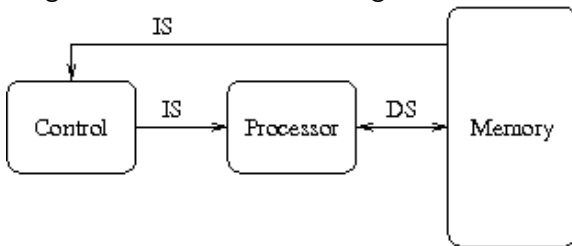


# Concurrent Architectures

Architectures can be classified based on multiplicity of instruction and data streams (Flynn's taxonomy):

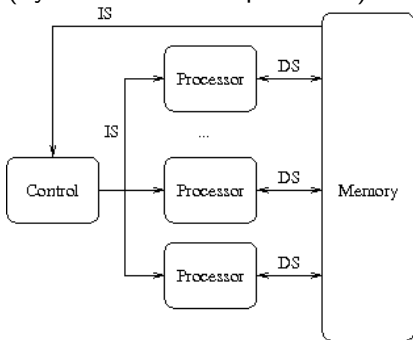
- Single Instruction Stream, Single Data Stream (SISD)



- Serial processing

# Concurrent Architectures (cont'd)

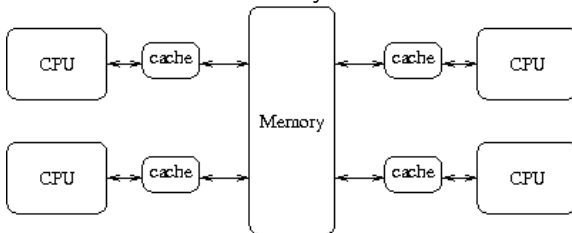
- Single Instruction Stream, Multiple Data Stream (SIMD)  
(Synchronous Multiprocessor)



- All processors execute same instruction.
- Well suited to data-parallel algorithms (e.g., Array operations, DSP)

# Concurrent Architectures (cont'd)

- MIMD Multi-Processor System

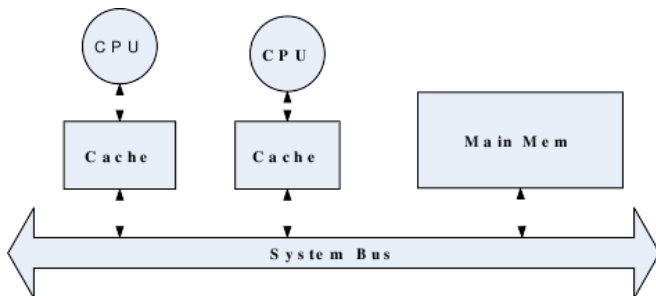


- Can use general purpose CPU.
- More complicated inter-processor communication.
- Processors communicate for synchronization.
- General purpose.

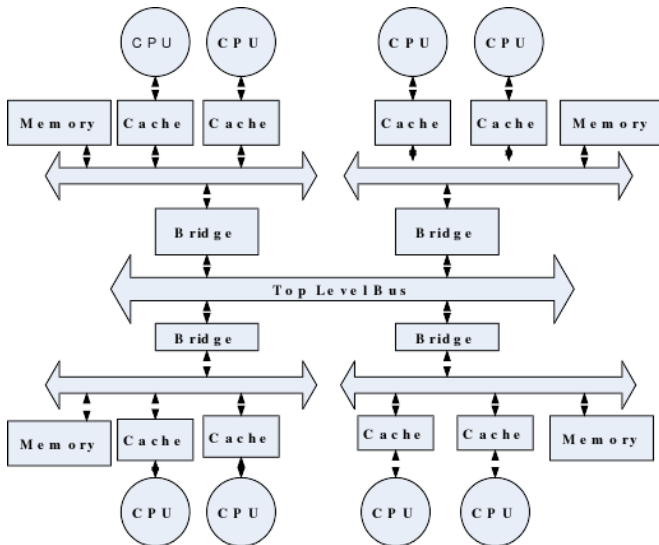
# Memory Architectures for MIMD 1: Shared Memory

- All processors 'see' the same address space.
- Physically memory may be shared or distributed.
- More flexibility in programming (message passing can be emulated).
- **Uniform (symmetric) memory access (UMA):**
  - Bus or crossbar connection.
  - Good for system with small number of processors ( $< 30$ ).
- **Non-uniform memory access (NUMA):**
  - Each processor has quicker access to some memory than others.
  - Tree-structured interconnection.
  - Reduces congestion in interconnection network.

