Introduction to High Performance Computing for Scientists and Engineers

Chapter 4: Parallel Computers

Parallel Computers

- World's fastest supercomputers have always exploited some degree of parallelism in their hardware
- With advent of multicore processors, virtually all computers today are parallel computers, even desktop and laptop computers
- Today's largest supercomputers have hundreds of thousands of cores and soon will have millions of cores
- Parallel computers require more complex algorithms and programming to divide computational work among multiple processors and coordinate their activities
- Efficient use of additional processors becomes increasingly difficult as total number of processors grows (*scalability*)

Flynn's Taxonomy

Computers can be classified by numbers of instruction and data streams

- SISD: single instruction stream, single data stream
  - conventional serial computers
- SIMD: single instruction stream, multiple data streams
  - vector or data parallel computers
- MISD: multiple instruction streams, single data stream
  - pipelined computers
- MIMD: multiple instruction streams, multiple data streams
  - general purpose parallel computers

SPMD Programming Style

SPMD (single program, multiple data): all processors execute same program, but each operates on different portion of problem data

- Easier to program than true MIMD but more flexible than SIMD
- Most parallel computers today have MIMD architecture but are programmed in SPMD style

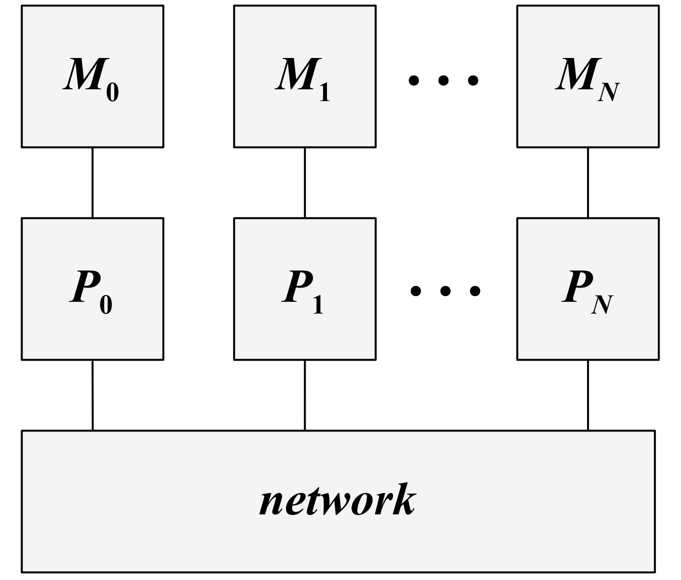
Parallel Computer Architectures

Parallel architectural design issues

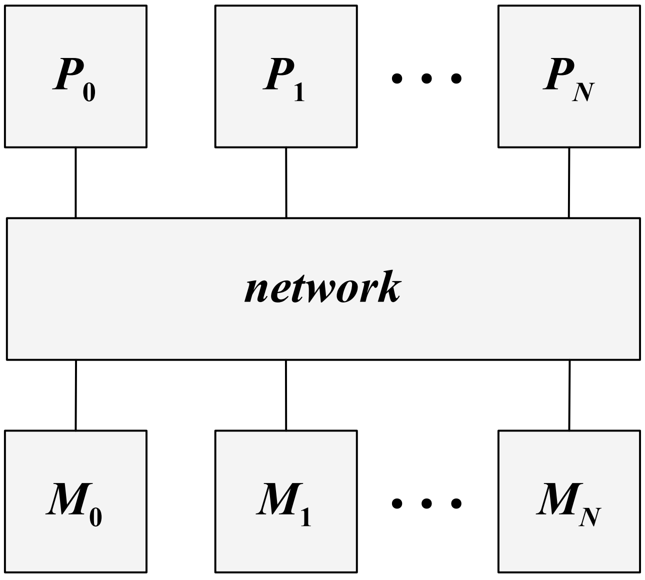
- Processor coordination: synchronous or asynchronous?
- Memory organization: distributed or shared?
- Address space: local or global?
- Memory access: uniform or nonuniform?
- Granularity: coarse or fine?
- Scalability: additional processors used efficiently?
- Interconnection network: topology, switching, routing?

Major Architectural Paradigms

Memory organization is fundamental architectural design choice: How are processors connected to memory?



distributed-memory multicomputer



shared-memory multiprocessor

Can also have hybrid combinations of these

Parallel Programming Styles

- Shared-memory multiprocessor
    - Entire problem data stored in common memory
    - Programs do loads and stores from common (and typically remote) memory
    - Protocols required to maintain data integrity
    - Often exploit loop-level parallelism using pool of tasks paradigm
- Distributed-memory multicomputer
    - Problem data partitioned among private processor memories
    - Programs communicate by sending messages between processors
    - Messaging protocol provides synchronization
    - Often exploit domain decomposition parallelism

Distributed vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity	easier	harder
incremental parallelization	harder	easier
automatic parallelization	harder	easier

Shared-Memory Computers

- UMA (uniform memory access): same latency and bandwidth for all processors and memory locations
  - sometimes called SMP (symmetric multiprocessor)
  - often implemented using bus, crossbar, or multistage network
  - multicore processor is typically SMP
- NUMA (nonuniform memory access): latency and bandwidth vary with processor and memory location
  - some memory locations “closer” than others, with different access speeds
  - consistency of multiple caches is crucial to correctness
  - ccNUMA: *cache coherent* nonuniform memory access

