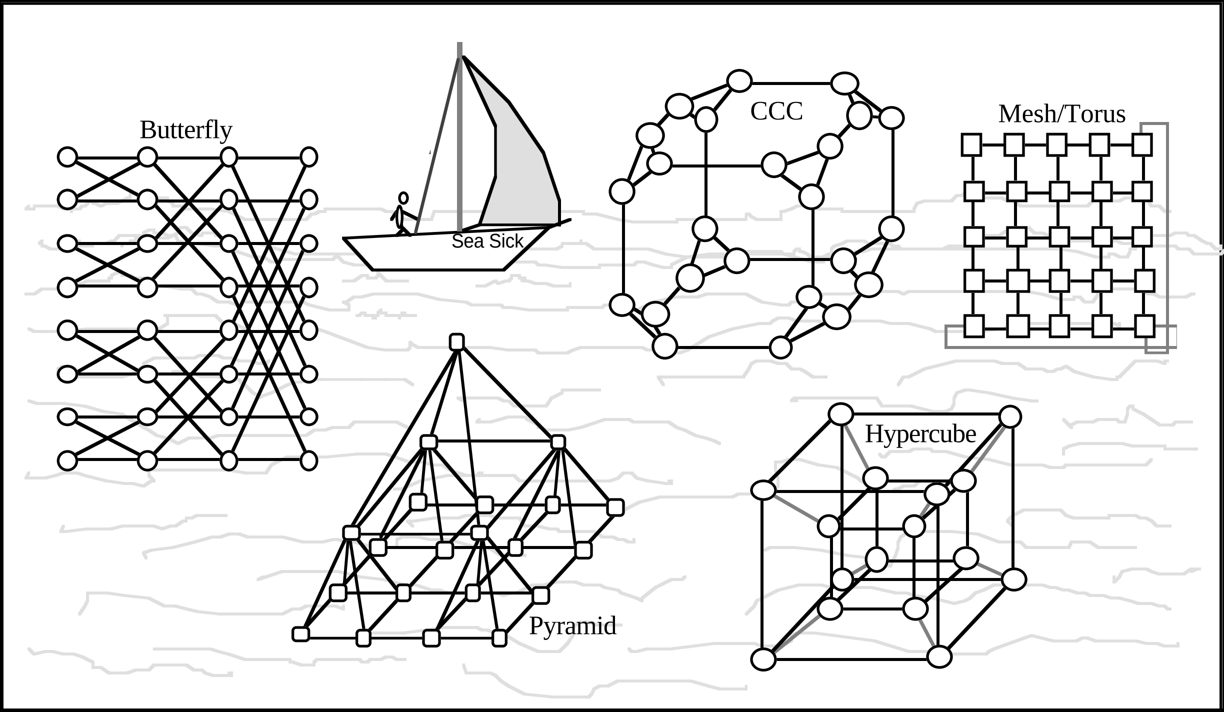
Instructor’s Manual

Vol. 2: Presentation Material

Plenum Series in Computer Science

Introduction to Parallel Processing

Algorithms and Architectures



Behrooz Parhami

This instructor’s manual is for

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

Plenum Series in Computer Science (ISBN 0-306-45970-1, QA76.58.P3798)

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. It also contains corrections and additions to the text, as well as other teaching aids. The spring 2000 edition of Volume 1 consists of the following parts (the next edition is planned for spring 2001):

Vol. 1: Problem Solutions

Part I	Selected Solutions and Additional Problems
Part II	Question Bank, Assignments, and Projects
Part III	Additions, Corrections, and Other Updates
Part IV	Sample Course Outline, Calendar, and Forms

Volume 2 contains enlarged versions of the figures and tables in the text, in a format suitable for use as transparency masters. It is accessible, as a large postscript file, via the author's Web address: http://www.ece.ucsb.edu/faculty/parhami

Vol. 2: Presentation Material

Parts I-VI	Lecture slides for Parts I-VI of the text

The author would appreciate the reporting of any error in the textbook or in this manual, suggestions for additional problems, alternate solutions to solved problems, solutions to other problems, and sharing of teaching experiences. Please e-mail your comments to

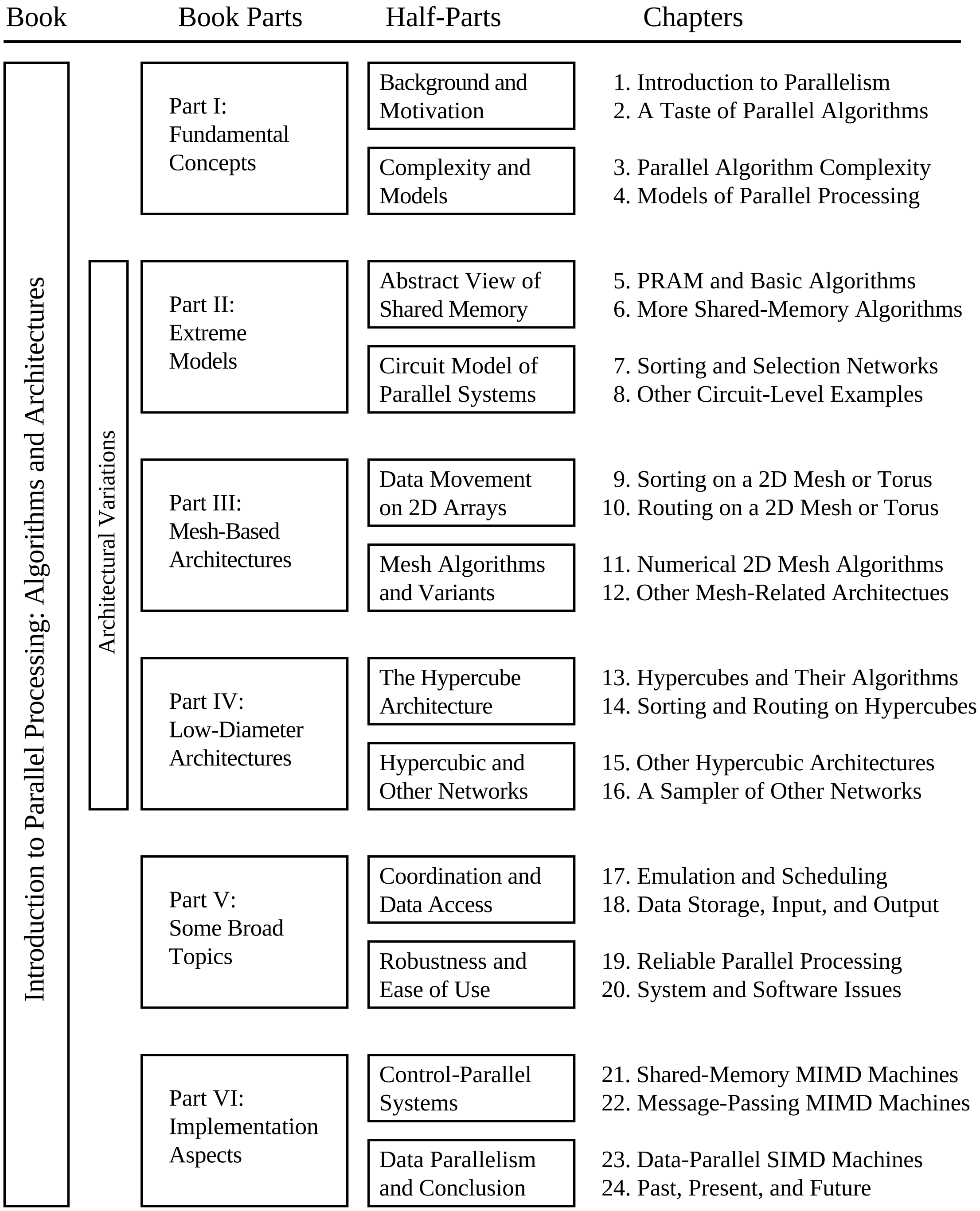
parhami@ece.ucsb.edu

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, California, USA
April 2000



Book		Book Parts	Half-Parts	Chapters
Introduction to Parallel Processing: Algorithms and Architectures		Part I: Fundamental Concepts	Background and Motivation	1. Introduction to Parallelism 2. A Taste of Parallel Algorithms
			Complexity and Models	3. Parallel Algorithm Complexity 4. Models of Parallel Processing
	Architectural Variations	Part II: Extreme Models	Abstract View of Shared Memory	5. PRAM and Basic Algorithms 6. More Shared-Memory Algorithms
			Circuit Model of Parallel Systems	7. Sorting and Selection Networks 8. Other Circuit-Level Examples
		Part III: Mesh-Based Architectures	Data Movement on 2D Arrays	9. Sorting on a 2D Mesh or Torus 10. Routing on a 2D Mesh or Torus
			Mesh Algorithms and Variants	11. Numerical 2D Mesh Algorithms 12. Other Mesh-Related Architectues
		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

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

Table of Contents, Vol. 2

Part I Fundamental Concepts

Part Goals

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

Part Contents

1 Introduction to Parallelism

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?

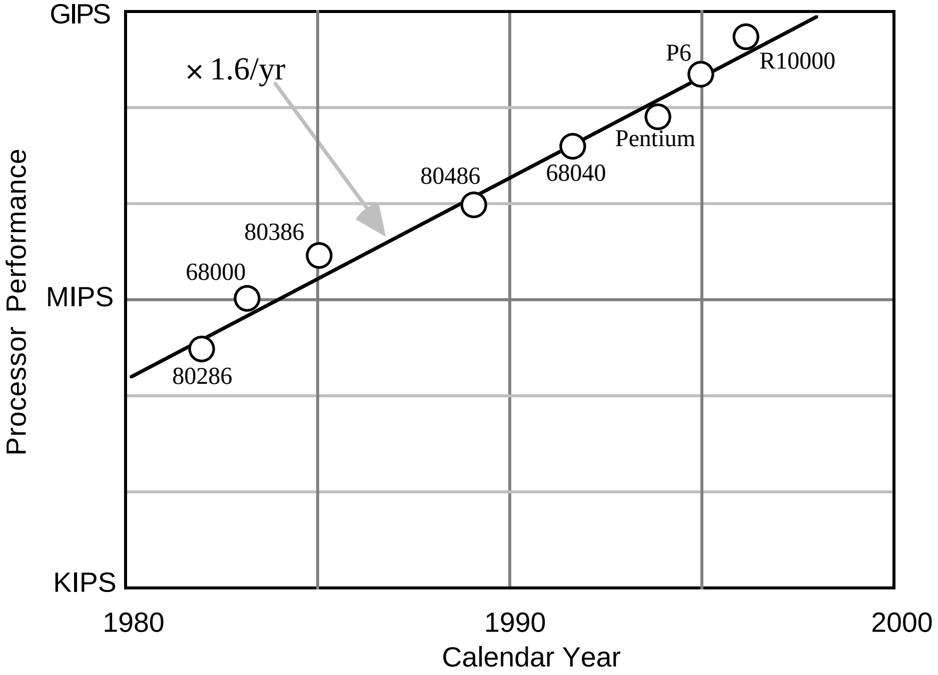


Fig. 1.1. The exponential growth of microprocessor performance, known as Moore's Law, shown over the past two decades.

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

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
If signals must travel 1 cm in an instruction cycle,
30 GIPS is the best we can hope for

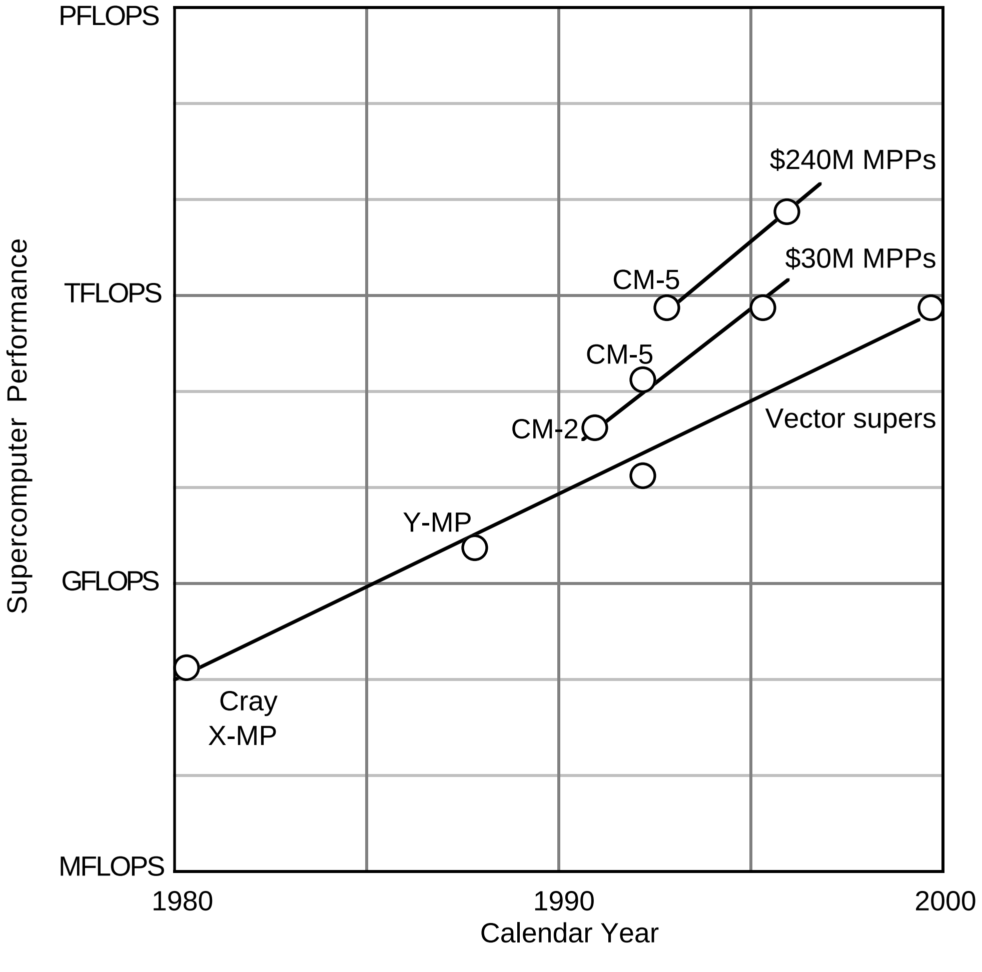


Fig. 1.2. The exponential growth in supercomputer per-formance over the past two decades [Bell92].

The need for TFLOPS

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

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

Fluid dynamics

1000 × 1000 × 1000 lattice ×
1000 FLOP per lattice point × 10 000 time steps =
10^{16} FLOP

Monte Carlo simulation of nuclear reactor

100 000 000 000 particles to track (for ≈1000 escapes) ×
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 / 10^4 s or 10^{15} FLOP / 10^3 s = 10^{12} FLOPS

Why the current quest for PFLOPS?

Same problems, perhaps with finer grids or longer simulated times

ASCI: Advanced Strategic Computing Initiative, US Department of Energy

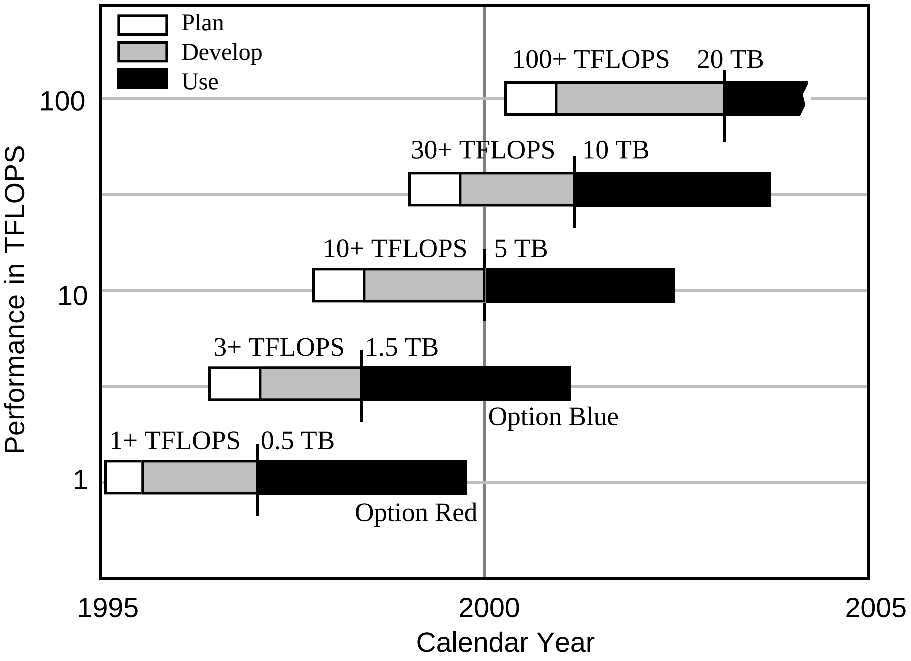


Fig. 24.1. Performance goals of the ASCI program.

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 × 4 × 4 cube

Tower: Each board linked to 4 neighbors in adjacent towers

System: 32 × 32 × 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.

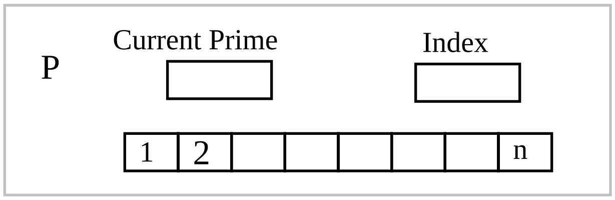


Fig. 1.4. Schematic representation of single-processor solution for the sieve of Eratosthenes.

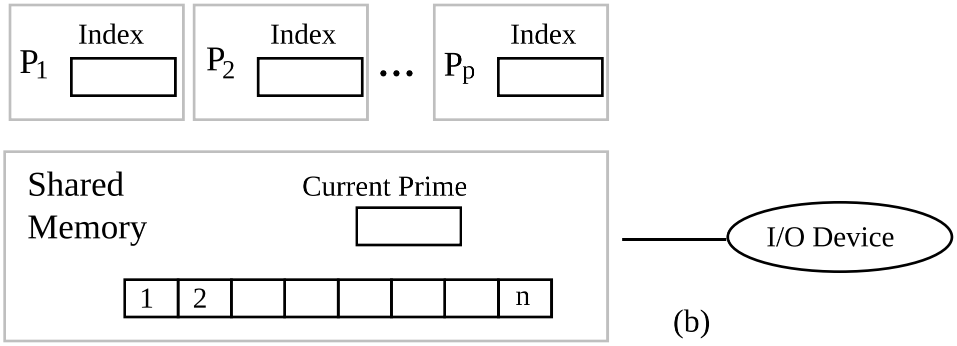


Fig. 1.5. Schematic representation of a control-parallel solution for the sieve of Eratosthenes.

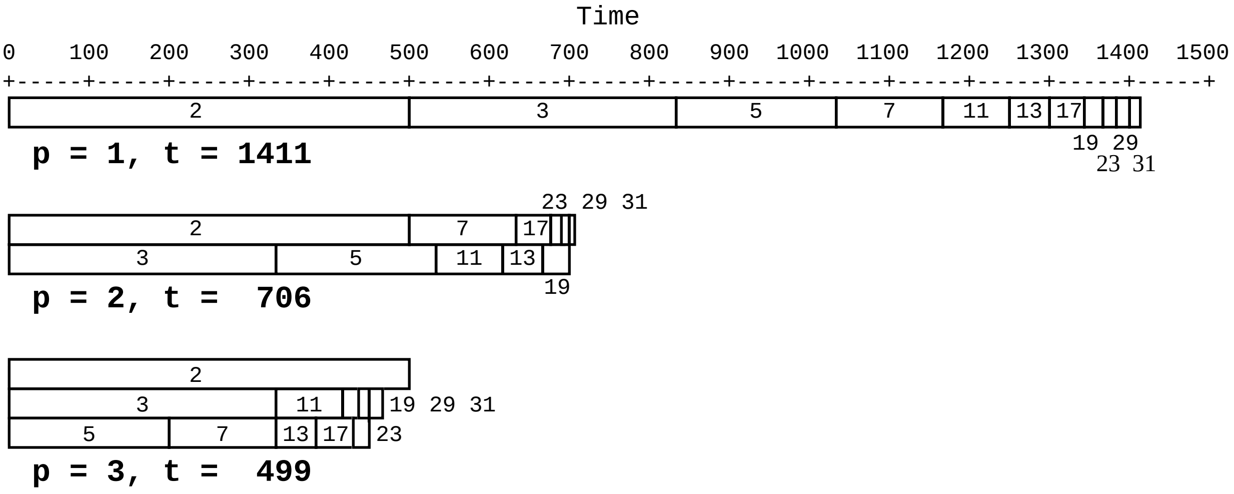


Fig. 1.6. Control-parallel realization of the sieve of Eratosthenes with n = 1000 and $1 \le p \le 3$.

P_1 finds each prime and broadcasts it to all other processors (assume $n/p \le \sqrt{n}$)

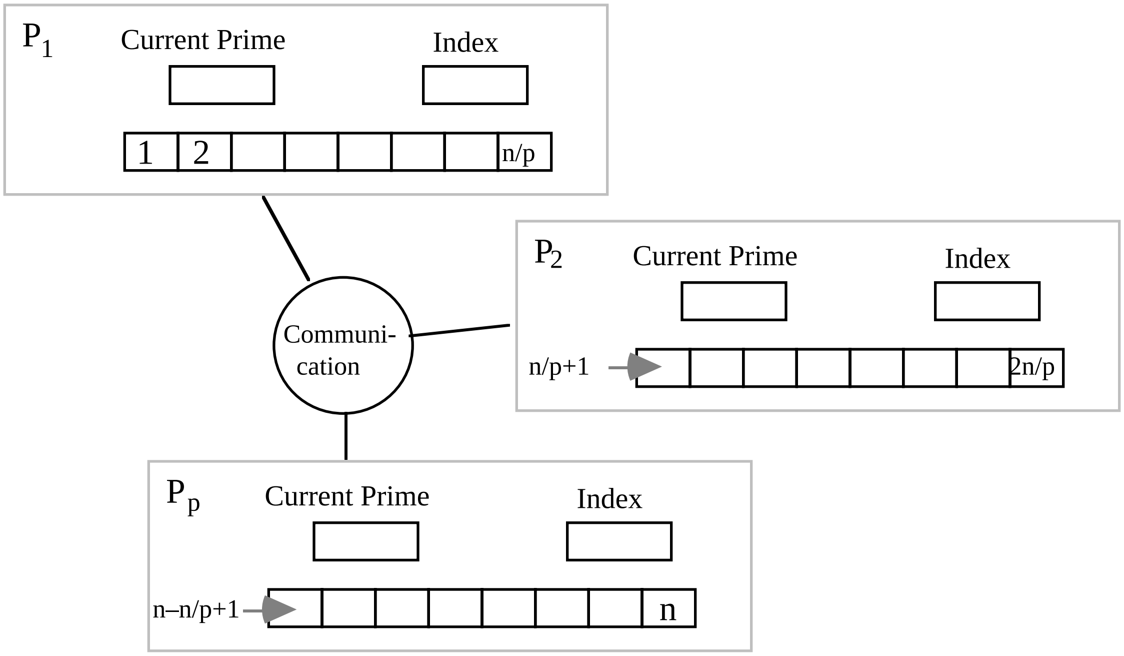


Fig. 1.7. Data-parallel realization of the sieve of Eratosthenes.

Some reasons for sublinear speed-up

Communication overhead

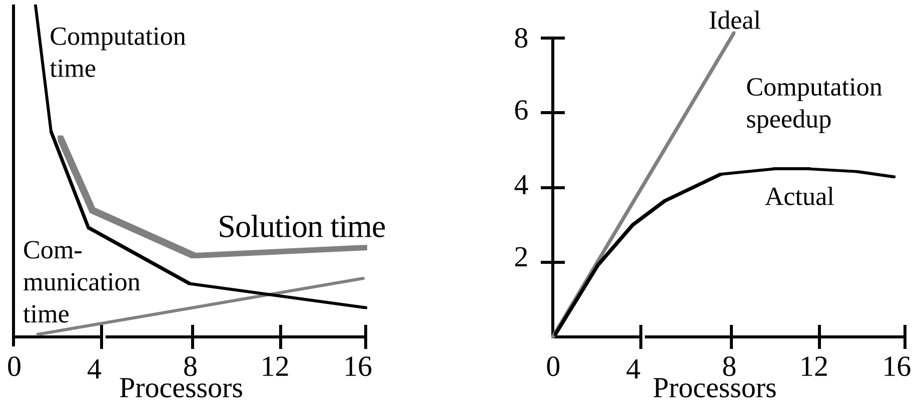


Fig. 1.8. Trade-off between communication time and computation time in the data-parallel realization of the sieve of Eratosthenes.

Input/output overhead

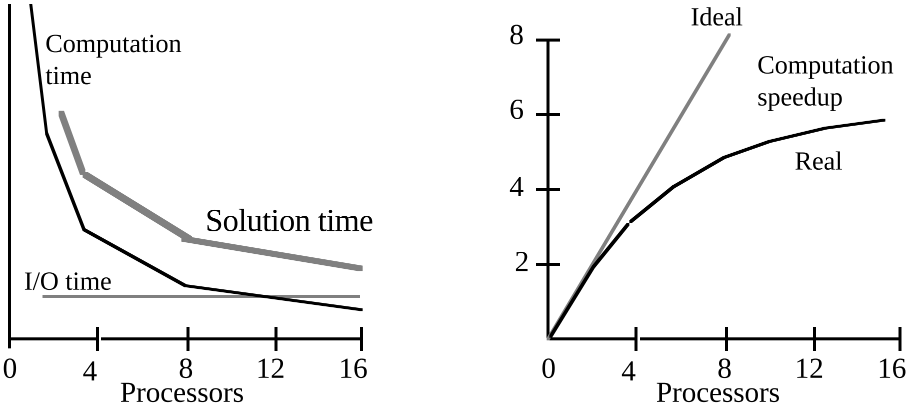


Fig. 1.9. Effect of a constant I/O time on the data-parallel realization of the sieve of Eratosthenes.

1.3 Parallel Processing Ups and Downs

Early 1900s: 1000s of "computers" (humans + calculators)

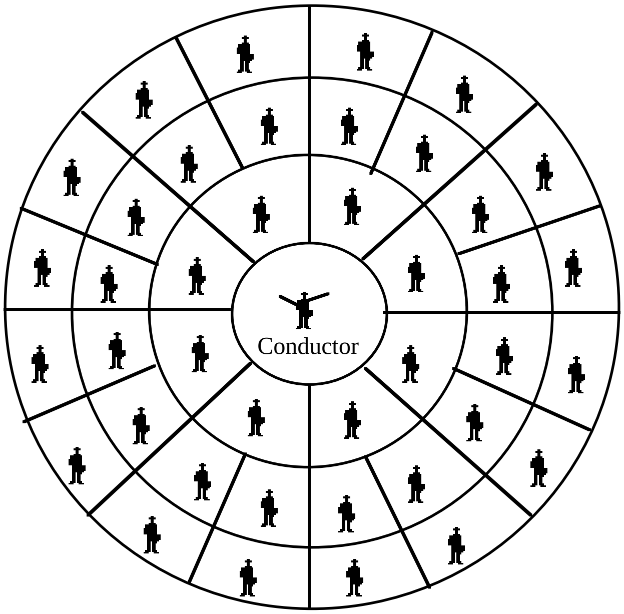


Fig. 1.10. Richardson's circular theater for weather forecasting calculations.

Parallel processing is used in virtually all computers

Compute-I/O overlap, pipelining, multiple function units

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

1960s: ILLIAC IV (U Illinois) – 4 quadrants, each 8 × 8 mesh

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

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

1.4 Types of Parallelism: A Taxonomy

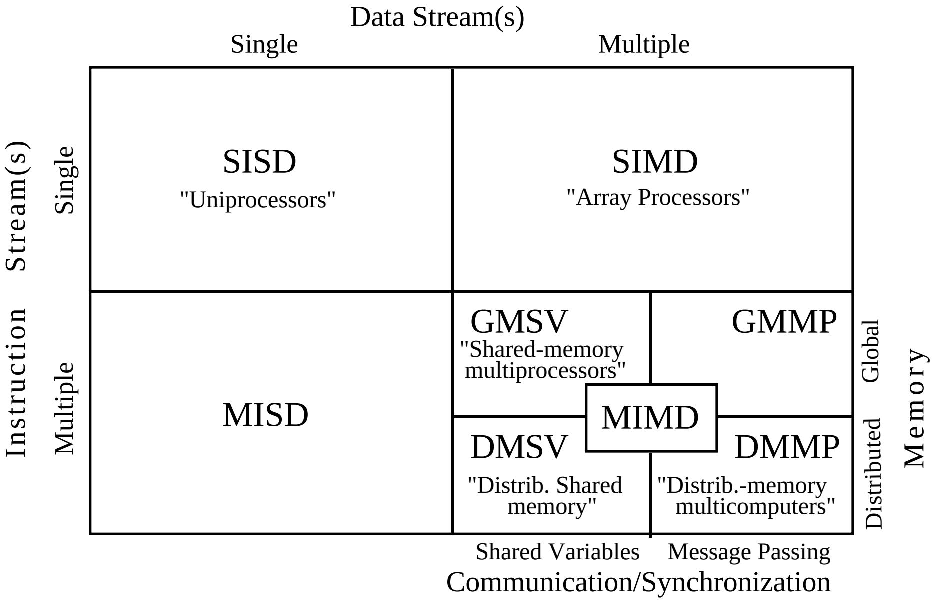


Fig. 1.11. The Flynn-Johnson classification of computer systems.

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

RISC, CISC, PRAM, NUMA, VLIW

JPDC, TPDS

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 is proportional to the square of cost)

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

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)

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

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

f. Amdahl's law

a small fraction f of inherently sequential

or unparallelizable computation

severely limits the speed-up)

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

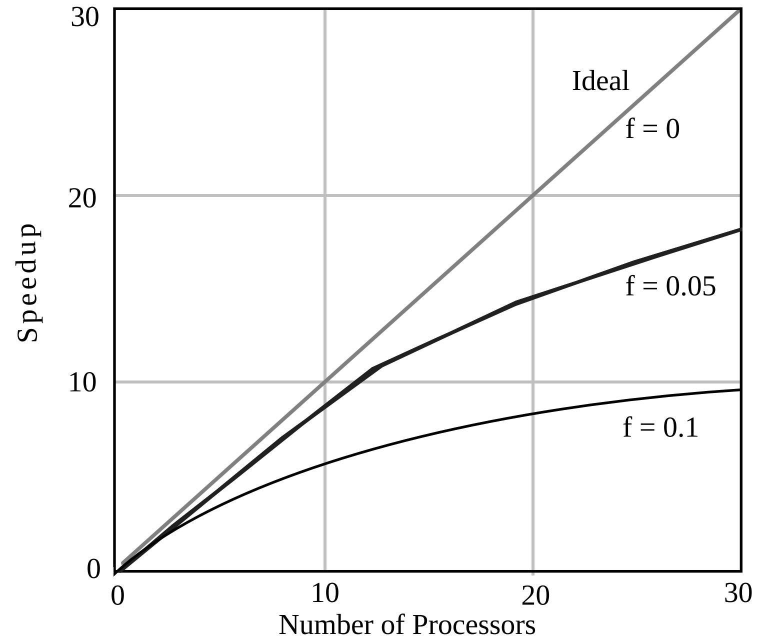


Fig. 1.12. The limit on speed-up according to Amdahl's law.

1.6 Effectiveness of Parallel Processing

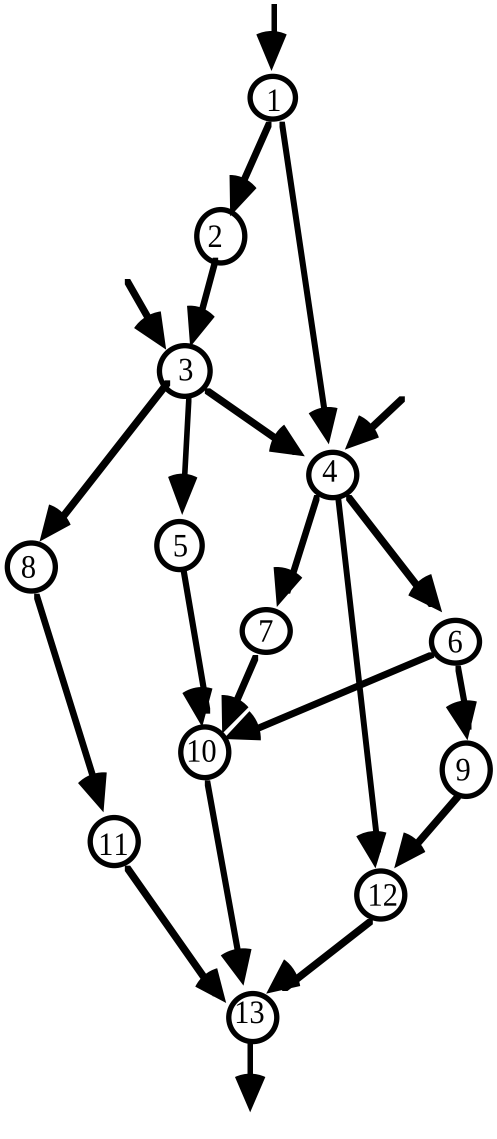


Fig. 1.13. Task graph exhibiting limited inherent parallelism.

Measures used in this course to compare parallel architectures and algorithms [Lee80]:

p	Number of processors
W(p)	Total number of unit operations performed by the p processors; computational work or energy
T(p)	Execution time with p processors; $T(1) = W(1)$ and $T(p) \le W(p)$

$$S(p) \quad \text{Speedup} = \frac{T(1)}{T(p)}$$

$$E(p) \quad \text{Efficiency} = \frac{T(1)}{pT(p)}$$

$$R(p) \quad \text{Redundancy} = \frac{W(p)}{W(1)}$$

$$U(p) \quad \text{Utilization} = \frac{W(p)}{pT(p)}$$

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

Relationships among the preceding measures:

$$1 \le S(p) \le p \qquad U(p) = R(p)E(p)$$

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

$$\frac{1}{p} \le E(p) \le U(p) \le 1 \qquad 1 \le R(p) \le \frac{1}{E(p)} \le p$$

$$Q(p) \le S(p) \le p$$

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

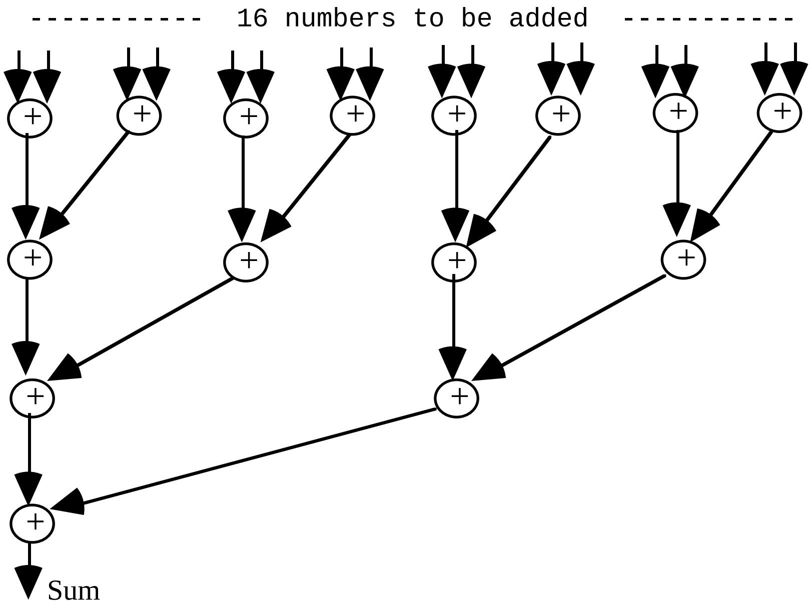


Fig. 1.14. Computation graph for finding the sum of 16 numbers.

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

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 interplay between algorithm and architecture

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

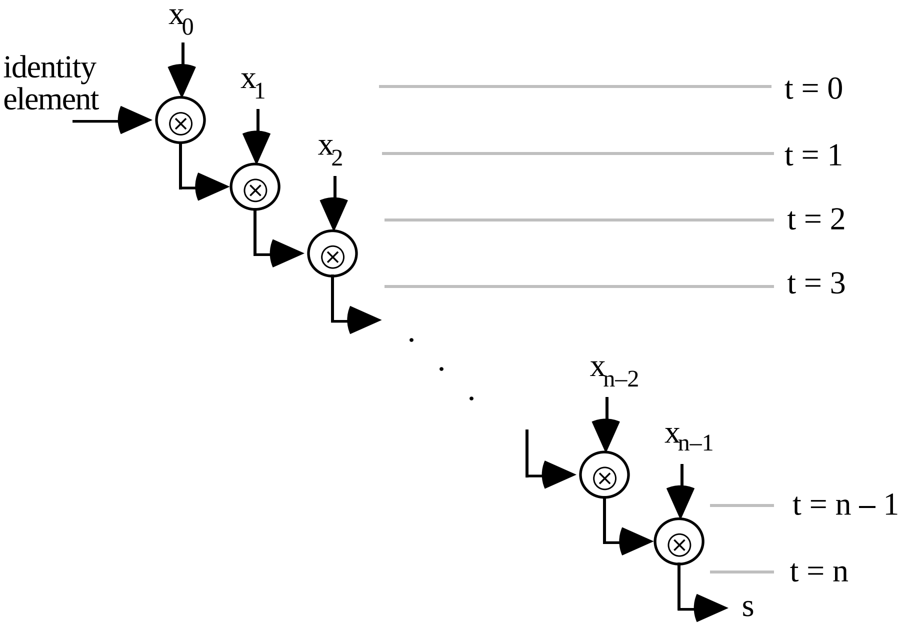
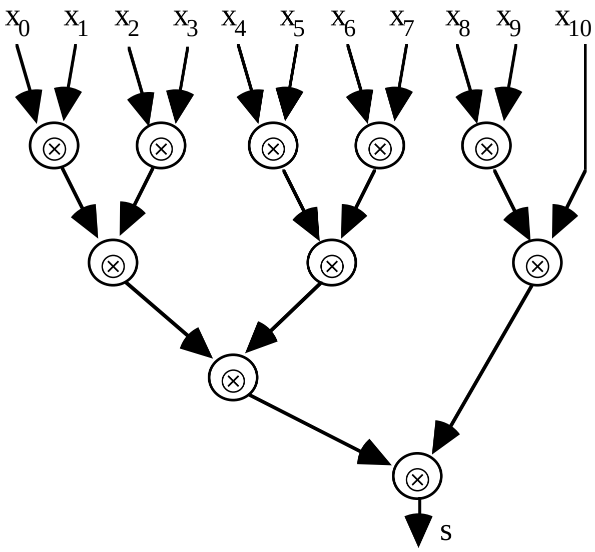
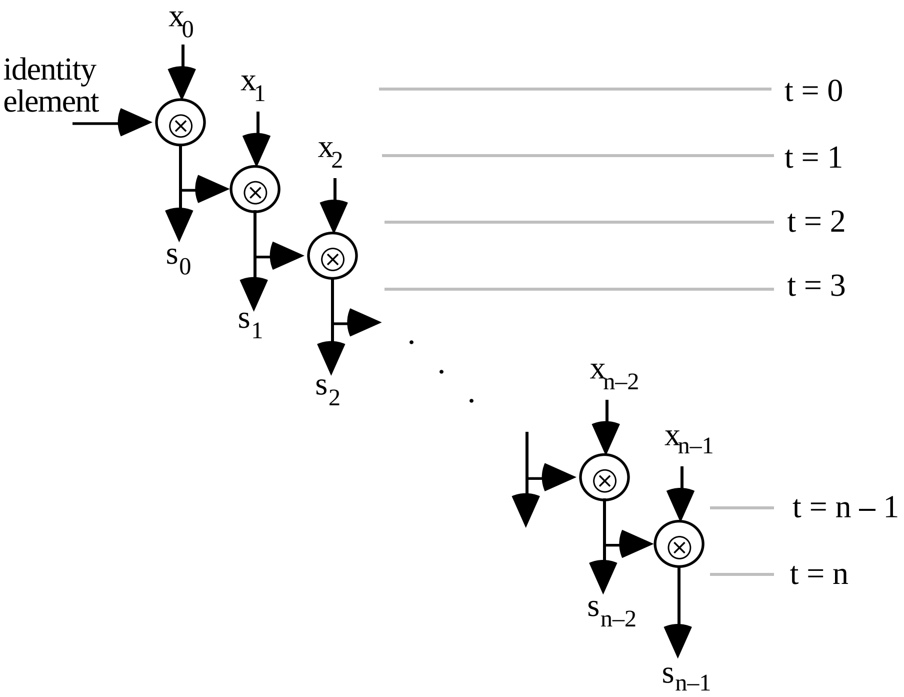


Fig. 2.1. Semigroup computation on a uniprocessor.



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

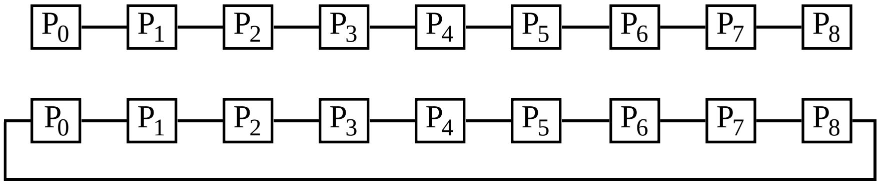


Fig. 2.2. A linear array of nine processors and its ring variant.

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

(Max) Node degree: $d = 2$

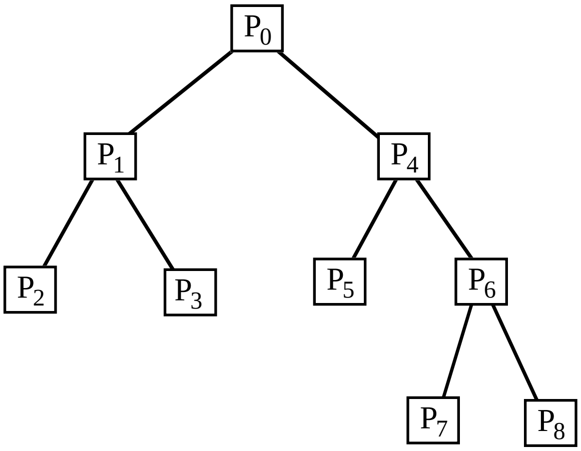


Fig. 2.3. A balanced (but incomplete) binary tree of nine processors.

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

(Max) Node degree: $d = 3$

We almost always deal with complete binary trees:

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

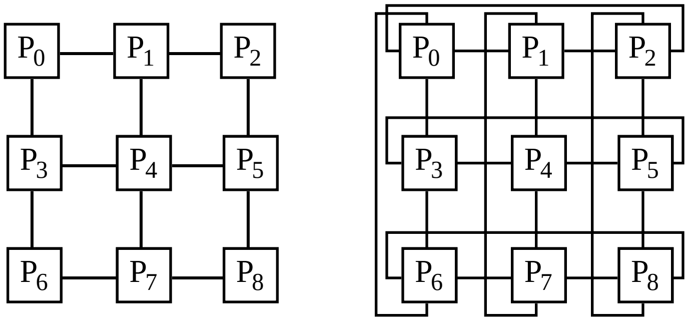


Fig. 2.4. A 2D mesh of nine processors and its torus variant.

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

(Max) Node degree: $d = 4$

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

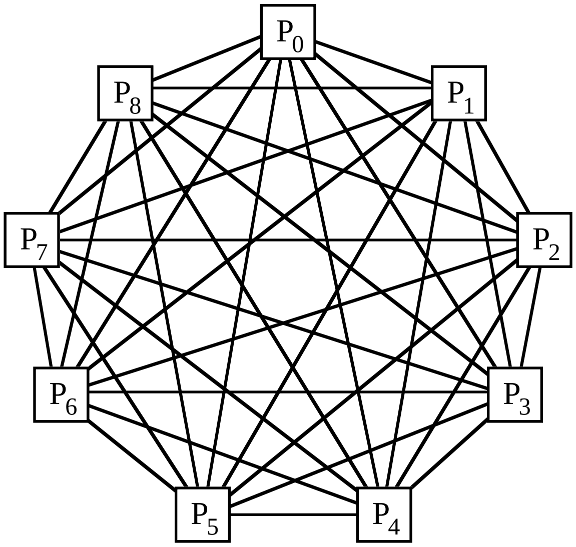
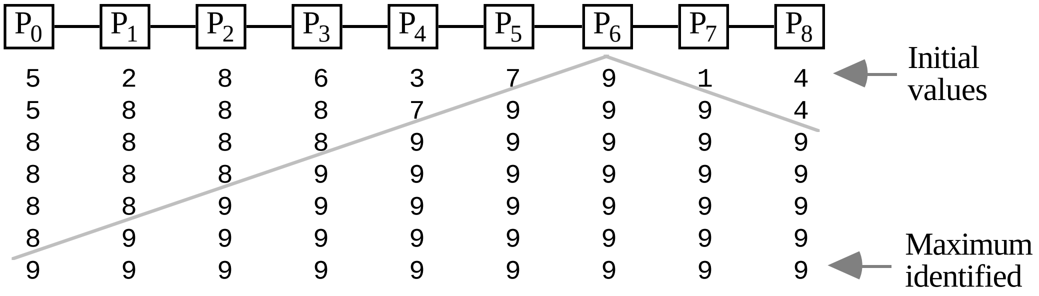


Fig. 2.5. A shared-variable architecture modeled as a complete graph.

Diameter of complete graph: $D = 1$

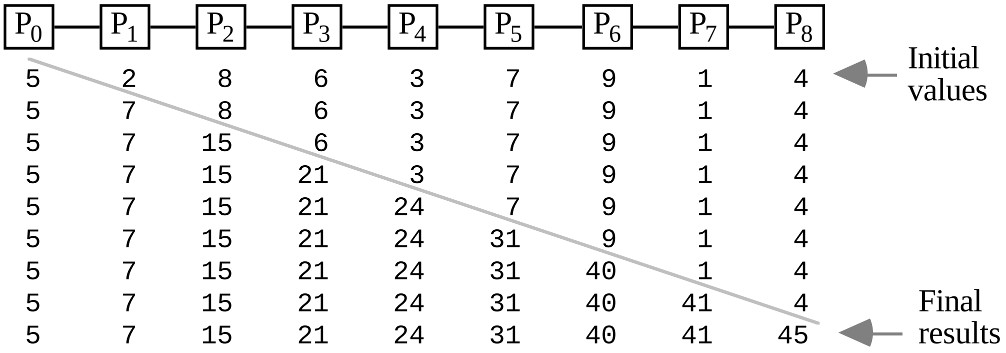
(Max) Node degree: $d = p - 1$

2.3 Algorithms for a Linear Array



P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
5	2	8	6	3	7	9	1	4
5	8	8	8	7	9	9	9	4
8	8	8	8	9	9	9	9	9
8	8	8	9	9	9	9	9	9
8	8	9	9	9	9	9	9	9
8	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9

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



P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
5	2	8	6	3	7	9	1	4
5	7	8	6	3	7	9	1	4
5	7	15	6	3	7	9	1	4
5	7	15	21	3	7	9	1	4
5	7	15	21	24	7	9	1	4
5	7	15	21	24	31	9	1	4
5	7	15	21	24	31	40	1	4
5	7	15	21	24	31	40	41	4
5	7	15	21	24	31	40	41	45

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

Diminished prefix computation: the ith result excludes the ith element (e.g., sum of the first i – 1 elements)

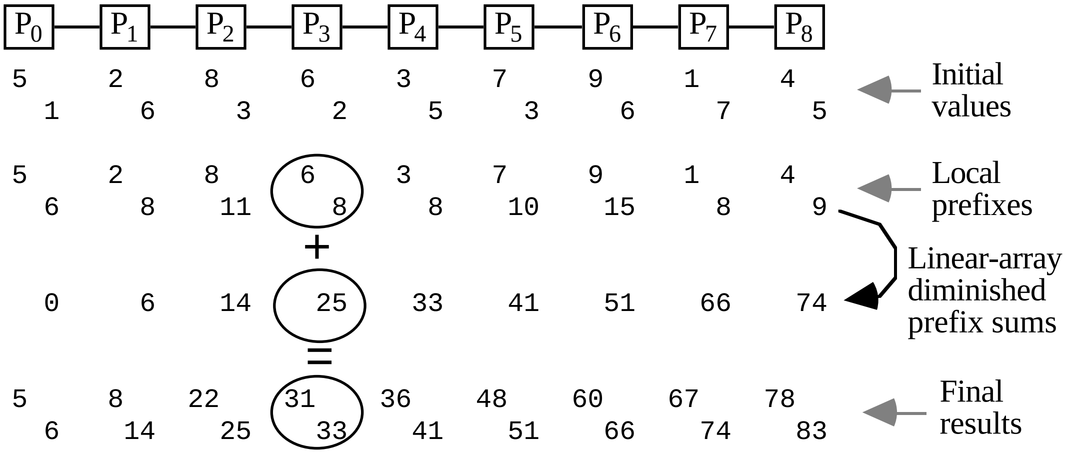


Fig. 2.8. Computing prefix sums on a linear array with two items per processor.

Packet routing or broadcasting:

right- and left-moving packets have no conflict

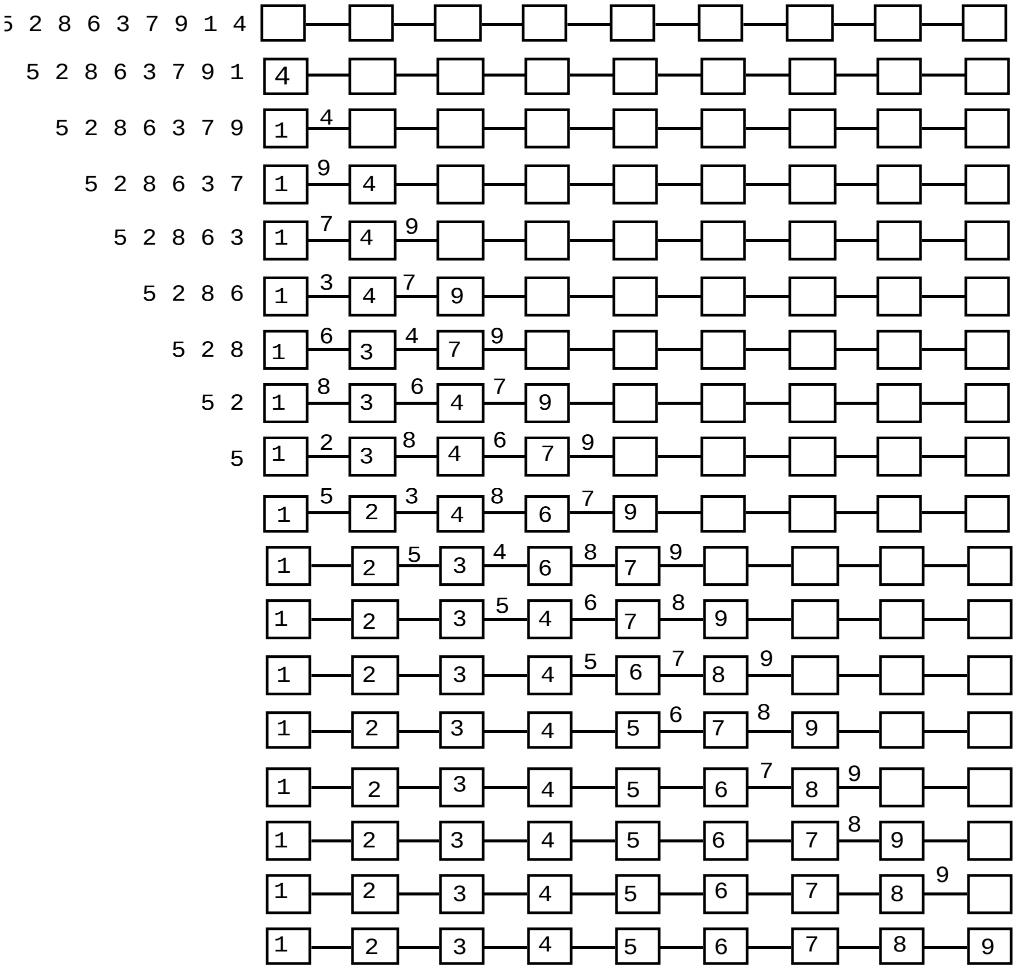
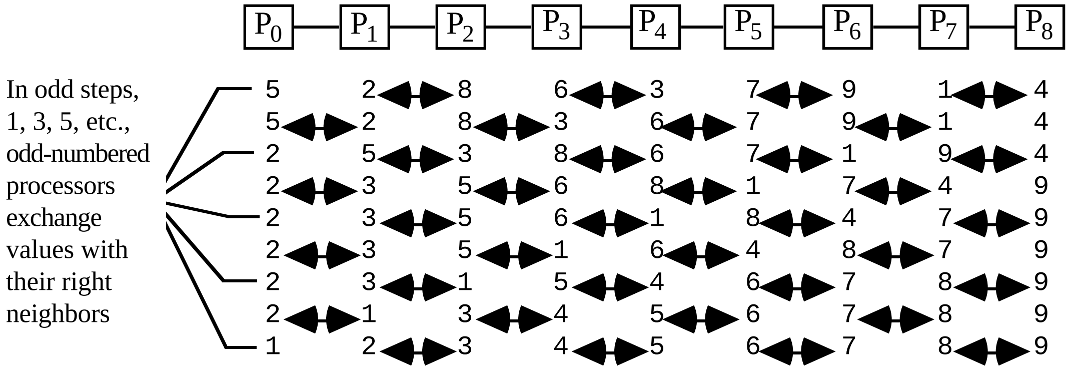


Fig. 2.9. Sorting on a linear array with the keys input sequentially from the left.



