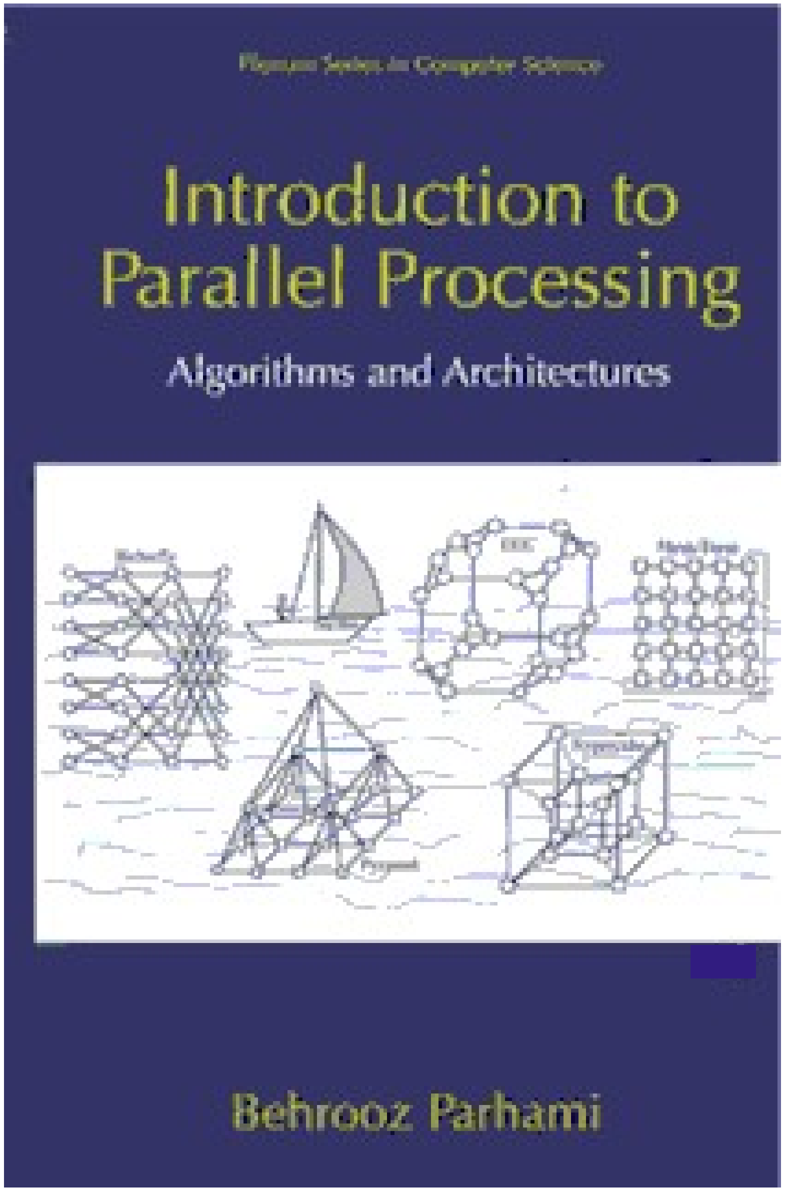


Part III
Mesh-Based Architectures

	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 Architectures
	Part IV: Low-Diameter Architectures	The Hypercube Architecture	13. Hypercubes and Their Algorithms 14. Sorting and Routing on Hypercubes
		Hypercubic and Other Networks	15. Other Hypercubic Architectures 16. A Sampler of Other Networks
	Part V: Some Broad Topics	Coordination and Data Access	17. Emulation and Scheduling 18. Data Storage, Input, and Output
		Robustness and Ease of Use	19. Reliable Parallel Processing 20. System and Software Issues
	Part VI: Implementation Aspects	Control-Parallel Systems	21. Shared-Memory MIMD Machines 22. Message-Passing MIMD Machines
		Data Parallelism and Conclusion	23. Data-Parallel SIMD Machines 24. Past, Present, and Future

About This Presentation

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

Edition	Released	Revised	Revised	Revised
First	Spring 2005	Spring 2006	Fall 2008	Fall 2010
		Winter 2013	Winter 2014	Winter 2016
		Winter 2019	Winter 2020	Winter 2021

III Mesh-Type Architectures

Study mesh, torus, and related interconnection schemes:

- Many modern parallel machines are mesh/torus-based
- Scalability and speed due to short, regular wiring
- Enhanced meshes, variants, and derivative networks

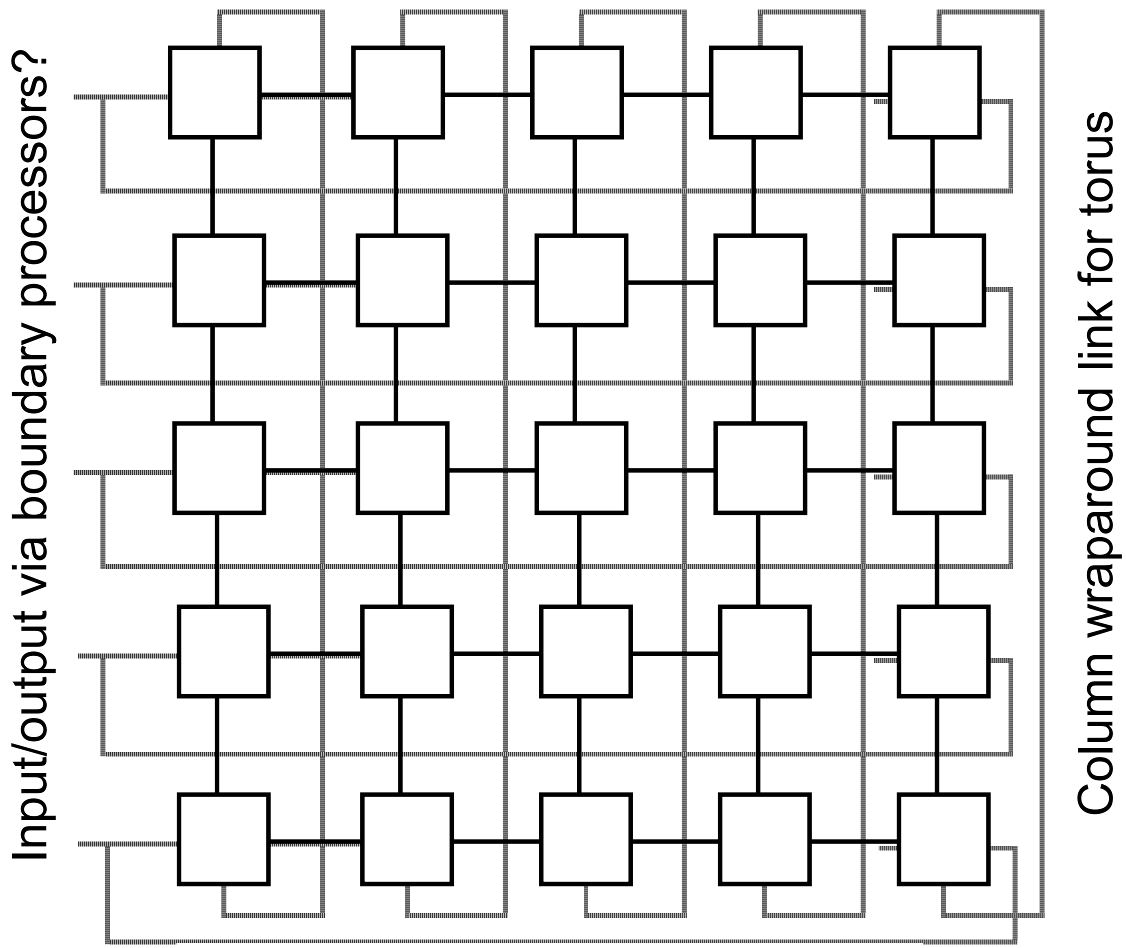
Topics in This Part	
Chapter 9	Sorting on a 2D Mesh or Torus
Chapter 10	Routing on a 2D Mesh or Torus
Chapter 11	Numerical 2D Mesh Algorithms
Chapter 12	Mesh-Related Architectures

9 Sorting on a 2D Mesh or Torus

Introduce the mesh model (processors, links, communication):
- Develop 2D mesh sorting algorithms
- Learn about strengths and weaknesses of 2D meshes

Topics in This Chapter
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



2D, four-neighbor (NEWS) mesh; other types in Chapter 12

Square $p^{1/2} \times p^{1/2}$ or rectangular $r \times p/r$

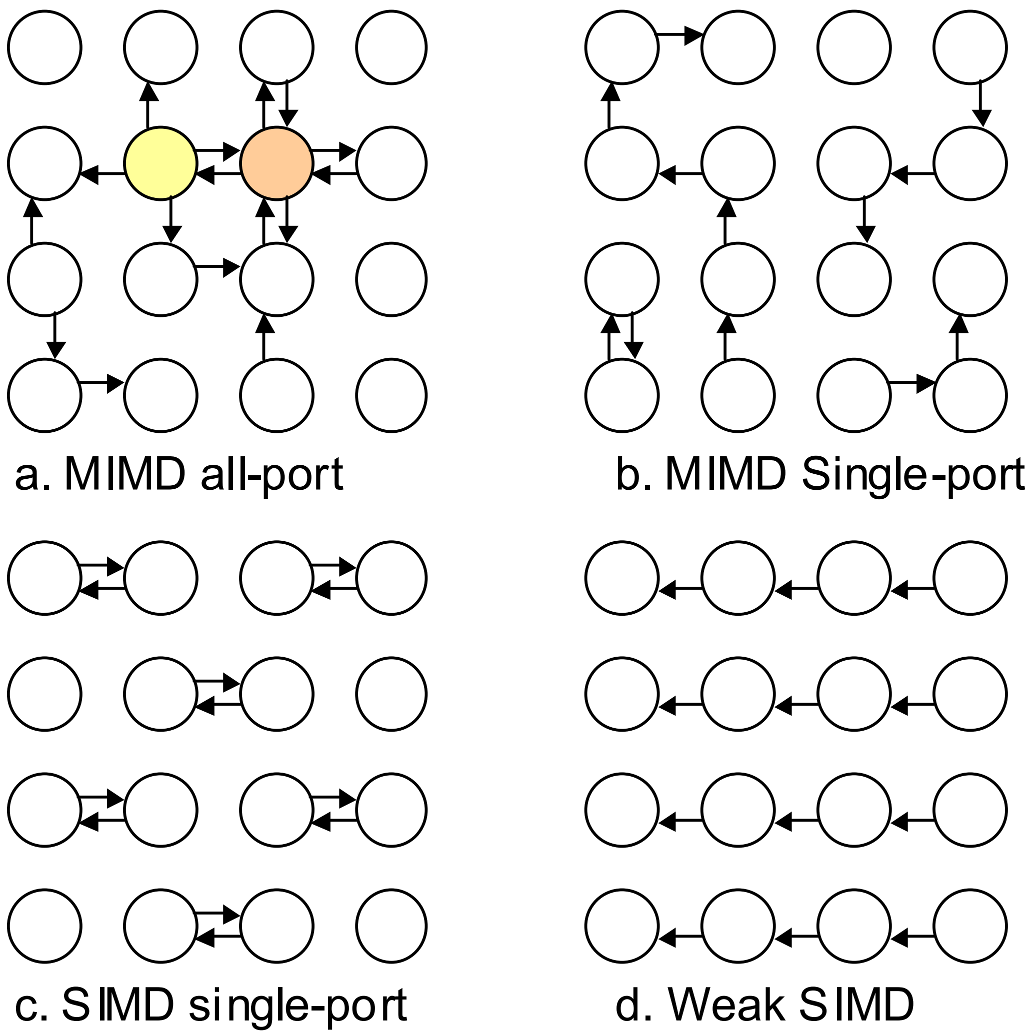
MIMD, SPMD, SIMD, Weak SIMD

Single/All-port model

Diameter-based or bisection-based lower bound: $O(p^{1/2})$

Fig. 9.1 Two-dimensional mesh-connected computer.

MIMD, SPMD, SIMD, or Weak SIMD Mesh



Some communication modes.

All-port: Processor can communicate with all its neighbors at once (in one cycle or time step)

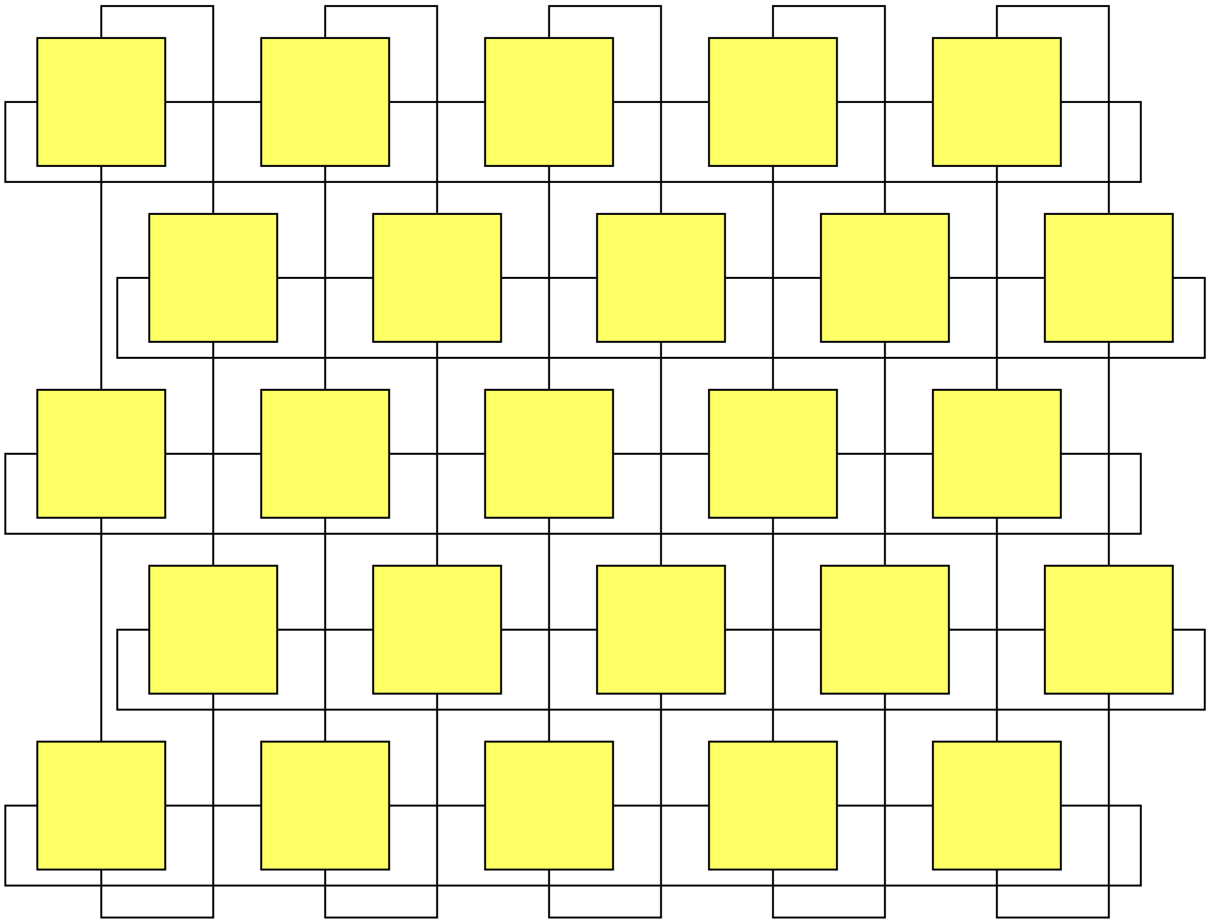
Single-port: Processor can send/receive one message per time step

MIMD: Processors choose their communication directions independently

SIMD: All processors directed to do the same

Weak SIMD: Same direction for all (uniaxis)

Torus Implementation without Long Wires



8 x 8 folded torus

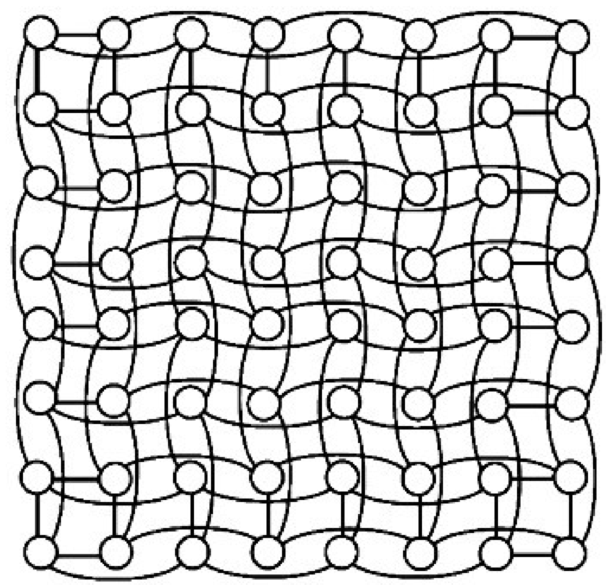


Fig. 9.2 A 5 $\times$ 5 torus folded along its columns. Folding this diagram along the rows will produce a layout with only short links.

Processor Indexing in Mesh or Torus

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

a. Row-major

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

b. Snakelike row-major

Our focus will be on row-major and snakelike row-major indexing

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

c. Shuffled row-major

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

d. Proximity order

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

Register-Based Communication

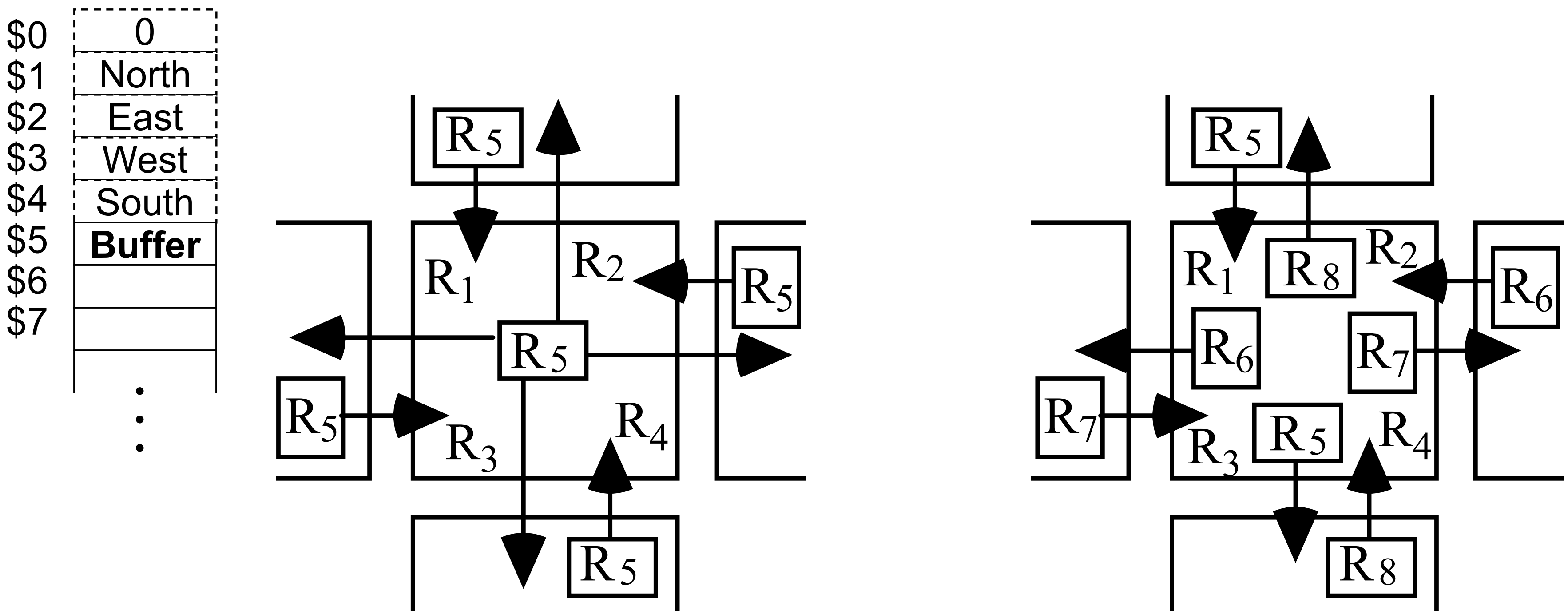


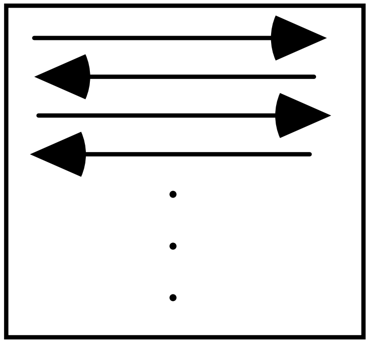
Fig. 9.4 Reading data from NEWS neighbors via virtual local registers.

9.2 The Shearsort Algorithm

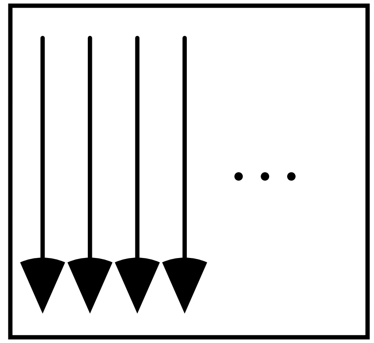
Shearsort algorihm for a 2D mesh with r rows

repeat $\lceil \log_2 r \rceil$ times

Sort the rows (snake-like)

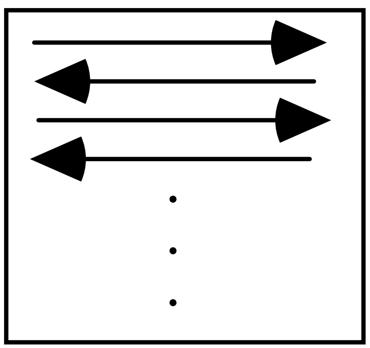


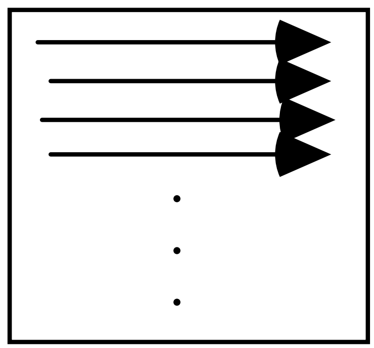
then sort the columns (top-to-bottom)



endrepeat

Sort the rows





Snakelike or Row-Major

(depending on the desired final sorted order)

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

On a square mesh:

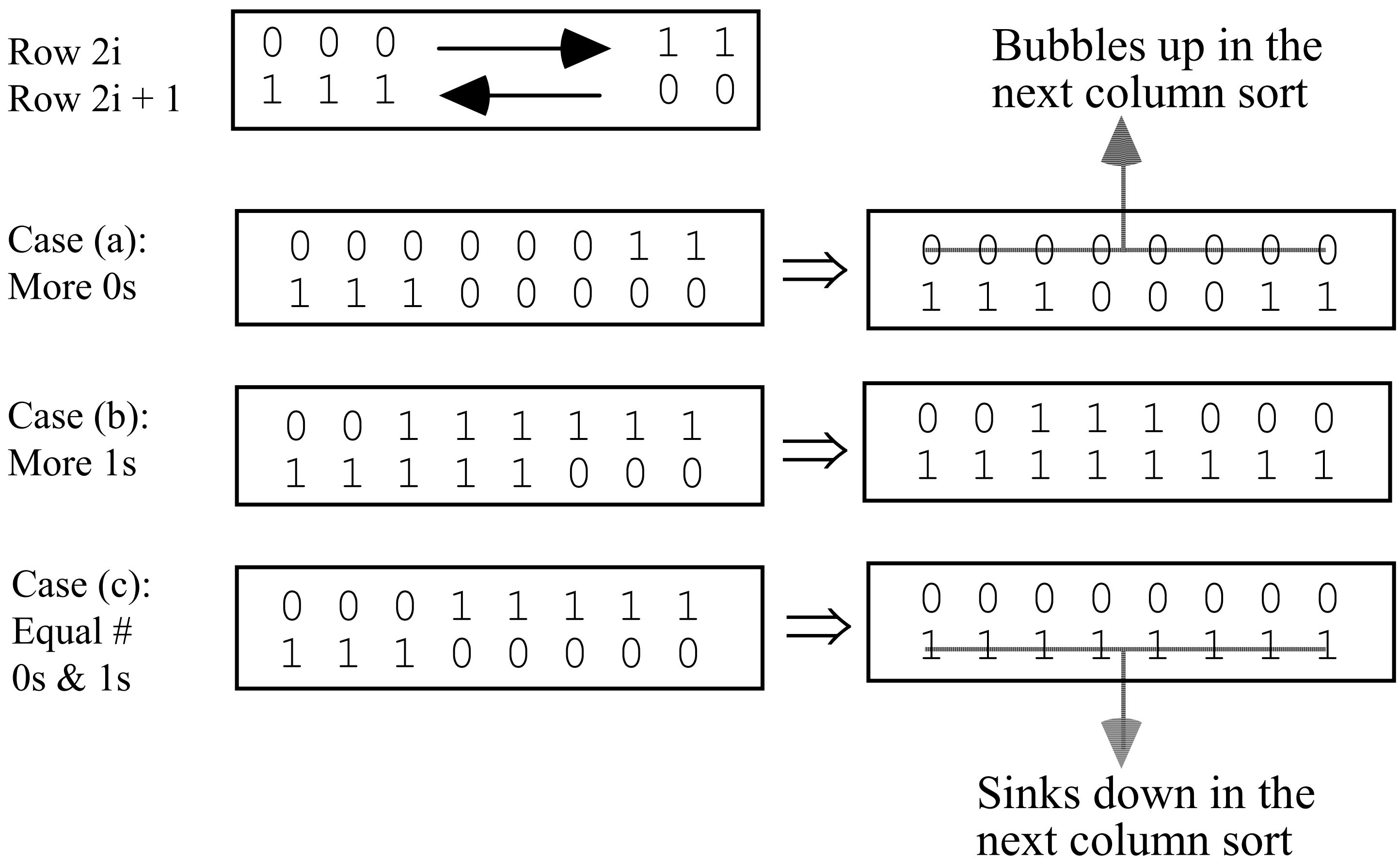
$T_{\text{shearsort}} = p^{1/2}(\log_2 p + 1)$

Diameter-based LB:

$T_{\text{sort}} \geq 2p^{1/2} - 2$

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

Proving Shearsort Correct



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

Shearsort Proof (Continued)

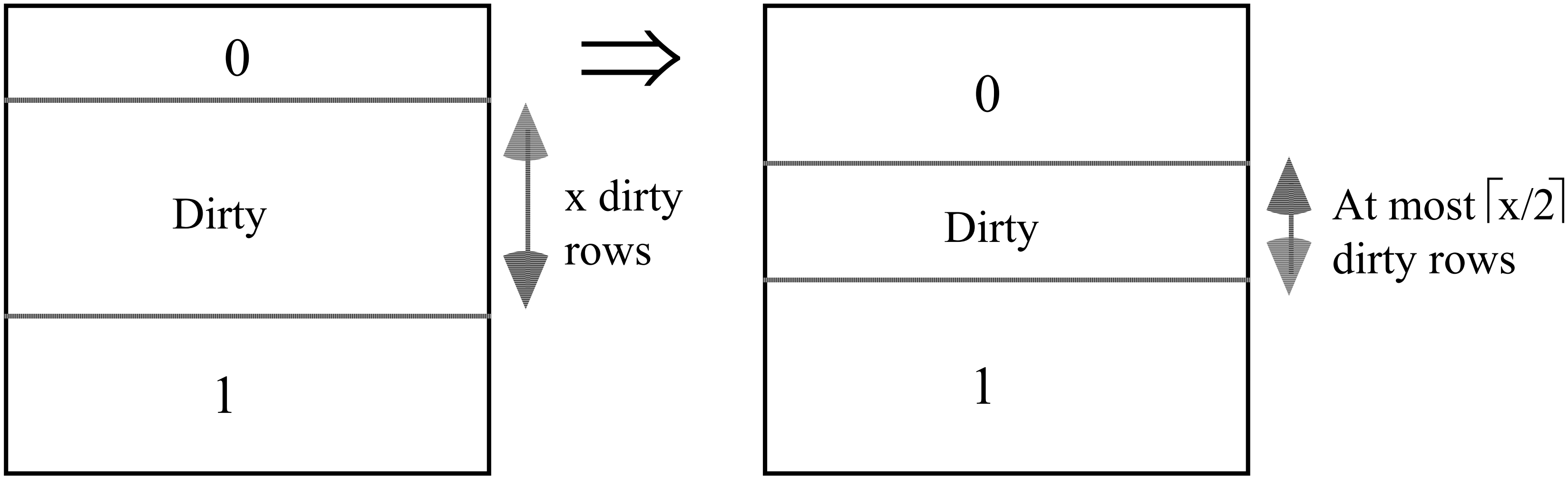
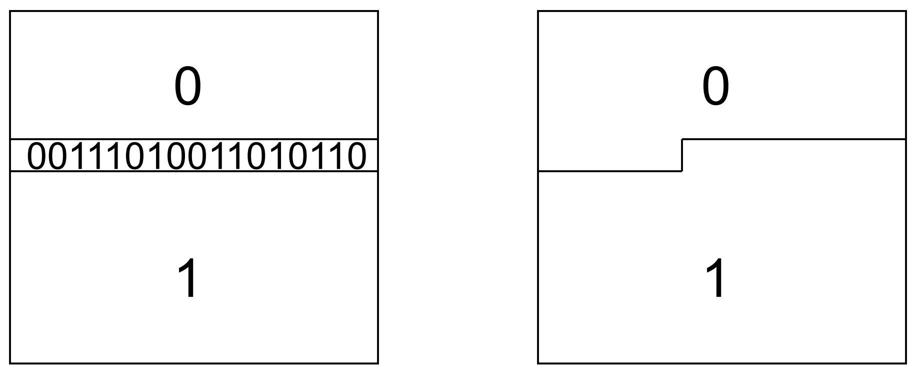


Fig. 9.7 The number of dirty rows halves with each shearsort iteration.

After $\log_2 r$ iterations, only one dirty row remains



Shearsort Example

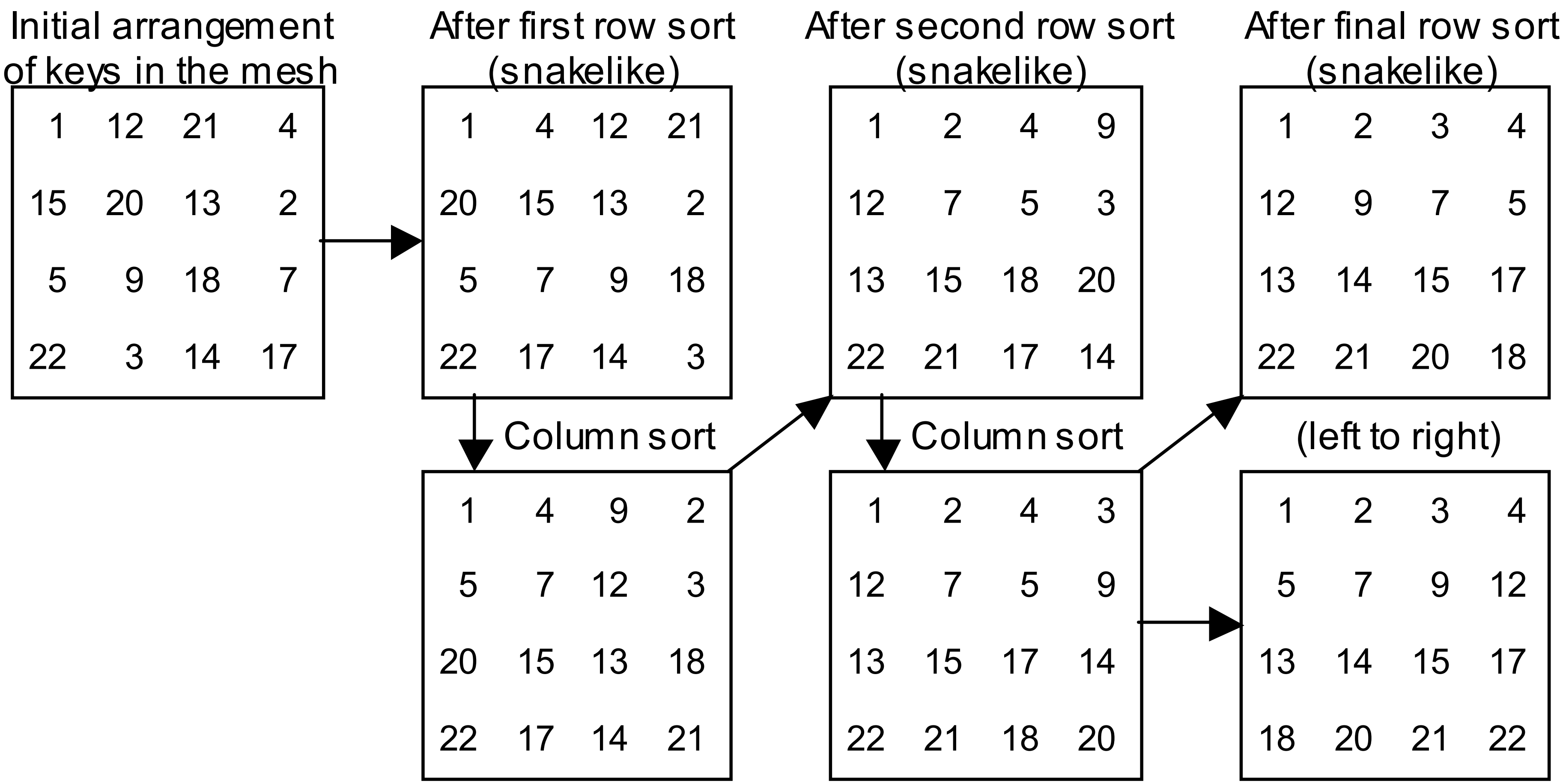


Fig. 9.8 Example of shearsort on a 4×4 mesh.

9.3 Variants of Simple Shearsort

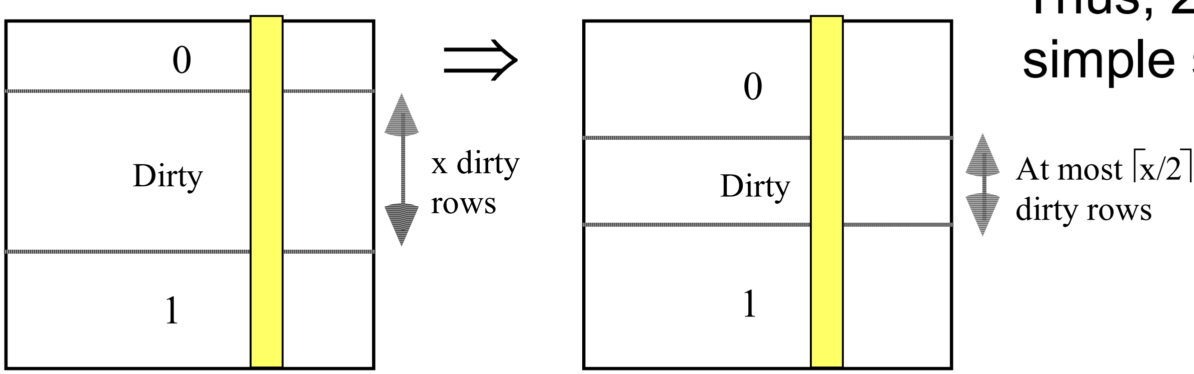
Observation: On a linear array, odd-even transposition sort needs only *k* steps if the "dirty" (unsorted) part of the array is of length *k*

$$0\quad 0\quad 0\quad 0\quad 0\quad 0\quad \underbrace{\boxed{1\quad 1\quad 0\quad 1\quad 0\quad 0\quad 0\quad 1\quad 0}}_{\text{Unsorted part}}\quad 1\quad 1\quad 1\quad 1\quad 1\quad 1$$

In shearsort, we do not have to sort columns completely, because only a portion of the column is unsorted (the portion shrinks in each phase)

$$T_{\text{opt shearsort}} = (p/r)(\lceil \log_2 r \rceil + 1) + \underbrace{r + r/2 + \ldots + 2}_{2r-2}$$

Thus, $2r - 2$ replaces $r \log_2 r$ in simple shearsort



On a square mesh:

$T_{\text{opt shearsort}} =$
$p^{1/2}(\tfrac{1}{2} \log_2 p + 3) - 2$

Shearsort with Multiple Items per Processor

x, y	Two keys held by one processor

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

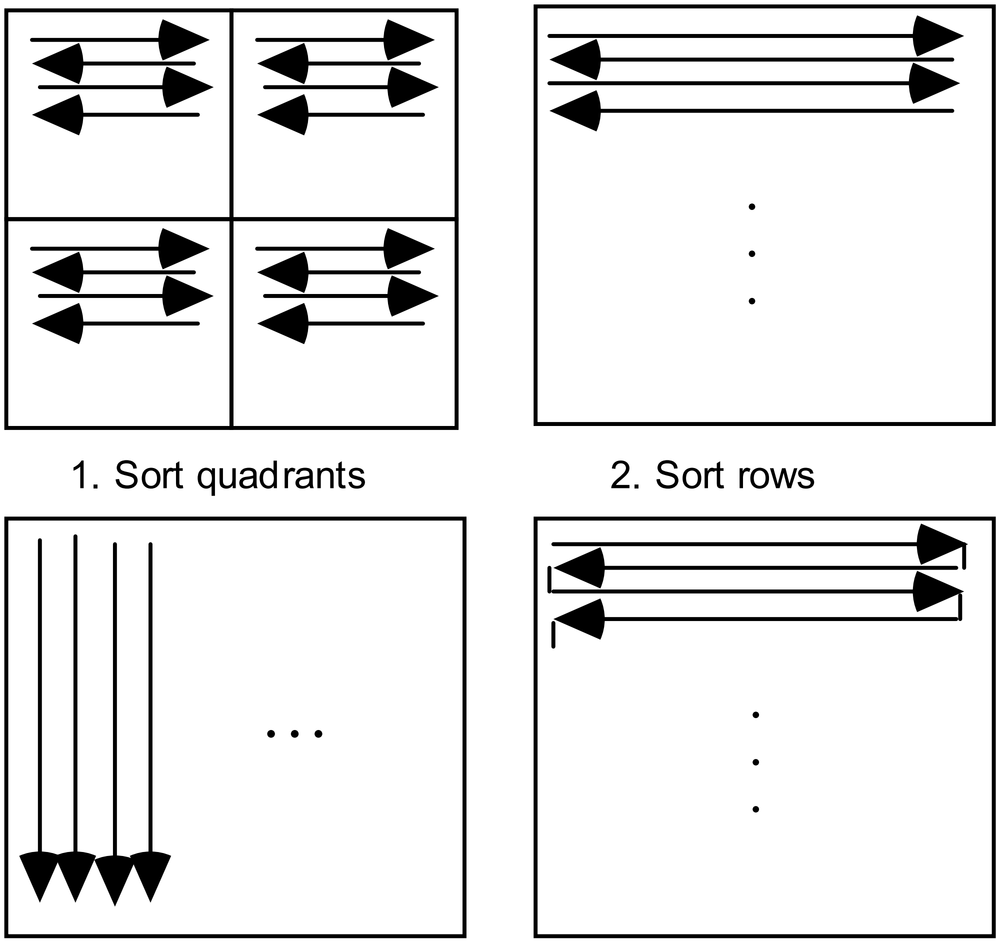
The final row sort (snake-like or row-major) is not shown.

