Evolution of parallel hardware

- I/O channels and DMA
- Instruction pipelining
- Pipelined functional units
- Vector processors (ILLIAV IV was built in 1974)
- Multiprocessors (cm* and c.mmp were built in the 70's)
- Massively parallel processors (Connection machine, T3E, Blue Gene, …)
- Symmetric Multiprocessors
- Cluster computing
- Multi-core processors
- Chip Multi-Processors

Two schools:

- Automatic detection of parallelism in serial programs and automatic distribution of data and computation.
- User specified parallelism (data distribution, computation distribution, or both).

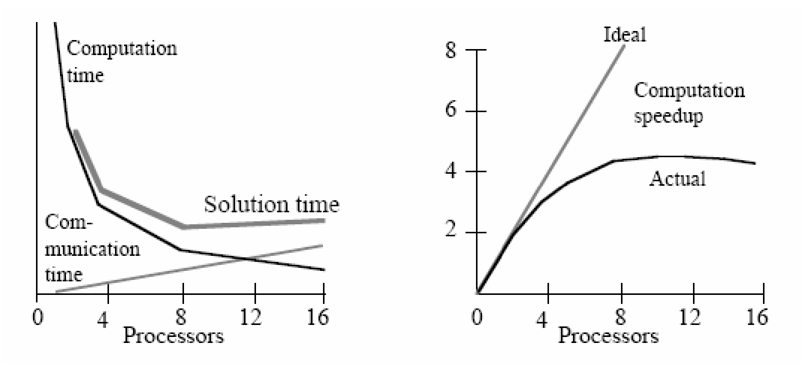
Problems:

- It is hard to think "parallel." (is it ???)
- The dusty-deck problem (software inertia) and the absence of good tools for parallelizing serial programs.
- The I/O bottleneck.
- Shrinking government funding with lack of commercial success (changing trend??).
- Communication and synchronization overhead

• Execution time:

1) Communication time

2) Computation time.



Tradeoff between computation and communication

Flynn's hardware taxonomy:

Looks at instructions and data parallelism. Best known of many proposals.

$\boxed{\begin{matrix}\mathbf{S}\\ \mathbf{M}\end{matrix}}$ **I** $\boxed{\begin{matrix}\mathbf{S}\\ \mathbf{M}\end{matrix}}$ **D**

• S for single	• I for instruction
• M for multiple	• D for data.

• An SISD machine is a serial computer.

• An SIMD machine is a vector machine or a lockstep machine with multiple units and one instruction stream.

• An MIMD machine is composed of different units, each having its own instruction stream.

• An MISD machine – need to be innovative to define it.

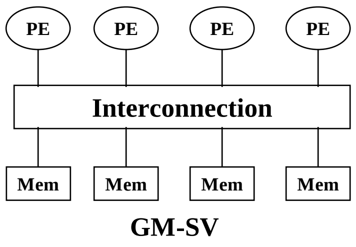
Taxonomy of MIMD machines.

- **According to physical memory layout:**
  GM = global memory , DM = distributed memory.
- **According to memory directly addressable by processors:**
  SV = shared variables -- a single shared address space,
  MP = message passing – each processor has its own space.

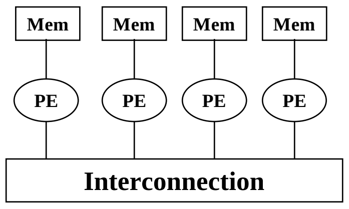
Note: Shared address space machines may be
UMA = uniform memory access, or NUMA = non-UMA.

DMSV = Memory physically distributed but logically shared.
GMSV = physically and logically shared memory – usually called symmetric multiprocessors (SMP) – usually use common bus.
DMMP = each processor has access to its local memory – data is shared through sending and receiving messages.

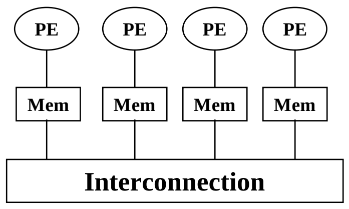
5



GM-SV



DM-MP



DM-SV

Which one is NUMA and which is UMA?

6

Speedup and efficiency.

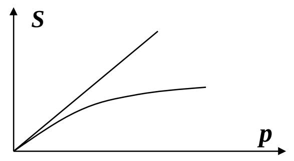
- For a given problem *A*, of size *n*, let $t_p(n)$ be the execution time on *p* processors, and $t_1(n)$ be the execution time (of the best algorithm for *A*) on one processor. Then,

  Speedup $S_p(n) = t_1(n) / t_p(n)$

  Efficiency $E_p(n) = S_p(n) / p$

  Speedup is between 0 and *p*, and efficiency is between 0 and 1.
- Linear Speedup means that *S* is linear with *p* (perfectly scalable machine)
- If speedup is independent of *n*, then the algorithm is said to be perfectly scalable.
- Minsky's conjecture:

  Speedup is logarithmic in *p*



Amdahl's law.

Let *f* be the fraction of a program that has to be performed serially, then, using *p* processors, the maximum possible speedup is:

$$S < \frac{1}{f + (1 - f)/p}$$

Hence, even with unlimited number of processors, the speedup cannot be larger than 1 / *f*.

- Algorithms for the same problem may have different values of *f*.
- The above formula ignores the effect of *n* on *f* (serial portion of code may be fixed, independent of the size of the problem).
- Ignores the effect of memory:

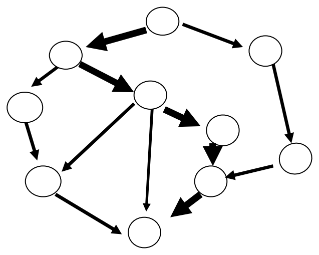
  Negative effect $\longrightarrow$ conflict.

  Positive effect $\longrightarrow$ more memory, cache and registers.
- Ignores the effect of communication.

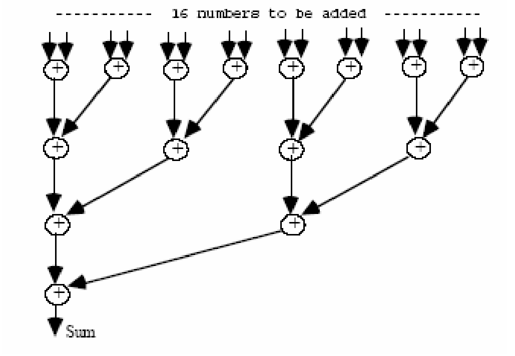
Critical path in task graphs:

1) In applications with dependent operations, speedup depends on the longest path in the dependency graph.

2) May have node labels (computation time) and link labels (communication time).

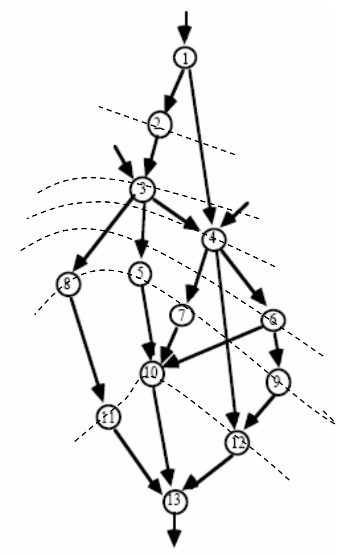


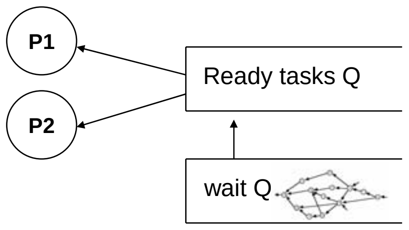
Example:



9

Scheduling task graphs to processors
Example: List scheduling

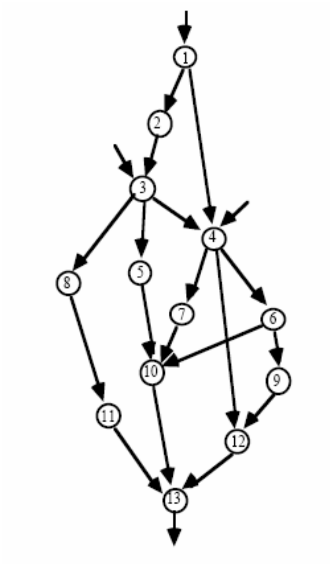


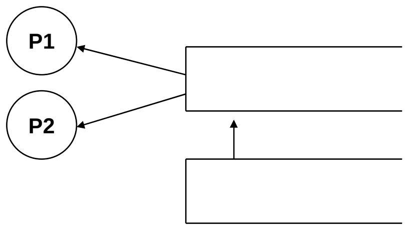


- Move a task to the ready Q when all predecessors finish execution
- Keep the ready Q ordered by some priority:
  - Depth of the task
  - Number of descendents
- The depth of a task is the length of the longest path from the task to the last task in the graph.

10

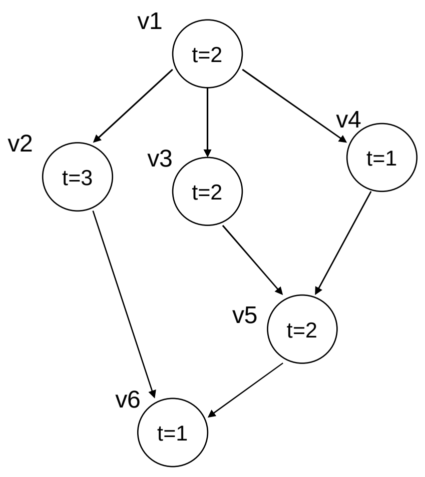
List scheduling





time	1	2	3	4	5	6	7	8	9
proc 1	1	2	3	4	6	8	10	12	13
proc 2	▨	▨	▨	5	7	9	11	▨	▨

Nodes with un-equal execution times



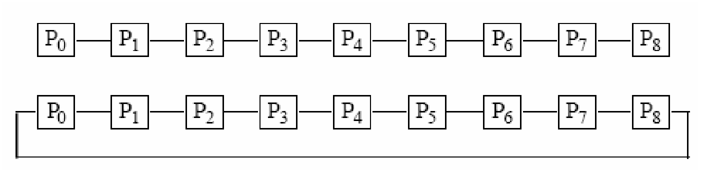
time	1	2	3	4	5	6	7	8
Proc 1	v1	v1	v3	v3	v4	v5	v5	v6
Proc 2	▨	▨	v2	v2	v2	▨	▨	▨

List scheduling

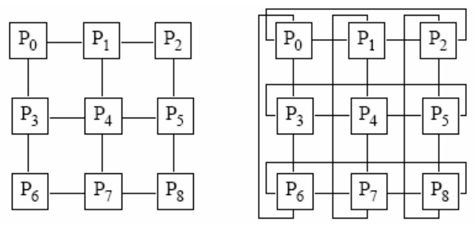
time	1	2	3	4	5	6	7	8
Proc 1	v1	v1	v3	v3	v5	v5	v6	▨
Proc 2	▨	▨	v4	v2	v2	v2	▨	▨