## UMA



# NUMA (Example)



# Memory Architectures for MIMD 2: Distributed Memory

(also called message passing, multicomputers)

- Each processor has private memory.
- Communication by message passing.

**Multicomputer:** Distributed-memory multiprocessor with all processors and memory co-located.

- also called a *tightly coupled machine*
- requires specialized interconnect for message passing
- Example: Transputer.

**Cluster:** Connected by LAN or WAN.

- Generic hardware.
- *Network of workstations (NOW), Cluster of workstations (COW) Beowulf Cluster.*

# Memory Architectures for MIMD 3: Distributed Shared Memory

- Emulates Shared Memory on a Distributed Memory hardware.
- Shared memory is implemented in software by OS or by a layer above the OS.
- Remote access is via messages sent over a network (e.g. an ether net)
- Sharing may be of
  - Pages — OS must be complicit
  - Named variables — system calls to read and write
  - Objects — operations are programmer defined.



# Multiprocessors and Distributed Shared Memory Machines

(after Tanenbaum, Distributed Operating Systems)

Shared Virtual Addr. Space?  
Remote access in hardware?  
Ops converted to message by?  
Transfer Medium  
Operations  
Migration done by  
Transfer unit

| Multiprocessor |      | DSM        |             |                 |
|----------------|------|------------|-------------|-----------------|
| UMA            | NUMA | Page Based | Shared Vars | Shared Object   |
| ✓              | ✓    | ✓          |             |                 |
| ✓              | ✓    | No         |             |                 |
| MMU            | MMU  | OS         | OS          | Run time system |
| Bus            | Bus  | Network    | Network     | Network         |
| R/W            | R/W  | R/W        | R/W         | General         |
| HW             | SW   | SW         | SW          | SW              |
| Cache block    | Word | Page       | Variable    | Object          |

# Atomic Actions

- In a shared-memory multiprocessor (even with a single time-shared processor), the usual rules of programming logic are not reliable.
- Consider two updates to the same variable executed by two processors at about the same time:

$$\frac{\mathbf{P1}}{x := x + 1} \quad \frac{\mathbf{P2}}{x := x + 1}$$

# Atomic Actions (cont'd)

- Two things could happen:

| P1         | P2         |    | P1         | P2         |
|------------|------------|----|------------|------------|
| LOAD x r1  |            |    | LOAD x r1  |            |
|            | LOAD x r1  |    | ADD r1 #1  |            |
| ADD r1 #1  |            | or | STORE r1 x |            |
|            | ADD r1 #1  |    |            | LOAD x r1  |
| STORE r1 x |            |    |            | ADD r1 #1  |
|            | STORE r1 x |    |            | STORE r1 x |

- By the normal rules of programming  $x$  should be increased by 2.
- We write  $\langle S \rangle$  to mean that the statement  $S$  is executed (as if) without interruption.

# Memory Consistency

For efficiency, local copies of memory must be made.

- In UMAs and NUMAs this is in Caches.
- In DSM machines, 1 page may be replicated in several frames.

Consider a multiprocessor using standard write-back caches.

|                 |           |           |  |                    |           |
|-----------------|-----------|-----------|--|--------------------|-----------|
| $x, y := 0, 0;$ | <b>co</b> | # P0:     |  | # P1:              | <b>oc</b> |
|                 |           | $x := 1;$ |  | $\text{print } y;$ |           |
|                 |           | $y := 2;$ |  | $\text{print } x;$ |           |

Suppose the following sequence of actions:

P0 writes 1 to its cached  $x$   
 P0 writes 2 to its cached  $y$   
 P0's cache writes 2 to global  $y$   
 P1 executes, printing 2, 0

# Consistency Models 1: Strict consistency

- A *consistency model* specifies what guarantees the hardware (or OS or run-time system) makes to the software about the apparent ordering of operations.
- **Strict consistency:** *Every read returns the value of the most recent write.*
  - Implicit in this defn is the assumption of a global time so that the “most recent” is well defined.
  - This model can be achieved using synchronous hardware and a global clock.

# Resolution

Resolution of concurrent read and writes must be addressed

- CREW — Concurrent reads are allowed. Software must ensure concurrent writes do not happen.
- CRCW — Concurrent writes are allowed. Resolution of conflicting writes can be:
  - *common* — All processors must write same value
  - *arbitrary* — Any arbitrary choice is made.
  - *priority* — Predictable choice is made.