Cache Coherence

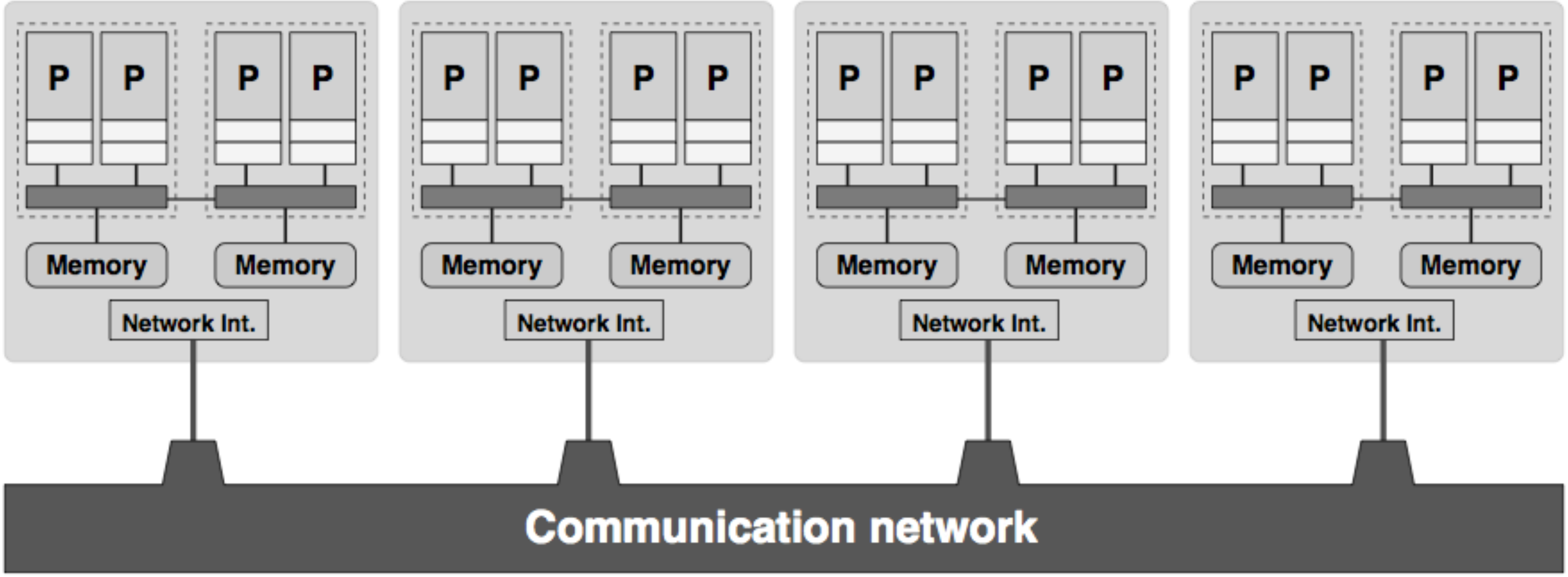
- In shared memory multiprocessor, same cache line in main memory may reside in cache of more than one processor, so values could be inconsistent
- Cache coherence protocol ensures consistent view of memory regardless of modifications of values in cache of any processor
- Cache coherence protocol keeps track of state of each cache line
- MESI protocol is typical
  - M, modified: has been modified, and resides in no other cache
  - E, exclusive: not yet modified, and resides in no other cache
  - S, shared: not yet modified, and resides in multiple caches
  - I, invalid: may be inconsistent, value not to be trusted

Cache Coherence

- Small systems often implement cache coherence using *bus snoop*
- Larger systems typically use *directory-based* protocol that keeps track of all cache lines in system
- Coherence traffic can hurt application performance, especially if same cache line is modified frequently by different processors, as in *false sharing*

Hybrid Parallel Architectures

- Most large computers today have hierarchical combination of shared and distributed memory, with memory shared locally within SMP nodes but distributed globally across nodes interconnected by network

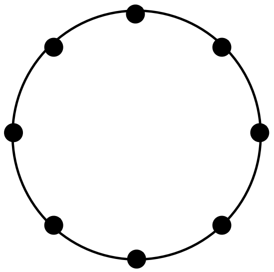


Communication Networks

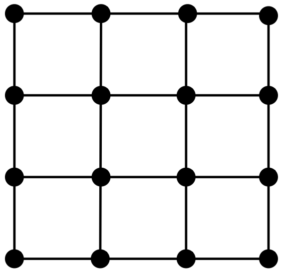
- Access to remote data requires communication
- Direct connections among p processors would require $O(p^2)$ wires and communication ports, which in infeasible for large p
- Limited connectivity necessitates routing data through intermediate processors or switches
- Topology of network affects algorithm design, implementation, and performance

Common Network Topologies

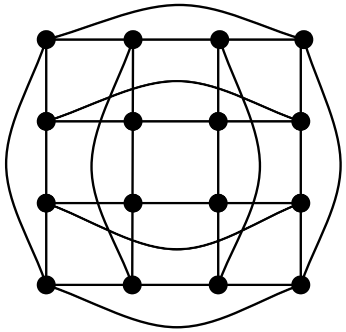
1-D *mesh*



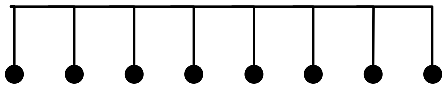
1-D *torus* (*ring*)



