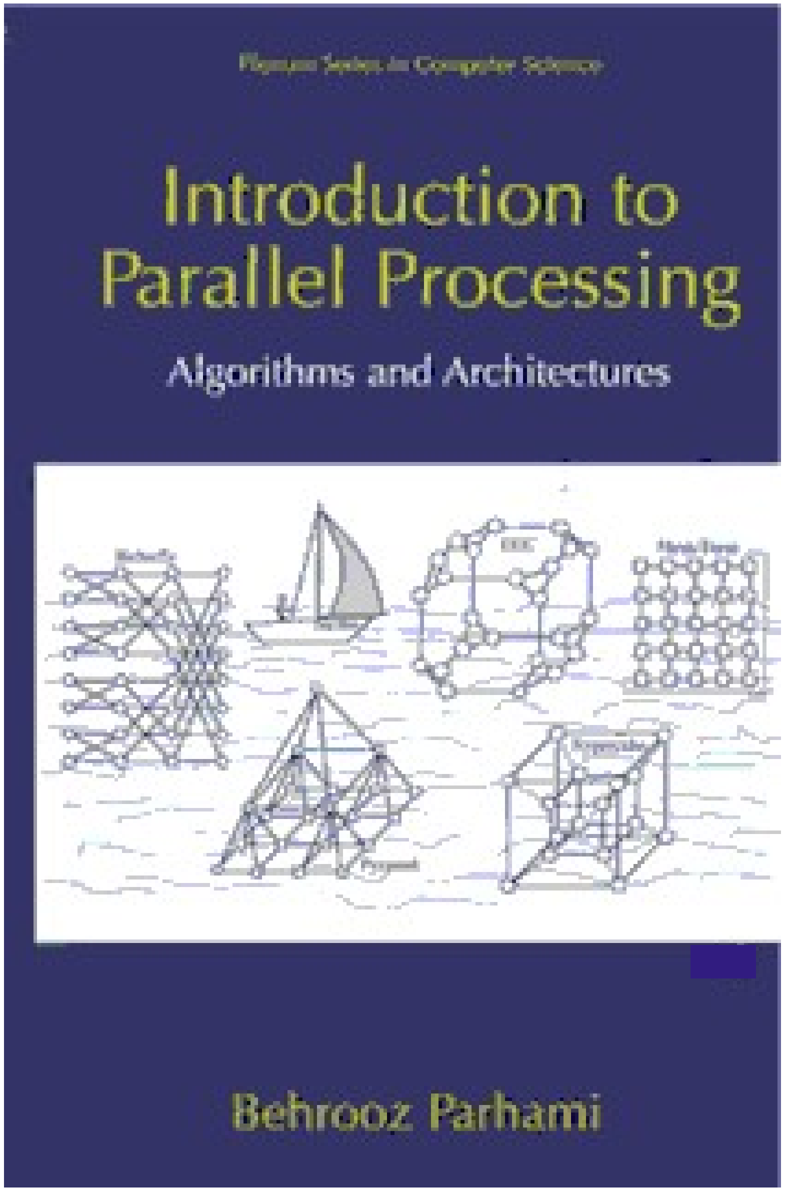


Part II′
Shared-Memory Parallelism

Part I: Fundamental Concepts	Background and Motivation	1. Introduction to Parallelism 2. A Taste of Parallel Algorithms
	Complexity and Models	3. Parallel Algorithm Complexity 4. Models of Parallel Processing
Part II′ Shared-Memory Parallelism	Shared-Memory Algorithms	5. PRAM and Basic Algorithms 6A. More Shared-Memory Algorithms
	Implementations and Models	6B. Implementation of Shared Memory 6C, Shared-Memory Abstractions
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

Architectural Variations

About This Presentation

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

II′ Shared-Memory Parallelism

Shared memory is the most intuitive parallel user interface:

- Abstract SM (PRAM); ignores implementation issues
- Implementation w/o worsening the memory bottleneck
- Shared-memory models and their performance impact

Topics in This Part	
Chapter 5	PRAM and Basic Algorithms
Chapter 6A	More Shared-Memory Algorithms
Chapter 6B	Implementation of Shared Memory
Chapter 6C	Shared-Memory Abstractions

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

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

Why Start with Shared Memory?

Study one extreme of parallel computation models:
- Abstract SM (PRAM); ignores implementation issues
- This abstract model is either realized or emulated
- In the latter case, benefits are similar to those of HLLs

In Part II″, we will study the other extreme case of models:
- Concrete circuit model; incorporates hardware details
- Allows explicit latency/area/energy trade-offs
- Facilitates theoretical studies of speed-up limits

Everything else falls between these two extremes

5.1 PRAM Submodels and Assumptions

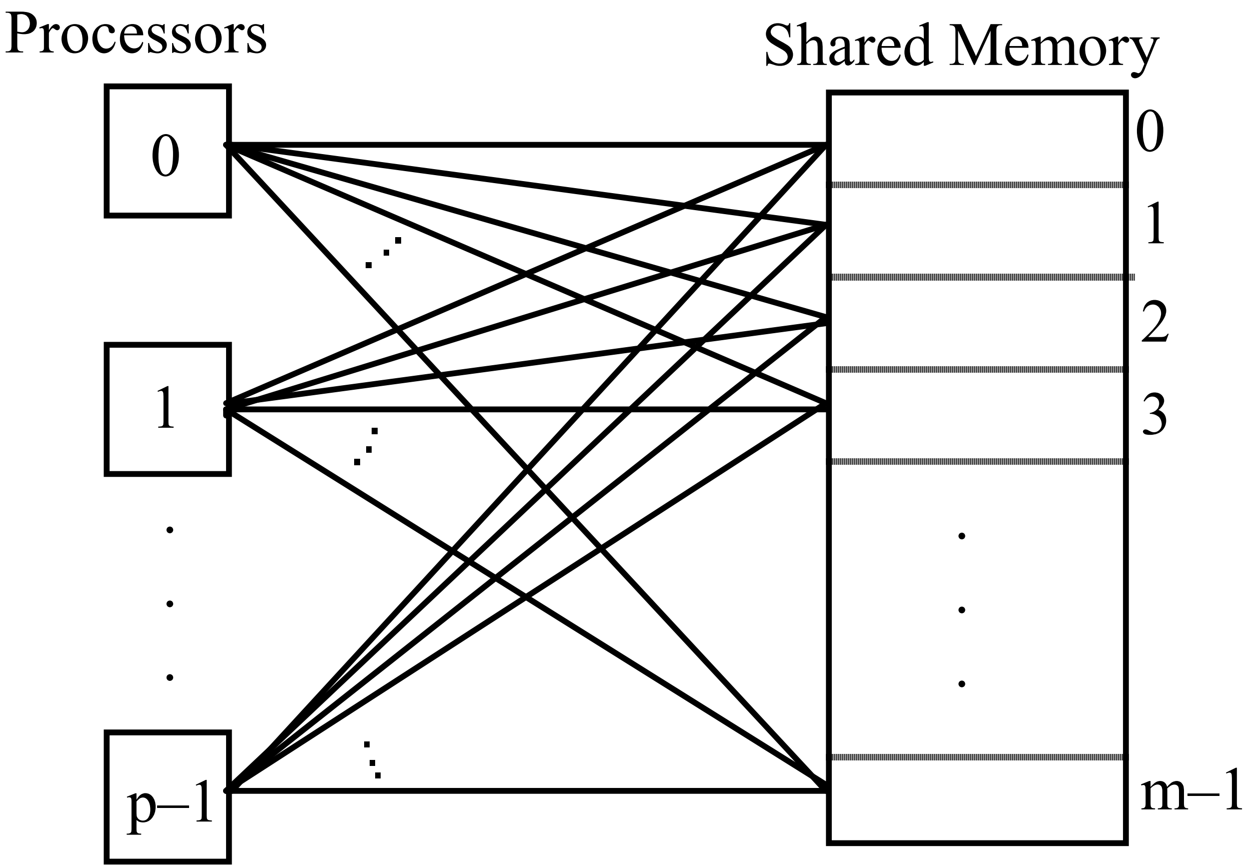


Fig. 4.6 Conceptual view of a parallel random-access machine (PRAM).

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** Least “powerful”, most “realistic”	**CREW** Default
Concurrent	**ERCW** Not useful	**CRCW** Most “powerful”, further subdivided

Fig. 5.1 Submodels of the PRAM model.

Examples of Exclusive/Concurrent Reads/Writes

Exclusivity not enforced by hardware; rather, it's done by the programmer

Exclusive read:

for $0 \le i < p$ processor i read from location i

Exclusive write:

for $0 \le i < p$ processor i write into location $i + 1 \bmod p$

Concurrent read:

for $0 \le i < p$ processor i read from location $i \bmod 2$

Concurrent write:

for $0 \le i < p$ processor i write into location d_i

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 in ascending order of addresses
Remove all duplicate addresses
Access data at desired addresses
Replicate data via parallel prefix computation

Each step requires constant or O(log *p*) time

Proc Addr		
0,1	0,1	0,1
1,6	6,1	
2,5	7,1	
3,2	3,2	3,2
4,3	8,2	
5,6	4,3	4,3
6,1	2,5	2,5
7,1	1,6	1,6
8,2	5,6	

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

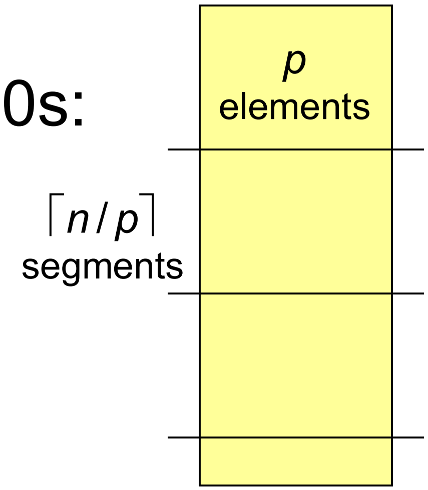
We need not be too concerned with the CRCW submodel used

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
    $h = jp + i$
    if $h < n$ then $M[B + h] := 0$
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

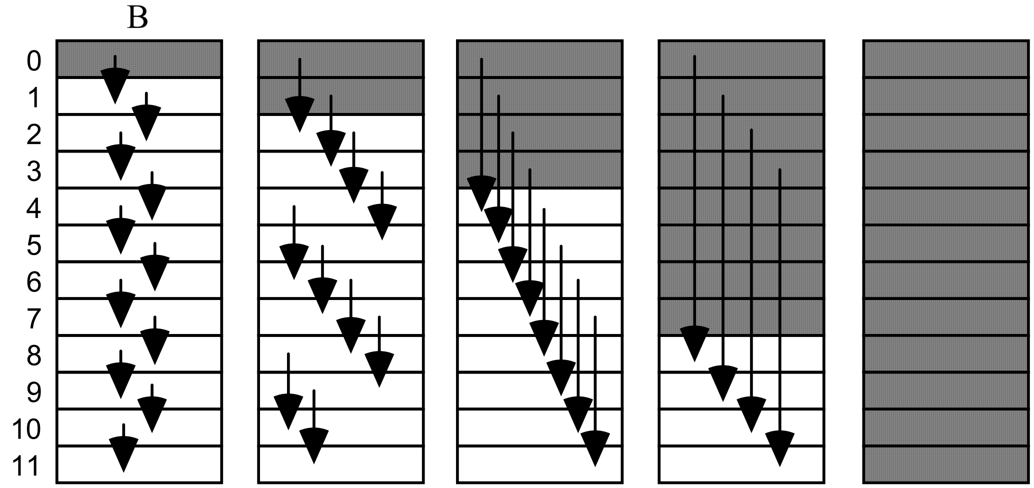


Fig. 5.2 Data broadcasting in EREW PRAM via recursive doubling.

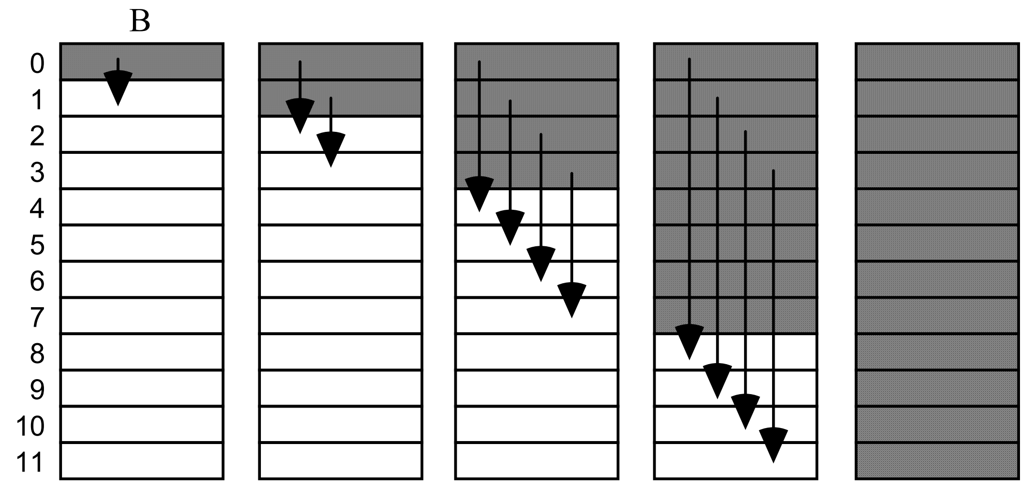


Fig. 5.3 EREW PRAM data broadcasting without redundant copying.

Class Participation: Broadcast-Based Sorting

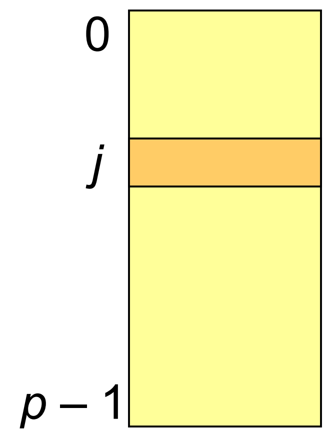
Each person write down an arbitrary nonnegative integer with 3 or fewer digits on a piece of paper

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

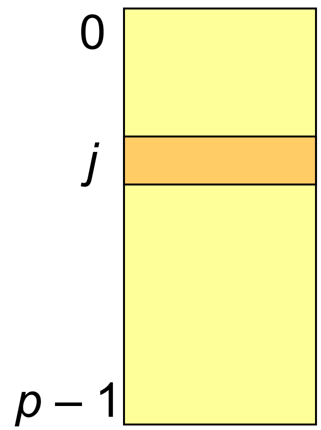
Students call out their numbers in order of the computed rank



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

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 proc's

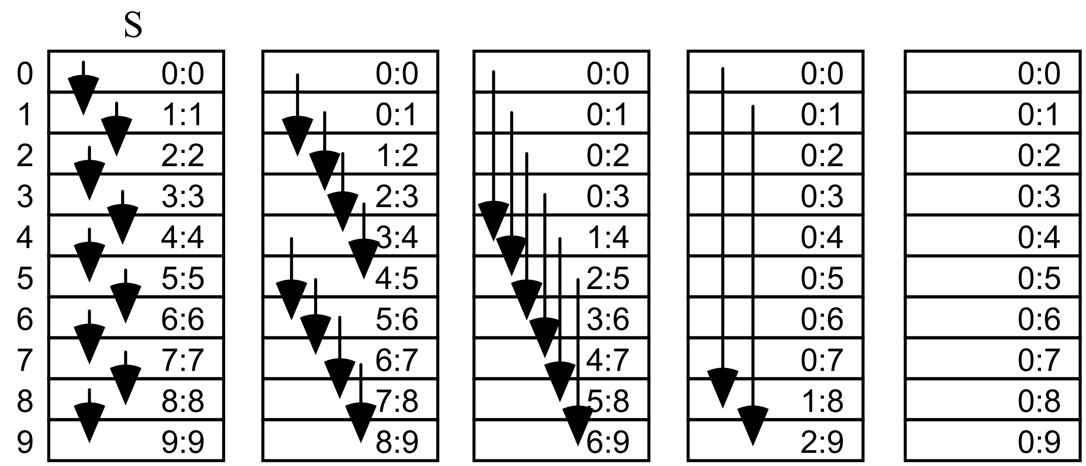


Fig. 5.4 Semigroup computation in EREW PRAM.

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

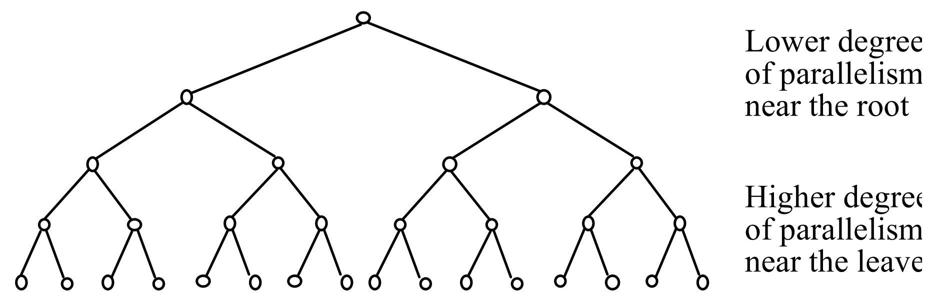


Fig. 5.5 Intuitive justification of why parallel slack helps improve the efficiency.

5.4 Parallel Prefix Computation

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

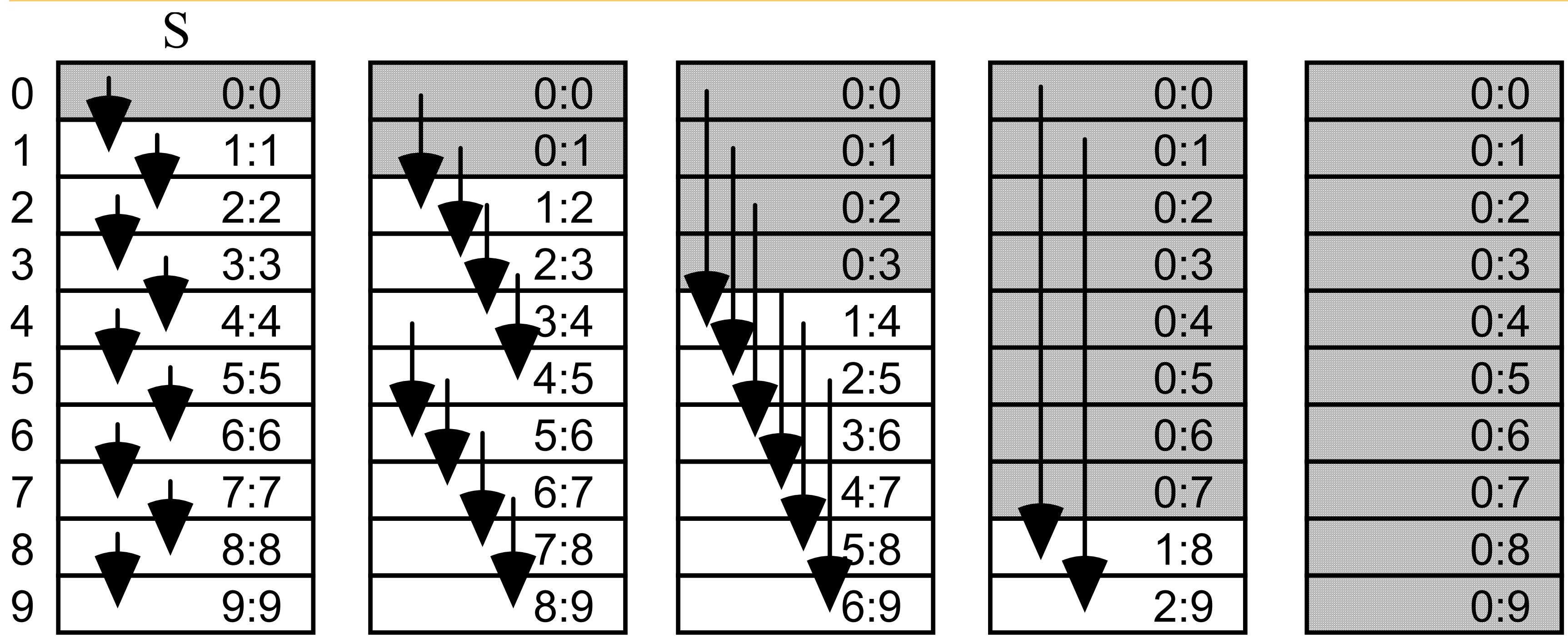
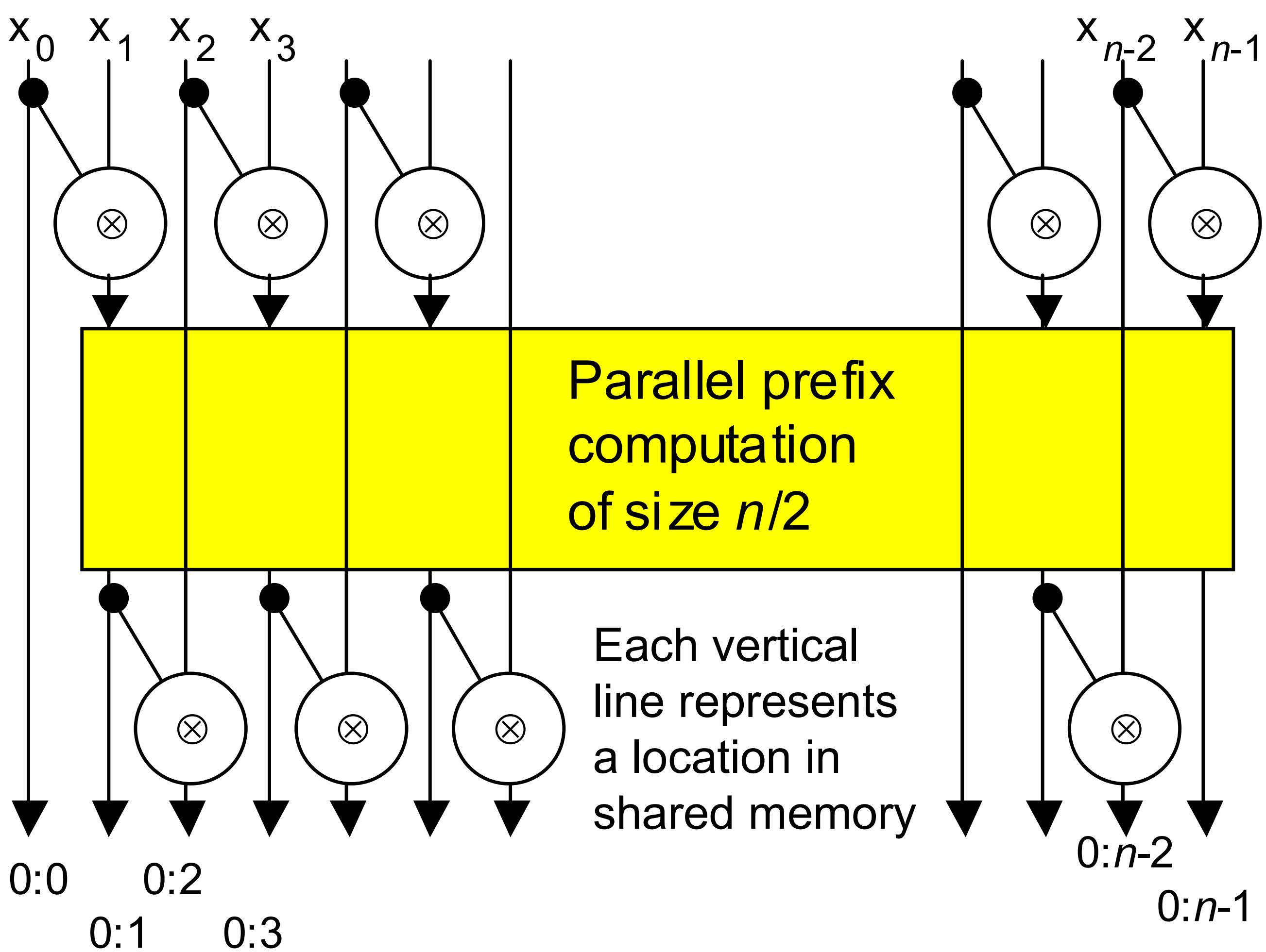


Fig. 5.6 Parallel prefix computation in EREW PRAM via recursive doubling.

A Divide-and-Conquer Parallel-Prefix Algorithm



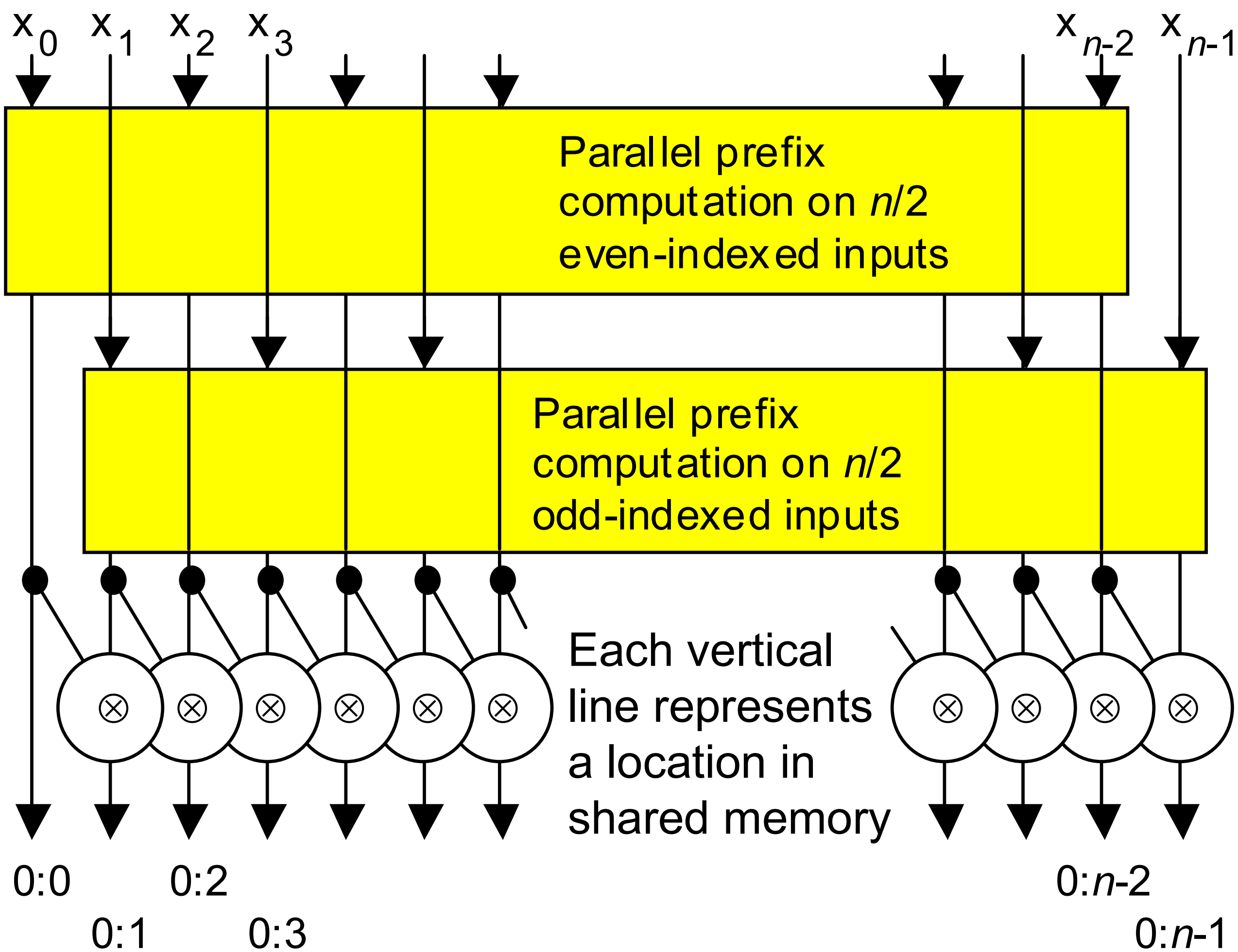
$T(p) = T(p/2) + 2$

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

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

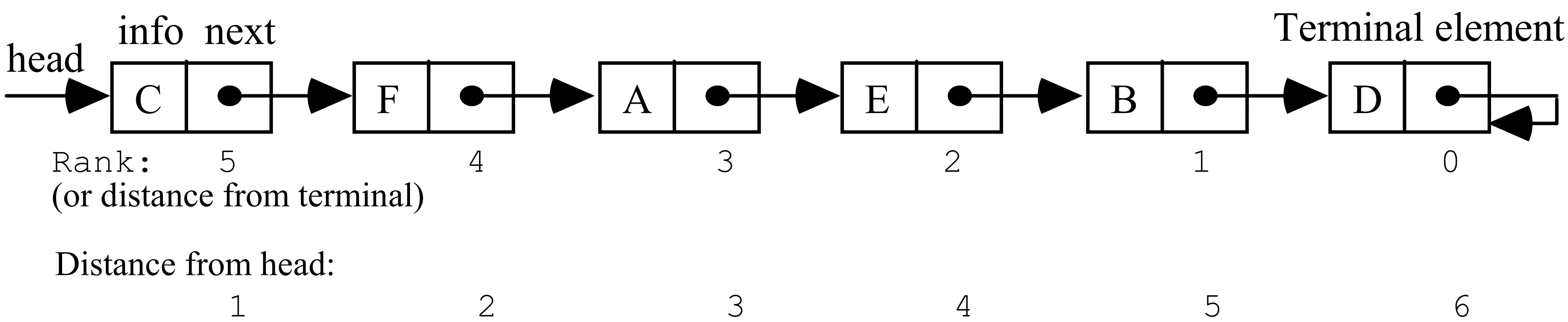


Fig. 5.9 Example linked list and the ranks of its elements.