P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
5	2	8	6	3	7	9	1	4
5	2	8	3	6	7	9	1	4
2	5	3	8	6	7	1	9	4
2	3	5	6	8	1	7	4	9
2	3	5	6	1	8	4	7	9
2	3	5	1	6	4	8	7	9
2	3	1	5	4	6	7	8	9
2	1	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

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)

Efficinecy = O((log p) / p)

Redundancy = O(p / (log p))

Utilization = 1/2

2.4 Algorithms for a Binary Tree

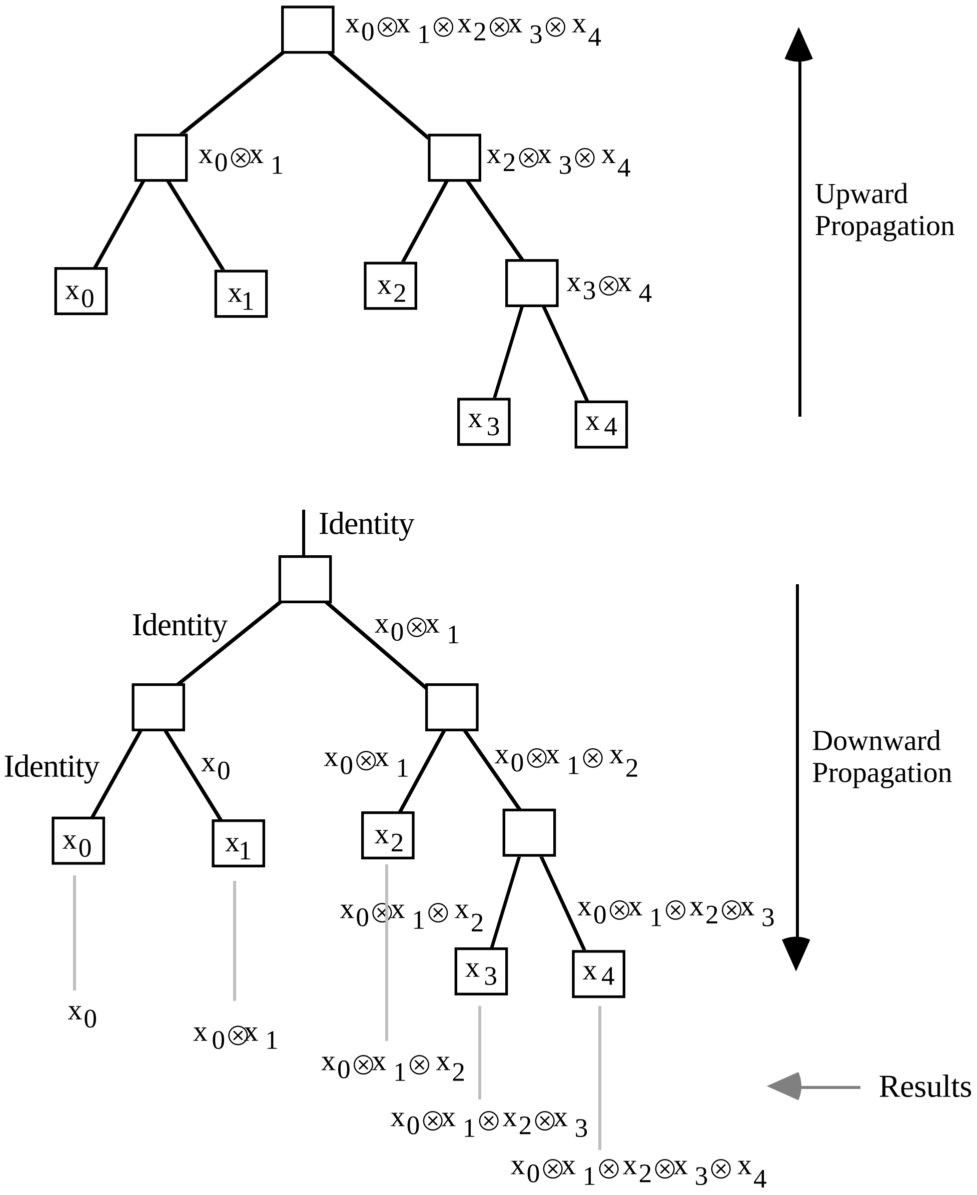

$x_0 \otimes x_1 \otimes x_2 \otimes x_3 \otimes x_4$
$x_0 \otimes x_1$
$x_2 \otimes x_3 \otimes x_4$
$x_3 \otimes x_4$
x_0
x_1
x_2
x_3
x_4
Upward Propagation
Identity
Identity
$x_0 \otimes x_1$
Identity
x_0
$x_0 \otimes x_1$
$x_0 \otimes x_1 \otimes x_2$
x_0
x_1
x_2
$x_0 \otimes x_1 \otimes x_2$
$x_0 \otimes x_1 \otimes x_2 \otimes x_3$
x_3
x_4
Downward Propagation
x_0
$x_0 \otimes x_1$
$x_0 \otimes x_1 \otimes x_2$
$x_0 \otimes x_1 \otimes x_2 \otimes x_3$
$x_0 \otimes x_1 \otimes x_2 \otimes x_3 \otimes x_4$
Results

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**
Diminished 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:

p g a g g p p p g a c_{in}

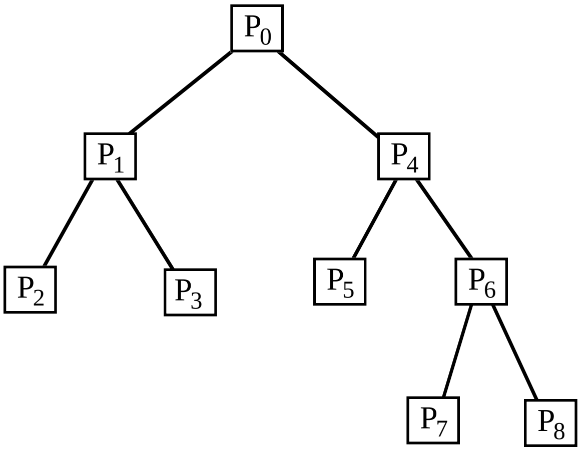
← **g or a**

direction of indexing

Parallel prefix computation using the carry operator “¢”

p ¢ x = x	**x propagates over p, for all x $\in$ {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

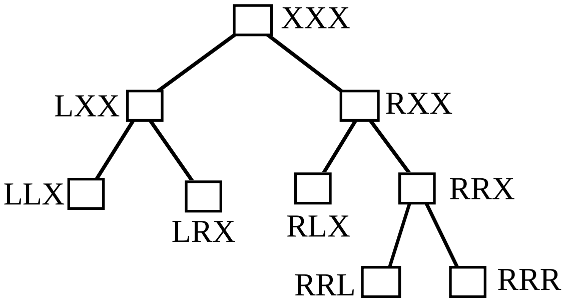


A balanced binary tree with preorder node indices.

maxl (maxr) = largest node number in the 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

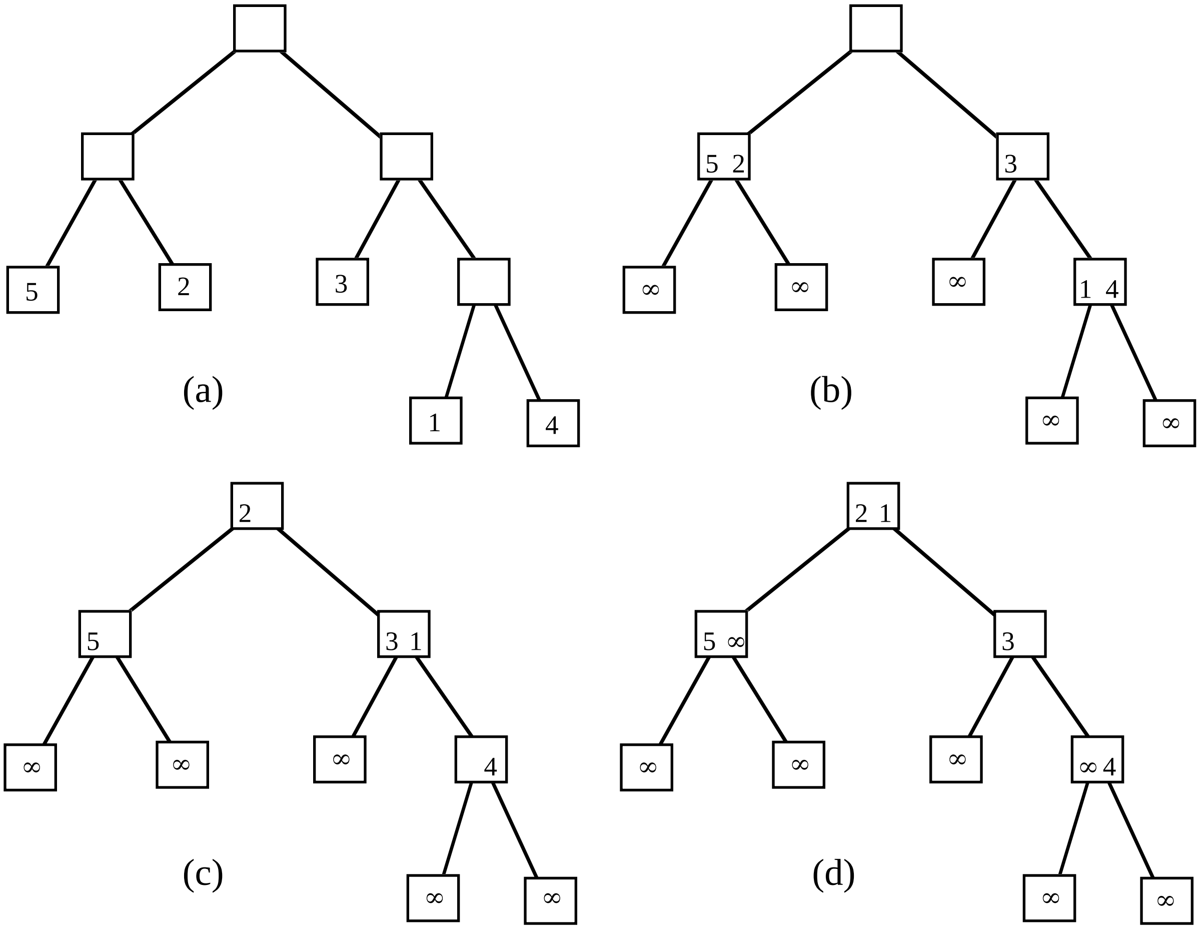


Fig. 2.12. The first few steps of the sorting algorithm on a binary tree.

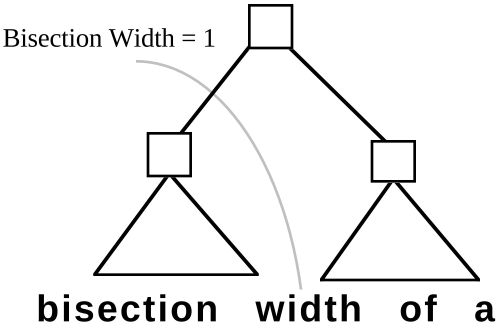
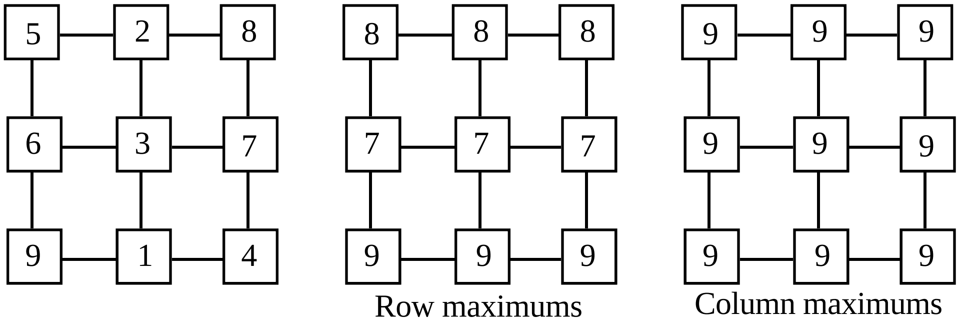
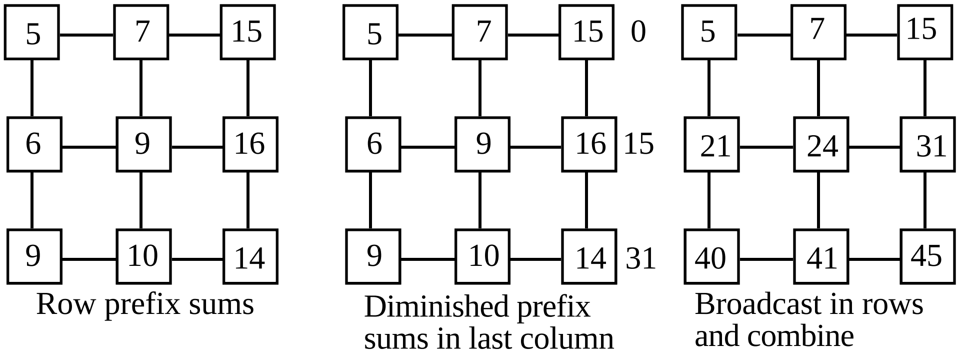


Fig. 2.13. The bisection width of a binary tree architecture.

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 the operator is not commutative

Routing and broadcasting done via row/column operations

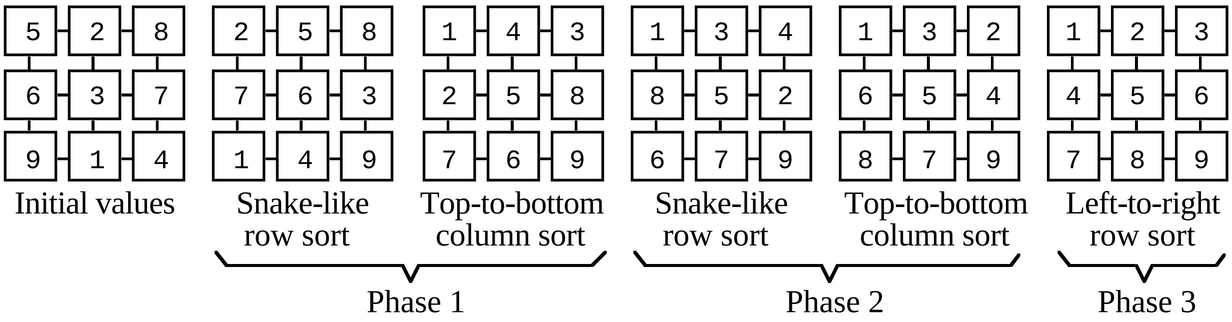


Fig. 2.14. The shearsort algorithm on a 3 × 3 mesh.

2.6 Algorithms with Shared Variables

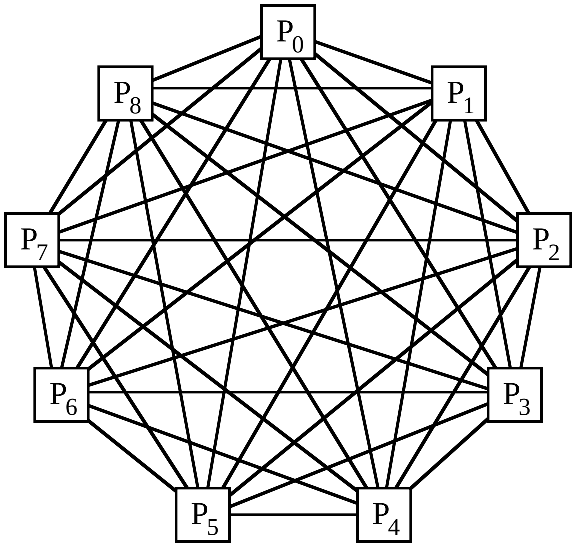


Fig. 2.5. A shared-variable architecture modeled as a complete graph.

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

Packet routing and broadcasting: one step, assuming all-port communication

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

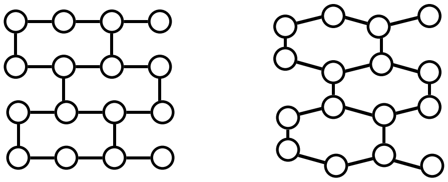


Figure for Problem 2.13.

3 Parallel Algorithm Complexity

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

$f(n) = O(g(n))$ 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

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

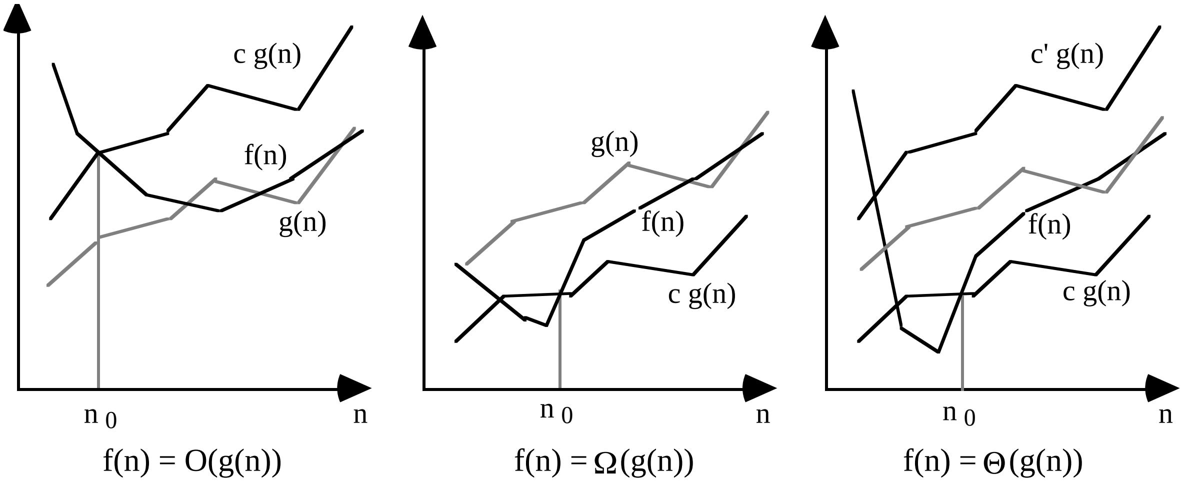


Fig. 3.1. Graphical representation of the notions of asymptotic complexity.

$f(n) = o(g(n))$	$<$	Growth rate strictly less than
$f(n) = O(g(n))$	$\le$	Growth rate no greater than
$f(n) = \Theta(g(n))$	$=$	Growth rate the same as
$f(n) = \Omega(g(n))$	$\ge$	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 (K = 1000, M = 1 000 000)

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	1 yr	1 yr
1M	3 yr	11 yr	3 yr	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

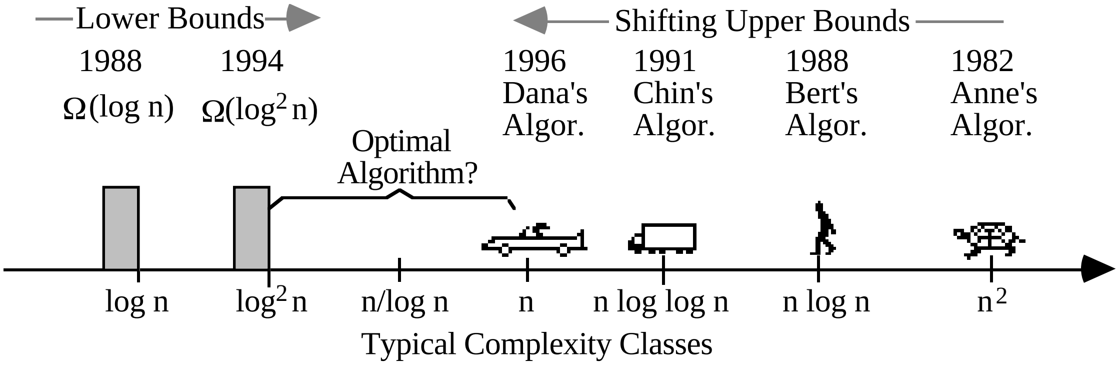


Fig. 3.2. Upper & lower bounds may tighten over time.

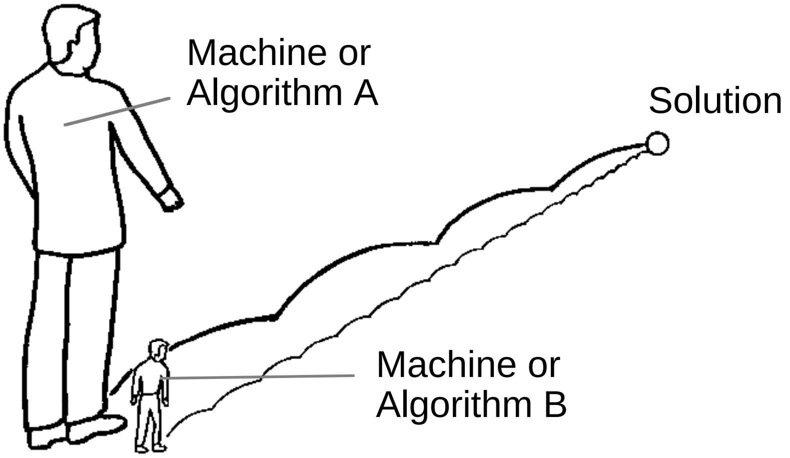
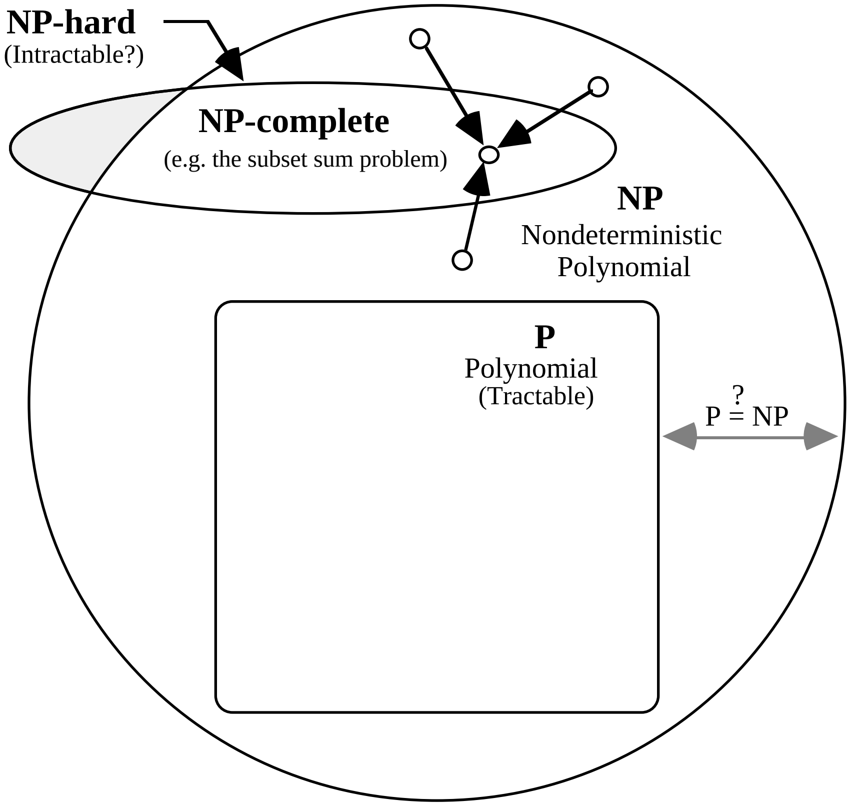


Fig. 3.3. Five times fewer steps does not necessarily mean five times faster.

3.3 Complexity Classes



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

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 in the set add up to s.

This 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 of these trials takes only one picosecond,
the problem is virtually unsolvable for n = 100.

3.4 Parallelizable Tasks and the NC Class

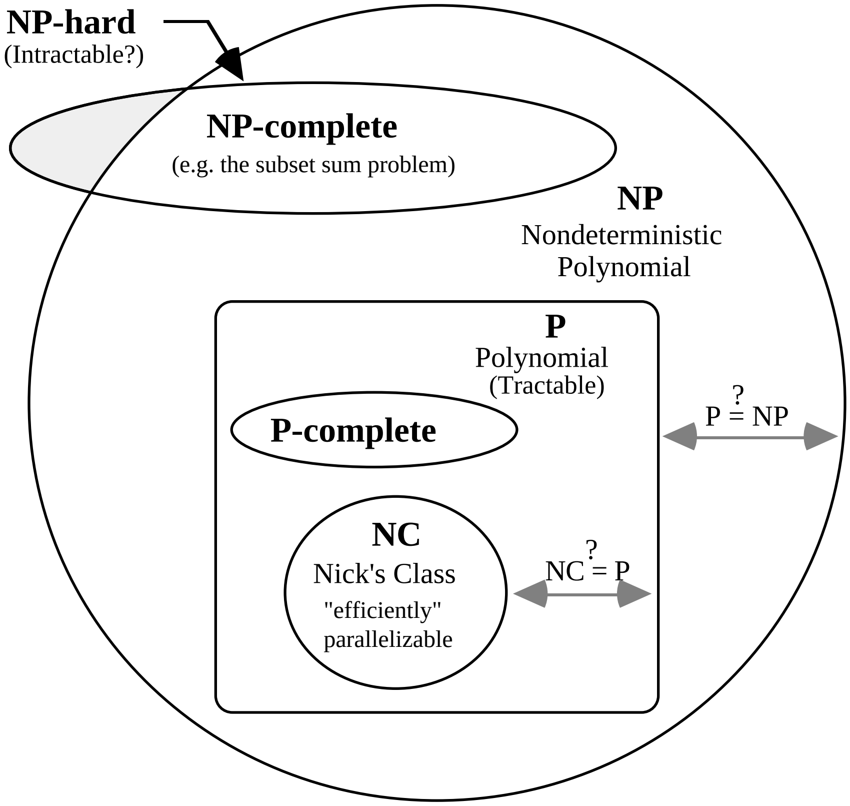


Fig. 3.4. A conceptual view of complexity classes and their relationships.

NC (Nick's class, Niclaus Pippenger)
Problems solvable in polylogrithmic time ($T = O(\log^k n)$)
using a polynomially bounded number of processors

Example P-complete problem: the circuit-value problem

Given a logic circuit with known inputs,
determine its output.

The circuit-value problem is obvioudly 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

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

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 ⇒ more accurate results

3.6 Solving Recurrences

Solution via unrolling

1. $f(n) = f(n-1) + n$ {Rewrite $f(n-1)$ as $f((n-1)-1) + n - 1$}

$$\begin{aligned}
&= f(n-2) + n - 1 + n \\
&= f(n-3) + n - 2 + n - 1 + n \\
&\ldots \\
&= f(1) + 2 + 3 + \cdots + n - 1 + n \\
&= n(n+1)/2 - 1 \\
&= \Theta(n^2)
\end{aligned}$$

2. $f(n) = f(n/2) + 1$ {Rewrite $f(n/2)$ as $f((n/2)/2 + 1$}

$$\begin{aligned}
&= f(n/4) + 1 + 1 \\
&= f(n/8) + 1 + 1 + 1 \\
&\ldots \\
&= f(n/n) + \underbrace{1 + 1 + 1 + \cdots + 1}_{\text{------ } \log_2 n \text{ times ------}} \\
&= \log_2 n \\
&= \Theta(\log n)
\end{aligned}$$

3. $f(n) = 2f(n/2) + 1$

$$\begin{aligned}
&= 4f(n/4) + 2 + 1 \\
&= 8f(n/8) + 4 + 2 + 1 \\
&\ldots \\
&= n\, f(n/n) + n/2 + \cdots + 4 + 2 + 1 \\
&= n - 1 \\
&= \Theta(n)
\end{aligned}$$

4. $$\begin{aligned} f(n) &= f(n/2) + n \\ &= f(n/4) + n/2 + n \\ &= f(n/8) + n/4 + n/2 + n \\ &\cdots \\ &= f(n/n) + 2 + 4 + \cdots + n/4 + n/2 + n \\ &= 2n - 2 \quad = \Theta(n) \end{aligned}$$

5. $$\begin{aligned} f(n) &= 2f(n/2) + n \\ &= 4f(n/4) + n + n \\ &= 8f(n/8) + n + n + n \\ &\cdots \\ &= n\,f(n/n) + \underbrace{n + n + n + \cdots + n}_{\text{------ } \log_2 n \text{ times ------}} \\ &= n \log_2 n \quad = \Theta(n \log n) \end{aligned}$$

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

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

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

6. $$\begin{aligned} f(n) &= f(n/2) + \log_2 n \\ &= f(n/4) + \log_2(n/2) + \log_2 n \\ &= f(n/8) + \log_2(n/4) + \log_2(n/2) + \log_2 n \\ &\cdots \\ &= f(n/n) + \log_2 2 + \log_2 4 + \cdots + \log_2(n/2) + \log_2 n \\ &= 1 + 2 + 3 + \cdots + \log_2 n \\ &= \log_2 n\,(\log_2 n + 1)/2 \;=\; \Theta(\log^2 n) \end{aligned}$$

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

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

The solution to the original recurrence then becomes

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

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

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

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

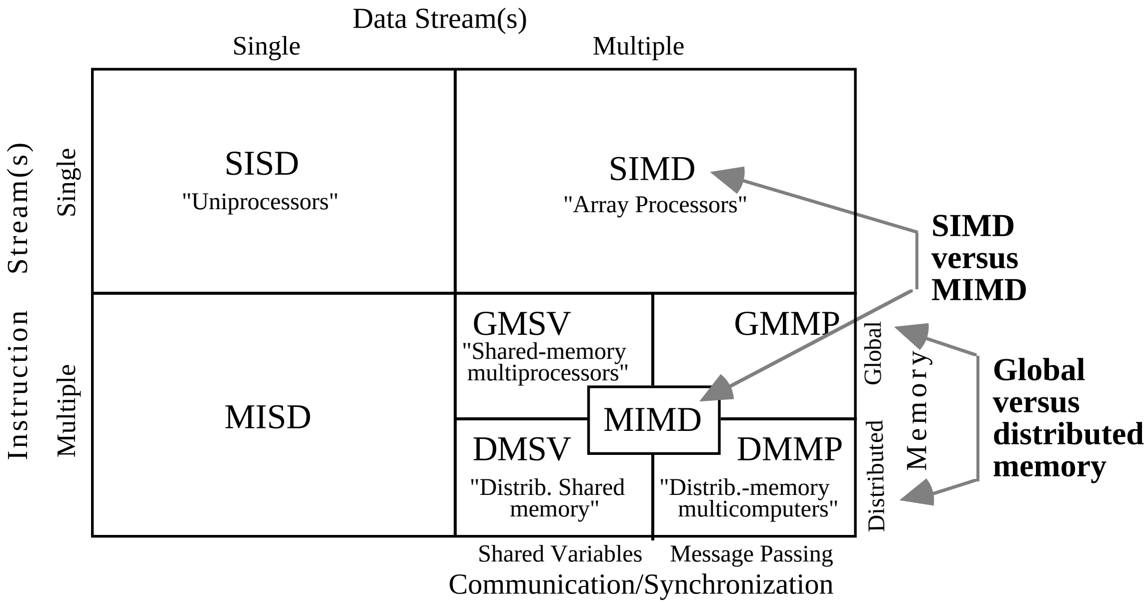


Fig. 4.1. The Flynn-Johnson classification of computer systems.

MISD can be viewed as a flexible (programmable) pipeline

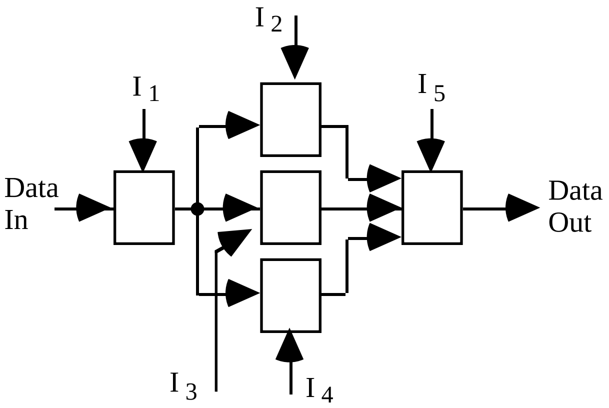


Fig. 4.2. Multiple instruction streams operating on a single data stream (MISD).

4.2 SIMD versus MIMD Architectures

Most early parallel machines were of SIMD type

Synchronous SIMD

- To perform 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 your 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

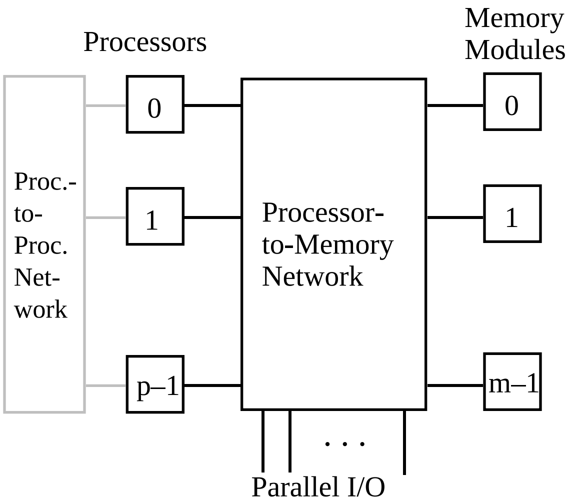


Fig. 4.3. A parallel processor with global memory.

Example processor-to-memory/processor networks:

1. Crossbar; p × 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

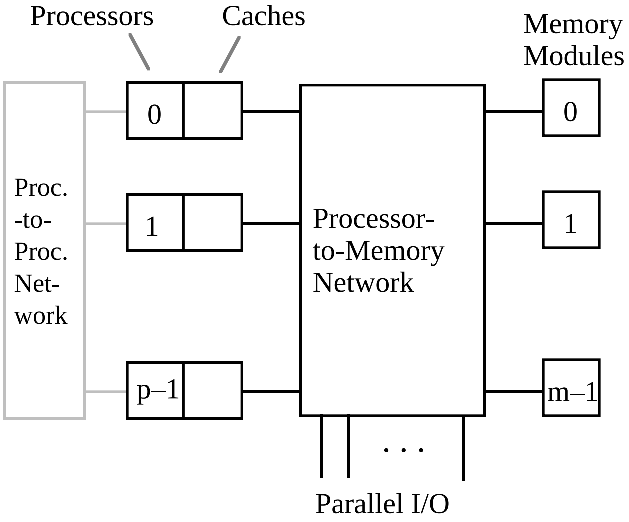


Fig. 4.4. A parallel processor with global memory and processor caches.

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)

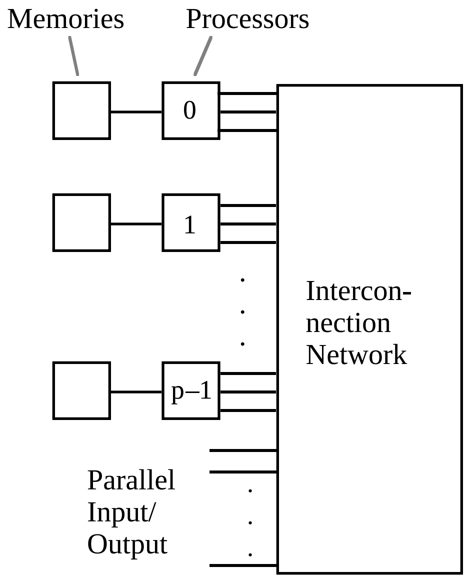


Fig. 4.5. A parallel processor with distributed memory.

Examples networks for distributed memory machines

1. Crossbar; cost too high for massively parallel systems
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

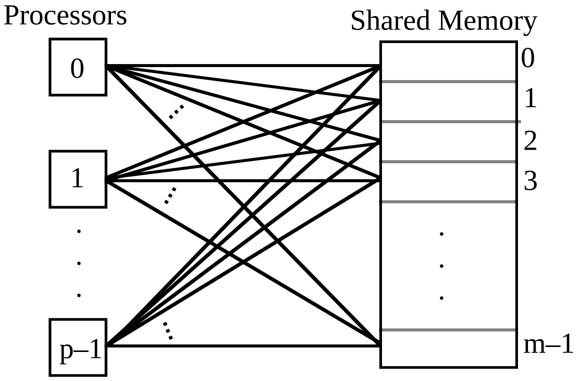


Fig. 4.6. Conceptual view of a parallel random-access machine (PRAM).

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

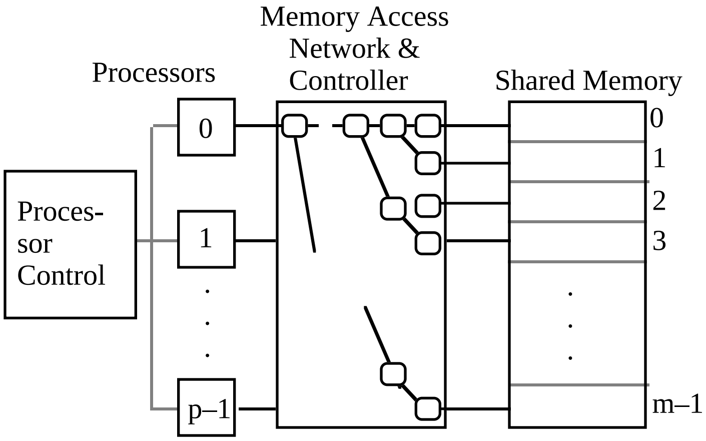


Fig. 4.7. PRAM with some hardware details shown.

In practice, memory is divided into modules and simultaneous accesses to the 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.

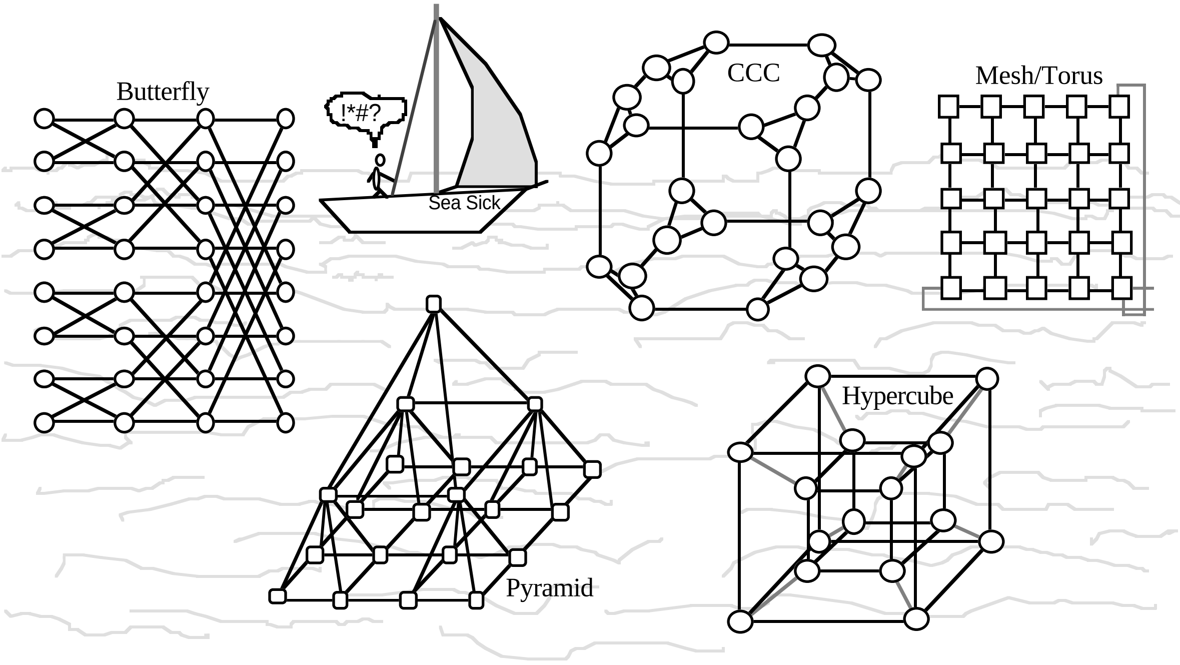


Fig. 4.8. The sea of interconnection networks.

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

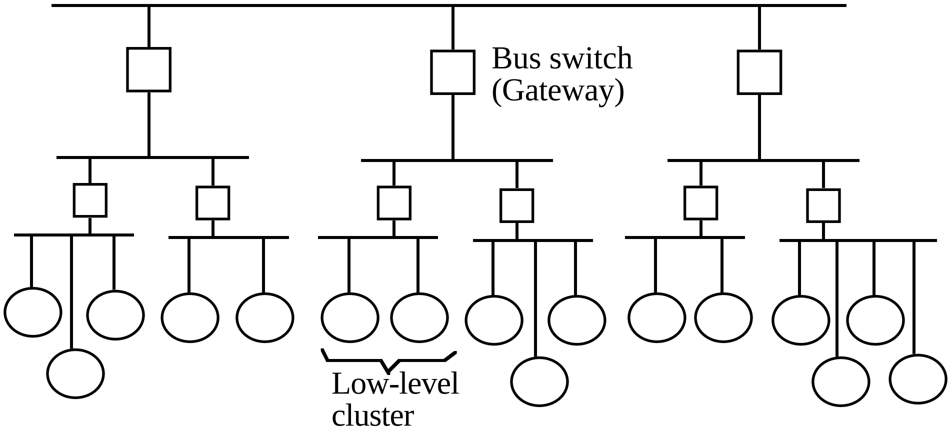


Fig. 4.9. Example of a hierarchical interconnection architecture.

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 the 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

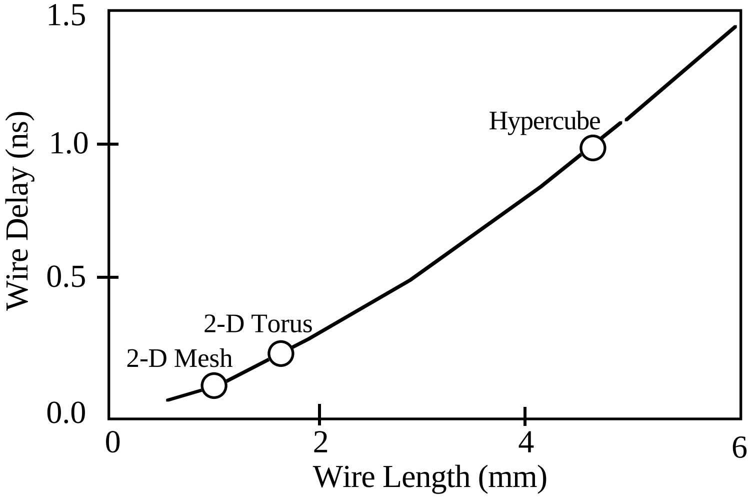
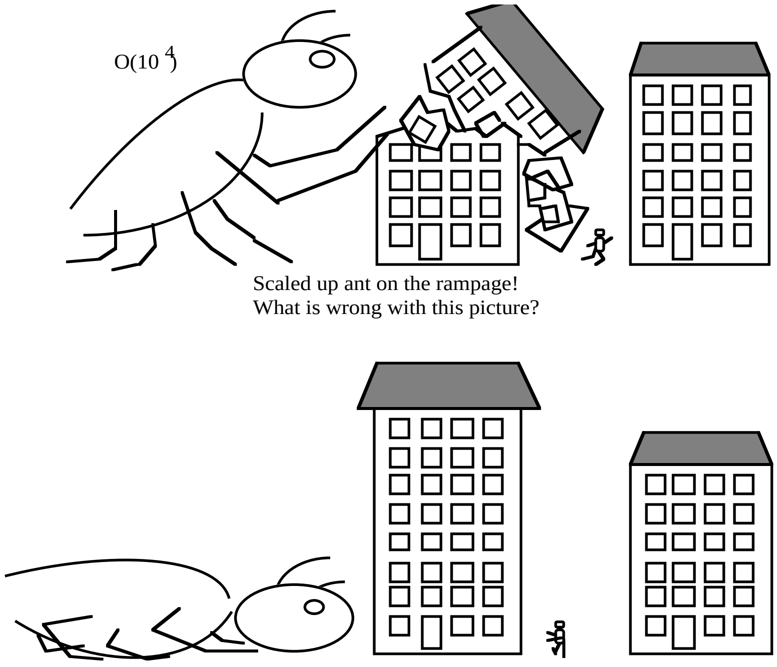


Fig. 4.10. Intrachip wire delay as a function of wire length.



Scaled up ant on the rampage!
What is wrong with this picture?

Scaled up ant collapses under own weight.

Fig. 4.11. Pitfalls of scaling up.

Part II Extreme Models

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 everthing 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

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

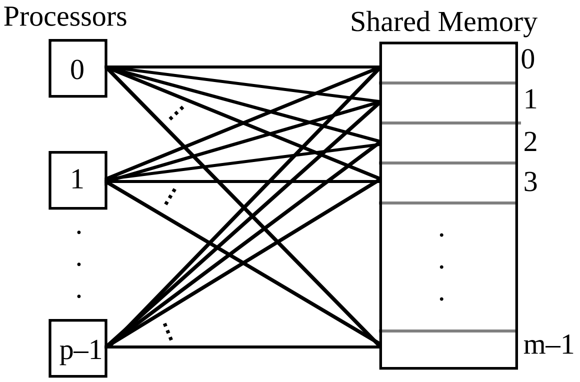
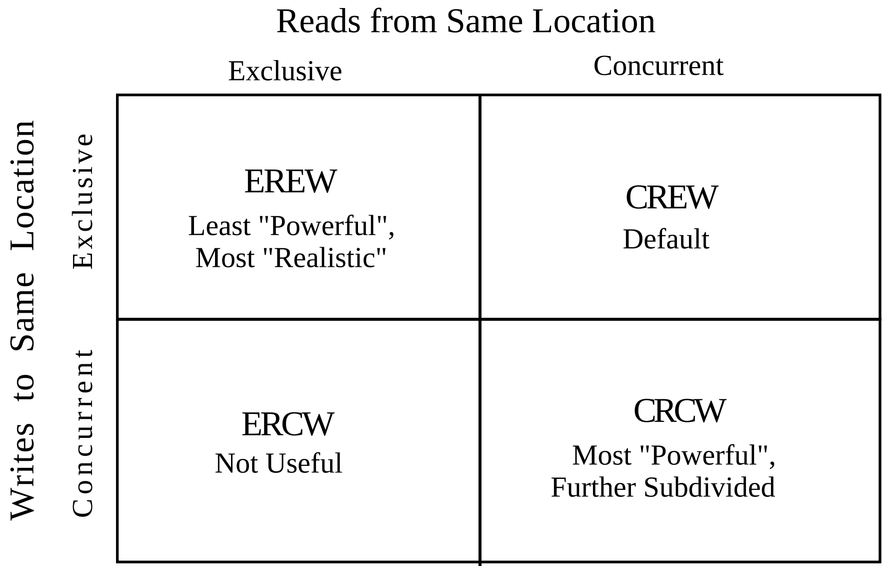


Fig. 4.6. Conceptual view of a parallel random-access machine (PRAM).

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



		Reads from Same Location	
		Exclusive	Concurrent
Writes to Same Location	Exclusive	EREW Least "Powerful", Most "Realistic"	CREW Default
	Concurrent	ERCW Not Useful	CRCW Most "Powerful", Further Subdivided

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.

One way to order these submodels is by their computational power:

EREW < CREW < CRCW-D
< CRCW-C < CRCW-R < 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 $\Theta(\log p)$.

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, $0 \le j < p$, do
    Copy B[j] into B[$j + 2^k$]
endfor
```

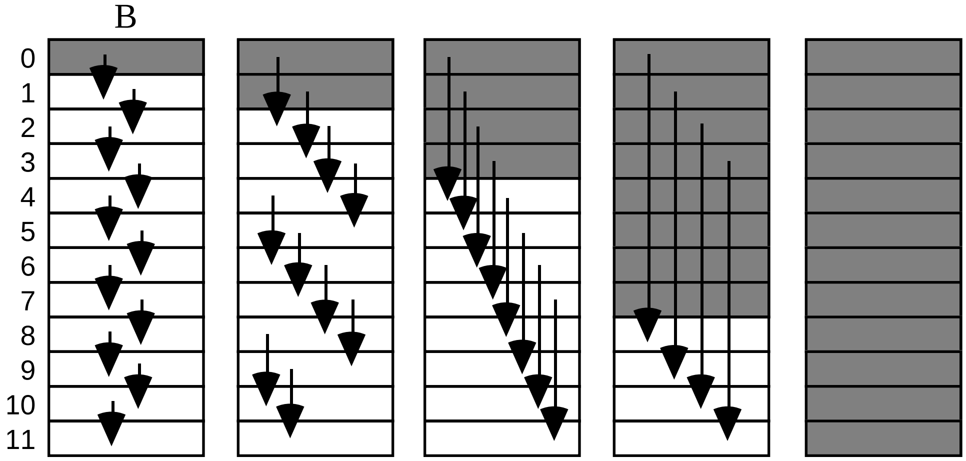


Fig. 5.2. Data broadcasting in EREW PRAM via recursive doubling.

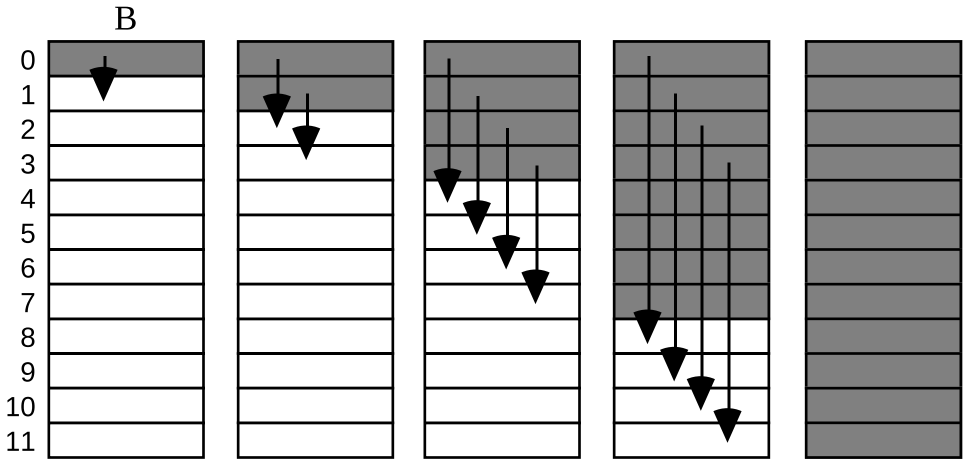


Fig. 5.3. EREW PRAM data broadcasting without redundant copying.

