Overview

1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes
Overview

1. Definition

2. MPI
   - Efficient communication

3. Collective Communications

4. Interconnection networks
   - Static networks
   - Dynamic networks

5. End notes
Message-passing paradigm

- Partitioned address space
  - Each process has its own exclusive address space
  - Typical 1 process per processor

- Only supports explicit parallelization
  - Adds complexity to programming
  - Encourages locality of data access

- Often Single Program Multiple Data (SPMD) approach
  - The same code is executed by every process.
  - Identical, except for the master
  - *loosely synchronous* paradigm: between interactions (through messages), tasks execute completely asynchronously
Clusters

- Message-passing
- Made from commodity parts
  - or blade servers
- Open-source software available
Computing Grids

Provide computing resources as a service

- Hiding details for the users (transparency)
- Users: enterprises such as financial services, manufacturing, gaming, ...
- Hire computing resources, besides data storage, web servers, etc.

Issues:

- Resource management, availability, transparency, heterogeneity, scalability, fault tolerance, security, privacy.
Cloud Computing, the new hype

- Internet-based computing, whereby shared resources, software, and information are provided to computers and other devices on demand
- Like the electricity grid.
The ability to send and receive messages is all we need

- void send(sendBuffer, messageSize, destination)
- void receive(receiveBuffer, messageSize, source)
- boolean probe(source)

But... we also want performance!
  - More functions will be provided
Message-passing Parallel Processing

Jan Lemeire
Overview

1. Definition

2. MPI
   - Efficient communication

3. Collective Communications

4. Interconnection networks
   - Static networks
   - Dynamic networks

5. End notes
MPI: the Message Passing Interface

- A standardized message-passing API.
- There exist nowadays more than a dozen implementations, like LAM/MPI, MPICH, etc.
- For writing portable parallel programs.
- Runs transparently on heterogeneous systems (platform independence).
- Aims at not sacrificing efficiency for genericity:
  - encourages overlap of communication and computation by nonblocking communication calls.
Replaces the good old PVM (Parallel Virtual Machine)
Fundamentals of MPI

- Each process is identified by its **rank**, a counter starting from 0.
- **Tags** let you distinguish different types of messages.
- **Communicators** let you specify groups of processes that can intercommunicate
  - Default is `MPI_COMM_WORLD`
- All MPI routines in C, data-types, and constants are prefixed by "`MPI_`"
- We use the MPJ API, an O-O version of MPI for java
### The minimal set of MPI routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>Initializes MPI.</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>Terminates MPI.</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>Determines the number of processes.</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>Determines the label of calling process.</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>Sends a message.</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Receives a message.</td>
</tr>
<tr>
<td>MPI_Probe</td>
<td>Test for message (returns Status object).</td>
</tr>
</tbody>
</table>
Counting 3s with MPI

- **Master**
  - Partition array
  - Send subarray to each slave
  - Receive results and sum them

- **Slaves**
  - Receive subarray
  - Count 3s
  - Return result

Different program on master and slave

We’ll see an alternative later
int rank = MPI.COMM_WORLD.Rank(); int size = MPI.COMM_WORLD.Size(); int nbrSlaves = size - 1;
if (rank == 0) { // we choose rank 0 for master program

// initialise data
int[] data = createAndFillArray(arraySize);

// divide data over slaves
int slavedata = arraySize / nbrSlaves; // # data for one slave
int index = 0;

for (int slaveID=1; slaveID < size; slaveID++) {
    MPI.COMM_WORLD.Send(data, index, slavedata + rest, MPI.INT, slaveID, INPUT_TAG);
    index += slavedata;
}

