Chapter 2: Program and Network Properties

2.1 Conditions of Parallelism

The advantage of multiprocessors lays when parallelism in the program is popularly exploited and implemented using multiple processors. Thus in order to implement the parallelism we should understand the various conditions of parallelism.

What are various bottlenecks in implementing parallelism? Thus for full implementation of parallelism there are three significant areas to be understood namely computation models for parallel computing, interprocessor communication in parallel architecture and system integration for incorporating parallel systems. Thus multiprocessor system poses a number of problems that are not encountered in sequential processing such as designing a parallel algorithm for the application, partitioning of the application into tasks, coordinating communication and synchronization, and scheduling of the tasks onto the machine.

Data and Resource Dependence

The ability to execute several program segments in parallel requires each segment to be independent of the other segments. We use a dependence graph to describe the relations. The nodes of a dependence graph correspond to the program statement (instructions), and directed edges with different labels are used to represent the ordered relations among the statements. The analysis of dependence graphs shows where opportunity exists for parallelization and vectorization.

Data dependence: The ordering relationship between statements is indicated by the data dependence. Five type of data dependence are defined below:

1. Flow dependence: A statement S2 is flow dependent on S1 if an execution path exists from s1 to S2 and if at least one output (variables assigned) of S1feeds in as input (operands to be used) to S2 also called RAW hazard and denoted as $S_1 \rightarrow S_2$

2. Antidependence: Statement S2 is antidependent on the statement S1 if S2 follows S1 in

the program order and if the output of S2 overlaps the input to S1 also called RAW hazard and denoted as $S_1 \mapsto S_2$

3. Output dependence : two statements are output dependent if they produce (write) the same output variable. Also called WAW hazard and denoted as $S_1 \multimap S_2$

4. I/O dependence: Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file referenced by both I/O statement.

5. Unknown dependence: The dependence relation between two statements cannot be determined in the following situations:

- The subscript of a variable is itself subscribed(indirect addressing)
- The subscript does not contain the loop index variable.
- A variable appears more than once with subscripts having different coefficients of the loop variable.
- The subscript is non linear in the loop index variable.

Parallel execution of program segments which do not have total data independence can produce non-deterministic results.

Consider the following fragment of any program:

S1: Load R1, A

S2:Add R2, R1

S3:Move R1, R3

S4:Store B, R1

Here the Forward dependency S1to S2, S3 to S4, S2 to S2

Anti-dependency from S2to S3

Output dependency S1 toS3

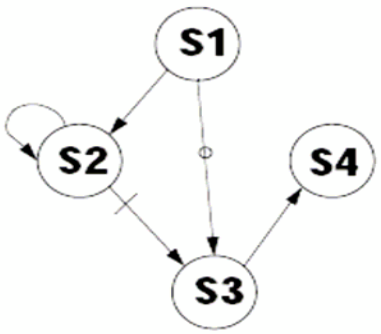


Figure 2.1 Dependence graph

Control Dependence

This refers to the situation where the order of the execution of statements cannot be determined before run time. For example all condition statement, where the flow of statement depends on the output. Different paths taken after a conditional branch may depend on the data hence we need to eliminate this data dependence among the instructions. This dependence also exists between operations performed in successive iterations of looping procedure. Control dependence often prohibits parallelism from being exploited.

Control-independent example:

```
for (i=0;i<n;i++)
{
a[i] = c[i];
if (a[i] < 0) a[i] = 1;
}
```

Control-dependent example:

```
for (i=1;i<n;i++)
{
if (a[i-1] < 0) a[i] = 1;
}
```

Control dependence also avoids parallelism to being exploited. Compilers are used to eliminate this control dependence and exploit the parallelism.

Resource dependence

Data and control dependencies are based on the independence of the work to be done.

Resource independence is concerned with conflicts in using shared resources, such as registers, integer and floating point ALUs, etc. ALU conflicts are called ALU dependence. Memory (storage) conflicts are called storage dependence.

Bernstein's Conditions

Bernstein's conditions are a set of conditions which must exist if two processes can execute in parallel.

Notation

Ii is the set of all input variables for a process *Pi* . *Ii* is also called the read set or domain of *Pi*. *Oi* is the set of all output variables for a process *Pi* .Oi is also called write set

If P*1* and P*2* can execute in parallel (which is written as P*1* || P*2*), then:

$$\mathbf{I}_1 \cap \mathbf{O}_2 = \varnothing$$
$$\mathbf{I}_2 \cap \mathbf{O}_1 = \varnothing$$
$$\mathbf{O}_1 \cap \mathbf{O}_2 = \varnothing$$

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.

Hardware and software parallelism

Hardware parallelism

Hardware parallelism is defined by machine architecture and hardware multiplicity i.e., functional parallelism times the processor parallelism .It can be characterized by the number of instructions that can be issued per machine cycle. If a processor issues *k* instructions per machine cycle, it is called a *k-issue* processor. Conventional processors are *one-issue* machines. This provide the user the information about **peak attainable performance**. Examples. Intel i960CA is a three-issue processor (arithmetic, memory access, branch). IBM RS -6000 is a four-issue processor (arithmetic, floating-point, memory access, branch).A machine with *n k*-issue processors should be able to handle a maximum of *nk* threads simultaneously.

Software Parallelism

Software parallelism is defined by the control and data dependence of programs, and is revealed in the program's flow graph i.e., it is defined by dependencies with in the code and is a function of algorithm, programming style, and compiler optimization.

Mismatch between software and hardware parallelism

Consider the example program graph in Fig.2.3a. There are eight instructions {four loads

and four arithmetic operations) to be executed in three consecutive machine cycles. Four load operations are performed in the first cycle, followed by two multiply operations in the second cycle and two add/subtract operations in the third cycle. Therefore the parallelism varies from 4to 2 in three cycles. The average software parallelism is equal to 8/3 =2.67 instructions per cycle in this example program.

