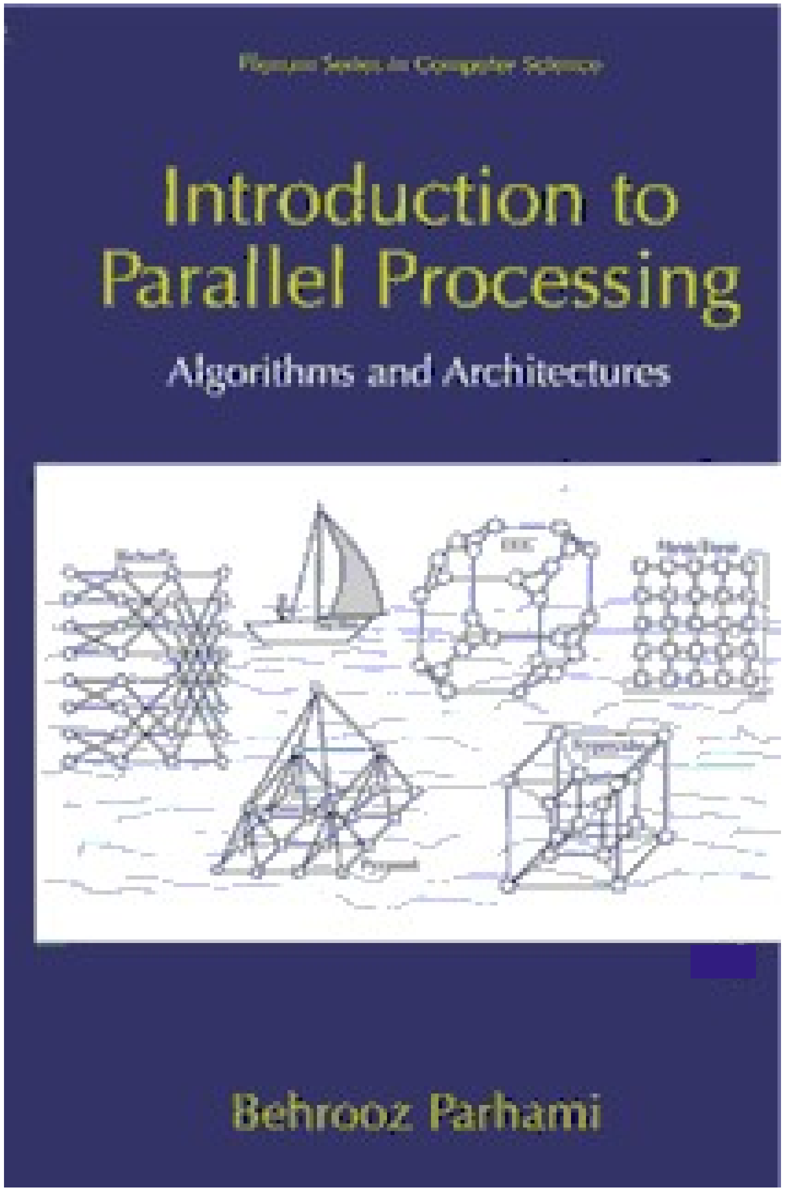

Plenum Series in Computer Science
Introduction to Parallel Processing
Algorithms and Architectures
Behrooz Parhami

Part V
Some Broad Topic

	Part I: Fundamental Concepts	Background and Motivation	1. Introduction to Parallelism 2. A Taste of Parallel Algorithms
		Complexity and Models	3. Parallel Algorithm Complexity 4. Models of Parallel Processing
Architectural Variations	Part II: Extreme Models	Abstract View of Shared Memory	5. PRAM and Basic Algorithms 6. More Shared-Memory Algorithms
		Circuit Model of Parallel Systems	7. Sorting and Selection Networks 8. Other Circuit-Level Examples
	Part III: Mesh-Based Architectures	Data Movement on 2D Arrays	9. Sorting on a 2D Mesh or Torus 10. Routing on a 2D Mesh or Torus
		Mesh Algorithms and Variants	11. Numerical 2D Mesh Algorithms 12. Other Mesh-Related Architectures
	Part IV: Low-Diameter Architectures	The Hypercube Architecture	13. Hypercubes and Their Algorithms 14. Sorting and Routing on Hypercubes
		Hypercubic and Other Networks	15. Other Hypercubic Architectures 16. A Sampler of Other Networks
	Part V: Some Broad Topics	Coordination and Data Access	17. Emulation and Scheduling 18. Data Storage, Input, and Output
		Robustness and Ease of Use	19. Reliable Parallel Processing 20. System and Software Issues
	Part VI: Implementation Aspects	Control-Parallel Systems	21. Shared-Memory MIMD Machines 22. Message-Passing MIMD Machines
		Data Parallelism and Conclusion	23. Data-Parallel SIMD Machines 24. Past, Present, and Future

About This Presentation

This presentation is intended to support the use of the textbook *Introduction to Parallel Processing: Algorithms and Architectures* (Plenum Press, 1999, ISBN 0-306-45970-1). It was prepared by the author in connection with teaching the graduate-level course ECE 254B: Advanced Computer Architecture: Parallel Processing, at the University of California, Santa Barbara. Instructors can use these slides in classroom teaching and for other educational purposes. Any other use is strictly prohibited.

Edition	Released	Revised	Revised	Revised
First	Spring 2005	Spring 2006	Fall 2008	Fall 2010
		Winter 2013	Winter 2014	Winter 2016
		Winter 2019*	Winter 2021*	

*Chapters 17-18 only

V Some Broad Topics

Study topics that cut across all architectural classes:

- Mapping computations onto processors (scheduling)
- Ensuring that I/O can keep up with other subsystems
- Storage, system, software, and reliability issues

Topics in This Part	
Chapter 17	Emulation and Scheduling
Chapter 18	Data Storage, Input, and Output
Chapter 19	Reliable Parallel Processing
Chapter 20	System and Software Issues

17 Emulation and Scheduling

Mapping an architecture or task system onto an architecture

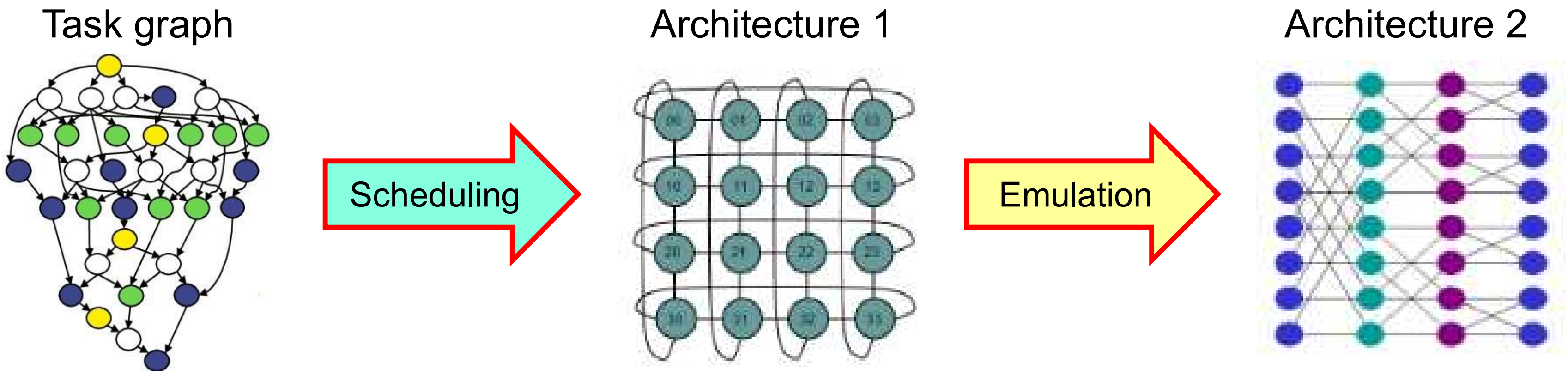
- Learn how to achieve algorithm portability via emulation
- Become familiar with task scheduling in parallel systems

Topics in This Chapter	
17.1	Emulations Among Architectures
17.2	Distributed Shared Memory
17.3	The Task Scheduling Problem
17.4	A Class of Scheduling Algorithms
17.5	Some Useful Bounds for Scheduling
17.6	Load Balancing and Dataflow Systems

17.1 Emulations Among Architectures

Need for scheduling:

a. Assign tasks to compute nodes so as to optimize system performance
b. The goal of scheduling is to make best use of nodes and links
c. Once derived, schedules may be adjusted via load balancing



Task graph

Scheduling

Architecture 1

Emulation

Architecture 2

Usefulness of emulation:

a. Develop algorithms/schedules quickly for a new architecture
b. Program/schedule on a user-friendly architecture, then emulate it
c. Show versatility of a new architecture by emulating the hypercube on it

Virtual Network Embedding (VNE)

VN requests & substrate net:

Topologies
Nodes, capacities
Links, capacities

VNE algorithm:

Map VN nodes to substrate nodes

Map VN links to substrate links/paths

Observe limits

Optimize

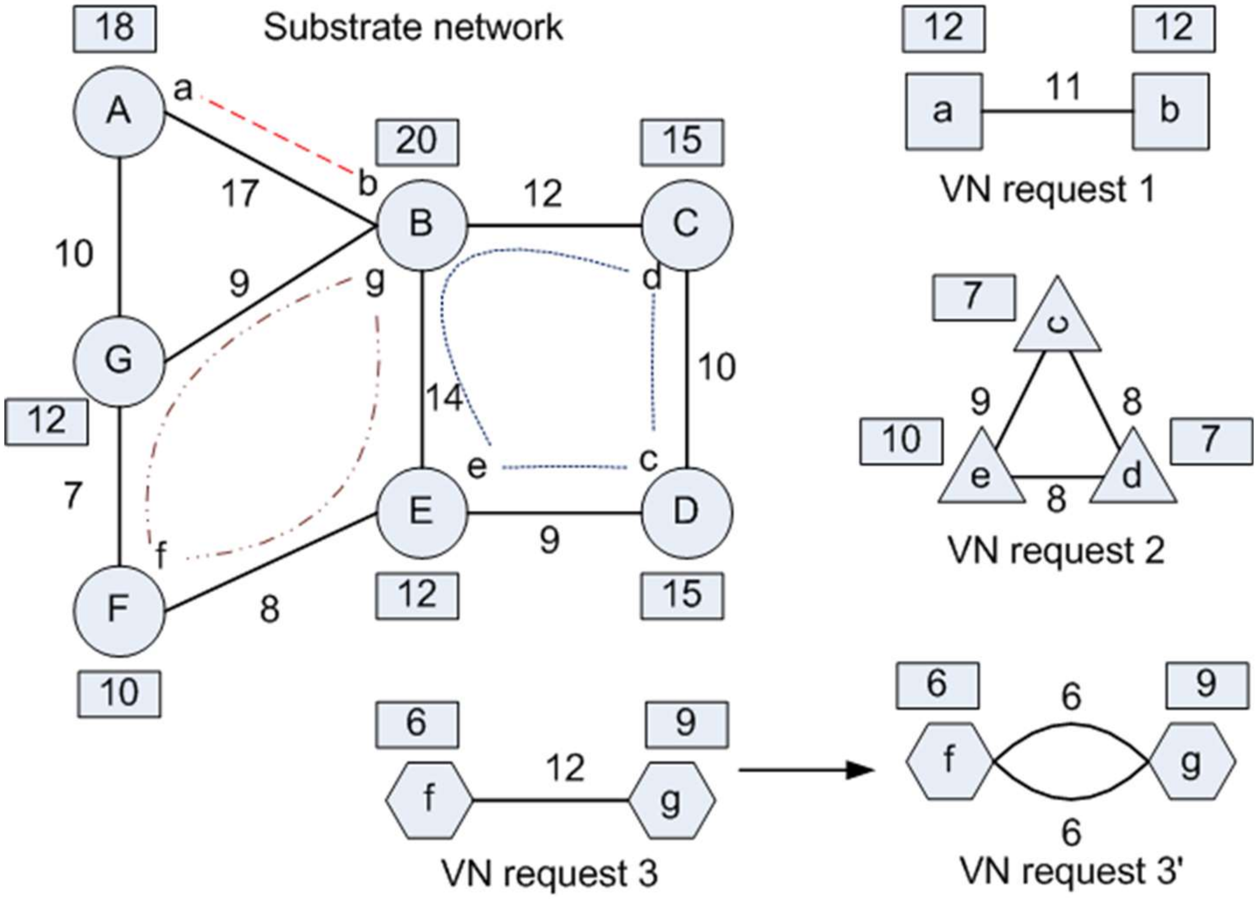
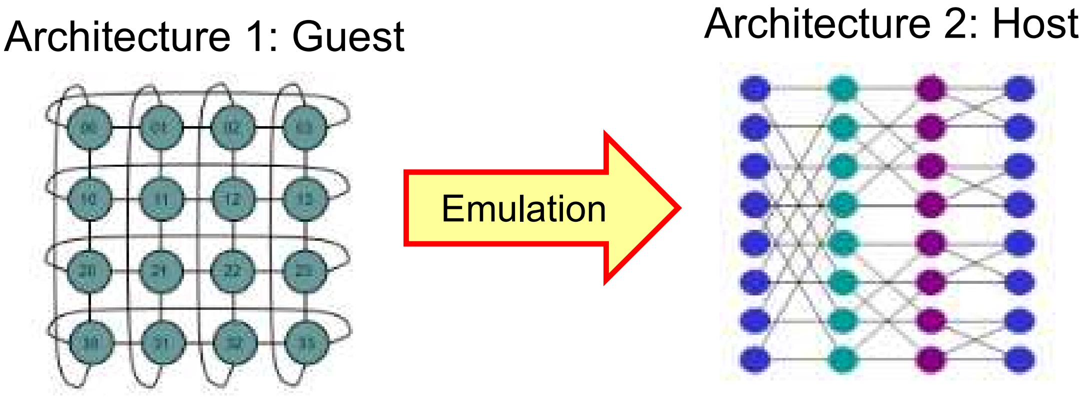


Image source: "FELL: A Flexible Virtual Network Embedding Algorithm with Guaranteed Load Balancing," *Proc. ICC*, 2011

Simple Emulation Results



Architecture 1: Guest

Emulation

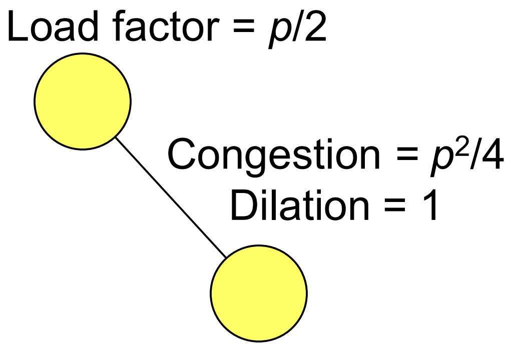
Architecture 2: Host

We saw, for example, that a 2^q-node hypercube has the 2^q-node cycle as a subgraph (is Hamiltonian), but not a balanced binary tree with $2^q - 1$ nodes

Two general emulation results:

1. Emulation via graph embedding

Slowdown ≤ dilation × congestion × load factor
Example: K_2 emulating K_p
In general, the effects are not multiplicative



Load factor = $p/2$

Congestion = $p^2/4$
Dilation = 1

2. PRAM emulating a degree-d network

EREW PRAM can emulate any degree-d network with slowdown $O(d)$

Versatility of the Butterfly Network

A (wrapped) butterfly can emulate any degree-*d* network with O(*d* log *p*) slowdown

Thus, butterfly is a bounded-degree network that is universally efficient

Idea used in proof: One communication step in a degree-*d* network can be decomposed into at most *d* permutation routing steps

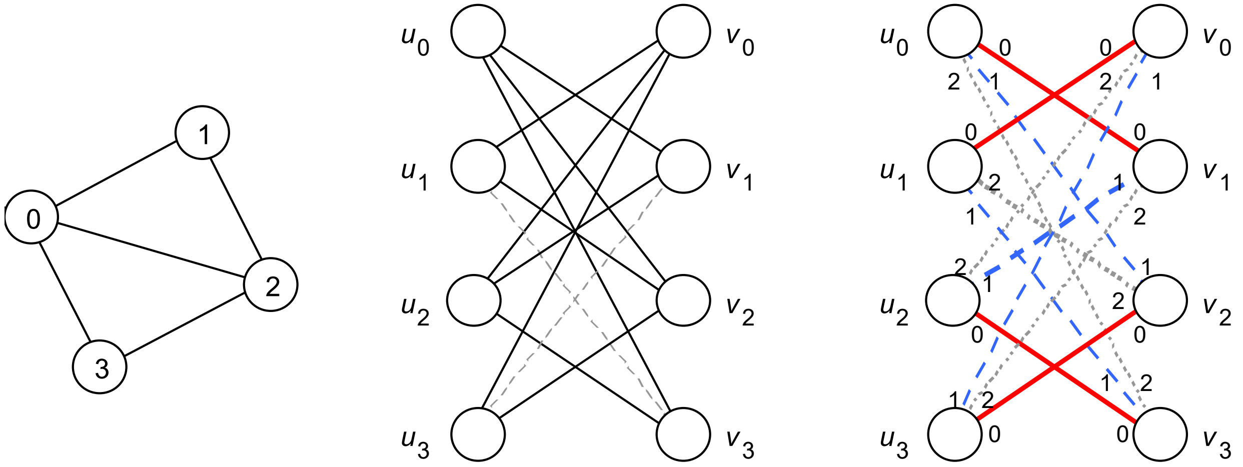
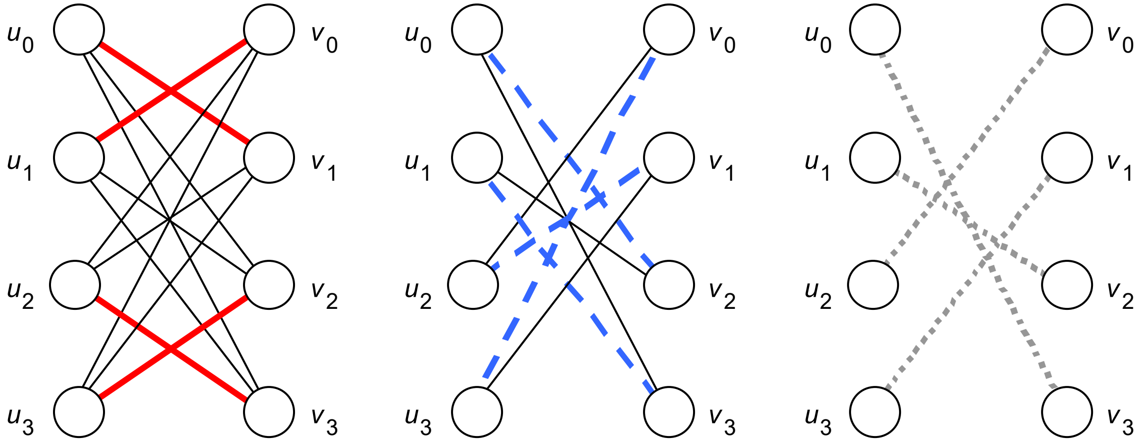


Fig. 17.1 Converting a routing step in a degree-3 network to three permutations or perfect matchings.



17.2 Distributed Shared Memory

Randomized emulation of the *p*-processor PRAM on *p*-node butterfly

Use hash function to map memory locations to modules

p locations → *p* modules, not necessarily distinct

With high probability, at most O(log *p*) of the *p* locations will be in modules located in the same row

Average slowdown = O(log *p*)

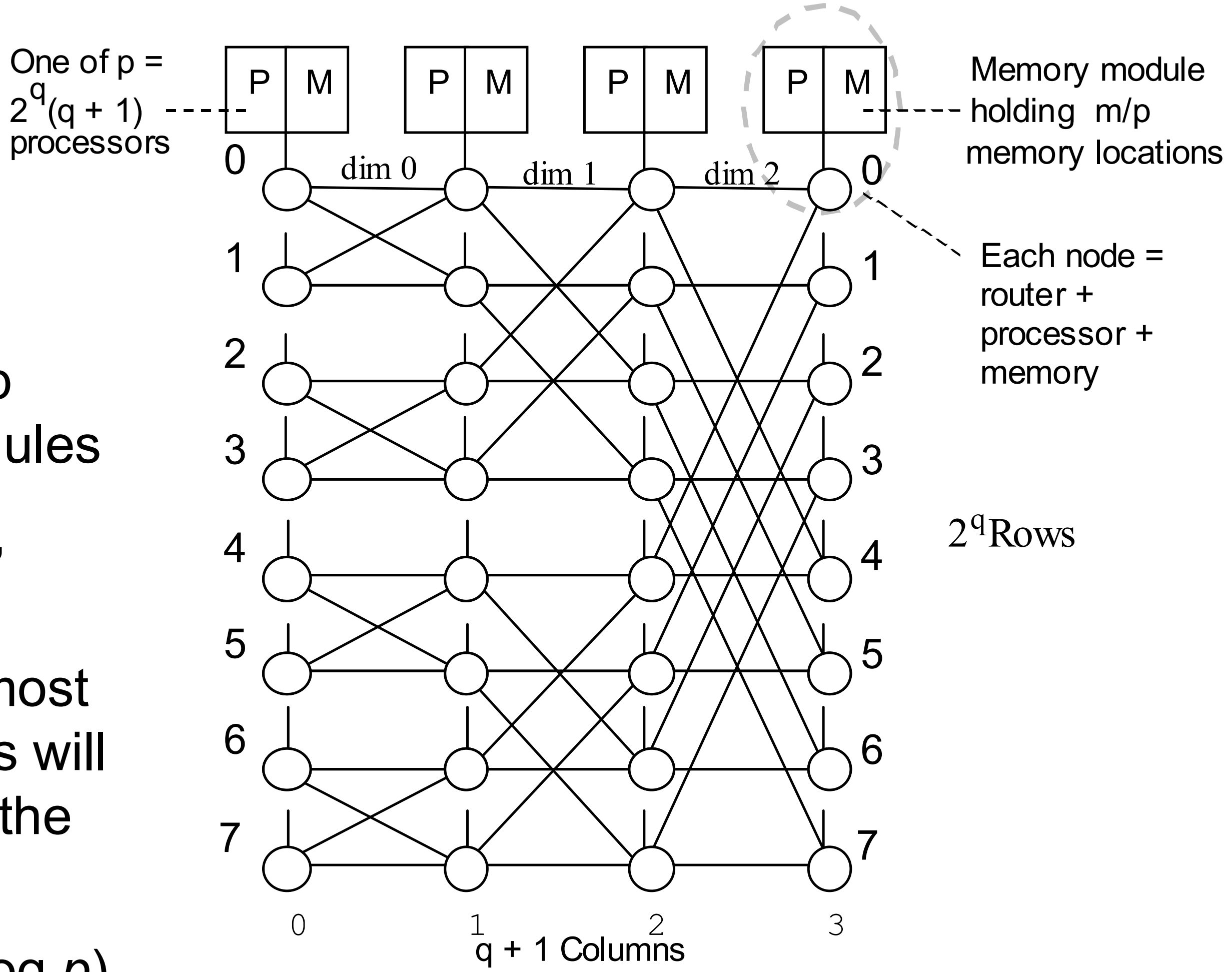

One of p = $2^q(q + 1)$ processors
P
M
P
M
P
M
P
M
Memory module holding m/p memory locations
dim 0
dim 1
dim 2
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
Each node = router + processor + memory
2^qRows
0
1
2
3
q + 1 Columns

Fig. 17.2 Butterfly distributed-memory machine emulating the PRAM.

PRAM Emulation with Butterfly MIN

Emulation of the *p*-processor PRAM on (*p* log *p*)-node butterfly, with memory modules and processors connected to the two sides; O(log *p*) avg. slowdown

Less efficient than Fig. 17.2, which uses a smaller butterfly

By using *p* / (log *p*) physical processors to emulate the *p*-processor PRAM, this new emulation scheme becomes quite efficient (pipeline the memory accesses of the log *p* virtual processors assigned to each physical processor)

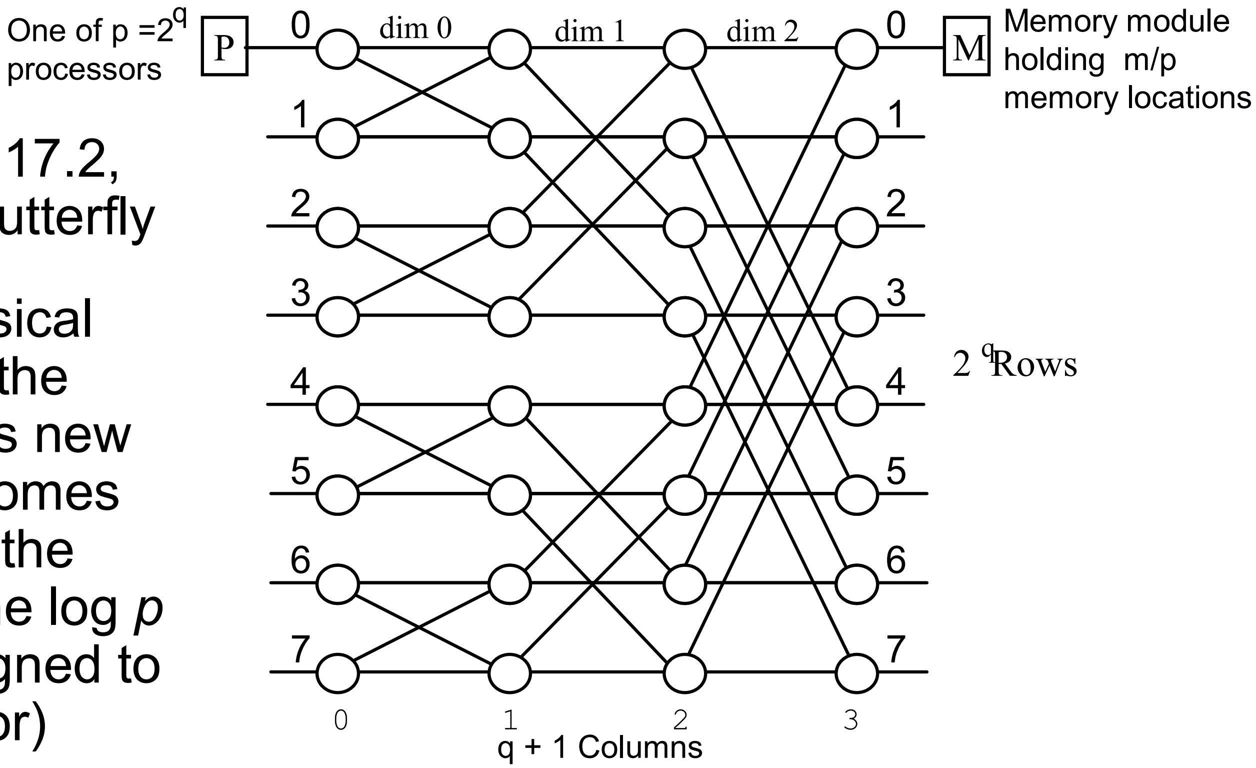


Fig. 17.3 Distributed-memory machine, with a butterfly multistage interconnection network, emulating the PRAM.

Deterministic Shared-Memory Emulation

Deterministic emulation of *p*-processor PRAM on *p*-node butterfly

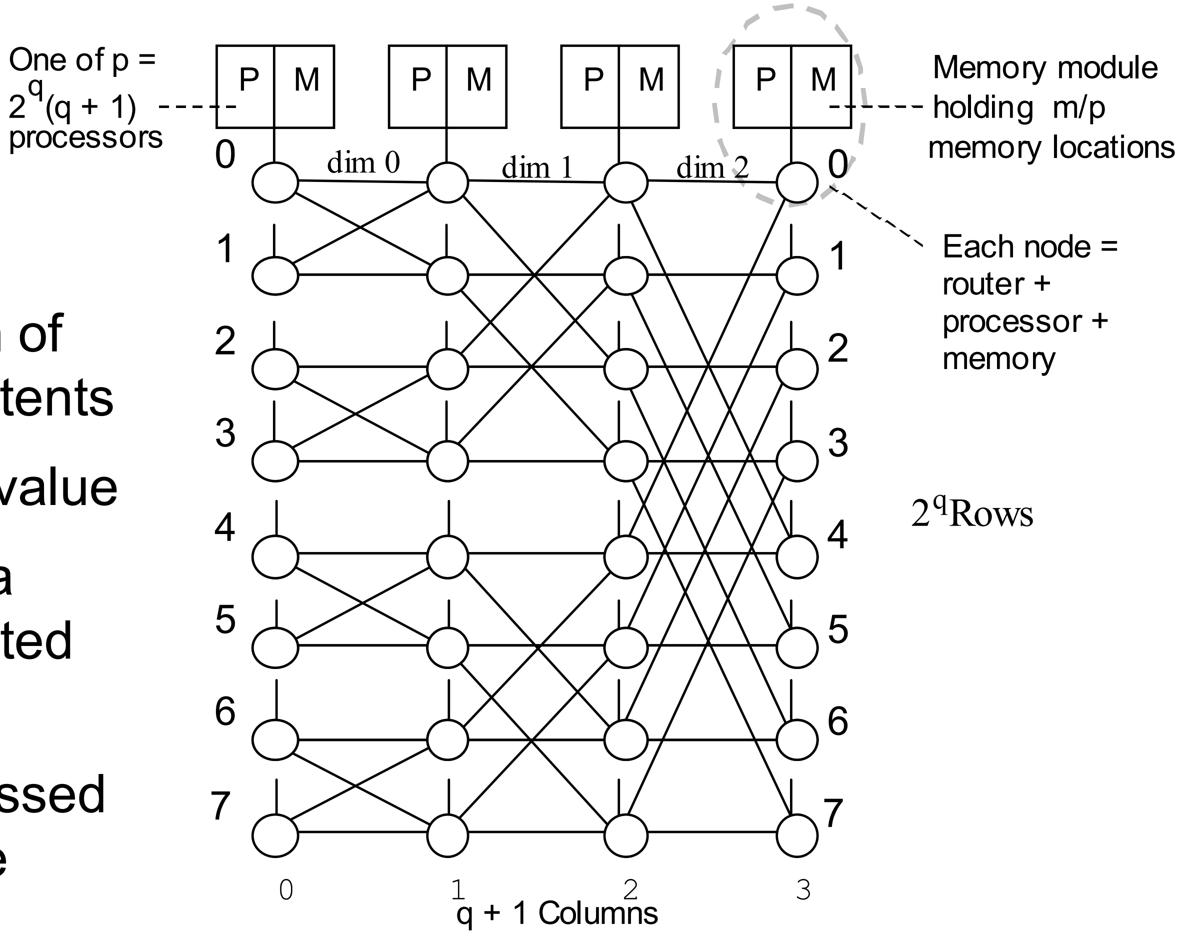
Store $\log_2 m$ copies of each of the m memory location contents

Time-stamp each updated value

A “write” is complete once a majority of copies are updated

A “read” is satisfied when a majority of copies are accessed and the one with latest time stamp is used

Why it works: A few congested links won’t delay the operation



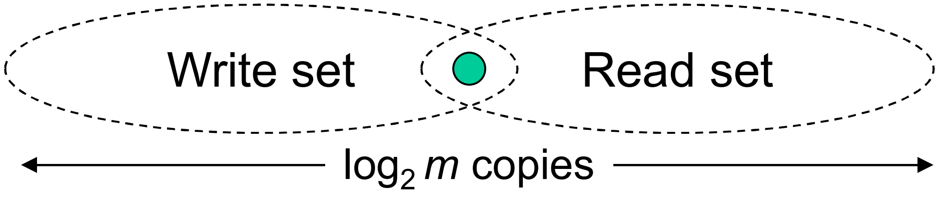
One of p = $2^q(q + 1)$ processors

Memory module holding m/p memory locations

Each node = router + processor + memory

2^q Rows

q + 1 Columns



Write set

Read set

$\log_2 m$ copies

PRAM Emulation Using Information Dispersal

Instead of (log *m*)-fold replication of data, divide each data element into *k* pieces and encode the pieces using a redundancy factor of 3, so that any *k*/3 pieces suffice for reconstructing the original data

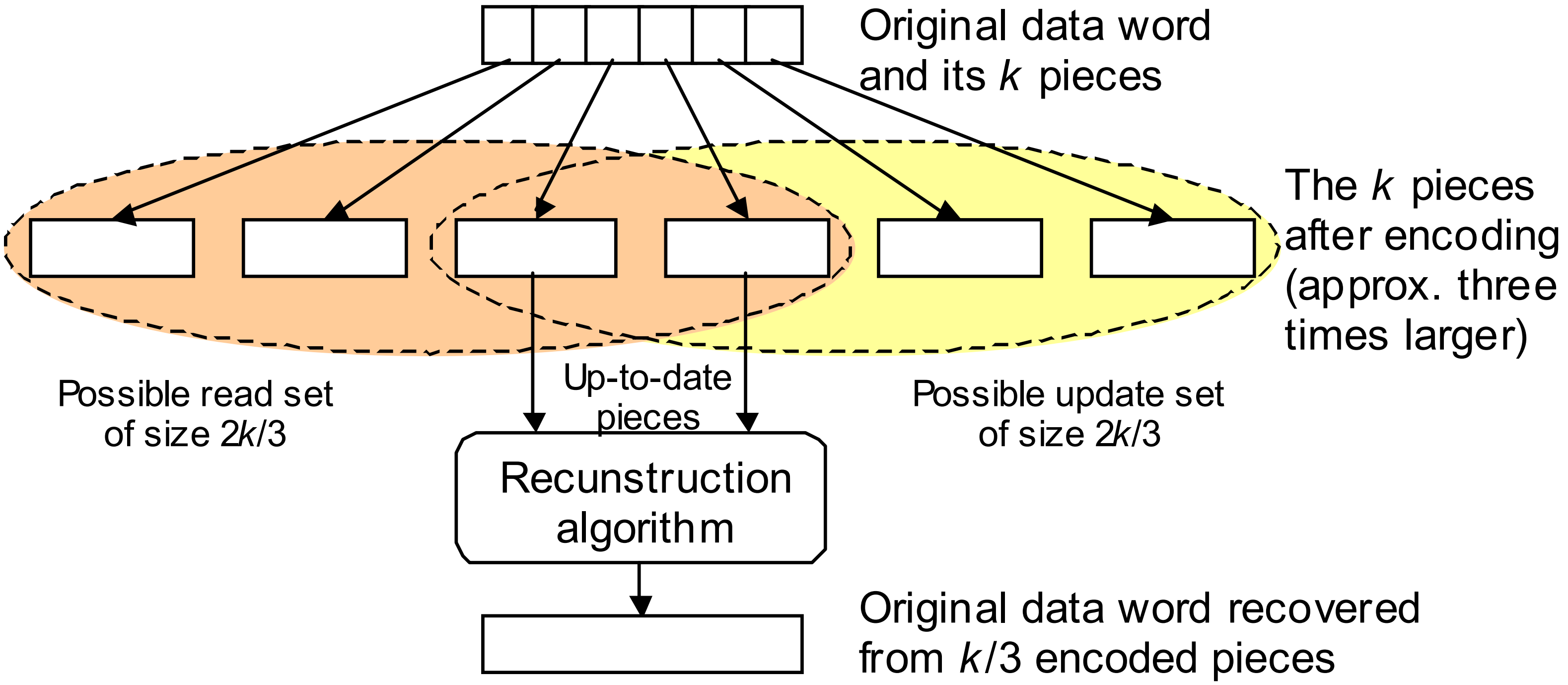


Fig. 17.4 Illustrating the information dispersal approach to PRAM emulation with lower data redundancy.

