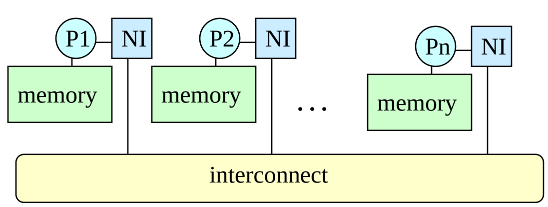
Distributed Memory Machines

Arvind Krishnamurthy
Fall 2004

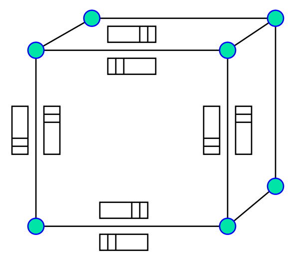
Distributed Memory Machines

- Intel Paragon, Cray T3E, IBM SP
- Each processor is connected to its own memory and cache:
  - cannot directly access another processor's memory.
- Each "node" has a network interface (NI) for all communication and synchronization
  - Key issues: design of NI and interconnection topology



Historical Perspective

- Early machines were:
  - Collection of microprocessors
  - bi-directional queues between neighbors
- Messages were forwarded by processors on path
- Strong emphasis on topology in algorithms



Network Analogy

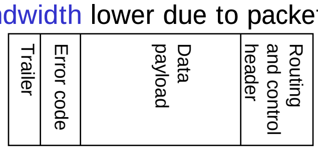
- To have a large number of transfers occurring at once, you need a large number of distinct wires
- Networks are like streets
  - link = street
  - switch = intersection
  - distances (hops) = number of blocks traveled
  - routing algorithm = travel plans
- Important Properties:
  - latency: how long to get somewhere in the network
  - bandwidth: how much data can be moved per unit time
    - limited by the number of wires
    - and the rate at which each wire can accept data

Network Characteristics

- Topology - how things are connected
  - two types of nodes: hosts and switches
  - Question: what nice properties do we want the network topology to possess?
- Routing algorithm - paths used
  - e.g., all east-west then all north-south in a mesh
- Switching strategy
  - how data in a message traverses a route
  - circuit switching vs. packet switching
- Flow control - what if there is congestion
  - if two or more messages attempt to use the same channel
  - may stall, move to buffers, reroute, discard, etc.

Topology Properties

- Routing Distance - number of links on route. Minimize average distance
- Diameter is the maximum shortest path between two nodes
- A network is partitioned if some nodes cannot reach others
- The bandwidth of a link is: w * 1/t
  - w is the number of wires
  - t is the time per bit
- Effective bandwidth lower due to packet overhead

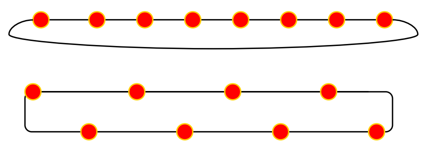


- Bisection bandwidth
  - sum of the minimum number of channels which, if removed, will partition the network

Linear and Ring Topologies

- Linear array

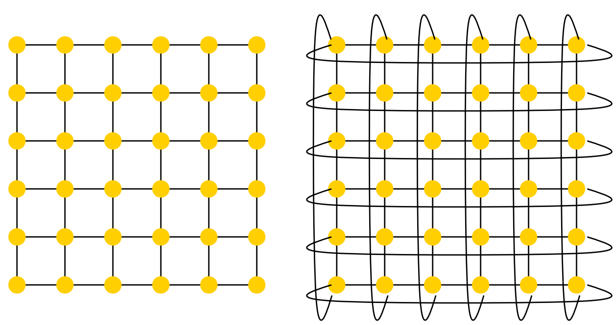
  - diameter is n-1, average distance ~2/3n
  - bisection bandwidth is 1
- Torus or Ring



  - diameter is n/2, average distance is n/3
  - bisection bandwidth is 2
- Used in algorithms with 1D arrays

Meshes and Tori

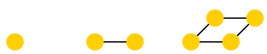
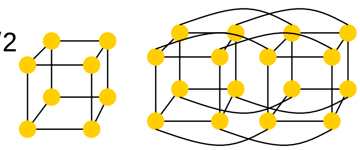
- 2D Mesh:
  - Diameter: $2\sqrt{n}$
  - Bisection bandwidth: $\sqrt{n}$



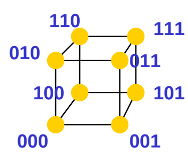
- Generalizes to 3D and higher dimensions
  - Cray T3D/T3E uses a 3D torus
  - Often easy to implement algorithms that use 2D-3D arrays

Hypercubes

- Number of nodes n = 2^d for dimension d
  - Diameter: d
  - Bisection bandwidth is n/2