// slaves are working...
int nbrPrimes = 0;
for (int slaveID=1; slaveID < size; slaveID++) {
    int[] buff = new int[1]; // allocate buffer size of 1
    MPI.COMM_WORLD.Recv(buff, 0, 1, MPI.INT, slaveID, RESULT_TAG);
    nbrPrimes += buff[0];
}
} else { // *** Slave Program ***
    Status status = MPI.COMM_WORLD.Probe(0, INPUT_TAG);
    int[] array = new int[status.count]; // check status to know data size
    MPI.COMM_WORLD.Recv(array, 0, status.count, MPI.INT, 0, INPUT_TAG);

    int result = countPrimes(array); // sequential program

    int[] buff = new int[] {result};
    MPI.COMM_WORLD.Send(buff, 0, 1, MPI.INT, 0, RESULT_TAG)
}

MPI.Finalize(); // Don't forget!!
void Comm.Send(java.lang.Object buf, int offset, int count, Datatype datatype, int dest, int tag)

Status Comm.Recv(java.lang.Object buf, int offset, int count, Datatype datatype, int source, int tag)
A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.

- Default is `COMM_WORLD`, includes all the processes
- Define others when communication is restricted to certain subsets of processes

Information about communication domains is stored in variables of type `Comm`.

Communicators are used as arguments to all message transfer MPI routines.

A process can belong to many different (possibly overlapping) communication domains.
A process has a specific rank in each communicator it belongs to.

**Other example**: use a different communicator in a library than application so that messages don’t get mixed.
# MPI Datatypes

<table>
<thead>
<tr>
<th>MPI++ Datatype</th>
<th>C Datatype</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI.CHAR</td>
<td>signed char</td>
<td>char</td>
</tr>
<tr>
<td>MPI.SHORT</td>
<td>signed short int</td>
<td>int</td>
</tr>
<tr>
<td>MPI.INT</td>
<td>signed int</td>
<td>int</td>
</tr>
<tr>
<td>MPI.LONG</td>
<td>signed long int</td>
<td>long</td>
</tr>
<tr>
<td>MPI.UNSIGNED_CHAR</td>
<td>unsigned char</td>
<td></td>
</tr>
<tr>
<td>MPI.UNSIGNED_SHORT</td>
<td>unsigned short int</td>
<td></td>
</tr>
<tr>
<td>MPI.UNSIGNED</td>
<td>unsigned int</td>
<td></td>
</tr>
<tr>
<td>MPI.UNSIGNED_LONG</td>
<td>unsigned long int</td>
<td></td>
</tr>
<tr>
<td>MPI.FLOAT</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>MPI.DOUBLE</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>MPI.LONG_DOUBLE</td>
<td>long double</td>
<td></td>
</tr>
<tr>
<td>MPIBYTE</td>
<td></td>
<td>byte</td>
</tr>
<tr>
<td>MPI.PACKED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
User-defined datatypes

- Specify displacements and types => commit
- Irregular structure: use DataType.Struct
- Regular structure: Indexed, Vector, ...
  - E.g. submatrix
- Alternative: packing & unpacking via buffer
Packing & unpacking

Example: tree

From objects and pointers to a linear structure... and back.
Inherent serialization in java

- For your class: implement interface `Serializable`
  - No methods have to be implemented, this turns on automatic serialization
- Example code of writing object to file:
  ```java
  public static void writeObject2File(File file, Serializable o) throws FileNotFoundException, IOException{
      FileOutputStream out = new FileOutputStream(file);
      ObjectOutputStream s = new ObjectOutputStream(out);
      s.writeObject(o);
      s.close();
  }
  ```
- Add `serialVersionUID` to denote class compatibility
  - `private static final long serialVersionUID = 2;`
- Attributes denoted as `transient` are not serialized
Overview

1. Definition

2. MPI
   - Efficient communication

3. Collective Communications

4. Interconnection networks
   - Static networks
   - Dynamic networks

5. End notes
Message-passing
Non-Buffered Blocking Message Passing Operations

- **Handshake** for a blocking non-buffered send/receive operation.
- There can be considerable idling overheads.
Non-Blocking communication

With support for overlapping communication with computation

Legend:
- process consuming cycles
- doing other things
Non-Blocking Message Passing Operations

- With HW support: communication overhead is completely masked (*Latency Hiding 1*)
  - Network Interface Hardware allow the transfer of messages without CPU intervention

- Message can also be buffered
  - Reduces the time during which the data is unsafe
  - Initiates a DMA operation and returns immediately
    - DMA (Direct Memory Access) allows copying data from one memory location into another without CPU support (*Latency Hiding 2*)

- Generally accompanied by a check-status operation (whether operation has finished)
Be careful!