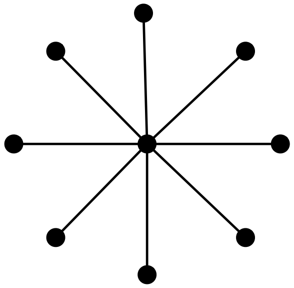
2-D *mesh*



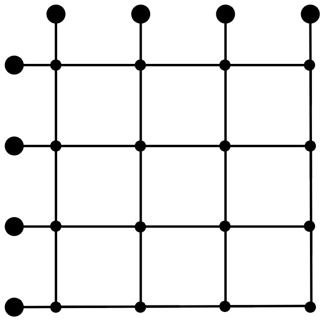
2-D *torus*



bus

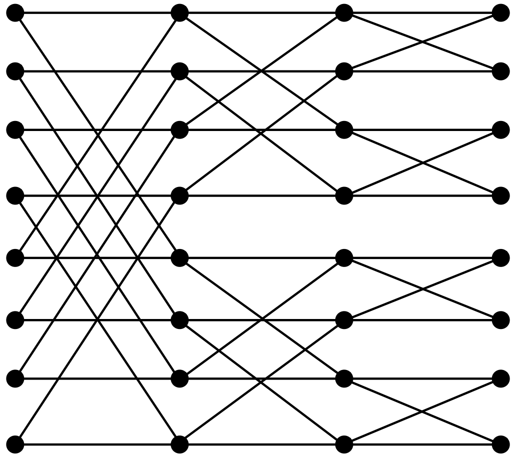


star

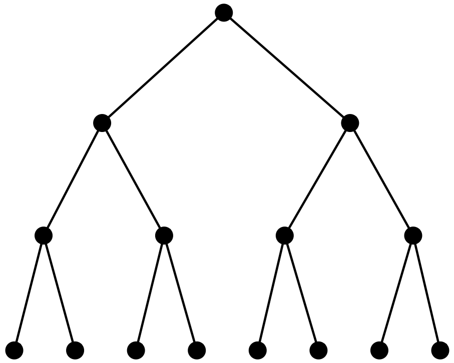


crossbar

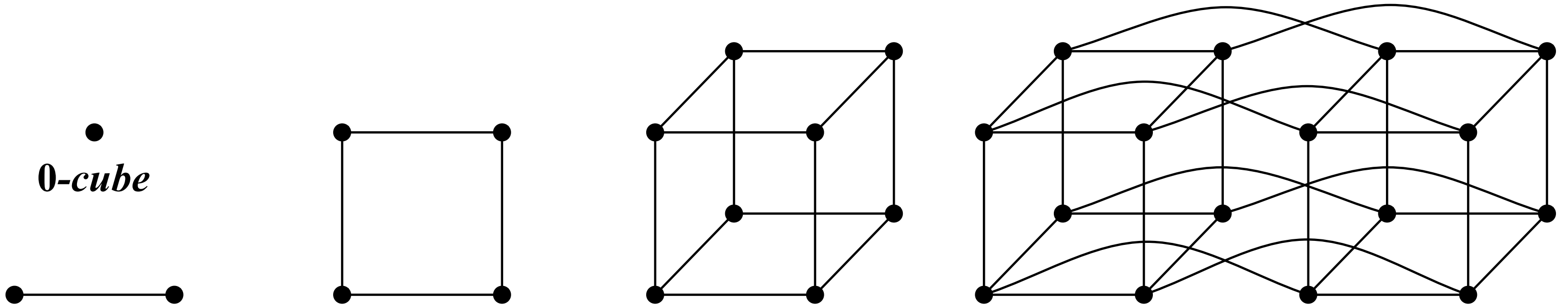
Common Network Topologies



butterfly



binary tree



0-*cube*

1-*cube*

2-*cube*

3-*cube*

4-*cube*

hypercubes

Graph Terminology

- *Graph*: pair (*V*, *E*), where *V* is set of vertices or nodes connected by set *E* of edges
- *Complete graph*: graph in which any two nodes are connected by an edge
- *Path*: sequence of contiguous edges in graph
- *Connected graph*: graph in which any two nodes are connected by a path
- *Cycle*: path of length greater than one that connects a node to itself
- *Tree*: connected graph containing no cycles
- *Spanning tree*: subgraph that includes all nodes of given graph and is also a tree

Graph Models

- Graph model of network: nodes are processors (or switches or memory units), edges are communication links
- Graph model of computation: nodes are tasks, edges are data dependences between tasks
- Mapping task graph of computation to network graph of target computer is instance of *graph embedding*
- *Distance* between two nodes: number of edges (hops) in shortest path between them

Network Properties

- *Degree*: maximum number of edges incident on any node
  - determines number of communication ports per processor
- *Diameter*: maximum distance between any pair of nodes
  - determines maximum communication delay between processors
- *Bisection width*: smallest number of edges whose removal splits graph into two subgraphs of equal size
  - determines ability to support simultaneous global communication
- *Edge length*: maximum physical length of any wire
  - may be constant or variable as number of processors varies

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const
n-D mesh	k^n	$2n$	$n(k-1)$	k^{n-1}	var
1-D torus	k	2	$k/2$	2	const
2-D torus	k^2	4	k	$2k$	const
3-D torus	k^3	6	$3k/2$	$2k^2$	var
n-D torus	k^n	$2n$	$nk/2$	$2k^{n-1}$	var
binary tree	2^k-1	3	$2(k-1)$	1	var
hypercube	2^k	k	k	2^{k-1}	var
butterfly	$(k+1)2^k$	4	$2k$	2^k	var

Graph Embedding

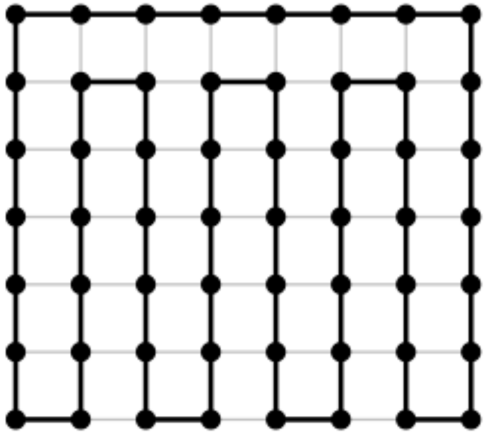
- *Graph embedding*: ϕ: $V_s \rightarrow V_t$ maps nodes in source graph $G_s = (V_s, E_s)$ to nodes in target graph $G_t = (V_t, E_t)$
- Edges in G_s mapped to paths in G_t
- *Load*: maximum number of nodes in V_s mapped to same node in V_t
- *Congestion*: maximum number of edges in E_s mapped to paths containing same edge in E_t
- *Dilation*: maximum distance between any two nodes $\phi(u)$, $\phi(v) \in V_t$ such that $(u,v) \in E_s$

