Parallel Architectures

CPS343

Parallel and High Performance Computing

Spring 2020

Outline

Outline

Flynn's Taxonomy (1966)

	Single Data	**Multiple Data**
Single Instruction	**SISD** uniprocessors	**SIMD** processor arrays pipelined vector processors
Multiple Instruction	**MISD** systolic arrays	**MIMD** multiprocessors multicomputers

Outline

Other taxonomies

- **SPMD** — Single Program, Multiple Data. This is similar to SIMD but indicates that a single program is used for the parallel application i.e. every node runs the same program.
- **MPMD** — Multiple Program, Multiple Data. Like MIMD except this explicitly uses more than one program for the parallel application. Typically one is a *master* or *control* program and the others are *slave* or *compute* programs.
- Modern clusters typically follow one of these two models. It is often convenient to use the SPMD model but have the program behave as either a *master* or *slave* based on some criteria determined at run-time (often the node on which the program is running).

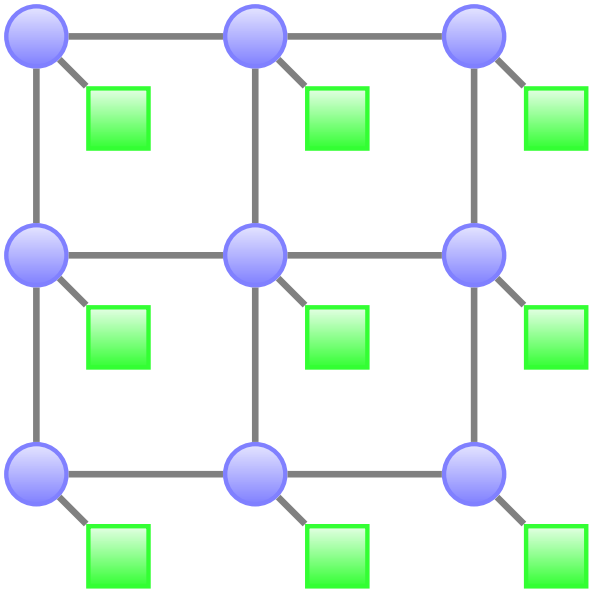
Outline

Definitions

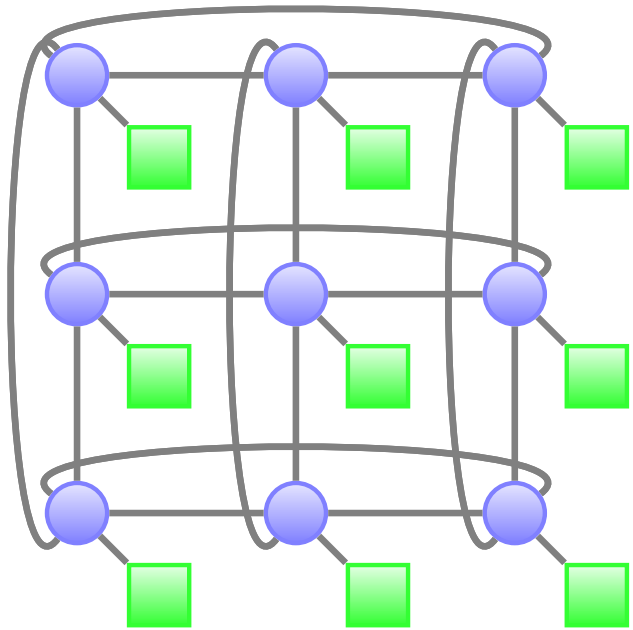
- **Diameter** – largest distance between two switch nodes. Want this to be as small as possible.
- **Bisection width** – smallest number of connections that need to be severed to separate network into two distinct subnetworks of the same size. Want this to be as large as possible.
- **Edges per switch node** – best if this is constant and independent of network size.
- **Edge length** – physical length of wire or other media required for connections. Ideally this is constant (does not grow with network size).
- **Direct Topology** – every switch is connected to a node and other switches.
- **Indirect Topology** – some switches are only connected to other switches.

2-D Meshes with n processors

2-D Mesh



2-D Mesh with wrap-around (torus)



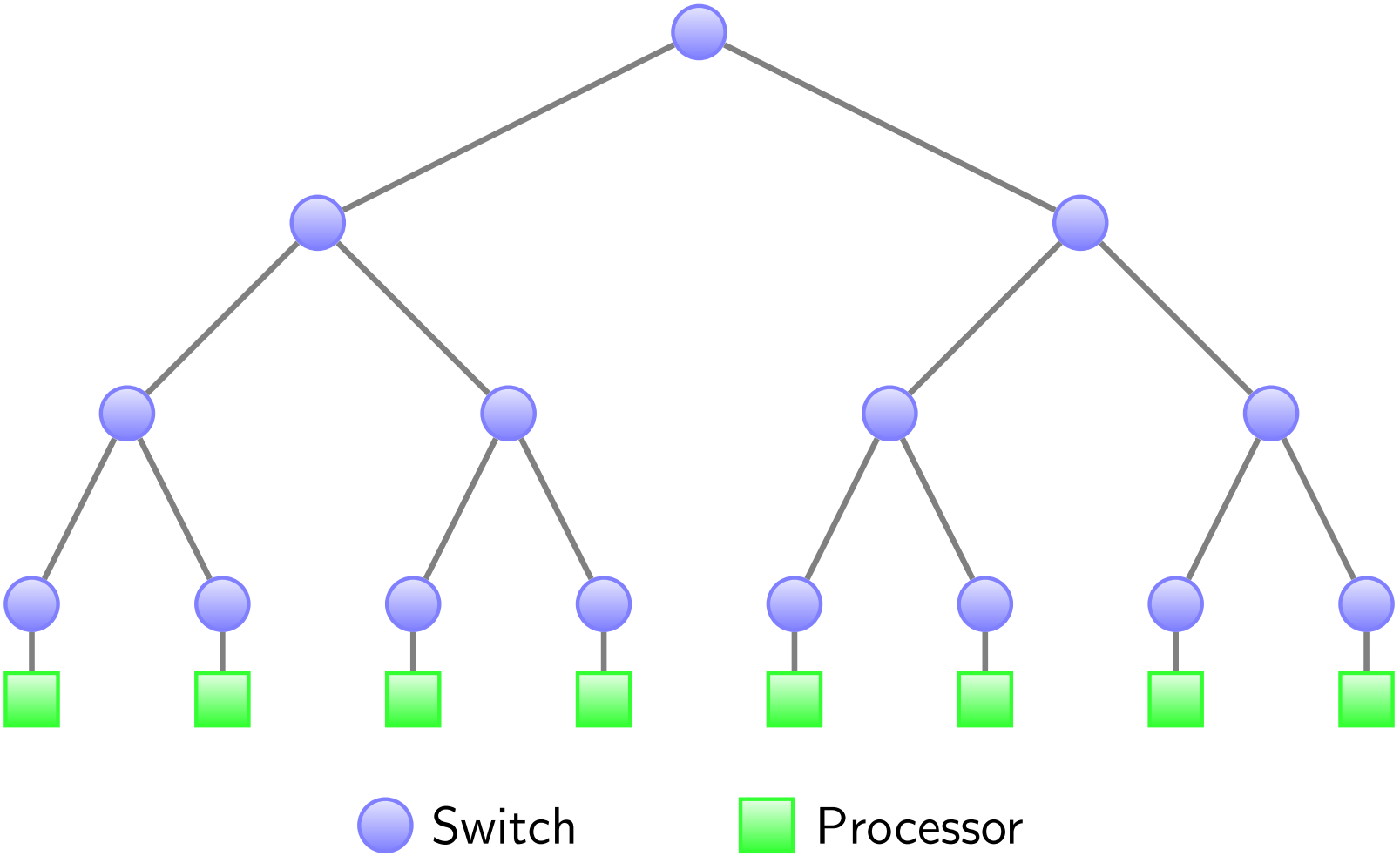
Switch

Processor

2-D Meshes with n processors

- Direct topology
- 2-D mesh with wrap-around connections is a torus.
- Diameter on non-wrap-around mesh is minimized when mesh is square: $2(\sqrt{n} - 1)$. For a square mesh with wrap-around the diameter is $\sqrt{n}$.
- Bisection width is also large—without wrap-around the bisection with of a square mesh is $\sqrt{n}$.
- Constant number of edges per switch (4 for the wrap-around version) and can have constant edge length.

