# **Interconnection Networks for Parallel Computers**

(cf. Grama et al.)

# Interconnection Networks for Parallel Computers

- **Interconnection networks carry data between the processor and memory**
- The interconnections are implemented through **switches** and **links** (wires, fiber)
- The interconnections are classified as **static** or **dynamic**
- **Static networks** consist of point-to-point communications between nodes and are referred to as **direct** networks
- **Dynamic networks** are implemented using switches and communication links. They are also called **indirect** networks

# Static and Dynamic Interconnection Networks

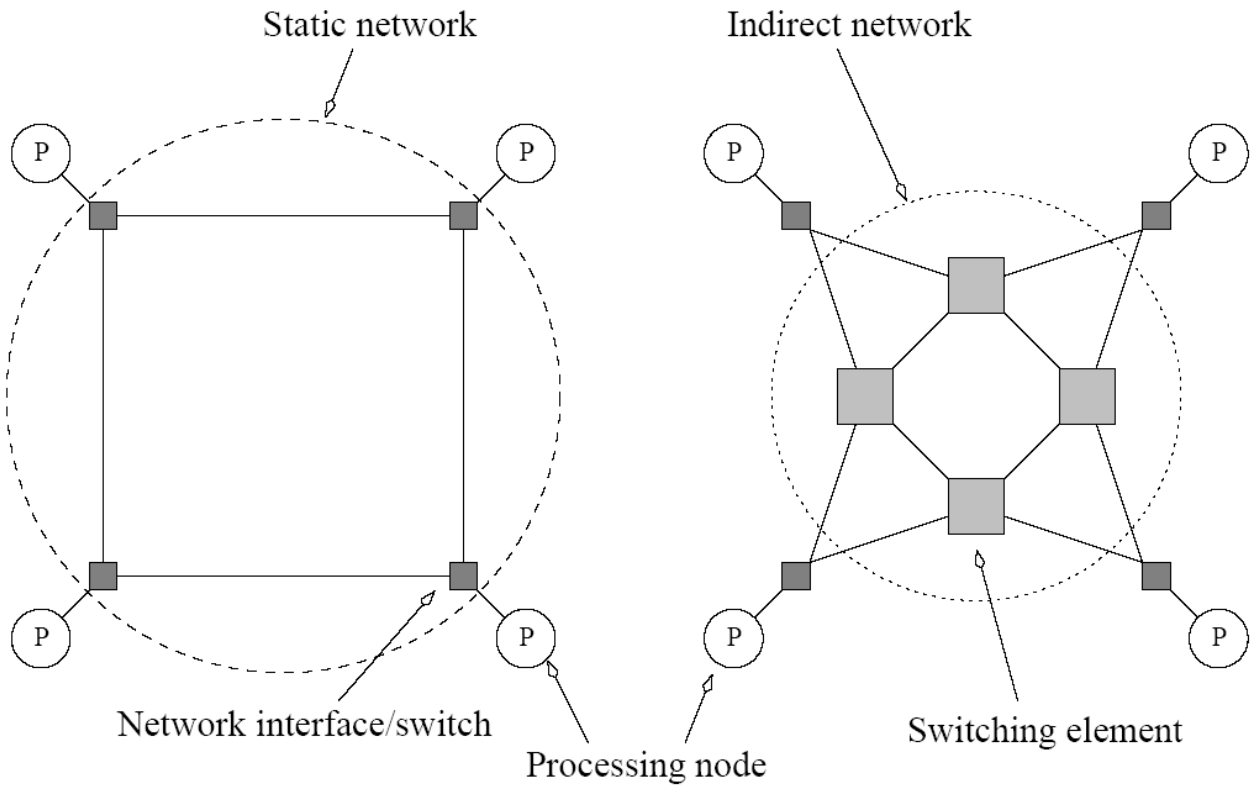



**Figure 2.6** Classification of interconnection networks: (a) a static network; and (b) a dynamic network.

# Metrics for the evaluation of networks

- **Diameter**
  - Maximum distance between 2 nodes (**better small diameters**)
- **Connectivity**
  - Minimum number of arcs that have to be removed to divide the network in 2 disconnected networks (**better high connectivity**)
- **Bisection bandwidth**
  - Applied to a network of weighted arcs, where weights indicate the quantity of data that can be transferred
  - Minimum volume of communications permitted between 2 halves of a network (**better high**)
- **Cost**
  - Number of links of the network (**better small**)

# Dynamic Networks

# Network Topologies: Buses

- Some of the earliest and simplest parallel machines used **buses**
- All processors have a common bus for the exchange of data
- The distance between any two nodes is **O(1)**. The bus also provides a convenient means of broadcast
- However, the bandwidth of the shared bus is a significant **bottleneck**
- Bus-based machines are limited to some tens of nodes. **Examples**: Sun Enterprise servers and Intel-based shared-bus multiprocessors. Our notebooks!

# Network Topologies: Buses

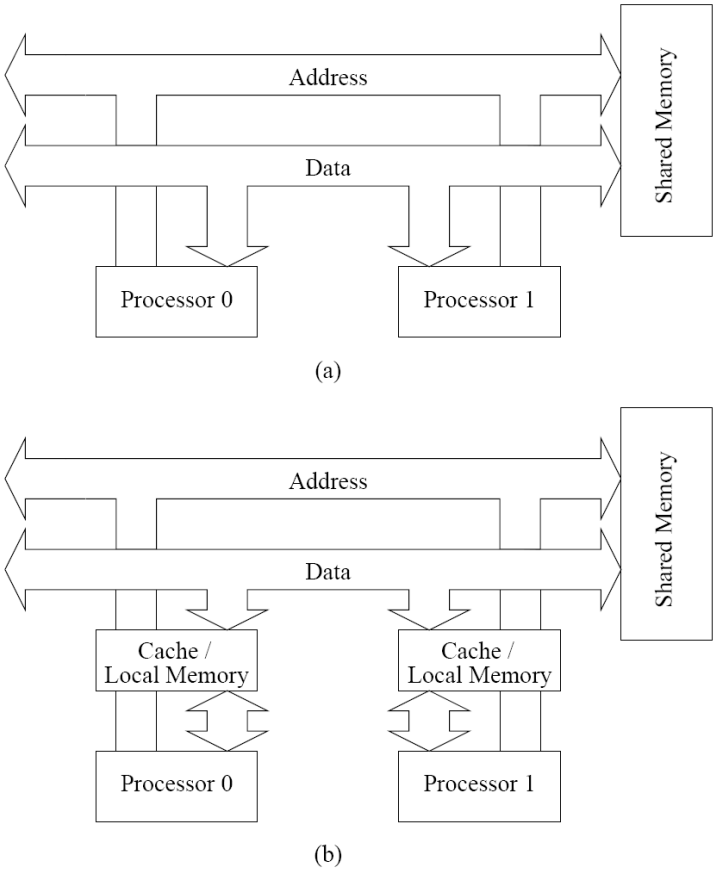



**Figure 2.7** Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

**NB** Given that the majority of the data accessed by the processor is local, a local memory (e.g. cache) for each node can improve the performance of such machines

# Network Topologies: Crossbars

A crossbar network uses a **p × b grid** of switches to connect **p inputs** to **b outputs** in a non-blocking manner