Graph Embedding

- Uniform load helps balance work across processors
- Minimizing congestion optimizes use of available bandwidth of network links
- Minimizing dilation keeps nearest-neighbor communications in source graph as short as possible in target graph
- Perfect embedding has load, congestion, and dilation 1, but not always possible
- Optimal embedding difficult to determine (NP-complete, in general), so heuristics used to determine good embedding

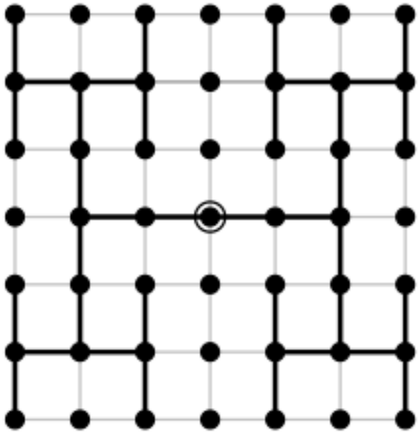
Graph Embedding Examples

- For some important cases, good or optimal embeddings are known



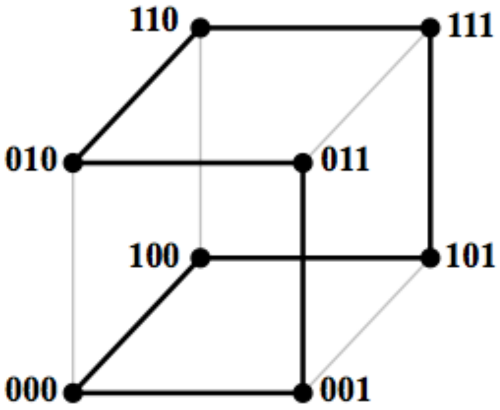
ring in 2-D mesh

dilation 1



binary tree in 2-D mesh

dilation $\lceil (k-1)/2 \rceil$



ring in hypercube

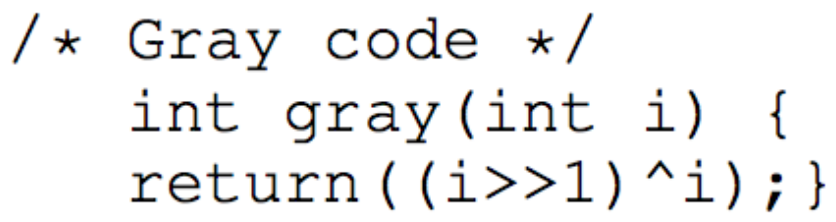
dilation 1

Gray Code

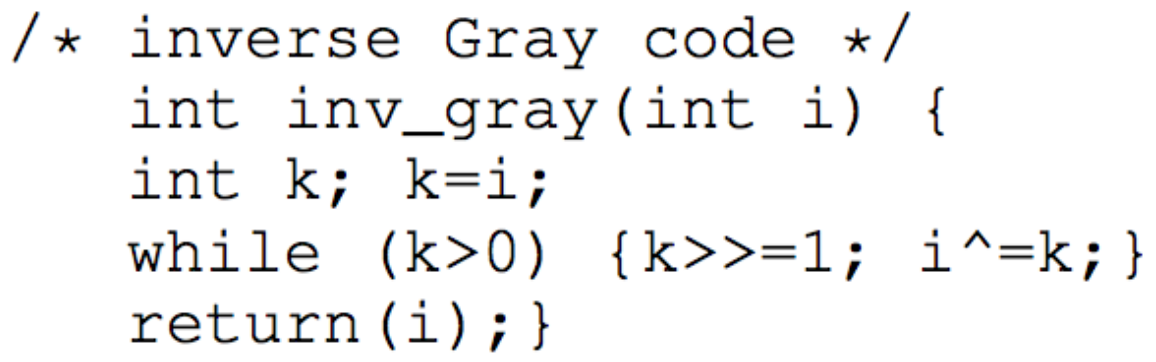
- *Gray code*: ordering of integers 0 to 2^{n-1} such that consecutive members differ in exactly one bit position
- Example: binary reflected Gray code of length 16

$$\begin{array}{rclcrcl}
0000 & = & 0 & \qquad & 1100 & = & 12 \\
0001 & = & 1 & & 1101 & = & 13 \\
0011 & = & 3 & & 1111 & = & 15 \\
0010 & = & 2 & & 1110 & = & 14 \\
0110 & = & 6 & & 1010 & = & 10 \\
0111 & = & 7 & & 1011 & = & 11 \\
0101 & = & 5 & & 1001 & = & 9 \\
0100 & = & 4 & & 1000 & = & 8
\end{array}$$

