

## PARALLEL PROCESSING

- Numerical problems and algorithms
- Impact of Amdahl's law
- Parallel computer architectures
  - ▷ Memory architectures
  - ▷ Interconnection networks
- Flynn's taxonomy (SIMD, MIMD, etc.)
- Shared-memory multiprocessors
- Distributed-memory, message-passing multicomputers

## COMPUTATIONAL PROBLEMS

- Numerical simulation has become as important as experimentation
  - ▷ Permits exploration of a much larger region in parameter space
  - ▷ All-numerical designs
- The most CPU-intensive application areas include:
  - ▷ Computational fluid dynamics
  - ▷ Computational electromagnetics
  - ▷ Molecular modeling
- Typical problem sizes (in double-precision floating-point operations) and execution times (in CPU-days @  $10^9$  FLOPS):

Problem size	Execution time (CPU-days)
$10^{14}$	1.16
$10^{15}$	11.57
$10^{16}$	115.7
$10^{17}$	1157

## NUMERICAL ALGORITHMS

- Most of the important CPU-intensive applications involve the solution of a system of coupled partial differential equations
- Initial-value problems
  - ▷ Time-dependent wave propagation (electromagnetics, geophysics, optical networking)
  - ▷ Time-dependent dynamics (fluid flow, mechanical engineering)
- Boundary-value problems
  - ▷ Typically require solution of very large systems of linear equations
  - ▷ Quasi-static electromagnetic problems
  - ▷ Static mechanical engineering (finite element analysis)
  - ▷ Eigenvalue problems (molecular energy surfaces)

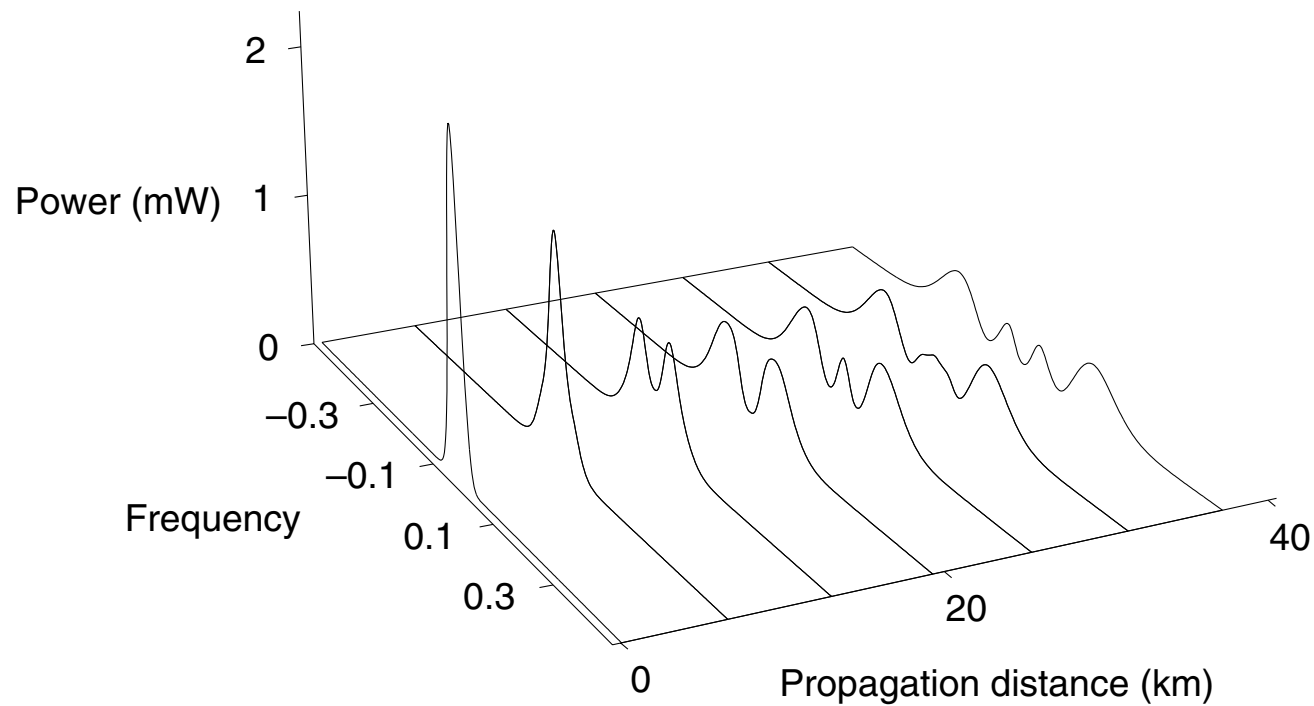
## PROPAGATION EQUATIONS FOR 4-WAVE MIXING

- Paraxial wave equation:

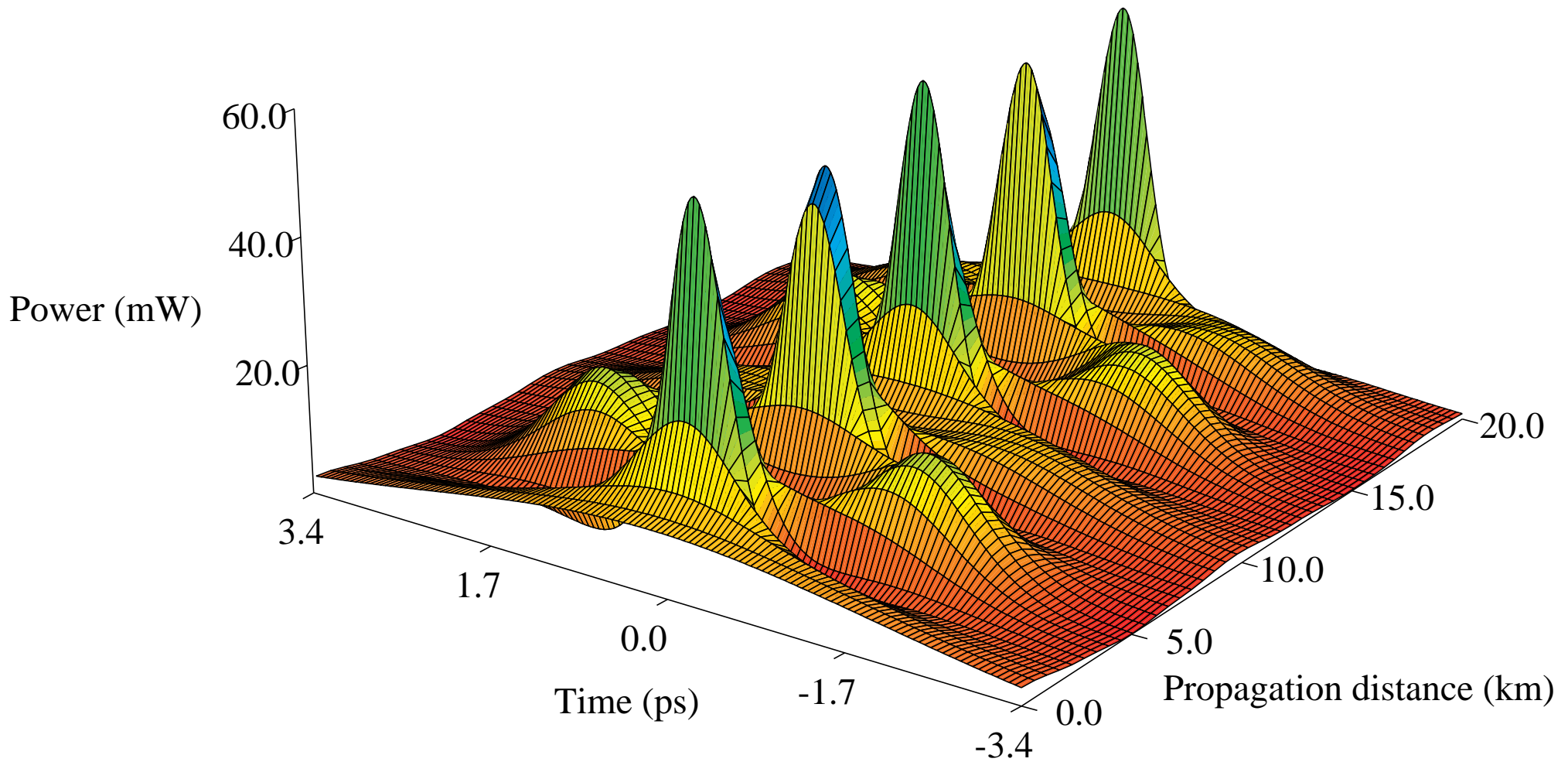
$$\begin{aligned} \left[ \frac{\partial}{\partial z'} + \left( i \frac{\beta_2}{2} \frac{\partial^2}{\partial t'^2} - \frac{\beta_3}{6} \frac{\partial^3}{\partial t'^3} \right) \right] \overline{\mathcal{F}}_n \\ = -\frac{\alpha}{2} \overline{\mathcal{F}}_n + i \frac{16\pi^2 \omega_n \chi^{(3)}}{n_{0,n}^2 c^2 A_e} \mu_{nklm} D_{|k|,|l|,|m|} \\ \times \overline{\mathcal{F}}_k \overline{\mathcal{F}}_l \overline{\mathcal{F}}_m e^{i(\Delta\beta_{nklm})z'} \end{aligned}$$