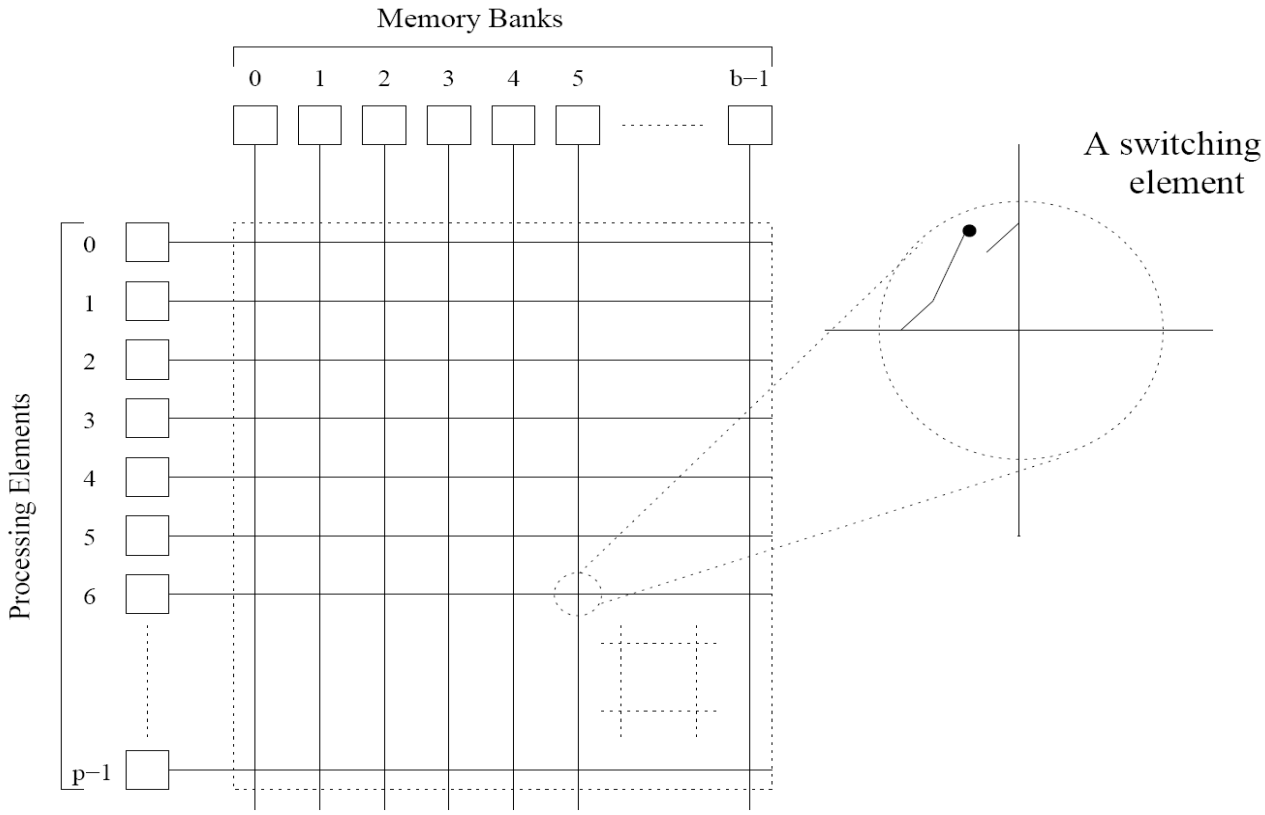



**Figure 2.8** A completely non-blocking crossbar network connecting $p$ processors to $b$ memory banks.

# Network Topologies: Crossbars

- The cost of a crossbar of **p** processors grows as $\mathbf{O(p^2)}$
- Hence, it is generally **difficult** to achieve **good scalability** in terms of cost for large values of **p**
- **Examples** of machines that use crossbars are Sun Ultra HPC 10000 and the Fujitsu VPP500

# Network Topologies: Multistage Networks

- **Crossbars** have excellent scalability performance but poor cost scalability
- **Buses** have excellent cost scalability but poor performance scaling
- **Multistage networks look for a balance between the two**

# Network Topologies: Multistage Networks

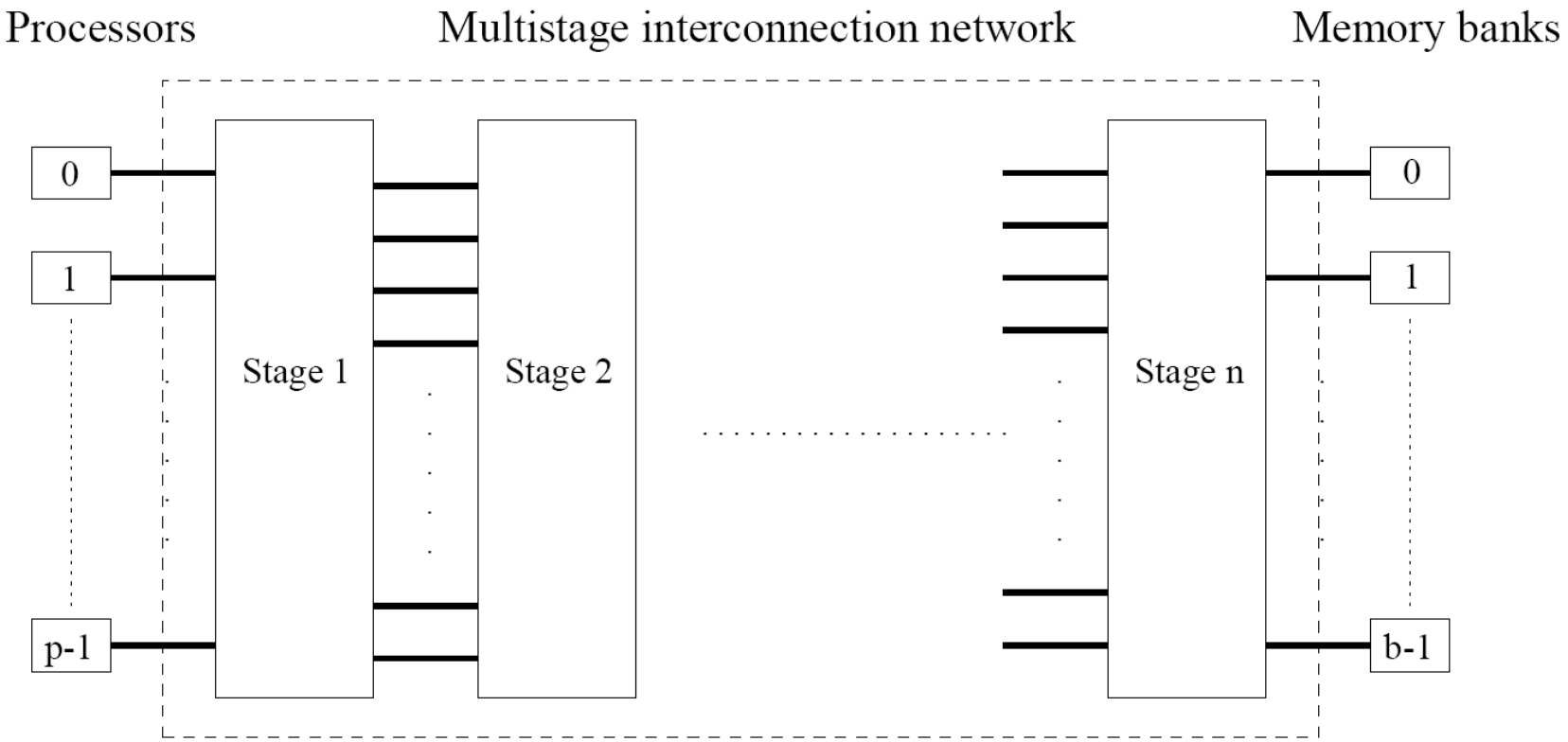



**Figure 2.9** The schematic of a typical multistage interconnection network.

# Network Topologies: Omega Multistage Networks

- One of the most well-known multistage networks is the **OMEGA** network
- This network consists of ***log p*** steps, where ***p*** is the number of inputs/outputs
- At each stage, the ***i*** input is connected to output ***j*** if (**left_rotation**):

$$j = \begin{cases} 2i, & 0 \leq i \leq p/2 - 1 \\ 2i + 1 - p, & p/2 \leq i \leq p - 1 \end{cases}$$

# Network Topologies: Omega Multistage Networks

**Each stage** of the Omega network implements a **perfect shuffle** as follows:

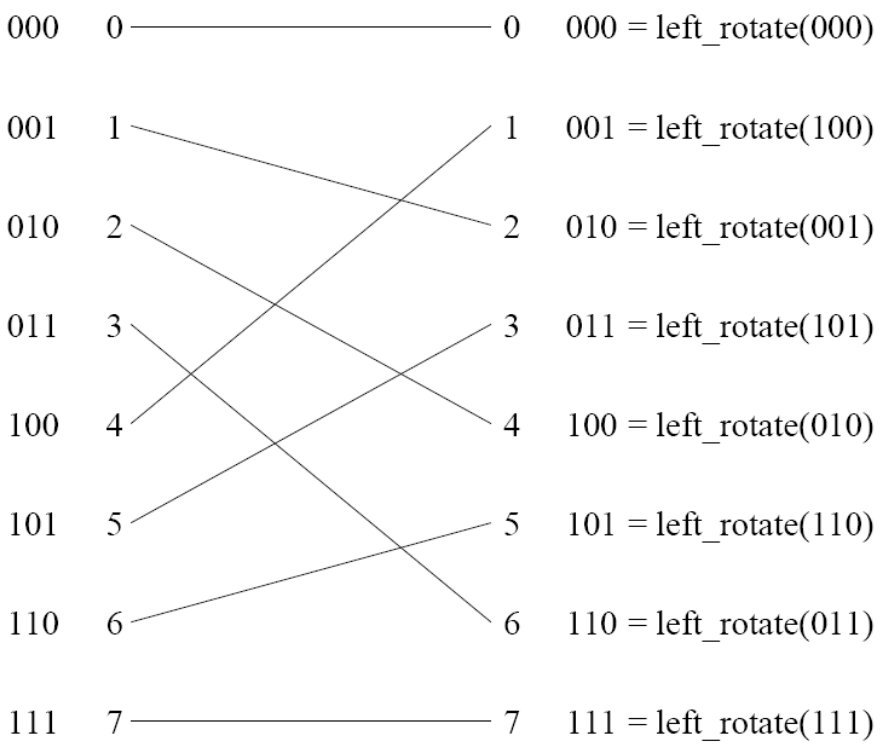



**Figure 2.10** A perfect shuffle interconnection for eight inputs and outputs.

# Network Topologies: Omega Multistage Networks

- The **perfect shuffle** patterns are connected using 2 × 2 switches
- The switches operate in two ways: **crossover** or **pass-through**


