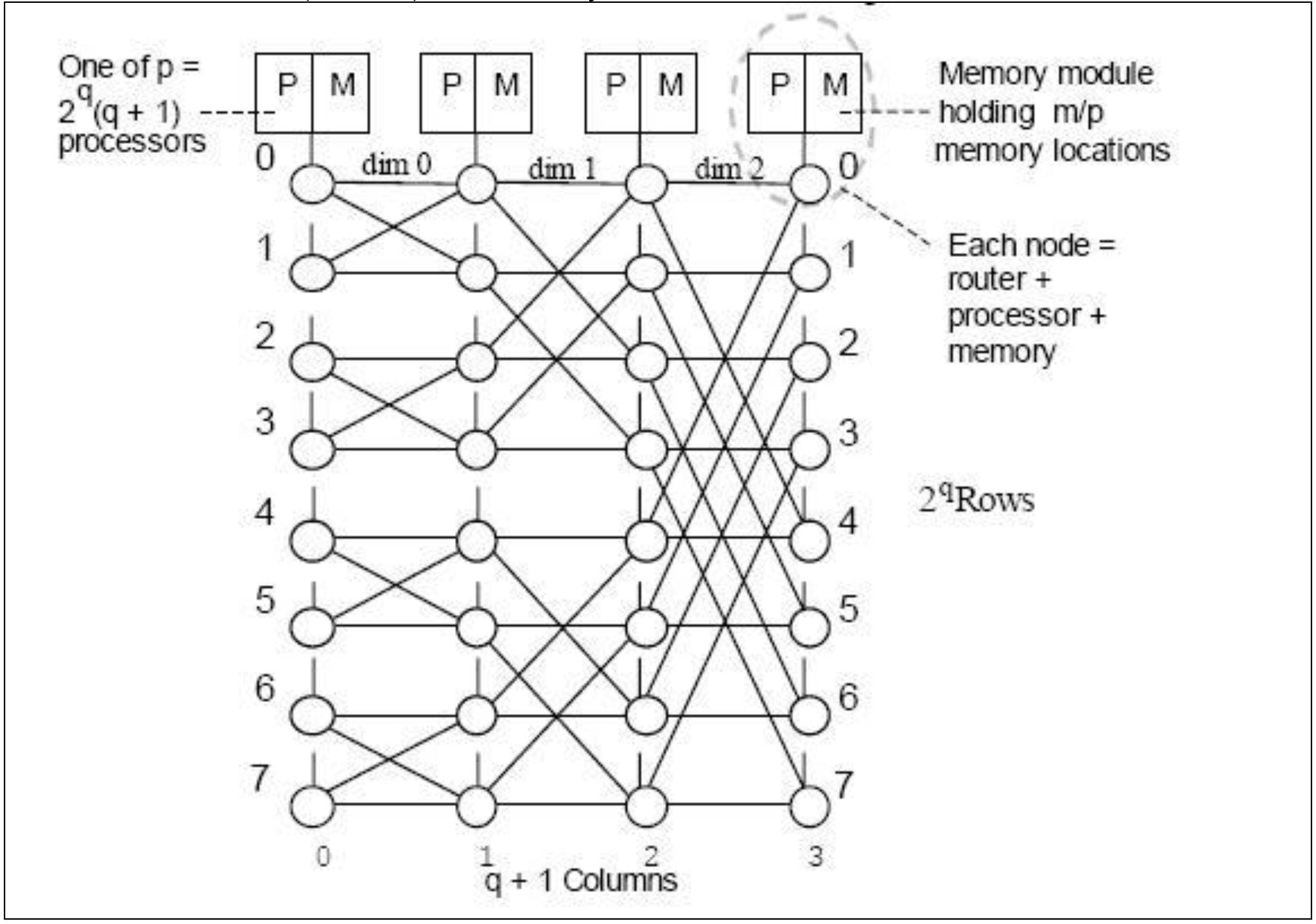
UNIT-4

MULTIPROCESSOR ARCHITECTURE AND PROGRAMMING

EMULATIONS AMONG ARCHITECTURES

Emulate means to copy.
Emulations are useful to quickly develop algorithms for a new architecture without expending the significant resources that would be required for native algorithm development.
They allow us to develop algorithms for architectures that are easier to program (e.g., shared-memory or PRAM) and then they can be run on machines that are realizable, easily accessible, or affordable.

Distributed Shared Memory:
The butterfly network can indeed emulate the PRAM efficiently. Such an emulation, by butterfly or other networks, provides the illusion of shared memory to the users or programmers of a parallel system that in fact uses message passing for interprocessor communication. This illusion, or programming view, is sometimes referred to as distributed (or virtual) shared memory.


One of p = 2^q(q + 1) processors
P
M
Memory module holding m/p memory locations
dim 0
dim 1
dim 2
Each node = router + processor + memory
0
1
2
3
4
5
6
7
2^q Rows
0
1
2
3
q + 1 Columns

Task Scheduling:
Parameters associated with each task that are used during task scheduling are as follows:

- **Execution or running time**. The worst case, average case, or the probability distribution of a task's execution time.
- **Creation**. A fixed set of tasks, known at compile time, or a probability distribution for the task creation times.
- **Relationship with other tasks**. This type of information may include criticality, priority order, and/or data dependencies.
- **Start or end time**. A **task's release time** is the time before which the task should not be executed. Also, a hard or soft deadline may be associated with each task. A **hard deadline** is specified when the results of a task become practically worthless if not obtained by a certain time. A **soft deadline** may penalize late results but does not render them totally worthless.

The scheduling algorithms are characterized as:

- **Preemption**: With **nonpreemptive scheduling**, a task must run to completion once started, whereas with **preemptive scheduling**, execution of a task may be suspended to accommodate a more critical or higher-priority task. In practice, preemption involves some overhead for storing the state of the partially completed task and for continually checking the task queue for the arrival of higher-priority tasks.
- **Granularity**: **Fine-grain and Coarse-grain scheduling** problems deal with tasks of various complexities, from simple multiply–add calculations to large, complex program segments, perhaps consisting of thousands of instructions. **Finegrain scheduling** involves dividing the main task into large number of small tasks and then scheduling them. **Coarse-grain scheduling** involves dividing the main task into small number of medium size tasks and then scheduling them.

List Scheduling:
In list scheduling, a priority level is assigned to each task. A task list is then constructed in which the tasks appear in priority order, with some tasks tagged as being ready for execution (initially, only tasks having no prerequisite are tagged). With each processor that becomes available for executing tasks, the highest-priority tagged task is removed from the list and assigned to the processor.