Binary tree



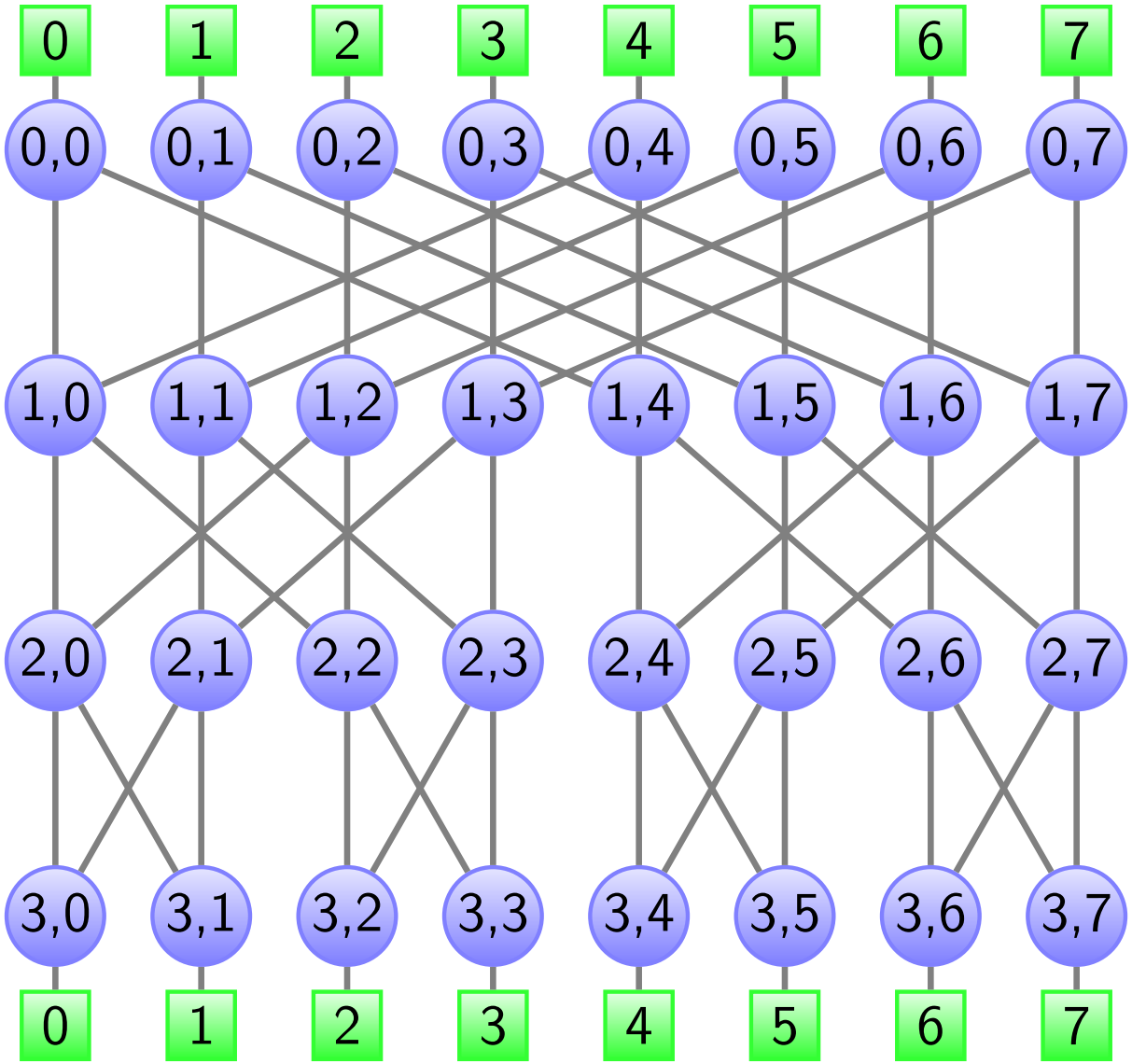
Binary tree

- Indirect topology.
- Interior nodes are switches only connected to other switches; leaf nodes are also connected to processors.
- $n = 2^d$ for some d and there are $2n - 1$ switches.
- Diameter is $2 \log n = 2d$.
- Bisection width is 1 — this is not good as all traffic from one side of tree must travel though a single switch on its way to the other side.
- Number of edges per switch is fixed at 3 for switches only connected to other switches and 2 switches connected to processors.
- Edge length grows as nodes are added.

Hypertree network of degree k and depth d

- Indirect topology
- Keeps the diameter small (good) while making bisection width larger (also good).

Butterfly network



Butterfly network

- Indirect topology
- $n = 2^d$ for some d and there are $n(\log n + 1) = n(d + 1)$ switches arranged in $d + 1$ rows (called ranks) of length n.
- Often the first and last rank of switches is physically the same but for the purposes of this discussion are counted separately.
- Let i, $0 \leq i < d$, be a switch's rank and let j, $0 \leq j \leq n$, be the switch's position within the rank. Switch(i,j) is connected to

  $$\text{switch}((i+1), j) \quad \text{and} \quad \text{switch}((i+1), \text{inv}(i,j))$$

  where inv(i,j) means that the i^{th} most-significant bit (counting from the left) in j is flipped.
- Diameter is $\log n = d$ and bisection width is n. Number of edges per switch is 4 but edge length does grow as depth of network increases.

Butterfly network

For example suppose $d = 3$ so $n = 2^3 = 8$ and consider switch(1,4):

Butterfly network

For example suppose $d = 3$ so $n = 2^3 = 8$ and consider switch(1,4):

- $(i + 1) = (1 + 1) = 2$. Thus
  switch(1,4) is connected to switch(2,4).
- The 0^{th} most-significant bit (counting from left) in $j = 4 = 100_2$ is 1.
- The 1^{st} most significant bit is 0; flipping this gives $110_2 = 6$ so inv(1,4)=6. Thus
  switch(1,4) is connected to switch(2,6).
- What switches is switch(2,5) connected to?
- $(i + 1) = (2 + 1) = 3$ and flipping the 3^{rd} most-significant bit in $5 = 101_2$ gives $100_2 = 4$ so **switch(2,5) is connected to switch(3,5) and switch(3,4)**.

Butterfly network

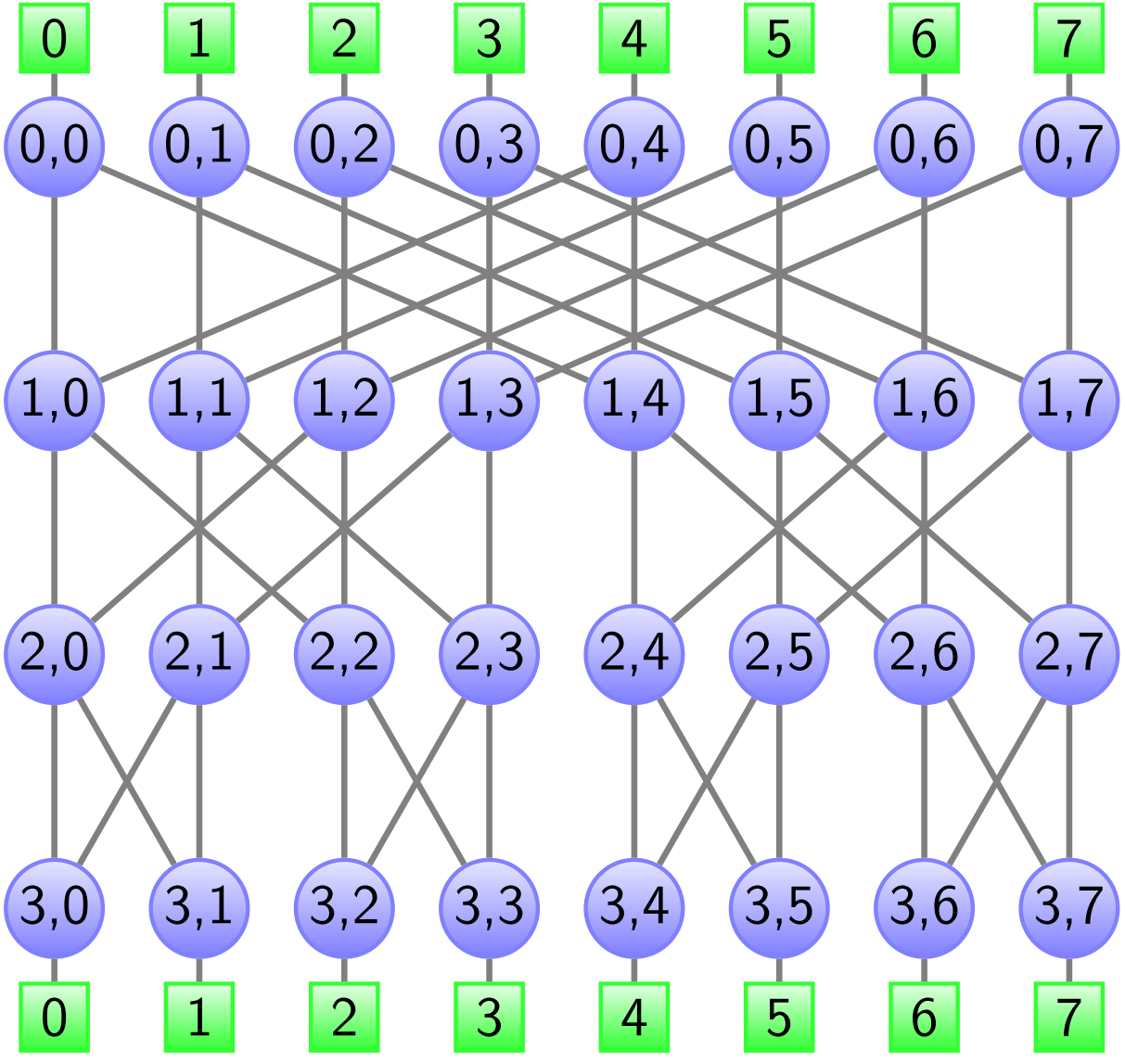
Routing turns out to be relatively easy.

Butterfly network

Routing turns out to be relatively easy.

