in six synchronized vector instructions over 64-component vector data and both driven by the same-speed clock. Calculate the total execution time on the SIMD machine, ignoring instruction broadcast and other delays.

(c) What is the speedup gain of the SIMD computer over the SISD computer?

Problem 1.9 Prove that the best parallel algorithm written for an n-processor EREW PRAM model can be no more than $O(\log n)$ times slower than any algorithm for a CRCW model of PRAM having the same number of processors.

Problem 1.10 Consider the multiplication of two n-bit binary integers using a 1.2-μm CMOS multiplier chip. Prove the lower bound $AT^2 > kn^2$, where A is the chip area, T is the execution time, n is the word length, and k is a technology-dependent constant.

Problem 1.11 Compare the PRAM models with physical models of real parallel computers in each of the following categories:

(a) Which PRAM variant can best model SIMD machines and how?

(b) Repeat the question in part (a) for shared-memory MIMD machines.

Problem 1.12 Answer the following questions related to the architectural development tracks presented in Section 1.5:

(a) For the shared-memory track (Fig. 1.17), explain the trend in physical memory organizations from the earlier system (C.mmp) to more recent systems (such as Dash, etc.).

(b) Distinguish between medium-grain and fine-grain multicomputers in their architectures and programming requirements.

(c) Distinguish between register-to-register and memory-to-memory architectures for building conventional multivector supercomputers.

(d) Distinguish between single-threaded and multithreaded processor architectures.

Problem 1.13 Design an algorithm to find the maximum of n numbers in $O(\log n)$ time on an EREW-PRAM model. Assume that initially each location holds one input value. Explain how you would make the algorithm processor time optimal.

Problem 1.14 Develop two algorithms for fast multiplication of two $n \times n$ matrices with a system of p processors, where $1 \leq p \leq n^3/\log n$. Choose an appropriate PRAM machine model to prove that the matrix multiplication can be done in $T = O(n^3/p)$ time.

(a) Prove that $T = O(n^2)$ if $p = n$. The corresponding algorithm must be shown, similar to that in Example 1.5.

(b) Show the parallel algorithm with $T = O(n)$ if $p = n^2$.

Problem 1.15 Match each of the following eight computer systems: KSR-1, RP3, Paragon, Dash, CM-2, VPP500, EM-5, and Tera, with one of the best descriptions listed below. The mapping is a one-to-one correspondence.

(a) A massively parallel system built with multiple-context processors and a 3-D torus architecture.

(b) A data-parallel computer built with bit-slice PEs interconnected by a hypercube/mesh network.

(c) A ring-connected multiprocessor using a cache-only memory architecture.

(d) An experimental multiprocessor built with a dynamic dataflow architecture.

(e) A crossbar-connected multiprocessor built with distributed processor/memory nodes forming a single address space.

(f) A multicomputer built with commercial microprocessors with multiple address spaces.

(g) A scalable multiprocessor built with distributed shared memory and coherent caches.

(h) An MIMD computer built with a large multistage switching network.

2

Program and Network Properties

This chapter covers fundamental properties of program behavior and introduces major classes of interconnection networks. We begin with a study of computational granularity, conditions for program partitioning, matching software with hardware, program flow mechanisms, and compilation support for parallelism. Interconnection architectures introduced include static and dynamic networks. Network complexity, communication bandwidth, and data-routing capabilities are discussed.

2.1 CONDITIONS OF PARALLELISM

The exploitation of parallelism has created a new dimension in computer science. In order to move parallel processing into the mainstream of computing, H.T. Kung (1991) has identified the need to make significant progress in three key areas: *computation models* for parallel computing, *interprocessor communication* in parallel architectures, and *system integration* for incorporating parallel systems into general computing environments.

A theoretical treatment of parallelism is thus needed to build a basis for the above challenges. In practice, parallelism appears in various forms in a computing environment. All forms can be attributed to levels of parallelism, computational granularity, time and space complexities, communication latencies, scheduling policies, and load balancing. Very often, tradeoffs exist among time, space, performance, and cost factors.

2.1.1 Data and Resource Dependences

The ability to execute several program segments in parallel requires each segment to be independent of the other segments. The independence comes in various forms as defined below separately. For simplicity, to illustrate the idea, we consider the dependence relations among instructions in a program. In general, each code segment may contain one or more statements.

We use a *dependence graph* to describe the relations. The nodes of a dependence graph correspond to the program statements (instructions), and the directed edges with different labels show 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 types of data dependence are defined below:

(1) *Flow dependence*: A statement S2 is *flow-dependent* on statement S1 if an execution path exists from S1 to S2 and if at least one output (variables assigned) of S1 feeds in as input (operands to be used) to S2. Flow dependence is denoted as S1 → S2.

(2) *Antidependence*: Statement S2 is *antidependent* on statement S1 if S2 follows S1 in program order and if the output of S2 overlaps the input to S1. A direct arrow crossed with a bar as in S1 $\not\rightarrow$ S2 dicates antidependence from S1 to S2.

(3) *Output dependence*: Two statements are *output-dependent* if they produce (write) the same output variable. S1 $\circ\!\!\rightarrow$ S2 indicates output dependence from S1 to S2.

(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 is referenced by both I/O statements.

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

- The subscript of a variable is itself subscribed.
- 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 nonlinear in the loop index variable.

When one or more of these conditions exist, a conservative assumption is to claim unknown dependence among the statements involved.

Example 2.1 Data dependence in programs

Consider the following code fragment of four instructions:

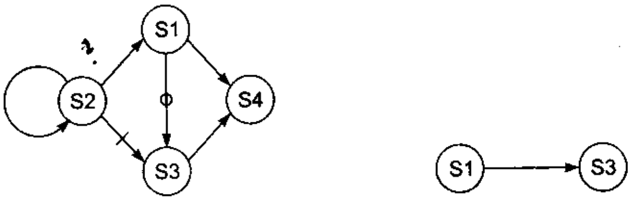
```
S1:        Load R1, A        /R1 ← Memory(A)/
S2:        Add R2, R1        /R2 ← (R1) + (R2)/
S3:        Move R1, R3       /R1 ← (R3)/
S4:        Store B, R1       /Memory(B) ← (R1)/
```

As illustrated in Fig. 2.1a, S2 is flow-dependent on S1 because the variable A is passed via the register R1. S3 is antidependent on S2 because of potential conflicts in register content in R1. S3 is output-dependent on S1 because they both modify the same register R1. Other data dependence relationships can be similarly revealed on a pairwise basis. Note that dependence is a partial ordering relation; that is, the members of not every pair of statements are related. For example, the statements S2 and S4 in the above program are totally *independent*.

Next, we consider a code fragment involving I/O operations:

```
S1:        Read (4), A(I)      /Read array A from file 4/
S2:        Process             /Process data/
S3:        Write (4), B(I)     /Write array B into file 4/
S4:        Close (4)           /Close file 4/
```

As shown in Fig. 2.1b, the read/write statements, S1 and S3, are I/O-dependent on each other because they both access the same file. The above data dependence relations should not be arbitrarily violated during program execution. Otherwise, erroneous results may be produced with changed program order. The order in which statements are executed in a sequential program is well defined and repetitive runs produce identical results. On a multiprocessor system, the program order may or may not be preserved, depending on the memory model used. Determinism yielding predictable results can be controlled by a programmer as well as by constrained modification of writable data in a shared memory.



(a) Dependence graph

(b) I/O dependence caused by accessing the same file by the read and write statements

Fig. 2.1 Data and I/O dependences in the program of Example 2.1

✓ **Control Dependence** This refers to the situation where the order of execution of statements cannot be determined before run time. For example, conditional statements will not be resolved until run time. Different paths taken after a conditional branch may introduce or eliminate data dependence among instructions. Dependence may also exist between operations performed in successive iterations of a looping procedure. In the following, we show one loop example with and another without control-dependent iterations. The successive iterations of the following loop are *control-independent*:

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

The following loop has *control-dependent* iterations:

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

Control dependence often prohibits parallelism from being exploited. Compiler techniques or hardware branch prediction techniques are needed to get around the control dependence in order to exploit more parallelism.

Resource Dependence This is different from data or control dependence, which demands the independence of the work to be done. *Resource dependence* is concerned with the conflicts in using shared resources, such as integer units, floating-point units, registers, and memory areas, among parallel events. When the conflicting resource is an ALU, we call it *ALU dependence.*

If the conflicts involve workplace storage, we call it *storage dependence*. In the case of storage dependence, each task must work on independent storage locations or use protected access (such as locks or monitors to be described in Chapter 11) to shared writable data.

The detection of parallelism in programs requires a check of the various dependence relations.

Bernstein's Conditions In 1966, Bernstein revealed a set of conditions based on which two processes can execute in parallel. A *process* is a software entity corresponding to the abstraction of a program fragment defined at various processing levels. We define the *input set* I_i of a process P_i as the set of all input variables needed to execute the process.

Similarly, the *output set* O_i consists of all output variables generated after execution of the process P_i. Input variables are essentially operands which can be fetched from memory or registers, and output variables are the results to be stored in working registers or memory locations.

Now, consider two processes P_1 and P_2 with their input sets I_1 and I_2 and output sets O_1 and O_2, respectively. These two processes can execute in parallel and are denoted $P_1 \parallel P_2$ if they are independent and therefore create deterministic results.

Formally, these conditions are stated as follows:

$$\left.\begin{aligned} I_1 \cap O_2 &= \phi \\ I_2 \cap O_1 &= \phi \\ O_1 \cap O_2 &= \phi \end{aligned}\right\} \qquad (2.1)$$

These three conditions are known as *Bernstein's conditions.* The input set I_i is also called the *read set* or the *domain* of P_i by other authors. Similarly, the output set O_i has been called the *write set* or the *range* of a process P_i. In terms of data dependences, Bernstein's conditions simply imply that two processes can execute in parallel if they are flow-independent, antiindependent, and output-independent.

The parallel execution of such two processes produces the same results regardless of whether they are executed sequentially in any order or in parallel. This is because the output of one process will not be used as input to the other process. Furthermore, the two processes do not modify (write) the same set of variables, either in memory or in the registers.

