Flynn classification

S = single, M = multiple, I = instruction (stream), D = data (stream)

SISD	**MISD**
SIMD	**MIMD**

Virginia Tech

Basic concepts

Def. The *speedup* of an algorithm is

$$S_p = \frac{T^*}{T_p} = \frac{\text{time for best serial algorithm}}{\text{parallel time with } p \text{ processors}} \approx \frac{T_1}{T_p}.$$

Def. The *efficiency* of an algorithm is $E_p = \dfrac{S_p}{p}$.

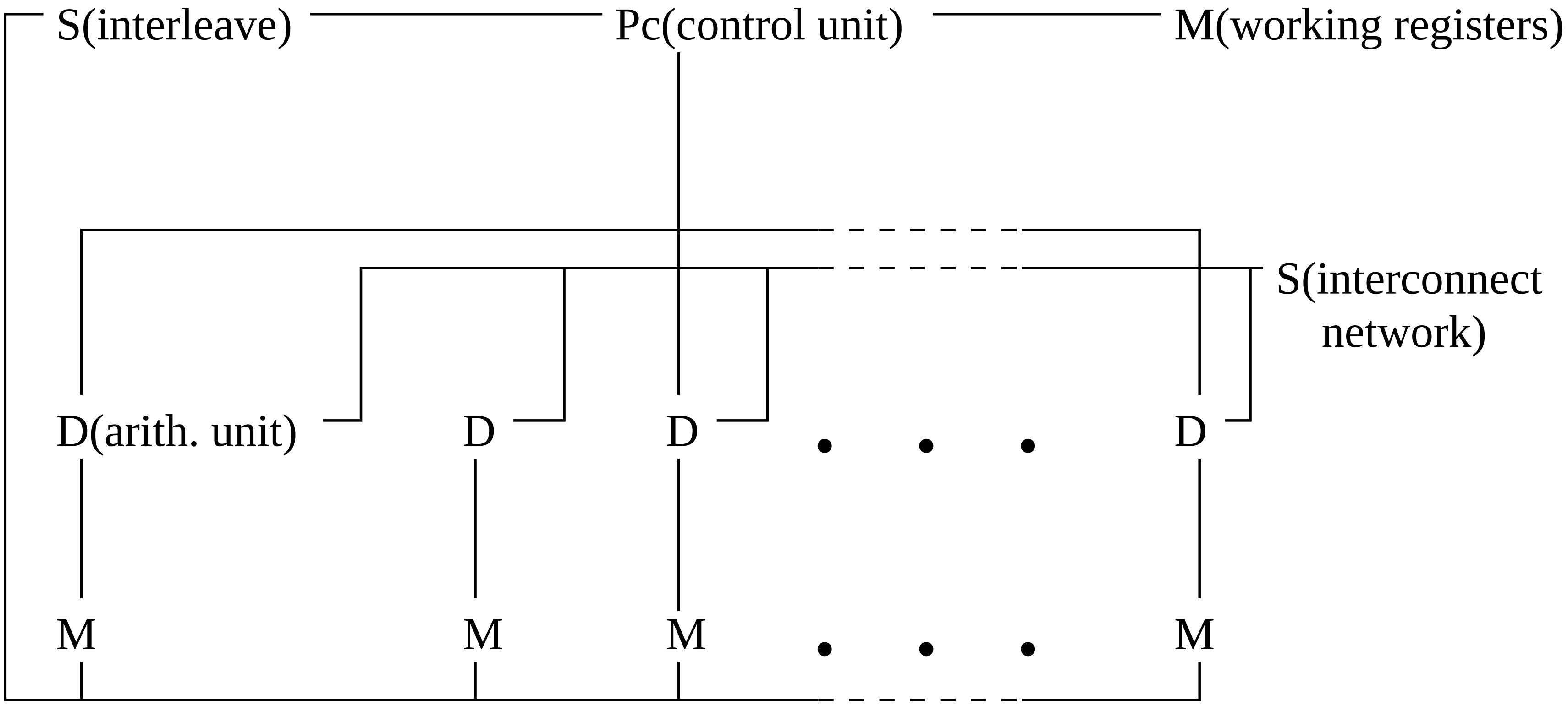
Amdahl's law: if a program consists of two parts, one that is inherently sequential and one that is fully parallelizable, and if the inherently sequential part consumes a fraction f of the total computation, then the speedup is limited by