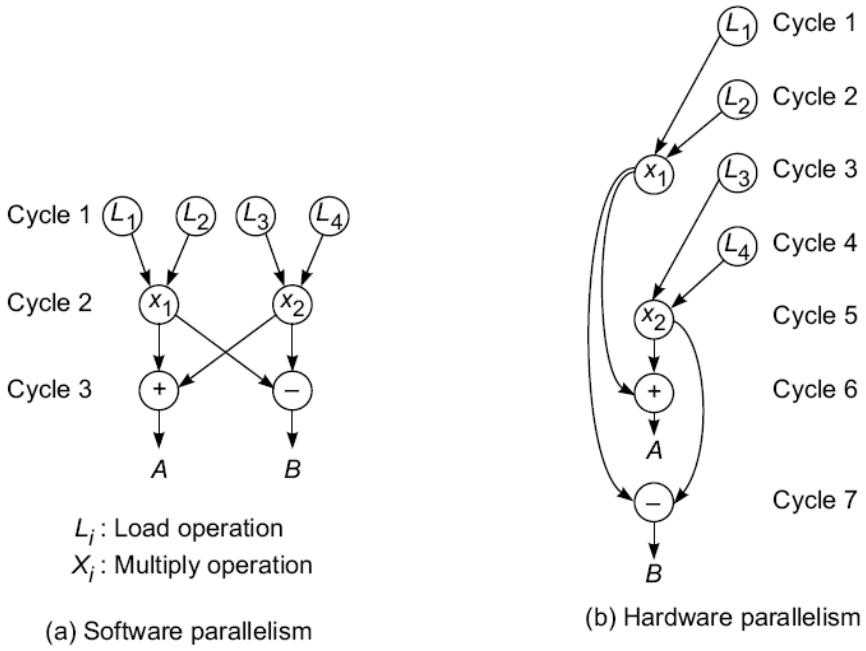


Fig. 2.3 Executing an example program by a two-issue superscalar processor

Now consider execution of the same program by a two-issue processor which can execute one memory access (load or write) and one arithmetic {add, subtract, multiply etc.) operation simultaneously. With this hardware restriction, the program must execute in seven machine cycles as shown in Fig. 2.3b. Therefore the hardware parallelism displays an average value of 8/7 = 1.14 instructions executed per cycle. This demonstrates a mismatch between the software parallelism and the hardware parallelism.

Match the software parallelism shown in fig 2.3a in a hardware platform of a dual processor system where single issue processors are used. The achievable hardware parallelism is shown in Fig 2.4.

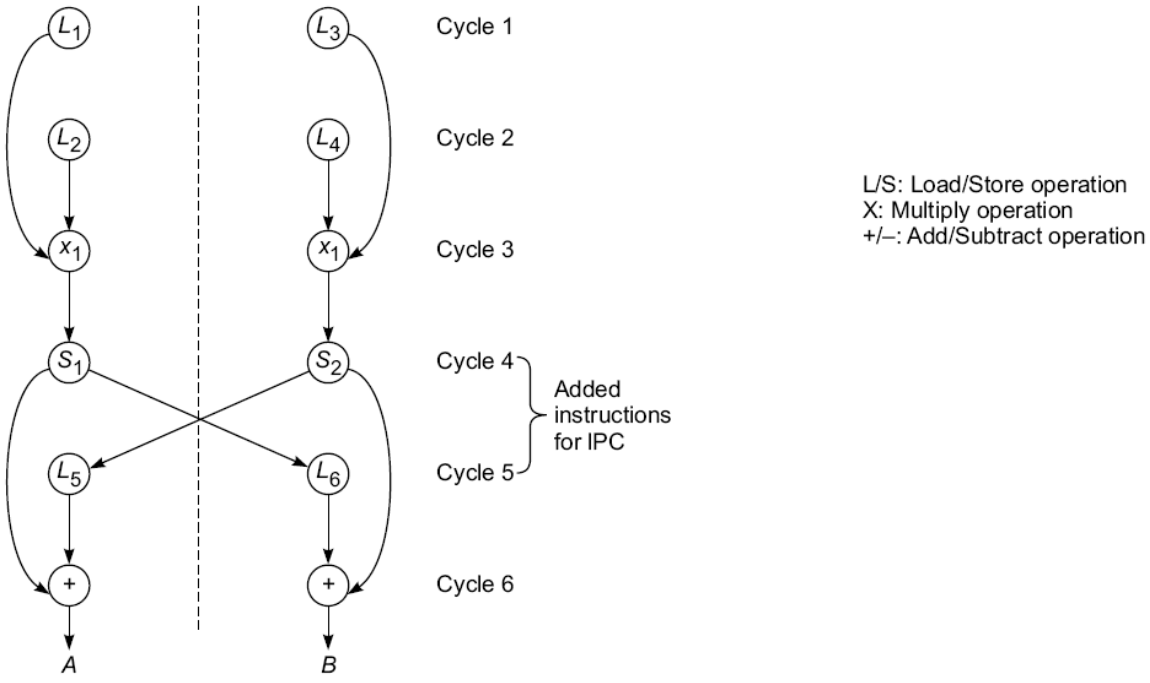


Fig. 2.4 Dual-processor execution of the program in Fig. 2.3a

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.

2.2 Program Partitioning & Scheduling

This section introduces the basic definitions of computational granularity or level of parallelism in programs. Communication latency and scheduling issues are illustrated with programming examples.

Grain size and latency

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 communication latency are closely related. Latency and grain size are interrelated and some general observation are

- As grain size decreases, potential parallelism increases, and overhead also increases.
- Overhead is the cost of parallelizing a task. The principle overhead is communication latency.
- As grain size is reduced, there are fewer operations between communication, and hence the impact of latency increases.
- Surface to volume: inter to intra-node comm.

