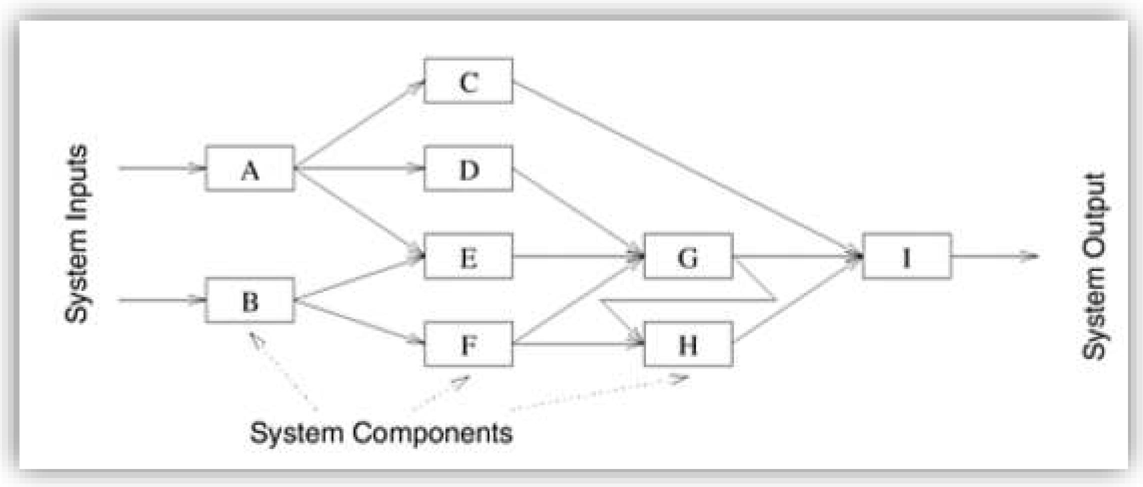
Speculative Decomposition

- In some applications, dependencies between tasks are not known a-priori.
- For such applications, it is impossible to identify independent tasks
- There are generally two approaches:
  conservative approaches, and, optimistic approaches

- Conservative approaches may yield little concurrency and optimistic approaches may require roll-back mechanism in the case of an error

Speculative Decomposition

Example: Discrete Event Simulation



Hybrid Decompositions

- Often, a mix of decomposition techniques is necessary for decomposing a problem
- In quicksort, recursive decomposition alone limits concurrency (Why?). A mix of data and recursive decompositions is more desirable
- In discrete event simulation, there might be concurrency in task processing. A mix of speculative decomposition and data decomposition may work well
- Even for simple problems like finding a minimum of a list of numbers, a mix of data and recursive decomposition works well.

Week 13

Introduction to Parallel I/O

Objectives

- Introduction of parallel I/O.
- Why Parallel I/O?
- I/O bottleneck
- I/O Performance

What is parallel Input/Output (I/O)?

"A Parallel I/O is the concurrent access to I/O devices by multiple processes or threads."

Why is Parallel I/O important?

- Parallel I/O can significantly improve the performance of parallel applications that are I/O-bound
- This is because I/O operations can often be the bottleneck in parallel applications..

How Parallel I/O Work?

- Parallel I/O uses multiple I/O devices to read or write data in parallel.
- This can be done by using multiple disks, multiple network interfaces, or a combination of both

I/O bottleneck

There are three main reason for I/O bottleneck:

  - Increasing CPU Speed as compared to I/O
  - Increase in number of CPUs
  - New application domains that increasing I/O demand

The I/O Challenge

- Problems are increasingly computationally challenging
  - Large parallel machines needed to perform calculations
  - Critical to leverage parallelism in all phases
- Data access is a huge challenge
  - Using parallelism to obtain performance
  - Finding usable, efficient, portable interfaces
  - Understanding and tuning I/O
- Data stored in a single simulation for some projects
  - 100 TB !!

Scalability Limitation of I/O

The most common I/O subsystems are typically very slow compared to other parts of a supercomputer

- – You can easily saturate the bandwidth:

  Once the bandwidth is saturated scaling in I/O stops
- – Adding more compute nodes increases aggregate memory bandwidth and flops/s, but not I/O

Factors which affect I/O

- I/O is simply data migration.
- – Memory←→Disk
  - **I/O is a very expensive operation:**

    – Interactions with data in memory and on disk.
  - **How is I/O performed?:**

    – I/O Pattern

      -- Number of processes and files

      -- Characteristics of file access
  - **Where is I/O performed?:**

    – Characteristics of the computational system.

    – Characteristics of the file system.

I/O Performance

- There is no “One Size Fits All” solution to the I/O problem.
- Many I/O patterns work well for some range of parameters

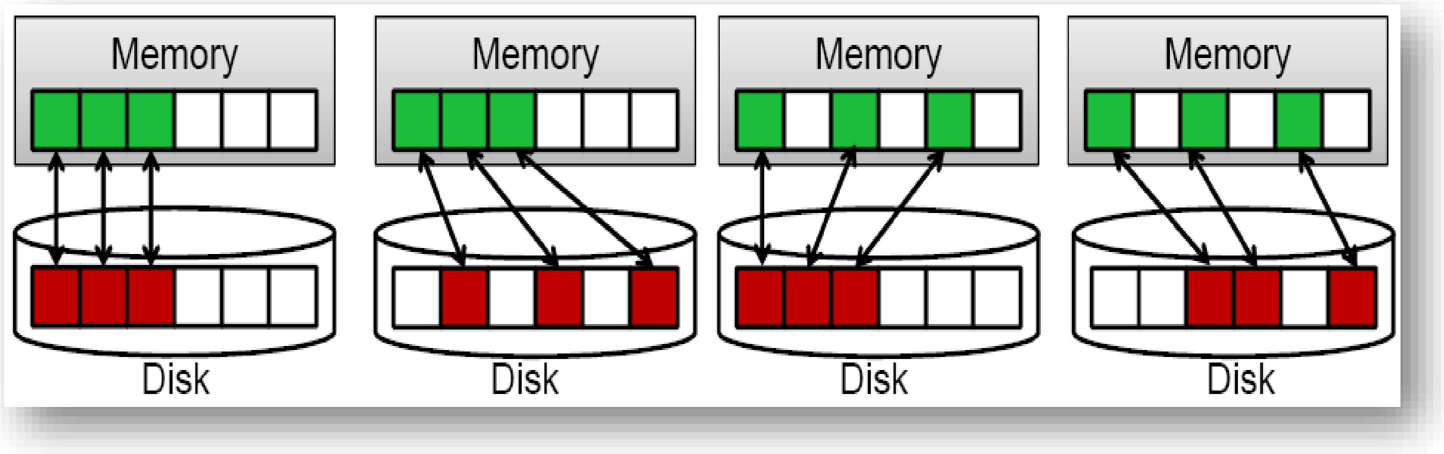
- Bottlenecks in performance can occur in many locations (Application and/or File system)
- Going to extremes with an I/O pattern will typically lead to problems
- Increase performance by decreasing number of I/O operations (latency) and increasing size (bandwidth).

Path from Application to File System

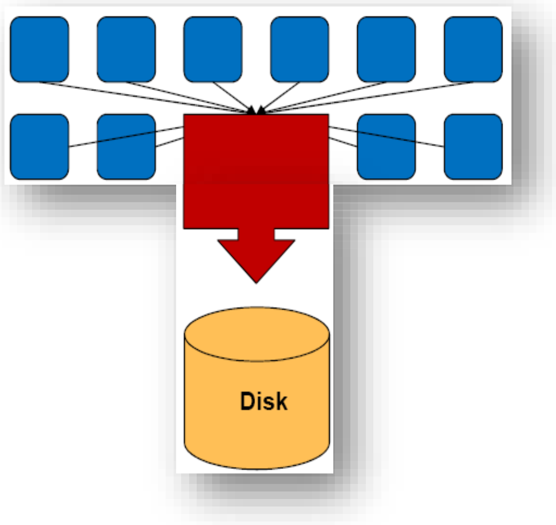
Objectives

- Data and Performance.
- – I/O Patterns
- Lustre File System.
- I/O Performance Results

Data Performance



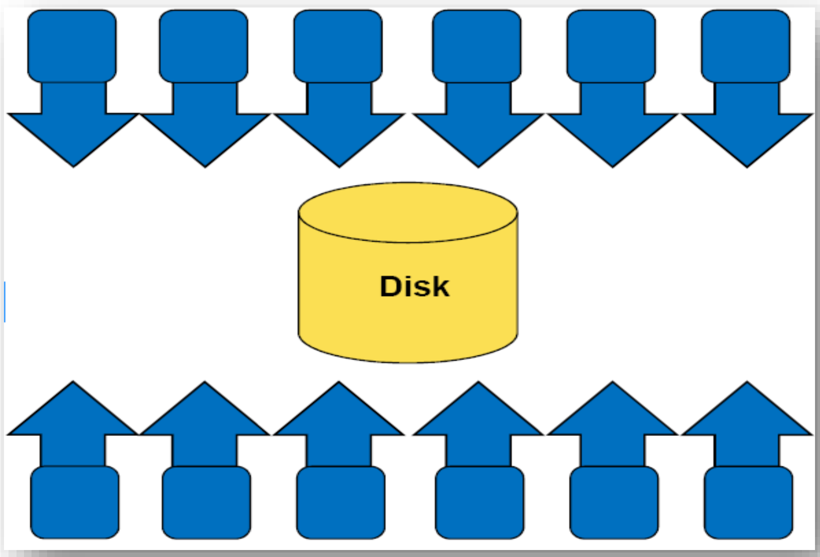
I/O Patterns: Serial I/O



- **One process performs I/O**.
    - Data Aggregation or Duplication
    - Limited by single I/O process.
- **Simple solution, easy to manage, but**
    - Pattern does not scale.
        - Time increases linearly with amount of data.
        - Time increases with number of processes.

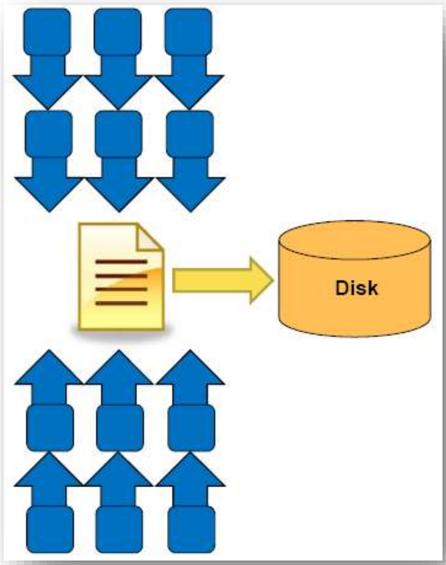
I/O Patterns: Parallel I/O

- **All processes perform I/O to individual files.**
    - Limited by file system.
- **Pattern does not scale at large process counts.**
    - Number of files creates bottleneck with metadata operations.
    - Number of simultaneous disk accesses creates contention for file system resources.



Parallel I/O: Shared File

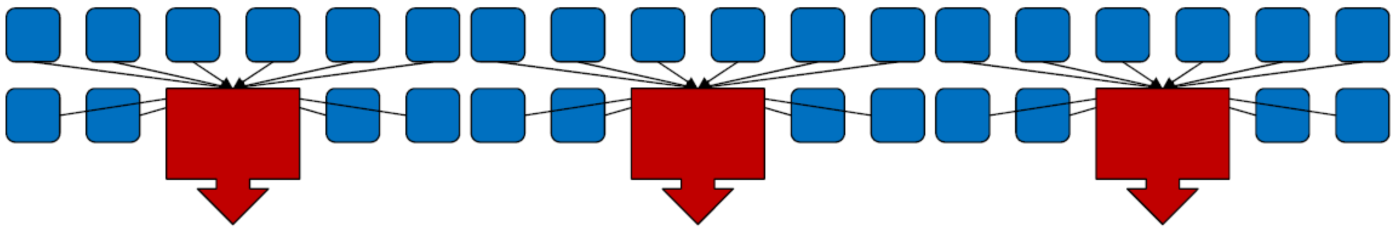
- Each process performs I/O to a single file which is shared.
- **Performance**
    - Data layout within the shared file is very important.
    - At large process counts contention can build for file system resources.



Pattern Combinations

- **Subset of processes which perform I/O.**
    - Aggregation of a group of processes data.
        - Serializes I/O in group.
    - I/O process may access independent files.
        - Limits the number of files accessed.
- Group of processes perform parallel I/O to a shared file.

- Increases the number of shared files
- Increase file system usage.
- Decreases number of processes which access a shared file
- Decrease file system contention.



Performance Mitigation Strategies

- **File-per-process I/O**
    - Restrict the number of processes/files written simultaneously.

      - Limits file system limitation.
    - Buffer output to increase the I/O operation size.
- **Shared file I/O**
    - Restrict the number of processes accessing file simultaneously.

      - Limits file system limitation.
    - Aggregate data to a subset of processes to increase the I/O operation size.
    - Decrease the number of I/O operations by writing/reading strided data.

Parallel I/O Tools

Objectives

- Which tools can be used for Parallel I/O?
- Understanding of Parallel I/O Tools

Parallel I/O Tools

Collections of system software and libraries have grownup to address I/O issues:

- At Parallel file systems
- MPI-IO
- High level libraries

- Relationships between these are not always clear.
- Choosing between tools can be difficult.

