Chapter 2: Program and Network Properties

- Conditions of parallelism
- Program partitioning and scheduling
- Program flow mechanisms
- System interconnect architectures

Conditions of Parallelism

- The exploitation of parallelism in computing requires understanding the basic theory associated with it. Progress is needed in several areas:
  - computation models for parallel computing
  - interprocessor communication in parallel architectures
  - integration of parallel systems into general environments

Data dependences

The ordering relationship between statements is indicated by the data dependence.

- Flow dependence
- Anti dependence
- Output dependence
- I/O dependence
- Unknown dependence

Data Dependence - 1

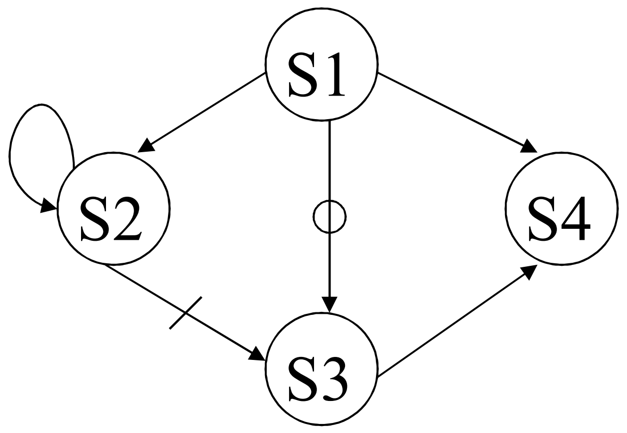
- Flow dependence: S1 precedes S2, and at least one output of S1 is input to S2.
- Antidependence: S1 precedes S2, and the output of S2 overlaps the input to S1.
- Output dependence: S1 and S2 write to the same output variable.
- I/O dependence: two I/O statements (read/write) reference the same variable, and/or the same file.

Data Dependence - 2

- Unknown dependence:
  - The subscript of a variable is itself subscripted.
  - The subscript does not contain the loop index variable.
  - A variable appears more than once with subscripts having different coefficients of the loop variable (that is, different functions of the loop variable).
  - The subscript is nonlinear in the loop index variable.
- Parallel execution of program segments which do not have total data independence can produce non-deterministic results.

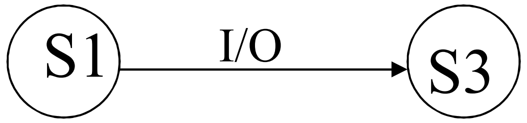
Data dependence example

S1: Load R1, A
S2: Add R2, R1
S3: Move R1, R3
S4: Store B, R1



I/O dependence example

S1: Read (4), A(I)
S2: Rewind (4)
S3: Write (4), B(I)
S4: Rewind (4)


$$\begin{tikzcd} \boxed{S1} \arrow[r, "I/O"] & \boxed{S3} \end{tikzcd}$$

Control dependence

- The order of execution of statements cannot be determined before run time
  - Conditional branches
  - Successive operations of a looping procedure

Control dependence examples

```
      Do 20 I = 1, N
      A(I) = C(I)
      IF(A(I) .LT. 0) A(I)=1
   20 Continue
```

```
      Do 10 I = 1, N
       IF(A(I-1) .EQ. 0) A(I)=0
   10 Continue
```

Resource dependence

- Concerned with the conflicts in using shared resources
  - Integer units
  - Floating-point units
  - Registers
  - Memory areas
  - ALU
  - Workplace storage

Bernstein's conditions

- Set of conditions for two processes to execute in parallel

  $I_1 \cap O_2 = \emptyset$

  $I_2 \cap O_1 = \emptyset$

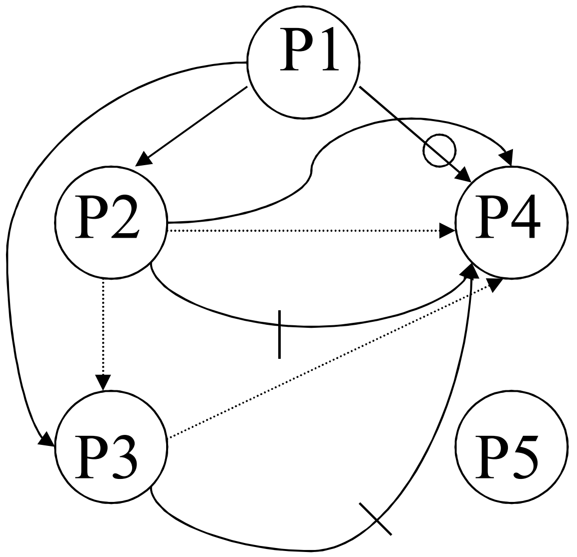
  $O_1 \cap O_2 = \emptyset$

Bernstein’s Conditions - 2

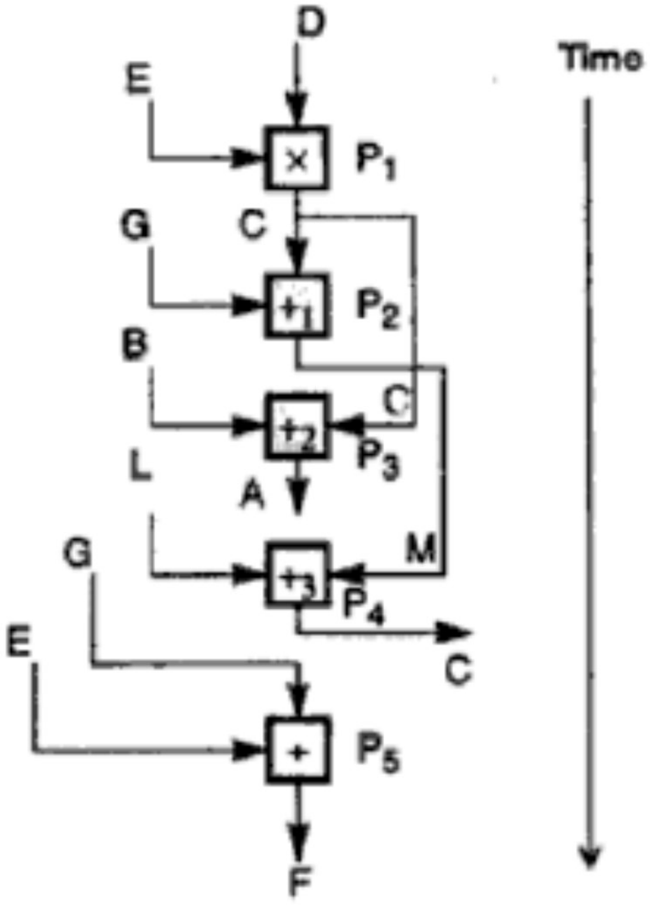
- In terms of data dependencies, Bernstein’s conditions imply that two processes can execute in parallel if they are flow-independent, antiindependent, and output-independent.
- The parallelism relation || is commutative ($P_i \,||\, P_j$ implies $P_j \,||\, P_i$), but not transitive ($P_i \,||\, P_j$ and $P_j \,||\, P_k$ does not imply $P_i \,||\, P_k$) . Therefore, || is not an equivalence relation.
- Intersection of the input sets is allowed.

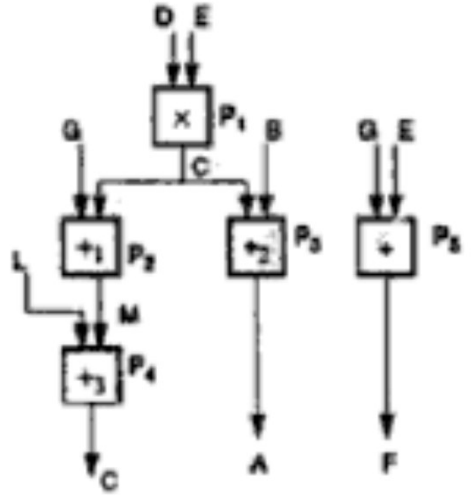
Utilizing Bernstein's conditions

P_1 :	C = D x E
P_2 :	M = G + C
P_3 :	A = B + C
P_4 :	C = L + M
P_5 :	F = G / E



Utilizing Bernstein's conditions





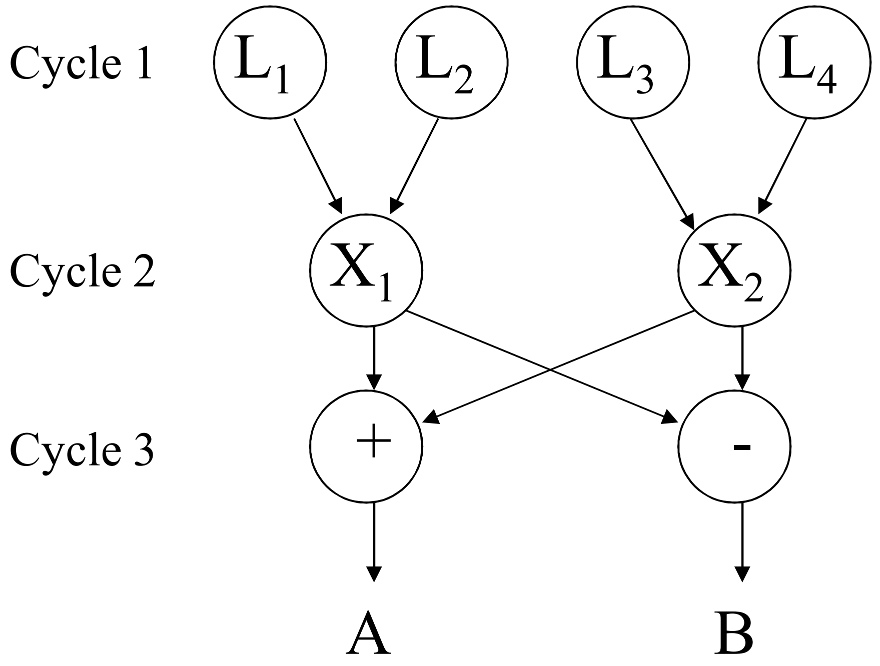
Hardware parallelism

- A function of cost and performance tradeoffs
- Displays the resource utilization patterns of simultaneously executable operations
- Denote the number of instruction issues per machine cycle: *k-issue* processor
- A multiprocessor system with *n* *k*-issue processors should be able to handle a maximum number of *nk* threads of instructions simultaneously

Software parallelism

- Defined by the control and data dependence of programs
- A function of algorithm, programming style, and compiler organization
- The program flow graph displays the patterns of simultaneously executable operations

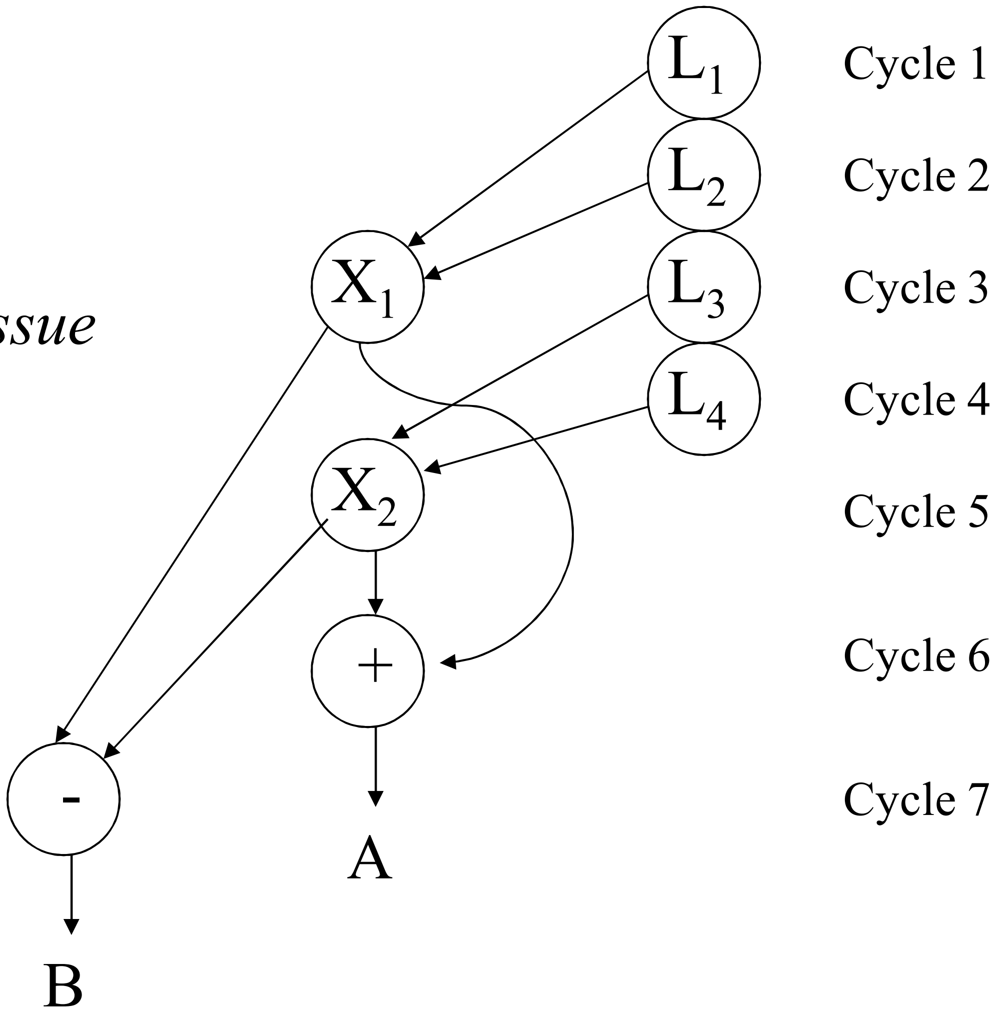
Mismatch between software and hardware parallelism - 1