Perform ordinary shearsort, but replace compare-exchange with merge-split

$(n/p) \log_2(n/p)$ steps for the initial sort; the rest multiplied by n/p

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 overall snake

Snakelike sorting order on a square mesh

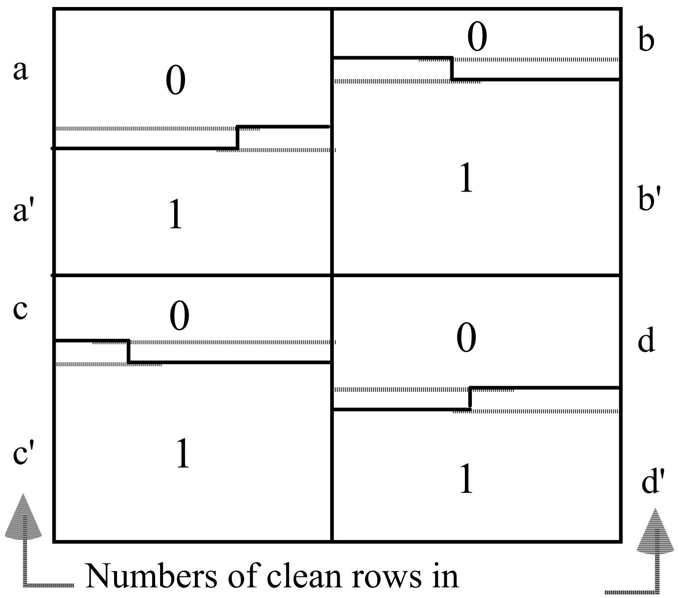
$$T(p^{1/2}) = T(p^{1/2}/2) + 5.5p^{1/2}$$

Note that row sort in phase 2 needs fewer steps

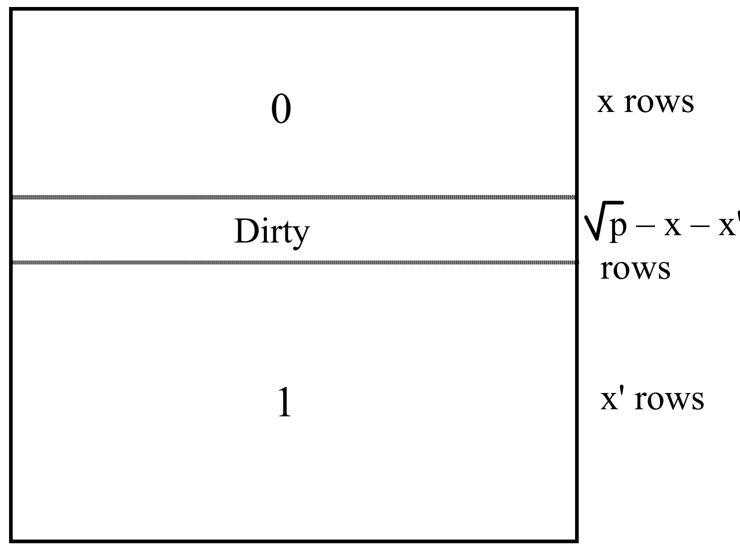
$$T_{\text{recursive 1}} \cong 11p^{1/2}$$

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

Proof of the $11p^{1/2}$-Time Sorting Algorithm



Numbers of clean rows in each of the four quadrants



State of the array after Phase 3

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

$x \geq b + c + \lfloor (a - b)/2 \rfloor + \lfloor (d - c)/2 \rfloor$ A similar inequality applies to x'

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

Some Programming Considerations

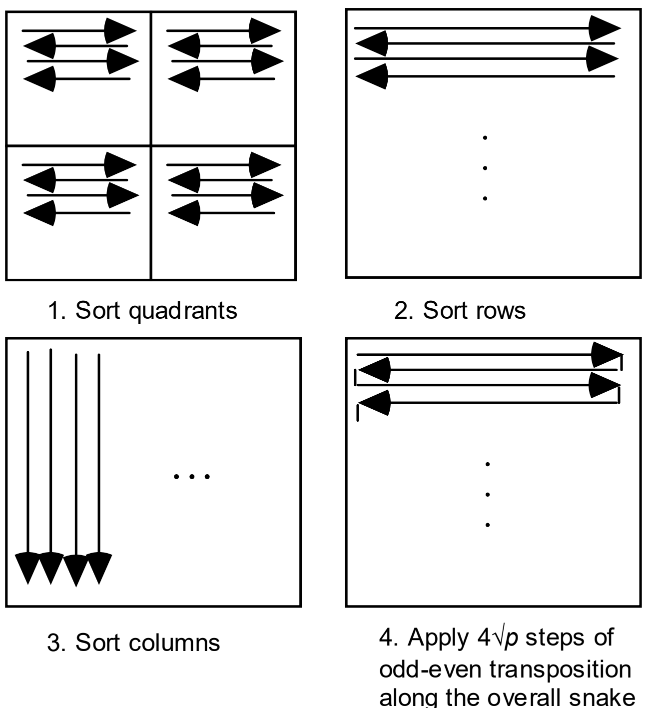


Fig. 9.10

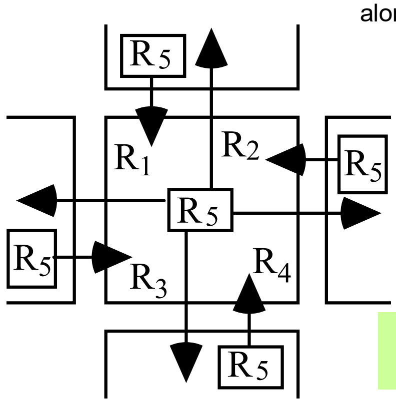


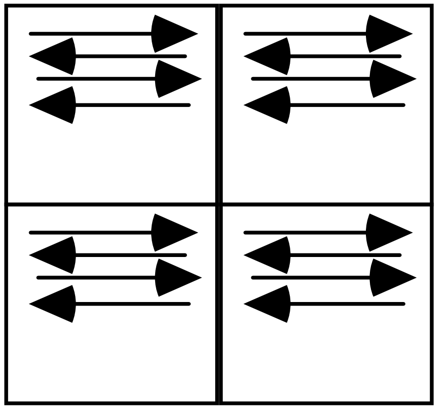
Fig. 9.4

Let b (a power of 2) be the block length for snakelike sorting

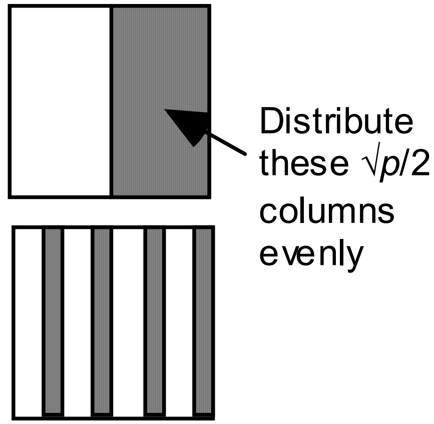
<u>snakelike-mesh-sort(b)</u>
  snakelike-mesh-sort(b/2)
  snakelike-row-sort(b)
  column-sort(b)
  snake-odd-even-xpose(4b)

<u>snakelike-row-sort(b)</u>
  for k = 0 to b – 1 Proc (i, j), j even, do
  case i, k
    even, even: if $j \neq 0 \bmod b$ AND
        (R5) < (R3) then R5 ↔ R3
    even, odd: if (R2) < (R5) then R2 ↔ R5
    . . .

Another Recursive Sorting Algorithm

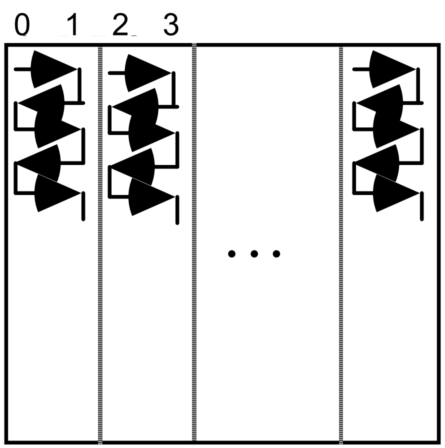


1. Sort quadrants

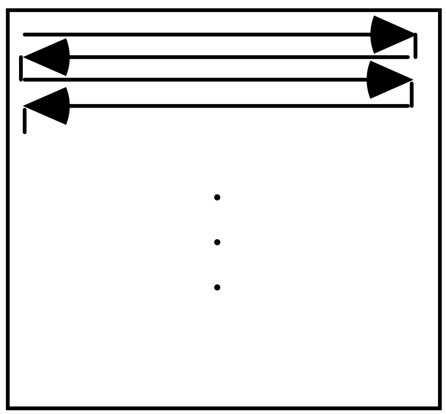


2. Shuffle row elements

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



3. Sort double columns in snakelike order



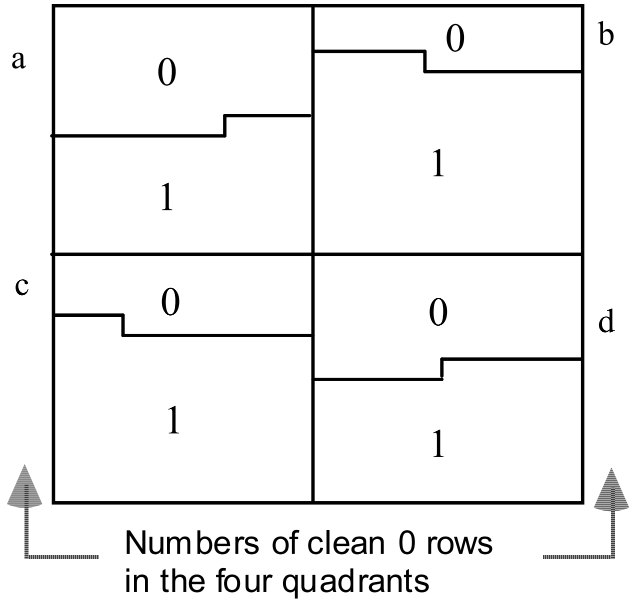
4. Apply $2\sqrt{p}$ steps of odd-even transposition along the overall snake

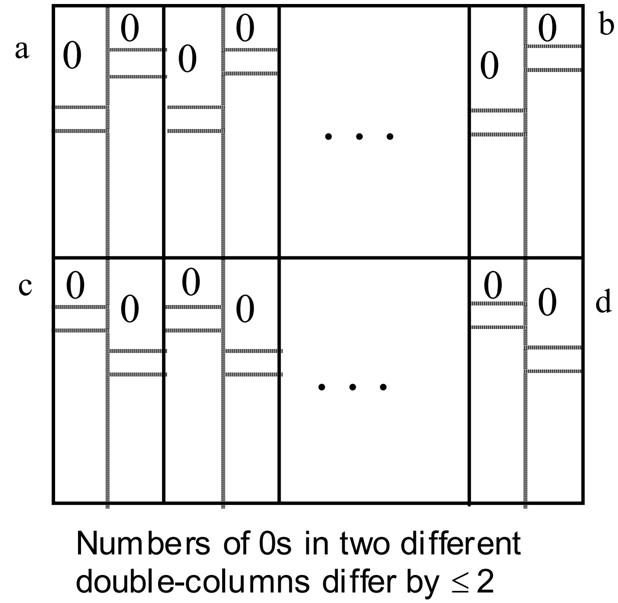
$$T(p^{1/2}) = T(p^{1/2}/2) + 4.5p^{1/2}$$

Note that the distribution in phase 2 needs $\frac{1}{2}p^{1/2}$ steps

$$T_{\text{recursive 2}} \cong 9p^{1/2}$$

Proof of the $9p^{1/2}$-Time Sorting Algorithm





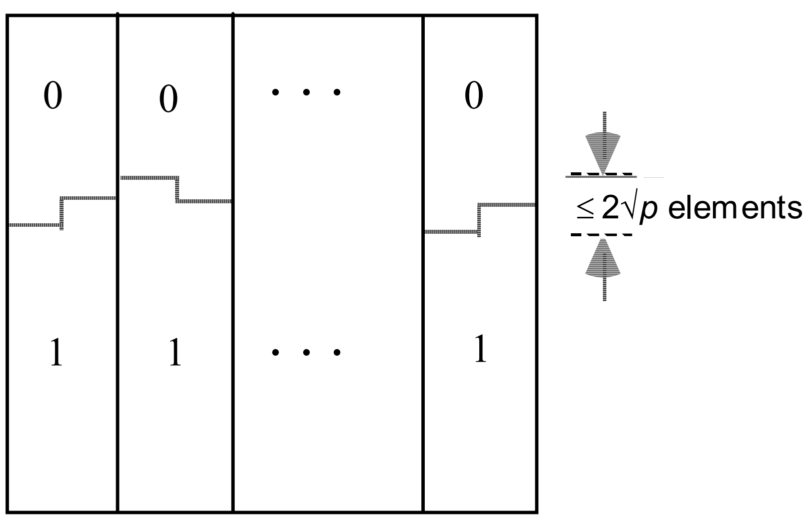
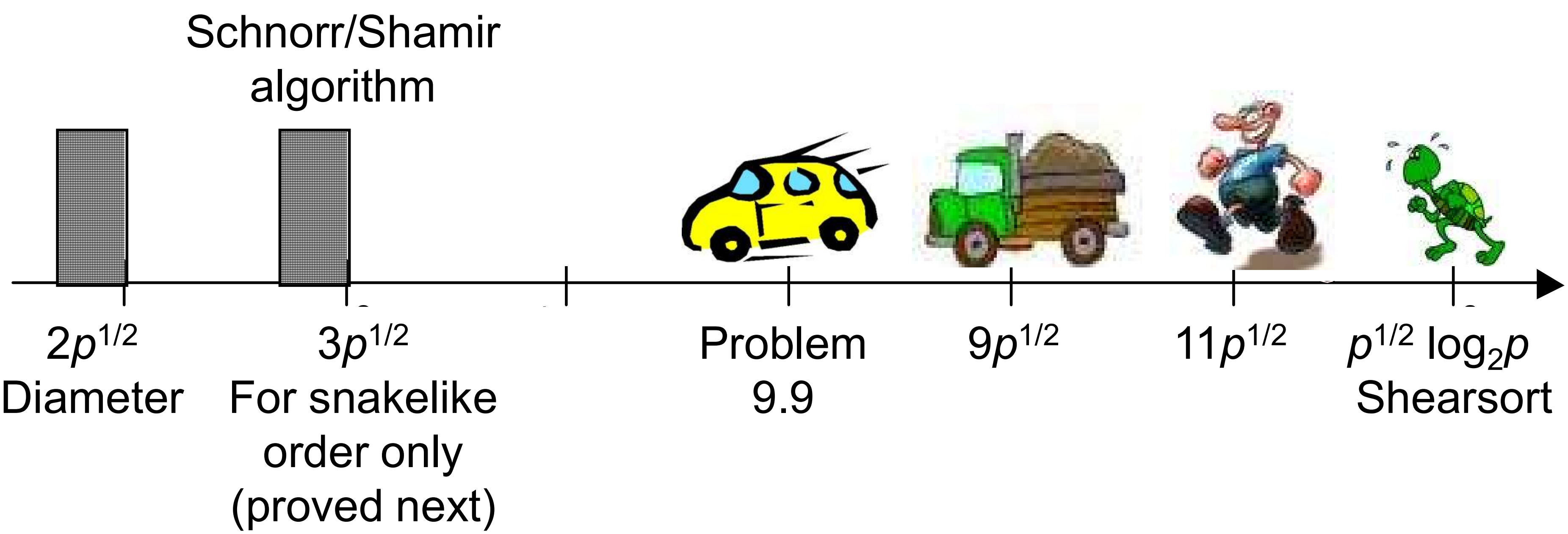


Fig. 9.13 The proof of the second recursive sorting algorithm for 2D meshes.

Our Progress in Mesh Sorting Thus Far

Lower bounds: Theoretical arguments based on bisection width, and the like

Upper bounds: Deriving/analyzing algorithms and proving them correct



9.5 A Nontrivial Lower Bound

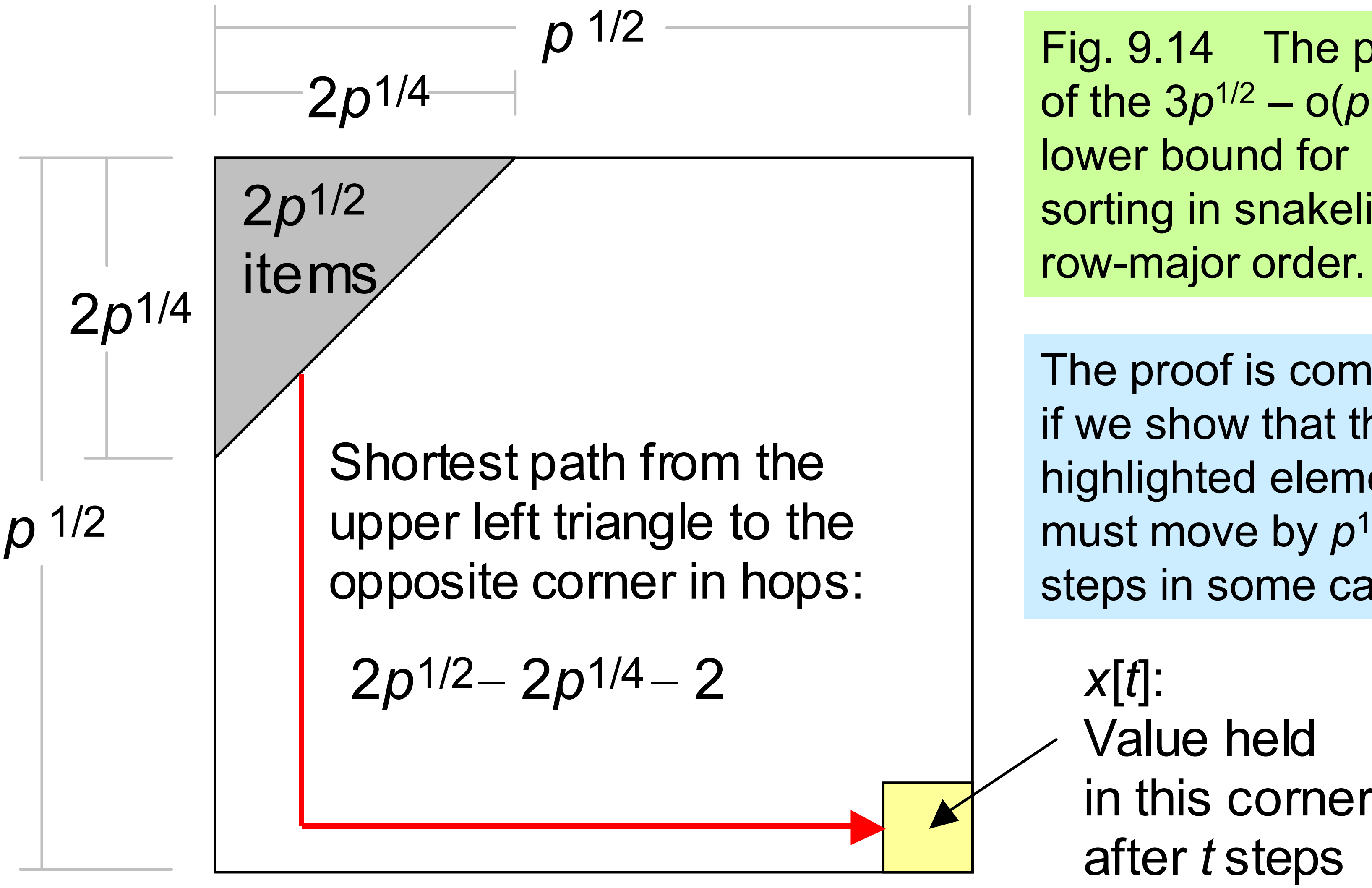
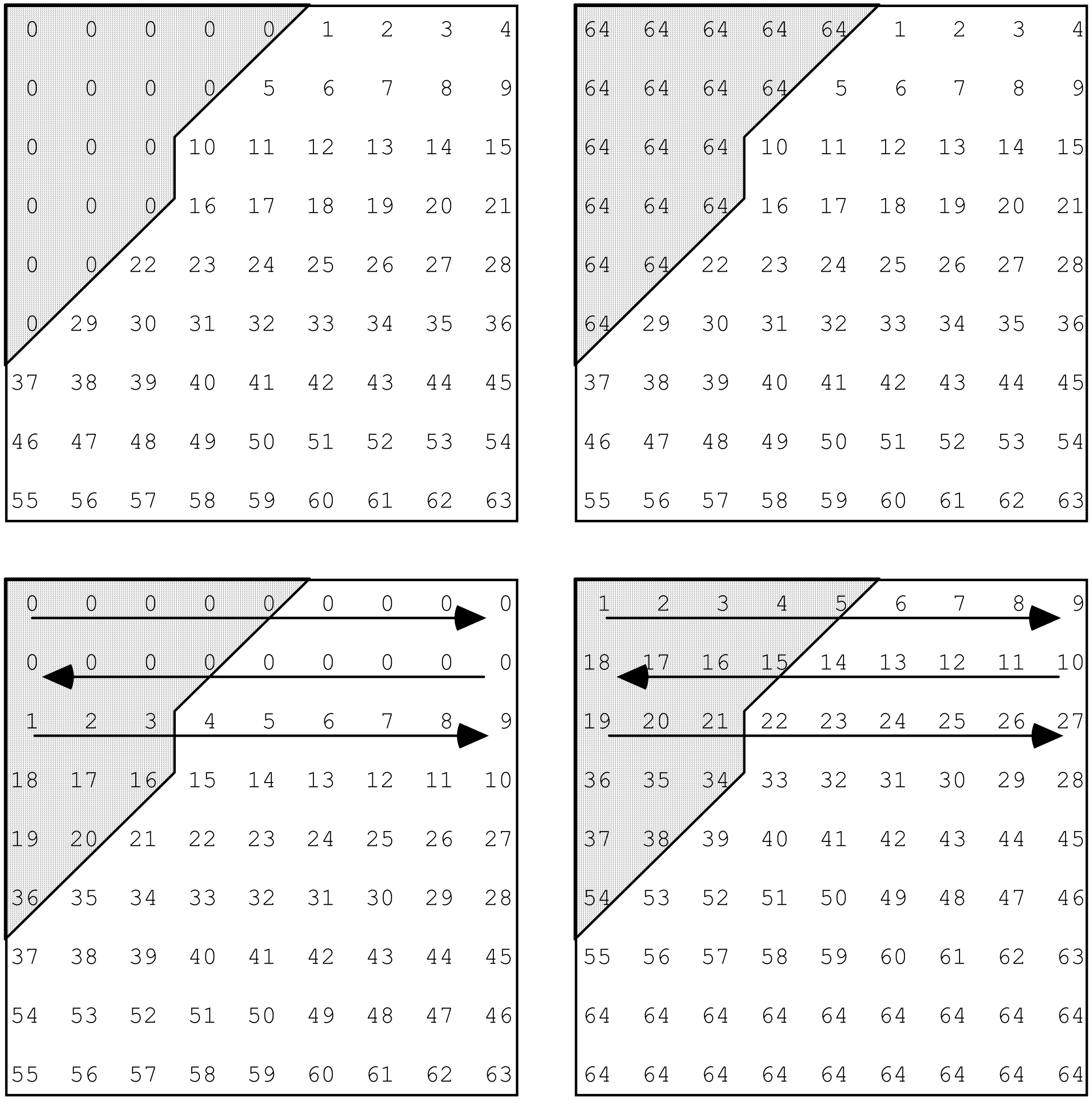


Fig. 9.14 The proof of the $3p^{1/2} - o(p^{1/2})$ lower bound for sorting in snakelike row-major order.

The proof is complete if we show that the highlighted element must move by $p^{1/2}$ steps in some cases

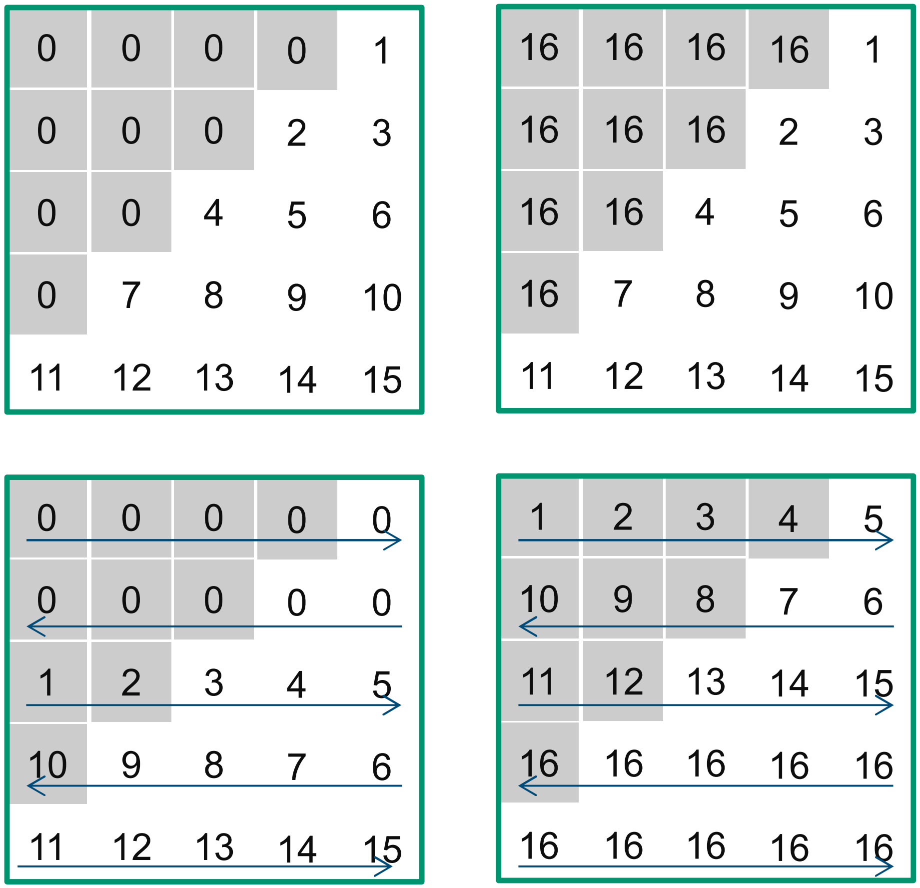


Proving the Lower Bound

Any of the values 1-63 can be forced into any desired column in sorted order by mixing 0s and 64s in the shaded area

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

Proving the Lower Bound



Any of the values 1-63 can be forced into any desired column in sorted order by mixing 0s and 64s in the shaded area

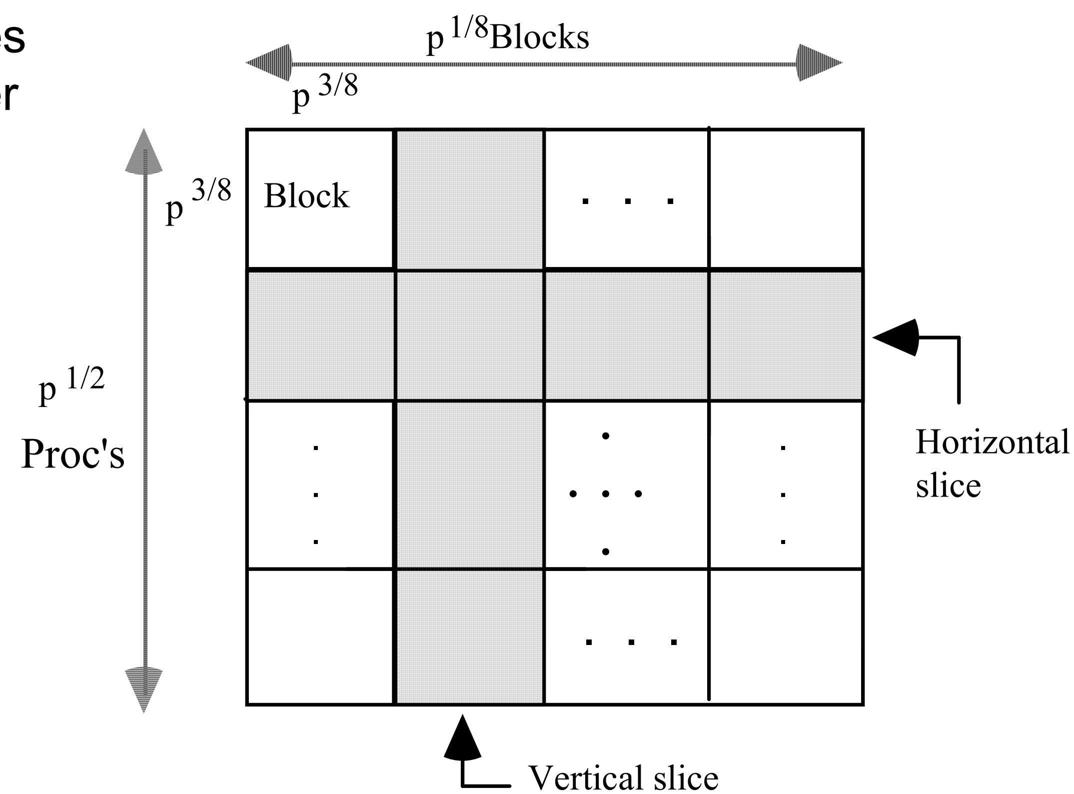
Fig. 9.15 (Alternate version) Illustrating the effect of fewer or more 0s in the shaded area.

9.6 Achieving the Lower Bound

Schnorr-Shamir snakelike sorting

1. Sort each block in snakelike order
2. Permute columns such that the columns of each vertical slice are evenly distributed among all slices
3. Sort each block in snakelike order
4. Sort columns 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 snakelike order
7. Sort rows in snakelike order
8. Apply $2p^{3/8}$ steps of odd-even transposition to the snake

Fig. 9.16 Notation for the asymptotically optimal sorting algorithm.