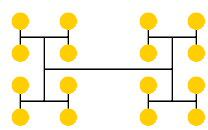
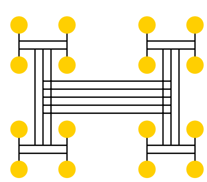
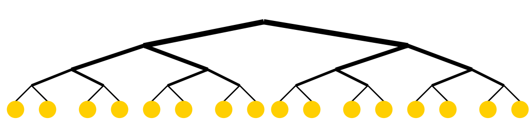



- Popular in early machines (Intel iPSC, NCUBE)
  - Lots of clever algorithms
- Greycode addressing
  - each node connected to “d” others with 1 bit different



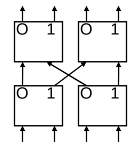
Trees

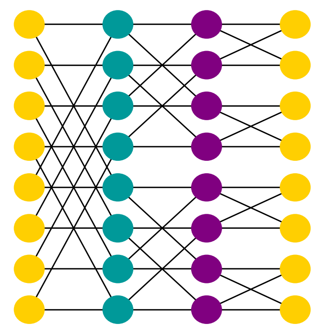
- Diameter: log n
- Bisection bandwidth: 1
- Easy layout as planar graph
- Many tree algorithms (summation)
- Fat trees avoid bisection bandwidth problem
  - more (or wider) links near top
  - example, Thinking Machines CM-5

Butterflies

- Butterfly building block
- Diameter: log n
- Bisection bandwidth: n
- Cost: lots of wires
- Use in BBN Butterfly
- Natural for FFT



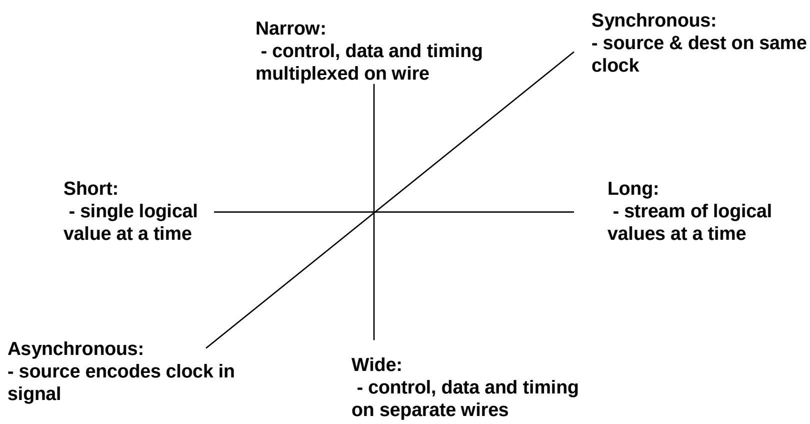


Outline

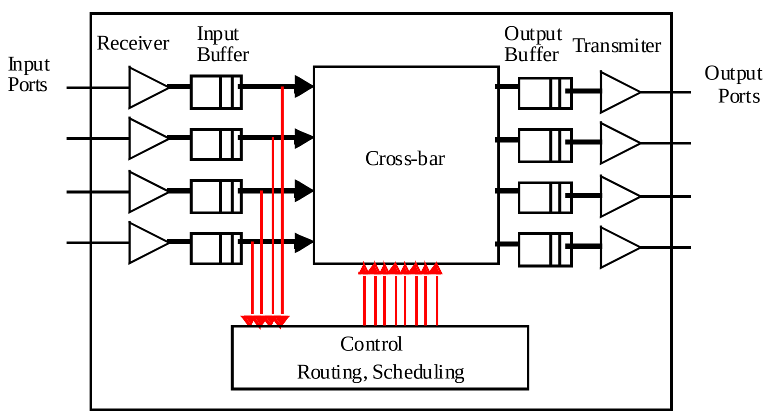
- Interconnection network issues:
  - Topology characteristics
    - Average routing distance
    - Diameter (maximum routing distance)
    - Bisection bandwidth
  - Link, switch design
  - Switching
    - Packet switching vs. circuit switching
    - Store-&-forward vs. cut-through routing
  - Routing

Link Design/Engineering Space

- Cable of one or more wires/fibers with connectors at the ends attached to switches or interfaces