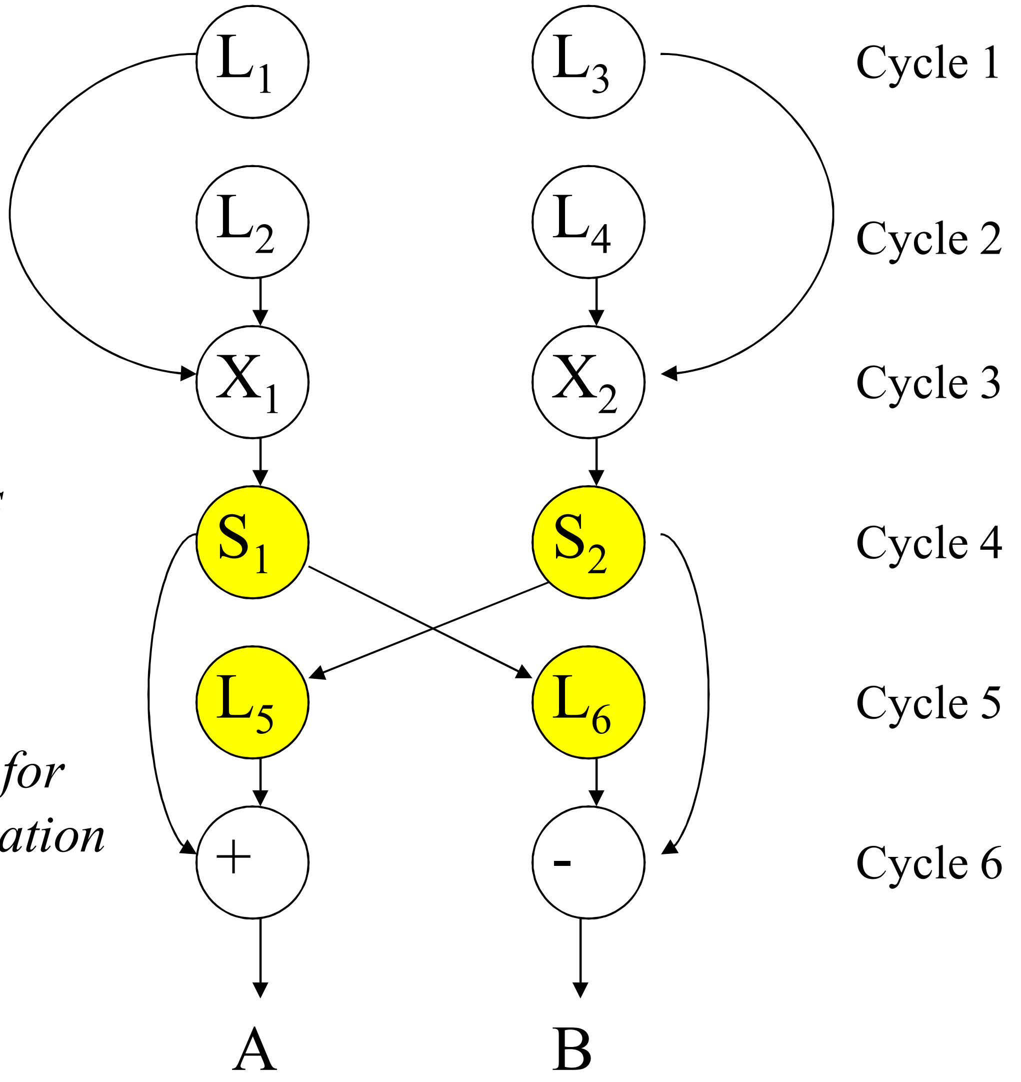
Maximum software parallelism (L=load, X/+/- = arithmetic).

Mismatch between software and hardware parallelism - 2

Same problem, but considering the parallelism on a two-issue superscalar processor.



Mismatch between software and hardware parallelism - 3



Same problem, with two single-issue processors

● *= inserted for synchronization*

Software parallelism

- Control parallelism – allows two or more operations to be performed concurrently
    - Pipelining, multiple functional units
- Data parallelism – almost the same operation is performed over many data elements by many processors concurrently
    - Code is easier to write and debug

Types of Software Parallelism

- Control Parallelism – two or more operations can be performed simultaneously. This can be detected by a compiler, or a programmer can explicitly indicate control parallelism by using special language constructs or dividing a program into multiple processes.
- Data parallelism – multiple data elements have the same operations applied to them at the same time. This offers the highest potential for concurrency (in SIMD and MIMD modes). Synchronization in SIMD machines handled by hardware.

Solving the Mismatch Problems

- Develop compilation support
- Redesign hardware for more efficient exploitation by compilers
- Use large register files and sustained instruction pipelining.
- Have the compiler fill the branch and load delay slots in code generated for RISC processors.

The Role of Compilers

- Compilers used to exploit hardware features to improve performance.
- Interaction between compiler and architecture design is a necessity in modern computer development.
- It is not necessarily the case that more software parallelism will improve performance in conventional scalar processors.
- The hardware and compiler should be designed at the same time.

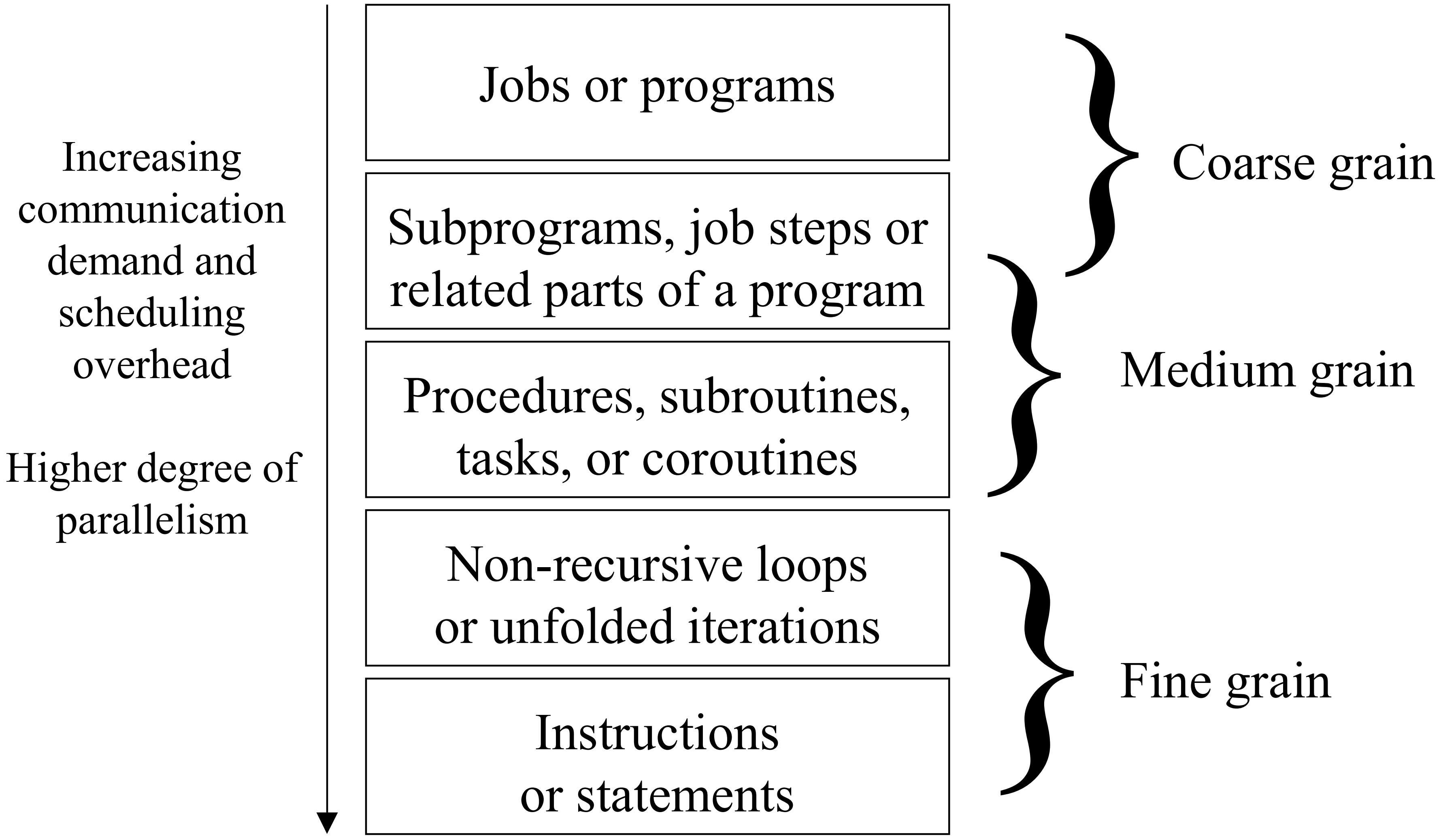
Program Partitioning & Scheduling

- The size of the parts or pieces of a program that can be considered for parallel execution can vary.
- The sizes are roughly classified using the term “granule size,” or simply “granularity.”
- The simplest measure, for example, is the number of instructions in a program part.
- Grain sizes are usually described as *fine*, *medium* or *coarse*, depending on the level of parallelism involved.

Latency

- *Latency* is the time required for communication between different subsystems in a computer.
- *Memory latency*, for example, is the time required by a processor to access memory.
- *Synchronization latency* is the time required for two processes to synchronize their execution.
- Computational granularity and communicatoin latency are closely related.

Levels of Parallelism



Increasing communication demand and scheduling overhead
Higher degree of parallelism
Jobs or programs
Subprograms, job steps or related parts of a program
Procedures, subroutines, tasks, or coroutines
Non-recursive loops or unfolded iterations
Instructions or statements
Coarse grain
Medium grain
Fine grain

Instruction Level Parallelism

- This fine-grained, or smallest granularity level typically involves less than 20 instructions per grain. The number of candidates for parallel execution varies from 2 to thousands, with about five instructions or statements (on the average) being the average level of parallelism.
- Advantages:
  - There are usually many candidates for parallel execution
  - Compilers can usually do a reasonable job of finding this parallelism

Loop-level Parallelism

- Typical loop has less than 500 instructions.
- If a loop operation is independent between iterations, it can be handled by a pipeline, or by a SIMD machine.
- Most optimized program construct to execute on a parallel or vector machine
- Some loops (e.g. recursive) are difficult to handle.
- Loop-level parallelism is still considered fine grain computation.

Procedure-level Parallelism

- Medium-sized grain; usually less than 2000 instructions.
- Detection of parallelism is more difficult than with smaller grains; interprocedural dependence analysis is difficult and history-sensitive.
- Communication requirement less than instruction-level
- SPMD (single procedure multiple data) is a special case
- Multitasking belongs to this level.

Subprogram-level Parallelism

- Job step level; grain typically has thousands of instructions; medium- or coarse-grain level.
- Job steps can overlap across different jobs.
- Multiprograming conducted at this level
- No compilers available to exploit medium- or coarse-grain parallelism at present.

Job or Program-Level Parallelism

- Corresponds to execution of essentially independent jobs or programs on a parallel computer.
- This is practical for a machine with a small number of powerful processors, but impractical for a machine with a large number of simple processors (since each processor would take too long to process a single job).

Communication Latency

- Balancing granularity and latency can yield better performance.
- Various latencies attributed to machine architecture, technology, and communication patterns used.
- Latency imposes a limiting factor on machine scalability. Ex. Memory latency increases as memory capacity increases, limiting the amount of memory that can be used with a given tolerance for communication latency.

Interprocessor Communication Latency

- Needs to be minimized by system designer
- Affected by signal delays and communication patterns
- Ex. n communicating tasks may require $n(n-1)/2$ communication links, and the complexity grows quadratically, effectively limiting the number of processors in the system.

Communication Patterns

- Determined by algorithms used and architectural support provided
- Patterns include
  - permutations
  - broadcast
  - multicast
  - conference
- Tradeoffs often exist between granularity of parallelism and communication demand.

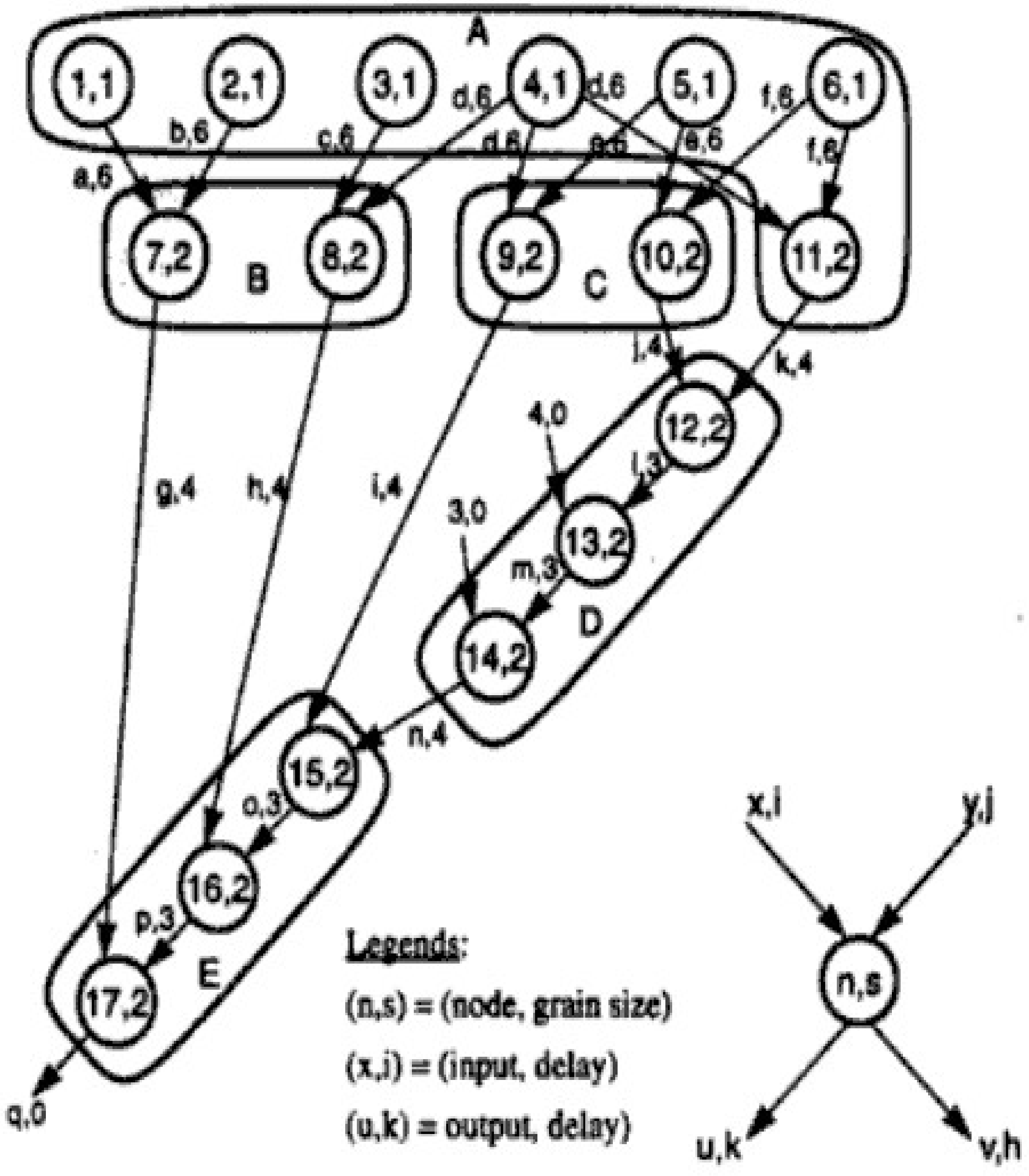
Grain Packing and Scheduling

- Two questions:
  - How can I partition a program into parallel “pieces” to yield the shortest execution time?
  - What is the optimal size of parallel grains?
- There is an obvious tradeoff between the time spent scheduling and synchronizing parallel grains and the speedup obtained by parallel execution.
- One approach to the problem is called “grain packing.”