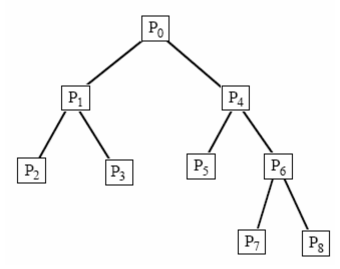
optimal scheduling

Some simple architectures



Linear arrays and rings



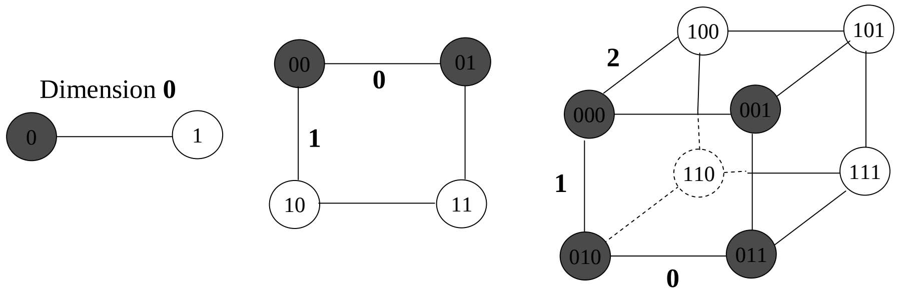


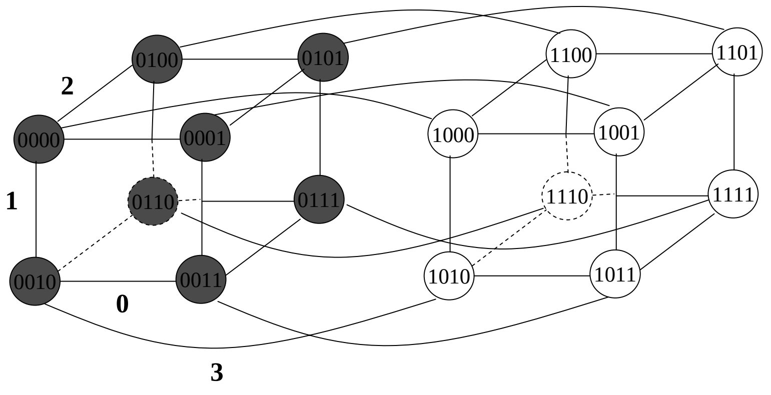
Meshes and torii

Tree architectures

Hypercube interconnections

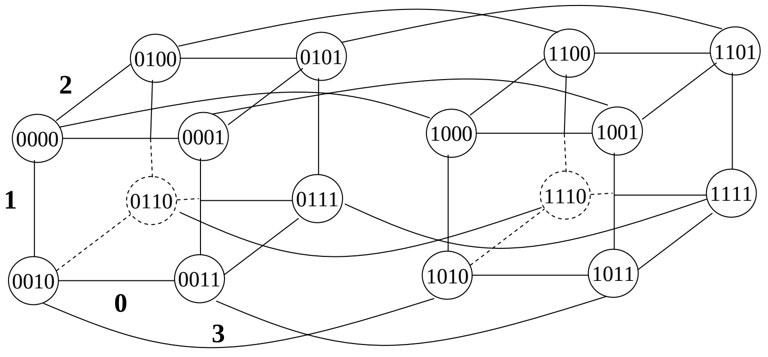
- An interconnection with low diameter and large bisection bandwidth.
- A *q*-dimensional hypercube is built from two *(q-1)*-dimensional hypercubes.





For a *q* dimension hypercube, calculate

- The number of nodes and the number of edges
- The node degree
- The diameter
- The bisection bandwidth



- Each node in a *q*-dimension hypercube has a *q*-bits identifier $x_{q-1}, \ldots, x_1, x_0$
- Identifiers of nodes across a dimension *j* differ in the *j*th bit (have a unit **Hamming distance**)
- Nodes connected by a link are called neighbors

Network parameters

Number of nodes (*p*)
Number of links
Interconnection graph

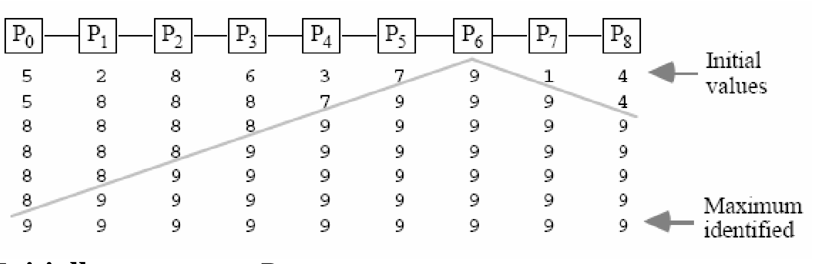
Network characteristics

Node degree = number of neighbors
Diameter = the maximum distance between any two nodes
Bisection width (BW) = the minimum number of links that partition the network into two, almost equal, halves.

	linear	**ring**	**2D mesh**	**2D torus**	**Binary tree**
degree					
diameter					
BW					

Examples of parallel algorithms

Finding the maximum on a linear array



Initially: processor, P_i , stores x_i

Each P_i , $i=0,\ldots,p-1$, executes

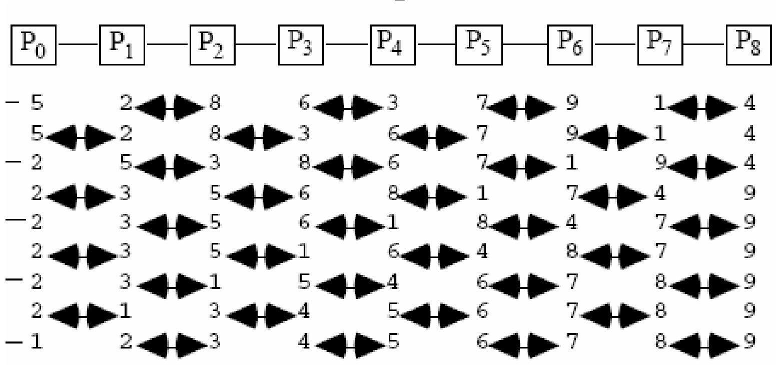
```
    For $k=1$ to $k=p-1$
        $x_i = \max\{x_{i-1}, x_i, x_{i+1}\}$
```

Result: P_i stores $\max\{x_0, \ldots, x_{p-1}\}$

Speedup??

What if the array has 900 elements equally distributed in the nodes?

Odd-even transposition sort



Initially: processor, P_i , stores x_i

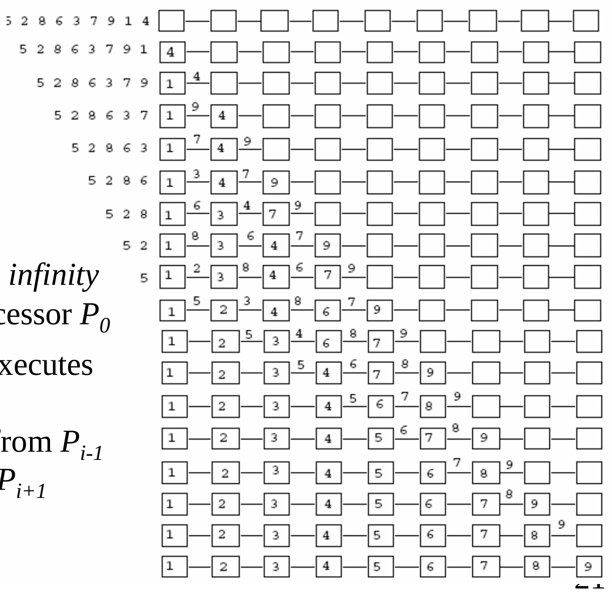
Each P_i , $i=0,\ldots,p-1$, executes

```
  For $k=1$ to $k=p$
      if ($i+k$ is odd) and ($i > 0$), then $x_i = \max\{x_i, x_{i-1}\}$
      else if ($i+k$ is even) and ($i < p-1$) then $x_i = \min\{x_i, x_{i+1}\}$
```

Result: $x_0 < \ldots < x_{p-1}$

Message passing??
Correct values?

Sorting an input stream of values



Initially: P_i stores x_i = *infinity*

Input: *p* values to processor P_0

```
Each $P_i$ , $i=0,\dots,p-1$ , executes
  Repeat ($p-i$) times
    receive a value, $y$, from $P_{i-1}$
    send max{ $x_i$ ,$y$} to $P_{i+1}$
    $x_i$ = min{ $x_i$ , $y$}
```

Result: $x_0 < \dots < x_{p-1}$

Sorting on tree-connected processors

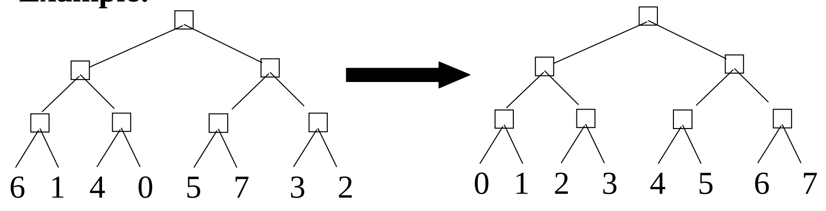
Initially: each leaf node stores a value

Upward phase: propagates values upwards so that they reach the root in sorted order

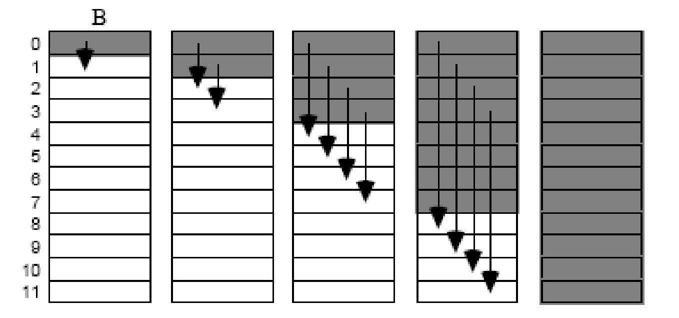
Downward phase: send the values back to the leaves

Result: the values at the leaves are sorted

Example:



Data broadcasting on *p* processors (recursive doubling)

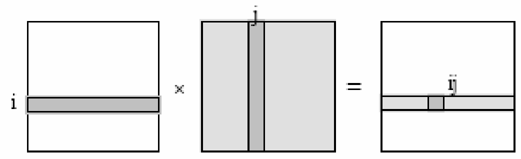


```
Each processor, $j = 0, …, p-1$ executes
For $k = 0, … , ceiling(\log p) – 1$
   if ( $j < 2^K$ ) and ( $j+2^k < p$ ) copy $B[ j ]$ into $B[ j+2^k ]$.
```

Matrix multiplication

On a single processor

```
for $i = 0, … , m-1$
   for $j = 0, … , m-1$
      $c[ i,j ] = 0$
      for $k = 0, … , m-1$
         $c[ i,j ] =+ a[ i,k ] * b[ k,j ]$
```

