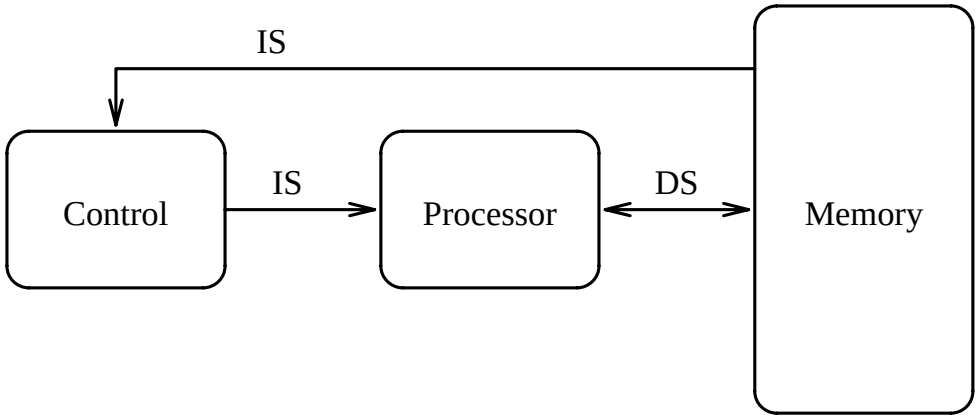
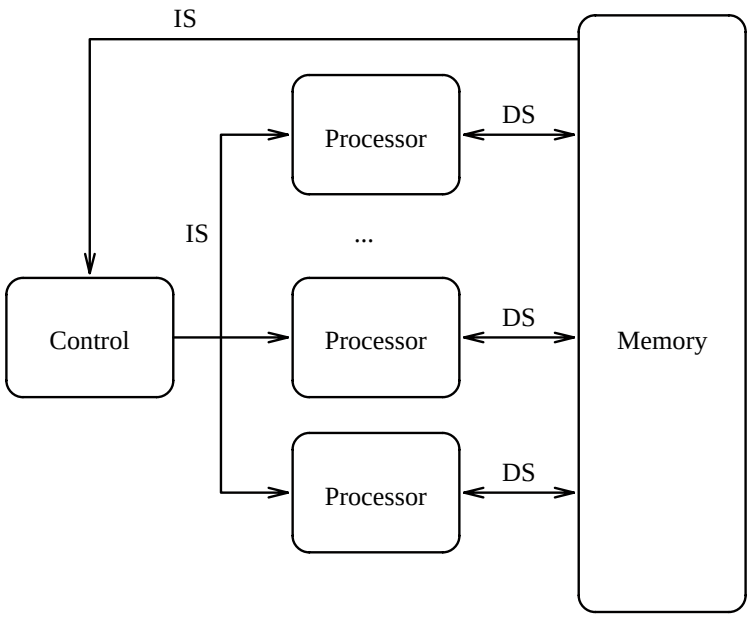
Concurrent Architectures

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

- Single Instruction Stream, Single Data Stream (SISD)
  Serial processing

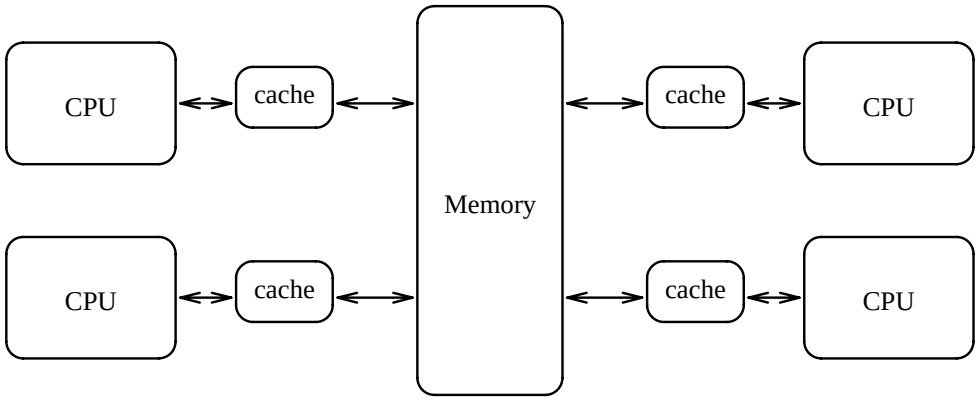


- Single Instruction Stream, Multiple Data Stream (SIMD) (Synchronous Mulitprocessor)



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

- MIMD Multi-Processor System



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

CPU Architectures for MIMD

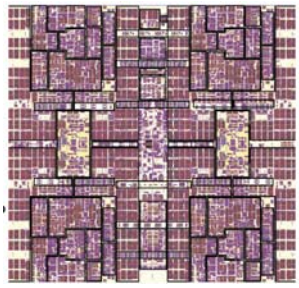
Single Processor Systems

- Architecturally a single CPU system is not a MIMD machine (it is SISD)
- However context switching (preemptive multitasking) allows one CPU to support many threads (instruction streams), thus simulating a MIMD.
- From a software designer's point of view the lack of multiple CPUs makes little difference.