17.3 The Task Scheduling Problem

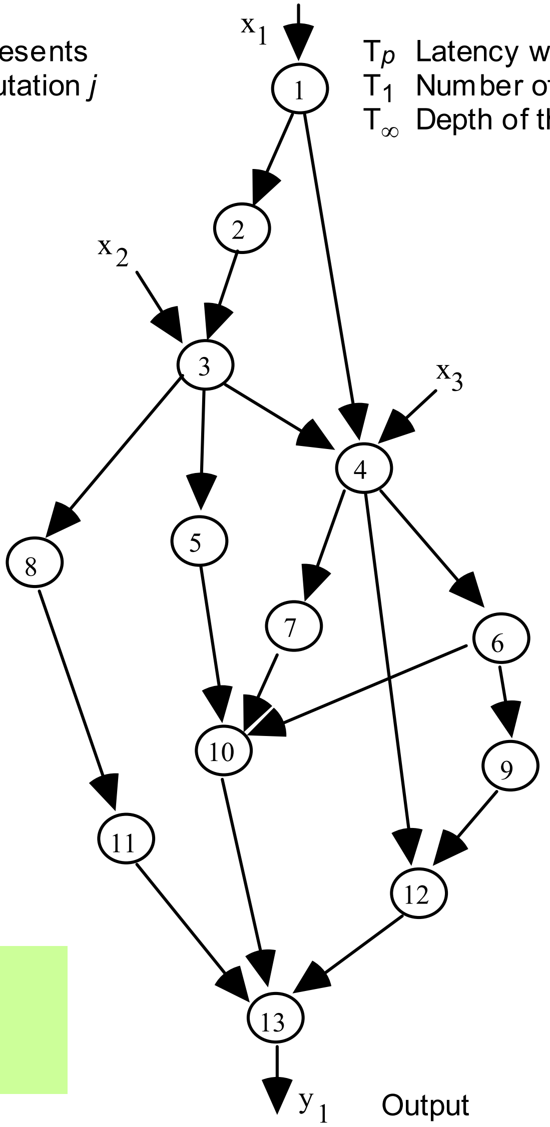
Task scheduling parameters and “goodness” criteria

Running times for tasks
Creation (static/dynamic)
Importance (priority)
Relationships (precedence)
Start times (release times)
End times (deadlines)

Types of scheduling algorithms

Preemptive/nonpreemptive,
fine/medium/coarse grain

Vertex v_j represents task or computation j



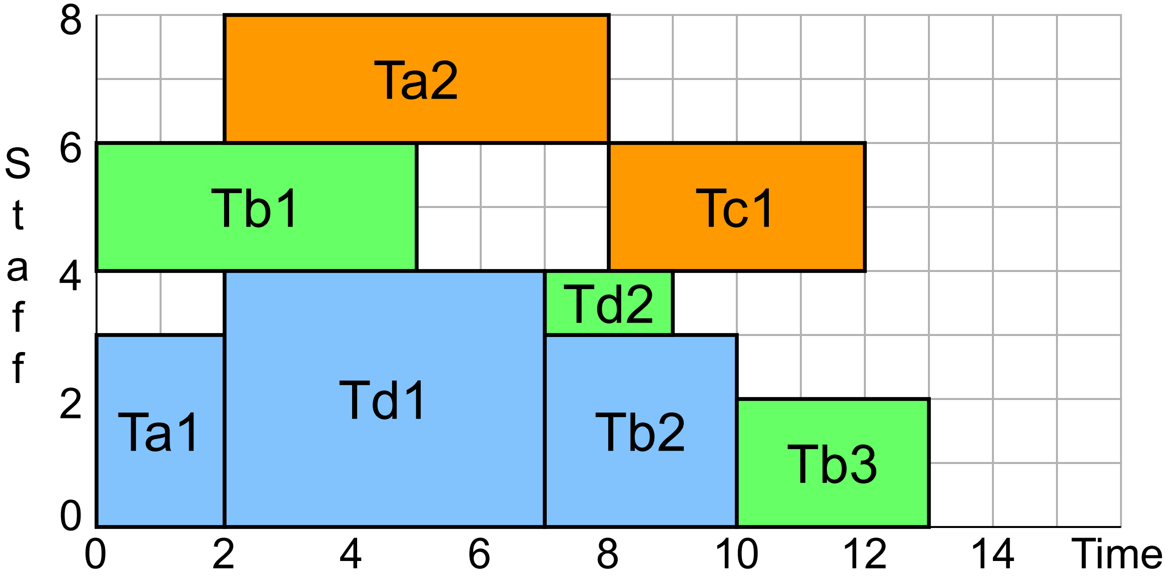
T_p Latency with p processors
T_1 Number of nodes (here 13)
T_∞ Depth of the graph (here 8)

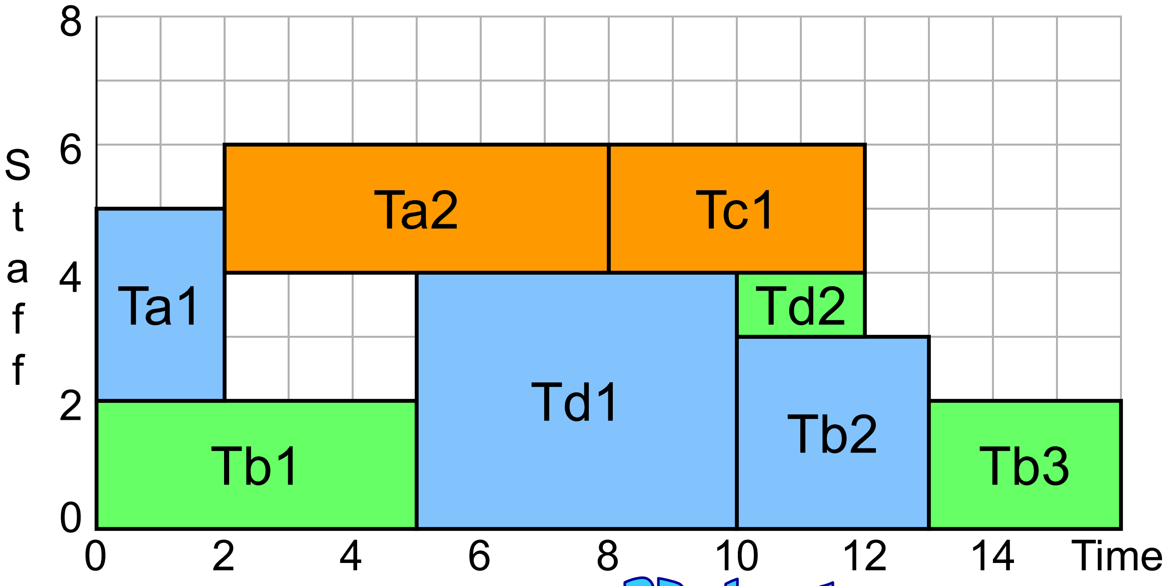
Fig. 17.5   Example task system showing communications or dependencies.

Job-Shop Scheduling

Job	Task	Machine	Time	Staff
Ja	Ta1	M1	2	3
Ja	Ta2	M3	6	2
Jb	Tb1	M2	5	2
Jb	Tb2	M1	3	3
Jb	Tb3	M2	3	2
Jc	Tc1	M3	4	2
Jd	Td1	M1	5	4
Jd	Td2	M2	2	1

M1 🟦 M2 🟩 M3 🟧

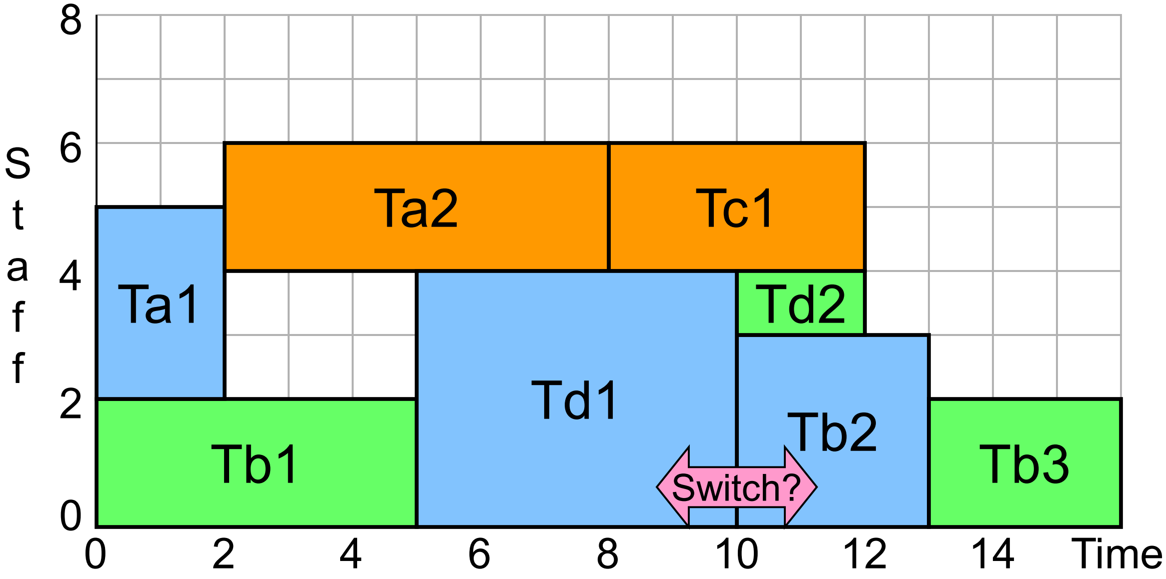


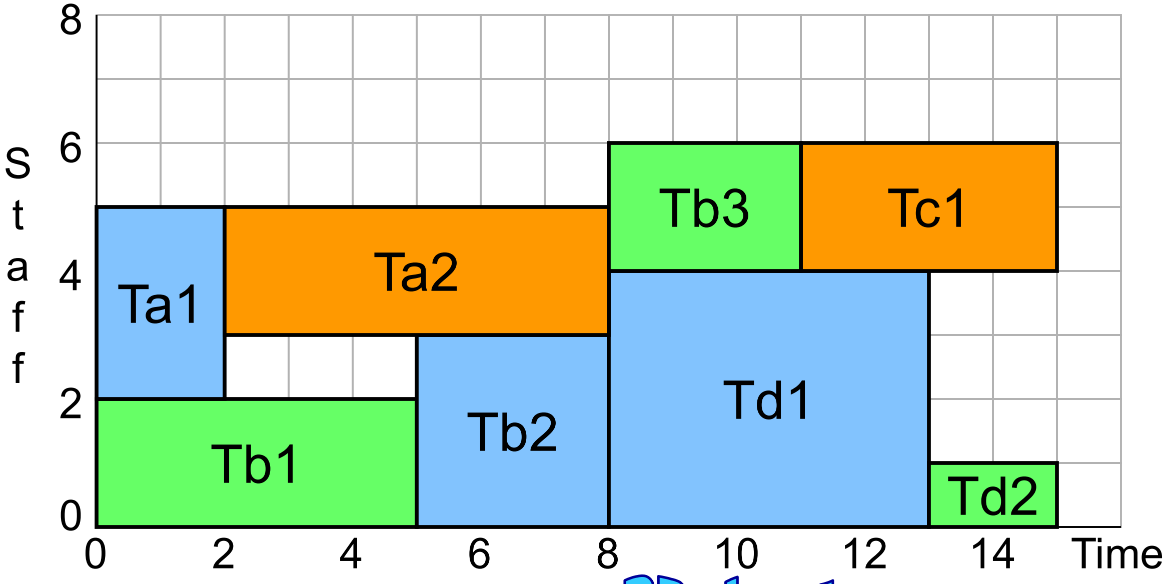


Schedule Refinement

Job	Task	Machine	Time	Staff
Ja	Ta1	M1	2	3
Ja	Ta2	M3	6	2
Jb	Tb1	M2	5	2
Jb	Tb2	M1	3	3
Jb	Tb3	M2	3	2
Jc	Tc1	M3	4	2
Jd	Td1	M1	5	4
Jd	Td2	M2	2	1

M1 🟦 M2 🟩 M3 🟧





Complexity of Scheduling Problems

Most scheduling problems, even with 2 processors, are NP-complete

Easy, or tractable (polynomial-time), cases include:

1. Unit-time tasks, with 2 processors
2. Task graphs that are forests, with any number of processors

Surprisingly hard, or intractable, cases include:

1. Tasks of running time 1 or 2, with 2 processors (nonpreemptive)
2. Unit-time tasks on 3 or more processors

Many practical scheduling problems are solved by heuristics

Heuristics typically have decision parameters that can be tuned to make them suitable for a particular application context

The scheduling literature is full of different heuristics and experimental studies of their performance in different domains

17.4 A Class of Scheduling Algorithms

List scheduling

Assign a priority level to each task
Construct task list in priority order; tag tasks that are ready for execution
At each step, assign to an available processor the first tagged task
Update the tags upon each task termination

With identical processors, list schedulers differ only in priority assignment

A possible priority assignment scheme for list scheduling:

1. Find the depth T_∞ of the task graph (indicator of min possible exec time)
2. Take T_∞ as a goal for the running time T_p
3. Determine the latest time when each task can be started if our goal is to be met (done by "layering" the nodes, beginning with the output node)
4. Assign task priorities in order of the latest possible times, breaking ties, e.g., by giving priority to tasks with more descendants

List Scheduling Example

A possible priority assignment scheme:

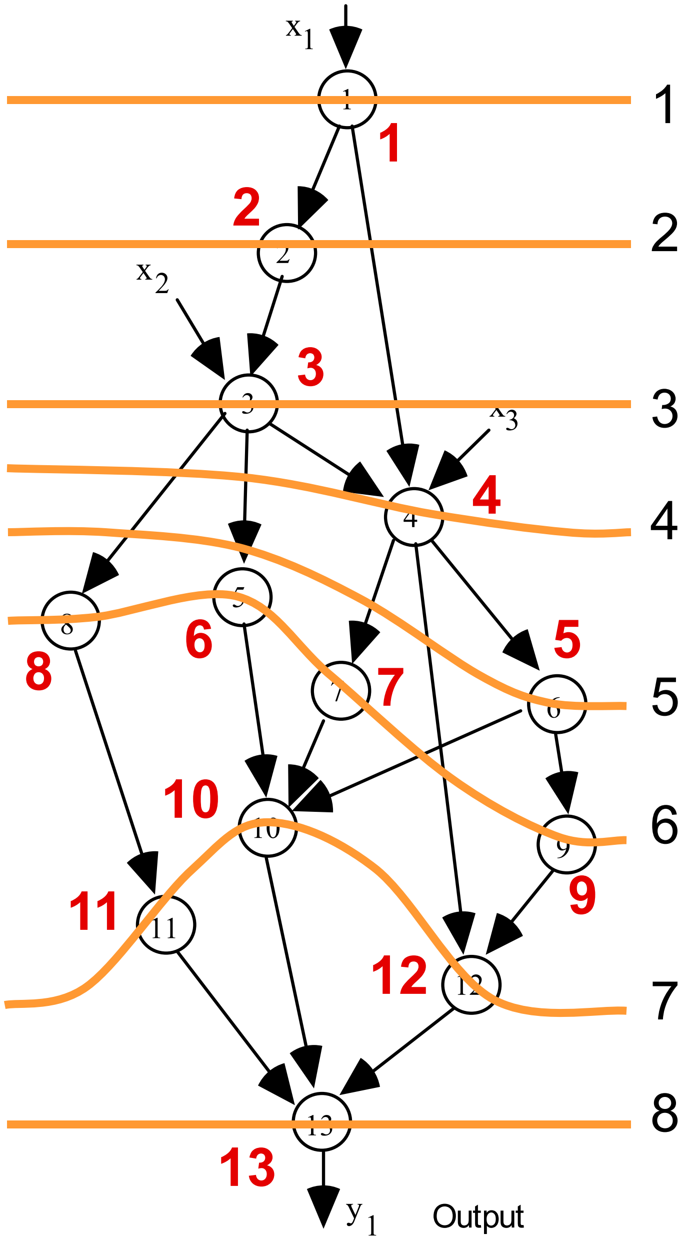
1. Find the depth T_∞ of the task graph
2. Take T_∞ as a goal for the running time T_p
3. Determine the latest possible start times
4. Assign priorities in order of latest times

T_∞ = 8 (execution time goal)

Latest start times: see the layered diagram

Priorities: shown on the diagram in red

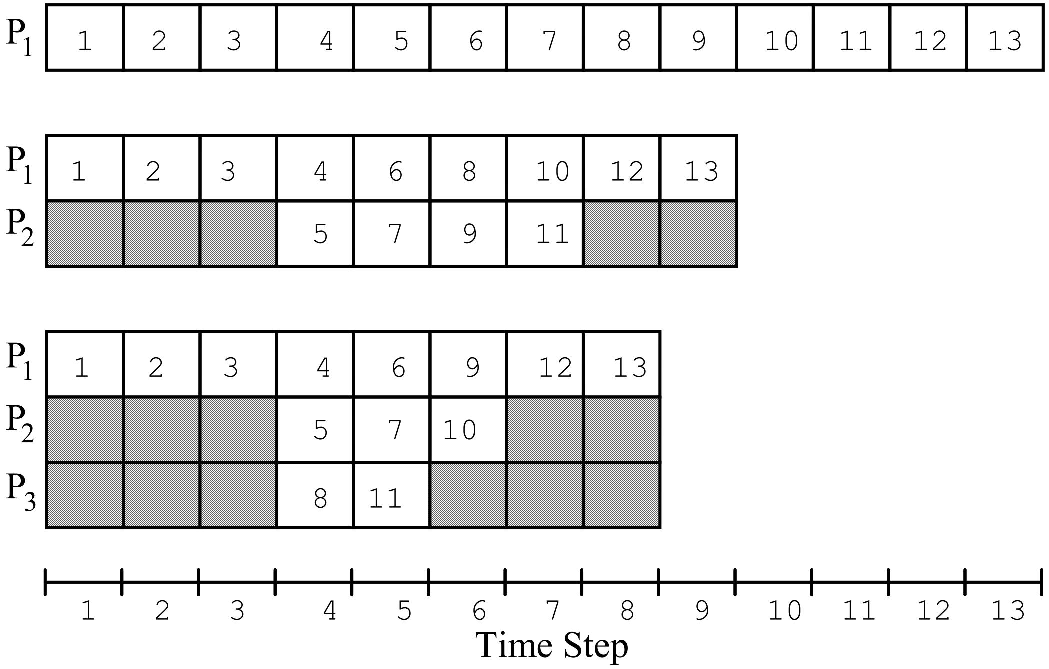
In this particular example, the tie-breaking rule of giving priority to a task with more descendants is of no help, but generally it leads to improvement in execution time



Assignment to Processors

Tasks listed in priority order

1* 2 3 4 6 5 7 8 9 10 11 12 13 $t = 1$ v_1 scheduled
2* 3 4 6 5 7 8 9 10 11 12 13 $t = 2$ v_2 scheduled



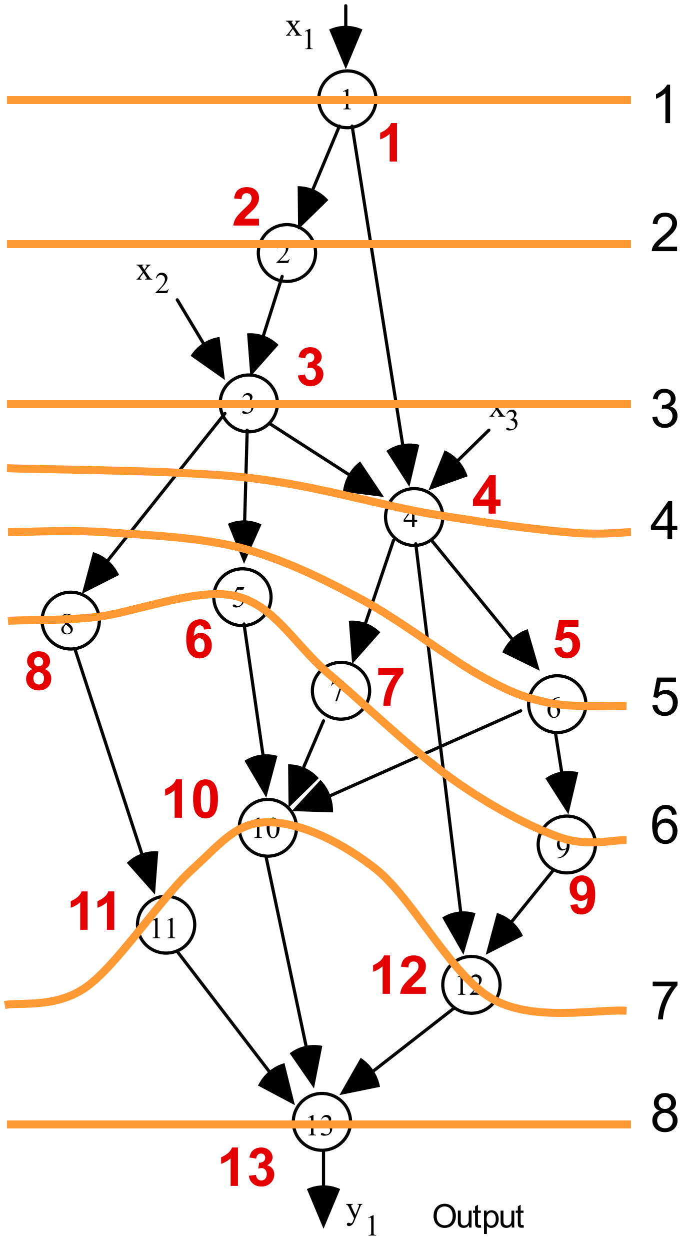
P_1	1	2	3	4	5	6	7	8	9	10	11	12	13

P_1	1	2	3	4	6	8	10	12	13
P_2	▨	▨	▨	5	7	9	11	▨	▨

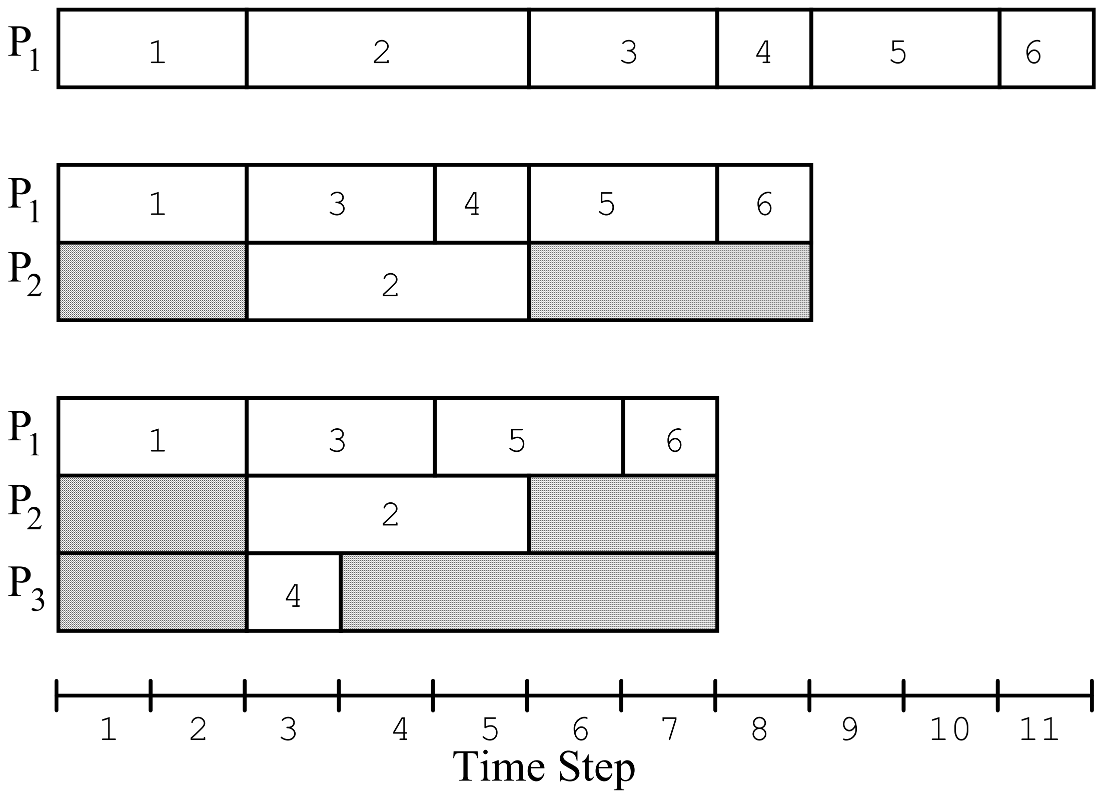
P_1	1	2	3	4	6	9	12	13
P_2	▨	▨	▨	5	7	10	▨	▨
P_3	▨	▨	▨	8	11	▨	▨	▨

1 2 3 4 5 6 7 8 9 10 11 12 13
Time Step

Fig. 17.6 Schedules with p = 1, 2, 3 processors for an example task graph with unit-time tasks.



Scheduling with Non-Unit-Time Tasks



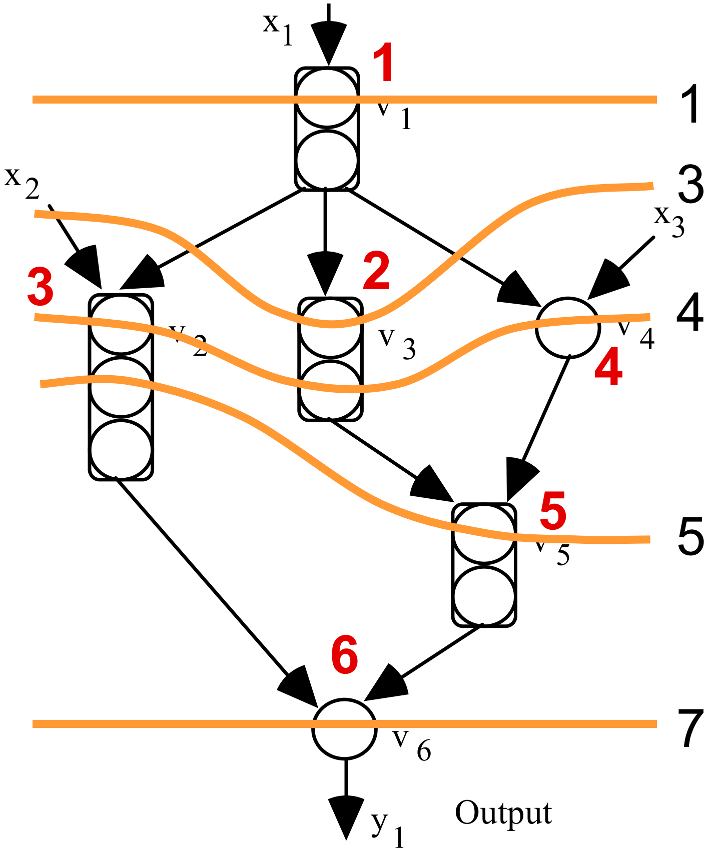


Fig. 17.8 Schedules with *p* = 1, 2, 3 processors for an example task graph with nonuniform running times.

Fig. 17.7 Example task system with task running times of 1, 2, or 3 units.

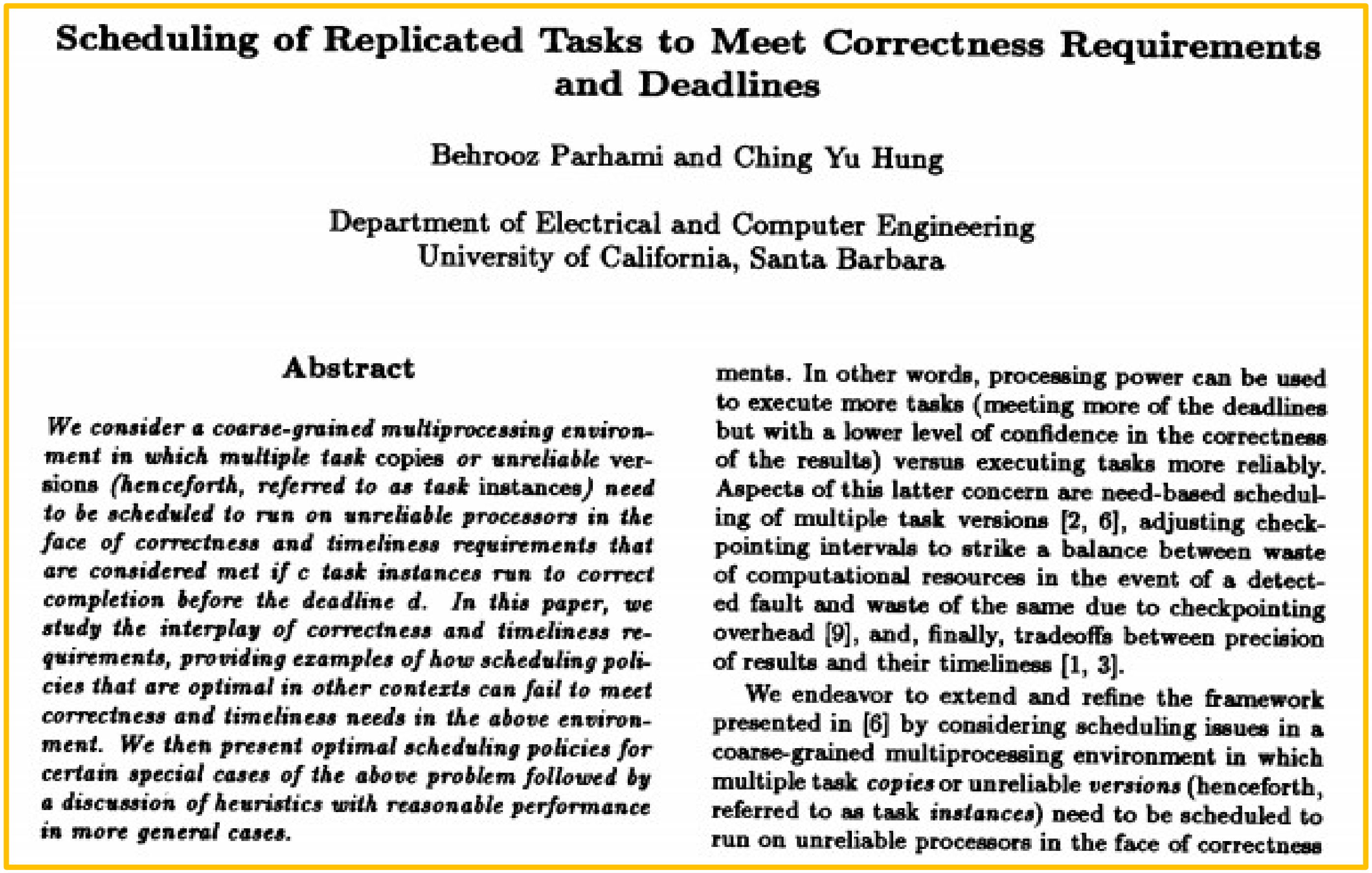
Fault-Tolerant Scheduling

Tasks, or nodes on which they run, may have imperfect reliabilities
[This topic to be developed further]

[Reference 1]

B. Parhami, "A Unified Approach to Correctness and Timeliness Requirements for Ultrareliable Concurrent Systems," *Proc. 4th Int'l Parallel Processing Symp.*, April 1990, pp. 733-747.