Consider the following code segments:

```
P0
a = 100;
send(&a, 1, 1);
a=0;
```

```
P1
receive(&a, 1, 0);
cout << a << endl;
```

Which protocol to use?

- **Blocking protocol**
  - Idling...

- **Non-blocking buffered protocol**
  - Buffering alleviates idling at the expense of copying overheads
Non-blocking buffered communication
Deadlock with blocking calls

**Solutions**

- Switch send and receive at uneven processor
- Use non-blocking calls
  - Receive should use a different buffer!
- MPI provides a built-in solution (see later)

```c
All processes
If (rank % 2 == 0){
    send(&a, 1, rank+1);
    receive(&a, 1, rank-1);
} else {
    receive(&b, 1, rank-1);
    send(&a, 1, rank+1);
    a=b;
}
```
Send and Receive Protocols

- **Buffered**
  - **Blocking Operations**
    - Sending process returns after data has been copied into communication buffer
  - **Non-Blocking Operations**
    - Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return

- **Non-Buffered**
  - **Blocking Operations**
    - Sending process blocks until matching receive operation has been encountered
  - **Non-Blocking Operations**
    - Programmer must explicitly ensure semantics by polling to verify completion

Send and Receive semantics assured by corresponding operation
MPI Point-to-point communication

- **Blocking**
  - Returns if locally complete (<> globally complete)

- **Non-blocking**
  - Wait & test for completion functions

- **Modes**
  - Buffered
  - Synchronous: wait for a rendez-vous
  - Ready: no hand-shaking or buffering
    - Assumes corresponding receive is posted

- **Send_recv & send_recv_replace**
  - Simultaneous send & receive. Solves slide 30 problem!
Overview

1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes
Collective Communication Operations

- MPI provides an extensive set of functions for performing common collective communication operations.
- Each of these operations is defined over a group corresponding to the communicator.
- All processors in a communicator must call these operations.
- For convenience & performance
  - Collective operations can be optimized by the library by taking the underlying network into consideration!
Counting 3s with MPI \textit{bis}

- The same program on master and slave

\begin{itemize}
  \item \textbf{All processes}
  \item allocate subarray
  \item \textit{scatter} array from master to subarrays
  \item count 3s
  \item \textit{reduce} subresults to master
\end{itemize}
public static int countPrimesPar(int[] data, String[] args) {
    final int myRank = MPI.COMM_WORLD.Rank();
    final int NBR_PROCESSES = MPI.COMM_WORLD.Size();
    final int NBR_ELEMENTS_PER_PROCESS = data.length/NBR_PROCESSES;
    final int NBR_REST_ELEMENTS = data.length%NBR_PROCESSES; // modulo.

    int[] process_data = new int[NBR_ELEMENTS_PER_PROCESS]; // send buffer cannot be reused in this MPI implementation...

    // scatter
    MPI.COMM_WORLD.Scatter(data, NBR_REST_ELEMENTS, process_data.length, MPI.INT, process_data, 0, process_data.length, MPI.INT, 0);

    // count primes
    int s_np = 0;

    for (int value: process_data)
        if (isPrime(value))
            s_np++;

    int[] send_buffer = new int[s_np];
    int[] recv_buffer = new int[1];

    // reduce
    MPI.COMM_WORLD.Reduce(send_buffer, 0, recv_buffer, 0, 1, MPI.INT, MPI.SUM, 0);

    return recv_buffer[0];
}
Optimization of Collective operations

broadcast

shift

star

ring
MPI Collective Operations

- **Barrier synchronization in MPI:**
  
  ```c
  int MPI_Barrier(MPI_Comm comm)
  ```

- **The one-to-all broadcast operation is:**
  
  ```c
  int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)
  ```

- **The all-to-one reduction operation is:**
  
  ```c
  int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int target, MPI_Comm comm)
  ```
MPI Collective Operations

