CS 621 Final Term

Lecture No: 07

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:

1) •Static network
2) • Dynamic network

Static Network:

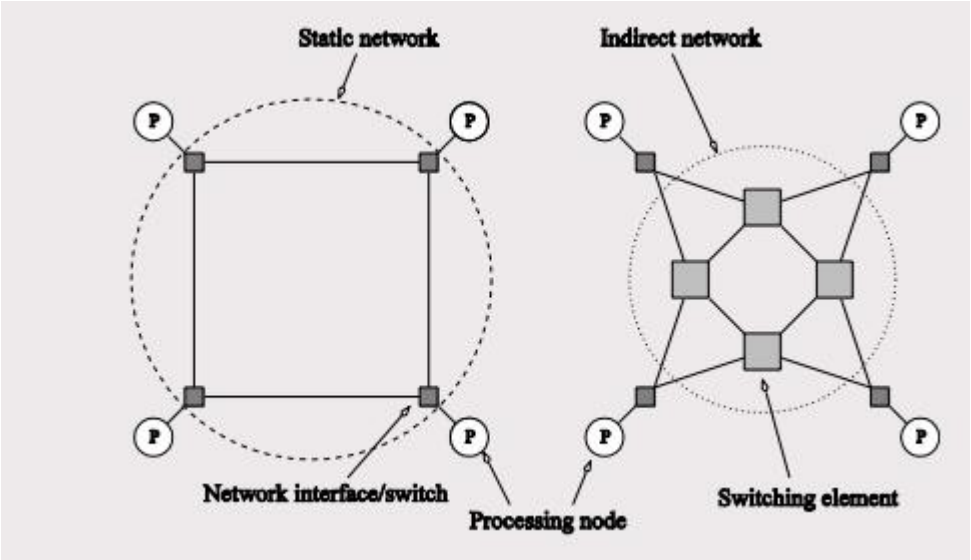
Static networks consist of point-to-point communication links among processing nodes and are also referred to as direct networks. Direct fixed links are established among nodes to form a fixed network.

Dynamic Network:

Connections are established when needed.

Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as **indirect networks**.

Communication links are connected to one another dynamically by the switches to establish paths among processing nodes and memory banks.



Classification of interconnection networks:

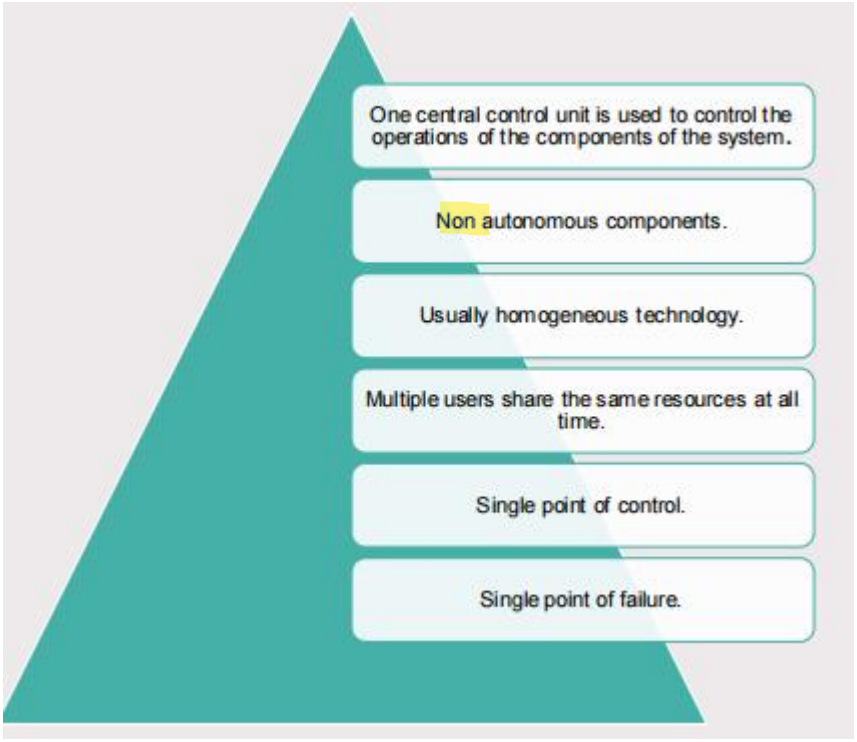
- static network;
- dynamic network.

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:

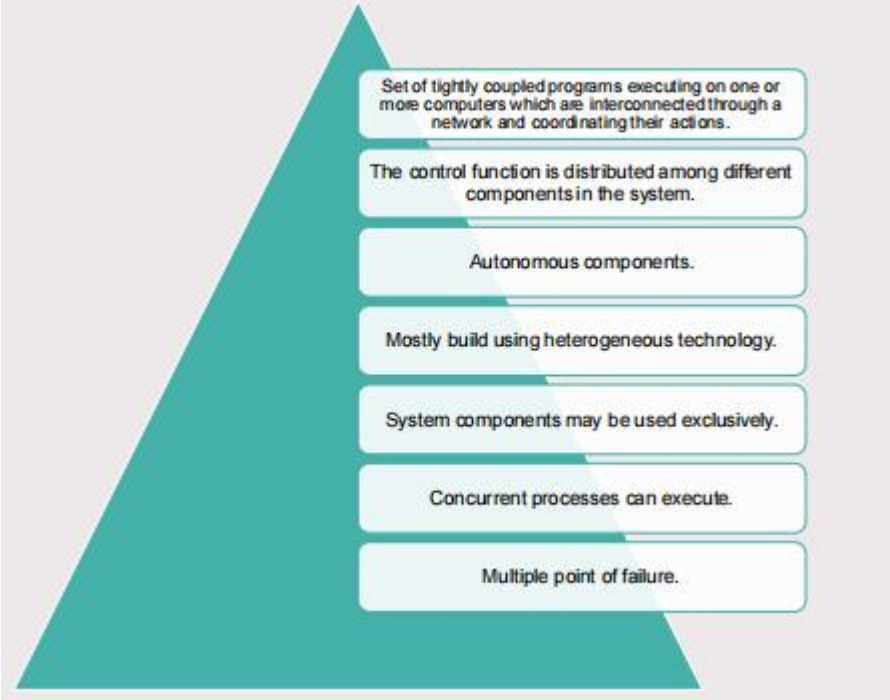
i. Centralized
ii. Decentralized

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.

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:

"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:

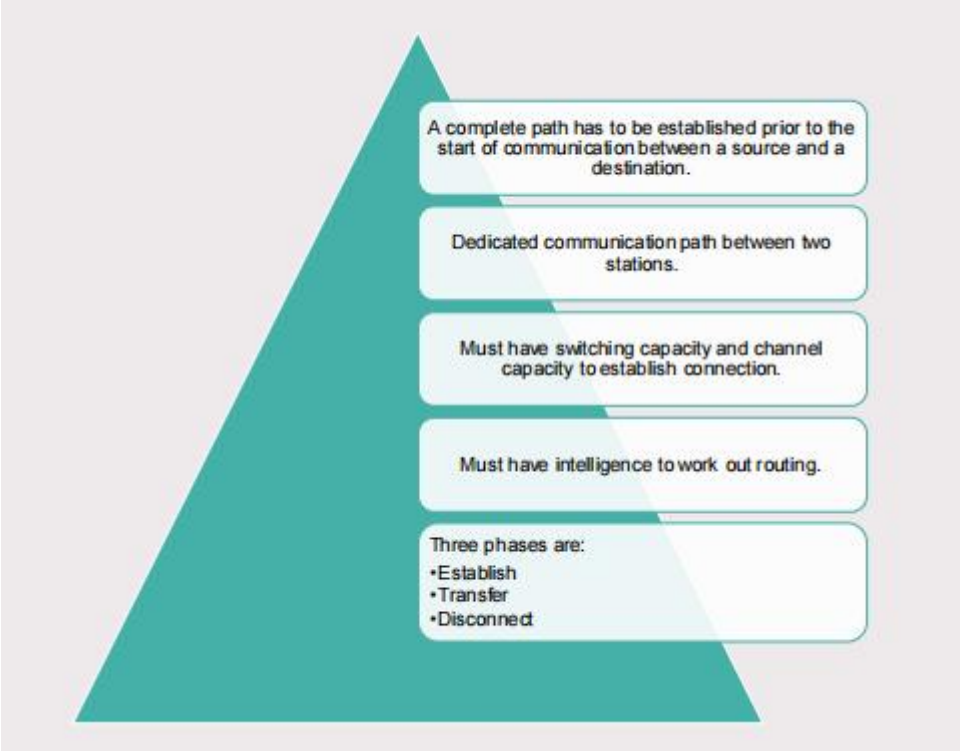
• Computer, terminal, phone, etc.

Two different switching technologies are:

i. Circuit switching

ii. Packet switching

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.

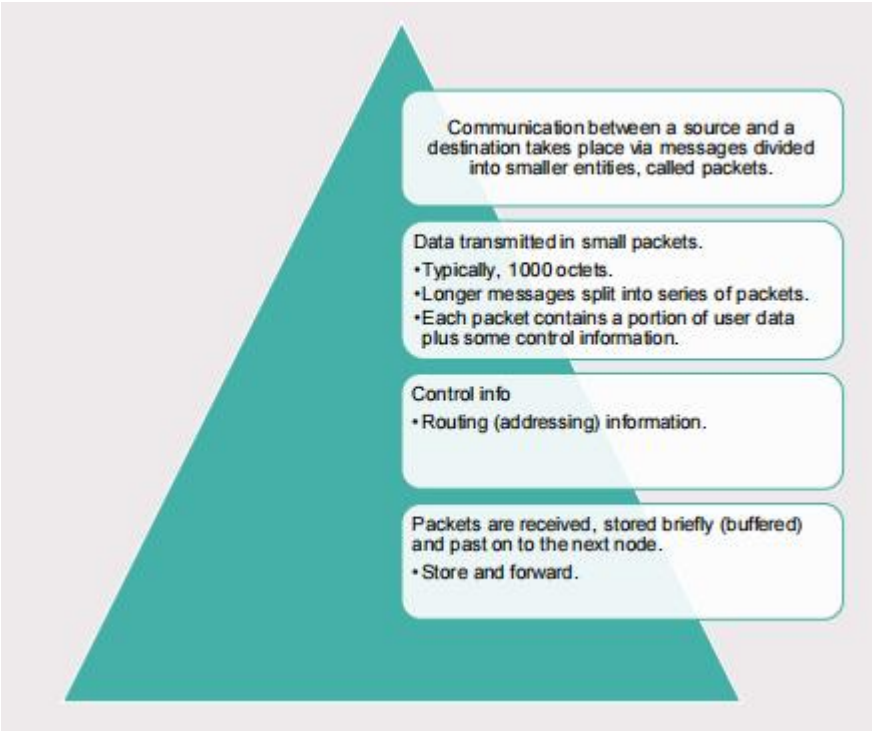
Three phases are:

I. Establish

II. Transfer

III. Disconnect

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:

"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 trade-off 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

Types of Topolgies:

Two basic types of topologies are:

1) Static topology.
2) Dynamic topology

Static Connection Networks:

I. • RingHypercube (Loop) networks
II. • Mesh
III. • Torus
IV. • Tree networks
V. • Hypercubenetwork

Dynamic Connection Networks:

I. • Bus-based
II. • Switch-based

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:

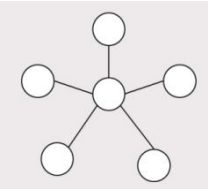
Number of bits transmitted in a unit of time.

Bisection:

The number of connections that need to be cut to partition the network in 2.

Static Topologies: Star

"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."



nodes distance 0(1)

A star connected network of nine nodes.

• 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.

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."

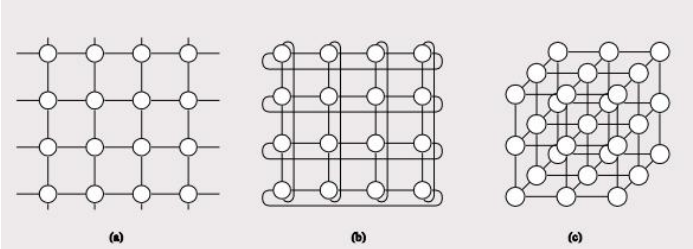
Two-dimensional mesh is an extension of the linear array to two dimensions.

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 n=2n processors

represents loosely coupled

d=log p

"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."

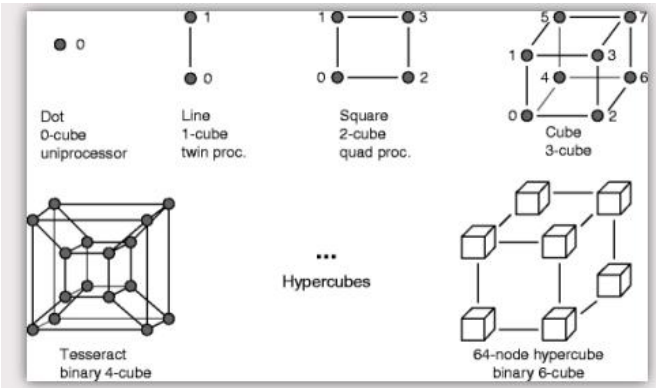
A special case of a dimensional mesh is a hypercube. Here, d = log p,

where p is the total number of nodes.

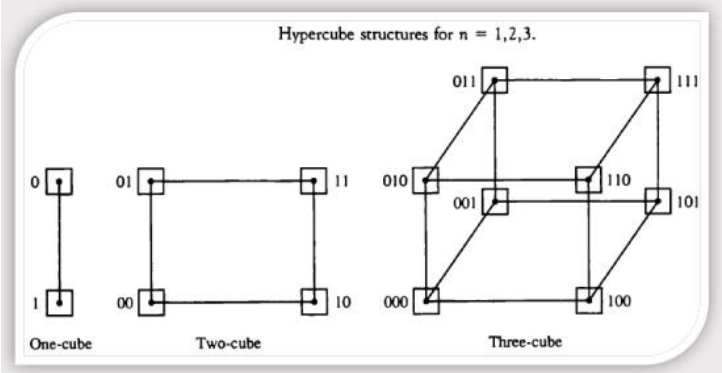
The distance between any two nodes is at most log p. Each node has log p neighbors.

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.



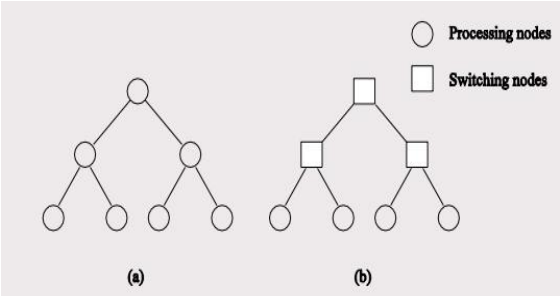
Construction of hypercubes from hypercubes of lower dimension.

Static Topologies: Tree

distance=2logp

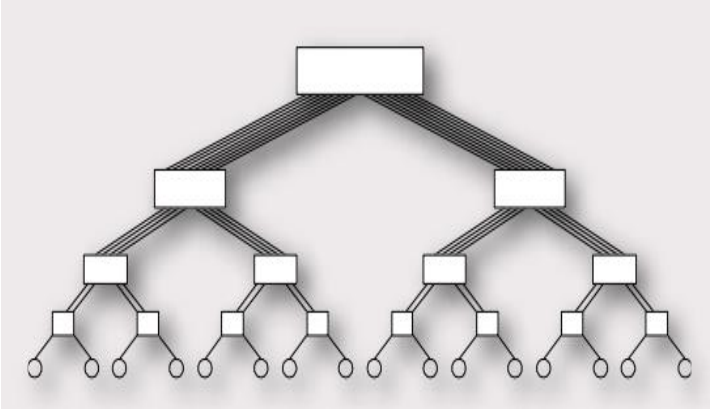
"A tree network is one in which there is only one path between any pair of nodes."

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 starconnected 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

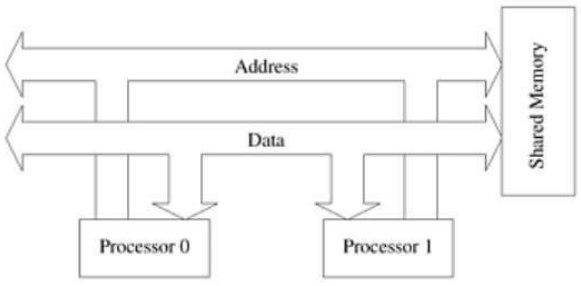
Dynamic Topologies:

Buses

"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."