Switches

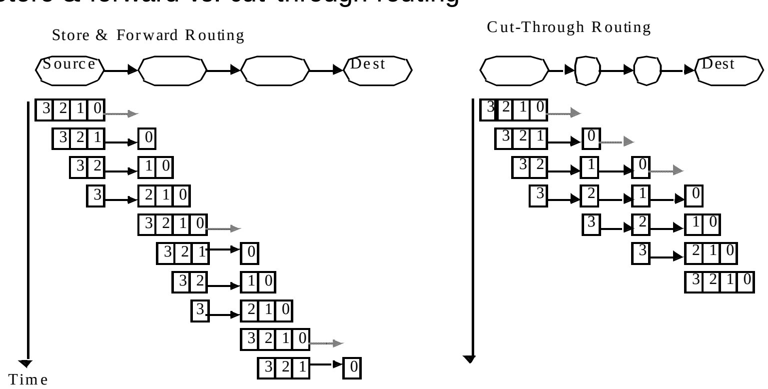


Switch Components

- Output ports
  - transmitter (typically drives clock and data)
- Input ports
  - synchronizer aligns data signal with local clock domain
  - essentially FIFO buffer
- Crossbar
  - connects each input to any output
  - degree limited by area or pinout
- Buffering
- Control logic
  - complexity depends on routing logic and scheduling algorithm
  - determine output port for each incoming packet
  - arbitrate among inputs directed at same output

Switching Strategies

- circuit switching: full path reserved for entire message
  - like the telephone
- packet switching: message broken into separately-routed packets
  - like the post office
- Question: what are the pros and cons of circuit switching & packet switching?
- Store & forward vs. cut-through routing



Outline

- Interconnection network issues:
  - Topology characteristics
    - Average routing distance
    - Diameter (maximum routing distance)
    - Bisection bandwidth
  - Switching
    - Packet switching vs. circuit switching
    - Store-&-forward vs. cut-through routing
  - Link, switch design
  - Routing

Routing

- Interconnection network provides multiple paths between a pair of source-dest nodes
- Routing algorithm determines
  - which of the possible paths are used as routes
  - how the route is determined
- Question: what desirable properties should the routing algorithm have?

Routing Mechanism

- need to select output port for each input packet
  - in a few cycles
- Simple arithmetic in regular topologies
  - ex: Δx, Δy routing in a grid
  - Encode distance to destination in header
    - west (-x) $\Delta x < 0$
    - east (+x) $\Delta x > 0$
    - south (-y) $\Delta x = 0, \Delta y < 0$
    - north (+y) $\Delta x = 0, \Delta y > 0$
    - processor $\Delta x = 0, \Delta y = 0$
- Reduce relative address of each dimension in order
  - Dimension-order routing in k-ary meshes

Routing Mechanism (cont)

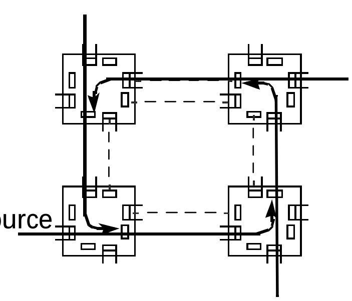
	P_3	P_2	P_1	P_0

- Source-based
  - message header carries series of port selects
  - used and stripped en route
  - Variable sized packets*: CRC? Packet Format?*
  - CS-2, Myrinet, MIT Arctic
- Table-driven
  - message header carried index for next port at next switch
    - o = R[i]
  - table also gives index for following hop
    - o, I' = R[i]
  - ATM, HPPI

Properties of Routing Algorithms

- Deterministic
  - route determined by (source, dest), not intermediate state (i.e., traffic)
- Adaptive
  - route influenced by traffic along the way
- Minimal
  - only selects shortest paths
- Deadlock free
  - no traffic pattern can lead to a situation where no packets cannot move forward

Deadlocks



- How can it arise?
  - necessary conditions:
    - shared resource
    - incrementally allocated
    - non-preemptible
  - think of a link/channel as a shared resource that is acquired incrementally
    - source buffer then dest. buffer
    - channels along a route
- How do you avoid it?
  - constrain how channel resources are allocated
  - Question: how do we avoid deadlocks in a 2D mesh?
- How do you prove that a routing algorithm is deadlock free

Proof Technique

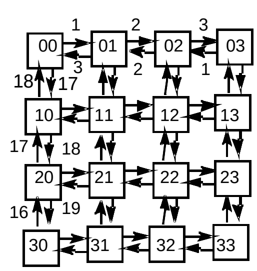
- resources are logically associated with channels
- messages introduce dependences between resources as they move forward
- need to articulate the possible dependences that can arise between channels;
- show that there are no cycles in Channel Dependence Graph
  - find a numbering of channel resources such that every legal route follows a monotonic sequence