**Figure 2.11** Two switching configurations of the $2 \times 2$ switch: (a) Pass-through; (b) Cross-over.

# Network Topologies: Omega Multistage Networks

A complete Omega network with a perfect shuffle

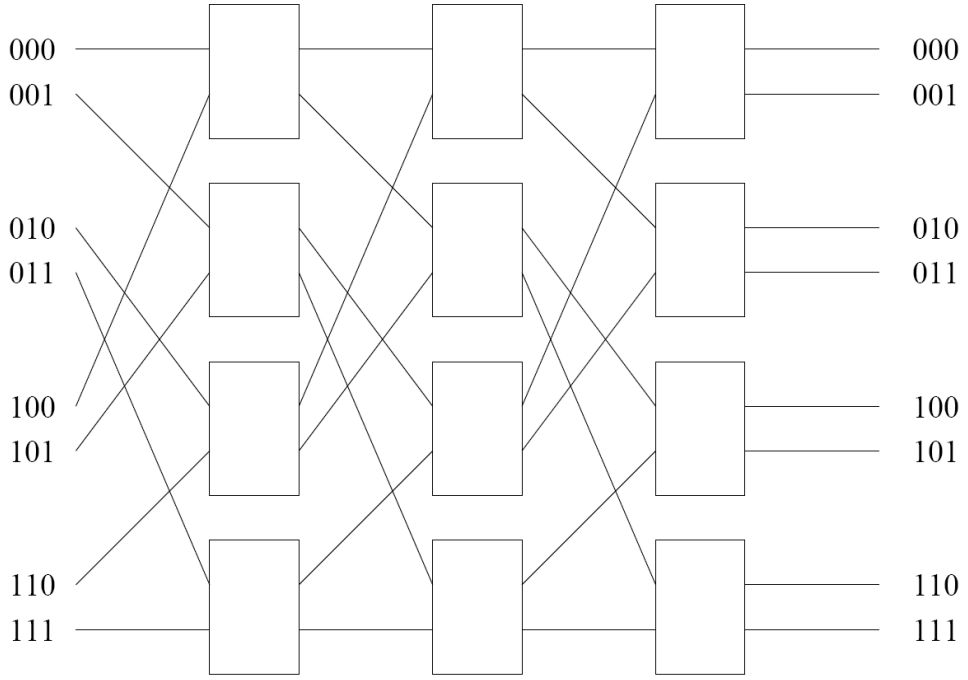



**Figure 2.12** A complete omega network connecting eight inputs and eight outputs.

An omega network has p/2 × log p switching nodes, and the cost of such a network grows as (p log p).

# Network Topologies: Omega Multistage Networks – Routing

- Let **s** be the binary representation of the source node and **d** the destination node
- Data crosses the link to the first node of the switch. **If** the most significant bits of **s** and **d** are the same, then data is routed by the switch in **pass-through** mode, or it will be in **crossover** mode
- This process is **repeated** for each of the **log p** switching stages (taking into consideration the **next most significant** bit)
- Note that this is **not** a non-blocking switch (i.e., not good!)

# Network Topologies: Omega Multistage Networks – Routing

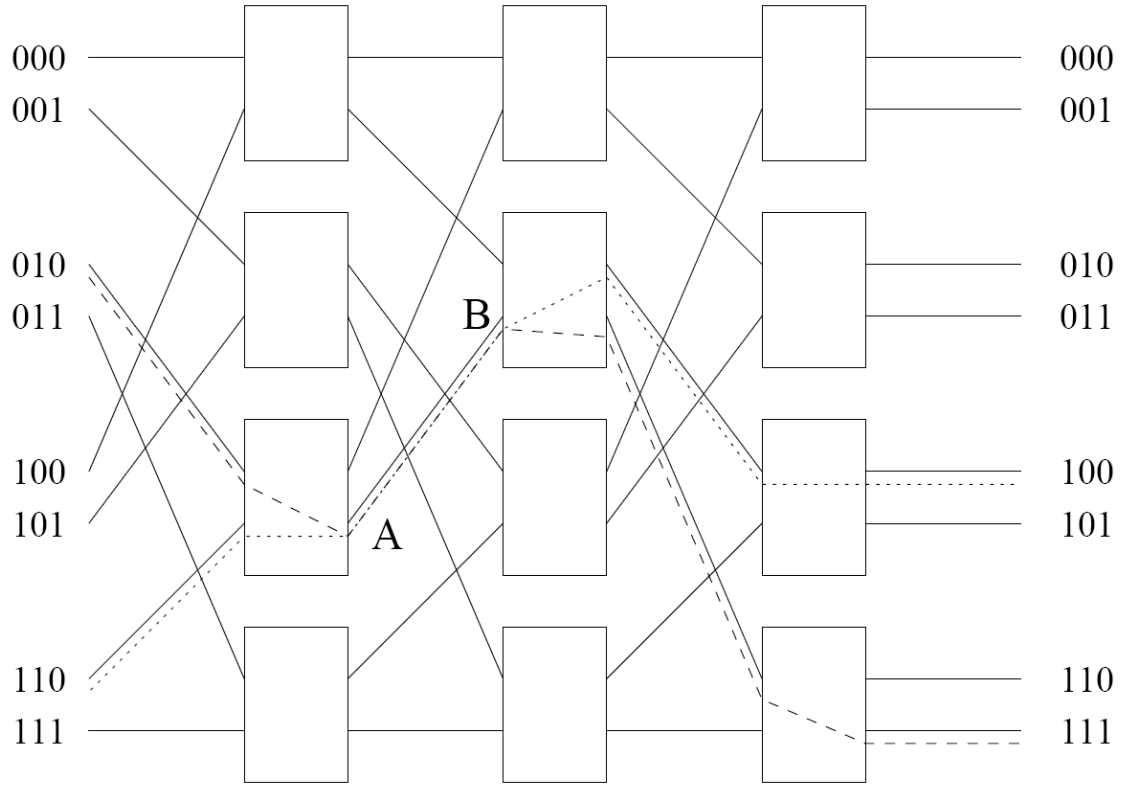



**Figure 2.13** An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.

# Static Networks

# Networks Topologies: Star Networks and Fully Interconnected Networks

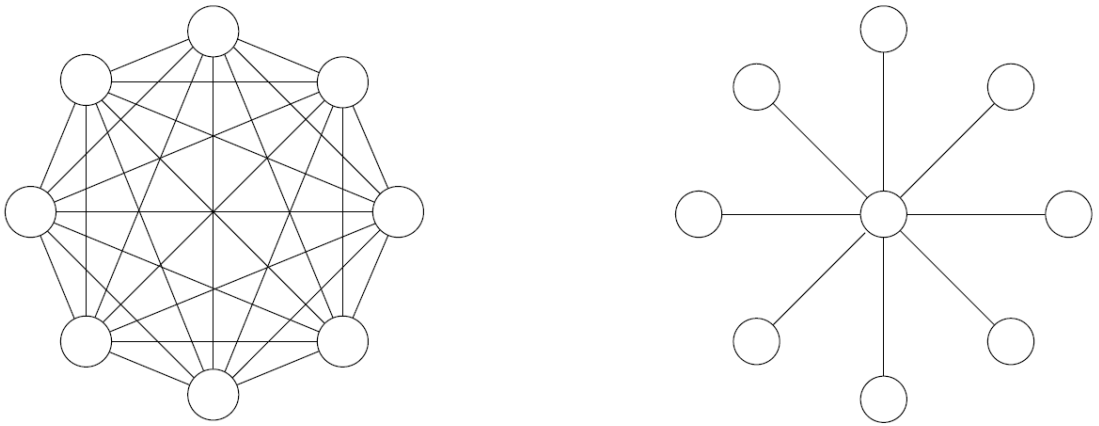



**Figure 2.14** (a) A completely-connected network of eight nodes; (b) a Star connected network of nine nodes.

# Networks Topologies: Fully Interconnected Networks

- **Each** processor is connected to every other processor
- The number of links in the network scales as $O(p^2)$
- While the scalability of performance is very good, the **hardware complexity** is not feasible for **large** values of p
- In this sense, these networks are the **static counterpart** of the crossbar