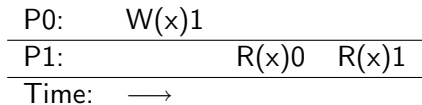
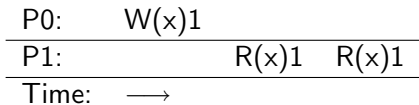
Why not to implement strict consistency?

- All processors must be informed of all writes. Takes time & bandwidth.

## Consistency Models 2: Sequential consistency

*Each process sees its own actions in process order; and there exists an interleaving of actions consistent with every process's view.*

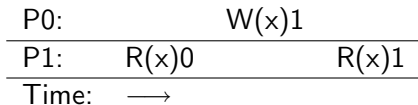
For example. Assume  $x$  is initially 0.



Strict consistency does not allow behaviour on the right.

Sequential consistency allows either behaviour.

The behaviour on the right is consistent with an interleaving



# Sequential consistency example

(Example after Tanenbaum Distributed OSs)

$a, b, c := 0, 0, 0;$

|           |                |  |                |  |                |           |
|-----------|----------------|--|----------------|--|----------------|-----------|
| <b>co</b> | # P0:          |  | # P1:          |  | # P2:          | <b>oc</b> |
|           | $a := 1;$      |  | $b := 1;$      |  | $c := 1;$      |           |
|           | $x := 2b + c;$ |  | $y := 2c + a;$ |  | $z := 2a + b;$ |           |

Many possible values for  $x, y, z$ . For example 0, 3, 3

|       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|
| P0:   | W(a)1 | R(b)0 | R(c)0 |       |       |       |
| P1:   |       |       | W(b)1 | R(c)1 | R(a)1 |       |
| P1    |       |       |       | W(c)1 | R(a)1 | R(b)1 |
| Time: | →     |       |       |       |       |       |

But  $x, y, z \neq 2, 2, 2$ . There is no interleaving that gives 2,2,2.

There are various weaker models of consistency that allow simpler (and faster) implementation.

See Tanenbaum, Distributed Operating Systems, for more.



# Cache coherence

- Ensures that *if any cache contains a modified line, then no two caches disagree as to its value.*
- Stronger consistency model than sequential consistency, but weaker than strict consistency

**Example:** MESI protocol for UMAs. Every cache sees every bus transaction (Snooping).

Each cache line is in one of 4 states

- **Modified.** Line is not consistent with memory. No other cache has the line.
- **Exclusive.** Line is consistent with memory. No other cache has the line.
- **Shared.** Line is consistent with memory. Some other cache has the line.
- **Invalid.** Line is not valid. (Line is not consistent with memory and some other cache has the line.)

# MESI (cont'd)

Bus transactions:

- **BusRd**. Request value put on bus.
- **BusWr**. Write line back to main memory.
- **BusRdX**. Read with intent to write.

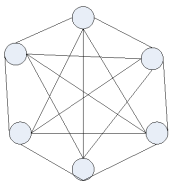
E.g.

- When a processor does a write and the line's state is not already **Modified**, its cache initiates a **BusRdX** and changes the state to **Modified**.
- When a cache sees a **BusRdX** on the bus (and has the line) it changes the state to **Invalid**, while (possibly) flushing the value of the line onto the bus.

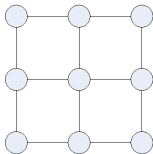
The sequence of bus transactions imposes a single order on what the processors see. Hence sequential consistency.

# Network Topologies

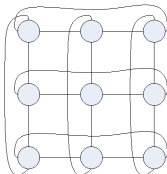
Physically, each processor can only connect to a limited number of other processors.



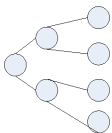
Fully connected



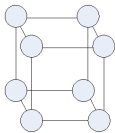
2D grid



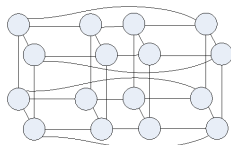
2D Toroidal Grid



Tree ( $b=2$ )



Hyper Cube  
( $n=8$ )



Hyper Cube ( $n=16$ )

Others are also possible.