[Reference 2]



Scheduling of Replicated Tasks to Meet Correctness Requirements and Deadlines

Behrooz Parhami and Ching Yu Hung

Department of Electrical and Computer Engineering
University of California, Santa Barbara

Abstract

We consider a coarse-grained multiprocessing environment in which multiple task copies *or* unreliable versions *(henceforth, referred to as* task instances*) need to be scheduled to run on unreliable processors in the face of correctness and timeliness requirements that are considered met if c task instances run to correct completion before the deadline d. In this paper, we study the interplay of correctness and timeliness requirements, providing examples of how scheduling policies that are optimal in other contexts can fail to meet correctness and timeliness needs in the above environment. We then present optimal scheduling policies for certain special cases of the above problem followed by a discussion of heuristics with reasonable performance in more general cases.*

ments. In other words, processing power can be used to execute more tasks (meeting more of the deadlines but with a lower level of confidence in the correctness of the results) versus executing tasks more reliably. Aspects of this latter concern are need-based scheduling of multiple task versions [2, 6], adjusting checkpointing intervals to strike a balance between waste of computational resources in the event of a detected fault and waste of the same due to checkpointing overhead [9], and, finally, tradeoffs between precision of results and their timeliness [1, 3].

We endeavor to extend and refine the framework presented in [6] by considering scheduling issues in a coarse-grained multiprocessing environment in which multiple task *copies* or unreliable *versions* (henceforth, referred to as task *instances*) need to be scheduled to run on unreliable processors in the face of correctness

17.5 Some Useful Bounds for Scheduling

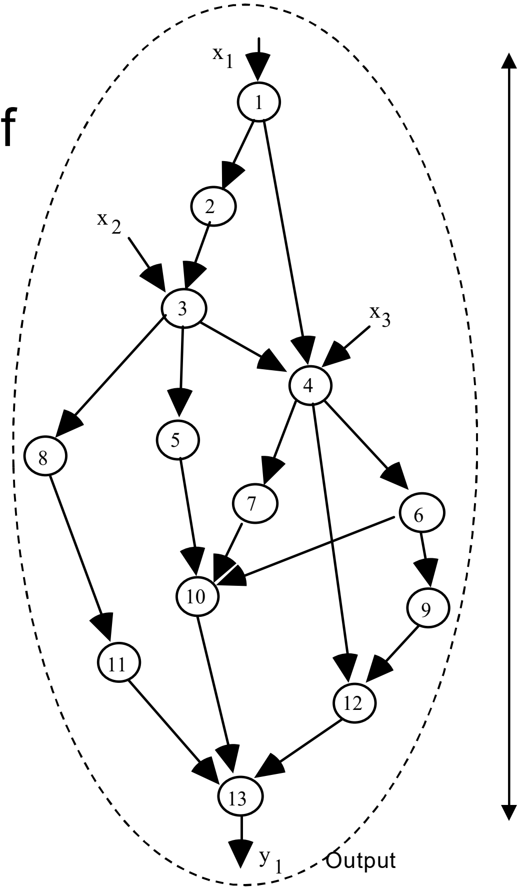
Brent's scheduling theorem: $T_p < T_\infty + T_1/p$

Ideal run time

Ideal speedup

In other words, one can always schedule a task graph so that the total running time does not exceed the best possible (T_1/p) by more than the depth of the task graph (T_∞)

T_1 (Size of task graph)



T_∞ (Depth of task graph)

Lower bound on speedup based on Brent's scheduling theorem:

$S > T_1 / (T_\infty + T_1/p) = p / (1 + pT_\infty / T_1)$

[Compare to Amdahl's law]

A large T_∞ / T_1 ratio indicates much sequential dependency (Amdahl's *f*)

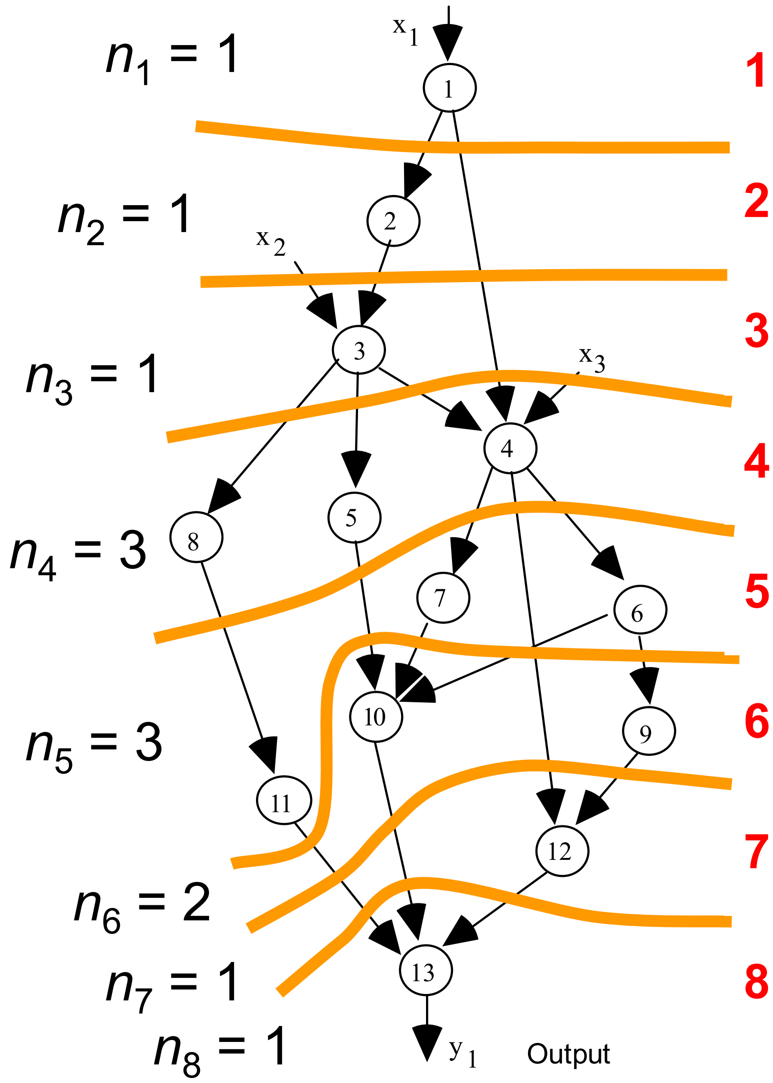
Proof of Brent’s Theorem: $T_p < T_\infty + T_1/p$

First assume the availability of an unlimited number of processors and schedule each node at the earliest possible time

Let there be n_t nodes scheduled at time t; Clearly, $\sum_t n_t = T_1$

With only p processors, tasks scheduled for time step t can be executed in $\lceil n_t / p \rceil$ steps by running them p at a time. Thus:

$$
\begin{aligned}
T_p &\le \sum_{t=1 \text{ to } T_\infty} \lceil n_t / p \rceil \\
&< \sum_{t=1 \text{ to } T_\infty} (n_t / p + 1) \\
&= T_\infty + (\textstyle\sum_t n_t)/p \\
&= T_\infty + T_1/p
\end{aligned}
$$



$n_1 = 1$

$n_2 = 1$

$n_3 = 1$

$n_4 = 3$

$n_5 = 3$

$n_6 = 2$

$n_7 = 1$

$n_8 = 1$

Good-News Corollaries

Brent's scheduling theorem: $T_p < T_\infty + T_1/p$

Ideal run time   Ideal speedup

Corollary 1: For $p \geq T_1/T_\infty$ we have $T_\infty \leq T_p < 2T_\infty$

$T_1/p \leq T_\infty$ For a sufficiently large number p of processors, we can come within a factor of 2 of the best possible run time, even when we use a naïve scheduling algorithm

Corollary 2: For $p \leq T_1/T_\infty$ we have $T_1/p \leq T_p < 2T_1/p$

$T_\infty \leq T_1/p$ If we do not have too many processors, we can come within a factor of 2 of the best possible speedup, even when we use a naïve scheduling algorithm

Choosing $p \cong T_1/T_\infty$ leads to O(p) speedup and near-minimal run time

ABCs of Parallel Processing in One Slide

A Amdahl’s Law (Speedup Formula)

Bad news – Sequential overhead will kill you, because:

$$\text{Speedup} = T_1/T_p \leq 1/[f + (1 - f)/p] \leq \mathit{min}(1/f, p)$$

Morale: For $f = 0.1$, speedup is at best 10, regardless of peak OPS.

B Brent’s Scheduling Theorem

Good news – Optimal scheduling is very difficult, but even a naïve scheduling algorithm can ensure:

$$T_1/p \leq T_p < T_1/p + T_\infty = (T_1/p)[1 + p/(T_1/T_\infty)]$$

Result: For a reasonably parallel task (large T_1/T_∞), or for a suitably small p (say, $p < T_1/T_\infty$), good speedup and efficiency are possible.

C Cost-Effectiveness Adage

Real news – The most cost-effective parallel solution may not be the one with highest peak OPS (communication?), greatest speed-up (at what cost?), or best utilization (hardware busy doing what?).

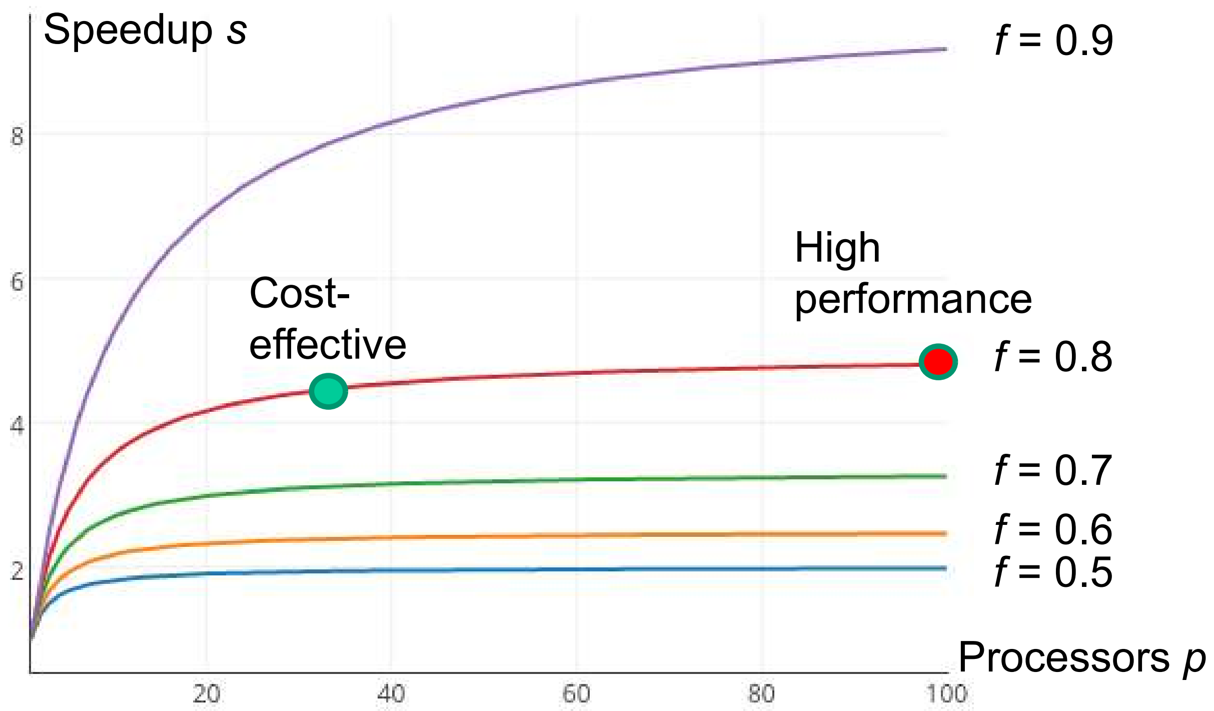
Analogy: Mass transit might be more cost-effective than private cars even if it is slower and leads to many empty seats.

Cost-Effectiveness Adage in Parallel Processing

The most cost-effective parallel solution may not be the one with
- Highest peak OPS (communication?)
- Greatest speed-up (at what cost?)
- Best utilization (hardware busy doing what?)

Why peak ops isn't a good measure
- 100 PFLOPS peak
- 10 PFLOPS sustained

Utilization analogy:
- Bus, 10 riders, 50 seats
- Auto, 2 riders, 5 seats



17.6 Load Balancing and Dataflow Systems

Task running times are not constants
A processor may run out of things to do before others complete their tasks
Some processors may remain idle, waiting to hear about prerequisite tasks
In these cases, a load balancing policy may be applied

Dynamic load balancing: Switching unexecuted tasks from overloaded processors to less loaded ones, as we learn about execution times and task interdependencies at run time

Load balancing can be initiated by a lightly loaded or by an overburdened processor (receiver/sender-initiated)
Unfortunately, load balancing may involve significant overhead
The ultimate in automatic load-balancing is a self-scheduling system that tries to keep all processing resources running at maximum efficiency
There may be a central location to which processors refer for work and where they return their results
An idle processor requests that it be assigned new work by the supervisor
This works nicely for tasks with small contexts or relatively long run times

Dataflow Systems

Computation represented by a dataflow graph

Tokens used to keep track of data availability

Once tokens appear on all inputs, node is “fired,” resulting in tokens being removed from its inputs and put on each output

Static dataflow: No more than one token on edge

Dynamic dataflow: Multiple tagged tokens on edges; “consumed” after matching their tags

Hardware-level implementation of a self-scheduling scheme

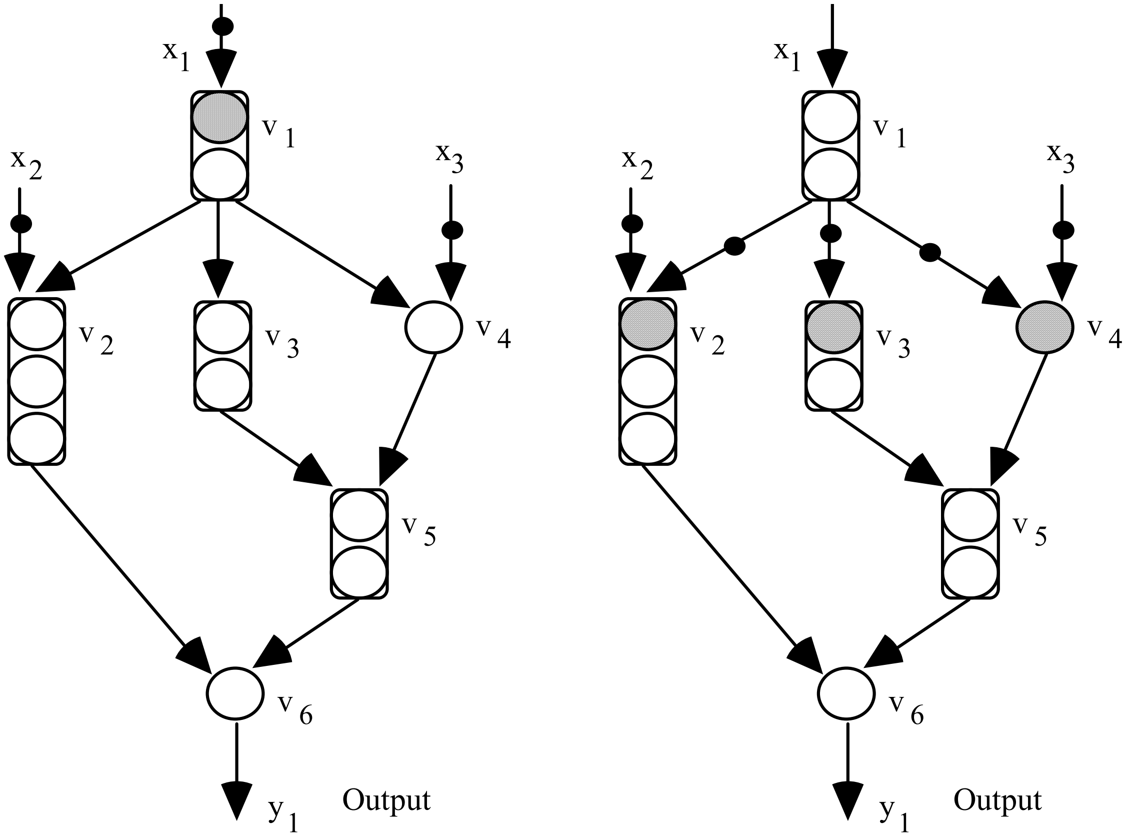


Fig. 17.9 Example dataflow graph with token distribution at the outset (left) and after 2 time units (right).

18 Data Storage, Input, and Output

Elaborate on problems of data distribution, caching, and I/O:

- Deal with speed gap between processor and memory
- Learn about parallel input and output technologies

Topics in This Chapter	
18.1	Data Access Problems and Caching
18.2	Cache Coherence Protocols
18.3	Multithreading and Latency Hiding
18.4	Parallel I/O Technology
18.5	Redundant Disk Arrays
18.6	Interfaces and Standards

18.1 Data Access Problems and Caching

Processor-memory speed gap is aggravated by parallelism
Centralized memory is slower; distributed memory needs remote accesses

Remedies: Judicious data distribution –- good with static data sets
Data caching –- introduces coherence problems
Latency tolerance/hiding –- e.g., via multithreading

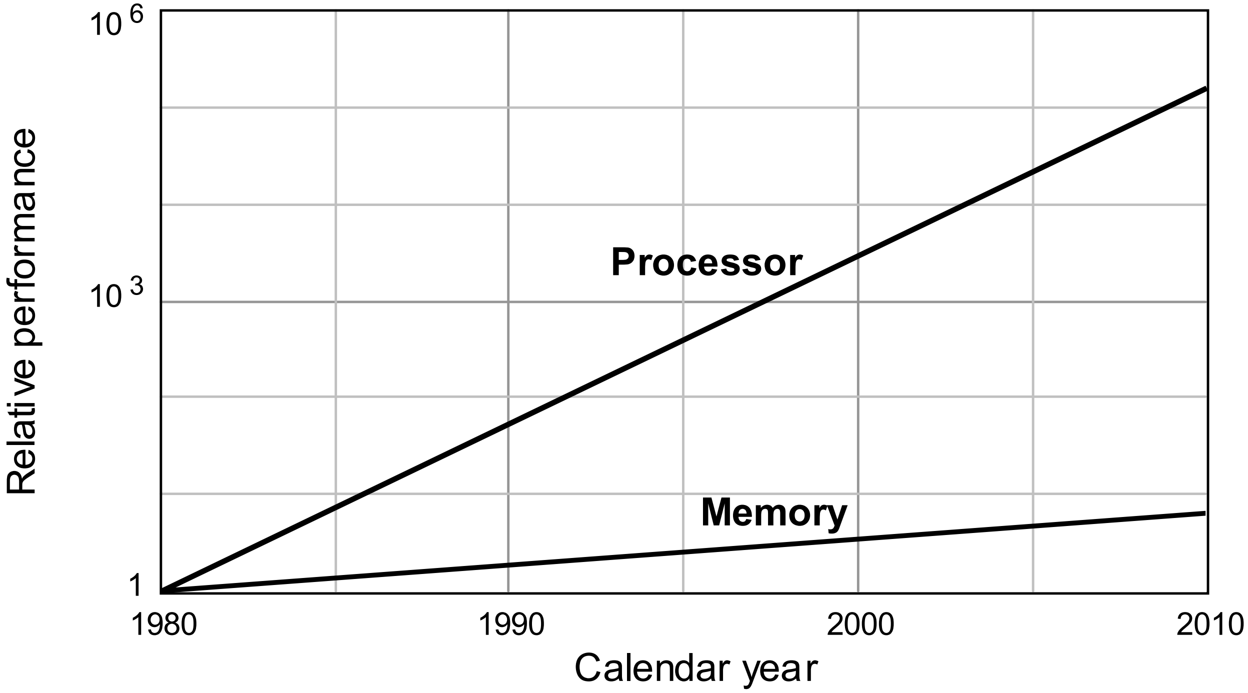


Fig. 17.8 of Parhami's Computer Architecture text (2005)

Why Data Caching Works

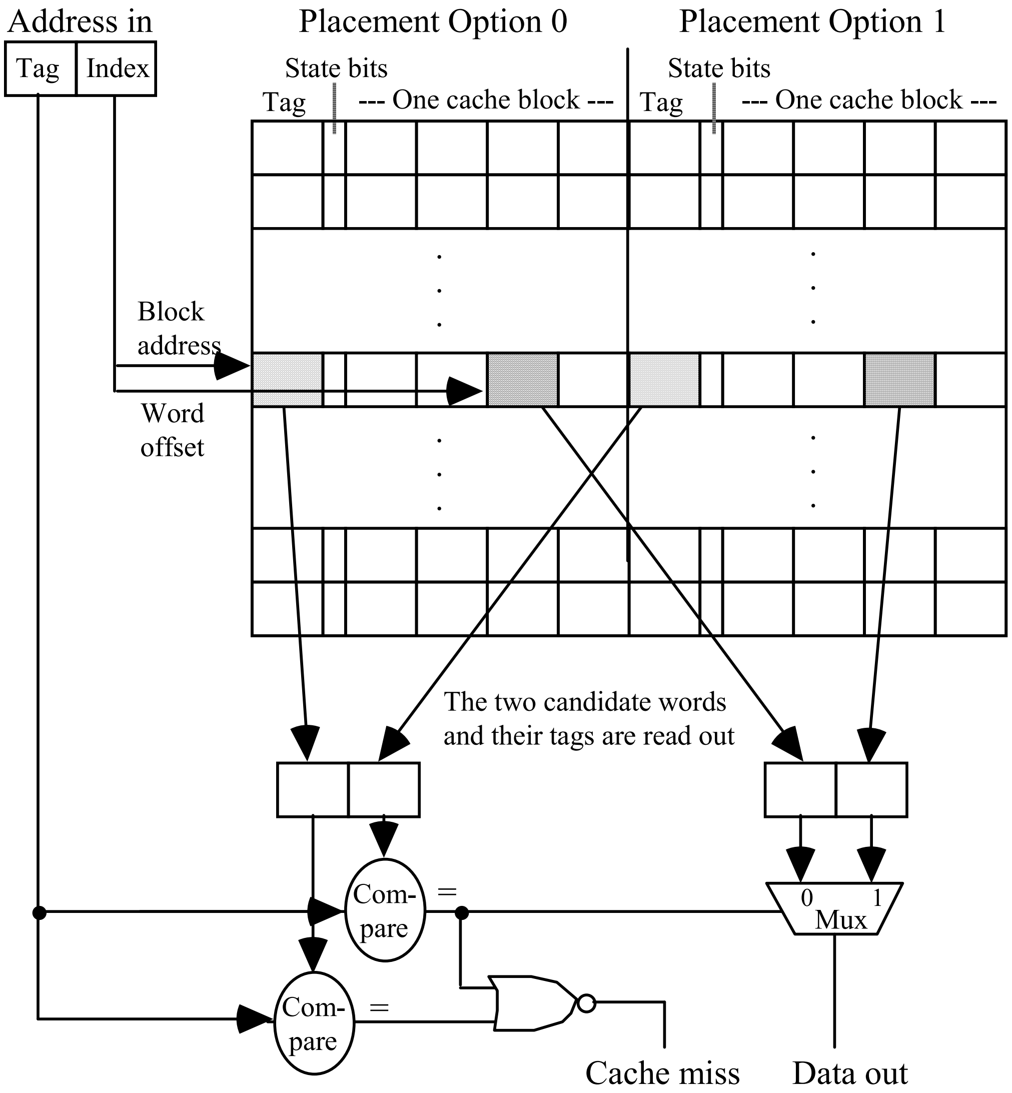
Hit rate r (fraction of memory accesses satisfied by cache)

$$C_{\text{eff}} = C_{\text{fast}} + (1 - r)C_{\text{slow}}$$

Cache parameters:

Size
Block length (line width)
Placement policy
Replacement policy
Write policy

Fig. 18.1 Data storage and access in a two-way set-associative cache.



Benefits of Caching Formulated as Amdahl's Law

Hit rate *r* (fraction of memory accesses satisfied by cache)

$$C_{\text{eff}} = C_{\text{fast}} + (1 - r)C_{\text{slow}}$$

$$S \;= C_{\text{slow}} / C_{\text{eff}}$$

$$= \frac{1}{(1 - r) + C_{\text{fast}}/C_{\text{slow}}}$$

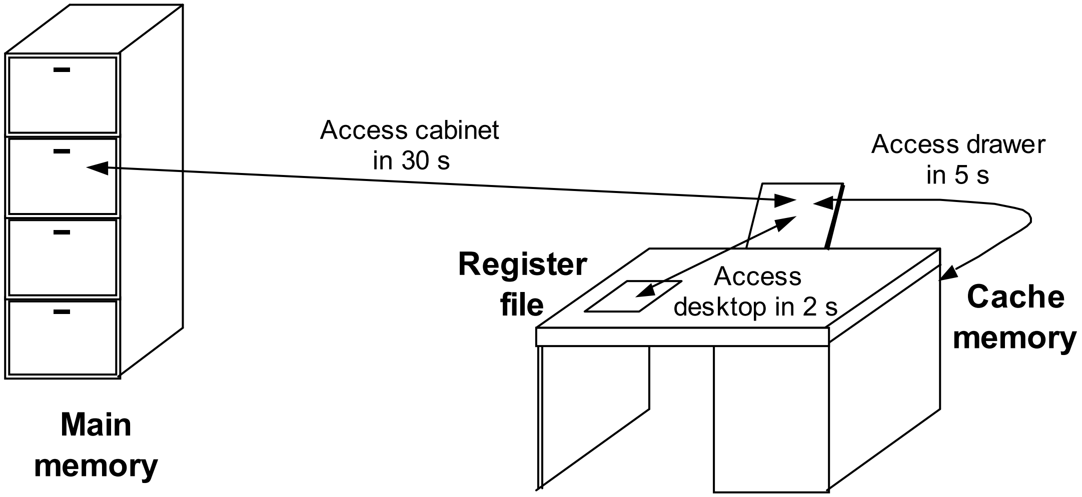


Fig. 18.3 of Parhami's Computer Architecture text (2005)

This corresponds to the miss-rate fraction 1 – *r* of accesses being unaffected and the hit-rate fraction *r* (almost 1) being speeded up by a factor $C_{\text{slow}}/C_{\text{fast}}$

Generalized form of Amdahl's speedup formula:

$$S \;=\; 1/(f_1/p_1 + f_2/p_2 + \ldots + f_m/p_m), \quad \text{with } f_1 + f_2 + \ldots + f_m = 1$$

In this case, a fraction 1 – *r* is slowed down by a factor $(C_{\text{slow}} + C_{\text{fast}}) / C_{\text{slow}}$, and a fraction *r* is speeded up by a factor $C_{\text{slow}} / C_{\text{fast}}$

18.2 Cache Coherence Protocols

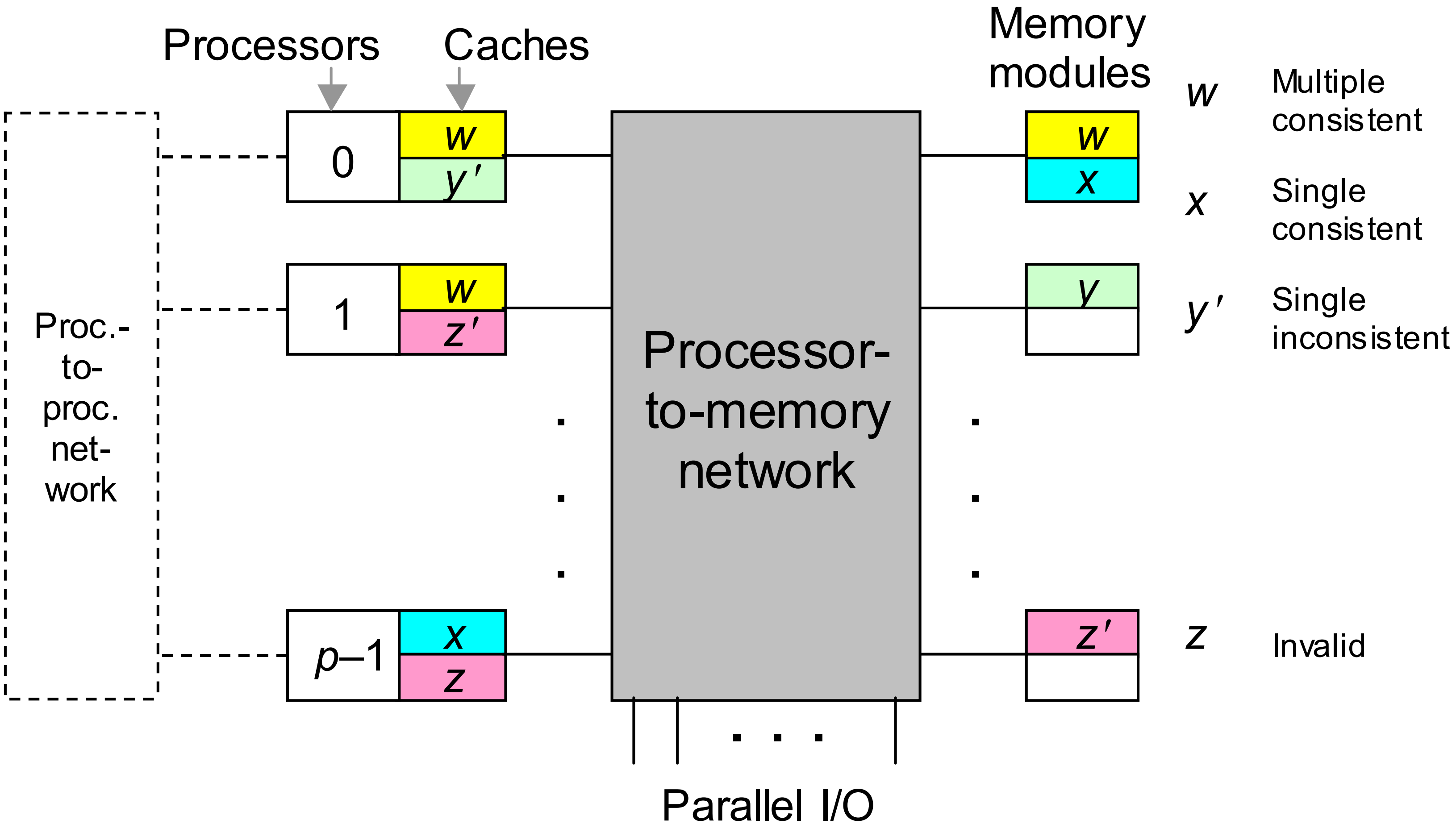


Fig. 18.2 Various types of cached data blocks in a parallel processor with global memory and processor caches.

