UNIT-3

PARALLEL MODELS & MESH-BASED ARCHITECTURES

RAM (RANDOM ACCESS MACHINE) MODEL

It is a model of a sequential computer.
Its main features are:

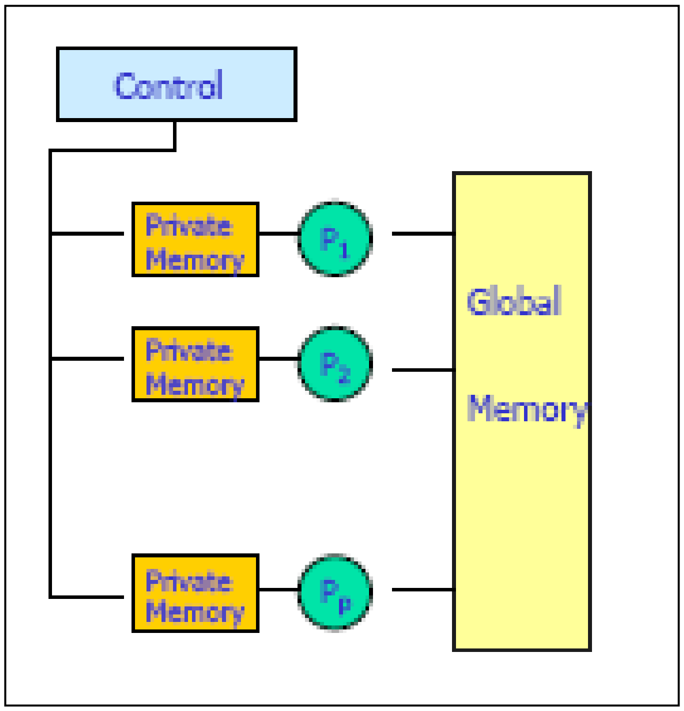
- Computation unit with a user defined program.
- Read-only input tape and write-only output tape.
- Unbounded number of local memory cells.
- Each memory cell is capable of holding an integer of unbounded size.
- Instruction set includes operations for moving data between memory cells, comparisons and conditional branches, and simple arithmetic operations.
- Execution starts with the first instruction and ends when a HALT instruction is executed.
- All operations take unit time regardless of the lengths of operands.
- Time complexity = the number of instructions executed.
- Space complexity = the number of memory cells accessed.

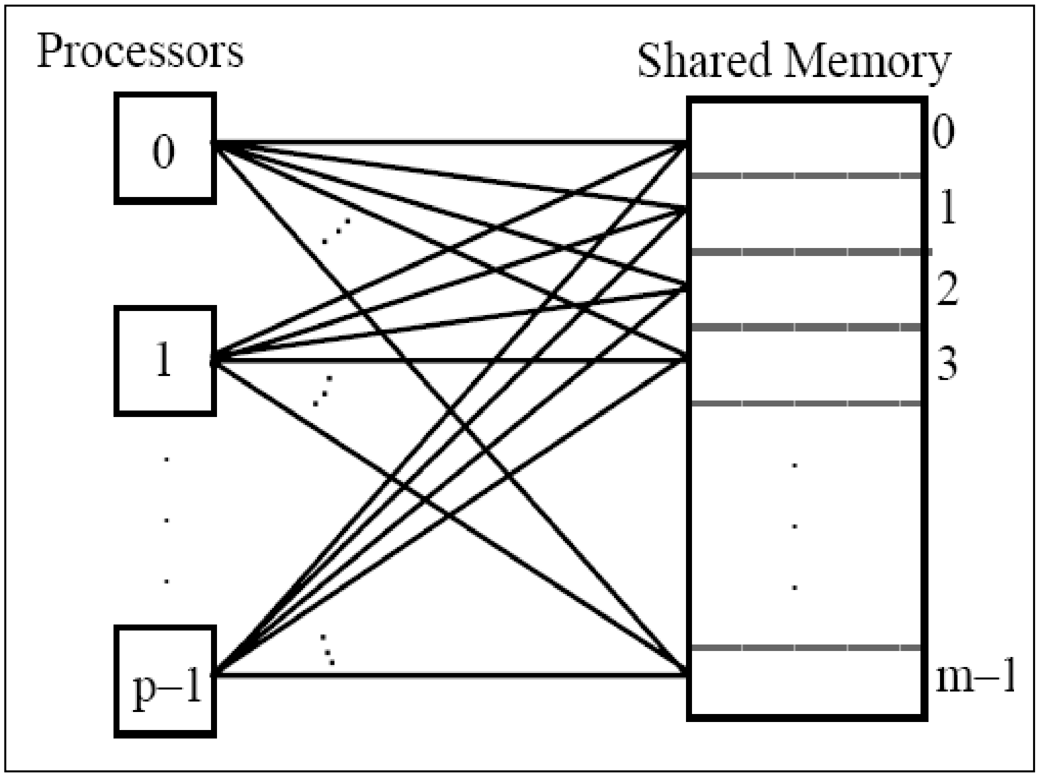
PRAM (PARALLEL RANDOM ACCESS MACHINE) MODEL

It is a straightforward and natural generalization of RAM
It is an idealized model of a shared memory SIMD machine.
The PRAM is one of the earliest and most widely studied parallel models of computation. However, it is important to realize that the PRAM is not a physically realizable machine. Although a machine with PRAM-type characteristics can be built with relatively few processors, such a machine could not be built with an extremely large number of processors
Its main features are:

- Unbounded collection of numbered RAM processors Pi (P0, P1, P2,..). (without tapes).
- Unbounded collection of shared memory cells M[0], M[1], M[2],....
- Each processor Pi has its own (unbounded) local memory (registers) and knows its index i.
- Each processor Pi can access any shared memory cell (unless there is an access conflict) in unit time.
- Input of a PRAM algorithm consists of n items stored in (usually the first) n shared memory cells.
- Output of a PRAM algorithm consists of n' items stored in n' shared memory cells.
- PRAM instructions execute in 3-phase cycles.
    - Read (if any) from a shared memory cell.
    - Local computation (if any).
    - Write (if any) to a shared memory cell.
- Processors execute these 3-phase PRAM instructions synchronously.
- Special assumptions have to be made about R-R and W-W shared memory access conflicts.
- The only way processors can exchange data is by writing into and reading from memory cells.
- P0 has a special activation register specifying the maximum index of an active processor. Initially, only P0 is active, it computes the number of required active processors and loads this register, and then the other corresponding processors start executing their programs.
- Computation proceeds until P0 halts, at which time all other active processors are halted.
- Parallel time complexity = the time elapsed for P0's computation.
- Space complexity = the number of shared memory cells accessed.

Basic PRAM model





PRAM models

To make the PRAM model realistic and useful, some mechanism has to be defined to resolve read and write access conflicts to the same shared memory cell.

Exclusive Read Exclusive Write (EREW) PRAM: No two processors are allowed to read or write the same shared memory cell simultaneously.

Concurrent Read Exclusive Write (CREW) PRAM: Simultaneous reads of the same memory cell are allowed, but only one processor may attempt to write to an individual cell

Concurrent Read Concurrent Write (CRCW) PRAM: Both simultaneous reads and both simultaneous writes of the same memory cell are allowed.

Concurrent Read has a clear semantics, whereas Concurrent Write has to be further constrained. There exist several basic sub-models:

PRIORITY CRCW: the processors are assigned fixed distinct priorities and the processor with the highest priority is allowed to complete the WRITE operation.

ARBITRARY CRCW: one randomly chosen processor is allowed to complete the WRITE operation. The algorithm may make no assumptions about which processor was chosen.

COMMON CRCW: all processors are allowed to complete the WRITE operation if and only if all the values to be written are equal.

Any algorithm for this model has to make sure that above conditions are satisfied.

BASIC PRAM ALGORITHMS

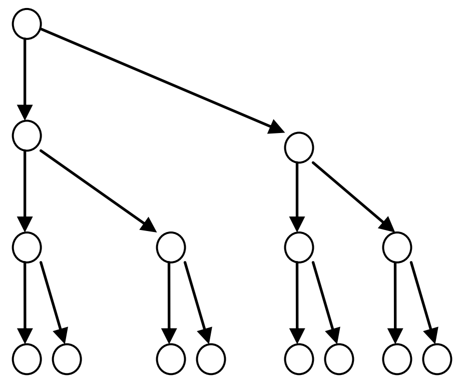
The PRAM algorithms have two basic phases:

- Initially only a single processor is active and which activates sufficient number of processors so that the computations can be done.
- The activated processors perform the computations in parallel.

Generally it is observed that (log p) activation steps are required to activate p number of processors as one instruction can activate two processors.

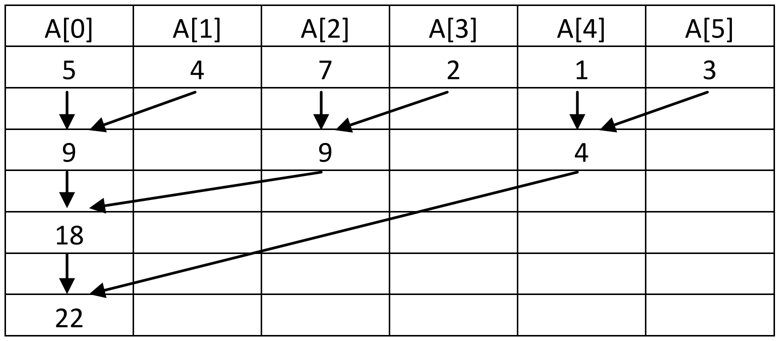
This activation processes follows a binary tree like structure.

Processor Activation in PRAM



Parallel Reduction:
In some algorithms the data flows from root to the leaves and they are called broadcast algorithms.
In some algorithms the tree represents the recursive subdivision of a problem into sub-problems.
In some algorithms the data flows from leaves to the root and these are called Reduction operations (Fan-in operations).
Parallel summation is an example of parallel reduction process. Here the PRAM manipulates the data stored in the global registers.
For example we want to find the sum of 6 numbers and the numbers are stored in an array A[0] to A[5]. The processors in the leaf perform the sum and forward the result to next upper level of the processors in the tree and so-on.

Parallel Reduction



Prefix Sums:
Given a set of n values $a_1, a_2, \ldots, a_n$ and a binary operator @. The @ operator can assume any operation like addition, subtraction etc. The prefix sum would be to calculate the values:
$S_1 = a_1$
$S_2 = a_1 @ a_2$
$S_3 = a_1 @ a_2 @ a_3$
..
..
$S_n = a_1 @ a_2 @ a_3 \ldots. @ a_n$

Algorithm for Parallel Prefix sum:

```
For j = 1 to (log n) do
    For i = 1 to n-1 do in parallel
        $S_i = S_{(i-2^j)} + S_i$
    End For
End For
```

Parallel Algorithm Design:
Algorithm development is a critical component of problem solving using computers.
Dividing a computation into smaller computations and assigning them to different processors for parallel execution are the two key steps in the design of parallel algorithms.
A parallel algorithm has the added dimension of concurrency and the algorithm designer must specify the steps that can be executed simultaneously.
The following issues need to be addressed while designing parallel algorithms:

- Identifying portions of the work that can be performed concurrently.
- Mapping the concurrent pieces of work onto multiple processes running in parallel.
- Distributing the input, output, and intermediate data associated with the program.
- Managing accesses to data shared by multiple processors.
- Synchronizing the processors at various stages of the parallel program execution.

In the context of parallel algorithm design, processes are logical computing agents that perform tasks. Processors are the hardware units that physically perform computations.

Granularity:
The number and size of tasks into which a problem is decomposed determines the granularity of the decomposition.
A decomposition into a large number of small tasks is called fine-grained decomposition.
A decomposition into a small number of large tasks is called coarse-grained decomposition.

Decomposition Techniques:
One of the fundamental steps that we need to undertake to solve a problem in parallel is to split the computations to be performed into a set of tasks for concurrent execution.
These techniques are broadly classified as

- Recursive decomposition,
- Data-decomposition,
- Exploratory decomposition, and
- Speculative decomposition.

The recursive- and data decomposition techniques are relatively general purpose as they can be used to decompose a wide variety of problems.
On the other hand, speculative- and exploratory-decomposition techniques are more of a special purpose nature because they apply to specific classes of problems.

Recursive Decomposition:
Recursive decomposition is a method of inducing concurrency in problems that can be solved using the divide-and-conquer strategy.
In this technique, a problem is solved by first dividing it into a set of independent subproblems.
Each one of these subproblems is solved by recursively applying a similar division into smaller subproblems followed by a combination of their results.
The divide-and-conquer strategy results in natural concurrency, as different subproblems can be solved concurrently.

Data Decomposition:
Data decomposition is a powerful and commonly used method for deriving concurrency in algorithms that operate on large data structures.
In this method, the decomposition of computations is done in two steps.

- In the first step, the data on which the computations are performed is partitioned, and
- In the second step, this data partitioning is used to induce a partitioning of the computations into tasks.

The operations that these tasks perform on different data partitions are usually similar or are chosen from a small set of operations.
Following techniques are used while data decomposition:

- **Partitioning Output Data:** In many computations, each element of the output can be computed independently of others as a function of the input. In such computations, a partitioning of the output data automatically induces a decomposition of the problems into tasks, where each task is assigned the work of computing a portion of the output. (Example: matrix multiplication)
- **Partitioning Input Data:** It is sometimes possible to partition the input data, and then use this partitioning to induce concurrency. A task is created for each partition of the input data and this task performs as much computation as possible using these local data. (Example sorting using partitioning)

Exploratory decomposition:
It is used to decompose problems whose underlying computations correspond to a search of a space for solutions. In exploratory decomposition, we partition the search space into smaller parts, and search each one of these parts concurrently, until the desired solutions are found.