# Network Topologies: Comparison

- $n$  is # of processors
- diameter is the max # hops between nodes.

|                  | Connections per node |         |            | Diameter                         |         |            |
|------------------|----------------------|---------|------------|----------------------------------|---------|------------|
|                  | form                 | $n = 8$ | $n = 4096$ | form                             | $n = 8$ | $n = 4096$ |
| Fully connected  | $n - 1$              | 7       | 4095       | 1                                | 1       | 1          |
| 2D Toroidal Grid | 4                    | 4       | 4          | $\simeq \sqrt{n}$                | 3       | 64         |
| 3D Toroidal Grid | 6                    | 6       | 6          | $\simeq \frac{3}{2} \sqrt[3]{n}$ | 2       | 24         |
| Tree             | $b + 1$              | 3       | 3          | $\simeq 2 \log_b n$              | 5       | 23         |
| Hyper-Cube       | $\lg n$              | 3       | 12         | $\lg n$                          | 3       | 12         |

- Trees have a bottleneck at the root, whereas hyper-cubes avoid bottlenecks.
- Consider the number of links that must be deleted to partition the network.

# Broad Application Classes

## Multithreaded Systems

- Divide overall (set of) problem(s) into (mostly) independent tasks — makes programming less complicated.
- Usually shared memory.
- Examples: Web-browser: One thread handles GUI, while “worker threads” obtain data from network, format displayed data etc. Word-Processor: “worker threads” handle printing spell checks.

## Distributed Systems

- Data or application is physically distributed.

## Parallel Computations

- Solve bigger problems faster by using more than one processor.
- *Data parallel* — each process does the same thing on part of the data.
- *Task parallel* — different processes carry out different tasks.

# Programming Patterns

- Iterative Parallelism (data parallel)
  - Multiple loop iterations executed in parallel
- Recursive parallelism (data parallel)
  - Recursive subroutine calls executed in parallel
- Producers and Consumers (task or data parallel)
  - One process feeds output to the next
- Client/Server (task parallel)
  - Clients make requests, servers respond.
- Peers
  - Similar processes communicate directly to each other.

# Iterative Parallelism

- Execute iterations of loops in parallel
- Typical for scientific computations.

## Example: Matrix Multiplication

Compute  $a := b \times c$ , where  $a$ ,  $b$  and  $c$  are  $n$  by  $n$  matrices. ( $n^2$  inner products)

```
double a[n,n], b[n,n], c[n,n];
```

Sequential version:

```
for i := 0 to n - 1 do
  for j := 0 to n - 1 do
    c[i,j] := 0.0
    for k := 0 to n - 1 do
      c[i,j] := c[i,j] + a[i,k] * b[k,j]
    end for
  end for
end for
```

## Aside: Independence

*read set* — the set of variables that an operation reads but does not modify.

*write set* — the set of variables that an operation modifies (may also read).

Operations can be executed in parallel if they are *independent*.  
Not safe (in general) if both write, or one writes and the other reads

Processes *a* and *b* are *independent* iff

$$W_a \cap (R_b \cup W_b) = \emptyset \wedge W_b \cap (R_a \cup W_a) = \emptyset$$



# Parallel Matrix Multiplication

In the matrix multiplication algorithm each of the  $n^2$  iterations of the dot product computation is independent of all the others. So:

```
co  $i := 0$  to  $n - 1$ 
```

▷ All rows

```
  co  $j := 0$  to  $n - 1$ 
```

▷ All columns

```
     $c[i, j] := 0.0$ 
```

```
    for  $k := 0$  to  $n - 1$  do
```

```
       $c[i, j] := c[i, j] + a[i, k] * b[k, j]$ 
```

```
    end for
```

```
  oc
```

```
oc
```

But if there are less than  $n^2$  processors then the above is wasteful. Having more processes than processors will slow down computation.

# Iterative Parallel Matrix Multiplication

If the number of processors  $P$  is less than or equal to  $n$ , we can divide the work among  $P$  processes thus

```
process worker [ $w = 0$  to  $P - 1$ ]  
  int first :=  $\lceil (w \times n) \div P \rceil$            ▷ first row of strip  
  int last :=  $\lceil ((w + 1) \times n) \div P \rceil - 1$    ▷ last row of strip  
  for  $i :=$  first to last do  
    for  $j := 0$  to  $n - 1$  do  
       $c[i, j] := 0.0$   
      for  $k := 0$  to  $n - 1$  do  
         $c[i, j] := c[i, j] + a[i, k] * b[k, j]$   
      end for  
    end for  
  end for  
end process
```

# Recursive Parallelism

Independent recursive procedures:

When a sequence of calls (recursive or not) are independent, they can run in parallel.

