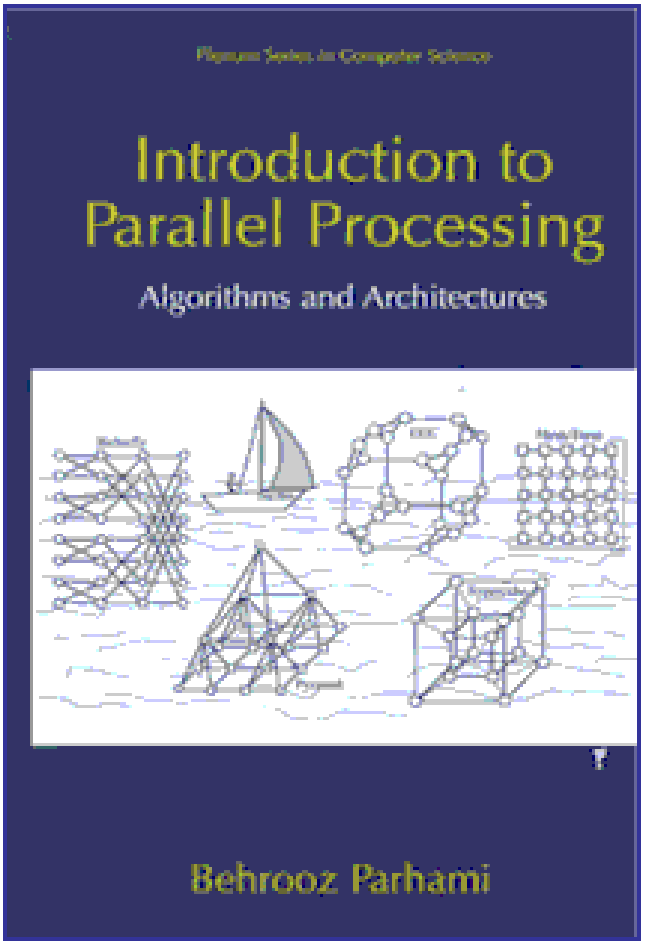
INSTRUCTOR'S MANUAL FOR



Volume 2: Presentation Material

Behrooz Parhami

Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA
E-mail: parhami@ece.ucsb.edu

© Plenum Press, Winter 2002

The structure of this book in parts, half-parts, and chapters

	Parts	Half-Parts	Chapters
Architectural Variations	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
	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

This instructor's manual is for

Introduction to Parallel Processing: Algorithms and Architectures, by Behrooz Parhami

ISBN 0-306-45970-1, QA76.58.P3798

©1999 Plenum Press, New York, http://www.plenum.com

For information and errata, see http://www.ece.ucsb.edu/Faculty/Parhami/text_par_proc.htm

All rights reserved for the author. No part of this instructor's manual may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission. Contact the author at: ECE Dept., Univ. of California, Santa Barbara, CA 93106-9560, USA (parhami@ece.ucsb.edu)

Preface to the Instructor's Manual

This instructor's manual consists of two volumes. Volume 1 presents solutions to selected problems and includes additional problems (many with solutions) that did not make the cut for inclusion in the text *Introduction to Parallel Processing: Algorithms and Architectures* (Plenum Press, 1999) or that were designed after the book went to print. Volume 2 contains enlarged versions of the figures and tables in the text as well as additional material, presented in a format that is suitable for use as transparency masters.

The winter 2002 edition Volume 1, which consists of the following parts, is available to qualified instructors through the publisher:

Volume 1	Part I	Selected solutions and additional problems
	Part II	Question bank, assignments, and projects

The winter 2002 edition of Volume 2, which consists of the following parts, is available as a large file in postscript format through the book's Web page:

Volume 2	Parts I-VI	Lecture slides and other presentation material

The book's Web page, given below, also contains an errata and a host of other material (please note the upper-case "F" and "P" and the underscore symbol after "text" and "par"):

http://www.ece.ucsb.edu/Faculty/Parhami/text_par_proc.htm

The author would appreciate the reporting of any error in the textbook or in this manual, suggestions for other tables, diagrams, or lecture topics, and sharing of teaching experiences. Please e-mail your comments to

parhami@ece.ucsb.edu

or send them by regular mail to the author's postal address:

Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA

Contributions will be acknowledged to the extent possible.

Behrooz Parhami
Santa Barbara, Winter 2002

© Winter 2002, Plenum Press

Table of Contents, Vol. 2

	Preface to the Instructor's Manual	3
Part I	**Fundamental Concepts**	**5**
1	Introduction to Parallelism	6
2	A Taste of Parallel Algorithms	29
3	Parallel Algorithm Complexity	45
4	Models of Parallel Processing	57
Part II	**Extreme Models**	**71**
5	PRAM and Basic Algorithms	72
6	More Shared-Memory Algorithms	92
7	Sorting and Selection Networks	108
8	Other Circuit-Level Examples	124
Part III	**Mesh-Based Architectures**	**141**
9	Sorting on a 2D Mesh or Torus	142
10	Routing on a 2-D Mesh or Torus	158
11	Numerical 2D Mesh Algorithms	171
12	Mesh-Related Architectures	195
Part IV	**Low-Diameter Architectures**	**222**
13	Hypercubes and Their Algorithms	223
14	Sorting and Routing on Hypercubes	243
15	Other Hypercubic Architectures	265
16	A Sampler of Other Networks	283
Part V	**Some Broad Topics**	**313**
17	Emulation and Scheduling	314
18	Data Storage, Input, and Output	332
19	Reliable Parallel Processing	342
20	System and Software Issues	359
Part VI	**Implementation Aspects**	**380**
21	Shared-Memory MIMD Machines	381
22	Message-Passing MIMD Machines	392
23	Data-Parallel SIMD Machines	404
24	Past, Present, and Future	417

Part I Fundamental Concepts

Back to TOC

Part Goals

- Motivate us to study parallel processing
- Paint the big picture
- Provide background in the three Ts:
  Taxonomy – including basic terminology
  Tools – for evaluation or comparison
  Theory – easy and hard problems

Part Contents

- Chapter 1: Introduction to Parallelism
- Chapter 2: A Taste of Parallel Algorithms
- Chapter 3: Parallel Algorithm Complexity
- Chapter 4: Models of Parallel Processing

1 Introduction to Parallelism

Back to TOC

Chapter Goals

- Set the context in which the course material will be presented
- Review challenges that face the designers and users of parallel computers
- Introduce metrics for evaluating the effectiveness of parallel systems

Chapter Contents

- 1.1. Why Parallel Processing?
- 1.2. A Motivating Example
- 1.3. Parallel Processing Ups and Downs
- 1.4. Types of Parallelism: A Taxonomy
- 1.5. Roadblocks to Parallel Processing
- 1.6. Effectiveness of Parallel Processing

1.1 Why Parallel Processing?

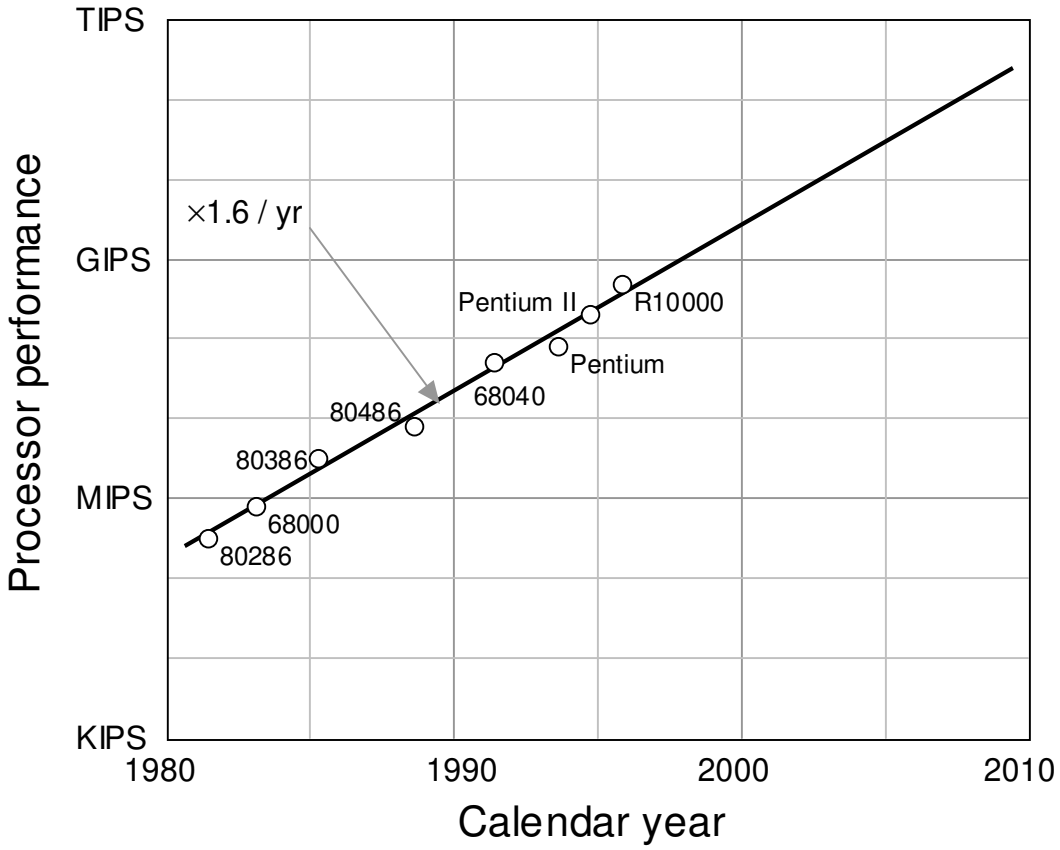


Fig. 1.1. The exponential growth of microprocessor performance, known as Moore's Law, shown over the past two decades.

Figures rounded/averaged from "2001 Technology Roadmap for Semiconductors" [Alla02]

Calendar year →	2001	2004	2007	2010	2013	2016
Halfpitch (nm)	140	90	65	45	32	22
Clock freq. (GHz)	2	4	7	12	20	30
Wiring levels	7	8	9	10	10	10
Power supply (V)	1.1	1.0	0.8	0.7	0.6	0.5
Max. power (W)	130	160	190	220	250	290

Factors contributing to the validity of Moore's law

- Denser circuits
- Architectural improvements

Measures of processor performance

- Instructions per second (MIPS, GIPS, TIPS, PIPS)
- Floating-point operations per second
  - (MFLOPS, GFLOPS, TFLOPS, PFLOPS)
- Running time on benchmark suites
  - Examples of benchmarks

Categories of supercomputers

- Uniprocessor (vector processor)
- Multiprocessor
- Multicomputer
- Massively parallel processor (MPP)

There is a limit to the speed of a single processor (the speed-of-light argument)

- Light travels 30 cm/ns;
  - signals on wires travel at a fraction of this speed
  - ($\cong c/E_r^{1/2}$, where $E_r \cong$ 2-4 is the dielectric coeff.)
- If signals *must* travel 1 cm in an instruction cycle,
  - cycle time cannot be shorter than 1/30 ns;
  - thus, 30 GIPS is the best we can hope for

Motivations for concurrency

1. Higher speed (solve problems faster)
   Important when there are "hard" or "soft" deadlines;
   e.g., 24-hour weather forecast

2. Higher throughput (solve more problems)
   Important when there are many similar tasks to perform;
   e.g., transaction processing

3. Higher computational power (solve larger problems)
   e.g., weather forecast for a week rather than 24 hours,
   or with a finer mesh for greater accuracy

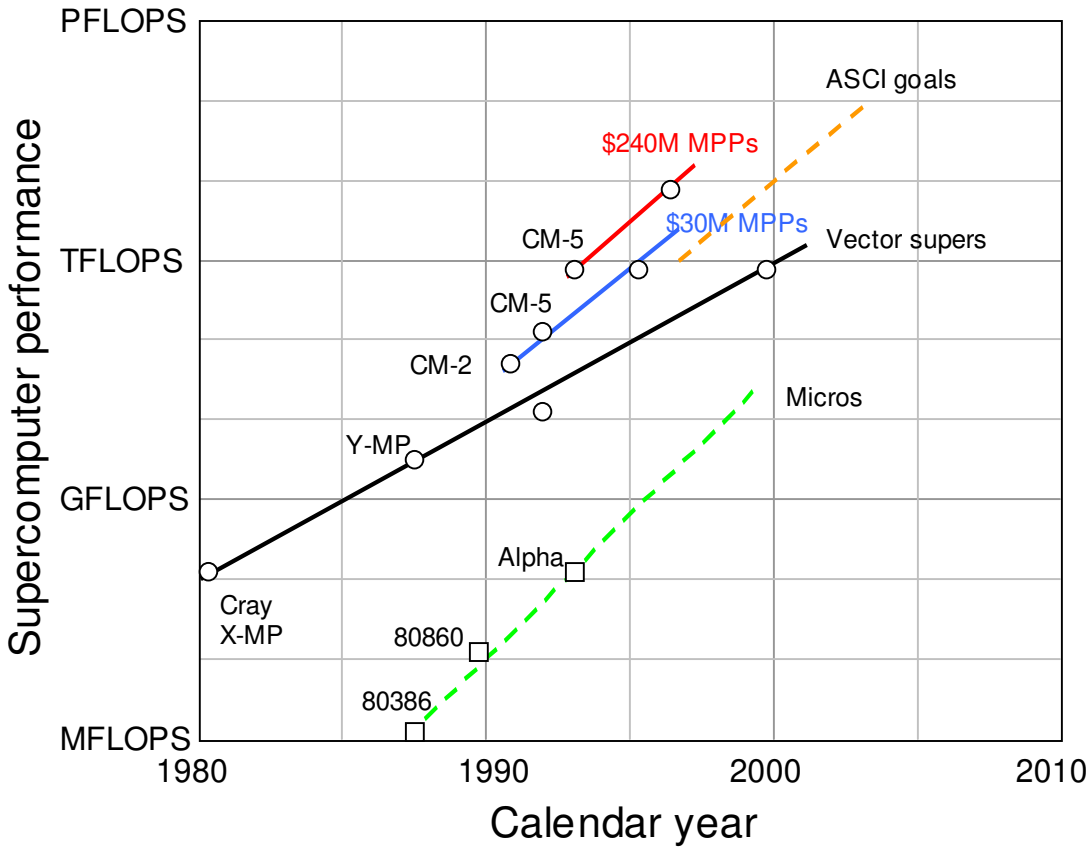


Fig. 1.2. **The exponential growth in supercomputer performance over the past two decades (from [Bell92], with ASCI performance goals and microprocessor peak FLOPS superimposed as dotted lines).**

The need for TFLOPS

Modeling of heat transport to the South Pole in the southern oceans [Ocean model: 4096 E-W regions $\times$ 1024 N-S regions $\times$ 12 layers in depth]

30 000 000 000 FLOP per 10-min iteration $\times$
300 000 iterations per six-year period =
10^{16} FLOP

Fluid dynamics

$1000 \times 1000 \times 1000$ lattice $\times$
1000 FLOP per lattice point $\times$ 10 000 time steps =
10^{16} FLOP

Monte Carlo simulation of nuclear reactor

100 000 000 000 particles to track (for $\cong$1000 escapes)
$\times$ 10 000 FLOP per particle tracked =
10^{15} FLOP

Reasonable running time =
Fraction of hour to several hours (10^3-10^4 s)

Computational power =
10^{16} FLOP / 10^4s or 10^{15} FLOP / 10^3s = 10^{12} FLOPS

Why the current quest for PFLOPS?

Same problems, perhaps with finer grids or longer simulated times

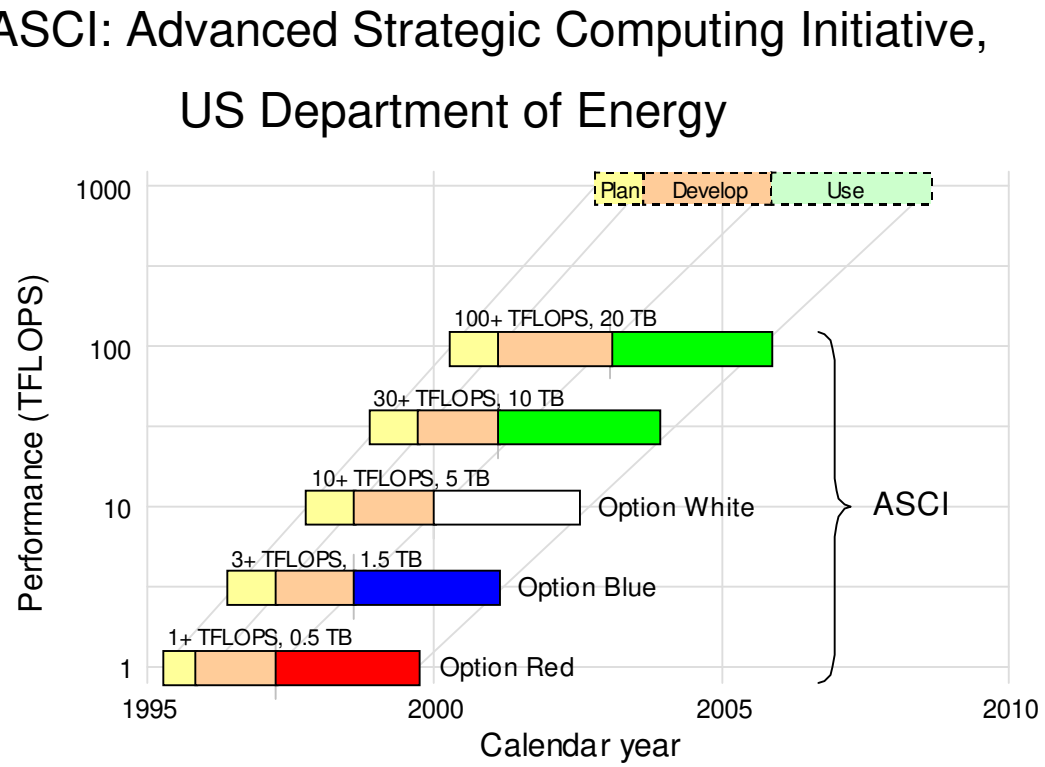


Fig. 24.1. Milestones in the Accelerated Strategic Computing Initiative (ASCI) program, sponsored by the US Department of Energy, with extrapolation up to the PFLOPS level.

Status of Computing Power (circa 2000)

GFLOPS on desktop

Apple Macintosh, with G4 processor

TFLOPS in supercomputer center

1152-processor IBM RS/6000 SP

uses a switch-based interconnection network

see *IEEE Concurrency*, Jan.-Mar. 2000, p. 9

Cray T3E, torus-connected

PFLOPS on drawing board

1M-processor IBM Blue Gene (2005?)

see IEEE Concurrency, Jan.-Mar. 2000, pp. 5-9

32 proc's/chip, 64 chips/board, 8 boards/tower, 64 towers

Processor: 8 threads, on-chip memory, no data cache

Chip: defect-tolerant, row/column rings in a 6×6 array

Board: 8×8 chip grid organized as $4 \times 4 \times 4$ cube

Tower: Boards linked to 4 neighbors in adjacent towers

System: $32 \times 32 \times 32$ cube of chips, 1.5 MW (water-cooled)

1.2 A Motivating Example

Sieve of Eratosthenes (ˌer-a-'taas-tha-neez)
for finding all primes in [1, *n*]

```
 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
m=2

 2  3     5     7     9    11    13    15    17    19    21    23    25    27    29
   m=3

 2  3     5     7          11    13          17    19          23    25          29
         m=5

 2  3     5     7          11    13          17    19          23                29
               m=7
```

Fig. 1.3. **The sieve of Eratosthenes yielding a list of 10 primes for *n* = 30. Marked elements have been distinguished by erasure from the list.**

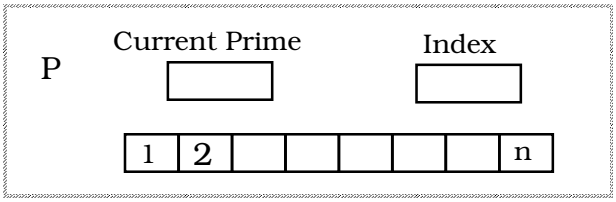


Fig. 1.4. **Schematic representation of single-processor solution for the sieve of Eratosthenes.**

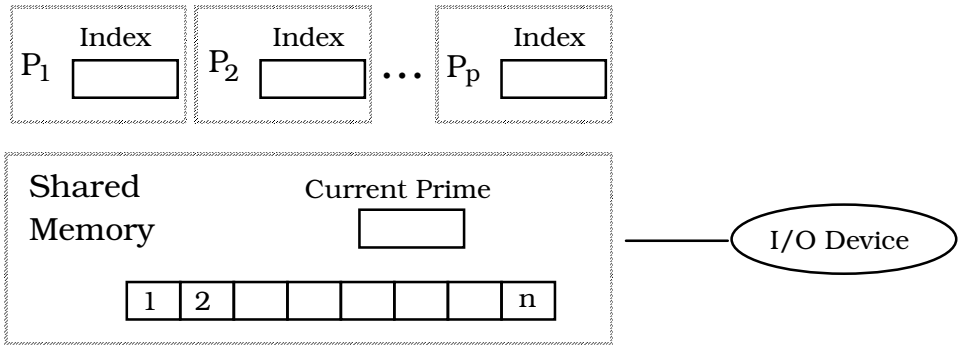


Fig. 1.5. Schematic representation of a control-parallel solution for the sieve of Eratosthenes.

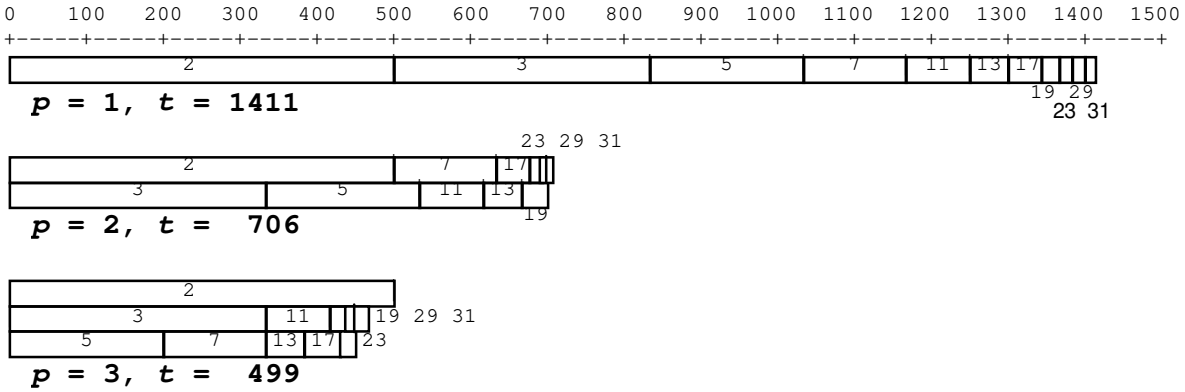


Fig. 1.6. Control-parallel realization of the sieve of Eratosthenes with $n = 1000$ and $1 \leq p \leq 3$.

P_1 finds each prime and broadcasts it to all other processors

Assume $n/p \geq \sqrt{n}$ ($p \leq \sqrt{n}$), so that all primes whose multiples are to be marked reside in P_1

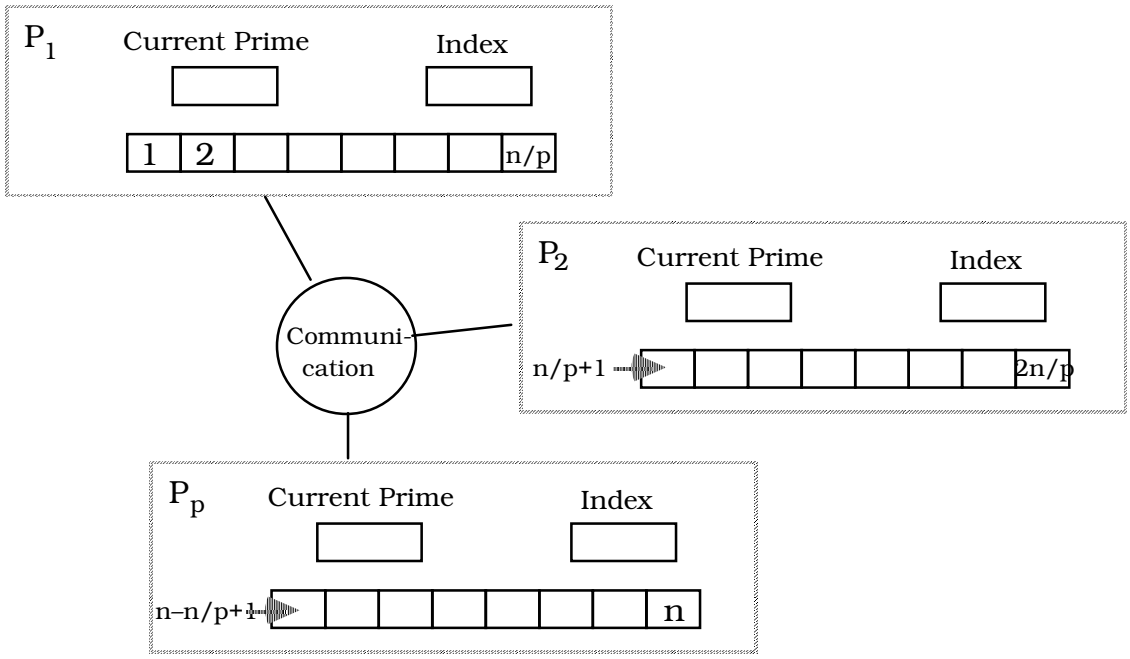


Fig. 1.7. Data-parallel realization of the sieve of Eratosthenes.

Some reasons for sublinear speed-up

Communication overhead

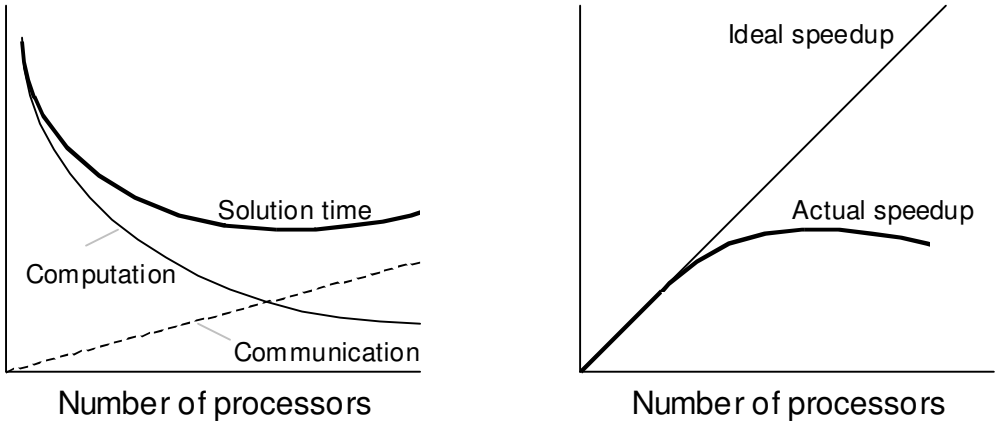


Fig. 1.8. Trade-off between communication time and computation time in the data-parallel realization of the sieve of Eratosthenes.

Input/output overhead

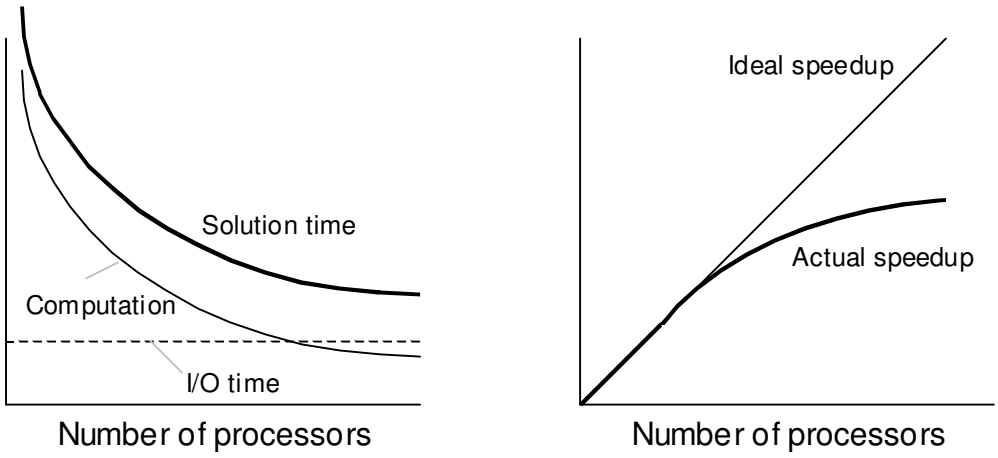


Fig. 1.9. Effect of a constant I/O time on the data-parallel realization of the sieve of Eratosthenes.

1.3 Parallel Processing Ups and Downs

Early 1900s: 1000s of "computers" (humans + calculators) to do 24-hour weather prediction in a few hours

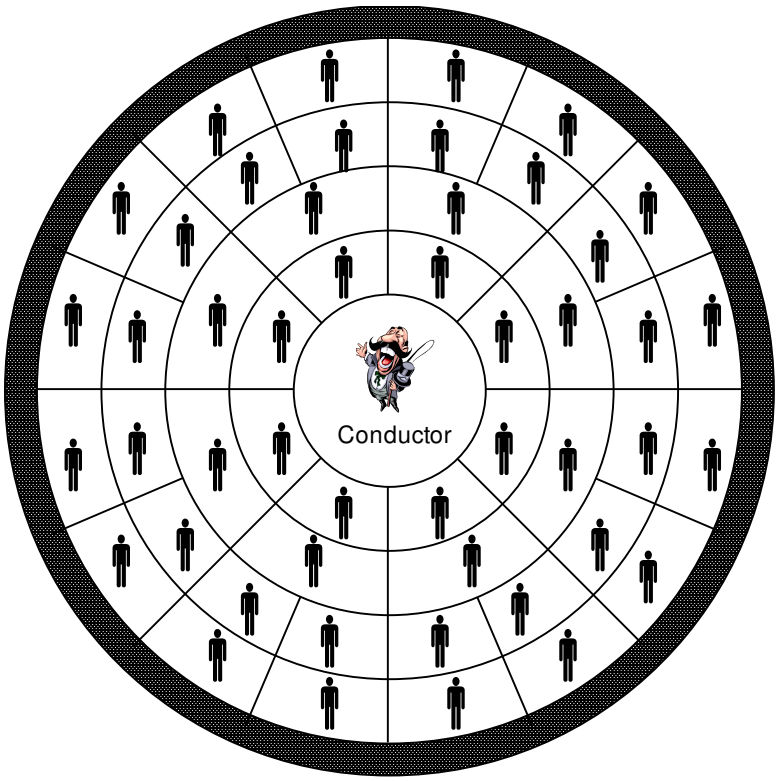


Fig. 1.10. Richardson's circular theater for weather forecasting calculations.

Parallel processing is used in virtually all computers

> Compute-I/O overlap, pipelining (fetch/exec overlap), multitasking, VLIW, multiple function units

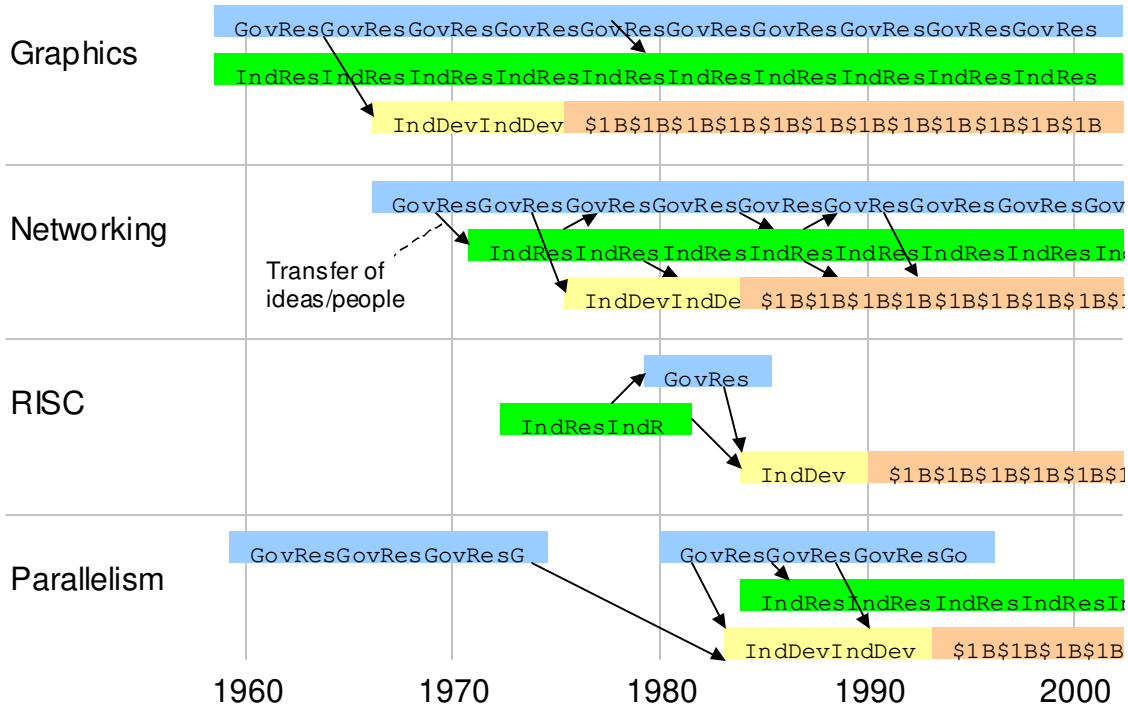
But ... in this course we use "parallel processing" in a stricter sense implying the availability of multiple CPUs

History of Parallel Processing

1960s: ILLIAC IV (U Illinois) – Four 8 × 8 mesh quadrants

1980s: Commercial interest resurfaced; technology was driven by government contracts. Once funding dried up, many companies went bankrupt

2000s: Internet revolution – info providers, multimedia, data mining, etc. need extensive computational power



Development of some technical fields into $1B businesses and the roles of government research and industrial R&D over time (*IEEE Computer*, early 90s?).

1.4 Types of Parallelism: A Taxonomy

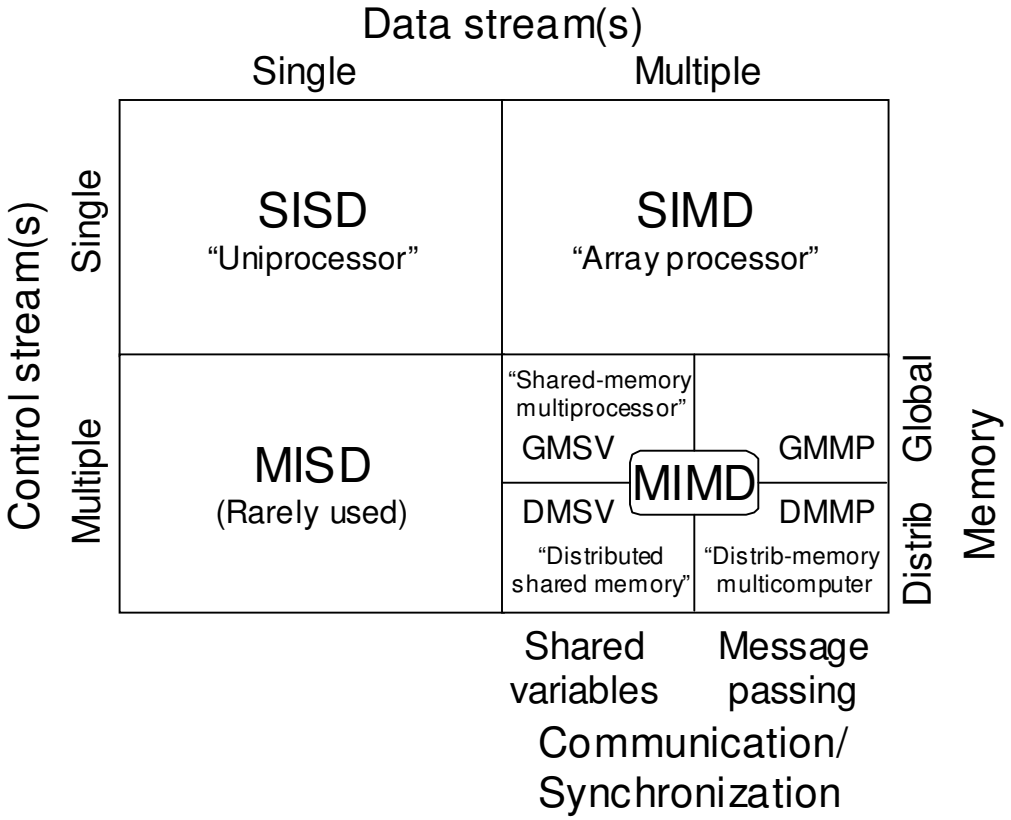


Fig. 1.11. The Flynn-Johnson classification of computer systems.

Why are computer architects so fascinated by four-letter acronyms and abbreviations?

Systems: RISC, CISC, PRAM, NUMA, VLIW

Journals: JPDC, TPDS

Conferences: ICPP, IPPS, SPDP, SPAA

My contribution:

SINC: Scant/Simple Interaction Network Cell

FINC: Full Interaction Network Cell

1.5 Roadblocks to Parallel Processing

a. Grosch's law (economy of scale applies, or computing power proportional to the square of cost)

Rebuttal: Not true any more. Even if it were, there is only one fastest computer; cannot get a faster one by spending more

b. Minsky's conjecture (speedup proportional to the logarithm of the number *p* of processors)

This is due to a statistical argument; you don't need a lot of people in a room to have some with identical birthdays (memory accesses will have conflicts)

Rebuttal: Just like the assumption of no conflict, and thus linear speedup, randomness is too pessimistic; perhaps $p/\log p$ is more realistic than either extreme

c. Tyranny of IC *technology* (since hardware becomes about 10 times faster every 5 years, by the time a parallel machine with 10-fold performance is built, uniprocessors will be just as fast)

Rebuttal: We might try to design parallel systems into which faster components can be incorporated as they become available. Also, we might aim for 100-fold or 1000-fold speedup, not just 10-fold

d. Tyranny of vector supercomputers
(vector supercomputers are rapidly improving in performance, offer a familiar programming model and excellent vectorizing compilers; why bother with parallel processors?)

Rebuttal: Many compute-intensive problems do not involve vector operations; besides, even vector machines nowadays use multiprocessing

e. Software inertia (Billions of dollars worth of existing software makes it hard to switch to parallel systems)

Rebuttal: Not all future applications have already been developed. Improved automatic tools can convert "dusty deck" programs into efficient parallel programs. Students are being trained to "think parallel"

f. Amdahl's law
(a small fraction *f* of inherently sequential
or unparallelizable computation
severely limits the speed-up)

$$\text{speedup} \leq \frac{1}{f + (1 - f)/p} = \frac{p}{1 + f(p - 1)}$$

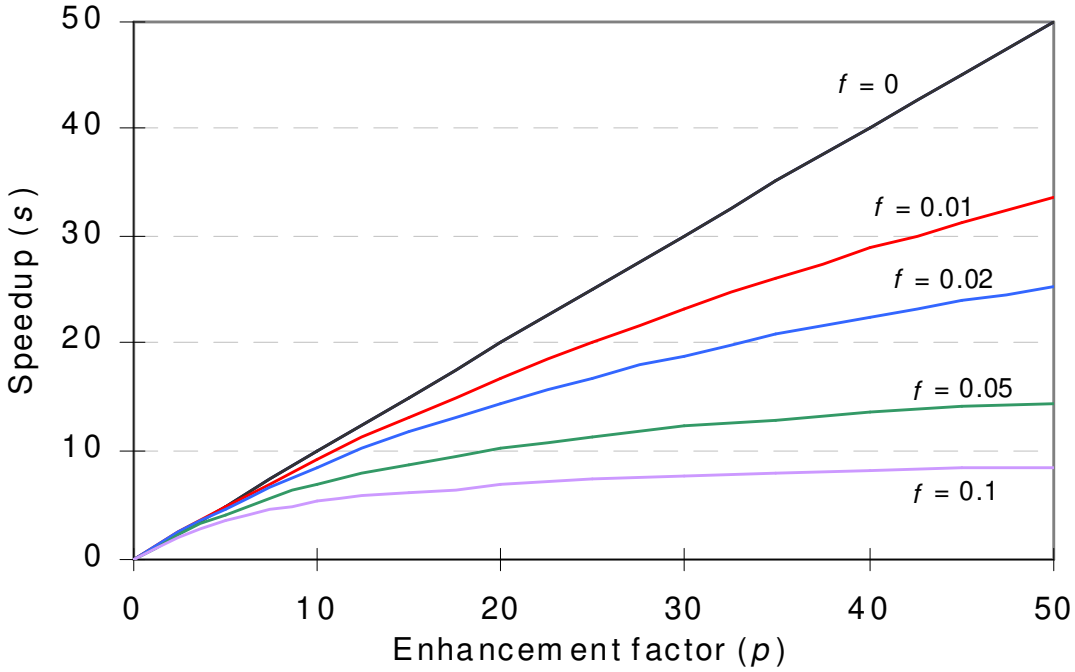


Fig. 1.12. Limit on speed-up according to Amdahl's law.

Rebuttal: Applications with very small *f* exist. Besides, sequential overhead need not be a fixed fraction

ABCs of Parallel Processing

in one transparency* (parhami@ece.ucsb.edu)

* Originally appeared in *Computer Architecture News*, Vol. 27, No. 1, p. 2, March 1999.

f = unparallelizable fraction of a task (sequential overhead)
T_x = running time of a task when executed on x processors

A Amdahl's Law (Speed-up Formula)

Bad news: Sequential overhead will kill you, since:

$$\text{Speed-up} = \frac{T_1}{T_p} \leq \frac{1}{f + \frac{1-f}{p}} \leq \min\left(\frac{1}{f}, p\right)$$

Morale: For $f = 0.1$, e.g., the speed-up will be at best 10, no matter what the number of processors (peak OPS).

B Brent's Scheduling Theorem

Good news: Optimal scheduling is a very difficult problem, but even a naive scheduling algorithm can ensure:

$$\frac{T_1}{p} \leq T_p < \frac{T_1}{p} + T_\infty = \frac{T_1}{p}\left(1 + \frac{p}{T_1/T_\infty}\right)$$

Result: For a reasonably parallel task (with small T_∞), or for a suitably small number of processors (say, $p < T_1/T_\infty$), good speed-up and high utilization are attainable.

C Cost-Effectiveness Adage

Real news: The most cost-effective parallel solution to a given problem is often not the one with:

Highest peak OPS	(communication can kill you)
Greatest speed-up	(at what cost?)
Best utilization	(hardware busy doing what?)

Analogy: Mass transit (SIMD) might be more cost-effective than using private vehicles (MIMD) even if it is slower and leads to many empty seats on some trips.

1.6 Effectiveness of Parallel Processing

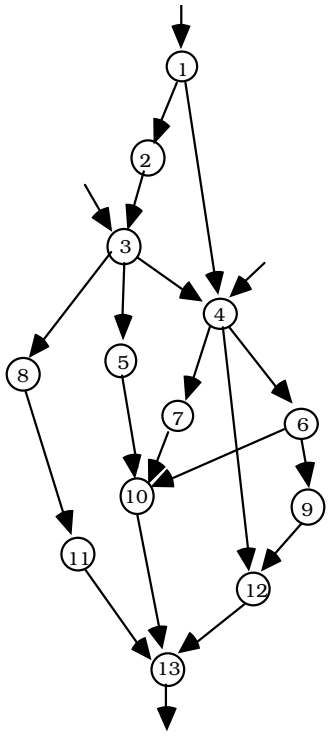


Fig. 1.13. Task graph exhibiting limited inherent parallelism.

Measures for comparing parallel architectures/algorithms:

p Number of processors

$W(p)$ Total number of unit operations performed by p processors; computational work or energy

$T(p)$ Execution time with p processors; $T(1) = W(1)$ and $T(p) \leq W(p)$

$S(p)$ Speedup $= \dfrac{T(1)}{T(p)}$

$E(p)$ Efficiency $= \dfrac{T(1)}{pT(p)}$

$R(p)$ Redundancy $= \dfrac{W(p)}{W(1)}$

$U(p)$ Utilization $= \dfrac{W(p)}{pT(p)}$

$Q(p)$ Quality $= \dfrac{T^3(1)}{pT^2(p)W(p)}$

Relationships among the preceding measures:

$1 \leq S(p) \leq p$

$U(p) = R(p)E(p)$

$E(p) = \dfrac{S(p)}{p}$

$Q(p) = E(p)\dfrac{S(p)}{R(p)}$

$\dfrac{1}{p} \leq E(p) \leq U(p) \leq 1$

$1 \leq R(p) \leq \dfrac{1}{E(p)} \leq p$

$Q(p) \leq S(p) \leq p$

Example: Adding 16 numbers, assuming unit-time additions and ignoring all else, with $p = 8$

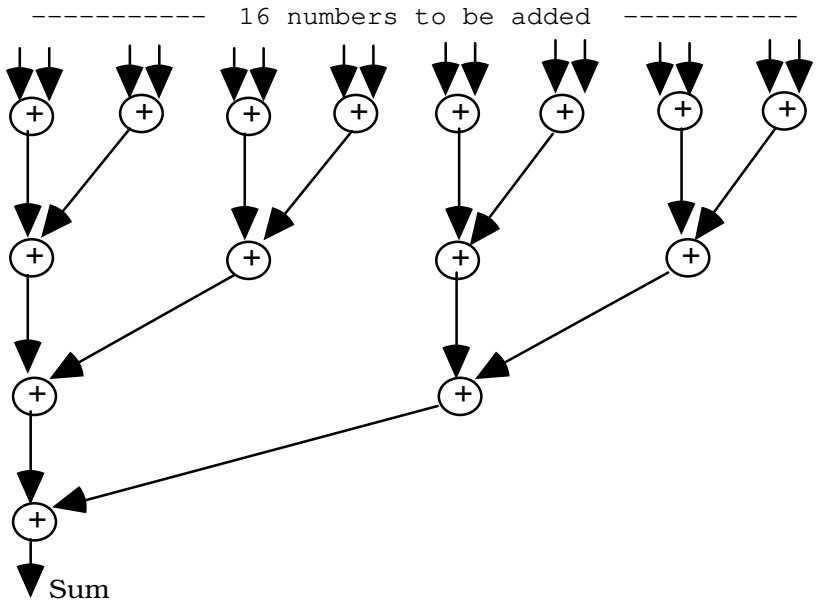


Fig. 1.14. Computation graph for finding the sum of 16 numbers.

Zero-time communication: $W(8) = 15$ $T(8) = 4$

$E(8) = 15 / (8 \times 4) = 47\%$

$S(8) = 15 / 4 = 3.75$ $R(8) = 15/15 = 1$ $Q(8) = 1.76$

Unit-time communication: $W(8) = 22$ $T(8) = 7$

$E(8) = 15 / (8 \times 7) = 27\%$

$S(8) = 15 / 7 = 2.14$ $R(8) = 22 / 15 = 1.47$ $Q(8) = 0.39$

2 A Taste of Parallel Algorithms

Back to TOC

Chapter Goals

- Consider five basic building-block parallel operations
- Implement them on four simple parallel architectures
- Learn about the nature of parallel computations, complexity analysis, and the algorithm/architecture interplay

Chapter Contents

- 2.1. Some Simple Computations
- 2.2. Some Simple Architectures
- 2.3. Algorithms for a Linear Array
- 2.4. Algorithms for a Binary Tree
- 2.5. Algorithms for a 2D Mesh
- 2.6. Algorithms with Shared Variables

2.1 Some Simple Computations

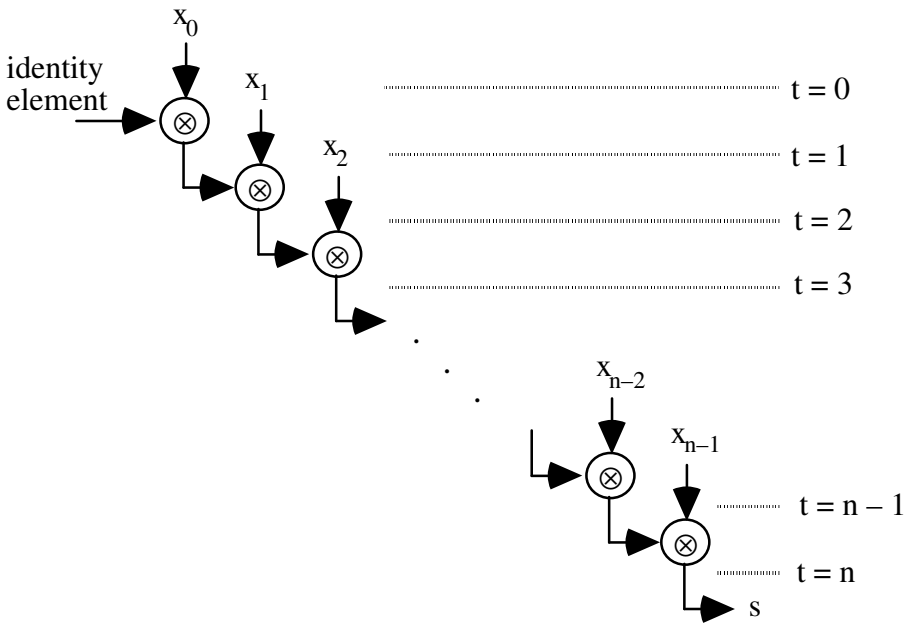
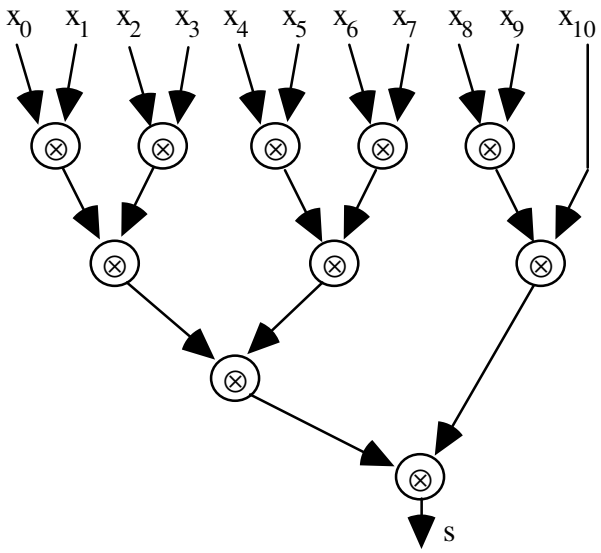
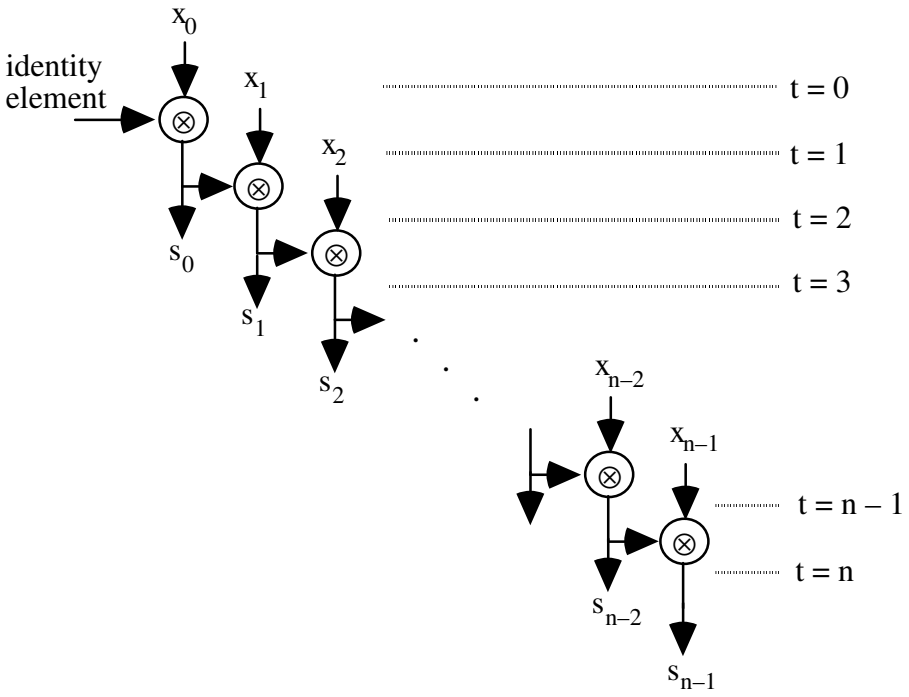


Fig. 2.1. Semigroup computation on a uniprocessor.



Semigroup computation viewed as a tree or fan-in computation.



Prefix computation on a uniprocessor.

3. Packet routing

   one processor sending a packet of data to another

4. Broadcasting

   one processor sending a packet of data to all others

5. Sorting

   processors cooperating in rearranging their data into desired order

2.2 Some Simple Architectures

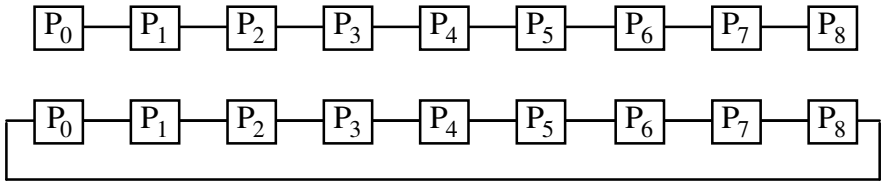


Fig. 2.2. A linear array of nine processors and its ring variant.

Diameter of linear array: $D = p - 1$

(Max) Node degree: $d = 2$

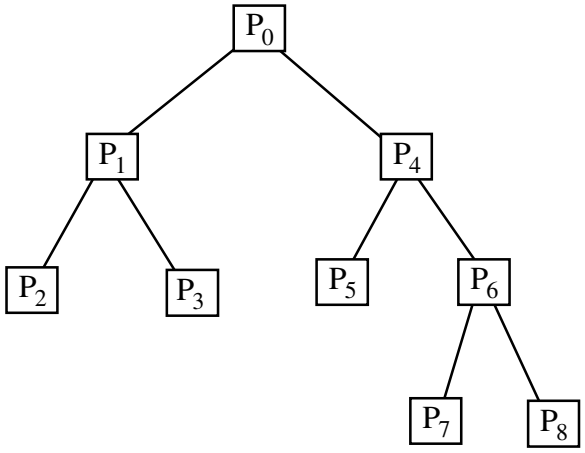


Fig. 2.3. A balanced (but incomplete) binary tree of nine processors.

Diameter of balanced binary tree: $D = 2 \lfloor \log_2 p \rfloor$; or 1 less

(Max) Node degree: $d = 3$

We almost always deal with complete binary trees:

p one less than a power of 2, $D = 2 \log_2(p + 1) - 2$

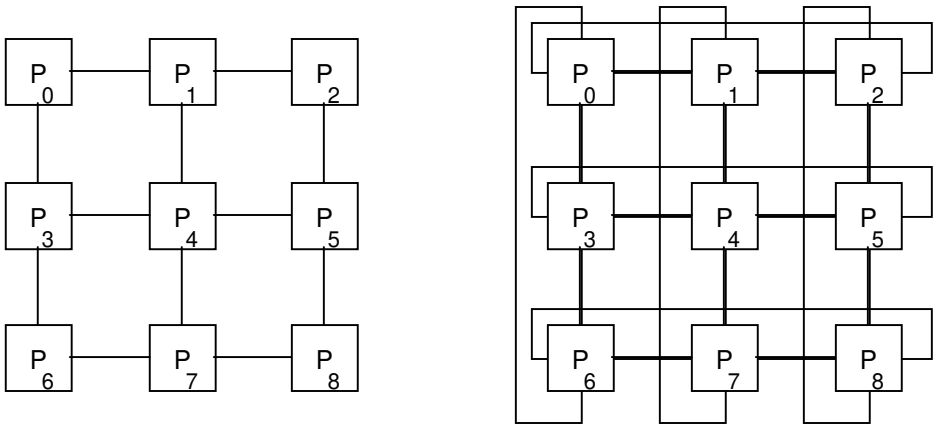


Fig. 2.4. 2D mesh of 9 processors and its torus variant.

Diameter of $r \times (p/r)$ mesh: $D = r + p/r - 2$

(Max) Node degree: $d = 4$

Square meshes preferred; they minimize D $(= 2\sqrt{p} - 2)$

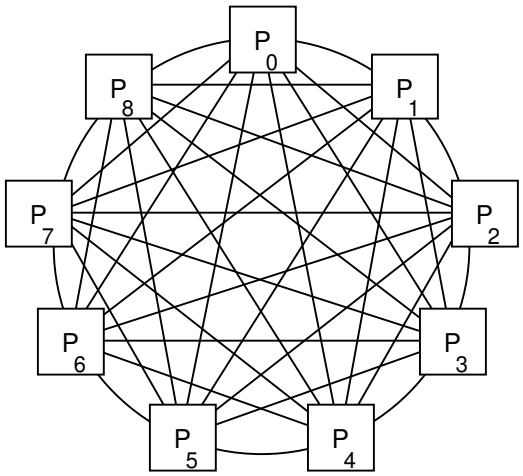


Fig. 2.5. A shared-variable architecture modeled as a complete graph.

Diameter of complete graph: $D = 1$

(Max) Node degree: $d = p - 1$

2.3 Algorithms for a Linear Array

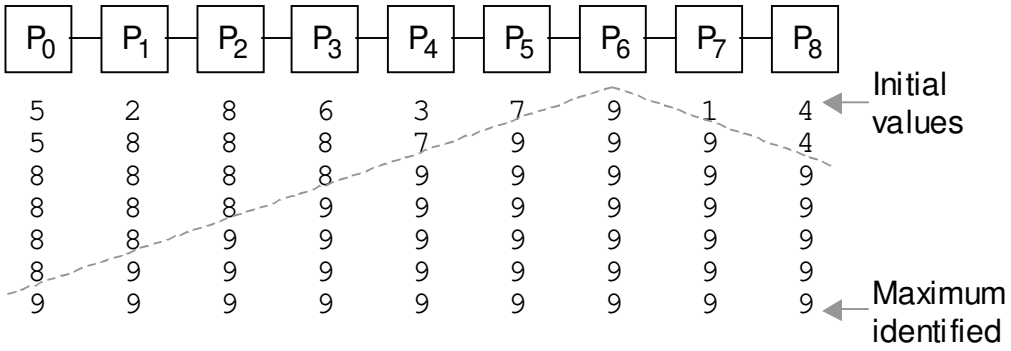


Fig. 2.6. Maximum-finding on a linear array of nine processors.

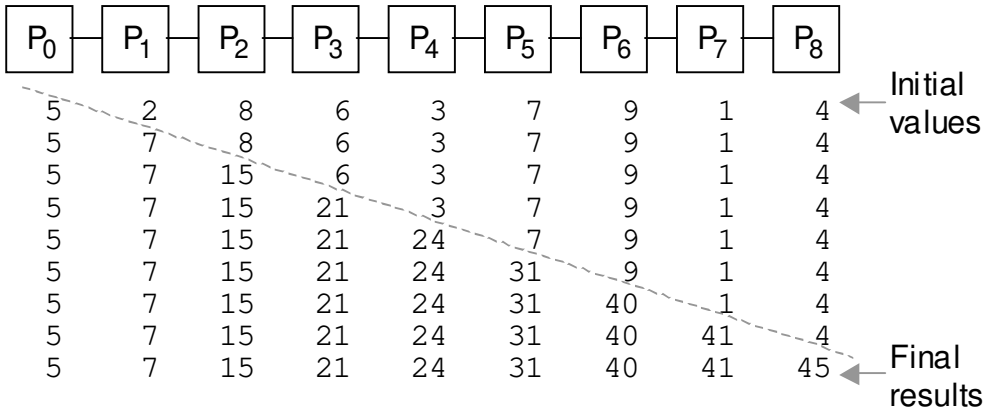


Fig. 2.7. Computing prefix sums on a linear array of nine processors.

Diminished prefix computation: the *i*th result excludes the *i*th element (e.g., sum of the first $i - 1$ elements)

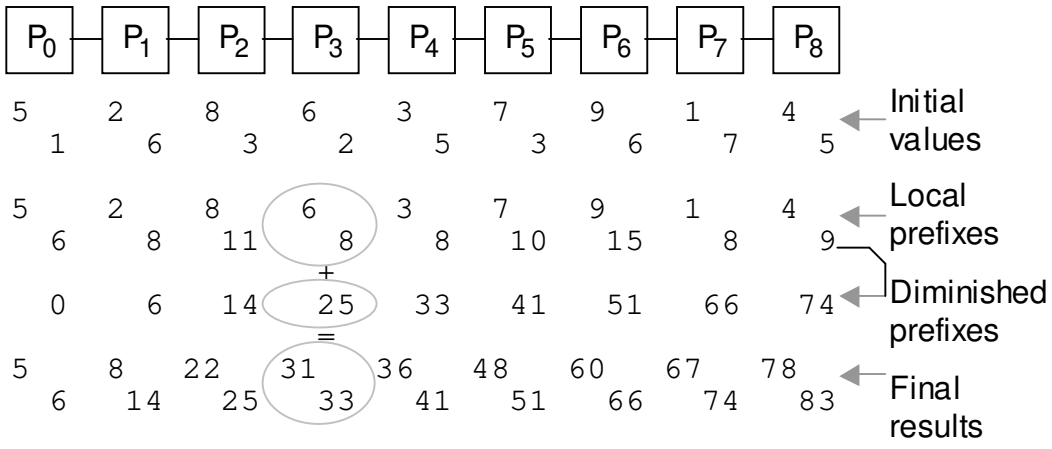
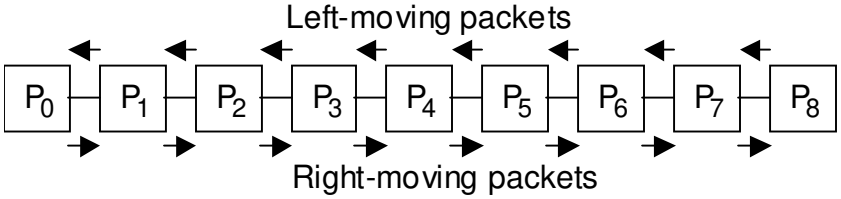


Fig. 2.8. Computing prefix sums on a linear array with two items per processor.

Packet routing or broadcasting:

right- and left-moving packets have no conflict



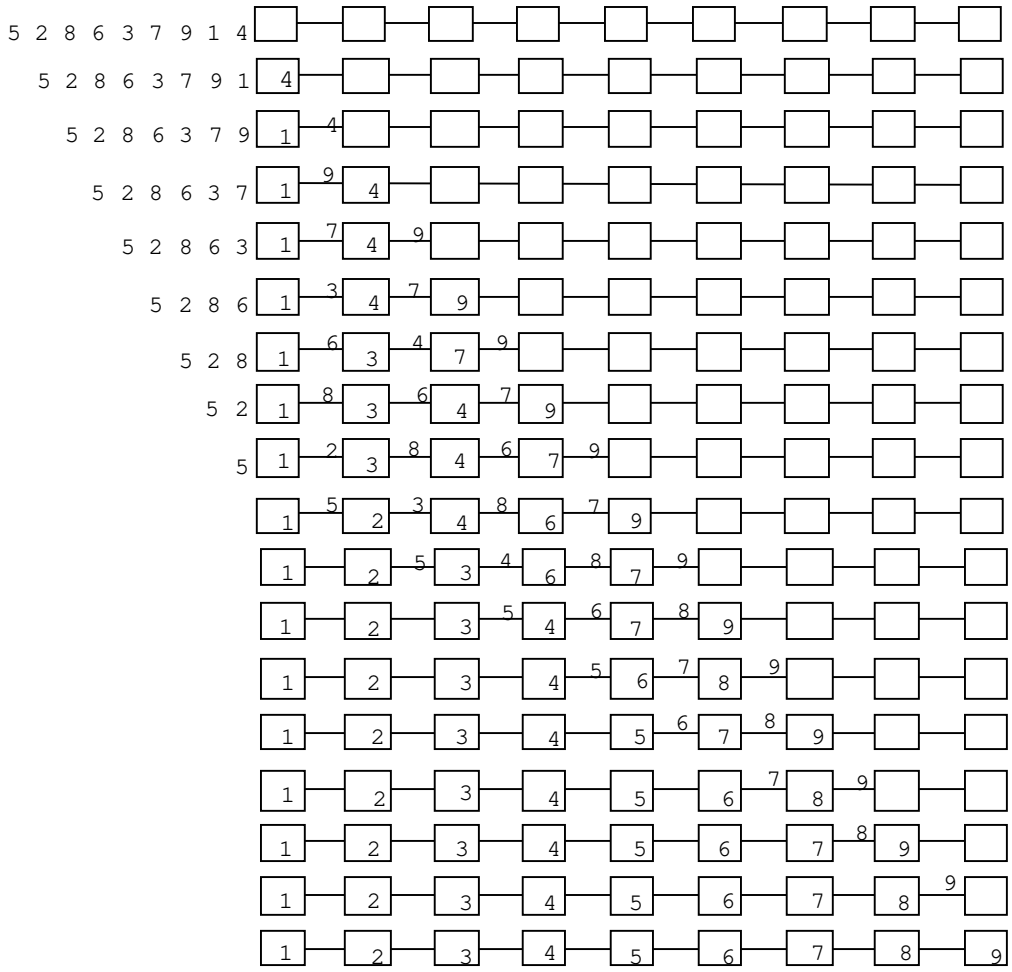


Fig. 2.9. Sorting on a linear array with the keys input sequentially from the left.

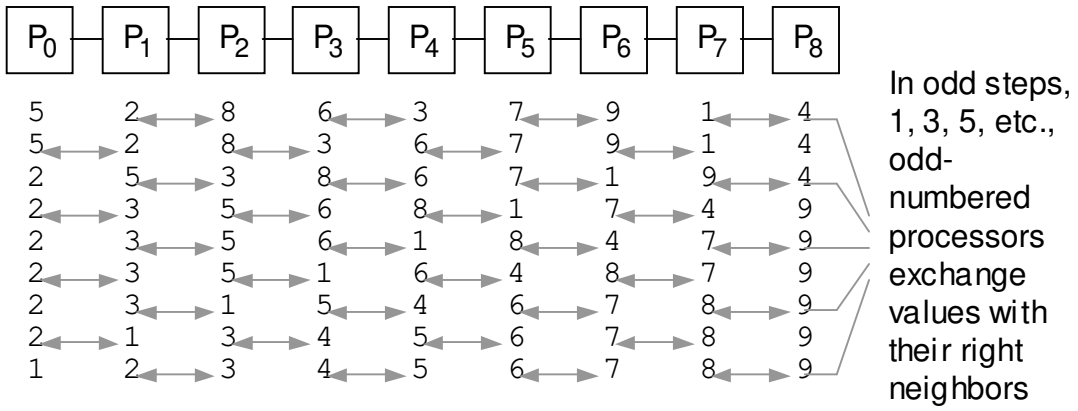


Fig. 2.10. Odd-even transposition sort on a linear array.

For odd-even transposition sort:

Speed-up = $O(p \log p) / p = O(\log p)$

Efficiency = $O((\log p) / p)$

Redundancy = $O(p / (\log p))$

Utilization = 1/2

2.4 Algorithms for a Binary Tree

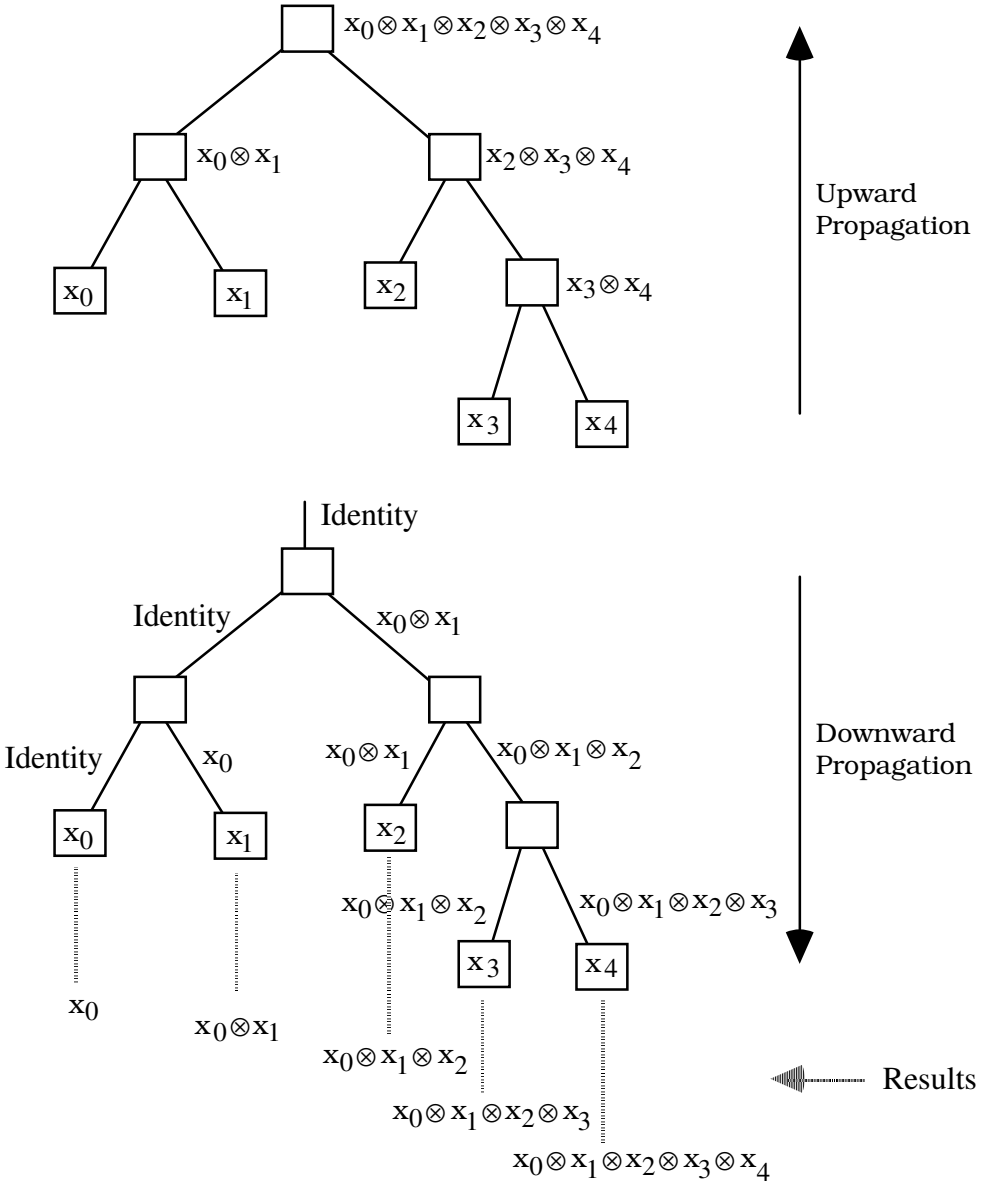


Fig. 2.11. Parallel prefix computation on a binary tree of processors.

Some applications of the parallel prefix computation

Finding the rank of each 1 in a list of 0s and 1s:

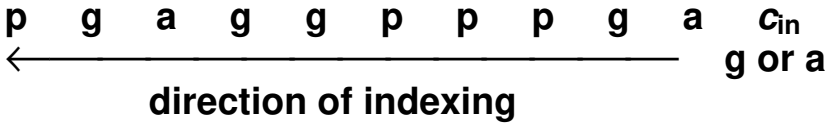
Data:	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**0**
Prefix sums:	**0**	**0**	**1**	**1**	**2**	**2**	**2**	**3**	**4**	**5**	**5**
Ranks of 1s:			**1**		**2**			**3**	**4**	**5**	

Priority circuit:

Data:	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**0**
Dim'd prefix ORs:	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Complement:	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
AND with data:	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Carry computation in fast adders

Let "g", "p", and "a" denote the event that a particular digit position in the adder generates, propagates, or annihilates a carry. The input data for the carry circuit consists of a vector of three-valued elements such as:



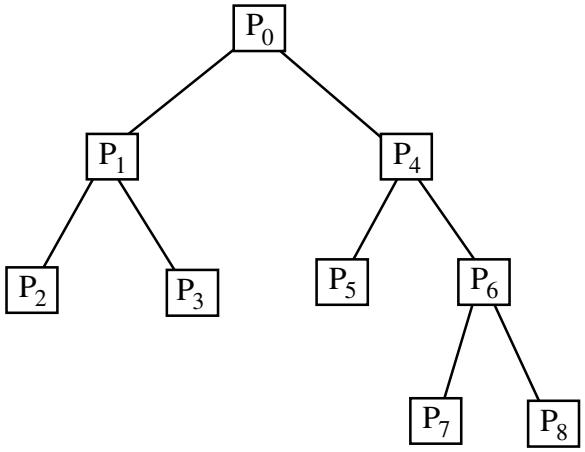
Parallel prefix computation using the carry operator "¢"

p ¢ *x* = *x* ***x* propagates over p, for all *x* ∈ {g, p, a}**

a ¢ *x* = a ***x* is annihilated or absorbed by a**

g ¢ *x* = g ***x* is immaterial; a carry is generated**

Packet routing on a tree

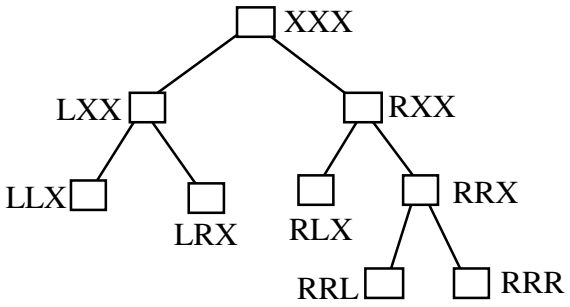


A balanced binary tree with preorder node indices.

maxl (*maxr*) = largest node number in left (right) subtree

```
if      dest = self
then  remove the packet {done}
else  if  dest < self   or  dest > maxr
        then  route upward
        else  if dest ≤ maxl
                then  route leftward
                else  route rightward
                endif
        endif
endif
```

Other indexing schemes might lead to simpler routing algorithms



Broadcasting is done via the root node

Sorting: let the root "see" all data in nondescending order

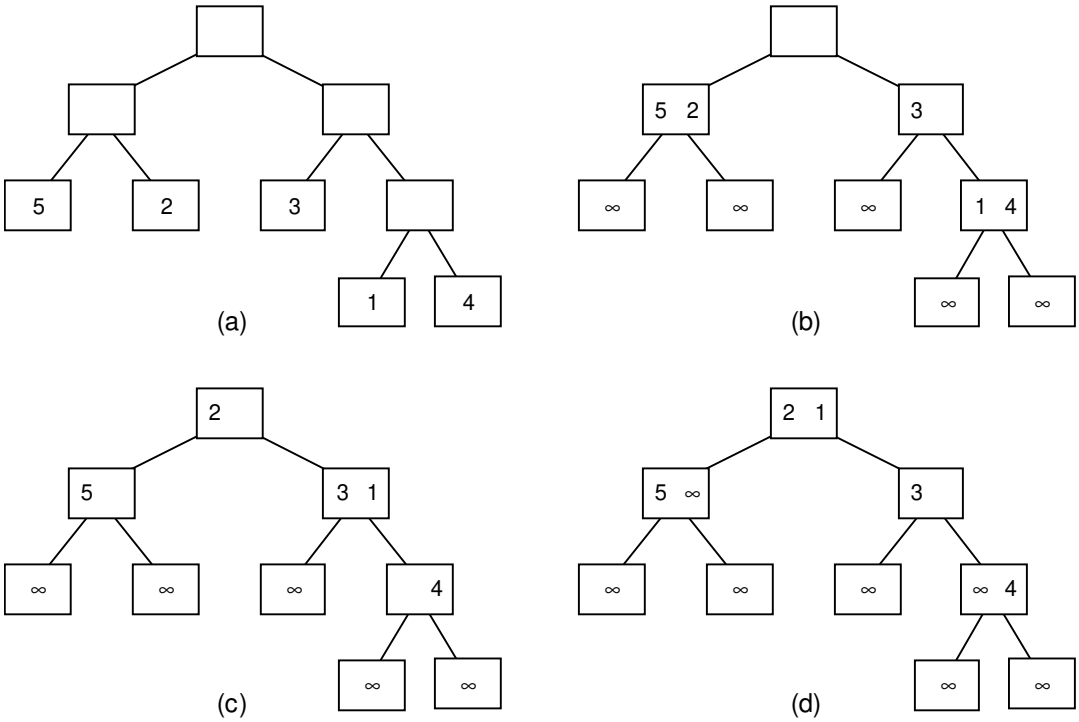


Fig. 2.12. The first few steps of the sorting algorithm on a binary tree.

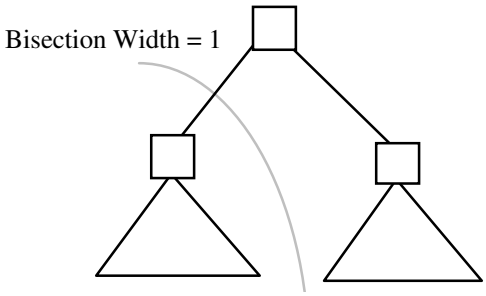
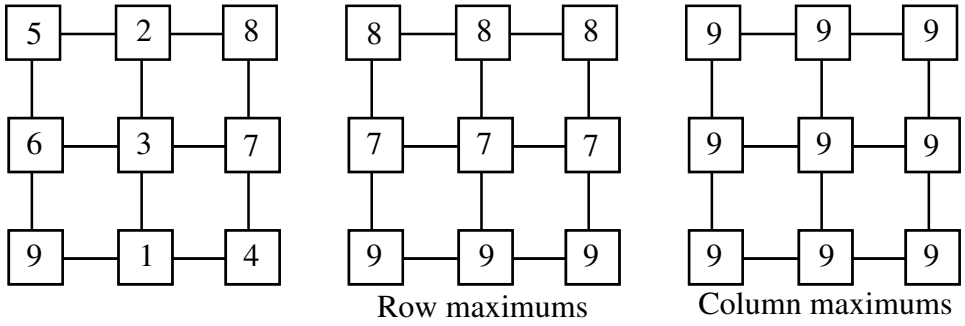
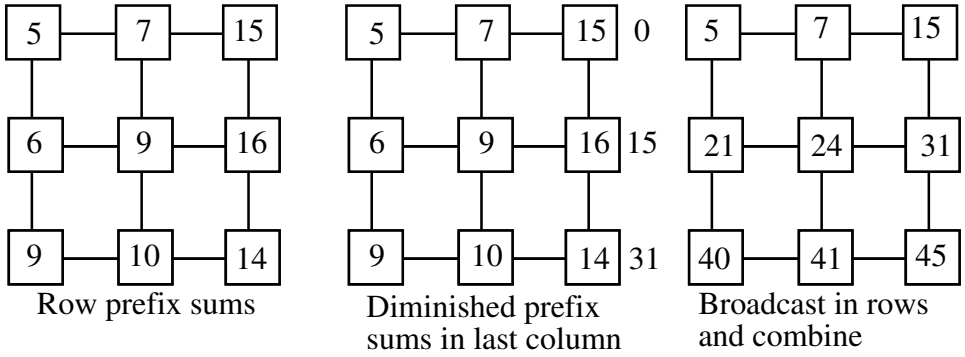


Fig. 2.13. The bisection width of a binary tree architecture.

2.5 Algorithms for a 2D Mesh



Finding the max value on a 2D mesh



Computing prefix sums on a 2D mesh

Row-major order required if operator not commutative

Routing and broadcasting done via row/column operations

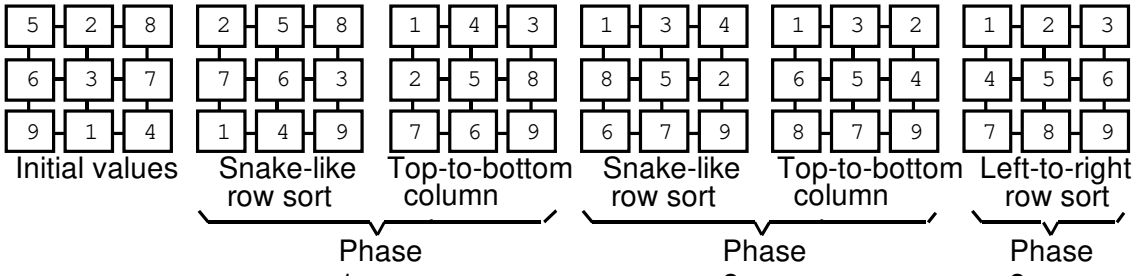


Fig. 2.14. The shearsort algorithm on a 3×3 mesh.

2.6 Algorithms with Shared Variables

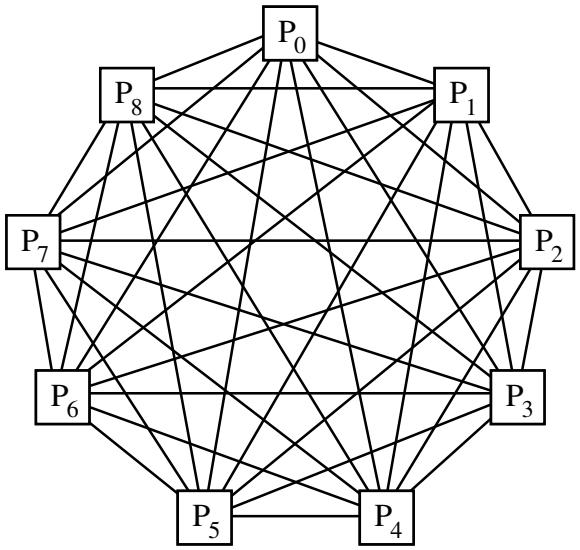


Fig. 2.5. A shared-variable architecture modeled as a complete graph.

Semigroup computation: each processor read all values in turn and combine

Parallel prefix: processor *i* read/combine values 0 to *i* – 1

Both of the above are quite inefficient, given the high cost

Routing/broadcasting: 1 step, with all-port communication

Sorting: rank each element by comparing it to all others, then permute according to ranks

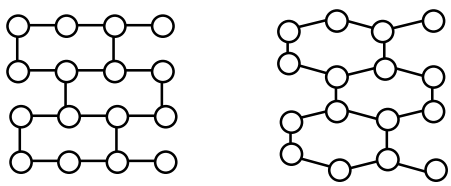


Figure for Problem 2.13.

3 Parallel Algorithm Complexity

Back to TOC

Chapter Goals

- Review algorithm complexity and various complexity classes
- Introduce the notions of time and time-cost optimality
- Derive tools for analyzing, comparing, and fine-tuning parallel algorithms

Chapter Contents

- 3.1. Asymptotic Complexity
- 3.2. Algorithm Optimality and Efficiency
- 3.3. Complexity Classes
- 3.4. Parallelizable Tasks and the NC Class
- 3.5. Parallel Programming Paradigms
- 3.6. Solving Recurrences

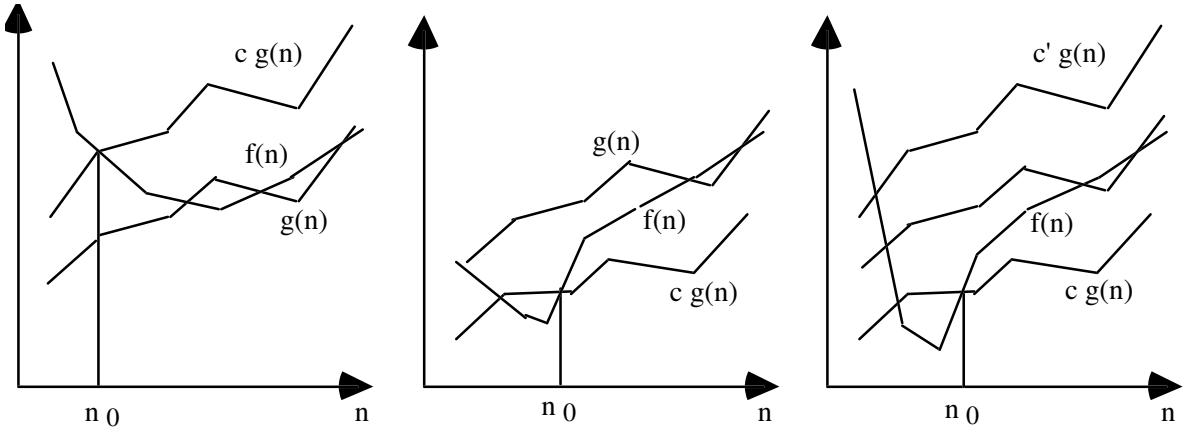
3.1 Asymptotic Complexity

$f(n) = O(g(n))$ if $\exists c, n_0$ such that $\forall n > n_0, f(n) < c\, g(n)$

$f(n) = \Omega(g(n))$ if $\exists c, n_0$ such that $\forall n > n_0, f(n) > c\, g(n)$

$f(n) = \Theta(g(n))$ if $\exists c, c', n_0$ such that

$$\forall n > n_0, cg(n) < f(n) < c'g(n)$$



***f*(*n*) = O(*g*(*n*))** ***f*(*n*) = Ω(*g*(*n*))** ***f*(*n*) = Θ(*g*(*n*))**

Fig. 3.1. Graphical representation of the notions of asymptotic complexity.

$f(n) = o(g(n))$	$<$	Growth rate strictly less than
$f(n) = O(g(n))$	$\leq$	Growth rate no greater than
$f(n) = \Theta(g(n))$	$=$	Growth rate the same as
$f(n) = \Omega(g(n))$	$\geq$	Growth rate no less than
$f(n) = \omega(g(n))$	$>$	Growth rate strictly greater than

Table 3.1. Comparing the Growth Rates of Sublinear and Superlinear Functions (K = 1000, M = 1 000 000)

Sublinear		Linear	Superlinear	
$\log^2 n$	$\sqrt{n}$	n	$n \log^2 n$	$n^{3/2}$
9	3	10	90	30
36	10	100	3.6K	1K
81	31	1K	81K	31K
169	100	10K	1.7M	1M
256	316	100K	26M	32M
361	1K	1M	361M	1000M

Table 3.2. Effect of Constants on the Growth Rates of Selected Functions Involving Constant Factors

n	$\frac{n}{4} \log^2 n$	$n \log^2 n$	$100\sqrt{n}$	$n^{3/2}$
10	22	90	300	30
100	900	3.6K	1K	1K
1K	20K	81K	3.1K	31K
10K	423K	1.7M	10K	1M
100K	6M	26M	32K	32M
1M	90M	361M	100K	1000M

Table 3.3. Effect of Constants on the Growth Rates of Selected Functions Using Larger Time Units and Round Figures

n	$\frac{n}{4} \log^2 n$	$n \log^2 n$	$100\sqrt{n}$	$n^{3/2}$
10	20 s	2 min	5 min	30 s
100	15 min	1 hr	15 min	15 min
1K	6 hr	1 day	1 hr	9 hr
10K	5 days	20 days	3 hr	10 days
100K	2 mo	1 yr	9 hr	1 yr
1M	3 yr	11 yr	1 day	32 yr

3.2 Algorithm Optimality and Efficiency

$f(n)$ Running time of fastest (possibly unknown) algorithm for solving a problem

$g(n)$ Running time of some algorithm A $\Rightarrow f(n) = O(g(n))$

$h(n)$ Min time for solving the problem $\Rightarrow f(n) = \Omega(h(n))$

$g(n) = h(n) \Rightarrow$ Algorithm A is time-optimal

Redundancy = Utilization = 1 $\Rightarrow$ A is cost-time optimal

Redundancy = Utilization = $\Theta(1) \Rightarrow$ A is cost-time efficient

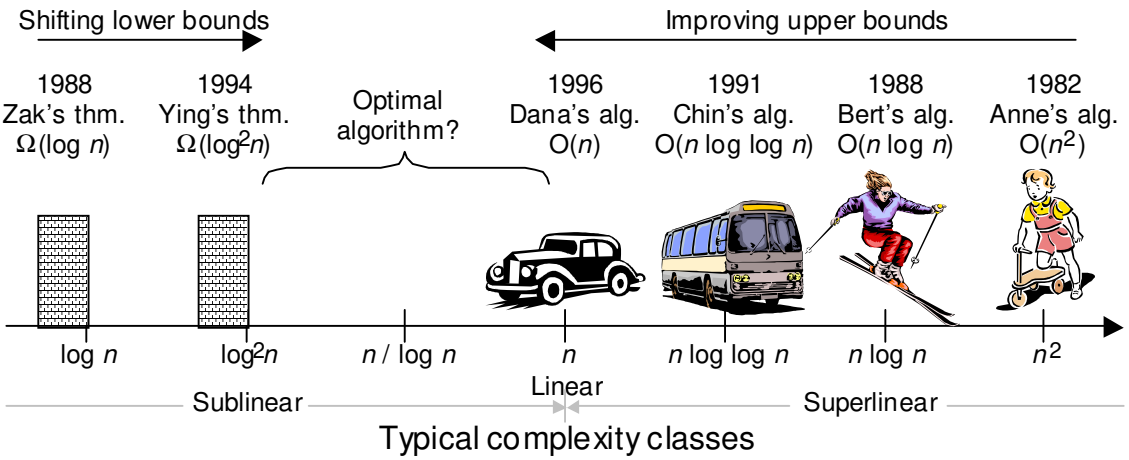


Fig. 3.2. Upper and lower bounds may tighten over time.

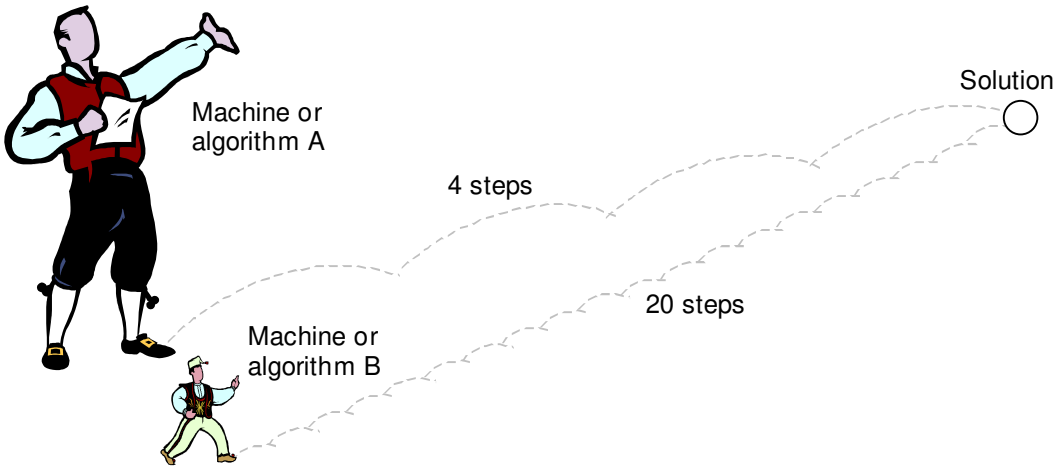
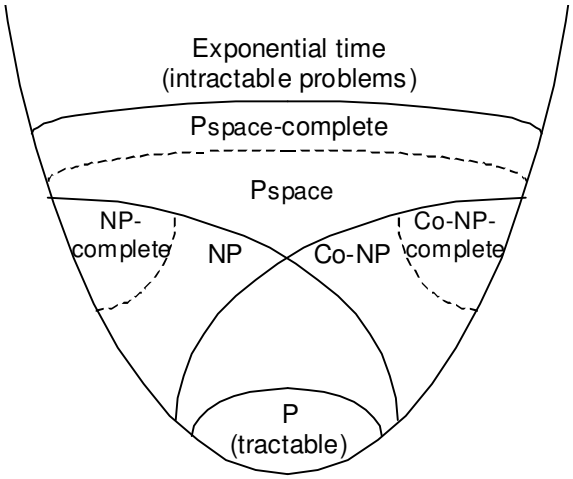
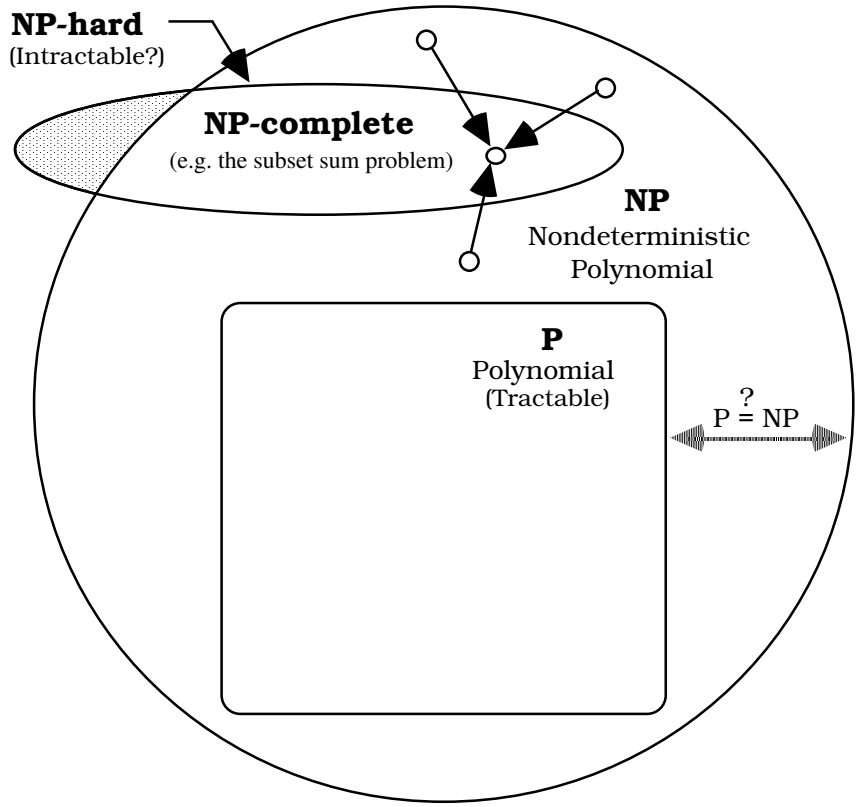


Fig. 3.3. Five times fewer steps does not necessarily mean five times faster.



Alternate, more detailed, form of the "complexity classes" diagram for Section 3.3.

3.3 Complexity Classes



Conceptual view of the P, NP, NP-complete, and NP-hard classes.

Example NP(-complete) problem: the subset sum problem

Given a set of n integers and a target sum s, determine if a subset of the integers add up to s.

The subset sum problem looks deceptively simple, yet no one knows how to solve it other than by trying practically all of the 2^n subsets of the given set.

Even if each trial takes only one picosecond (10^{-12} s), the problem is virtually unsolvable for $n = 100$.

