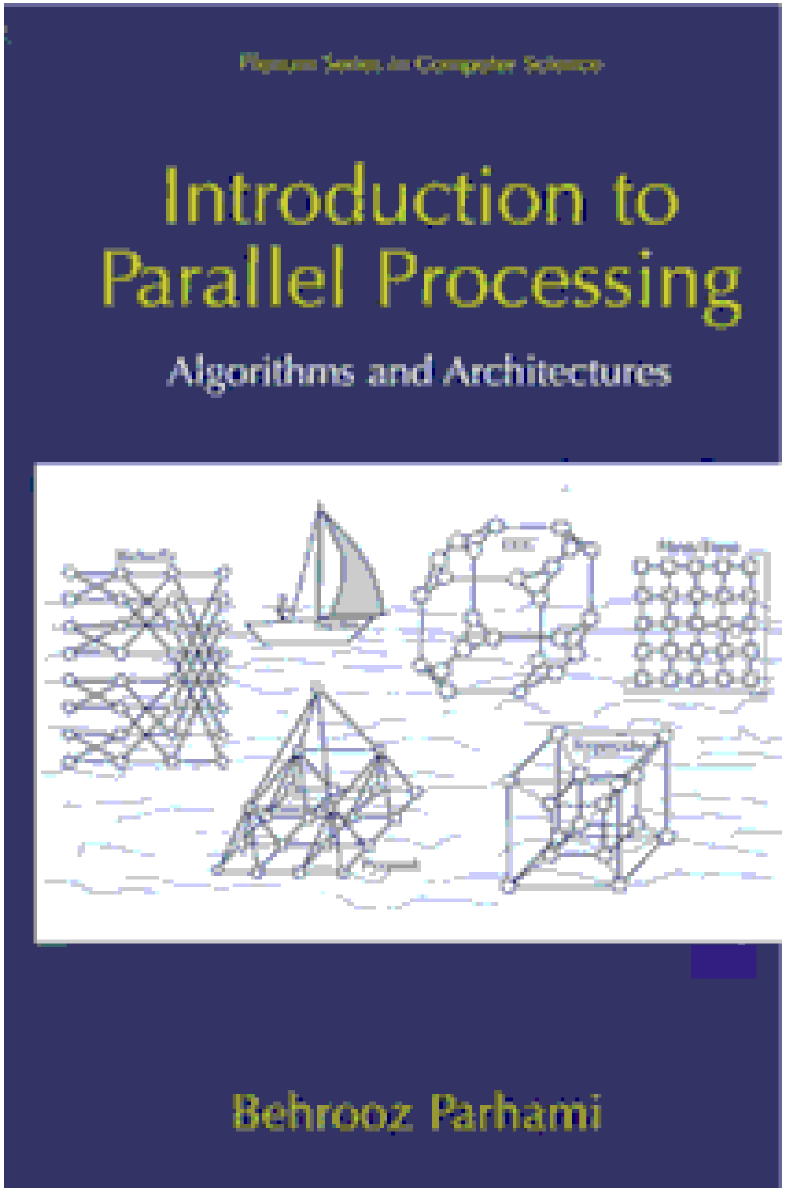



# Part II
## Extreme Models

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

Architectural Variations

# 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** |
|---|---|---|---|
| **First** | Spring 2005 | Spring 2006 | Fall 2008 |

# II Extreme Models

Study the two extremes of parallel computation models:

- Abstract SM (PRAM); ignores implementation issues
- Concrete circuit model; incorporates hardware details
- Everything else falls between these two extremes

# 5 PRAM and Basic Algorithms

PRAM, a natural extension of RAM (random-access machine):
- Present definitions of model and its various submodels
- Develop algorithms for key building-block computations

# 5.1 PRAM Submodels and Assumptions

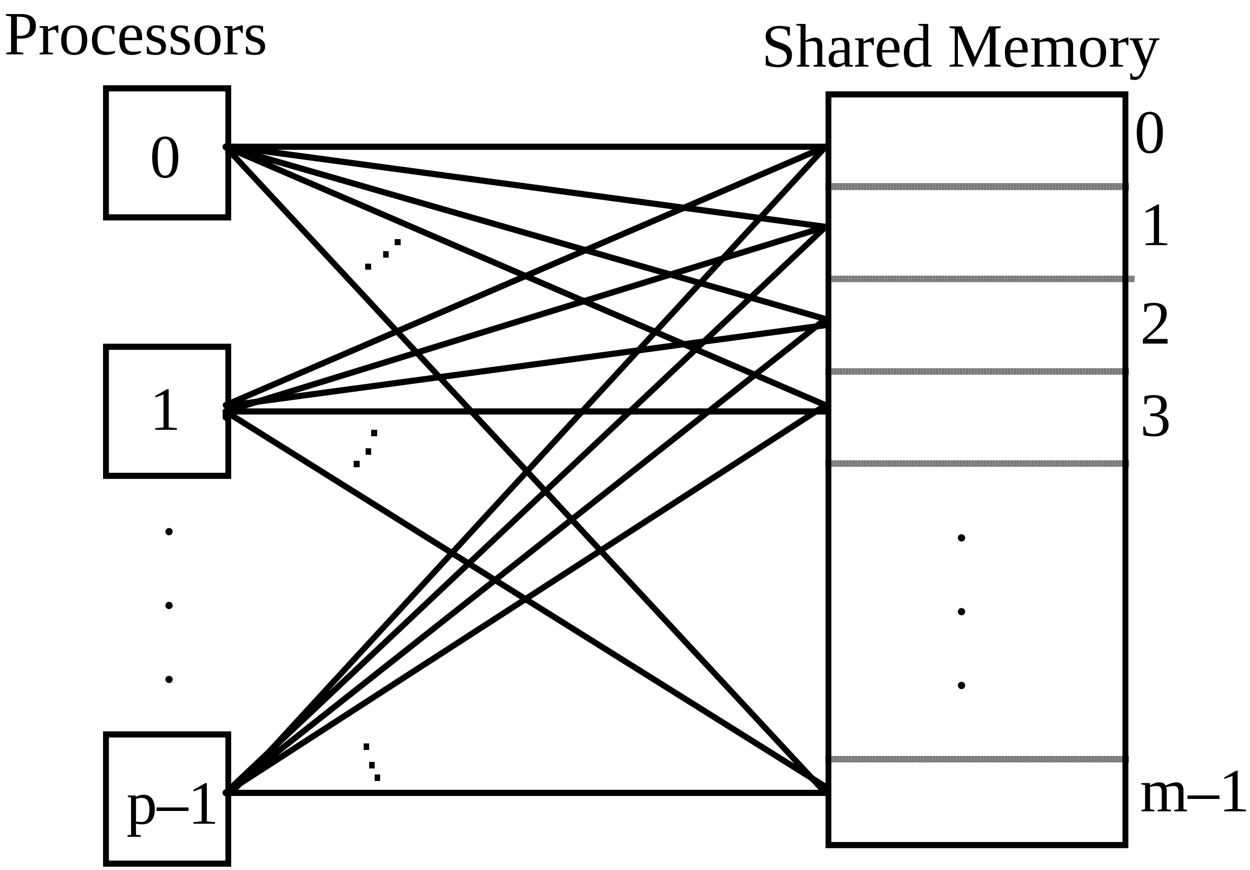



Fig. 4.6 Conceptual view of a parallel random-access machine (PRAM).

Processor *i* can do the following in three phases of one cycle:

1. Fetch a value 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

# Types of PRAM

| Writes to same location / Reads from same location | Exclusive | Concurrent |
|---|---|---|
| **Exclusive** | **EREW**<br>Least “powerful”, most “realistic” | **CREW**<br>Default |
| **Concurrent** | **ERCW**<br>Not useful | **CRCW**<br>Most “powerful”, further subdivided |

Fig. 5.1 Submodels of the PRAM model.

# Types of CRCW PRAM

CRCW submodels are distinguished by the way they treat multiple writes:

Undefined: The value written is undefined (CRCW-U)

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

Common: Allowed only if they all store the same value (CRCW-C)
[ This is sometimes called the consistent-write submodel ]

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

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

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

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

# Power of CRCW PRAM Submodels

Model U is more powerful than model V if $T_U(n) = o(T_V(n))$ for some problem

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 slowdown factor $\Theta(\log p)$.

Intuitive justification for concurrent read emulation (write is similar):

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

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 6 | 1 | |
| 5 | 1 | |
| 2 | 2 | 2 |
| 3 | 2 | |
| 6 | 3 | 3 |
| 1 | 5 | 5 |
| 1 | 6 | 6 |
| 2 | 6 | |

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

# Implications of the CRCW Hierarchy of Submodels

EREW < CREW < CRCW-D < CRCW-C < CRCW-R < CRCW-P

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

Our most powerful PRAM CRCW submodel can be emulated by the least powerful submodel with logarithmic slowdown

Efficient parallel algorithms have polylogarithmic running times

Running time still polylogarithmic after slowdown due to emulation

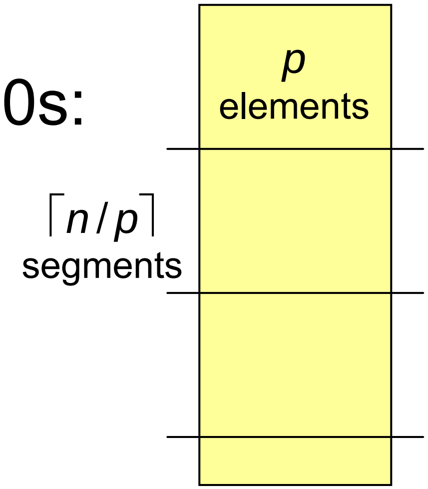
We need not be too concerned with the CRCW submodel used

Simply use whichever submodel is most natural or convenient

# Some Elementary PRAM Computations

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

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




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

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

# 5.2 Data Broadcasting

<u>Making $p$ copies of $B[0]$ by recursive doubling</u>

for $k = 0$ to $\lceil \log_2 p \rceil - 1$
  Proc $j$, $0 \le j < p$, do
 Copy $B[j]$ into $B[j + 2^k]$
endfor

Can modify the algorithm so that redundant copying does not occur and array bound is not exceeded

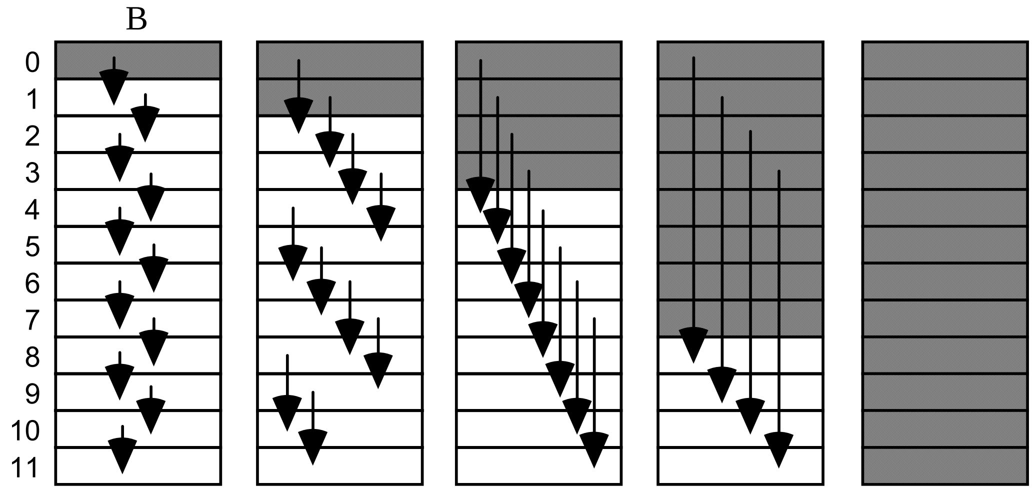



Fig. 5.2 Data broadcasting in EREW PRAM via recursive doubling.

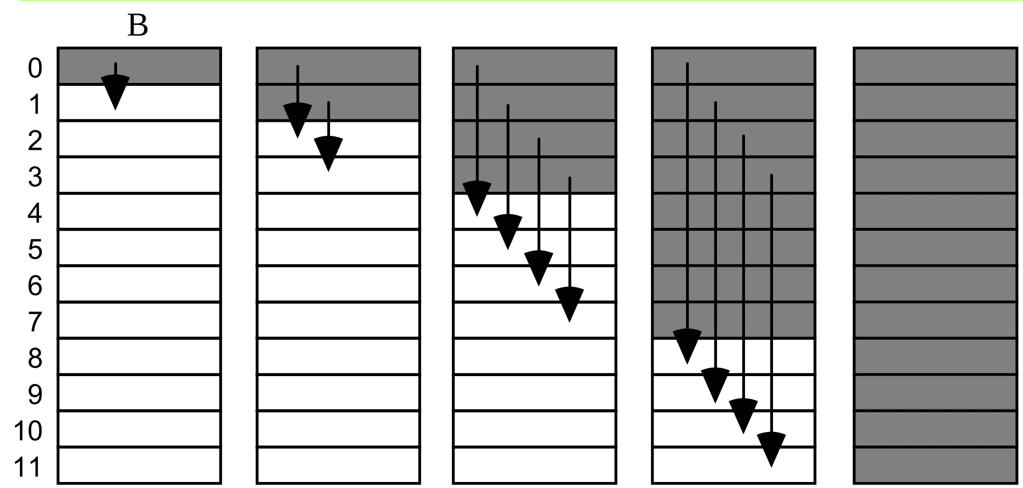
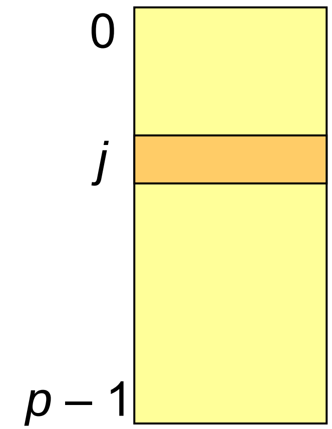



Fig. 5.3 EREW PRAM data broadcasting without redundant copying.

# All-to-All Broadcasting on EREW PRAM

<u>EREW PRAM algorithm for all-to-all broadcasting</u>

Processor $j$, $0 \le j < p$, write own data value into $B[j]$
for $k$ = 1 to $p - 1$ Processor $j$, $0 \le j < p$, do
    Read the data value in $B[(j + k) \bmod p]$
endfor




This O($p$)-step algorithm is time-optimal

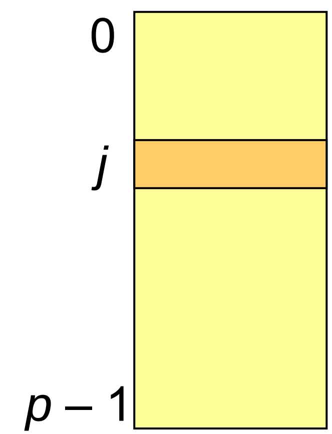
<u>Naive EREW PRAM sorting algorithm (using all-to-all broadcasting)</u>

Processor $j$, $0 \le j < p$, write 0 into $R[\,j\,]$
for $k$ = 1 to $p - 1$ Processor $j$, $0 \le j < p$, do
    $l := (j + k) \bmod p$
    if $S[\,l\,] < S[\,j\,]$ or $S[\,l\,] = S[\,j\,]$ and $l < j$
    then $R[\,j\,] := R[\,j\,] + 1$
    endif
endfor
Processor $j$, $0 \le j < p$, write $S[\,j\,]$ into $S[R[\,j\,]]$

This O($p$)-step sorting algorithm is far from optimal; sorting is possible in O(log $p$) time

# Class Participation: Broadcast-Based Sorting

Each person write down an arbitrary nonnegative integer with 3 or fewer digits on a piece of paper




Students take turn broadcasting their numbers by calling them out aloud

Each student puts an X on paper for every number called out that is smaller than his/her own number, or is equal but was called out before the student’s own value

Each student counts the number of Xs on paper to determine the rank of his/her number

Students call out their numbers in order of the computed rank

# 5.3 Semigroup or Fan-in Computation

<u>EREW PRAM semigroup computation algorithm</u>

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

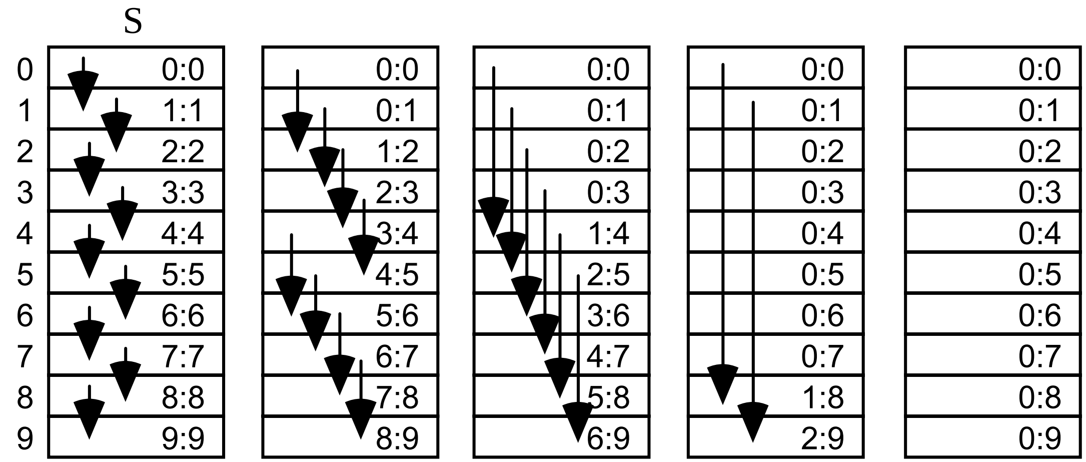



Fig. 5.4 Semigroup computation in EREW PRAM.

This algorithm is optimal for PRAM, but its speedup of $O(p / \log p)$ is not

If we use $p$ processors on a list of size $n = O(p \log p)$, then optimal speedup can be achieved

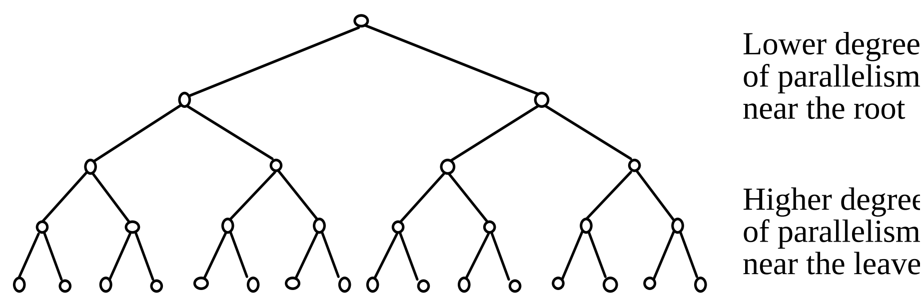



Fig. 5.5 Intuitive justification of why parallel slack helps improve the efficiency.

# 5.4 Parallel Prefix Computation

Same as the first part of semigroup computation (no final broadcasting)

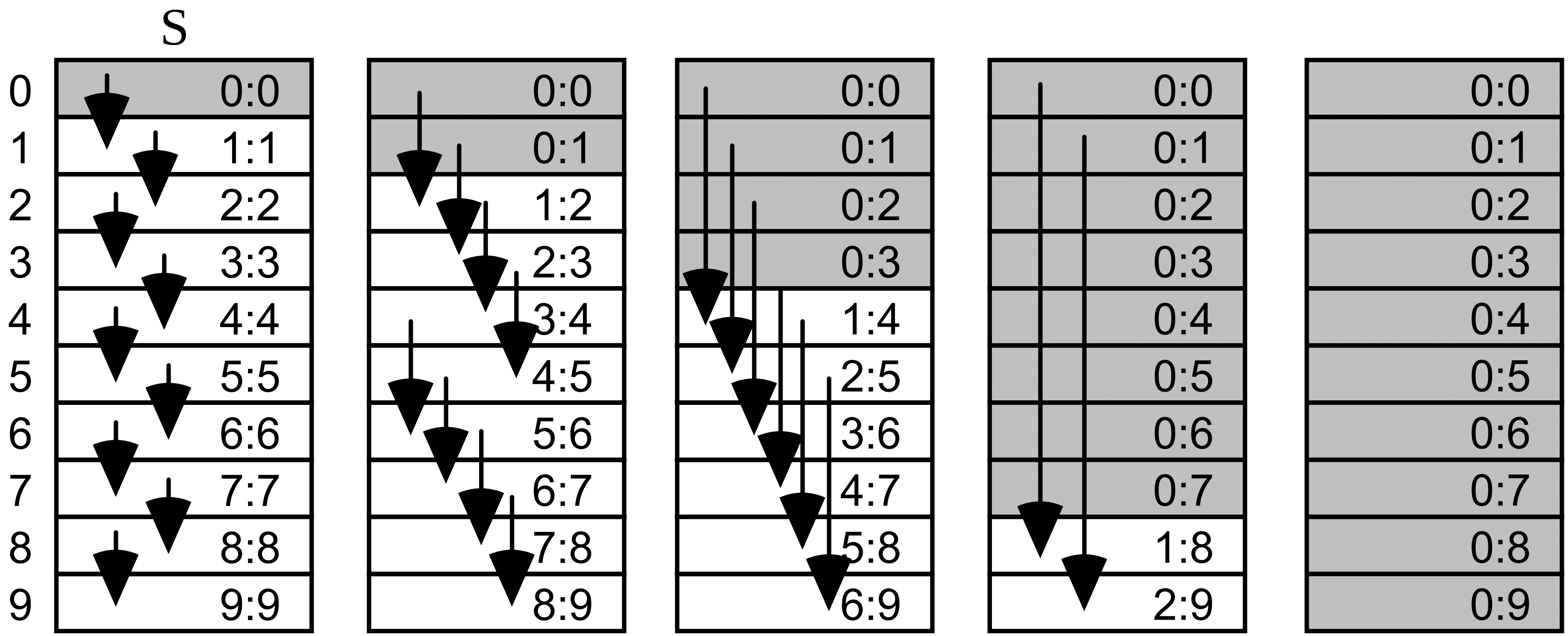
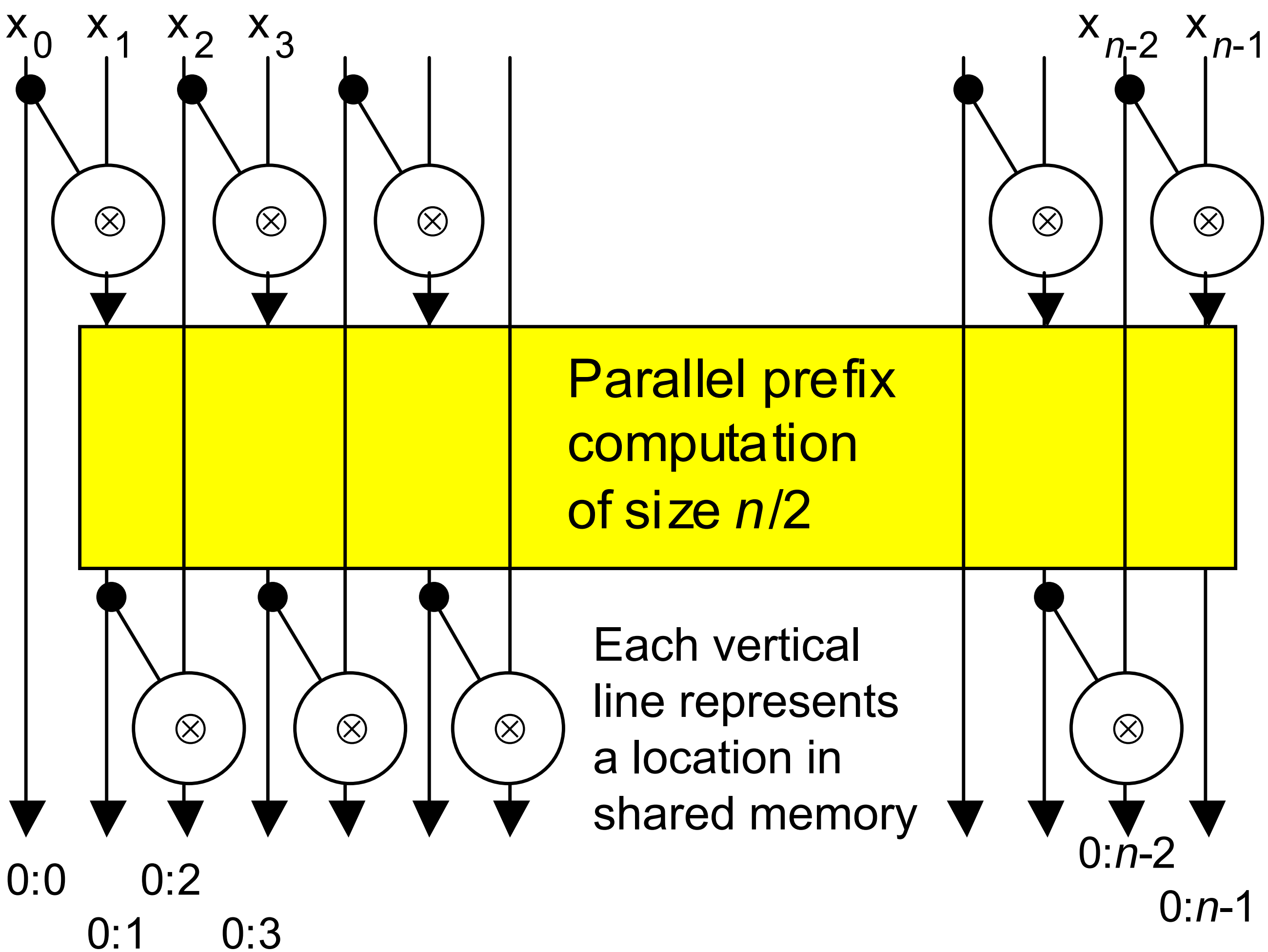



Fig. 5.6 Parallel prefix computation in EREW PRAM via recursive doubling.

# A Divide-and-Conquer Parallel-Prefix Algorithm



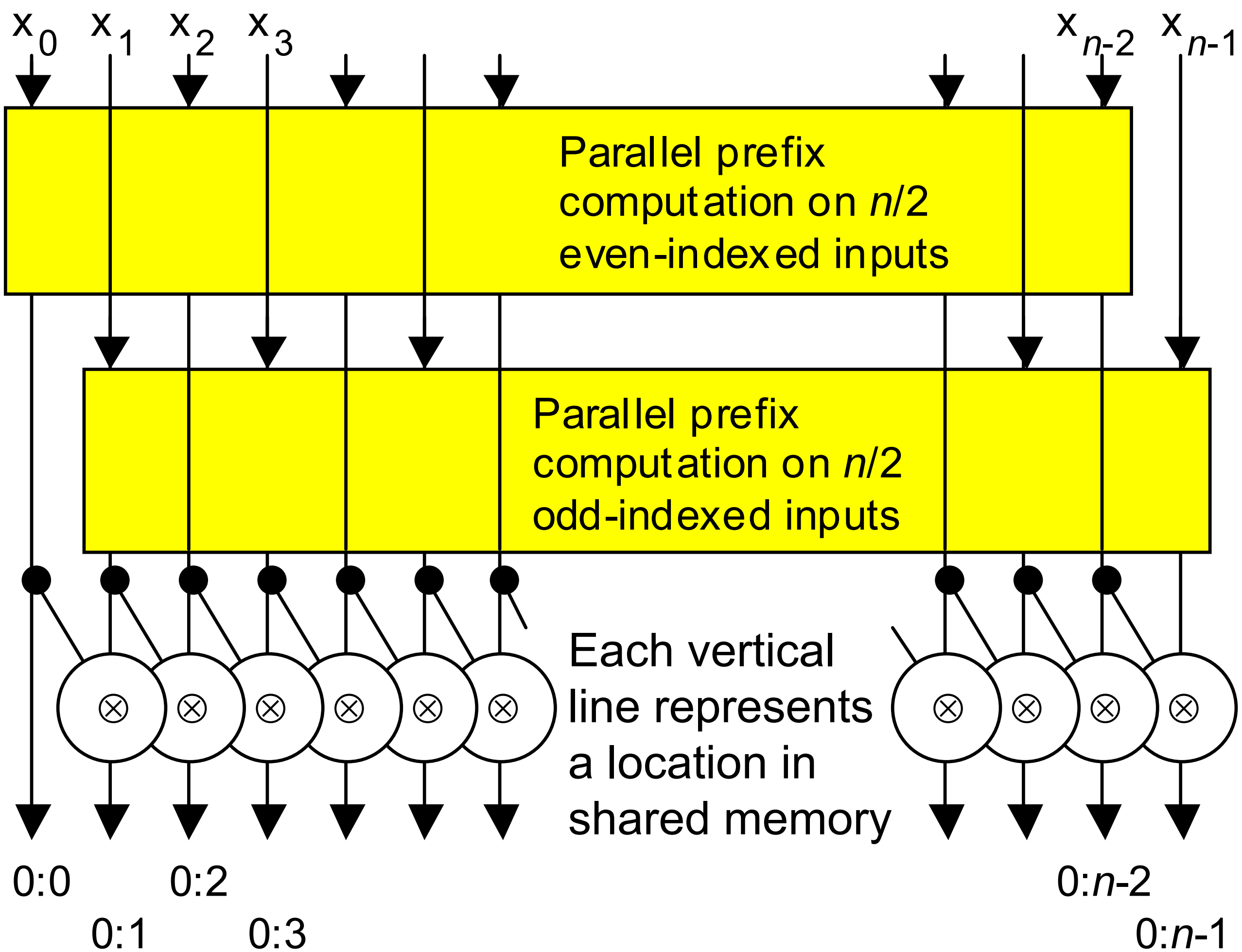

$T(p) = T(p/2) + 2$

$T(p) \cong 2 \log_2 p$

In hardware, this is the basis for Brent-Kung carry-lookahead adder

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

# Another Divide-and-Conquer Algorithm




$T(p) = T(p/2) + 1$

$T(p) = \log_2 p$

Strictly optimal algorithm, but requires commutativity

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

# 5.5 Ranking the Elements of a Linked List

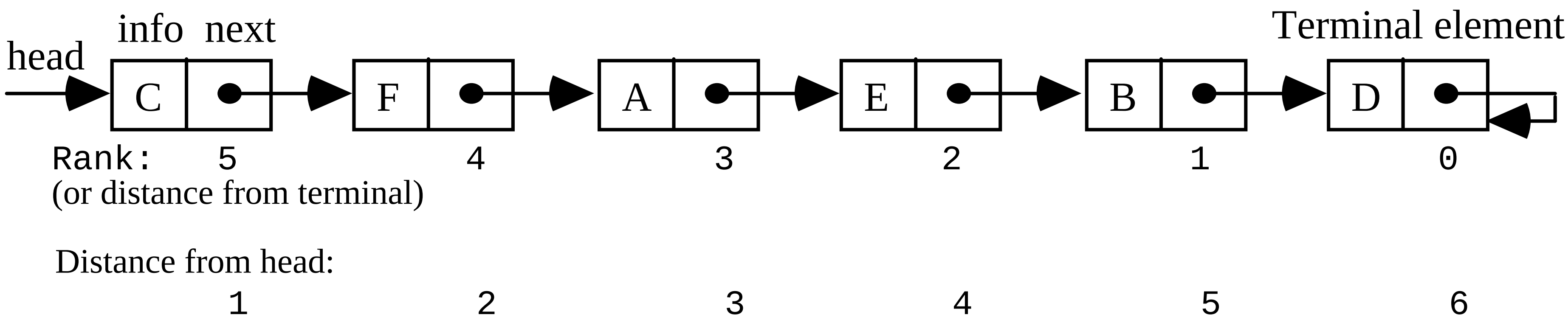



Fig. 5.9 Example linked list and the ranks of its elements.

**List ranking appears to be hopelessly sequential; one cannot get to a list element except through its predecessor!**

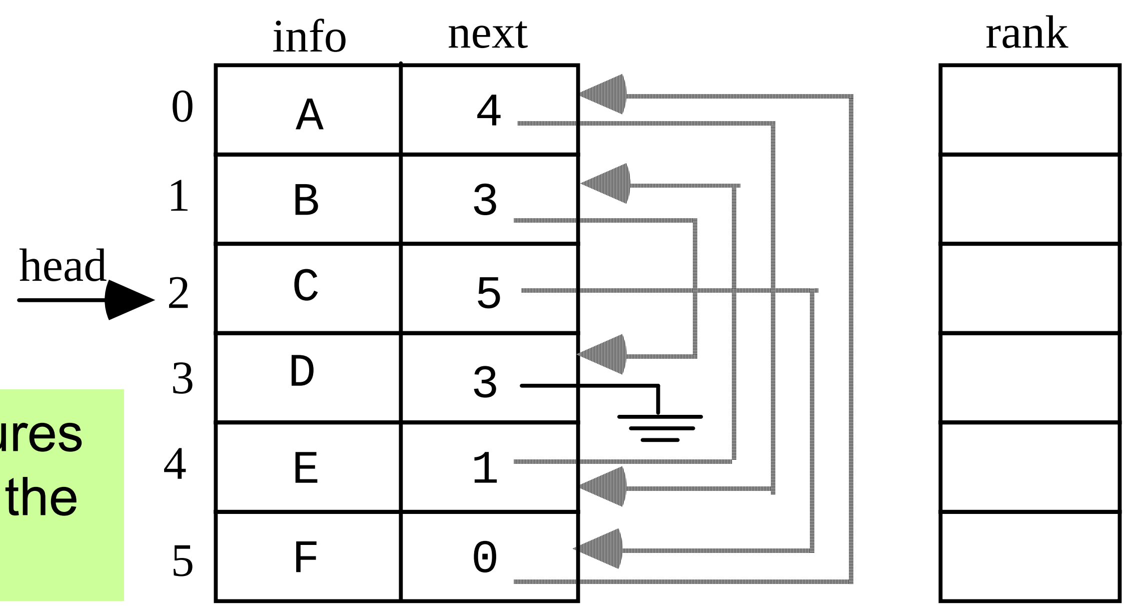
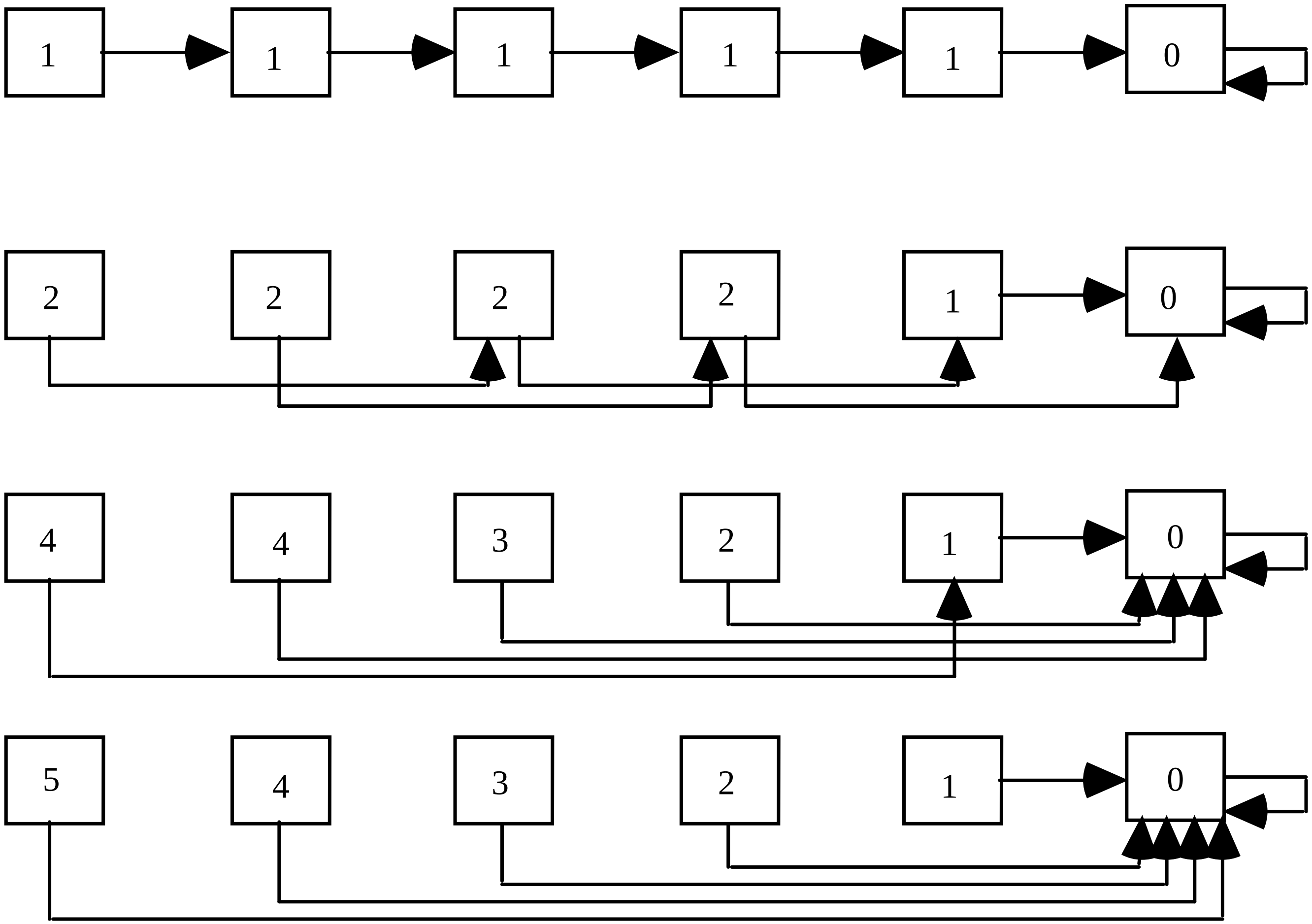



Fig. 5.10 PRAM data structures representing a linked list and the ranking results.

# List Ranking via Recursive Doubling




Many problems that appear to be unparallelizable to the uninitiated are parallelizable; Intuition can be quite misleading!

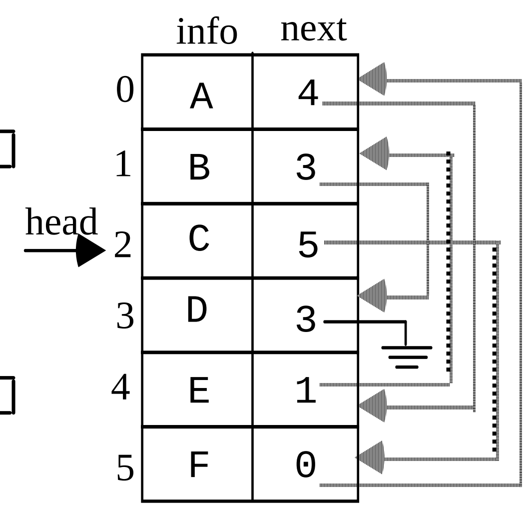



Fig. 5.11 Element ranks initially and after each of the three iterations.