- **Broadcast**: Data sent from one process to all others.
- **Allgather**: Collect data from all processes.
- **Scatter**: Data distributed to multiple processes.
- **Alltoall**: Bi-directional data exchange between all processes.
with computations
# Predefined Reduction Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bit-wise AND</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bit-wise OR</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical XOR</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bit-wise XOR</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max-min value-location</td>
<td>Data-pairs</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min-min value-location</td>
<td>Data-pairs</td>
</tr>
</tbody>
</table>
Maximum + location

- **MPI_MAXLOC** returns the pair \((v, l)\) such that \(v\) is the maximum among all \(v_i\)'s and \(l\) is the corresponding \(l_i\) (if there are more than one, it is the smallest among all these \(l_i\)'s).

- **MPI_MINLOC** does the same, except for minimum value of \(v_i\).

An example use of the **MPI_MINLOC** and **MPI_MAXLOC** operators.

<table>
<thead>
<tr>
<th>Value</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

\[\text{MinLoc}(\text{Value}, \text{Process}) = (11, 2)\]
\[\text{MaxLoc}(\text{Value}, \text{Process}) = (17, 1)\]
Scan operation

*Parallel prefix sum*: every node got sum of previous nodes + itself
Overview

1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes
Interconnection Networks

- Interconnection networks carry data between processors and memory.
- Interconnects are made of switches and links (wires, fiber).
- Interconnects are classified as *static* or *dynamic*.
  - Static networks consist of point-to-point communication links among processing nodes and are also referred to as *direct* networks.
  - Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as *indirect* networks.
Static and Dynamic Interconnection Networks

Static network

Indirect network

Network interface/switch

Processing node

Switching element
Important characteristics

- **Performance**
  - Depends on application:

- **Cost**

- **Difficulty to implement**

- **Scalability**
  - Can processors be added with the same cost
Overview

1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes
Network Topologies: Completely Connected and Star Connected Networks

(a) A completely-connected network of eight nodes; 
(b) a star connected network of nine nodes.
Completely Connected Network

- Each processor is connected to every other processor.
- The number of links in the network scales as $O(p^2)$.
- While the performance scales very well, the hardware complexity is not realizable for large values of $p$.
- In this sense, these networks are static counterparts of crossbars (see later).
Star Connected Network

- Every node is connected only to a common node at the center.
- Distance between any pair of nodes is $O(1)$. However, the central node becomes a bottleneck.
- In this sense, star connected networks are static counterparts of buses.
Linear Arrays

(a) with no wraparound links; (b) with wraparound link.

Linear arrays: (a) with no wraparound links; (b) with wraparound link.
Network Topologies: Two- and Three Dimensional Meshes

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.
Network Topologies: Linear Arrays, Meshes, and $k$-$d$ Meshes

In a \textit{linear array}, each node has two neighbors, one to its left and one to its right. If the nodes at either end are connected, we refer to it as a \textit{1D torus or a ring}.

\textbf{Mesh}: generalization to 2 dimensions has nodes with 4 neighbors, to the north, south, east, and west.

A further generalization to $d$ dimensions has nodes with $2d$ neighbors.

A special case of a $d$-dimensional mesh is a \textit{hypercube}. Here, $d = \log p$, where $p$ is the total number of nodes.
Hypercubes and torus

Construction of hypercubes from hypercubes of lower dimension.