```
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 := 2s
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)

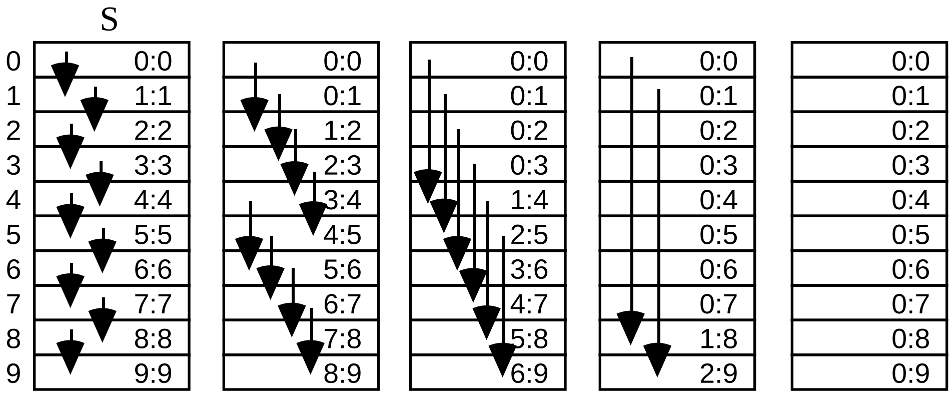
In the following naive sorting algorithm, processor j determines the rank R[j] of its data element S[j] by examining all the other data elements; it then writes S[j] in element R[j] of the output (sorted) vector

```
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 algorithms 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 ⊗



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

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] ⊗ S[j + s]
      s := 2s
endwhile
Broadcast S[p – 1] to all processors
```

The preceding algorithm is time-optimal (CRCW can do better: problem 5.16)

Speed-up = $p/\log_2 p$

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

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

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 $p = \Theta(n)$, a sublinear speedup of $\Theta(n/\log n)$ is obtained

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

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

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

$\text{Efficiency}(n, p) = \Theta(1)$

Using fewer processors than tasks = parallel slack

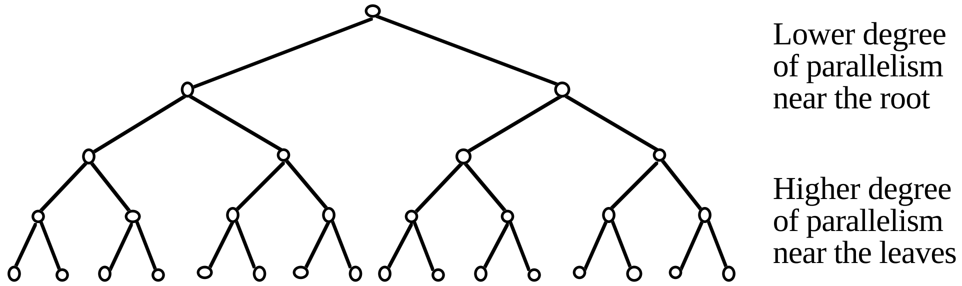

Lower degree of parallelism near the root
Higher degree of parallelism near the leaves

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

5.4 Parallel Prefix Computation

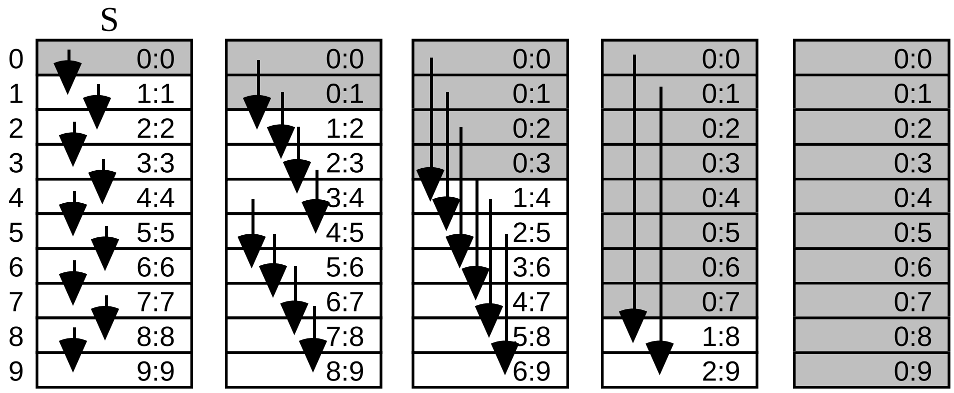


Fig. 5.6. Parallel prefix computation in EREW PRAM via recursive doubling.

Two other solutions, based on divide and conquer

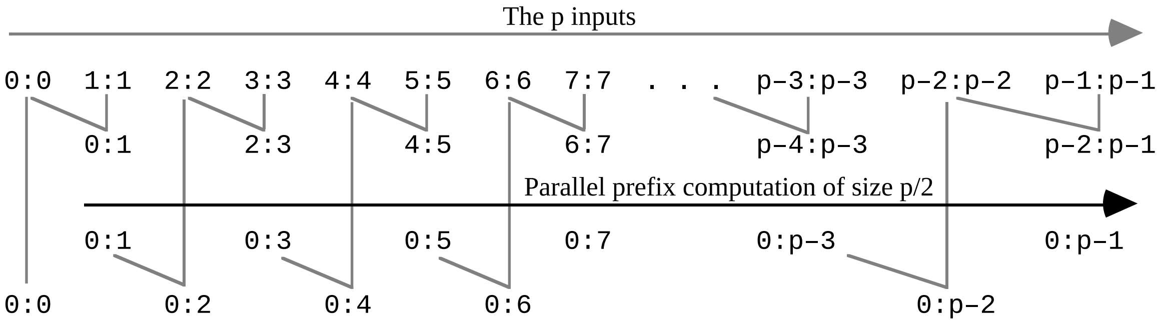


Fig. 5.7 Parallel prefix computation using a divide-and-conquer scheme.

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

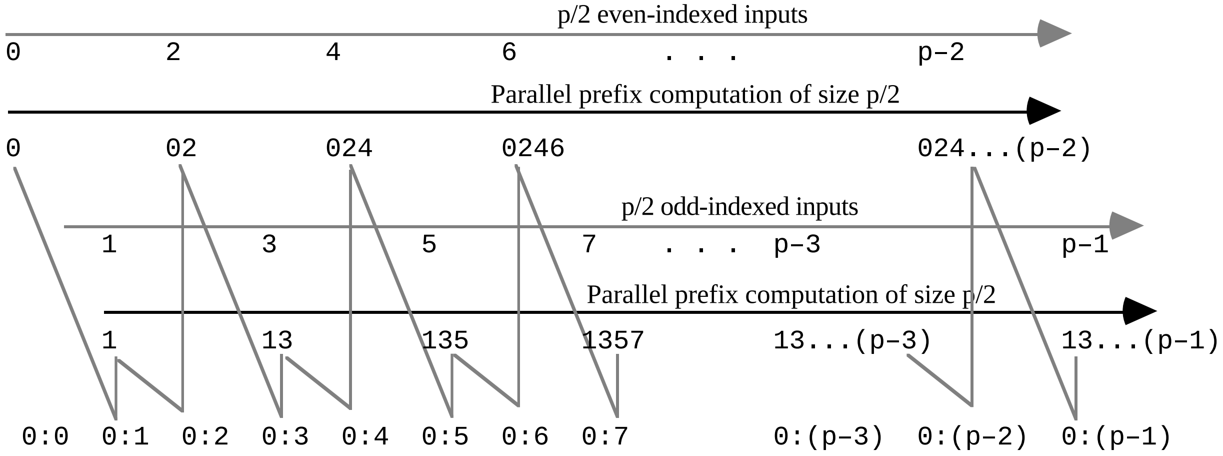


Fig. 5.8. Another divide-and-conquer scheme for parallel prefix computation.

$$T(p) = T(p/2) + 1 = \log_2 p \qquad \text{Requires commutativity}$$

5.5 Ranking the Elements of a Linked List

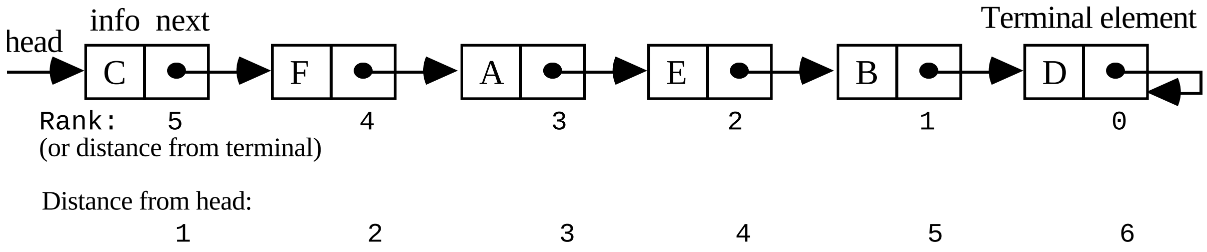
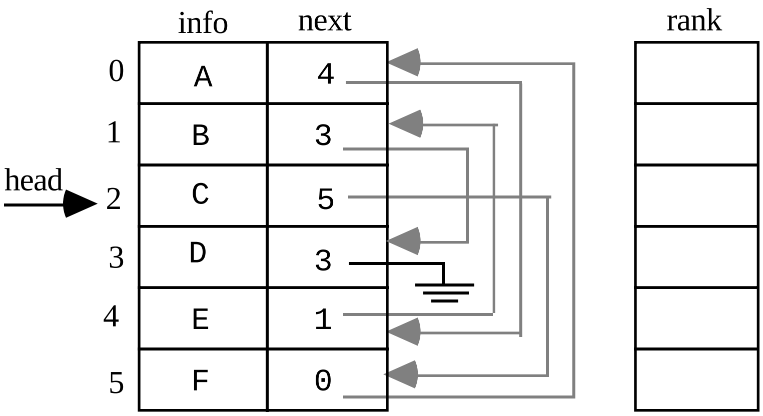


Fig. 5.9. Example linked list and the ranks of its elements.



	info	next	rank
0	A	4	
1	B	3	
2	C	5	
3	D	3	
4	E	1	
5	F	0	

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 appear to be unparallizable

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

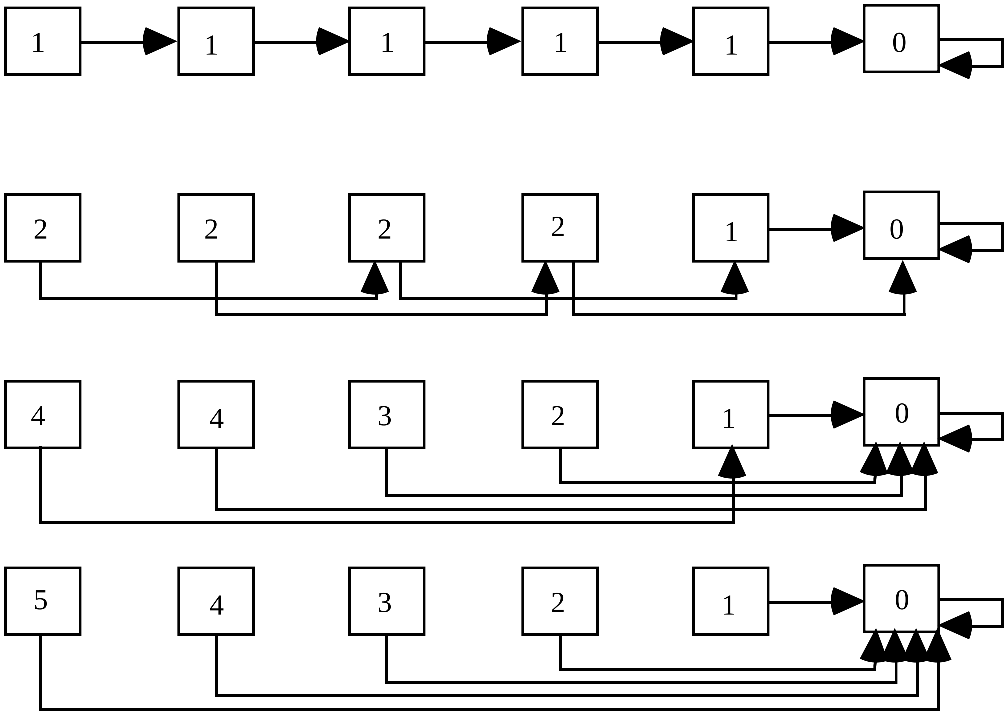


Fig. 5.11. Element ranks initially and after each of the three iterations.

PRAM list ranking algorithm (via pointer jumping)

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

Which PRAM submodel is implicit in the preceding algorithm?

5.6 Matrix Multiplication

For m × m matrices, C = A × 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
```

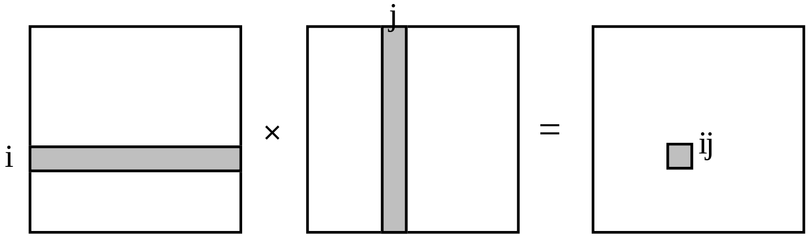
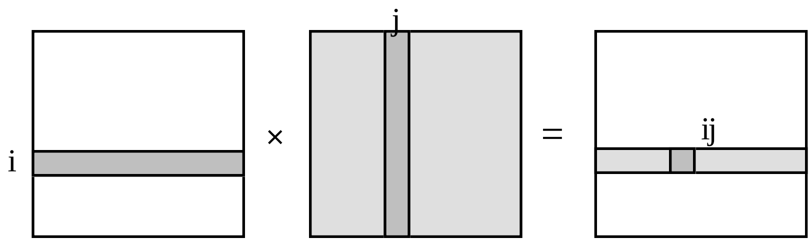


Fig. 5.12. PRAM matrix multiplication by using p = m² processors.

PRAM matrix multiplication algorithm using m² 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
```

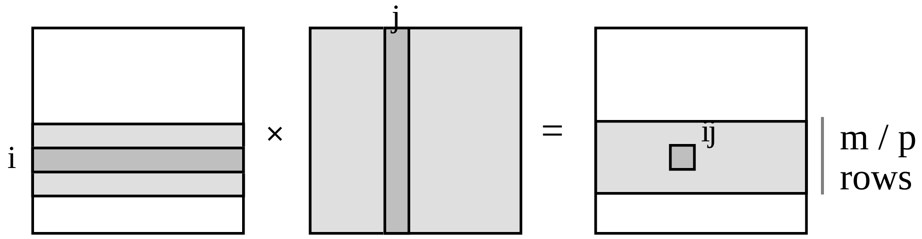


PRAM matrix multiplication algorithm using m processors

```
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

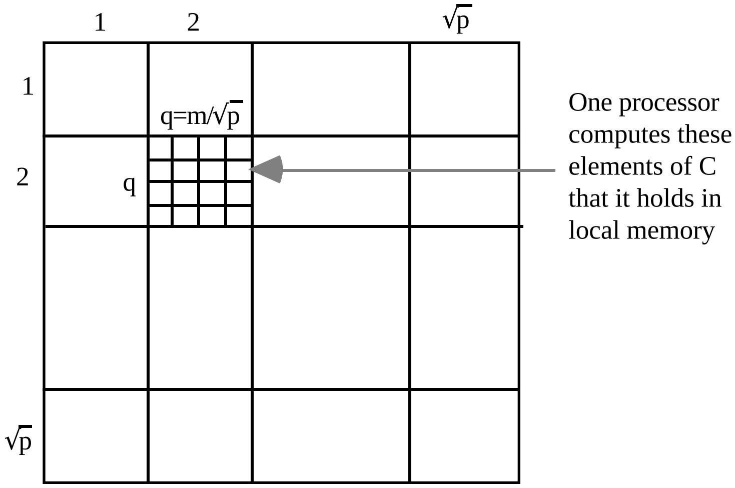


The preceding solution is not efficient for NUMA parallel architectures

Each element of B is fetched m/p times

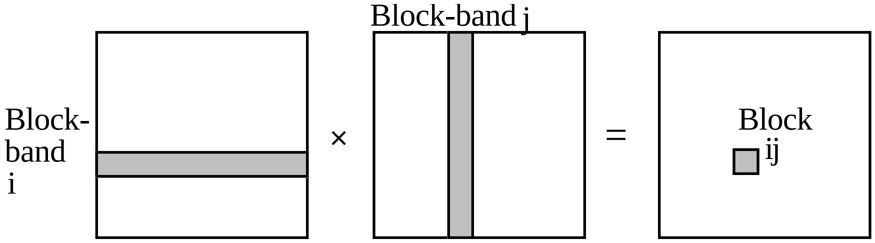
For each such data access, only two arithmetic operations are performed

Block matrix multiplication



One processor computes these elements of C that it holds in local memory

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



Each multiply-add computation on q × 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

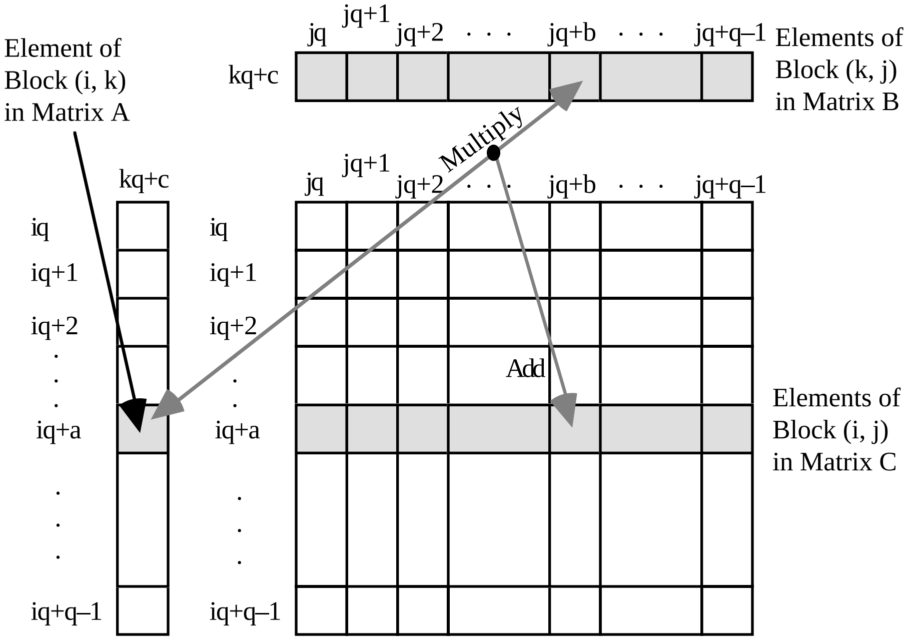


Fig. 5.14. How Processor (i, j) operates on an element of A and one block-row of B to update one block-row of C.

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 larger 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

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

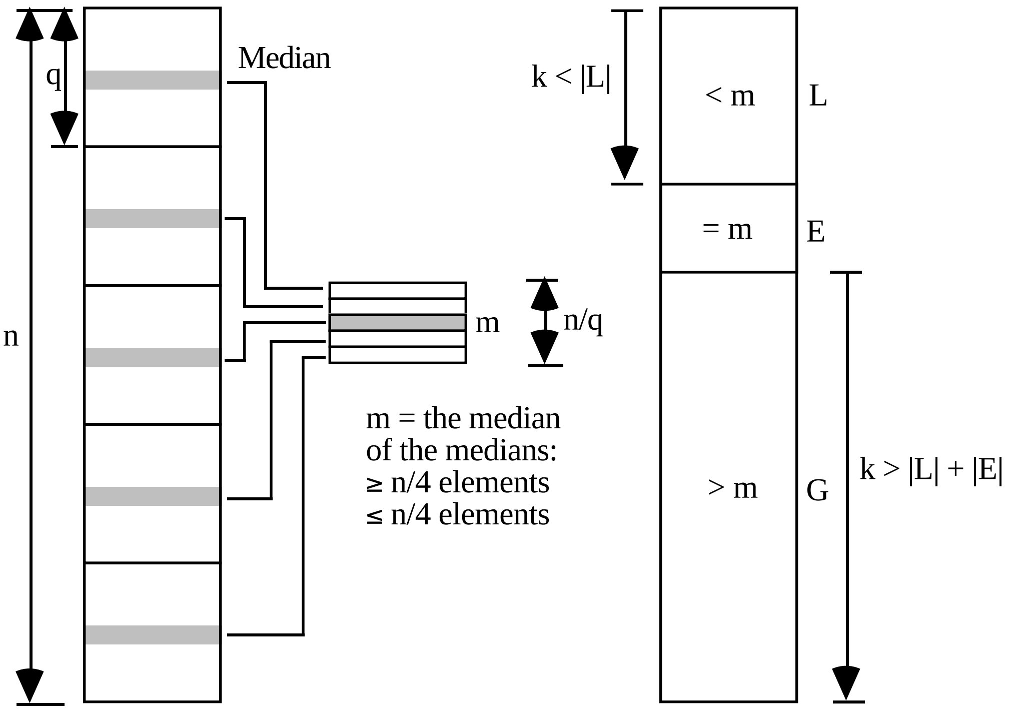
Chapter Contents

6.1 Sequential Rank-Based Selection

Selection: Find a (the) kth 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| ≥ k
    then return select(L, k)
    else if |L| + |E| ≥ k
         then return m
         else return select(G, k – |L| – |E|)
    endif
```

Analysis:

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

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

```
    ←----------- n/q sublists of q elements -----------→
S   6 4 5 6 7  1 5 3 8 2  1 0 3 4 5  6 2 1 7 1  4 5 4 9 5
    ---------  ---------  ---------  ---------  ---------
T         6        3          3        2          5
m                             3
    1 2 1 0 2 1 1   3 3   6 4 5 6 7 5 8 4 5 6 7 4 5 4 9 5
    -------------   ---   -------------------------------
          L          E                   G
       |L|=7       |E|=2               |G|=16
```

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

```
S   1 2 1 0 2  1 1
    ---------  ---
T       1        1
m                1
    0   1 1 1 1   2 2
    -   -------   ---
    L      E       G                      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

```
S   6 4 5 6 7  5 8 4 5 6  7 4 5 4 9  5
    ---------  ---------  ---------  -
T         6          5        5      5
m                    5
    4 4 4 4   5 5 5 5 5   6 6 7 8 6 7 9
    -------   ---------   -------------
       L          E             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., $x = 1/2 \Rightarrow p = \sqrt{n}$

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

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

$$\text{Efficiency} = \Omega(1)$$

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

6.3 A Selection-Based Sorting Algorithm

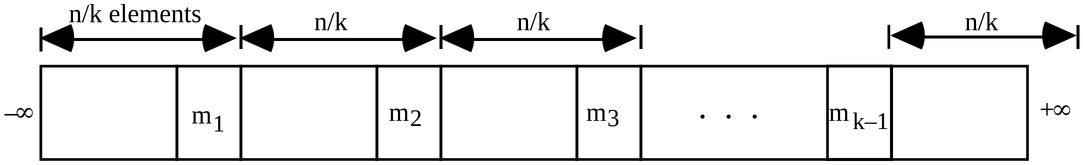


Fig. 6.1. Partitioning of the sorted list for selection-based sorting.

```
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 $m_0$ := –∞ ;  $m_k$ := +∞}
    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: Let $p = n^{1-x}$, 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 handled 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:

$m_0 = -\infty$

$n/k = 25/4 \approx 6$ $m_1 = \text{PRAMselect}(S,\ 6,\ 5) = 2$

$2n/k = 50/4 \approx 13$ $m_2 = \text{PRAMselect}(S,\ 13,\ 5) = 4$

$3n/k = 75/4 \approx 19$ $m_3 = \text{PRAMselect}(S,\ 19,\ 5) = 6$

$m_4 = +\infty$

```
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 establish k comparison thresholds

In fact, it would be 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, $0 \le j < p$, 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, $0 \le j < p$, store its elements falling
    in $(m_i\,, m_{i+1})$ into $T^{(i)}$
4.  Processor j, $0 \le j < p$, 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 least-significant bit (LSB)

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

The 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

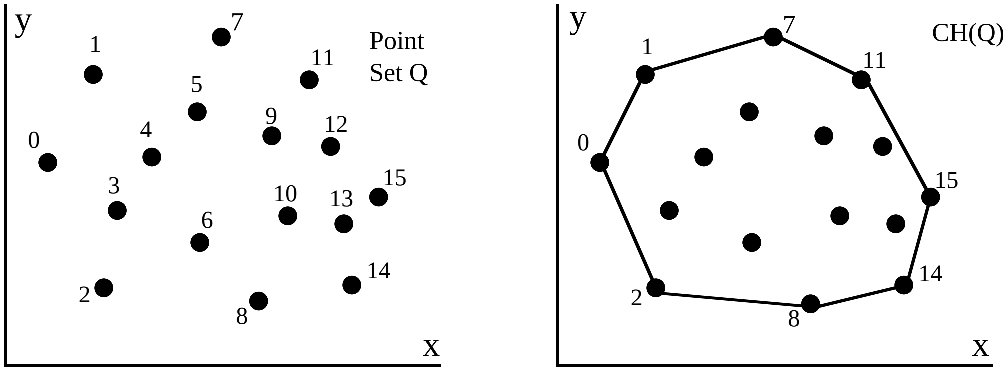


Fig. 6.2. Defining the convex hull problem.

Best sequential algorithm for p points: Ω(p log p) steps

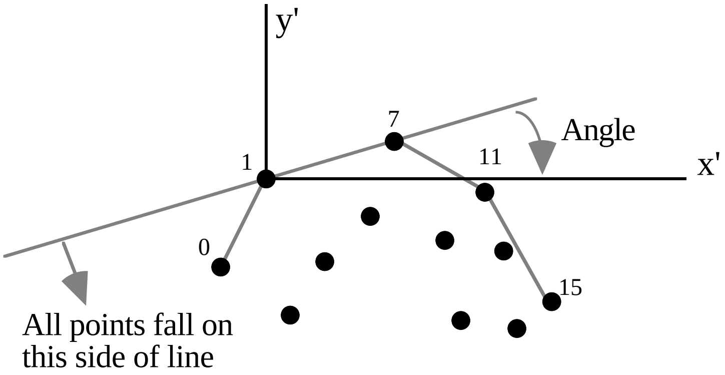


Fig. 6.3. Illustrating the properties of the convex hull.

<u>Parallel convex hull algorithm PRAMconvexhull(S, p)</u>

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

4. Merge the $\sqrt{p}$ convex hulls CH($Q^{(i)}$) into the overall hull CH(Q)

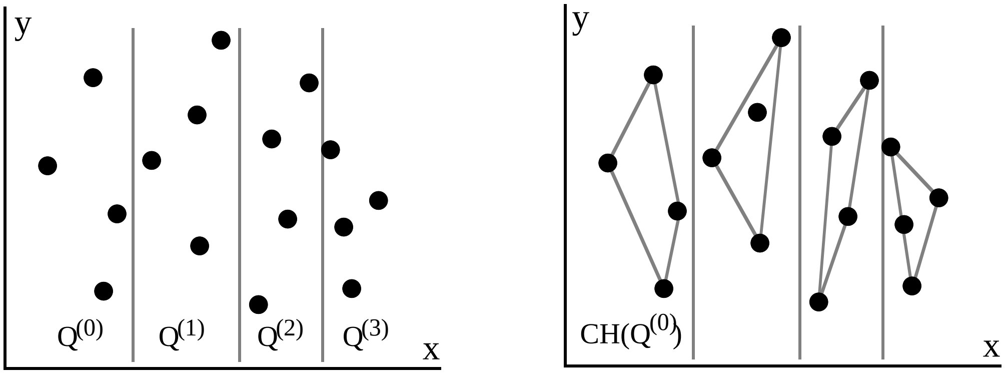


Fig. 6.4. Multiway divide and conquer for the convex hull problem.

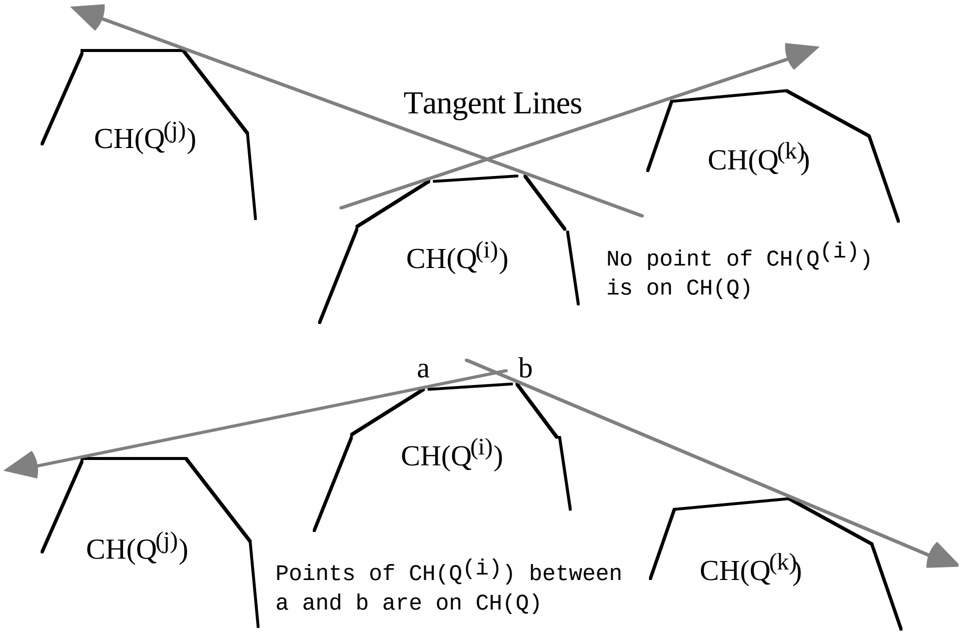

Tangent Lines
CH(Q(j))
CH(Q(k))
CH(Q(i))
No point of CH(Q(i)) is on CH(Q)
a
b
CH(Q(i))
CH(Q(j))
Points of CH(Q(i)) between a and b are on CH(Q)
CH(Q(k))

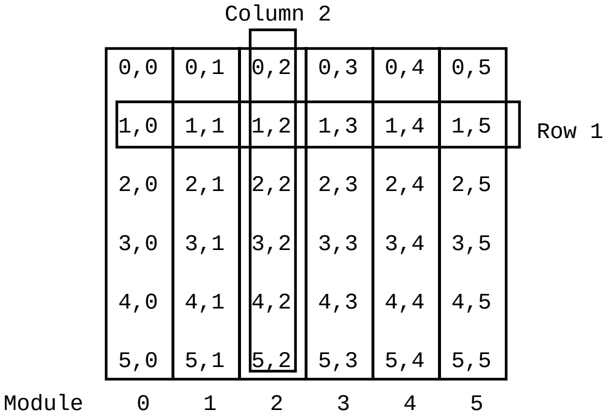
Fig. 6.5. Finding points in a partial hull that belong to the combined hull.

Analysis:

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

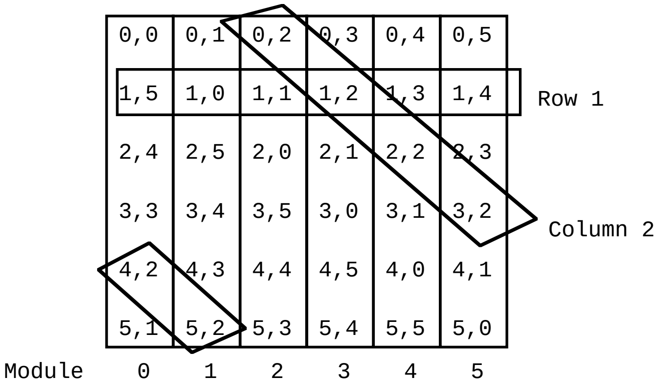
The intiail sorting time is also O(log p)

6.6 Some Implementation Aspects



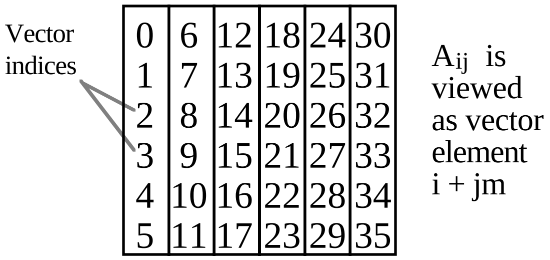
Module	0	1	2	3	4	5
	0,0	0,1	0,2	0,3	0,4	0,5
	1,0	1,1	1,2	1,3	1,4	1,5
	2,0	2,1	2,2	2,3	2,4	2,5
	3,0	3,1	3,2	3,3	3,4	3,5
	4,0	4,1	4,2	4,3	4,4	4,5
	5,0	5,1	5,2	5,3	5,4	5,5

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



Module	0	1	2	3	4	5
	0,0	0,1	0,2	0,3	0,4	0,5
	1,5	1,0	1,1	1,2	1,3	1,4
	2,4	2,5	2,0	2,1	2,2	2,3
	3,3	3,4	3,5	3,0	3,1	3,2
	4,2	4,3	4,4	4,5	4,0	4,1
	5,1	5,2	5,3	5,4	5,5	5,0

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



0	6	12	18	24	30
1	7	13	19	25	31
2	8	14	20	26	32
3	9	15	21	27	33
4	10	16	22	28	34
5	11	17	23	29	35

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 the bank a + kb mod 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, ··· , 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)

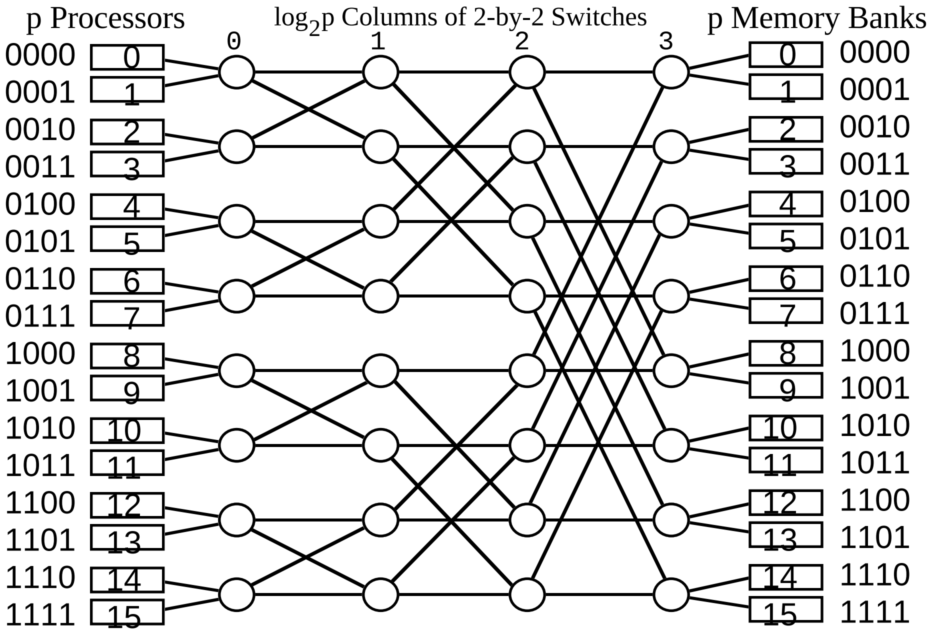


Fig. 6.9. Example of a multistage memory access network.

7 Sorting and Selection Networks

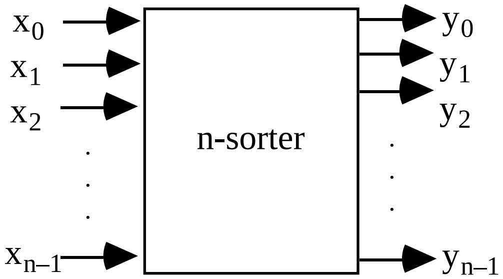
Chapter Goals

- Become familiar with the circuit-level models of parallel processing
- Introduce useful design tools and trade-off issues via a familiar problem

  (three more application-specific examples to come in Chapter 8)

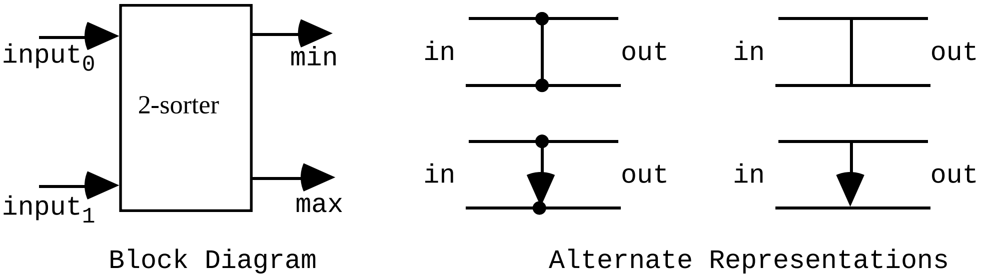
Chapter Contents

7.1 What Is a Sorting Network?



The outputs are a permutation of the inputs satisfying $y_0 \le y_1 \le \ldots \le y_{n-1}$ (non-descending)

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



Block Diagram

Alternate Representations

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

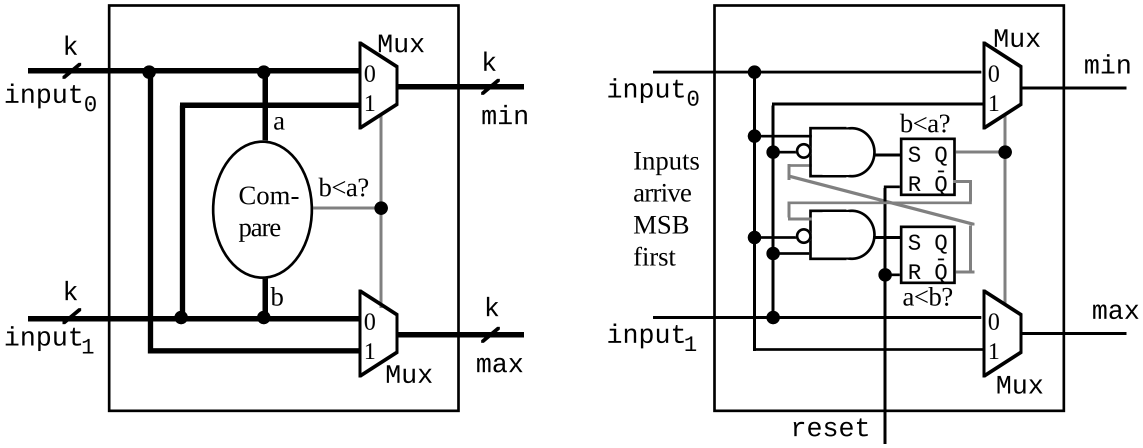


Fig. 7.3. Parallel and bit-serial hardware realizations of a 2-sorter.

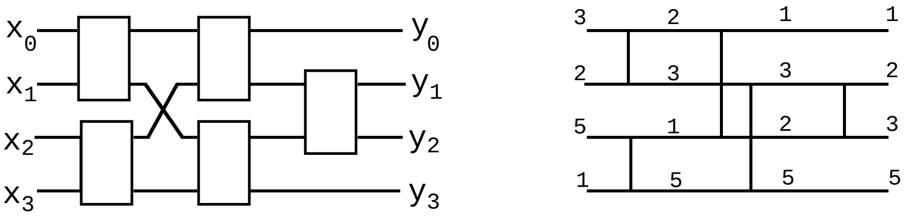


Fig. 7.4. Block diagram and schematic representation of a 4-sorter.

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

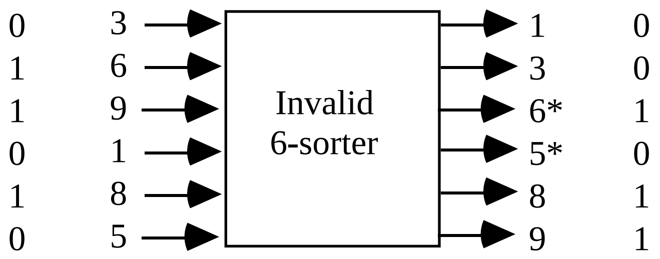
The answer is easy in this case

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.

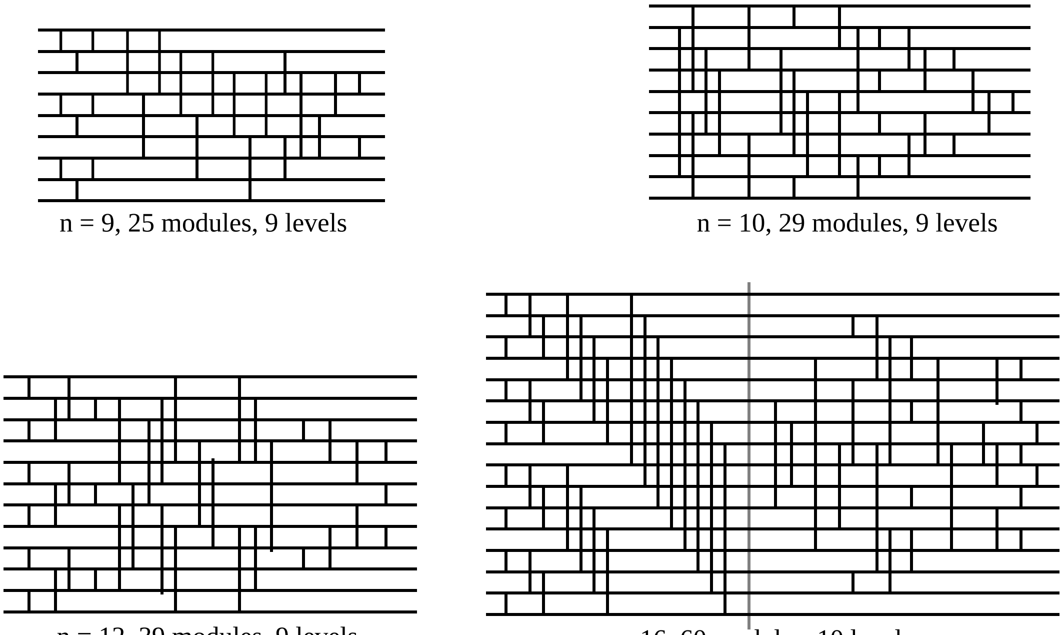
The zero-one principle: A comparison-based sorter is valid iff it correctly sorts all 0/1 sequences.



7.2 Figures of Merit for Sorting Networks

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

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



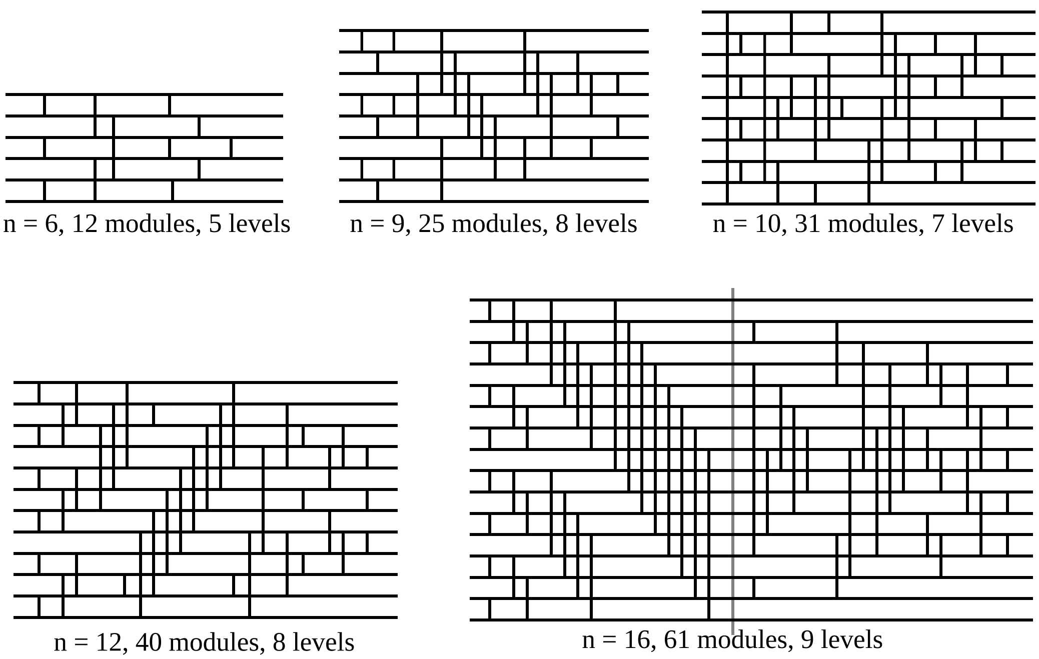
n = 9, 25 modules, 9 levels

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.



n = 6, 12 modules, 5 levels

n = 9, 25 modules, 8 levels

n = 10, 31 modules, 7 levels

n = 12, 40 modules, 8 levels

n = 16, 61 modules, 9 levels

Fig. 7.6. Some fast sorting networks.

7.3 Design of Sorting Networks

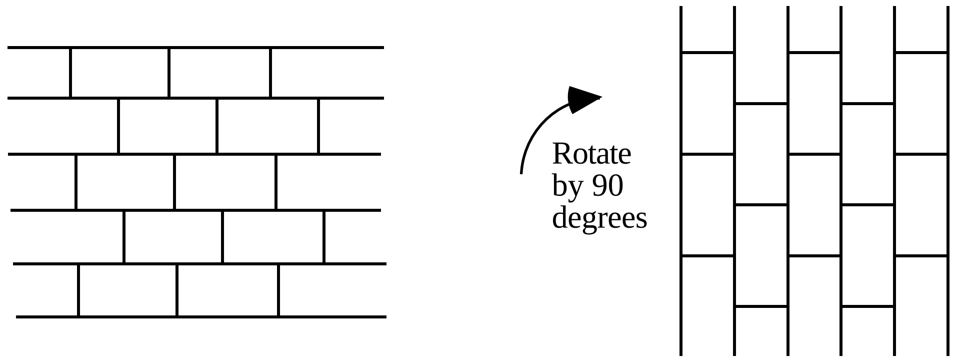
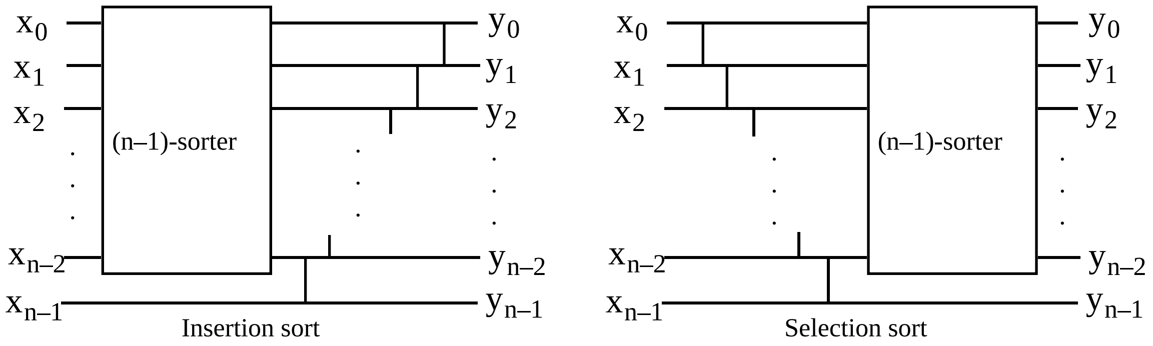


Fig. 7.7. Brick-wall 6-sorter based on odd–even transposition.

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

$$D(n) = D(n-1) + 2 = 2 + 2 + \cdots + 2 + 1 = 2(n-2) + 1 = 2n-3$$

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



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

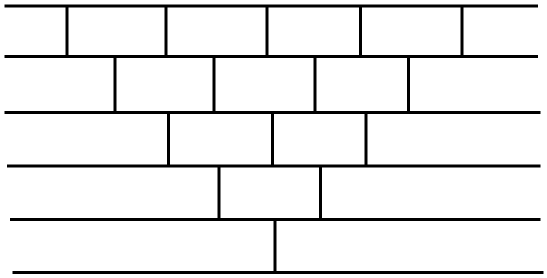


Fig. 7.8. Sorting network based on insertion sort or selection sort.

7.4 Batcher Sorting Networks

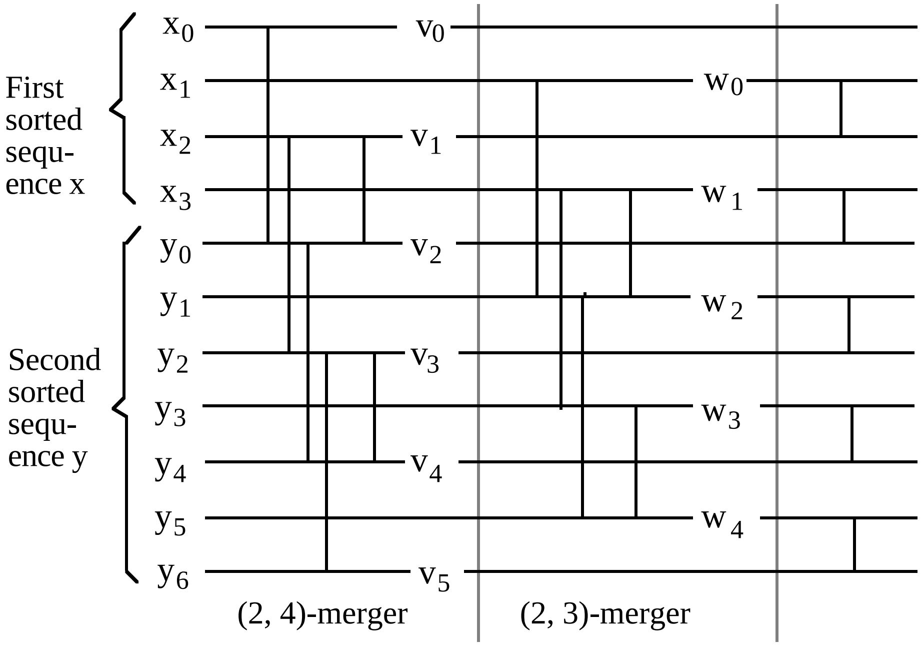


Fig. 7.9. Batcher's even–odd merging network for 4 + 7 inputs.

$x_0 \le x_1 \le \cdots \le x_{m-1}$ (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 $v_0\ w_0\ v_1\ w_1\ v_2\ w_2 \cdots$ is already sorted
Case b: $k_{even} = k_{odd}+1$ The sequence $v_0\ w_0\ v_1\ w_1\ v_2\ w_2 \cdots$ is already sorted
Case c: $k_{even} = k_{odd}+2$

v 0 0 0 0 0 0 0 $\underline{0}$ 1 1 1 1
w 0 0 0 0 0 0 $\underline{1}$ 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:

$C(m) = 2C(m/2) + m - 1 = (m - 1) + 2(m/2 - 1) + 4(m/4 - 1) + \cdots = m \log_2 m + 1$

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

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

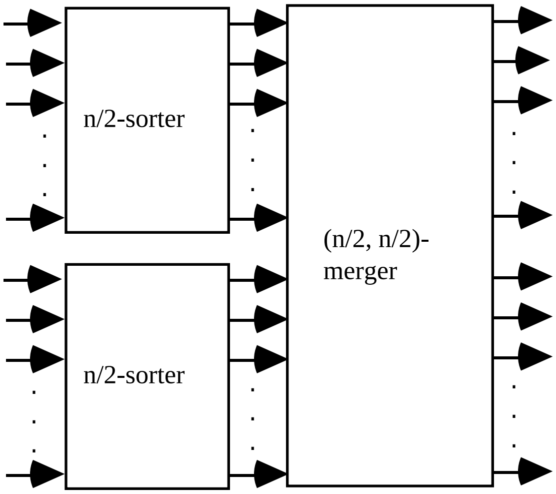


Fig. 7.10. The recursive structure of Batcher's even–odd merge sorting network.

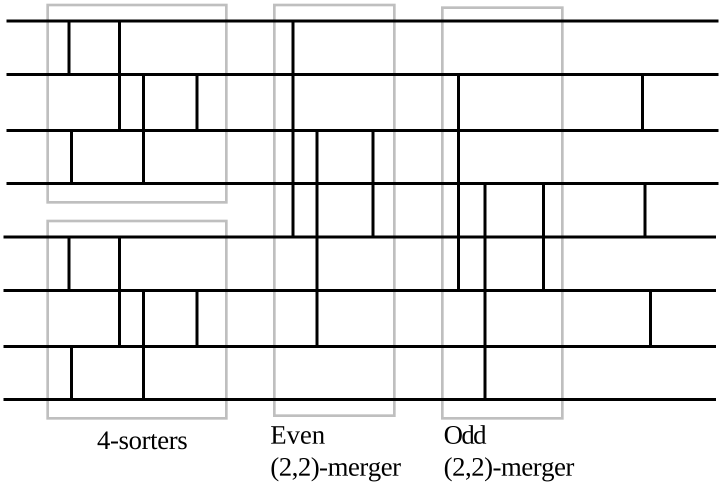


Fig. 7.11. Batcher's even-odd merge sorting network for eight inputs.