Speculative decomposition:
It is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it. In this situation, while one task is performing the computation whose output is used in deciding the next computation, other tasks can concurrently start the computations of the next stage.

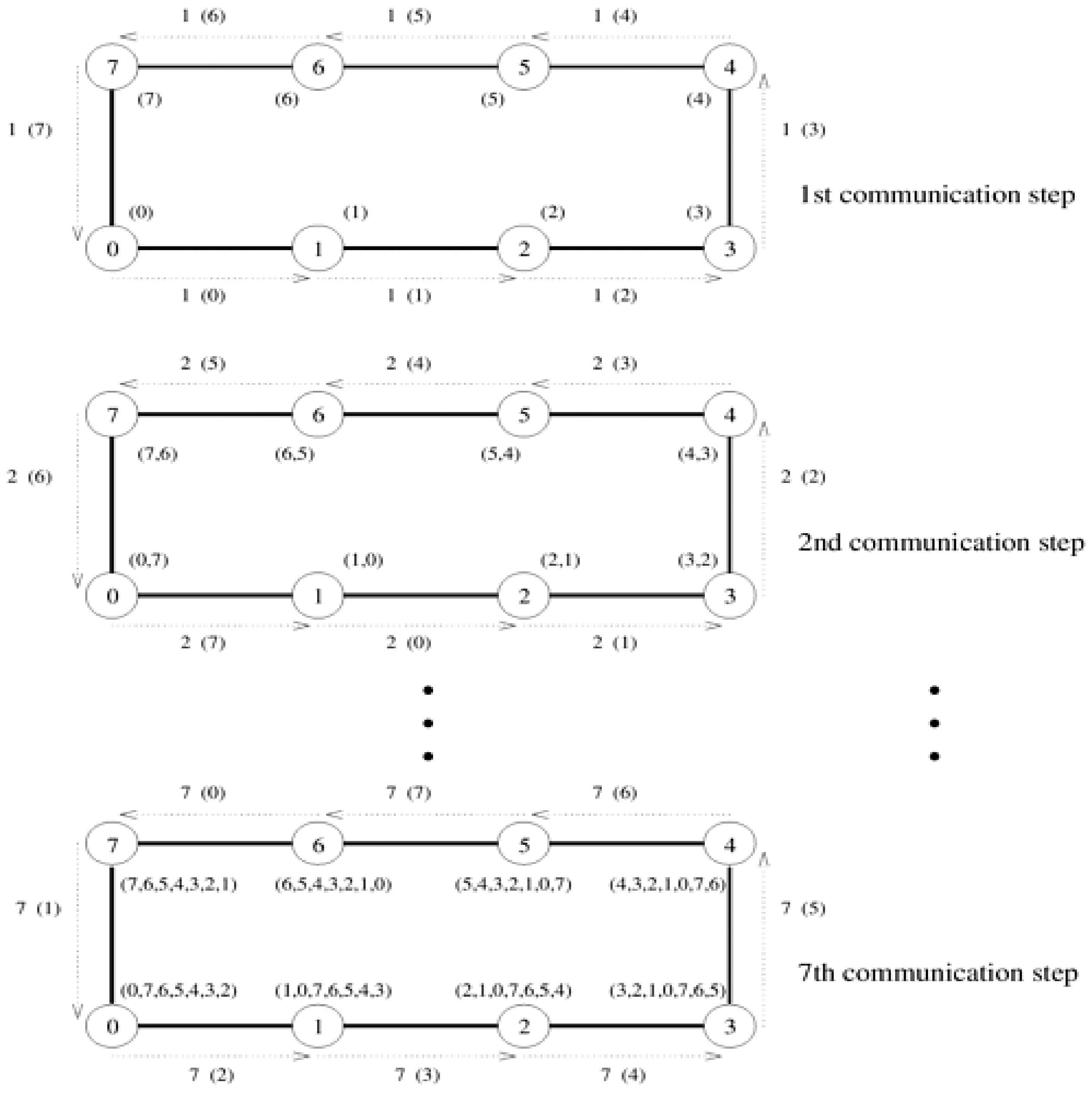
Broadcast and Reduction Models:

Broadcast:
Parallel algorithms often require a single process to send identical data to all other processes or to a subset of them. This operation is known as one-to-all broadcast . All-to-all broadcast is a generalization of one-to-all broadcast in which all p nodes simultaneously initiate a broadcast. A process sends the same m-word message to every other process, but different processes may broadcast different messages. All-to-all broadcast is used in matrix operations, including matrix multiplication and matrix-vector multiplication.
The various models built on this concept are:

- Ring or Linear Array
- 2D Mesh
- Hypercube
- Balanced Binary Tree

Ring or Linear Array:
While performing all-to-all broadcast on a linear array or a ring, all communication links can be kept busy simultaneously until the operation is complete because each node always has some information that it can pass along to its neighbor. Each node first sends to one of its neighbors the data it needs to broadcast. In subsequent steps, it forwards the data received from one of its neighbors to its other neighbor.

Ring or Linear Array:



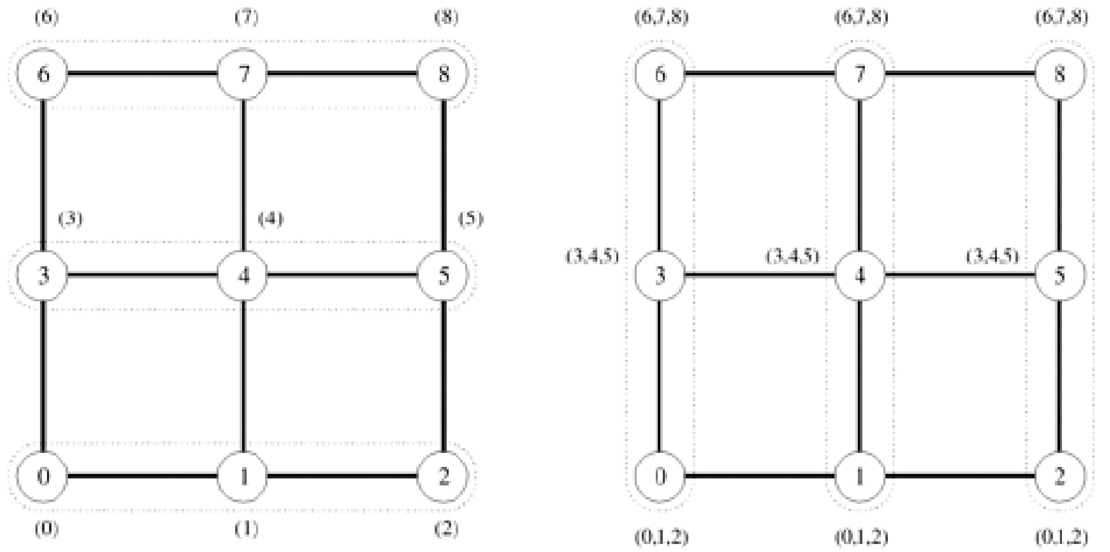
2D Mesh:

The all-to-all broadcast algorithm for the 2-D mesh is based on the linear array algorithm, treating rows and columns of the mesh as linear arrays.

The communication takes place in two phases.

- In the first phase, each row of the mesh performs an all-to-all broadcast using the procedure for the linear array. In this phase, all nodes collect √P messages corresponding to the √P nodes of their respective rows. Each node consolidates this information into a single message of size (m √P), and proceeds to the second communication phase of the algorithm.
- The second communication phase is a column wise all-to-all broadcast of the consolidated messages. By the end of this phase, each node obtains all P pieces of m-word data that originally resided on different nodes.

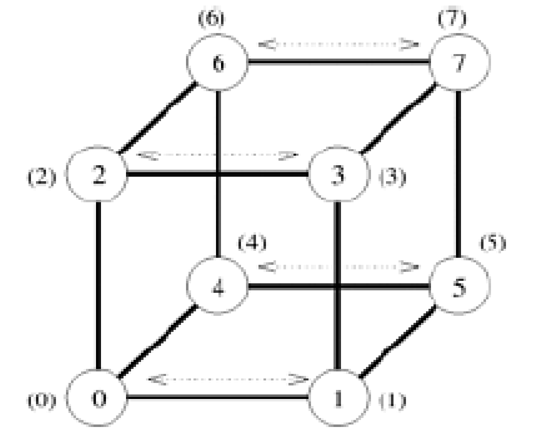
2D Mesh broadcasting



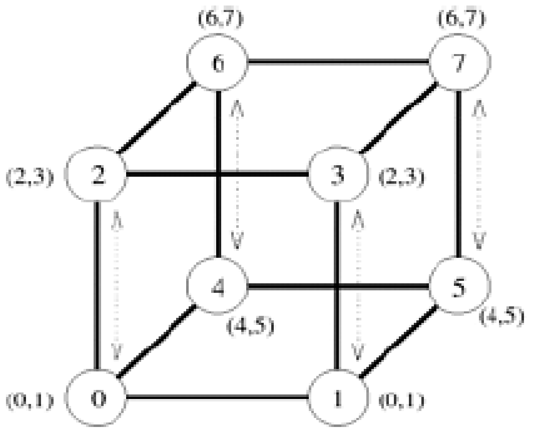
(a) Initial data distribution (b) Data distribution after rowwise broadcast

Hypercube:
The hypercube algorithm for all-to-all broadcast is an extension of the mesh algorithm to log p dimensions.
The procedure requires log p steps.
Communication takes place along a different dimension of the p-node hypercube in each step.
In every step, pairs of nodes exchange their data and double the size of the message to be transmitted in the next step by concatenating the received message with their current data.

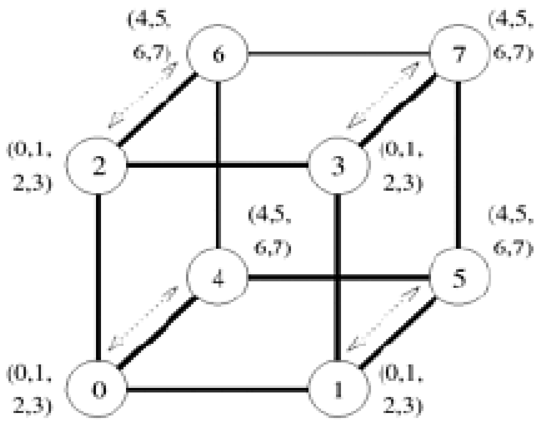
Hypercube broadcasting



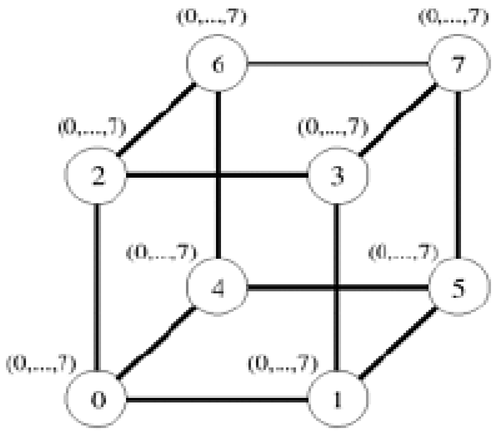
(a) Initial distribution of messages



(b) Distribution before the second step



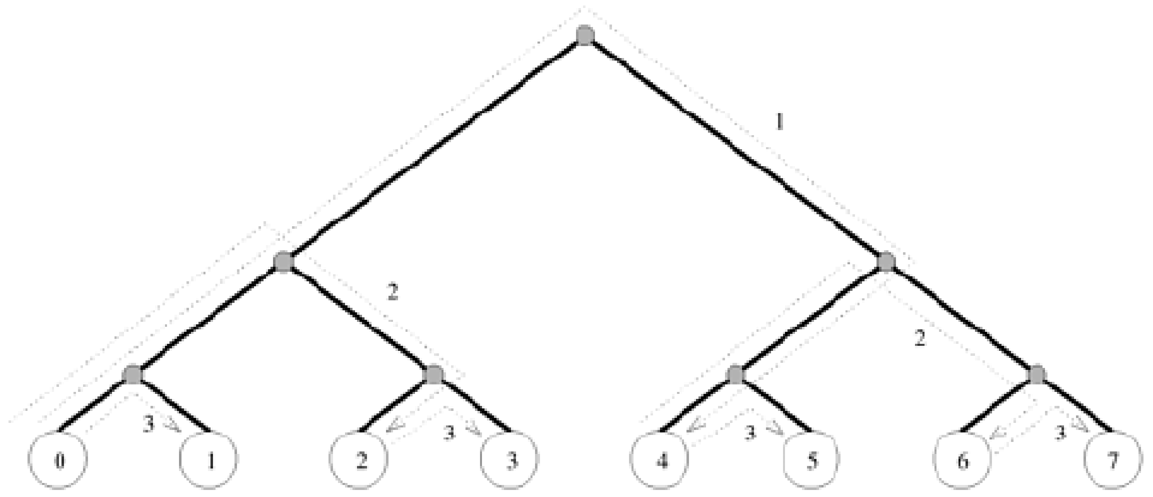
(c) Distribution before the third step



(d) Final distribution of messages

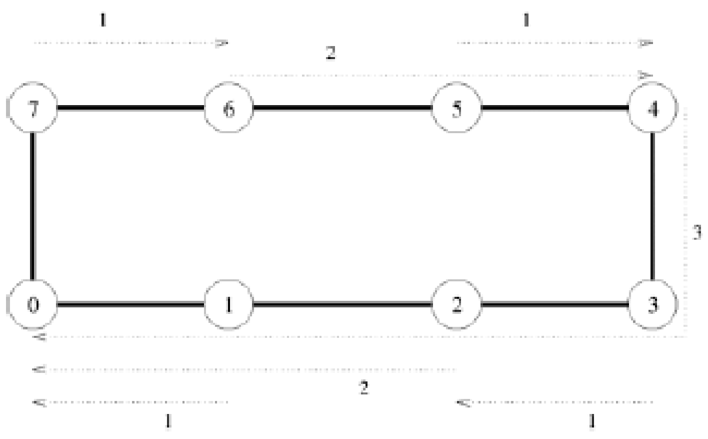
Balanced Binary Tree:
The hypercube algorithm for one-to-all broadcast maps naturally onto a balanced binary tree in which each leaf is a processing node and intermediate nodes serve only as switching units.
There is no congestion on any of the communication links at any time.