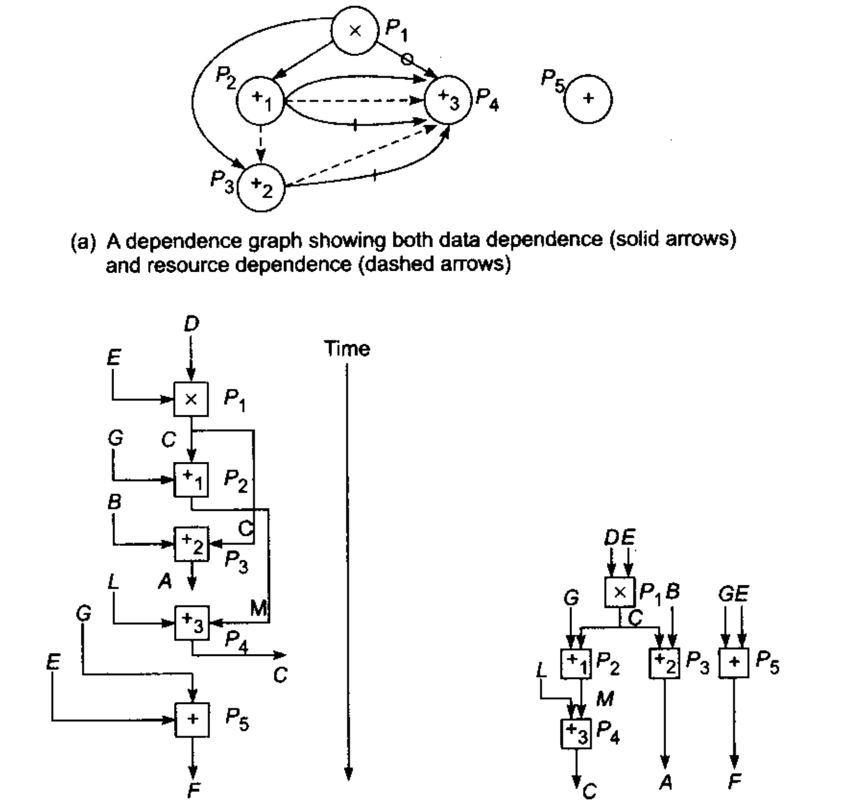
In general, a set of processes, $P_1, P_2, \ldots, P_k$, can execute in parallel if Bernstein's conditions are satisfied on a pairwise basis; that is, $P_1 \parallel P_2 \parallel P_3 \parallel \ldots \parallel P_k$ if and only if $P_i \parallel P_j$ for all $i \neq j$. This is exemplified by the following program illustrated in Fig. 2.2.

Example 2.2 Detection of parallelism in a program using Bernstein's conditions

Consider the simple case in which each process is a single HLL statement. We want to detect the parallelism embedded in the following five statements labeled P_1, P_2, P_3, P_4, and P_5 in program order.

$$\left.\begin{aligned} P_1 &: C = D \times E\\ P_2 &: M = G + C\\ P_3 &: A = B + C\\ P_4 &: C = L + M\\ P_5 &: F = G \div E \end{aligned}\right\} \qquad (2.2)$$

Assume that each statement requires one step to execute. No pipelining is considered here. The dependence graph shown in Fig. 2.2a demonstrates flow dependence as well as resource dependence. In sequential execution, five steps are needed (Fig. 2.2b).



(a) A dependence graph showing both data dependence (solid arrows) and resource dependence (dashed arrows)

(b) Sequential execution in five steps, assuming one step per statement (no pipelining)

(c) Parallel execution in three steps, assuming two adders are available per step

Fig. 2.2 Detection of parallelism in the program of Example 2.2

If two adders are available simultaneously, the parallel execution requires only three steps as shown in Fig. 2.2c. Pairwise, there are 10 pairs of statements to check against Bernstein's conditions. Only 5 pairs, $P_1 \| P_5$, $P_2 \| P_3$, $P_2 \| P_5$, $P_5 \| P_3$, and $P_4 \| P_5$, can execute in parallel as revealed in Fig. 2.2a if there are no resource conflicts. Collectively, only $P_2 \parallel P_3 \parallel P_5$ is possible (Fig. 2.2c) because $P_2 \parallel P_3$, $P_3 \parallel P_5$, and $P_5 \parallel P_2$ are all possible.

In general, the parallelism relation $\parallel$ is commutative; i.e., $P_i \parallel P_j$ implies $P_j \parallel P_i$. But the relation is not transitive; i.e., $P_i \parallel P_j$ and $P_j \parallel P_k$ do not necessarily guarantee $P_i \parallel P_k$. For example, we have $P_1 \parallel P_5$ and $P_5 \parallel P_2$, but $P_1 \nparallel P_2$, where $\nparallel$ means P_1 and P_2 cannot execute in parallel. In other words, the order in which P_1 and P_2 are executed will make a difference in the computational results.

Therefore, $\parallel$ is not an equivalence relation. However, $P_i \parallel P_j \parallel P_k$ implies associativity; i.e. $(P_i \parallel P_j) \parallel P_k = P_i \parallel (P_j \parallel P_k)$, since the order in which the parallel executable processes are executed should not make any difference in the output sets. It should be noted that the condition $I_i \cap I_j \neq \phi$ does not prevent parallelism between P_i and P_j.

Violations of any one or more of the three conditions in Eq. 2.1 prohibits parallelism between two processes. In general, violation of any one or more of the $3n(n-1)/2$ Bernstein's conditions among n processes prohibits parallelism collectively or partially.

In general, data dependence, control dependence, and resource dependence all prevent parallelism from being exploitable.

The statement-level dependence can be generalized to higher levels, such as code segment, subroutine, process, task, and program levels. The dependence of two higher level objects can be inferred from the dependence of statements in the corresponding objects. The goals of analyzing the data dependence, control dependence, and resource dependence in a code are to identify opportunities for parallelization or vectorization. Hardware techniques for detecting instruction-level parallelism in a running program are described in Chapter 12.

Very often program restructuring or code transformations need to be performed before such opportunities can be revealed. The dependence relations are also used in instruction issue and pipeline scheduling operations described in Chapter 6 and 12.

2.1.2 Hardware and Software Parallelism ✓

For implementation of parallelism, we need special hardware and software support. In this section, we address these support issues. We first distinguish between hardware and software parallelism. The mismatch problem between hardware and software is discussed. Then we describe the fundamental concept of compilation support needed to close the gap between hardware and software.

Details of special hardware functions and software support for parallelism will be treated in the remaining chapters. The key idea being conveyed is that parallelism cannot be achieved free. Besides theoretical conditioning, joint efforts between hardware designers and software programmers are needed to exploit parallelism in upgrading computer performance.

Hardware Parallelism This refers to the type of parallelism defined by the machine architecture and hardware multiplicity. Hardware parallelism is often a function of cost and performance tradeoffs. It displays the resource utilization patterns of simultaneously executable operations. It can also indicate the peak performance of the processor resources.

One way to characterize the parallelism in a processor is by the number of instruction issues per machine cycle. If a processor issues k instructions per machine cycle, then it is called a *k-issue* processor.

A conventional pipelined processor takes one machine cycle to issue a single instruction. These types of processors are called *one-issue* machines, with a single instruction pipeline in the processor. In a modern processor, two or more instructions can be issued per machine cycle.

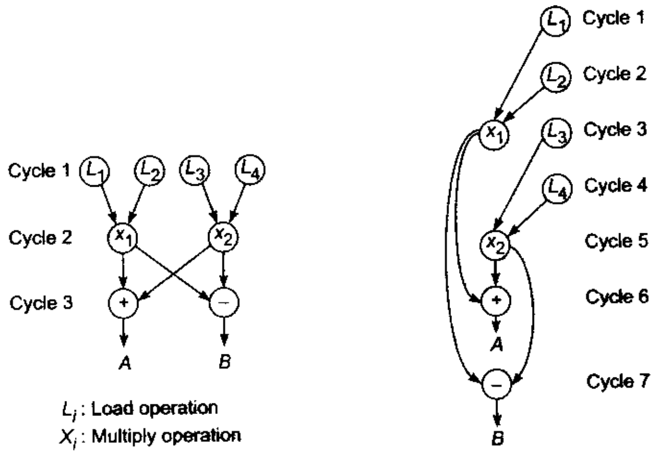
For example, the Intel i960CA was a three-issue processor with one arithmetic, one memory access, and one branch instruction issued per cycle. The IBM RISC/System 6000 is a four-issue processor capable of issuing one arithmetic, one memory access, one floating-point, and one branch operation per cycle.

Software Parallelism This type of parallelism is revealed in the program profile or in the program flow graph. Software parallelism is a function of algorithm, programming style, and program design. The program flow graph displays the patterns of simultaneously executable operations.

Example 2.3 Mismatch between software parallelism and hardware parallelism (Wen-Mei Hwu, 1991)

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 4 to 2 in three cycles. The average software parallelism is equal to 8/3 = 2.67 instructions per cycle in this example program.

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.



(a) Software parallelism

(b) Hardware parallelism

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

Let us try to 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, where *L/S* stands for *load/store* operations. Note that six processor cycles are needed to execute the 12 instructions by two processors. S_1 and S_2 are two inserted *store* operations, and l_5 and l_6 are two inserted *load* operations. These added instructions are needed for interprocessor communication through the shared memory.

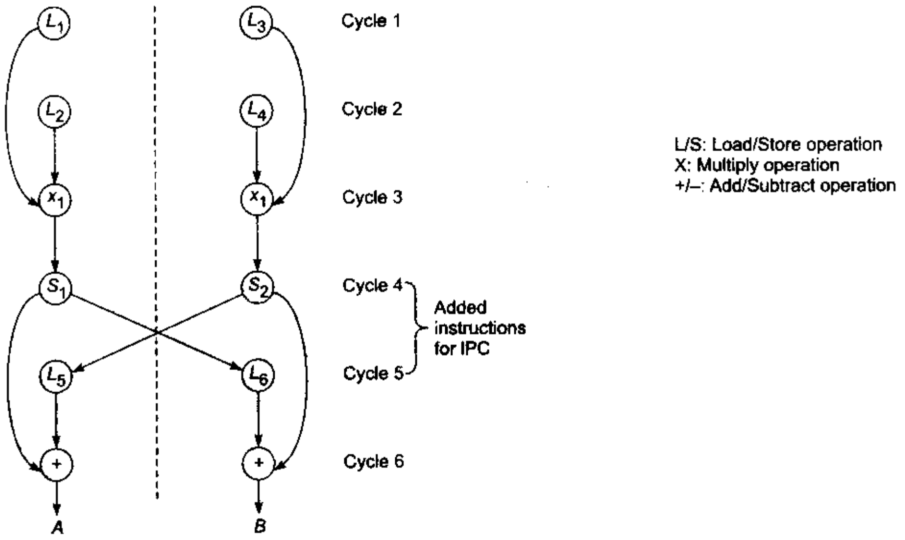


Fig. 2.4 Dual-processor execution of the program in Fig. 2.3a

Of the many types of software parallelism, two are most frequently cited as important to parallel programming: The first is *control parallelism*, which allows two or more operations to be performed simultaneously. The second type has been called *data parallelism*, in which almost the same operation is performed over many data elements by many processors simultaneously.

Control parallelism, appearing in the form of pipelining or multiple functional units, is limited by the pipeline length and by the multiplicity of functional units. Both pipelining and functional parallelism are handled by the hardware; programmers need take no special actions to invoke them.

Data parallelism offers the highest potential for concurrency. It is practiced in both SIMD and MIMD modes on MPP systems. Data parallel code is easier to write and to debug than control parallel code. Synchronization in SIMD data parallelism is handled by the hardware. Data parallelism exploits parallelism in proportion to the quantity of data involved. Thus data parallel computations appeal to scaled problems, in which the performance of an MPP system does not drop sharply with the possibly small sequential fraction in the program.

To solve the mismatch problem between software parallelism and hardware parallelism, one approach is to develop compilation support, and the other is through hardware redesign for more efficient exploitation of parallelism. These two approaches must cooperate with each other to produce the best result.