  => no traffic pattern can lead to deadlock

- network need not be acyclic, only channel dependence graph

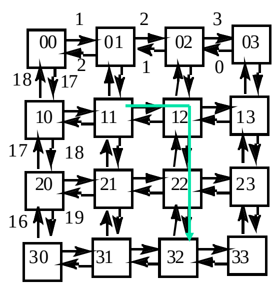
Example: 2D array

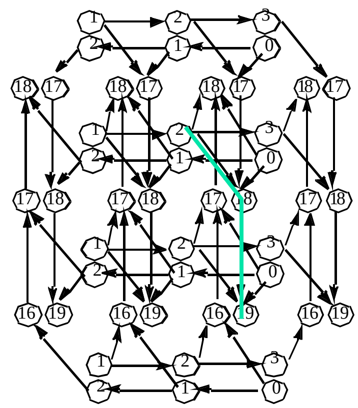
- Theorem: x,y routing is deadlock free
- Numbering
  - +x channel (i,y) → (i+1,y) gets i
  - -x channels are numbered in the reverse direction
  - +y channel (x,j) → (x,j+1) gets N+j
  - -y channels are numbered in the reverse direction
- any routing sequence: x direction, turn, y direction is increasing



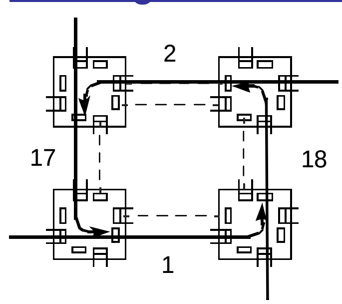
Channel Dependence Graph

Consider a message traveling from node 11 to node 12 and then to node 22, and finally to node 32.
It obtains channels numbered 2 and then 18 and then 19.





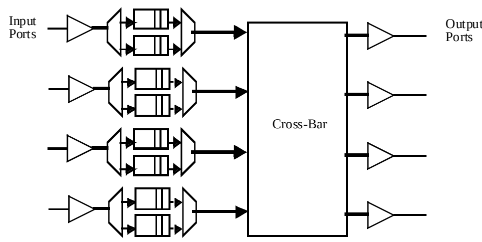
Routing Deadlocks



- If all turns are allowed, then channels are not obtained in increasing order
- Channel dependency graph will have a cycle:
  - Edges between 2:17, 17:1, 1:18, and 18:2
- Question: what happens with a torus (or wraparound connections)?
  - How do we avoid deadlocks in such a situation?

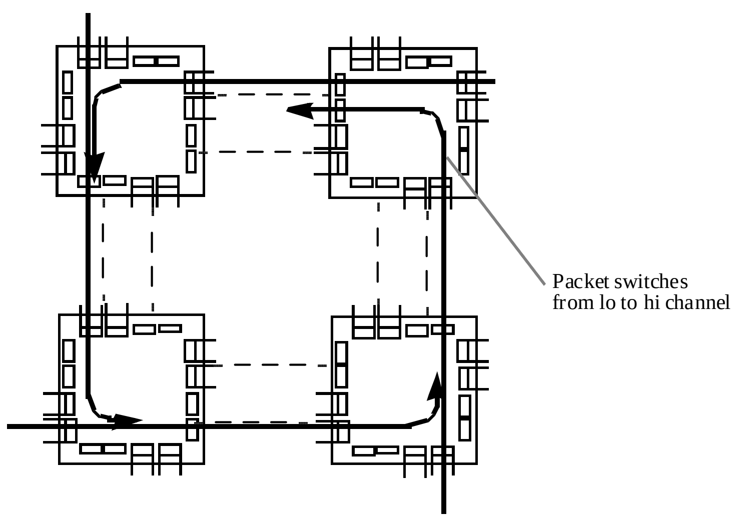
Deadlock free wormhole networks

- Basic dimension order routing techniques don't work with wrap-around edges
- Idea: add channels!
  - provide multiple "virtual channels" to break the dependence cycle
  - good for BW too!

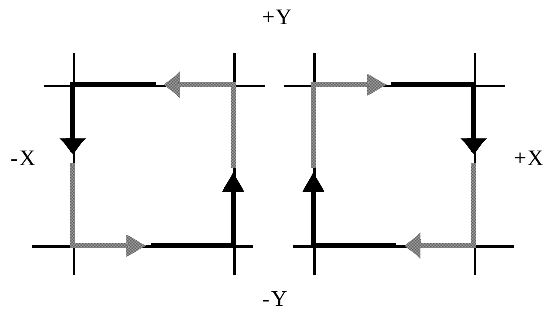


  - Do not need to add links, or xbar, only buffer resources
- This adds nodes to the CDG
  - Previous scheme removed edges