DATA STORAGE

Data Access problems:

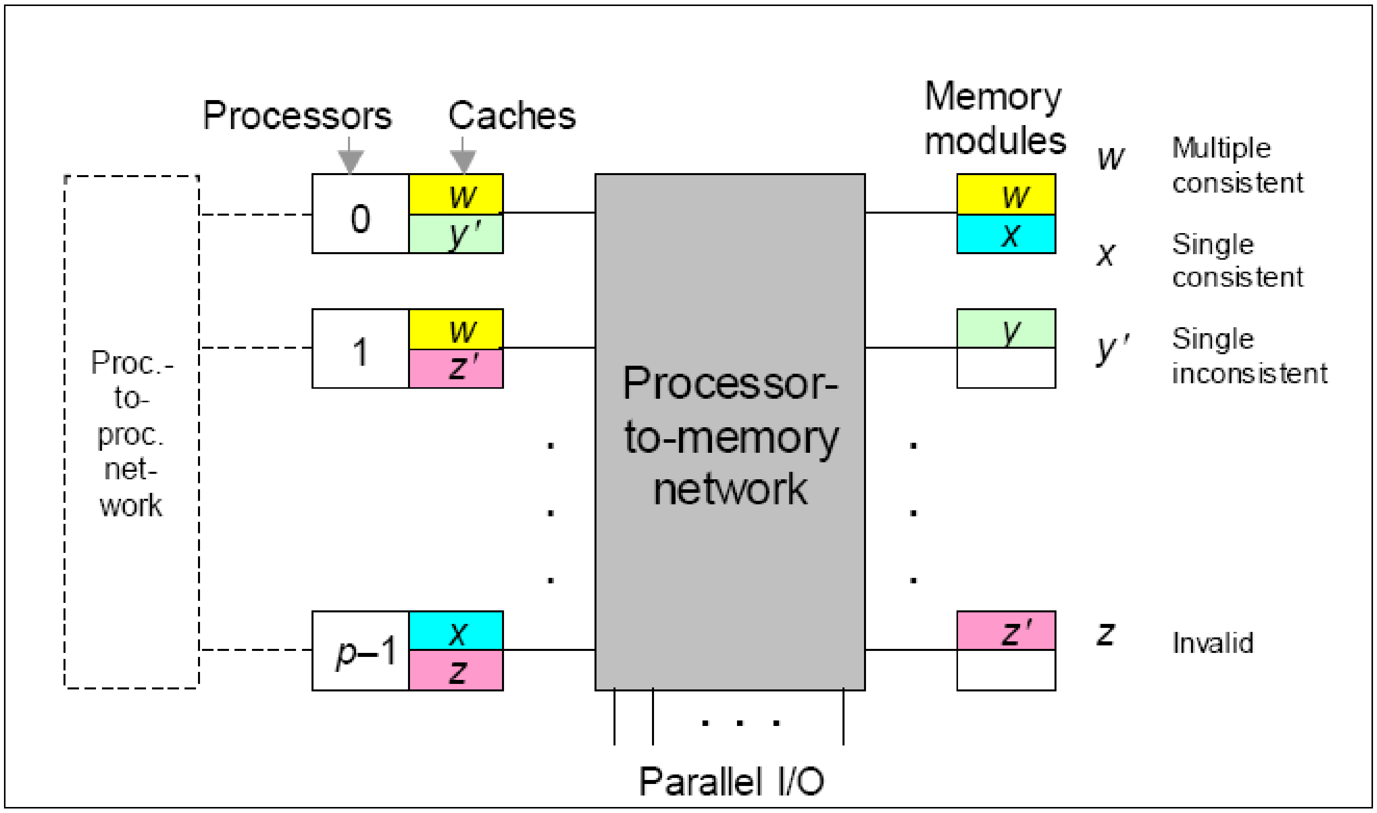
- Memory access latency is a major performance hindrance in parallel systems. Even in sequential processors, the gap between processor and memory speeds has created difficult design issues necessitating pipelined memory access and multiple levels of cache memories to bridge the gap.
- **In vector processors**, the mismatch between processor and memory speeds is made tolerable through the provision of vector registers (that can be loaded/stored as computations continue with data in other registers) and pipeline chaining, allowing intermediate results to be forwarded between function units without first being stored in memory or even in registers.
- **In a Parallel shared-memory system**, the memory access mechanism is much more complicated, and thus slower, than in uniprocessors. In distributed-memory machines, severe speed penalties are associated with access to nonlocal data. In both cases, three complementary approaches are available to mediate the problem:
  - **Distributing the data** so that each item is located where it is needed most. This involves an initial assignment and periodic redistribution as conditions change.
  - Automatically bringing the most useful, and thus most frequently accessed, data into local memory whenever possible. This is known as **data caching**.
  - Making the processors' computational throughput relatively insensitive to the memory access latency. This is referred to as latency tolerance or **latency hiding**.

Caching:

- **A cache** is any fast memory that is used to store useful or frequently used data.
- **A processor cache**, e.g., is a very fast (usually static RAM) memory unit holding the instructions and data that were recently accessed by the processor or that are likely to be needed in the near future.
- **A Level-2 cache** is somewhat slower than a processor cache, but still faster than the high-capacity dynamic RAM main memory.
- **A disk cache or file cache** is usually a portion of main memory that is set aside to hold blocks of data from the secondary memory. This allows multiple records or pages to be read and/or updated with a single disk access, which is extremely low compared with processor or even main memory speeds. To access a required data word, the cache is consulted first.
- Finding the required data in the cache is referred to as a **cache hit**; not finding it is a **cache miss**. An important parameter in evaluating the effectiveness of cache memories of any type is the **hit rate**, defined as the fraction of data accesses that can be satisfied from the cache as opposed to the slower memory that sits beyond it.
- In typical microprocessors, accessing the cache memory is part of the instruction execution cycle. As long as the required data are in the cache, instruction execution continues at full speed. When a cache miss occurs and the slower memory must be accessed, instruction execution is interrupted.

Cache Coherence Protocols:

Placement of shared data into the cache memories of multiple processors creates the possibility of the multiple copies becoming inconsistent.
If caching of shared modifiable data is to be allowed, hardware provisions for enforcing cache coherence are needed.



The above image shows four data blocks w, x, y, and z in the shared memory of a parallel processor along with some copies in the processor caches. The primed values y' and z' represent modified or updated versions of blocks y and z, respectively. The state of a data block being cached can be one of the following:

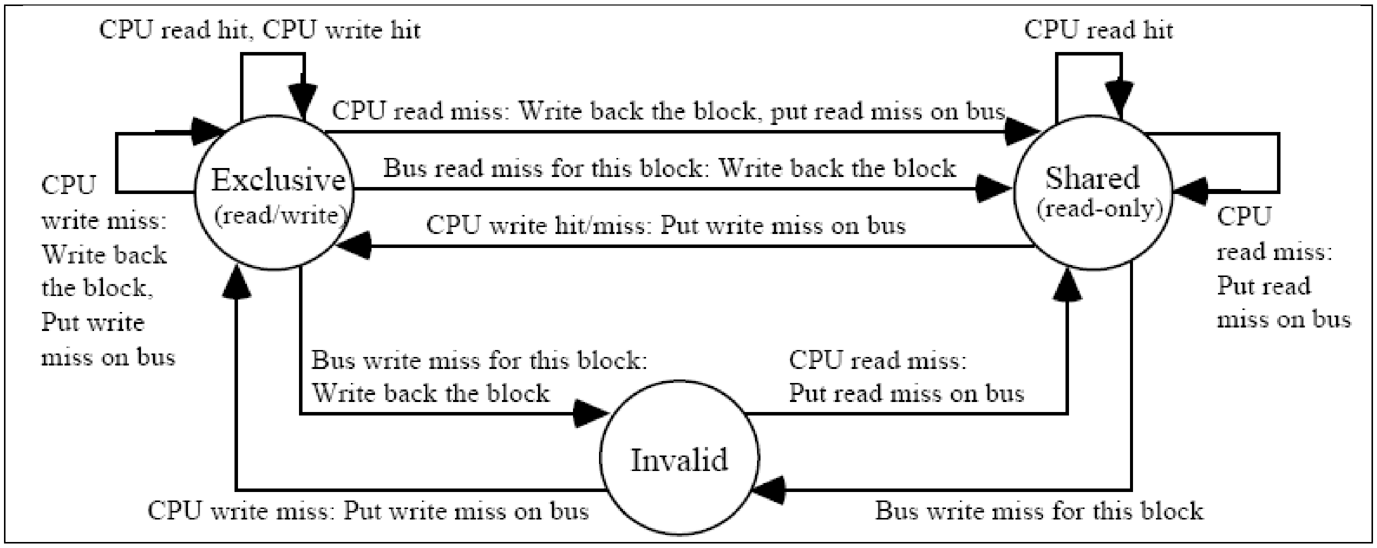
- **Multiple-consistent.** Several caches contain copies of w; the cache copies are consistent with each other and with the copy of w in the main memory.
- **Single-consistent.** Only one cache contains a copy of x and that copy is consistent with the copy in the main memory.
- **Single-inconsistent.** Only one cache contains a copy of y which has been modified and is thus different from the copy in main memory.
- **Invalid.** A cache contains an invalid copy of z; the reason for this invalidity is that z has been modified in a different cache which is now holding a single consistent copy.