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

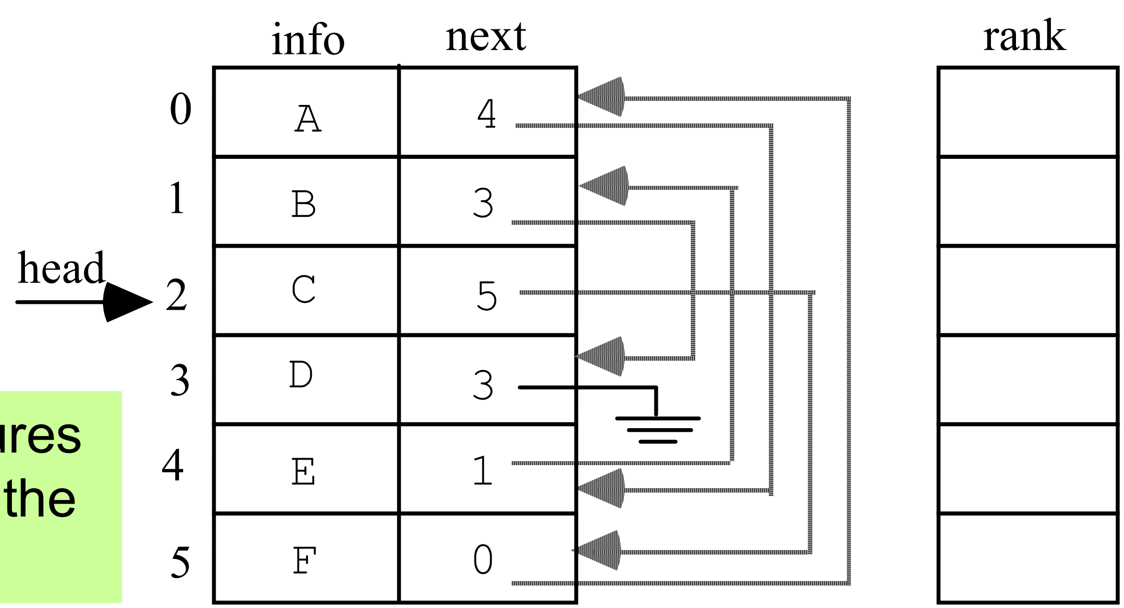
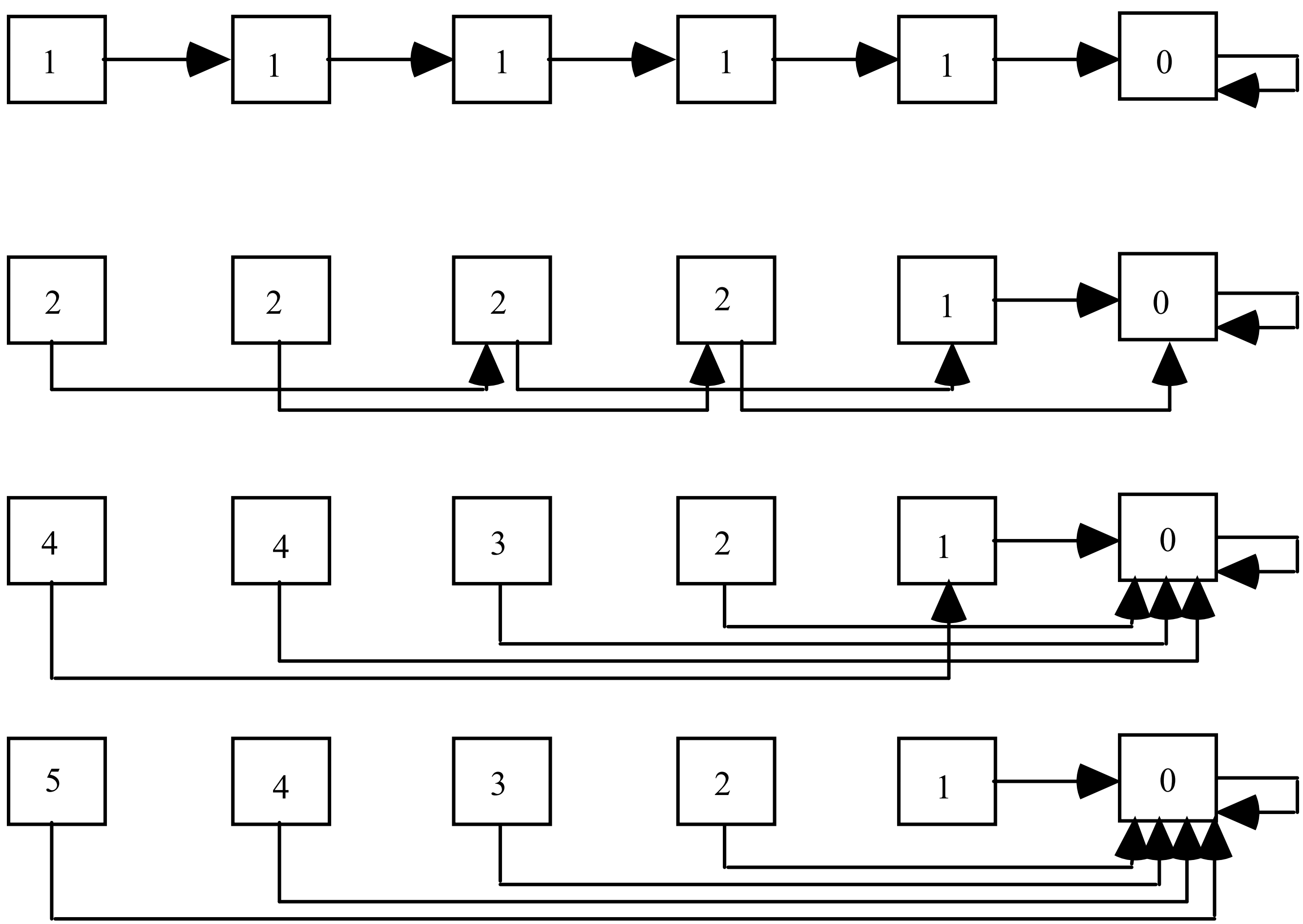


Fig. 5.10 PRAM data structures representing a linked list and the ranking results.

List Ranking via Recursive Doubling



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

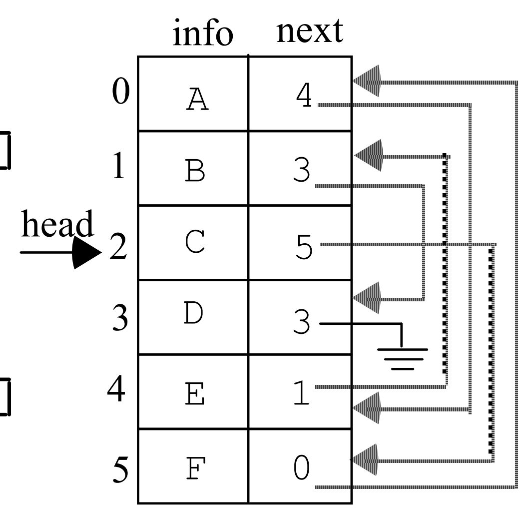


Fig. 5.11 Element ranks initially and after each of the three iterations.

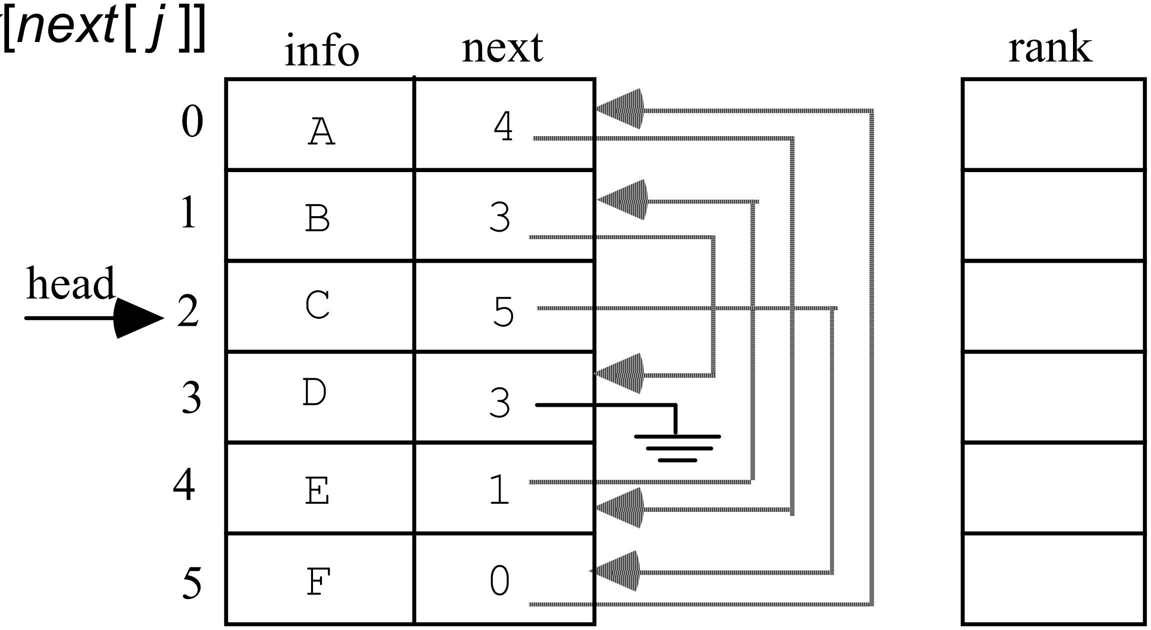
PRAM List Ranking Algorithm

<u>PRAM list ranking algorithm (via pointer jumping)</u>
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 $rank[next[head]] \ne 0$ 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

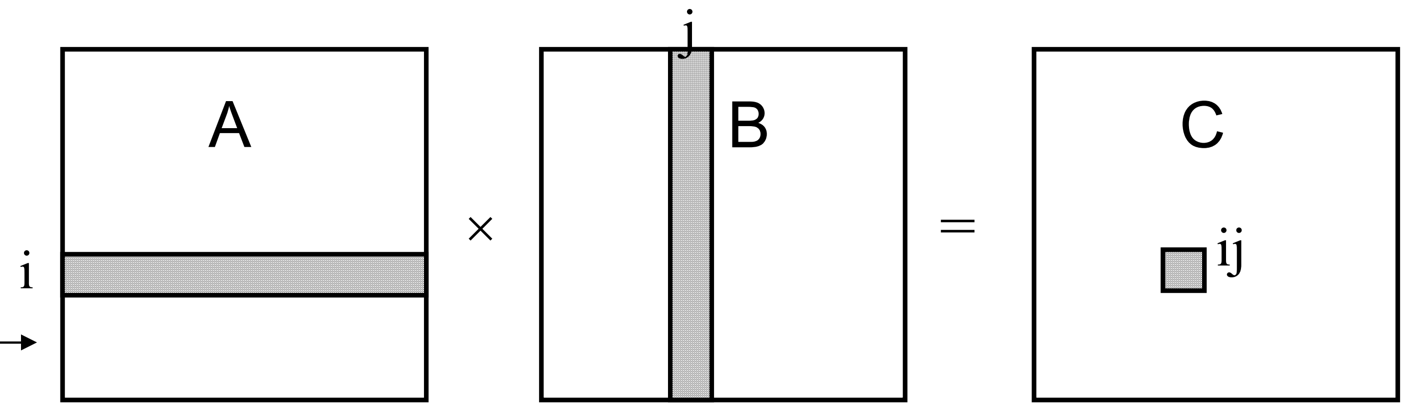
<u>Sequential matrix multiplication</u>

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

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), $0 \le i, j < m$, 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

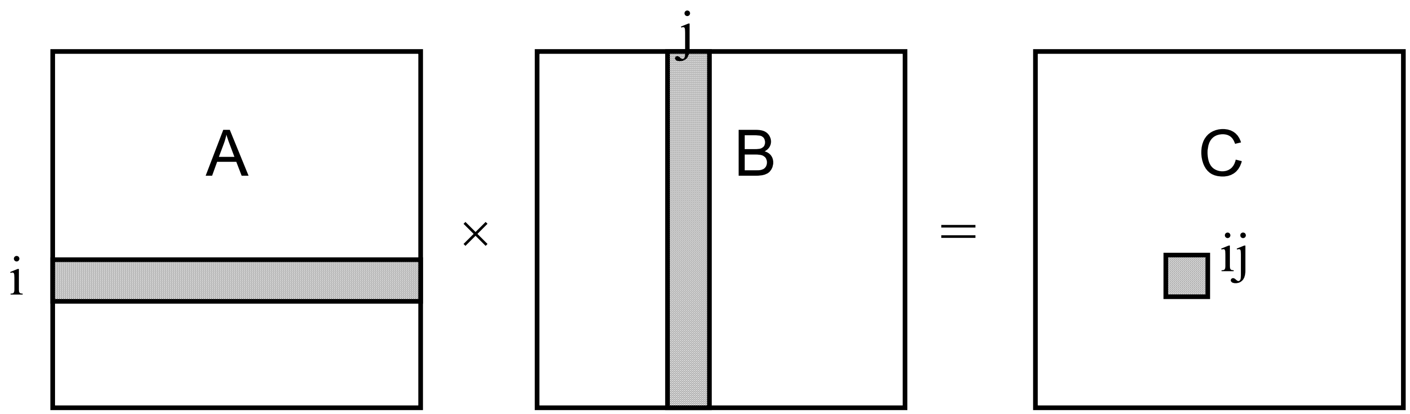


Fig. 5.12 PRAM matrix multiplication; $p = m^2$ processors.

PRAM Matrix Multiplication with m Processors

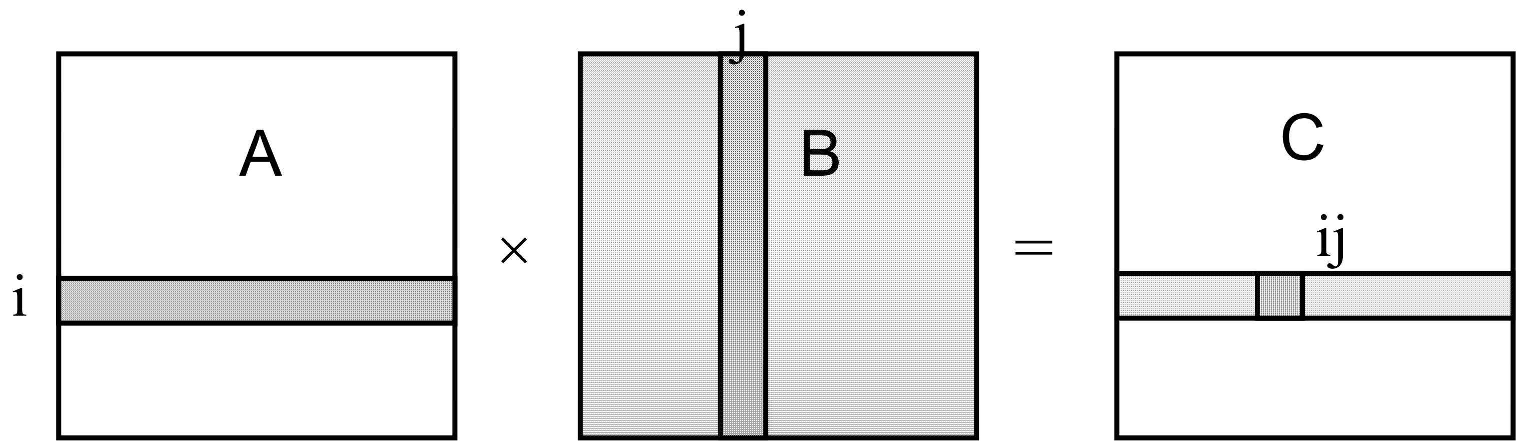
PRAM matrix multiplication using m processors

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

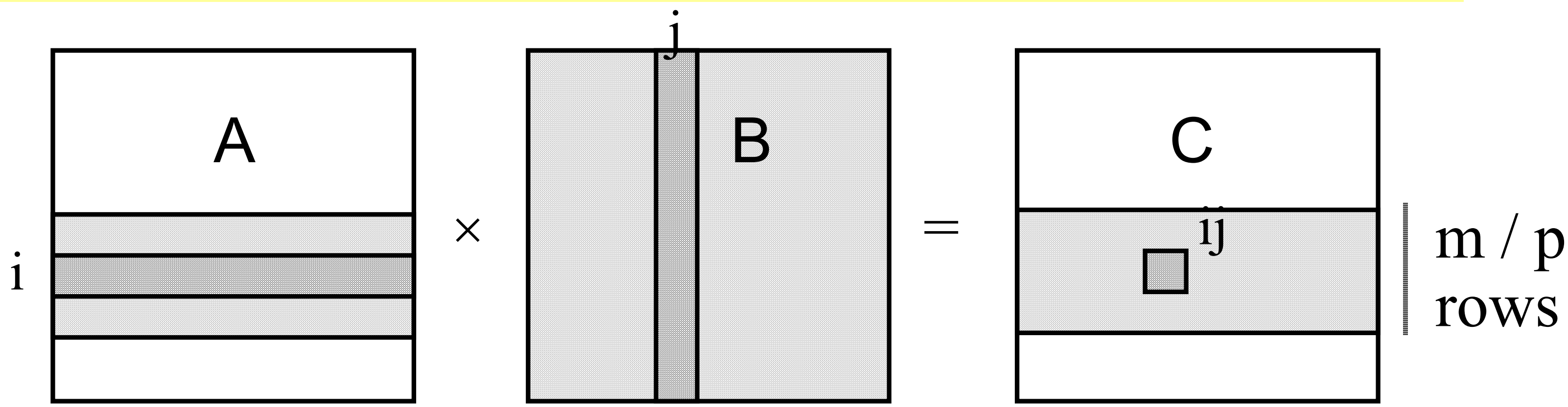
Algorithm is similar, except that each processor is in charge of computing m / p rows of C

$\Theta(m^3/p)$ 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.

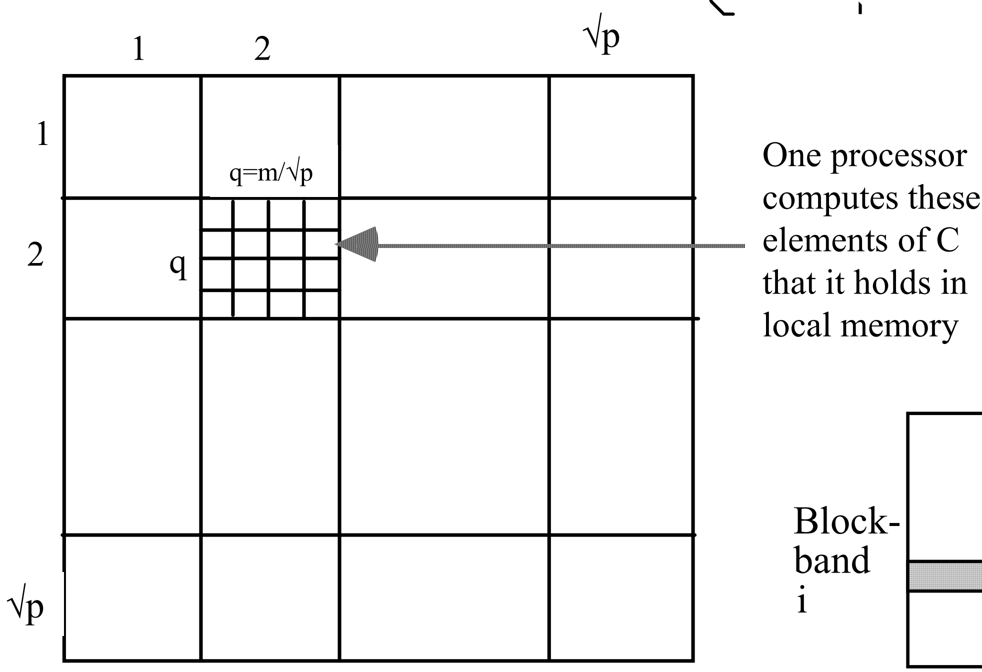
This is particularly costly for NUMA shared-memory machines.



More Efficient Matrix Multiplication (for NUMA)

Partition the matrices into *p* square blocks

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$



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

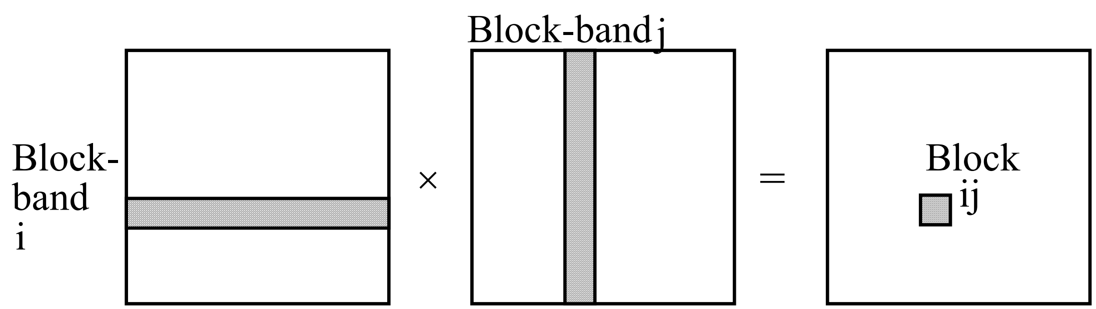
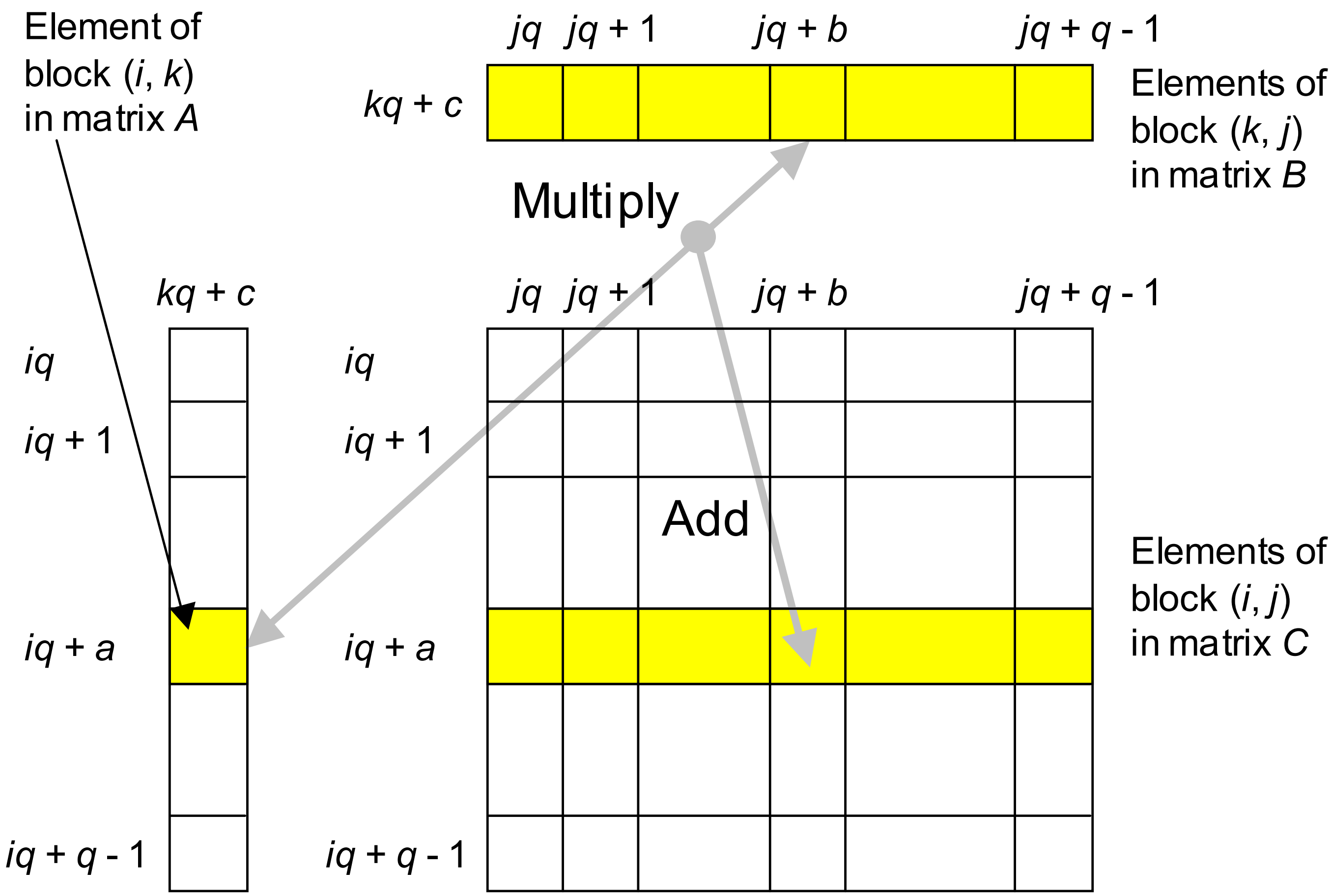


Fig. 5.13 Partitioning the matrices for block matrix multiplication .

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.

6A More Shared-Memory Algorithms

Develop PRAM algorithm for more complex problems:

- Searching, selection, sorting, other nonnumerical tasks
- Must present background on the problem in some cases

Topics in This Chapter
6A.1 Parallel Searching Algorithms
6A.2 Sequential Rank-Based Selection
6A.3 A Parallel Selection Algorithm
6A.4 A Selection-Based Sorting Algorithm
6A.5 Alternative Sorting Algorithms
6A.6 Convex Hull of a 2D Point Set

6A.1 Parallel Searching Algorithms

Searching an unordered list in PRAM

Sequential time: n worst-case
$n/2$ on average

Divide the list of n items into p segments of $\lceil n / p \rceil$ items (last segment may have fewer)

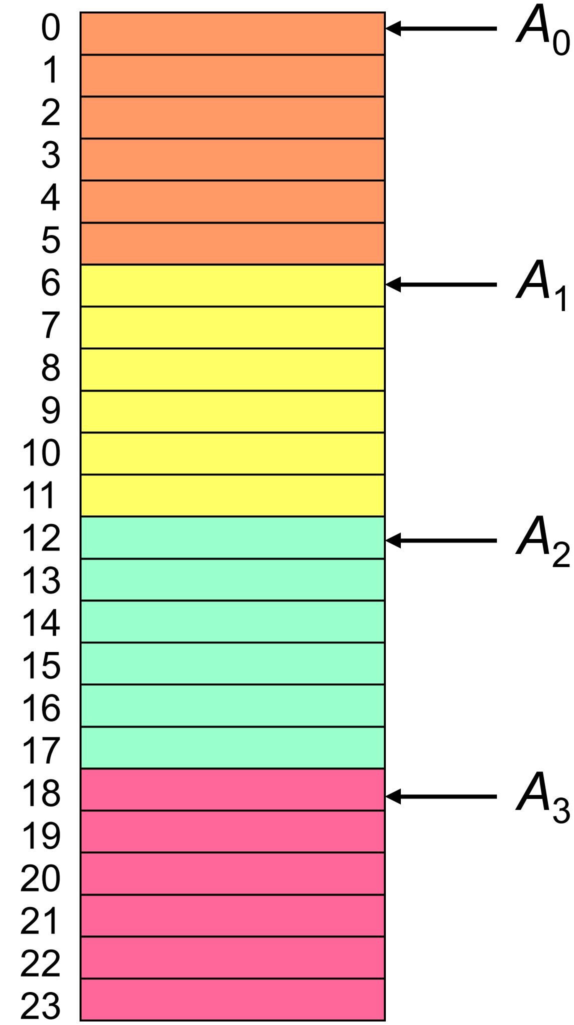
Processor i will be in charge of $\lceil n / p \rceil$ list elements, beginning at address $i \lceil n / p \rceil$

Parallel time: $\lceil n / p \rceil$ worst-case
??? on average

Perfect speed-up of p with p processors?