Elaboration on the $3p^{1/2}$ Lower Bound

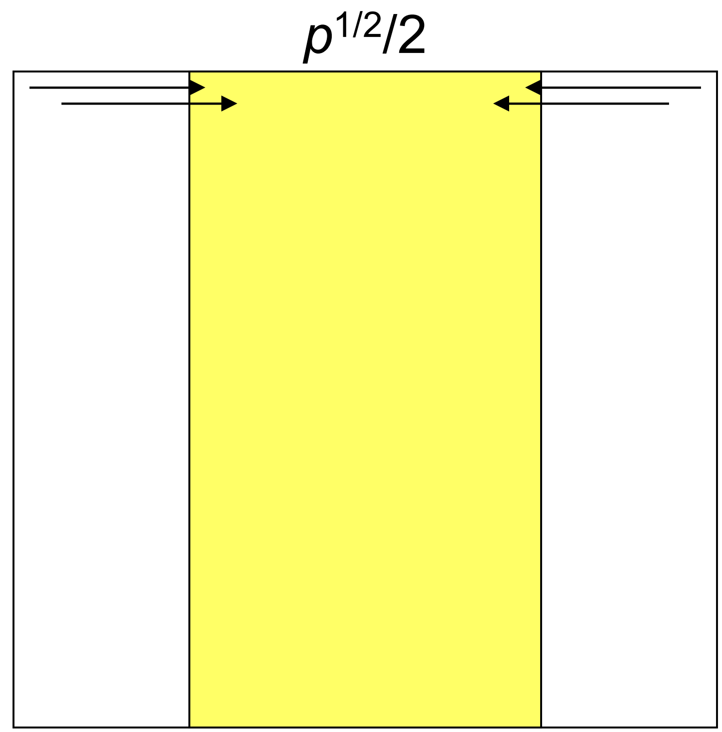
In deriving the $3p^{1/2}$ lower bound for snakelike sorting on a square mesh, we implicitly assumed that each processor holds one item at all times

Without this assumption, the following algorithm leads to a running time of about $2.5p^{1/2}$

Phase 1: Move all data to the center $p^{1/2}/2$ columns — $p^{1/2}/4$

Phase 2: Perform 2-2 sorting in the half-wide center mesh — $2p^{1/2}$

Phase 3: Distribute data from center half of each row to the entire row — $p^{1/2}/4$



10 Routing on a 2D Mesh or Torus

Routing is nonexistent in PRAM, hardwired in circuit model:
- Study point-to-point and collective communication
- Learn how to route multiple data packets to destinations

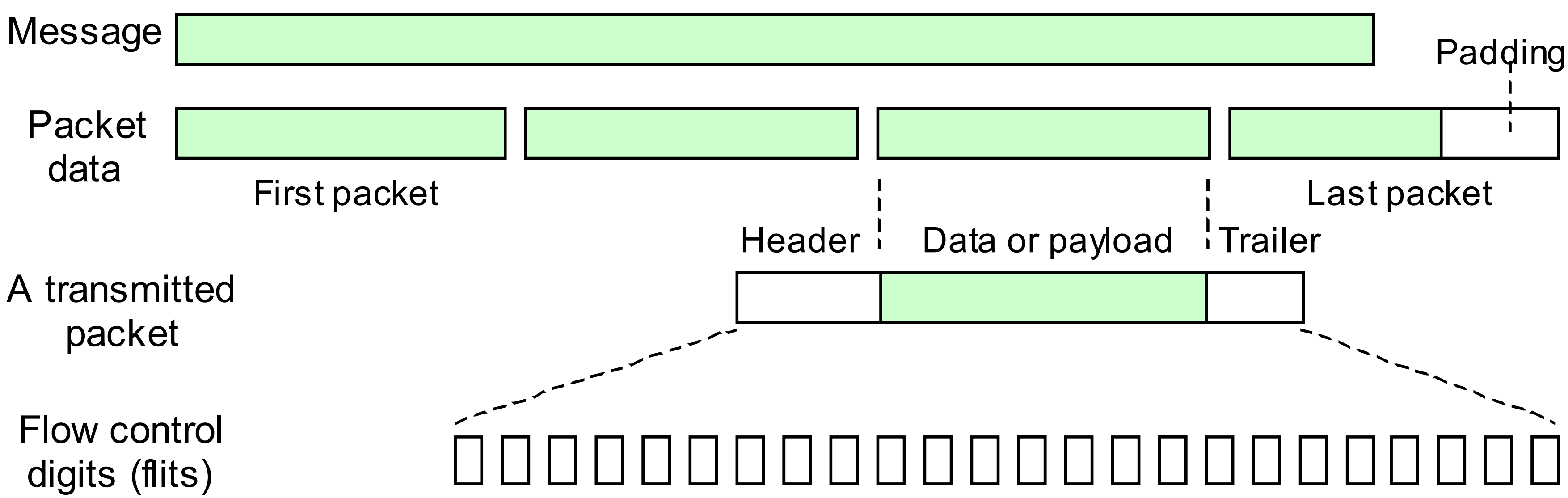
Topics in This Chapter
10.1 Types of Data Routing Operations
10.2 Useful Elementary Operations
10.3 Data Routing on a 2D Array
10.4 Greedy Routing Algorithms
10.5 Other Classes of Routing Algorithms
10.6 Wormhole Routing

10.1 Types of Data Routing Operations

Point-to-point communication: one source, one destination

Collective communication

One-to-many: multicast, broadcast (one-to-all), scatter
Many-to-one: combine (fan-in), global combine, gather
Many-to-many: all-to-all broadcast (gossiping), scatter-gather



Types of Data Routing Algorithms

Oblivious: A source-destination pair leads to a unique path; non-fault-tolerant

Adaptive: One of the available paths is chosen dynamically; can avoid faulty nodes/links or route around congested areas

Degree of adaptivity leads to trade-offs between decision simplicity (e.g., hard to avoid infinite loops) and routing flexibility

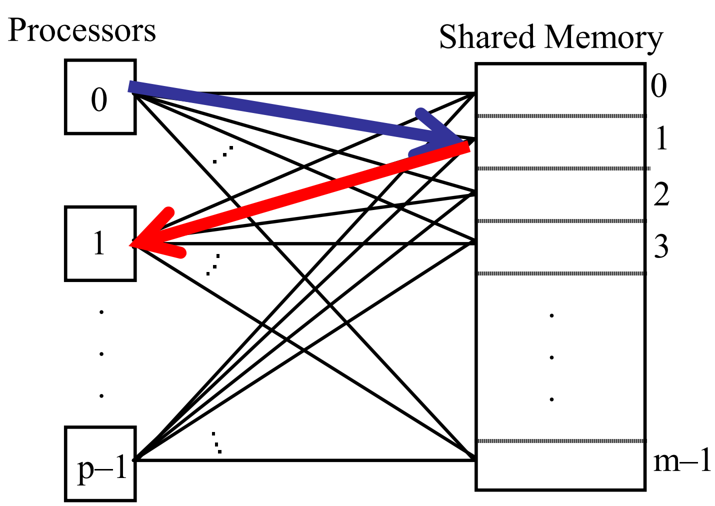
Optimal (shortest-path): Only shortest paths considered; can be oblivious or adaptive

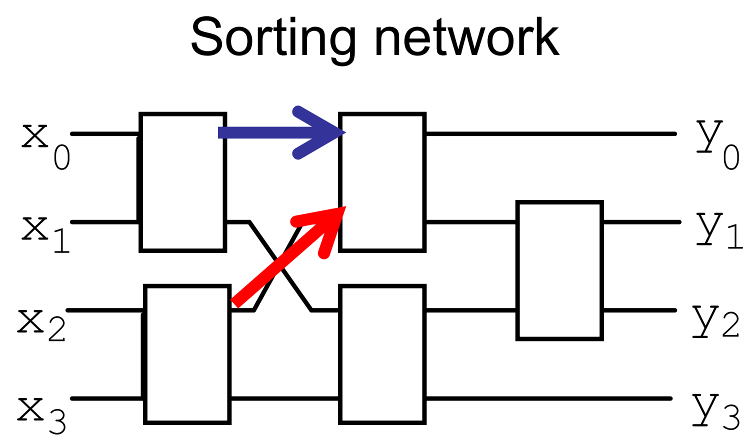
Non-optimal (non-shortest-path): Selection of shortest path is not guaranteed, although most algorithms tend to choose a shortest path if possible

Our First Encounter with Data Routing Issues

Shared memory: Processors can communicate by storing data into and reading data from the memory

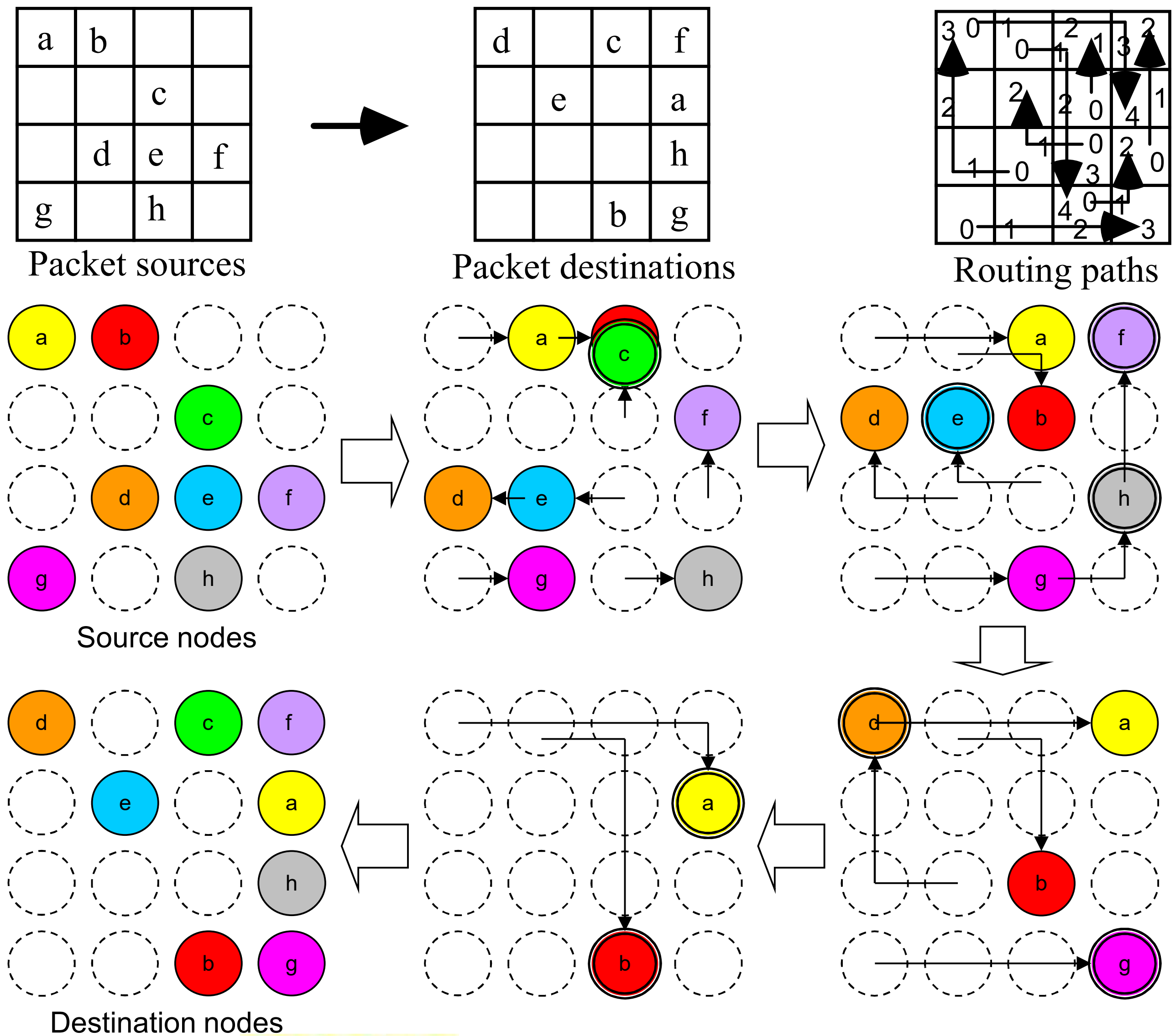
Circuit model: Sending results from one part of the system to other parts is hardwired at design time





Graph model: We must specify the routing process explicitly

1-to-1 Communication (Point-to-Point Messages)



Message sources, destinations, and routes

Routing Operations Specific to Meshes

Data compaction or packing

Move scattered data elements to the smallest possible submesh (e.g., for problem size reduction)

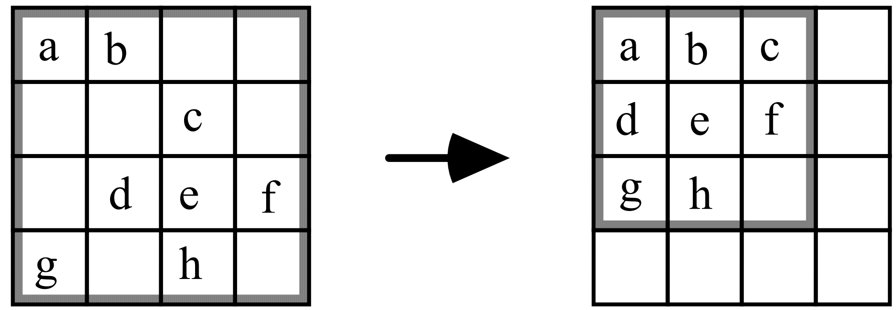
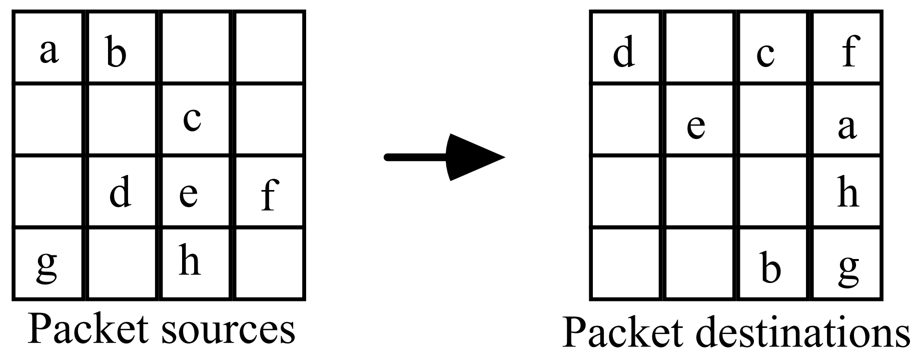


Fig. 10.1 Example of data compaction or packing.

Random-access write (RAW)

Emulates one write step in PRAM (EREW vs CRCW)

Routing algorithm is critical



Random-access read (RAR)

Can be performed as two RAWs: Write source addresses to destinations; write data back to sources (emulates on PRAM memory read step)

10.2 Useful Elementary Operations

Row/Column rotation

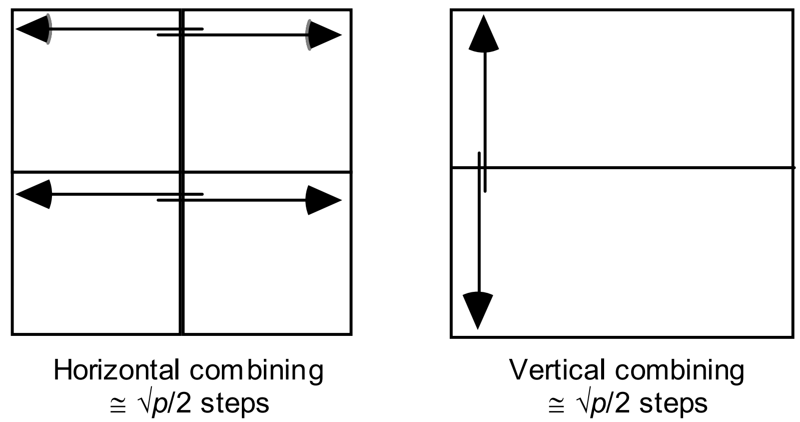
All-to-all broadcasting in a row or column

Sorting in various orders

Chapter 9

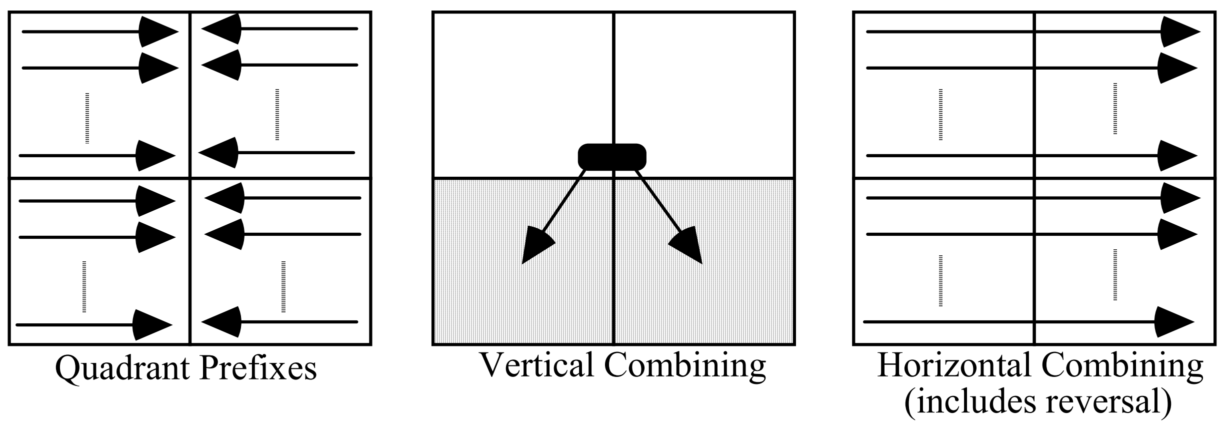
Semigroup computation

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 on a Linear Array (Mesh Row or Column)

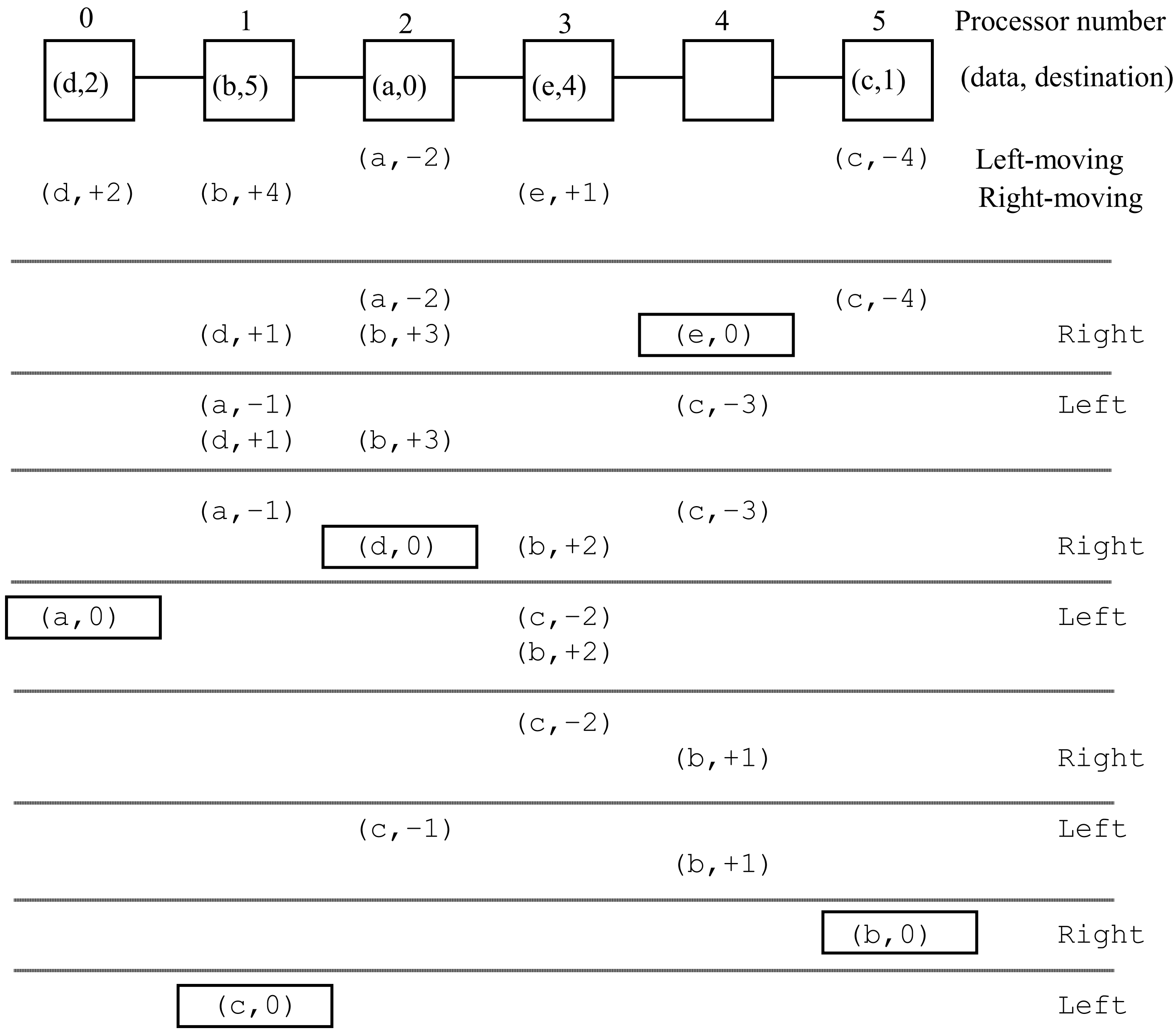


Fig. 10.4 Example of routing multiple packets on a linear array.

10.3 Data Routing on a 2D Array

<u>Exclusive random-access write on a 2D mesh: *MeshRAW*</u>

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 (no conflict; at most one element in a row 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.

Analysis of Sorting-Based Routing Algorithm

	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

Not a shortest-path algorithm

T	$=$	$3p^{1/2} + o(p^{1/2})$	{snakelike sorting}
		$+\ p^{1/2}$	{odd column reversals}
		$+\ 2p^{1/2} - 2$	{row & column routing}
	$=$	$6p^{1/2} + o(p^{1/2})$	
	$=$	$11p^{1/2} + o(p^{1/2})$	with unidirectional commun.

Node buffer space requirement: 1 item at any given time

10.4 Greedy Routing Algorithms

Greedy algorithm: In each step, try to make the most progress toward the solution based on current conditions or information available

This local or short-term optimization often does not lead to a globally optimal solution; but, problems with optimal greedy algorithms do exist

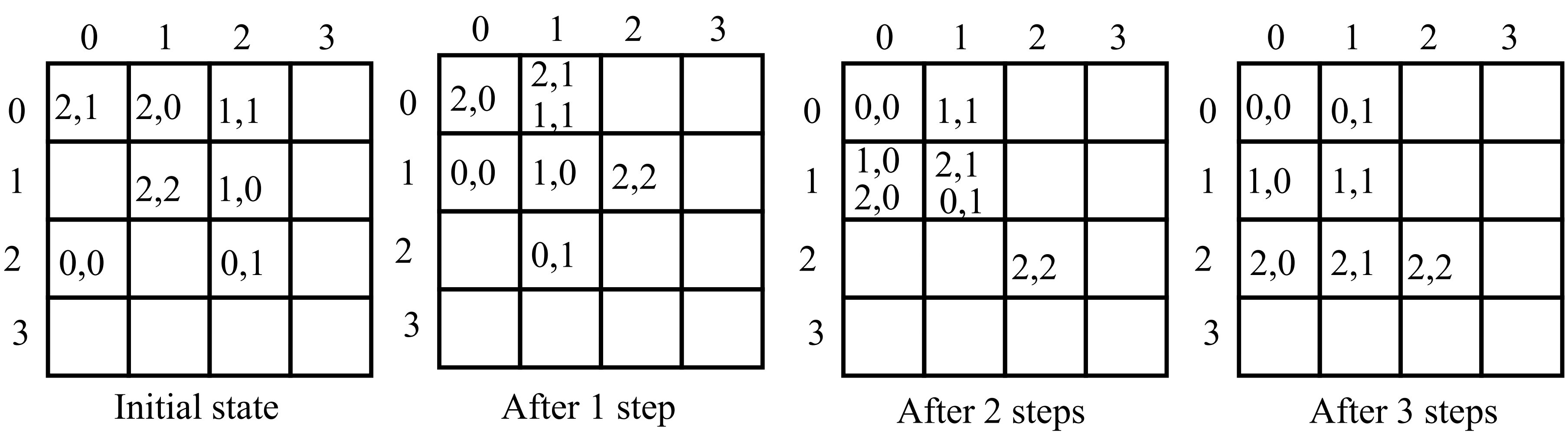


Fig. 10.6 Greedy row-first routing on a 2D mesh.

Analysis of Row-First Greedy Routing

$T = 2p^{1/2} - 2$

This optimal time achieved if we give priority to messages that need to go further along a column

Thus far, we have two mesh routing algorithms:

$6p^{1/2}$-step, 1 buffer per node

$2p^{1/2}$-step, time-optimal, but needs large buffers

Question: Is there a middle ground?

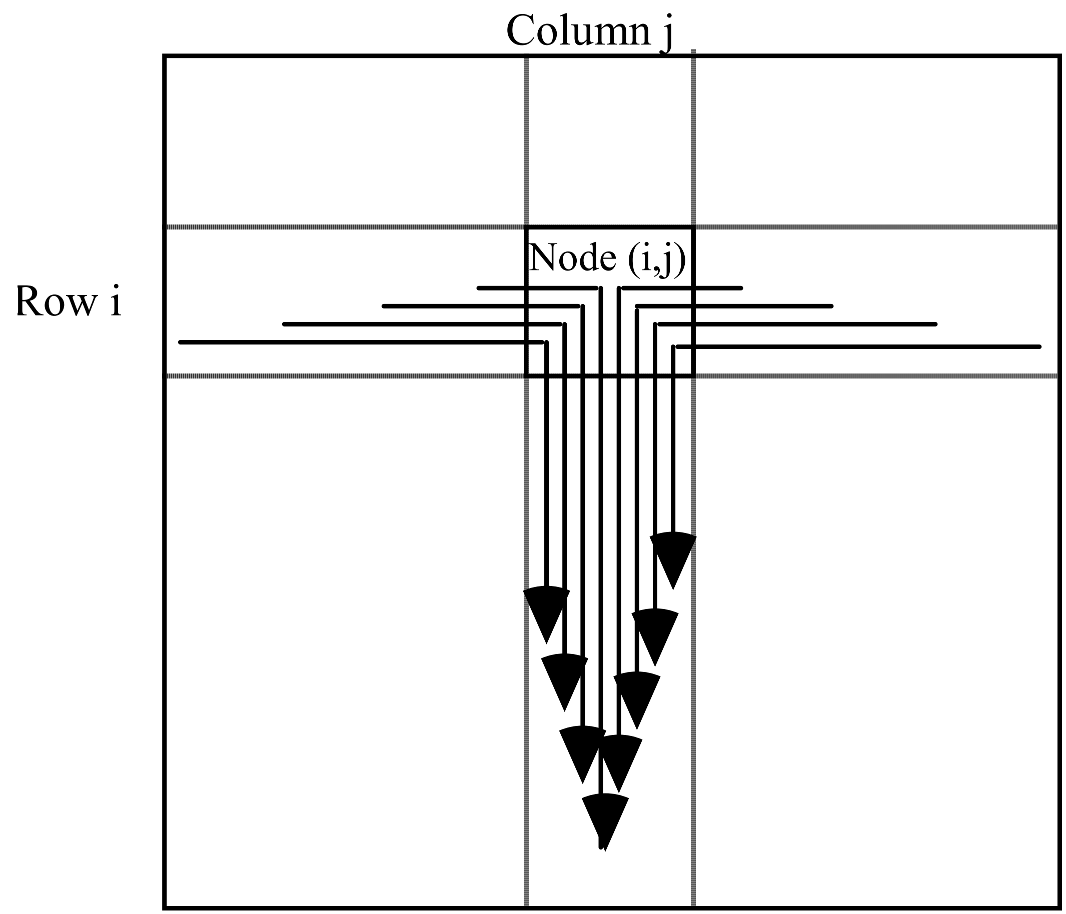


Fig. 10.7 Demonstrating the worst-case buffer requirement with row-first routing.

An Intermediate Routing Algorithm

Sort $(p^{1/2}/q) \times (p^{1/2}/q)$ submeshes in column-major order

Perform greedy routing

Let there be r_k packets in B_k headed for column j

Number of row-i packets headed for column j:

$$\sum_{k=0 \text{ to } q-1} \lceil r_k / (p^{1/2}/q) \rceil$$
$$< \sum [1 + r_k / (p^{1/2} / q)]$$
$$\leq q + (q / p^{1/2}) \sum r_k \leq 2q$$

So, $2q - 1$ buffers suffice

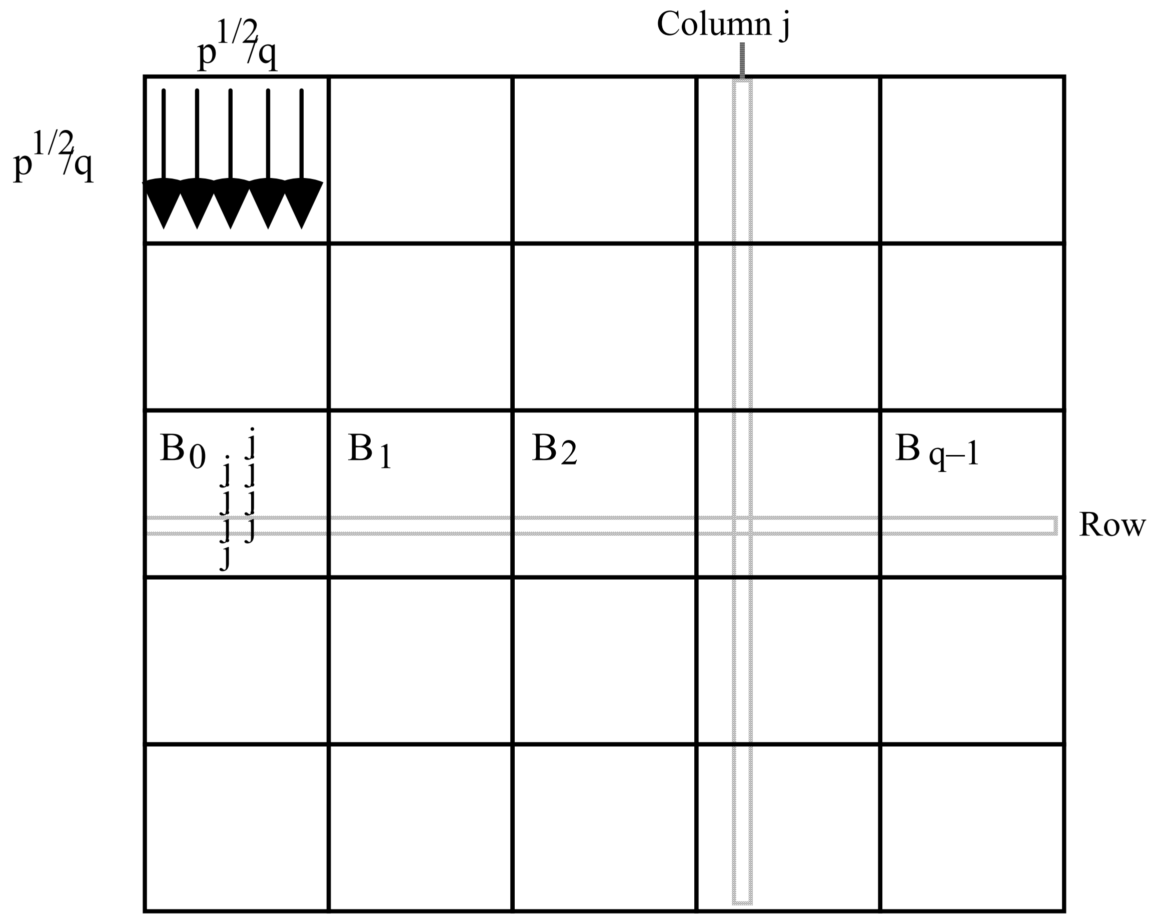


Fig. 10.8 Illustrating the structure of the intermediate routing algorithm.

Analysis of the Intermediate Algorithm

Buffers: $2q - 1$,
Intermediate between
1 and $O(p^{1/2})$

Sort time: $4p^{1/2}/q + o(p^{1/2}/q)$

Routing time: $2p^{1/2}$

Total time: $\cong 2p^{1/2} + 4p^{1/2}/q$

One extreme, $q = 1$:
Degenerates into
sorting-based routing

Another extreme, large q:
Approaches the greedy
routing algorithm

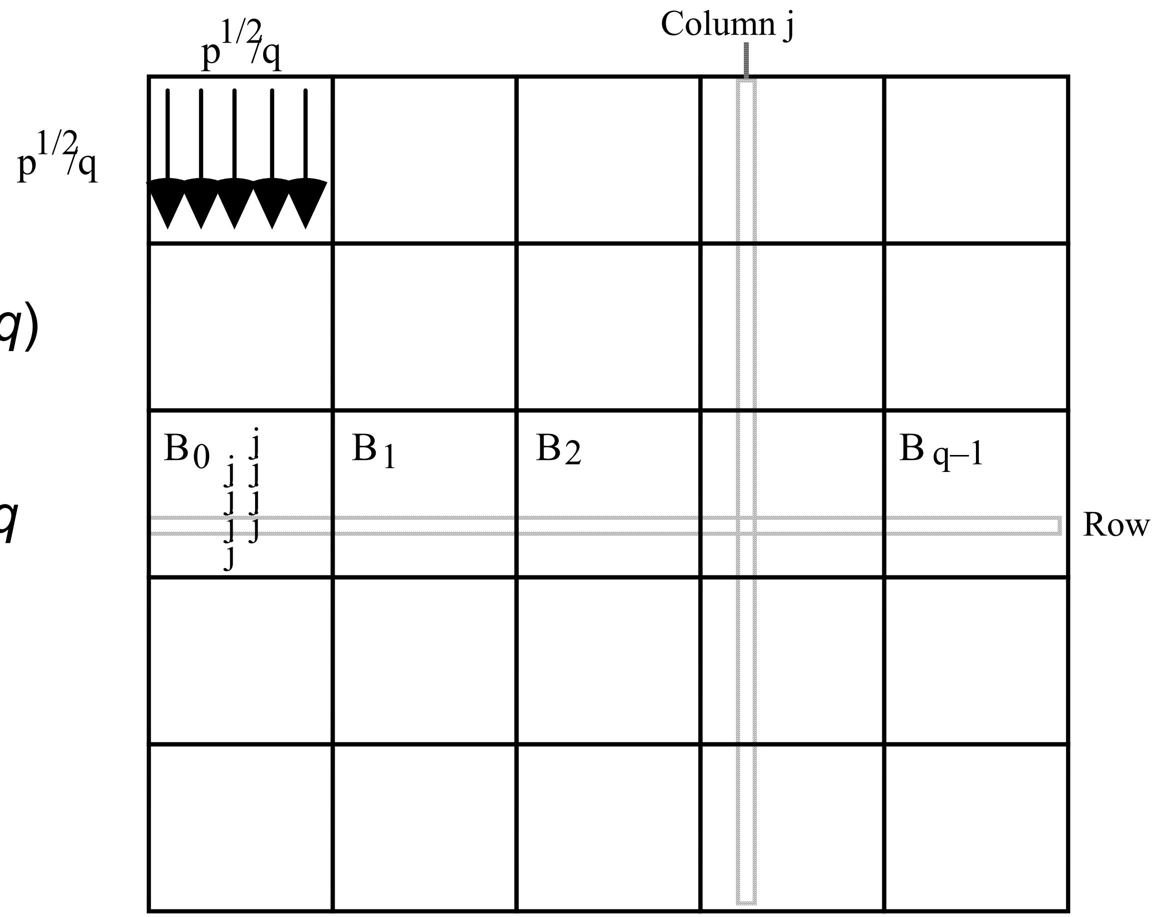


Fig. 10.8 Illustrating the structure of the intermediate routing algorithm.

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

Regardless of the routing algorithm used, concurrent writes can degrade the performance

Priority or combining scheme can be built into the routing algorithm so that congestion close to the common destination is avoided

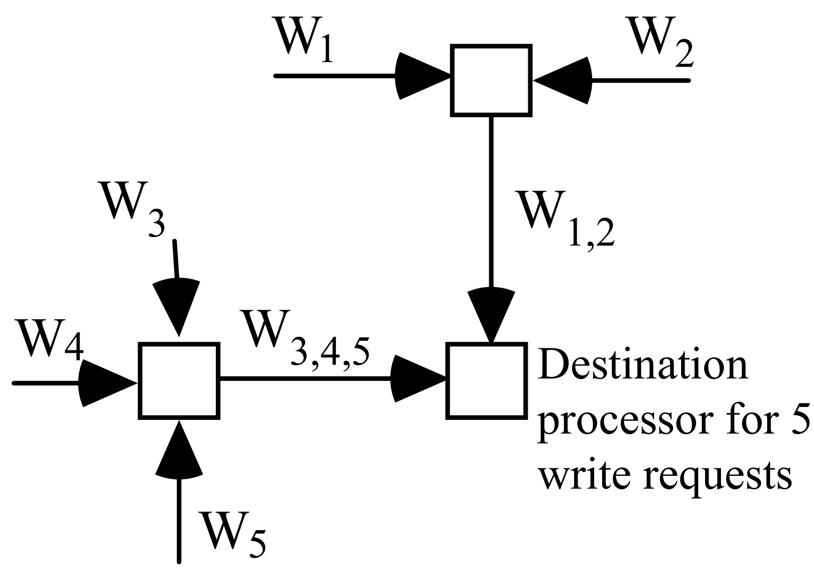
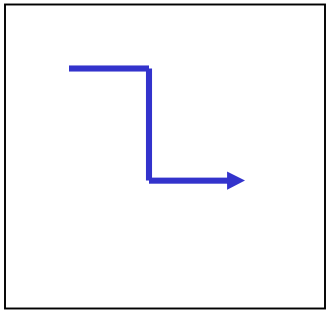
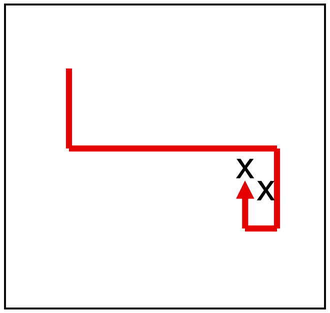


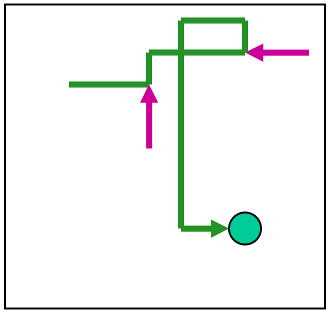
Fig. 10.9 Combining of write requests headed for the same destination.

Types of Routing Problems or Algorithms

Static:	Packets to be routed all available at $t = 0$
Dynamic:	Packets "born" in the course of computation
Off-line:	Routes precomputed, stored in tables
On-line:	Routing decisions made on the fly
Oblivious:	Path depends only on source and 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

Circuit switching: A circuit is established between source and destination before message is sent (as in old telephone networks)

Advantage: Fast transmission after the initial overhead

Packet switching: Packets are sent independently over possibly different paths

Advantage: Efficient use of channels due to sharing

Wormhole switching: Combines the advantages of circuit and packet switching

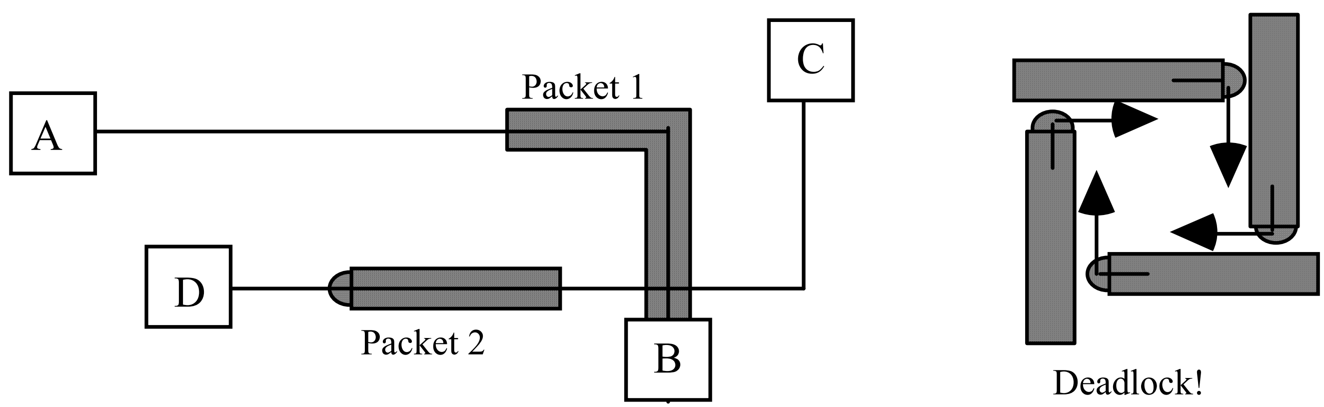


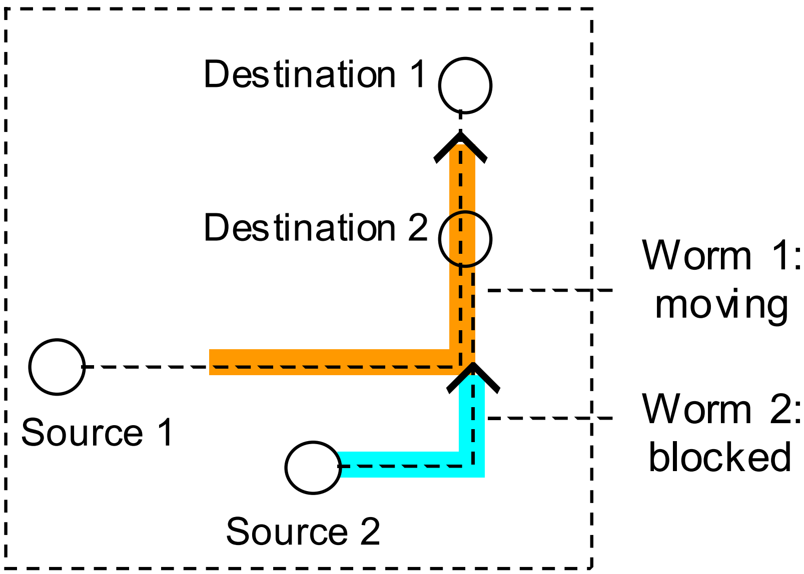
Fig. 10.10 Worms and deadlock in wormhole routing.