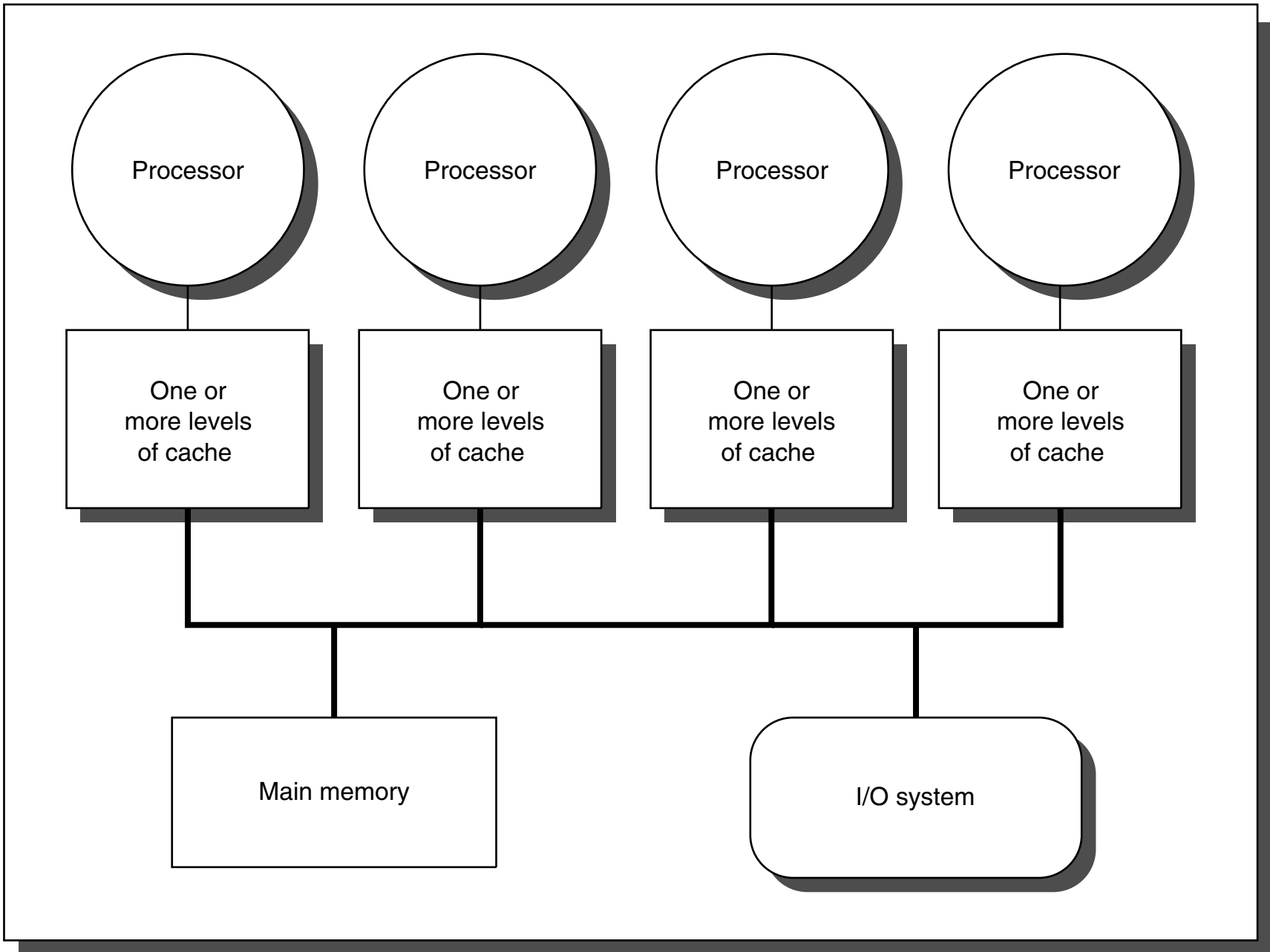
- Two regimes to study:
  - ▷ Three strong waves ( $\mathcal{F}_k, \mathcal{F}_l, \mathcal{F}_m$ ) generate nine weak waves ( $\mathcal{F}_n$ )
    - Useful for estimating crosstalk among channels
  - ▷ Parametric coupling of three strong waves
    - Leads to coherent amplification of some channels and de-amplification of others

# PULSE PROPAGATION IN AN OPTICAL FIBER (1)



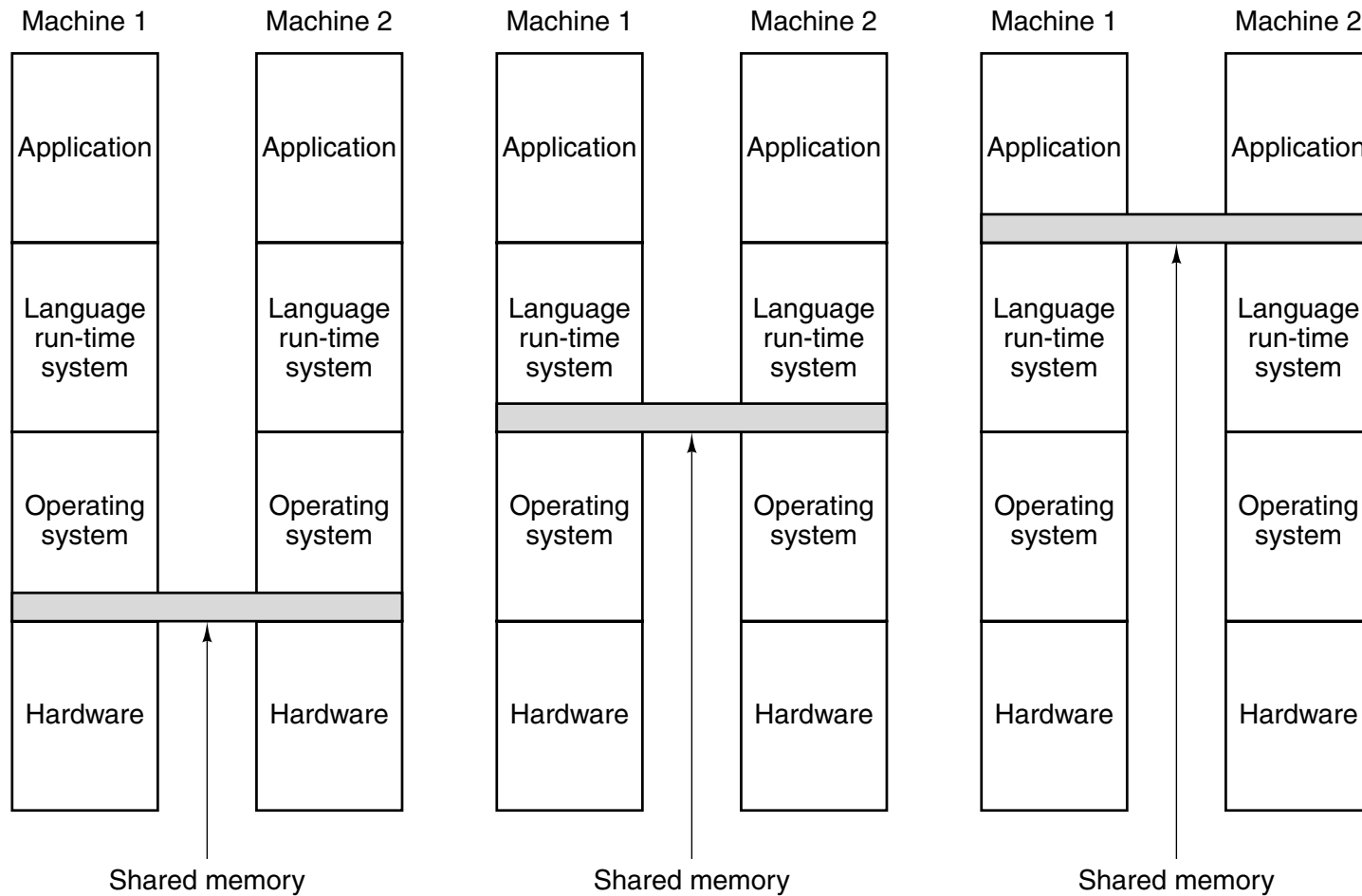
# PULSE PROPAGATION IN AN OPTICAL FIBER (2)

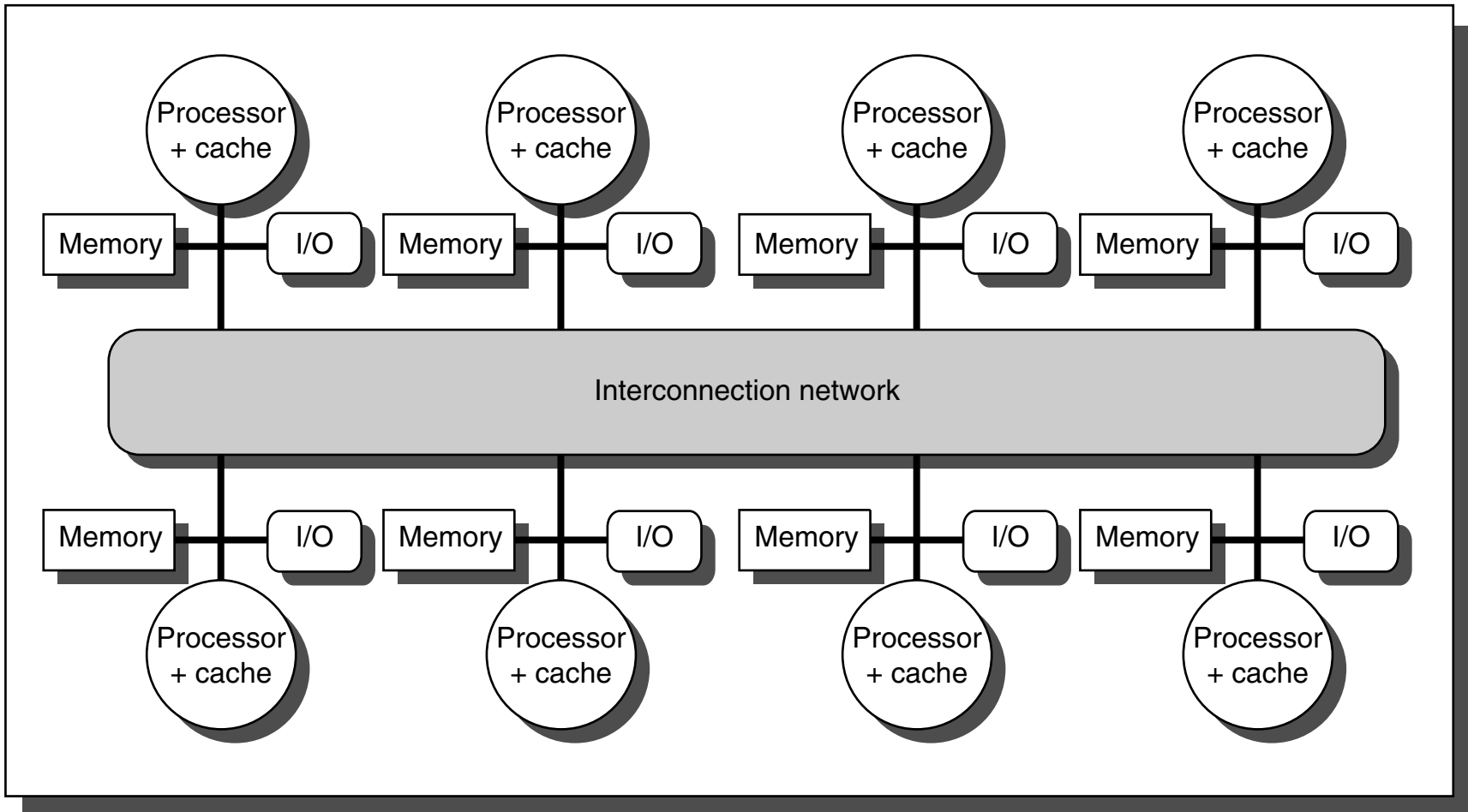




**A centralized, shared-memory multiprocessor**

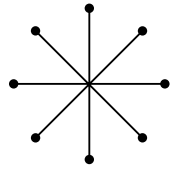
# Layers where shared memory can be implemented