Example: A Bus-Based Snoopy Protocol

Each transition is labeled with the event that triggers it, followed by the action(s) that must be taken

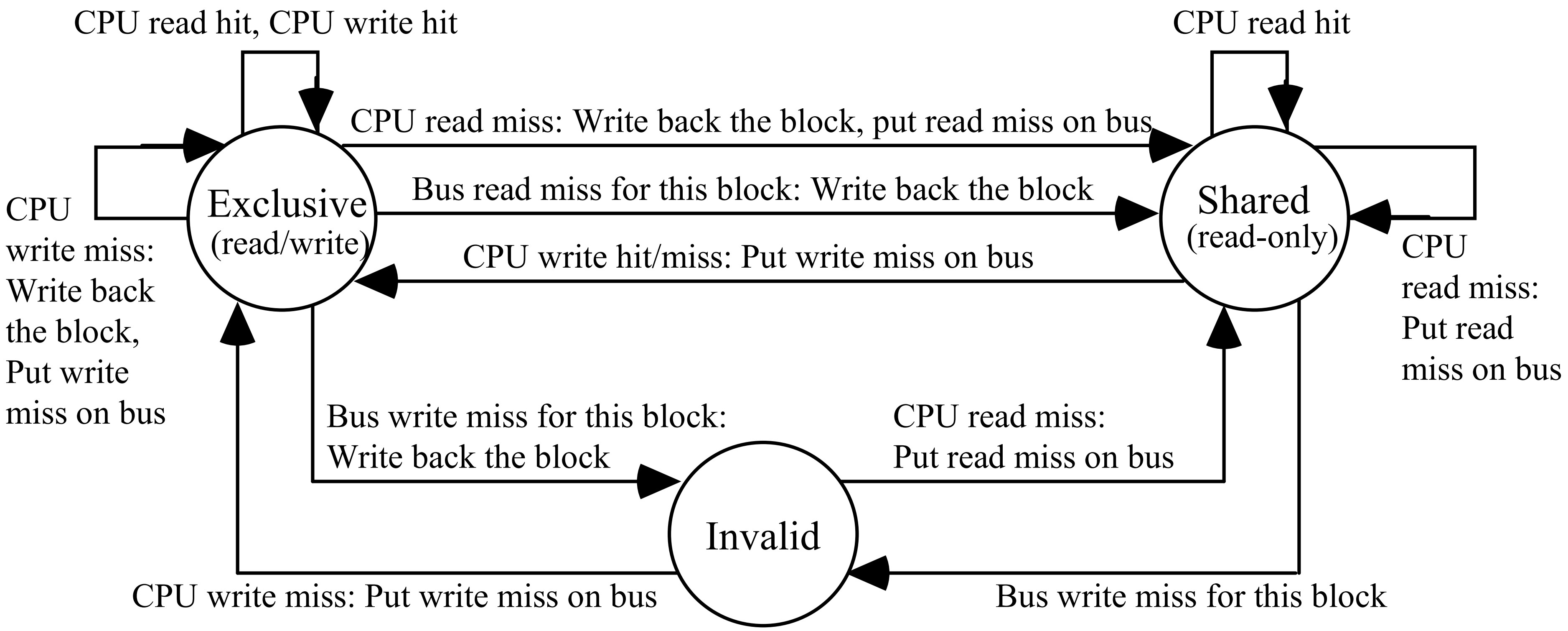

CPU read hit, CPU write hit
CPU read hit
CPU read miss: Write back the block, put read miss on bus
Exclusive
(read/write)
Bus read miss for this block: Write back the block
Shared
(read-only)
CPU write hit/miss: Put write miss on bus
CPU
write miss:
Write back
the block,
Put write
miss on bus
CPU
read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

Fig. 18.3 Finite-state control mechanism for a bus-based snoopy cache coherence protocol.

Implementing a Snoopy Protocol

A second tags/state storage unit allows snooping to be done concurrently with normal cache operation

Getting all the implementation timing and details right is nontrivial

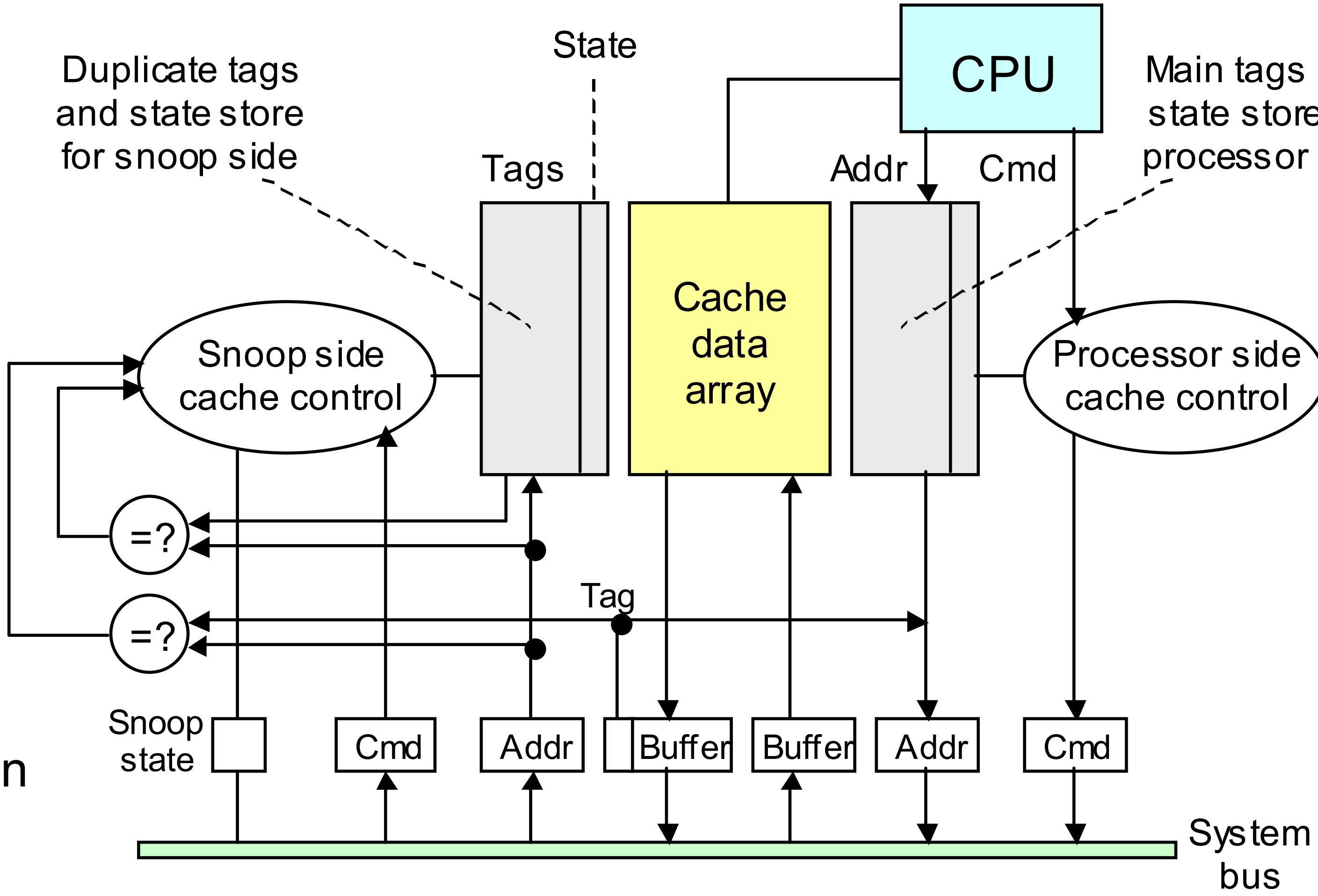

State
Duplicate tags and state store for snoop side
Tags
CPU
Addr
Cmd
Main tags and state store for processor side
Cache data array
Snoop side cache control
Processor side cache control
=?
=?
Tag
Snoop state
Cmd
Addr
Buffer
Buffer
Addr
Cmd
System bus

Fig. 27.7 of Parhami’s Computer Architecture text.

Example: A Directory-Based Protocol

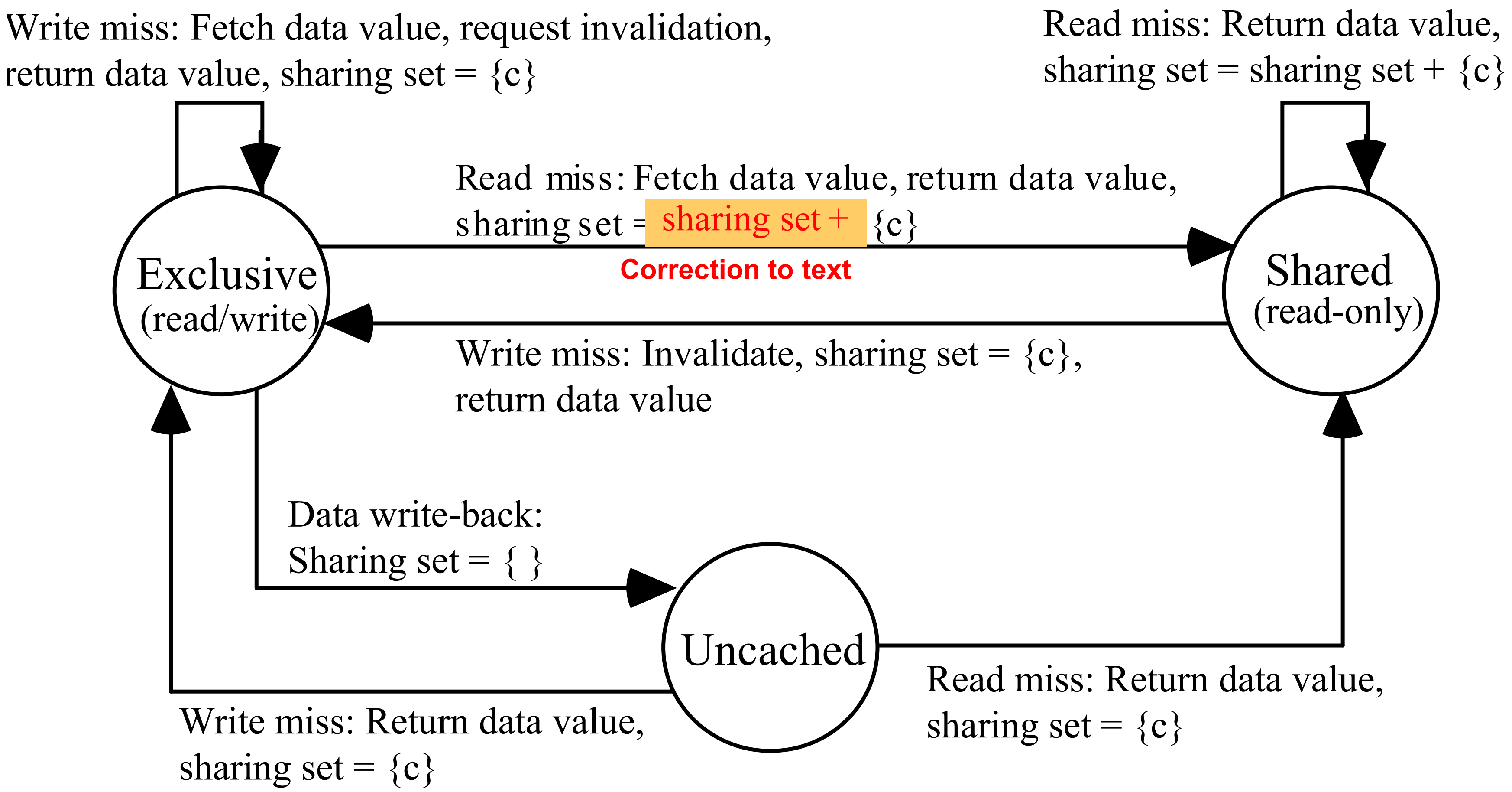

Write miss: Fetch data value, request invalidation, return data value, sharing set = {c}
Read miss: Return data value, sharing set = sharing set + {c}
Read miss: Fetch data value, return data value, sharing set = sharing set + {c}
Correction to text
Exclusive (read/write)
Shared (read-only)
Write miss: Invalidate, sharing set = {c}, return data value
Data write-back: Sharing set = { }
Uncached
Write miss: Return data value, sharing set = {c}
Read miss: Return data value, sharing set = {c}

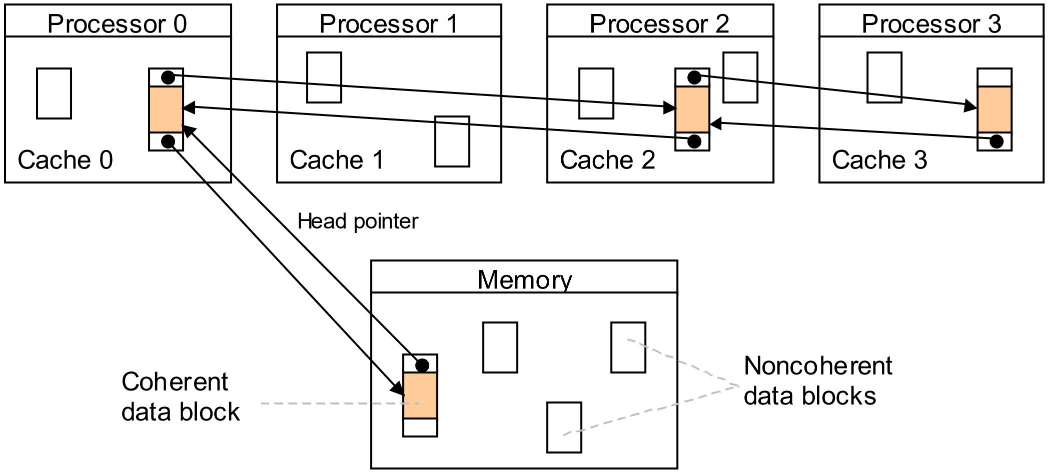
Fig. 18.4 States and transitions for a directory entry in a directory-based coherence protocol (*c* denotes the cache sending the message).

Implementing a Directory-Based Protocol

0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0

Sharing set implemented as a bit-vector (simple, but not scalable)

When there are many more nodes (caches) than the typical size of a sharing set, a list of sharing units may be maintained in the directory

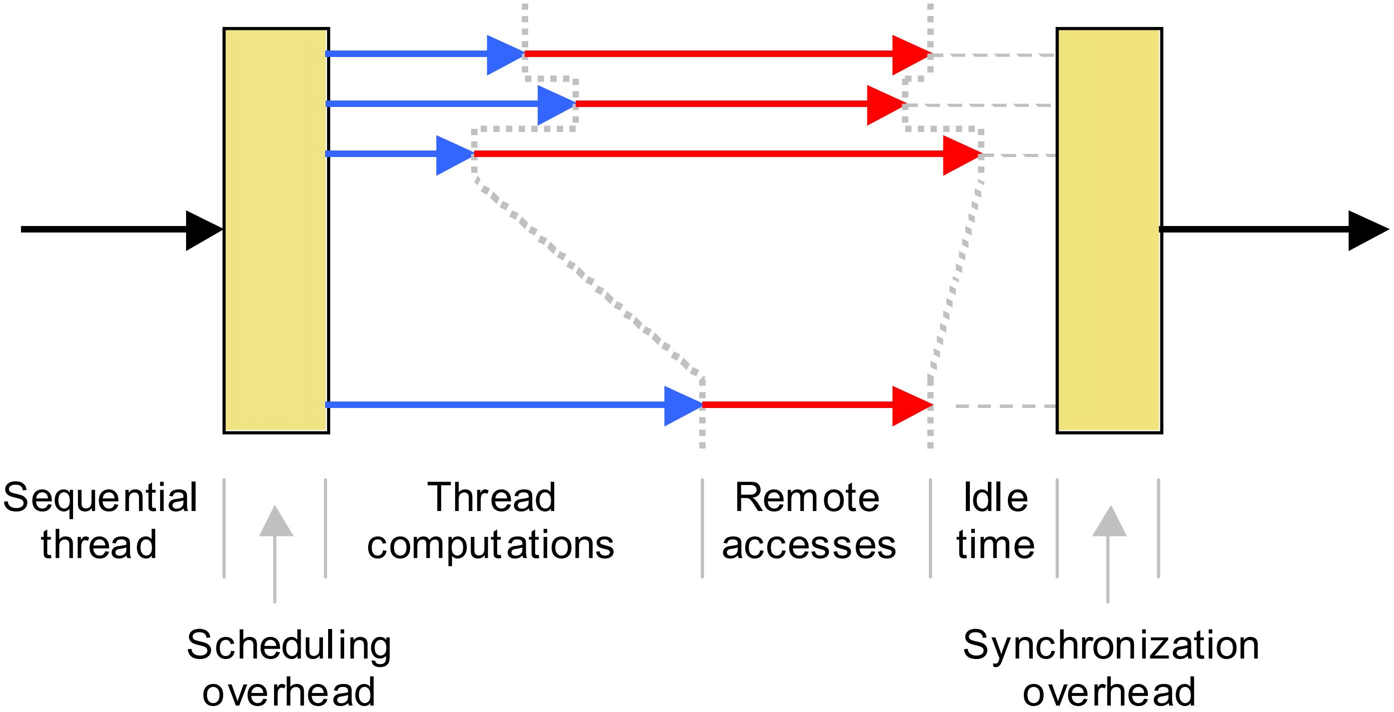


The sharing set can be maintained as a distributed doubly linked list (will discuss in Section 18.6 in connection with the SCI standard)

18.3 Multithreading and Latency Hiding

Latency hiding: Provide each processor with useful work to do as it awaits the completion of memory access requests

Multithreading is one way to implement latency hiding



Sequential thread

Thread computations

Remote accesses

Idle time

Scheduling overhead

Synchronization overhead

Fig. 18.5 The concept of multithreaded parallel computation.

Multithreading on a Single Processor

Here, the motivation is to reduce the performance impact of data dependencies and branch misprediction penalty

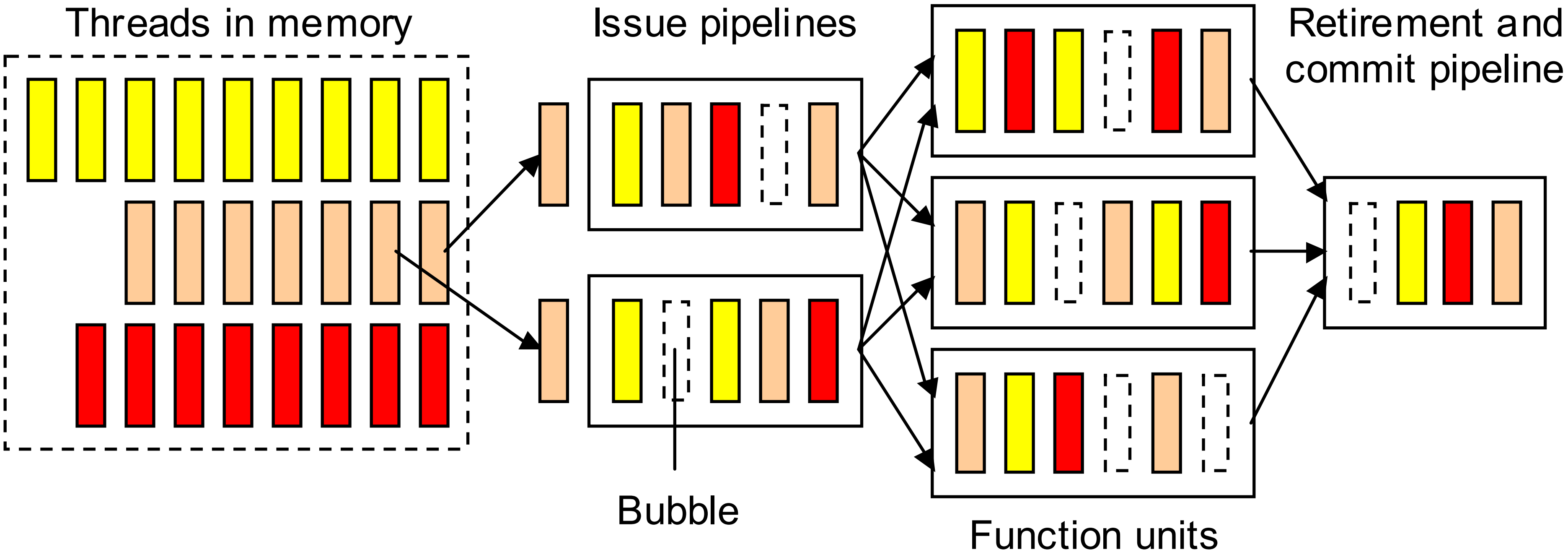


Fig. 24.9 of Parhami's Computer Architecture text (2005)

18.4 Parallel I/O Technology

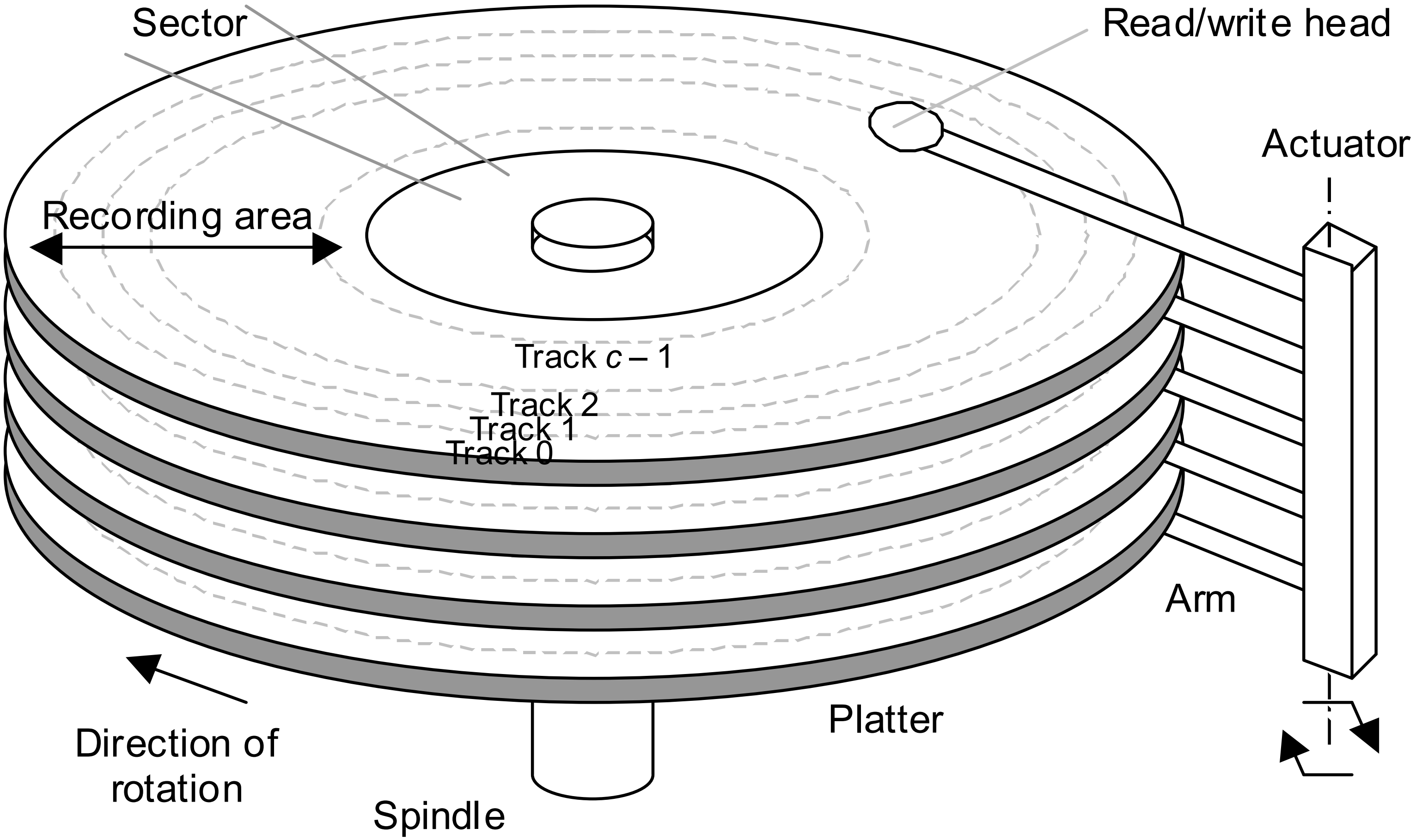


Fig. 18.6 Moving-head magnetic disk elements.

Comprehensive info about disk memory: http://www.storageview.com/guide/

Access Time for a Disk

Data transfer time = Bytes / Data rate

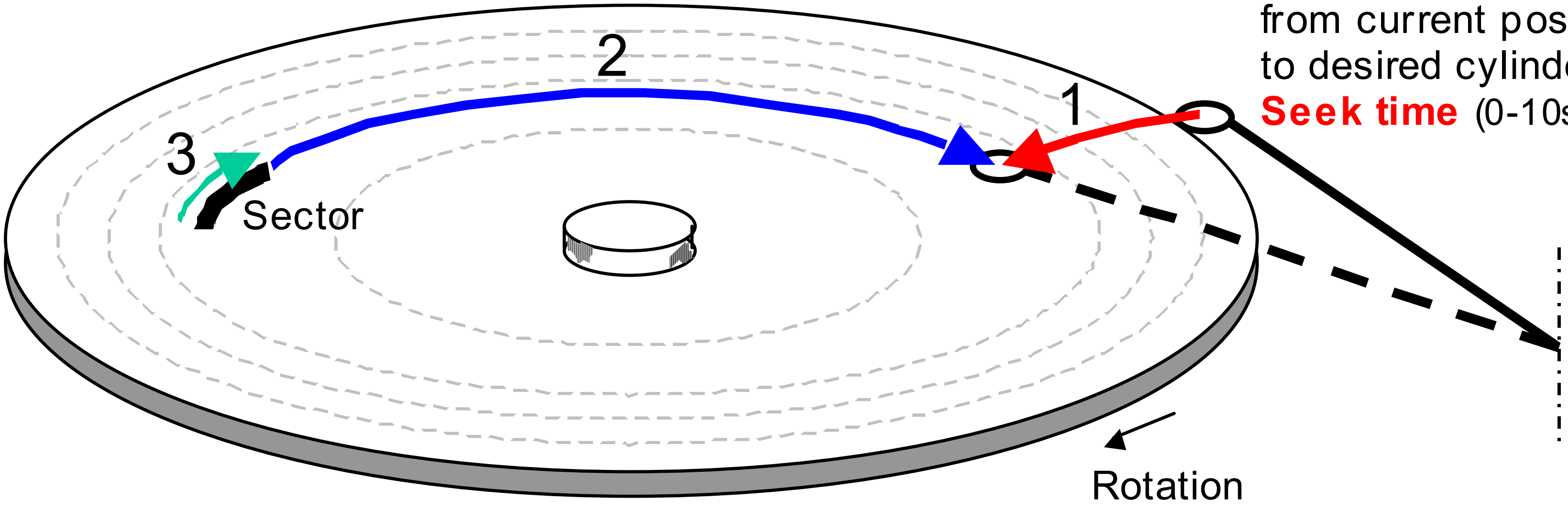
Average rotational latency = 30 000 / rpm (in ms)

Seek time = $a + b(c - 1) + \beta(c - 1)^{1/2}$

3. Disk rotation until sector has passed under the head: **Data transfer time** (< 1 ms)

2. Disk rotation until the desired sector arrives under the head: **Rotational latency** (0-10s ms)

1. Head movement from current position to desired cylinder: **Seek time** (0-10s ms)

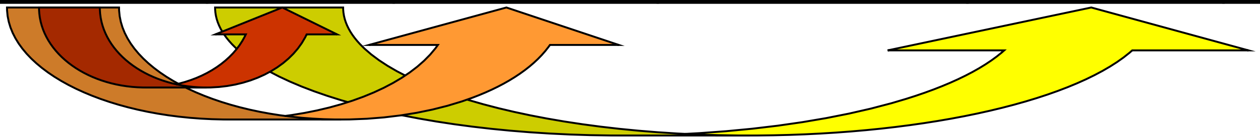


The three components of disk access time. Disks that spin faster have a shorter average and worst-case access time.

Amdahl’s Rules of Thumb for System Balance

The need for high-capacity, high-throughput secondary (disk) memory

Processor speed	RAM size	Disk I/O rate	Number of disks	Disk capacity	Number of disks
1 GIPS	1 GB	100 MB/s	1	100 GB	1
1 TIPS	1 TB	100 GB/s	1000	100 TB	100
1 PIPS	1 PB	100 TB/s	1 Million	100 PB	100 000
1 EIPS	1 EB	100 PB/s	1 Billion	100 EB	100 Million



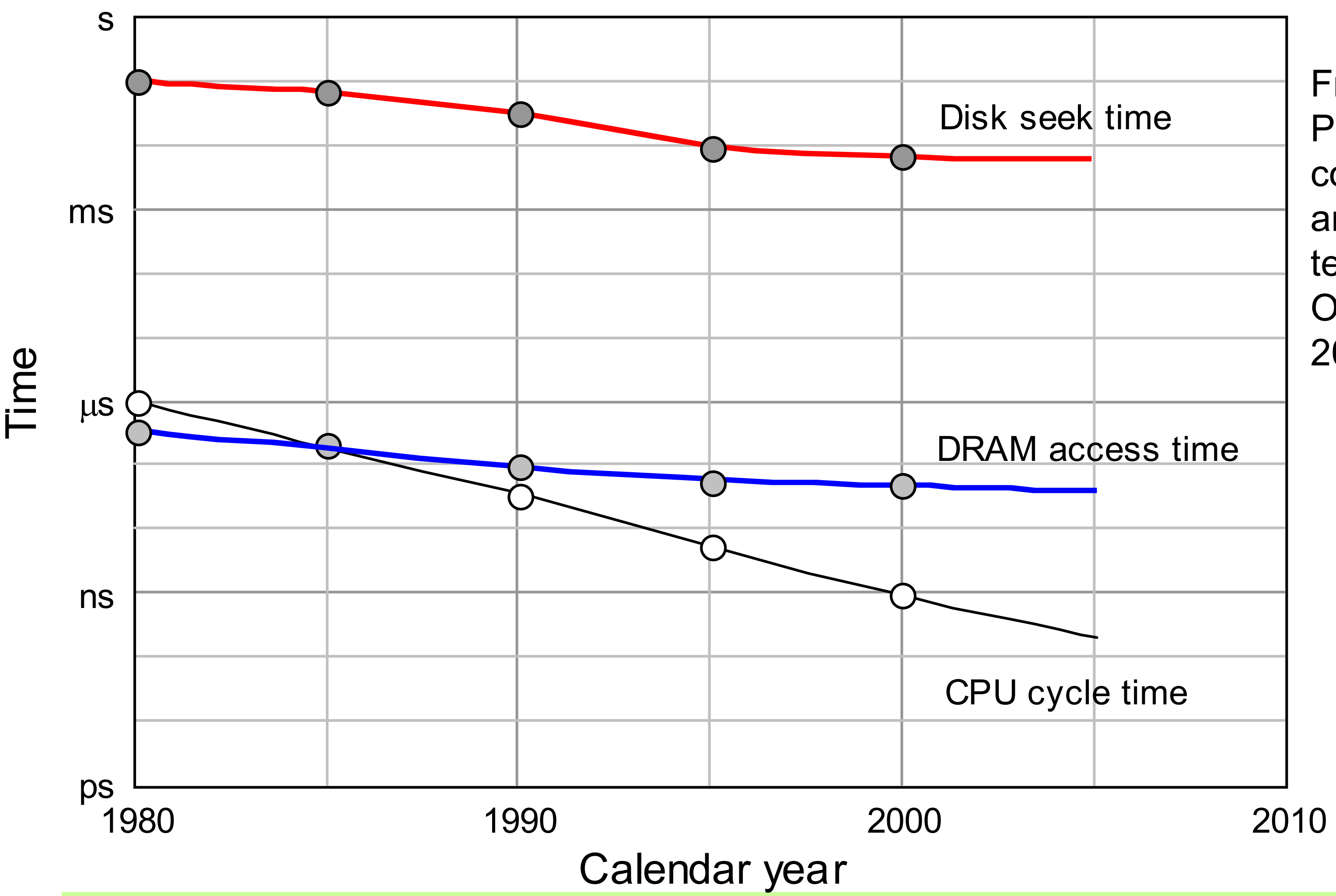
1 RAM byte
for each IPS

1 I/O bit per sec
for each IPS

100 disk bytes
for each RAM byte

G	Giga
T	Tera
P	Peta
E	Exa

Growing Gap Between Disk and CPU Performance



From Parhami's computer architecture textbook, Oxford, 2005

Fig. 20.11   Trends in disk, main memory, and CPU speeds.