Computing Gray Code



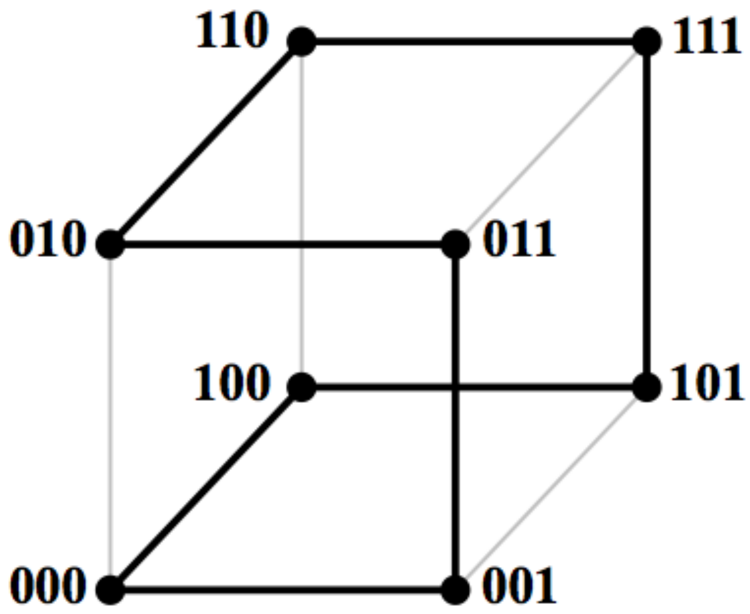
```
/* Gray code */
   int gray(int i) {
   return((i>>1)^i);}
```



```
/* inverse Gray code */
   int inv_gray(int i) {
   int k; k=i;
   while (k>0) {k>>=1; i^=k;}
   return(i);}
```

Hypercube Embeddings

- Visiting nodes of hypercube in Gray code order gives *Hamiltonian cycle* embedding ring in hypercube



- For mesh or torus of higher dimension, concatenating Gray codes for each dimension gives embedding in hypercube

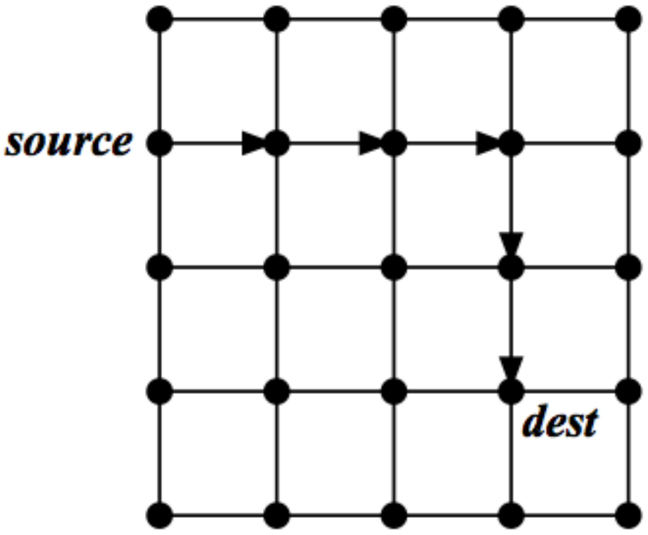
Communication Cost

- Simple model for time required to send message (move data) between adjacent nodes: $T_{\text{msg}} = t_s + t_w L$, where
  - t_s = startup time = latency (time to send message of length 0)
  - t_w = incremental transfer time per word (*bandwidth* $= 1/t_w$)
  - L = length of message in words
- For most real parallel systems, $t_s >> t_w$
- Caveats
  - Some systems treat message of length 0 as special case or may have minimum message size greater than 0
  - Many systems use different protocols depending on message size (e.g. 1-trip vs. 3-trip)

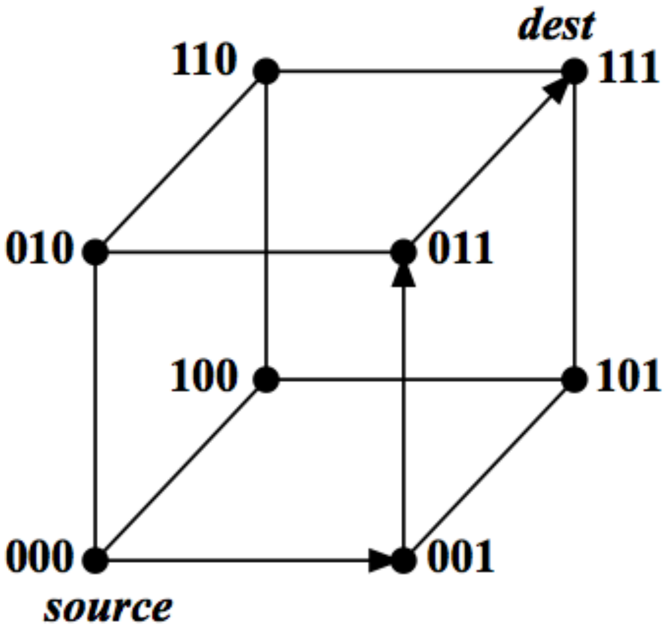
Message Routing

- Messages sent between nodes that are not directly connected must be routed through intermediate nodes
- Message routing algorithms can be
  - *minimal* or *nonminimal,* depending on whether shortest path is always taken
  - *static* or *dynamic,* depending on whether same path is always taken
  - *deterministic* or *randomized,* depending on whether path is chosen systematically or randomly
  - *circuit switched* or *packet switched,* depending on whether entire message goes along reserved path or is transferred in segments that may not all take same path
- Most regular network topologies admit simple routing schemes that are static, deterministic, and minimal