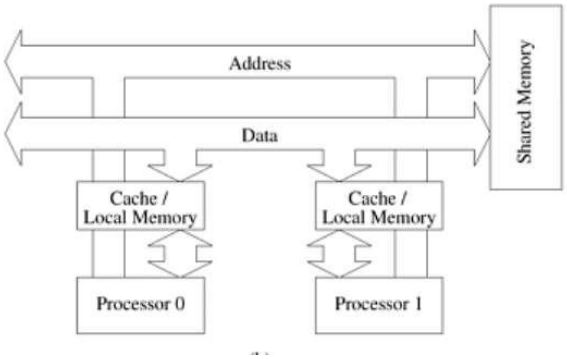
Some of the simplest and earliest parallel machines used buses. All processors access a common bus for exchanging data. A bus has the desirable property that the cost of the network scales linearly as the number of nodes, p. This cost is typically associated with bus interfaces. The distance between 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 Topolgies:Bus

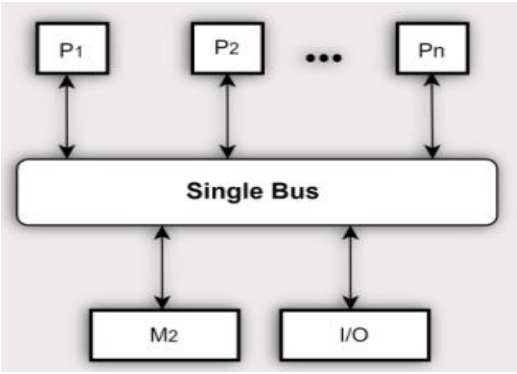
Bus-based dynamic topologies are broadly classified into two categories:

1) • Single bus
2) • Multi bus

SingleBus:

• 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:

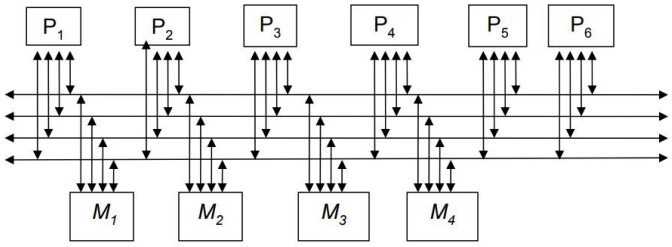
- • Bandwidth of bus.
- • 1 processor can access the bus.
- • 1 memory access can take place at any given time.



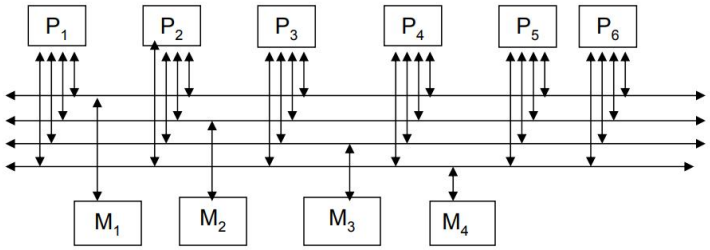
Multi Bus:

- • 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).

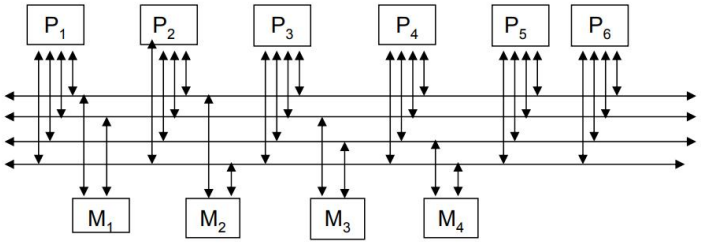
Multiple Bus with Full Bus – Memory Connection (MBFBMC):



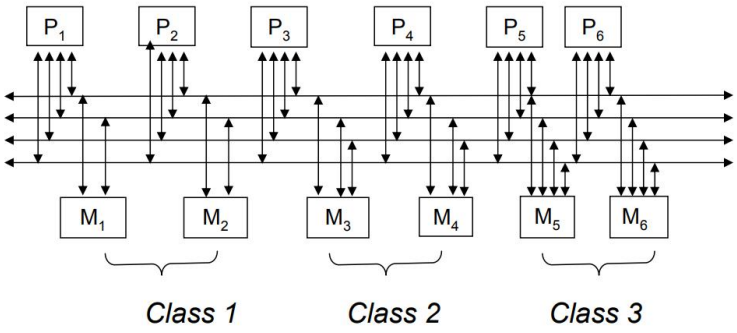
Multiple Bus with Single Bus – Memory Connection (MBSBMC)



Multiple Bus with Partial Bus – Memory Connection (MBPBMC)



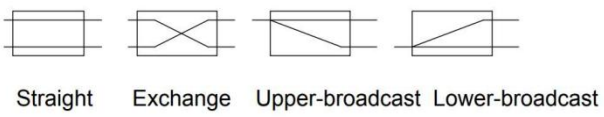
Multiple Bus with Class-based Bus – Memory Connection (MBCBMC).



Switch-based Dynamic Topologies:

Single-Stage Switch:

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



Multi-Stage Switch:

Crossbars have excellent performance scalability but poor cost scalability.

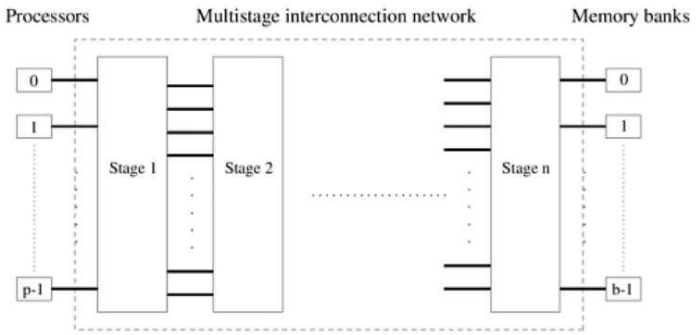
Buses have excellent cost scalability, but poor performance scalability.

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.

used omega 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.

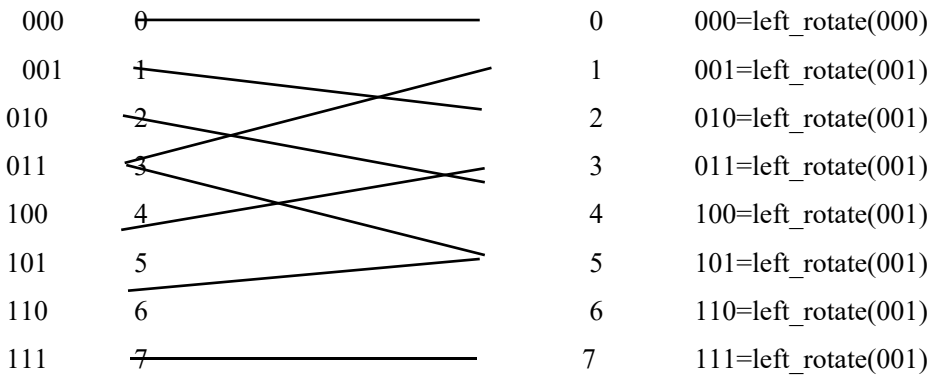
log p stages, p is the number of i/o

• At each stage, input i is connected to output j

• if:

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

• 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

cost of cross bar is o(p)^2

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.

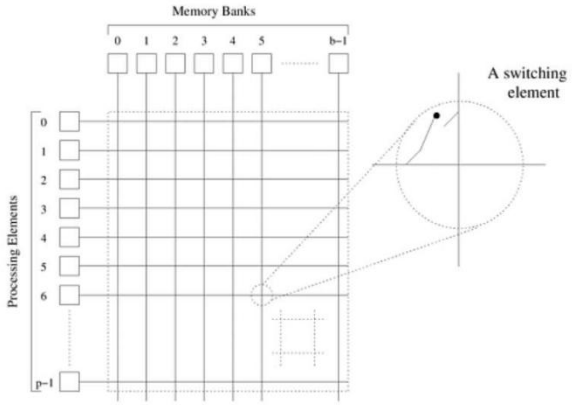
A Switching Element (SE) is at the intersection of any 2 lines extended horizontally or vertically inside the switch. It is a non-blocking network allowing multiple input output connection pattern to be achieved simultaneously.

Examples of machines that employ crossbars include the Sun Ultra HPC 10000 and the Fujitsu VPP500.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

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 of the hypercube is p/2
completely connected network is p^2/4

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	Arc Connectivity	Cost (No. Of Links)
Star	**2**	**1**	**1**	$\mathbf{p-1}$
Linear array	$\mathbf{p-1}$	**1**	**1**	$\mathbf{p-1}$
2-D mesh, no wraparound	$\mathbf{2(\sqrt{p}-1)}$	$\mathbf{\sqrt{p}}$	**2**	$\mathbf{2(p-\sqrt{p})}$
2-D wraparound mesh	$\mathbf{2(\sqrt{p}/2)}$	$\mathbf{2\sqrt{p}}$	**4**	**2p**
Hypercube	$\mathbf{\log p}$	$\mathbf{p/2}$	$\mathbf{\log p}$	$\mathbf{(p\log p)/2}$

Analysis and Performance Metrics:

Dynamic 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

Lecture No: 08

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 and scaling Technique:

Replication and caching for performance are widely applied as scaling technique.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

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?

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:

i. Data-centric Consistency Models

ii. Client-Centric Consistency Models

Data-centric Consistency Models:

I. Continuous Consistency

II. Consistent Ordering of Operations

III. Causal Consistency

IV. Grouping Operations

Client-Centric Consistency Models:

i. Eventual Consistency

ii. Monotonic Writes

iii. Read Your Writes

iv. Writes Follow Reads

Data-Centric Consistency Models:

Continuous Consistency Data Consistency Model

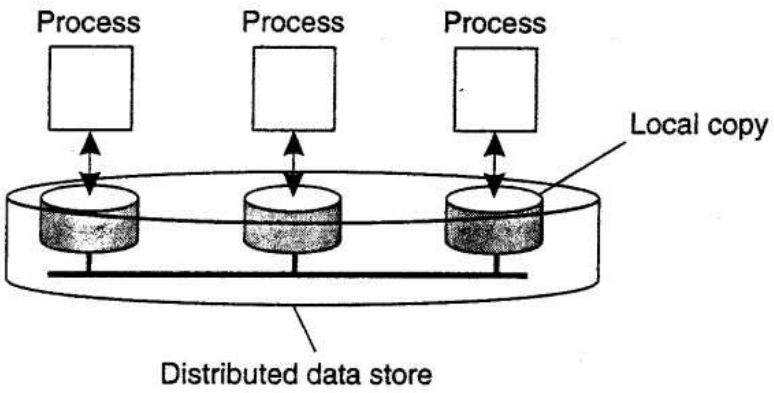
"A contract between processes and the data store that says that if processes agree to obey certain rules, the store promises to work correctly."

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.

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

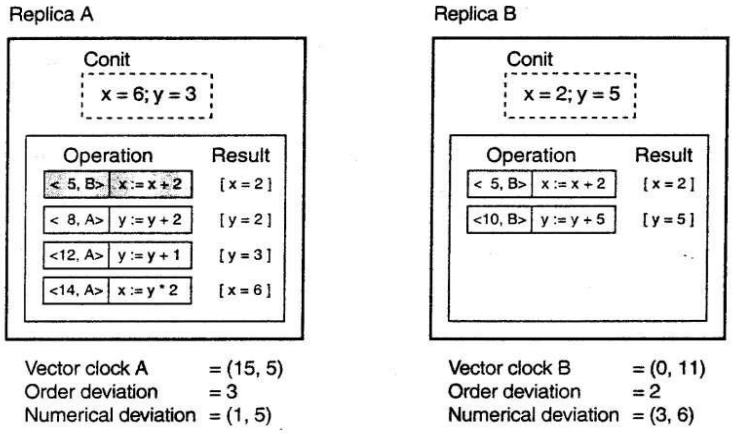
• B sends A operation [h5,Bi: x := x + 2];

• 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)



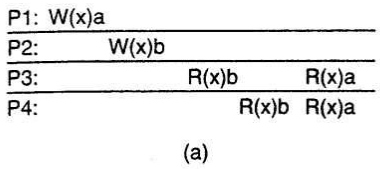
An example of keeping track of consistency deviations [adapted from (Yu and Vahdat, 2002)].

Data-Centric Consistency Models:

Sequential Consistency:

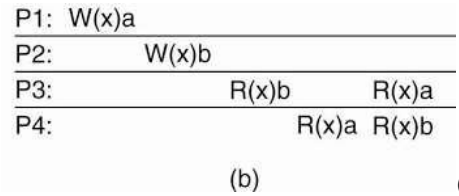
"The result of any execution is the same as if the (read and write) operations by all processes on the data-store were executed in the same sequential order and the operations of each individual process appear in this sequence in the order specified by its program."

• 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

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.

Casual 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.

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

(a)

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

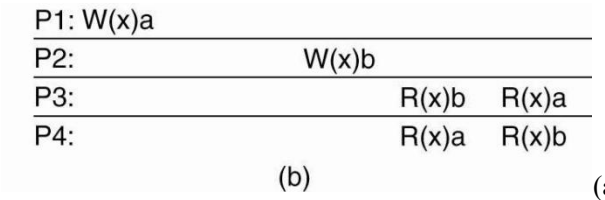
Example 2:

• 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.



(b)

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

Client-Centric Consistency Models:

Eventual Consistency:

" A special class of distributed data-store which is characerized 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 update 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.

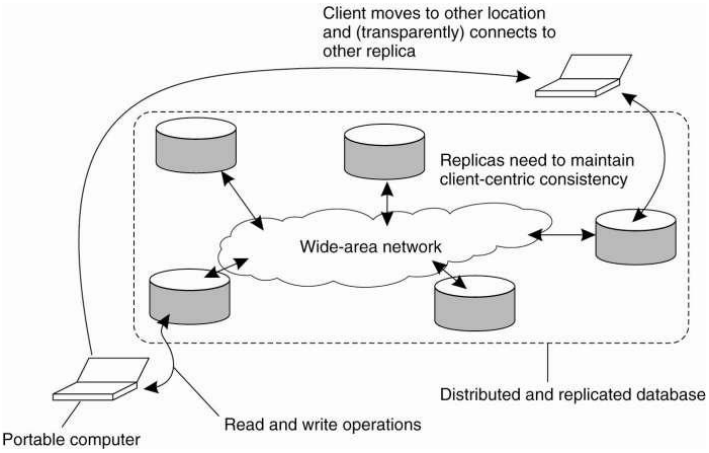
• At location A you access the database doing reads and updates.

• 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



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

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.

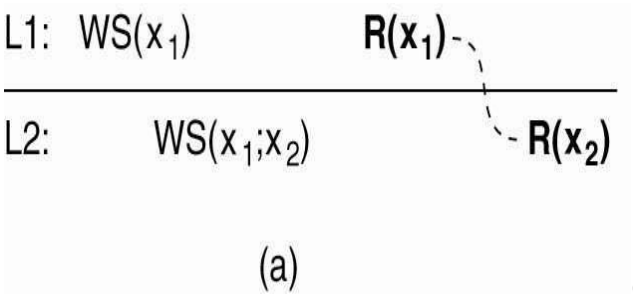
• Operations carried out by a single process P in boldface are connected by a dashed line representing the order in which they are carried out.

• 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.



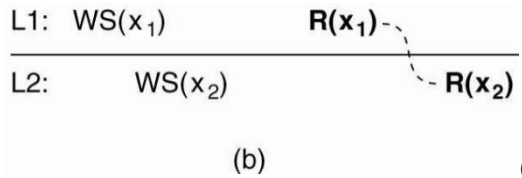
(a) A monotonic-read consistent data store.

• 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

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

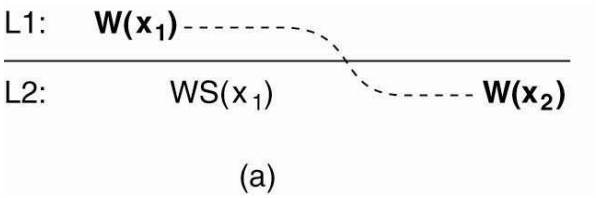
Updating:

Updating a program at server S2, and ensuring that all components on which compilation and linking depends, are also placed at S2.

Maintaining:

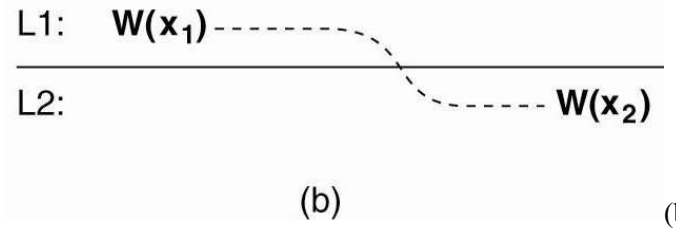
Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

• 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

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.

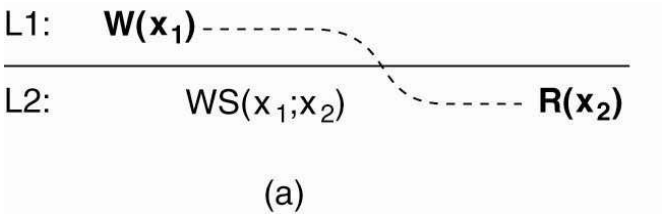
A write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.

Example: Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.

• Process P performed a write operation W(x1) and later a read operation at a different local copy.

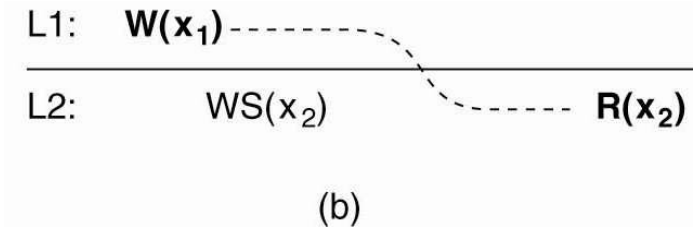
• Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation.

• 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:

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.