Parallel I/O Tools: Break up

· **Break up support into multiple layers:**

- High level I/O library maps app. abstractions to a structured, portable file format (e.g. HDF5, Parallel netCDF, ADIOS)
- Middleware layer deals with organizing access by many processes (e.g. MPI-IO)
- Parallel file system maintains logical space, provides efficient access to data (e.g. Lustre)

- Application

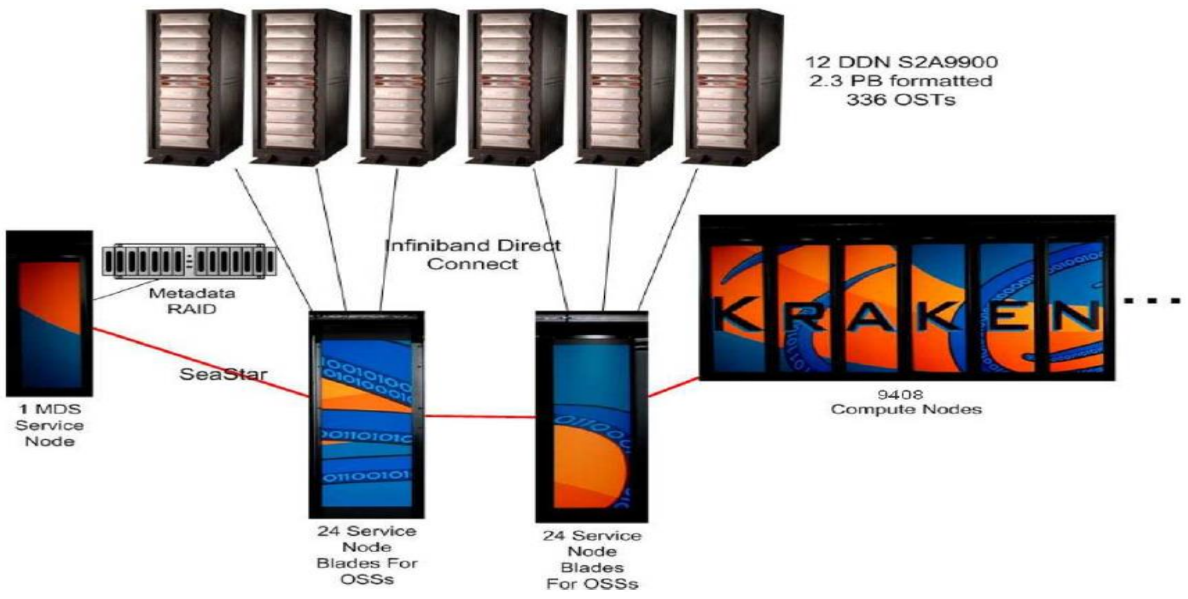
- High level I/O library
- MPI-IO Implementation
- Parallel file system
- Storage Hardware

Parallel File System

Manage storage hardware

- Present single view
- Focus on concurrent, independent access
- Transparent: files accessed over the network can be treated the same as files on local disk by programs and users
- Scalable

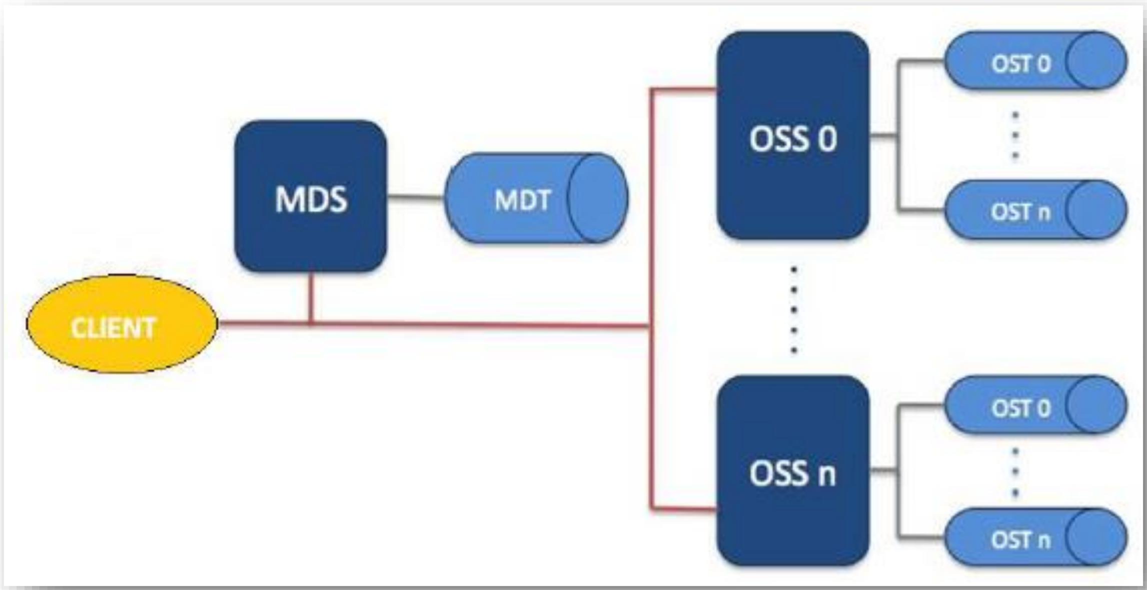
Parallel I/O Tools: Overview of Kraken Lustre



File I/O: Lustre File System

- Metadata Server (MDS) makes metadata stored in the MDT(Metadata Target) available to Lustre clients.
- Object Storage Server(OSS) provides file service, and network request handling for one or more local OSTs.
- Object Storage Target (OST) stores file data (chunks of files).

File I/O: Lustre File System

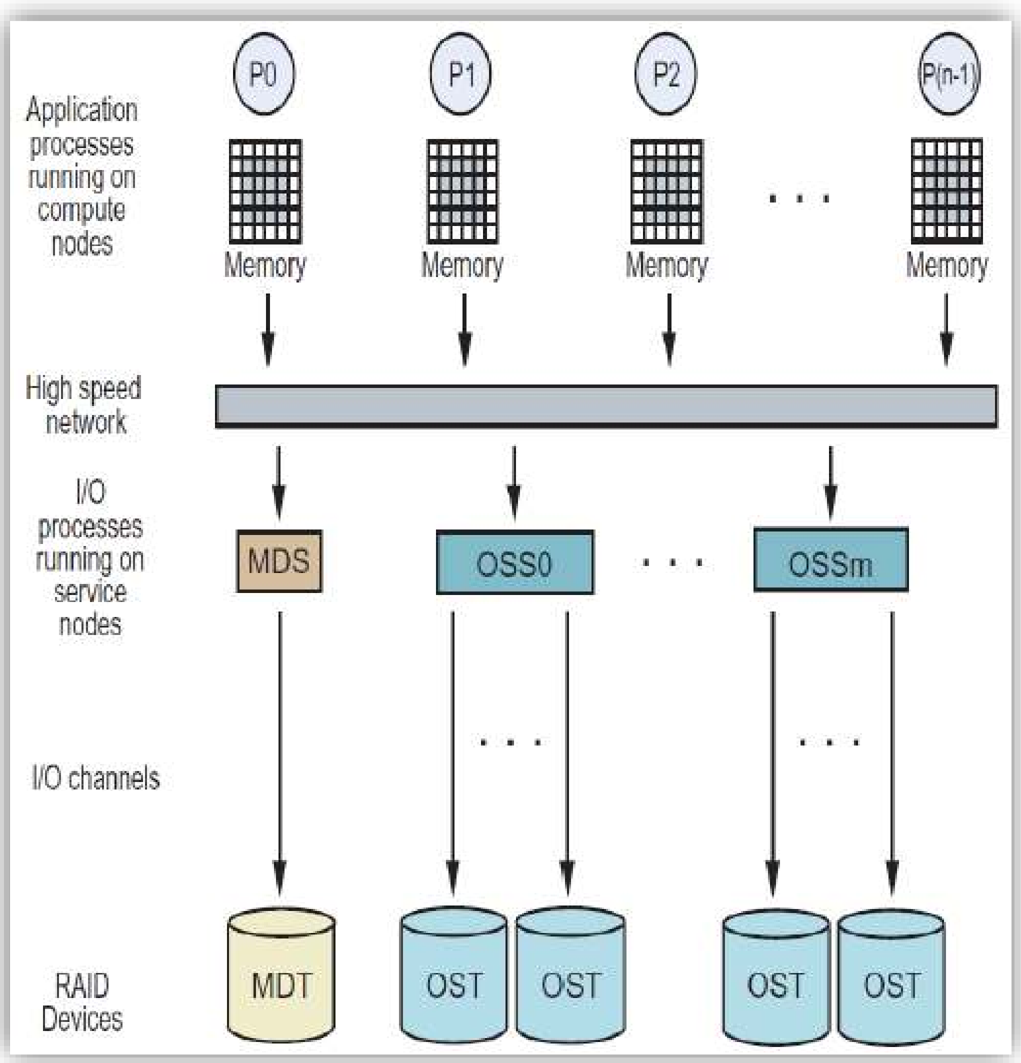


Lustre

Once a file is created, write operations take place directly between compute node processes (P0, P1, ...) and Lustre object storage targets (OSTs), going through the OSSs and bypassing the MDS.

For read operations, file data flows from the OSTs to memory.

Each OST and MDT maps to a distinct subset of the RAID devices



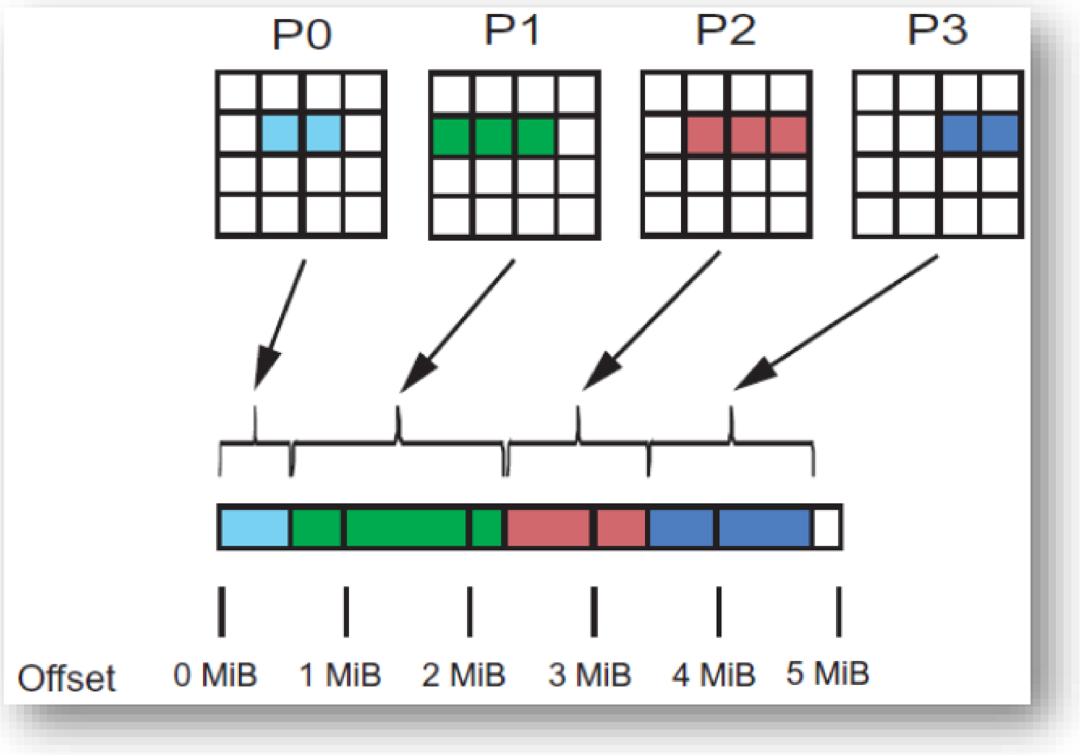
<u>Striping:</u> Storing a single file across multiple OSTs

- A single file may be stripped across one or more OSTs (chunks of
- the file will exist on more than one OST).
- **Advantages:**
- - an increase in the bandwidth available when accessing the file
- - an increase in the available disk space for storing the file
- **Disadvantage:**

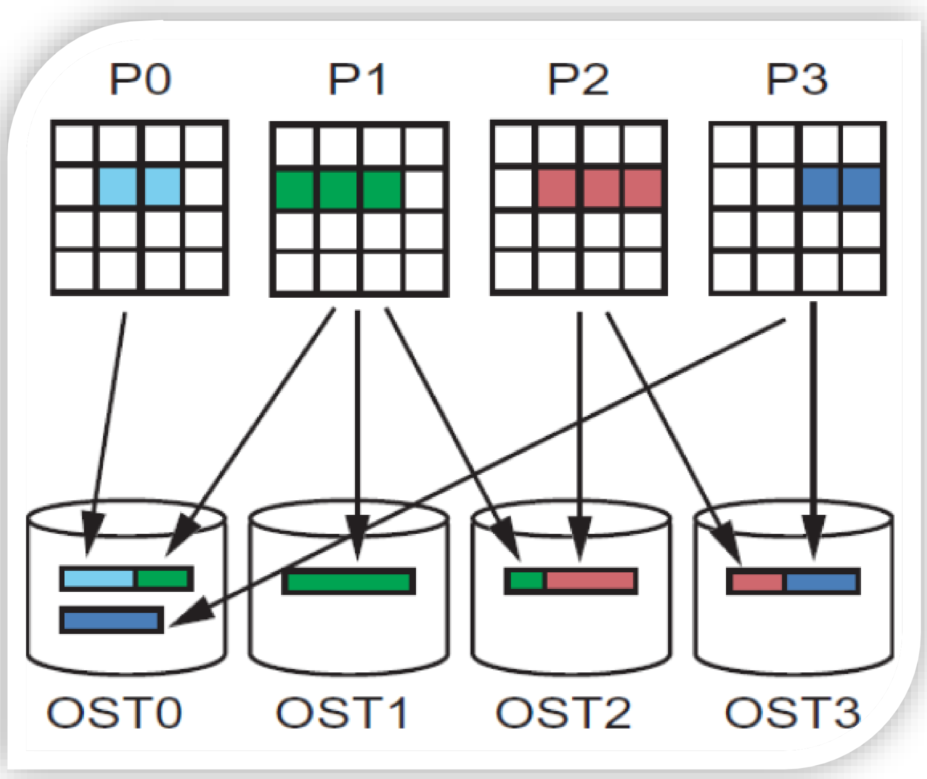
  - increased overhead due to network operations and server contention

File Striping: Physical and Logical Views

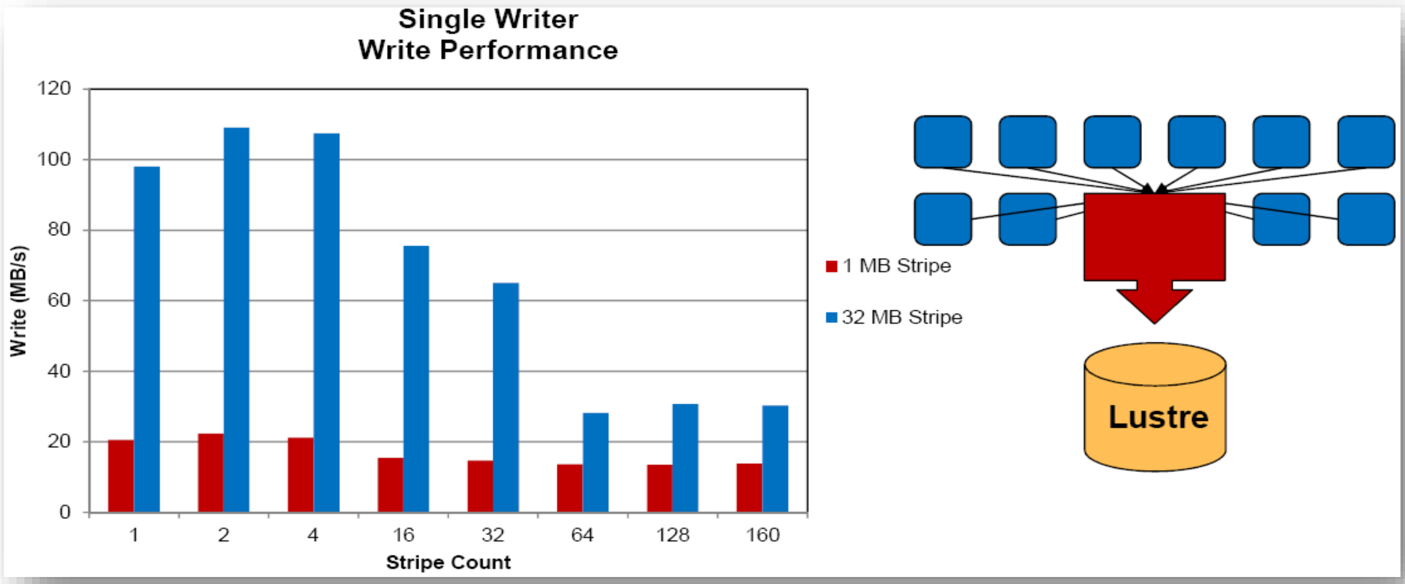
- Four application processes write a variable amount of data sequentially within a shared file.
- This shared file is striped over 4 OSTs with 1 MB stripe sizes.



- This write operation is not stripe aligned therefore some processes write their data to stripes used by other processes.
- Some stripes are accessed by more than one process
- May cause contention !