3.4 Parallelizable Tasks and the NC Class

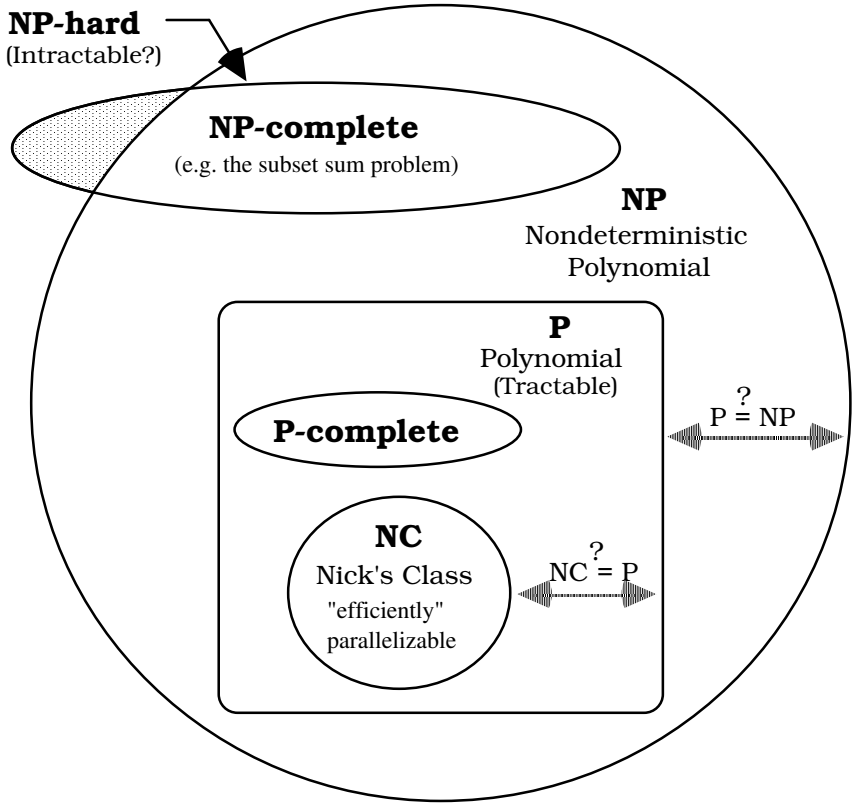


Fig. 3.4. A conceptual view of complexity classes and their relationships.

NC (Nick's class, Niclaus Pippenger)

Problems solvable in polylog time ($T = O(\log^k n)$)
using a polynomially bounded number of processors

Example P-complete problem: the circuit-value problem

For a logic circuit with known inputs, find its output
The circuit-value problem is obviously in P,
but no general algorithm exists for
efficient parallel evaluation of a circuit's output.

3.5 Parallel Programming Paradigms

Divide and conquer

Decompose problem of size n into smaller problems
Solve the subproblems independently
Combine subproblem results into final answer

$$T(n) = \underbrace{T_d(n)}_{\text{Decompose}} + \underbrace{T_s}_{\text{Solve in parallel}} + \underbrace{T_c(n)}_{\text{Combine}}$$

Randomization

Often it is impossible or difficult to decompose a large problem into subproblems with equal solution times.

In these cases, one might use random decisions that lead to good results with very high probability.

Example: sorting with random sampling

Other forms of randomization: Random search, control randomization, symmetry breaking

Approximation

Iterative numerical methods often use approximation to arrive at the solution(s).

Example: Solving linear systems using Jacobi relaxation.

Under proper conditions, the iterations converge to the correct solutions; more iterations $\Rightarrow$ greater accuracy

3.6 Solving Recurrences

Solution via unrolling

1. $f(n) = f(n-1) + n$ {rewrite $f(n-1)$ as $f((n-1)-1) + n - 1$}
$= f(n-2) + n - 1 + n$
$= f(n-3) + n - 2 + n - 1 + n$
...
$= f(1) + 2 + 3 + \cdots + n - 1 + n$
$= n(n+1)/2 - 1$
$= \Theta(n^2)$

2. $f(n) = f(n/2) + 1$ {Rewrite $f(n/2)$ as $f((n/2)/2 + 1)$}
$= f(n/4) + 1 + 1$
$= f(n/8) + 1 + 1 + 1$
. . .
$= f(n/n) + 1 + 1 + 1 + \cdots + 1$
------ $\log_2 n$ times ------
$= \log_2 n$
$= \Theta(\log n)$

3. $f(n) = 2f(n/2) + 1$
$= 4f(n/4) + 2 + 1$
$= 8f(n/8) + 4 + 2 + 1$
. . .
$= n\, f(n/n) + n/2 + \cdots + 4 + 2 + 1$
$= n - 1$
$= \Theta(n)$

4. $f(n) = f(n/2) + n$
$= f(n/4) + n/2 + n$
$= f(n/8) + n/4 + n/2 + n$
. . .
$= f(n/n) + 2 + 4 + \cdots + n/4 + n/2 + n$
$= 2n - 2 = \Theta(n)$

5. $f(n) = 2f(n/2) + n$
$= 4f(n/4) + n + n$
$= 8f(n/8) + n + n + n$
. . .
$= n\, f(n/n) + n + n + n + \cdots + n$
------ $\log_2 n$ times ------
$= n \log_2 n = \Theta(n \log n)$

Alternate solution for the recurrence $f(n) = 2f(n/2) + n$:

Rewrite the recurrence as $\frac{f(n)}{n} = \frac{f(n/2)}{n/2} + 1$

and denote $f(n)/n$ by $h(n)$ to convert the problem to Example 2

6. $f(n) = f(n/2) + \log_2 n$
$= f(n/4) + \log_2(n/2) + \log_2 n$
$= f(n/8) + \log_2(n/4) + \log_2(n/2) + \log_2 n$
. . .
$= f(n/n) + \log_2 2 + \log_2 4 + \cdots + \log_2(n/2) + \log_2 n$
$= 1 + 2 + 3 + \cdots + \log_2 n$
$= \log_2 n\, (\log_2 n + 1)/2 = \Theta(\log^2 n)$

Solution via guessing

Guess the solution and verify it by substitution

Substitution also useful to find the constant multiplicative factors and lower-order terms

Example: $f(n) = f(n-1) + n$; guess $f(n) = \Theta(n^2)$

Write $f(n) = an^2 + g(n)$, where $g(n) = o(n^2)$

Substituting in the recurrence equation, we get:

$$an^2 + g(n) = a(n-1)^2 + g(n-1) + n$$

This equation simplifies to:

$$g(n) = g(n-1) + (1-2a)n + a$$

Choose $a = 1/2$ to make $g(n) = o(n^2)$ possible

$$g(n) = g(n-1) + 1/2 = n/2 - 1 \quad \{g(1) = -1/2, g(2) = 0\}$$

The solution to the original recurrence then becomes

$$f(n) = n^2/2 + g(n) = n^2/2 + n/2 - 1$$

Solution via a basic theorem

Theorem 3.1 (basic theorem for recurrences): Given $f(n) = a\, f(n/b) + h(n)$; a, b constant, h an arbitrary function the asymptotic solution to the recurrence is

$f(n) = \Theta(n^{\log_b a})$ if $h(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$

$f(n) = \Theta(n^{\log_b a} \log n)$ if $h(n) = \Theta(n^{\log_b a})$

$f(n) = \Theta(h(n))$ if $h(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$

4 Models of Parallel Processing

Back to TOC

Chapter Goals

- Elaborate on the taxonomy of parallel processing from Chapter 1
- Introduce abstract models of shared and distributed memory
- Understand the differences between abstract models and real hardware

Chapter Contents

- 4.1. Development of Early Models
- 4.2. SIMD versus MIMD Architectures
- 4.3. Global versus Distributed Memory
- 4.4. The PRAM Shared-Memory Model
- 4.5. Distributed-Memory or Graph Models
- 4.6. Circuit Model & Physical Realizations

4.1 Development of Early Models

Thousands of processors were found in some computers as early as the 1960s

These architectures were variously referred to as

associative memories
associative processors
logic-in-memory machines

More recent names are

processor-in-memory and
intelligent RAM

Table 4.1. Entering the Second Half-Century of Associative Processing

Decade	Events and Advances	Technology	Performance
1940s	Formulation of need & concept	Relays	
1950s	Emergence of cell technologies	Magnetic, Cryogenic	Mega-bit-OPS
1960s	Introduction of basic architectures	Transistors	
1970s	Commercialization & applications	ICs	Giga-bit-OPS
1980s	Focus on system/software issues	VLSI	Tera-bit-OPS
1990s	Scalable & flexible architectures	ULSI, WSI	Peta-bit-OPS?

Revisiting the Flynn-Johnson classification

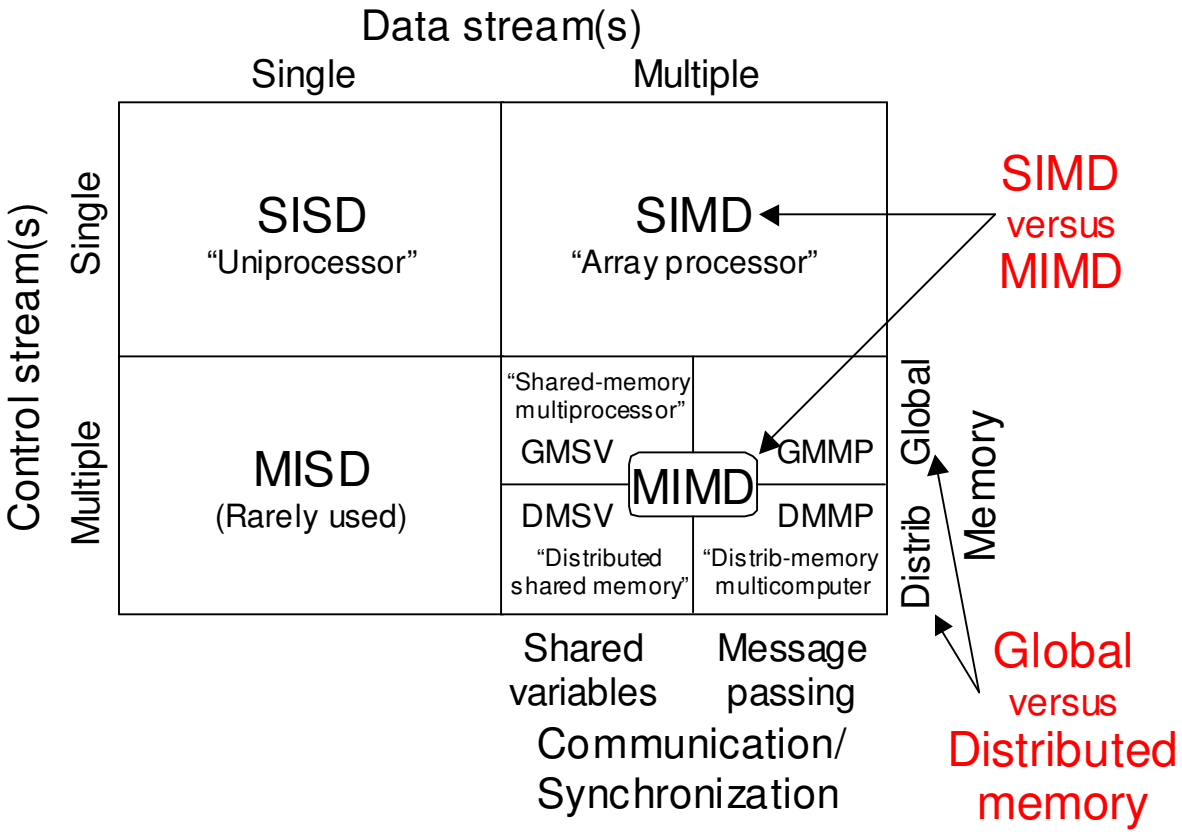


Fig. 4.1. The Flynn-Johnson classification of computer systems.

MISD can be viewed as a flexible (programmable) pipeline

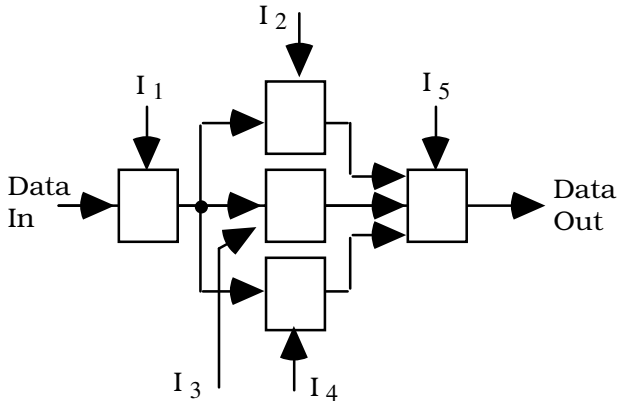


Fig. 4.2. Multiple instruction streams operating on a single data stream (MISD).

4.2 SIMD versus MIMD Architectures

Most early parallel machines were of SIMD type

Synchronous SIMD

To run data-dependent conditionals (if-then-else), first processors satisfying the condition are enabled, next the remainder are enabled for the "else" part

Critics of SIMD view the above as being wasteful

But: are buses less efficient than private cars, or is PC hardware wasted when you answer the phone?

Asynchronous SIMD = SPMD

Custom- versus commodity-chip SIMD

Most recent parallel machines are MIMD-type

MPP: massively or moderately parallel processor?

Tight versus loose coupling of processors

Tightly coupled: multiprocessors
Loosely coupled: multicomputers
Network or cluster of workstations (NOW, COW)
Hybrid: loosely coupled clusters, each tightly coupled

Message passing versus virtual shared memory

Shared memory is easier to program
Message passing is more efficient

4.3 Global versus Distributed Memory

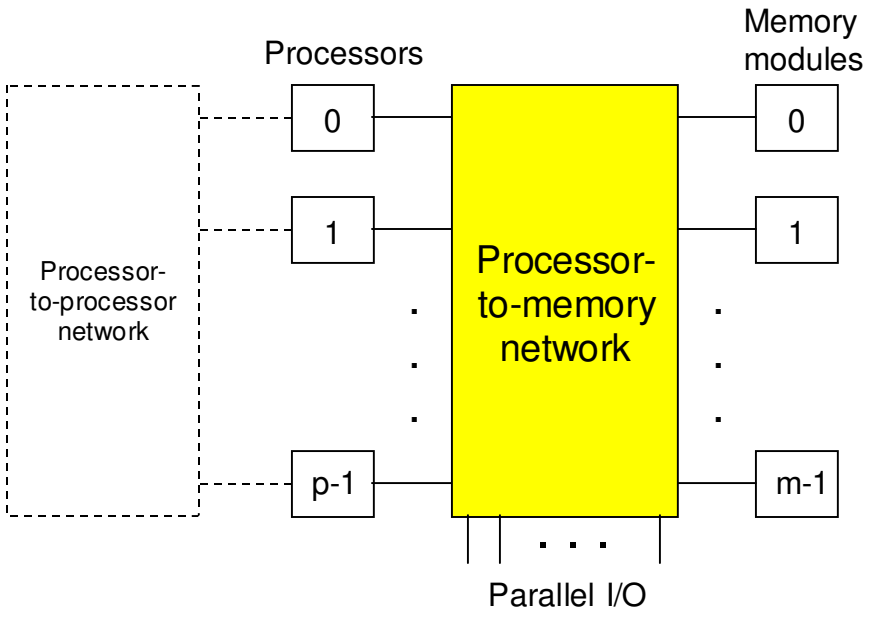


Fig. 4.3. A parallel processor with global memory.

Example processor-to-memory/processor networks:

1. Crossbar; $p \times m$ array of switches or crosspoints; cost too high for massively parallel systems
2. Single/multiple bus (complete or partial connectivity)
3. Multistage interconnection network (MIN); cheaper than crossbar, more bandwidth than bus

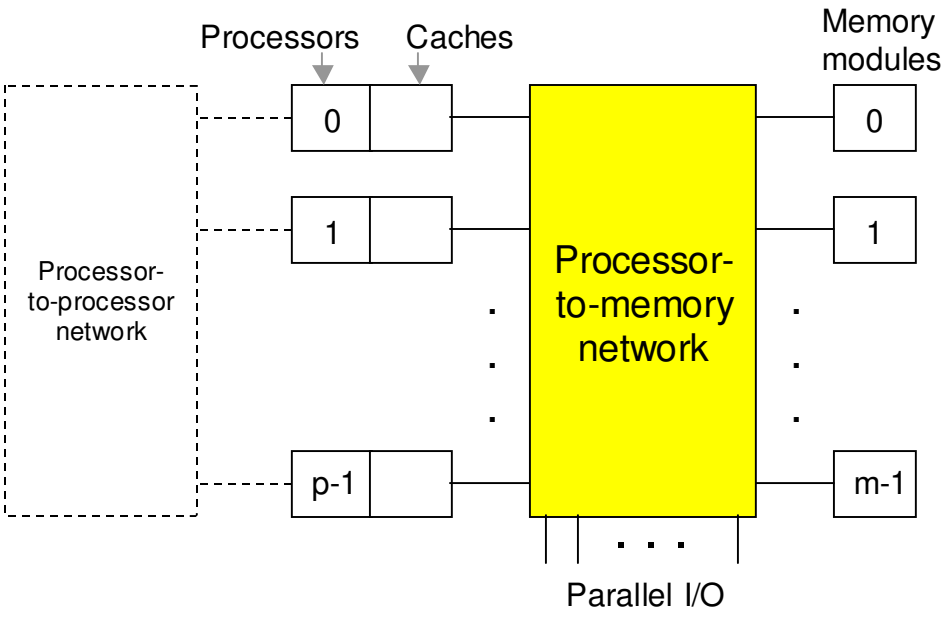


Fig. 4.4. A parallel processor with global memory and processor caches.

Solving the cache coherence problem

1. Do not cache any shared data
2. Do not cache "writeable" shared data
   or allow only one cache copy
3. Use a cache coherence protocol (Chapter 18)

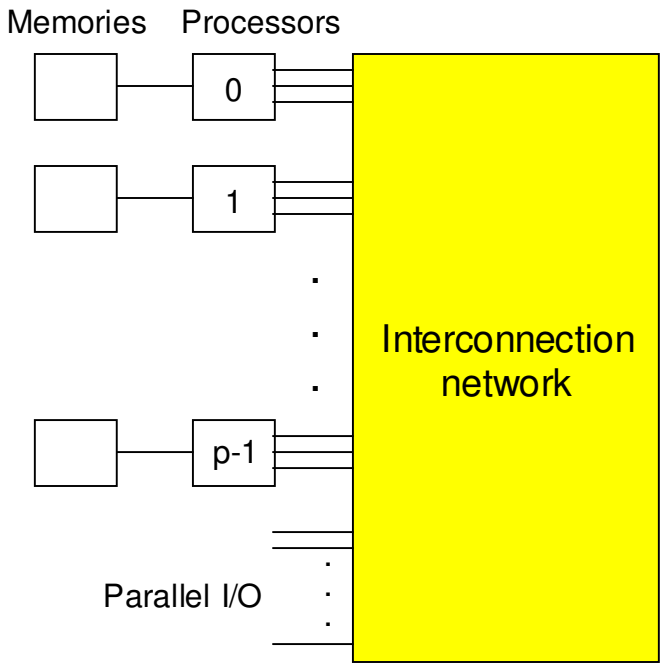


Fig. 4.5. A parallel processor with distributed memory.

Examples networks for distributed memory machines

1. Crossbar; cost too high for massively parallel system
2. Single/multiple bus (complete or partial connectivity)
3. Multistage interconnection network (MIN)
4. Various direct networks (Section 4.5)

Terminology

UMA Uniform memory access
NUMA Nonuniform memory access
COMA Cache-only memory architecture (aka all-cache)

4.4 The PRAM Shared-Memory Model

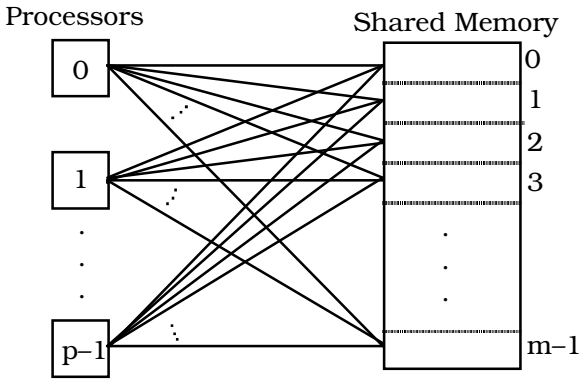


Fig. 4.6. Conceptual view of a parallel random-access machine (PRAM).

PRAM cycle
1. Processors access memory (usually different locations)
2. Processors perform a computation step
3. Processors store their results in memory

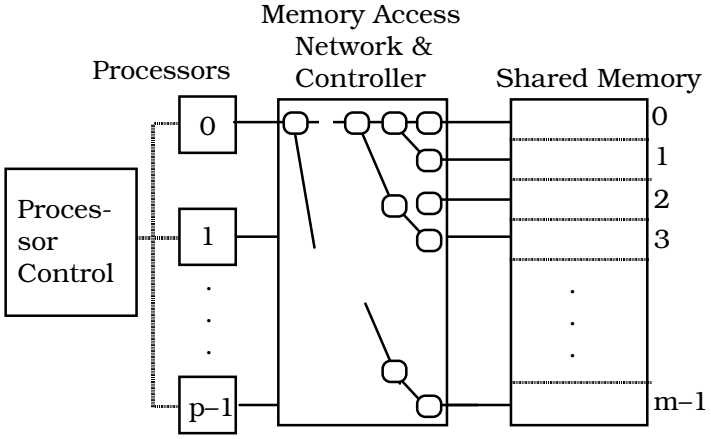


Fig. 4.7. PRAM with some hardware details shown.

In practice, memory is divided into modules and simultaneous accesses to same module are disallowed

4.5 Distributed-Memory or Graph Models

Parameters of interest for direct interconnection networks
- Diameter
- Bisection (band)width
- Node degree

Symmetry properties simplify algorithm development:
- Node or vertex symmetry
- Link or edge symmetry

Table 4.2. Topological Parameters of Selected Interconnection Networks

Network name(s)	Number of nodes	Network diameter	Bisection width	Node degree	Local links?
1D mesh (linear array)	k	$k-1$	1	2	Yes
1D torus (ring, loop)	k	$k/2$	2	2	Yes
2D Mesh	k^2	$2k-2$	k	4	Yes
2D torus (k-ary 2-cube)	k^2	k	$2k$	4	Yes[1]
3D mesh	k^3	$3k-3$	k^2	6	Yes
3D torus (k-ary 3-cube)	k^3	$3k/2$	$2k^2$	6	Yes[1]
Pyramid	$(4k^2-1)/3$	$2\log_2 k$	$2k$	9	No
Binary tree	2^l-1	$2l-2$	1	3	No
4-ary hypertree	$2^l(2^{l+1}-1)$	$2l$	2^{l+1}	6	No
Butterfly	$2^l(l+1)$	$2l$	2^l	4	No
Hypercube	2^l	l	2^{l-1}	l	No
Cube-connected cycles	$2^l l$	2^l	2^{l-1}	3	No
Shuffle-exchange	2^l	$2l-1$	$\geq 2^{l-1}/l$	4 unidir.	No
De Bruijn	2^l	l	$2^l/l$	4 unidir.	No

[1] With folded layout.

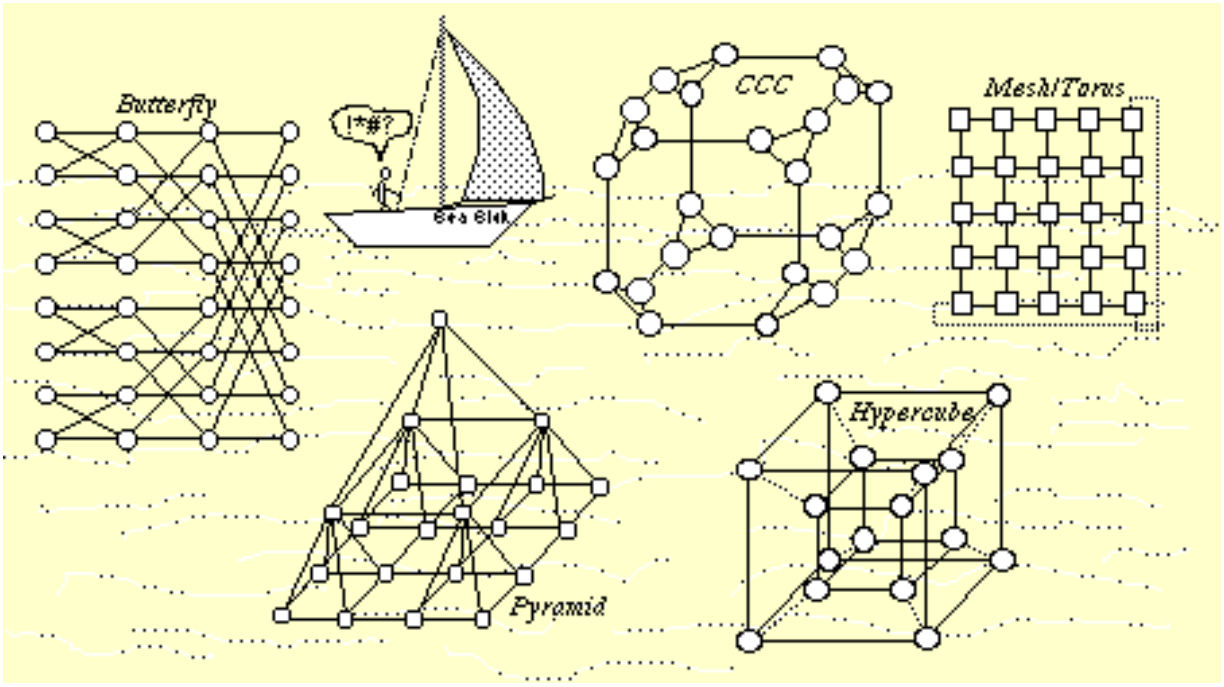


Fig. 4.8. The sea of interconnection networks.

Bus-based architectures are dominant in small-scale parallel systems.

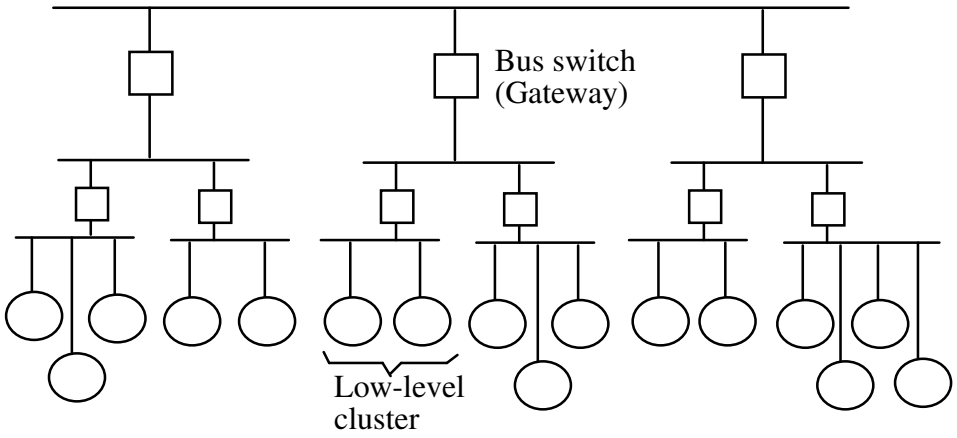


Fig. 4.9. Example of a hierarchical interconnection architecture.

Because each interconnection network requires its own algorithms, various abstract (architecture-independent) models have been suggested for such networks

The LogP model

Characterizes an architecture with just four parameters:

L *Latency* upper bound when a small message is sent from an arbitrary source to an arbitrary destination

o *overhead*, defined as the length of time a processor is dedicated to transmission or reception of a message, thus being unable to do any other computation

g *gap*, defined as the minimum time that must elapse between consecutive message transmissions or receptions by a single processor (1/*g* is the available per-processor communication bandwidth)

P *Processor* multiplicity (*p* in our notation)

If LogP is in fact an accurate model for capturing the effects of communication in parallel processors, then details of interconnection network do not matter

The BSP model (bulk-synchronous parallel)

Hides the communication latency altogether through a specific parallel programming style, thus making the network topology irrelevant

Synchronization of processors occurs once every *L* time steps, where *L* is a periodicity parameter

Computation consists of a sequence of supersteps

In a given superstep, each processor performs a task consisting of local computation steps, message transmissions, and message receptions

Data received in messages will not be used in the current superstep but rather beginning with the next superstep

After each period of *L* time units, a global check is made to see if the current superstep has been completed

> If so, then the processors move on to executing the next superstep
>
> Else, the next period of length *L* is allocated to the unfinished super-step

4.6 Circuit Model and Physical Realizations

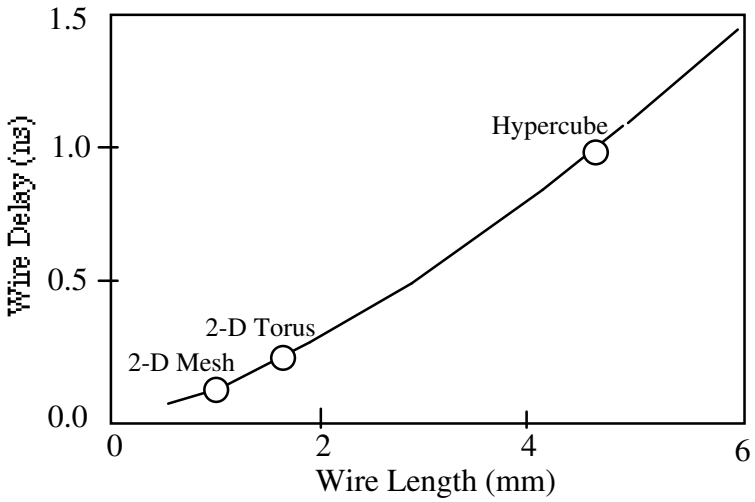


Fig. 4.10. Intrachip wire delay as a function of wire length.

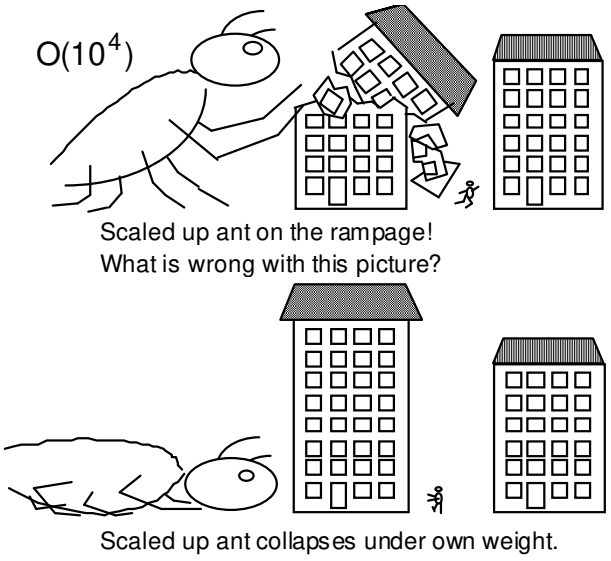


Fig. 4.11. Pitfalls of scaling up.

Part II Extreme Models

Back to TOC

Part Goals

- Study two extreme parallel machine models
  - Abstract PRAM shared-memory model ignores implementation issues altogether
  - Concrete circuit model accommodates details like circuit depth and layout area
- Prepare for everything else that falls in between the two extremes

Part Contents

- Chapter 5: PRAM and Basic Algorithms
- Chapter 6: More Shared-Memory Algorithms
- Chapter 7: Sorting and Selection Networks
- Chapter 8: Other Circuit-Level Examples

5 PRAM and Basic Algorithms

Back to TOC

Chapter Goals

- Define PRAM and its various submodels
- Show PRAM to be a natural extension of the sequential computer (RAM)
- Develop five important parallel algorithms that can serve as building blocks
  (more algorithms in the next chapter)

Chapter Contents

- 5.1. PRAM Submodels and Assumptions
- 5.2. Data Broadcasting
- 5.3. Semigroup or Fan-in Computation
- 5.4. Parallel Prefix Computation
- 5.5. Ranking the Elements of a Linked List
- 5.6. Matrix Multiplication

5.1 PRAM Submodels and Assumptions

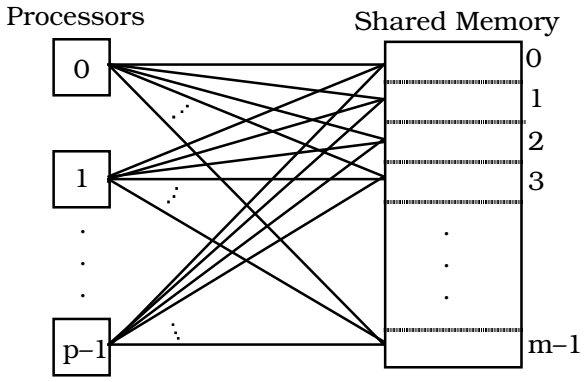


Fig. 4.6. Conceptual view of a parallel random-access machine (PRAM).

Processor *i* can do the following in 3 phases of one cycle:
1. Fetch an operand from address s_i in shared memory
2. Perform computations on data held in local registers
3. Store a value into address d_i in shared memory

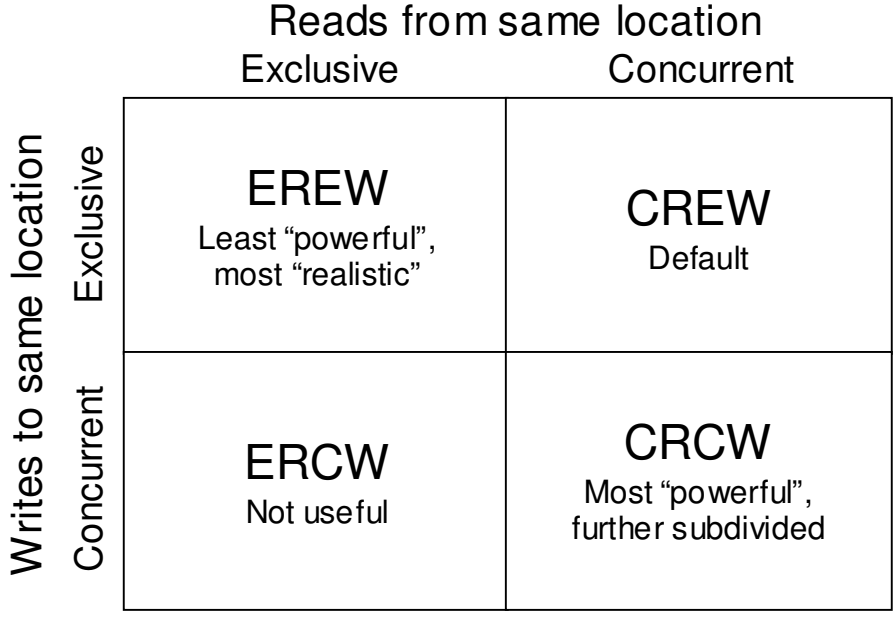


Fig. 5.1 Submodels of the PRAM model.

CRCW PRAM is classified according to how concurrent writes are handled. These submodels are all different from each other and from EREW and CREW.

Undefined: In case of multiple writes, the value written is undefined (CRCW-U)

Detecting: A code representing "detected collision" is written (CRCW-D)

Common: Multiple writes allowed only if all store the same value (CRCW-C); this is sometimes called the consistent-write submodel

Random: The value written is randomly chosen from those offered (CRCW-R)

Priority: The processor with the lowest index succeeds in writing (CRCW-P)

Max/Min: The largest/smallest of the multiple values is written (CRCW-M)

Reduction: The arithmetic sum (CRCW-S), logical AND (CRCW-A), logical XOR (CRCW-X), or another combination of the multiple values is written.

Ordering the submodels by computational power:

$$\text{EREW} < \text{CREW} < \text{CRCW-D} < \text{CRCW-C} < \text{CRCW-R} < \text{CRCW-P}$$

Theorem 5.1: A p-processor CRCW-P (priority) PRAM can be simulated (emulated) by a p-processor EREW PRAM with a slowdown factor of $\Theta(\log p)$.

Intuitive justification for concurrent read emulation:

Write the p desired addresses in a list
Sort the list of addresses in ascending order
Remove all duplicate addresses
Access data from desired addresses
Replicate data via parallel prefix computation

Each of these steps requires constant or $O(\log p)$ time

Some elementary PRAM computations

Initializing an n-vector (base address = B) to all 0s:

for $j = 0$ to $\lceil n/p \rceil - 1$ processor i do
    if $jp + i < n$ then $M[B + jp + i] := 0$
endfor

Adding two n-vectors and storing the results in a third
(base addresses B', B'', B)

Convolution of two n-vectors: $W_k = \sum_{i+j=k} U_i \times V_j$
(base addresses B_W, B_U, B_V)

5.2 Data Broadcasting

Broadcasting is built-in for the CREW and CRCW models

EREW broadcasting: make p copies of the data in a broadcast vector B

Making p copies of $B[0]$ by recursive doubling

for $k = 0$ to $\lceil \log_2 p \rceil - 1$ Processor j, $0 \leq j < p$, do

    Copy $B[j]$ into $B[j + 2^k]$

endfor

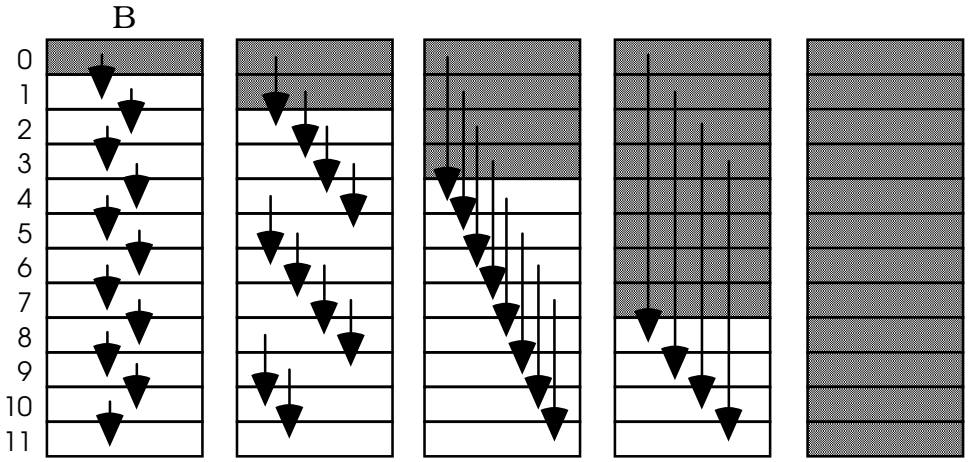


Fig. 5.2. Data broadcasting in EREW PRAM via recursive doubling.

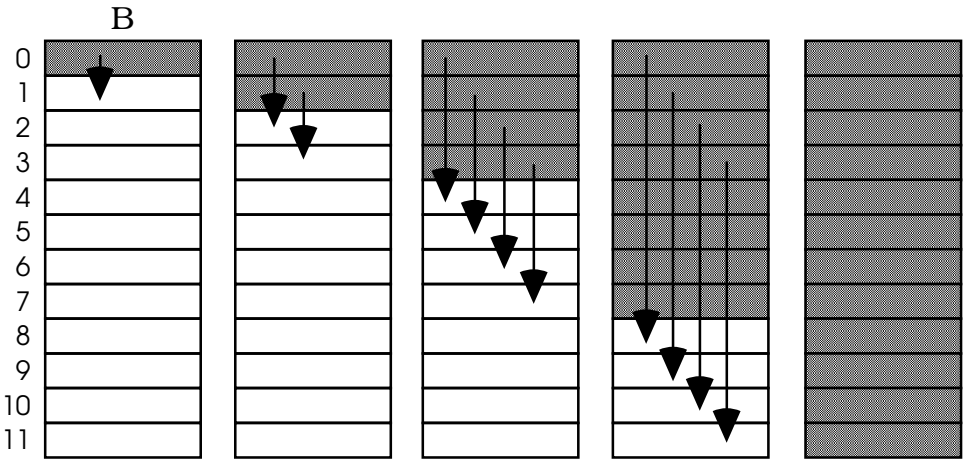


Fig. 5.3. EREW PRAM data broadcasting without redundant copying.

EREW PRAM algorithm for broadcasting by Processor *i*

Processor *i* write the data value into *B*[0]
s := 1
while *s* < *p* Processor *j*, 0 ≤ *j* < min(*s*, *p* – *s*), do
    Copy *B*[*j*] into *B*[*j* + *s*]
    s := 2*s*
endwhile
Processor *j*, 0 ≤ *j* < *p*, read the data value in *B*[*j*]

EREW PRAM algorithm for all-to-all broadcasting

Processor *j*, 0 ≤ *j* < *p*, write own data value into *B*[*j*]
for *k* = 1 to *p* – 1 Processor *j*, 0 ≤ *j* < *p*, do
    Read the data value in *B*[(*j* + *k*) mod *p*]
endfor

Both of the preceding algorithms are time-optimal (shared memory is the only communication mechanism and each processor can read but one value per cycle)

In the following naive sorting algorithm, processor j determines the rank $R[j]$ of its data element $S[j]$ by examining all the other data elements; it then writes $S[j]$ in element $R[j]$ of the output (sorted) vector

Naive EREW PRAM sorting algorithm
(using all-to-all broadcasting)

```
Processor j, 0 ≤ j < p, write 0 into R[j]
for k = 1 to p – 1 Processor j, 0 ≤ j < p, do
    l := (j + k) mod p
    if S[l] < S[j] or S[l] = S[j] and l < j
    then R[j] := R[j] + 1
    endif
endfor
Processor j, 0 ≤ j < p, write S[j] into S[R[j]]
```

This O(p)-time algorithm is far from being optimal

5.3 Semigroup or Fan-in Computation

This computation is trivial for a CRCW PRAM of the reduction variety if the reduction operator happens to be ⊗

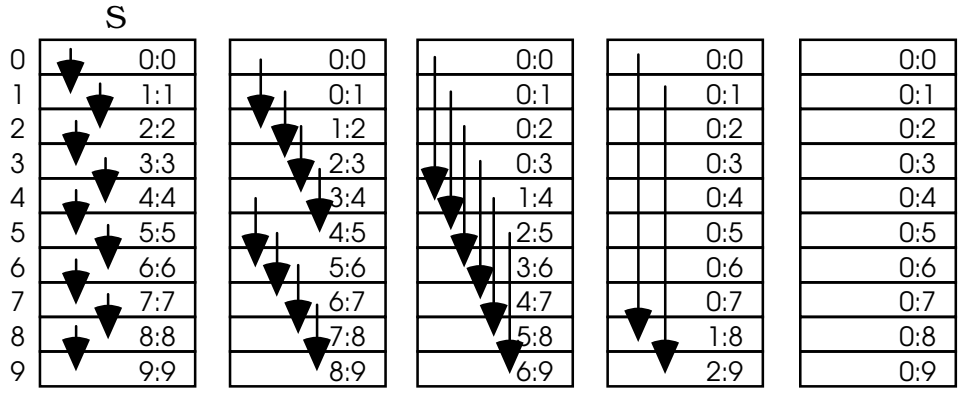


Fig. 5.4. Semigroup computation in EREW PRAM.

EREW PRAM semigroup computation algorithm
Processor j, $0 \le j < p$, copy $X[j]$ into $S[j]$
$s := 1$
while $s < p$ Processor j, $0 \le j < p - s$, do
    $S[j + s] := S[j] \otimes S[j + s]$
    $s := 2s$
endwhile
Broadcast $S[p - 1]$ to all processors

Time-optimal algorithm (CRCW can do better: prob. 5.16)

Speed-up = $p/\log_2 p$

Efficiency = Speed-up/p = $1/\log_2 p$

$$\text{Utilization} = \frac{W(p)}{pT(p)} \cong \frac{(p-1)+(p-2)+(p-4)+ \ldots +(p-p/2)}{p \log_2 p} \cong 1 - 1/\log_2 p$$

Semigroup computation with each processor holding n/p data elements:

Each processor combine its sublist n/p steps

Do semigroup computation on results $\log_2 p$ steps

$$\text{Speedup}(n, p) = \frac{n}{n/p + 2\log_2 p} = \frac{p}{1 + (2p\log_2 p)/n}$$

$$\text{Efficiency}(n, p) = \text{Speedup}/p = \frac{1}{1 + (2p\log_2 p)/n}$$

For $p = \Theta(n)$, the speedup of $\Theta(n/\log n)$ is sublinear

The efficiency in this case is $\Theta(n/\log n)/\Theta(n) = \Theta(1/\log n)$

Limit the number of processors to $p = O(n/\log n)$:

Speedup$(n, p) = n/O(\log n) = \Omega(n/\log n) = \Omega(p)$

Efficiency$(n, p) = \Theta(1)$

Using fewer processors than tasks = parallel slack

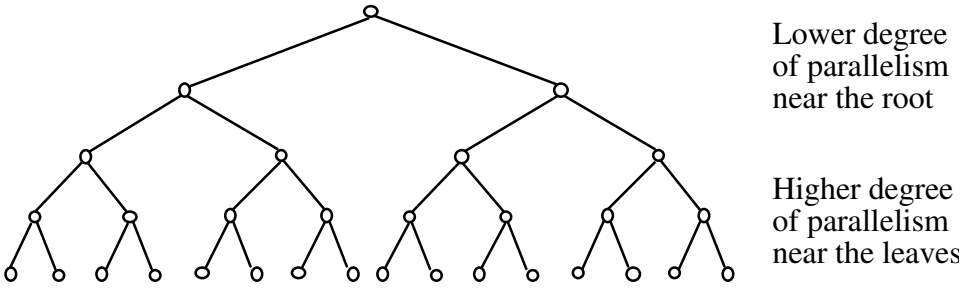


Fig. 5.5. Intuitive justification of why parallel slack helps improve the efficiency.

Inner product of two n-vectors, storing the result in s

Base addresses B' and B'',

auxiliary vector of length p with base address B

```
for j = 0 to ⌈n/p⌉ – 1 processor i do
    if jp + i < n then
        load M[B′ + jp + i]
        multiply by M[B″ + jp + i]
        add to M[B + i]
    endif
    find sum or the p-vector, store the result in s
endfor
```

$T(n, p) = O(n/p + \log p)$

Matrix-by-vector multiplication $U := M \times V$

U_i is the inner product of row i of M and V

$T(n, p) = O(n^2/p + n \log p))$

5.4 Parallel Prefix Computation

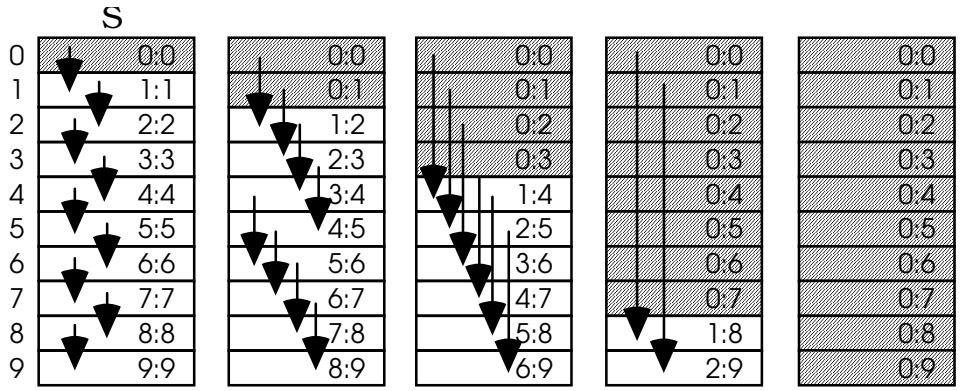


Fig. 5.6. Parallel prefix computation in EREW PRAM via recursive doubling.

Two other solutions, based on divide and conquer

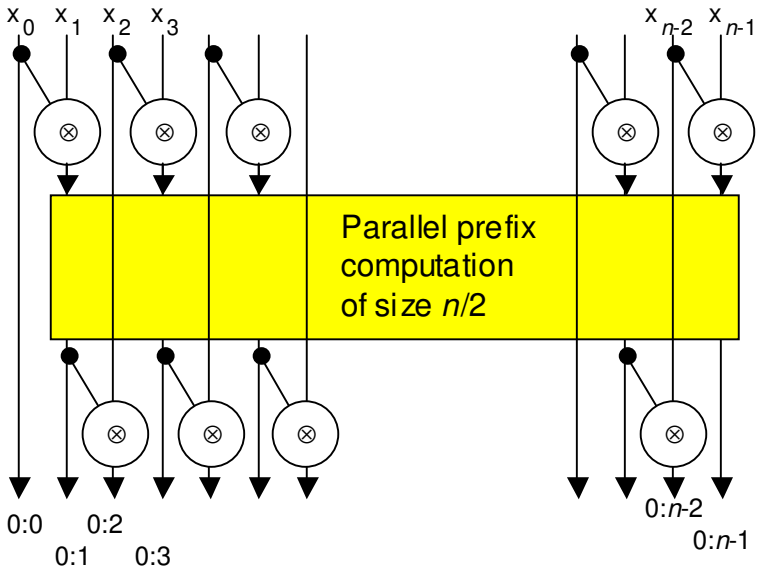


Fig. 5.7 Parallel prefix computation using a divide-and-conquer scheme.

Assume $n = p$

$T(p) = T(p/2) + 2 = 2 \log_2 p$

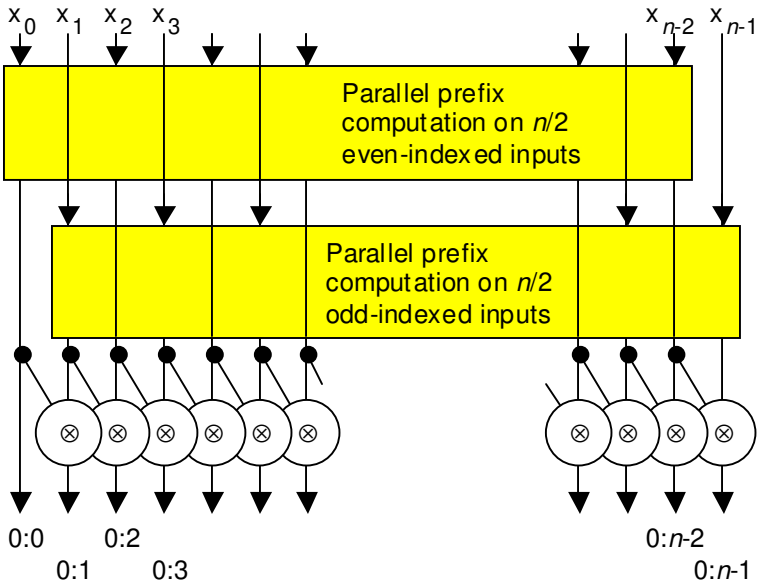


Fig. 5.8. Another divide-and-conquer scheme for parallel prefix computation.

Assume $n = p$

$T(p) = T(p/2) + 1 = \log_2 p$ Requires commutativity

5.5 Ranking the Elements of a Linked List

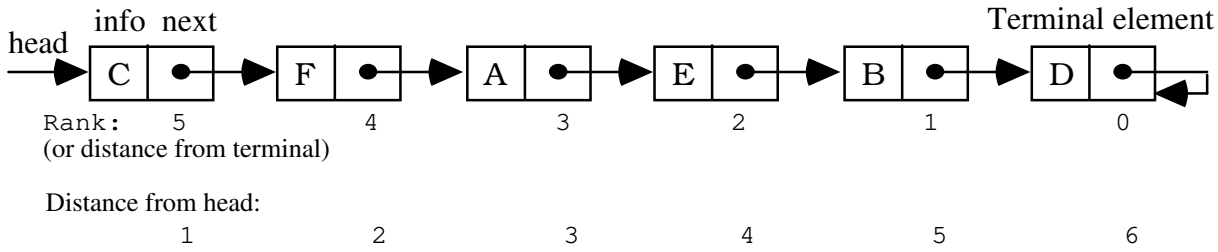


Fig. 5.9. Example linked list and the ranks of its elements.

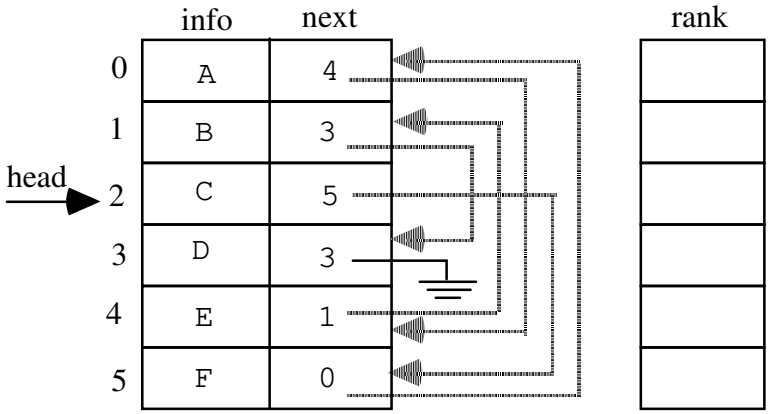


Fig. 5.10. PRAM data structures representing a linked list and the ranking results.

List-ranking appears to be hopelessly sequential

However, we can in fact use a recursive doubling scheme to determine the rank of each element in optimal time

There exist other problems that seem unparallizable

This is why intuition can be misleading when it comes to determining which computation is or is not efficiently parallelizable (i.e., it is or is not in NC)

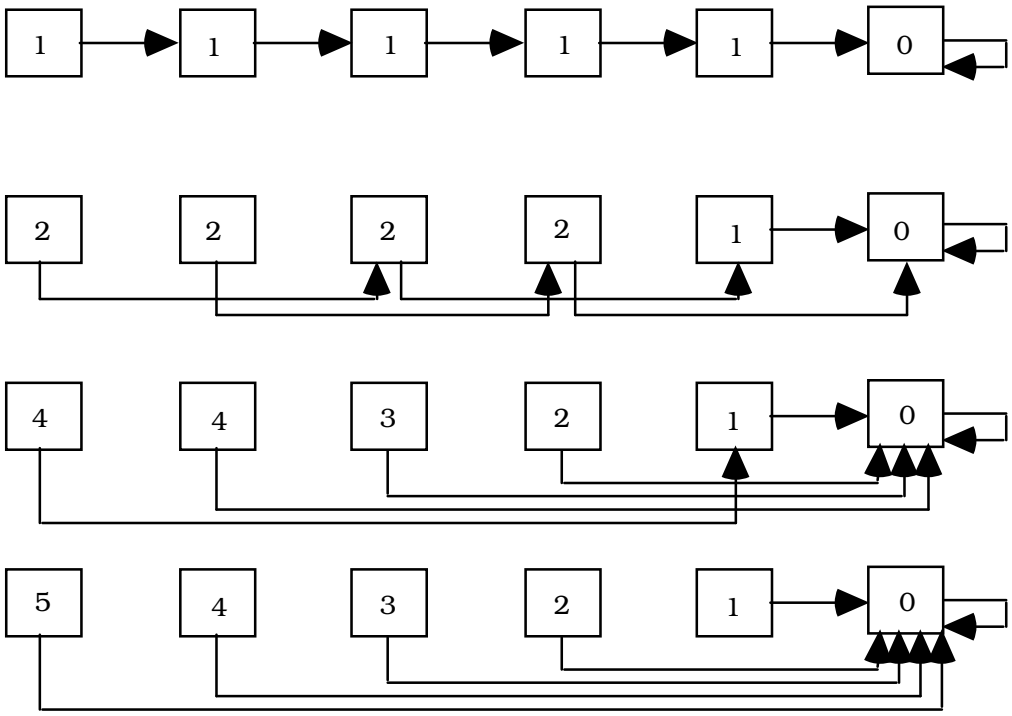


Fig. 5.11. Element ranks initially and after each of the three iterations.

PRAM list ranking algorithm (via pointer jumping)

Processor j, $0 \le j < p$, do {initialize the partial ranks}
    if $next[j] = j$
    then $rank[j] := 0$
    else $rank[j] := 1$
    endif
while $rank[next[head]] \neq 0$ Processor j, $0 \le j < p$, do
    $rank[j] := rank[j] + rank[next[j]]$
    $next[j] := next[next[j]]$
endwhile

Which PRAM submodel is implicit in this algorithm?

5.6 Matrix Multiplication

For $m \times m$ matrices, $C = A \times B$ means: $c_{ij} = \sum_{k=0}^{m-1} a_{ik}\, b_{kj}$

Sequential matrix multiplication algorithm

```
for i = 0 to m – 1 do
    for j = 0 to m – 1 do
        t := 0
        for k = 0 to m – 1 do
            t := t + a_ik b_kj
        endfor
        c_ij := t
    endfor
endfor
```

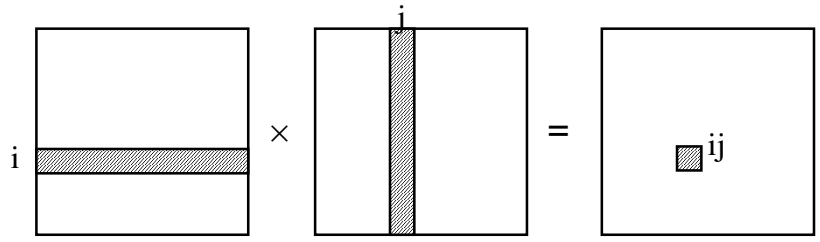
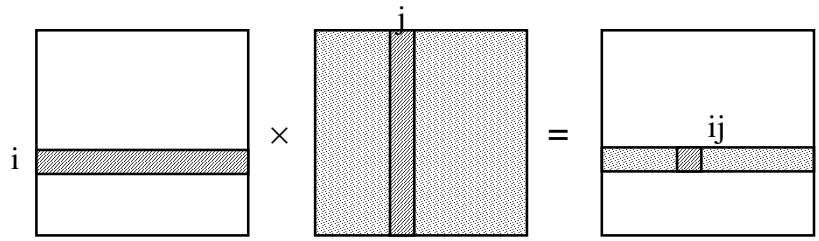


Fig. 5.12. PRAM matrix multiplication; $p = m^2$ processors.

PRAM matrix multiplication algorithm using m^2 processors

```
Processor (i, j), 0 ≤ i, j < m, do
begin
    t := 0
    for k = 0 to m – 1 do
        t := t + a_ik b_kj
    endfor
    c_ij := t
end
```

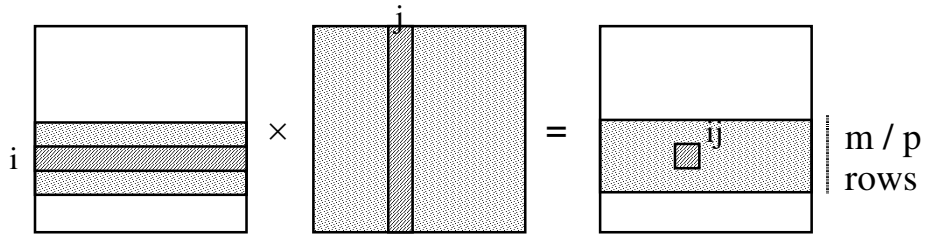


PRAM matrix multiplication algorithm using m processors

```
for j = 0 to m – 1 Processor i, 0 ≤ i < m, do
    t := 0
    for k = 0 to m – 1 do
        t := t + a_ik b_kj
    endfor
    c_ij := t
endfor
```

Both of the preceding algorithms are efficient and provide linear speedup

Using fewer than m processors: each processor computes m/p rows of C



This solution inefficient for NUMA parallel architectures

Each element of B is fetched m/p times

For each such access, only two arith ops are performed

Block matrix multiplication

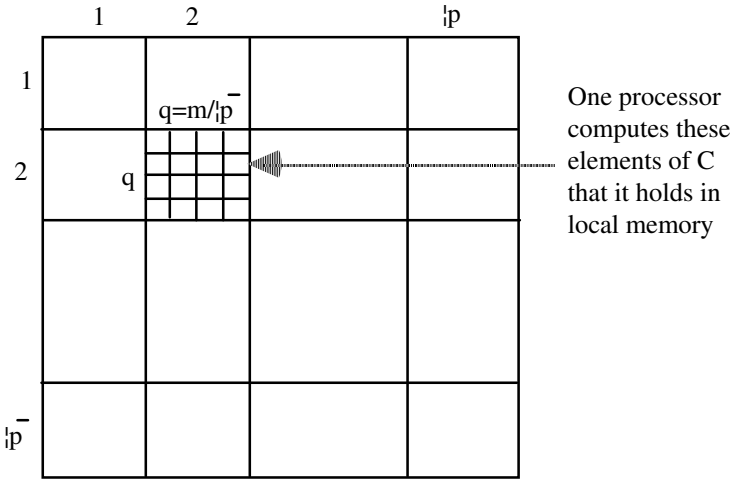
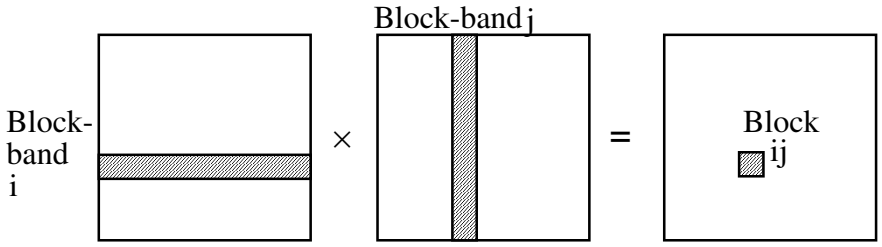


Fig. 5.13. Partitioning the matrices for block matrix multiplication.



Each multiply-add computation on $q \times q$ blocks needs

$2q^2 = 2m^2/p$ memory accesses to read the blocks

$2q^3$ arithmetic operations

So, q arithmetic operations are done per memory access

We assume that processor (i, j) has local memory to hold

Block (i, j) of the result matrix C (q^2 elements)

One block-row of B; say row $kq + c$ of block (k, j) of B

(Elements of A can be brought in one at a time)

For example, as element in row $iq + a$ of column $kq + c$ in block (i, k) of A is brought in, it is multiplied in turn by the locally stored q elements of B, and the results added to the appropriate q elements of C

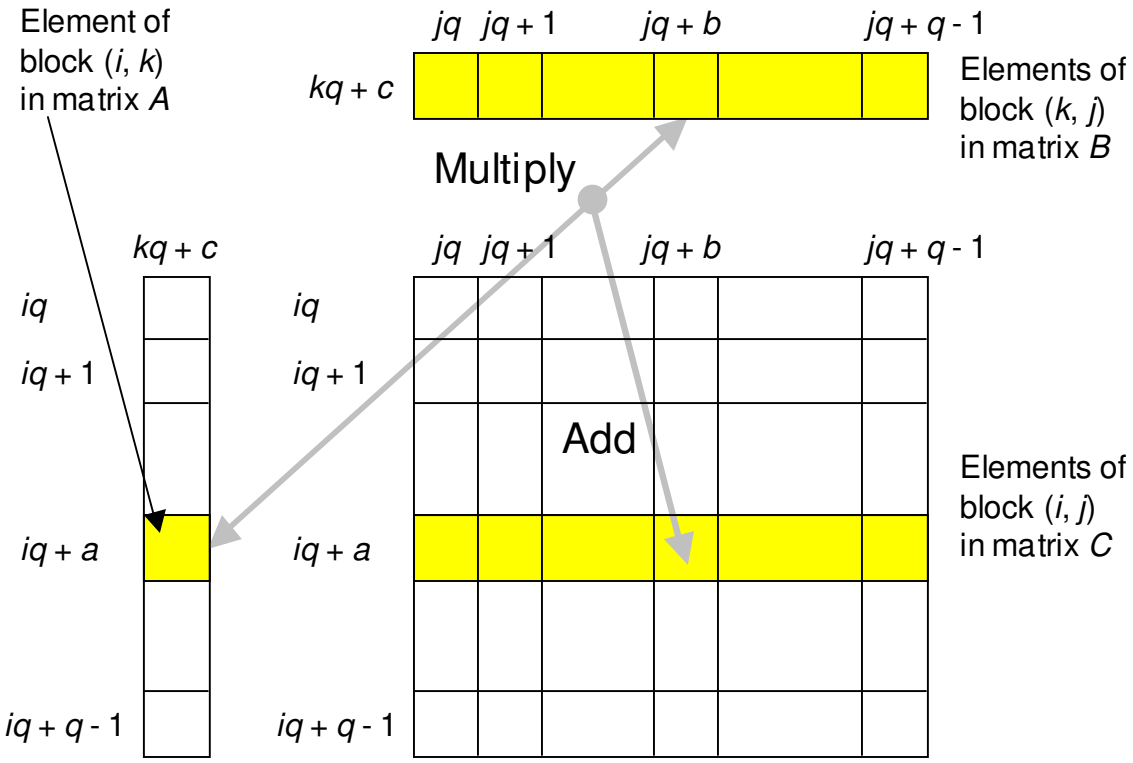


Fig. 5.14. How Processor (*i*, *j*) operates on an element of *A* and one block-row of *B* to update one block-row of *C*.

On the Cm* NUMA-type shared-memory multiprocessor, this block algorithm exhibited good, but sublinear, speedup

$p = 16$, speed-up = 5 in multiplying 24×24 matrices;

improved to 9 (11) for 36×36 (48×48) matrices

The improved locality of block matrix multiplication can also improve the running time on a uniprocessor, or distributed shared-memory multiprocessor with caches

Reason: higher cache hit rates.

6 More Shared-Memory Algorithms

Back to TOC

Chapter Goals

- Develop PRAM algorithms for more complex problems
  (background on corresponding sequential algorithms also presented)
- Discuss some practical implementation issues such as data distribution

Chapter Contents

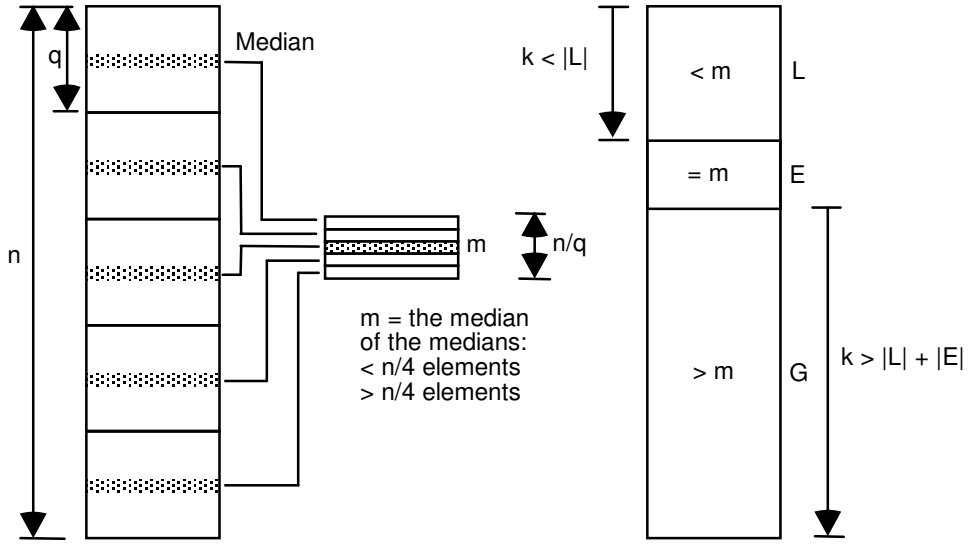
- 6.1. Sequential Rank-Based Selection
- 6.2. A Parallel Selection Algorithm
- 6.3. A Selection-Based Sorting Algorithm
- 6.4. Alternative Sorting Algorithms
- 6.5. Convex Hull of a 2D Point Set
- 6.6. Some Implementation Aspects

6.1 Sequential Rank-Based Selection

Selection: Find the (a) *k*th smallest among *n* elements

Naive solution through sorting, O(*n* log *n*) time

Linear-time sequential algorithm can be developed



Sequential rank-based selection algorithm *select*(S, k)

1. if $|S| < q$ {q is a small constant}
   then sort S and return the kth smallest element of S
   else divide S into $|S|/q$ subsequences of size q
      Sort each subsequence and find its median
      Let the $|S|/q$ medians form the sequence T
   endif
2. $m = select(T, |T|/2)$
   {find the median m of the $|S|/q$ medians}
3. Create 3 subsequences
   L: Elements of S that are $< m$
   E: Elements of S that are $= m$
   G: Elements of S that are $> m$
4. if $|L| \geq k$
   then return *select*(L, k)
   else if $|L| + |E| \geq k$
      then return m
      else return *select*(G, $k - |L| - |E|$)
   endif

Analysis:

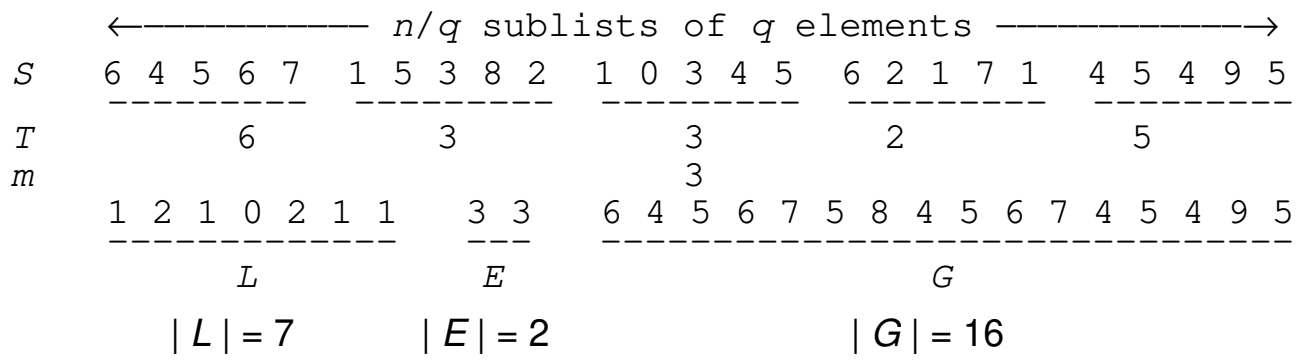
$$T(n) = T(n/q) + T(3n/4) + cn$$

Let $q = 5$; we guess the solution to be $T(n) = dn$

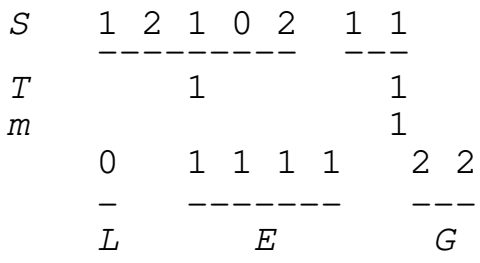
$$dn = dn/5 + 3dn/4 + cn \quad \Rightarrow \quad d = 20c$$

Examples for sequential selection

from an input list of size $n = 25$ using $q = 5$



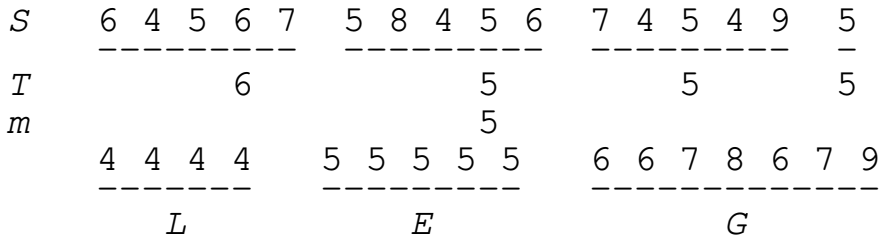
To find the 5th smallest element in S, select the 5th smallest element in L



Answer: 1

The 9th smallest element of S is 3

The 13th smallest element of S is found by selecting the 4th smallest element in G



Answer: 4

6.2 A Parallel Selection Algorithm

Parallel rank-based selection algorithm *PRAMselect*(S, k, p)

1. if $|S| < 4$
   then sort S and return the kth smallest element of S
   else broadcast $|S|$ to all p processors
   divide S into p subsequences $S^{(j)}$ of size $|S|/p$
   Processor j, $0 \le j < p$, compute $T_j := select(S^{(j)}, |S^{(j)}|/2)$
   endif
2. $m = PRAMselect(T, |T|/2, p)$ {median of the medians}
3. Broadcast m to all processors and create 3 subsequences
   L: Elements of S that are $< m$
   E: Elements of S that are $= m$
   G: Elements of S that are $> m$
4. if $|L| \ge k$
   then return $PRAMselect(L, k, p)$
   else if $|L| + |E| \ge k$
   then return m
   else return $PRAMselect(G, k - |L| - |E|, p)$
   endif

Analysis: Let $p = n^{1-x}$, with $x > 0$ a known constant

e.g., $x = 1/2 \Rightarrow p = \sqrt{n}$

$T(n, p) = T(n^{1-x}, p) + T(3n/4, p) + cn^x = O(n^x)$

Speed-up$(n, p) = \Theta(n)/O(n^x) = \Omega(n^{1-x}) = \Omega(p)$

Efficiency $= \Omega(1)$

What if $x = 0$, i.e., we use $p = n$ processors for an n-input selection problem?

6.3 A Selection-Based Sorting Algorithm

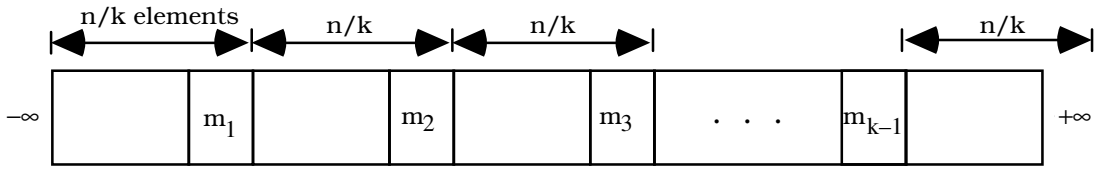


Fig. 6.1. Partitioning of the sorted list for selection-based sorting.

Parallel selection-based sort *PRAMselectionsort*(S, p)

1. if $|S| < k$ then return *quicksort*(S)
2. for $i = 1$ to $k - 1$ do
   m_j := *PRAMselect*(S, $i|S|/k$, p)
   {for notational convenience, let $m_0 := -\infty$; $m_k := +\infty$}
   endfor
3. for $i = 0$ to $k - 1$ do
   make the sublist $T^{(i)}$ from elements of S in (m_i, m_{i+1})
   endfor
4. for $i = 1$ to $k/2$ do in parallel
   PRAMselectionsort($T^{(i)}$, $2p/k$)
   {$p/(k/2)$ processors used for each
   of the $k/2$ subproblems}
   endfor
5. for $i = k/2 + 1$ to k do in parallel
   PRAMselectionsort($T^{(i)}$, $2p/k$)
   endfor

Analysis: $p = n^{1-x}$, with $x > 0$ a known constant, $k = 2^{1/x}$

$$T(n, p) = 2T(n/k, 2p/k) + cn^x = O(n^x \log n)$$

Why can't all k subproblems be solved in step 4 at once?

$$\text{Speedup}(n, p) = \Omega(n \log n)/O(n^x \log n) = \Omega(n^{1-x}) = \Omega(p)$$

$$\text{Efficiency} = \text{Speedup} / p = \Omega(1)$$

$$\text{Work}(n, p) = pT(n, p) = \Theta(n^{1-x})\, O(n^x \log n) = O(n \log n)$$

Our asymptotic analysis is valid for $x > 0$ but not for $x = 0$;
i.e., *PRAMselectionsort* does not allow us to sort p keys
in optimal O(log p) time

Example:

S: 6 4 5 6 7 1 5 3 8 2 1 0 3 4 5 6 2 1 7 0 4 5 4 9 5

Threshold values:

$n/k = 25/4 \cong 6$
$2n/k = 50/4 \cong 13$
$3n/k = 75/4 \cong 19$

$m_0 = -\infty$
$m_1 = PRAMselect(S, 6, 5) = 2$
$m_2 = PRAMselect(S, 13, 5) = 4$
$m_3 = PRAMselect(S, 19, 5) = 6$
$m_4 = +\infty$

T: - - - - - - 2 | - - - - - - - 4 | - - - - - 6 | - - - - - -

T: 0 0 1 1 1 2 | 2 3 3 4 4 4 4 | 5 5 5 5 5 6 | 6 6 7 7 8 9

6.4 Alternative Sorting Algorithms

Sorting via random sampling

Given a large list S of inputs, a random sample of the elements can be used to find k comparison thresholds

In fact, it is easier if we pick $k = p$, so that each of the resulting subproblems is handled by a single processor.

Assume $p << \sqrt{n}$:

Parallel randomized sort *PRAMrandomsort*(S, p)

1. Processor j, $0 \leq j < p$, pick $|S|/p^2$ random samples of its $|S|/p$ elements and store them in its corresponding section of a list T of length $|S|/p$
2. Processor 0 sort the list T
   {the comparison threshold m_i is
   the $(i\,|S|\,/\,p^2)$th element of T}
3. Processor j, $0 \leq j < p$, store its elements falling in $(m_i\,,\,m_{i+1})$ into $T^{(i)}$
4. Processor j, $0 \leq j < p$, sort the sublist $T^{(i)}$

Parallel radixsort

In binary version of *radixsort*, we examine every bit of the k-bit keys in turn, starting from the LSB

In Step i, bit i is examined, $0 \leq i < k$

Records are stably sorted by the value of the ith key bit

Example (keys are followed by their binary representations in parentheses):

Input list	Sort by LSB	Sort by middle bit	Sort by MSB
5 (101)	4 (100)	4 (100)	1 (001)
7 (111)	2 (010)	5 (101)	2 (010)
3 (011)	2 (010)	1 (001)	2 (010)
1 (001)	5 (101)	2 (010)	3 (011)
4 (100)	7 (111)	2 (010)	4 (100)
2 (010)	3 (011)	7 (111)	5 (101)
7 (111)	1 (001)	3 (011)	7 (111)
2 (010)	7 (111)	7 (111)	7 (111)

Performing the required data movements

Input list	Compl. of Bit 0	Diminished prefix sums	Bit 0	Prefix sums plus 2	Shifted list
5 (101)	0	–	1	1 + 2 = 3	4 (100)
7 (111)	0	–	1	2 + 2 = 4	2 (010)
3 (011)	0	–	1	3 + 2 = 5	2 (010)
1 (001)	0	–	1	4 + 2 = 6	5 (101)
4 (100)	1	0	0	–	7 (111)
2 (010)	1	1	0	–	3 (011)
7 (111)	0	–	1	5 + 2 = 7	1 (001)
2 (010)	1	2	0	–	7 (111)

The running time consists mainly of the time to perform $2k$ parallel prefix computations: $O(\log p)$ for k constant

6.5 Convex Hull of a 2D Point Set

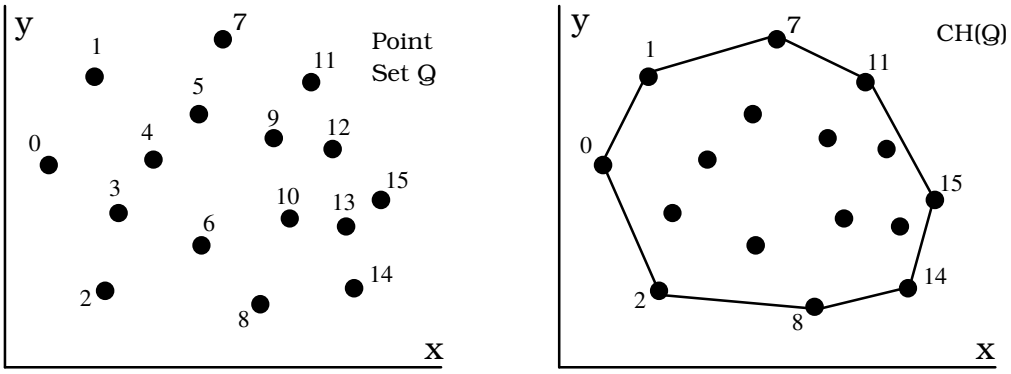


Fig. 6.2. Defining the convex hull problem.

Best sequential algorithm for p points: $\Omega(p \log p)$ steps

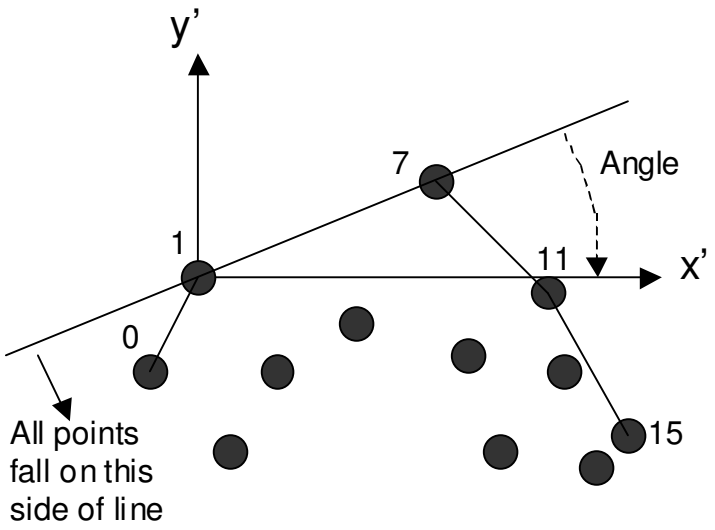


Fig. 6.3. Illustrating the properties of the convex hull.

Parallel convex hull algorithm *PRAMconvexhull*(*S*, *p*)

1. Sort point set by *x* coordinates
2. Divide sorted list into $\sqrt{p}$ subsets $Q^{(i)}$ of size $\sqrt{p}$, $0 \le i < \sqrt{p}$
3. Find convex hull of each subset $Q^{(i)}$ using $\sqrt{p}$ processors
4. Merge $\sqrt{p}$ convex hulls CH($Q^{(i)}$) into overall hull CH(Q)

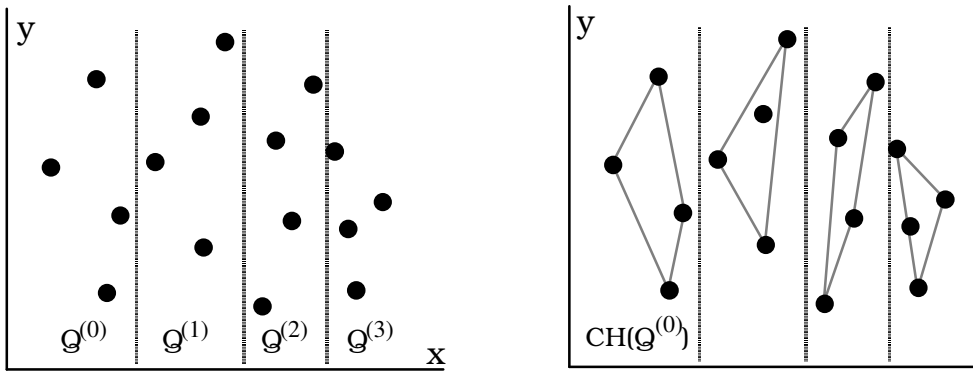
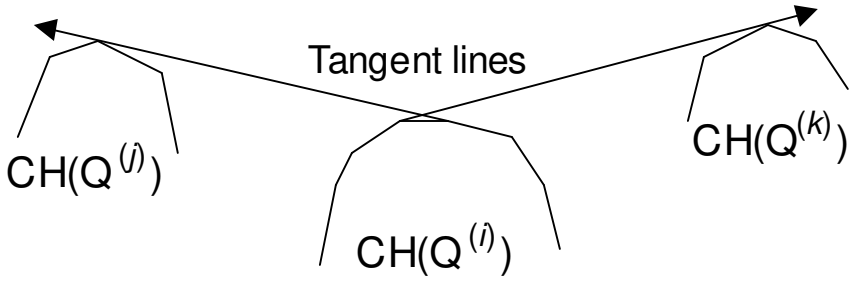
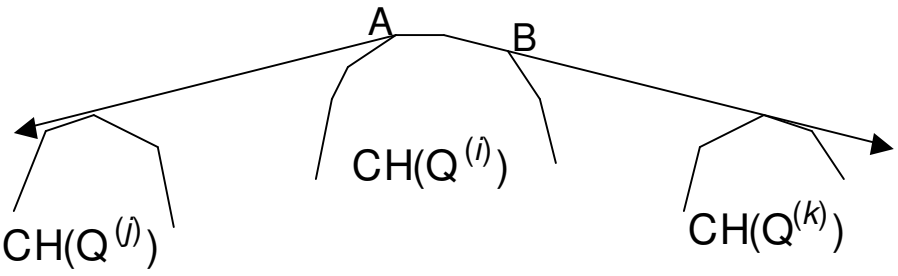


Fig. 6.4. Multiway divide and conquer for the convex hull problem.



(a) No point of CH(Q(*i*)) is on CH(Q)



(b) Points of CH(Q(*i*)) from A to B are on CH(Q)

Fig. 6.5. Finding points in a partial hull that belong to the combined hull.

Analysis:

$$T(p, p) = T(p^{1/2}, p^{1/2}) + c \log p \cong 2c \log p$$

The initial sorting time is also O(log *p*)

6.6 Some Implementation Aspects

EREW-PRAM: Any *p* locations accessible by *p* processors

Realistic: *p* locations must be in different memory modules

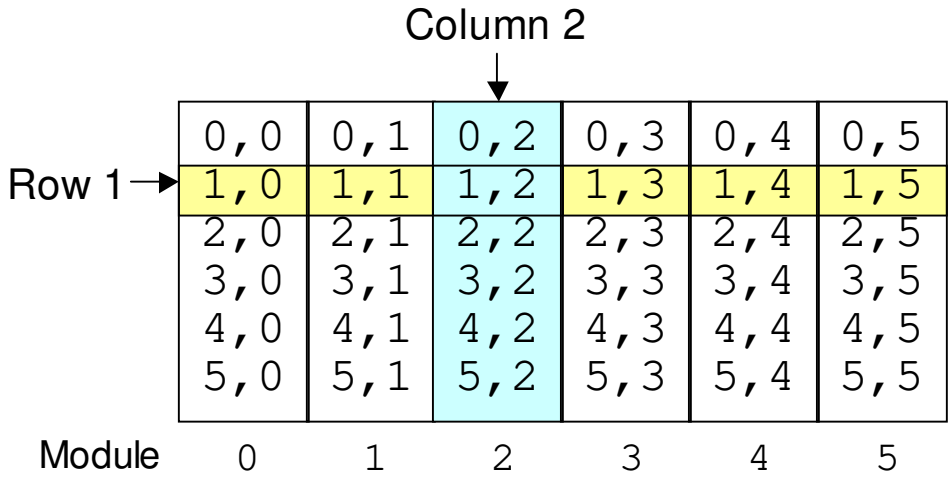


Fig. 6.6. Matrix storage in column-major order to allow concurrent accesses to rows.

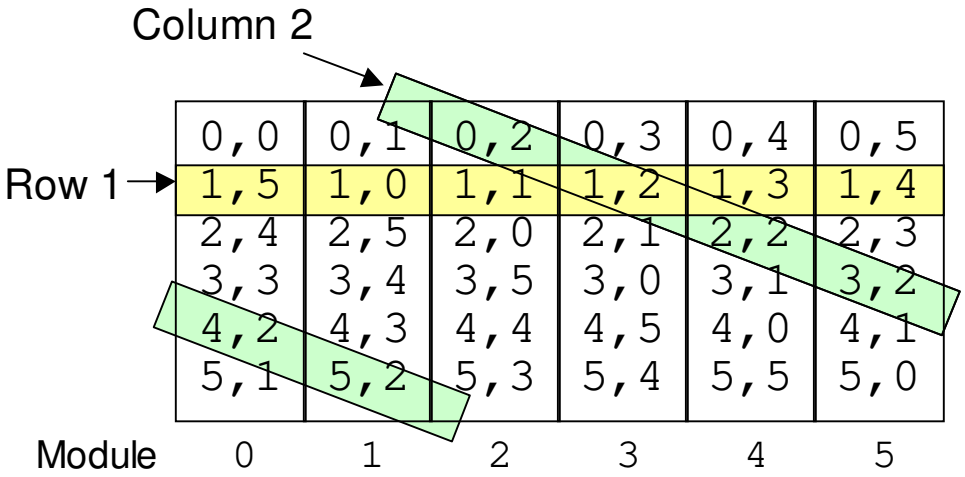


Fig. 6.7. Skewed matrix storage for conflict-free accesses to rows and columns.

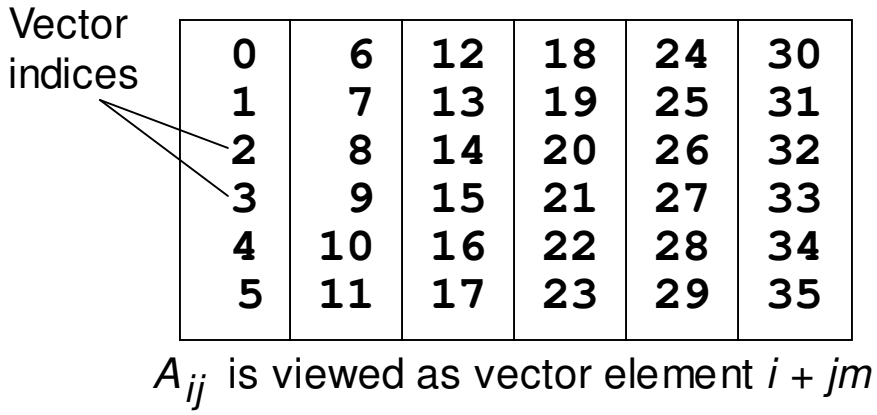


Fig. 6.8. **A 6 × 6 matrix viewed, in column-major order, as a 36-element vector.**

The vector in Fig. 6.8 may be accessed in some or all of the following ways

Column:	$k, k+1, k+2, k+3, k+4, k+5$	Stride = 1
Row:	$k, k+m, k+2m, k+3m, k+4m, k+5m$	Stride = m
Diagonal:	$k, k+m+1, k+2(m+1), k+3(m+1), k+4(m+1), k+5(m+1)$	Stride = $m+1$
Antidiagonal:	$k, k+m-1, k+2(m-1), k+3(m-1), k+4(m-1), k+5(m-1)$	Stride = $m-1$

Linear skewing scheme:

stores the kth vector element in bank $a + kb \bmod B$

The address within the bank is irrelevant to conflict-free parallel access

In fact, the constant a above is also irrelevant and can be safely ignored

So we can limit our attention to linear skewing schemes that assign V_k to memory module $M_{kb \bmod B}$

With a linear skewing scheme, the vector elements k, $k+s$, $k+2s$, $\cdots$, $k+(B-1)s$ will be assigned to different memory modules iff sb is relatively prime with respect to the number B of memory banks.

To allow access from each processor to every memory bank, we need a permutation network

Even with a full permutation network (complex, expensive), full PRAM functionality is not realized

Practical processor-to-memory network cannot realize all permutations (they are *blocking*)

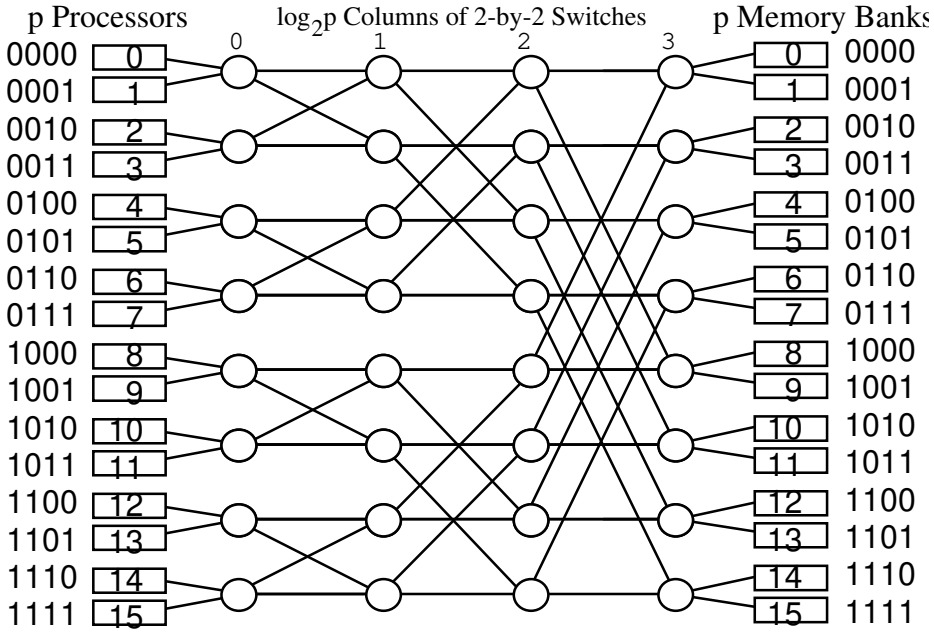


Fig. 6.9. Example of a multistage memory access network.

7 Sorting and Selection Networks

Back to TOC

Chapter Goals

- Become familiar with the circuit-level models of parallel processing
- Architecture $\Rightarrow$ algorithm (studied so far)
  Problem $\Rightarrow$ develop a suitable architecture
  (three more application-specific examples to come in Chapter 8)
- Introduce useful design tools and study trade-off issues via a familiar problem

Chapter Contents

- 7.1. What Is a Sorting Network?
- 7.2. Figures of Merit for Sorting Networks
- 7.3. Design of Sorting Networks
- 7.4. Batcher Sorting Networks
- 7.5. Other Classes of Sorting Networks
- 7.6. Selection Networks

7.1 What Is a Sorting Network?

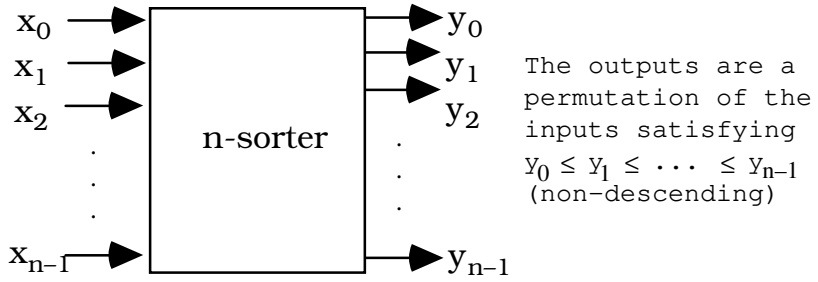


Fig. 7.1. An *n*-input sorting network or an *n*-sorter.

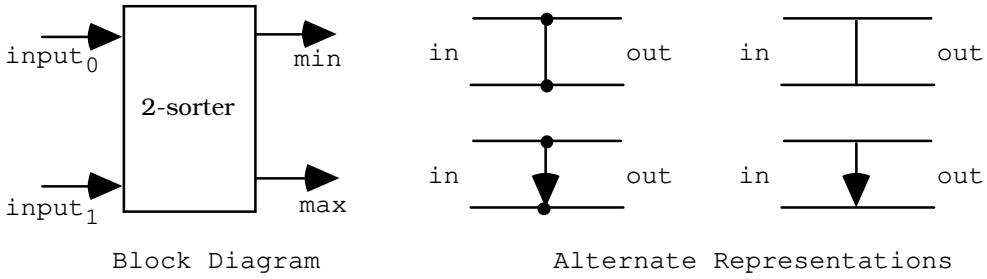


Fig. 7.2. Block diagram and four different schematic representations for a 2-sorter.

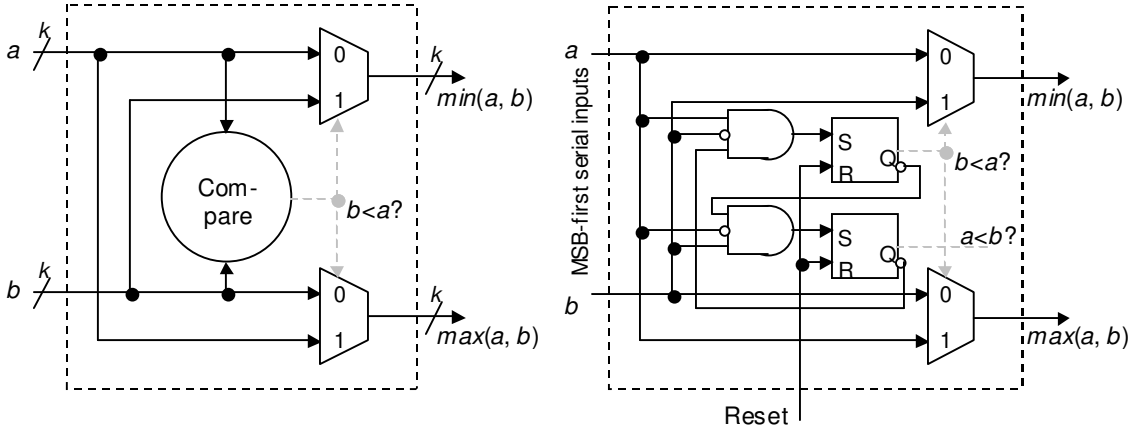


Fig. 7.3. Parallel and bit-serial hardware realizations of a 2-sorter.

