Parallel Hardware

Parallel Computing – COSC 3P93

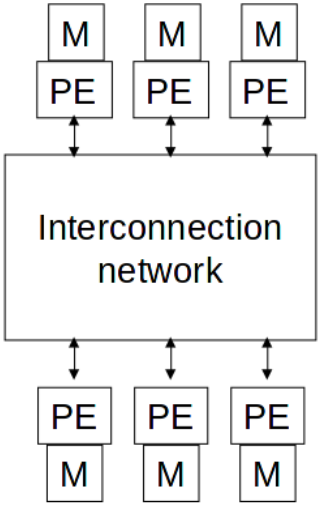
Course Outline

- Introduction
- Parallel Hardware
- Parallel Software
- Performance
- Parallel Programming Models
- Patterns
- SMP with Threads and OpenMP
- Distributed-Memory Programming with MPI
- Algorithms
- Tools
- Parallel Program Design

Today Class

- Parallel Hardware
  - MIMD
    - Distributed Memory
    - Interconnection networks
    - Cache coherence
    - Shared-memory versus distributed-memory

Distributed-memory System



- Multiple computers/nodes
- Message-passing network
- Local memories are private with their own program and data
- **No memory contention**
  - The number of processors is very large
- The processors are connected by
  - **Communication lines**
  - The precise way in which the lines are connected is called the **topology** of the multicomputer
- A typical program consists of subtasks residing in all the memories

Distributed-memory Systems

- **Clusters**
  - Collection of commodity systems → PCs
  - Interconnected by a commodity network → Ethernet
- **Nodes** of a cluster
  - Individual computational units
    - Containing shared-memory systems with multiple multicore processors
    - Joined together by the communication network
- Clusters are not pure distributed-memory systems
  - They may be called **hybrid systems**
  - Nowadays → a cluster will have shared-memory nodes
- **Grid**
  - Infrastructure to support large networks of geographically distributed computers as a unified distributed-memory system
  - They are **heterogeneous**
    - Individual nodes maybe have different types of hardware

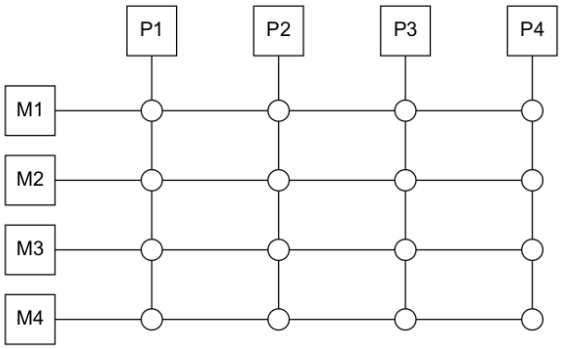
Interconnection Networks

- **Essential** → All communication through them
- Interconnection
  - Great responsible for overall **performance degradation**
- Interconnects have several similarities
  - However → enough particular distinctions to treat them differently
    - **Shared-memory interconnects**
    - **Distributed-memory interconnects**

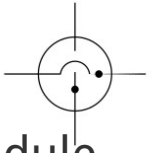
Shared-memory Interconnects

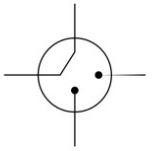
- Widely used interconnects → bus and crossbar
  - **Bus** → parallel communication wires
    - Shared among devices
    - Low cost and flexibility
      - Multiple devices connected at low cost
    - However
      - More connected devices, higher **contention**
        - Lower performance
      - Not good for large-scale shared-memory systems
        - Processors frequently waiting to access the bus

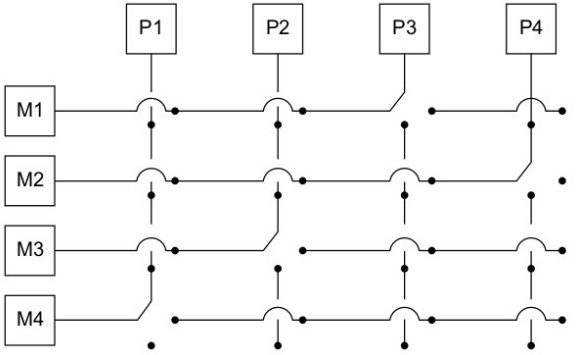
Shared-memory Interco



- Widely used interconnects → bus and crossbar
  - **Crossbar**
    - Bidirectional comm. links, cores, memory modules, and **switches**
      - **Switched** interconnect → routing of data among devices
    -
    -
    - Simultaneous comm. among different devices
    - Conflicts of two processors accessing the same mem. module
    - Downside → high cost
    - Example
      - P1 writes to M4
      - P2 reads from M3
      - P3 reads M1
      - P4 writes to M2