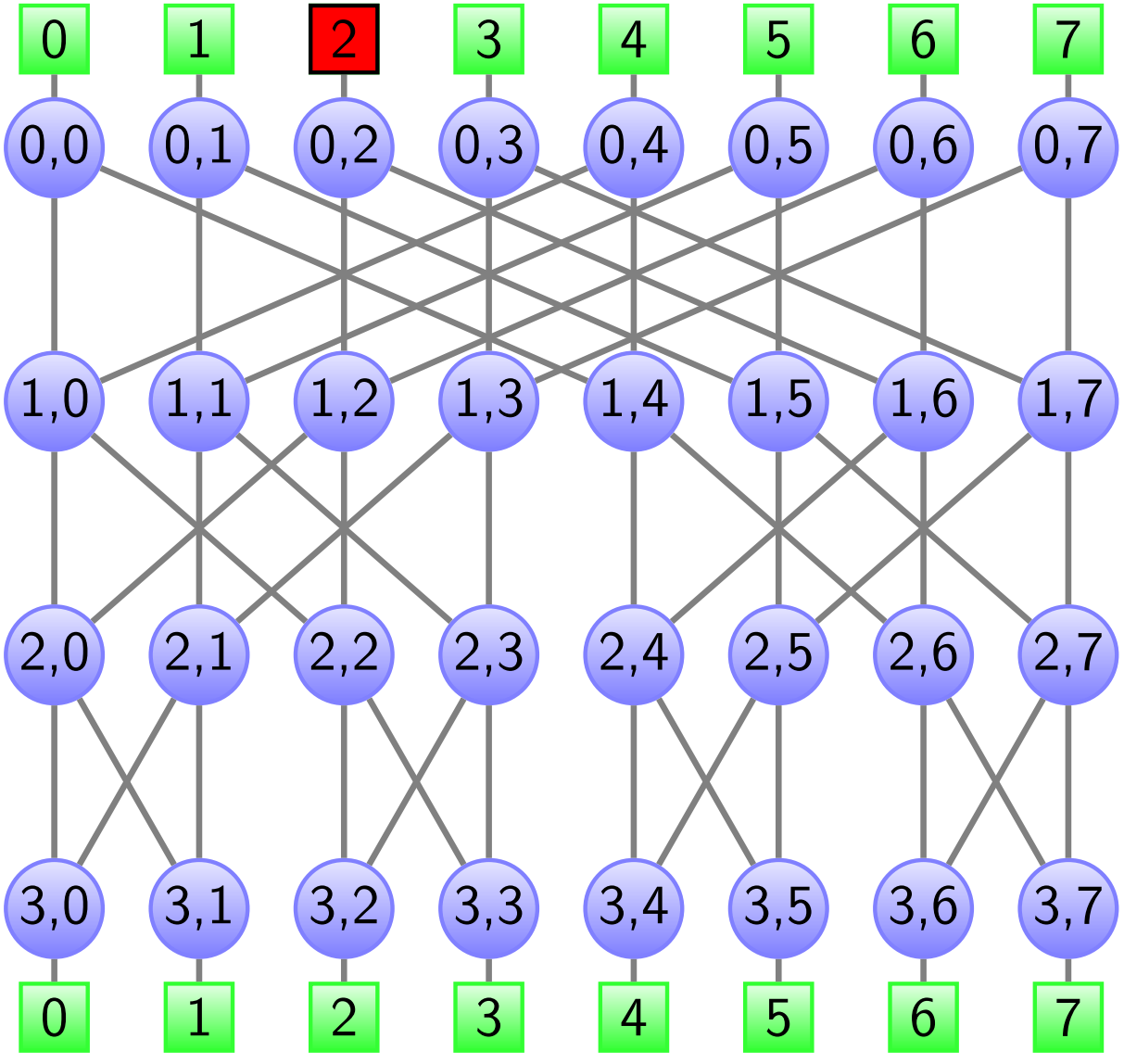
1. Destination address is appended to message.
2. Each switch that processes message "pops" most-significant bit from destination address and routes message left if the bit is 0, else routes message right.
3. Remaining address bits are sent with message.

Butterfly network



Send message:
Node 2 to Node 5

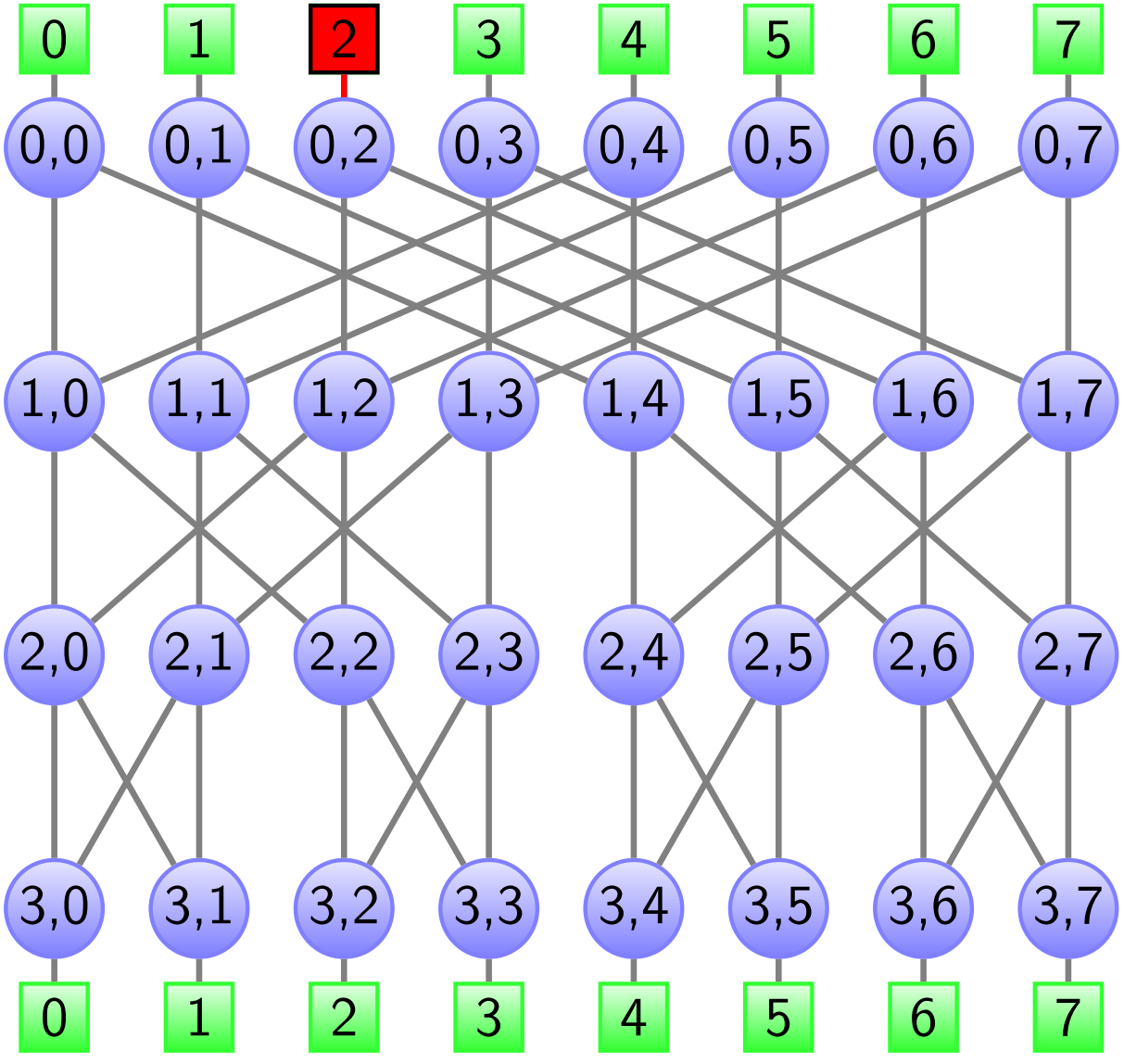
Butterfly network



Send message:
Node 2 to Node 5

Node 2 initiates a message to node 5.