Hardware processors can be better designed to exploit parallelism by an optimizing compiler. Pioneer work in processor technology with this objective was seen in the IBM 801, Stanford MIPS, and Berkeley RISC. Such processors use a large register file and sustained instruction pipelining to execute nearly one instruction per cycle. The large register file supports fast access to temporary values generated by an optimizing compiler. The registers are exploited by the code optimizer and global register allocator in such a compiler.

The instruction scheduler exploits the pipeline hardware by filling *branch* and *load* delay slots. In superscalar processors, hardware and software branch prediction, multiple instruction issue, speculative execution, high bandwidth instruction cache, and support for dynamic scheduling are needed to facilitate the detection of parallelism opportunities. Further discussion on these topics can be found in Chapters 6 and 12.

2.1.3 The Role of Compilers

Compiler techniques are used to exploit hardware features to improve performance. The pioneer work on the IBM PL.8 and Stanford MIPS compilers aimed for this goal. Other early optimizing compilers for exploiting parallelism included the CDC STACKLIB, Cray CFT, Illinois Parafrase, Rice PFC, Yale Bulldog, and Illinois IMPACT.

In Chapter 10, we will study loop transformation, software pipelining, and features developed in existing optimizing compilers for supporting parallelism. Interaction between compiler and architecture design is a necessity in modern computer development. Conventional scalar processors issue at most one instruction per cycle and provide a few registers. This may cause excessive spilling of temporary results from the available registers. Therefore, more software parallelism may not improve performance in conventional scalar processors.

There exists a vicious cycle of limited hardware support and the use of a naive compiler. To break the cycle, ideally one must design the compiler and the hardware jointly at the same time. Interaction between the two can lead to a better solution to the mismatch problem between software and hardware parallelism.

The general guideline is to increase the flexibility in hardware parallelism and to exploit software parallelism in control-intensive programs. Hardware and software design tradeoffs also exist in terms of cost, complexity, expandability, compatibility, and performance. Compiling for multiprocessors is much more challenging than for uniprocessors. Both granularity and communication latency play important roles in the code optimization and scheduling process.

2.2 PROGRAM PARTITIONING AND 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.

2.2.1 Grain Sizes and Latency

Grain size or *granularity* is a measure of the amount of computation involved in a software process. The simplest measure is to count the number of instructions in a grain (program segment). Grain size determines the basic program segment chosen for parallel processing. Grain sizes are commonly described as *fine, medium,* or *coarse,* depending on the processing levels involved.

Latency is a time measure of the communication overhead incurred between machine subsystems. For example, the *memory latency* is the time required by a processor to access the memory. The time required for two processes to synchronize with each other is called the *synchronization latency*. Computational granularity and communication latency are closely related, as we shall see below.

Parallelism has been exploited at various processing levels. As illustrated in Fig. 2.5, five levels of program execution represent different computational grain sizes and changing communication and control requirements. The lower the level, the finer the granularity of the software processes.

In general, the execution of a program may involve a combination of these levels. The actual combination depends on the application, formulation, algorithm, language, program, compilation support, and hardware characteristics. We characterize below the parallelism levels and review their implementation issues from the viewpoints of a programmer and of a compiler writer.

Instruction Level At the lowest level, a typical grain contains less than 20 instructions, called *fine grain* in Fig. 2.5. Depending on individual programs, fine-grain parallelism at this level may range from two to thousands. Butler et al. (1991) has shown that single-instruction-stream parallelism is greater than two. Wall (1991) finds that the average parallelism at instruction level is around five, rarely exceeding seven, in an ordinary program. For scientific applications, Kumar (1988) has measured the average parallelism in the range of 500 to 3000 Fortran statements executing concurrently in an idealized environment.

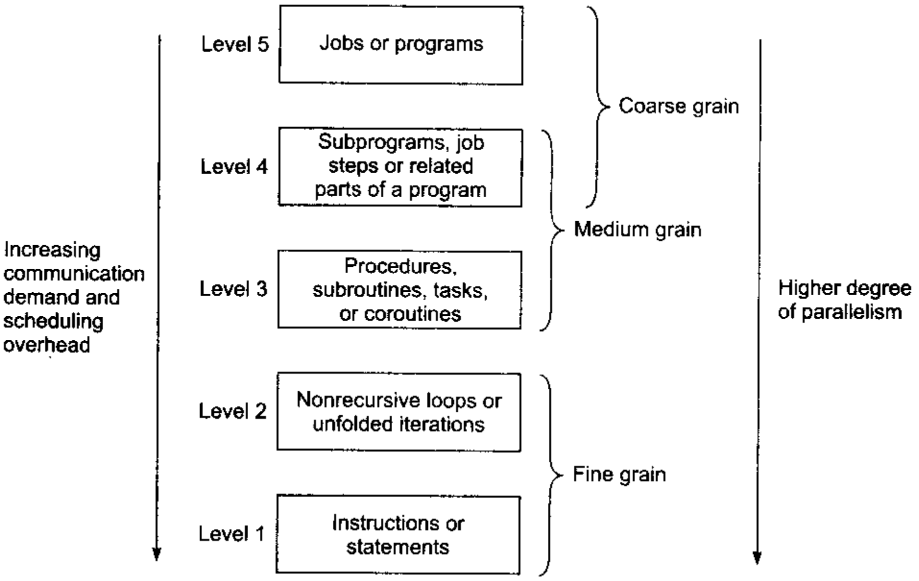

Level 5
Jobs or programs
Coarse grain
Level 4
Subprograms, job steps or related parts of a program
Medium grain
Increasing communication demand and scheduling overhead
Level 3
Procedures, subroutines, tasks, or coroutines
Higher degree of parallelism
Level 2
Nonrecursive loops or unfolded iterations
Fine grain
Level 1
Instructions or statements

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

The exploitation of fine-grain parallelism can be assisted by an optimizing compiler which should be able to automatically detect parallelism and translate the source code to a parallel form which can be recognized by the run-time system. Instruction-level parallelism can be detected and exploited within the processors, as we shall see in Chapter 12.

Loop Level This corresponds to the iterative loop operations. A typical loop contains less than 500 instructions. Some loop operations, if independent in successive iterations, can be vectorized for pipelined execution or for lock-step execution on SIMD machines. Some loop operations can be self-scheduled for parallel execution on MIMD machines.

Loop-level parallelism is often the most optimized program construct to execute on a parallel or vector computer. However, recursive loops are rather difficult to parallelize. Vector processing is mostly exploited at the loop level (level 2 in Fig. 2.5) by a vectorizing compiler. The loop level may also be considered a fine grain of computation.

Procedure Level This level corresponds to medium-grain parallelism at the task, procedural, subroutine, and coroutine levels. A typical grain at this level contains less than 2000 instructions. Detection of parallelism at this level is much more difficult than at the finer-grain levels. Interprocedural dependence analysis is much more involved and history-sensitive.

Communication requirement is often less compared with that required in MIMD execution mode. SPMD execution mode is a special case at this level. Multitasking also belongs in this category. Significant efforts by programmers may be needed to restructure a program at this level, and some compiler assistance is also needed.

Subprogram Level This corresponds to the level of job steps and related subprograms. The grain size may typically contain tens or hundreds of thousands of instructions. Job steps can overlap across different jobs. Subprograms can be scheduled for different processors in SPMD or MPMD mode, often on message-passing multicomputers.

Multiprogramming on a uniprocessor or on a multiprocessor is conducted at this level. Traditionally, parallelism at this level has been exploited by algorithm designers or programmers, rather than by compilers. Good compilers for exploiting medium- or coarse-grain parallelism require suitably designed parallel programming languages.

Job (Program) Level This corresponds to the parallel execution of essentially independent jobs (programs) on a parallel computer. The grain size can be as high as millions of instructions in a single program. For supercomputers with a small number of very powerful processors, such coarse-grain parallelism is practical. Job-level parallelism is handled by the program loader and by the operating system in general. Time-sharing or space-sharing multiprocessors explore this level of parallelism. In fact, both time and space sharing are extensions of multiprogramming.

To summarize, fine-grain parallelism is often exploited at instruction or loop levels, preferably assisted by a parallelizing or vectorizing compiler. Medium-grain parallelism at the task or job step demands significant roles for the programmer as well as compilers. Coarse-grain parallelism at the program level relies heavily on an effective OS and on the efficiency of the algorithm used. Shared-variable communication is often used to support fine-grain and medium-grain computations.

Message-passing multicomputers have been used for medium- and coarse-grain computations. Massive parallelism is often explored at the fine-grain level, such as data parallelism on SIMD or MIMD computers.

Communication Latency By balancing granularity and latency, one can achieve better performance of a computer system. Various latencies are attributed to machine architecture, implementing technology, and communication patterns involved. The architecture and technology affect the design choices for latency tolerance between subsystems. In fact, latency imposes a limiting factor on the scalability of the machine size. For example, over the years memory latency has increased with respect to processor cycle time. Various latency hiding or tolerating techniques will be studied in Chapters 9 and 12.

The latency incurred with interprocessor communication is another important parameter for a system designer to minimize. Besides signal delays in the data path, IPC latency is also affected by the communication patterns involved. In general, n tasks communicating with each other may require $n(n-1)/2$ communication links among them. Thus the complexity grows quadratically. This leads to a communication bound which limits the number of processors allowed in a large computer system.

Communication patterns are determined by the algorithms used as well as by the architectural support provided. Frequently encountered patterns include *permutations* and *broadcast, multicast*, and *conference* (many-to-many) communications. The communication demand may limit the granularity or parallelism. Very often tradeoffs do exist between the two.

The communication issue thus involves the reduction of latency or complexity, the prevention of deadlock, minimizing blocking in communication patterns, and the tradeoff between parallelism and communication overhead. We will study techniques that minimize communication latency, prevent deadlock, and optimize grain size in latter chapters of the book.

2.2.2 Grain Packing and Scheduling

Two fundamental questions to ask in parallel programming are: (i) How can we partition a program into parallel branches, program modules, microtasks, or grains to yield the shortest possible execution time? and (ii) What is the optimal size of concurrent grains in a computation?

This grain-size problem demands determination of both the number and the size of grains (or microtasks) in a parallel program. Of course, the solution is both problem-dependent and machine-dependent. The goal is to produce a short schedule for fast execution of subdivided program modules.

There exists a tradeoff between parallelism and scheduling/synchronization overhead. The time complexity involves both computation and communication overheads. The program partitioning involves the algorithm designer, programmer, compiler, operating system support, etc. We describe below a *grain packing* approach introduced by Kruatrachue and Lewis (1988) for parallel programming applications.