Torus (2D wraparound mesh).
Super computer: BlueGene/L

- IBM 2007
- 65,536 dual core nodes
  - E.g. one processor dedicated to communication, other to computation
- Each 512 MB RAM
- Eight in Top 500 Supercomputer list (2010)
  - www.top500.org

<table>
<thead>
<tr>
<th>4 MB</th>
<th>Shared L3 cache/mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 prefetch buffer</td>
<td>Snoop</td>
</tr>
<tr>
<td>L1-I</td>
<td>L1-D</td>
</tr>
<tr>
<td>PowerPC</td>
<td>770 MHz</td>
</tr>
<tr>
<td>Double Hummer FPU</td>
<td></td>
</tr>
<tr>
<td>Torus interconnect</td>
<td>Collective interconnect</td>
</tr>
</tbody>
</table>
BlueGene/L communication networks

(a) 3D torus (64x32x32) for standard interprocessor data transfer
   • Cut-through routing (see later)

(b) collective network for fast evaluation of reductions.

(c) Barrier network by a common wire
Network Topologies: Tree-Based Networks

Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.
Tree Properties

- \( p = 2^d - 1 \) with \( d \) depth of tree
- The distance between any two nodes is no more than \( 2 \log p \).
- Links higher up the tree potentially carry more traffic than those at the lower levels.
- For this reason, a variant called a fat-tree, fattens the links as we go up the tree.
- Trees can be laid out in 2D with no wire crossings. This is an attractive property of trees.
Network Topologies: Fat Trees

A fat tree network of 16 processing nodes.
Network Properties

- **Diameter**: The distance between the farthest two nodes in the network.

- **Bisection Width**: The minimum number of links you must cut to divide the network into two equal parts.

- **Arc connectivity**: minimal number of links you must cut to isolate two nodes from each other. A measure of the multiplicity of paths between any two nodes.

- **Cost**: The number of links. Is a meaningful measure of the cost.
  
  However, a number of other factors, such as the ability to layout the network, the length of wires, etc., also factor into the cost.
# Static Network Properties

<table>
<thead>
<tr>
<th>Network</th>
<th>Diameter</th>
<th>Bisection Width</th>
<th>Arc Connectivity</th>
<th>Cost (No. of links)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completely-connected</td>
<td>1</td>
<td>$p^2/4$</td>
<td>$p - 1$</td>
<td>$p(p - 1)/2$</td>
</tr>
<tr>
<td>Star</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>$p - 1$</td>
</tr>
<tr>
<td>Complete binary tree</td>
<td>$2 \log((p + 1)/2)$</td>
<td>1</td>
<td>1</td>
<td>$p - 1$</td>
</tr>
<tr>
<td>Linear array</td>
<td>$p - 1$</td>
<td>1</td>
<td>1</td>
<td>$p - 1$</td>
</tr>
<tr>
<td>2-D mesh, no wraparound</td>
<td>$2(\sqrt{p} - 1)$</td>
<td>$\sqrt{p}$</td>
<td>2</td>
<td>$2(p - \sqrt{p})$</td>
</tr>
<tr>
<td>2-D wraparound mesh</td>
<td>$2[\sqrt{p}/2]$</td>
<td>$2\sqrt{p}$</td>
<td>4</td>
<td>$2p$</td>
</tr>
<tr>
<td>Hypercube</td>
<td>$\log p$</td>
<td>$p/2$</td>
<td>$\log p$</td>
<td>$(p \log p)/2$</td>
</tr>
<tr>
<td>Wraparound $k$-ary $d$-cube</td>
<td>$d[ k/2 ]$</td>
<td>$2k^{d-1}$</td>
<td>$2d$</td>
<td>$dp$</td>
</tr>
</tbody>
</table>
The total time to transfer a message over a network comprises of the following:

- **Startup time** ($t_s$): Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).

- **Per-hop time** ($t_h$): This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.

- **Per-word transfer time** ($t_w$): This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.
Routing Techniques

Passing a message from node $P_0$ to $P_3$:
(a) a **store-and-forward** communication network;
(b) and (c) extending the concept to **cut-through routing**. The shaded regions: message is in transit. The startup time of message transfer is assumed to be zero.
Store-and-Forward Routing

- A message traversing multiple hops is completely received at an intermediate hop before being forwarded to the next hop.