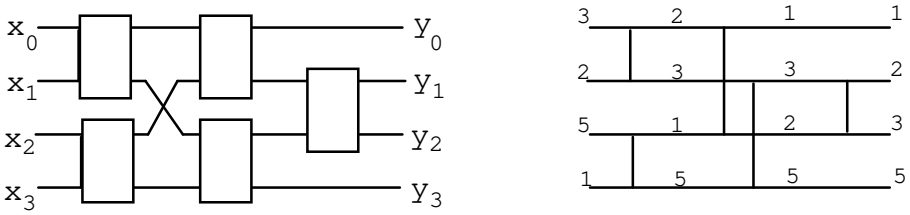


Fig. 7.4. Block diagram and schematic representation of a 4-sorter.

How to verify that the circuit of Fig. 7.4 is a valid 4-sorter?

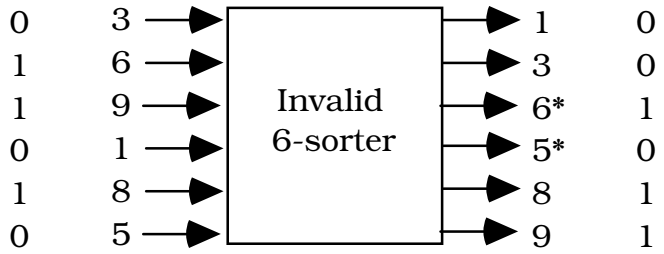
The answer is easy in this case

After the first two circuit levels, the top line carries the smallest and the bottom line the largest of the four values

The final 2-sorter orders the middle two values

More generally, we need to verify the correctness of an *n*-sorter through formal proofs or by time-consuming exhaustive testing. Neither approach is attractive.

The zero-one principle: A comparison-based sorter is valid iff it correctly sorts all 0/1 sequences.

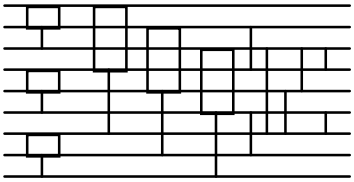


7.2 Figures of Merit for Sorting Networks

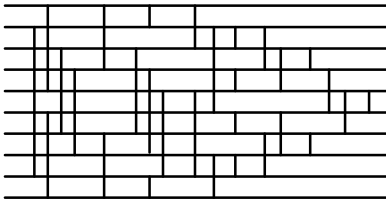
a. Cost: number of 2-sorters used in the design

b. Delay: number of 2-sorters on the critical path

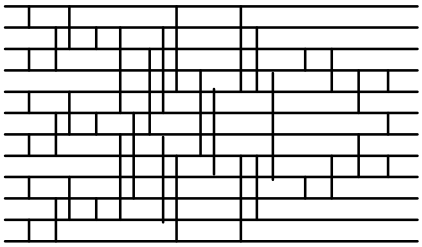
c. Cost $\times$ Delay



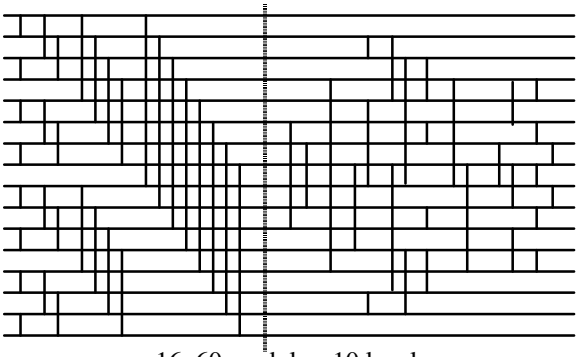
n = 9, 25 modules, 9 levels



n = 10, 29 modules, 9 levels



n = 12, 39 modules, 9 levels



n = 16, 60 modules, 10 levels

Fig. 7.5. Some low-cost sorting networks.

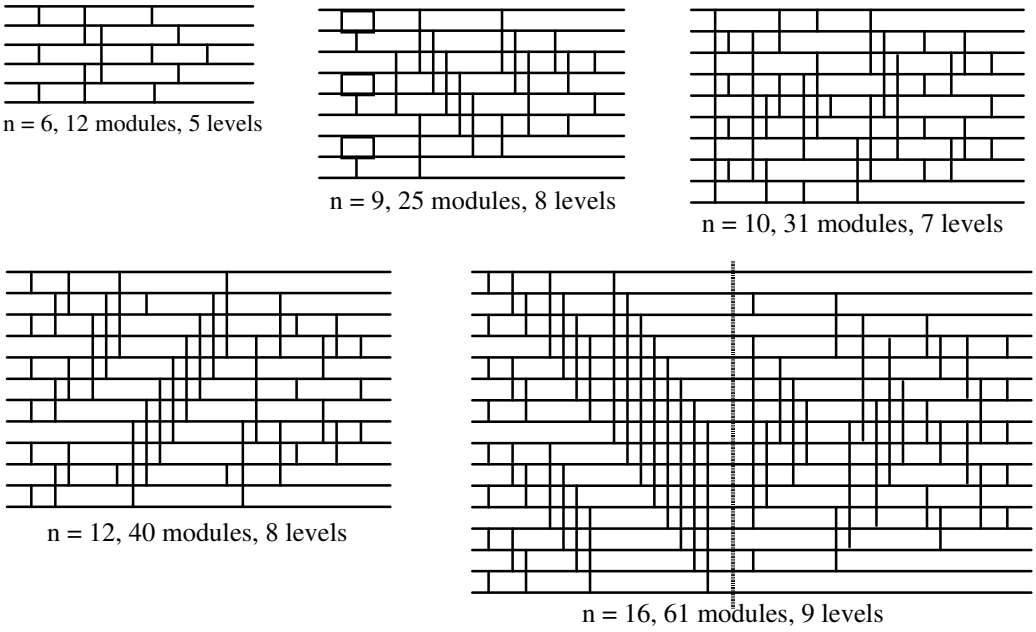


Fig. 7.6. Some fast sorting networks.

7.3 Design of Sorting Networks

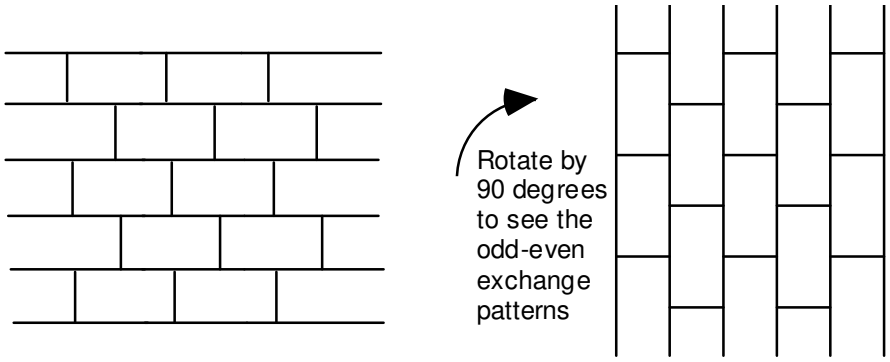
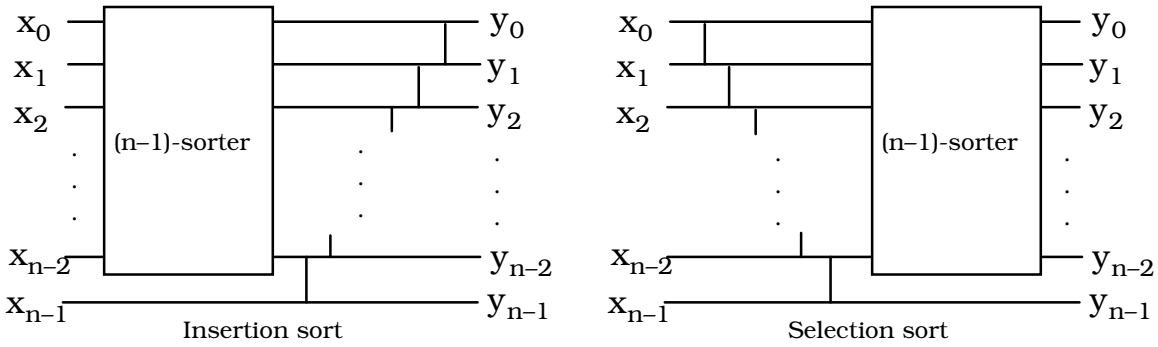


Fig. 7.7. Brick-wall 6-sorter based on odd–even transposition.

$$C(n) = C(n-1) + n - 1 = (n-1) + (n-2) + \cdots + 2 + 1 = n(n-1)/2$$
$$D(n) = D(n-1) + 2 = 2 + 2 + \cdots + 2 + 1 = 2(n-2) + 1 = 2n - 3$$
$$\text{Cost} \times \text{Delay} = n(n-1)(2n-3)/2 = \Theta(n^3)$$



Parallel insertion sort = Parallel selection sort = Parallel bubble sort!

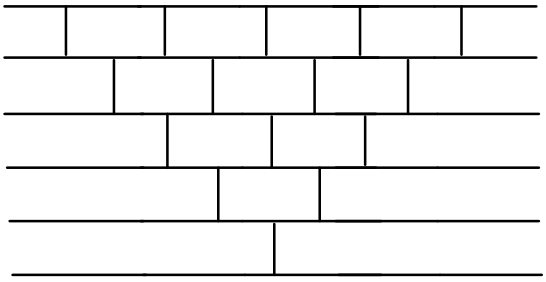


Fig. 7.8. Sorting network based on insertion sort or selection sort.

7.4 Batcher Sorting Networks

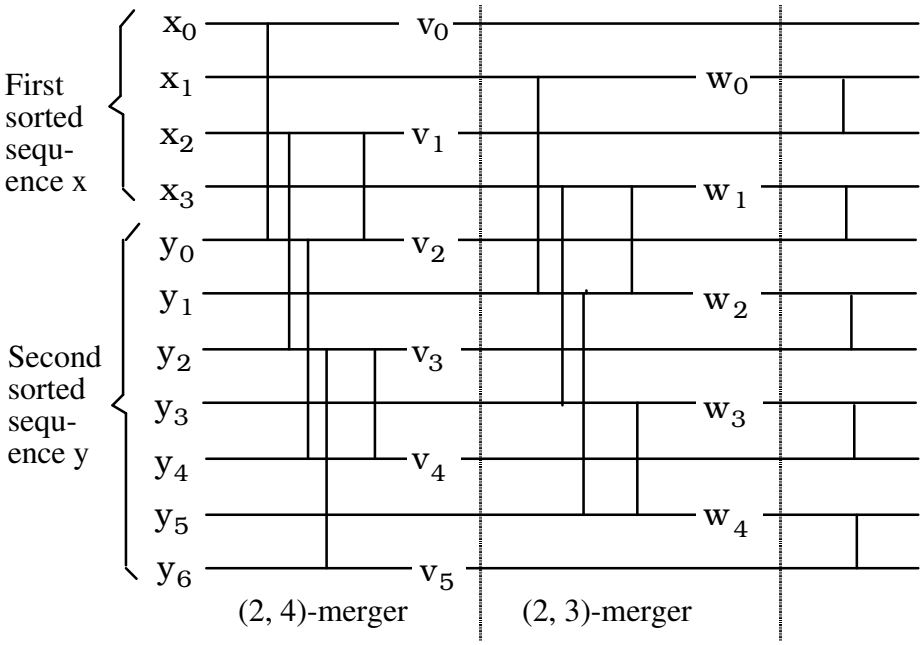


Fig. 7.9. Batcher's even–odd merging network for 4 + 7 inputs.

$x_0 \le x_1 \le \cdots \le x_{m-1}$ (k 0s) $y_0 \le y_1 \le \cdots \le y_{m'-1}$ (k' 0s)

Merge $x_0, x_2, \cdots$ and $y_0, y_2, \cdots$ to get $v_0, v_1, \cdots$ $k_{even} = \lceil k/2 \rceil + \lceil k'/2 \rceil$ 0s

Merge $x_1, x_3, \cdots$ and $y_1, y_3, \cdots$ to get $w_0, w_1, \cdots$ $k_{odd} = \lfloor k/2 \rfloor + \lfloor k'/2 \rfloor$ 0s

Compare-exchange the pairs of elements $w_0{:}v_1, w_1{:}v_2, w_2{:}v_3, \cdots$

Case a: $k_{even} = k_{odd}$ The sequence $v_0\ w_0\ v_1\ w_1\ v_2\ w_2 \cdots$ already sorted

Case b: $k_{even} = k_{odd}+1$ The sequence $v_0\ w_0\ v_1\ w_1\ v_2\ w_2 \cdots$ already sorted

Case c: $k_{even} = k_{odd}+2$

v	0	0	0	0	0	0	0	0	1	1	1	1	
w		0	0	0	0	0	0	1	1	1	1	1	

Out of order

Batcher's (m, m) even-odd merger, when m is a power of 2, is characterized by the following recurrences:

$$\begin{aligned} C(m) &= 2C(m/2) + m - 1 \\ &= (m-1) + 2(m/2-1) + 4(m/4-1) + \cdots \\ &= m \log_2 m + 1 \end{aligned}$$

$$\begin{aligned} D(m) &= D(m/2) + 1 \\ &= \log_2 m + 1 \end{aligned}$$

$$\text{Cost} \times \text{Delay} = \Theta(m \log^2 m)$$

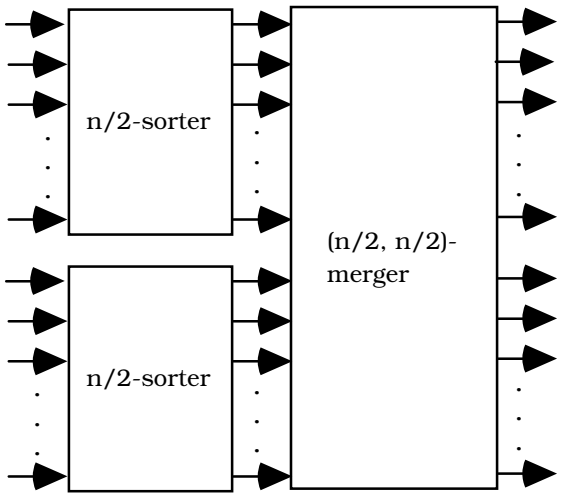


Fig. 7.10. The recursive structure of Batcher's even–odd merge sorting network.

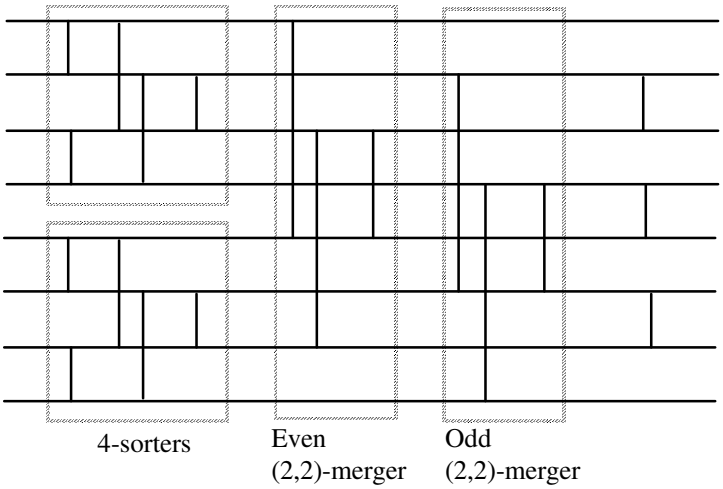


Fig. 7.11. Batcher's even-odd merge sorting network for eight inputs.

Batcher sorting networks based on the even-odd merge technique are characterized by the following recurrences:

$$C(n) = 2C(n/2) + (n/2)(\log_2(n/2)) + 1$$

$$\cong n(\log_2 n)^2/2$$

$$D(n) = D(n/2) + \log_2(n/2) + 1$$

$$= D(n/2) + \log_2 n$$

$$= \log_2 n\,(\log_2 n + 1)/2$$

$$\text{Cost} \times \text{Delay} = \Theta(n \log^4 n)$$

Bitonic sorters

Bitonic sequence: "rises then falls", "falls then rises", or is obtained from the first two categories through cyclic shifts or rotations. Examples include:

1 3 3 4 6 6 6 2 2 1 0 0 Rises, then falls

8 7 7 6 6 6 5 4 6 8 8 9 Falls, then rises

8 9 8 7 7 6 6 6 5 4 6 8 The previous sequence, right-rotated by 2

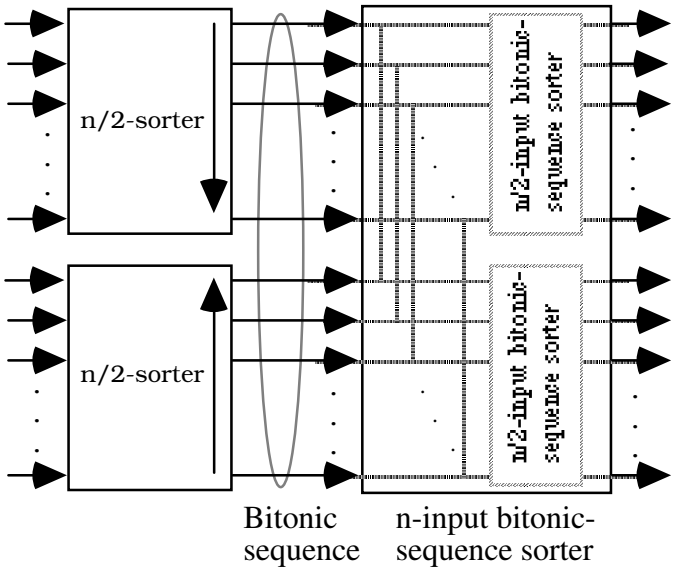


Fig. 7.12. The recursive structure of Batcher's bitonic sorting network.

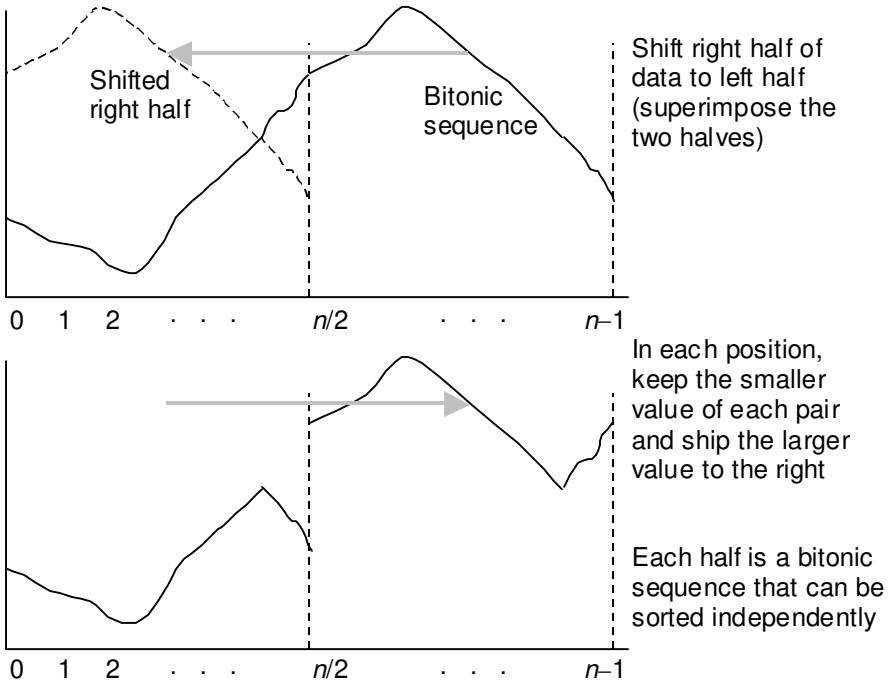


Fig. 14.2. Sorting a bitonic sequence on a linear array.

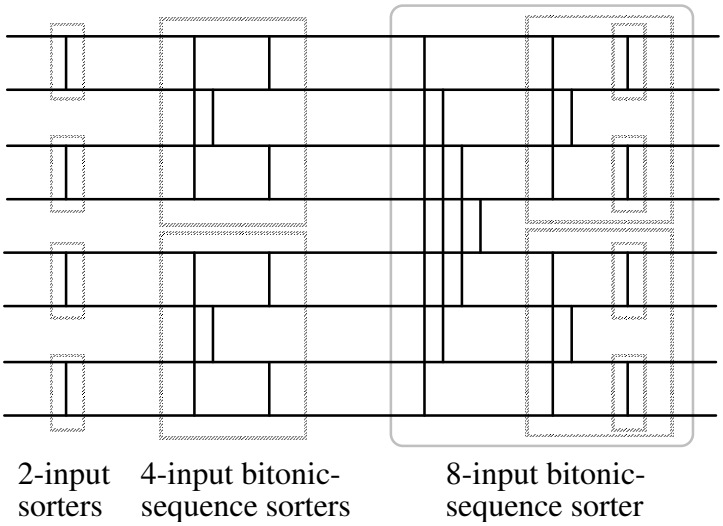


Fig. 7.13. Batcher's bitonic sorting network for eight inputs.

7.5 Other Classes of Sorting Networks

Periodic balanced sorting networks

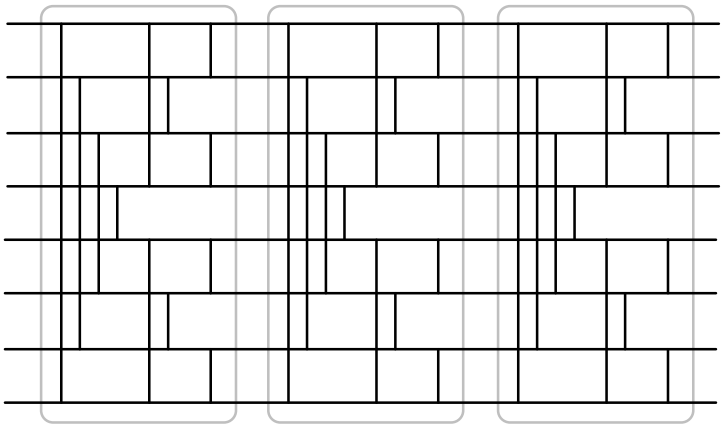


Fig. 7.14. Periodic balanced sorting network for eight inputs.

Desirable properties:

a. Regular and modular (easier VLSI layout).

b. Slower, but more economical, implementations are possible by reusing the blocks

c. Using an extra block provides tolerance to some faults (missed exchanges)

d. Using 2 extra blocks provides tolerance to any single fault (a missed or incorrect exchange)

e. Multiple passes through a faulty network can lead to correct sorting (graceful degradation)

f. Single-block design can be made fault-tolerant by adding an extra stage to the block

Shearsort-based sorting networks

Offer some of the same advantages enumerated for periodic balanced sorting networks

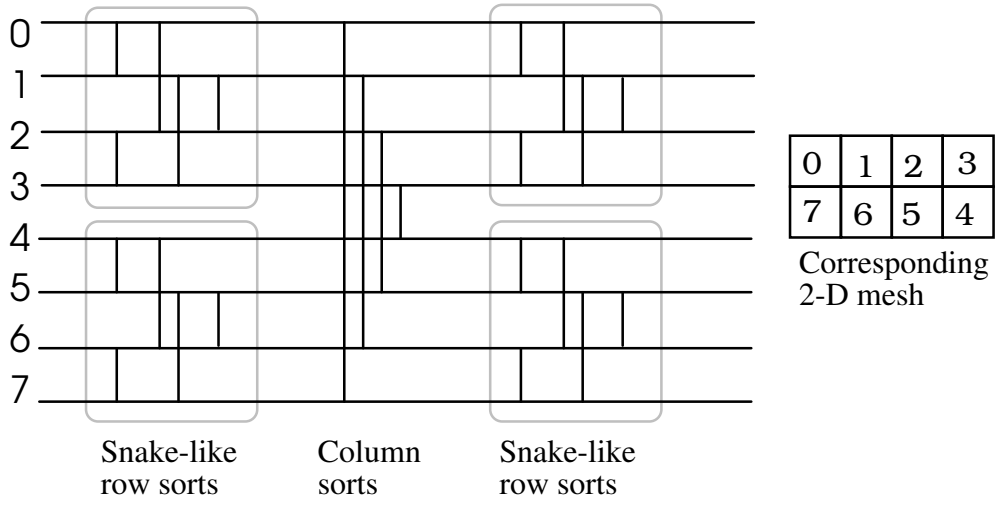


Fig. 7.15. Design of an 8-sorter based on shearsort on 2×4 mesh.

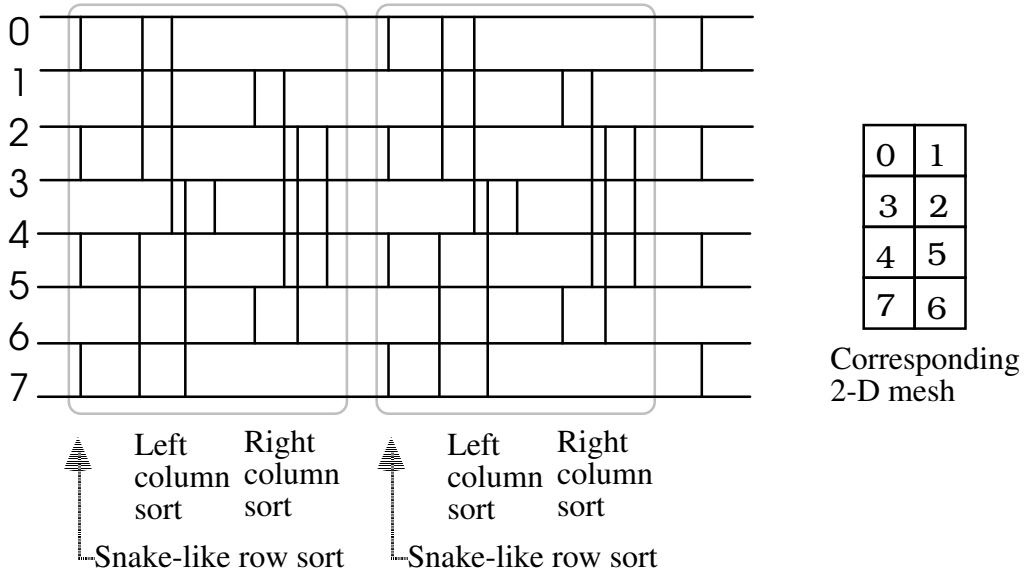
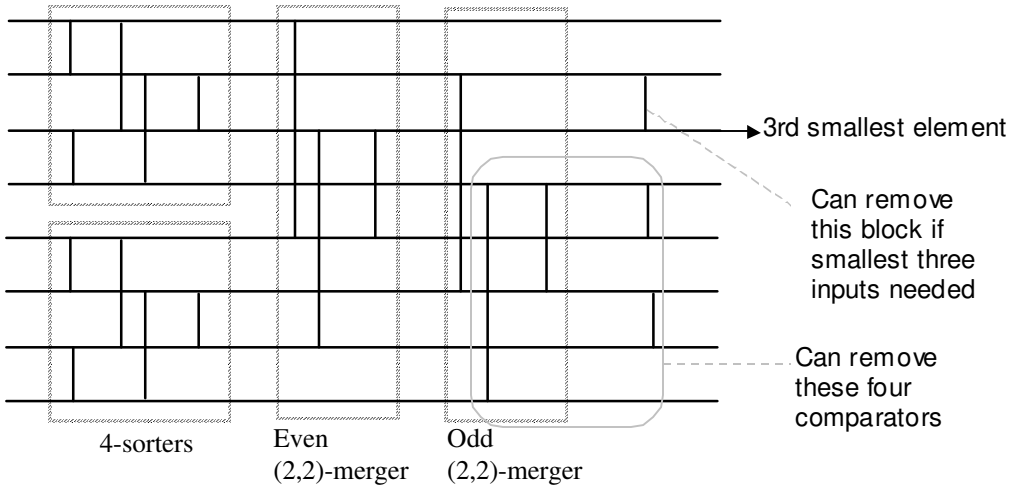


Fig. 7.16. Design of an 8-sorter based on shearsort on 4×2 mesh.

7.6 Selection Networks

Any sorting network can be used as a selection network, but a selection network (yielding the *k*th smallest or largest input value) is in general simpler and faster

One way to get a selection network is by pruning a sorting network



Deriving an (8, 3)-selector from Batcher's even-odd merge 8-sorter.

Direct design is likely to lead to more efficient networks, but unfortunately we know even less about selection networks than we do about sorting networks.

One can define three selection problems:

I. Select the *k* smallest values; present in sorted order
II. Select *k*th smallest value
III. Select the *k* smallest values; present in any order

Circuit and time complexity: (I) hardest, (III) easiest

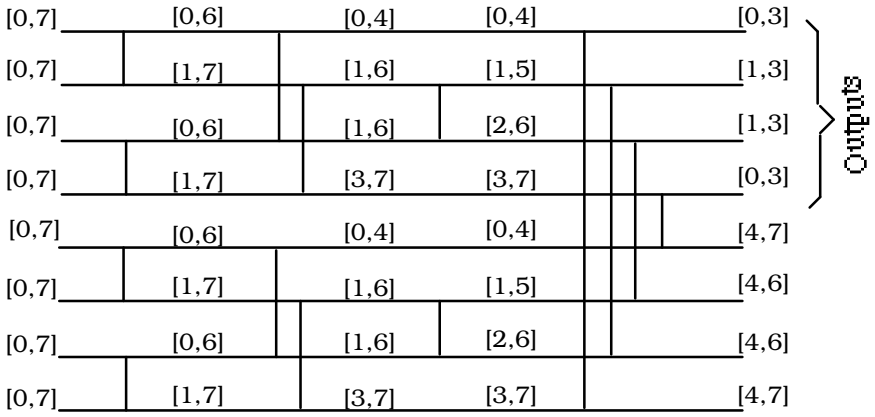


Fig. 7.17. A type III (8, 4)-selector.

Classifier: a selection network that can divide a set of *n* values into *n*/2 largest and *n*/2 smallest values

The selection network of Fig. 7.17 is an 8-input classifier

Generalizing from Fig. 7.17, an *n*-input classifier can be built from two (*n*/2)-sorters followed by *n*/2 comparators

An *n*-classifier and two *n*/2-sorters can form an *n*-sorter. For such a sorting network:

$$T(n) = 2T(n/2) + 1 = n - 1$$

$$C(n) = 4C(n/2) + n/2 = n(n-1)/2$$

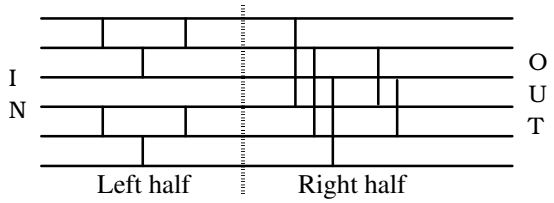


Figure for Problem 7.7.

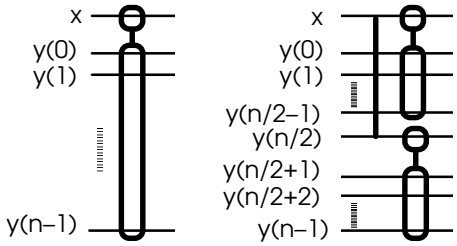


Figure for Problem 7.9.

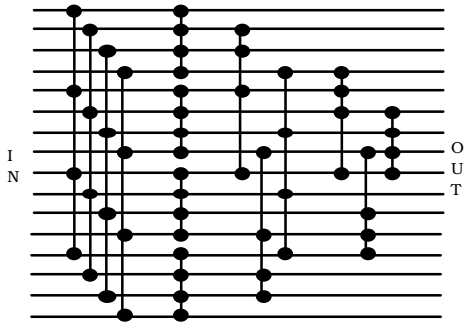


Figure for Problem 7.11.

8 Other Circuit-Level Examples

Back to TOC

Chapter Goals

- Study three application areas: dictionary operations, parallel prefix, DFT
- Develop circuit-level parallel architectures for solving these problems:
  - Tree machine
  - Parallel prefix networks
  - FFT circuits

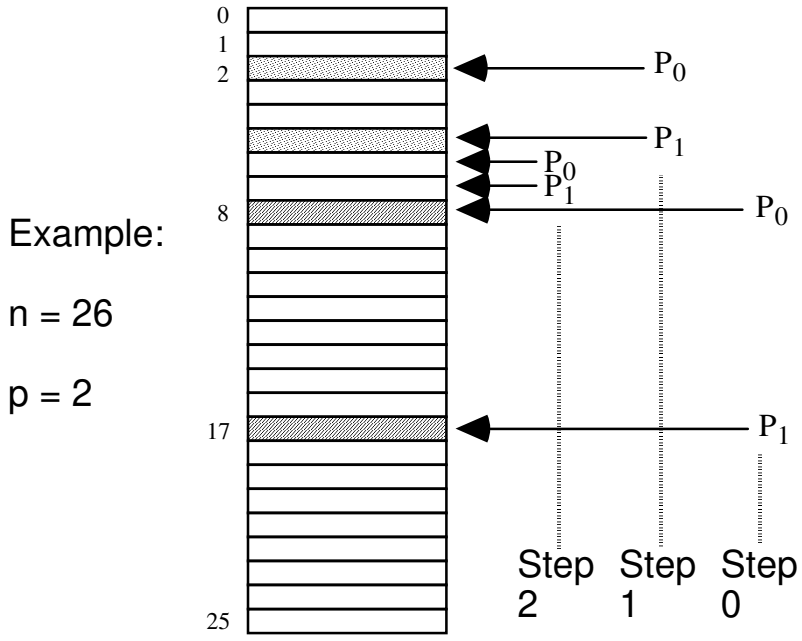
Chapter Contents

- 8.1. Searching and Dictionary Operations
- 8.2. A Tree-Structured Dictionary Machine
- 8.3. Parallel Prefix Computation
- 8.4. Parallel Prefix Networks
- 8.5. The Discrete Fourier Transform
- 8.6. Parallel Architectures for FFT

8.1 Searching and Dictionary Operations

Parallel $(p + 1)$-ary search:

$$\log_{p+1}(n + 1) = \log_2(n + 1)/\log_2(p + 1) \text{ steps}$$



This algorithm is optimal: no comparison-based search algorithm can be faster

Speed-up $\cong \log_2(p + 1)$

A single search in a sorted list cannot be significantly speeded up by parallel processing, but all hope is not lost

Dynamic data sets (sorting implies large overhead)

Batch searching (finding multiple keys at once)

Basic dictionary operations: record keys $x_0, x_1, \cdots, x_{n-1}$

search(y)	Find record with key y and return its data
insert(y, z)	Augment list with a record: key = y, data = z
delete(y)	Remove record with key y, return data

Some or all of the following ops might also be of interest:

findmin	Find record with smallest key; return data
findmax	Find record with largest key; return data
findmed	Find record with median key; return data
findbest(y)	Find record with key "nearest" to y
findnext(y)	Find record whose key would appear immediately after y if ordered
findprev(y)	Find record whose key would appear immediately before y if ordered
extractmin	Remove record(s) with min key; return data?
extractmax	Remove record(s) with max key; return data?
extractmed	Remove the record(s) with median key value; return data?

The operations "findmin" and "extractmin" (or "findmax" and "extractmax") are priority queue operations

8.2 A Tree-Structured Dictionary Machine

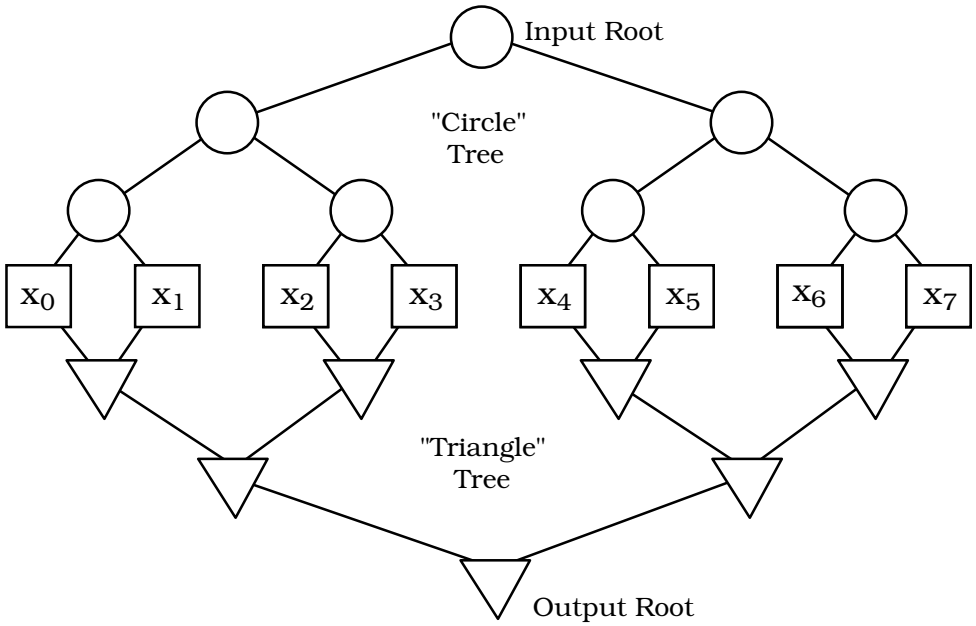


Fig. 8.1. A tree-structured dictionary machine.

Combining function of the triangular nodes is as follows:

search(*y*) Pass OR of "yes" signals, with data from "yes" side, or from either side if both "yes"

findmin Pass smaller of two key values, with data (*findmax* is similar; *findmed* not supported)

findbest(*y*) Pass the larger of two match-degree indicators, with the corresponding record

findnext(*y*) Leaf nodes generate a "larger" flag bit; *findmin* is performed among all larger values (*findprev* is similar)

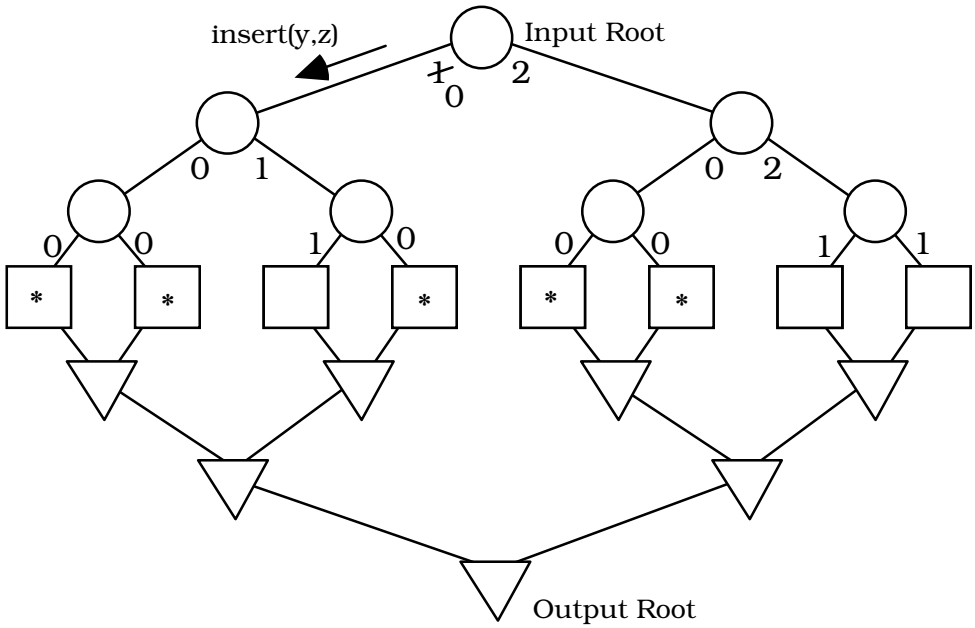


Fig. 8.2. Tree machine storing five records and containing three free slots.

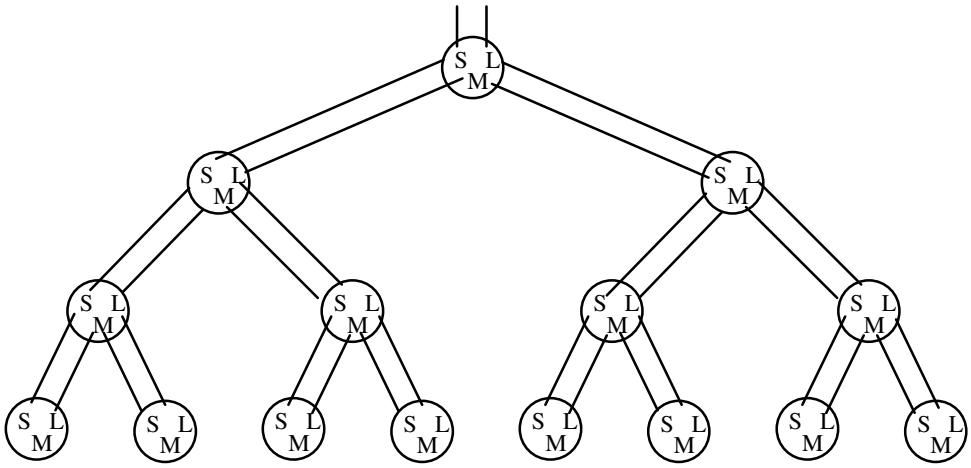
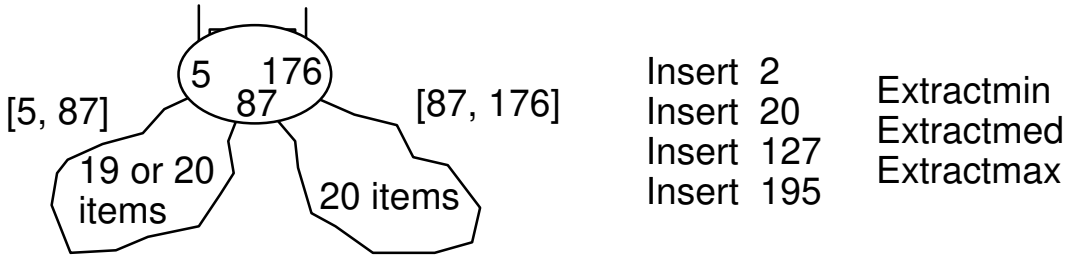


Fig. 8.3. Systolic data structure for minimum, maximum, and median finding.



Update/access examples for the systolic data structure of Fig. 8.3.

8.3 Parallel Prefix Computation

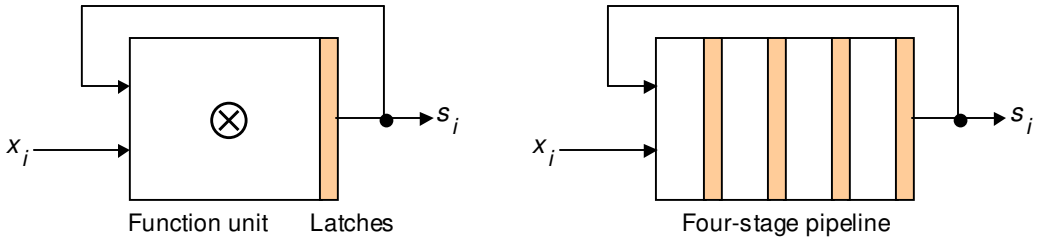


Fig. 8.4. Prefix computation using a latched or pipelined function unit.

Example: Prefix sums

x_0	x_1	x_2	. . .	x_i
x_0	$x_0 + x_1$	$x_0 + x_1 + x_2$	. . .	$x_0 + x_1 + . . . + x_i$
s_0	s_1	s_2	. . .	s_i

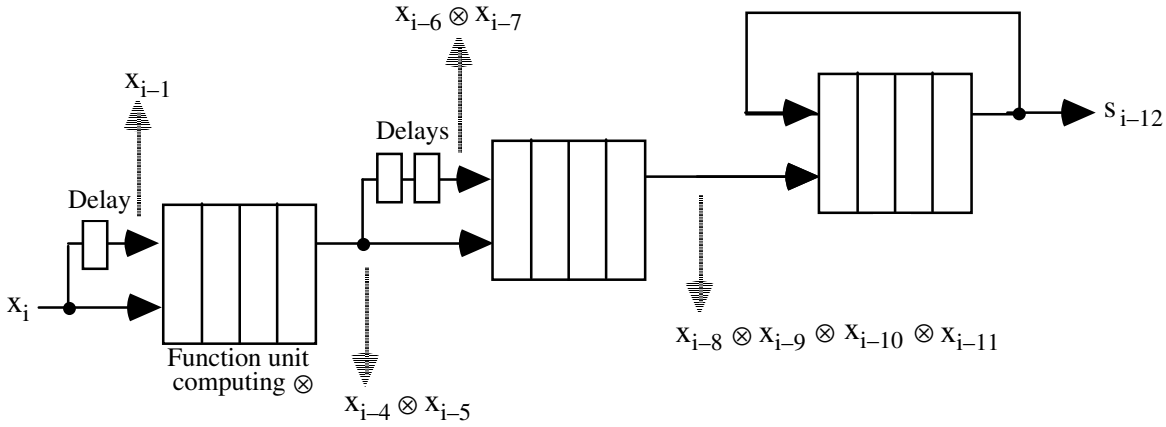


Fig. 8.5. High-throughput prefix computation using a pipelined function unit.

8.4 Parallel Prefix Networks

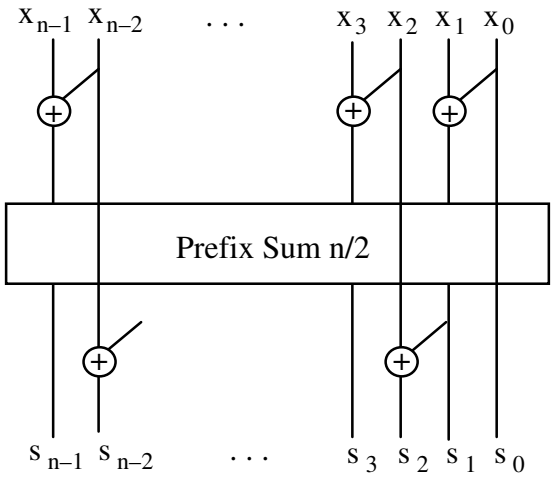


Fig. 8.6. Prefix sum network built of one *n*/2-input networks and *n* – 1 adders.

$$T(n) = T(n/2) + 2 = 2\log_2 n - 1$$
$$C(n) = C(n/2) + n - 1 = 2n - 2 - \log_2 n$$

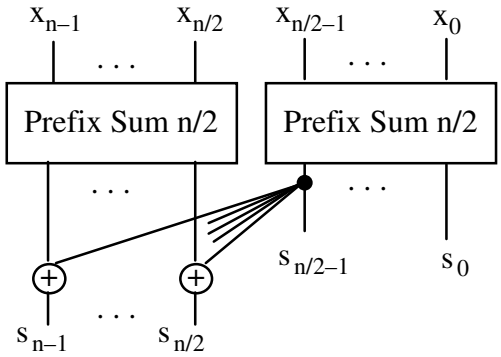


Fig. 8.7. Prefix sum network built of two *n*/2-input networks and *n*/2 adders.

$$T(n) = T(n/2) + 1 = \log_2 n$$
$$C(n) = 2C(n/2) + n/2 = (n/2)\log_2 n$$

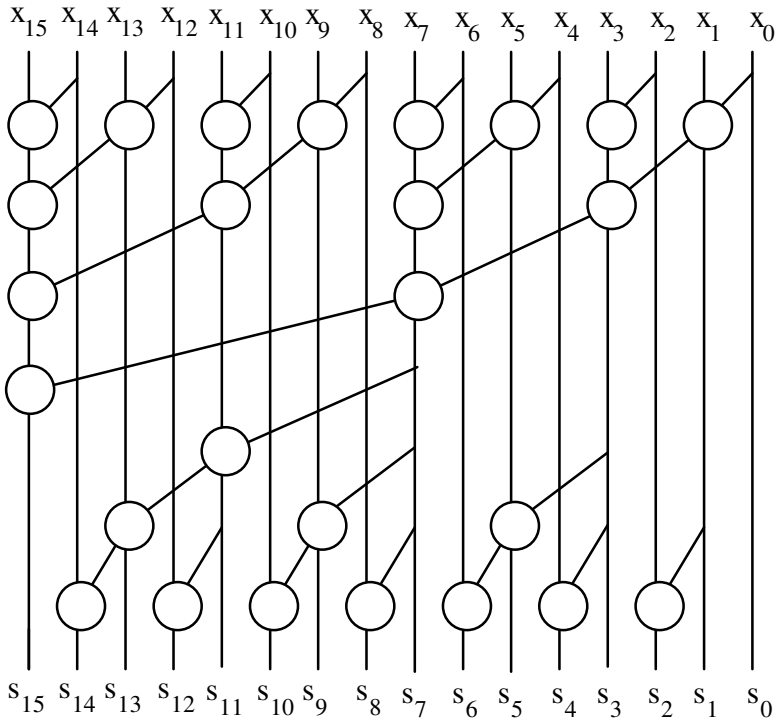


Fig. 8.8. Brent–Kung parallel prefix graph for *n* = 16.

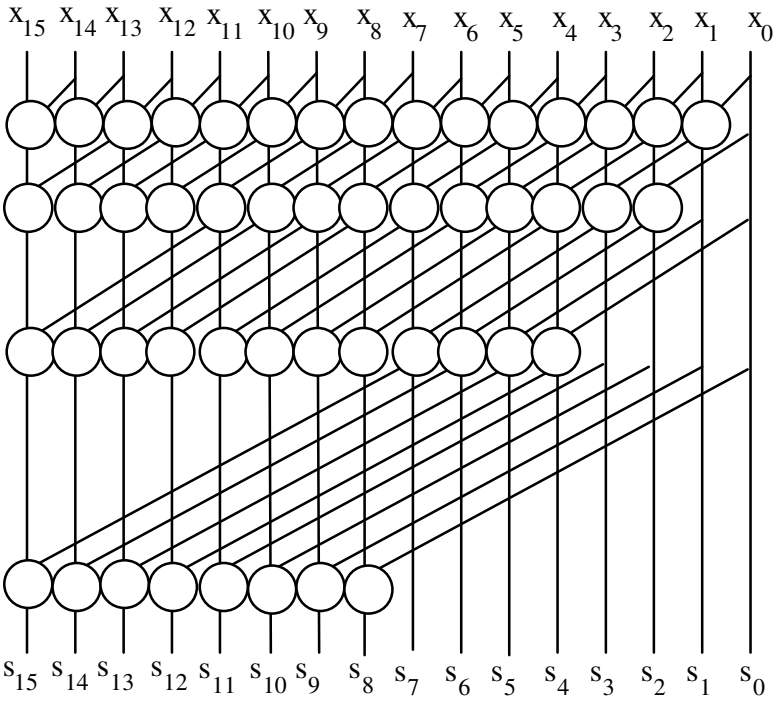


Fig. 8.9. Kogge–Stone parallel prefix graph for *n* = 16.

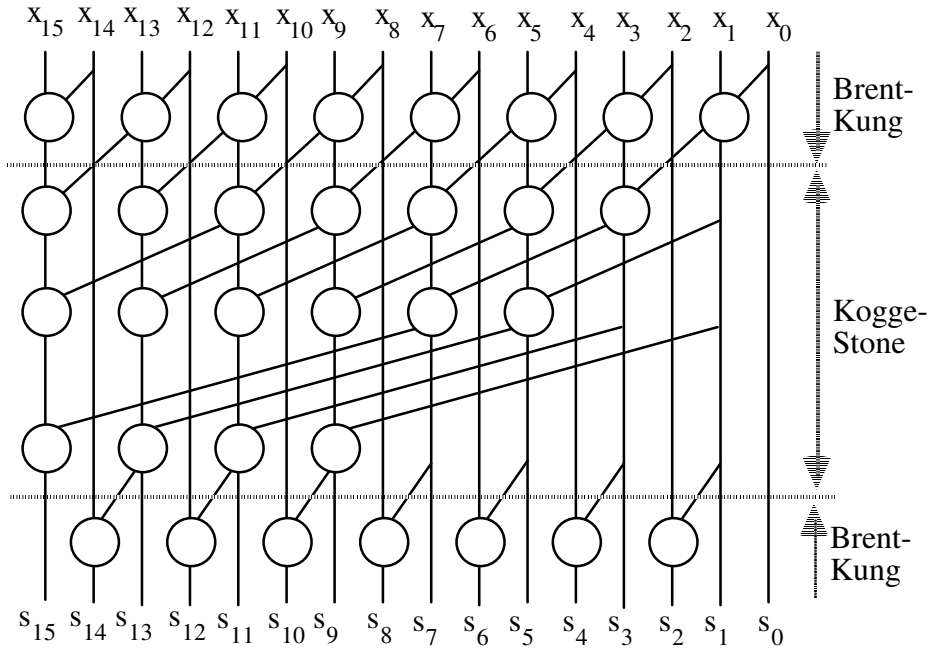


Fig. 8.10. A hybrid Brent–Kung/Kogge–Stone parallel prefix graph for 16 inputs.

Brent-Kung: $\cong 2n$ cost, $2 \log_2 n - 2$ delay

Kogge-Stone: $\cong n \log_2 n$ cost, $\log_2 n$ delay

Hybrid: intermediate in cost and delay

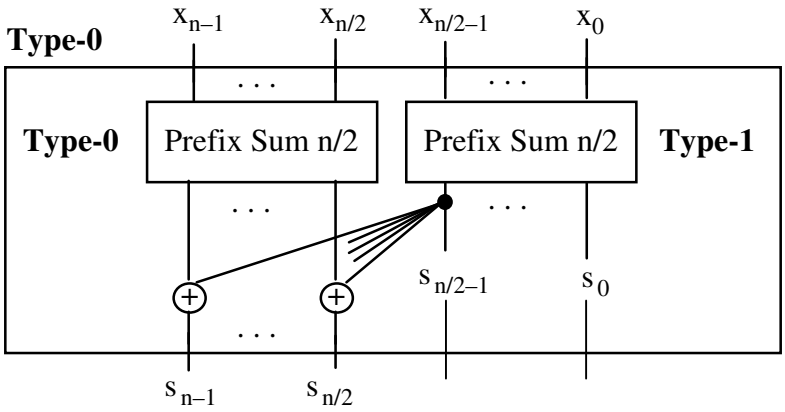
Linear-cost, $\log_2 n$-delay parallel prefix networks

Define a type-x parallel prefix network as one that:

Produces the leftmost output in $\log_2(n)$ time

Yields all other outputs with at most x additional delay

Recursive construction of the fastest possible parallel prefix networks (type-0)



8.5 The Discrete Fourier Transform

$$y_i = \sum_{j=0}^{n-1} \omega_n^{ij} x_j$$

The DFT is expressed in matrix form as $y = F_n x$

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega_n & \omega_n^2 & \ldots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \ldots & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

ω_n: nth primitive root of unity; $\omega_n^n = 1$, $\omega_n^j \neq 1$ for $1 \leq j < n$

Examples: $\omega_4 = i = \sqrt{-1}$, $\omega_3 = -1/2 + i\sqrt{3}/2$

Inverse DFT, for recovering x, given y, is essentially the same computation as DFT:

$$x_i = \frac{1}{n} \sum_{j=0}^{n-1} \omega_n^{-ij} y_j$$

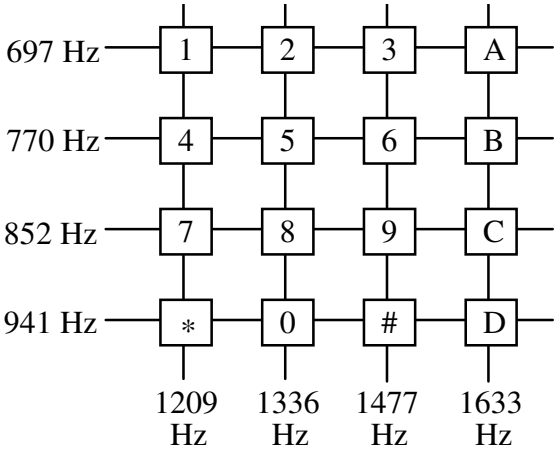
Can do DFT by any matrix-vector multiplication algorithm

However, the special structure of F_n can be exploited to devise a much faster divide-and-conquer algorithm:

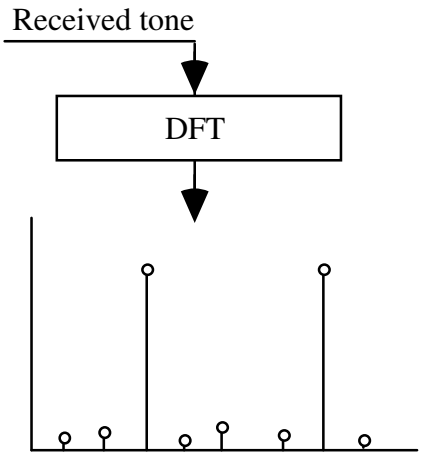
the fast Fourier transform (FFT)

DFT Applications

Spectral analysis

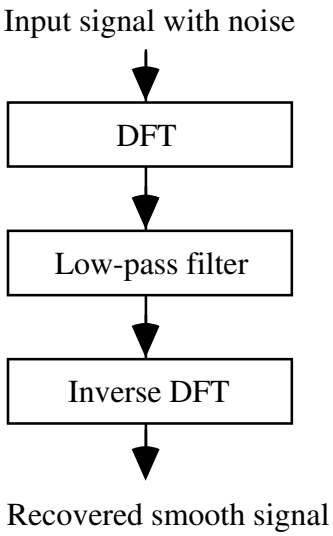


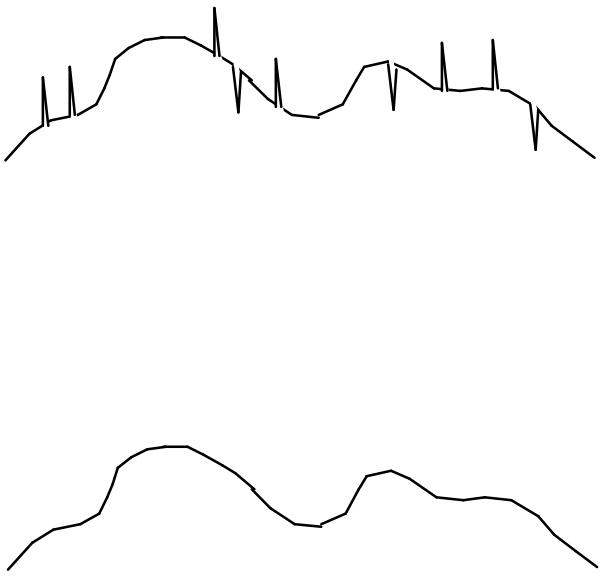
Tone frequency assignments for touch-tone dialing



Frequency spectrum of received tone

Signal smoothing or filtering





Fast Fourier Transform (FFT)

Partition the DFT sum into odd- and even-indexed terms

$$y_i = \sum_{j=0}^{n-1} \omega_n^{ij} x_j = \sum_{j \text{ even } (2r)} \omega_n^{ij} x_j + \sum_{j \text{ odd } (2r+1)} \omega_n^{ij} x_j$$

$$= \sum_{r=0}^{n/2-1} \omega_{n/2}^{ir} x_{2r} + \omega_n \sum_{r=0}^{n/2-1} \omega_{n/2}^{ir} x_{2r+1}$$

The identity $\omega_{n/2} = \omega_n^2$ has been used in the derivation

The two terms in the last expression are $n/2$-point DFTs

$$u = F_{n/2} \begin{bmatrix} x_0 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-2} \end{bmatrix} \qquad v = F_{n/2} \begin{bmatrix} x_1 \\ x_3 \\ \vdots \\ \vdots \\ x_{n-1} \end{bmatrix}$$

Then:

$$y_i = \begin{cases} u_i + \omega_n^i v_i & 0 \le i < n/2 \\ u_{i-n/2} + \omega_n^i v_{i-n/2} & n/2 \le i < n \text{ (or } y_{i+n/2} = u_i + \omega_n^{i+n/2} v_i) \end{cases}$$

Hence: n-point FFT = two $n/2$-point FFTs + n multiply-adds

Sequential complexity of FFT: $T(n) = 2T(n/2) + n = n \log_2 n$

Unit of time = latency of one multiply-add operation

If the $n/2$-point subproblems are solved in parallel and the n multiply-add operations are also concurrent, with their inputs supplied instantly, the parallel time complexity is:

$$T(n) = T(n/2) + 1 = \log_2 n$$

8.6 Parallel Architectures for FFT

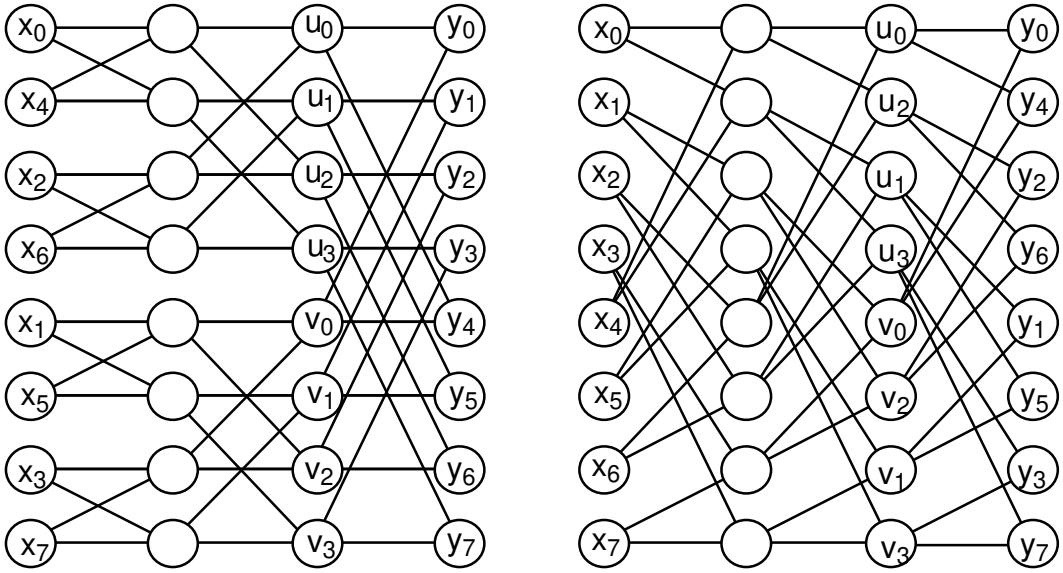


Fig. 8.11. Butterfly network for an 8-point FFT.

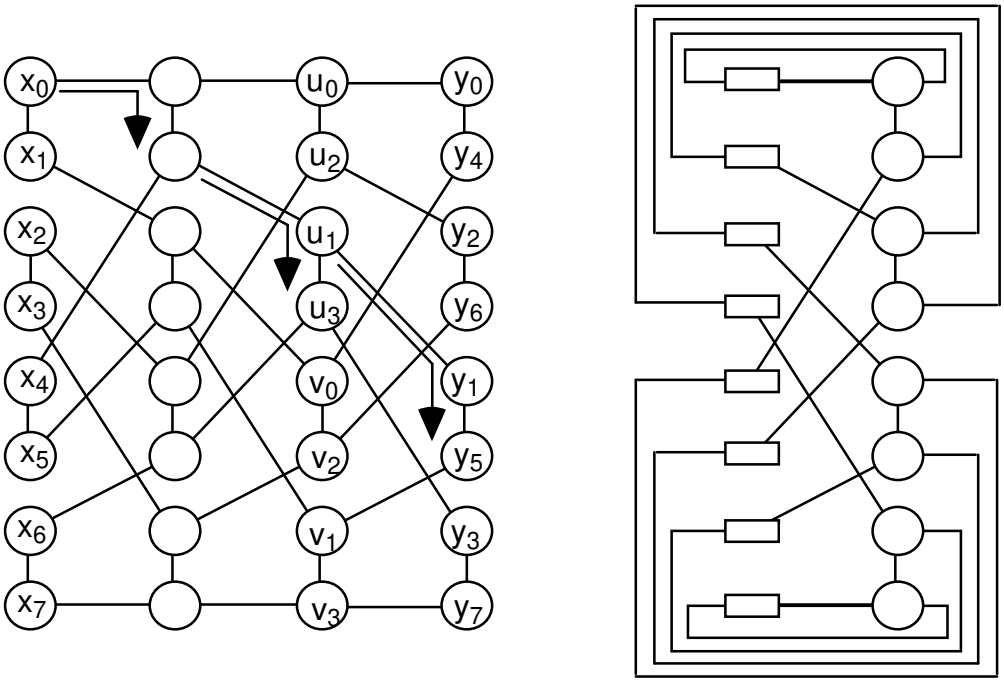
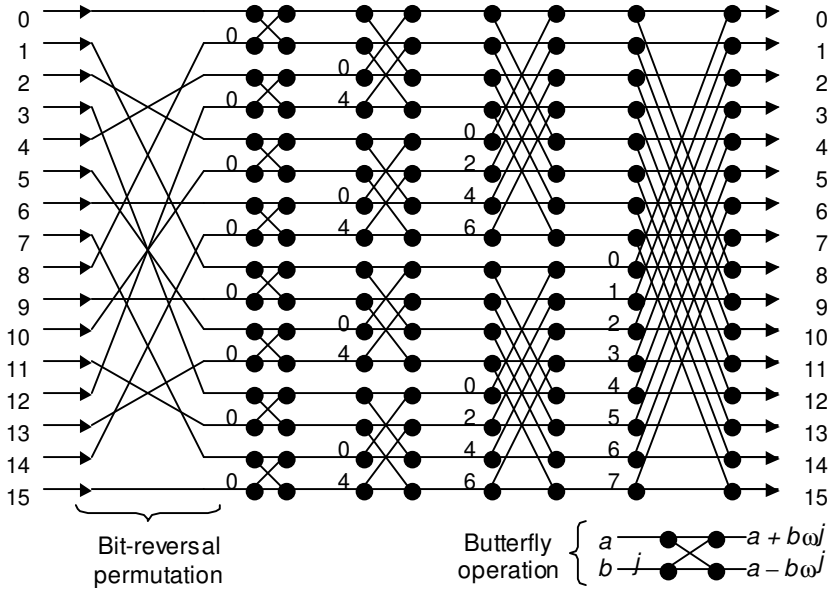


Fig. 8.12. FFT network variant and its shared-hardware realization.



Computation scheme of 16-point FFT.

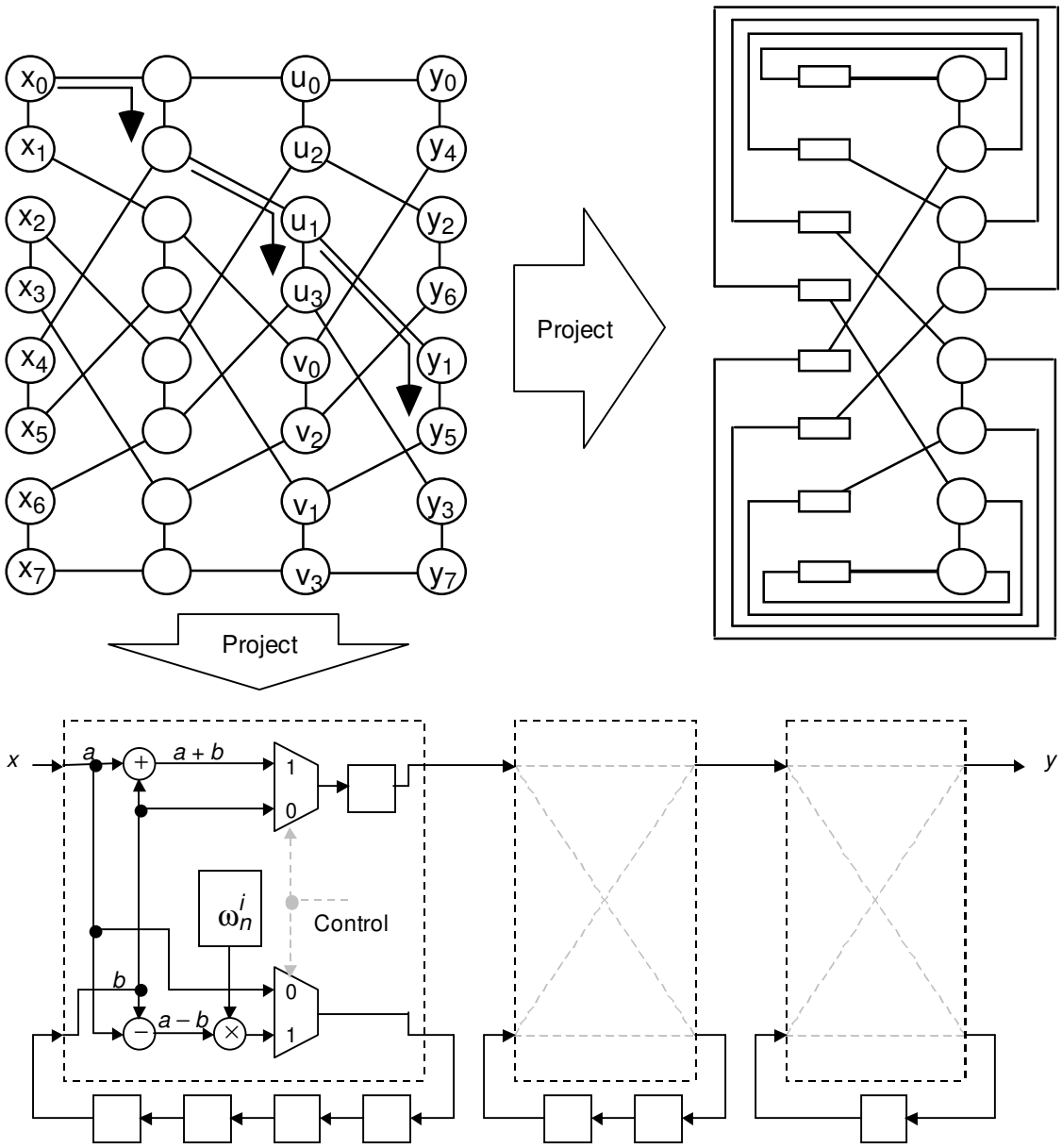


Fig. 8.13. Linear array of $\log_2 n$ cells for n-point FFT computation.

Part III Mesh-Based Architectures

Back to TOC

Part Goals

- Study 2D mesh & torus networks in depth
  - of great practical significance
  - used in recent parallel machines
  - regular with short wires -- scalable
- Briefly review other mesh(like) networks
  - higher-dimensional meshes/tori
  - variants and derivative architectures

Part Contents

- Chapter 9: Sorting on a 2D Mesh or Torus
- Chapter 10: Routing on a 2D Mesh or Torus
- Chapter 11: Numerical 2D Mesh Algorithms
- Chapter 12: Mesh-Related Architectures

9 Sorting on a 2D Mesh or Torus

Back to TOC

Chapter Goals

- Introduce the mesh model (processors, links, communication)
- Develop 2D mesh sorting algorithms
- Learn about mesh strengths/weaknesses in communication-intensive problems

Chapter Contents

- 9.1. Mesh-Connected Computers
- 9.2. The Shearsort Algorithm
- 9.3. Variants of Simple Shearsort
- 9.4. Recursive Sorting Algorithms
- 9.5. A Nontrivial Lower Bound
- 9.6. Achieving the Lower Bound

9.1 Mesh-Connected Computers

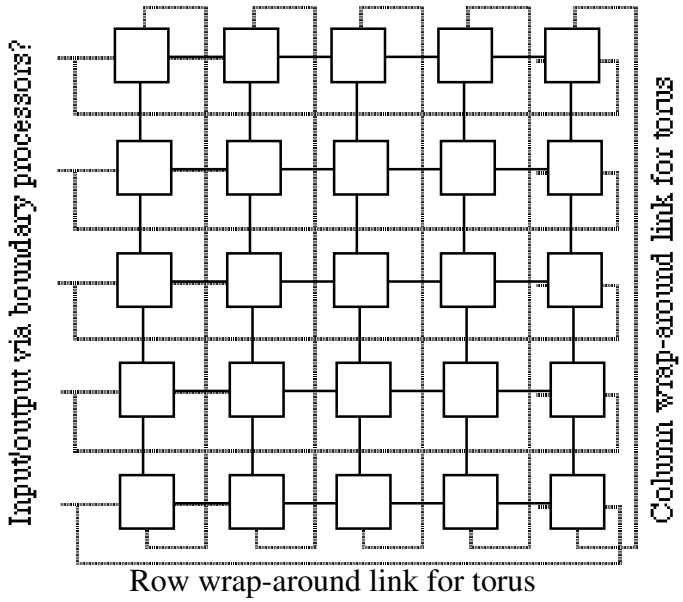


Fig. 9.1. Two-dimensional mesh-connected computer.

We focus on 2D mesh (>2D in Chapter 12)

NEWS or four-neighbor mesh (others in Chapter 12)

Square ($\sqrt{p} \times \sqrt{p}$) or rectangular ($r \times p/r$) mesh

MIMD, SPMD, or SIMD mesh

All-port versus single-port communication

Weak SIMD model: all communications in same direction

Diameter-based and bisection-based lower bounds

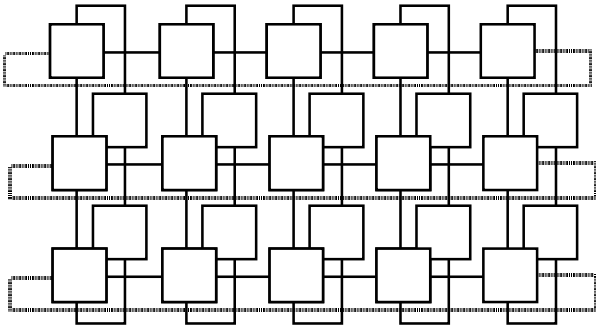


Fig. 9.2. **A 5 × 5 torus folded along its columns. Folding this diagram along the rows will produce a layout with only short links.**

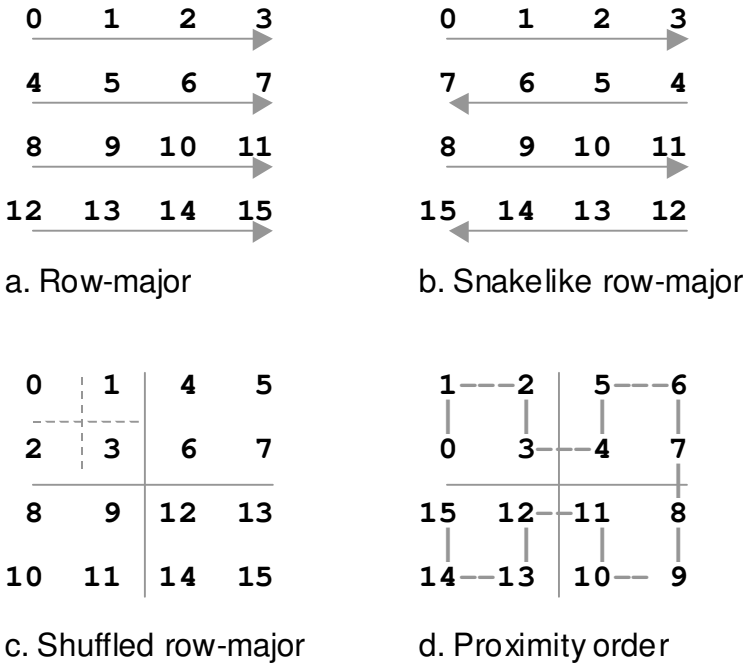


Fig. 9.3. **Some linear indexing schemes for the processors in a 2D mesh.**

Interprocessor communication

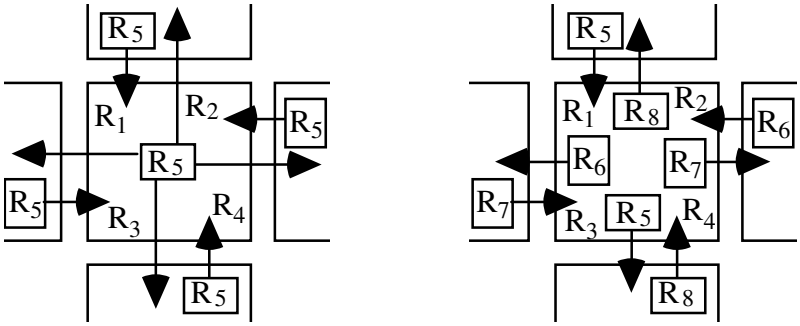
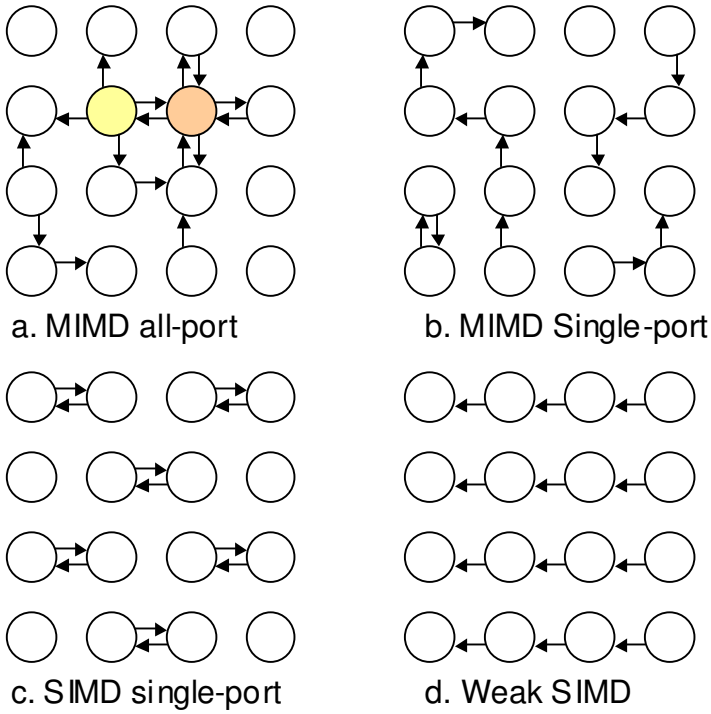


Fig. 9.4. Reading data from NEWS neighbors via virtual local registers.



Some communication modes.

9.2 The Shearsort Algorithm

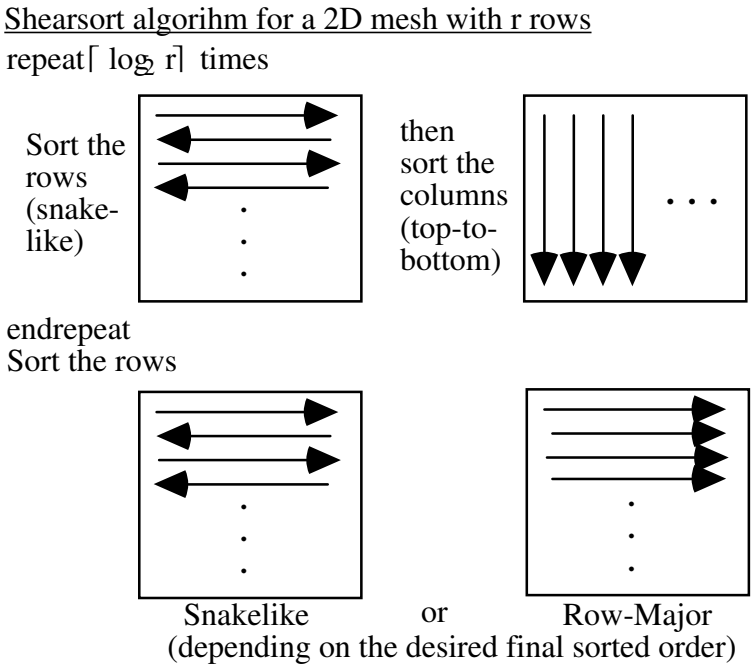


Fig. 9.5. Description of the shearsort algorithm on an *r*-row 2D mesh.

$$T_{\text{shearsort}} = \lceil \log_2 r \rceil (p/r + r) + p/r$$

On a square $\sqrt{p} \times \sqrt{p}$ mesh, $T_{\text{shearsort}} = \sqrt{p}\,(\log_2 p + 1)$

Proof of correctness of shearsort via the 0-1 principle

Assume that in doing the column sorts, we first sort pairs of elements in the column and then sort the entire column

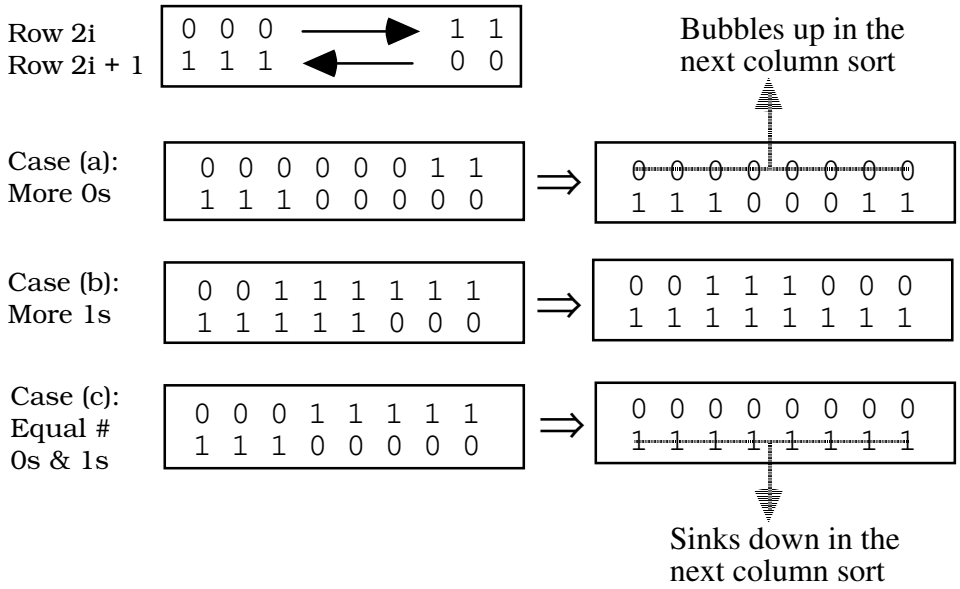


Fig. 9.6. A pair of dirty rows create at least one clean row in each shearsort iteration.

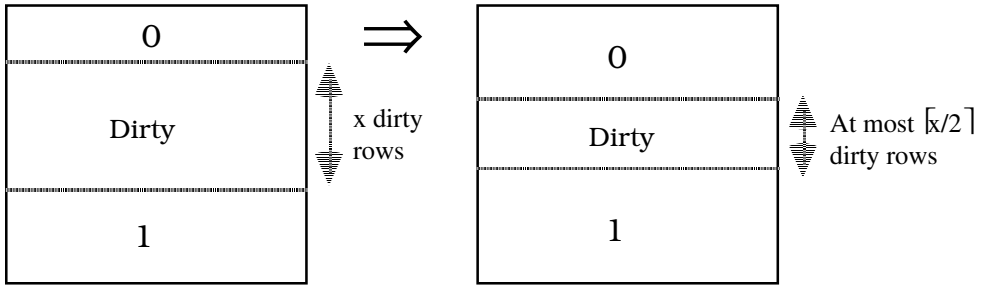


Fig. 9.7. The number of dirty rows halves with each shearsort iteration.

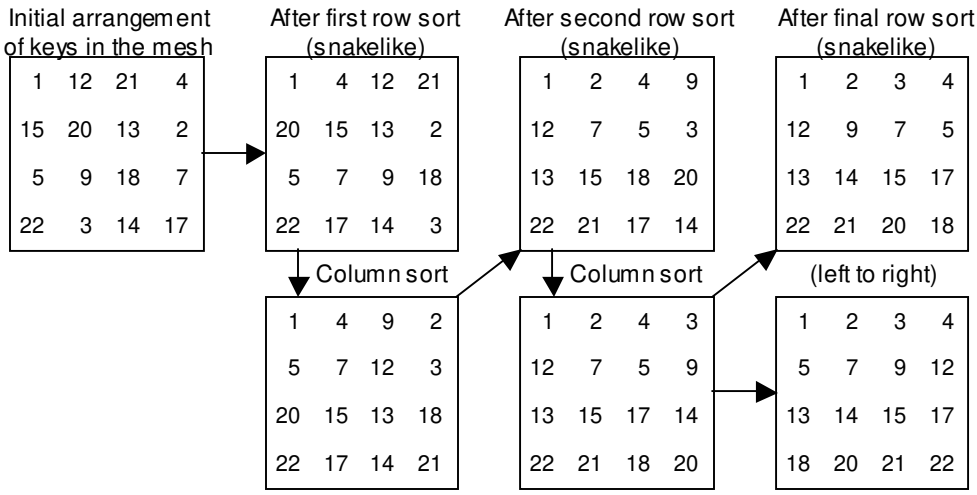


Fig. 9.8. Example of shearsort on a 4×4 mesh.

9.3 Variants of Simple Shearsort

Sorting 0s & 1s on a linear array: odd-even transposition steps can be limited to the number of dirty elements

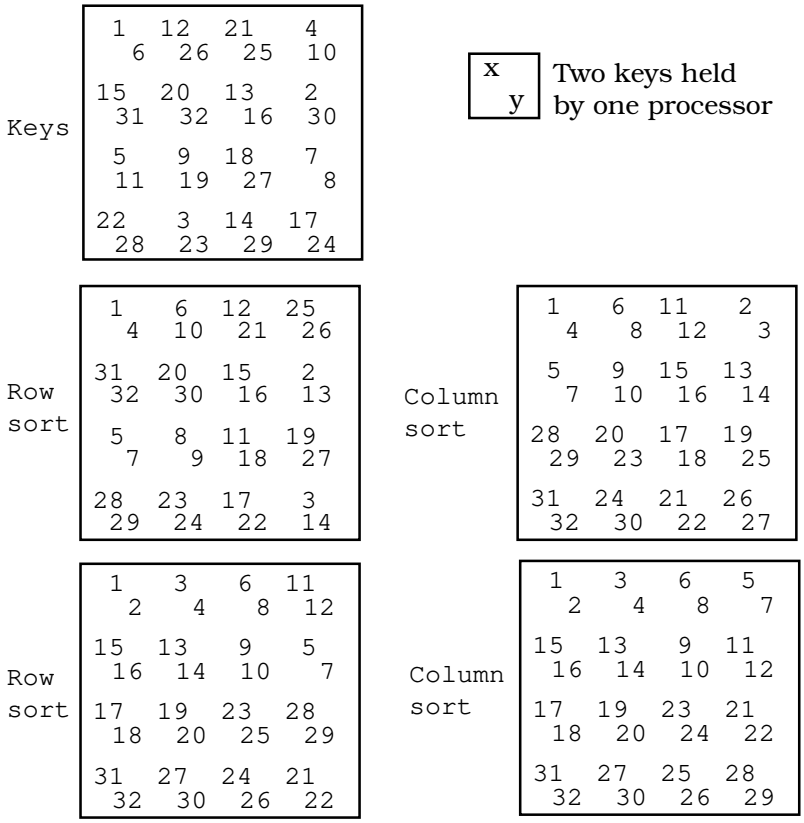
Example: sorting 000001011111 requires at most 2 steps

Thus, we can replace complete column sorts of shearsort with successively fewer odd-even transposition steps

$$T_{\text{opt shearsort}} = (p/r)(\log_2 r + 1) + r + r/2 + \cdots + 2$$

$$= (p/r)(\log_2 r + 1) + 2r - 2$$

$$[r = \sqrt{p} : \sqrt{p}(\tfrac{1}{2}\log_2 p + 3) - 2]$$



The final row sort (snake-like or row-major) is not shown.

Fig. 9.9. Example of shearsort on a 4×4 mesh with two keys stored per processor.

9.4 Recursive Sorting Algorithms

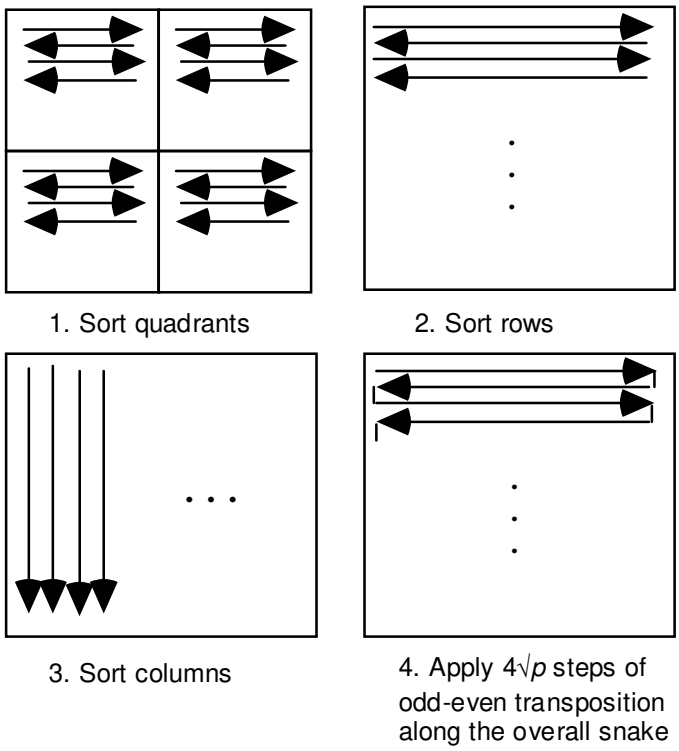


Fig. 9.10. Graphical depiction of the first recursive algorithm for sorting on a 2D mesh based on four-way divide and conquer.

$$T(\sqrt{p}\,) = T(\sqrt{p}\,/2) + 5.5\sqrt{p} \cong 11\sqrt{p}$$

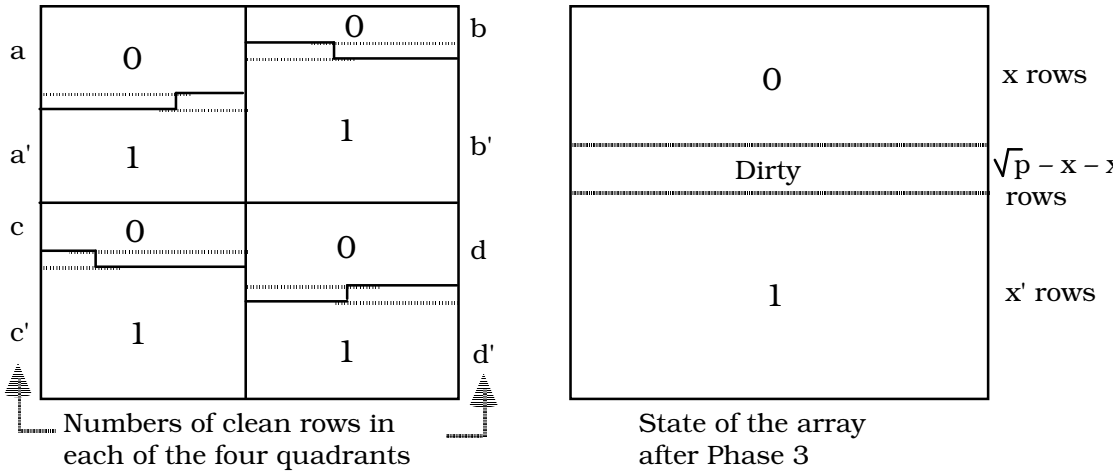


Fig. 9.11. The proof of the first recursive sorting algorithm for 2D meshes.

$$x \geq b + c + \lfloor (a - b)/2 \rfloor + \lfloor (d - c)/2 \rfloor$$

A similar inequality for x' leads to:

$$\begin{aligned} x + x' \geq\ & b + c + \lfloor (a - b)/2 \rfloor + \lfloor (d - c)/2 \rfloor \\ & + a' + d' + \lfloor (b' - a')/2 \rfloor + \lfloor (c' - d')/2 \rfloor \\ \geq\ & b + c + a' + d' + (a - b)/2 + (d - c)/2 \\ & + (b' - a')/2 + (c' - d')/2 - 4 \times 1/2 \\ =\ & (a + a')/2 + (b + b')/2 + (c + c')/2 + (d + d')/2 - 2 \\ \geq\ & \sqrt{p} - 4 \end{aligned}$$

The number of dirty rows after Phase 3: $\sqrt{p} - x - x' \leq 4$

Thus, at most $4\sqrt{p}$ of the p elements are out of order along the overall snake

Another recursive sorting algorithm

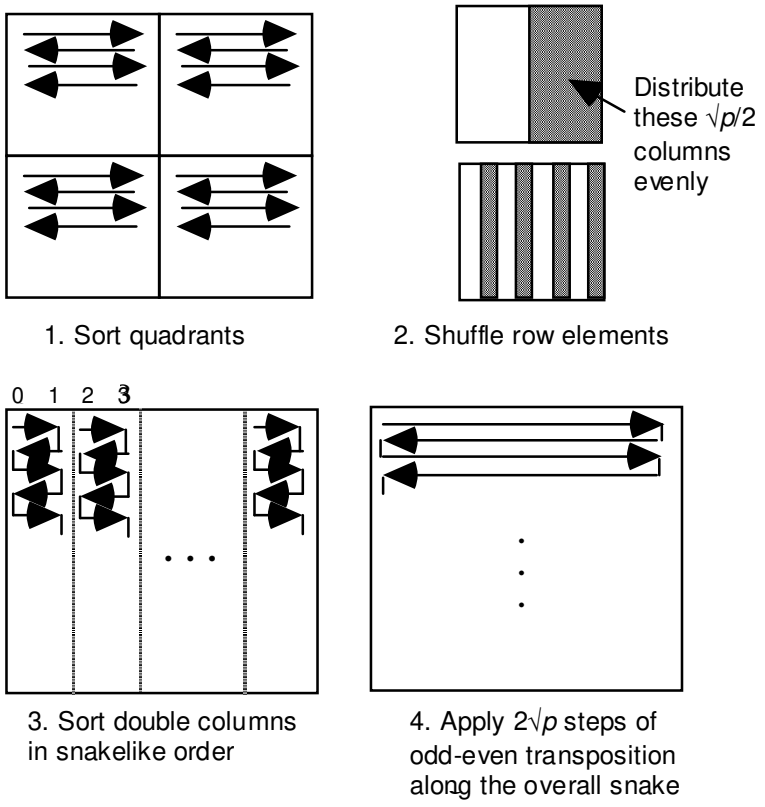


Fig. 9.12. Graphical depiction of the second recursive algorithm for sorting on a 2D mesh based on four-way divide and conquer.

$$T(\sqrt{p}) = T(\sqrt{p}/2) + 4.5\sqrt{p} \cong 9\sqrt{p}$$

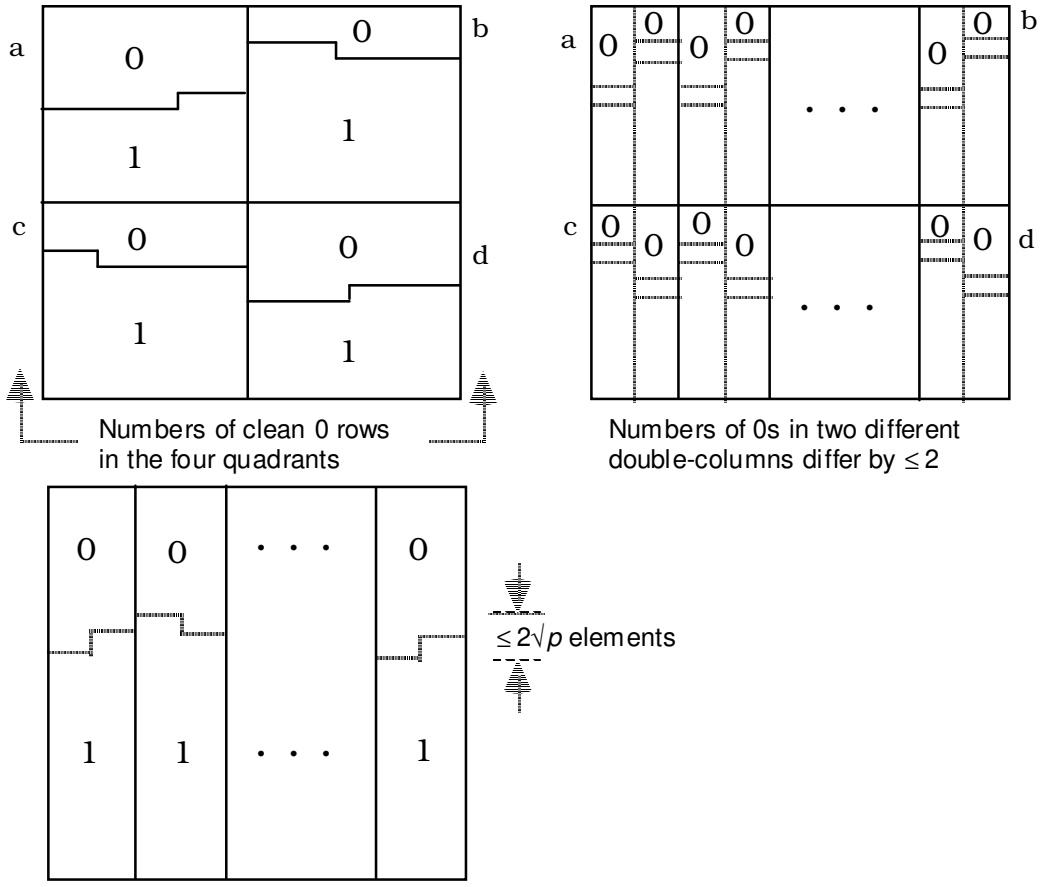


Fig. 9.13. The proof of the second recursive sorting algorithm for 2D meshes.