Balanced Binary Tree



Reduction:
The opposite of broadcast is reduction. In reduction each of the P participating processes starts with a buffer M containing **m** words. The data from all processes are combined through an associative operator and accumulated at a single destination process into one buffer of size **m** .
Reduction can be used to find the sum, product, maximum, or minimum of sets of numbers.
In all-to-all reduction each node starts with P messages, each of the messages are destined to be accumulated at a distinct node. Upon receiving a message, a node must combine the received message with the local copy of its own message that has the same destination as the received message and forward the combined message to the next neighbor.

Reduction in Ring



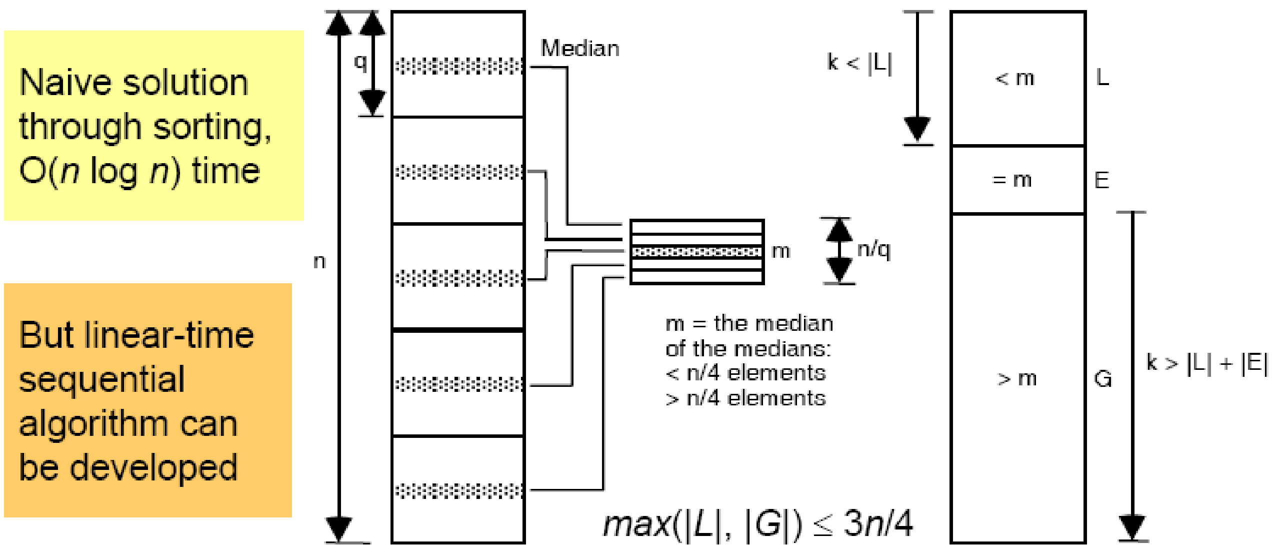
Rank Based Algorithm:
Rank-based selection is the problem of finding a (the) kth smallest element in a sequence S = x0 , x 1 , . . . , x n -1 whose elements belong to a linear order. Median, maximum, and minimum finding are special cases of this general problem.

Sequential rank-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
```



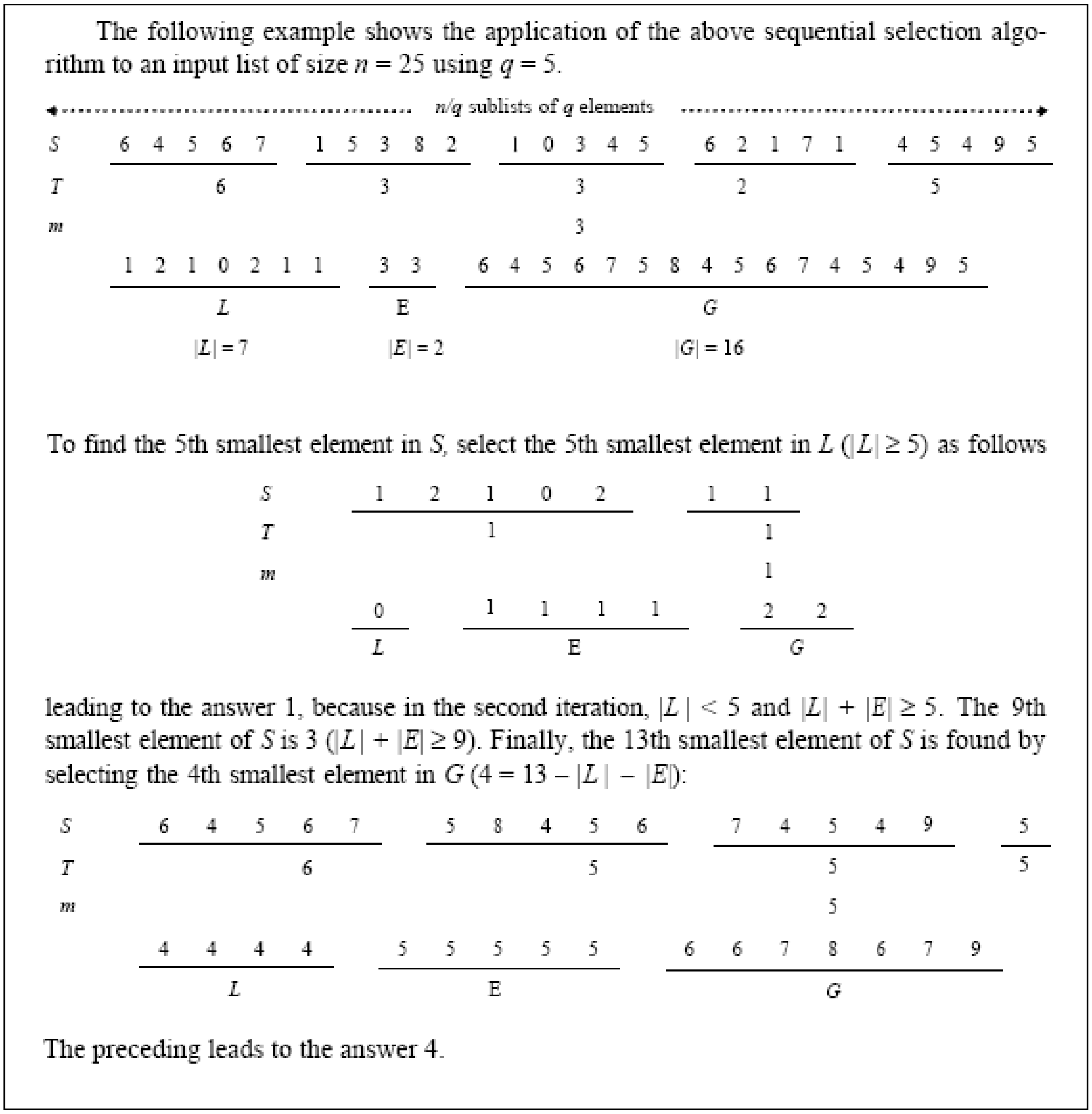
Naive solution through sorting, O(*n* log *n*) time

But linear-time sequential algorithm can be developed

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

```
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 $q$
                Sort each subsequence and find its median
                Let the $|S|/q$ medians form the sequence $T$
        Endif
2. $m$ = select($T$, $|T|/2$) {find the median $m$ of the $|S|/q$ medians}
3. Create 3 subsequences
        $L$:Elements of $S$ that are $< m$
        $E$:Elements of $S$ that are $= m$
        $G$:Elements of $S$ that are $> m$
4. If $|L| \geq k$
        Then
                return select($L$, $k$)
        Else if $|L| + |E| \geq k$
                Then return $m$
                Else
                        return select($G$, $k - |L| - |E|$)
        Endif
    Endif
```

If S is small, then its median is found through sorting in Step 1 (the size threshold constant q will be defined later). This requires constant time, say c0 . Otherwise, we divide the list into a number of subsequences of length q, sort each subsequence to find its median, and put the medians together into a list T.
Step 2 of the algorithm that finds the median of the medians constitutes a smaller, |S |/q-input, selection problem. Given the median m of the medians,
Step 3 of the algorithm involves scanning the entire list, comparing each element to m, and putting it in one of three output lists according to the comparison result. Finally,
Step 4 is another smaller selection problem.



The following example shows the application of the above sequential selection algorithm to an input list of size $n = 25$ using $q = 5$.

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

To find the 5th smallest element in S, select the 5th smallest element in L ($|L| \geq 5$) as follows

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

leading to the answer 1, because in the second iteration, $|L| < 5$ and $|L| + |E| \geq 5$. The 9th smallest element of S is 3 ($|L| + |E| \geq 9$). Finally, the 13th smallest element of S is found by selecting the 4th smallest element in G ($4 = 13 - |L| - |E|$):

```
S     6  4  5  6  7     5  8  4  5  6     7  4  5  4  9     5
      -------------     -------------     -------------     -
T           6                 5                 5           5
m                                               5
      4  4  4  4     5  5  5  5  5     6  6  7  8  6  7  9
      ----------     -------------     -------------------
          L                E                    G
```

The preceding leads to the answer 4.

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

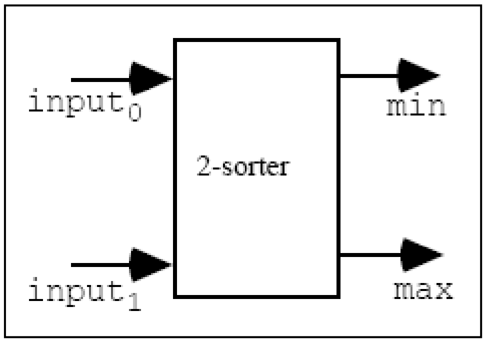
```
1. If $|S| < 4$
        Then
                sort $S$ and return the $k$th smallest element of $S$
        Else
                broadcast $|S|$ to all $p$ processors
                divide $S$ into $p$ subsequences $S(j)$ of size $|S|/p$
                Processor $j$, $0 \le j < p$, compute $Tj := select(S(j), |S(j)|/2)$
        EndIf