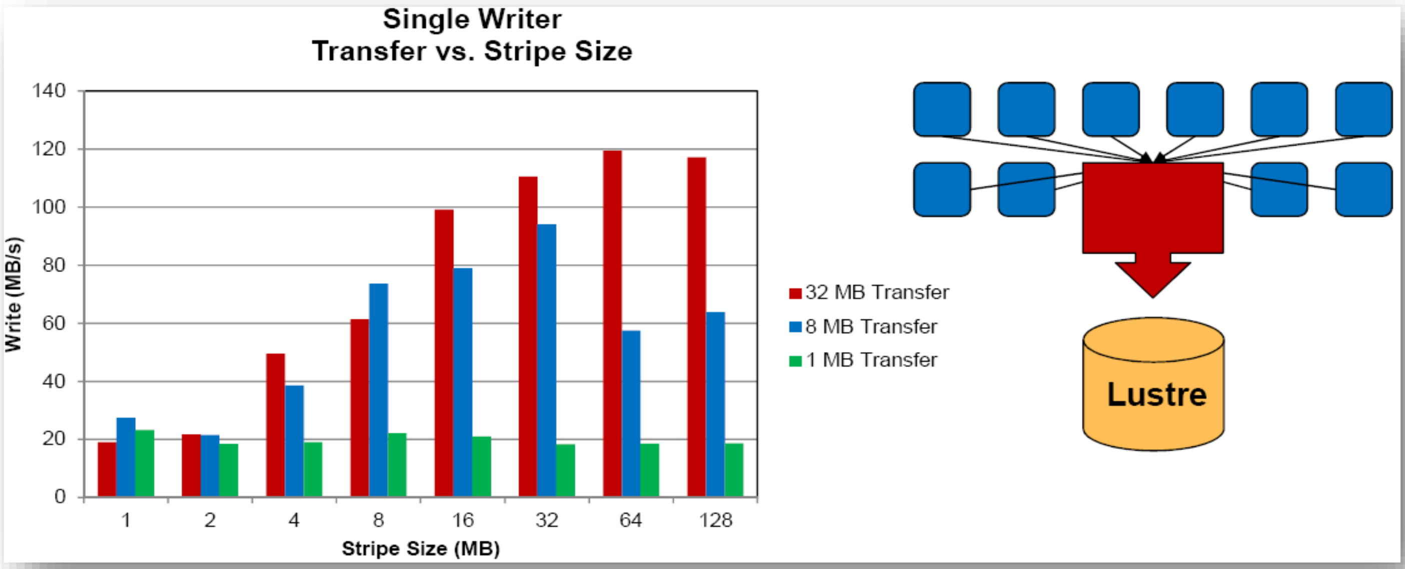
Single writer performance and Lustre



Single writer performance and Lustre

- 32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size
    - Unable to take advantage of file system parallelism
    - Access to multiple disks adds overhead which hurts performance
- Using more OSTs does not increase write performance. (Parallelism in Lustre cannot be exploit)

Stripe size and I/O Operation size



Single OST, 256 MB File Size

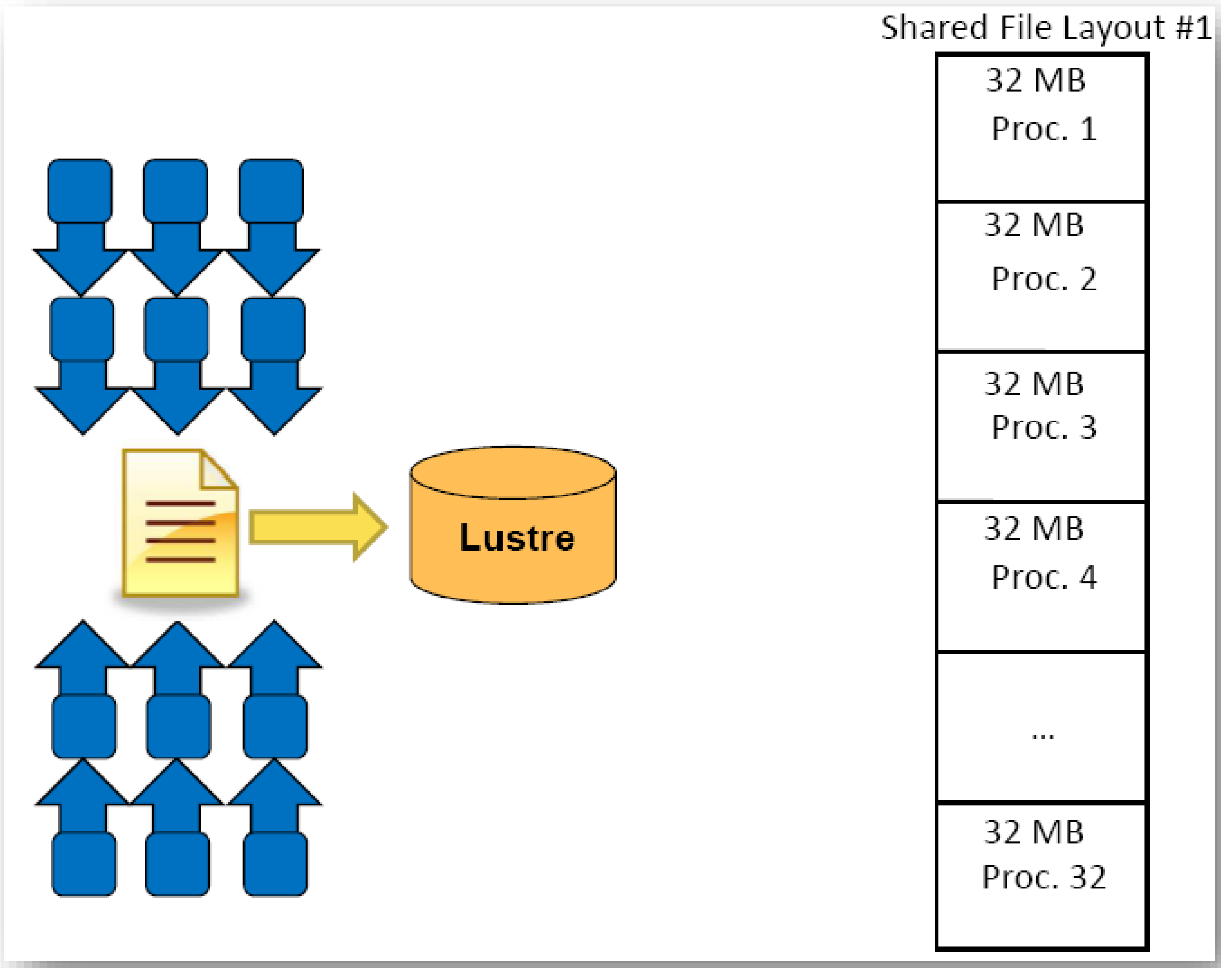
- Single OST, 256 MB File Size
    - Performance can be limited by the process (transfer size) or file system (stripe size).
    - Either can become a limiting factor in write performance.
- Observations:

- The best performance is obtained in each case when the I/O operation and stripe sizes are similar.
- Larger I/O operations and matching Lustre stripe setting may improve performance (reduces the latency of I/O op.)

Single Shared Files and Lustre Stripes

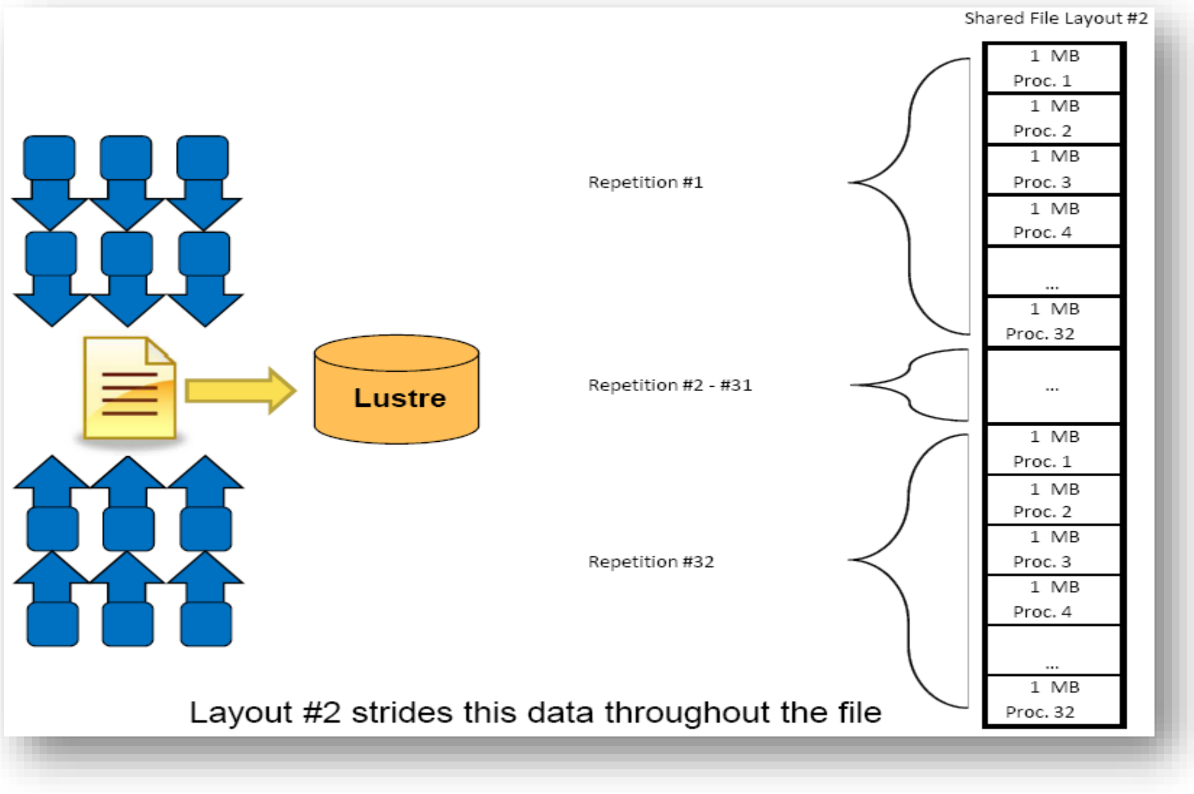
Layout #1:

Keeps data from a process in a contiguous block

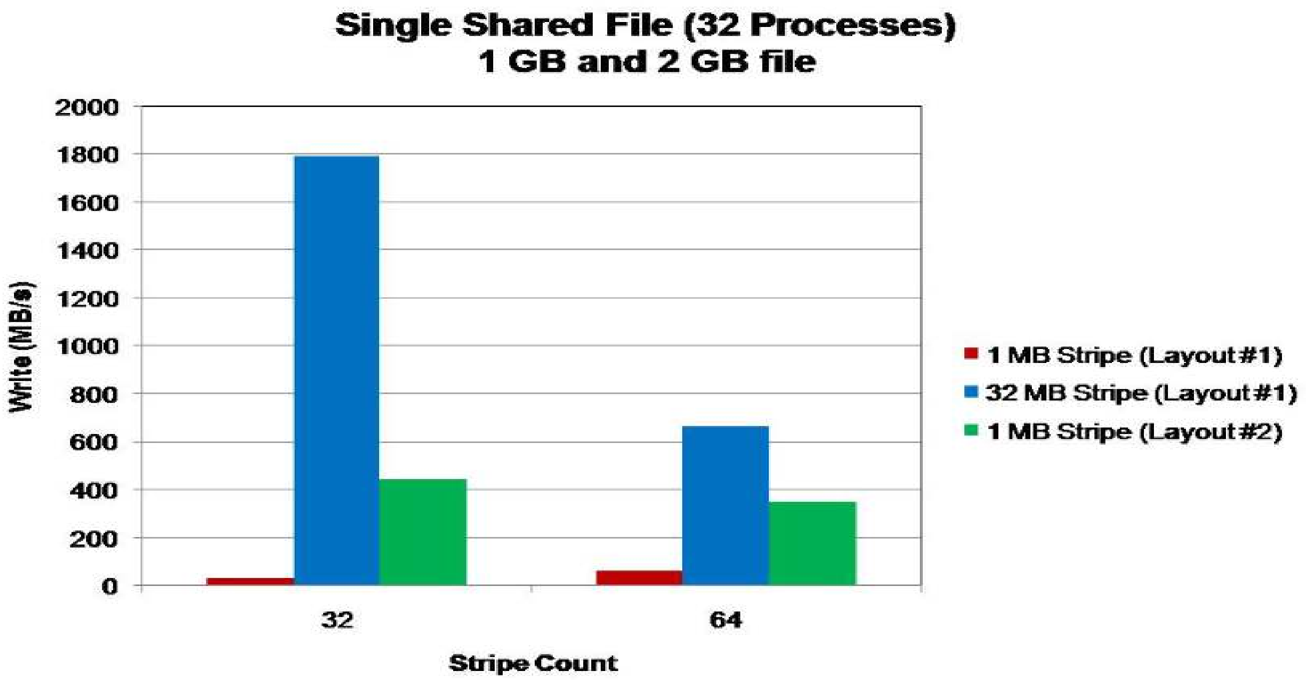


Layout #2:

strides this data throughout the file



File Layout and Lustre Stripe Pattern



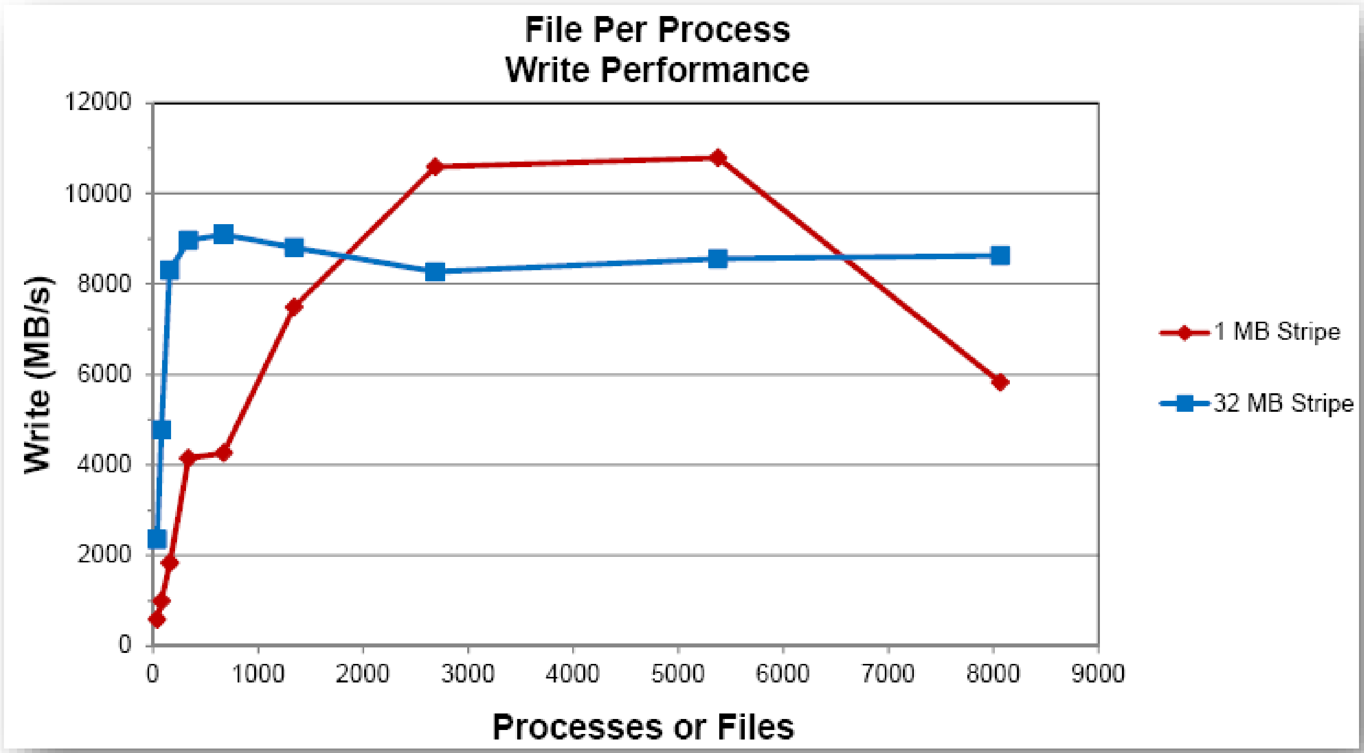
File Layout and Lustre Stripe Pattern

- A 1 MB stripe size on Layout #1 results in the lowest performance due to OST contention. Each OST is accessed by every process. (31.18 MB/s)
- The highest performance is seen from a 32 MB stripe size on Layout #1. Each OST is accessed by only one process. (1788,98 MB/s