$$S_p \leq \frac{1}{f + (1-f)/p} \leq \frac{1}{f}, \qquad \text{for all } p.$$

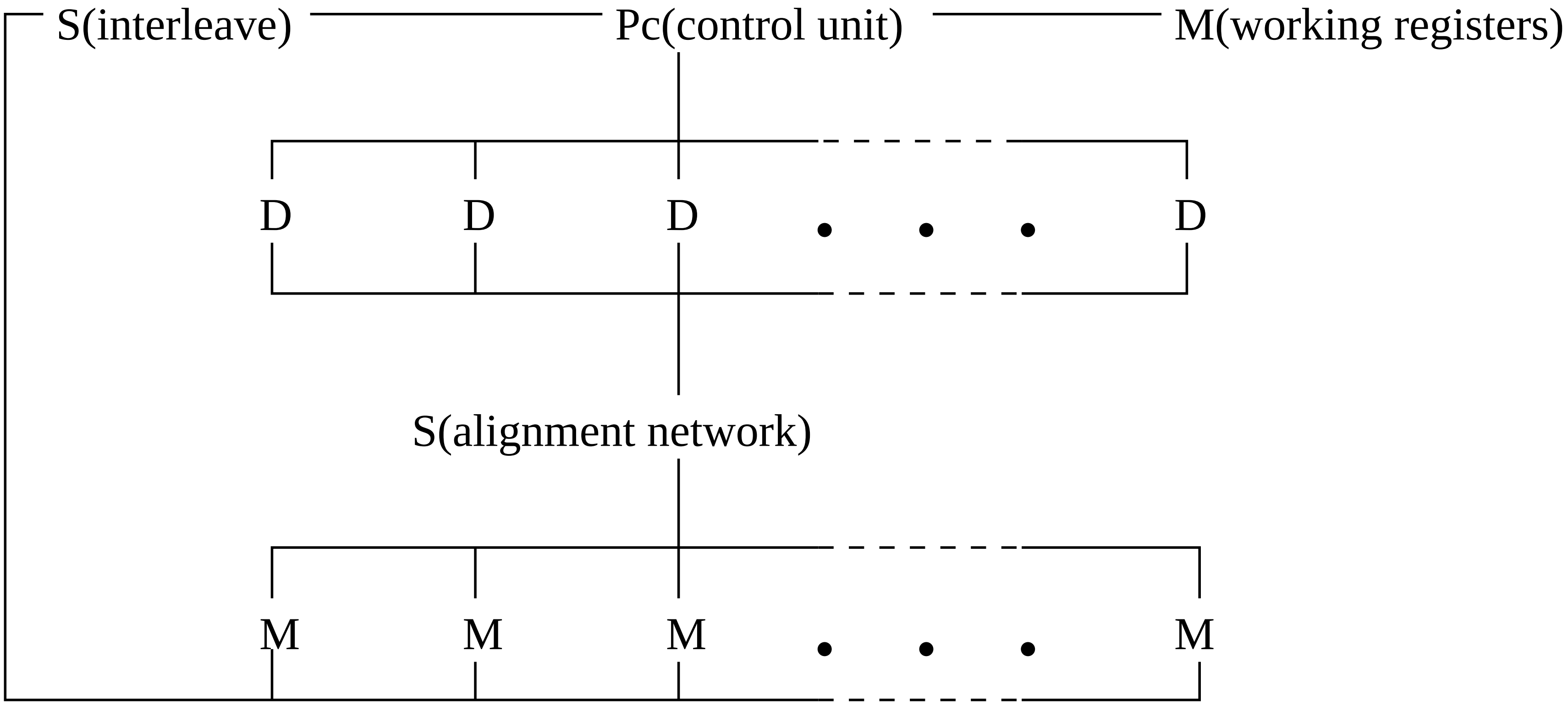
PMS notation

P	processor, including instruction interpretation and execution
M	memory, registers, cache, secondary storage
S	switch, often implicit in line junction
L	link, often just a line
T	transducer, I/O device
K	controller, generates microsteps for single operations applied externally
D	data processing, arithmetic, any transformation of data
C	computer, complete system