Head-Per-Track Disks

Dedicated track heads eliminate seek time (replace it with activation time for a head)

Multiple sets of head reduce rotational latency

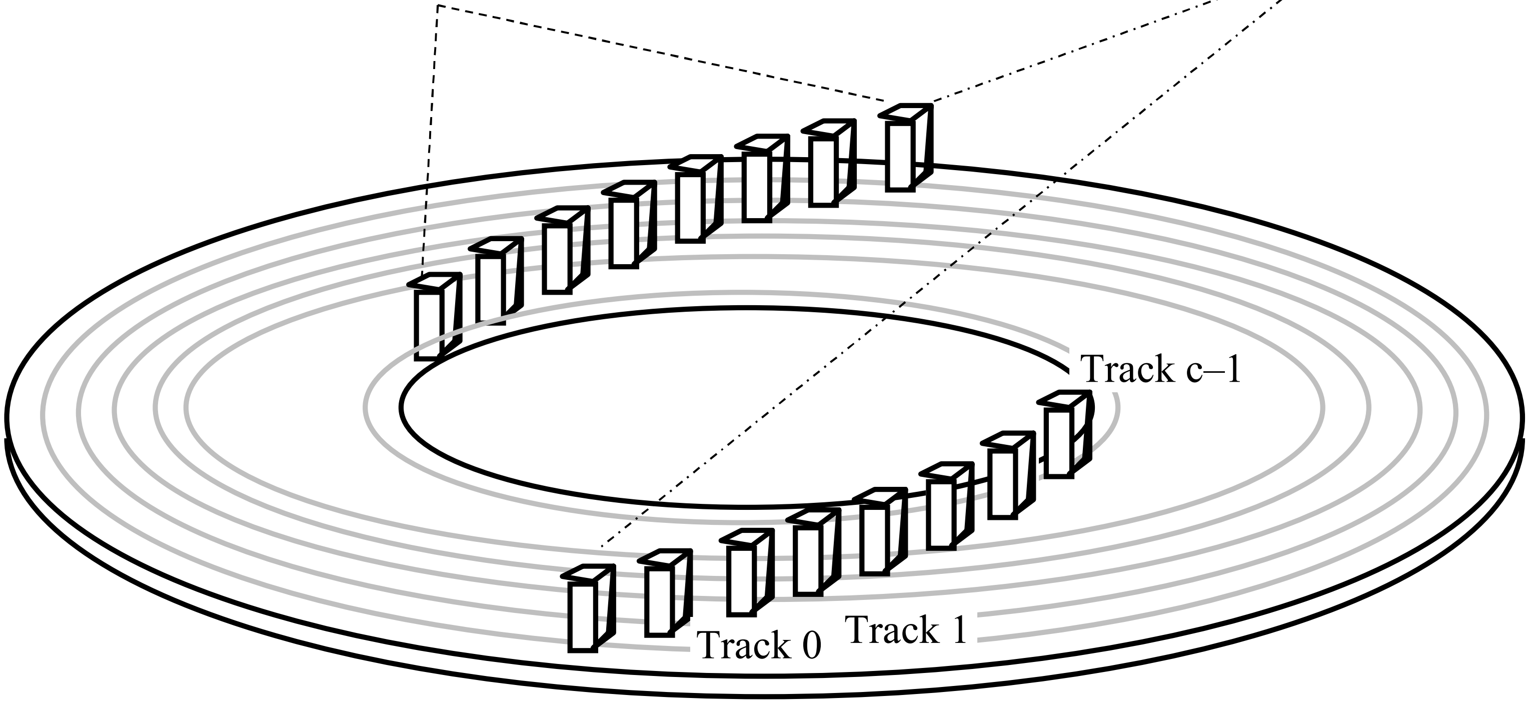


Fig. 18.7 Head-per-track disk concept.

18.5 Redundant Disk Arrays

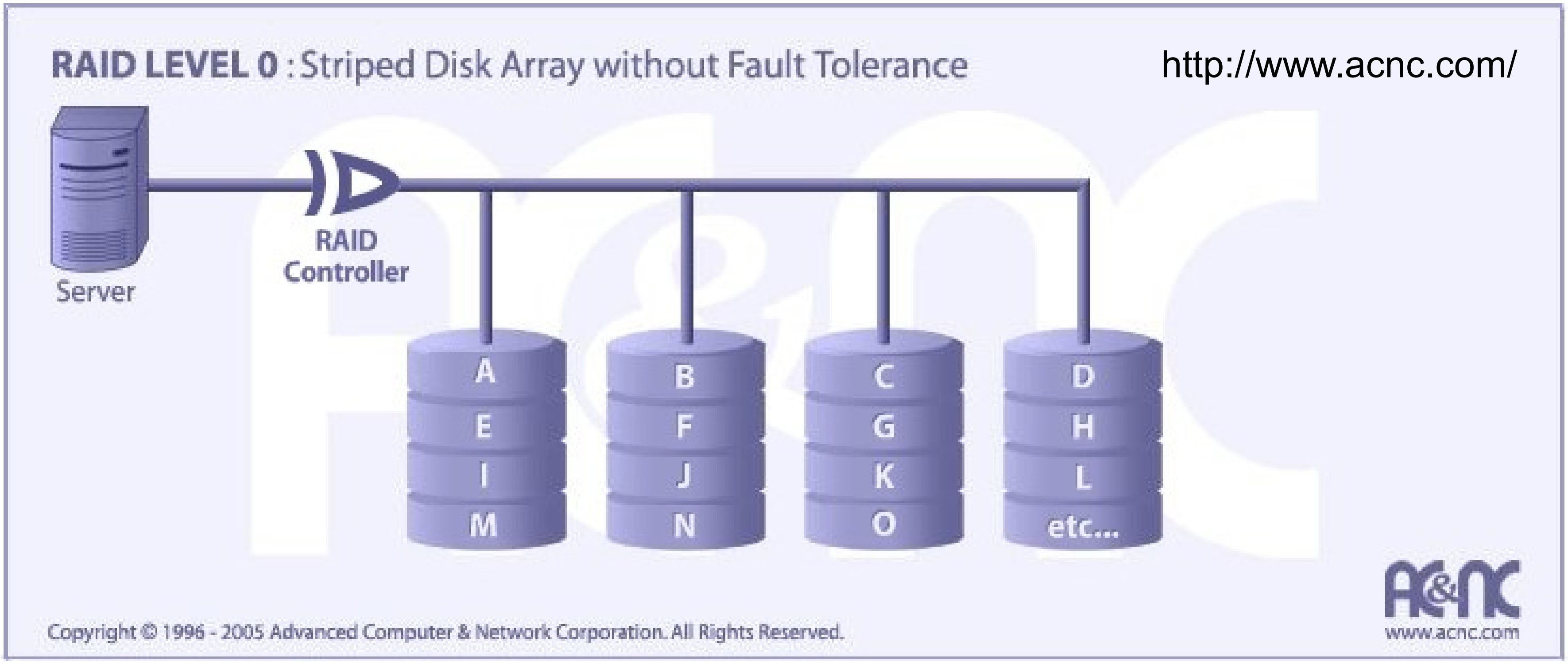
High capacity
(many disks)

High reliability
(redundant data, back-up disks)

High bandwidth
(parallel accesses)

IBM ESS Model 750

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.
AC&NC
www.acnc.com

Structure:

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

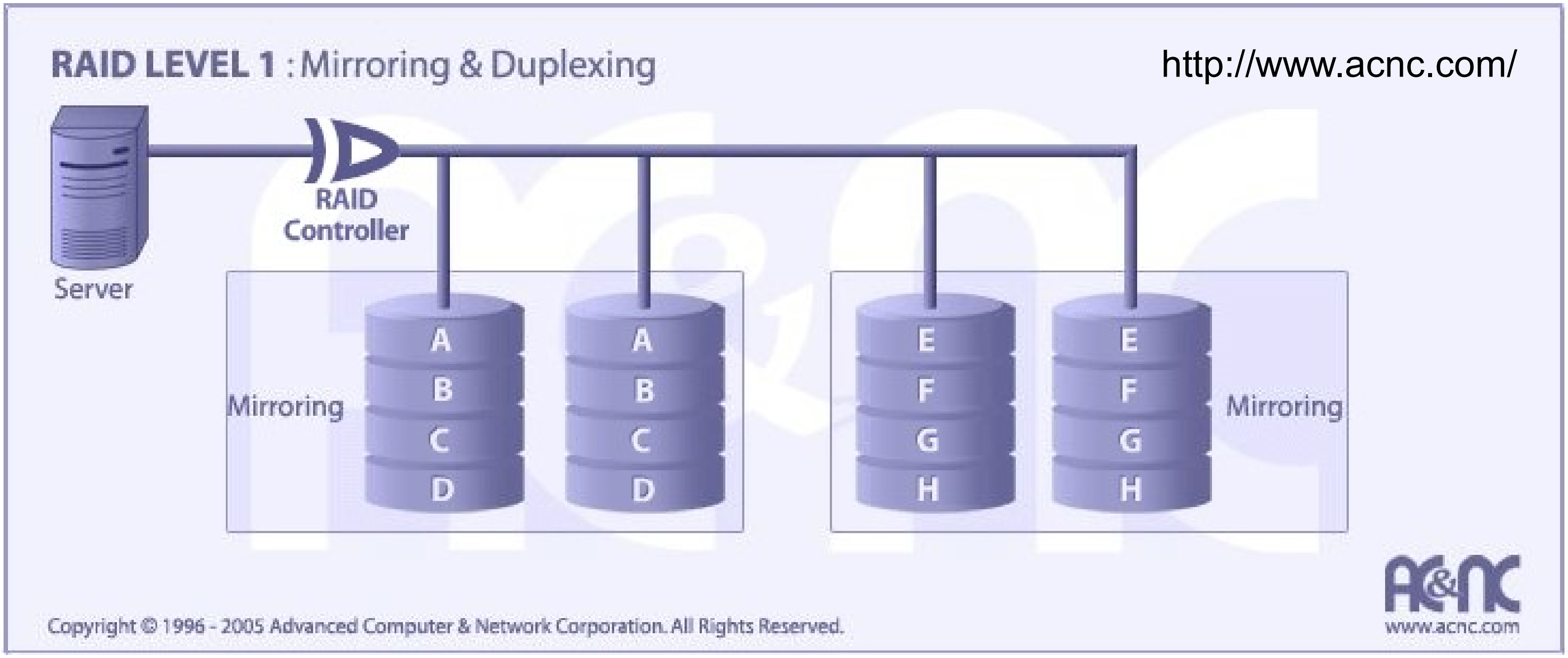
Advantages:

Spreads I/O load across many channels and drives

Drawbacks:

No fault tolerance (data lost with single disk failure)

RAID Level 1



Structure:

Each disk replaced by a mirrored pair

Advantages:

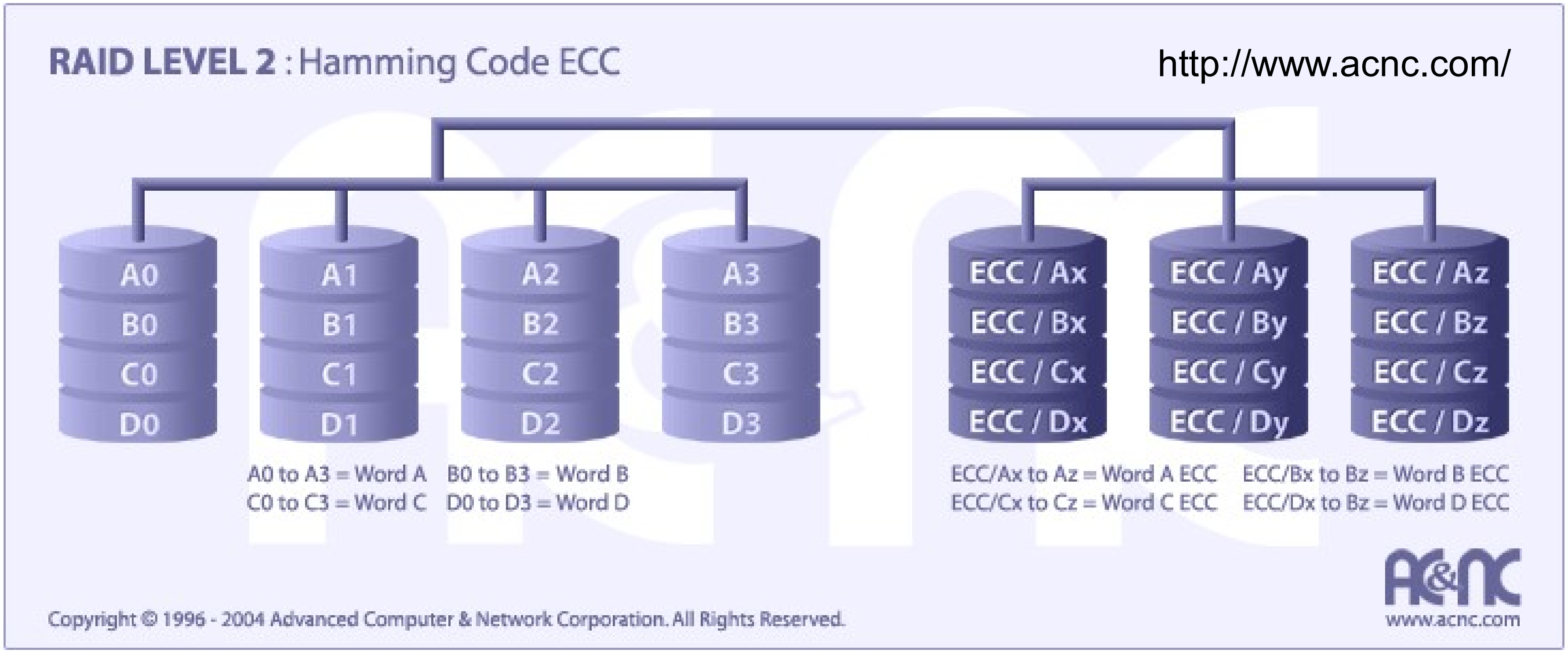
Can double the read transaction rate

No rebuild required

Drawbacks:

Overhead is 100%

RAID Level 2



Structure:

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

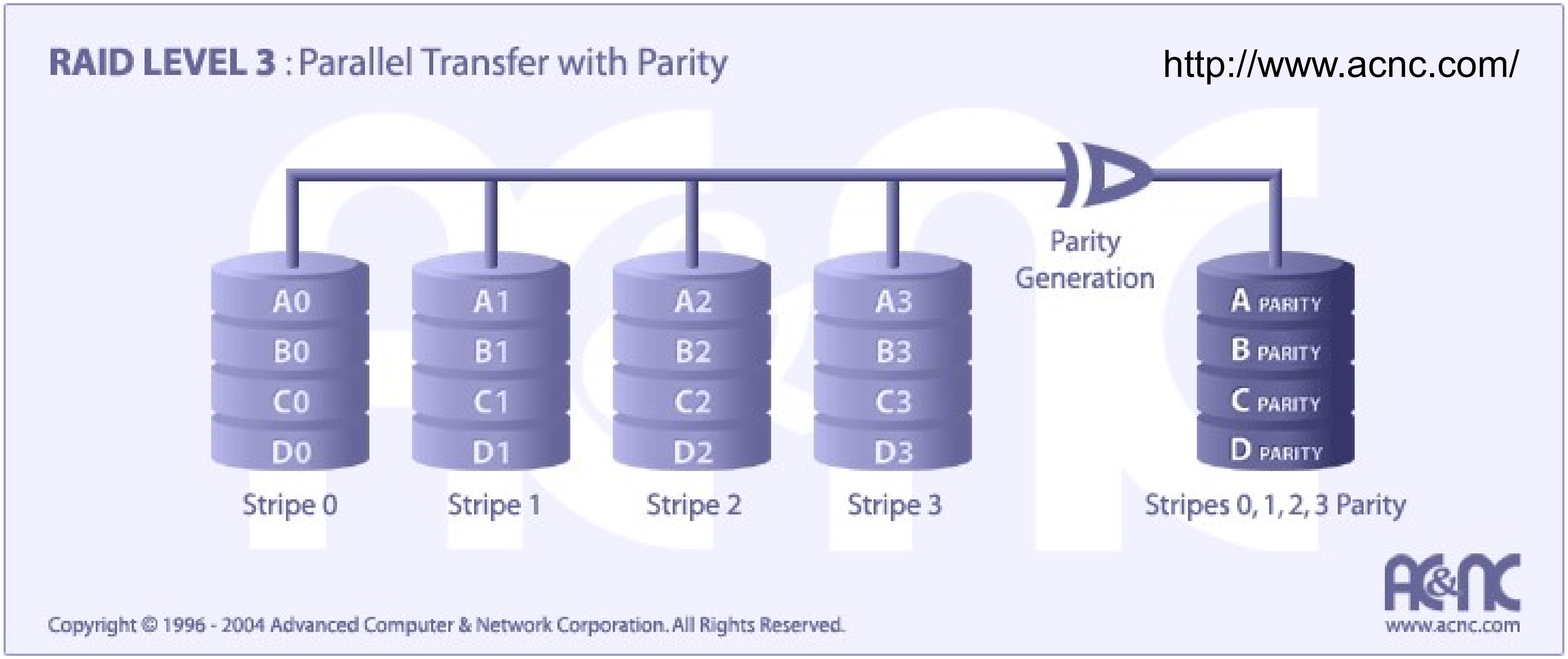
Advantages:

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

Drawbacks:

Potentially high
redundancy
High entry-level cost

RAID Level 3



Structure:

Data striped across several disks, parity provided on another

Advantages:

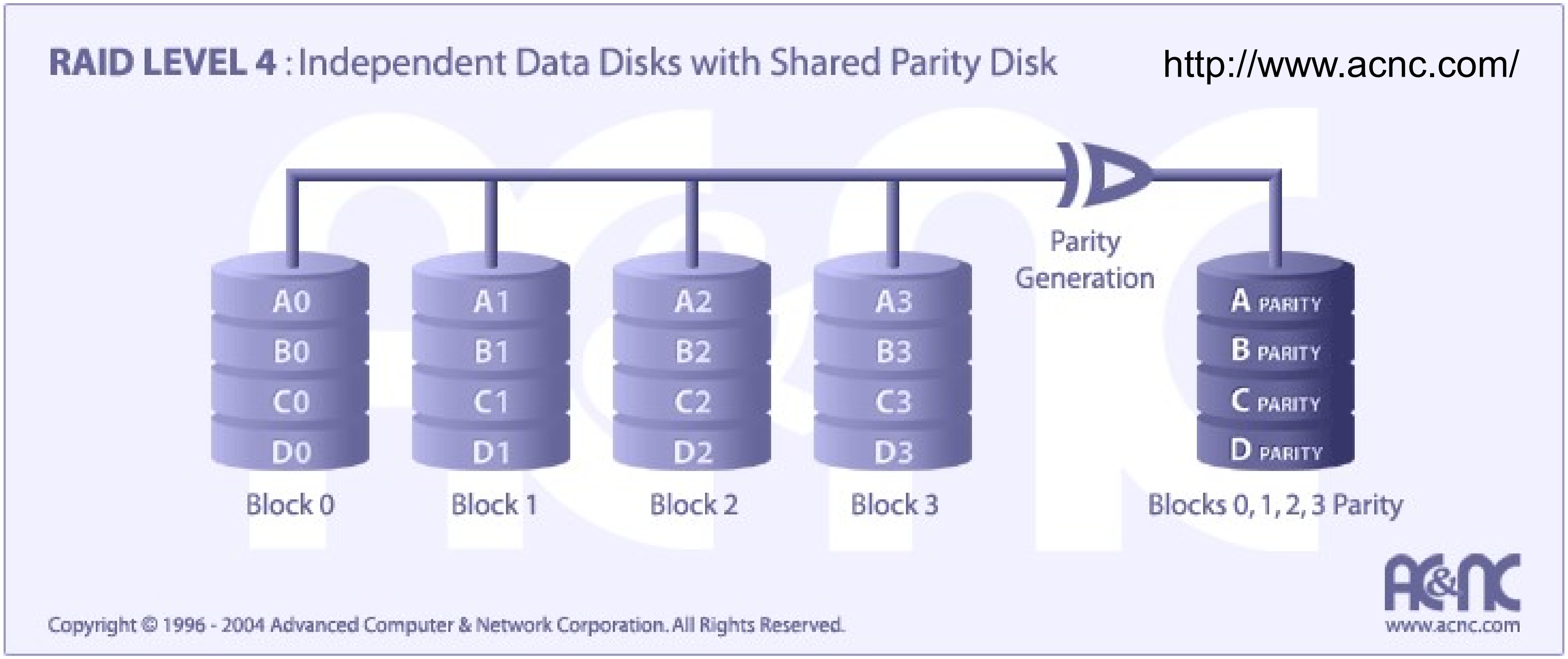
Maintains good throughput even when a disk fails

Drawbacks:

Parity disk forms a bottleneck

Complex controller

RAID Level 4



Structure:

Independent blocks on multiple disks share a parity disk

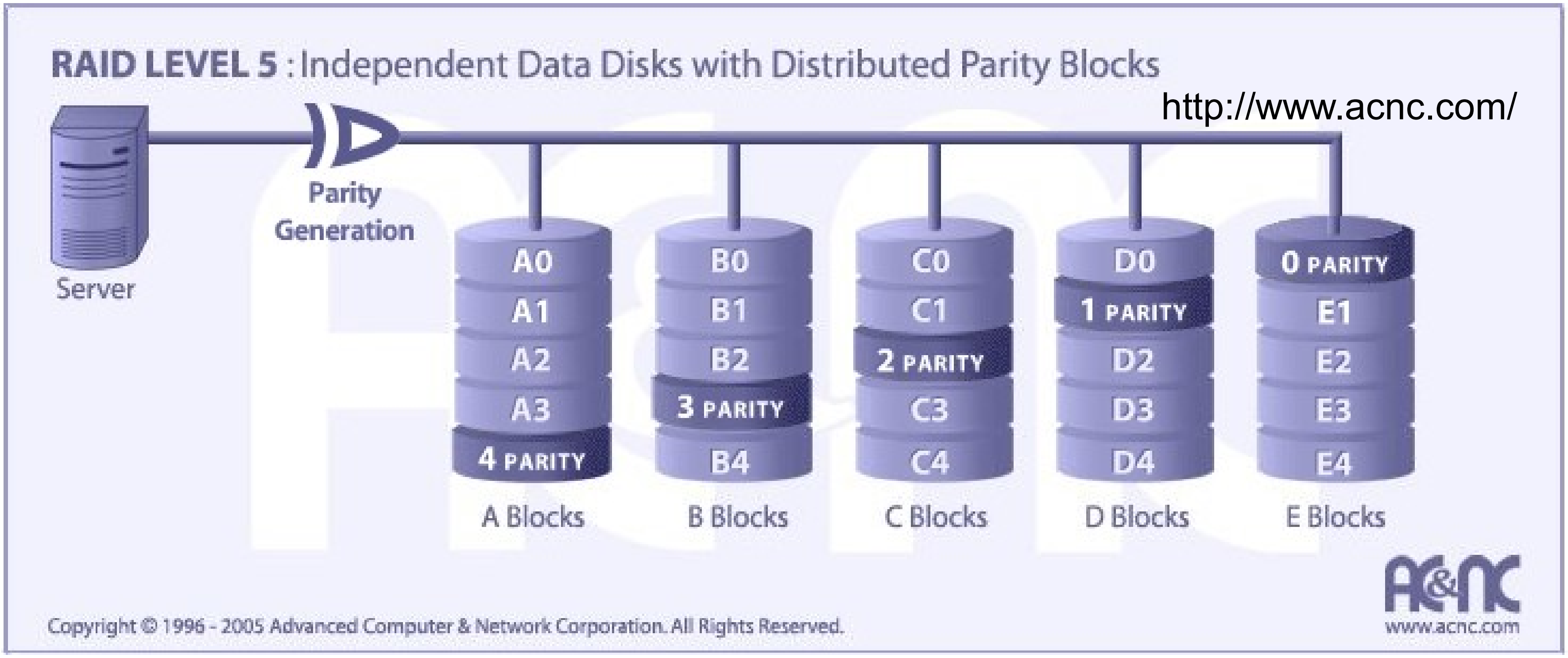
Advantages:

Very high read rate
Low redundancy

Drawbacks:

Low write rate
Inefficient data rebuild

RAID Level 5



Structure:

Parity and data blocks distributed on multiple disks

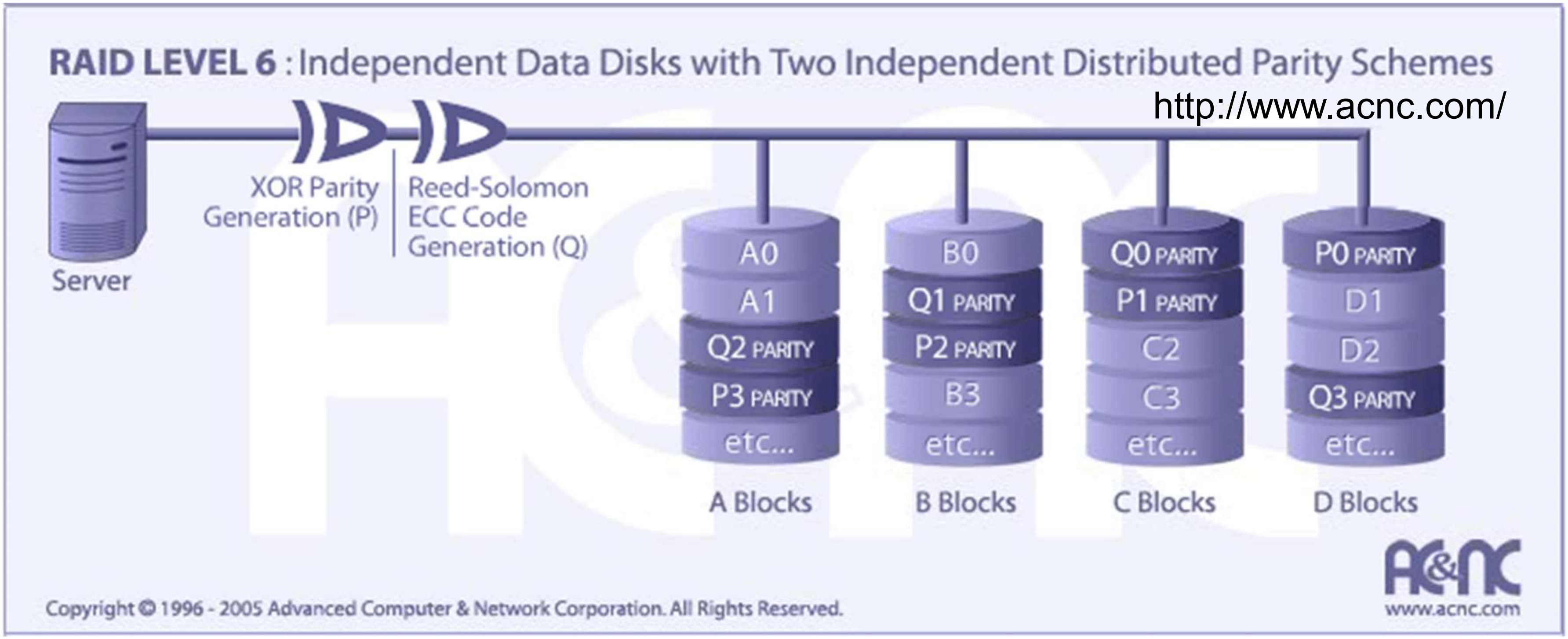
Advantages:

Very high read rate
Medium write rate
Low redundancy

Drawbacks:

Complex controller
Difficult rebuild

RAID Level 6



http://www.acnc.com/

Structure:

RAID Level 5, extended with second parity check scheme

Advantages:

Tolerates 2 failures

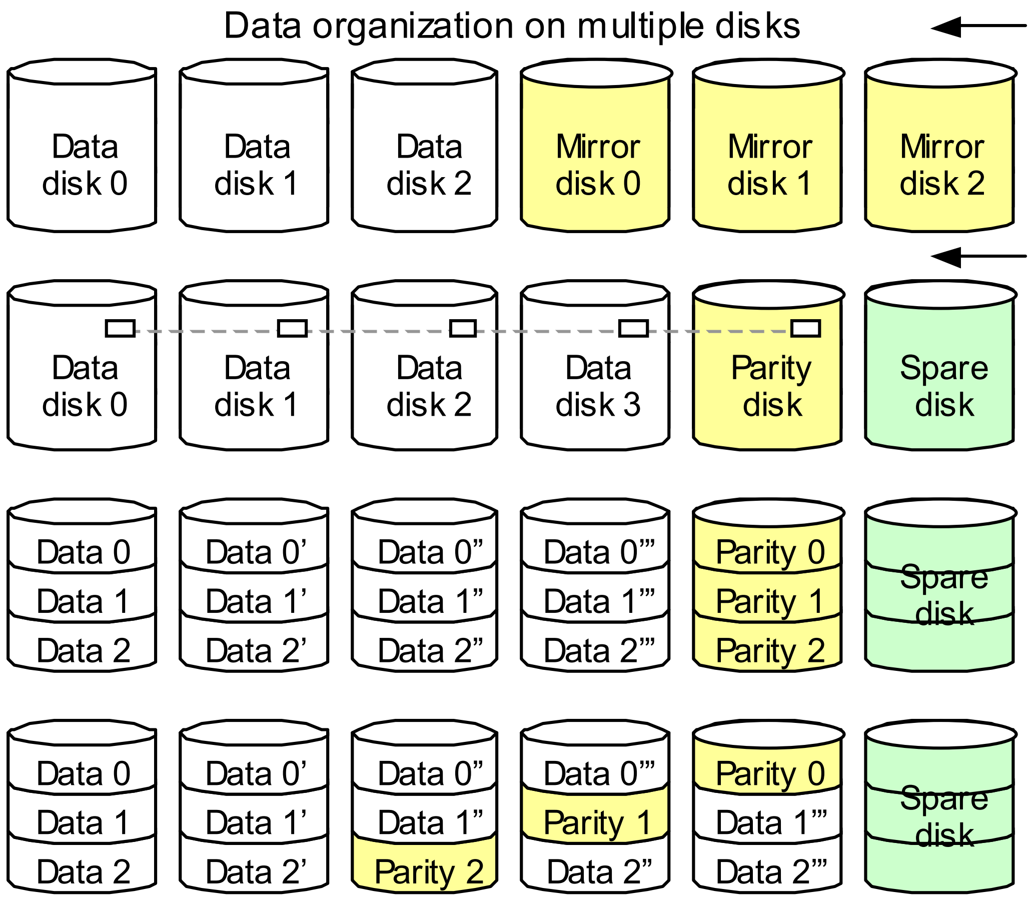
Protected even during recovery

Drawbacks:

More complex controller

Greater overhead

RAID Summary



RAID0: Multiple disks for higher data rate; no redundancy

RAID1: Mirrored disks

RAID2: Error-correcting code

RAID3: Bit- or byte-level striping with parity/checksum disk

RAID4: Parity/checksum applied to sectors,not bits or bytes

RAID5: Parity/checksum distributed across several disks

RAID6: Parity and 2nd check distributed across several disks

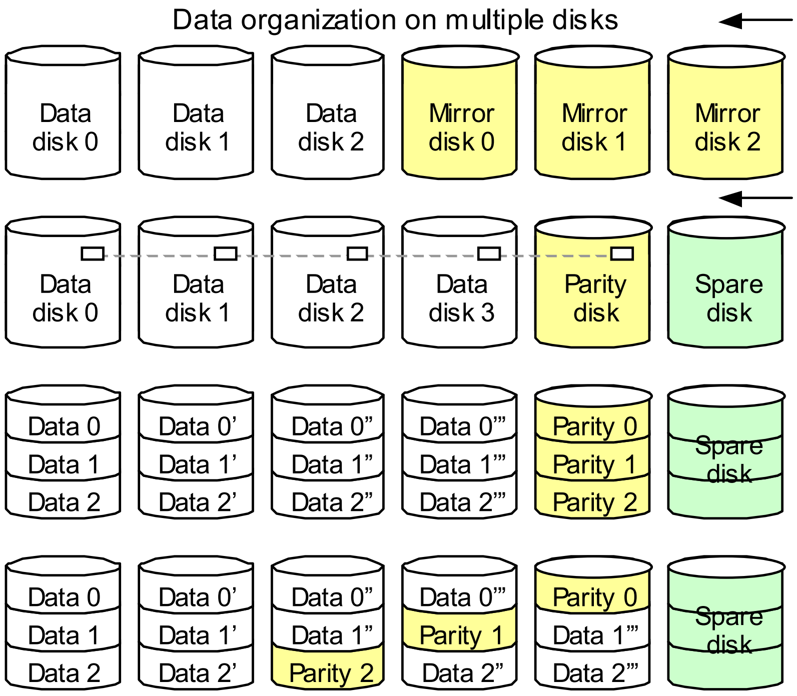
Fig. 18.8 Alternative data organizations on redundant disk arrays.