# Networks Topologies: Star Networks

- Each node is connected to a common "central" node
- The distance between any two nodes is O(1). However, the central node can become a **bottleneck**
- In this sense, star networks are **static counterparts** of bus networks

# Networks Topologies
# Linear Arrays, Meshes and *k-d* Meshes

- In a **linear array**, each node has two neighbors, one at the left and one to the right. If the terminal nodes are connected, we refer to a **1-D torus** or **ring**
- A generalization to two dimensions has nodes with 4 neighbors to the north, south, east and west
- A **generalization** to more dimensions has nodes with 2d neighbors
- A special case of d-dimensional mesh is the **hypercube**. In this case, d = log p, where p is the total number of nodes

# Networks Topologies : Linear Arrays, Bi- and Tri-Dimensional Meshes

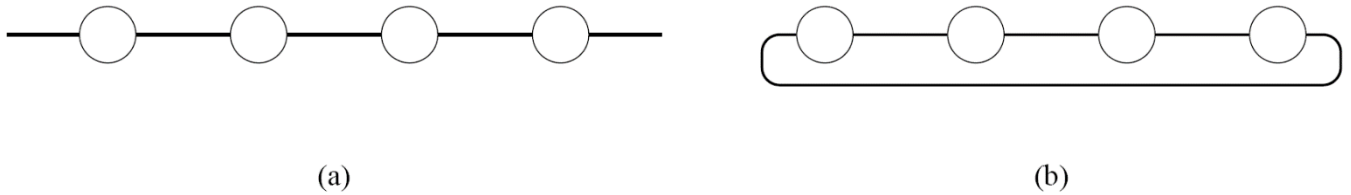



**Figure 2.15** Linear arrays: (a) with no wraparound links; (b) with wraparound link.

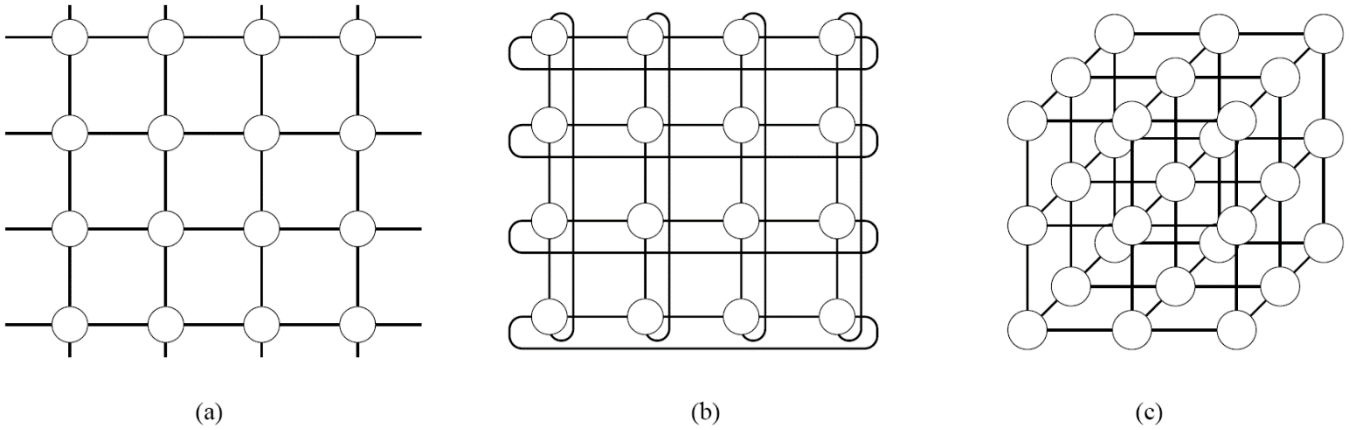



**Figure 2.16** Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

# Networks Topologies : Hypercubes and their construction

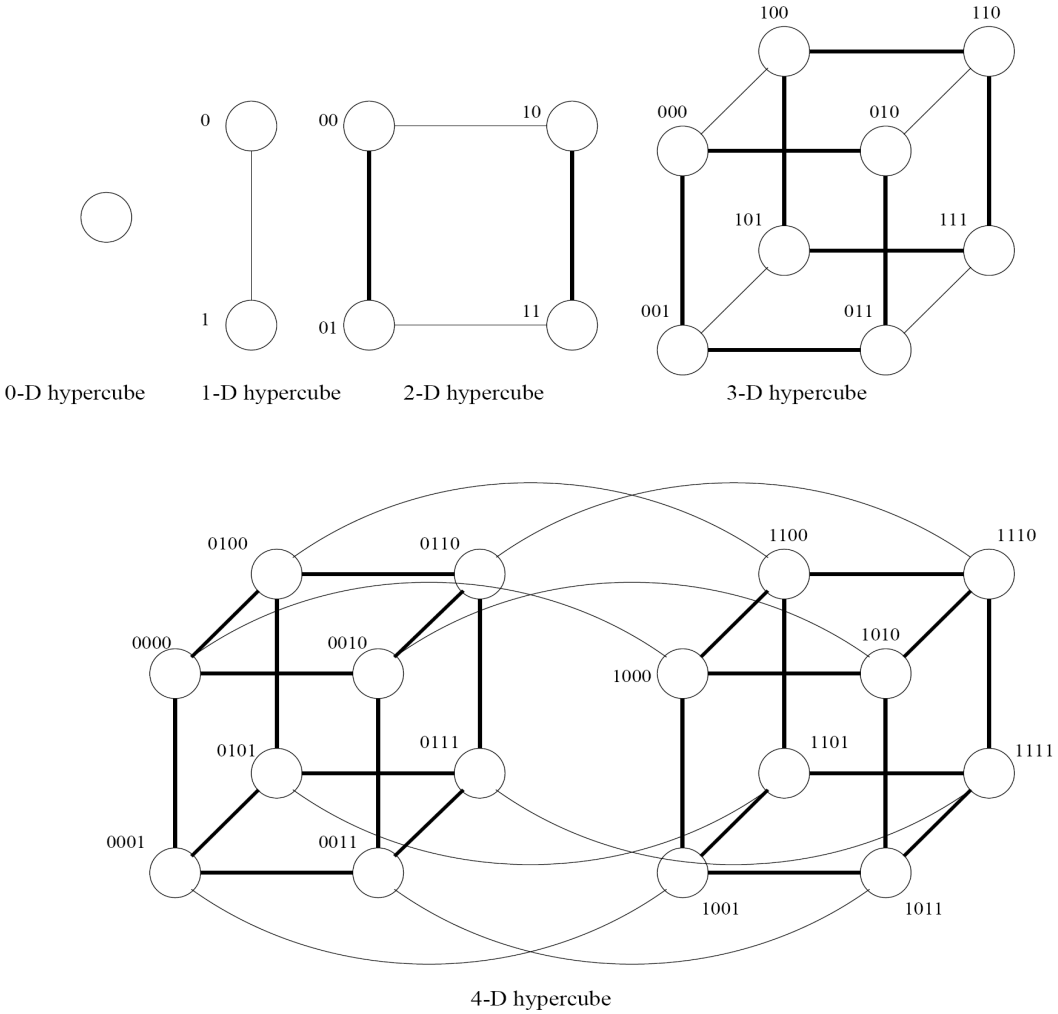



**Figure 2.17** Construction of hypercubes from hypercubes of lower dimension.

# Networks Topologies : Hypercubes properties

1. The **distance** between any two nodes is at most **log p**
2. Each node has exactly **log p** neighbors
3. The **distance** between two nodes is given by the **number** of **bit positions** in which the two nodes differ (e.g., 0110 and 0101 are distant 2 nodes)