Pre- and postprocessing overheads

Example: $n = 24$, $p = 4$



Parallel $(p + 1)$-ary Search on PRAM

From Sec. 8.1

p probes, rather than 1, per step

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

Speedup $\cong \log p$

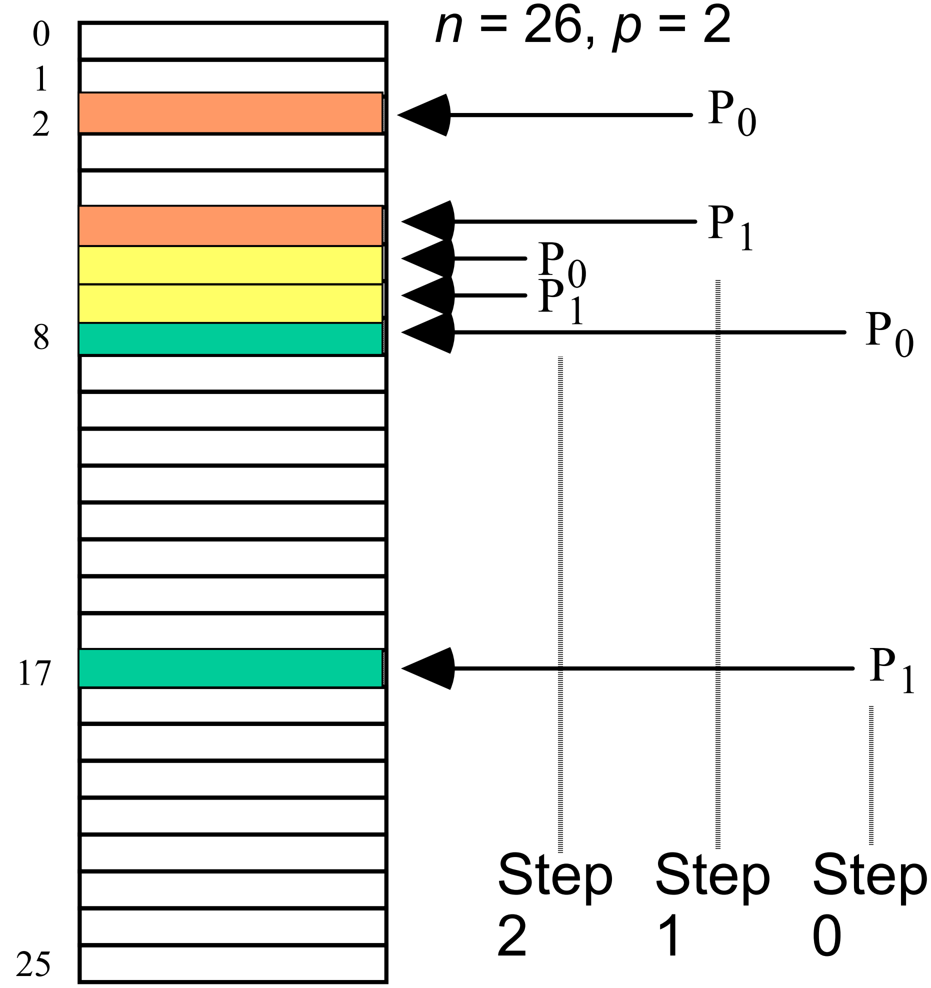
Optimal: no comparison-based search algorithm can be faster

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



6A.2 Sequential Ranked-Based Selection

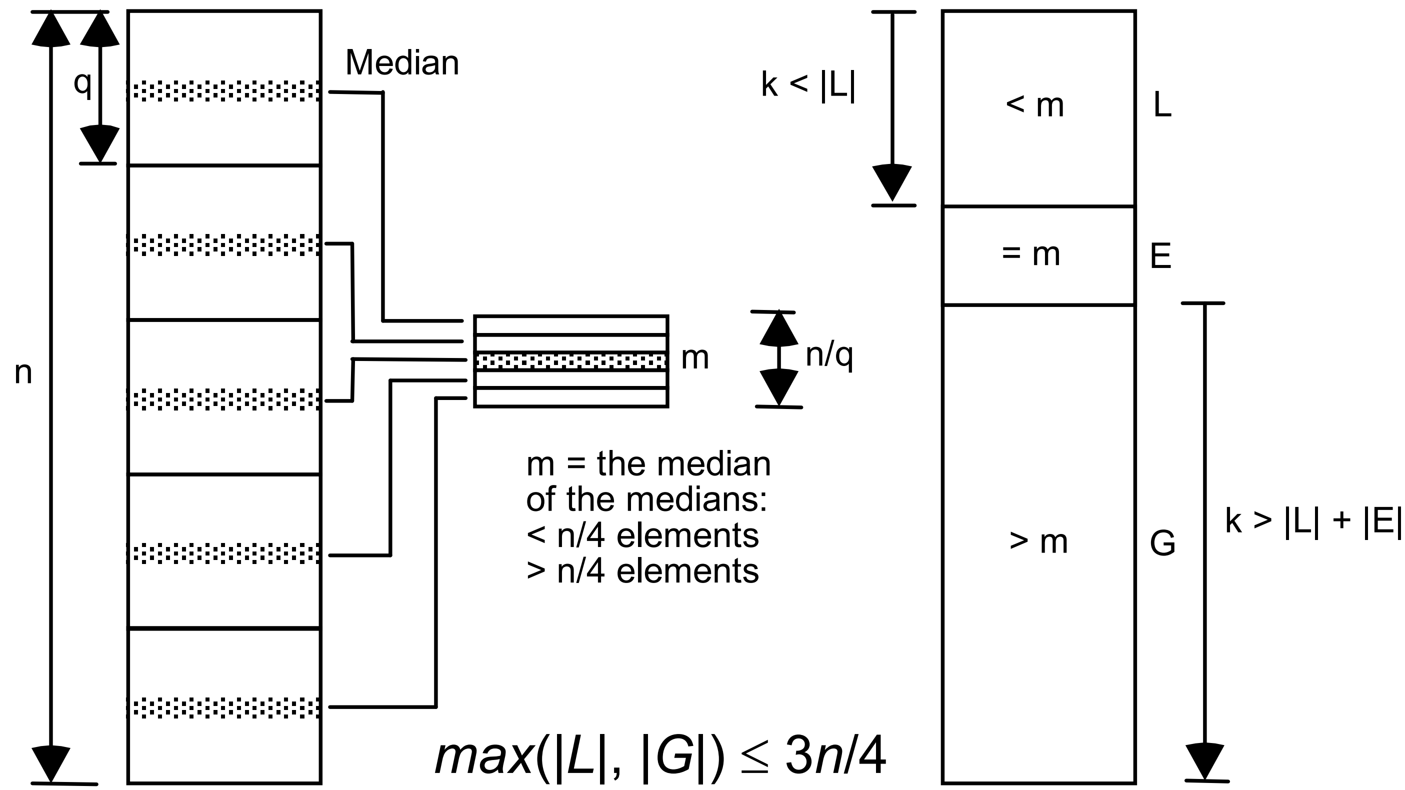
Selection: Find the (or a) *k*th smallest among *n* elements

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

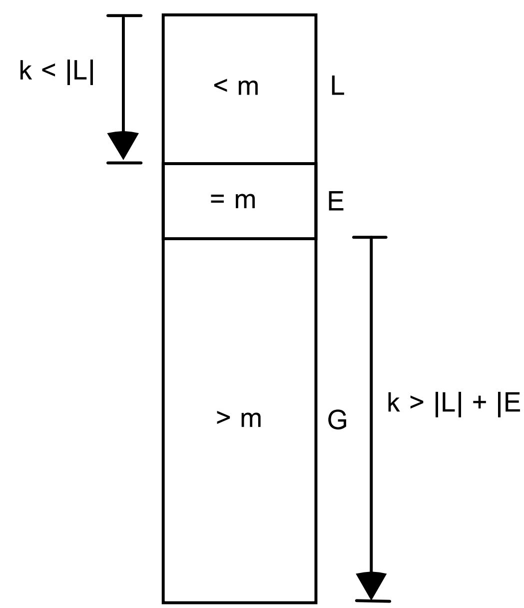


$max(|L|, |G|) \leq 3n/4$

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 kth 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. $m = select(T, |T|/2)$ {find the median m of the $|S|/q$ medians} — $T(n/q)$
3. Create 3 subsequences — $O(n)$ (To be justified)
   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

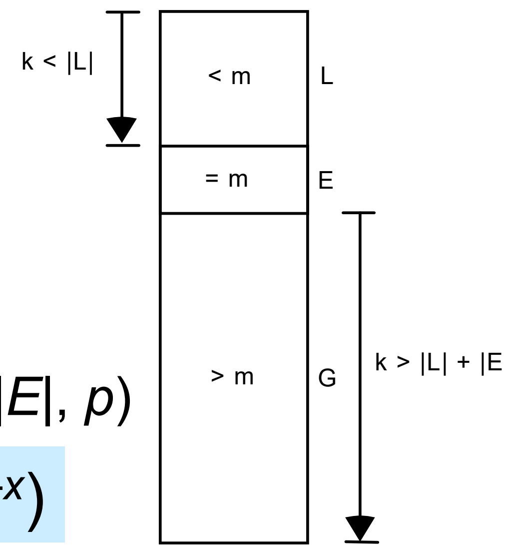
The 9th smallest element of *S* is 3.

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

6A.3 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 kth smallest element of S
   else broadcast $|S|$ to all p processors — $O(n^x)$
   divide S into $|S|/q$ subsequences $S(j)$ of size q
   Processor j, $0 \le j < p$, compute $T_j := select(S(j), |S(j)|/2)$
   endif
2. $m = PRAMselect(T, |T|/2, p)$ {median of the medians} — $T(n^{1-x}, p)$
3. Broadcast m to all processors and create 3 subsequences — $O(n^x)$
   L: Elements of S that are $< m$
   E: Elements of S that are $= m$
   G: Elements of S that are $> m$
4. if $|L| \ge k$
   then return $PRAMselect(L, k, p)$
   else if $|L| + |E| \ge k$
   then return m
   else return $PRAMselect(G, k - |L| - |E|, p)$ — $T(3n/4, p)$
   endif
   endif



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)

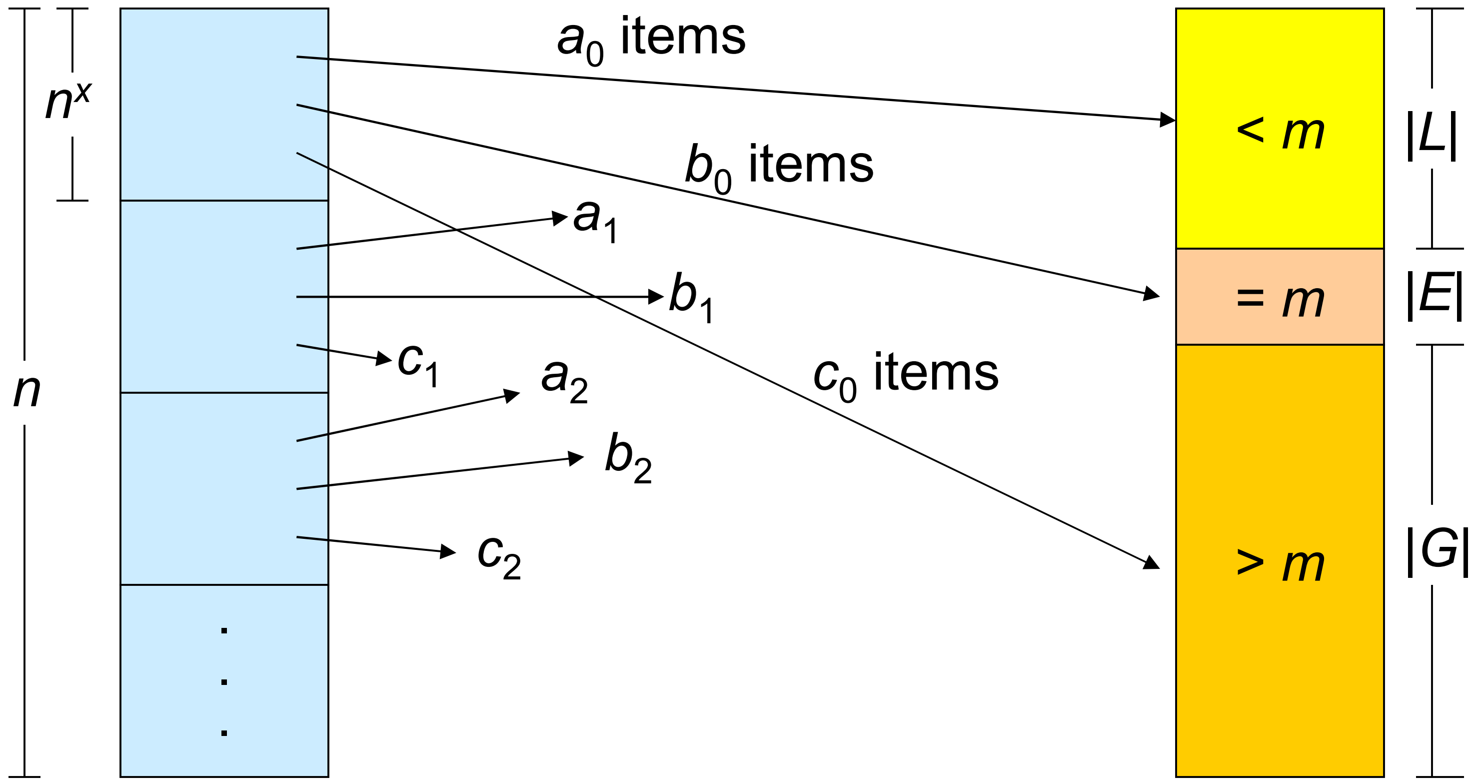
$\boldsymbol{T(n, 1) = O(n^x) = O(n)}$

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

$\boldsymbol{T(n, n) = O(n^x) = O(1)}$ **?**

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

Data Movement in Step 2 of the Algorithm



Consider the sublist L: Processor i contributes a_i items to this sublist

Processor 0 starts storing at location 0, processor 1 at location a_0, processor 2 at location $a_0 + a_1$, Processor 3 at location $a_0 + a_1 + a_2$, …

6A.4 A Selection-Based Sorting Algorithm

<u>Parallel selection-based sort *PRAMselectionsort*(S, p)</u>

$O(1)$ 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

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

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

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

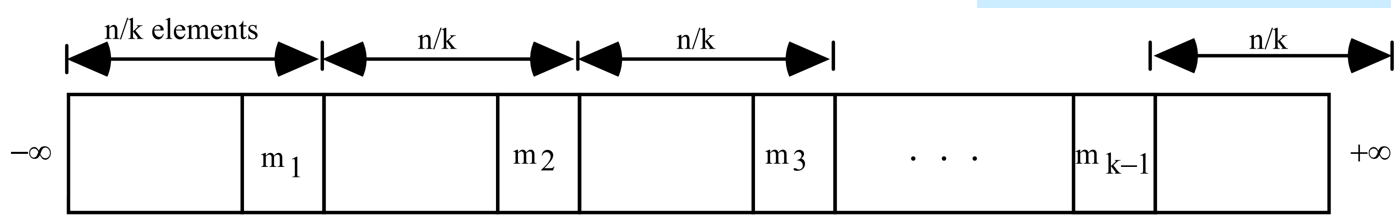


Fig. 6.1 Partitioning of the sorted list for selection-based sorting.

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):

$m_0 = -\infty$

$n/k = 25/4 \cong 6$ $\quad m_1 = PRAMselect(S, 6, 5) = 2$

$2n/k = 50/4 \cong 13$ $\quad m_2 = PRAMselect(S, 13, 5) = 4$

$3n/k = 75/4 \cong 19$ $\quad m_3 = PRAMselect(S, 19, 5) = 6$

$m_4 = +\infty$

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

6A.5 Alternative Sorting Algorithms

Sorting via random sampling (assume $p \ll \sqrt{n}$)

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

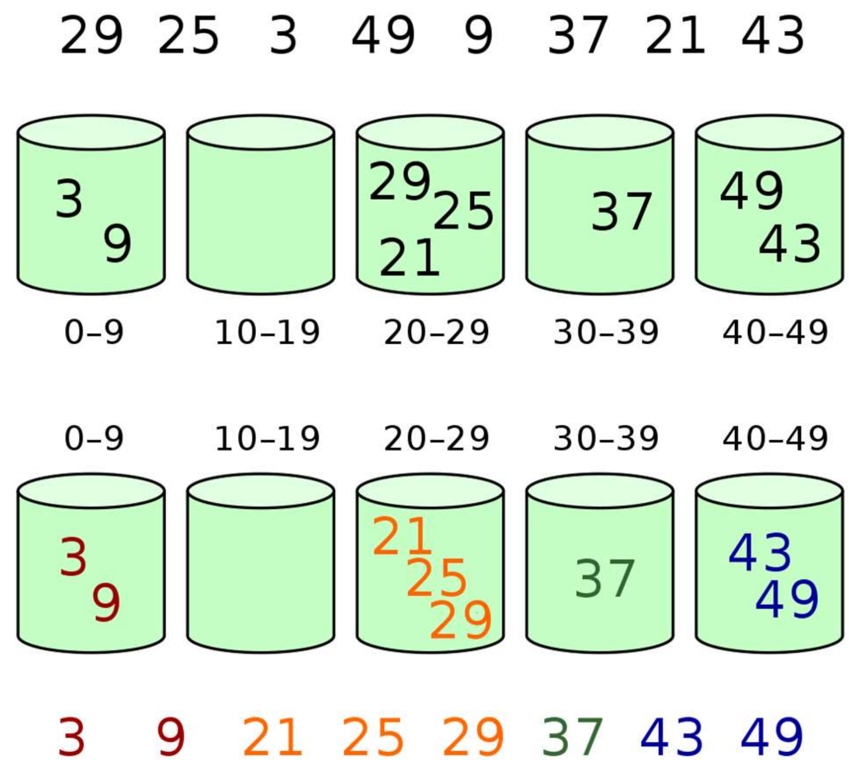
It is easier if we pick $k = p$, so that each of the resulting subproblems is handled by a single processor

<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 Binsort or Bucketsort

Wikipedia images



Suppose input values are in [0, *m*)

Divide the range into *p* subranges

[0, *m*/*p*), [*m*/*p*, 2*m*/*p*), ….

Processor *j* sorts the elements in the *j*th subrange

If values are uniformly distributed, each processor gets ~*n*/*p* values

Avg. time = $\underbrace{O(n/p)}_{\text{Fill buckets}} + \underbrace{O((n/p)\log(n/p))}_{\text{Sort buckets}}$

If the list consists of small range of values (say, numbers 0-9), bucketsort can be used to count the number of occurrences of each value in $O(n)$ time serially and in $O(n/p)$ time in parallel

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

6A.6 Convex Hull of a 2D Point Set

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

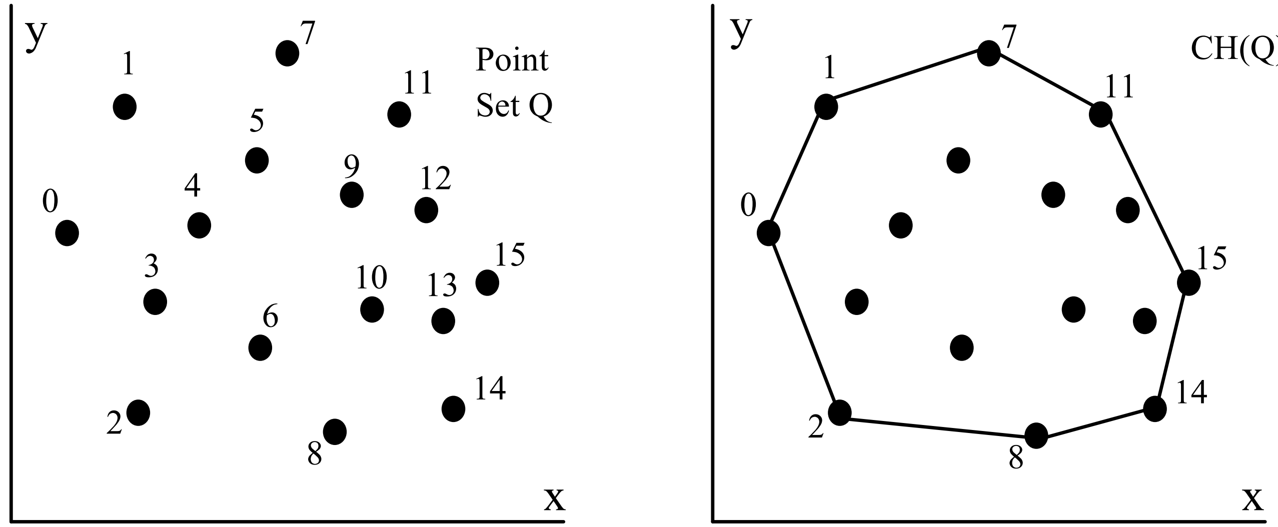


Fig. 6.2 Defining the convex hull problem.

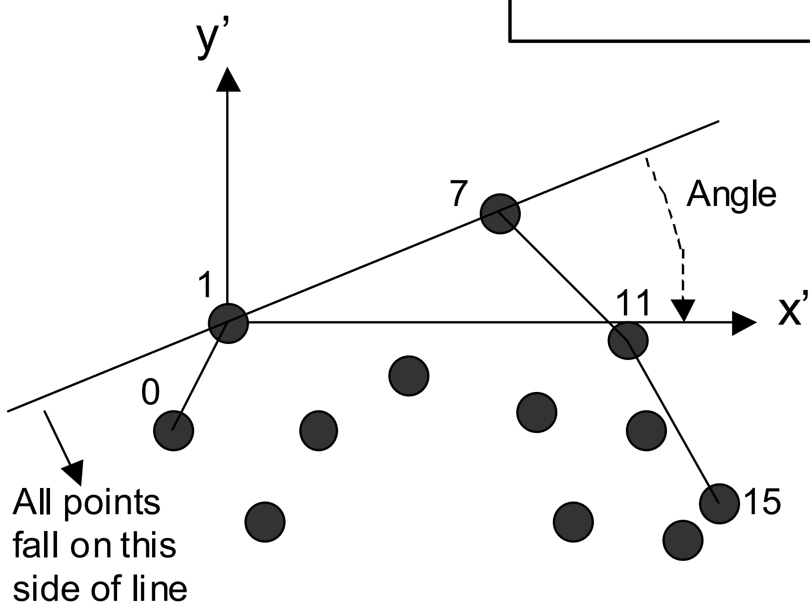


Fig. 6.3 Illustrating the properties of the convex hull.

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)

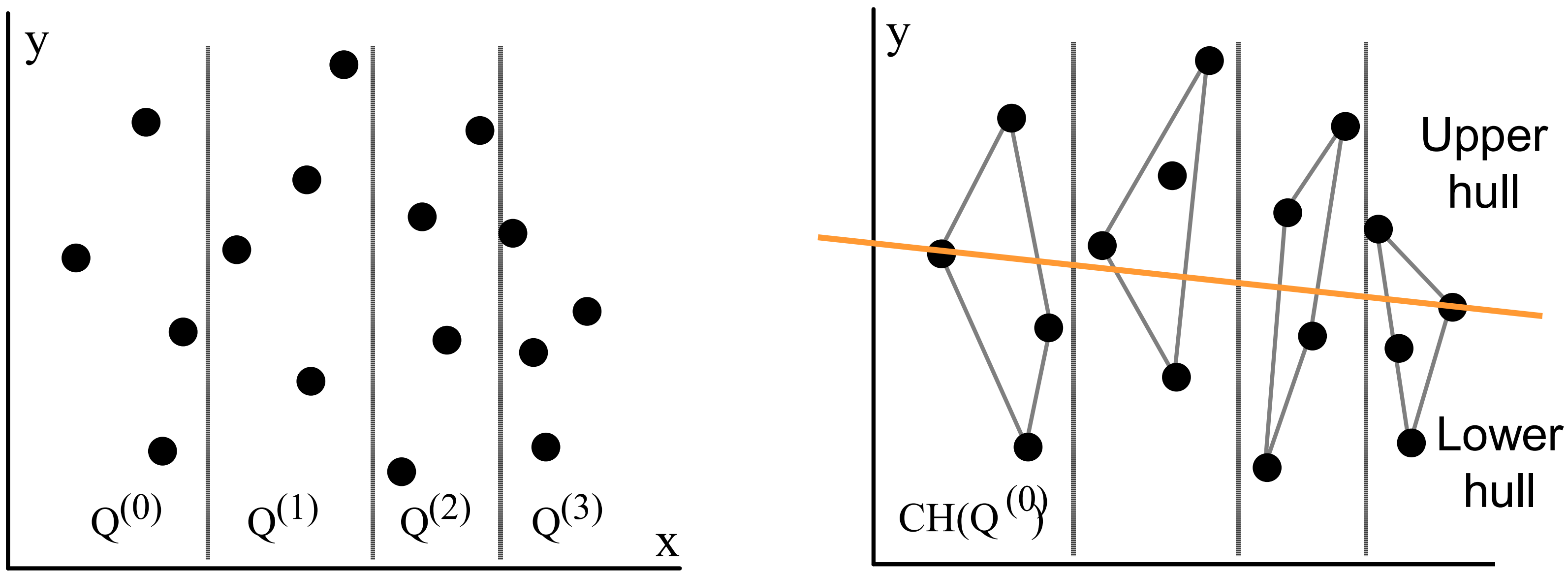
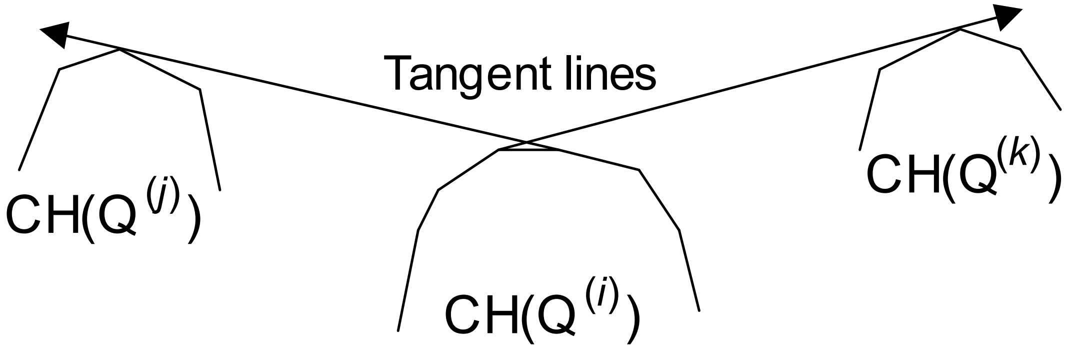
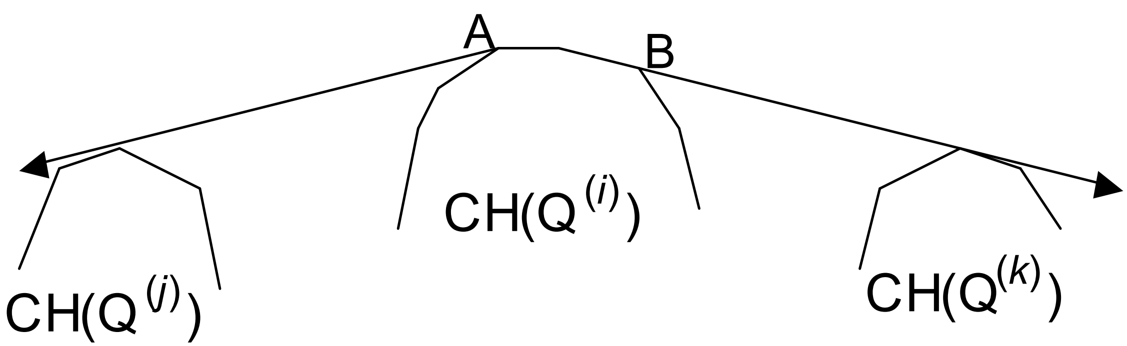


Fig. 6.4 Multiway divide and conquer for the convex hull problem

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.