Butterfly network



Send message:
Node 2 to Node 5

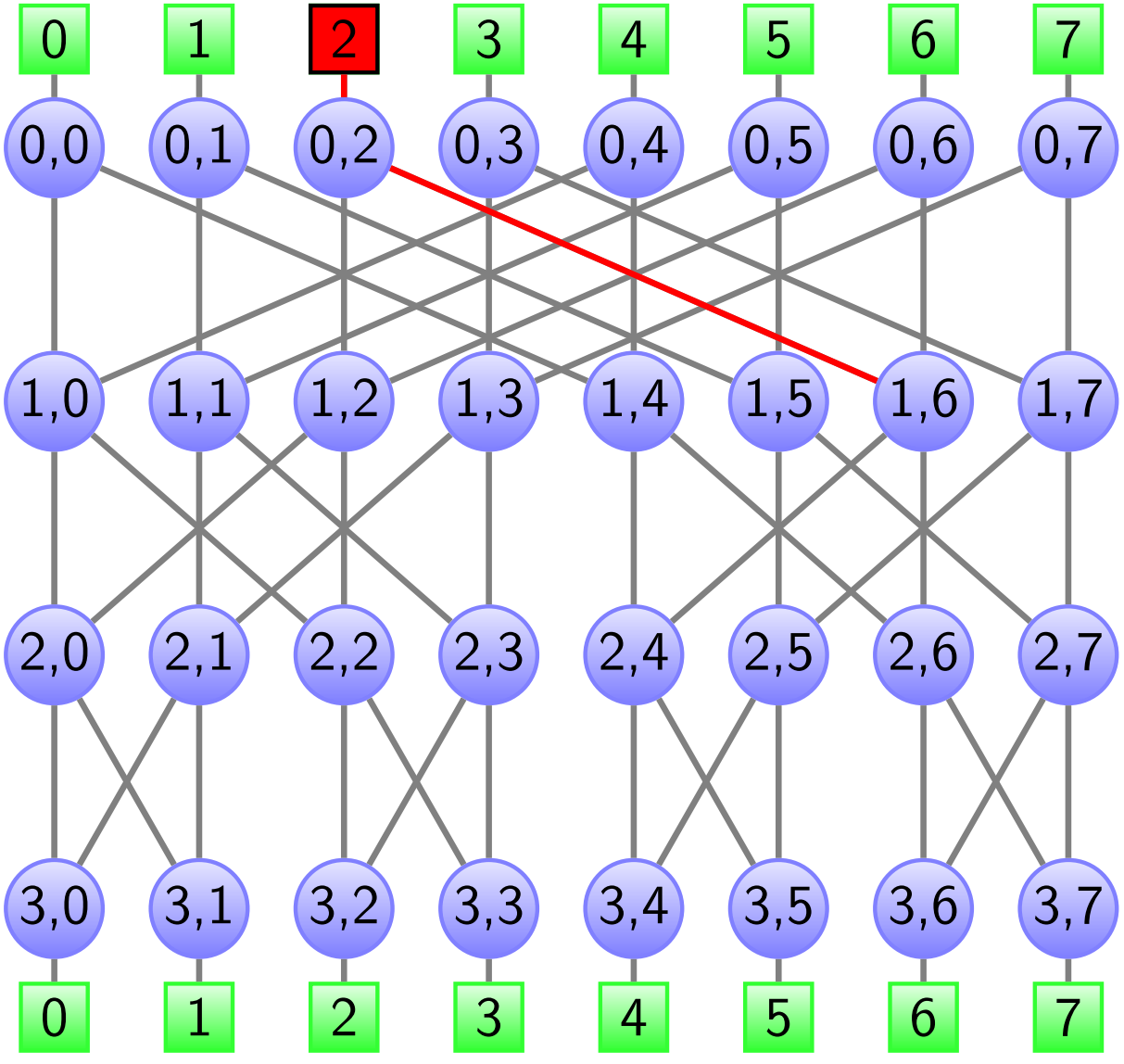
Dest: 101

Switch 0,2 "pops" leftmost bit from destination address.

Bit is 1 so message is routed down right path.

Remaining destination address is 01.

Butterfly network



Send message:
Node 2 to Node 5

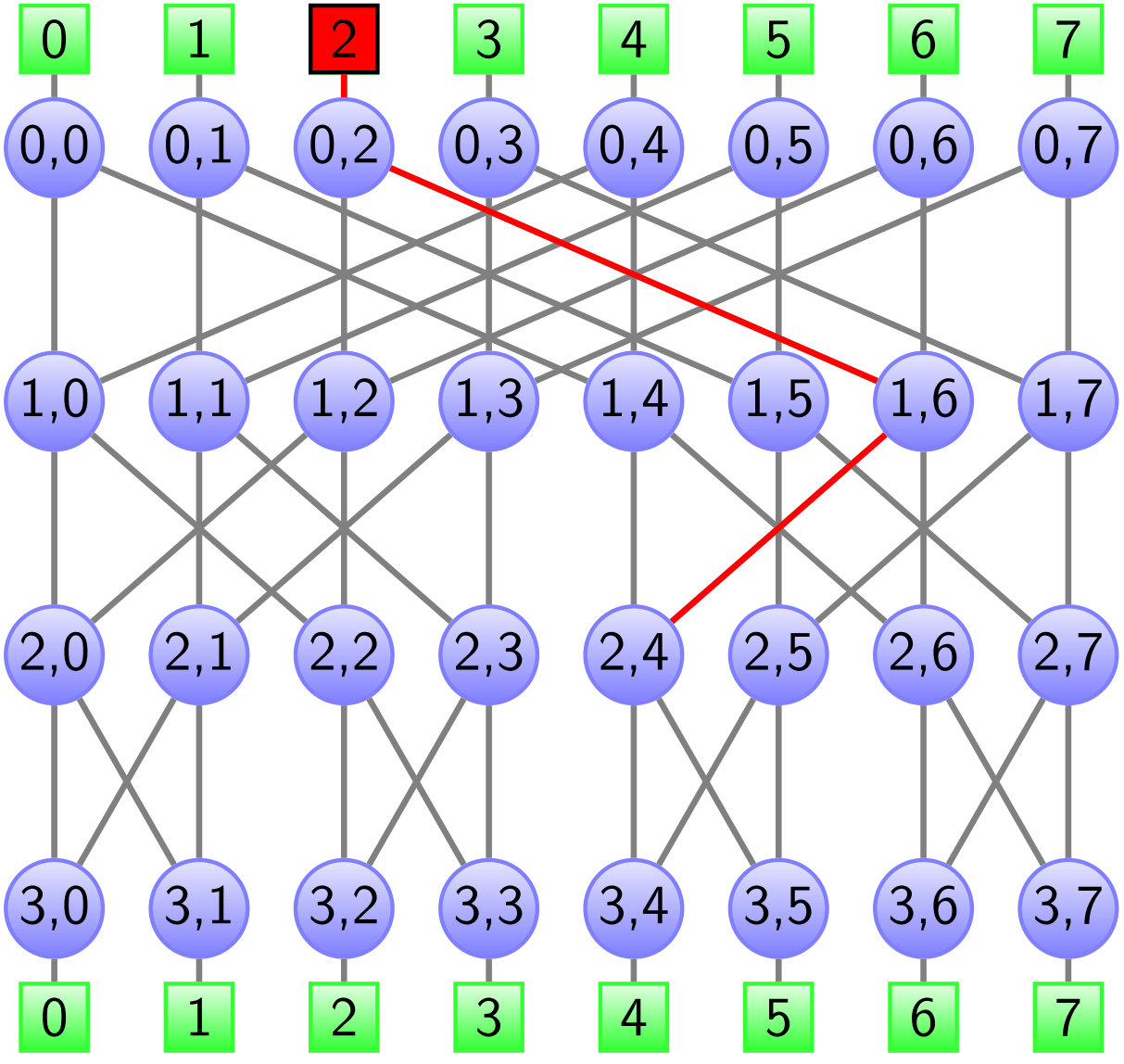
Dest: 01

Switch 1,6 "pops" leftmost bit from destination address.

Bit is 0 so message is routed down left path.

Remaining destination address is 1.

Butterfly network



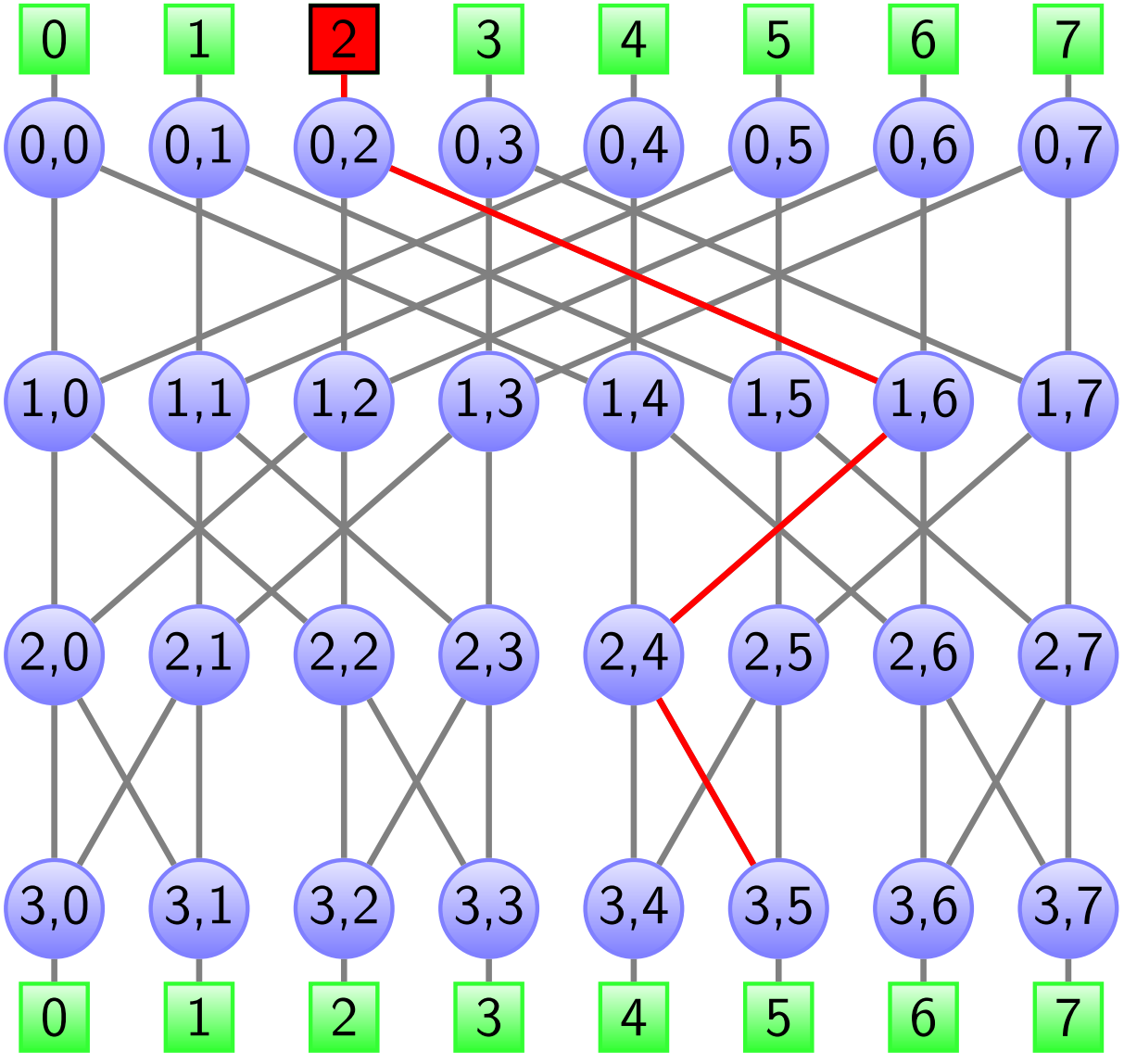
Send message:
Node 2 to Node 5

Dest: 1

Switch 2,4 "pops" leftmost bit from destination address.