# Networks Topologies: Tree-Based Networks

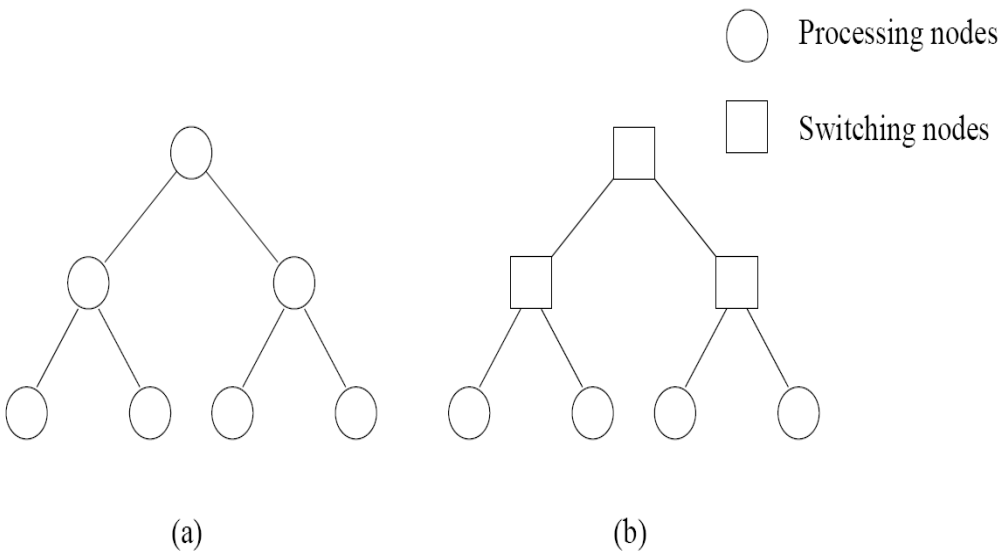
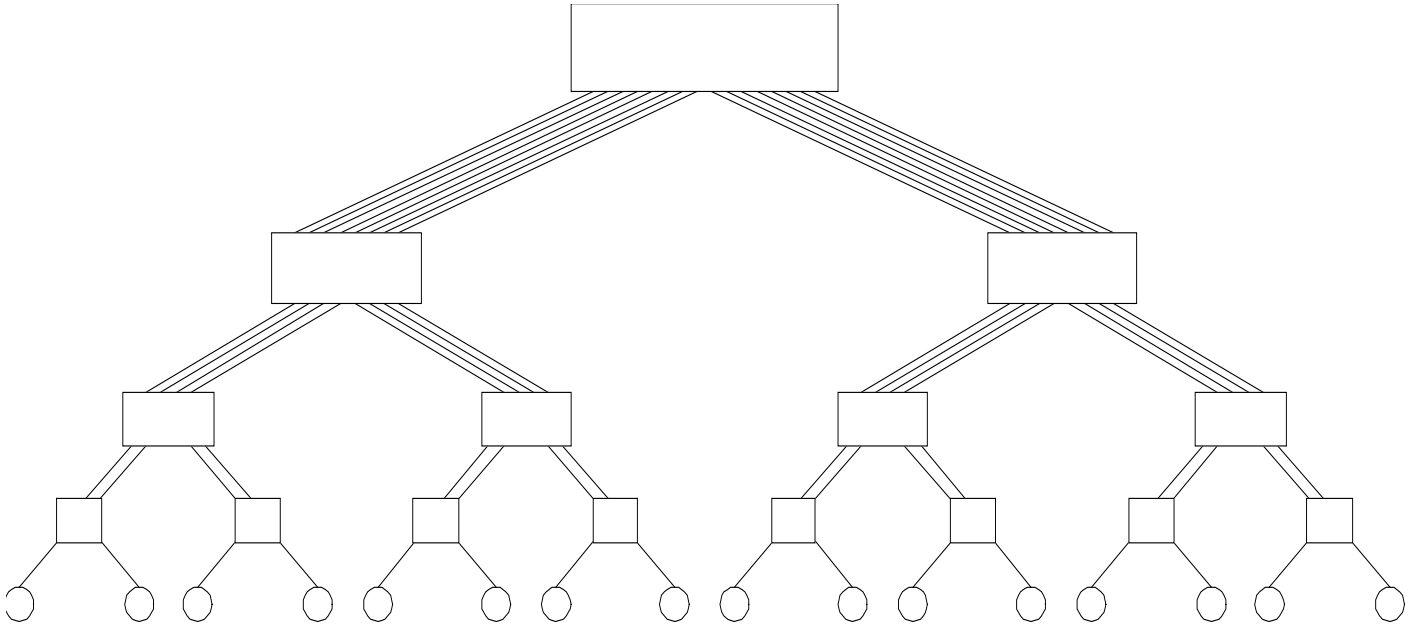



**Figure 2.18** Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

# Networks Topologies: Tree Properties

- The **distance** between any two nodes is **no more** than **2logp**.
- The **links** that are upward require more communications of those located in the lower part of the tree
- For this reason, a variant called **fat-tree**, "thickens" the links as we climb the tree
- The trees can be arranged in 2D with **no intersection**. *This is a very important property*

# Network Topologies: Fat Trees



A fat tree network of 16 processing nodes.

# Evaluation of Static Interconnection Networks

| Network | Diameter | Bisection Width | Arc Connectivity | Cost (No. of links) |
|---|---|---|---|---|
| Completely-connected | $1$ | $p^2/4$ | $p-1$ | $p(p-1)/2$ |
| Star | $2$ | $1$ | $1$ | $p-1$ |
| Complete binary tree | $2\log((p+1)/2)$ | $1$ | $1$ | $p-1$ |
| Linear array | $p-1$ | $1$ | $1$ | $p-1$ |
| 2-D mesh, no wraparound | $2(\sqrt{p}-1)$ | $\sqrt{p}$ | $2$ | $2(p-\sqrt{p})$ |
| 2-D wraparound mesh | $2\lfloor\sqrt{p}/2\rfloor$ | $2\sqrt{p}$ | $4$ | $2p$ |
| Hypercube | $\log p$ | $p/2$ | $\log p$ | $(p\log p)/2$ |
| Wraparound *k*-ary *d*-cube | $d\lfloor k/2\rfloor$ | $2k^{d-1}$ | $2d$ | $dp$ |

# Evaluation of Dynamic Interconnection Networks

| Network | Diameter | Bisection Width | Arc Connectivity | Cost (No. of links) |
|---|---|---|---|---|
| Crossbar | $1$ | $p$ | $1$ | $p^2$ |
| Omega Network | $\log p$ | $p/2$ | $2$ | $p/2$ |
| Dynamic Tree | $2\log p$ | $1$ | $2$ | $p-1$ |

# Communication costs

- Together with idling and resource contention, **communication** is the **main cause** of overhead in parallel programs (*the cause that does not allow a speed-up = p*)

- The **cost** of communication depends on **several factors**, including the semantics of the programming model, the network topology, data processing, and adopted routing software protocols

# Communication Costs for Message Passing

- The total time to transfer a message over the network comprises:

  - **Startup time ($t_s$):** *Time spent at the sender and receiver nodes* (execution of the algorithm, routers, etc.)
  - **Per-hop time ($t_h$):** *Time taken by the header of the message to reach the next node*. This time is a function of the number of **hops** (next nodes) and includes factors such as the latencies of switches, network delays, etc.
  - **Per-word transfer time ($t_w$):** *Given by 1 / r, where r is the bandwidth (words / s).* This time includes all the overheads that are determined by the length of the message. This includes the bandwidth of the links, error checking and correction, etc.

# Store-and-Forward Routing

- A message that traverses multiple hops is completely received in an intermediate hop before being forwarded to the next hop

- The total cost of communication for a message of size *m* to cross *l* communication links is

$$t_{comm} = t_s + (mt_w + t_h)l.$$

- In most platforms, $t_h$ is small and the expression can be approximated by

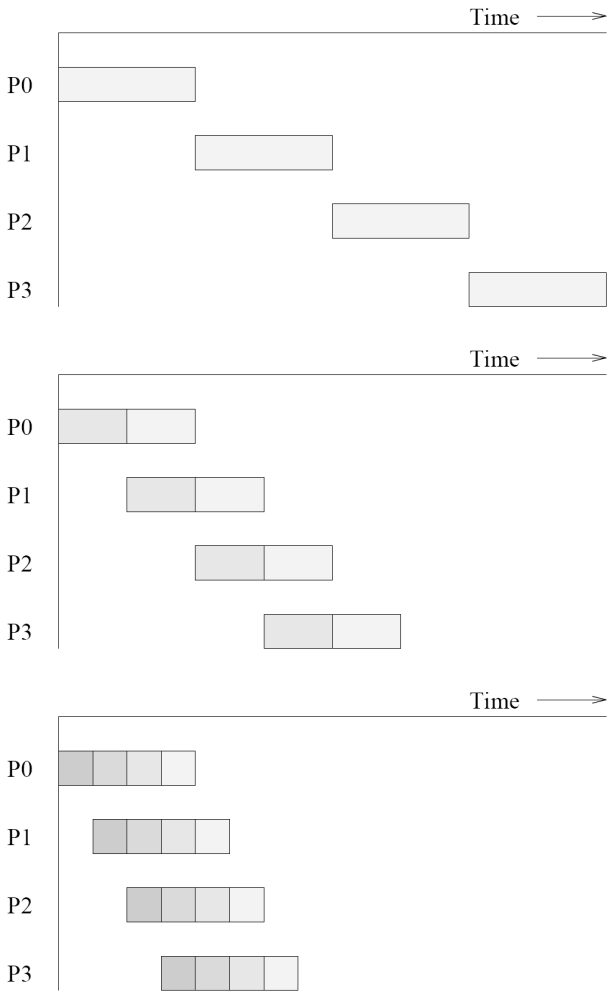
$$t_{comm} = t_s + mlt_w.$$

# Packet Routing

- The **store-and-forward** technique makes little use of communication resources
- The **Packet Routing** breaks messages into packets and forwards them, **pipeline-type** on the network (e.g.: Internet)
- Since different packets may take different routes (like for the TCP/IP protocol), each packet must contain a **header** with information on **routing**, **error checking**, **sequencing**, and other information
- The total time of communication for packet routing is approximated by $t_{comm} = t_s + t_h l + t_w m.$

where the factor $t_w$ **also** takes into consideration the overheads of the headers of each packet (that is different from that of the former)

# Routing Tecniques




(a) A single message sent over a store-and-forward network

(b) The same message broken into two parts and sent over the network.