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

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

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

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

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

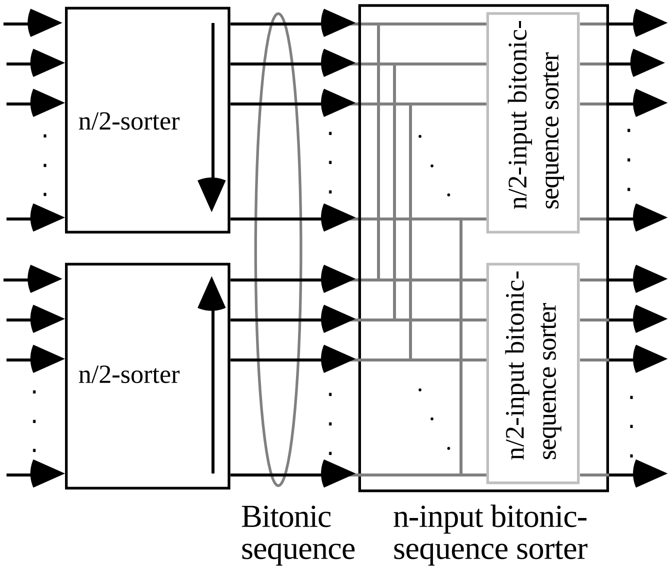
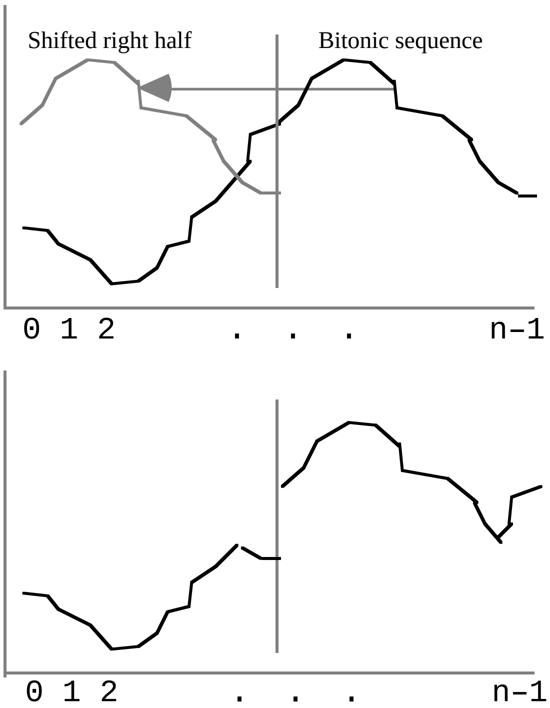

n/2-sorter
n/2-sorter
n/2-input bitonic-sequence sorter
n/2-input bitonic-sequence sorter
Bitonic sequence
n-input bitonic-sequence sorter

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



Shift right half of data to left half

Keep smaller value of each pair and ship the larger value to right

Each half is a bitonic sequence that can be sorted independently

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

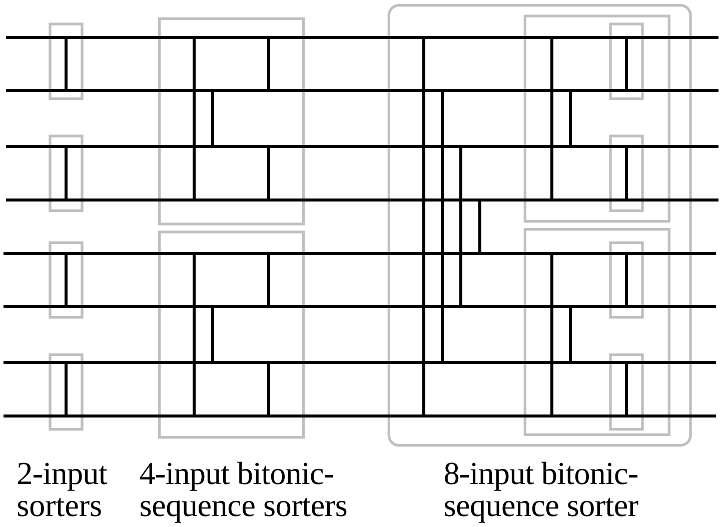

2-input sorters
4-input bitonic-sequence sorters
8-input bitonic-sequence sorter

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

7.5 Other Classes of Sorting Networks

Periodic balanced sorting networks

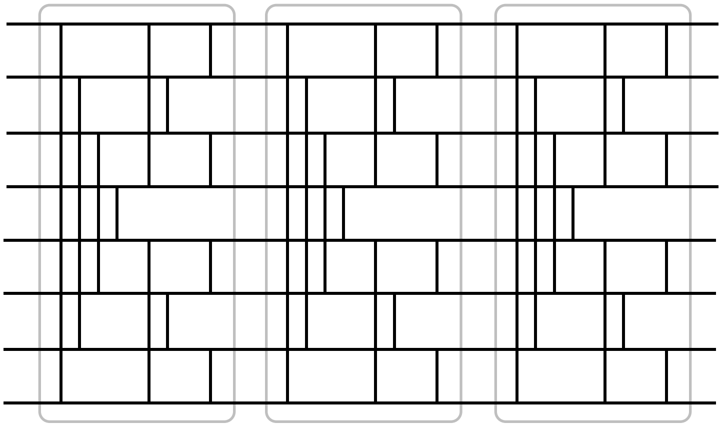


Fig. 7.14. Periodic balanced sorting network for eight inputs.

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

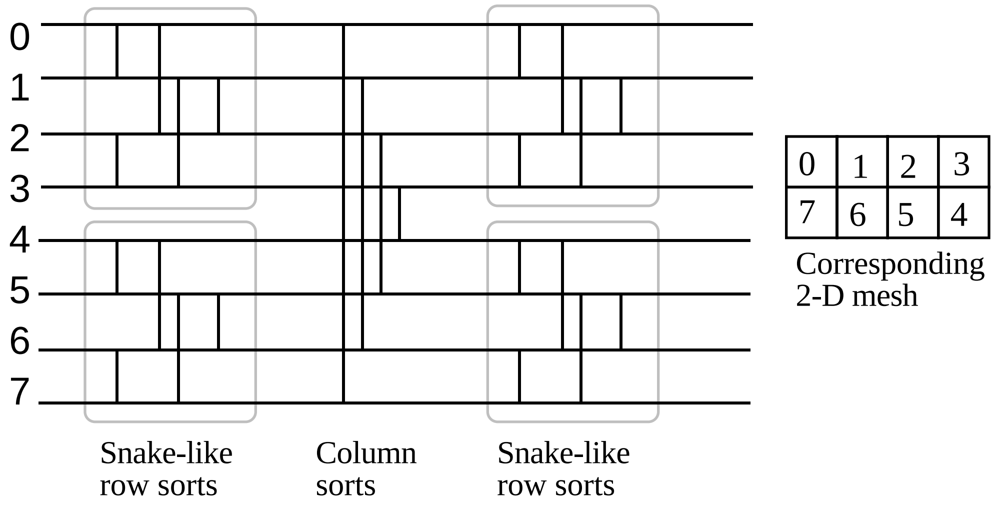


Fig. 7.15. Design of an 8-sorter based on shearsort on 2×4 mesh.

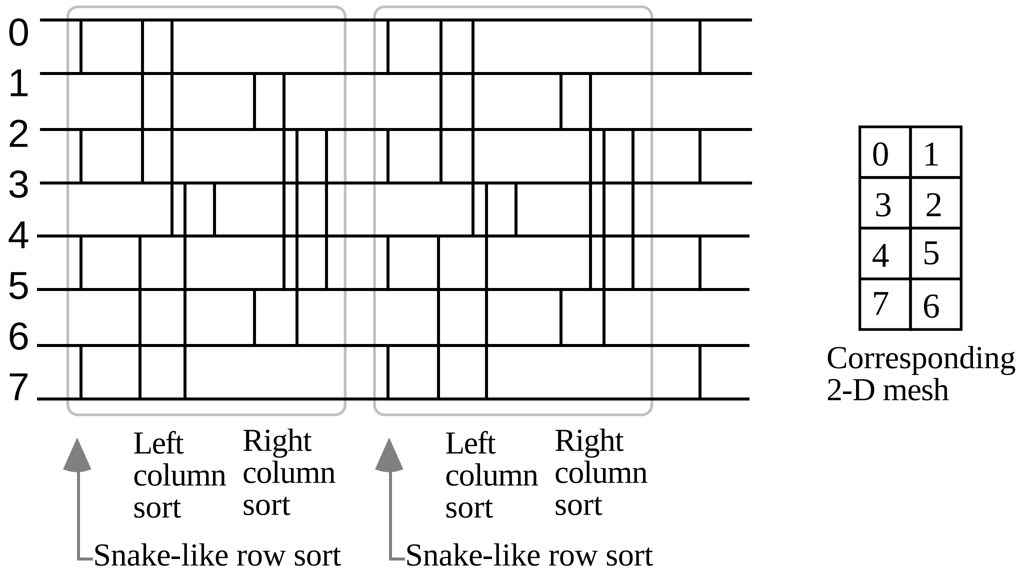


Fig. 7.16. Design of an 8-sorter based on shearsort on 4×2 mesh.

7.6 Selection Networks

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

One can define three selection problems:

I. Select the k smallest values; present in sorted order

II. Select kth smallest value

III. Select the k smallest values; present in any order

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

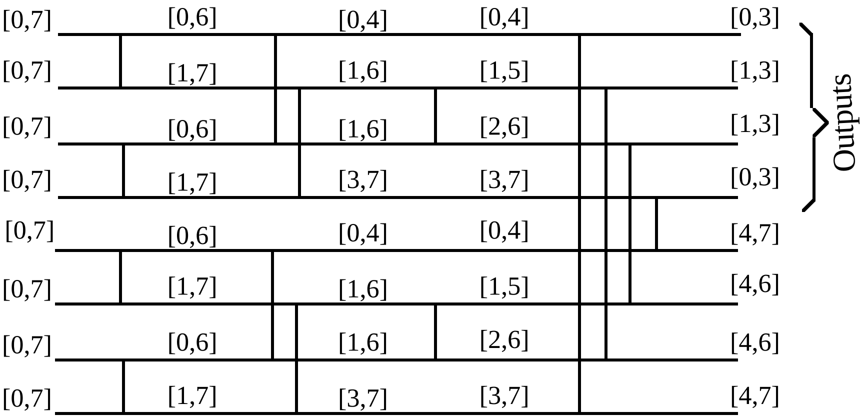


Fig. 7.17. A type III (8, 4)-selector.

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 in fact an 8-input classifier

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

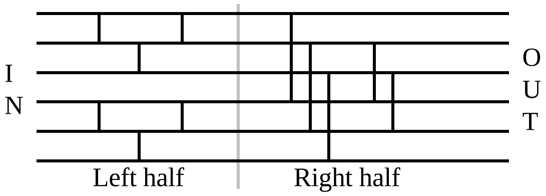


Figure for Problem 7.7.

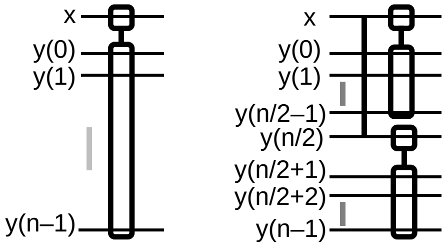


Figure for Problem 7.9.

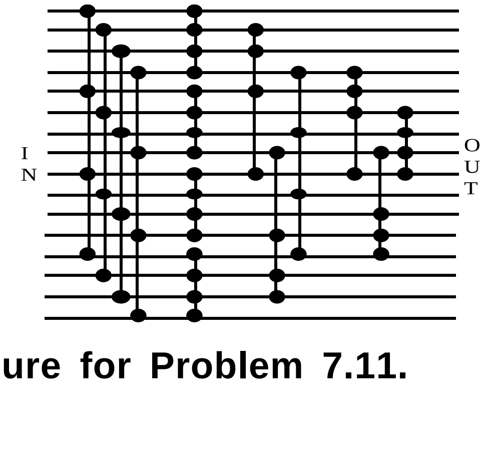


Figure for Problem 7.11.

8 Other Circuit-Level Examples

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

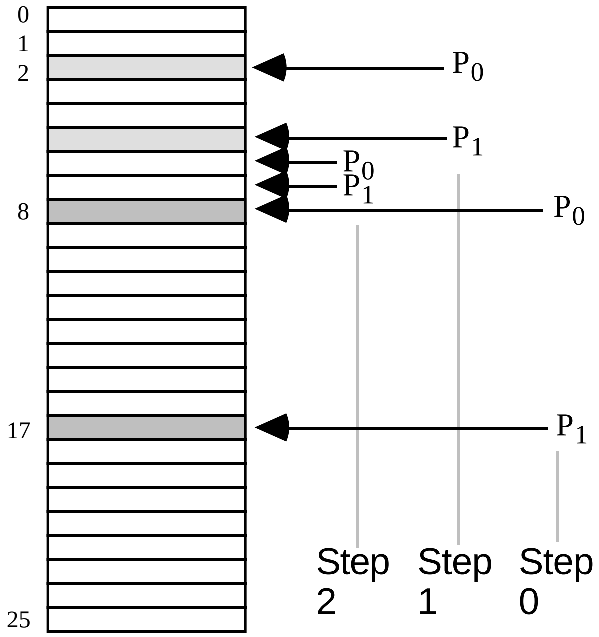
Parallel (p + 1)-ary search:

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

Example:

n = 26

p = 2



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

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

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

Dynamic data sets

Batch searching

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 smallest key; return data?

extractmax Remove record(s) with largest key; return data?

extractmed Remove the record(s) with median key value; return data?

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

8.2 A Tree-Structured Dictionary Machine

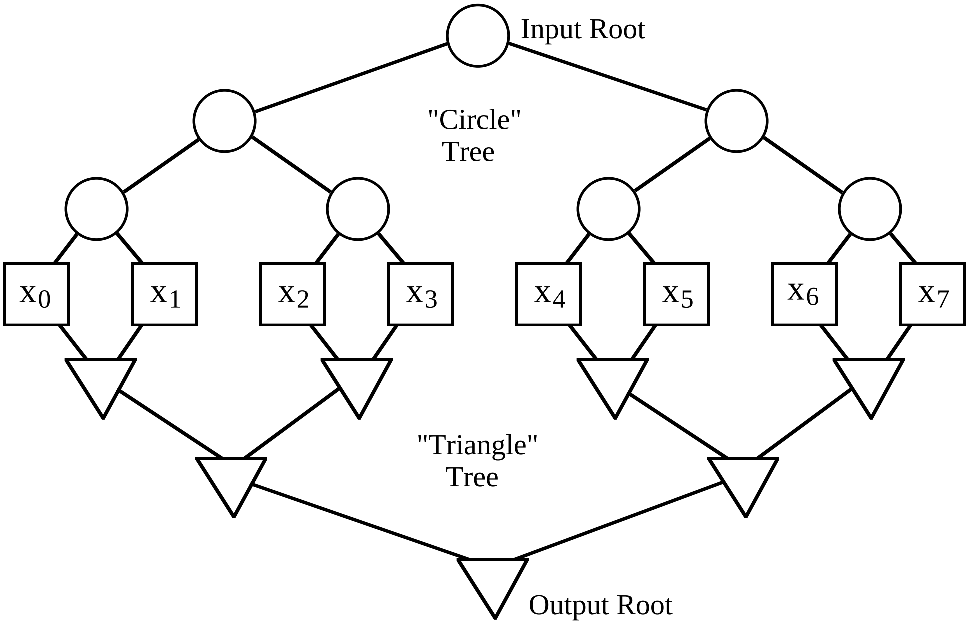


Fig. 8.1. A tree-structured dictionary machine.

The 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 indicate "yes"

findmin Pass the 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)

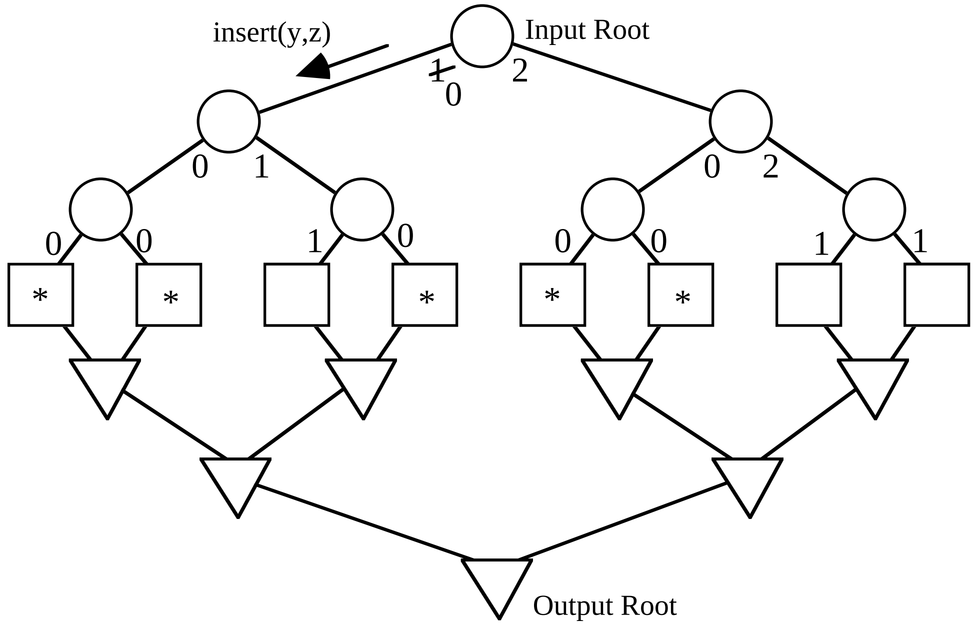


Fig. 8.2. Tree machine storing five records and containing three free slots.

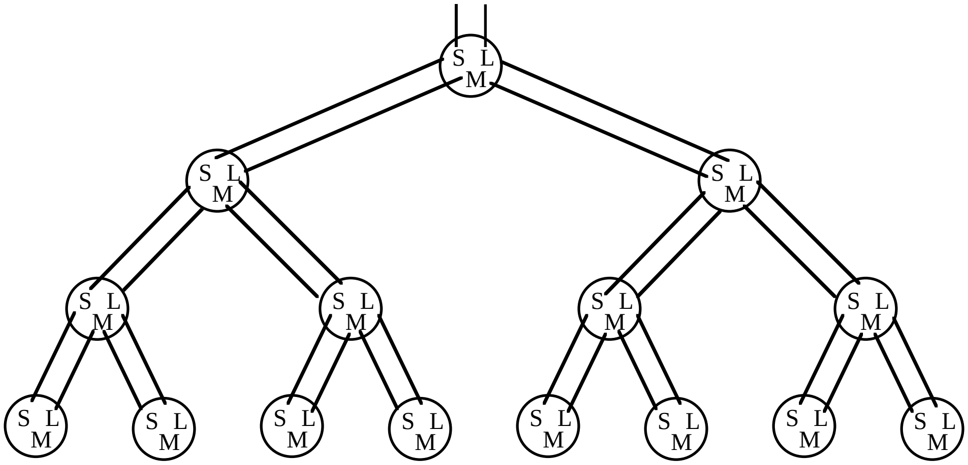
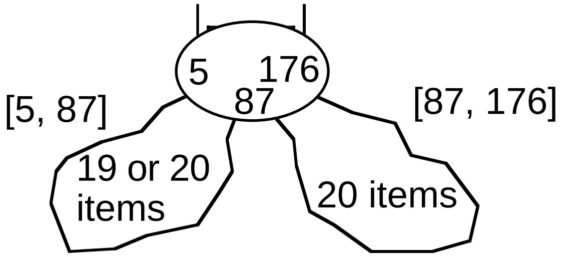


Fig. 8.3. Systolic data structure for minimum, maximum, and median finding.



Insert 2
Insert 20
Insert 127
Insert 195

Extractmin
Extractmed
Extractmax

8.3 Parallel Prefix Computation

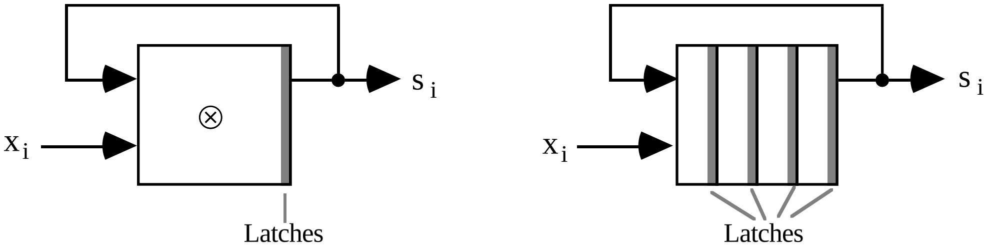


Fig. 8.4. Prefix computation using a latched or pipelined function unit.

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

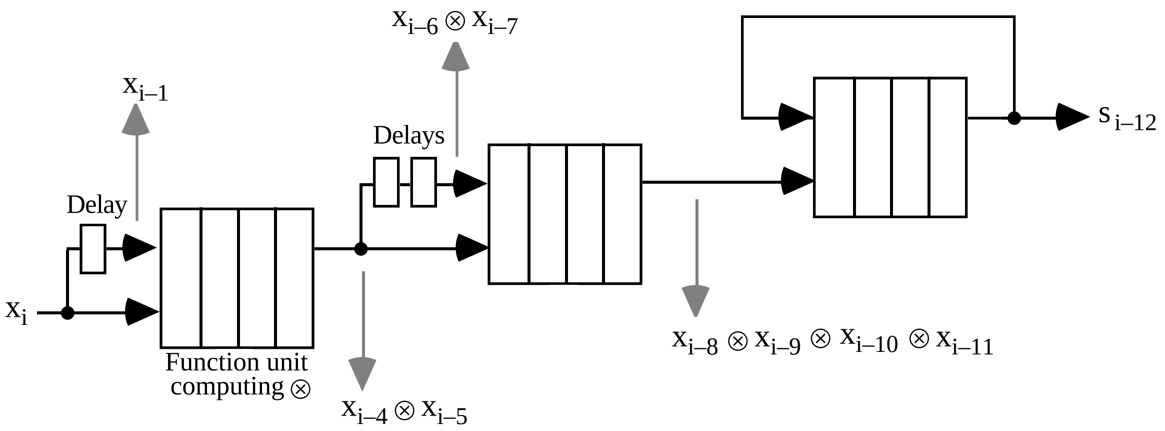

$x_{i-6} \otimes x_{i-7}$
x_{i-1}
Delays
s_{i-12}
Delay
x_i
Function unit computing ⊗
$x_{i-8} \otimes x_{i-9} \otimes x_{i-10} \otimes x_{i-11}$
$x_{i-4} \otimes x_{i-5}$

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

8.4 Parallel Prefix Networks

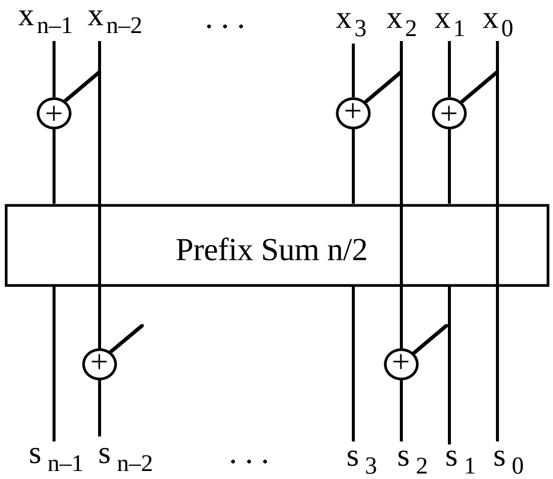


Fig. 8.6. Prefix sum network built of one n/2-input networks and n – 1 adders.

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

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

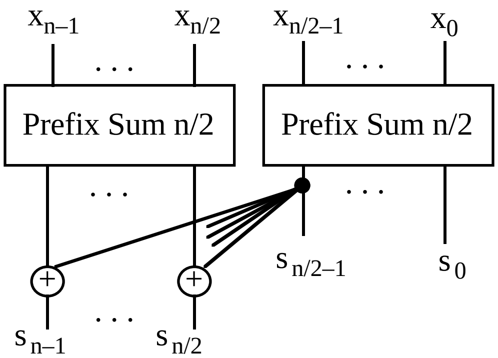


Fig. 8.7. Prefix sum network built of two n/2-input networks and n/2 adders.

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

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

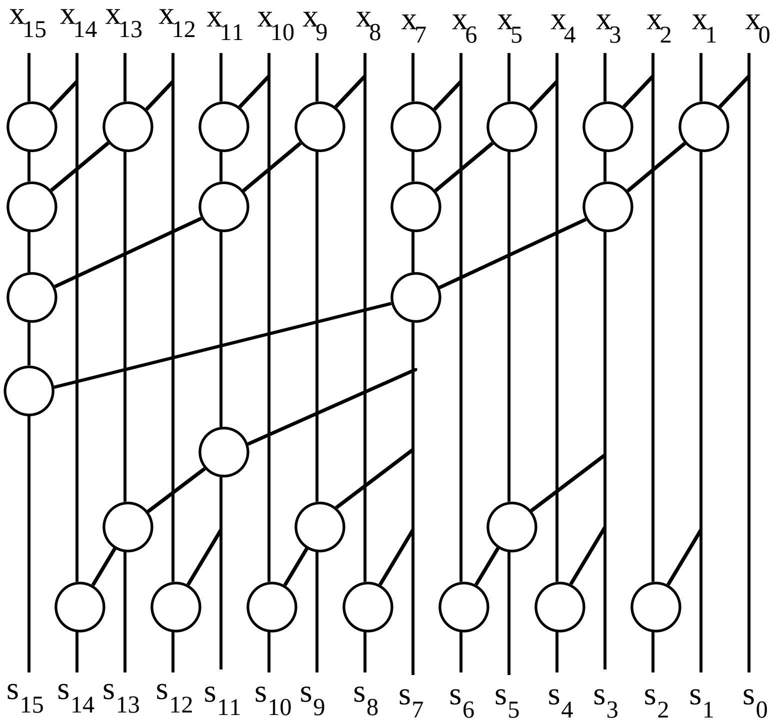

x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

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

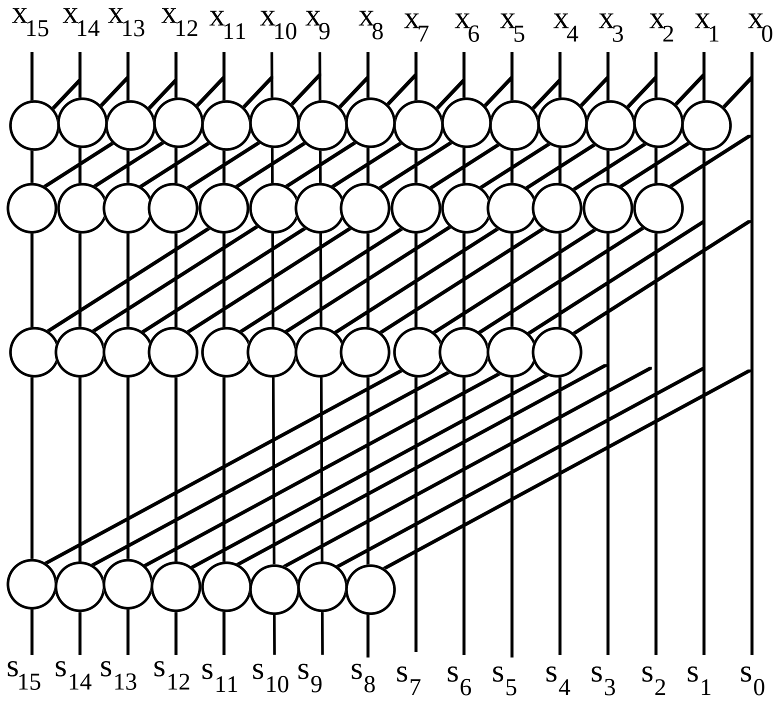

x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

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

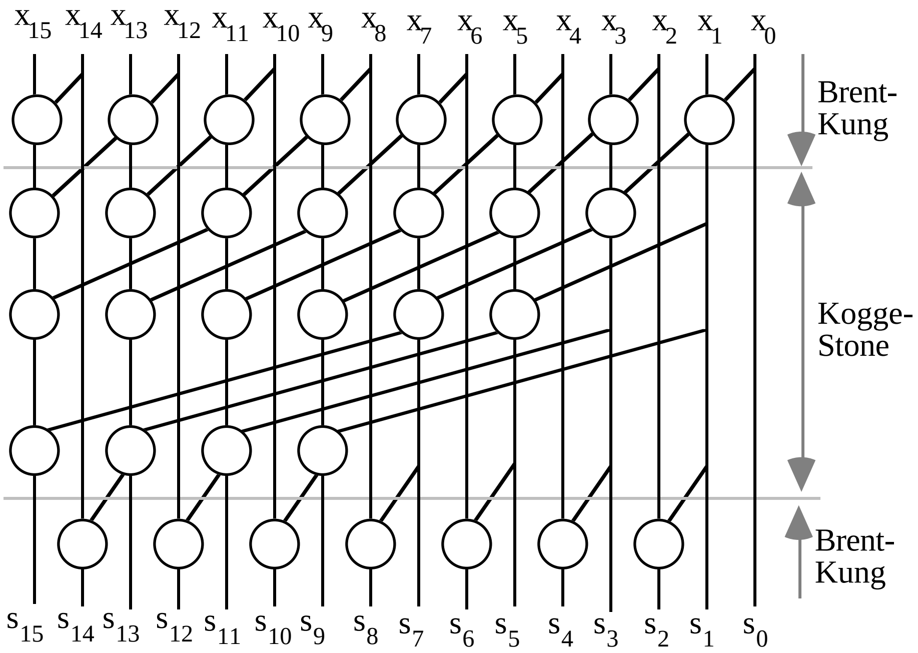

x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
Brent-Kung
Kogge-Stone
Brent-Kung
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

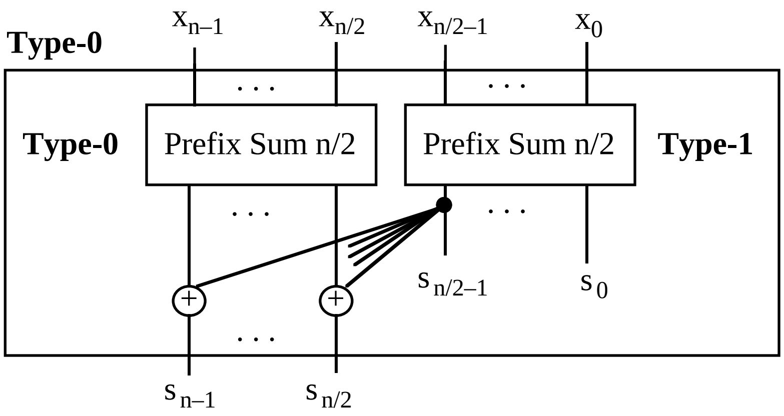
Fig. 8.10. A hybrid Brent–Kung/Kogge–Stone parallel prefix graph for 16 inputs.

Type-x parallel prefix network

Produces the leftmost output in $\log_2$(# of inputs) time

Yields all other outputs with at most x additional delay

Building the fastest possible parallel prefix network (type-0)


Type-0
x_{n-1}
$x_{n/2}$
$x_{n/2-1}$
x_0
Type-0
Prefix Sum n/2
Prefix Sum n/2
Type-1
$s_{n/2-1}$
s_0
+
+
s_{n-1}
$s_{n/2}$

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 $y = F_n x$

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

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

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:

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

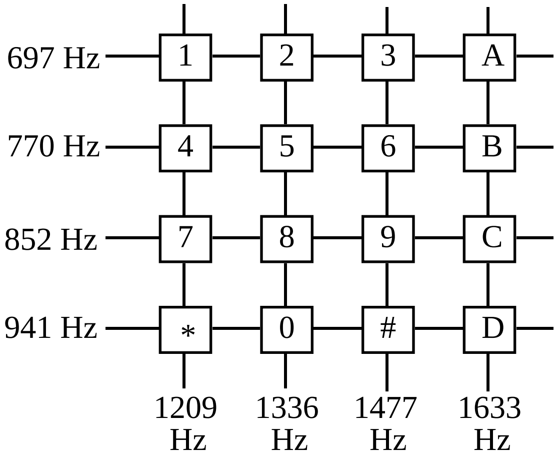
Any matrix-vector multiplication algorithm can be used to compute DFTs

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

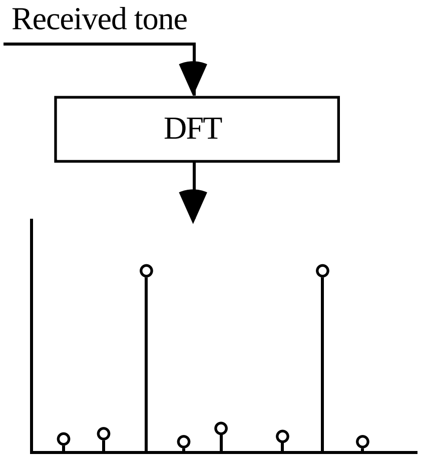
The resulting algorithm is known as the fast Fourier transform (FFT)

DFT Applications

Spectral analysis

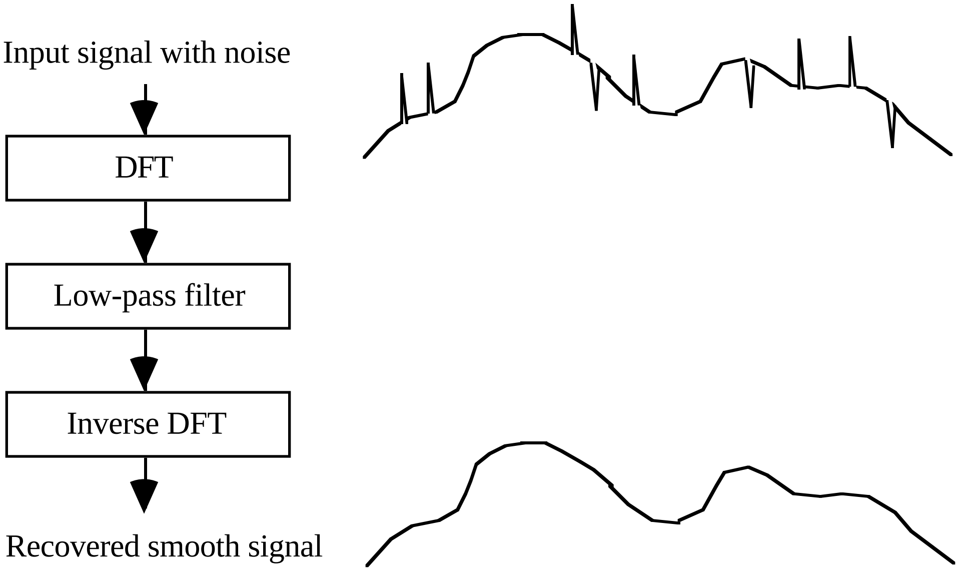


Tone frequency assignments for touch-tone dialing



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

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

Then:

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

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 two 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

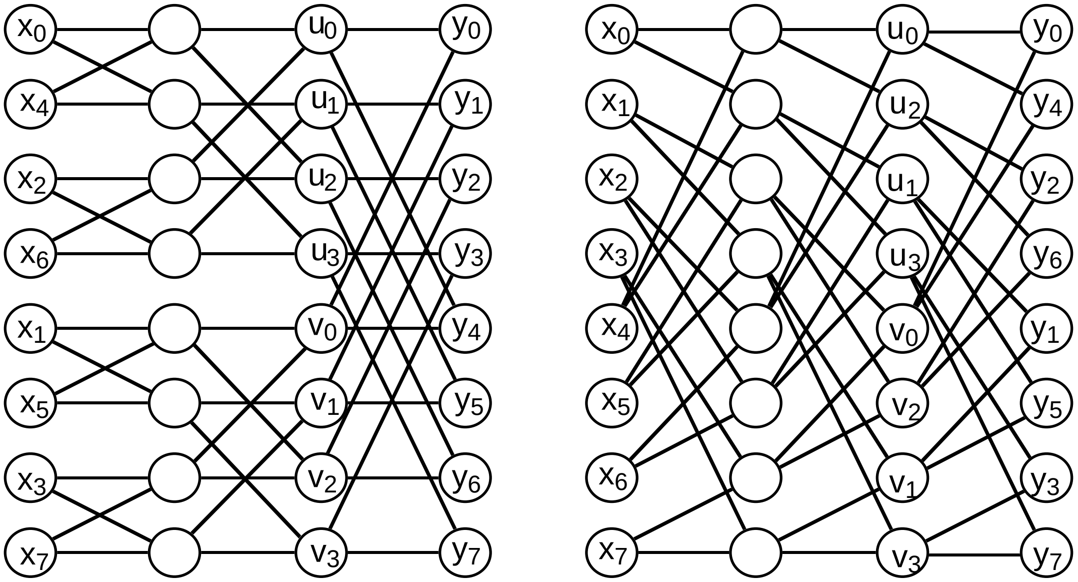


Fig. 8.11. Butterfly network for an 8-point FFT.

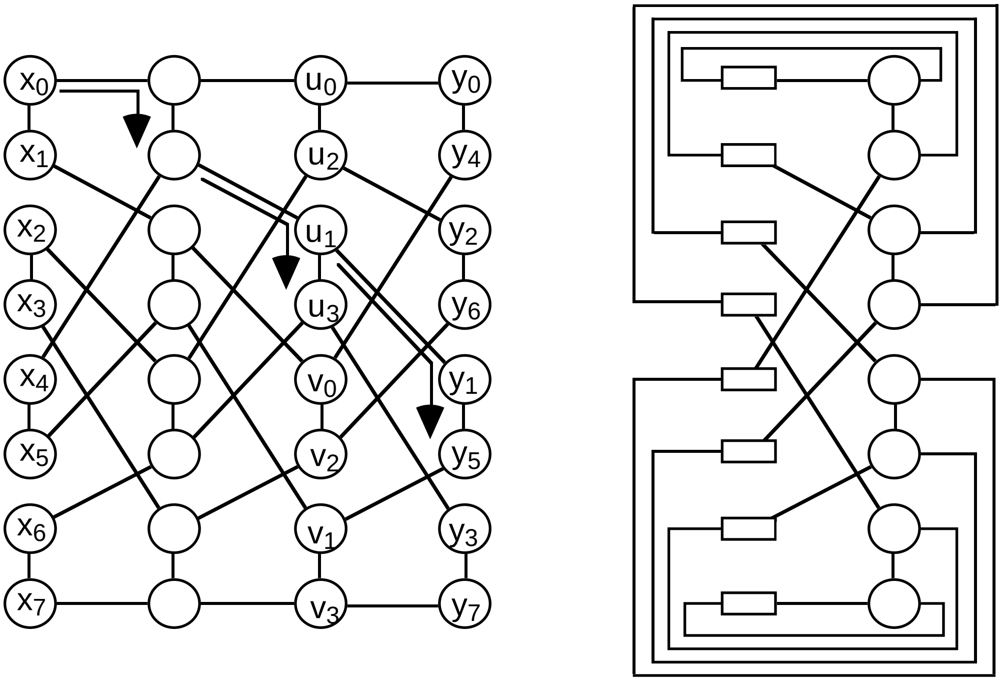


Fig. 8.12. FFT network variant and its shared-hardware realization.

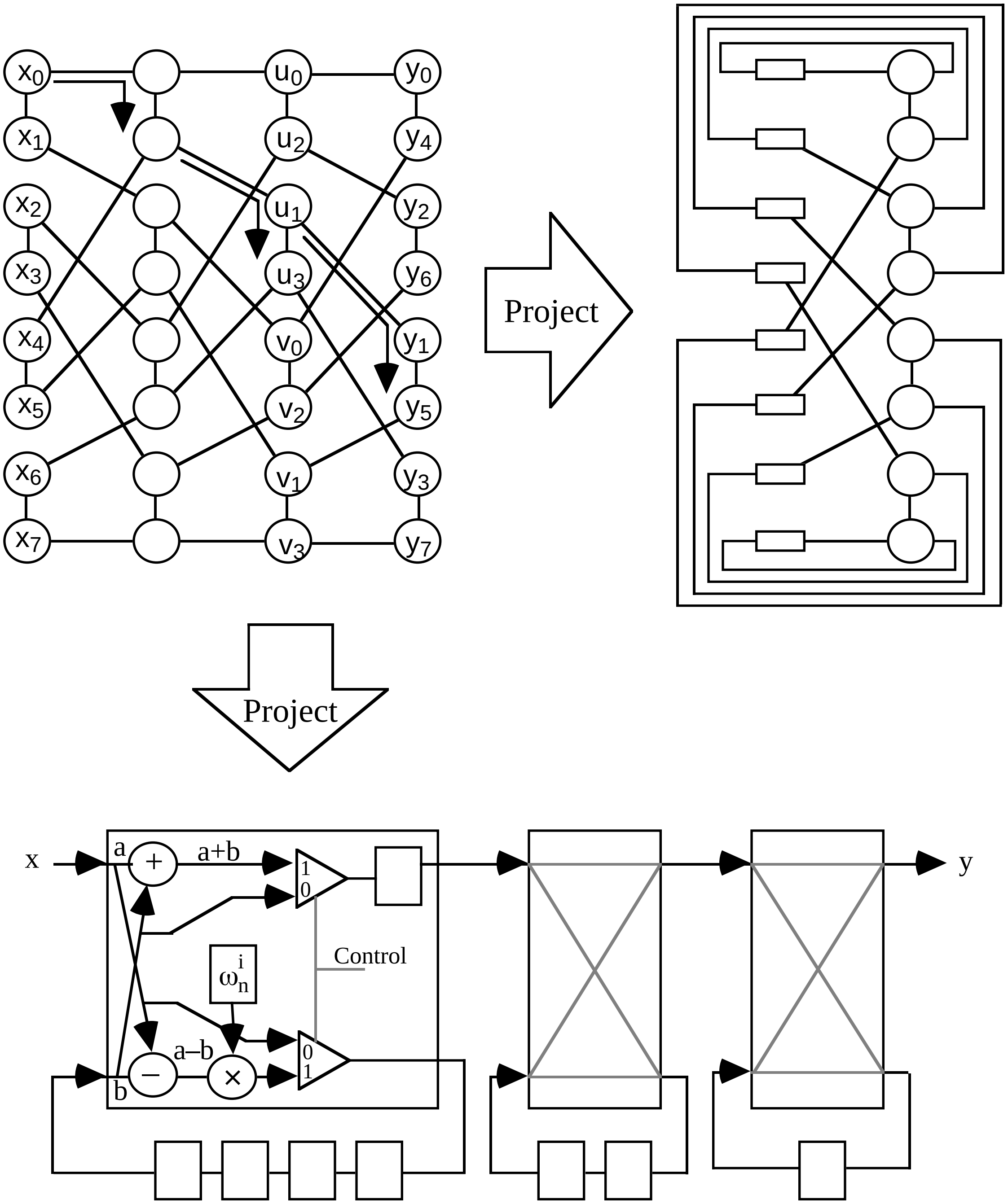


Fig. 8.13. **Linear array of $\log_2 n$ cells for n-point FFT computation.**

Part III Mesh-Based Architectures

Part Goals

- Study 2D mesh and torus networks in depth
  - of great practical significance
  - used in recent parallel machines
  - regular with short wires (scalable)
- Discuss 3D and higher-dimensional meshes/tori as well as 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

Chapter Goals

- Introduce the mesh model (processors, links, communication)
- Develop 2D mesh sorting algorithms
- Learn about mesh capabilities/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

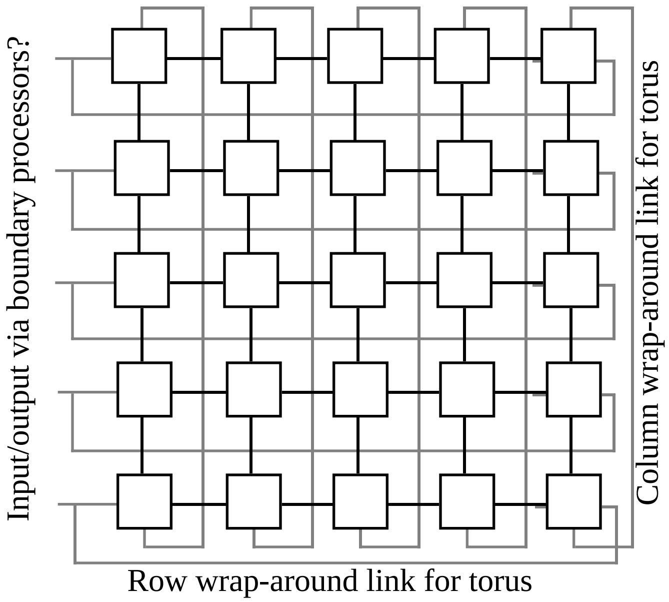

Input/output via boundary processors?
Column wrap-around link for torus
Row wrap-around link for torus

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

We focus on 2D mesh (>2D discussed 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

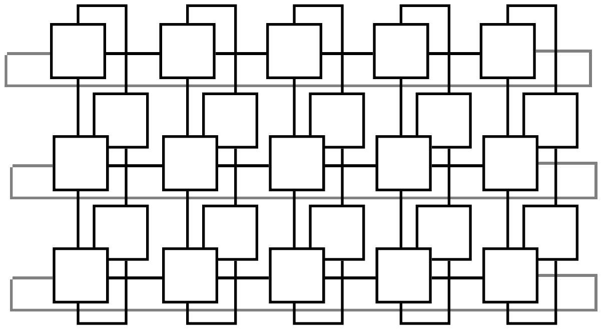
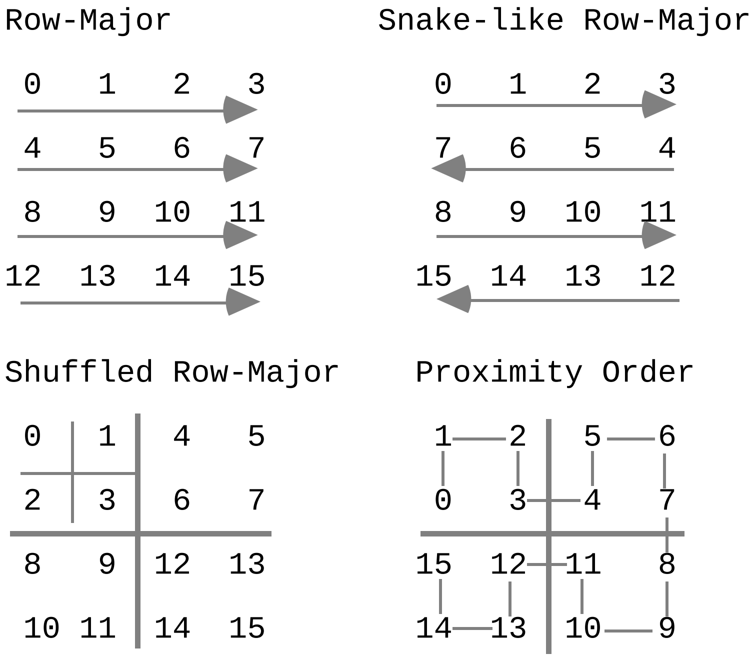


Fig. 9.2. **A 5 × 5 torus folded along its columns. Folding this diagram along the rows will produce a layout with only short links.**



Row-Major

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Snake-like Row-Major

0	1	2	3
7	6	5	4
8	9	10	11
15	14	13	12

Shuffled Row-Major

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Proximity Order

1	2	5	6
0	3	4	7
15	12	11	8
14	13	10	9

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

Interprocessor communication

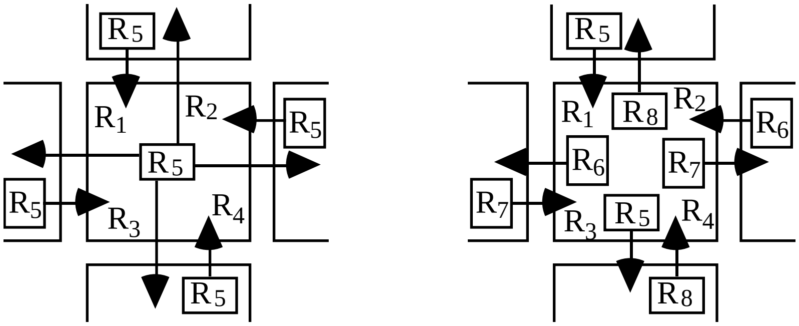


Fig. 9.4. Reading data from NEWS neighbors via virtual local registers.

9.2 The Shearsort Algorithm

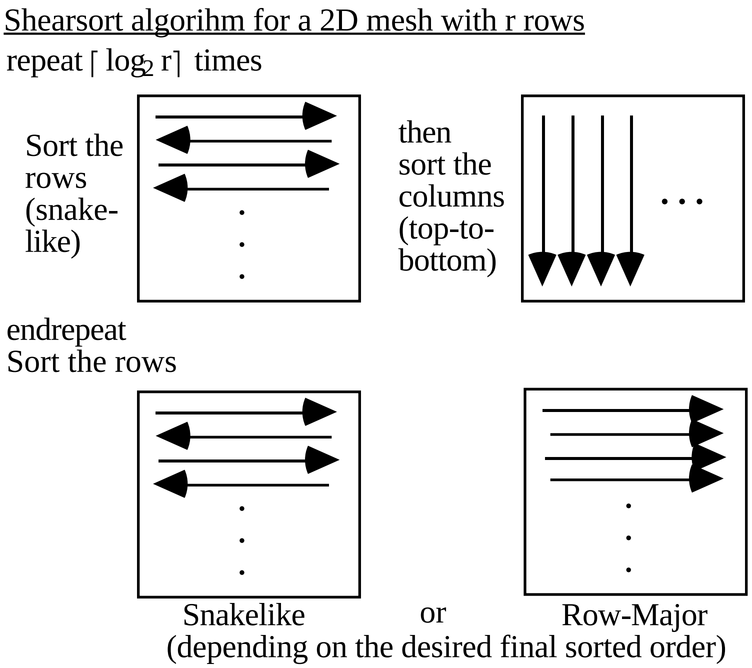

Shearsort algorihm for a 2D mesh with r rows
repeat ⌈log2 r⌉ times
Sort the rows (snake-like)
then sort the columns (top-to-bottom)
. . .
endrepeat
Sort the rows
Snakelike
or
Row-Major
(depending on the desired final sorted order)

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

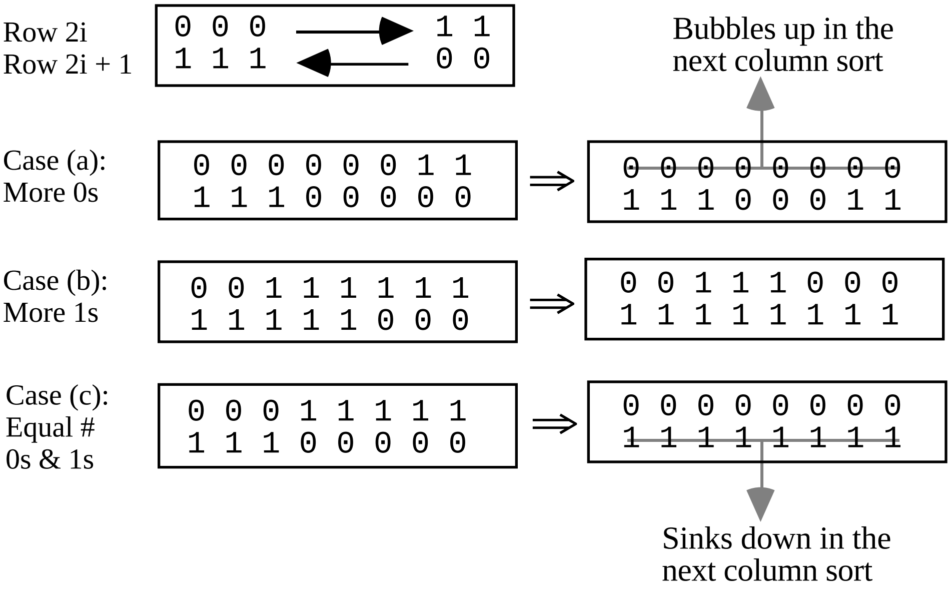


Fig. 9.6. A pair of dirty rows create at least one clean row in each shearsort iteration.