Distributed memory SIMD computer

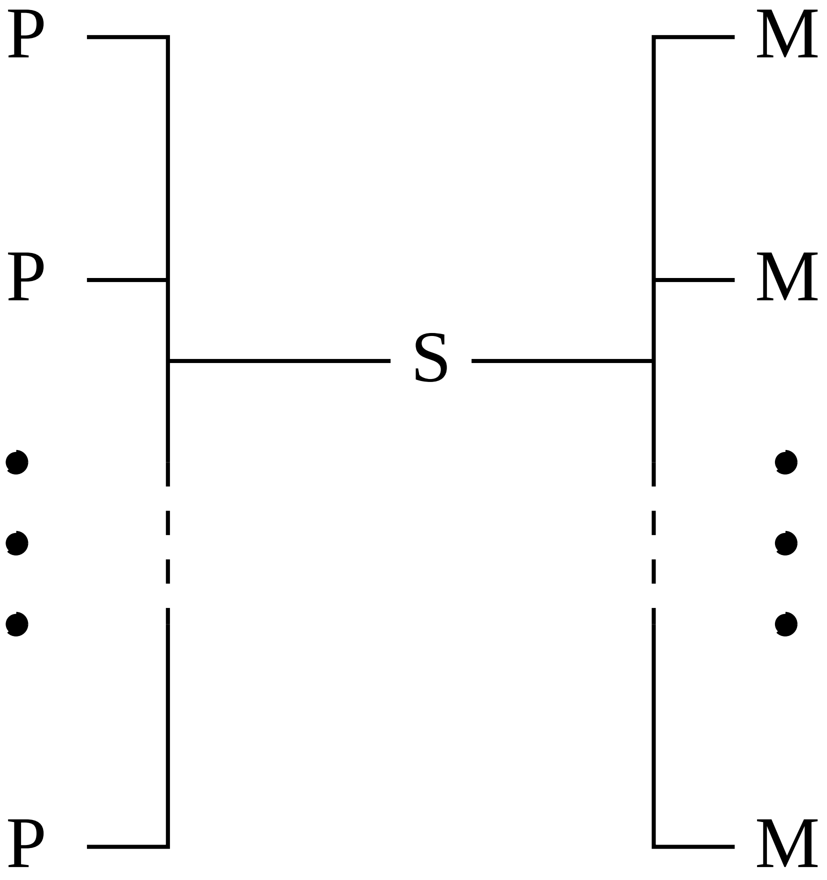


Shared memory SIMD computer

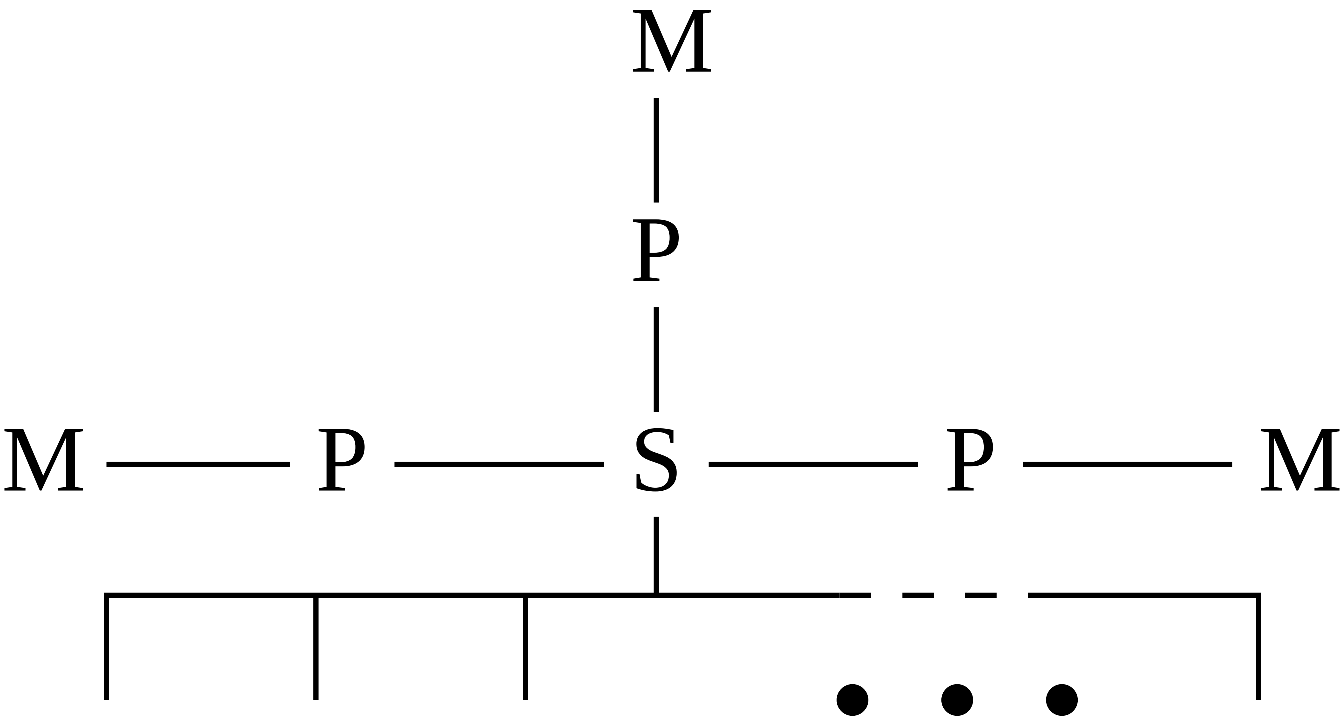


S(interleave) Pc(control unit) M(working registers)