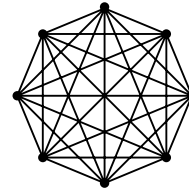


**A distributed-memory multicomputer**

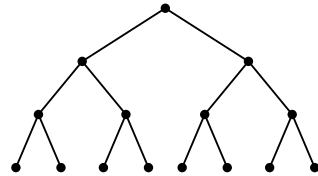
# Interconnection topologies



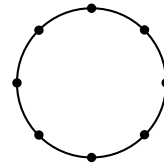
**star**



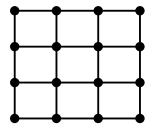
**full interconnection**



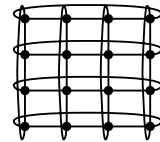
**tree**



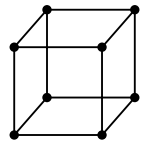
**ring**



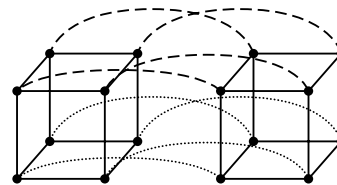
**grid**



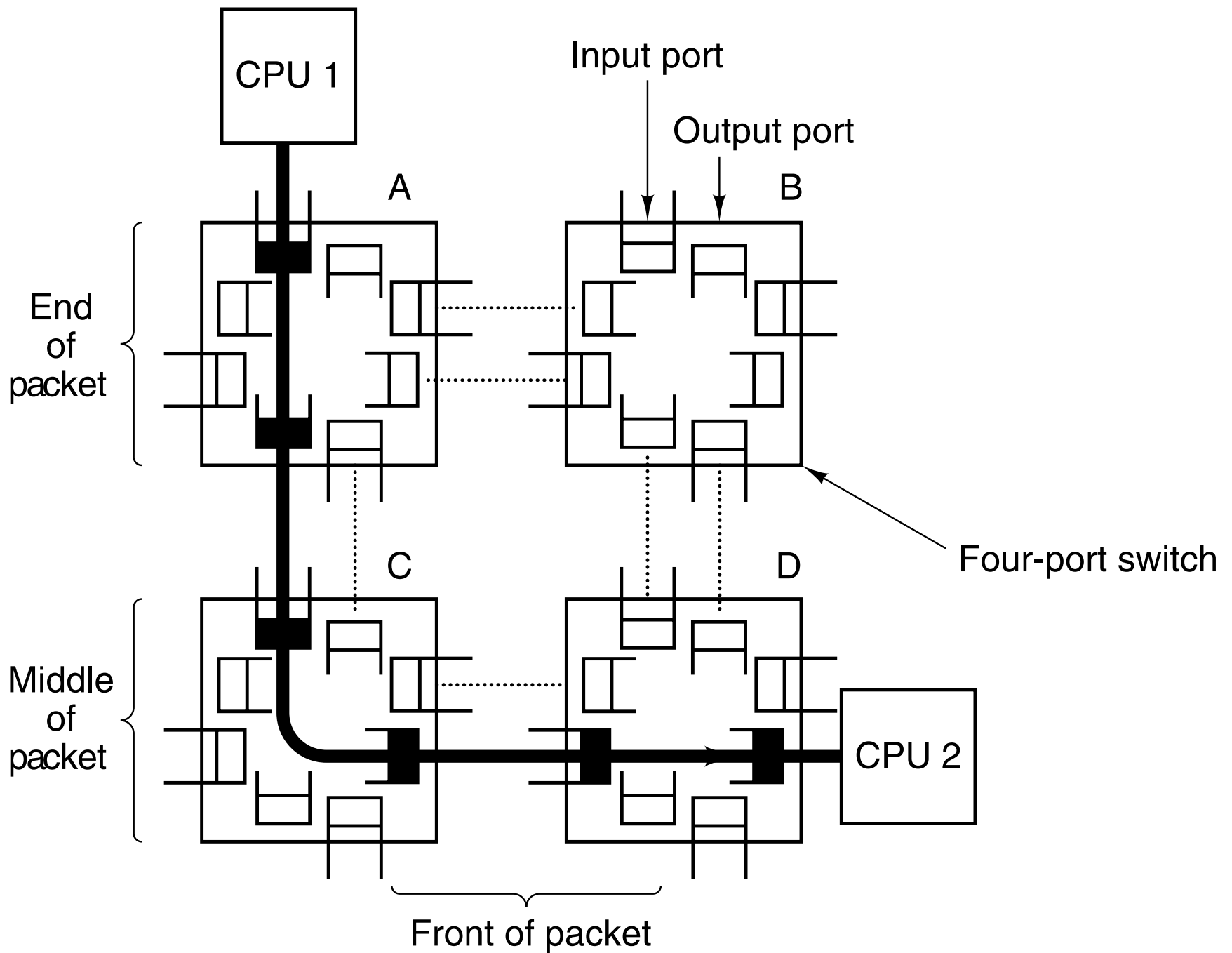
**double torus**



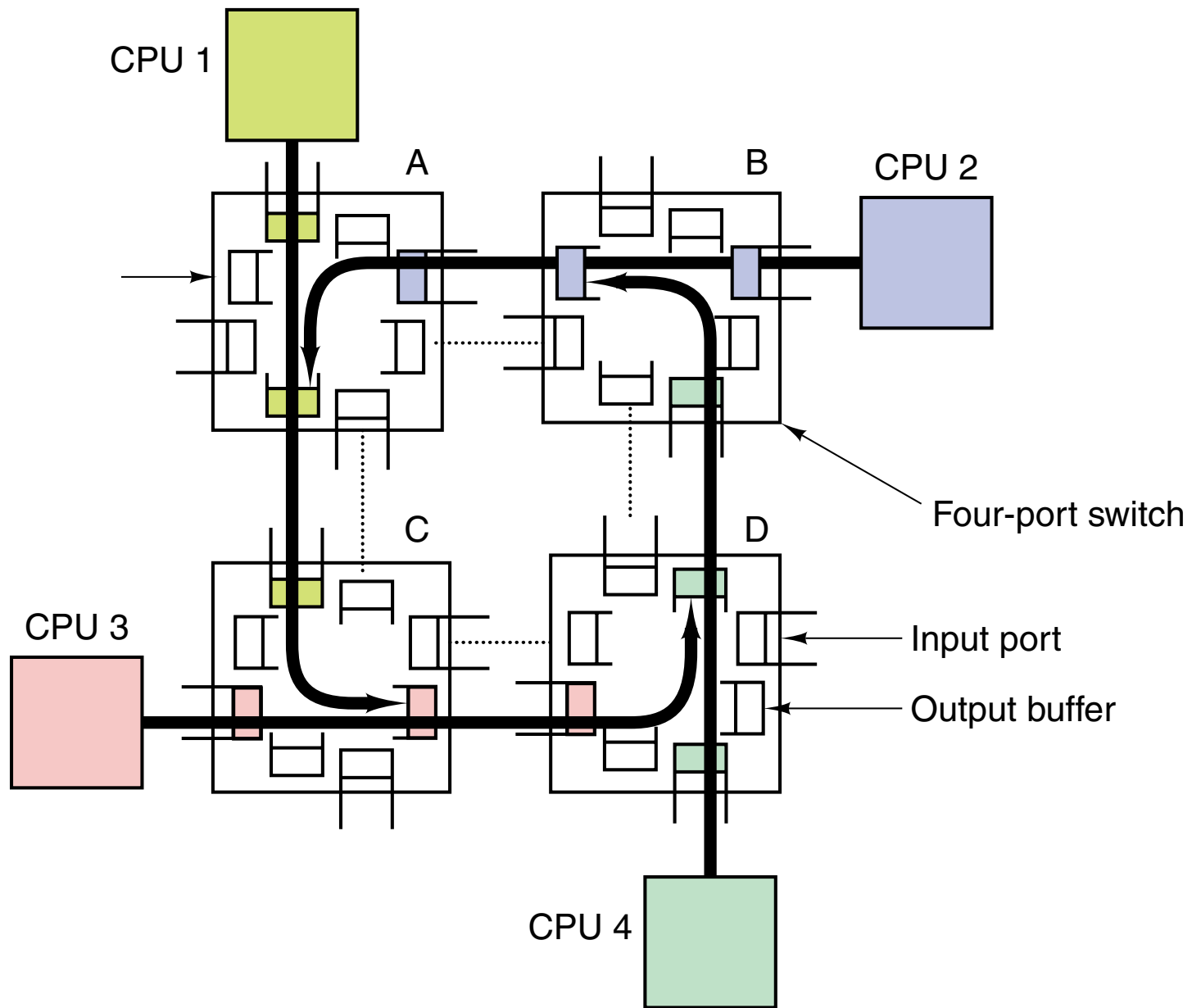
**cube**



**4-D hypercube**

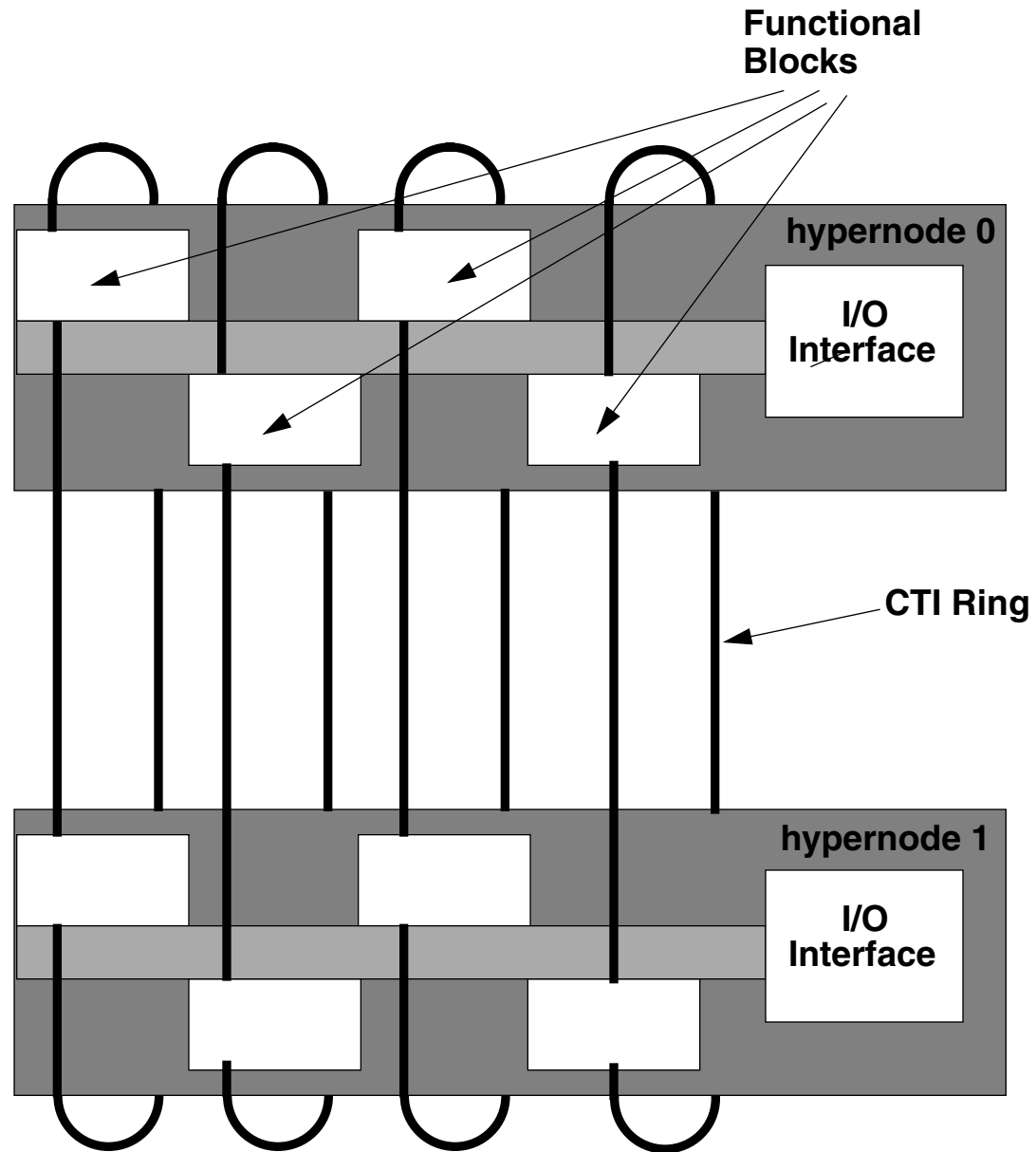


Interconnection network consisting of a 4-switch square grid



Deadlock in a circuit-switched interconnection network

# Two-Hypernode Convex Exemplar System



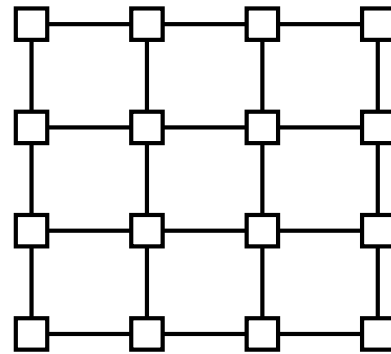