RAID Performance Considerations

Parity updates may become a bottleneck, because the parity changes with every write, no matter how small

Computing sector parity for a write operation:

$$\text{New parity} = \text{New data} \oplus \text{Old data} \oplus \text{Old parity}$$



Data organization on multiple disks

Data disk 0 | Data disk 1 | Data disk 2 | Mirror disk 0 | Mirror disk 1 | Mirror disk 2

Data disk 0 | Data disk 1 | Data disk 2 | Data disk 3 | Parity disk | Spare disk

Data 0, Data 1, Data 2 | Data 0’, Data 1’, Data 2’ | Data 0”, Data 1”, Data 2” | Data 0’”, Data 1’”, Data 2’” | Parity 0, Parity 1, Parity 2 | Spare disk

Data 0, Data 1, Data 2 | Data 0’, Data 1’, Data 2’ | Data 0”, Data 1”, Parity 2 | Data 0’”, Parity 1, Data 2” | Parity 0, Data 1’”, Data 2’” | Spare disk

RAID0: Multiple disks for higher data rate; no redundancy

RAID1: Mirrored disks

RAID2: Error-correcting code

RAID3: Bit- or byte-level striping with parity/checksum disk

RAID4: Parity/checksum applied to sectors,not bits or bytes

RAID5: Parity/checksum distributed across several disks

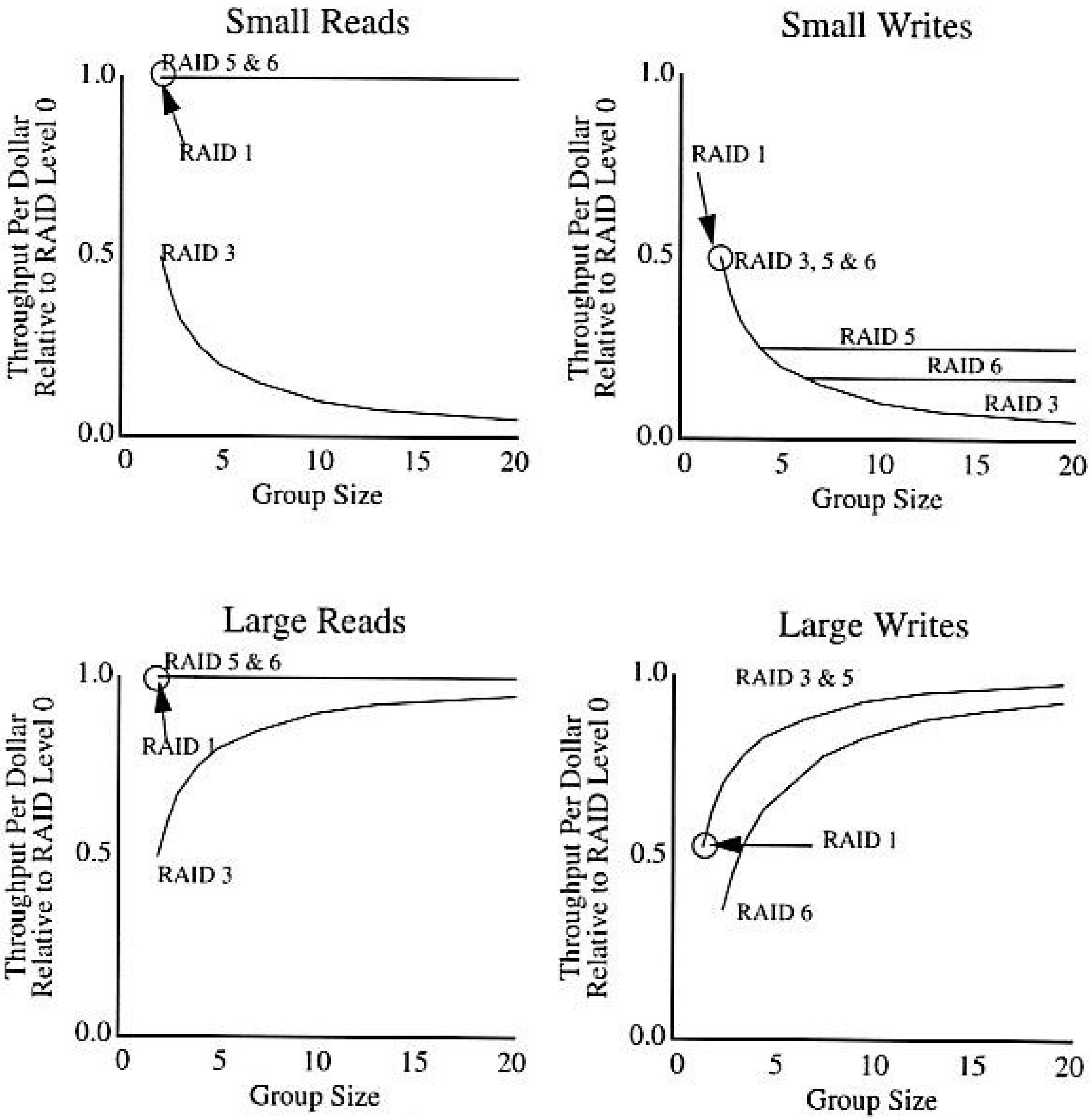
RAID6: Parity and 2nd check distributed across several disks

RAID Tradeoffs

Source: Chen, Lee, Gibson, Katz, and Patterson, “RAID: High-Performance Reliable Secondary Storage,” *ACM Computing Surveys*, 26(2):145-185, June 1994.

RAID5 and RAID 6 impose little penalty on read operations

In choosing group size, balance must be struck between the decreasing penalty for small writes vs. increasing penalty for large writes



18.6 Interfaces and Standards

The Scalable Coherent Interface (SCI) standard: Allows the implementation of large-scale cache-coherent parallel systems (see Section 21.6 for a parallel computer based on SCI)

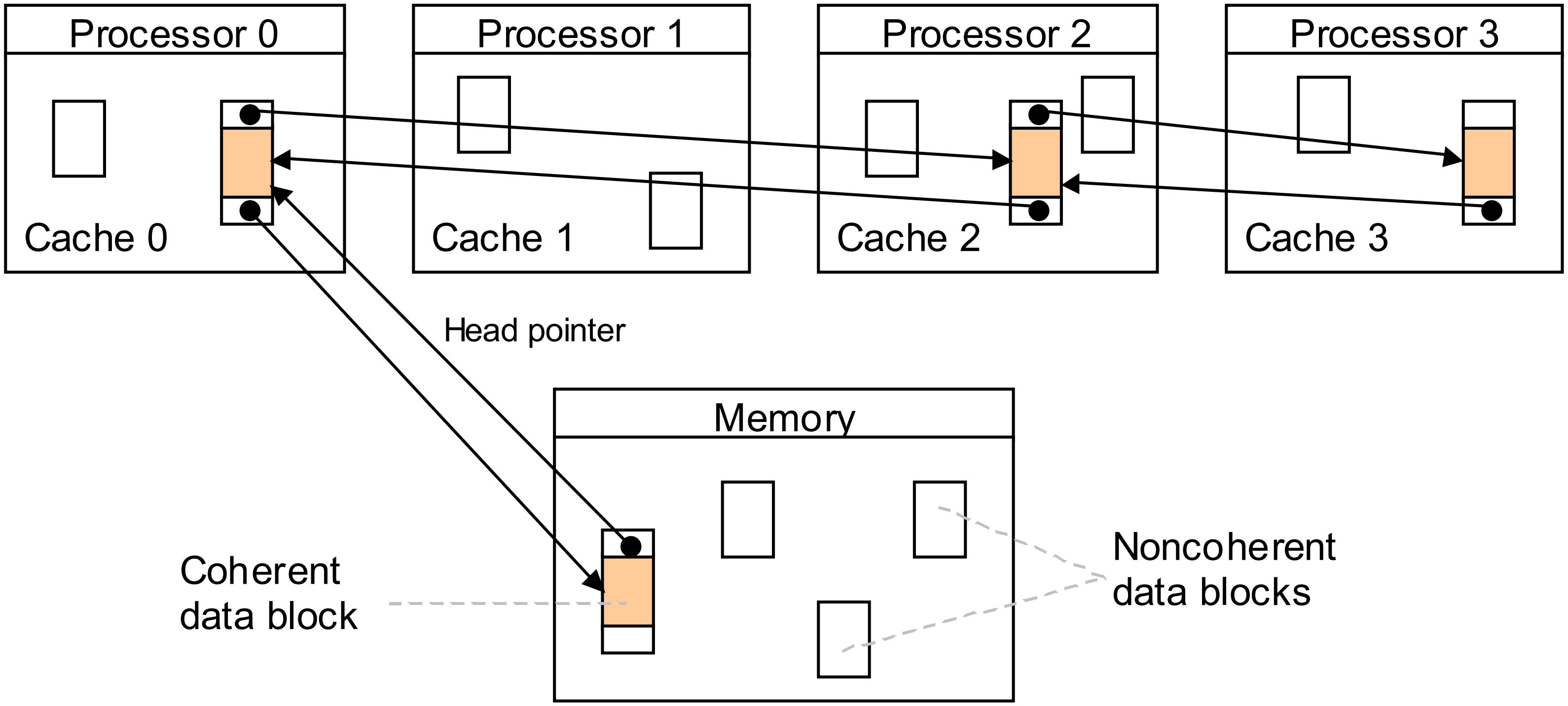


Fig. 18.9 Two categories of data blocks and the structure of the sharing set in the Scalable Coherent Interface.

Other Interface Standards

High-Performance Parallel Interface (HiPPI) ANSI standard:

Allows point-to-point connectivity between two devices (typically a supercomputer and a peripheral)

Data rate of 0.8 or 1.6 Gb/s over a (copper) cable of 25m or less

Uses very wide cables with clock rate of only 25 MHz

Establish, then tear down connections (no multiplexing allowed)

Packet length ranges from 2 B to 4 GB, up to 1016 B of control info

HiPPI (later versions renamed GSN, or gigabyte system network) is no longer in use and has been superseded by new, even faster standards such as Ultra3 SCSI and Fibre Channel

Modern interfaces tend to have fewer wires with faster clock rates

19 Reliable Parallel Processing

Develop appreciation for reliability issues in parallel systems:

- Learn methods for dealing with reliability problems
- Deal with all abstraction levels: components to systems

Topics in This Chapter	
19.1	Defects, Faults, . . . , Failures
19.2	Defect-Level Methods
19.3	Fault-Level Methods
19.4	Error-Level Methods
19.5	Malfunction-Level Methods
19.6	Degradation-Level Methods

19.1 Defects, Faults, . . . , Failures

Opportunities for fault tolerance in parallel systems:

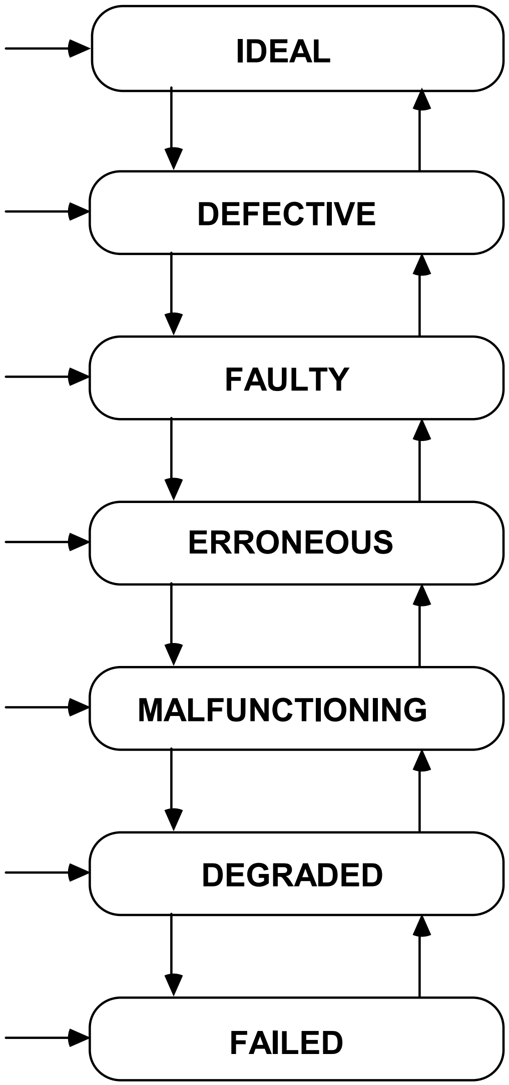
Built-in spares, load redistribution, graceful degradation

Difficulties in achieving fault tolerance:

Change in structure, bad units disturbing good ones

The multilevel model of dependable computing

Abstraction level	Dealing with deviant
Defect / Component	Atomic parts
Fault / Logic	Signal values or decisions
Error / Information	Data or internal states
Malfunction / System	Functional behavior
Degradation / Service	Performance
Failure / Result	Outputs or actions



IDEAL

DEFECTIVE

FAULTY

ERRONEOUS

MALFUNCTIONING

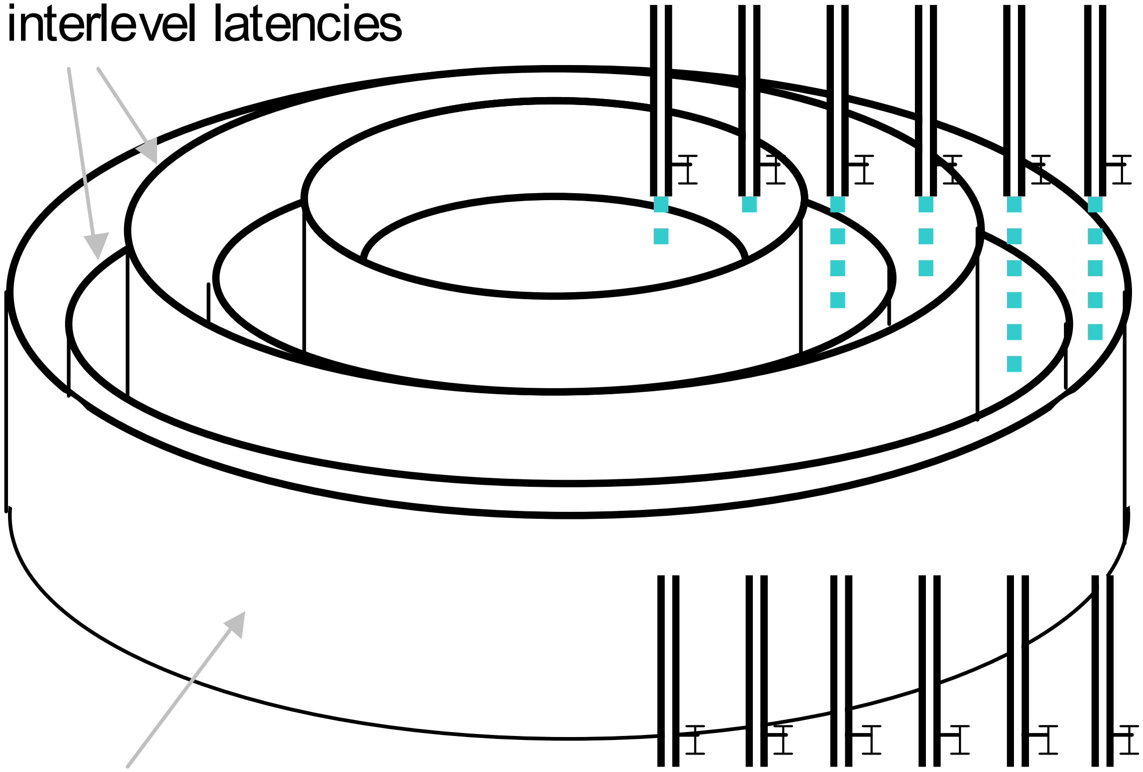
DEGRADED

FAILED

Fig. 19.1 System states and state transitions in our multilevel model.

Analogy for the Multilevel Model

Many avoidance and tolerance methods are applicable to more than one level, but we deal with them at the level for which they are most suitable, or at which they have been most successfully applied



Wall heights represent interlevel latencies

Inlet valves represent avoidance techniques

Concentric reservoirs are analogues of the six model levels (defect is innermost)

Drain valves represent tolerance techniques

Fig. 19.2 An analogy for the multilevel model of dependable computing.

19.2 Defect-Level Methods

Defects are caused in two ways (sideways and downward transitions into the defective state):

a. Design slips leading to defective components
b. Component wear and aging, or harsh operating conditions (e.g., interference)

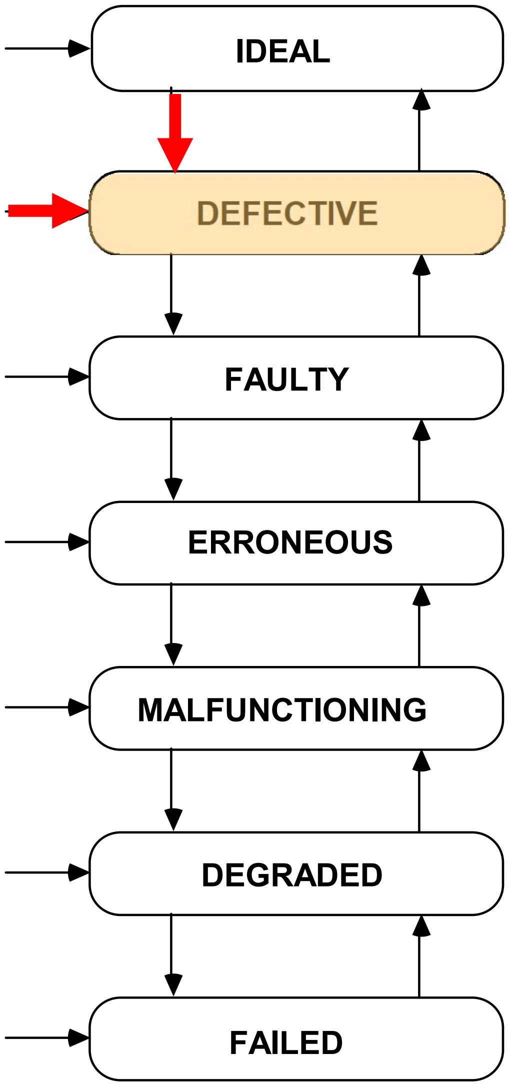
A dormant (ineffective) defect is very hard to detect

Methods for coping with defects during dormancy:

Periodic maintenance
Burn-in testing

Goal of defect tolerance methods:

Improving the manufacturing yield
Reconfiguration during system operation



Defect Tolerance Schemes for Linear Arrays

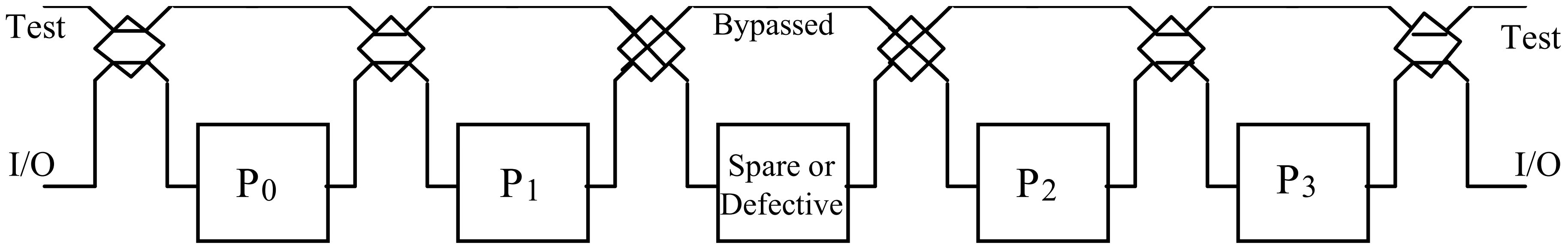


Fig. 19.3 A linear array with a spare processor and reconfiguration switches.

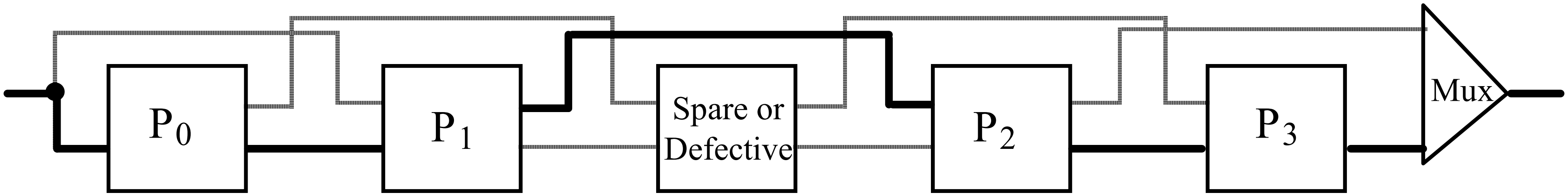


Fig. 19.4 A linear array with a spare processor and embedded switching.

Defect Tolerance in 2D Arrays

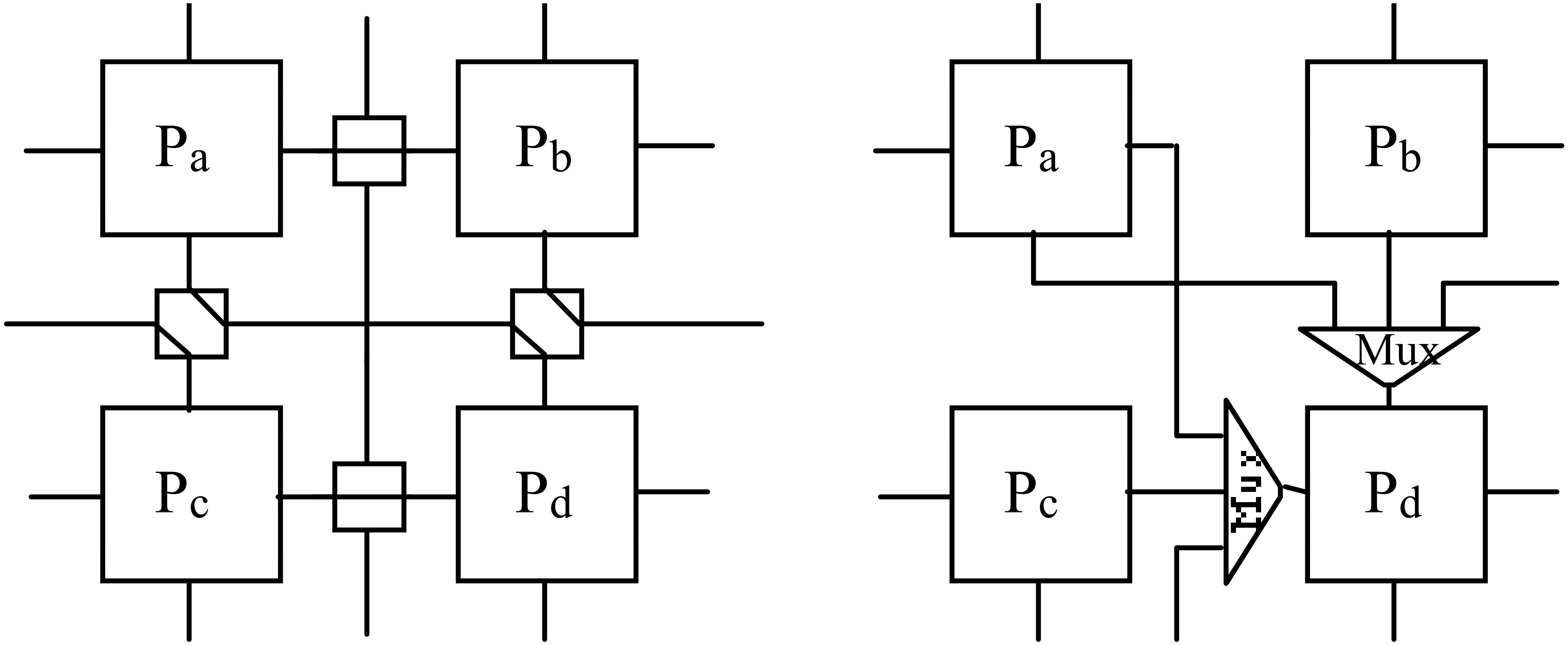


Fig. 19.5 Two types of reconfiguration switching for 2D arrays.

Assumption: A malfunctioning processor can be bypassed in its row/column by means of a separate switching mechanism (not shown)

A Reconfiguration Scheme for 2D Arrays

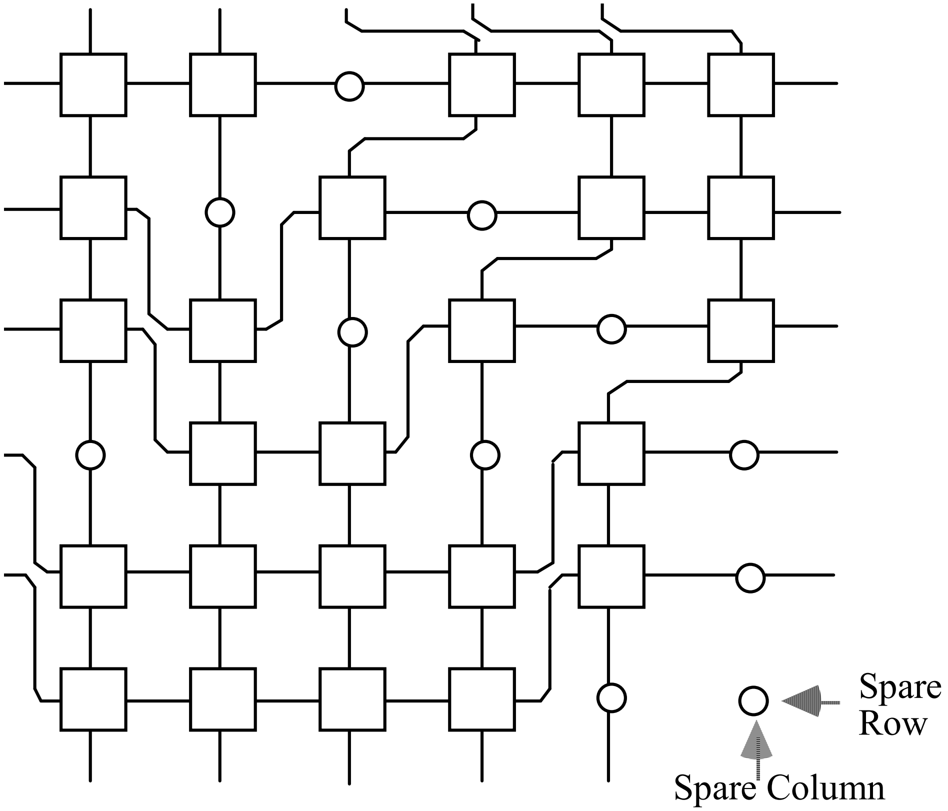


Fig. 19.6 A 5 × 5 working array salvaged from a 6 × 6 redundant mesh through reconfiguration switching.

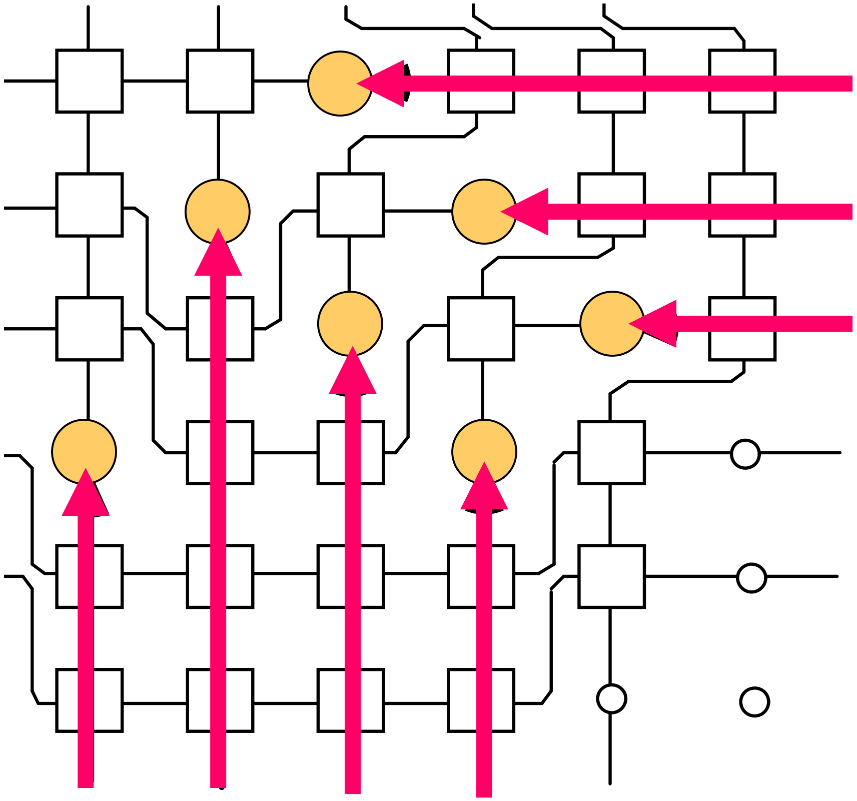
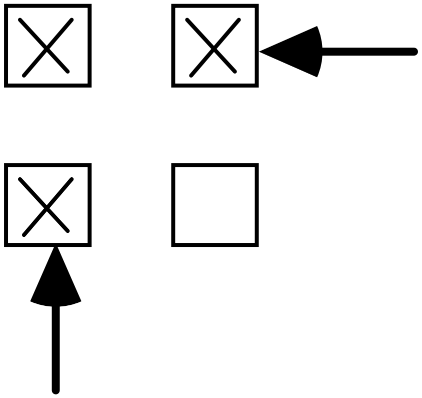


Fig. 19.7 Seven defective processors and their associated compensation paths.

Limits of Defect Tolerance



No compensation path exists for this faulty node

A set of three defective nodes, one of which cannot be accommodated by the compensation-path method.

Extension: We can go beyond the 3-defect limit by providing spare rows on top and bottom and spare columns on either side

19.3 Fault-Level Methods

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

a. Design slips leading to incorrect logic circuits
b. Exercising of defective components, leading to incorrect logic values or decisions

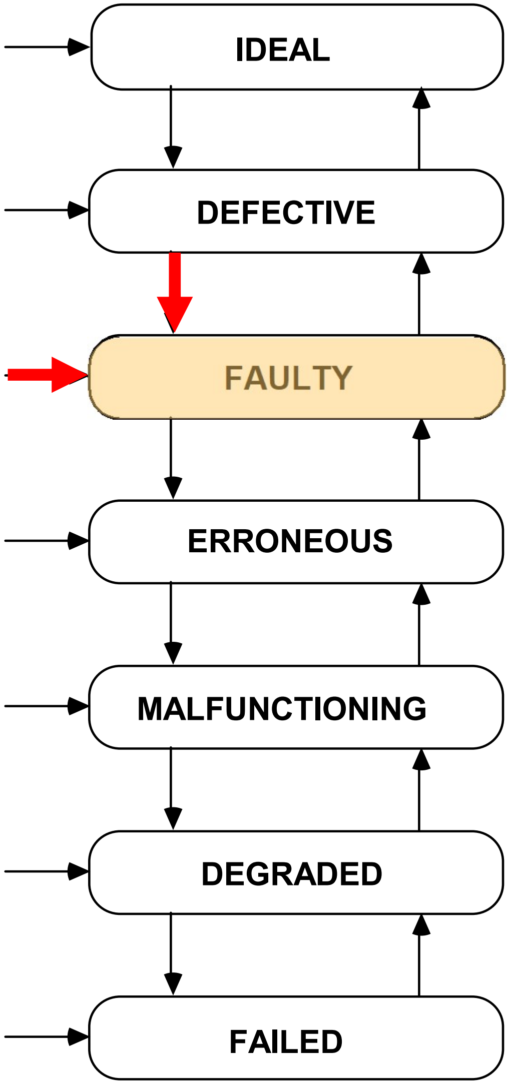
Classified as permanent / intermittent / transient, local / catastrophic, and dormant / active

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 belonging to a given class (fault model)



Fault Tolerance via Replication

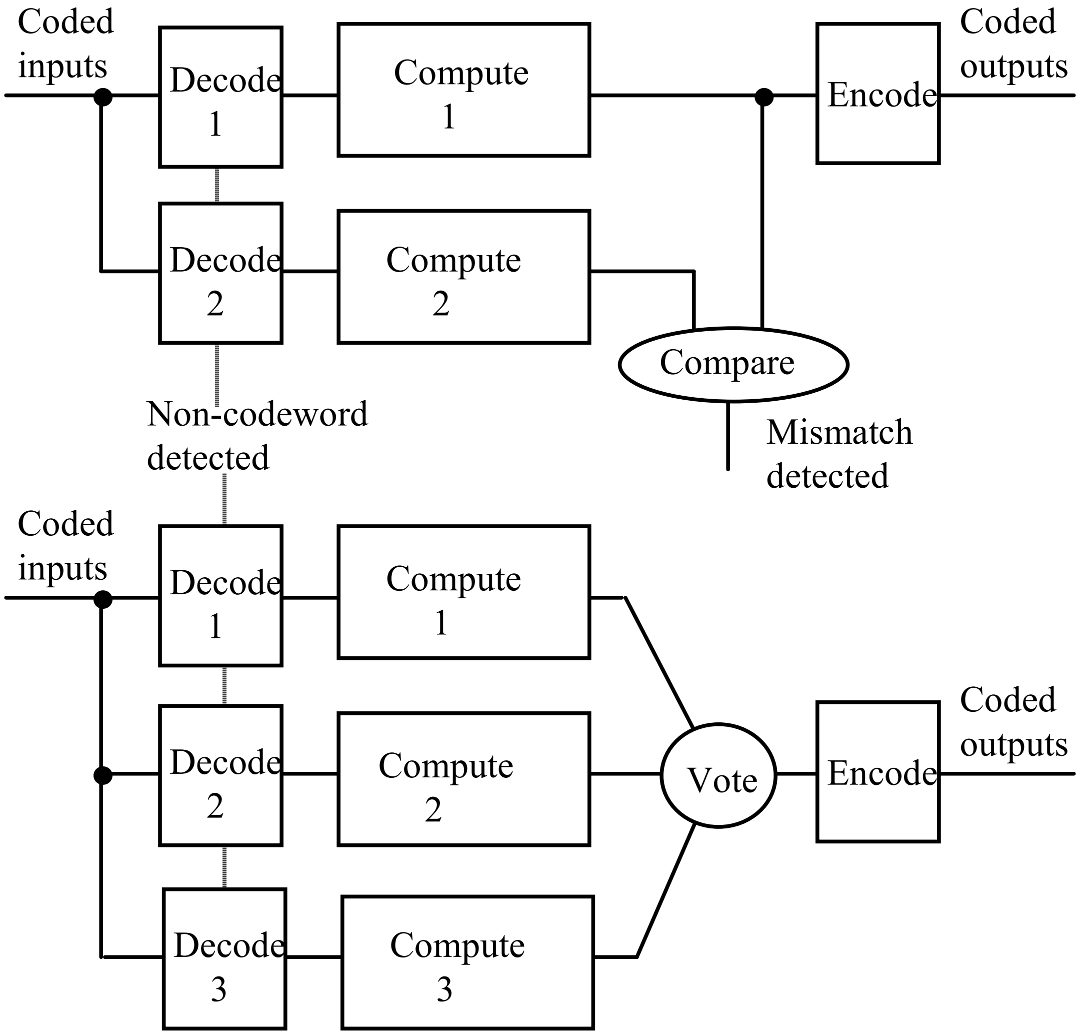


Fig. 19.8 Fault detection or tolerance with replication.