Message Routing Examples



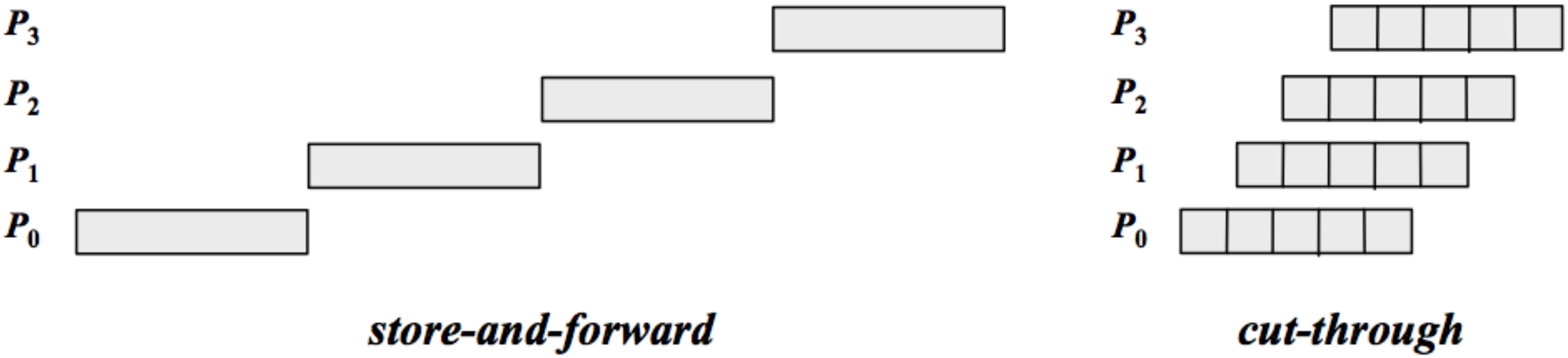
2-D mesh



hypercube

Routing Schemes

- *Store-and-forward* routing: entire message is received and stored at each node before being forwarded to next node on path, so $T_{\text{msg}} = (t_s + t_w L)\, D$, where D = distance in hops
- *Cut-through* (or *wormhole*) routing: message broken into segments that are pipelined through network, with each segment forwarded as soon as it is received, so $T_{\text{msg}} = t_s + t_w L + t_h D$, where t_h = incremental time per hop



Communication Concurrency

- For given communication system, it may or may not be possible for each node to
    - send message while receiving another simultaneously on *same* communication link
    - send message on one link while receiving simultaneously on *different* link
    - send or receive, or both, simultaneously on *multiple* links
- Depending on concurrency supported, time required for each step of communication algorithm is effectively multiplied by appropriate factor (e.g., degree of network graph)

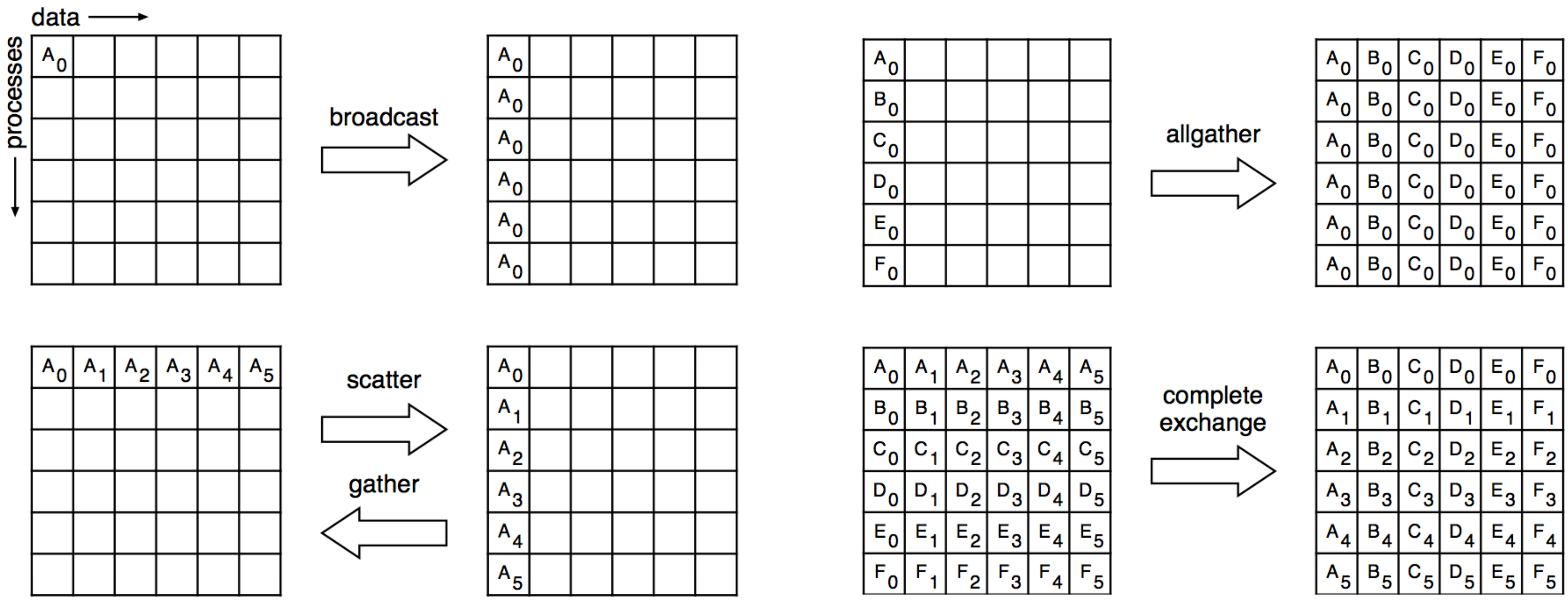
Communication Concurrency

- When multiple messages contend for network bandwidth, time required to send message modeled by $T_{\mathrm{msg}} = t_s + t_w\, S\, L$, where S is number of messages sent concurrently over same communication link
- In effect, each message uses $1/S$ of available bandwidth