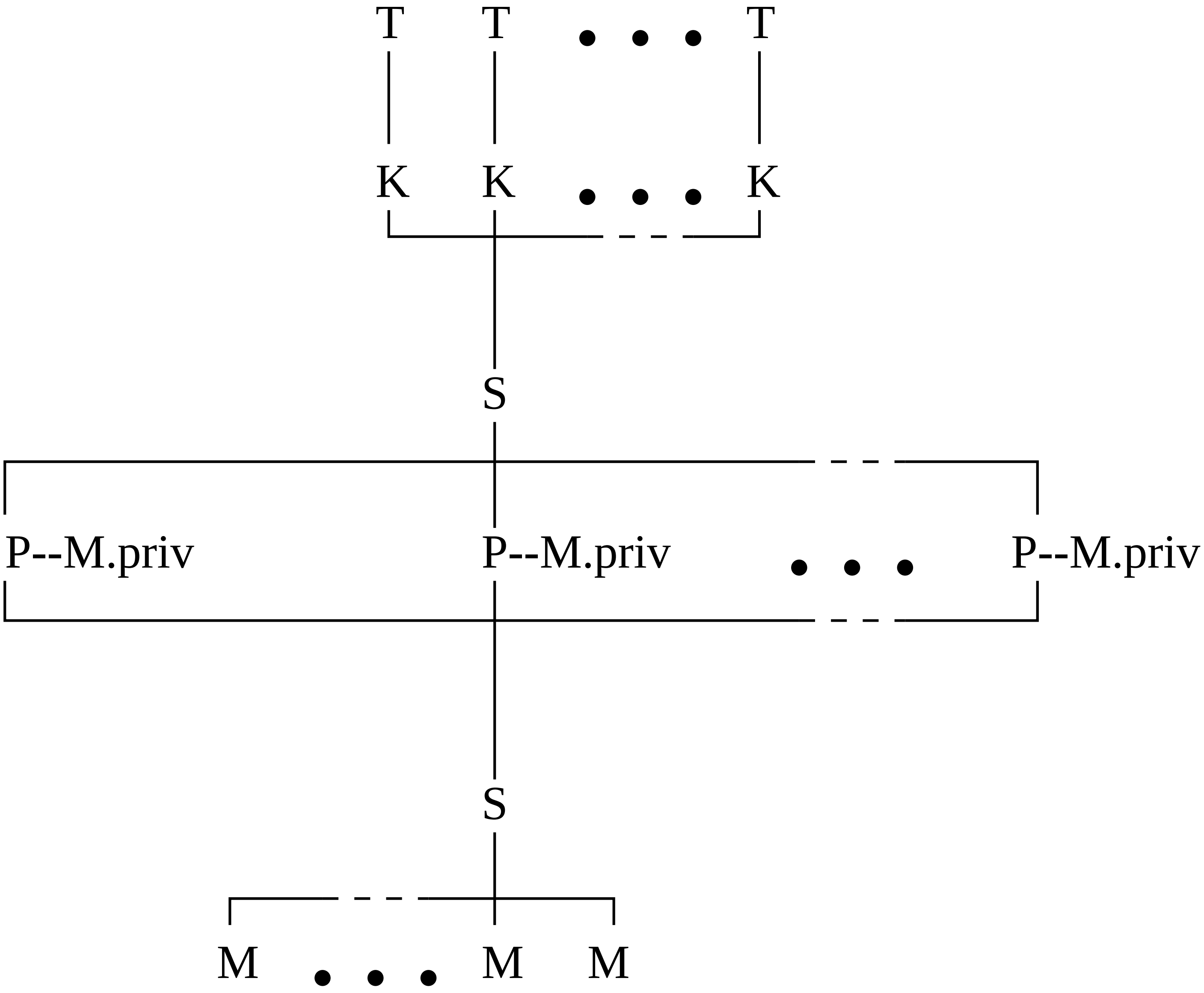
Shared memory multiprocessor



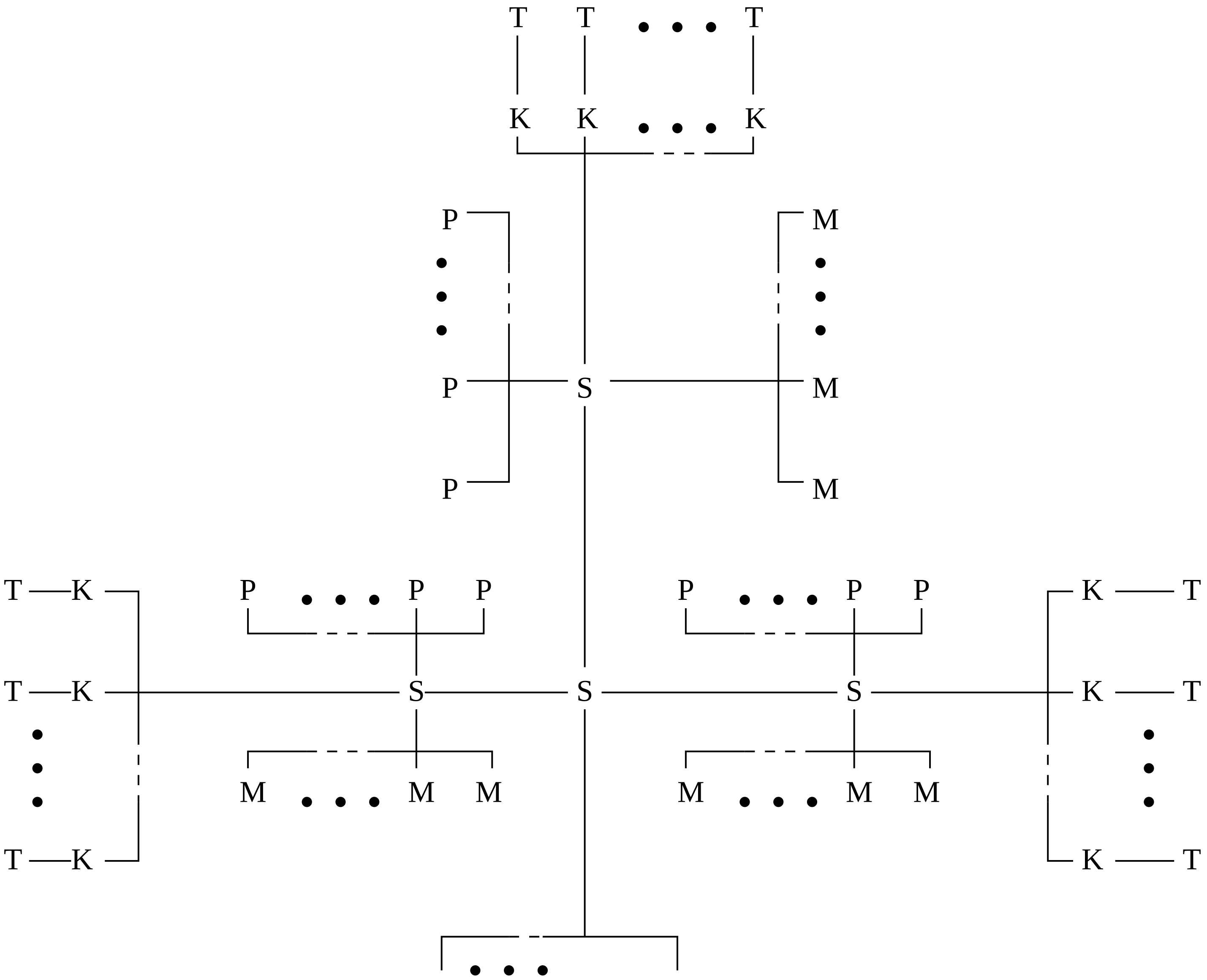
Message passing multiprocessor



Shared memory multiprocessor with private memories



Interconnected shared memory clusters



SIMD algorithms—linear recurrence

An mth order linear recurrence $R(n,m)$, where $m \le n-1$, is

$$x_i = 0, \qquad i \le 0,$$
$$x_i = c_i + \sum_{j=i-m}^{i-1} a_{ij} x_j, \qquad 1 \le i \le n.$$

The case $m = n-1$ is called a *general linear recurrence*. SIMD code for a general linear recurrence is

```
$x[i] := c[i],$        $(1 \le i \le n);$                (column sweep)
for $j := 1$ step $1$ until $n-1$
    $x[i] := x[i] + a[i,j] * x[j],$        $(j+1 \le i \le \min(j+m, n));$
```

SIMD algorithms—matrix-matrix multiply

SIMD matrix-matrix multiply (outer product version):

```
for $i := 1$ step $1$ until $N$
    for $j := 1$ step $1$ until $N$
        $c[i, j] := 0$;
for $k := 1$ step $1$ until $N$                (sum of $N$  $N \times N$ matrices)
    for $i := 1$ step $1$ until $N$
        for $j := 1$ step $1$ until $N$
            $c[i, j] := c[i, j] + a[i, k] \times b[k, j]$;
```

Shared vs. distributed memory

Task	Shared memory	Message passing
interprocessor communication	memory read/write	software send/receive