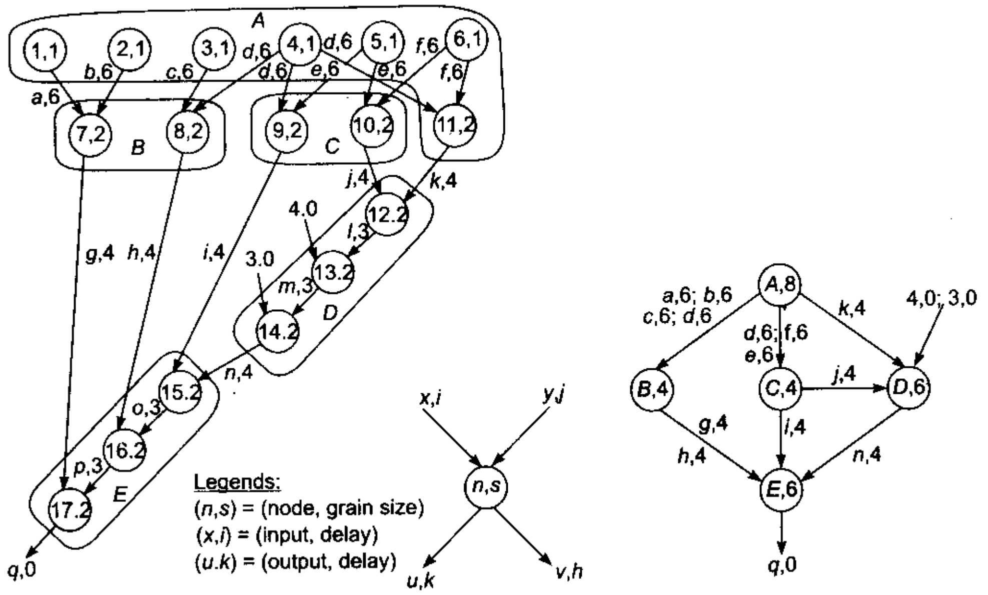
Example 2.4 Program graph before and after grain packing (Kruatrachue and Lewis, 1988)

The basic concept of program partitioning is introduced below. In Fig. 2.6, we show an example *program graph* in two different grain sizes. A program graph shows the structure of a program. It is very similar to the dependence graph introduced in Section 2.1.1. Each node in the program graph corresponds to a computational unit in the program. The *grain size* is measured by the number of basic machine cycles (including both processor and memory cycles) needed to execute all the operations within the node.

We denote each node in Fig. 2.6 by a pair (n, s), where n is the *node name* (id) and s is the grain size of the node. Thus grain size reflects the number of computations involved in a program segment. Fine-grain nodes have a smaller grain size, and coarse-grain nodes have a larger grain size.

The edge label (v, d) between two end nodes specifies the output variable v from the source node or the input variable to the destination node, and the communication delay d between them. This delay includes all the path delays and memory latency involved.

There are 17 nodes in the fine-grain program graph (Fig. 2.6a) and 5 in the coarse-grain program graph (Fig. 2.6b). The coarse-grain node is obtained by combining (grouping) multiple fine-grain nodes. The fine grain corresponds to the following program:



(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)**

```
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 × b
    8. h := c × d
    9. i := d × e
    10. j := e × f
    11. k := d × f
    12. l := j × k
    13. m := 4 × 1
    14. n := 3 × m
    15. o := n × i
    16. p := o × h
    17. q := p × q
End
```

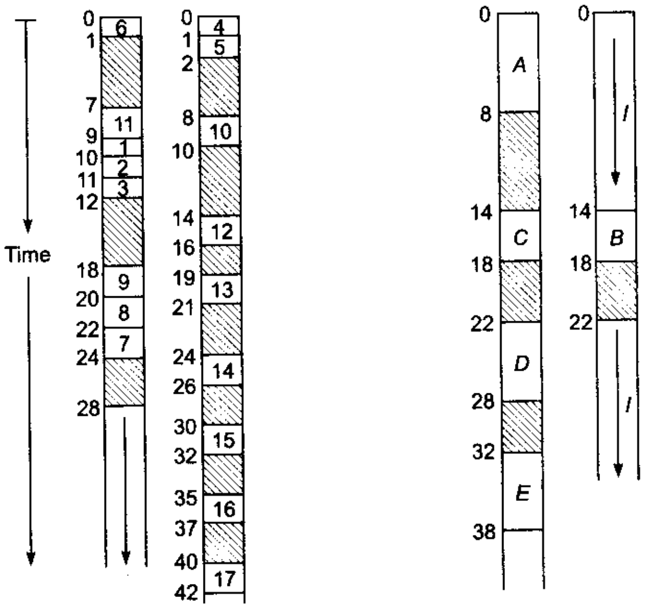
Nodes 1, 2, 3, 4, 5, and 6 are memory reference (data fetch) operations. Each takes one cycle to address and six cycles to fetch from memory. All remaining nodes (7 to 17) are CPU operations, each requiring two cycles to complete. After packing, the coarse-grain nodes have larger grain sizes ranging from 4 to 8 as shown.

The node (A, 8) in Fig. 2.6b is obtained by combining the nodes (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), and (11, 2) in Fig. 2.6a. The grain size, 8, of node A is the summation of all grain sizes (1 + 1 + 1 + 1 + 1 + 1 + 2 = 8) being combined.

The idea of grain packing is to apply fine grain first in order to achieve a higher degree of parallelism. Then one combines (packs) multiple fine-grain nodes into a coarsegrain node if it can eliminate unnecessary communications delays or reduce the overall scheduling overhead.

Usually, all fine-grain operations within a single coarse-grain node are assigned to the same processor for execution. Fine-grain partition of a program often demands more interprocessor communication than that required in a coarse-grain partition. Thus grain packing offers a tradeoff between parallelism and scheduling/communication overhead.

Internal delays among fine-grain operations within the same coarse-grain node are negligible because the communication delay is contributed mainly by interprocessor delays rather than by delays within the same processor. The choice of the optimal grain size is meant to achieve the shortest schedule for the nodes on a parallel computer system.



(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)

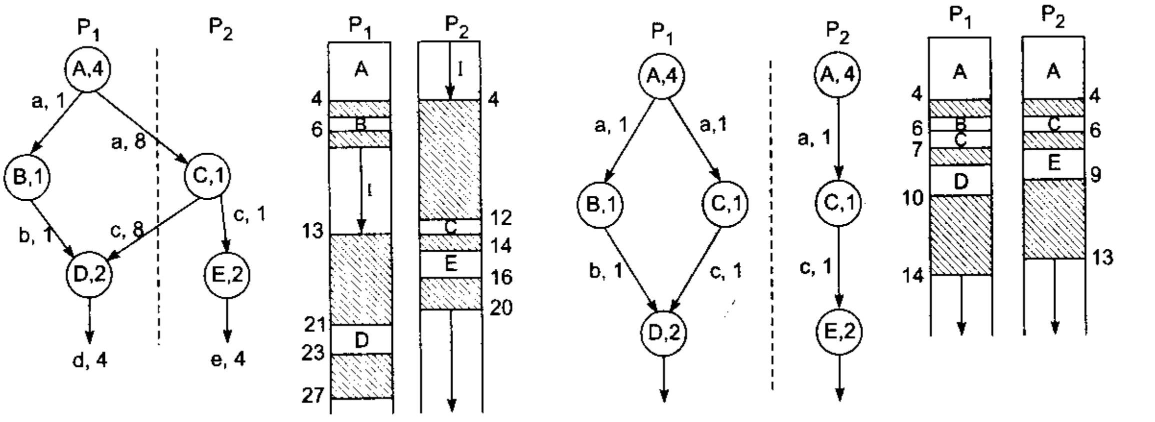
With respect to the fine-grain versus coarse-grain program graphs in Fig. 2.6, two multiprocessor schedules are shown in Fig. 2.7. The fine-grain schedule is longer (42 time units) because more communication delays were included as shown by the shaded area. The coarse-grain schedule is shorter (38 time units) because communication delays among nodes 12, 13, and 14 within the same node D (and also the delays among 15, 16, and 17 within the node E) are eliminated after grain packing.

2.2.3 Static Multiprocessor Scheduling

Grain packing may not always produce a shorter schedule. In general, dynamic multiprocessor scheduling is an NP-hard problem. Very often heuristics are used to yield suboptimal solutions. We introduce below the basic concepts behind multiprocessor scheduling using static schemes.

Node Duplication In order to eliminate the idle time and to further reduce the communication delays among processors, one can duplicate some of the nodes in more than one processor.

Figure 2.8a shows a schedule without duplicating any of the five nodes. This schedule contains idle time as well as long interprocessor delays (8 units) between P1 and P2. In Fig. 2.8b, node A is duplicated into A′ and assigned to P2 besides retaining the original copy A in P1. Similarly, a duplicated node C′ is copied into P1 besides the original node C in P2. The new schedule shown in Fig. 2.8b is almost 50% shorter than that in Fig. 2.8a. The reduction in schedule time is caused by elimination of the (a, 8) and (c, 8) delays between the two processors.



(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)

Grain packing and node duplication are often used jointly to determine the best grain size and corresponding schedule. 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. Perform grain packing to produce the coarse grains.

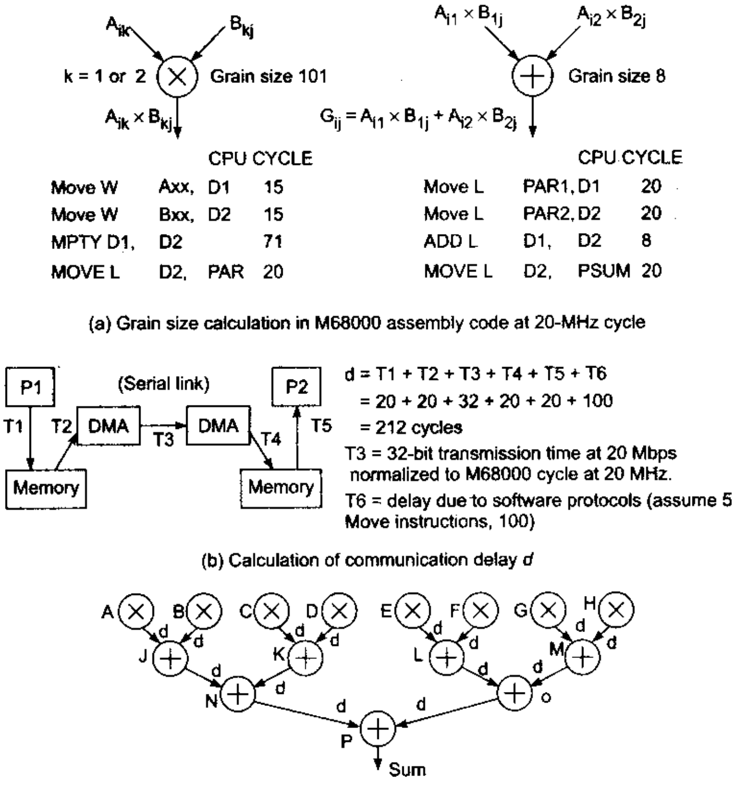
Step 4. Generate a parallel schedule based on the packed graph.

The purpose of multiprocessor scheduling is to obtain a minimal time schedule for the computations involved. The following example clarifies this concept.