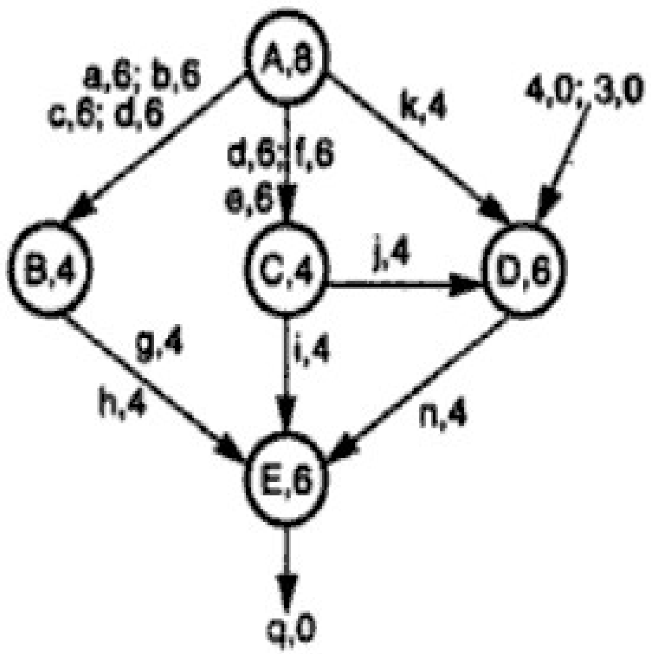
Program Graphs and Packing

- A program graph is similar to a dependence graph
  - Nodes = { (n,s) }, where n = node name, s = size (larger s = larger grain size).
  - Edges = { (v,d) }, where v = variable being "communicated," and d = communication delay.
- Packing two (or more) nodes produces a node with a larger grain size and possibly more edges to other nodes.
- Packing is done to eliminate unnecessary communication delays or reduce overall scheduling overhead.

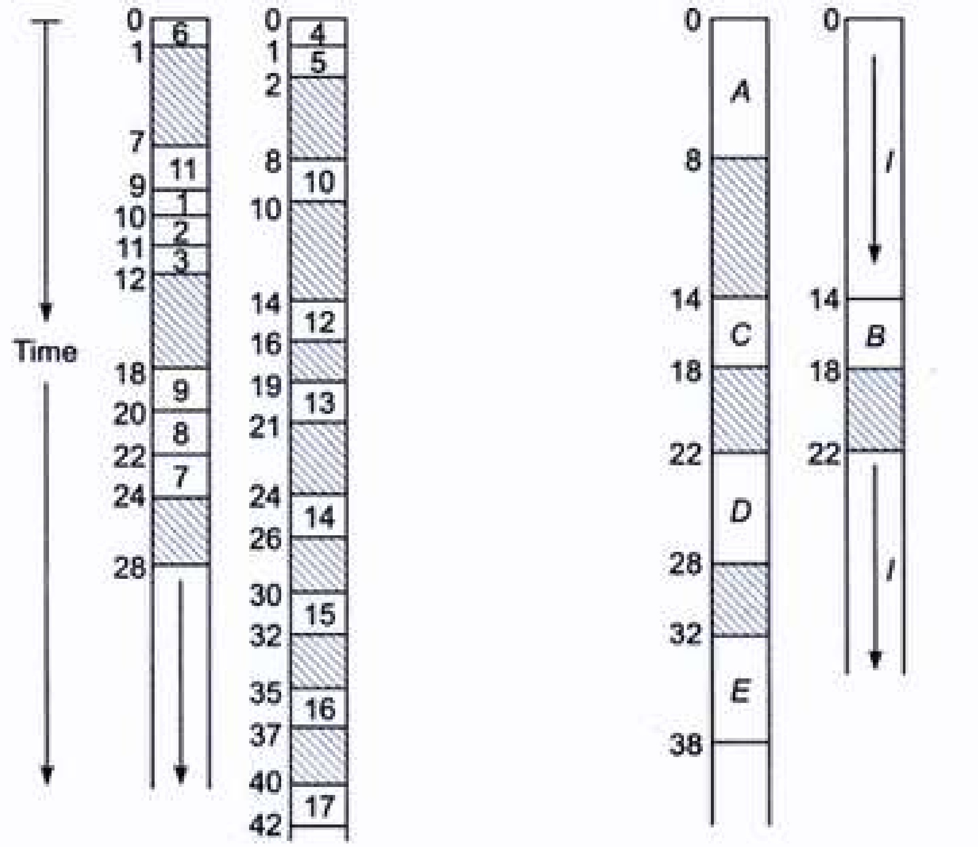
```
Var $a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q$
Begin
    1.  $a := 1$
    2.  $b := 2$
    3.  $c := 3$
    4.  $d := 4$
    5.  $e := 5$
    6.  $f := 6$
    7.  $g := a \times b$
    8.  $h := c \times d$
    9.  $i := d \times e$
    10. $j := e \times f$
    11. $k := d \times f$
    12. $l := j \times k$
    13. $m := 4 \times l$
    14. $n := 3 \times m$
    15. $o := n \times i$
    16. $p := o \times h$
    17. $q := p \times q$
End
```



(a) Fine-grain program graph before packing



(b) Coarse-grain program graph after packing



(a) Fine grain (Fig. 2.6a) (b) Coarse grain (Fig. 2.6b)

Fig. 2.7 Scheduling of the fine-grain and coarse-grain programs (arrows: idle time; shaded areas: communication delays)

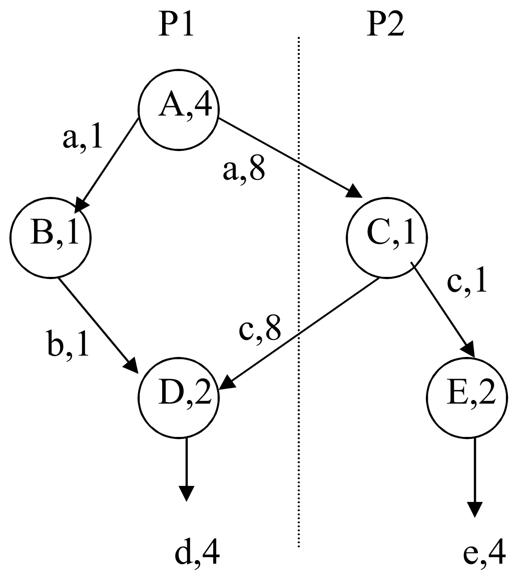
Static multiprocessor scheduling

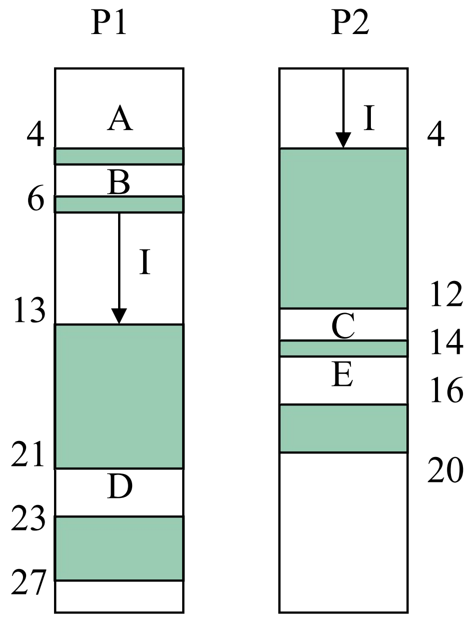
- Grain packing may not be optimal
- Dynamic multiprocessor scheduling is an NP-hard problem
- Node duplication is a static scheme for multiprocessor scheduling

Node duplication

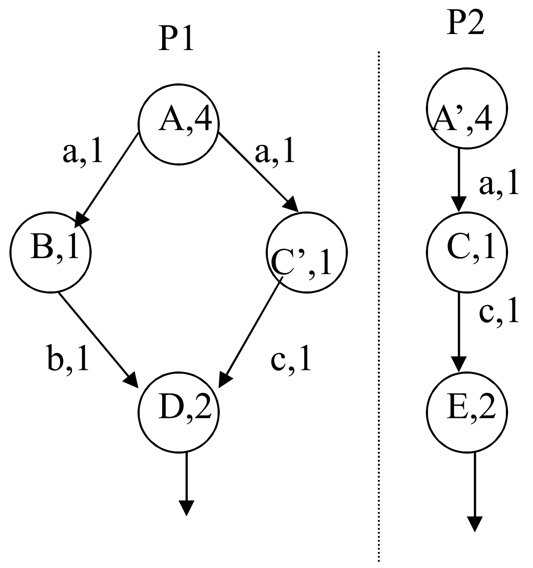
- Duplicate some nodes to eliminate idle time and reduce communication delays
- Grain packing and node duplication are often used jointly to determine the best grain size and corresponding schedule

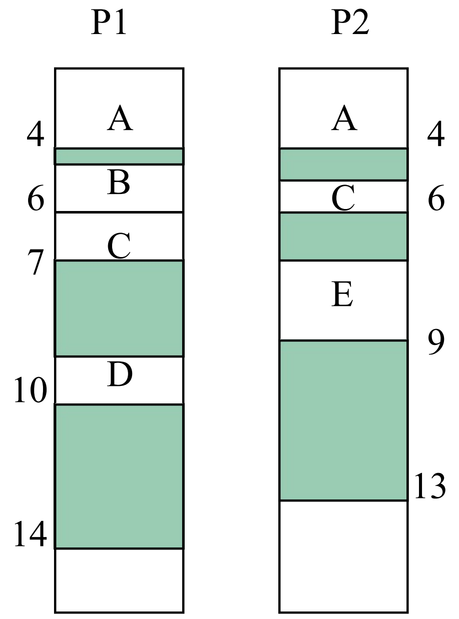
Schedule without node duplication





Schedule with node duplication





Scheduling

- A schedule is a mapping of nodes to processors and start times such that communication delay requirements are observed, and no two nodes are executing on the same processor at the same time.
- Some general scheduling goals
  - Schedule all fine-grain activities in a node to the same processor to minimize communication delays.
  - Select grain sizes for packing to achieve better schedules for a particular parallel machine.

Node Duplication

- Grain packing may potentially eliminate interprocessor communication, but it may not always produce a shorter schedule (see figure 2.8 (a)).
- By duplicating nodes (that is, executing some instructions on multiple processors), we may eliminate some interprocessor communication, and thus produce a shorter schedule.

Grain determination and scheduling optimization

Four major steps are involved in the grain determination and the process of scheduling optimization:

Step 1: Construct a fine-grain program graph

Step 2: Schedule the fine-grain computation

Step 3: Grain packing to produce coarse grains

Step 4: Generate a parallel schedule based on the packed graph

Example 2.5

- Example 2.5 illustrates a matrix multiplication program requiring 8 multiplications and 7 additions.
- Using various approaches, the program requires:
  - 212 cycles (software parallelism only)
  - 864 cycles (sequential program on one processor)
  - 741 cycles (8 processors) - speedup = 1.16
  - 446 cycles (4 processors) - speedup = 1.94

Program decomposition for static multiprocessor scheduling

- two 2 x 2 matrices *A* and *B* are multiplied to compute the sum of the four elements in the resulting product matrix *C* = *A x B*. There are eight multiplications and seven additions to be performed in this program, as written below:

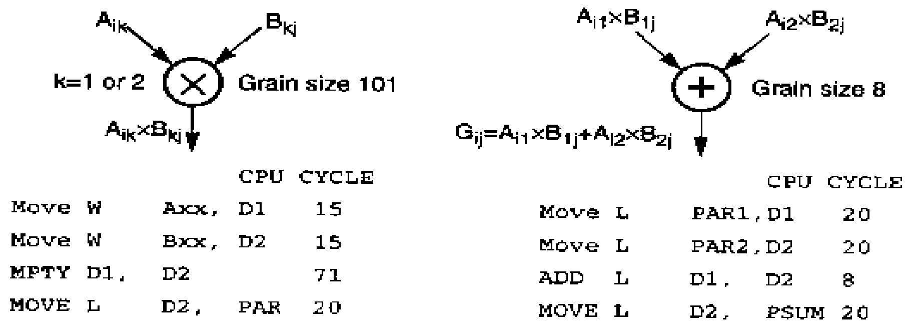
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Example 2.5 Ctd’

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

- $C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$
- $C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$
- $C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$
- $C_{22} = A_{21} \times B_{11} + A_{22} \times B_{22}$
- $Sum = C_{11} + C_{12} + C_{21} + C_{22}$

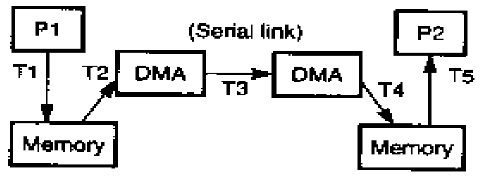
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$



```
                          CPU  CYCLE
Move W      Axx,  D1       15
Move W      Bxx,  D2       15
MPTY D1,    D2             71
MOVE L      D2,   PAR      20
```

```
                          CPU  CYCLE
Move  L     PAR1,D1        20
Move  L     PAR2,D2        20
ADD   L     D1,   D2       8
MOVE  L     D2,   PSUM     20
```

(a) Grain size calculation in M68000 assembly code at 20-MHz cycle

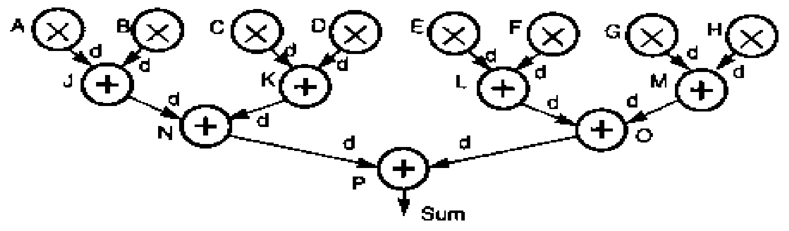


d = T1+T2+T3 +T4+T5+T6
= 20+20+32+20+20+100
= 212 cycles

T3 = 32-bit transmission time at 20 Mbps normalized to M68000 cycle at 20 MHz.