Levels of parallelism

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.

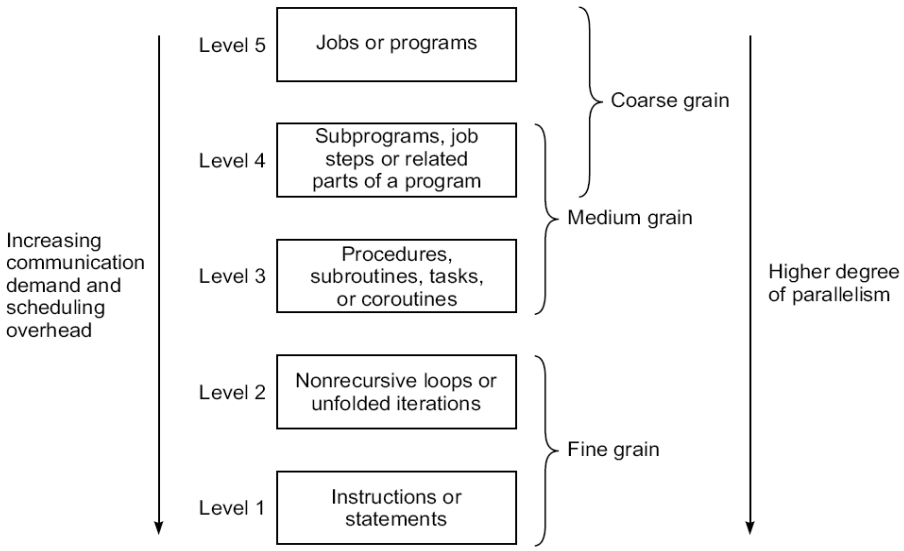


Fig. 2.5 Levels of parallelism in program execution on modern computers (Reprinted from Hwang, *Proc. IEEE*, October 1987)

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.

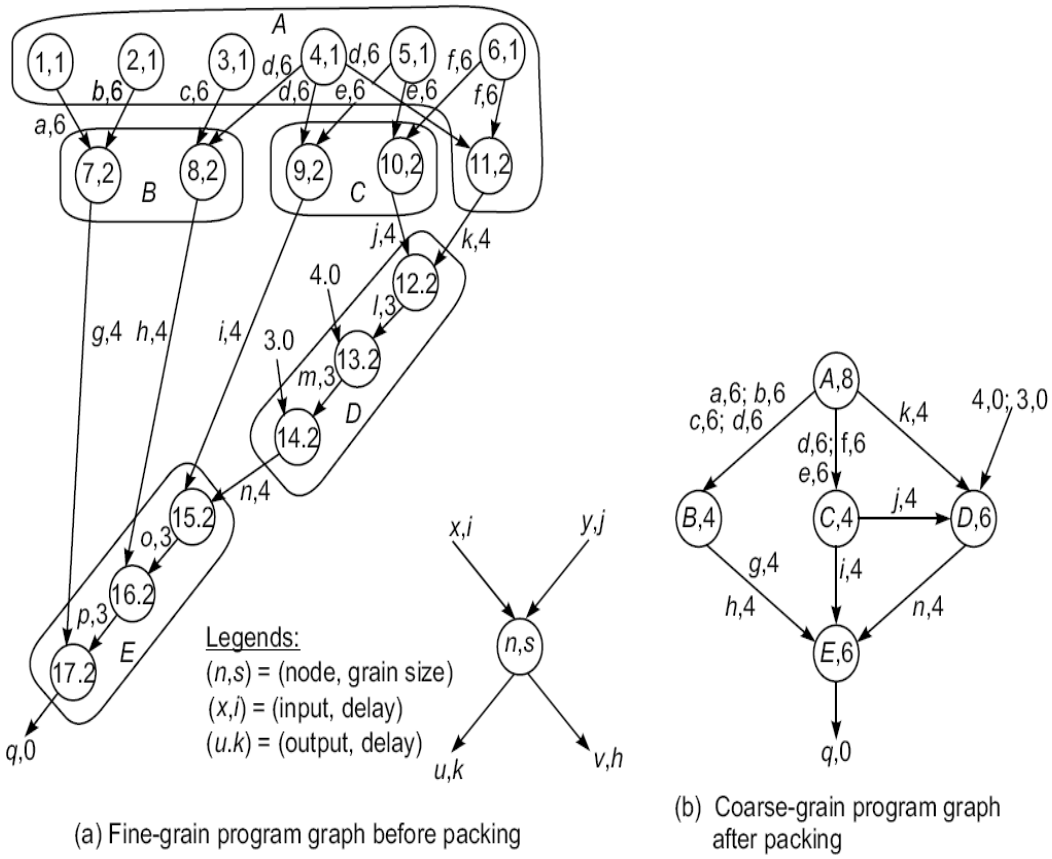
Packing two (or more) nodes produces a node with a larger grain size and possibly more edges to other nodes. Packing is done to eliminate unnecessary communication delays or reduce overall scheduling overhead.

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

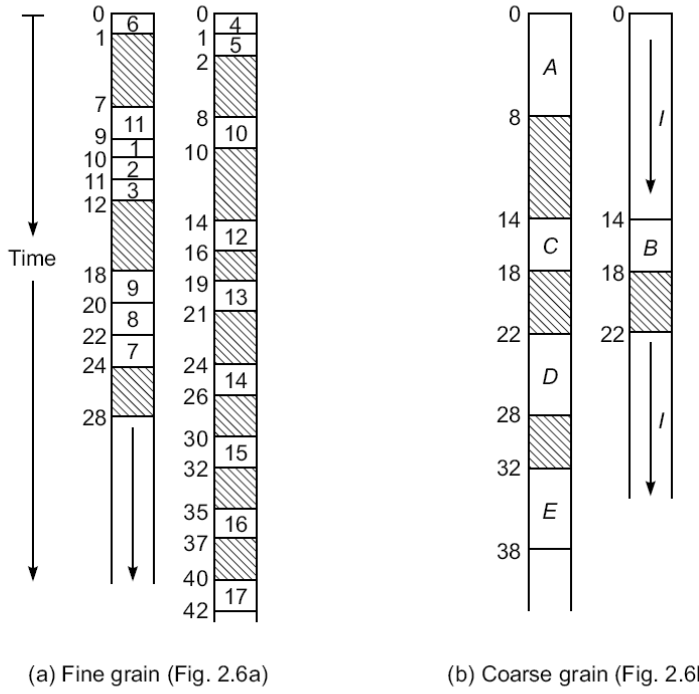
machine.