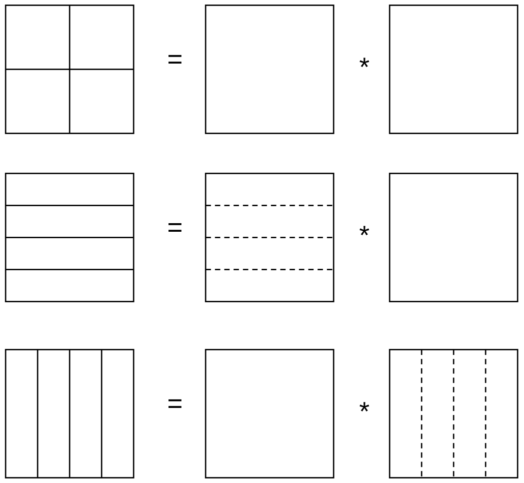


On m^2 processors

```
Each processor $(i,j)$,  $0 <= i,j <= m-1$, executes
  $c[ i,j ] = 0$
   for $k = 0, … , m-1$
      $c[ i,j ] =+ a[ i,k ] * b[ k,j ]$
```

- **What if we do not have a shared memory??**
- **What if the number of processors, *p = m* and not *m*^{2} ??**
- **What if *p = m/q* ??**

EXAMPLE: 32x32 matrices using *4* processors

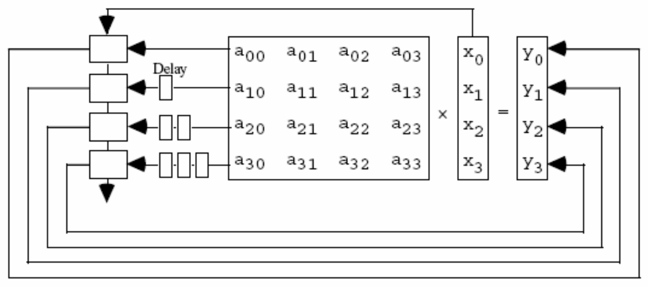


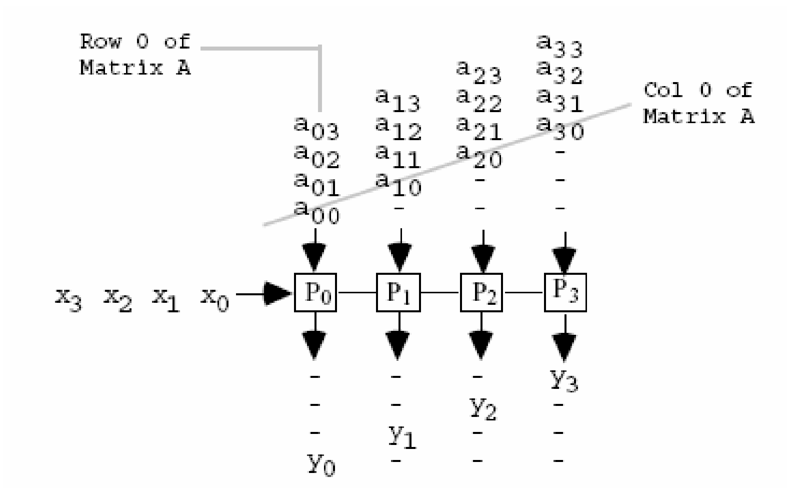
```
for $i$ = $0, … , 31$
  for $j$ = $0, … , 31$
    $c[\,i,j\,] = 0$
    for $k$ = $0, … , 31$
        $c[\,i,j\,]$ =+ $a[\,i,k\,] * b[\,k,j\,]$
```

Numerical 2D mesh algorithms

Matrix Vector multiplication

```
for $i$ = $0, … , m-1$
    $y\,[\,i,j\,] = 0$ ;
    for $j$ = $0, … , m-1$
        $y\,[\,i,j\,]$ =+ $a\,[\,i,j\,] * x\,[\,j\,]$ ;
```


Delay
a_{00} a_{01} a_{02} a_{03}
a_{10} a_{11} a_{12} a_{13}
a_{20} a_{21} a_{22} a_{23}
a_{30} a_{31} a_{32} a_{33}
×
x_0 x_1 x_2 x_3
=
y_0 y_1 y_2 y_3

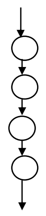


Row 0 of Matrix A
Col 0 of Matrix A
a_{03} a_{02} a_{01} a_{00}
a_{13} a_{12} a_{11} a_{10}
a_{23} a_{22} a_{21} a_{20}
a_{33} a_{32} a_{31} a_{30}
x_3 x_2 x_1 x_0
P_0 P_1 P_2 P_3
y_0 y_1 y_2 y_3

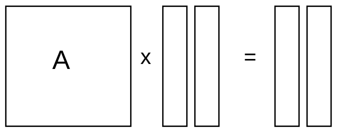
Time to completion for an *m* x *m* problem on *p* = *m* processors is *2m-1*

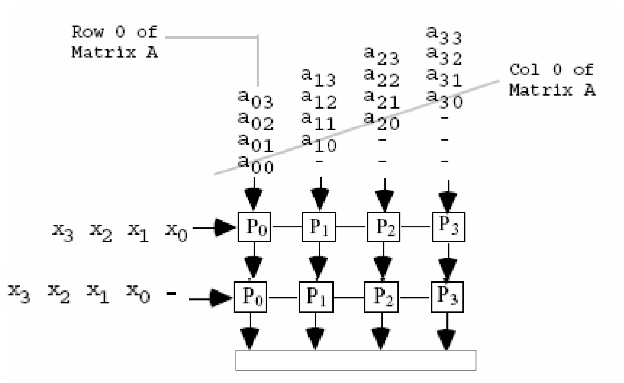
$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

$$\begin{array}{cccc|c} x_0 & x_1 & x_2 & x_3 & \\ \hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & y_0 \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & y_1 \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & y_2 \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & y_3 \end{array}$$

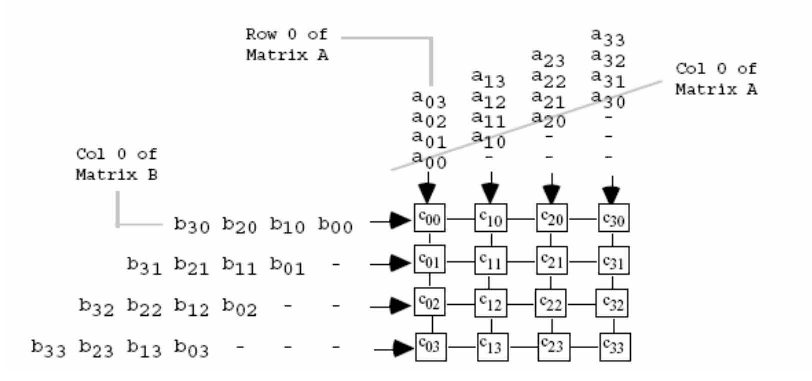


May repeat for multiple vectors



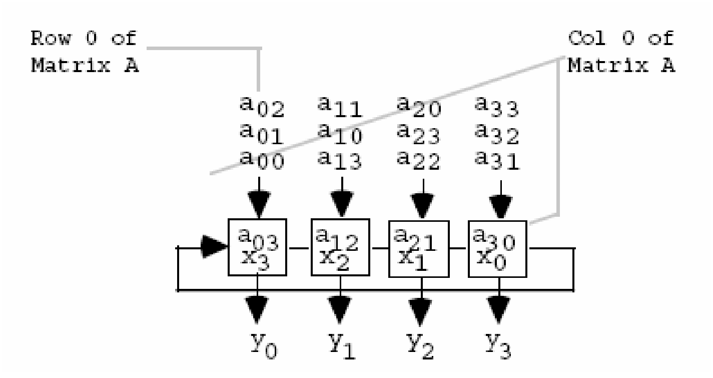


Matrix-Matrix multiplication



Time to completion for *m* x *m* matrices on $p = m^2$ processors is *3m-2*

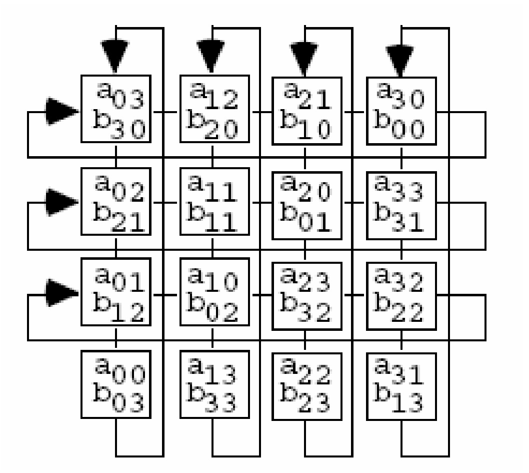
On a ring with $p = m$ processors, we can finish matrix-vector multiplication in m steps (what is the speedup?).



Each element $a_{i,j}$ can be a k x k sub-matrix and each x_i and y_i can be a k-dimensional sub-vector (here $m = k\ p$. What is the speedup?)

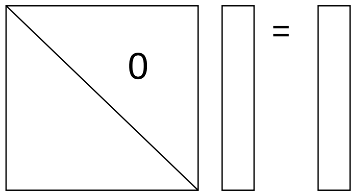
31

On a torus with $p = m^2$ processors, we can finish matrix-matrix multiplication in m steps (what is the speedup?).



32

Solution of diagonal systems



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

Example

$a_{0,0}\ x_0 = b_0$

$a_{1,0}\ x_0 + a_{1,1}\ x_1 = b_1$

$a_{2,0}\ x_0 + a_{2,1}\ x_1 + a_{2,2}\ x_2 = b_2$

$a_{3,0}\ x_0 + a_{3,1}\ x_1 + a_{3,2}\ x_2 + a_{3,3}\ x_3 = b_3$

Solution

$x_0 = b_0 / a_{0,0}$

$x_1 = (b_1 - a_{1,0}\ x_0) / a_{1,1}$

$x_2 = (b_2 - a_{2,0}\ x_0 - a_{2,1}\ x_1) / a_{2,2}$

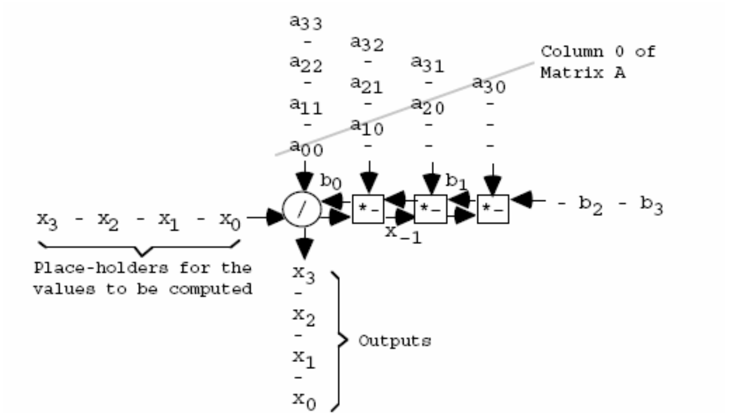
$x_3 = (b_3 - a_{3,0}\ x_0 - a_{3,1}\ x_1 - a_{3,2}\ x_2) / a_{3,3}$

$x_0 = b_0 / a_{0,0}$

$x_1 = (b_1 - a_{1,0}\ x_0) / a_{1,1}$

$x_2 = (b_2 - a_{2,0}\ x_0 - a_{2,1}\ x_1) / a_{2,2}$

$x_3 = (b_3 - a_{3,0}\ x_0 - a_{3,1}\ x_1 - a_{3,2}\ x_2) / a_{3,3}$



Routing problems

- Routing a single message
- Permutation routing
- Multicasting and broadcasting (one to many)
- Reduction or combine operations (many to one)
- All-to-all broadcasting (many to many)
- Personalized all-to-all (scatter-gather or gossiping)
- Data packing and compaction

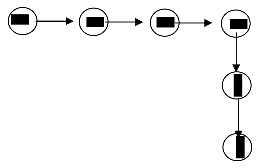
Terminology for routing problems

- Static: packets to be routed are all known before routing starts
- Dynamic: packets are created dynamically
- Off-line: routing decisions are pre-computed and stored in routing tables
- On-line: routing decisions are computed on-line
- Oblivious: routes depends only on source and destination
- Adaptive: routes are determined based on condition of the environment (link or node congestion, faults, delay, ….)
- Deflection routing: if shortest path is congested, use some detour (hot potato routing).