- The total communication cost for a message of size $m$ words to traverse $l$ communication links is

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

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

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

- Store-and-forward makes poor use of communication resources.
- Packet routing breaks messages into packets and pipelines them through the network.
- Packets may take different paths, thus each packet must carry routing information, error checking, sequencing, ....
- The total communication time for packet routing is approximated by:
  \[ t_{\text{comm}} = t_s + t_h l + t_w m. \]
- The factor \( t_w \) accounts for overheads in packet headers.
Cut-Through Routing

- Takes the concept of packet routing to an extreme by further dividing messages into basic units called *flits (flow control digits)*.
- Since *flits* are typically small, the header information must be minimized.
- This is done by forcing all flits to take the same path, in sequence.
- A tracer message first programs all intermediate routers. All flits then take the same route.
- Error checks are performed on the entire message.
- No sequence numbers are needed.
Cut-Through Routing

- The total communication time for cut-through routing is approximated by:
  \[ t_{comm} = t_s + t_h l + t_w m. \]

- Identical to packet routing, however, \( t_w \) is typically much smaller.

- \( t_h \) is typically smaller than \( t_s \) and \( t_w \). Thus, particularly, when \( m \) is large:
  \[ t_{comm} = t_s + t_w m. \]
Routing a message from node $P_s (010)$ to node $P_d (111)$ in a three-dimensional hypercube using E-cube routing.
A broadcast in a Hypercube

Message from node 0 to all others: $d$ steps

for(int $d$: dimensions)
    if (all bits with index > $d$ are 0)
        if ($d$th bit == 0)
            send message to (flip $d$th bit)
        else
            receive message from (flip $d$th bit)

Reduce operation is the opposite…

Message-passing Parallel Processing
Jan Lemeire
Cost of Communication Operations

- Broadcast on hypercube: $\log p$ steps
  - With cut-through routing: $T_{\text{comm}} = (t_s + t_wm) \log p$

- All-to-all broadcast (full duplex links)
  - Hypercube: $\log p$ steps
  - Linear array: $p-1$ steps
  - Ring: $p/2$ steps
  - 2D-Mesh: $2 \sqrt{p}$ steps

- Scatter and gather: similar to broadcast

- Circular q-shift: send msg to $(i+q) \mod p$
  - Mesh: maximal $\sqrt{p}/2$ steps
  - In a hypercube: embedding a linear array
All-to-all personalized communication on hypercube

(a) Initial distribution of messages

(b) Distribution before the second step

(c) Distribution before the third step

(d) Final distribution of messages
Embedding a Linear Array into a Hypercube

Gray code problem:
arrange nodes in a ring so that neighbors only differ by 1 bit

(a) A three-bit reflected Gray code ring
(b) its embedding into a three-dimensional hypercube.
Application of Gray code

To facilitate **error correction** in digital communications

The problem with **natural binary codes** is that, with real switches, it is very unlikely that switches will change states exactly in synchrony

transition from 001 to 100 might look like 011 — 001 — 101 — 100
Overview

1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes
Dynamic networks: Buses

Bus-based interconnect
A crossbar network uses an $p \times m$ grid of switches to connect $p$ inputs to $m$ outputs in a non-blocking manner.
Crossbars have excellent performance scalability but poor cost scalability.
- The cost of a crossbar of $p$ processors grows as $O(p^2)$.
- This is generally difficult to scale for large values of $p$.

Buses have excellent cost scalability, but poor performance scalability.

Multistage interconnects strike a compromise between these extremes.
Multistage Dynamic Networks

The schematic of a typical multistage interconnection network.
An Omega network is based on $2 \times 2$ switches.