A 1 MB stripe size gives better performance with Layout #2. Each OST is accessed by only one process. However, the overall performance is lower due to the increased latency in the write (smaller I/O operations). (442.63MB/s)

Scalability: File Per Process

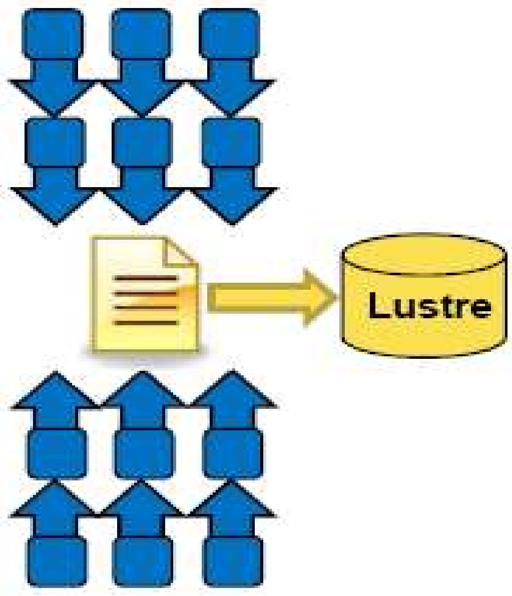
The 128 MB per file and a 32 MB Transfer size



Scalability: File Per Process

- Performance increases as the number of processes/files increases until OST and metadata contention hinder performance improvements.
- At large process counts (large number of files) metadata operations may hinder overall performance due to OSS and OST contention**.**

Case Study: Parallel I/O



- A particular code both reads and writes a 377 GB file. Runs on 6000 cores.

- – Total I/O volume (reads and writes) is 850 GB.
- – Utilizes parallel HDF5
- Default Stripe settings: count 4, size 1M, index -1**.**
- **– 1800 s run time (~ 30 minutes)**
- Stripe settings: count -1, size 1M, index -1.
- – 625 s run time (~ 10 minutes)

Results

– 66% decrease in run time

I/O Scalabity

- Lostre
    - Minimize contention for file system resources.
    - A process should not access more than one or two OSTs.
    - Decrease the number of I/O operations (latency).
    - Increase the size of I/O operations (bandwidth).

Scalability: Summary

- Serial I/O
    - Is not scalable.
    - Limited by single process which performs I/O
- File per Process
    - Limited at large process/file counts by:

      Metadata Operations

      File System Contention
- Single Shared File
    - Limited at large process counts by file system contention

High Level Libraries

Objectives

- Understanding of High Level Libraries.
- POSIX Interface.

High Level Libraries

- Provide an appropriate
- abstraction for domain
    - Block Multidimensional datasets
    - Typed variables
    - Attributes
- Self-describing, structured file Format
- Provide optimizations that middleware cannot
- Map to middleware interface

- – Encourage collective I/O

POSIX:

- POSIX interface is a useful, ubiquitous interface for building basic I/O tools.
    - Standard I/O interface across many platforms.
    - open, read/write, close functions in C/C++/Fortran
    - Mechanism almost all serial applications use to perform I/O
    - No way of describing collective access
- No constructs useful for parallel I/O.
- Should not be used in parallel applications if performance is desired !.

I/O Libraries

- One of the most used libraries on Jaguar and Kraken.
- Many I/O libraries such as HDF5 , Parallel NetCDF and ADIOS are built atop MPI-IO
- Such libraries are abstractions from MPI-IO
- Such implementations allow for higher information propagation to MPI-IO (without user intervention).

MPI-IO

Objectives

- Understanding of MPI-IO basics
- MPI-IO Interfaces

MPI-I/O Basics

- The sending MPI-IO provides a low-level interface to carrying out parallel I/O
- The MPI-IO API has a large number of routines.
- As MPI-IO is part of MPI, you simply compile and link as you would any normal MPI program.
- Facilitate concurrent access by groups of processes

  – Collective I/O

  – Atomicity rules

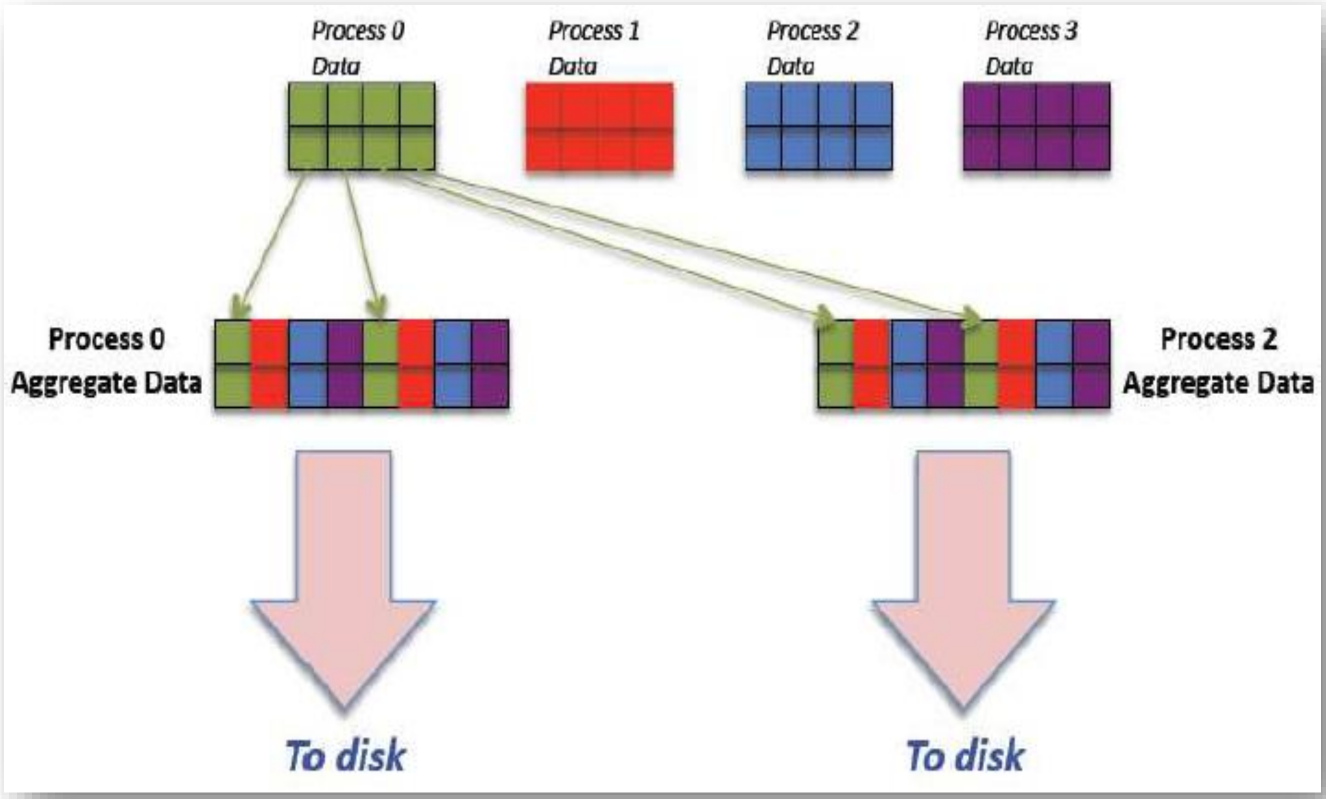
I/O Interfaces: MPI-IO can be done in 2 basic ways

- Independent MPI-IO
    - For independent I/O each MPI task is handling the I/O independently using non collective calls like MPI_File_write() and MPI_File_read().
    - Similar to POSIX I/O, but supports derived datatypes and thus noncontiguous data and non-uniform strides and can take advantages of MPI_Hints
- Collective MPI-IO
    - When doing collective I/O all MPI tasks participating in I/O has to call the same routines. Basic routines are MPI_File_write_all() andMPI_File_read_all()
    - This allows the MPI library to do IO optimization

File MPI Collective Writes and Optimizations

When writing in collective mode, the MPI library carries out a number of optimizations

- It uses fewer processes to actually do the writing
    - Typically one per node
- It aggregates data in appropriate chunks before writing



MPI-IO Interaction with Lustre

Included in the Cray MPT library.

Environmental variable used to help MPI-IO optimize I/O performance:

- BackwarMPICH_MPIIO_CB_ALIGN Environmental Variable. (Default 2)
- MPICH_MPIIO_HINTS Environmental
- Can set striping_factor and striping_unit for files created with MPI-IO.
- If writes and/or reads utilize collective calls, collective buffering can be utilized (romio_cb_read/write) to approximately stripe align I/O within Lustre.
- man mpi for more information

- **Buffered I/O**

  Advantages:
- Aggregates smaller read/write operations into larger operations.
- **Examples:** OS Kernel Buffer and MPI-IO Collective Buffering.
- **Disadvantage:**

  Requires additional memory for the buffer.
- Can tend to serialize I/O.
- Caution:

-

  Frequent buffer flushes can adversely affect performance.

Case Study: Buffered I/O

A post processing application writes a 1GB file

- This occurs from one writer, but occurs in many small write operations.
- – Takes 1080 s (~ 18 minutes) to complete
- IO buffers were utilized to intercept these writes with 4 64 MB buffers.
- – Takes 4.5 s to complete. A 99.6% reduction in time

I/O Best Practices

- Read small, shared files from a single task
- Small files (< 1 MB to 1 GB) accessed by a single process
- Medium sized files (> 1 GB) accessed by a single process
- Large files (>> 1 GB)
- Limit the number of files within a single directory
- Place small files on single OSTs
- Place directories containing many small files on single OSTs
- Avoid opening and closing files frequently

Week 14

Performance and Scalability

Objectives

- What is performance?
- Introduction to Analytical Modeling

What is Performance?

“Computation performance is a measure of how well a computer system can execute a given set of instructions. It can be measured in terms of execution time, overhead, speedup, and cost among others.

Analytical Modeling – Basics

The parallel runtime of a program depends on the input size, the number of processors, and the communication parameters of the machine.

A sequential algorithm is evaluated by its runtime (in general, asymptotic runtime as a function of input size).

- An algorithm must therefore be analyzed in the context of the underlying platform.
- The asymptotic runtime of a sequential program is identical on any serial platform.
- The parallel runtime of a program depends on the input size, the number of processors, and the communication parameters of the machine.

A number of performance measures are intuitive

Wall clock time - the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble. But how does this scale when the number of processors is changed of the program is ported to another machine altogether?

How much faster is the parallel version?

This begs the obvious follow up question –

what's the baseline serial version with which we compare?

Can we use a suboptimal serial program to make our parallel program look .

Raw FLOP count - What good are FLOP counts when they don't solve a problem?

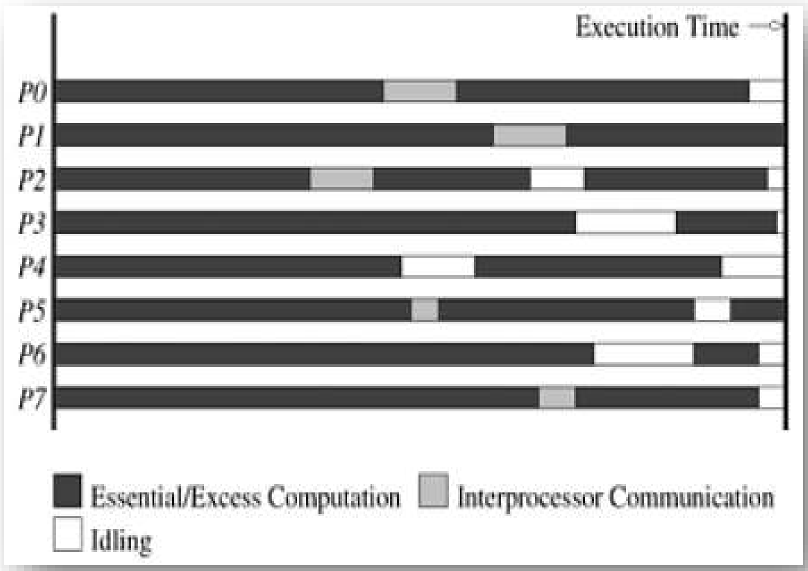
Sources of Overhead in Parallel Programs

Objectives

- Introduction Execution Overhead.
- Sources of Overhead in Parallel Programs

Sources of Overhead in Parallel Programs

- If I use two processors, should not my program run twice as fast?
- No - a number of overheads, including wasted computation, communication, idling, and contention cause degradation in performance.
- The execution profile of a hypothetical parallel program executing on eight processing elements.
- Profile indicates times spent performing computation (both essential and excess), communication, and idling.



- **Inter-process interactions:**
    - Processors working on any non-trivial parallel problem will need to talk to each other.
- **Idling:**

  Processes may idle because of load imbalance, synchronization, or serial components.
- Excess Computation:
    - The difference in computation performed by the parallel program and the best serial program is the excess computation overhead incurred by the parallel program.

- This is computation not performed by the serial version.
- This might be because the serial algorithm is difficult to parallelize, or that some computations are repeated across processors to minimize communication.

Performance Metrics for Parallel Systems

Objectives

- Serial vs Parallel Performance
- Performance Metrics

Performance Metrics for Parallel Systems

- It is important to study the performance of parallel programs with a view to determining the best algorithm, evaluating hardware platforms, and examining the benefits from parallelism.
- A number of metrics have been used based on the desired outcome of performance analysis.

Performance Metrics for Parallel Systems: Execution Time

- Serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer.
- The parallel runtime is the time that elapses from the moment the first processor starts to the moment the last processor finishes execution.
- We denote the serial runtime by T_S and the parallel runtime by T_P .

Performance Metrics: Total Parallel Overhead/Overhead function

- The total time collectively spent by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element.
- Let T_{all} be the total time collectively spent by all the processing elements and T_S is the serial time.
- $T_{all} - T_S$ is then the total time spend by all processors combined in non-useful work. This is called the total overhead.

Performance Metrics: Total Parallel Overhead/Overhead function

The total time collectively spent by all the processing elements

$T_{all} = p\, T_P$ (p is the number of processors).