Multiprocessor Systems

- Several CPU connected by memory or network
- Traditionally, each CPU runs one thread at a time.
- By context switching, each CPU can run multiple threads (though one at a time — traditionally)

Multi-core



IBM z10 4-core processor

- Multi-core technology means multiple CPUs on one die (chip)
- This is just multiprocessing
- What's important is that it is cheap, since it means no increase in chip count
- Multi-core is brings multiprocessing to the masses

Hardware Multithreading

- Several instruction streams share a common CPU.
- Think of a CPU with 2–16 sets of registers (including 2–16 Program Counters)
- When the CPU dispatches $\leq$1 instruction per clock (e.g.

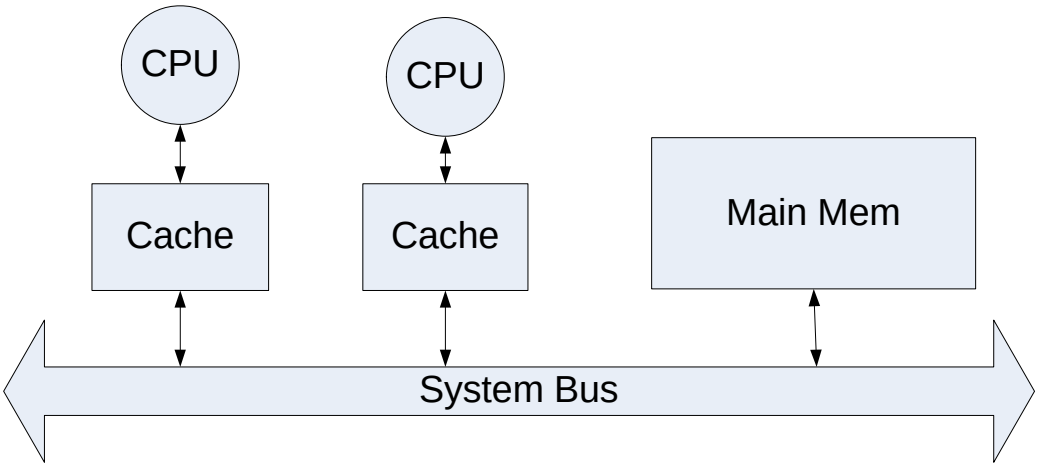
single pipeline), we have **fine-grained multithreading**
  - Example: Sun UltraSparc T1 & T2 (Niagara)
  - Example: Barrel processors
- When the CPU can dispatch >1 instruction per clock (superscalar), we have **simultaineous multithreading**
  - In each clock, instructions are dispatched from one or more instruction streams
  - Example: Intel Hyper-Threading processors