# PRAM List Ranking Algorithm

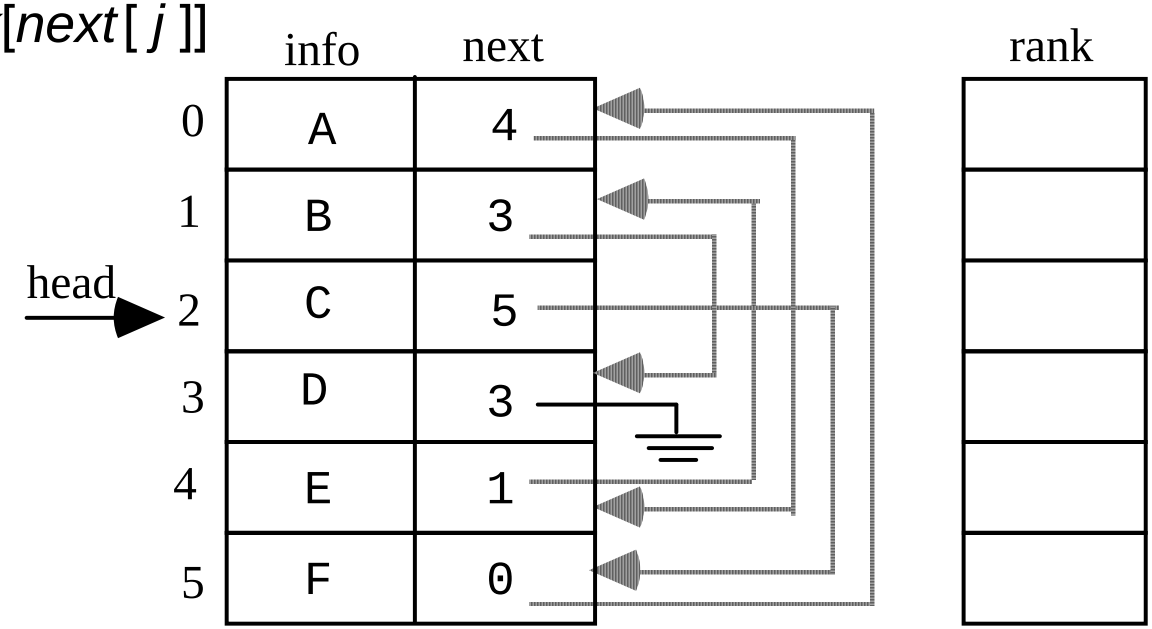
PRAM list ranking algorithm (via pointer jumping)

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

If we do not want to modify the original list, we simply make a copy of it first, in constant time

Question: Which PRAM submodel is implicit in this algorithm?

Answer: CREW



# 5.6 Matrix Multiplication

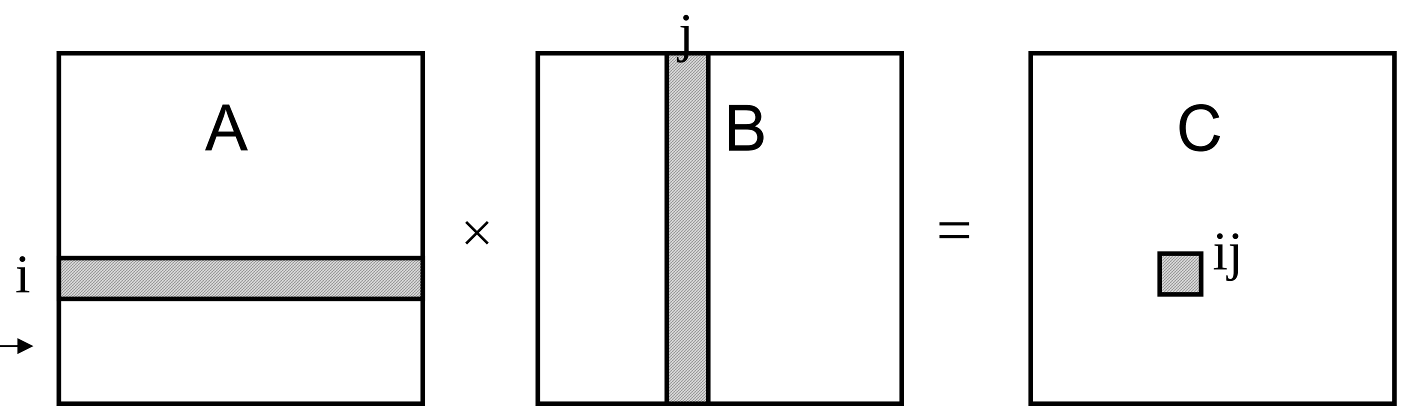
<u>Sequential matrix multiplication</u>

for $i = 0$ to $m - 1$ do<br>
    for $j = 0$ to $m - 1$ do<br>
        $t := 0$<br>
        for $k = 0$ to $m - 1$ do<br>
            $t := t + a_{ik}b_{kj}$<br>
        endfor<br>
        $c_{ij} := t$<br>
    endfor<br>
endfor

PRAM solution with $m^3$ processors: each processor does one multiplication (not very efficient)

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

$m \times m$ matrices



# PRAM Matrix Multiplication with $m^2$ Processors

PRAM matrix multiplication using $m^2$ processors

Proc $(i, j)$, $0 \leq i, j < m$, do
begin
  $t := 0$
  for $k = 0$ to $m - 1$ do
    $t := t + a_{ik} b_{kj}$
  endfor
  $c_{ij} := t$
end

Processors are numbered $(i, j)$, instead of 0 to $m^2 - 1$

$\Theta(m)$ steps: Time-optimal

CREW model is implicit

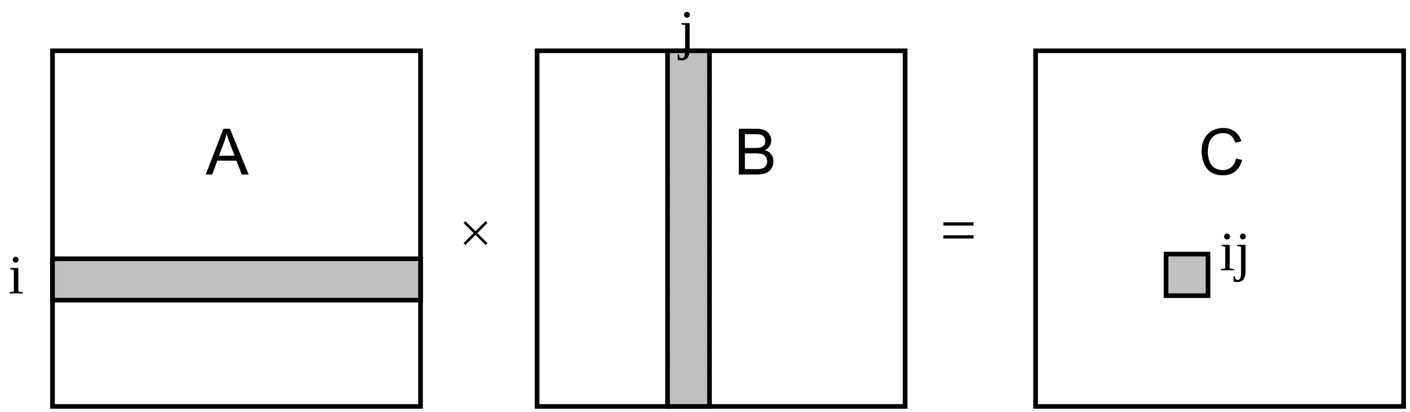



Fig. 5.12 PRAM matrix multiplication; $p = m^2$ processors.

# PRAM Matrix Multiplication with $m$ Processors

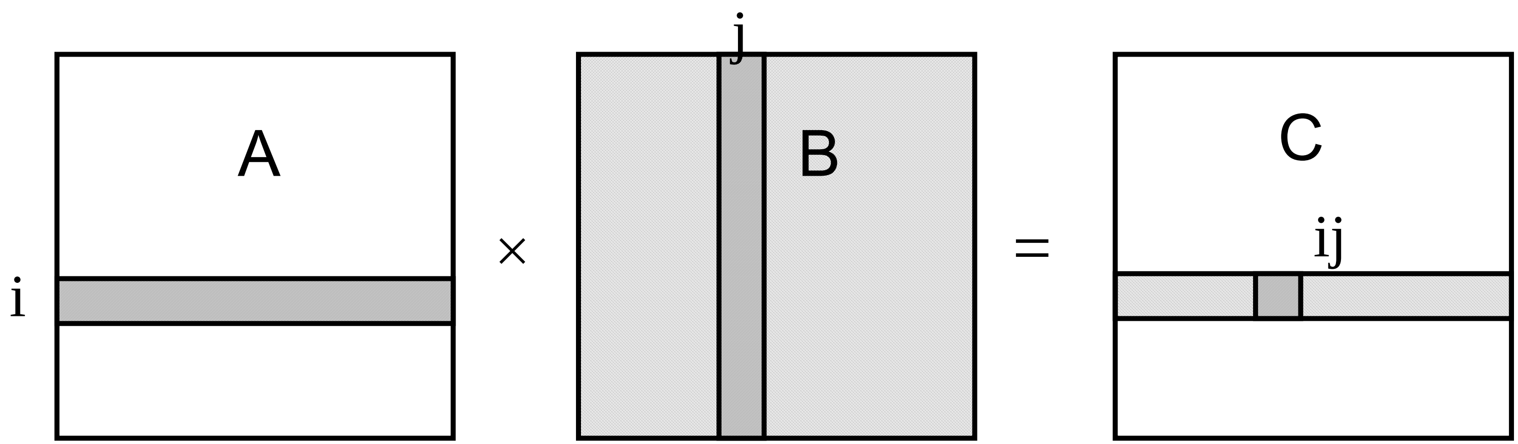
PRAM matrix multiplication using $m$ processors

for $j = 0$ to $m - 1$ Proc $i$, $0 \le i < m$, do
  $t := 0$
  for $k = 0$ to $m - 1$ do
    $t := t + a_{ik}b_{kj}$
  endfor
  $c_{ij} := t$
endfor

$\Theta(m^2)$ steps: Time-optimal

CREW model is implicit

Because the order of multiplications is immaterial, accesses to $B$ can be skewed to allow the EREW model



# PRAM Matrix Multiplication with Fewer Processors

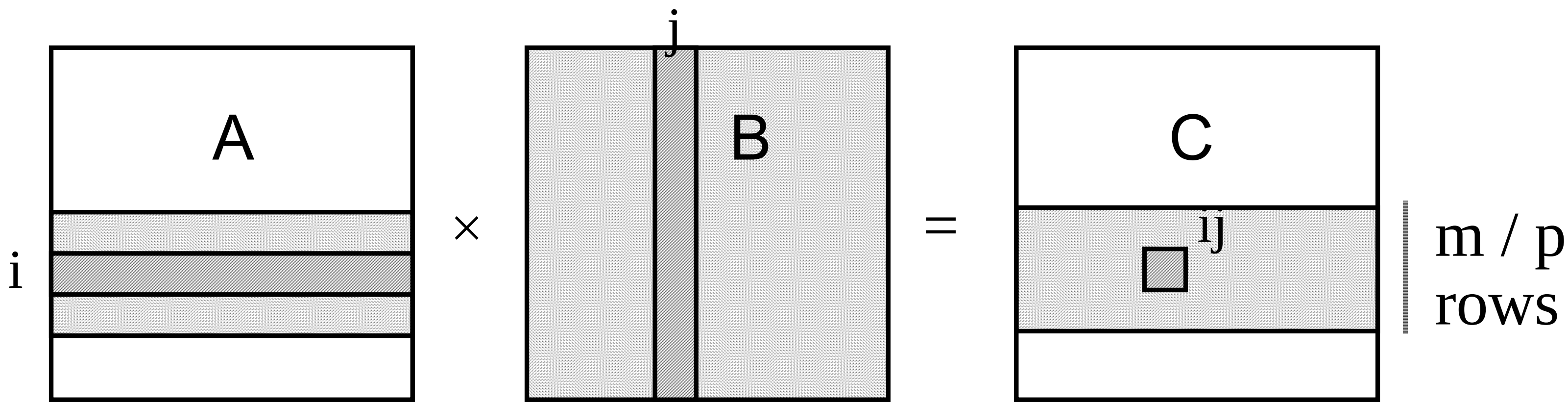
Algorithm is similar, except that each processor is in charge of computing $m / p$ rows of $C$

$\Theta(m^3/p)$ steps: Time-optimal

EREW model can be used

A drawback of all algorithms thus far is that only two arithmetic operations (one multiplication and one addition) are performed for each memory access.

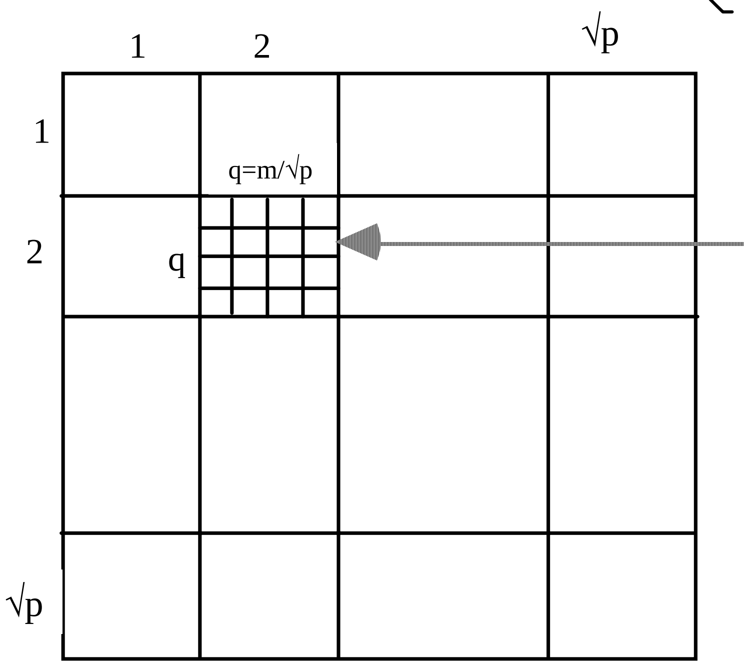
This is particularly costly for NUMA shared-memory machines.



# More Efficient Matrix Multiplication (for NUMA)

Partition the matrices into *p* square blocks

$$\left[\begin{array}{c:c} A & B \\ \hdashline C & D \end{array}\right] \times \left[\begin{array}{c:c} E & F \\ \hdashline G & H \end{array}\right] = \left[\begin{array}{c:c} AE+BG & AF+BH \\ \hdashline CE+DG & CF+DH \end{array}\right]$$




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

Block matrix multiplication follows the same algorithm as simple matrix multiplication.

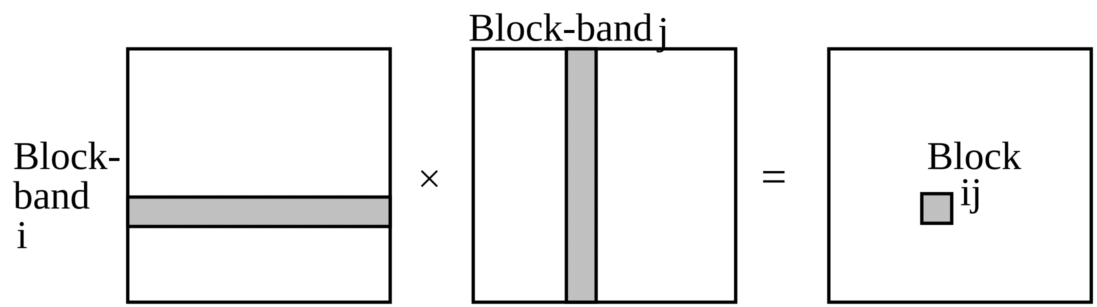
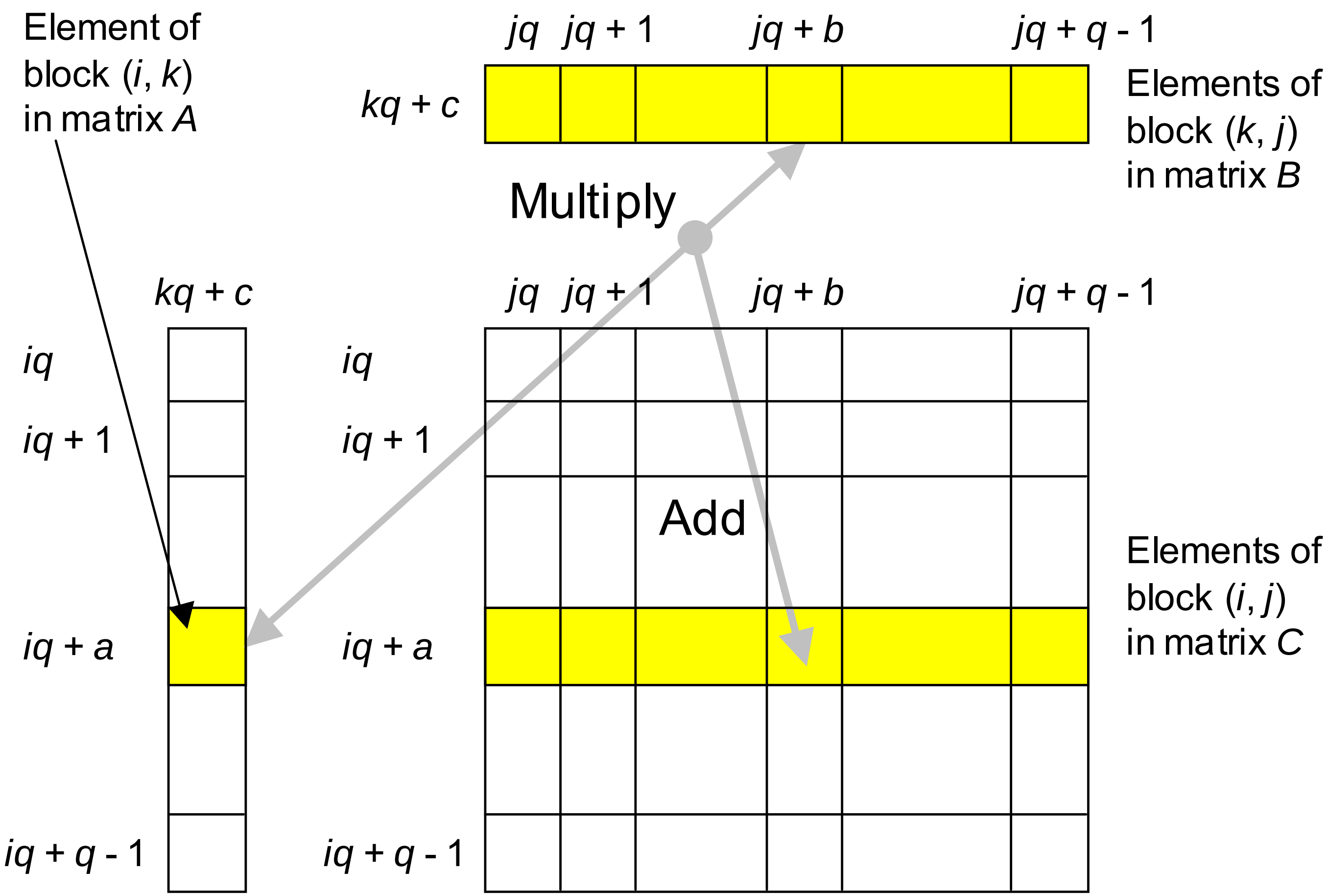



Fig. 5.13 Partitioning the matrices for block matrix multiplication .

# Details of Block Matrix Multiplication




A multiply-add computation on $q \times q$ blocks needs $2q^2 = 2m^2/p$ memory accesses and $2q^3$ arithmetic operations

So, $q$ arithmetic operations are done per memory access

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 $C$.

# 6 More Shared-Memory Algorithms

Develop PRAM algorithm for more complex problems:

- Must present background on the problem in some cases
- Discuss some practical issues such as data distribution

| Topics in This Chapter | |
|---|---|
| 6.1 | Sequential Ranked-Based Selection |
| 6.2 | A Parallel Selection Algorithm |
| 6.3 | A Selection-Based Sorting Algorithm |
| 6.4 | Alternative Sorting Algorithms |
| 6.5 | Convex Hull of a 2D Point Set |
| 6.6 | Some Implementation Aspects |

# 8.1 Searching and Dictionary Operations

Parallel $p$-ary search on PRAM

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

Speedup $\cong \log p$

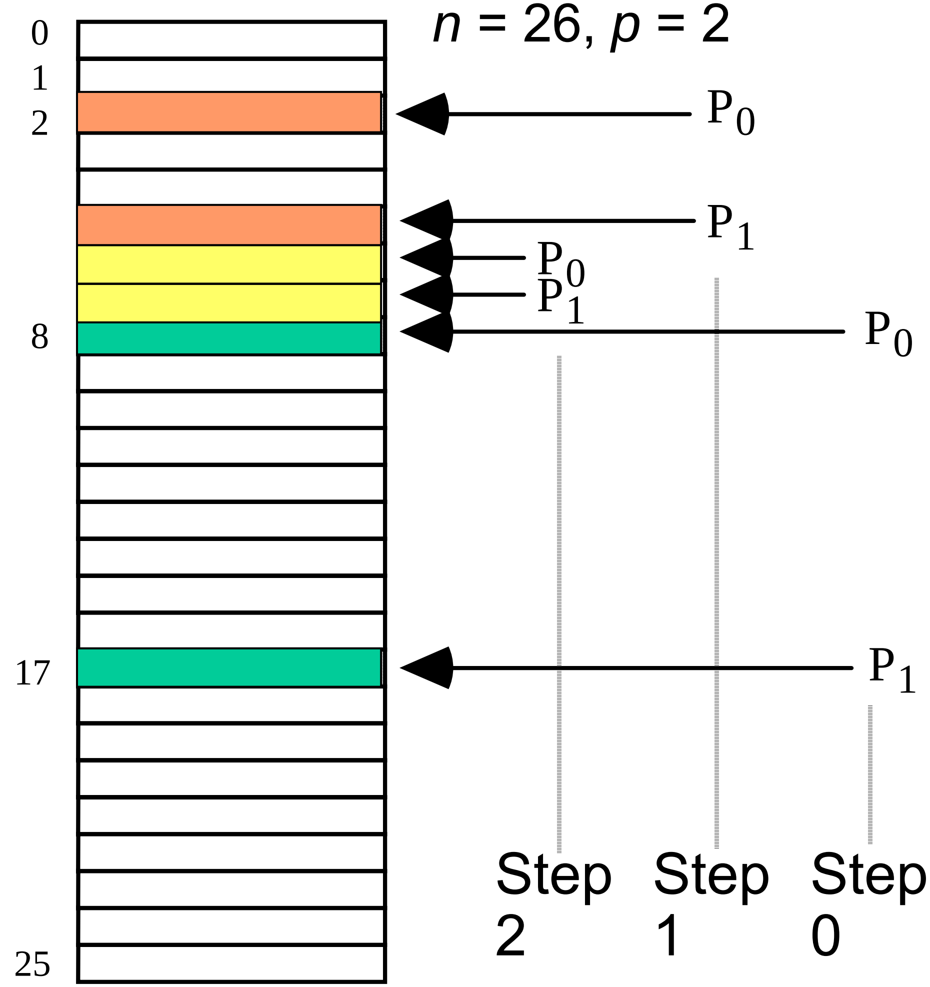
Optimal: no comparison-based search algorithm can be faster

A single search in a sorted list can't be significantly speeded up through parallel processing, but all hope is not lost:

Dynamic data (sorting overhead)

Batch searching (multiple lookups)

Example:
$n = 26$, $p = 2$



# 6.1 Sequential Ranked-Based Selection

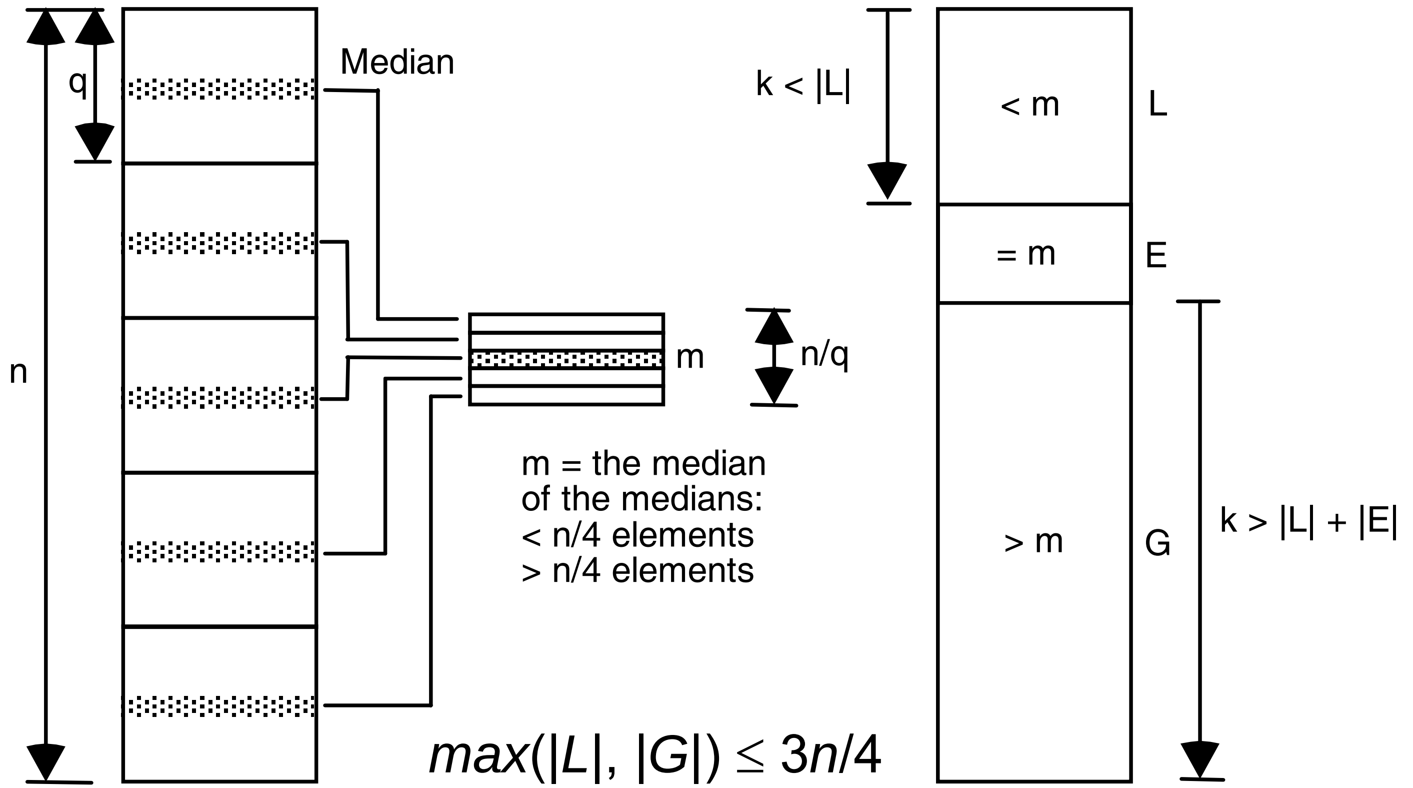
Selection: Find the (or a) *k*th smallest among *n* elements

Example: 5th smallest element in the following list is 1:

`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`

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

But linear-time sequential algorithm can be developed

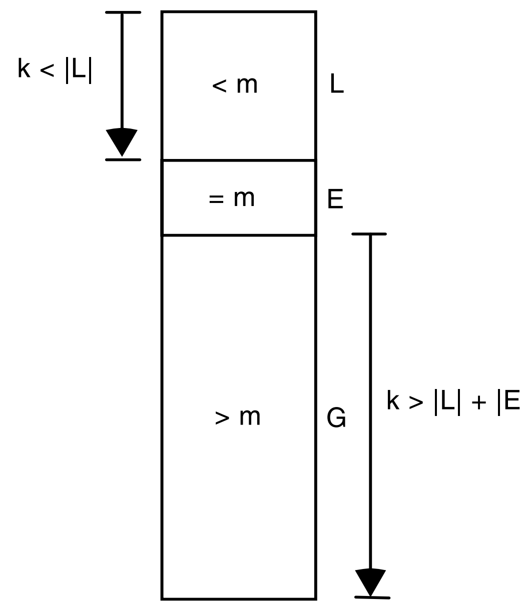



$max(|L|, |G|) \leq 3n/4$

# Linear-Time Sequential Selection Algorithm

<u>Sequential rank-based selection algorithm *select*($S$, $k$)</u>

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



# Algorithm Complexity and Examples

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

We must have $q \geq 5$;
for $q = 5$, the solution is $T(n) = 20cn$

```
    ------------- 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*.**

# 6.2 A Parallel Selection Algorithm

<u>Parallel rank-based selection algorithm *PRAMselect*($S$, $k$, $p$)</u>

1. if $|S| < 4$
   then sort $S$ and return the $k$th 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

$O(n^x)$

2. $m = PRAMselect(T, |T|/2, p)$ {median of the medians}

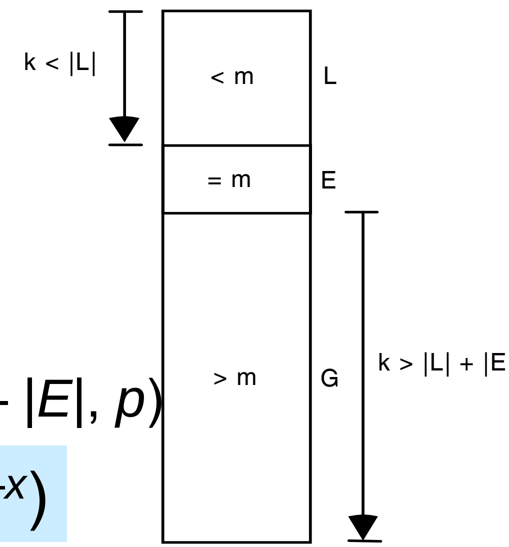
$T(n^{1-x}, p)$

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 $> m$

$O(n^x)$

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
   endif

$T(3n/4, p)$




Let $p = O(n^{1-x})$

# Algorithm Complexity and Efficiency

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

The solution is $O(n^x)$; verify by substitution

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

Efficiency = $\Omega(1)$

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

Remember
$p = O(n^{1-x})$

**What happens if we set $x$ to 1? ( i.e., use one processor)**

**$T(n, 1) = O(n^x) = O(n)$**

**What happens if we set $x$ to 0? (i.e., use $n$ processors)**

**$T(n, n) = O(n^x) = O(1)$ ?**

No, because in asymptotic analysis, we ignored several $O(\log n)$ terms compared with $O(n^x)$ terms

# 6.3 A Selection-Based Sorting Algorithm

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

$O(1)$ — 1. if $|S| < k$ then return *quicksort*($S$)

$O(n^x)$ — 2. for $i = 1$ to $k - 1$ do
  $m_j$ := *PRAMselect*($S$, $i|S|/k$, $p$) {let $m_0 := -\infty$; $m_k := +\infty$}
endfor

$O(n^x)$ — 3. for $i = 0$ to $k - 1$ do
  make the sublist $T(i)$ from elements of $S$ in $(m_i, m_{i+1})$
endfor

$T(n/k, 2p/k)$ — 4. for $i = 1$ to $k/2$ do in parallel
  *PRAMselectionsort*($T(i)$, $2p/k$)
  {$p/(k/2)$ proc's used for each of the $k/2$ subproblems}
endfor

$T(n/k, 2p/k)$ — 5. for $i = k/2 + 1$ to $k$ do in parallel
  *PRAMselectionsort*($T(i)$, $2p/k$)
endfor

Let $p = n^{1-x}$
and $k = 2^{1/x}$

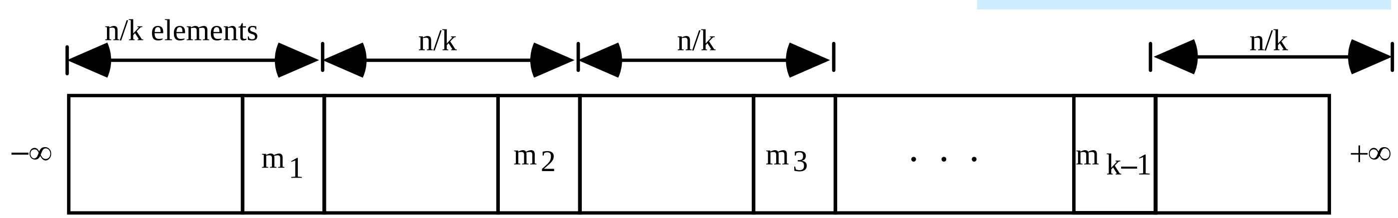



Fig. 6.1 Partitioning of the sorted list for selection-based sorting.

# Algorithm Complexity and Efficiency

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

The solution is $O(n^x \log n)$; verify by substitution

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

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

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

**What happens if we set $x$ to 1? ( i.e., use one processor)**

**$T(n, 1) = O(n^x \log n) = O(n \log n)$**

Remember $p = O(n^{1-x})$

**Our asymptotic analysis is valid for $x > 0$ but not for $x = 0$;**

**i.e., *PRAMselectionsort* cannot sort $p$ keys in optimal $O(\log p)$ time.**

# Example of Parallel Sorting

$S$: 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 for $k = 4$ (i.e., $x = ½$ and $p = n^{1/2}$ processors):

| | |
|---|---|
| | $m_0 = -\infty$ |
| $n/k = 25/4 \cong 6$ | $m_1 = PRAMselect(S, 6, 5) = 2$ |
| $2n/k = 50/4 \cong 13$ | $m_2 = PRAMselect(S, 13, 5) = 4$ |
| $3n/k = 75/4 \cong 19$ | $m_3 = PRAMselect(S, 19, 5) = 6$ |
| | $m_4 = +\infty$ |

$m_0$ $m_1$ $m_2$ $m_3$ $m_4$

$T$: - - - - - 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 (assume $p << \sqrt{n}$)

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

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

<u>Parallel randomized sort *PRAMrandomsort*($S$, $p$)</u>

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 $T$
   {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 $T(j)$

# Parallel Radixsort

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

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

Records are stably sorted by the value of the $i$th key bit

Binary forms

| 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) |

Question: How are the data movements performed?

# Data Movements in Parallel Radixsort

| 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) |

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

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

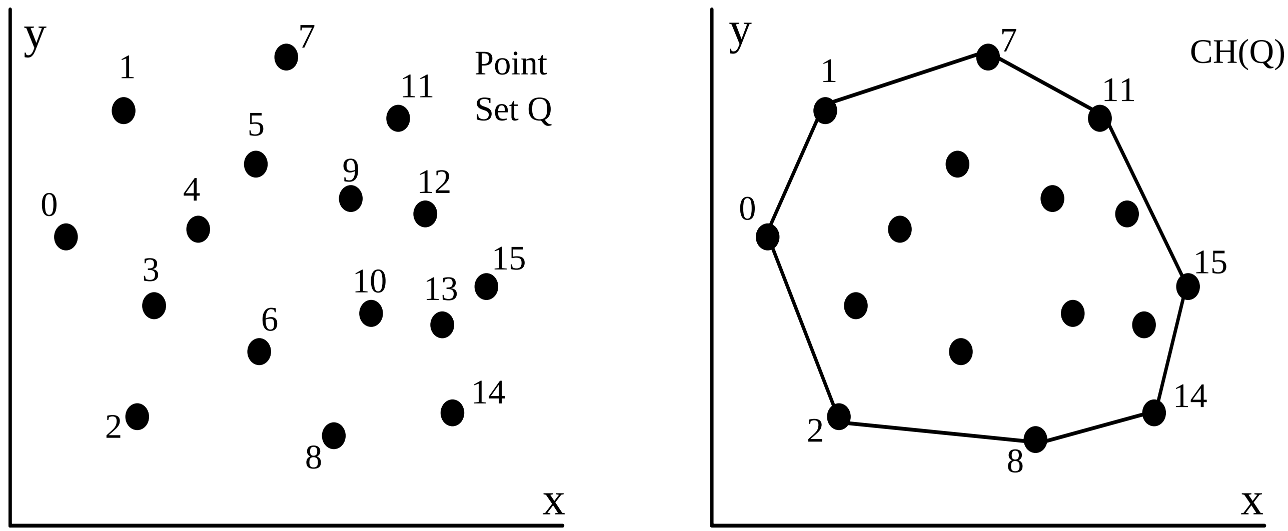



Fig. 6.2 Defining the convex hull problem.

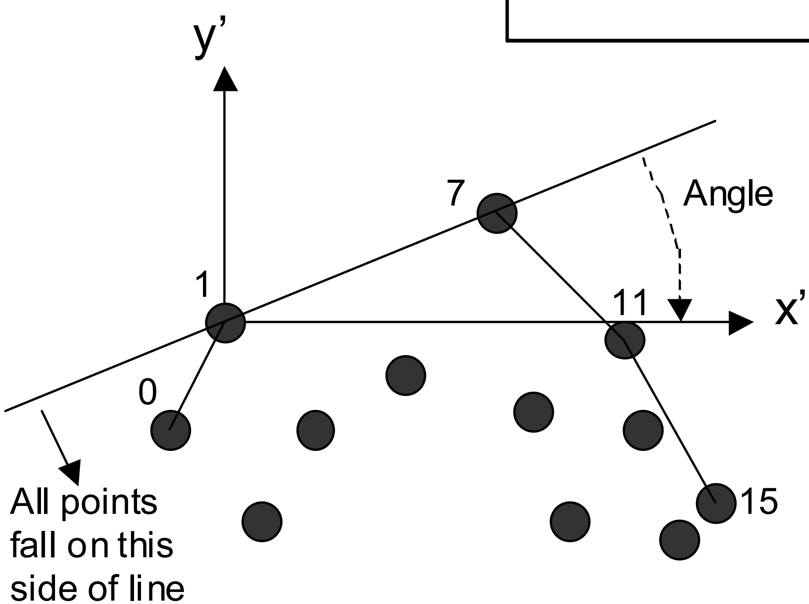



Fig. 6.3 Illustrating the properties of the convex hull.

# PRAM Convex Hull Algorithm

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

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

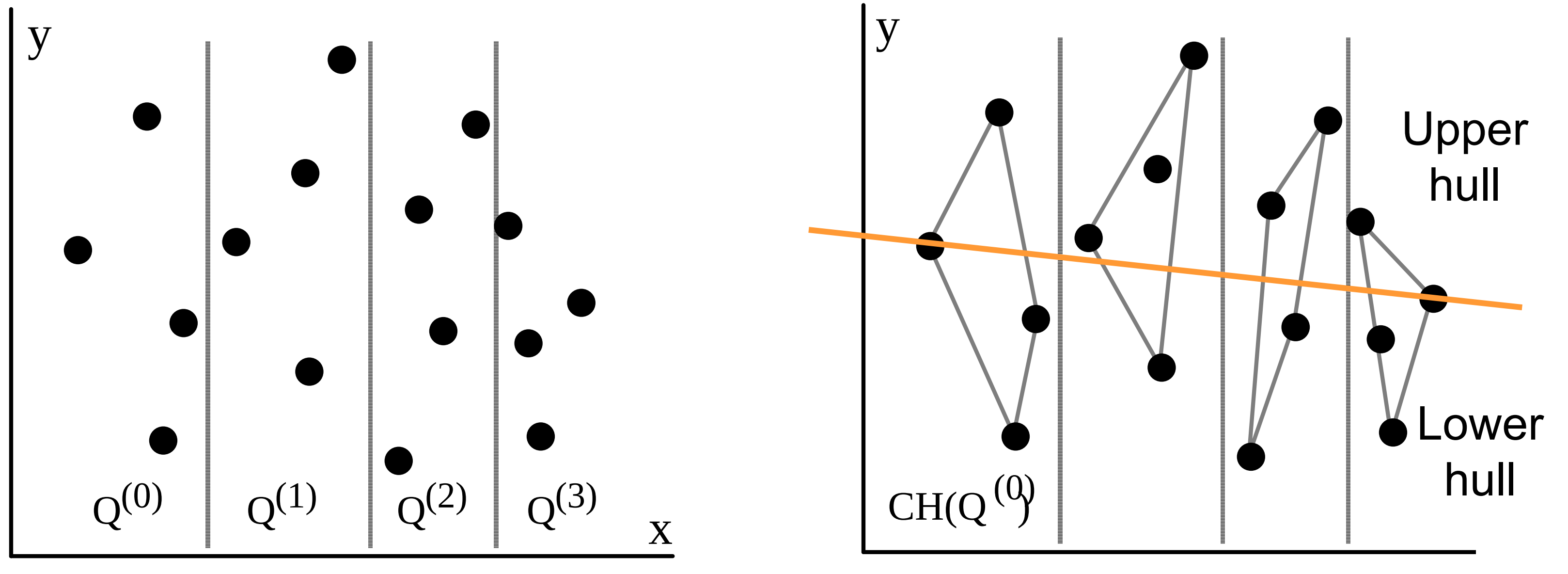
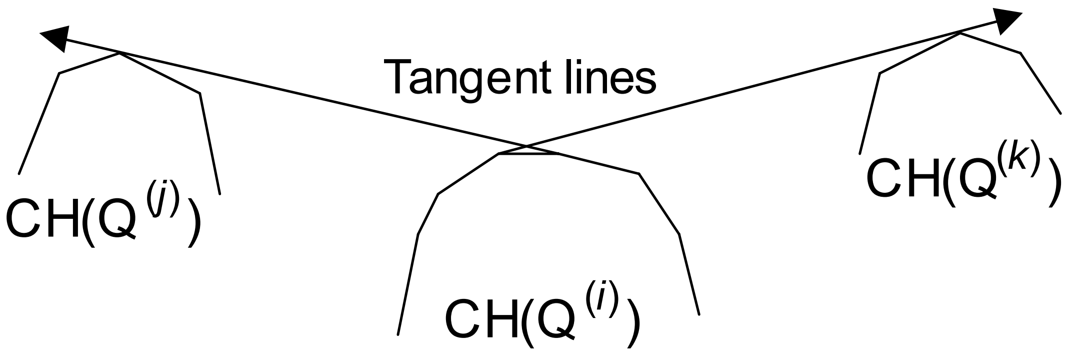
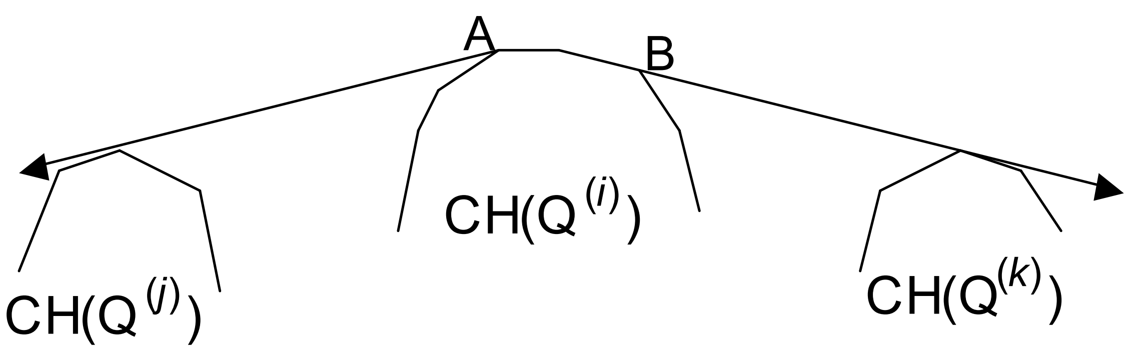



Fig. 6.4 Multiway divide and conquer for the convex hull problem

# Merging of Partial Convex Hulls




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




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

Tangent lines are found through binary search in log time

Analysis:

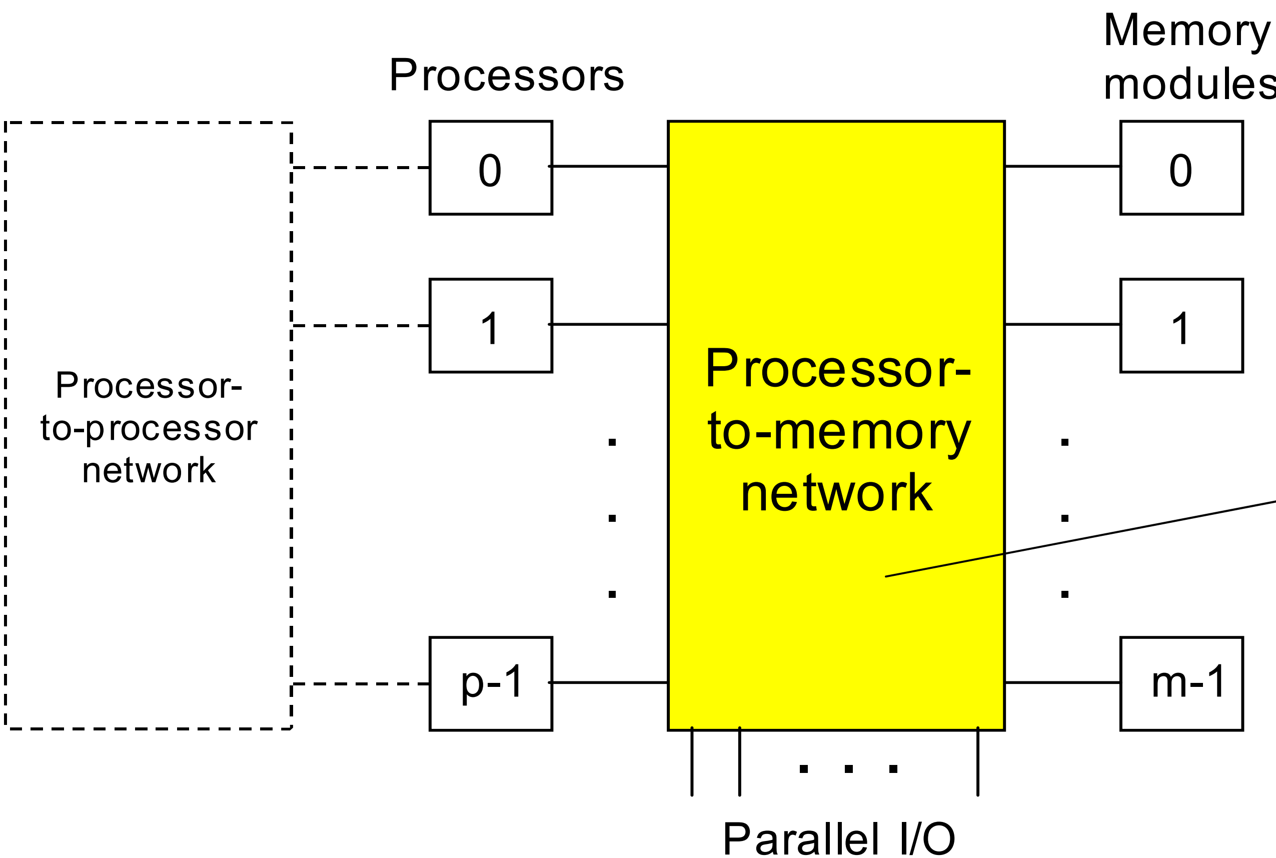