Route Selection in Wormhole Switching

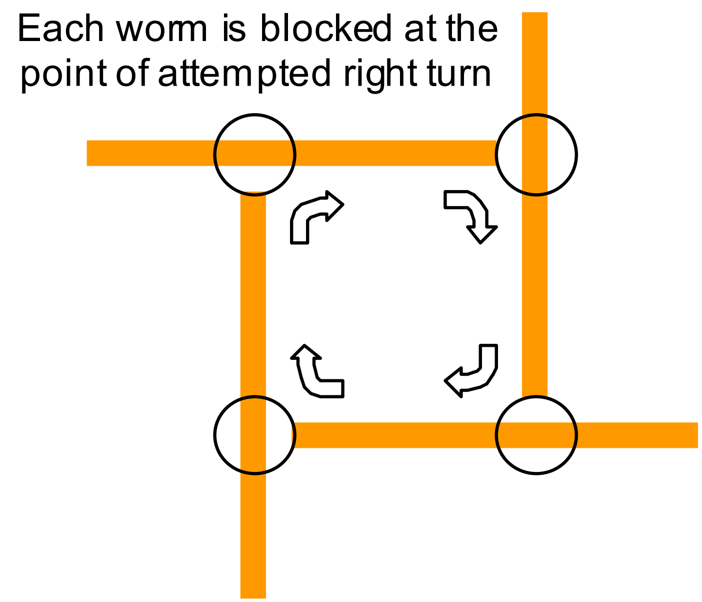
Routing algorithm must be simple to make the route selection quick

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

But . . . care must be taken to avoid excessive blocking and deadlock



(a) Two worms en route to their respective destinations



(b) Deadlock due to circular waiting of four blocked worms

Dealing with Conflicts

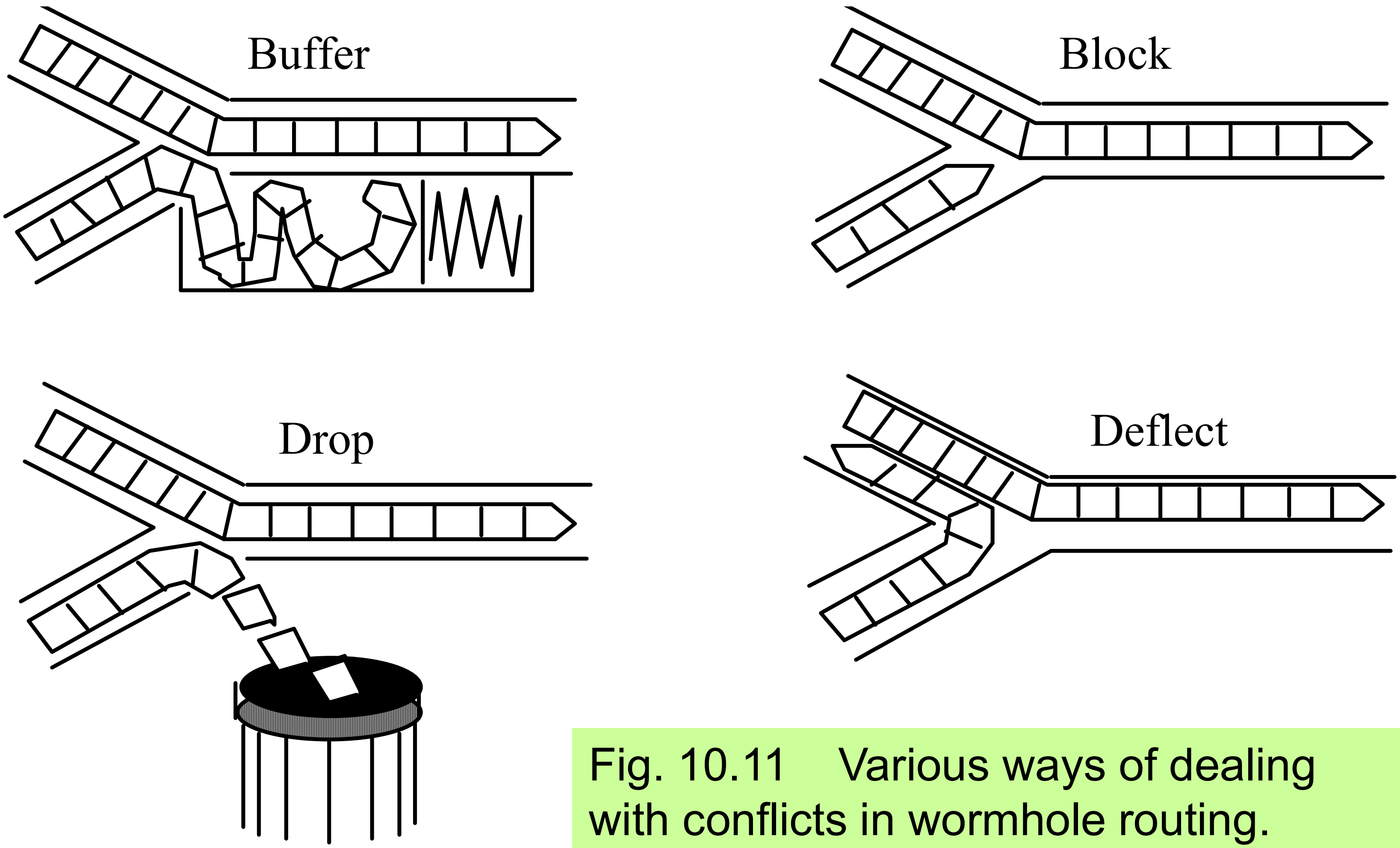
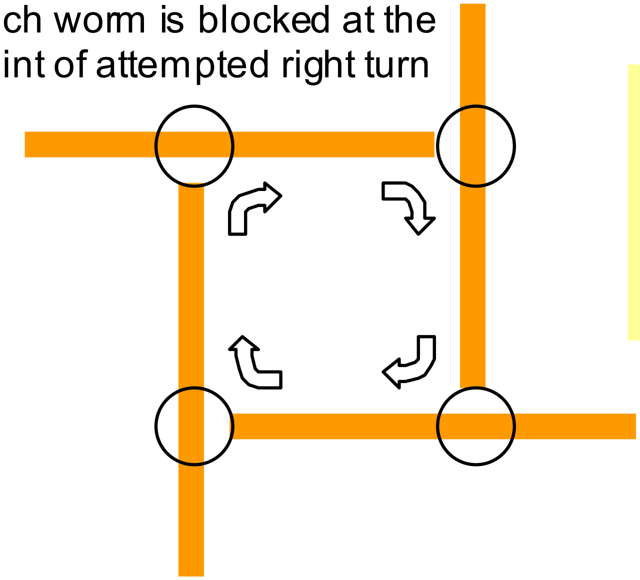


Fig. 10.11 Various ways of dealing with conflicts in wormhole routing.

Deadlock in Wormhole Switching

Each worm is blocked at the point of attempted right turn



Two strategies for dealing with deadlocks:

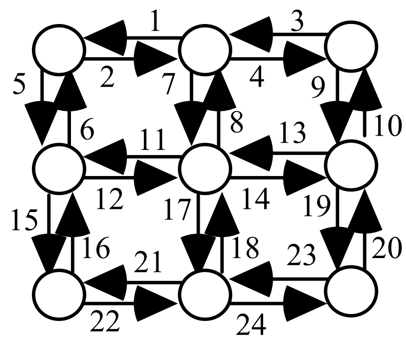
1. Avoidance

2. Detection and recovery

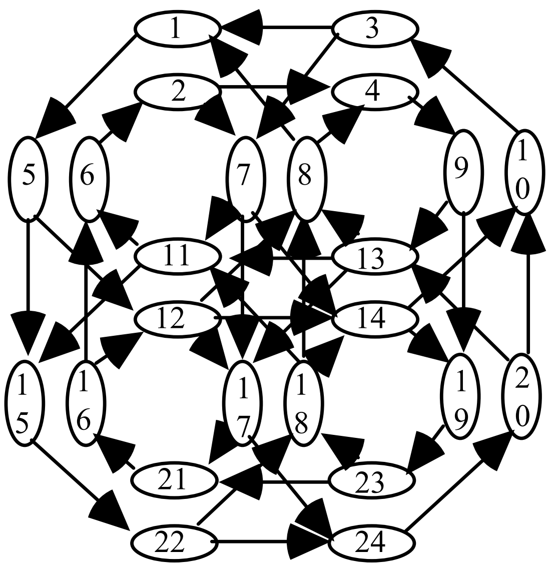
Deadlock avoidance requires a more complicated routing algorithm and/or more conservative routing decisions

. . . nontrivial performance penalties

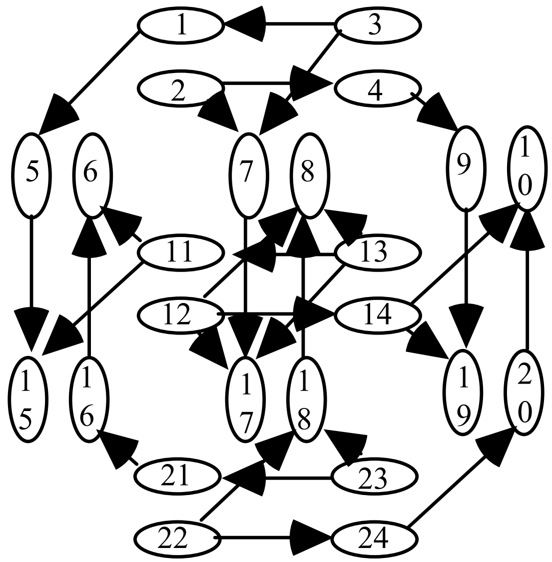
Deadlock Avoidance via Dependence Analysis



3-by-3 mesh with its links numbered



Unrestricted routing
(following shortest path)



E-cube routing
(row-first)

A sufficient condition for lack of deadlocks is to have a link dependence graph that is cycle-free

Less restrictive models are also possible; e.g., the turn model allows three of four possible turns for each worm

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

Deadlock Avoidance via Virtual Channels

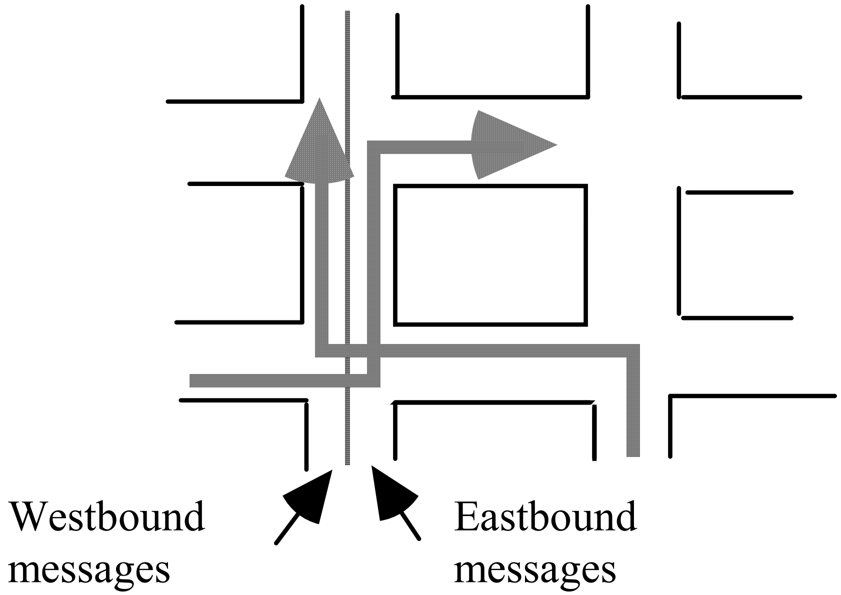
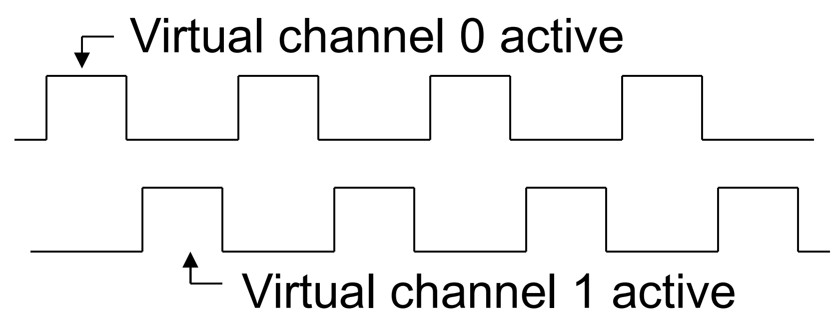
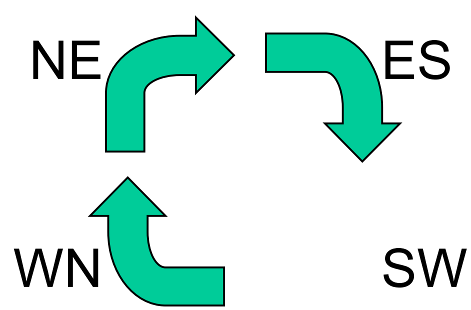


Fig. 10.13 Use of virtual channels for avoiding deadlocks.



Deadlock Avoidance via Routing Restrictions

Allow only three of the four possible turns



11 Numerical 2D Mesh Algorithms

Become more familiar with mesh/torus architectures by:

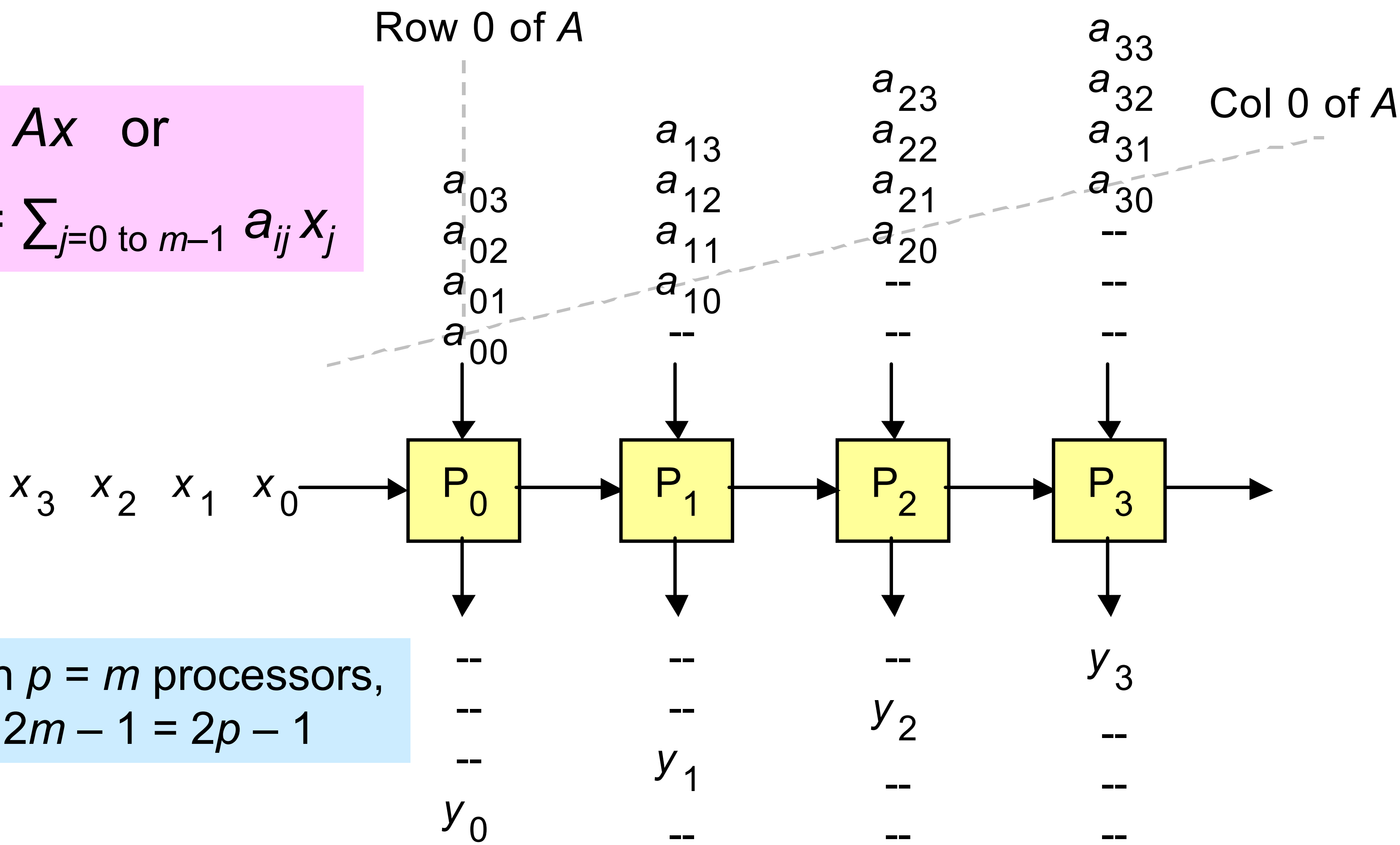
- Developing a number of useful numerical algorithms
- Studying seminumerical applications (graphs, images)

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

11.1 Matrix Multiplication

$y = Ax$ or

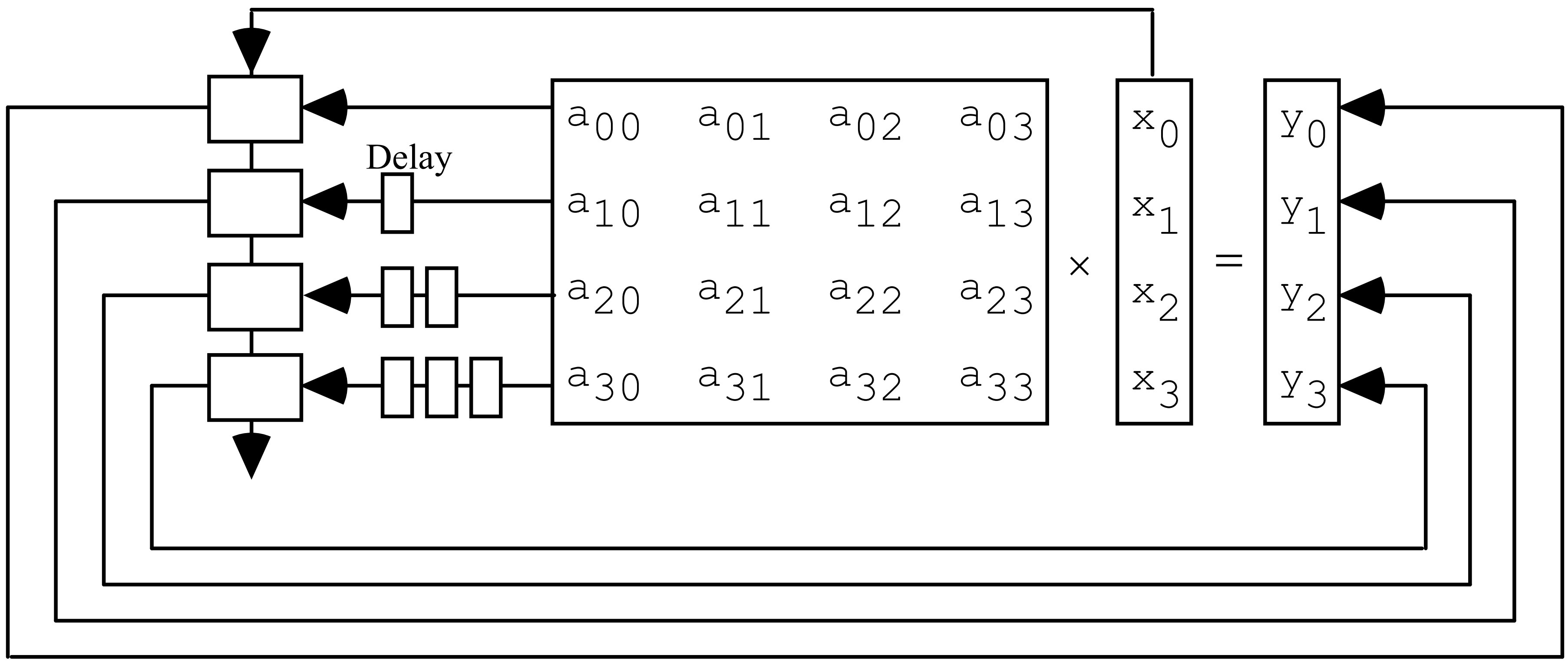
$y_i = \sum_{j=0 \text{ to } m-1} a_{ij} x_j$



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

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

Another View of Matrix-Vector Multiplication



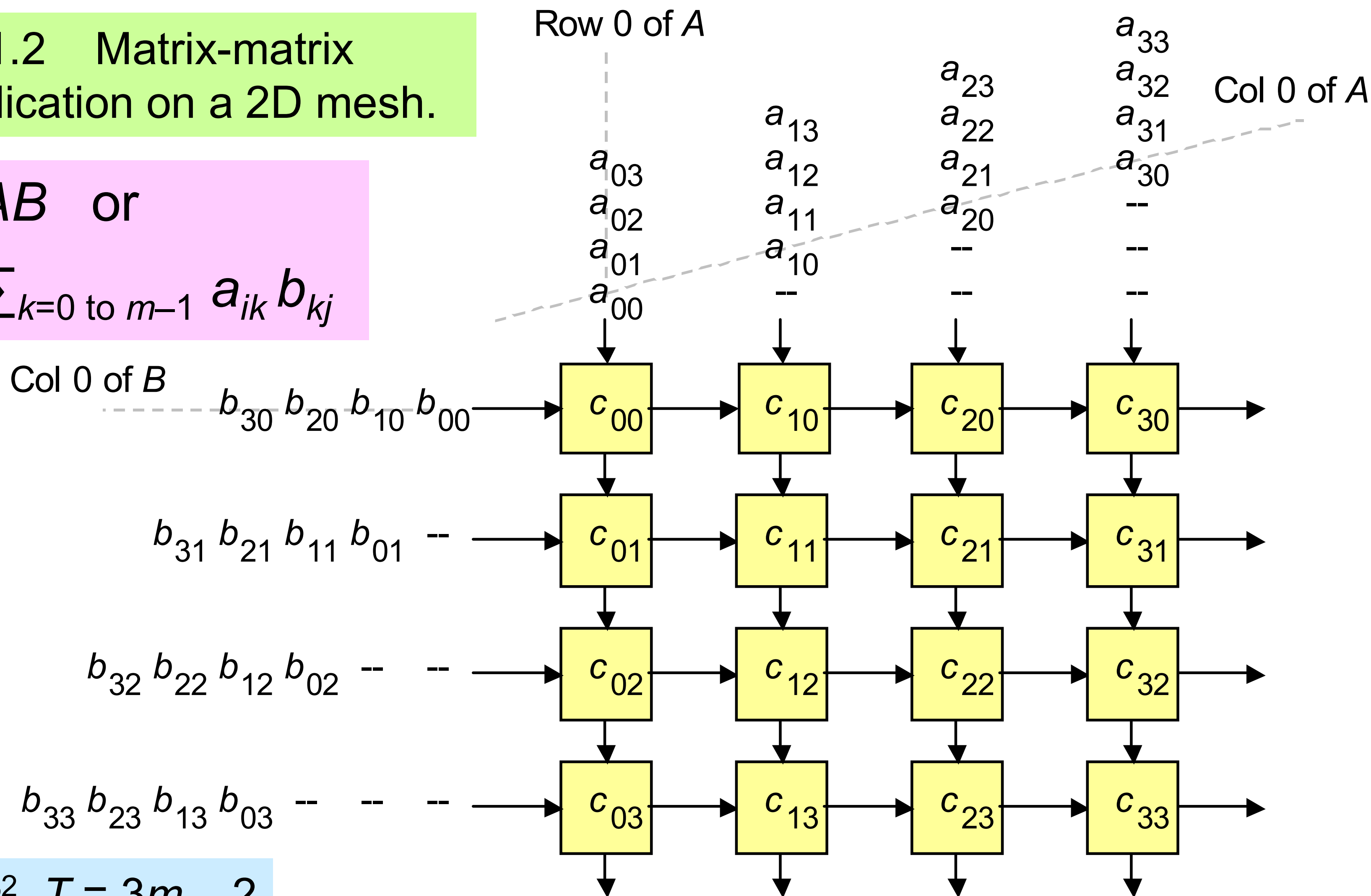
m-processor linear array for multiplying an *m*-vector by an $m \times m$ matrix.

Mesh Matrix Multiplication

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

$C = AB$ or

$c_{ij} = \sum_{k=0 \text{ to } m-1} a_{ik} b_{kj}$



$p = m^2$, $T = 3m - 2$

Matrix-Vector Multiplication on a Ring

$y = Ax$ or

$y_i = \sum_{j=0 \text{ to } m-1} a_{ij} x_j$

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

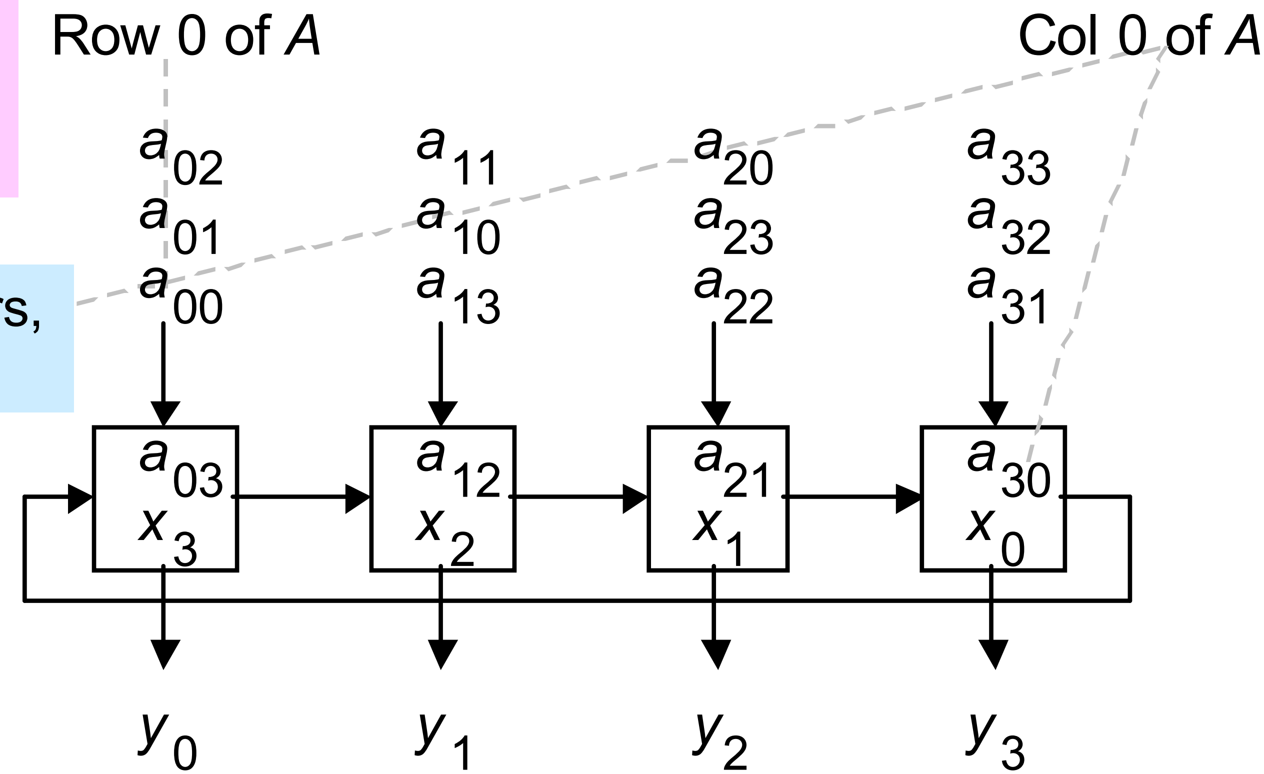


Fig. 11.3 Matrix-vector multiplication on a ring.

Torus Matrix Multiplication

Fig. 11.4 Matrix–matrix multiplication on a 2D torus.

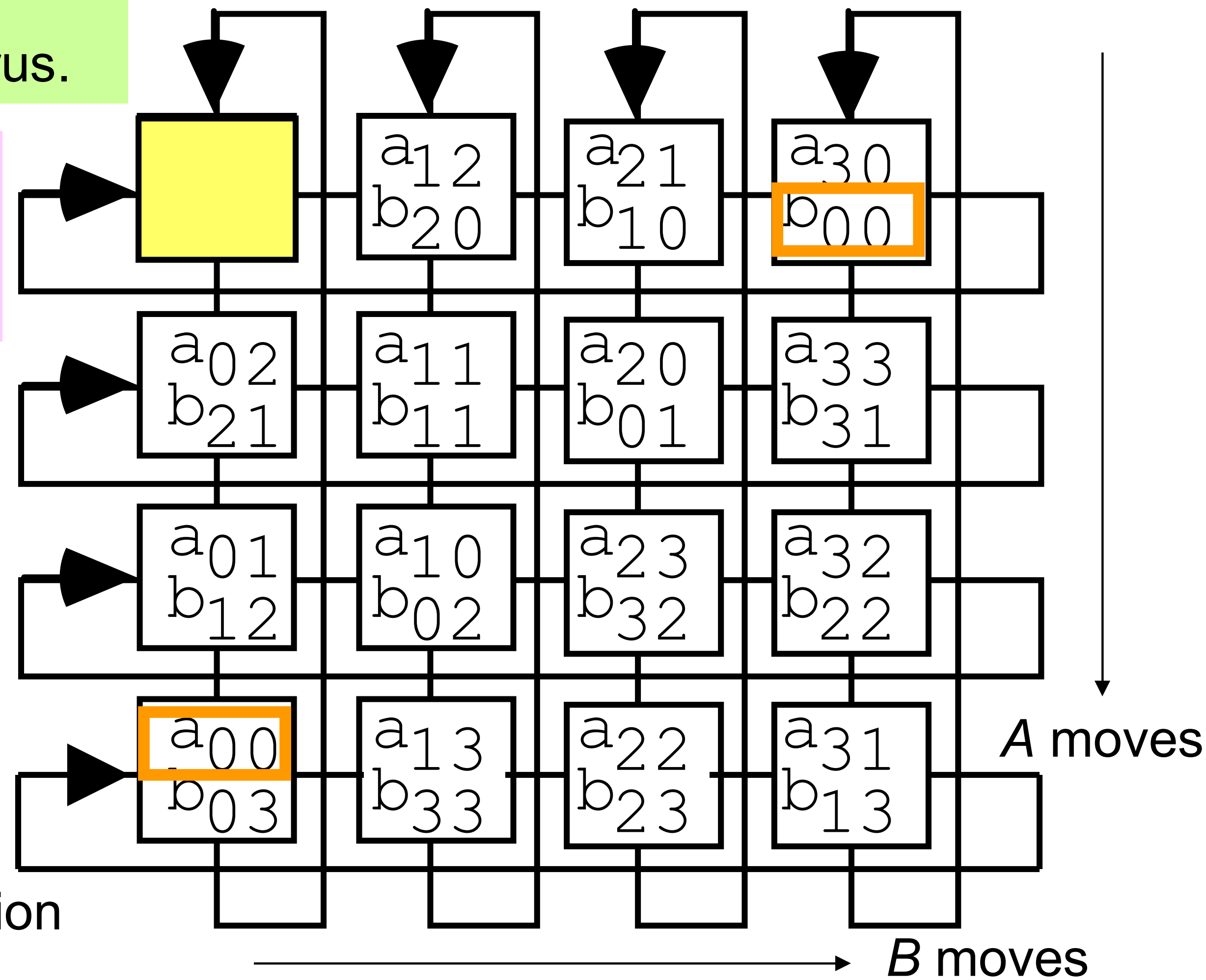
$C = AB$ or

$c_{ij} = \sum_{k=0 \text{ to } m-1} a_{ik} b_{kj}$

$p = m^2$, $T = m = p^{1/2}$

For $m > p^{1/2}$, use block matrix multiplication

Can gain efficiency from overlapping communication with computation



11.2 Triangular System of Linear Equations

Solution: Use forward (lower) or back (upper) substitution

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

Lower triangular: Find x_0 from the first equation, substitute in the second equation to find x_1, etc.

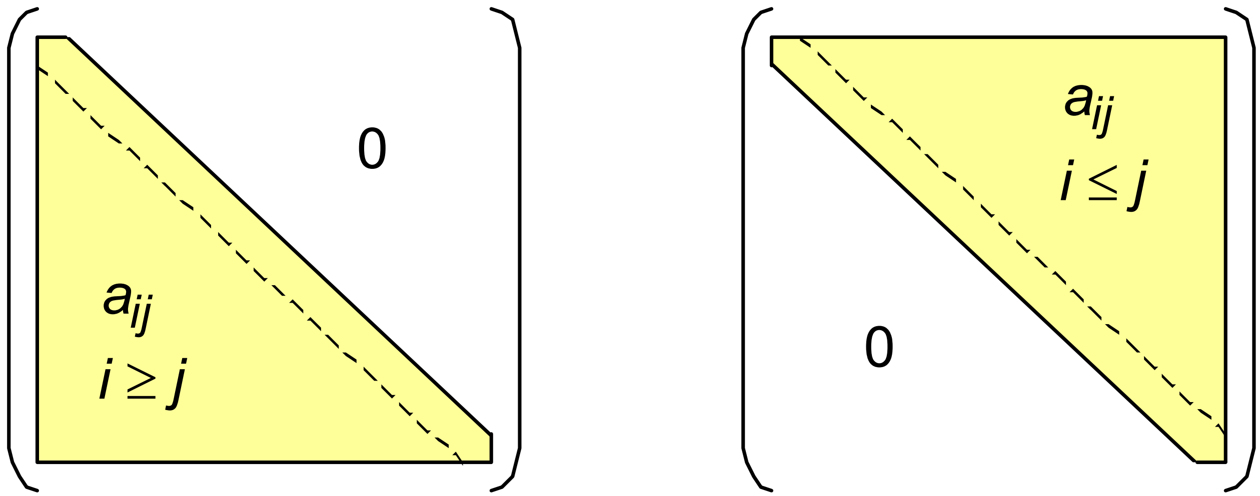


Fig. 11.5 Lower/upper triangular square matrix; if $a_{ii} = 0$ for all i, then it is strictly lower/upper triangular.

Forward Substitution on a Linear Array

$$a_{00}x_0 = b_0$$

$$a_{10}x_0 + a_{11}x_1 = b_1$$

.

.

.