The overhead function (T_o) is therefore given by

$T_o = p\, T_P - T_S$

Performance Metrics for Parallel Systems: Speedup

Objectives

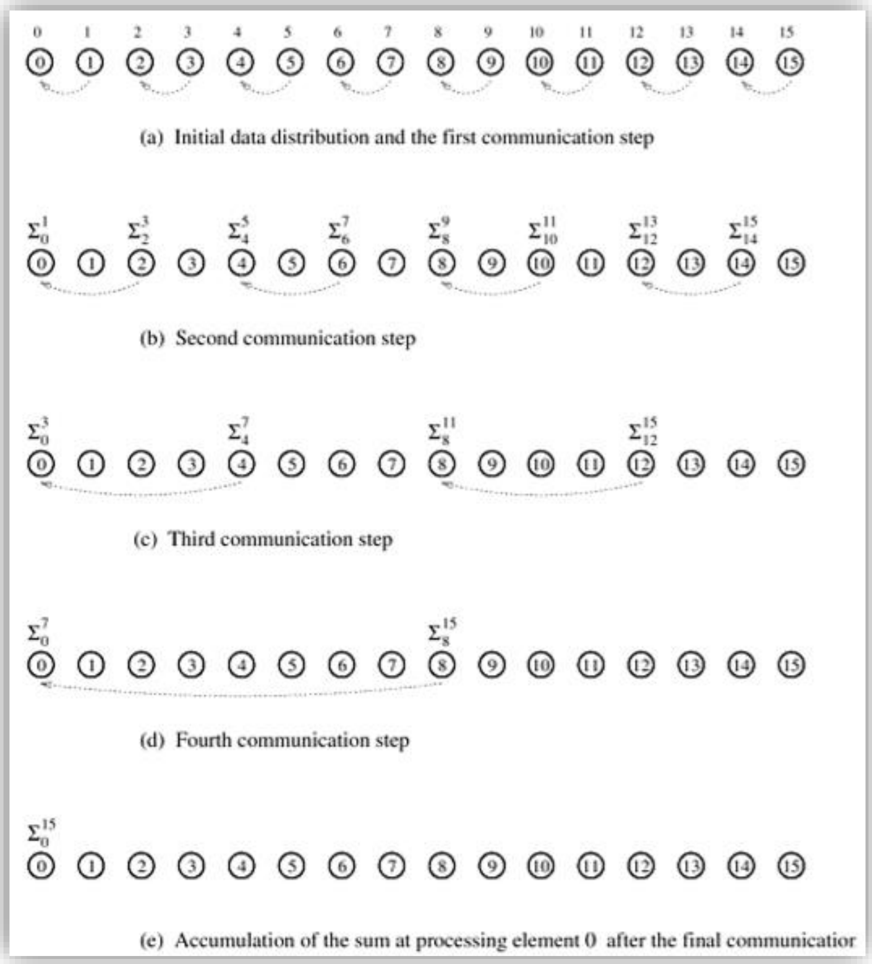
- Introduction to Speedup.
- Speedup Example

Performance Metrics for Parallel Systems: Speedup

“Speedup (S) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p identical processing elements”

Performance Metrics: Example

- Consider the problem of adding ***n*** numbers by using ***n*** processing elements.
- If ***n*** is a power of two, we can perform this operation in **log *n*** steps by propagating partial sums up a logical binary tree of processors.
- This figure illustrates the procedure for n = 16.
- The processing elements are labeled from 0 to 15.
- Similarly, the 16 numbers to be added are labeled from 0 to 15.
- The sum of the numbers with consecutive labels from i to j is denoted by Σ_i^j .
- Each step shown in Figure consists of one addition and the communication of a single word.



(a) Initial data distribution and the first communication step
(b) Second communication step
(c) Third communication step
(d) Fourth communication step
(e) Accumulation of the sum at processing element 0 after the final communicatior

If an addition takes constant time, say, t_c and communication of a single word takes time $t_s + t_w$,

we have the parallel time

$T_P = \Theta(\log n)$

- We know that $T_S = \Theta(n)$

Speedup ***S*** is given by $S = \Theta(n / \log n)$

Performance Metrics: Speedup

- For a given problem, there might be many serial algorithms available. These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees.

- For the purpose of computing speedup, we always consider the best sequential program as the baseline.

Performance Metrics: Speedup Example

- Consider the problem of parallel bubble sort.
- The serial time for bubble sort is 150 seconds.
- The parallel time for odd-even sort (efficient parallelization of bubble sort) is 40 seconds
- The speedup would appear to be 150/40 = 3.75.
- But is this really a fair assessment of the system?
- What if serial quicksort only took 30 seconds? In this case, the speedup is 30/40 = 0.75. This is a more realistic assessment of the system. .

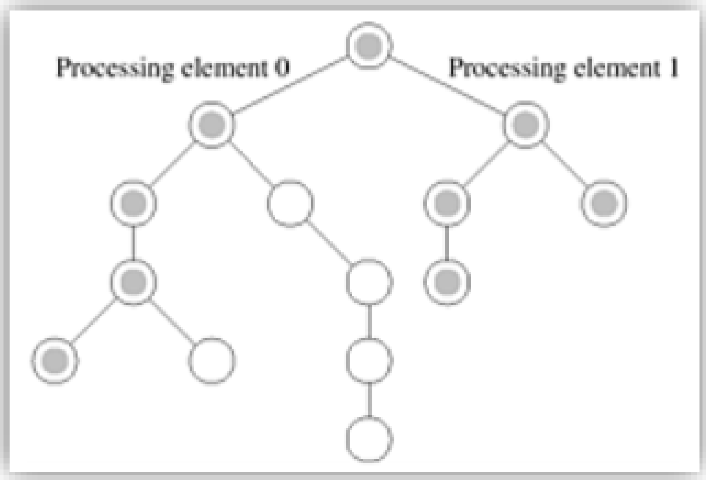
Performance Metrics: Speedup Bounds

- Speedup can be as low as 0 (the parallel program never terminates).
- Speedup can never exceed the number of processing elements, ***p***.

A speedup greater than p is possible only if each processing element spends less than time T_S / **p** solving the problem In this case, a single processor could be time-slided to achieve a faster serial program, which contradicts our assumption of fastest serial program as basis for speedup.

Performance Metrics: Super-linear Speedups

- The phenomenon when the speedup become greater than p is known as superlinear speedup.
- One reason for super linearity is that the parallel version does less work than corresponding serial algorithm.



Performance Metrics: Super-linear Speedups

Resource-based super-linearity:

- The higher aggregate cache/memory bandwidth can result in better cache-hit ratios, and therefore super-linearity.

Example:

- A processor with 64KB of cache yields an 80% hit ratio. If two processors are used, since the problem size/processor is smaller, the hit ratio goes up to 90%. Of the remaining 10% access, 8% come from local memory and 2% from remote memory.

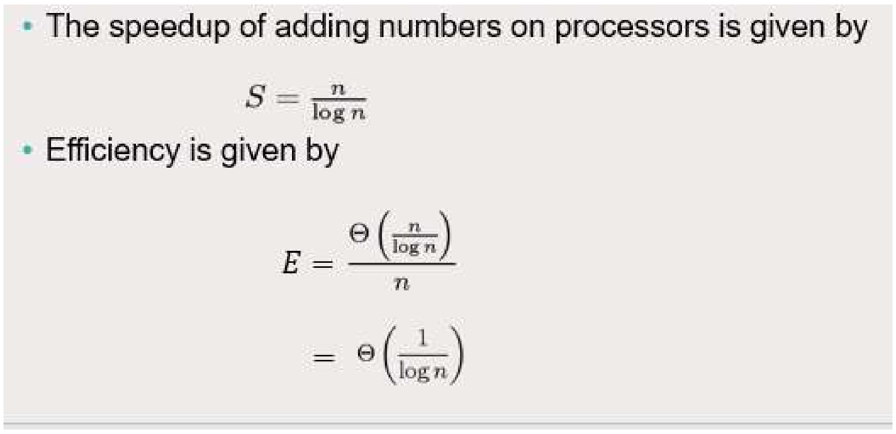
If DRAM access time is 100 ns, cache access time is 2 ns, and remote memory access time is 400ns, this corresponds to a speedup of 2.43!

Performance Metrics: Efficiency

- Efficiency is a measure of the fraction of time for which a processing element is usefully employed
- Mathematically, it is given by
- $E = \frac{S}{T}$

Following the bounds on speedup, efficiency can be as low as 0 and as high as 1.

Performance Metrics: Super-linear Speedups



- The speedup of adding numbers on processors is given by

$$S = \frac{n}{\log n}$$

- Efficiency is given by

$$E = \frac{\Theta\left(\frac{n}{\log n}\right)}{n} = \Theta\left(\frac{1}{\log n}\right)$$

Cost of a Parallel System

- Cost is the product of parallel runtime and the number of processing elements used (p x T_P).
- Cost reflects the sum of the time that each processing element spends solving the problem.
- A parallel system is said to be *cost-optimal* if the cost of solving a problem on a parallel computer is asymptotically identical to serial cost.
- Since $E = T_S / p\, T_P$, for cost optimal systems, $E = O(1)$.
- Cost is sometimes referred to as *work* or *processor-time product*

Cost of a Parallel System: Example

- Consider the problem of adding numbers on processors
- We have, $T_P = \log n$ (for $p = n$).
- The cost of this system is given by $p\, T_P = n \log n$.
- Since the serial runtime of this operation is $\Theta(n)$, the algorithm is not cost optimal.

Impact of Non-Cost Optimality

- Consider a sorting algorithm that uses n processing elements to sort the list in time $(\log n)^2$.
- Since the serial runtime of a (comparison-based) sort is $n \log n$, the speedup and efficiency of this algorithm are given by $n / \log n$ and $1 / \log n$, respectively.
- The $p\, T_P$ product of this algorithm is $n (\log n)^2$.
- This algorithm is not cost optimal but only by a factor of $\log n$.

Impact of Non-Cost Optimality

- If $p < n$, assigning n tasks to p processors gives $T_P = n (\log n)^2 / p$

- The corresponding speedup of this formulation is

  p / **log** ***n***.

- This speedup goes down as the problem size ***n*** is increased for a given ***p*** !.

Scalability of Parallel Systems

Objectives

- The effect of Granularity on Performance
- Introduction to Scalability of Parallel Systems

Scaling Characteristics

Effect of Granularity on Performance

- Often, using fewer processors improves performance of parallel systems.
- Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called *scaling down* a parallel system.
- A naive way of scaling down is to think of each processor in the original case as a virtual processor and to assign virtual processors equally to scaled down processors.
- Since the number of processing elements decreases by a factor of ***n*** / ***p***, the computation at each processing element increases by a factor of ***n*** / ***p***
- The communication cost should not increase by this factor since some of the virtual processors assigned to a physical processors might talk to each other.

This is the basic reason for the improvement from building granularity.

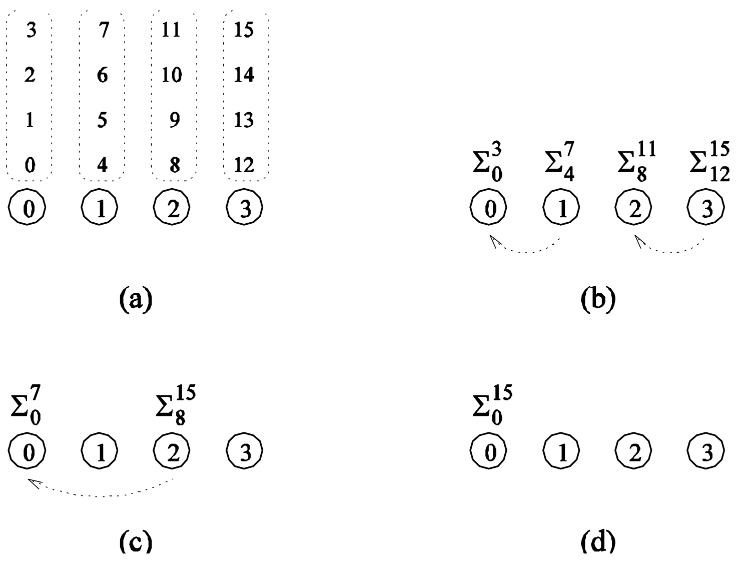
Scalability of Parallel Systems

Can we build granularity in the previous example in a cost-optimal fashion?

Each processing element locally adds its ***n*** / ***p*** numbers in time Θ (***n*** / ***p***).

The ***p*** partial sums on ***p*** processing elements can be added in time Θ(***n*** /***p***).

A cost-optimal way of computing the sum of 16 numbers using four processing elements.


3 7 11 15
2 6 10 14
1 5 9 13
0 4 8 12
0 1 2 3
(a)
Σ_0^3 Σ_4^7 Σ_8^{11} Σ_{12}^{15}
(b)
Σ_0^7 Σ_8^{15}
(c)
Σ_0^{15}
(d)

Scaling Characteristics of Parallel Programs

The efficiency of a parallel program can be written as:

$$E = \frac{S}{p} = \frac{T_S}{pT_P}$$

or

$$E = \frac{1}{1 + \frac{T_o}{T_S}}.$$

Scaling Characteristics of Parallel Programs: Example

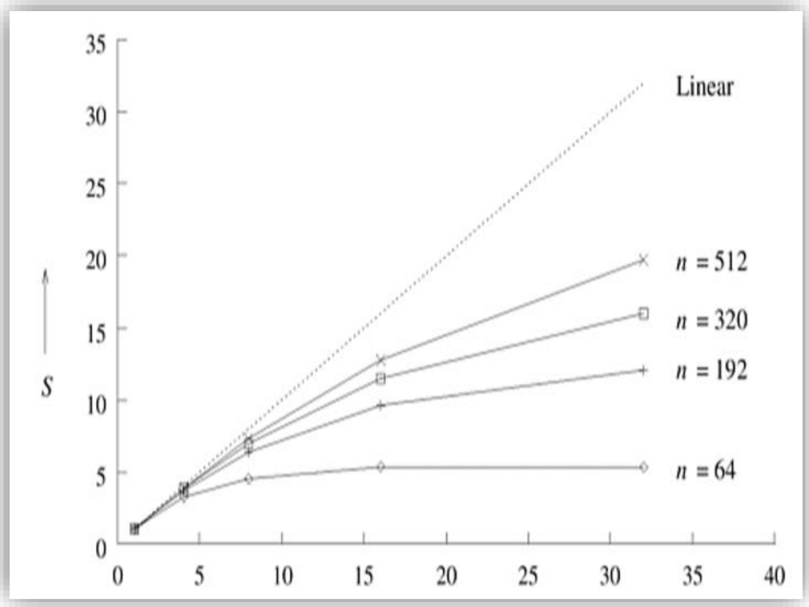
- Consider the problem of adding   numbers on processing elements.
- We have seen that:

$$T_P \quad \frac{n}{p} + 2\log p$$

$$S \quad \frac{n}{\frac{n}{p} + 2\log p}$$

$$E \quad \frac{1}{1 + \frac{2p\log p}{n}}$$

- These expressions can be used to calculate the speedup and efficiency for any pair of n and p
- Plotting the speedup for various input sizes gives us:
- Speedup versus the number of processing elements for adding a list of numbers.
- Speedup tends to saturate and efficiency drops as a consequence of Amdahl's law



Scaling Characteristics of Parallel Programs

- Total overhead function T_o is a function of both problem size T_s and the number of processing elements p.
- In many cases, To grows sub-linearly with respect to Ts

In such cases, the efficiency increases if the problem size is increased keeping the number of processing elements constant.