$T(p, p)$
$= T(p^{1/2}, p^{1/2}) + c \log p$
$\cong 2c \log p$

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

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

# 6.6 Some Implementation Aspects

This section has been expanded; it will eventually become a separate chapter

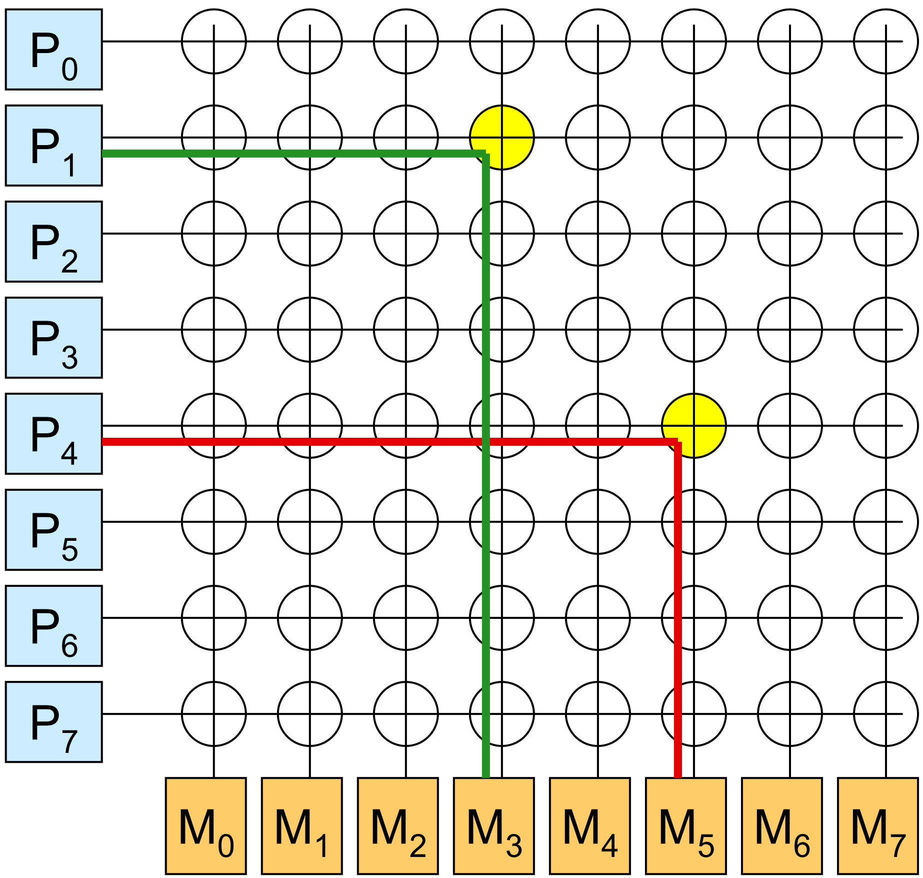



**Options:**
Crossbar
Bus(es)
MIN

*Bottleneck*
*Complex*
*Expensive*

Fig. 4.3 A parallel processor with global (shared) memory.

# Processor-to-Memory Network




An 8 × 8 crossbar switch

Crossbar switches offer full permutation capability (they are *nonblocking*), but are complex and expensive: $O(p^2)$

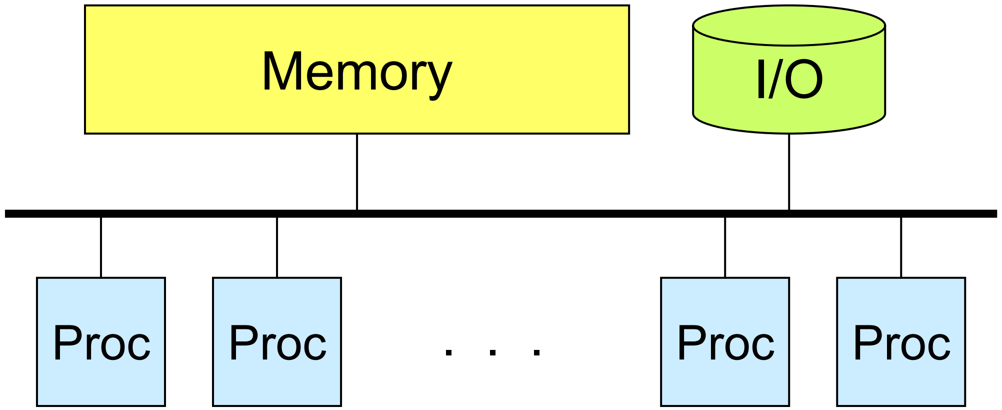
Even with a permutation network, full PRAM functionality is not realized: two processors cannot access different addresses in the same memory module

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

# Bus-Based Interconnections

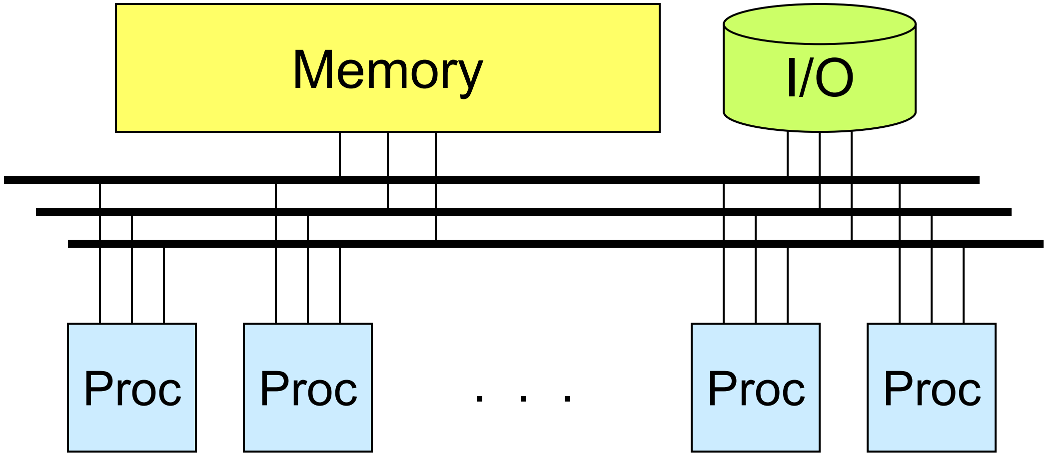
**Single-bus system:**

Bandwidth bottleneck
Bus loading limit
Scalability: very poor
Single failure point
Conceptually simple
Forced serialization




**Multiple-bus system:**

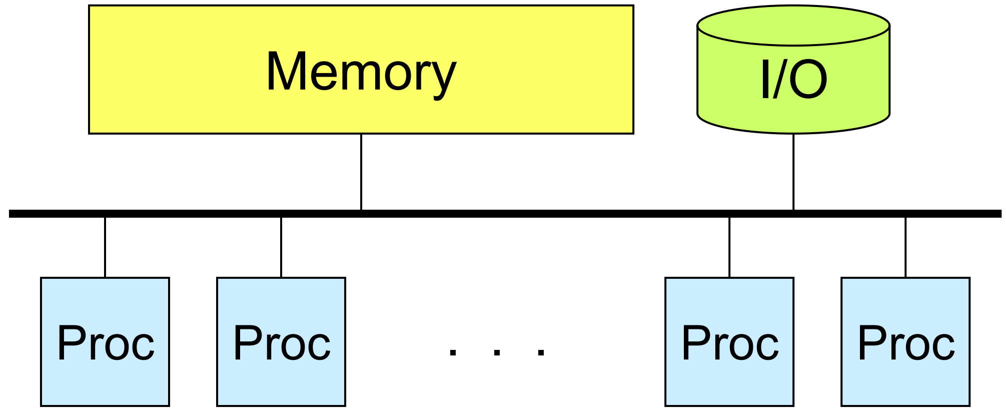
Bandwidth improved
Bus loading limit
Scalability: poor
More robust
More complex scheduling
Simple serialization



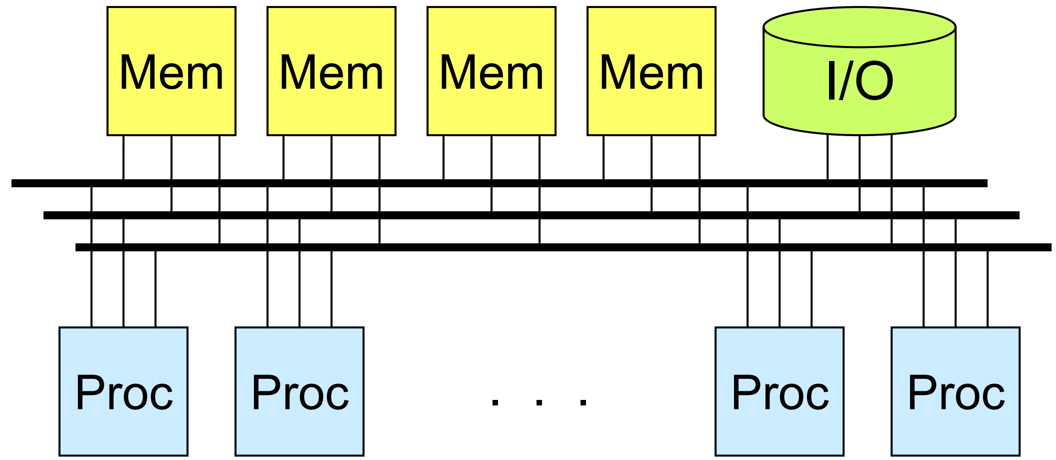
# Back-of-the-Envelope Bus Bandwidth Calculation

**Single-bus system:**
Bus frequency: 0.5 GHz
Data width: 256 b (32 B)
Mem. Access: 2 bus cycles
(0.5G)/2 × 32 = 8 GB/s

Bus cycle = 2 ns
Memory cycle = 100 ns
1 mem. cycle = 50 bus cycles




**Multiple-bus system:**
Peak bandwidth multiplied by the number of buses (actual bandwidth is likely to be much less)



# Hierarchical Bus Interconnection

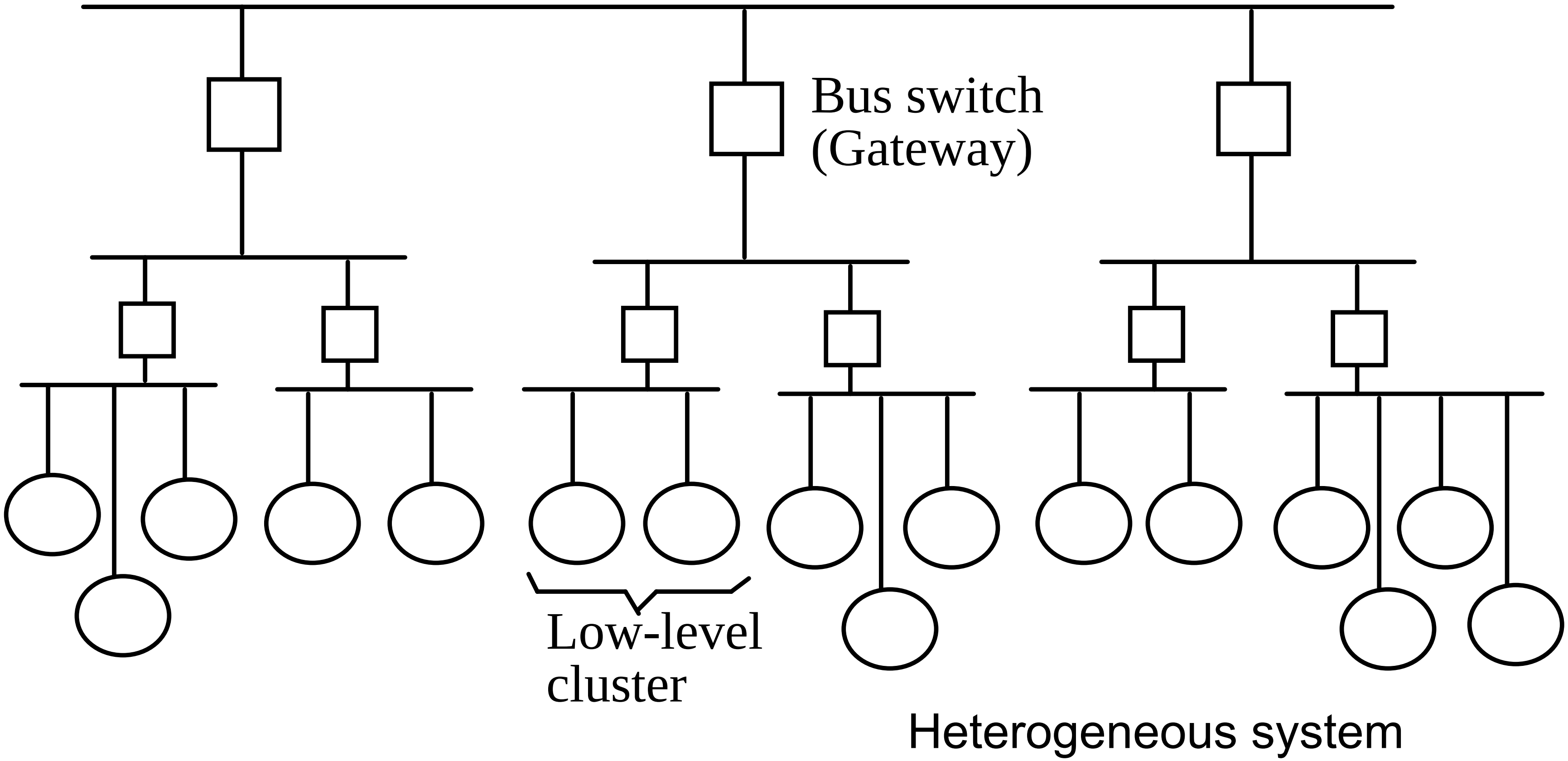



Fig. 4.9 Example of a hierarchical interconnection architecture.

# Removing the Processor-to-Memory Bottleneck

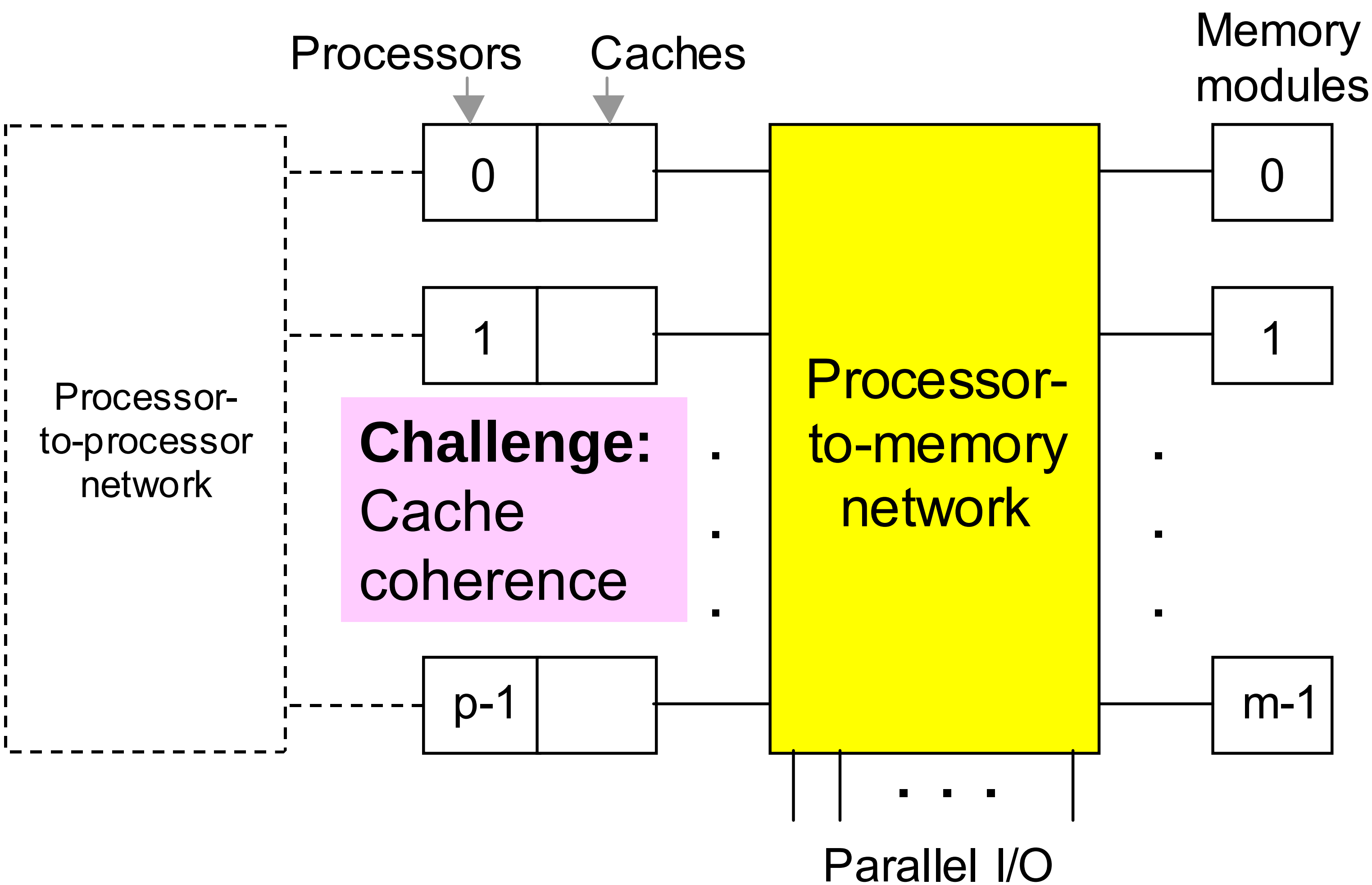



Fig. 4.4 A parallel processor with global memory and processor caches.

# Why Data Caching Works

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

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

**Cache parameters:**

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

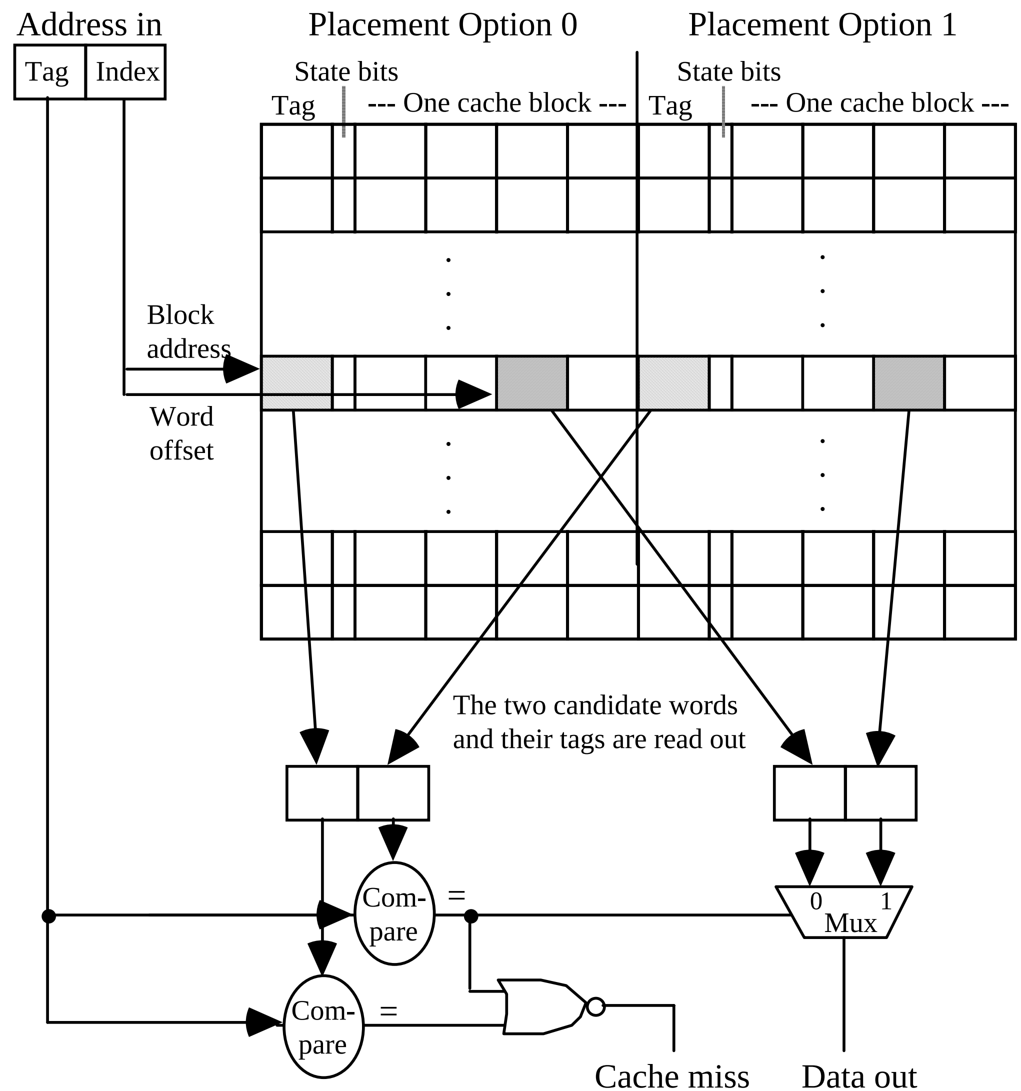



Fig. 18.1 Data storage and access in a two-way set-associative cache.

# Benefits of Caching Formulated as Amdahl's Law

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

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

$$S = C_{slow} / C_{eff}$$

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

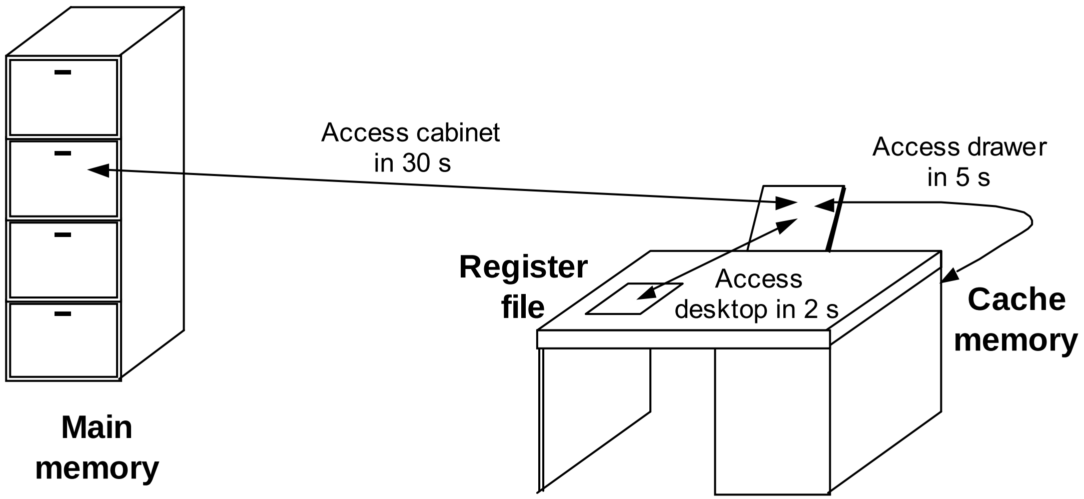



Fig. 18.3 of Parhami's Computer Architecture text (2005)

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

**Generalized form of Amdahl's speedup formula:**

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

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

# 18.2 Cache Coherence Protocols

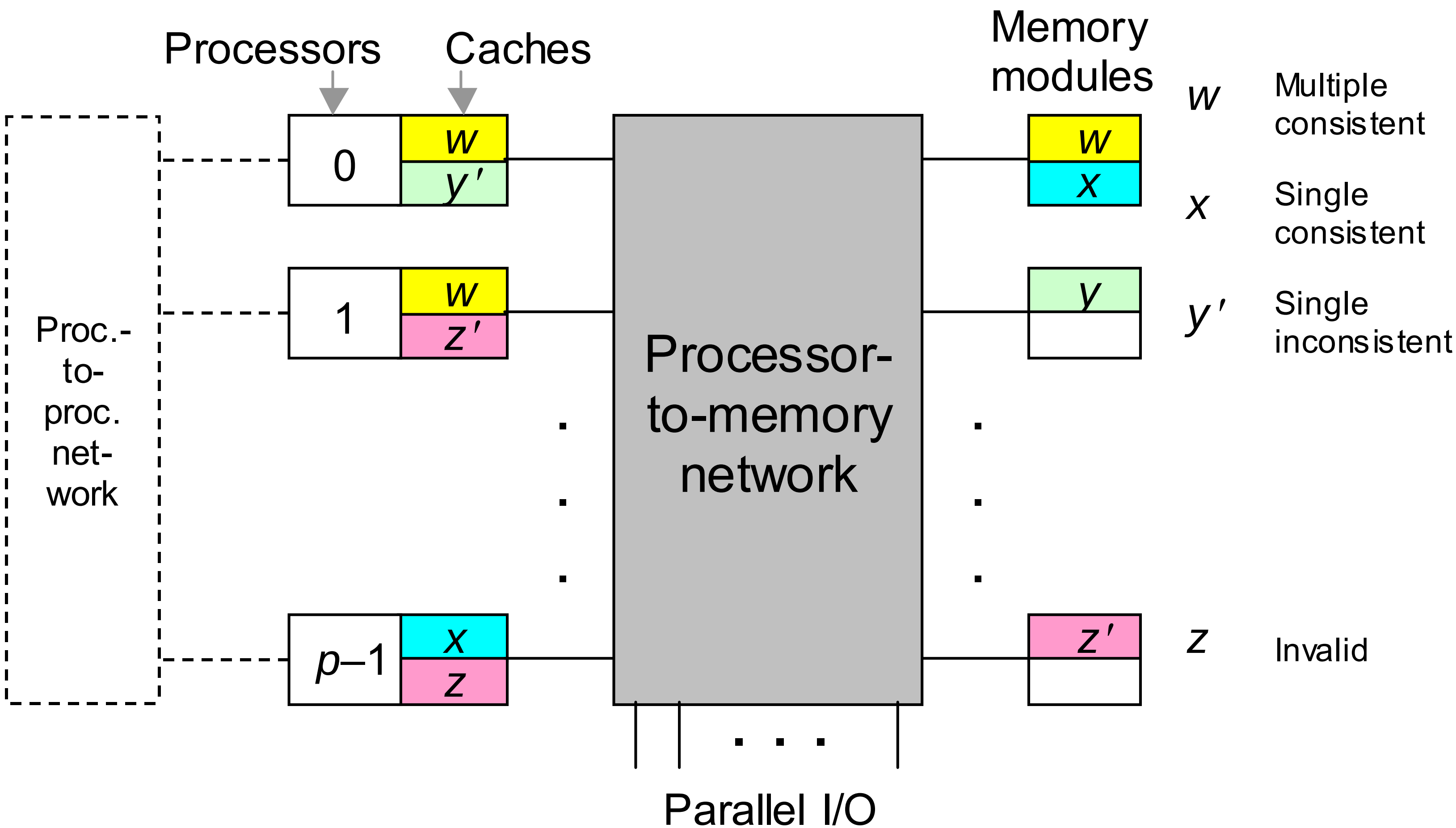



Fig. 18.2 Various types of cached data blocks in a parallel processor with global memory and processor caches.

# Example: A Bus-Based Snoopy Protocol

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

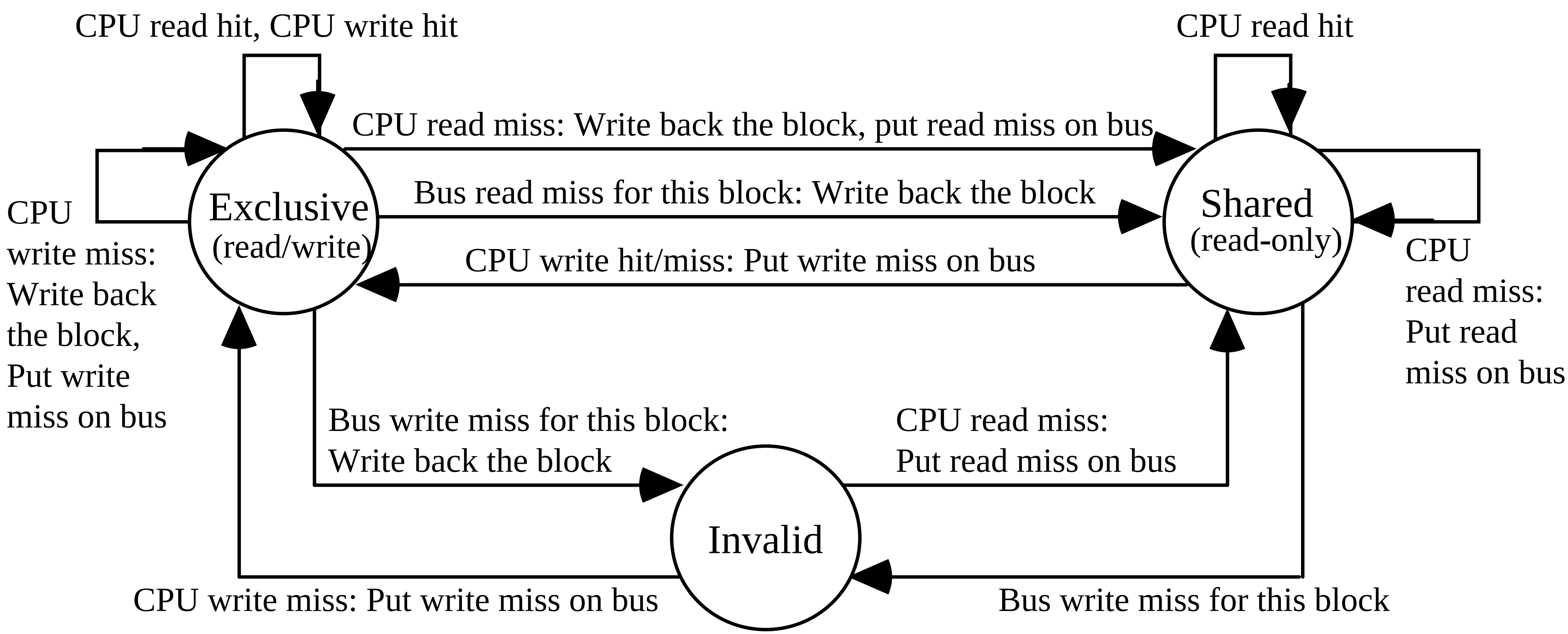



Fig. 18.3 Finite-state control mechanism for a bus-based snoopy cache coherence protocol.

# Implementing a Snoopy Protocol

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

Getting all the implementation timing and details right is nontrivial

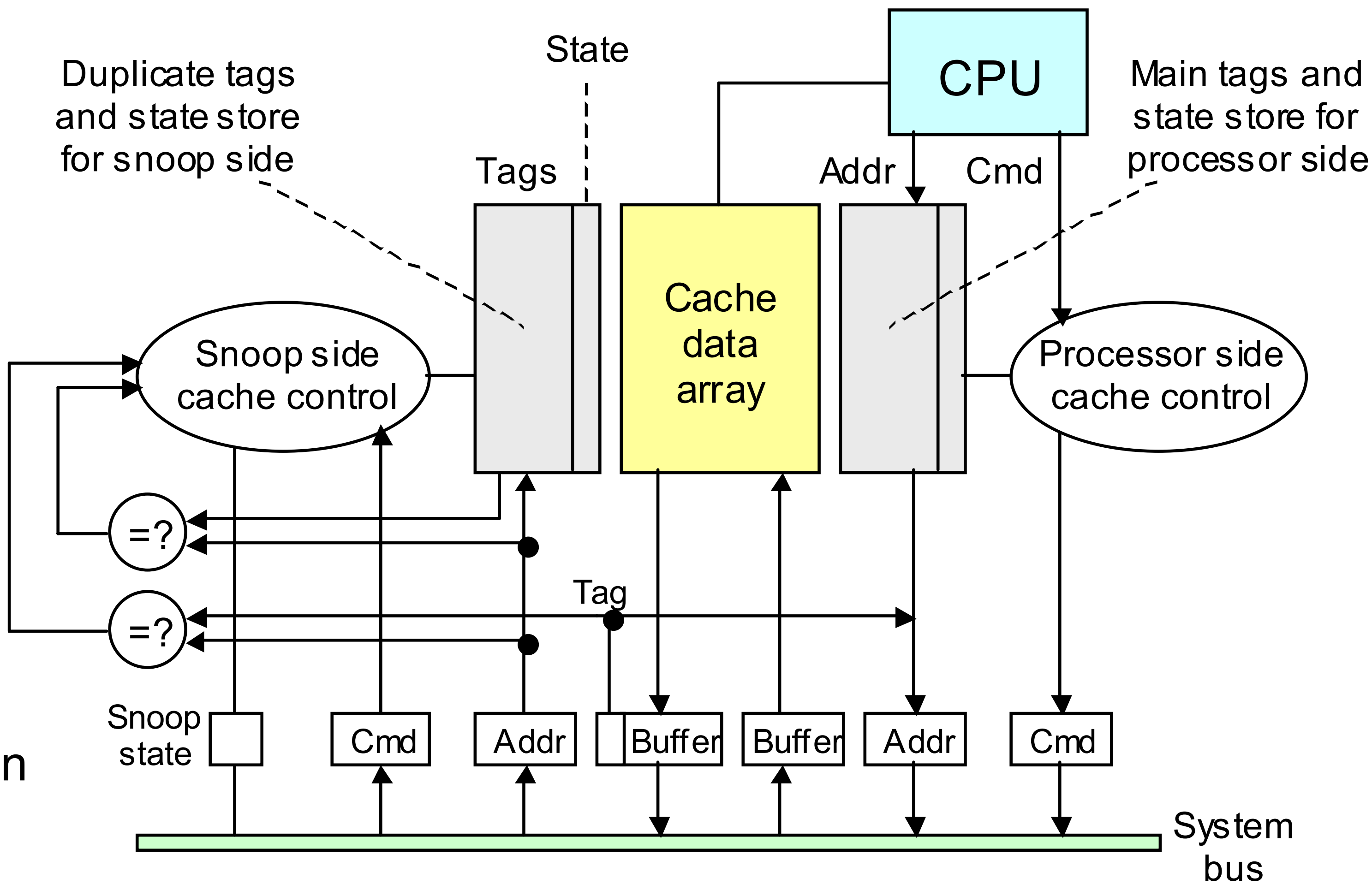



Fig. 27.7 of Parhami's Computer Architecture text.

# Distributed Shared Memory

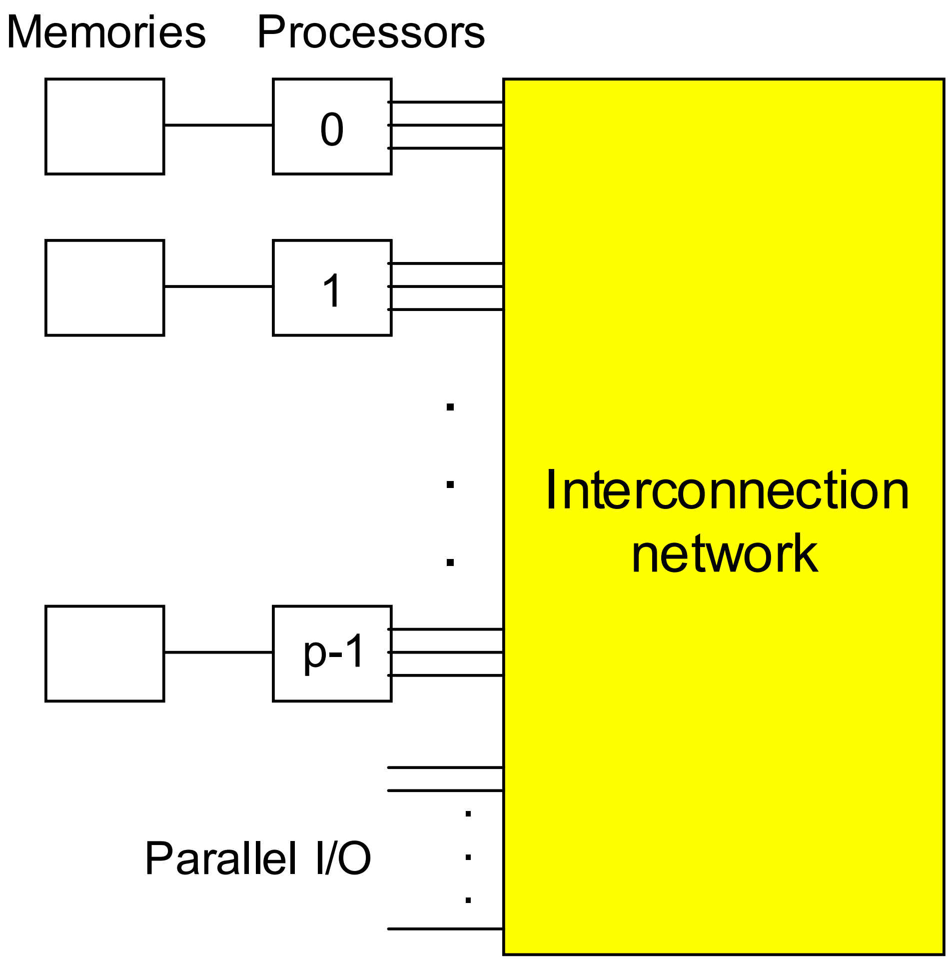



Fig. 4.5 A parallel processor with distributed memory.

**Some Terminology:**

NUMA
Nonuniform memory access
(distributed shared memory)

UMA
Uniform memory access
(global shared memory)

COMA
Cache-only memory arch

# Example: A Directory-Based Protocol

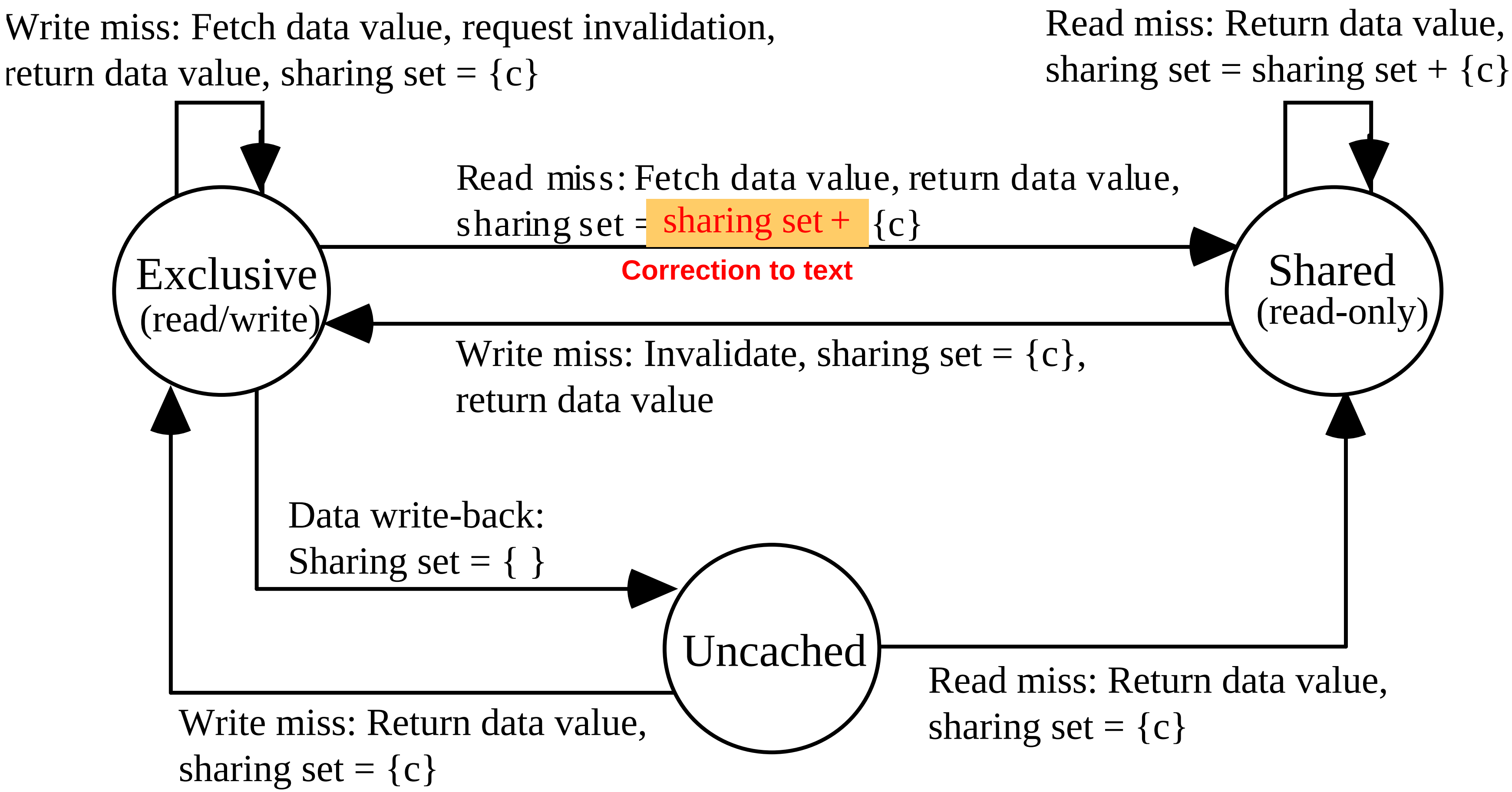


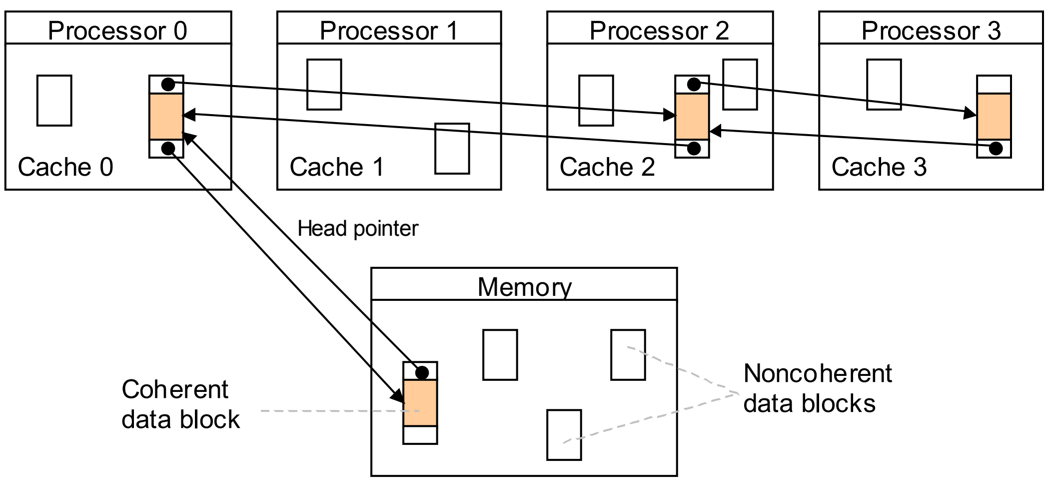

Fig. 18.4 States and transitions for a directory entry in a directory-based coherence protocol (*c* denotes the cache sending the message).

# Implementing a Directory-Based Protocol

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

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

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



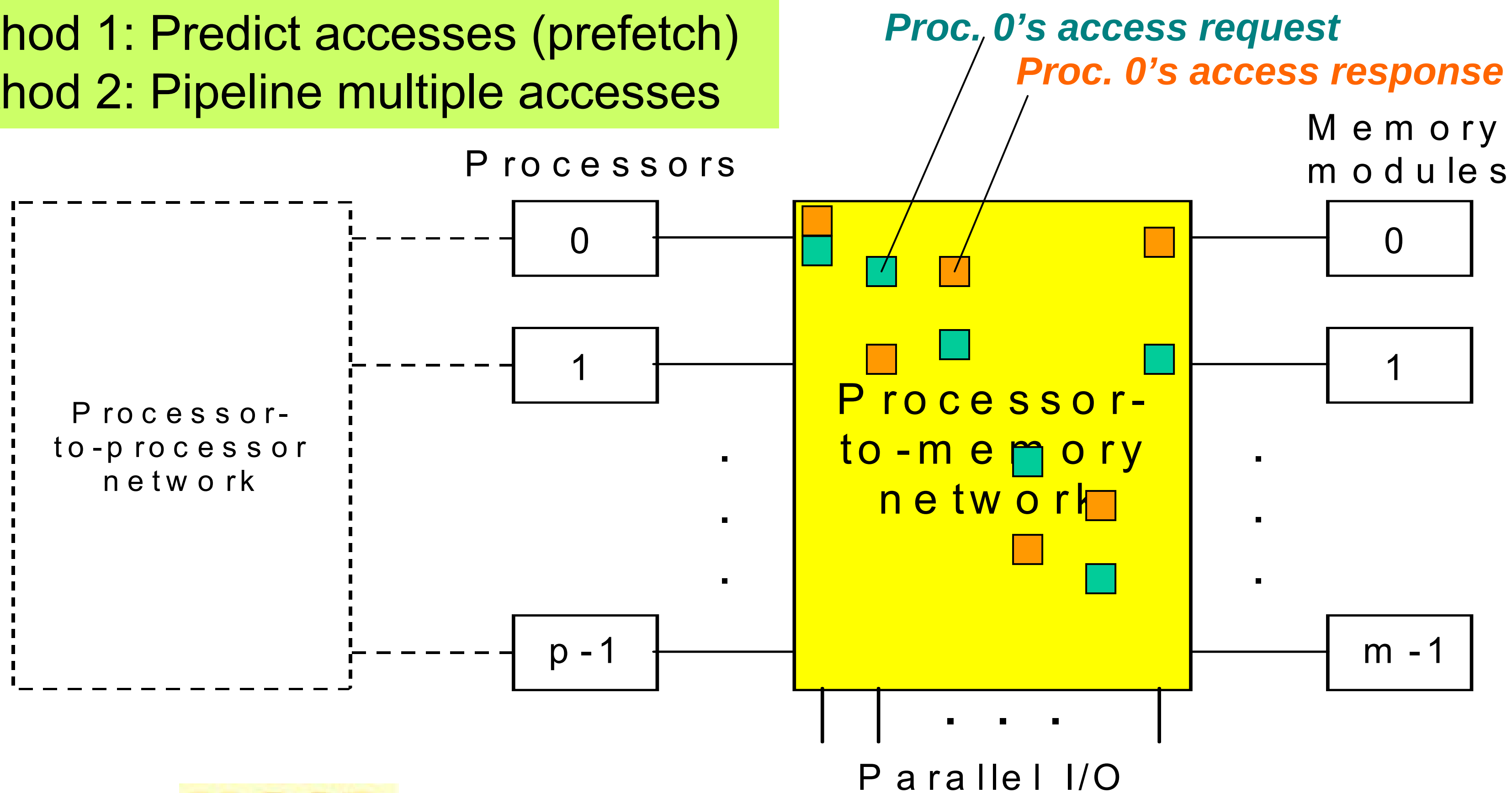

The sharing set can be maintained as a distributed doubly linked list (will discuss in Section 18.6 in connection with the SCI standard)

# Hiding the Memory Access Latency

By assumption, PRAM accesses memory locations right when they are needed, so processing must stall until data is fetched

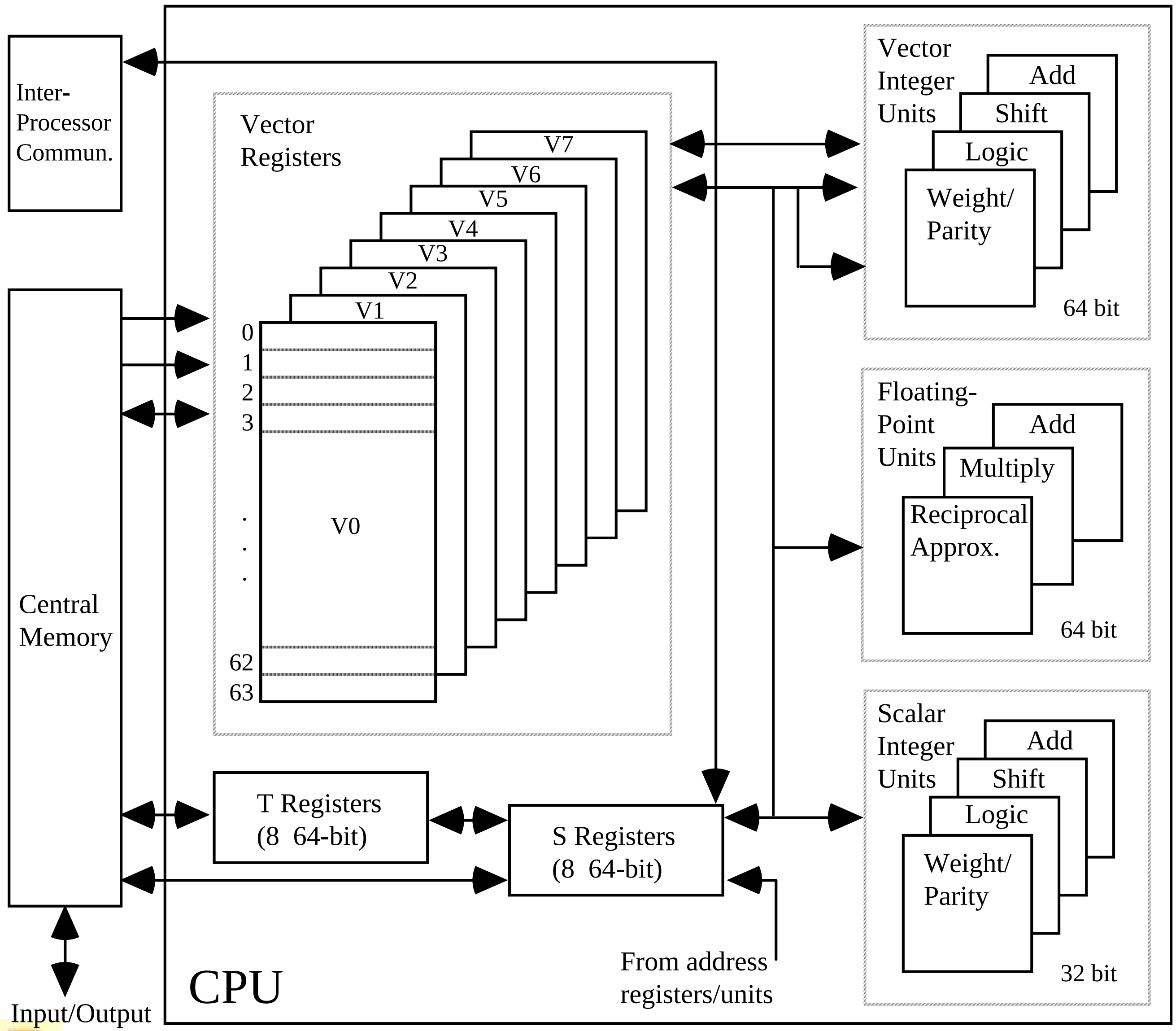
**Not a smart strategy:**
Memory access time = 100s times that of add time

Method 1: Predict accesses (prefetch)
Method 2: Pipeline multiple accesses



# 21.3 Vector-Parallel Cray Y-MP

Fig. 21.5 Key elements of the Cray Y-MP processor. Address registers, address function units, instruction buffers, and control not shown.



# Cray Y-MP's Interconnection Network

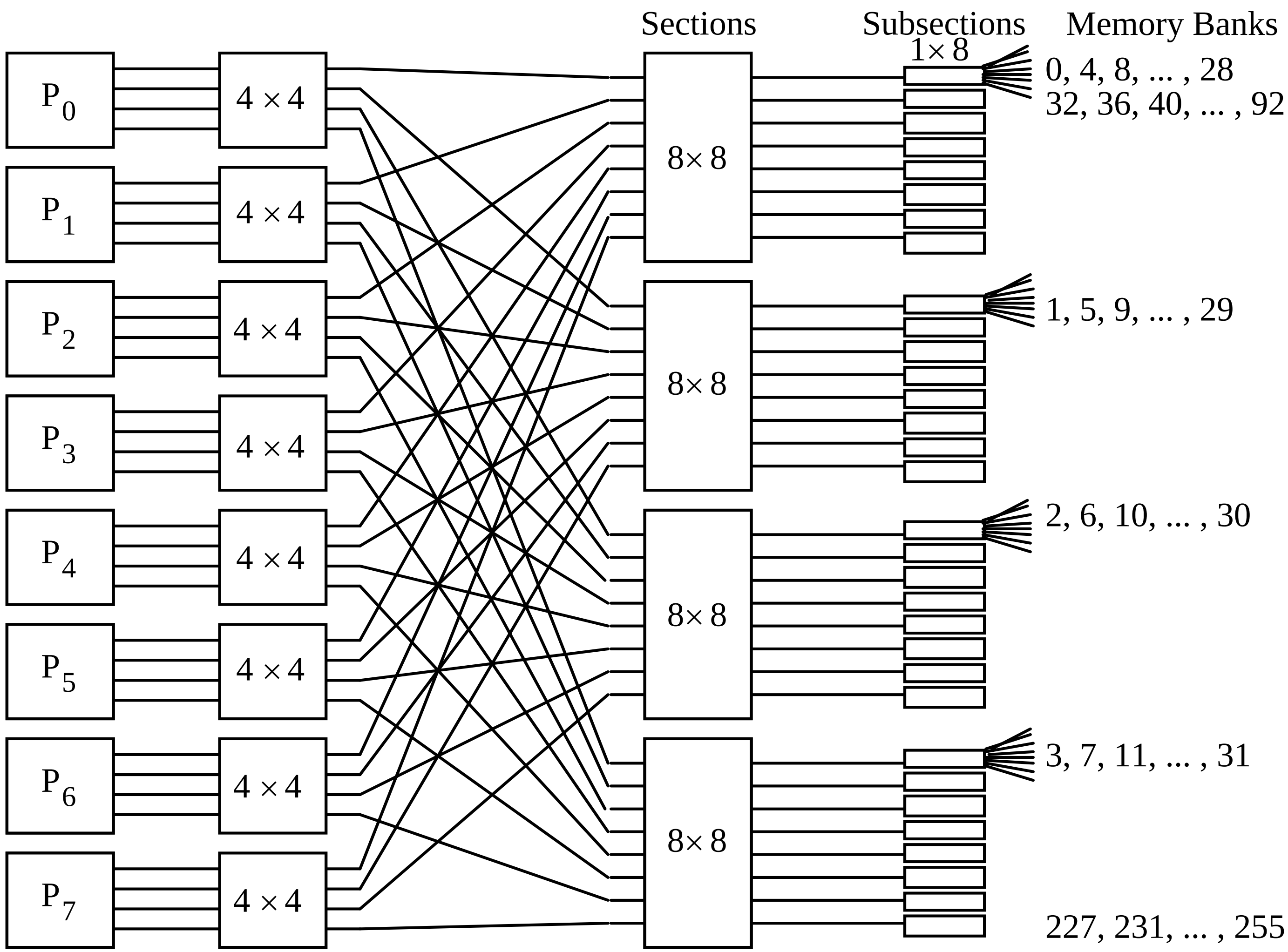



Fig. 21.6 The processor-to-memory interconnection network of Cray Y-MP.

# Butterfly Processor-to-Memory Network

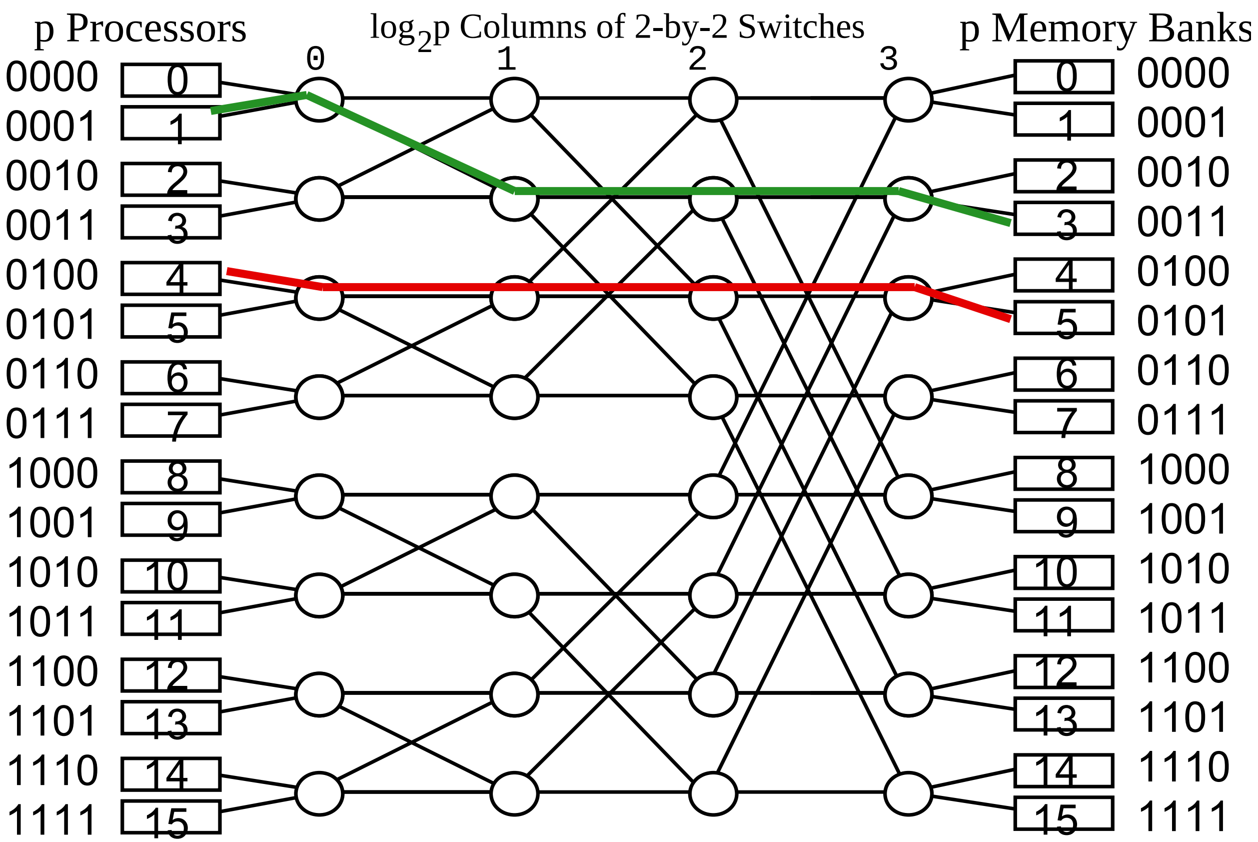



Fig. 6.9 Example of a multistage memory access network.