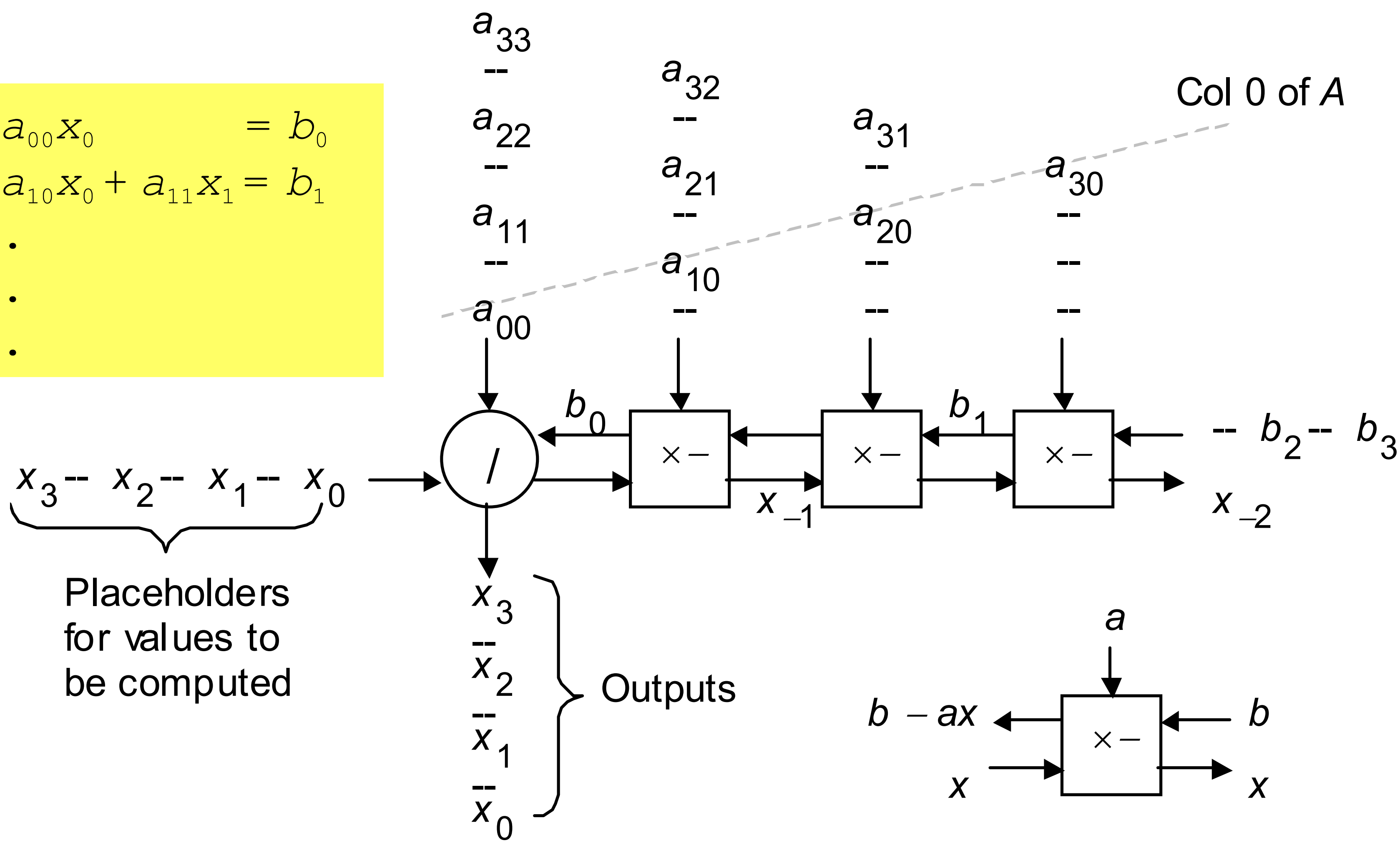
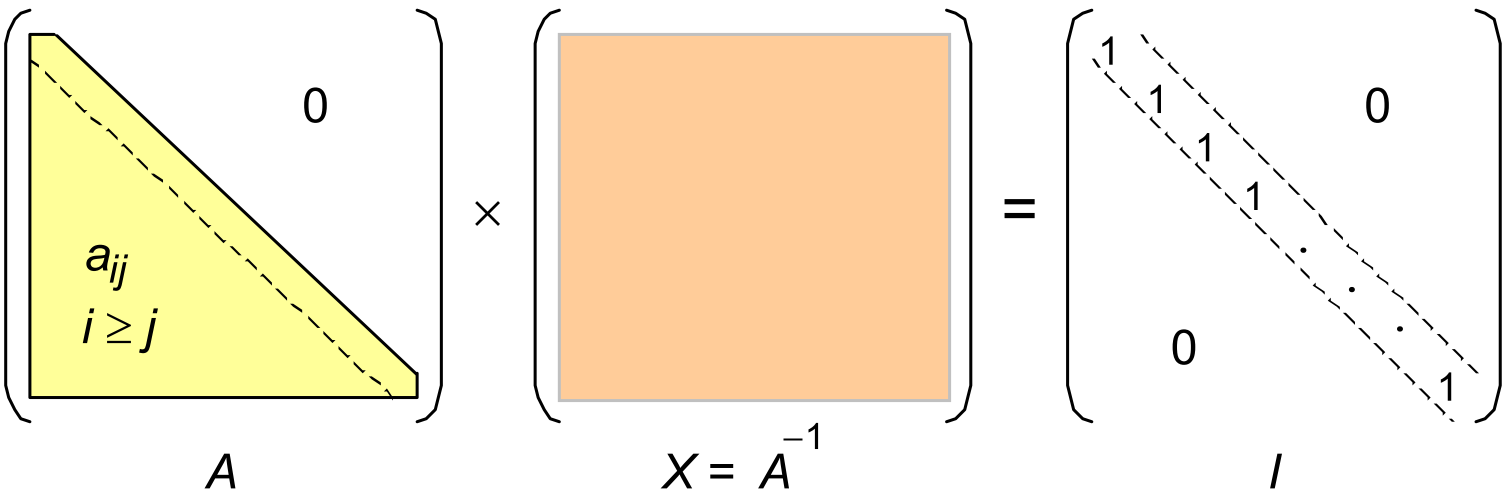
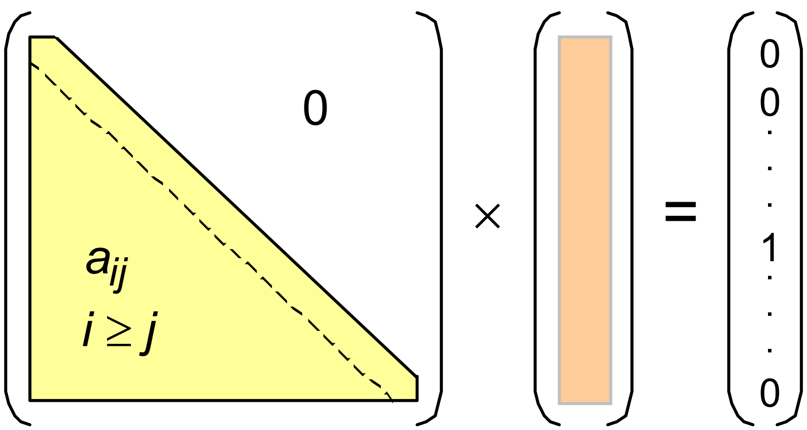


Fig. 11.6 Solving a triangular system of linear equations on a linear array.

Triangular Matrix Inversion: Algorithm





A multiplied by *i*th column of *X* yields *i*th column of the identity matrix *I* (solve *m* such triangular systems to invert *A*)

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

Triangular Matrix Inversion on a Mesh

$T = 3m - 2$

Can invert two matrices using one more step

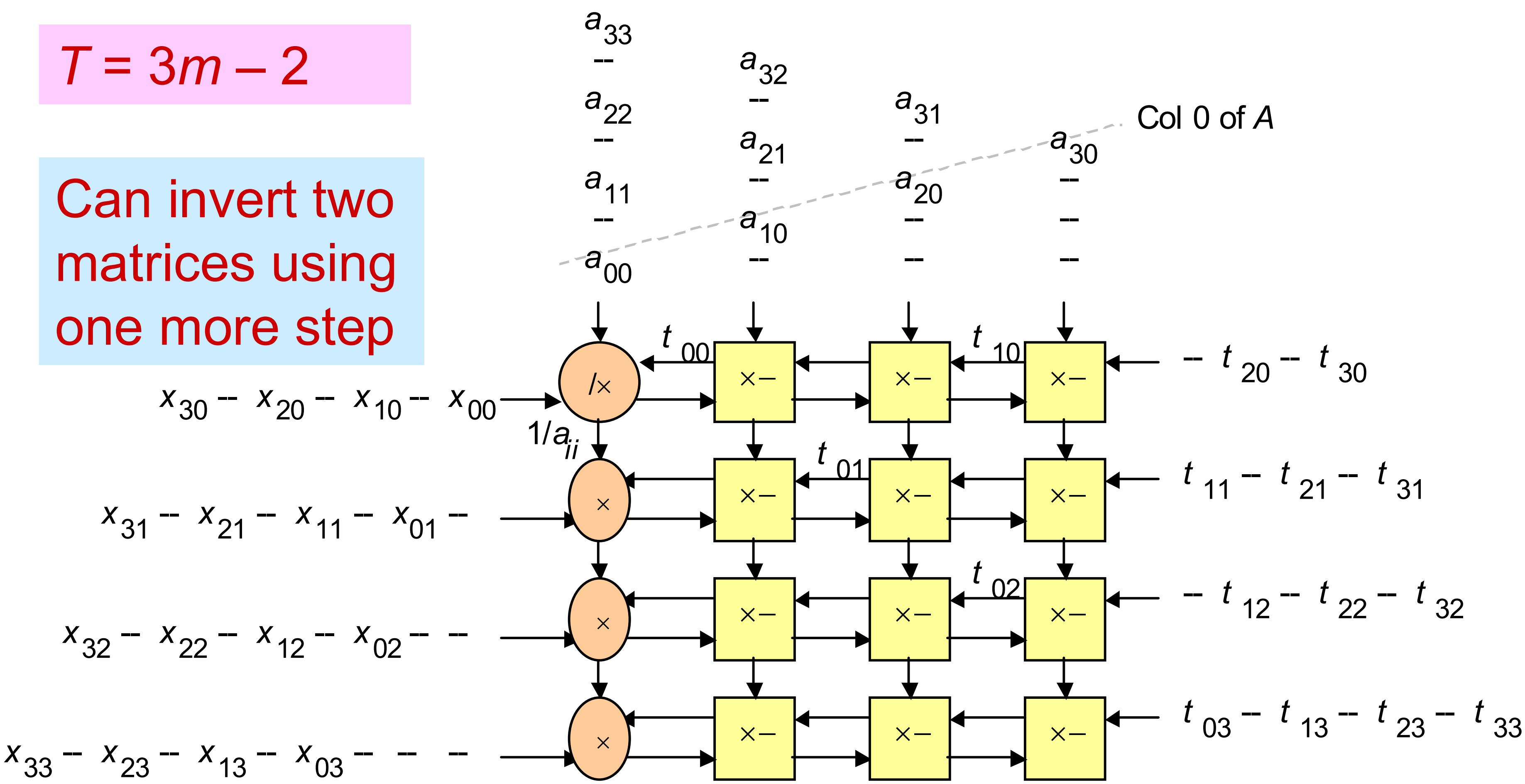
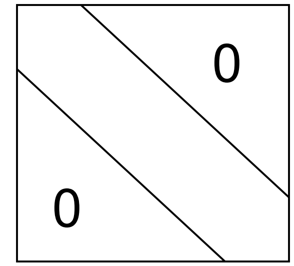


Fig. 11.8 Inverting a lower triangular matrix on a 2D mesh.

11.3 Tridiagonal System of Linear Equations

$$l_0\, x_{-1} + d_0\, x_0 + u_0\, x_1 = b_0$$
$$l_1\, x_0 + d_1\, x_1 + u_1\, x_2 = b_1$$
$$l_2\, x_1 + d_2\, x_2 + u_2\, x_3 = b_2$$
$$\vdots$$

Special case of a band matrix



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

Fig. 11.9 A tridiagonal system of linear equations.

Other Types of Diagonal Matrices

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

Matrices with more than three diagonals can be viewed as tridiagonal blocked matrices

$$
\begin{matrix}
x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & x & x & x & x & x & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & x & x & x & x & x & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & x & x & x & x & x & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & x & x & x & x & x & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x & x & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x & x \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x
\end{matrix}
$$

A pentadiagonal matrix.

Odd-Even Reduction

$$l_0\, x_{-1} + d_0\, x_0 + u_0\, x_1 = b_0$$
$$l_1\, x_0 + d_1\, x_1 + u_1\, x_2 = b_1$$
$$l_2\, x_1 + d_2\, x_2 + u_2\, x_3 = b_2$$
$$l_3\, x_2 + d_3\, x_3 + u_3\, x_4 = b_3$$
$$\vdots$$

Use odd equations to find odd-indexed variables in terms of even-indexed ones

$$d_1\, x_1 = b_1 - l_1\, x_0 - u_1\, x_2$$
$$d_3\, x_3 = b_3 - l_3\, x_2 - u_3\, x_4$$
$$\vdots$$

The six divides are replaceable with one reciprocation per equation, to find $1/d_j$ for odd j, and six multiplies

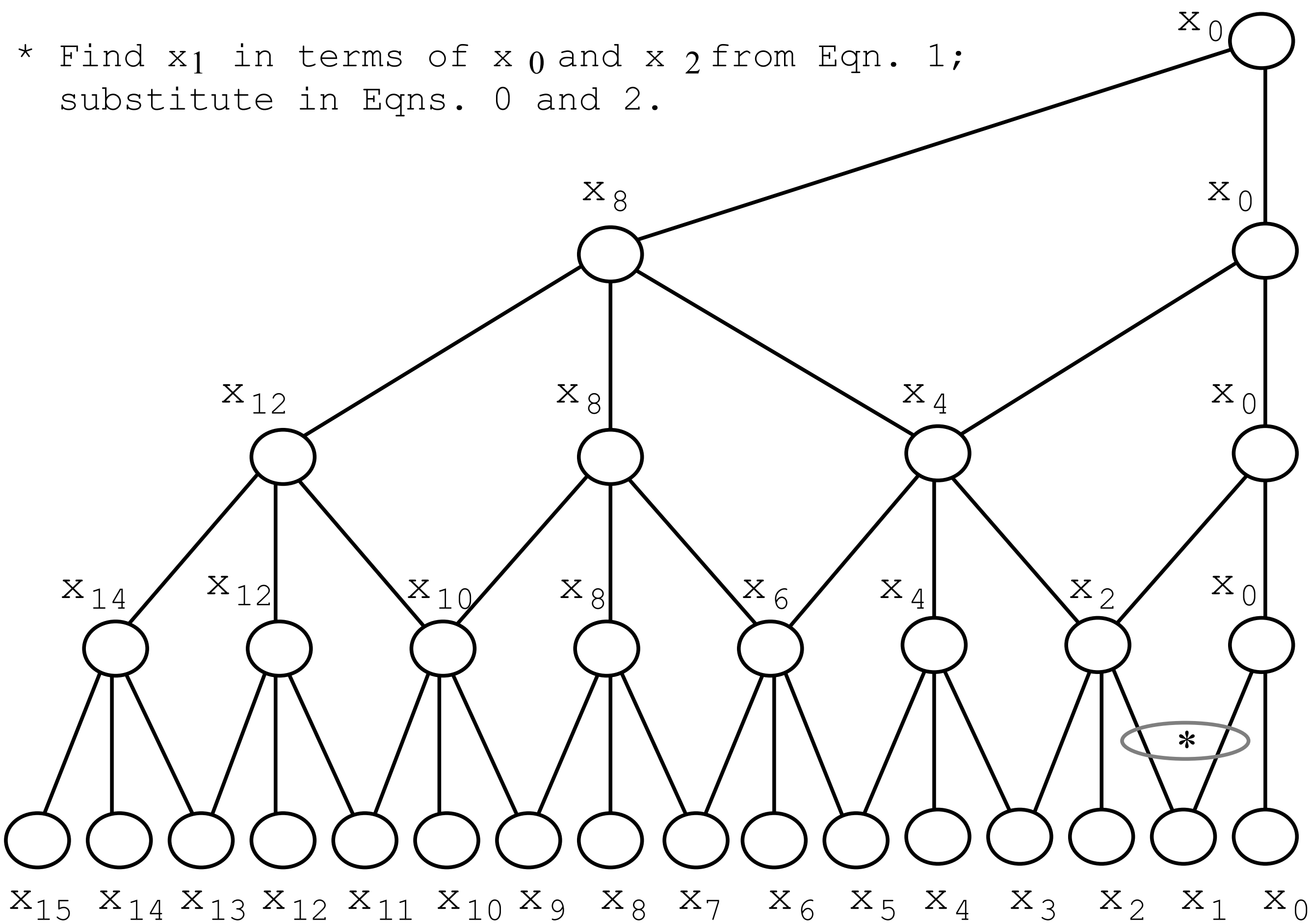
Substitute in even equations to get a tridiagonal system of half the size

$$L_0\, x_{-2} + D_0\, x_0 + U_0\, x_2 = B_0$$
$$L_2\, x_0 + D_2\, x_2 + U_2\, x_4 = B_2$$
$$L_4\, x_2 + D_4\, x_4 + U_4\, x_6 = B_4$$
$$\vdots$$

$$L_i = -\, l_i\, l_{i-1}/d_{i-1}$$
$$D_i = d_i - l_i\, u_{i-1}/d_{i-1} - u_i\, l_{i+1}/d_{i+1}$$
$$U_i = -\, u_i\, u_{i+1}/d_{i+1}$$
$$B_i = b_i - l_i\, b_{i-1}/d_{i-1} - u_i\, b_{i+1}/d_{i+1}$$

Sequential solution:
$T(m) = T(m/2) + cm = 2cm$

Architecture for Odd-Even Reduction



Parallel solution:

$T(m) = T(m/2) + c$

$= c \log_2 m$

Because we ignored communication, our analysis is valid for PRAM or for an architecture whose topology matches that of Fig. 11.10.

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

Odd-Even Reduction on a Linear Array

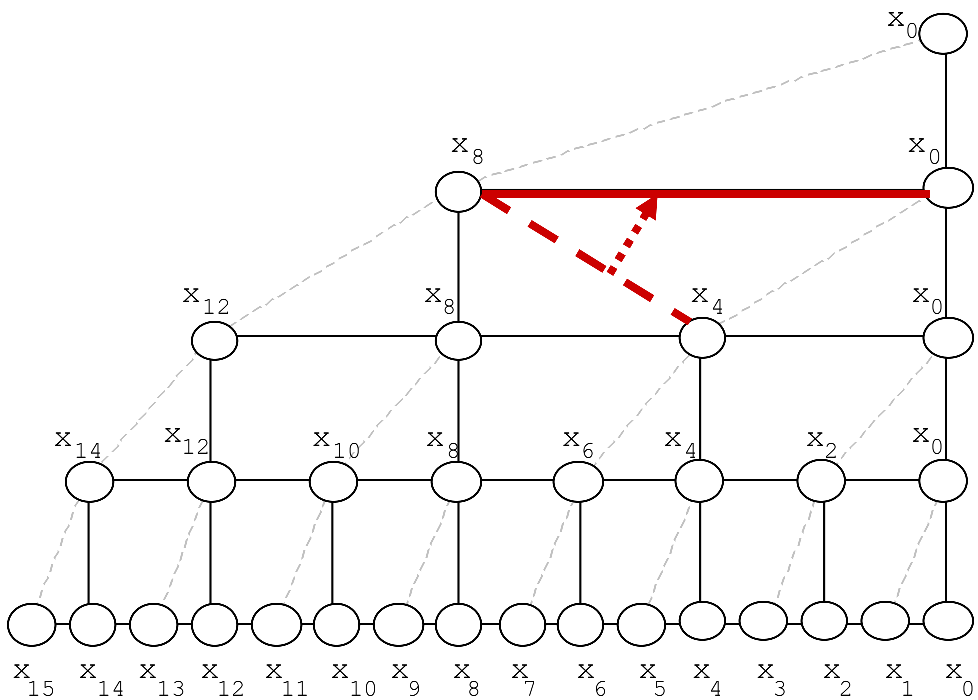
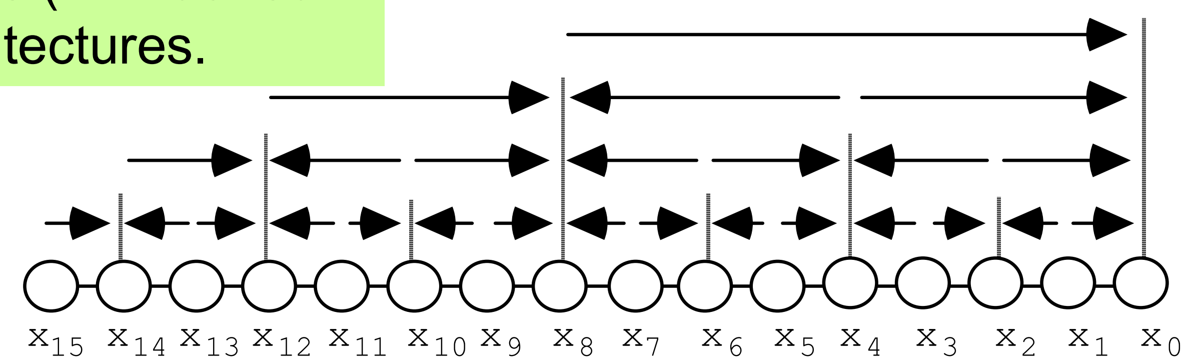


Fig. 11.11 Binary X-tree (with dotted links) and multigrid architectures.

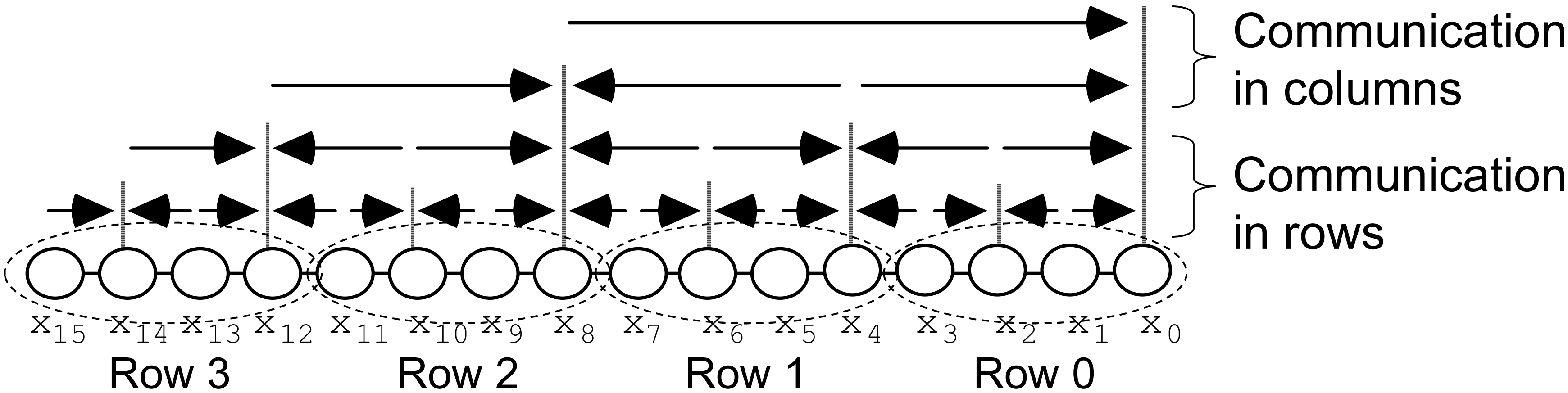
Architecture of Fig. 11.10 can be modified to binary X-tree and then simplified to 2D multigrid

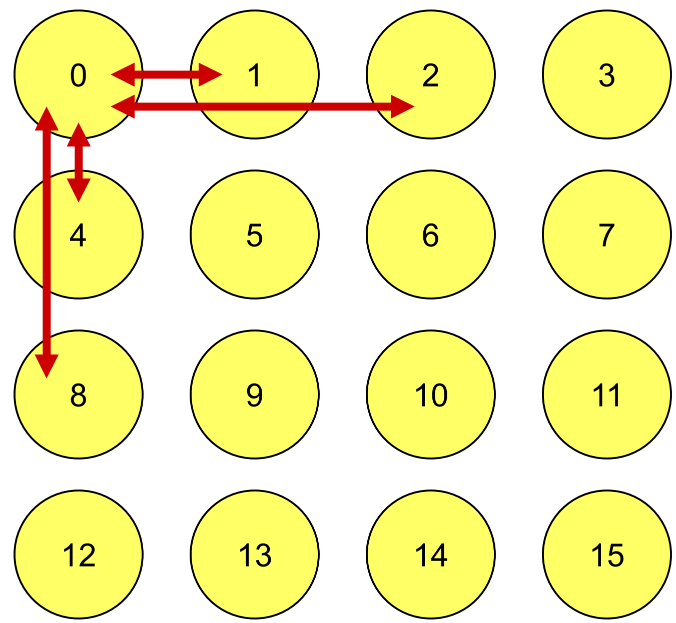
Communication time on linear array:

$$T(m) = 2(1 + 2 + \ldots + m/2)$$
$$= 2m - 2$$



Odd-Even Reduction on a 2D Mesh





Communication time on 2D mesh:

$T(m) \cong 2[2(1 + 2 + \ldots + m^{1/2}/2)]$

$\cong 2m^{1/2}$

11.4 Arbitrary System of Linear Equations

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

$$\boxed{Ax = b}$$

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

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

A — b for system 1 — b for system 2

Divide row 0 by 2; subtract 3 times from row 1 (pivoting oper)

Gaussian elimination

Extended matrix $A' = \begin{bmatrix} 1 & 2 & -3.5 & 1.5 & 3.5 \\ 0 & 0 & 0.5 & -0.5 & -2.5 \\ 0 & 5 & -7.5 & 7.5 & 2.5 \end{bmatrix}$

Extended matrix $A' = \begin{bmatrix} 1 & 0 & 0 & -2 & 0 \\ 0 & 1 & 0 & 0 & -7 \\ 0 & 0 & 1 & -1 & -5 \end{bmatrix}$

Repeat until identity matrix appears in first *n* columns; read solutions from remaining columns

Performing One Step of Gaussian Elimination

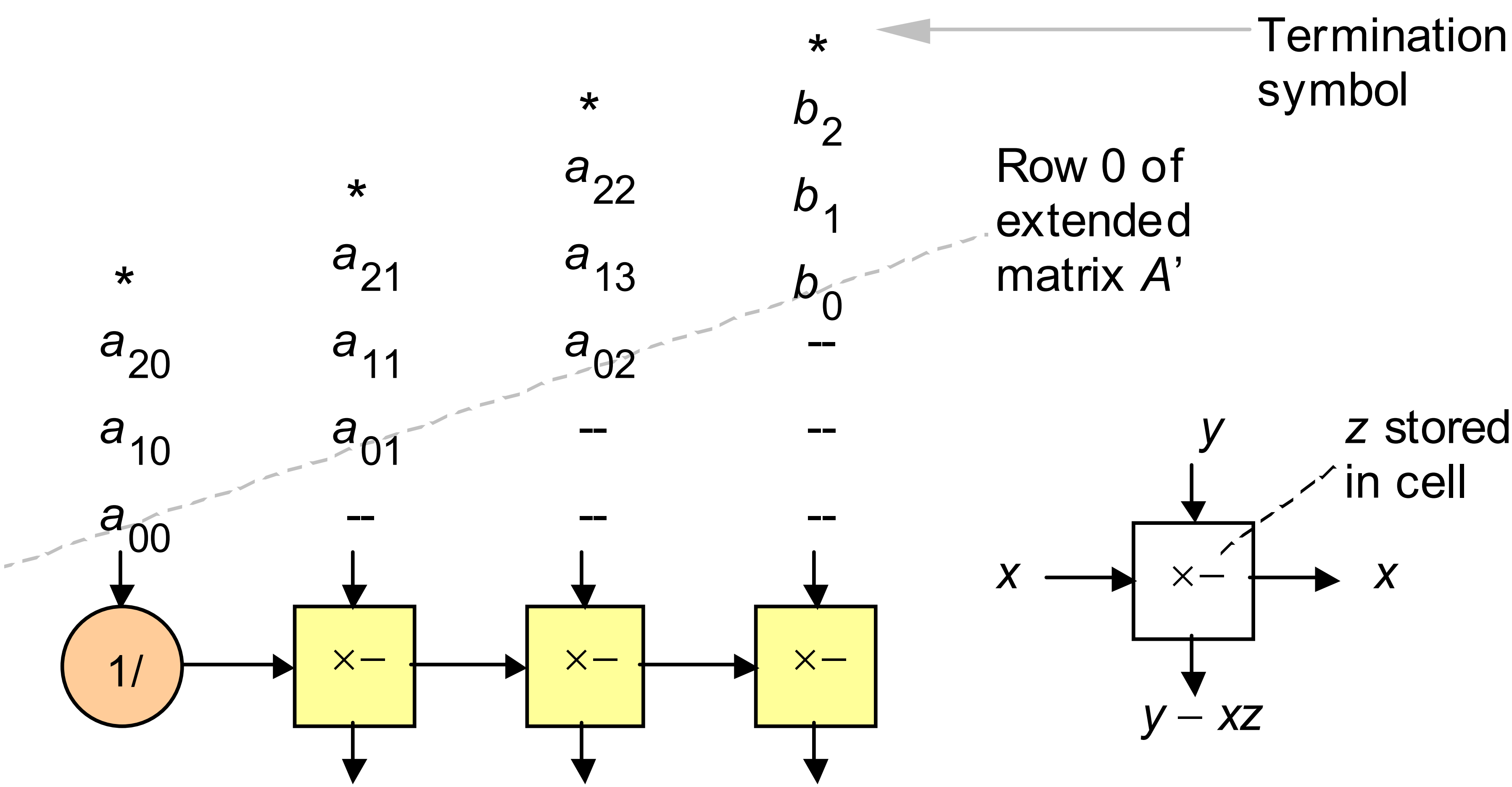


Fig. 11.12 A linear array performing the first phase of Gaussian elimination.

Gaussian Elimination on a 2D Mesh

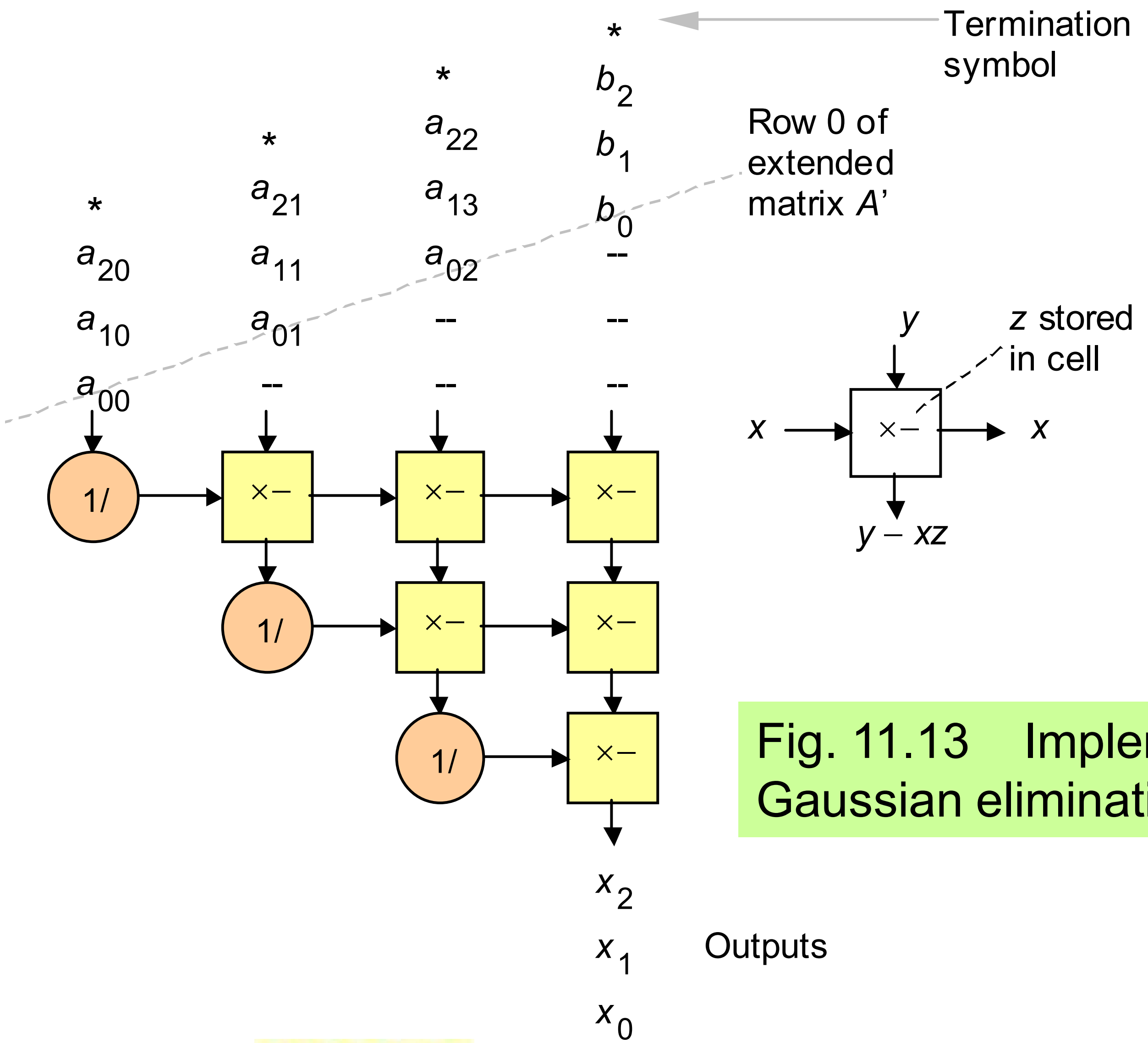


Fig. 11.13 Implementation of Gaussian elimination on a 2D array.

Matrix Inversion on a 2D Mesh

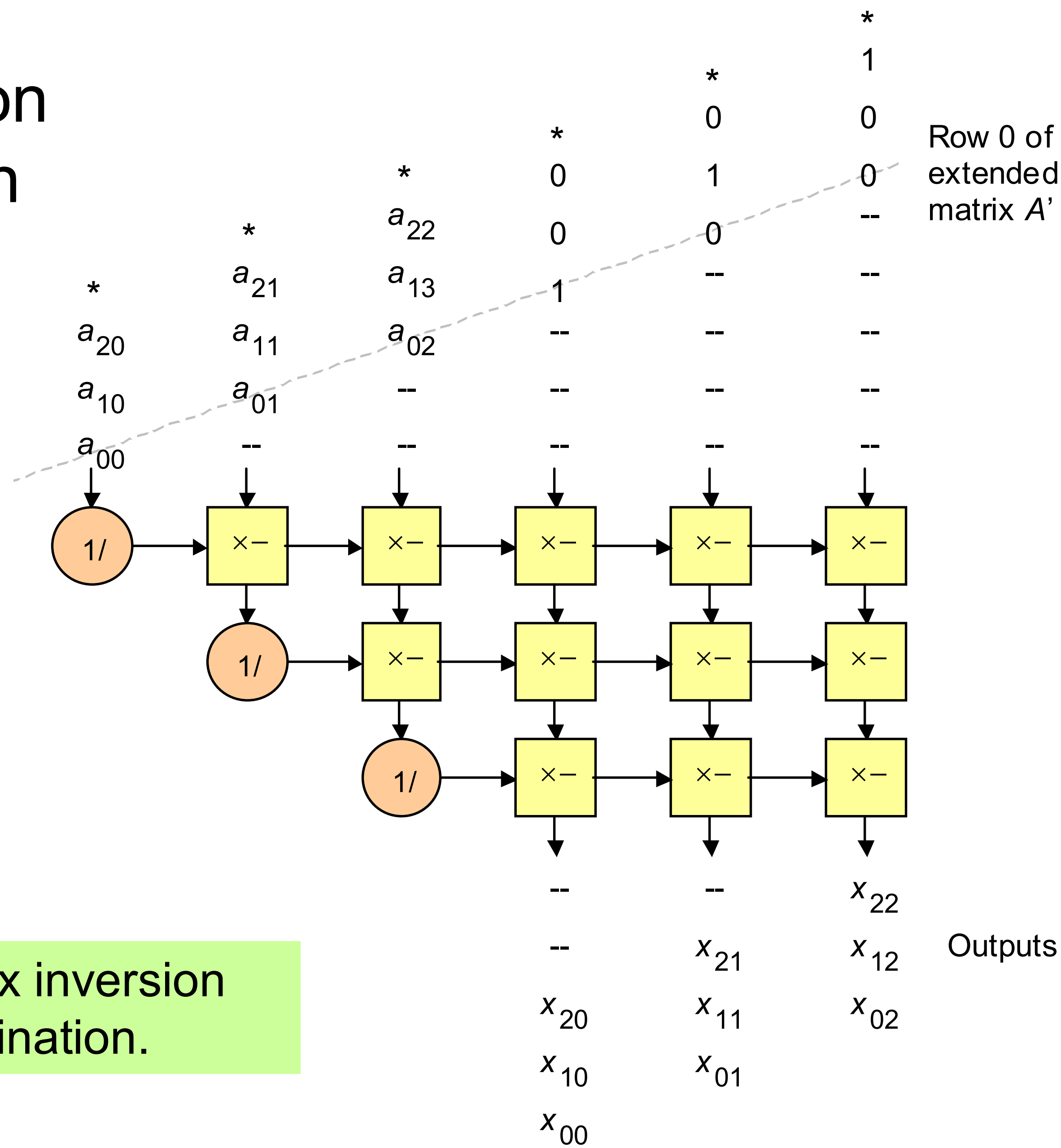


Fig. 11.14 Matrix inversion by Gaussian elimination.