For such systems, we can simultaneously increase the problem size and number of processors to keep efficiency constant.

- We call such systems scalable parallel systems
- Recall that cost-optimal parallel systems have an efficiency of Θ(1).
- Scalability and cost-optimality are therefore related.
  A scalable parallel system can always be made cost-optimal if the number of processing elements and the size of the computation are chosen appropriately.

Isoefficiency Metric of Scalability

Objectives

- Isoefficiency Metric
- Isoefficiency Metric Example

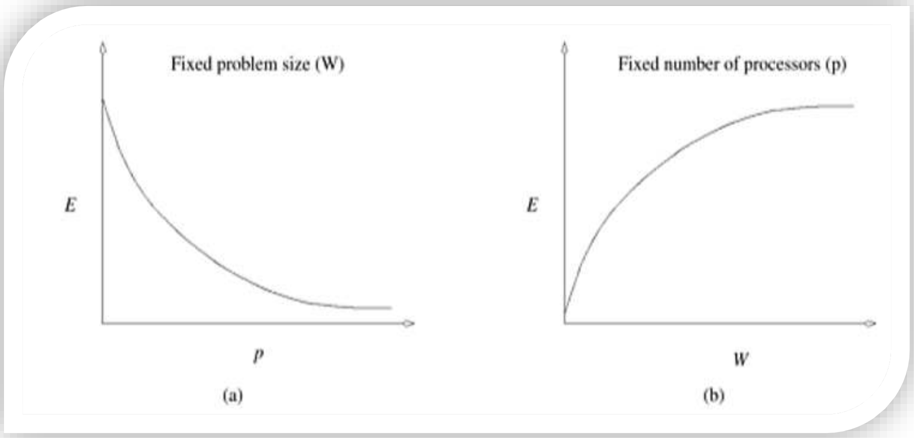
Isoefficiency Metric of Scalability

- For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down for all systems.
- For some systems, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.

Isoefficiency Metric of Scalability

Variation of efficiency: (a) as the number of processing elements is increased for a given problem size; and (b) as the problem size is increased for a given number of processing elements.

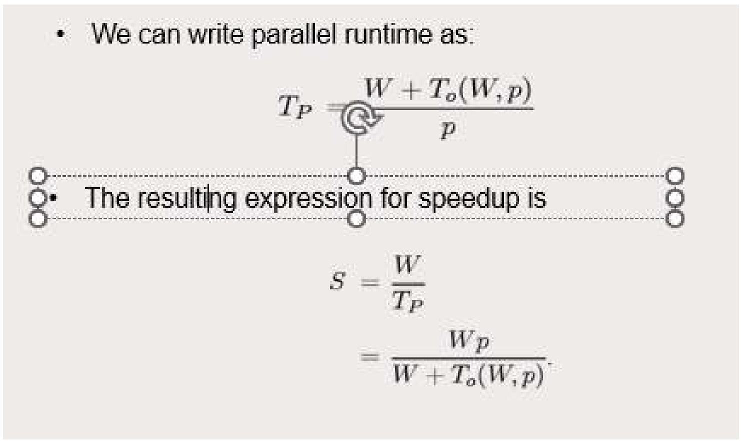
The phenomenon illustrated in graph (b) is not common to all parallel systems.



- What is the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed?

This rate determines the scalability of the system. The slower this rate, the better.

Before we formalize this rate, we define the problem size W as the asymptotic number of operations associated with the best serial algorithm to solve the problem.

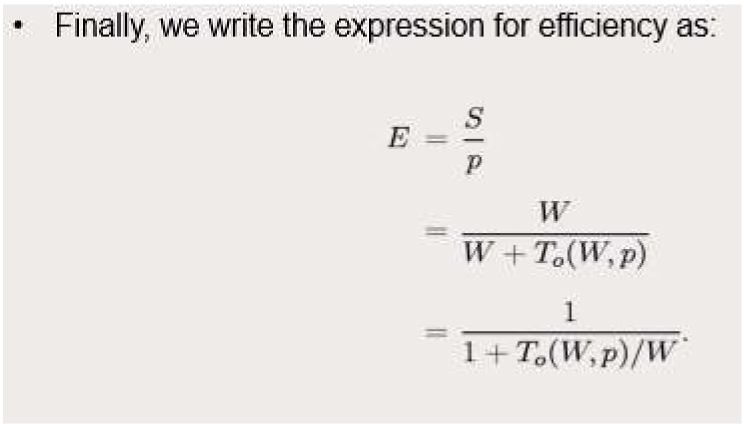


- We can write parallel runtime as:

$$T_P = \frac{W + T_o(W,p)}{p}$$

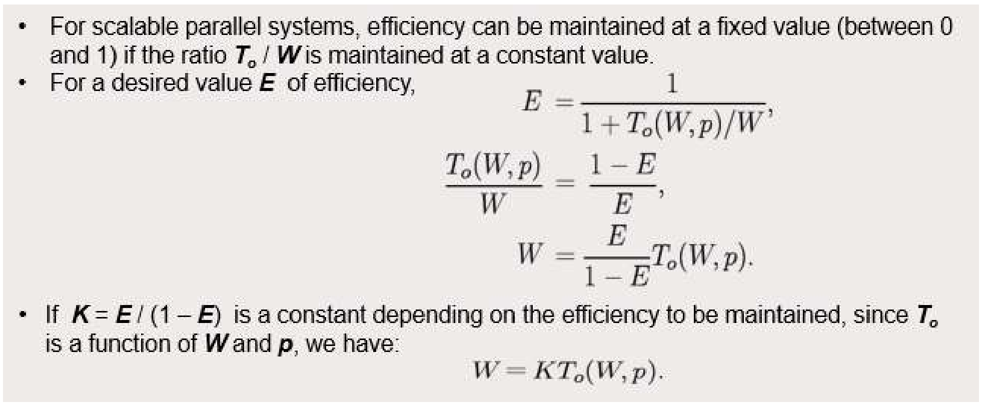
- The resulting expression for speedup is

$$S = \frac{W}{T_P} = \frac{Wp}{W + T_o(W,p)}.$$



- Finally, we write the expression for efficiency as:

$$E = \frac{S}{p} = \frac{W}{W + T_o(W,p)} = \frac{1}{1 + T_o(W,p)/W}.$$



- For scalable parallel systems, efficiency can be maintained at a fixed value (between 0 and 1) if the ratio ***T***$_o$ / ***W*** is maintained at a constant value.
- For a desired value ***E*** of efficiency,

$$E = \frac{1}{1 + T_o(W,p)/W},$$

$$\frac{T_o(W,p)}{W} = \frac{1-E}{E},$$

$$W = \frac{E}{1-E} T_o(W,p).$$

- If ***K*** = ***E*** / (1 – ***E***) is a constant depending on the efficiency to be maintained, since ***T***$_o$ is a function of ***W*** and ***p***, we have:

$$W = K T_o(W,p).$$

- The problem size ***W*** can usually be obtained as a function of ***p*** by algebraic manipulations to keep efficiency constant.
- This function is called the **isoefficiency function**.
- This function determines the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processing elements

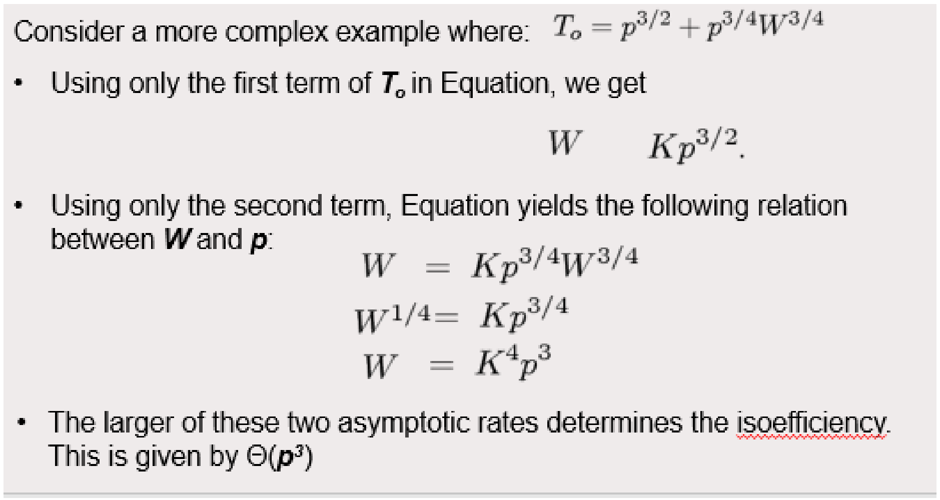
Isoefficiency Metric: Example

- The overhead function for the problem of adding ***n*** numbers on ***p*** processing elements is approximately ***2p* log *p*** .

Substituting ***T$_o$*** by ***2p* log *p*** , we get

$W = K\,2p \log p$

- Thus, the asymptotic isoefficiency function for this parallel system is: Θ(*p* log *p*)
- If the number of processing elements is increased from ***p*** to ***p'***, the problem size (in this case, n) must be increased by a factor of (***p'* log *p'***) / (***p* log *p***) to get the same efficiency as on ***p*** processing elements.



Consider a more complex example where: $T_o = p^{3/2} + p^{3/4}W^{3/4}$

- Using only the first term of T_o in Equation, we get

$$W \qquad Kp^{3/2}.$$

- Using only the second term, Equation yields the following relation between ***W*** and ***p***:

$$W = Kp^{3/4}W^{3/4}$$
$$W^{1/4} = Kp^{3/4}$$
$$W = K^4 p^3$$

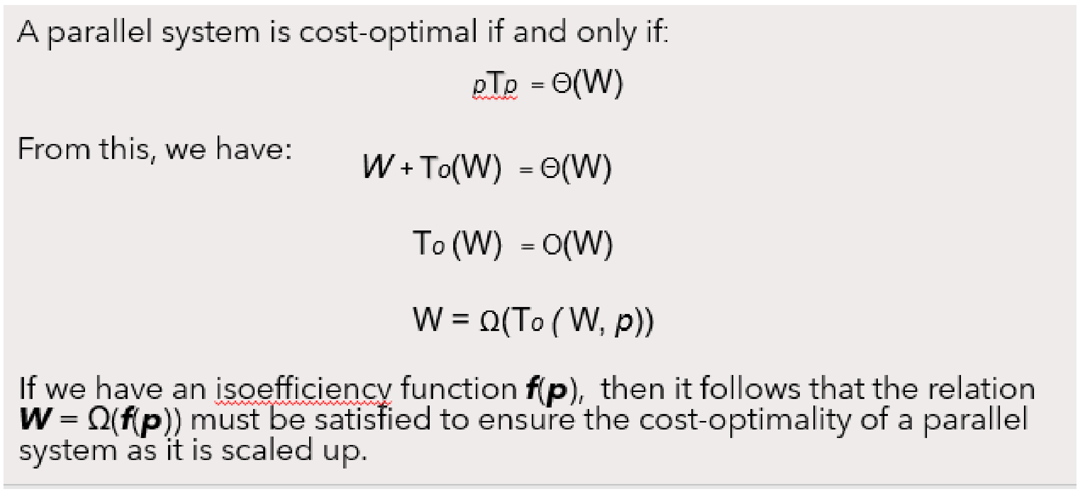
- The larger of these two asymptotic rates determines the isoefficiency. This is given by Θ(p^3)

Isoefficiency Function and Performance metrics

Objectives

- Lower Bound and the Isoefficiency Function
- Degree of Concurrency and Isoefficiency Function

Cost-Optimality and the Isoefficiency Function



A parallel system is cost-optimal if and only if:

$$pT_p = \Theta(W)$$

From this, we have:

$$W + T_o(W) = \Theta(W)$$

$$T_o(W) = O(W)$$

$$W = \Omega(T_o(W, p))$$

If we have an isoefficiency function ***f***(***p***), then it follows that the relation ***W*** = Ω(***f***(***p***)) must be satisfied to ensure the cost-optimality of a parallel system as it is scaled up.

Lower Bound on the Isoefficiency Function

- For a problem consisting of ***W*** units of work, no more than ***W*** processing elements can be used cost-optimally.
- The problem size must increase at least as fast as Θ(***p***) to maintain fixed efficiency; hence, Ω(***p***) is the asymptotic lower bound on the isoefficiency function.

Degree of Concurrency and the Isoefficiency Function

- The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its *degree of concurrency*

- If ***C***(***W***) is the degree of concurrency of a parallel algorithm, then for a problem of size ***W***, no more than ***C***(***W***) processing elements can be employed effectively.

Degree of Concurrency and the Isoefficiency Function: Example

- Consider solving a system of n equations in n variables by using Gaussian elimination ($W = \Theta(n^3)$)
- The ***n*** variables must be eliminated one after the other, and eliminating each variable requires $\Theta(n^2)$ computations.
- At most $\Theta(n^2)$ processing elements can be kept busy at any time.
- Since $W = \Theta(n^3)$ for this problem, the degree of concurrency $C(W)$ is $\Theta(W^{2/3})$
- Given ***p*** processing elements, the problem size should be at least $\Omega(p^{3/2})$ to use them all

Minimum Execution Time and Minimum Cost-Optimal Execution Time

- Often, we are interested in the minimum time to solution.
- We can determine the minimum parallel runtime T_P^{min} for a given ***W*** by differentiating the expression for T_P w.r.t. ***p*** and equating it to zero.
- $\frac{d}{dp} T_P \; = \; 0$
- If p_0 is the value of ***p*** as determined by this equation, $T_P(p_0)$ is the minimum parallel time.

Minimum Execution Time: Example

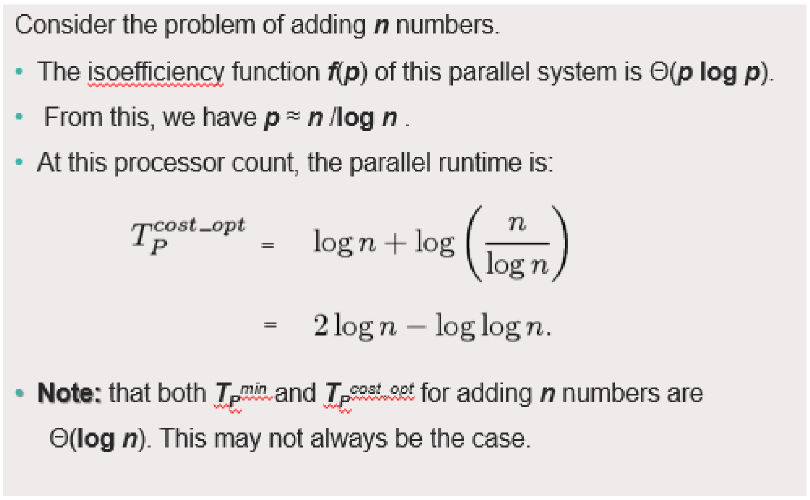
- Consider the minimum execution time for adding ***n*** numbers.
- $T_P \; = \; \frac{n}{p} + 2 \log p \; = \; 0$
- Setting the derivative w.r.t. ***p*** to zero, we have ***p = n/2***
- $Tp^{min} = 2\ logn$
- (One may verify that this is indeed a min by verifying that the second derivative is positive).
- **Note:** that at this point, the formulation is not cost-optimal.