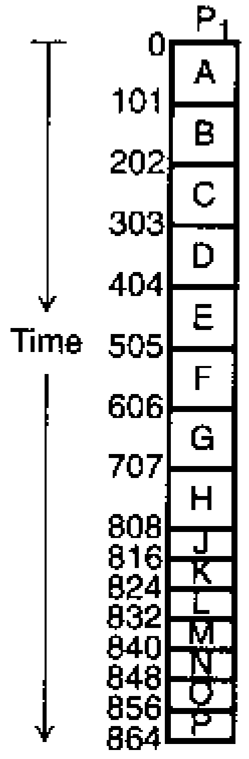
T6 = delay due to software protocols (assume 5 Move instructions, 100)

(b) Calculation of communication delay d

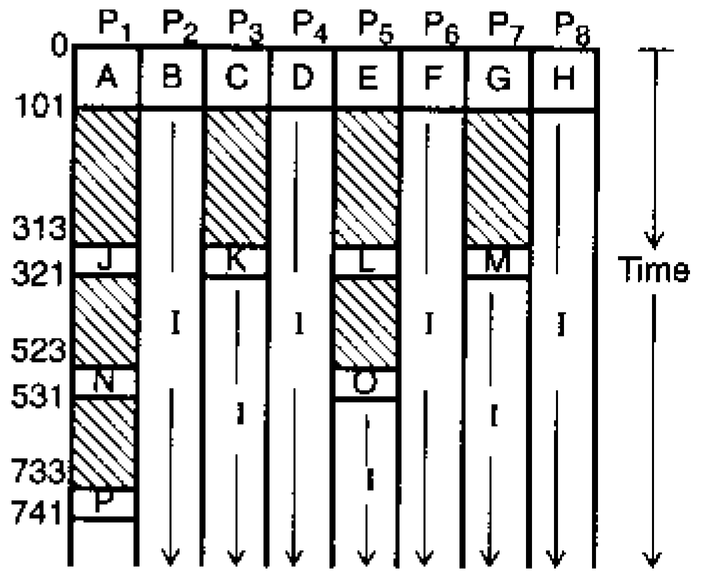


(c) Fine-grain program graph

Figure 2.9 Calculation of grain size and communication delay for the program graph in Example 2.5. (Courtesy of Kruatrachue and Lewis; reprinted with permission from *IEEE Software*, 1988)

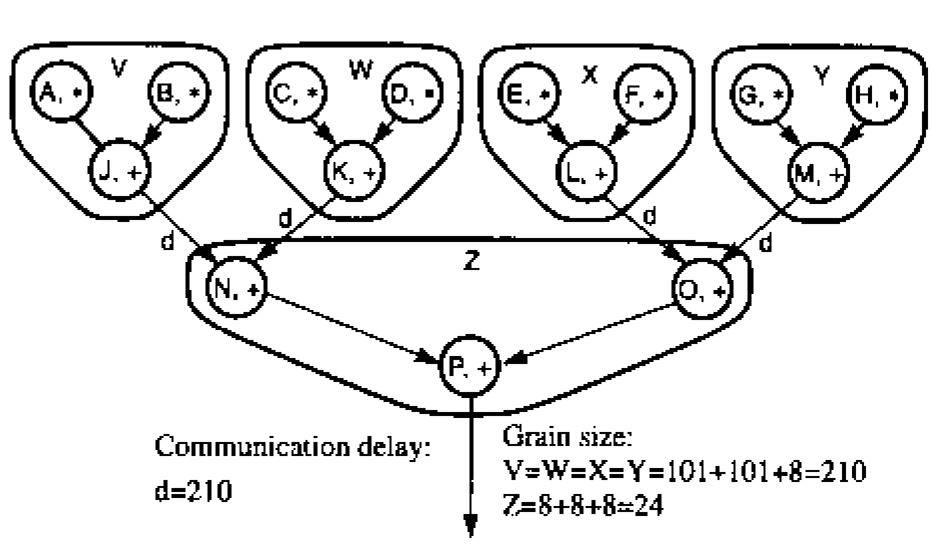


(a) A sequential schedule

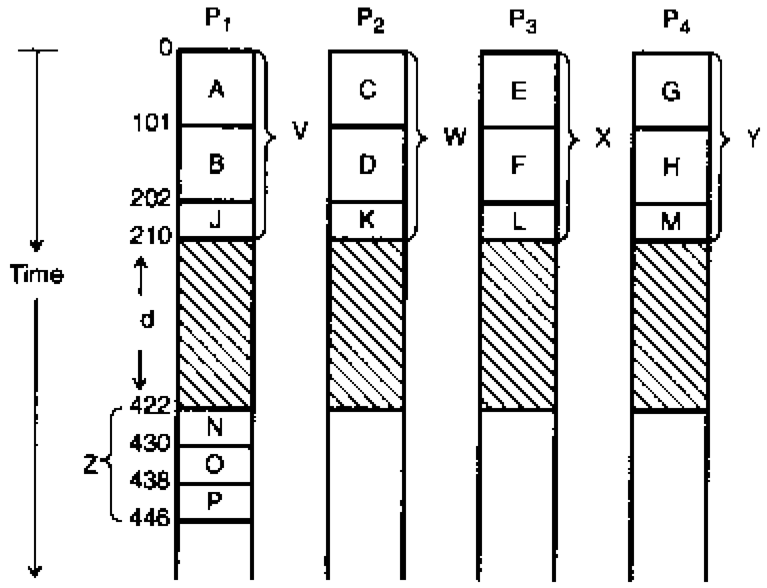


(b) A parallel schedule

Figure 2.10 Sequential versus parallel scheduling in Example 2.5.



(a) Grain packing of 15 small nodes into 5 bigger nodes



(b) Parallel schedule for the packed program

Figure 2.11 Parallel scheduling for Example 2.5 after grain packing to reduce communication delays.

Program Flow Mechanisms

- Conventional machines used control flow mechanism in which order of program execution explicitly stated in user programs.
- *Dataflow machines* which instructions can be executed by determining operand availability.
- *Reduction machines* trigger an instruction's execution based on the demand for its results.

Control Flow vs. Data Flow

- Control flow machines used shared memory for instructions and data. Since variables are updated by many instructions, there may be side effects on other instructions. These side effects frequently prevent parallel processing. Single processor systems are inherently sequential.
- Instructions in dataflow machines are unordered and can be executed as soon as their operands are available; data is held in the instructions themselves. *Data tokens* are passed from an instruction to its dependents to trigger execution.

Data Flow Features

- No need for
  - shared memory
  - program counter
  - control sequencer
- Special mechanisms are required to
  - detect data availability
  - match data tokens with instructions needing them
  - enable chain reaction of asynchronous instruction execution

A Dataflow Architecture - 1

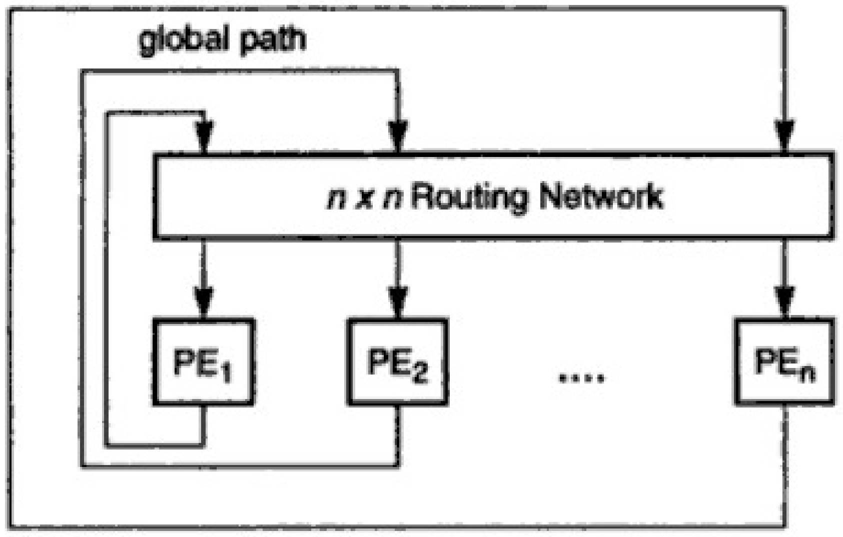
- The Arvind machine (MIT) has N PEs and an N-by-N interconnection network.
- Each PE has a token-matching mechanism that dispatches only instructions with data tokens available.
- Each datum is tagged with
  - address of instruction to which it belongs
  - context in which the instruction is being executed
- Tagged tokens enter PE through local path (pipelined), and can also be communicated to other PEs through the routing network.

A Dataflow Architecture - 2

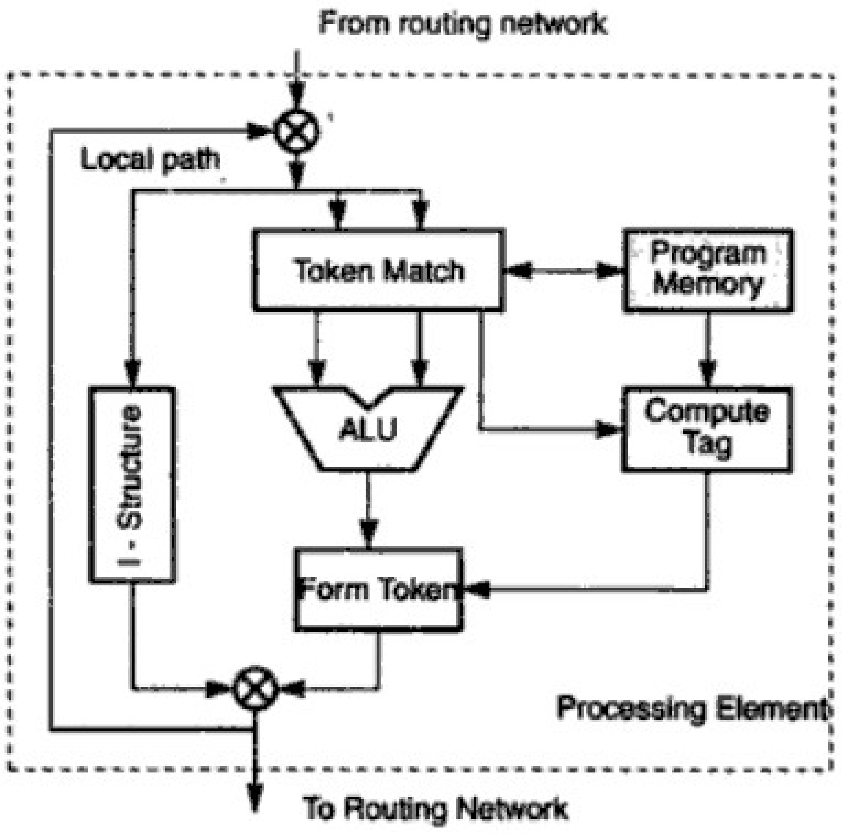
- Instruction address(es) effectively replace the program counter in a control flow machine.
- Context identifier effectively replaces the frame base register in a control flow machine.
- Since the dataflow machine matches the data tags from one instruction with successors, synchronized instruction execution is implicit.

A Dataflow Architecture - 3

- An *I-structure* in each PE is provided to eliminate excessive copying of data structures.
- Each word of the I-structure has a two-bit tag indicating whether the value is empty, full, or has pending read requests.
- This is a retreat from the pure dataflow approach.
- Example 2.6 shows a control flow and dataflow comparison.
- Special compiler technology needed for dataflow machines.



(a) The global architecture



(b) Interior design of a processing element

Demand-Driven Mechanisms

- Data-driven machines select instructions for execution based on the availability of their operands; this is essentially a bottom-up approach.
  - Eager evaluation
- Demand-driven machines take a top-down approach, attempting to execute the instruction (a *demander*) that yields the final result. This triggers the execution of instructions that yield its operands, and so forth.
  - Lazy evaluation
- The demand-driven approach matches naturally with functional programming languages (e.g. LISP and SCHEME).

Reduction Machine Models

- String-reduction model:
  - each demander gets a separate copy of the expression string to evaluate
  - each reduction step has an operator and embedded reference to demand the corresponding operands
  - each operator is suspended while arguments are evaluated
- Graph-reduction model:
  - expression graph reduced by evaluation of branches or subgraphs, possibly in parallel, with demanders given pointers to results of reductions.
  - based on sharing of pointers to arguments; traversal and reversal of pointers continues until constant arguments are encountered.