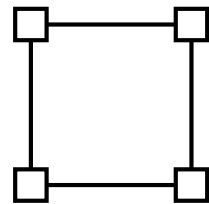
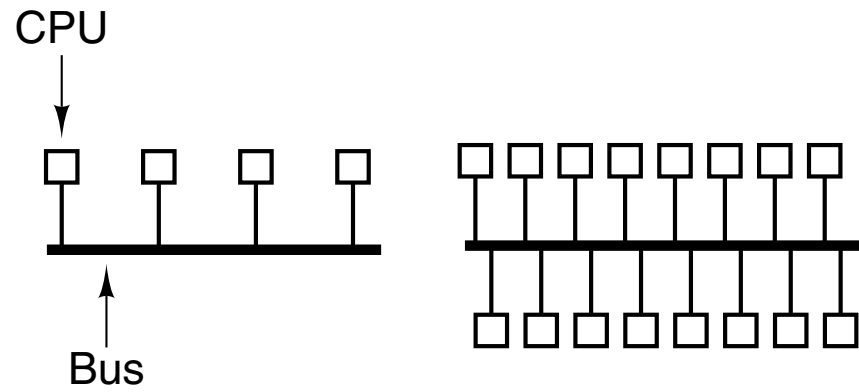
## Typical access times to retrieve a word from a remote memory

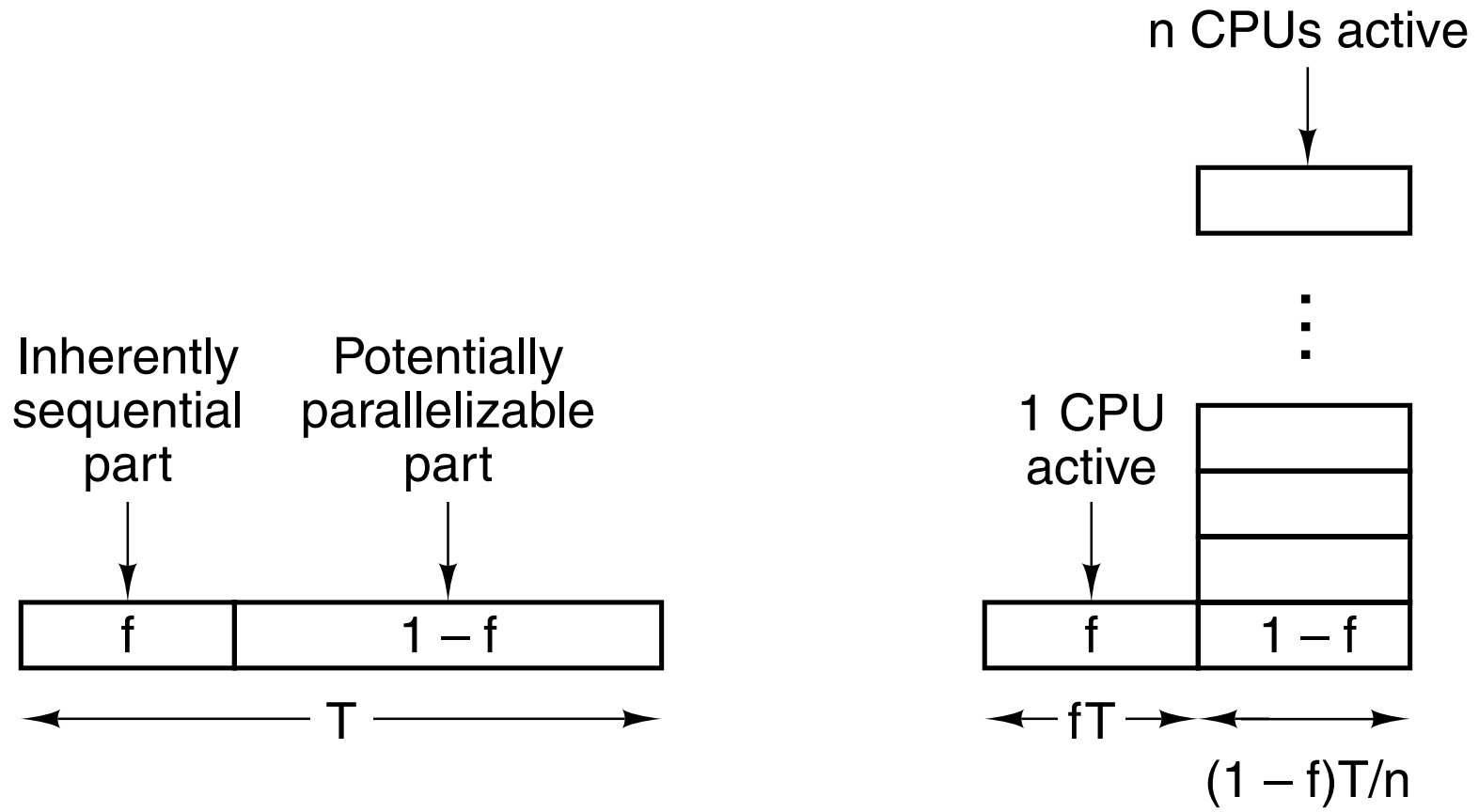
<b>Machine</b>	<b>Communication mechanism</b>	<b>Interconnection network</b>	<b>Processor count</b>	<b>Typical remote memory access time</b>
SPARCCenter	Shared memory	Bus	$\leq 20$	1 $\mu$ s
SGI Challenge	Shared memory	Bus	$\leq 36$	1 $\mu$ s
Cray T3D	Shared memory	3D torus	32–2048	1 $\mu$ s
Convex Exemplar	Shared memory	Crossbar + ring	8–64	2 $\mu$ s
KSR-1	Shared memory	Hierarchical ring	32–256	2–6 $\mu$ s
CM-5	Message passing	Fat tree	32–1024	10 $\mu$ s
Intel Paragon	Message passing	2D mesh	32–2048	10–30 $\mu$ s
IBM SP-2	Message passing	Multistage switch	2–512	30–100 $\mu$ s