Collective Communication

- *Collective communication*: multiple nodes communicating simultaneously in systematic pattern, such as
  - broadcast: one-to-all
  - reduction: all-to-one
  - multinode broadcast: all-to-all
  - scatter/gather: one-to-all/all-to-one
  - total or complete exchange: personalized all-to-all
  - scan or prefix
  - circular shift
  - barrier

Collective Communication

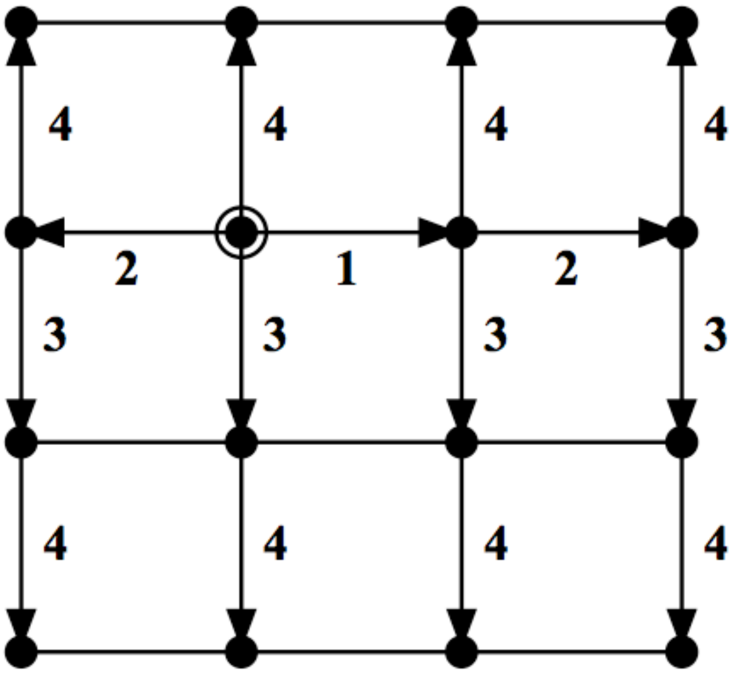


Broadcast

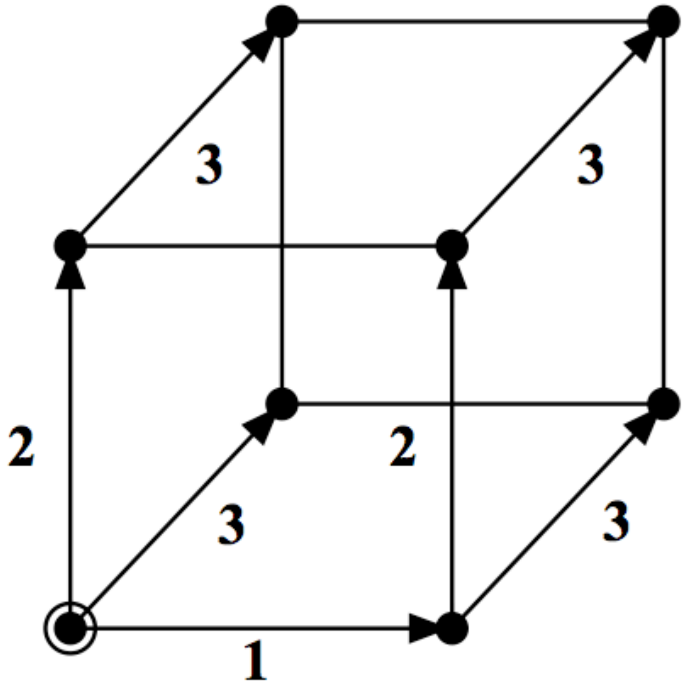
- *Broadcast*: source node sends same message to each of $p-1$ other nodes
- Generic broadcast algorithm generates spanning tree, with source node as root

```
if $source \neq me$ then
    receive message
end
for each neighbor
    if neighbor has not already received message then
        send message to neighbor
    end
end
```

Broadcast



2-D mesh



hypercube

Broadcast

- Cost of broadcast depends on network, for example
  - 1-D mesh: $T_{\text{bcast}} = (p - 1)\,(t_s + t_w L)$
  - 2-D mesh: $T_{\text{bcast}} = 2\,(\sqrt{p} - 1)\,(t_s + t_w L)$
  - hypercube: $T_{\text{bcast}} = \log p\,(t_s + t_w L)$
- For long messages, bandwidth utilization may be enhanced by breaking message into segments and either
  - pipeline segments along single spanning tree, or
  - send each segment along different spanning tree having same root
  - can also use scatter/allgather

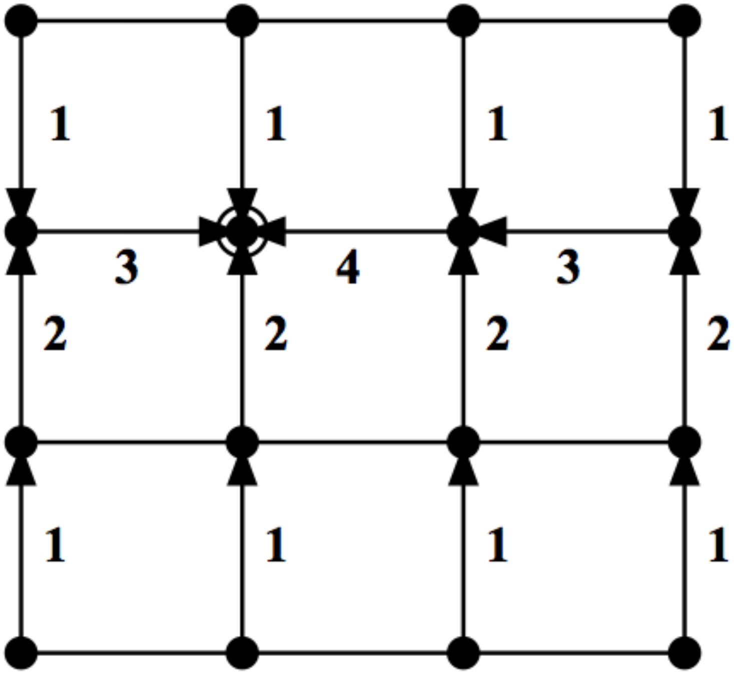
Reduction

- *Reduction*: data from all p nodes are combined by applying specified associative operation $\oplus$ (e.g., sum, product, max, min, logical OR, logical AND) to produce overall result

