Flynn classification

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

<table>
<thead>
<tr>
<th></th>
<th>SISD</th>
<th>MISD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD</td>
<td></td>
<td>MIMD</td>
</tr>
</tbody>
</table>
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 \( E_p = \frac{S_p}{p}. \)

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

\[ S_p \leq \frac{1}{f + (1 - f)/p} \leq \frac{1}{f}, \quad \text{for all } p. \]
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

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

S (interconnect network)

D (arith. unit)

M

M

M

M

M

M

M
Shared memory SIMD computer

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

D  D  D  D  •  •  •  D

S(alignment network)

M  M  M  M  •  •  •  M
Shared memory multiprocessor
Message passing multiprocessor
Shared memory multiprocessor with private memories
Interconnected shared memory clusters
**SIMD algorithms—linear recurrence**

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

$$x_i = 0, \quad i \leq 0,$$

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

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

$$x[i] := c[i], \quad (1 \leq i \leq n); \quad \text{(column sweep)}$$

**for** $j := 1$ **step** 1 **until** $n - 1$

$$x[i] := x[i] + a[i, j] * x[j], \quad (j + 1 \leq i \leq \min(j + m, n));$$
SIMD algorithms—matrix-matrix multiply

SIMD matrix-matrix multiply (outer product version):

\[
\text{for } i := 1 \text{ step } 1 \text{ until } N \\
\quad \text{for } j := 1 \text{ step } 1 \text{ until } N \\
\quad \quad c[i, j] := 0; \\
\quad \text{for } k := 1 \text{ step } 1 \text{ until } N \quad \text{(sum of } N \times N \times N \text{ matrices)} \\
\quad \quad \text{for } i := 1 \text{ step } 1 \text{ until } N \\
\quad \quad \quad \text{for } j := 1 \text{ step } 1 \text{ until } N \\
\quad \quad \quad \quad c[i, j] := c[i, j] + a[i, k] \times b[k, j];
\]
## Shared vs. distributed memory

<table>
<thead>
<tr>
<th>Task</th>
<th>Shared memory</th>
<th>Message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>interprocessor communication</td>
<td>memory read/write</td>
<td>software send/receive</td>
</tr>
<tr>
<td>memory read/write</td>
<td>long and variable latency</td>
<td>only to private memory</td>
</tr>
<tr>
<td>messages through switch</td>
<td>single memory word</td>
<td>long, aggregated</td>
</tr>
<tr>
<td>collision avoidance</td>
<td>request randomization</td>
<td>global scheduling of messages</td>
</tr>
</tbody>
</table>
Multiprocessor recurrence solver

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

```plaintext
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:  
\[
\begin{align*}
\text{produce} & \quad < \text{shared variable} > \ := \ < \text{expression} > \\
\text{consume} & \quad < \text{shared variable} > \ \text{into} \ < \text{private variable} > \\
\text{copy} & \quad < \text{shared variable} > \ \text{into} \ < \text{private variable} > \\
\text{void} & \quad < \text{shared variable} >
\end{align*}
\]

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:  
\[
\begin{align*}
\text{critical} & \quad < \text{code} > \\
\text{end critical}
\end{align*}
\]
Recurrence solver producer/consumer synchronized on \(x[j]\)

```plaintext
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)

```c
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 nth one. */
while (done ≠ 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)

```c
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 ≠ 0) ;
    <code to use x[]>
```
Loop scheduling algorithms

Consider the FOR loop:  \texttt{forall } i := lwr \texttt{ step stp until upr}

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

\texttt{shared } lwr, stp, upr, np; /* Cyclic mapping. */
\texttt{private } i, me;
\texttt{for } i := lwr + me * stp \texttt{ step np * stp until } upr
\qquad (\textit{loop body}(i));
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;
     (loop body(i));
    }
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)