Breaking deadlock with virtual channels

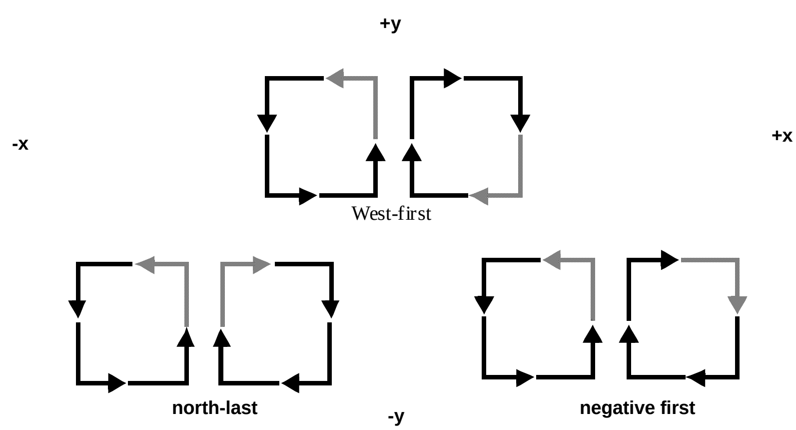


Turn Restrictions in X,Y

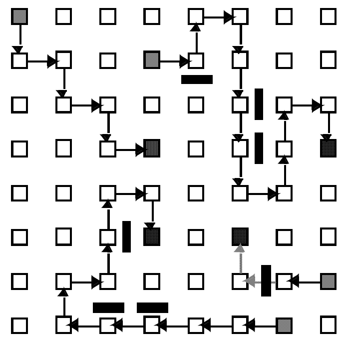


- XY routing forbids 4 of 8 turns and leaves no room for adaptive routing
- Can you allow more turns and still be deadlock free

Minimal turn restrictions in 2D



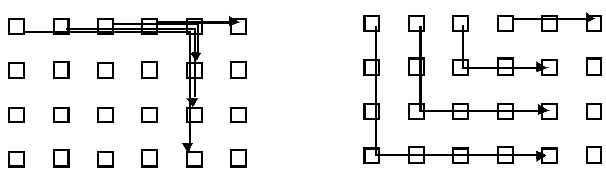
Example legal west-first routes



- Can route around failures or congestion
- Can combine turn restrictions with virtual channels

Adaptive Routing

- R: C x N x Σ -> C
- Essential for fault tolerance
- Can improve utilization of the network
- Simple deterministic algorithms easily run into bad permutations



- choices: fully/partially adaptive, minimal/non-minimal
- can introduce complexity or anomalies
- little adaptation goes a long way!

Up*-Down* routing

- Given any bi-directional network
- Construct a spanning tree
- Number of the nodes increasing from leaves to roots
  - Just a topological sort of the spanning tree
- Any Source -> Dest by UP*-DOWN* route
  - up edges, single turn, down edges
  - Up edge: any edge going from a lower numbered node to higher number
  - Down edges are the opposite
  - Not constrained to just using the spanning tree edges
- Performance?
  - Some numberings and routes much better than others
  - interacts with topology in strange ways

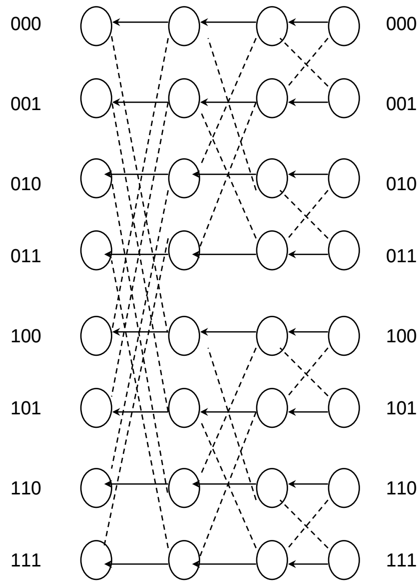
Topology Summary

Topology	Degree	Diameter	Ave Dist	Bisection	D (D ave) @ P=1024
1D Array	2	N-1	N / 3	1	huge
1D Ring	2	N/2	N/4	2	
2D Mesh	4	$2 (N^{1/2} - 1)$	$2/3\ N^{1/2}$	$N^{1/2}$	63 (21)
2D Torus	4	$N^{1/2}$	$1/2\ N^{1/2}$	$2N^{1/2}$	32 (16)
Butterfly	4	log N	log N	N	10 (10)
Hypercube	n =log N	n	n/2	N/2	10 (5)

- n = 2 or n = 3
  - Short wires, easy to build; Many hops, low bisection bandwidth
- n >= 4
  - Harder to build, more wires, longer average length
  - Fewer hops, better bisection bandwidth

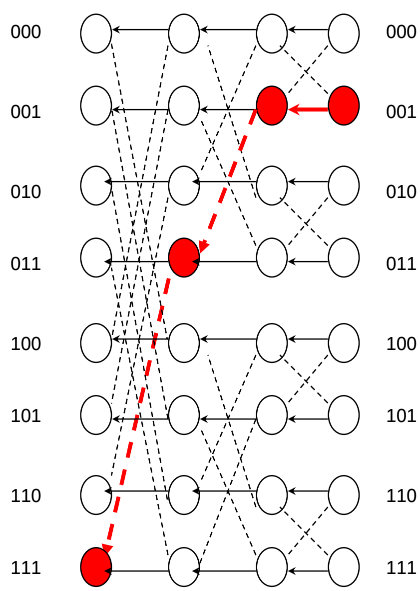
Butterfly Network

- Low diameter:
  - O(log N)
- Switches:
  - 2 incoming links
  - 2 outgoing links
- Processors:
  - Connected to the first and last levels