An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.
# Evaluating Dynamic Interconnection Networks

<table>
<thead>
<tr>
<th>Network</th>
<th>Diameter</th>
<th>Bisection Width</th>
<th>Arc Connectivity</th>
<th>Cost (No. of links)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crossbar</td>
<td>$1$</td>
<td>$p$</td>
<td>$1$</td>
<td>$p^2$</td>
</tr>
<tr>
<td>Omega Network</td>
<td>$\log p$</td>
<td>$p/2$</td>
<td>$1$</td>
<td>$p \log p$</td>
</tr>
<tr>
<td>Dynamic Tree</td>
<td>$2 \log p$</td>
<td>$1$</td>
<td>$2$</td>
<td>$p - 1$</td>
</tr>
</tbody>
</table>
Recent trend: networks-on-chip

- Many-cores (such as cell processor)
- Increasing number of cores
  - bus or crossbar switch become infeasible
  - specific network has to be chosen
- When even more cores
  - scalable network required
Memory Latency $\lambda$

Memory Latency = *delay required to make a memory reference*, relative to processor’s local memory latency, $\approx$ unit time $\approx$ one word per instruction

<table>
<thead>
<tr>
<th>Architecture Family</th>
<th>Computer</th>
<th>Lambda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip Multiprocessor*</td>
<td>AMD Opteron</td>
<td>100</td>
</tr>
<tr>
<td>Shared-memory Multiprocessor</td>
<td>Sun Fire E25K</td>
<td>400–660</td>
</tr>
<tr>
<td>Co-processor</td>
<td>Cell</td>
<td>N/A</td>
</tr>
<tr>
<td>Cluster</td>
<td>HP BL6000 w/GbE</td>
<td>4,160–5,120</td>
</tr>
<tr>
<td>Supercomputer</td>
<td>BlueGene/L</td>
<td>8960</td>
</tr>
</tbody>
</table>

*CMP’s $\lambda$ value measures a transfer between L1 data caches on chip.*
Overview

1. Definition
2. MPI
   ♦ Efficient communication
3. Collective Communications
4. Interconnection networks
   ♦ Dynamic networks
   ♦ Static networks
5. End notes
Choose MPI

- Makes the fewest assumptions about the underlying hardware, is the least common denominator. It can execute on any platform.
- Currently the best choice for writing large, long-lived applications.
MPI Issues

- MPI messages incur large overheads for each message
  - Minimize cross-process dependences
  - Combine multiple message into one

- Safety
  - Deadlock & livelock still possible...
    - But easier to deal with since synchronization is explicit
  - Sends and receives should be properly matched
  - Non-blocking and non-buffered messages are more efficient but make additional assumptions that should be enforced by the programmer.
MPI-2: also supports one-sided communication

- process accesses remote memory without interference of the remote ‘owner’ process

- Process specifies all communication parameters, for the sending side and the receiving side
  - exploits an interconnect with RDMA (Remote DMA) facilities

- Additional synchronization calls are needed to assure that communication has completed before the transferred data are locally accessed.
  - User imposes right ordering of memory accesses
One-sided primitives

- **Communication calls**
  - `MPI_Get`: Remote read.
  - `MPI_Put`: Remote write.
  - `MPI_Accumulate`: accumulate content based on predefined operation

- **Initialization**: first, process must create window to give access to remote processes
  - `MPI_Win_create`

- **Synchronization to prevent conflicting accesses**
  - `MPI_Win_fence`: like a barrier
  - `MPI_Win_post, MPI_Win_start, MPI_Win_complete, MPI_Win_wait`: like message-passing
  - `MPI_Win_lock, MPI_Win_unlock`: like multi-threading
Partitioned Global Address Space Languages (PGAS)

Higher-level abstraction: overlay a single address space on the virtual memories of the distributed machines.

Programmers can define global data structures

- Language eliminates details of message passing, all communication calls are generated.
- Programmer must still distinguish between local and non-local data.
Parallel Paradigms

**Shared-memory architecture**
- PThreads
- Direct, uncontrolled memory access
- Protection of critical sections (lock-unlock)

**Distributed-memory architecture**
- MPI
- Controlled remote memory access via messages
- Start and end of ‘transactions’ (post-start-complete-wait)

**Message-passing Parallel Processing**

**Parallel Processing**

- Direct, uncontrolled memory access
- Protection of critical sections (lock-unlock)

**Shared-memory architecture**

**Distributed-memory architecture**

**PGAS**

**Erlang**

**one-sided comm**