Bit is 1 so message is routed down right path.

Butterfly network

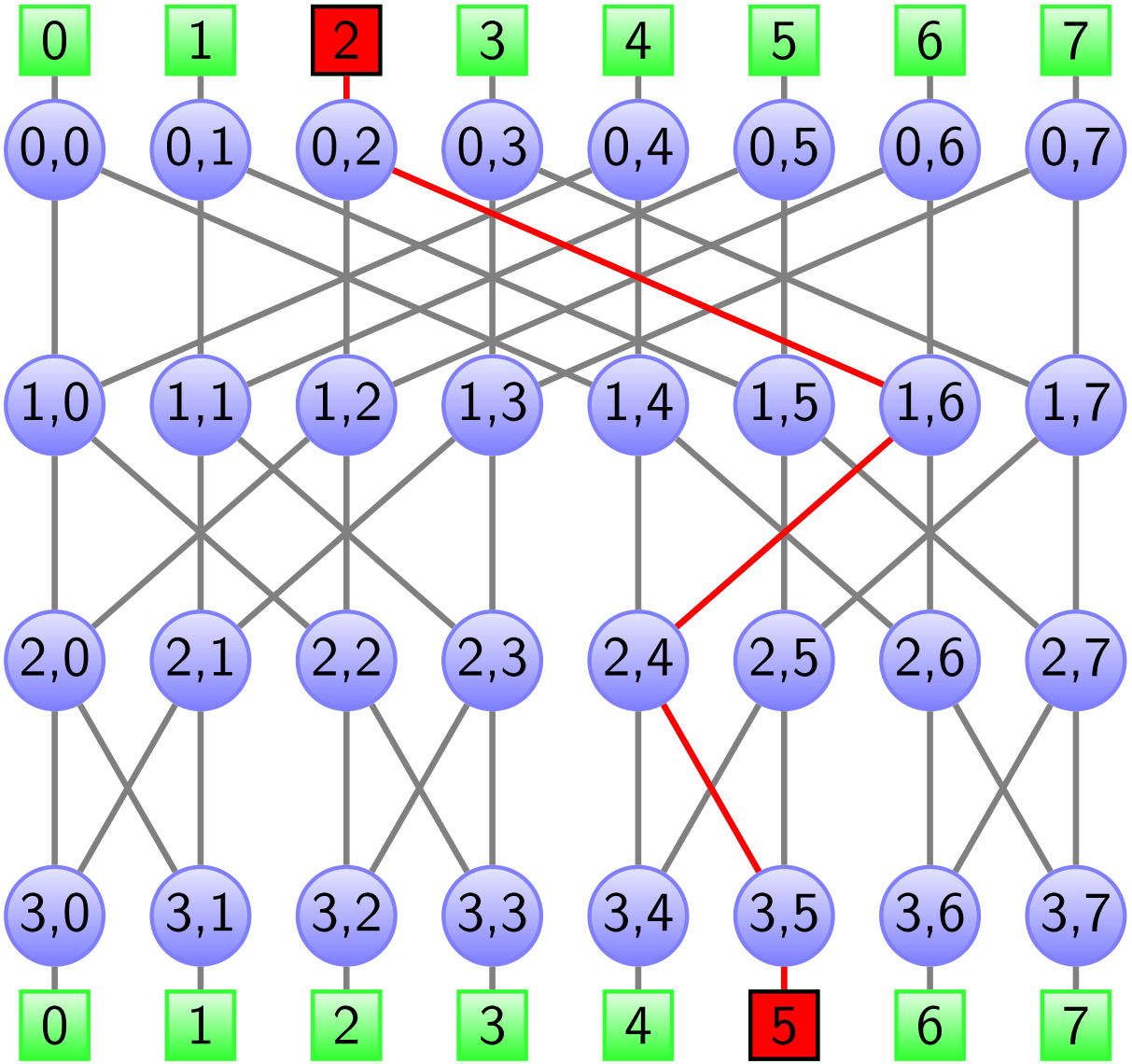


Send message:
Node 2 to Node 5

Dest:

Switch 3,5 makes message available to node 5.

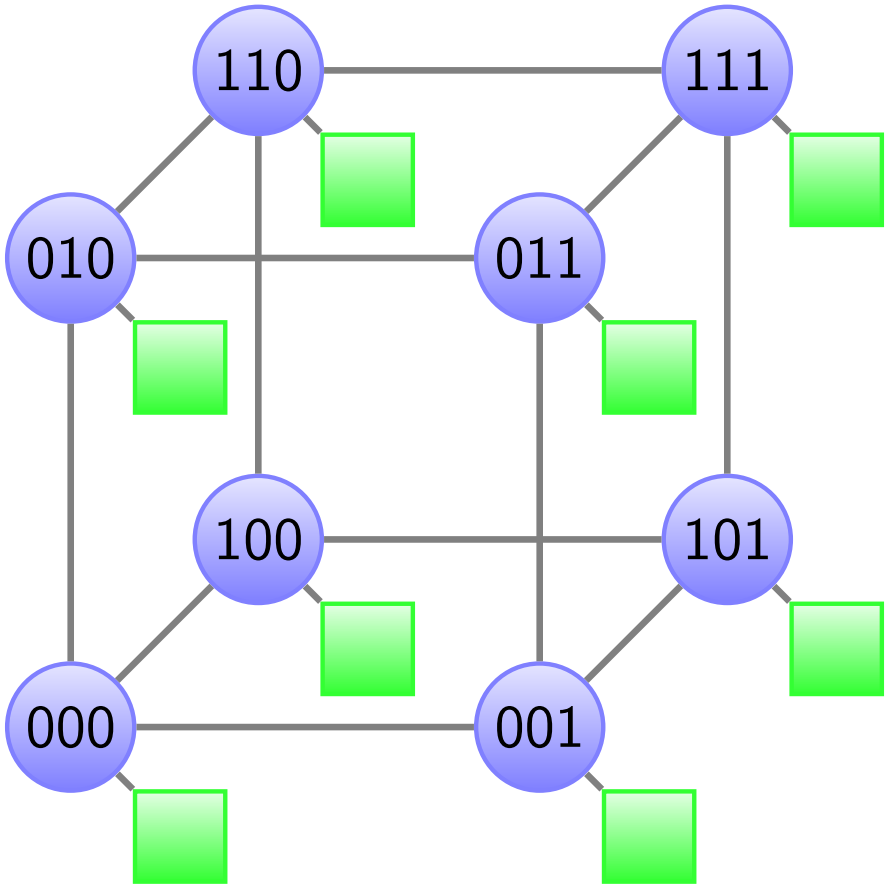
Butterfly network



Send message:
Node 2 to Node 5

Message arrives at node 5.