2. $m = PRAMselect(T, |T|/2, p)$ {median of the medians}
3. Broadcast $m$ to all processors and create 3 subsequences
        $L$: Elements of $S$ that are $< m$
        $E$: Elements of $S$ that are $= m$
        $G$: Elements of $S$ that are $> m$
4. If $|L| \ge k$
        Then return $PRAMselect(L, k, p)$
        Elseif $|L| + |E| \ge k$
                Then return $m$
                Else return $PRAMselect(G, k - |L| - |E|, p)$
        Endif
   Endif
```

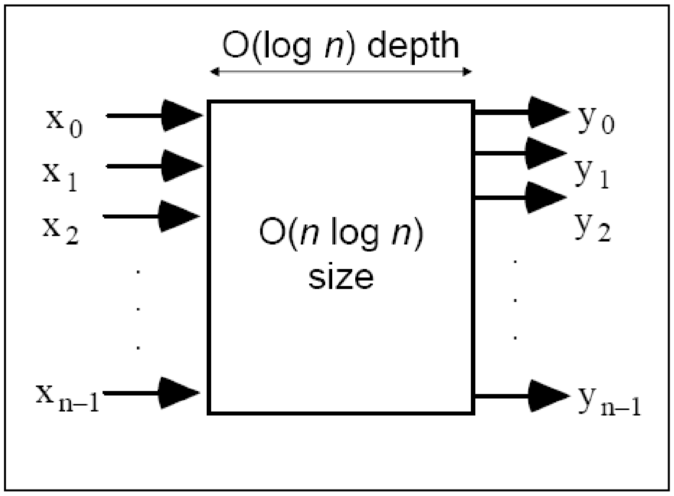
SORTING NETWORKS:

These networks sort the input and give output in ascending or descending order depending upon the design.

A sorting network is a circuit that receives *n* inputs, , and permutes them to produce *n* outputs, such that the outputs satisfy $y0 \le y\ 1 \le y2 \le . . . yn-1$.

Sorting Networks

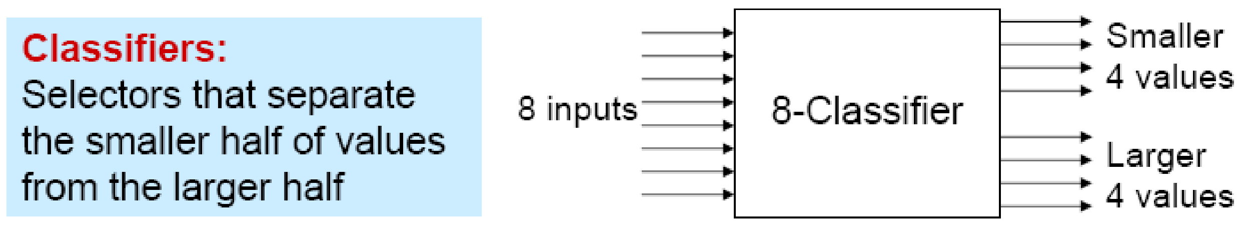




SELECTION NETWORKS:

The most common use of a selection network is to use it as a classifier

Selection Network as Classifier:


Classifiers:
Selectors that separate the smaller half of values from the larger half
8 inputs
8-Classifier
Smaller 4 values
Larger 4 values

TREE STRUCTURED DICTIONARY MACHINE:

The tree machine consists of two back-to-back complete binary trees whose leaves have been merged. The "circles" in the tree is responsible for broadcasting the dictionary Operations that enter via the "input root" to all of the leaf nodes that hold the records. The "triangles" in the tree combines the results of individual operations by the leaf nodes into an overall Result that emerges from the "output root."

Dictionary Operations:

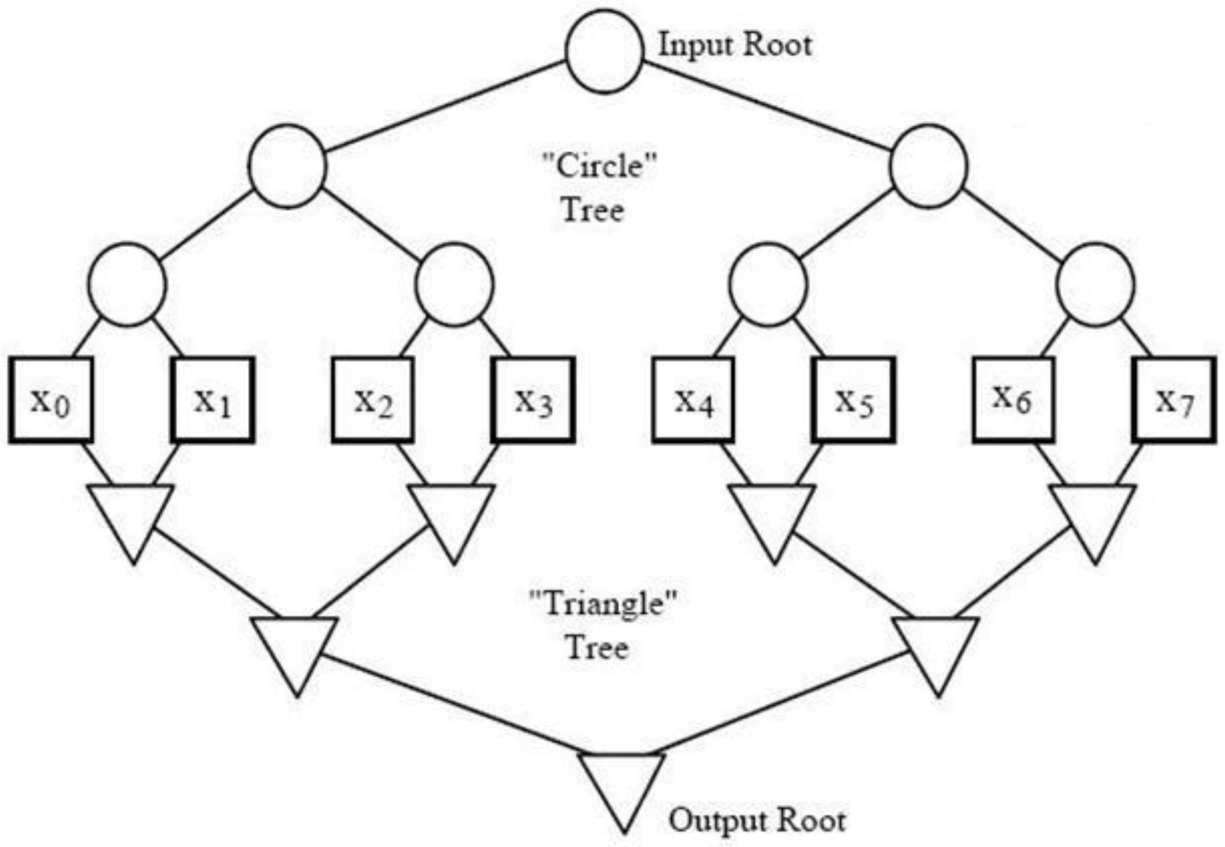
Basic dictionary operations: record keys *x*0, *x*1, . . . , *xn*–1

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

Additional operations include:

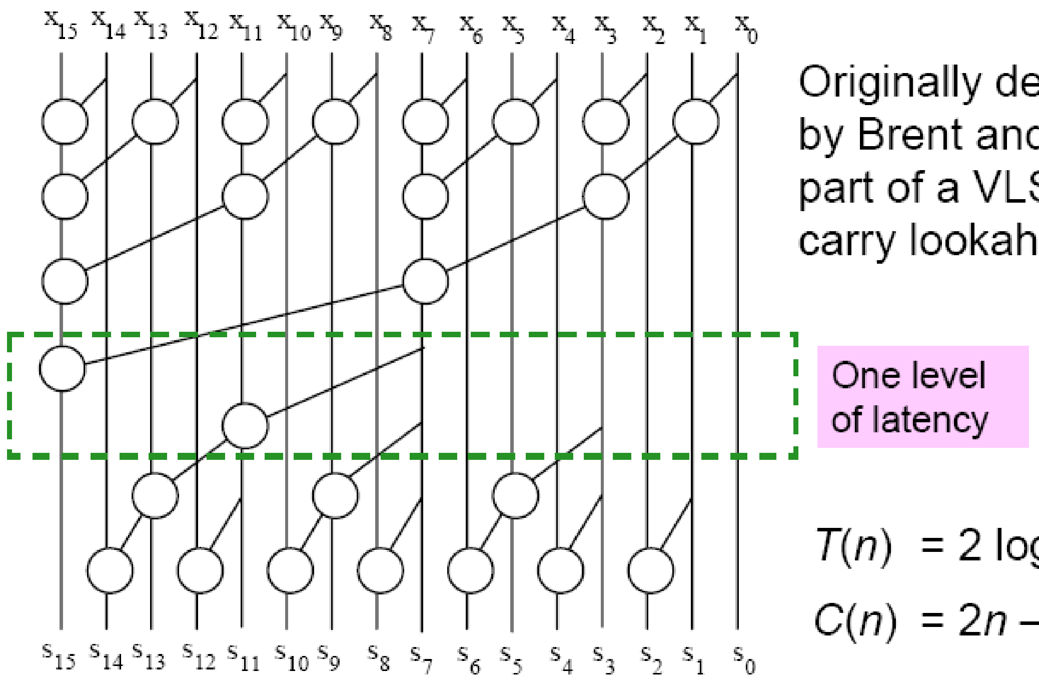
*findmin*Find record with smallest key; return data
*findmax*Find record with largest key; return data
*findmed*Find record with median key; return data
findbest(*y*)Find record with key "nearest" to *y*
findnext(*y*)Find record whose key is right after *y*in sorted order
findprev(*y*)Find record whose 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

Tree Structured Dictionary Machine:



PARALLEL PREFIX NETWORK (Brent and Kung Parallel Prefix Network):

Parallel Prefix Network:


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

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

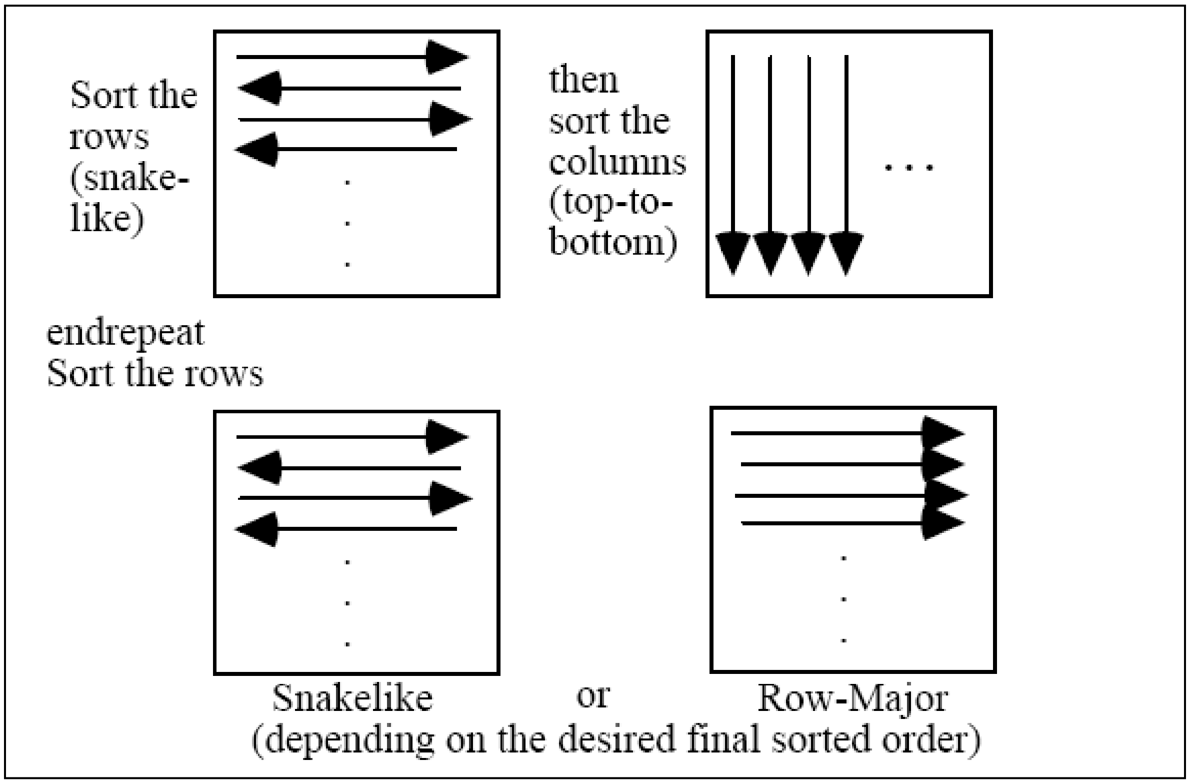
$$T(n) = 2\log_2 n - 2$$

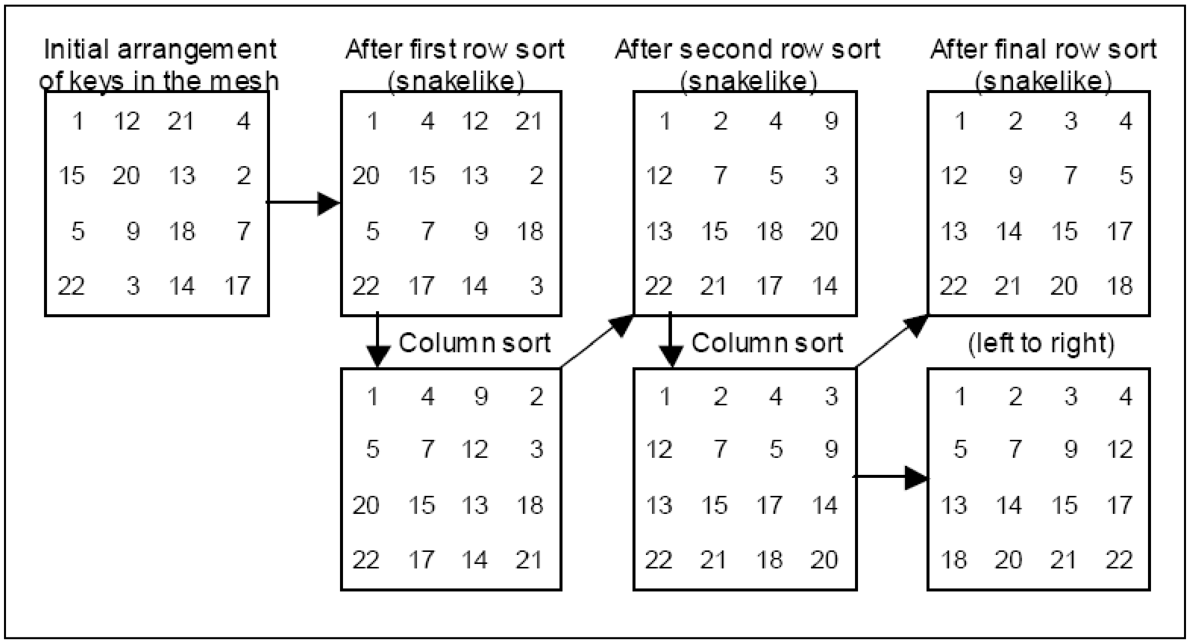
$$C(n) = 2n - 2 - \log_2 n$$

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

SORTING ON 2-D MESH OR TORUS:

Shear Sort:


Sort the rows (snake-like)
then sort the columns (top-to-bottom)
endrepeat
Sort the rows
Snakelike
or
Row-Major
(depending on the desired final sorted order)



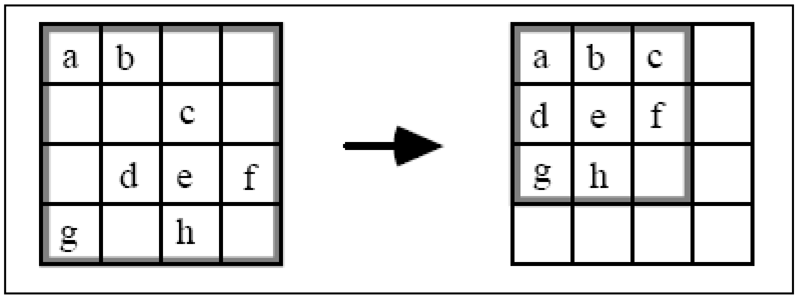
Time Complexity:

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

ROUTING ON 2D-MESH:

Types of Data Routing Operations:

- Point-to-point communication: one source, one destination
- Collective communication:
  - One-to-many: multicast, broadcast (one-to-all), scatter
  - Many-to-one: combine (fan-in), global combine, gather, reduction.
  - Many-to-many: all-to-all broadcast (gossiping), scatter-gather
- Data Compaction or packing:



GREEDY ROUTING

The Greedy strategy is trying to make the most progress towards the solution, based on current conditions or information available.

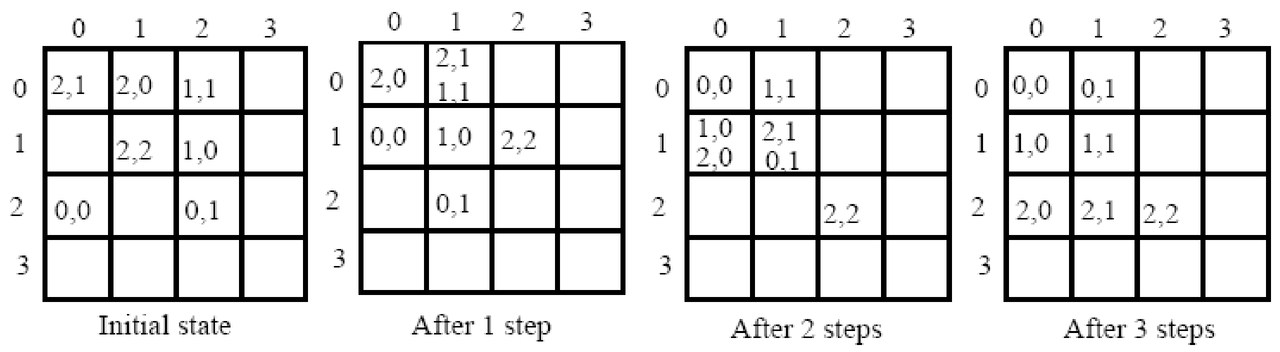
A greedy routing algorithm is one that tries to reduce the distance of a packet from its destination with every routing step. The term greedy refers to the fact that such an algorithm only considers local short-term gains as opposed to the global or long-term effects of each routing decision. The simplest greedy algorithm is dimension-ordered routing or e-cube routing.

Algorithm for Greedy row-first routing on a 2D mesh:

```
If      the packet is not in the destination column
Then
        route it along the row toward the destination column
        {processors have buffers to hold the incoming messages.}
Else
        route it along the column toward the destination node
        {of the messages that need to use an upward or downward link, the one that needs to go farthest
        along the column goes first.}
Endif
```

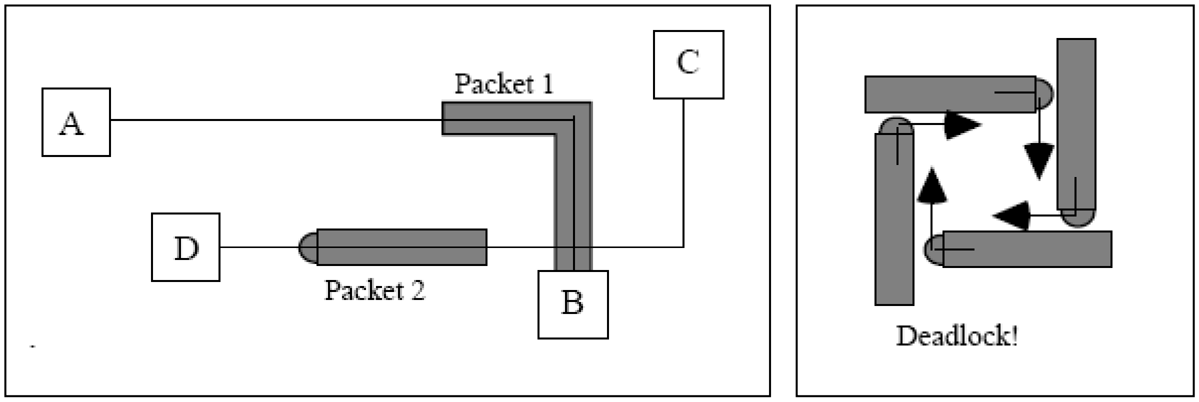
Example: dimension-ordering in a cube
In the initial state, all packets, except the one in Row 2, Column 0, have to move to a different column. Thus, they are routed horizontally toward their destination columns. The packet destined for Processor (0, 0) starts its column routing phase right away, moving to Row 1, Column 0, in the first step. Similarly, other packets begin their column routing phases as soon as each arrives into its destination column.
If the buffer space in each processor is unlimited, the greedy row-first algorithm is optimal.

Dimension-ordering in a cube using Greedy strategy:

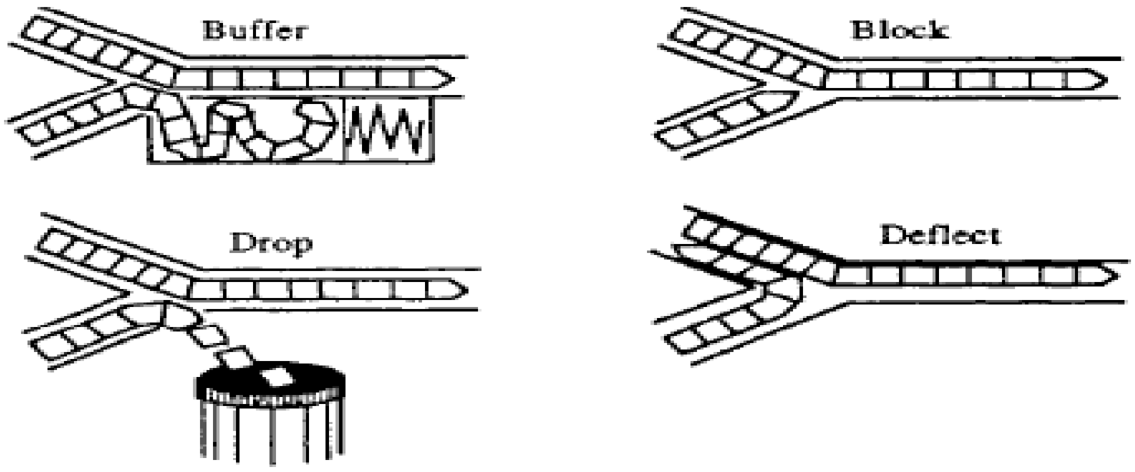


WORMHOLE ROUTING
Wormhole switching:Combines the advantages of circuit and packet switching.
Circuit switching:A circuit is established between source and destination before message is sent (as in old telephone networks)
Advantage: Fast transmission after the initial overhead
Packet switching:Packets are sent independently over possibly different paths
Advantage: Efficient use of channels due to sharing.
In wormhole routing, each packet is viewed as consisting of a sequence of flits (flow-control digits, typically 1–2 bytes). Flits, rather than complete packets, are forwarded between nodes, with all flits of a packet following its head flit like a worm.
At any given time, the flits of a packet occupy a set of nodes on the selected path from the source to the destination. However, links become available for use by other worms as soon as the tail of a worm has passed through. Therefore, links are used more efficiently compared with circuit switching.
The down side of not reserving the entire path before transmitting a message is that deadlocks may arise when multiple worms block each other in a circular fashion.
Any routing algorithm can be used to choose the path taken by the worm.
Routing algorithm must be simple to make the route selection quick.
Also deadlock and collision avoidance strategies need to be designed.

Wormhole switching:

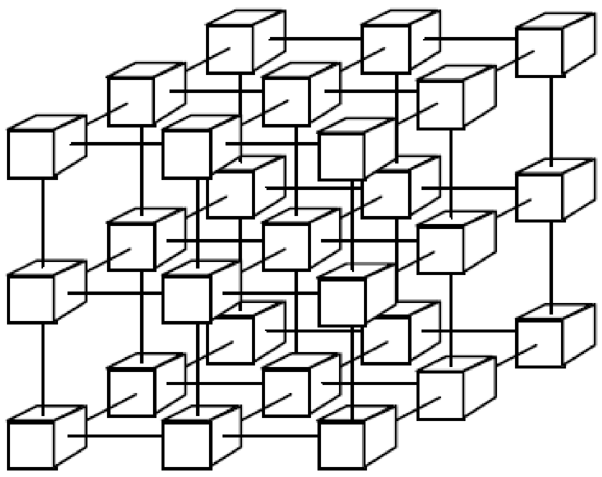


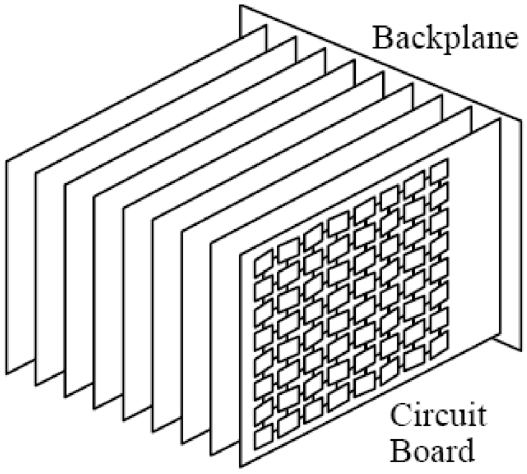
Collision Avoidance strategies in Wormhole routing:



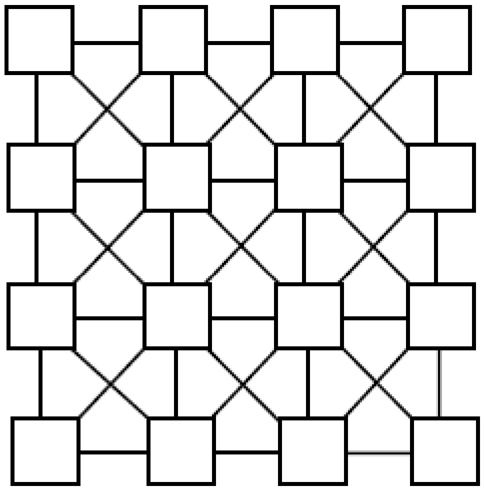
MESH RELATED ARCHITECTURES:

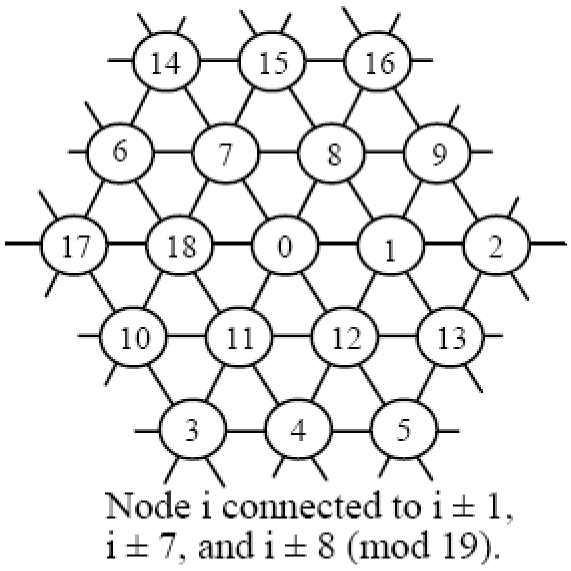
3D and 2.5D mesh:





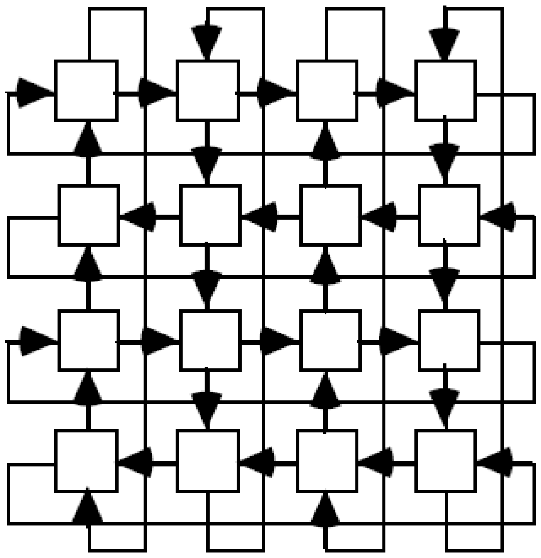
Eight neighbor and Hexagonal Mesh:





4 x 4 Manhattan-Street Network:

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

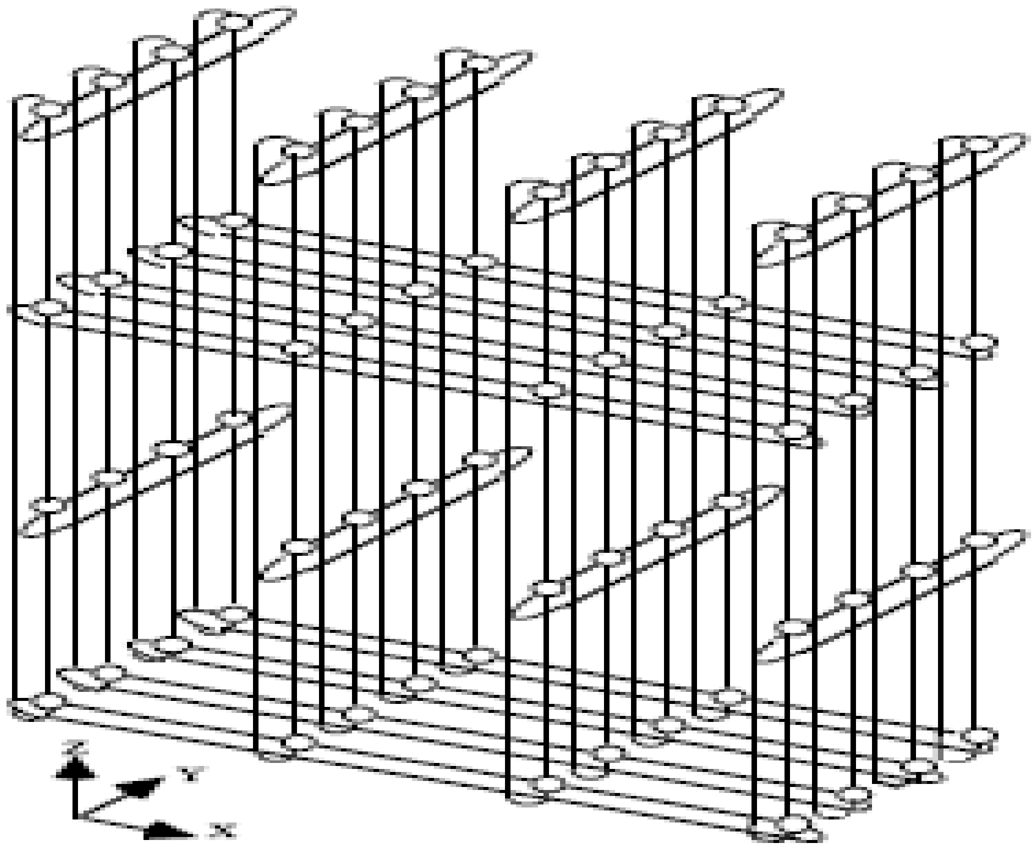


With even side lengths, the diameter does not change

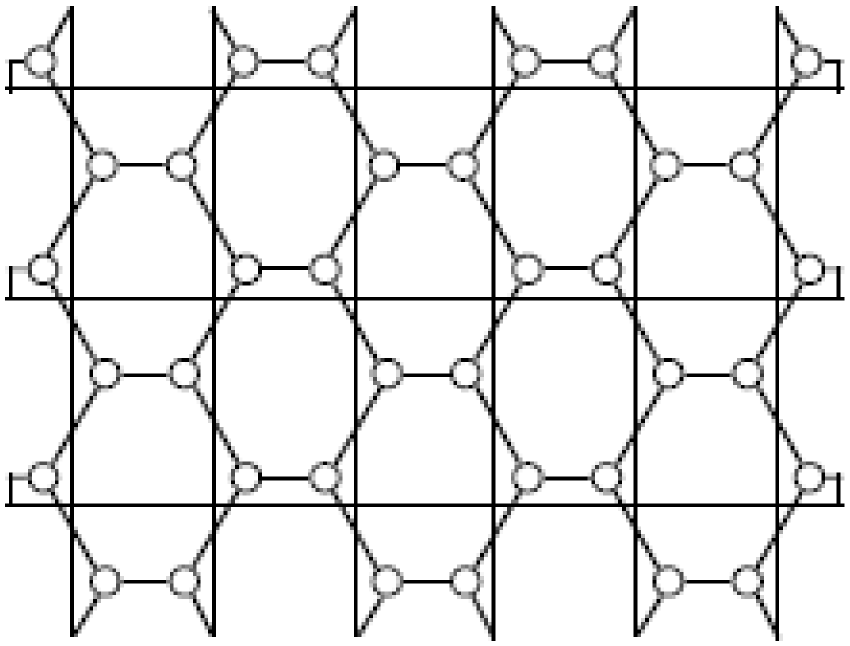
Some shortest paths become longer, however

Can be more cost-effective than 2D mesh

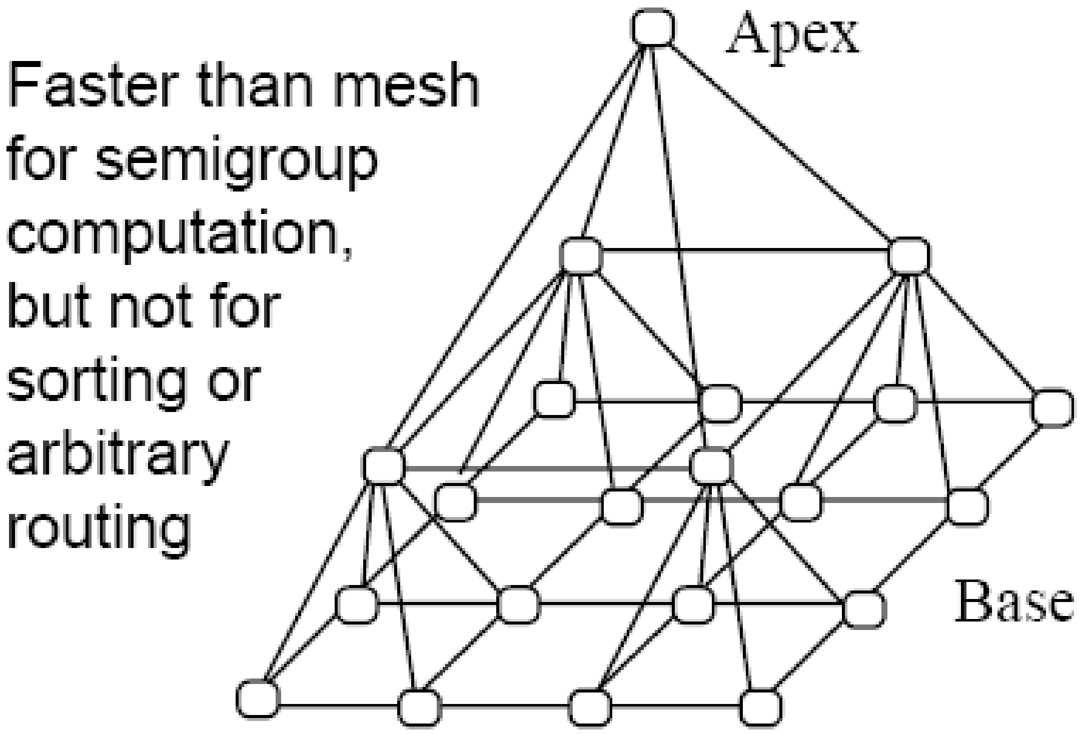
4 x 4 x 4 Mesh (Torus):



Honeycomb Mesh (Torus):

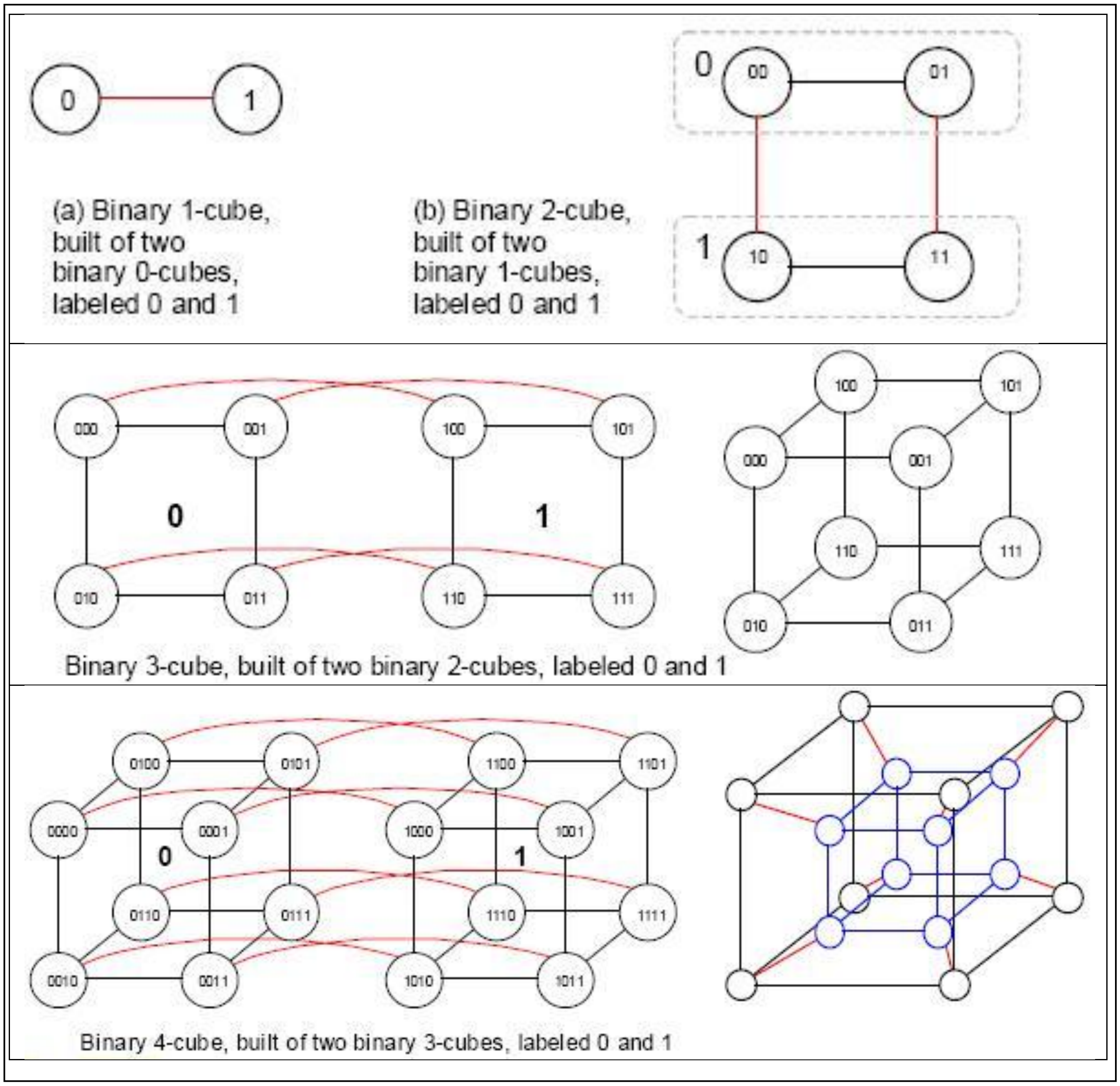


Pyramid with 3-levels and 4 x 4 base:



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

Hypercubes and Their Algorithms:
The logarithmic diameter, linear bisection, and highly symmetric recursive structure of the hypercube support a variety of elegant and efficient parallel algorithms. Hypercubes are also called binary q-cubes, or simply q-cubes, where q indicates the number of dimensions.



(a) Binary 1-cube, built of two binary 0-cubes, labeled 0 and 1

(b) Binary 2-cube, built of two binary 1-cubes, labeled 0 and 1

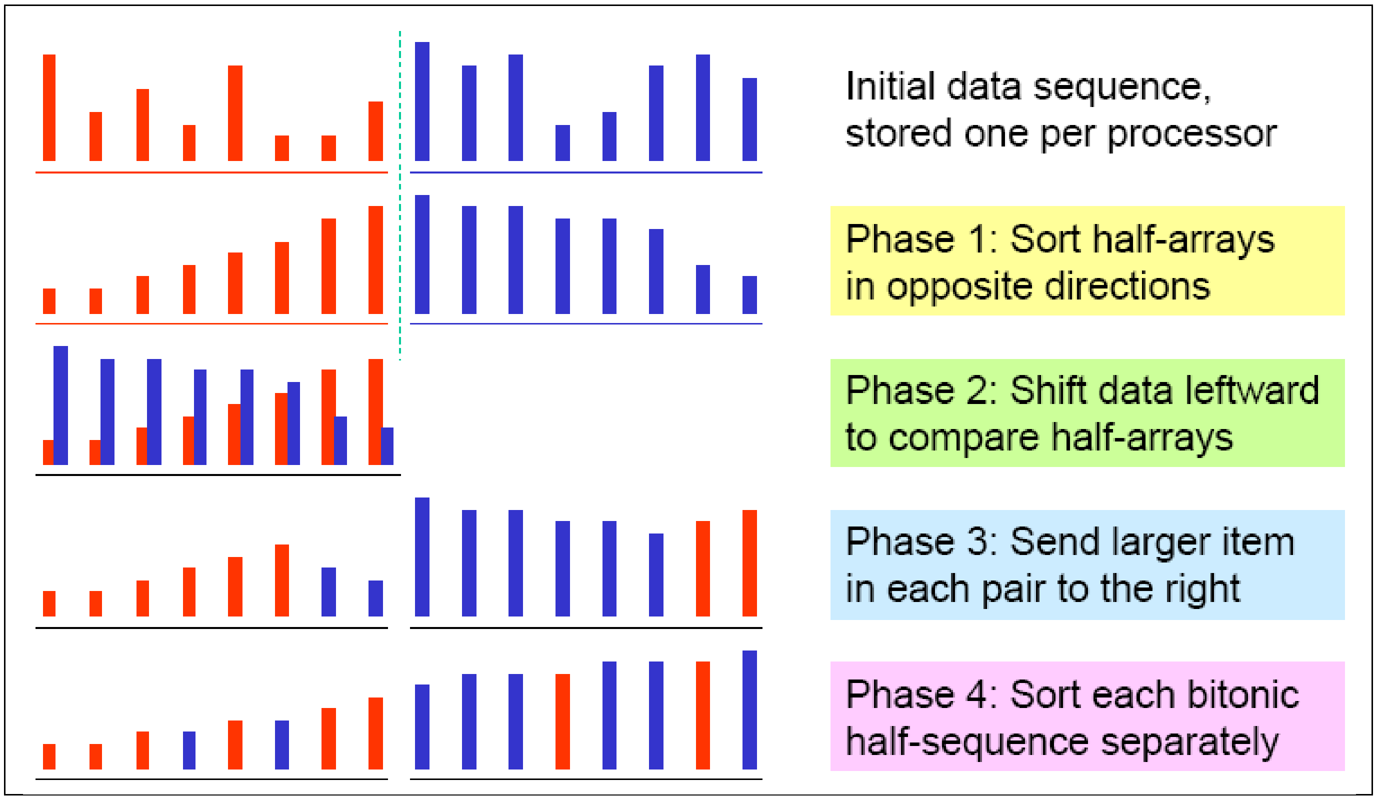
Binary 3-cube, built of two binary 2-cubes, labeled 0 and 1

Binary 4-cube, built of two binary 3-cubes, labeled 0 and 1

SORTING AND ROUTING ON HYPERCUBES:

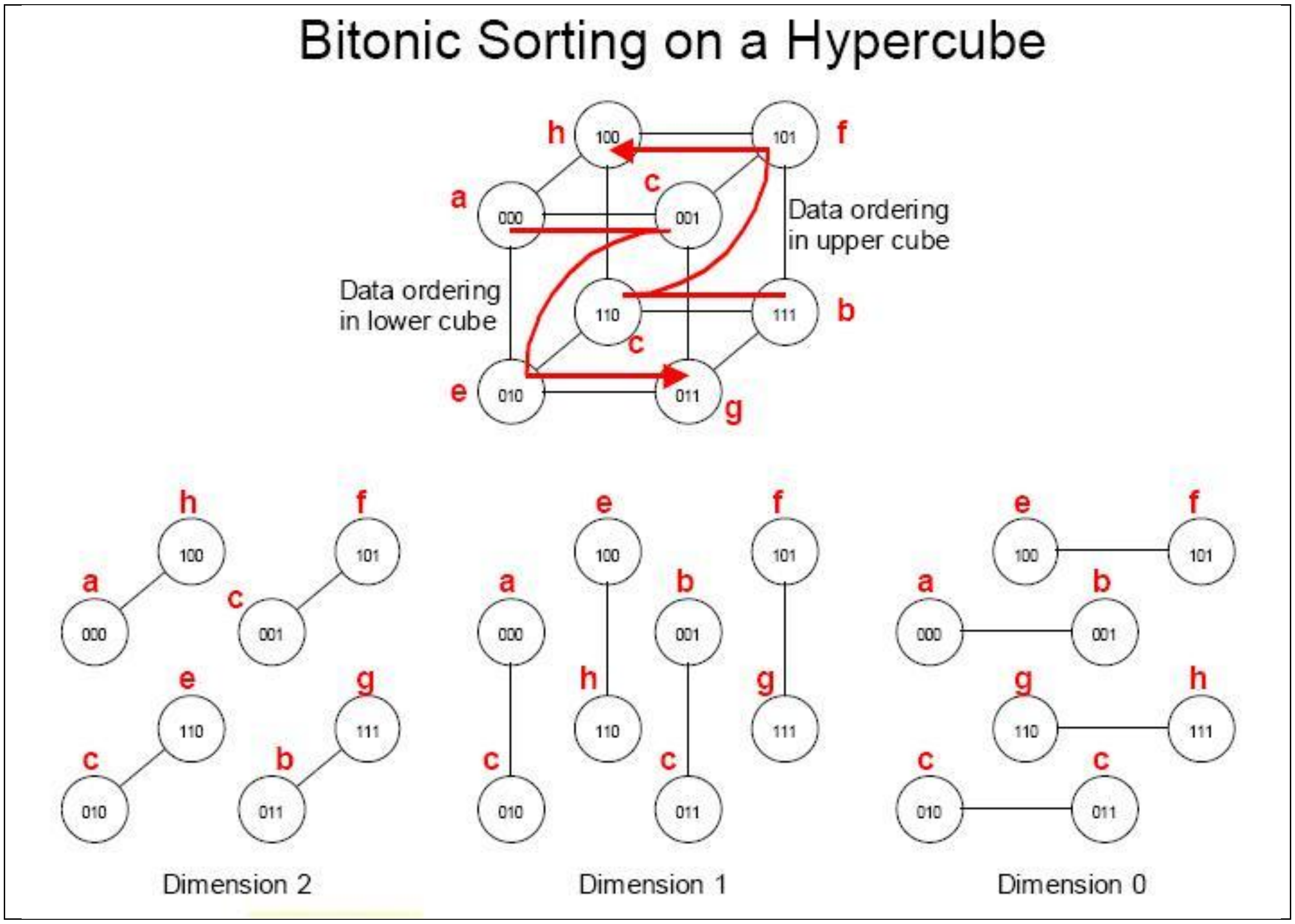
Sorting on Hypercubes:
The general problem of sorting on a hypercube is as follows: Given n records distributed evenly among the p = 2q processors of a q-cube (with each processor holding n/p records), rearrange the records so that the key values are in the same order as the processor node labels.
The most practical hypercube sorting algorithms are based on Batcher's odd–even merge or bitonic sort.

Bitonic Sequence:
A bitonic sequence is one that "rises then falls" "falls then rises" or is obtained from the above two types of sequences through cyclic shifts or rotations.
Such sequences "change direction" at most once; contrast this to monotonic sequences that do not change direction


Initial data sequence, stored one per processor
Phase 1: Sort half-arrays in opposite directions
Phase 2: Shift data leftward to compare half-arrays
Phase 3: Send larger item in each pair to the right
Phase 4: Sort each bitonic half-sequence separately

Batcher's bitonic sort on a hypercube:
Compare-exchange values in the upper subcube (nodes with $xq–1= 1$) with those in the lower subcube ($xq–1= 0$); sort the resulting bitonic half-sequences.