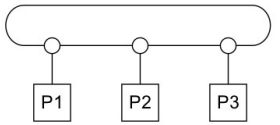




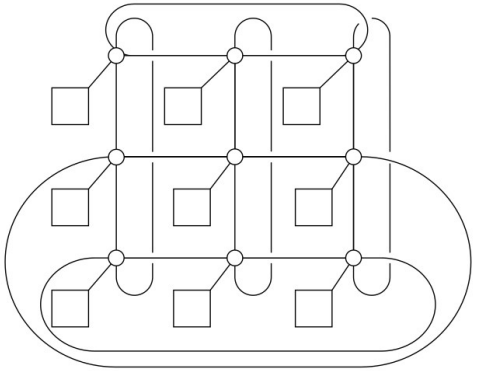
Distributed-memory Interconnects

- Directly related to Interconnection Network
- Delimiting factors
  - **Mode of Operation**
    - Synchronous vs. Asynchronous
  - **Control Strategy**
    - Centralized vs. Decentralized
  - **Switching Techniques**
    - Packet switching vs. Circuit switching
  - **Topology**
    - Static vs. Dynamic

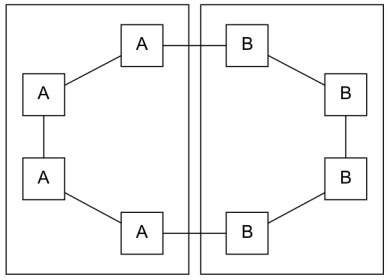
Distributed-memory Interconnects

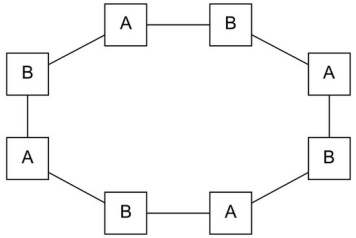


- Two groups → **indirect** and **direct interconnects**
  - **Direct interconnect**
    - Directly connected to a processor-memory pair
    - Switches connected to each other
    - Examples (p = # of processors)
      - Ring → 2p links
      - 2D toroidal mesh → 3p links



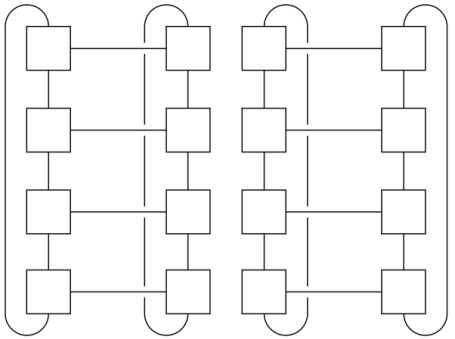
    - **Connectivity** factor → # of simultaneous communications
      - **Bisection width**
        - Splitting the system into two equal halves (#nodes)
        - Ex → split the ring topology
          - Two ways of splitting
            - One gives two communications
            - Another gives four communications
              - Incorrect





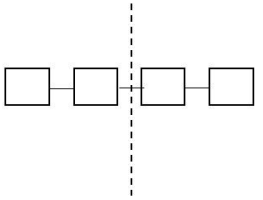
Bisection Width

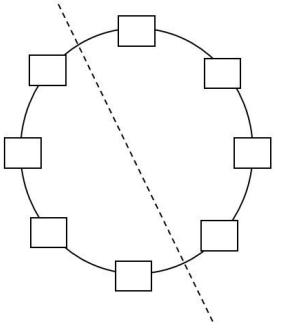
- It is supposed to give the worst case estimate
  - Previous example → bisection width of ring topology is 2
- **Alternative way of computing bisection width**
  - Setting the minimum number of links needed to be removed to splitting the topology in two equal halves
    - # of links equals to the bisection width
  - Ex → square 2D toroidal mesh → p = $\{q^2 \mid q \text{ is even}\}$
    - Removing the middle horizontal links and respec. Wraparounds
      - Bisection width => 2q = 2.sqrt(p)