(c) The same message broken into four parts and sent over the network.

**Figure 2.26** Passing a message from node $P_0$ to $P_3$ (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.

# Cut-Through Routing

- It takes the concept of packet routing in an "extreme“ manner, by further **dividing** messages into basic units called **flits (4-32 bytes)**
- Each flit is **forced** to take the same path, in sequence (to save routing information)
- Since the flits are typically small, the **header** of the message is **minimized**
- A **tracer message** first "programs" all the intermediate routers. Subsequently, the flits take the same path

# Cut-Through Routing

- The total time of communication to the **cut-through** is approximated by:

$$t_{comm} = t_s + t_h l + t_w m.$$

***l*** = hops; ***m*** = message length

- This is identical to the packet routing, although $t_w$ is typically smaller
- Much better than store-and-forward, where ***l*** and ***m*** were both **multiplied**

# Simplified Cost Model for Communication Messages

- The **cost** of communicating a message between two remote nodes (hops) using the **cut-through** routing is given by

$$t_{comm} = t_s + lt_h + t_w m.$$

- In this expression, $t_h$ is typically **smaller** than $t_s$ and $t_w$. For this reason, the second term of the formula $lt_h$ may be omitted, when **m** is large
- Moreover, it is often **impossible** to control the routing (i.e., the actual calculation of ***l***) and the allocation of tasks (e.g. the user has little control over the mechanisms of communication in MPI)
- So, **in conclusion and in general**, one can approximate the cost of a transfer of the message by:

$$t_{comm} = t_s + t_w m.$$

# Reviewing…

$$t_{comm} = t_s + lt_h + t_w m.$$

implies that:

1. It 'better to **aggregate** messages and not send many small (to avoid every time $t_s$)
2. **Reduce** the size of the message (to minimize $\boldsymbol{t_w}$)
3. **Reduce** the distance between hops (to decrease ***l***)

but .... 1 and 2 can be easily handled, but not 3 !

That's why we approximate all by:

$$t_{comm} = t_s + t_w m.$$

# Cost Models for Shared Address Space Machines

- While the basic mechanisms for the costs are valid for this kind of machines, a number of other factors may make **difficult** an accurate estimate:

- **The memory layout is typically determined by the system**
- **Limited** cache size can result in a **cache thrashing** (i.e. requested data is not present in cache)
- The associated **overheads** with the invalidate and update operations can be difficult to quantify
- **Spatial locality** is difficult to model
- **False sharing** and **contention** are difficult to model

# Routing Mechanisms

- **Routing**
  - Algorithm that is used to determine the route that a message will take form a source node to a destination one
- **Minimum**
  - Selects always shorter route (but can produce congestion
- **Not minimum**
  - Takes longer routes to avoid congestion
- **Deterministic**
  - Determines a unique route
- **Adaptive**
  - Uses information regarding the status of the network

# Routing mechanisms for Communication Networks

- *How do you calculate the physical path of a message from the source processor to the destination one?*

  - Routing must avoid deadlocks - for this reason, we use the **dimension-ordered** (for meshes) or **E-cube routing** (for hypercubes)
  - The routing should avoid **hot-spots**. For this reason, the **two-step routing** is often used. In this case, a message from the **source s** to the **recipient d** is first sent to an intermediate **node i** and then randomly "forwarded" to **destination d**

# Routing mechanisms for Communication Networks

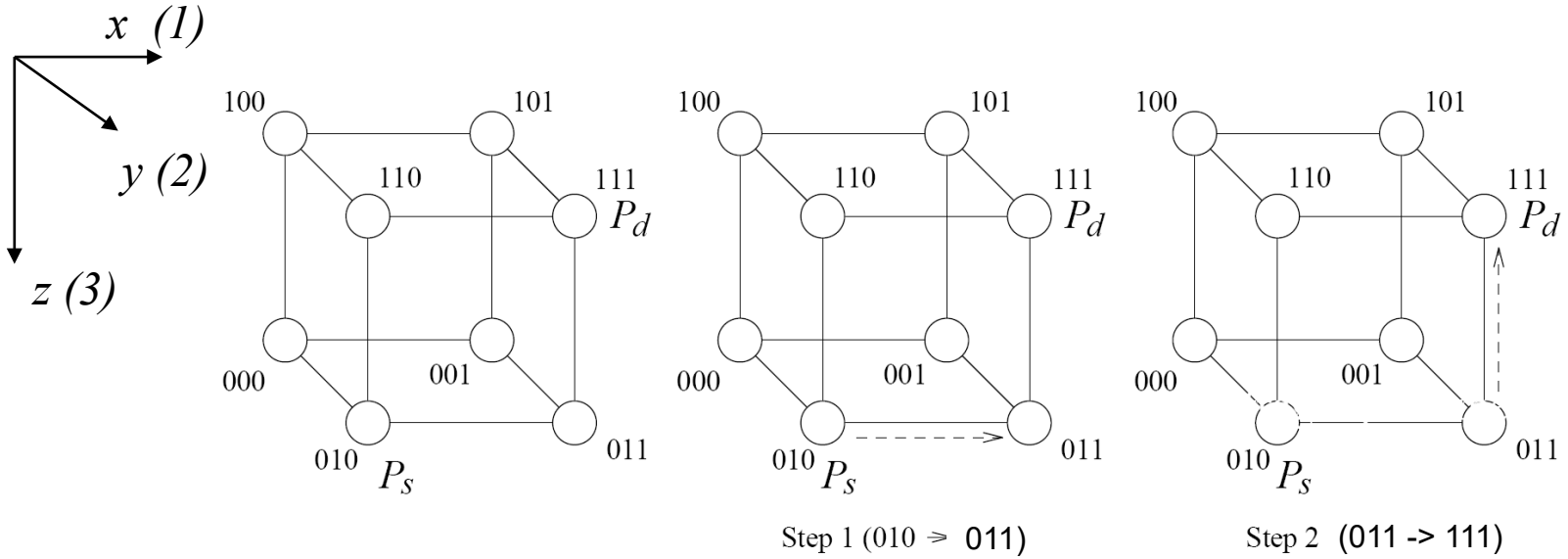


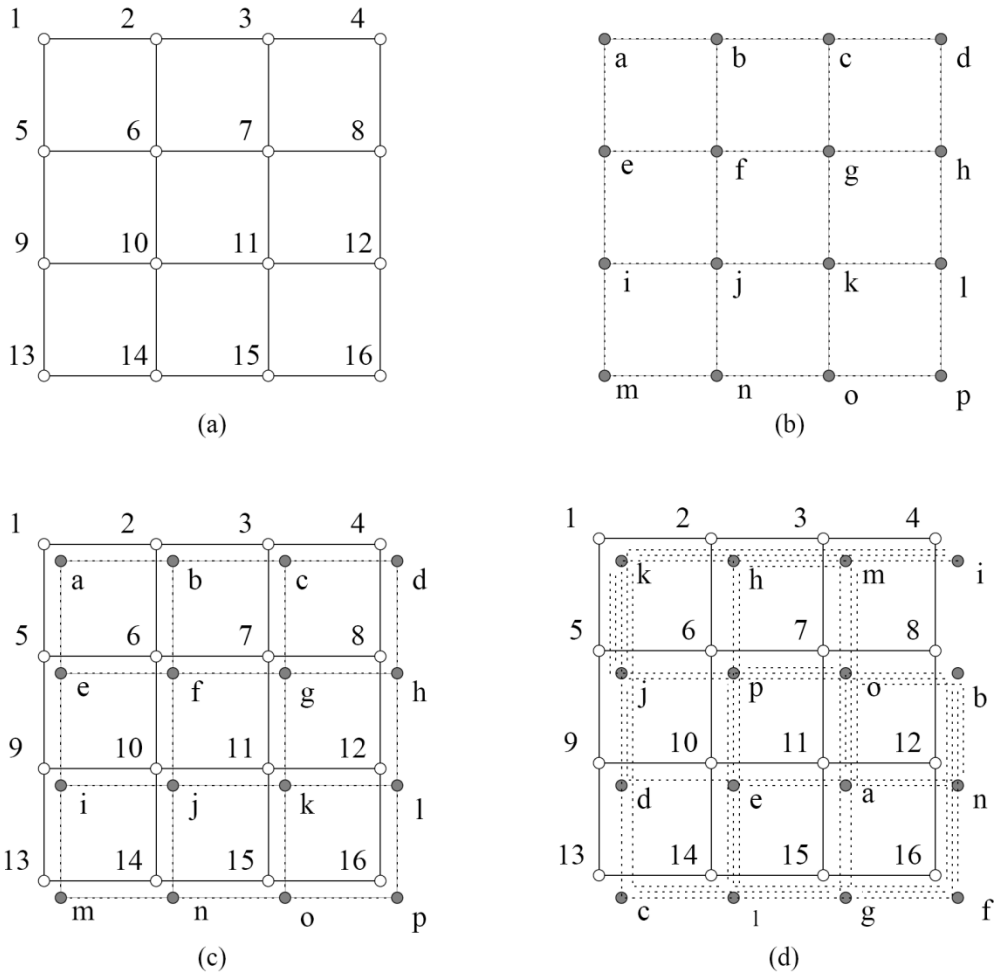

**Figure 2.28** Routing a message from node $P_s$ (010) to node $P_d$ (111) in a three-dimensional hypercube using E-cube routing.