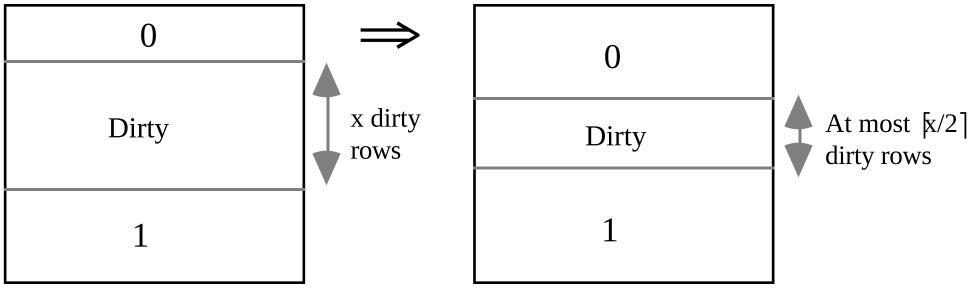
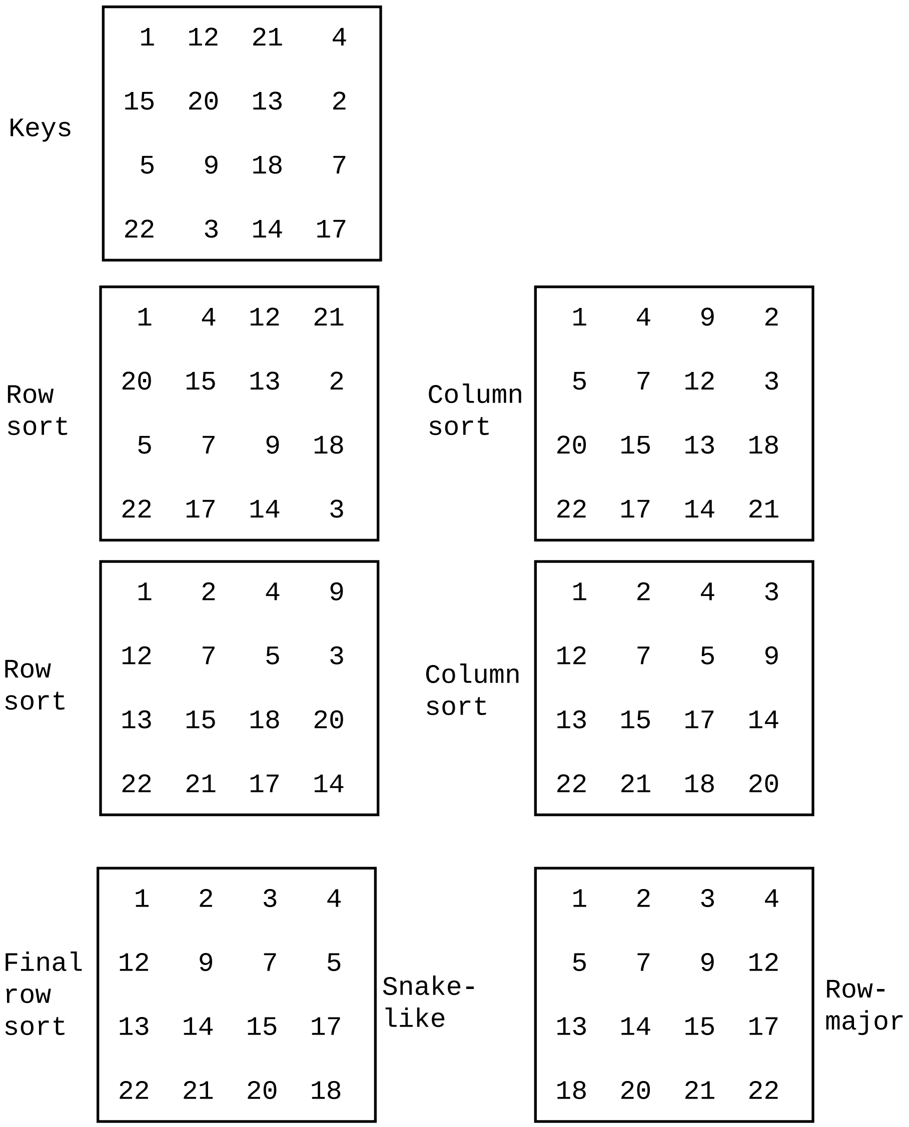


Fig. 9.7. The number of dirty rows halves with each shearsort iteration.



Keys

1	12	21	4
15	20	13	2
5	9	18	7
22	3	14	17

Row sort

1	4	12	21
20	15	13	2
5	7	9	18
22	17	14	3

Column sort

1	4	9	2
5	7	12	3
20	15	13	18
22	17	14	21

Row sort

1	2	4	9
12	7	5	3
13	15	18	20
22	21	17	14

Column sort

1	2	4	3
12	7	5	9
13	15	17	14
22	21	18	20

Final row sort

1	2	3	4
12	9	7	5
13	14	15	17
22	21	20	18

Snake-like

1	2	3	4
5	7	9	12
13	14	15	17
18	20	21	22

Row-major

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

9.3 Variants of Simple Shearsort

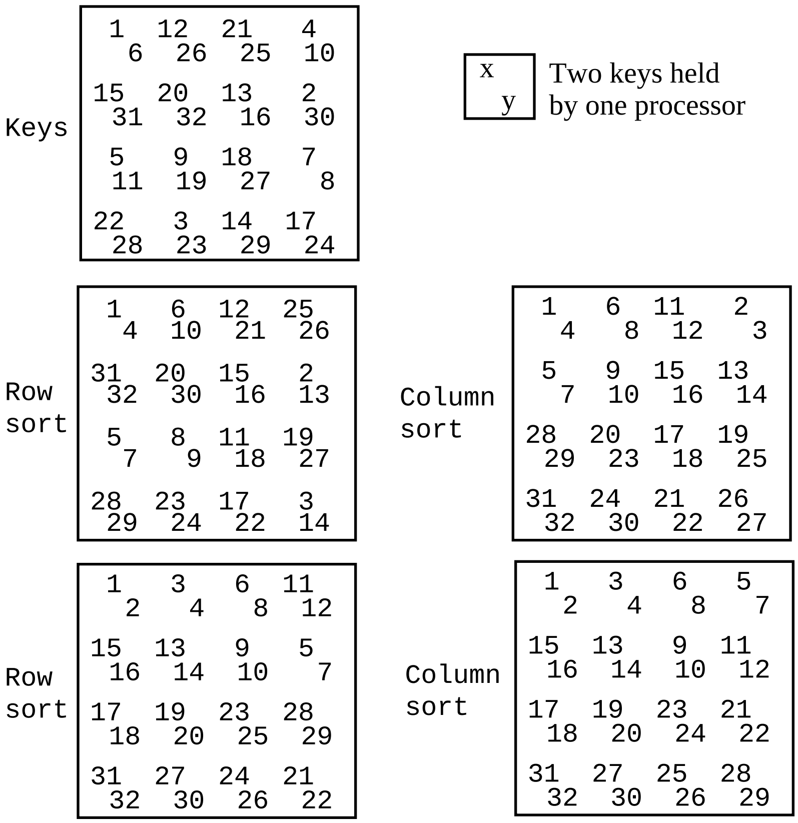
Sorting 0s and 1s on a linear array: odd-even transposition steps can be limited to the number of dirty elements

Example: sorting 000001011111 requires at most two steps

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

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

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



Keys

1 6	12 26	21 25	4 10
15 31	20 32	13 16	2 30
5 11	9 19	18 27	7 8
22 28	3 23	14 29	17 24

Row sort

1 4	6 10	12 21	25 26
31 32	20 30	15 16	2 13
5 7	8 9	11 18	19 27
28 29	23 24	17 22	3 14

Column sort

1 4	6 8	11 12	2 3
5 7	9 10	15 16	13 14
28 29	20 23	17 18	19 25
31 32	24 30	21 22	26 27

Row sort

1 2	3 4	6 8	11 12
15 16	13 14	9 10	5 7
17 18	19 20	23 25	28 29
31 32	27 30	24 26	21 22

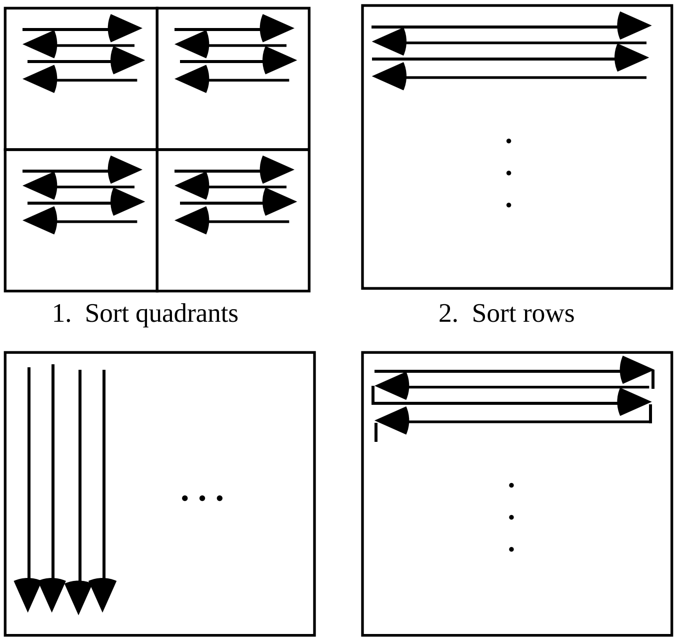
Column sort

1 2	3 4	6 8	5 7
15 16	13 14	9 10	11 12
17 18	19 20	23 24	21 22
31 32	27 30	25 26	28 29

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



1. Sort quadrants

2. Sort rows

3. Sort columns

4. Apply $4\sqrt{p}$ steps of odd-even transposition along the snake

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

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

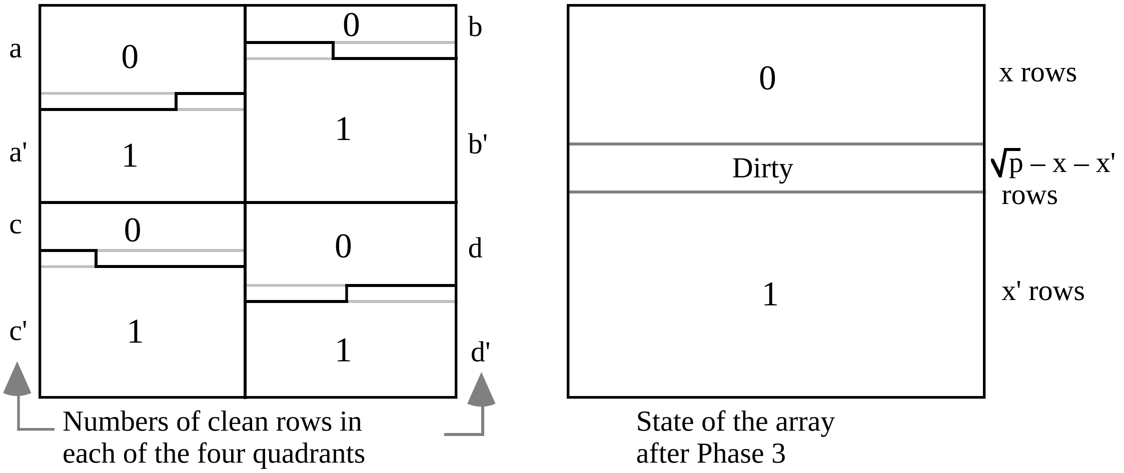

a
0
a'
1
c
0
c'
1
0
b
1
b'
0
d
1
d'
Numbers of clean rows in each of the four quadrants
0
x rows
Dirty
√p – x – x' rows
1
x' rows
State of the array after Phase 3

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

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

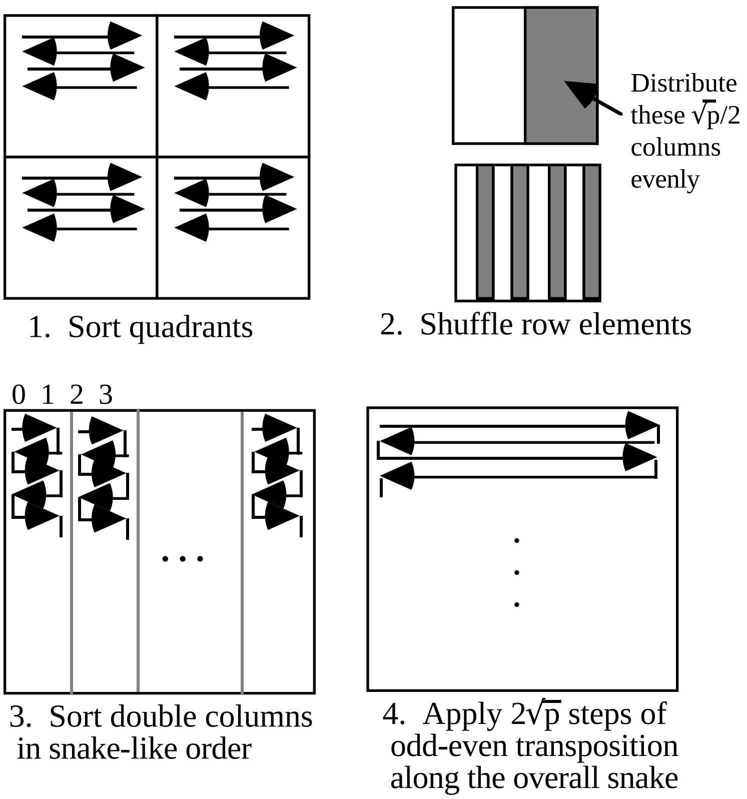
A similar inequality for x' leads to:

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

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

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

Another recursive sorting algorithm



1. Sort quadrants

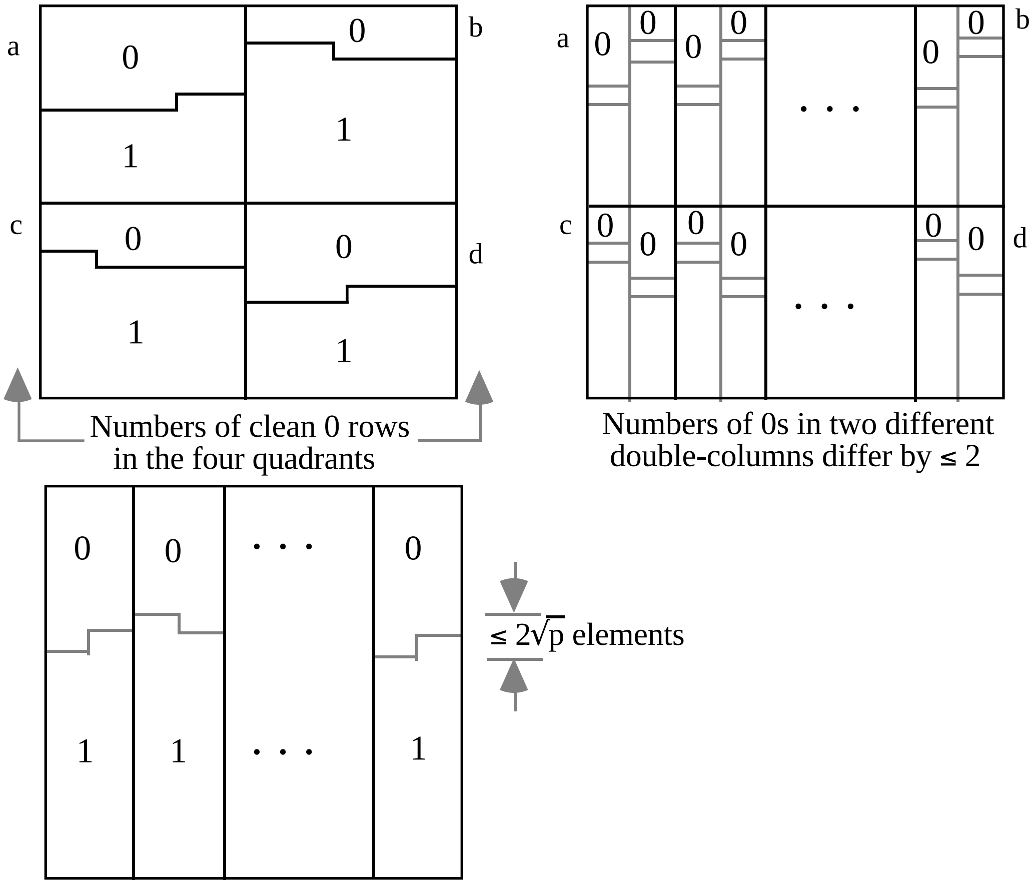
2. Shuffle row elements

3. Sort double columns in snake-like order

4. Apply 2√p steps of odd-even transposition along the overall snake

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

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



Numbers of clean 0 rows in the four quadrants

Numbers of 0s in two different double-columns differ by ≤ 2

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. Can we 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. Can we design an algorithm with better time than $9\sqrt{p}$?

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

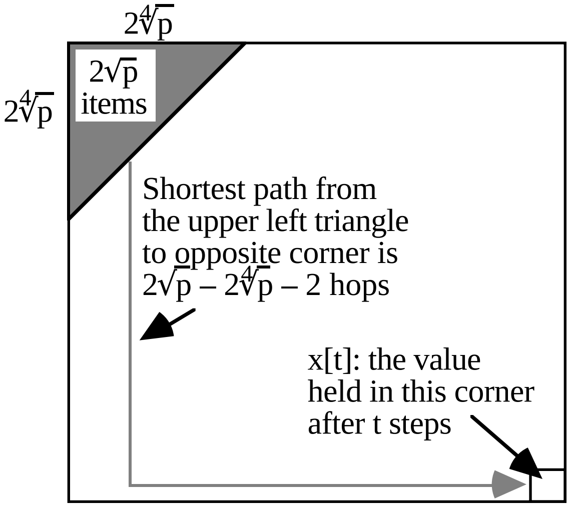
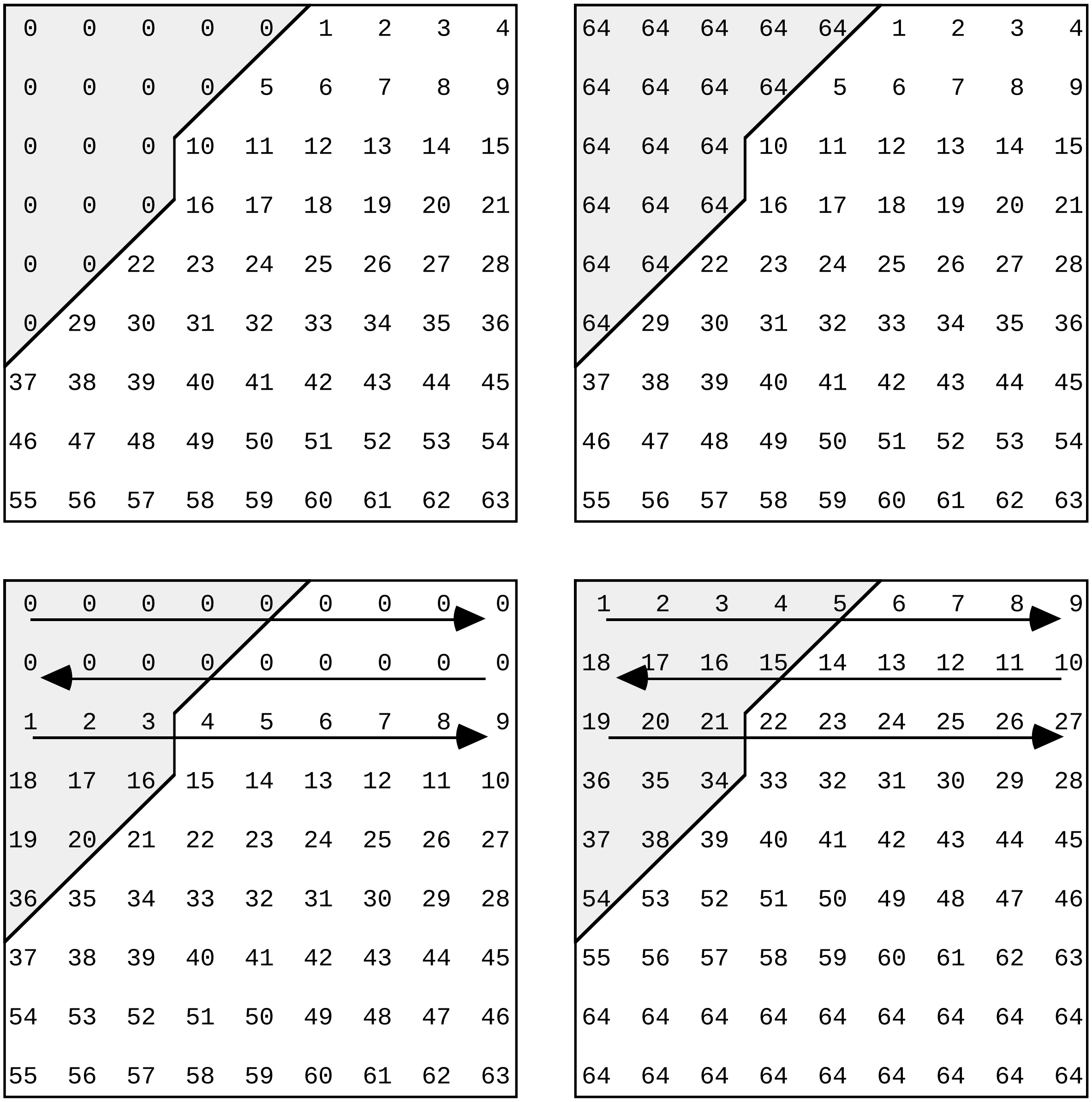

$2\sqrt[4]{p}$
$2\sqrt[4]{p}$
$2\sqrt{p}$ items
Shortest path from the upper left triangle to opposite corner is $2\sqrt{p} - 2\sqrt[4]{p} - 2$ hops
x[t]: the value held in this corner after t steps

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



0	0	0	0	0	1	2	3	4
0	0	0	0	5	6	7	8	9
0	0	0	10	11	12	13	14	15
0	0	0	16	17	18	19	20	21
0	0	22	23	24	25	26	27	28
0	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63

64	64	64	64	64	1	2	3	4
64	64	64	64	5	6	7	8	9
64	64	64	10	11	12	13	14	15
64	64	64	16	17	18	19	20	21
64	64	22	23	24	25	26	27	28
64	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9
18	17	16	15	14	13	12	11	10
19	20	21	22	23	24	25	26	27
36	35	34	33	32	31	30	29	28
37	38	39	40	41	42	43	44	45
54	53	52	51	50	49	48	47	46
55	56	57	58	59	60	61	62	63

1	2	3	4	5	6	7	8	9
18	17	16	15	14	13	12	11	10
19	20	21	22	23	24	25	26	27
36	35	34	33	32	31	30	29	28
37	38	39	40	41	42	43	44	45
54	53	52	51	50	49	48	47	46
55	56	57	58	59	60	61	62	63
64	64	64	64	64	64	64	64	64
64	64	64	64	64	64	64	64	64

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

9.6 Achieving the Lower Bound

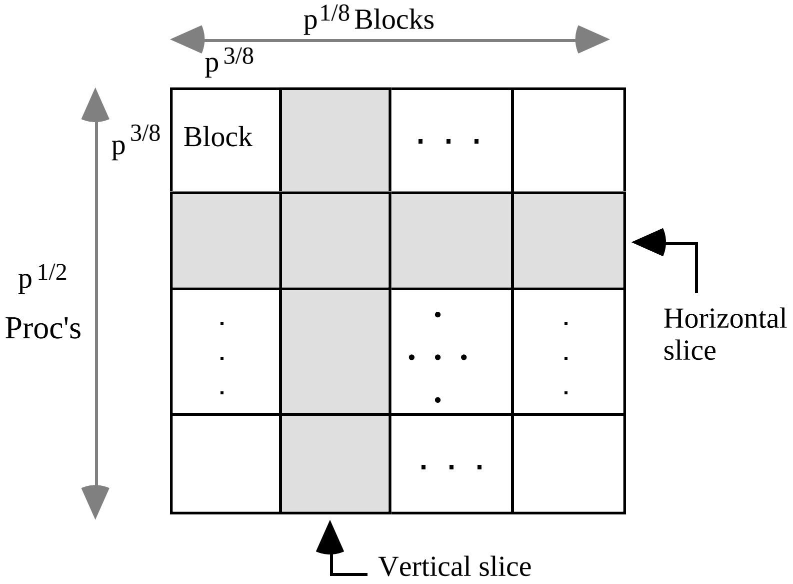


Fig. 9.16. Notation for the asymptotically optimal sorting algorithm.

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

Chapter Goals

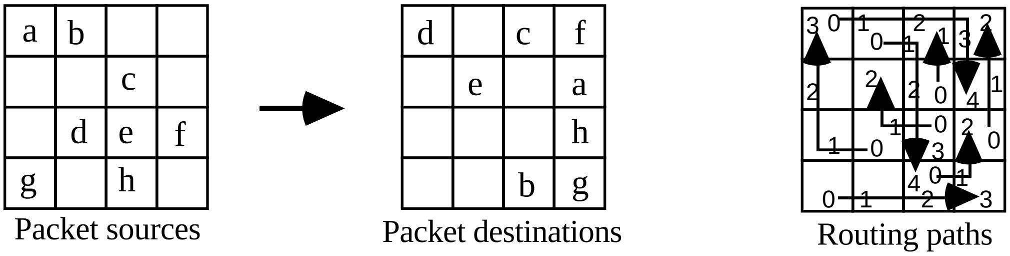
- Learn how to route multiple data items to their respective destinations

  (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

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



Packet sources

Packet destinations

Routing paths

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

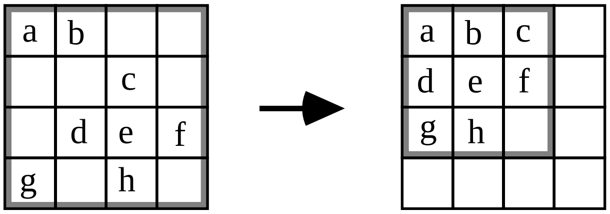


Fig. 10.1. Example of data compaction or packing.

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

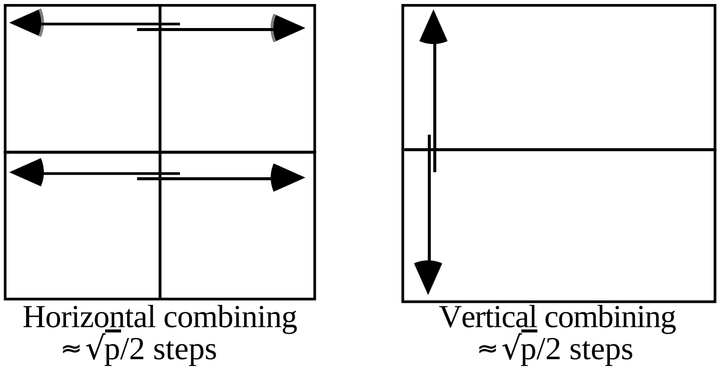


Fig. 10.2. Recursive semigroup computation in a 2D mesh.

Parallel prefix computation

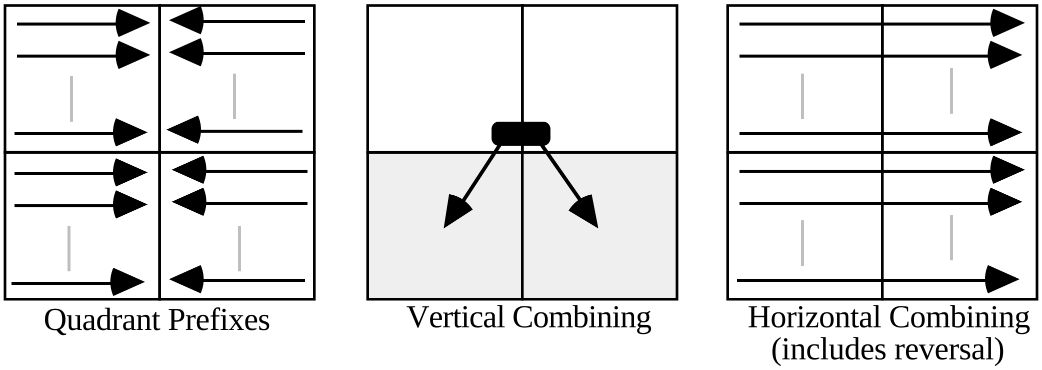


Fig. 10.3. Recursive parallel prefix computation in a 2D mesh.

Routing within a row or column

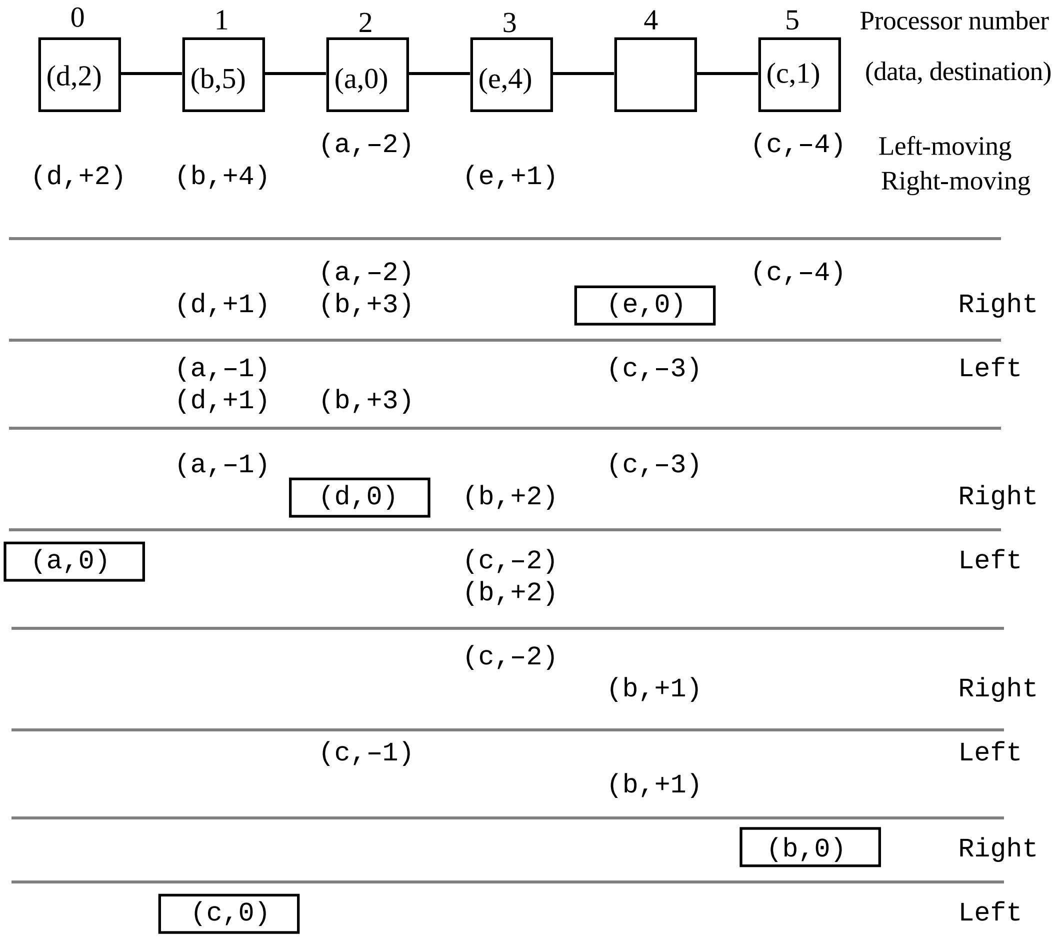
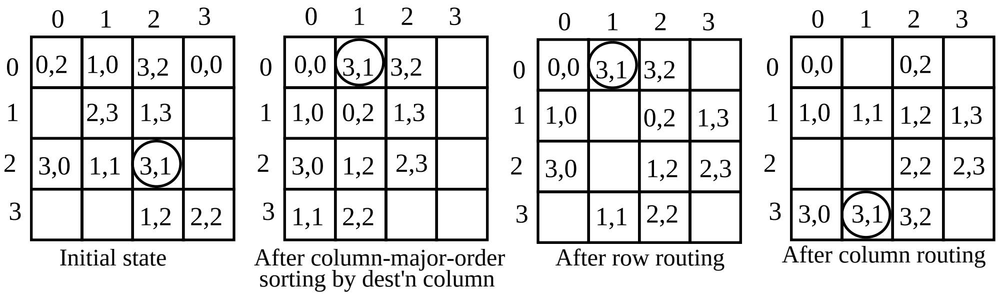


Fig. 10.4. Example of routing multiple packets on a linear array.

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



	0	1	2	3
0	0,2	1,0	3,2	0,0
1		2,3	1,3	
2	3,0	1,1	3,1	
3			1,2	2,2

Initial state

	0	1	2	3
0	0,0	3,1	3,2	
1	1,0	0,2	1,3	
2	3,0	1,2	2,3	
3	1,1	2,2		

After column-major-order sorting by dest'n column

	0	1	2	3
0	0,0	3,1	3,2	
1	1,0		0,2	1,3
2	3,0		1,2	2,3
3		1,1	2,2	

After row routing

	0	1	2	3
0	0,0		0,2	
1	1,0	1,1	1,2	1,3
2			2,2	2,3
3	3,0	3,1	3,2	

After column routing

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

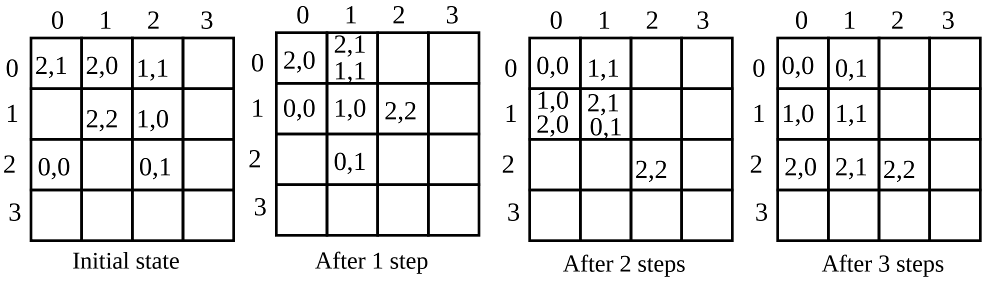
$$
\begin{aligned}
T &= 3p^{1/2} + o(p^{1/2}) && \text{\{snakelike sorting\}} \\
&+ p^{1/2} && \text{\{column reversal\}} \\
&+ 2p^{1/2} - 2 && \text{\{row \& column routing\}} \\
&= 6p^{1/2} + o(p^{1/2})
\end{aligned}
$$

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)



Initial state

	0	1	2	3
0	2,1	2,0	1,1	
1		2,2	1,0	
2	0,0		0,1	
3				

After 1 step

	0	1	2	3
0	2,0	2,1 1,1		
1	0,0	1,0	2,2	
2		0,1		
3				

After 2 steps

	0	1	2	3
0	0,0	1,1		
1	1,0 2,0	2,1 0,1		
2			2,2	
3				

After 3 steps

	0	1	2	3
0	0,0	0,1		
1	1,0	1,1		
2	2,0	2,1	2,2	
3				

Fig. 10.6. Example of greedy row-first routing on a 2D mesh.

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

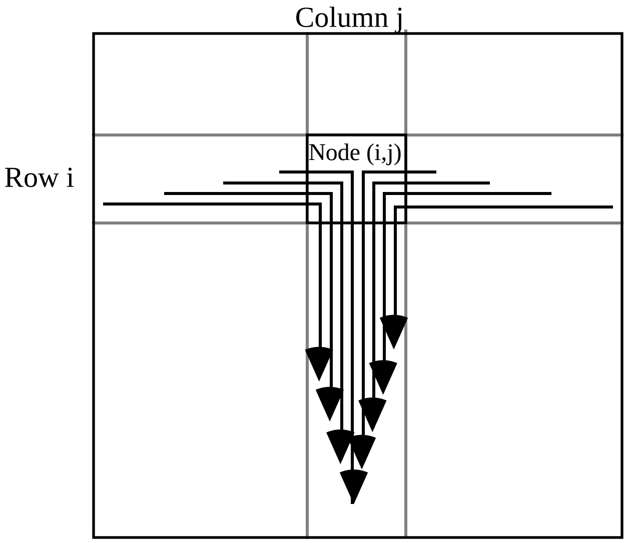


Fig. 10.7. Demonstrating the worst-case buffer requirement with row-first routing.

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

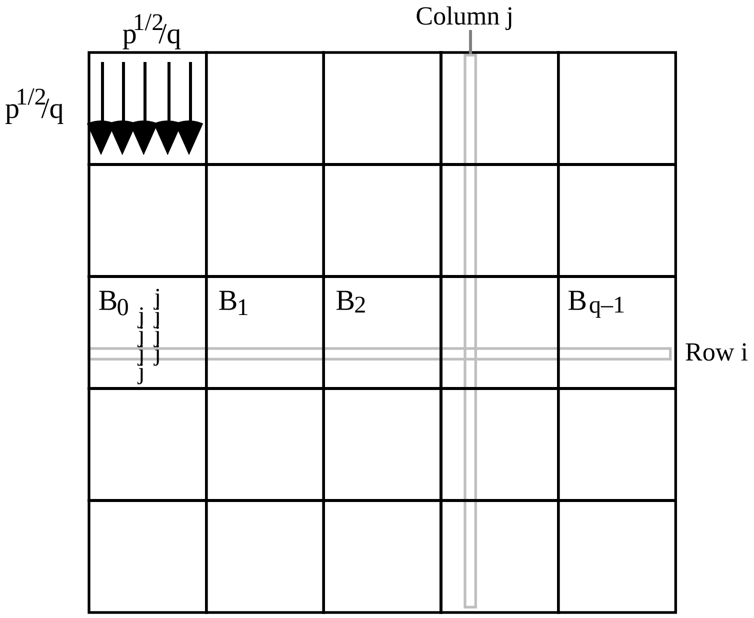


Fig. 10.8. Illustrating the structure of the intermediate routing algorithm.

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

Buffer space per node

r_k = number of packets in B_k headed for column j

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

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 two random instances by picking a random intermediate node for each message

Using combining for concurrent writes:

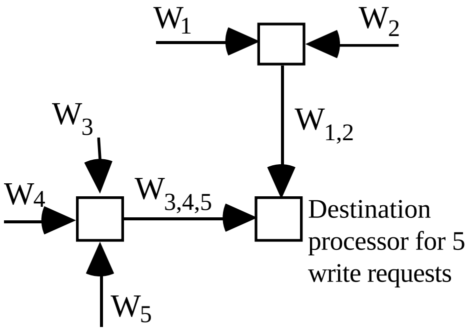


Fig. 10.9. Combining of write requests headed for the same destination.

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 & destination

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

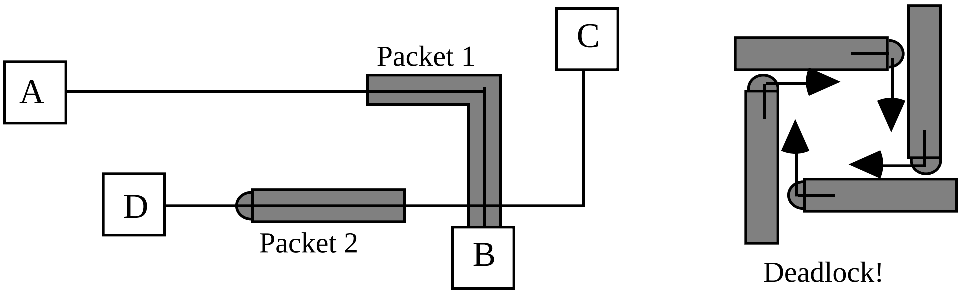


Fig. 10.10. The notions of worms and deadlock in wormhole routing.

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

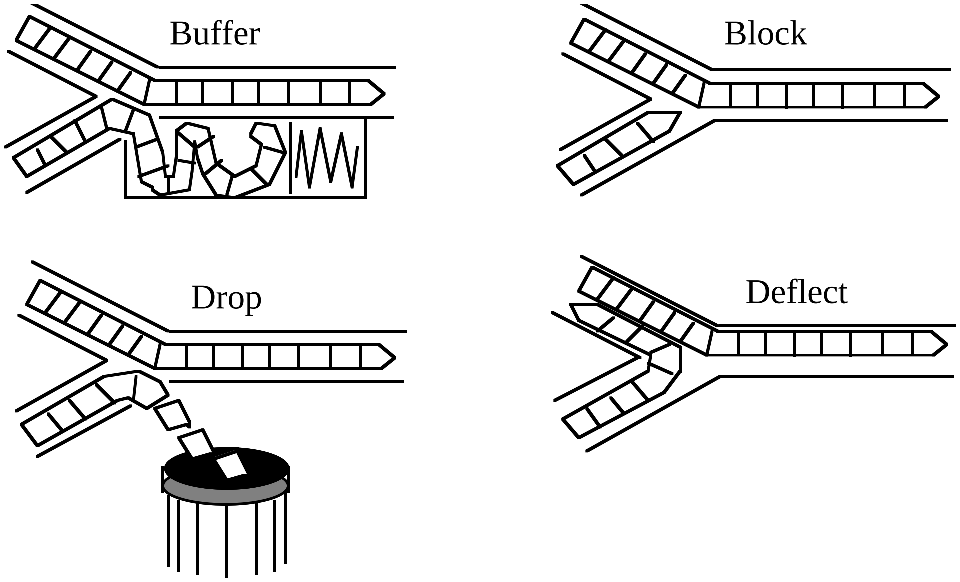
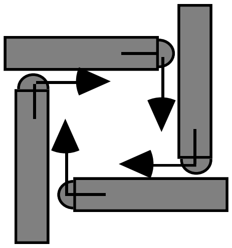


Fig. 10.11. Various ways of dealing with conflicts in wormhole routing.

The deadlock problem in wormhole routing

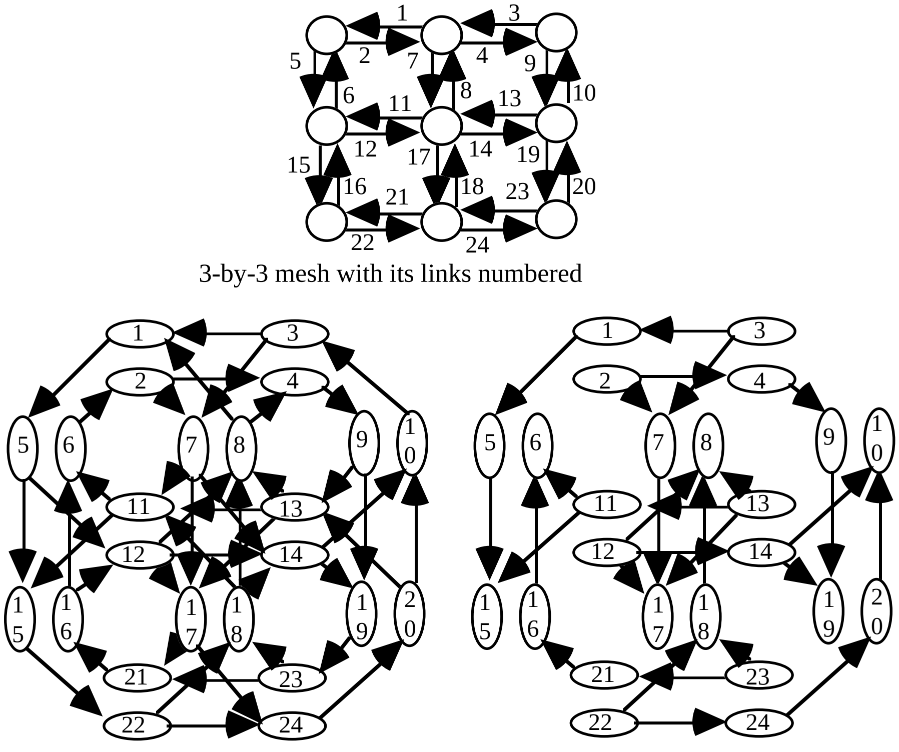


Deadlock!

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

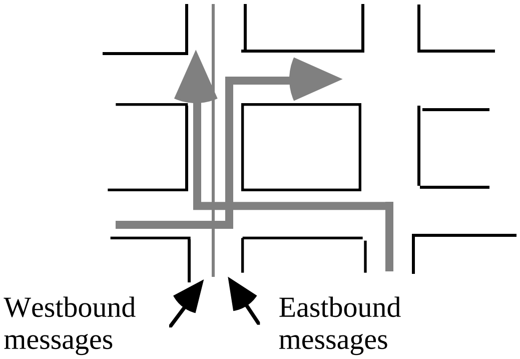


Fig. 10.13. Use of virtual channels for avoiding deadlocks.

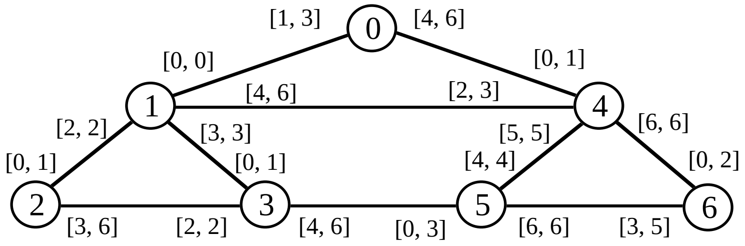


Figure for Problem 10.14.

11 Numerical 2D Mesh Algorithms

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

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

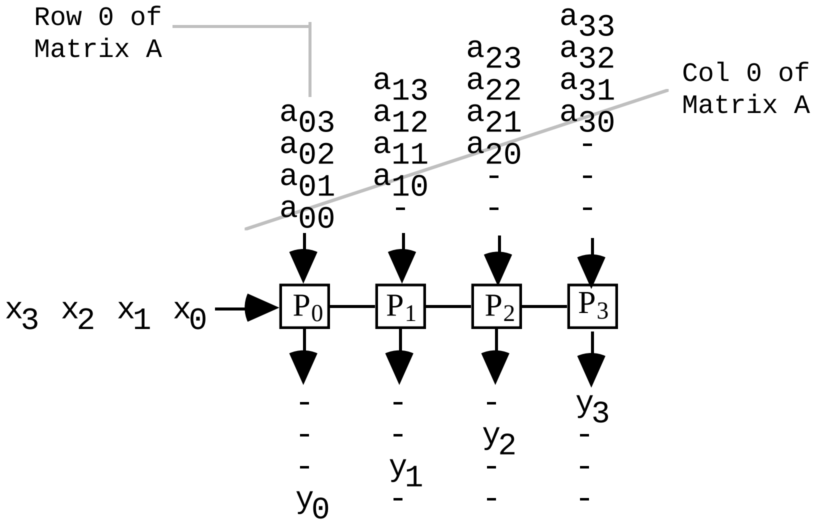
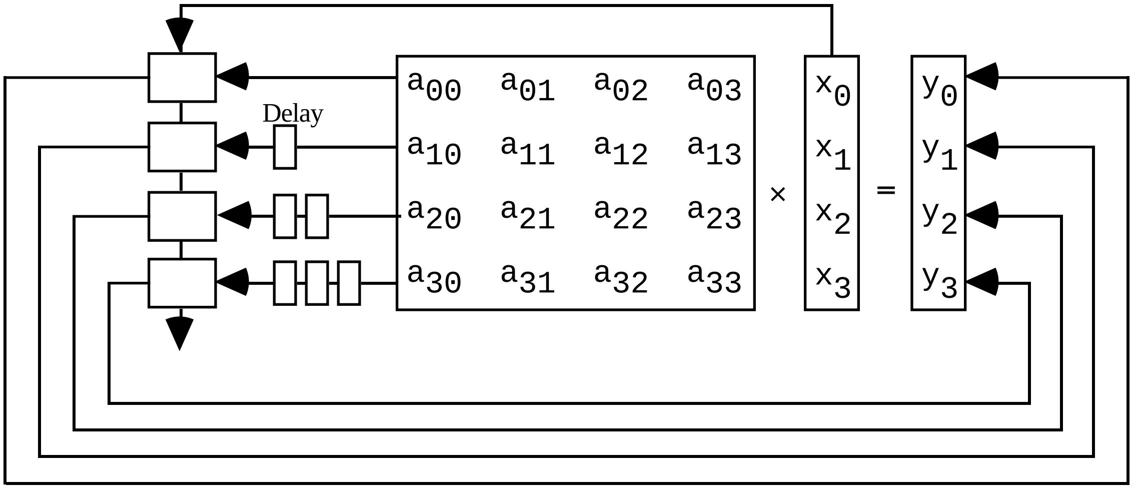


Fig. 11.1. Matrix–vector multiplication on a linear array.



With p = m processors, T = 2m – 1 = 2p – 1

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

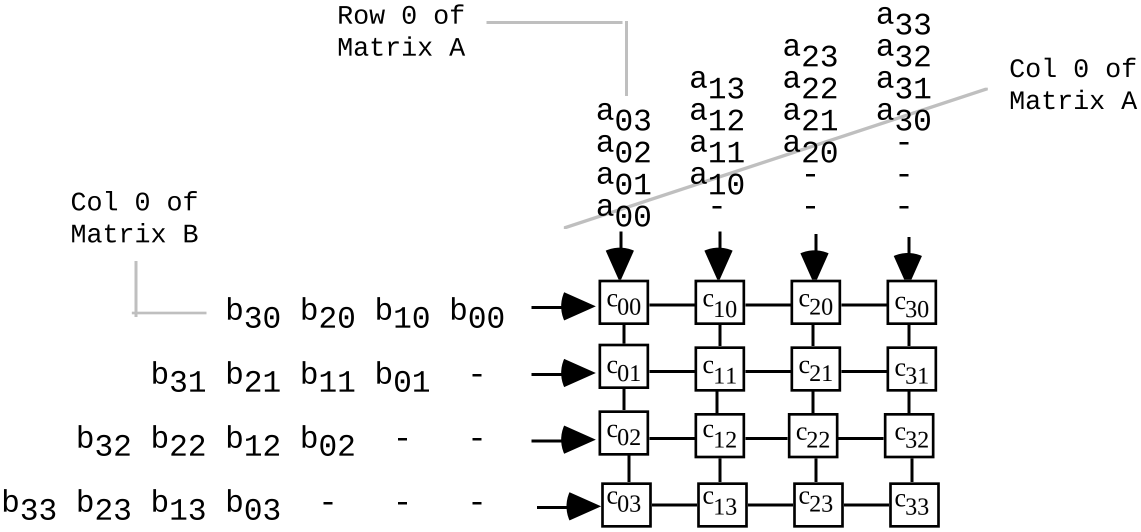


Fig. 11.2. Matrix–matrix multiplication on a 2D mesh.

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

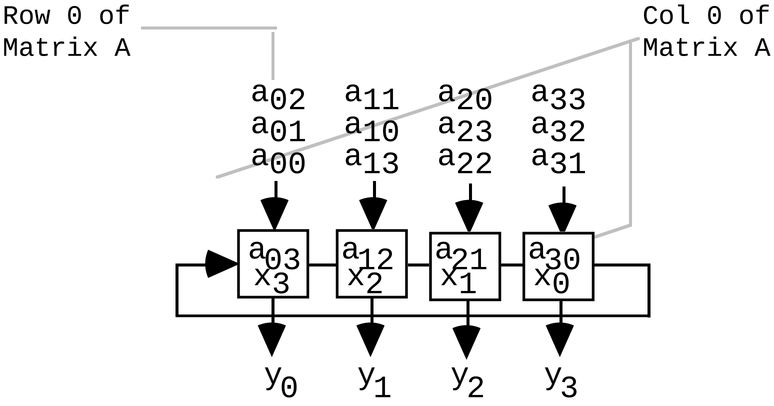


Fig. 11.3. Matrix-vector multiplication on a ring.

With p = m processors, T = m = p

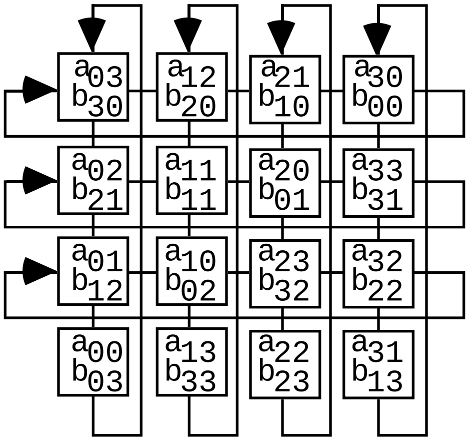


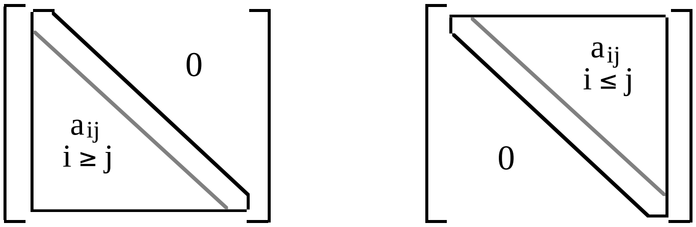
Fig. 11.4. Matrix-matrix multiplication on a torus.