```
Sorting a bitonic sequence of size $n$ on $q$-cube, $q= \log_2 n$
for $l= q–1$ downto0 processor $x$, $0 \le x < p$, do
        if $x_l = 0$
        then
                get $y:= v[N_l(x)]$; keep $\min(v(x), y)$; send $\max(v(x), y)$ to $N_l(x)$
        endif
endfor
```


Bitonic Sorting on a Hypercube
Data ordering in upper cube
Data ordering in lower cube
Dimension 2
Dimension 1
Dimension 0

Routing on Hypercubes:
The routing on hypercubes is more exhaustive and must consider many dimensions because

- their larger node degree translates to the availability of more alternate (shortest) paths between processors,
- their logarithmic diameter leads to shorter routes, and
- their large bisection width allows many concurrent data transfers to take place between distant nodes.

Types of routing algorithms in bypercubes:

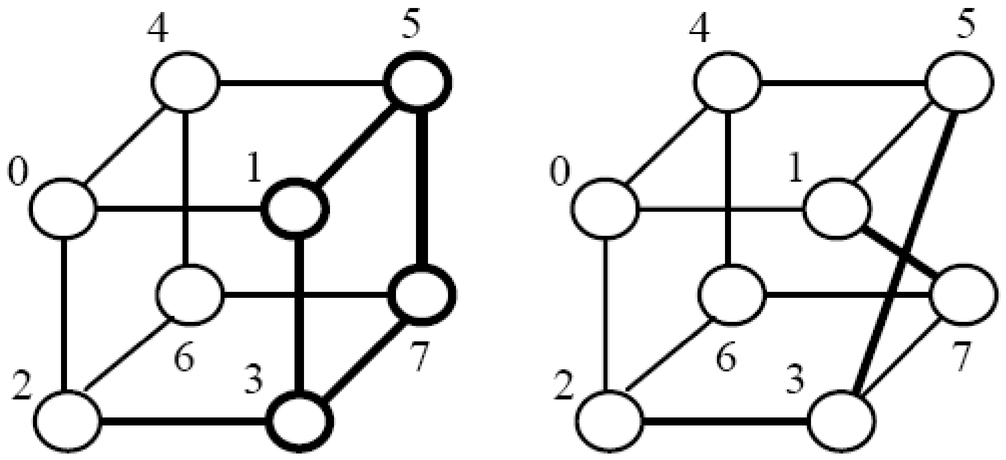
- **Oblivious:** path uniquely determined by node addresses. A routing algorithm is oblivious if the path selected for the message going from Processor i to Processor j is dependent only on i and j and is in no way affected by other messages that may exist in the system
- **Non-oblivious or adaptive**: the path taken by a message may also depend on other messages in the network.
- **On-line routing algorithms** make the routing decisions on the fly in the course of routing: Route selections are made by a parallel/distributed algorithm that runs on the same system for which routing is being performed. It allows the path to change as a result of link/processor congestion or failure
- **Off-line routing algorithms** are applied to routing problems that are known before routing is actually started: Route selections are precomputed for each problem of interest and stored, usually in the form of routing tables, within the nodes.

Broadcasting on a hypercube:
Two models can be used to broadcast in a hyoercube:

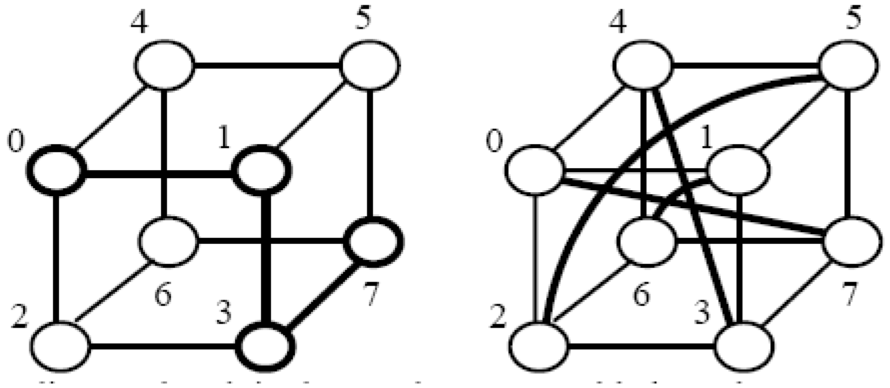
1. **All-port communication model**: A simple "flooding" scheme can be used for broadcasting a message from one node to all nodes in a q-cube in q steps, provided that each node can send a message simultaneously to all q neighbors (the all-port communication model). The source node sends the broadcast message to all of its neighbors. Each node, on receiving a broadcast message for the first time, relays it to all of its neighbors, except the one from which the message was received.