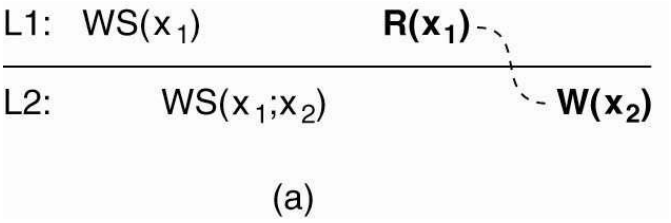
Example: See reactions to posted articles only if you have the original posting (a read .pulls in. the corresponding write operation).

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.

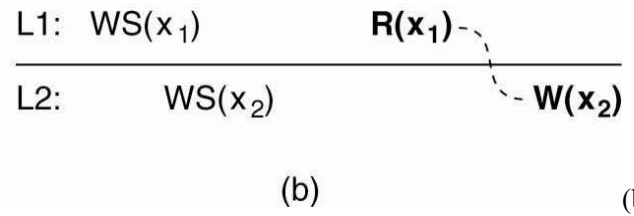
• (Note that other processes at L2 see those write operations as well).



(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

Lecture No: 09

Introduction to GPU:

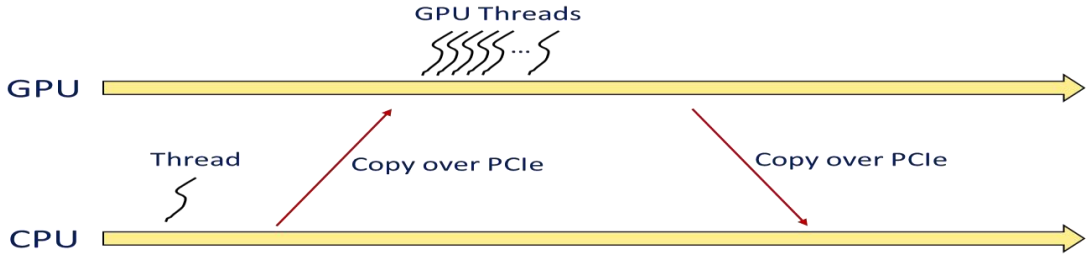
"Graphics Processing Unit ()GPU) is a chip or electrnic circuict capable of rendering graphics for display on an electronic device. GPUs work by using a method called parallel processing, where multiple processors handle seprate parts separate parts of the same task."

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 conventionalmicro processor 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	CPU
GPU is designed for highly parallel operations	CPU execute the programs serially
GPUs have many parallel executing units	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	
GPUs have much deeper pipelines several thousand	

stages vs 10 to 20 for CPUs	

Application of GPU:

Many applications have been developed to use GPUs for super-computing 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

Parallel coprocessor to conventional CPUs

Implement a SIMD structure, multiple threads running the same code.

Gris of Block of Threads:

- Thread local registers
- Block local memory and control
- Global memory

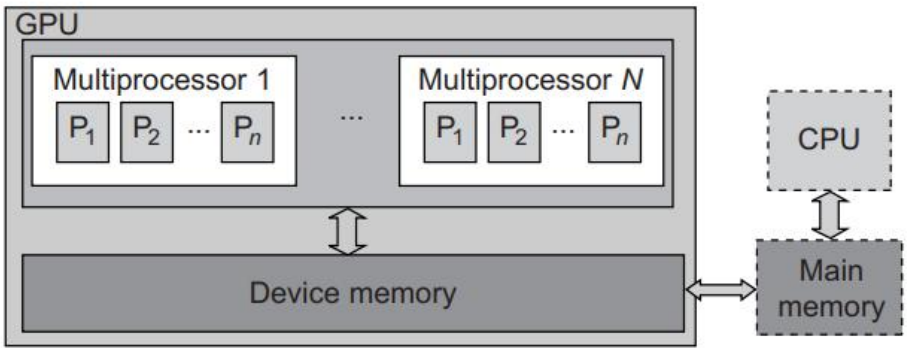
The CPU is the conventional multi-core processor with limited parallelism to exploit.

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.

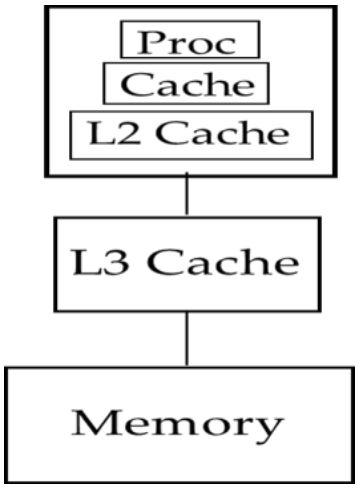
Essentially, the CPU's floating-point kernel computation role is largely offloaded to the manycore GPU. The CPU instructs the GPU to perform massive data processing

• Scale code to hundreds of cores running thousands of threads.

• The task runs on the GPU independently from the CPU.



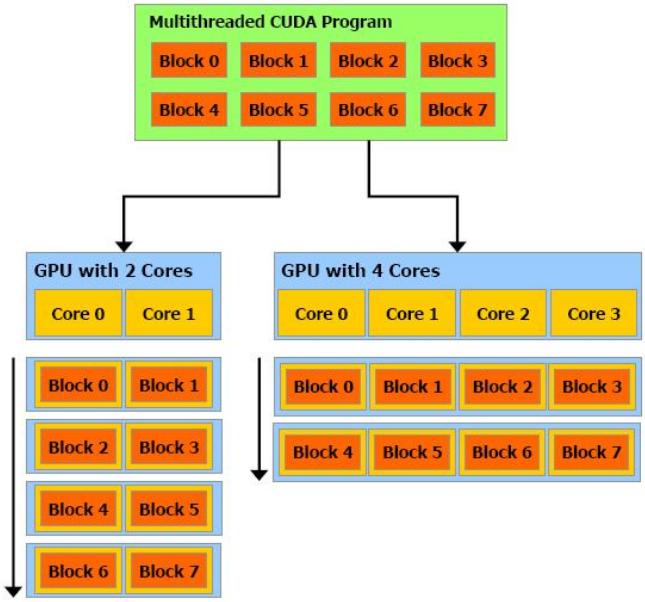
The use of a GPU along with a CPU for massively parallel execution in hundreds or thousands of processing cores.



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

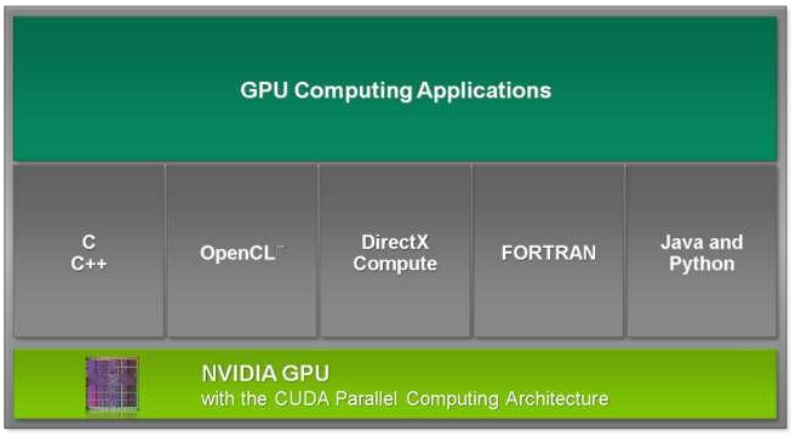
"GPU Programming is a method of running highly parallel general-purpose computations on GPU accelerators. While the past GPUs were designed exclusively for computer graphics, today they are being used extensively for general-purpose computing (GPGPU computing) as well."

GPU Programming Models: CUDA

CUDA is NVIDIA's general purpose parallel computing architecture .

• Designed for calculationintensive computation on GPU hardware.

• CUDA is not a language, it is an API.



• 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:

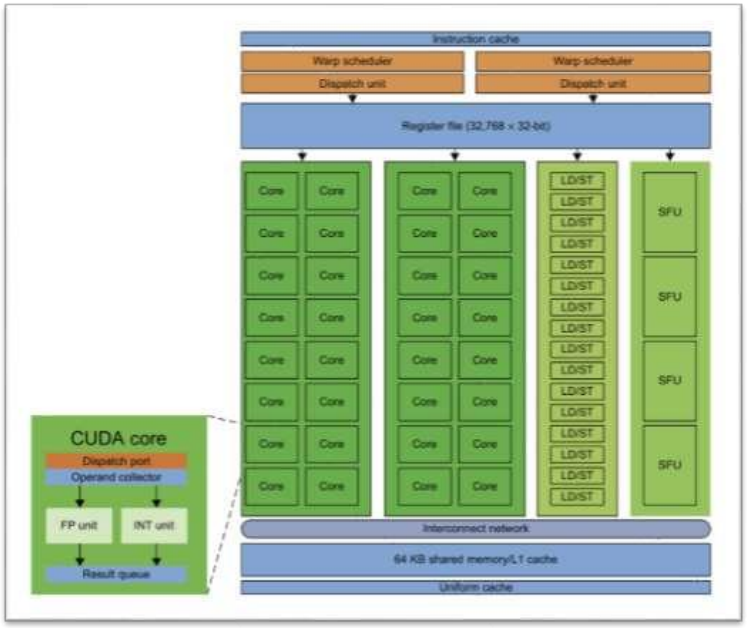
• Is a coprocessor to the CPU or host

• 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

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

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

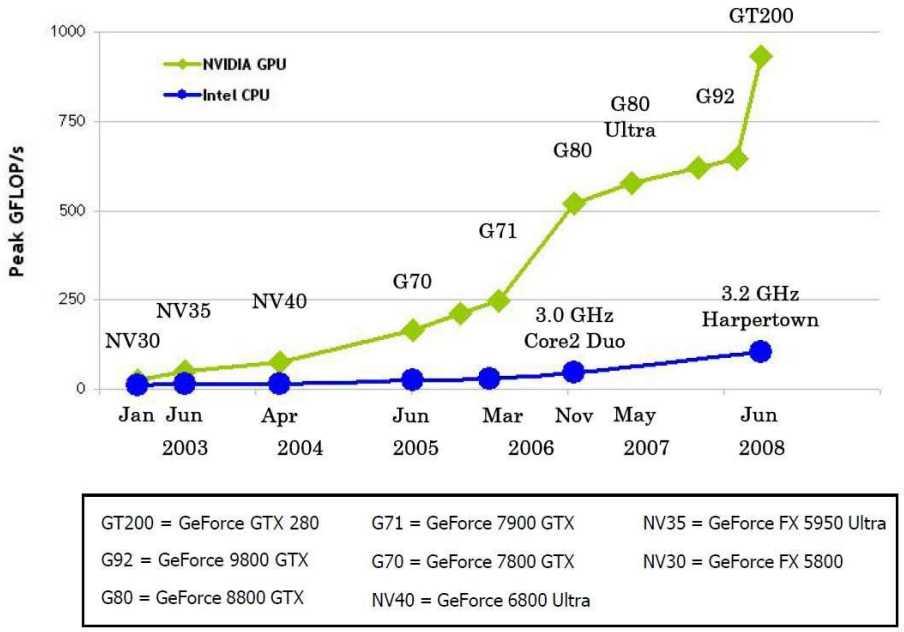
High-End CPU today

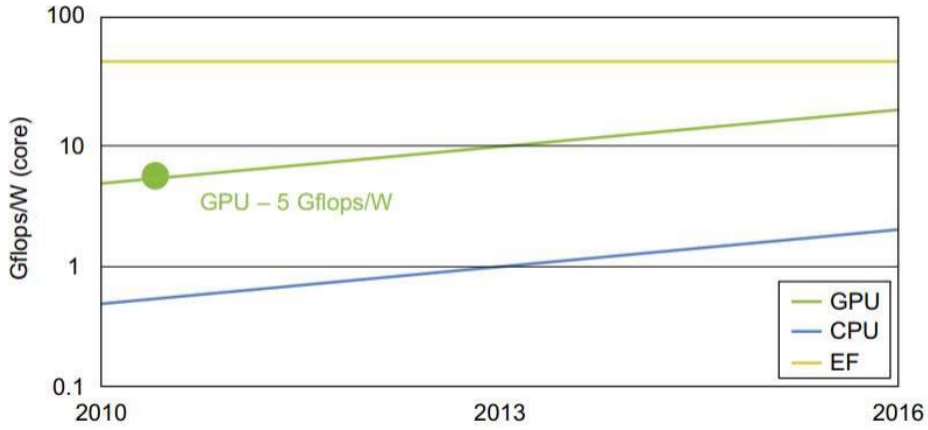
• 3.4Ghz x 8 FOPS/cycle = 27 GFLOPS
• Assumes SSE

High-End GPU today / GTX280

• 933.1 GFLOPS or 34x capability

Power Efficiency of GPU: GPU vs. CPU





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?

"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?

"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 Migeration: Motivation:

Performance:

• Move code on a faster machine.

• Move code closer to data.

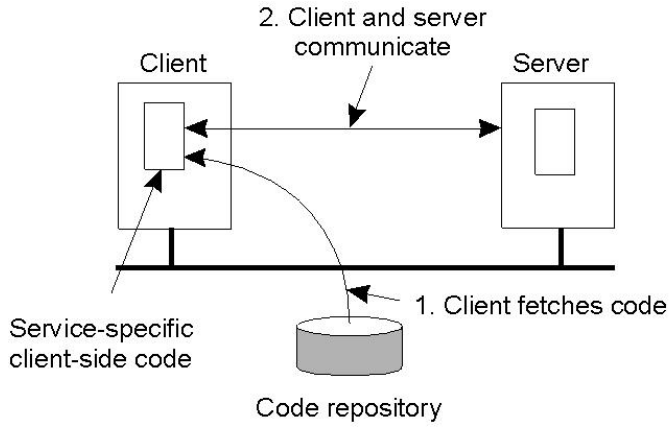
Flexibility:

• 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



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

Process model for code migration (Fugetta et al., 98)

• Code segment: set of instructions that make up the program

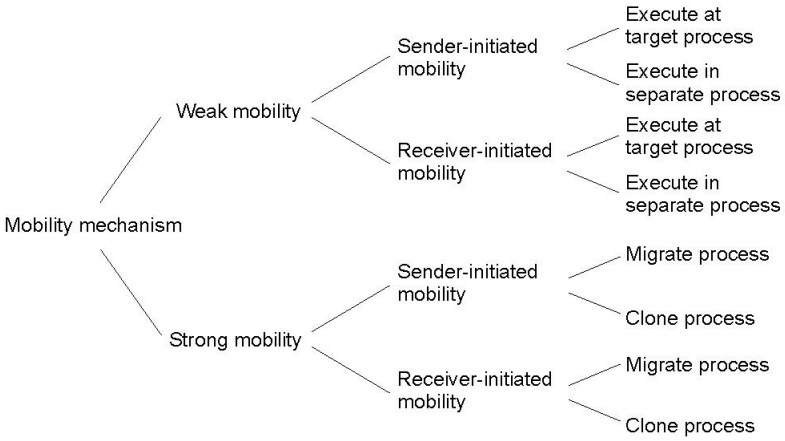
• Resource segment: references to external resources

• Execution segment: store current execution state

Type of mobility

• **Weak mobility:** migrate only code segment

• **Strong mobility**: migrate execution segment and resource segment



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, SR)	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

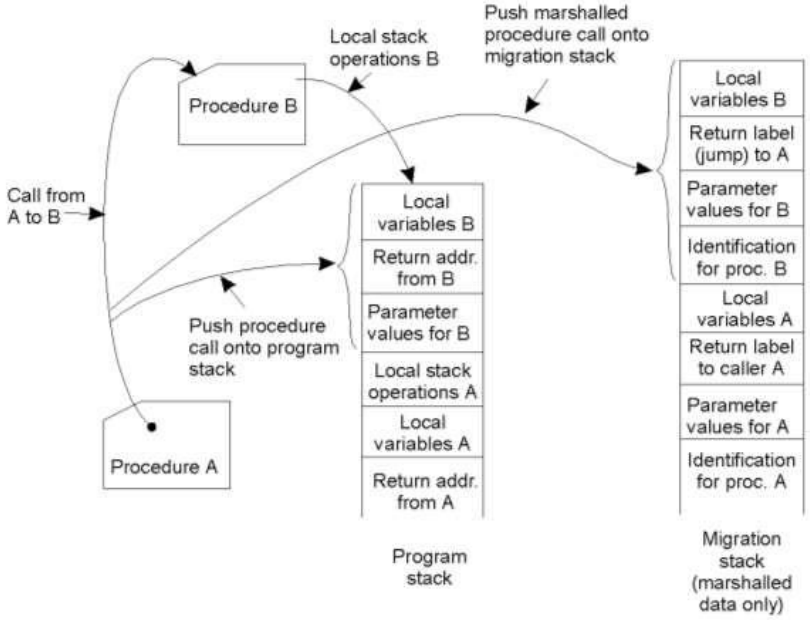
• CP: copy the value of resource

• RB: rebind the process to locally available resource

Code Migration in Heterogeneous Systems

• 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

"virtual machines (VMs) can be a useful tool for handling heterogeneity in computing enviornments. VMs allow for the creation of multiple virtuailzed instances of operaring systems and applications, which can be run on a single physical machine. This allows for the consolidation of multiple computing enviornments 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 re-sending 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.