Memory Architectures for MIMD

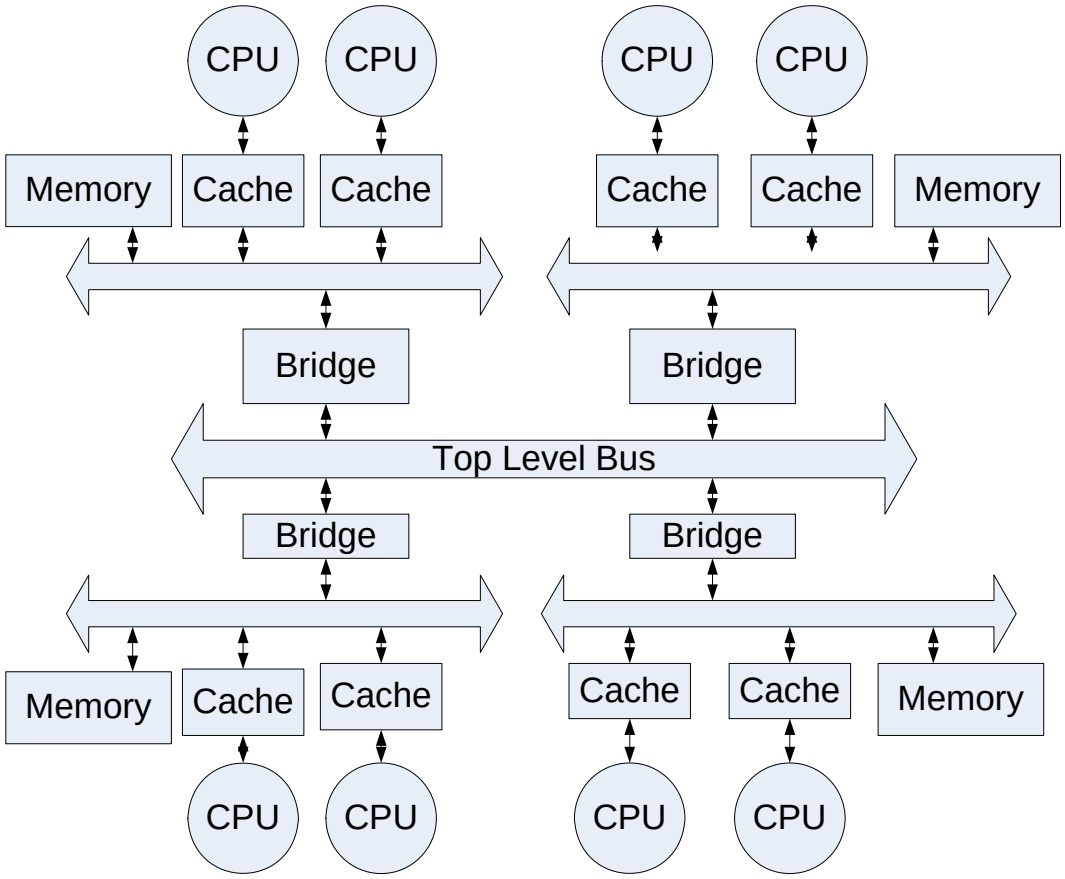
Shared Memory

- All processors ‘see’ the same address space.
- Physically memory may be shared or distributed.
- More flexibility in programming (message passing can be emulated).
- **Uniform memory access (UMA):**
  - Access time is uniform.
  - 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.

A UMA



A NUMA (Example)



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 [picture].

Cluster: Connected by LAN or WAN.

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

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)

	Multiprocessor		**DSM**		
	UMA	**NUMA**	**Page Based**	**Shared Vars**	**Shared Object**
Shared Virutal Addr. Space?	√	√	√		
Remote access in hardware?	√	√	No		
Ops converted to message by?	MMU	MMU	OS	OS	Run time system
Transfer Medium	Bus	Bus	Network	Network	Network
Operations	R/W	R/W	R/W	R/W	General
Migration done by	HW	SW	SW	SW	SW
Transfer unit	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

P1: $x := x + 1$

P2: $x := x + 1$

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.

$$
\begin{array}{rlll}
 & \text{\# P0:} & \| & \text{\# P1:} \\
x, y := 0, 0; \ \textbf{co} & x := 1; & \| & \mathsf{print}\ y; \quad \textbf{oc} \\
 & y := 2; & \| & \mathsf{print}\ x;
\end{array}
$$

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

A consistency model specifies what guarantees the hardware (or OS or run-time system) makes to the software about the apparent ordering of operations.

I'll assume the operations are Read (R) and Write (W)

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 of concurrent reads 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.

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.

The following obeys strict and sequential consistency

P0:		W(x)0	W(x)1		
P1:				R(x)1	R(x)1
True Time:	$\longrightarrow$				

The following obeys sequential consistancy

P0:		W(x)0	W(x)1		
P1:				R(x)0	R(x)1
True Time:	$\longrightarrow$				

The second behaviour

P0:		W(x)0	W(x)1				
P1:						R(x)0	R(x)1
True Time:	$\longrightarrow$						

is consistent with an interleaving

P0:	W(x)0				W(x)1		
P1:			R(x)0				R(x)1
Time:	$\longrightarrow$						

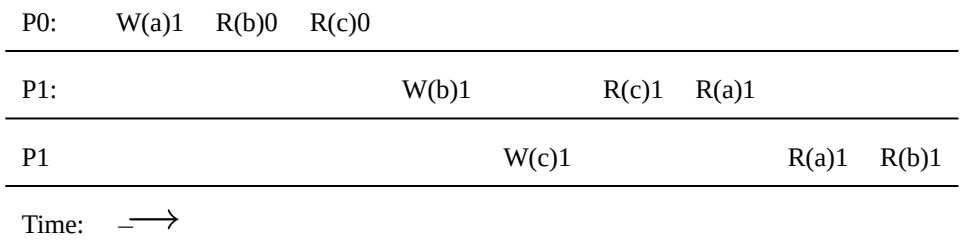
Sequential consistency example

(Example after Tanenbaum Distributed OSs)

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

	# P0:	‖	# P1:	‖	# P2:	
co	$a := 1;$	‖	$b := 1;$	‖	$c := 1;$	**oc**
	$x := 2b + c;$	‖	$y := 2c + a;$	‖	$z := 2a + b;$	

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



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

Weaker Models of Consistency

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

See Tanenbaum, Distributed Operating Systems, for more.

Cache coherence

Coherent caches ensure

- *If any cache contains a modified line, then no two caches disagree as to its value.*

This implements a constancy model stronger 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.
- **Invalid**. Line is not valid.