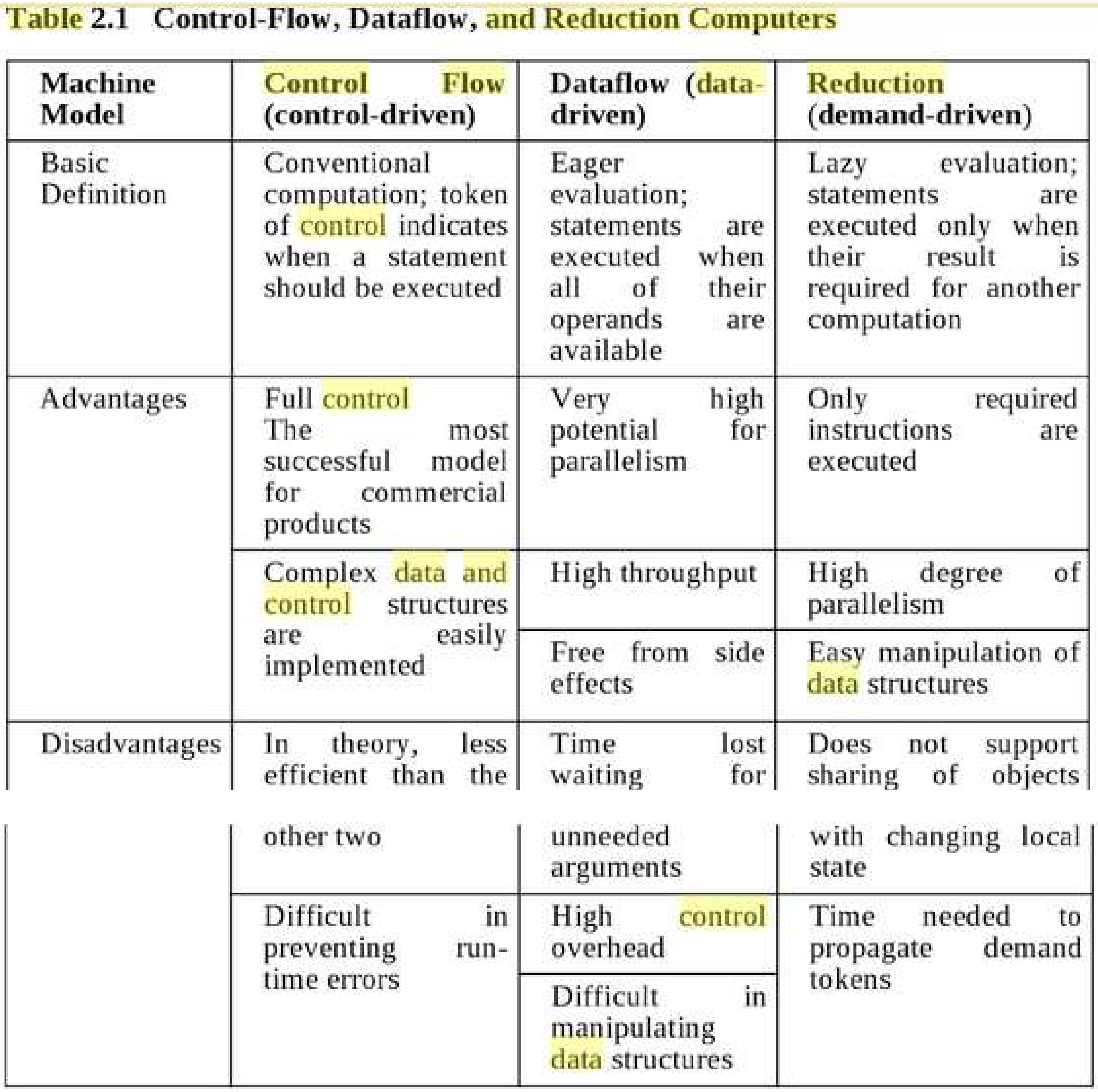


Table 2.1 Control-Flow, Dataflow, and Reduction Computers

Machine Model	Control Flow (control-driven)	Dataflow (data-driven)	Reduction (demand-driven)
Basic Definition	Conventional computation; token of control indicates when a statement should be executed	Eager evaluation; statements are executed when all of their operands are available	Lazy evaluation; statements are executed only when their result is required for another computation
Advantages	Full control The most successful model for commercial products	Very high potential for parallelism	Only required instructions are executed
	Complex data and control structures are easily implemented	High throughput	High degree of parallelism
		Free from side effects	Easy manipulation of data structures
Disadvantages	In theory, less efficient than the other two	Time lost waiting for unneeded arguments	Does not support sharing of objects with changing local state
	Difficult in preventing run-time errors	High control overhead	Time needed to propagate demand tokens
		Difficult in manipulating data structures	

System Interconnect Architectures

- Direct networks for static connections
- Indirect networks for dynamic connections
- Networks are used for
  - internal connections in a centralized system among
    - processors
    - memory modules
    - I/O disk arrays
  - distributed networking of multicomputer nodes

Goals and Analysis

- The goals of an interconnection network are to provide
  - low-latency
  - high data transfer rate
  - wide communication bandwidth
- Analysis includes
  - latency
  - bisection bandwidth
  - data-routing functions
  - scalability of parallel architecture

Network Properties and Routing

- Static networks: point-to-point direct connections that will not change during program execution
- Dynamic networks:
  - switched channels dynamically configured to match user program communication demands
  - include buses, crossbar switches, and multistage networks
- Both network types also used for inter-PE data routing in SIMD computers

Network Parameters

- Network size: The number of nodes in the graph used to represent the network
- Node Degree d: The number of edges incident to a node. Sum of in degree and out degree
- Network Diameter D: The maximum shortest path between any two nodes

Network Parameters (cont.)

- Bisection Width:
  - Channel bisection width b: The minimum number of edges along the cut that divides the network in two equal halves
  - Each channel has w bit wires
  - Wire bisection width: B=b*w; B is the wiring density of the network.
  - It provides a good indicator of the max communication bandwidth along the bisection of the network

Terminology - 1

- Network usually represented by a graph with a finite number of nodes linked by directed or undirected edges.
- Number of nodes in graph = *network size* .
- Number of edges (links or channels) incident on a node = *node degree* *d* (also note in and out degrees when edges are directed). Node degree reflects number of I/O ports associated with a node, and should ideally be small and constant.
- *Diameter* *D* of a network is the maximum shortest path between any two nodes, measured by the number of links traversed; this should be as small as possible (from a communication point of view).

Terminology - 2

- *Channel bisection width* b = minimum number of edges cut to split a network into two parts each having the same number of nodes. Since each channel has w bit wires, the *wire bisection width* $B = bw$. Bisection width provides good indication of maximum communication bandwidth along the bisection of a network, and all other cross sections should be bounded by the bisection width.
- *Wire* (or *channel*) *length* = length (e.g. weight) of edges between nodes.
- Network is symmetric if the topology is the same looking from any node; these are easier to implement or to program.
- Other useful characterizing properties: homogeneous nodes? buffered channels? nodes are switches?

Data Routing Functions

- Shifting
- Rotating
- Permutation (one to one)
- Broadcast (one to all)
- Multicast (many to many)
- Personalized broadcast (one to many)
- Shuffle
- Exchange
- Etc.

Permutations

- For n objects there are n! permutations by which the n objects can be reordered. The set of all permutations form a permutation group with respect to a composition operation. Cycle notation can be used to specify a permutation operation.
- Permutation $\pi = (a, b, c)(d, e)$ means: a->b, b->c, c->a, d->e and e->d in a circular fashion. The cycle (a, b, c) has a period of 3, and the cycle (d, e) has a period of 2. π will have a period equal to $2 \times 3 = 6$.

Permutations (cont.)

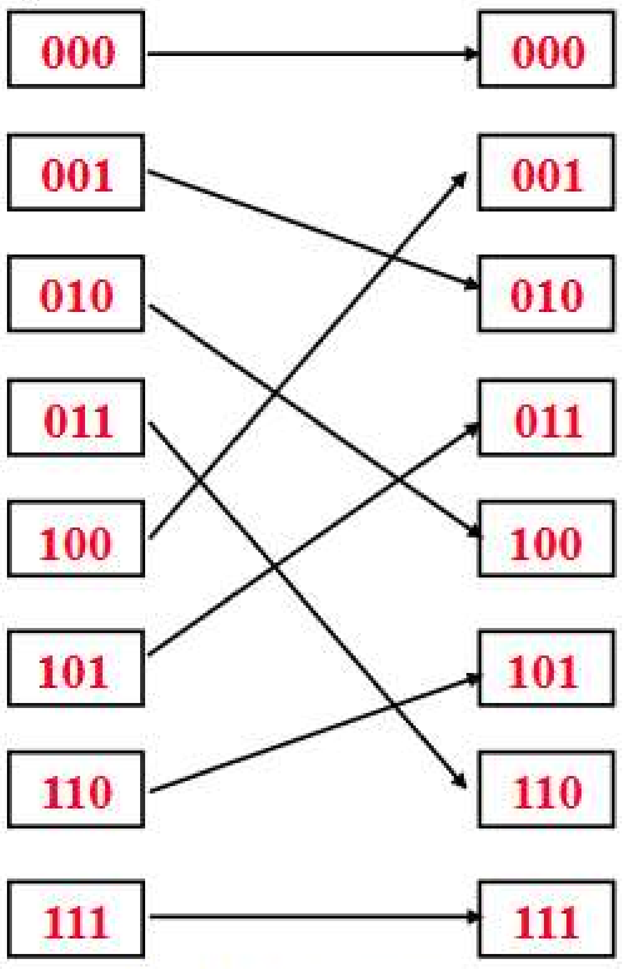
- Can be implemented using crossbar switches, multistage networks or with shifting or broadcast operations.
- Permutation capability is an indication of network’s data routing capabilities

Perfect Shuffle and Exchange

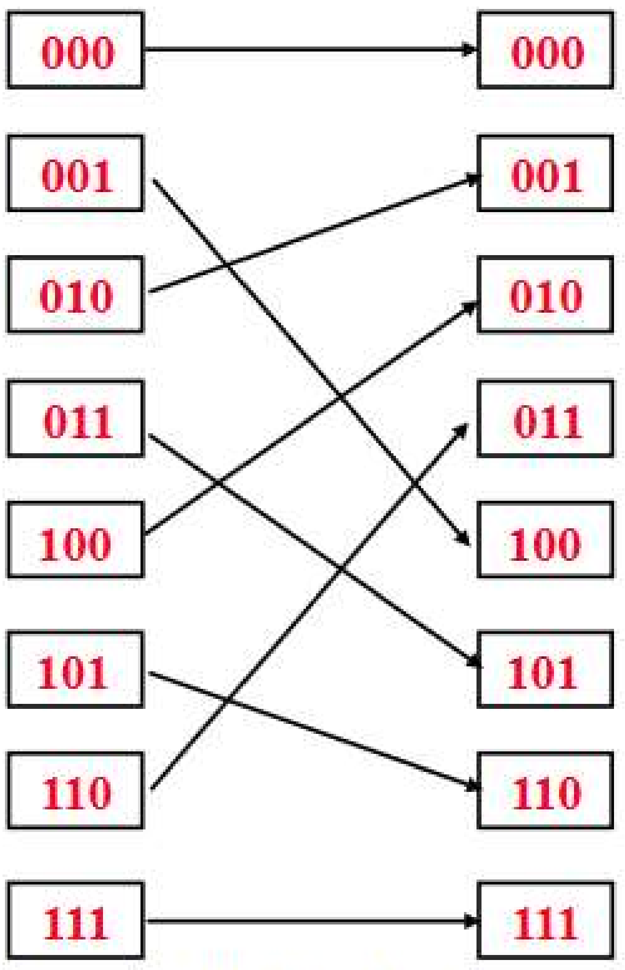
- Harold Stone suggested the special permutation that entries according to the mapping of the k-bit binary number *a b* … *k* to *b c* … *k a* (that is, shifting 1 bit to the left and wrapping it around to the least significant bit position).
- The inverse perfect shuffle reverses the effect of the perfect shuffle.

Perfect Shuffle

- Special permutation function
- $n = 2^k$ objects; each object representation requires k bits
- Perfect shuffle maps x to y where:
    - $x = (x_{k-1}, \ldots, x_1, x_0)$
    - $y = (x_{k-2}, \ldots, x_1, x_0, x_{k-1})$



Perfect Shuffle



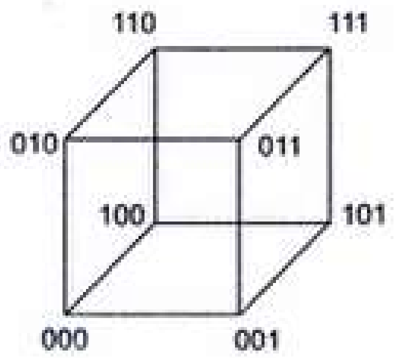
Inverse Perfect Shuffle

Hypercube Routing Functions

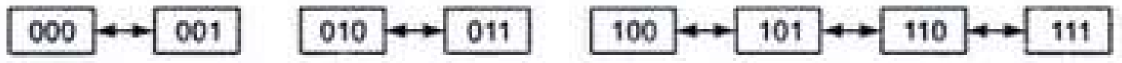
- If the vertices of a *n*-dimensional cube are labeled with *n*-bit numbers so that only one bit differs between each pair of adjacent vertices, then *n* **routing functions** are defined by the bits in the node (vertex) address.
- For example, with a 3-dimensional cube, we can easily identify routing functions that exchange data between nodes with addresses that differ in the least significant, most significant, or middle bit.

Exchange

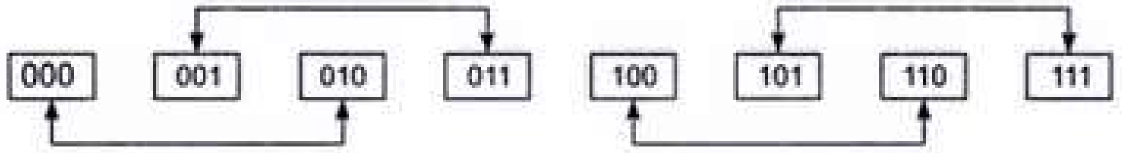
- $n = 2^k$ objects; each object representation requires k bits
- The exchange maps x to y where:
  - $x = (x_{k-1}, \ldots, x_1, x_0)$
  - $y = (x_{k-1}, \ldots, x_1, x_0')$
- Hypercube routing functions are exchanges



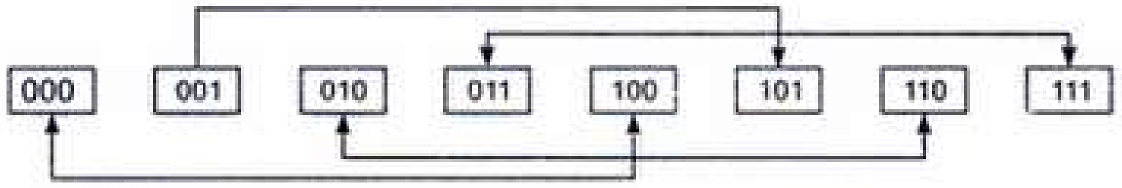
(a) A 3-cube with nodes denoted as $C_2C_1C_0$ in binary



(b) Routing by least significant bit, C_0



(c) Routing by middle bit, C_1

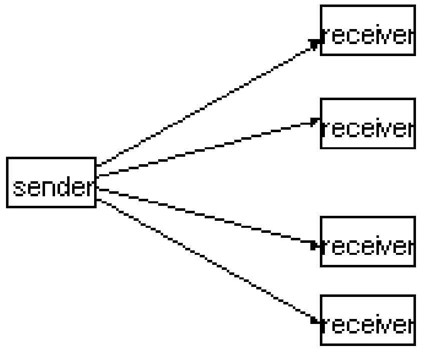


(d) Routing by most significant bit, C_2

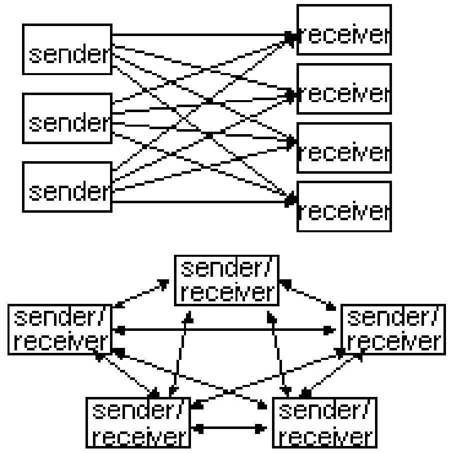
Fig. 2.15 Three routing functions defined by a binary 3-cube

Broadcast and Multicast

- Broadcast: One-to-all mapping
- Multicast: one subset to another subset(many to many)
- Personalized Broadcast: Personalized messages to only selected receivers
  - One-to-Many



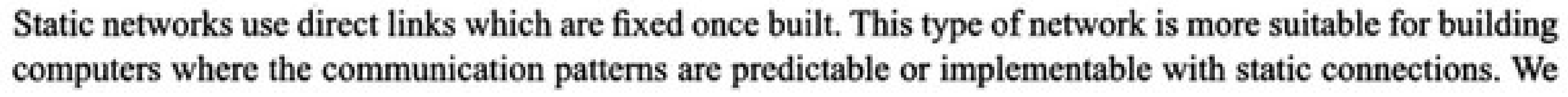
Many-to-Many



Factors Affecting Performance

- Functionality – how the network supports data routing, interrupt handling, synchronization, request/message combining, and coherence
- Network latency – worst-case time for a unit message to be transferred
- Bandwidth – maximum data rate
- Hardware complexity – implementation costs for wire, logic, switches, connectors, etc.
- Scalability – how easily does the scheme adapt to an increasing number of processors, memories, etc.?