37

Switching schemes

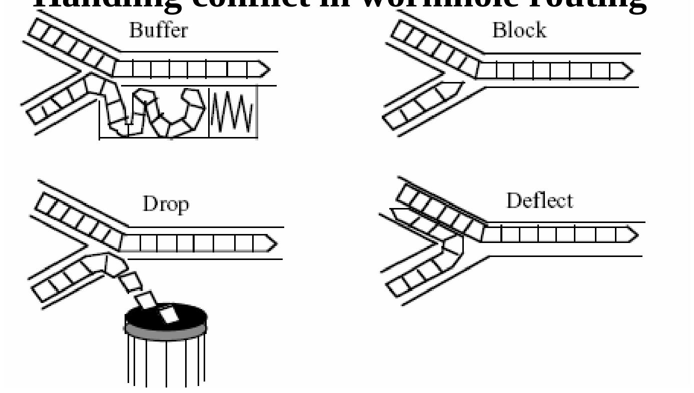
- Packet switching (store and forward)
- Circuit switching
- Wormhole switching (routing)
  - Packet broken into FLITs
  - Buffering FLITs
  - Header FLIT sets a “virtual circuit”
  - Tail FLIT destroy the virtual circuit



- Latency for a message depends on the time to transfer (and buffer) a flit (*f*), the number of flits (*m*), as well as the number of hops (*h*).
- Latency is much lower than that for packet switching (depends on the time to transfer and buffer a packet (*mf*) and the number of hops (*h*).

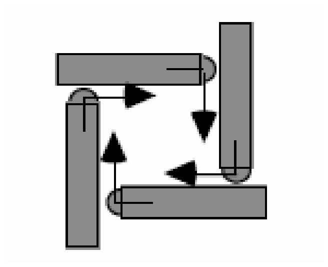
38

Handling conflict in wormhole routing



- Virtual cut-through = wormhole routing with buffering of blocked Flits.
- Dropped messages are dealt with by higher layers protocols
- Should prevent "*LIVE-LOCK*" and "*DEAD-LOCK*" when using deflection routing (or any other adaptive routing).

Handling deadlock in wormhole routing



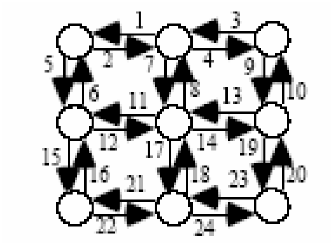
- Deadlock occurs when there is a circular waiting on some resources (in this case, buffer space)

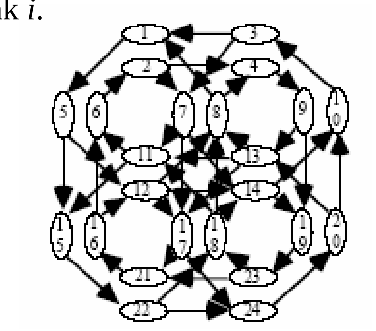
- Deadlock occurs when there is a circular waiting on some resources (in this case, buffer space)

- Deadlock detection (how and what to do when detected?)
- Deadlock avoidance (the resource dependence graph should be cycle-free).

Resource dependence graph in wormhole routing

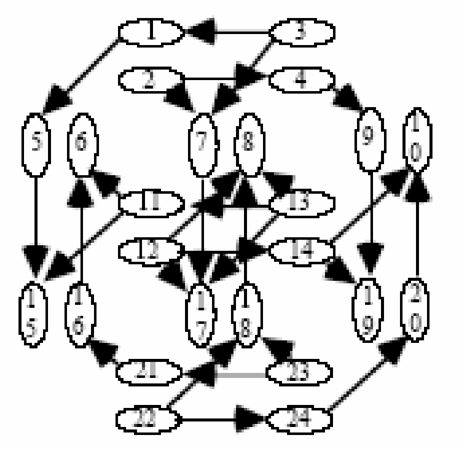
- Buffers (at either end of a communication link) is the resource
- The resource graph:
  - a node for each link in the network
  - an edge from node *i* to node *j* if the routing allows a message to cross link *j* after link *i*.





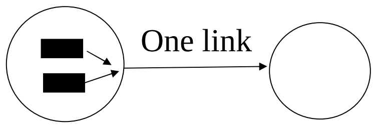
Routing on 2D meshes, in general, is not deadlock free.

Row-first routing is deadlock free



Deadlock free routing in meshes

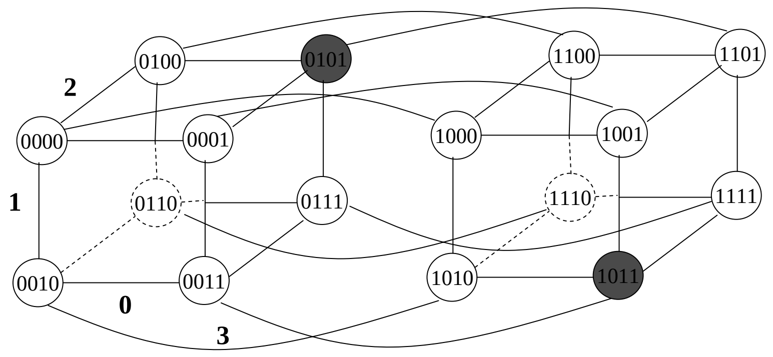
If you do not want to restrict routing to "row first", then you need to use two virtual channels for each communication links.



Two buffers
(one for each channel)

- A message starts on Channel 1, and moves to channel 2 when it makes a turn – If no more than one turn, then no deadlock can occur.

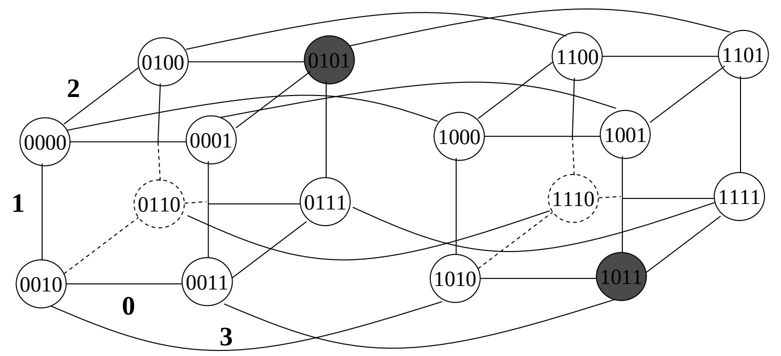
Routing on a hypercube



A message from a source, $s_{q-1}, \ldots, s_0$, to a destination $x_{q-1}, \ldots, x_0$ has to cross any dimension, b, for which $x_b \neq s_b$

How many distinct routes there are between any source and destination?

Dimension-order routing



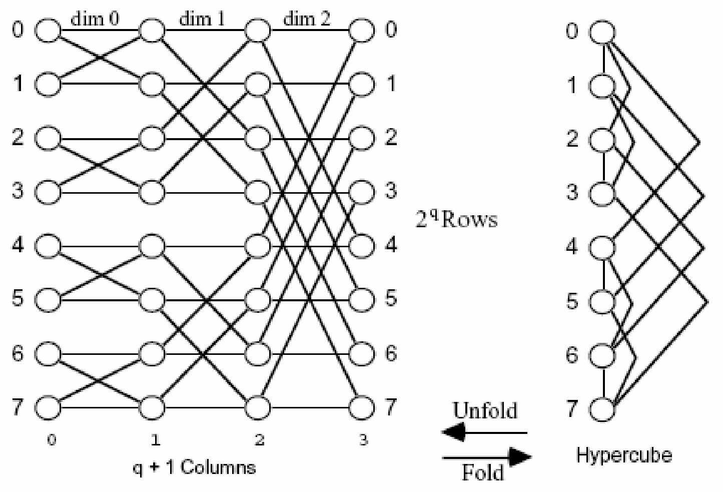
When a node, $n_{q-1}, \ldots, n_0$, receives a message for destination node $x_{q-1}, \ldots, x_0$, it executes the following

- If $x_k = n_k$ for $k = 0, \ldots, q-1$, , keep the message
- Else { Find the largest k such that $x_k \neq n_k$;

  Send the message to the neighbor across dimension k }

45

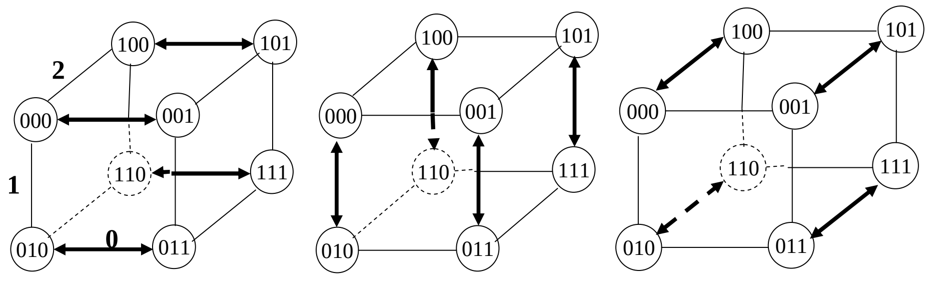
Visualizing the route by unfolding the hypercube



The unfolded hypercube forms the so called “Butterfly network”

46

Semigroup operation on a hypercube (EX: global sum)



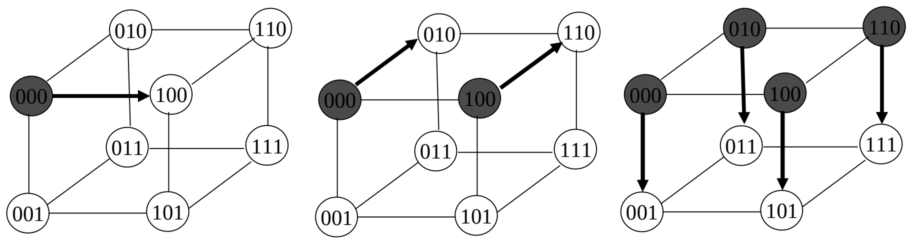
Initially: each node has a value *x*

Each node executes the following:

```
For $j = 0 , \dots , q-1$ do
    send $x$ to neighbor across dimension $j$
    receive the value sent from the neighbor across dimension $j$
    $x = x +$ the received value
```