Jacobi Methods

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

$\boxed{Ax = b}$

Solution: $x_0 = -2$, $x_1 = 0$, $x_2 = -1$

Use each equation to find one of the variables in terms of all others

$$\begin{aligned} x_0 &= -2.000x_1 + 3.500x_2 + 1.500 \\ x_1 &= -0.500x_0 + 1.667x_2 + 0.667 \\ x_2 &= -0.250x_0 + 0.750x_1 - 1.500 \end{aligned}$$

Iterate: Plug in estimates for the unknowns on the right-hand side to find new estimates on the left-hand side

Example: Estimate $x_0 = 1$, $x_1 = 1$, $x_2 = 1$

$$\begin{aligned} x_0 &= -2.000 + 3.500 + 1.500 = 3.000 \\ x_1 &= -0.500 + 1.667 + 0.667 = 1.834 \\ x_2 &= -0.250 + 0.750 - 1.500 = -1.000 \end{aligned}$$

Jacobi Relaxation and Overrelaxation

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

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

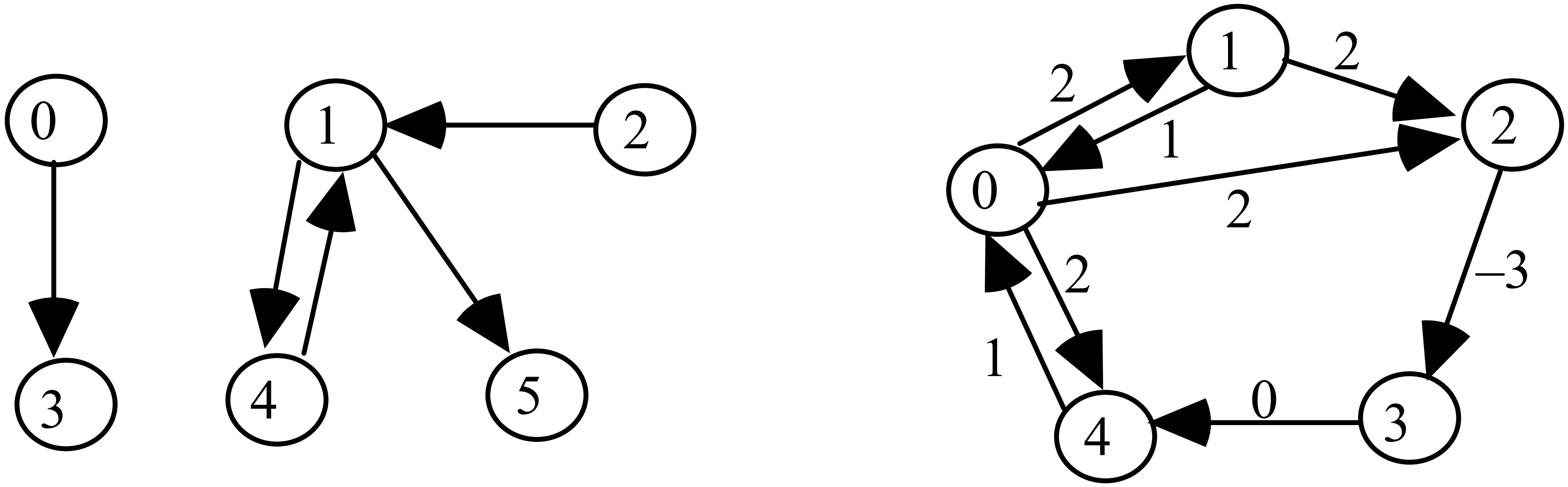
On an m-processor linear array, each iteration takes $O(m)$ steps. The number of iterations needed is $O(\log m)$ if certain conditions are satisfied, leading to $O(m \log m)$ average time.

A variant: **Jacobi overrelaxation**

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

For $\gamma = 1$, the method is the same as Jacobi relaxation
For smaller γ, overrelaxation may offer better performance

11.5 Graph Algorithms



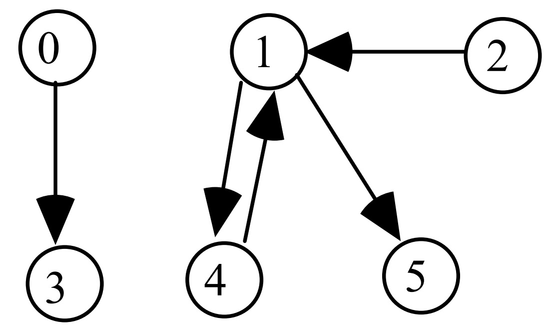
$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

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

Fig. 11.15 Matrix representation of directed graphs.

Transitive Closure of a Graph

$A^0 = I$	Paths of length 0 (identity matrix)
$A^1 = A$	Paths of length 1
$A^2 = A \times A$	Paths of length 2
$A^3 = A^2 \times A$	Paths of length 3 etc.



Compute "powers" of A via matrix multiplication, but use AND/OR in lieu of multiplication/addition

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Transitive closure of G has the adjacency matrix

$$A^* = A^0 + A^1 + A^2 + \ldots$$

$A^*_{ij} = 1$ iff node j is reachable from node i

Graph G with adjacency matrix A

Powers need to be computed up to A^{n-1} (why?)

Transitive Closure Algorithm

Initialization: Insert the edges (i, i), $0 \le i \le n - 1$, into the graph

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

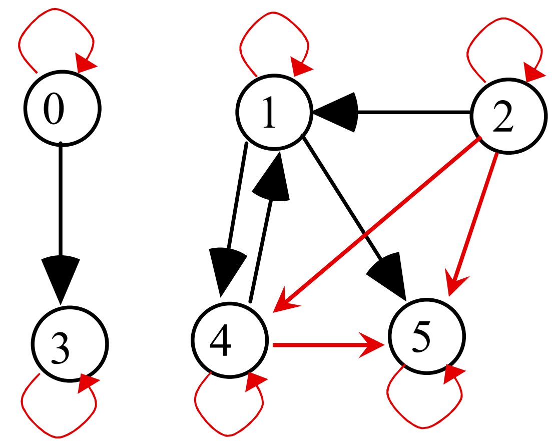
Phase 1 Insert the edge (i, j) into the graph if $(i, 1)$ and $(1, j)$ are in the graph

.
.
.

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 $n - 1$ Graph $A^{(n-1)}$ is the answer A^*



$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Graph G with adjacency matrix A

Transitive Closure on a 2D Mesh

The key to the algorithm is to ensure that each phase takes constant time; overall O(n) steps. This would be optimal on an $n \times n$ mesh because the best sequential algorithm needs O(n^3) time.

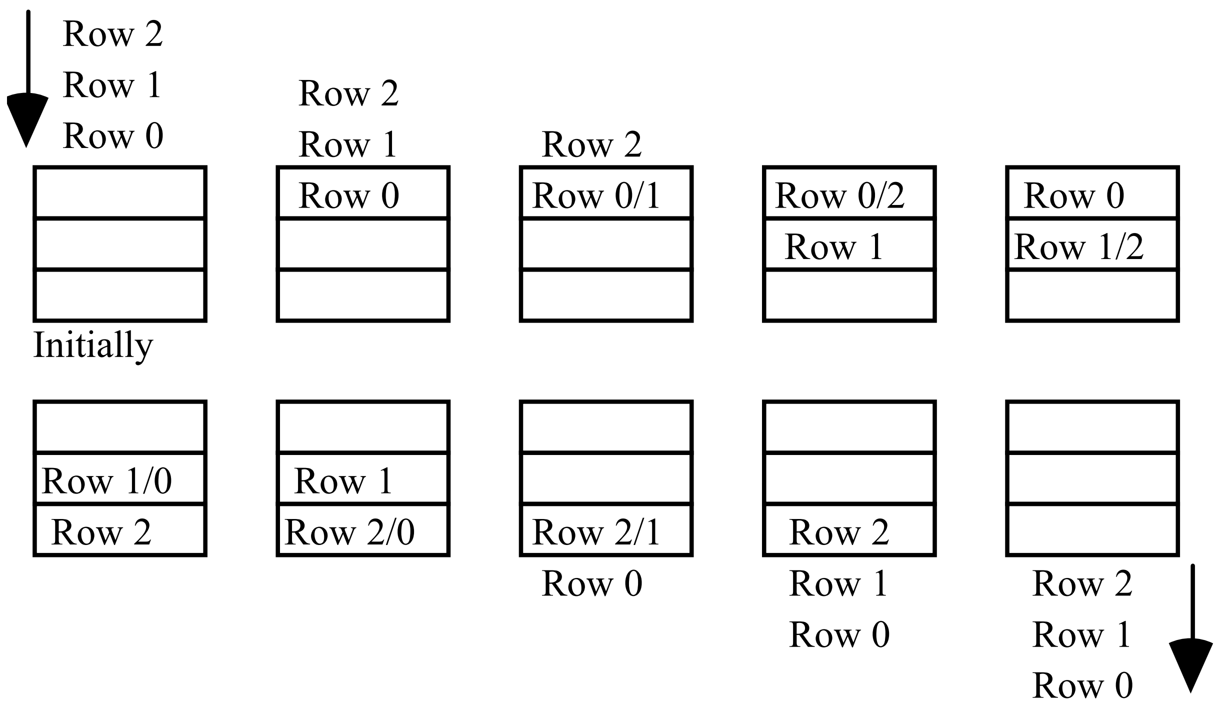
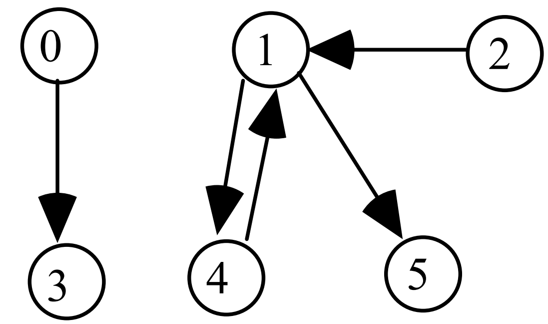


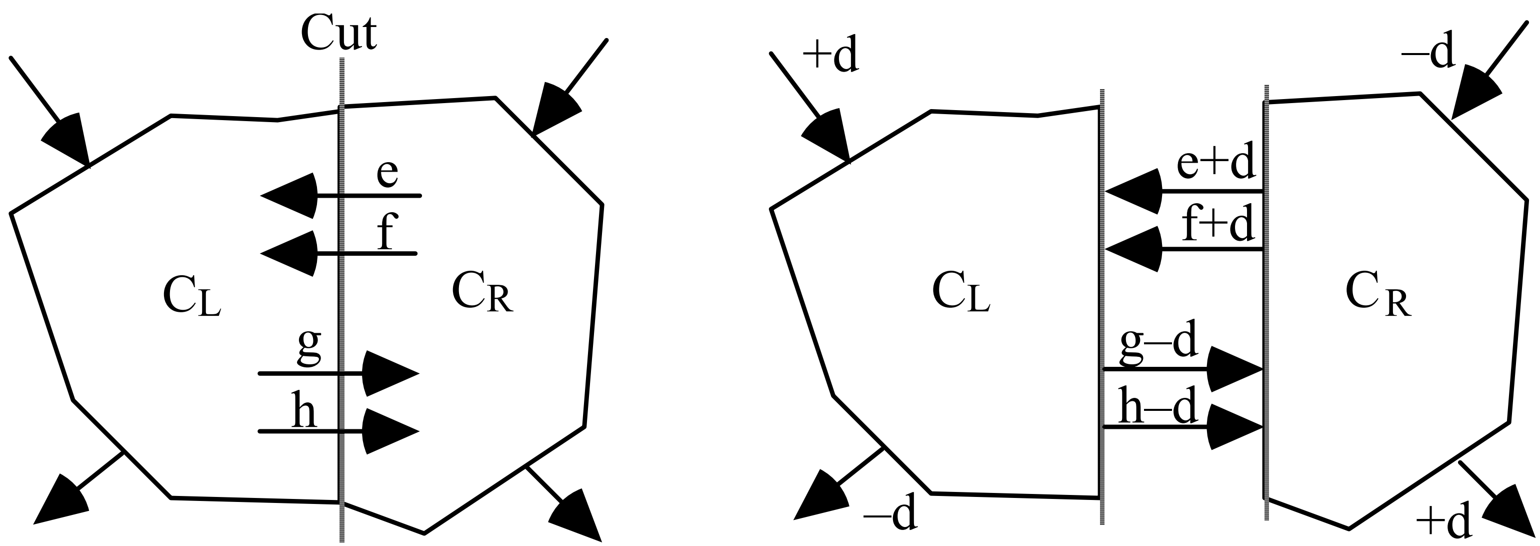
Fig. 11.16 Transitive closure algorithm on a 2D mesh.



$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

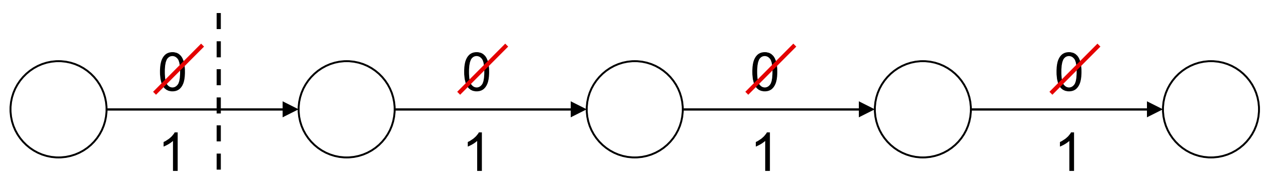
Graph G with adjacency matrix A

Elimination of Broadcasting via Retiming



Original delays   Adjusted delays

Example of systolic 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]



Systolic Retiming for Transitive Closure

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)

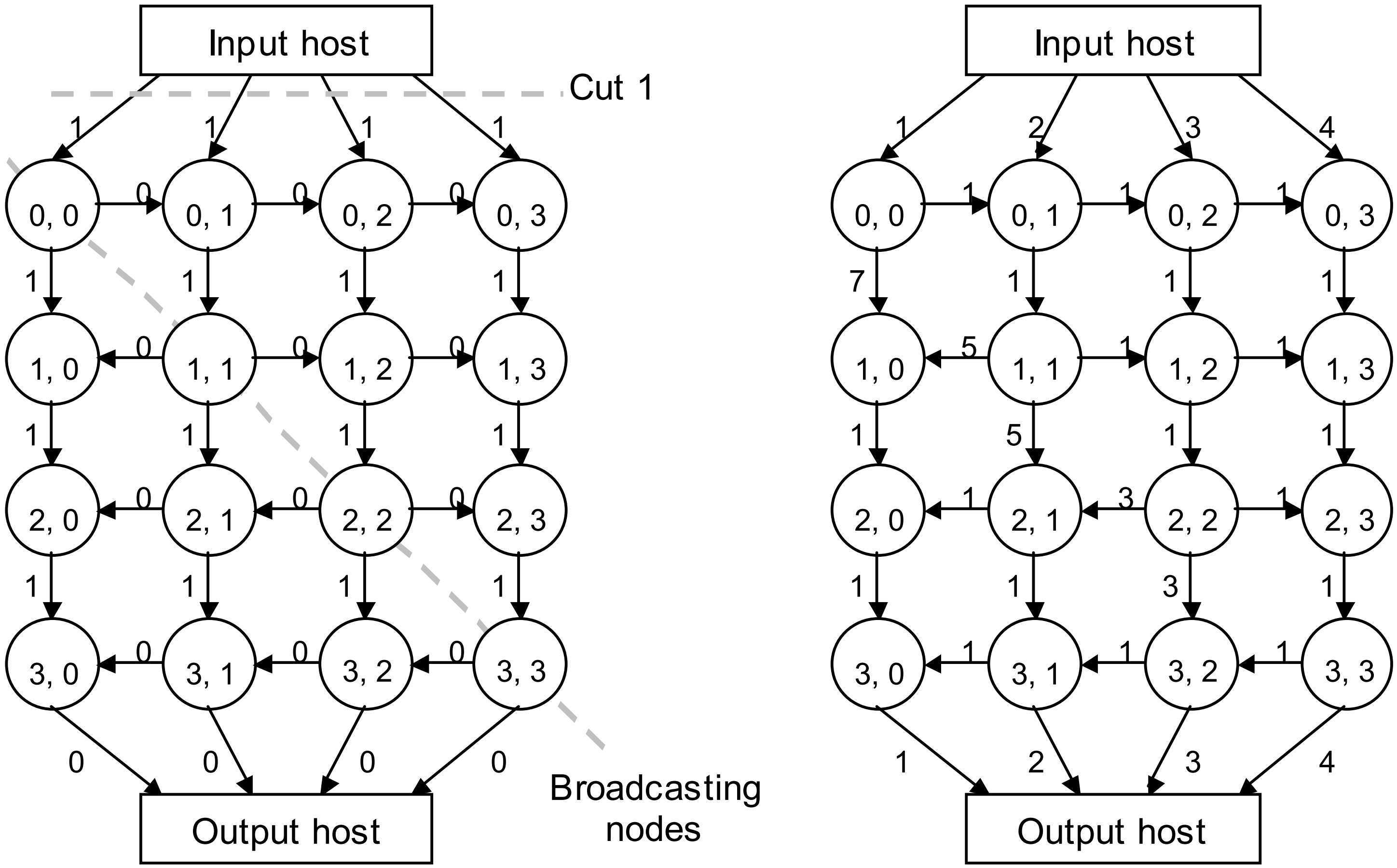
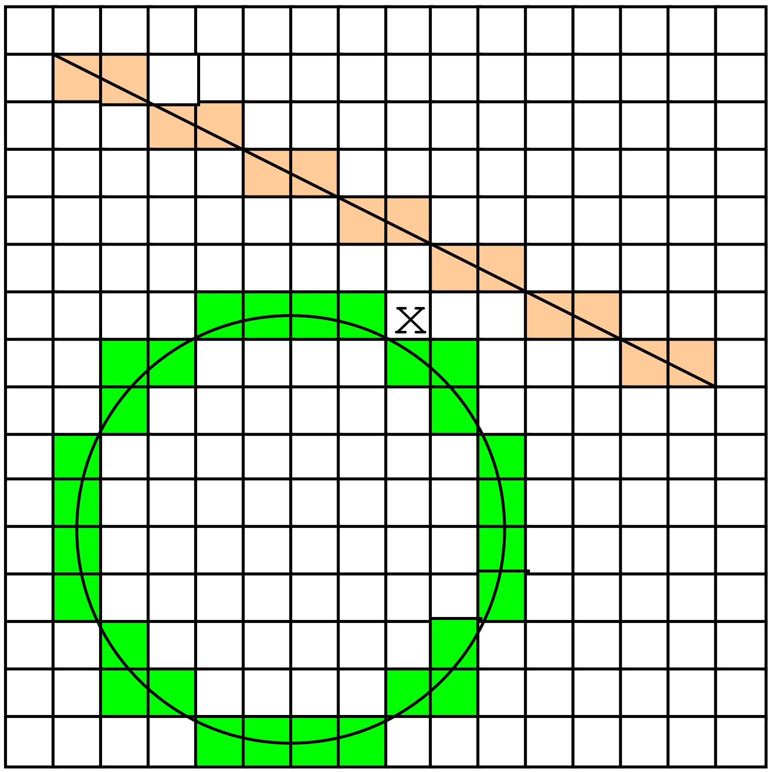


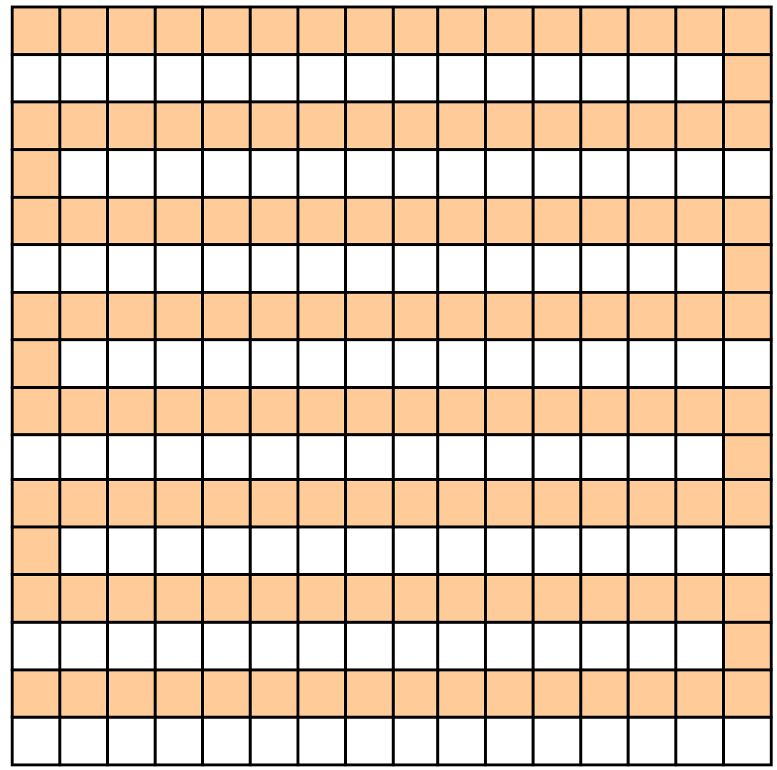
Fig. 11.17 Systolic retiming to eliminate broadcasting.

11.6 Image Processing Algorithms

Labeling connected components in a binary image (matrix of pixels)



The reason for considering diagonally adjacent pixels parts of the same component.



Worst-case component showing that a naïve “propagation” algorithm may require O(p) time.

Recursive Component Labeling on a 2D Mesh

C 0	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
C 49	1	1	0	0	0	0	0	1

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

C 3	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}
C 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.

$$T(p) = T(p/4) + O(p^{1/2}) = O(p^{1/2})$$

Levialdi's Algorithm

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

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

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

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

Initial image

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

After step 1

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

After step 2

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

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

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

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

After step 6

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

After step 7

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	1	1	0	0	0
0	0	0	1	1	0	0	*

After step 8

Analysis and Proof of Levialdi's Algorithm

$\begin{matrix} 0 & 1 \\ 1 & \boxed{0} \end{matrix}$	$\begin{matrix} 1 & 1 \\ 1 & \boxed{0} \end{matrix}$	`0 is changed to 1` `if N = W = 1`	
	$\begin{matrix} 0 & 0 \\ 0 & \boxed{1} \end{matrix}$	`1 is changed to 0` `if N = W = NW = 0`	

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

Latency of Levialdi's algorithm

$T(n) = 2n^{1/2} - 1$ {shrinkage} $+ 2n^{1/2} - 1$ {expansion}

x	1	y
1	0	y
y	y	z

Component do not merge in shrinkage phase
Consider a 0 that is about to become a 1
If any y is 1, then already connected
If z is 1 then it will change to 0 unless
at least one neighboring y is 1

12 Mesh-Related Architectures

Study variants of simple mesh and torus architectures:

- Variants motivated by performance or cost factors
- Related architectures: pyramids and meshes of trees

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

12.1 Three or More Dimensions

3D vs 2D mesh: $D = 3p^{1/3} - 3$ vs $2p^{1/2} - 2$; $B = p^{2/3}$ vs $p^{1/2}$

Example:	3D $8 \times 8 \times 8$ mesh	$p = 512$, $D = 21$, $B = 64$
	2D 22×23 mesh	$p = 506$, $D = 43$, $B = 23$

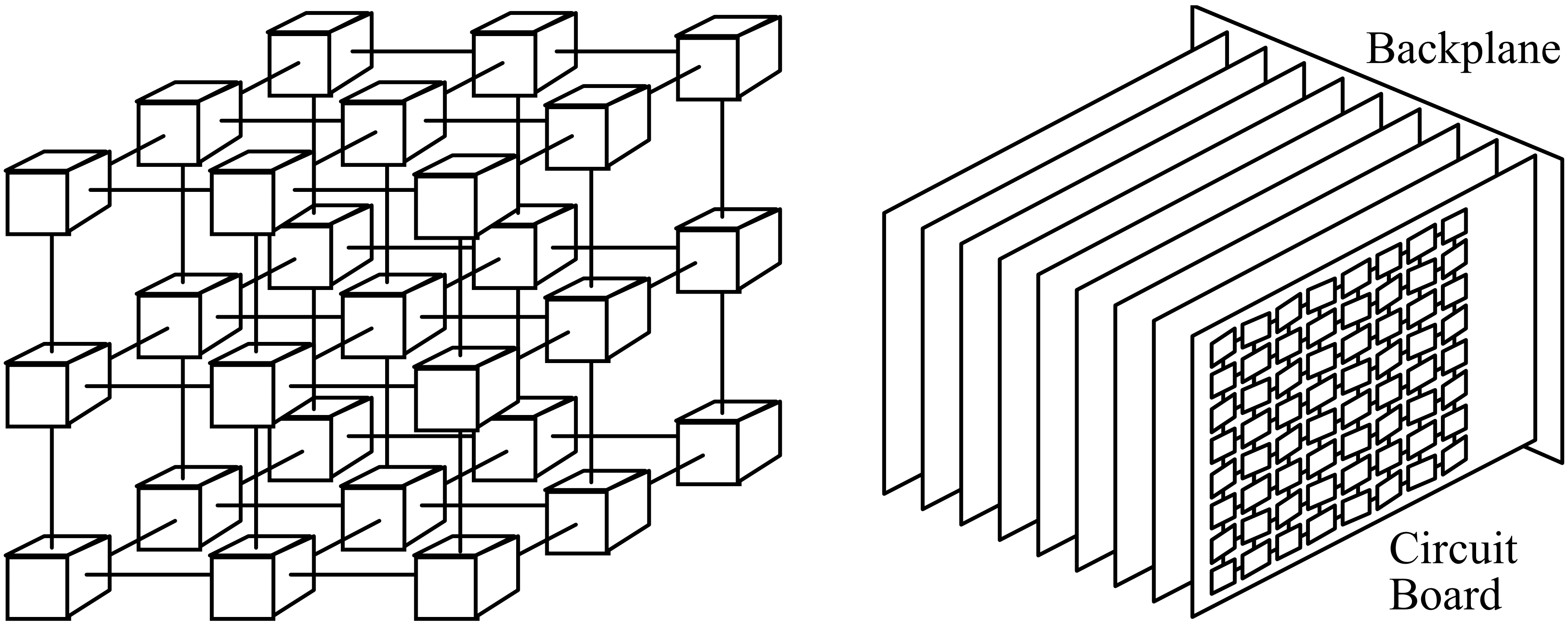
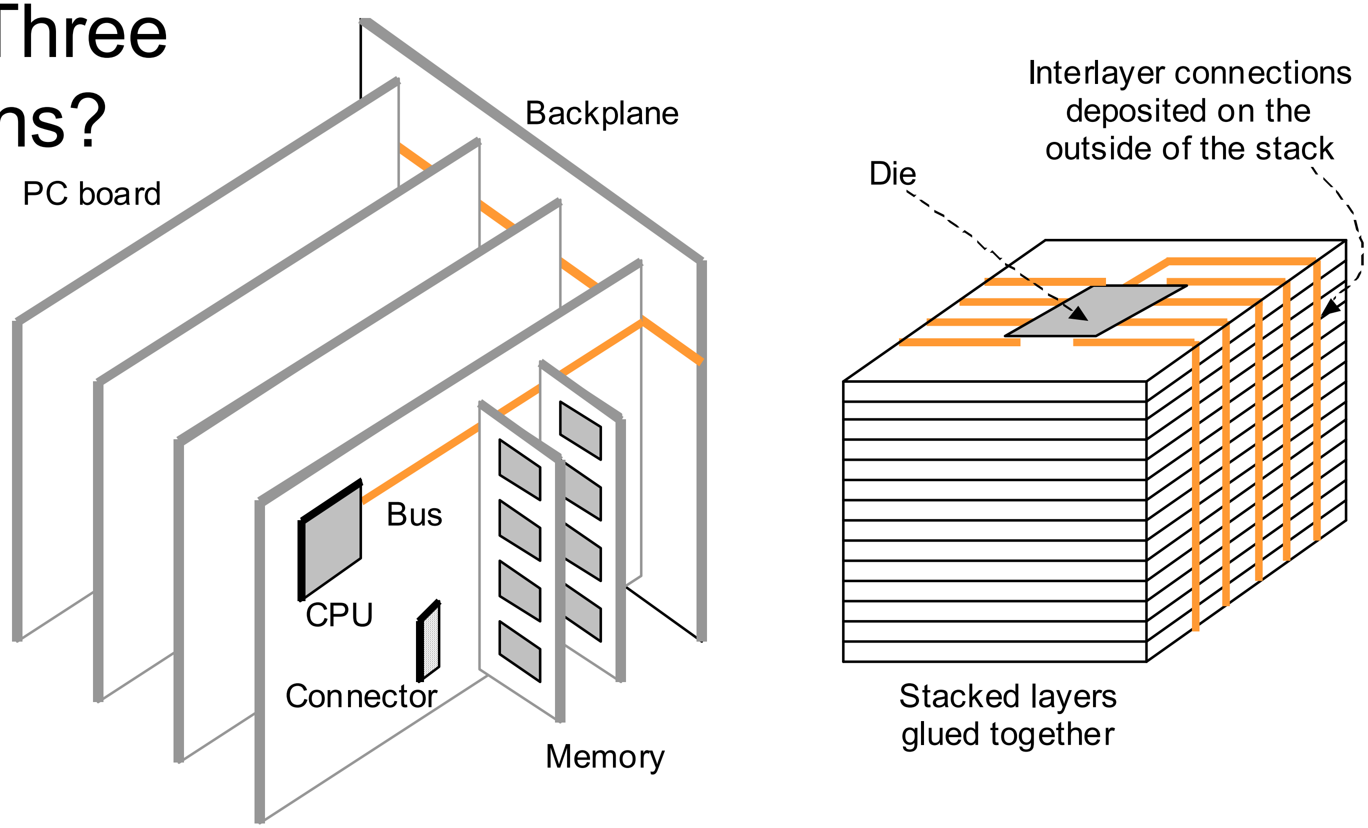


Fig. 12.1 3D and 2.5D physical realizations of a 3D mesh.

More than Three Dimensions?

2.5D and 3D packaging technologies

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



(a) 2D or 2.5D packaging now common

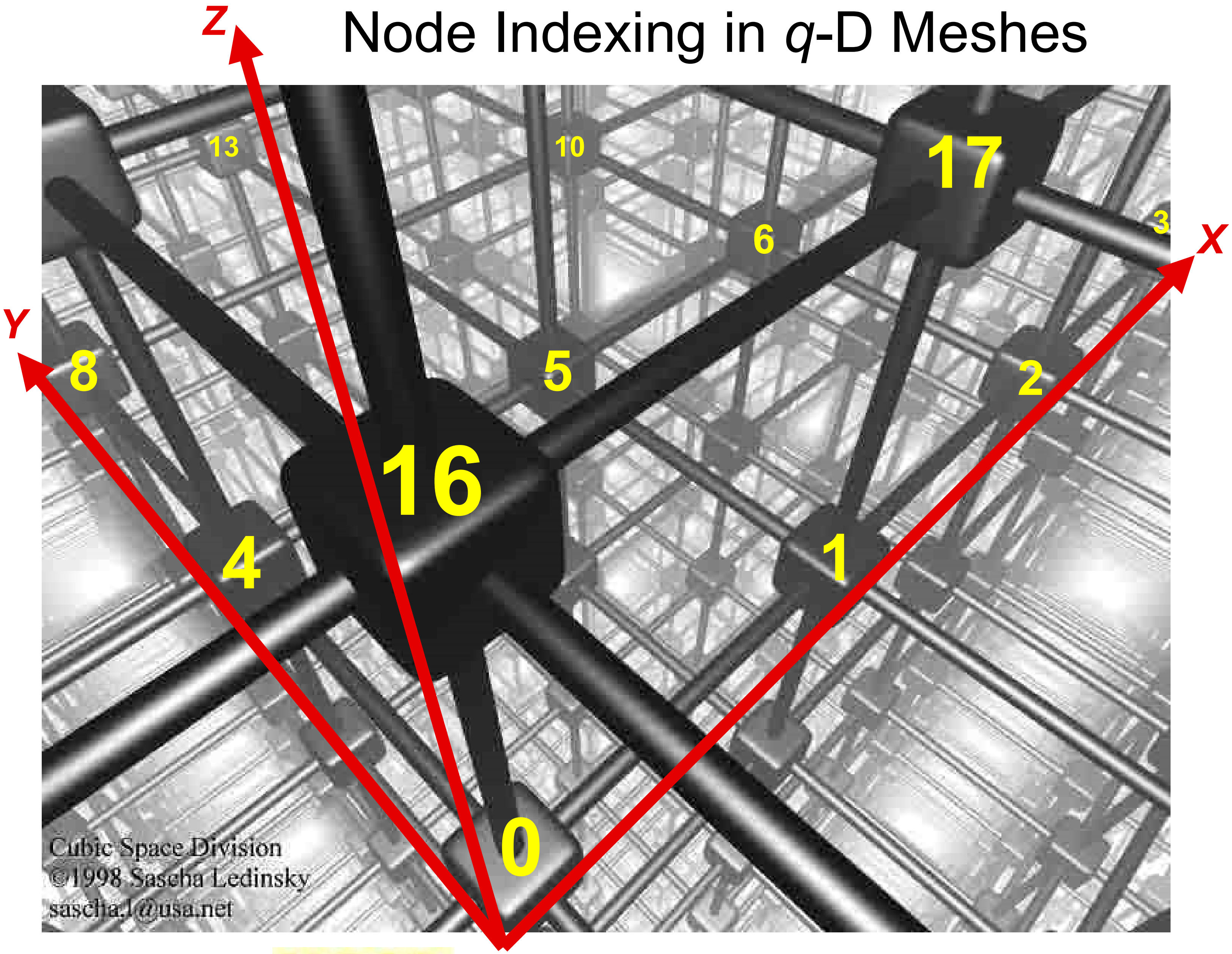
(b) 3D packaging of the future

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

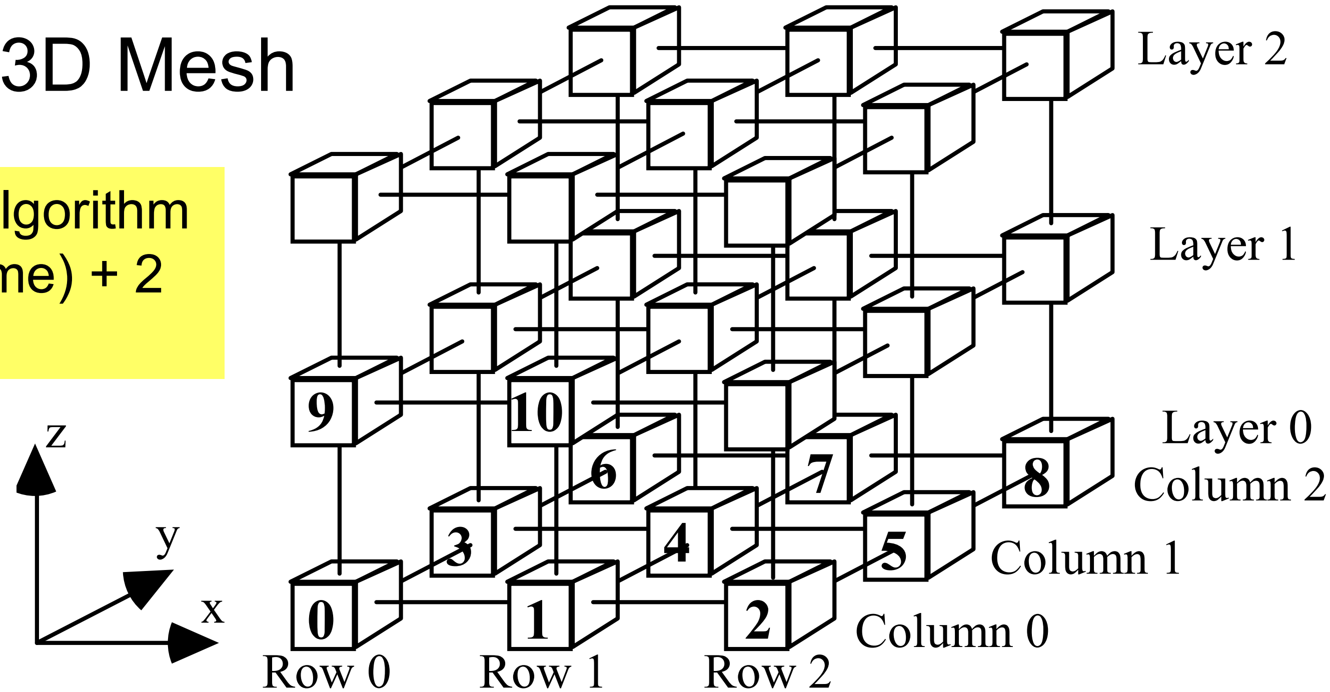
Node Indexing in q-D Meshes



zyx order	
000	0
001	1
002	2
003	3
010	4
011	5
012	6
013	7
020	8
. . .	
100	16
101	17
. . .	
200	32
201	33

Sorting on a 3D Mesh

Time for Kunde's algorithm
$= 4 \times$ (2D-sort time) + 2
$\cong 16p^{1/3}$ steps



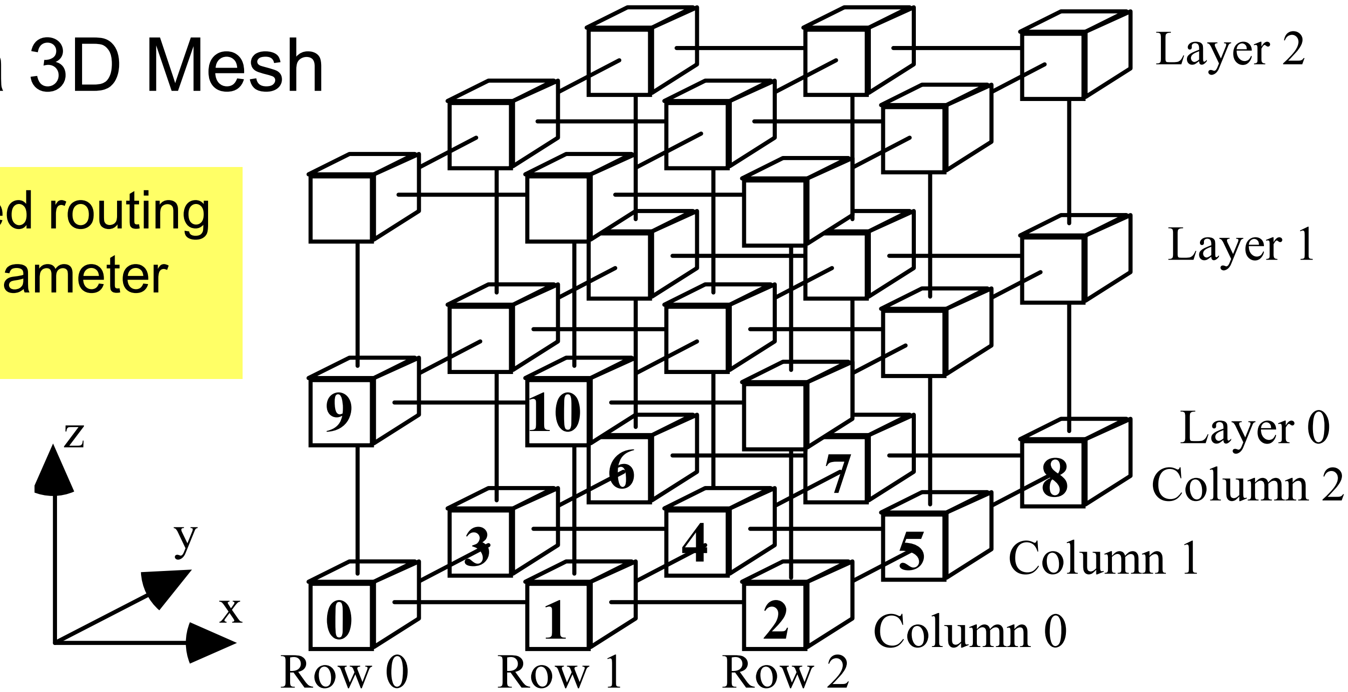
Defining the zyx processor ordering

A variant of shearsort is available, but Kunde's algorithm is faster and simpler

<u>Sorting on 3D mesh (zyx order; reverse of node index)</u>

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 sorted in reverse order)
Phase 4: Apply 2 steps of odd-even transposition along z
Phase 5: Sort elements on each xy layer into yx order