## Example: Adaptive Quadrature

Estimate the area under a curve,  $f(x)$ , on an interval  $[left, right]$ .

```
function double QUAD(double left, right, fleft, fright, area)  
  double mid := (left + right)/2  
  double fmid := f(mid)  
  double larea := (fleft + fmid) * (mid - left)/2  
  double rarea := (fmid + fright) * (right - mid)/2  
  if (|larea + rarea| - area) >  $\epsilon$  then  
    larea := QUAD(left, mid, fleft, fmid, larea)  
    rarea := QUAD(mid, right, fmid, fright, rarea)  
  end if  
  return larea + rarea  
end function
```

# Recursive Parallel Adaptive Quadrature

Since recursive calls only use local variables and value parameters, we can do them in parallel.

```
function double QUAD(double left, right, fleft, fright, area)
  double mid := (left + right)/2
  double fmid := f(mid)
  double larea := (fleft + fmid) * (mid - left)/2
  double rarea := (fmid + fright) * (right - mid)/2
  if (|larea + rarea| - area) >  $\epsilon$  then
    co
      larea := QUAD(left, mid, fleft, fmid, larea)
    ||
      rarea := QUAD(mid, right, fmid, fright, rarea)
    oc
  end if
  return larea + rarea
end function
```

# Producers and Consumers (pipelines)

- Processes may act as filters — consuming output from upstream process and producing for downstream.
- Example: Unix pipe.

```
sed -f Script $* | tbl | eqn | groff Macros -
```

Pipe acts as bounded FIFO queue.

# Clients & Servers

- Dominant pattern for distributed systems.
- Distributed analog to procedure call.
- Examples: Remote file systems, http, ftp, telnet.
- Also OS kernels: Kernel is a set of kernel-mode threads that services system calls on behalf of user-level processes.
- Servers may service multiple clients, possibly concurrently.

# Simple multithreaded server pseudocode

```
process server [ $s := 1$  to  $n$ ]  
  while system is not shutdown do  
    await new client  
    loop  
      receive request from client  
      process request  
      send reply  
      if client request quit then  
        break  
      end if  
    end loop  
    clean up  
  end while  
end process
```

# Peers

Similar distributed processes cooperate to accomplish a task.

## Example: Distributed Matrix Multiplication

Assume an  $n$  by  $n$  matrix and  $n$  distributed workers.

```
process worker [i := 0 to n - 1]
  double a[n]
  double b[n]
  double c[n]
  receive a
  receive b
  int j := i
  repeat
    c[j] := 0.0
    for k := 0 to n - 1 do
      c[j] += a[k] * b[k]
    end for
    j := (j - 1) % n
    if j ≠ i then
      send b to worker[(i + 1) % n]
      receive b
    end if
  until j == i
  send i, c to coordinator
end process
```

- ▷ row  $i$  of  $a$
- ▷ one column of  $b$
- ▷ row  $i$  of  $c$  (result)
- ▷ row  $i$  from coordinator
- ▷ col  $i$  from coordinator
- ▷ **Inv:**  $b$  holds column  $j$  of matrix  $B$

▷ col  $j$



## Peers (cont'd)

```
process coordinator
  for  $i := 0$  to  $n - 1$  do
    send  $A[i][*]$  to worker[ $i$ ]
  end for
  for  $j := 0$  to  $n - 1$  do
    send  $B[*][j]$  to worker[ $j$ ]
  end for
  for  $i := 0$  to  $n - 1$  do
    receive  $C[i][*]$  from worker[ $i$ ]
  end for
end process
```

- First each row of  $A$  is sent to a worker.
- Each column of  $B$  is sent to a worker.
- The workers pass the columns of  $B$  among themselves (in a ring) until each worker has seen all  $n$  columns of  $B$ .
- The rows of  $C$  are now sent from the workers to the coordinator.

### Connectivity required

- Workers in a (1-way) ring.
- All workers connected (2-way) to the coordinator.

# Typical Applicability

| Programming Pattern   | Application Class |      |       |
|-----------------------|-------------------|------|-------|
|                       | MT                | Dist | Comp. |
| Iterative Parallelism |                   |      | ✓     |
| Recursive parallelism |                   |      | ✓     |
| Producer/Consumer     | ✓                 | ✓    |       |
| Client/Server         | ✓                 | ✓    |       |
| Peers                 | ✓                 | ✓    | ✓     |

Don't take this too literally; there are exceptions.