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



Lower triangular (if $a_{ii} = 0$, then it is strictly lower triangular)

Upper triangular (if $a_{ii} = 0$, then it is strictly upper triangular)

Fig. 11.5. Lower/upper triangular square matrix.

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

Forward substitution (lower triangular)

Back substitution (upper triangular)

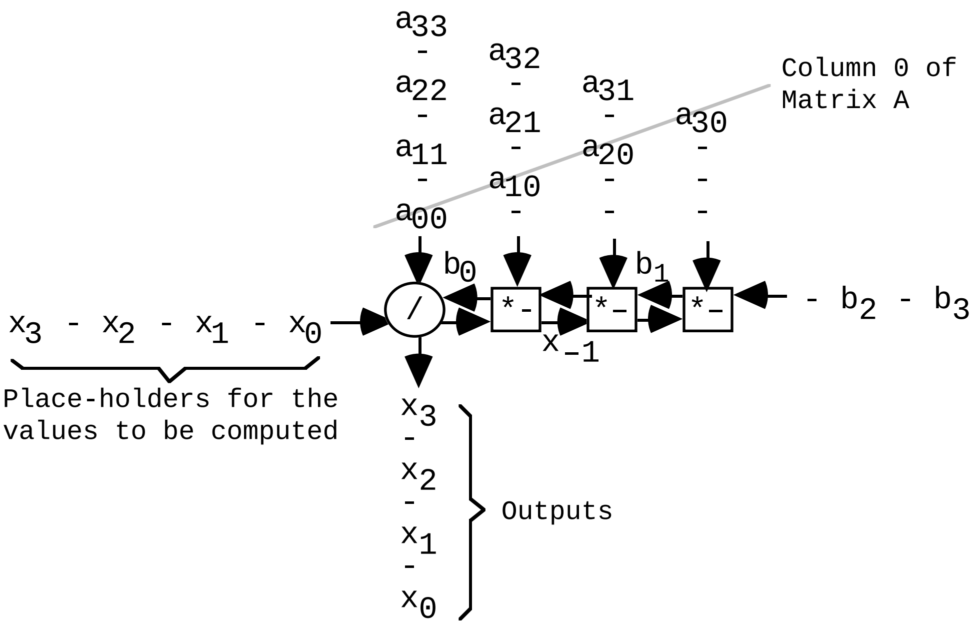

a_{33} - a_{22} - a_{11} - a_{00}
a_{32} - a_{21} - a_{10} -
a_{31} - a_{20} - - -
a_{30} - - - -
Column 0 of Matrix A
b_0
b_1
- b_2 - b_3
/
*-
*-
*-
x_3 - x_2 - x_1 - x_0
x_{-1}
Place-holders for the values to be computed
x_3 - x_2 - x_1 - x_0
Outputs

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

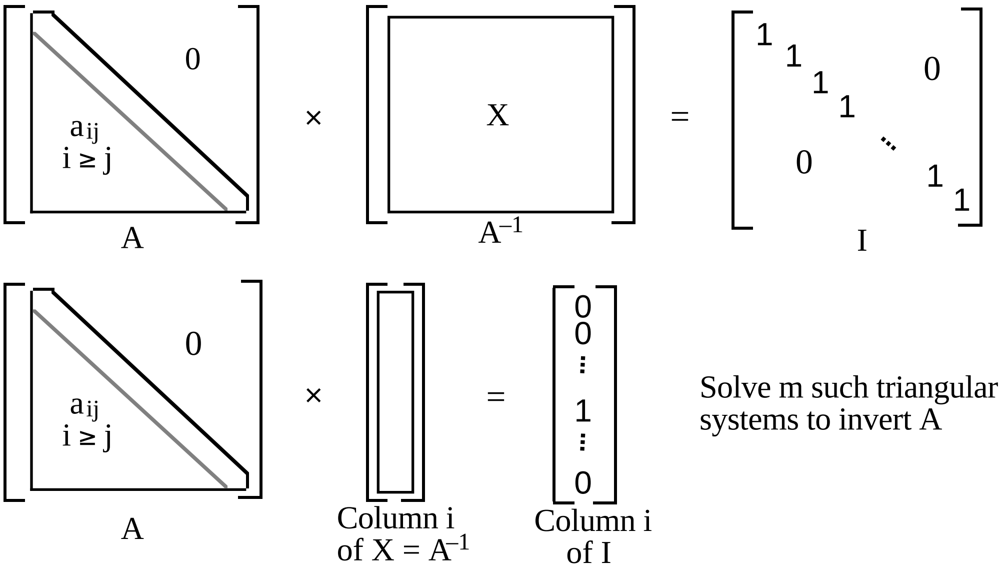


Fig. 11.7. Inverting a triangular matrix by solving triangular systems of linear equations.

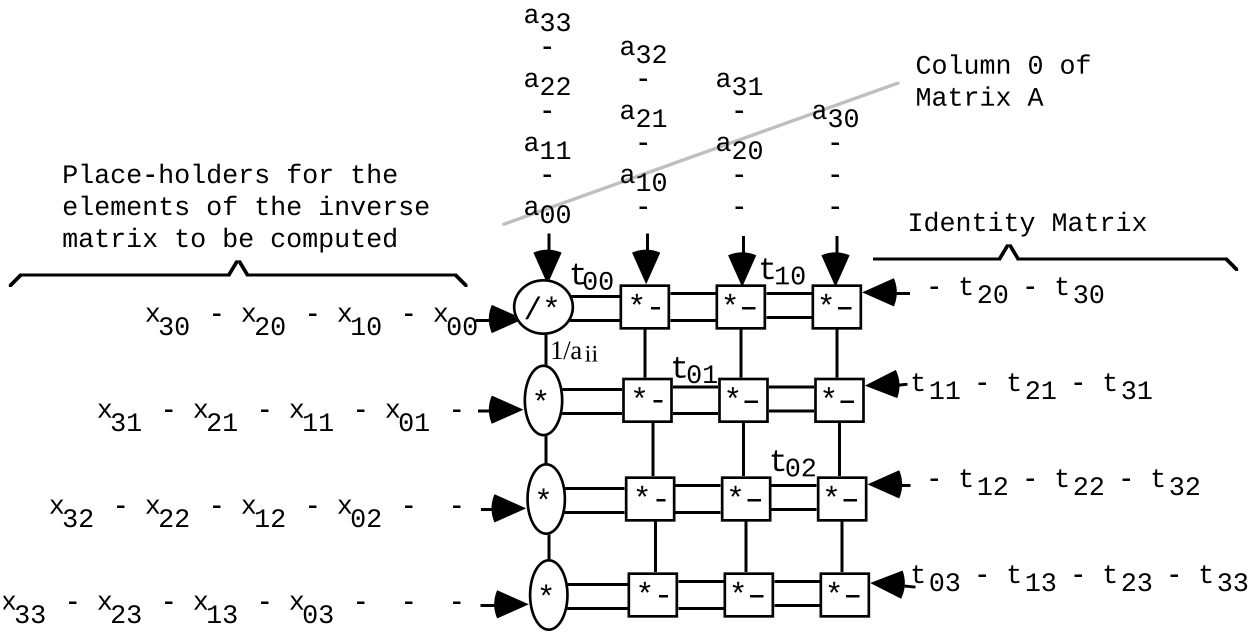


Fig. 11.8. Inverting a lower triangular matrix on a 2D mesh.

11.3 Tridiagonal System of Linear Equations

$$l_0 \begin{bmatrix} d_0 & u_0 & & & & \\ l_1 & d_1 & u_1 & & 0 & \\ & l_2 & d_2 & u_2 & & \\ & & l_3 & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & u_{m-2} \\ & & & & l_{m-1} & d_{m-1} \end{bmatrix} u_{m-1} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{m-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-1} \end{bmatrix}$$

Fig. 11.9. A tridiagonal system of linear equations.

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

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

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

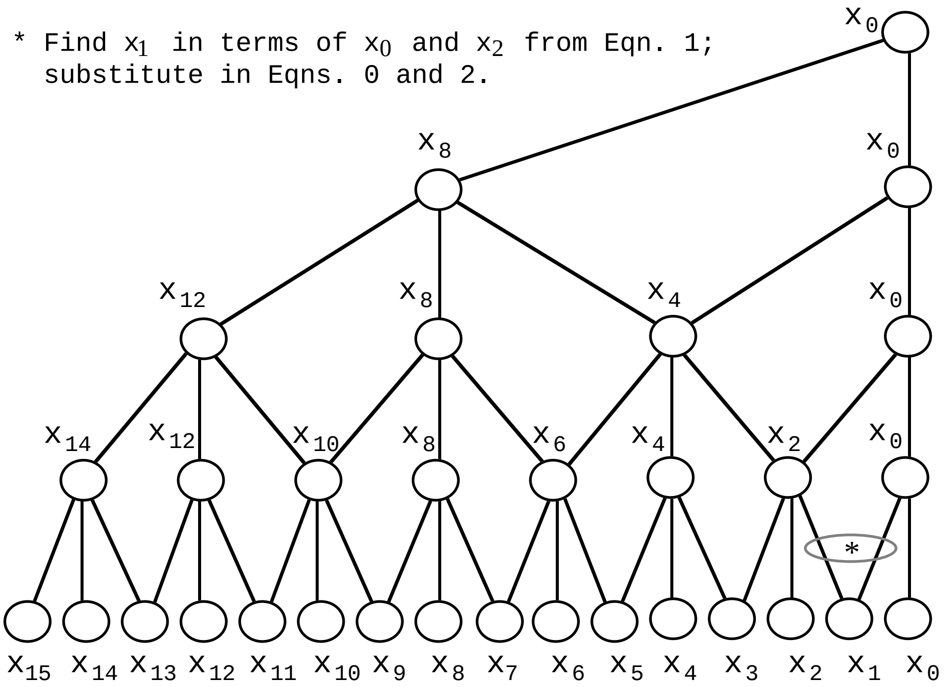
$$x_i = (1/d_i)\,(b_i - l_i\,x_{i-1} - u_i\,x_{ii+1})$$

Take the x_j 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 ($0 \leq i < m$) an equation of the form:

$$-\frac{l_{i-1}l_i}{d_{i-1}}x_{i-2} + (d_i - \frac{l_i u_{i-1}}{d_{i-1}} - \frac{u_i l_{i+1}}{d_{i+1}})\, x_i - \frac{u_i u_{i+1}}{d_{i+1}}x_{i+2} = b_i - \frac{l_i b_{i-1}}{d_{i-1}} - \frac{u_i b_{i+1}}{d_{i+1}}$$

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



* Find x_1 in terms of x_0 and x_2 from Eqn. 1; substitute in Eqns. 0 and 2.

x_0
x_8
x_0
x_{12}
x_8
x_4
x_0
x_{14}
x_{12}
x_{10}
x_8
x_6
x_4
x_2
x_0
*
x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0

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

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

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

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.

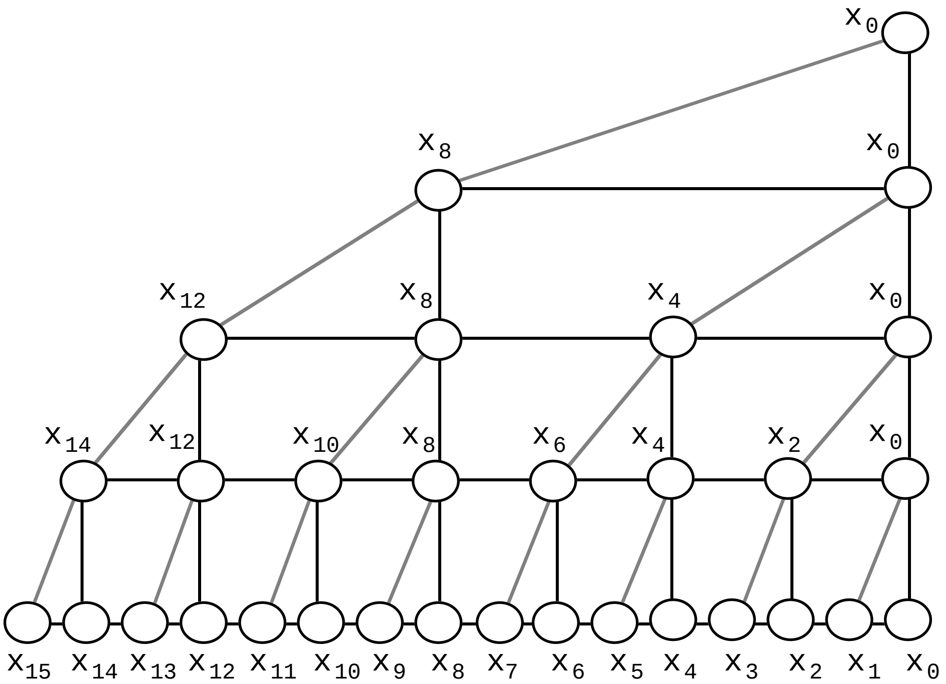
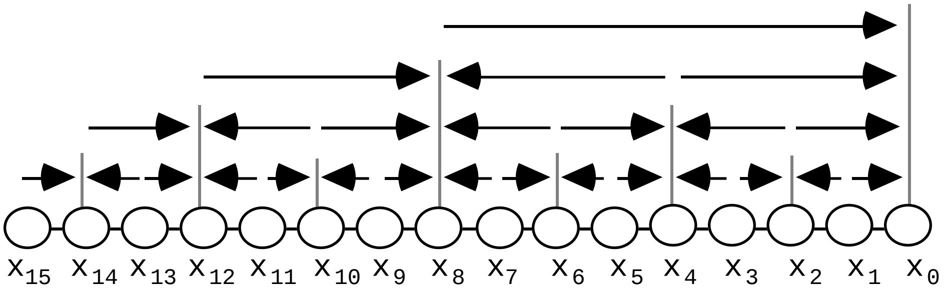

x_0
x_8
x_0
x_{12}
x_8
x_4
x_0
x_{14}
x_{12}
x_{10}
x_8
x_6
x_4
x_2
x_0
x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0

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

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


x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0

Communication time = 2(1 + 2 + 4 + . . . + 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

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

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

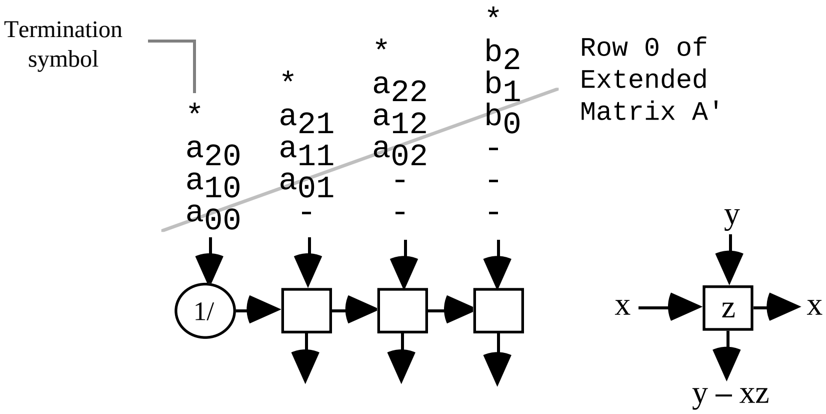


Fig. 11.12. A linear array performing the first phase of Gaussian elimination.

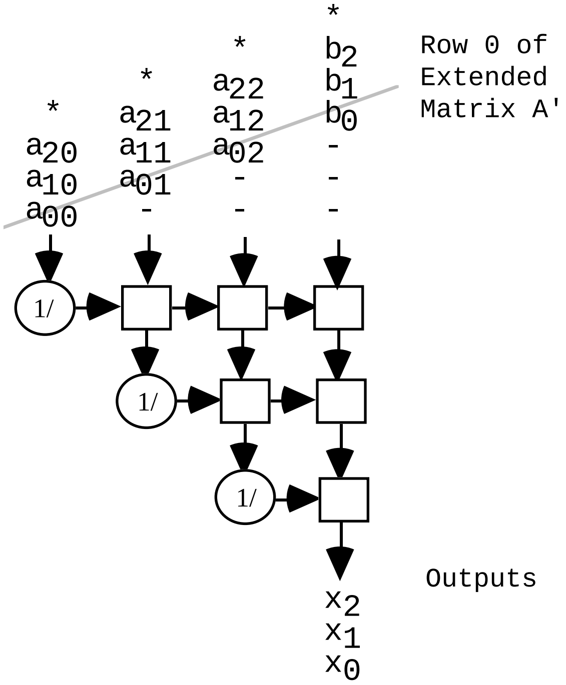


Fig. 11.13. Implementation of Gaussian elimination on a 2D array.

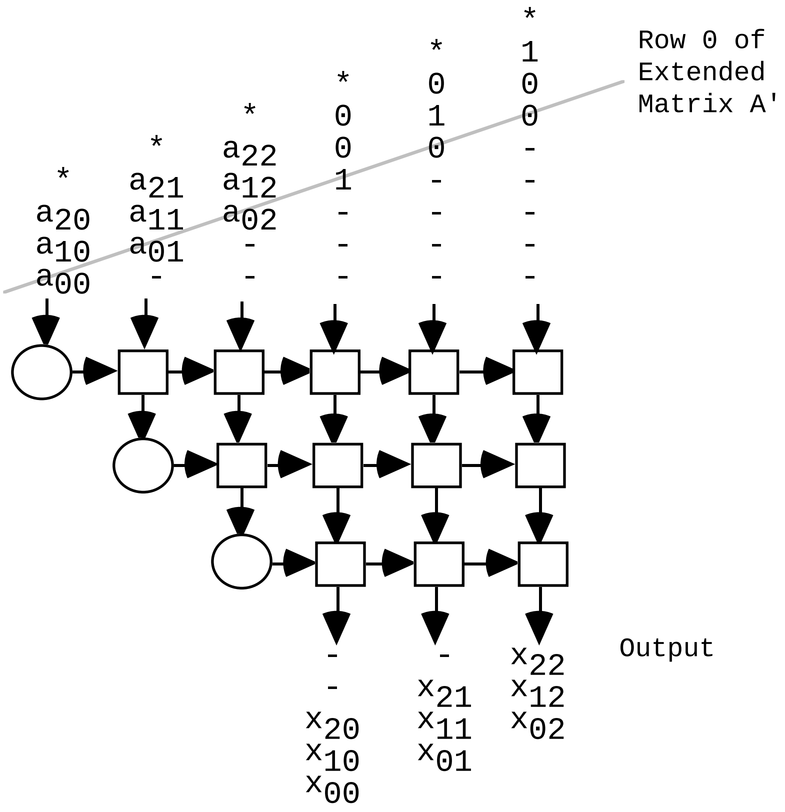


Fig. 11.14. Matrix inversion by Gaussian elimination.

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.

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

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.

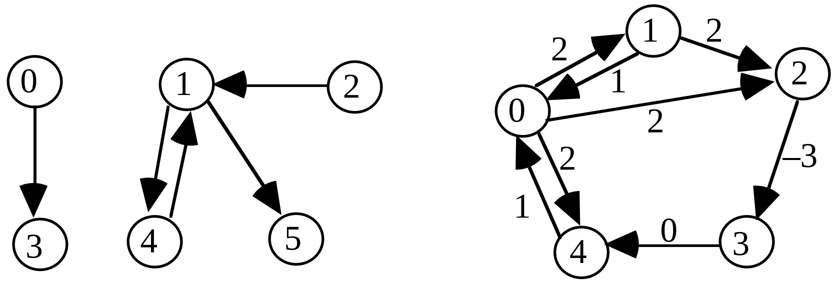
A variant: Jacobi overrelaxation

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

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

For smaller γ, overrelaxation may offer better performance

11.5 Graph Algorithms



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

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^*

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

To compute A^*, we need only proceed up to the term A^{n-1}; if there is a path from i to j, there must be 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, · · · , 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 in view of the $O(n^3)$ sequential complexity of the transitive closure problem

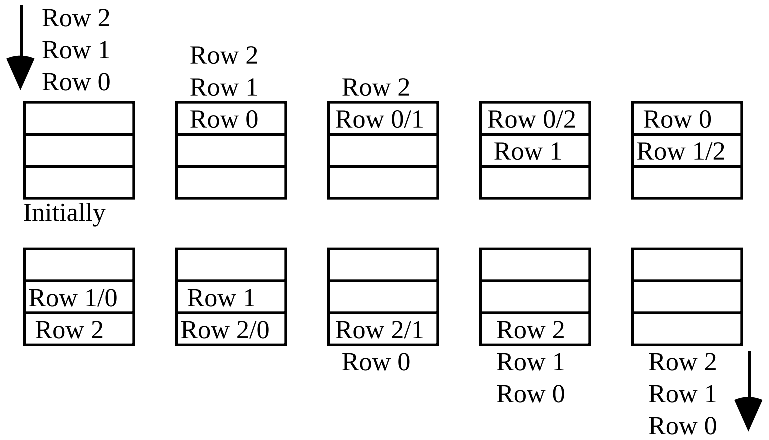
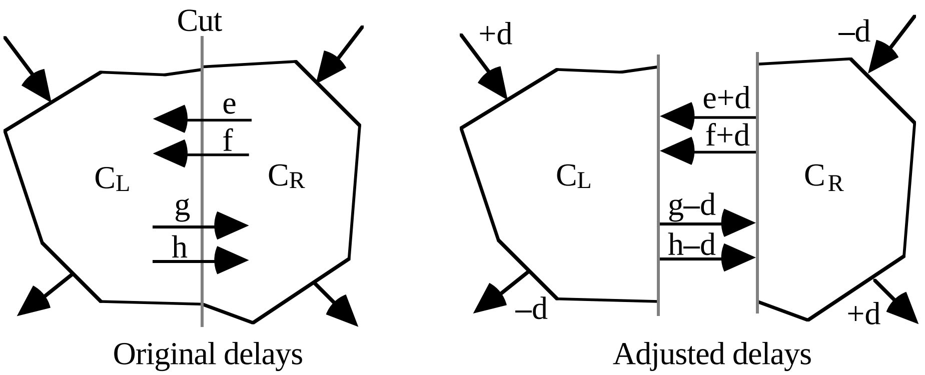


Fig. 11.16. Transitive closure algorithm on a 2D mesh.

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]

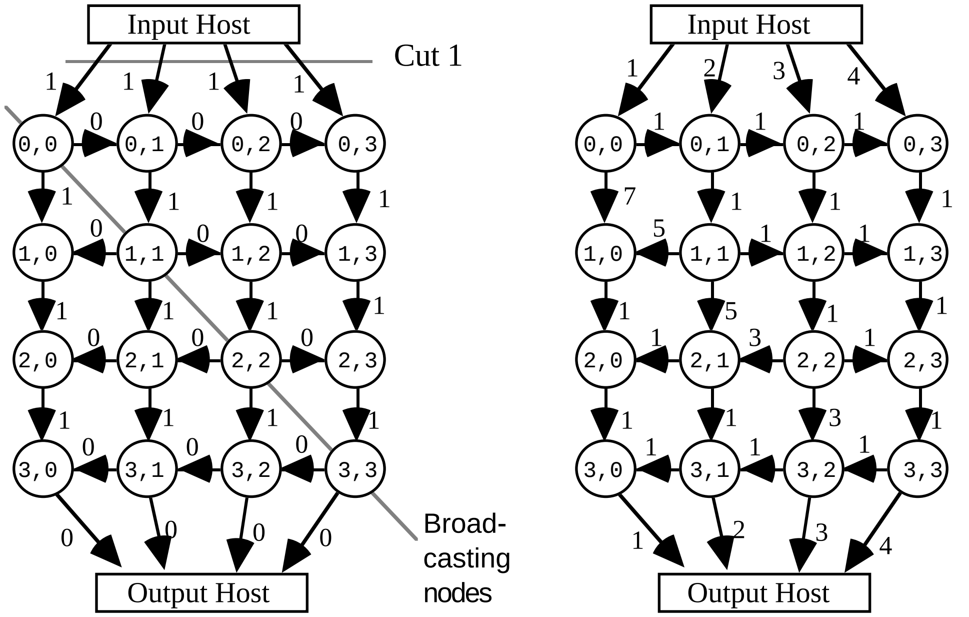


Fig. 11.17. Systolic retiming to eliminate broadcasting.

The diagram on the left represents the preceding algorithm

Zero-time horizontal arrows represent brodcasting by diagonal elements

The goal of systolization is to eliminate all zero-time transitions

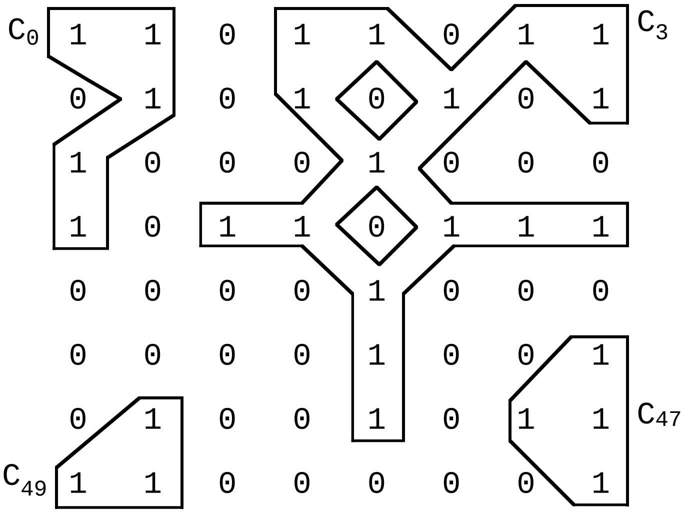
To systolize the preceding example:

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

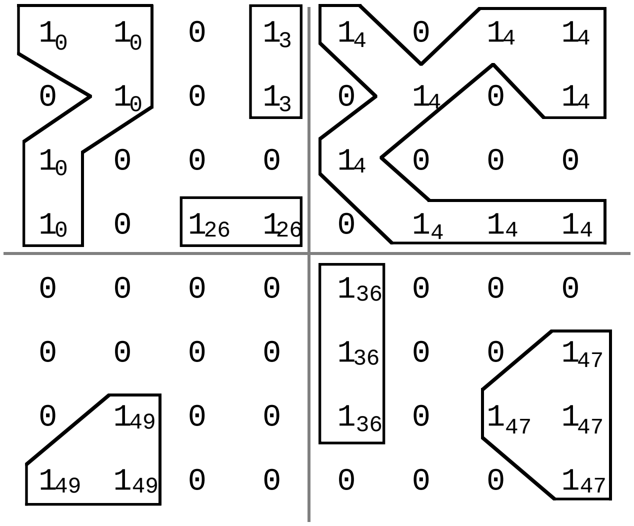
Labeling the connected components of a binary image



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

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

Recursive algorithm, p = n: $T(n) = T(n/4) + O(\sqrt{n}) = O(\sqrt{n})$



1_{0}	1_{0}	0	1_{3}	1_{4}	0	1_{4}	1_{4}
0	1_{0}	0	1_{3}	0	1_{4}	0	1_{4}
1_{0}	0	0	0	1_{4}	0	0	0
1_{0}	0	1_{26}	1_{26}	0	1_{4}	1_{4}	1_{4}
0	0	0	0	1_{36}	0	0	0
0	0	0	0	1_{36}	0	0	1_{47}
0	1_{49}	0	0	1_{36}	0	1_{47}	1_{47}
1_{49}	1_{49}	0	0	0	0	0	1_{47}

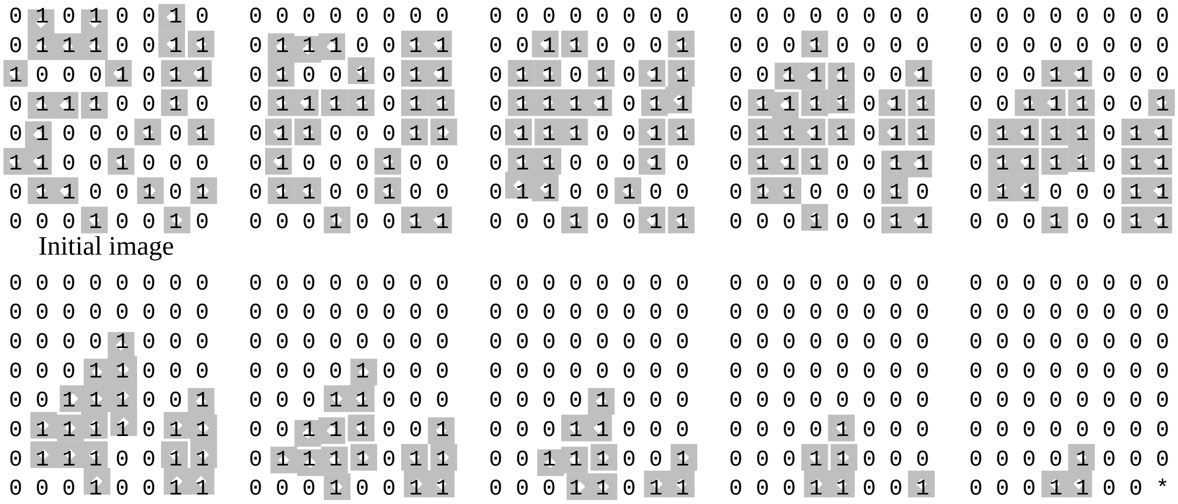
Fig. 11.19. Finding the connected components via divide and conquer.

Lavialdi's algorithm

$\begin{array}{ll} 0 & 1 \\ 1 & \boxed{0} \end{array}$ $\quad$ $\begin{array}{ll} 1 & 1 \\ 1 & \boxed{0} \end{array}$ $\quad$ 0 is changed to 1 if N = W = 1

$\begin{array}{ll} 0 & 0 \\ 0 & \boxed{1} \end{array}$ $\quad$ 1 is changed to 0 if N = W = NW = 0

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



```
0 1 0 1 0 0 1 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
0 1 1 1 0 0 1 1   0 1 1 1 0 0 1 1   0 0 1 1 0 0 0 1   0 0 0 1 0 0 0 0   0 0 0 0 0 0 0 0
1 0 0 0 1 0 1 1   0 1 0 0 1 0 1 1   0 1 1 0 1 0 1 1   0 0 1 1 1 0 0 1   0 0 0 1 1 0 0 0
0 1 1 1 0 0 1 0   0 1 1 1 1 0 1 1   0 1 1 1 1 0 1 1   0 1 1 1 1 0 1 1   0 0 1 1 1 0 0 1
0 1 0 0 0 1 0 1   0 1 1 0 0 0 1 1   0 1 1 1 0 0 1 1   0 1 1 1 1 0 1 1   0 1 1 1 1 0 1 1
1 1 0 0 1 0 0 0   0 1 0 0 0 1 0 0   0 1 1 0 0 0 1 0   0 1 1 1 0 0 1 1   0 1 1 1 1 0 1 1
0 1 1 0 0 1 0 1   0 1 1 0 0 1 0 0   0 1 1 0 0 1 0 0   0 1 1 0 0 0 1 0   0 1 1 0 0 0 1 1
0 0 0 1 0 0 1 0   0 0 0 1 0 0 1 1   0 0 0 1 0 0 1 1   0 0 0 1 0 0 1 1   0 0 0 1 0 0 1 1
  Initial image
0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0   0 0 0 0 1 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
0 0 1 1 1 0 0 1   0 0 0 1 1 0 0 0   0 0 0 0 1 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
0 1 1 1 1 0 1 1   0 0 1 1 1 0 0 1   0 0 0 1 1 0 0 0   0 0 0 0 1 0 0 0   0 0 0 0 0 0 0 0
0 1 1 1 0 0 1 1   0 1 1 1 1 0 1 1   0 0 1 1 1 0 0 1   0 0 0 1 1 0 0 0   0 0 0 0 1 0 0 0
0 0 0 1 0 0 1 1   0 0 0 1 0 0 1 1   0 0 0 1 1 0 1 1   0 0 0 1 1 0 0 1   0 0 0 1 1 0 0 *
```

Fig. 11.21. Example of the shrinkage phase of Lavialdi'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

Chapter Goals

- Study vairants 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

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 $D = 43, B = 23$

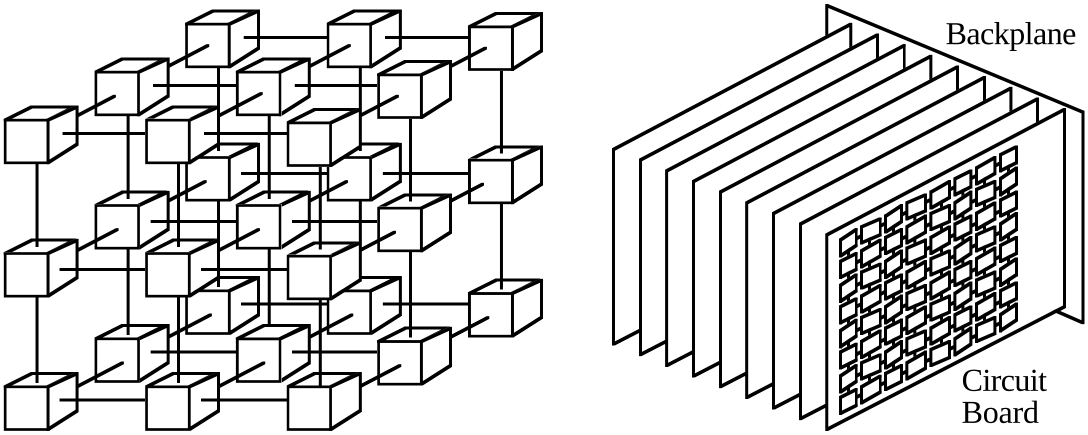


Fig. 12.1. 3D and 2.5D physical realizations of a 3D mesh.

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

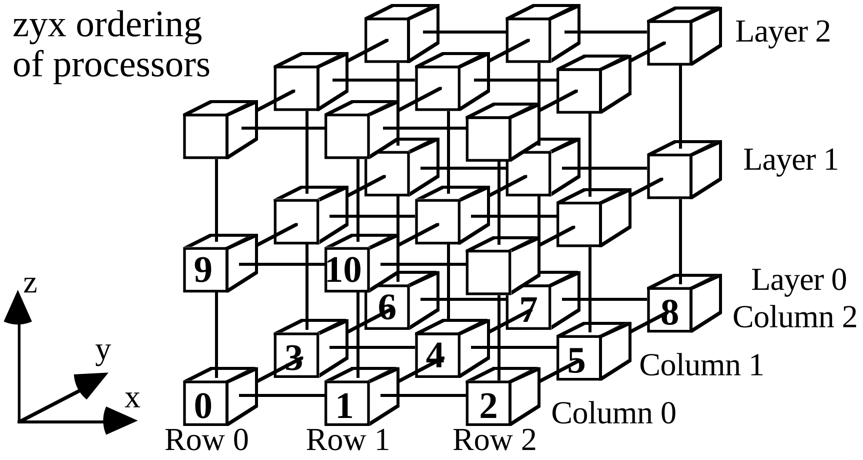
qD torus with m processors along each dimension

= 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 × m × 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 two steps of odd-even transposition along the z direction

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

Time = 4 × 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

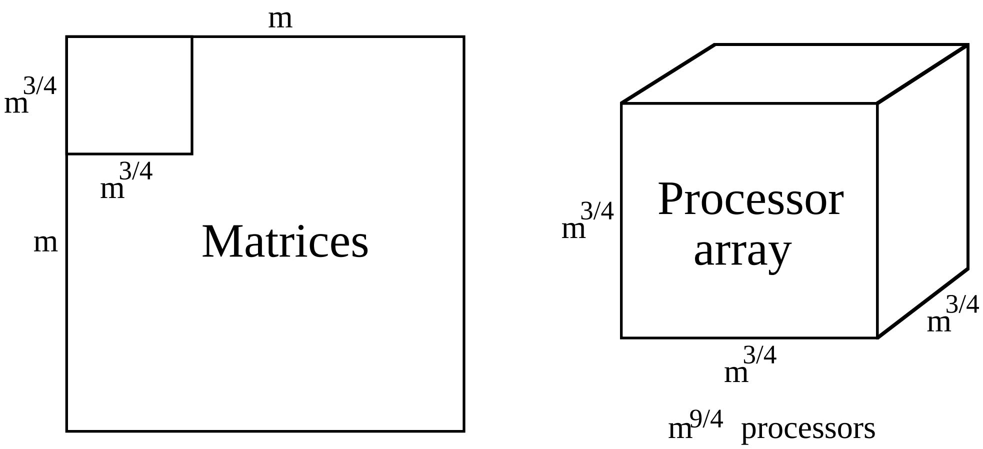
Phase 2: Route along z dimension to correct xy layer

Phase 3: Route along y dimension to correct column

Phase 4: Route along x dimension to destination node

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-add steps)

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

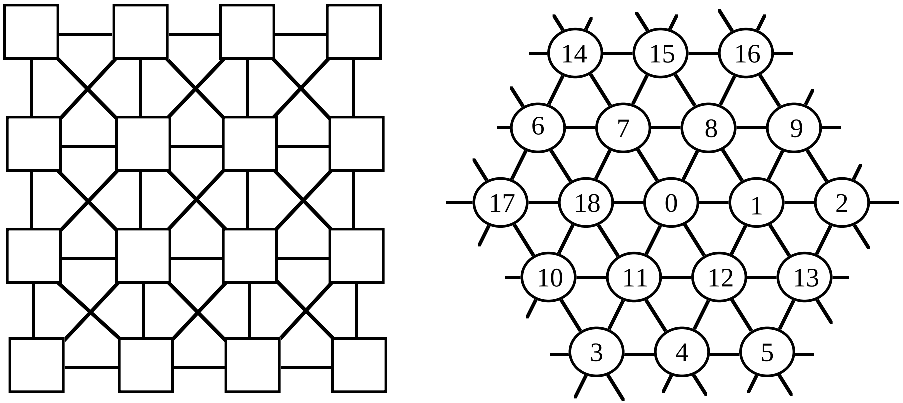
A low-dimensional mesh can simulate a high-dimensional mesh quite efficiently

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 allows us to make each channel wider?

12.2 Stronger and Weaker Connectivities

Fortified meshes



Node i connected to i ± 1,
i ± 7, and i ± 8 (mod 19).

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

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

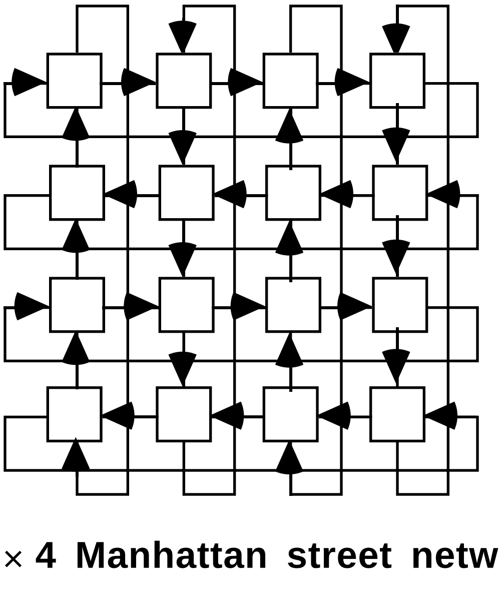


Fig. 12.3. A 4 × 4 Manhattan street network.

Pruned meshes

Same diameter as ordinary mesh, but much lower cost.

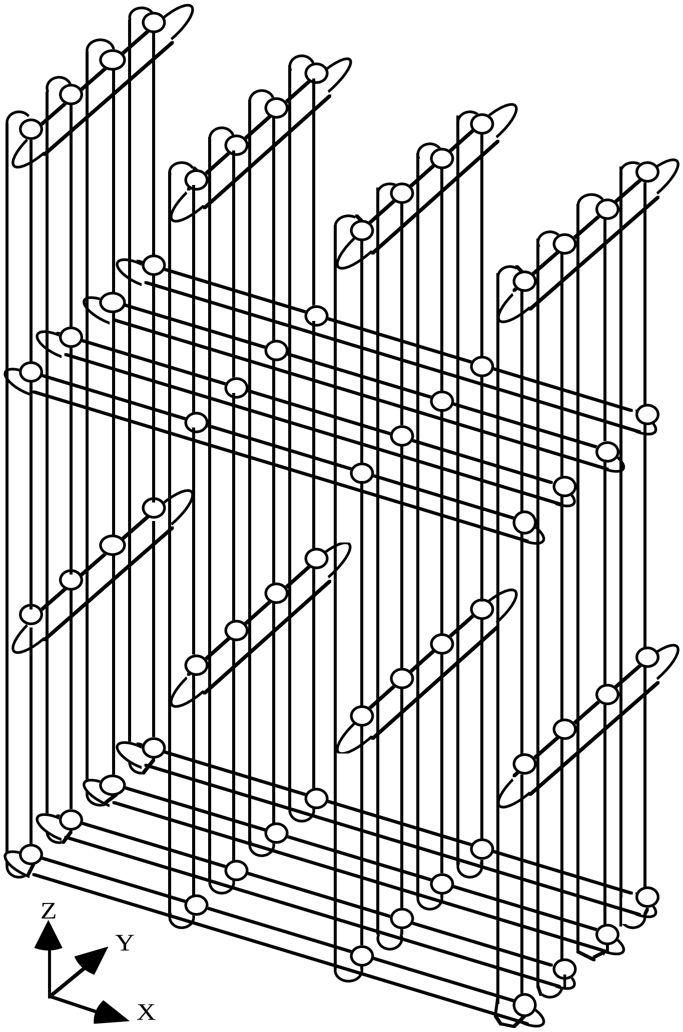
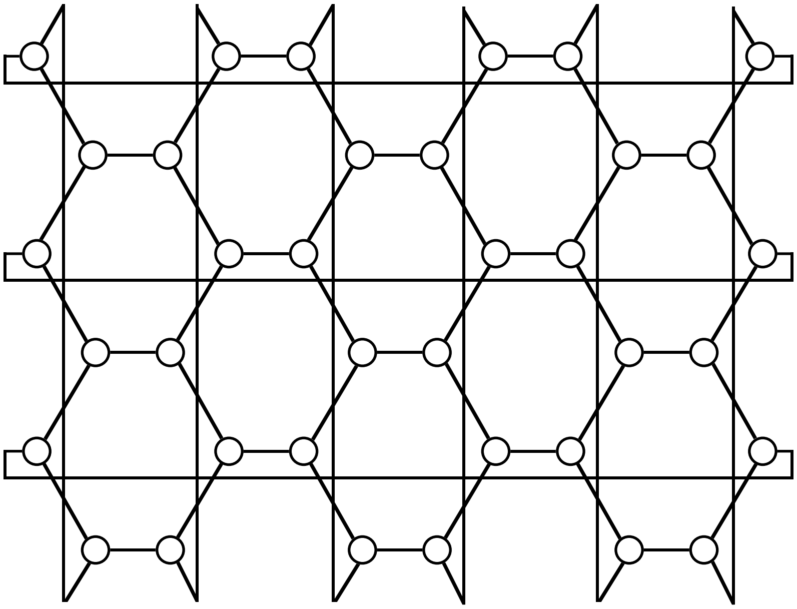


Fig. 12.4. A pruned 4 × 4 × 4 torus with nodes of degree four [Kwai97].

Pruning and orientation can be combined

Another form of pruning



Honeycomb mesh or torus.

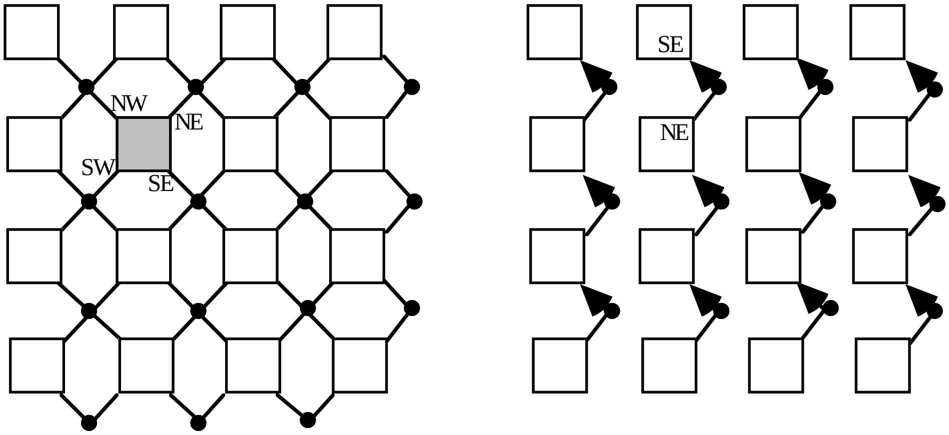


Fig. 12.5. Eight-neighbor mesh with shared links and example data paths.

12.3 Meshes Augmented with Nonlocal Links

Motivation: to reduce the diameter; a weakness of meshes

Bypass links or express channels along rows/columns

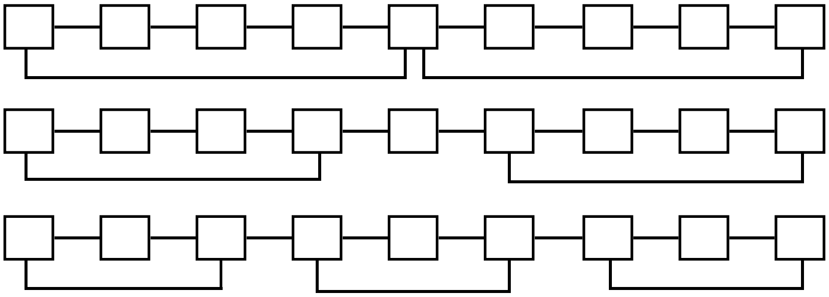


Fig. 12.6. Three examples of bypass links along the rows of a 2D mesh.

Using a single global bus

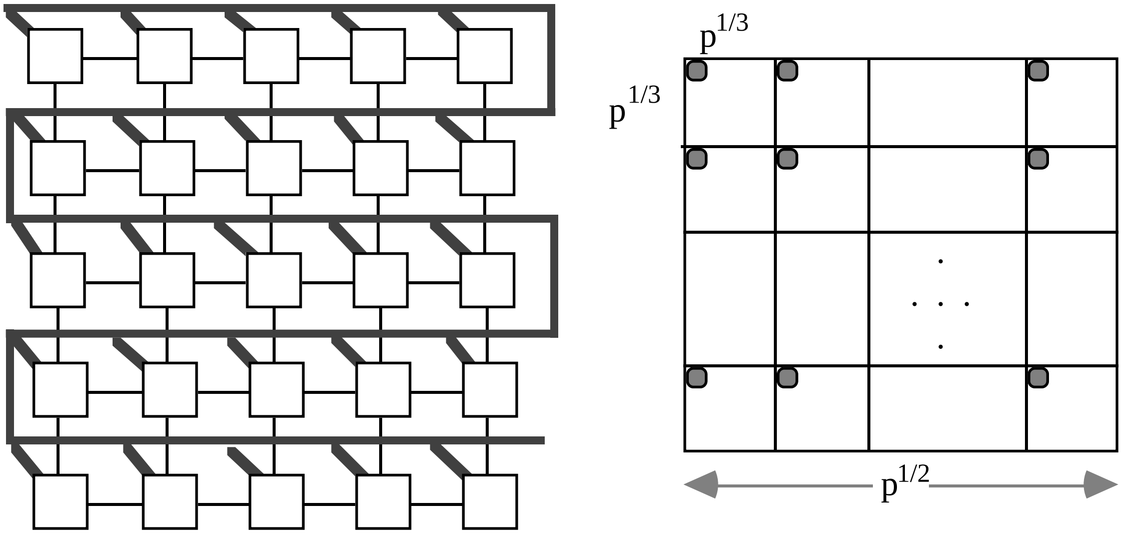


Fig. 12.7. Mesh with a global bus and semigroup computation on it.

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

<u>Semigroup computation on 2D mesh with a global bus</u>

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 in one step

Row and column buses

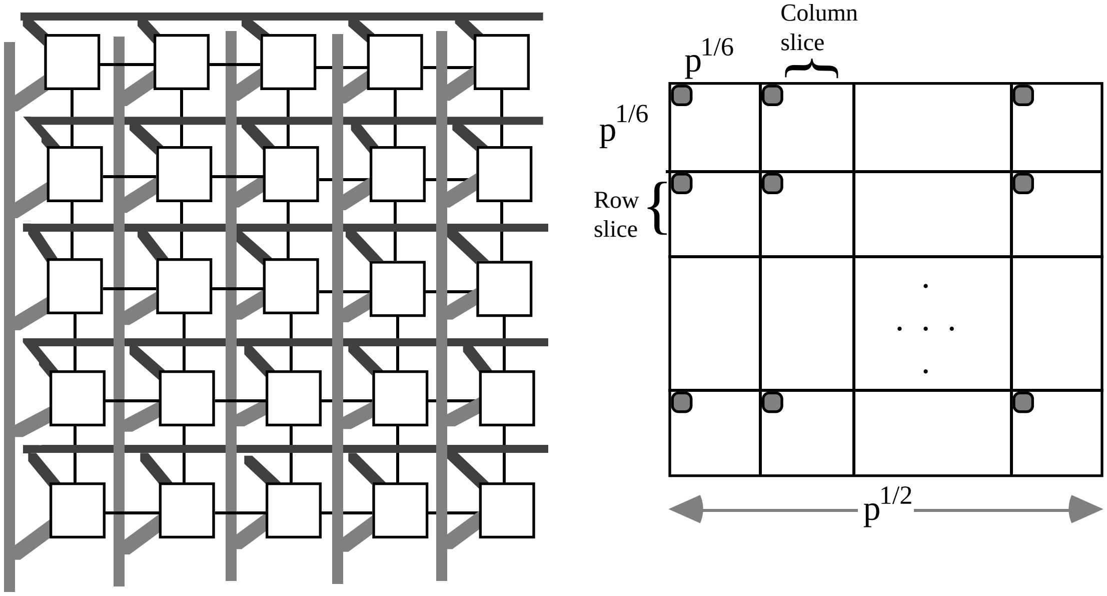


Fig. 12.8. Mesh with row/column buses and semigroup computation on it.

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 of the 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

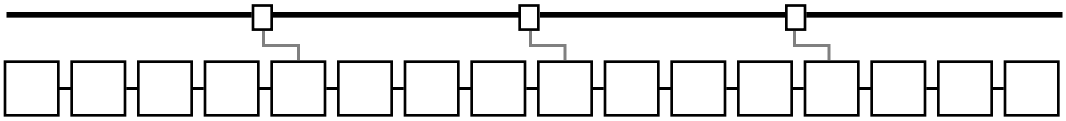


Fig. 12.9. Linear array with a separable bus using reconfiguration switches.

Semigroup computation: O(log p) steps

2D mesh with separable row/column buses

Reconfigurable mesh architecture

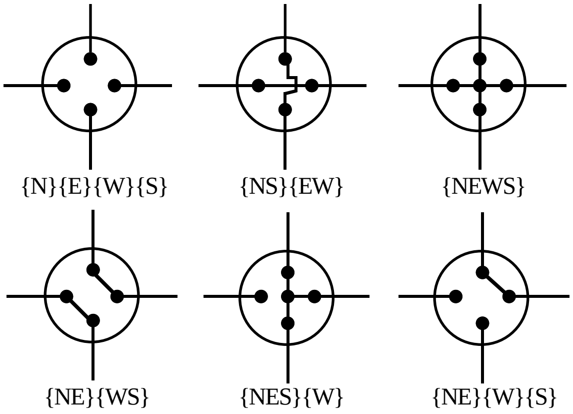


Fig. 12.10. Some processor states in a reconfigurable mesh.

12.5 Pyramid and Multigrid Systems

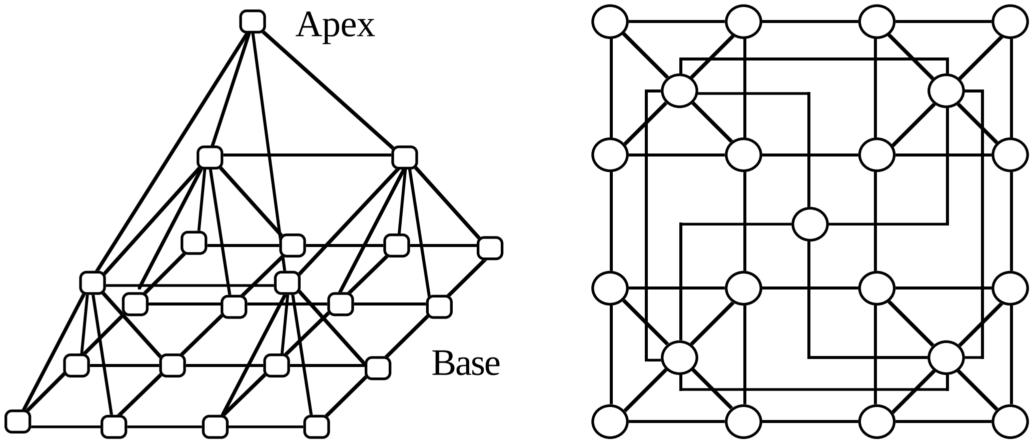


Fig. 12.11. Pyramid with 3 levels and 4 × 4 base along with its 2D layout.