6B Implementation of Shared Memory

Main challenge: Easing the memory access bottleneck

- Providing low-latency, high-bandwidth paths to memory
- Reducing the need for access to nonlocal memory
- Reducing conflicts and sensitivity to memory latency

About the New Chapter 6B

This new chapter incorporates material from the following existing sections of the book:

6.6 Some Implementation Aspects

14.4 Dimension-Order Routing

15.3 Plus-or-Minus-2^i Network

16.6 Multistage Interconnection Networks

17.2 Distributed Shared Memory

18.1 Data Access Problems and Caching

18.2 Cache Coherence Protocols

18.3 Multithreading and Latency Hiding

20.1 Coordination and Synchronization

Making PRAM Practical

PRAM needs a read and a write access to memory in every cycle

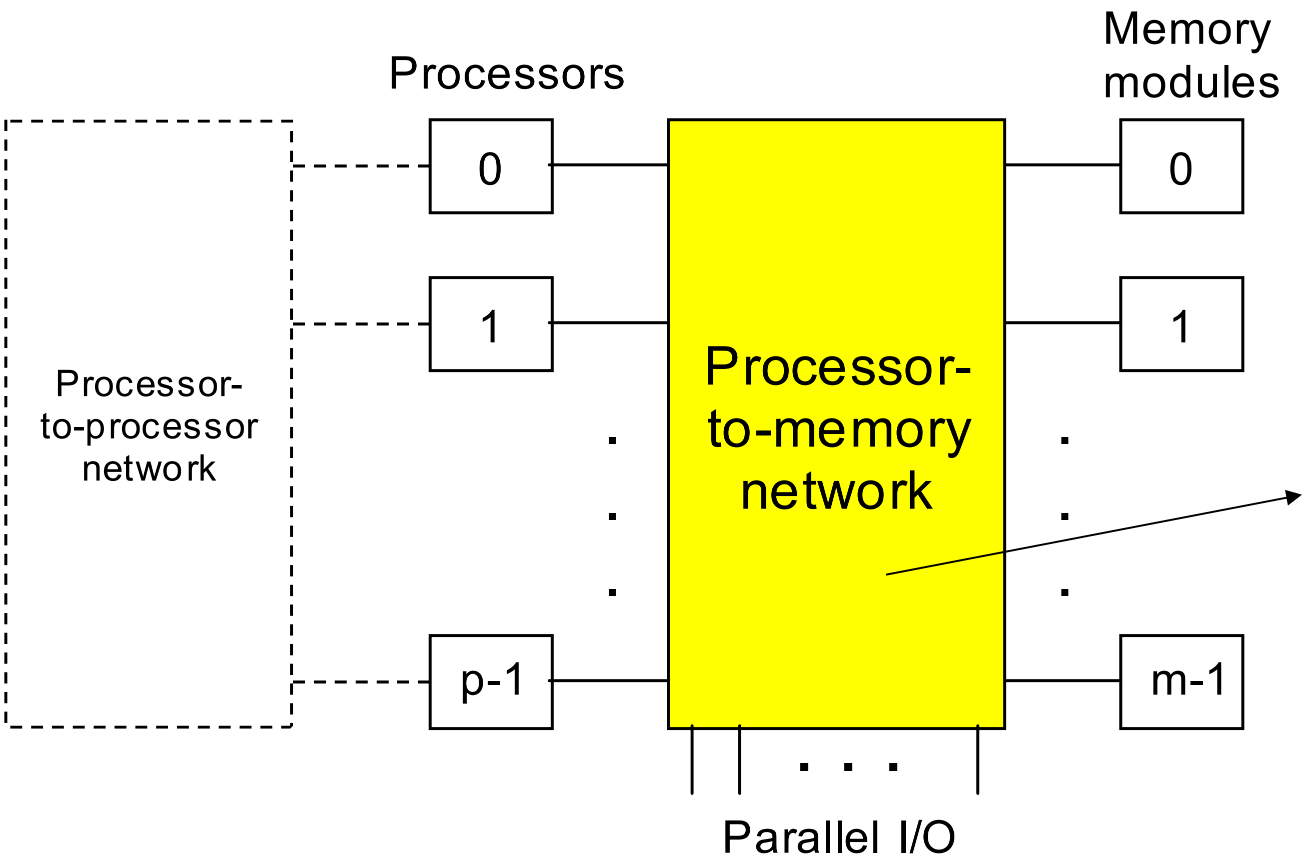
Even for a sequential computer, memory is tens to hundreds of times slower than arithmetic/logic operations; multiple processors accessing a shared memory only makes the situation worse

Shared access to a single large physical memory isn't scalable

Strategies and focal points for making PRAM practical

1. Make memory accesses faster and more efficient (pipelining)
2. Reduce the number of memory accesses (caching, reordering of accesses so as to do more computation per item fetched/stored)
3. Reduce synchronization, so that slow memory accesses for one computation do not slow down others (synch, memory consistency)
4. Distribute the memory and data to make most accesses local
5. Store data structures to reduce access conflicts (skewed storage)

6B.1 Processor-Memory Interconnection

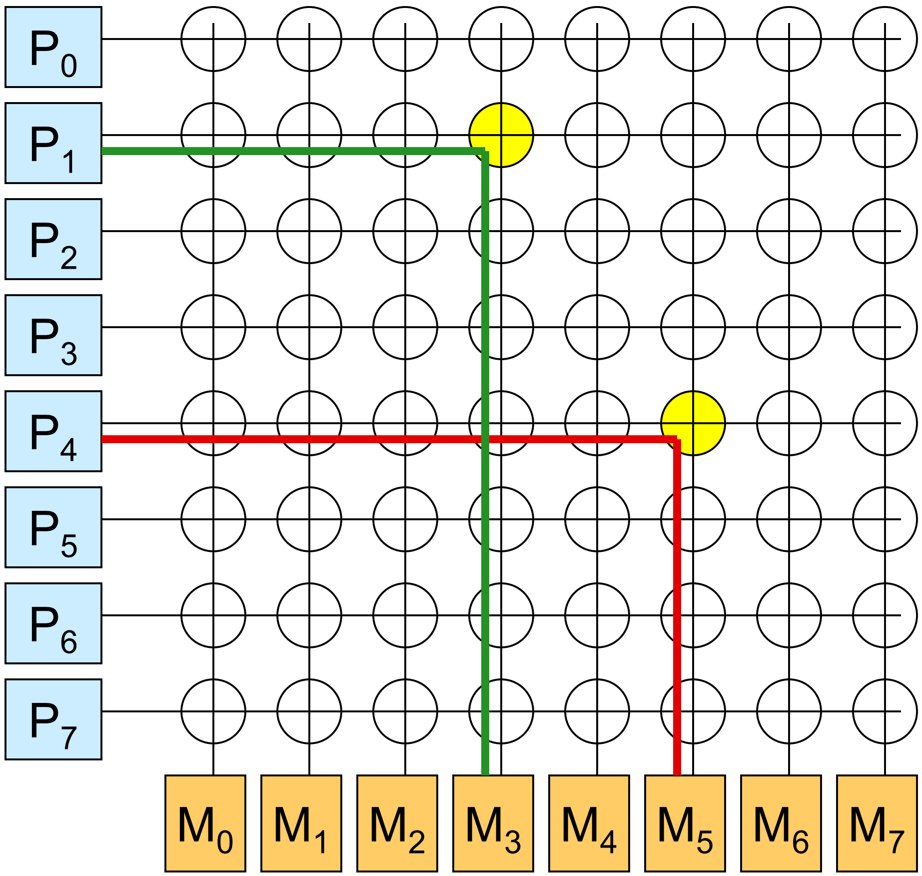


Options:
Crossbar
Bus(es)
MIN

Bottleneck
Complex
Expensive

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

Processor-to-Memory Network



An 8×8 crossbar switch

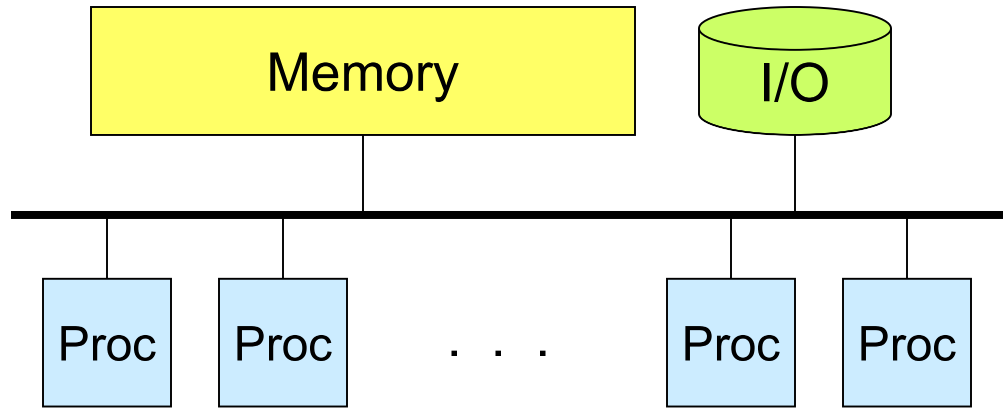
Crossbar switches offer full permutation capability (they are *nonblocking*), but are complex and expensive: $O(p^2)$

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

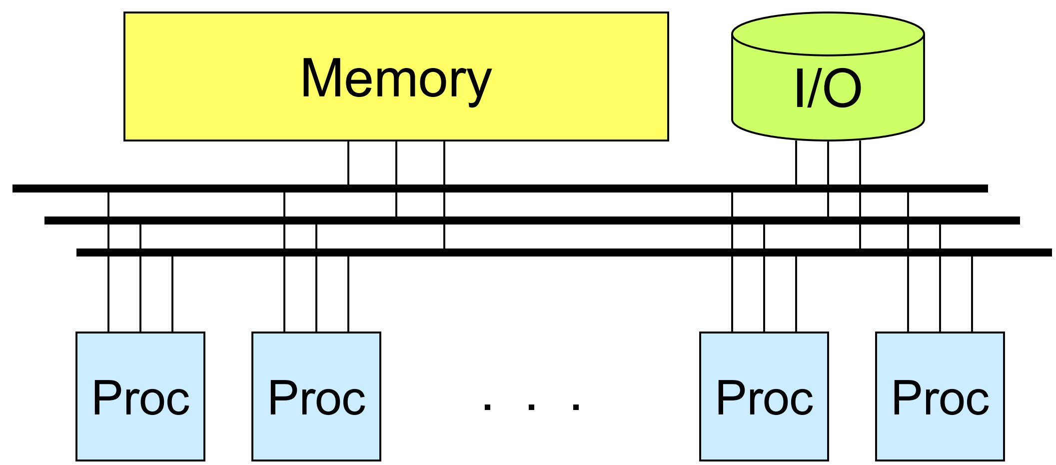
Practical processor-to-memory networks cannot realize all permutations (they are *blocking*)

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

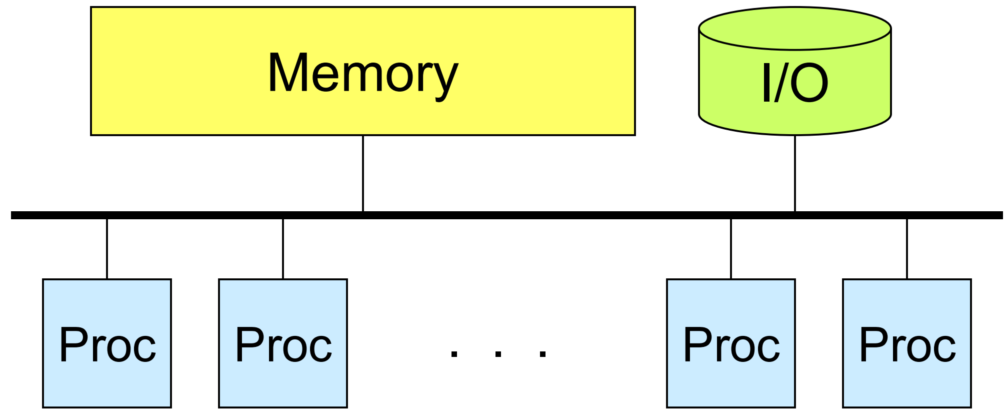


Back-of-the-Envelope Bus Bandwidth Calculation

Single-bus system:

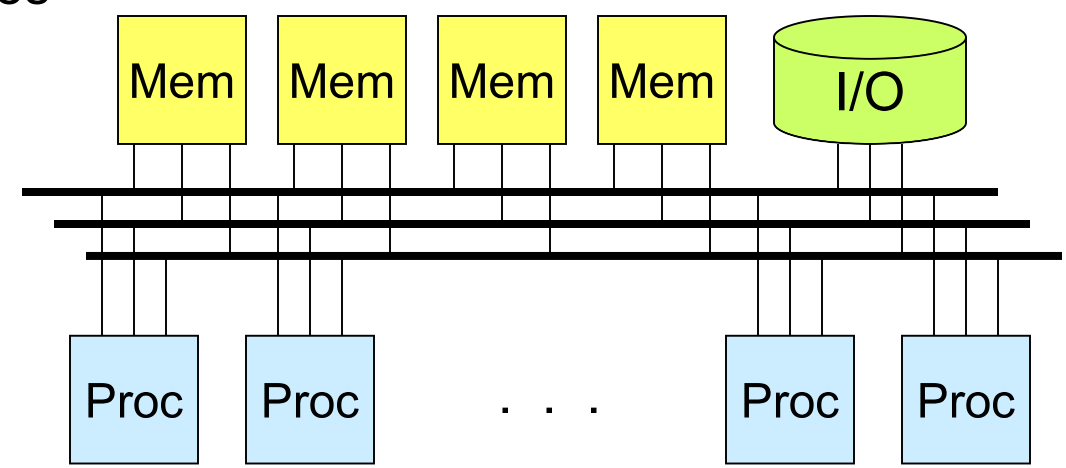
Bus frequency: 0.5 GHz
Data width: 256 b (32 B)
Mem. Access: 2 bus cycles
$(0.5\text{G})/2 \times 32 = 8$ GB/s

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

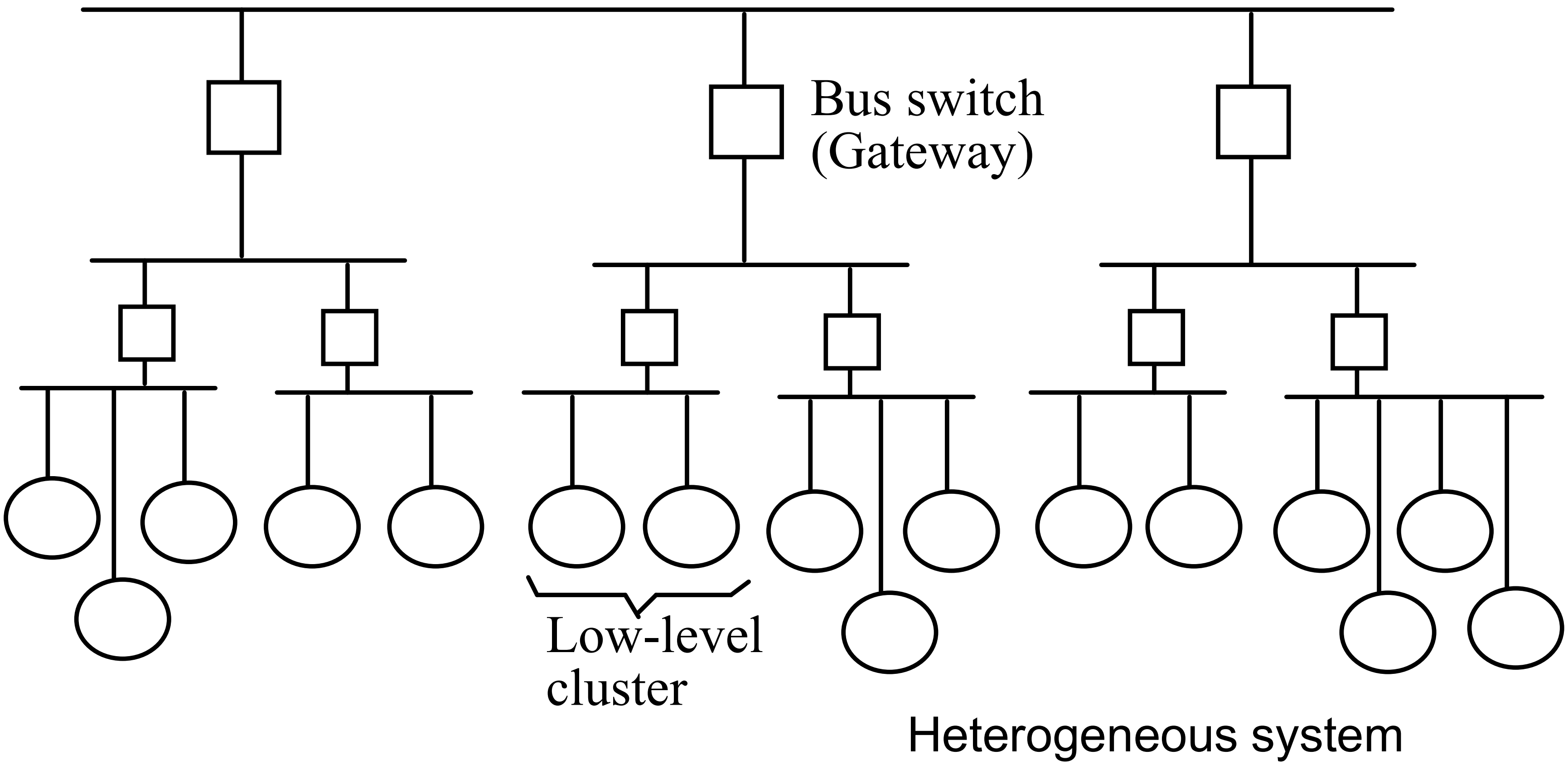


Fig. 4.9 Example of a hierarchical interconnection architecture.

Removing the Processor-to-Memory Bottleneck

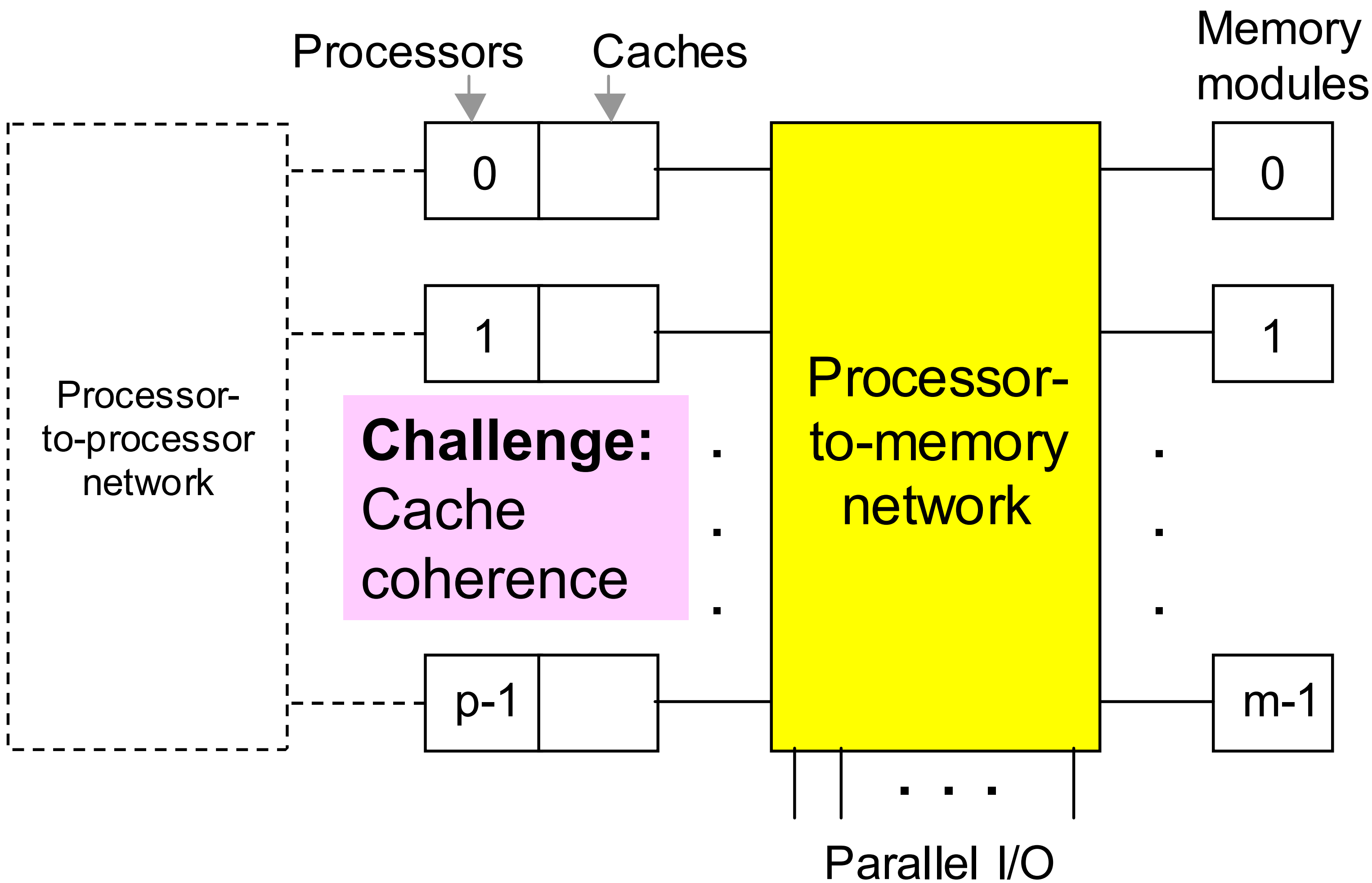


Fig. 4.4 A parallel processor with global memory and processor caches.

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

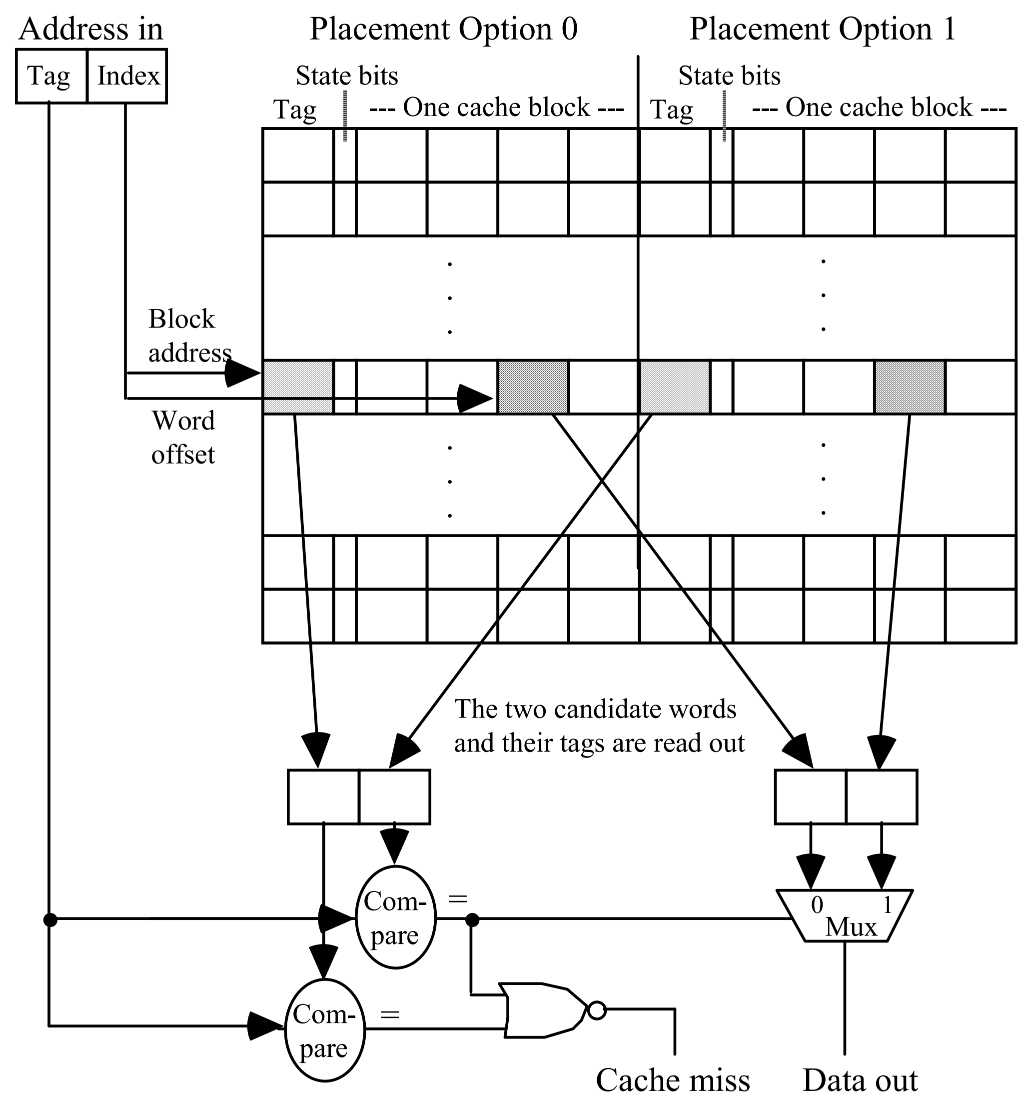


Fig. 18.1 Data storage and access in a two-way set-associative cache.

Benefits of Caching Formulated as Amdahl's Law

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

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

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

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

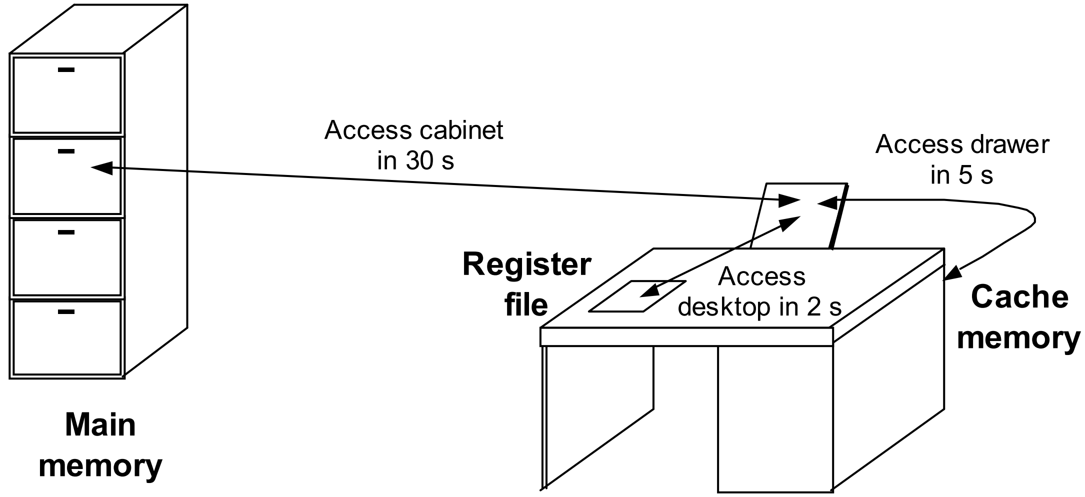


Fig. 18.3 of Parhami's Computer Architecture text (2005)

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}

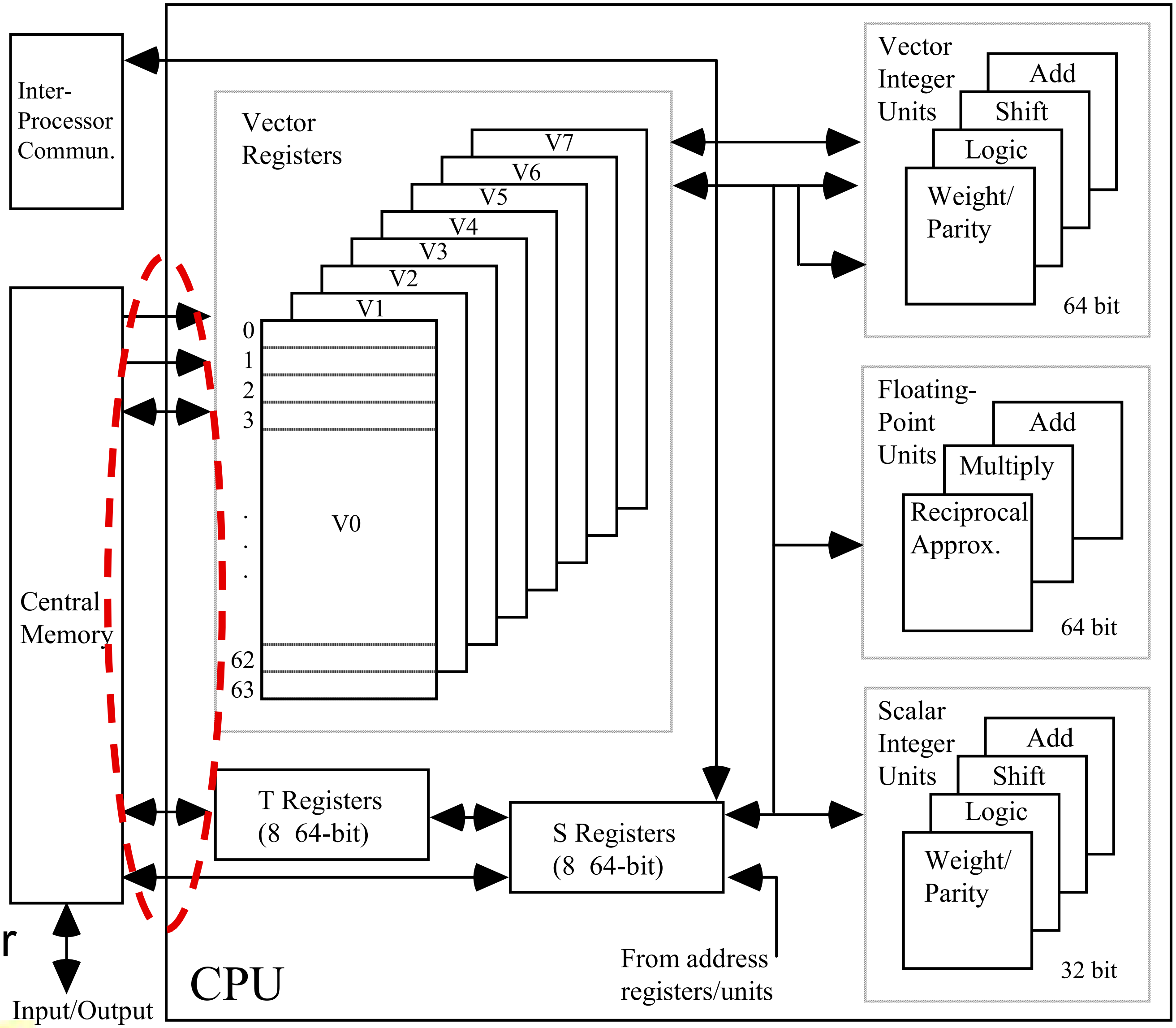
Generalized form of Amdahl's speedup formula:

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

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}

6B.2 Multistage Interconnection Networks

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



The Vector-Parallel Cray Y-MP Computer

Cray Y-MP's Interconnection Network

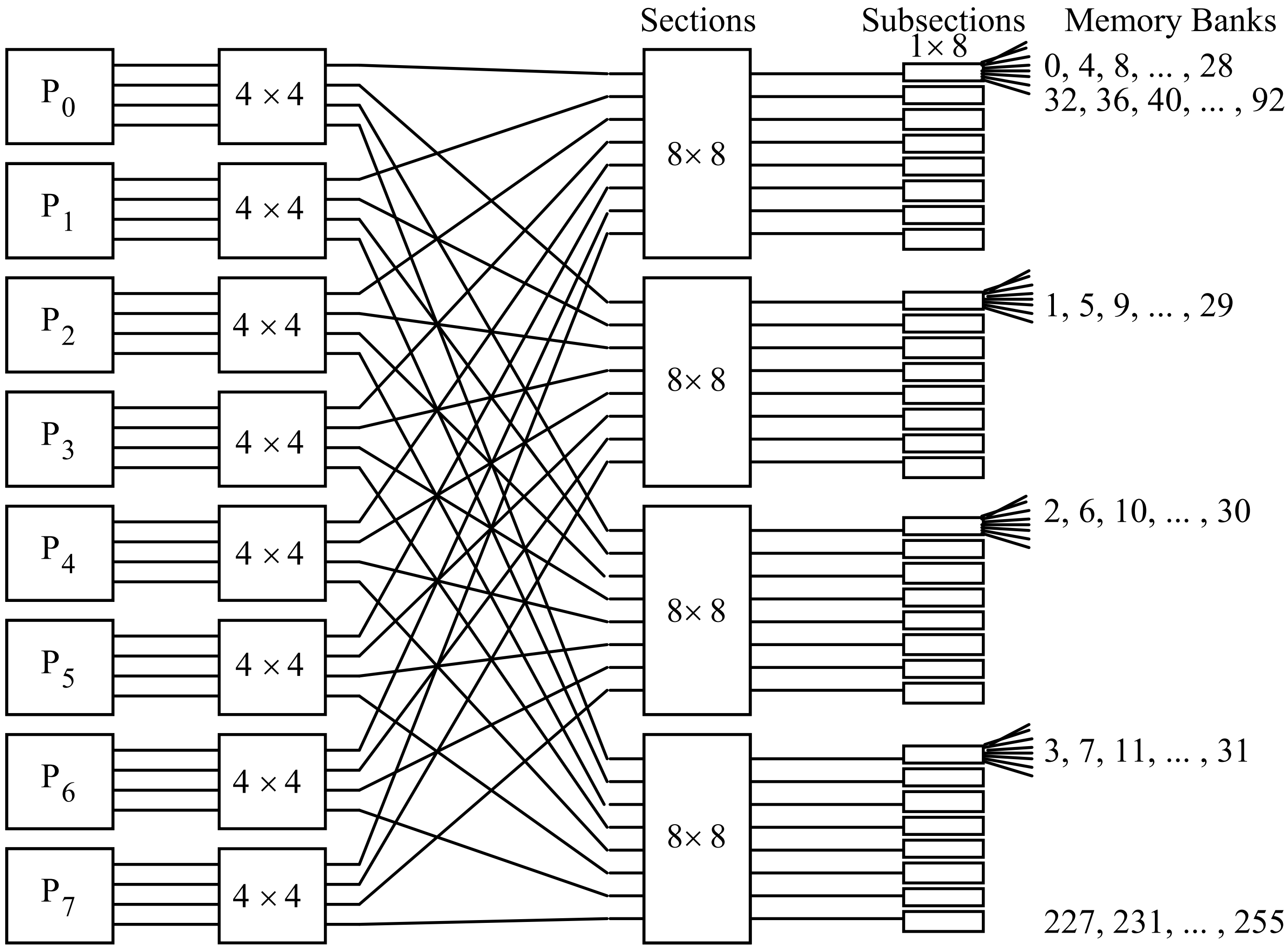
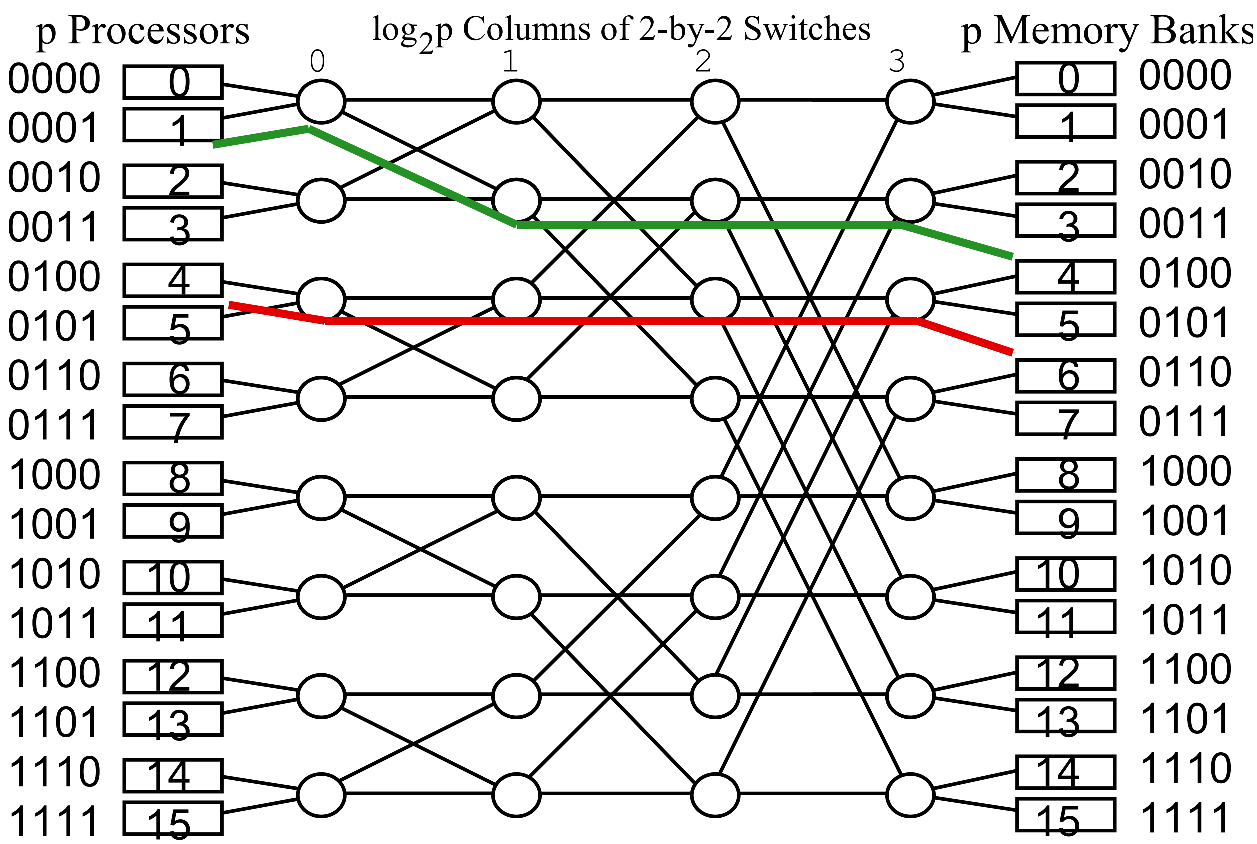


Fig. 21.6 The processor-to-memory interconnection network of Cray Y-MP.

Butterfly Processor-to-Memory 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

Fig. 6.9 Example of a multistage memory access network.

Two ways to use the butterfly:
- Edge switches 1 x 2 and 2 x 1
- All switches 2 x 2

Butterfly as Multistage Interconnection Network

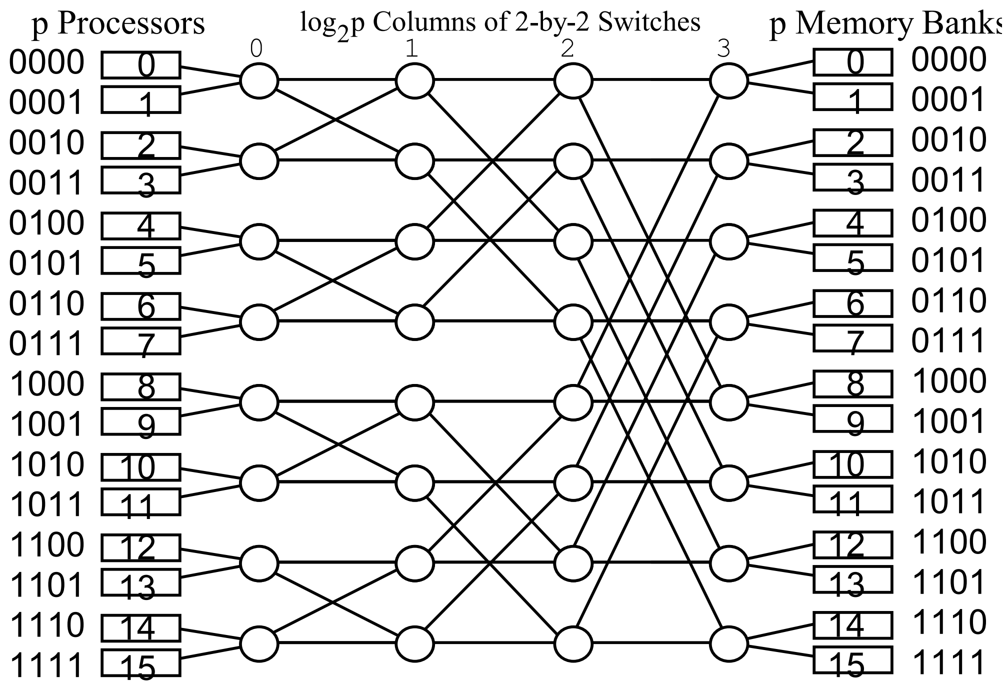


Fig. 6.9 Example of a multistage memory access network

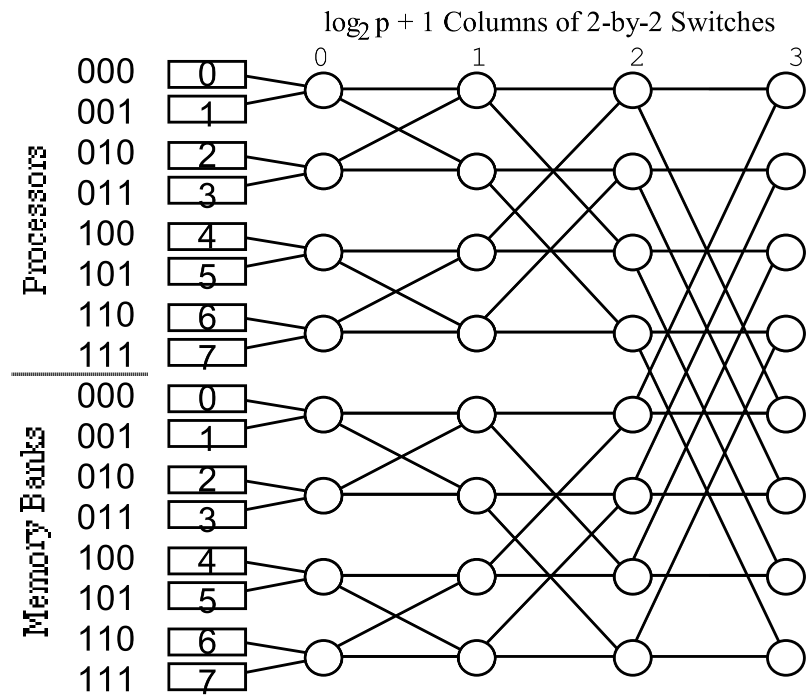


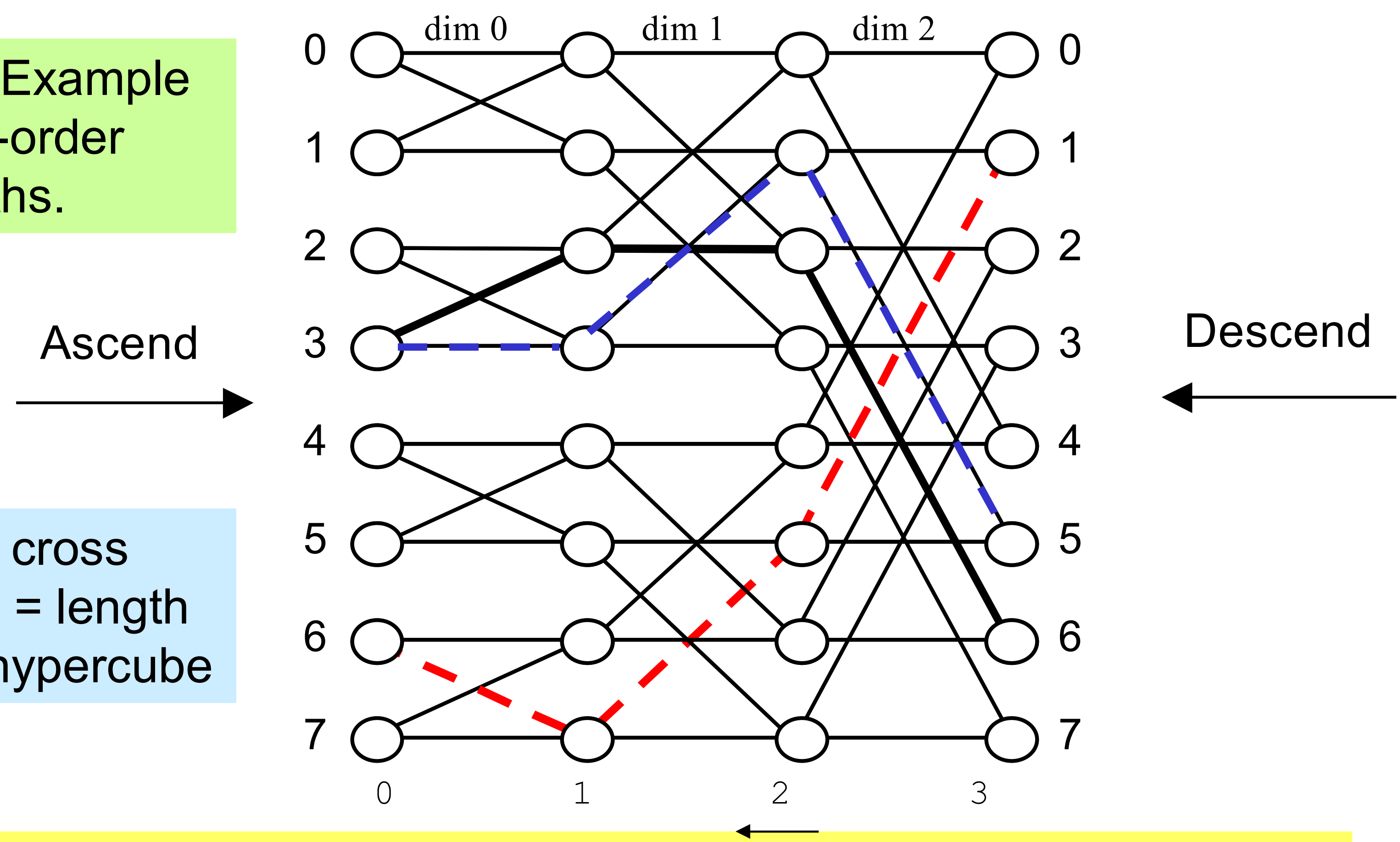
Fig. 15.8 Butterfly network used to connect modules that are on the same side

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)

Self-Routing on a Butterfly Network

Fig. 14.6 Example dimension-order routing paths.



Number of cross links taken = length of path in hypercube

From node 3 to 6: routing tag = 011 ⊕ 110 = 101 “cross-straight-cross”
From node 3 to 5: routing tag = 011 ⊕ 101 = 110 “straight-cross-cross”
From node 6 to 1: routing tag = 110 ⊕ 001 = 111 “cross-cross-cross”

Butterfly Is Not a Permutation Network

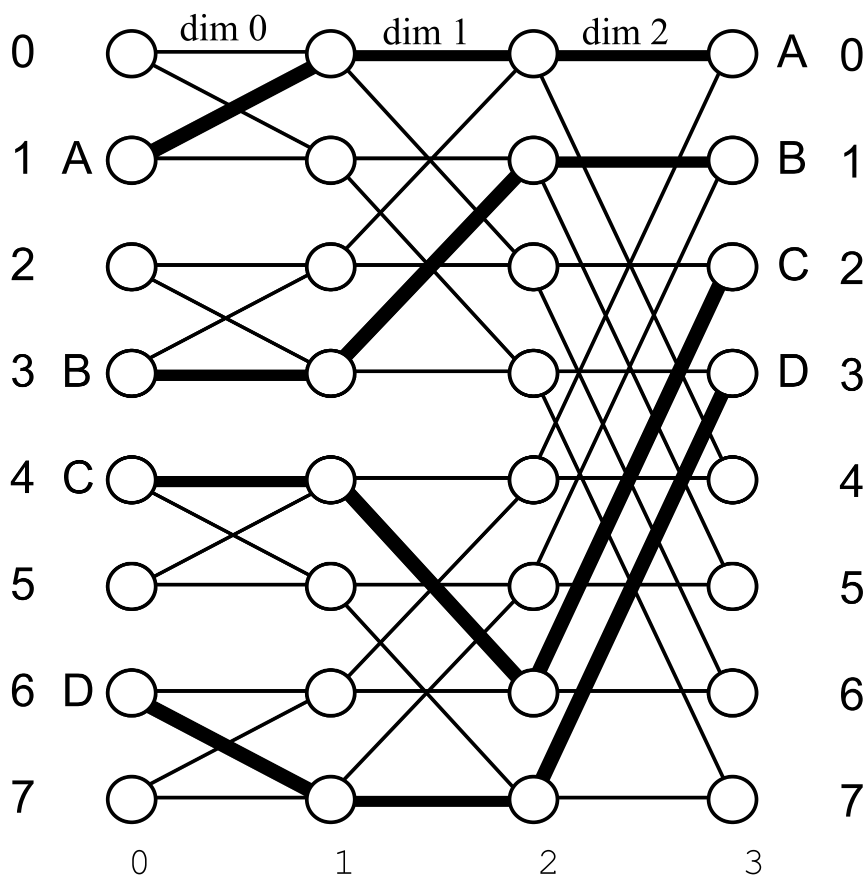


Fig. 14.7 Packing is a "good" routing problem for dimension-order routing on the hypercube.

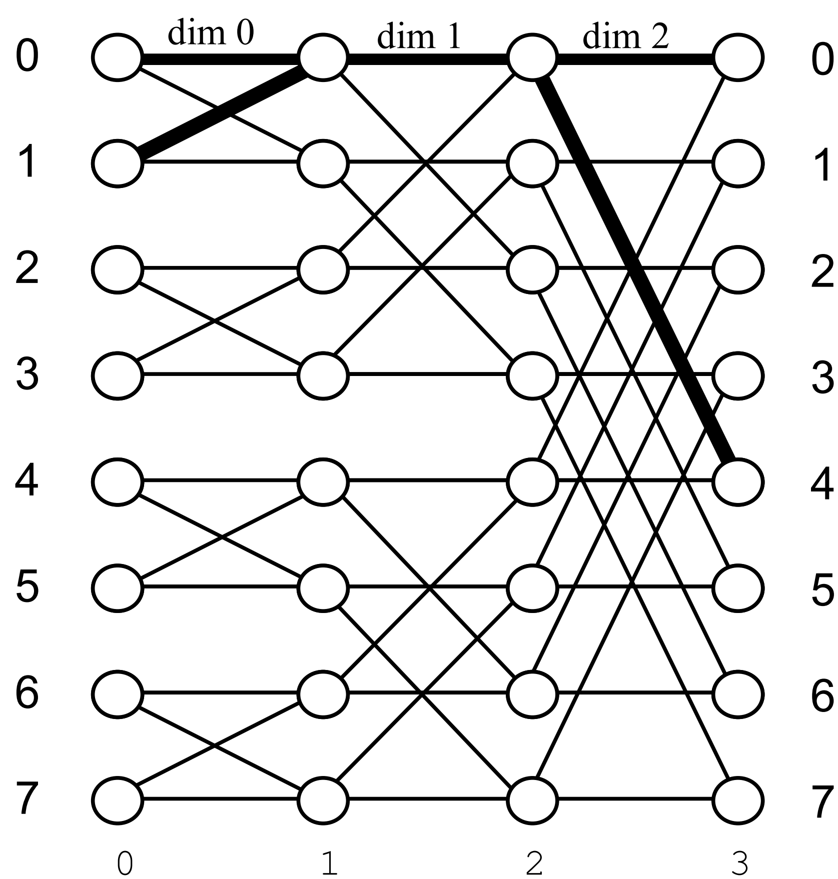


Fig. 14.8 Bit-reversal permutation is a "bad" routing problem for dimension-order routing on the hypercube.

Structure of Butterfly Networks

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

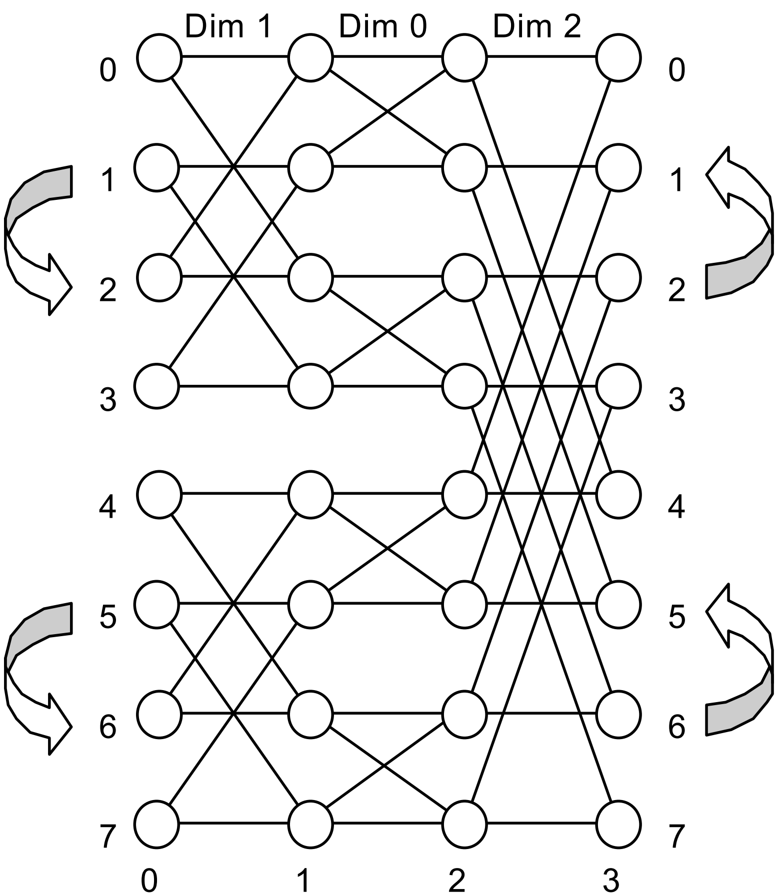
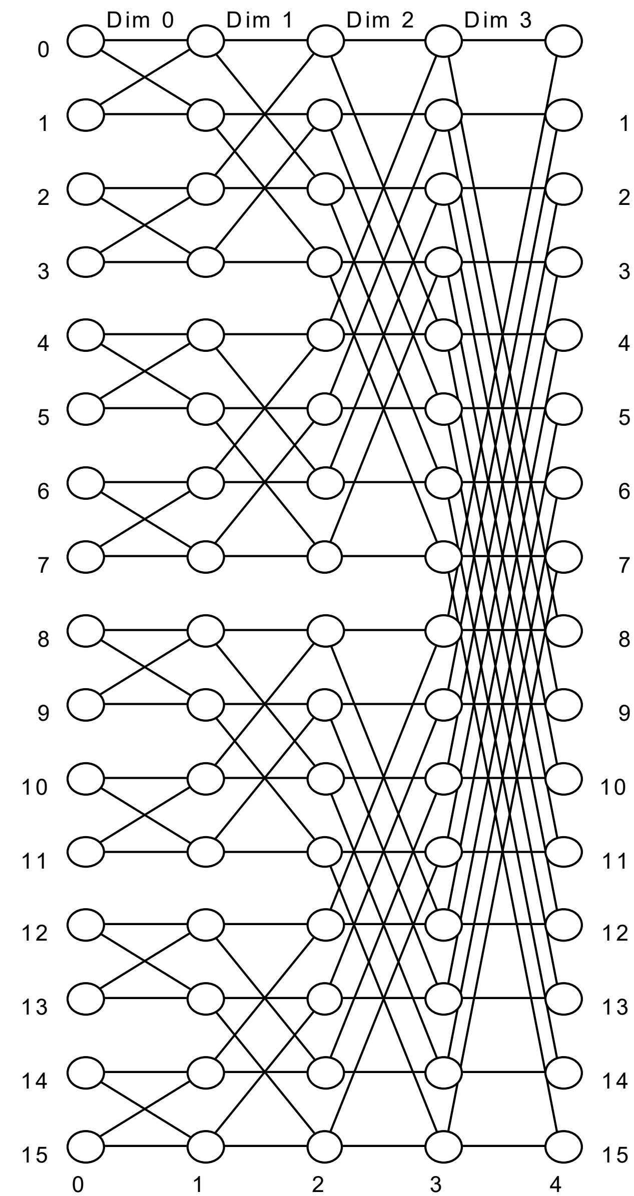


Fig. 15.5 Butterfly network with permuted dimensions.



The 16-row butterfly network.

Beneš Network

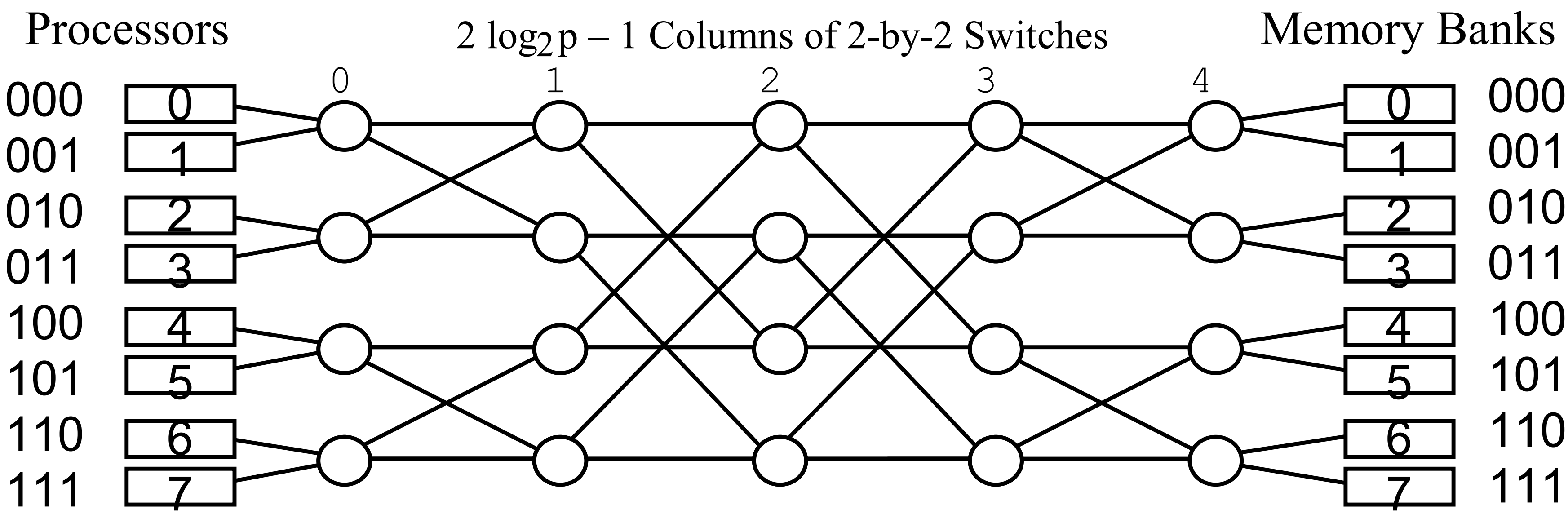


Fig. 15.9 Beneš network formed from two back-to-back butterflies.

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?

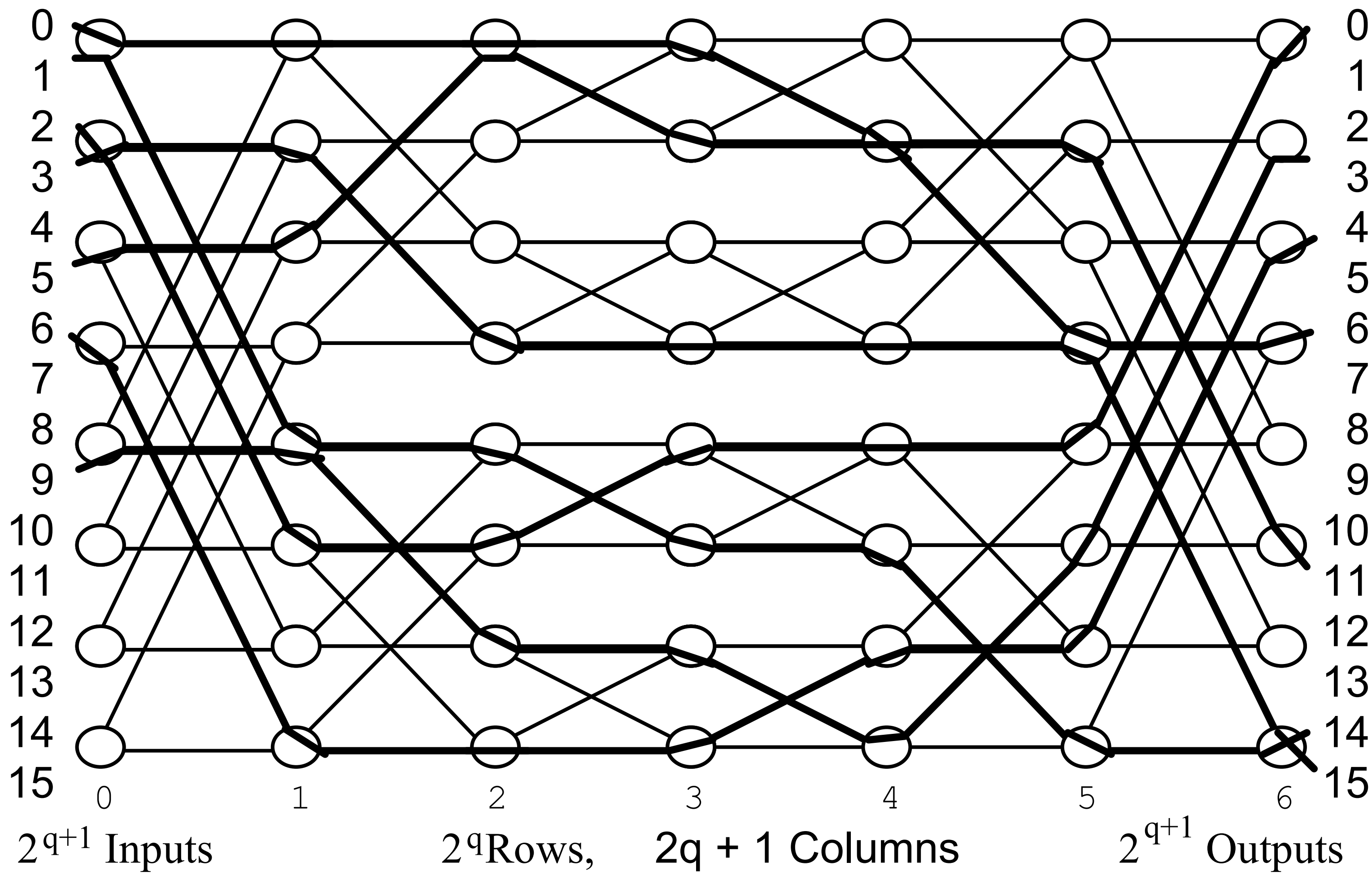


Fig. 15.10 Another example of a Beneš network.