Not a full permutation network (e.g., processor 0 cannot be connected to memory bank 2 alongside the two connections shown)

Is self-routing: i.e., the bank address determines the route

A request going to memory bank 3 (0 0 1 1) is routed:

lower upper upper

# Butterfly as Multistage Interconnection Network

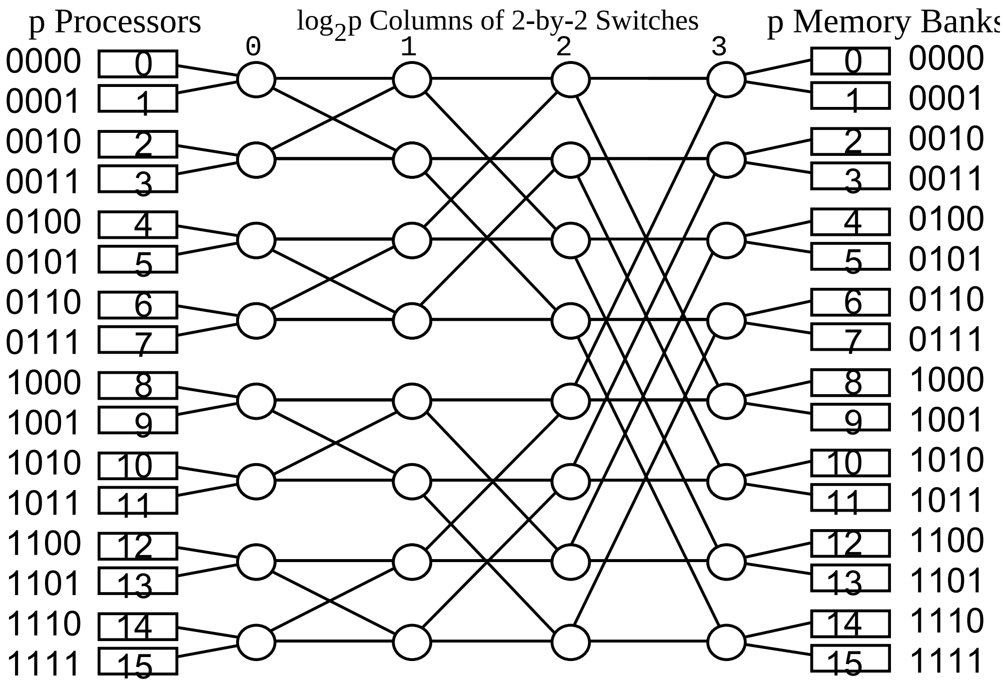



Fig. 6.9 Example of a multistage memory access network

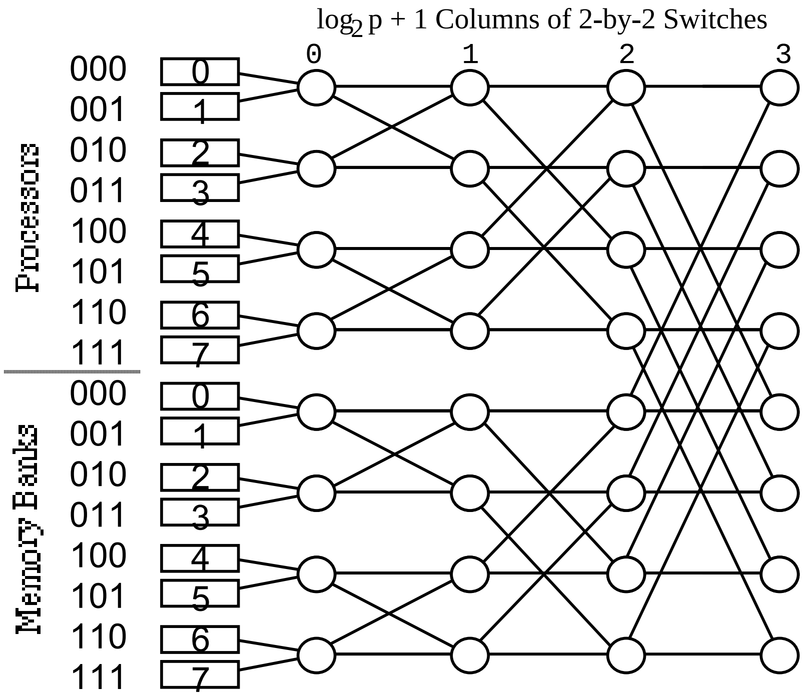



Fig. 15.8 Butterfly network used to connect modules that are on the same side

Generalization of the butterfly network

- High-radix or $m$-ary butterfly, built of $m \times m$ switches
- Has $m^q$ rows and $q + 1$ columns ($q$ if wrapped)

# Beneš Network

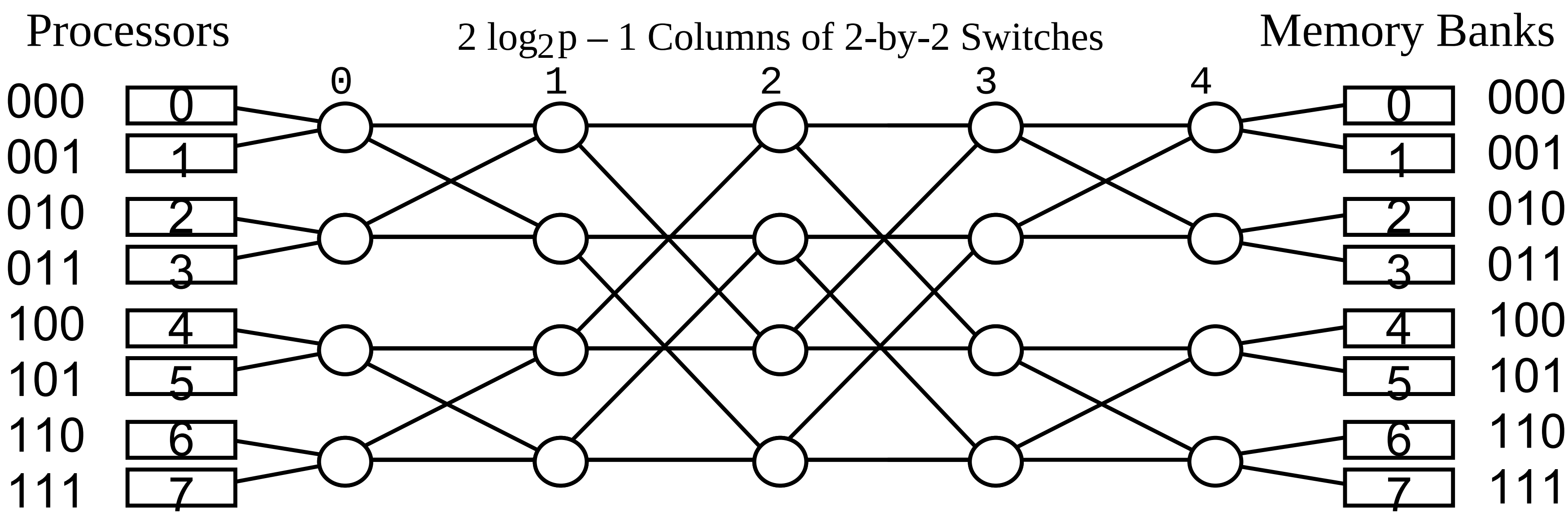



Fig. 15.9 Beneš network formed from two back-to-back butterflies.

A $2^q$-row Beneš network:

Can route any $2^q \times 2^q$ permutation

It is “rearrangeable”

# Routing Paths in a Beneš Network

To which memory modules can we connect proc 4 without rearranging the other paths?

What about proc 6?

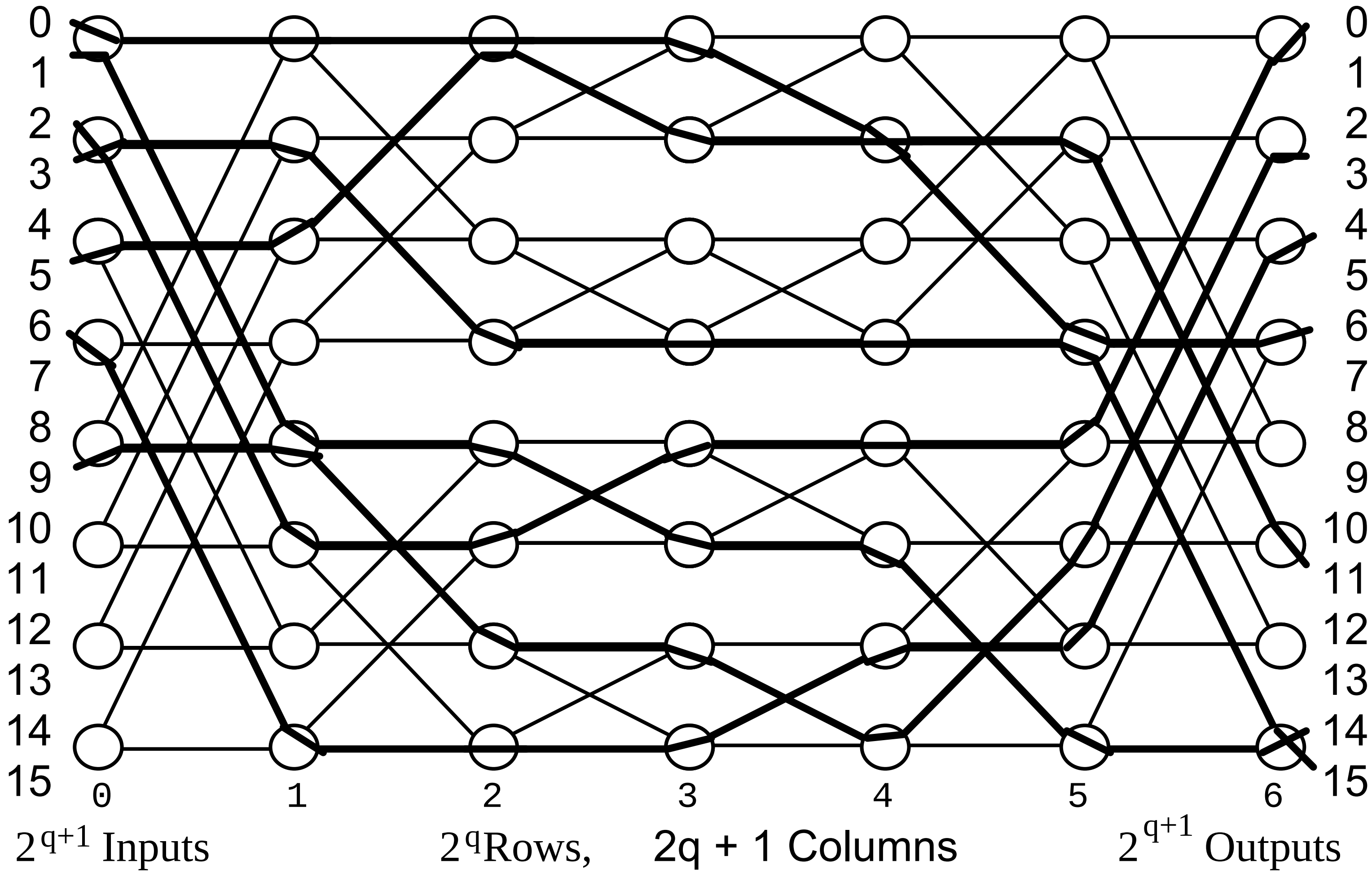



Fig. 15.10 Another example of a Beneš network.

# 16.6 Multistage Interconnection Networks

Numerous indirect or multistage interconnection networks (MINs) have been proposed for, or used in, parallel computers

They differ in topological, performance, robustness, and realizability attributes

We have already seen the butterfly, hierarchical bus, beneš, and ADM networks

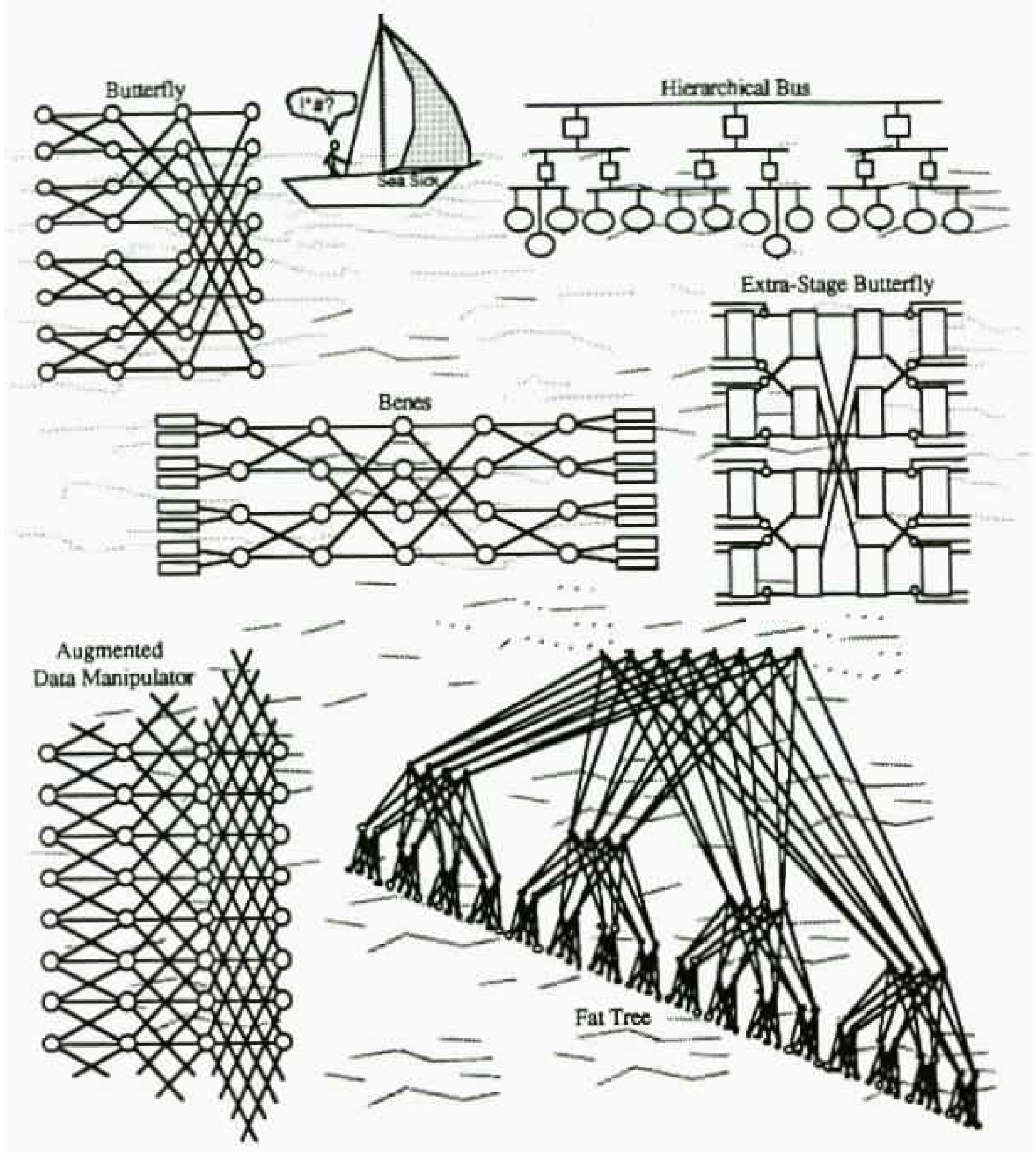



Fig. 4.8 (modified) The sea of indirect interconnection networks.

# Self-Routing Permutation Networks

Do there exist self-routing permutation networks? (The butterfly network is self-routing, but it is not a permutation network)

Permutation routing through a MIN is the same problem as sorting

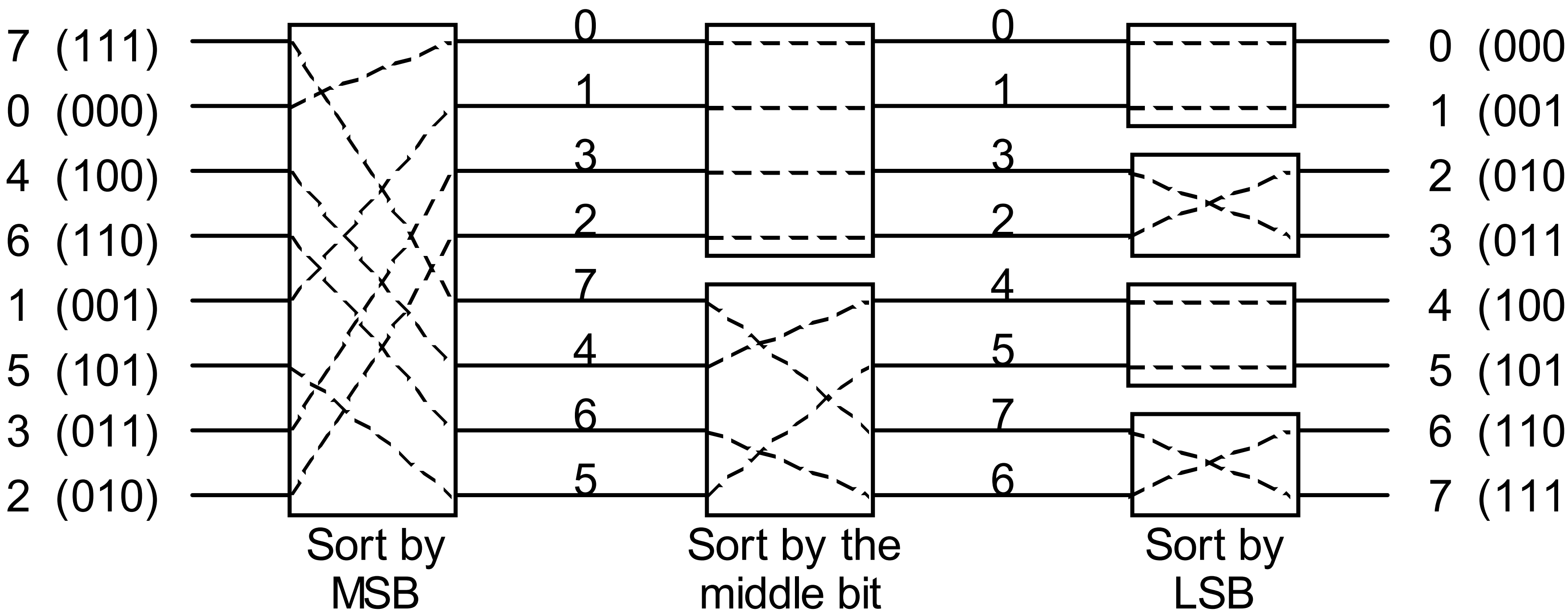



Fig. 16.14 Example of sorting on a binary radix sort network.

# Partial List of Important MINs

**Augmented data manipulator (ADM)**: aka unfolded PM2I (Fig. 15.12)

**Banyan**: Any MIN with a unique path between any input and any output (e.g. butterfly)

**Baseline**: Butterfly network with nodes labeled differently

**Beneš**: Back-to-back butterfly networks, sharing one column (Figs. 15.9-10)

**Bidelta**: A MIN that is a delta network in either direction

**Butterfly**: aka unfolded hypercube (Figs. 6.9, 15.4-5)

**Data manipulator**: Same as ADM, but with switches in a column restricted to same state

**Delta**: Any MIN for which the outputs of each switch have distinct labels (say 0 and 1 for $2 \times 2$ switches) and path label, composed of concatenating switch output labels leading from an input to an output depends only on the output

**Flip**: Reverse of the omega network (inputs $\times$ outputs)

**Indirect cube**: Same as butterfly or omega

**Omega**: Multi-stage shuffle-exchange network; isomorphic to butterfly (Fig. 15.19)

**Permutation**: Any MIN that can realize all permutations

**Rearrangeable**: Same as permutation network

**Reverse baseline**: Baseline network, with the roles of inputs and outputs interchanged

# Conflict-Free Memory Access

Try to store the data such that parallel accesses are to different banks

For many data structures, a compiler may perform the memory mapping

Column 2

| | | | | | | |
|---|---|---|---|---|---|---|
| | 0,0 | 0,1 | 0,2 | 0,3 | 0,4 | 0,5 |
| Row 1 | 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 |
| Module | 0 | 1 | 2 | 3 | 4 | 5 |

Each matrix column is stored in a different memory module (bank)

Accessing a column leads to conflicts

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

# Skewed Storage Format

Column 2

Row 1

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 0,0 | 0,1 | 0,2 | 0,3 | 0,4 | 0,5 |
| Row 1 | 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 |

Module

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

# A Unified Theory of Conflict-Free Access

Vector indices

| | | | | | |
|---|---|---|---|---|---|
| 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 |

$A_{ij}$ is viewed as vector element $i + jm$

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

A $q$D array can be viewed as a vector, with "row" / "column" accesses associated with constant strides

| | | |
|---|---|---|
| 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 Schemes

Vector indices

| | | | | | |
|---|---|---|---|---|---|
| 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 |

$A_{ij}$ is viewed as vector element $i + jm$

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

Place vector element $i$ in memory bank $a + bi \bmod B$ (word address within bank is irrelevant to conflict-free access; also, $a$ can be set to 0)

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

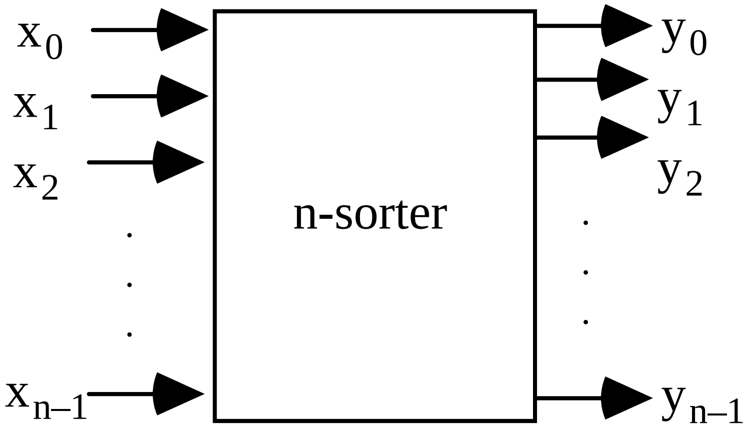
A prime value for $B$ ensures this condition, but is not very practical.

# 7 Sorting and Selection Networks

Become familiar with the circuit model of parallel processing:
- Go from algorithm to architecture, not vice versa
- Use a familiar problem to study various trade-offs

| Topics in This Chapter |
|---|
| 7.1 What is a Sorting Network? |
| 7.2 Figures of Merit for Sorting Networks |
| 7.3 Design of Sorting Networks |
| 7.4 Batcher Sorting Networks |
