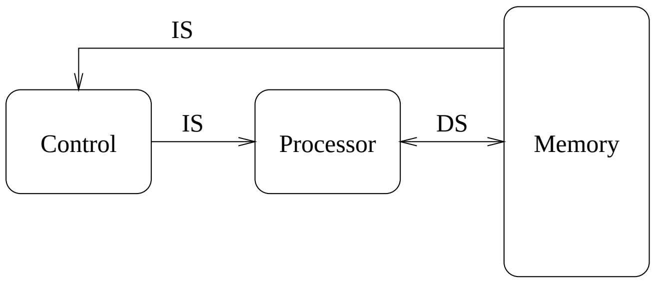
Concurrent Architectures

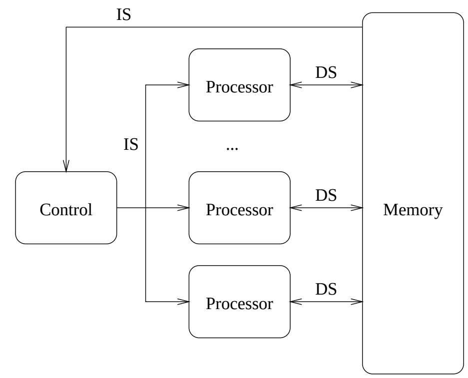
Architectures can be classified based on multiplicity of instruction and data streams (Flynn's taxonomy):

Single Instruction stream, Single Data Stream (SISD)

Serial processing

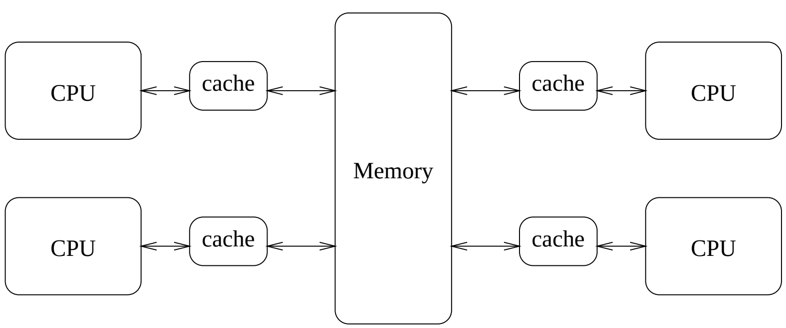


SI, Multiple Data Stream (SIMD) (Synchronous Mulitprocessor)



- All processors execute same instruction.
- Global clock.
- 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.

Memory Architectures

Shared Memory

- All processors 'see' the same address space.
- Actual 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.

Atomic Actions

process $p1$	**process** $p2$
$x := x + 1$	$x := x + 1$
end	**end**

What is the final value of x?

P1	**P2**		**P1**	**P2**
`LOAD x r1`			`INC x`	
	`LOAD x r1`			`INC x`
`ADD r1 #1`		**or**		
	`ADD r1 #1`			
`STORE r1 x`				
	`STORE r1 x`			

Cache Problems

Caching complicates things — processes may see updates at different times or in different orders.

process $p1$	**process** $p2$
$x := x + 1$	$y := y + 1$
end	**end**

False sharing — If x and y are in the same cache line then they are effectively shared. (We hope this is looked after by the cache hardware, but it might make processing slower.)

Distributed memory

(a.k.a. message passing, multicomputers)

- Each processor has private memory.
- Communication by message passing.
- Not good if processes must share large amounts of data.

Multicomputer — Distributed-memory multiprocessor with all processors and memory co-located.
- a.k.a. *tightly coupled machine*
- typically requires specialized hardware

Network system — Connected by LAN or WAN.
- Generic hardware.
- *Network of workstations* (NOW), *Cluster of workstations* (COW).

Software Architectures

Multithreaded Systems

- Typically more processes than processors.
- Divide overall (set of) problem(s) into (mostly) independent tasks — makes programming less complicated.
- Usually shared memory.

Distributed Systems

- E.g., data or application is physically distributed, or for fault tolerance.

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.

Iterative Parallelism

- Program with several, often identical process, each containing loops.
- Typical for scientific computations.

Example: Matrix Multiplication

Compute `c = a * b`, where `a`, `b` and `c` are $n \times n$ matricies. (n^2 inner products)

Sequential version:

```
double a[n,n], b[n,n], c[n,n];
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 — variables that an operation reads but does not modify.

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

Operations can be executed in parallel if they are *independent*.

It's always safe for processes to read variables that do not change.

Not safe (in general) if both write, or one writes and the other reads.

Processes a and b are *independent* iff

$$(\mathbf{W_a} \cap (\mathbf{W_b} \cup \mathbf{R_b}) = \oslash \wedge \mathbf{W_b} \cap (\mathbf{W_a} \cup \mathbf{R_a}) = \oslash)$$

In the matrix mulitiplication algorithm, each of the n^2 iterations of the dot product computation is independent of all the others so:

```
double a[n,n], b[n,n], c[n,n];
co [i = 0 to n-1] {   # All rows in parallel
  co [j = 0 to n-1] { # All columns in parallel
    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 this is wasteful. Having more processes than processors will slow down computation.

A better version: `P workers`, each of which computes a horizontal strip of `c`:

```
process worker[w = 1 to P] {
  int first = (w-1) * n/P;  # first row of strip
  int last = first + 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

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

Independent recursive procedures:

- At most read global (shared) variables.
- Reference/result parameters are distinct.

Example: Adaptive Quadrature

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

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


Producers & 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
- Servers may service multiple clients, possibly concurrently.

Peers

- Similar distributed processes cooperate to accomplish a task.

Example: Distributed Matrix Multiplication

```
process worker[i = 0 to n-1] {
  double a[n];          # row i of a
  doubel b[n];          # one column of b
  double c[n];          # row i of c (result)
  double sum = 0.0;
  int nextCol = i;
  receive row i of a and column i of b;
  for [k = 0 to n-1] sum = sum + a[k] * b[k];
  c[nextCol] = sum;
  for [j = 1 to n-1] {
    send b to worker[(i+1)%n];
    receive column of b from worker[(i+(n-1))%n];
    sum = 0.0;
```



```
    for [k = 0 to n-1] sum = sum + a[k] * b[k];
    nextCol = (nextCol + (n-1))%n;
    c[nextCol] = sum;
  }
  send c to coordinator
}
```