Example 2.5 Program decomposition for static multiprocessor scheduling (Kruatrachue and Lewis, 1988)

Figure 2.9 shows an example of how to calculate the grain size and communication latency. In this example, two 2 × 2 matrices A and B are multiplied to compute the sum of the four elements in the resulting product matrix $C = A \times B$. There are eight multiplications and seven additions to be performed in this program, as written below:

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



```
                   CPU CYCLE                                CPU CYCLE
Move W     Axx, D1      15              Move L    PAR1, D1      20
Move W     Bxx, D2      15              Move L    PAR2, D2      20
MPTY D1,   D2           71              ADD L     D1,  D2       8
MOVE L     D2,  PAR     20              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

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)

$$\begin{aligned}
C_{11} &= A_{11} \times B_{11} + A_{12} \times B_{21} \\
C_{12} &= A_{11} \times B_{12} + A_{12} \times B_{22} \\
C_{21} &= A_{21} \times B_{11} + A_{22} \times B_{21} \\
C_{22} &= A_{21} \times B_{12} + A_{22} \times B_{22} \\
\text{Sum} &= C_{11} + C_{12} + C_{21} + C_{22}
\end{aligned}$$

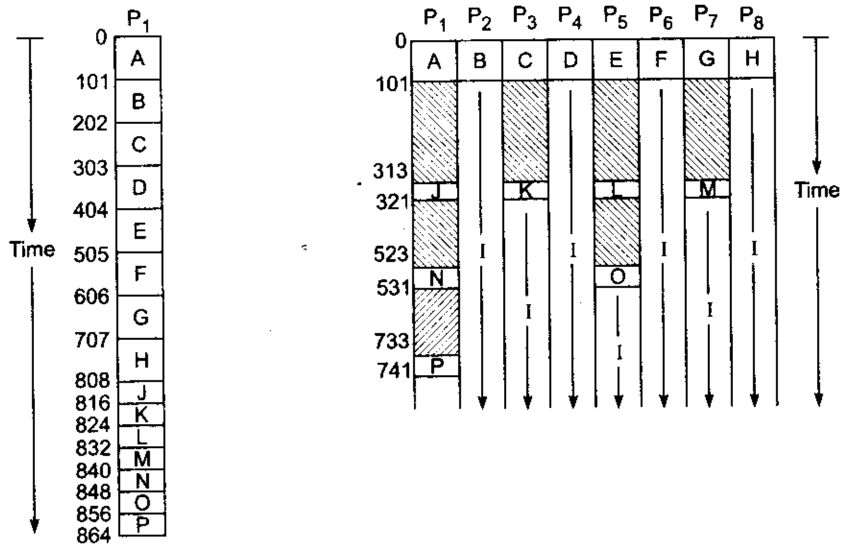
As shown in Fig. 2.9a, the eight multiplications are performed in eight ⊗ nodes, each of which has a grain size of 101 CPU cycles. The remaining seven additions are performed in a 3-level binary tree consisting of seven ⊕ nodes. Each additional node requires 8 CPU cycles.

The interprocessor communication latency along all edges in the program graph is eliminated as $d = 212$ cycles by adding all path delays between two communicating processors (Fig. 2.9b).

A fine-grain program graph is thus obtained in Fig. 2.9c. Note that the grain size and communication delay may vary with the different processors and communication links used in the system.

Figure 2.10 shows scheduling of the fine-grain program first on a sequential uniprocessor (P1) and then on an eight-processor (P1 to P8) system (Step 2). Based on the fine-grain graph (Fig. 2.9c), the sequential execution requires 864 cycles to complete without incurring any communication delay.

Figure 2.10b shows the reduced schedule of 741 cycles needed to execute the 15 nodes on 8 processors with incurred communication delays (shaded areas). Note that the communication delays have slowed down the parallel execution significantly, resulting in many processors idling (indicated by I), except for P1 which produces the final sum. A speedup factor of 864/741 = 1.16 is observed.

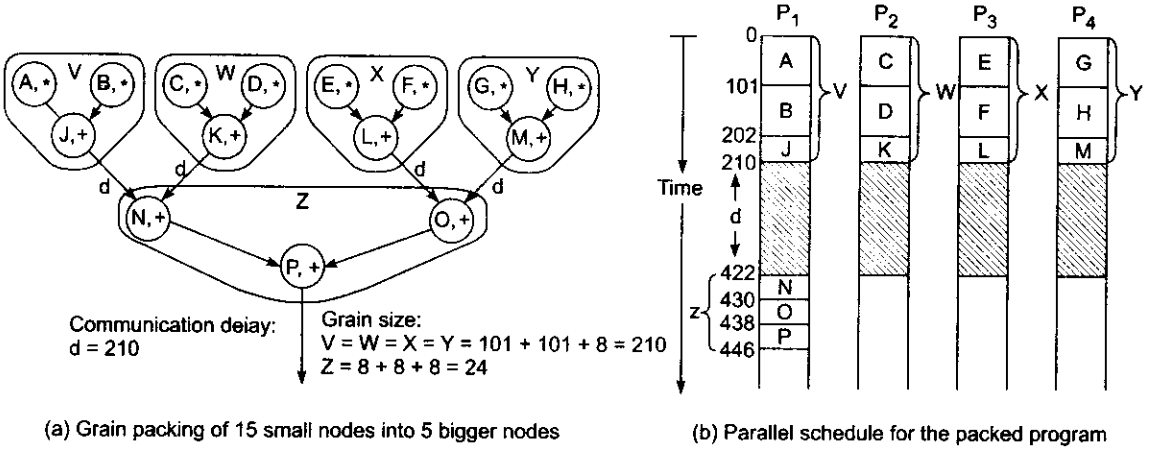


(a) A sequential schedule

(b) A parallel schedule

Fig. 2.10 Sequential versus parallel scheduling in Example 2.5

Next we show how to use grain packing (Step 3) to reduce the communication overhead. As shown in Fig. 2.11, we group the nodes in the top two levels into four coarse-grain nodes labeled V, W, X, and Y. The remaining three nodes (N, O, P) then form the fifth node Z. Note that there is only one level of interprocessor communication required as marked by *d* in Fig. 2.11a.



(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

Since the maximum degree of parallelism is now reduced to 4 in the program graph, we use only four processors to execute this coarse-grain program. A parallel schedule is worked out (Fig. 2.11) for this program in 446 cycles, resulting in an improved speedup of 864/446 = 1.94.

2.3 PROGRAM FLOW MECHANISMS

Conventional computers are based on a control flow mechanism by which the order of program execution is explicitly stated in the user programs. Dataflow computers are based on a data-driven mechanism which allows the execution of any instruction to be driven by data (operand) availability. Dataflow computers emphasize a high degree of parallelism at the fine-grain instructional level. Reduction computers are based on a demand-driven mechanism which initiates an operation based on the demand for its results by other computations.

2.3.1 Control Flow Versus Data Flow

Conventional von Neumann computers use a *program counter* (PC) to sequence the execution of instructions in a program. The PC is sequenced by instruction flow in a program. This sequential execution style has been called *control-driven*, as program flow is explicitly controlled by programmers.

A uniprocessor computer is inherently sequential, due to use of the control driven mechanism. However, control flow can be made parallel by using parallel language constructs or parallel compilers. In this book, we study primarily parallel control-flow computers and their programming techniques. Until the data-driven or demand-driven mechanism is proven to be cost-effective, the control-flow approach will continue to dominate the computer industry.

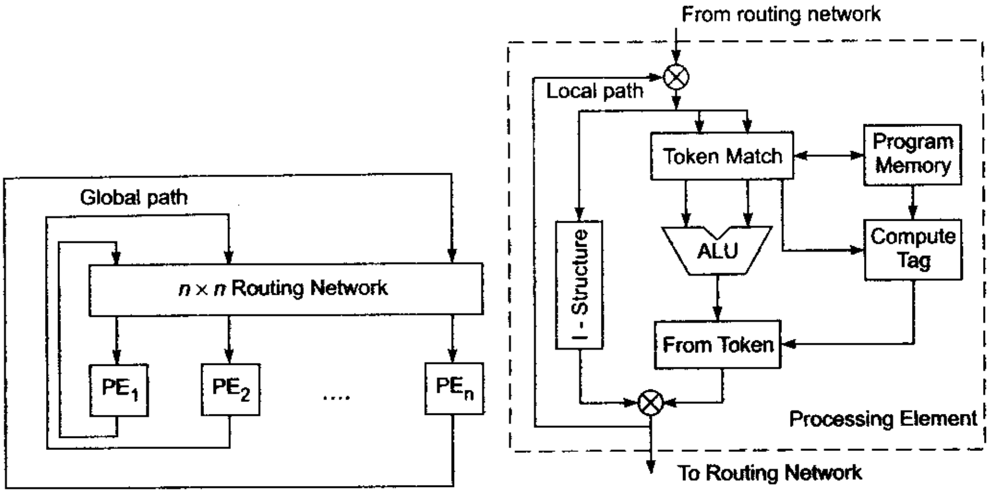
In a *dataflow computer*, the execution of an instruction is driven by data availability instead of being guided by a program counter. In theory, any instruction should be ready for execution whenever operands become available. The instructions in a data-driven program are not ordered in any way. Instead of being stored separately in a main memory, data are directly held inside instructions.

Computational results (*data tokens*) are passed directly between instructions. The data generated by an instruction will be duplicated into many copies and forwarded directly to all needy instructions. Data tokens, once consumed by an instruction, will no longer be available for reuse by other instructions.

This data-driven scheme requires no program counter, and no control sequencer. However, it requires special mechanisms to detect data availability, to match data tokens with needy instructions, and to enable the chain reaction of asynchronous instruction executions. No memory sharing between instructions results in no side effects.

Asynchrony implies the need for handshaking or token-matching operations. A pure dataflow computer exploits fine-grain parallelism at the instruction level. Massive parallelism would be possible if the data-driven mechanism could be cost-effectively implemented with low instruction execution overhead.

A Dataflow Architecture There have been quite a few experimental dataflow computer projects. Arvind and his associates at MIT developed a tagged-token architecture for building dataflow computers. As shown in Fig. 2.12, the global architecture consists of *n* processing elements (PEs) interconnected by an *n* × *n* routing network. The entire system supports pipelined dataflow operations in all *n* PEs. Inter-PE communications are done through the pipelined routing network.



(a) The global architecture

(b) Interior design of a processing element

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

Within each PE, the machine provides a low-level *token-matching* mechanism which dispatches only those instructions whose input data (tokens) are already available. Each datum is tagged with the address of the instruction to which it belongs and the context in which the instruction is being executed. Instructions are stored in the program memory. Tagged tokens enter the PE through a local path. The tokens can also be passed to other PEs through the routing network. All internal token circulation operations are pipelined without blocking.

One can think of the instruction address in a dataflow computer as replacing the program counter, and the context identifier replacing the frame base register in a control flow computer. It is the machine's job to match up data with the same tag to needy instructions. In so doing, new data will be produced with a new tag indicating the successor instruction(s). Thus, each instruction represents a synchronization operation. New tokens are formed and circulated along the PE pipeline for reuse or to other PEs through the global path, which is also pipelined.

Another synchronization mechanism, called the *I-structure*, is provided within each PE. The *I*-structure is a tagged memory unit for overlapped usage of a data structure by both the producer and consumer processes. Each word of *I*-structure uses a 2-bit tag indicating whether the word is *empty*, is *full*, or has *pending read* requests. The use of *I*-structure is a retreat from the pure dataflow approach. The purpose is to reduce excessive copying of large data structures in dataflow operations.

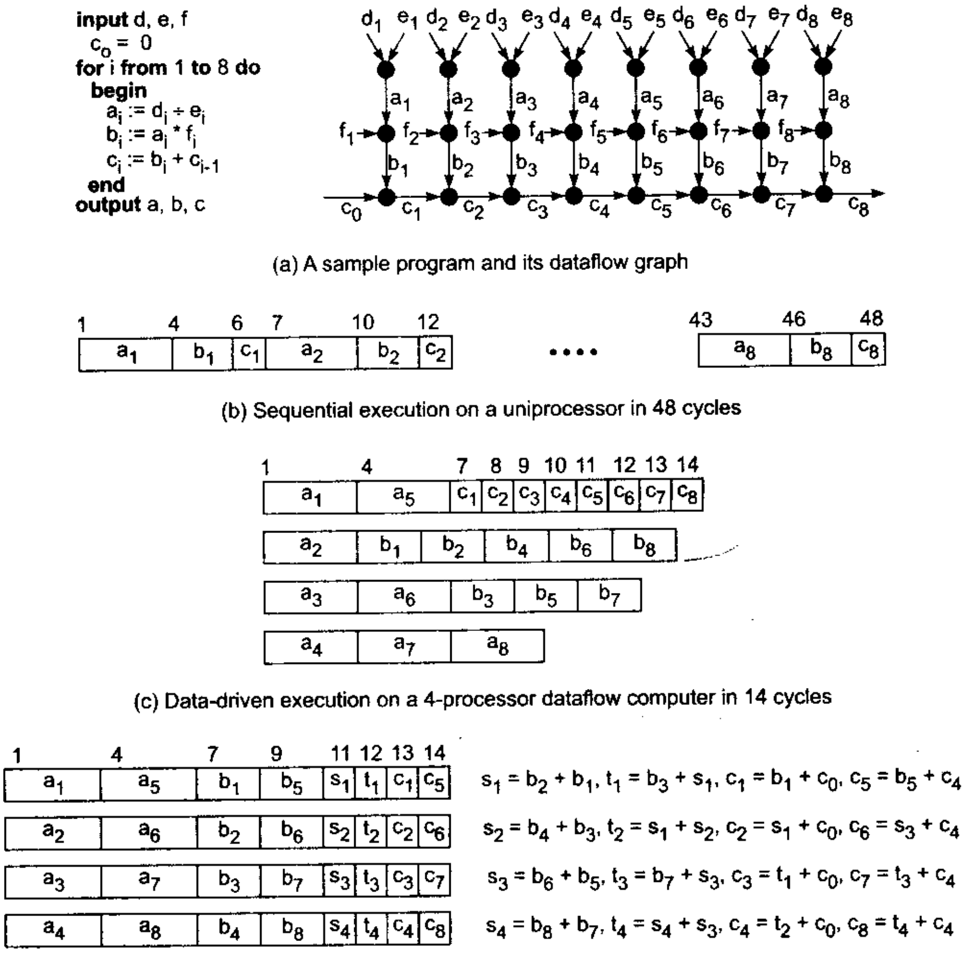
Example 2.6 Comparison of dataflow and control-flow computers (Gajski, Padua, Kuck, and Kuhn, 1982)

The dataflow graph in Fig. 2.13a shows that 24 instructions are to be executed (8 *divides*, 8 *multiplies*, and 8 *adds*). A dataflow graph is similar to a dependence graph or program graph. The only difference is that data tokens are passed around the edges in a dataflow graph. Assume that each *add*, *multiply*, and *divide* requires 1, 2, and 3 cycles to complete, respectively. Sequential execution of the 24 instructions on a control flow uniprocessor takes 48 cycles to complete, as shown in Fig. 2.13b.

On the other hand, a dataflow multiprocessor completes the execution in 14 cycles in Fig. 2.13c. Assume that all the external inputs (d_i, c_i, f_i for $i = 1, 2, \ldots, 8$ and c_0) are available before entering the loop. With four processors, instructions a_1, a_2, a_3, and a_4 are all ready for execution in the first three cycles. The results produced then trigger the execution of a_5, b_1, a_6, and a_7 starting from cycle 4. The data-driven chain reactions are shown in Fig. 2.13c. The output c_8 is the last one to produce, due to its dependence on all the previous c_i's.

Figure 2.13d shows the execution of the same set of computations on a conventional multiprocessor using shared memory to hold the intermediate results (s_i and t_i for $i = 1, 2, 3, 4$). Note that no shared memory is used in the dataflow implementation. The example does not show any time advantage of dataflow execution over control flow execution.

The theoretical minimum time is 13 cycles along the critical path $a_1b_1c_1c_2 \ldots c_8$. The chain reaction control in dataflow is more difficult to implement and may result in longer overhead, as compared with the uniform operations performed by all the processors in Fig. 2.13d.



