**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

```
+----+      +----+      +----+      +----+
| P1 |   NI   | P2 |   NI   | Pn |   NI   |
|     |        |     |        |     |        |
| memory |      | memory |      | memory |      |
```

**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

**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: \( w * \frac{1}{t} \)
  - \( w \) 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

**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.

**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

```
+----+      +----+      +----+      +----+
| P1 |   NI   | P2 |   NI   | Pn |   NI   |
|     |        |     |        |     |        |
| memory |      | memory |      | memory |      |
```

**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

**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: \( w * \frac{1}{t} \)
  - \( w \) 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

**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.
**Linear and Ring Topologies**

- Linear array
  - Diameter is $n-1$, average distance $\sim \frac{2}{3}n$
  - Bisection bandwidth is 1
- Torus or Ring
  - Diameter is $\frac{n}{2}$, average distance $\sim \frac{n}{3}$
  - Bisection bandwidth is 2
  - Used in algorithms with 1D arrays

**Meshes and Tori**

- 2D Mesh:
  - Diameter: $\sqrt{n}$
  - Bisection bandwidth: $n$
- 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 $\frac{n}{2}$
- Popular in early machines (Intel iPSC, NCUBE)
  - Lots of clever algorithms
- Greycode addressing
  - Each node connected to 6 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
  - Used 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**

- Input Ports
- Buffer
- Transmitter
- Cross-bar
- Output Ports

**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: $\Delta x$, $\Delta y$ 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)

- 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 $(i,y) \rightarrow (i+1,y)$ gets i
  - $-x$ channels are numbered in the reverse direction
  - $+y$ channel $(x,j) \rightarrow (x,j+1)$ 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

---

**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**

<table>
<thead>
<tr>
<th>Topology</th>
<th>Degree</th>
<th>Diameter</th>
<th>Ave Dist</th>
<th>Bisection</th>
<th>D (D ave) @ P=1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D Array</td>
<td>2</td>
<td>N-1</td>
<td>N / 3</td>
<td>1</td>
<td>huge</td>
</tr>
<tr>
<td>1D Ring</td>
<td>2</td>
<td>N/2</td>
<td>N/4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2D Mesh</td>
<td>4</td>
<td>2 (N^1/2 - 1)</td>
<td>2/3 N^1/2</td>
<td>N^1/2</td>
<td>63 (21)</td>
</tr>
<tr>
<td>2D Torus</td>
<td>4</td>
<td>N^1/2</td>
<td>1/2 N^1/2</td>
<td>2N^1/2</td>
<td>32 (16)</td>
</tr>
<tr>
<td>Butterfly</td>
<td>4</td>
<td>log N</td>
<td>log N</td>
<td>N</td>
<td>10 (10)</td>
</tr>
<tr>
<td>Hypercube</td>
<td>n=log N</td>
<td>n</td>
<td>n/2</td>
<td>N/2</td>
<td>10 (4)</td>
</tr>
</tbody>
</table>

- 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

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:
  - $b_1 b_2 \ldots b_2r \rightarrow b_2 b_{2r-1} \ldots b_1$
- Consider just the following source-dest pairs:
  - Source: low-order $r$ bits are zero
    - Of the form: $b_1 b_2 \ldots b_r 0 0 \ldots 0 \rightarrow 0 0 \ldots 0 b_1 b_2 \ldots b_r$
    - 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: $2^r = \sqrt{2^{2r}} = \sqrt{N}$
- 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: $2^k \times 2 \log N - k / 2N = 1$

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\(\times\)x, and 2\(\times\)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
  - \( \text{latency} = \frac{\text{message size}}{\text{bandwidth}} \)

- Critical message size = latency \times bandwidth
- Example: 50 µs \times 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**

- Time to send a large message:
  \( L + o + \text{size} \times G \)

- Time to send n small messages from one processor to another processor:
  \( L + o + (n-1)g \)
  - Processor has \( n \times o \) cycles of overhead
  - Has \( (n-1)(g-o) \) idle cycles that could be overlapped with other computation

**Using the Model**
Some Typical LogGP values

- CM5:
  - L = 16.5 \text{ us}
  - o = 6.0 \text{ us}
  - g = 6.2 \text{ us}
  - G = 0.125 \text{ us (8}\text{MB/s)}

- Intel Paragon:
  - L = 20.5 \text{ us}
  - o = 5.9 \text{ us}
  - g = 8.3 \text{ us}
  - G = 0.007 \text{ us (140}\text{ MB/s)}

- T3D:
  - L = 0.85 \text{ us}
  - o = 0.40 \text{ us}
  - g = 0.40 \text{ us}
  - G = 0.007 \text{ us (140}\text{ 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.

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

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

Hello World (Independent Processes)

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

```c
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

```c
Process 0  Process 1
Send(data)  Receive(data)
```

- 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" 

```c
#define TAG 123

double A[10];
MPI_Send(A, 10, MPI_DOUBLE, 1,
         TAG, MPI_COMM_WORLD)
```

```c
#define TAG 123
double B[10];
MPI_Recv(B, 10, MPI_DOUBLE, 0,
         TAG, MPI_COMM_WORLD, &status)
```

- status: useful for querying the tag, source after reception
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 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 a 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_Rcvc(buf, count, type, dest, tag, comm, &status);
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);
```

Two advantages:
- No deadlock (correctness)
- Data may be transferred concurrently (performance)

Process 0
- Send(1)
- Recv(1)

Process 1
- Send(0)
- Recv(0)

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(...)`, `MPI_Testall(...)`, `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