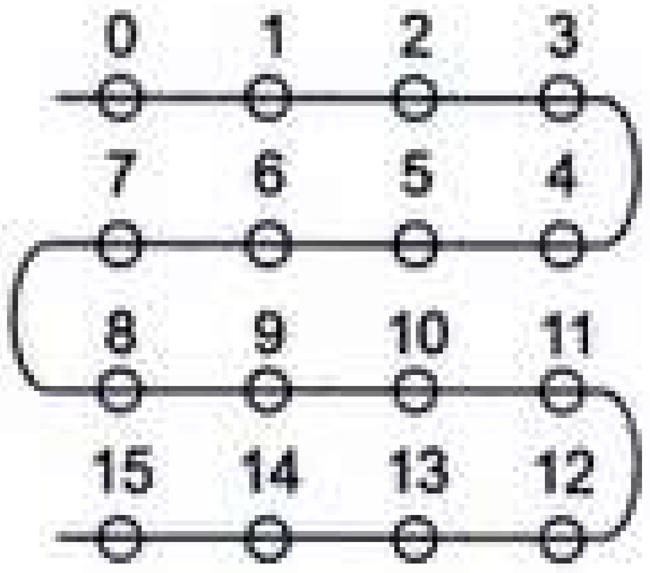
Static Networks


Static networks use direct links which are fixed once built. This type of network is more suitable for building computers where the communication patterns are predictable or implementable with static connections. We

- Linear Array
- Ring and Chordal Ring
- Barrel Shifter
- Tree and Star
- Fat Tree
- Mesh and Torus

Static Networks – Linear Array

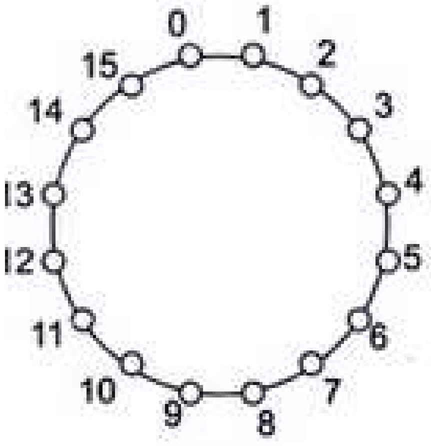
- N nodes connected by *n*-1 links (not a bus);
- segments between different pairs of nodes can be used in parallel.
- Internal nodes have degree 2; end nodes have degree 1.
- Diameter = n-1
- Bisection = 1
- For small n, this is economical, but for large n, it is obviously inappropriate.



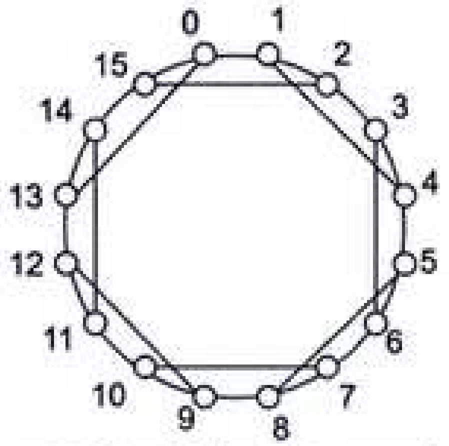
(a) Linear array

Static Networks – Ring, Chordal Ring

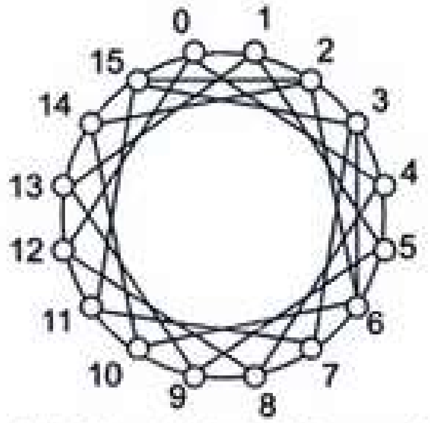
- Like a linear array, but the two end nodes are connected by an n th link; the ring can be uni- or bi-directional. Diameter is $\lfloor n/2 \rfloor$ for a bidirectional ring, or n for a unidirectional ring.
- By adding additional links (e.g. "chords" in a circle), the node degree is increased, and we obtain a chordal ring. This reduces the network diameter.
- In the limit, we obtain a fully-connected network, with a node degree of n -1 and a diameter of 1.



(b) Ring



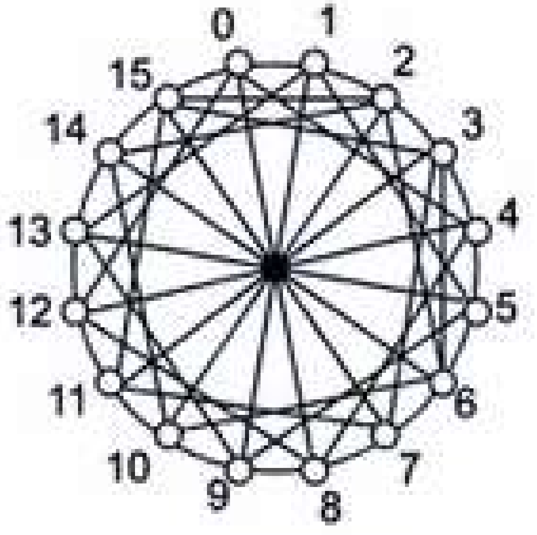
(c) Chordal ring of degree 3



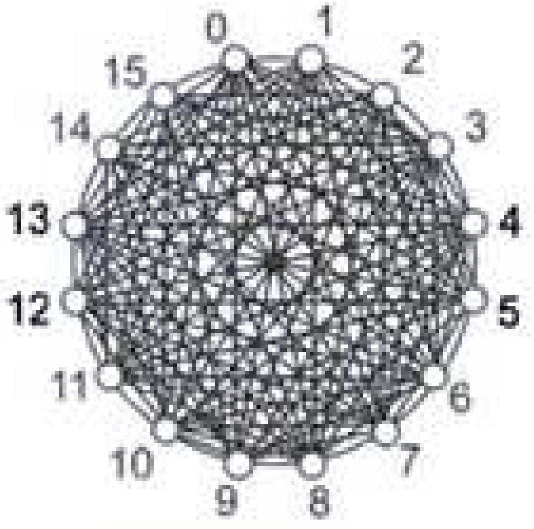
(d) Chordal ring of degree 4 (same as Illiac mesh)

Static Networks – Barrel Shifter

- Like a ring, but with additional links between all pairs of nodes that have a distance equal to a power of 2.
- With a network of size $N = 2^n$, each node has degree $d = 2n - 1$, and the network has diameter $D = n/2$.
  - For ex: N=16,d=7,D=2
- Barrel shifter connectivity is greater than any chordal ring of lower node degree.
- Barrel shifter much less complex than fully-interconnected network.



(e) Barrel shifter



(f) Completely connected

Static Networks – Tree and Star

- A k-level completely balanced binary tree will have $N = 2k - 1$ nodes, with maximum node degree of 3 and network diameter is $2(k - 1)$.
- The balanced binary tree is scalable, since it has a constant maximum node degree.
- A star is a two-level tree with a node degree $d = N - 1$ and a constant diameter of 2.

Static Networks – Fat Tree

- A fat tree is a tree in which the number of edges between nodes increases closer to the root (similar to the way the thickness of limbs increases in a real tree as we get closer to the root).
- The edges represent communication channels (“wires”), and since communication traffic increases as the root is approached, it seems logical to increase the number of channels there.

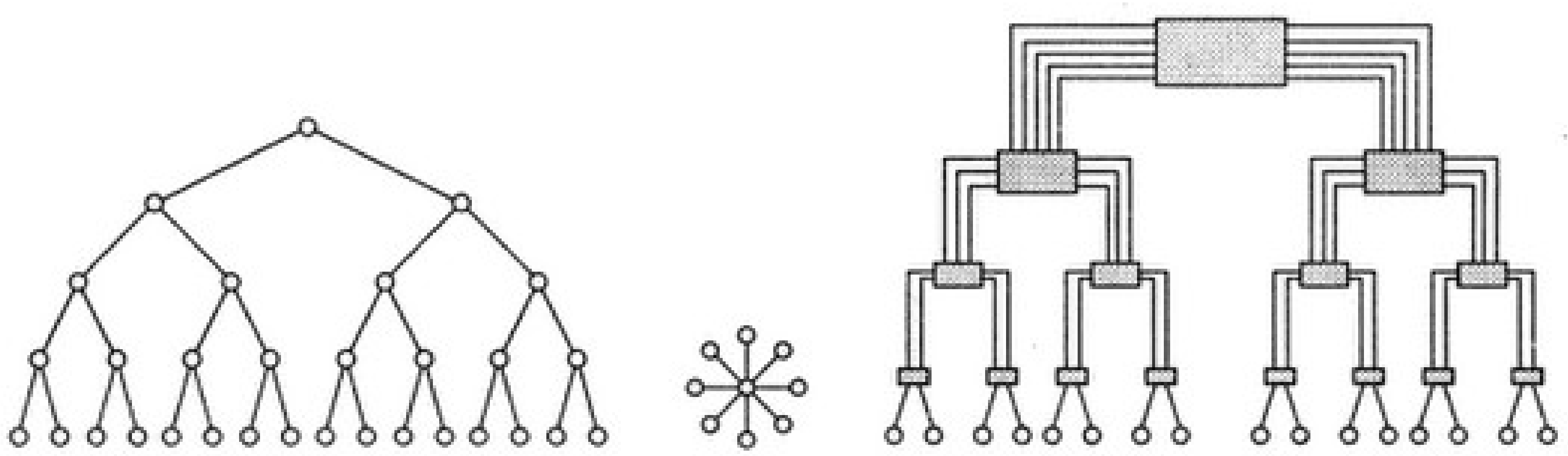
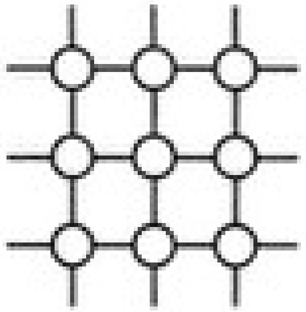
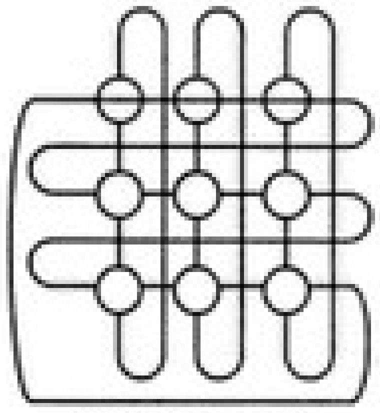


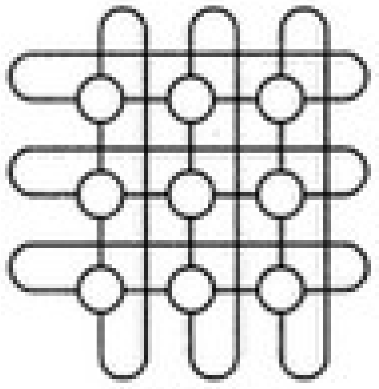
Fig. 2.17 Tree, star, and fat tree

Static Networks – Mesh and Torus

- *Pure mesh* – $N = n^k$ nodes with links between each adjacent pair of nodes in a row or column (or higher degree). This is not a symmetric network; interior node degree $d = 2k$, diameter = $k(n - 1)$.
- *Illiac mesh* (used in Illiac IV computer) – wraparound is allowed, thus reducing the network diameter to about half that of the equivalent pure mesh.
- A *torus* has ring connections in each dimension, and is symmetric. An n × n binary torus has node degree of 4 and a diameter of $2 \times \lfloor n / 2 \rfloor$.

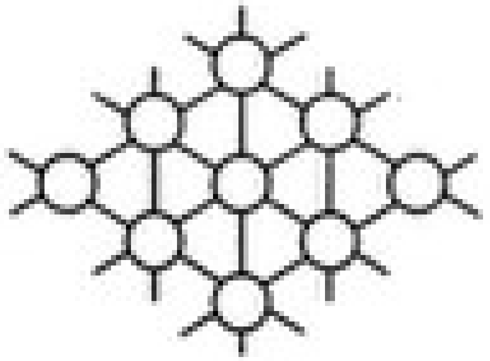

(a) Mesh


(b) Illiac mesh


(c) Torus

Static Networks – Systolic Array

- A systolic array is an arrangement of processing elements and communication links designed specifically to match the computation and communication requirements of a specific algorithm (or class of algorithms).
- This specialized character may yield better performance than more generalized structures, but also makes them more expensive, and more difficult to program.



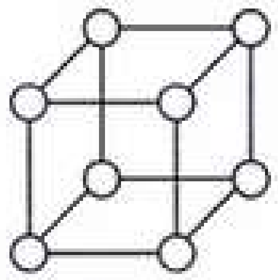
(d) Systolic array

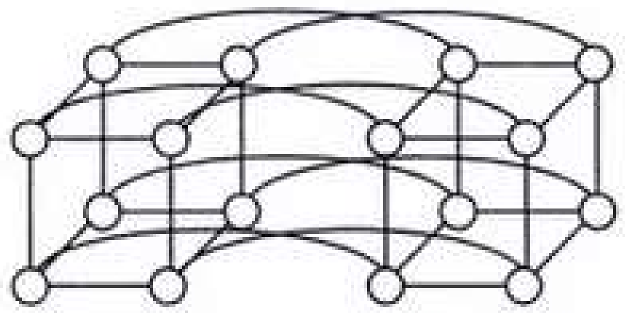
Static Networks – Hypercubes

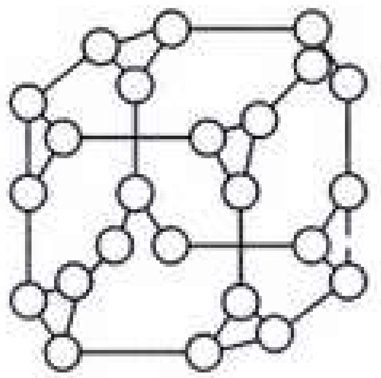
- A binary n-cube architecture with $N = 2^n$ nodes spanning along n dimensions, with two nodes per dimension.
- The hypercube scalability is poor, and packaging is difficult for higher-dimensional hypercubes.