<b>Application</b>	<b>Scaling of computation</b>	<b>Scaling of communication</b>	<b>Scaling of computation-to-communication</b>
FFT	$\frac{n \log n}{p}$	$\frac{n}{p}$	$\log n$
LU	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$
Barnes	$\frac{n \log n}{p}$	Approximately $\frac{\sqrt{n} (\log n)}{\sqrt{p}}$	Approximately $\frac{\sqrt{n}}{\sqrt{p}}$
Ocean	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$

**Scaling of computation, of communication, and of the ratio are critical factors in determining performance on parallel machines**

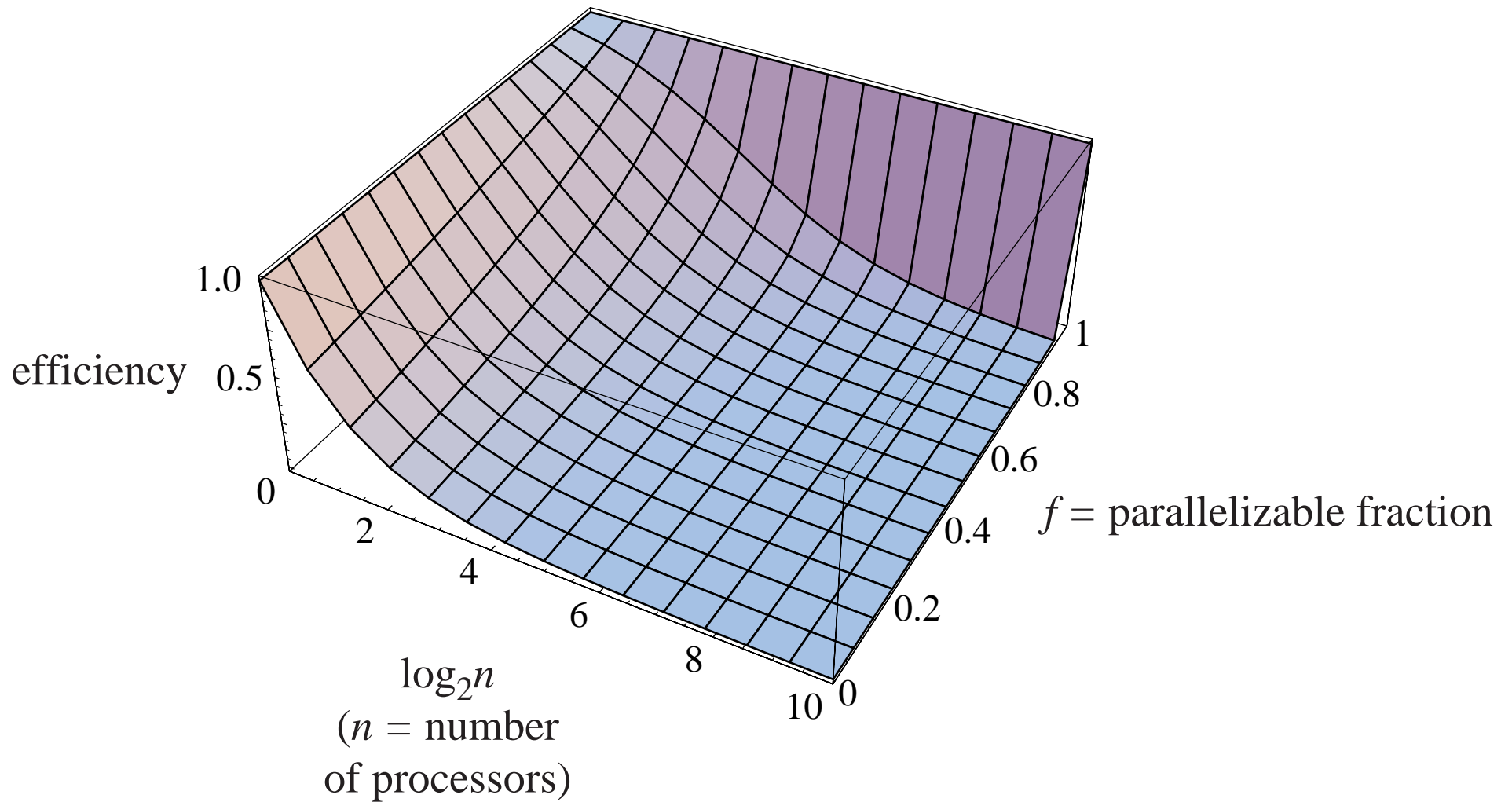
# Scalable vs. non-scalable systems





## Amdahl's law for parallel processing

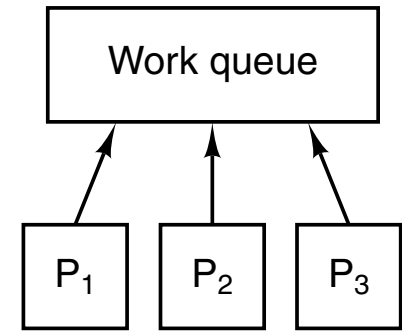
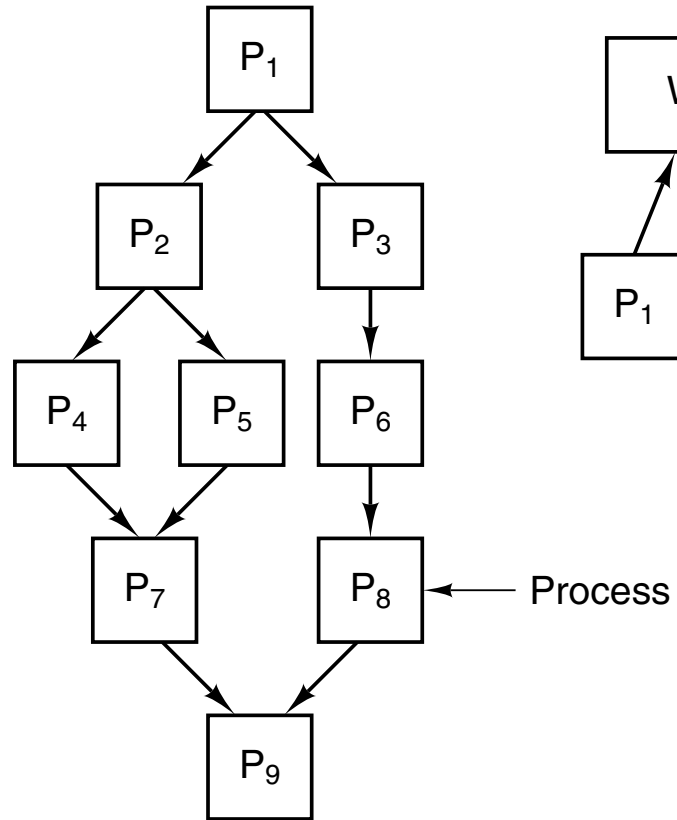
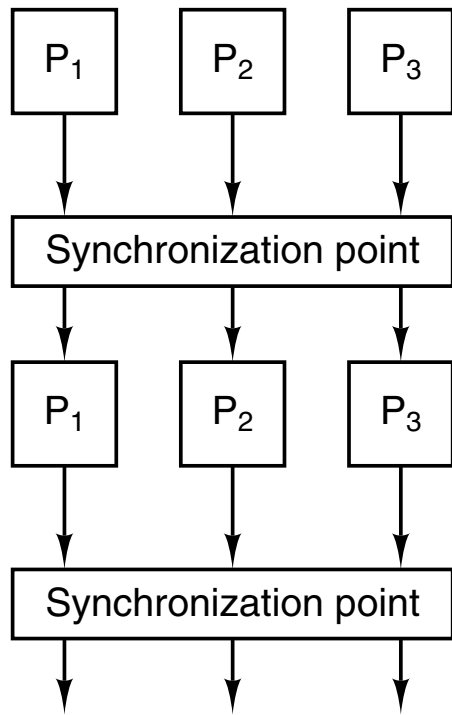
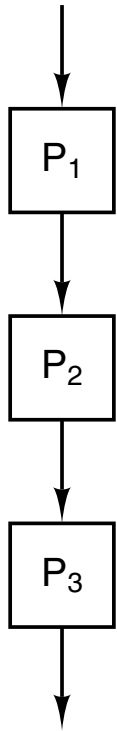
# AMDAHL'S LAW FOR PARALLEL COMPUTING