```
input d, e, f
  c$_0$ = 0
for i from 1 to 8 do
  begin
    $a_i := d_i \div e_i$
    $b_i := a_i * f_i$
    $c_i := b_i + c_{i-1}$
  end
output a, b, c
```

d_1 e_1 d_2 e_2 d_3 e_3 d_4 e_4 d_5 e_5 d_6 e_6 d_7 e_7 d_8 e_8
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
f_1 f_2 f_3 f_4 f_5 f_6 f_7 f_8
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8
c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8

(a) A sample program and its dataflow graph

1 4 6 7 10 12 43 46 48
a_1 b_1 c_1 a_2 b_2 c_2 •••• a_8 b_8 c_8

(b) Sequential execution on a uniprocessor in 48 cycles

1 4 7 8 9 10 11 12 13 14
a_1 a_5 c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8
a_2 b_1 b_2 b_4 b_6 b_8
a_3 a_6 b_3 b_5 b_7
a_4 a_7 a_8

(c) Data-driven execution on a 4-processor dataflow computer in 14 cycles

1 4 7 9 11 12 13 14
a_1 a_5 b_1 b_5 s_1 t_1 c_1 c_5
a_2 a_6 b_2 b_6 s_2 t_2 c_2 c_6
a_3 a_7 b_3 b_7 s_3 t_3 c_3 c_7
a_4 a_8 b_4 b_8 s_4 t_4 c_4 c_8

$s_1 = b_2 + b_1,\ t_1 = b_3 + s_1,\ c_1 = b_1 + c_0,\ c_5 = b_5 + c_4$

$s_2 = b_4 + b_3,\ t_2 = s_1 + s_2,\ c_2 = s_1 + c_0,\ c_6 = s_3 + c_4$

$s_3 = b_6 + b_5,\ t_3 = b_7 + s_3,\ c_3 = t_1 + c_0,\ c_7 = t_3 + c_4$

$s_4 = b_8 + b_7,\ t_4 = s_4 + s_3,\ c_4 = t_2 + c_0,\ c_8 = t_4 + c_4$

(d) Parallel execution on a shared-memory 4-processor system in 14 cycles

Fig. 2.13 Comparison between dataflow and control-flow computers (adapted from Gajski, Padua, Kuck, and Kuhn, 1982; reprinted with permission from *IEEE Computer*, Feb. 1982)

One advantage of tagging each datum is that data from different contexts can be mixed freely in the instruction execution pipeline. Thus, instruction-level parallelism of dataflow graphs can absorb the communication latency and minimize the losses due to synchronization waits. Besides token matching and I-structure, compiler technology is also needed to generate dataflow graphs for tagged-token dataflow computers. The dataflow architecture offers in theory a promising model for massively parallel computations because all far-reaching side effects are removed. However, implementation of these concepts on a commercial scale has proved to be very difficult.

2.3.2 Demand-Driven Mechanisms

In a *reduction machine,* the computation is triggered by the demand for an operation's result. Consider the evaluation of a nested arithmetic expression $a = ((b + 1) \times c - (d \div e))$. The data-driven computation seen above chooses a bottom-up approach, starting from the innermost operations $b + 1$ and $d \div e$, then proceeding to the × operation, and finally to the outermost operation –. Such a computation has been called *eager evaluation* because operations are carried out immediately after all their operands become available.

A *demand-driven* computation chooses a top-down approach by first demanding the value of a, which triggers the demand for evaluating the next-level expressions $(b + 1) \times c$ and $d \div e$, which in turn triggers the demand for evaluating $b + 1$ at the innermost level. The results are then returned to the nested demander in the reverse order before a is evaluated.

A demand-driven computation corresponds to *lazy evaluation,* because operations are executed only when their results are required by another instruction. The demand driven approach matches naturally with the functional programming concept. The removal of side effects in functional programming makes programs easier to parallelize. There are two types of reduction machine models, both having a recursive control mechanism as characterized below.

Reduction Machine Models In a *string reduction* model, each demander gets a separate copy of the expression for its own evaluation. A long string expression is reduced to a single value in a recursive fashion. Each reduction step has an operator followed by an embedded reference to demand the corresponding input operands. The operator is suspended while its input arguments are being evaluated. An expression is said to be fully reduced when all the arguments have been replaced by literal values.

In a *graph reduction* model, the expression is represented as a directed graph. The graph is reduced by evaluation of branches or subgraphs. Different parts of a graph or subgraphs can be reduced or evaluated in parallel upon demand. Each demander is given a pointer to the result of the reduction. The demander manipulates all references to that graph.

Graph manipulation is based on sharing the arguments using pointers. This traversal of the graph and reversal of the references are continued until constant arguments are encountered. This proceeds until the value of a is determined and a copy is returned to the original demanding instruction.

2.3.3 Comparison of Flow Mechanisms

Control-flow, dataflow, and reduction computer architectures are compared in Table 2.1. The degree of explicit control decreases from control-driven to demand-driven to data-driven. Highlighted in the table are the differences between *eager evaluation* and *lazy evaluation* in data-driven and demand-driven computers, respectively.

Furthermore, control tokens are used in control-flow computers and reduction machines, respectively. The listed advantages and disadvantages of the dataflow and reduction machine models are based on research findings rather than on extensive operational experience.

Even though conventional von Neumann model has many disadvantages, the industry is still building computers following the control-flow model. The choice was based on cost-effectiveness, marketability, and the narrow windows of competition used by the industry. Program flow mechanisms dictate architectural choices. Both dataflow and reduction models, despite a higher potential for parallelism, are still concepts in the research stage. Control-flow machines still dominate the market.

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	

(Courtesy of Wah, Lowrie, and Li; reprinted with permission from *Computers for Artificial Intelligence Processing* edited by Wah and Ramamoorthy, Wiley and Sons, Inc., 1990)

In this book, we study mostly control-flow parallel computers. But dataflow and multithreaded architectures will be further studied in Chapter 9. Dataflow or hybrid von Neumann and dataflow machines offer design alternatives; *stream processing* (see Chapter 13) can be considered an example.

As far as innovative computer architecture is concerned the dataflow or hybrid models cannot be ignored. Both the Electrotechnical Laboratory (ETL) in Japan and the Massachusetts Institute of Technology have paid attention to these approaches. The book edited by Gaudiot and Bic (1991) provides details of some development on dataflow computers in that period.

2.4 SYSTEM INTERCONNECT ARCHITECTURES

Static and dynamic networks for interconnecting computer subsystems or for constructing multiprocessors or multicomputers are introduced below. We study first the distinctions between direct networks for static connections and indirect networks for dynamic connections. These networks can be used for internal connections among processors, memory modules, and I/O adaptors in a centralized system, or for distributed networking of multicomputer nodes.

Various topologies for building networks are specified below. Then we focus on the communication properties of interconnection networks. These include latency analysis, bisection bandwidth, and data-routing functions. Finally, we analyze the scalability of parallel architecture in solving scaled problems.

The communication efficiency of the underlying network is critical to the performance of a parallel computer. What we hope to achieve is a low-latency network with a high data transfer rate and thus a wide communication bandwidth. These network properties help make design choices for machine architecture.

2.4.1 Network Properties and Routing

The topology of an interconnection network can be either static or dynamic. *Static networks* are formed of point-to-point direct connections which will not change during program execution. *Dynamic networks* are implemented with switched channels, which are dynamically configured to match the communication demand in user programs. Packet switching and routing is playing an important role in modern multiprocessor architecture, which is discussed in Chapter 13; the basic concepts are discussed in Chapter 7.