Hardware replication:

Duplication with comparison
Pair and spare
Triplication with voting

These schemes involve high redundancy (100 or 200%)

Lower redundancy possible in some cases: e.g., periodic balanced sorting networks can tolerate certain faults if provided with extra stages

Fault Detection and Bypassing

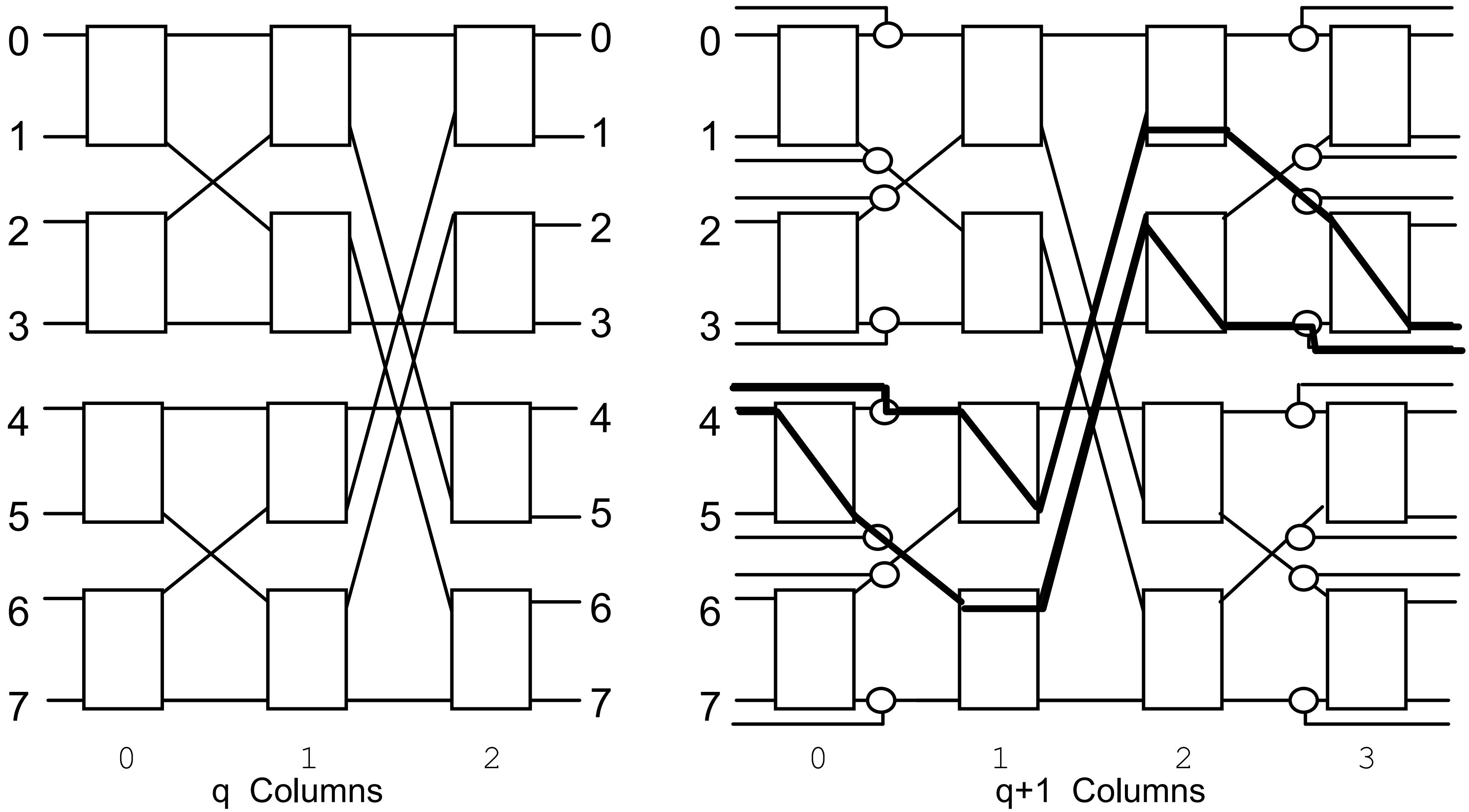


Fig. 19.9 Regular butterfly and extra-stage butterfly networks.

19.4 Error-Level Methods

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

a. Design slips leading to incorrect initial state
b. Exercising of faulty circuits, leading to deviations in stored values or machine state

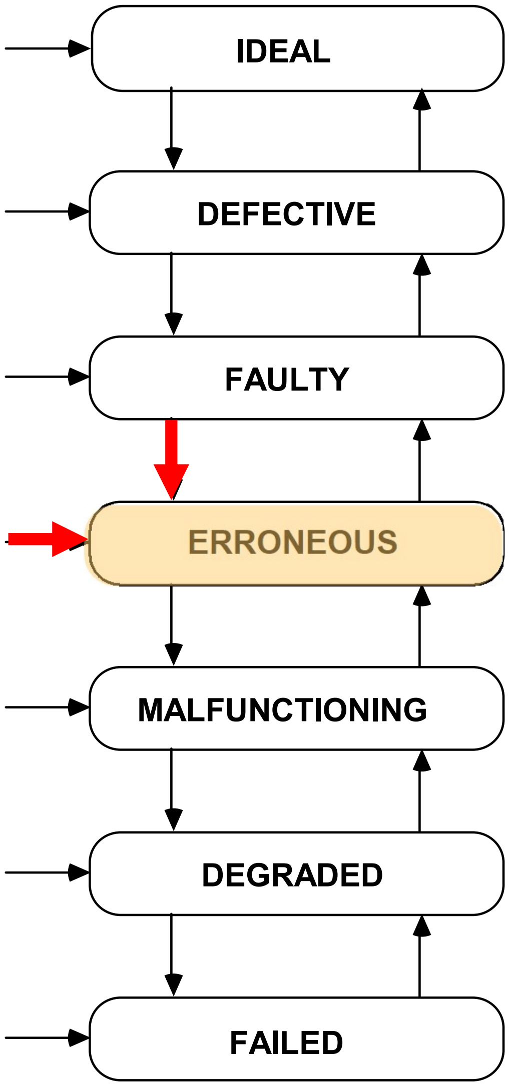
Classified as single / multiple, inversion / erasure, random / correlated, and symmetric / asymmetric

Errors are detected through:

Encoded (redundant) data, plus code checkers
Reasonableness checks or activity monitoring

Goal of error tolerance methods:

Allow uninterrupted operation in the presence of errors belonging to a given class (error model)



Application of Coding to Error Control

Ordinary codes can be used for storage and transmission errors; they are not closed under arithmetic / logic operations

Error-detecting, error-correcting, or combination codes (e.g., Hamming SEC/DED)

Arithmetic codes can help detect (or correct) errors during data manipulations:

1. Product codes (e.g., 15*x*)
2. Residue codes (*x* mod 15)

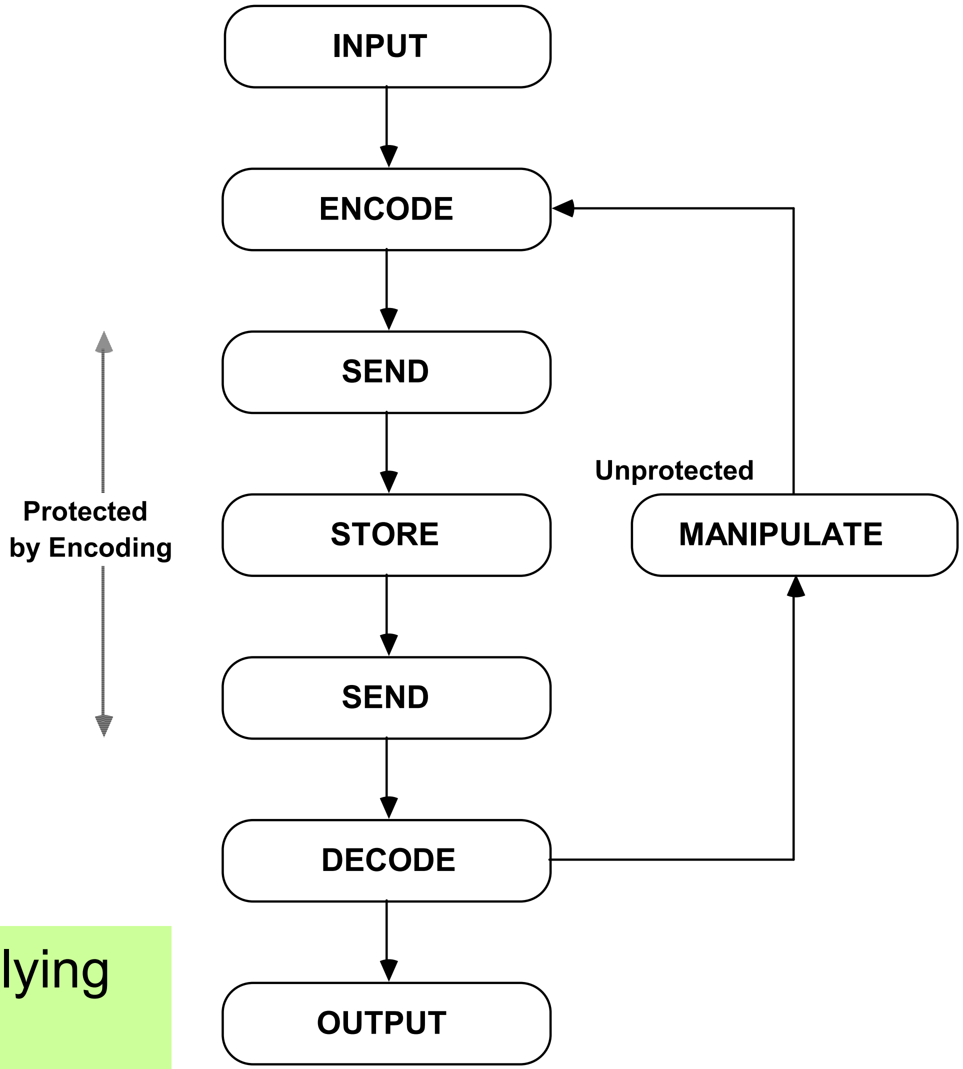


Fig. 19.10 A common way of applying information coding techniques.

Algorithm-Based Error Tolerance

Error coding applied to data structures, rather than at the level of atomic data elements

Example: mod-8 checksums used for matrices

If $Z = X \times Y$ then $Z_f = X_c \times Y_r$

In M_f, any single error is correctable and any 3 errors are detectable

Four errors may go undetected

Matrix M

$$M = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \end{pmatrix}$$

Row checksum matrix

$$M_r = \begin{pmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \end{pmatrix}$$

Column checksum matrix

$$M_c = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \\ 2 & 6 & 1 \end{pmatrix}$$

Full checksum matrix

$$M_f = \begin{pmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \\ 2 & 6 & 1 & 1 \end{pmatrix}$$

19.5 Malfunction-Level Methods

Malfunctions are caused in two ways (sideways and downward transitions into the malfunctioning state):

a. Design slips leading to incorrect modules
b. Propagation of errors to outside the module boundary, leading to incorrect interactions

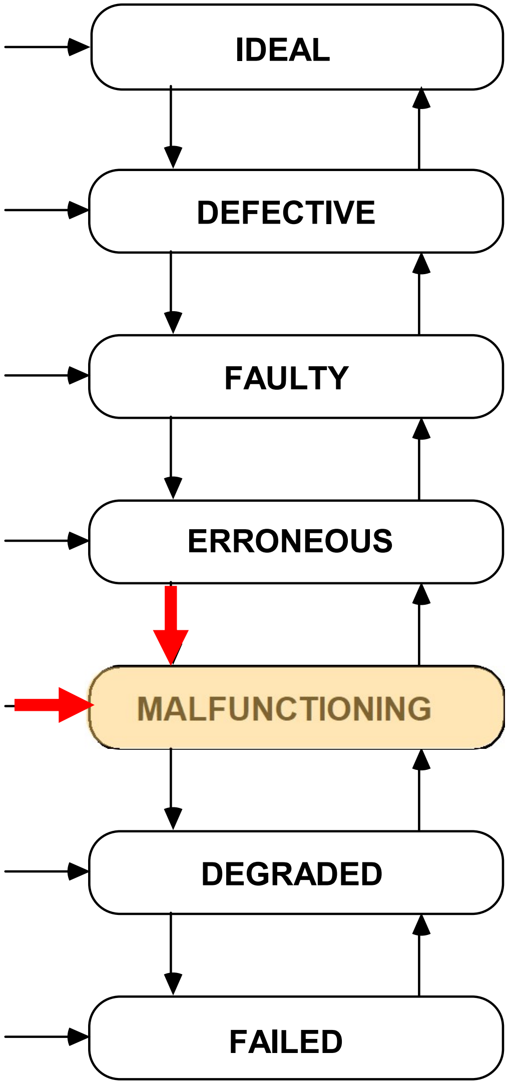
Module or subsystem malfunctions are sometimes referred to as system-level “faults”

Malfunctions are identified through diagnosis:

Begin with a trusted fault-free core, and expand
Process diagnostic data from multiple sources

Goal of malfunction tolerance methods:

Allow uninterrupted (possibly degraded) operation in the presence of certain expected malfunctions



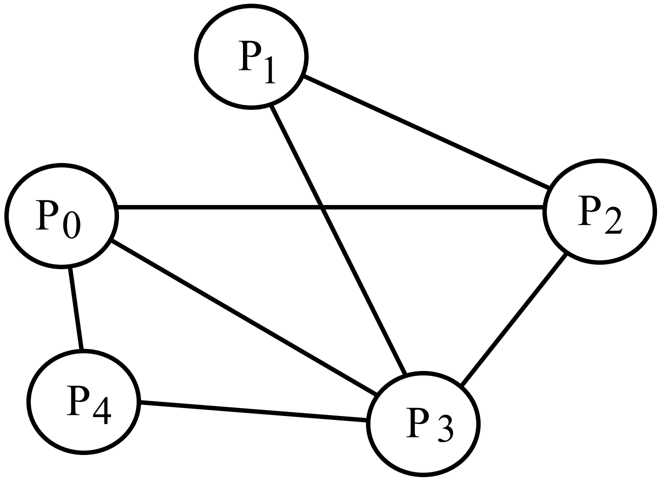
Malfunction Diagnosis in Parallel Systems

In a *p*-processor system in which processors can test each other, the diagnosis information can be viewed as a $p \times p$ matrix of test outcomes (D_{ij} represents the assessment of process *i* regarding processor *j*)

Assume that a healthy processor can reliably indicate the status of another processor, but that a malfunctioning processor cannot be trusted

The given matrix *D* is consistent with two conclusions:

1. Only P_3 is malfunctioning
2. Only P_3 is healthy



p_0 considers p_3 malfunctioning

$$D = \begin{bmatrix} x & x & 1 & 0 & 1 \\ x & x & 1 & 0 & x \\ 1 & 1 & x & 0 & x \\ 0 & 0 & 0 & x & 0 \\ 1 & x & x & 0 & x \end{bmatrix}$$

p_4 considers p_0 healthy

Fig. 19.11 A testing graph and the resulting diagnosis matrix.

Diagnosability Models and Reconfiguration

Problem: Given a diagnosis matrix, identify:

1. All malfunctioning units (complete diagnosability)
2. At least one malfunctioning unit (sequential diagnosability)
3. A subset of processors guaranteed to contain all malfunctioning ones

The last option is useful only if the designated subset is not much larger than the set of malfunctioning modules

When one or more malfunctioning modules have been identified, the system must be reconfigured to allow it to function without the unavailable resources. Reconfiguration may involve:

1. Recovering state info from removed modules or from back-up storage
2. Reassigning tasks and reallocating data
3. Restarting the computation at the point of interruption or from scratch

In bus-based systems, we isolate the bad modules and proceed; otherwise, we need schemes similar to those used for defect tolerance

Malfunction Tolerance with Low Redundancy

The following scheme uses only one spare processor for a 2D mesh (no increase in node degree), yet it allows system reconfiguration to circumvent any malfunctioning processor, replacing it with the spare via relabeling of the nodes

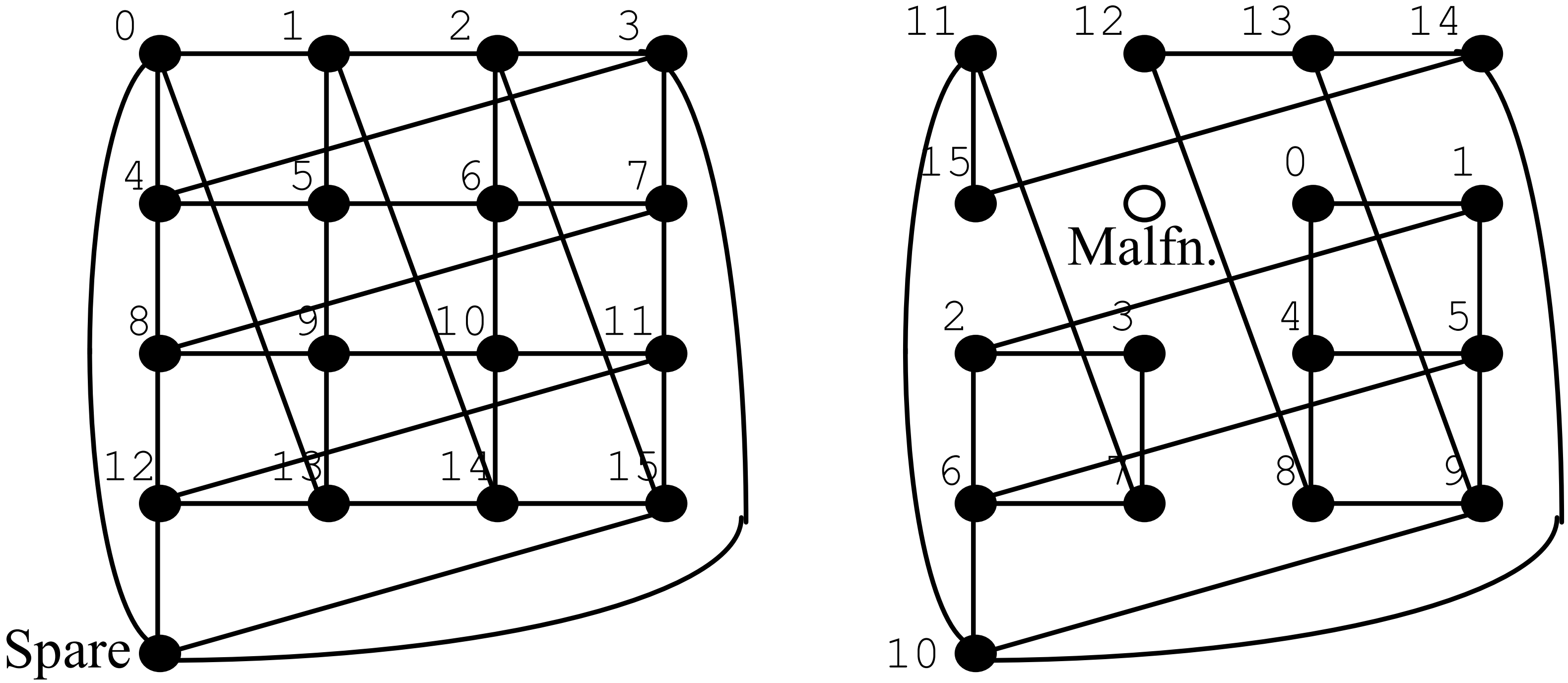


Fig. 19.12 Reconfigurable 4 × 4 mesh with one spare.

19.6 Degradation-Level Methods

Degradations are caused in two ways (sideways and downward transitions into the degraded state):

a. Design slips leading to substandard modules
b. Removal of malfunctioning modules, leading to fewer computational resources

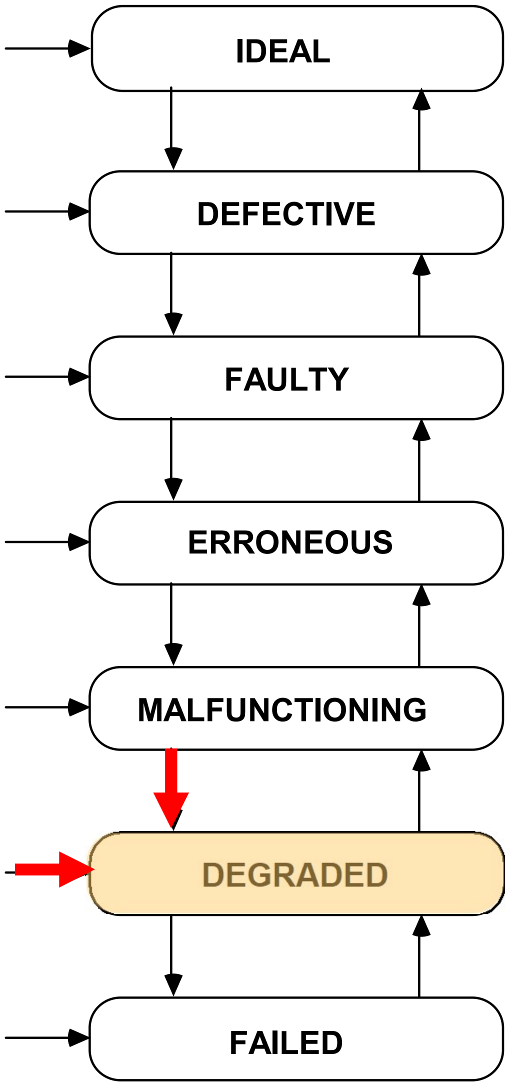
A system that can degrade “gracefully” is *fail-soft*; otherwise it is *fail-hard*

Graceful degradation has two key requirements:

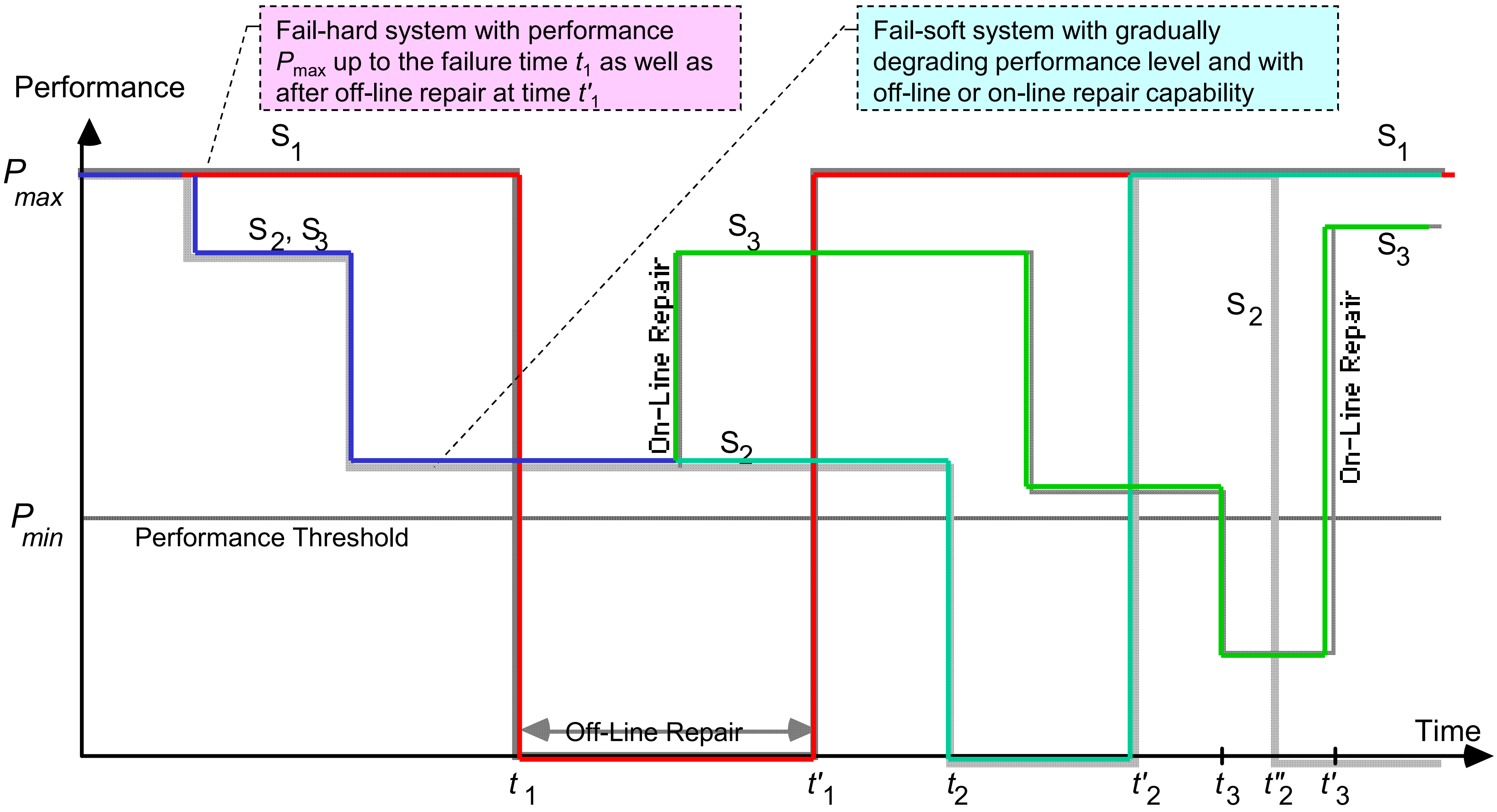
System: Diagnosis and reconfiguration
Application: Scalable and robust algorithms

Goal of degradation tolerance methods:

Allow continued operation while malfunctioning units are repaired (hot-swap capability is a plus)



Fail-Hard and Fail-Soft Systems



Fail-hard system with performance P_{max} up to the failure time t_1 as well as after off-line repair at time t'_1

Fail-soft system with gradually degrading performance level and with off-line or on-line repair capability

Fig. 19.13 Performance variations in three example parallel computers.

Checkpointing for Recovery from Malfunctions

Periodically save partial results and computation state in stable storage
Upon detected malfunction, *roll back* the computation to last checkpoint

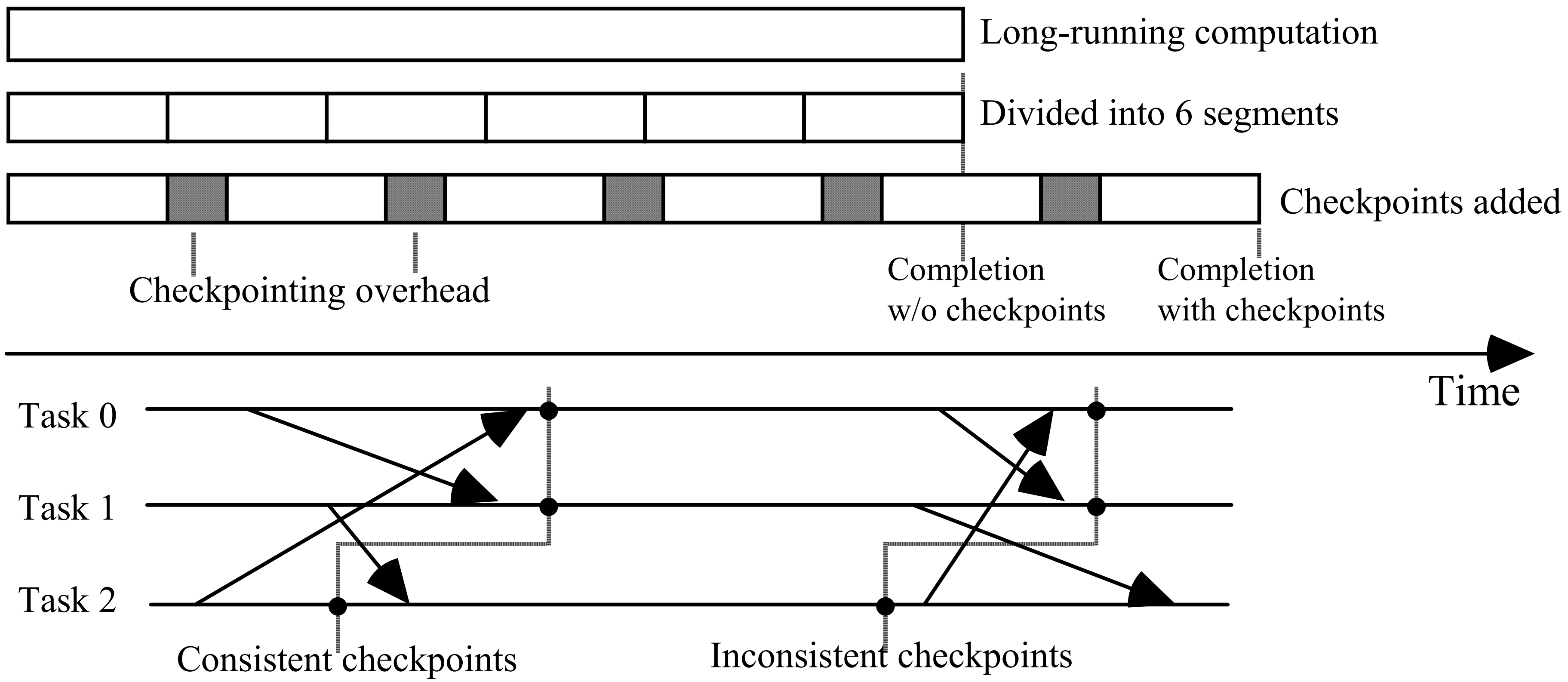


Figure 19.14 Checkpointing, its overhead, and pitfalls.