Routing on a 3D Mesh



Time for sort-based routing
= Sort time + Diameter
≅ $19p^{1/3}$ steps

As in 2D case, partial sorting can be used

Simple greedy algorithm does fine usually, but sorting first reduces buffer requirements

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

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

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

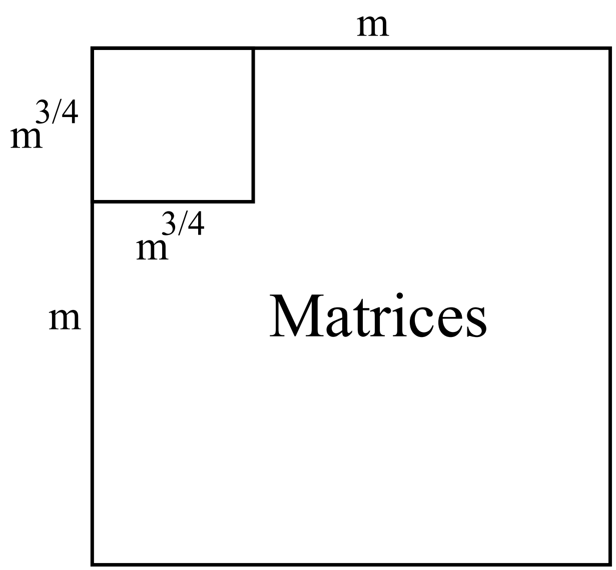
Phase 3: Route along *y* dimension to correct column

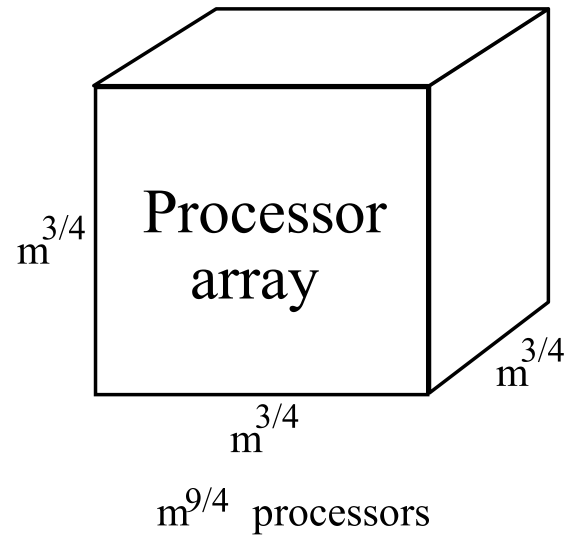
Phase 4: Route along *x* dimension to destination

Matrix Multiplication on a 3D Mesh

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

Matrix blocking for multiplication on a 3D mesh





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

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 can take time that is of lower order

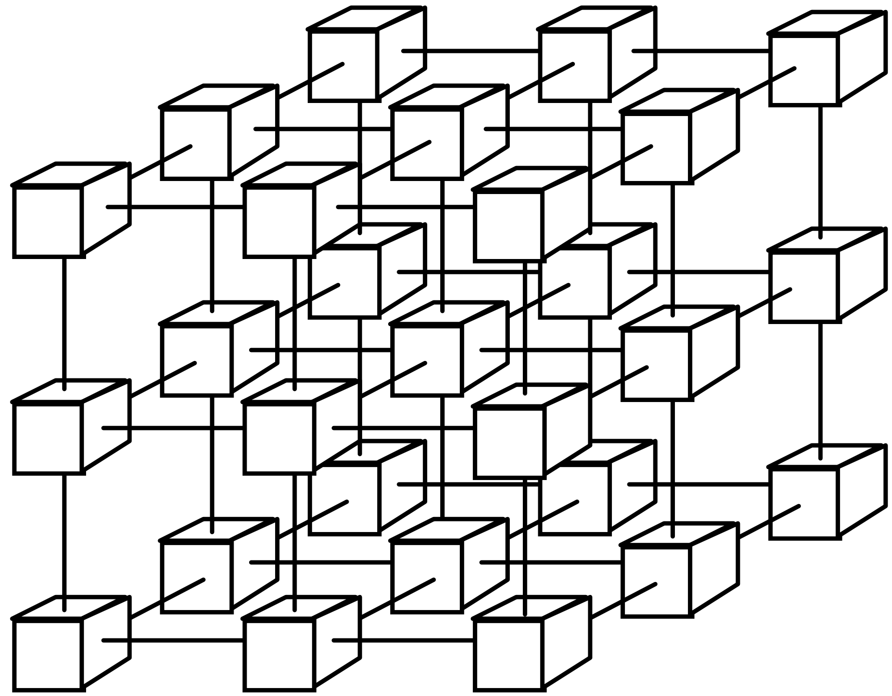
Optimal: Matches sequential work and diameter-based lower bound

Low- vs High-Dimensional Meshes

There is a good match between the structure of a 3D mesh and communication requirements of physical modeling problems

6×6 mesh emulating $3 \times 3 \times 3$ mesh (not optimal)

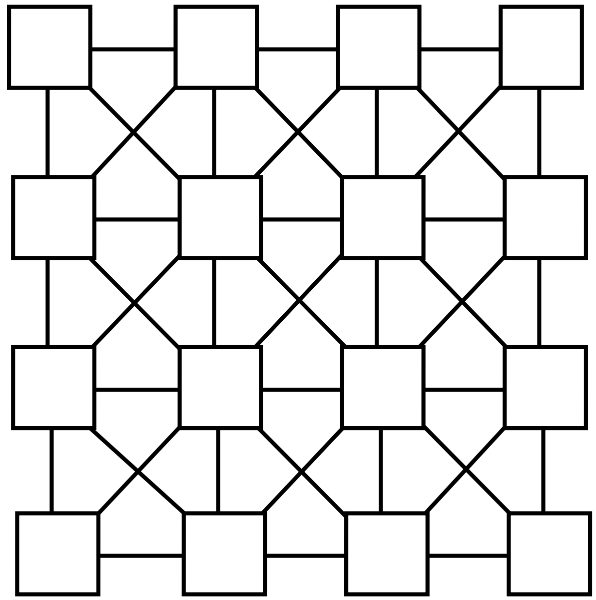
Middle layer	Upper layer
Lower layer	

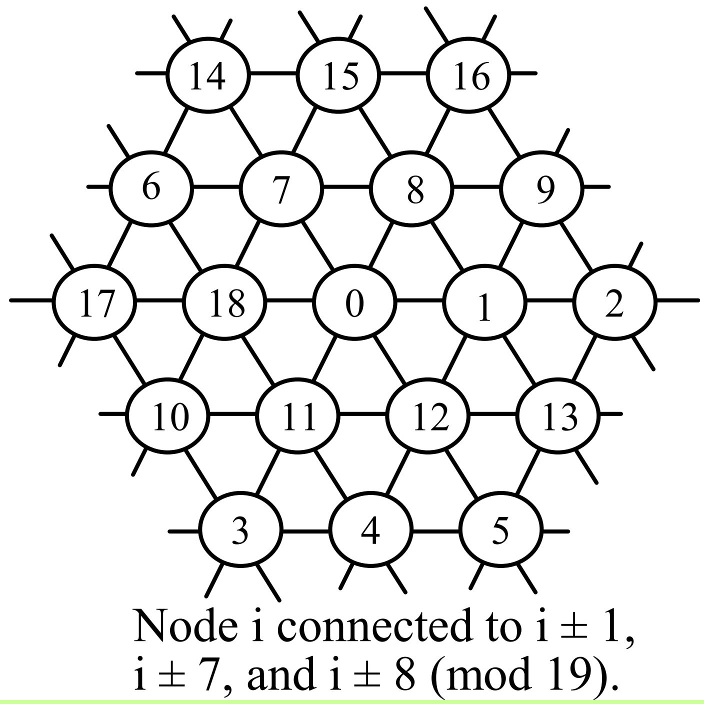


A low-dimensional mesh can efficiently emulate a high-dimensional one

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 architecture, given that for the 4-port processors, fewer ports and ease of layout allow us to make each channel wider?

12.2 Stronger and Weaker Connectivities





Fortified meshes and other models with stronger connectivities:

Eight-neighbor
Six-neighbor
Triangular
Hexagonal

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

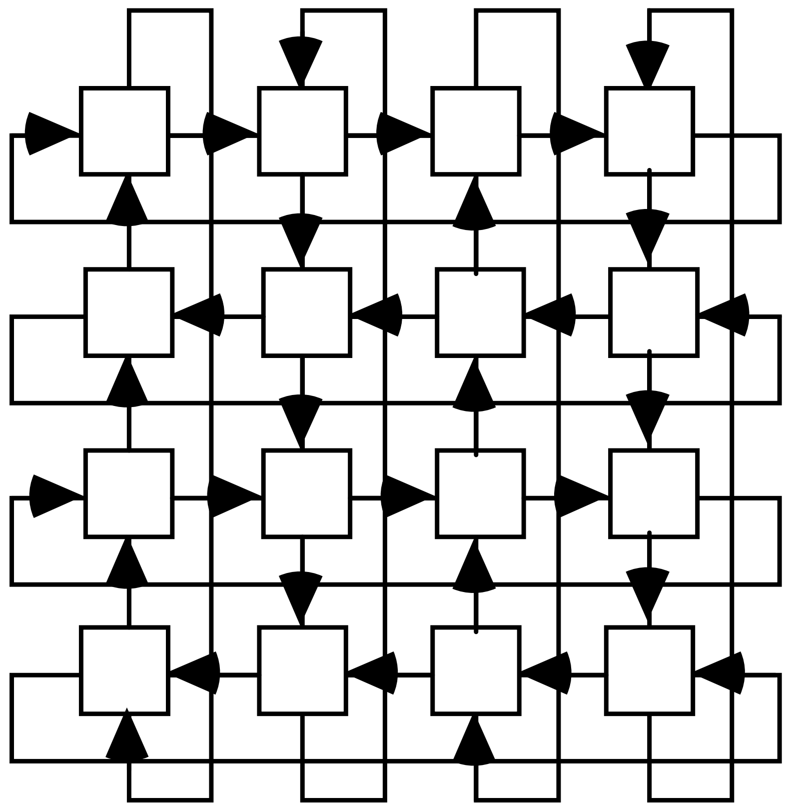
As in higher-dimensional meshes, greater connectivity does not automatically translate into greater performance

Area and signal-propagation delay penalties must be factored in

Simplification via Link Orientation

Two in- and out-channels per node, instead of four

With even side lengths, the diameter does not change



Some shortest paths become longer, however

Can be more cost-effective than 2D mesh

Figure 12.3 A 4 × 4 Manhattan street network.

Simplification via Link Removal

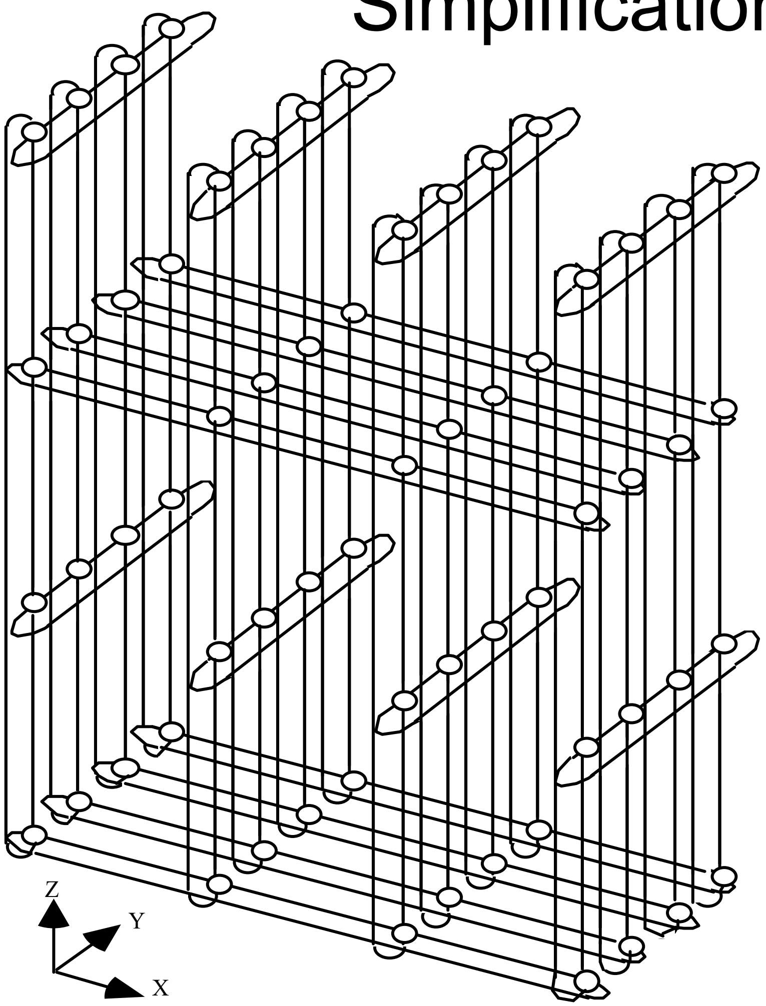
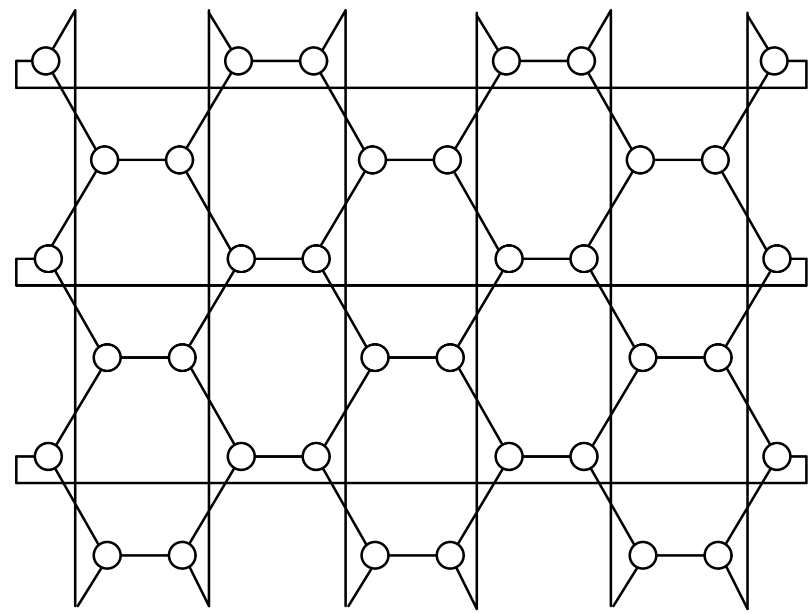


Figure 12.4 A pruned 4 × 4 × 4 torus with nodes of degree four [Kwai97].



Honeycomb torus

Pruning a high-dimensional mesh or torus can yield an architecture with the same diameter but much lower implementation cost

Simplification via Link Sharing

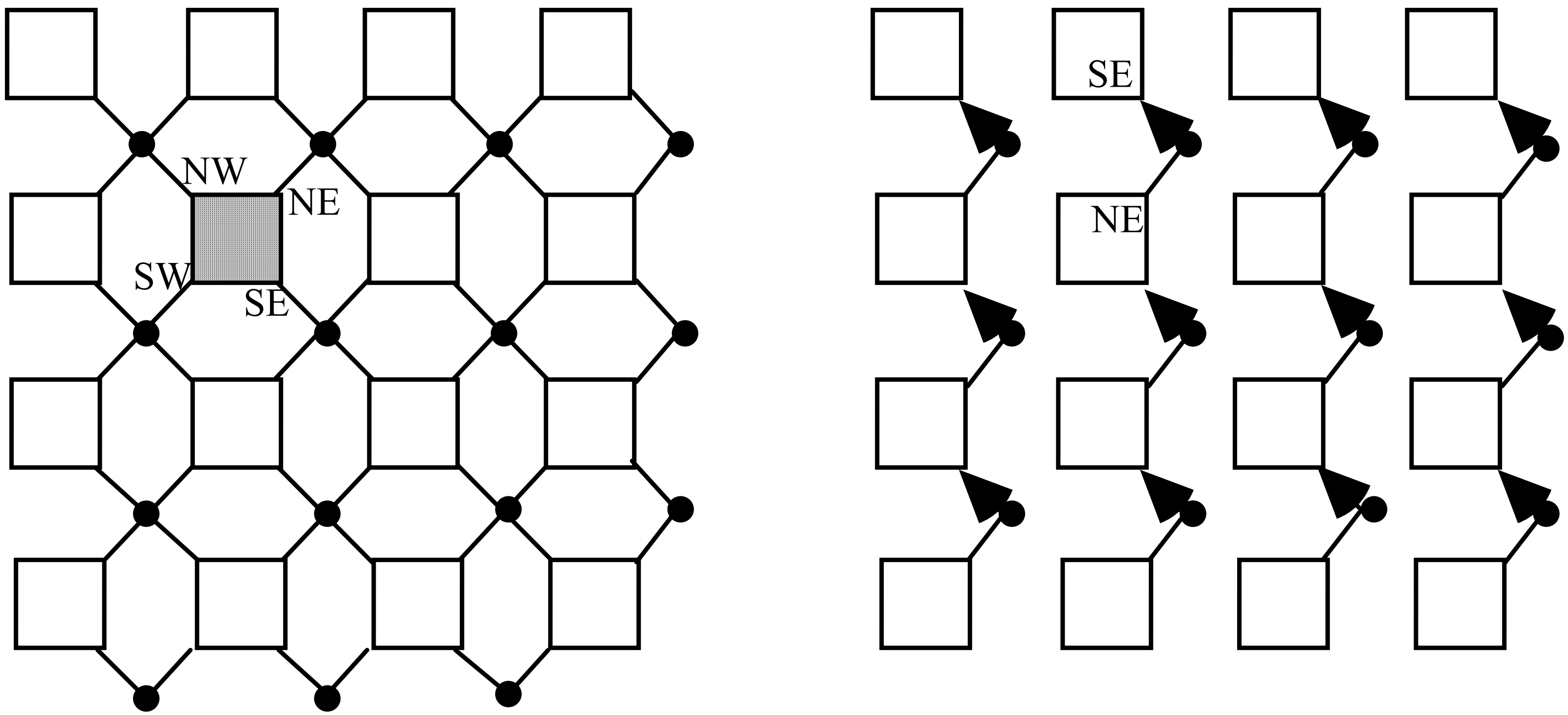


Fig. 12.5   Eight-neighbor mesh with shared links and example data paths.

Factor-of-2 reduction in ports and links, with no performance degradation for uniaxis communication (weak SIMD model)

12.3 Meshes Augmented with Nonlocal Links

Motivation: Reduce the wide diameter (which is a weakness of meshes)

Increases max node degree and hurts the wiring locality and regularity

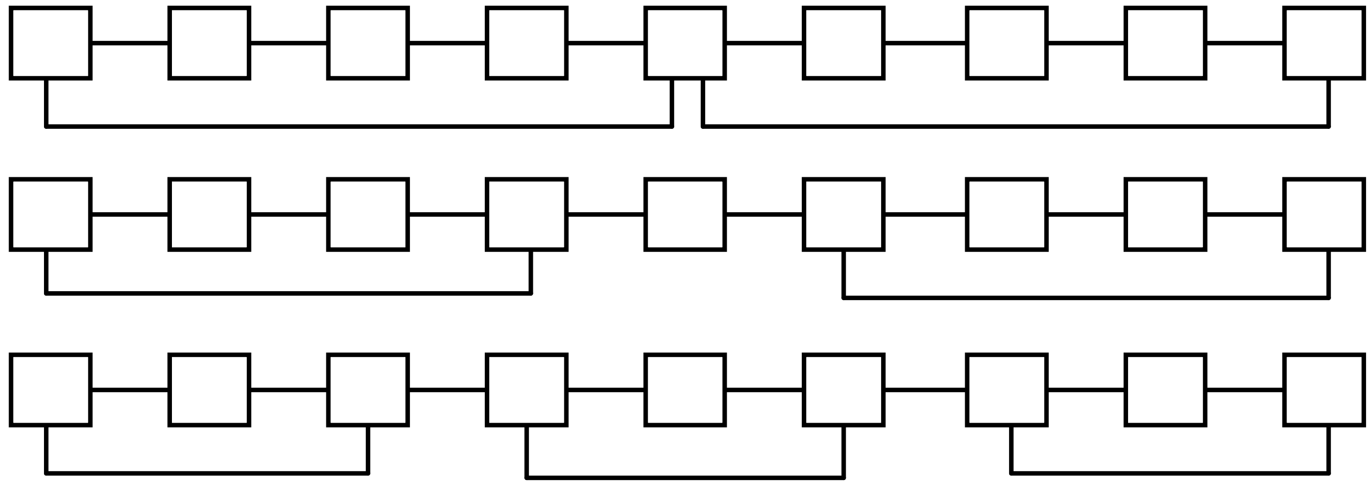
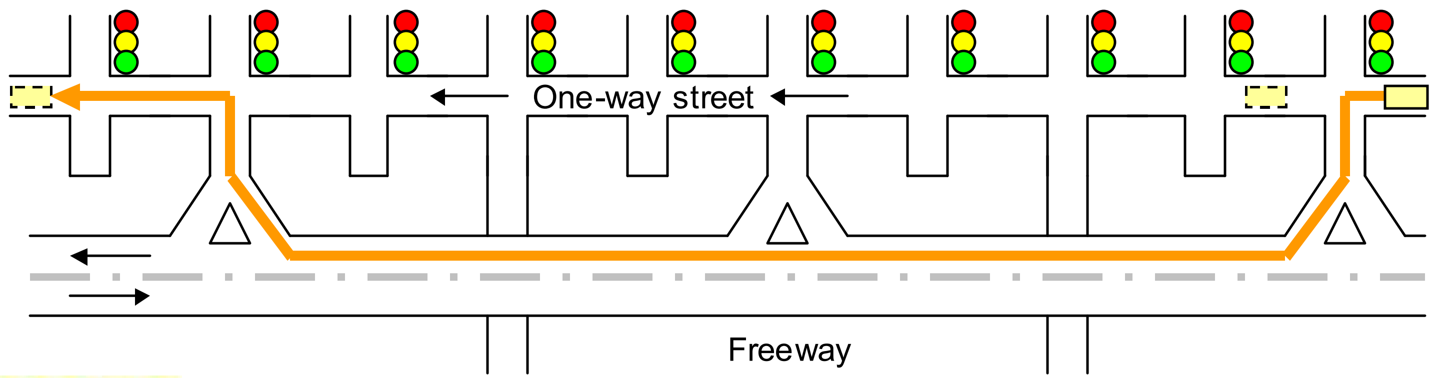
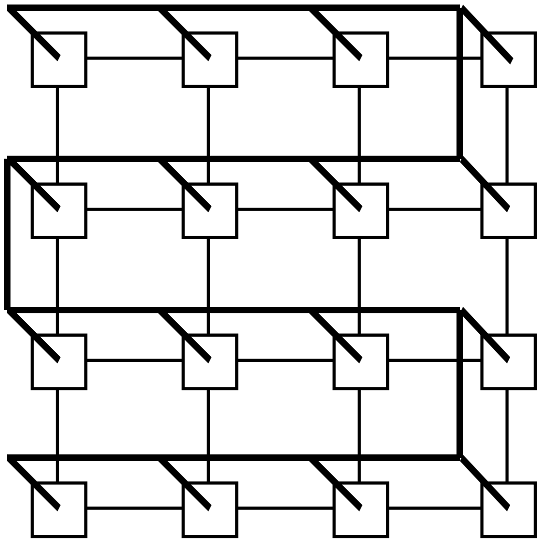


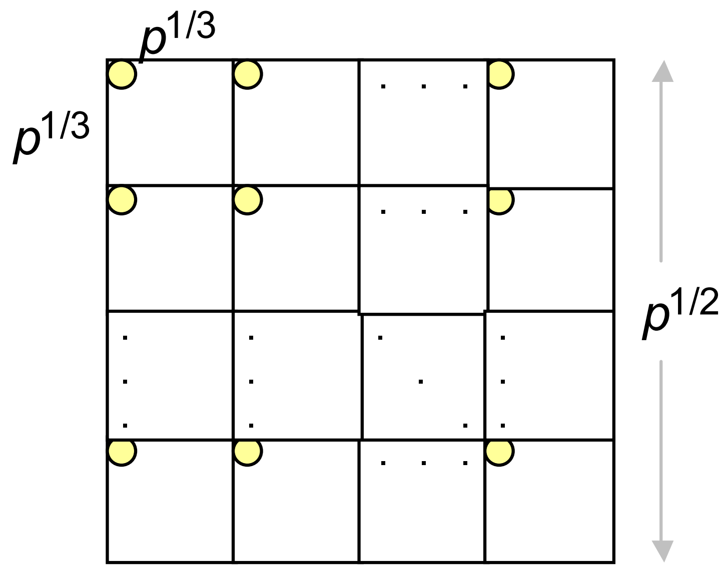
Fig. 12.6 Three examples of bypass links along the rows of a 2D mesh.

Road analogy for bypass connections



Using a Single Global Bus





The single bus increases the bisection width by 1, so it does not help much with sorting or other tasks that need extensive data movement

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

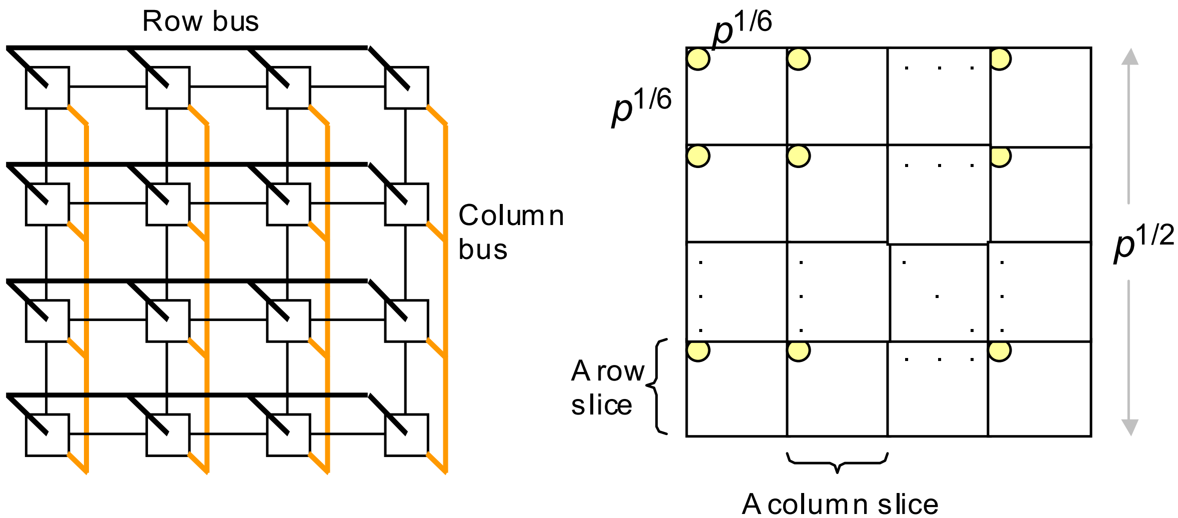
<u>Semigroup computation on 2D mesh with a global bus</u>

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

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

Mesh with Row and Column Buses



The bisection width doubles, so row and column buses do not fundamentally change the performance of sorting or other tasks that need extensive data movement

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

<u>Semigroup computation on 2D mesh with row and column buses</u>

Phase 1: Find partial results in $p^{1/6} \times p^{1/6}$ submeshes in O($p^{1/6}$) steps
Phase 2: Distribute $p^{1/3}$ row values left among the $p^{1/6}$ rows in same slice
Phase 3: Combine row values in $p^{1/6}$ steps using the row buses
Phase 4: Distribute column-0 values to $p^{1/3}$ columns using the row buses
Phase 5: Combine column values in $p^{1/6}$ steps using the column buses
Phase 6: Distribute $p^{1/3}$ values on row 0 among $p^{1/6}$ rows of row slice 0
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

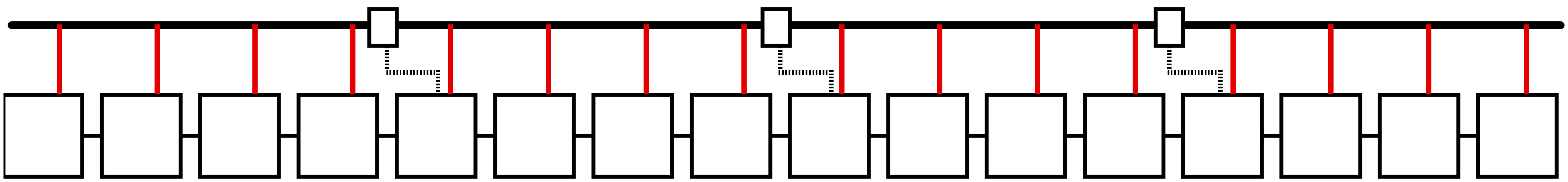
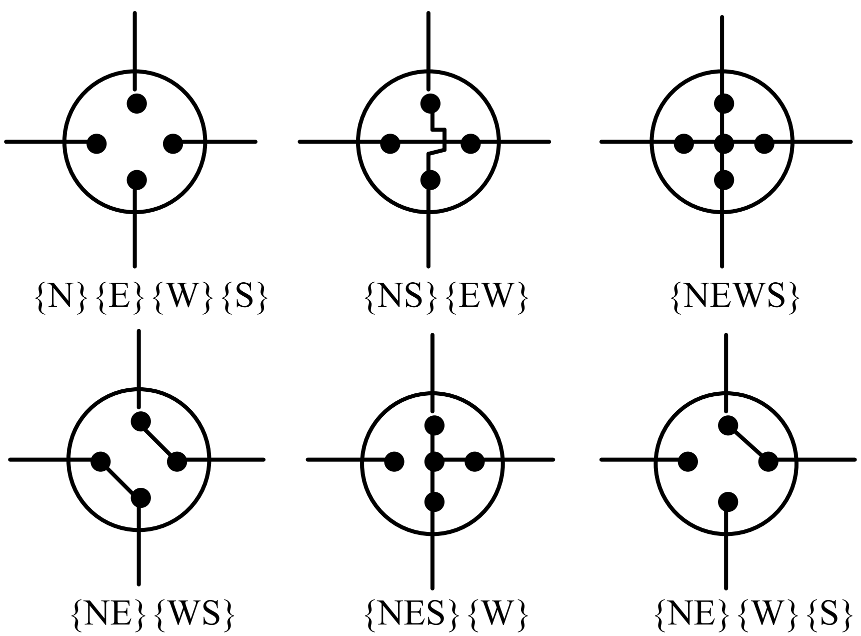


Fig. 12.9 Linear array with a separable bus using reconfiguration switches.

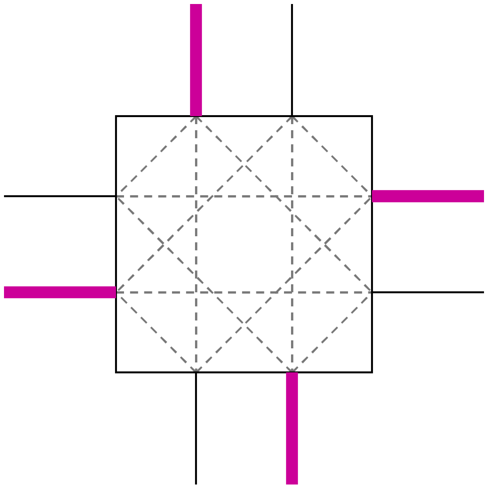
Semigroup computation in O(log *p*) steps; both 1D and 2D meshes



Various subsets of processors (not just rows and columns) can be configured, to communicate over shared buses

Fig. 12.10 Some processor states in a reconfigurable mesh.