Lecture No: 10

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:

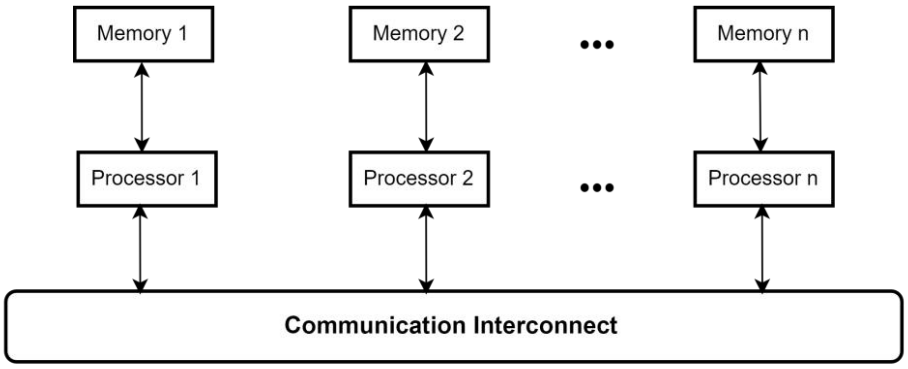
Message passing is the most commonly used parallel programming approach in distributed memory systems. Here, the programmer has to determine the parallelism. In this model, all the processors have their own local memory unit and they exchange data through a communication network.

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)?

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

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 messagepassing 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

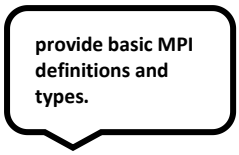
–MPI_COMM_SIZE

–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:

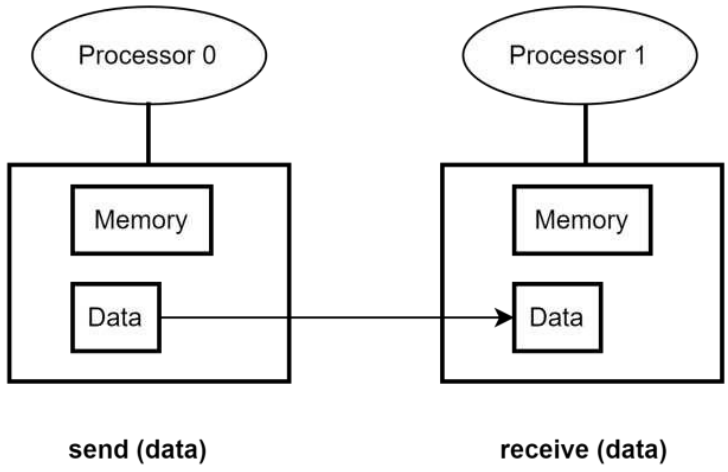
Distributed memory processes have access only to local data. The sender process issues a send call, and the receiver process issues a matching receive call.

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.

Un-buffered:
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:
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).

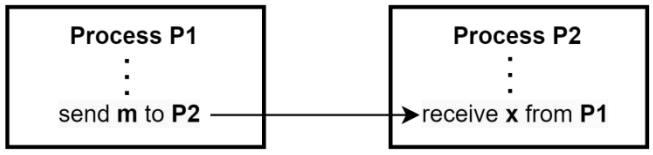
BLOCKING send and receive operations.

• 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

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

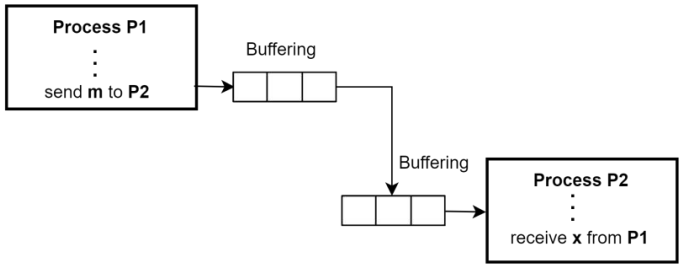
• Receive operation completes when message has been delivered.

• 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

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. Vendor implementations should be able to exploit native hardware features to optimize performance.

Functionality:

Over 115 routines are defined in MPI-1 Alone.

Scalability:

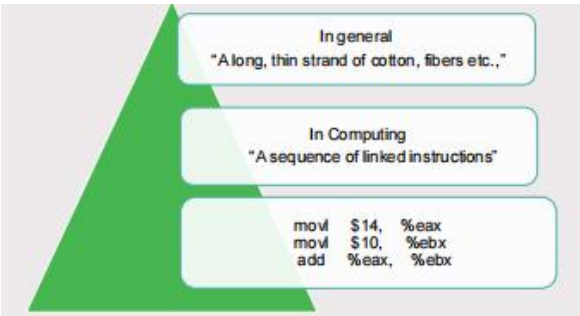
MPI is designed to scale to large numbers of processors, which makes it well-suited for high performance computing.

Availability:

A variety of implementations are available, both vendor and public domain.

Lecture No: 11

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

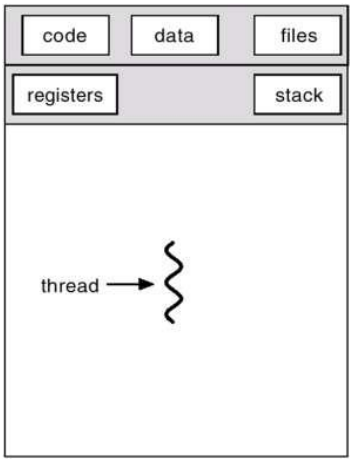
Executes within the address space of a process

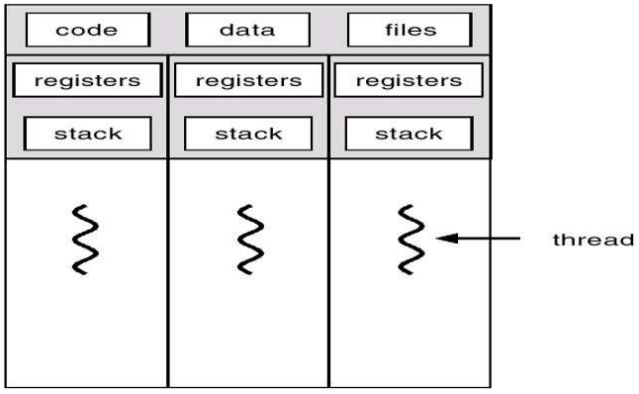
Lightweight process

Can be scheduled to run on a CPU as an independent unit and terminate

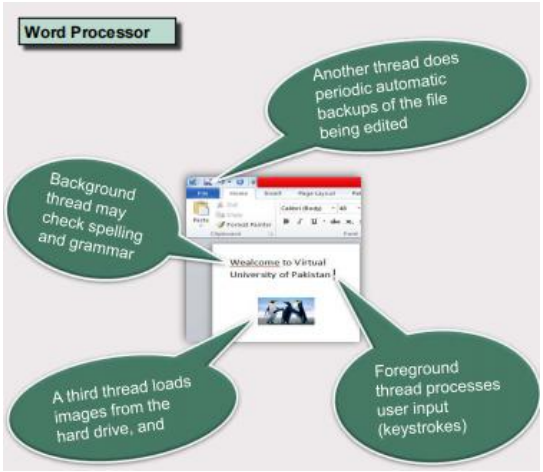
Multiple threads can run simultaneously .

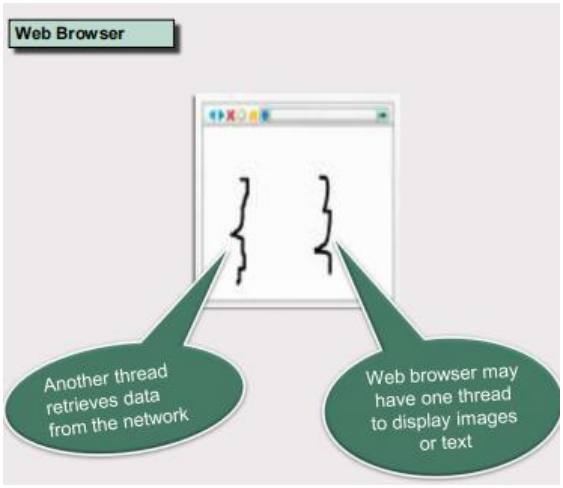
Single thread vs Multiple threads





Threads Example:



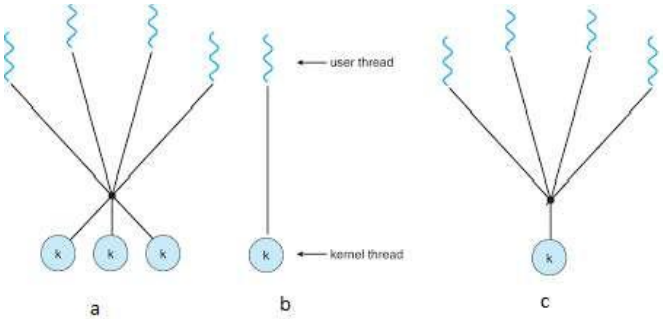


Thread Model?

• Many to Many

• One to One

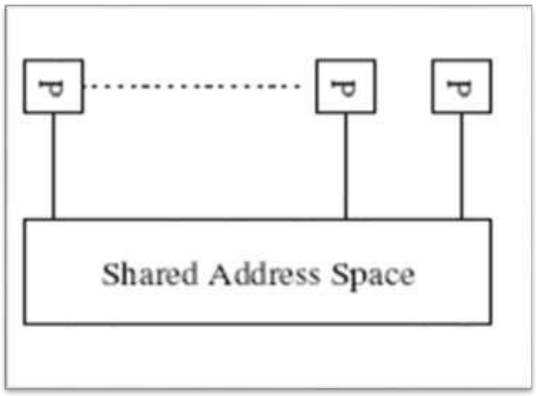
• Many to One



• **Thread Example**

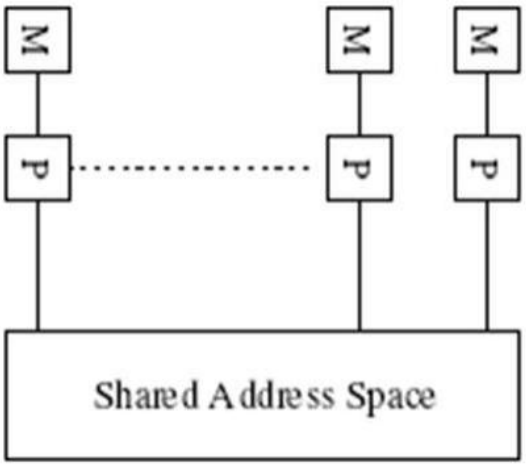
```
1    for (row = 0; row < n; row++)
2         for (column = 0; column < n; column++)
3              c[row][column] =
4                   create_thread(dot_product(get_row(a, row),
5                                             get_col(b, col)));
```

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



The POSIX Thread API

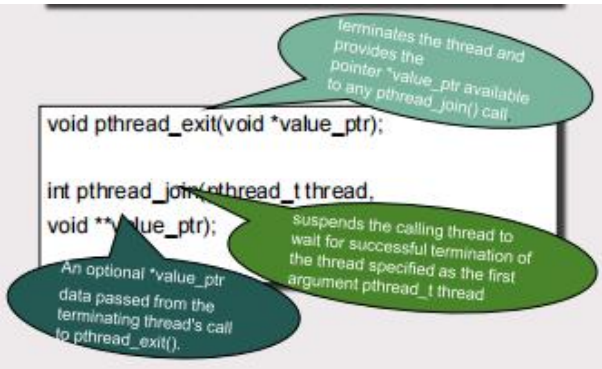
A number of vendors provide vendor-specific thread APIs (NT threads, Solaris threads, Java threads, etc.)

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.

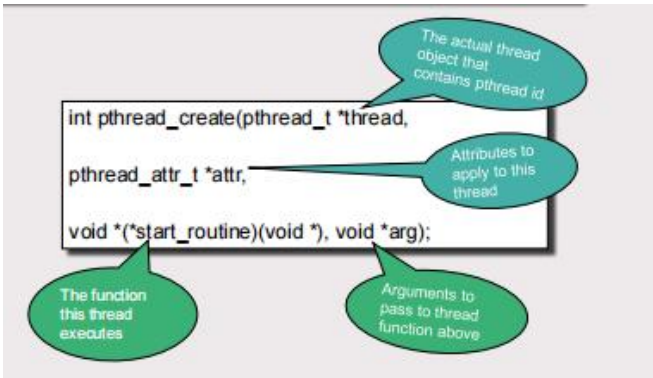
Thread Termination:

A pthread is represented by the type pthread_t.



Thread Creation

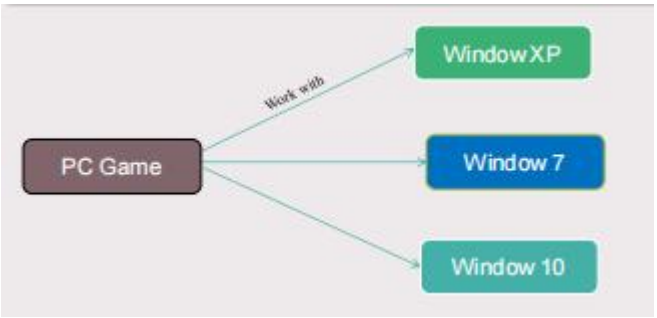
A pthread is represented by the type pthread_t.



Why Thread?

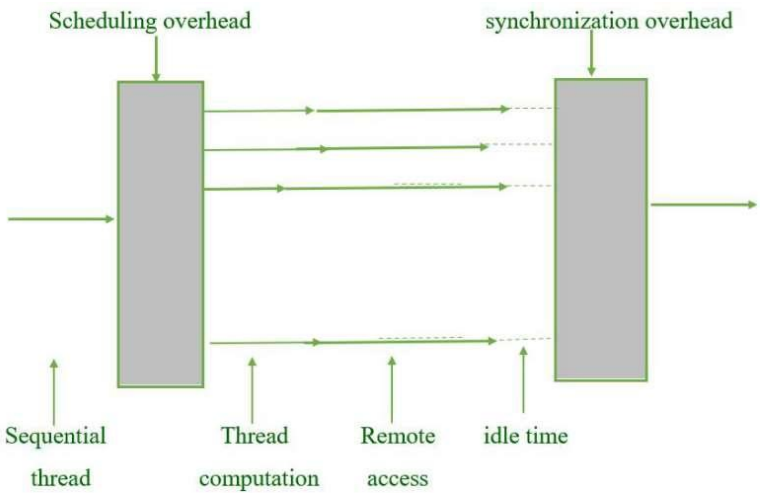
Portability:

The possibility to use the same software in different environments



Latency hiding

Multithreading enables latency reduction/hiding



Scheduling and Load Balancing

Ease of Programming

Threaded programs are significantly easier towrite 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 :

What is the biggest challenge with using thread?

Communication is implicit in shared-addressspace programming.

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

Synchronization:

What will happen when multiple threadsattempt 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

Mutual Exclusion

The Pthreads API functions for handling mutex locks:

• int pthread_mutex_lock (pthread_mutex_t *mutex_lock)

The Pthreads API functions for handling mutex-locks:

int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);

The Pthreads API functions for handling mutexlocks:

• 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

1. A normal mutex
2. A recursive mutex
3. Error check mutex

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:

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);

Pthreads functions for condition variables

• int pthread_cond_signal(pthread_cond_t *cond);

Pthreads functions for condition variables

• int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

• int pthread_cond_destroy(pthread_cond_t *cond);

Pthreads functions for condition variables

• int pthread_cond_broadcast(pthread_cond_t *cons);

Lecture No: 12

Principles of Parallel Algorithm Design

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?

Dividing a computation into smaller computations

Assigning them to different processors for parallel execution

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?

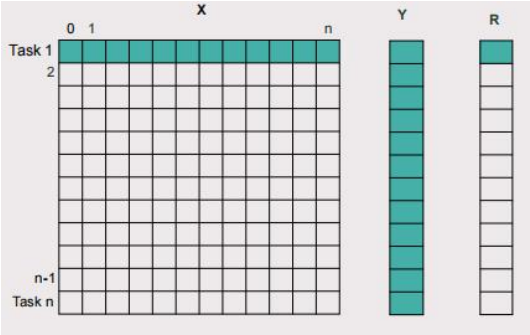
"Tasks are programmer-defined units of computation into which the main computation is subdivided by means of decomposition."

Types of tasks in term of inter-dependency?

Independent tasks

Dependent tasks

Example decomposition: Dense matrix-vector Multiplication



Task dependency graph:

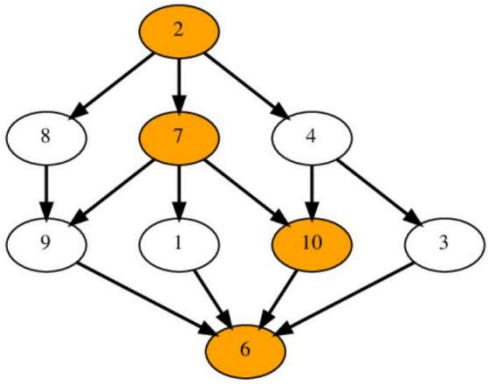
Decomposition can be illustrated in the form of a directed graph with:

• 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 a Critical Path Length?