9.5 A Nontrivial Lower Bound

We now have a $9\sqrt{p}$ -time mesh sorting algorithm

Two questions of interest:

1. Raise the $2\sqrt{p} - 2$ diameter-based lower bound?

   Yes, for snakelike sort, the bound $3\sqrt{p} - o(\sqrt{p})$ can be derived

2. Design an algorithm with better time than $9\sqrt{p}$?

   Yes, the Schnorr-Shamir sorting algorithm requires $3\sqrt{p} + o(\sqrt{p})$ steps

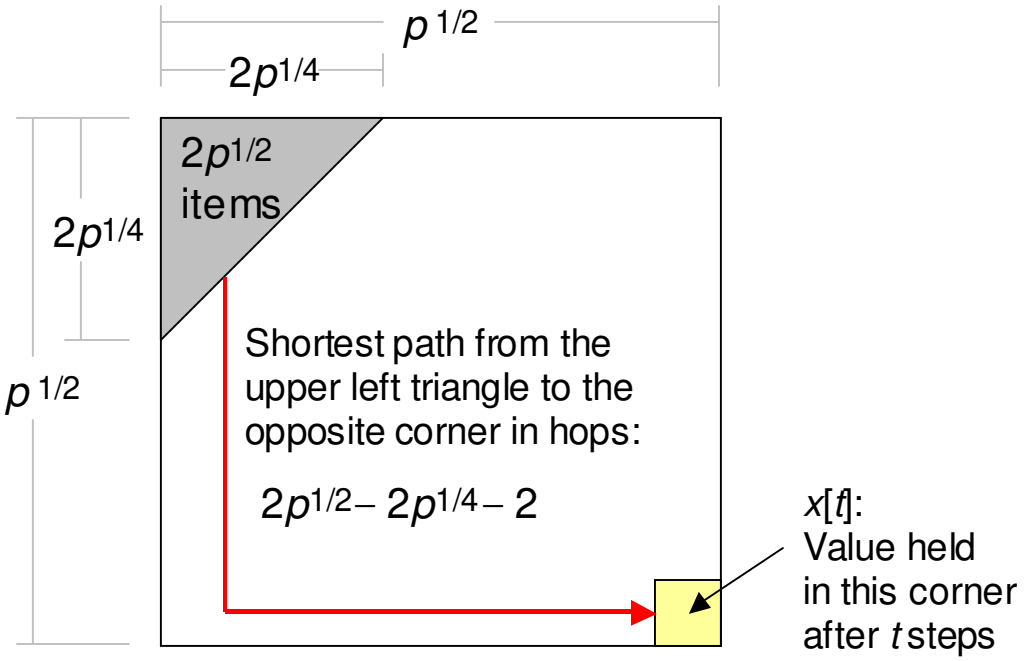


Fig. 9.14. The proof of the $3\sqrt{p} - o(\sqrt{p})$ lower bound for sorting in snakelike row-major order.

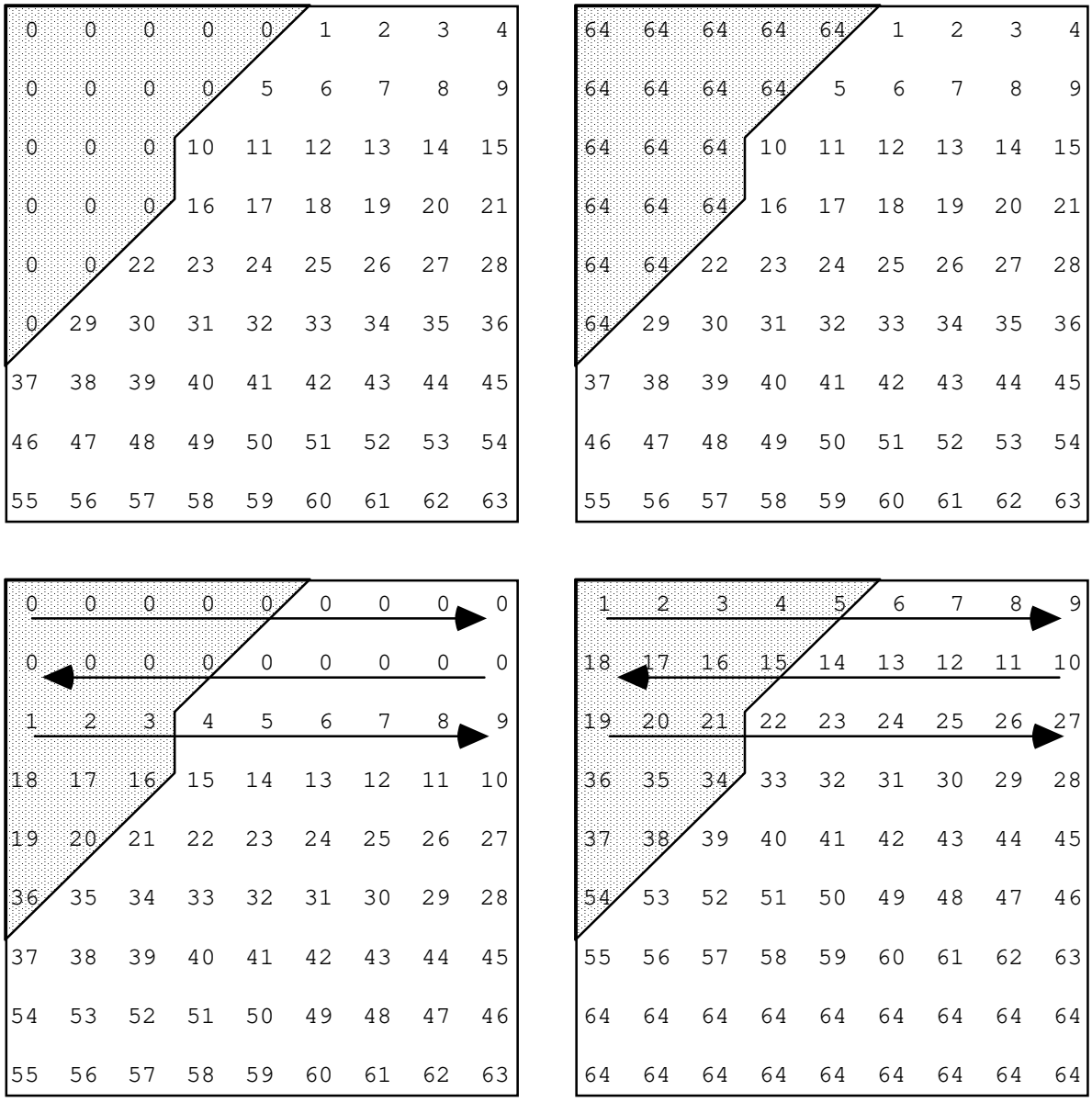


Fig. 9.15. Illustrating the effect of fewer or more 0s in the shaded area.

9.6 Achieving the Lower Bound

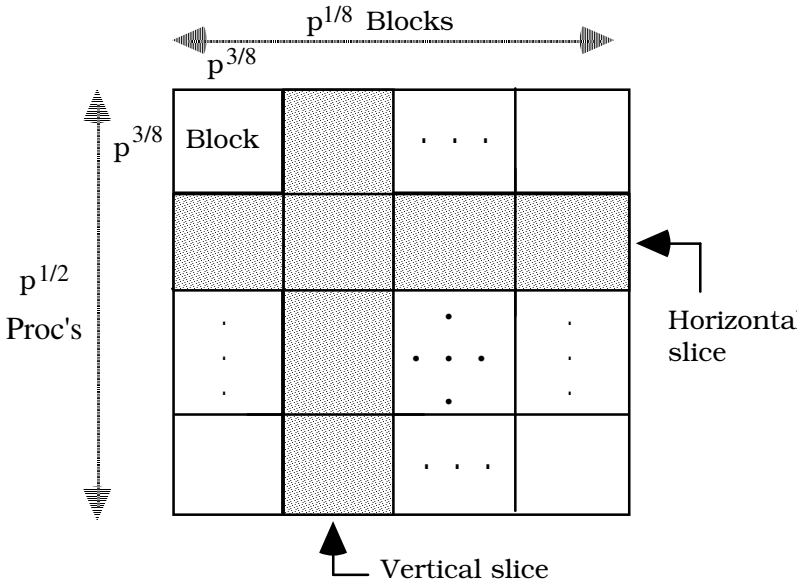


Fig. 9.16. Notation for the asymptotically optimal sorting algorithm.

Schnorr-Shamir algorithm for snakelike sorting on a 2D mesh

1. Sort all blocks in snakelike order, independently & in parallel
2. Permute the columns such that the columns of each vertical slice are evenly distributed among all vertical slices
3. Sort each block in snakelike order
4. Sort the columns independently from top to bottom
5. Sort Blocks 0&1, 2&3, · · · of all vertical slices together in snakelike order; i.e., sort within $2p^{3/8} \times p^{3/8}$ submeshes
6. Sort Blocks 1&2, 3&4, · · · of all vertical slices together in snake-like order; again done within $2p^{3/8} \times p^{3/8}$ submeshes
7. Sort the rows independently in snakelike order
8. Apply $2p^{3/8}$ steps of odd-even transposition to the snake

10 Routing on a 2-D Mesh or Torus

Back to TOC

Chapter Goals

- Learn how to route multiple data items to their respective destinations

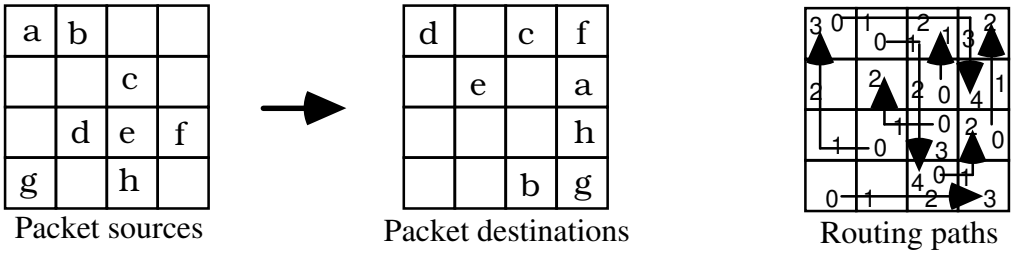
  (in PRAM routing is nonexistent and in the circuit model it is hardwired)
- Become familiar with issues in packet routing and wormhole routing

Chapter Contents

- 10.1. Types of Data Routing Operations
- 10.2. Useful Elementary Operations
- 10.3. Data Routing on a 2D Array
- 10.4. Greedy Routing Algorithms
- 10.5. Other Classes of Routing Algorithms
- 10.6. Wormhole Routing

10.1 Types of Data Routing Operations

One-to-one communication (point-to-point messages)



Collective communication (per the MPI standard)

a. One to many: broadcast, multicast, scatter

b. Many to one: combine, fan-in, gather

c. Many to many: many-to-many m-cast, all-to-all b-cast, scatter-gather (gossiping), total exchange

Some special data routing operations

a. Data compaction or packing

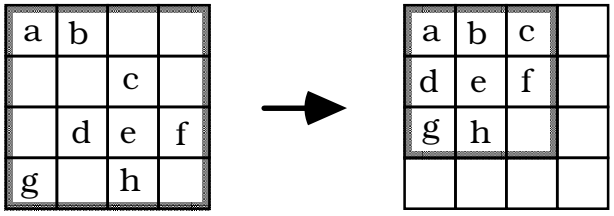


Fig. 10.1. Example of data compaction or packing.

b. Random-access write (RAW): Emulating one memory write step of a PRAM with *p* processors

c. Random-access read (RAR): Emulating one memory read step of a PRAM with *p* processors

10.2 Useful Elementary Operations

Row or column rotation

Sorting records by a key field

Semigroup computation

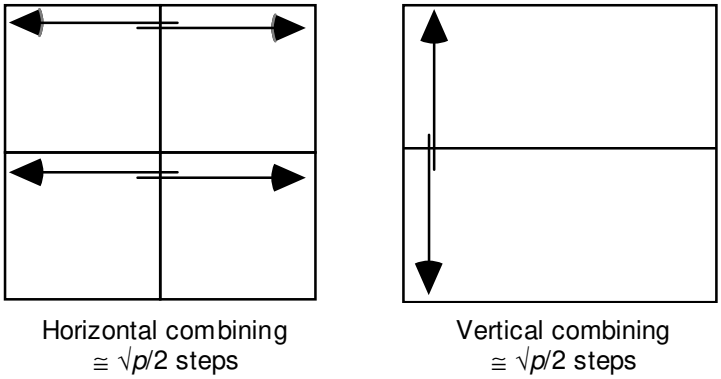


Fig. 10.2. Recursive semigroup computation in a 2D mesh.

Parallel prefix computation

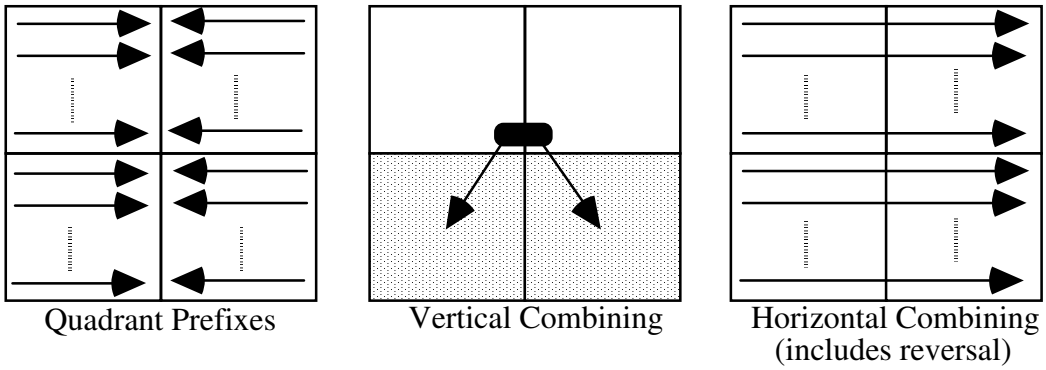


Fig. 10.3. Recursive parallel prefix computation in a 2D mesh.

Routing within a row or column

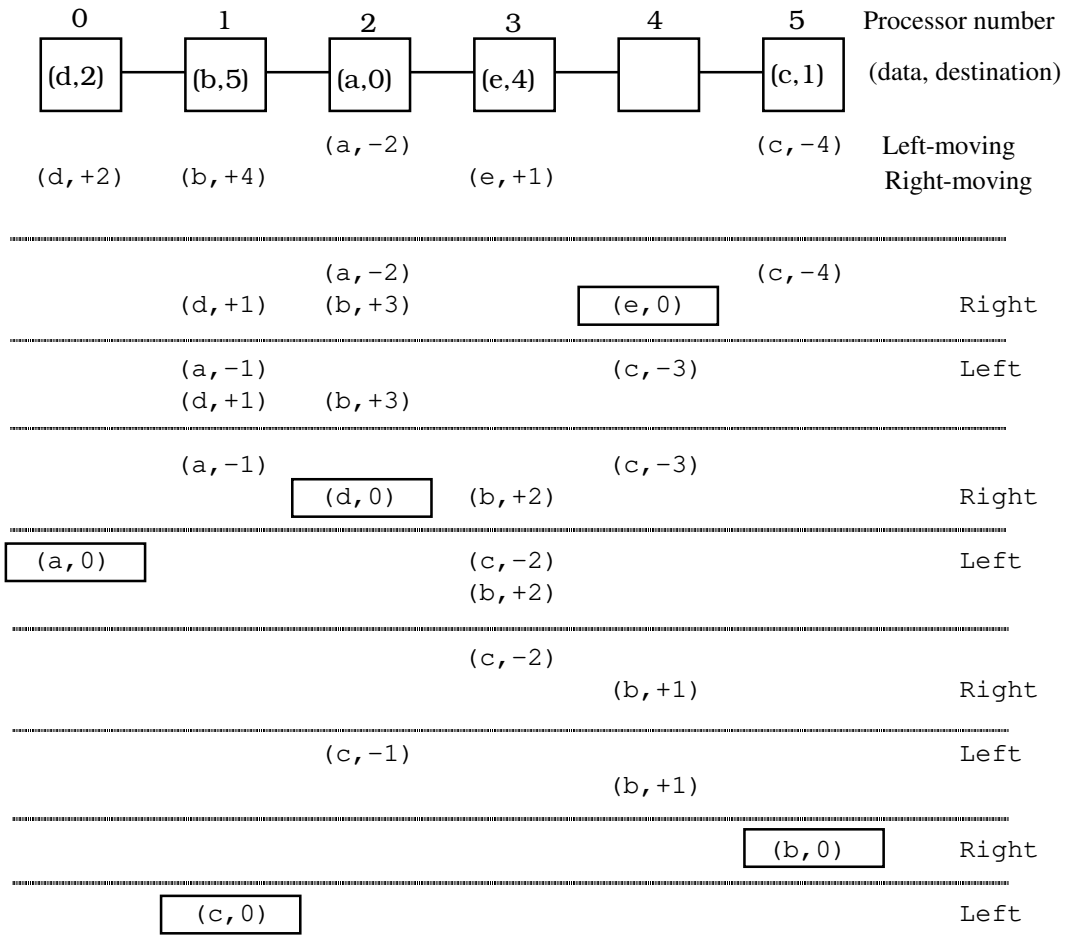


Fig. 10.4. Example of routing multiple packets on a linear array.

10.3 Data Routing on a 2D Array

Exclusive random-access write on a 2D mesh: MeshRAW

1. Sort packets in column-major order by destination column number; break ties by destination row number
2. Shift packets to the right, so that each item is in the correct column. There will be no conflict since at most one element in each row is headed for a given column
3. Route the packets within each column

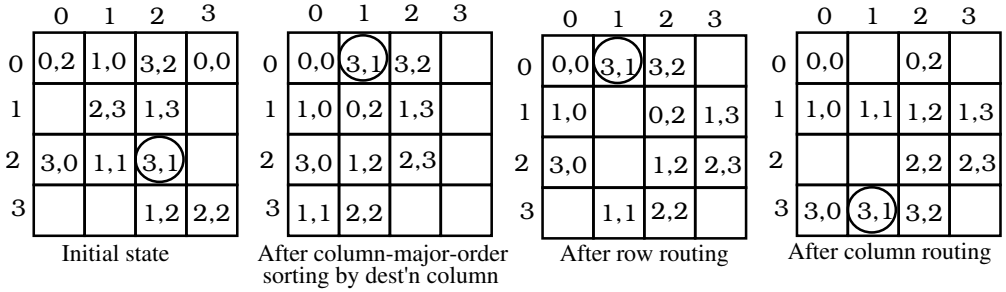


Fig. 10.5. Example of random-access write on a 2D mesh.

Not a shortest-path routing algorithm

e.g., packet headed to (3, 1) first goes to (0, 1)

But fairly efficient

$$\begin{aligned} T &= 3p^{1/2} + o(p^{1/2}) && \{\text{snakelike sorting}\} \\ &+ p^{1/2} && \{\text{column reversal}\} \\ &+ 2p^{1/2} - 2 && \{\text{row \& column routing}\} \\ &= 6p^{1/2} + o(p^{1/2}) \end{aligned}$$

Or $11p^{1/2} + o(p^{1/2})$ with unidirectional communication

10.4 Greedy Routing Algorithms

Greedy: pick a move that causes the most progress toward the destination in each step

Example greedy algorithm: dimension-order (e-cube)

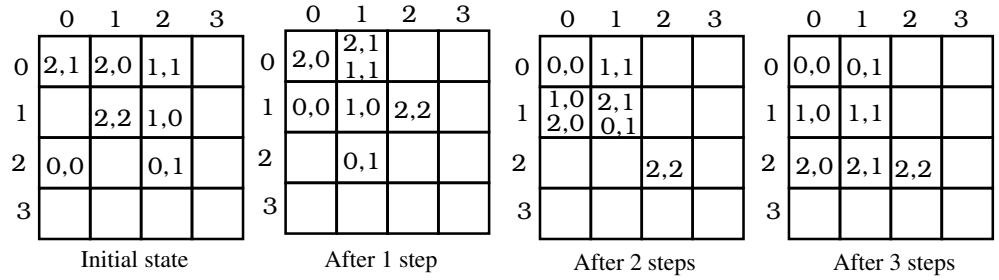


Fig. 10.6. Greedy row-first routing on a 2D mesh.

$T = 2p^{1/2} - 2$ {but requires large buffers}

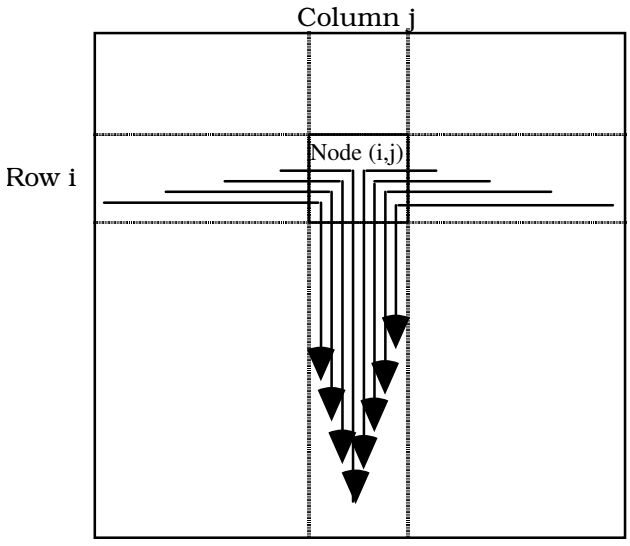


Fig. 10.7. Demonstrating the worst-case buffer requirement with row-first routing.

Routing algorithms thus far

Slow $6p^{1/2}$, but with no conflict (no additional buffer)

Fast $2p^{1/2}$, but with large node buffers

An algorithm that allows trading off time for buffer space

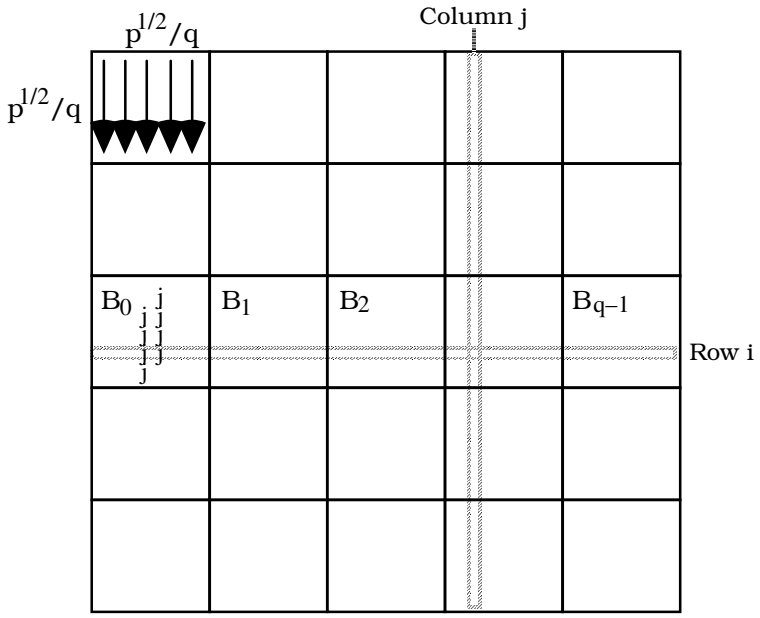


Fig. 10.8. Illustrating the structure of the intermediate routing algorithm.

$$\begin{aligned} T &= 4p^{1/2}/q + o(p^{1/2}/q) && \{\text{column-major block sort}\} \\ &\quad + 2p^{1/2} - 2 && \{\text{route}\} \\ &= (2 + 4/q)p^{1/2} + o(p^{1/2}/q) \end{aligned}$$

Buffer space per node

r_k = number of packets in B_k headed for column j

$$\sum_{k=0}^{q-1} \left\lceil \frac{r_k}{p^{1/2}/q} \right\rceil < \sum_{k=0}^{q-1} \left(1 + \frac{r_k}{p^{1/2}/q}\right) \leq q + (q/p^{1/2}) \sum_{k=0}^{q-1} r_k \leq 2q$$

10.5 Other Classes of Routing Algorithms

Row-first greedy routing has very good average-case performance, even if the node buffer size is restricted

Idea: Convert any routing problem to 2 random instances by picking a random intermediate node for each message

Using combining for concurrent writes:

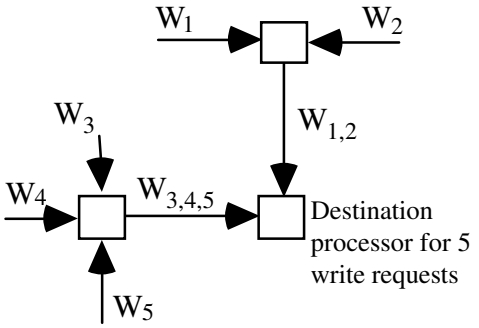


Fig. 10.9. Combining of write requests headed for the same destination.

Terminology for routing problems or algorithms

Static: packets to be routed all available at $t = 0$

Dynamic: packets "born" in course of computation

Off-line: routes precomputed, stored in tables

On-line: routing decisions made on the fly

Oblivious: path depends only on source & dest'n

Adaptive: path may vary by link and node conditions

Deflection: any received packet leaves immediately, even if this means misrouting (via detour path); also known as hot-potato routing

10.6 Wormhole Routing

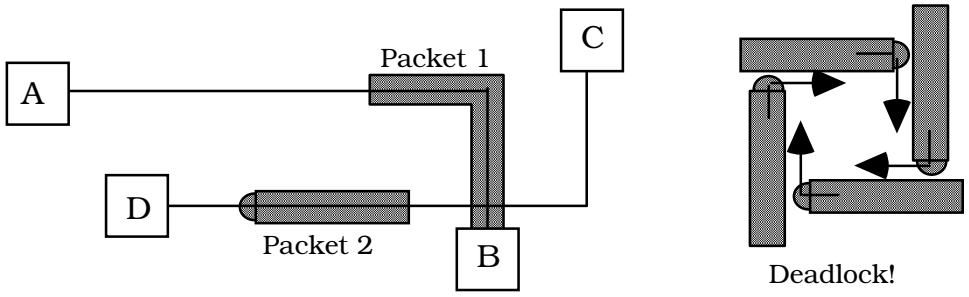


Fig. 10.10. Worms and deadlock in wormhole routing.

Any routing algorithm can be used to choose the path taken by the worm, but practical choices limited by the need for a quick decision

Example: row-first routing, with 2-byte header for row & column displacements

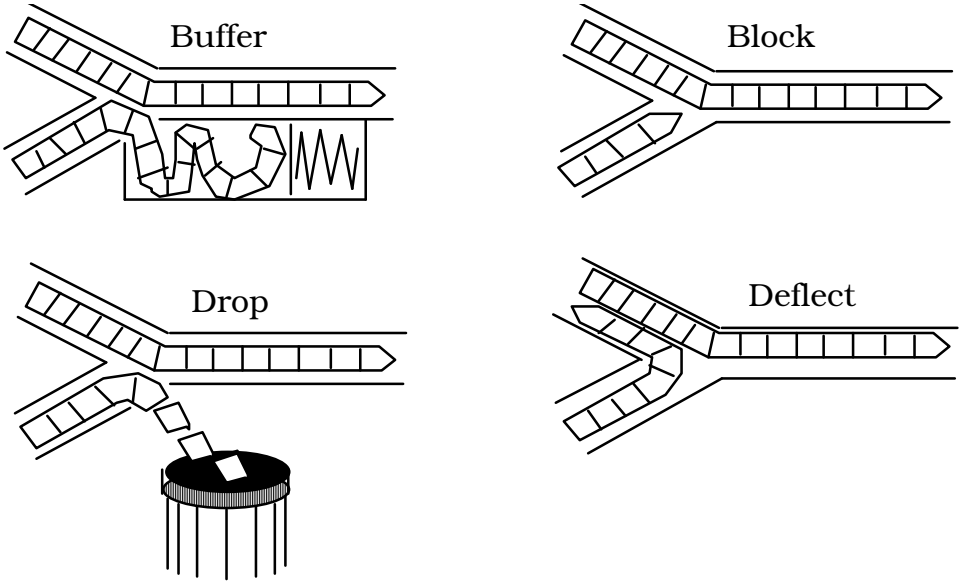
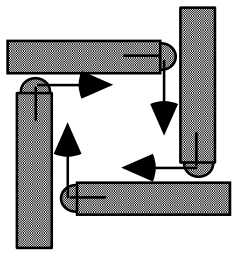


Fig. 10.11. Various ways of dealing with conflicts in wormhole routing.

The deadlock problem in wormhole routing

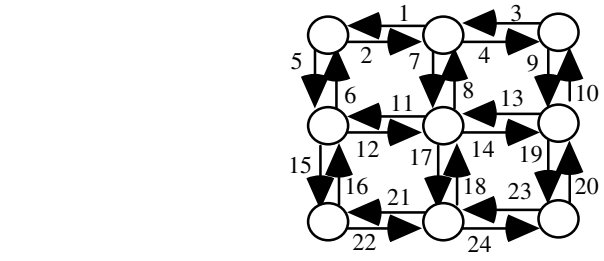


Deadlock!

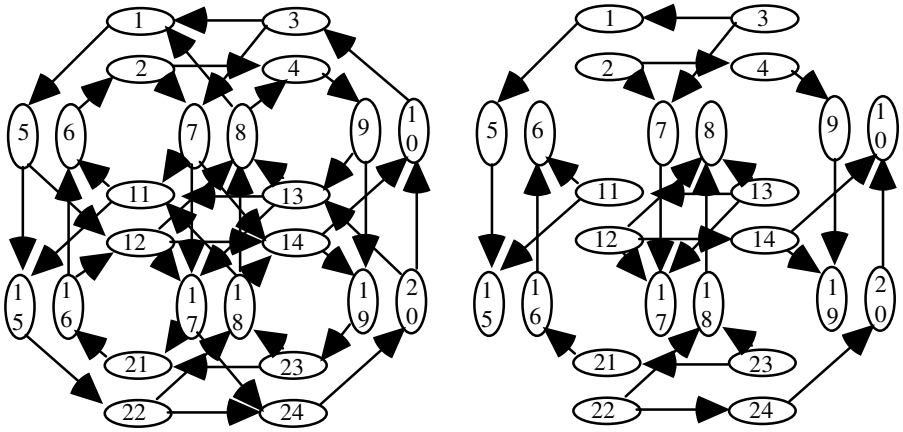
Two strategies for dealing with deadlocks:

(1) Avoidance (2) Detection and recovery

Checking for deadlock potential via link dependence graph; existence of cycles may lead to deadlock



3-by-3 mesh with its links numbered



Unrestricted routing
(following shortest path)

E-cube routing
(row-first)

Fig. 10.12. Use of dependence graph to check for the possibility of deadlock.

Using virtual channels

Several virtual channels time-share one physical channel

Virtual channels serviced in round-robin fashion

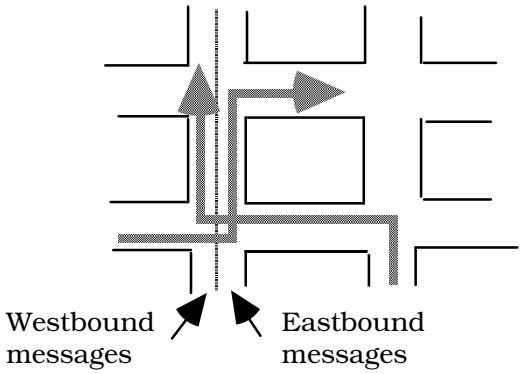


Fig. 10.13. Use of virtual channels for avoiding deadlocks.

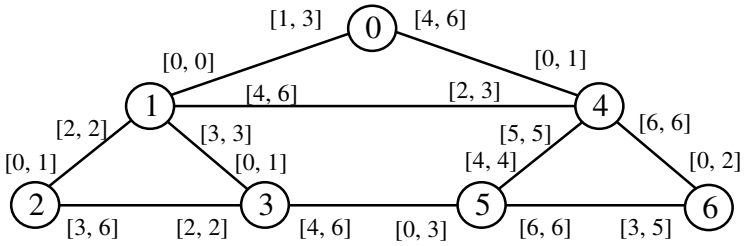


Figure for Problem 10.14.

11 Numerical 2D Mesh Algorithms

Back to TOC

Chapter Goals

- Deal with a sample of numerical and seminumerical algorithms for meshes
- Introduce additional techniques for the design of mesh algorithms

Chapter Contents

- 11.1. Matrix Multiplication
- 11.2. Triangular System of Equations
- 11.3. Tridiagonal System of Equations
- 11.4. Arbitrary System of Linear Equations
- 11.5. Graph Algorithms
- 11.6. Image-Processing Algorithms

11.1 Matrix Multiplication

Matrix-vector multiplication $y_i = \sum_{j=0}^{m-1} a_{ij}x_j$

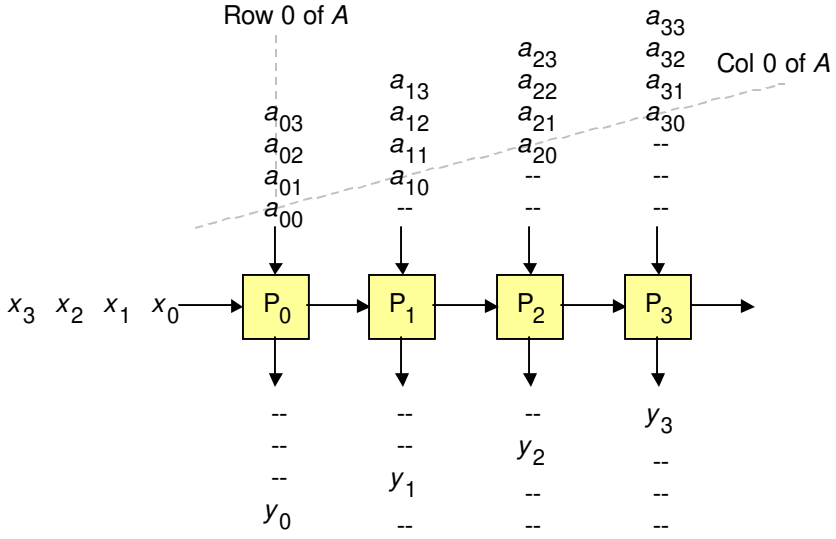
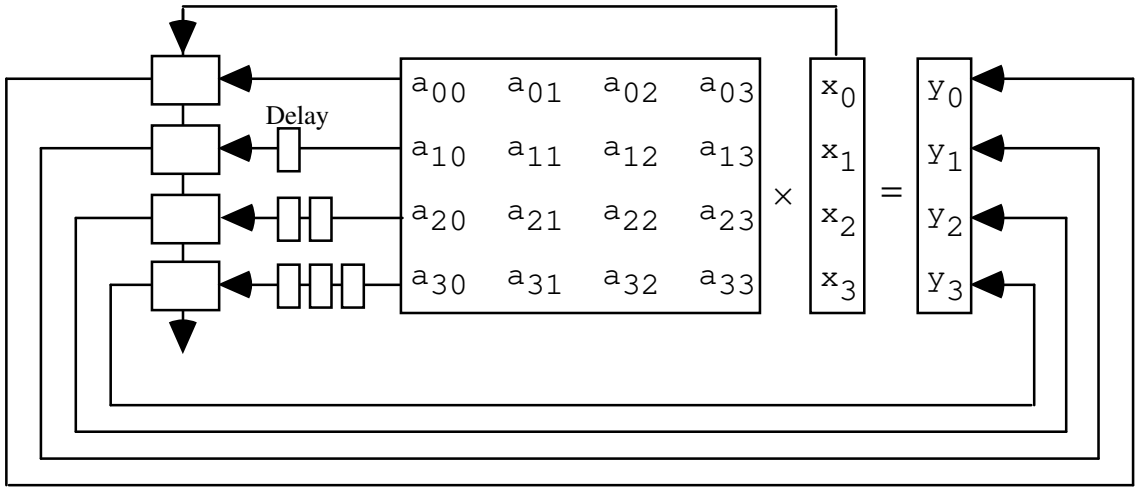


Fig. 11.1. Matrix–vector multiplication on a linear array.



With $p = m$ processors, $T = 2m - 1 = 2p - 1$

Matrix-matrix multiplication $C_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj}$

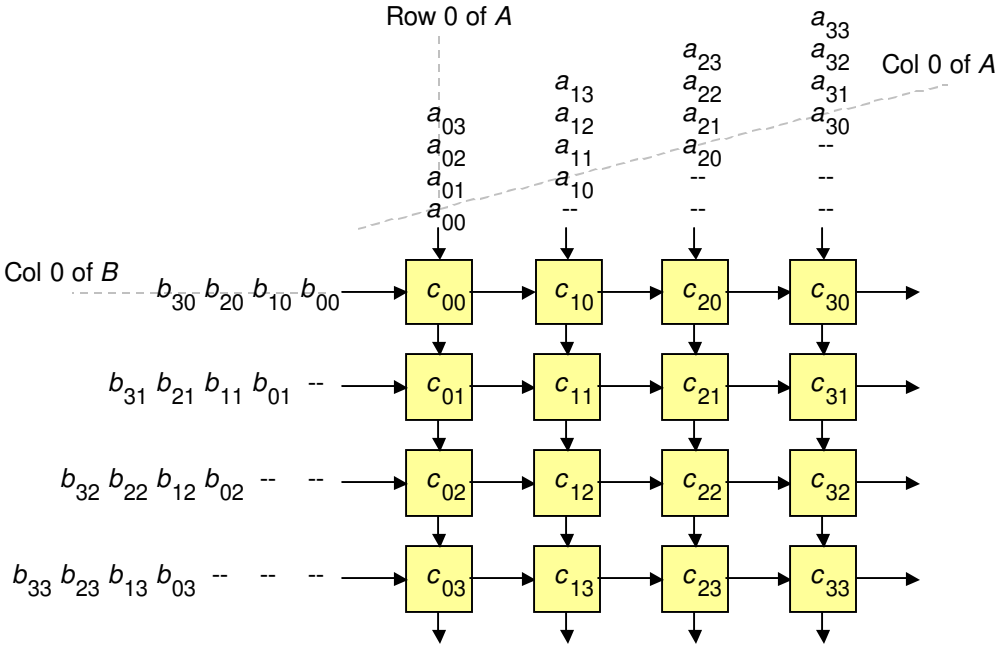


Fig. 11.2. Matrix–matrix multiplication on a 2D mesh.

With $p = m^2$ processors, $T = 3m - 2 = 3\sqrt{p} - 2$

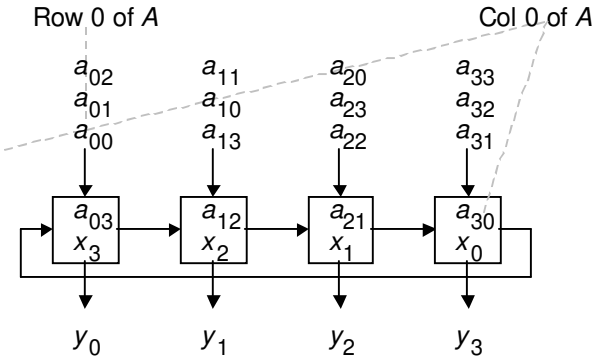


Fig. 11.3. Matrix-vector multiplication on a ring.

With $p = m$ processors, $T = m = p$

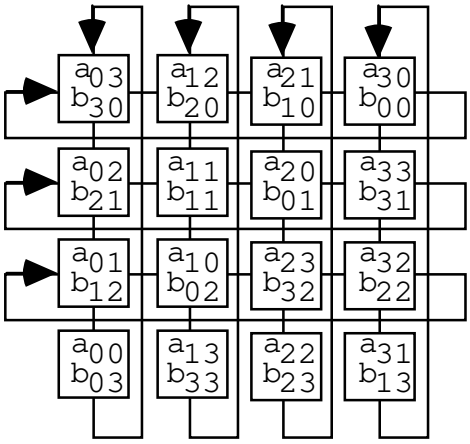


Fig. 11.4. Matrix-matrix multiplication on a torus.

With $p = m^2$ processors, $T = m = \sqrt{p}$

For $m > \sqrt{p}$, use block matrix multiplication

communication can be overlapped with computation

11.2 Triangular System of Equations

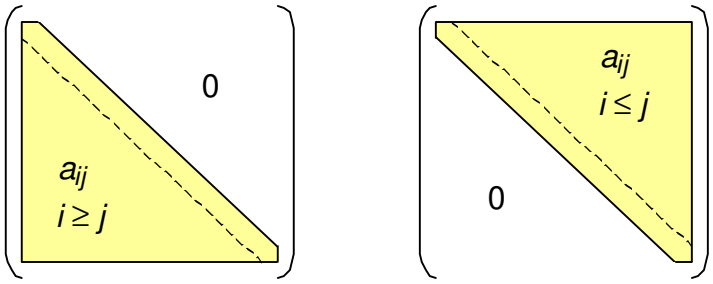


Fig. 11.5. Lower/upper triangular square matrix; if $a_{ii} = 0$ for all i, then it is strictly lower/upper triangular.

$$
\begin{aligned}
a_{00}x_0 &= b_0 \\
a_{10}x_0 + a_{11}x_1 &= b_1 \\
a_{20}x_0 + a_{21}x_1 + a_{22}x_2 &= b_2 \\
&\vdots \\
a_{m-1,0}x_0 + a_{m-1,1}x_1 + \ \ldots \ + a_{m-1,m-1}x_{m-1} &= b_{m-1}
\end{aligned}
$$

Forward substitution (lower triangular)

Back substitution (upper triangular)

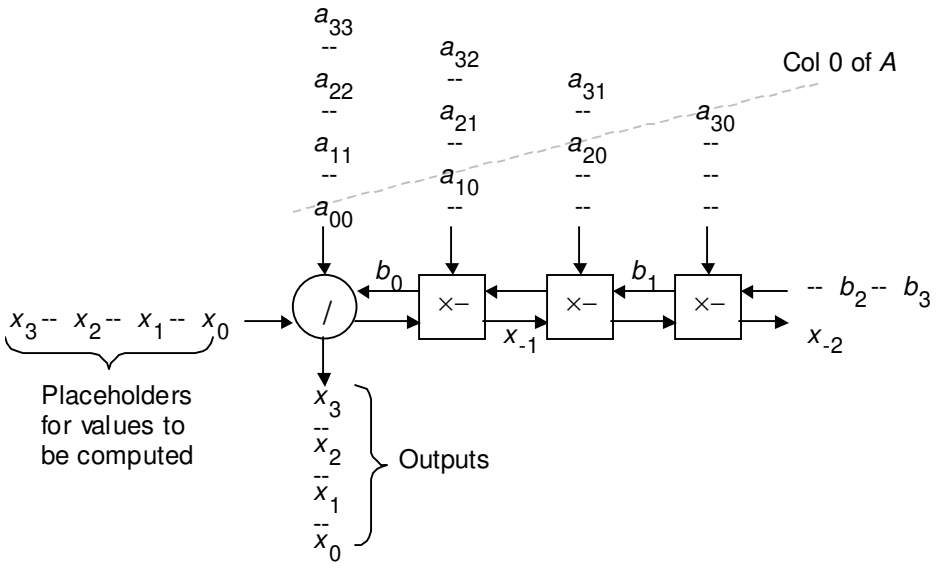


Fig. 11.6. Solving a triangular system of linear equations on a linear array.

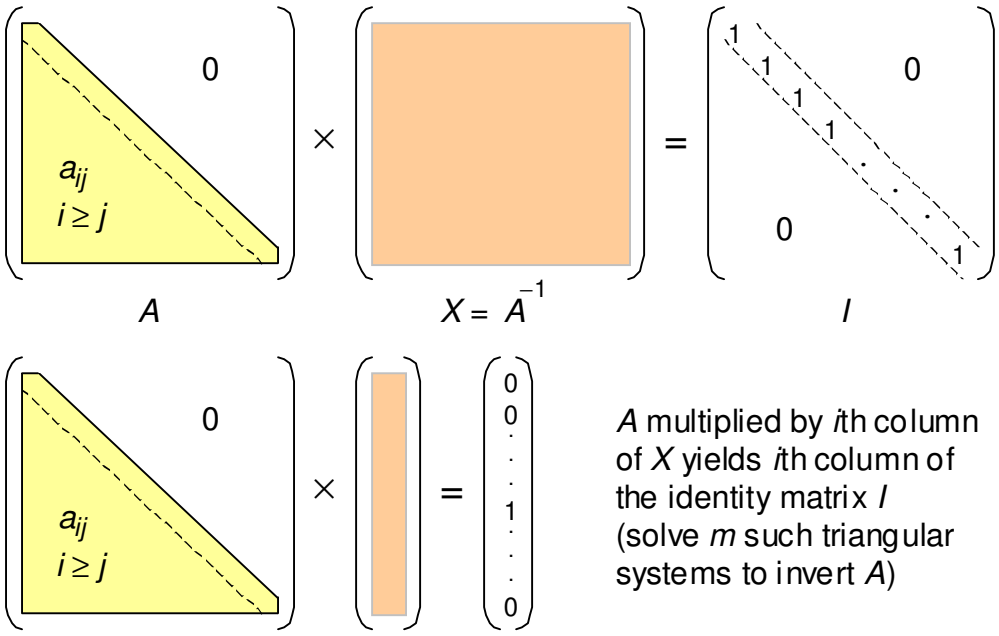


Fig. 11.7. Inverting a triangular matrix by solving triangular systems of linear equations.

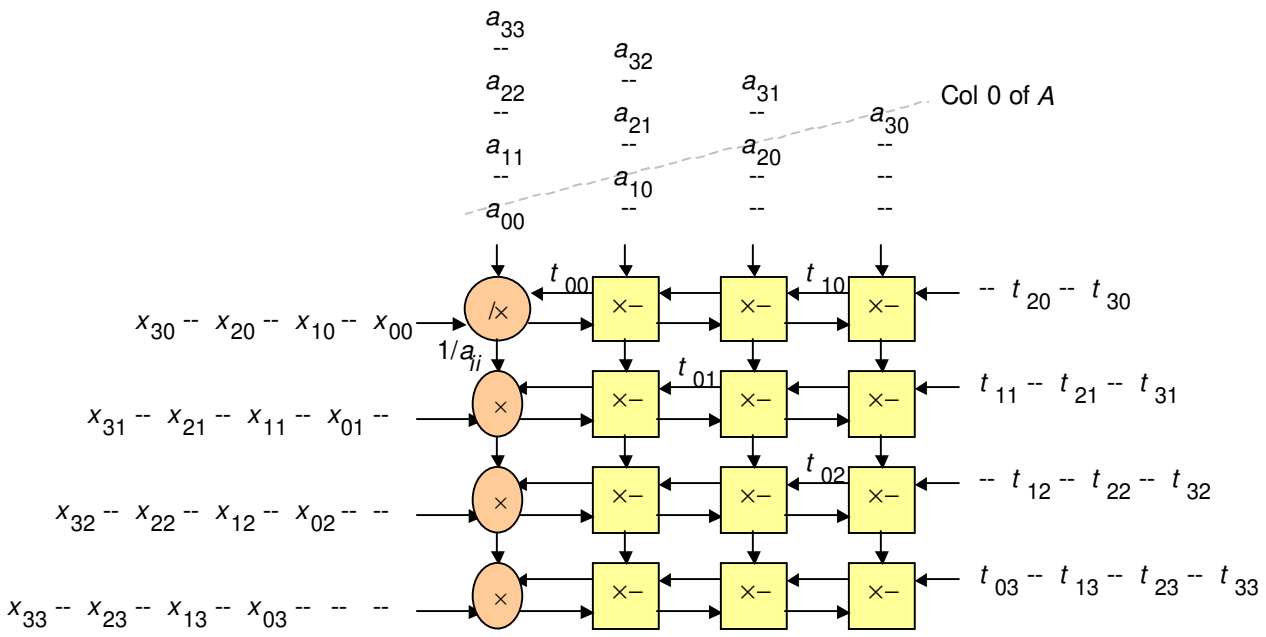


Fig. 11.8. Inverting a lower triangular matrix on a 2D mesh.

11.3 Tridiagonal System of Linear Equations

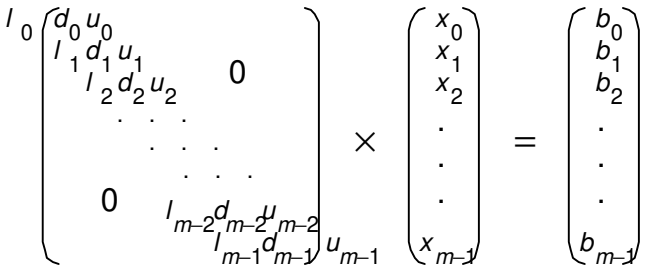


Fig. 11.9. A tridiagonal system of linear equations.

$$
\begin{aligned}
l_0\, x_{-1} &+ d_0\, x_0 &+ u_0\, x_1 &= b_0\\
l_1\, x_0 &+ d_1\, x_1 &+ u_1\, x_2 &= b_1\\
l_2\, x_1 &+ d_2\, x_2 &+ u_2\, x_3 &= b_2\\
&\vdots\\
l_{m-1} x_{m-2} &+ d_{m-1}\, x_{m-1} &+ u_{m-1}\, x_m &= b_{m-1}
\end{aligned}
$$

Tridiagonal, pentadiagonal, matrices arise in the solution of differential equations using finite difference methods

Odd-even reduction: the ith equation can be rewritten as:

$$x_i = (1/d_i)\,(b_i - l_i\,x_{i-1} - u_i\,x_{ii+1})$$

Take the x_i equations for odd i and plug into even-indexed equations (the ones with even subscripts for l, d, u, b)

We get for each even i ($0 \le i < m$) an equation of the form:

$$-\frac{l_{i-1}l_i}{d_{i-1}}\,x_{i-2} + (d_i - \frac{l_iu_{i-1}}{d_{i-1}} - \frac{u_il_{i+1}}{d_{i+1}})x_i - \frac{u_iu_{i+1}}{d_{i+1}}\,x_{i+2} = b_i - \frac{l_ib_{i-1}}{d_{i-1}} - \frac{u_ib_{i+1}}{d_{i+1}}$$

Each new equation needs 6 multiplies, 6 divides, 4 adds

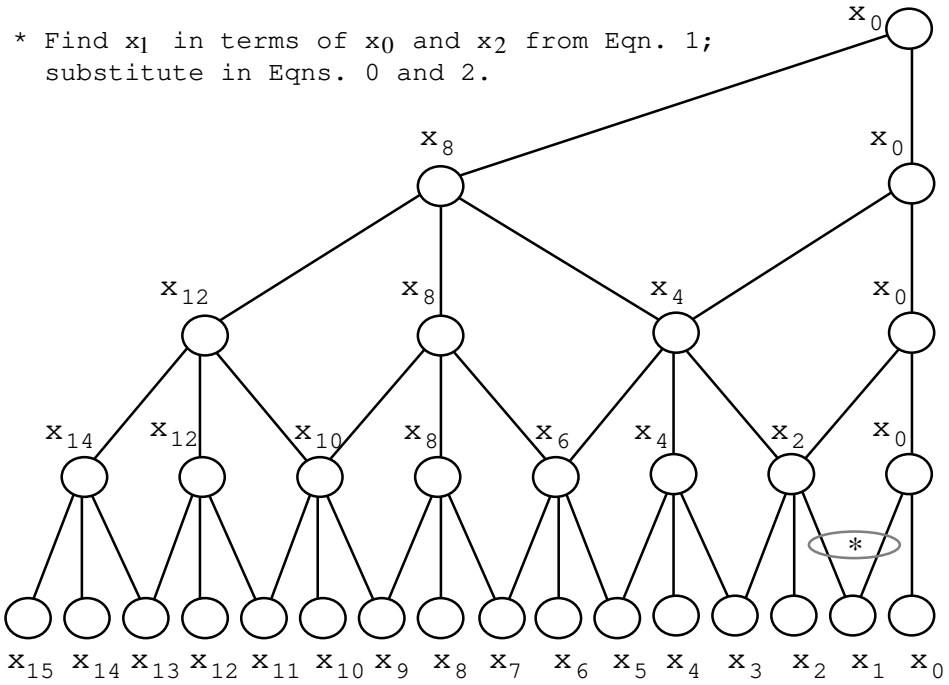


Fig. 11.10. The structure of odd-even reduction for solving a tridiagonal system of equations.

Assuming unit-time arithmetic operations and $p = m$

$$T(m) = T(m/2) + 8 \cong 8 \log_2 m$$

The 6 divides can be replaced with 1 reciprocation per equation, to find $1/d_j$ for each odd j, plus 6 multiplies

We have ignored interprocessor communication time. The analysis is thus valid only for PRAM or for an architecture whose topology matches the structure of Fig. 11.10.

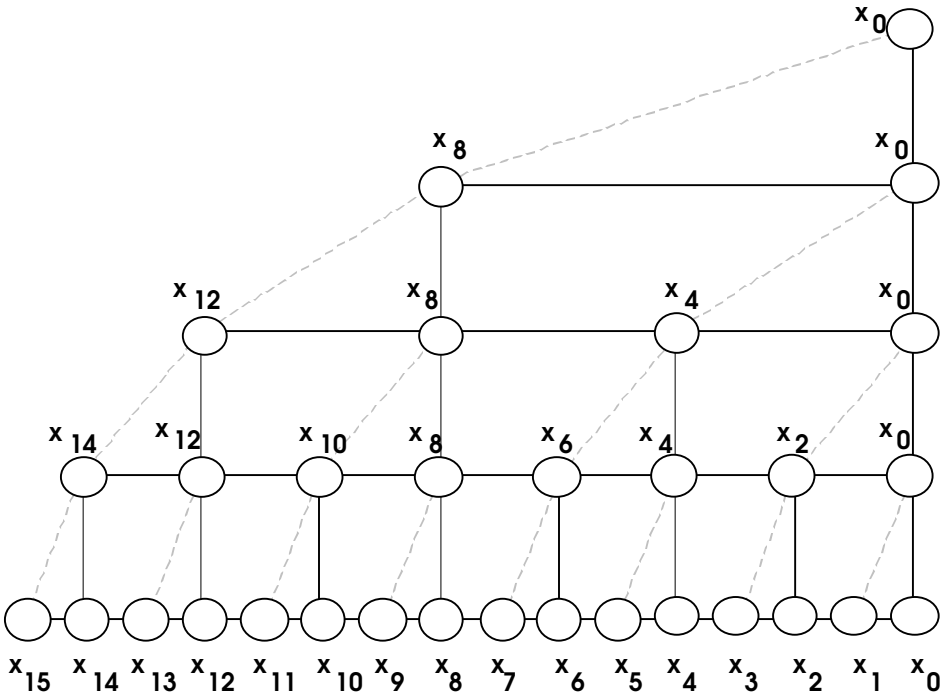
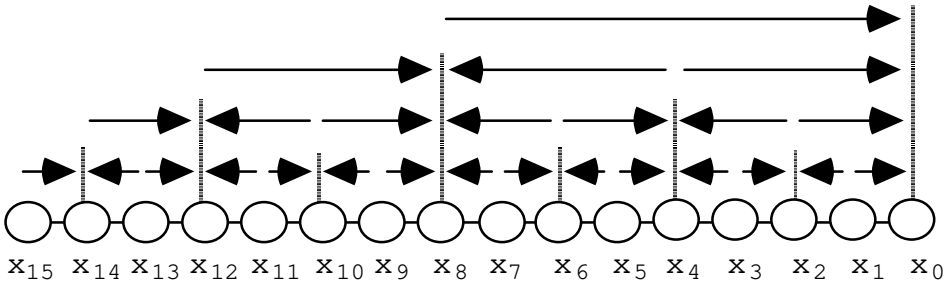


Fig. 11.11. Binary X-tree (with dotted links) and multigrid architectures.

Odd-even reduction on a linear array of $p = m$ processors



Communication time = $2(1 + 2 + 4 + \ldots + m/2) = 2m - 2$

Sequential complexity of odd-even reduction is also $O(m)$

On an m-processor 2D mesh, odd-even reduction can be easily organized to require $\Theta(\sqrt{m})$ time

11.4 Arbitrary System of Linear Equations

Gaussian elimination

$$\begin{aligned} 2x_0 + 4x_1 - 7x_2 &= 3 \\ 3x_0 + 6x_1 - 10x_2 &= 4 \\ -x_0 + 3x_1 - 4x_2 &= 6 \end{aligned} \qquad \begin{aligned} 2x_0 + 4x_1 - 7x_2 &= 7 \\ 3x_0 + 6x_1 - 10x_2 &= 8 \\ -x_0 + 3x_1 - 4x_2 &= -1 \end{aligned}$$

The extended A' matrix for these $k = 2$ sets of equations in $m = 3$ unknowns has $m + k = 5$ columns:

$$A' = \begin{bmatrix} 2 & 4 & -7 & 3 & 7 \\ 3 & 6 & -10 & 4 & 8 \\ -1 & 3 & -4 & 6 & -1 \end{bmatrix}$$

Divide row 0 by 2; add –3 times row 0 to row 1 and add 1 times row 0 to row 2:

$$A'^{(0)} = \begin{bmatrix} 1 & 2 & -7/2 & 3/2 & 7/2 \\ 0 & 0 & 1/2 & -1/2 & -5/2 \\ 0 & 5 & -15/2 & 15/2 & 5/2 \end{bmatrix}$$

$$A''^{(0)} = \begin{bmatrix} 1 & 2 & -7/2 & 3/2 & 7/2 \\ 0 & 5 & -15/2 & 15/2 & 5/2 \\ 0 & 0 & 1/2 & -1/2 & -5/2 \end{bmatrix}$$

$$A'^{(1)} = \begin{bmatrix} 1 & 0 & -1/2 & -3/2 & 5/2 \\ 0 & 1 & -3/2 & 3/2 & 1/2 \\ 0 & 0 & 1/2 & -1/2 & -5/2 \end{bmatrix}$$

$$A'^{(2)} = \begin{bmatrix} 1 & 0 & 0 & -2 & 0 \\ 0 & 1 & 0 & 0 & -7 \\ 0 & 0 & 1 & -1 & -5 \end{bmatrix}$$

Solutions are read out from the last column of $A'^{(2)}$

Gaussian elimination on a 2D array

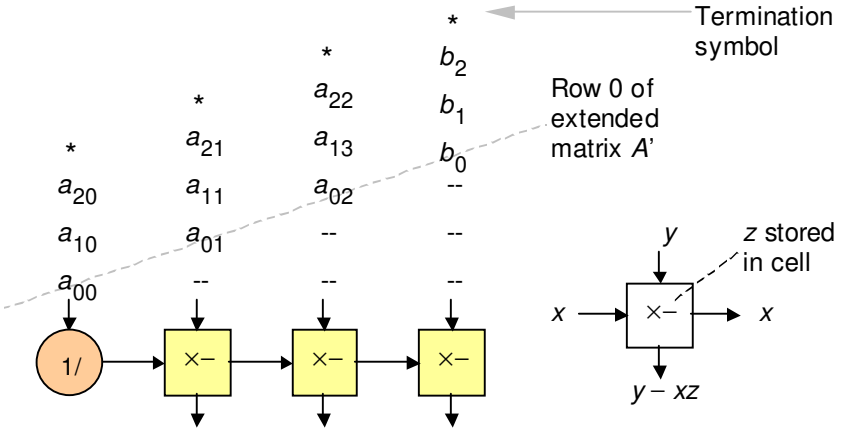


Fig. 11.12. A linear array performing the first phase of Gaussian elimination.

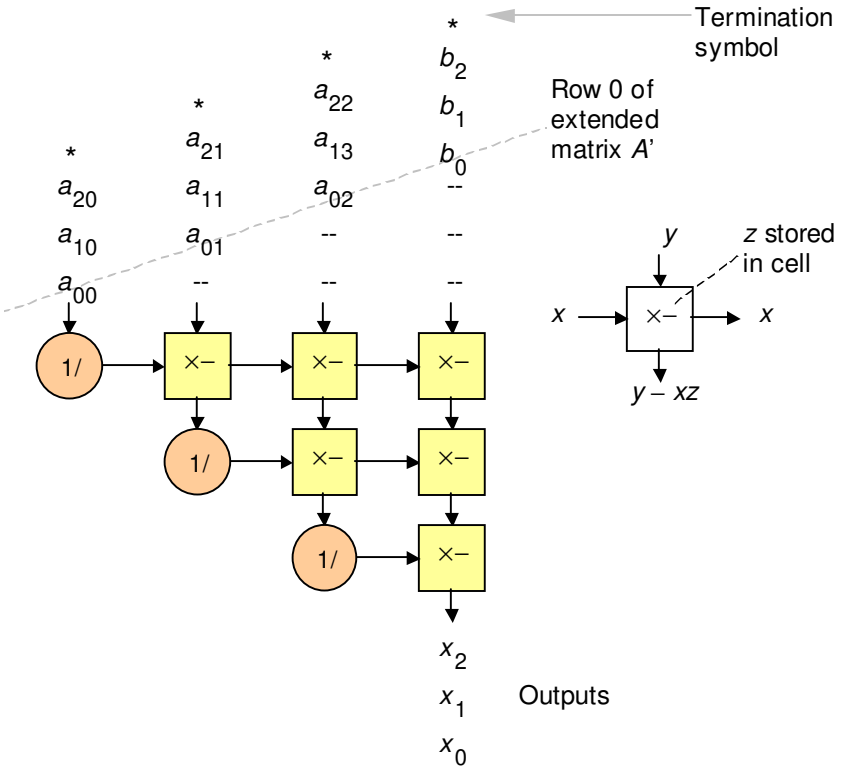


Fig. 11.13. Implementation of Gaussian elimination on a 2D array.

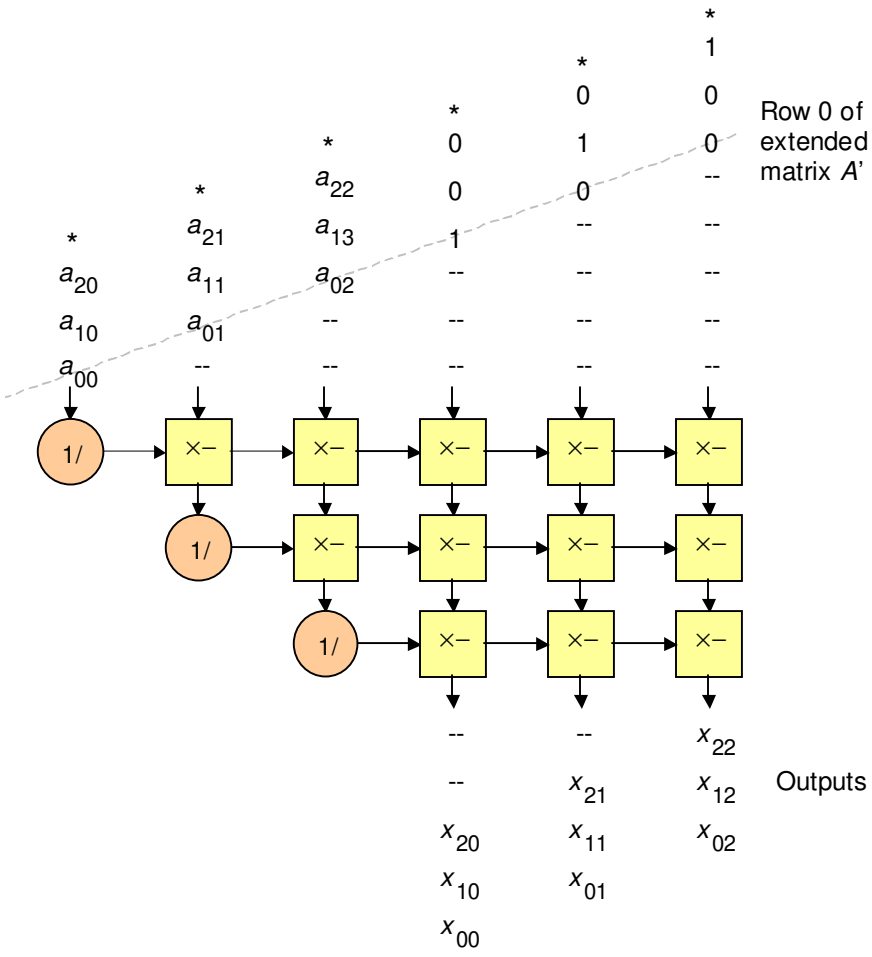


Fig. 11.14. Matrix inversion by Gaussian elimination.

Jacobi relaxation

Assuming $a_{ii} \neq 0$, solve the ith equation for x_i, yielding m equations from which new (better) approximations to the answers can be obtained.

$$x_i^{(t+1)} = (1/a_{ii})[b_i - \sum_{j \neq i} a_{ij} x_j^{(t)}]; \; x_i^{(0)} = \text{initial approx for } x_i$$

On an m-processor linear array, each iteration takes O(m) steps. The number of iterations needed is O(log m) in most cases, leading to O(m log m) average time.

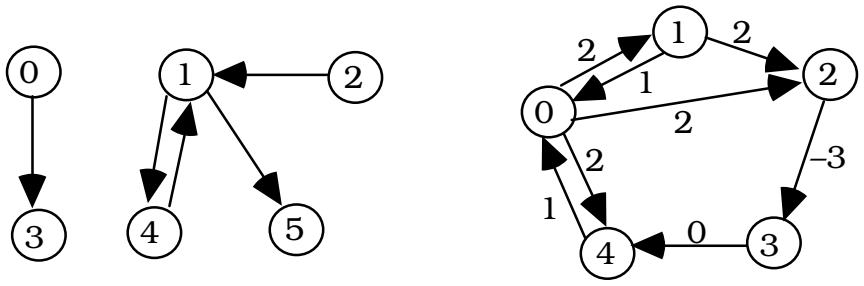
A variant: Jacobi overrelaxation

$$x_i^{(t+1)} = (1 - \gamma) x_i^{(t)} + (\gamma/a_{ii})[b_i - \sum_{j \neq i} a_{ij} x_j^{(t)}] \qquad 0 < \gamma \leq 1$$

For $\gamma = 1$, the method is the same as Jacobi relaxation

For smaller γ, overrelaxation may offer better performance

11.5 Graph Algorithms



$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad W = \begin{pmatrix} 0 & 2 & 2 & \infty & 2 \\ 1 & 0 & 2 & \infty & \infty \\ \infty & \infty & 0 & -3 & \infty \\ \infty & \infty & \infty & 0 & 0 \\ 1 & \infty & \infty & \infty & 0 \end{pmatrix}$$

Fig. 11.15. Matrix representation of directed graphs.

The transitive closure of a graph

Graph with same node set but with an edge between two nodes if there is any path between them in original graph

$A^0 = I$ Paths of length 0 (the identity matrix)

$A^1 = A$ Paths of length 1

Compute higher "powers" of A using matrix multiplication, except that AND/OR replace multiplication/addition

$A^2 = A \times A$ Paths of length 2

$A^3 = A^2 \times A$ Paths of length 3 etc.

The transitive closure has the adjacency matrix A^*

$$A^* = A^0 + A^1 + A^2 + \cdots \quad (A^*_{ij} = 1 \text{ iff } j \text{ is reachable from } i)$$

To compute A^*, we need only proceed up to the term A^{n-1}; if there exists a path from i to j, there is one of length $< n$

Rather than base the derivation of A^* on computing the various powers of the Boolean matrix A, we can use the following simpler algorithm:

Phase 0 — Insert the edge (i, j) into the graph if $(i, 0)$ and $(0, j)$ are in the graph

Phase 1 — Insert the edge (i, j) into the graph if $(i, 1)$ and $(1, j)$ are in the graph

.
.
.

Phase k — Insert the edge (i, j) into the graph if (i, k) and (k, j) are in the graph

Graph $A^{(k)}$ then has an edge (i, j) iff there is a path from i to j that goes only through nodes $\{1, 2, \cdots, k\}$ as intermediate hops

.
.
.

Phase $n-1$ — The graph $A^{(n-1)}$ is the required answer A^*

A key question is how to proceed so that each phase takes O(1) time for an overall O(n) time on an $n \times n$ mesh

The O(n) running time would be optimal due to the O(n^3) sequential complexity of the transitive closure problem

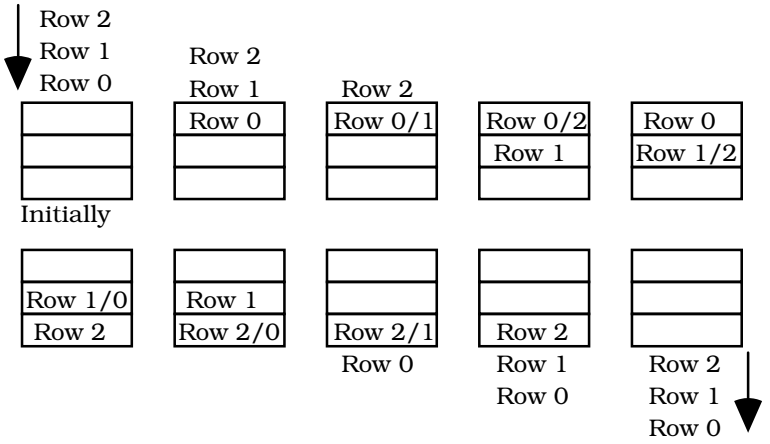
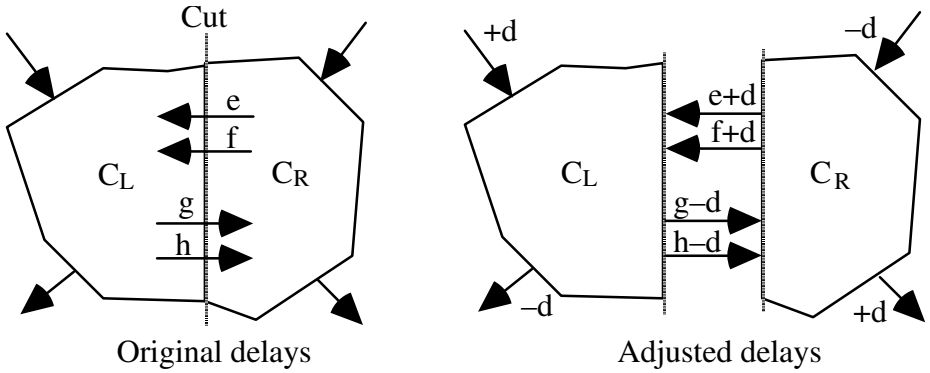


Fig. 11.16. Transitive closure algorithm on a 2D mesh.

Systolic retiming



Example of retiming by delaying the inputs to C_L and advancing the outputs from C_L by *d* units [Fig. 12.8 in *Computer Arithmetic: Algorithms and Hardware Designs*, by Parhami, Oxford, 2000]

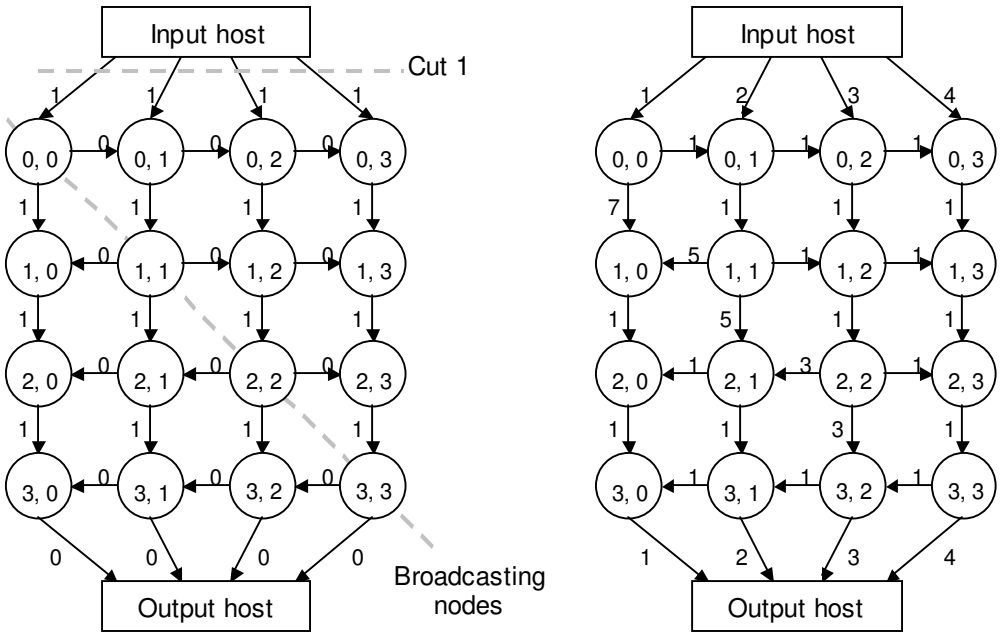


Fig. 11.17. Systolic retiming to eliminate broadcasting.

Diagram on the left represents the algorithm

Zero-time horizontal arrows represent broadcasting by diagonal elements

Goal of systolization is to eliminate zero-time transitions

To systolize the preceding example:

Add $2n - 2 = 6$ units of delay to edges crossing cut 1

Move 6 units of delay from inputs to outputs of node (0, 0)

11.6 Image-Processing Algorithms

Labeling the connected components of a binary image

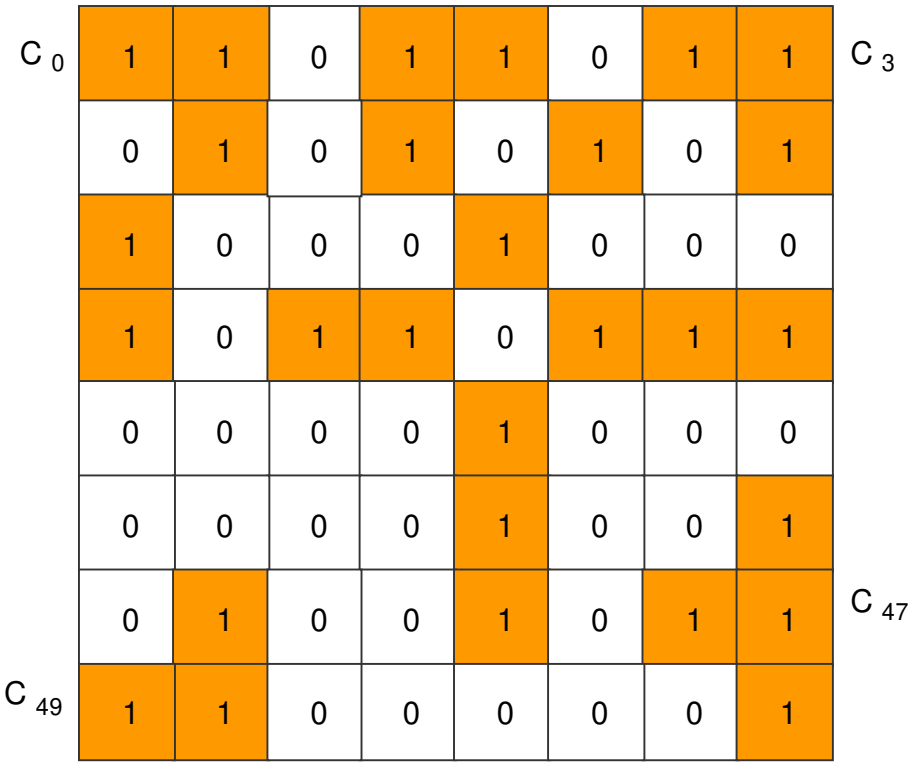


Fig. 11.18. Connected components in an 8×8 binary image.

Recursive algorithm, $p = n$:

$$T(n) = T(n/4) + O(\sqrt{n}) = O(\sqrt{n})$$

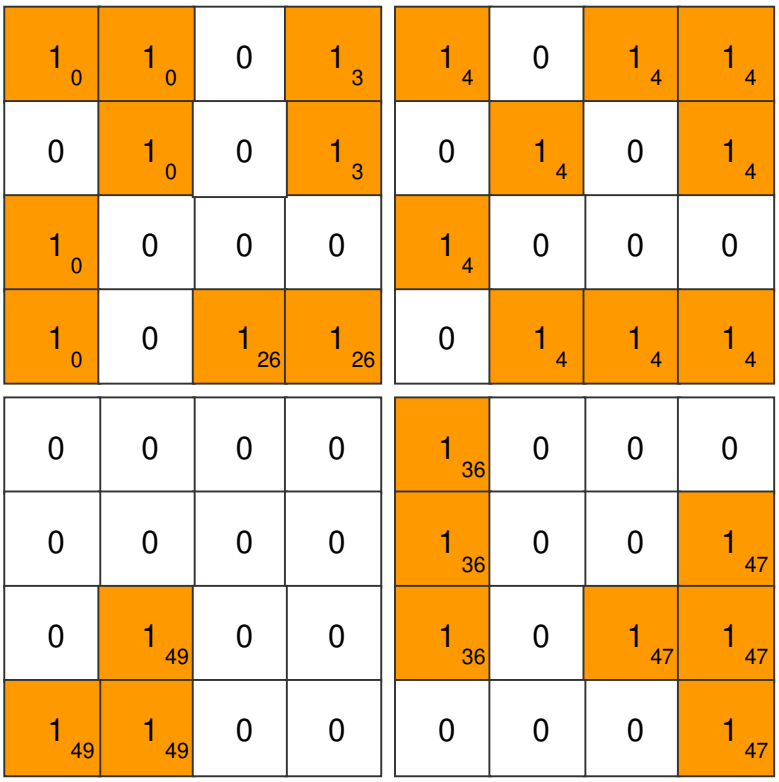


Fig. 11.19. Finding the connected components via divide and conquer.

Levialdi's algorithm

```
0 1    1 1    0 is changed to 1
1[0]   1[0]   if N = W = 1

       0 0    1 is changed to 0
       0[1]   if N = W = NW = 0
```

Fig. 11.20. Transformation or rewriting rules for Levialdi's algorithm in the shrinkage phase (no other pixel changes).

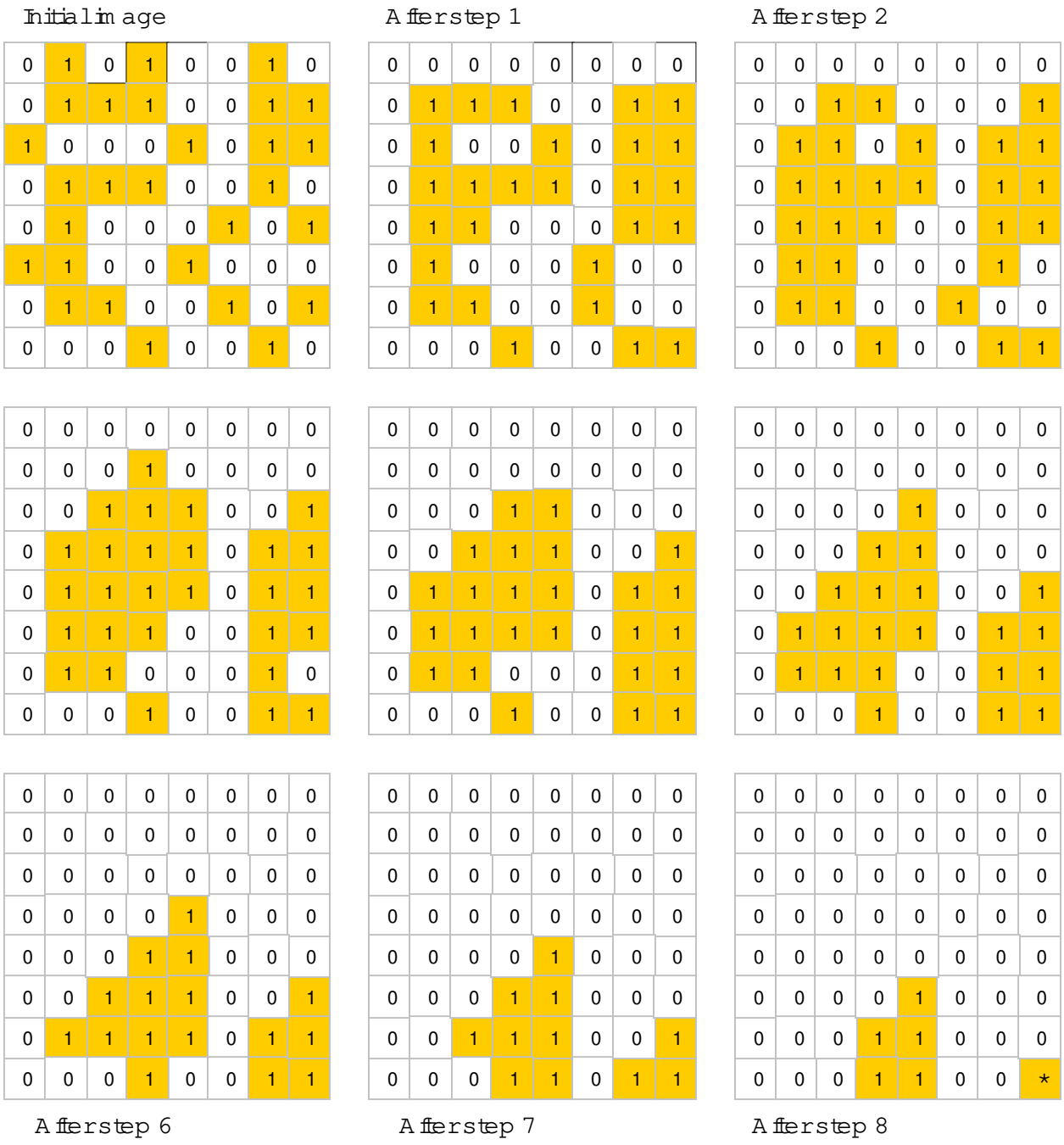


Fig. 11.21. Example of the shrinkage phase of Levialdi's component labeling algorithm.

Latency of Levialdi's algorithm

$$T(n) = 2\sqrt{n} - 1 \text{ \{shrinkage\}} + 2\sqrt{n} - 1 \text{ \{expansion\}}$$

Component do not merge in the shrinkage phase

Consider a 0 that is about to become a 1

x	1	*y*	If any *y* is 1, then already connected
1	0	*y*	If *z* is 1 then it will change to 0 unless
y	*y*	*z*	at least one neighboring *y* is 1

12 Mesh-Related Architectures

Back to TOC

Chapter Goals

- Study variants of simple mesh architectures that offer higher performance or greater cost-effectiveness
- Learn about related architectures such as pyramids and mesh of trees

Chapter Contents

- 12.1. Three or More Dimensions
- 12.2. Stronger and Weaker Connectivities
- 12.3. Meshes Augmented with Nonlocal Links
- 12.4. Meshes with Dynamic Links
- 12.5. Pyramid and Multigrid Systems
- 12.6. Meshes of Trees

12.1 Three or More Dimensions

3D mesh: $D = 3p^{1/3} - 3$ instead of $2p^{1/2} - 2$
$B = p^{2/3}$ rather than $p^{1/2}$

Example: $8 \times 8 \times 8$ mesh $D = 21$, $B = 64$
22×23 mesh $D = 43$, $B = 23$

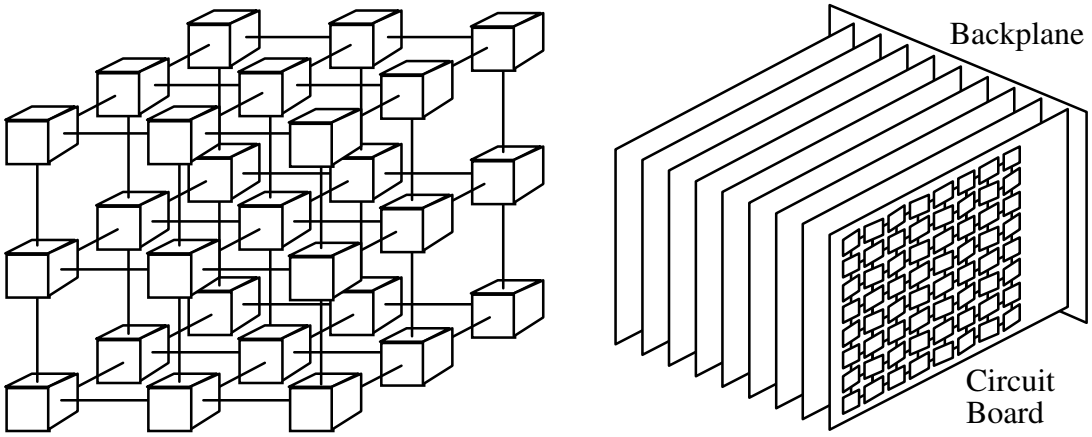
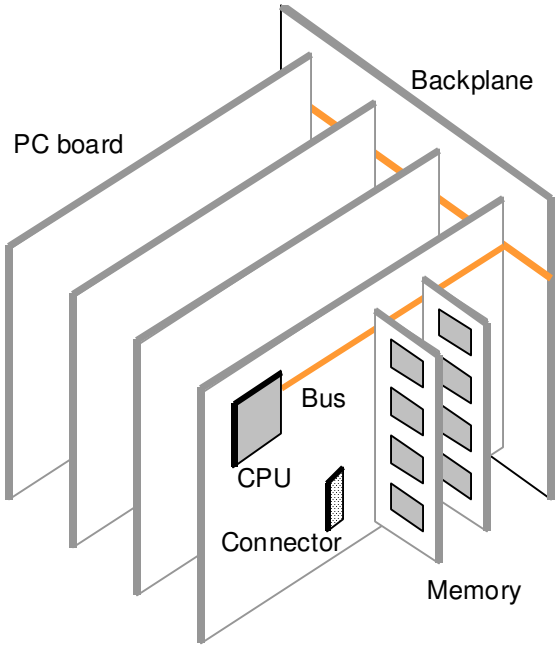
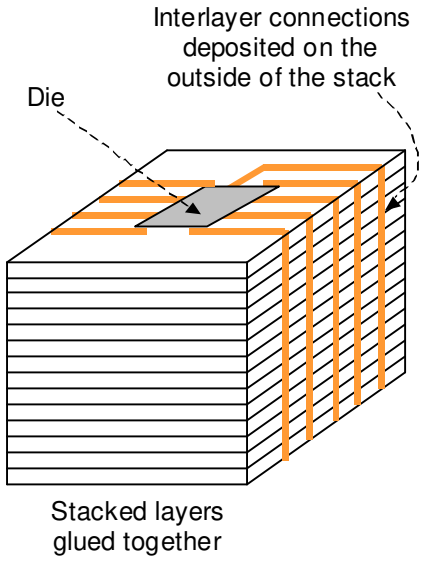


Fig. 12.1. 3D and 2.5D physical realizations of a 3D mesh.

Packaging issues for higher-dimensional meshes



(a) 2D or 2.5D packaging now common



(b) 3D packaging of the future

4D, 5D, . . . meshes: optical links?

qD mesh with m processors along each dimension: $p = m^q$

Node degree $d = 2q$

Diameter $D = q(m - 1) = q\,(p^{1/q} - 1)$

Bisection width: $B = p^{1-1/q}$ when $m = p^{1/q}$ is even

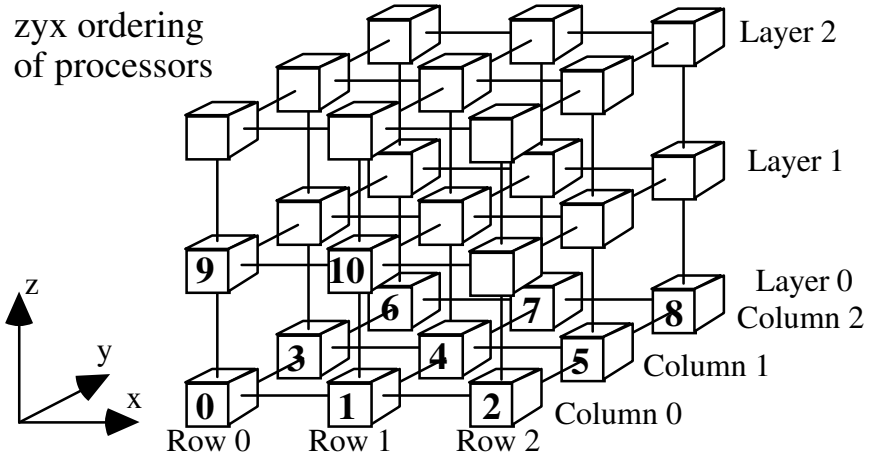
qD torus with m processors along each dimension

= m-ary q-cube

Sorting on a 3D mesh

A generalized form of shearsort is available

However, the following algorithm (due to Kunde) is both faster and simpler. Let Processor (*i*, *j*, *k*) in an $m \times m \times m$ mesh be in Row *i*, Column *j*, and Layer *k*



Sorting on 3D mesh (*zyx* order; reverse of node index)

Phase 1: Sort elements on each *zx* plane into *zx* order

Phase 2: Sort elements on each *yz* plane into *zy* order

Phase 3: Sort elements on each *xy* layer into *yx* order (odd layers in reverse order).

Phase 4: Apply 2 steps of odd-even transposition along *z*

Phase 5: Sort elements on each *xy* layer into *yx* order

Time = $4 \times$ (2D-sort time) + 2 steps

Data routing on a 3D mesh

Greedy *zyx* (layer-first, row last) routing algorithm

Phase 1: Sort into *zyx* order by destination addresses

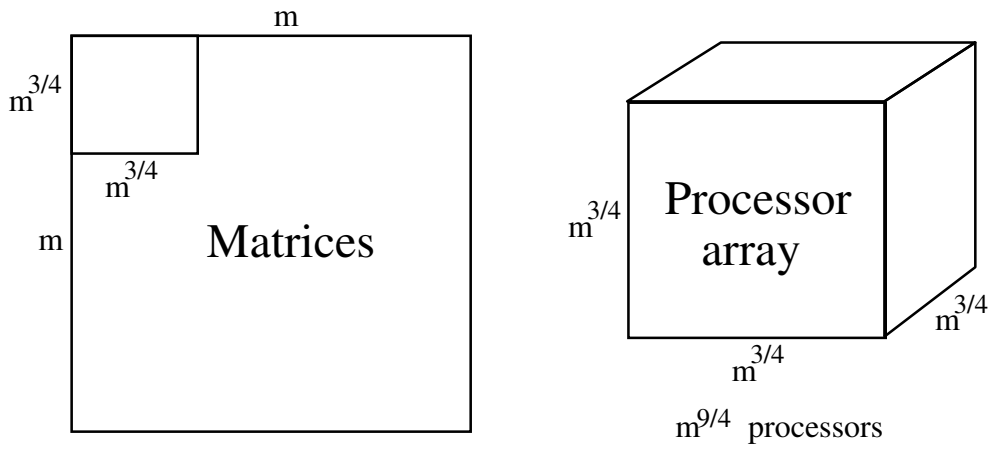
Phase 2: Route along *z* dimension to correct *xy* layer

Phase 3: Route along *y* dimension to correct column

Phase 4: Route along *x* dimension to destination

Matrix multiplication on a 3D mesh

Divide matrices into $m^{1/4} \times m^{1/4}$ arrays of $m^{3/4} \times m^{3/4}$ blocks



A total of $(m^{1/4})^3 = m^{3/4}$ block multiplications are needed

Assume the use of an $m^{3/4} \times m^{3/4} \times m^{3/4}$ mesh with $p = m^{9/4}$

Each $m^{3/4} \times m^{3/4}$ layer of the mesh is assigned to one of the $m^{3/4} \times m^{3/4}$ matrix multiplications ($m^{3/4}$ multiply-adds)

The rest of the process takes time that is of lower order

The algorithm matches both the sequential work and the diameter-based lower bound

Modeling of physical systems

Natural mapping of a 3D physical model to a 3D mesh

Low- vs. high-dimensional meshes

A low-dimensional mesh can simulate a high-dimensional mesh quite efficiently

It is thus natural to ask the following question:

Is it more cost effective, e.g., to have 4-port processors in a 2D mesh architecture or 6-port processors in a 3D mesh, given that for the 4-port processors, fewer ports and ease of layout allow us to make each channel wider?

12.2 Stronger and Weaker Connectivities

Fortified meshes

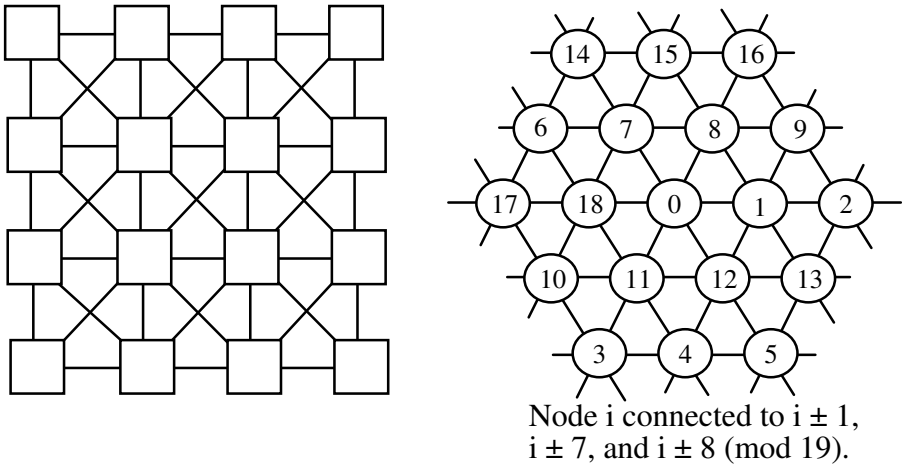


Fig. 12.2. Eight-neighbor and hexagonal (hex) meshes.

Oriented meshes (can be viewed as a type of pruning)

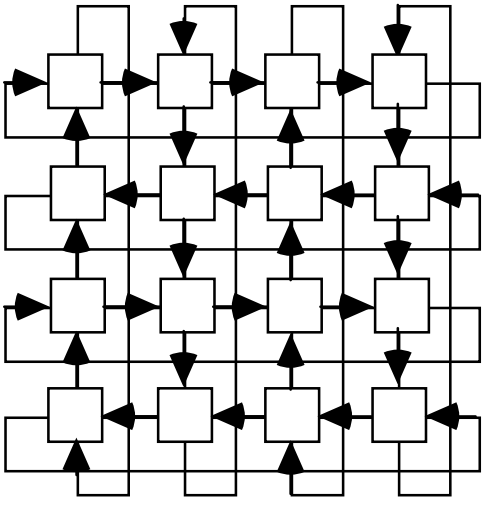


Fig. 12.3. A 4 × 4 Manhattan street network.