Minimum Cost-Optimal Parallel Time

- Let $T_P^{cost\_opt}$ be the minimum cost-optimal parallel time.
- If the isoefficiency function of a parallel system is $\Theta(f(p))$, then a problem of size ***W*** can be solved cost-optimally if and only if $W = \Omega(f(p))$.
- In other words, for cost optimality, $p = O(f^{-1}(W))$.
- For cost-optimal systems, $T_P = \Theta(W/p)$, therefore,

$$T_P^{cost\_opt} = \Omega\left(\frac{W}{f^{-1}(W)}\right).$$

Minimum Cost-Optimal Parallel Time: Example



Consider the problem of adding ***n*** numbers.

- The isoefficiency function ***f***(***p***) of this parallel system is Θ(***p* log *p***).
- From this, we have ***p* ≈ *n* /log *n*** .
- At this processor count, the parallel runtime is:

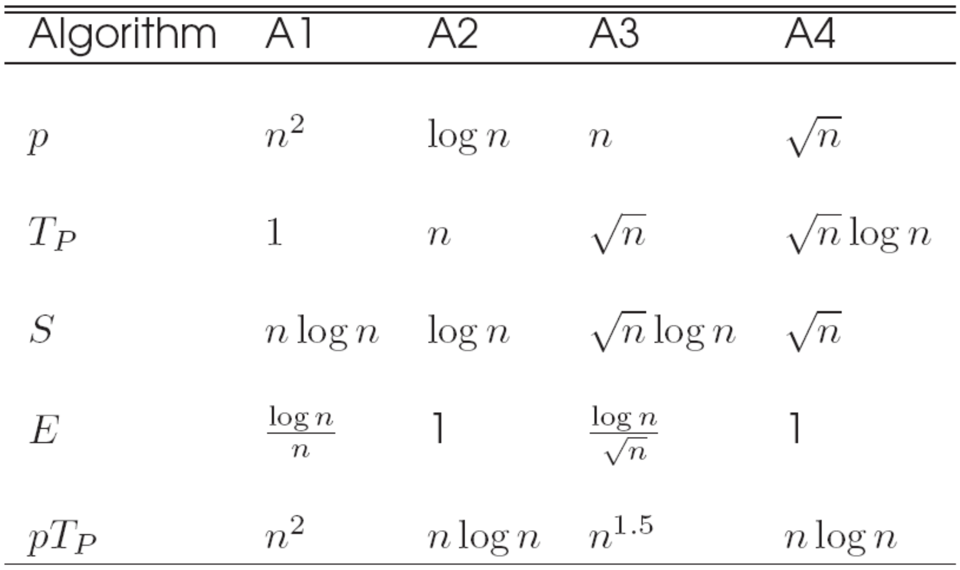
$$
\begin{aligned}
T_P^{cost\_opt} &= \log n + \log\left(\frac{n}{\log n}\right) \\
&= 2\log n - \log\log n.
\end{aligned}
$$

- **Note:** that both T_P^{min} and $T_P^{cost\_opt}$ for adding ***n*** numbers are Θ(**log *n***). This may not always be the case.

Asymptotic Analysis of Parallel Programs

Consider the problem of sorting a list of ***n*** numbers. The fastest serial programs for this problem run in time Θ(***n* log *n***). Consider four parallel algorithms, A1, A2, A3, and A4.

- Comparison of four different algorithms for sorting a given list of numbers. The table shows number of processing elements, parallel runtime, speedup, efficiency and the ***pT_P*** product.



Algorithm	A1	A2	A3	A4
p	n^2	$\log n$	n	$\sqrt{n}$
T_P	1	n	$\sqrt{n}$	$\sqrt{n}\log n$
S	$n\log n$	$\log n$	$\sqrt{n}\log n$	$\sqrt{n}$
E	$\frac{\log n}{n}$	1	$\frac{\log n}{\sqrt{n}}$	1
pT_P	n^2	$n\log n$	$n^{1.5}$	$n\log n$

Asymptotic Analysis of Parallel Programs

- If the metric is speed, algorithm A1 is the best, followed by A3, A4, and A2 (in order of increasing T_P).
- In terms of efficiency, A2 and A4 are the best, followed by A3 and A1.
- In terms of cost, algorithms A2 and A4 are cost optimal, A1 and A3 are not
- It is important to identify the objectives of analysis and to use appropriate metrics!

Week 15

MapReduce

Objectives

- What is MapReduce?
- Usage of MapReduce.

What is MapReduce?

“MapReduce is a software framework which supports parallel and distributed computing on large data sets.”

MapReduce – Introduction

- Simple data-parallel programming model designed for:
    - Scalability and
    - Fault-tolerance
- Pioneered by Google
    - Processes 200 petabytes of data per day (Updated 2022)
- Popularized by open-source Hadoop project
    - Used at Yahoo!, Facebook, Amazon

What is MapReduce used for?

- At Google
    - Index construction for Google Search
    - Article clustering for Google News
    - Statistical machine translation
- At Facebook
    - Data mining
    - Ad optimization
    - Spam detection
- At Yahoo
    - “Web map” powering Yahoo! Search
    - Spam detection for Yahoo! Mail

MapReduce Usage In Research?

- In Research

Astronomical image analysis (Washington)

Bioinformatics (Maryland)

Analyzing Wikipedia conflicts (PARC)

Natural language processing (CMU)

- Particle physics (Nebraska

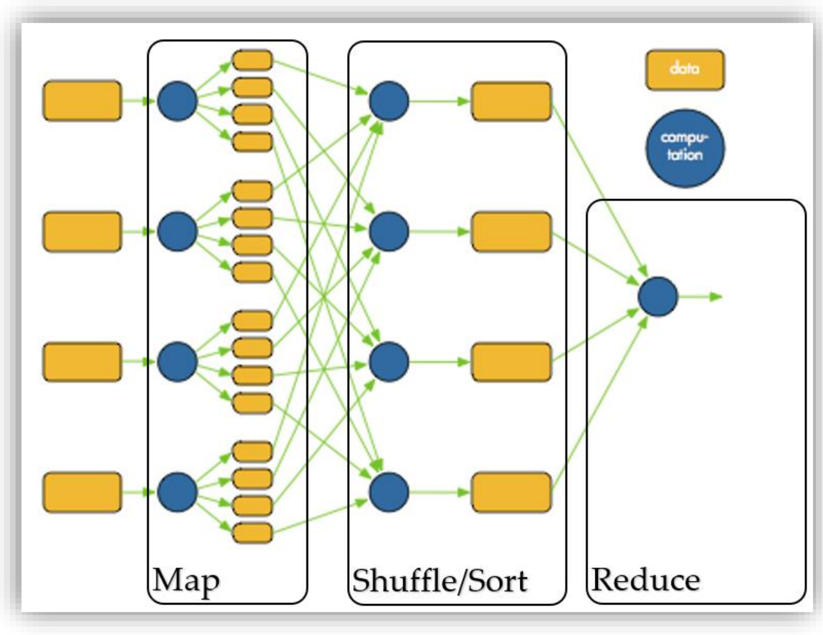
- Ocean climate simulation (Washington)

How MapReduce work?

- MapReduce has three main phases:
    - Map
    - Sort

- Reduce

MapReduce Overview



MapReduce: Examples

Objectives.

- MapReduce example based on three phases.
- Five processing stages based MapReduce example.

MapReduce Example

(based on Three Phases)

The canonical MapReduce Example: Word Count

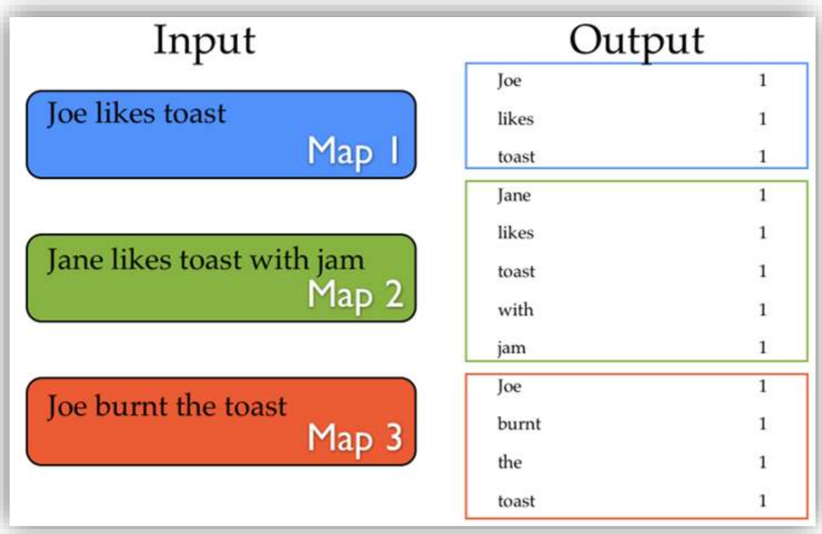
- Example corpus:

  Jane likes toast with jam

  Joe likes toast

  Joe burnt the toast

MapReduce: Map (Slow Motion)



Input

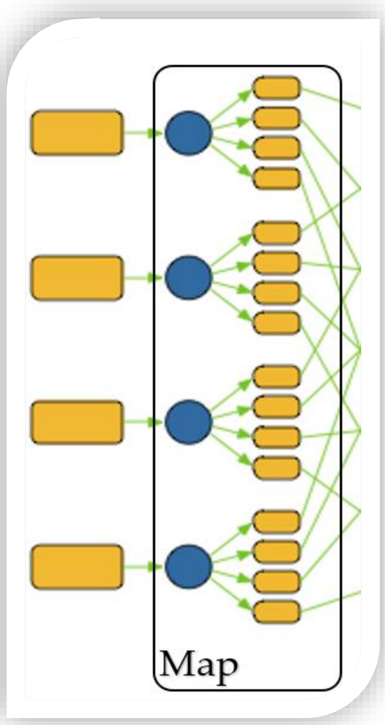
Joe likes toast
Map 1

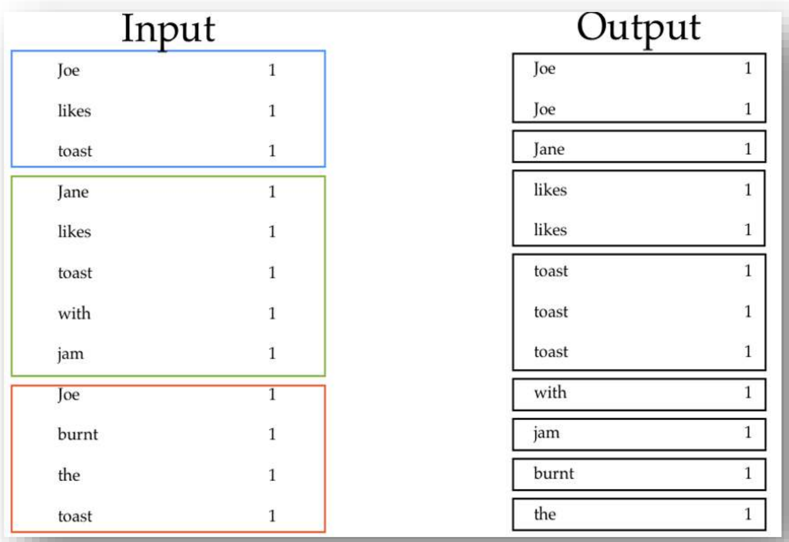
Jane likes toast with jam
Map 2

Joe burnt the toast
Map 3

Output

Joe	1
likes	1
toast	1
Jane	1
likes	1
toast	1
with	1
jam	1
Joe	1
burnt	1
the	1
toast	1



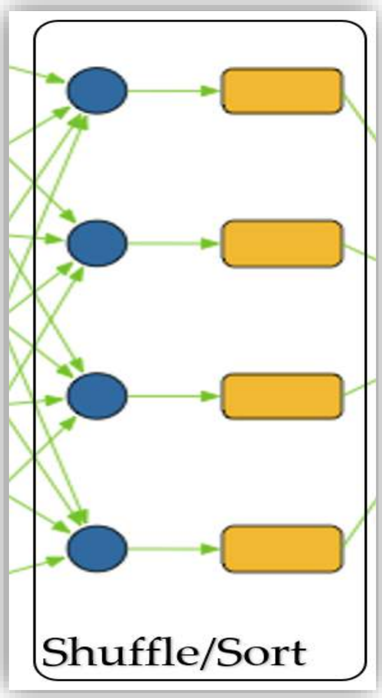


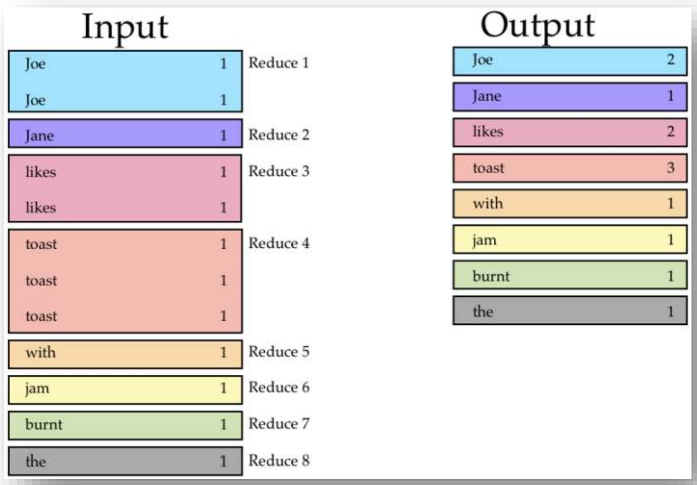
Input

Joe	1
likes	1
toast	1
Jane	1
likes	1
toast	1
with	1
jam	1
Joe	1
burnt	1
the	1
toast	1

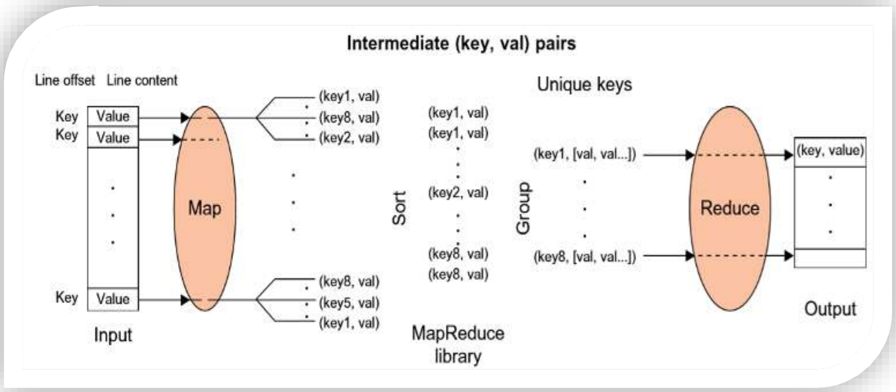
Output

Joe	1
Joe	1
Jane	1
likes	1
likes	1
toast	1
toast	1
toast	1
with	1
jam	1
burnt	1
the	1

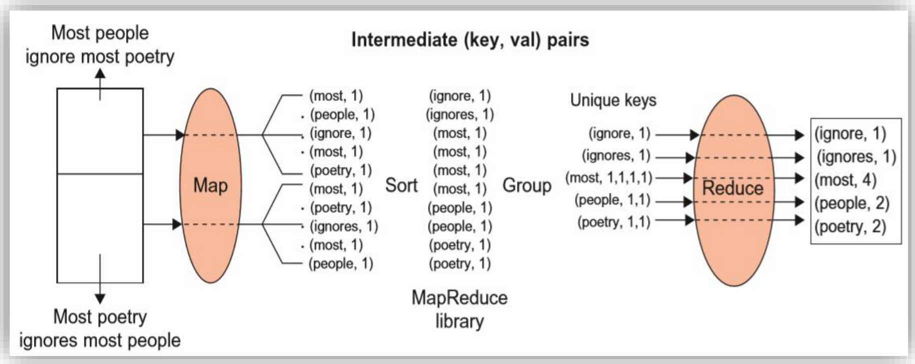




MapReduce logical data flow in 5 processing stages over successive (key, value) pairs.