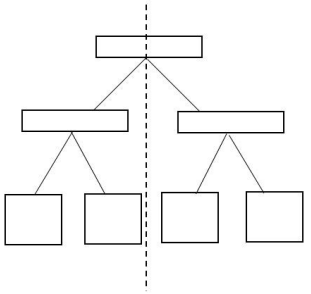


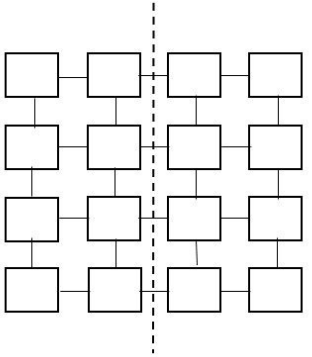
Bisection Width

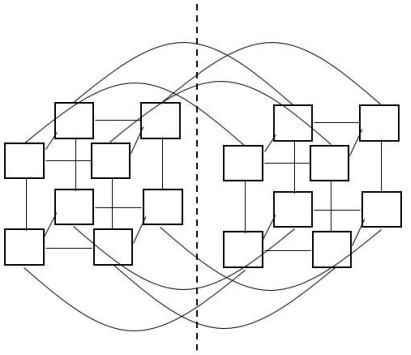
- Linear array
  - 1 link
- Ring topology
  - 2 links
- Tree topology
  - 1 link
- Mesh topology
  - sqrt(n) links
- Hyper-cube
  - n/2 links





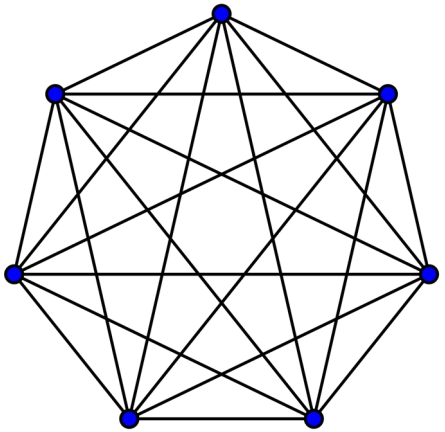




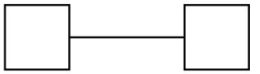


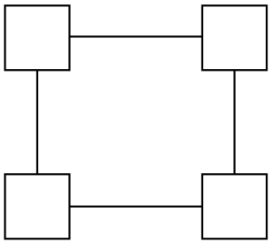
Bandwidth

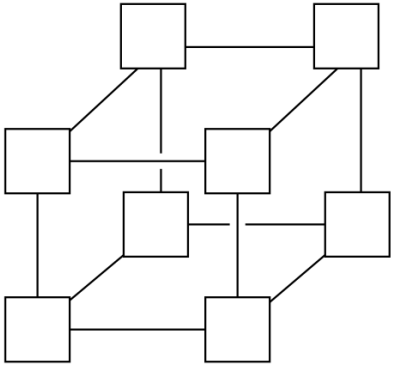
- Data transmission rate → Mbits or Mbytes per second
- **Bisection bandwidth** → network quality metric
  - Not just the number of links
    - Sum of bandwidths
- Ideal direct interconnect → **fully connected network**
  - Each switch directly connected with every other switch
  - Bisection width → $p^2/4$
  - However
    - Impractical build → $p^2/2 + p/2$
    - But → theoretical best build



Hypercube





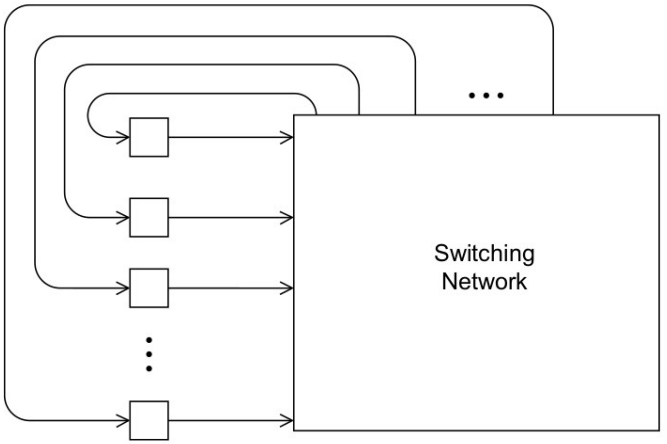


- Direct interconnected
  - Highly connected
- 1D hypercube
  - Fully-connected system
- 2D hypercube
  - 2 1D hypercubes “connected”
- 3D hypercube
  - 2 2D hypercubes “connected”
- Hypercube of dimension d
  - $p = 2^d$
  - Bisection width = p/2
  - Each switch supports 1+d = 1 $\log_2 p$ wires