Pruned meshes

Same diameter as ordinary mesh, but much lower cost.

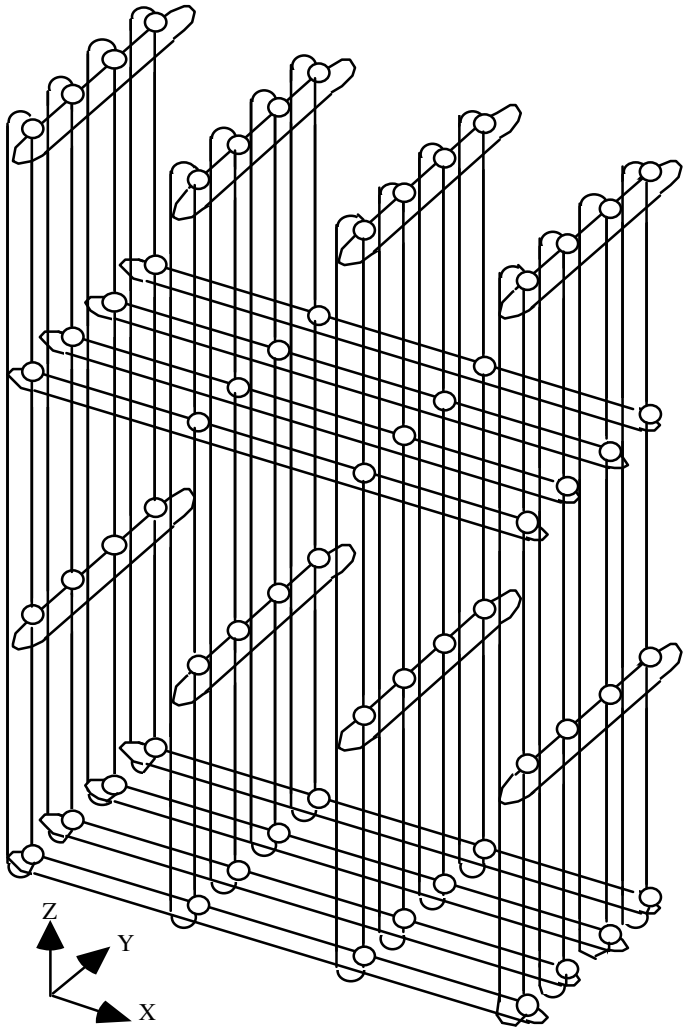
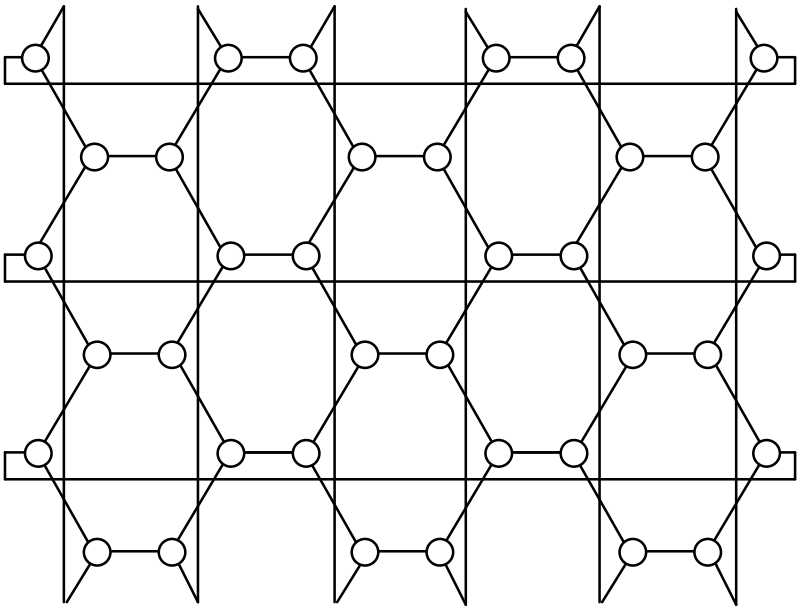


Fig. 12.4. A pruned $4 \times 4 \times 4$ torus with nodes of degree four [Kwai97].

Pruning and orientation can be combined

Another form of pruning



Honeycomb mesh or torus.

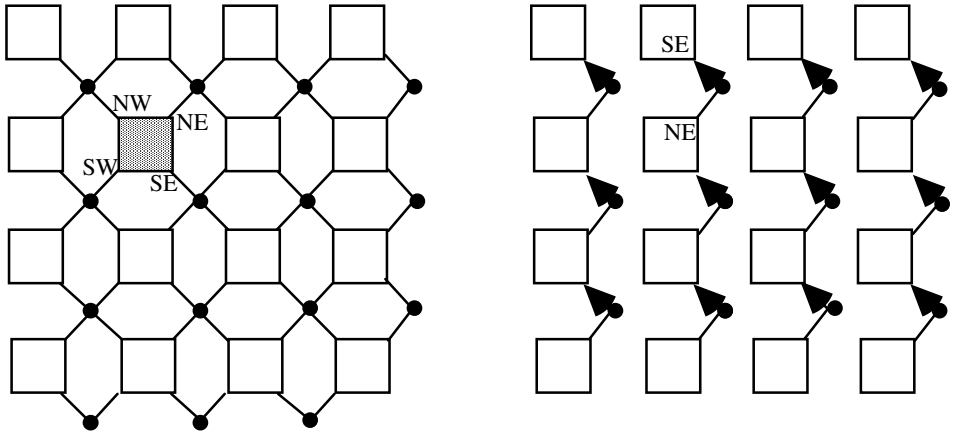


Fig. 12.5. Eight-neighbor mesh with shared links and example data paths.

12.3 Meshes Augmented with Nonlocal Links

Motivation: reduce the diameter, a weakness of meshes

Bypass links or express channels along rows/columns

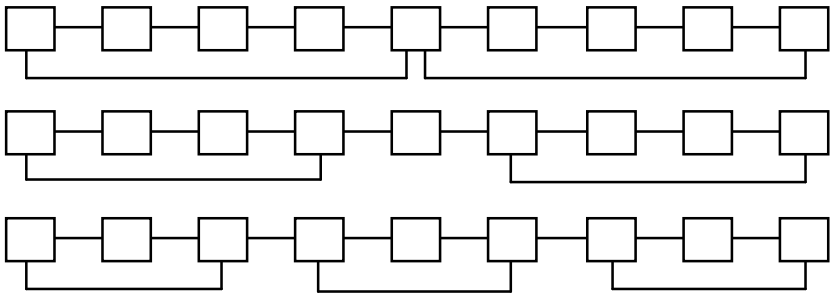
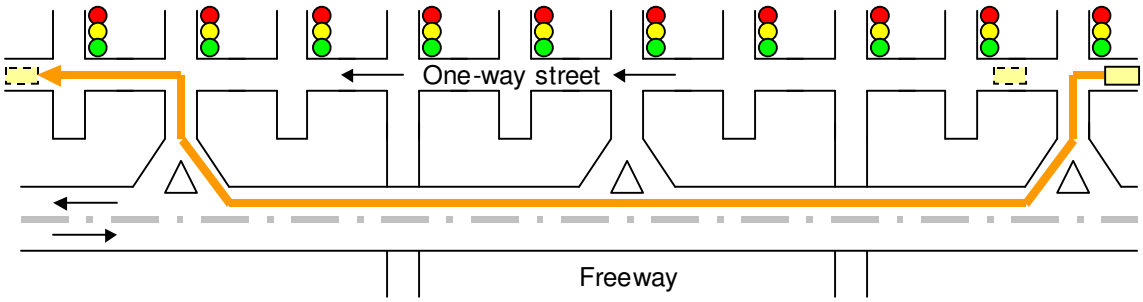


Fig. 12.6. Three examples of bypass links along the rows of a 2D mesh.



Road analogy for bypass connections.

Using a single global bus

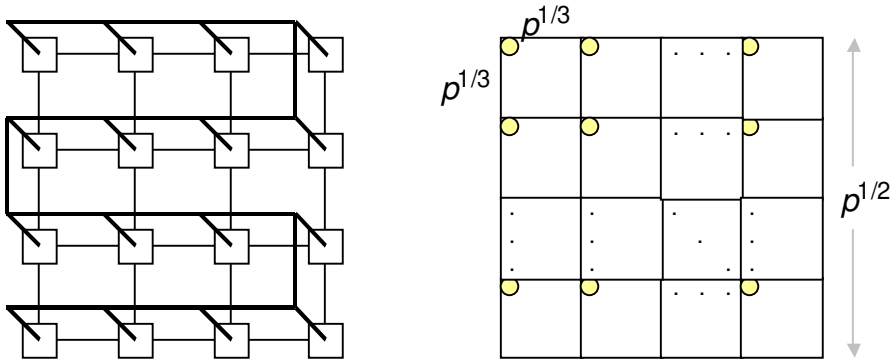


Fig. 12.7. Mesh with a global bus and semigroup computation on it.

A $\sqrt{p} \times \sqrt{p}$ mesh with a single global bus can perform a semigroup computation $O(p^{1/3})$ rather than $O(p^{1/2})$ steps

Assume that the semigroup operation $\otimes$ is commutative

Semigroup computation on 2D mesh with a global bus

Phase 1: Find the partial results in $p^{1/3} \times p^{1/3}$ submeshes in $O(p^{1/3})$ steps; results stored in the upper left corner of each submesh

Phase 2: Combine the partial results in $O(p^{1/3})$ steps, using a sequential algorithm in one node and the global bus for data transfers

Phase 3: Broadcast the result to all nodes (one step)

Row and column buses

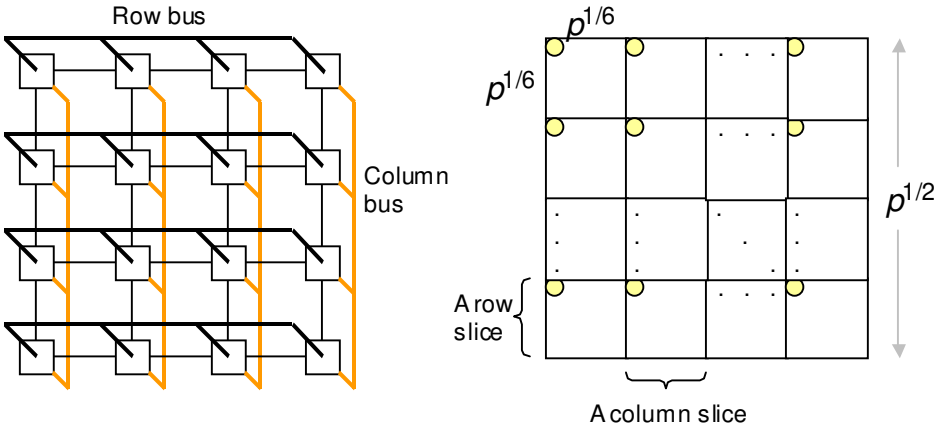


Fig. 12.8. Mesh with row/column buses and semigroup computation on it.

2D-mesh semigroup computation, row/column buses

Phase 1: Find the partial results in $p^{1/6} \times p^{1/6}$ submeshes in $O(p^{1/6})$ steps

Phase 2: Distribute the $p^{1/3}$ values left on some rows among the $p^{1/6}$ rows in the same slice

Phase 3: Combine row values in $p^{1/6}$ steps (row bus)

Phase 4: Distribute column-0 values to $p^{1/3}$ columns

Phase 5: Combine column values in $p^{1/6}$ steps

Phase 6: Use column buses to distribute the $p^{1/3}$ values on row 0 among the $p^{1/6}$ rows of row slice 0 in constant time

Phase 7: Combine row values in $p^{1/6}$ steps

Phase 8: Broadcast the result to all nodes (2 steps)

12.4 Meshes with Dynamic Links

Linear array with a separable bus

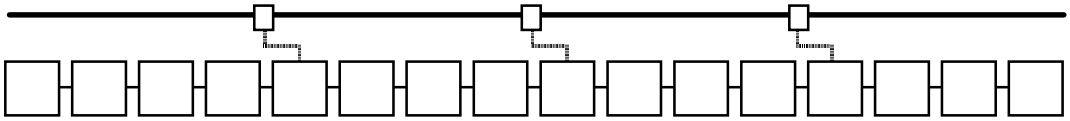


Fig. 12.9. Linear array with a separable bus using reconfiguration switches.

Semigroup computation: O(log *p*) steps

2D mesh with separable row/column buses

Reconfigurable mesh architecture

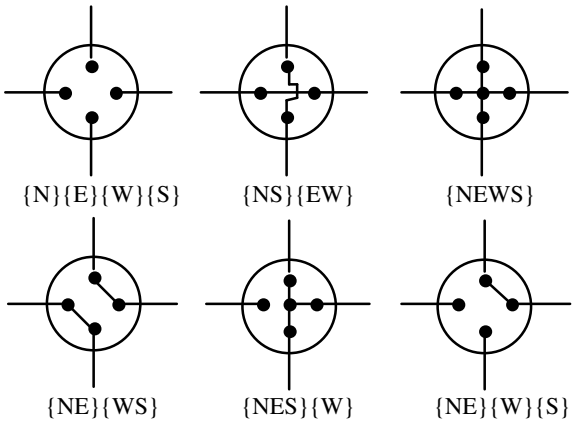


Fig. 12.10. Some processor states in a reconfigurable mesh.

12.5 Pyramid and Multigrid Systems

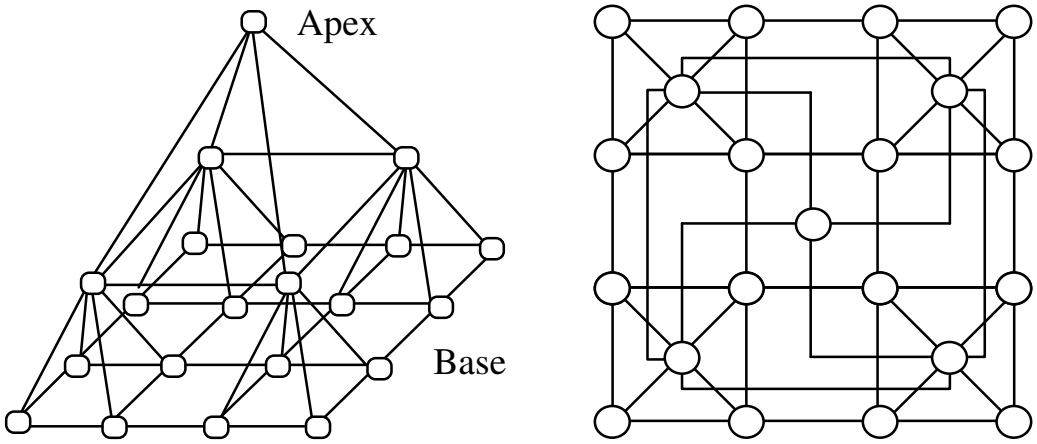


Fig. 12.11. Pyramid with 3 levels and 4×4 base along with its 2D layout.

Originally developed for image processing applications

Roughly 3/4 of the processors belong to the base

For an l-level pyramid: $D = 2l - 2$ $d = 9$ $B = 2^l$

Semigroup computation faster than on mesh, but not sorting or arbitrary routing

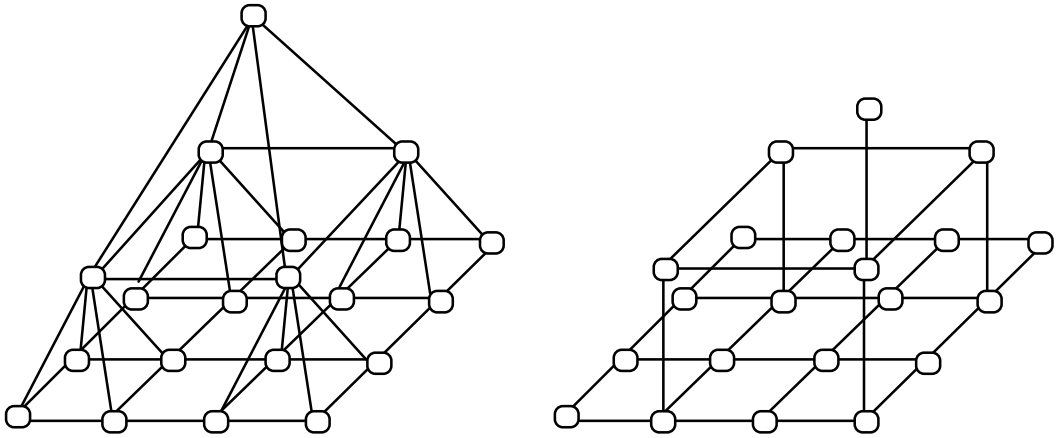


Fig. 12.12. The relationship between pyramid and 2D multigrid architectures.

12.6 Meshes of Trees

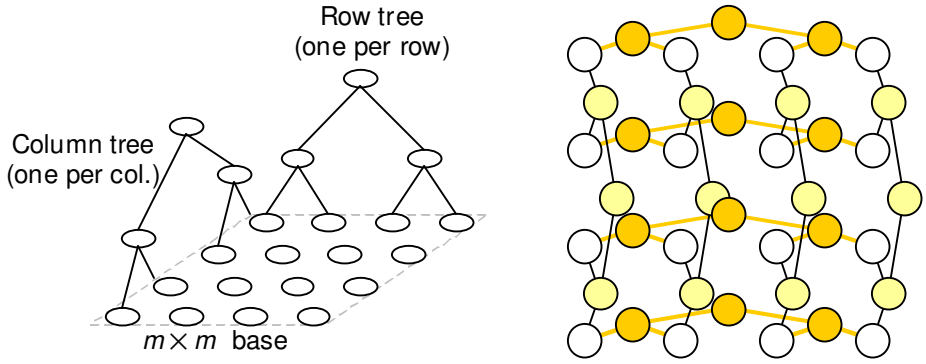


Fig. 12.13. Mesh of trees architecture with 3 levels and a 4 × 4 base.

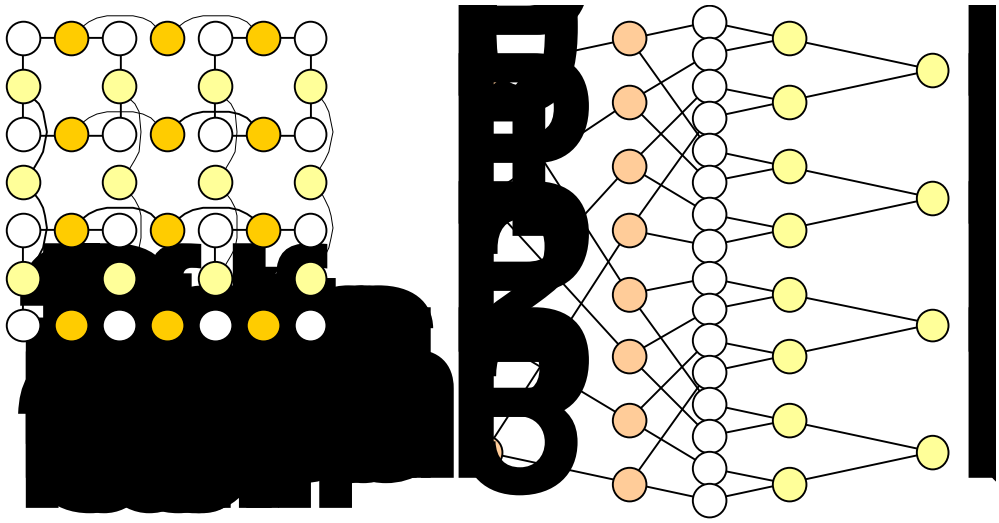


Fig. 12.14. Alternate views of the mesh of trees architecture with a 4×4 base.

Semigroup computation: done via row/column combining

Parallel prefix computation: similar

Routing m^2 packets, one per processor on the $m \times m$ base: row-first routing yields an $\Omega(m) = \Omega(\sqrt{p})$ scheme

In the view of Fig. 12.14, with only m packets to be routed from one side of the network to the other, $2 \log_2 m$ steps are required, provided that destination nodes are distinct

Sorting m^2 keys, one per processor on the $m \times m$ base: emulate shearshort

In the view of Fig. 12.14, with only m keys to be sorted, the following algorithm can be used (assume that row/column root nodes have been merged and each holds one key)

Sorting m keys on a mesh of trees with an $m \times m$ base

Phase 1: Broadcast keys to leaves within both trees

(leaf i,j gets x_i and x_j)

Phase 2: At a base node:

if $x_j > x_i$ or $x_j = x_i$ and $j > i$ then $flag := 1$ else $flag := 0$

Phase 3: Add the "flag" values in column trees

(root i obtains the rank of x_i)

Phase 4: Route x_i from root i to root $rank[i]$

Matrix-vector multiplication $Ax = y$: matrix A is stored on the base and vector x in the column roots, say; the result vector y is obtained in the row roots

Multiplying $m \times m$ matrix by m-vector on mesh of trees

Phase 1: Broadcast x_j in the ith column tree

(leaf i,j has a_{ij} and x_i)

Phase 2: At each base processor compute $a_{ij} x_j$

Phase 3: Sum over row trees

(row root i obtains $\sum_{j=0}^{m-1} a_{ij} x_j = y_i$)

With pipelining, r matrix-vector pairs multiplied in $2l - 2 + r$ steps

Convolution of two vectors

Assume the mesh of trees with an $m \times (2m - 1)$ base contains m diagonal trees in addition to the row and column trees, as shown in Fig. 12.15

Convolution of two m-vectors on a mesh of trees with an $m \times (2m - 1)$ base

Phase 1: Broadcast x_j from the ith row root to all row nodes on the base

Phase 2: Broadcast y_{m-1-j} from the diagonal root to the base diagonal

Phase 3: Leaf i,j, which has x_i and $y_{2m-2-i-j}$, multiplies them to get $x_i y_{2m-2-i-j}$

Phase 4: Sum columns to get $z_{2m-2-j}=\sum_{i=0}^{m-1} x_i y_{2m-2-i-j}$ in column root j

Phases 1 and 2 can be overlapped

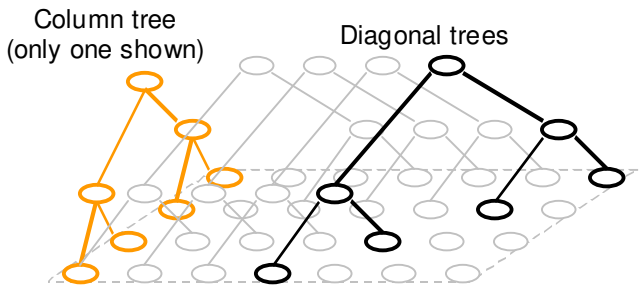


Fig. 12.15. Mesh of trees variant with row, column, and diagonal trees.

Minimal-weight spanning tree for an undirected graph

A spanning tree of a connected graph is a subset of its edges that preserves the connectivity of all nodes in the graph but does not contain any cycle

A minimal-weight spanning tree (MWST) is a subset of edges that has the minimum total weight among all spanning trees

This is an important problem: if the graph represents a communication (transportation) network, MWSP tree might correspond to the best way to broadcast a message to all nodes (deliver products to the branches of a chain store from a central warehouse)

Greedy sequential MWST algorithm

Assume weights are distinct: min-weight edge is unique

At each step, we have a set of connected components or "supernodes" (initially *n* single-node components)

We connect each component to its "nearest" neighbor; i.e., we find the min-weight edge connecting it to another

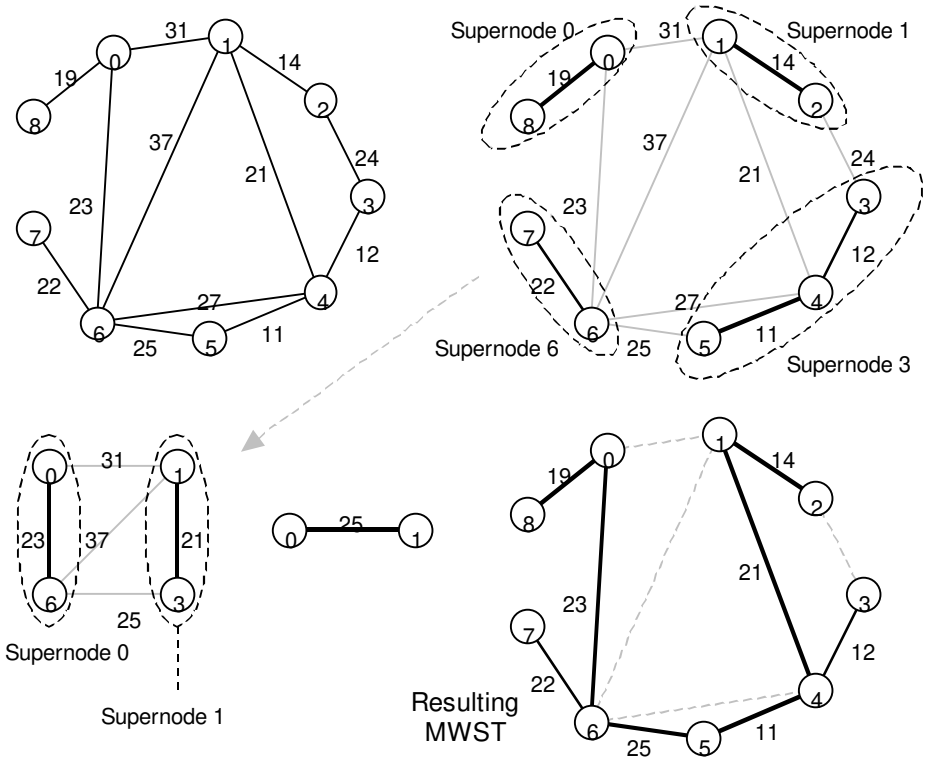


Fig. 12.16. Example for min-weight spanning tree algorithm.

If the graph's weight matrix W is stored in the leaves of a mesh of trees architecture, each phase requires $O(\log^2 n)$ steps with a simple algorithm (to be shown) and $O(\log n)$ steps with a more sophisticated algorithm.

The total running time is thus $O(\log^3 n)$ or $O(\log^2 n)$.

Sequential algorithms and their time complexities:

Kruskal's: $O(e \log e) \Rightarrow O(n^2 \log n)$ for dense graphs

Prim's (binary heap): $O((e + n)\log n) \Rightarrow O(n^2 \log n)$

Prim's (Fibonacci heap): $O(e + n \log n) \Rightarrow O(n^2)$

Our best parallel solution offers a speedup of $O(n^2/\log^2 n)$; sublinear in the number $p = O(n^2)$ of processors

Key part of the simple parallel version of greedy algorithm is showing that each phase takes $O(\log^2 n)$ steps.

The algorithm for each phase consists of two subphases:

a. Find the min-weight edge incident to each supernode

b. Merge the supernodes for the next phase

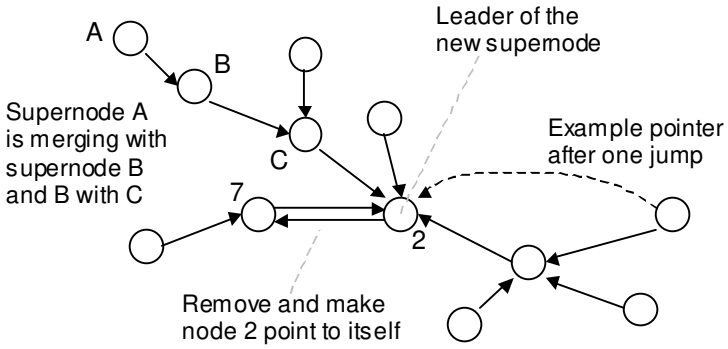


Fig. 12.17. Finding the new supernode ID when several supernodes merge.

Part IV Low-Diameter Architectures

Back to TOC

Part Goals

- Study the hypercube as an example of architectures with
  - low (logarithmic) diameter
  - wide bisection
  - rich theoretical properties
- Discuss hypercube's realizability/scalability problems and present alternatives
- Complete our view of the "sea of interconnection networks"

Part Contents

- Chapter 13: Hypercubes and Their Algorithms
- Chapter 14: Sorting and Routing on Hypercubes
- Chapter 15: Other Hypercubic Architectures
- Chapter 16: A Sampler of Other Networks

13 Hypercubes and Their Algorithms

Back to TOC

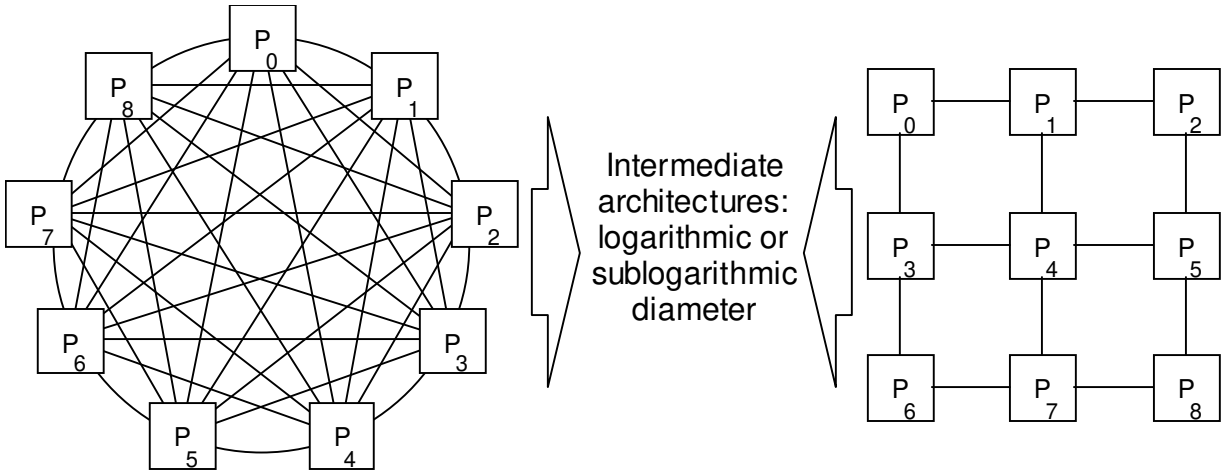
Chapter Goals

- Introduce the hypercube and its topological and algorithmic properties
- Design simple hypercube algorithms (sorting & routing to follow in Chapter 14)
- Learn about embeddings and their role in algorithm design and evaluation

Chapter Contents

- 13.1. Definition and Main Properties
- 13.2. Embeddings and Their Usefulness
- 13.3. Embedding of Arrays and Trees
- 13.4. A Few Simple Algorithms
- 13.5. Matrix Multiplication
- 13.6. Inverting a Lower Triangular Matrix

13.1 Definition and Main Properties



Binary tree has logarithmic diameter, but small bisection

Hypercube has a much larger bisection

Hypercube can be viewed as a mesh with the largest possible number of dimensions

$$2 \times 2 \times 2 \times \ \ldots \ \times 2$$
$$\longleftarrow \log_2 p \longrightarrow$$

We saw that increasing the number of dimensions made it harder to design and visualize algorithms for the mesh

Oddly, at the extreme of $\log_2 p$ dimensions, things become simple again!

Brief history of the hypercube (binary q-cube) architecture

- Concept developed: early 1960s [Squi63]
- Direct (single-stage) & indirect or multistage versions proposed for parallel processing: mid 1970s (early proposals [Peas77], [Sull77], no hardware)
- Caltech's 64-node Cosmic Cube: early 1980s [Seit85] elegant solution to routing (wormhole routing)
- Several commercial machines: mid to late 1980s Intel PSC, CM-2, nCUBE (Section 22.3)

Terminology

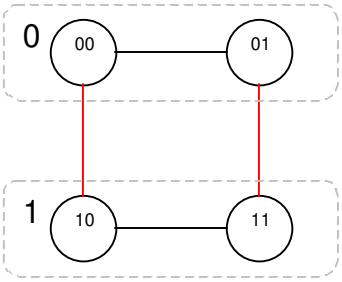
- Hypercube: generic term
- 3-cube, 4-cube, . . . , q-cube
  - when the number of dimensions is of interest

A qD binary hypercube (q-cube) is defined recursively:

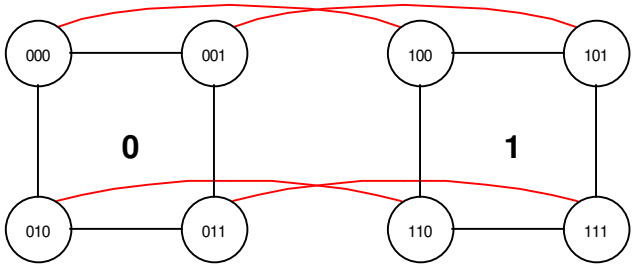
- 1-cube: 2 connected nodes, labeled 0 and 1
- q-cube consists of two $(q-1)$-cubes; 0 & 1 subcubes

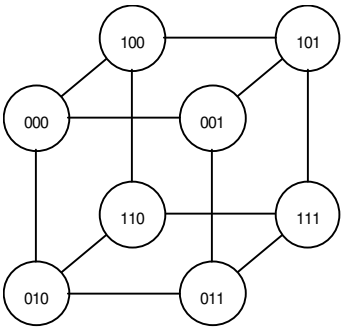
q-cube nodes labeled by preceding subcube node labels with 0 and 1 and connecting node $0x$ to node $1x$

(a) Binary 1-cube, built of two binary 0-cubes, labeled 0 and 1

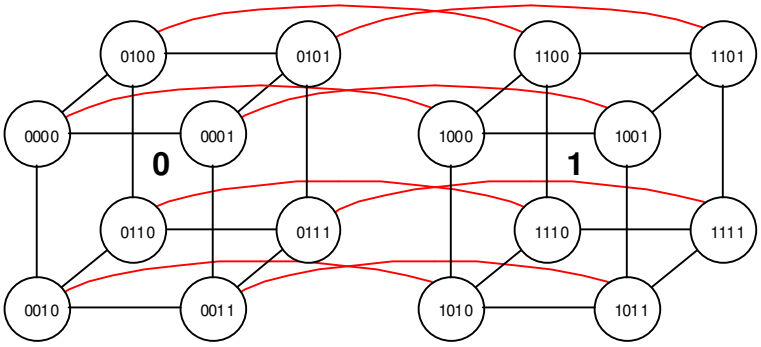


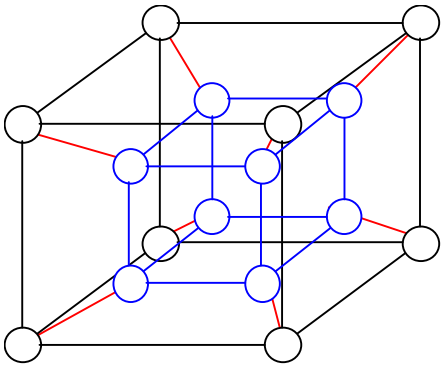
(b) Binary 2-cube, built of two binary 1-cubes, labeled 0 and 1





(c) Binary 3-cube, built of two binary 2-cubes, labeled 0 and 1





(d) Binary 4-cube, built of two binary 3-cubes, labeled 0 and 1

Fig. 13.1. The recursive structure of binary hypercubes.

Number of nodes in a q-cube: $p = 2^q$

Bisection width: $B = p / 2 = 2^{q-1}$

Diameter: $D = q = \log_2 p$

Node degree: $d = q = \log_2 p$

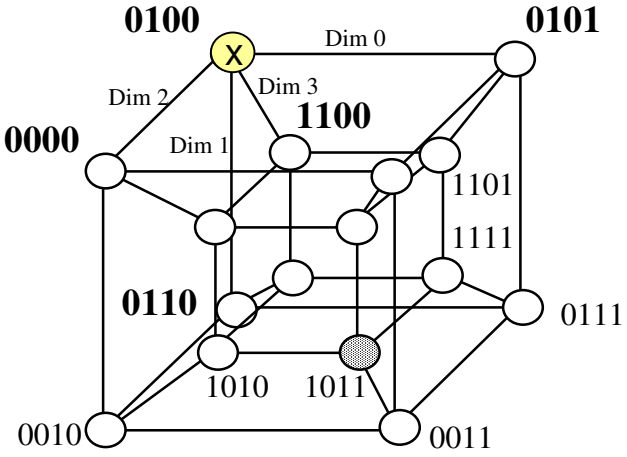
q neighbors of node x with binary ID $x_{q-1}x_{q-2} \cdots x_2x_1x_0$:

$x_{q-1}x_{q-2} \cdots x_2x_1 \bar{x}_0$ dimension-0 neighbor; $N_0(x)$

$x_{q-1}x_{q-2} \cdots x_2 \bar{x}_1x_0$ dimension-1 neighbor; $N_1(x)$

. . .

$\bar{x}_{q-1}x_{q-2} \cdots x_2x_1x_0$ dimension-$(q-1)$ neighbor; $N_{q-1}(x)$



Some properties of hypercubes:

Two nodes whose labels differ in *k* bits (have a Hamming distance of *k*) are connected by a shortest path of length *k*

Logarithmic diameter and linear bisection width are key reasons for the hypercube's high performance

Hypercube is both node- and edge-symmetric

Logarithmic node degree hinders hypercube's scalability

13.2 Embeddings and Their Usefulness

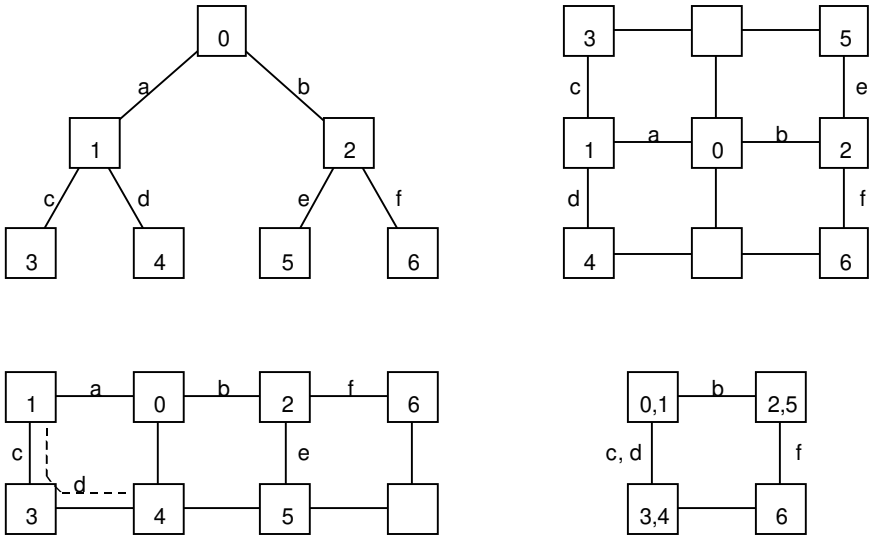


Fig. 13.2. Embedding a seven-node binary tree into 2D meshes of various sizes.

		Examples of Fig. 13.2 →	3×3	2×4	2×2
Dilation	Longest path onto which any edge is mapped (indicator of communication slowdown)		1	2	1
Congestion	Max number of edges mapped onto one edge (indicator of contention during emulation)		1	2	2
Load factor	Max number of nodes mapped onto one node (indicator of processing slowdown)		1	1	2
Expansion	Ratio of number of nodes in the two graphs (indicator of emulation cost)		9/7	8/7	4/7

13.3 Embedding of Arrays and Trees

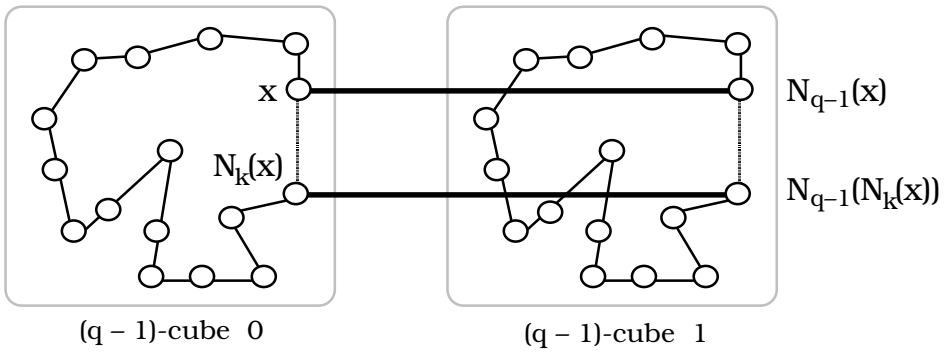


Fig. 13.3. Hamiltonian cycle in the *q*-cube.

Proof of Hamiltonicity using Gray code:

	Assumed Gray code	Assumed Gray code in reverse
	←——————→	←——————→
(q–1)-bit codes	0^{q-1} $0^{q-2}1$ · · · 10^{q-2}	10^{q-2} · · · $0^{q-2}1$ 0^{q-1}
q-bit Gray code	0^{q} $0^{q-1}1$ · · · 010^{q-2}	110^{q-2} · · · $10^{q-2}1$ 10^{q-1}
	←——————→	←——————→
	Prefix with 0	Prefix with 1

The $2^{m_0} \times 2^{m_1} \times \cdots \times 2^{m_{h-1}}$ mesh/torus is a subgraph of q-cube

where $q = m_0 + m_1 + \cdots + m_{h-1}$

This is akin to the mesh/torus being embedded in q-cube with dilation 1, congestion 1, load factor 1, expansion 1

The proof is based on the notion of cross-product graphs

Given k graphs $G_i = (V_i, E_i)$, $1 \leq i \leq k$, their (cross-)product graph $G = G_1 \times G_2 \times \cdots \times G_k = (V, E)$ has:

node set $V = \{(v_1, v_2, \cdots, v_k) \mid v_i \in V_i, 1 \leq i \leq k\}$

edge set $E = \{[(u_1, u_2, \cdots, u_k), (v_1, v_2, \cdots, v_k)] \mid$

for some j, $(u_j, v_j) \in E_j$ and for $i \neq j$, $u_i = v_i\}$

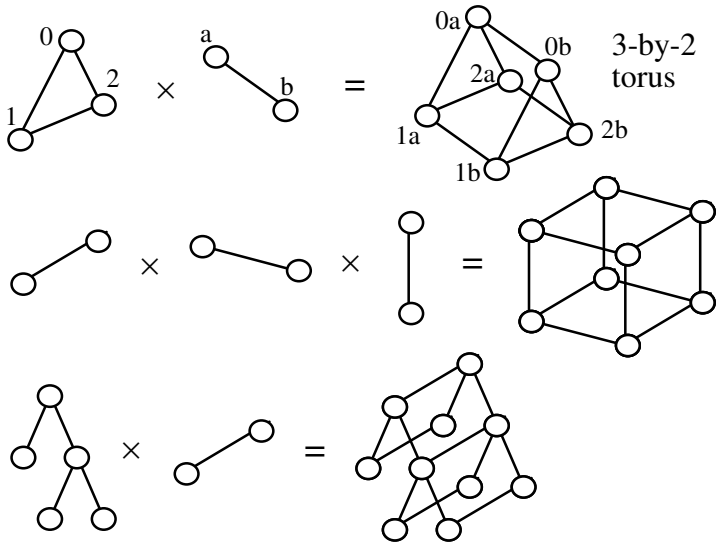


Fig. 13.4. Examples of product graphs.

a. The $2^{m_0} \times 2^{m_1} \times \cdots \times 2^{m_{h-1}}$ torus is the product of h rings of sizes 2^{m_0}, 2^{m_1}, . . . , $2^{m_{h-1}}$

b. The $(m_0 + m_1 + \cdots + m_{h-1})$-cube is the product of an m_0-cube, an m_1-cube, . . . , an m_{h-1}-cube

c. The 2^{m_i}-node ring is a subgraph of the m_i-cube

d. If component graphs are subgraphs of other component graphs, then the product graph will be a subgraph of the other product graph

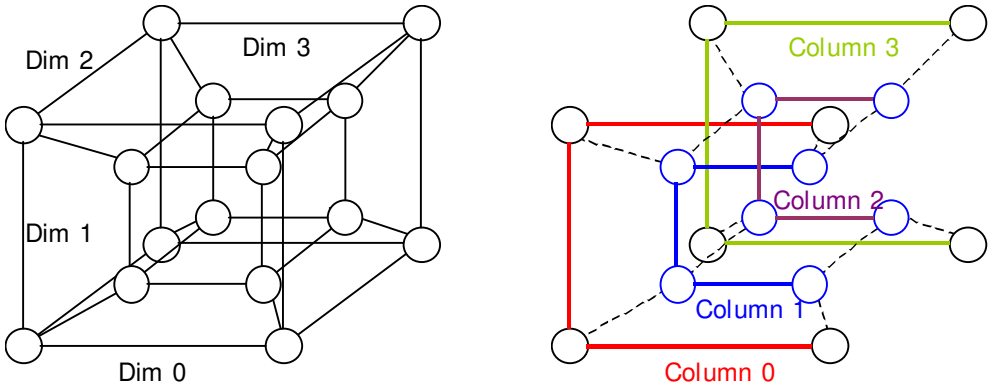
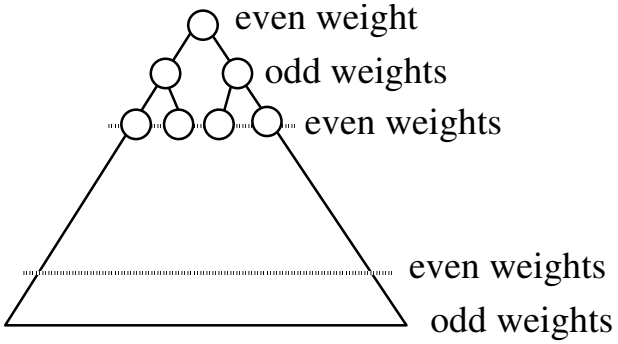


Fig. 13.5. The 4 × 4 mesh/torus is a subgraph of the 4-cube.

Embedding ($2^q - 1$)-node complete binary tree in q-cube

Achieving dilation 1 is impossible



Embedding the 2^q-node double-rooted complete binary tree in q-cube

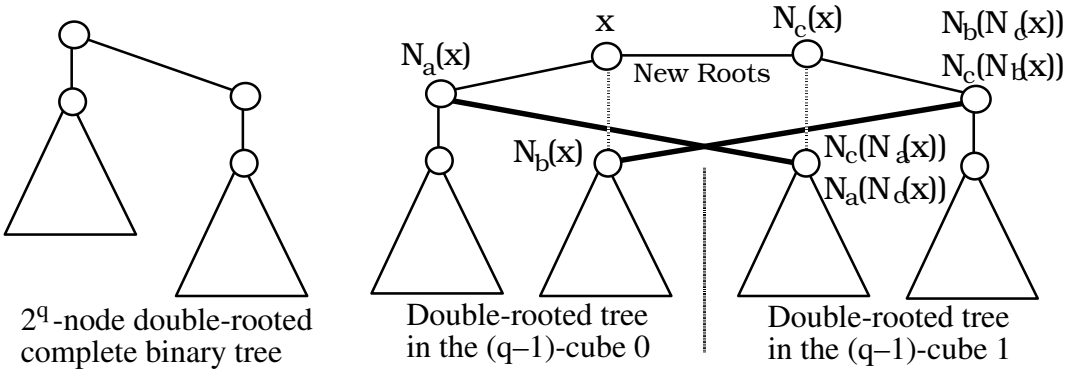


Fig. 13.6. The 2^q-node double-rooted complete binary tree is a subgraph of the q-cube.

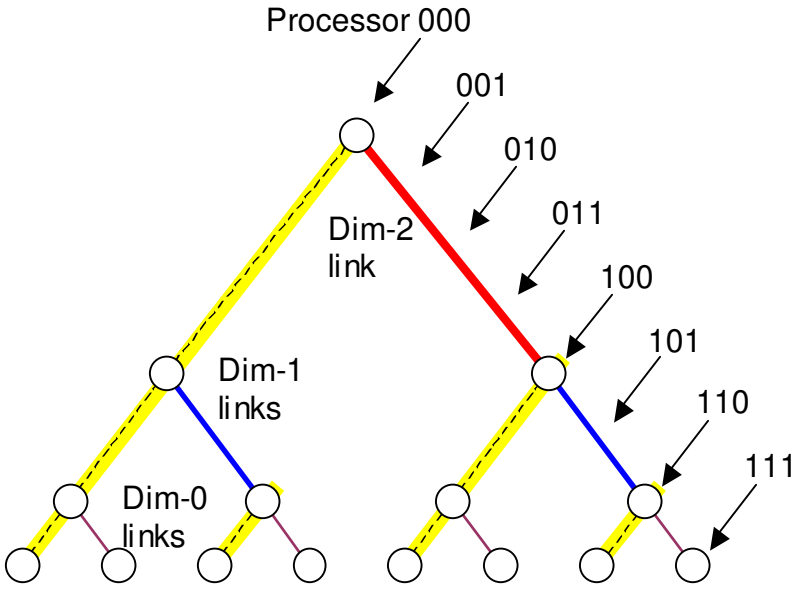


Fig. 13.7. Embedding a 15-node complete binary tree into the 3-cube.

13.4 A Few Simple Algorithms

Semigroup computation on the q-cube
Processor x, $0 \le x < p$ do $t[x] := v[x]$
{initialize "total" to own value}
for $k = 0$ to $q - 1$ Processor x, $0 \le x < p$, do
get $y := t[N_k(x)]$
set $t[x] := t[x] \otimes y$
endfor

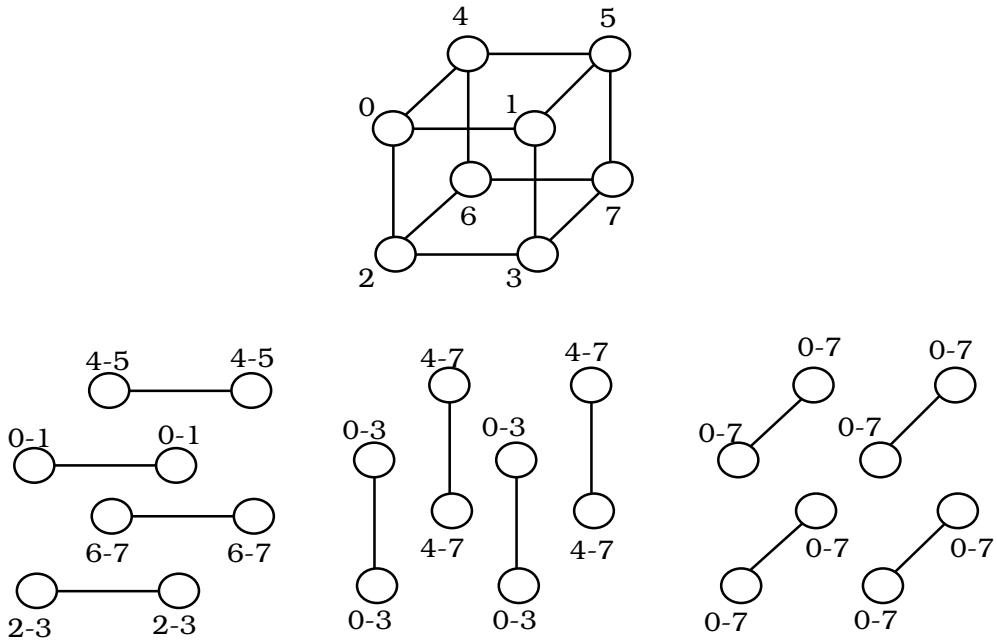


Fig. 13.8. Semigroup computation on a 3-cube.

Commutativity of the operator $\otimes$ is implicit in this algorithm

How to remove this assumption?

Parallel prefix computation on the q-cube

Processor x, $0 \le x < p$, do $t[x] := u[x] := v[x]$
    {initialize subcube "total" and partial prefix to own value}
for $k = 0$ to $q - 1$ Processor x, $0 \le x < p$, do
    get $y := t[N_k(x)]$
    set $t[x] := t[x] \otimes y$
    if $x > N_k(x)$ then set $u[x] := y \otimes u[x]$
endfor

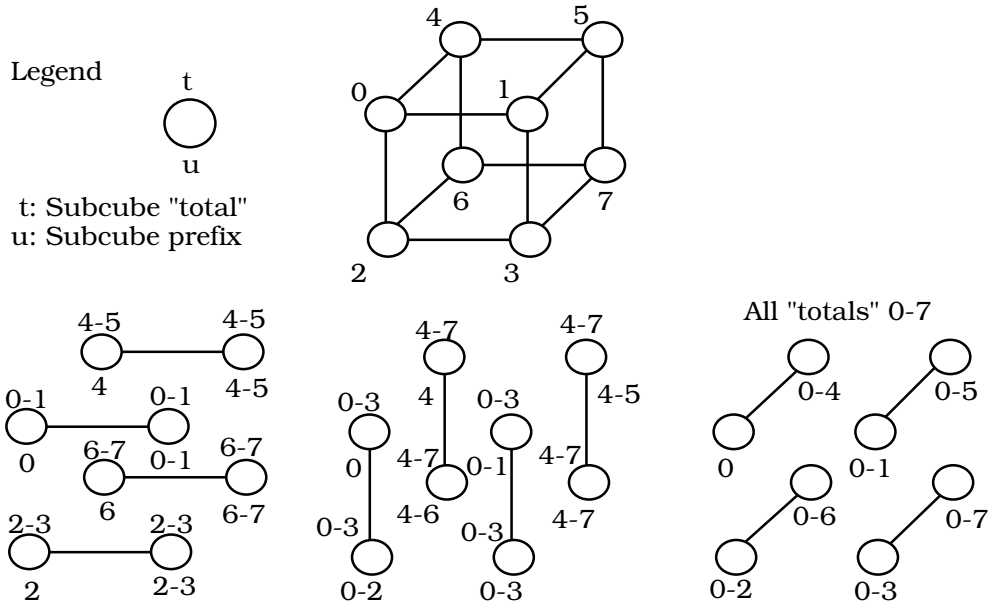


Fig. 13.9. Parallel prefix computation on a 3-cube.

Again, commutativity of $\otimes$ is implicit in this algorithm

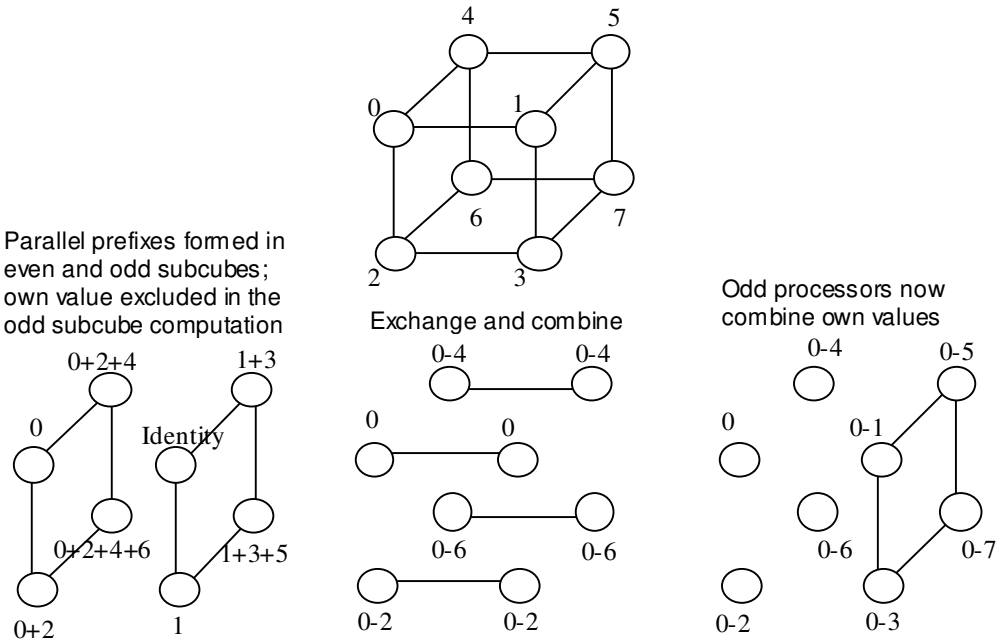
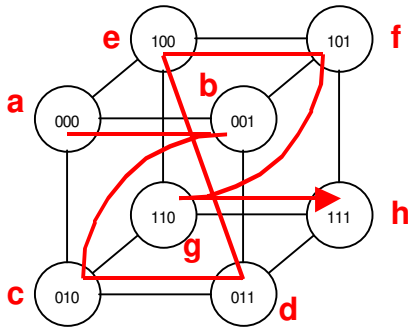


Fig. 13.10. A second algorithm for parallel prefix computation on a 3-cube.

Reversing a sequence on the q-cube

for $k = 0$ to $q - 1$ Processor x, $0 \leq x < p$, do
    get $y := v[N_k(x)]$
    set $v[x] := y$
endfor



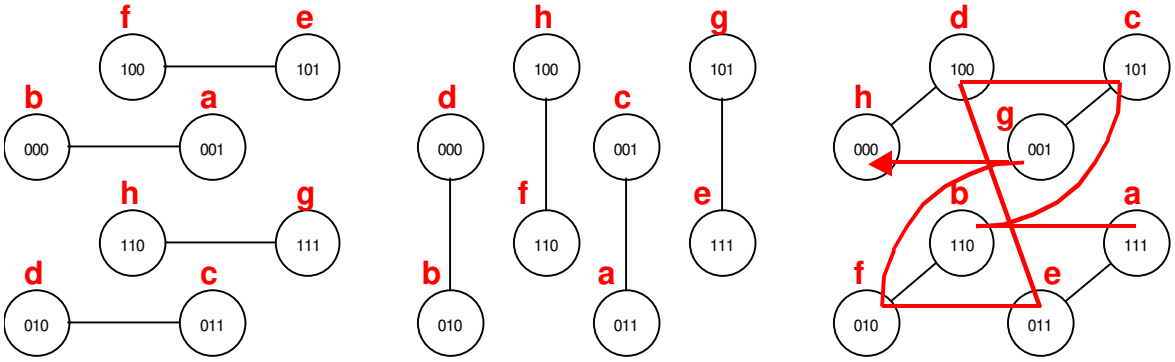
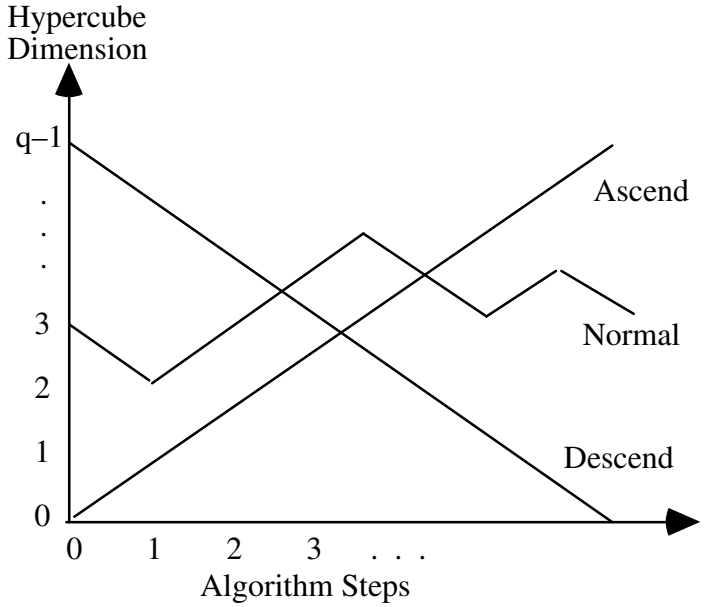


Fig. 13.11. Sequence reversal on a 3-cube.

Ascend, descend, and normal algorithms



13.5 Matrix Multiplication

Multiplying $m \times m$ matrices ($C = A \times B$) on a q-cube,
where $m = 2^{q/3}$ and $p = m^3$

Processor $(0,j,k)$ begins with A_{jk} & B_{jk} in registers R_A & R_B
and ends with element C_{jk} in register R_C

Multiplying $m \times m$ matrices on a q-cube, with $q = 3 \log_2 m$
for $l = q/3 - 1$ downto 0 Processor $x = ijk$, $0 \le i, j, k < m$, do
    if bit l of i is 1
    then get $y := R_A[N_{l \pm 2q/3}(x)]$ and $z := R_B[N_{l \pm 2q/3}(x)]$
        set $R_A[x] := y$; $R_B[x] := z$
    endif
endfor
for $l = q/3 - 1$ downto 0 Processor $x = ijk$, $0 \le i, j, k < m$, do
    if bit l of i and k are different
    then get $y := R_A[N_l(x)]$; set $R_A[x] := y$
    endif
endfor
for $l = q/3 - 1$ downto 0 Processor $x = ijk$, $0 \le i, j, k < m$, do
    if bit l of i and j are different
    then get $y := R_B[N_{l \pm q/3}(x)]$; set $R_B[x] := y$
    endif
endfor
Processor x, $0 \le x < p$, do $R_C := R_A \times R_B$
    {$p = m^3 = 2^q$ parallel multiplications in one step}
for $l = 0$ to $q/3 - 1$ Processor $x = ijk$, $0 \le i, j, k < m$, do
    if bit l of i is 0
    then get $y := R_C[N_{l \pm 2q/3}(x)]$; set $R_C[x] := R_C[x] + y$
    endif
endfor

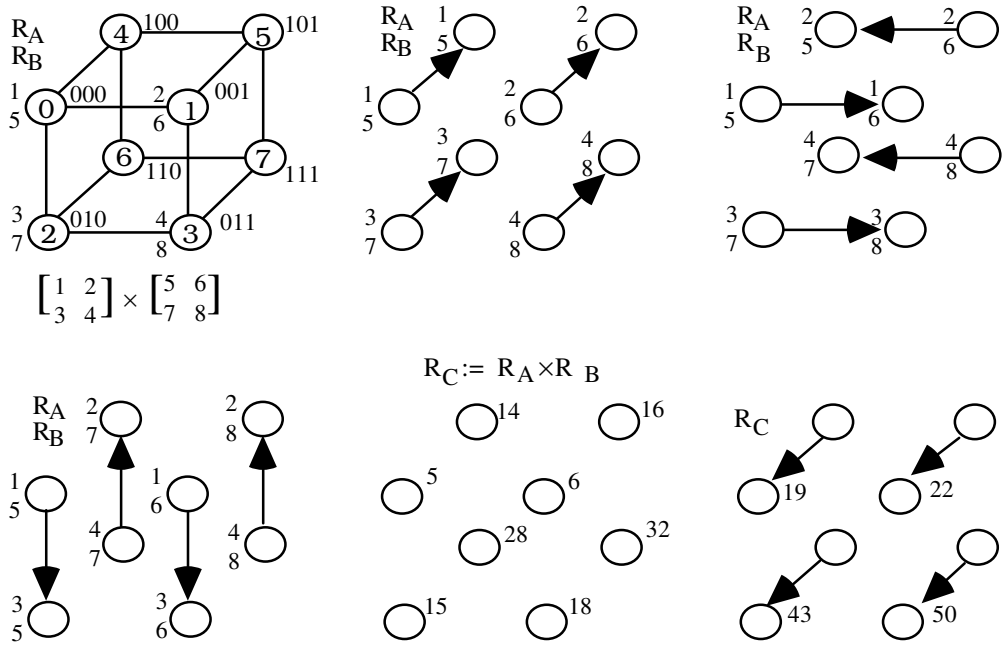


Fig. 13.12. Multiplying two 2×2 matrices on a 3-cube.

Running time of the preceding algorithm: $O(q) = O(\log p)$

Analysis in the case of block matrix multiplication:

The $m \times m$ matrices are partitioned into $p^{1/3} \times p^{1/3}$ blocks of size $(m/p^{1/3}) \times (m/p^{1/3})$

Each communication step involves $m^2/p^{2/3}$ block elements

Each multiplication involves $2m^3/p$ arithmetic operations

$$T_{\text{mul}}(m, p) = \underbrace{m^2/p^{2/3} \times O(\log p)}_{\text{Communication}} + \underbrace{2m^3/p}_{\text{Computation}}$$

13.6 Inverting a Lower Triangular Matrix

For $A = \begin{bmatrix} B & 0 \\ C & D \end{bmatrix}$ we have $A^{-1} = \begin{bmatrix} B^{-1} & 0 \\ -D^{-1}CB^{-1} & D^{-1} \end{bmatrix}$

If B and D are inverted in parallel by independent subcubes, the algorithm's running time is given by:

$$\begin{aligned} T_{inv}(m) &= T_{inv}(m/2) + 2T_{mul}(m/2) \\ &= T_{inv}(m/2) + O(\log m) = O(\log^2 m) \end{aligned}$$

14 Sorting and Routing on Hypercubes

Back to TOC

Chapter Goals

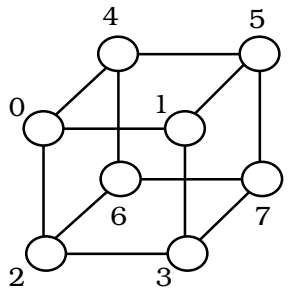
- Present hypercube sorting algorithms, showing perfect fit to bitonic sorting
- Derive hypercube routing algorithms, utilizing elegant recursive methods
- Learn about inherent limitations of oblivious routing schemes

Chapter Contents

- 14.1. Defining the Sorting Problem
- 14.2. Bitonic Sorting on a Hypercube
- 14.3. Routing Problems on a Hypercube
- 14.4. Dimension-Order Routing
- 14.5. Broadcasting on a Hypercube
- 14.6. Adaptive and Fault-Tolerant Routing

14.1. Defining the Sorting Problem

Arrange data in order of processor ID numbers (labels)



The ideal parallel sorting algorithm

$T(p) = \Theta((n \log n)/p)$

We cannot achieve this optimal time for all n and p

1-1 sorting ($n = p$)

Batcher's bitonic sort: $O(\log^2 n) = O(\log^2 p)$ time

Same for Batcher's odd-even merge sort

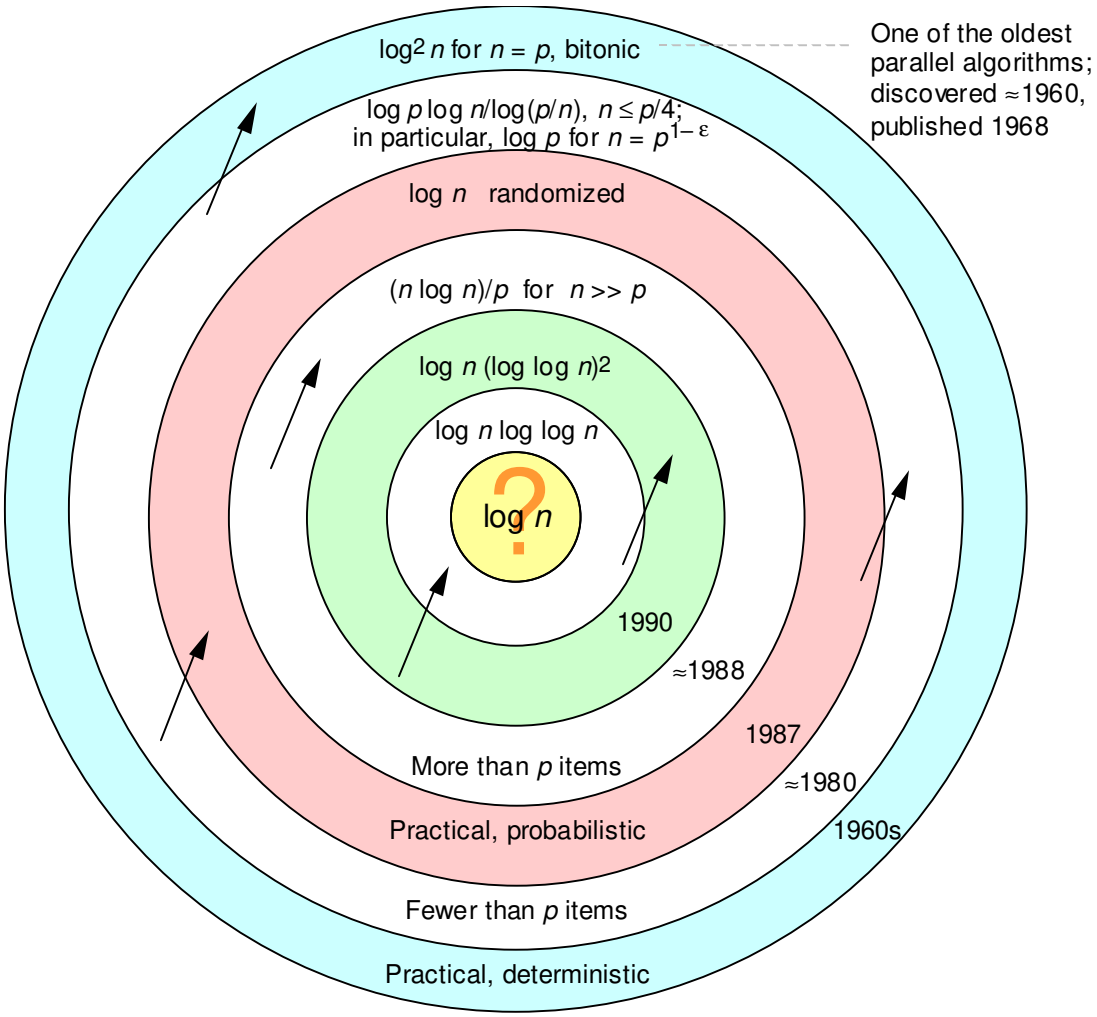
$O(\log n)$-time deterministic algorithm not known

k-k sorting ($n = pk$)

Optimal algorithms known for $n >> p$ or when

average running time is considered (randomized)

Attempts and progress in hypercube sorting algorithms



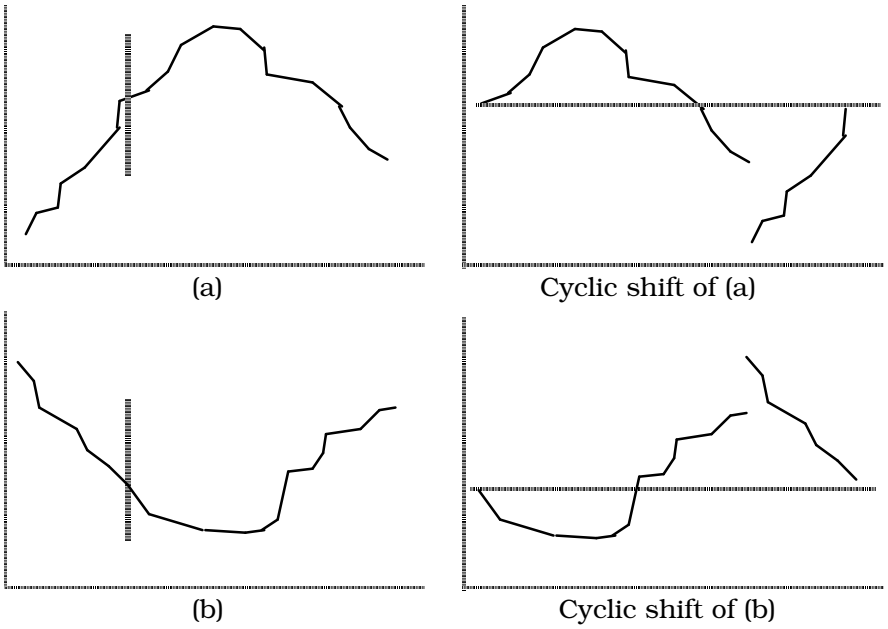
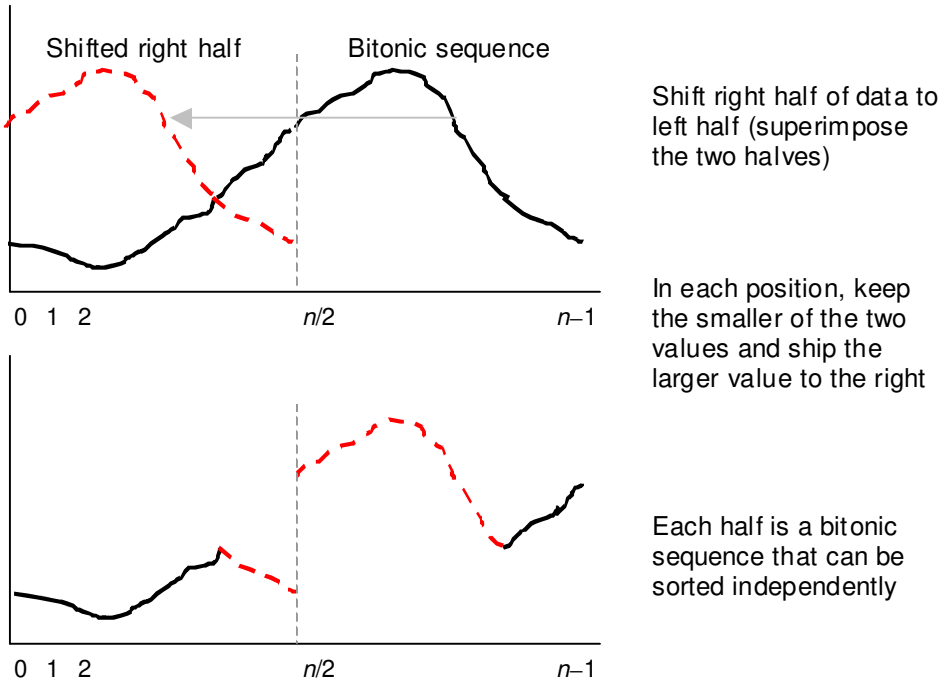


Fig. 14.1. Examples of bitonic sequences.



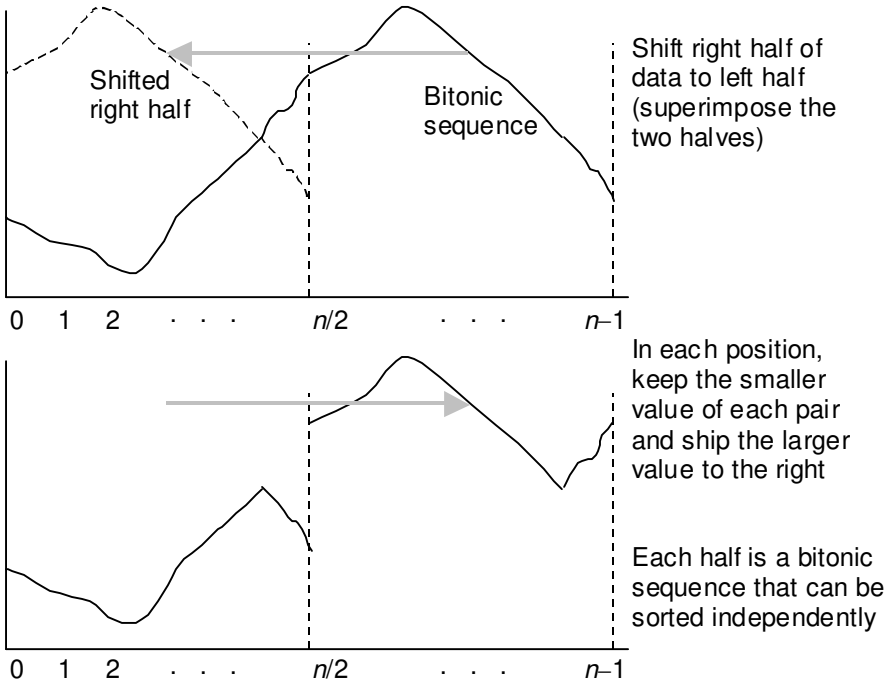


Fig. 14.2. Sorting a bitonic sequence on a linear array.

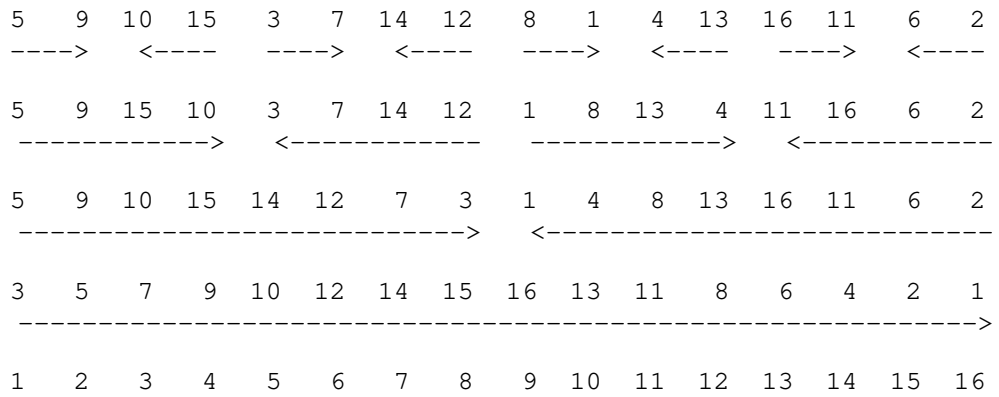


Fig. 14.3. Sorting an arbitrary sequence on a linear array through recursive application of bitonic sorting.

$$T(p) = T(p/2) + B(p)$$
$$= T(p/2) + 2p - 2 = 4p - 4 - 2\log_2 p$$

Alternate derivation for the running time of bitonic sorting on a linear array:

$$T(p) = B(2) + B(4) + \cdots + B(p)$$

$$= 2 + 6 + 14 + \cdots + (2p - 2) = 4p - 4 - 2\log_2 p$$

For linear array, the bitonic sorting algorithm is inferior to simpler odd-even transposition sort which requires only p compare-exchanges or $2p$ unidirectional communications

However, the situation is quite different for a hypercube

14.2 Bitonic Sorting on a Hypercube

Sort lower ($x_{q-1} = 0$) and upper ($x_{q-1} = 1$) subcubes

in opposite directions; yields a bitonic sequence

Shifting the halves takes one compare-exchange step

$$B(q) = B(q-1) + 1 = q$$

Sorting a bitonic sequence of size n on q-cube, $q = \log_2 n$

```
for l = q – 1 downto 0 Processor x, 0 ≤ x < p, do
    if x_l = 0
    then get y := v[N_l(x)]; keep min(v(x), y);
        send max(v(x), y) to N_l(x)
    endif
endfor
```

Bitonic sorting algorithm

$$T(q) = T(q-1) + B(q) = T(q-1) + q$$
$$= q(q+1)/2 = \log_2 p\,(\log_2 p + 1)/2$$

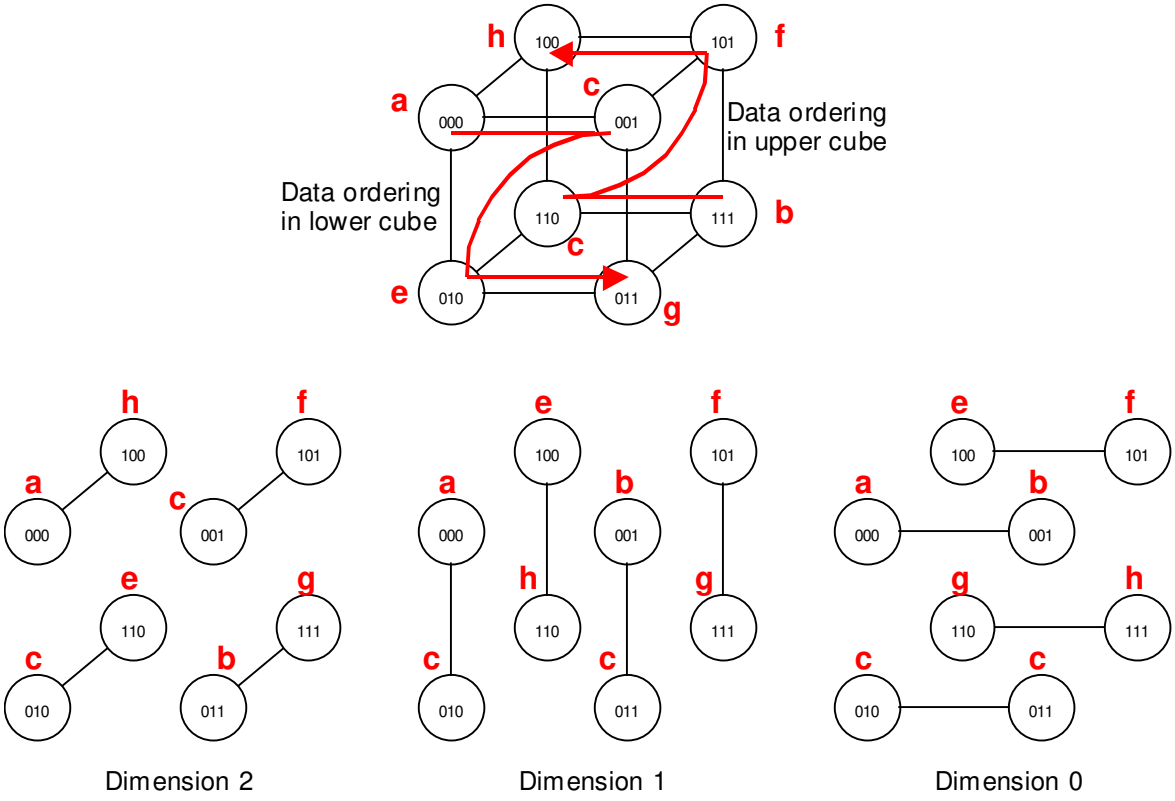


Fig. 14.4. Sorting a bitonic sequence of size 8 on the 3-cube.

14.3 Routing Problems on a Hypercube

Types of routing algorithms

Oblivious: path uniquely determined by node addresses

Nonoblivious or adaptive: the path taken by a message may also depend on other messages in the network

On-line: make the routing decisions on the fly as you route

Off-line: route selections are precomputed for each problem of interest and stored within nodes (routing tables)

Positive result for off-line routing on a p-node hypercube

Any 1-1 routing problem with p or fewer packets can be solved in O(log p) steps, using an off-line algorithm

The off-line algorithm chooses routes in a way that the route taken by one message does not significantly overlap or conflict with those of others (for each source/destination pair, there are many paths to choose from)

Negative result for oblivious routing on any network

Theorem 14.1: Let $G = (V, E)$ be a p-node, degree-d network. Any oblivious routing algorithm for routing p packets in G needs $\Omega(\sqrt{p} / d)$ worst-case time

For a hypercube: oblivious routing requires $\Omega(\sqrt{p} / \log p)$ time in the worst case (only slightly better than mesh)

In most instances, actual routing performance is much closer to the log-time best case than to the worst case.

Proof Sketch for Theorem 14.1

Let $P_{u,v}$ be the unique path used for routing from u to v

There are $p(p-1)$ paths for routing among all node pairs

These paths are predetermined and independent of other traffic within the network

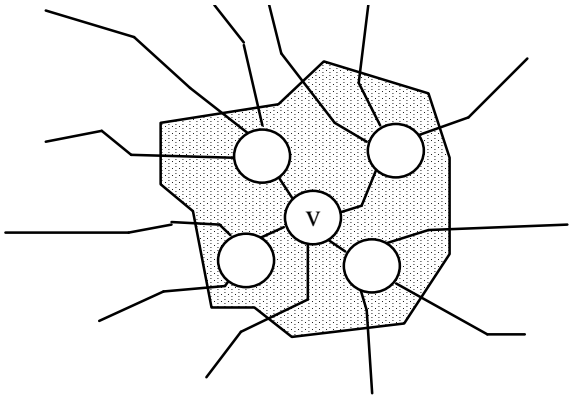
Our strategy: find k node pairs u_i, v_i $(1 \le i \le k)$ such that

$u_i \ne u_j$ and $v_i \ne v_j$ for $i \ne j$, and

$P_{ui,vi}$ all pass through the same edge e

Because ≤ 2 packets can go through a link in one step, $\Omega(k)$ steps will be needed for some 1-1 routing problem

The main part of the proof consists of showing that k can be as large as $\sqrt{p}/d$



14.4 Dimension-Order Routing

Route from node 01011011
to node 11010110
^ ^^ ^ Dimensions that differ

Path: 01011011, 11011011, 11010011,
11010111, 11010110

Unfolded hypercube (indirect cube, butterfly network) facilitates the discussion of routing algorithms

Dimension-order routing between nodes *i* and *j* in a hypercube can be viewed as routing from node *i* in column 0 (*q*) to node *j* in column *q* (0) of the butterfly

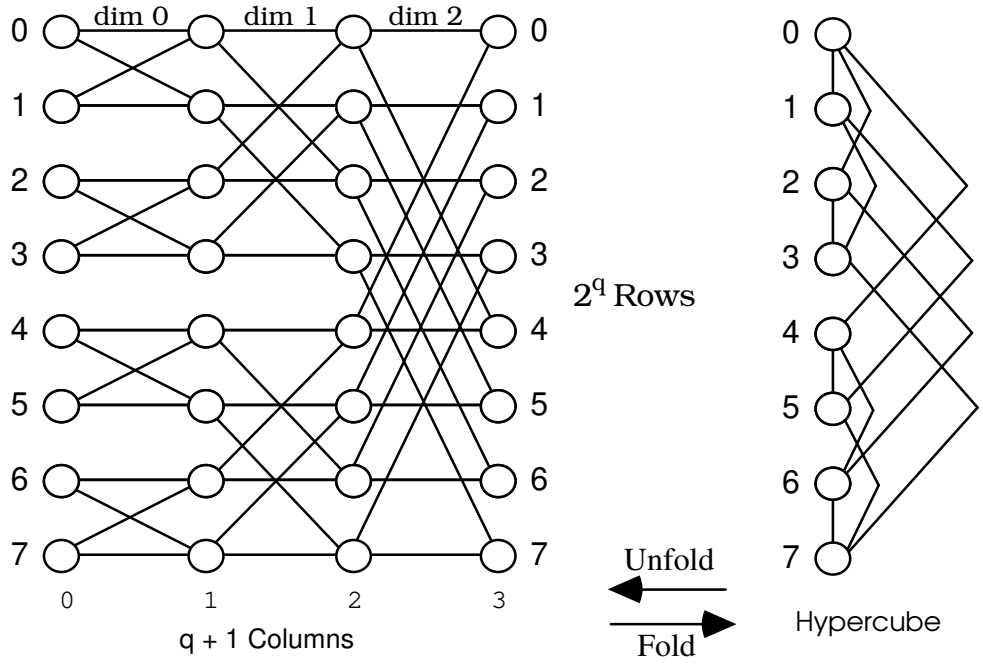


Fig. 14.5. Unfolded 3-cube or the 32-node butterfly network.

Self-routing in a butterfly

From node 3 to node 6: routing tag = 011 ⊕ 110 = 101
(this indicates the "cross-straight-cross" path)

From node 6 to node 1: routing tag 110 ⊕ 001 = 111
(this represents a "cross-cross-cross" path)

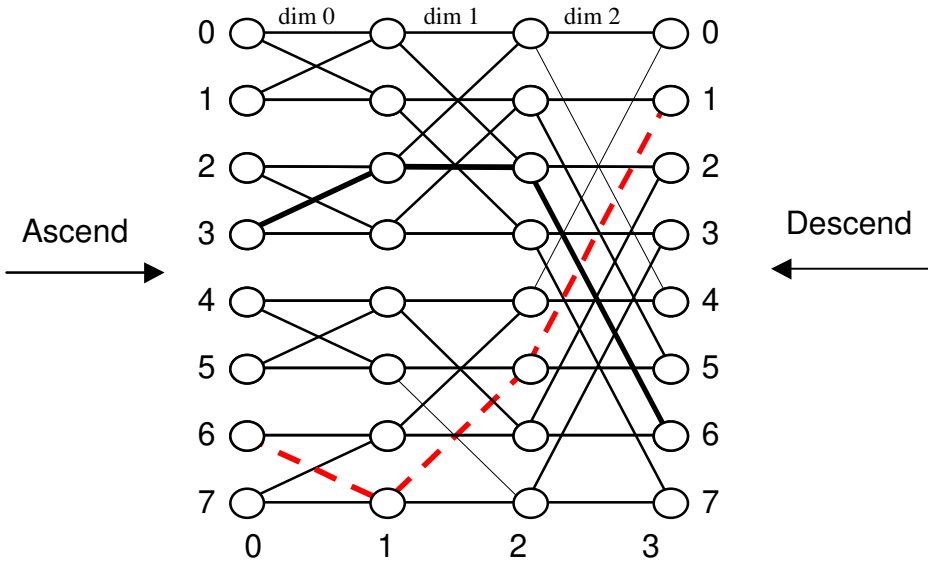


Fig. 14.6. Example dimension-order routing paths.

The butterfly network cannot route all permutations without node or edge conflicts; e.g., any permutation involving the routes (1, 7) and (0, 3) leads to a conflict

The extent of conflicts depends on the routing problem

There exist "good" routing problems for which conflicts are non-existent or rare

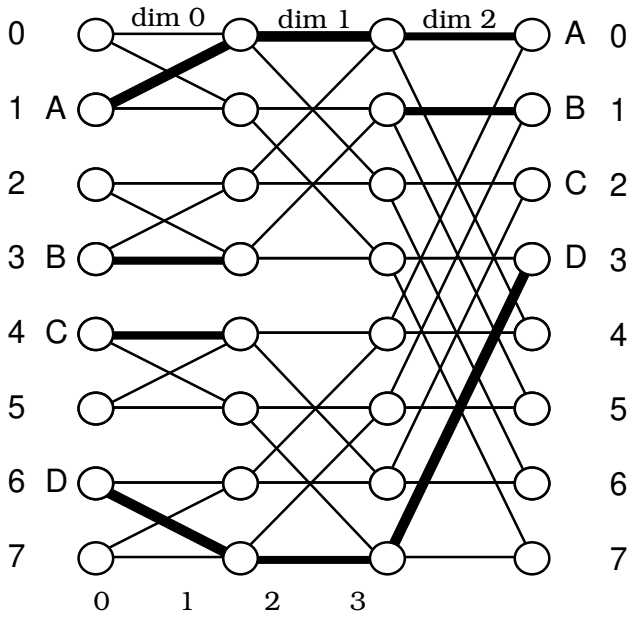


Fig. 14.7. Packing is a "good" routing problem for dimension-order routing on the hypercube.

There are also "bad" routing problems that lead to maximum conflicts and thus the worst-case running time predicted by Theorem 14.1

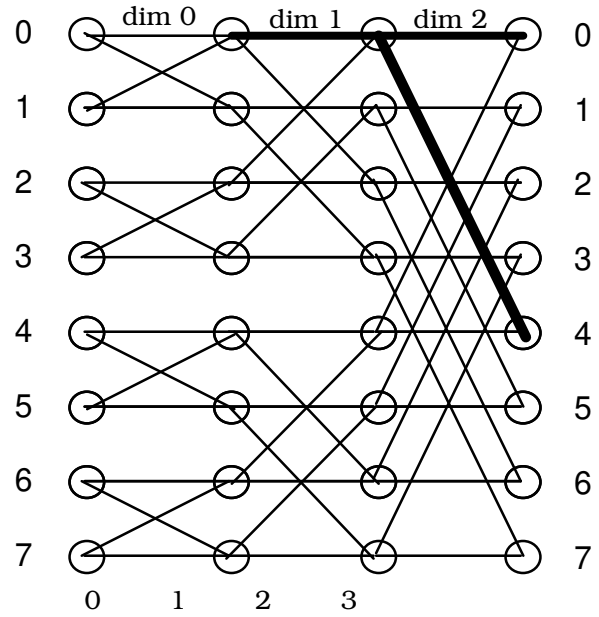


Fig. 14.8. Bit-reversal permutation is a "bad" routing problem for dimension-order routing on the hypercube.

Message buffer needs of dimension-order routing

True or false: if we limit nodes to a constant number of message buffers, then the above bound still holds, except that messages are queued at several levels before reaching node 0

False: queuing messages at multiple intermediate nodes introduces added delays that we have not accounted for, so that even the $\Theta(\sqrt{p})$ running time is not guaranteed

Bad news: if each node of the hypercube is limited to O(1) buffers, there exist permutation routing problems that require O(p) time; i.e., as bad as on a linear array!

Good news: the performance is usually much better; i.e., $\log_2 p$ + o(log p) for most permutations. The average running time of dimension-order routing is very close to its best case and message buffer requirements are modest

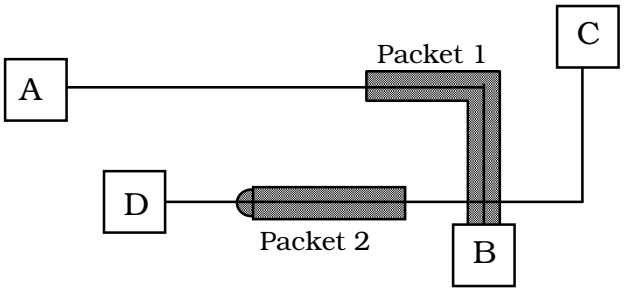
If we anticipate encountering (near) worst-case routing patterns in an application, two options are available to us:

Compute the routing paths off-line and store in tables

Use randomized routing to convert the worst-case to average-case performance

Probabilistic analyses for showing the good average-case performance of dimension-order routing are complicated

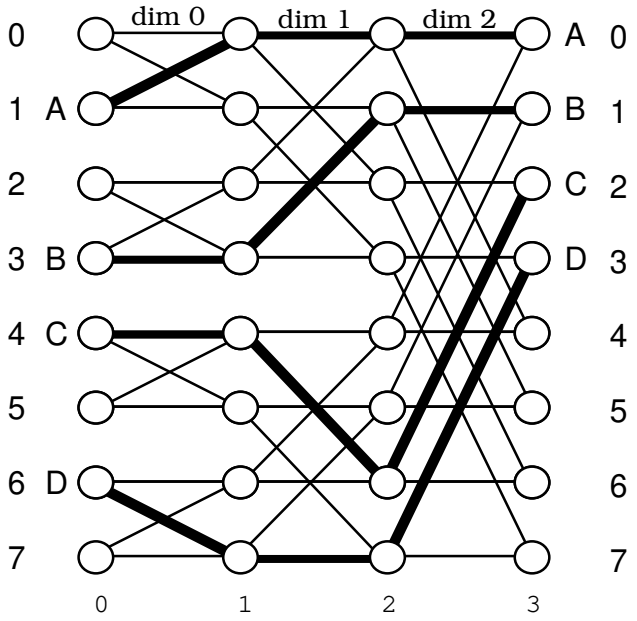
Wormhole routing on a hypercube



Some of the preceding results are directly applicable here

Any good routing problem, yielding node- and edge-disjoint paths, will remain good for wormhole routing

In Fig. 14.7, the four worms carrying messages *A*, *B*, *C*, *D*, will move with no conflict among them. Each message is delivered to its destination in the shortest possible time, regardless of the length of the worms



For bad routing problems, on the other hand, wormhole routing aggravates the difficulties, given that one message can now tie up a number of nodes and links

In the case of wormhole routing, one also needs to be concerned with deadlocks resulting from circular waiting of messages for one another

Dimension-order routing is always deadlock-free

With hot-potato or deflection routing, which is attractive for reducing the message buffering requirements, dimension orders are occasionally modified or more than one routing step along some dimensions may be allowed

Deadlock considerations in this case are similar to those of other adaptive routing schemes (see Section 14.6)

14.5 Broadcasting on a Hypercube

Simple "flooding" scheme with all-port communication

00000	Source node
00001, 00010, 00100, 01000, 10000	Neighbors of source
00011, 00101, 01001, 10001, 00110, 01010, 10010, 01100, 10100, 11000	Distance-2 nodes
00111, 01011, 10011, 01101, 10101, 11001, 01110, 10110, 11010, 11100	Distance-3 nodes
01111, 10111, 11011, 11101, 11110	Distance-4 nodes
11111 Distance-5 node	

Binomial broadcast tree with single-port communication

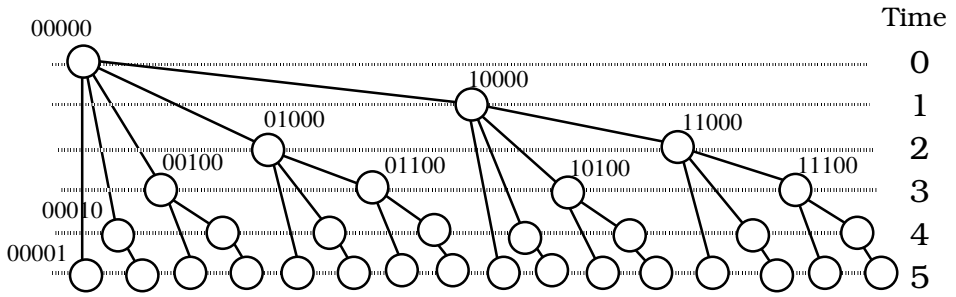


Fig. 14.9. The binomial broadcast tree for a 5-cube.