Augmented Data Manipulator Network

Data manipulator network was used in Goodyear MPP, an early SIMD parallel machine.

"Augmented" means that switches in a column are independent, as opposed to all being set to same state (simplified control).

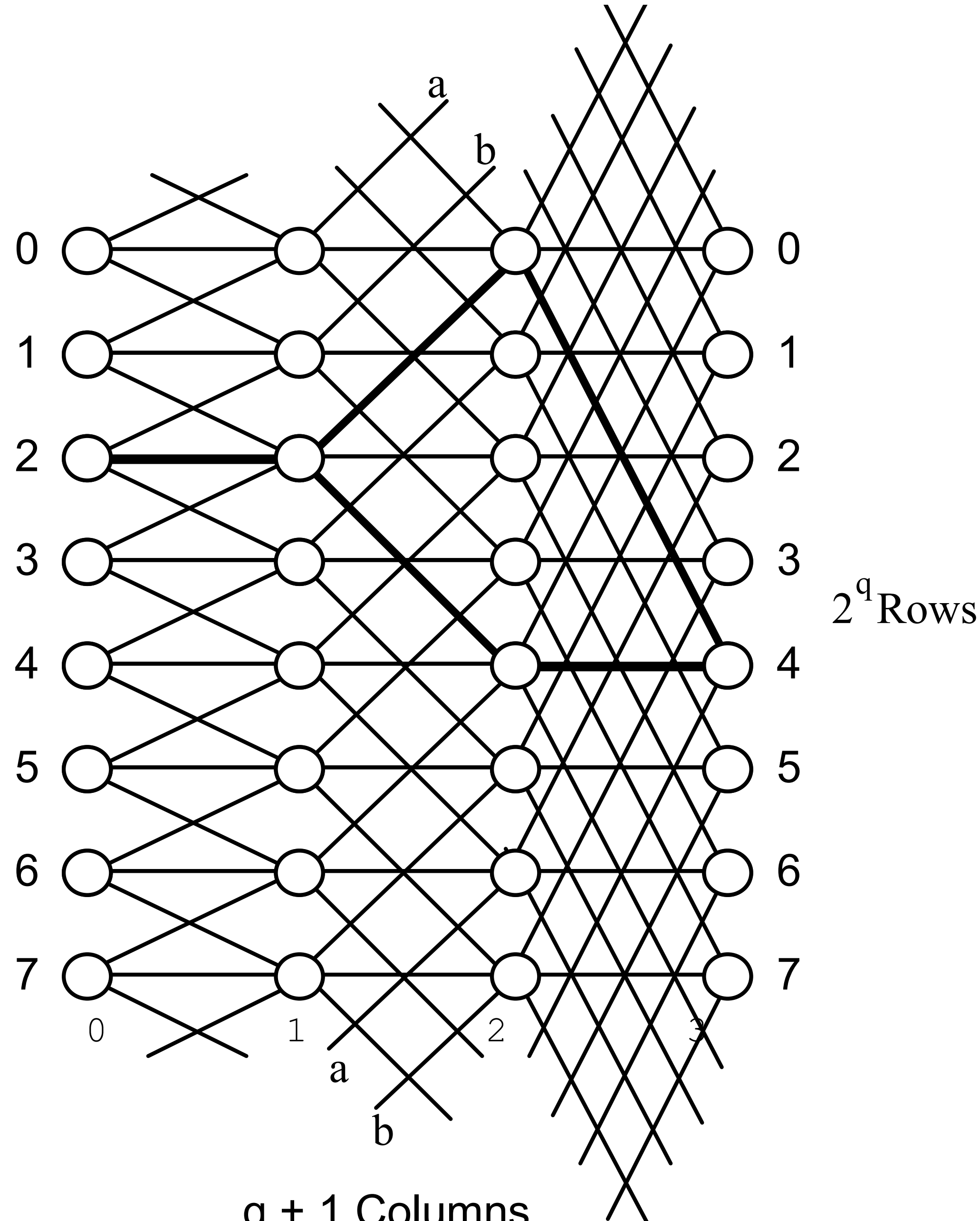


Fig. 15.12 Augmented data manipulator network.

Fat Trees

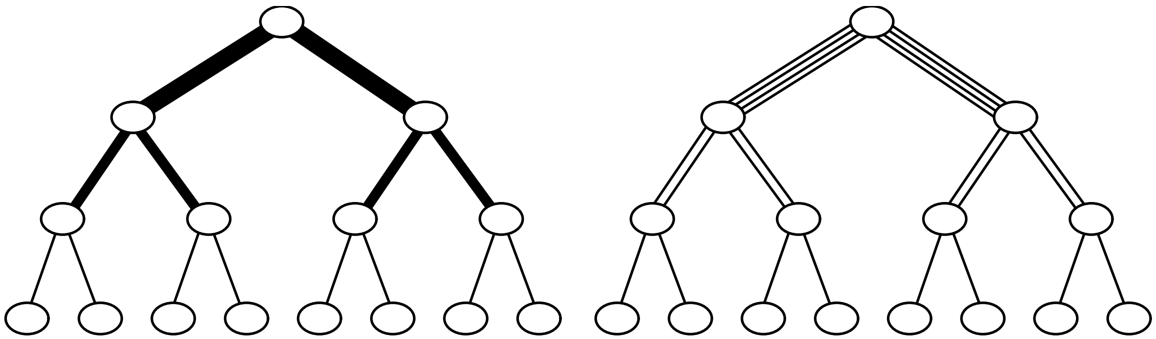
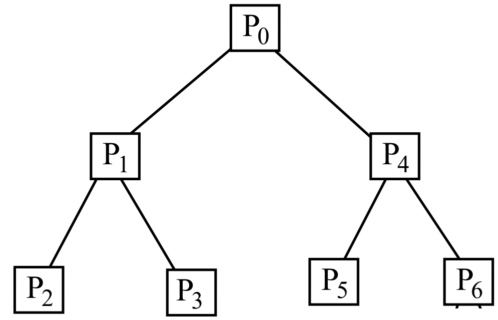


Fig. 15.6 Two representations of a fat tree.



Skinny tree?

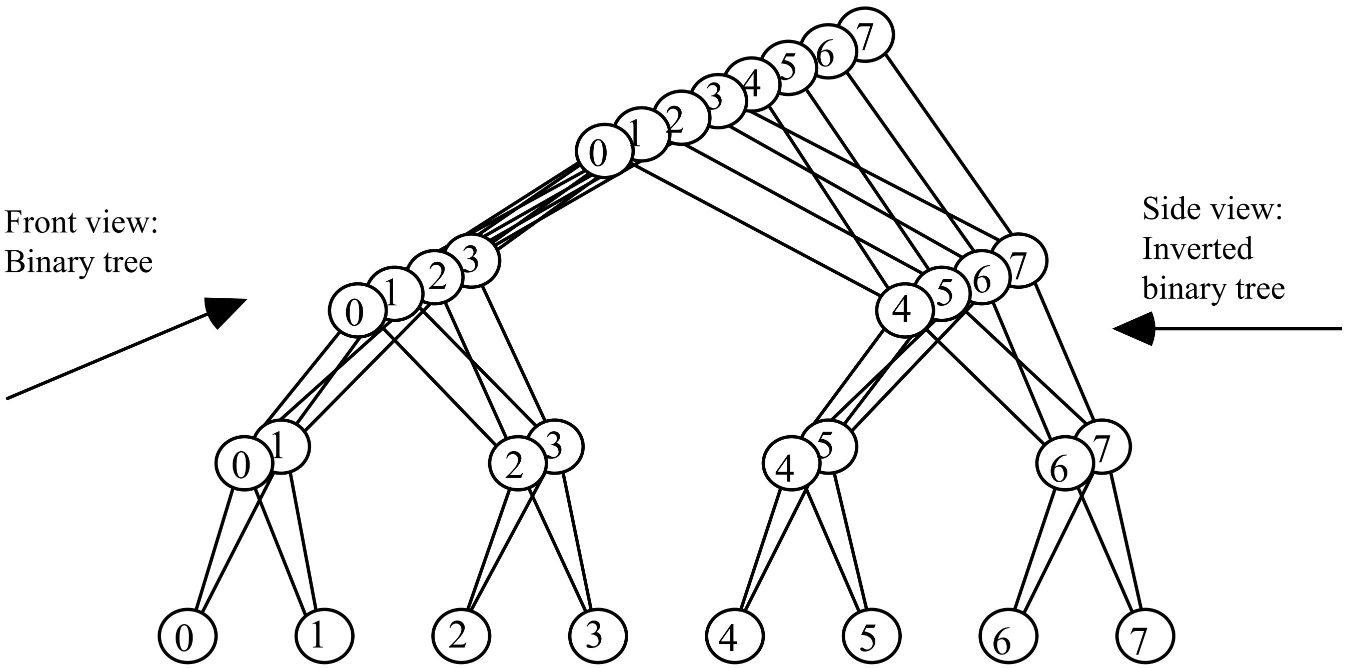


Fig. 15.7 Butterfly network redrawn as a fat tree.

The Sea of Indirect 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

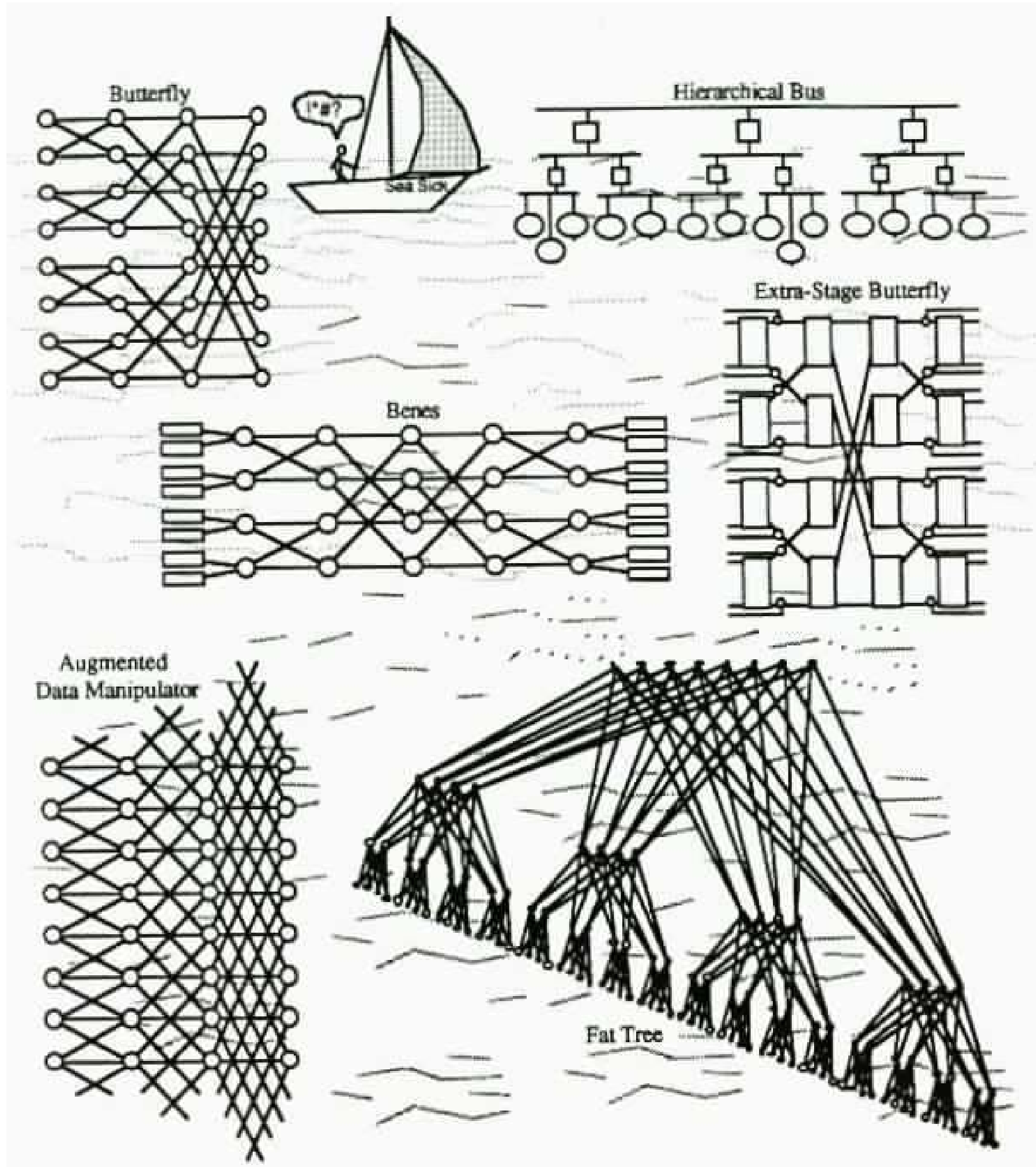


Fig. 4.8 (modified)
The sea of indirect interconnection networks.

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

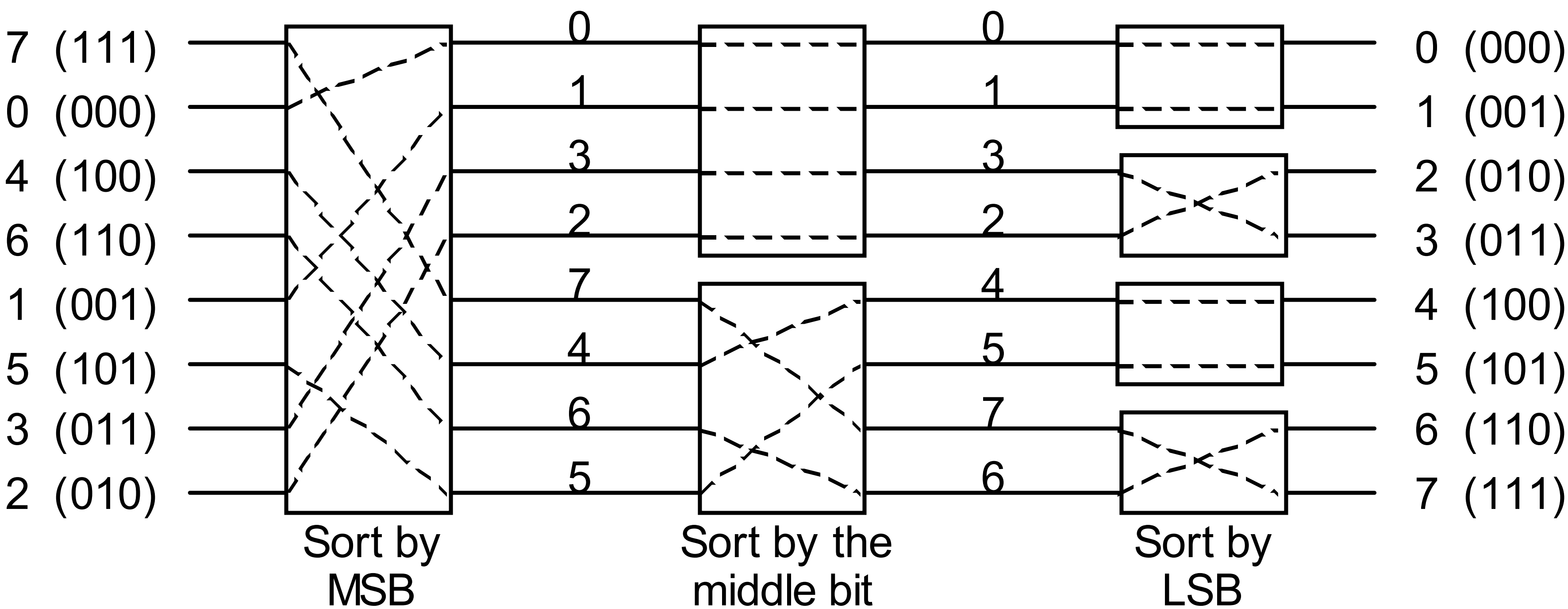


Fig. 16.14 Example of sorting on a binary radix sort network.

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 2×2 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

6B.3 Cache Coherence Protocols

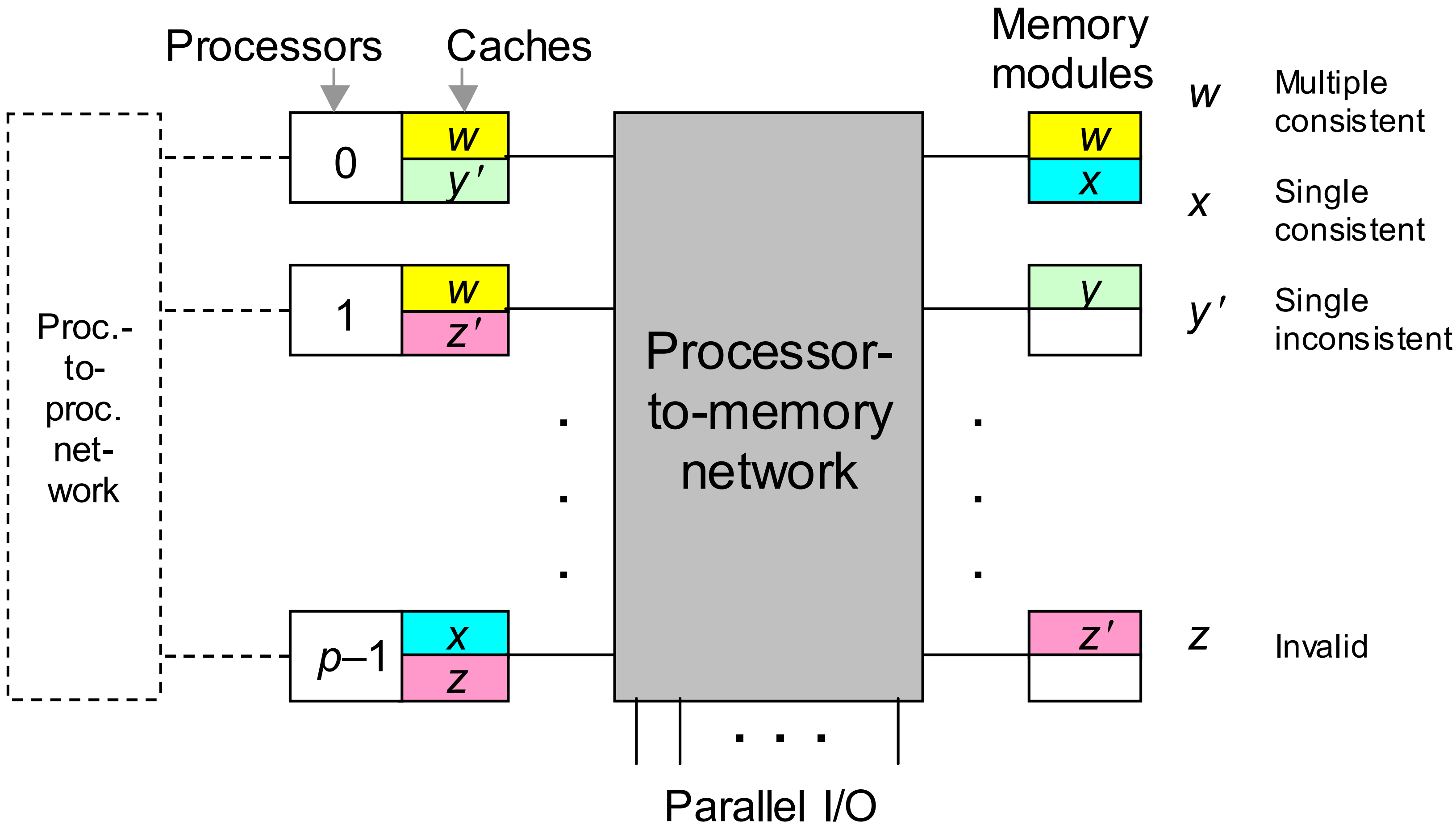


Fig. 18.2 Various types of cached data blocks in a parallel processor with global memory and processor caches.

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

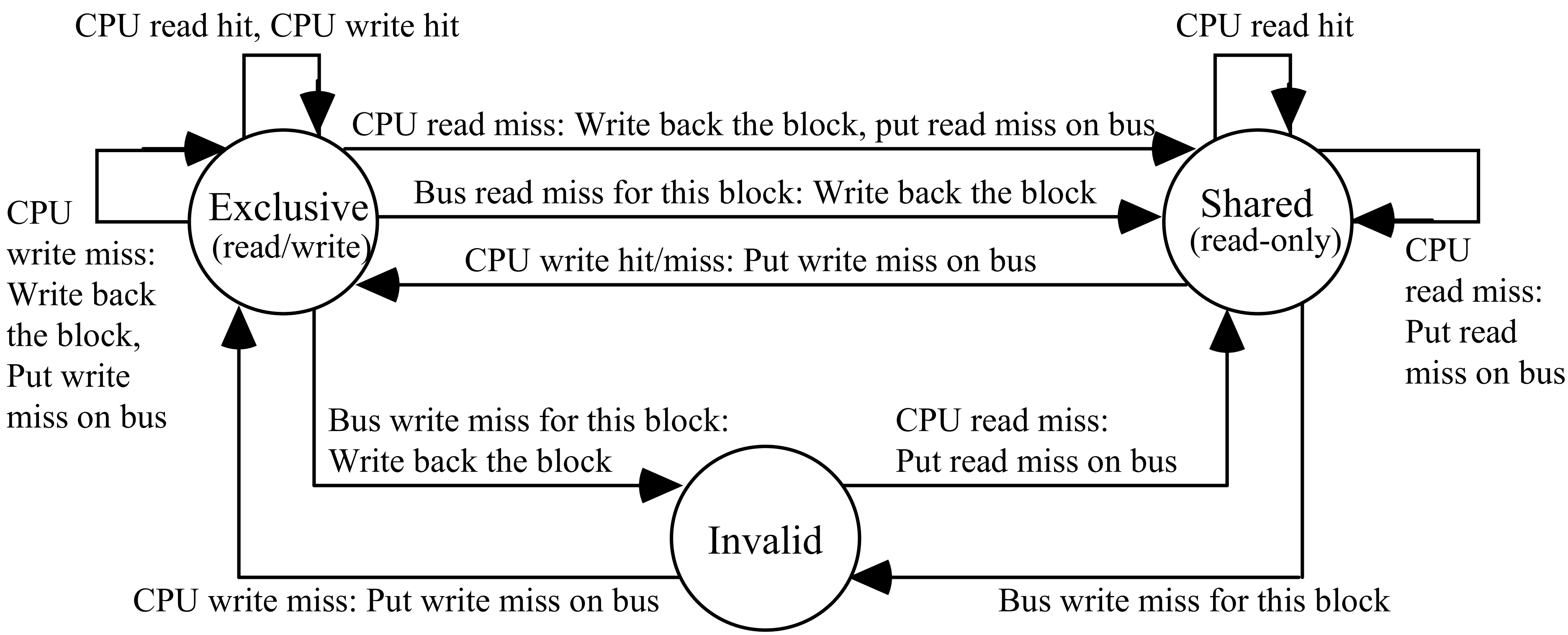


Fig. 18.3 Finite-state control mechanism for a bus-based snoopy cache coherence protocol.

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

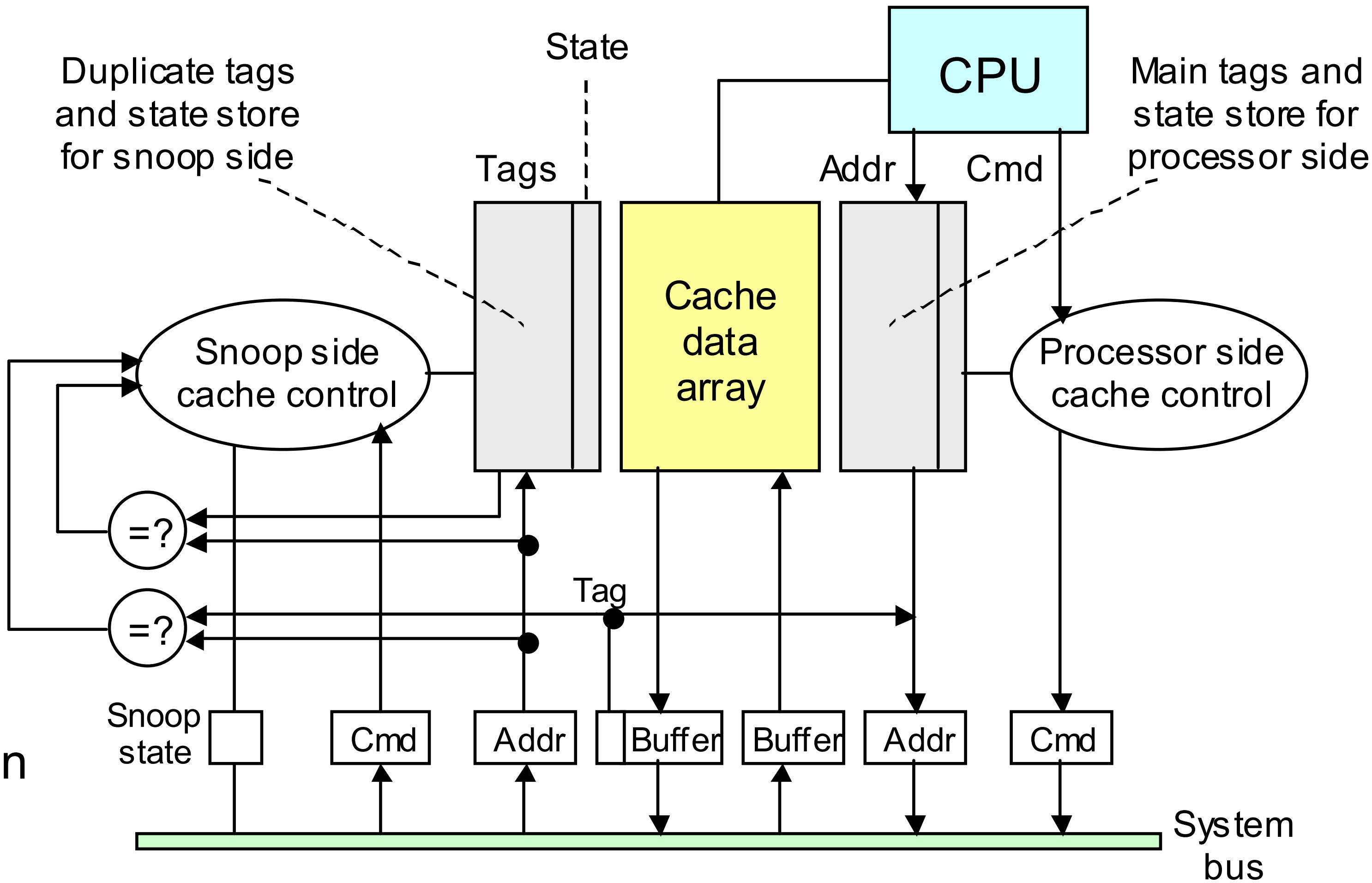
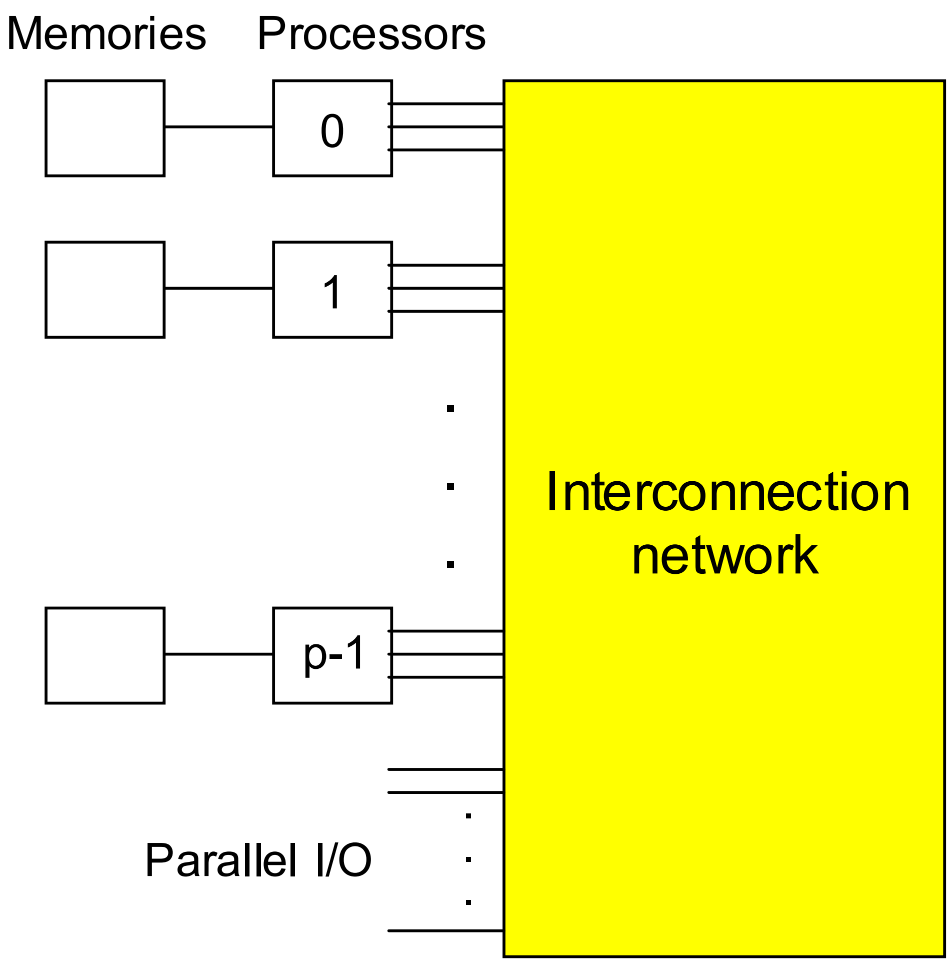


Fig. 27.7 of Parhami's Computer Architecture text.