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

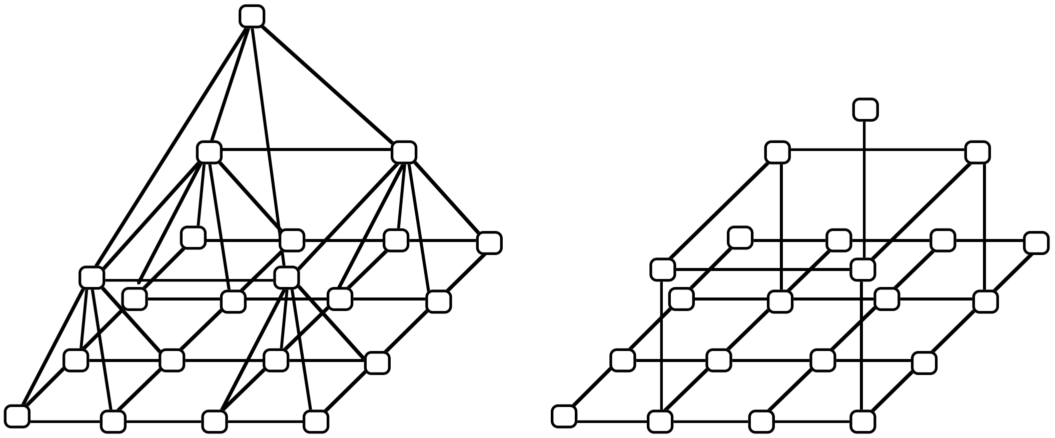


Fig. 12.12. The relationship between pyramid and 2D multigrid architectures.

12.6 Meshes of Trees

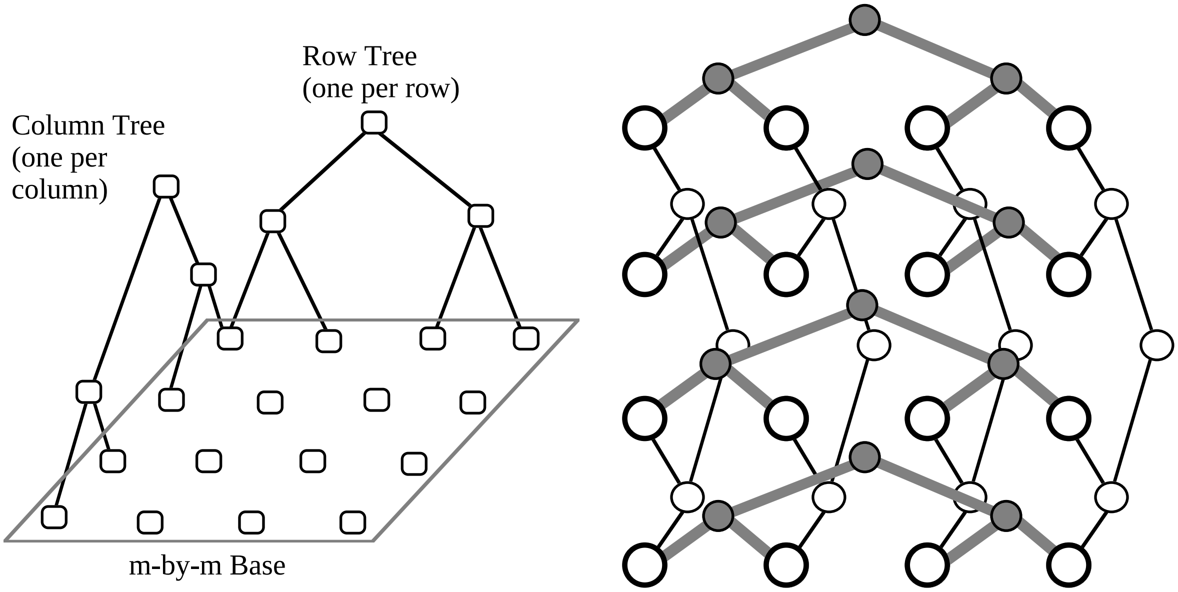
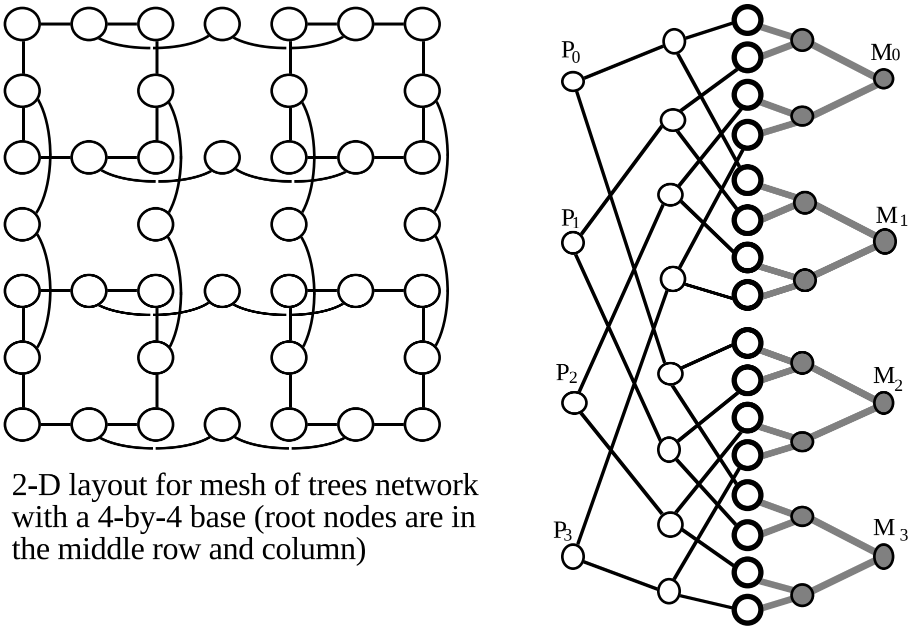


Fig. 12.13. Mesh of trees architecture with 3 levels and a 4 × 4 base.



2-D layout for mesh of trees network with a 4-by-4 base (root nodes are in the middle row and column)

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 × 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 the destination nodes are all distinct

Sorting m^2 keys, one per processor on the m × 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×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 × 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 $\sum_{i=0}^{m-1} a_{ij}\, x_j = y_i$)

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

Convolution of two vectors

Assume the mesh of trees with an m × (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×(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

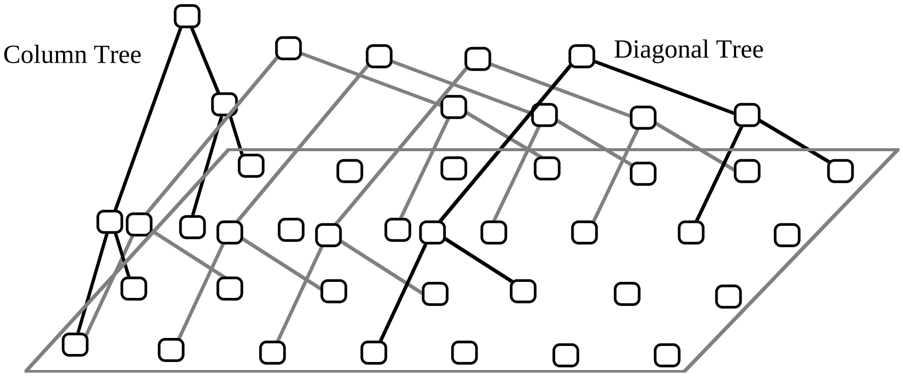


Fig. 12.15. Mesh of trees variant with row, column, and diagonal trees.

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 all edge weights are unique so that there is always a single minimum-weight edge among any subset

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 minimum-weight edge that connects the component to another one

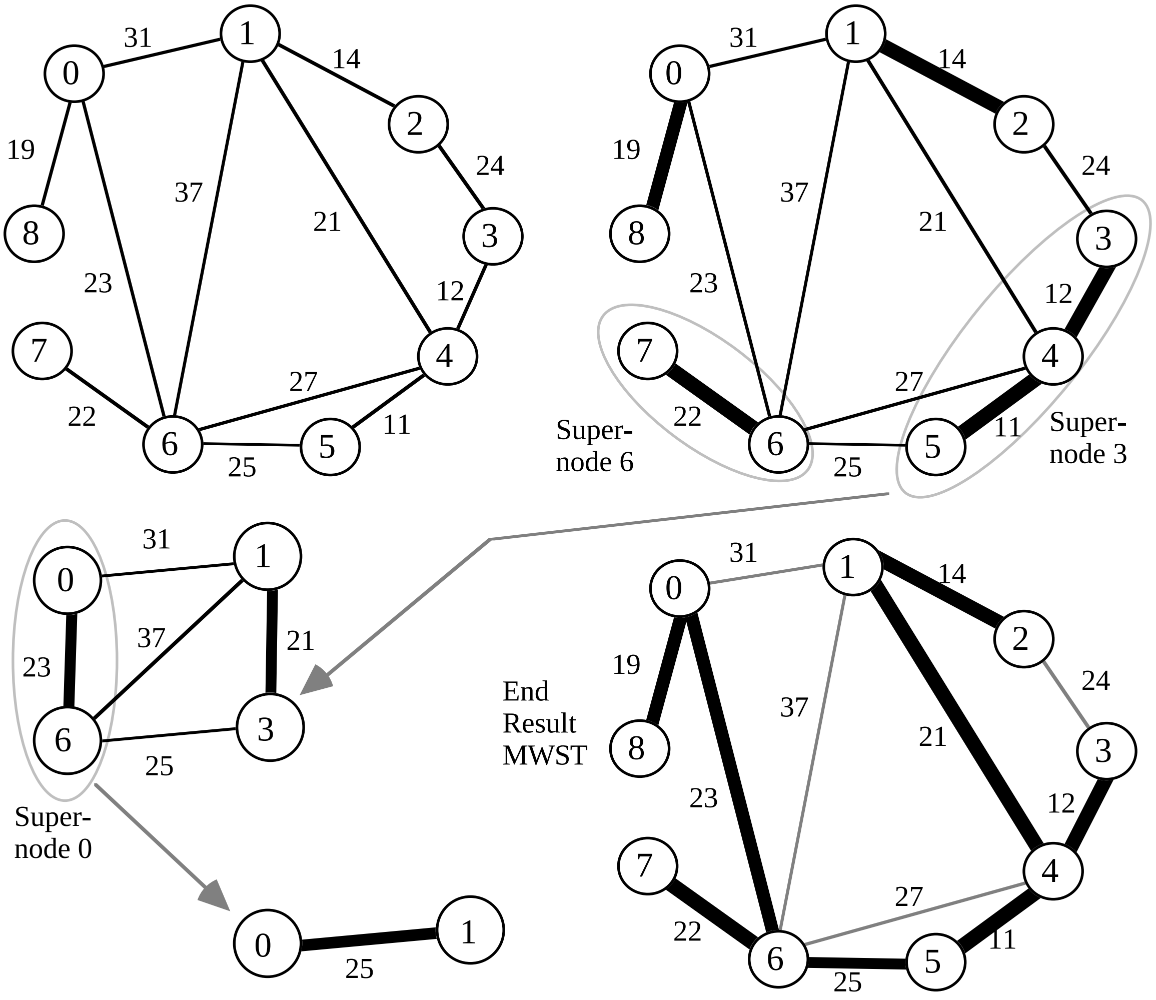


Fig. 12.16. Example for the minimal-weight spanning tree algorithm.

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 algorihms and their time complexities:

Kruskal's: O(e log e) ⇒ O(n^2 log n) for dense graphs

Prim's (binary heap): O((e + n)log n) ⇒ O(n^2 log n)

Prim's (Fibonacci heap): O(e + n log n) ⇒ O(n^2)

Thus, our best parallel solution offers a speedup of O(n^2/log^{2}n); sublinear in the number p = O(n^2) of processors

The key part of the simple parallel version of the greedy algorithm is showing that each phase can be done in 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

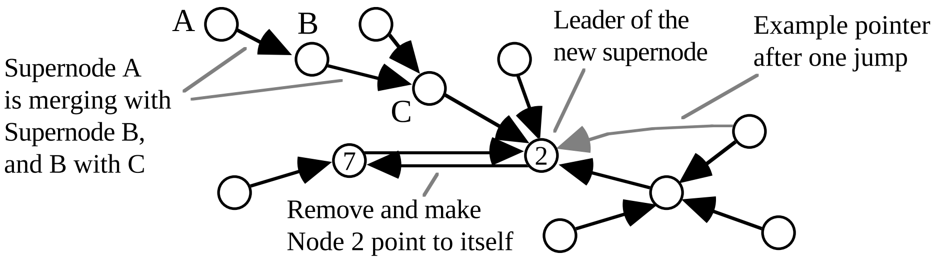

A
B
C
Supernode A is merging with Supernode B, and B with C
Leader of the new supernode
Example pointer after one jump
7
2
Remove and make Node 2 point to itself

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

Part IV Low-Diameter Architectures

Part Goals

- Study the hypercube as an example of architectures with
  - low (logarithmic) diameter
  - wide bisection
  - rich theoretical properties
- Present hypercube derivatives/alternatives that mitigate its realizability and scalability problems
- Complete our view of the "sea of interconnection networks"

Part Contents

13 Hypercubes and Their Algorithms

Chapter Goals

- Introduce the hypercube and its topological and algorithmic properties
- Design simple hypercube algorithms (sorting and 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

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

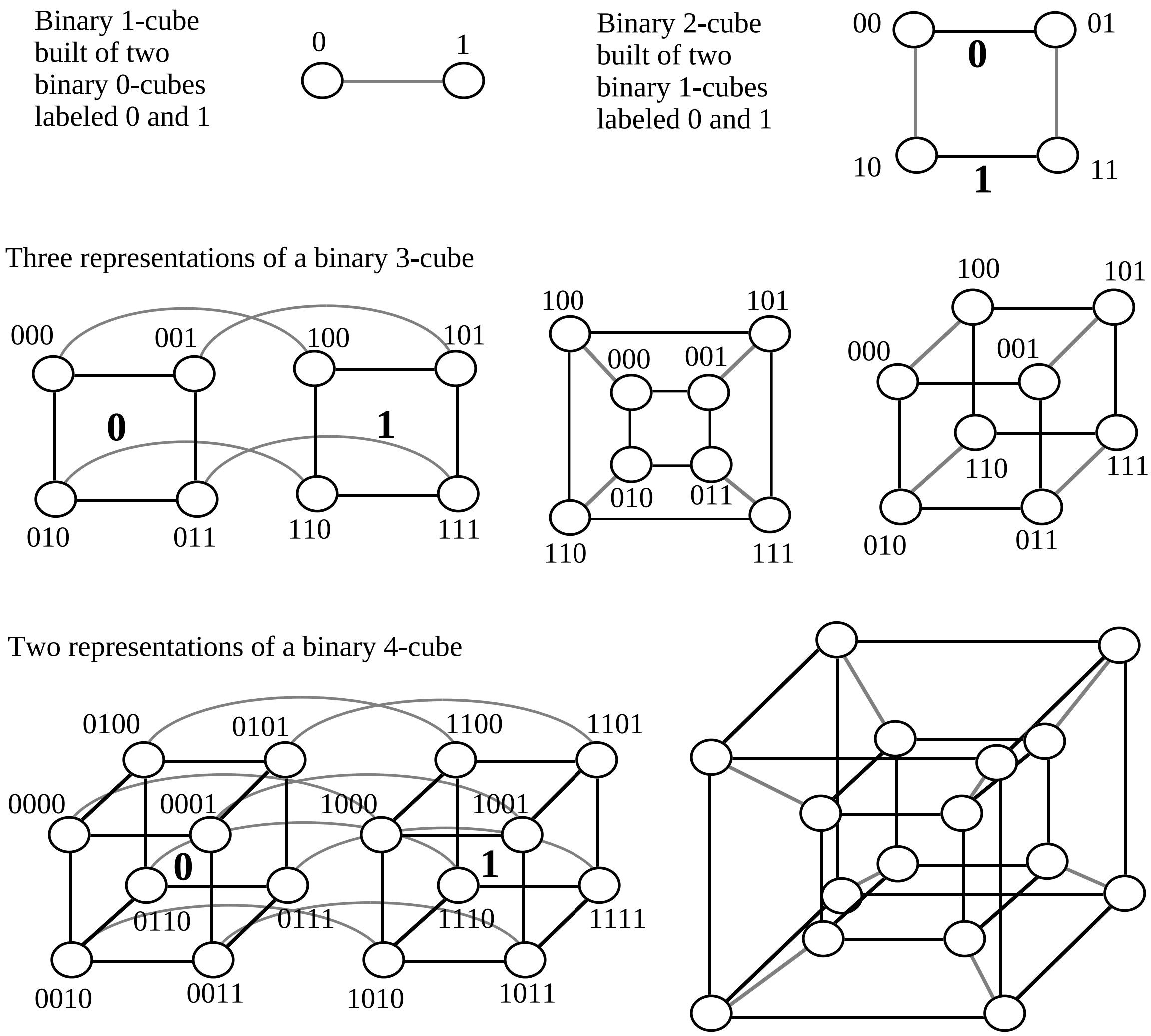

Binary 1-cube built of two binary 0-cubes labeled 0 and 1
0
1
Binary 2-cube built of two binary 1-cubes labeled 0 and 1
00
01
0
10
11
1
Three representations of a binary 3-cube
000
001
100
101
0
1
010
011
110
111
100
101
000
001
010
011
110
111
100
101
000
001
110
111
010
011
Two representations of a binary 4-cube
0100
0101
1100
1101
0000
0001
1000
1001
0
1
0110
0111
1110
1111
0010
0011
1010
1011

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: $d = q = \log_2 p$

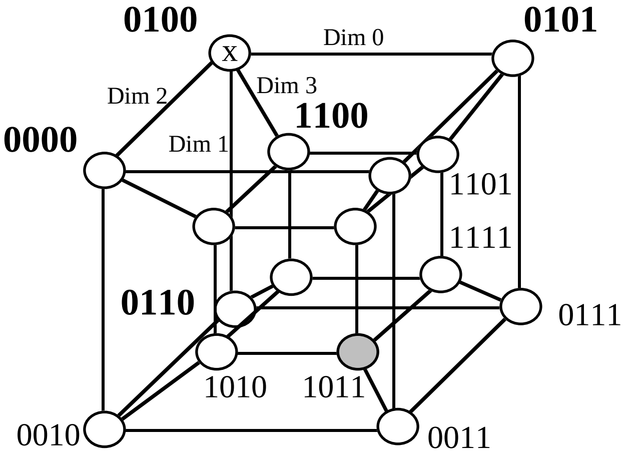
The q neighbors of node x with binary ID $x_{q-1}x_{q-2} \cdots x_2x_1x_0$:

$x_{q-1}x_{q-2} \cdot\cdot\cdot x_2x_1 \bar{x}_0$ dimension-0 neighbor; $N_0(x)$

$x_{q-1}x_{q-2} \cdot\cdot\cdot x_2 \bar{x}_1x_0$ dimension-1 neighbor; $N_1(x)$

. . .

$\bar{x}_{q-1}x_{q-2} \cdot\cdot\cdot x_2x_1x_0$ dimension-(q – 1) neighbor; $N_{q-1}(x)$



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

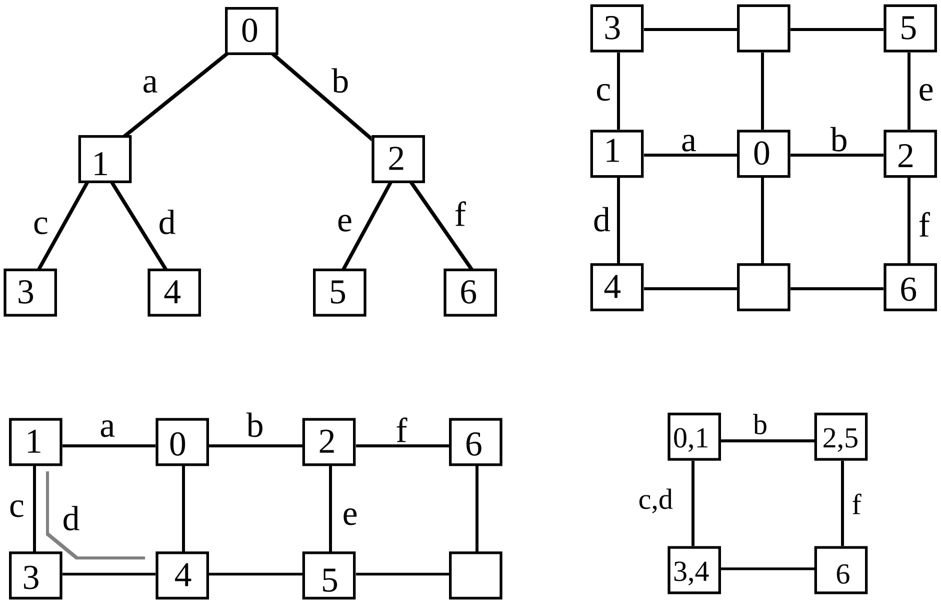


Fig. 13.2. Embedding a seven-node binary tree into 2D meshes of various sizes.

	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

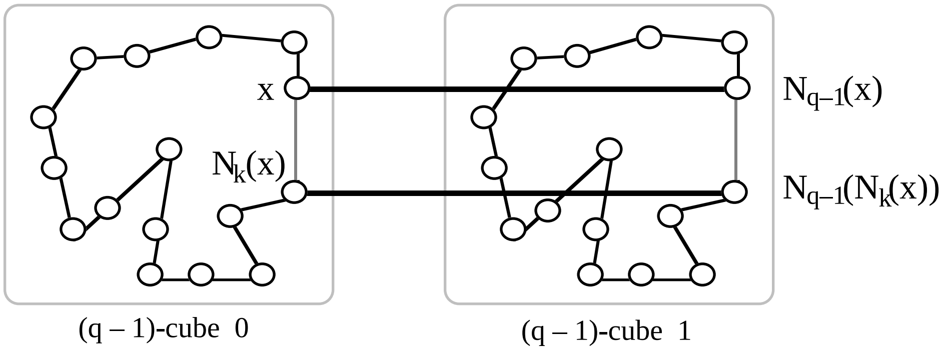


Fig. 13.3. Hamiltonian cycle in the q-cube.

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, and expansion 1

The proof is based on the notion of cross-product graphs, which we first define

Given k graphs $G_i = (V_i, E_i)$, $1 \le i \le k$, their (cross-)product graph $G = G_1 \times G_2 \times \cdots \times G_k = (V, E)$ has:

node set $V = \{(v_1, v_2, \cdots, v_k) \mid v_i \in V_i, 1 \le i \le k\}$

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 $i \ne j$, $u_i = v_i\}$

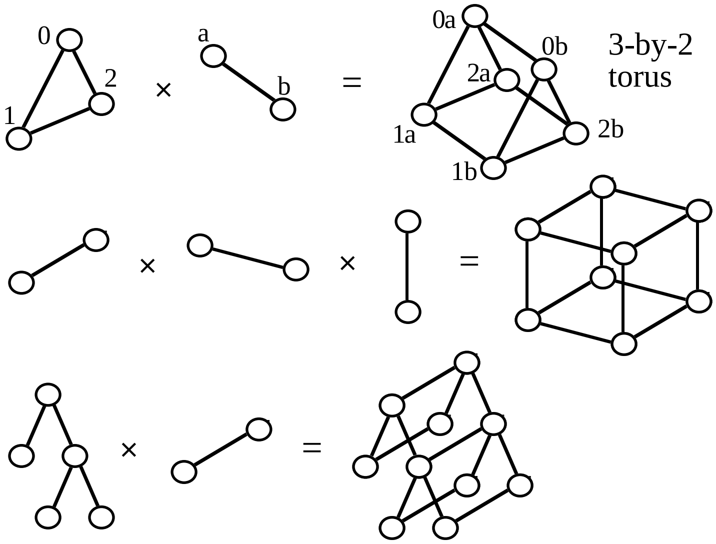


Fig. 13.4. Examples of product graphs.

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 $2^{m_0}, 2^{m_1}, \cdots, 2^{m_{h-1}}$

b. The $(m_0 + m_1 + \cdots + m_{h-1})$-cube is the product of an m_0-cube, an m_1-cube, $\cdots$, 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

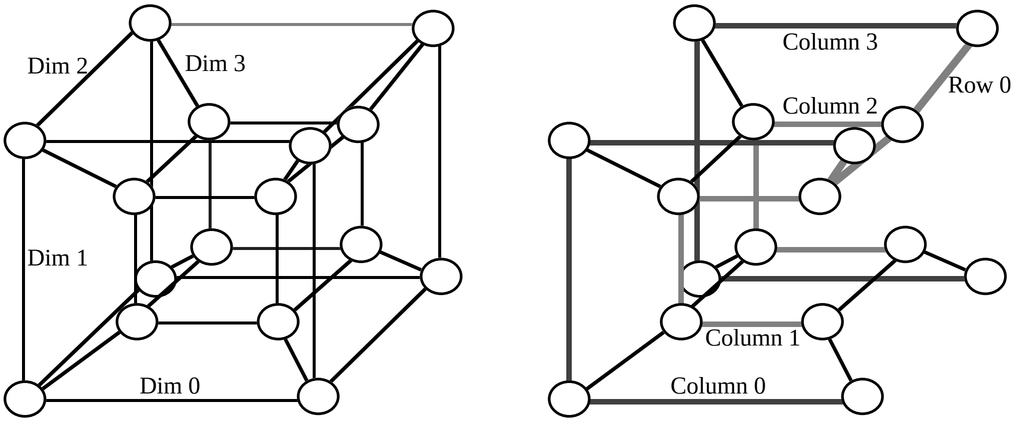
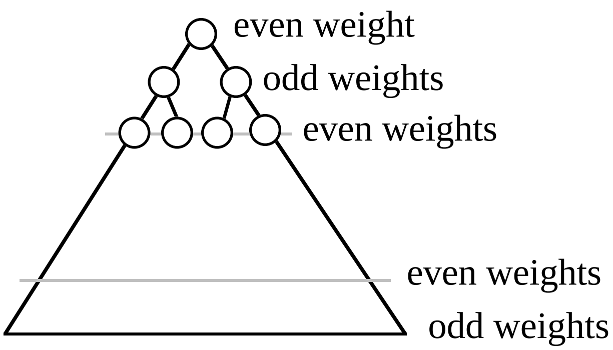


Fig. 13.5. The 4 × 4 mesh/torus is a subgraph of the 4-cube.

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

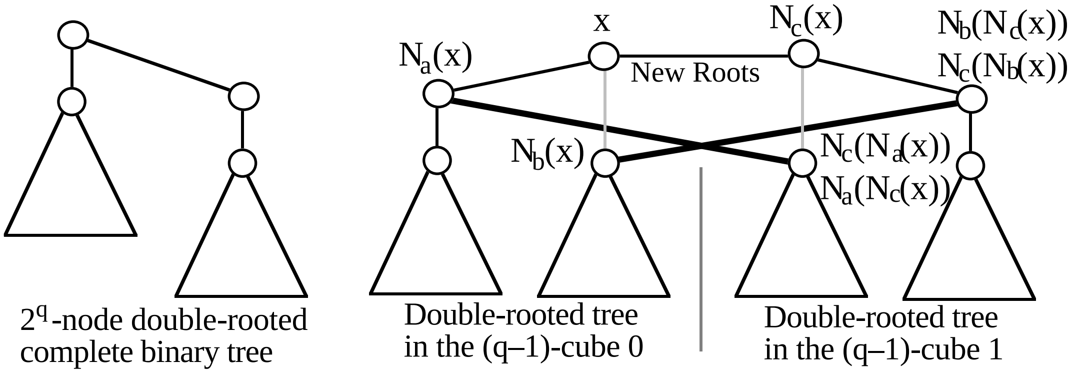

x
$N_c(x)$
$N_b(N_c(x))$
$N_c(N_b(x))$
$N_a(x)$
New Roots
$N_b(x)$
$N_c(N_a(x))$
$N_a(N_c(x))$
2^q-node double-rooted complete binary tree
Double-rooted tree in the (q–1)-cube 0
Double-rooted tree in the (q–1)-cube 1

Fig. 13.6. The 2^q-node double-rooted complete binary tree is a subgraph of the q-cube.

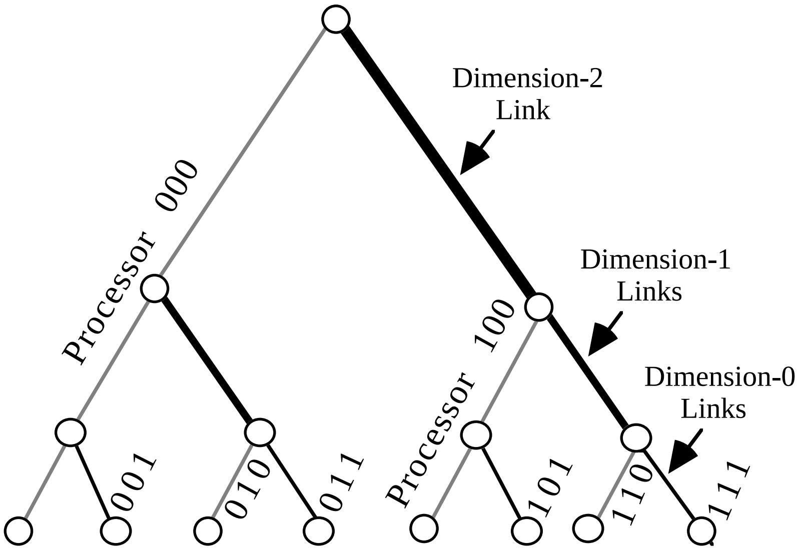


Fig. 13.7. Embedding a 15-node complete binary tree into the 3-cube.

13.4 A Few Simple Algorithms

Semigroup computation on the q-cube

```
Processor x, 0 ≤ x < p do t[x] := v[x] {initialize “total” to own value}
for k = 0 to q – 1 Processor x, 0 ≤ x < p, do
      get y :=t[$N_k$(x)]
      set t[x] := t[x] ⊗ y
endfor
```

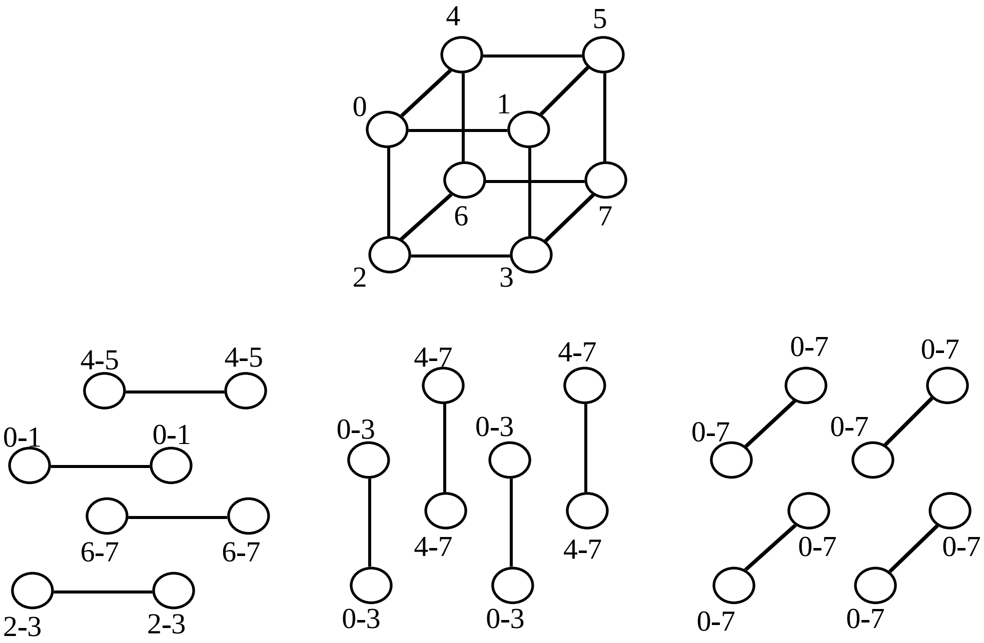


Fig. 13.8. Semigroup computation on a 3-cube.

Parallel prefix computation on the q-cube

```
Processor x, $0 \le x < p$, dot[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 u[x] := u[x] $\otimes$ y
endfor
```

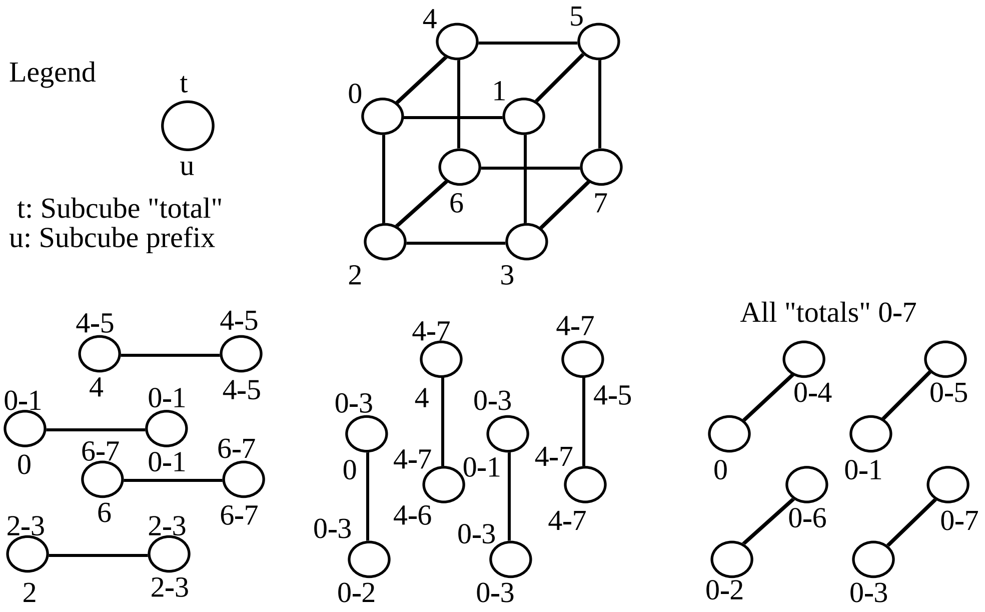


Fig. 13.9. Parallel prefix computation on a 3-cube.

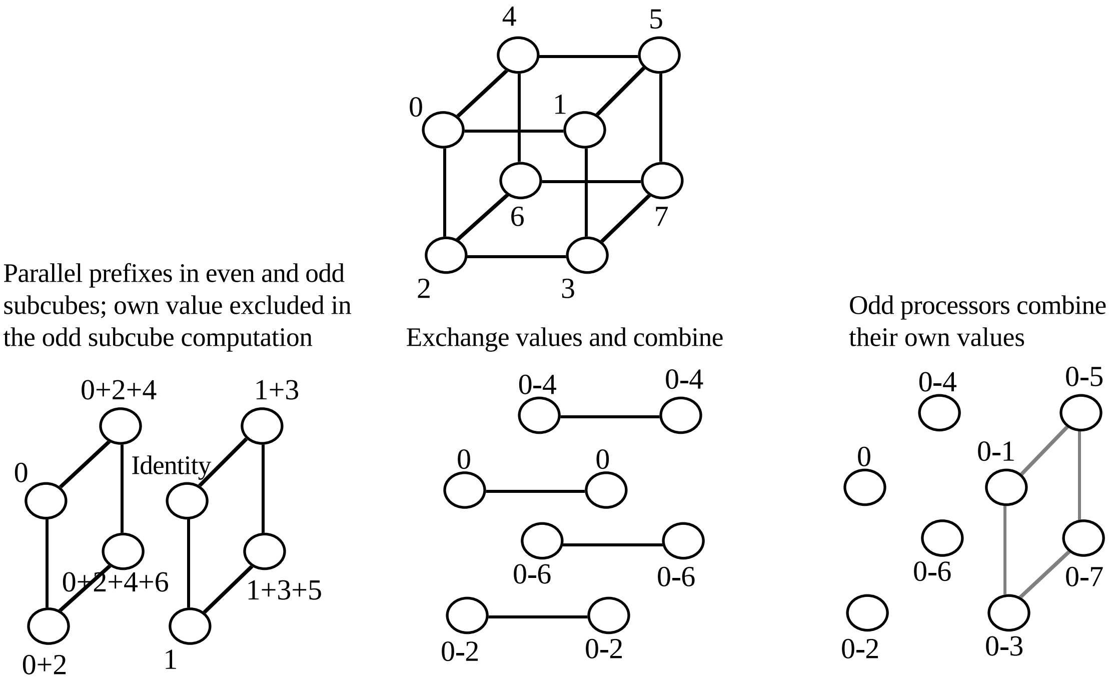

4
5
0
1
6
7
2
3
Parallel prefixes in even and odd subcubes; own value excluded in the odd subcube computation
Exchange values and combine
Odd processors combine their own values
0+2+4
1+3
0
Identity
0+2+4+6
1+3+5
0+2
1
0-4
0-4
0
0
0-6
0-6
0-2
0-2
0-4
0-5
0
0-1
0-6
0-7
0-2
0-3

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 ≤ x < p, do
    get y :=v[N_k(x)]
    set v[x] := y
endfor
```

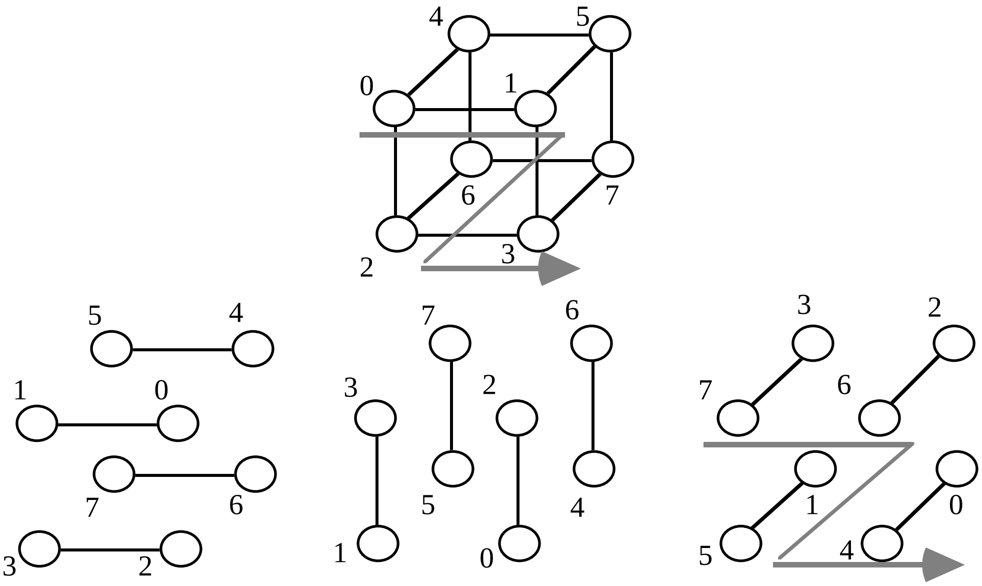
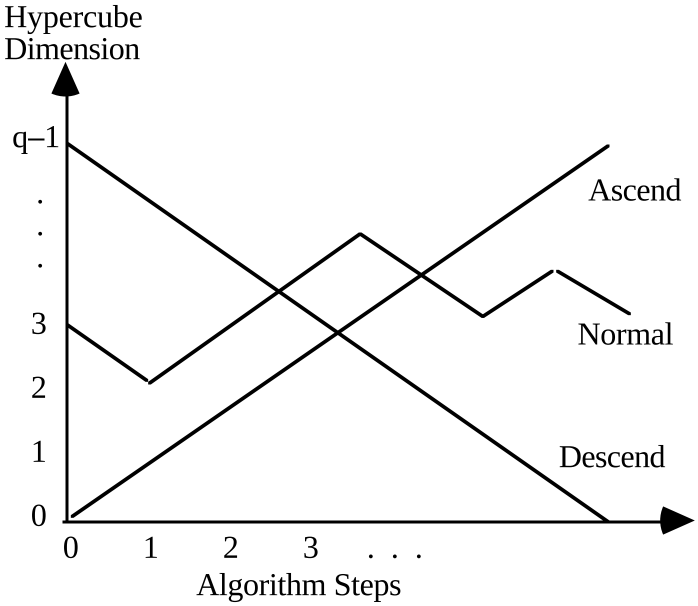


Fig. 13.11. Sequence reversal on a 3-cube.



13.5 Matrix Multiplication

Multiplying m × 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 × 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
```

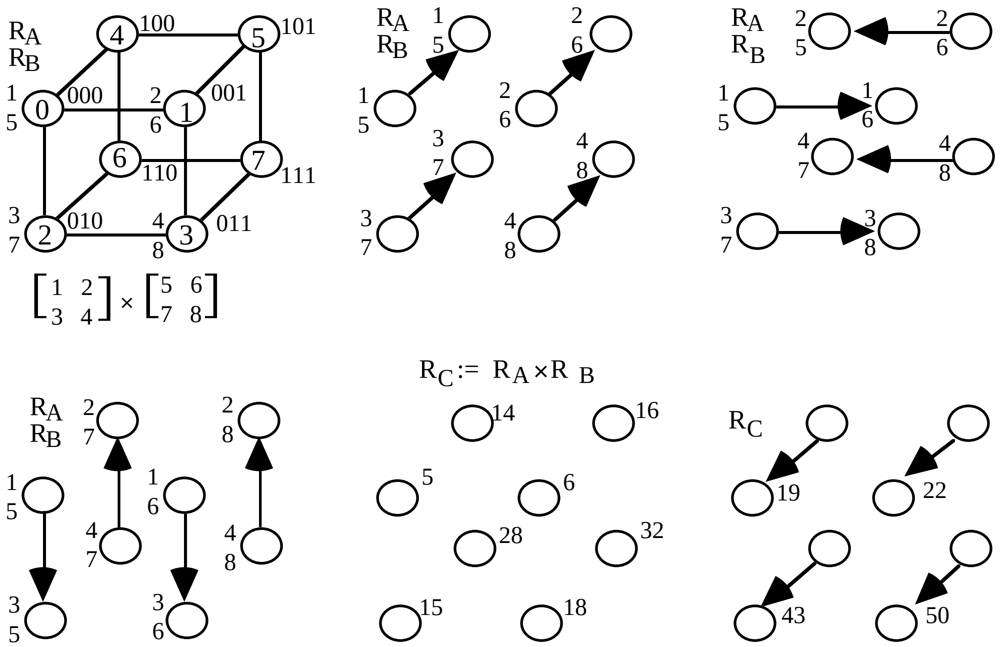


Fig. 13.12. Multiplying two 2 × 2 matrices on a 3-cube.

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

$$T_{\mathrm{mul}}(m, p) = \underbrace{m^2/p^{2/3} \times O(\log p)}_{\text{Communication}} + \underbrace{2m^3/p}_{\text{Computation}}$$

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 characterized 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

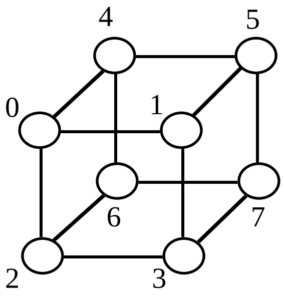
Chapter Goals

- 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

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 running time for all n and p

1-1 sorting (n = p)

Batcher's bitonic sort: $O(\log^2 n) = O(\log^2 p)$ time

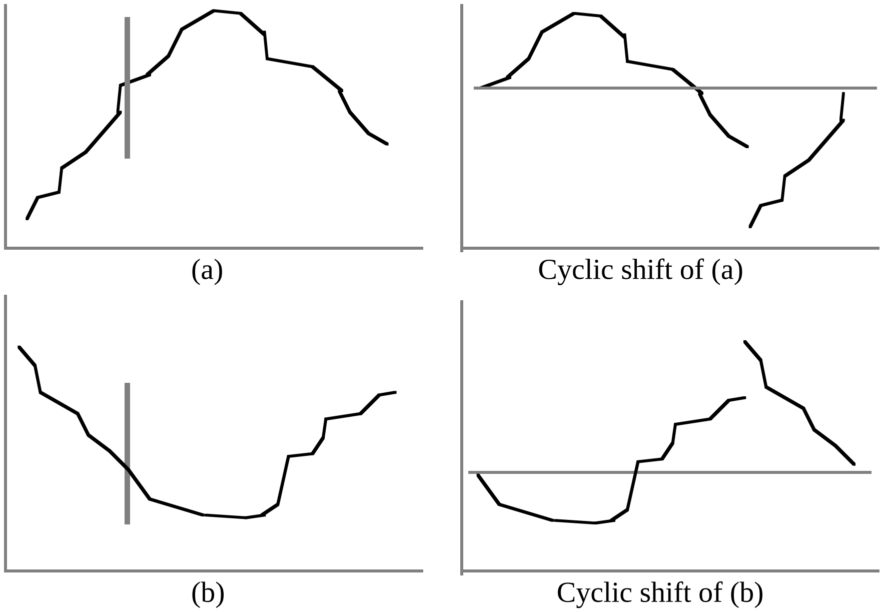
Same for Batcher's odd-even merge sort

$O(\log n)$-time deterministic algorithm not known

k-k sorting (n = pk)

Optimal algorithms known for n >> p or when

average running time is considered (randomized)



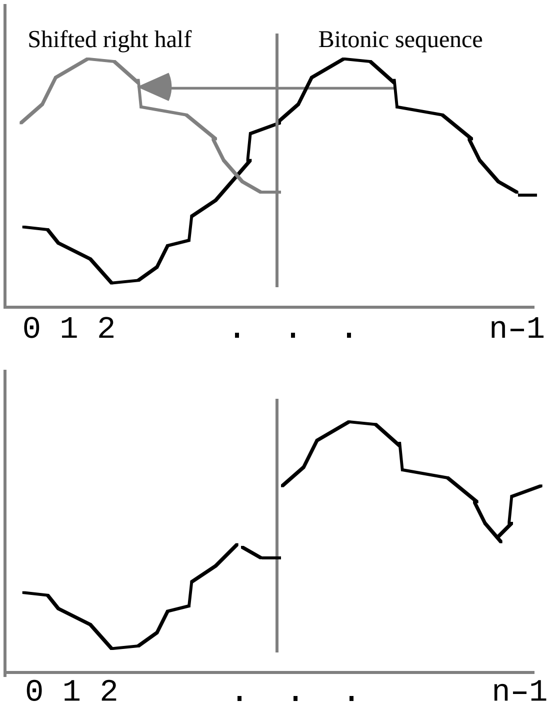
(a)

Cyclic shift of (a)

(b)

Cyclic shift of (b)

Fig. 14.1. Examples of bitonic sequences.



Shift right half of data to left half

Keep smaller value of each pair and ship the larger value to right

Each half is a bitonic sequence that can be sorted independently

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

```
5   9  10  15   3   7  14  12   8   1   4  13  16  11   6   2

---->   <----   ---->   <----   ---->   <----   ---->   <----

5   9  15  10   3   7  14  12   1   8  13   4  11  16   6   2

 ------------>   <------------   ------------>   <------------

5   9  10  15  14  12   7   3   1   4   8  13  16  11   6   2

 ---------------------------->   <----------------------------

3   5   7   9  10  12  14  15  16  13  11   8   6   4   2   1

 ------------------------------------------------------------>

1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
```

Fig. 14.3. Sorting an arbitrary sequence on a linear array through recursive application of bitonic sorting.

$$
\begin{aligned}
T(p) &= T(p/2) + B(p) \\
&= T(p/2) + 2p - 2 \;=\; 4p - 4 - 2\log_2 p
\end{aligned}
$$

Alternate derivation for the running time of bitonic sorting on a linear array:

$$
\begin{aligned}
T(p) &= B(2) + B(4) + \cdots + B(p) \\
&= 2 + 6 + 14 + \cdots + (2p - 2) \;=\; 4p - 4 - 2\log_2 p
\end{aligned}
$$

For a linear array of processors, the bitonic sorting algorithm is clearly inferior to the simpler odd-even transposition sort which requires only p compare-exchange steps or 2p unidirectional communication steps

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

$$T(q) = T(q - 1) + B(q) = T(q - 1) + q$$

$$= q(q + 1)/2 = \log_2 p\ (\log_2 p + 1)/2$$

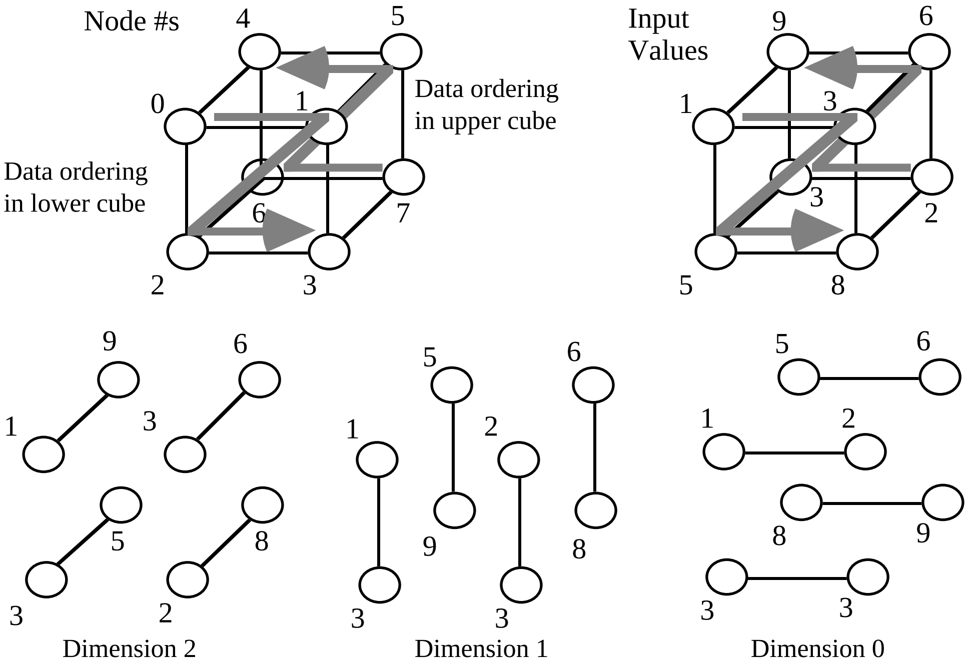


Fig. 14.4. Sorting a bitonic sequence of size 8 on the 3-cube.

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 such a way that the route taken by one message does not significantly overlap or conflict with those of other messages

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

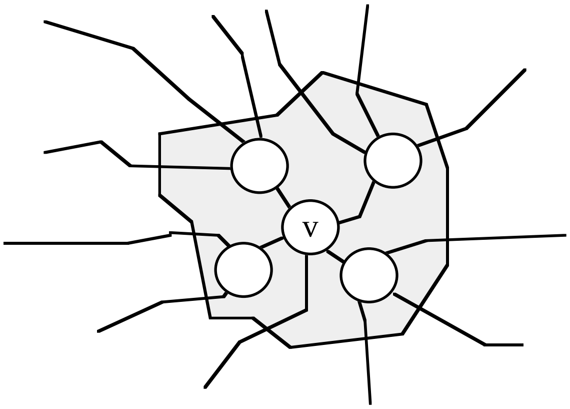
Our strategy is to find k node pairs u_i, v_i $(1 \le i \le k)$ such that

$u_i \ne u_j$ and $v_i \ne v_j$ for $i \ne j$, and

P_{u_i,v_i} all pass through the same edge e

Since at most 2 packets can go through a link in each 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: $\underline{0}1011011$, $1101\underline{1}011$, $11010\underline{0}11$,
$\overline{1}101011\underline{1}$, $1101\overline{0}110$

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

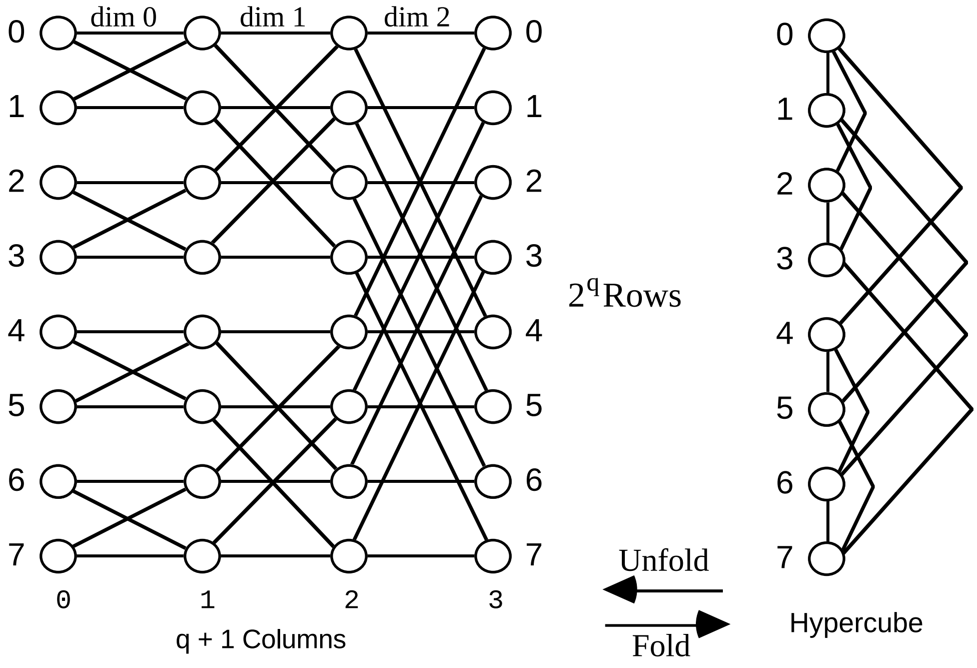


Fig. 14.5. Unfolded 3-cube or the 32-node butterfly network.