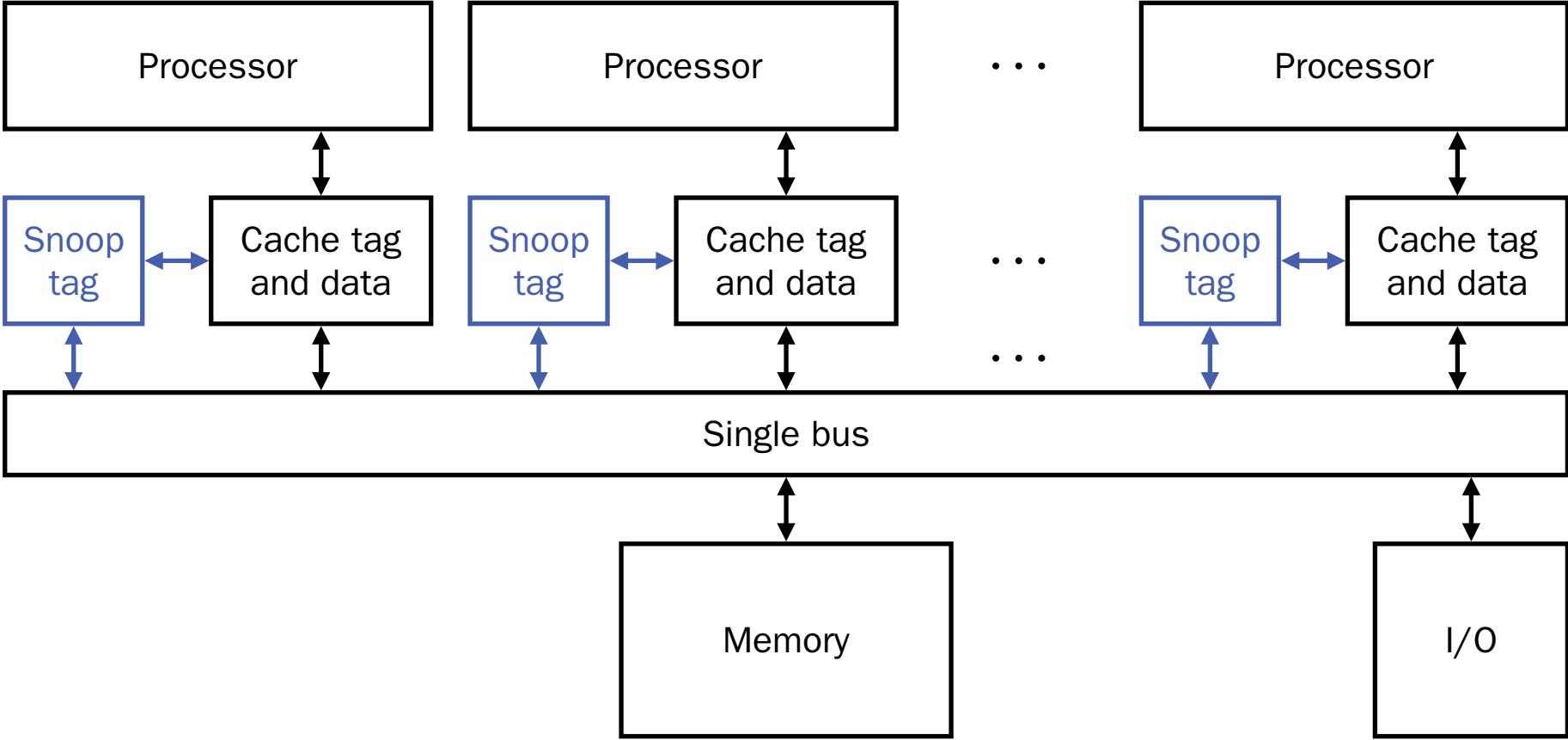


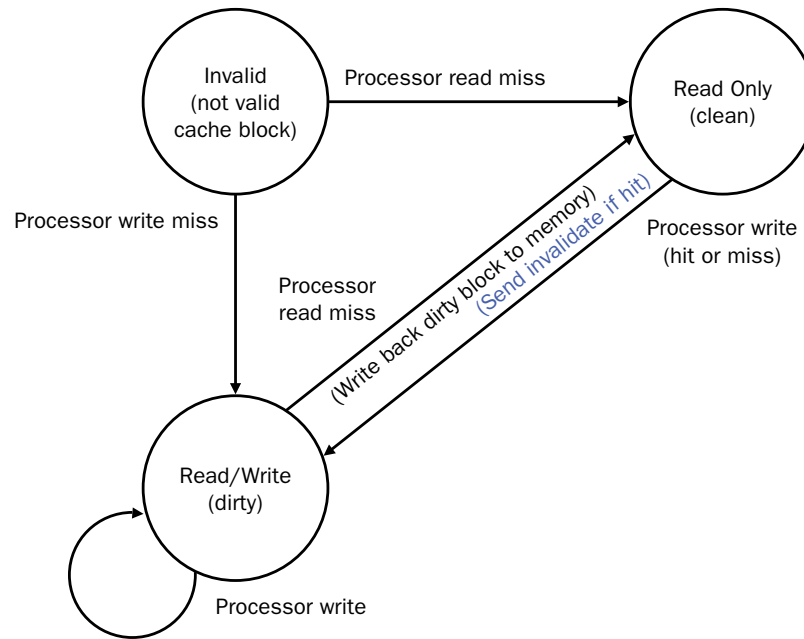
## THREADS

- Threads are the software equivalent of hardware functional units
- Properties of threads:
  - ▷ Exist in user process space or kernel space
    - User threads are faster to launch than processes
  - ▷ Mappings required for execution:  
user thread → lightweight process → kernel thread → CPU
- Reference: Bil Lewis and Daniel J. Berg, *Threads Primer*  
(SunSoft/Prentice Hall, 1996)