| 7.5 Other Classes of Sorting Networks |
| 7.6 Selection Networks |

# 7.1 What is a Sorting Network?




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.

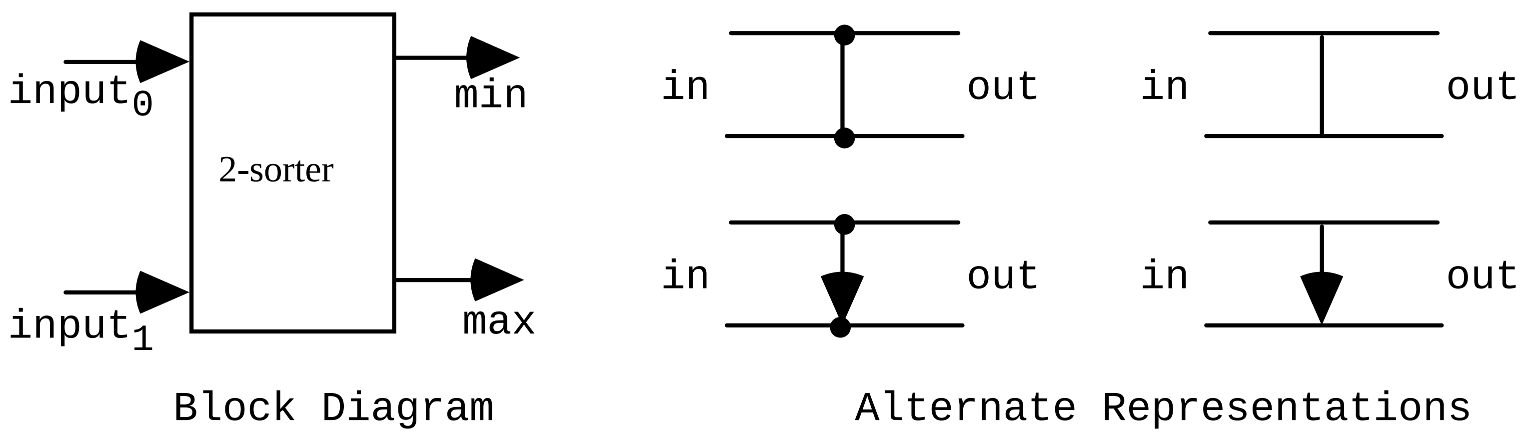
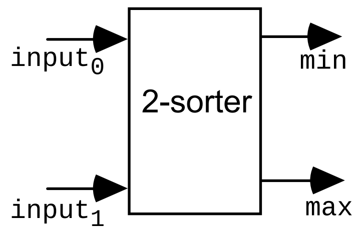



Fig. 7.2 Block diagram and four different schematic representations for a 2-sorter.

# Building Blocks for Sorting Networks




Implementation with bit-parallel inputs

Implementation with bit-serial inputs

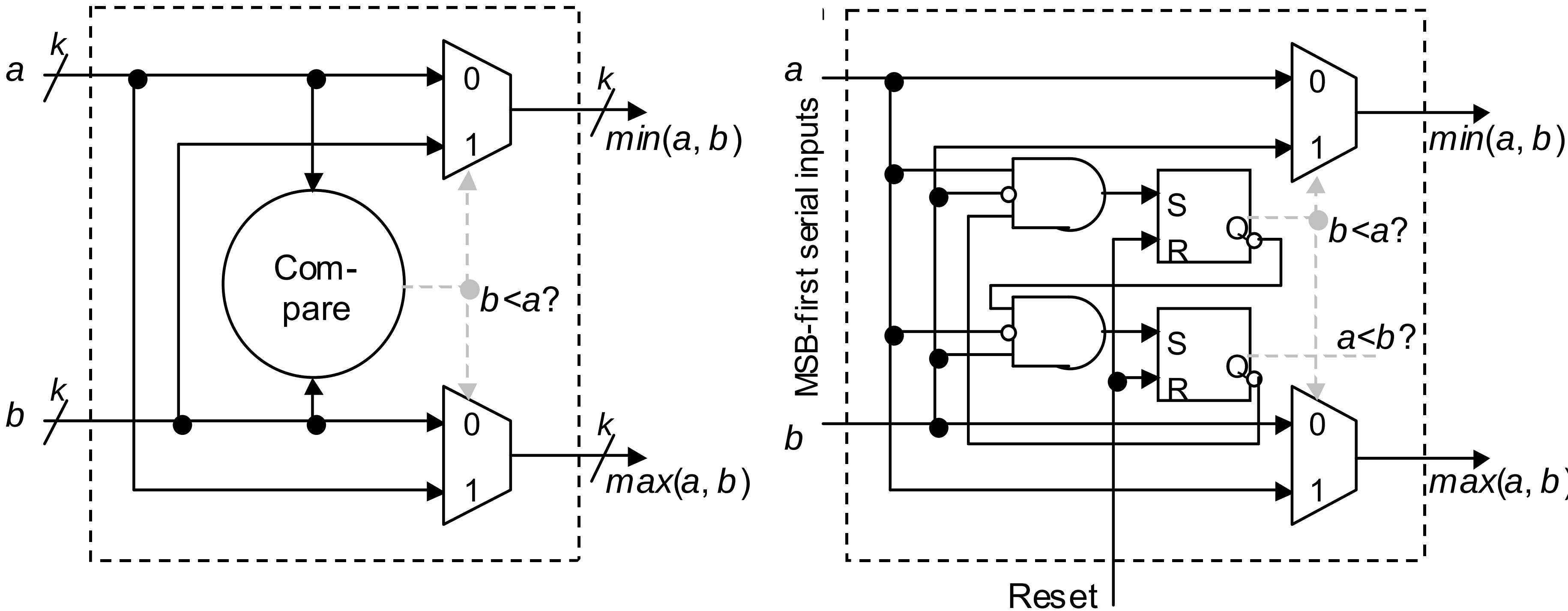
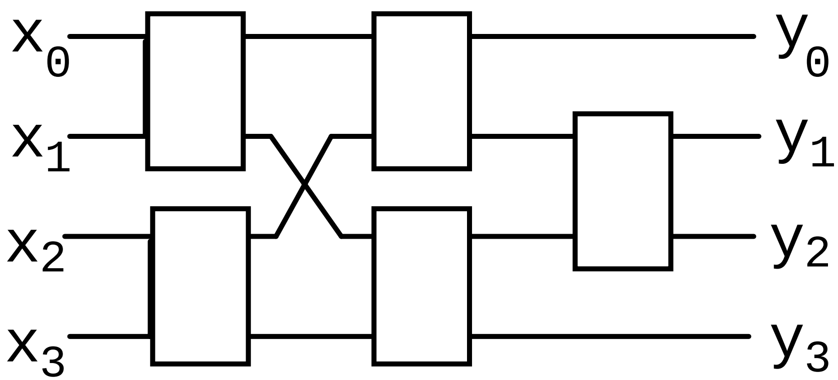



Fig. 7.3 Parallel and bit-serial hardware realizations of a 2-sorter.

# Proving a Sorting Network Correct



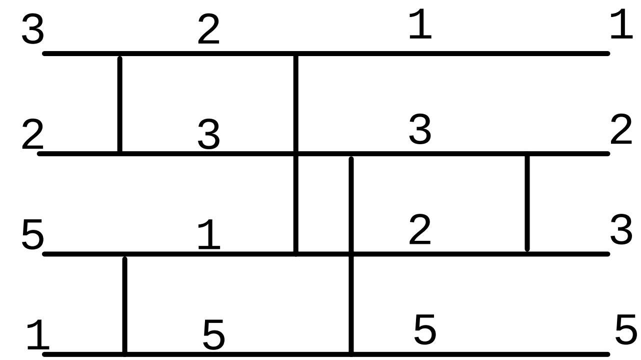



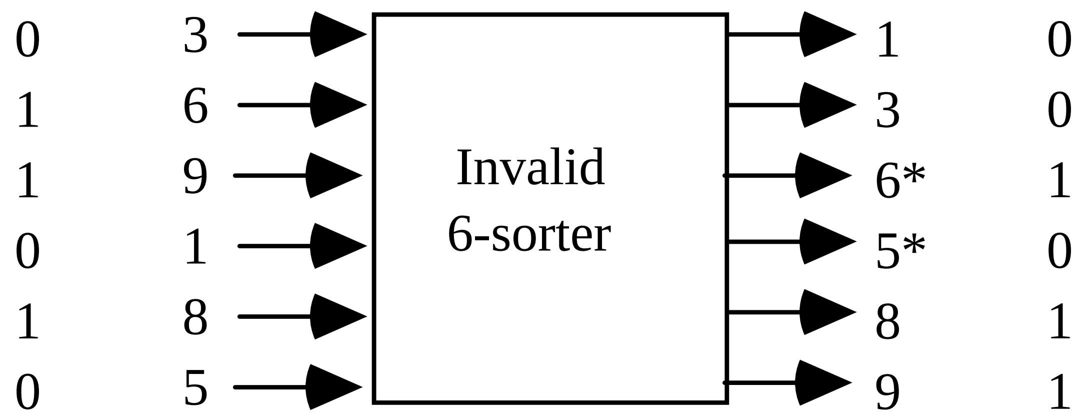

Fig. 7.4 Block diagram and schematic representation of a 4-sorter.

Method 1: Exhaustive test – Try all *n*! possible input orders

Method 2: Ad hoc proof – for the example above, note that $y_0$ is smallest, $y_3$ is largest, and the last comparator sorts the other two outputs

Method 3: Use the zero-one principle – A comparison-based sorting algorithm is correct iff it correctly sorts all 0-1 sequences ($2^n$ tests)

# Elaboration on the Zero-One Principle




Deriving a 0-1 sequence that is not correctly sorted, given an arbitrary sequence that is not correctly sorted.

Let outputs $y_i$ and $y_{i+1}$ be out of order, that is $y_i > y_{i+1}$

Replace inputs that are strictly less than $y_i$ with 0s and all others with 1s

The resulting 0-1 sequence will not be correctly sorted either

# 7.2 Figures of Merit for Sorting Networks

**Cost:** Number of comparators

In the following example, we have 5 comparators

**Delay:** Number of levels

The following 4-sorter has 3 comparator levels on its critical path

**Cost × Delay**

The cost-delay product for this example is 15

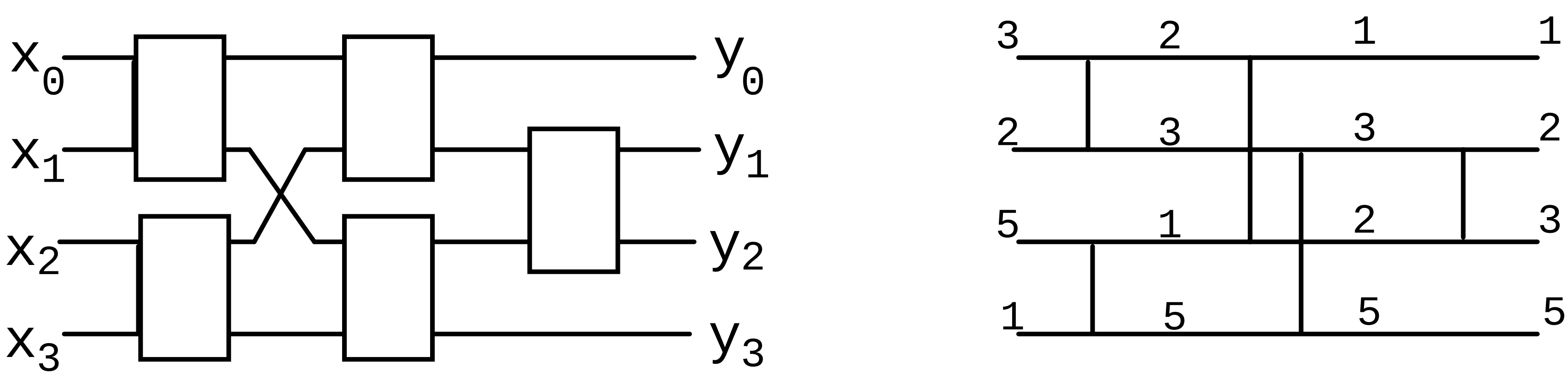
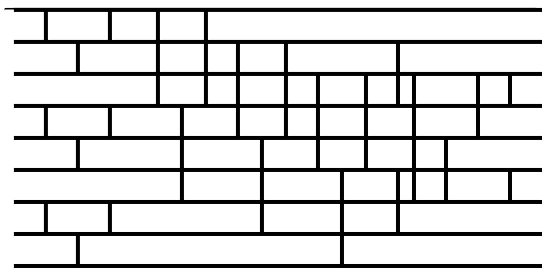
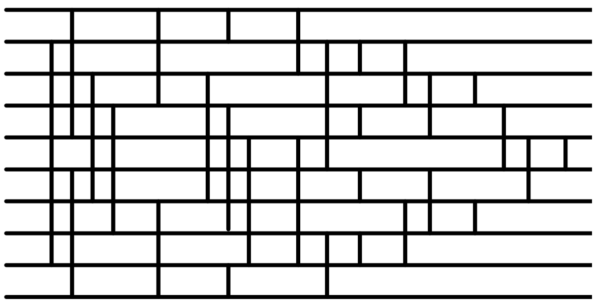



Fig. 7.4 Block diagram and schematic representation of a 4-sorter.

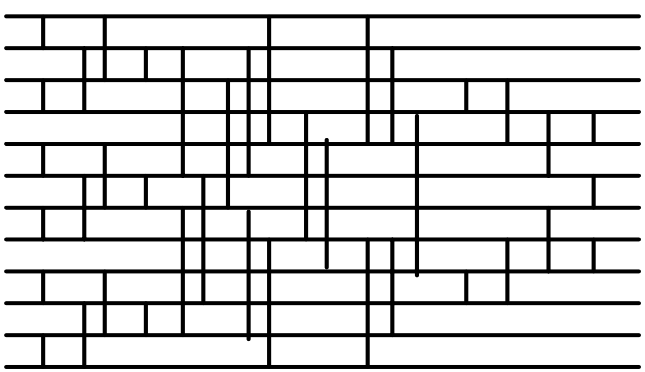
# Cost as a Figure of Merit



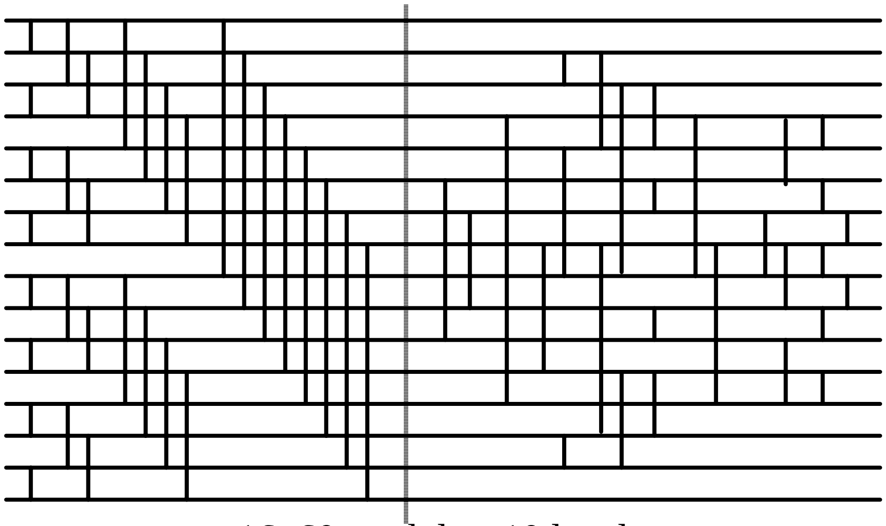
n = 9, 25 modules, 9 levels



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.

# Delay as a Figure of Merit

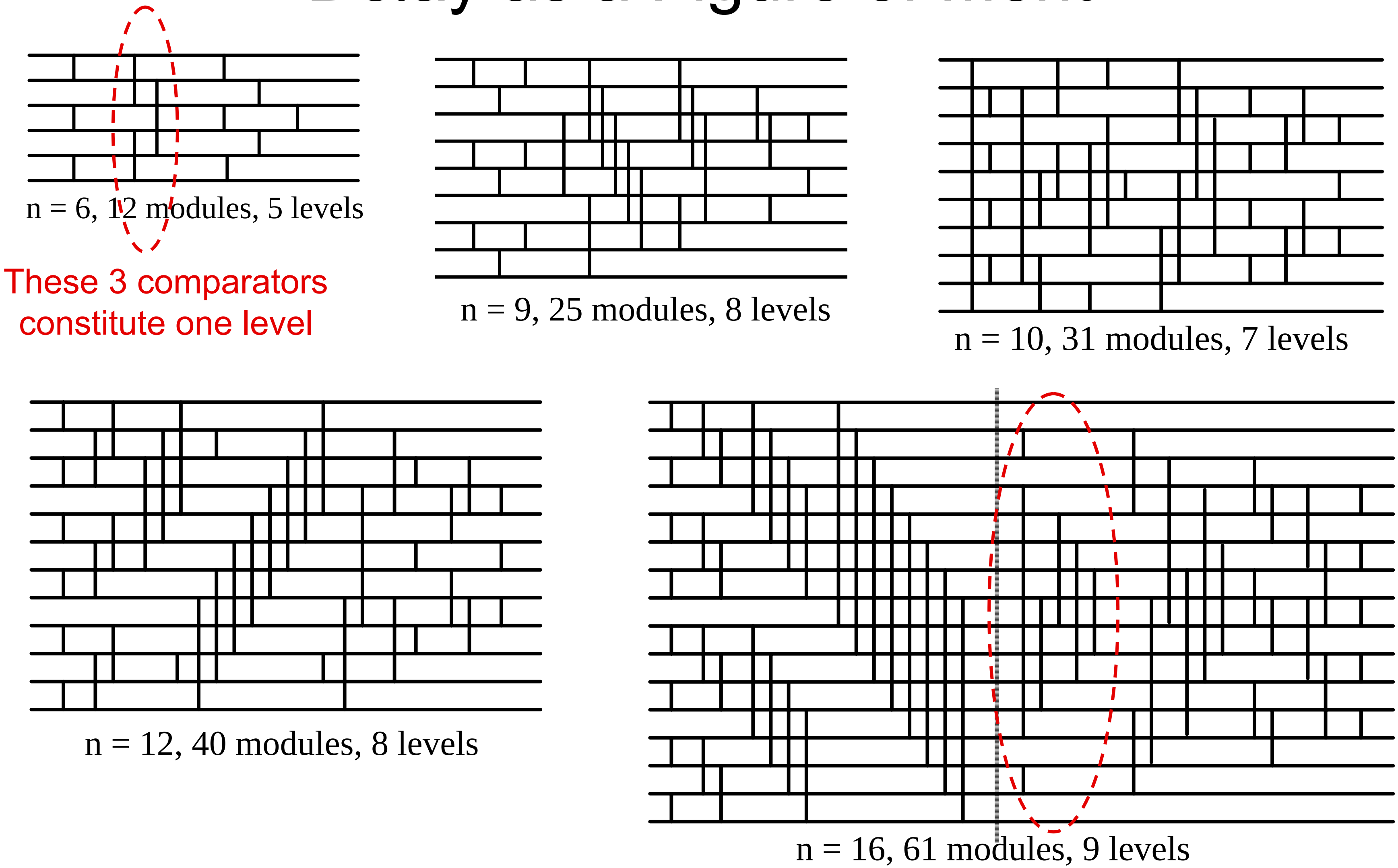
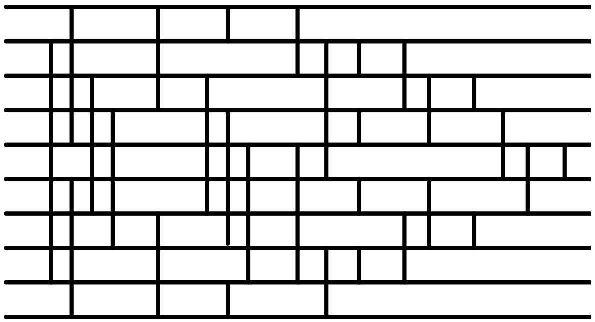
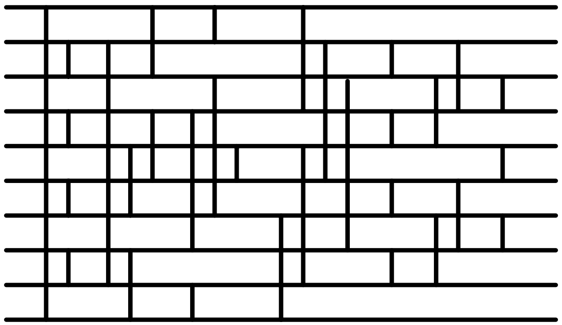



Fig. 7.6 Some fast sorting networks.

# Cost-Delay Product as a Figure of Merit



n = 10, 29 modules, 9 levels



n = 10, 31 modules, 7 levels

Low-cost 10-sorter from Fig. 7.5

Fast 10-sorter from Fig. 7.6

Cost $\times$ Delay = 29 $\times$ 9 = 261

Cost $\times$ Delay = 31 $\times$ 7 = 217

The most cost-effective *n*-sorter may be neither the fastest design, nor the lowest-cost design

# 7.3 Design of Sorting Networks

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

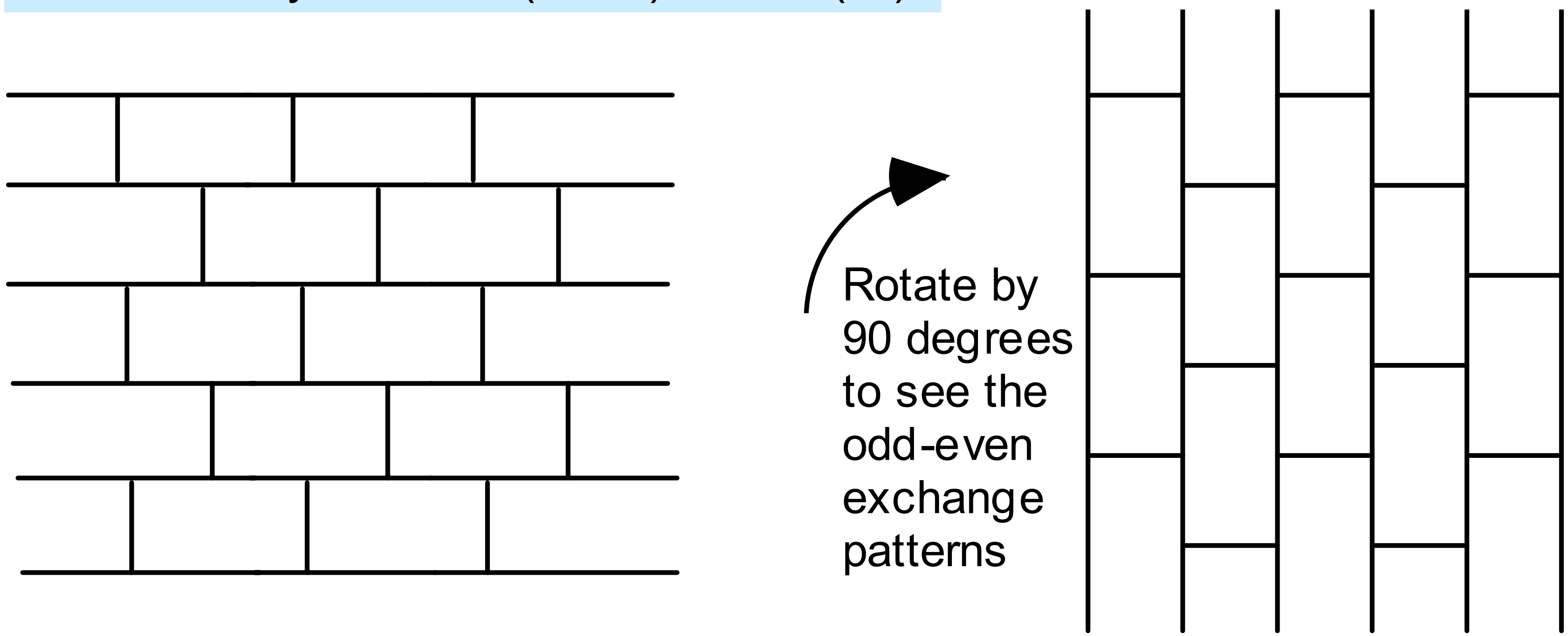
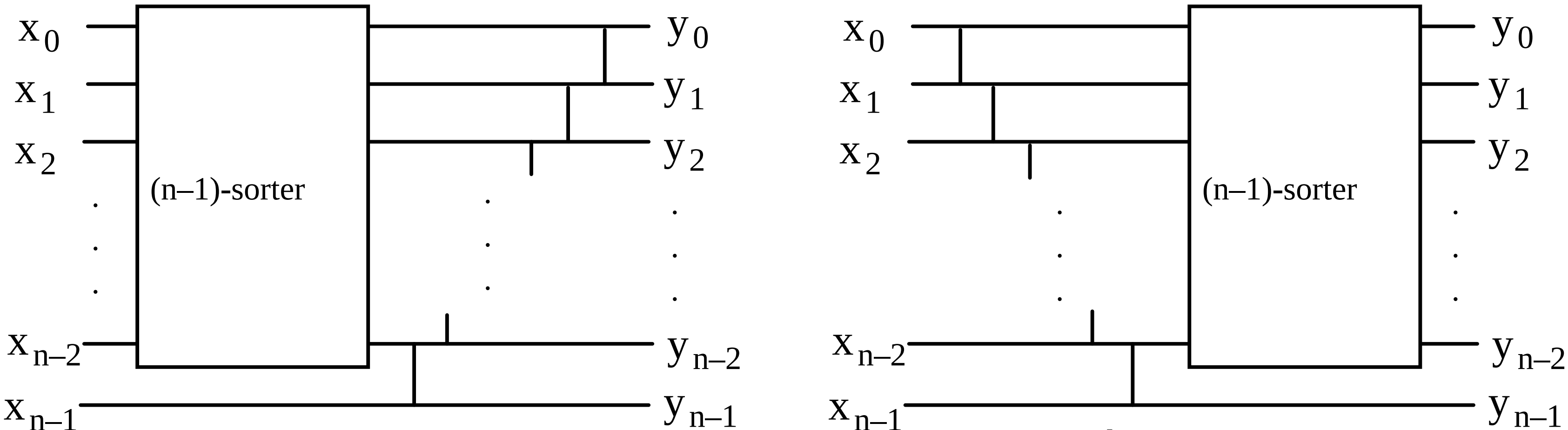



Fig. 7.7 Brick-wall 6-sorter based on odd–even transposition.

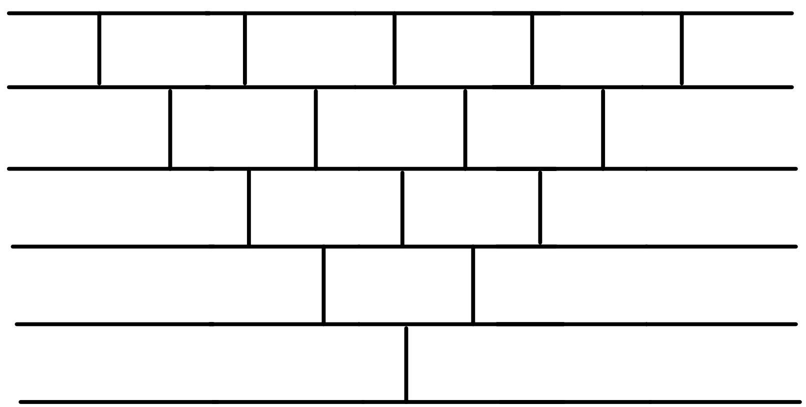
# Insertion Sort and Selection Sort




Insertion sort

Selection sort

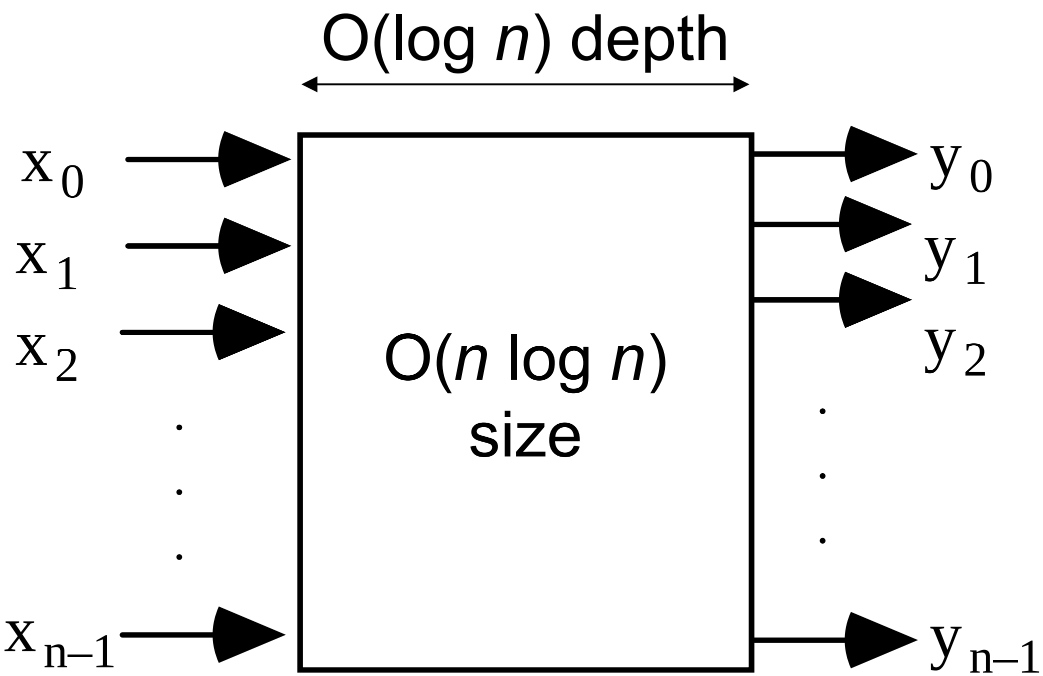
Parallel insertion sort = Parallel selection sort = Parallel bubble sort!



$C(n) = n(n - 1)/2$
$D(n) = 2n - 3$
Cost $\times$ Delay
$= \Theta(n^3)$

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

# Theoretically Optimal Sorting Networks




AKS sorting network
(Ajtai, Komlos, Szemeredi: 1983)

Note that even for these optimal networks, delay-cost product is suboptimal; but this is the best we can do

Existing sorting networks have $O(\log^2 n)$ latency and $O(n \log^2 n)$ cost

Given that $\log_2 n$ is only 20 for $n$ = 1 000 000, the latter are more practical

Unfortunately, AKS networks are not practical owing to large (4-digit) constant factors involved; improvements since 1983 not enough

# 7.4 Batcher Sorting Networks

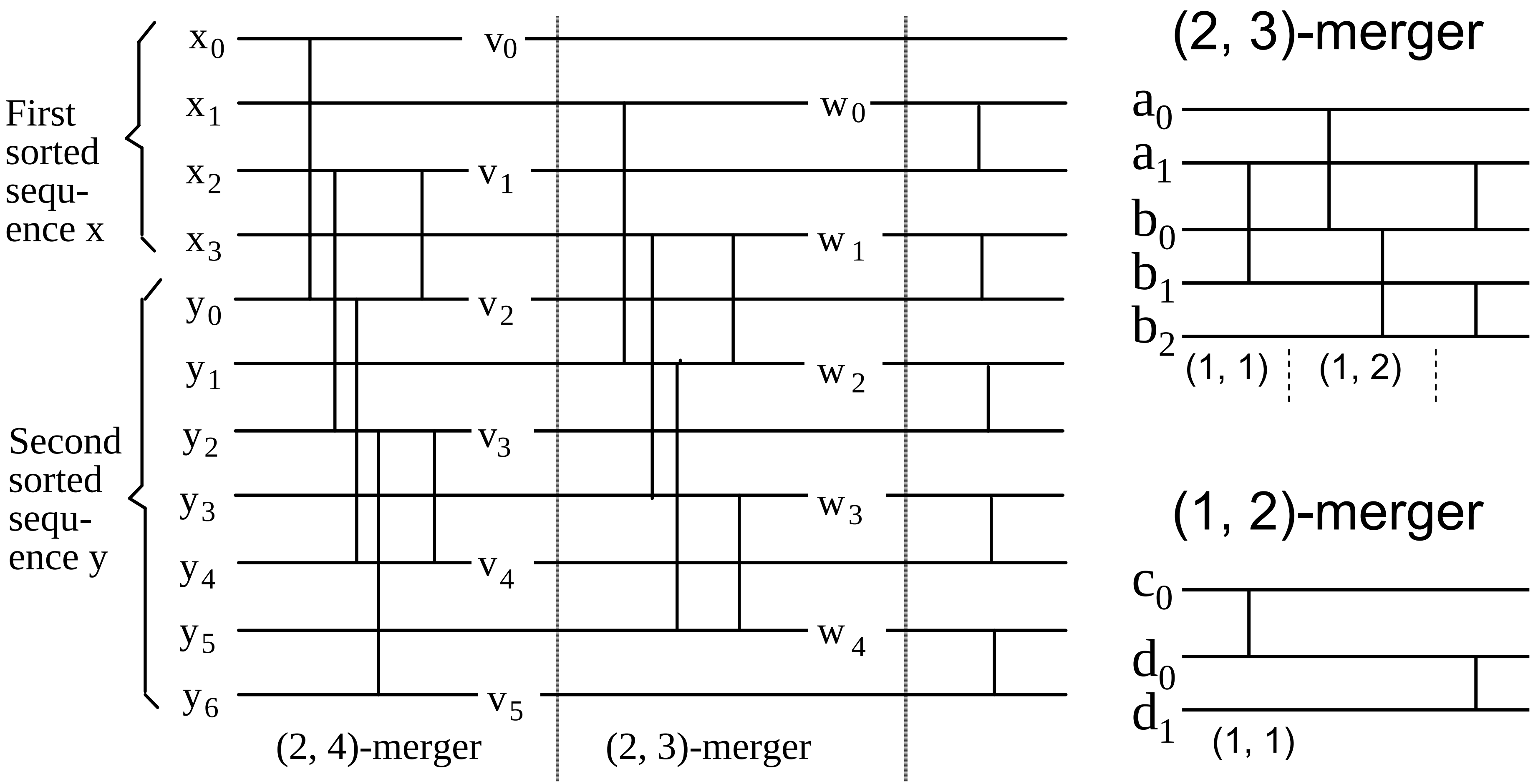
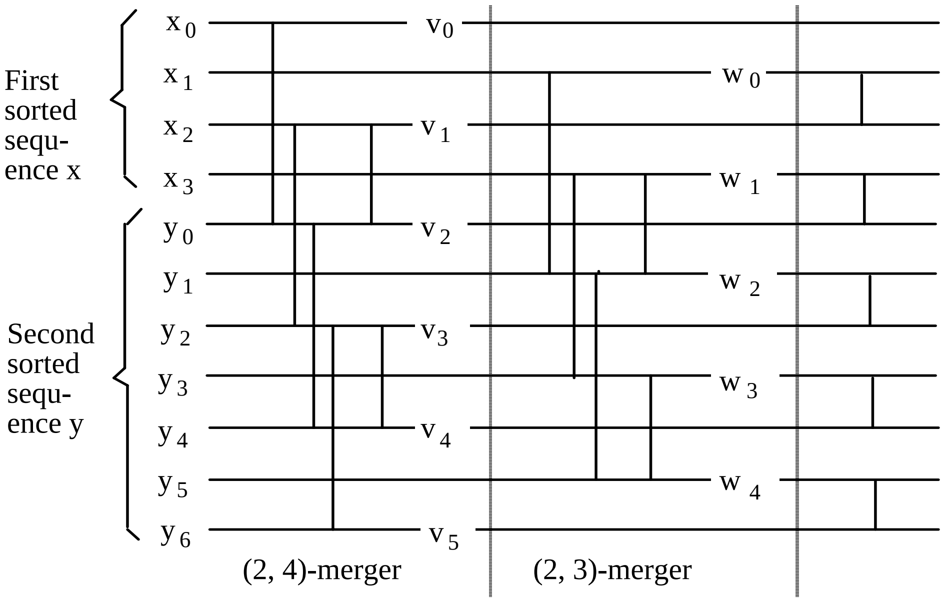



Fig. 7.9 Batcher's even–odd merging network for 4 + 7 inputs.

# Proof of Batcher's Even-Odd Merge




Use the zero-one principle

Assume: $x$ has $k$ 0s
$y$ has $k'$ 0s

$v$ has $k_{even} = \lceil k/2 \rceil + \lceil k'/2 \rceil$ 0s

$w$ has $k_{odd} = \lfloor k/2 \rfloor + \lfloor k'/2 \rfloor$ 0s

Case a: $k_{even} = k_{odd}$
$v$ 0 0 0 0 0 0 1 1 1 1 1 1
$w$ 0 0 0 0 0 0 1 1 1 1 1

Case b: $k_{even} = k_{odd}+1$
$v$ 0 0 0 0 0 0 0 1 1 1 1 1
$w$ 0 0 0 0 0 0 1 1 1 1 1

Case c: $k_{even} = k_{odd}+2$
$v$ 0 0 0 0 0 0 0 0 1 1 1 1
$w$ 0 0 0 0 0 0 1 1 1 1 1
Out of order

# Batcher's Even-Odd Merge Sorting

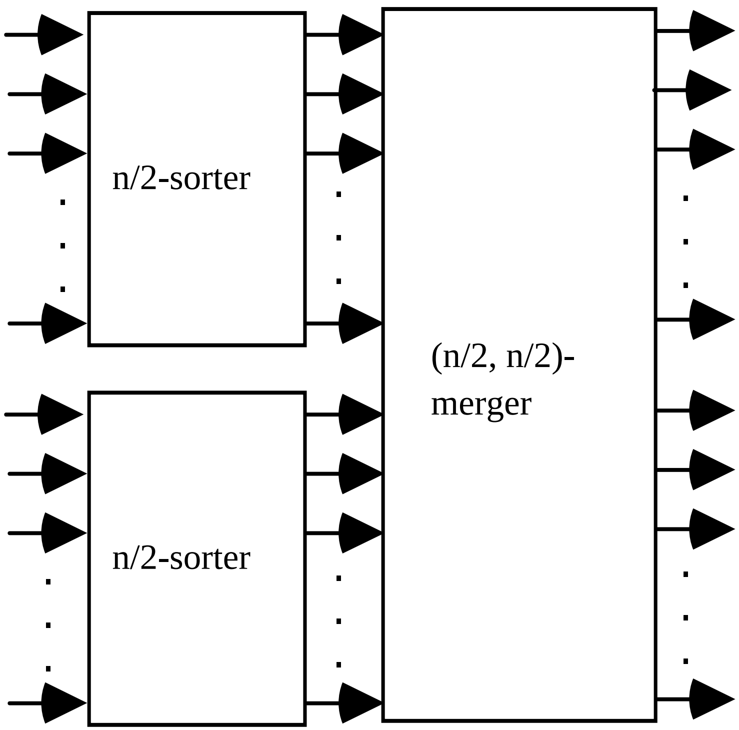



Fig. 7.10 The recursive structure of Batcher's even–odd merge sorting network.

Batcher's $(m, m)$ even-odd merger, for $m$ a power of 2:

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

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

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

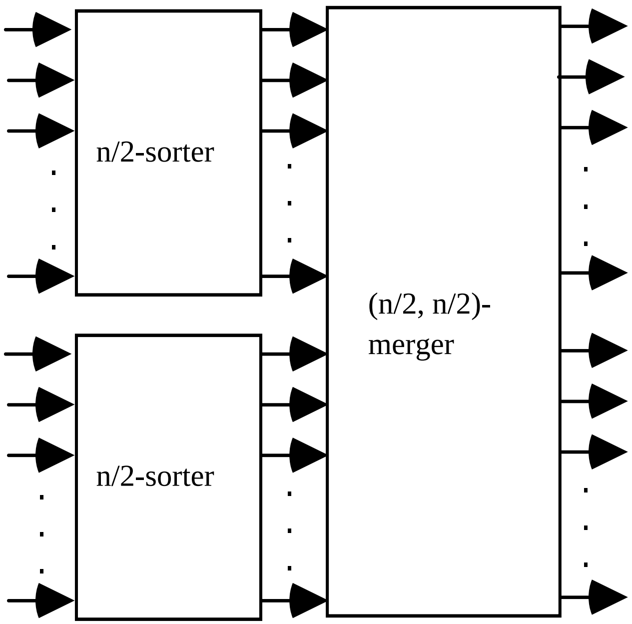
Batcher sorting networks based on the even-odd merge technique:

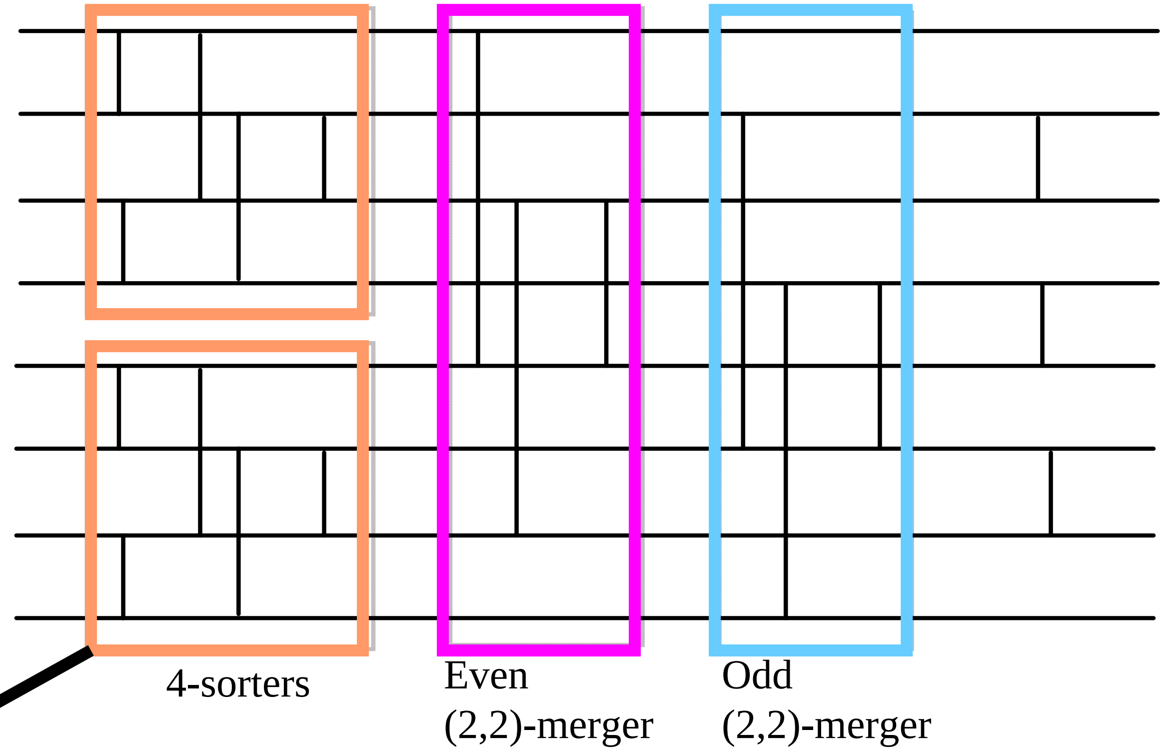
$$\begin{aligned} C(n) &= 2C(n/2) + (n/2)(\log_2(n/2)) + 1 \\ &\cong n(\log_2 n)^2/ 2 \end{aligned}$$

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

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