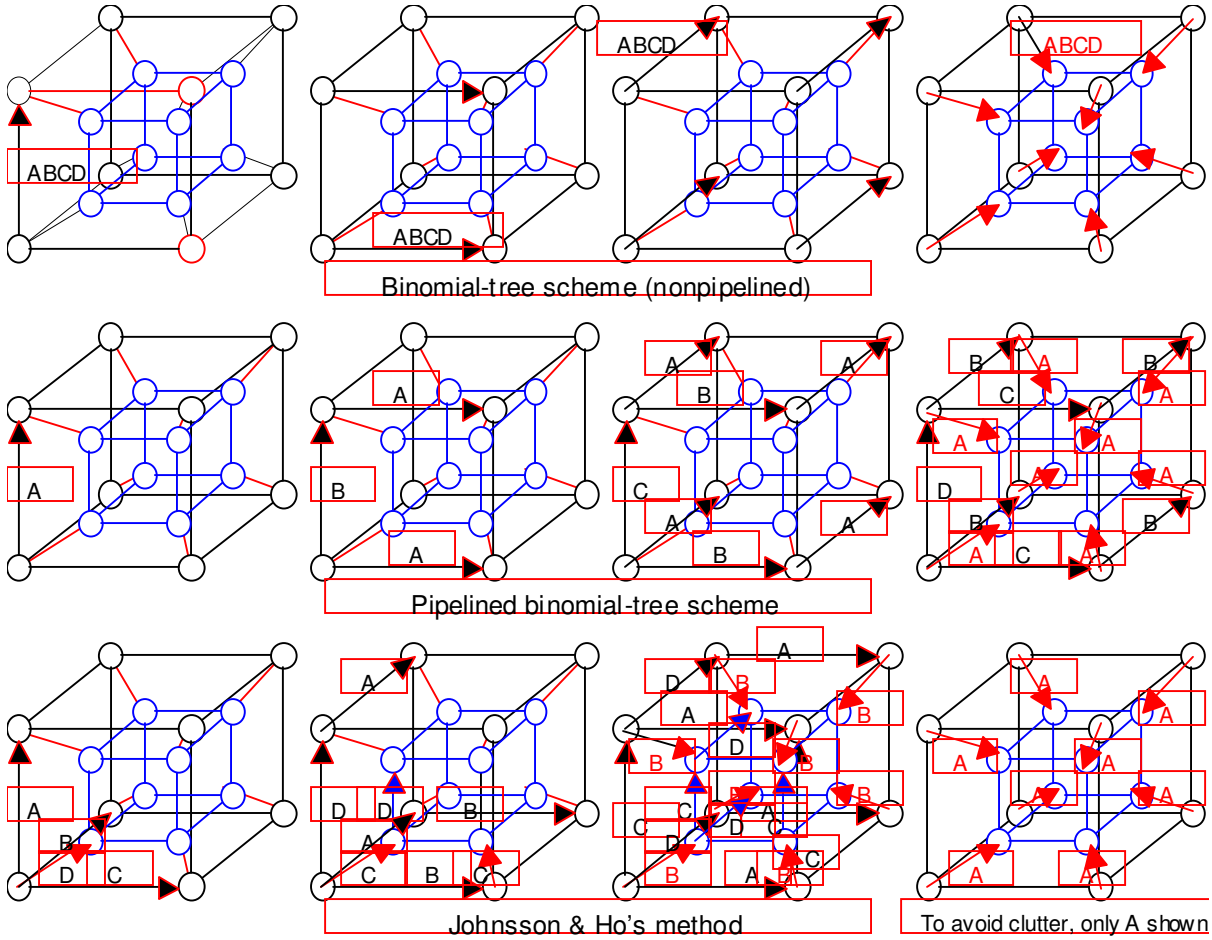


Fig. 14.10. Three hypercube broadcasting schemes as performed on a 4-cube.

14.6 Adaptive and Fault-Tolerant Routing

There are up to *q* node-disjoint and edge-disjoint shortest paths between any node pairs in a *q*-cube

Thus, one can route messages around congested or failed nodes/links

A useful notion for designing adaptive wormhole routing algorithms is that of virtual communication networks

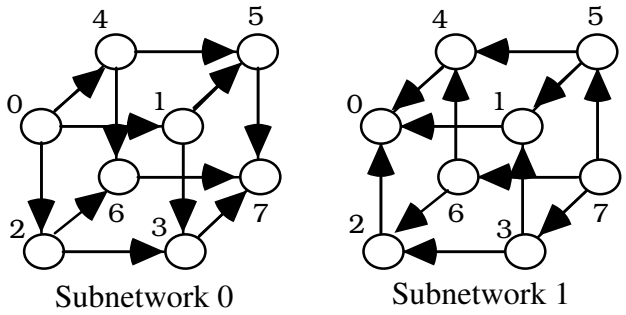


Fig. 14.11. Partitioning a 3-cube into subnetworks for deadlock-free routing.

Because each of the subnetworks in Fig. 14.11 is acyclic, any routing scheme that begins by using links in Subnet 0, at some point switches the routing path to Subnet 1, and from then on remains in Subnet 1, is deadlock-free

Fault diameter of q-cube is at most $q + 1$ with $\leq q - 1$ faults and at most $q + 2$ with $\leq 2q - 3$ faults [Lati93]

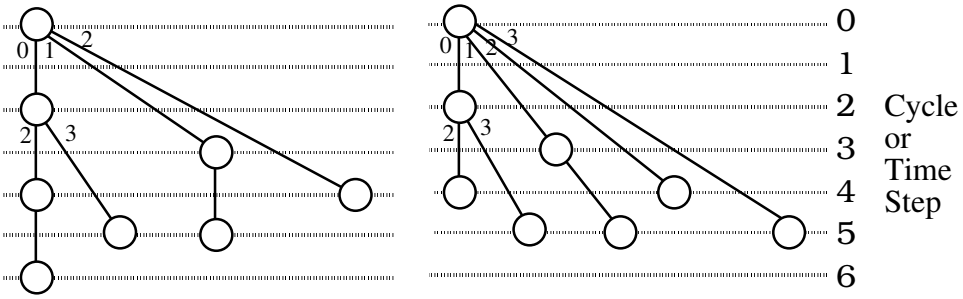


Figure for Problem 14.15.

15 Other Hypercubic Architectures

Back to TOC

Chapter Goals

- Learn how the binary hypercube can be generalized to provide cost or performance benefits over the original version
- Derive algorithms for these architectures based on emulating a hypercube

Chapter Contents

- 15.1. Modified and Generalized Hypercubes
- 15.2. Butterfly and Permutation Networks
- 15.3. Plus-or-Minus-2^i Network
- 15.4. The Cube-Connected Cycles Network
- 15.5. Shuffle and Shuffle-Exchange Networks
- 15.6. That's Not All Folks!

15.1 Modified and Generalized Hypercubes

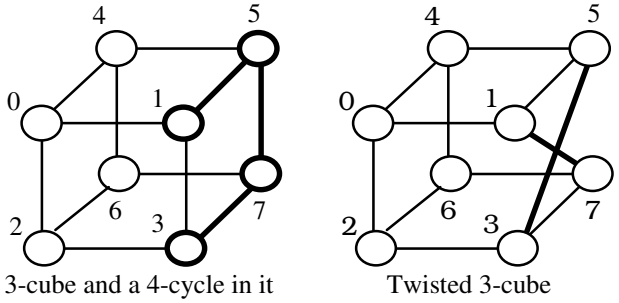


Fig. 15.1. Deriving a twisted 3-cube by redirecting two links in a 4-cycle.

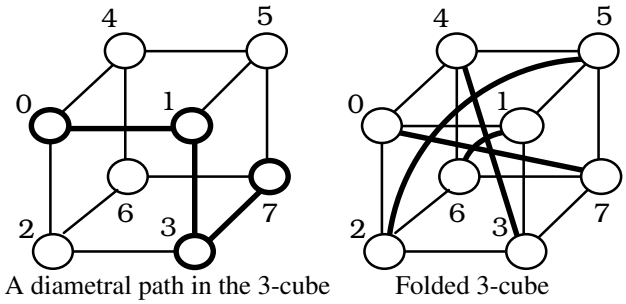


Fig. 15.2. Deriving a folded 3-cube by adding four diametral links.

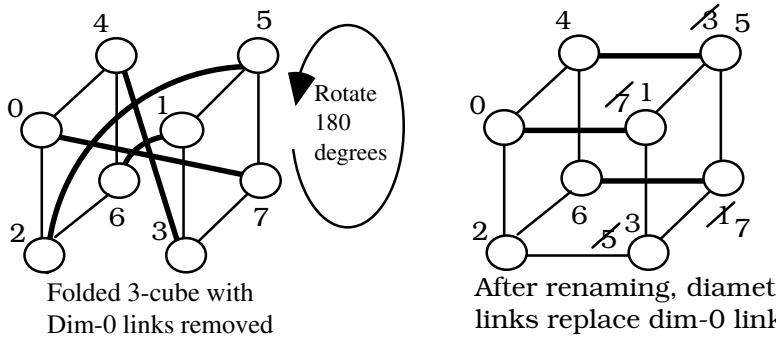


Fig. 15.3. Folded 3-cube viewed as 3-cube with a redundant dimension.

A hypercube is a power or homogeneous product network

q-cube = $(\circ\!\!-\!\!-\!\!\circ)^q$

q-cube = qth power of K_2

Generalized hypercube = qth power of K_r

(node labels are radix-r numbers)

Example: radix-4 generalized hypercube

Node labels are radix-4 numbers

Node x is connected to y iff x and y differ in one digit

Each node has $r - 1$ dimension-k links

15.2 Butterfly and Permutation Networks

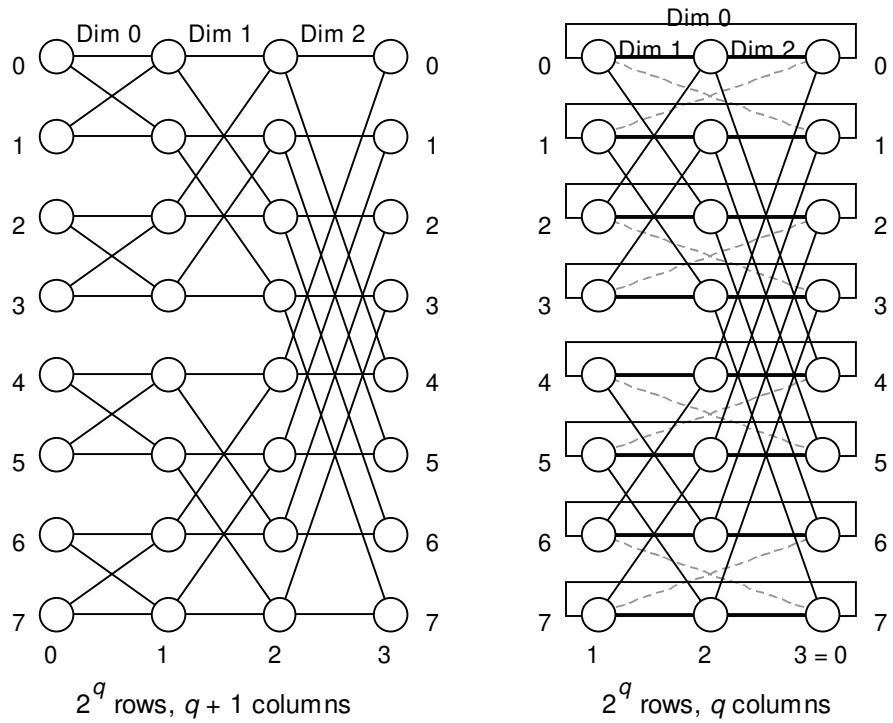
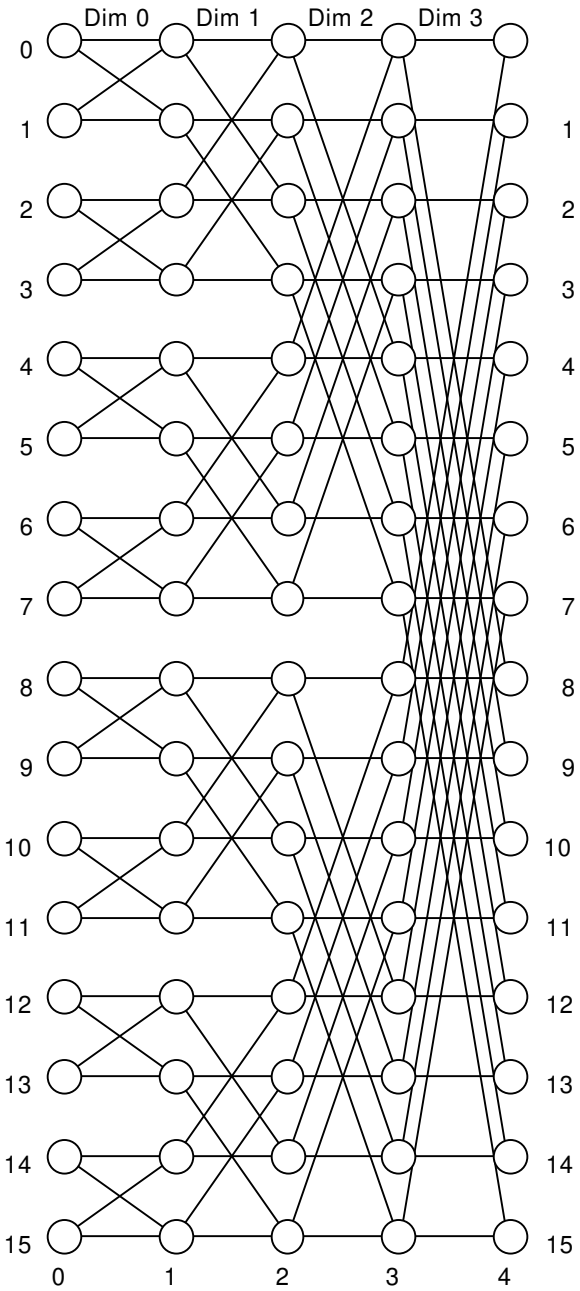


Fig. 15.4. Butterfly and wrapped butterfly networks.


Dim 0
Dim 1
Dim 2
Dim 3
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
1
2
3
4

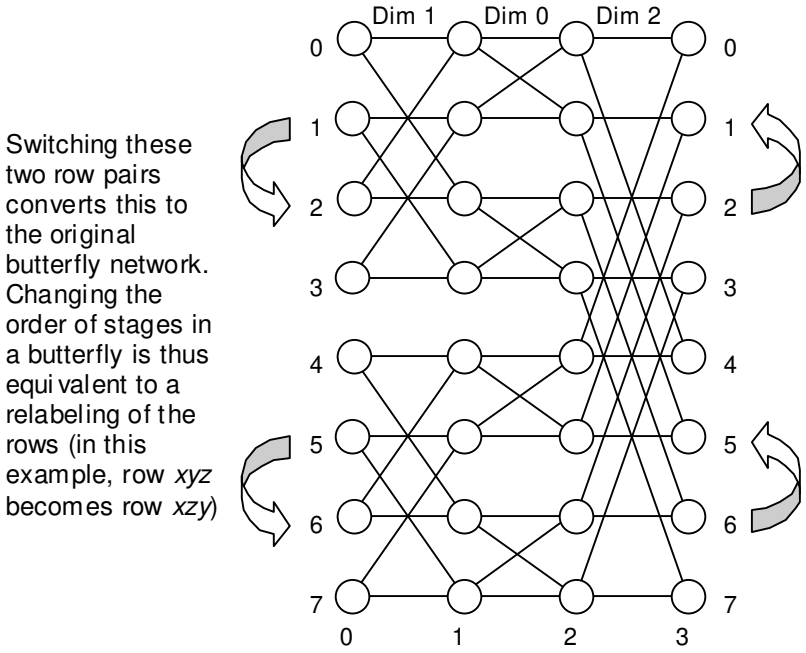


Fig. 15.5. Butterfly network with permuted dimensions.

Fat trees eliminate the bisection bottleneck of a "skinny" tree by making the bandwidth of links correspondingly higher near the root

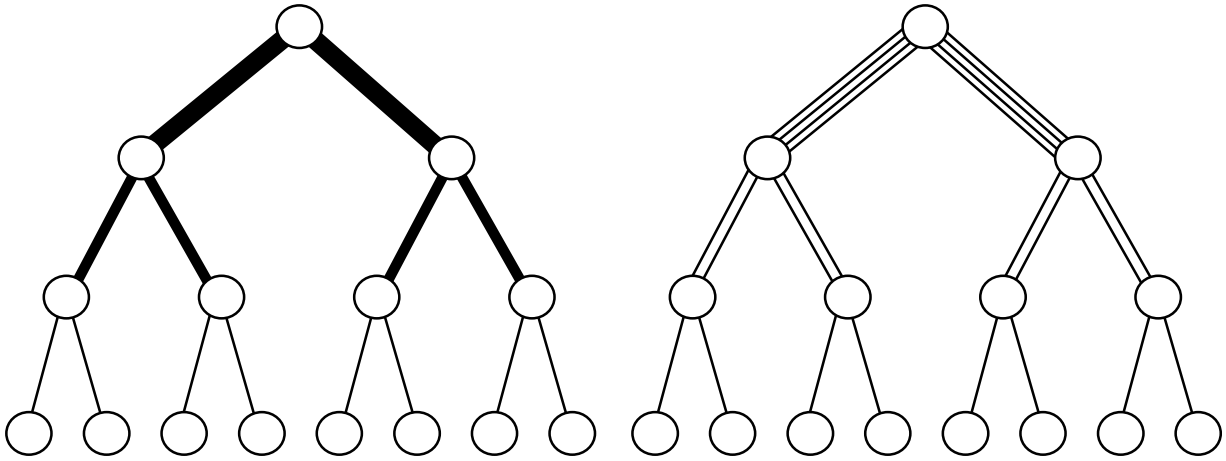


Fig. 15.6. Two representations of a fat tree.

One way of realizing a fat tree

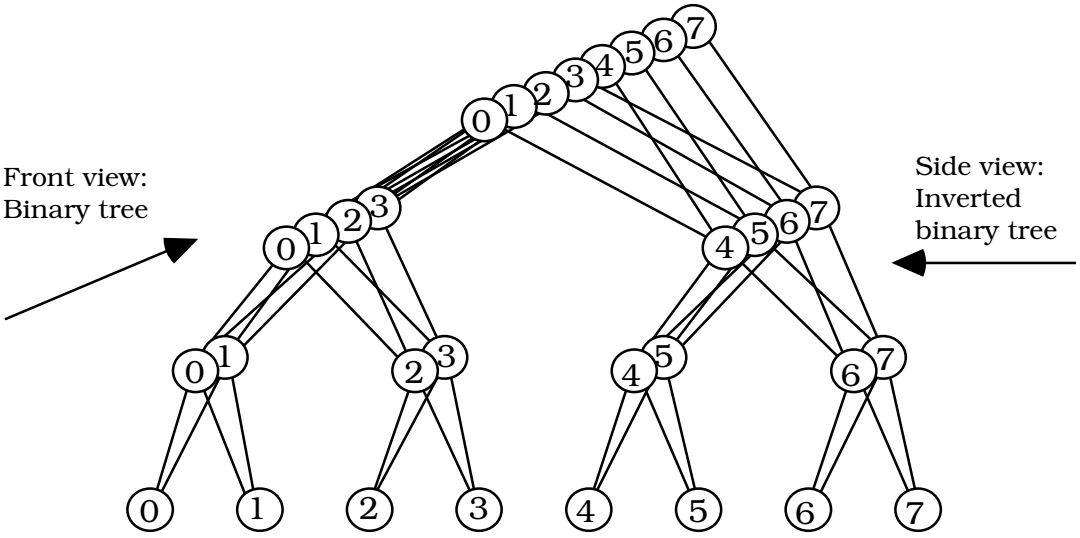


Fig. 15.7. Butterfly network redrawn as a fat tree.

Butterfly as a multistage interconnection network

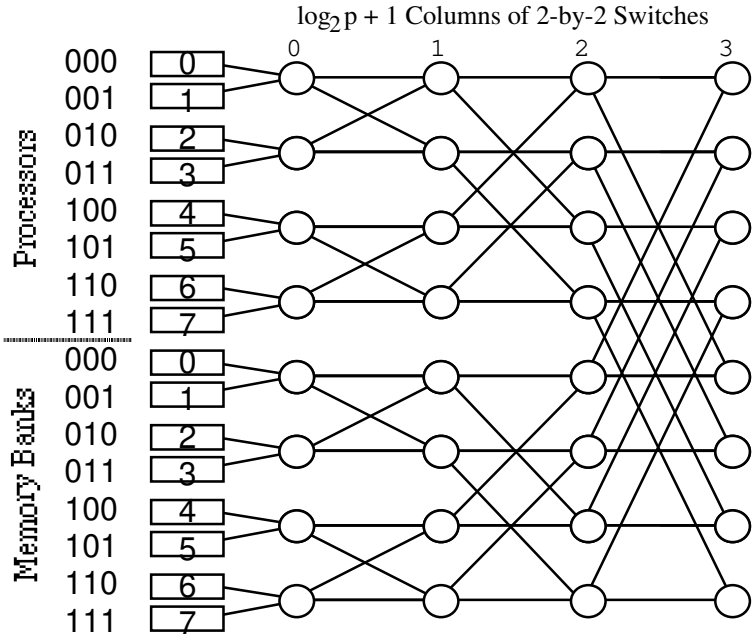


Fig. 15.8. Butterfly network used to connect modules that are on the same side.

Generalization of the butterfly network

High-radix or m-ary butterfly (built of $m \times m$ switches)

Has m^q rows and $q + 1$ columns (q if wrapped)

Beneš network can route any permutation

(it is rearrangeable)

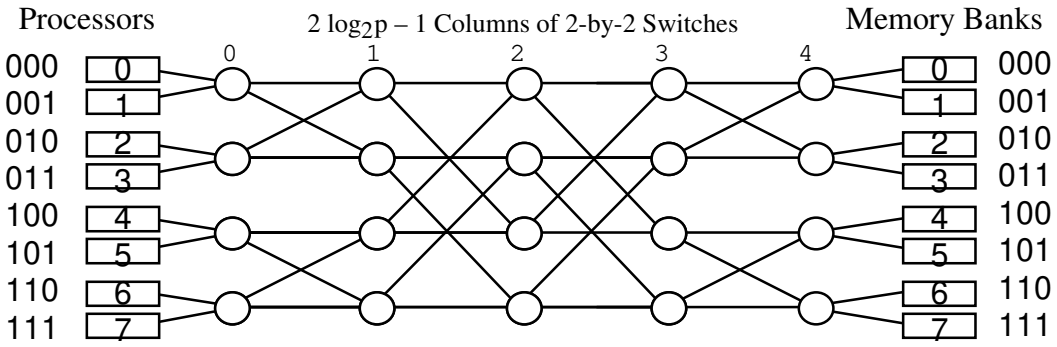


Fig. 15.9. Beneš network formed from two back-to-back butterflies.

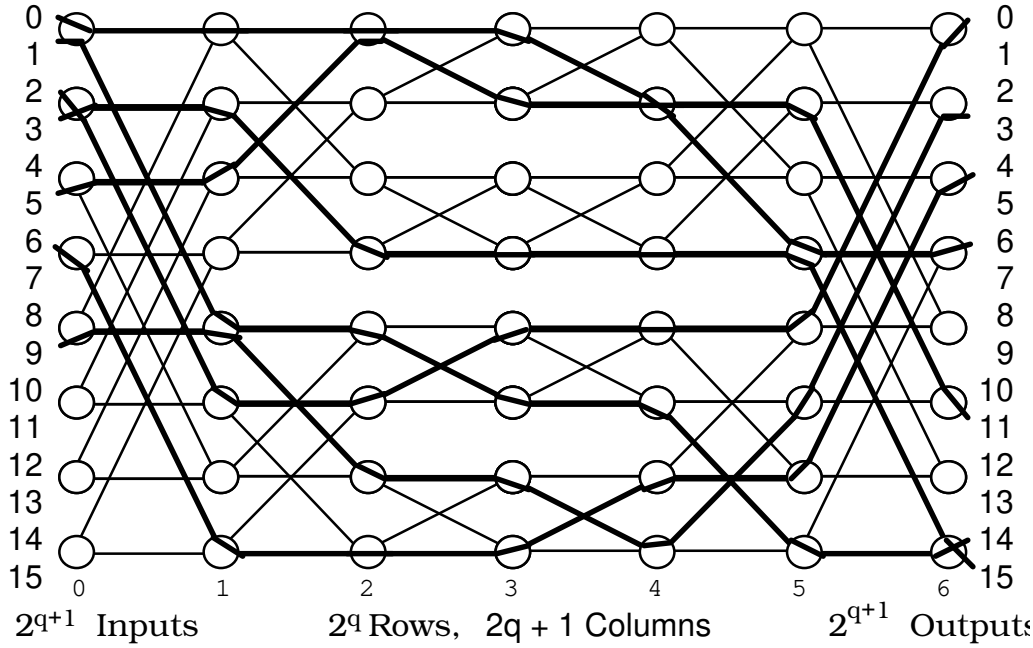


Fig. 15.10. Another example of a Beneš network.

15.3 Plus-or-Minus-2^i Network

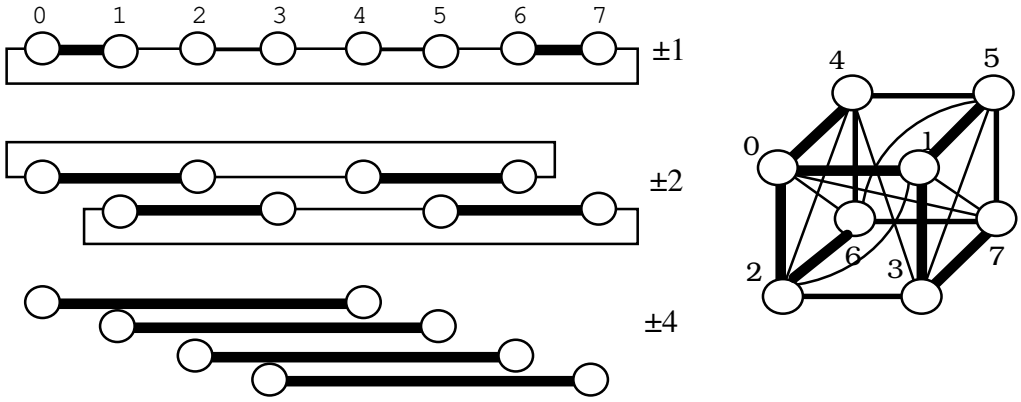


Fig. 15.11. Two representations of the eight-node PM2I network.

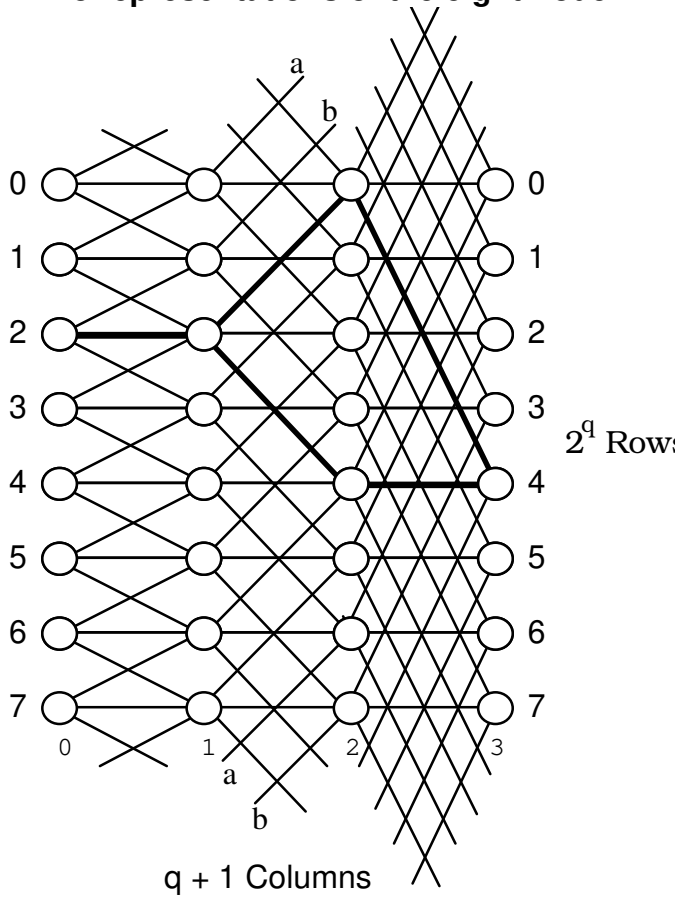


Fig. 15.12. Augmented data manipulator network.

15.4 The Cube-Connected Cycles Network

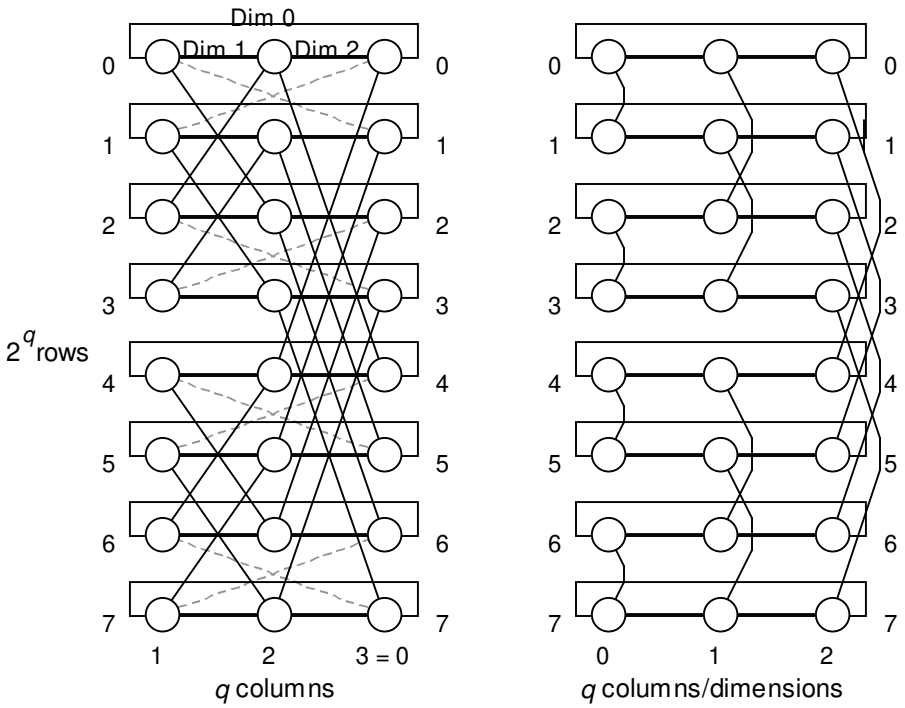


Fig. 15.13. A wrapped butterfly (left) converted into cube-connected cycles.

How CCC was originally defined:

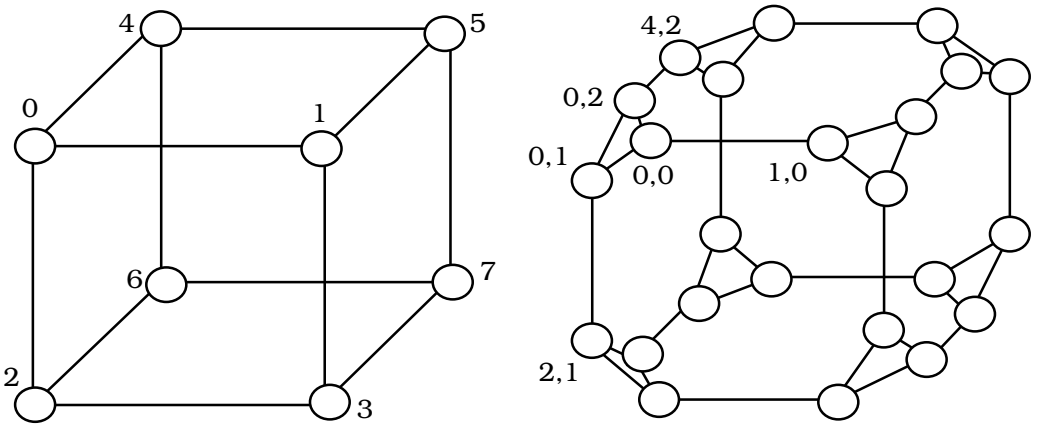
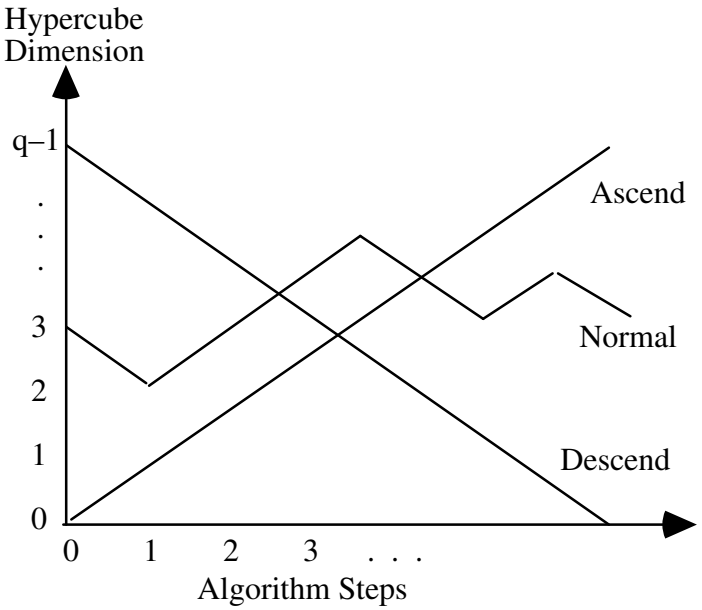


Fig. 15.14. Alternate derivation of CCC from a hypercube.

Emulating normal hypercube algorithms on CCC



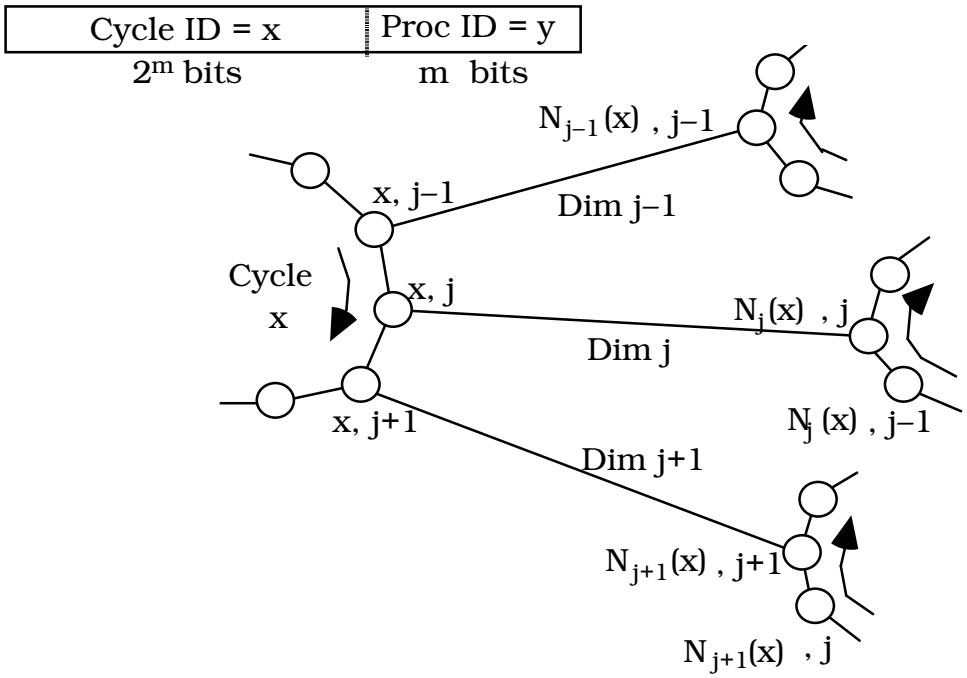


Fig. 15.15. CCC emulating a normal hypercube algorithm.

15.5 Shuffle and Shuffle-Exchange Networks

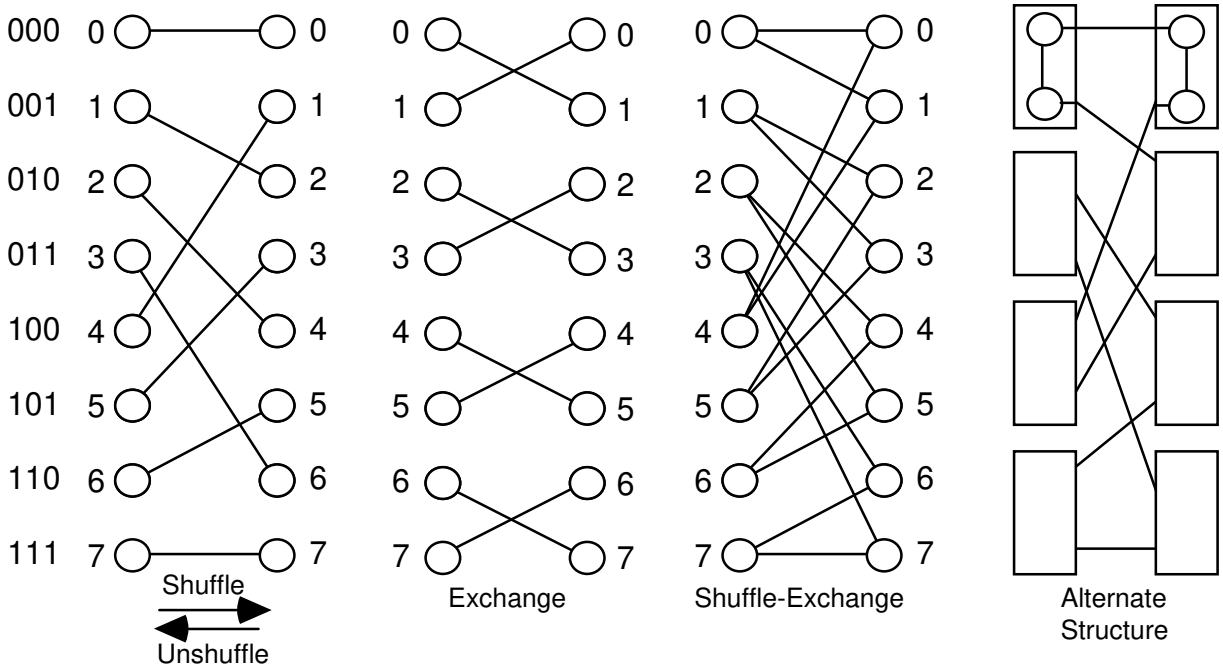


Fig. 15.16. Shuffle, exchange, and shuffle–exchange connectivities.

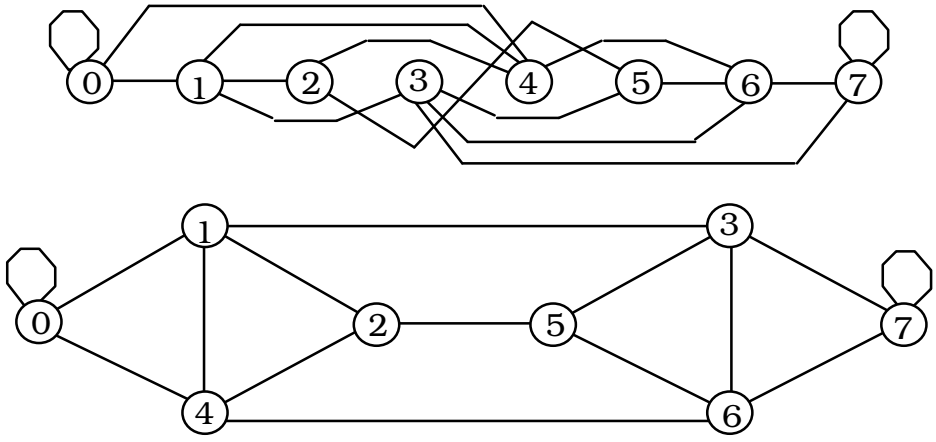


Fig. 15.17. Alternate views of an eight-node shuffle–exchange network.

In a 2^q-node shuffle network, node $x = x_{q-1}x_{q-2} \cdots x_2x_1x_0$

is connected to $x_{q-2} \cdots x_2x_1x_0x_{q-1}$ (cyclic left-shift of x)

In the shuffle-exchange network, node x is additionally connected to $x_{q-2} \cdots x_2x_1x_0 \ \bar{x}_{q-1}$

Routing in a shuffle-exchange network

```
Source                          01011011
Destination                     11010110
Positions that differ             ^   ^^ ^
Route  01011011  Shuffle to 10110110  Exchange to  10110111
       10110111  Shuffle to 01101111
       01101111  Shuffle to 11011110
       11011110  Shuffle to 10111101
       10111101  Shuffle to 01111011  Exchange to  01111010
       01111010  Shuffle to 11110100  Exchange to  11110101
       11110101  Shuffle to 11101011
       11101011  Shuffle to 11010111  Exchange to  11010110
```

For 2^q-node shuffle-exchange network:

$$D = q = \log_2 p, \qquad d = 4$$

With shuffle and exchange links provided separately, as shown in Fig. 15.18, the diameter increases to $2q - 1$ and node degree reduces to 3

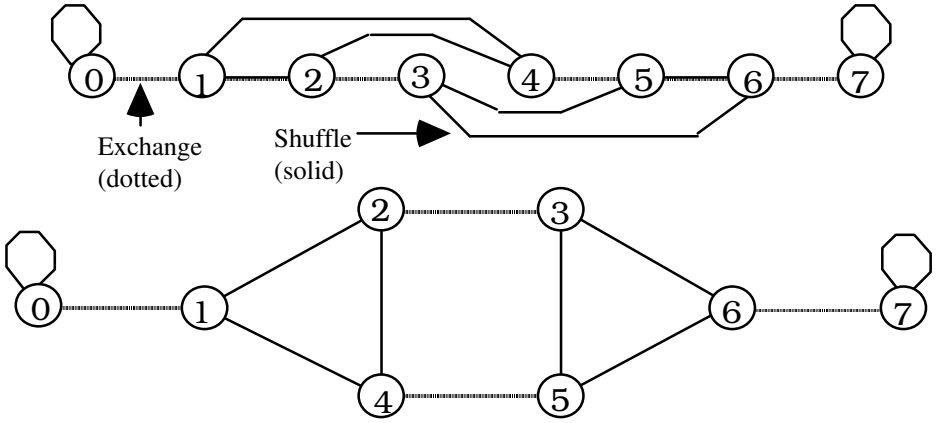


Fig. 15.18. Eight-node network with separate shuffle and exchange links.

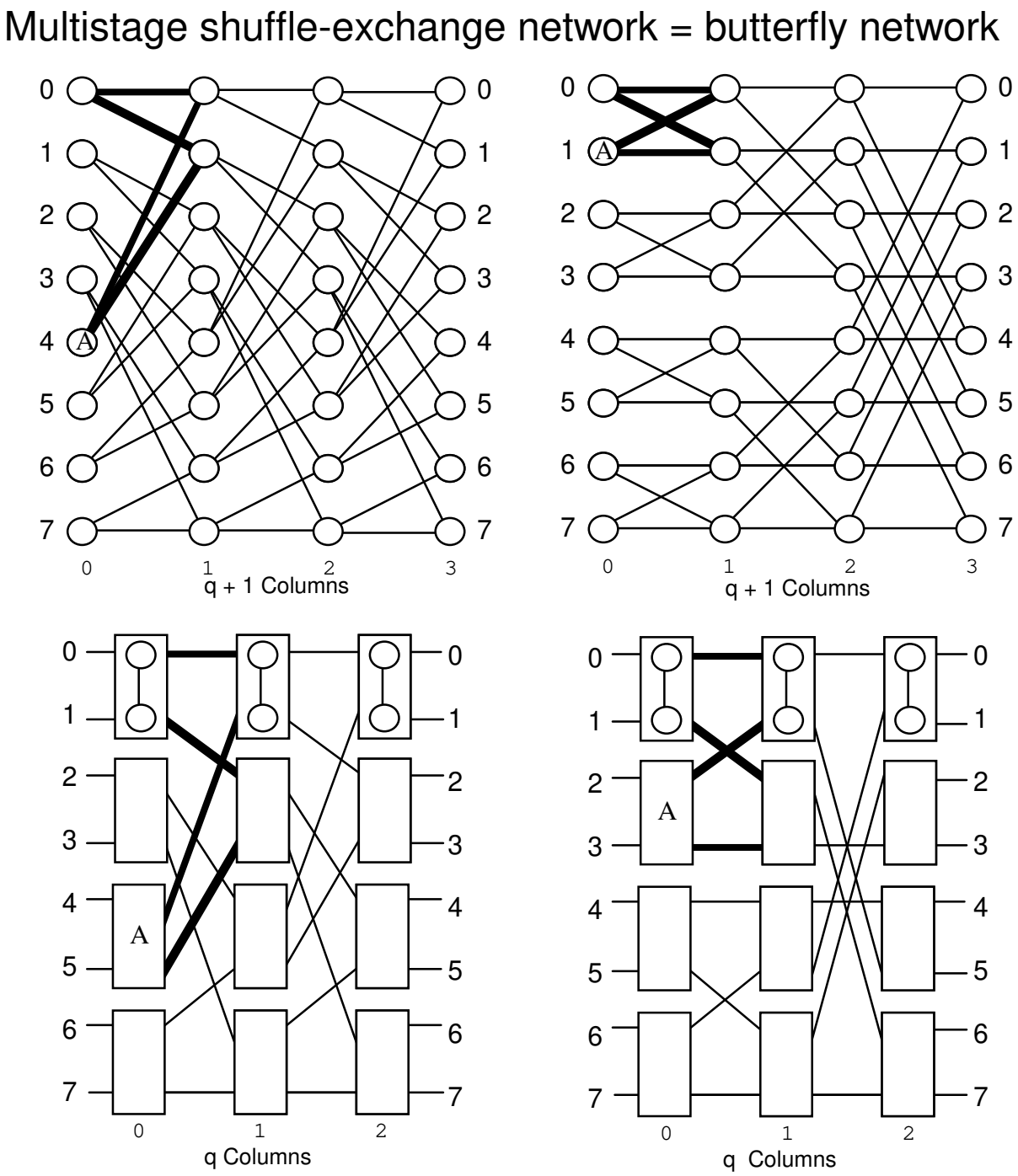


Fig. 15.19. Multistage shuffle–exchange network (omega network) is the same as butterfly network.

15.6 That's Not All, Folks!

When q is a power of 2, the $2^q q$-node cube-connected cycles network derived from the q-cube, by replacing each node with a q-cycle, is a subgraph of the $(q + \log_2 q)$-cube

Thus, CCC can be viewed as a pruned hypercube

Other pruning strategies are possible, leading to interesting tradeoffs

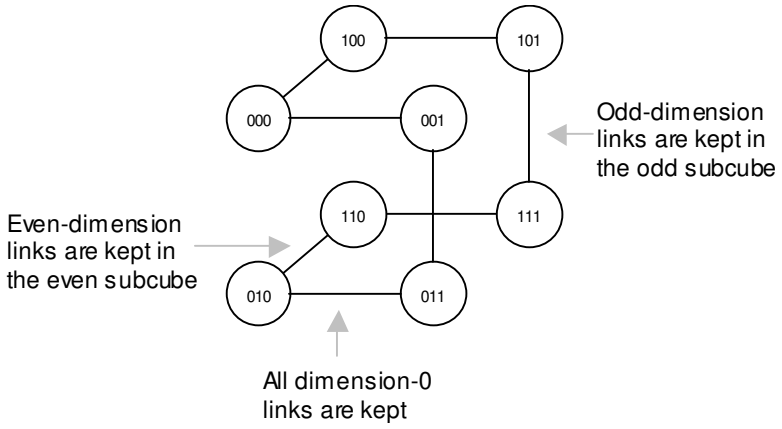


Fig. 15.20. Example of a pruned hypercube.

Möbius cube

Dimension-i neighbor of $x = x_{q-1}x_{q-2} \cdots x_{i+1}x_i \cdots x_1x_0$ is

$x_{q-1}x_{q-2} \cdots 0\ \bar{x}_i \cdots x_1x_0$ if $x_{i+1} = 0$

(as in the hypercube, x_i is complemented)

$x_{q-1}x_{q-2} \cdots 1\ \bar{x}_i \cdots \bar{x}_1\ \bar{x}_0$ if $x_{i+1} = 1$

(x_i and all the bits to its right are complemented)

For dimension $q-1$, since there is no x_q,

the neighbor can be defined in two ways,

leading to 0- and 1-Mobius cubes

A Möbius cube has a diameter of about 1/2 and an average inter-node distance of about 2/3 of that of a hypercube

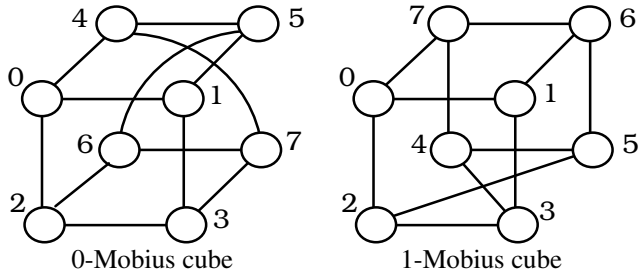


Fig. 15.21. Two 8-node Möbius cubes.

16 A Sampler of Other Networks

Back to TOC

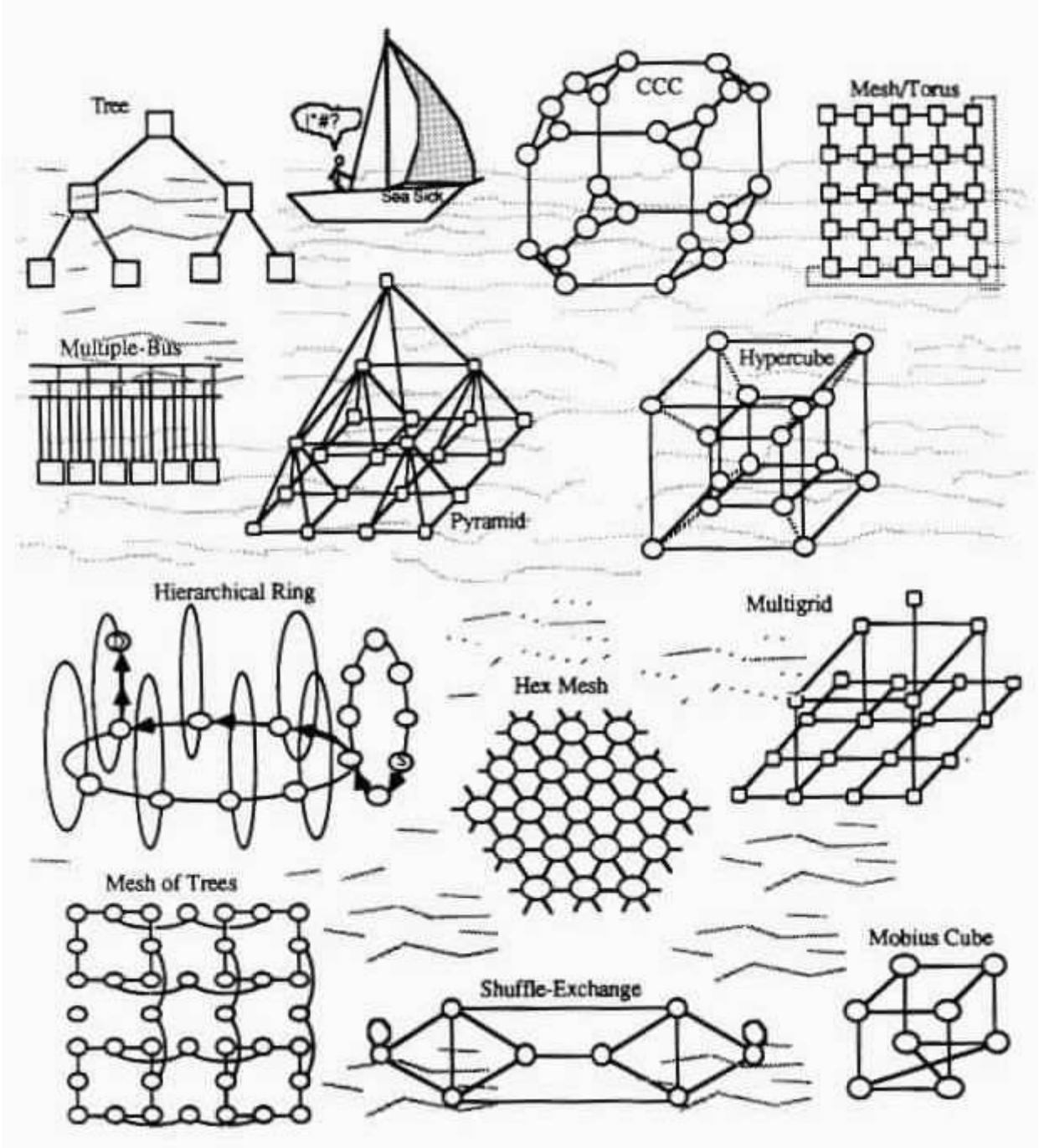
Chapter Goals

- Study examples of composite or hybrid architectures
- Study examples of hierarchical or multilevel architectures
- Complete the picture of the sea of interconnection networks

Chapter Contents

- 16.1. Performance Parameters for Networks
- 16.2. Star and Pancake Networks
- 16.3. Ring-Based Networks
- 16.4. Composite or Hybrid Networks
- 16.5. Hierarchical (Multilevel) Networks
- 16.6. Multistage Interconnection Networks

16.1 Performance Parameters for Networks



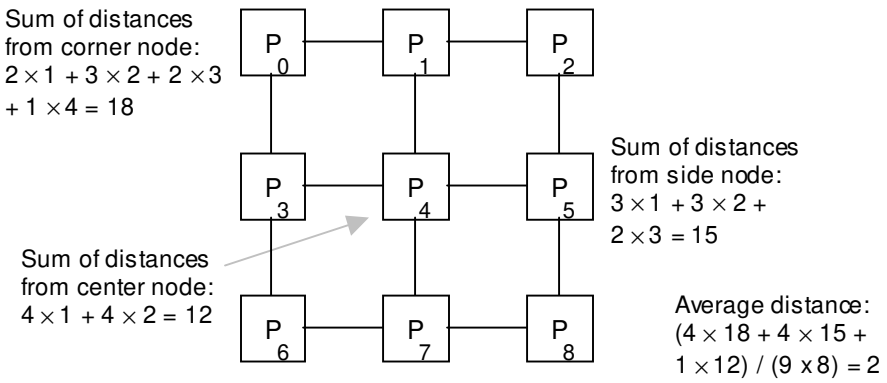
The sea of direct interconnection networks (Fig. 4.8, expanded).

Diameter D (indicator of worst-case message latency)

Routing diameter $D(R)$; based on routing algorithm R

Average internode distance Δ (based on shortest paths)

Routing average internode distance $\Delta(R)$

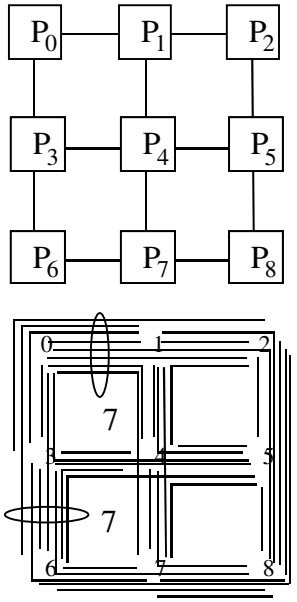


For the node-symmetric 3×3 torus, the average internode distance is determined by considering only paths from a single source node:

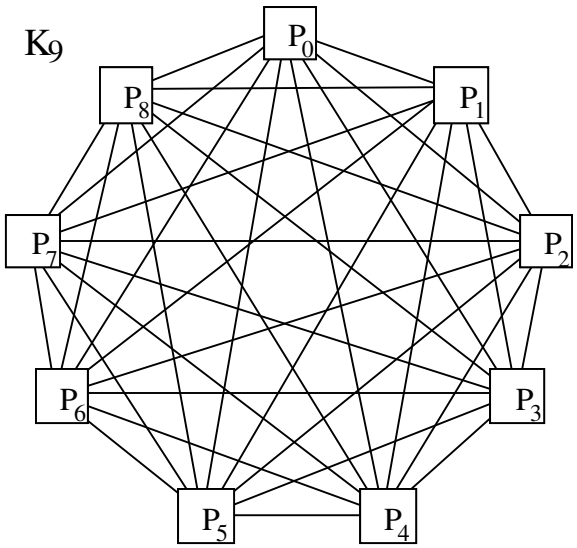
$$\Delta_{3\times3 \text{ torus}} = (4 \times 1 + 4 \times 2) / 8 = 1.5$$

Bisection width (indicator of random communication capacity)

Bisection bandwidth incorporates link capacities as well as their number



An embedding of K_9 into 3×3 mesh



Bisection width = $4 \times 5 = 20$

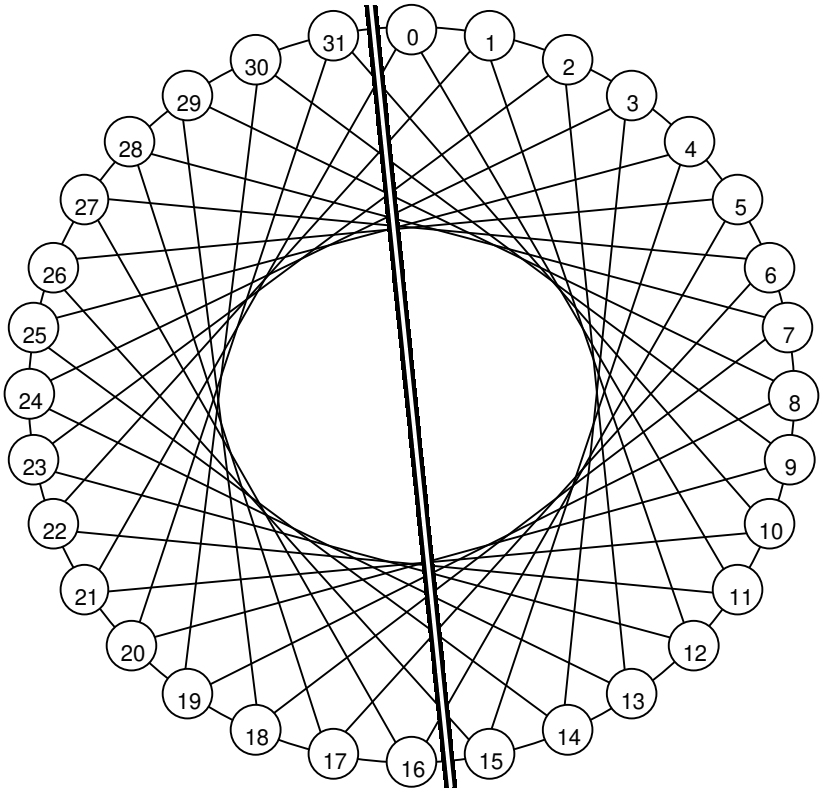


Fig. 16.2. A network whose bisection width is not as large at it appears.

Why so many different interconnection networks?

No single network provides optimal performance under all conditions

Each network has its advantages and drawbacks in terms of cost, latency, and bandwidth

We need to understand the interplay of these parameters to select suitable interconnection structures or to evaluate the relative merits of networks (parallel architectures)

Interplay between the node degree *d* and diameter *D*

Node degree is related to cost

Given *p* nodes of known degree *d*, we can interconnect them in different ways, leading to varying diameters

Question: What is the best way to interconnect *p* nodes of degree *d* to minimize the diameter of the resulting graph?

The problem of constructing a network of minimal diameter, given *p* nodes of degree *d*, or alternatively, building the largest possible network for a given node degree *d* and diameter *D*, is quite difficult

However, some useful bounds can be established that serve as benchmarks

Moore's bounds

A diameter-D regular digraph can have no more than $1 + d + d^2 + \cdots + d^D$ nodes

This yields a lower bound on the diameter of a p-node digraph of degree d which is known as Moore's bound

$$p \leq 1 + d + d^2 + \cdots + d^D = \frac{d^{D+1} - 1}{d - 1}$$

$$D \geq \log_d[p(d - 1) + 1] - 1$$

A graph matching this bound is a Moore digraph

The only possible Moore digraphs are:

Rings ($d = 1$, $D = p - 1$)

Complete graphs ($d = p - 1$, $D = 1$)

But there are near-optimal graphs that come close

A similar bound can be derived for undirected graphs

The largest undirected graph of diameter D has at most $1 + d + d(d-1) + d(d-1)^2 + \cdots + d(d-1)^{D-1}$ nodes

This leads to Moore's bound on the diameter of a p-node undirected graph of degree d:

$$p \leq 1 + d\,[1 + (d-1) + (d-1)^2 + \cdots + (d-1)^{D-1}]$$

$$= \quad 1 + d\,\frac{(d-1)^D - 1}{d-2}$$

$$D \geq \log_{d-1}\left[\frac{(p-1)(d-2)}{d} + 1\right]$$

For $d = 2$: $p \leq 1 + 2D$ or $D \geq (p-1)/2$

This diameter lower bound is achieved by ring with odd p

For $d = 3$: $D \geq \log_2[(p+2)/3]$ or $p \leq 3 \times 2^D - 2$

$D = 1$ allows us to have 4 nodes (the complete graph K_4)

The first interesting or nontrivial case is for $D = 2$ which allows at most $p = 10$ nodes (the Petersen graph)

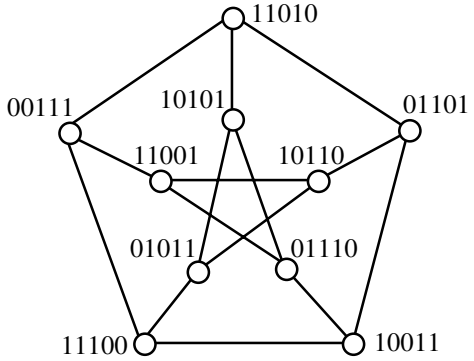


Fig. 16.1. The 10-node Petersen graph.

For larger networks, Moore's bound cannot be matched; but there exist networks that come very close to this bound

e.g. shuffle-exchange and CCC networks, with $d = 3$, have asymptotically optimal diameters within constant factors

For $d = 4$, Moore's diameter lower bound is $\log_2[(p + 1)/2]$

So, 2D mesh and torus networks are far from optimal in terms of their diameters, whereas the butterfly network is asymptotically optimal within a constant factor

For a q-cube with $p = 2^q$ and $d = q$, Moore's lower bound yields $D = \Omega(q/\log q)$. So, the diameter of a q-cube is a factor of $\log q$ worse than the optimal

Summary: for node degree d, Moore's bound establishes the lowest possible diameter that we can hope to achieve. Coming within a constant factor of this bound is usually good enough; the smaller the constant factor, the better.

Layout area and longest wire

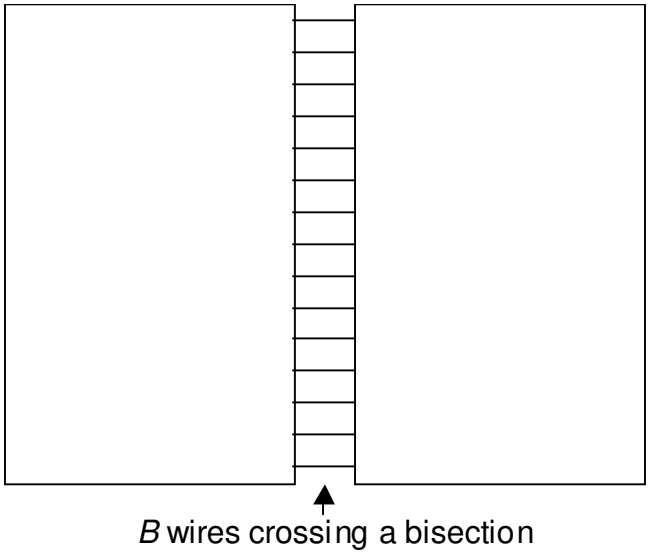
The VLSI layout area required by an interconnection network is intimately related to its bisection width B

If B wires must cross the bisection in a 2D layout and wire separation is to be 1 unit, then the smallest dimension of the VLSI chip will be at least B units

The chip area will thus be $\Omega(B^2)$ units

p-node 2D mesh needs $O(p)$ area

p-node hypercube needs at least $\Omega(p^2)$ area



The total number $pd/2$ of links (edges) is a very crude measure of network cost

With this measure, constant-degree networks have linear $O(p)$ cost and the p-node hypercube has $O(p \log p)$ cost.

The longest wire required in VLSI layout also affects the network performance

For example, any 2D layout of a p-node hypercube requires wires of length $\Omega(\sqrt{p/\log p})$

Because the length of the longest wire grows with system size, the per-node performance is bound to degrade for larger systems, thus implying sublinear speed-up

Composite figures of merit -- Example: dD, the product of node degree and network diameter, is a good measure for comparing networks of the same size, since it is a rough indicator of the cost of unit performance (d is proportional to cost, $1/D$ represents performance)

This measure has its limitations, particularly when applied to bus-based systems

Other network parameters include robustness and fault tolerance

16.2 Star and Pancake Networks

A qD star network, or q-star, has $p = q!$ (q factorial) nodes

Each node is labeled with a string $x_1 x_2 \cdots x_q$

where $(x_1, x_2, \cdots, x_q)$ is a permutation of $\{1, 2, \cdots, q\}$

Node $x_1 x_2 \cdots x_i \cdots x_q$ is connected to $x_i\, x_2 \cdots x_1 \cdots x_q$

for each i (note that x_1 and x_i are interchanged)

The node degree of a q-star with $q!$ nodes is $q - 1$

When the ith symbol is switched with x_1, the corresponding link is referred to as a dimension-i link

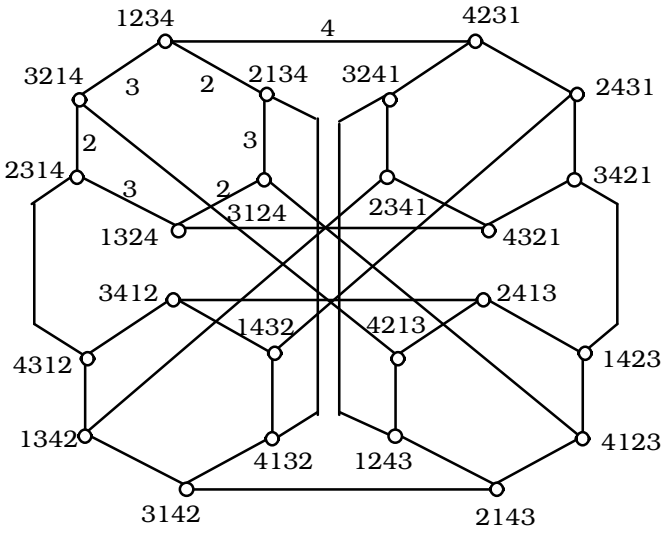


Fig. 16.3. The four-dimensional star graph.

The diameter of a q-star is at most $2q-3$

Justification: the following routing algorithm

```
Source node                  1 5 4 3 6 2
   Dimension-2 link to       5 1 4 3 6 2
   Dimension-6 link to       2 1 4 3 6 5
Last symbol now adjusted
   Dimension-2 link to       1 2 4 3 6 5
   Dimension-5 link to       6 2 4 3 1 5
Last 2 symbols now adjusted
   Dimension-2 link to       2 6 4 3 1 5
   Dimension-4 link to       3 6 4 2 1 5
Last 3 symbols now adjusted
   Dimension-2 link to       6 3 4 2 1 5
   Dimension-3 link to       4 3 6 2 1 5
Last 4 symbols now adjusted
   Dimension-2 link to       3 4 6 2 1 5   Destination
```

$D = \Theta(q)$ and $d = \Theta(q)$; but how is q related to p?

A q-star contains $p = q! \cong e^{-q} q^q \sqrt{2\pi q}$ processors

(using Stirling's approximation)

$$\ln p \cong -q + (q + 1/2) \ln q + \ln(2\pi)/2 = \Theta(q \log q)$$

or $q = \Theta(\log p / \log \log p)$

Hence, node degree and diameter are sublogarithmic

Star graph is asymptotically optimal to within a constant factor with regard to Moore's diameter lower bound

Routing on star graphs is simple and reasonably efficient; however, virtually all other algorithms are more complex than the corresponding algorithms on a hypercube

Because the node degree of a star network grows with its size, making it non-scalable, a degree-3 version of it, known as star-connected cycles (SCC) has been proposed

The diameter of SCC is about the same as a comparably sized CCC network

However, the routing algorithm for SCC is somewhat more complex

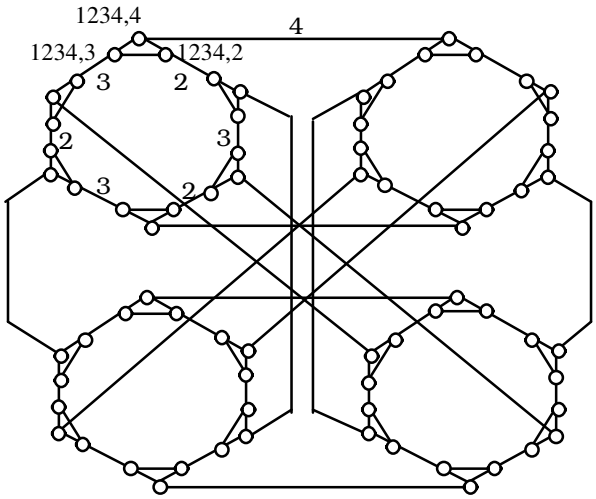


Fig. 16.4. The four-dimensional star-connected cycles network.

Like the star graph, the pancake network also has $p = q!$ nodes that are labeled by the various permutations of the symbols $\{1, 2, \cdots, q\}$

In the q-pancake, Node $x_1 x_2 \cdots x_i\, x_{i+1} \cdots x_q$ is connected to nodes $x_i\, x_{i-1} \cdots x_2 x_1 x_{i+1} \cdots x_q$ for each i ($x_1 x_2 \cdots x_i$ is flipped, like a pancake)

Routing in pancake networks is very similar to routing in star graphs

Denoting the connection that results from flipping the first i symbols ($2 \leq i \leq q$) as the dimension-i link, we have for example:

```
Source node                  1 5 4 3 6 2
   Dimension-2 link to       5 1 4 3 6 2
   Dimension-6 link to       2 6 3 4 1 5
Last 2 symbols now adjusted
   Dimension-4 link to       4 3 6 2 1 5
Last 4 symbols now adjusted
   Dimension-2 link to       3 4 6 2 1 5   Destination
```

Generally, we need 2 flips per symbol; one flip to bring the symbol to the front from its current position i, and another one to send it to its desired position j

Thus, the diameter of the q-pancake is $2q - 3$

One can define the connectivities of the $q!$ nodes labeled by the permutations of $\{1, 2, \cdots, q\}$ in other ways

In a rotator graph, node $x_1 x_2 \cdots x_i\, x_{i+1} \cdots x_q$ is connected to $x_2 \cdots x_i\, x_1 x_{i+1} \cdots x_q$ (obtained by a left rotation of the first i symbols) for each i in the range $2 \leq i \leq q$

The node degree of a q-rotator is $q - 1$, as in star and pancake graphs, but its diameter and average inter-node distance are smaller

Except for SCC, all of the networks introduced in this section represent special cases of a class of networks known as Cayley graphs

A Cayley graph is characterized by a set Λ of node labels and a set Γ of generators, each defining one neighbor of a node x

The ith generator γ_i can be viewed as a rule for permuting the node label to get the label of its "dimension-i" neighbor

For example, the star graph has $q - 1$ generators that correspond to interchanging the 1st and ith symbols in the node label

Index-permutation graphs, a generalization of Cayley graphs in which the node labels are not restricted to consist of distinct symbols, can lead to other interesting and useful interconnection networks.

16.3 Ring-Based Networks

Ring: simple, but low-performance

Multilevel rings and chordal rings

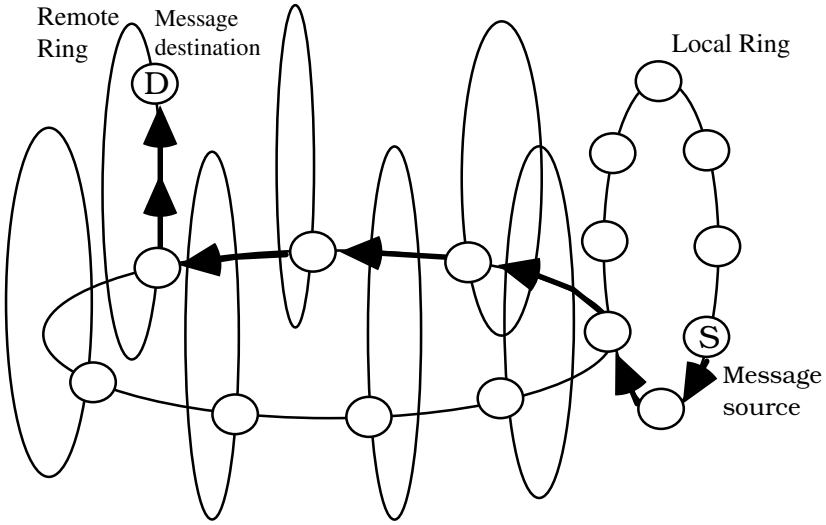


Fig. 16.5. A 64-node ring-of-rings architecture composed of eight 8-node local rings and one second-level ring.

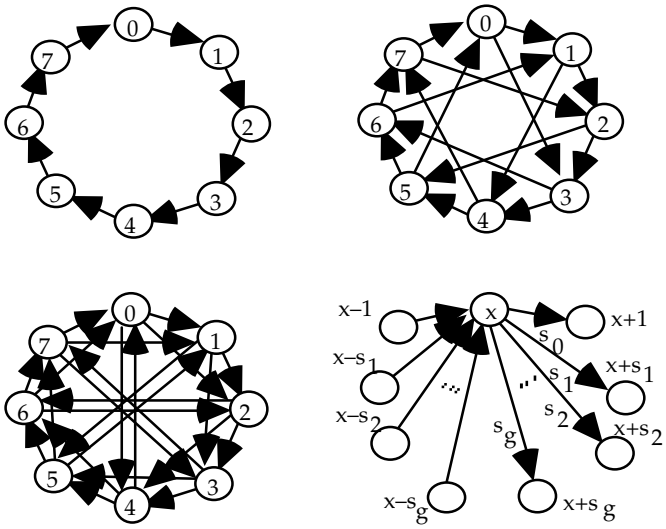


Fig. 16.6. Unidirectional ring, two chordal rings, and node connectivity in general.

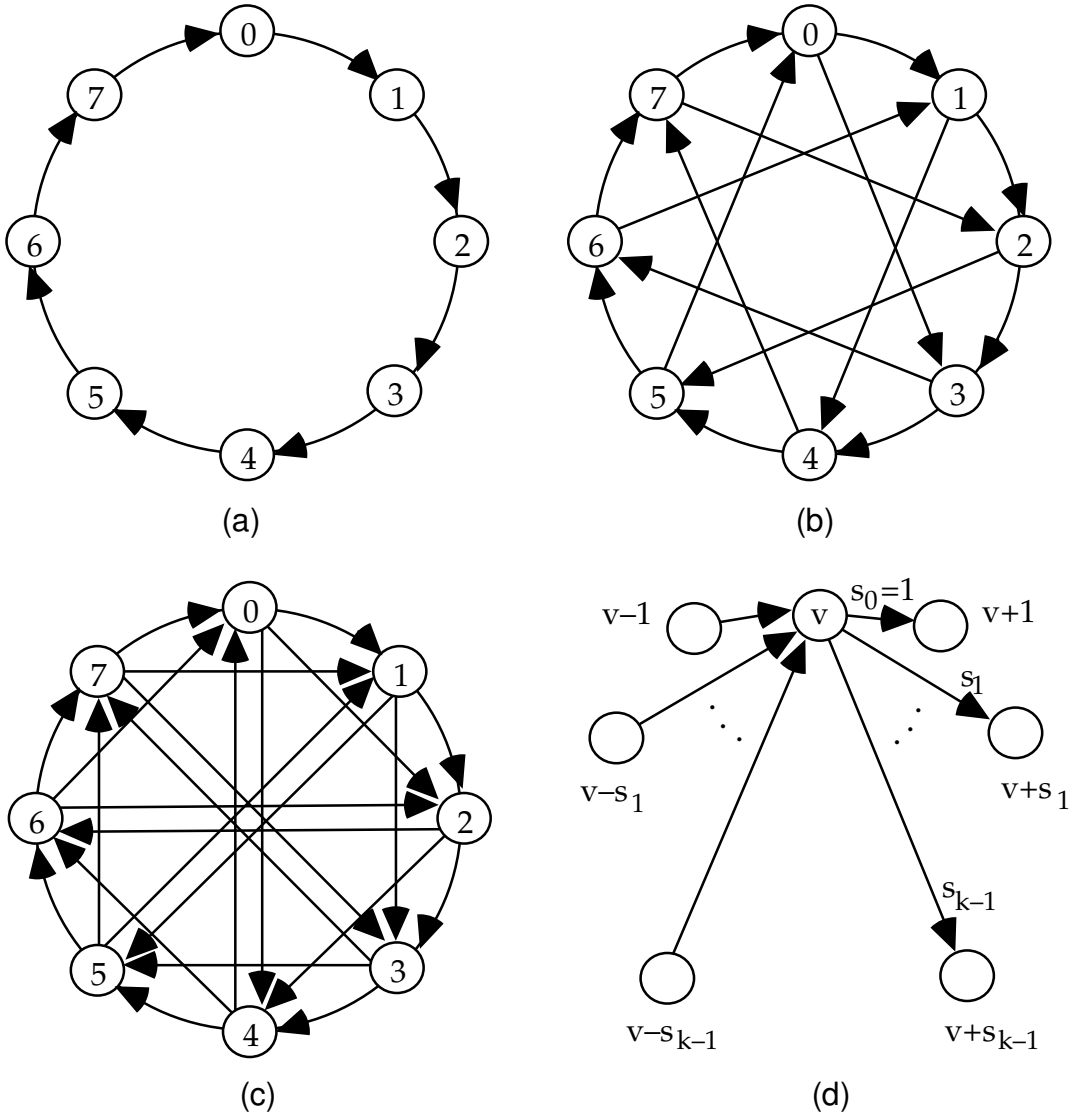


Fig. 16.6. Unidirectional ring, two chordal rings, and node connectivity in general.

Chordal rings are node symmetric

Optimal chordal rings derived as above are very similar, though not isomorphic, to (g+1)-dimensional tori

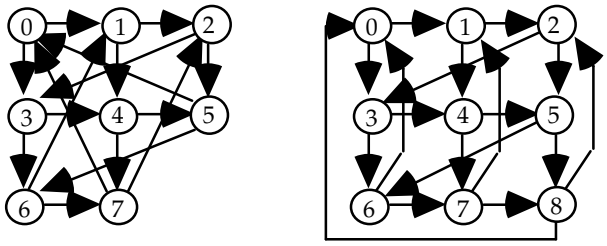


Fig. 16.7. Chordal rings redrawn to show their similarity to torus networks.

Periodically regular chordal ring

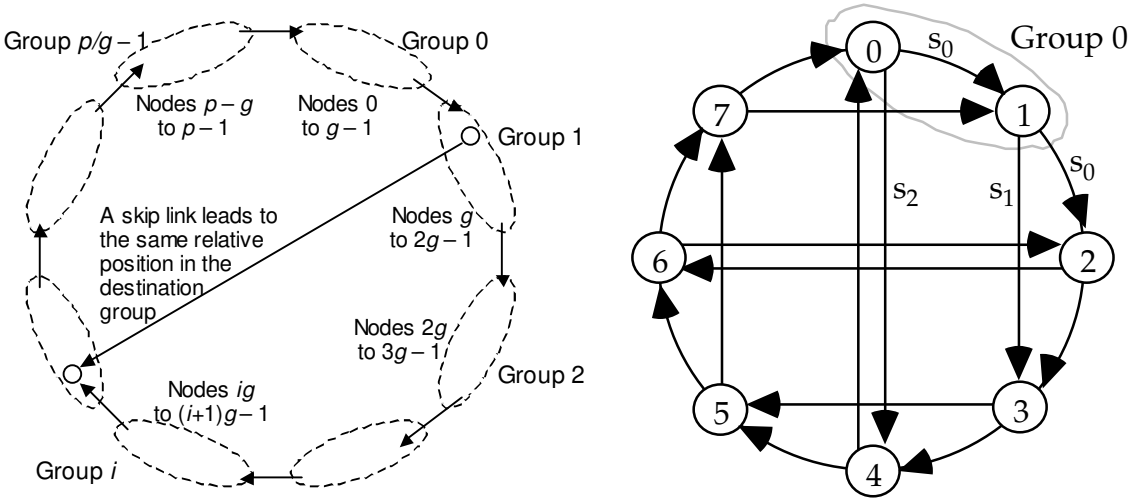


Fig. 16.8. Periodically regular chordal ring.

A variant of the greedy routing algorithm (first route a packet to the head of a group) works nicely

Chordal rings and PRC rings have bidirectional variants with similar properties to the unidirectional versions

Area-efficient VLSI layouts are known for PRC rings

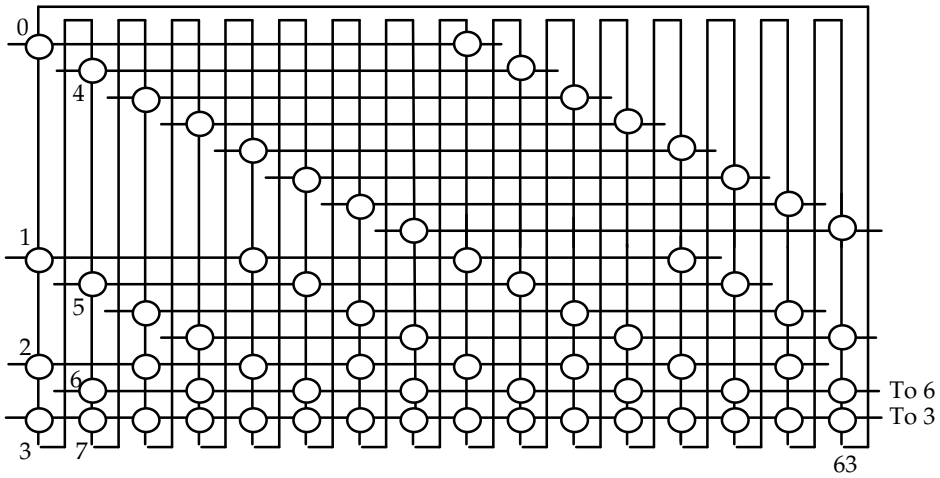


Fig. 16.9. VLSI layout for a 64-node periodically regular chordal ring.

Providing nil skips for some of the nodes in each group constitutes a mechanism for performance-cost tradeoffs that are identical in nature to those offered by the *q*-D CCC architecture when rings have more than *q* nodes

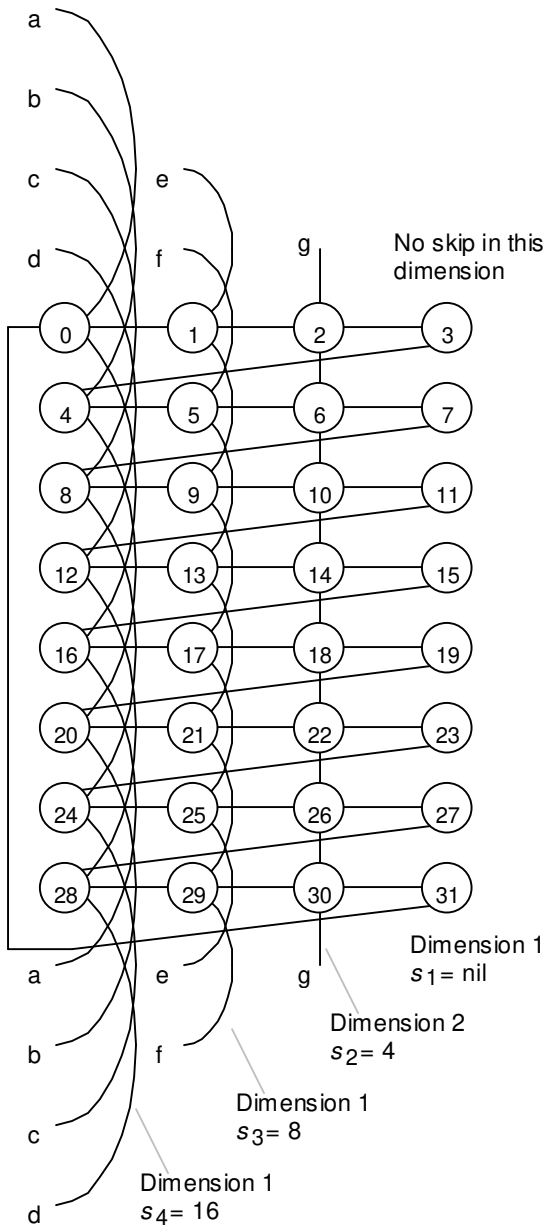


Fig. 16.10. A PRC ring redrawn as a butterfly- or ADM-like network.

16.4 Composite or Hybrid Networks

Composite or hybrid networks combine the connectivity rules from two (or more) pure networks in order to

- achieve some advantages from each structure
- derive network sizes that are unavailable with either pure architecture
- realize any number of performance/cost benefits

Network composition by Cartesian product operation

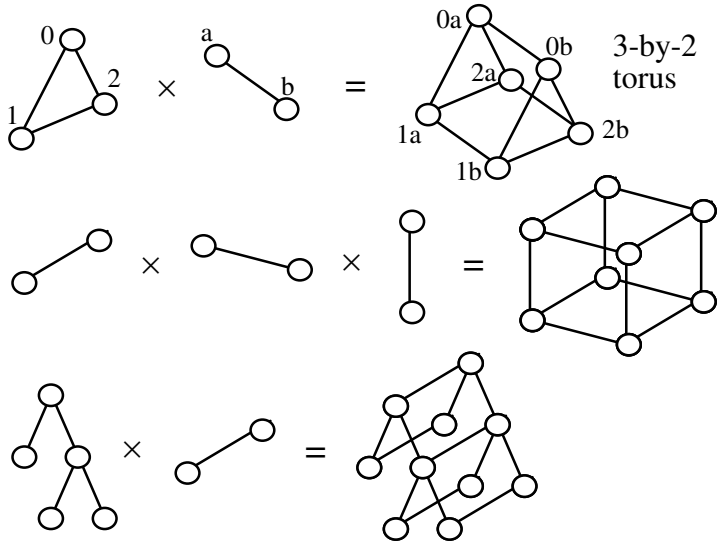


Fig. 13.4. Examples of product graphs.

Topological properties of product graphs

$$p = p'p'' \qquad d = d' + d'' \qquad D = D' + D'' \qquad \Delta = \Delta' + \Delta''$$

Routing on product graphs:

Given optimal/efficient/deadlock-free routing algorithms for G' and G'', the following 2-phase algorithm will be optimal/efficient/deadlock-free for routing from $u'u''$ to $v'v''$ in the product graph G

Phase 1. Route from $u'u''$ to $v'u''$ via G' edges

Phase 2. Route from $v'u''$ to $v'v''$ via G'' edges

The algorithm above may be called the "G'-first" routing

Broadcasting on product graphs:

First send from $v'v''$ to all nodes xv'', $x \in V'$, using a broadcasting algorithm for G'; then broadcast from each node xv'' to all nodes xy, $y \in V''$, using a broadcasting algorithm for G''

Semigroup and parallel prefix computations can be similarly performed by using the respective algorithms for the component networks

If the component graphs are Hamiltonian, then the $p' \times p''$ torus will be a subgraph of G

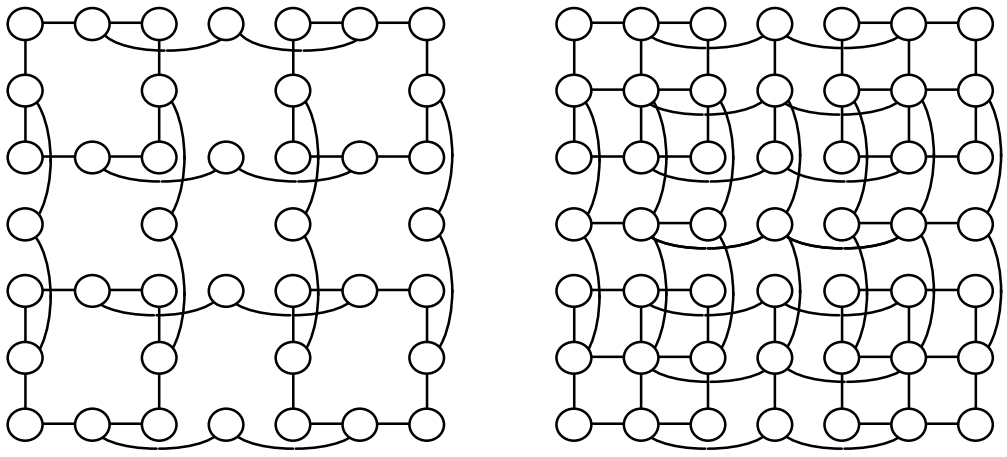


Fig. 16.11. Mesh of trees compared with mesh-connected trees.

16.5 Hierarchical (Multilevel) Networks

Hierarchical or multilevel interconnection networks can be defined in a variety of ways

Example: hierarch. composition by recursive substitution (replacing each node with a network, as in CCC)

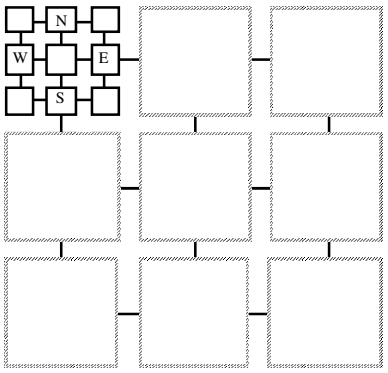


Fig. 16.12. The mesh of meshes network exhibits greater modularity than a mesh.

Motivations for designing hierarchical networks include

greater modularity	lower cost
finer scalability	better fault tolerance

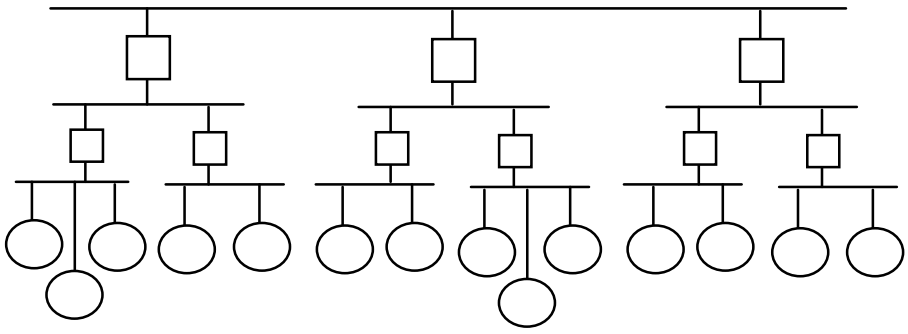


Fig. 16.13. Hierarchical or multilevel bus network.

16.6 Multistage Interconnection Networks

Direct versus indirect (multistage) network

Rearrangeable network (e.g. Beneš network)

Self-routing MIN

The butterfly network is a self-routing MIN, but it is not a permutation network

Beneš network can realize any permutation, but is not self-routing

A natural question is whether there exist self-routing permutation networks (yes there are!)

A full permutation can be realized via sorting of the destination addresses

Any p-sorter of the type discussed in Chapter 7 can be viewed as a self-routing MIN capable of routing $p \times p$ permutations

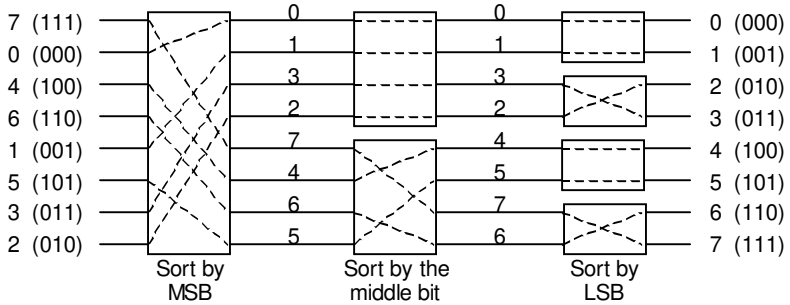
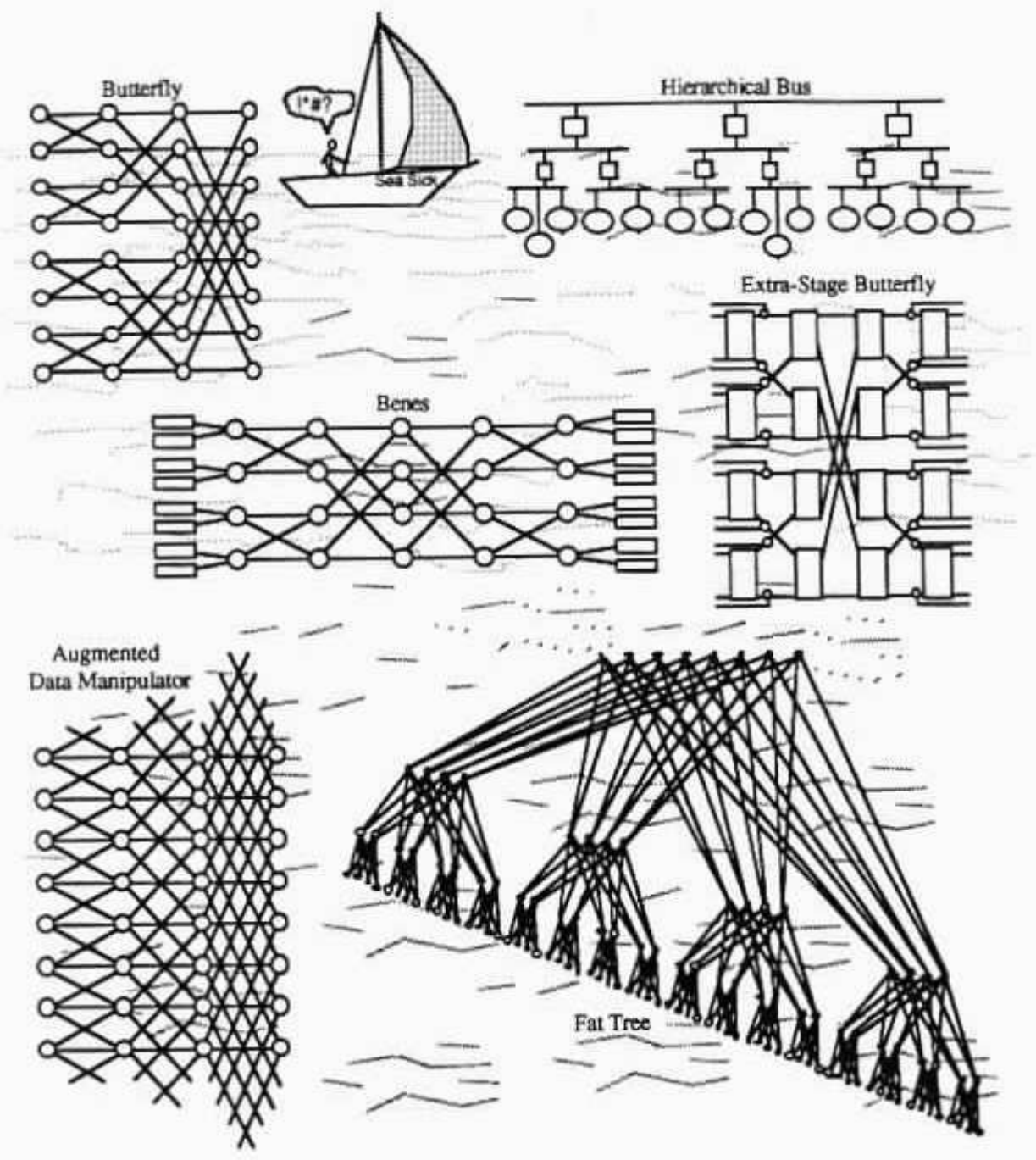


Fig. 16.14. Example of sorting on a binary radix sort network.