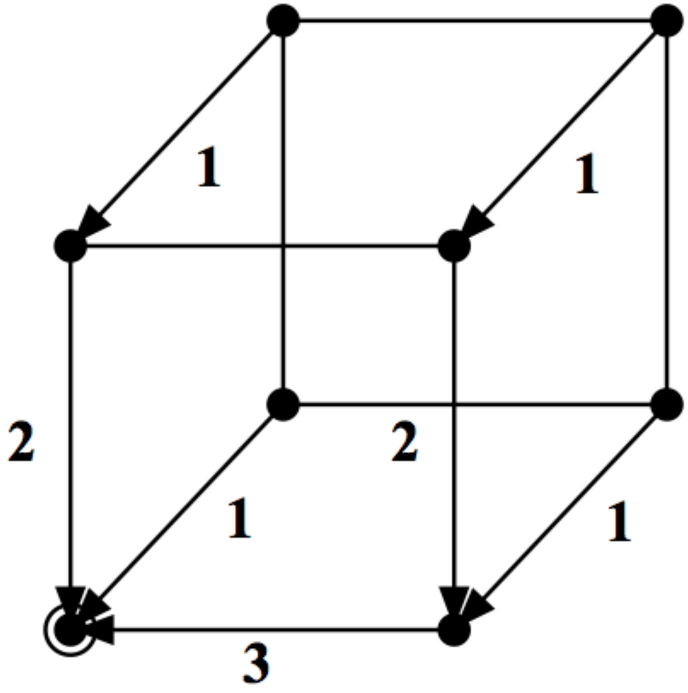
- Generic broadcast algorithm generates spanning tree, with source node as root

```
for each child in spanning tree
    receive value from child
    $my\_value = my\_value \oplus value$
end
if $root \neq me$ then
    send my_value to parent
end
```

Reduction



2-D mesh



hypercube

Reduction

- Subsequent broadcast required if all nodes need result of reduction
- Cost of reduction depends on network, for example
  - 1-D mesh: $T_{\text{bcast}} = (p - 1)\ (t_s + (t_w + t_c)\ L)$
  - 2-D mesh: $T_{\text{bcast}} = 2\ (\sqrt{p} - 1)\ (t_s + (t_w + t_c)\ L)$
  - hypercube: $T_{\text{bcast}} = \log p\ (t_s + (t_w + t_c)\ L)$
- Time per word for associative reduction operation, t_c, is often much smaller than t_w, so is sometimes omitted from performance analyses

Multinode Broadcast

- *Multinode broadcast*: each of p nodes sends message to all other nodes (all-to-all)
- Logically equivalent to p broadcasts, one from each node, but efficiency can often be enhanced by overlapping broadcasts
- Total time for multinode broadcast depends strongly on concurrency supported by communication system
- Multinode broadcast need be no more costly than standard broadcast if aggressive overlapping of communication is supported

Multinode Broadcast

- Implementation of multinode broadcast in specific networks
  - 1D torus (ring): initiate broadcast from each node simultaneously in same direction around ring; completes after $p - 1$ steps at same cost as single-node broadcast
  - 2D or 3D torus: apply ring algorithm successively in each dimension
  - hypercube: exchange messages pairwise in each of $\log p$ dimensions, with messages concatenated at each stage
- Multinode broadcast can be used to implement reduction by combining messages using associative operation instead of concatenation, which avoids subsequent broadcast when result needed by all nodes

Multinode Reduction

- *Multinode reduction*: each of *p* nodes is destination of reduction from all other nodes
- Algorithms for multinode reduction are essentially reverse of corresponding algorithms for multinode broadcast

Personalized Communication

- *Personalized collective communication*: each node sends (or receives) distinct message to (or from) each other node
  - *scatter*: analogous to broadcast, but root sends different message to each other node
  - *gather*: analogous to reduction, but data received by root are concatenated rather than combined using associative operation
  - *total exchange*: analogous to multinode broadcast, but each node exchanges different message with each other node

Scan or Prefix

- *Scan* (or *prefix*): given data values $x_0, x_1, \ldots, x_{p-1}$, one per node, along with associative operation $\oplus$, compute sequence of partial results $s_0, s_1, \ldots, s_{p-1}$, where $s_k = x_0 \oplus x_1 \oplus \cdots \oplus x_k$ and s_k is to reside on node k, $k = 0, \ldots, p-1$
- Scan can be implemented similarly to multinode broadcast, except intermediate results received by each node are selectively combined depending on sending node's numbering, before being forwarded

Circular Shift

- *Circular k-shift*: for $0 < k < p$, node i sends data to node $(i + k) \bmod p$
- Circular shift implemented naturally in ring network, and by embedding ring in other networks

Barrier

- *Barrier*: synchronization point that all processes must reach before any process is allowed to proceed beyond it
- For distributed-memory systems, barrier usually implemented by message passing, using algorithm similar to all-to-all
    - Some systems have special network for fast barriers
- For shared-memory systems, barrier usually implemented using mechanism for enforcing mutual exclusion, such as test-and-set or semaphore, or with atomic memory operations