Result: *x* in each node contains the global sum

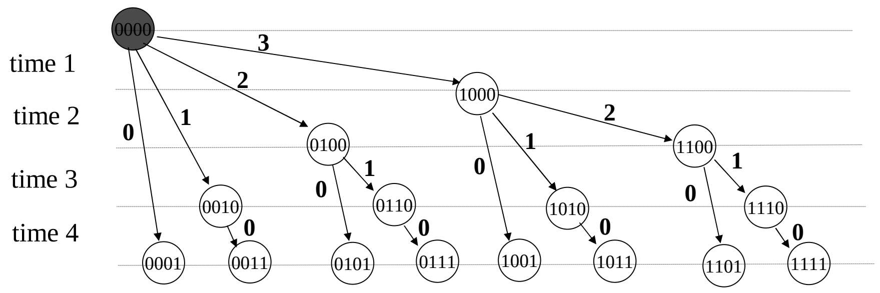
Broadcasting in hypercubes

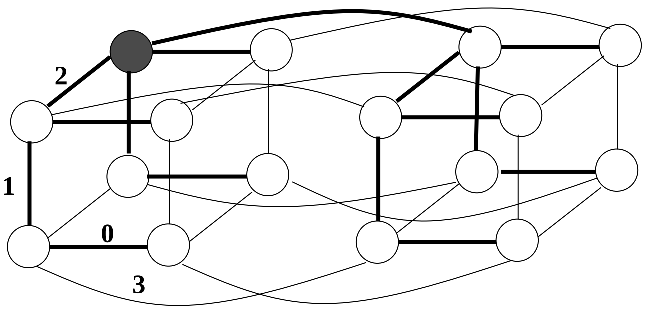


Each node executes the following:

```
Let $Ng(i)$ be the neighboring node across dimension $i$.
If root, set $K = q-1$ ;
    else if received a message from $Ng(i)$, set $K = i-1$ ;
For $j = K , \dots , 0$ do
    send the message to $Ng(j)$
```

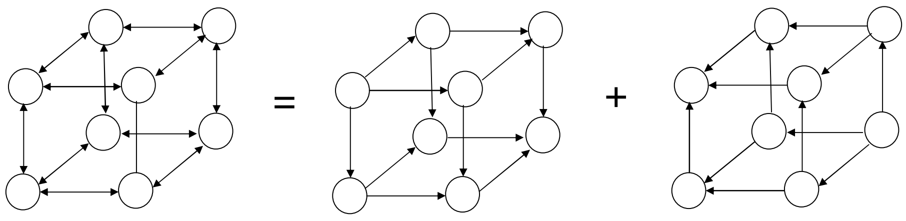
Broadcast on a binomial broadcast tree





Adaptive routing in hypercubes

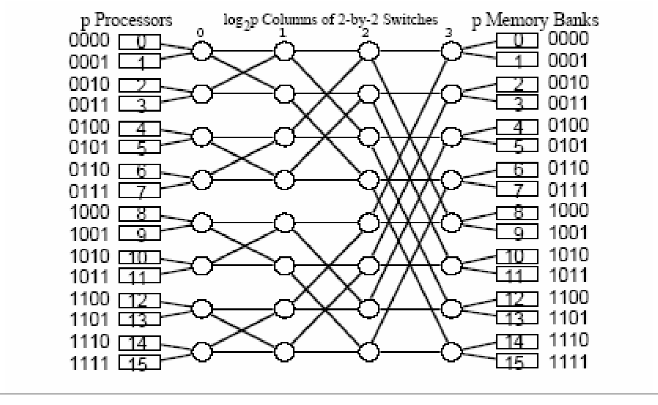
- Use two sets of channels (0-channels and 1-channels)
- Each set of channels cannot form cycles



- A message from a source, s_{q-1} , . . . , s_0, to a destination x_{q-1} , . . ., x_0 has to cross any dimension, *b*, for which $x_b \neq s_b$
- First, use 0-channels to cross those dimensions for which $s_b = 0$ and $x_b = 1$
- Then, use 1-channels to cross those dimensions for which $s_b = 1$ and $x_b = 0$.

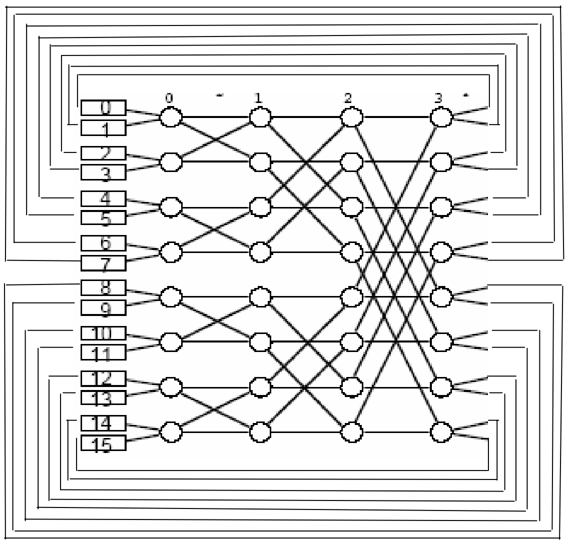
Multistage Interconnection Networks (MINs)

A modular way of building large switches from smaller switches



Example: 16x16 switch using 32 switches organized as 4 columns of 8 switch. Each switch is a 2x2 switch.

Can also be used to connect processors, each with its own memory (in a distributed memory system)



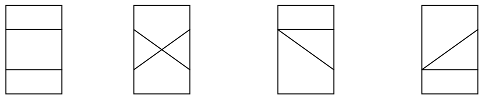
SIMD modes (synchronized communication):

Either use message routing and synchronize processors at each step, or set the switches before sending data (circuit switching).

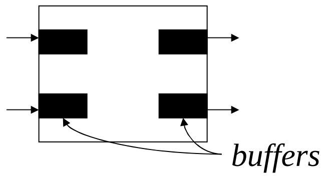
MIMD mode (unsynchronized communication):

Use circuit switching, packet switching (without synchronization), or wormhole routing (virtual circuit switching).

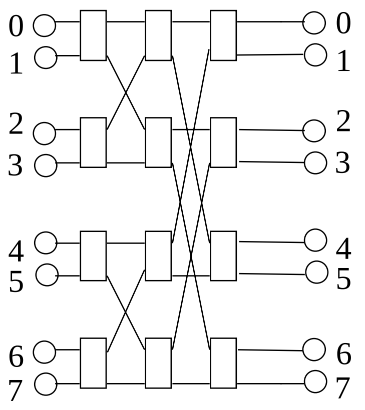
Circuit-switch states



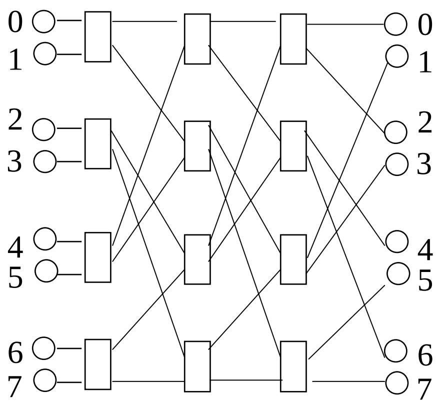
A routing switch



Examples of MINs connecting 2^q inputs to 2^q outputs (using q stages of 2x2 switches):



A butterfly network
(unfolded hypercube)



An Omega network
(multiple shuffle/exchange)

The perfect shuffle and the exchange functions

Shuffle(x_{q-1} , x_{q-2} , … , x_0) = x_{q-2} , … , x_0 , x_{q-1}

Exchange(x_{q-1} , x_{q-2} , … , x_0) = x_{q-1} , x_{q-2} , … , $\bar{x}_0$

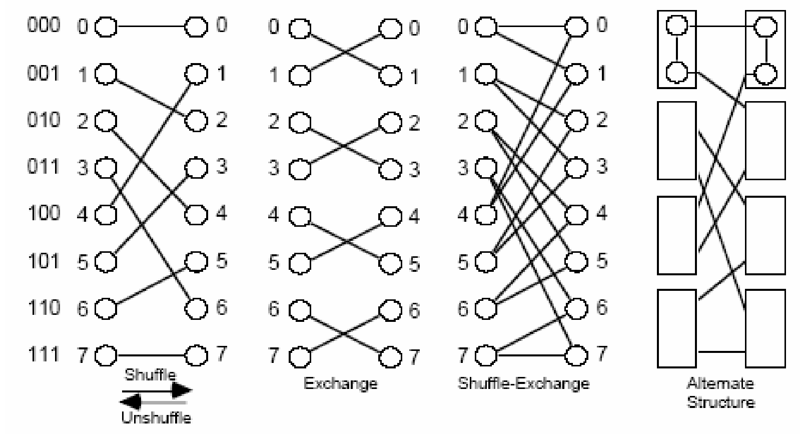


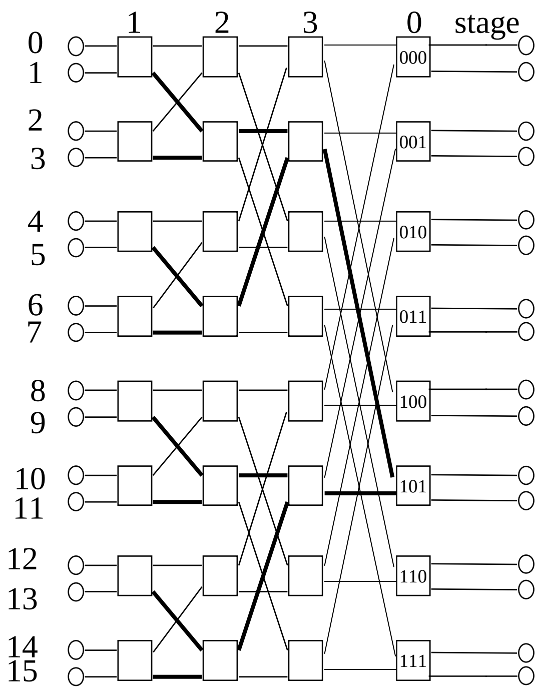
Fig. 15.16. Shuffle, exchange, and shuffle–exchange

- If 2x2 switches are used to build an *N*x*N* switch (to connect *N* processors – *N* being a power of 2), we need at least *log N* stages.

- Number of 2x2 switches =

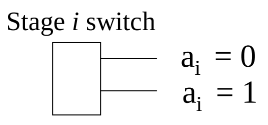
- If synchronous mode
  - Each switch is set to either cross or straight
  - A configuration = a specific setting of the 2x2 switches
  - How many possible configurations
  - Each configuration corresponds to a permutation (one to one communication pattern)
  - A log *N* stage MIN cannot realize all possible permutations (why?)

- A MIN that cannot realize all permutations is called **Blocking**.