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:

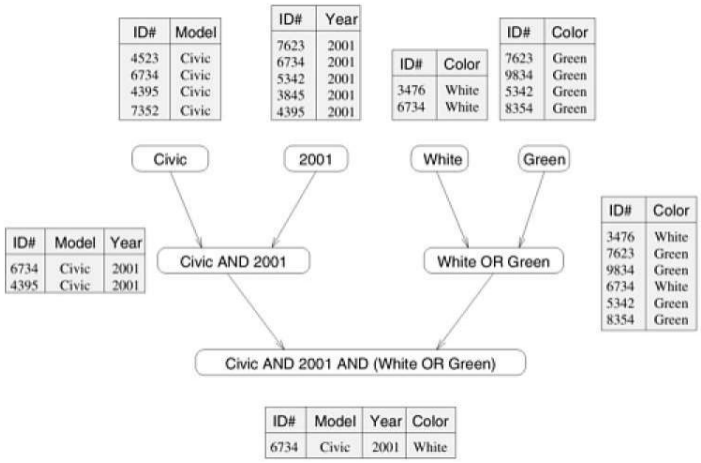
Consider the execution of the query:

“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 (Task Size)

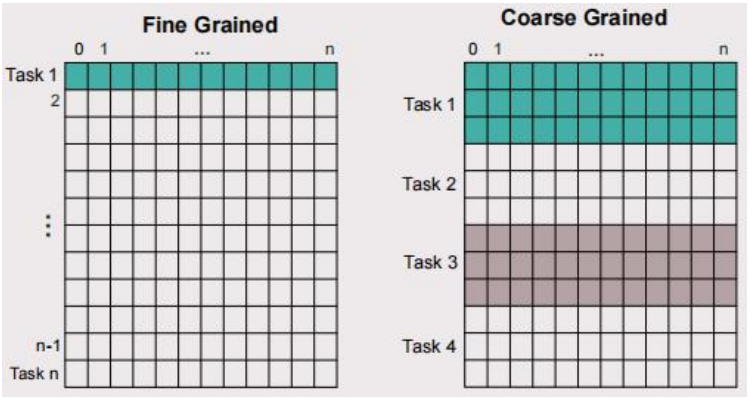
Fined-grained

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.

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:

i. Recursive Decomposition

ii. Data Decomposition

iii. Exploratory Decomposition

iv. Speculative Decomposition

Recursive Decomposition:

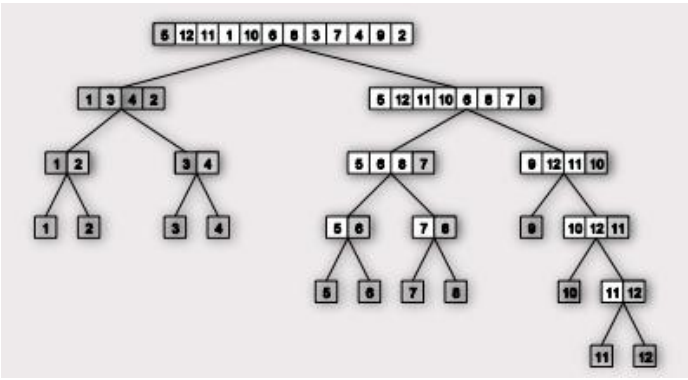
Generally suited to problems that are solved using the divide-and-conquer strategy.

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.

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**;

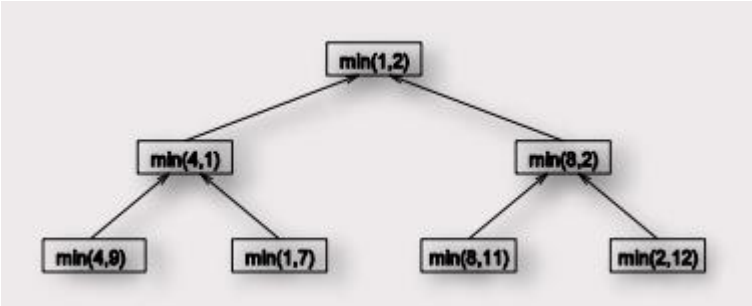
13. **endelse**;

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

"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 step1, the data on which the computations are performed is partitioned.

In step2, this data partitioning is used to induce a partitioning of the computations into tasks.

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."

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

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

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

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

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

Output Data Partitioning:

Decomposition I	**Decomposition II**
Task 1: ***C*1,1 = *A*1,1 *B*1,1**	Task 1: ***C*1,1 = *A*1,1 *B*1,1**
Task 2: ***C*1,1 = *C*1,1 + *A*1,2 *B*2,1**	Task 2: ***C*1,1 = *C*1,1 + *A*1,2 *B*2,1**
Task 3: ***C*1,2 = *A*1,1 *B*1,2**	Task 3: ***C*1,2 = *A*1,2 *B*2,2**
Task 4: ***C*1,2 = *C*1,2 + *A*1,2 *B*2,2**	Task 4: ***C*1,2 = *C*1,2 + *A*1,1 *B*1,2**
Task 5: ***C*2,1 = *A*2,1 *B*1,1**	Task 5: ***C*2,1 = *A*2,2 *B*2,1**
Task 6: ***C*2,1 = *C*2,1 + *A*2,2 *B*2,1**	Task 6: ***C*2,1 = *C*2,1 + *A*2,1 *B*1,1**
Task 7: ***C*2,2 = *A*2,1 *B*1,2**	Task 7: ***C*2,2 = *A*2,1 *B*1,2**
Task 8: ***C*2,2 = *C*2,2 + *A*2,2 *B*2,2**	Task 8: ***C*2,2 = *C*2,2 + *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.

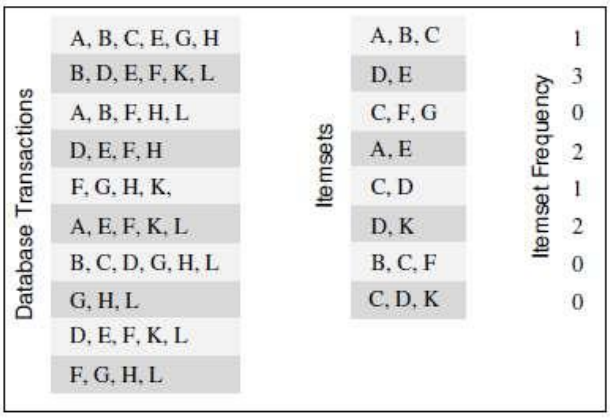
In many cases, this is the only natural decomposition because the output is not clearly known a-priori (e.g., the problem of finding the minimum in a list, sorting a given list, etc.).

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.

Input Data Partitioning:

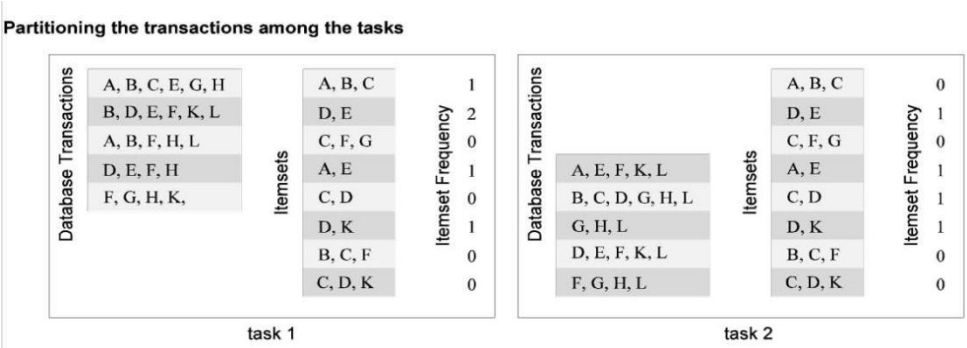
Example

Transactions (input), item sets (input), and frequencies (output)



Input Data Partitioning:

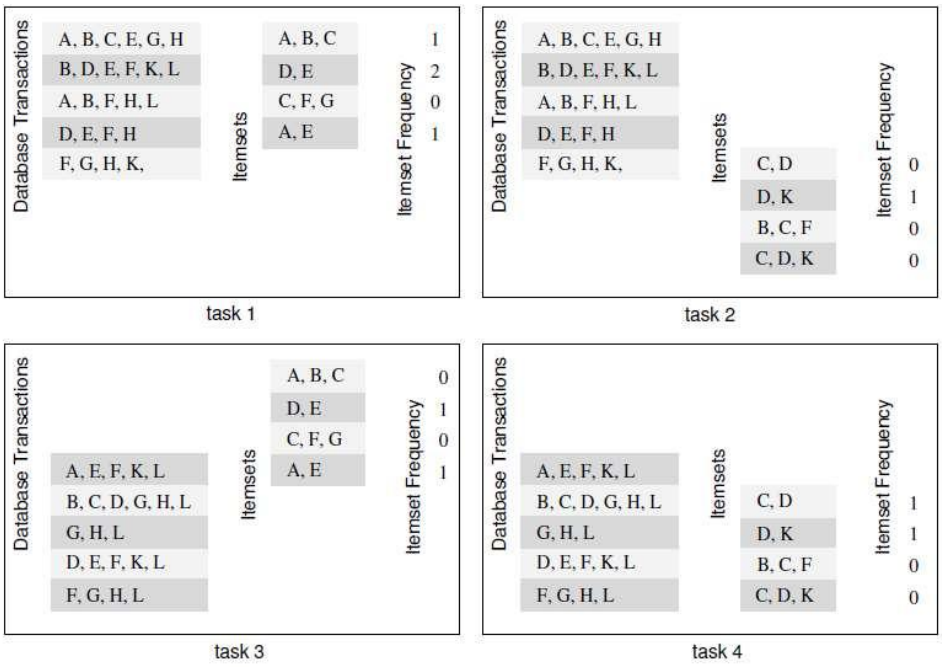
Example



Input and Output Data Partitioning:

Example

Partitioning both transactions and frequencies among the tasks



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

Exploratory decomposition:

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

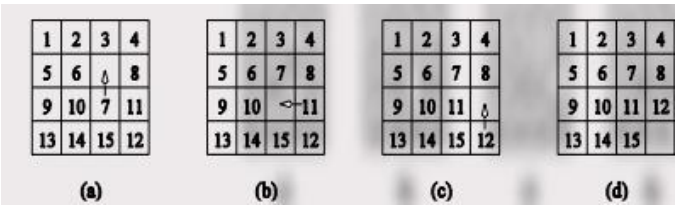
In many cases, the decomposition of the problem goes hand-in-hand with its execution.

These problems typically involve the exploration (search) of a state space of solutions.

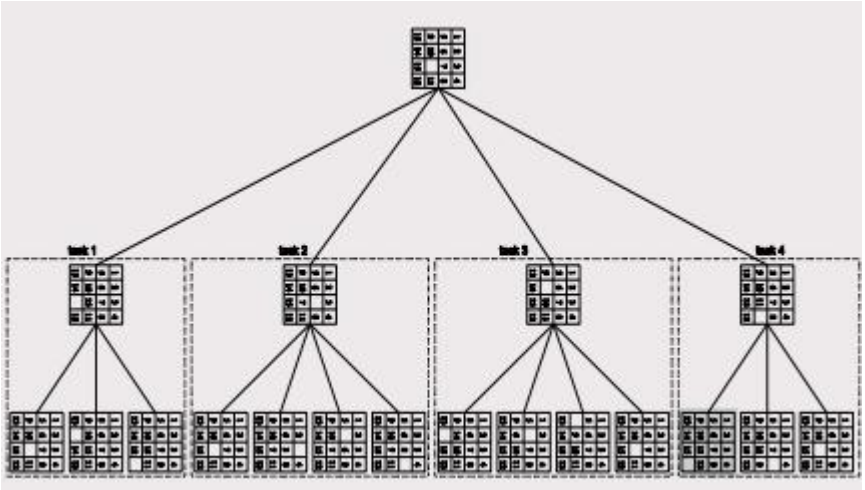
Problems in this class include a variety of discrete optimization problems (0/1 integer programming, QAP, etc.), theorem proving, game playing, etc.

Exploratory Decomposition

Example:



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:

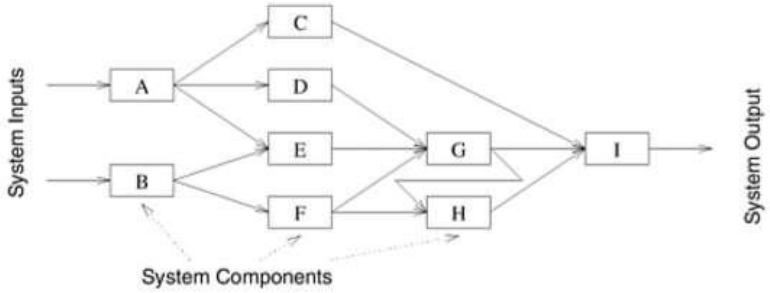
conservative approaches

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.

Lecture NO: 13

Introduction to Parallel I/O

What is paralllel Input/Output ?

"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.

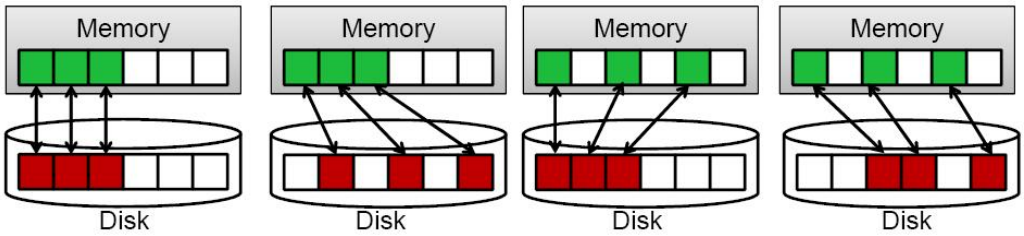
Bottlenecks in performance can occur in many locations (Application and/or File system)

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).

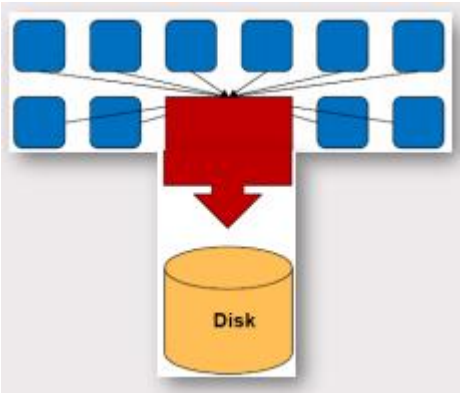
Path from Application to File System

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

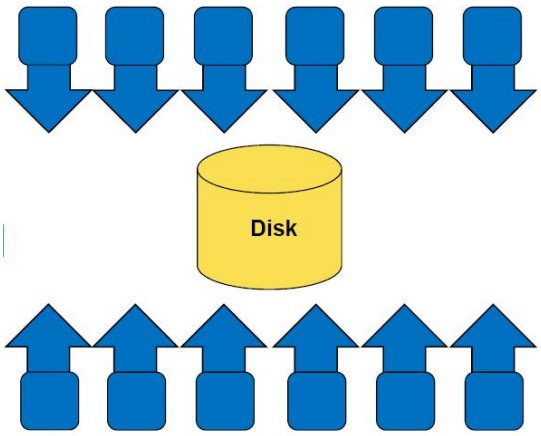
• **All processes perform I/O to individual files.**

o Limited by file system.

• **Pattern does not scale at large process counts.**

o Number of files creates bottleneck with metadata operations.

o Number of simultaneous disk accesses creates contention for file system resources.



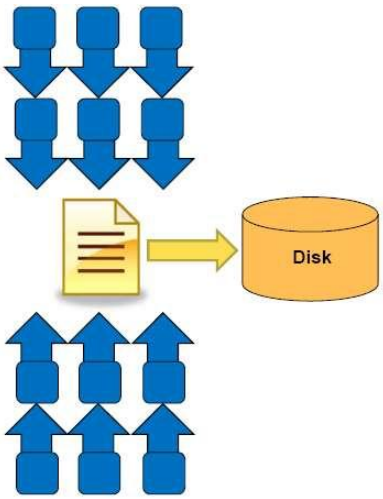
Parallel I/O: Shared File

• 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.

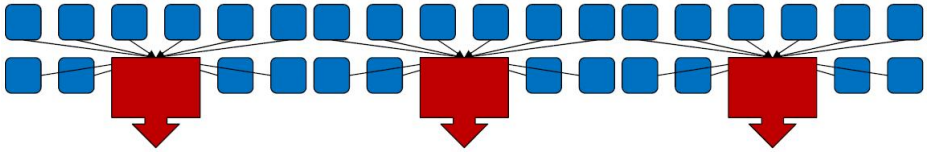
• Group of processes perform parallel I/O to a shared file.