**E-cube routing: It makes XOR representations of Ps and Pd, and sends the message along the direction k of the least significant bit that is different from zero in the XOR operation.**
**The same is done for the intermediate nodes (considering Pi with Pd)**

# Mapping Techniques for Graphs

- MPI (but also other solutions) does not allow to have control over **how processes are mapped onto processors**
- Often, we need to **map** a communication pattern on a interconnection topology
- For example, we have a certain algorithm designed for a certain topology, and we are implementing it on another
- For this purpose, it is helpful to **understand** the mapping between different graphs

# Example




- (a) Mapping of real processors
- (b) Process Mapping
- (c) “Intuitive" Mapping
- (d) Random Mapping: the communications between processors increase up to 6 times!

**Figure 2.29** Impact of process mapping on performance: (a) underlying architecture; (b) processes and their interactions; (c) an intuitive mapping of processes to nodes; and (d) a random mapping of processes to nodes.

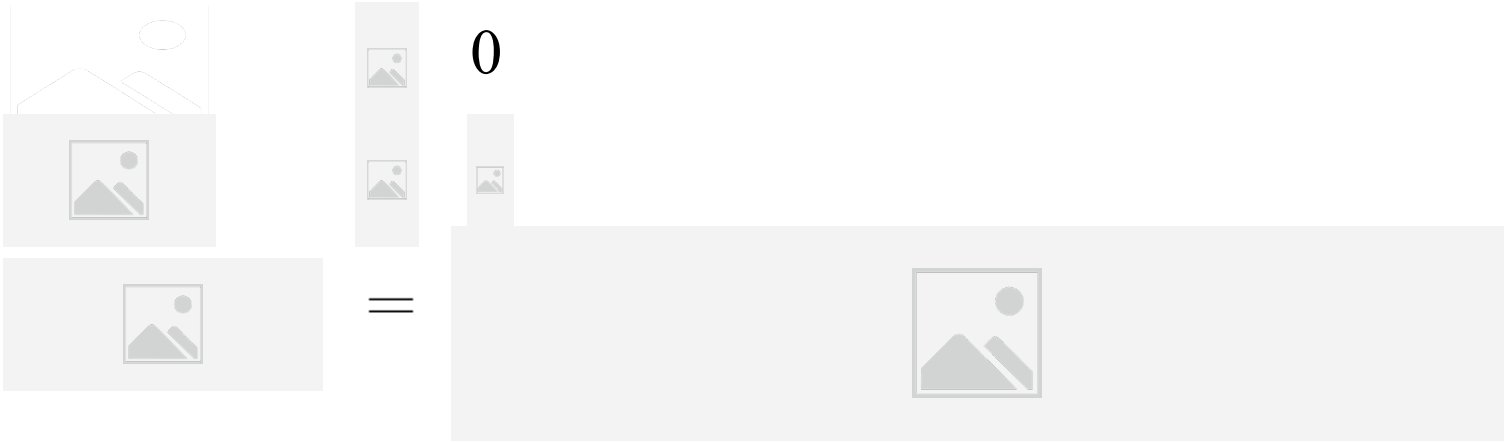
# Mapping Techniques for Graphs: Metrics

When you **map** a graph G(V, E) on another graph G'(V', E'), the following metrics are important:

- The *maximum number* of arcs mapped to any arc of E' is called **congestion** of the mapping
- The *maximum number* of arcs of E' that any side of E is mapped is called **dilation** mapping.
- The *ratio* of the number of nodes in V' and the set V is called the **expansion** of the mapping

# Mapping of a Linear Array on a Hypercube

- A linear array (or ring) consists of $2^d$ nodes (labeled 0 to $2^d$ - 1) can be mapped to a d-dimensional hypercube by mapping a node i of the node G (i, d) of the hypercube using the function G (i, x) defined as follows:

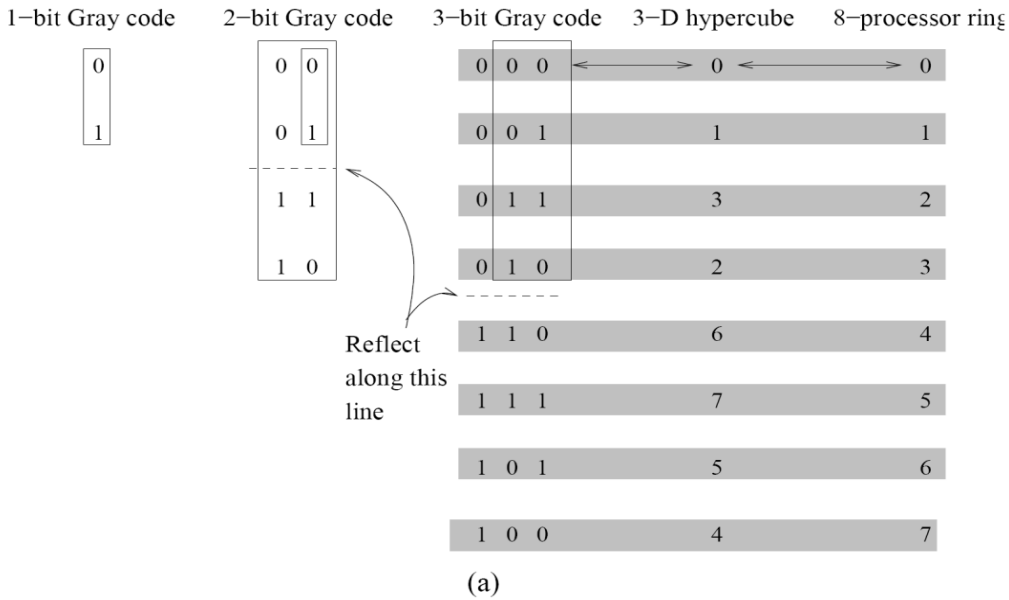


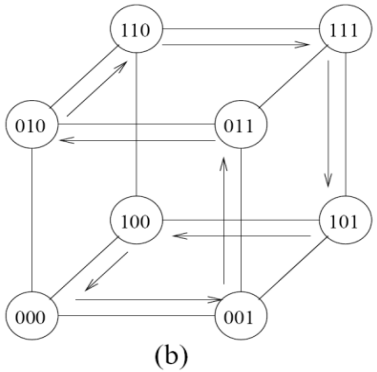
# Mapping of a Linear Array on a Hypercube

The function G is called the *Binary Reflected Gray* code (RGC)

With this encoding, the adjacent nodes (G(i, d) and G (i + 1, d)) differ by **only one** bit position, so the corresponding processors are mapped to neighboring nodes in the hypercube. Therefore, congestion, dilation and expansion are 1