(a) Fine-grain program graph before packing

(b) Coarse-grain program graph after packing

Fig. 2.6 A program graph before and after grain packing in Example 2.4 (Modified from Kruatrachue and Lewis, *IEEE Software*, Jan. 1988)

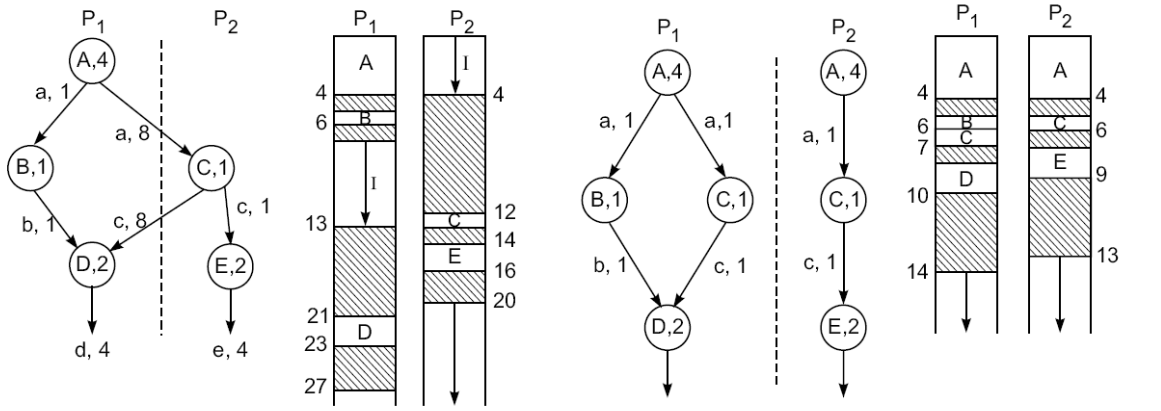


(a) Fine grain (Fig. 2.6a) (b) Coarse grain (Fig. 2.6b)

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

Node Duplication

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

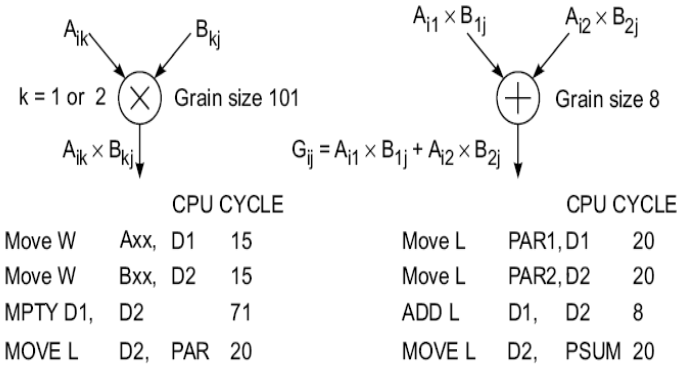


(a) Schedule without node duplication (b) Schedule with node duplication (A → A and A′; C → C and C′)

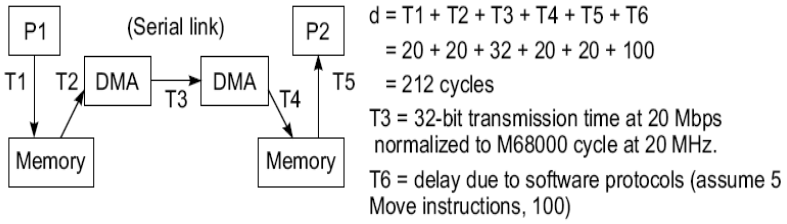
Fig. 2.8 Node-duplication scheduling to eliminate communication delays between processors (I: idle time; shaded areas: communication delays)

Program portioning and scheduling

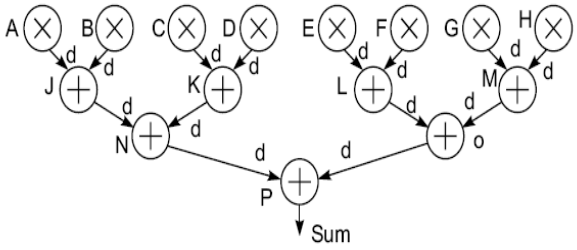
Scheduling and allocation is a highly important issue since an inappropriate scheduling of tasks can fail to exploit the true potential of the system and can offset the gain from parallelization. In this paper we focus on the scheduling aspect. The objective of scheduling is to minimize the completion time of a parallel application by properly allocating the tasks to the processors. In a broad sense, the scheduling problem exists in two forms: *static* and *dynamic*. In static scheduling, which is usually done at compile time, the characteristics of a parallel program (such as task processing times, communication, data dependencies, and synchronization requirements) are known before program execution