Static networks are used for fixed connections among subsystems of a centralized system or multiple computing nodes of a distributed system. Dynamic networks include buses, crossbar switches, multistage networks, and routers which are often used in shared-memory multiprocessors. Both types of networks have also been implemented for inter-PE data routing in SIMD computers.

Before we discuss various network topologies, let us define several parameters often used to estimate the complexity, communication efficiency, and cost of a network. In general, a network is represented by the graph of a finite number of nodes linked by directed or undirected edges. The number of nodes in the graph is called the *network size.*

Node Degree and Network Diameter The number of edges (links or channels) incident on a node is called the *node degree d.* In the case of unidirectional channels, the number of channels into a node is the *in degree*, and that out of a node is the *out degree*. Then the node degree is the sum of the two. The node degree reflects the number of I/O ports required per node, and thus the cost of a node. Therefore, the node degree should be kept a (small) constant, in order to reduce cost. A constant node degree helps to achieve modularity in building blocks for scalable systems.

The *diameter D* of a network is the maximum shortest path between any two nodes. The path length is measured by the number of links traversed. The network diameter indicates the maximum number of distinct hops between any two nodes, thus providing a figure of communication merit for the network. Therefore, the network diameter should be as small as possible from a communication point of view.

Bisection Width When a given network is cut into two equal halves, the minimum number of edges (channels) along the cut is called the *channel bisection width b.* In the case of a communication network, each edge may correspond to a *channel* with w bit wires. Then the *wire bisection width* is $B = bw$. This parameter B reflects the wiring density of a network. When B is fixed, the *channel width* (in bits) $w = B/b$. Thus the bisection width provides a good indicator of the maximum communication bandwidth along the bisection of a network.

Another quantitative parameter is the *wire length* (or channel length) between nodes. This may affect the signal latency, clock skewing, or power requirements. We label a network *symmetric* if the topology is the same looking from any node. Symmetric networks are easier to implement or to program. Whether the nodes are homogeneous, the channels are buffered, or some of the nodes are switches, are some other useful properties for characterizing the structure of a network.

Data-Routing Functions A data-routing network is used for inter-PE data exchange. This routing network can be static, such as the hypercube routing network used in the TMC/CM-2, or dynamic such as the multistage network used in the IBM GF11. In the case of a multicomputer network, the data routing is achieved through message passing. Hardware routers are used to route messages among multiple computer nodes.

We specify below some primitive data-routing functions implementable on an inter-PE routing network. The versatility of a routing network will reduce the time needed for data exchange and thus can significantly improve the system performance.

Commonly seen data-routing functions among the PEs include *shifting, rotation, permutation* (one-to-one), *broadcast* (one-to-all), *multicast* (one-to-many), *shuffle, exchange,* etc. These routing functions can be implemented on ring, mesh, hypercube, or multistage networks.

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 composition operation. One can use cycle notation to specify a permutation function.

For example, the permutation $\pi = (a, b, c)\ (d, e)$ stands for the bijection mapping: $a \to b$, $b \to c$, $c \to a$, $d \to e$, and $e \to d$ in a circular fashion. The cycle (a, b, c) has a period of 3, and the cycle (d, e) a period of 2. Combining the two cycles, the permutation π has a period of $2 \times 3 = 6$. If one applies the permutation π six times, the identity mapping $I = (a), (b), (c), (d), (e)$ is obtained.

One can use a crossbar switch to implement the permutation in connecting n PEs among themselves. Multistage networks can implement some of the permutations in one or multiple passes through the network. Permutations can also be implemented with shifting or broadcast operations. The permutation capability of a network is often used to indicate the data routing capability. When n is large, the permutation speed often dominates the performance of a data routing network.

Perfect Shuffle and Exchange Perfect shuffle is a special permutation function suggested by Harold Stone (1971) for parallel processing applications. The mapping corresponding to a perfect shuffle is shown in Fig. 2.14a. Its inverse is shown on the right-hand side (Fig. 2.14b).

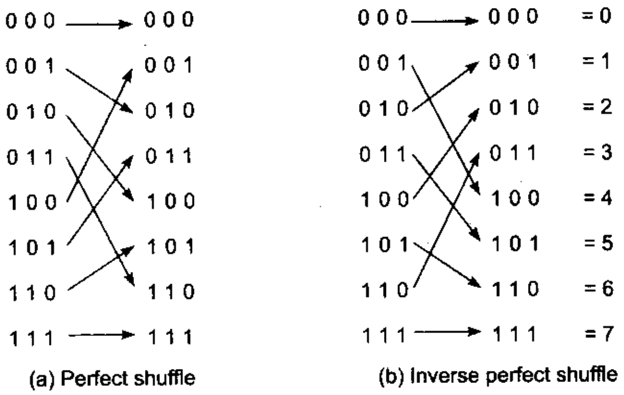
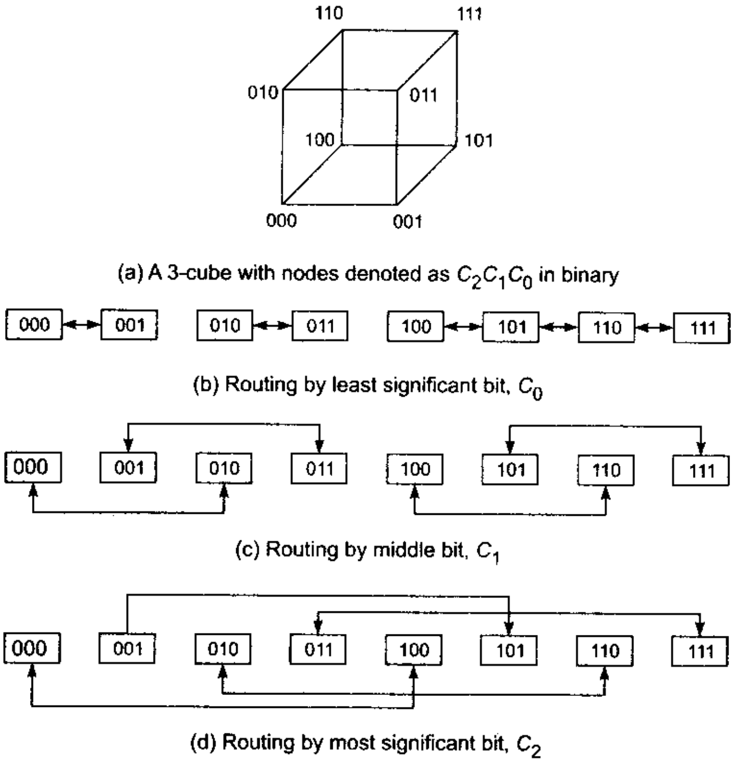


Fig. 2.14 Perfect shuffle and its inverse mapping over eight objects (Courtesy of H. Stone; reprinted with permission from *IEEE Trans. Computers*, 1971)

In general, to shuffle $n = 2^k$ objects evenly, one can express each object in the domain by a k- bit binary number $x = (x_{k-1}, \ldots, x_1, x_0)$. The perfect shuffle maps x to y, where $y = (x_{k-2}, \ldots, x_1, x_0, x_{k-1})$ is obtained from x by shifting 1 bit to the left and wrapping around the most significant to the least significant position.

Hypercube Routing Functions A three-dimensional binary cube network is shown in Fig. 2.15. Three routing functions are defined by three bits in the node address. For example, one can exchange the data between adjacent nodes which differ in the least significant bit C_0, as shown in Fig. 2.15b.

Similarly, two other routing patterns can be obtained by checking the middle bit C_1 (Fig. 2.15c) and the most significant bit C_2 (Fig. 2.15d), respectively. In general, an n-dimensional hypercube has n routing functions, defined by each bit of the n-bit address. These data exchange functions can be used in routing messages in a hypercube multicomputer.



(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* is a one-to-all mapping. This can be easily achieved in an SIMD computer using a broadcast bus extending from the array controller to all PEs. A message-passing multicomputer also has mechanisms to broadcast messages. *Multicast* corresponds to a mapping from one PE to other PEs (one to many).

Broadcast is often treated as a global operation in a multicomputer. Multicast has to be implemented with matching of destination codes in the network.

Network Performance To summarize the above discussions, the performance of an interconnection network is affected by the following factors:

(1) *Functionality*—This refers to how the network supports data routing, interrupt handling, synchronization, request/message combining, and coherence.

(2) *Network latency*—This refers to the worst-case time delay for a unit message to be transferred through the network.

(3) *Bandwidth*—This refers to the maximum data transfer rate, in terms of Mbytes/s or Gbytes/s, transmitted through the network.

(4) *Hardware complexity*—This refers to implementation costs such as those for wires, switches, connectors, arbitration, and interface logic.

(5) *Scalability*—This refers to the ability of a network to be modularly expandable with a scalable performance with increasing machine resources.

2.4.2 Static Connection 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 describe their topologies below in terms of network parameters and comment on their relative merits in relation to communication and scalability.