The Robust Algorithm Approach

Scalable mesh algorithm: Can run on different mesh sizes

Robust mesh algorithm: Can run on incomplete mesh, with its performance degrading gracefully as the number of unavailable nodes increases

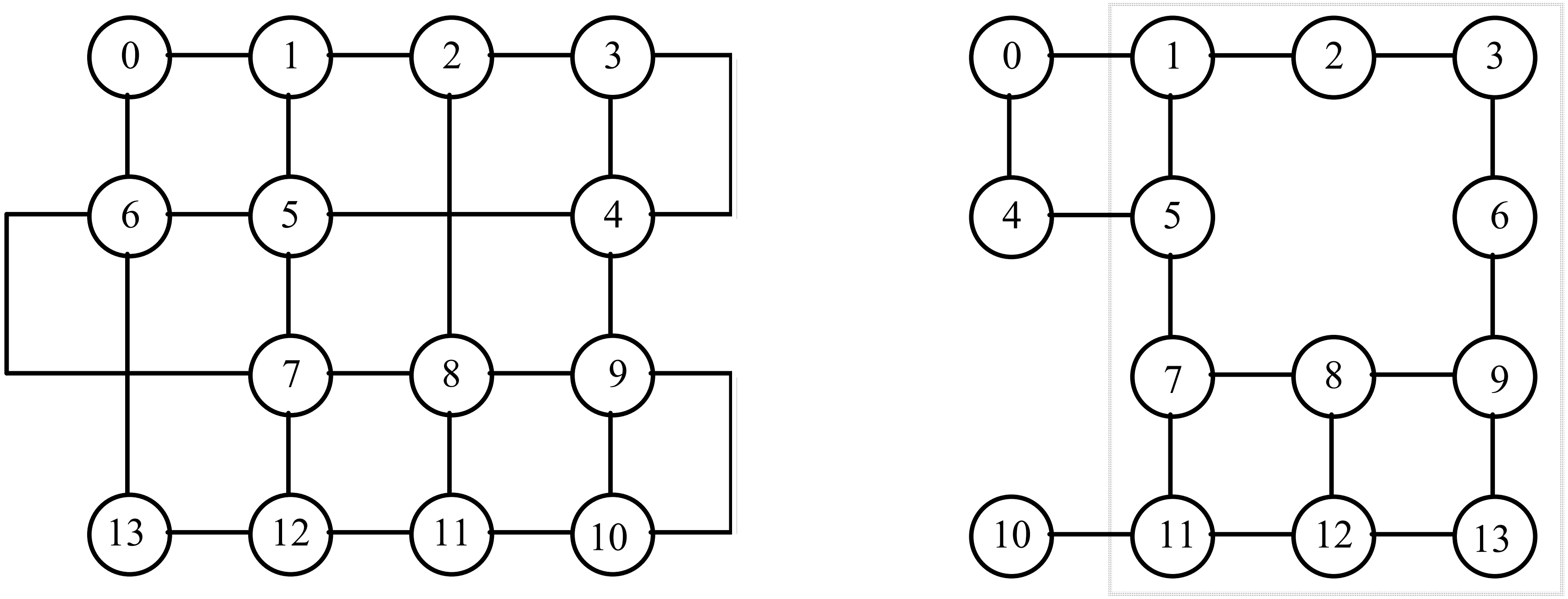


Figure 19.15 Two types of incomplete meshes, with and without bypass links.

The End of the Line: System Failure

Failures are caused in two ways (sideways and downward transitions into the failed state):

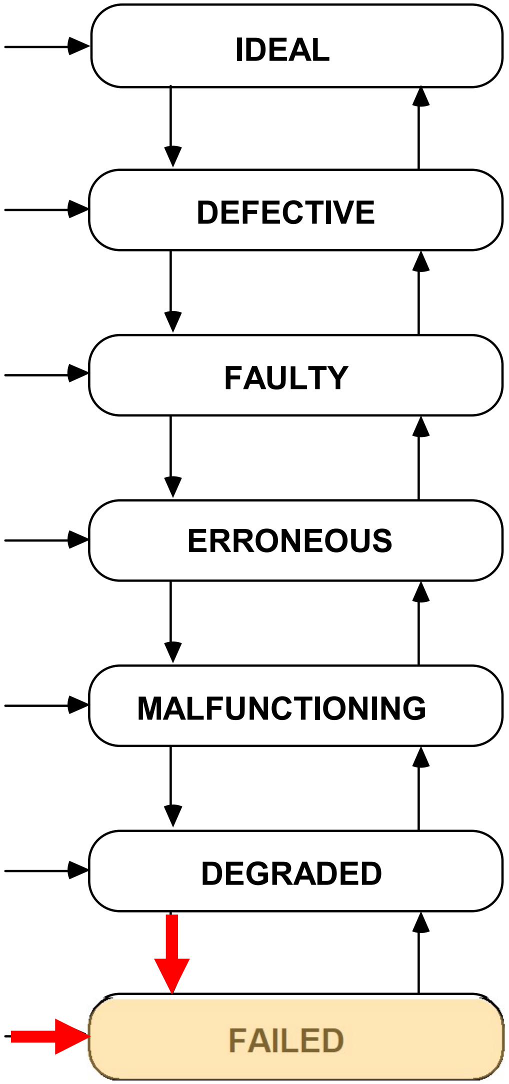
a. It is hard to believe, but some systems do not work as intended from the outset
b. Degradation beyond an acceptable threshold

It is instructive to skip a level and relate failures of a gracefully degrading system directly to malfunctions

Then, failures can be attributed to:

1. Isolated malfunction of a critical subsystem
2. Catastrophic malfunctions (space-domain)
3. Accumulation of malfunctions (time-domain)
4. Resource exhaustion

Experimental studies have shown that the first two causes of failures are the most common



20 System and Software Issues

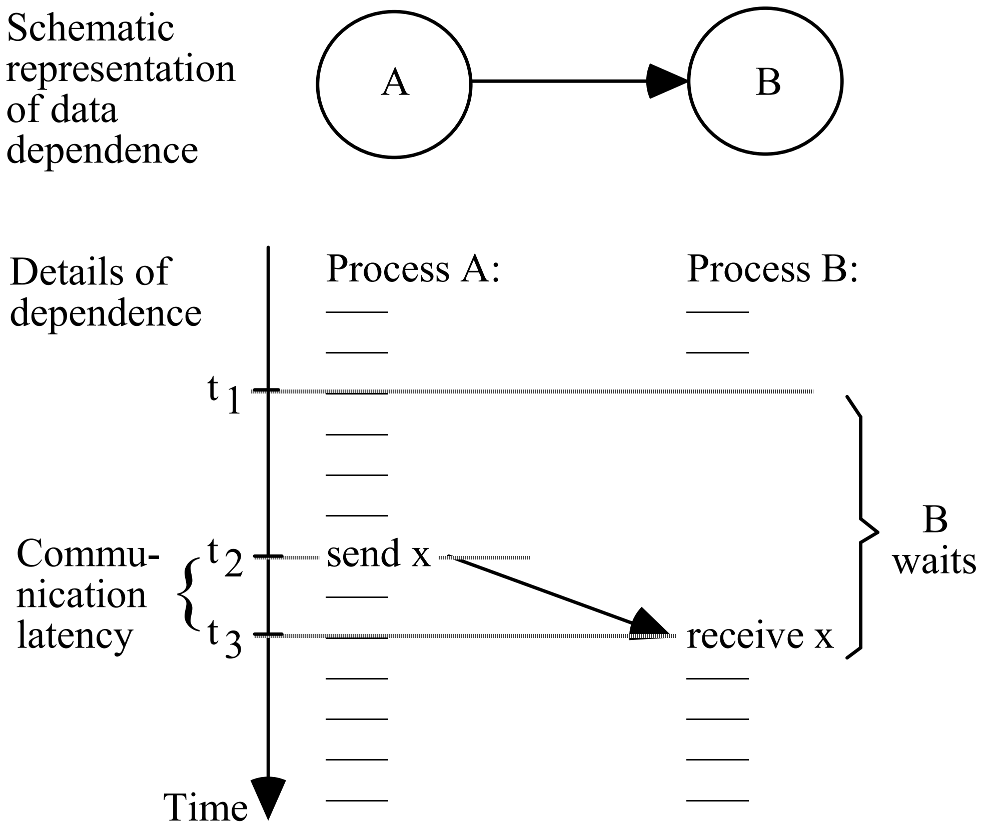
Fill void in covering system, software and application topics:

- Introduce interaction and synchronization issues
- Review progress in system and application software

Topics in This Chapter	
20.1	Coordination and Synchronization
20.2	Parallel Programming
20.3	Software Portability and Standards
20.4	Parallel Operating Systems
20.5	Parallel File Systems
20.6	Hardware/Software Interaction

20.1 Coordination and Synchronization

Task interdependence is often more complicated than the simple prerequisite structure thus far considered



Schematic representation of data dependence

Details of dependence

Communication latency

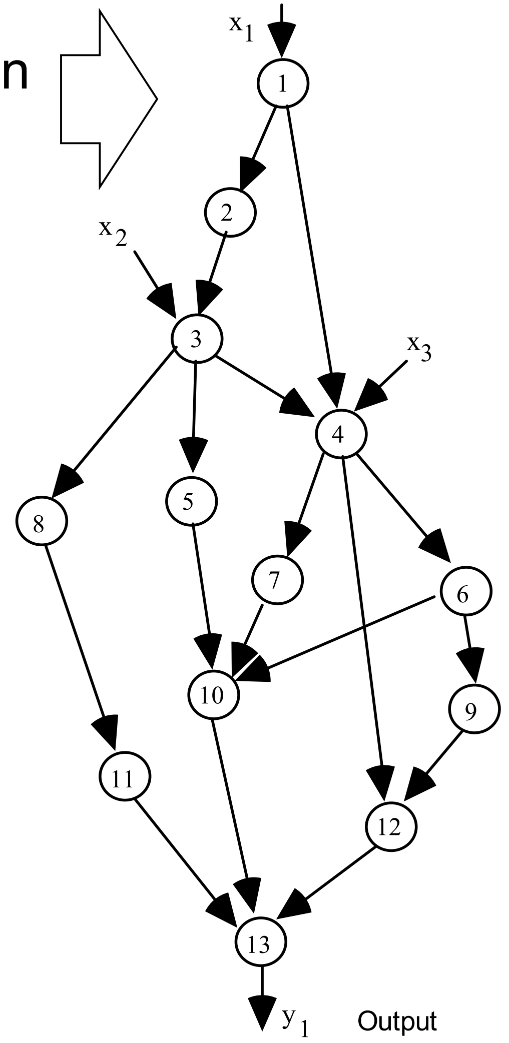


Fig. 20.1 Automatic synchronization in message-passing systems.

Synchronization with Shared Memory

Accomplished by accessing specially designated shared control variables

The fetch-and-add instruction constitutes a useful atomic operation

If the current value of *x* is *c*, fetch-and-add(*x*, *a*) returns *c* to the process and overwrites *x* = *c* with the value *c* + *a*

A second process executing fetch-and-add(*x*, *b*) then gets the now current value *c* + *a* and modifies it to *c* + *a* + *b*

Why atomicity of fetch-and-add is important: With ordinary instructions, the 3 steps of fetch-and-add for *A* and *B* may be interleaved as follows:

	Process *A*	Process *B*	Comments
Time step 1	read *x*		*A*'s accumulator holds *c*
Time step 2		read *x*	*B*'s accumulator holds *c*
Time step 3	add *a*		*A*'s accumulator holds *c* + *a*
Time step 4		add *b*	*B*'s accumulator holds *c* + *b*
Time step 5	store *x*		*x* holds *c* + *a*
Time step 6		store *x*	*x* holds *c* + *b* (not *c* + *a* + *b*)

Barrier Synchronization

Make each processor, in a designated set, wait at a barrier until all other processors have arrived at the corresponding points in their computations

Software implementation via fetch-and-add or similar instruction

Hardware implementation via an AND tree (raise flag, check AND result)

A problem with the AND-tree: If a processor can be randomly delayed between raising it flag and checking the tree output, some processors might cross the barrier and lower their flags before others have noticed the change in the AND tree output

Solution: Use two AND trees for alternating barrier points

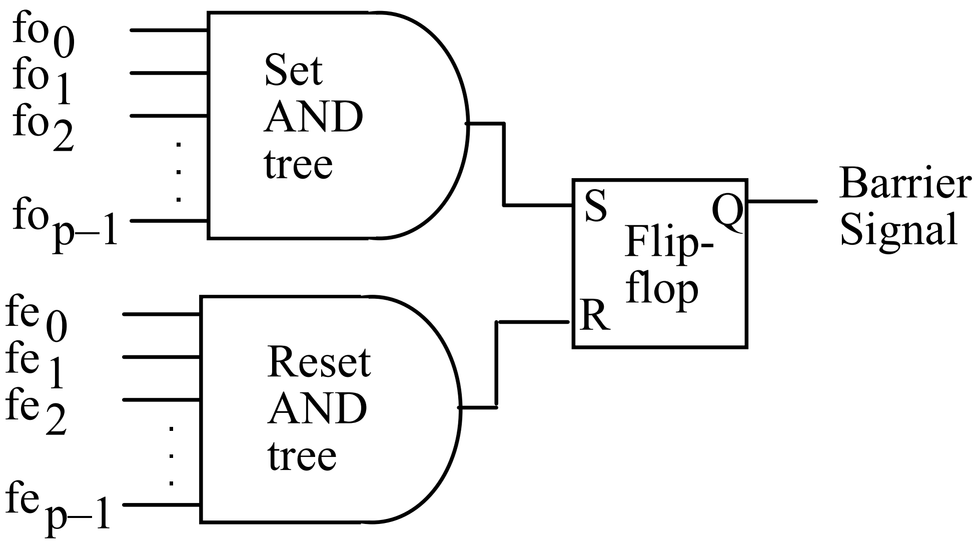
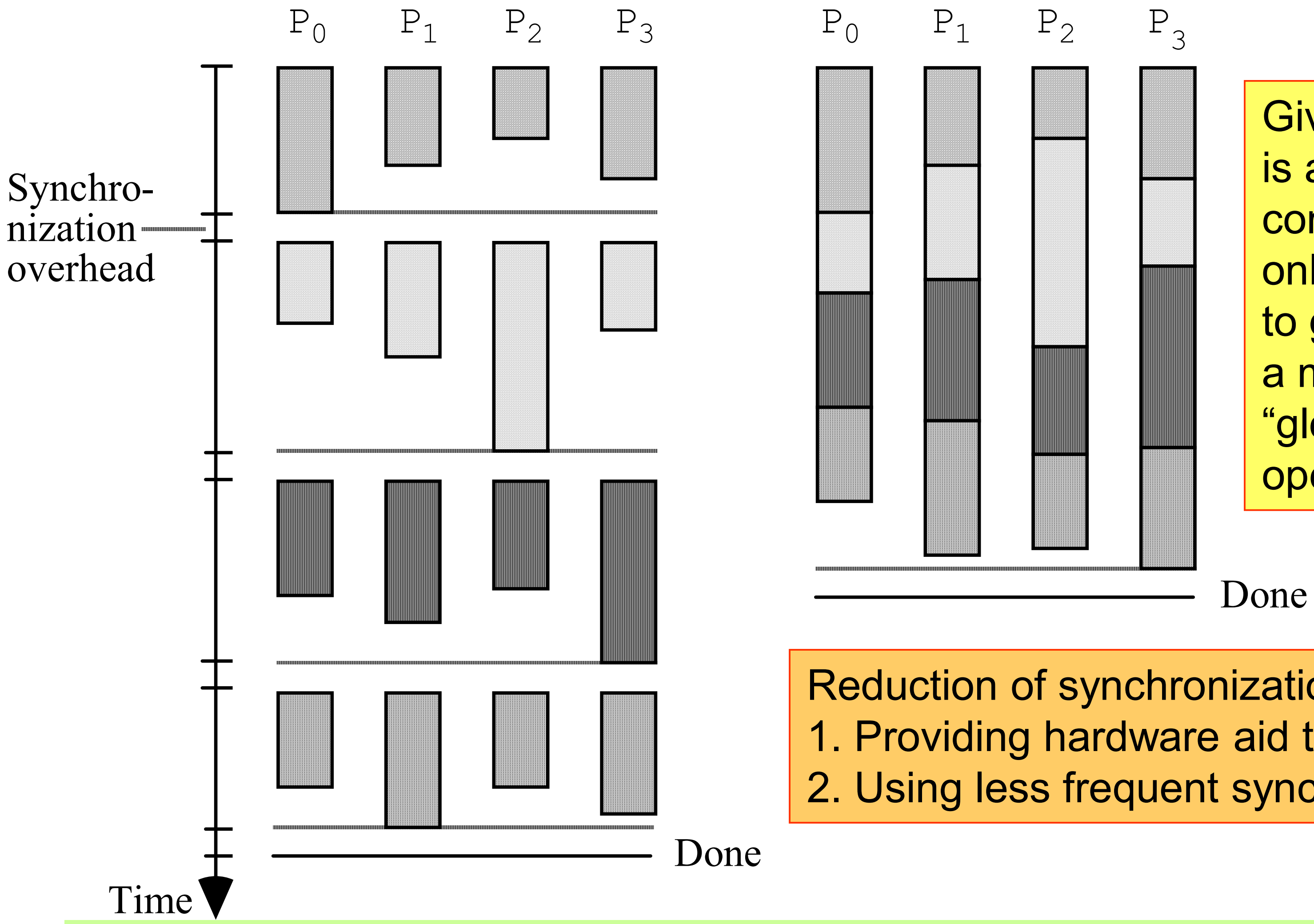


Fig. 20.4 Example of hardware aid for fast barrier synchronization [Hoar96].

Synchronization Overhead



Given that AND is a semigroup computation, it is only a small step to generalize it to a more flexible "global combine" operation

Reduction of synchronization overhead:
1. Providing hardware aid to do it faster
2. Using less frequent synchronizations

Fig. 20.3 The performance benefit of less frequent synchronization.

20.2 Parallel Programming

Some approaches to the development of parallel programs:

a. Automatic extraction of parallelism
b. Data-parallel programming
c. Shared-variable programming
d. Communicating processes
e. Functional programming

Examples of, and implementation means for, these approaches:

a. Parallelizing compilers
b. Array language extensions, as in HPF
c. Shared-variable languages and language extensions
d. The message-passing interface (MPI) standard
e. Lisp-based languages

Automatic Extraction of Parallelism

An ironic, but common, approach to using parallel computers:

Force naturally parallel computations into sequential molds by coding them in standard languages

Apply the powers of intelligent compilers to determine which of these artificially sequentialized computations can be performed concurrently

Parallelizing compilers extract parallelism from sequential programs, primarily through concurrent execution of loop iterations:

```
for $i$ = 2 to $k$ do
    for $j$ = 2 to $k$ do
        $a_{i,j}$ : = $(a_{i,j-1} + a_{i,j+1})/2$
    endfor
endfor
```

Various iteration of the *i* loop can be executed on a different processor with complete asynchrony due to their complete independence

Data-Parallel Programming

Has its roots in the math-based APL language that had array data types and array operations (binary and reduction operations on arrays)

$C \leftarrow A + B$ {array add}
$x \leftarrow +/V$ {reduction}
$U \leftarrow +/V \times W$ {inner product}

APL’s powerful operators allowed the composition of very complicated computations in a few lines (a write-only language?)

Fortran-90 (superset of Fortran-77) had extensions for array operations

```
A = SQRT(A) + B ** 2          {A and B are arrays}
WHERE (B /= 0)  A = A / B
```

When run on a distributed-memory machine, some Fortran-90 constructs imply interprocessor communication

```
A = S/2                       {assign scalar value to array}
A(I:J) = B(J:I:–1)            {assign a section of B to A}
A(P) = B                      {A(P(I)) = B(I) for all I}
S = SUM(B)                    {may require gather operation}
```

Data-Parallel Languages

High Performance Fortran (HPF) extends Fortran-90 by:

Adding new directives and language constructs
Imposing some restrictions for efficiency reasons

HPF directives assist the compiler in data distribution, but do not alter the program’s semantics (in Fortran-90, they are interpreted as comments and thus ignored)

```
!HPF ALIGN A(I) WITH B(I + 2)
```

Data-parallel extensions have also been implemented for several other programming languages

C* language introduced in 1987 by TMC
pC++, based on the popular C++

Other Approaches to Parallel Programming

Shared-variable programming

Languages: Concurrent Pascal, Modula-2, Sequent C

Communicating processes

Languages: Ada, Occam
Language-independent libraries: MPI standard

Functional programming

Based on reduction and evaluation of expressions
There is no concept of storage, assignment, or branching
Results are obtained by applying functions to arguments

One can view a functional programming language as allowing only one assignment of value to each variable, with the assigned value maintained throughout the course of the computation

20.3 Software Portability and Standards

Portable parallel applications that can run on any parallel machine have been elusive thus far

Program portability requires strict adherence to design and specification standards that provide machine-independent views or logical models

Programs are developed according to these logical models and are then adapted to specific hardware by automatic tools (e.g., compilers)

HPF is an example of a standard language that, if implemented correctly, should allow programs to be easily ported across platforms

Two other logical models are: MPI and PVM

The Message Passing Interface (MPI) Standard

The MPI Forum, a consortium of parallel computer vendors and software development specialists, specified a library of functions that implement the message-passing model of parallel computation

MPI provides a 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 for:
Point-to-point communication (blocking/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, …)

The Parallel Virtual Machine (PVM) Standard

Software platform for developing and running parallel applications on a set of independent heterogeneous computers, variously interconnected

PVM defines a suite of user-interface primitives that support both the shared-memory and the message-passing programming paradigms

These primitives provide functions similar to those of MPI and are embedded within a procedural host language (usually Fortran or C)

A 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 in-transit packets
- Coordinate task execution on associated hosts

The available pool of processors may change dynamically
Names can be associated with groups or processes
Group membership can change dynamically
One process can belong to many groups
Group-oriented functions (bcast, barrier) take group names as arguments

20.4 Parallel Operating Systems

Classes of parallel processors:

Back-end, front-end, stand-alone

Back-end system: Host computer has a standard OS, and manages the parallel processor essentially like a coprocessor or I/O device

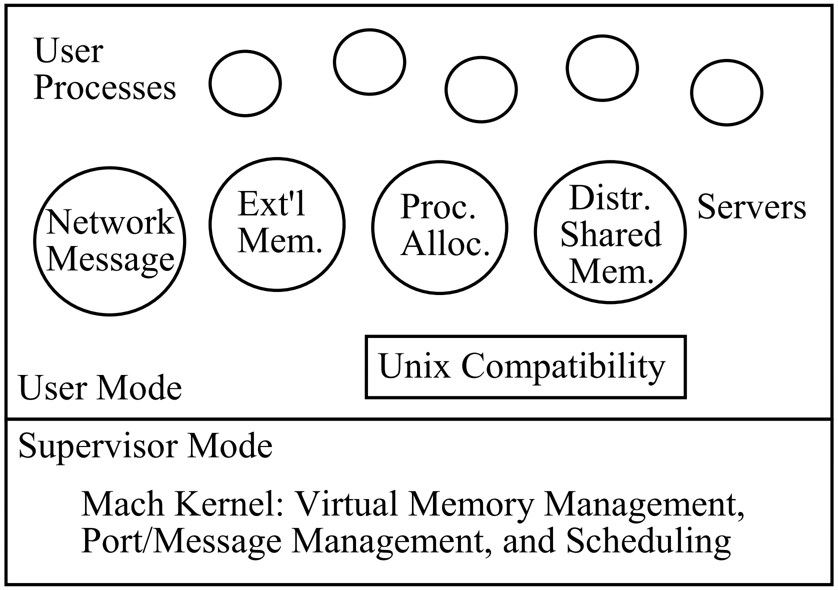
Front-end system: Similar to backend system, except that the parallel processor handles its own data (e.g., an array processor doing radar signal processing) and relies on the host computer for certain postprocessing functions, diagnostic testing, and interface with users

Stand-alone system: A special OS is included that can run on one, several, or all of the processors in a floating or distributed (master-slave or symmetric) fashion

Most parallel OSs are based on Unix

The Mach Operating System

Fig. 20.5 Functions of the supervisor and user modes in the Mach operating system.



Mach is based on Unix and has many similarities with it

To make a compact, modular kernel possible, Mach incorporates a small set of basic abstractions:

a. Task: A “container” for resources like virtual address space and ports
b. Thread: A program with little context; a task may contain many threads
c. Port: A communication channel along with certain access rights
d. Message: A basic unit of information exchange
e. Memory object: A “handle” to part of a task’s virtual memory

Some Features of The Mach OS

Unlike Unix, whose memory consists of contiguous areas, Mach’s virtual address space contains pages with separate protection and inheritance

Messages in Mach are communicated via ports

Messages are typed according to their data and can be sent over a port only if the sending / receiving thread has the appropriate access rights

For efficiency, messages involving a large amount of data do not actually carry the data; instead a pointer to the actual data pages is transmitted

Copying of the data to the receiver’s pages occurs only upon data access

Mach scheduler assigns to each thread a time quantum upon starting its execution. When the time quantum expires, a context switch is made to a thread with highest priority, if such a thread is awaiting execution

To avoid starvation of low-priority threads (and to favor interactive tasks over computation-intensive ones), priorities are reduced based on “age”; the more CPU time a thread uses, the lower its priority becomes.

20.5 Parallel File Systems

A parallel file system efficiently maps data access requests by the processors to high-bandwidth data transfers between primary and secondary memories

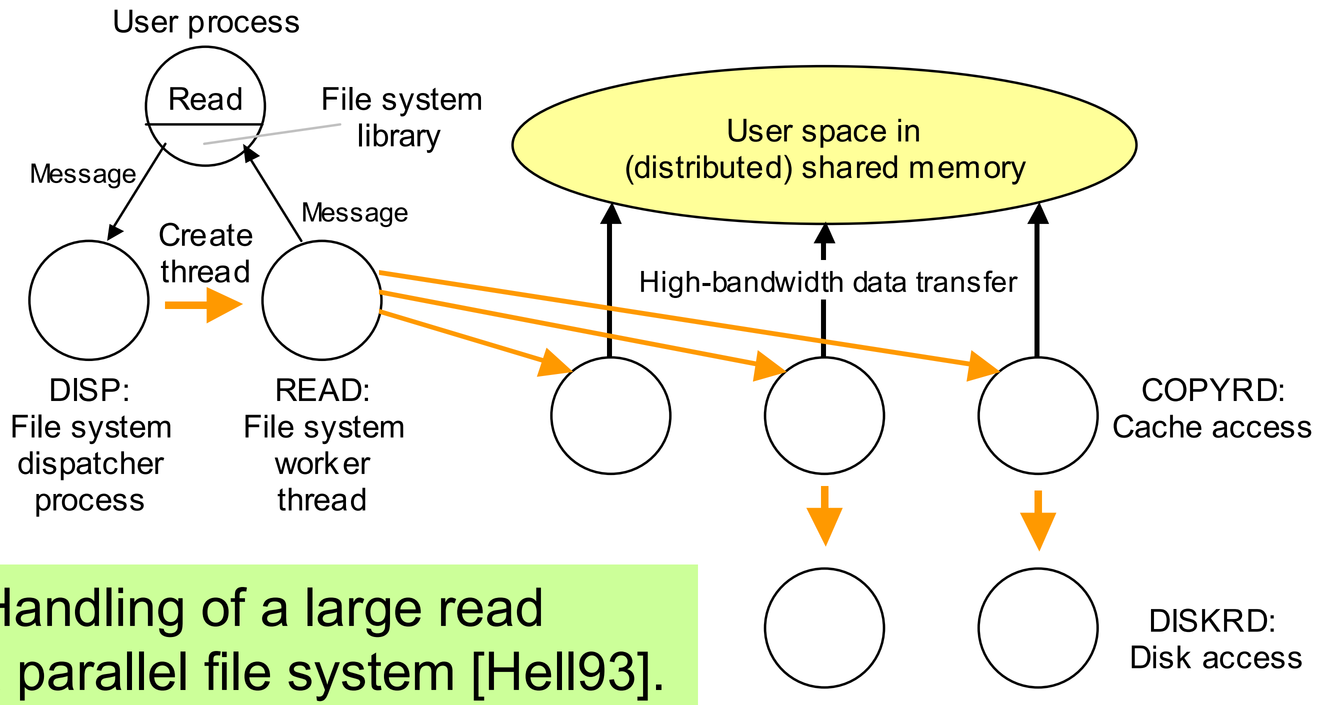

User process
Read
File system library
Message
Message
Create thread
DISP: File system dispatcher process
READ: File system worker thread
User space in (distributed) shared memory
High-bandwidth data transfer
COPYRD: Cache access
DISKRD: Disk access

Fig. 20.6 Handling of a large read request by a parallel file system [Hell93].

To avoid a performance bottleneck, a parallel file system must be a highly parallel and scalable program that can deal with many access scenarios:

a. Concurrent file access by independent processes
b. Shared access to files by cooperating processes
c. Access to large data sets by a single process

20.6 Hardware / Software Interaction

A parallel application should be executable, with little or no modification, on a variety of hardware platforms that differ in architecture and scale

Changeover from an 8-processor to 16-processor configuration, say, should not require modification in the system or application programs

Ideally, upgrading should be done by simply plugging in new processors, along with interconnects, and rebooting

Scalability in time: Introduction of faster processors and interconnects leads to an increase in system performance with little or no redesign (difficult at present but may become possible in future via the adoption of implementation and interfacing standards)

Scalability in space: Computational power can be increased by simply plugging in more processors (many commercially available parallel processors are scalable in space within a range; say 4-256 processors)

Speedup and Amdahl’s Law Revisited

Speedup, with the problem size *n* explicitly included, is:

$$S(n, p) = T(n, 1) / T(n, p)$$

The total time $pT(n, p)$ spent by the processors can be divided into computation time $C(n, p)$ and overhead time

$$H(n, p) = pT(n, p) - C(n, p)$$

Assuming for simplicity that we have no redundancy

$$C(n, p) = T(n, 1)$$
$$H(n, p) = pT(n, p) - T(n, 1)$$
$$S(n, p) = p / [1 + H(n, p) / T(n, 1)]$$
$$E(n, p) = S(n, p) / p = 1 / [1 + H(n, p) / T(n, 1)]$$

If the overhead per processor, $H(n, p)/p$, is a fixed fraction f of $T(n, 1)$, speedup and efficiency become:

$$S(n, p) = p / (1 + pf) < 1/f \qquad \text{\{Alternate form of Amdahl's law\}}$$
$$E(n, p) = 1 / (1 + pf)$$

Maintaining a Reasonable Efficiency

Speedup and efficiency formulas

$$E(n, p) = S(n, p) / p = 1 / (1 + pf)$$

Assume that efficiency is to be kept above 50%, but the arguments that follow apply to any fixed efficiency target

For $E(n, p) > ½$ to hold, we need $pf < 1$ or $p < 1/f$

That is, for a fixed problem size and assuming that the per-processor overhead is a fixed fraction of the single-processor running time, there is a limit to the number of processors that can be used cost-effectively

Going back to our efficiency equation $E(n, p) = 1 / [1 + H(n, p) / T(n, 1)]$, we note that keeping $E(n, p)$ above ½ requires:

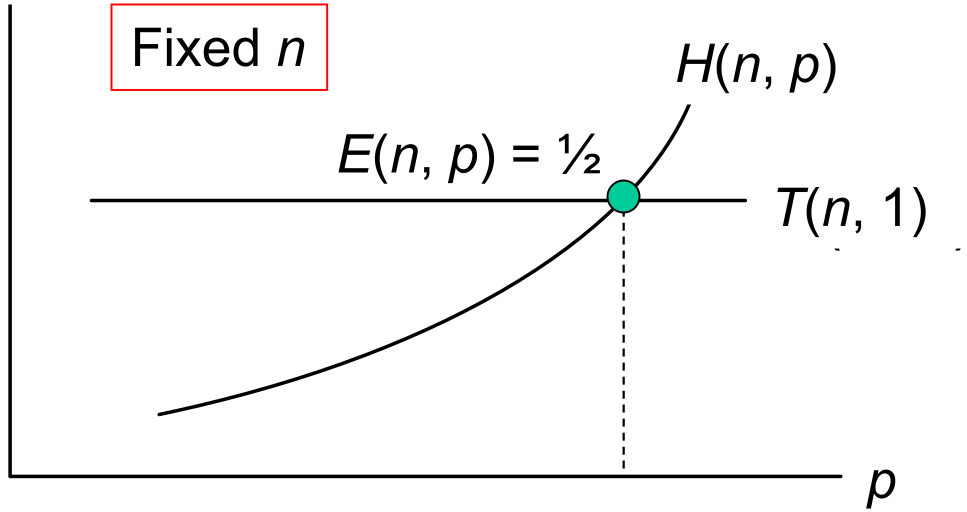
$$T(n, 1) > H(n, p)$$

Generally, the cumulative overhead $H(n, p)$ goes up with both n and p, whereas $T(n, 1)$ only depends on n

Scaled Speedup and Isoefficiency

Graphical depiction of the limit to cost-effective utilization of processors

For many problems, good efficiency can be achieved provided that we sufficiently scale up the problem size



The growth in problem size that can counteract the effect of increase in machine size p in order to achieve a fixed efficiency is referred to as the isoefficiency function $n(p)$ which can be obtained from:

$$T(n, 1) = H(n, p)$$

Scaled speedup of $p/2$ or more is achievable for suitably large problems

Because the execution time $T(n, p) = [T(n, 1) + H(n, p)] / p$ grows with problem size for good efficiency, usefulness of scaled speedup is limited