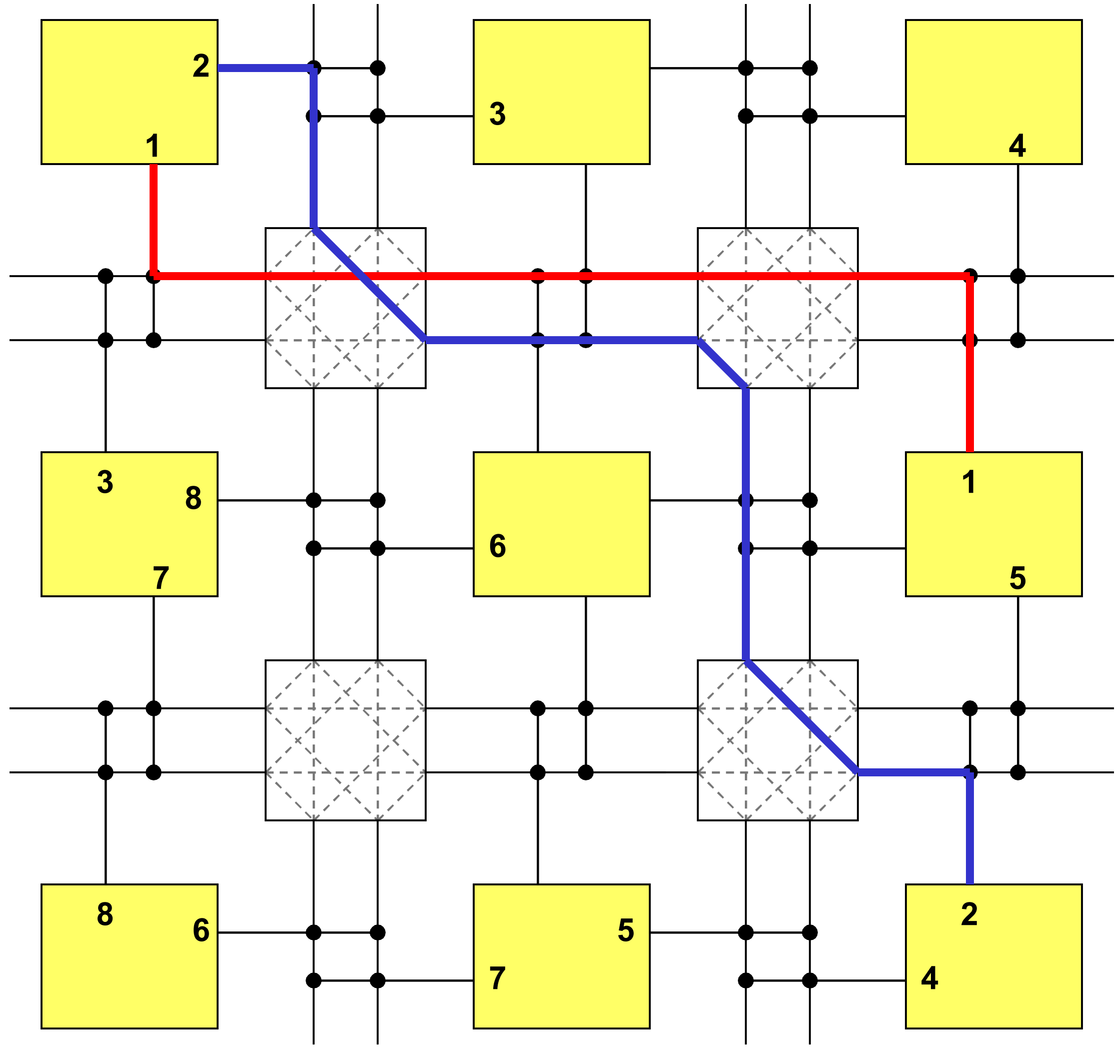
Programmable Connectivity in FPGAs



Interconnection switch with 8 ports and four connection choices for each port:

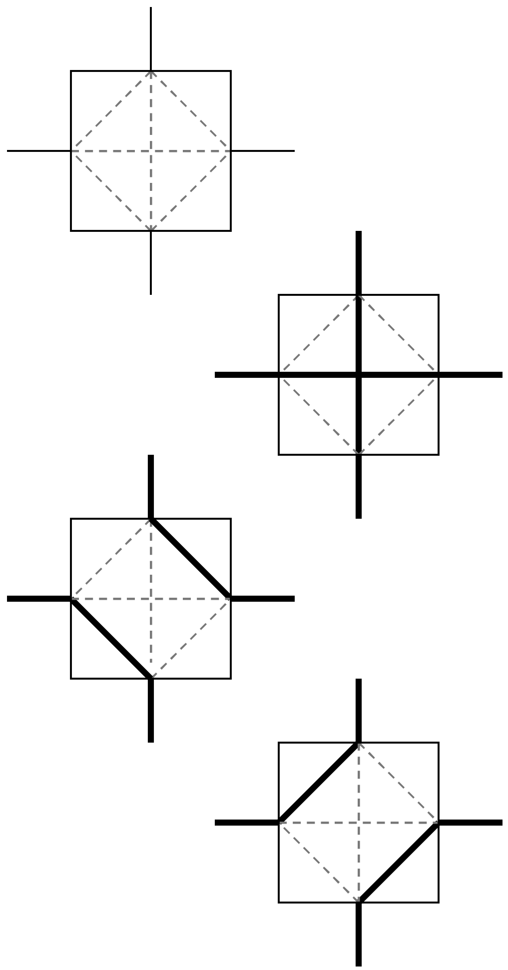
0 – No connection
1 – Straight through
2 – Right turn
3 – Left turn

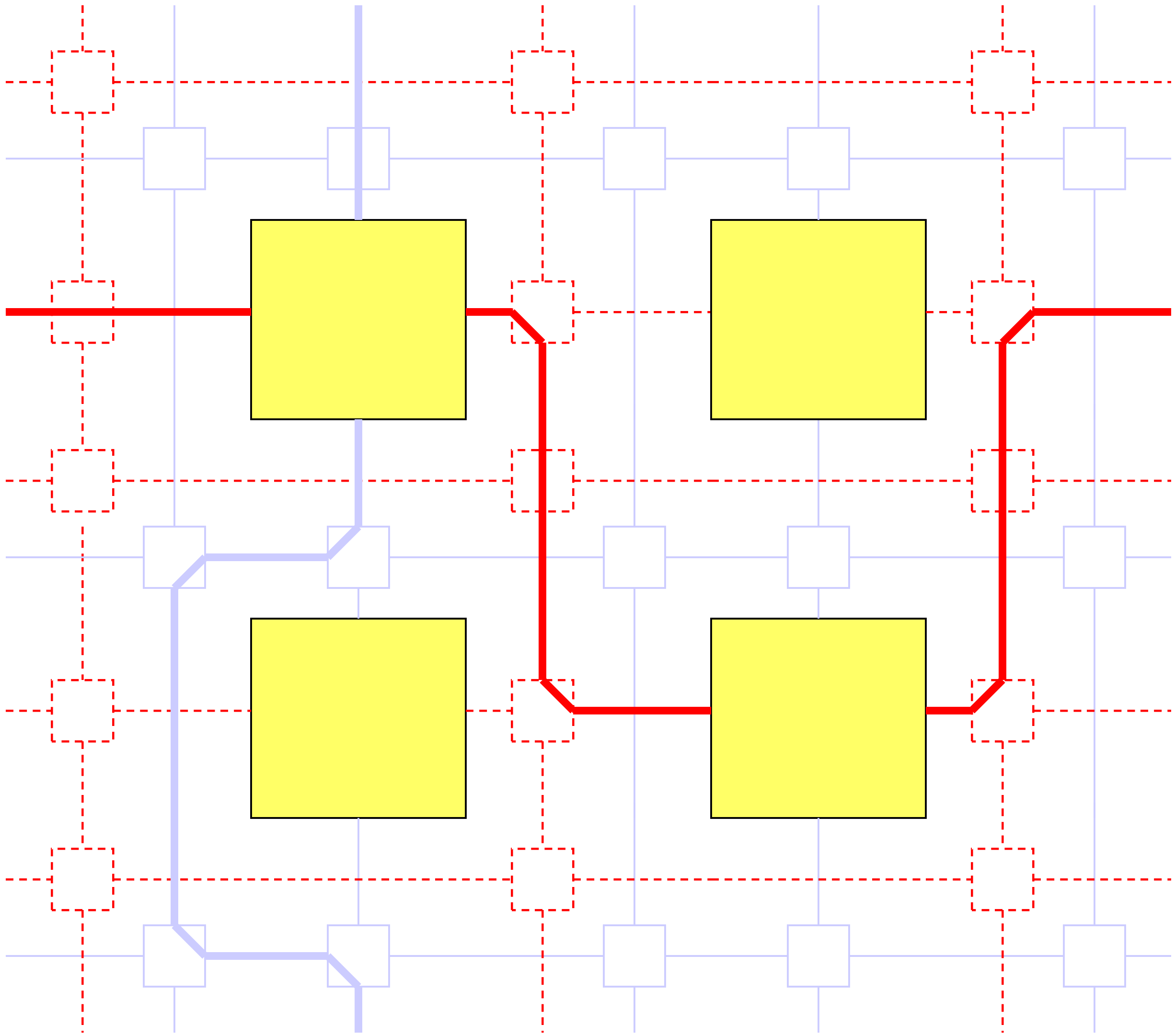
8 control bits (why?)



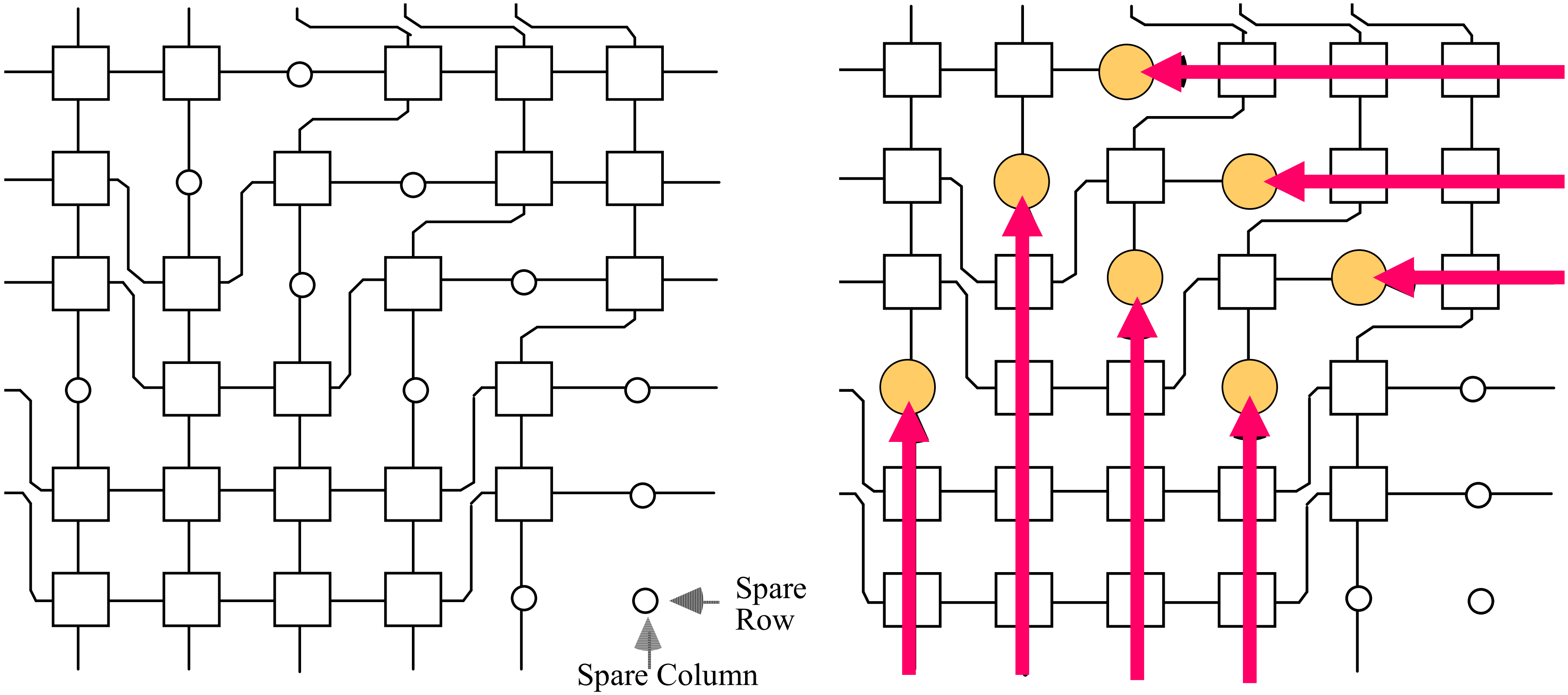
An Array Reconfiguration Scheme

3-state 2 × 2 switch





Reconfiguration of Faulty Arrays

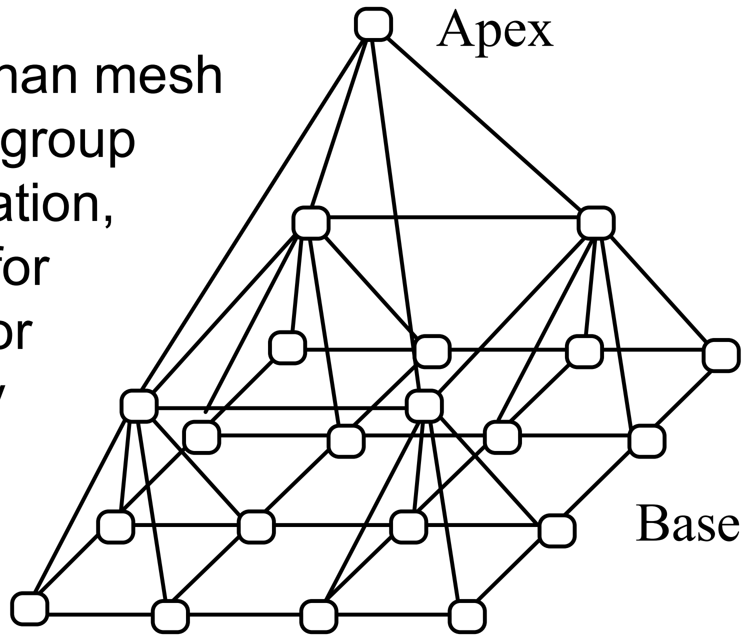


Question: How do we know which cells/nodes must be bypassed?

Must devise a scheme in which healthy nodes set the switches

12.5 Pyramid and Multigrid Systems

Faster than mesh for semigroup computation, but not for sorting or arbitrary routing



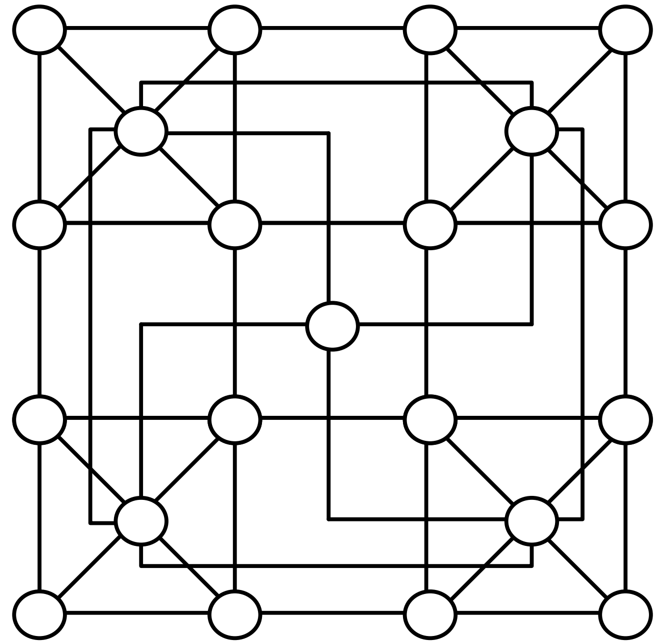


Fig. 12.11 Pyramid with 3 levels and 4 $\times$ 4 base along with its 2D layout.

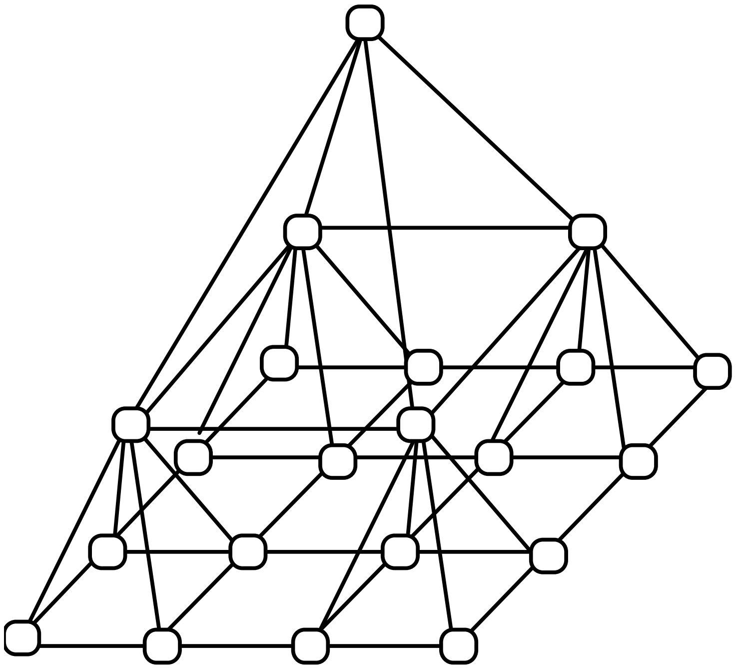
Originally developed for image processing applications

Roughly 3/4 of the processors belong to the base

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

Pyramid and 2D Multigrid Architectures

Pyramid is to 2D multigrid what X-tree is to 1D multigrid



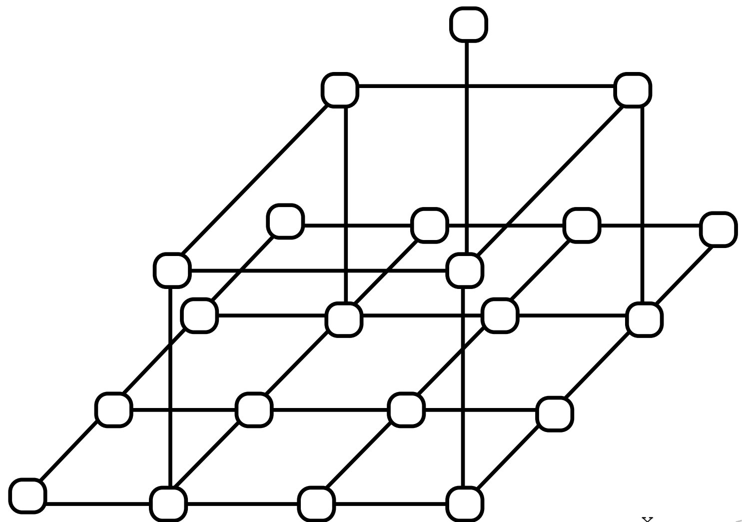
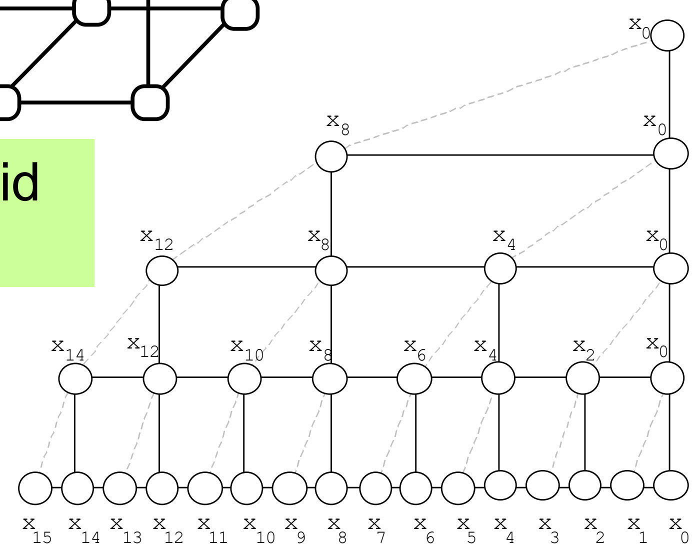


Fig. 12.12 The relationship between pyramid and 2D multigrid architectures.

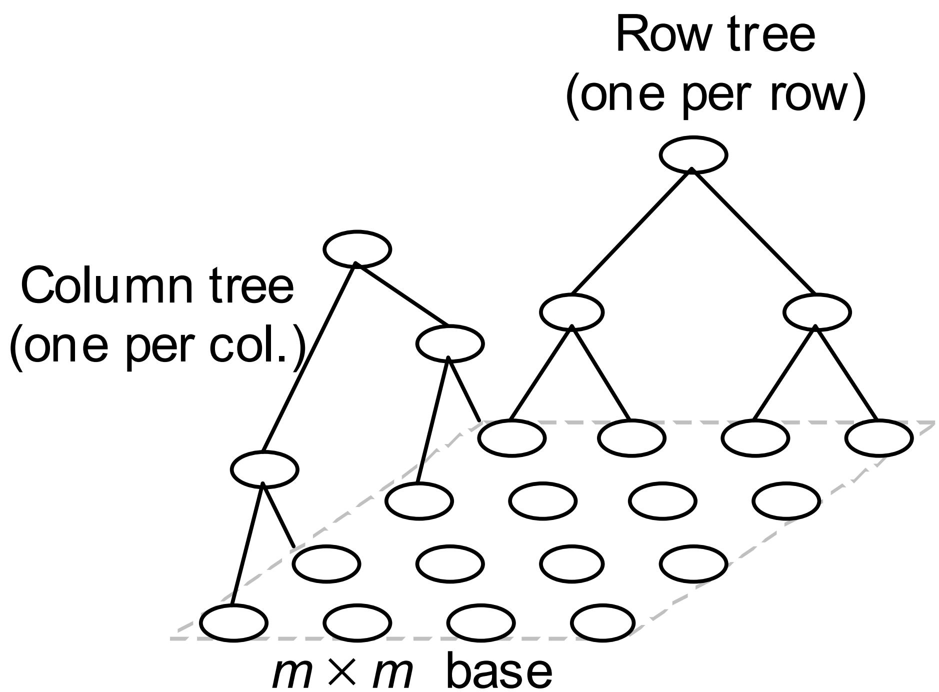


Multigrid architecture is less costly and can emulate the pyramid architecture quite efficiently

12.6 Meshes of Trees

2*m* trees, each with *m* leaves, sharing leaves in the base

Row and column roots can be combined into *m* degree-4 nodes



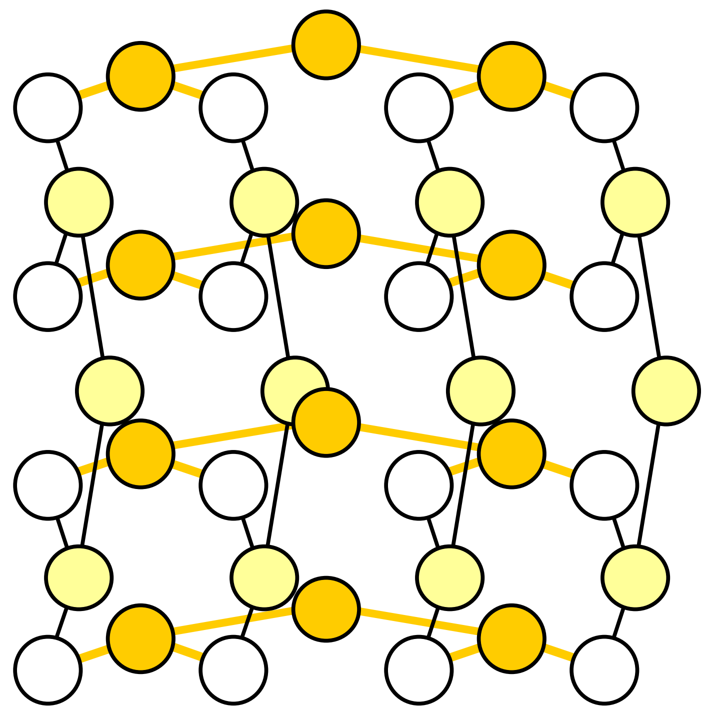
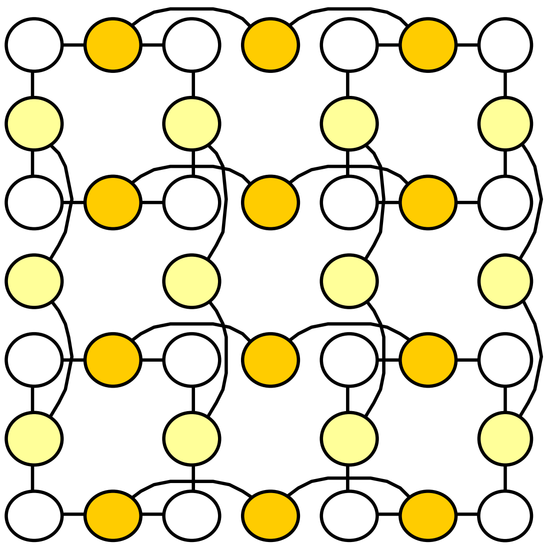


Fig. 12.13 Mesh of trees architecture with 3 levels and a 4×4 base.

Alternate Views of a Mesh of Trees



2D layout for mesh of trees network with a 4 × 4 base; root nodes are in the middle row and column

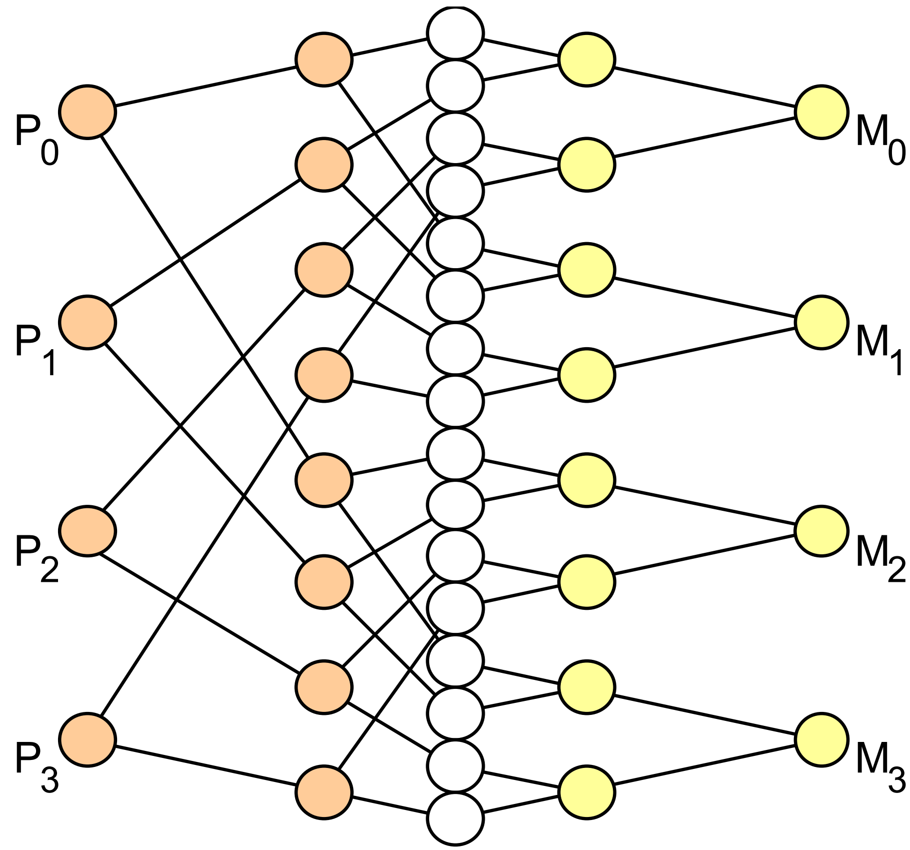


Fig. 12.14 Alternate views of the mesh of trees architecture with a 4 × 4 base.

Simple Algorithms for Mesh of Trees

Semigroup computation: row/column combining

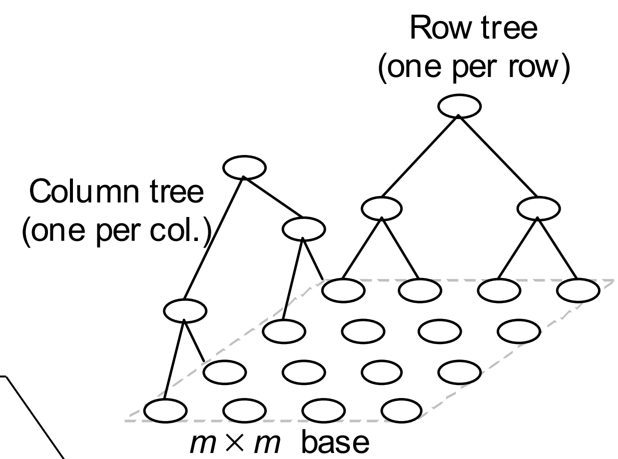
Parallel prefix computation: similar

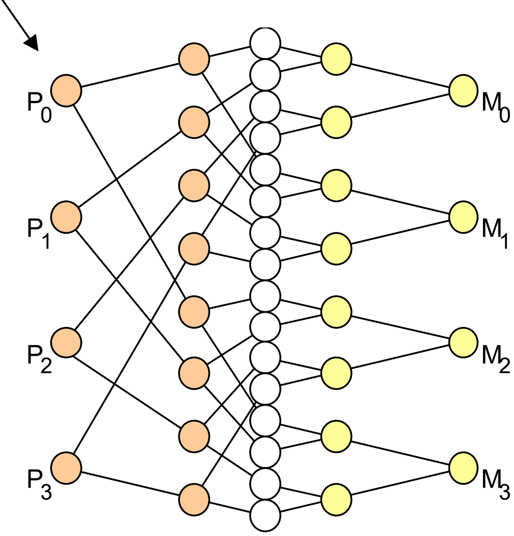
Routing m^2 packets, one per processor on the $m \times m$ base: requires $\Omega(m) = \Omega(p^{1/2})$ steps

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 destination nodes are distinct

Sorting m^2 keys, one per processor on the $m \times m$ base: emulate any mesh sorting algorithm

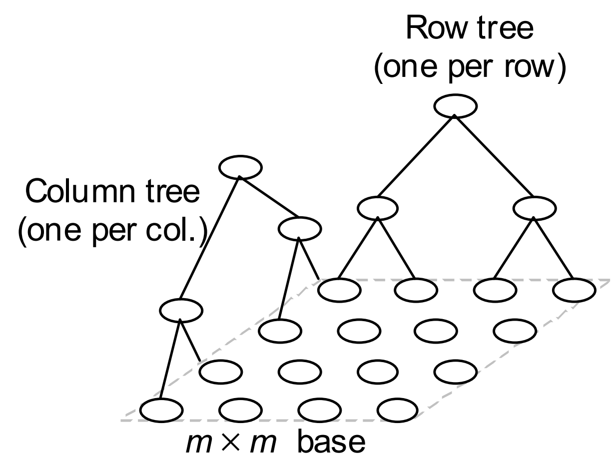
Sorting m keys stored in merged roots: broadcast x_i to row i and column i, compare x_i to x_j in leaf (i, j) to set a flag, add flags in column trees to find the rank of x_i, route x_i to node $rank[x_i]$





Some Numerical Algorithms for Mesh of Trees

Matrix-vector multiplication $Ax = y$ (A stored on the base and vector x in the column roots, say; result vector y is obtained in the row roots): broadcast x_j in the jth column tree, compute $a_{ij}x_j$ in base processor (i, j), sum over row trees



Convolution of two vectors: similar

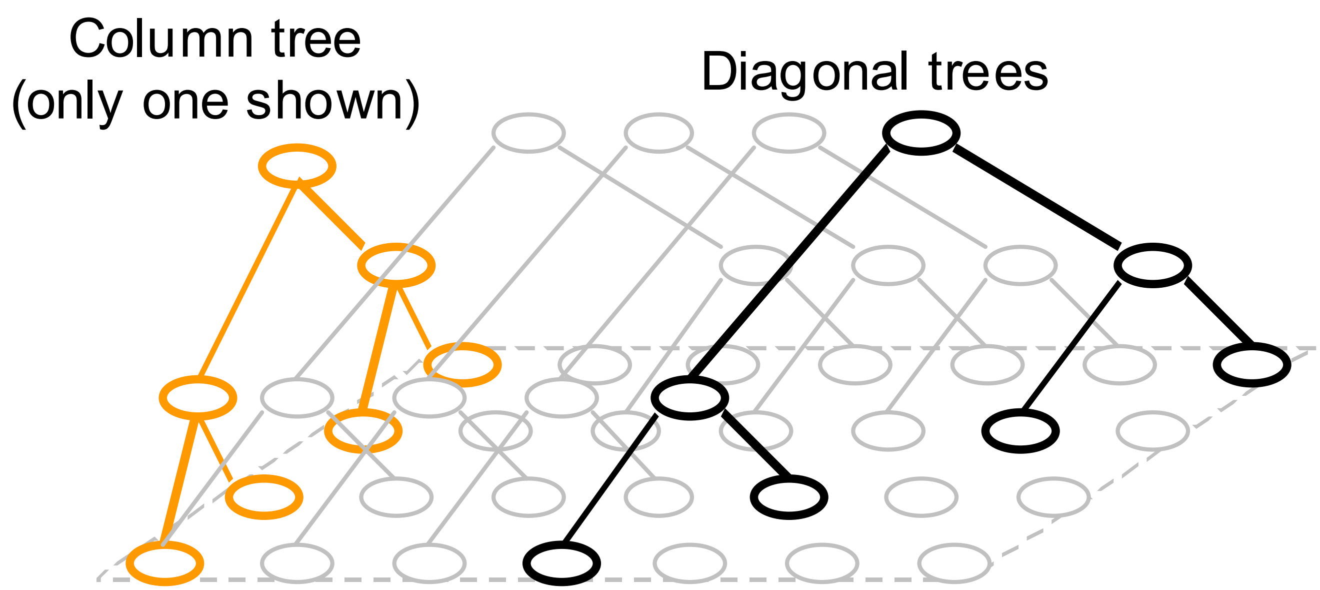
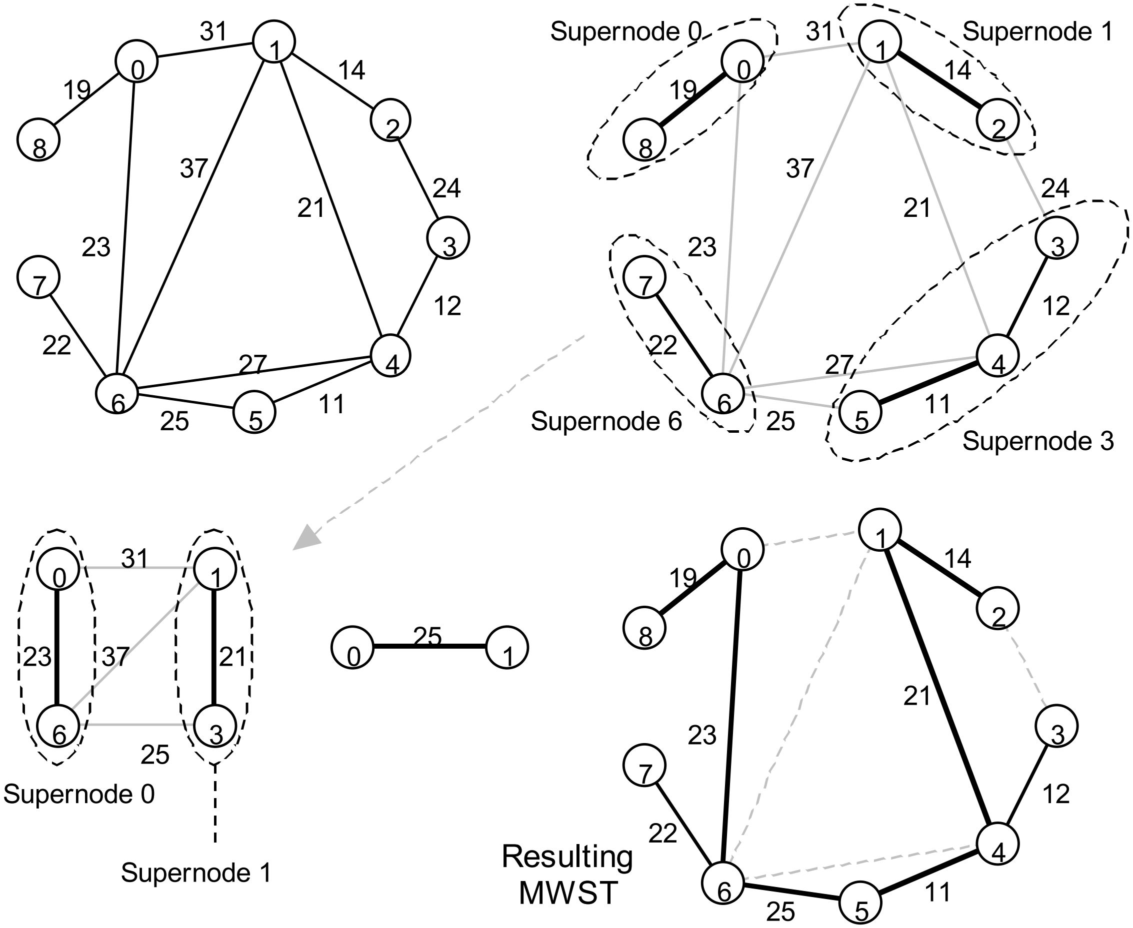


Fig. 12.15 Mesh of trees variant with row, column, and diagonal trees.

Minimal-Weight Spanning Tree Algorithm

Greedy algorithm: in each of at most $\log_2 n$ phases, add the minimal-weight edge that connects a component to a neighbor



Sequential algorithms, for an n-node, e-edge graph:

Kruskal's: $O(e \log e)$

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

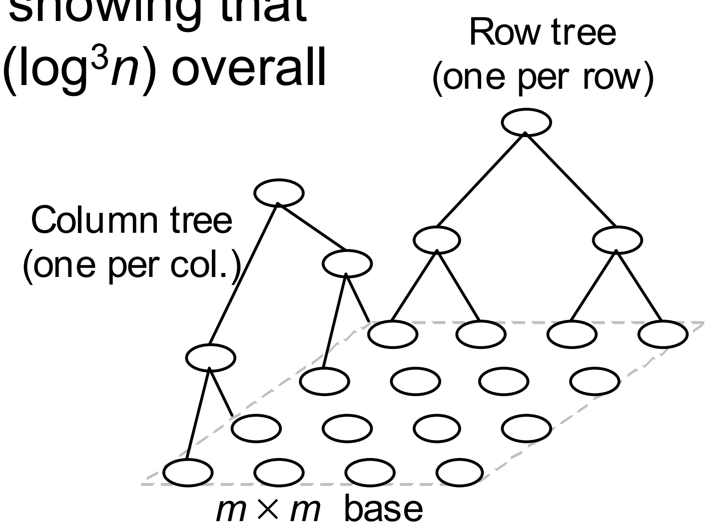
Both of these algorithms are $O(n^2 \log n)$ for dense graphs, with $e = O(n^2)$

Prim's (Fibonacci heap): $O(e + n \log n)$, or $O(n^2)$ for dense graphs

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

MWST Algorithm on a Mesh of Trees

The key to parallel version of the algorithm is showing that each phase can be done in O($\log^2 n$) steps; O($\log^3 n$) overall



Leaf (i, j) holds the weight $W(i, j)$ of edge (i, j) and "knows" whether the edge is in the spanning tree, and if so, in which supernode. In each phase, we must:

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

b. Merge supernodes for next phase

Subphase a takes O($\log n$) steps

Subphase b takes O($\log^2 n$) steps

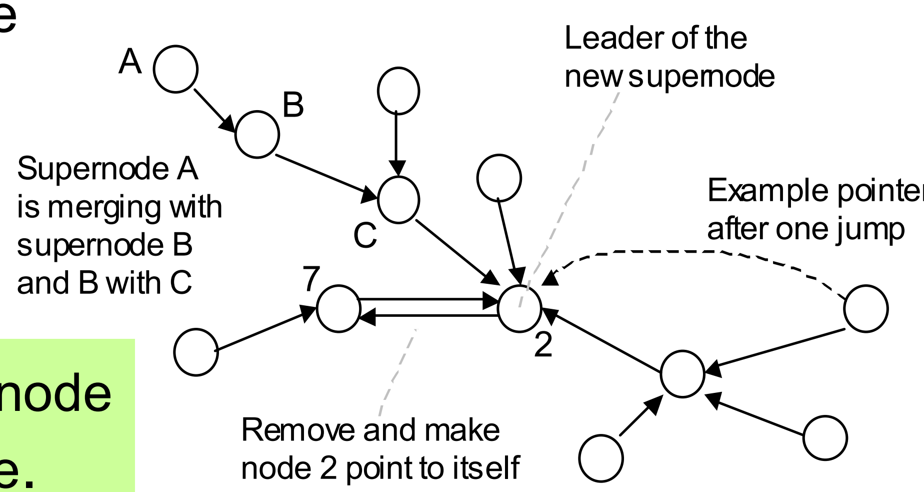


Fig. 12.17 Finding the new supernode ID when several supernodes merge.