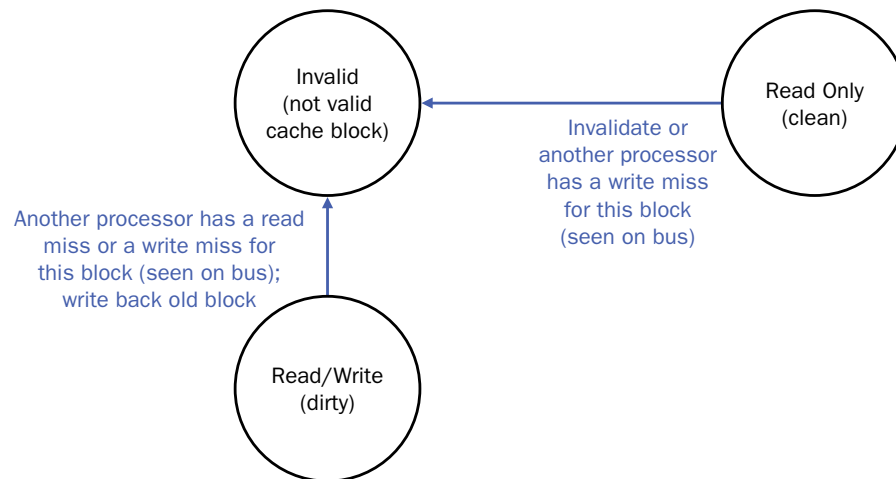
# Computational paradigms





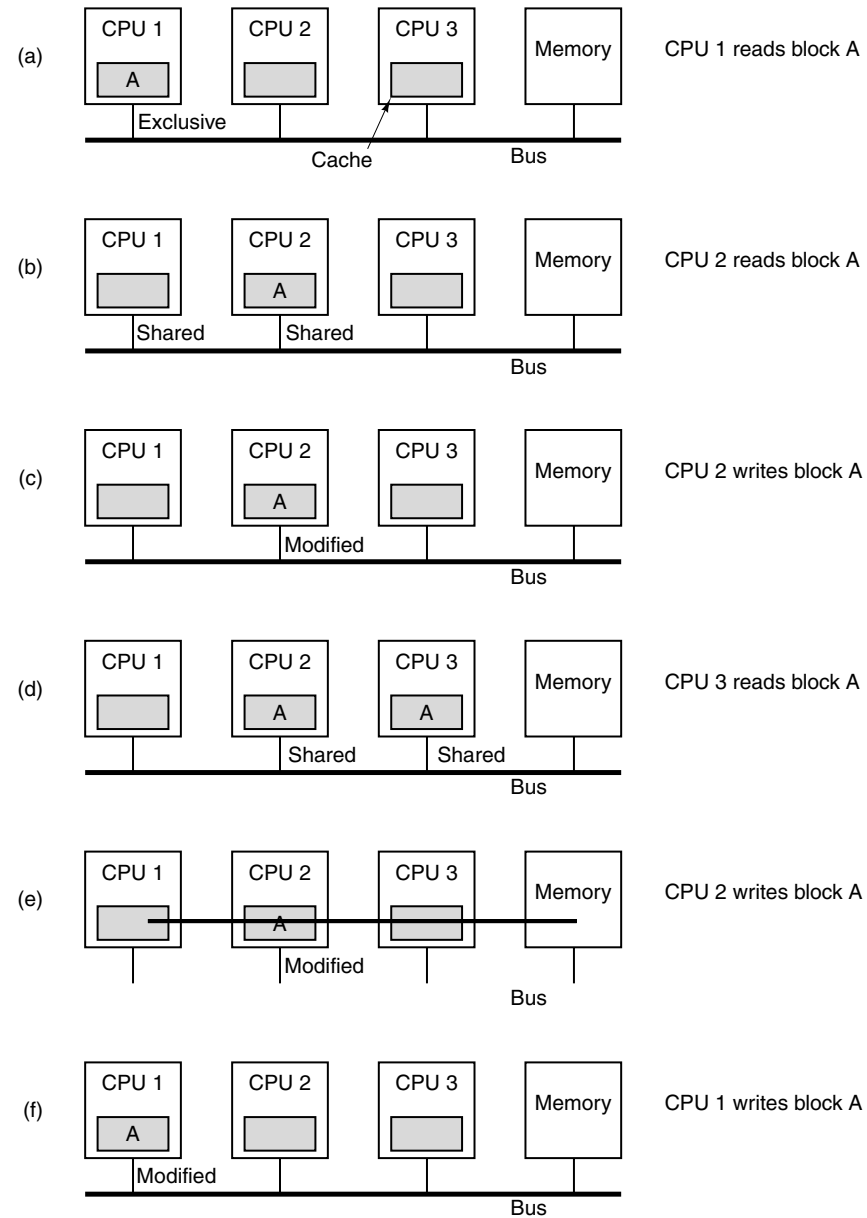


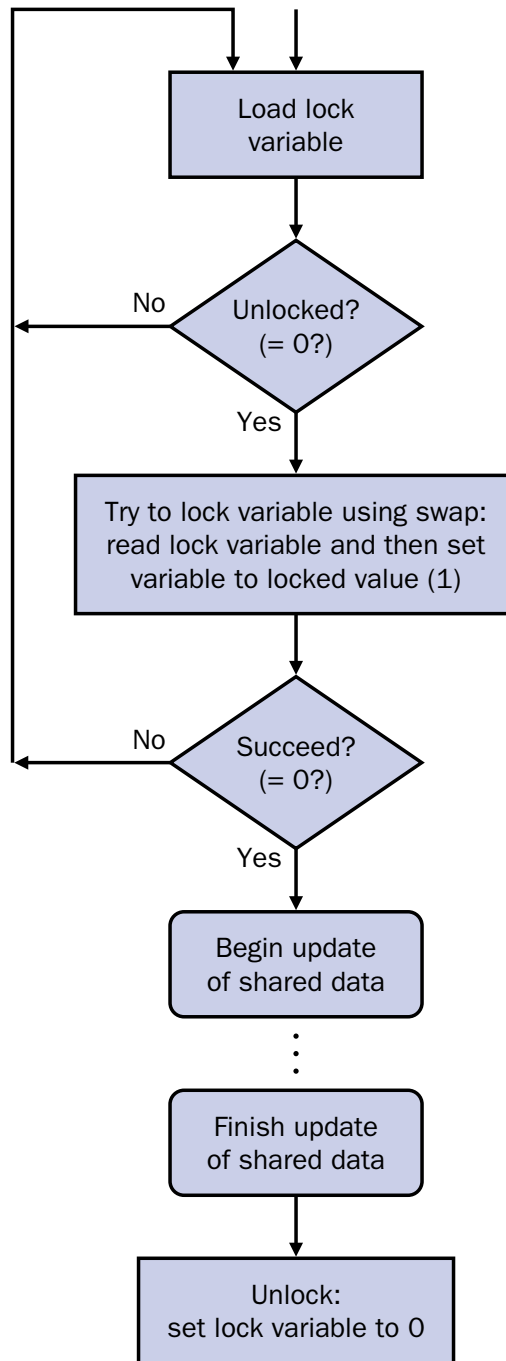
a. Cache state transitions using signals from the processor



b. Cache state transitions using signals from the bus

# The MESI cache coherence protocol

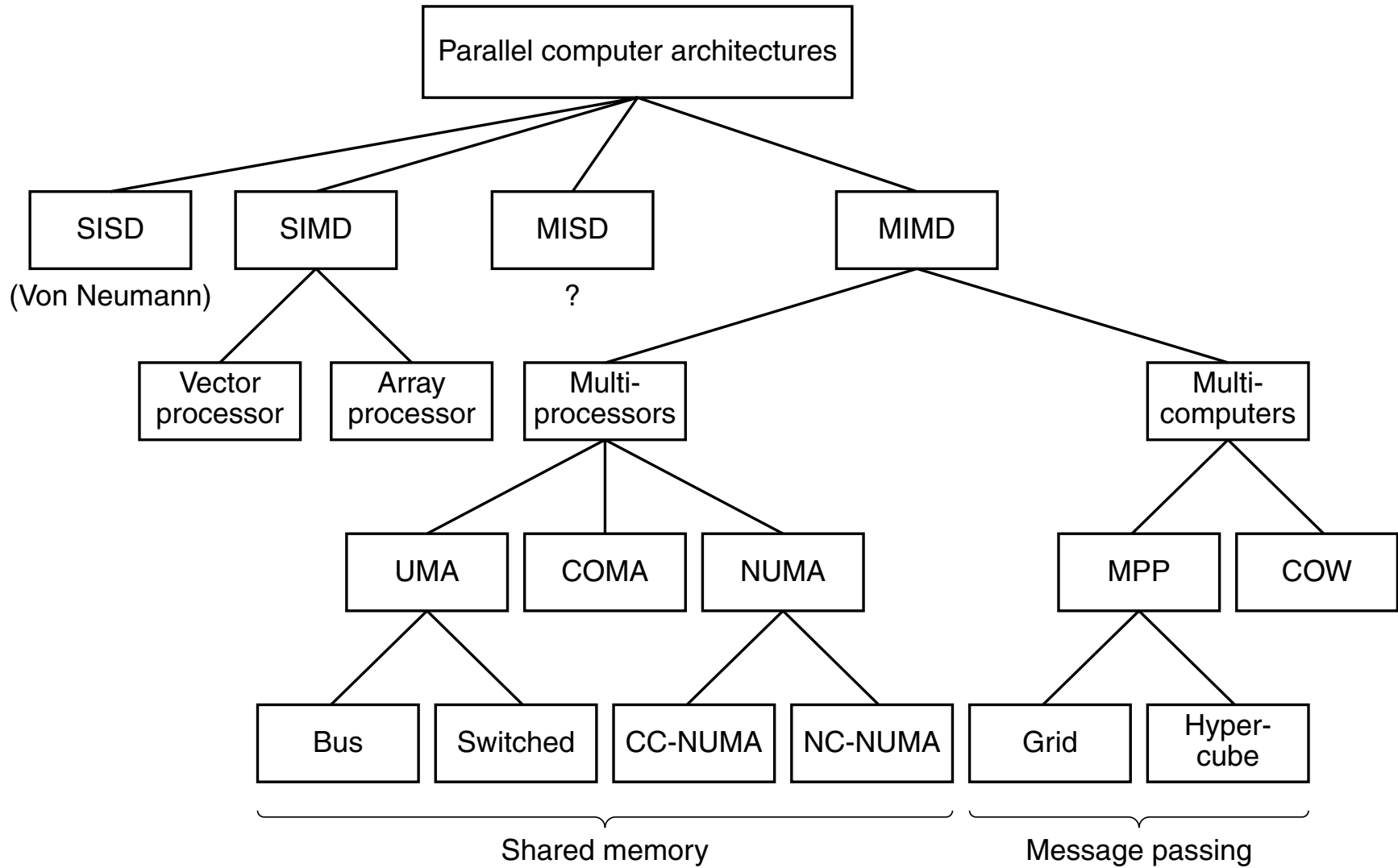




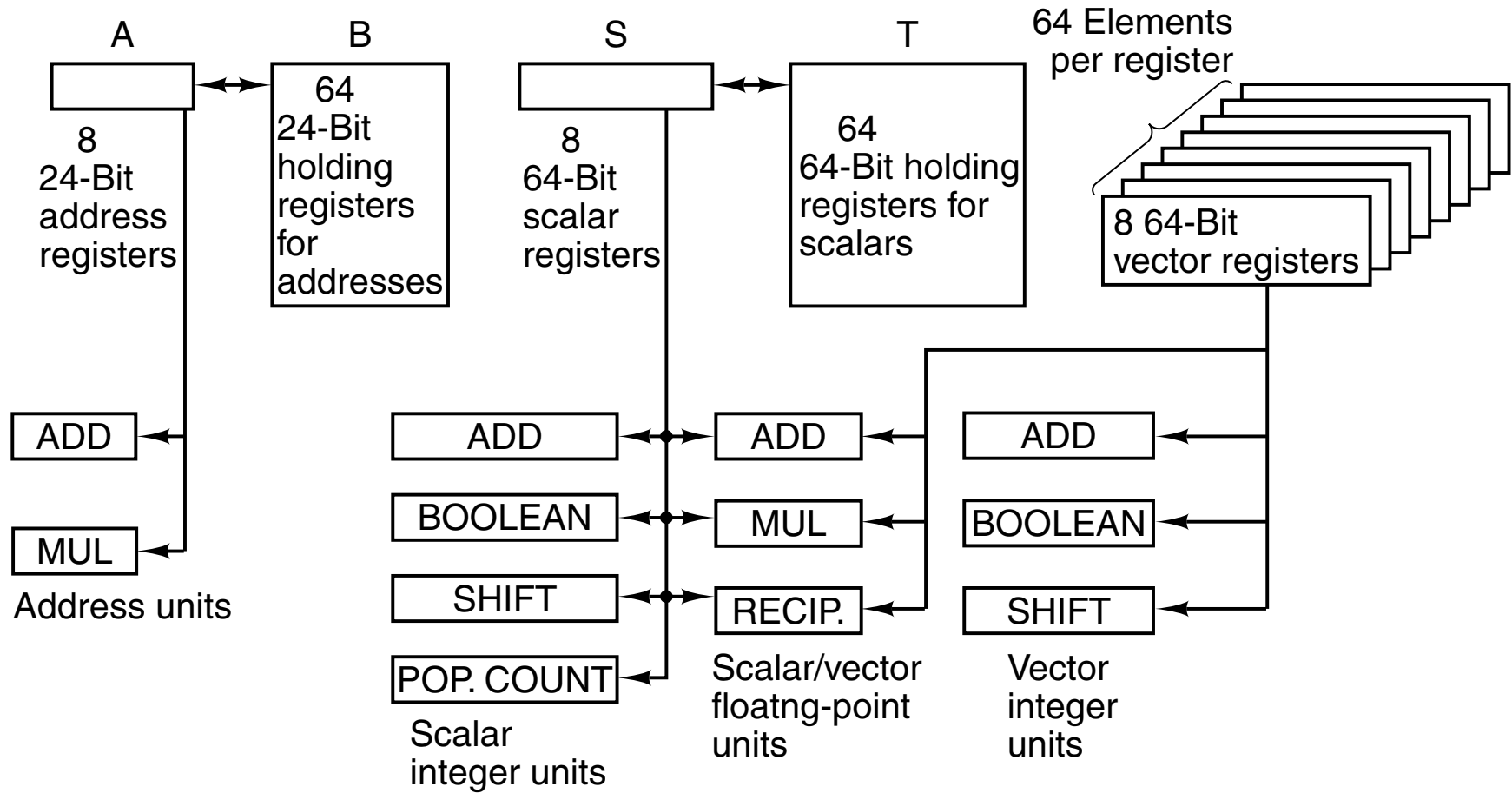
# Flynn's taxonomy of parallel computers

<b>Instruction streams</b>	<b>Data streams</b>	<b>Name</b>	<b>Examples</b>
1	1	SISD	Classical Von Neumann machine
1	Multiple	SIMD	Vector supercomputer, array processor
Multiple	1	MISD	Arguably none
Multiple	Multiple	MIMD	Multiprocessor, multicomputer