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



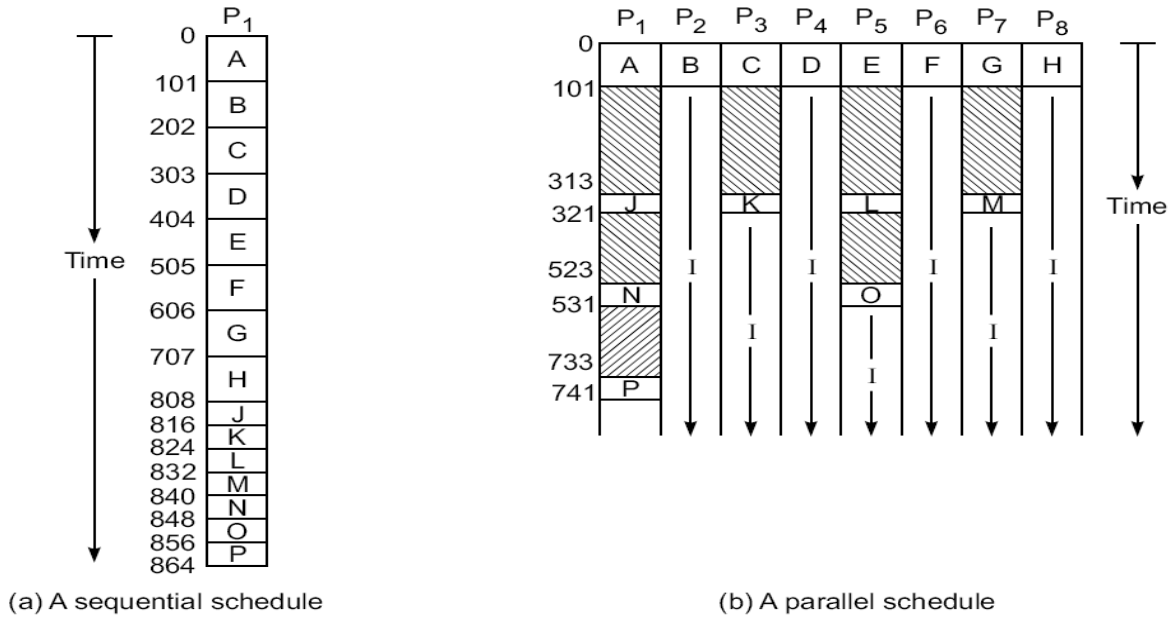
(b) Calculation of communication delay *d*



(c) Fine-grain program graph

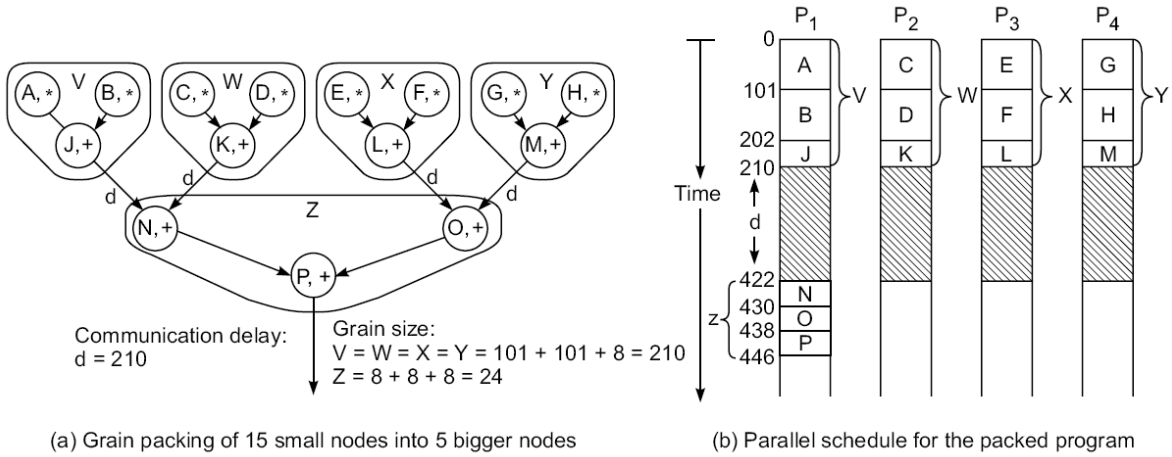
Fig. 2.9 Calculation of grain size and communication delay for the program graph in Example 2.5 (Courtesy of Kruatrachue and Lewis; reprinted with permission from *IEEE Software*, 1988)

A parallel program, therefore, can be represented by a node- and edge-weighted directed acyclic graph (DAG), in which the node weights represent task processing times and the edge weights represent data dependencies as well as the communication times between tasks. In dynamic scheduling only, a few assumptions about the parallel program can be made before execution, and thus, scheduling decisions have to be made on-the-fly. The goal of a dynamic scheduling algorithm as such includes not only the minimization of the program completion time but also the minimization of the scheduling overhead which constitutes a significant portion of the cost paid for running the scheduler. In general dynamic scheduling is an NP hard problem.



(a) A sequential schedule (b) A parallel schedule

Fig. 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

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

2.3 Program flow mechanism

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

In Control flow computers the next instruction is executed when the last instruction as stored in the program has been executed where as in Data flow computers an instruction executed when the data (operands) required for executing that instruction is available