Routing in a butterfly network

The destination $a_3\, a_2\, a_1\, a_0$ is coded in message header

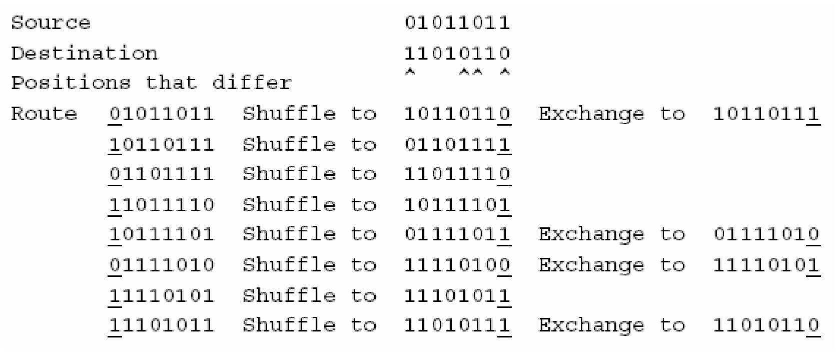


When a message is received:
Send to upper port if $a_i = 0$
Send to lower port if $a_i = 1$

Routing in an OMEGA network

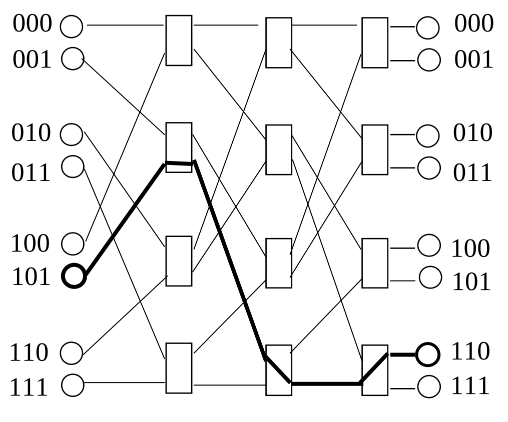
To get from $s_{q-1}, s_{q-2}, \ldots, s_0$ to $x_{q-1}, x_{q-2}, \ldots, x_0$

- Do q shuffles
- After each shuffle, do an exchange to match the corresponding destination bit



```
Source                          01011011
Destination                     11010110
Positions that differ           ^   ^^ ^
Route   01011011  Shuffle to    10110110  Exchange to   10110111
        10110111  Shuffle to    01101111
        01101111  Shuffle to    11011110
        11011110  Shuffle to    10111101
        10111101  Shuffle to    01111011  Exchange to   01111010
        01111010  Shuffle to    11110100  Exchange to   11110101
        11110101  Shuffle to    11101011
        11101011  Shuffle to    11010111  Exchange to   11010110
```

Routing in an OMEGA network

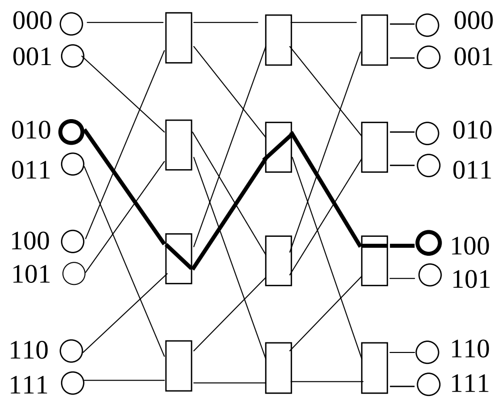


Example: to route from source 101 to destination 110 (xor = 011)

101 → 011 → 011 → 110 → 111 → 111 → 110

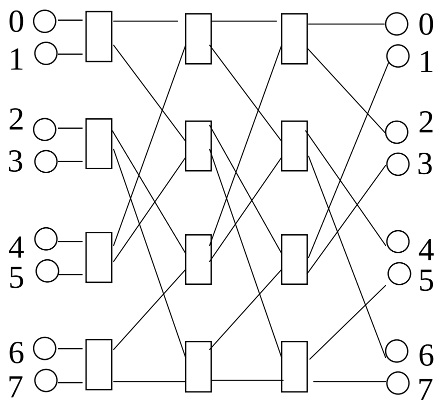
shuffle, straight, shuffle, exchange cross, shuffle, exchange cross

Routing in an OMEGA network



Example: to route from source 010 to destination 100
010 xor 100 = 110 = (cross, cross, straight)
Route: cross at level 0, cross at level 1, straight at level 2

How is routing in the following OMEGA network different?

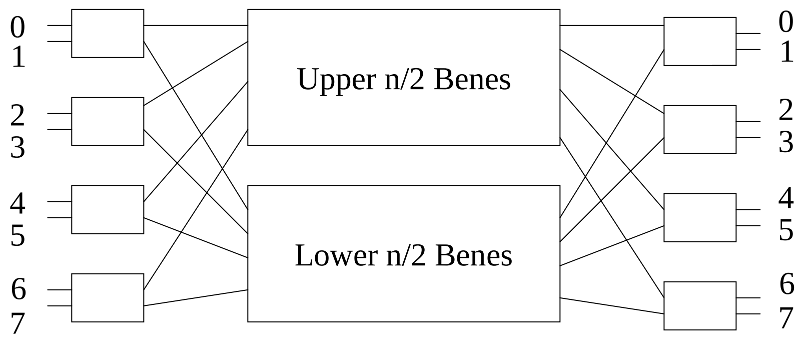


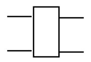
Capabilities of MINs for realizing arbitrary permutations:

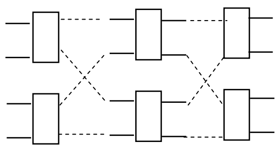
- *Blocking networks*: cannot realize an arbitrary permutation without conflict -- for example, Omega can realize only $n^{n/2}$ permutations.
- *Non-blocking networks*: can realize any permutation on-line -- for example, cross-bar switches.
- *Re-arrangeable networks*: can realize any permutation off-line -- for example, a Benes network can establish any connection in a permutation if it is allowed to re-arrange already established connections.

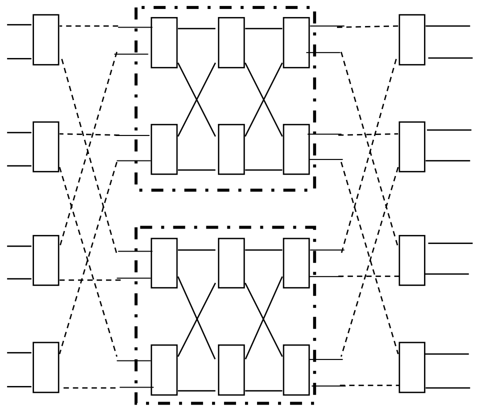
The Benes network

Can be built recursively -- an *n*x*n* Benes is built from two (*n/2* x *n/2)* Benes networks plus two columns of switches.

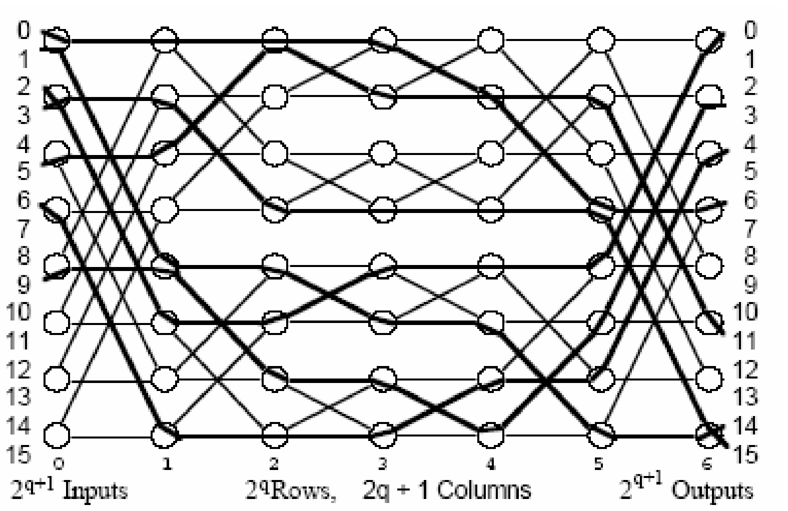



A 2x2 Benes network


A 4x4 Benes network



An 8x8 Benes network



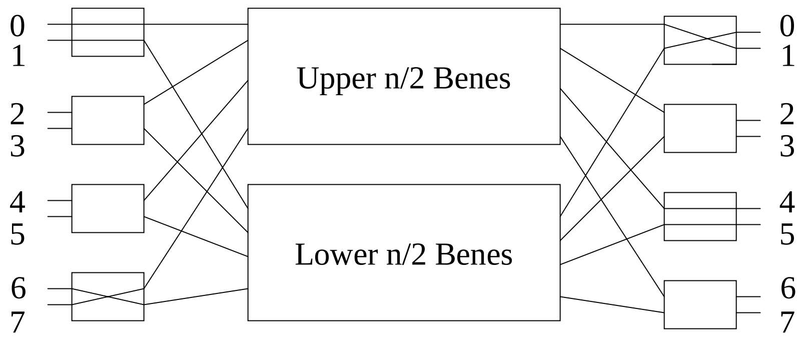
A 16x16 network Benes network

To realize a permutation (i , o_i , $i= 0,\ldots, n-1$) in an nxn Benes network:

- For each connection (i , o_i), determine whether it goes through the upper or lower n/2 Benes.
- Repeat this process recursively for each of the $n/2$ networks.

- Start with $k=0$ and (k, o_k) going through the upper Benes,
- If o_k shares a switch at the last column with o_m , then route (m, o_m) through the lower Benes.
- If j shares a switch at the first column with m, then route (j, o_j) through the upper Benes.
- Continue until you get an input that shares a switch with input 0.
- If there are still unsatisfied connections, repeat the looping.