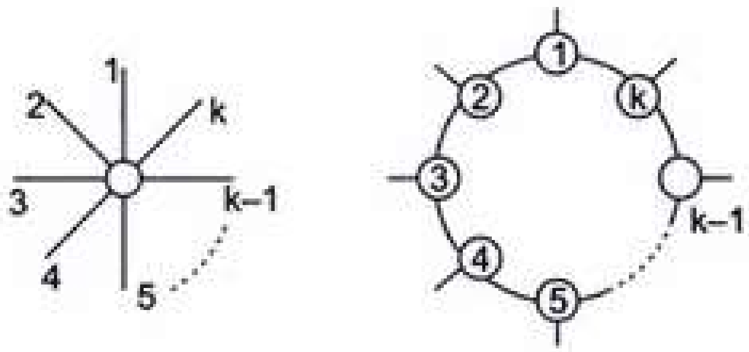
Static Networks – Cube-connected Cycles

- *k*-cube connected cycles (CCC) can be created from a *k*-cube by replacing each vertex of the k-dimensional hypercube by a ring of k nodes.
- A k-cube can be transformed to a *k*-CCC with $k \times 2^k$ nodes.
- The major advantage of a CCC is that each node has a constant degree (but longer latency) than in the corresponding *k*-cube. In that respect, it is more scalable than the hypercube architecture.


(a) 3-cube


(b) A 4-cube formed by interconnecting two 3-cubes


(c) 3-cube-connected cycles



(d) Replacing each node of a *k*-cube by a ring (cycle) of *k* nodes to form the *k*-cube-connected cycles

Static Networks – k-ary n-Cubes

- Rings, meshes, tori, binary *n*-cubes, and Omega networks (to be seen) are topologically isomorphic to a family of *k*-ary *n*-cube networks.
- *n* is the dimension of the cube, and *k* is the radix, or number of of nodes in each dimension.
- The number of nodes in the network, *N*, is k^{n}.
- Folding (alternating nodes between connections) can be used to avoid the long "end-around" delays in the traditional implementation.

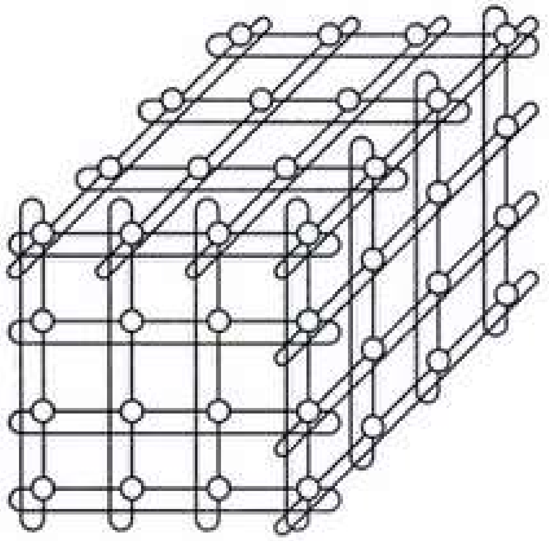


Fig. 2.20 The *k*-ary *n*-cube network shown with $k = 4$ and $n = 3$; hidden nodes or connections are not shown

Static Networks – k-ary n-Cubes

- The cost of k-ary n-cubes is dominated by the amount of wire, not the number of switches.
- With constant wire bisection, low-dimensional networks with wider channels provide lower latecny, less contention, and higher “hot-spot” throughput than higher-dimensional networks with narrower channels.

Network Throughput

- *Network throughput* – number of messages a network can handle in a unit time interval.
- One way to estimate is to calculate the maximum number of messages that can be present in a network at any instant (its *capacity*); throughput usually is some fraction of its capacity.
- A *hot spot* is a pair of nodes that accounts for a disproportionately large portion of the total network traffic (possibly causing congestion).
- *Hot spot throughput* is maximum rate at which messages can be sent between two specific nodes.

Minimizing Latency

- Latency is minimized when the network radix k and dimension n are chose so as to make the components of latency due to distance (# of hops) and the message aspect ratio L / W (message length L divided by the channel width W) approximately equal.
- This occurs at a very low dimension. For up to 1024 nodes, the best dimension (in this respect) is 2.

Dynamic Connection Networks

- Dynamic connection networks can implement all communication patterns based on program demands.
- In increasing order of cost and performance, these include
  - bus systems
  - multistage interconnection networks
  - crossbar switch networks
- Price can be attributed to the cost of wires, switches, arbiters, and connectors.
- Performance is indicated by network bandwidth, data transfer rate, network latency, and communication patterns supported.

Dynamic Networks – Bus Systems

- A *bus* system (*contention bus*, *time-sharing bus*) has
  - a collection of wires and connectors
  - multiple modules (processors, memories, peripherals, etc.) which connect to the wires
  - data transactions between pairs of modules
- Bus supports only one transaction at a time.
- Bus arbitration logic must deal with conflicting requests.
- Lowest cost and bandwidth of all dynamic schemes.
- Many bus standards are available.

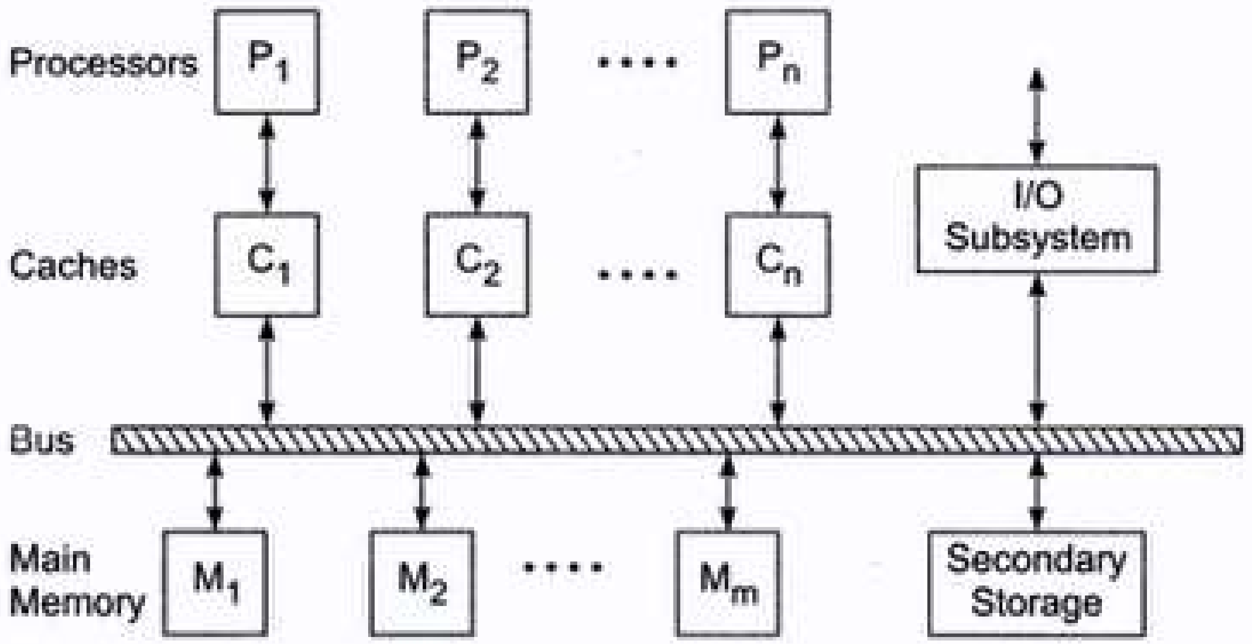


Fig. 2.22 A bus-connected multiprocessor system, such as the Sequent Symmetry S1

Dynamic Networks – Switch Modules

- An $a \times b$ switch module has a inputs and b outputs. A binary switch has $a = b = 2$.
- It is not necessary for $a = b$, but usually $a = b = 2^k$, for some integer k.
- In general, any input can be connected to one or more of the outputs. However, multiple inputs may not be connected to the same output.
- When only one-to-one mappings are allowed, the switch is called a *crossbar switch*.

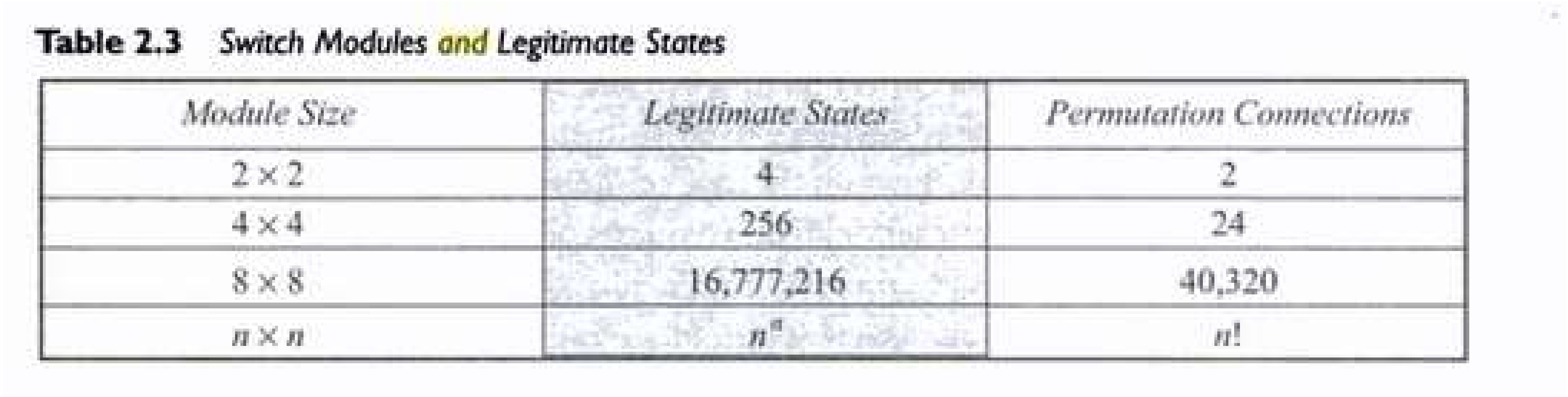


Table 2.3 *Switch Modules and Legitimate States*

Module Size	*Legitimate States*	*Permutation Connections*
2×2	4	2
4×4	256	24
8×8	16,777,216	40,320
$n \times n$	n^n	$n!$

Multistage Networks

- In general, any multistage network is comprised of a collection of $a \times b$ switch modules and fixed network modules. The $a \times b$ switch modules are used to provide variable permutation or other reordering of the inputs, which are then further reordered by the fixed network modules.
- A generic multistage network consists of a sequence alternating dynamic switches (with relatively small values for a and b) with static networks (with larger numbers of inputs and outputs). The static networks are used to implement interstage connections (ISC).

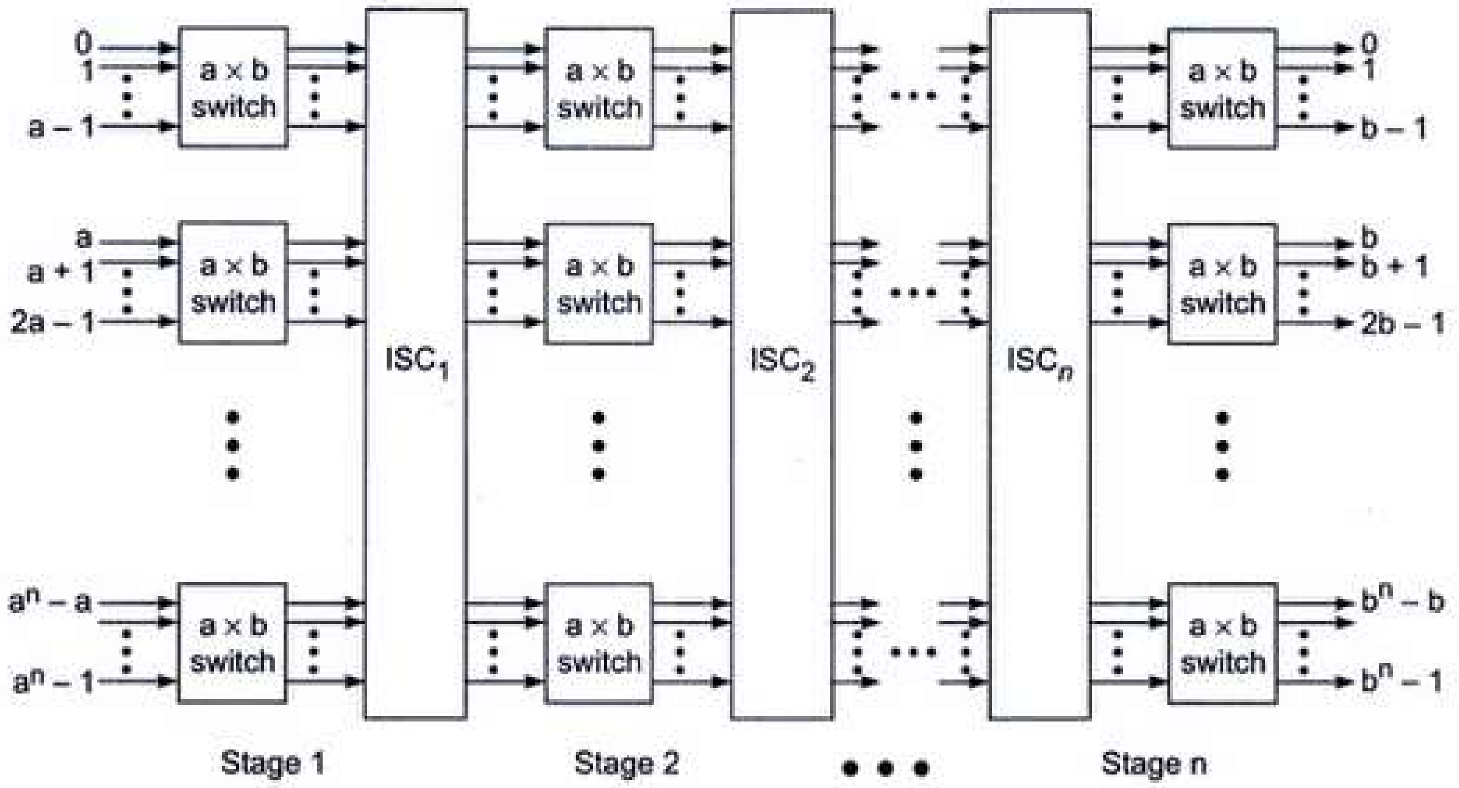
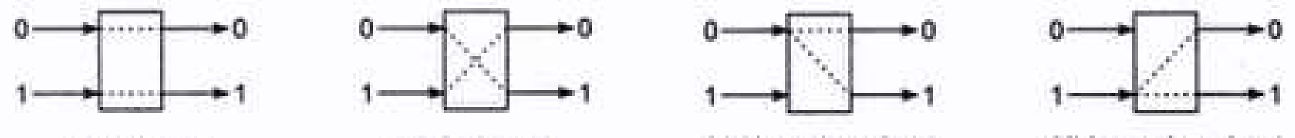