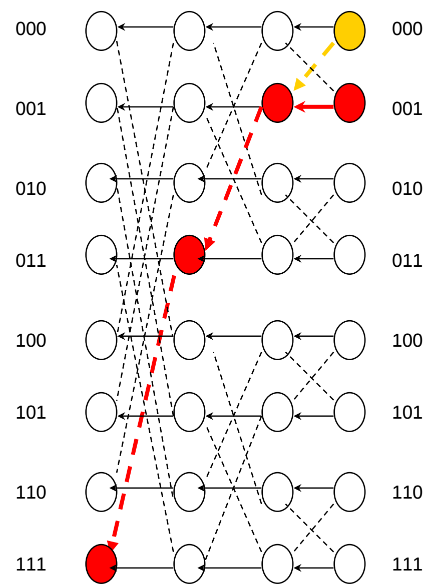
Routing in Butterfly Network

- Routes:
  - Single path from a source to a destination
  - Deterministic
  - Non-adaptive
  - Can run into congestion
- Routing algorithm
  - Correct bits one at a time
  - Consider: 001 → 111



Congestion

- Easy to have two routes share links
  - Consider: 001 → 111
  - And 000 → 011
- How bad can it get?
  - Consider general butterfly with 2r = log N levels
  - Consider routing from:
    Source: 00...0 11...1
    Dest: 11...1 00...0
  - Must pass through (after r): 00...0 00...0



Congestion: worst case scenario

- Bit reversal permutation:
  - $b_1\ b_2 \ldots b_{2r-1}\ b_{2r} \rightarrow b_{2r}\ b_{2r-1} \ldots b_2\ b_1$
- Consider just the following source-dest pairs:
  - Source: low-order r bits are zero
  - Of the form: $b_1\ b_2 \ldots b_r\ 0\ 0 \ldots 0 \rightarrow 0\ 0 \ldots 0\ b_r\ b_{r-1} \ldots b_1$
  - All of these pass through 0 0 ... 0 0 0 ... 0 after r routing steps
  - How many such pairs exist?
    - Every combination of $b_1\ b_2 \ldots b_r$
    - Number of combinations : $2^r = \text{sqrt}(2^{2r}) = \text{sqrt}(N)$
- Bad permutations exist for all interconnection networks
- Many networks perform well when you have locality or in the average case

Average Case Behavior: Butterfly Networks

- Question:
  - Assume one packet from each source, assume random destinations
  - How many packets go through some intermediate switch at level k in the network (on average)?

  - Sources that could generate a message: 2^k
  - Number of possible destinations: $2^{\log N - k}$
  - Expected congestion: $2^k * 2^{\log N - k} / 2^N = 1$

Randomized Algorithm

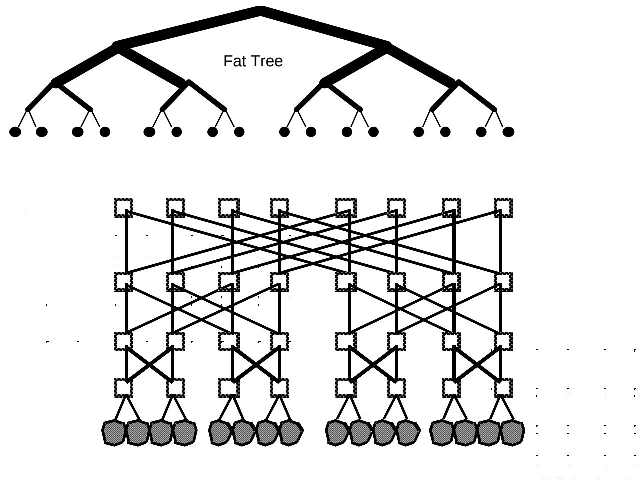
- How do we deal with bad permutations?
  - Turn them into two average-case behavior problems!

  - To route from source to dest:
    - Route from source to random node
    - Route from random node to destination

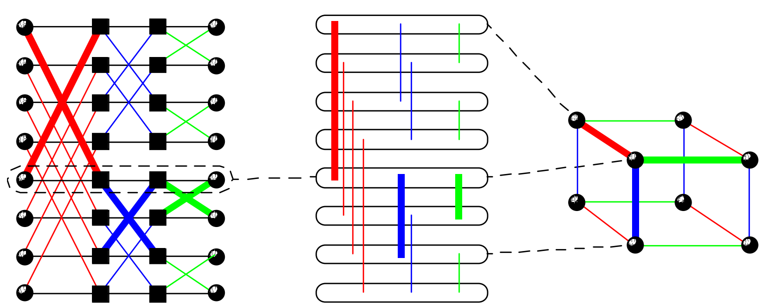
  - Turn initial routing problem into two average case permutations

Why Butterfly networks?