Clearly, we want to avoid having multiple inconsistent cache copies. For this reason, when there are multiple cache copies and one of the copies is updated, we must either invalidate all other cache copies (write-invalidate policy) or else modify all of these copies as well (write-update policy).

Snoopy cache coherence protocols are so named because they require that all caches "snoop" on the activities of other caches, typically by monitoring the transactions on a bus, to determine if the data that they hold will be affected. Several implementations of snoopy protocols can be found in bus-based shared-memory multiprocessors.

A Bus-Based Snoopy Cache Coherence Protocol:

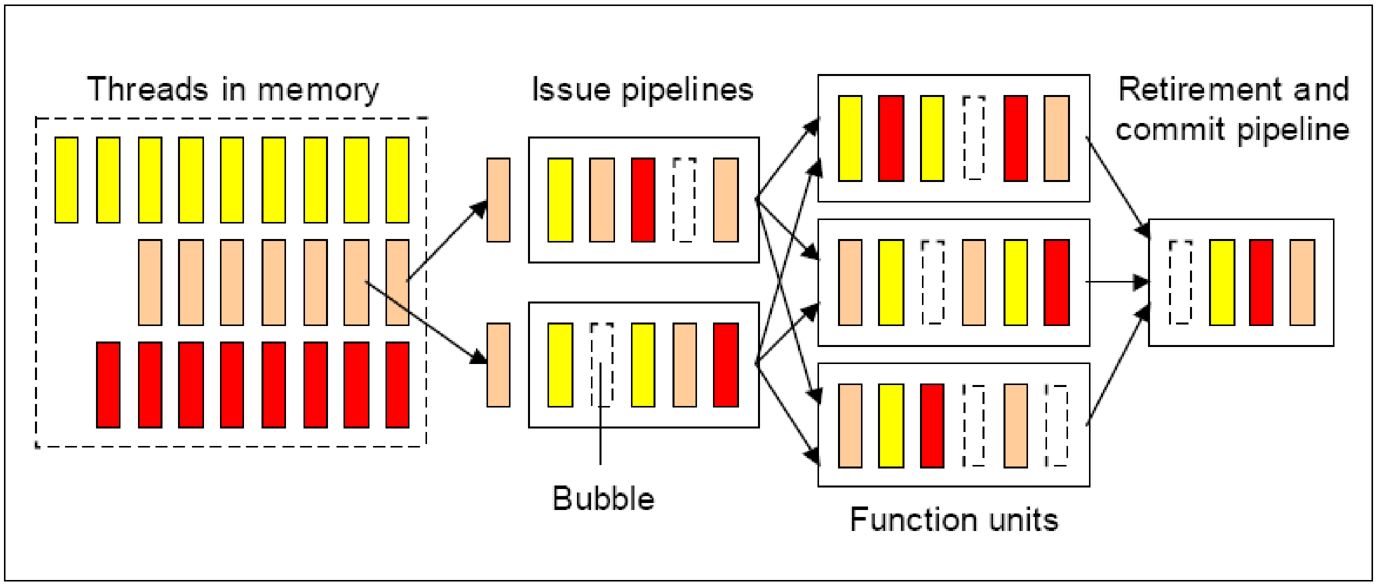
1. **CPU read hit.** A shared block remains shared; an exclusive block remains exclusive.
2. **CPU read miss**. A block is selected for replacement; if the block to be overwritten is invalid or shared, then "read miss" is put on the bus; if it is exclusive, it will have to be written back into main memory. In all cases, the new state of the block is "shared."
3. **CPU write hit.** An exclusive block remains exclusive; a shared block becomes exclusive and a "write miss" is put on the bus so that other copies of the block are invalidated.
4. **CPU write miss.** A block is selected for replacement; if the block to be overwritten is invalid or shared, then "write miss" is put on the bus; if it is exclusive, it will have to be written back into main memory. In all cases, the new state of the block is "exclusive."
5. **Bus read miss**. This is only relevant if the block is exclusive, in which case the block must be written back to main memory. The two copies of the block will then beconsistent with each other and with the copy in main memory; hence the new state will be "shared."
6. **Bus write miss**. If the block is exclusive, it must be written back into main memory; in any case, the new state of the block becomes "invalid."


CPU read hit, CPU write hit
CPU read hit
CPU read miss: Write back the block, put read miss on bus
Exclusive (read/write)
Bus read miss for this block: Write back the block
Shared (read-only)
CPU write hit/miss: Put write miss on bus
CPU write miss: Write back the block, Put write miss on bus
CPU read miss: Put read miss on bus
Bus write miss for this block: Write back the block
CPU read miss: Put read miss on bus
Invalid
CPU write miss: Put write miss on bus
Bus write miss for this block

MULTITHREADING AND LATENCY HIDING:

Latency Hiding:
The general idea of latency hiding methods is to provide each processor with some useful work to do as it waits for remote memory access requests to be satisfied. In the ideal extreme, latency hiding allows communications to be completely overlapped with computation, leading to high efficiency and hardware utilization.

Multithreading:
A multithreaded computation typically starts with a sequential thread, followed by some supervisory overhead to set up (schedule) various independent threads, followed by computation and communication (remote accesses) for individual threads, and concluded by a synchronization step to terminate the threads prior to starting the next unit of parallel work.
Multithreading leads to latency hiding.