The sea of indirect interconnection networks.

Partial List of Important MINs

Augmented data manipulator (ADM): aka unfolded PM2I (Fig. 15.12)

Banyan: Any MIN with a unique path between any input and any output (e.g. butterfly)

Baseline: Butterfly network with nodes labeled differently

Beneš: Back-to-back butterfly networks, sharing one column (Figs. 15.9-10)

Bidelta: A MIN that is a delta network in either direction

Butterfly: aka unfolded hypercube (Figs. 6.9, 15.4-5)

Data manipulator: Same as ADM, but with switches in a column restricted to same state

Delta: Any MIN for which the outputs of each switch have distinct labels (say 0 & 1 for 2×2 switches) and path label, composed of concatenating switch output labels leading from an input to an output depends only on the output

Flip: Reverse of the omega network (inputs $\times$ outputs)

Indirect cube: Same as butterfly or omega

Omega: Multi-stage shuffle-exchange network; isomorphic to butterfly (Fig. 15.19)

Permutation: Any MIN that can realize all permutations

Rearrangeable: Same as permutation network

Reverse baseline: Baseline network, with the roles of inputs and outputs interchanged

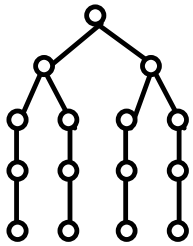


Figure for Problem 16.11.

Part V Some Broad Topics

Back to TOC

Part Goals

- Study topics that cut across architectures
  - Mapping a computation onto a machine
  - Previously dealt with computation and communication; what about I/O?
  - Hardware faults and resultant errors
  - System and software issues?

Part Contents

- 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

Back to TOC

Chapter Goals

- Learn how to achieve algorithm portability via emulation
- Study task scheduling for parallel systems, including complexity aspects and bounds

Chapter Contents

- 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

General result 1 (emulation via graph embedding)

Slowdown $\leq$ dilation $\times$ congestion $\times$ load factor

The bound is tight; e.g., embedding K_p into K_2

dilation = 1, congestion = $p^2/4$, load = $p/2$

General result 2 (PRAM emulating degree-d network)

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

General result 3 (butterfly emulating degree-d network)

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

Butterfly is a universally efficient bounded-degree net

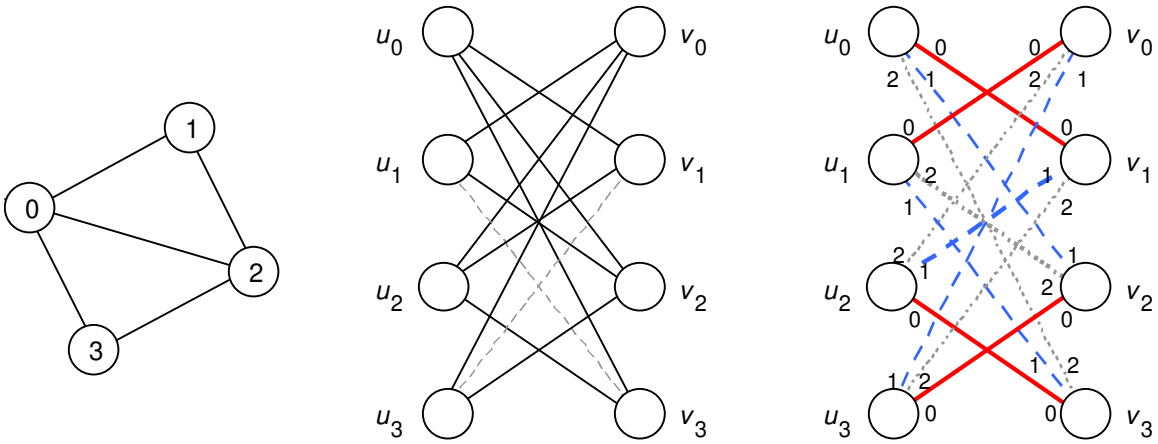
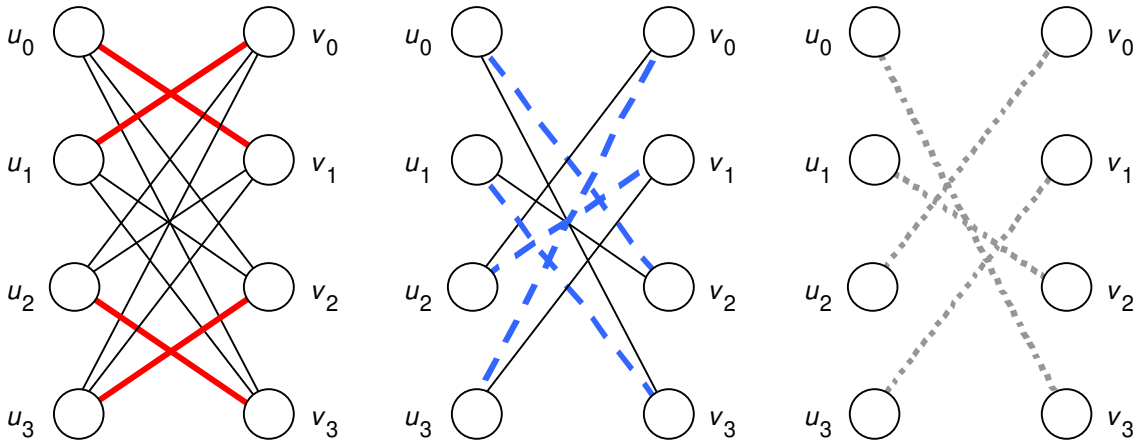


Fig. 17.1. Converting a routing step in a degree-3 network to three permutations or perfect matchings.



A set of three perfect matchings for a degree-3 bipartite graph.

17.2 Distributed Shared Memory

Randomized emulation of 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*)

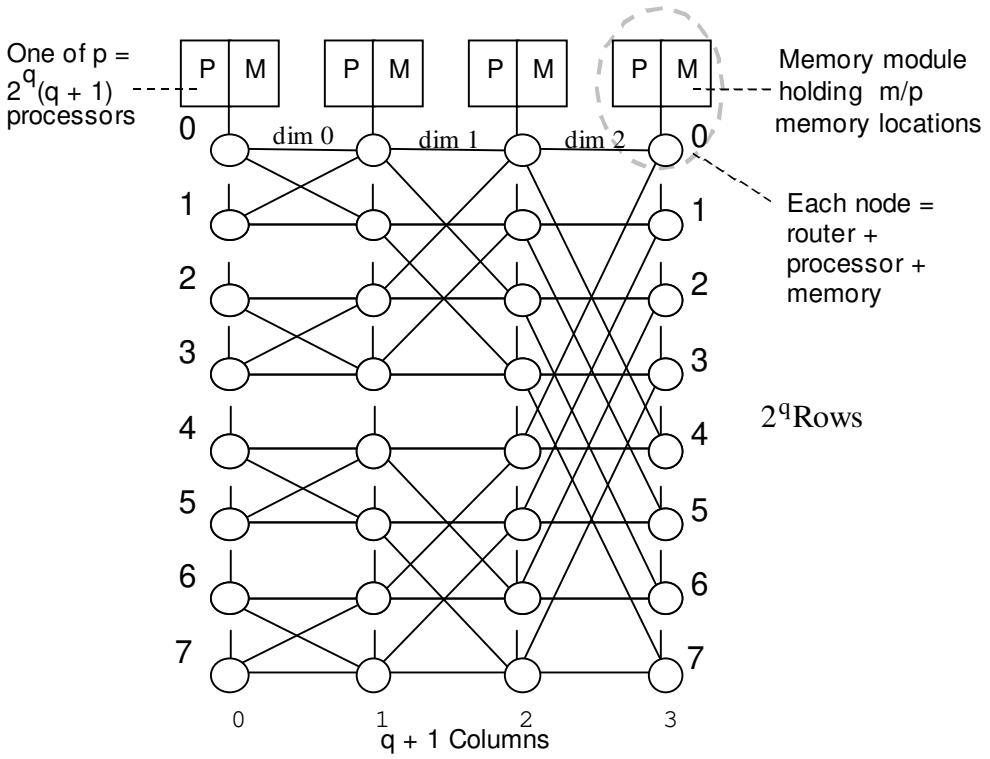


Fig. 17.2. Butterfly distributed-memory machine emulating the PRAM.

Emulation of PRAM using (p log p)-node butterfly MIN

Average slowdown = O(log p)

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

By using only p/(log p) physical processors to emulate a p-processor PRAM this emulation scheme becomes quite efficient (memory accesses of the log p virtual processors assigned to each physical processor can be pipelined)

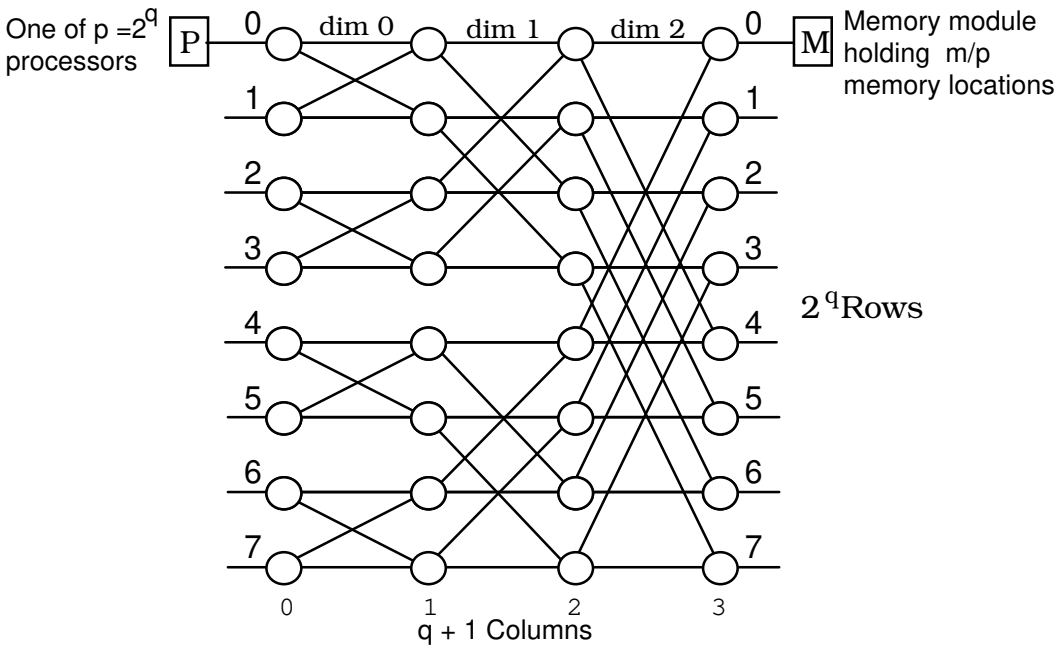


Fig. 17.3. Distributed-memory machine, with a butterfly multistage interconnection network, emulating the PRAM.

Deterministic emulation of PRAM on a network

Both more difficult and less efficient

Recall that a butterfly can route random permutations in O(log p) steps on the average but that worst-case communication patterns take O($\sqrt{p}$) time

One idea:

Store $\log_2 m$ copies of each of m memory locations

Time-stamp each updated value

A "write" is complete if 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

Thus, a few congested links won't delay the operation

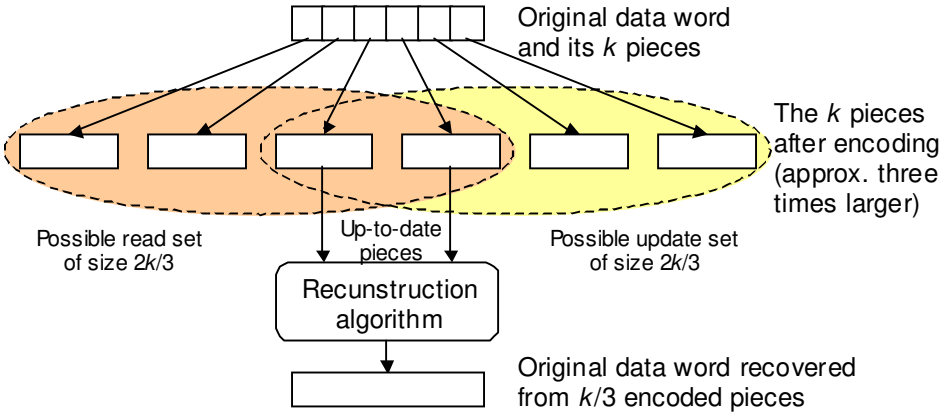


Fig. 17.4. Illustrating the information dispersal approach to PRAM emulation with lower data redundancy.

17.3 The Task Scheduling Problem

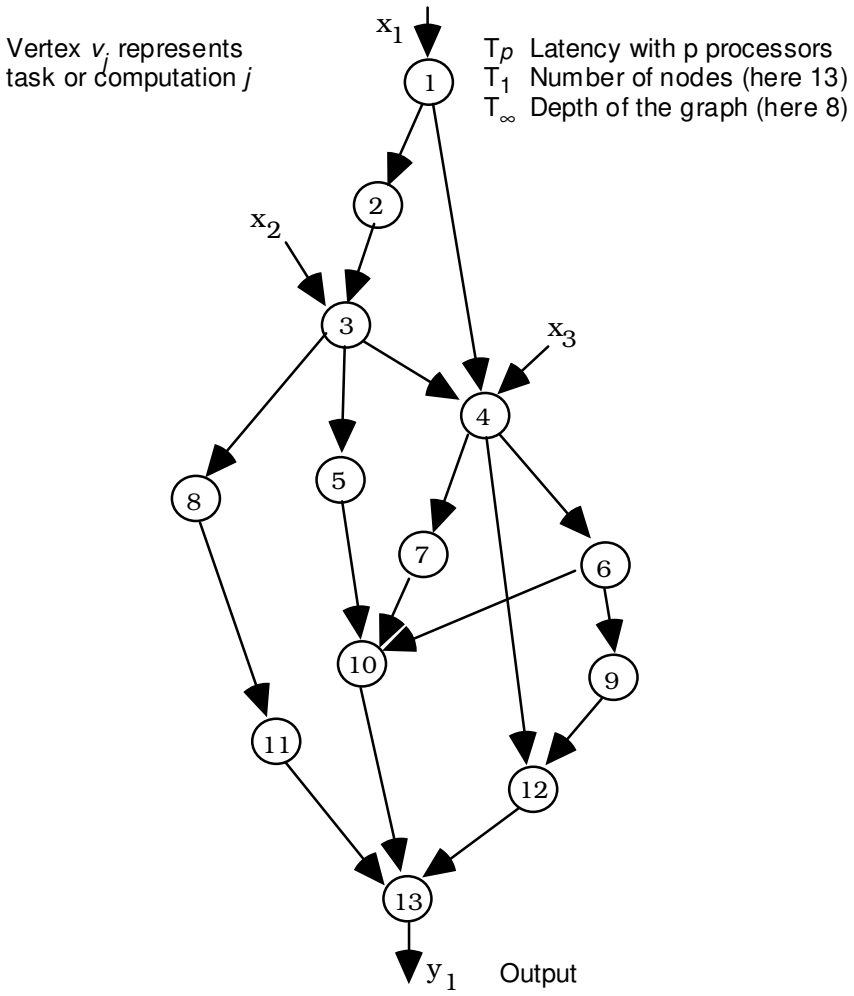


Fig. 17.5. Example task system showing communications or dependencies.

Scheduling parameters and criteria
Running time, task creation (static/dynamic), relationships (priority, precedence, ...), start/end time (release, deadline)

Types of scheduling algorithms
Preemptive/nonpreemptive, fine/medium/coarse grain

17.4 A Class of Scheduling Algorithms

List scheduling

Assign a priority level to each task

Construct a task list in priority order

(tag the tasks that are ready for execution)

Assign to an available processor the first tagged task

(update the list tags when tasks terminate)

When all processors are identical, list schedulers differ only in their priority assignment schemes

A possible priority assignment scheme for list scheduling:

Determine the depth T_∞ of the task graph, which is an indicator of its minimum possible execution time

Take T_∞ as a goal for the total running time T_p

Determine the latest possible time step in which each task can be scheduled if our goal is to be met (done by "layering" the nodes beginning with the output node)

The results of layering for the task graph of Fig. 17.5 are:

1	2	3	4	5	6	7	8	9	10	11	12	13	Tasks in numerical order
1	2	3	4	6	5	6	6	6	7	7	7	8	Latest possible times (layers)

Assign task priorities in order of the latest possible times. Ties broken, e.g., by giving priority to a task with a larger number of descendants

For our example, this secondary criterion is of no help, but generally, if a task with more descendants is executed first, the running time will likely be improved.

1*	2	3	4	6	5	7	8	9	10	11	12	13	Tasks in priority order
1	2	3	4	5	6	6	6	6	7	7	7	8	Latest possible times
2	1	3	3	2	1	1	1	1	1	1	1	0	Number of descendants

Schedule on $p = 2$ 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
3* 4 6 5 7 8 9 10 11 12 13	$t = 3$	v_3 scheduled
4* 6 5* 7 8* 9 10 11 12 13	$t = 4$	v_4, v_5 scheduled
6* 7* 8* 9 10 11 12 13	$t = 5$	v_6, v_7 scheduled
8* 9* 10* 11 12 13	$t = 6$	v_8, v_9 scheduled
10* 11* 12* 13	$t = 7$	v_{10}, v_{11} scheduled
12* 13	$t = 8$	v_{12} scheduled
13*	$t = 9$	v_{13} scheduled (done)

Schedule on $p = 3$ 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
3* 4 6 5 7 8 9 10 11 12 13	$t = 3$	v_3 scheduled
4* 6 5* 7 8* 9 10 11 12 13	$t = 4$	v_4, v_5, v_8 scheduled
6* 7* 9 10 11* 12 13	$t = 5$	v_6, v_7, v_{11} scheduled
9* 10* 12 13	$t = 6$	v_9, v_{10} scheduled
12* 13	$t = 7$	v_{12} scheduled
13*	$t = 8$	v_{13} scheduled (done)

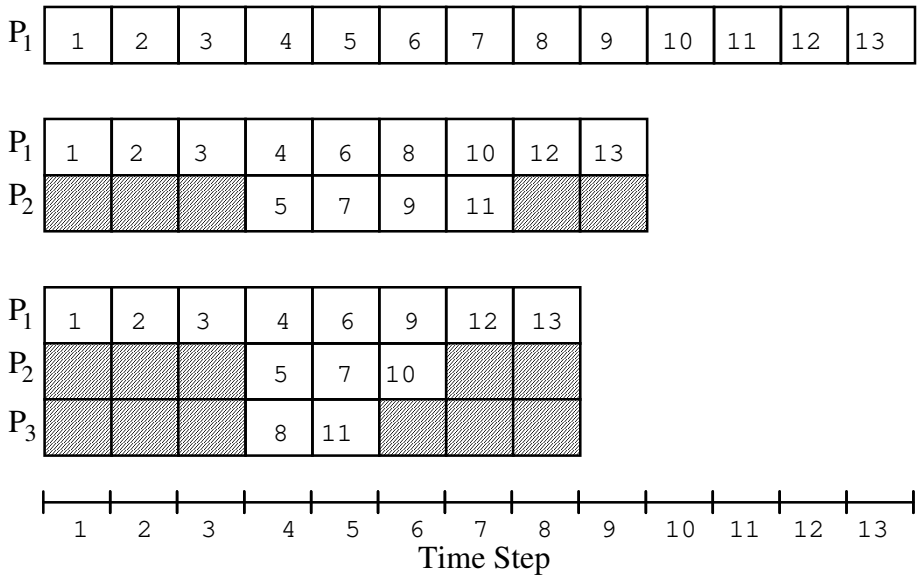


Fig. 17.6. Schedules with $p = 1, 2, 3$ processors for an example task graph with unit-time tasks.

Scheduling with non-unit-time tasks

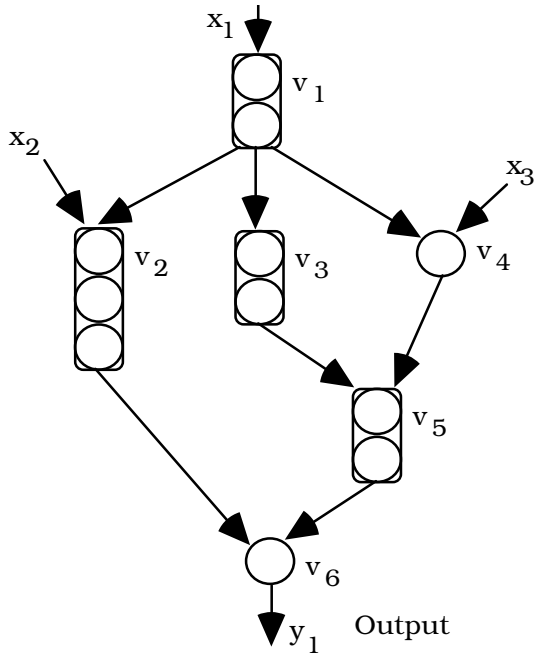


Fig. 17.7. Example task system with task running times of 1, 2, or 3 units.

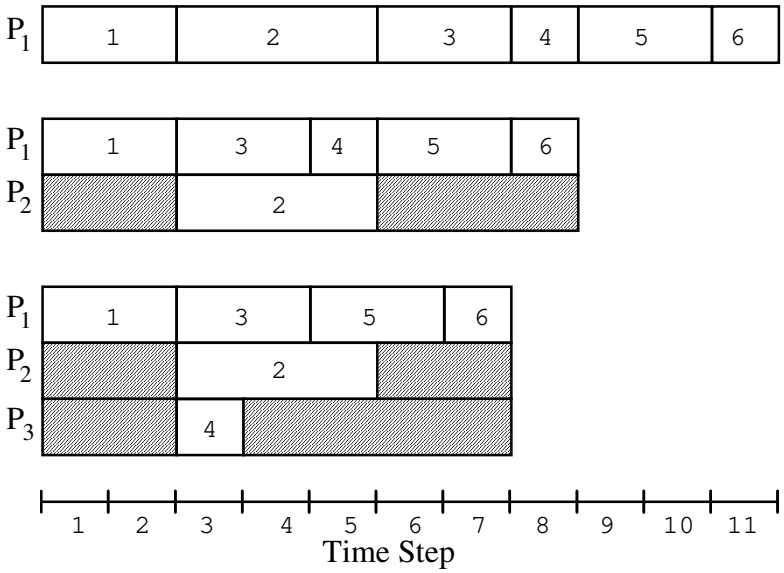


Fig. 17.8. Schedules with *p* = 1, 2, 3 processors for an example task graph with nonuniform running times.

17.5 Some Useful Bounds for Scheduling

Brent's scheduling theorem:

$$T_p < T_\infty + T_1/p$$

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

Let there be n_t nodes scheduled at time t.

Clearly, $\sum_t n_t = T_1$

With 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:

$$T_p \leq \sum_{t=1}^{T_\infty} \lceil n_t / p \rceil < \sum_{t=1}^{T_\infty} (n_t / p + 1) = T_\infty + T_1/p$$

Brent's theorem offers an approximation to the speedup:

$$\text{Speedup} \cong \frac{T_1}{T_\infty + T_1/p} = \frac{p}{1 + pT_\infty/T_1}$$

This can be viewed as a generalized form of Amdahl's law

A large value for T_∞/T_1 is an indication that the task has a great deal of sequential dependencies, which limits the speedup to at most T_1/T_∞ with any number of processors

A small value for T_∞/T_1 allows us to approach the ideal speedup of p with p processors

Good-news corollary 1: $T_\infty \leq T_p < 2T_\infty$ for $p \geq T_1/T_\infty$

Good-news corollary 2: $T_1/p \leq T_p < 2T_1/p$ for $p \leq T_1/T_\infty$

ABCs of Parallel Processing

in one transparency* (parhami@ece.ucsb.edu)

* Originally appeared in *Computer Architecture News*, Vol. 27, No. 1, p. 2, March 1999.

f = unparallelizable fraction of a task (sequential overhead)
T_x = running time of a task when executed on x processors

A Amdahl's Law (Speed-up Formula)

Bad news: Sequential overhead will kill you, since:

$$\text{Speed-up} = \frac{T_1}{T_p} \leq \frac{1}{f + \frac{1-f}{p}} \leq \min\left(\frac{1}{f}, p\right)$$

Morale: For $f = 0.1$, e.g., the speed-up will be at best 10, no matter what the number of processors (peak OPS).

B Brent's Scheduling Theorem

Good news: Optimal scheduling is a very difficult problem, but even a naive scheduling algorithm can ensure:

$$\frac{T_1}{p} \leq T_p < \frac{T_1}{p} + T_\infty = \frac{T_1}{p}\left(1 + \frac{p}{T_1/T_\infty}\right)$$

Result: For a reasonably parallel task (with small T_∞), or for a suitably small number of processors (say, $p < T_1/T_\infty$), good speed-up and high utilization are attainable.

C Cost-Effectiveness Adage

Real news: The most cost-effective parallel solution to a given problem is often not the one with:

Highest peak OPS	(communication can kill you)
Greatest speed-up	(at what cost?)
Best utilization	(hardware busy doing what?)

Analogy: Mass transit (SIMD) might be more cost-effective than using private vehicles (MIMD) even if it is slower and leads to many empty seats on some trips.

17.6 Load Balancing and Dataflow Systems

Task running times are not constants

A processor may run out of things to do before other processors complete their assigned tasks

Some processors may remain idle for long periods of time as they wait for prerequisite tasks to be executed

In these cases, a load balancing policy may be applied

As we learn about execution times and interdependencies of tasks at run time, we may switch as yet unexecuted tasks from an overloaded processor to a less loaded one

Load balancing can be initiated by a lightly loaded or by an overburdened node (receiver/sender-initiated)

Unfortunately, load balancing may involve a great deal of overhead that reduces the potential gains

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 sending a message to this central supervisor and in return receives one or more tasks to perform

This works nicely for tasks with small contexts and/or relatively long running times

Dataflow systems

Hardware-level implementation of self-scheduling scheme

A dataflow computation is characterized by a dataflow graph (we consider only decision/loop-free graphs)

Tokens are used to keep track of data availability

Once tokens appear on all inputs of a node, the node is enabled or "fired", resulting in tokens to be removed from its inputs and placed on each of its outputs

Static dataflow: an edge can carry no more than one token

Dynamic dataflow: multiple tagged tokens can appear on the edges and are "consumed" after matching their tags

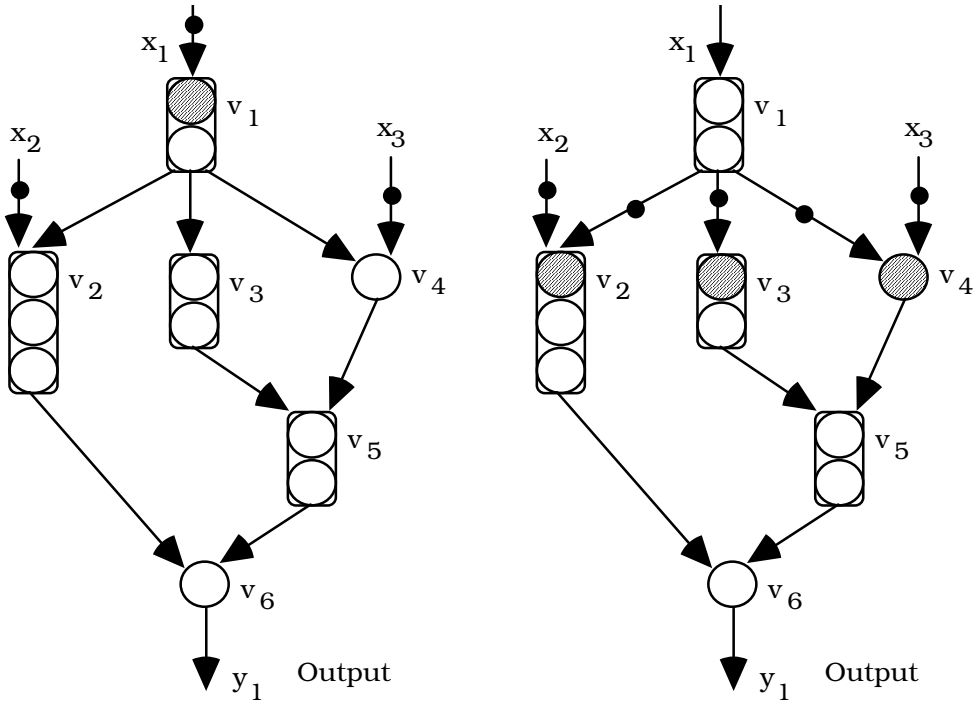


Fig. 17.9. Example dataflow graph with token distribution at the outset (left) and after 2 time units (right).

18 Data Storage, Input, and Output

Back to TOC

Chapter Goals

- Elaborate on problems of data distribution and caching
- Deal with the memory/processor speed gap which is particularly severe in distributed-memory systems
- Learn about parallel I/O technology

Chapter Contents

- 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 aggravated by parallelism

- Global shared memory access mechanism slower
- Distributed memory penalizes remote accesses

Remedies

- Data distribution -- good with static data sets
- Data caching -- introduces coherence problems
- Latency tolerance (hiding) -- e.g., multithreading

Why data caching works

Hit rate r = fraction of accesses satisfied by the cache

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

Cache parameters: size, block length (line width), placement policy, replacement policy, write policy

Example: two-way set-associative cache

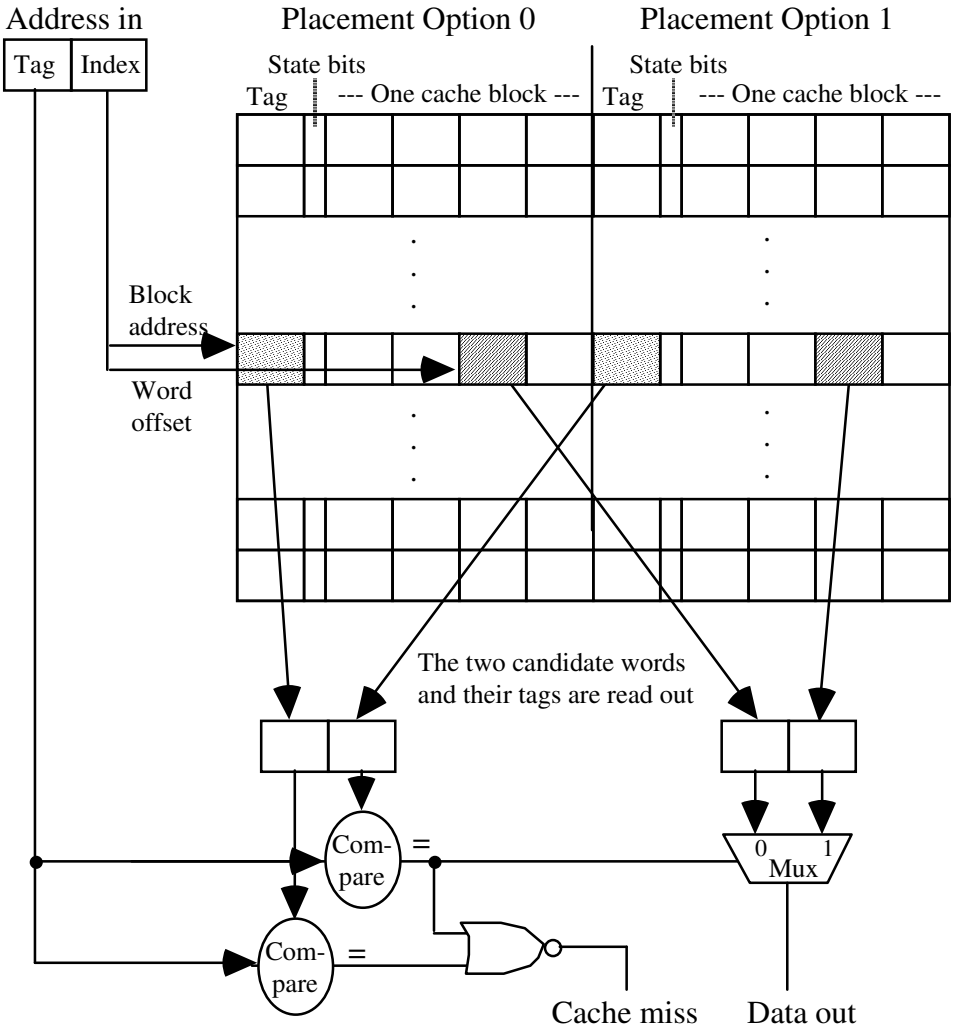


Fig. 18.1. Data storage and access in a two-way set-associative cache.

18.2 Cache Coherence Protocols

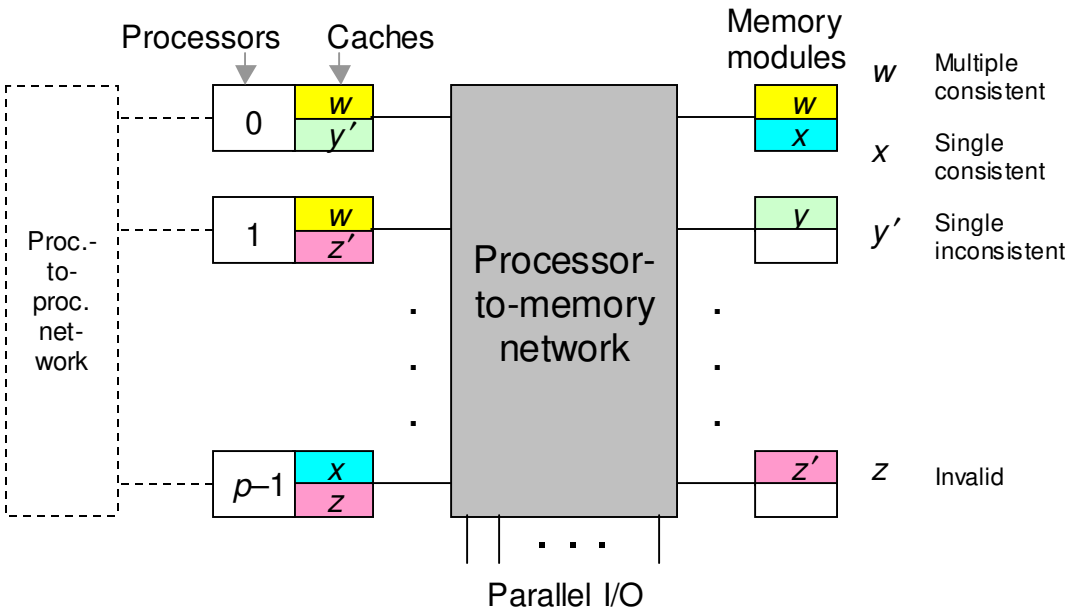


Fig. 18.2. Various types of cached data blocks in a parallel processor with global memory and processor caches.

Example: a bus-based write-invalidate write-back snoopy cache coherence protocol

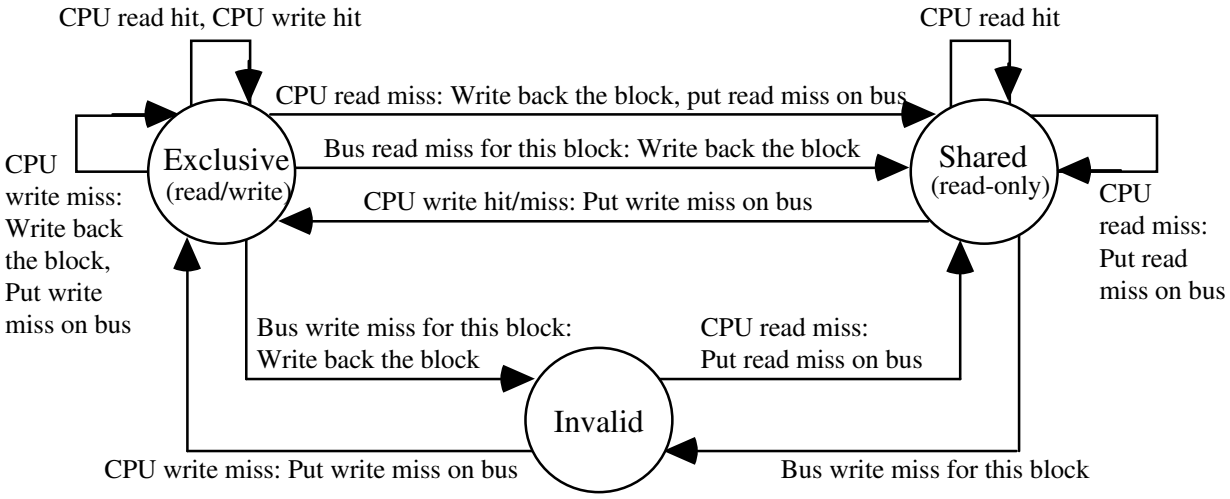


Fig. 18.3. Finite-state control mechanism for a bus-based snoopy cache coherence protocol.

Example: state transition diagram for a directory entry in a directory-based cache coherence protocol

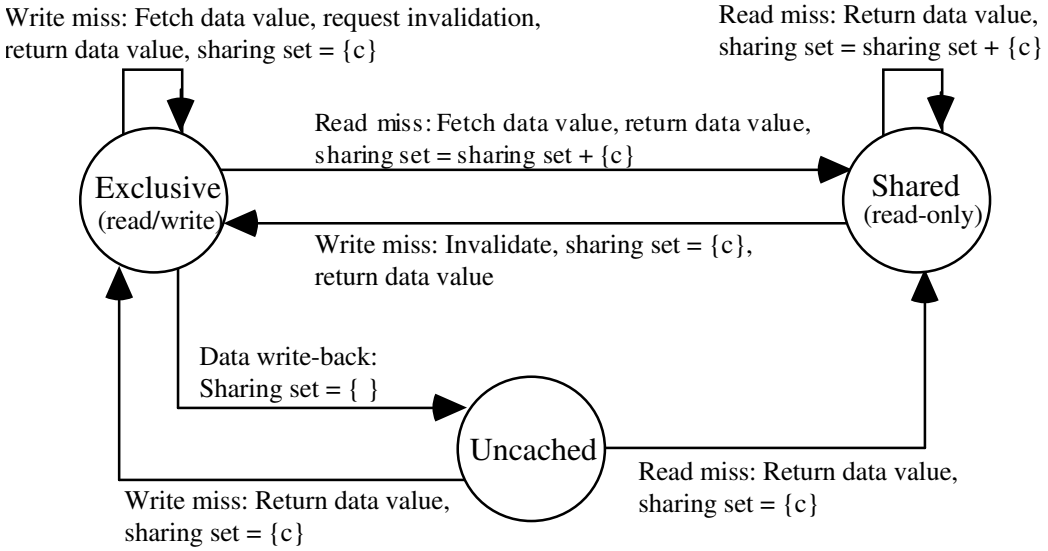


Fig. 18.4. States and transitions for a directory entry in a directory-based coherence protocol (*c* denotes the cache sending the message).

18.3 Multithreading and Latency hiding

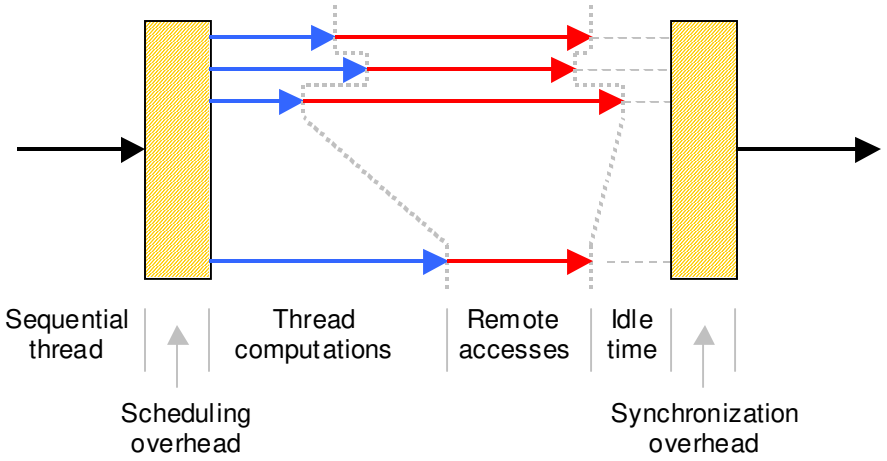


Fig. 18.5. The concept of multithreaded parallel computation.

18.4 Parallel I/O Technology

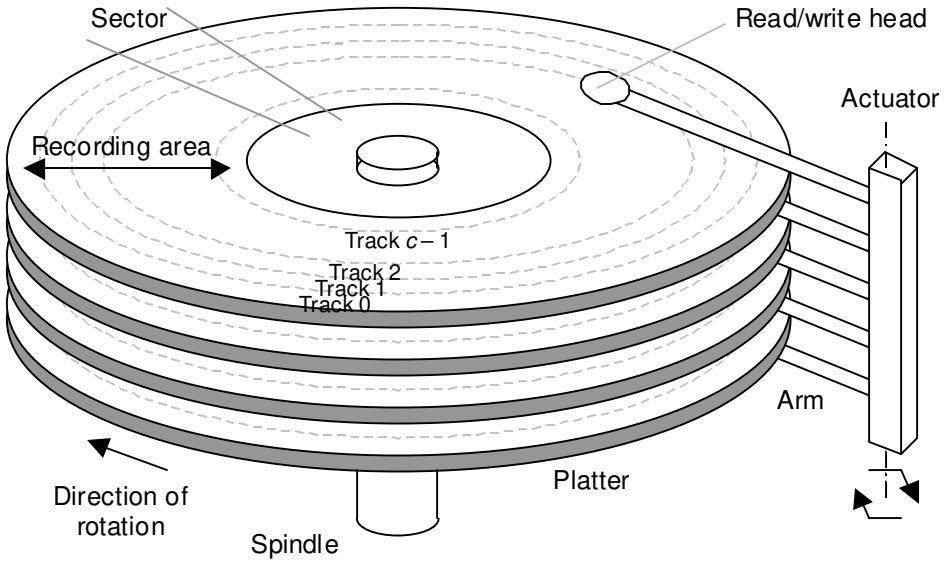


Fig. 18.6. Moving-head magnetic disk elements.

Moving-head disk access =

seek cylinder + rotate to sector + transfer data

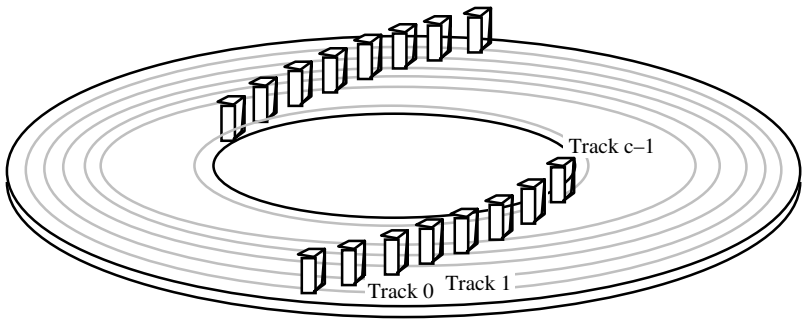
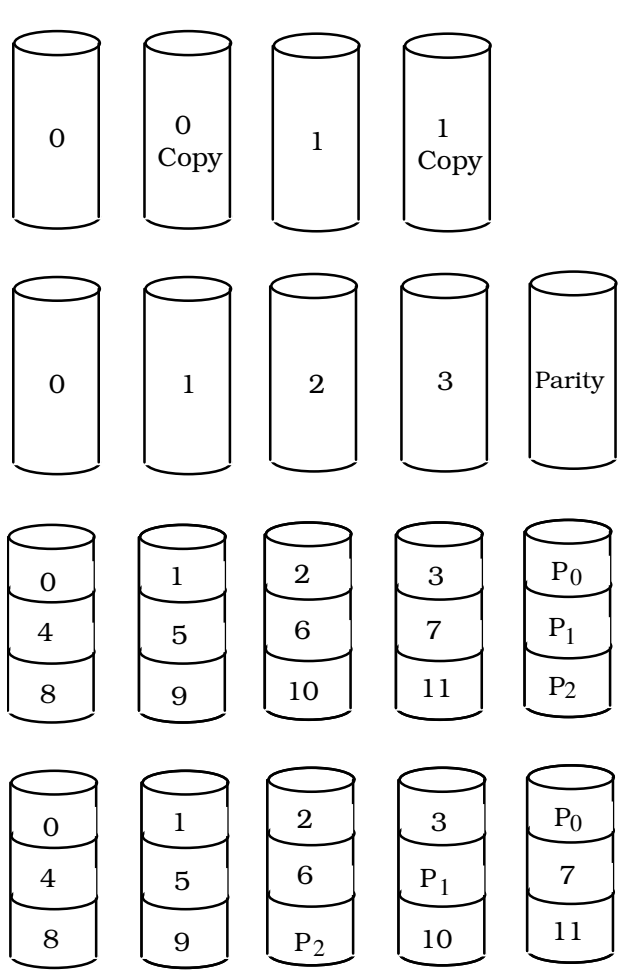


Fig. 18.7. Head-per-track disk concept.

18.5 Redundant Disk Arrays



(RAID 0 used multiple disks for higher data rate; no redundancy)

Mirrored disks

RAID1

(RAID 2 used Hamming code for error correction)

Bit- or byte-level striping with parity or checksum

RAID3

Parity or checksum applied to sectors; parity disk still a performance bottleneck

RAID4

Distributed parity (only "small" writes suffer an overhead)

RAID5

Fig. 18.8. Alternative data organizations on redundant disk arrays.

Computing sector parity for a disk write operation

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

18.6 Interfaces and Standards

Scalable Coherent Interface (SCI) standard

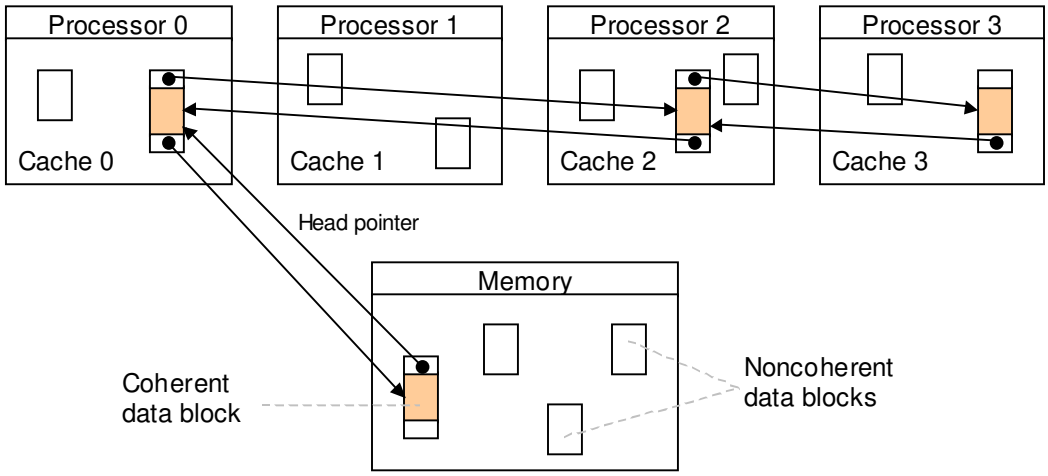


Fig. 18.9. Two categories of data blocks and the structure of the sharing set in the Scalable Coherent Interface.

High-Performance Parallel Interface (HiPPI) standard:

point-to-point connectivity between two devices

(typically a supercomputer and a peripheral)

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

19 Reliable Parallel Processing

Back to TOC

Chapter Goals

- Develop appreciation of reliability problems in parallel systems
- Examine key methods for dealing with such problems at various levels, from circuit redundancy to robustness features for algorithms or applications

Chapter Contents

- 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

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.

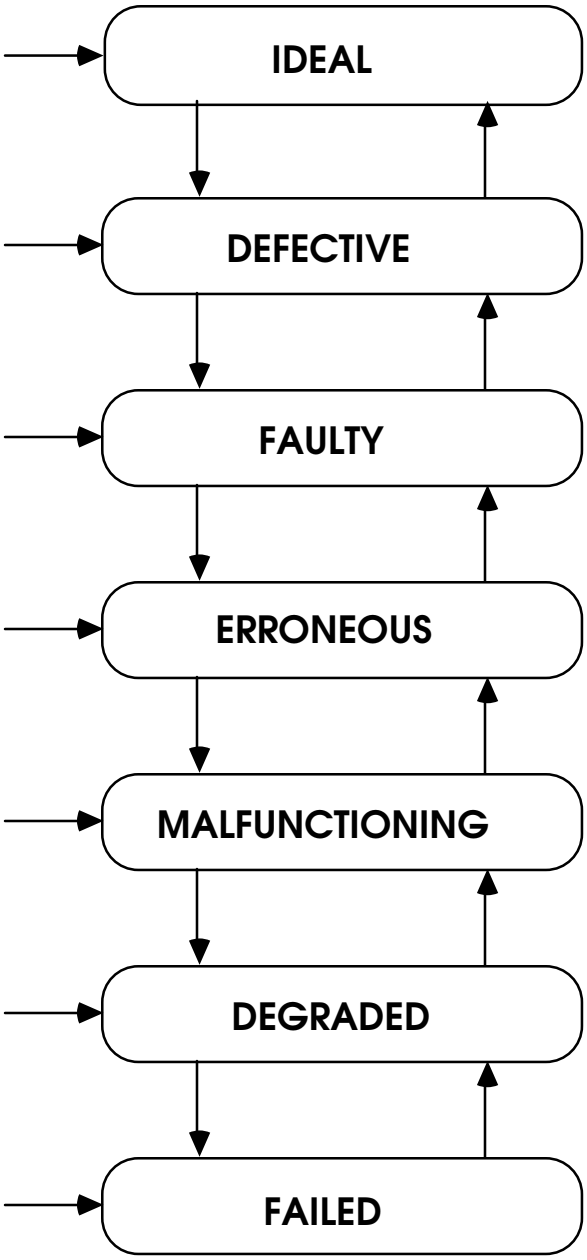


Fig. 19.1. System states and state transitions in our multilevel model.

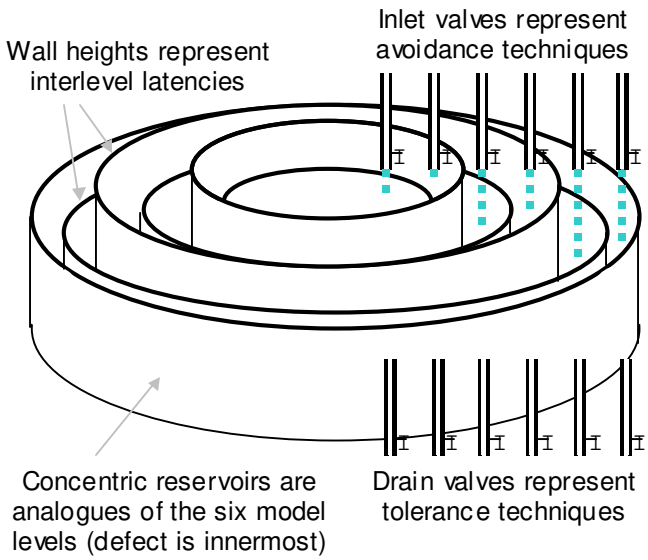


Fig. 19.2. An analogy for the multilevel model of dependable computing.

Fault tolerance in parallel systems

Opportunities:

- multiple resources of same type (built in spares)
- load redistribution
- graceful degradation

Difficulties:

- change in structure due to faults (e.g., 2D mesh)
- bad units disturbing good ones (e.g., on a bus)

19.2 Defect-Level Methods

Defects are caused in two ways (sideways and downward transitions into the defective state of Fig. 19.1)

- Physical design slips leading to defective components
- Component wear/aging or harsh operating conditions

A dormant or ineffective defect is extremely hard to detect

Methods for coping with defects during dormancy

- Periodic maintenance
- Burn-in testing

Goal of defect tolerance methods

- Improving the manufacturing yield
- dynamic reconfiguration during system operation

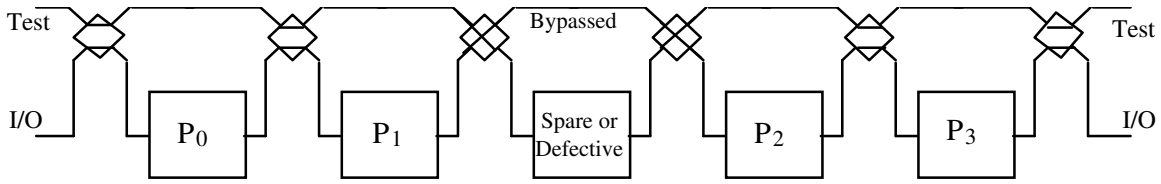


Fig. 19.3. A linear array with a spare processor and reconfiguration switches.

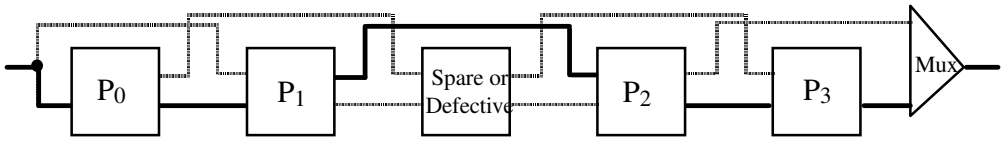


Fig. 19.4. A linear array with a spare processor and embedded switching.

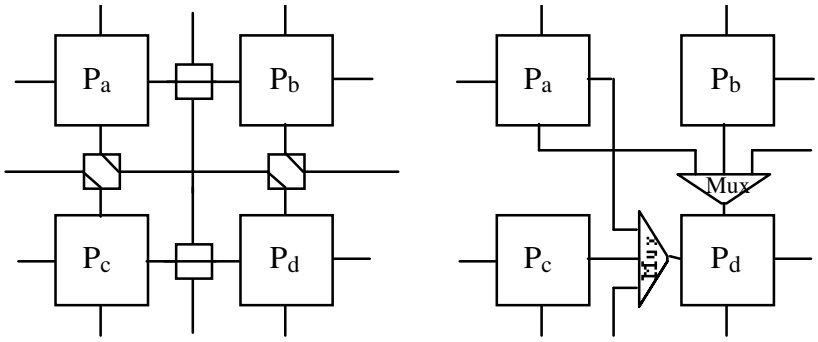


Fig. 19.5. Two types of reconfiguration switching for 2D arrays.

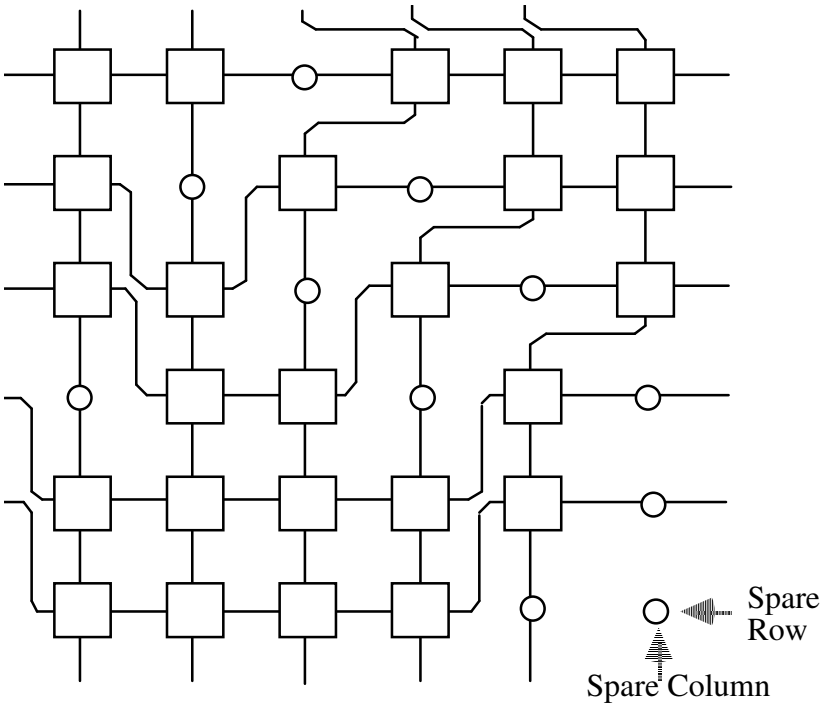


Fig. 19.6. A 5 × 5 working array salvaged from a 6 × 6 redundant mesh through reconfiguration switching.

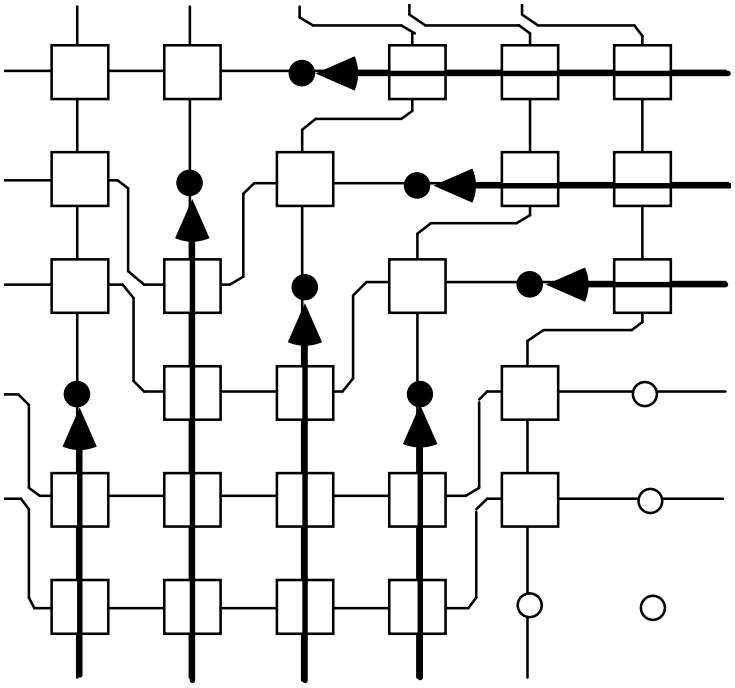
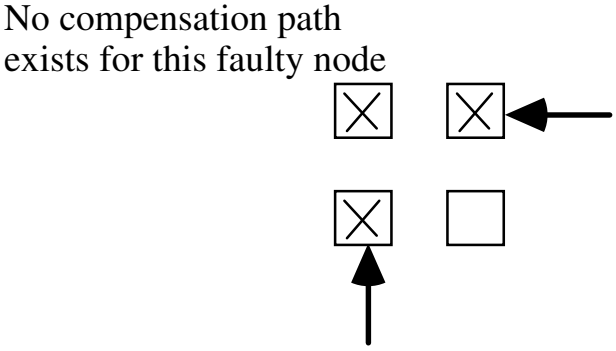


Fig. 19.7. Seven faulty processors and their associated compensation paths.



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

19.3 Fault-Level Methods

Hardware replication

Duplication with comparison

Triplication with voting

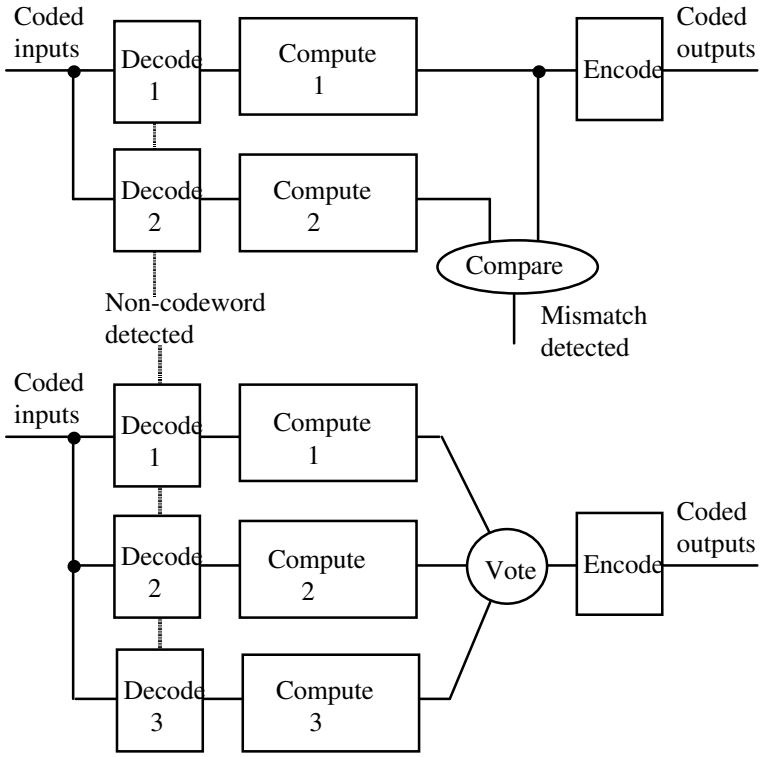


Fig. 19.8. Fault detection or tolerance with replication.

These schemes involve high redundancy: 100 or 200%

Lower redundancy is possible in some cases: e.g., periodic balanced sorters tolerate certain faults with extra stages

Fault detection and bypassing (extra-stage MIN)

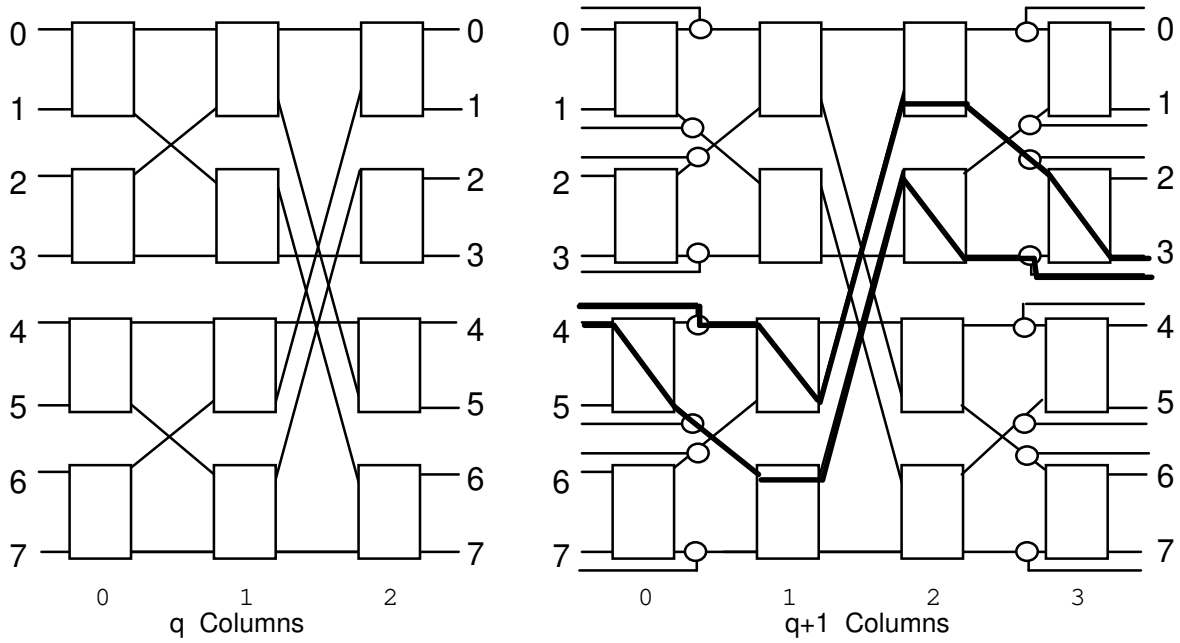


Fig. 19.9. Regular butterfly and extra-stage butterfly networks.

19.4 Error-Level Methods

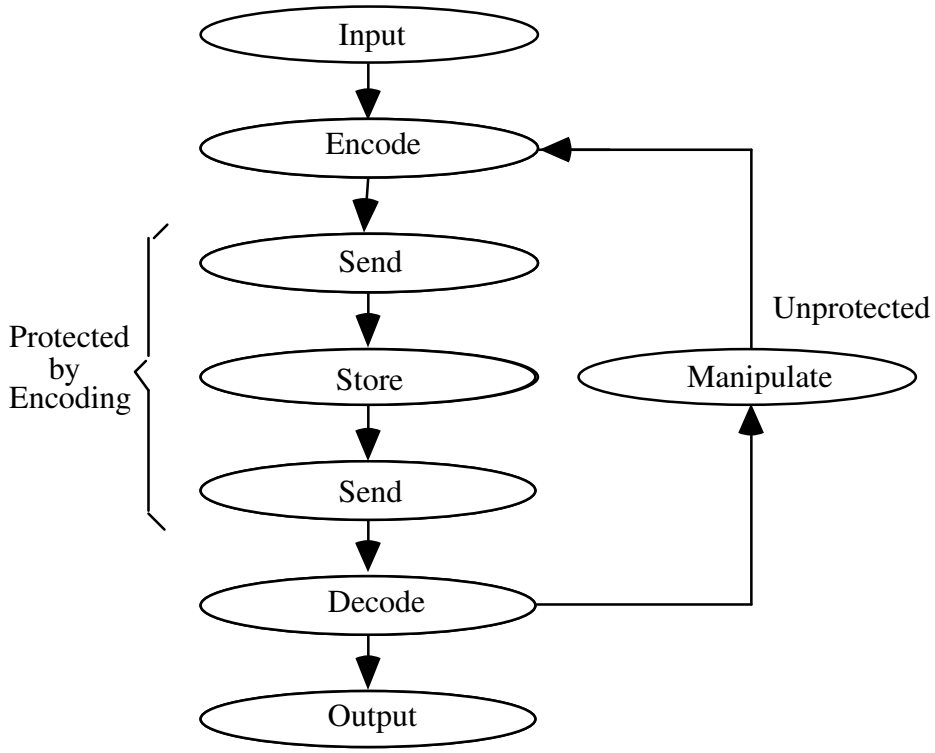


Fig. 19.10. A common way of applying information coding techniques.

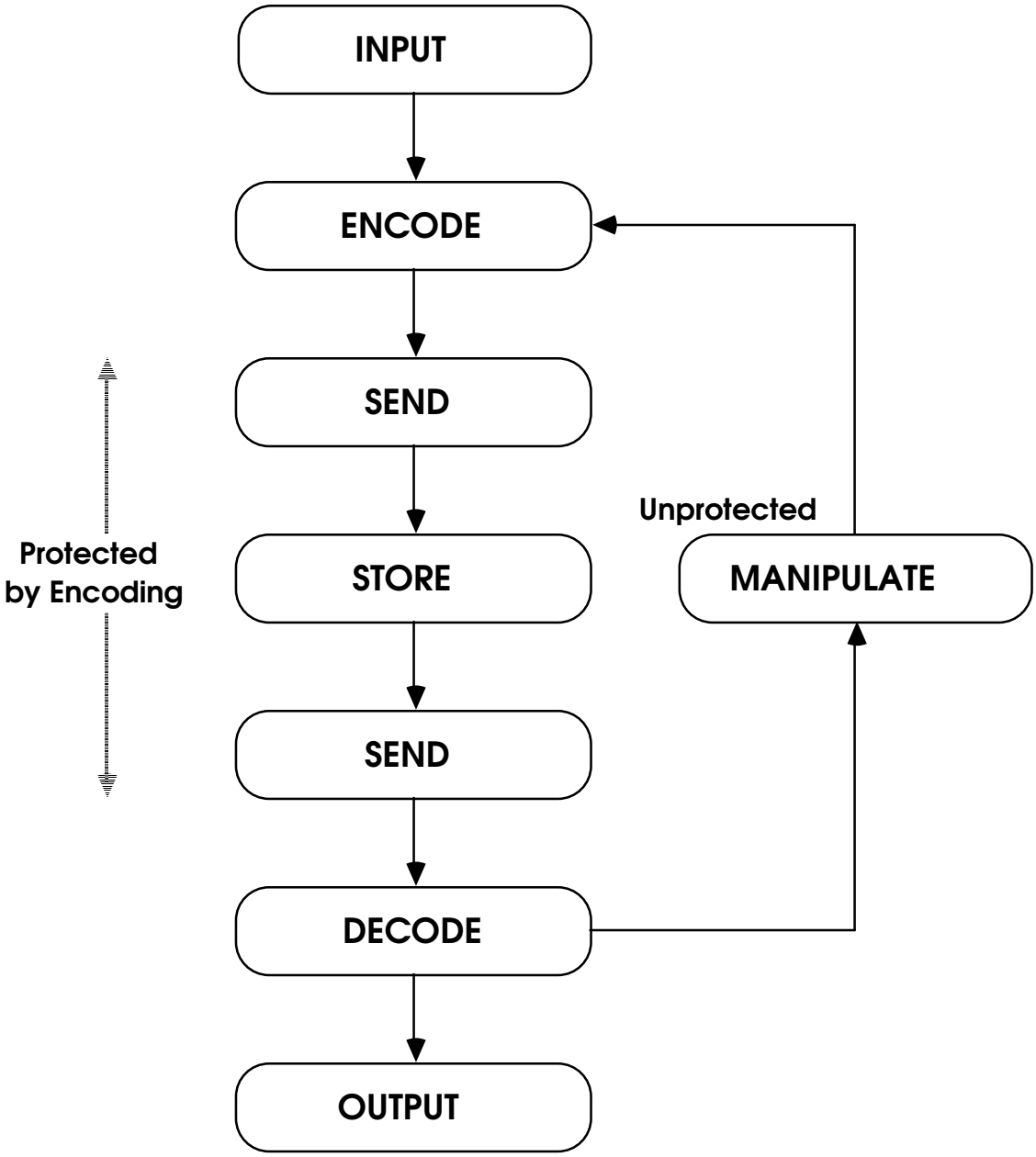


Fig. 19.10. A common way of applying information coding techniques.

Special coding methods; e.g., arithmetic codes

Robust data structures

Algorithm-based error tolerance

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

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

If X, Y, and Z are matrices satisfying $Z = X \times Y$

$$Z_f = X_c \times Y_r$$

In a full-checksum matrix, any single erroneous element can be corrected and any three erroneous elements can be detected

19.5 Malfunction-Level Methods

System-level testing and diagnosis

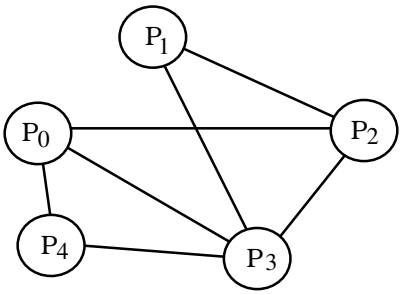
Start from a core and expand to the whole system

Modules test each other and draw inferences from results

The theory of malfunction diagnosis

Given a diagnosis matrix, identify:

- all malfunctioning units
- at least one malfunctioning unit
- a subset guaranteed to contain all malfunctions



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

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

Low-redundancy sparing

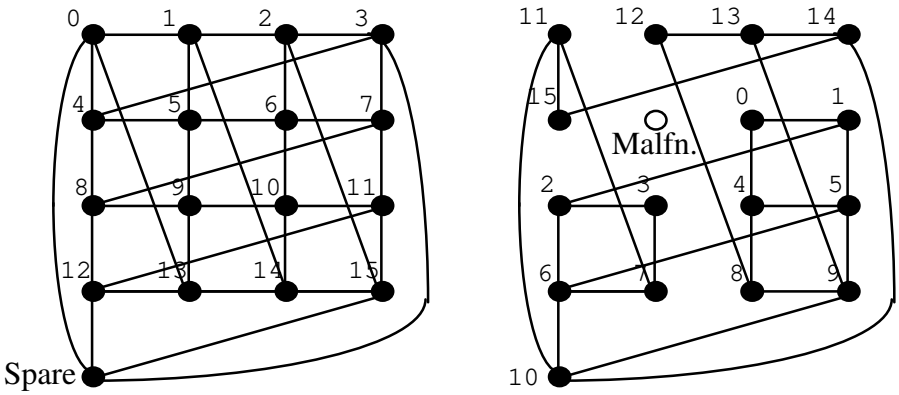


Fig. 19.12. Reconfigurable 4×4 mesh with one spare.

19.6 Degradation-Level Methods

Fig. 19.13 depicts the performance variations in three types of parallel systems:

S_1: 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

S_2: fail-soft system with gradually degrading performance level and off-line repair at time t_2

S_3: fail-soft system with on-line repair which, from the viewpoint of an application that requires a performance level of at least P_{min}, postpones its failure time to t_3

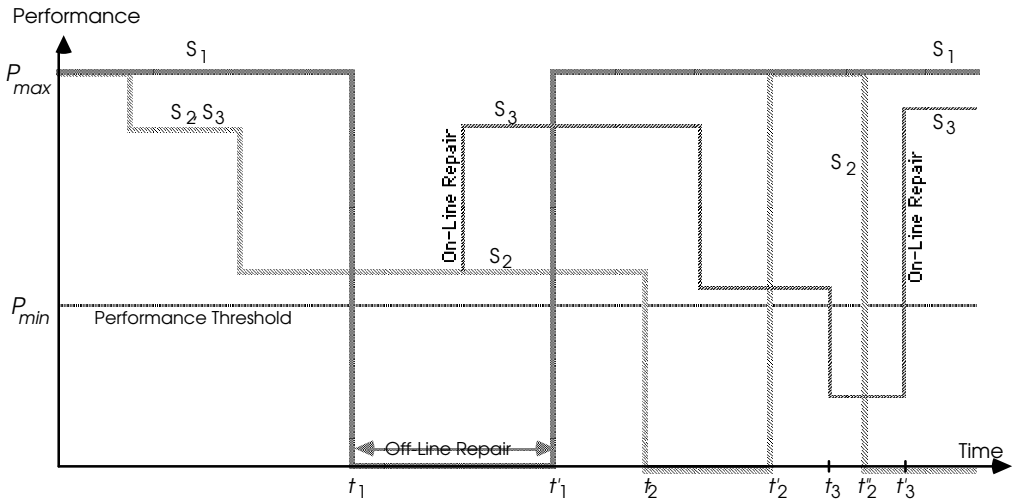


Fig. 19.13. Performance variations in three example parallel computers.

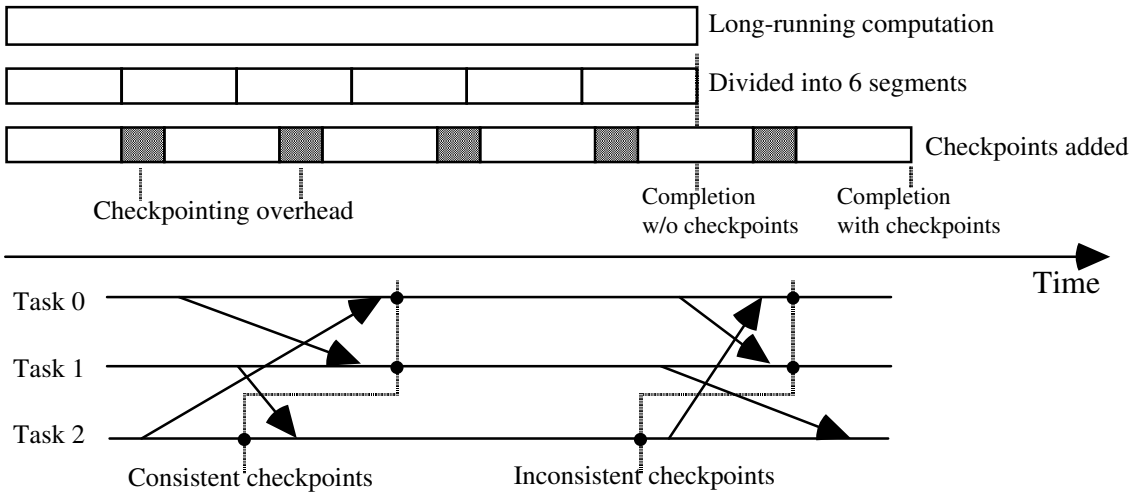


Fig. 19.14. Checkpointing, its overhead, and pitfalls.

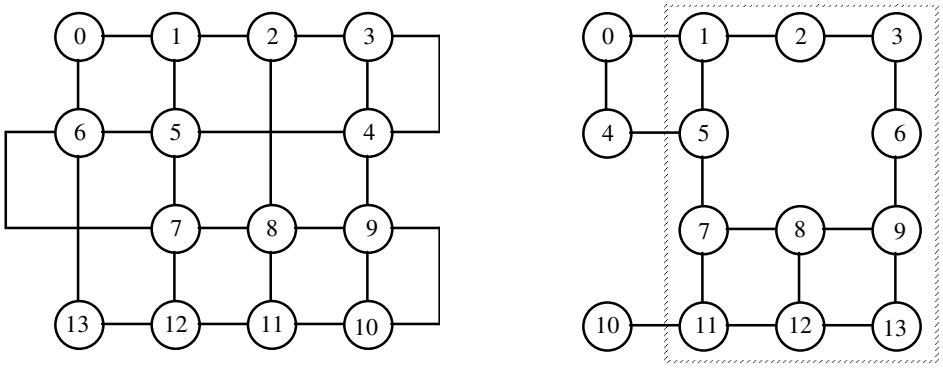


Fig. 19.15. Two types of incomplete meshes, with and without bypass links.

A system fails when its degradation tolerance capacity is exhausted and, as a result, its performance falls below an acceptable threshold

As degradations are themselves consequences of malfunctions, it is interesting to skip a level and relate system failures directly to malfunctions

It has been noted that failures in a gracefully degrading system can be attributed to:

a. Isolated malfunction of a critical subsystem

b. Catastrophic (multiple space-domain) malfunctions

c. Accumulation of (multiple time-domain) malfunctions

d. Resource exhaustion causing inadequate performance or total shutdown

20 System and Software Issues

Back to TOC

Chapter Goals

- Deal with some system, software, and application topics so that there isn't a complete void in these areas
- Review key issues and introduce references for further study on these topics

Chapter Contents

- 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

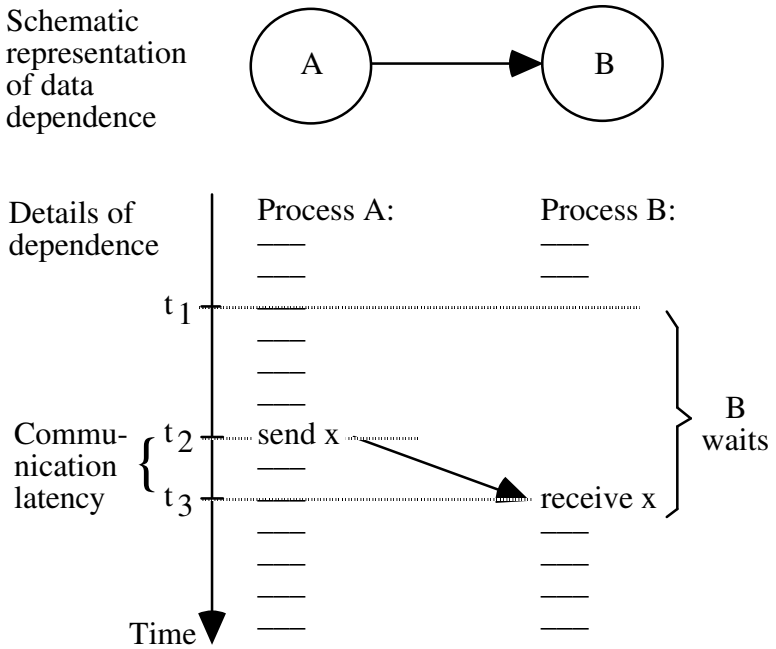


Fig. 20.1. Automatic synchronization in message-passing systems.

With shared memory, synchronization is accomplished by accessing specially designated shared control variables

A popular way is through atomic fetch-and-add instruction

The fetch-and-add instruction has two parameters:

A shared variable *x* and an increment *a*

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 the atomicity of fetch-and-add is important

Consider the following timing of events if each of two processes were to execute fetch-and-add by

reading the *x* value from memory into an accumulator

adding its increment to the accumulator

storing the sum back into *x*

The three steps of fetch-and-add for the two processes may be interleaved in time 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*

This leads to incorrect semantics, as both processes receive the same value *c* in return and the final value of *x* in memory will be *c* + *b* rather than *c* + *a* + *b*

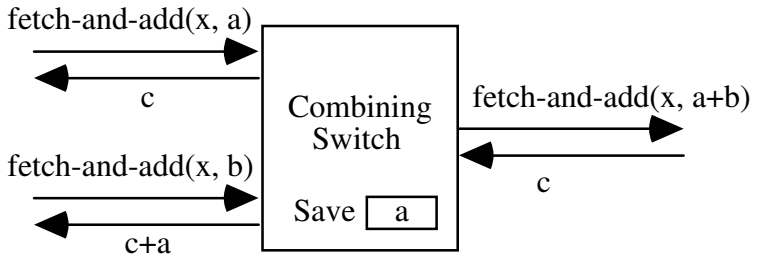


Fig. 20.2. Combining of two fetch-and-add requests.

Barrier synchronization: A processor, in a designated set, must wait at a barrier until each of the other processors has arrived at the corresponding point in its computation

Strategy 1: Reduce the synchronization overhead

Using a single AND tree: if it is possible for a processor to be randomly delayed between raising it flag and checking the AND tree output, then some processors might cross the barrier and lower their flags before others have had a chance to examine the AND tree output

Using two AND trees that are connected to the set and reset inputs of a flip-flop

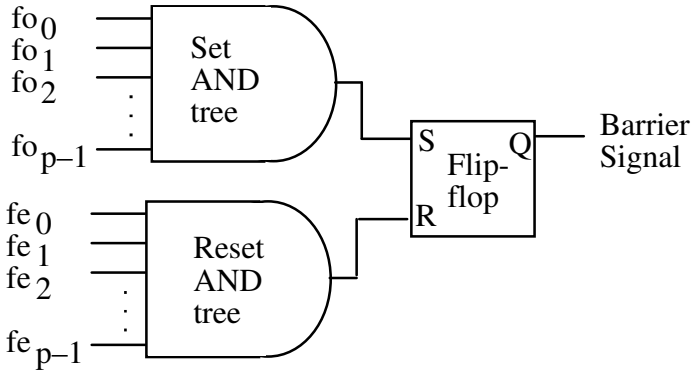


Fig. 20.4. Example of hardware aid for fast barrier synchronization [Hoar96].

Once we provide a mechanism like Fig. 20.4 for barrier synchronization, it is only a small step to generalize it to a "global combine" (semigroup computation) facility

The AND tree implements a semigroup computation using the binary AND operator. The generalization might involve doing OR and XOR logical reductions as well

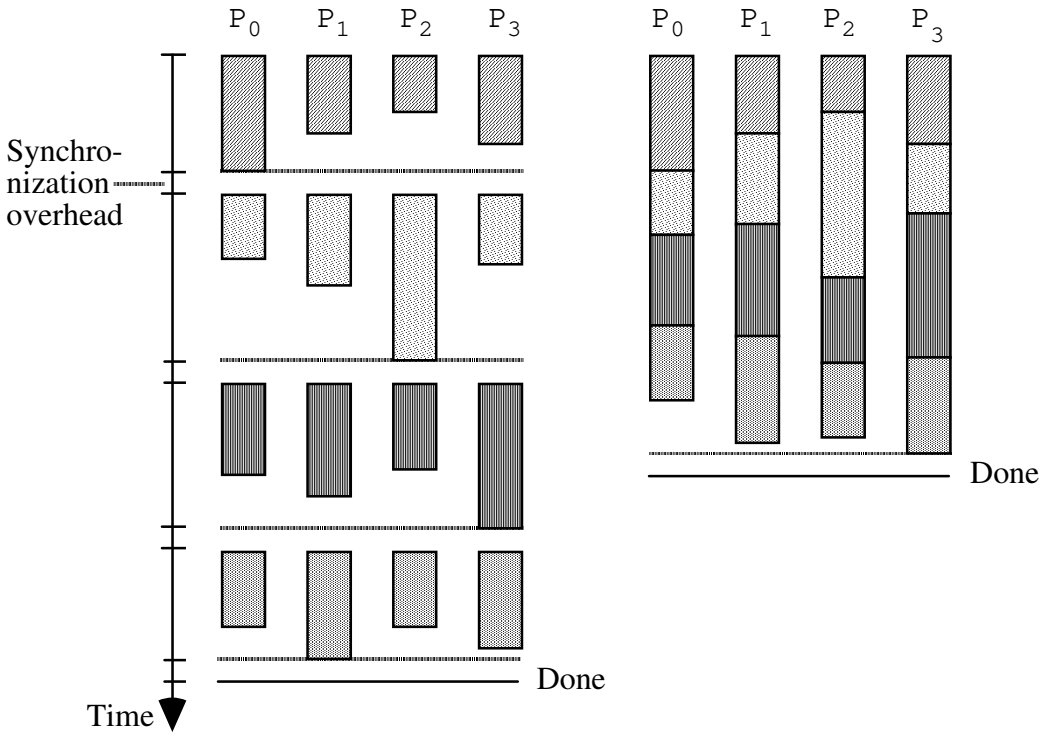


Fig. 20.3. The performance benefit of less frequent synchronization.

Strategy 2: Perform less frequent synchronization

Bulk-synchronous parallel (BSP) mode of computation

Synchronization of processors occurs once every *L* time steps, where *L* is a periodicity parameter

A parallel computation consists of a sequence of supersteps

In one superstep, each processor performs a task composed of local computation, message transmissions, and message receptions from other processors

Data received in messages will not be used in the current super-step but rather beginning with the next super-step

20.2 Parallel Programming

Approaches to parallel program development:

a. Parallelizing compilers
b. Data-parallel programming
c. Shared-variable programming
d. Communicating processes
e. Functional programming

Parallelizing compiler

Each iteration of the *i* loop below can be assigned to a different processor for asynchronous execution; successive iterations are totally independent

```
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
```

The irony in parallelizing compilers:

Force a naturally parallel computation into sequential mold

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

Data-parallel programming

The APL programming language

$C \leftarrow A + B$ array add

$x \leftarrow +/V$ reduction

$U \leftarrow +/V \times W$ inner product

A write-only language?

Fortran-90 (superset of Fortran-77)

Extensions that include facilities for array operations

```
A = SQRT(A) + B ** 2

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)              gather operation
```

High-performance Fortran (HPF) extends Fortran-90 by

- adding new directives and language constructs
- imposing some restrictions for efficiency reasons

HPF includes a number of compiler directives that assist the compiler in data distribution

These directives, which do not alter the semantics of the program, are presented as Fortran-90 comments (begin with the comment symbol "!")

If an HPF program is presented to a Fortran-90 compiler, it will be compiled, and subsequently executed, correctly

As an example, the HPF statement

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

is a hint to the compiler that it should distribute the elements of arrays A and B among processors or memory banks such that A(I) and B(I + 2) are stored together

If this statement is ignored, the program will still execute correctly, but perhaps less efficiently.

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

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

Shared-variable programming

Concurrent Pascal, Modula-2, Sequent C

Communicating processes

Languages: Ada, OccamLanguage-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

Thus, computations have the property of referential transparency or freedom from side effects

Due to inefficiencies inherent in the single-assignment approach, practical application of functional programming has been limited to

- Lisp-based systems (MIT's Multilisp)
- Data-flow architectures (Manchester U's SISAL)

20.3 Software Portability and Standards

Portable parallel applications elusive

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 architectures 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

Message passing interface (MPI) standard

Specifies a library of functions that implement the message-passing model of parallel computation

Was developed by the MPI Forum, a consortium of parallel computer vendors and software development specialists

As a standard, 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 for:

Point-to-point communication
(Blocking and non-blocking 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)

Software platform for developing and running parallel applications on a set 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 of 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 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 take group names as arguments
e.g., broadcast and barrier synchronization

20.4 Parallel Operating Systems

Classes of parallel processors:

Back-end, front-end, stand-alone

Back-end system: the 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, 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 post-processing functions, diagnostic testing, and interface with the 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

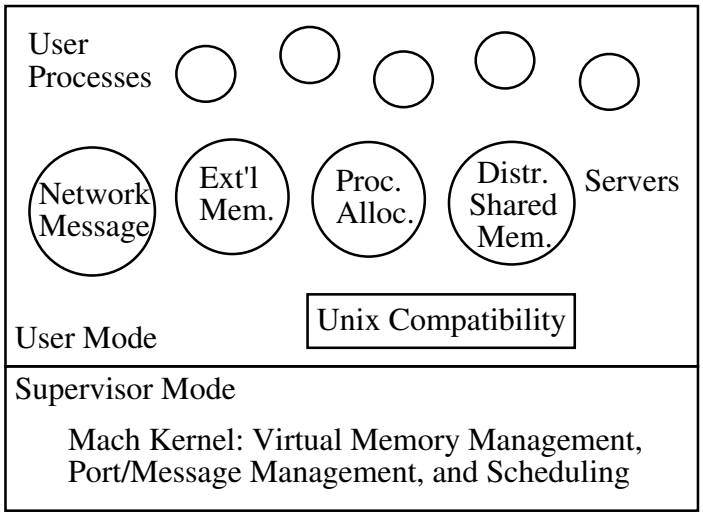


Fig. 20.5. Functions of the supervisor and user modes in the Mach operating system.

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 communication ports

b. Thread: An executing 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

Unlike Unix whose memory consists of contiguous areas, the virtual address space in Mach is composed of individual pages with separate protection and inheritance

Messages in Mach are communicated via ports

Messages are typed to indicate the data type they carry and can be communicated over a port only if the sending/receiving thread has the appropriate access rights

For efficiency purposes, messages that involve 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 does not occur until the receiver accesses the data

So, even though a message may refer to an extensive data set, only the segments actually referenced by the receiver will ever be copied

The Mach scheduler has some interesting features

Each thread is assigned 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, priorities are reduced based on "age"; the more CPU time a thread uses, the lower its priority becomes. This policy not only prevents starvation, but also tends to favor interactive tasks over computation-intensive ones

20.5 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

To avoid a performance bottleneck, a parallel file system must itself be a highly parallel and scalable program that efficiently deals 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

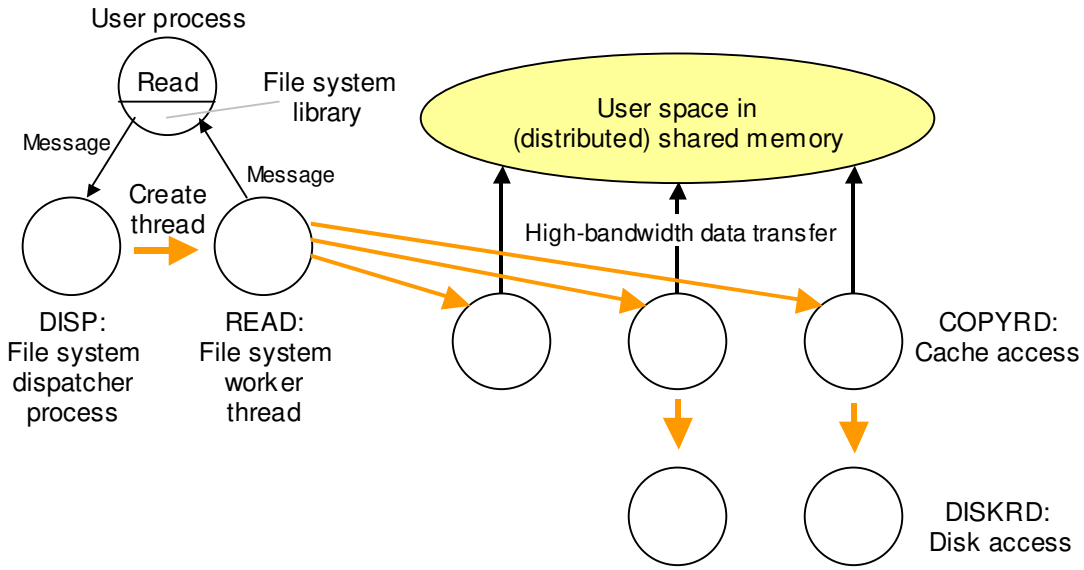


Fig. 20.6. Handling of a large read request by a parallel file system [Hell93].

20.6 Hardware/Software Interaction

A parallel application program should be executable, with little or no modification, on a variety of parallel 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, the upgrade should be done by simply plugging in new processors, along with interconnects, and rebooting

Thus, workstation clusters are ideal in that they are readily scalable both in time and space

Scalability in time: introduction of faster workstations and interconnects leads to a corresponding increase in system performance with little or no redesign

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)

Scalability in time is difficult at present but may be made possible in future through the adoption of implementation and interfacing standards

Users are also interested in software/application scalability (for degradation tolerance and/or portability)

Scaled speedup and isoefficiency are relevant here

We use parallel processing not just to speed up the solution of fixed problems but also to make the solution of larger problems feasible with realistic turn-around times

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

$$S(n, p) = \frac{T(n, 1)}{T(n, p)}$$

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

$$C(n, p) = T(n, 1) \qquad H(n, p) = pT(n, p) - T(n, 1)$$

$$S(n, p) = \frac{p}{1 + H(n, p)/T(n, 1)}$$

$$E(n, p) = S(n, p)/p = \frac{1}{1 + H(n, p)/T(n, 1)}$$

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

$$S(n, p) = \frac{p}{1 + pf} < 1/f \qquad E(n, p) = \frac{1}{1 + pf}$$

Assume that efficiency is to be kept above 1/2, but the arguments apply to any fixed efficiency target

To have $E(n, p) > 1/2$, we need $pf < 1$ or

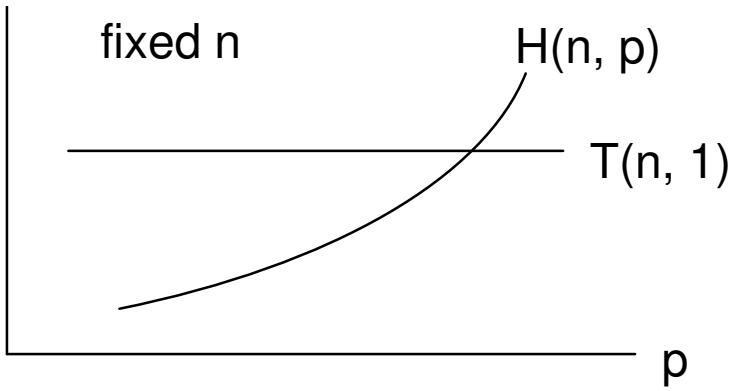
$$p < 1/f$$

That is, for a fixed problem size and under the assumption of the per-processor overhead being a fixed fraction of the single-processor running time, there is an upper limit to the number of processors that can be applied cost-effectively

Going back to our initial efficiency equation, we note that keeping $E(n, p)$ above 1/2 requires:

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

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



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

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

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

With the above provisions, a scaled speedup of $p/2$ or more is achievable for problems of suitably large size

Note, however, that the parallel execution time

$$T(n, p) = \frac{T(n, 1) + H(n, p)}{p}$$

grows as we scale up the problem size to obtain good efficiency

Thus, there is a limit to the usefulness of scaled speedup

In particular, when there is a fixed computation time available due to deadlines (as in daily or weekly weather forecasting), the ability to achieve very good scaled speedup may be irrelevant

Part VI Implementation Aspects

Back to TOC

Part Goals

- Study real parallel machines, MIMD & SIMD
- Learn about parallel machines that
  - are of historical significance
  - incorporate key ideas, influencing the development of parallel processing
  - are currently in production and/or use
- Put our knowledge in historical context

Part Contents

- Chapter 21: Shared-Memory MIMD Machines
- Chapter 22: Message-Passing MIMD Machines
- Chapter 23: Data-Parallel SIMD Machines
- Chapter 24: Past, Present, and Future

21 Shared-Memory MIMD Machines

Back to TOC

Chapter Goals

- Survey topics pertaining to the practical implementation and performance of shared memory
- Case studies of research prototypes and production machines that use global or distributed shared memory

Chapter Contents

- 21.1. Variations in Shared Memory
- 21.2. MIN-Based BBN Butterfly
- 21.3. Vector-Parallel Cray Y-MP
- 21.4. Latency-Tolerant Tera MTA
- 21.5. CC-NUMA Stanford DASH
- 21.6. SCI-Based Sequent NUMA-Q

21.1 Variations in Shared Memory

	Single Copy of Modifiable Data	Multiple Copies of Modifiable Data
Central Main Memory	**UMA** BBN Butterfly Cray Y-MP	**CC-UMA**
Distributed Main Memory	**NUMA** Tera MTA	**COMA** **CC-NUMA** Stanford DASH Sequent NUMA-Q

Fig. 21.1. Classification of shared-memory hardware architectures and example systems that will be studied in the rest of this chapter.