- http://www.math.union.edu/~dpvc/math/4d/models/welcome.html
- http://hypersolid.milosz.ca/

Indirect Interconnects

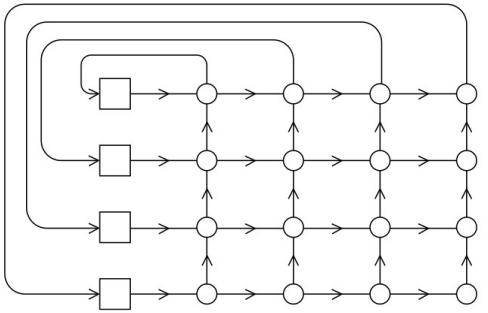
- **Alternative to direct connects**
- Switches not directly connected to processors
  - Each has an incoming, an outgoing link, and a switching network



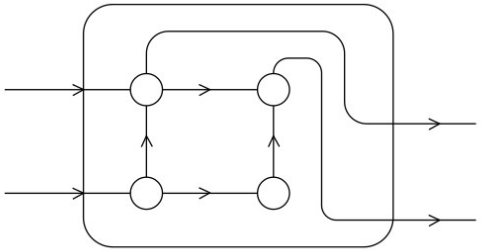
- More difficult to define bisection widt
  - Same principle
    - Dividing the system in two halves
    - The minimum number of links removed so that the two halves do not communicate

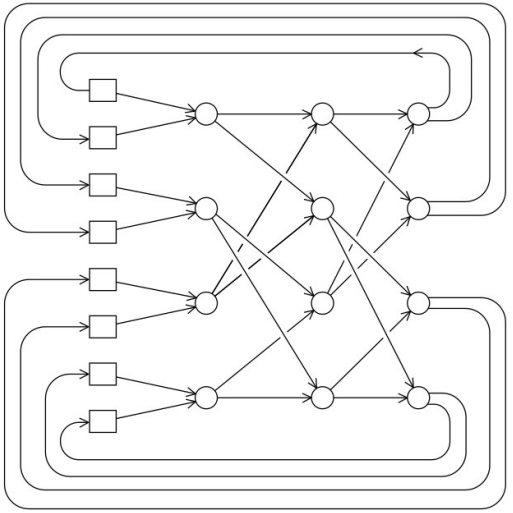
Indirect Interconnects Examples

- Crossbar and omega network → for distributed memory
  - **Crossbar** with unidirectional links
    - No 2 procs. attempting to comm. with same proc.
      - Support to simultaneous communication
    - p^2 switches



  - **Omega** network
    - Two-by-two crossbars
    - Less simultaneous communications
    - But, less expensive
    - $2p\log_2 p$ switches





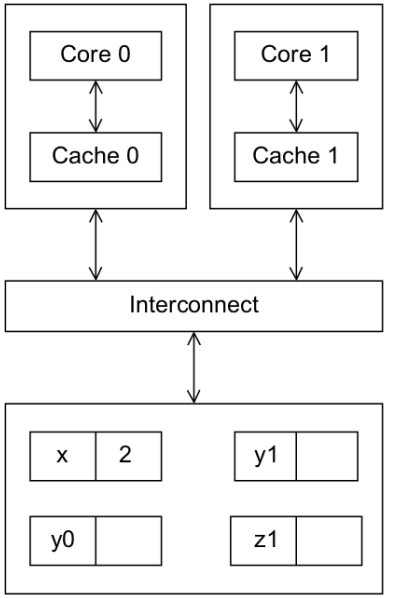
Latency and Bandwidth

- Time it takes for data to reach its destination
  - Data transmission between
    - Main memory and cache
    - Cache and register
    - HD and memory
    - Between nodes
- **Latency**
  - Elapsing time between beginning of data transmission at source and beginning data receipt at destination → 1st byte
- **Bandwidth**
  - Rate at wich data is received at the destination after the 1st byte is received
- Thus → **message transmission time**
  - l = n / b
    - l = latency in seconds
    - b = bandwidth in bytes per sec
    - n = message size in bytes

Latency and Bandwidth

- These terms may be used differently
  - Latency
    - Total message transmission time
    - Or any overhead related to data transmission
    - In other contexts
      - It is not just the time to transmit raw data → it includes
        - Packet headers (encapsulation), CRCs, retransmissions
        - Time to assemble the message and unwrap to the other end
        - Time to disassemble packets and recombine fragmented frames