# Example Batcher's Even-Odd 8-Sorter



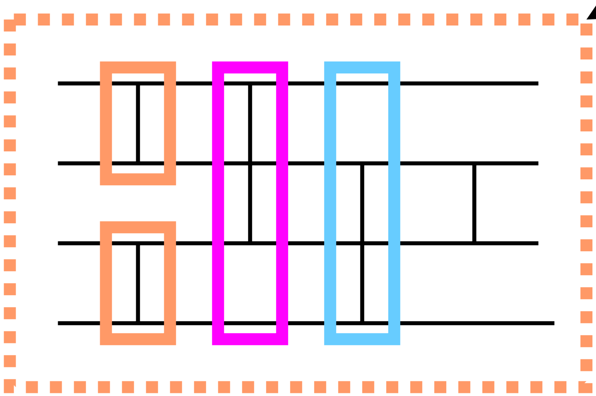
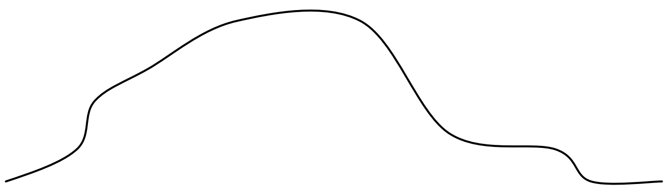



Fig. 7.11 Batcher's even-odd merge sorting network for eight inputs .

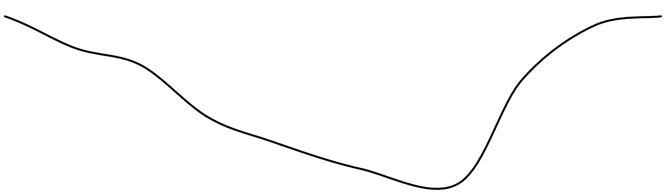
# Bitonic-Sequence Sorter

Bitonic sequence:

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



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



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

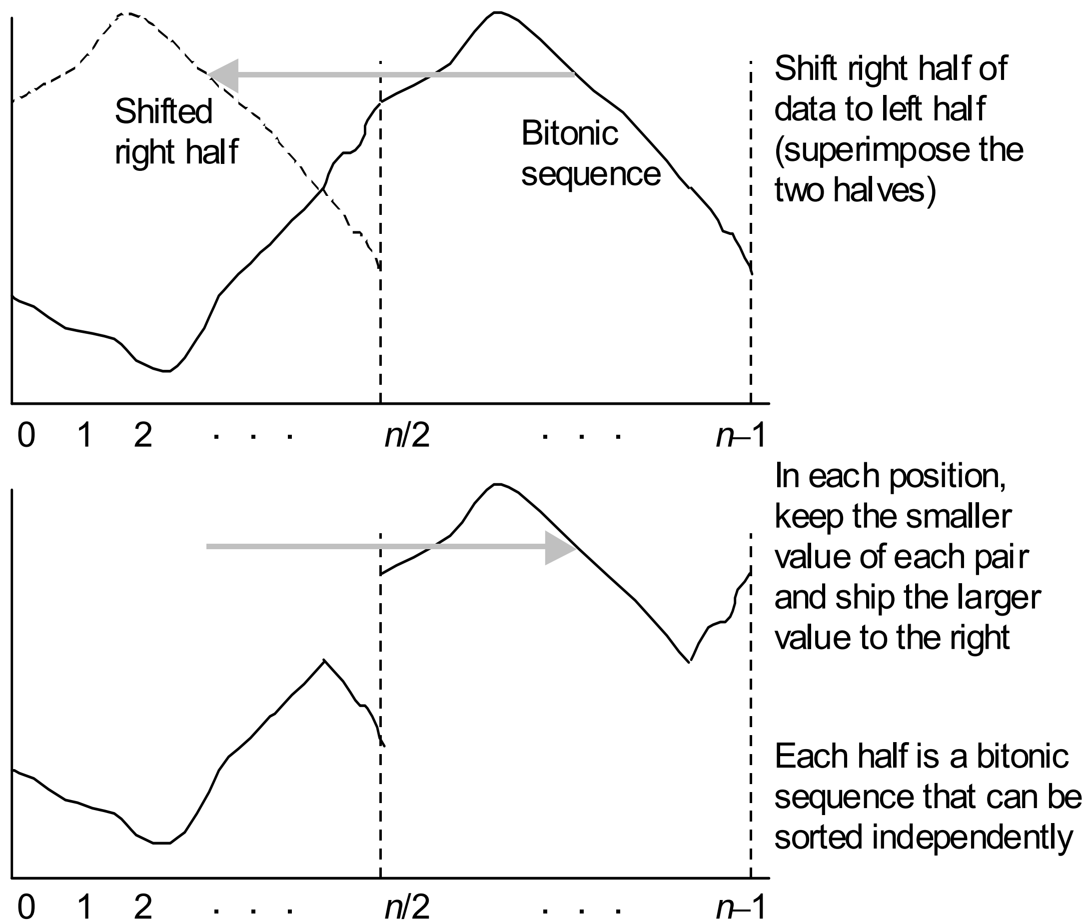



Fig. 14.2 Sorting a bitonic sequence on a linear array.

# Batcher's Bitonic Sorting Networks

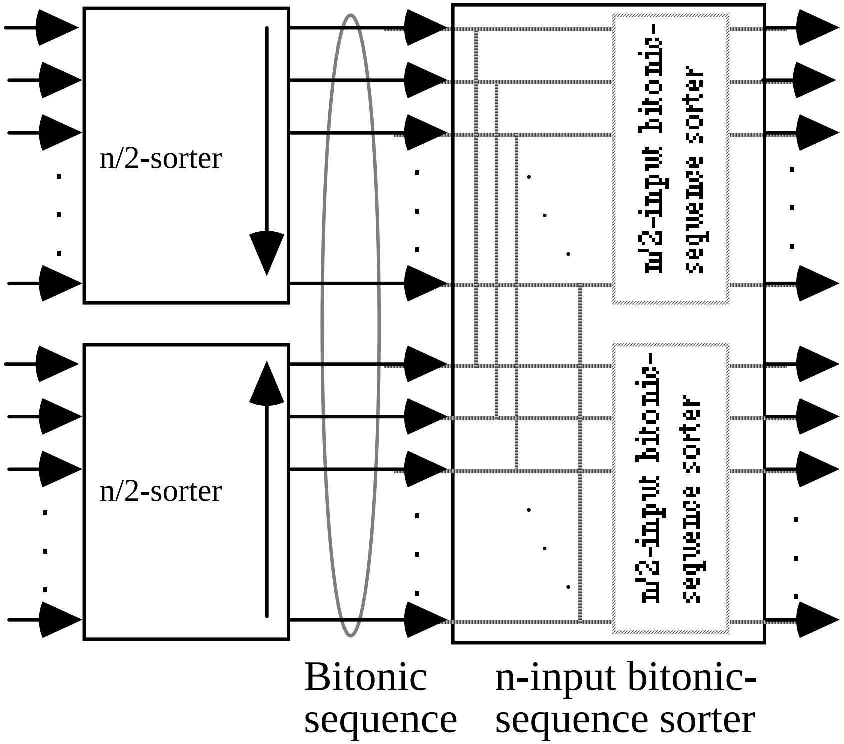



Fig. 7.12 The recursive structure of Batcher's bitonic sorting network.

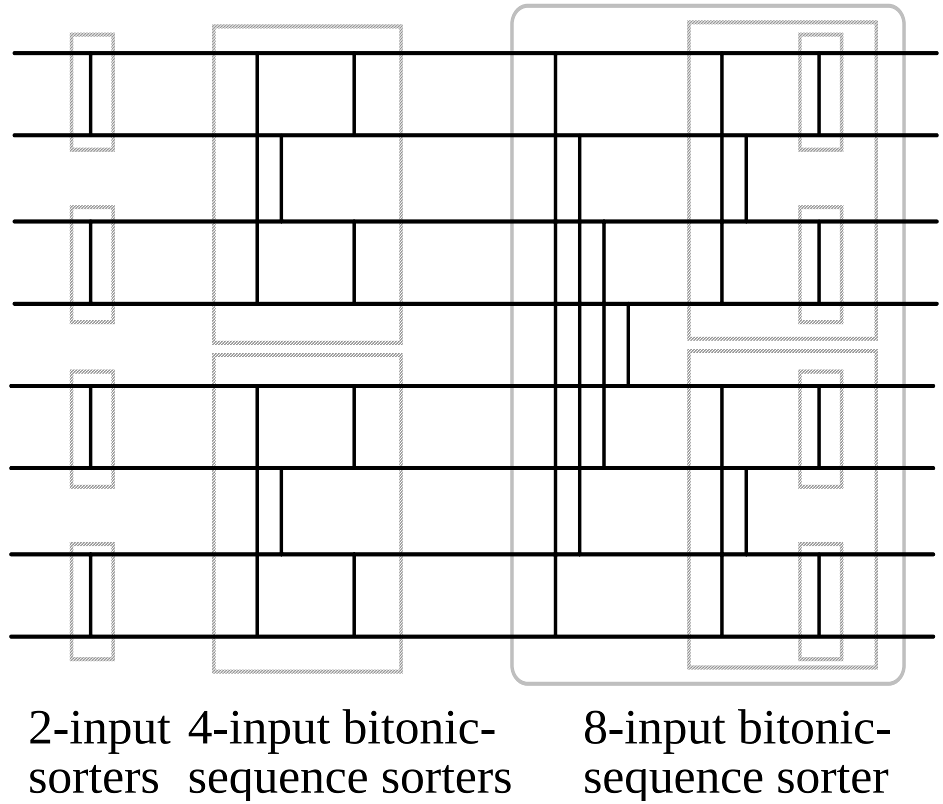



Fig. 7.13 Batcher's bitonic sorting network for eight inputs.

# 7.5 Other Classes of Sorting Networks

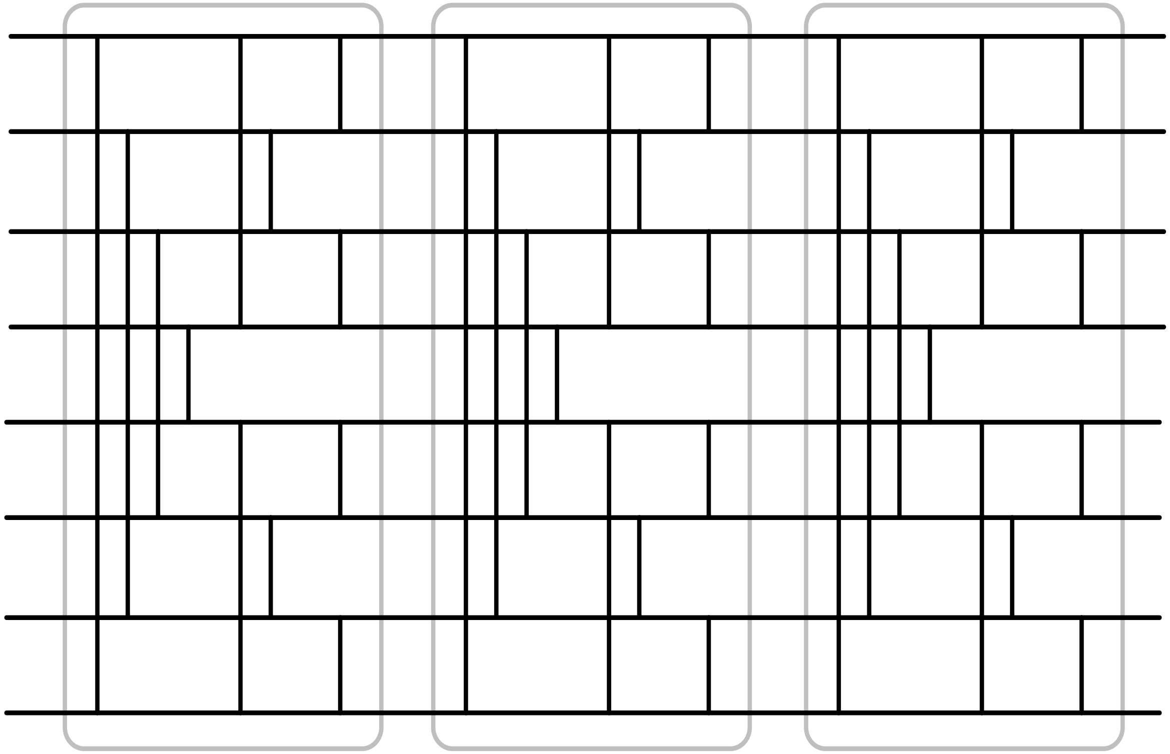


Fig. 7.14 Periodic balanced sorting network for eight inputs.

Desirable properties:

a. Regular / modular (easier VLSI layout).

b. Simpler circuits via reusing the blocks

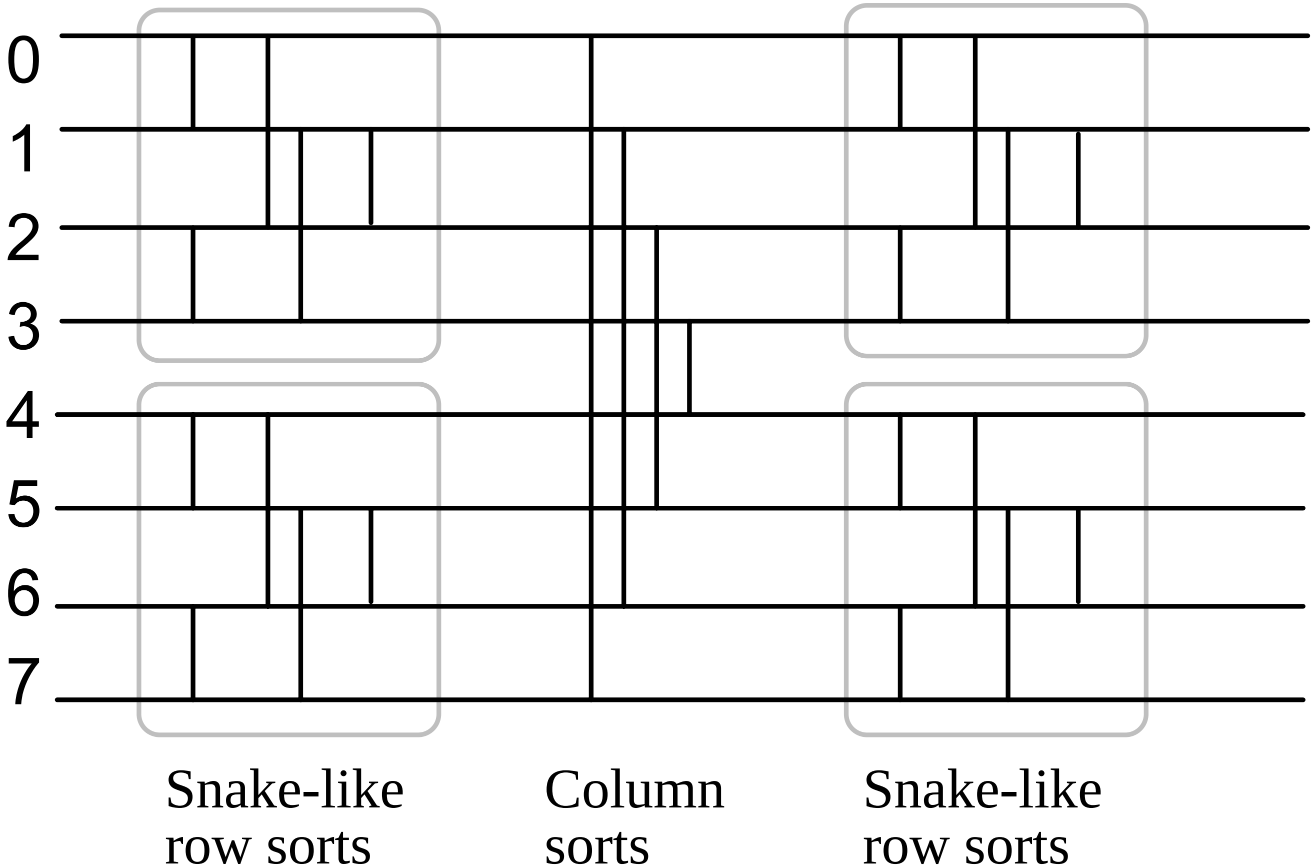
c. With an extra block tolerates some faults (missed exchanges)

d. With 2 extra blocks provides tolerance to single faults (a missed or incorrect exchange)

e. Multiple passes through faulty network (graceful degradation)

f. Single-block design becomes fault-tolerant by using an extra stage

# Shearsort-Based Sorting Networks (1)

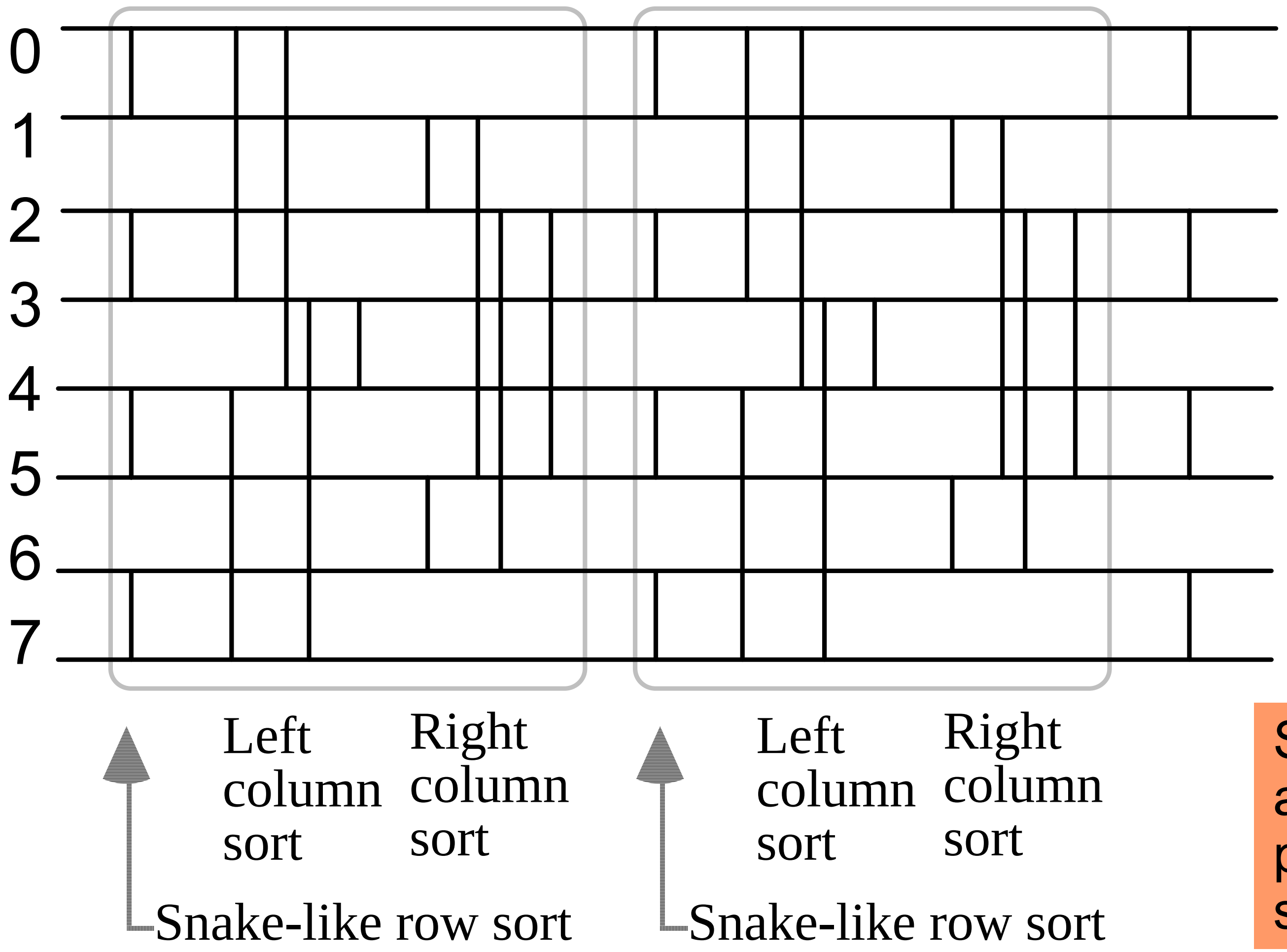


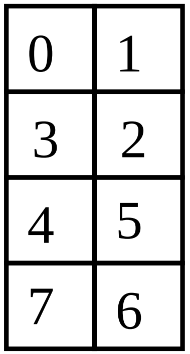

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 7 | 6 | 5 | 4 |

Corresponding 2-D mesh

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

# Shearsort-Based Sorting Networks (2)



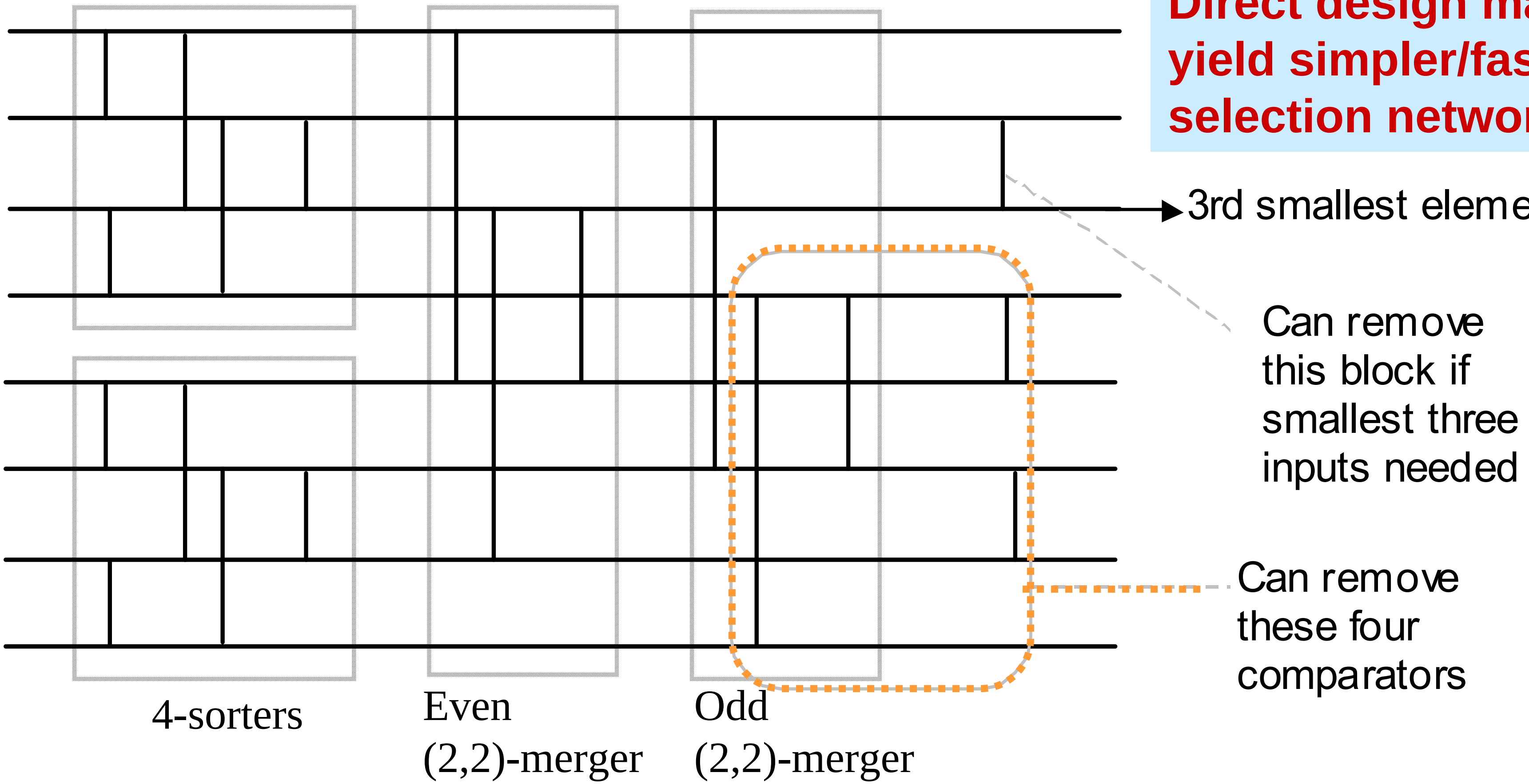

Corresponding 2-D mesh

Some of the same advantages as periodic balanced sorting networks

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

# 7.6 Selection Networks

**Direct design may yield simpler/faster selection networks**




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

# Categories of Selection Networks

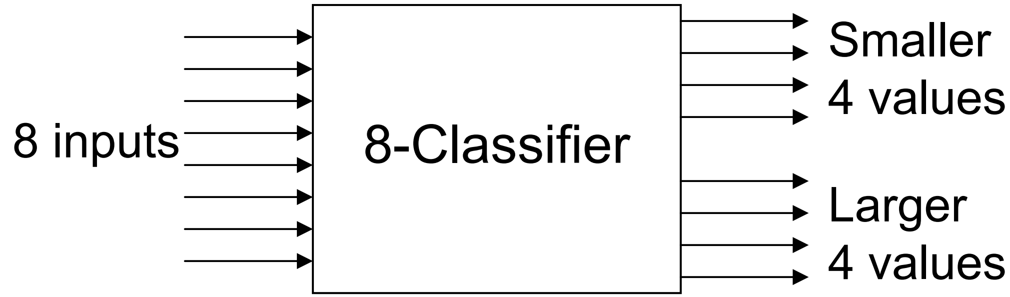
Unfortunately we know even less about selection networks than we do about sorting networks.

One can define three selection problems [Knut81]:

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

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

**Classifiers:**
Selectors that separate the smaller half of values from the larger half



# Type-III Selection Networks

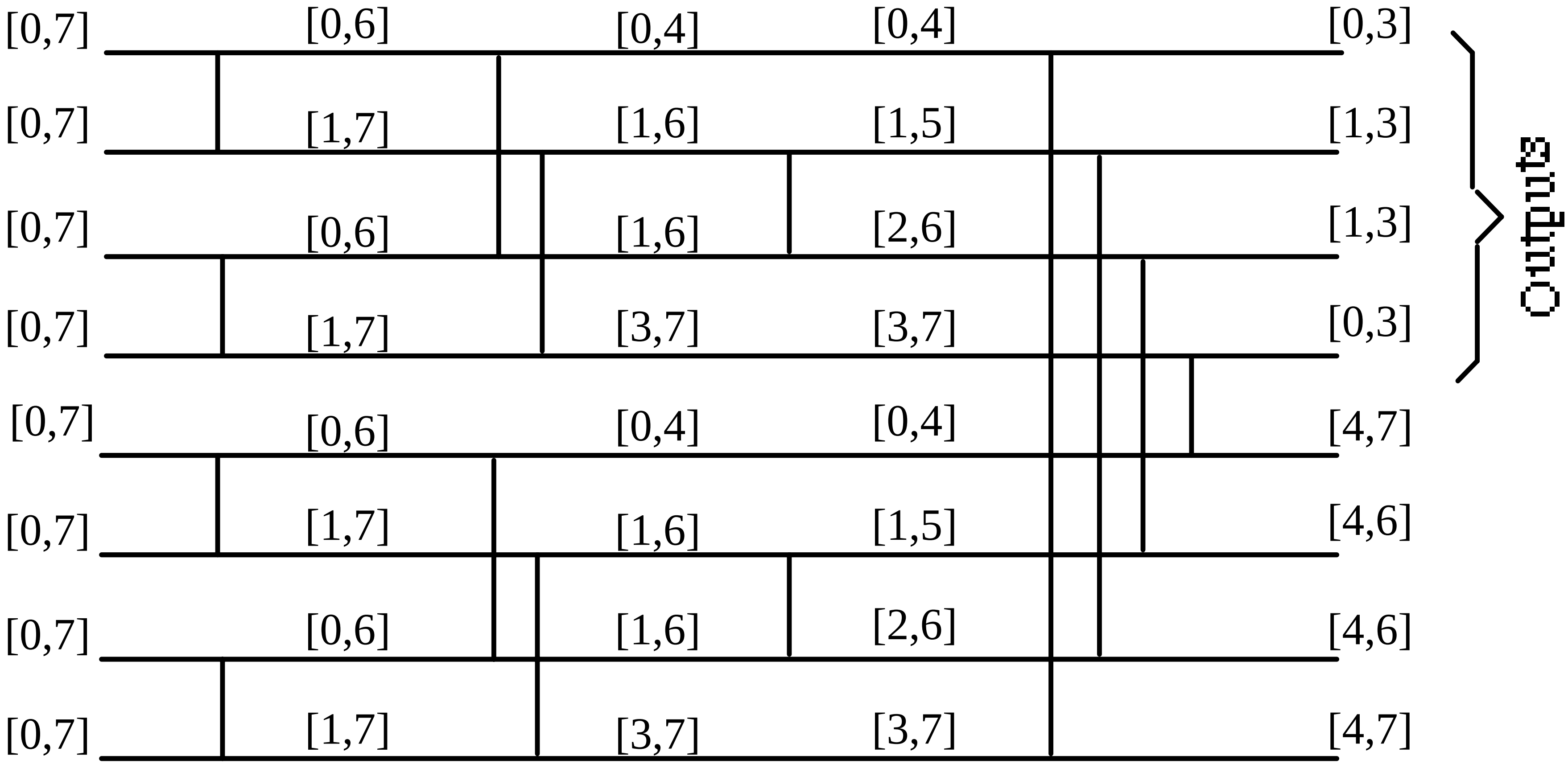



Figure 7.17 A type III (8, 4)-selector.

8-Classifier

# 8 Other Circuit-Level Examples

Complement our discussion of sorting and sorting nets with:

- Searching, parallel prefix computation, Fourier transform
- Dictionary machines, parallel prefix nets, FFT circuits

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

# 8.1 Searching and Dictionary Operations

Parallel $p$-ary search on PRAM

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

Speedup $\cong \log p$

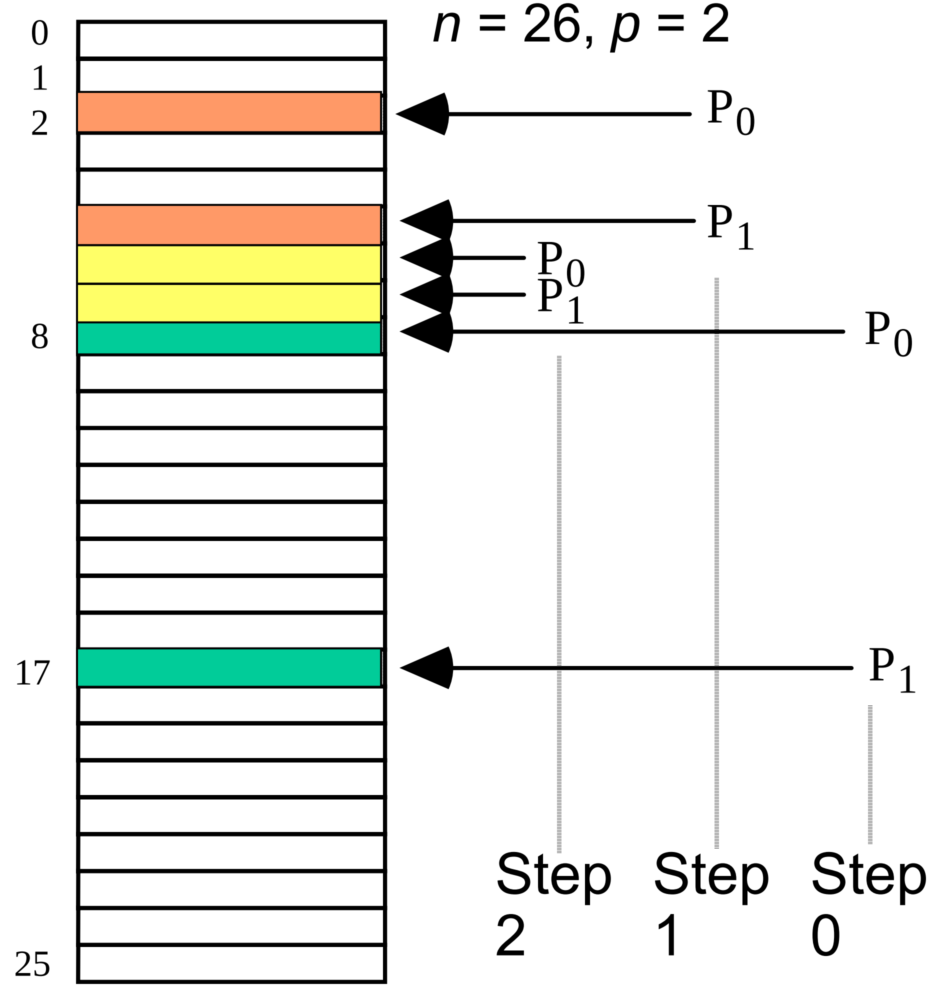
Optimal: no comparison-based search algorithm can be faster

A single search in a sorted list can't be significantly speeded up through parallel processing, but all hope is not lost:

Dynamic data (sorting overhead)

Batch searching (multiple lookups)

Example:
$n = 26$, $p = 2$



# Dictionary Operations

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

| | |
|---|---|
| *search*($y$) | Find record with key $y$; return its associated data |
| *insert*($y$, $z$) | Augment list with a record: key = $y$, data = $z$ |
| *delete*($y$) | Remove record with key $y$; return its associated data |

Some or all of the following operations 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 is right after $y$ in sorted order |
| *findprev*($y$) | Find record whose key is right before $y$ in sorted order |
| *extractmin* | Remove record(s) with min key; return data |
| *extractmax* | Remove record(s) with max key; return data |
| *extractmed* | Remove record(s) with median key value; return data |

Priority queue operations: *findmin*, *extractmin* (or *findmax*, *extractmax*)

# 8.2 A Tree-Structured Dictionary Machine

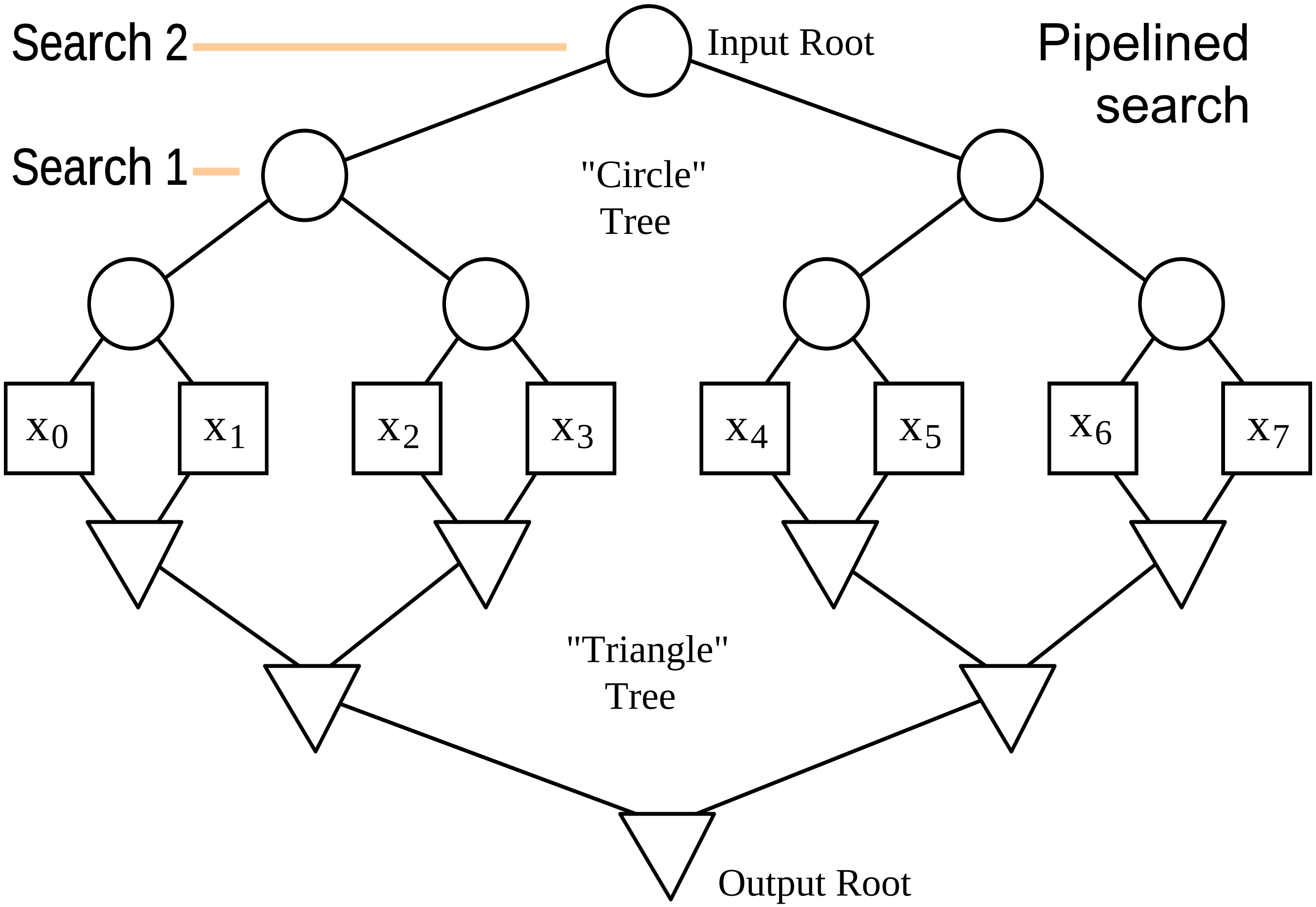



Fig. 8.1 A tree-structured dictionary machine.

**Combining in the triangular nodes**

*search*(*y*): Pass OR of match signals & data from “yes” side

*findmin* / *findmax*: Pass smaller / larger of two keys & data

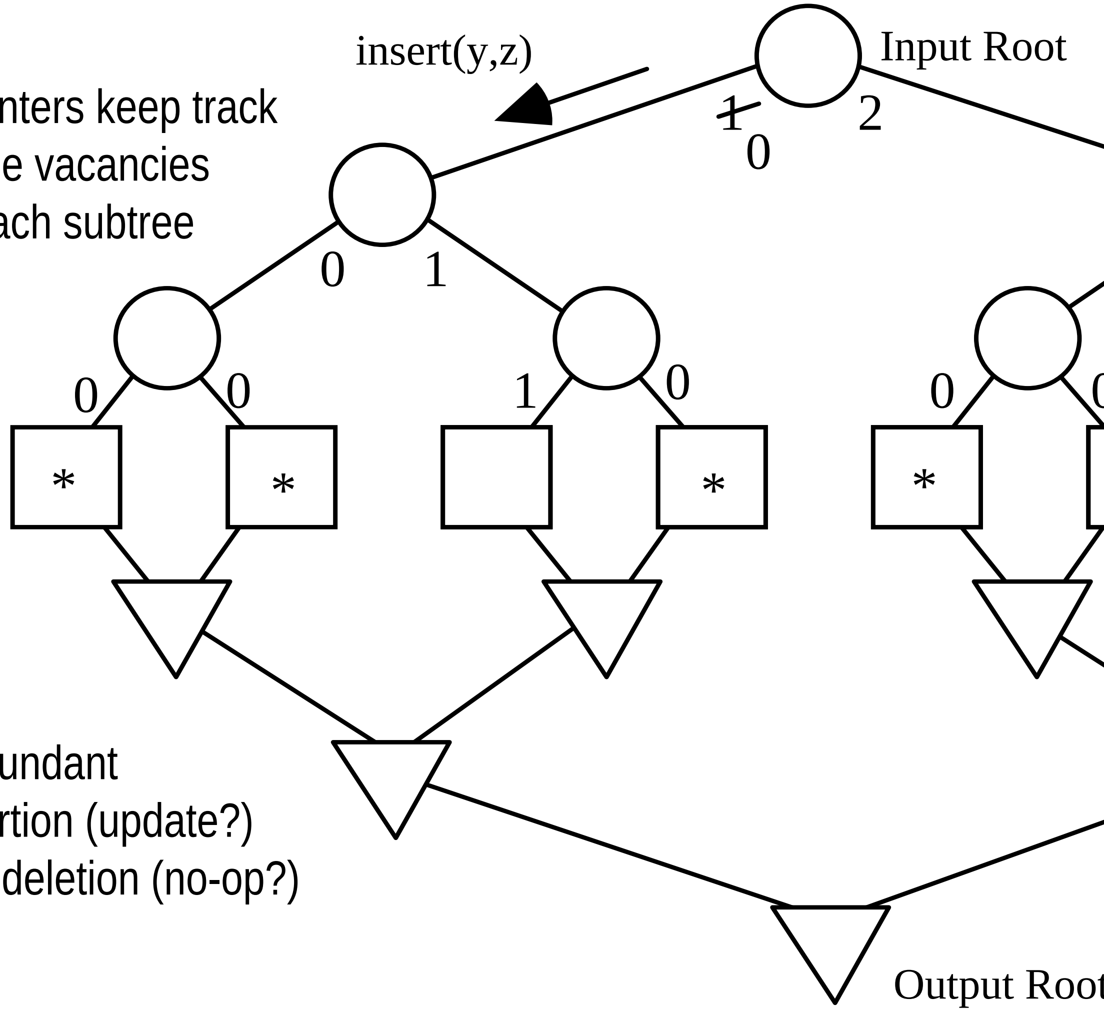
*findmed*: Not supported here

*findbest*(*y*): Pass the larger of two match-degree indicators along with associated record

# Insertion and Deletion in the Tree Machine

Counters keep track of the vacancies in each subtree

Deletion needs second pass to update the vacancy counters




Redundant insertion (update?) and deletion (no-op?)

Implementation: Merge the circle and triangle trees by folding

Figure 8.2 Tree machine storing 5 records and containing 3 free slots.

# Systolic Data Structures

Each node holds the smallest (S), median (M),,and largest (L) value in its subtree

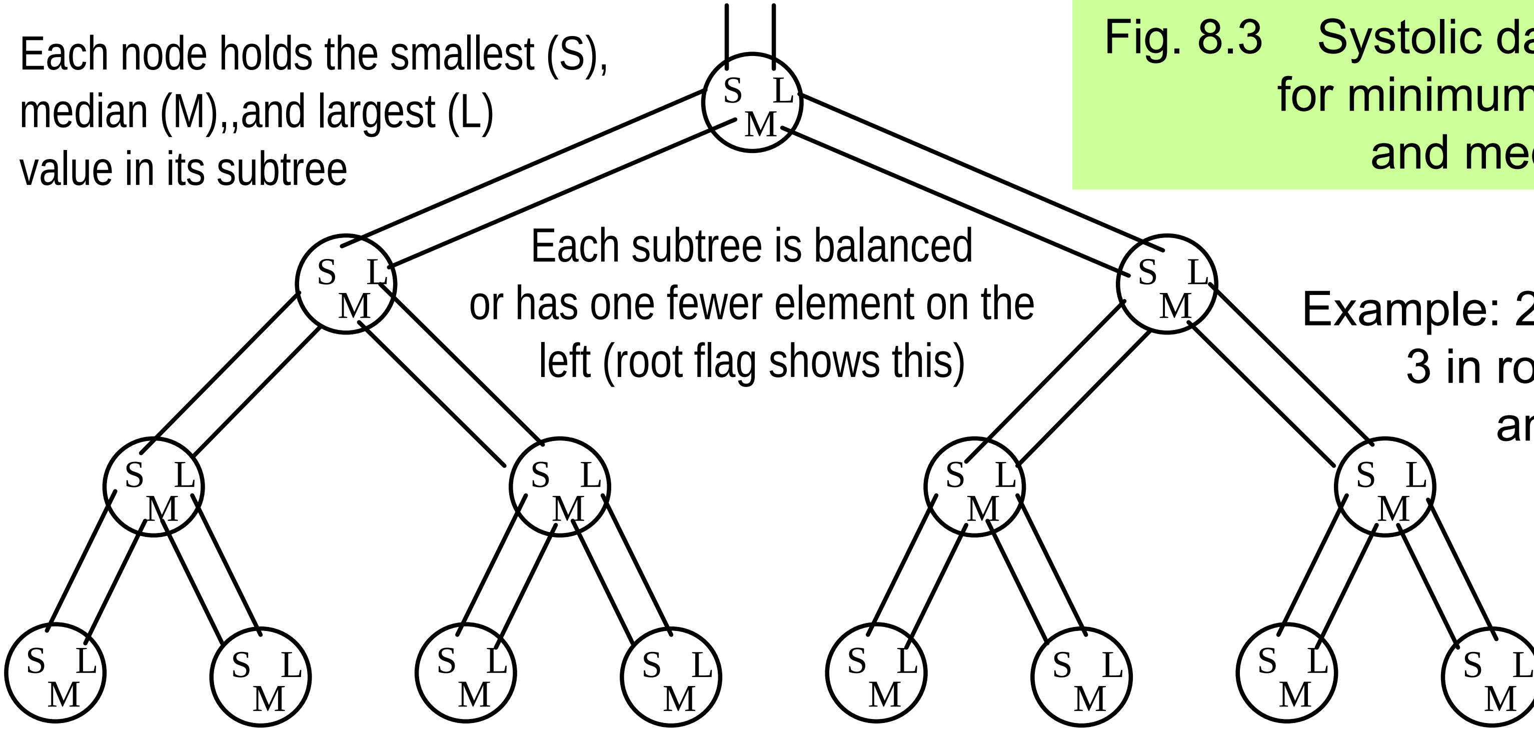
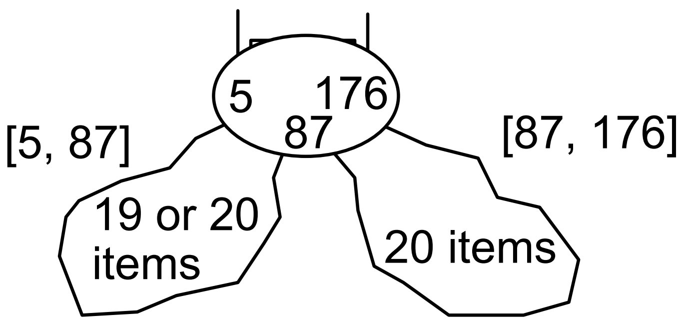



Fig. 8.3 Systolic data structure for minimum, maximum, and median finding.

Example: 20 elements, 3 in root, 8 on left, and 9 on right

8 elements: 3 + 2 + 3