# Mapping of a Linear Array on a Hypercube: Example



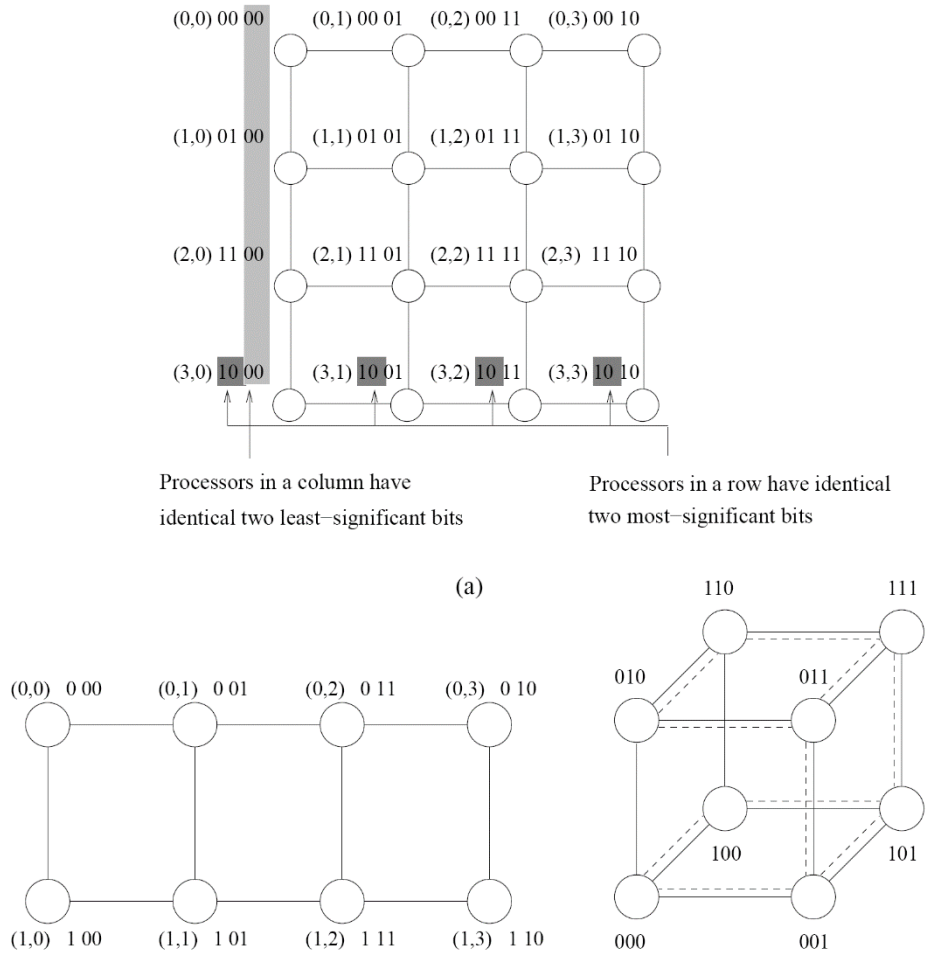

(a) A ring based on the 3-bit Gray code; and (b) its mapping on a 3-D hypercube

# Mapping of a Mesh on a Hypercube

- A $2^r \times 2^s$ toroidal mesh can be mapped onto a hypercube $2^{r+s}$ nodes mapping the node *(i, j)* of the mesh node G (i, r - 1) || G (j, s - 1) of the hypercube

(where the operator || denotes the concatenation of two codes Gray)

# Mapping of a Mesh on a Hypercube



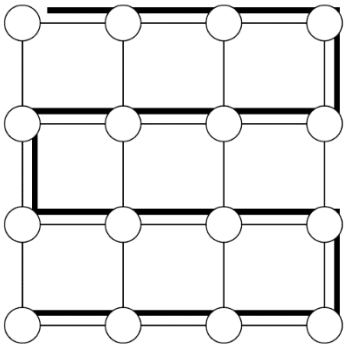

(a) A 4 × 4 mesh mapped onto a hypercube in four dimensions; and (b) A 2 × 4 mesh mapped on a three-dimensional hypercube

Even in this case, congestion, dilation and expansion are 1

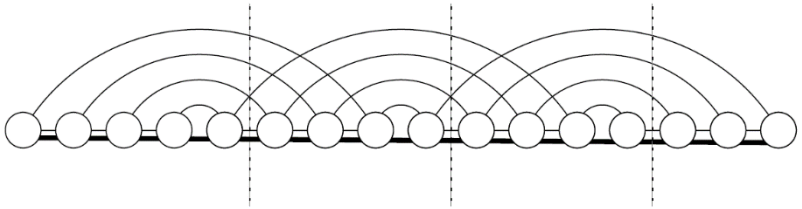
# Mapping a Mesh on a 1D Array

- Given that a mesh has more sides of a 1D array, we will **not** have a mapping with optimal congestion/dilation
- Let's analyze first the mapping of a linear array on a mesh and subsequently reverse the mapping
- In terms of congestion, this mapping is, however, **optimal**

# Mapping a Mesh on a 1D Array: Example



(a) Mapping a linear array into a 2D mesh (congestion 1).



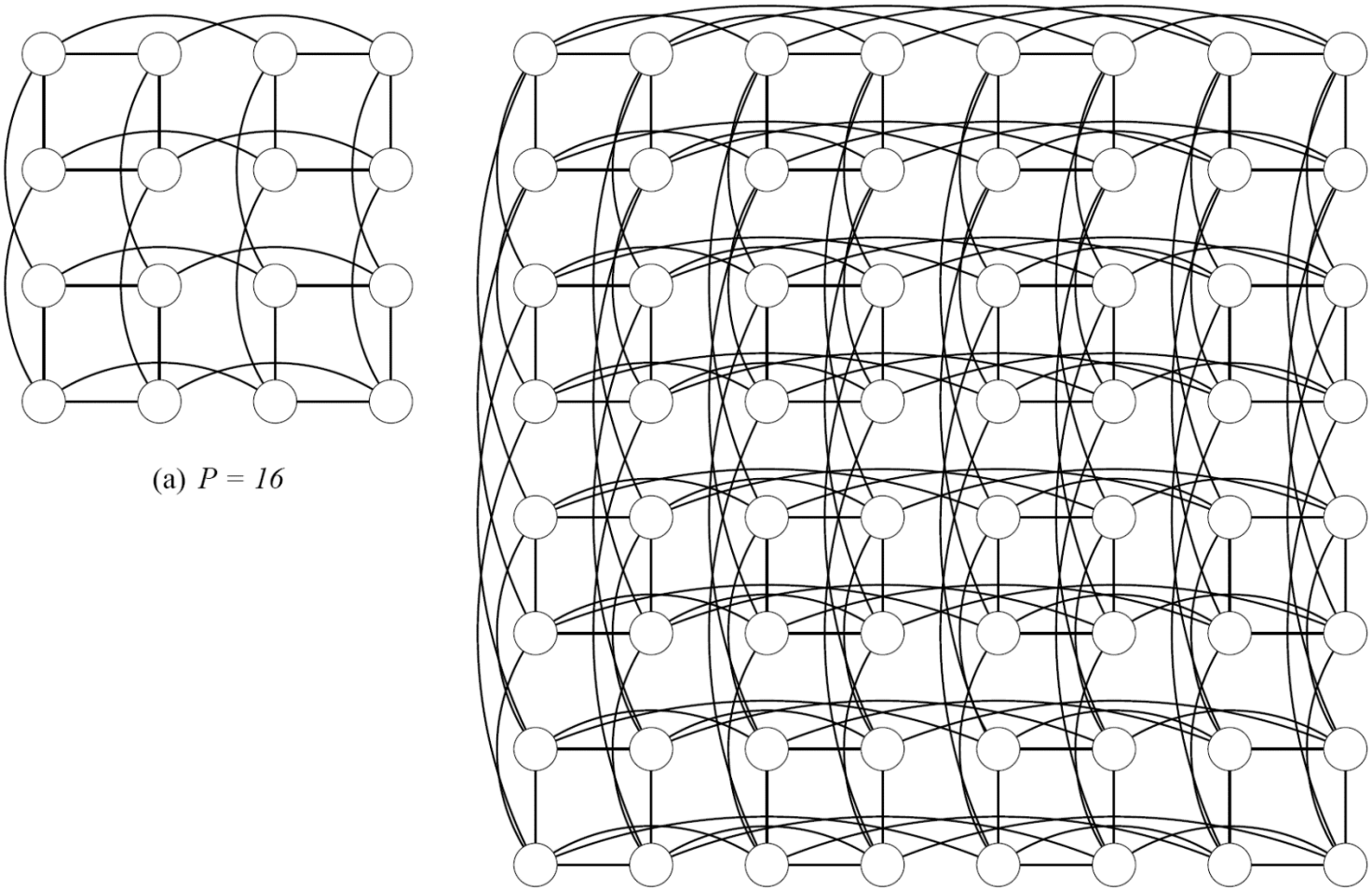
(b) Inverting the mapping − mapping a 2D mesh into a linear array (congestion 5)

(a) Mapping of a linear array to 16 nodes on a 2-D mesh; and (b) reverse mapping. **Bold lines** correspond to linear arcs in the array - the normal lines to arcs of the mesh.

# Mapping a Hypercube on a 2-D mesh

- Each sub-cube of $\sqrt{p}$ nodes of the hypercube is mapped on a row of $\sqrt{p}$ nodes of the mesh
- This is done by inverting the mapping of linear array on hypercube
- It can be shown that it is optimal!

# Mapping a Hypercube on a 2-D mesh : Example



(a) $P = 16$

(b) $P = 32$

Mapping of a hypercube on a 2-D mesh