• 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-processo 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

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:

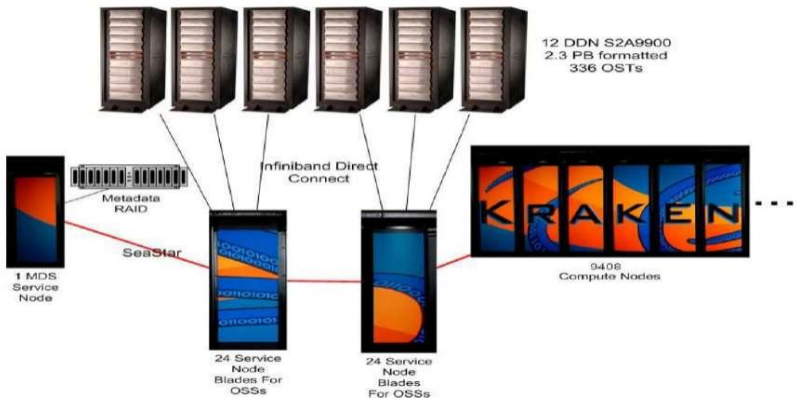
• Present single view

• 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

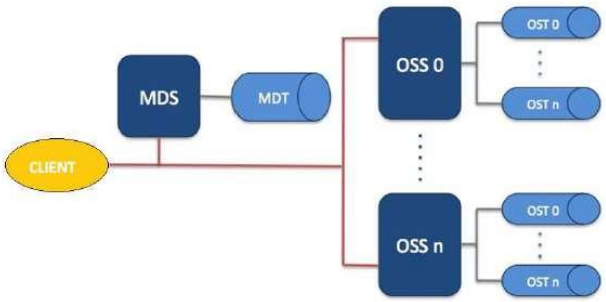


File I/O: Lustre File System

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).

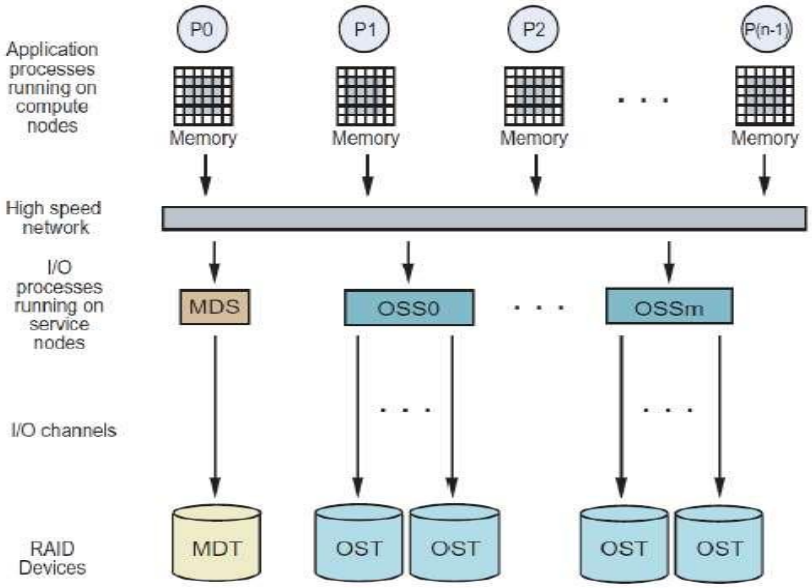


Lustre

• 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.



Striping:

Storing a single file across multiple OSTs

A single file may be stripped across one or more OSTs (chunks ofthe file will exist on more than one OST)

Disadvantage:

- increased overhead due to network operations and server contention

Advantages:

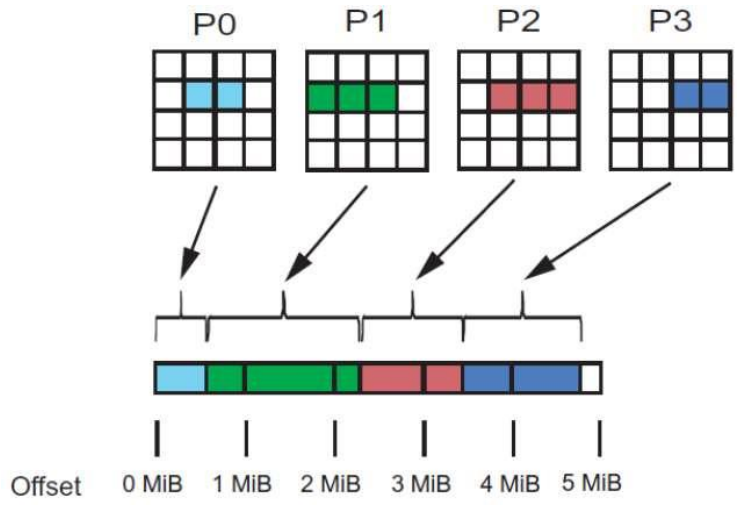
- an increase in the bandwidth available when accessing the file
- an increase in the available disk space for storing the file.

"Lustre file system allows users to specify the striping policy for each file or directory of files using the lfs utility"

File Striping: Physical and Logical Views

• Four application processes write a variable amount of data sequentially within a shared file.

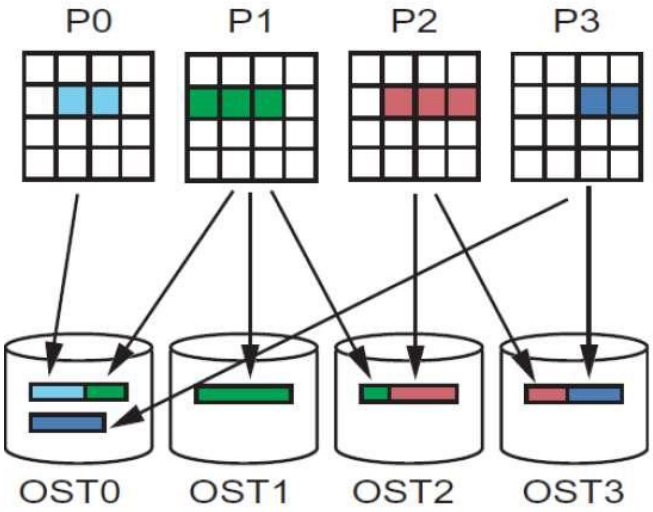
• This shared file is striped over 4 OSTs with 1 MB stripe sizes.



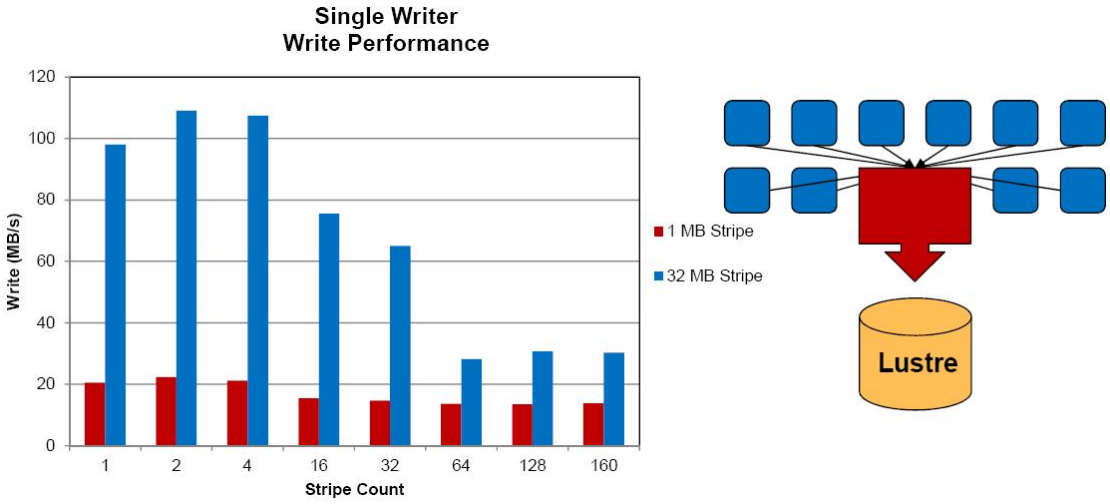
• This write operation is not stripe aligned therefore some processes write their data to stripes used by other processes.

• Some stripes are accessed by more than one process

• May cause contention !



Single write performance and luster:



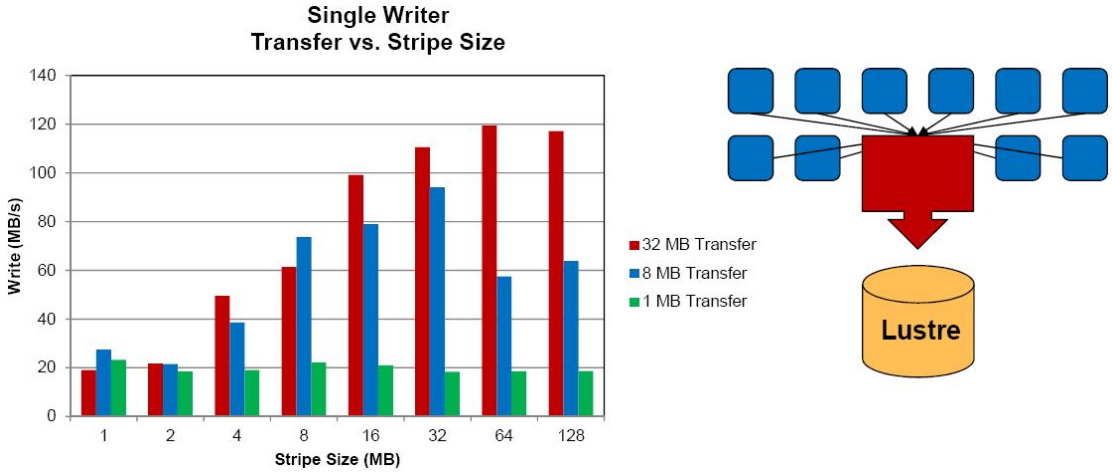
• 32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size

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

• Performance can be limited by the process (transfer size) or file system (stripe size).

• Either can become a limiting factor in write performance.

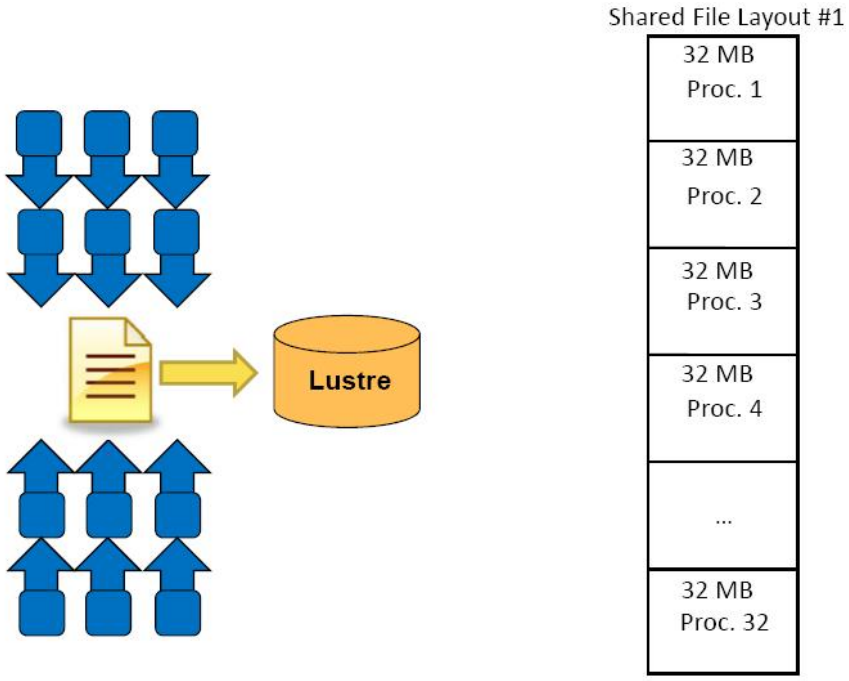
Observation:

• 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.)

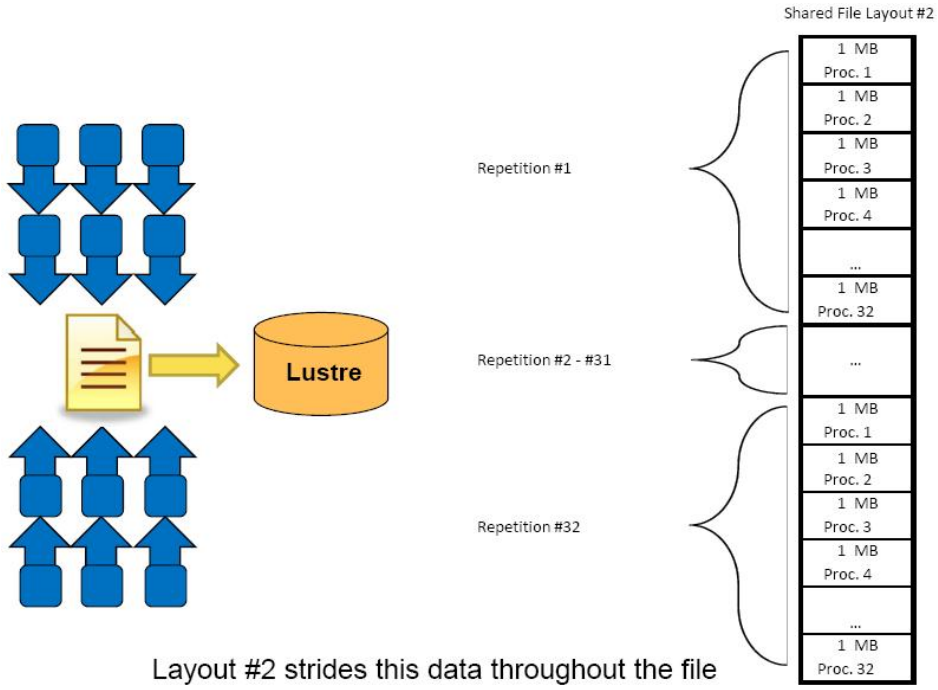
Single Shared Files and Lustre Stripes

Layout #1: Keeps data from a process in a contiguous block

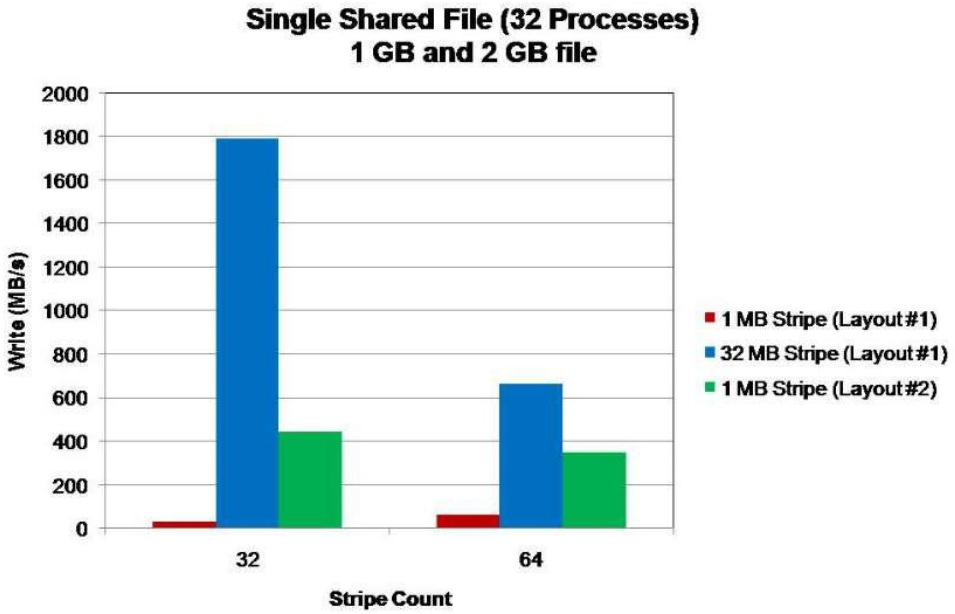


Single Shared Files and Lustre Stripes

Layout #2: strides this data throughout the File



Find layout and luster 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)

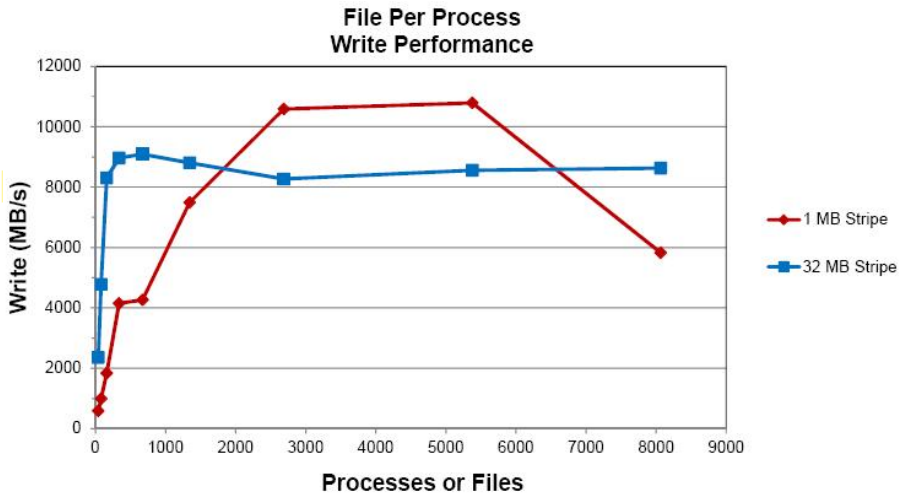
The highest performance is seen from a 32 MB stripe size on Layout #1. Each OST is accessed by only one process. (1788,98 MB/s)

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 HDF5Default Stripe settings: count 4, size 1M, index -1.