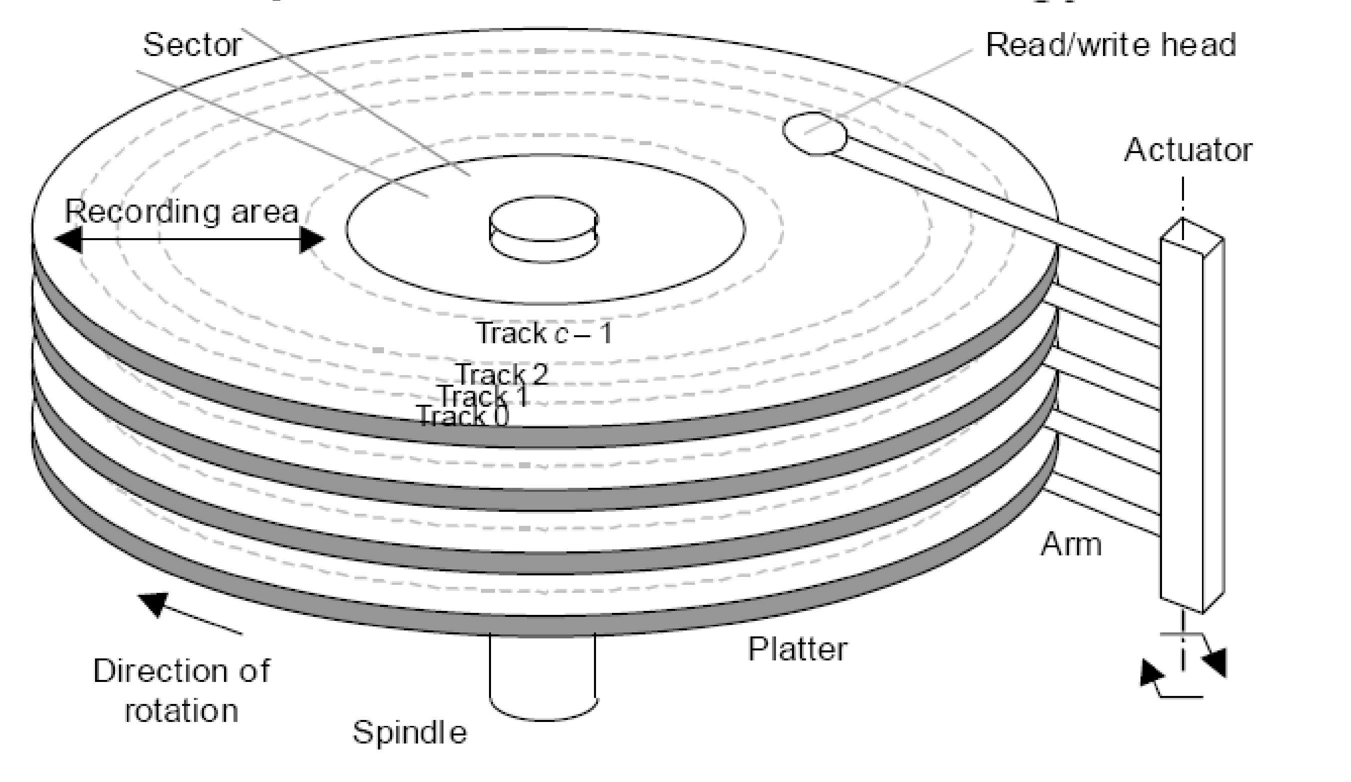
Parallel I/O Technology:

An important requirement for highly parallel systems is the provision of high-bandwidth I/O capability. For some data-intensive applications, the high processing power of a massively parallel system is not of much value unless the I/O subsystem can keep up with the processors.

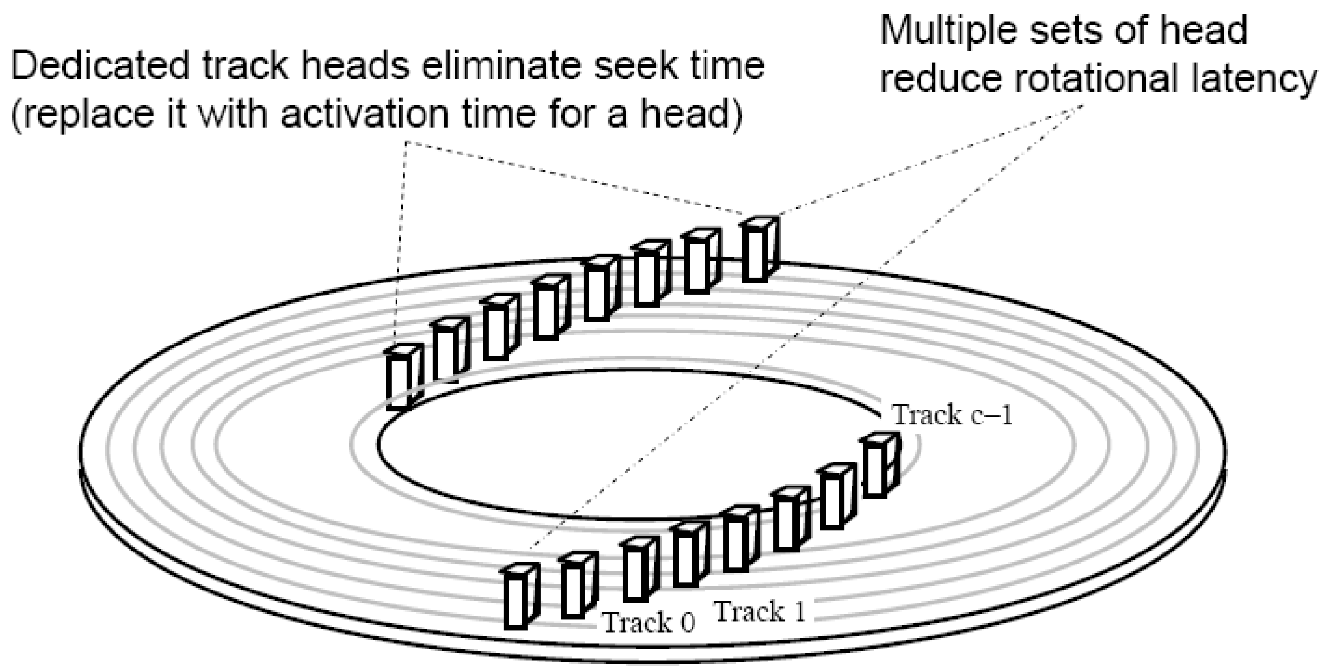
Magnetic Disks:
Multiple-platter high-capacity magnetic disk are also called Hard-disks or Hard-drives.
Each platter has two recording surfaces and a read/write head mounted on an arm. The access arms are attached to an actuator that can move the heads radially in order to align them with a desired cylinder (i.e., a set of tracks, one per recording surface).
A sector or disk block is part of a track that forms the unit of data transfer to/from the disk.
Access to a block of data on disk typically consists of three phases:

1. **Cylinder seek**: moving the heads to the desired cylinder (seek time)
2. **Sector alignment**: waiting until the desired sector is under the head (rotational latency)
3. **Data transfer**: reading the bytes out as they pass under the head (transfer time)

Hard-Disk with multiple read-write heads for fast data access:



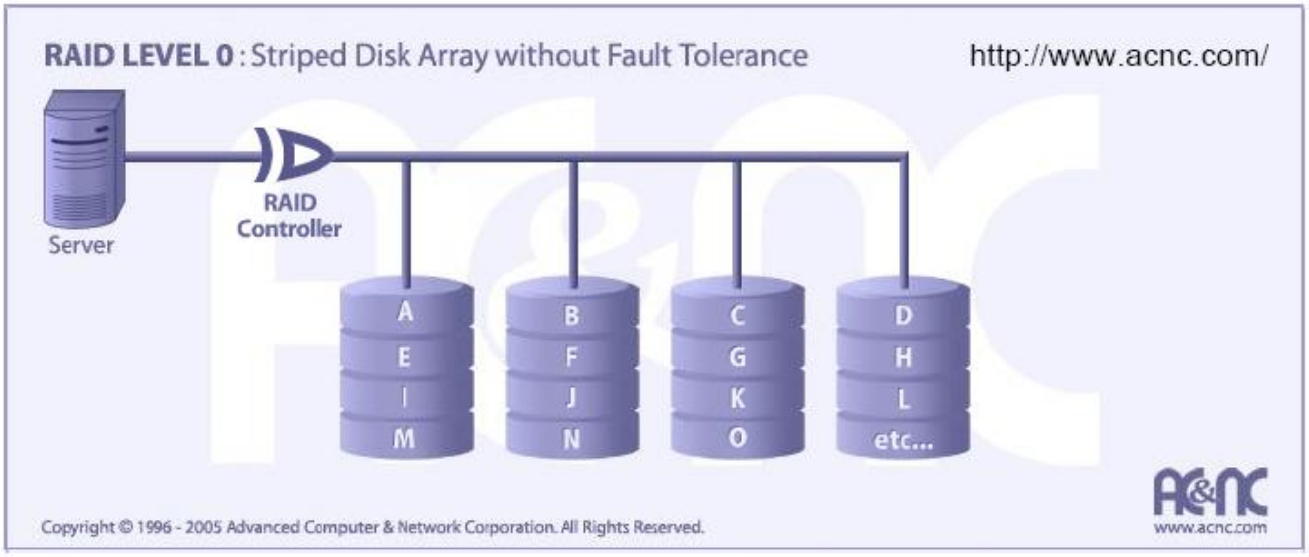
One of the earliest attempts at achieving high-throughput parallel I/O was the use of head-per-track disks , with their multiple read/write heads capable of being activated at the same time. They proved to be more expensive.