Scalable (Distributed) Shared Memory



Some Terminology:

NUMA
Nonuniform memory access
(distributed shared memory)

UMA
Uniform memory access
(global shared memory)

COMA
Cache-only memory arch

Fig. 4.5 A parallel processor with distributed memory.

Example: A Directory-Based Protocol

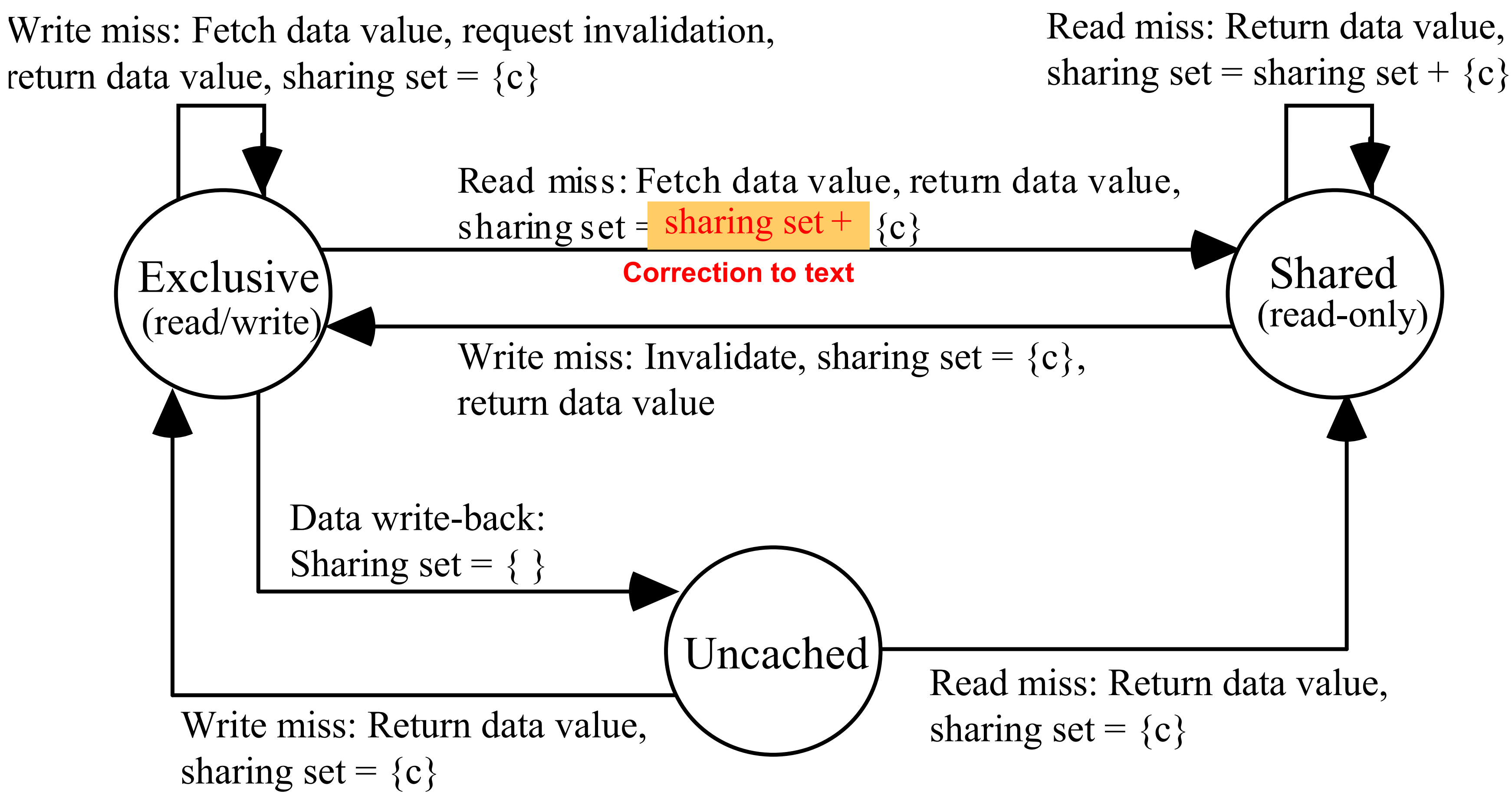


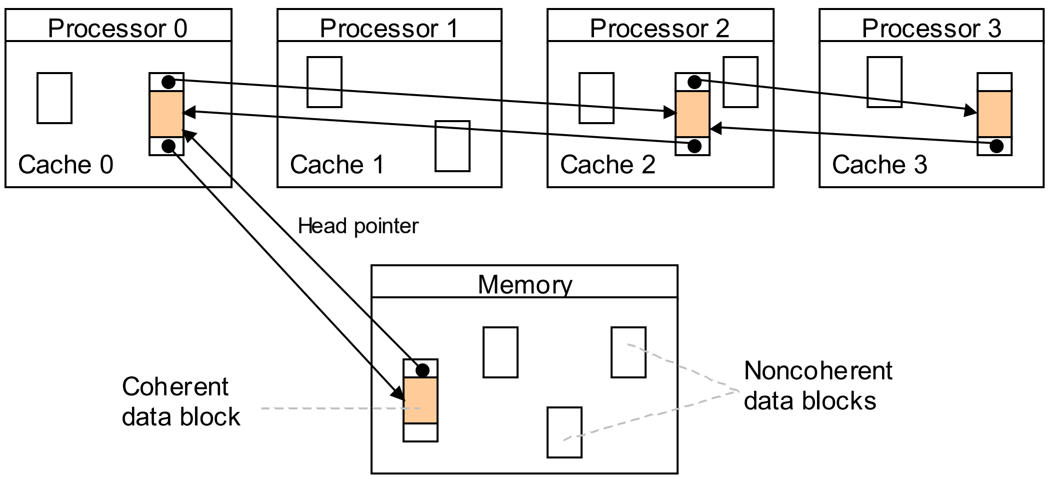
Fig. 18.4 States and transitions for a directory entry in a directory-based coherence protocol (*c* denotes the cache sending the message).

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)

6B.4 Data Allocation for Conflict-Free 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

	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	0	1	2	3	4	5

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 qD 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.

6B.5 Distributed Shared Memory

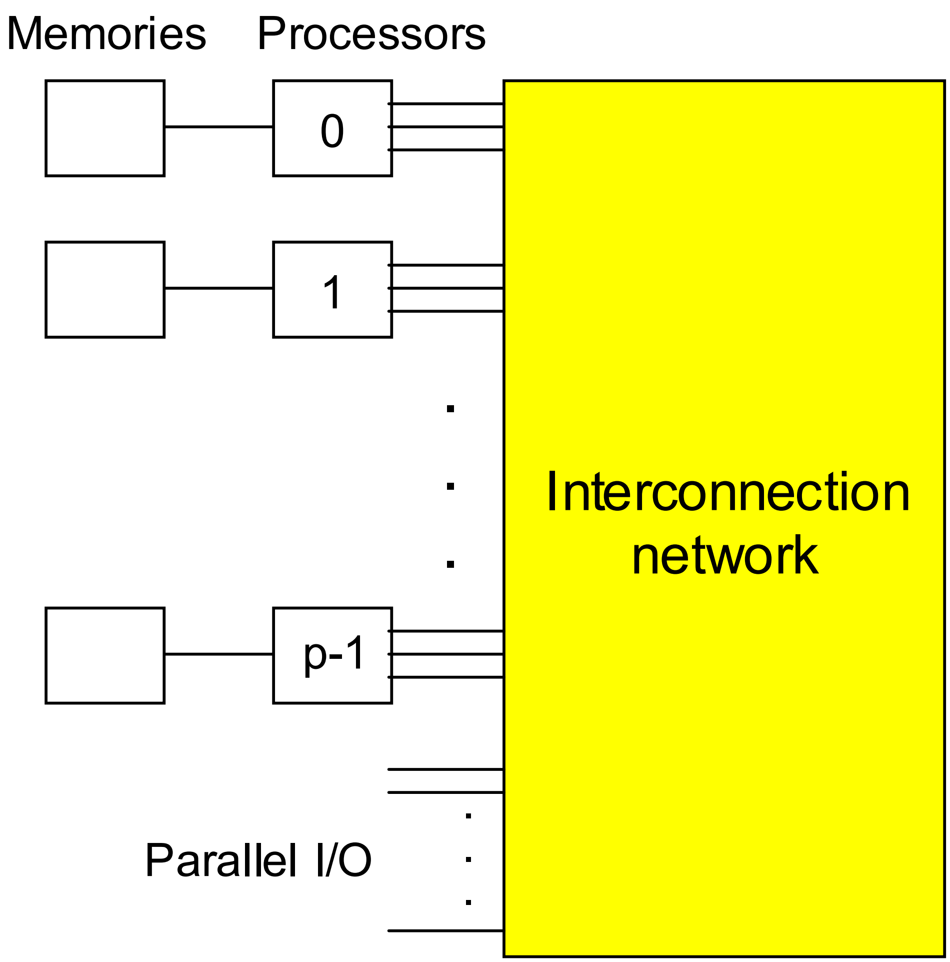


Fig. 4.5 A parallel processor with distributed memory.

Some Terminology:

NUMA
Nonuniform memory access
(distributed shared memory)

UMA
Uniform memory access
(global shared memory)

COMA
Cache-only memory arch

Butterfly-Based Distributed Shared Memory

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

Use hash function to map memory locations to modules

p locations → *p* modules, not necessarily distinct

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

Average slowdown = O(log *p*)

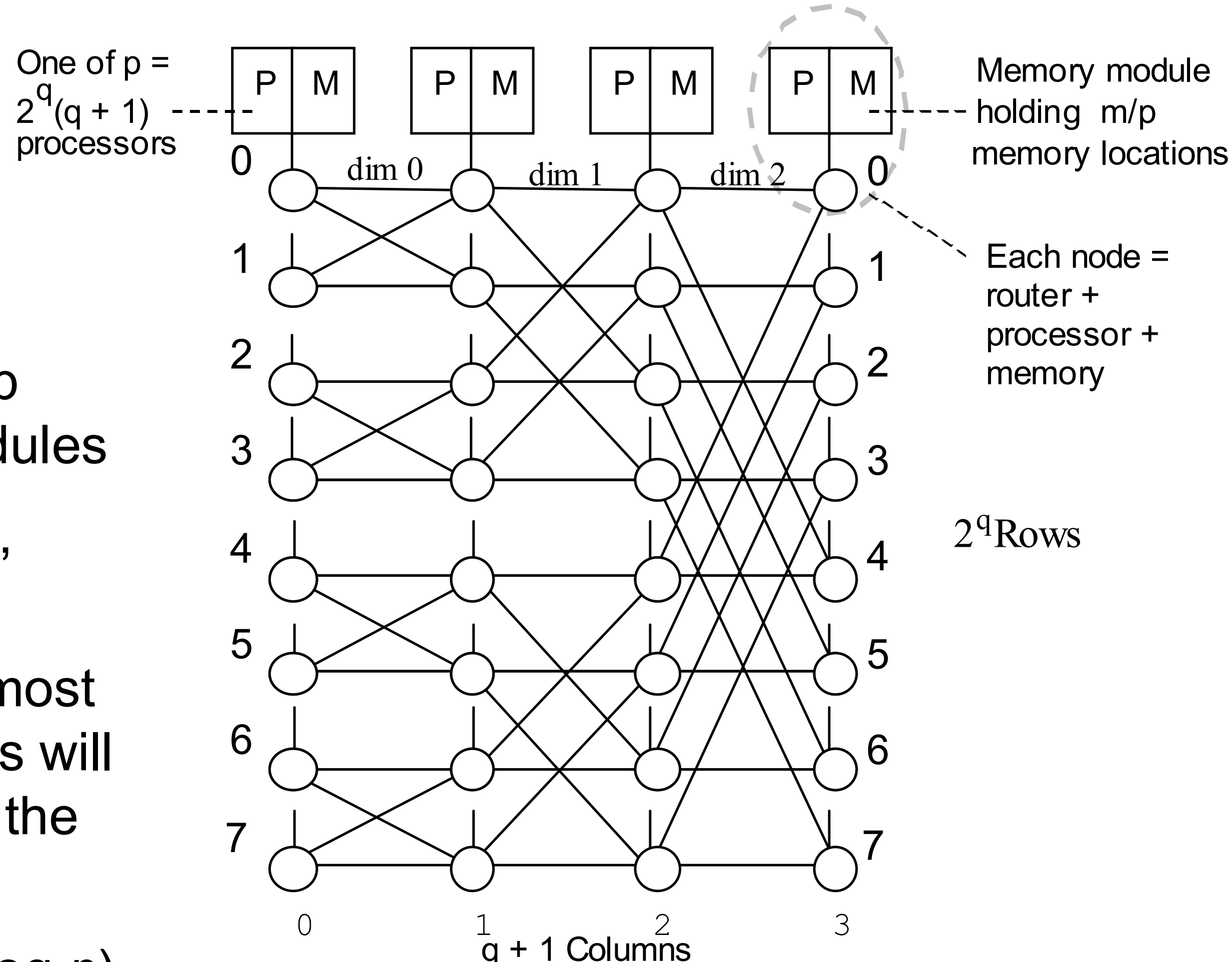


Fig. 17.2 Butterfly distributed-memory machine emulating the PRAM.

PRAM Emulation with Butterfly MIN

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

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

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

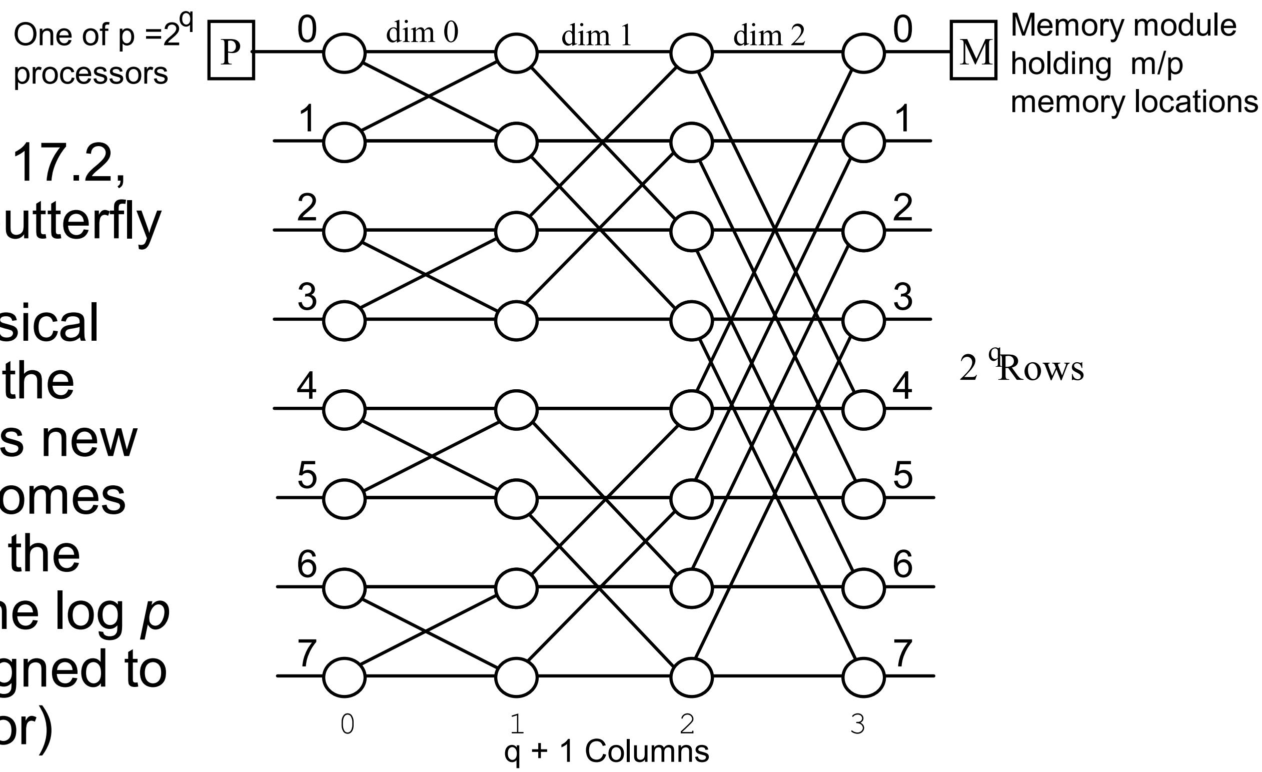


Fig. 17.3 Distributed-memory machine, with a butterfly multistage interconnection network, emulating the PRAM.

Deterministic Shared-Memory Emulation

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

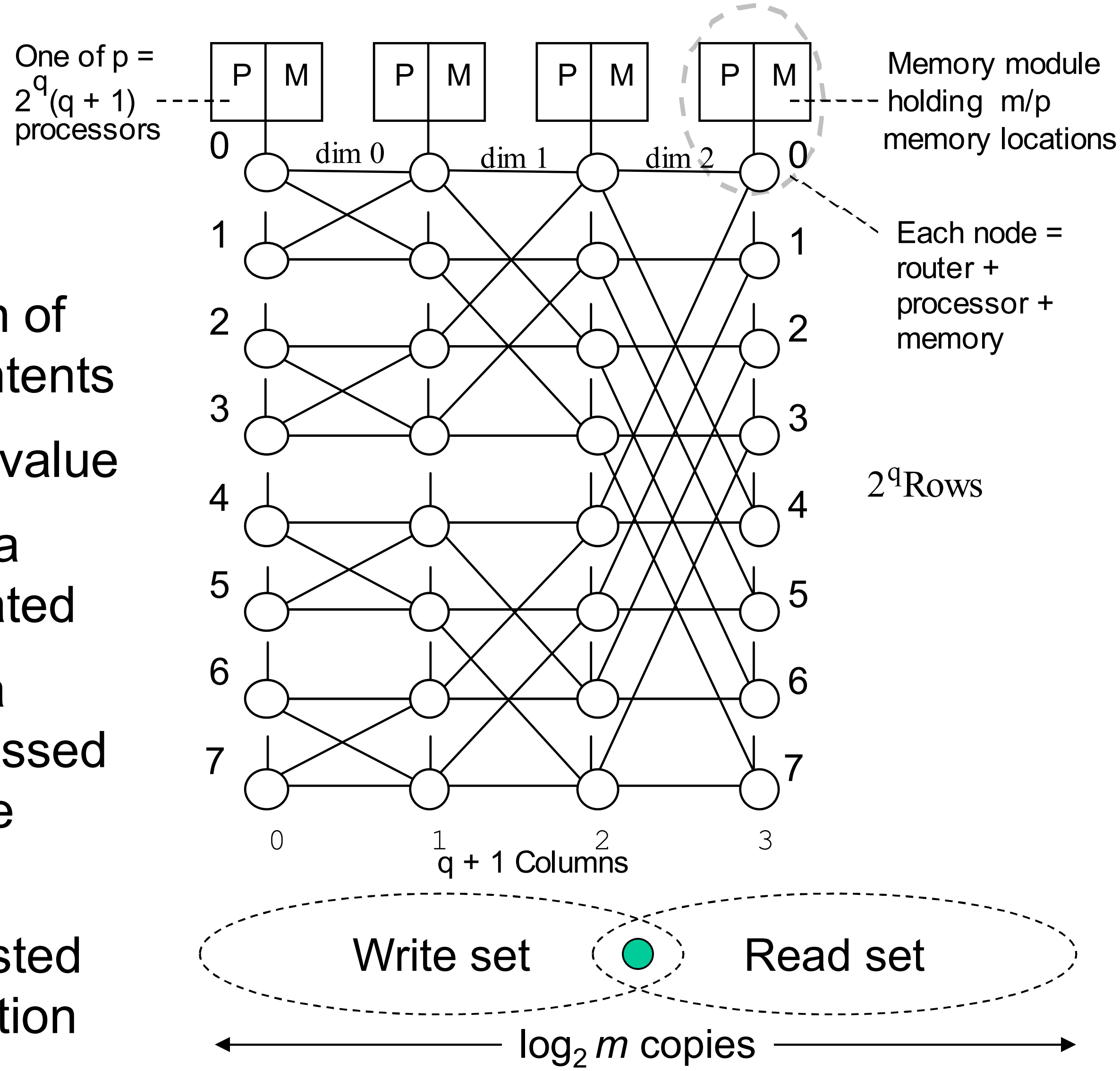
Store $\log_2 m$ copies of each of the *m* memory location contents

Time-stamp each updated value

A "write" is complete once a majority of copies are updated

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

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



PRAM Emulation Using Information Dispersal

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

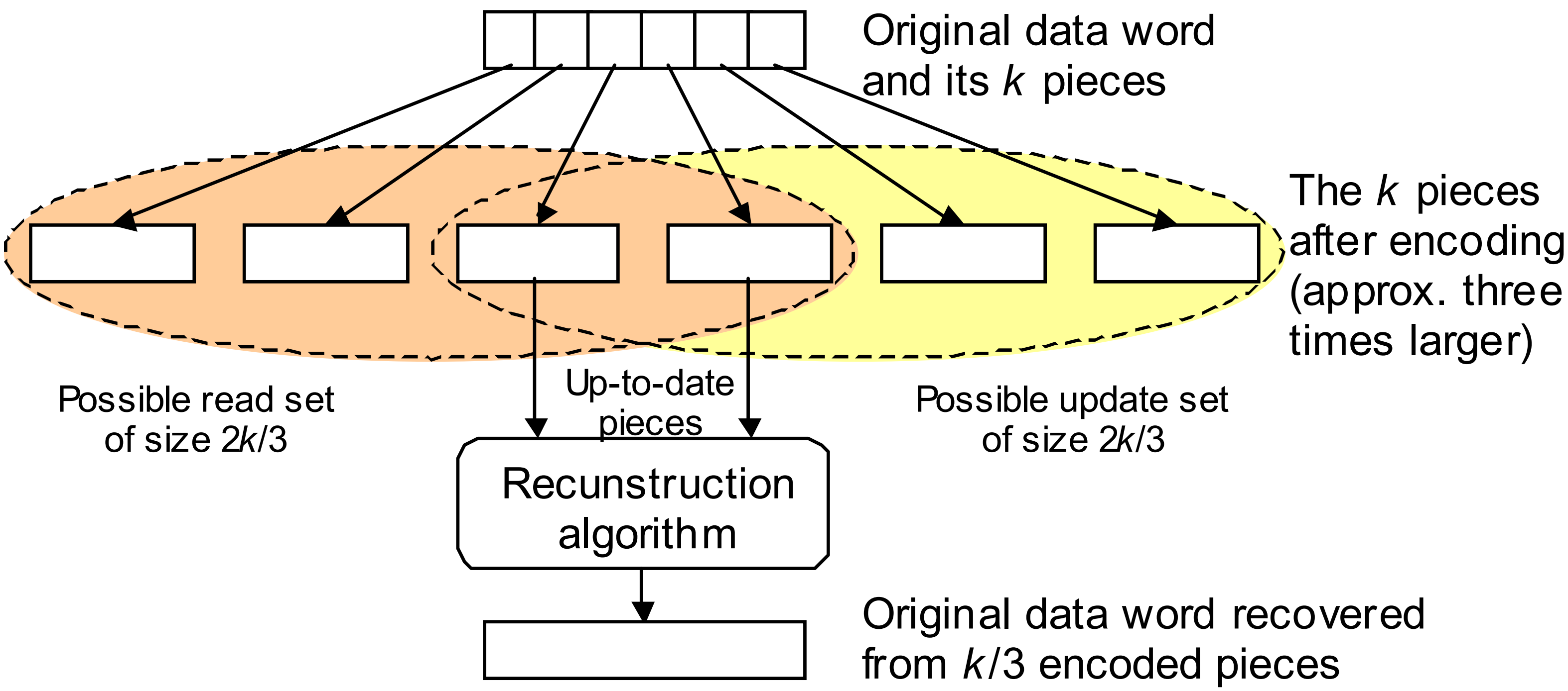


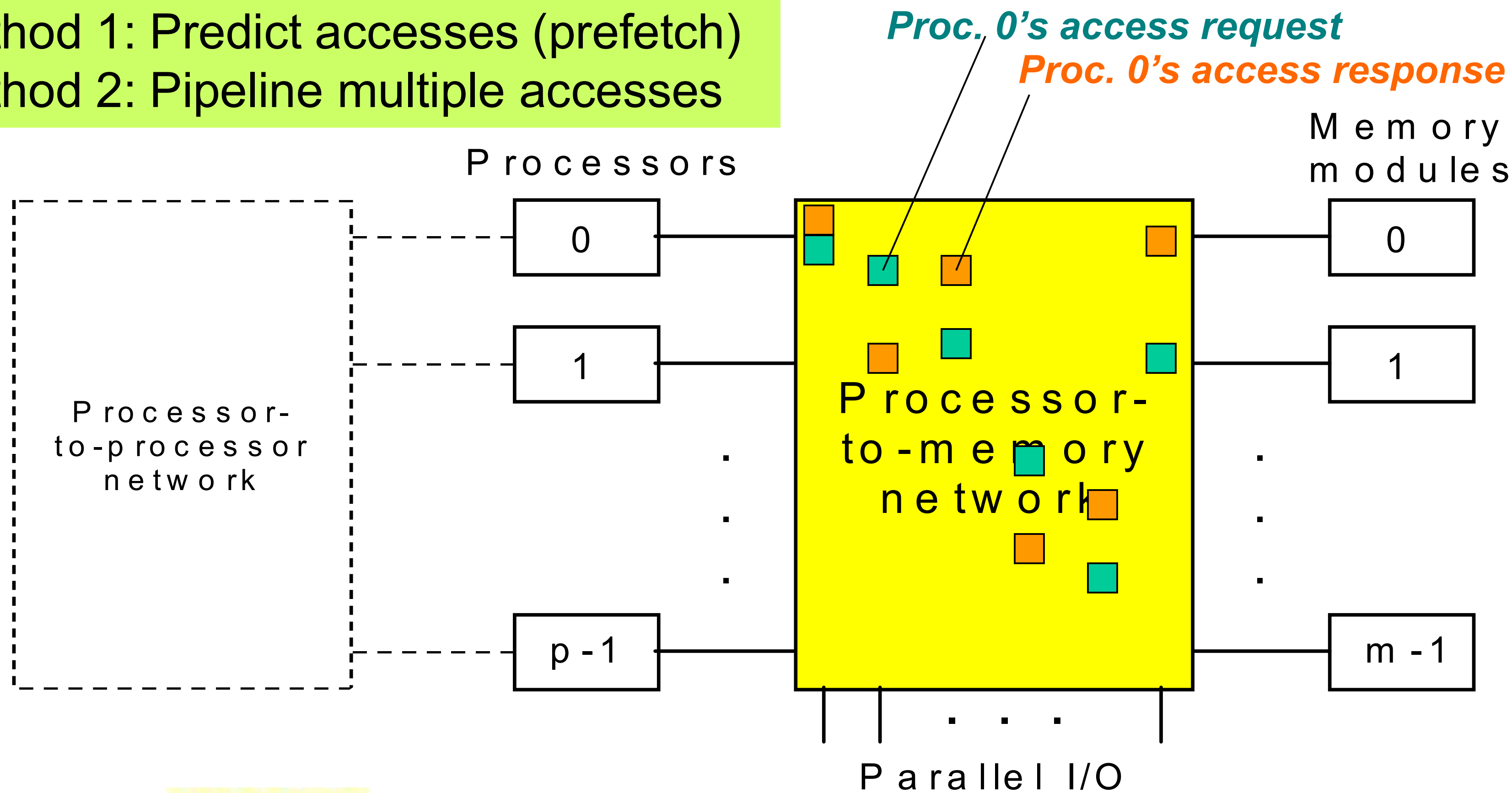
Fig. 17.4 Illustrating the information dispersal approach to PRAM emulation with lower data redundancy.