Self-routing in a butterfly

Fom 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)

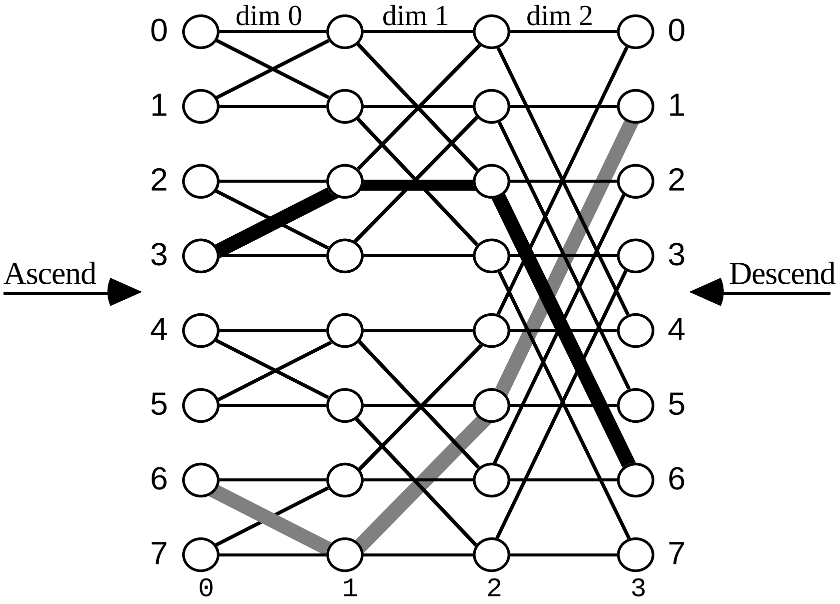


Fig. 14.6. Example dimension-order routing paths.

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

There are also “bad” routing problems that lead to maximum conflicts and thus the worst-case running time predicted by Theorem 14.1

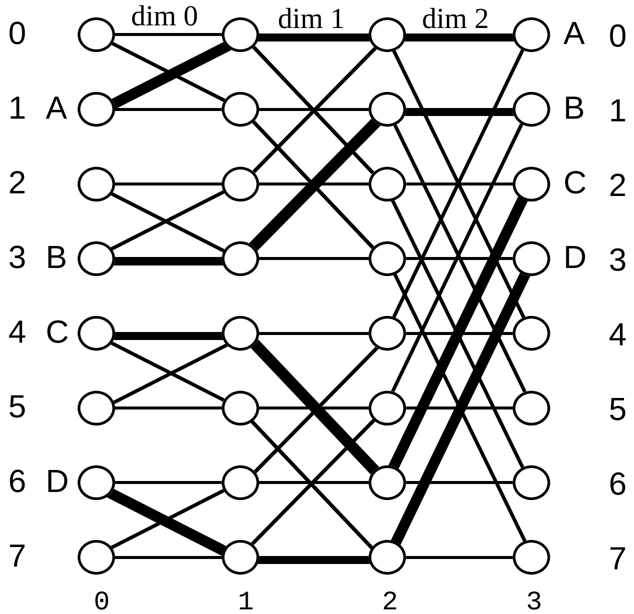


Fig. 14.7. **Packing is a “good” routing problem for dimension-order routing on the hypercube.**

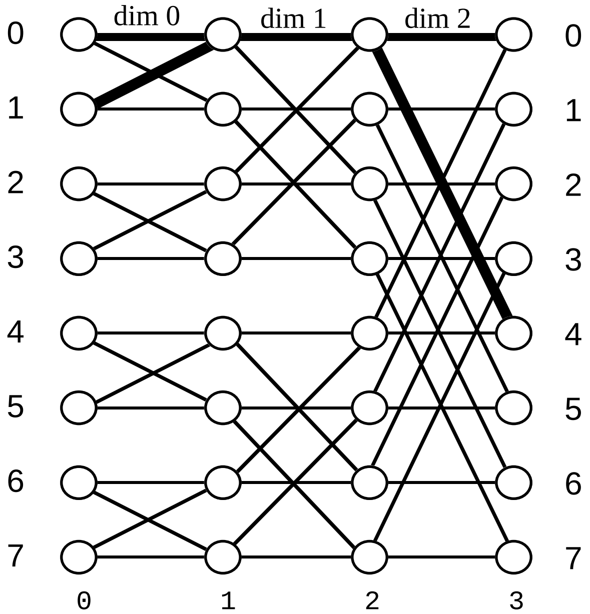


Fig. 14.8. **Bit-reversal permutation is a “bad” routing problem for dimension-order routing on the hypercube.**

Message buffer needs of dimension-order routing

One may think that if we limit each node to a small, constant number of message buffers, then the above bound still holds, except that messages will be queued at several levels before reaching node 0

However, queuing the messages at multiple intermediate nodes may introduce additional delays that we have not accounted for, so that even the $\Theta(\sqrt{p})$ running time can no longer be 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. Hence, the average running time of the dimension-order routing algorithm is very close to its best case and its message buffer requirements are quite modest.

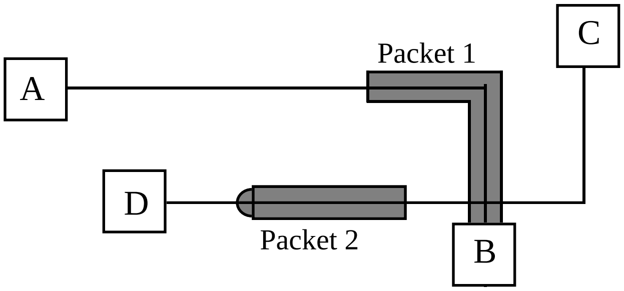
Besides, if we anticipate any (near) worst-case routing pattern to occur in a given application, two options are available to us:

Compute the routing paths off-line and store in tables

Use randomized routing to convert the worst-case to average-case performance

The probabilistic analyses required to show the good average-case performance of dimension-order routing are quite 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, e.g., the four worms carrying messages A, B, C, D, will move with no conflict among them. Each message is thus 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, since each 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 within nodes, 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 discussed in 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

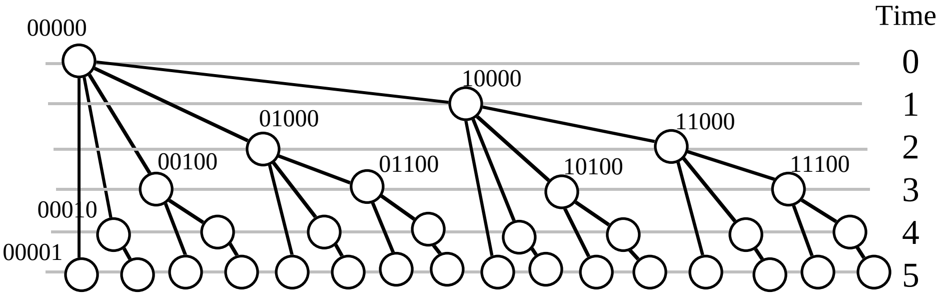
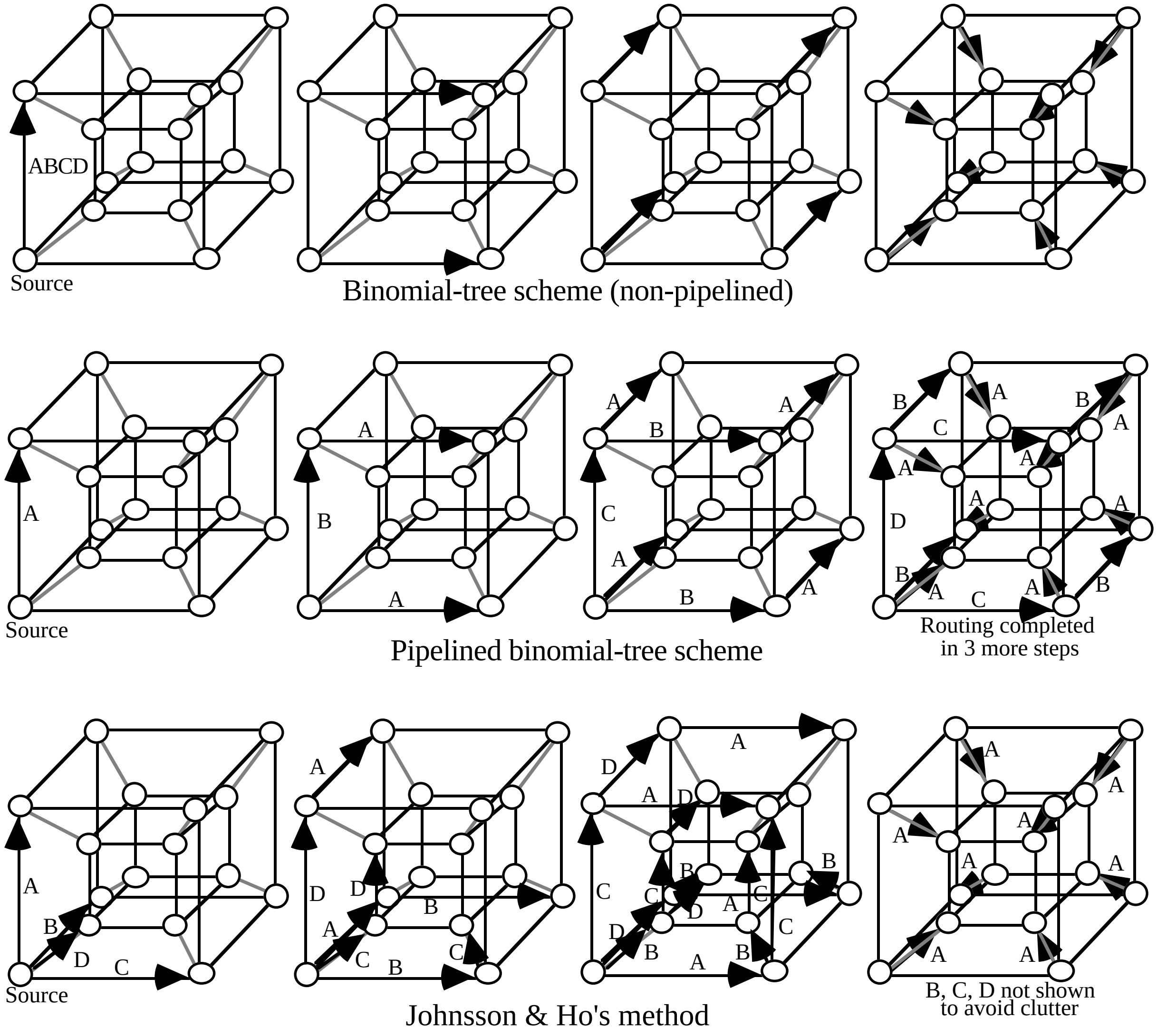


Fig. 14.9. The binomial broadcast tree for a 5-cube.



Binomial-tree scheme (non-pipelined)

Pipelined binomial-tree scheme

Johnsson & Ho's method

Fig. 14.10. Three hypercube broadcasting schemes as performed on a 4-cube.

14.6 Adaptive and Fault-Tolerant Routing

Because there are up to q node-disjoint and edge-disjoint shortest paths between any node pairs in a q-cube, 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

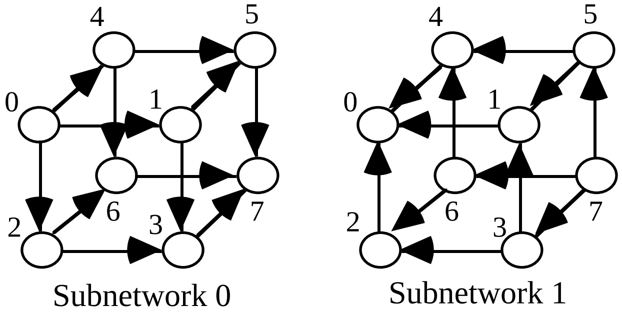


Fig. 14.11. Partitioning a 3-cube into subnetworks for deadlock-free routing.

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 q – 1 or fewer faults and at most q + 2 with 2q – 3 or fewer faults [Lati93]

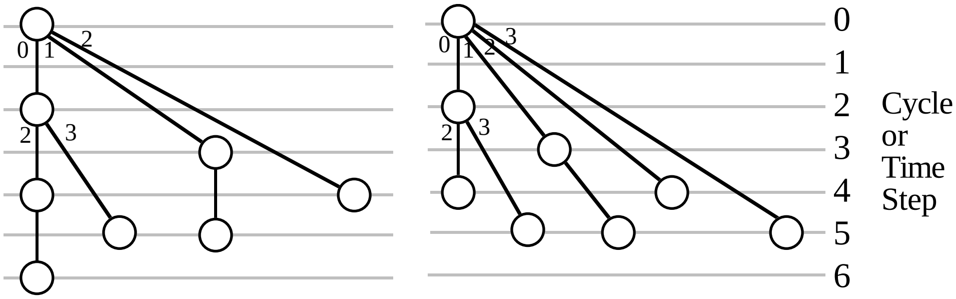


Figure for Problem 14.15.

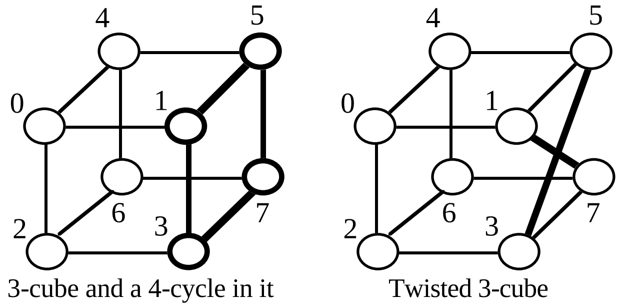
15 Other Hypercubic Architectures

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

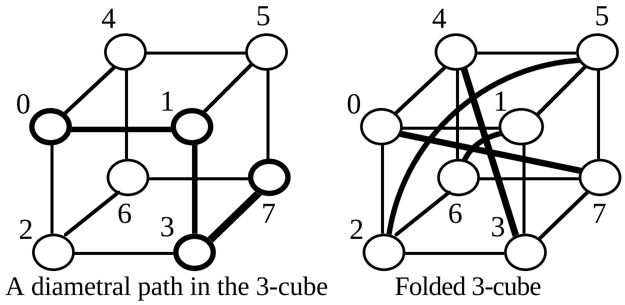
15.1 Modified and Generalized Hypercubes



3-cube and a 4-cycle in it

Twisted 3-cube

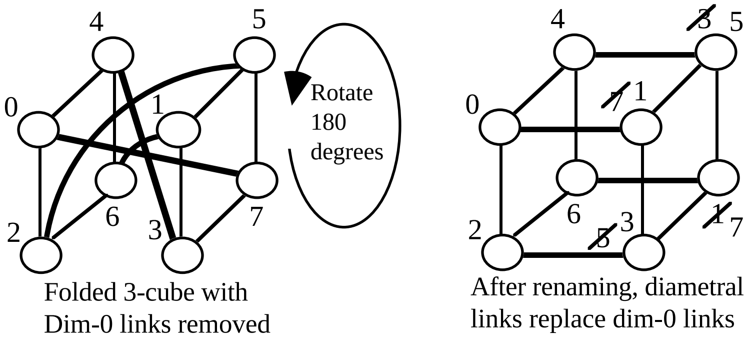
Fig. 15.1. Deriving a twisted 3-cube by redirecting two links in a 4-cycle.



A diametral path in the 3-cube

Folded 3-cube

Fig. 15.2. Deriving a folded 3-cube by adding four diametral links.

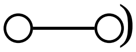


Folded 3-cube with Dim-0 links removed

After renaming, diametral links replace dim-0 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 = ()^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

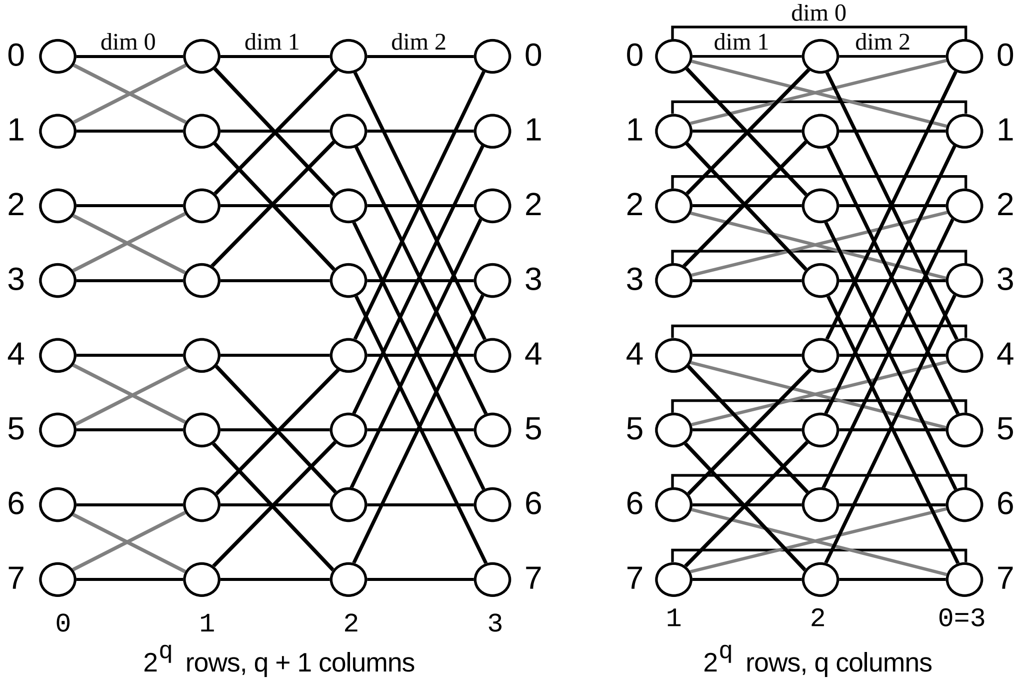
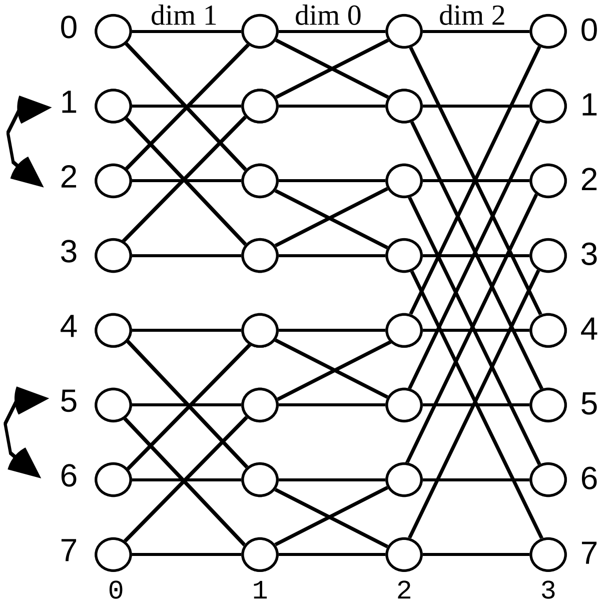


Fig. 15.4. Butterfly and wrapped butterfly networks.



Switching these two row pairs converts this to the original butterfly network. Changing the order of stages in a butterfly is thus equivalent to a relabeling of the rows (in this example, row xyz becomes row xzy).

Fig. 15.5. Butterfly network with permuted dimensions.

Fat trees eliminate the bisection bottleneck of an "skinny" tree by making the bandwidth of links correspondingly higher near the root

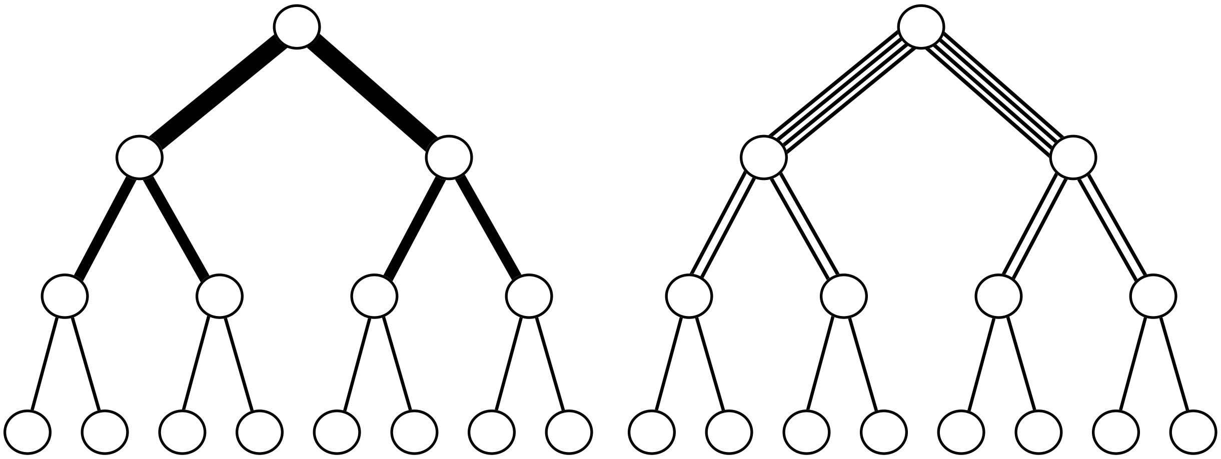


Fig. 15.6. Two representations of a fat tree.

One way of realizing a fat tree

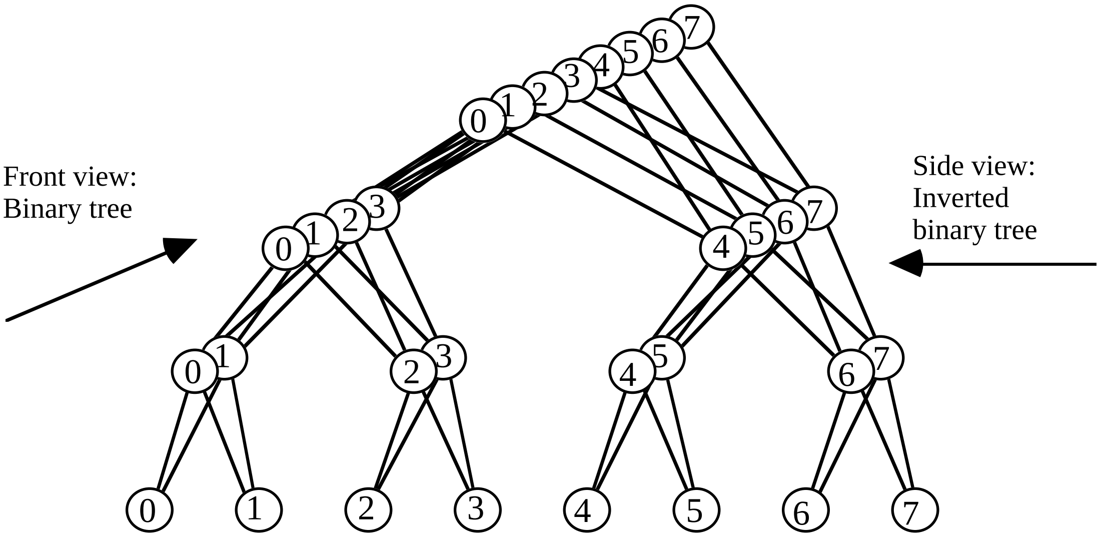


Fig. 15.7. Butterfly network redrawn as a fat tree.

Butterfly as a multistage interconnection network

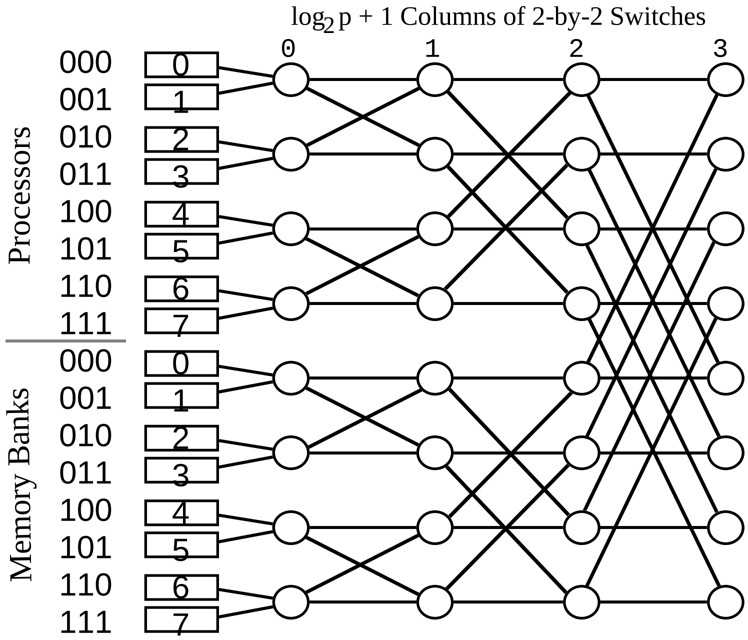


Fig. 15.8. Butterfly network used to connect modules that are on the same side.

Generalization of the butterfly network

High-radix or m-ary butterfly (built of m × m switches)

Has m^q rows and q + 1 columns (q if wrapped)

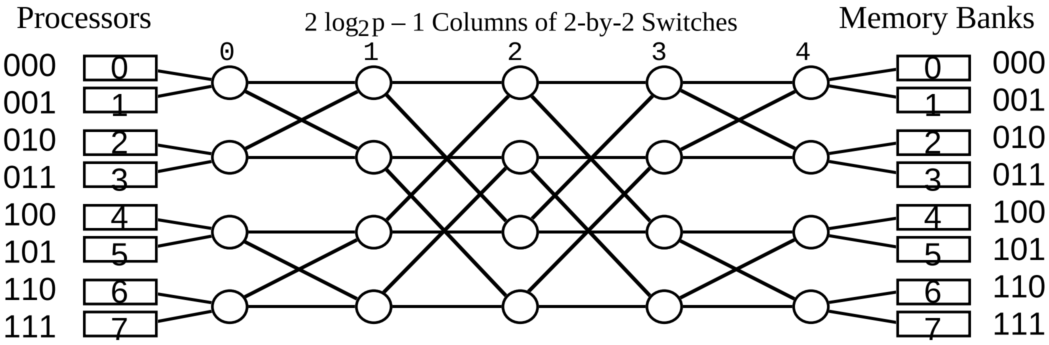


Fig. 15.9. Benes˘ network formed from two back-to-back butterflies.

Benes˘ network can route any permutation

(it is rearrangeable)

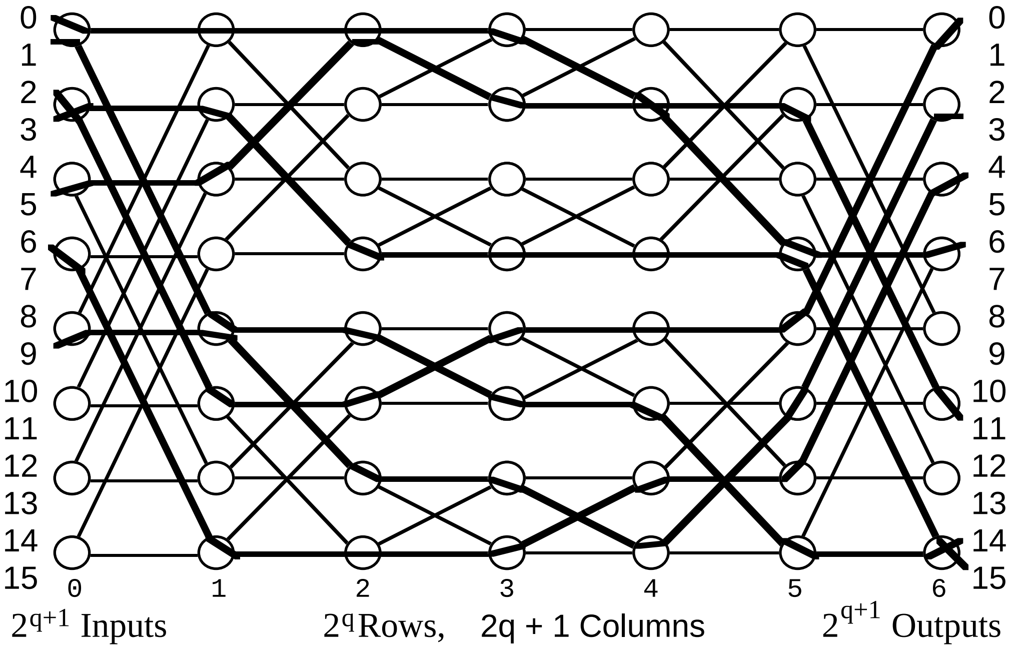


Fig. 15.10. Another example of a Benes˘ network.

15.3 Plus-or-Minus-2^i Network

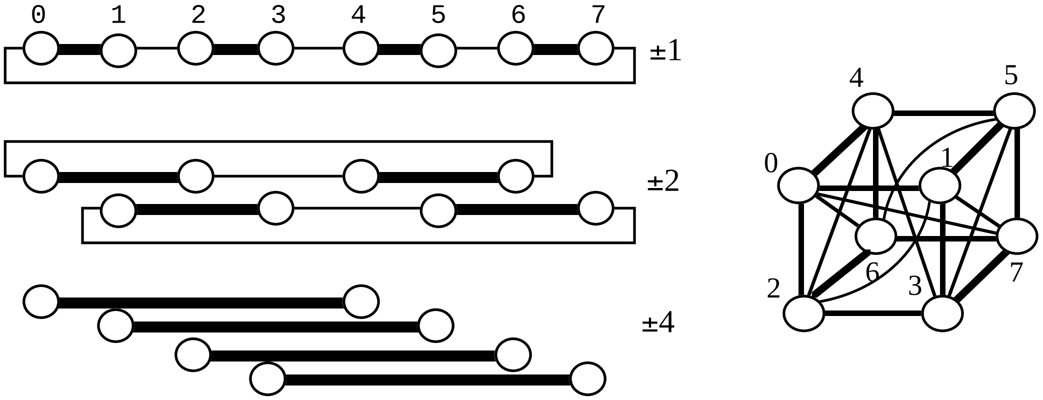


Fig. 15.11. Two representations of the eight-node plus-or-minus-2^i network.

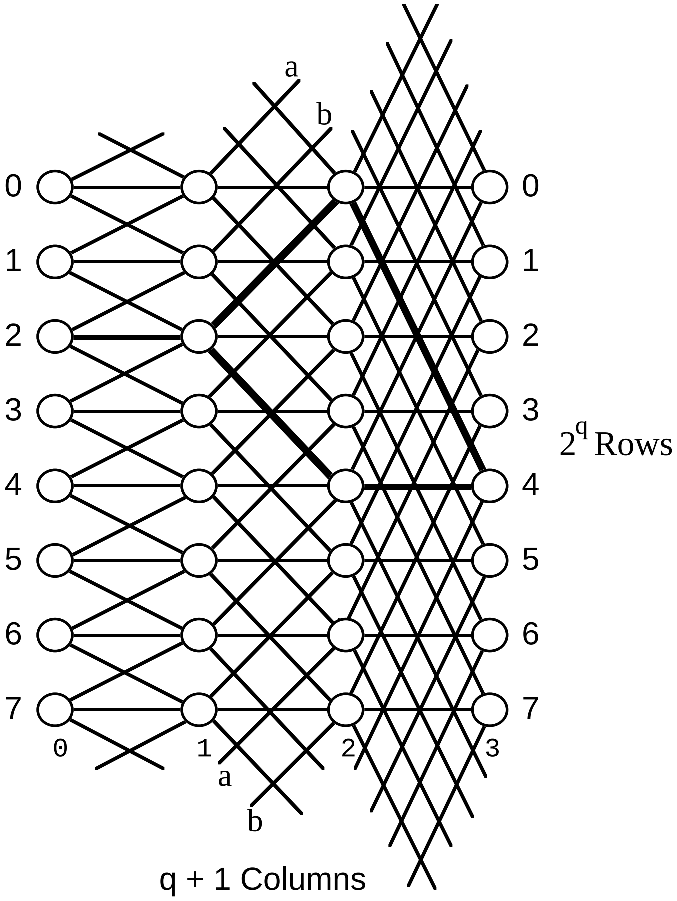


Fig. 15.12. Augmented data manipulator network.

15.4 The Cube-Connected Cycles Network

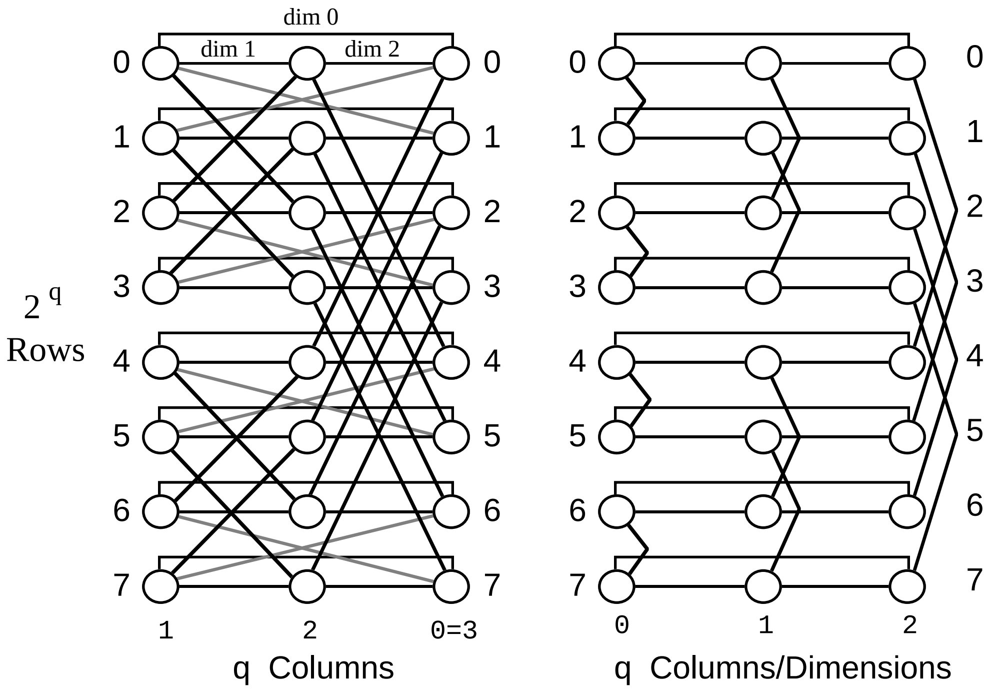


Fig. 15.13. A wrapped butterfly (left) converted into cube-connected cycles.

How CCC was originally defined:

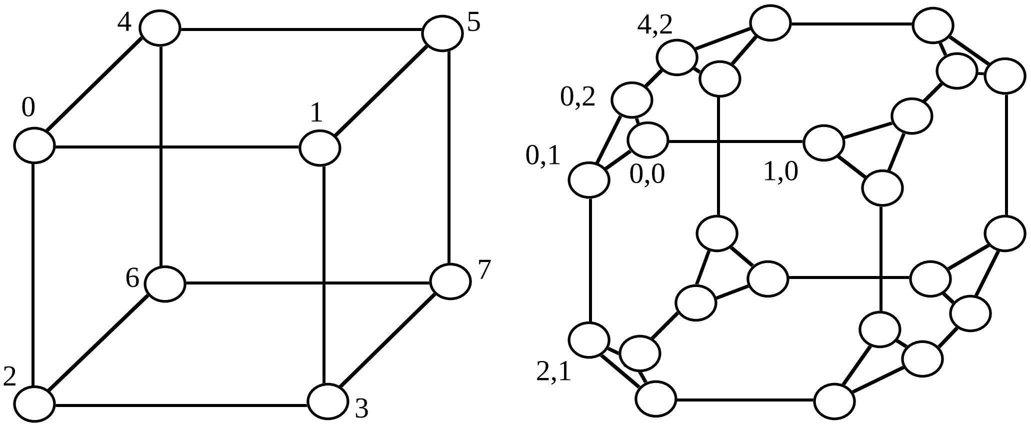


Fig. 15.14. Alternate derivation of CCC from a hypercube.

Emulating normal hypercube algorithms on CCC

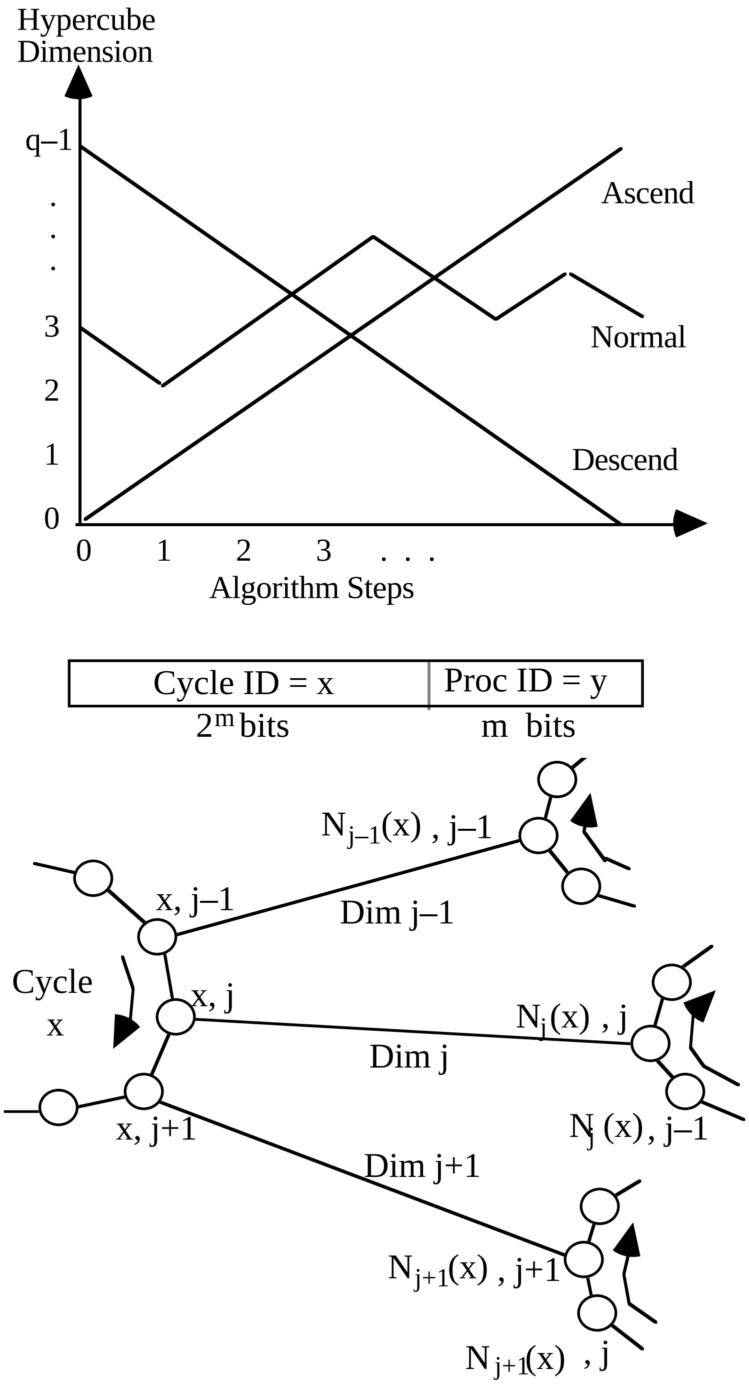


Fig. 15.15. CCC emulating a normal hypercube algorithm.

15.5 Shuffle and Shuffle-Exchange Networks

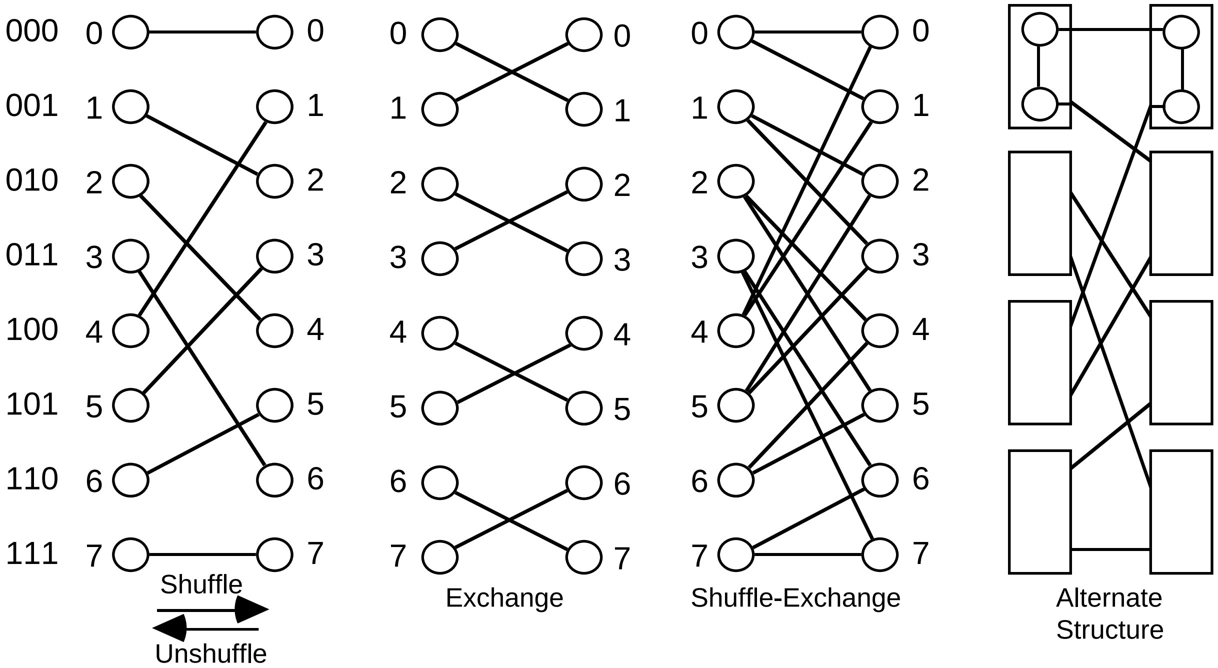


Fig. 15.16. Shuffle, exchange, and shuffle–exchange connectivities.

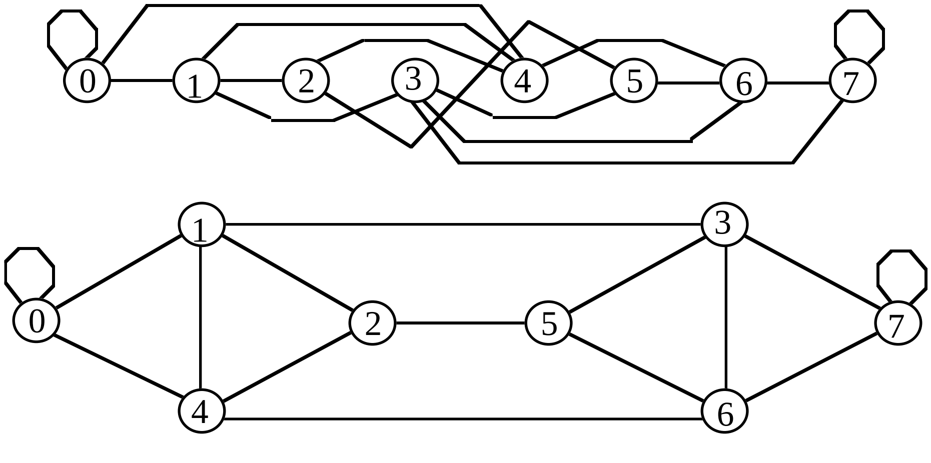


Fig. 15.17. Alternate views of an eight-node shuffle–exchange network.

In a 2^q-node shuffle network, node $x = x_{q-1}x_{q-2} \cdot\cdot\cdot x_2x_1x_0$ is connected to $x_{q-2} \cdot\cdot\cdot x_2x_1x_0x_{q-1}$ (cyclic left-shift of x)

In the shuffle-exchange network, node x is additionally connected to $x_{q-2} \cdot\cdot\cdot x_2x_1x_0 \, \bar{x}_{q-1}$

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: $D = q = \log_2 p$, $d = 4$

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

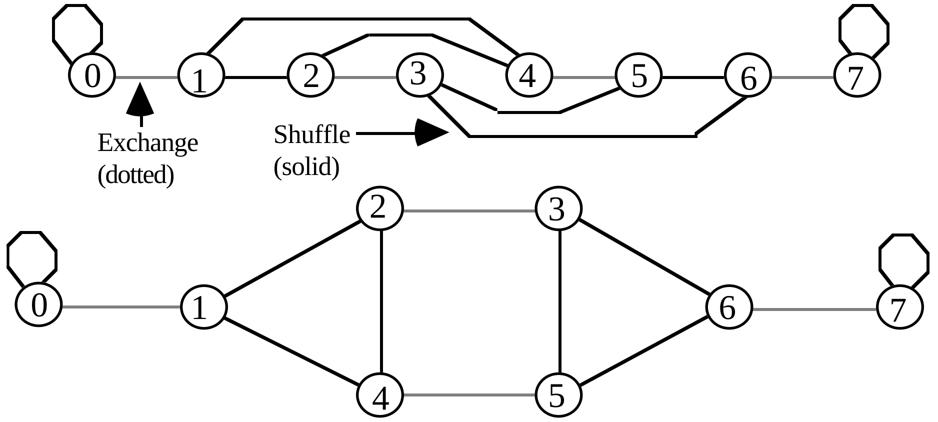


Fig. 15.18. Eight-node network with separate shuffle and exchange links.

Multistage shuffle-exchange network = butterfly network

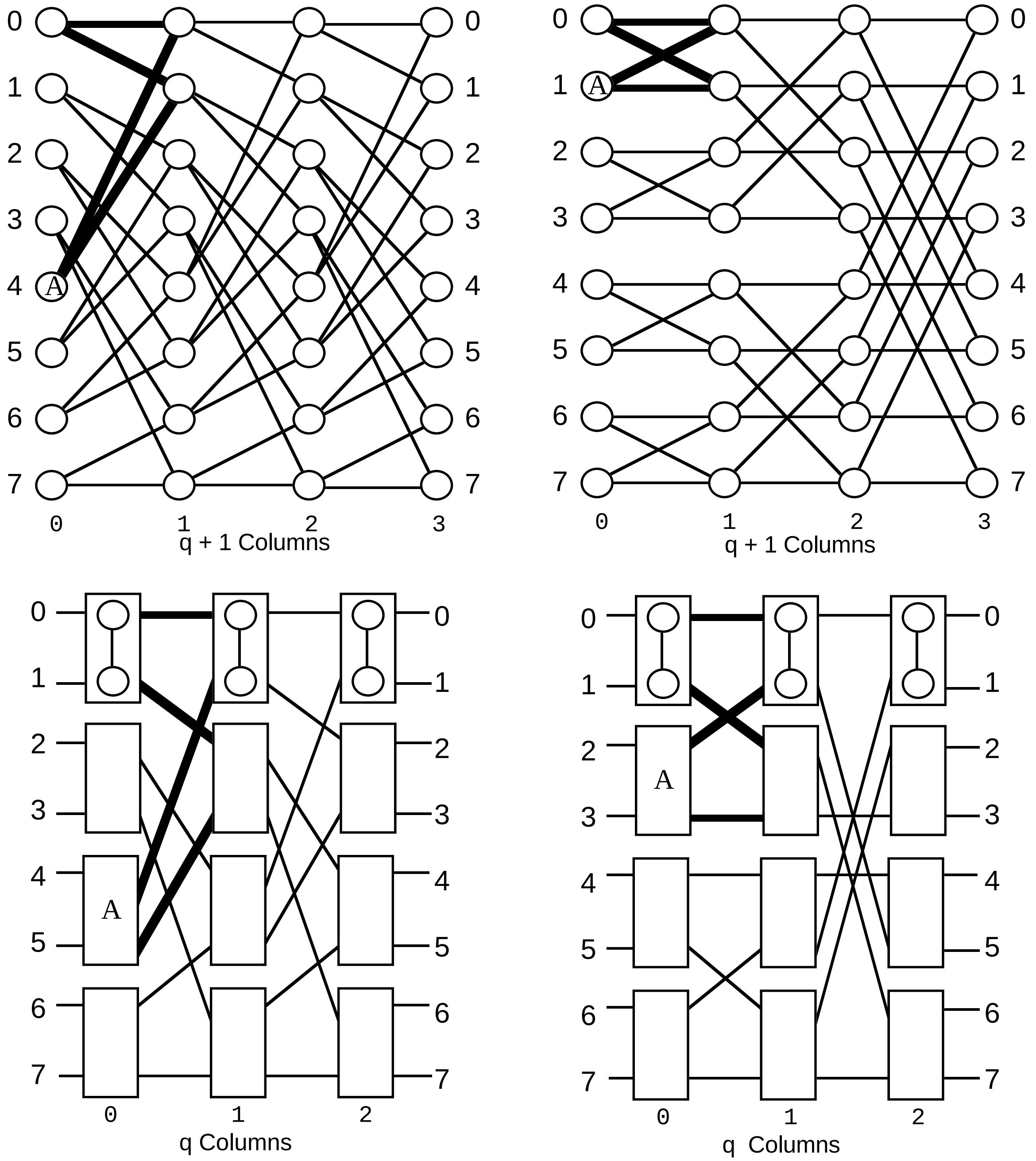


Fig. 15.19. Multistage shuffle–exchange network (omega network) is the same as butterfly network.

15.6 That's Not All, Folks!

When q is a power of 2, the 2^qq-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

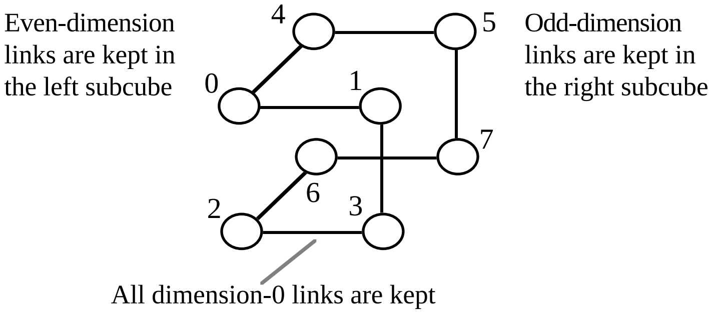

Even-dimension links are kept in the left subcube
Odd-dimension links are kept in the right subcube
4
5
0
1
7
6
2
3
All dimension-0 links are kept

Fig. 15.20. Example of a pruned hypercube.

Mobius 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 Mobius cube has a diameter of about 1/2 and an average inter-node distance of about 2/3 of that of a hypercube

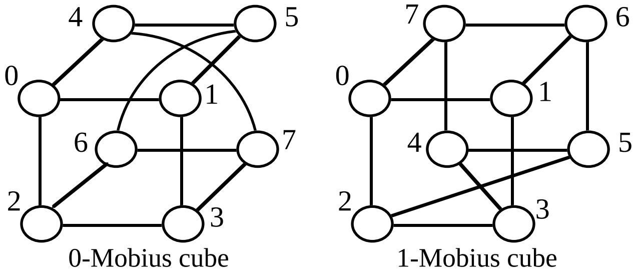


Fig. 15.21. Two 8-node Mobius cubes.

16 A Sampler of Other Networks

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