Cache Coherence



- CPU caches managed by system hardware
  - Important consequences
  - Example
    - Shared-memory system with 2 cores → private data cache each
    - Everything fine while just reading shared data
    - Incoherence happens when modifying shared data → x = 7
      - Core 1 may have kept previous value of x in its local cache
        - Unless variable was evicted
      - Not guaranteed that z1 will be 28 at time 2 in Core 1

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

Cache Coherence

- Unpredictability regardless of data caching write-through or write-back policies
  - Write-through policy
    - Main memory will be updated with x = 7
    - However, no effect on the local cache of Core 1
  - Write-back policy
    - X = 7 in Core 0 will not even make to main memory and be available to Core 1 when z1 is updated
- Clearly, a major problem → lack of control
  - Apparently correct statements may lead to inconsistency
  - No mechanism for ensuring caching of multiple processors
  - **Cache coherence** problem
    - Two approaches
      - **Snooping cache coherence**
      - **Directory-based cache coherence**

Snooping Cache Coherence

- Approach from bus-based systems
  - Any signal transmitted on the bus is seen by all cores connected to it
- Example
  - Core 0 updating shared variable → broadcasts it
    - Core 1 snoops the bus and catches the updates
- Actually
  - The snooping protocol broadcasts the cache line containing x and not x
- Characteristics
  - Cores do not need to be interconnected by a bus
  - It works fine with write-through and write-back
    - Just additional steps for write-back

Directory-based Cache Coherence

- Problem → Broadcasting is not always possible
  - Large-scale systems
- Use of a **directory**
  - Distributed data structure
  - Storing the status of each cache line
- Characteristics
  - Substantial additional storage needed
  - Updates restrict to contacting only related cores
- Example
  - Core 0 reads a line into its local cache
  - Directory is updated → core 0 has a copy
  - If variable is updated → directory is consulted
    - Cache controllers of the cores containing copies of the line are invalidated

False Sharing

- Reminder → CPU caches implemented in hardware
  - Operating over lines and not variables
    - Consequences on performance
- Example → iterative calls to f(i,j)
  - Code can be parallelized by # of cores (core_count)

```
int i , j , m , n ;
double y [ m ];
/* Assign y = 0 */
. . .
for ( i = 0; i < m ; i ++)
    for ( j = 0; j < n ; j ++)
        y [ i ] += f ( i , j );
```

```
/* Private variables */
int i, j, iter_count;
/* Shared variables initialized by one core */
int m, n, core_count;
double y [ m ];
iter_count = m / core_count

/* Core 0 does this */
for ( i = 0; i < iter_count; i ++)
    for ( j = 0; j < n; j ++)
        y [ i ] += f ( i, j );

/* Core 1 does this */
for ( i = iter_count+1; i < 2*iter_count; i ++)
    for ( j = 0; j < n; j++)
        y [ i ] += f ( i, j );
. . .
```

False Sharing

- Assume
  - m = 8, a large n...
  - Two cores → 0 and 1
  - 8-byte doubles → y occuping whole cache line
  - And a directory-based coherence
- What would happen if both cores run simultaneously?
  - Every time they execute y[i] = f(i, j)
    - The whole cache line will be invalidated → update
      - Next time statement is run, system will have to fetch whole line from memory
      - However → core 0 and 1 never access each other's elements of y
- This condition is **false sharing**
  - It does not affect coherence
  - It affects **performance**
- How to solve (partially) the problem?
  - The use of temporary storage

Shared-memory Versus Distributed-memory

- Why so many MIMD systems being shared-memory?
  - Most programmers find the implicit coordination of processors through shared data structure more appealing than explicitly sending messages
- **Major disadvantages of shared-memory**
  - **Cost** of scaling the interconnect
    - Large crossbars are expensive
    - Dramatic increase of conflicts as scale grows
      - They are suitable for systems with a few processors
  - Distributed-memory interconnects “inexpensive”
    - Hypercube and toroidal mesh
    - Better suited for systems with vast amounts of data and computation
      - Many thousand-processor systems been already built

Class Recap

- Parallel Hardware
  - MIMD
    - Distributed Memory
    - Interconnection networks
    - Cache coherence
    - Shared-memory versus distributed-memory

Next Class

- Parallel Software
  - Caveats
  - Coordinating the processes/threads
  - Shared-memory
  - Distributed-memory
  - Programming hybrid systems