MapReduce logical data in 5 processing stages : Example



MapReduce Actual Data and Control Flow:

The main responsibility of the MapReduce framework is to efficiently run a user's program on a distributed computing system.

Therefore, the MapReduce framework meticulously handle all processing steps like:

Data and Computation

Partitioning

- Determining the master and worker
- Reading the input data
- (Data Distribution
- Map and Combiner function
- Synchronization
- Communication
- Sorting and Grouping
- Reduce function

MapReduce Design Goals

- **Scalability to large data volumes**
    - 1000's of machines, 10,000's of disks
- **Cost-efficiency:**
    - Commodity machines (cheap, but unreliable)
    - Commodity network
    - Automatic fault-tolerance (fewer administrators),
    - Easy to use (fewer programmers)

Hadoop

Objectives

- Introduction Hadoop.
- Key functions of Hadoop

What is Hadoop?

"Open source platform for distributed processing of large data. Hadoop is a simplified programming model that make it easy to write distributed algorithms"

Key functions of Hadoop

- The Distribution of data and processing across machine
- Management of the cluster

Hadoop scalability

- Hadoop can reach massive scalability by exploiting a simple distribution architecture and coordination model
- Huge clusters can be made up using (cheap) commodity hardware

A 1000-CPU machine would be much more expensive than 1000 single-CPU or 250 quad-core machines

- Cluster can easily scale up with little or no modifications to the programs

Hadoop Components

HDFS: Hadoop Distributed File System:

Abstraction of a file system over a cluster

Stores large amount of data by transparently spreading it on different machines

MapReduce

Simple programming model that enables parallel execution of data processing programs

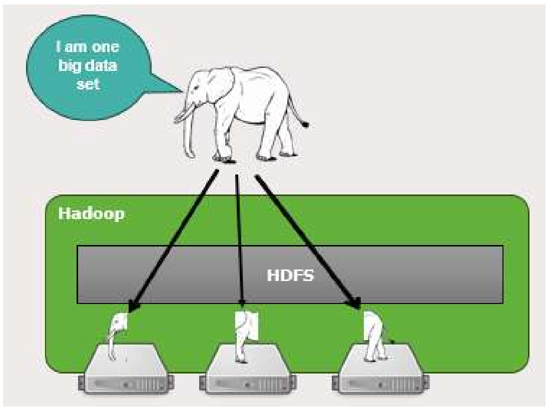
Executes the work on the data near the data

In a nutshell:

HDFS places the data on the cluster and MapReduce does the processing work

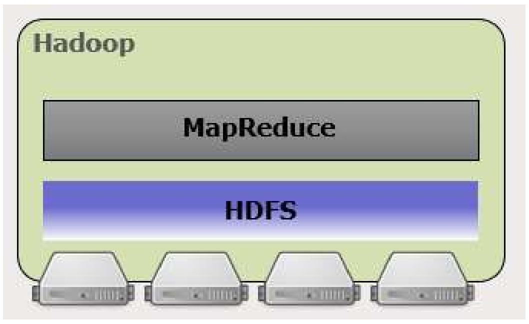
Hadoop Principle

- Hadoop is basically a middleware platforms that manages a cluster of machines
- The core components is a distributed file system (HDFS)
- Files in HDFS are split into blocks that are scattered over the cluster
- The cluster can grow indefinitely simply by adding new nodes



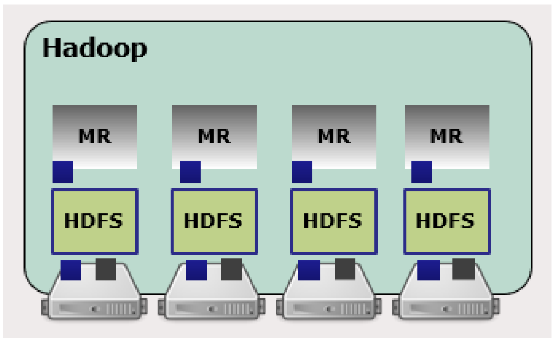
Hadoop Components

MapReduce and Hadoop



Hadoop and MapReduce

- MR works on (big) files loaded on HDFS
- Each node in the cluster executes the MR program in parallel, applying map and reduces phases on the blocks it stores
- Output is written on HDFS



Hadoop Goods & Bads

- Good for:
  - Repetitive tasks on big size data
- Not Good for
  - Replacing a RDMBS
  - Complex processing requiring various phases and/or iterations
  - Processing small to medium size data

GFS: Google File System

Objectives

- Introduction to GFS
- GFS Working Process

GFS: Google File System

- "GFS was built primarily as the fundamental storage service for Google's search engine.
- As the size of the web data that was crawled and saved was quite substantial, Google needed a distributed file system to redundantly store massive amounts of data on cheap and unreliable computers"

Why GFS?

- Component failures
  - Component failures are the norm, not the exception
- Files are huge
  - By traditional standards (many TB)
  - Typically 1000 nodes & 300 TB
- Most mutations are mutations
  - Not random access overwrite
- Co-Designing apps & file system
  - GFS was co-designed with the applications using it

GFS: Design Assumptions?

- Must monitor & recover from comp failures
- Modest number of large files
- Workload:
  - Large streaming reads + small random reads

    - Many large sequential writes
- Need semantics for concurrent
- High sustained bandwidth (More important than low latency)

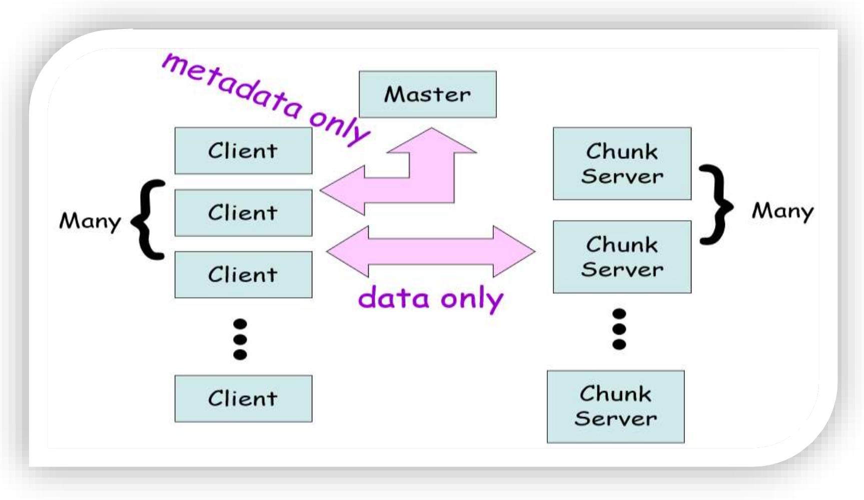
GFS: Interface

Familiar

    - Create, delete, open, close, read, write

- **Novel**
    - Snapshot
    - Low cost
    - Record append
    - Atomicity with multiple concurrent writes

GFS: Architecture



GFS: Architecture details

Objectives

- What are the GFS Architecture Components functions?
- GFS implementation.

GFS Architecture: Master

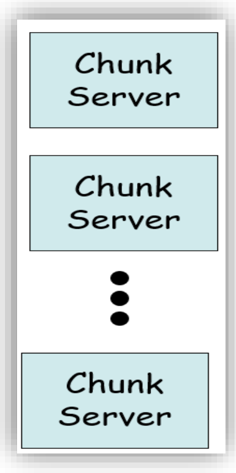
Master

- Stores all metadata
    - Namespace
    - Access-control information
    - Chunk locations
    - ‘Lease’ management
- Heartbeats
- Having one master ➔ global knowledge
    - Allows better placement / replication

        - Simplifies design

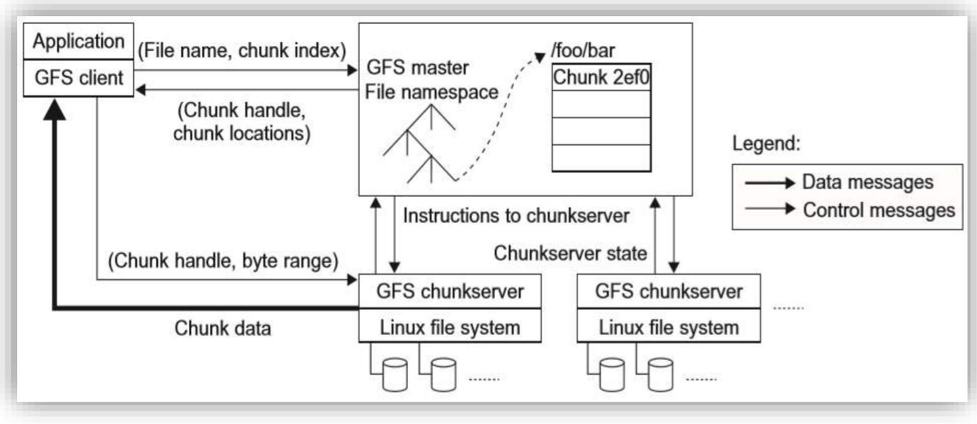
GFS Architecture: Chunk Servers

- Store all files
    - In fixed-size chucks
        - 64 MB
        - 64 bit unique handle
- Triple redundancy



GFS Architecture

- Contact single *master*
- Obtain chunk locations
- Contact one of chunk servers
- Obtain data



GFS Architecture: Master-> Metadata

- Master stores three types
    - File & chunk namespaces
    - Mapping from files → chunks
    - Location of chunk replicas

Stored in memory

Kept persistent through logging

GFS Architecture: Master

Operations

- Replica placement
- New chunk and replica creation
- Load balancing
- Unused storage reclaim

GFS: Consistency Model

- All file namespace mutations are ***atomic***
  - Handled exclusively by the master
- Status of a file region can be
  - ***Consistent:*** all clients see the same data
  - ***Defined:*** all clients see the same data, which include the entirety of the last mutation
  - ***Undefined but consistent***: all clients see then same data but it may not reflect what any one mutation has written
  - ***Inconsistent***

GFS: Leases and Mutation Order

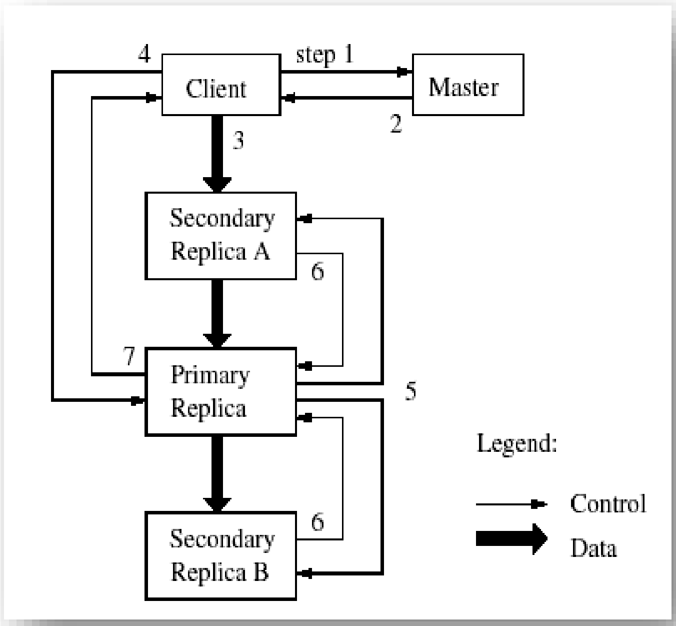
- Master uses *leases* to maintain a consistent mutation order among replicas
- *Primary* is the chunkserver who is granted a chunk lease
- All others containing replicas are *secondaries*
- Primary defines a mutation order between mutations
- All *secondaries* follows this order

GFS Write Control & Dataflow

<u>Mutation Order</u>

→ identical replicas

→ File region may end up containing mingled fragments from different clients (consistent but undefined)



GFS: Limitations

- Custom designed
- Only viable in a specific environment
- Limited security

HDFS: Hadoop Distributed File System

Objectives

- Introduction to HDFS
- HDFS Blocks and Nodes.

HDFS: Background

- At Google MapReduce operation are run on a special file system called Google File System (GFS) that is highly optimized for this purpose.
- GFS is not open source
- Doug Cutting and Yahoo! reverse engineered the GFS and called it Hadoop Distributed File System (HDFS).
- The software framework that supports *HDFS*, MapReduce and other related entities is called the project Hadoop or simply Hadoop
- This is open source and distributed by Apache

HDFS: Basic Features

- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware

HDFS: Basic Features

- HHDFS was designed to be optimal in performance for a WORM (Write Once, Read Many times) pattern
- HDFS is designed to run on clusters of general computers & servers from multiple vendors

HDFS: Blocks

- Files in HDFS are divided into block size chunks
    - 64 Megabyte default block size
- Block is the minimum size of data that it can read or write
- Blocks simplifies the storage and replication process
    - Provides fault tolerance & processing speed enhancement for larger files

HDFS: Nodes

- HDFS clusters use 2 types of nodes
    - Namenode (master node)
    - Datanode (worker node)

HDFS: Nodes

- Namenode
    - Manages the file system namespace
        - Namenode keeps track of the data nodes that have blocks of a distributed file assigned
    - Maintains the file system tree and the metadata for all the files and directories in the tree
    - Stores on the local disk using 2 file forms
        - Namespace Image
        - Edit Log

HDFS: Namenode

- Namenode holds the filesystem metadata in its memory
- Namenode's memory size determines the limit to the number of files in a filesystem
- But then, what is Metadata?

HDFS: Metadata

- Traditional concept of the library card catalogs
- Categorizes and describes the contents and context of the data files
- Maximizes the usefulness of the original data file by making it easy to find and use

  Structural Metadata
    - Focuses on the data structure's design and specification

  Descriptive Metadata
    - Focuses on the individual instances of application data or the data content

HDFS: Metadata Types

- **Structural Metadata**
    - Focuses on the data structure's design and specification
- **Descriptive Metadata**
    - Focuses on the individual instances of application data or the data content

HDFS: Datanodes

- Workhorse of the filesystem
- Store and retrieve blocks when requested by the client or the namenode
- Periodically reports back to the namenode with lists of blocks that were stored

HDFS: Client Access

- *Client* can access the filesystem (on behalf of the user) by communicating with the namenode and datanodes
- Client can use a filesystem interface similar to a POSIX (Portable Operating System Interface)) so the user code does not need to know about the namenode and datanodes to function properly

HDFS: Namenode Failure

- **Namenode keeps track of the datanodes that have blocks of a distributed file assigned**
    - Without the namenode, the filesystem cannot be used
- If the computer running the namenode malfunctions then reconstruction of the files (from the blocks on the datanodes) would not be possible
    - Files on the filesystem would be lost

HDFS: Namenode Failure Resilience

Namenode failure prevention schemes

- Namenode File Backup
- Secondary Namenode

HDFS

Hadoop 2.x Release Series HDFS Reliability Enhancements

- HDFS Federation
- HDFS HA (High-Availability)