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

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

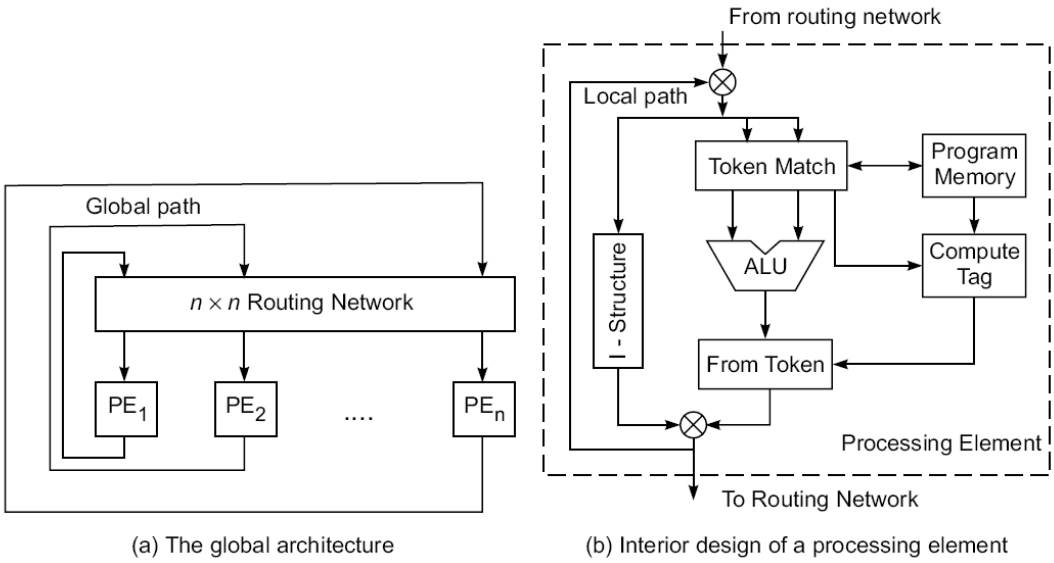


Fig. 2.12 The MIT tagged-token dataflow computer (adapted from Arvind and Iannucci, 1986 with permission)

Tagged tokens enter PE through local path (pipelined), and can also be communicated to other PEs through the routing network. 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.

Demand-Driven Mechanisms

Data-driven machines select instructions for execution based on the availability of their operands; this is essentially a bottom-up approach.

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. The demand-driven approach matches naturally with functional programming languages (e.g. LISP and SCHEME).

Pattern driven computers : An instruction is executed when we obtain a particular data patterns as output. There are two types of pattern driven computers

(1) 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

(2) 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.

2.4 System interconnect architecture

Various types of interconnection networks have been suggested for SIMD computers. These are basically classified have been classified on network topologies into two categories namely

Static Networks

Dynamic Networks

Static versus Dynamic Networks

The topological structure of an SIMD array processor is mainly characterized by the data routing network used in interconnecting the processing elements.

The topological structure of an SIMD array processor is mainly characterized by the data routing network used in the interconnecting the processing elements. To execute the communication the routing function f is executed and via the interconnection network the PEi copies the content of its Ri register into the Rf(i) register of PEf(i). The f(i) the processor identified by the mapping function f. The data routing operation occurs in all active PEs simultaneously.

Network properties and routing

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

These 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.

Network is symmetric if the topology is the same looking from any node; these are easier to implement or to program.

Diameter : The maximum distance between any two processors in the network or in other words we can say **Diameter**, is the maximum number of (routing) processors through which a message must pass on its way from source to reach destination. Thus diameter measures the maximum delay for transmitting a message from one processor to another as it determines communication time hence smaller the diameter better will be the network topology.

Connectivity: How many paths are possible between any two processors i.e., the multiplicity of paths between two processors. Higher connectivity is desirable as it minimizes contention.

Arch connectivity of the network: the minimum number of arcs that must be removed for the network to break it into two disconnected networks. The arch connectivity of various network are as follows

- 1 for linear arrays and binary trees
- 2 for rings and 2-d meshes
- 4 for 2-d torus
- d for d-dimensional hypercubes

Larger the arch connectivity lesser the conjunctions and better will be network topology.

Channel width : The channel width is the number of bits that can communicated simultaneously by a interconnection bus connecting two processors

Bisection Width and Bandwidth: In order divide the network into equal halves we require the remove some communication links. The minimum number of such communication links that have to be removed are called the Bisection Width. **Bisection width basically provide us the information about** the largest number of messages which can be sent simultaneously (without needing to use the same wire or routing processor at the same time and so delaying one another), no matter which processors are sending to which other processors. Thus larger the bisection width is the better the network topology is considered. Bisection Bandwidth is the minimum volume of communication allowed

between two halves of the network with equal numbers of processors This is important for the networks with weighted arcs where the weights correspond to the *link width i.e.*, (how much data it can transfer). The Larger bisection width the better network topology is considered.

Cost the cost of networking can be estimated on variety of criteria where we consider the the number of communication links or wires used to design the network as the basis of cost estimation. Smaller the better the cost

Data Routing Functions: A data routing network is used for inter –PE data exchange. It can be static as in case of hypercube routing network or dynamic such as multistage network. Various type of data routing functions are Shifting, Rotating, Permutation (one to one), Broadcast (one to all), Multicast (many to many), Personalized broadcast (one to many), Shuffle, Exchange Etc.

Permutations

Given n objects, there are n ! ways in which they can be reordered (one of which is no reordering). A permutation can be specified by giving the rule for reordering a group of objects. Permutations can be implemented using crossbar switches, multistage networks, shifting, and broadcast operations. The time required to perform permutations of the connections between nodes often dominates the network performance when n is large.

Perfect Shuffle and Exchange

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.

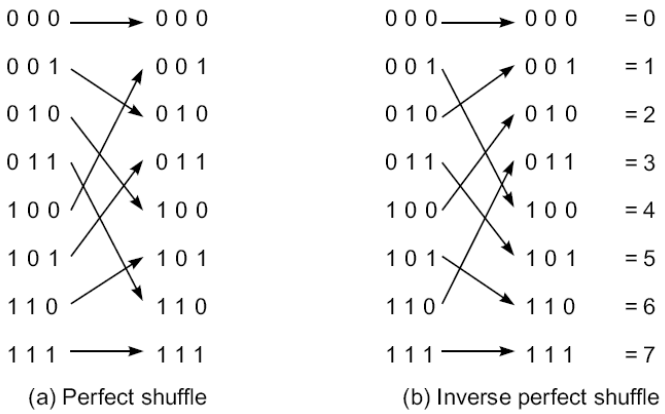
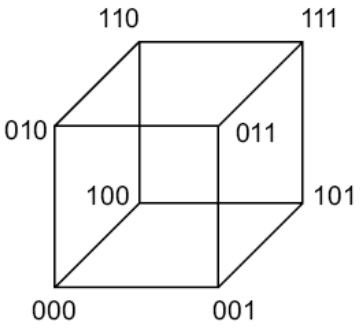