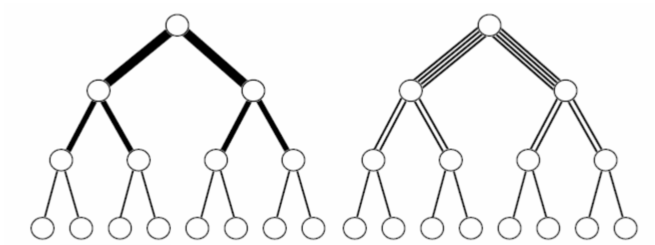
Example for establishing a permutation:
(0,4), (4,2), (3,6), (1,0), (2,3), (6,5), (5,7), (7,1)



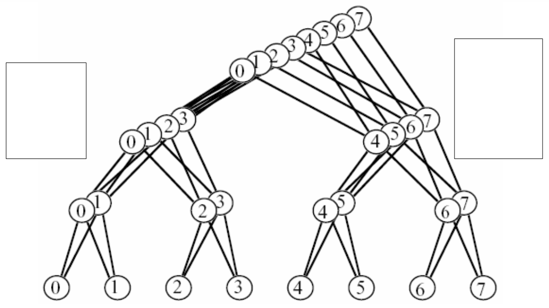
(0,4) upper,		(2,3) upper,
(6,5) lower,	+	(4,2) lower,
(7,1) upper,		(5,7) upper,
(1,0) lower,		(3,6) lower,

Fat tree networks

Eliminates the bisection bottleneck of a binary tree

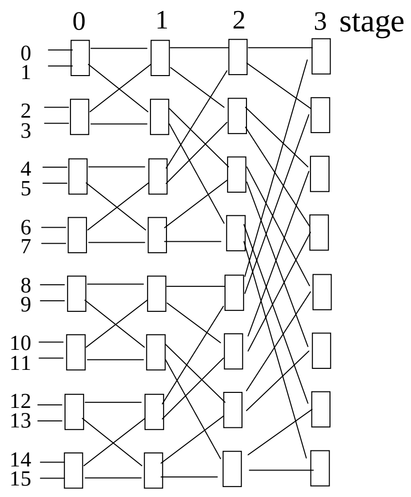


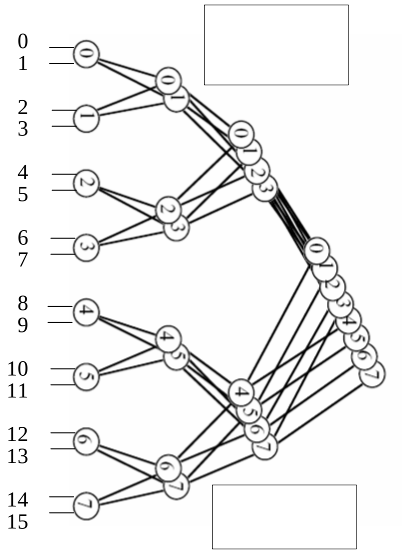
Two representations of a fat tree.



69

A 16-node fat tree networks

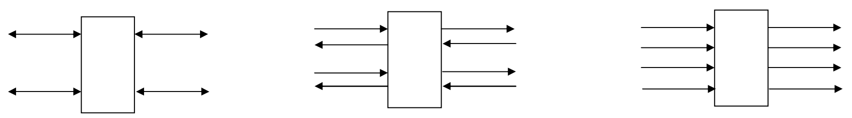




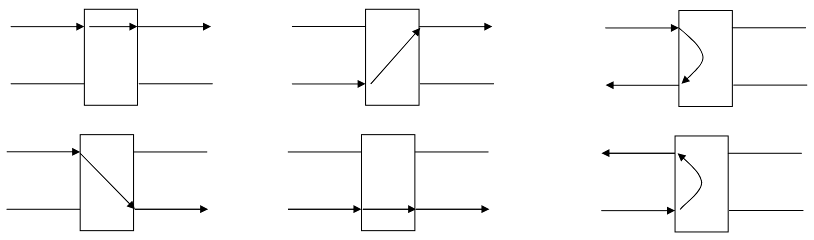
A fat tree networks using 2x2 bidirectional switches

70

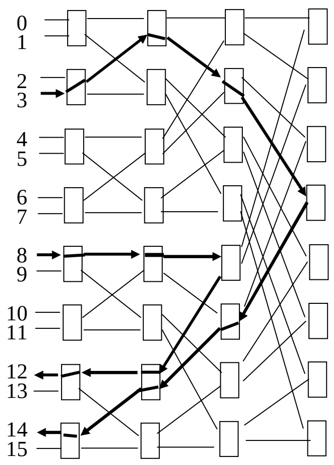
2x2 bidirectional switch = 4x4 uni-directional crossbar



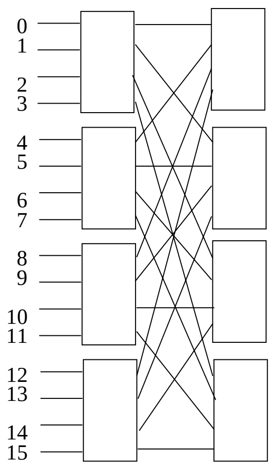
Possible routes



Only possibilities at stage *q*-1

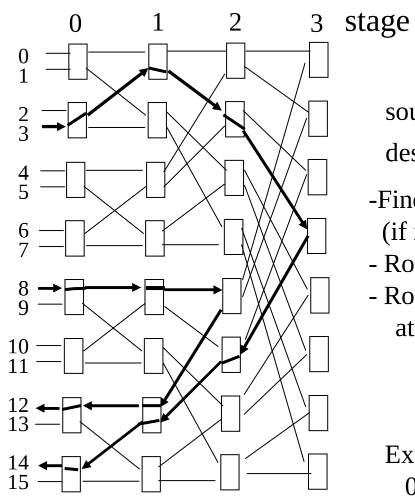


Routing in a fat tree
- multiple paths
- un-equal path lengths



A fat tree networks using 4x4 bidirectional switches

Routing in a fat tree



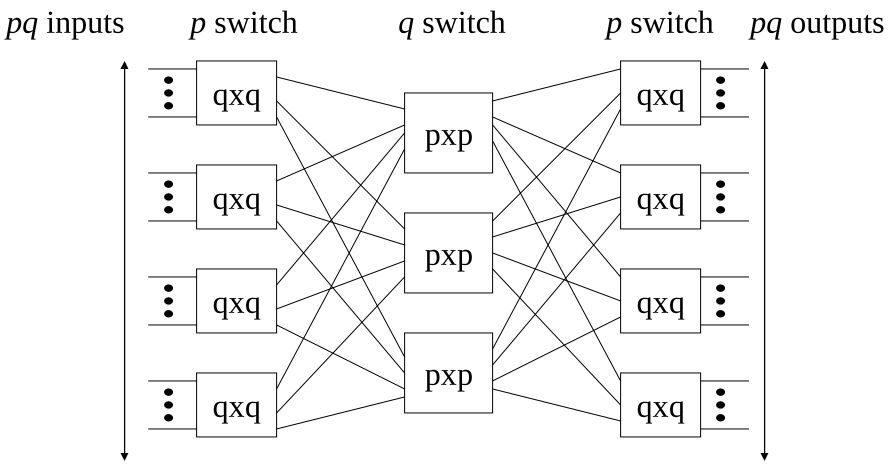
source $s_{q-1}, s_{q-2}, \ldots, s_0$

destination $x_{q-1}, x_{q-2}, \ldots, x_0$

-Find smallest k such that $s_i = x_i$, $i=k+1,\ldots,q-1$
(if no such k exists, then $k = q-1$)

- Route arbitrarily up the tree to a switch in stage k
- Route down the tree as follows:
  at stage i, $i= k, \ldots , 0$
    if $x_i = 0$, route to upper port
    else route to lower port

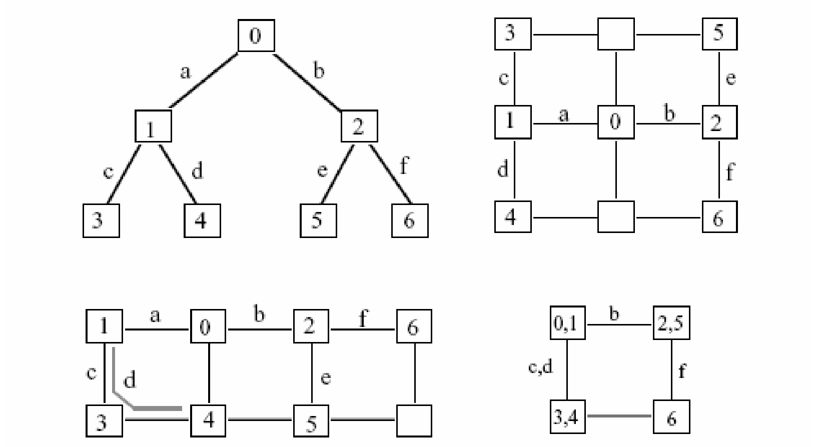
Examples (q = 4):
 0011 → 1110 (k = 3)
 1000 → 1100 (k = 2)



A Clos network (shown for $p = 4$)

Embedding task graphs into processors

Embedding a logical topology into a physical topology



Embedding a seven-node binary tree in 2D meshes of various sizes

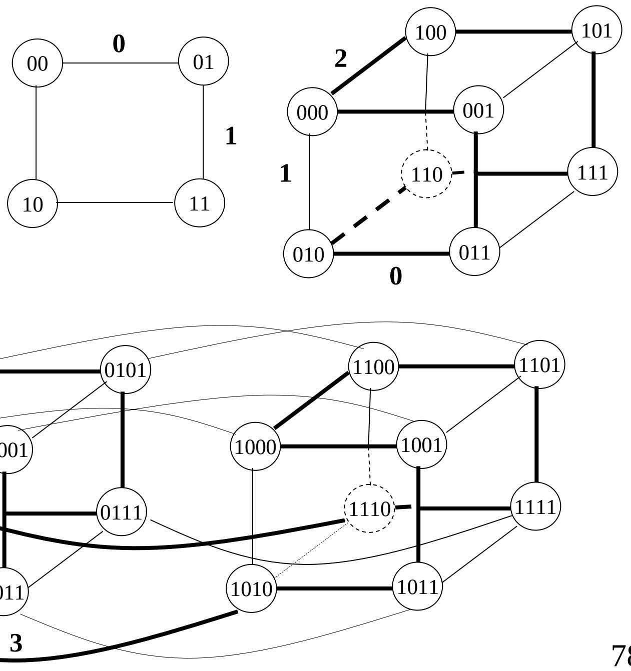
Embedding = node mapping + edge mapping

Properties of embeddings

	In the examples in the last slide		
	3x3	2x4	2x2
Dilation: length of the longest path to which a logical edge is mapped	1	2	1
Congestion: maximum number of logical edges mapped onto one physical edge	1	2	2
Load Factor: maximum number of logical nodes mapped onto one physical node	1	1	2
Expansion: ratio of the number of nodes in the two topologies	9/7	8/7	4/7

Why is each of the above factors important?

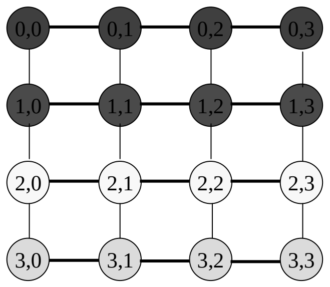
Embedding a linear array into a hypercube



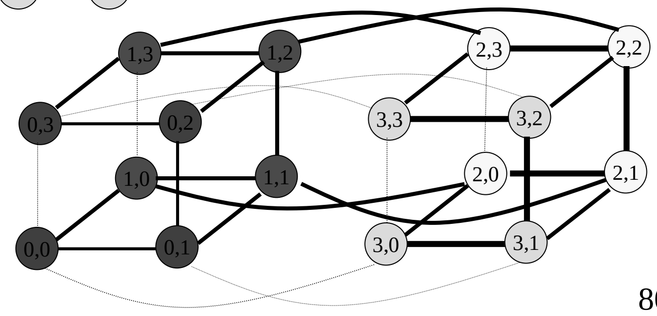
Binary Gray code (Hamming distance between any two code words = 1)

0	0 0	0 00	0 000
1	0 1	0 01	0 001
	1 1	0 11	0 011
	1 0	0 10	0 010
		1 10	0 110
		1 11	0 111
		1 01	0 101
		1 00	0 100
			1 100
			1 101
			1 111
			1 110
			1 010
			1 011
			1 001
			1 000

Embedding a 2D array into a hypercube



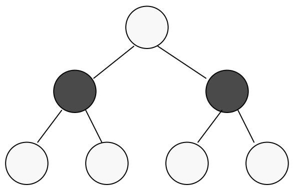
Theorem: we can embed a 2^a x 2^b array into a *(a+b)*-dimensional hypercube with dilation 1.