- Equivalence to hypercubes and fat-trees



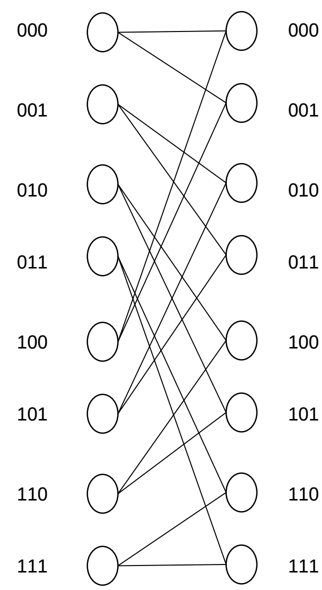
Relationship Butterflies to Hypercubes



- Wiring is isomorphic
- Except that Butterfly always takes log n steps

de Bruijn Network

- Each node has two outgoing links
- Node x is connected to 2*x, and 2*x + 1
- Example:
  - Node 000 is connected to Node 000 and Node 001
  - Node 001 is connected to Node 010 and Node 011
- How do we perform routing on such a network?
- What is the diameter of this network?



Summary

- We covered:
  - Popular topologies
  - Routing issues
    - Cut-through/store-and-forward/packet-switching/circuit-switching
    - Deadlock-free routes:
      - Limit paths
      - Introduce virtual channels
  - Link/switch design issues
  - Some popular routing algorithms
- From software perspective:
  - All that matters is that the interconnection network takes a chunk of bytes and communicates it to the target processor
  - Would be useful to abstract the interconnection network to some useful performance metrics

Linear Model of Communication Cost

- How do you model and measure point-to-point communication performance?
  - mostly independent of source and destination!
  - linear is often a good approximation
  - piecewise linear is sometimes better
  - the latency/bandwidth model helps understand performance

- A simple linear model:

  data transfer time = latency + message size / bandwidth

- latency is startup time, independent of message size
- bandwidth is number of bytes per second

Latency and Bandwidth

- for short messages, latency dominates transfer time
- for long messages, the bandwidth term dominates transfer time
- What are short and long?

  latency term = bandwidth term

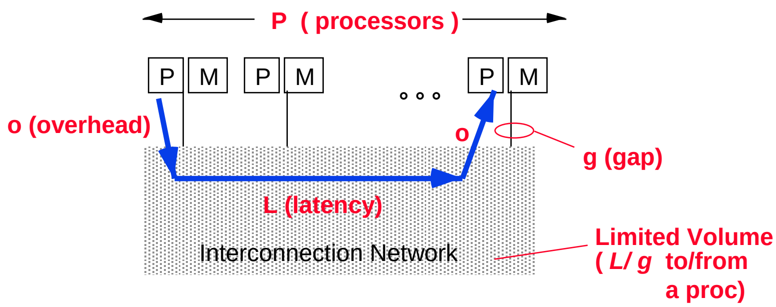
when

  latency = message_size/bandwidth

- Critical message size = latency * bandwidth
- Example: 50 us * 50 MB/s = 2500 bytes
  - messages longer than 2500 bytes are bandwidth dominated
  - messages shorter than 2500 bytes are latency dominated

- But linear model not enough
  - When can next transfer be initiated?
  - Can cost be overlapped?

LogGP Model



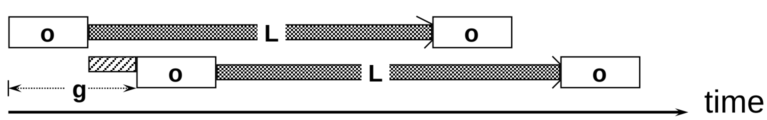
- Latency in sending a (small) message between modules
- overhead felt by the processor on sending or receiving message
- gap between successive sends or receives
- G: gap between successive bytes of the same message
- Processors

Using the Model

- Time to send a large message:

  L + o + size * G
- Time to send n small messages from one processor to another processor

  L + o + (n-1)*g
  - processor has *n*o* cycles of overhead
  - Has *(n-1)*(g-o)* idle cycles that could be overlapped with other computation



Some Typical LogGP values

- CM5:
  - L = 16.5 us
  - o = 6.0 us
  - g = 6.2 us
  - G = 0.125 us (8MB/s)
- Intel Paragon:
  - L = 20.5 us
  - o = 5.9 us
  - g = 8.3 us
  - G = 0.007 us (140 MB/s)
- T3D:
  - L = 0.85 us
  - o = 0.40 us
  - g = 0.40 us
  - G = 0.007 us (140 MB/s)

Message Passing Programs

- Separate processes, separate address spaces
- Processes execute independently and concurrently
- Processes transfer data cooperatively
- General version: Multiple Program Multiple Data (MPMD)

- Slightly constrained version:
  - Single Program Multiple Data (SPMD)
  - Single code image running on different processors
  - Can execute independently (or asynchronously), take different branches for instance

- MPI: most popular message passing library
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product

Hello World (Trivial)

- A simple, but not very interesting SPMD Program.
- To make this legal MPI, we need to add 2 lines.