Hypercube network



Hypercube network

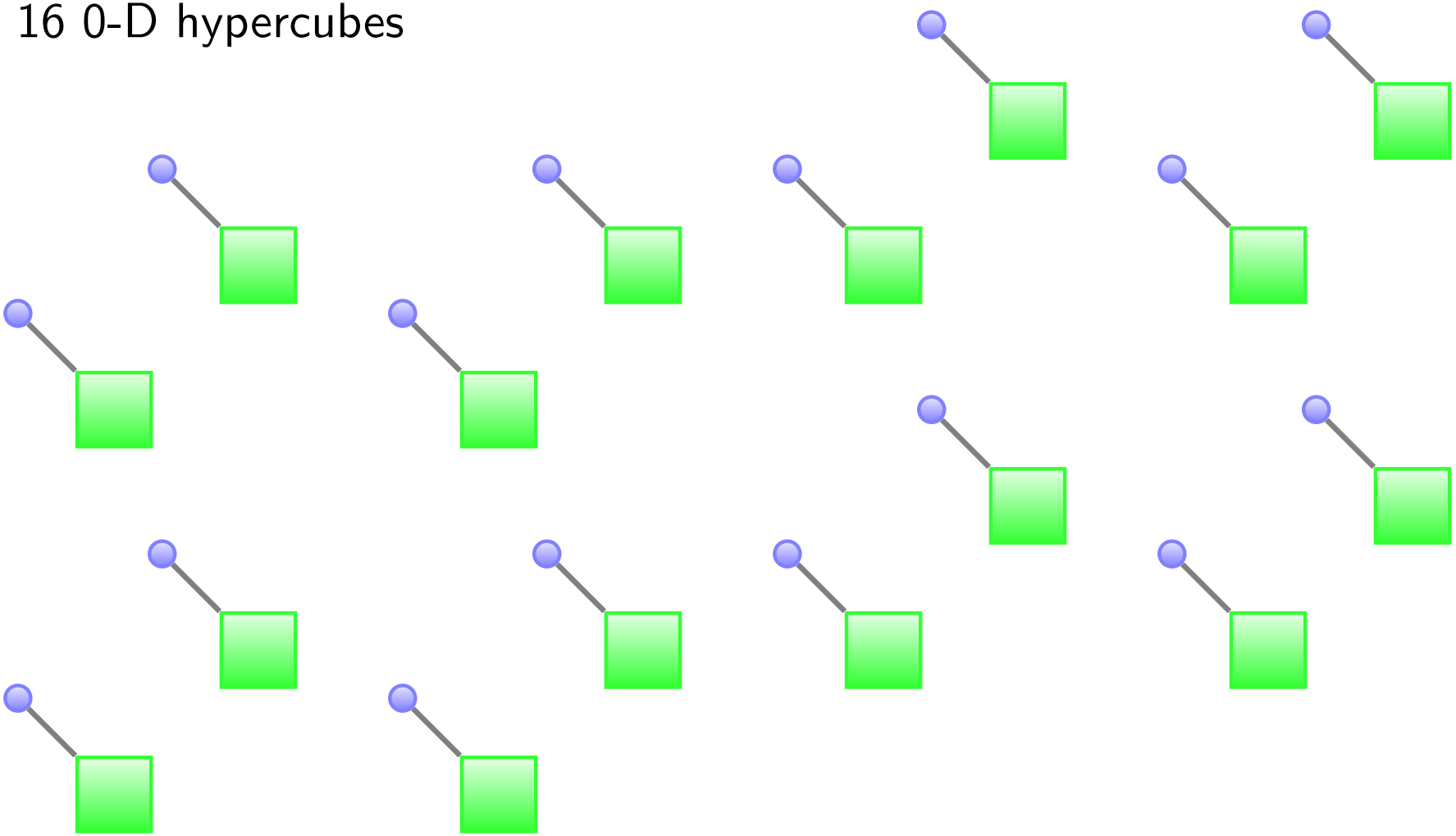
- Direct topology
- $n = 2^d$ processors
- 0-D hypercube is a single processor.
- 1-D hypercube is a "line" of $2^1 = 2$ processors with a single edge connecting them.
- 2-D hypercube is a square of $2^2 = 4$ processors with four edges.
- 3-D hypercube is a cube of $2^3 = 8$ processors with 12 edges.
- 4-D hypercube is a hypercube of $2^4 = 16$ processors with 32 edges.

Hypercube network

- Rule: to construct a $(d+1)$-D hypercube, take two d-D dimensional hypercubes and connect the corresponding vertices.
- Nodes (or switches for nodes) are numbered using a **Gray code**: if two nodes are adjacent then their codes differ by only a single bit.
- Diameter is $\log n = d$ and bisection width is $n/2$. There are $\log n = d$ edges per switch (not constant, which is not ideal) and the edge lengths do grow as the number of processors grows (also not ideal).