memory read/write	long and variable latency	only to private memory
messages through switch	single memory word	long, aggregated
collision avoidance	request randomization	global scheduling of messages

Multiprocessor recurrence solver

Here is a naive (and incorrect) parallel program for a shared memory (or implicit message passing) multiprocessor:

```
shared $n$, $a[n,n]$, $x[n]$, $c[n]$;
private $i$, $j$;
for $i := 1$ step $1$ until $n-1$ fork DOROW;
$i := n$;        /* Initial process handles $i = n$. */
DOROW:    $x[i] := c[i]$;
          for $j := 1$ step $1$ until $i - 1$
              $x[i] := x[i] + a[i,j] * x[j]$;
          join $n$;
```

Synchronization has two flavors: *control-based* involves progress of other processes/threads, *data-based* involves status of some variable.

Parallel programming concepts

Producer/consumer synchronization associates a full/empty state with each variable and uses synchronized read and write operations that operate only when the variable has a specified state.

Syntax: **produce** <*shared variable*> := <*expression*>
consume <*shared variable*> **into** <*private variable*>
copy <*shared variable*> **into** <*private variable*>
void <*shared variable*>

Atomicity: an atomic operation takes place indivisibly with respect to other parallel operations. Atomic operations can be achieved by *mutual exclusion*; such a region of code is called a *critical section*.

```
Syntax: critical
            <code>
        end critical
```

Recurrence solver producer/consumer synchronized on x[j]

```
procedure dorow(value $i$, $done$, $n$, $a$, $x$, $c$)
        shared $n$, $a[n,n]$, $x[n]$, $c[n]$, $done$;
        private $i$, $j$, $sum$, $priv$;
        $sum = c[i]$;
        for $j := 1$ step $1$ until $i - 1$
            {copy $x[j]$ into $priv$;        /* Get $x[j]$ when available. */
             $sum := sum + a[i,j] * priv$; }
        produce $x[i] := sum$;        /* Make $x[i]$ available to others. */
        $done := done - 1$;
        return;
end procedure
```

Recurrence solver producer/consumer synchronized on x[j] (continued)

```
shared $n$, $a[n, n]$, $x[n]$, $c[n]$, $done$;
private $i$;
$done := n$;
for $i := 1$ step $1$ until $n - 1$
    {void $x[i]$;
     create dorow($i$, $done$, $n$, $a$, $x$, $c$); }        /* Create $n - 1$ procedures. */
$i := n$;
void $x[i]$;
call dorow($i$, $done$, $n$, $a$, $x$, $c$);        /* Call the $n$th one. */
while $(done \neq 0)$   ;        /* Loop until all procedure instances finish. */
    <code to use $x[\,]$>
```