Dedicated track heads eliminate seek time (replace it with activation time for a head)
Multiple sets of head reduce rotational latency
Track c–1
Track 0
Track 1

Redundant Array of Inexpensive (& Independent) Disks (RAID):
Conceptually, both performance and reliability can be improved through redundancy: Storing multiple copies of a file, e.g., allows us to read the copy that is most readily accessible and to tolerate the loss of one or more copies related to disk crashes and other hardware problems. For these reasons, disk array technology began with the notion of redundancy built in. The acronym “RAID,” for redundant array of inexpensive disks, reflects this view.

RAID-0:
The reference point for RAID architectures is Level 0, or RAID0 for short, which involves the use of multiple disks for I/O performance improvement without any redundancy for fault tolerance. For example, if a file is declustered or striped across disks 0 and 1, then the file data can be accessed at twice the speed, once they have been located on the two disks. The performance gain is significant only for large files, as otherwise the disk access delay reduces the readout or write time.

RAID Level 0


RAID LEVEL 0 : Striped Disk Array without Fault Tolerance
http://www.acnc.com/
Server
RAID Controller
A
E
I
M
B
F
J
N
C
G
K
O
D
H
L
etc...
Copyright © 1996 - 2005 Advanced Computer & Network Corporation. All Rights Reserved.
www.acnc.com

Structure:

Striped (data broken into blocks & written to separate disks)

Advantages:

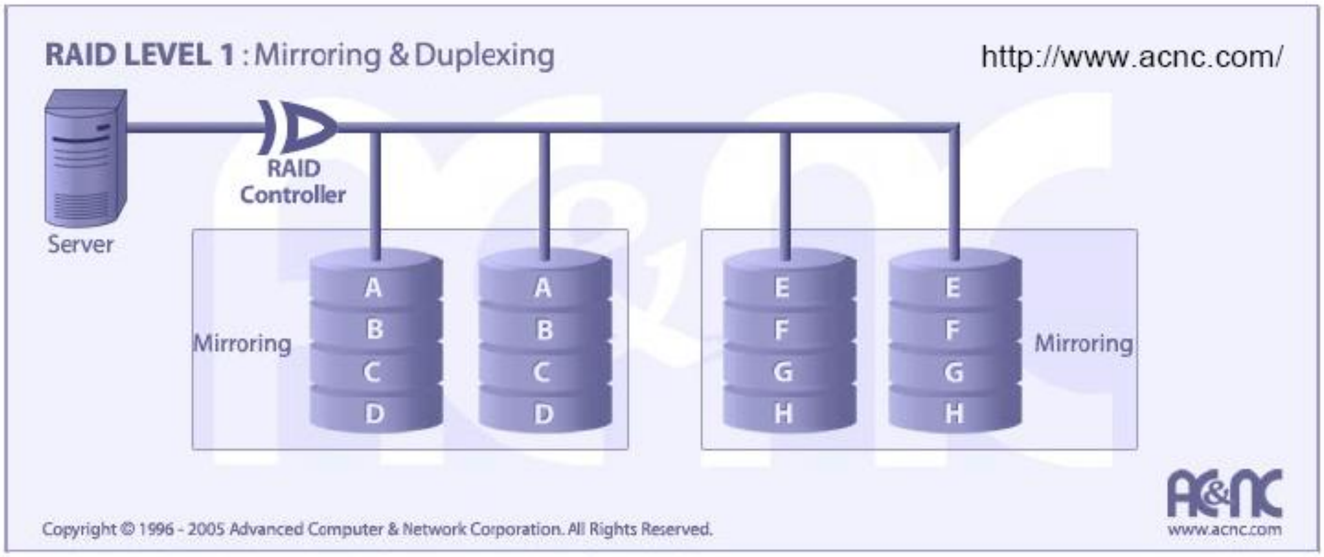
Spreads I/O load across many channels and drives

Drawbacks:

No fault tolerance (data lost with single disk failure)

RAID-1:
RAID1 takes the above concept and introduces fault tolerance via mirrored disks. For each data disk, there is a backup that holds exactly the same data.
However, there is a performance overhead to be paid whether or not striping is used. Each update operation on a file must also update the backup copy. If updating of the backup is done right away, then the larger of the two disk head movements will dictate the delay. If backup updates are queued and performed when convenient, then the probability of data loss from a disk crash increases.

RAID Level 1


RAID LEVEL 1 : Mirroring & Duplexing
http://www.acnc.com/
Server
RAID Controller
Mirroring
A
B
C
D
A
B
C
D
E
F
G
H
E
F
G
H
Mirroring
Copyright © 1996 - 2005 Advanced Computer & Network Corporation. All Rights Reserved.
www.acnc.com

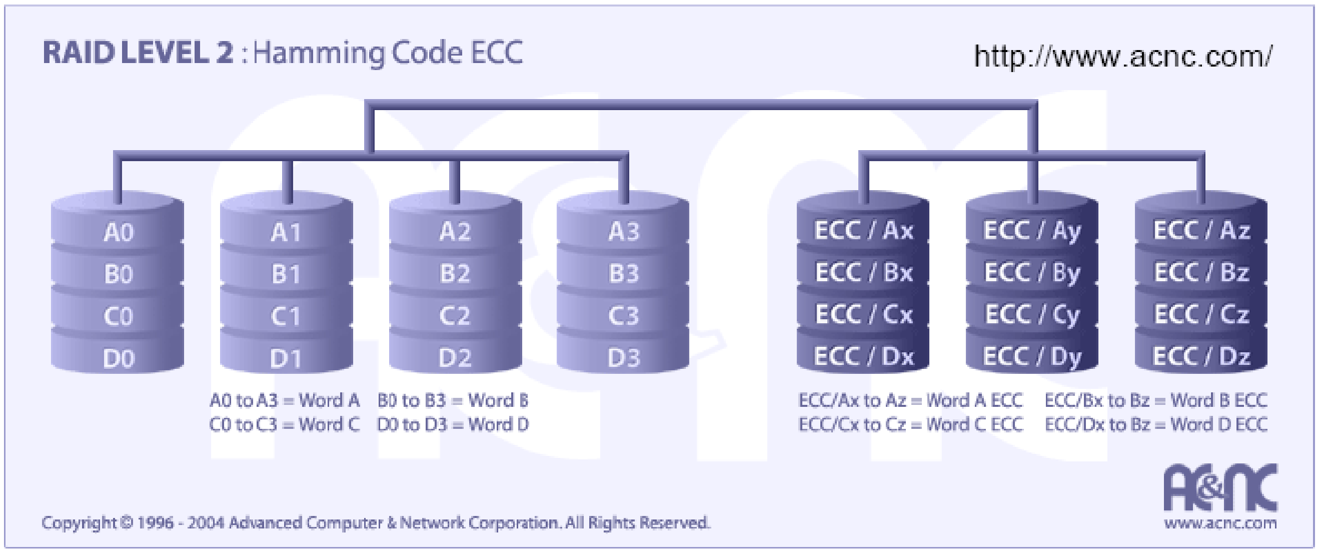
Structure:
Each disk replaced by a mirrored pair

Advantages:
Can double the read transaction rate
No rebuild required

Drawbacks:
Overhead is 100%

RAID-2:
For reducing the factor-of-2 redundancy of mirrored disk arrays is to encode, rather than duplicate, the data. RAID2 used a single-error-correcting Hamming code, combined with striping, to allow for single-disk fault tolerance.