Insert 2
Insert 20
Insert 127
Insert 195

Extractmin
Extractmed
Extractmax

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

# 8.3 Parallel Prefix Computation

Example: Prefix sums

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

Sequential time with one processor is O($n$)
Simple pipelining does not help

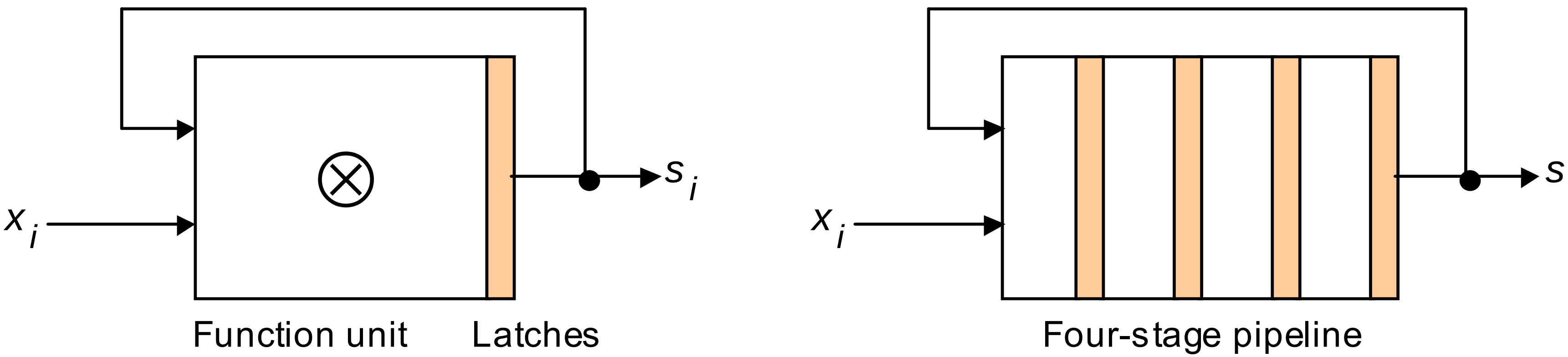



Fig. 8.4 Prefix computation using a latched or pipelined function unit.

# Improving the Performance with Pipelining

Ignoring pipelining overhead, it appears that we have achieved a speedup of 4 with 3 "processors." Can you explain this anomaly? (Problem 8.6a)

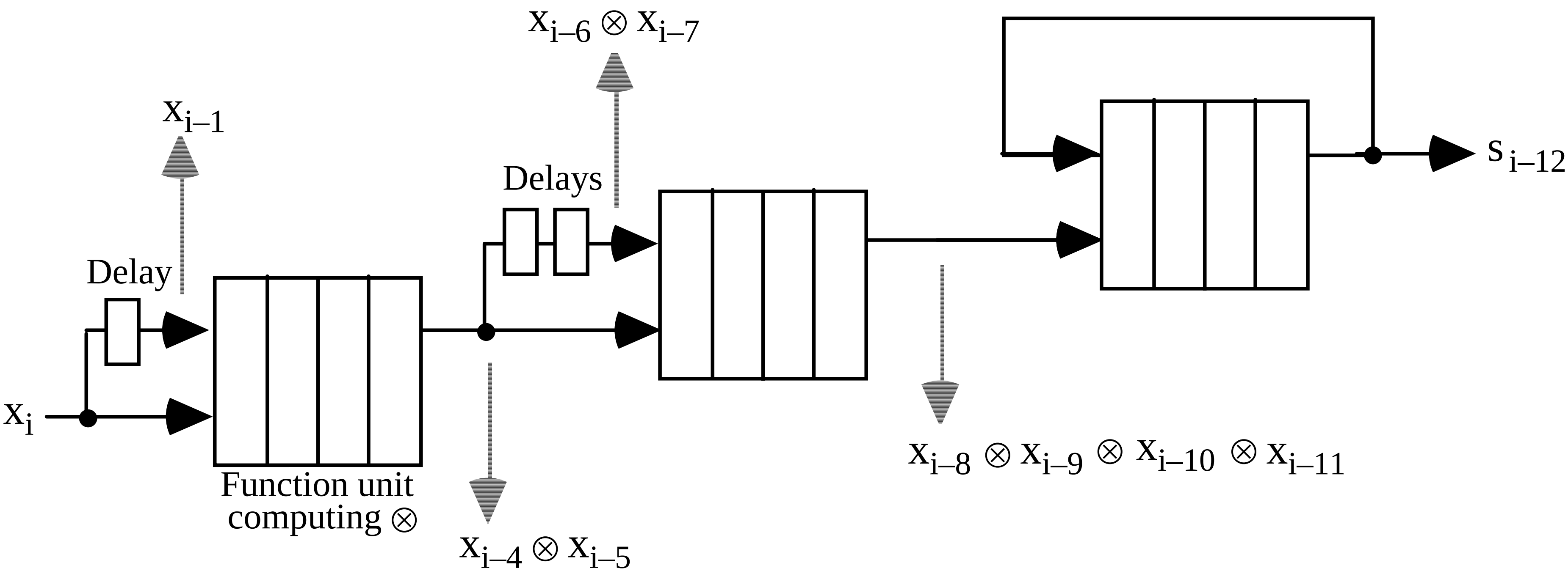
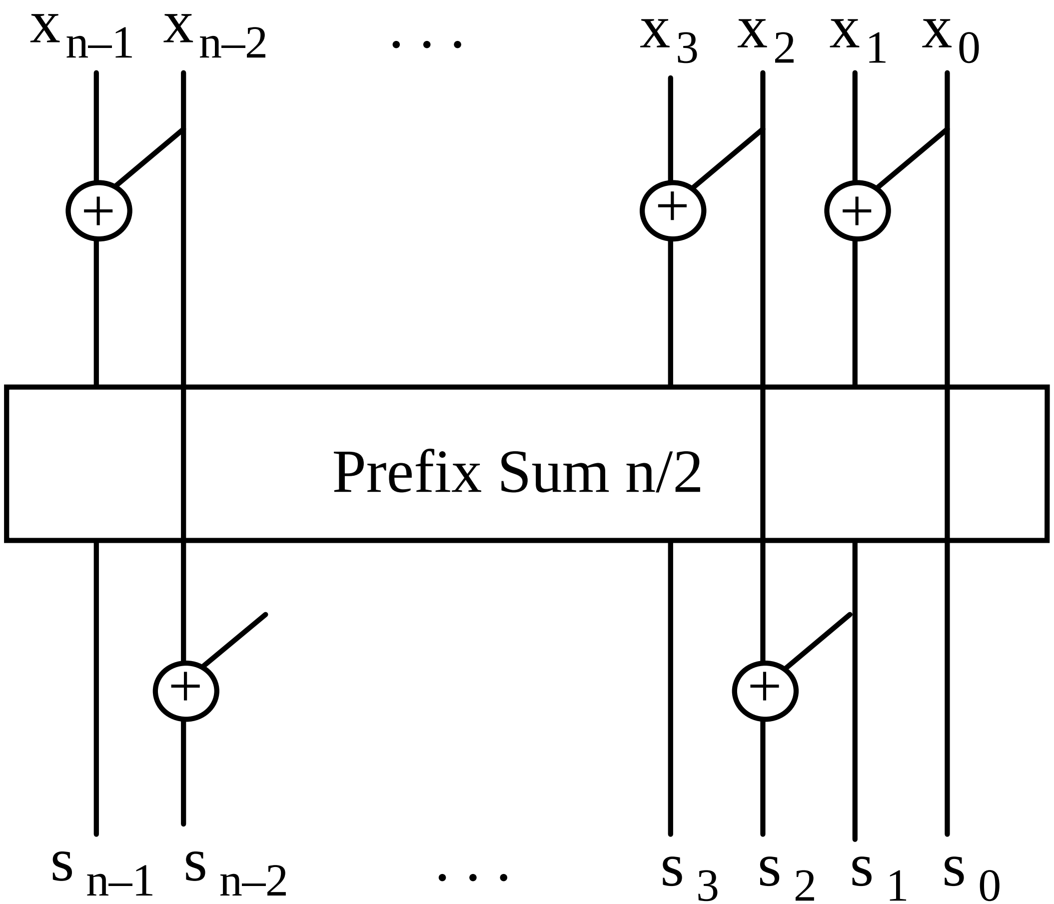



Fig. 8.5 High-throughput prefix computation using a pipelined function unit.

# 8.4 Parallel Prefix Networks



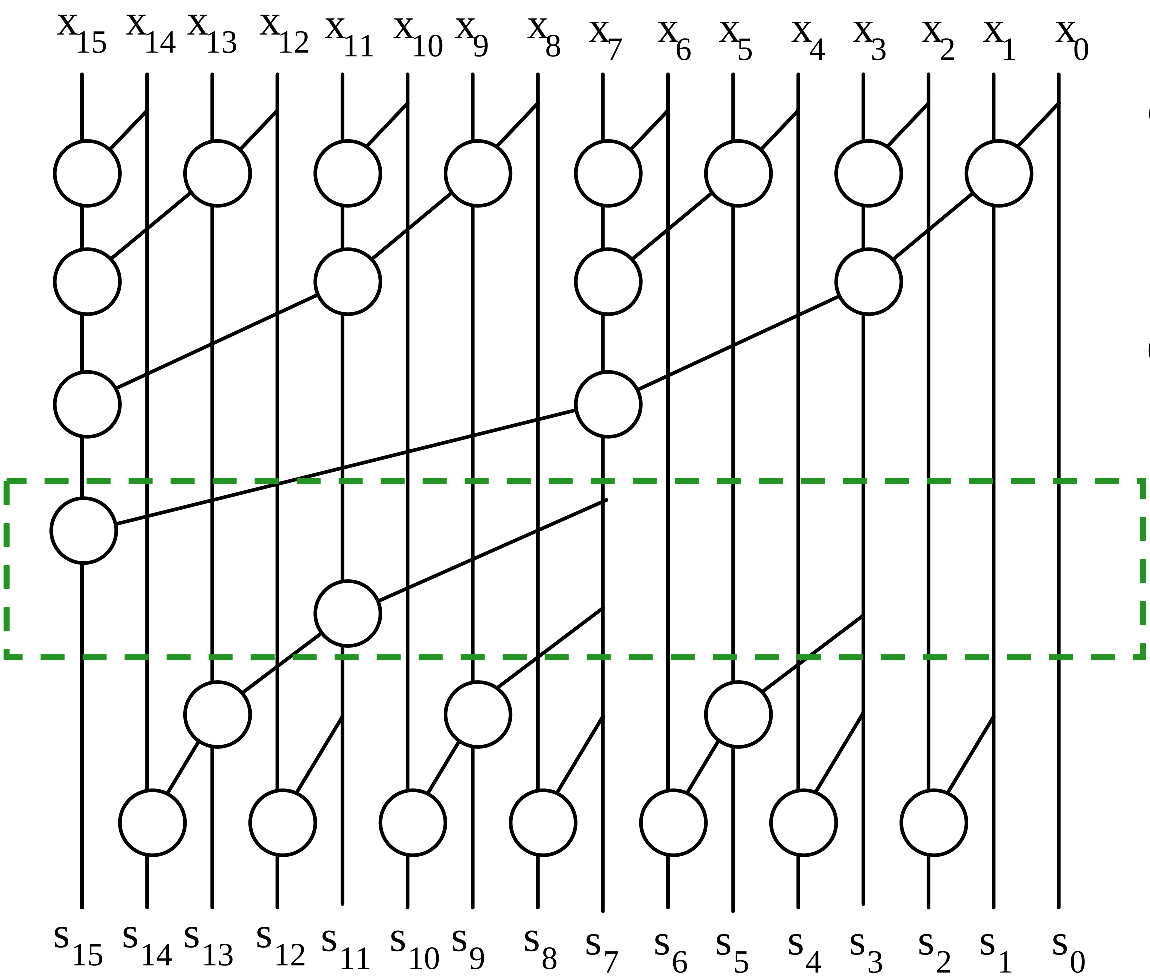

$$T(n) = T(n/2) + 2 = 2\log_2 n - 1$$

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

This is the Brent-Kung Parallel prefix network (its delay is actually $2 \log_2 n - 2$)

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

# Example of Brent-Kung Parallel Prefix Network




Originally developed by Brent and Kung as part of a VLSI-friendly carry lookahead adder

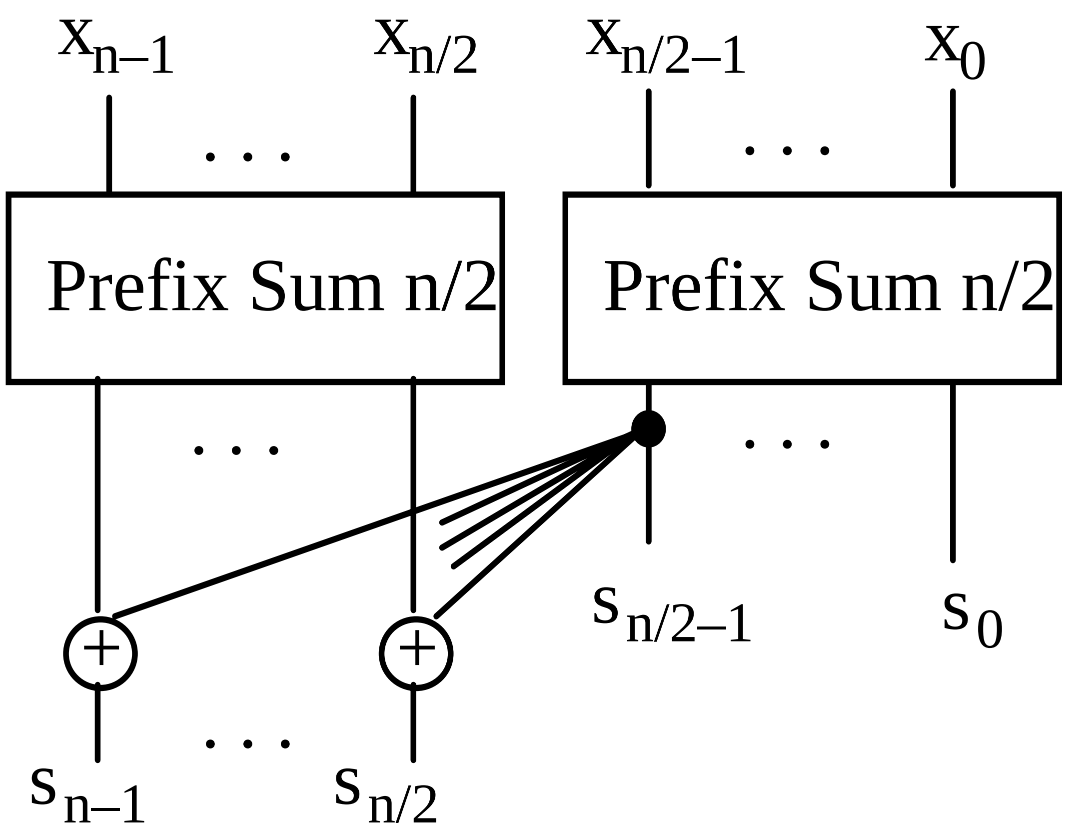
$T(n) = 2 \log_2 n - 2$

$C(n) = 2n - 2 - \log_2 n$

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

# Another Divide-and-Conquer Design

Ladner-Fischer construction



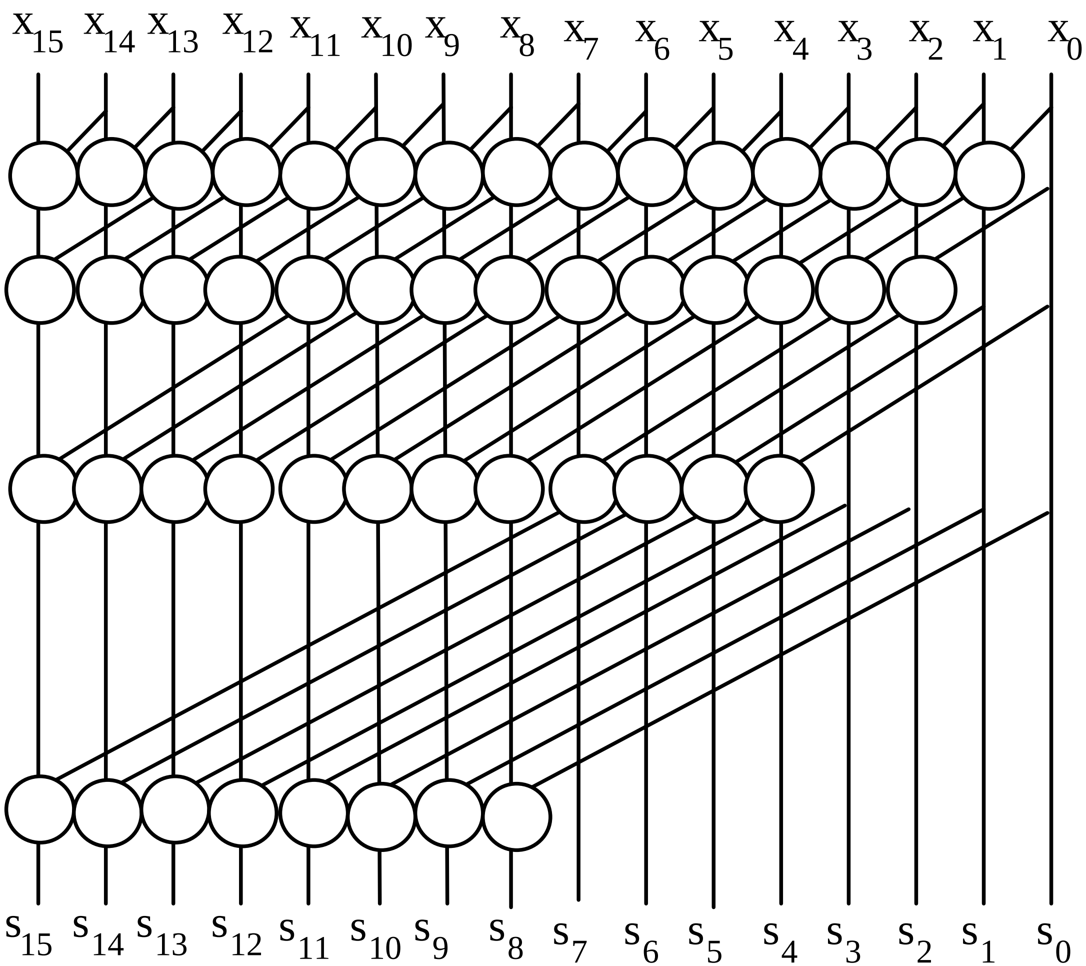

$T(n) = T(n/2) + 1$
$= \log_2 n$

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

Simple Ladner-Fisher Parallel prefix network (its delay is optimal, but has fan-out issues if implemented directly)

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

# Example of Kogge-Stone Parallel Prefix Network



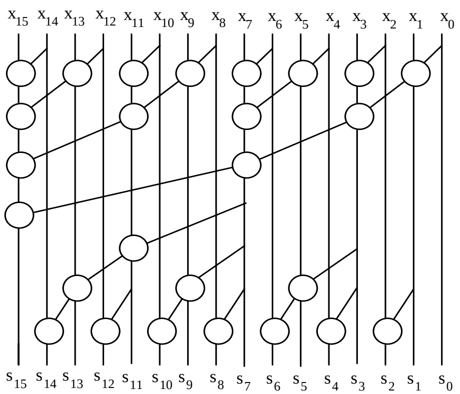

$T(n) = \log_2 n$

$C(n) = (n-1) + (n-2) + (n-4) + \ldots + n/2 = n \log_2 n - n - 1$

Optimal in delay, but too complex in number of cells and wiring pattern

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

# Comparison and Hybrid Parallel Prefix Networks




Brent/Kung
6 levels
26 cells

Kogge/Stone
4 levels
49 cells

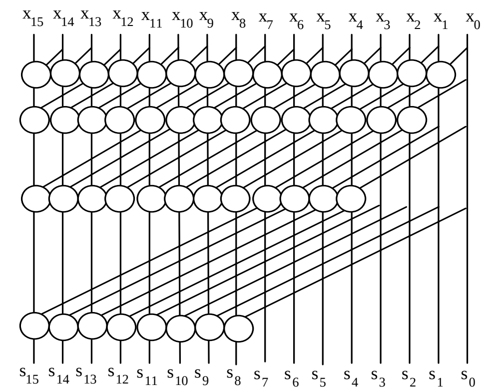
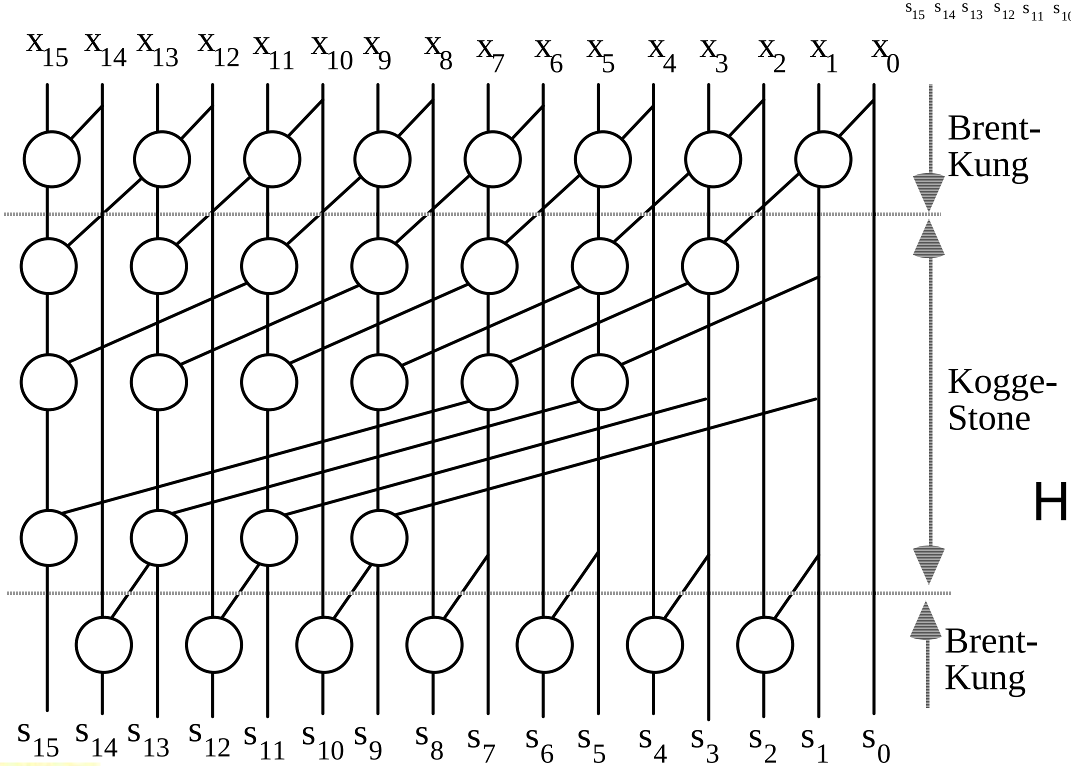



Fig. 8.10 A hybrid Brent–Kung / Kogge–Stone parallel prefix graph for $n$ = 16.



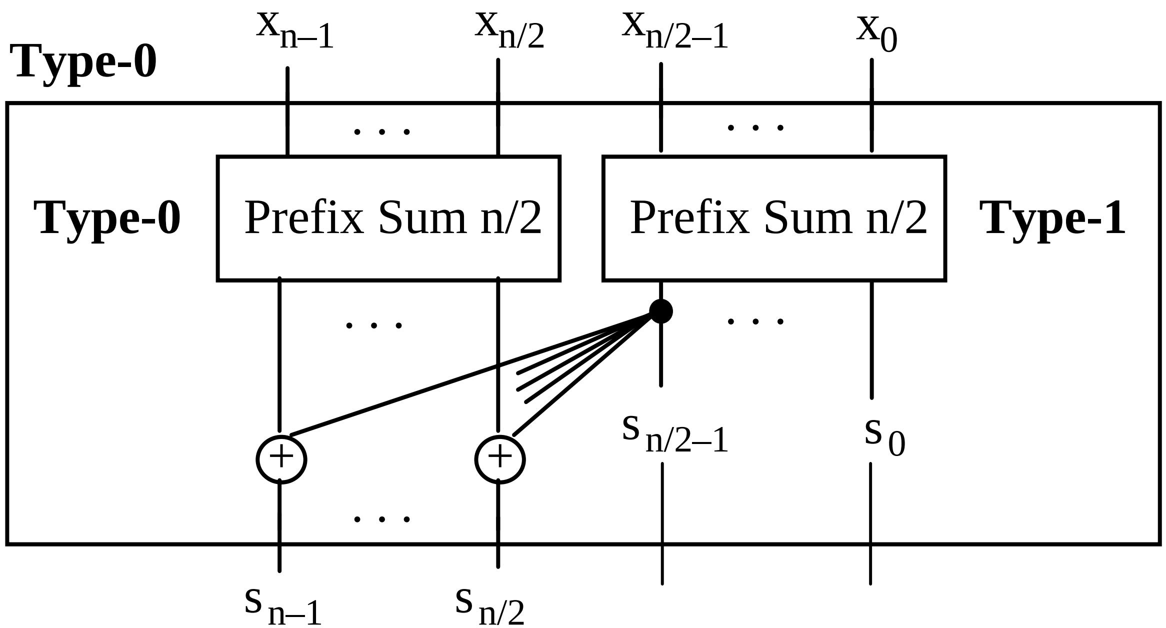

Han/Carlson
5 levels
32 cells

# Linear-Cost, Optimal Ladner-Fischer Networks

Define a type-$x$ parallel prefix network as one that:
- Produces the leftmost output in optimal $\log_2 n$ time
- Yields all other outputs with at most $x$ additional delay

Note that even the Brent-Kung network produces the leftmost output in optimal time

We are interested in building a type-0 overall network, but can use type-$x$ networks ($x$ > 0) as component parts




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

# 8.5 The Discrete Fourier Transform

DFT yields output sequence $y_i$ based on input sequence $x_i$ $(0 \le i < n)$

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

O($n^2$)-time naïve algorithm

where $\omega_n$ is the $n$th primitive root of unity; $\omega_n^n = 1$, $\omega_n^j \ne 1$ $(1 \le j < n)$

Examples: $\omega_4$ is the imaginary number $i$ and $\omega_3 = (-1 + i\sqrt{3})/2$

The inverse DFT is almost exactly the same computation:

$$x_i = (1/n) \sum_{j=0 \text{ to } n-1} \omega_n^{-ij} y_j$$

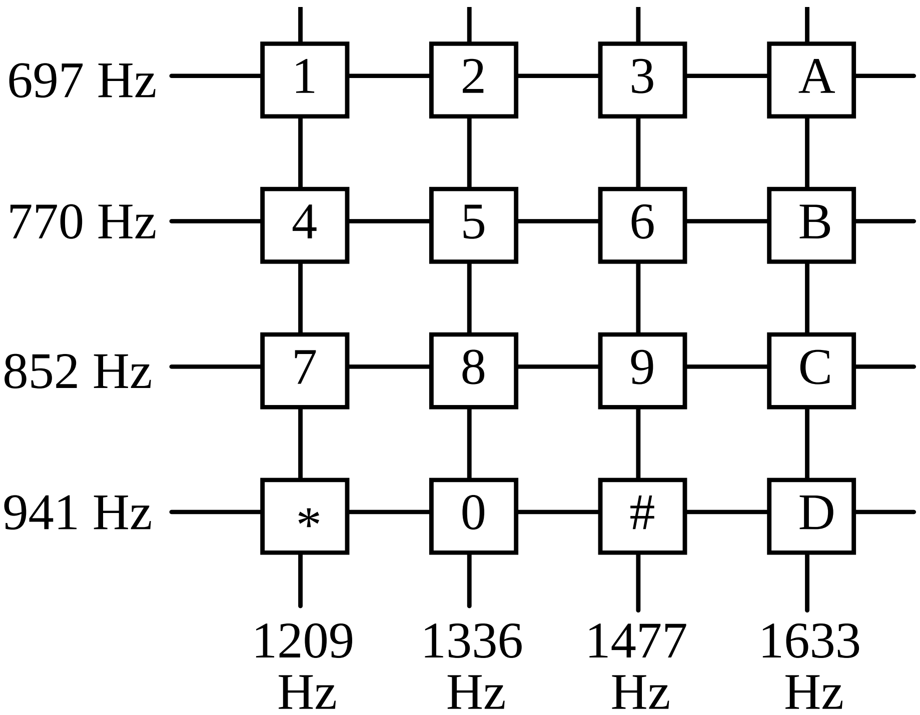
**Fast Fourier Transform (FFT):**

O($n \log n$)-time DFT algorithm that derives $y$ from half-length sequences $u$ and $v$ that are DFTs of even- and odd-indexed inputs, respectively

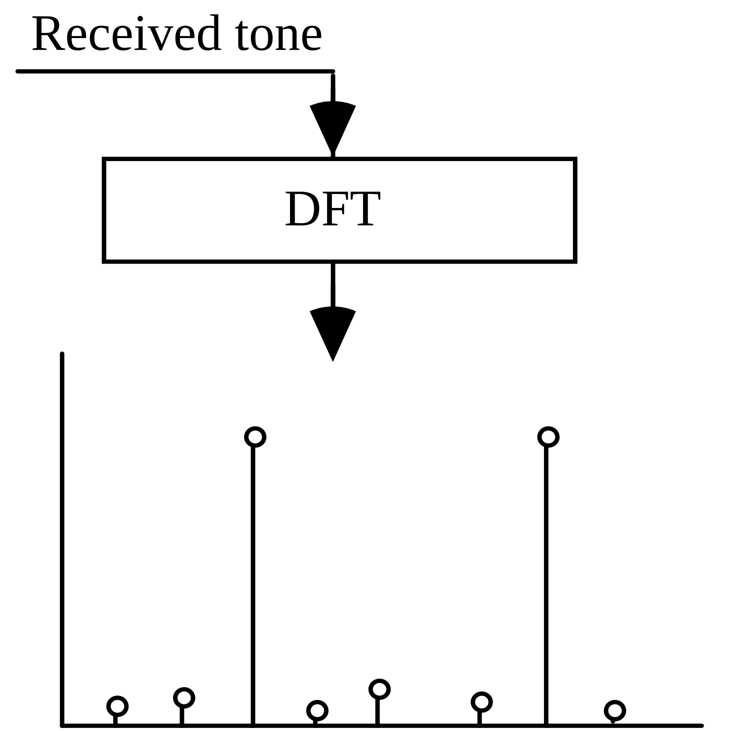
$$y_i = u_i + \omega_n^i v_i \qquad (0 \le i < n/2)$$
$$y_{i+n/2} = u_i + \omega_n^{i+n/2} v_i$$

| | |
|---|---|
| $T(n) = 2T(n/2) + n = n \log_2 n$ | sequentially |
| $T(n) = T(n/2) + 1 = \log_2 n$ | in parallel |

# Application of DFT to Spectral Analysis

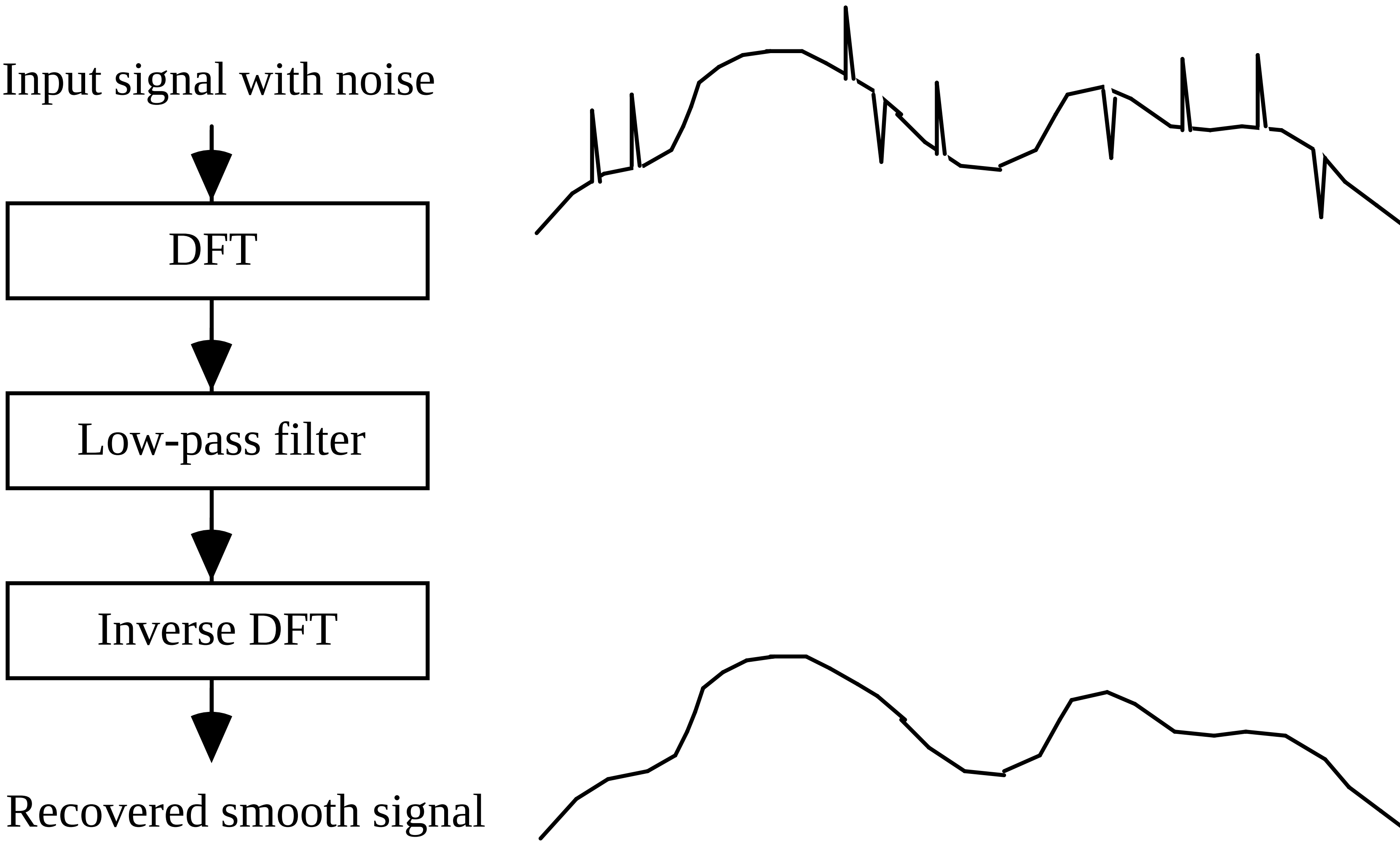



Tone frequency assignments
for touch-tone dialing




Frequency spectrum of received tone

# Application of DFT to Smoothing or Filtering



# 8.6 Parallel Architectures for FFT

*u*: DFT of even-indexed inputs
*v*: DFT of odd-indexed inputs

$$y_i = u_i + \omega_n^i v_i \qquad (0 \le i < n/2)$$
$$y_{i+n/2} = u_i + \omega_n^{i+n/2} v_i$$

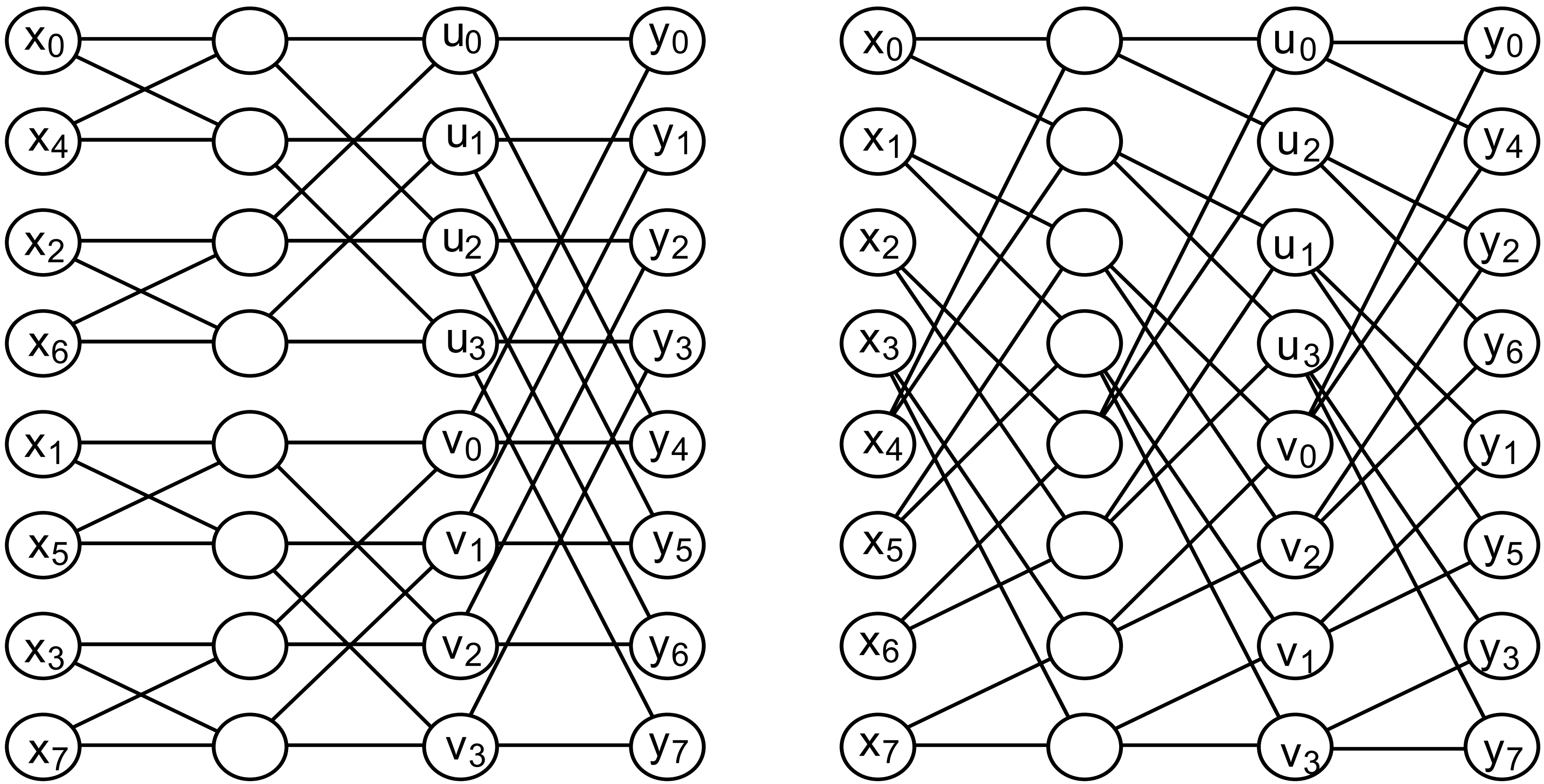



Fig. 8.11 Butterfly network for an 8-point FFT.

# Variants of the Butterfly Architecture

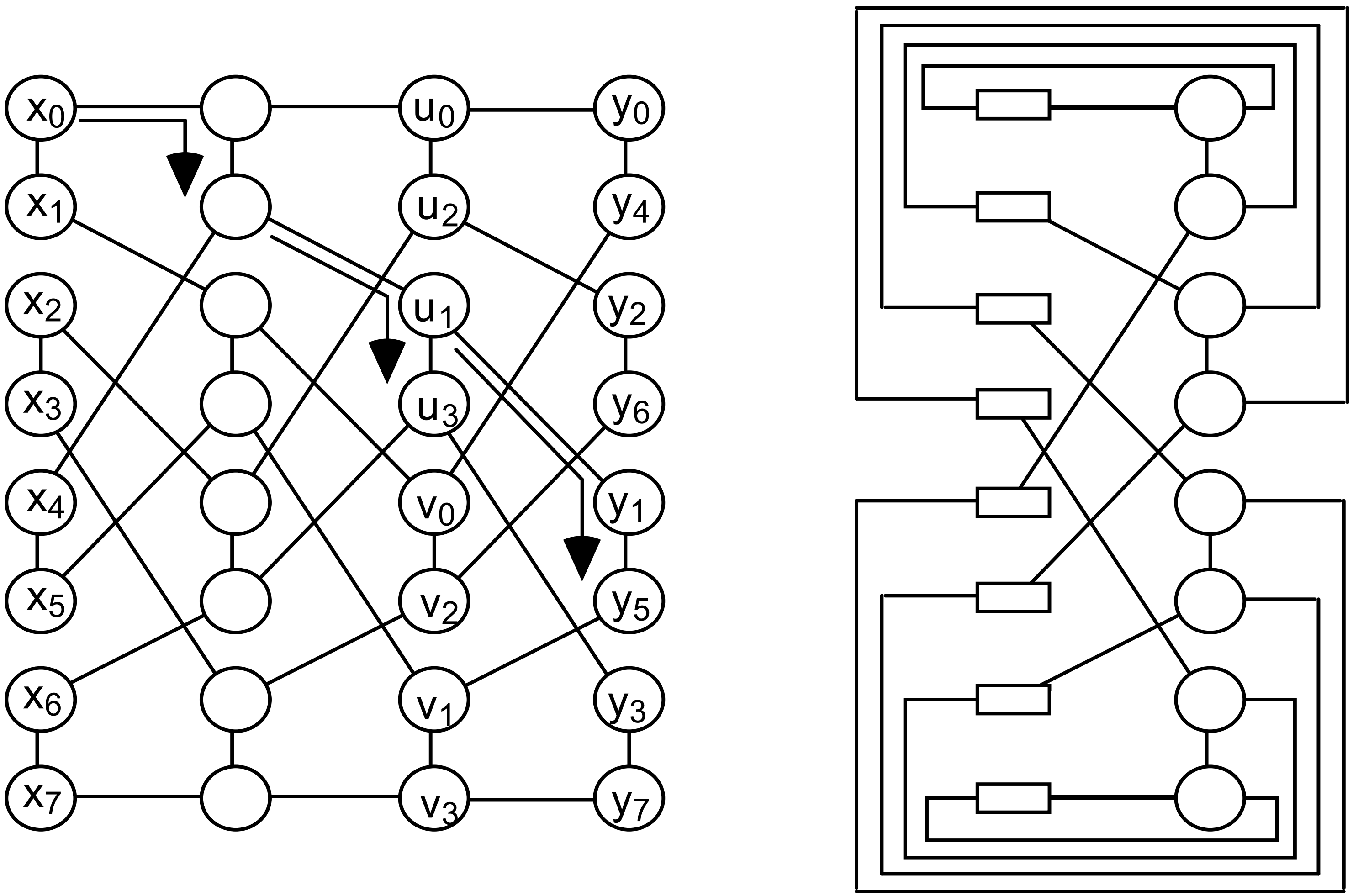
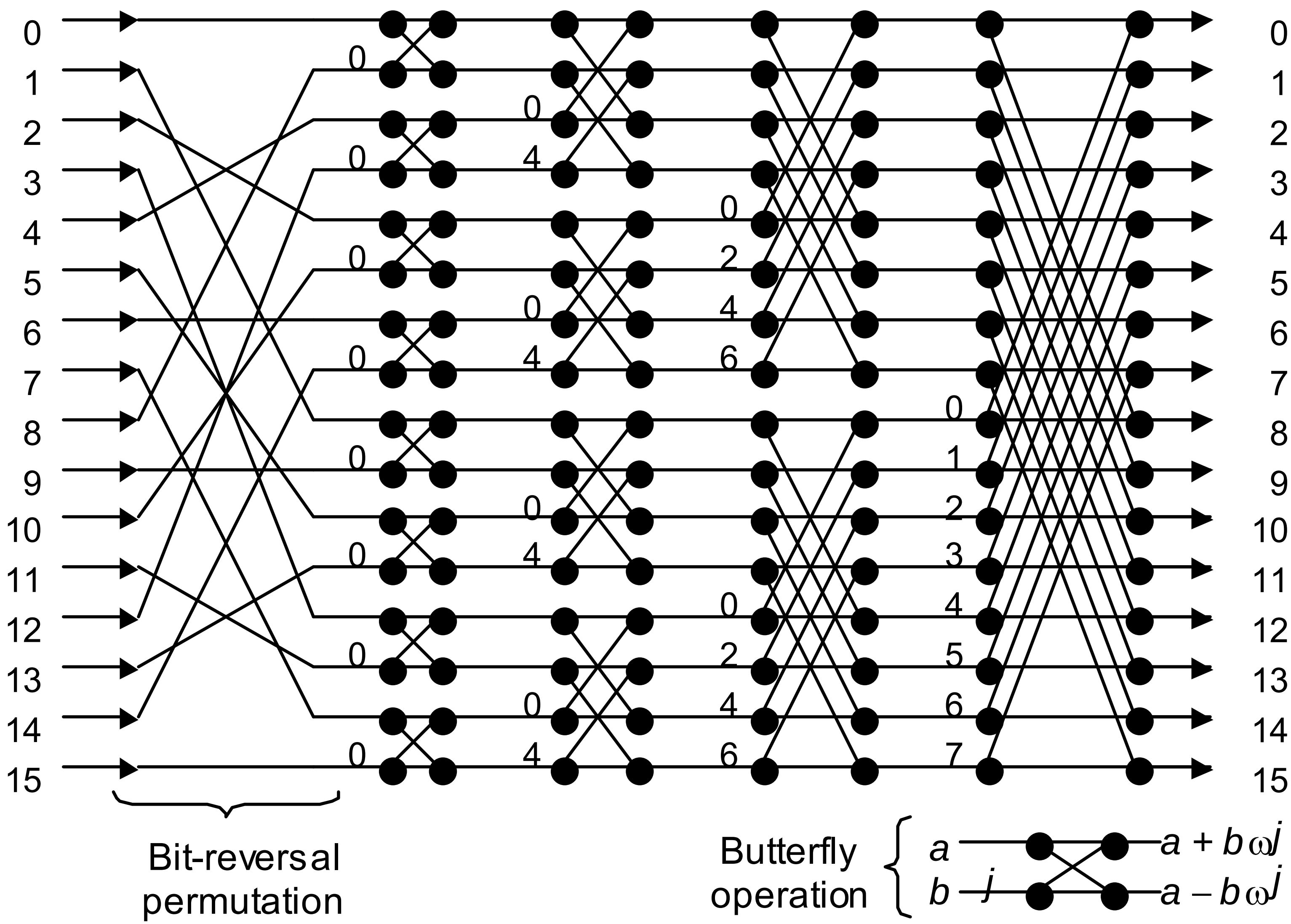



Fig. 8.12 FFT network variant and its shared-hardware realization.

# Computation Scheme for 16-Point FFT



# More Economical FFT Hardware

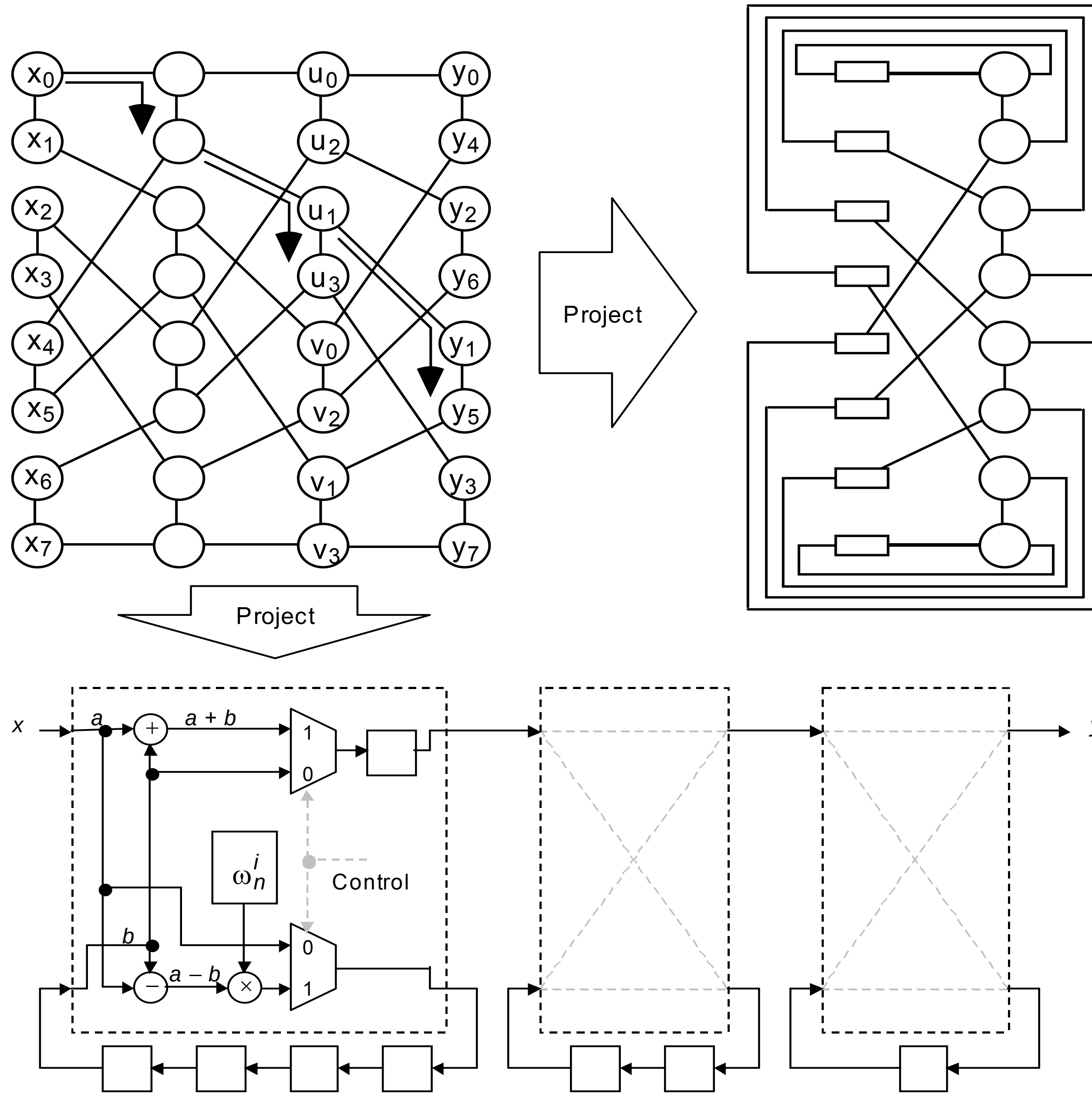
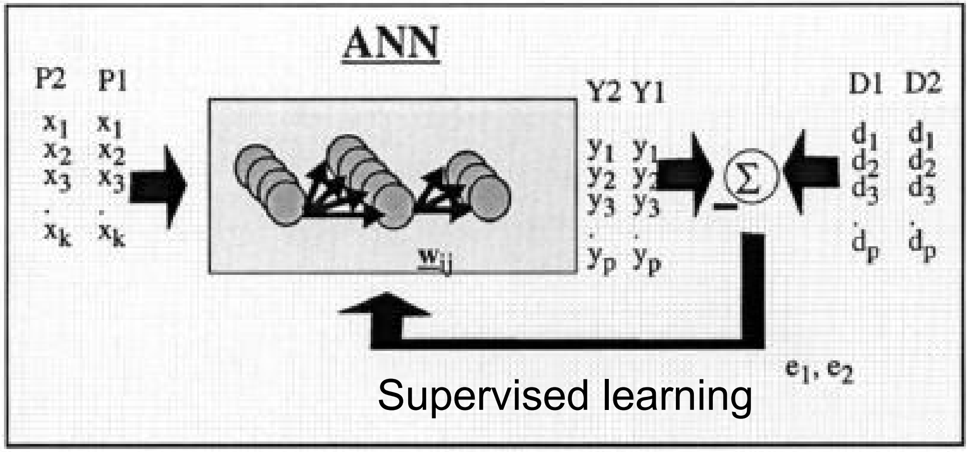
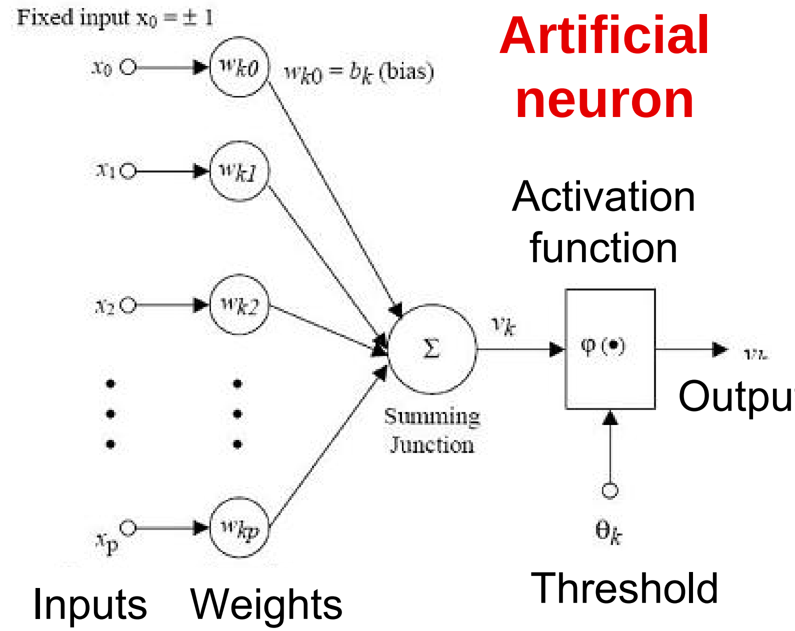



Fig. 8.13 Linear array of $\log_2 n$ cells for $n$-point FFT computation.

# Another Circuit Model: Artificial Neural Nets




**Feedforward network**

Three layers: input, hidden, output

No feedback

**Recurrent network**

Simple version due to Elman

Feedback from hidden nodes to special nodes at the input layer

**Hopfield network**

All connections are bidirectional

Characterized by connection topology and learning method

Diagrams from http://www.learnartificialneuralnetworks.com/