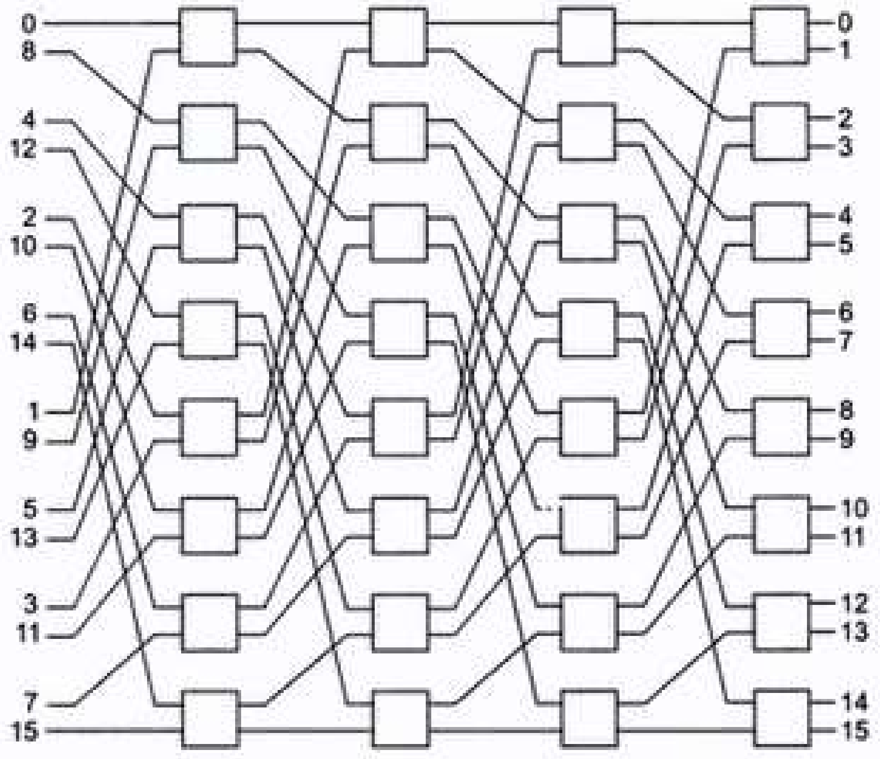
Fig. 2.23 A generalized structure of a multistage interconnection network (MIN) built with a × b switch modules and interstage connection patterns $ISC_1, ISC_2, \ldots, ISC_n$

Omega Network

- A 2×2 switch can be configured for
  - Straight-through
  - Crossover
  - Upper broadcast (upper input to both outputs)
  - Lower broadcast (lower input to both outputs)
  - (No output is a somewhat vacuous possibility as well)
- With four stages of eight 2×2 switches, and a static perfect shuffle for each of the four ISCs, a 16 by 16 Omega network can be constructed (but not all permutations are possible).
- In general , an n-input Omega network requires $\log_2 n$ stages of 2×2 switches and $n / 2$ switch modules.



(a) Straight (b) Crossover (c) Upper broadcast (d) Lower broadcast

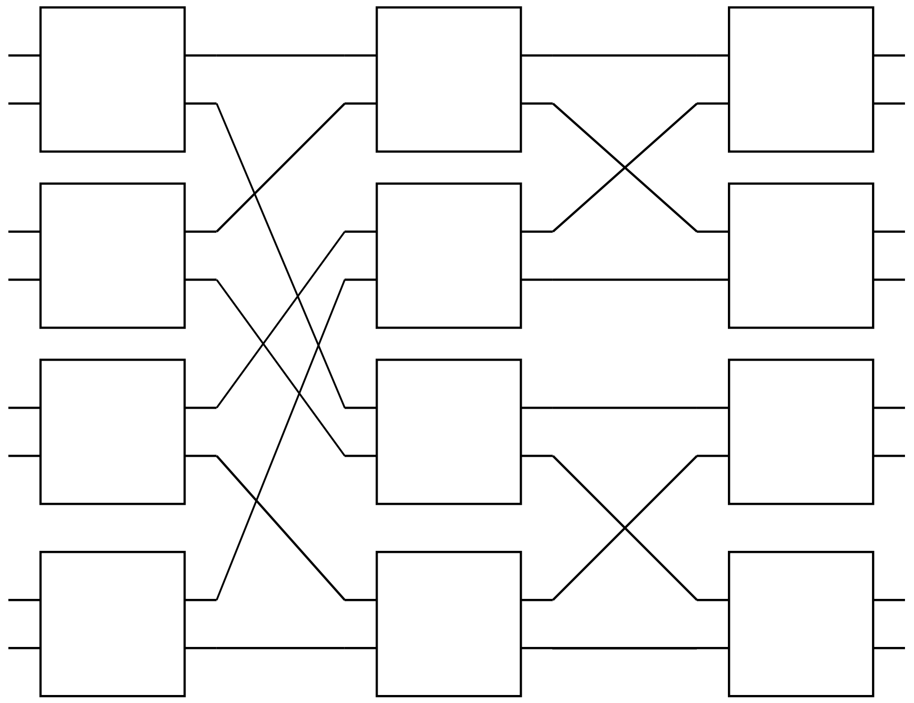


(e) 16 × 16 Omega network

Baseline Network

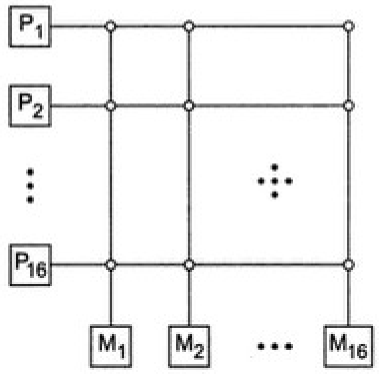
- A baseline network can be shown to be topologically equivalent to other networks (including Omega), and has a simple recursive generation procedure.
- Stage k (k = 0, 1, …) is an $m \times m$ switch block (where $m = N / 2^k$) composed entirely of 2×2 switch blocks, each having two configurations: straight through and crossover.

4×4 Baseline Network

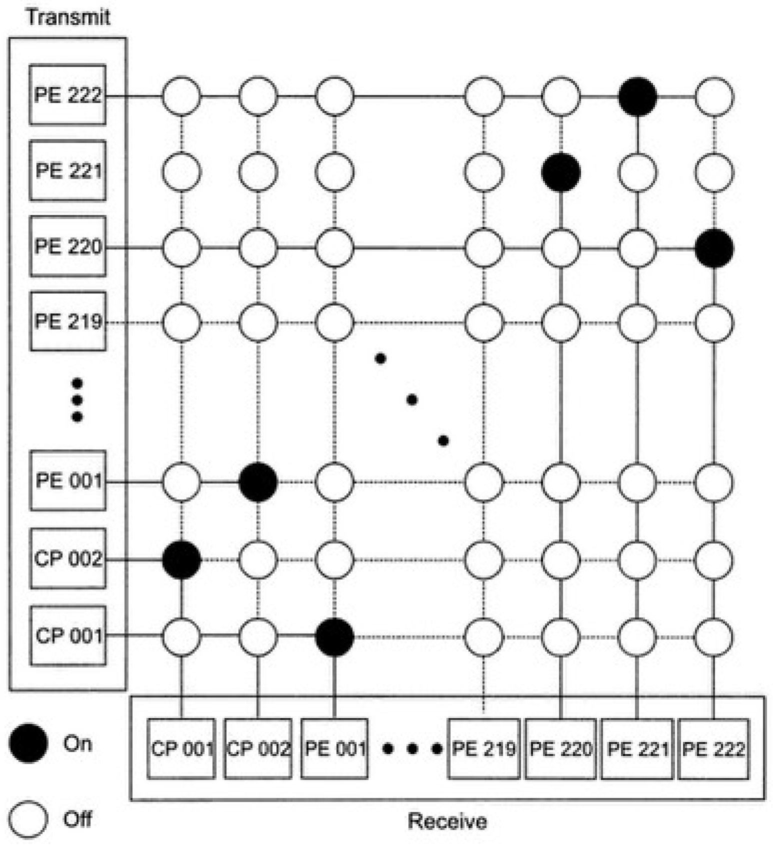


Crossbar Networks

- A m × n crossbar network can be used to provide a constant latency connection between devices; it can be thought of as a single stage switch.
- Different types of devices can be connected, yielding different constraints on which switches can be enabled.
  - With m processors and n memories, one processor may be able to generate requests for multiple memories in sequence; thus several switches might be set in the same row.
  - For m × m interprocessor communication, each PE is connected to both an input and an output of the crossbar; only one switch in each row and column can be turned on simultaneously. Additional control processors are used to manage the crossbar itself.

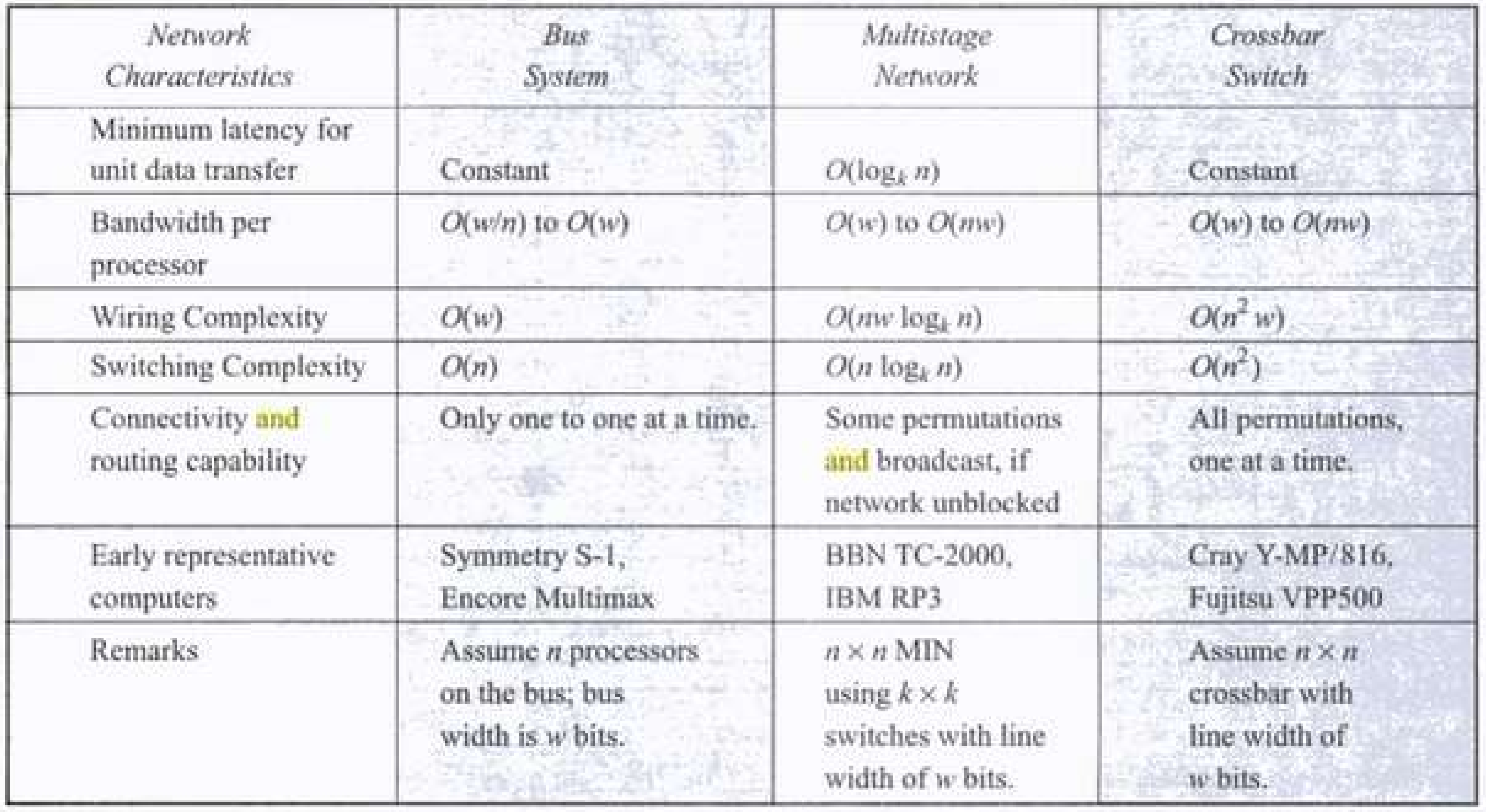


(a) Interprocessor-memory crossbar network built in the C.mmp multiprocessor at Carnegie-



(b) The interprocessor crossbar network built in the Fujitsu VPP500 vector parallel processor (1992)

Table 2.4 *Summary of Dynamic Network Characteristics*



Network Characteristics	*Bus System*	*Multistage Network*	*Crossbar Switch*
Minimum latency for unit data transfer	Constant	$O(\log_k n)$	Constant
Bandwidth per processor	$O(w/n)$ to $O(w)$	$O(w)$ to $O(nw)$	$O(w)$ to $O(nw)$
Wiring Complexity	$O(w)$	$O(nw \log_k n)$	$O(n^2 w)$
Switching Complexity	$O(n)$	$O(n \log_k n)$	$O(n^2)$
Connectivity and routing capability	Only one to one at a time.	Some permutations and broadcast, if network unblocked	All permutations, one at a time.
Early representative computers	Symmetry S-1, Encore Multimax	BBN TC-2000, IBM RP3	Cray Y-MP/816, Fujitsu VPP500
Remarks	Assume n processors on the bus; bus width is w bits.	$n \times n$ MIN using $k \times k$ switches with line width of w bits.	Assume $n \times n$ crossbar with line width of w bits.