# A taxonomy of parallel computers



# CRAY-1 registers and functional units



## COMBINED VECTOR AND SCALAR OPERATIONS

- SAXPY (Scalar A times X Plus Y)

$$\mathbf{z} = \alpha \mathbf{x} + \mathbf{y}$$

- GAXPY (General A times X Plus Y)

$$\mathbf{z} = \mathbf{A} \mathbf{x} + \mathbf{y}$$

## VECTORIZATION INHIBITORS

- Loops containing
  - ▷ Dependences involving an array
  - ▷ Recursion
  - ▷ I/O statements
  - ▷ Function or subroutine calls
  - ▷ References to a nonvectorizable statement function
  - ▷ A limit or stride that varies with respect to an outer loop
  - ▷ Multiple entrances or exits
  - ▷ Equivalenced variables
  - ▷ Computed or assigned GO TO statements
  - ▷ Vector reduction operations (on some vector computers)
  - ▷ Nonlinear array references to memory
  - ▷ Ambiguous subscripts

**IMPORTANCE OF VECTOR LENGTH**

- Suppose that the vector length is  $l_v$ , *i.e.*  $l_v$  vectors can be held in hardware registers.
- If we need  $s$  cycles to set up a vector pipeline, then we need  $s + n$  cycles to perform  $n$  operations if  $n \leq l_v$ .
- If  $n > l_v$ , we need  $n + s(\lfloor n/l_v \rfloor + 1)$  cycles to perform  $n$  operations. This reduces the rate of computation by the factor

$$L(n) = \frac{1}{1 + \frac{s}{n} (\lfloor n/l_v \rfloor + 1)}.$$

- The factor  $n_{1/2}$ , such that

$$L(n_{1/2}) = \frac{1}{2},$$

measures the startup overhead. A computer with a large value of  $n_{1/2}$  performs poorly in calculations on short vectors.

## VECTOR OPTIMIZATION TECHNIQUES (1)

- Design the algorithm to yield an optimized, vectorizable innermost DO loop:
  - ▷ Avoid nonvectorizable constructions
  - ▷ Avoid scalar temporary variables
  - ▷ Use unit-stride memory references
  - ▷ Unroll loops if there is only one pipe to memory
  - ▷ Distribute loops
  - ▷ Put longest vectors in innermost loop
  - ▷ Reorder loops or statements to avoid recurrence on innermost loop
- Use parallel operations or chaining to perform as much computation as possible in each vector period or cycle
- Use strip mining to minimize vector load overhead
- Minimize subroutine or function call overhead (several  $\times 10^2$  CPU cycles per call)

## VECTOR OPTIMIZATION TECHNIQUES (2)

- Avoid scalar temporary variables in a vector loop. Scalar temporaries force unnecessary scalar stores to memory, and may inhibit chaining on CRAYs.

▷ Slow matrix multiplication inner loop:

```
SUM = 0.  
DO 100 K = 1, N  
    SUM = SUM + A(J, K) * B(K, I)  
100 CONTINUE  
C(I, J) = SUM
```

▷ Faster loop:

```
C(I, J) = 0.  
DO 100 K = 1, N  
    C(I, J) = C(I, J) + A(J, K) * B(K, I)  
100 CONTINUE
```

▷ Some processors vectorize reduction operations such as the above dot product

**VECTOR OPTIMIZATION TECHNIQUES (3)**

- Use unit-stride memory references in the innermost loop to take maximum advantage of bank memory (if present), or if they will permit cache bypass:

- ▷ Non-unit-stride innermost loop:

```
DO 20 I = 1, M
  DO 20 J = 1, N
    A(I, J) = B(I, J) * C(I, J)
  20 CONTINUE
  C(I, J) = SUM
```

- ▷ Interchange loops to obtain unit stride:

```
DO 20 J = 1, N
  DO 20 I = 1, M
    A(I, J) = B(I, J) * C(I, J)
  20 CONTINUE
```

- ▷ Some compilers perform this optimization

**VECTOR OPTIMIZATION TECHNIQUES (4)**

- On a system with interleaved memory, avoid using strides through memory that have common factors with the number of memory banks

▷ Original loop:

```
DIMENSION A(32, N), B(32, N)
DO 100 I = 1, 32
  DO 100 J = 1, N
    A(I, J) = A(I, J) + B(I, J)
100 CONTINUE
```

- ▷ The inner loop has stride 32, which causes a bank conflict at every memory access on a machine with 32 banks (*e.g.*, CRAY X-MPs).

```
DIMENSION A(32, N), B(32, N)
DO 100 J = 1, N
  DO 100 I = 1, 32
    A(I, J) = A(I, J) + B(I, J)
100 CONTINUE
```

**VECTOR OPTIMIZATION TECHNIQUES (5)**

- Unroll an outer loop if the processor has only one vector pipeline between memory and the CPU (*e.g.*, the CRAY-1)
  - ▷ Unrolling can decrease conflicts between loads and stores
  - ▷ Dongarra and Eisenstat's SAXPY for  $\mathbf{y} = \mathbf{A}\mathbf{x}$ :

```
DO 20 J = 1, N
  DO 10 I = 1, M
    Y(I) = Y(I) + X(J) * A(I, J)
  10 CONTINUE
20 CONTINUE
```

- ▷ Each instance of the inner loop requires 2 vector loads, a (possibly chained) multiply-add, and a vector store, or a total of 3 memory references for 2 FLOPs

**VECTOR OPTIMIZATION TECHNIQUES (6)**

- Unroll the outer loop of Dongarra and Eisenstat's SAXPY to a depth of four:

```
DO 20 J = 1, N, 4
  DO 10 I = 1, M
    Y(I) = ((Y(I) + X(J-3) * M(I, J-3))
*          + X(J-2) * M(I, J-2))
*          + X(J-1) * M(I, J-1))
*          + X(J) * M(I, J)
  10 CONTINUE
20 CONTINUE
```

- Now each instance of the inner loop makes 6 memory references for 8 FLOPs, so that on a single-vector-pipe processor the speed is twice that of the original loop

**VECTOR OPTIMIZATION TECHNIQUES (7)**

- Distribute a DO loop for partial avoidance of recursion

▷ Original loop:

```
DO 20 I = 1, M
  A(I) = A(I-1) + B(I) * C(I)
20 CONTINUE
```

▷ Distributed loop:

```
DO 10 I = 1, M
  T(I) = B(I) * C(I)
10 CONTINUE
DO 20 I = 1, M
  A(I) = A(I-1) + T(I)
20 CONTINUE
```

▷ Some compilers perform such partial vectorizations automatically

**VECTOR OPTIMIZATION TECHNIQUES (8)**

- Distribute nested DO loops to enhance vectorization

▷ Original loops:

```
DO 20 I = 1, M
  DO 10 J = 1, N
    A(I) = A(I) + B(I, J) * C(I, J)
10  CONTINUE
    D(I) = E(I) + A(I)
20 CONTINUE
```

▷ Distributed and reordered loops:

```
DO 20 J = 1, N
  DO 10 I = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
10  CONTINUE
20 CONTINUE
DO 30 I = 1, M
  D(I) = E(I) + A(I)
30 CONTINUE
```

**VECTOR OPTIMIZATION TECHNIQUES (9)**

- Reorder loops to avoid multidimensional recursion

▷ Original loops:

```
DO 100 I = 1, M
  DO 100 J = 1, N
    A(I, J) = A(I, J-1) * B(I, J)
100 CONTINUE
```

▷ Reordered loops:

```
DO 100 J = 1, N
  DO 100 I = 1, M
    A(I, J) = A(I, J-1) * B(I, J)
100 CONTINUE
```

**VECTOR OPTIMIZATION TECHNIQUES (10)**

- Use **strip mining** to minimize vector load overhead

▷ Original loop:

```
DO 100 I = 1, M
  DO 100 J = 1, N
    A(I) = B(I) * C(I)
100 CONTINUE
```

▷ Strip-mined loop (assuming a processor with vector registers of length 128):

```
J = 0
DO 110 K = M, 0, -128
  DO 100 I = 1, 128
    A(I+J) = B(I+J) * C(I+J)
100 CONTINUE
  J = J + 128
110 CONTINUE
```

▷ If the upper limit  $M$  is not a multiple of 128, there will be some overhead in cleaning up the last part of the loop.

---

# Strip Mining

```
DO I = 1, 10000
  A(I) = A(I) * B(I)
ENDDO
```

Becomes...

```
DO IOUSER = 1, 10000, 1000
  DO ISTRIP = IOUSER, IOUSER+999
    A(ISTRIP) = A(ISTRIP) * B(ISTRIP)
  ENDDO
ENDDO
```

# Loop Interchange

```
DO I = 1, N
  DO J = 1, N
    A(I,J) = B(I,J) + C(I,J)
  ENDDO
ENDDO
```

Becomes...

```
DO J = 1, N
  DO I = 1, N
    A(I,J) = B(I,J) + C(I,J)
  ENDDO
ENDDO
```

---

## ARRAY DEPENDENCIES (1)

- Reference to a value calculated in a previous instance of the loop

▷ Nonvectorizable construction:

```
DO 100 I = 2, 100
  B(I) = A(I-1)
  A(I) = C(I)
100 CONTINUE
```

- ▷ If all of the elements of A, B and C were operated on in groups, then execution of  $B(3) = A(2)$  would occur before execution of  $A(2) = C(2)$ .
- ▷ Vectorization is possible after interchanging the statements:

```
DO 100 I = 2, 100
  A(I) = C(I)
  B(I) = A(I-1)
100 CONTINUE
```

- ▷ Some vectorizing FORTRAN compilers make such statement interchanges automatically

## ARRAY DEPENDENCIES (2)

- Reference to a value that should be calculated in a later instance of the loop

▷ Nonvectorizable construction:

```
DO 100 I = 1, 99
  A(I) = C(I)
  B(I) = A(I+1)
100 CONTINUE
```

▷ If all of the elements of A, B and C were operated on in groups, then execution of  $A(2) = C(2)$  would occur before execution of  $B(1) = A(2)$ .

▷ Vectorization is possible after interchanging the statements:

```
DO 100 I = 2, 100
  B(I) = A(I+1)
  A(I) = C(I)
100 CONTINUE
```

▷ Some vectorizing FORTRAN compilers make such statement interchanges automatically