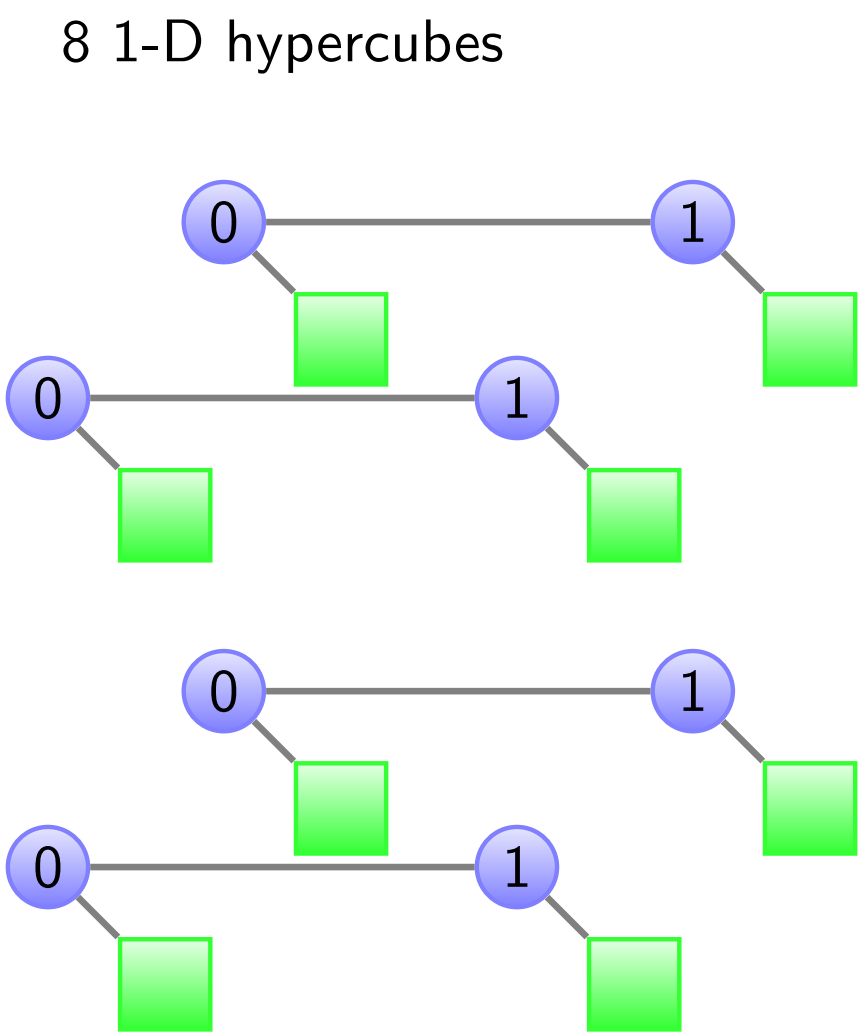
Hypercube network

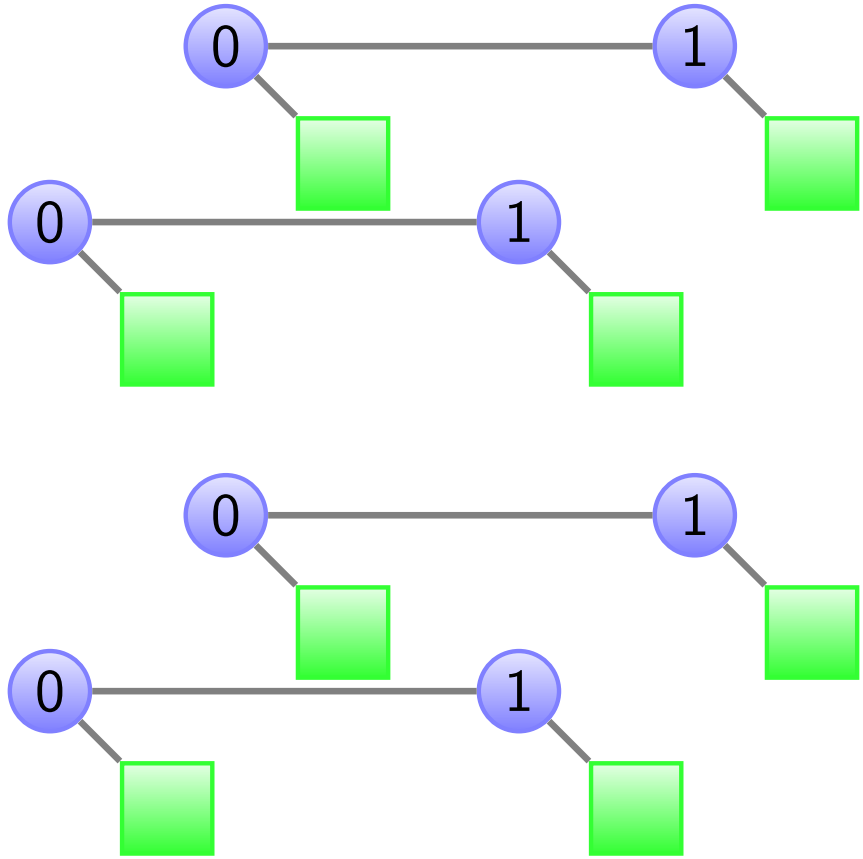
16 0-D hypercubes



Hypercube network

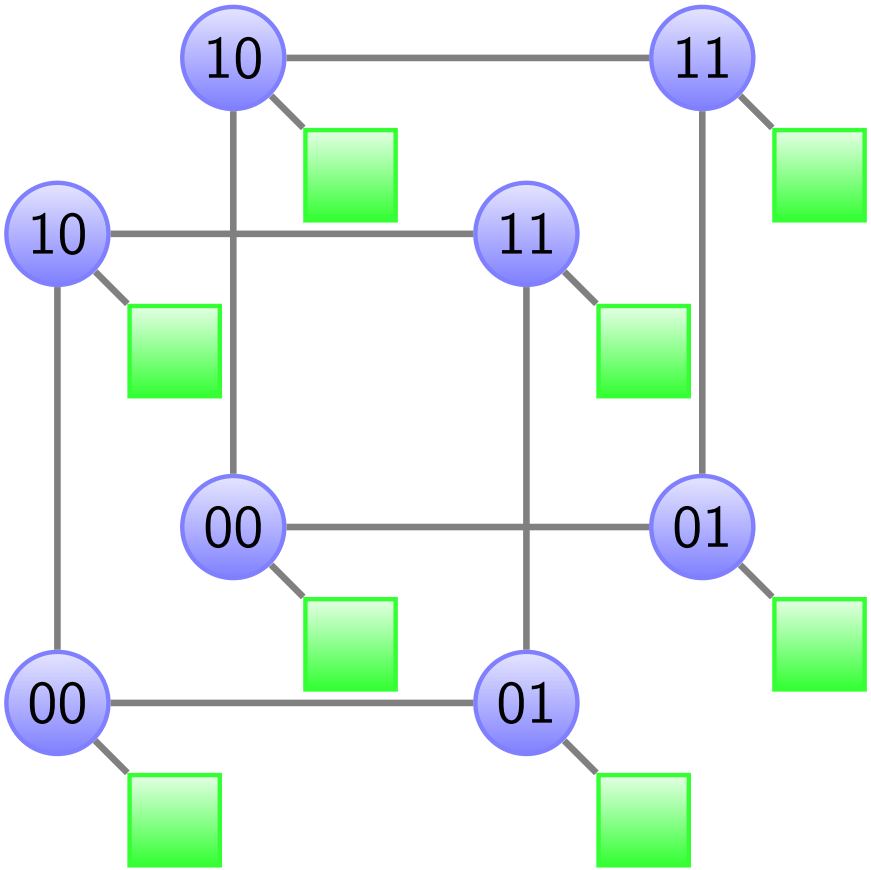
8 1-D hypercubes

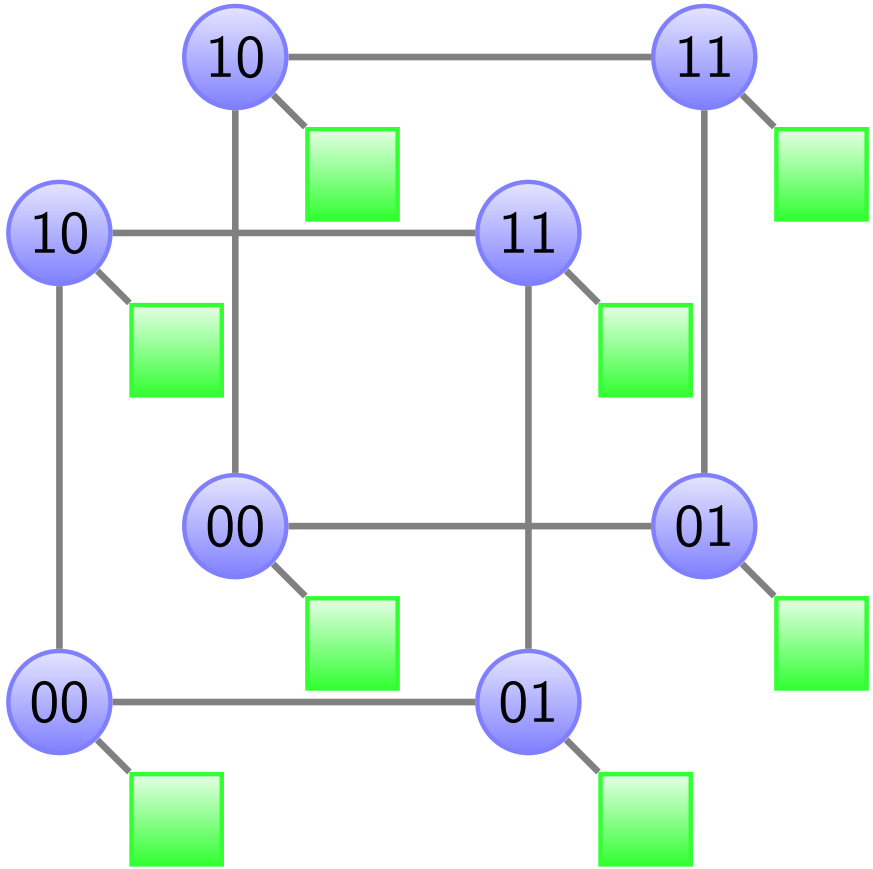




Hypercube network

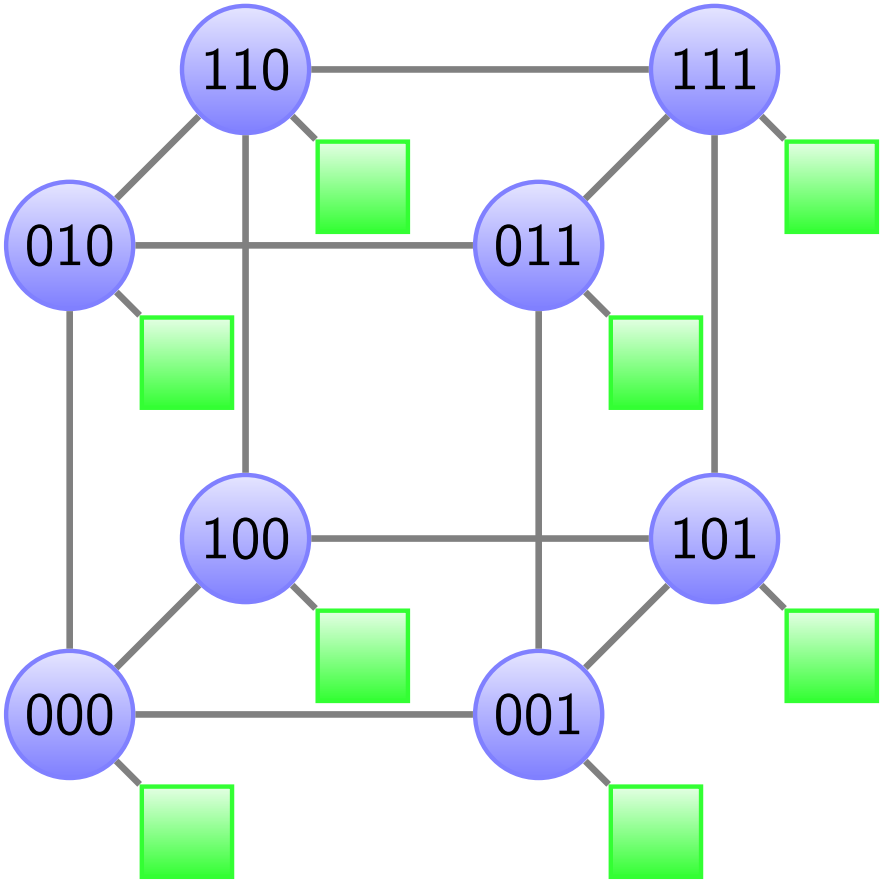
4 2-D hypercubes

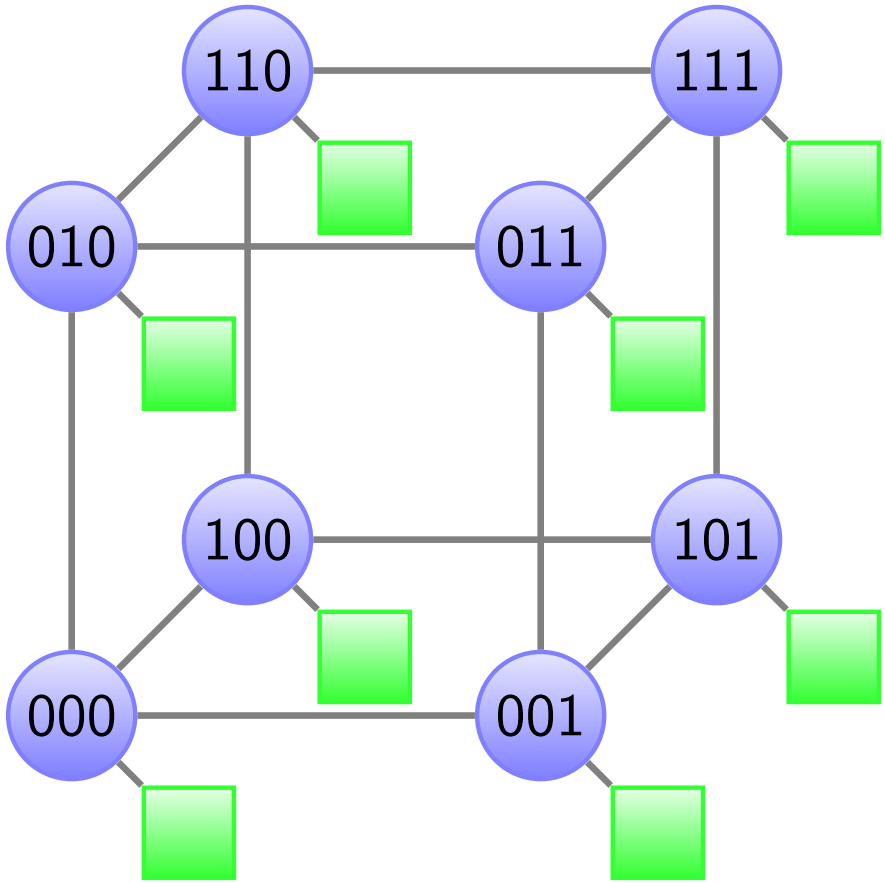




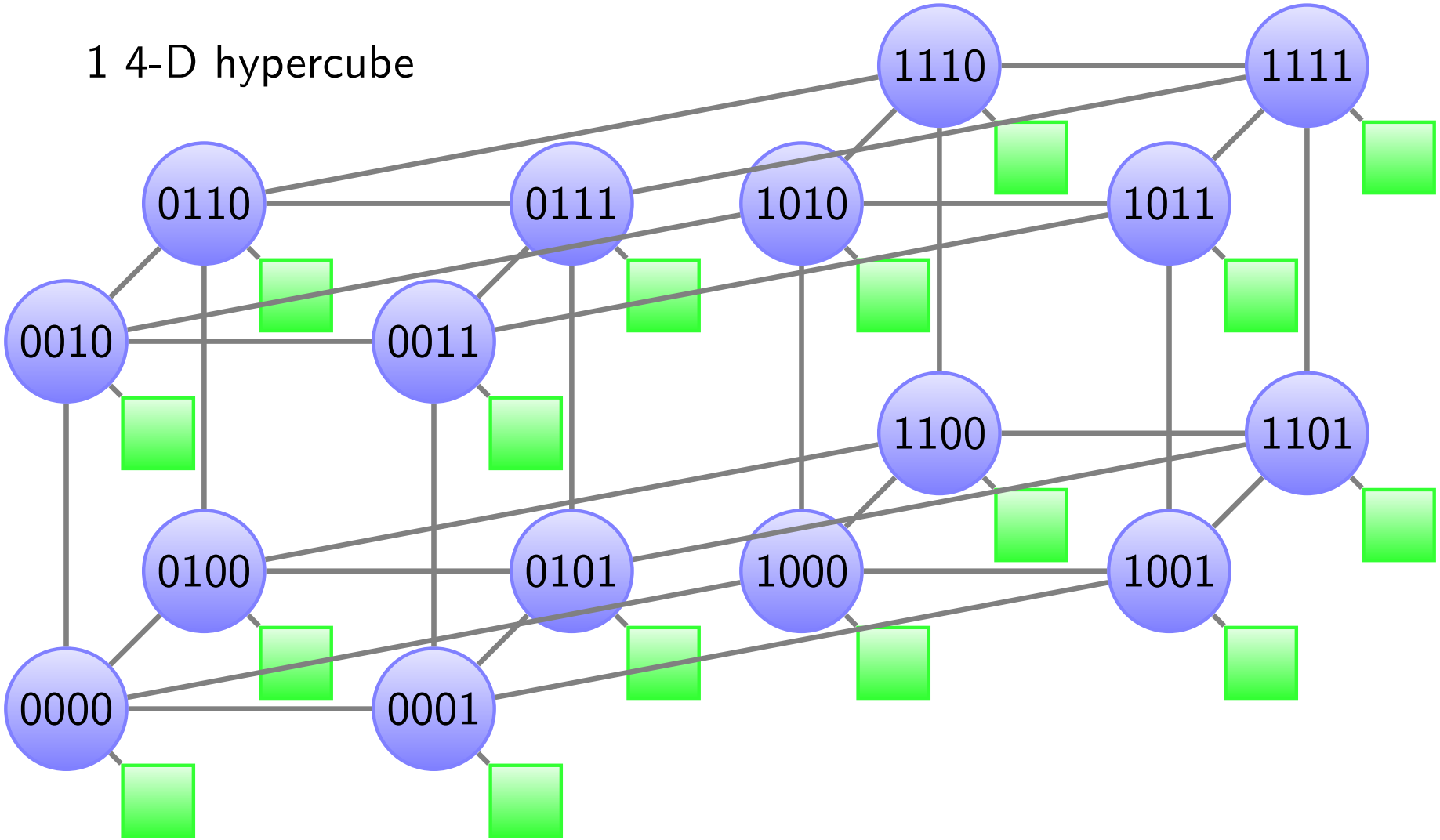
Hypercube network

2 3-D hypercubes





Hypercube network



Hypercube network

Routing in a hypercube network is fairly easy:

- To route from one processor to another: start with address of source node and flip any bit that differs from the corresponding bit in the destination address. Send to the node whose address was just generated. Repeat, always flipping a bit in a different position.
- Example: consider a 5-D hypercube with 32 processors. Each node has a 5 bit address. Suppose node 9 (01001_2) wants to send a message to node 30 (11110_2). The differing bits are $\mathbf{11111}$.
- The order in which the bits are selected is immaterial but does lead to non-unique communication paths. In this case there are $4! = 24$ possible paths; here are two of them:
  1. $01001 \rightarrow \mathbf{1}1001 \rightarrow 11\mathbf{1}01 \rightarrow 111\mathbf{1}1 \rightarrow 1111\mathbf{0}$
  2. $01001 \rightarrow 0100\mathbf{0} \rightarrow 010\mathbf{1}0 \rightarrow 01\mathbf{1}10 \rightarrow \mathbf{1}1110$

Outline

Processor arrays

- Multiple processors that carry out the same instruction in a given instruction cycle.
- Typically paired with a "front-end" processor that is handing all the other work of the system.
- Often called *vector processors* since operations on vectors typically involve identical operations on different data.
- Processors have local memory and some form of interconnection network (often a 2-D mesh) allowing for communication between processors.
- Masks can be used to selective enable/disable operations on individual processors. (e.g. think absolute value—only want to change sign if value is less than zero).

Processor array disadvantages

As late as the mid 2000's processor arrays were viewed as old technology. Reasons for this at the time included:

Processor array disadvantages

As late as the mid 2000's processor arrays were viewed as old technology. Reasons for this at the time included:

- many problems do not map well to strictly data-parallel solution
- conditionally executed parallel code is does not perform well
- not well suited for multi-user situations—requires dedicated access to processor array for best performance.
- cost does not scale-down well
- hard or impossible to do with COTS (commodity off-the-shelf) parts
- CPUs are relatively cheap (and getting faster)

Processor arrays: GPU and accelerators

- Recently processor arrays in the form of GPU and accelerators like Intel's Phi have become quite popular.
- We'll talk more about these soon...

Outline

Multiprocessors

- A computer with multiple CPUs or one or more multicore CPUs with shared memory.
- Common examples include dual-processor workstations and systems with multiple cores in a single processor. Past supercomputers include Cray X-MP and Y-MP.
- In a **Symmetric multiprocessor (SMP)** all processors are the same and have equal (but shared) access to resources.