Bus transactions:

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

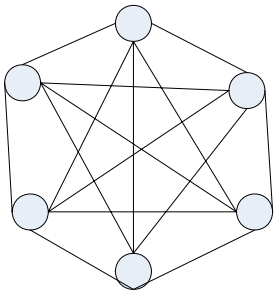
E.g.

- 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.

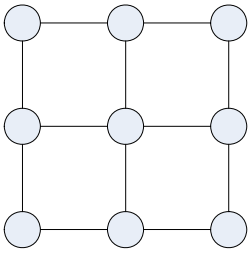
The sequence of bus transactions imposes a single order on what the processors see. Hence sequential consistency.

Network Topologies

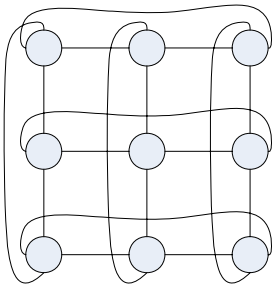
Physically, each processor can only connect to a limited number of other processors.



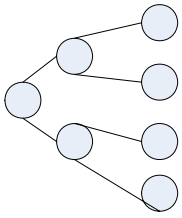
Fully connected



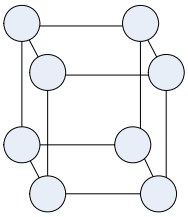
2D grid



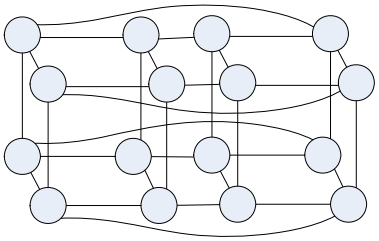
2D Toroidal Grid



Tree (*b*=2)



Hyper Cube (*n*=8)



Hyper Cube (*n*=16)

Others are also possible.

Network Topologies: Comparison

n is # of processors

	connections per node	$n = 8$	$n = 4096$
Fully connected	$n - 1$	7	4095
2D Toroidal Grid	4	4	4
3D Toroidal Grid	6	6	6
Tree	$b + 1$	3	3
Hyper-Cube	$\lg n$	3	12

Diameter is the max # hops between nodes.

	diameter	$n = 8$	$n = 4096$
Fully connected	1	1	1
2D Toroidal Grid	$\simeq \sqrt{n}$	3	64
3D Toroidal Grid	$\simeq \frac{3}{2}\sqrt[3]{n}$	2	24
Tree	$\simeq 2 \log_b n$	5	23
Hyper-Cube	$\lg n$	3	12

Furthermore, 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×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] {
        for [j := 0 to n-1] {
                c[i,j] := 0.0;
                for [k := 0 to n-1]
                        c[i,j] := c[i,j] + a[i,k] * b[k,j]; } }
```

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

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

End of aside

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 in ||
      co [j := 0 to n-1] { # All columns in ||
            c[i,j] := 0.0;
            for [k := 0 to n-1]
                  c[i,j] := c[i,j] + a[i,k] * b[k,j]; } }
```

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

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 := ⌈(w * n) ÷ P⌉ ; # first row of strip
    int last := ⌈((w + 1) * n) ÷ P⌉ − 1; # last row of strip
    for [i := first to last] {
        for [j := 0 to n-1] {
            c[i,j] := 0.0;
            for [k := 0 to n-1]
                c[i,j] := c[i,j] + a[i,k] * b[k,j]; } } }
```

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 $[left, right]$.

```
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( abs((larea+rarea) - area)) > EPSILON ) {
                larea := quad(left, mid, fleft, fmid, larea);
                rarea := quad(mid, right, fmid, fright, rarea); }
        return larea + rarea; }
```

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

```
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( abs((larea+rarea) - area) > EPSILON ) {
            co larea := quad(left, mid, fleft, fmid, larea);
            // rarea := quad(mid, right, fmid, fright, rarea);
            oc }
        return larea + rarea;
```

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 ) {
                await new client
                while( true ) {
                        receive request from client
                        process request
                        send reply
                        if( client requested quit ) break }
                clean up } }
```

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]; # row i of a
        double b[n]; # one column of b
        double c[n]; # row i of c (result)
        receive a ; # row i from coordinator
        receive b ; # col i from coordinator
        int j = i;
        ##Inv: b holds column j of matrix B.
        loop {
                c[j] := := 0.0 ;
                for [k = 0 to n-1] c[j] += a[k] * b[k];
                j := (j-1) mod n; # subtract 1 from j
        exit when j==i
                send b to worker[(i+1) mod n];
                receive b; # col j }
        send (i,c) to coordinator
```

The coordinator

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

- 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

	Application Class		
Programming Pattern	MT	Dist	‖ Comp.
Iterative Parallelism			√
Recursive parallelism			√
Producer/Consumer	√	√	
Client/Server	√	√	
Peers	√	√	√

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