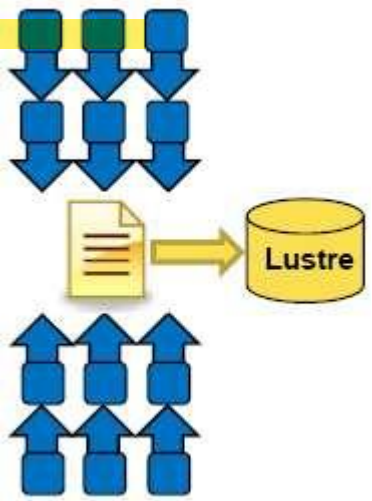
– 1800 s run time (~ 30 minutes)

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

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:

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 data types and thus non-contiguous 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() and MPI_File_read_all()

• This allows the MPI library to do IO optimization

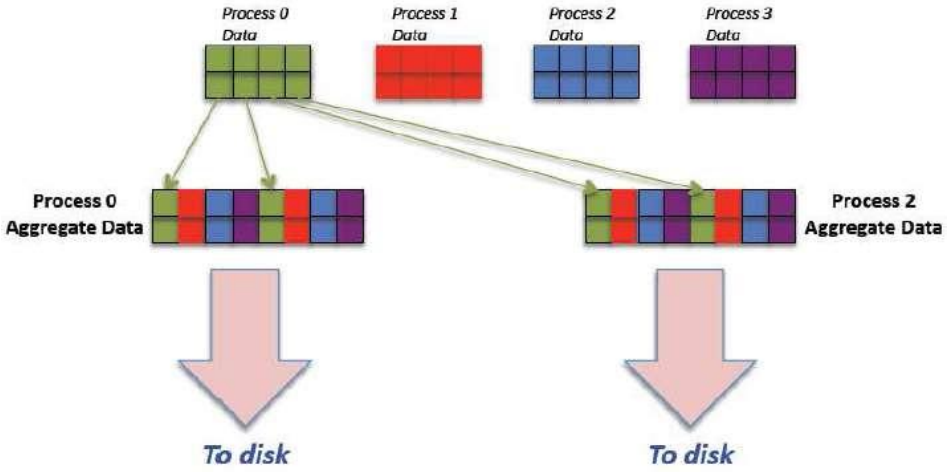
File MPI Collective Writes and Optimizations

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

Lecture No: 14

Performance and Scalability

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 measure 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 sub-optimal serial program to make our parallel program look .

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

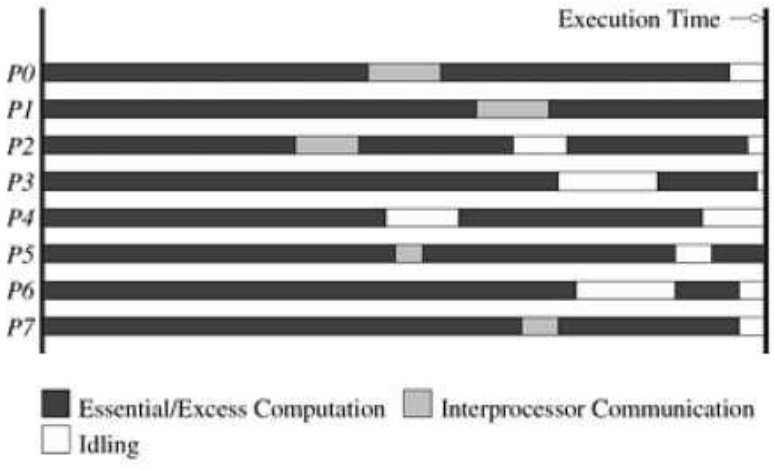
Sources of Overhead in Parallel Programs:

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

• 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.

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 tims e that elapses from the moment the first processor starts to the moment the last processor finishes execution.

We denote the serial runtime by TS 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

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

"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.

Performance Metrics: Example Cont..

• This figure illustrates the procedure for n = 16.

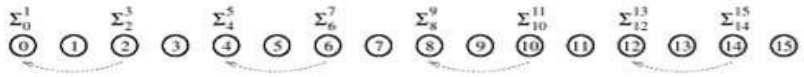
• The processing elements are labeled from 0 to 15.

• Similarly, the 16 numbers to be added are labeled from 0 to 15.

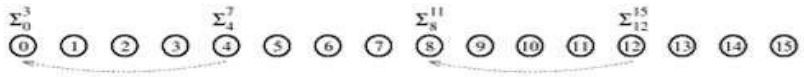
• The sum of the numbers with consecutive labels from i to j is denoted by Σ_j^i.

• Each step shown in Figure consists of one addition and the communication of a single word.

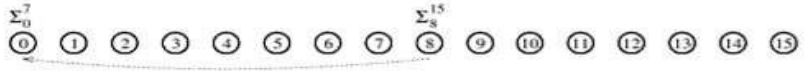
(a) Initial data distribution and the first communication step



(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

Performance metrics: example

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.

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 neverTerminate.

Speedup can never exceed the number of processing elements, p.

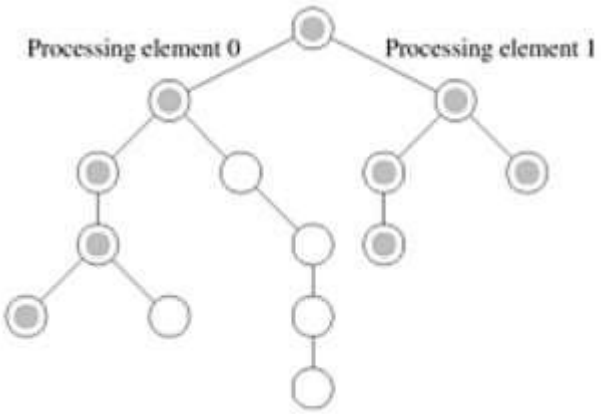
A speedup greater than p is possible only if each processing element spends less than time T_s/p solving the problem.

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= 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

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

• Efficiency is given by

$$\mathbf{E} = \frac{\boldsymbol{\theta}\left(\frac{\mathbf{n}}{\mathbf{\log n}}\right)}{\mathbf{n}}$$

$$= \boldsymbol{\theta}\left(\frac{\mathbf{1}}{\mathbf{\log n}}\right)$$

Cost of a Parallel System

Cost is the product of parallel runtime and the number of processing elements used ($p \times T_p$).

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, $TP = \log n$ (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 TP product of this algorithm is $n (\log n)^2$.

This algorithm is not cost optimal but only by a factor of log n.

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

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 virtualprocessor 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.

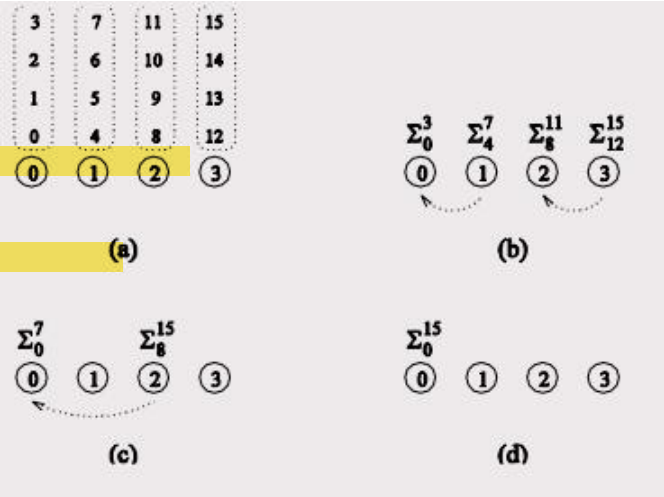
Scalability of Parllel Systems:

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

• Each processing element locally adds its n / p numbers in time $\Theta (n / p)$.

• The p partial sums on p processing elements can be added in time $\Theta(n / p)$.

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



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

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

The total overhead function ***To*** is an increasing function of ***p***

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

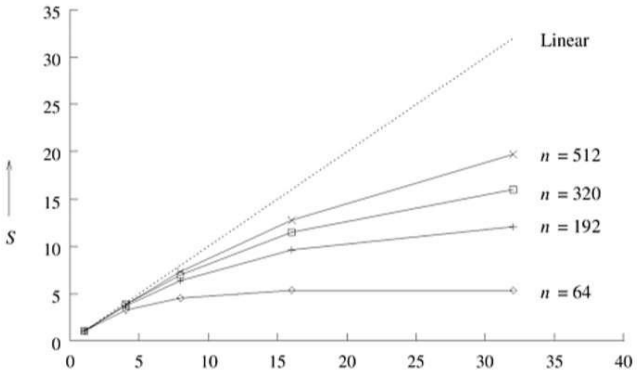
Scaloing Characteristics if Parallel Programs:

Example

• 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 T_s.

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.

Scaling Characteristics of Parallel Programs

Recall that cost-optimal parallel systems have an efficiency of $\Theta(1)$.

Scalability and cost-optimality are therefore related.

A scalable parallel system can always be made costoptimal if the number of processing elements and the size of the computation are chosen appropriately.

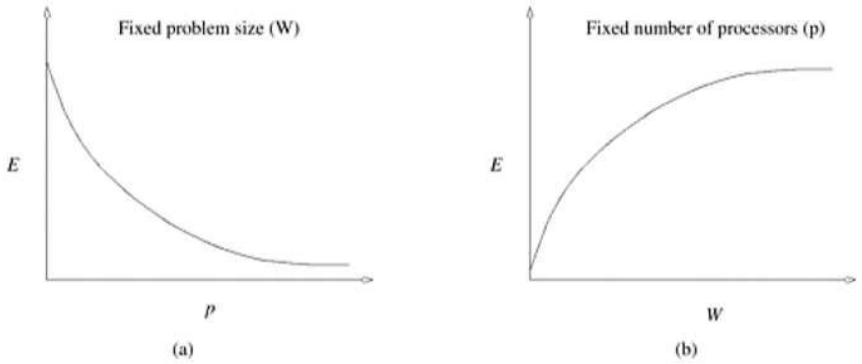
Isoefficiency Metric of Scalability:

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.

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 ***To*** / ***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 ***To*** is a function of ***W*** and ***p***, we have:

$$W = KT_o(W,p).$$

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

Example:

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

Substituting To 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.

Example:

Consider a more complex example where:

$T_o = p^{3/2} + p^{3/4}W^{3/4}$

• Using only the first term of ***To*** in Equation, we get

$\mathrm{W} = \mathrm{K}p^{3/2}$

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

$\mathrm{W} = \mathrm{K}p^{3/4}W^{3/4}$

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

$\mathrm{W} = K^4p^3$

• The larger of these two asymptotic rates determines the isoefficiency. This is given by Θ(***p3***)

Isoefficiency Function and Performance metrics

Cost-Optimality and the Isoefficiency Function

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

$_pT_p = \Theta(\mathrm{W})$

From this we have:

$W + T_o(W) = \Theta(\mathrm{W})$

$T_o(W) = O(\mathrm{W})$

$\mathrm{W} = \Omega(T_p(W, p)$

If we have an isoefficiency function $f(p)$, then it follows that the relation $W = \Omega(f(p))$ 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 $\Theta(p)$ to maintain fixed efficiency; hence, $\Omega(p)$ is the asymptotic lower bound on the isoefficiencyfunction.

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.

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 TPmin

for a given W by differentiating the expression for TPw.r.t. p and equating it to zero.

$$\frac{d}{dp}T_p = 0$$

If $p_o$is the value of p as determined by this equation, T_p (p_o) is the minimum parallel time.

Minimum Execution Time: Example

Consider the minimum execution time for adding n numbers.

$$T_p = \frac{n}{p}\log p = 0$$

Setting the derivative w.r.t. p to zero, we have $\mathrm{p}\frac{\mathrm{n}}{2}$ $\mathrm{Tp}^{\min} = 2\log n$ (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 costoptimal.

MInimum Cost-Optimal Parallel Time:

• Let ***TP*** *cost_opt* be the minimum cost-optimal parallel time.
• If the isoefficiency function of a parallel system is Θ(***f*(*p*)**) , then a problem of size ***W*** can besolved cost-optimally if and only if ***W*= Ω(*f*(*p*))** .
• In other words, for cost optimality, ***p*** = *O*(***f*** --1 (***W***)) .
• For cost-optimal systems, ***TP*** = Θ(***W*/*p***) , therefore,

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

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:

$$\boldsymbol{T_P^{cost\_opt} = \log n + \log\left(\frac{n}{\log n}\right)}$$

$$\boldsymbol{= 2\log n - \log\log n}$$

• **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 $\boldsymbol{pT_p}$ 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	$n\log n$	$n^{1.5}$	$n\log n$

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!

Lecture No: 15

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) Pioneered by Google

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?

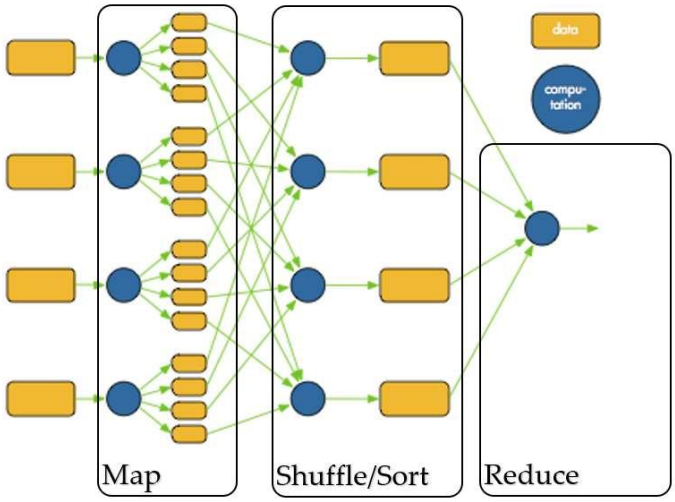
MapReduce has three main phases:

• Map

• Sort

• Reduce

MapReduce Overview



MapReduce: Examples:

MapReduce Example (based on Three Phases)

The canonical MapReduce Example: Word Count

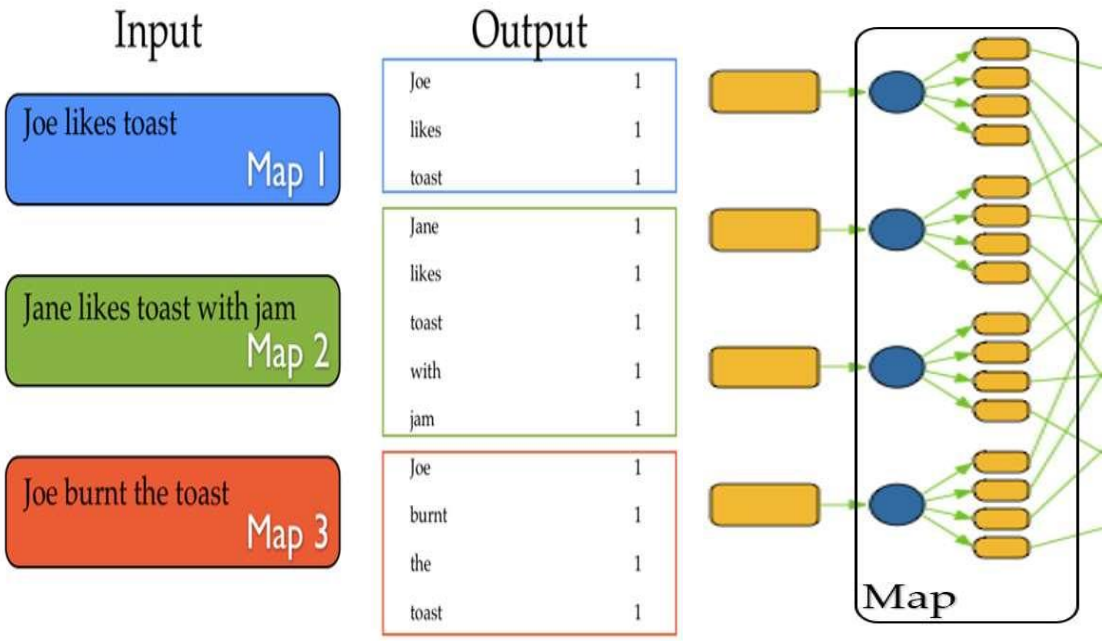
• Example corpus:

Jane likes toast with jam

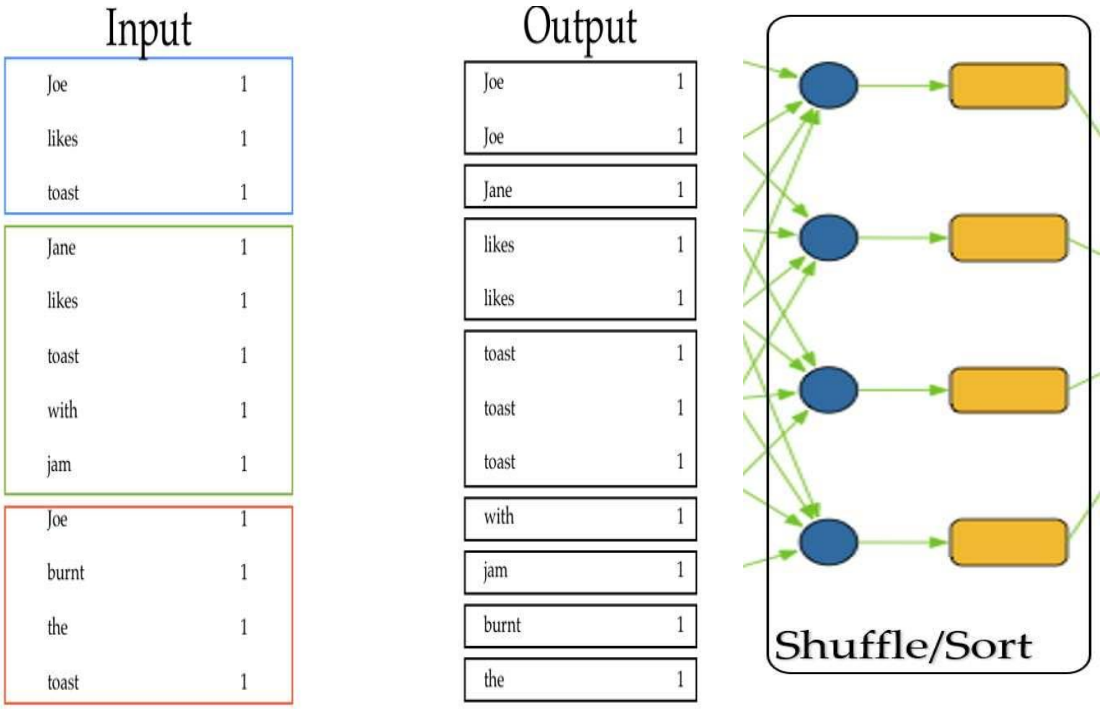
Joe likes toast

Joe burnt the toast

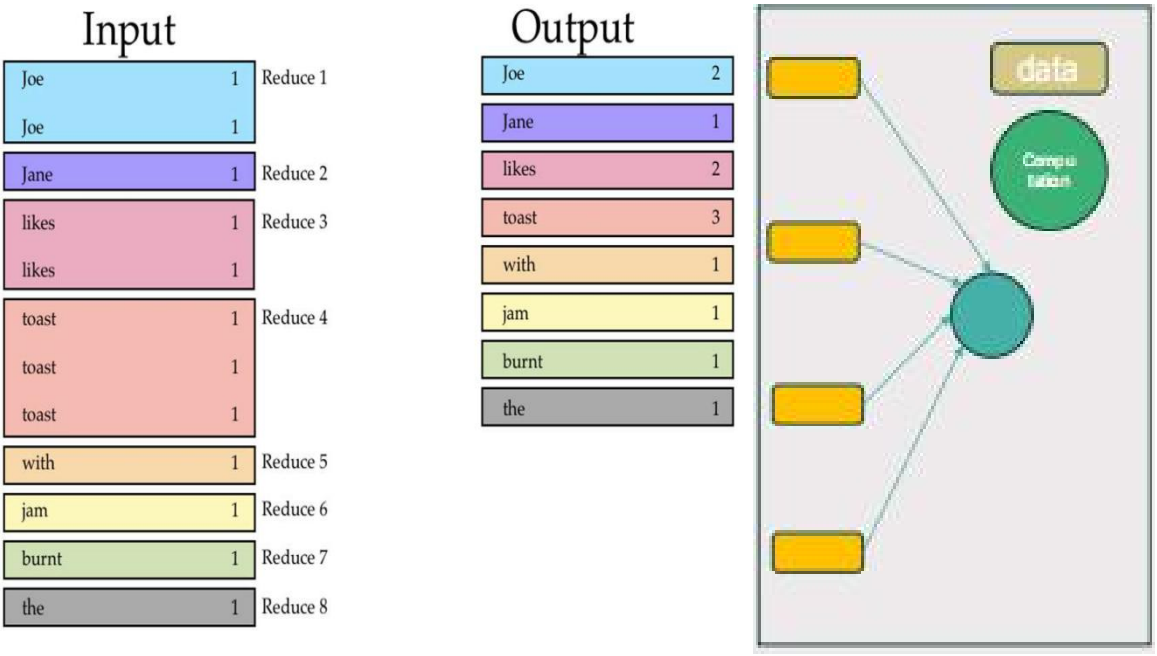
MapReduce: Map (Slow Motion)



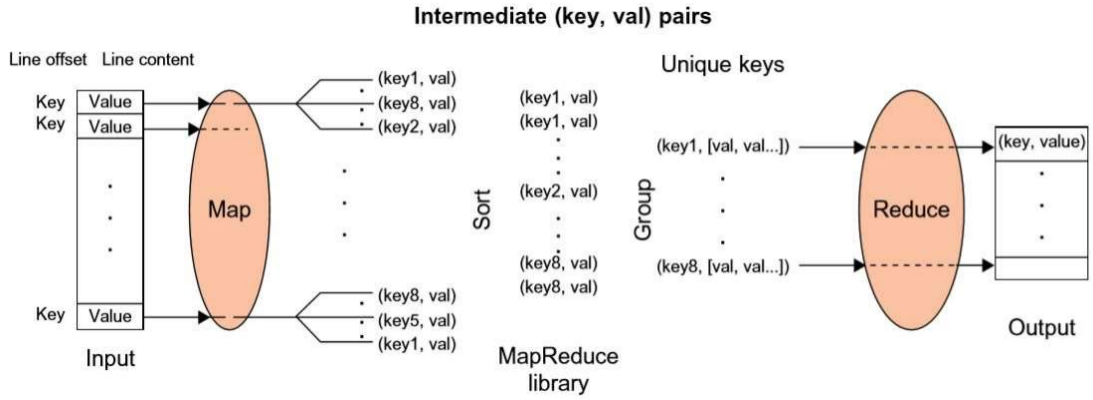
MapReduce: Sort (Slow Motion)



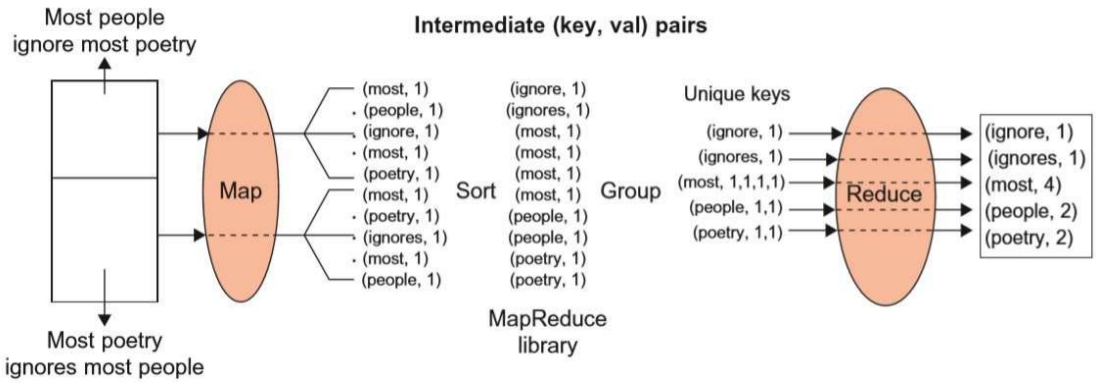
MapReduce: Reduce (Slow Motion)



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



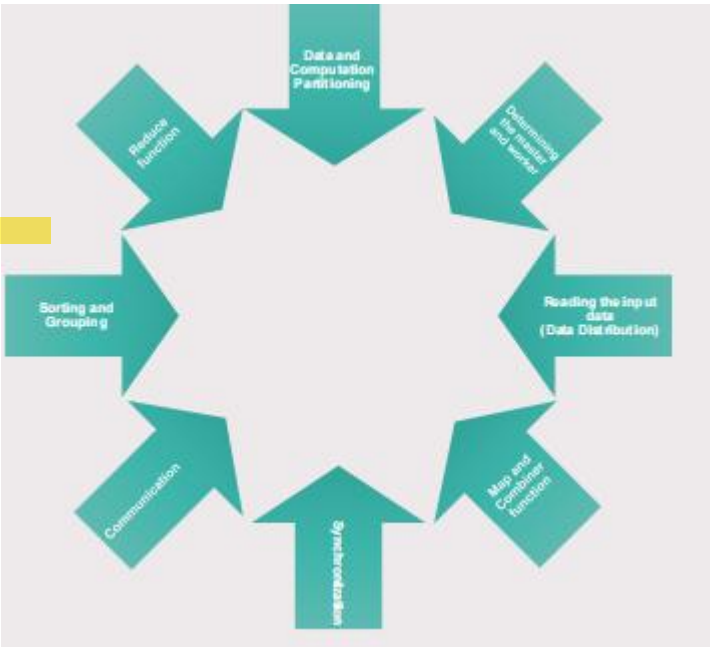
MapReduce logical data in 5 processing stages : Example



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:



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

"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 machine

Cluster can easily scale up with little or no modifications to theprograms

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:

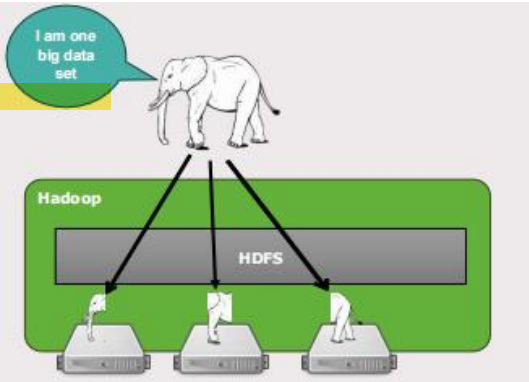
Simple programming model that enables parallel execution of data processing programs

•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:

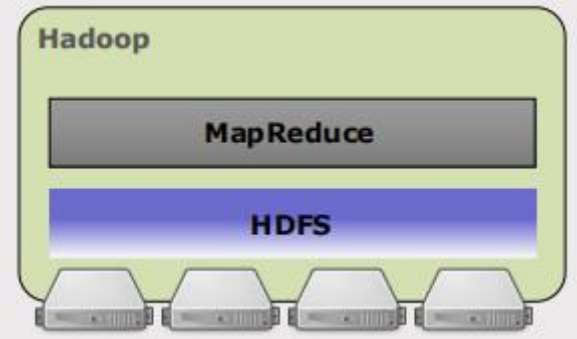
• Hadoop is basically a middleware platforms that manages a cluster of machines
• The core components is a distributed file system (HDFS)
• Files in HDFS are split into blocks that are scattered over the cluster
• The cluster can grow indefinitely simply by adding new nodes



Hadoop Components

MapReduce and Hadoop

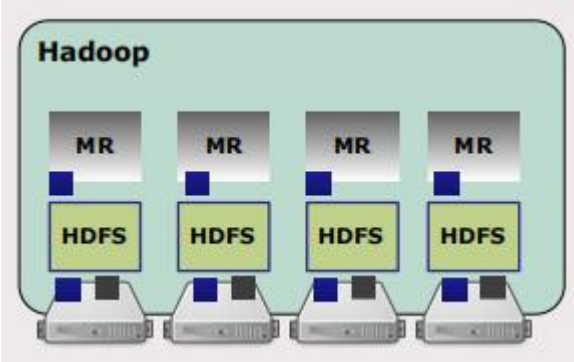
Hadoop



MapReduce

HDFS

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

• "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

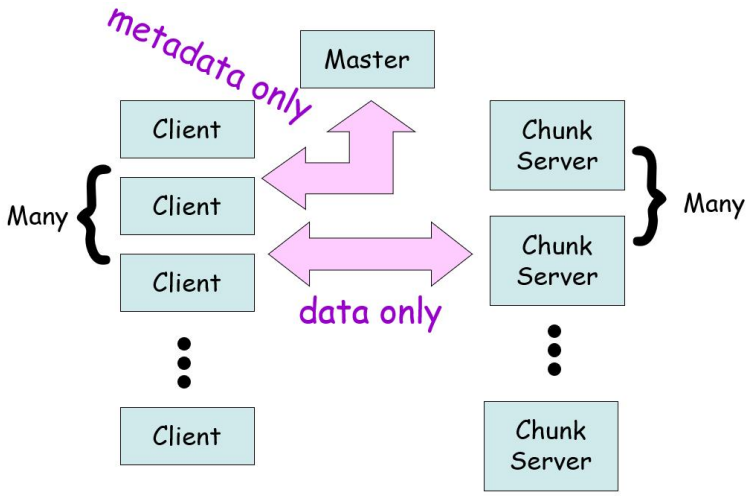
• Snapshot

• Low cost

• Record append

• Atomicity with multiple concurrent writes

GFS: Architecture



GFS: Architecture details

GFS Architecture: Master

• Stores all metadata

• Namespace

• Access-control information

• Chunk locations

• 'Lease' management

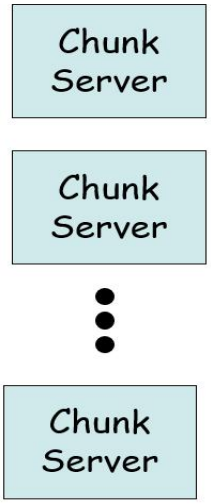
• Heartbeats

• 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

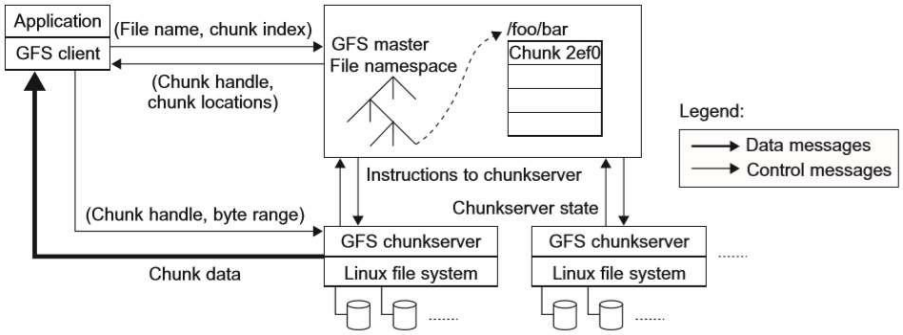
GFS Architecture

• 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

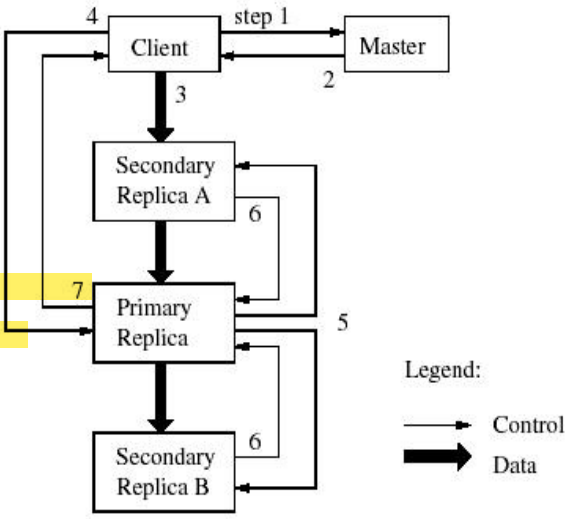
All secondaries follows this order

GFS Write Control & Dataflow

Mutation Order

File 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

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 tobe 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

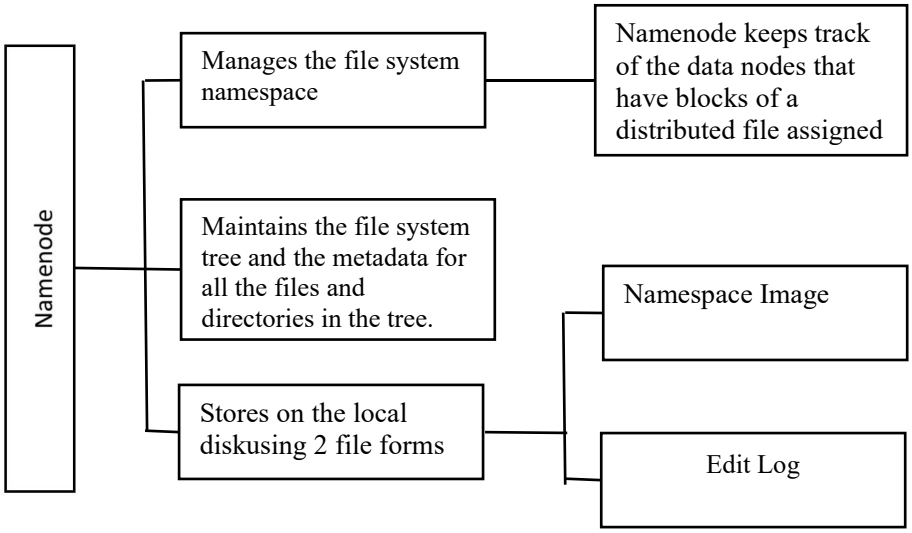
HDFS: Nodes

HDFS clusters use 2 types ofnodes

1. Namenode (master node)

2. Datanode (worker node)

HDFS: Nodes



HDFS: Namenode

Namenode holds the filesystem metadata in its memory

Namenode's memory size determines the limit to the number of filesin a filesystem

But then, what is Metadata?

HDFS: Metadata

Traditional concept ofthe library card catalogs

Categorizes and describes thecontents and context of thedata files

Maximizes the usefulness of the original data file by makingit easy to find and use

HDFS: Metadata

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 file system

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 file system (on behalf of the user) by communicating with the namenode and datanodes

Client can use a filesystem interface similar to a POSIX (PortableOperating System Interface)) so the user code does not need to know about the namenode and datanodesto function properly

HDFS: Namenode Failure

Namenode keeps track of the datanodes that have blocks of a distributed fileassigned

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 SeriesHDFS Reliability Enhancements

- HDFS Federation
- HDFS HA (High-Availability)