- This is currently the standard model of desktop or laptop computers; a CPU with multiple cores, various levels of cache, and common access to single pool of RAM.

Multiprocessor cache issues

- Typically each CPU (or core; henceforth we'll just say CPU) has at least one level of cache. Data in the cache should be consistent with corresponding data in memory. When a CPU writes to its cache, that update must also be carried out in memory **and** in the caches of other CPUs where the updated data may also be stored. This is called the **cache coherence problem**.
- **Snooping** – each CPU's cache controller monitors bus so it is aware what is stored in other caches. System uses this information to avoid coherence problems.
- One solution is the **write invalidate protocol**: When one CPU write to its cache, the corresponding data in other's cache is marked as invalid. This causes a cache miss when any other CPU tries to read the data from its own cache.

Multiprocessor synchronization

- **Barriers** – Points in a parallel program where all processors must be at the same spot before proceeding.
- **Mutual Exclusion** – Often there are *critical sections* in code where the program must guarantee that only a single process accesses certain memory for a period of time. (e.g., don't want one process trying to read a memory location at the same instant another is writing to it.) **Semaphores** provide one way to do this.

Multiprocessor Memory Access

- **UMA (Unified Memory Access)**: Every processor has the same view and access to memory. Connections can be through a bus or a switched network. SMP machines typically belongs to the UMA category.
- Designs using a bus become less practical as the number of processors increases.
- **NUMA (NonUniform Memory Access) Distributed Multiprocessor**: Each processor has access to all memory but some access is indirect. Often memory is distributed and associated with each processor. There is a uniformly accessed virtual memory composed of all the distributed segments.

Outline

Multicomputers

- Commonly called *distributed memory computers*, these have multiple CPUs each having exclusive access to a certain segment of memory.
- Multicomputers can be **symmetrical** or **asymmetrical**.
- Symmetrical multicomputers are typically a collection of identical compute nodes.

Asymmetric vs symmetric multicomputers

- An asymmetrical multicomputer is a cluster of nodes with differentiated tasks and resources.
- Usually composed of front-end computer(s) called *head nodes* or *login nodes* and multiple back-end *compute nodes.*
- The head node typically handles program launching and supervision, I/O, network connections, as well as user-interface activities such as software development.
- Often programmed using either SPMD or MPMD model. In the SPMD case the program usually detects if it is running on the head node or a compute node and behaves accordingly.
- Supercomputer clusters that have dominated the supercomputer industries for the last twenty years are examples of asymmetrical multicomputers; many compute nodes and relatively few nodes for I/O, cluster supervision, and program development.
- Many small Beowulf clusters are symmetric multicomputers (or very nearly so).

Acknowledgements

Some material used in creating these slides comes from

- Parallel Programming in C with MPI and OpenMP, Michael Quinn, McGraw-Hill, 2004.