Fig. 2.14 Perfect shuffle and its inverse mapping over eight objects (Courtesy of H. Stone; reprinted with permission from *IEEE Trans. Computers*, 1971)

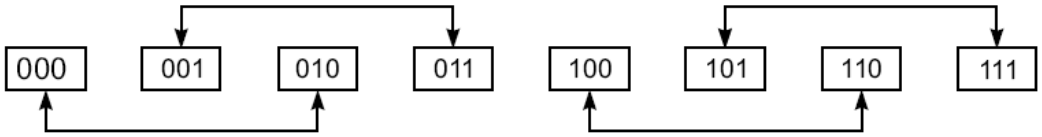
Hypercube Routing Functions

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.

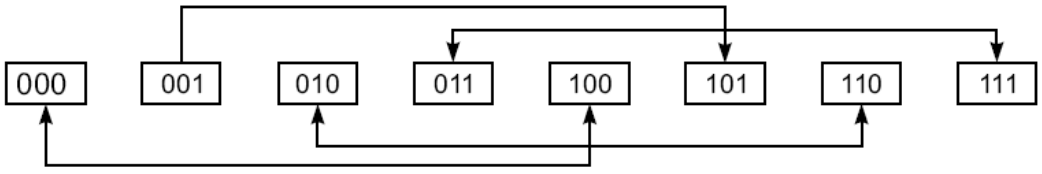


(a) A 3-cube with nodes denoted as $C_2C_1C_0$ in binary

(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

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 connection Networks

In static network the interconnection network is fixed and permanent interconnection path between two processing elements and data communication has to follow a fixed route to reach the destination processing element. Thus it Consist of a number of point-to-point links. Topologies in the static networks can be classified according to the dimension required for layout i.e., it can be 1-D, 2-D, 3-D or hypercube.

One dimensional topologies include Linear array as shown in figure 2.2 (a) used in some pipeline architecture.

Various 2-D topologies are

- The ring (figure 2.2(b))
- Star (figure 2.2(c))
- Tree (figure 2.2(d))
- Mesh (figure 2.2(e))
- Systolic Array (figure 2.2(f))

3-D topologies include

- Completely connected chordal ring (figure 2.2(g))
- Chordal ring (figure 2.2(h))
- 3 cube (figure 2.2(i))

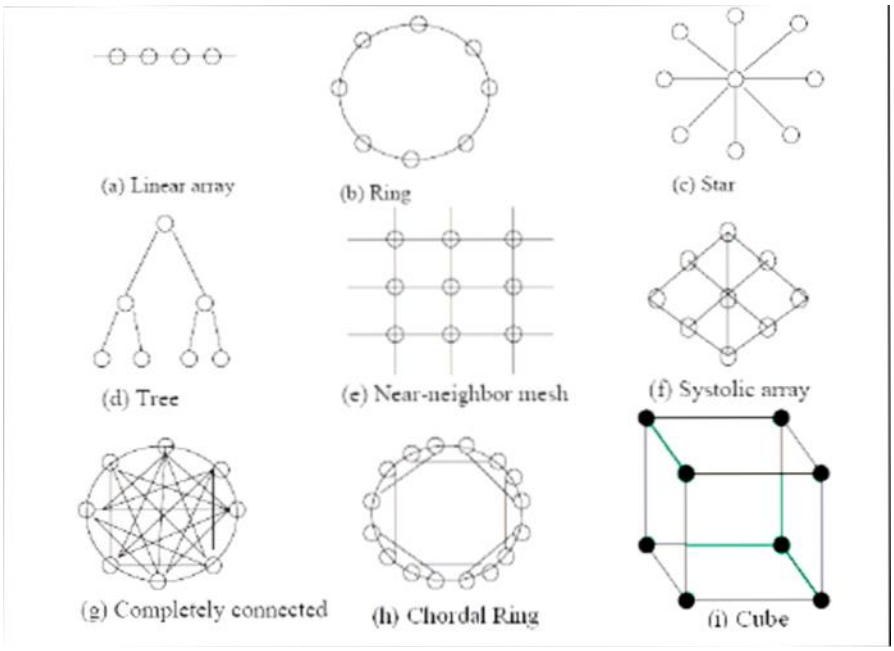


Figure 2.2 Static interconnection network topologies.

Torus architecture is also one of popular network topology it is extension of the mesh by having wraparound connections Figure below is a 2D Torus This architecture of torus is a symmetric topology unlike mesh which is not. The wraparound connections reduce the torus diameter and at the same time restore the symmetry. It can be

- o 1-D torus
- 2-D torus
- 3-D torus

The torus topology is used in Cray T3E

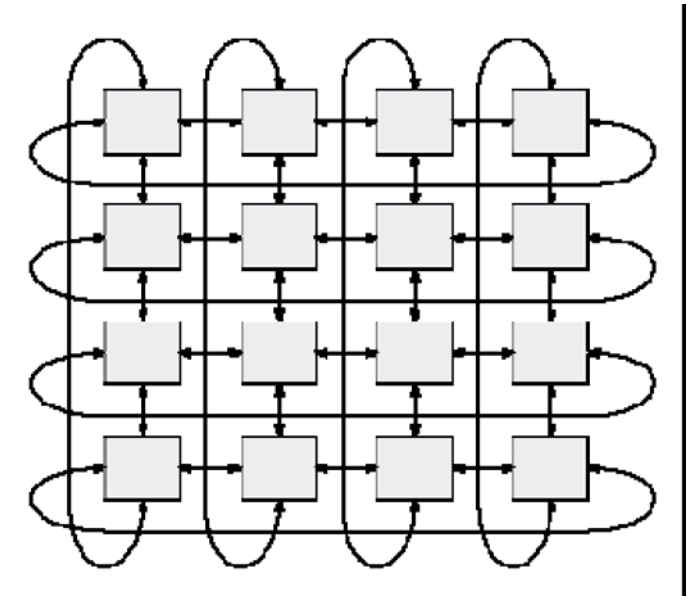


Figure 2.3 Torus technology

We can have further higher dimension circuits for example 3-cube connected cycle. A D-dimension W-wide hypercube contains W nodes in each dimension and there is a connection to a node in each dimension. The mesh and the cube architecture are actually 2-D and 3-D hypercube respectively. The below figure we have hypercube with dimension 4.