RAID Level 2



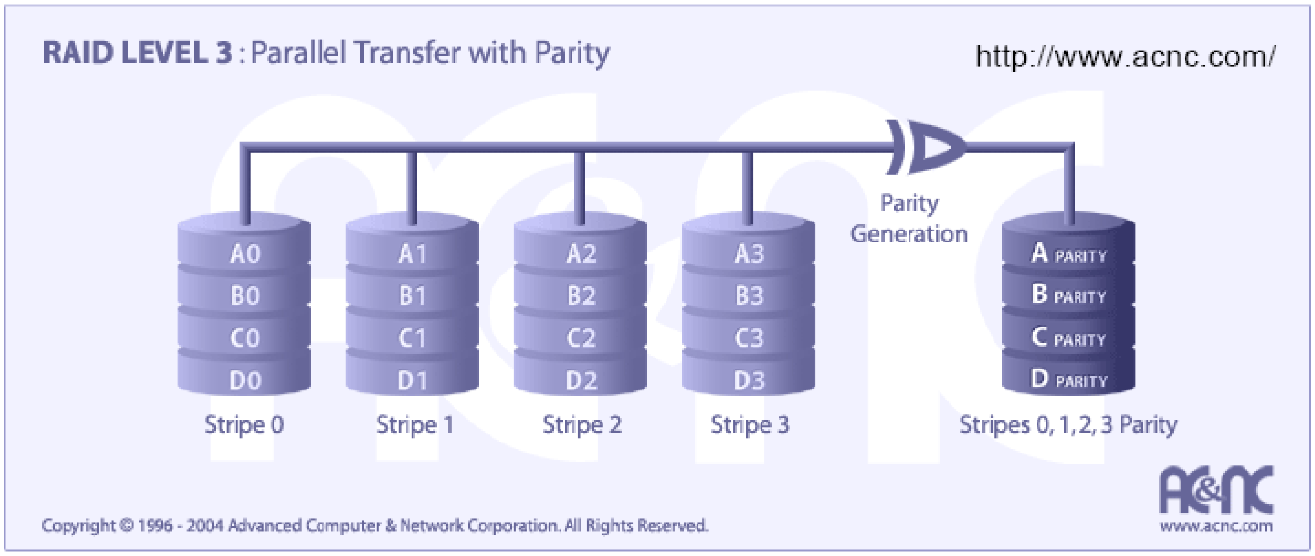
Structure:
Data bits are written to separate disks and ECC bits to others

Advantages:
On-the-fly correction
High transfer rates possible (w/ sync)

Drawbacks:
Potentially high redundancy
High entry-level cost

RAID-3:
In RAID3, data are striped across multiple disks, with parity or checksum information stored on a separate redundant disk. For example, with parity, the file may be four-way striped at the bit or byte level and the XOR of 4 bits or bytes stored on a fifth disk. If a data disk fails, its data can be reconstructed as the XOR of the other three data disks and the parity disk. The reconstructed data may then be stored on a spare disk that is normally unused.

RAID Level 3



Structure:

Data striped across several disks, parity provided on another

Advantages:

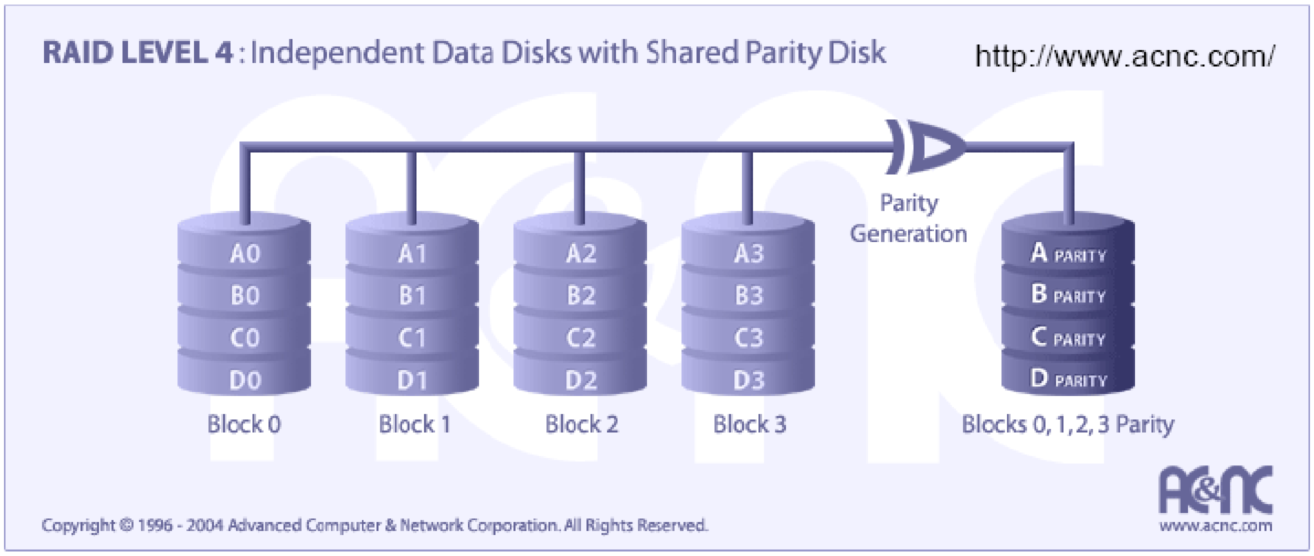
Maintains good throughput even when a disk fails

Drawbacks:

Parity disk forms a bottleneck
Complex controller

RAID-4:
Because disk data are usually accessed in units of sectors rather than bits or bytes, RAID4 applies the parity or checksum to disk sectors. In this case, striping does not affect small files that fit entirely on one sector.
But multiple files cannot be modified simultaneously in view of the need to access the parity disk at the same time.

RAID Level 4



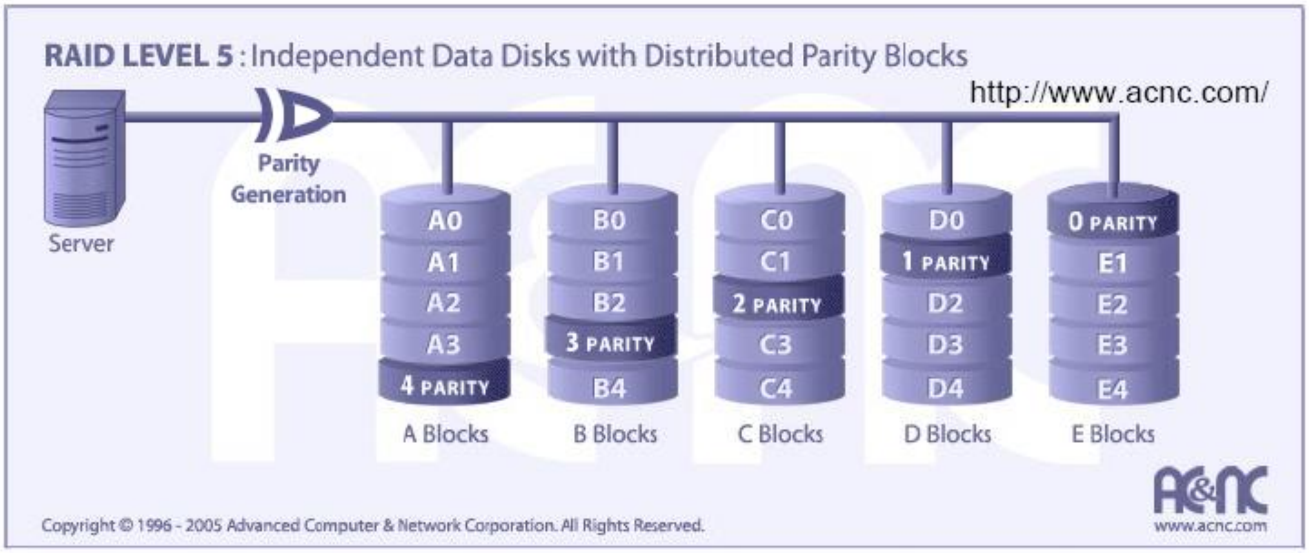
Structure:
Independent blocks on multiple disks share a parity disk