6B.6 Methods for Memory Latency Hiding

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



6C Shared-Memory Abstractions

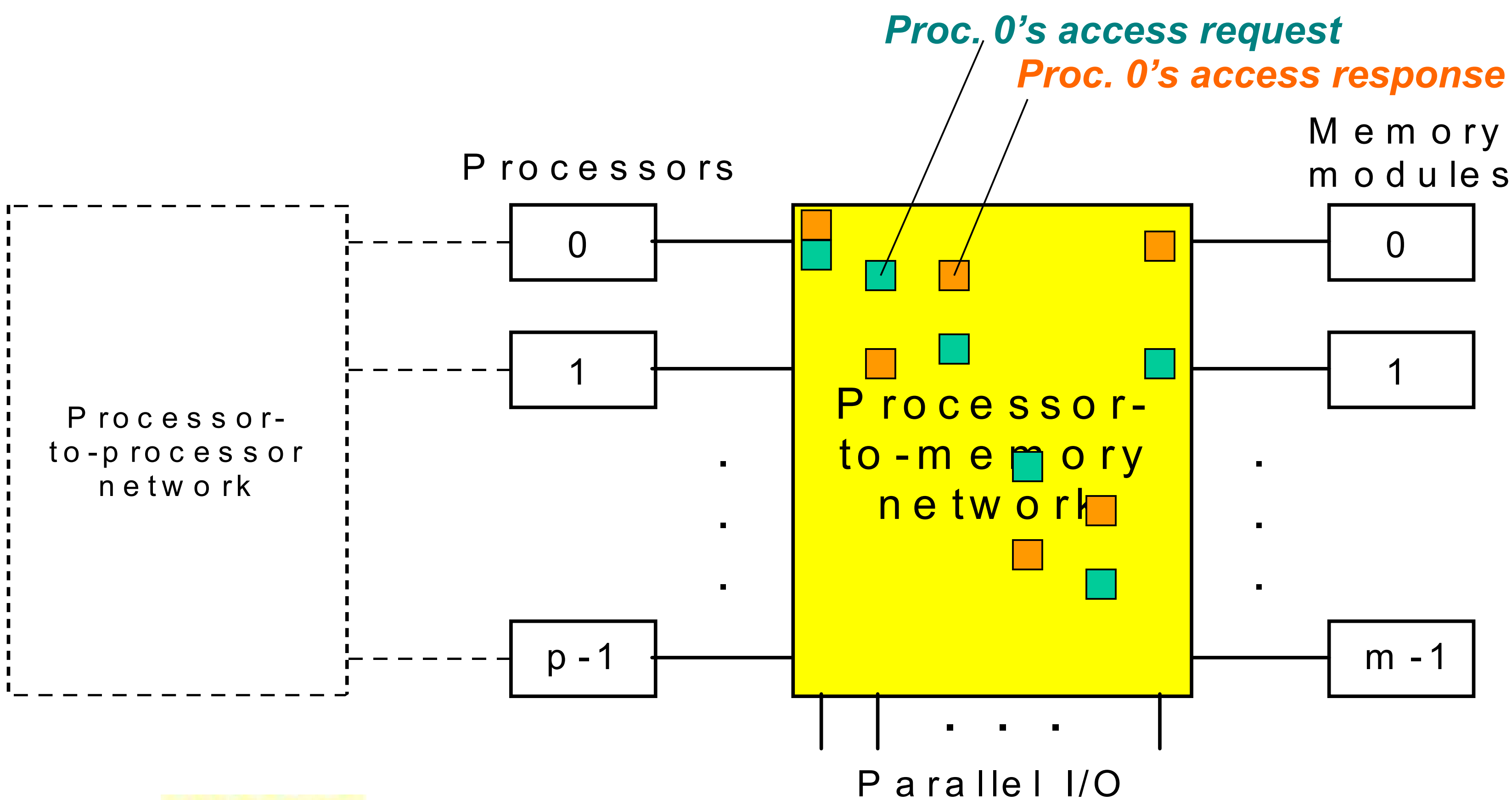
A precise memory view is needed for correct algorithm design

- Sequential consistency facilitates programming
- Less strict consistency models offer better performance

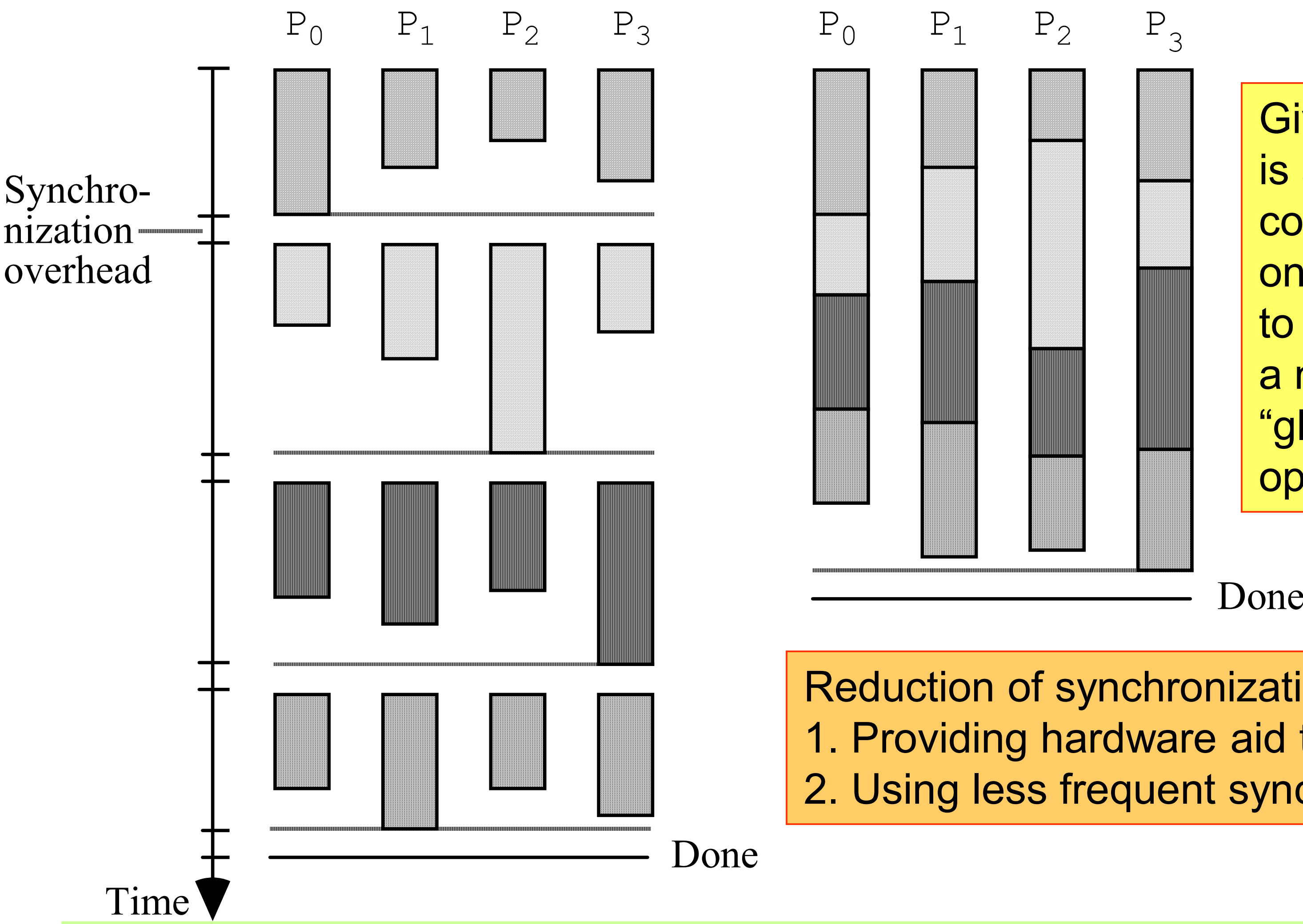
Topics in This Chapter
6C.1 Atomicity in Memory Access
6C.2 Strict and Sequential Consistency
6C.3 Processor Consistency
6C.4 Weak or Synchronization Consistency
6C.5 Other Memory Consistency Models
6C.6 Transactional Memory

6C.1 Atomicity in Memory Access

Performance optimization and latency hiding often imply that memory accesses are interleaved and perhaps not serviced in the order issued



Barrier Synchronization Overhead



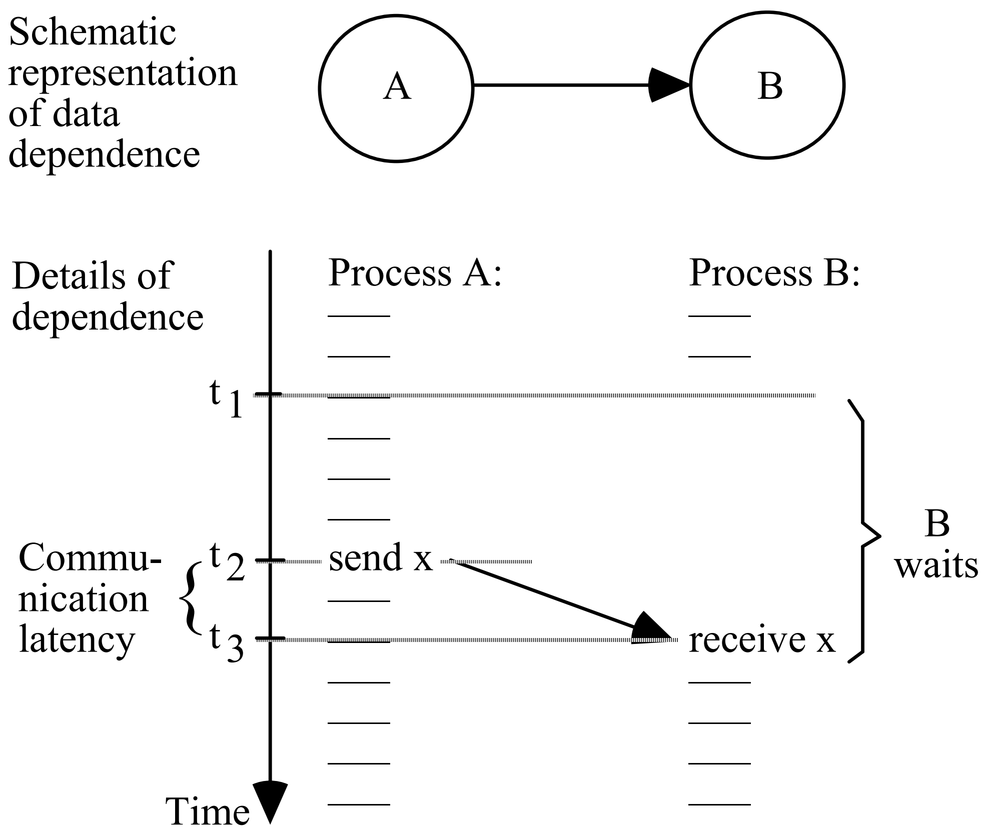
Given that AND is a semigroup computation, it is only a small step to generalize it to a more flexible "global combine" operation

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

Fig. 20.3 The performance benefit of less frequent synchronization.

Synchronization via Message Passing

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



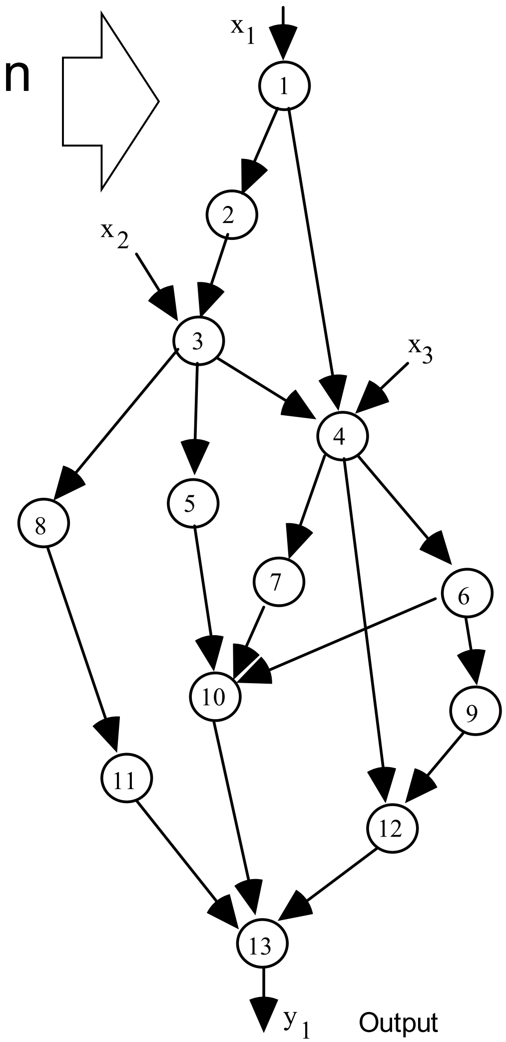


Fig. 20.1 Automatic synchronization in message-passing systems.

Synchronization with Shared Memory

Accomplished by accessing specially designated shared control variables

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

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

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

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

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

Barrier Synchronization: Implementations

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

Software implementation via fetch-and-add or similar instruction

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

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

Solution: Use two AND trees for alternating barrier points

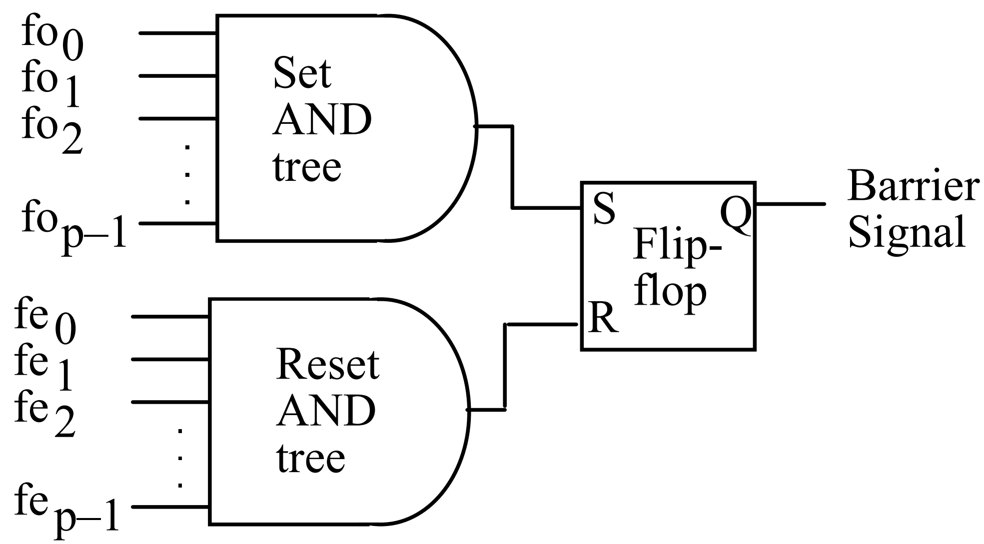


Fig. 20.4 Example of hardware aid for fast barrier synchronization [Hoar96].

6C.2 Strict and Sequential Consistency

A global notion of time does not exist: The speed of light is finite; therefore, we do not become aware of events instantaneously

Suppose a group of people decide to synchronize their watches

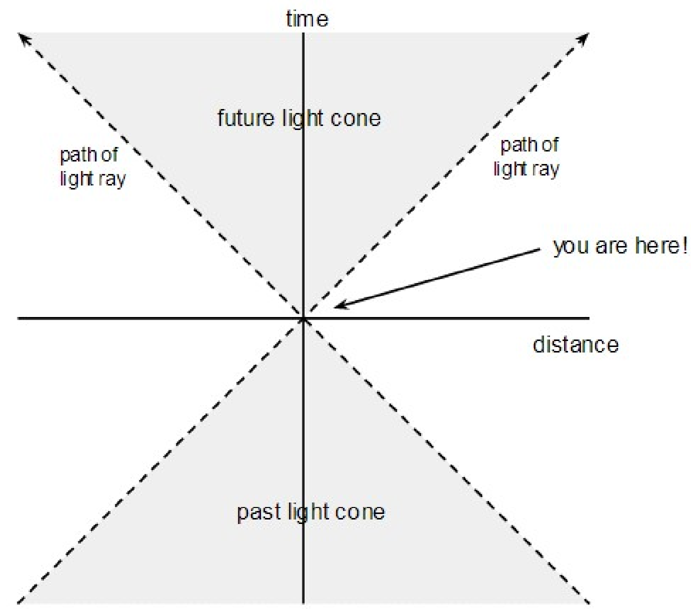
One person shouts: On the count of 3, set your watches to 1:00 PM

Not everyone will hear the command at the same time

In physics, we learn that two observers do not see the same time

Furthermore, the speed of time passing varies for different observers

Conclusion: A universal notion of time does not exist!



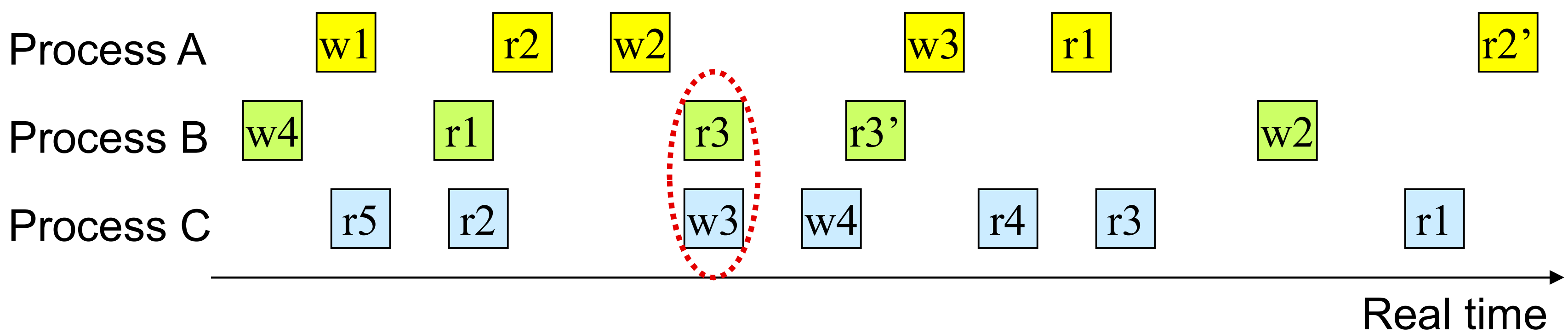
Strict Consistency

With strict consistency, a read operation always returns the result of the latest write operation on that data object

Strict consistency is impossible to maintain in a distributed system which does not have a global clock

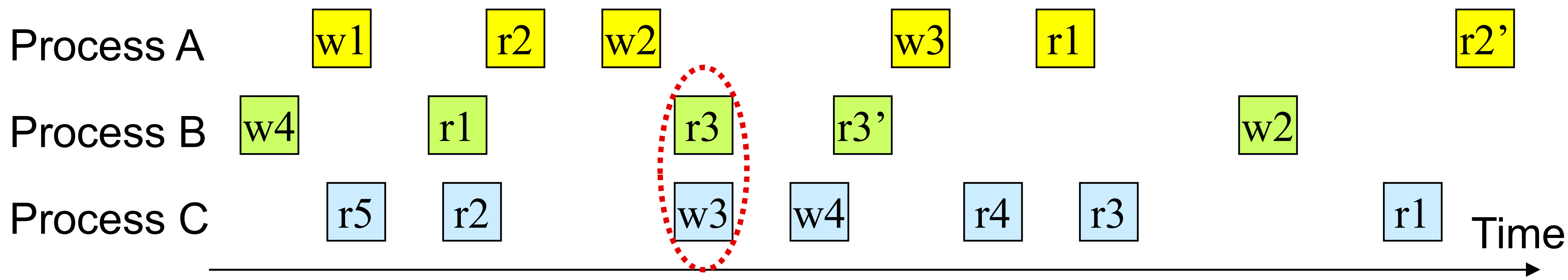
While clocks can be synchronized, there is always some error that causes trouble in near-simultaneous operations

Example: Three processes sharing variables 1-4 (r = read, w = write)



Sequential Consistency

Sequential consistency (original def.): The result of any execution is the same as if processor operations were executed in some sequential order, and the operations of a particular processor appear in the sequence specified by the program it runs



A possible ordering

w4	w1	r1	r5	r2	r2	w2	r3	r3’	w3	w4	w3	r4	w2	r1	r3	r1	r2

A possible ordering

w4	r5	r2	w1	r2	r1	w2	r3	w3	w4	w3	r4	r1	r3’	r3	r1	r2	w2

Sequential consistency (new def.): Write operations on the same data object are seen in exactly the same order by all system nodes

The Performance Penalty of Sequential Consistency

Initially	Thread 1	Thread 2
X = Y = 0	**X := 1**	**Y := 1**
	R1 := Y	**R2 :=X**

Exec 1	Exec 2	Exec 3
X := 1	**Y := 1**	**X := 1**
R1 := Y	**R2 :=X**	**Y := 1**
Y := 1	**X := 1**	**R1 := Y**
R2 := X	**R1 := Y**	**R2 := X**

If a compiler reorders the seemingly independent statements in Thread 1, the desired semantics (R1 and R2 not being both 0) is compromised

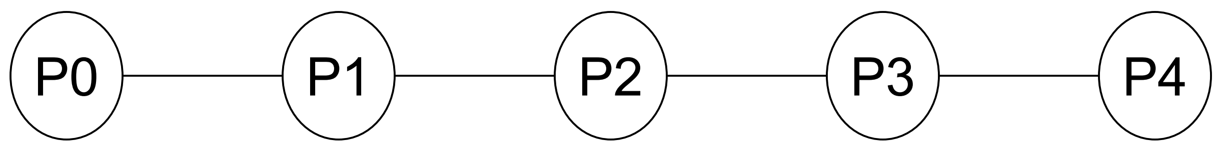
Relaxed consistency (memory model): Ease the requirements on doing things in program order and/or write atomicity to gain performance

When maintaining order is absolutely necessary, we use synchronization primitives to enforce it

6C.3 Processor Consistency

Processor consistency: Writes by the same processors are seen by all other processors as occurring in the same order; writes by different processors may appear in different order at various nodes

Example: Linear array in which changes in values propagate at the rate of one node per time step

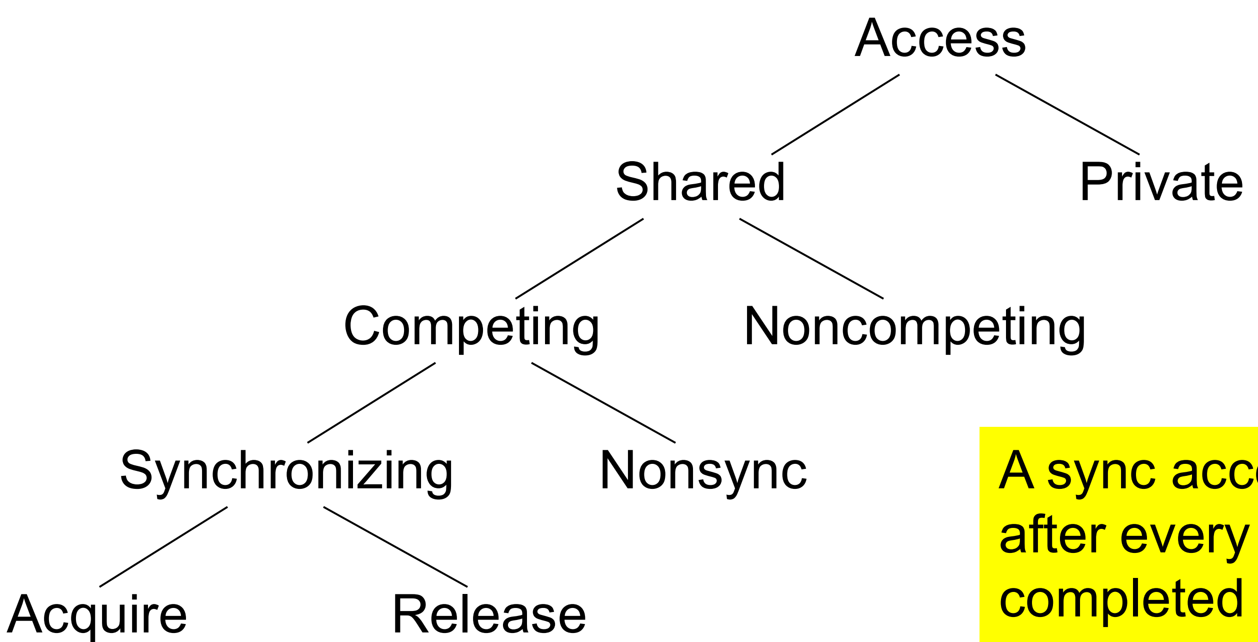


If P0 and P4 perform two write operations on consecutive time steps, then this is how the processors will see them

Step 1	WA	--	--	--	WX
Step 2	WB	WA	--	WX	WY
Step 3	--	WB	WA, WX	WY	--
Step 4	--	WX	WB, WY	WA	
Step 5	WX	WY	--	WB	WA
Step 6	WY	--	--	--	WB

6C.4 Weak or Synchronization Consistency

Weak consistency: Memory accesses are divided into two categories:
(1) Ordinary data accesses (2) Synchronizing accesses
Category-1 accesses can be reordered with no limitation
If ordering of two operations is to be maintained, the programmer must specify at least one of them as a synchronizing, or Category-2, access



A sync access is performed after every preceding write has completed and before any new data access is allowed

6C.5 Other Memory Consistency Models

Release consistency: Relaxes synchronization consistency somewhat
(1) A process can access a shared variable only if all of its previous acquires have completed successfully
(2) A process can perform a release operation only if all of its previous reads and writes have completed
(3) Acquire and release accesses must be sequentially consistent

For more on memory consistency models, see:

Adve, S. V. and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, Dec. 1996.

Adve, S. V., H.-J. Boehm, "Memory Models: A Case for Rethinking Parallel Languages and Hardware," *Communications of the ACM*, Aug. 2010.

For general info on memory management, see:

Gaud, F. *et al.*, "Challenges of Memory Management on Modern NUMA Systems," *Communications of the ACM*, Dec. 2015.

6C.6 Transactional Memory

TM systems typically provide atomic statements that allow the execution of a block of code as an all-or-nothing entity (much like a transaction)

Example of transaction: Transfer $x from account A to Account B

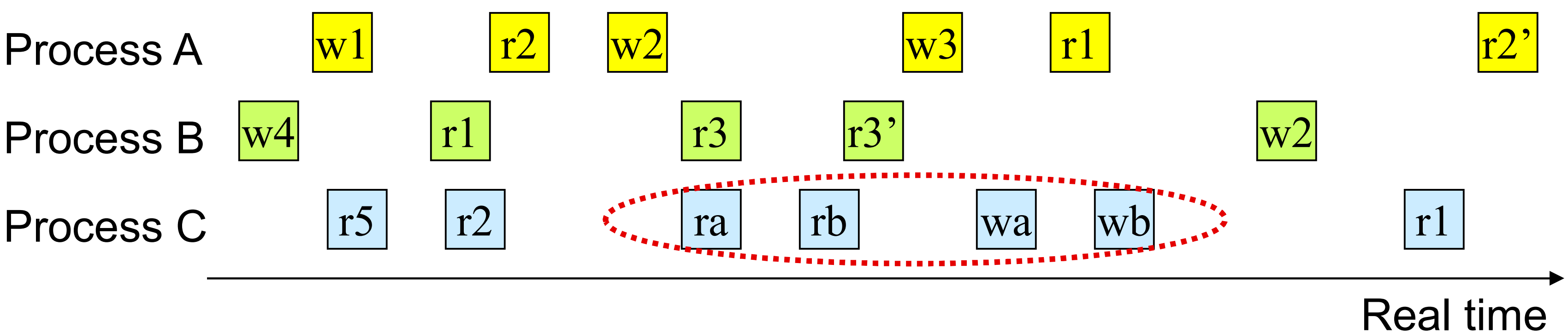
(1) if $a \geq x$
  then $a := a - x$
  else return "insufficient funds"
(2) $b := b + x$
(3) return "transfer successful"

TM allows a group of read & write operations to be enclosed in a block, so that any changed values become observable to the rest of the system only upon the completion of the entire block

Examples of Memory Transactions

A group of reads, writes, and intervening operations can be grouped into an atomic transaction

Example: If wa and wb are made part of the same memory transaction, every processor will see both changes or neither of them



Implementations of Transactional Memory

Software: 2-7 times slower than sequential code [Laru08]

Hardware acceleration: Hardware assists for the most time-consuming parts of TM operations

e.g., maintenance and validation of read sets

Hardware implementation: All required bookkeeping operations are implemented directly in hardware

e.g., by modifying the L1 cache and the coherence protocol

For more information on transactional memory, see:

[Laru08] Larus, J. and C. Kozyrakis, "Transactional Memory," *Communications of the ACM*, Vol. 51, No. 7, pp. 80-88, July 2008.