2. **Single-port communication model**: The single-port communication model is more reasonable and is the one usually implemented in practice. In this model, each processor (or actually the router associated with it) can send or receive only one message in each communication cycle. A simple recursive algorithm allows us to broadcast a message in the same q steps with this more restricted model.

Types of Modified Hypercubes:

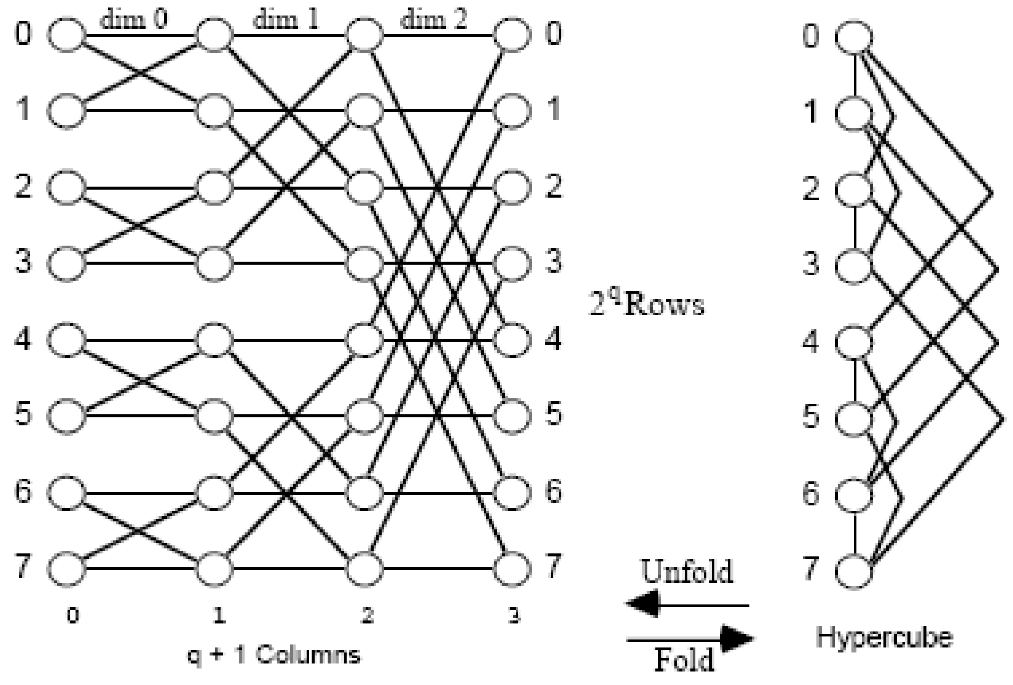
Twisted Hypercube:
In general, any 4-cycle 1,3,5,7 can be chosen in the q-cube, its 1,3 and 5,7 edges removed, and new edges 1,7, 3,5 inserted to obtain a twisted q-cube.



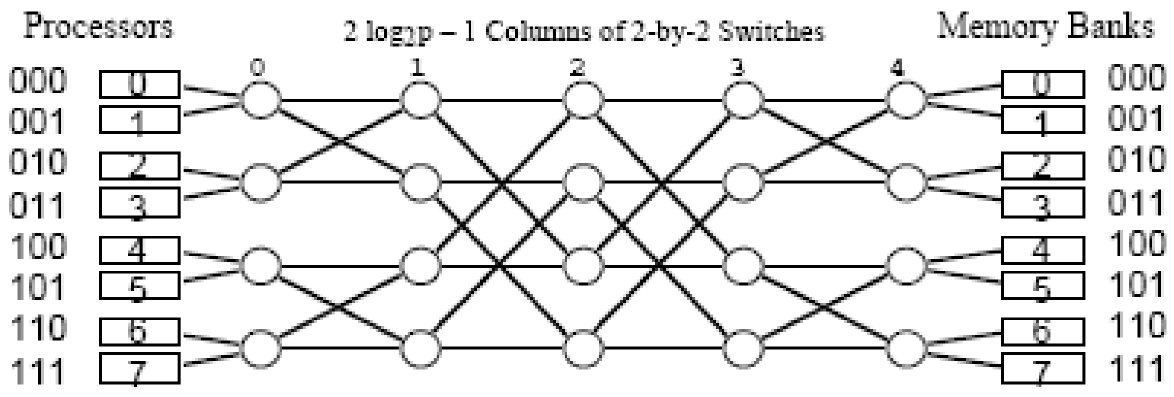
Folded Hypercube:
It is obtained from the hypercube by linking all pairs of diametrically (diagonally) opposite nodes.



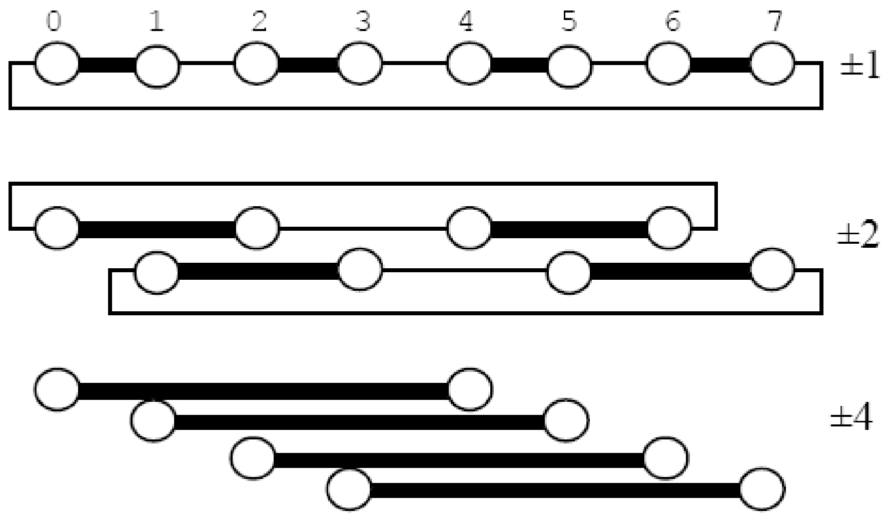
Butterfly Network:
A butterfly network is achieved by unfolding a hyoercube.

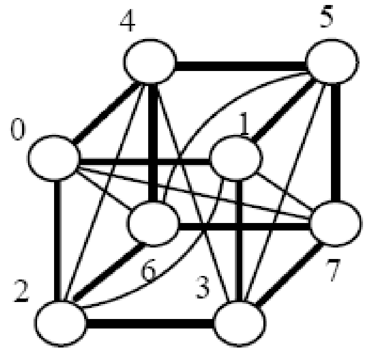


Butterfly network as multilevel interconnection network:
It can be viewed as a multilevel interconnection network connecting processors on one side and memory modules on the other side.
This arrangement is also known as Benes network.

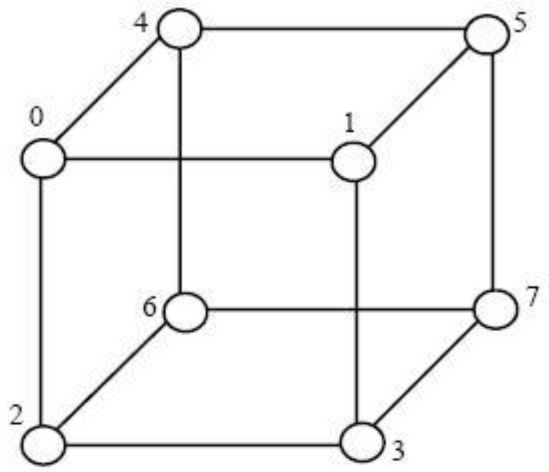


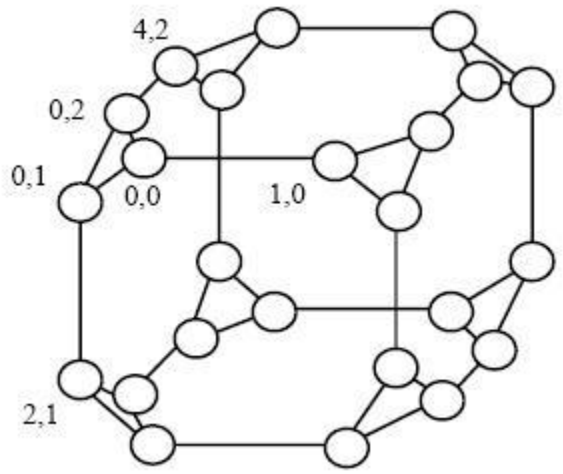
Plus-or-minus-2^i (PM2I) network:
A plus-or-minus-2^i network with eight nodes ($p = 2q$ nodes in general) is a network in which each Node x is connected to every node whose label is $x \pm 2^i$ mod p for some i..



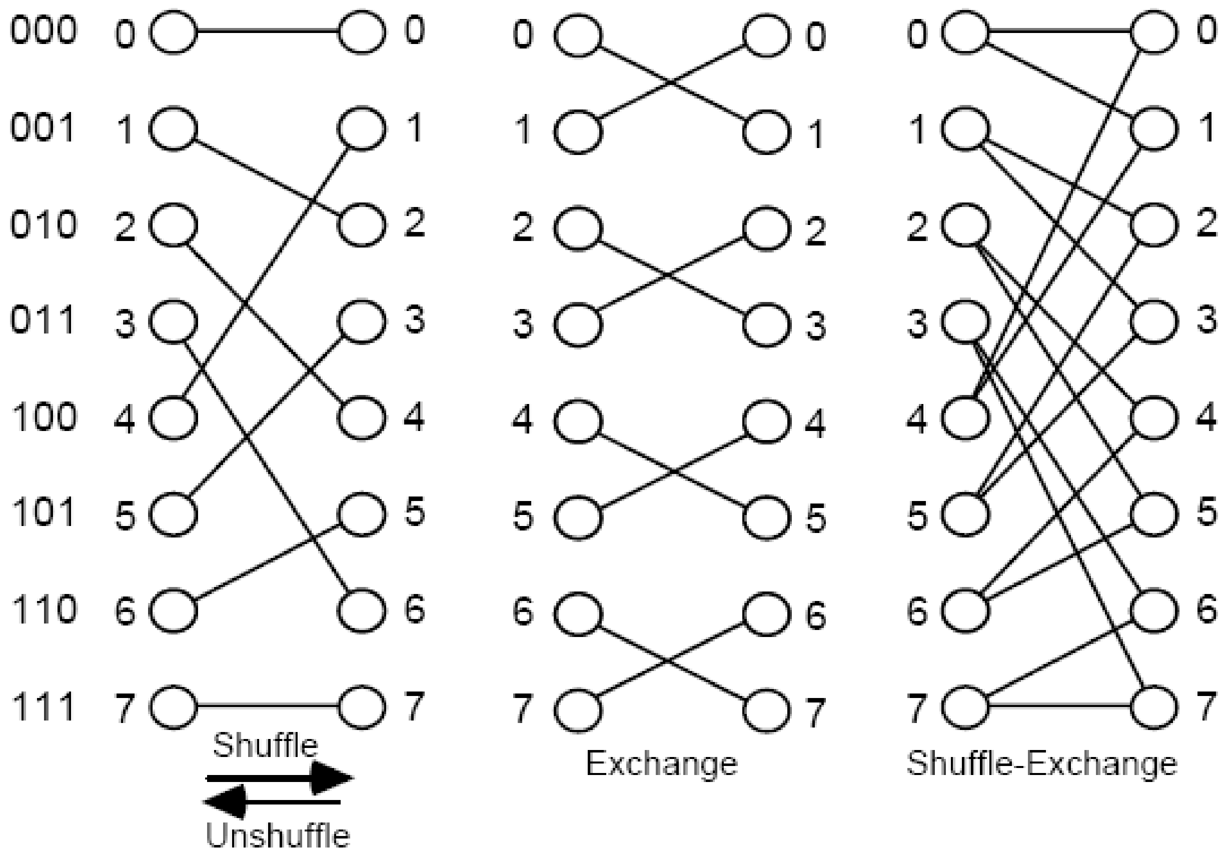


The cube-connected cycles (CCC) network:



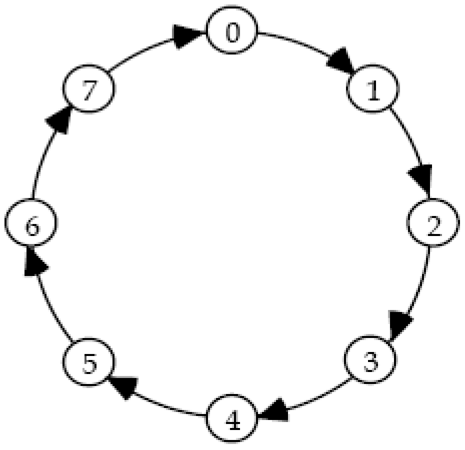


Shuffle and Shuffle-Exchange Network:
A perfect shuffle, or simply shuffle, connectivity is one that interlaces the nodes in a way that is similar to a perfect shuffle of a deck of cards.
The "exchange" connectivity, which links each even-numbered node with the next odd-numbered node.
Combining the shuffle and exchange connections, we get the connectivity of a shuffle–exchange network.

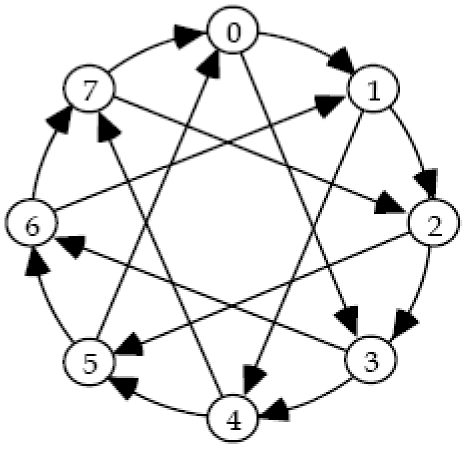


Ring-based Networks:

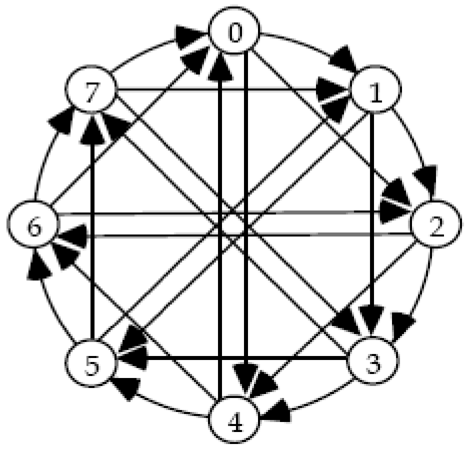
The ring interconnection scheme has proven quite effective in certain distributed and small-scale parallel architectures [Berm95] in view of its low node degree and simple routing algorithm.



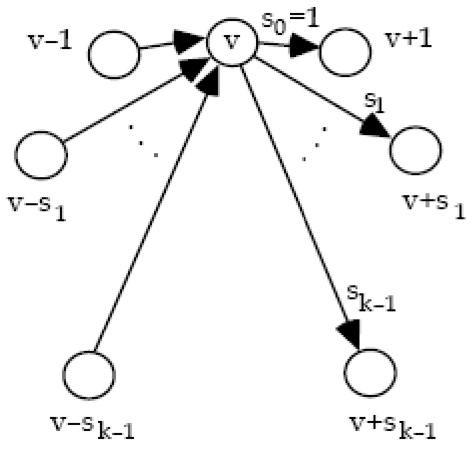
(a)



(b)



(c)



(d)