Advantages:
Very high read rate
Low redundancy

Drawbacks:
Low write rate
Inefficient data rebuild

RAID-5:
In RAID5, the above problem is solved by distributing the parity data on multiple disks instead of putting them on a dedicated disk. This scheme distributes the parity accesses and thus eases the performance bottleneck resulting from parity updates.

RAID Level 5

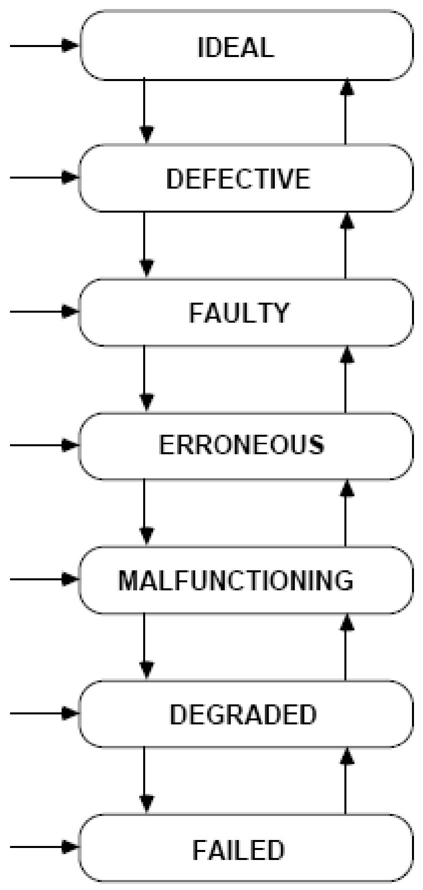


Structure:	Advantages:	Drawbacks:
Parity and data blocks distributed on multiple disks	Very high read rate Medium write rate Low redundancy	Complex controller Difficult rebuild

FAULT-TOLERANT COMPUTING:
Ensuring correct functioning of digital systems in the presence of (permanent and transient) faults is known as fault-tolerant (or dependable) computing.
We distinguish among various undesirable conditions, or impairments, that lead to unreliability in any digital system under following categories:

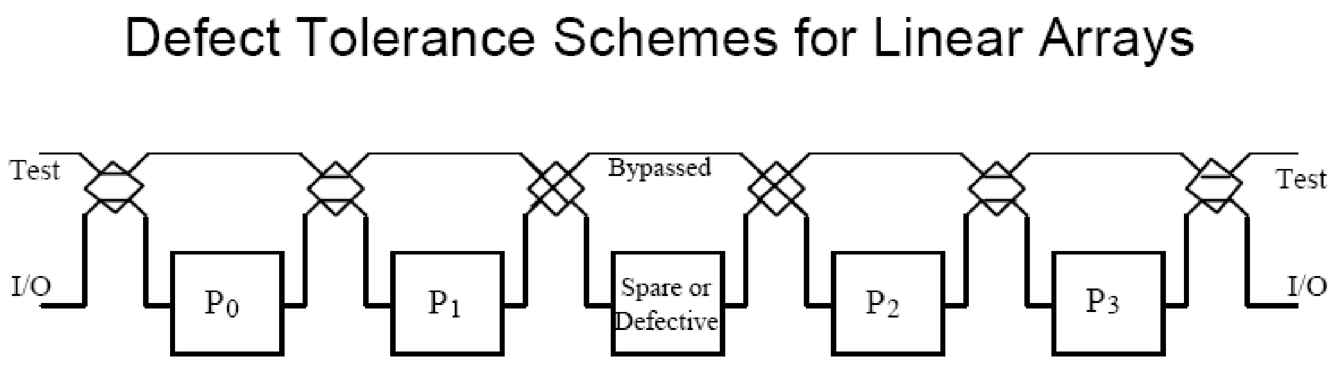
1. **Defect level or component level**, dealing with deviant atomic parts
2. **Fault level or logic level**, dealing with deviant signal values or decisions
3. **Error level or information level**, dealing with deviant data or internal states
4. **Malfunction level or system level**, dealing with deviant functional behavior
5. **Degradation level or service level**, dealing with deviant performance
6. **Failure level or result level**, dealing with deviant outputs or actions

Levels of undesirable conditions in a system:

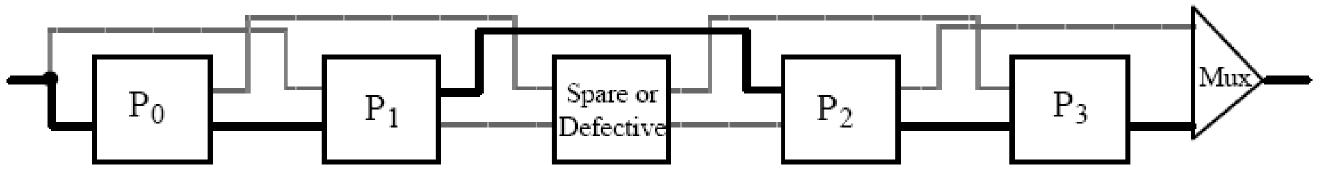


Defect-Level Methods:
Defects are caused in two ways:(1) Improper design or inadequate screening leading to defective system components, and (2) Development of defects as a result of component wear and aging or operating conditions.
A defect may be dormant or ineffective long after its occurrence.
Replacement of sensitive components, as part of scheduled **periodic maintenance**, is one way of removing defects before they develop into faults.
Burn Test or Torture testing the performed on components are usually subjected to loads and stresses that are much higher than those encountered during normal operation. This burning in or torture testing of components results in the development of faults in marginal components (components which are weak) and then identified by fault testing methods.