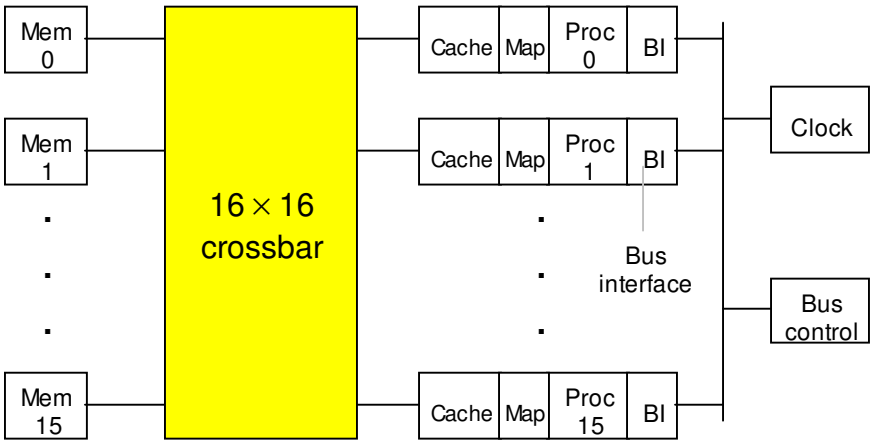


Fig. 21.2. Organization of the C.mmp multiprocessor.

Shared-memory consistency models:

a Sequential consistency (strictest and most intuitive); it mandates that interleaving of reads and writes be the same from the viewpoint of all processors. This provides the illusion of a FCFS single-port memory.

b Processor consistency (less strict); it only mandates that writes be observed in the same order by all processors. This allows reads to overtake writes, providing better performance due to optimizations afforded by out-of-order execution.

c Weak consistency separates ordinary memory accesses from synchronization accesses and only mandates that memory become consistent on synchronization accesses. Synch accesses must wait for completion of all previous accesses, while ordinary read and write accesses can proceed as long as there is no pending synch access.

d Release consistency is similar to weak consistency but recognizes two synch accesses, called "acquire" and "release", with protected shared accesses sandwiched between them. Ordinary read/write accesses can proceed only when there is no pending acquire access from the same processor and a release access must wait for all reads and writes to be completed.

21.2 MIN-Based BBN Butterfly

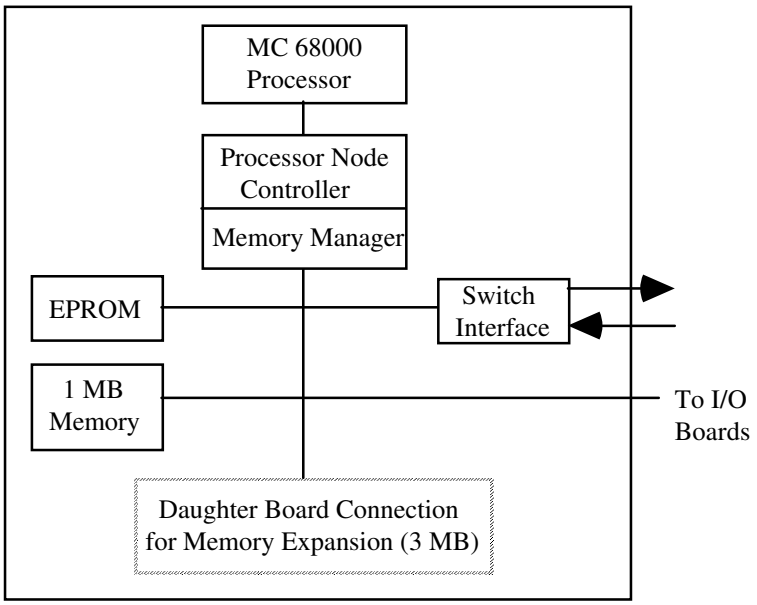


Fig. 21.3. Structure of a processing node in the BBN Butterfly.

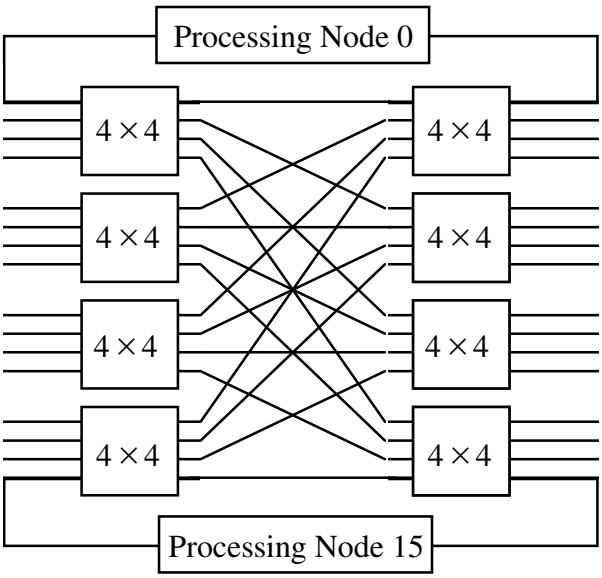


Fig. 21.4. A small 16-node version of the multistage interconnection network of the BBN Butterfly.

21.3 Vector-Parallel Cray Y-MP

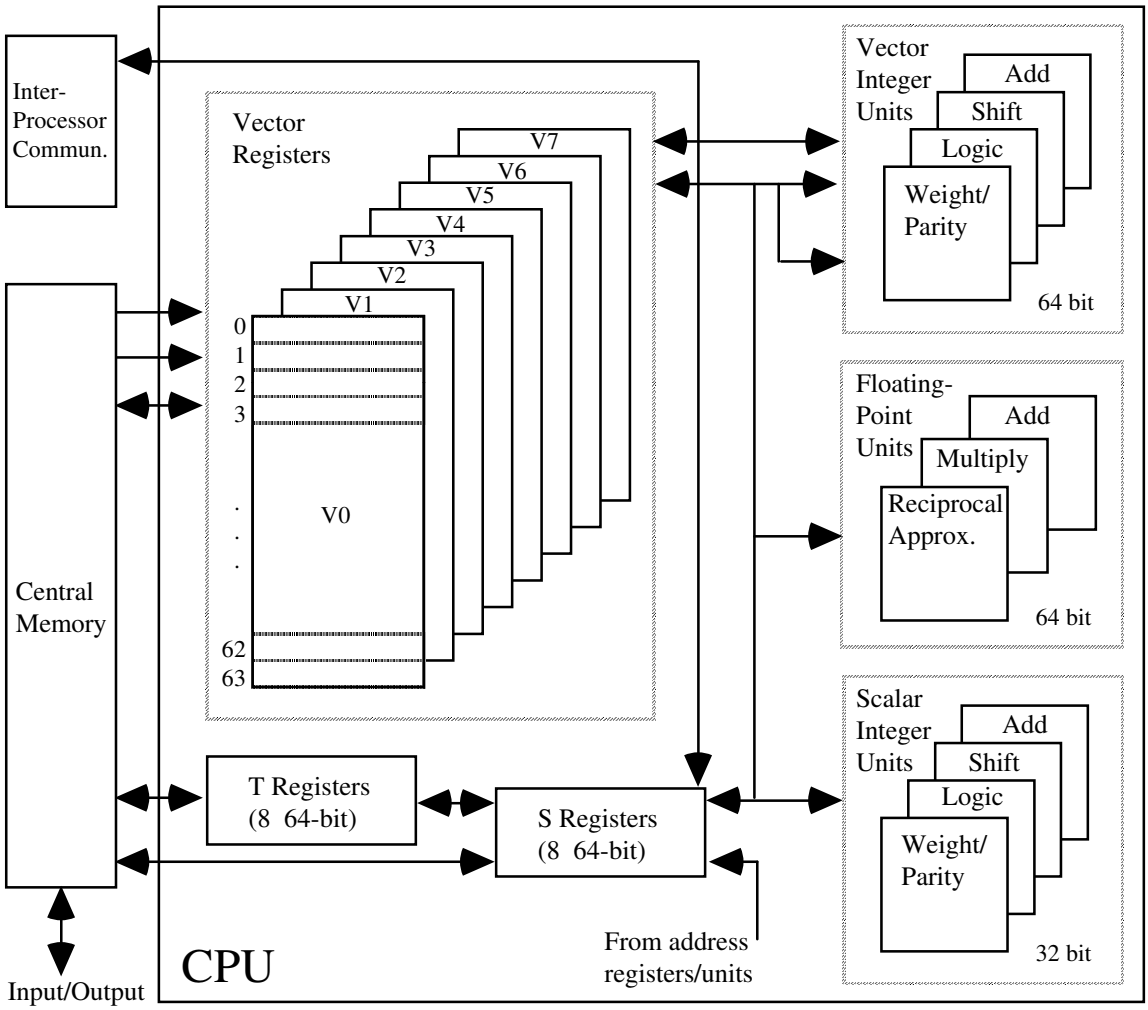


Fig. 21.5. Key elements of the Cray Y-MP processor. Address registers, address function units, instruction buffers, and control not shown.

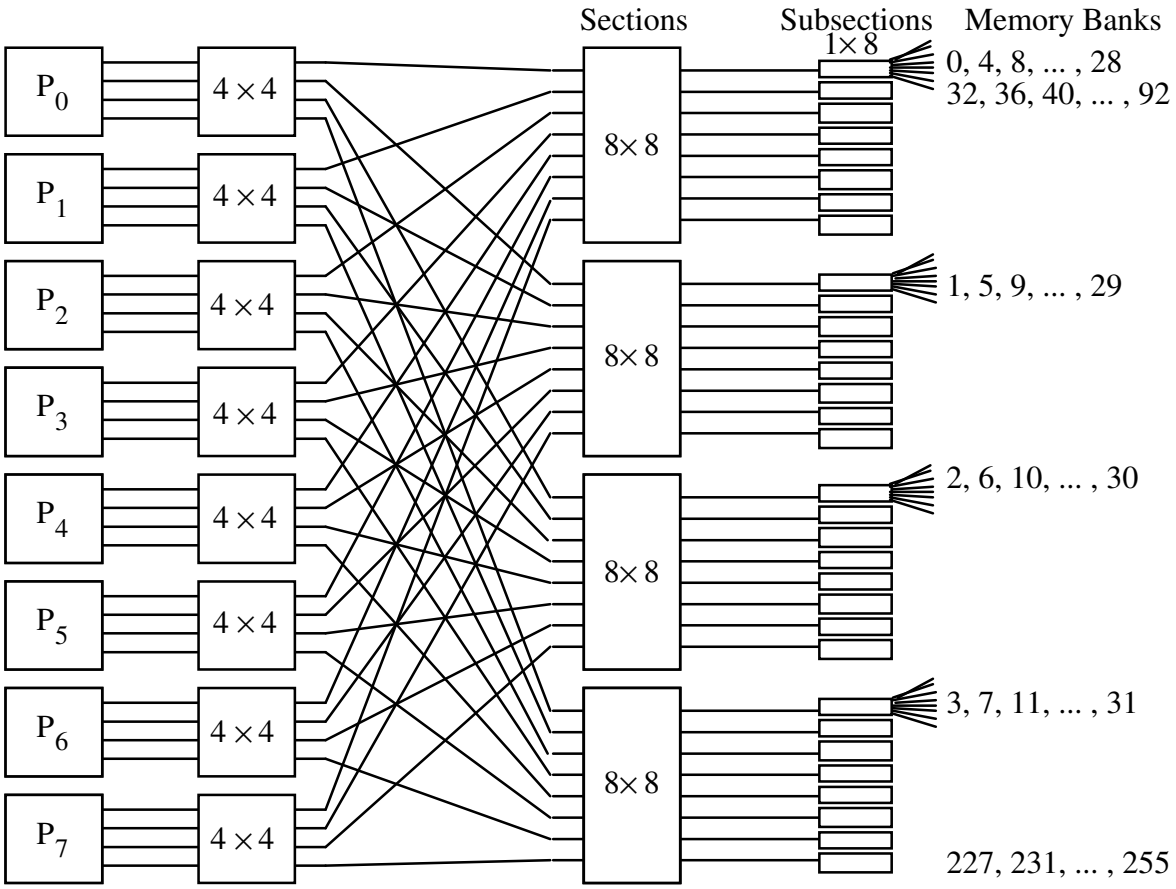


Fig. 21.6. The processor-to-memory interconnection network of Cray Y-MP.

21.4 Latency-Tolerant Tera MTA

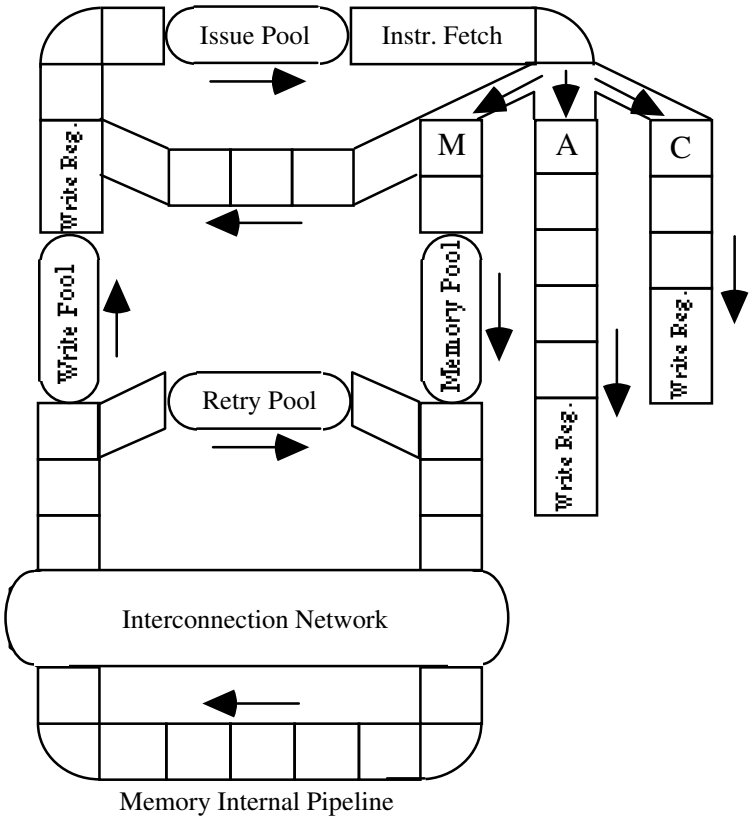


Fig. 21.7. The instruction execution pipelines of Tera MTA.

21.5 CC-NUMA Stanford Dash

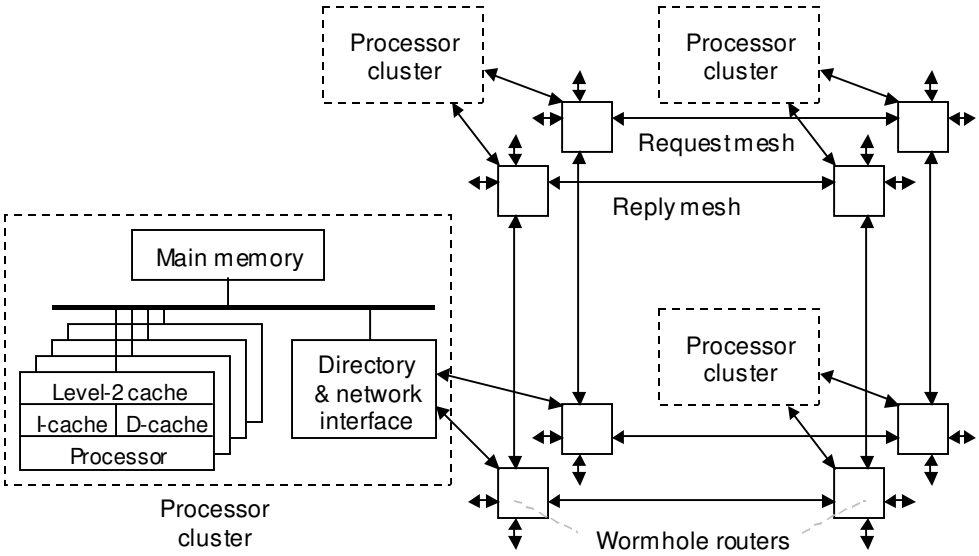


Fig. 21.8. The architecture of Stanford DASH.

21.6 SCI-Based Sequent NUMA-Q

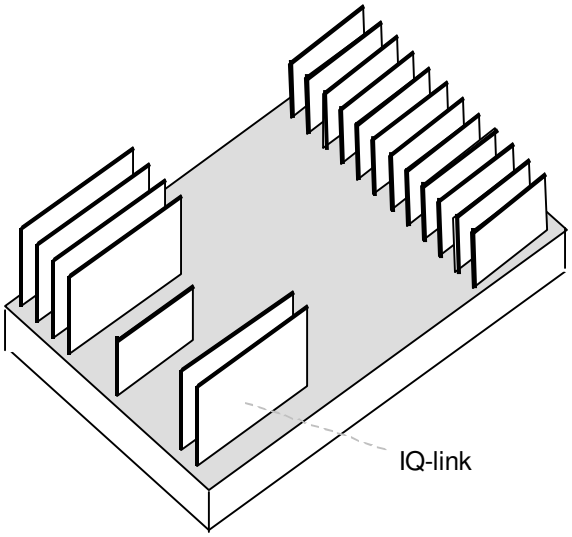


Fig. 21.9. The physical placement of Sequent's quad components on a rackmount baseboard (not to scale).

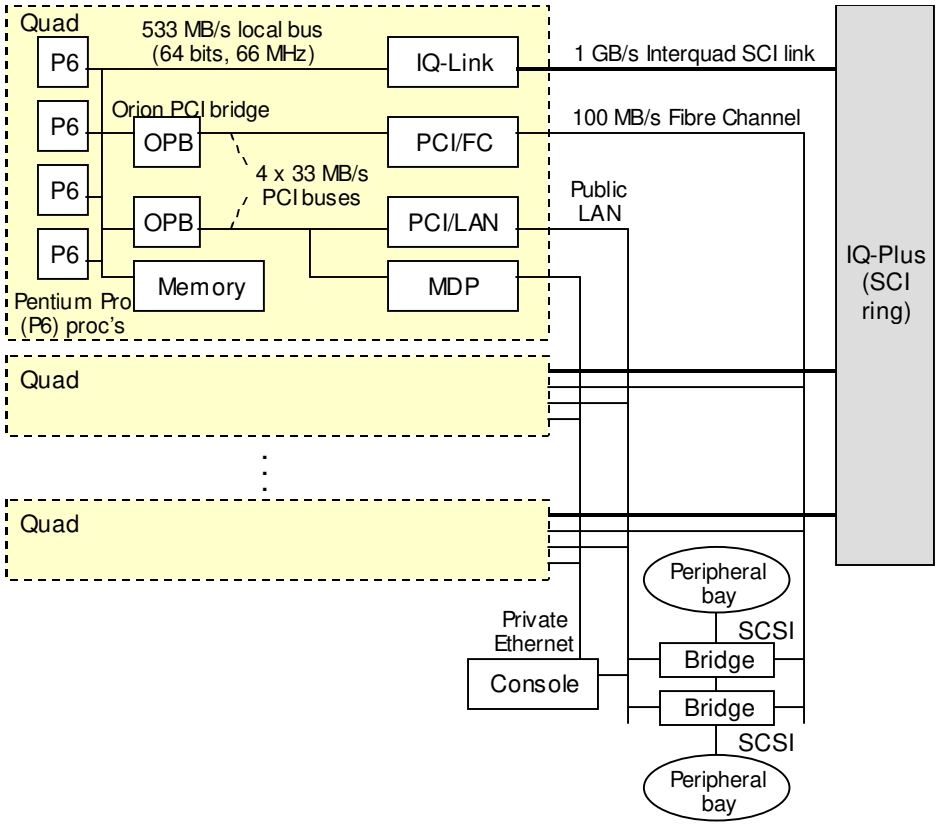


Fig. 21.10. The architecture of Sequent NUMA-Q 2000.

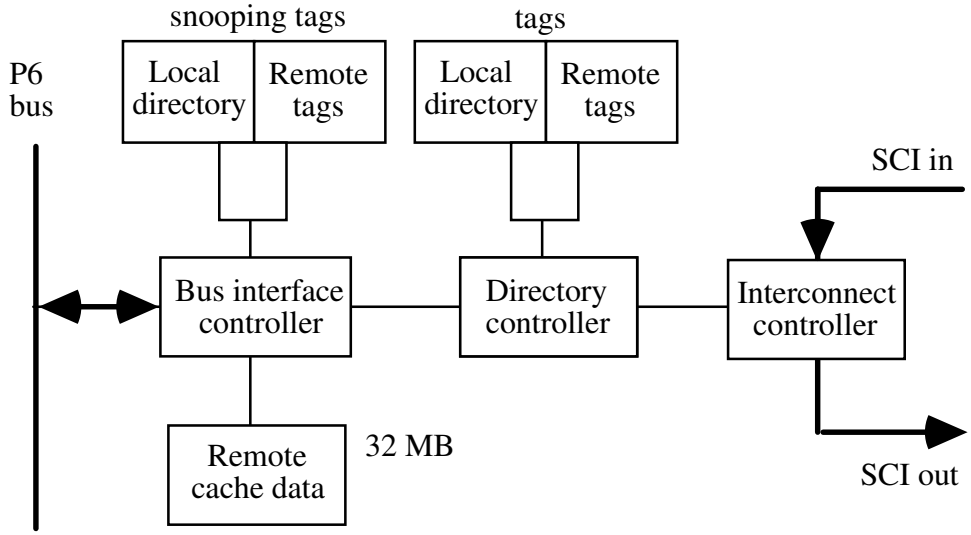


Fig. 21.11. Block diagram of the IQ-Link board.

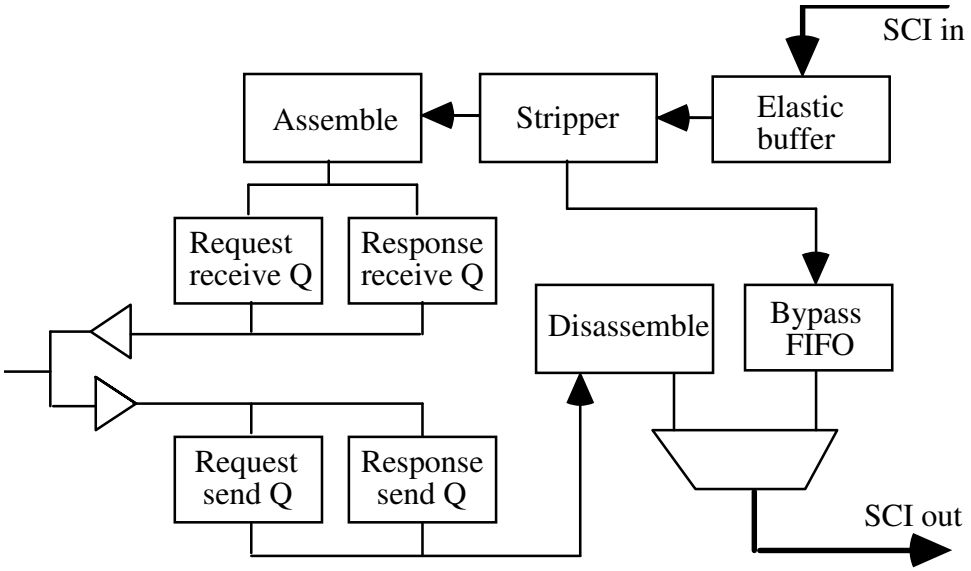


Fig. 21.12. Block diagram of IQ-Link's interconnect controller.

22 Message-Passing MIMD Machines

Back to TOC

Chapter Goals

- Survey topics pertaining to the practical implementation and performance of message passing mechanisms
- Case studies of research prototypes and production machines that use explicit message passing for communication

Chapter Contents

- 22.1. Mechanisms for Message Passing
- 22.2. Reliable Bus-Based Tandem NonStop
- 22.3. Hypercube-Based nCUBE3
- 22.4. Fat-Tree-Based Connection Machine 5
- 22.5. Omega-Network-Based IBM SP2
- 22.6. Commodity-Based Berkeley NOW

22.1 Mechanisms for Message Passing

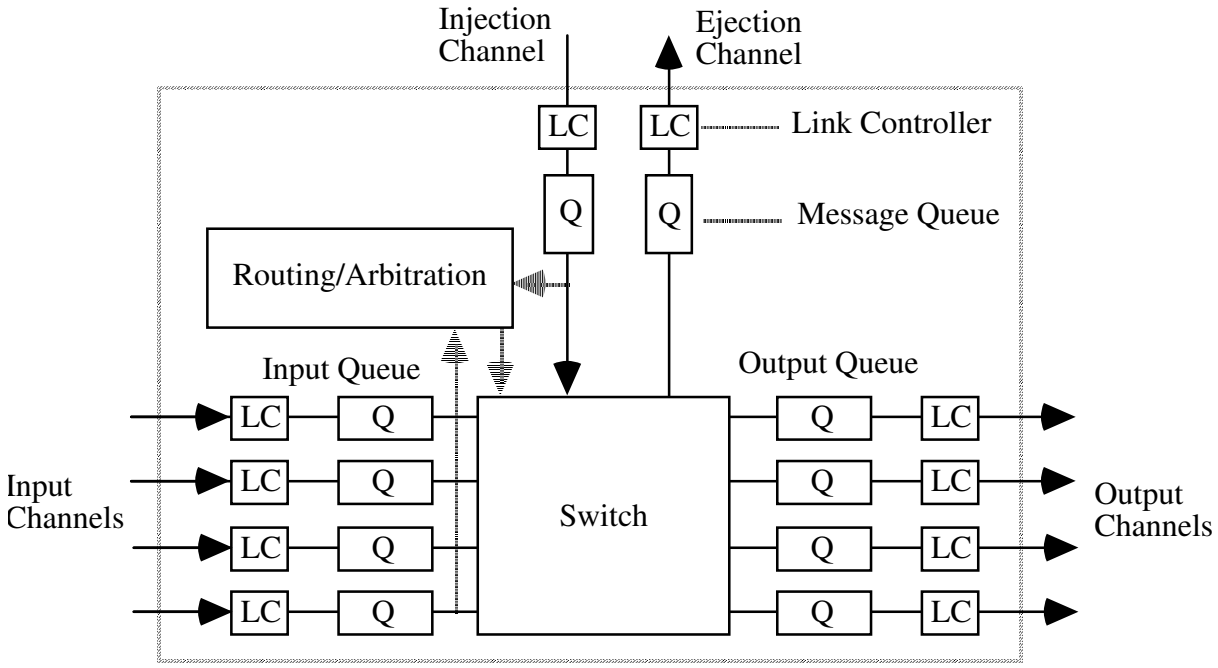


Fig. 22.1. The structure of a generic router.

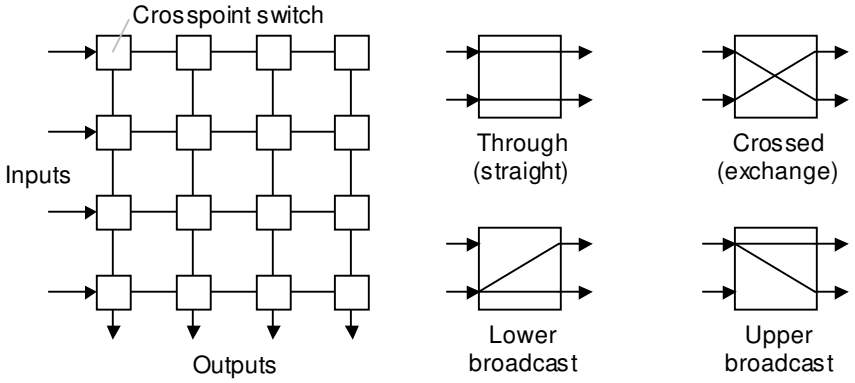


Fig. 22.2. Example 4 × 4 and 2 × 2 switches used as building blocks for larger networks.

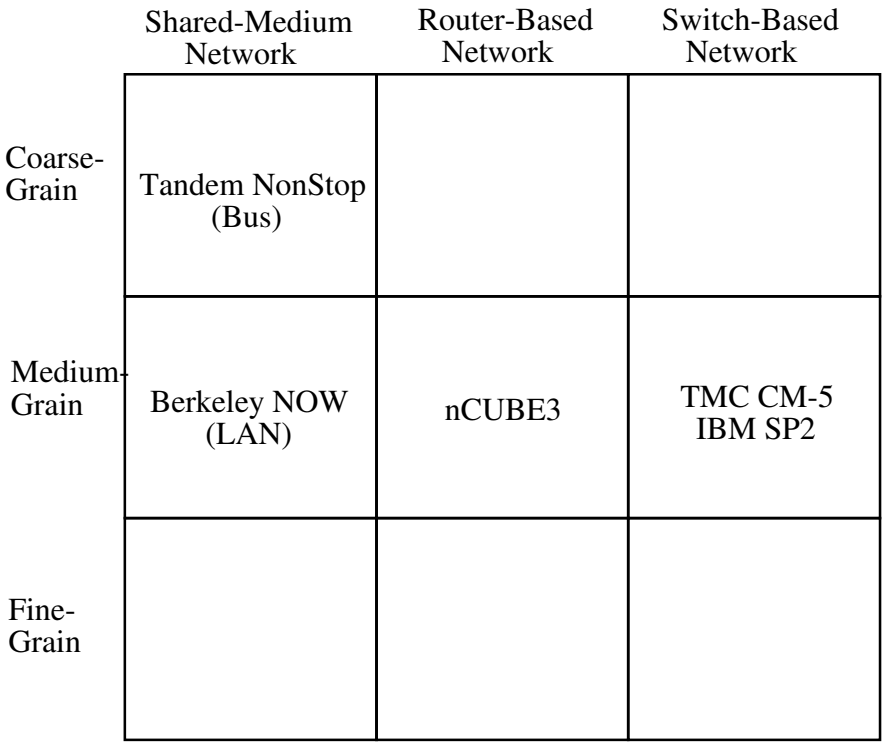


Fig. 22.3. Classification of message-passing hardware architectures and example systems that will be studied in this chapter.

22.2 Reliable Bus-Based Tandem Nonstop

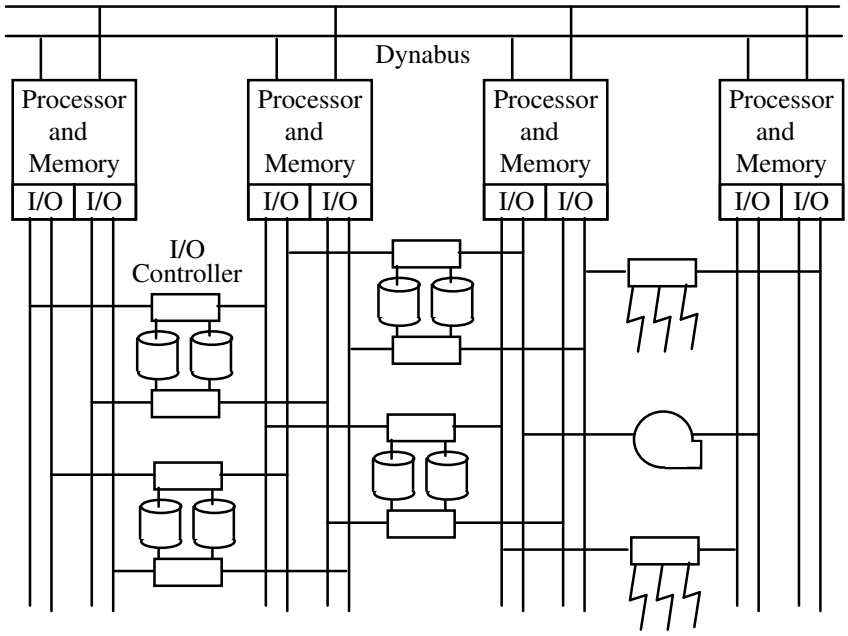


Fig. 22.4. One section of the Tandem NonStop Cyclone system.

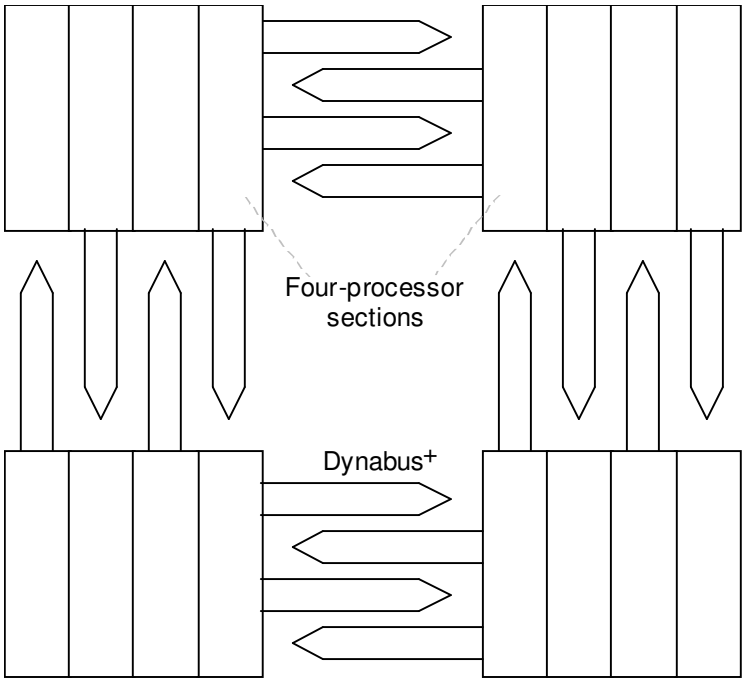


Fig. 22.5. Four four-processor sections interconnected by Dynabus+.

22.3 Hypercube-Based nCUBE3

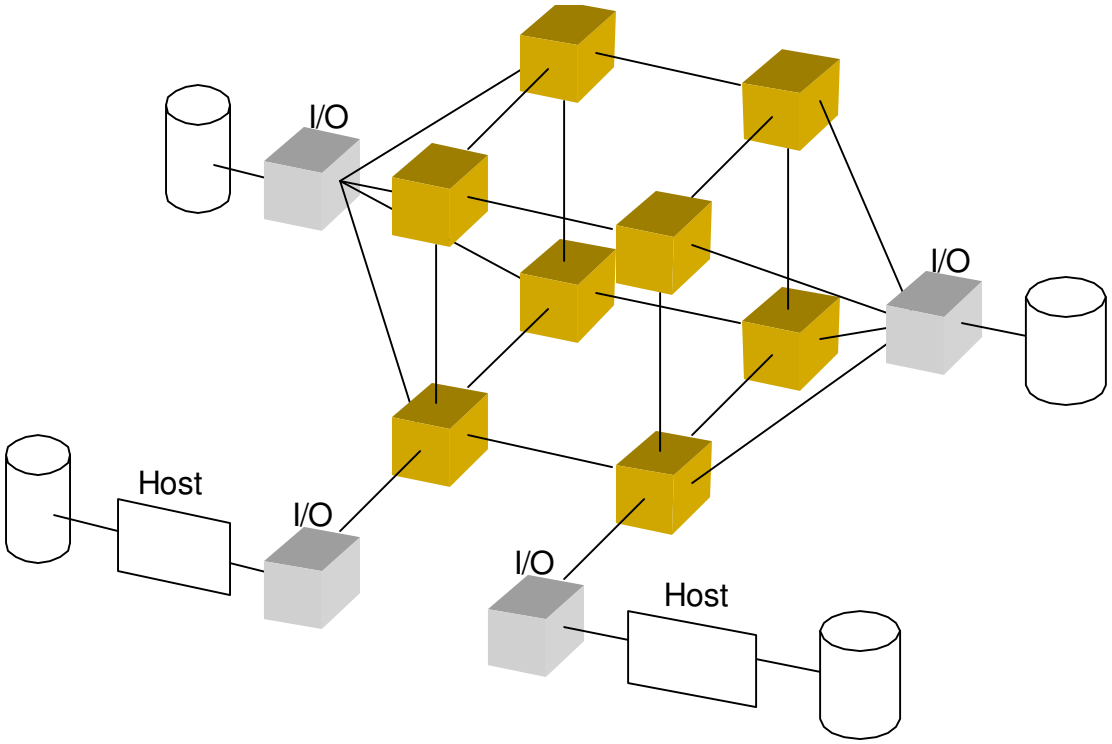


Fig. 22.6. An eight-node nCUBE architecture.

22.4 Fat-Tree-Based Connection Machine 5

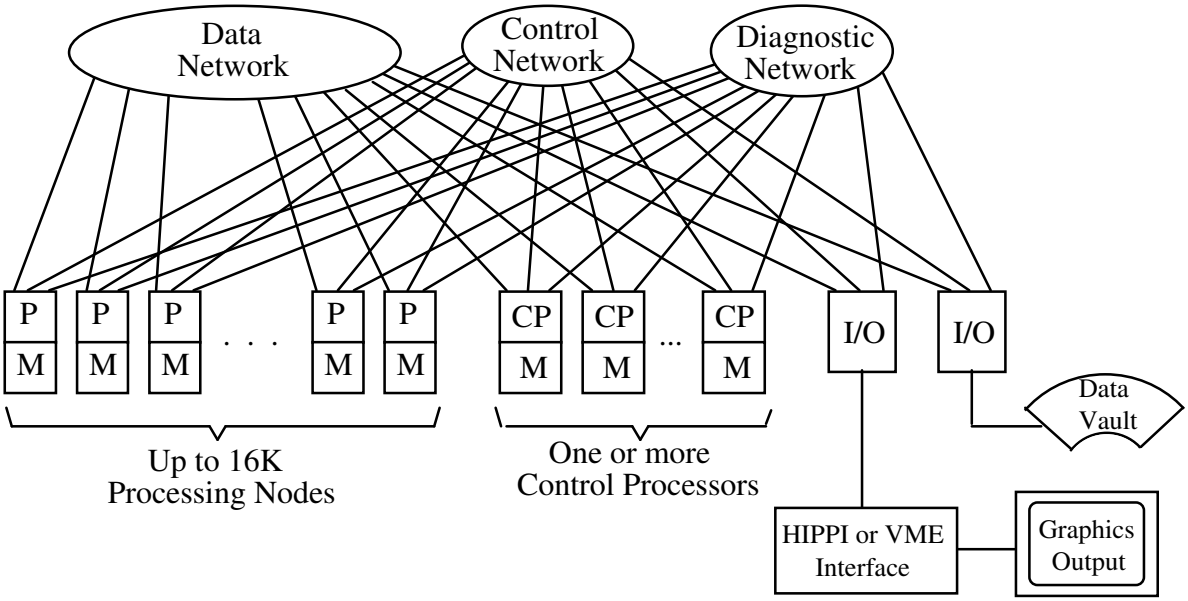


Fig. 22.7. The overall structure of CM-5.

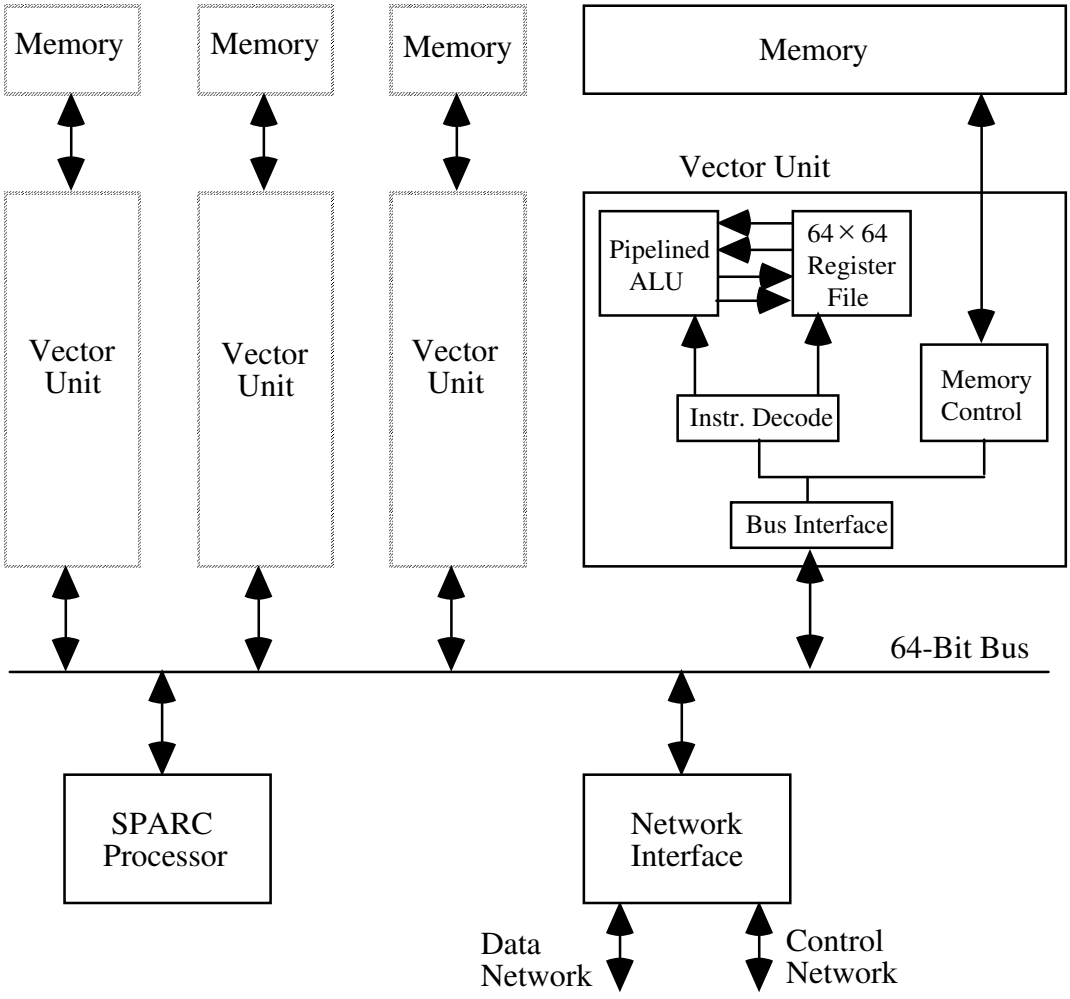


Fig. 22.8. The components of a processing node in CM-5.

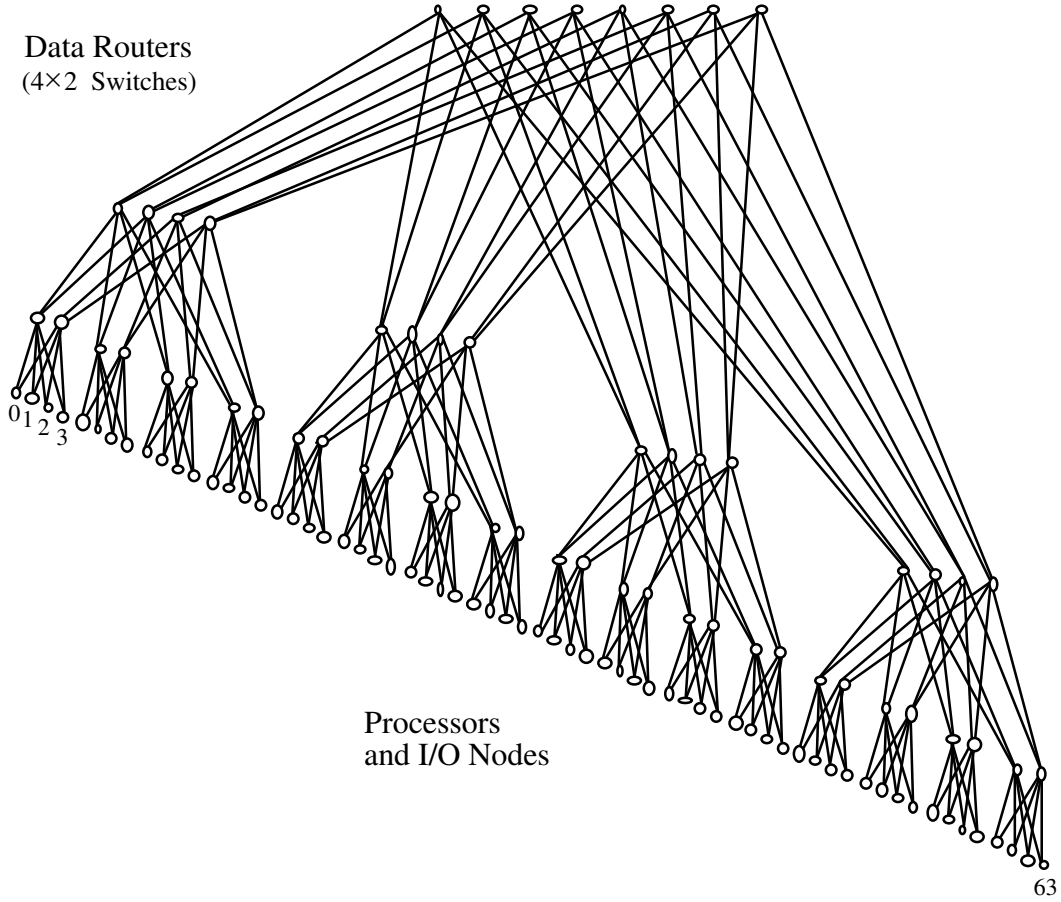


Fig. 22.9. The fat-tree (hyper-tree) data network of CM-5.

22.5 Omega-Network Based IBM SP2

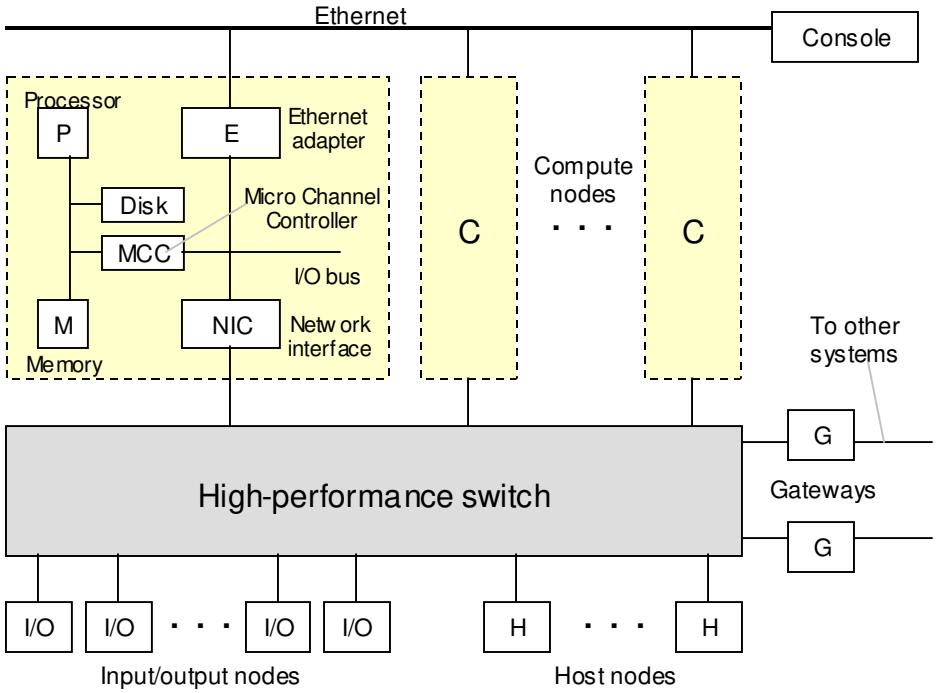


Fig. 22.10. The architecture of IBM SP series of systems.

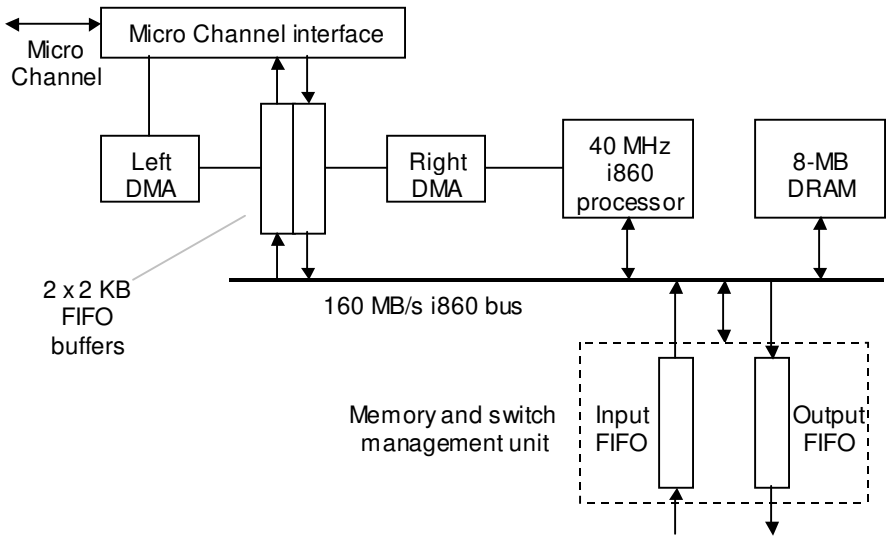


Fig. 22.11. The network interface controller of IBM SP2.

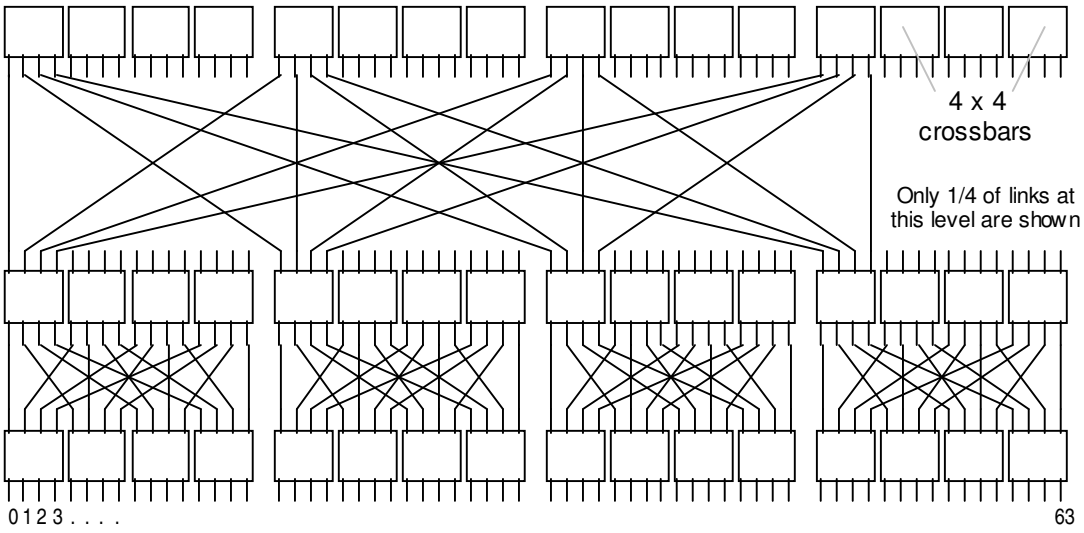


Fig. 22.12. A section of the high-performance switch network of IBM SP2.

22.6 Commodity-Driven Berkeley NOW

23 Data-Parallel SIMD Machines

Back to TOC

Chapter Goals

- Examining SIMD in more depth
- Discussing SIMD's successes and failures
- Looking at real SIMD machines, old and new

Chapter Contents

- 23.1. Where Have All the SIMDs Gone?
- 23.2. The First Supercomputer: ILLIAC IV
- 23.3. Massively Parallel Goodyear MPP
- 23.4. Distributed Array Processor (DAP)
- 23.5. Hypercubic Connection Machine 2
- 23.6. Multiconnected MasPar MP-2

23.1 Where Have All the SIMDs Gone?

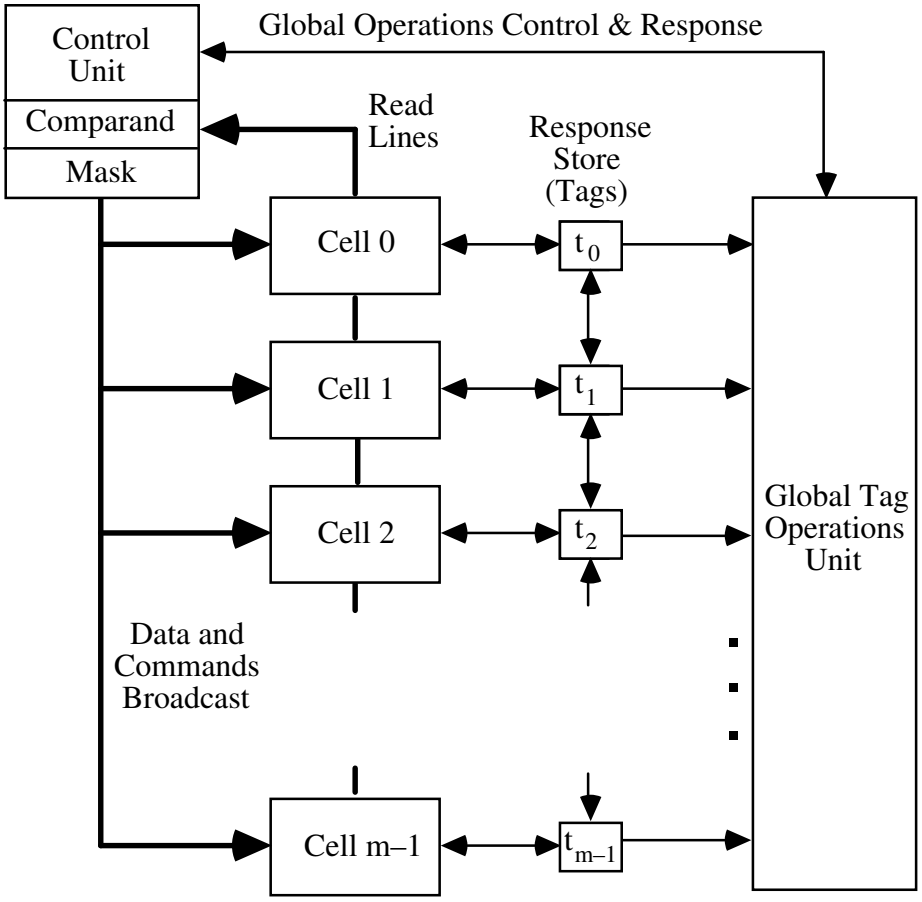


Fig. 23.1. Functional view of an associative memory/processor.

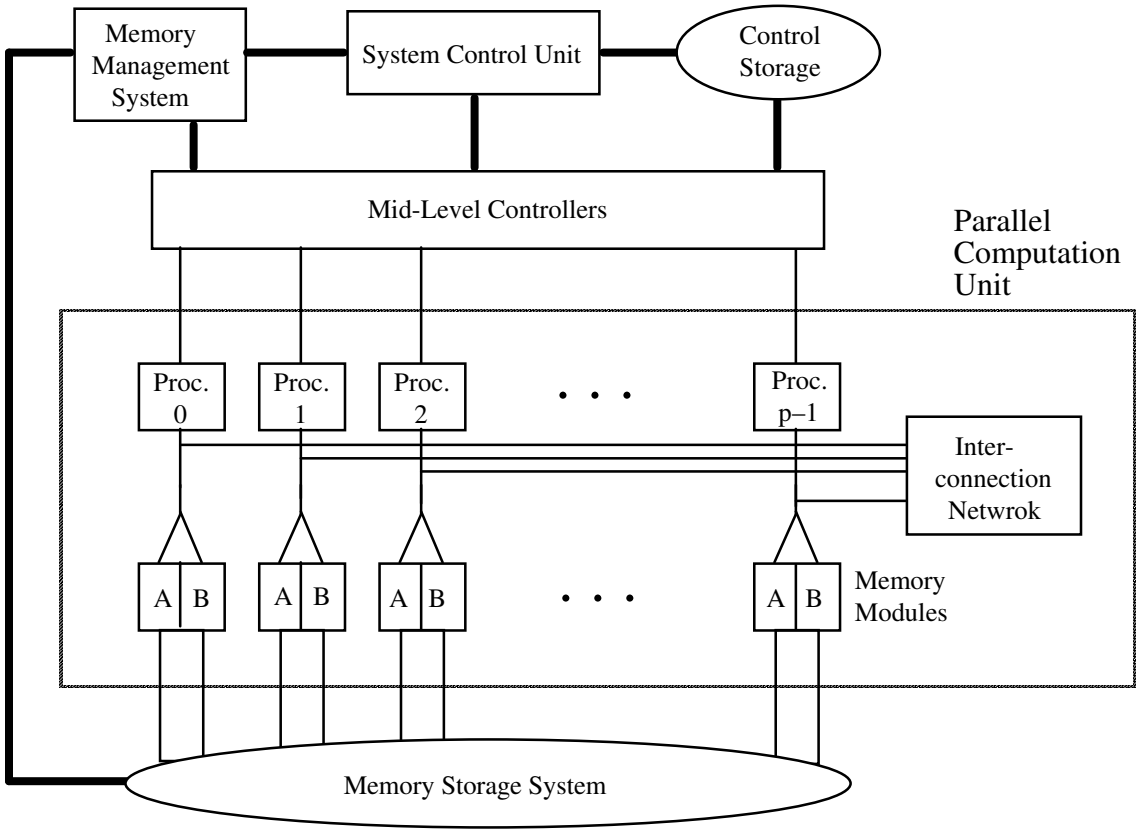


Fig. 23.2. The architecture of Purdue PASM.

23.2 The First Supercomputer: ILLIAC IV

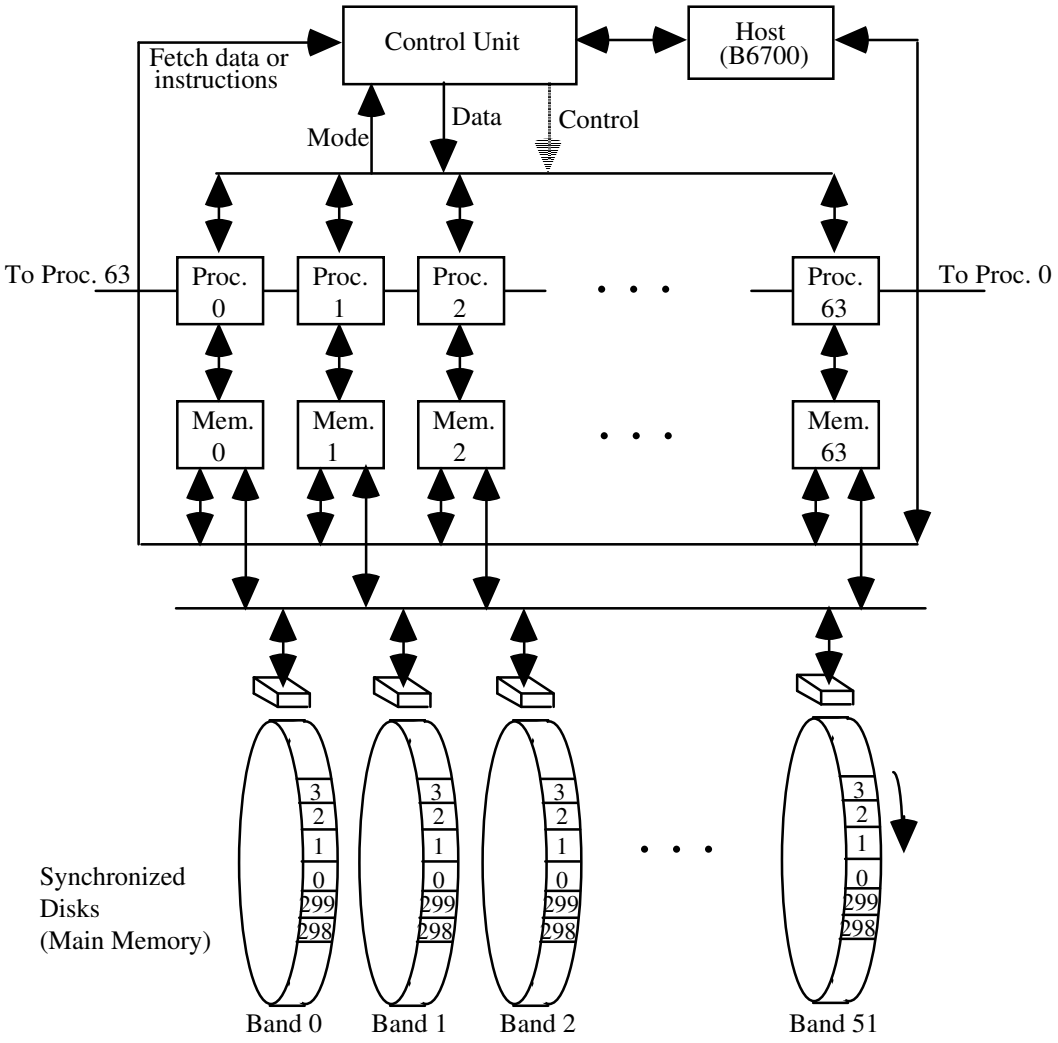


Fig. 23.3. The ILLIAC IV computer (the inter-processor routing network is only partially shown).

23.3 Massively Parallel Goodyear MPP

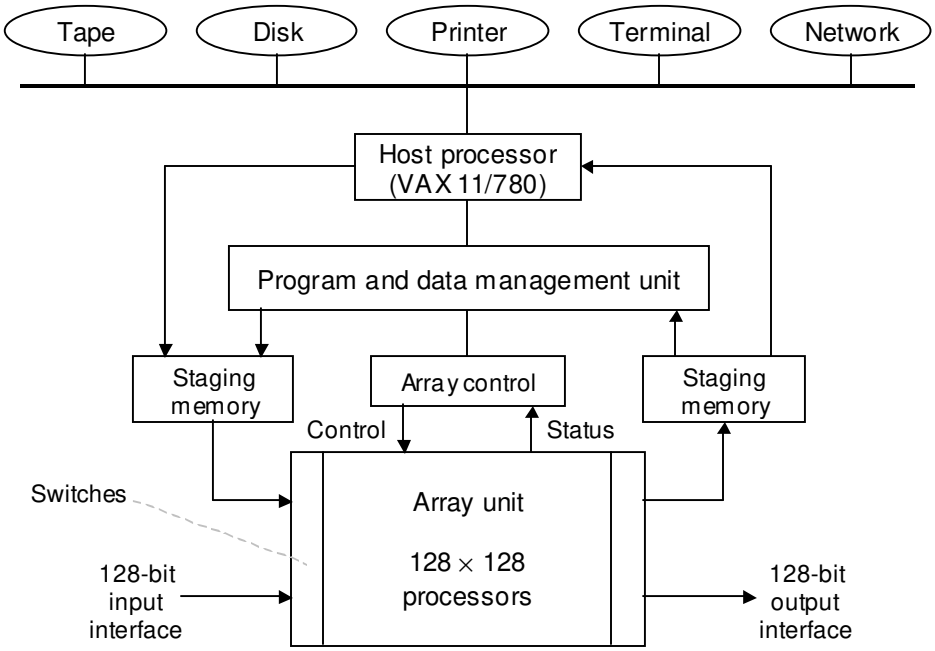


Fig. 23.4. The architecture of Goodyear MPP.

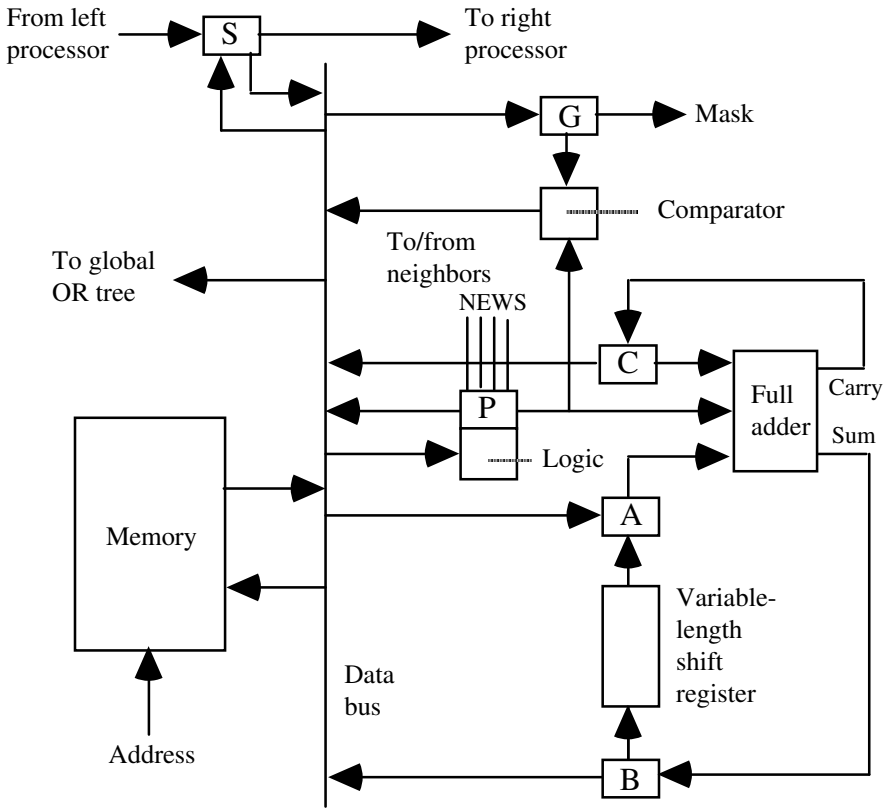


Fig. 23.5. The single-bit processor of MPP.

23.4 Distributed Array Processor (DAP)

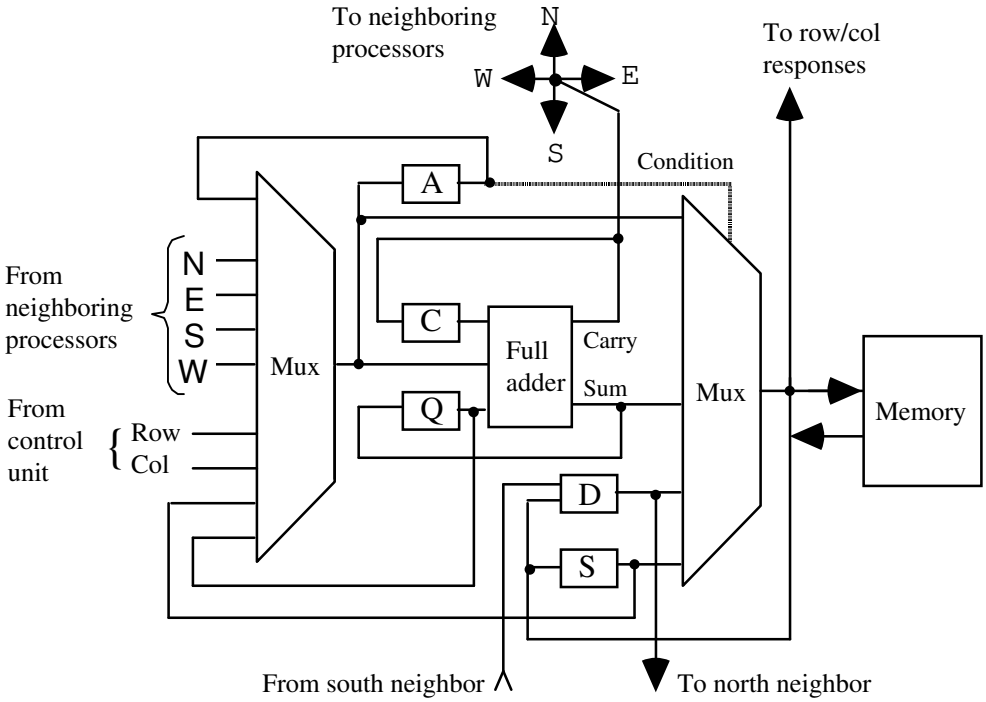


Fig. 23.6. The bit-serial processor of DAP.

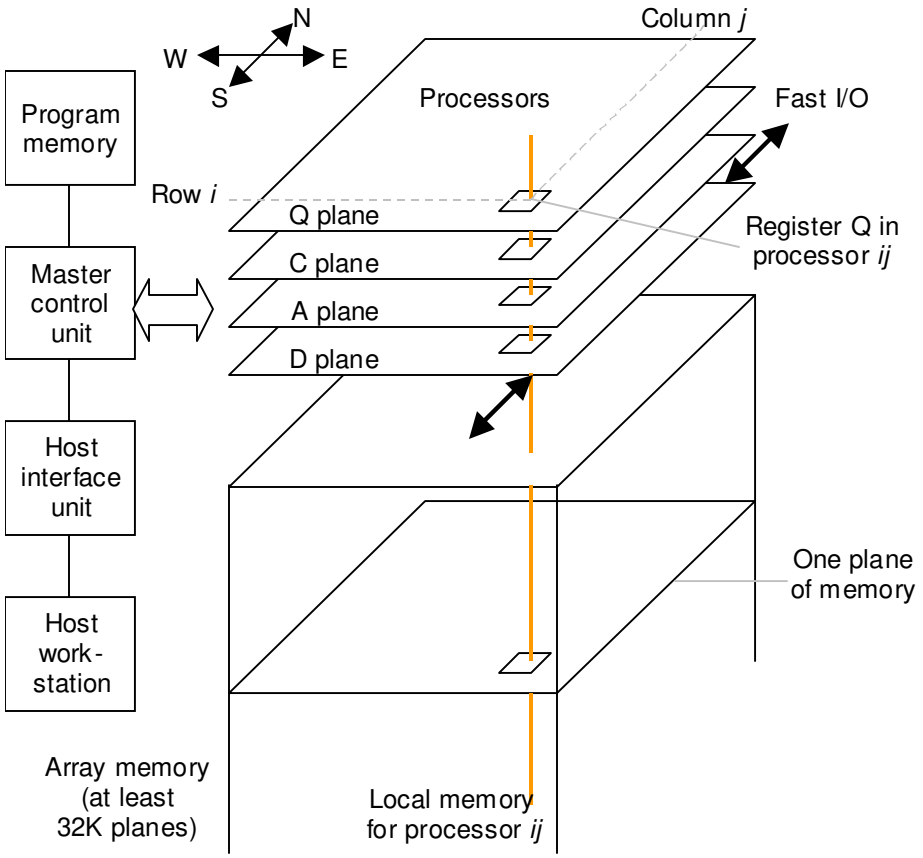


Fig. 23.7. The high-level architecture of DAP system.

23.5 Hypercubic Connection Machine 2

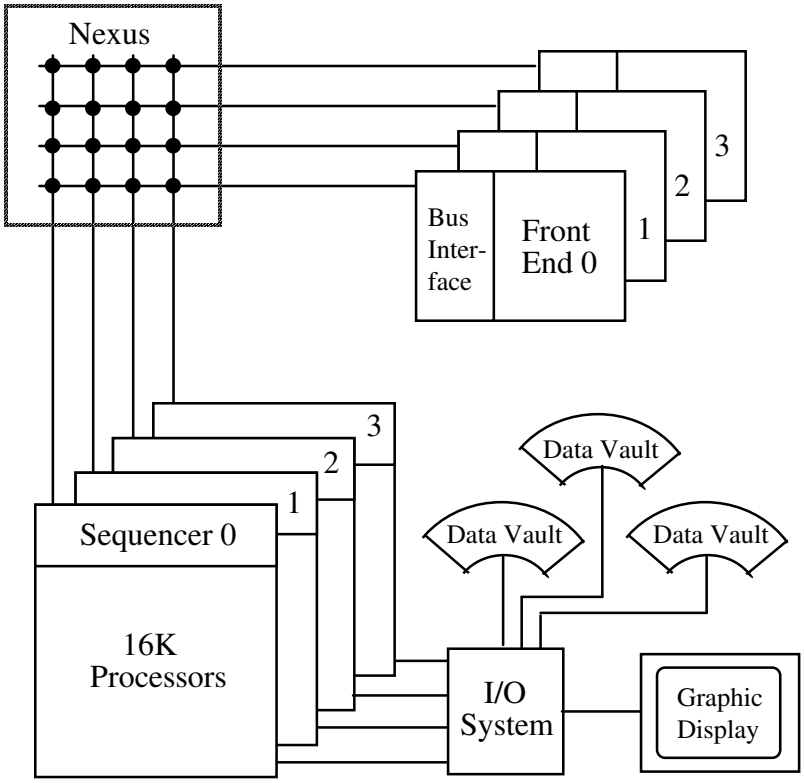


Fig. 23.8 The architecture of CM-2.

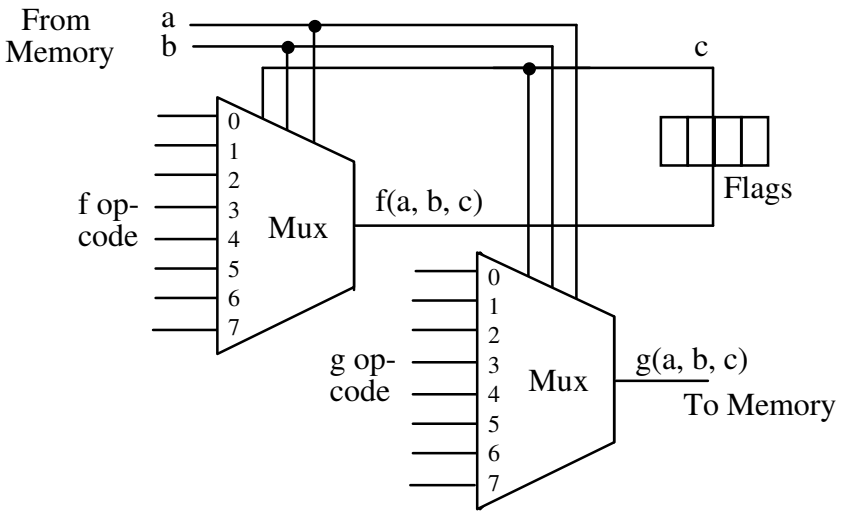


Fig. 23.9 The bit-serial ALU of CM-2.

23.6 Multiconnected MasPar MP-2

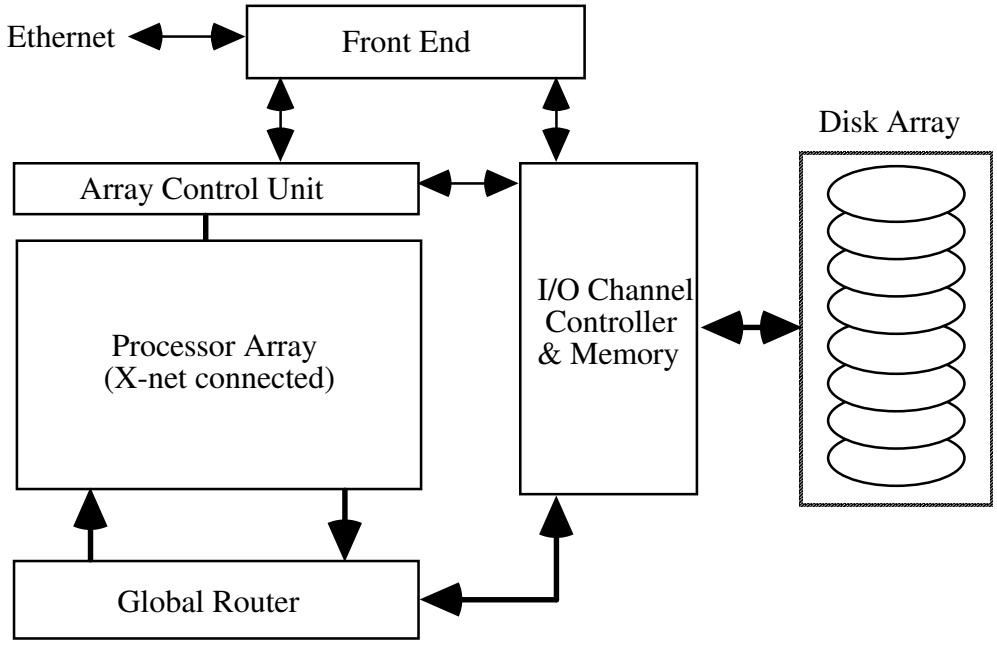


Fig. 23.10. The architecture of MasPar MP-2.

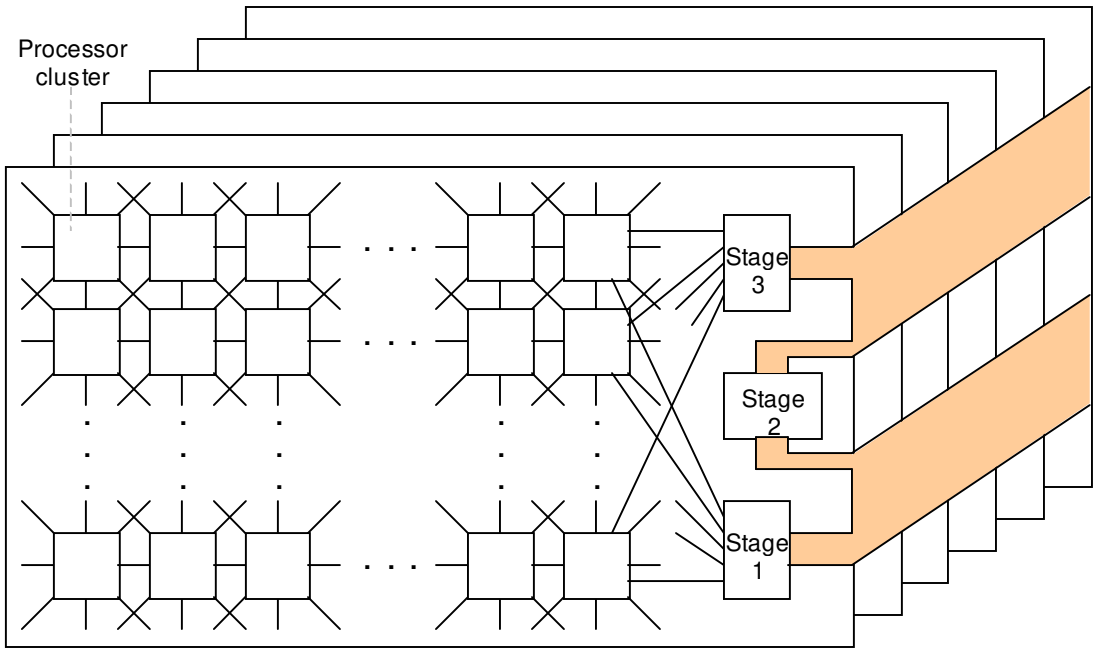


Fig. 23.11. The physical packaging of processor clusters and the 3-stage global router in MasPar MP-2.

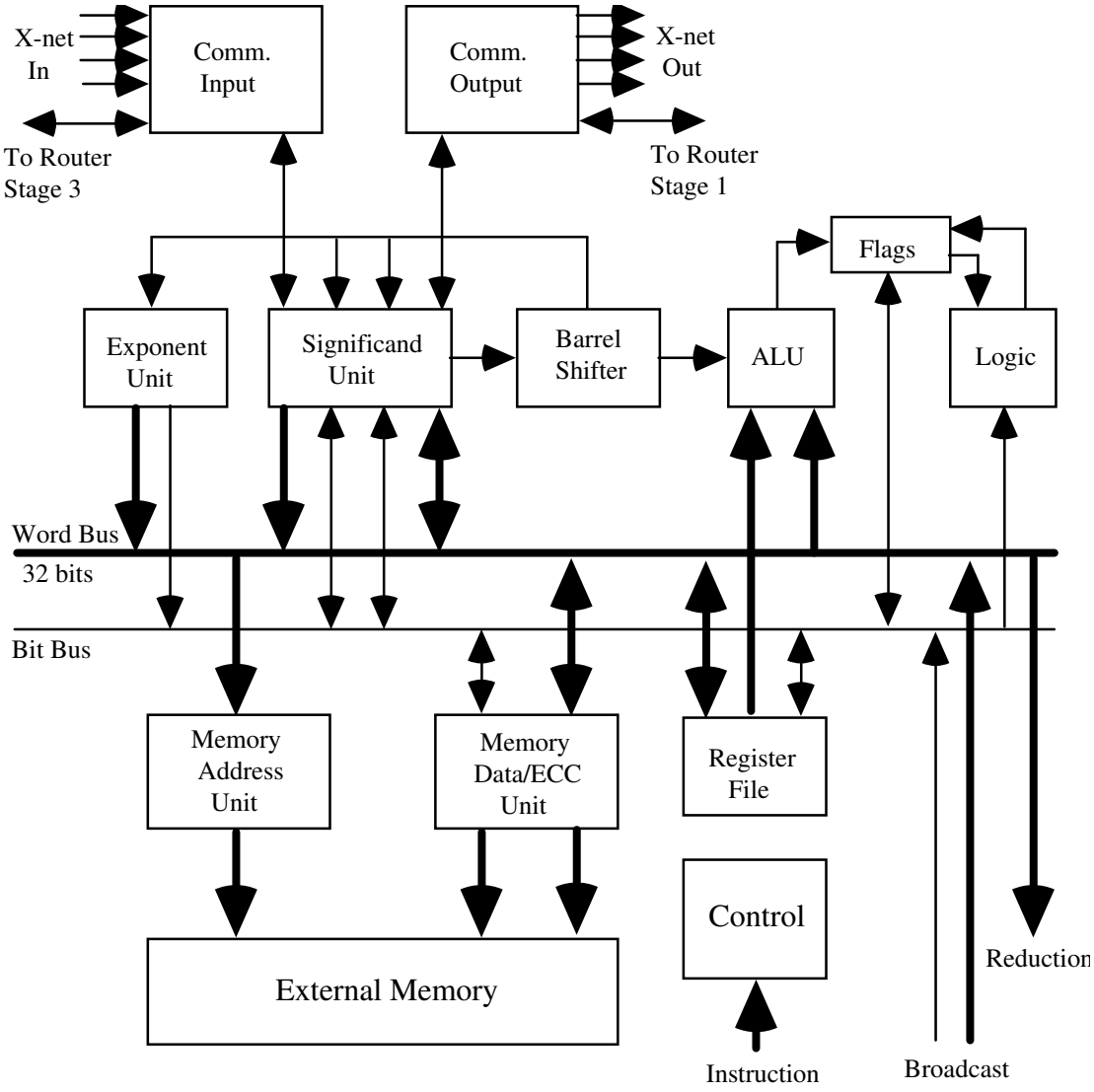


Fig. 23.12. Processor architecture in MasPar MP-2.

24 Past, Present, and Future

Back to TOC

Chapter Goals

- Review the history of parallel processing
- Discuss the current trends and debates
- Preview emerging technologies and architectures

Chapter Contents

- 24.1. Milestones in Parallel Processing
- 24.2. Current Status, Issues, and Debates
- 24.3. TFLOPS, PFLOPS, and Beyond
- 24.4. Processor and Memory Technologies
- 24.5. Interconnection Technologies
- 24.6. The Future of Parallel Processing

24.1 Milestones in Parallel Processing

24.2 Current Status, Issues, and Debates

24.3 TFLOPS, PFLOPS, and Beyond

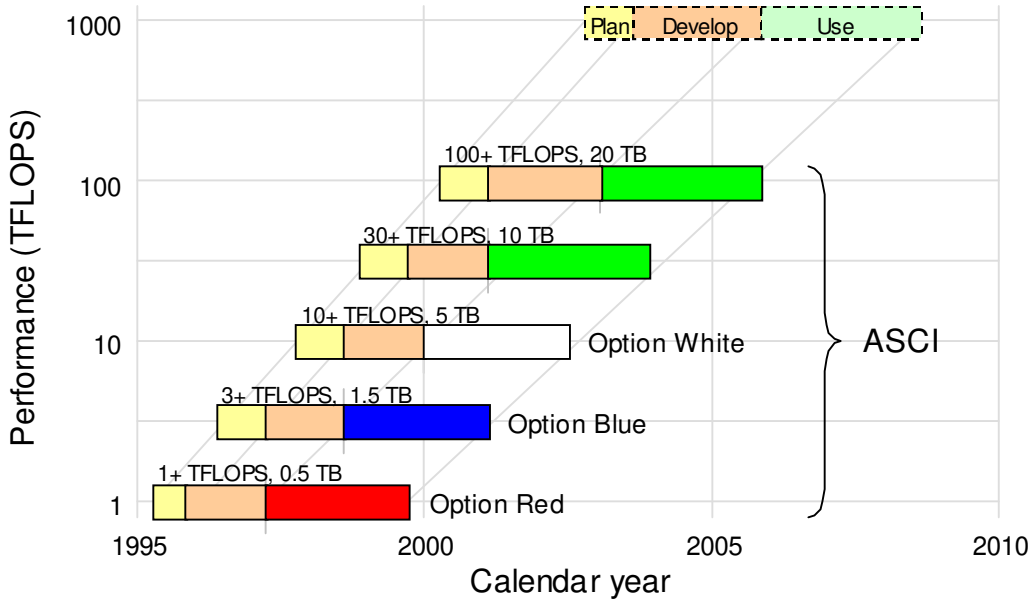


Fig. 24.1. Milestones in the Accelerated Strategic Computing Initiative (ASCI) program, sponsored by the US Department of Energy, with extrapolation up to the PFLOPS level.

24.4 Processor and Memory Technologies

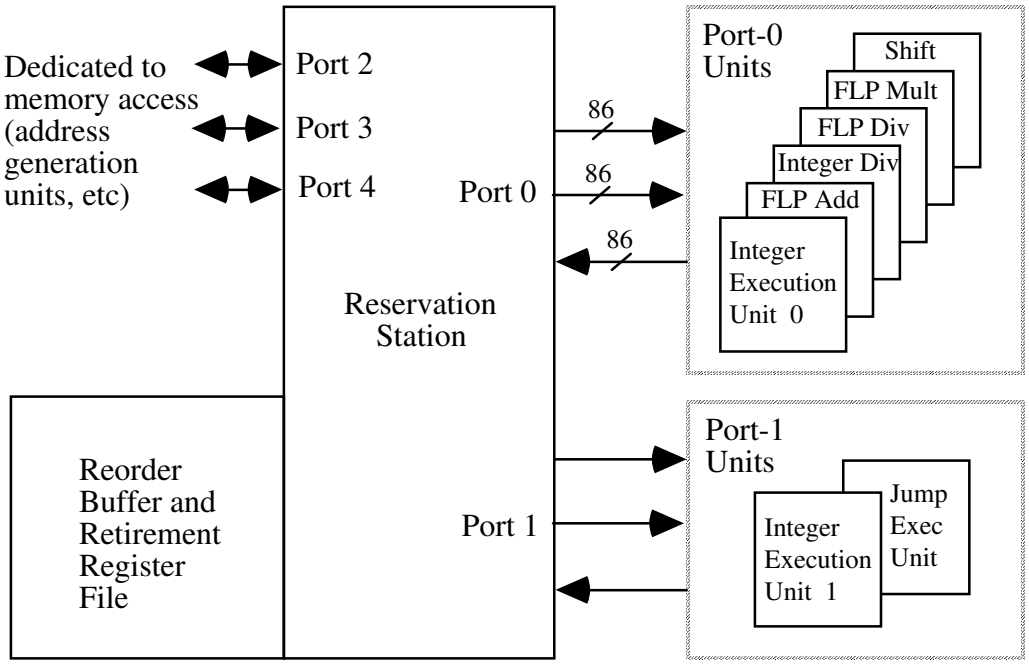


Fig. 24.2. Key parts of the CPU in the Intel Pentium Pro microprocessor.

24.5 Interconnection Technologies

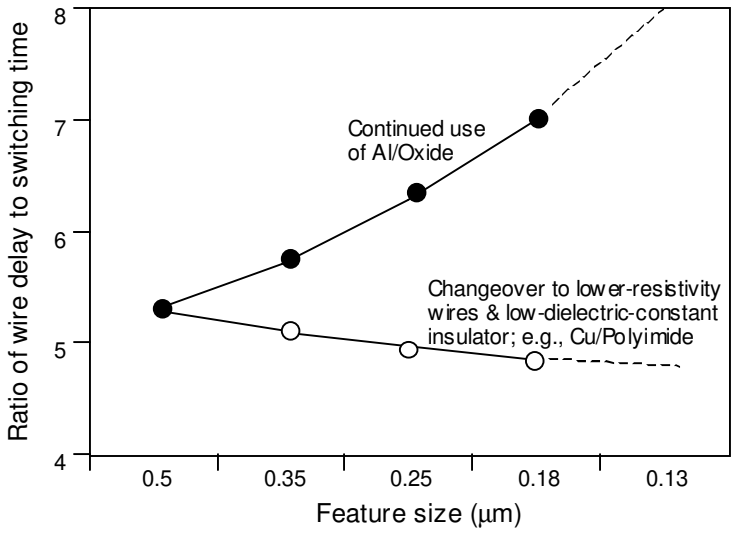


Fig. 24.3. Changes in the ratio of a 1-cm wire delay to device switching time as the feature size is reduced.

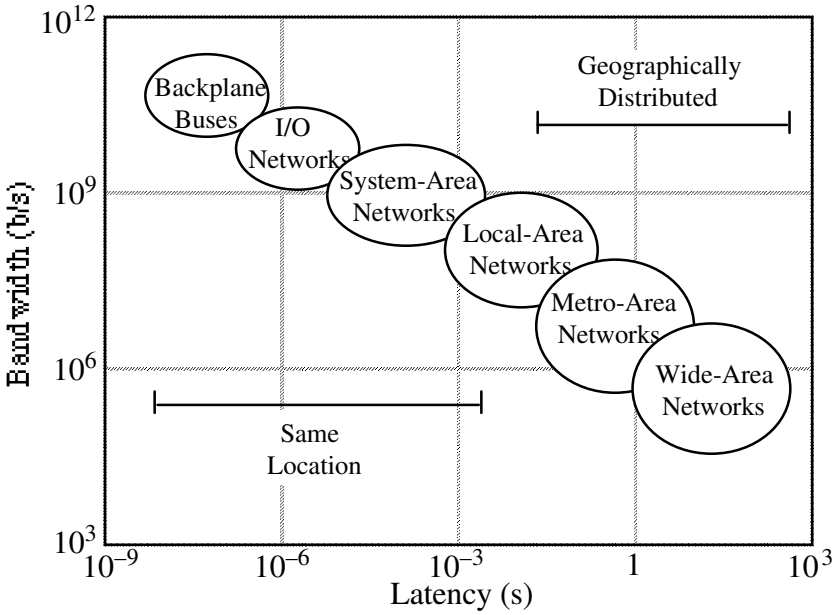


Fig. 24.4. Various types of intermodule and intersystem connections.

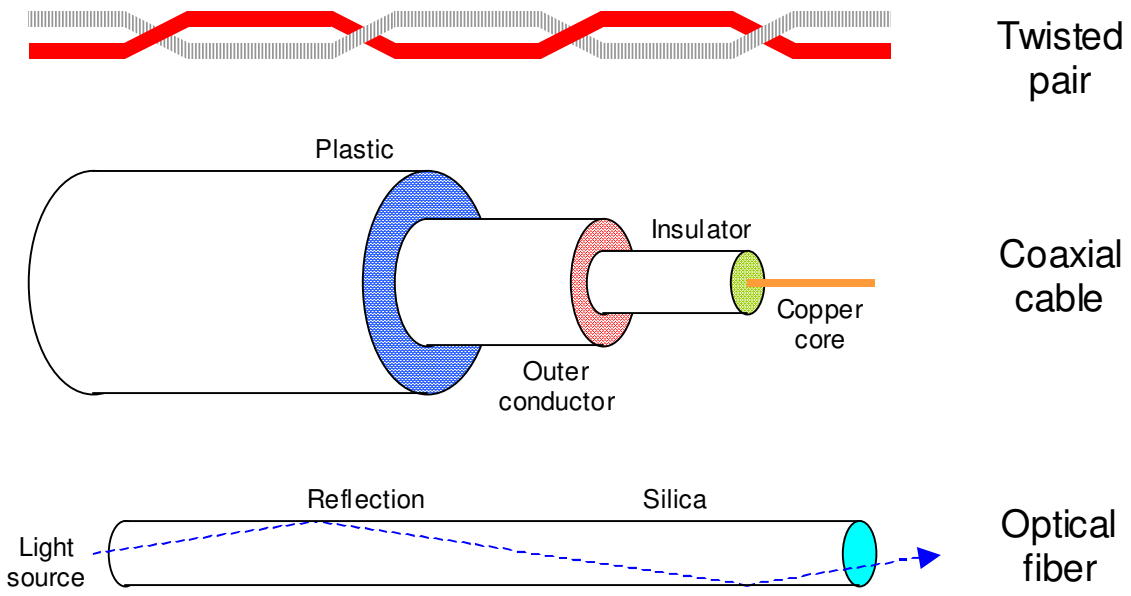


Fig. 24.5. The three commonly used media for computer and network connections.

24.6 The Future of Parallel Processing

ABCs of Parallel Processing

in one transparency* (parhami@ece.ucsb.edu)

* Originally appeared in *Computer Architecture News*, Vol. 27, No. 1, p. 2, March 1999.

f = unparallelizable fraction of a task (sequential overhead)
T_x = running time of a task when executed on x processors

A Amdahl's Law (Speed-up Formula)

Bad news: Sequential overhead will kill you, since:

$$\text{Speed-up} = \frac{T_1}{T_p} \leq \frac{1}{f + \frac{1-f}{p}} \leq \min\left(\frac{1}{f}, p\right)$$

Morale: For $f = 0.1$, e.g., the speed-up will be at best 10, no matter what the number of processors (peak OPS).

B Brent's Scheduling Theorem

Good news: Optimal scheduling is a very difficult problem, but even a naive scheduling algorithm can ensure:

$$\frac{T_1}{p} \leq T_p < \frac{T_1}{p} + T_\infty = \frac{T_1}{p}\left(1 + \frac{p}{T_1/T_\infty}\right)$$

Result: For a reasonably parallel task (with small T_∞), or for a suitably small number of processors (say, $p < T_1/T_\infty$), good speed-up and high utilization are attainable.

C Cost-Effectiveness Adage

Real news: The most cost-effective parallel solution to a given problem is often not the one with:

Highest peak OPS	(communication can kill you)
Greatest speed-up	(at what cost?)
Best utilization	(hardware busy doing what?)

Analogy: Mass transit (SIMD) might be more cost-effective than using private vehicles (MIMD) even if it is slower and leads to many empty seats on some trips.