Final, synchronized, multiprocessor recurrence solver

```
procedure dorow(value $i$, $done$, $n$, $a$, $x$, $c$)
          shared $n$, $a[n,n]$, $x[n]$, $c[n]$, $done$;
          private $i$, $j$, $sum$, $priv$;
          $sum = c[i]$;
          for $j := 1$ step $1$ until $i-1$
             {copy $x[j]$ into $priv$;
              $sum := sum + a[i,j] * priv;$ }
          produce $x[i] := sum$;
          critical        /* Lock out other processes. */
             $done := done - 1$;        /* Decrement shared $done$. */
          end critical        /* Allow other processes. */
          return;
end procedure
```

Final, synchronized, multiprocessor recurrence solver (continued)

```
shared $n$, $a[n,n]$, $x[n]$, $c[n]$, $done$;
private $i$;
$done := n$;
for $i := 1$ step $1$ until $n - 1$
      {void $x[i]$;
        create dorow($i$, $done$, $n$, $a$, $x$, $c$); }
$i := n$;
void $x[i]$;
call dorow($i$, $done$, $n$, $a$, $x$, $c$);
while $(done \neq 0)$    ;
      <code to use $x[\,]$>
```

Loop scheduling algorithms

Consider the FOR loop: **forall** $i := lwr$ **step** stp **until** upr

```
shared $lwr$, $stp$, $upr$, $np$;        /* Block mapping. */
private $i$, $lb$, $ub$, $me$;
/* Compute private lower and upper bounds from $lwr$, $upr$, $stp$, process number
$me$, and number $np$ of processes. */
for $i := lb$ step $stp$ until $ub$
    $\langle$loop body($i$)$\rangle$;
```

```
shared $lwr$, $stp$, $upr$, $np$;        /* Cyclic mapping. */
private $i$, $me$;
for $i := lwr + me * stp$ step $np * stp$ until $upr$
    $\langle$loop body($i$)$\rangle$;
```

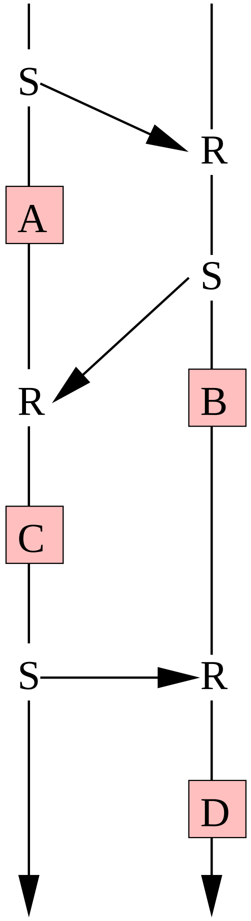
Loop scheduling algorithms (continued)

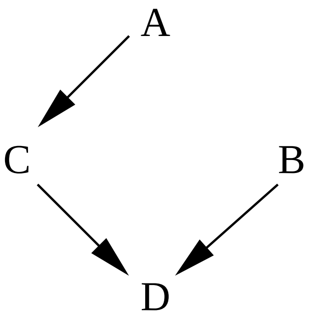
```
shared $lwr$, $stp$, $upr$, $np$, $isync$;       /* Self-scheduling code for each process. */
private $i$;
barrier
        void $isync$;
        produce $isync := lwr$;
end barrier
while (true)
begin
      consume $isync$ into $i$;
      if $(i > upr)$ then
        {produce $isync := i$;
         break;}       /* End while loop. */
      else
        {produce $isync := i + stp$;
              $\langle$loop body($i$)$\rangle$; }
end
```

Distributed memory multiprocessors

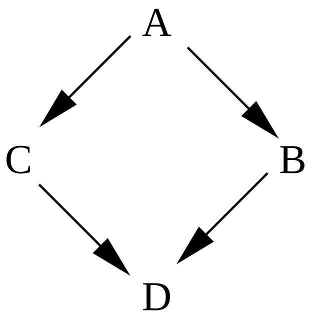
The type of locality required for good distributed memory multiprocessor performance is called *partitionable locality*. This is often achieved in real problems by physical *domain decomposition*. (Large area/perimeter or volume/surface ratios are desirable.)

Example of precedence imposed by interprocess communication:





Only receive blocking



Process rendezvous (both send and receive blocking)