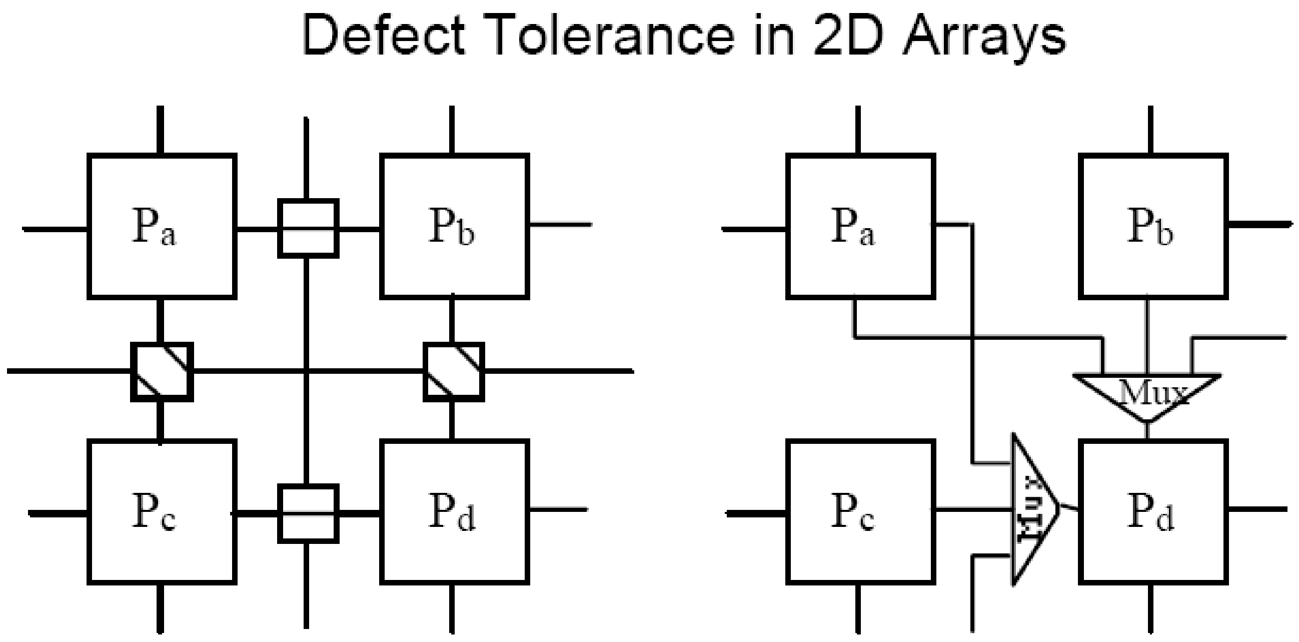
A linear array with a spare processor and reconfiguration switches:



A linear array with a spare processor and embedded switching:



Two types of reconfiguration switching for 2D arrays:
A malfunctioning processor can be bypassed in its row/column by means of a separate switching mechanism



Fault-Level Methods:
Faults are caused in two ways (sideways and downward transitions into the faulty state):

- Design errors leading to incorrect logic circuits
- Using defective components, leading to incorrect logic values or decisions

Defect-based faults can be classified according to duration (permanent, intermittent/recurring, or transient), extent (local or distributed/catastrophic), and effect (dormant or active).
Only active faults produce incorrect logic signals.
Faults are detected through testing:

- Off-line (initially, or periodically in test mode)
- On-line or concurrent (self-testing logic)

Goal of fault tolerance methods:

- Allow uninterrupted operation in the presence of faults.

One way to protect the computations against fault-induced errors is to use duplication and compare the two results or triplication with two-out of-three voting on the three results.

Error-Level Methods:
Errors are caused in two ways (sideways and downward transitions into the erroneous state):

- Improper initialization (e.g., incorrect ROM contents)
- Exercising of faulty circuits, leading fault-induced deviations in stored values or machine state

An error is any deviation of a system's state from the reference state as defined by its specification.
Errors are detected through:

- Encoded (redundant) data, plus code checkers
- Activity monitoring

Goal of error tolerance methods:

- Allow uninterrupted operation in the presence of errors.

PARALLEL PROGRAMMING:

Approaches to develop parallel programs:

- Parallelizing compilers
- Data-parallel programming
- Shared-variable programming
- Communicating processes
- Functional programming

Parallelizing compilers:
A parallelizing compiler is one that takes a standard sequential program (written in Fortran, C, or other language) as input and produces a parallel program as output.
Vectorizing compilers are similar and have been in use for many years in connection with pipelined vector processors.
The idea in both cases is to deduce, through automatic dependence analysis, which computations lack interdependencies and can thus be executed concurrently.

Data-parallel programming:
The APL (Array Processing Language) programming language, with its array and reduction operations, was an early example of an explicit data-parallel programming scheme.
APL's powerful operators allowed the composition of very complicated computations in a few lines
When run on a distributed-memory machine, some Fortran-90 constructs imply inter-processor communication.

Shared-variable programming:
Concurrent Pascal also provides facilities for initiating, delaying, and continuing the execution of processes and a way of scheduling the various calls made by outside procedures in FIFO order.
Modula-2 contains only primitive concurrency features that allow processes to interact through shared variables or via (synchronizing) signals.

Communicating processes:
Communicating processes form the basis of several concurrent programming languages such as Ada and Occam. This approach, which involves passing of messages among concurrently executing processes, has four basic features: (1) process/task naming, (2) synchronization, (3) messaging scheme, and (4) failure handling.

Functional programming:
Based on reduction and evaluation of expressions.
There is no concept of storage, assignment, or branching in a functional program. Rather, results are obtained by applying functions to arguments, incorporating the results as arguments of still other functions, and so forth, until the final results are obtained.

Message Passing Interface (MPI):
MPI standard specifies a library of functions that implement the message-passing model of parallel computation.
MPI provides a common high-level view of a message-passing environment that can be mapped to various physical systems.
Software implemented using MPI functions can be easily ported among machines that support the MPI model
MPI includes functions related to:

- Point-to-point communication (blocking and nonblocking send/receive, . . .)
- Collective communication (broadcast, gather, scatter, total exchange, . . .)
- Aggregate computation (barrier, reduction, and scan or parallel prefix)
- Group management (group construction, destruction, inquiry, . . .)
- Communicator specification (inter-/intracommunicator construction, destruction, . . .)
- Virtual topology specification (various topology definitions, . . .)

Parallel Virtual Machine (PVM):
It is a software platform for developing and running parallel applications on a collection of independent, heterogeneous, computers that are interconnected in a variety of ways.
PVM defines a suite of user-interface primitives that support both the shared-memory and the message-passing parallel programming paradigms. These primitives provide functions similar to those listed for MPI and are embedded within a procedural host language (usually Fortran or C).
A PVM support process or daemon (PVMD) runs independently on each host, performing message routing and control functions.
PVMDs perform the following functions:

- Exchange network configuration information
- Allocate memory to packets that are in transit between distributed tasks
- Coordinate the execution of tasks on their associated hosts

PARALLEL OPERATING SYSTEMS

- In 1960, Burroughs introduced its AOSP (Automatic Operating and Scheduling Program) for a 4-processor shared-memory computer.
- Hydra operating system was developed in 1970 for handling 16-processor machine named Medusa of Carnegie-Mellon University.
- StarOS was developed for the system Cm* having 50-processors.
- OSF/1 operating system introduced by the nonprofit Open Software Foundation was developed for the Intel Paragon, and IBM's AIX parallel computers.
- The Mach operating system developed at Carnegie-Mellon University is sometimes classified as a Unix-based system in view of its ability to run serial Unix applications as single-threaded tasks. Mach's capability-based object-oriented design supports multiple tasks and threads that can communicate with each other via messages or memory objects. Tasks and threads are conceptually the same in that they specify concurrently executable code segments. However, context switching between two threads executing the same task is much faster than that between two arbitrary threads.

PARALLEL FILE SYSTEMS

A parallel file system efficiently maps data access requests by processors to high-bandwidth data transfers between primary and secondary memory devices.
It has following characteristics:

- Concurrent access to files by multiple independent processes
- Shared access to files by groups of cooperating processes
- Access to large amounts of data by a single process

A read request issued by a user process is sent to a dispatcher process which creates a server task or thread to perform the required transfer. The actual data needed may come in part from cache copies of various disk pages and will be integrated finally.