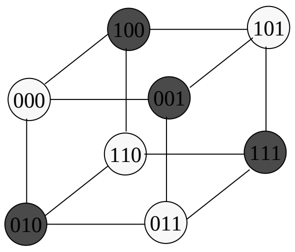


Embedding a complete binary tree into a hypercube

Theorem: we cannot embed a *2*^{q} -1 complete binary tree in a *q*-dimensional hypercube with dilation 1.

Proof: first divide the nodes in the cube to *odd nodes* (those with an odd number of 1's in the address) and *even nodes* (those with an even number of 1's in the address). To preserve unit dilation when embedding the tree, more than half the nodes need to be even (or odd) nodes. This is not possible.



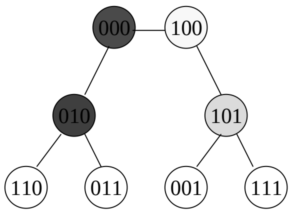


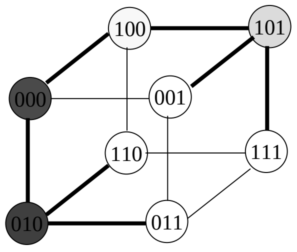
Embedding a double rooted tree into a hypercube

Theorem: we can embed a *2*^{q} double-rooted complete binary tree in a *q*-dimensional hypercube with dilation 1.

Will not provide a general proof but will give you an example for the embedding in the case of 8 nodes.

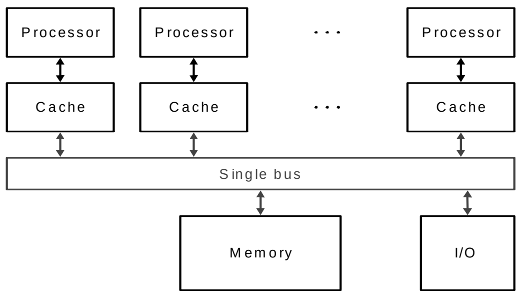
Note: embedding a double-rooted tree with dilation 1 is equivalent to embedding a sinngle rooted tree with dilation 2.





Cache coherence in SMP's

- Different caches may contain different value for the same memory location.

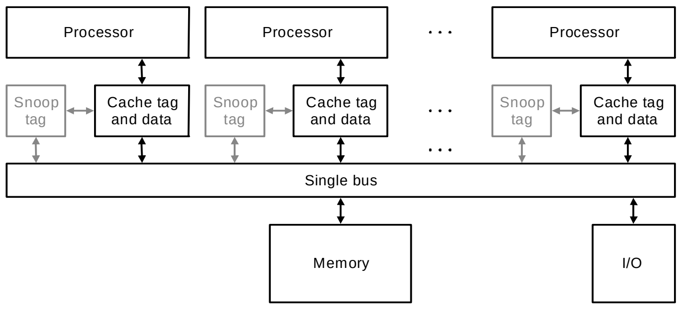


Time	Event	Cache Contents for CPU A	Cache Contents for CPU B	Memory Contents for location X
0				1
1	CPU A Reads X	X = 1		1
2	CPU B reads X	X = 1	X = 1	1
3	CPU A stores 0 into X	X = 0	X = 1	0

Approaches to cache coherence

- Do not cache shared data
- Do not cache writeable shared data
- Use snoopy caches (if connected by a bus)
- If GMSV not connected by a bus or DMSV (physically distributed memory), then need another solution – directory-based protocols.

Snooping cache coherence protocols



- Each processor monitors the activity on the bus
- **On a read miss**, all caches check to see if they have a copy of the requested block. If yes, they supply the data (will see how).
- **On a write miss**, all caches check to see if they have a copy of the requested data. If yes, they either invalidate the local copy, or update it with the new value.
- Can have either *write back* or *write through* policy.

Example: Write Invalidate

Processor Activity	Bus Activity	Cache Contents for CPU A	Cache Contents for CPU B	Memory Contents for location X
				0
CPU A Reads X	Cache Miss for X	0		0
CPU B Reads X	Cache Miss for X	0	0	0
CPU A writes 1 to X	Invalidation for X	1		0
CPU B Reads X	Cache Miss for X	1	1	1

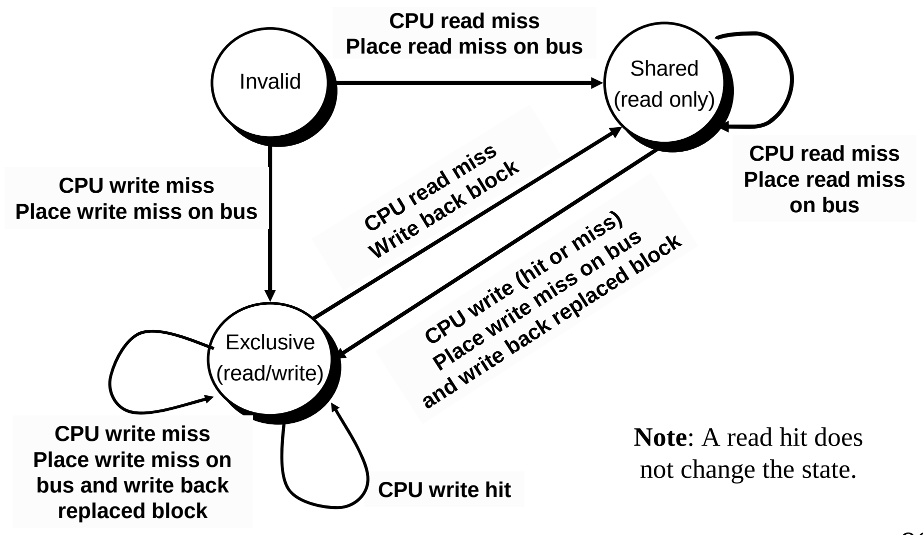
Example: Write update

Processor Activity	Bus Activity	Cache Contents for CPU A	Cache Contents for CPU B	Memory Contents for location X
				0
CPU A Reads X	Cache Miss for X	0		0
CPU B Reads X	Cache Miss for X	0	0	0
CPU A writes 1 to X	update for X	1	1	1
CPU B Reads X	Cache hit for X	1	1	1

An Example Snoopy Protocol

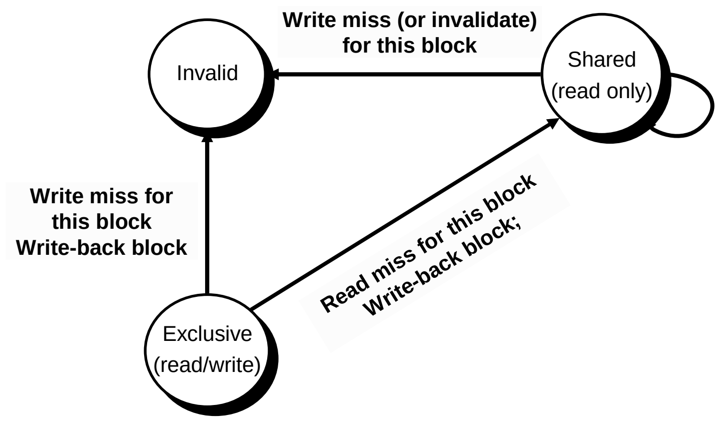
- Invalidation protocol, write-back cache
- Each block of memory is in one state:
  - Clean in all caches and up-to-date in memory (Read-Only),
  - Dirty in exactly one cache (Read/Write), OR
  - Not in any caches
- Each cache block is in one state:
  - **Shared** : block can be read (clean, read-only)
  - **Exclusive** : cache has only copy, its writeable, and dirty
  - **Invalid** : block contains no data
- Read misses: cause all caches to snoop bus
- Writes to clean blocks are treated as misses -- invalidates all other caches

Snoopy Cache State Machine (CPU Events)



Note: A read hit does not change the state.

Snoopy Cache State Machine (Bus Events)



Write miss (or invalidate) for this block
Invalid
Shared (read only)
Write miss for this block Write-back block
Read miss for this block Write-back block;
Exclusive (read/write)

Example

- Assumes A1 and A2 map to same cache block B.
- Initial cache state is invalid

Event	In P1's cache	In P2's cache
	B = invalid	B = invalid
P1 writes 10 to A1	A1 = 10 (exclusive)	B = invalid
P1 reads A1	A1 = 10 (exclusive)	B = invalid
P2 reads A1	A1 = 10 (shared)	A1 = 10 (shared)
P2 write 20 to A1	B = invalid	A1 = 20 (exclusive)
P2 writes 40 to A1	B = invalid	A2 = 40 (exclusive)

Directory-based coherence protocols

- For shared address space over physically distributed memory
- A controller decides if access is Local or Remote
- A *directory* that tracks state of every block in every cache (dirty, clean, …)
- Info per memory block vs. per cache block?
    - PLUS: In memory => simpler protocol (centralized/one location)
    - MINUS: In memory => directory is f(memory size) vs. f(cache size)

- With each block in each memory keep a state:
    - *Shared:* cached in one or more processors, and memory is up-to-date
    - *Uncached:* no processor has it; not valid in any cache)
    - *Exclusive*: 1 processor (owner) has data; memory out-of-date

- In addition to the state, must track *which processors* cached the block
- The owner (home) of each block in a cache is stored with the block.

Directory protocols

- No bus and don't want to broadcast:
    - interconnect no longer single arbitration point
    - all messages have explicit responses

- Keep it simple(r):
    - Processor blocks until access completes
    - Assume messages received and acted upon in order sent

- Typically 3 processors involved
    - Local node where a request originates
    - Home node where the memory location of an address resides
    - Remote node has a copy of a cache block

- Example messages on next slides:
  P = local node, H = home node, A = address (block)

Directory Protocol Messages

Message type	***Source***	***Destination***	***Msg Content***
Read miss	P	H	P, A
If A is shared or uncached, H sends "data (P,A)" message and sets State(A) = shared, If A is exclusive at another processor R, H sends "Fetch (P,H,A)" message to R			
Write miss	P	H	P, A
If A is uncached, H sends A to P and sets State(A) = exclusive If A is shared, H sends A to P, H sets State(A) = exclusive and send "Invalidate A" to every remote cache, R, sharing A, If A is exclusive at another processor R, H sends "Fetch (P,H,A)" message to R			
Invalidate	H	Remote node, R	A
R invalidates A in its cache.			
Fetch	H	Remote node, R	P,H,A
R fetches A and sends "data(P,A)" and "data(H,A)", and Invalidates A in local cache of R			
data	a processor	P	P,A
A message containing the data in A			