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

Hello World (Independent Processes)

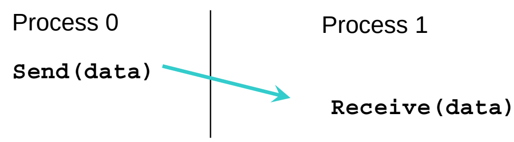
```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

- Processors belong to “communicators” (process groups)
- Default communicator is “MPI_COMM_WORLD”
- Communicators have a “size” and define a “rank” for each member

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will processes be identified?
  - How will “data” be described?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Point-to-Point Example

Process 0 sends array “A” to process 1 which receives it as “B”

```
1:
#define TAG 123
double A[10];
MPI_Send(A, 10, MPI_DOUBLE, 1,
               TAG, MPI_COMM_WORLD)
2:
#define TAG 123
double B[10];
MPI_Recv(B, 10, MPI_DOUBLE, 0,
               TAG, MPI_COMM_WORLD, &status)
```

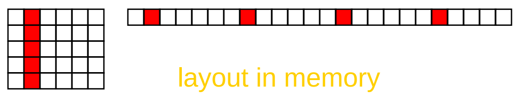
or

```
MPI_Recv(B, 10, MPI_DOUBLE, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &status)
```

status: useful for querying the tag, source after reception

MPI DataTypes

- The data in a message to be sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
    - Goal: support heterogeneous clusters
  - a contiguous array of MPI datatypes
  - a strided block of datatypes



  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- May improve performance:
  - reduces memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available

Collective Communication in MPI

- Collective operations are called by all processes in a communicator.
  - **MPI_BCAST** distributes data from one process to all others in a communicator.

```
MPI_Bcast(start, count, datatype,
          source, comm);
```

  - **MPI_REDUCE** combines data from all processes in communicator and returns it to one process.

```
MPI_Reduce(in, out, count, datatype,
           operation, dest, comm);
```

For example:

```
MPI_Reduce(&mysum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
```

Non-blocking Operations

Split communication operations into two parts.

- First part initiates the operation. It does not block.
- Second part waits for the operation to complete.

```
MPI_Request request;
MPI_Recv(buf, count, type, dest, tag, comm, status)
    =
MPI_Irecv(buf, count, type, dest, tag, comm, &request)
    +
MPI_Wait(&request, &status)

MPI_Send(buf, count, type, dest, tag, comm)
    =
MPI_Isend(buf, count, type, dest, tag, comm, &request)
    +
MPI_Wait(&request, &status)
```

Using Non-blocking Receive

- Two advantages:
  - No deadlock (correctness)

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

  - Data may be transferred concurrently (performance)

Process 0

```
Isend(1)
…compute…
Wait()
```

Operations on MPI_Request

- MPI_Wait(INOUT request, OUT status)
  - Waits for operation to complete and returns info in status
  - Frees request object (and sets to MPI_REQUEST_NULL)
- MPI_Test(INOUT request, OUT flag, OUT status)
  - Tests to see if operation is complete and returns info in status
  - Frees request object if complete
- MPI_Request_free(INOUT request)
  - Frees request object but does not wait for operation to complete
- Wildcards:
  - MPI_Waitall(..., INOUT array_of_requests, ...)
  - MPI_Testall(..., INOUT array_of_requests, ...)
  - MPI_Waitany/MPI_Testany/MPI_Waitsome/MPI_Testsome

Non-Blocking Communication Gotchas

- Obvious caveats:
  - 1. You may not modify the buffer between Isend() and the corresponding Wait(). Results are undefined.
  - 2. You may not look at or modify the buffer between Irecv() and the corresponding Wait(). Results are undefined.
  - 3. You may not have two pending Irecv()s for the same buffer.
- Less obvious:
  - 4. You may not *look* at the buffer between Isend() and the corresponding Wait().
  - 5. You may not have two pending Isend()s for the same buffer.
- **Why the isend() restrictions?**
  - Restrictions give implementations more freedom, e.g.,
    - Heterogeneous computer with differing byte orders
    - Implementation swap bytes in the original buffer