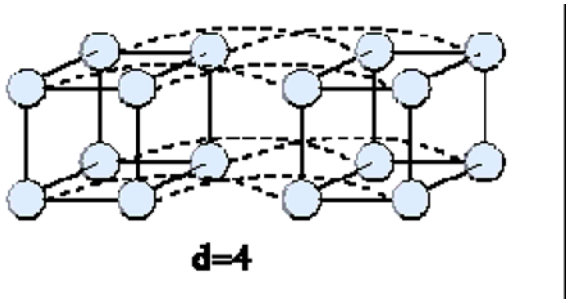


Figure 2.4 4-D hypercube.

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.

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.

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*.

Multistage Networks

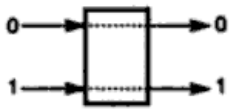
- In general, any multistage network is comprised of a collection of $a \times b$ switch modules and fixed network modules. The $a \times b$ switch modules are used to provide variable permutation or other reordering of the inputs, which are then further reordered by the fixed network modules.
- A generic 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).

Omega Network

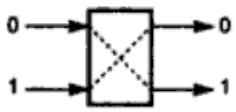
- A 2×2 switch can be configured for
  - Straight-through
  - Crossover
  - Upper broadcast (upper input to both outputs)

- Lower broadcast (lower input to both outputs)
- (No output is a somewhat vacuous possibility as well)

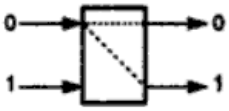
- With four stages of eight 2 × 2 switches, and a static perfect shuffle for each of the four ISCs, a 16 by 16 Omega network can be constructed (but not all permutations are possible).
- In general , an *n*-input Omega network requires $\log_2 n$ stages of 2 × 2 switches and $n / 2$ switch modules.



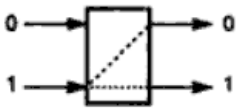
(a) Straight



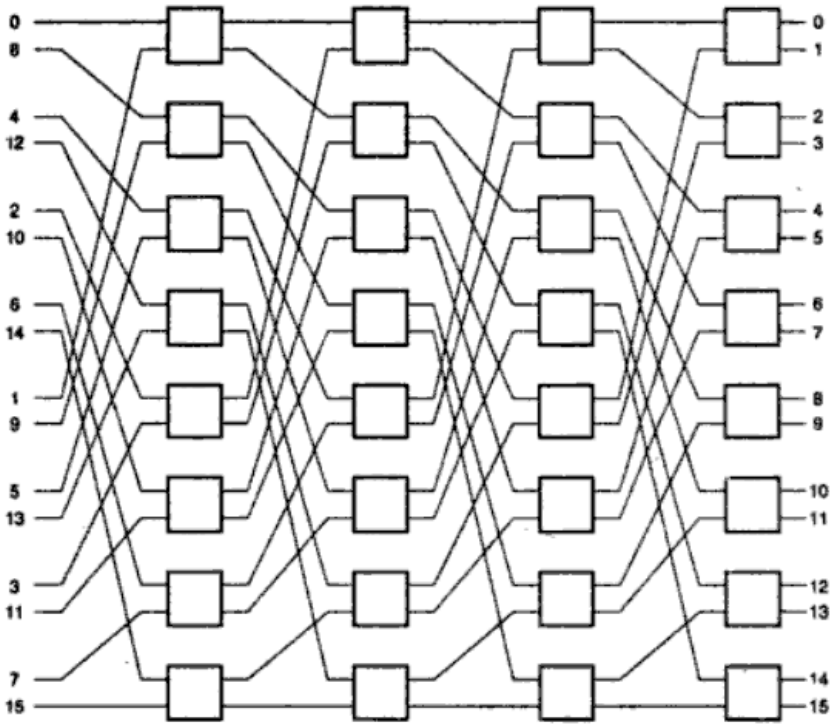
(b) Crossover



(c) Upper broadcast



(d) Lower broadcast

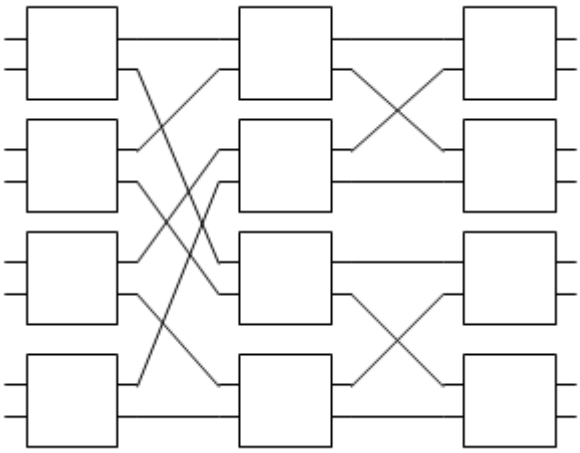


(e) A 16 × 16 Omega network

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 × m switch block (where m = N / 2^k) composed entirely of 2 × 2 switch blocks, each having two configurations: straight through and crossover.



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.