Linear Array This is a one-dimensional network in which N nodes are connected by $N - 1$ links in a line (Fig. 2.16a). Internal nodes have degree 2, and the terminal nodes have degree 1. The diameter is $N - 1$, which is rather long for large N. The bisection width $b = 1$. *Linear arrays* are the simplest connection topology. The structure is not symmetric and poses a communication inefficiency when N becomes very large.

For $N = 2$, it is clearly simple and economic to implement a linear array. As the diameter increases linearly with respect to N, it should not be used for large N. It should be noted that a linear array is very different from a *bus* which is time-shared through switching among the many nodes attached to it. A linear array allows concurrent use of different sections (channels) of the structure by different source and destination pairs.

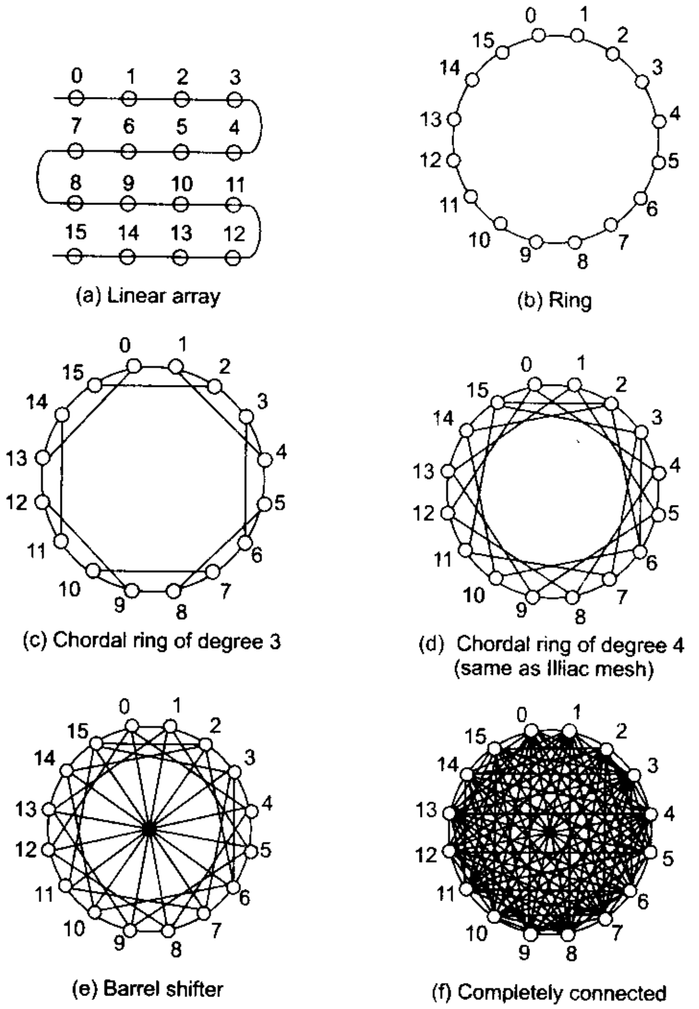
Ring and Chordal Ring A *ring* is obtained by connecting the two terminal nodes of a linear array with one extra link (Fig. 2.16b). A ring can be unidirectional or bidirectional. It is symmetric with a constant node degree of 2. The diameter is $\lfloor N/2 \rfloor$ for a bidirectional ring, and N for unidirectional ring.

The IBM *token ring* had this topology, in which messages circulate along the ring until they reach the destination with a matching ID. Pipelined or packet-switched rings have been implemented in the CDC Cyberplus multiprocessor (1985) and in the KSR-1 computer system (1992) for interprocessor communications.

By increasing the node degree from 2 to 3 or 4, we obtain two *chordal rings* as shown in Figs. 2.16c and 2.16d, respectively. One and two extra links are added to produce the two chordal rings, respectively. In general, the more links added, the higher the node degree and the shorter the network diameter.

Comparing the 16-node ring (Fig. 2.16b) with the two chordal rings (Figs. 2.16c and 2.16d), the network diameter drops from 8 to 5 and to 3, respectively. In the extreme, the *completely connected network* in Fig. 2.16f has a node degree of 15 with the shortest possible diameter of 1.

Barrel Shifter As shown in Fig. 2.16e for a network of $N = 16$ nodes, the *barrel shifter* is obtained from the ring by adding extra links from each node to those nodes having a distance equal to an integer power of 2. This implies that node i is connected to node j if $|j - i| = 2^r$ for some $r = 0, 1, 2, \ldots, n - 1$ and the network size is $N = 2^n$. Such a barrel shifter has a node degree of $d = 2n - 1$ and a diameter $D = n/2$.



(a) Linear array

(b) Ring

(c) Chordal ring of degree 3

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

(e) Barrel shifter

(f) Completely connected

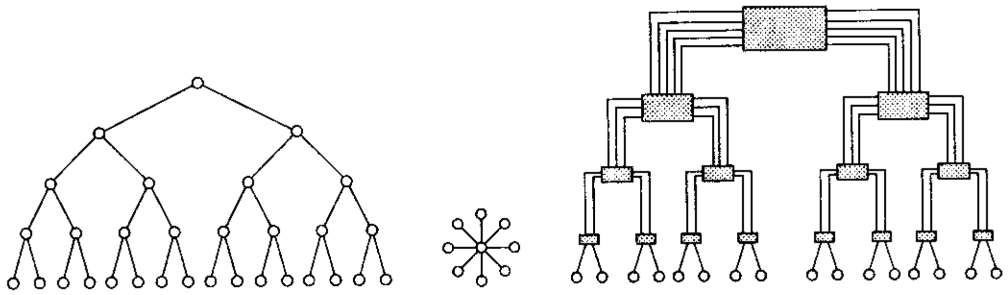
Fig. 2.16 Linear array, ring, chordal rings of degrees 3 and 4, barrel shifter, and completely connected network

Obviously, the connectivity in the barrel shifter is increased over that of any chordal ring of lower node degree. For $N = 16$, the barrel shifter has a node degree of 7 with a diameter of 2. But the barrel shifter complexity is still much lower than that of the completely connected network (Fig. 2.16f).

Tree and Star A *binary tree* of 31 nodes in five levels is shown in Fig. 2.17a. In general, a k-level, completely balanced binary tree should have $N = 2^k - 1$ nodes. The maximum node degree is 3 and the diameter is $2(k - 1)$. With a constant node degree, the binary tree is a scalable architecture. However, the diameter is rather long.

The *star* is a two-level tree with a high node degree at the central node of $d = N - 1$ (Fig. 2.17b) and a small constant diameter of 2. A DADO multiprocessor was built at Columbia University (1987) with a 10-level binary tree of 1023 nodes. The star architecture has been used in systems with a centralized supervisor node.

Fat Tree The conventional tree structure used in computer science can be modified to become the *fat tree*, as introduced by Leiserson in 1985. A binary fat tree is shown in Fig. 2.17c. The channel width of a fat tree increases as we ascend from leaves to the root. The fat tree is more like a real tree in that branches get thicker toward the root.



(a) Binary tree (b) Star (c) Binary fat tree

Fig. 2.17 Tree, star, and fat tree

One of the major problems in using the conventional binary tree is the bottleneck problem toward the root, since the traffic toward the root becomes heavier. The fat tree has been proposed to alleviate the problem. The idea of a fat tree was applied in the Connection Machine CM-5, to be studied in Chapter 8. The idea of binary fat trees can also be extended to multiway fat trees.

Mesh and Torus A 3 × 3 example mesh network is shown in Fig. 2.18a. The mesh is a frequently used architecture which has been implemented in the Illiac IV, MPP, DAP, and Intel Paragon with variations.

In general, a k-dimensional mesh with $N = n^k$ nodes has an interior node degree of $2k$ and the network diameter is $k(n - 1)$. Note that the pure mesh as shown in Fig. 2.18a is not symmetric. The node degrees at the boundary and corner nodes are 3 or 2.

Figure 2.18b shows a variation of the mesh by allowing wraparound connections. The Illiac IV assumed an 8×8 mesh with a constant node degree of 4 and a diameter of 7. The Illiac mesh is topologically equivalent to a chordal ring of degree 4 as shown in Fig. 2.16d for an $N = 9 = 3 \times 3$ configuration.

In general, an $n \times n$ Illiac mesh should have a diameter of $d = n - 1$, which is only half of the diameter for a pure mesh. The *torus* shown in Fig. 2.18c can be viewed as another variant of the mesh with an even shorter diameter. This topology combines the ring and mesh and extends to higher dimensions.

The torus has ring connections along each row and along each column of the array. In general, an $n \times n$ binary torus has a node degree of 4 and a diameter of $2\lfloor n/2 \rfloor$. The torus is a symmetric topology. All added wraparound connections help reduce the diameter by one-half from that of the mesh.

Systolic Arrays This is a class of multidimensional pipelined array architectures designed for implementing fixed algorithms. What is shown in Fig. 2.18d is a systolic array specially designed for performing matrix multiplication. The interior node degree is 6 in this example.

In general, static systolic arrays are pipelined with multidirectional flow of data streams. The commercial machine Intel iWarp system (Anaratone et al., 1986) was designed with a systolic architecture. The systolic array has become a popular research area ever since its introduction by Kung and Leiserson in 1978.

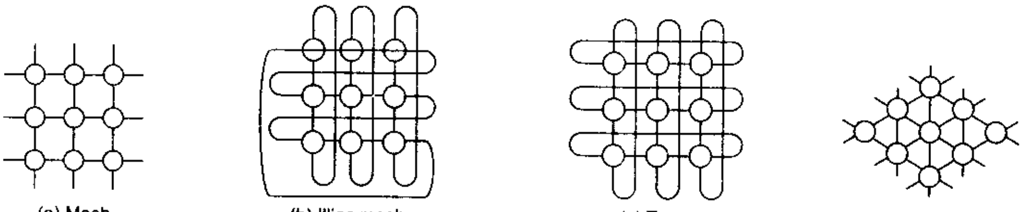


Fig. 2.18 Mesh, Illiac mesh, torus, and systolic array

With fixed interconnection and synchronous operation, a systolic array matches the communication structure of the algorithm. For special applications like signal/image processing, systolic arrays may offer a better performance/cost ratio. However, the structure has limited applicability and can be very difficult to program. Since this book emphasizes general-purpose computing, we will not study systolic arrays further. Interested readers may refer to the book by S.Y. Kung (1988) for using systolic and wavefront architectures in building VLSI array processors.

Hypercubes This is a binary n-cube architecture which has been implemented in the iPSC, nCUBE, and CM-2 systems. In general, an n-cube consists of $N = 2^n$ nodes spanning along n dimensions, with two nodes per dimension. A 3-cube with 8 nodes is shown in Fig. 2.19a.

A 4-cube can be formed by interconnecting the corresponding nodes of two 3 cubes, as illustrated in Fig. 2.19b. The node degree of an n-cube equals n and so does the network diameter. In fact, the node degree increases linearly with respect to the dimension, making it difficult to consider the hypercube a scalable architecture.

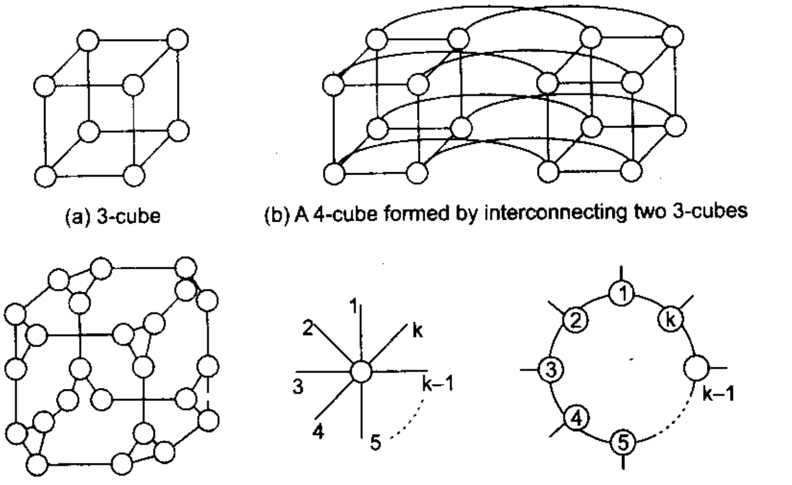
Binary hypercube has been a very popular architecture for research and development in the 1980s. Both Intel iPSC/1, iPSC/2, and nCUBE machines were built with the hypercube architecture. The architecture has dense connections. Many other architectures, such as binary trees, meshes, etc., can be embedded in the hypercube.

With poor scalability and difficulty in packaging higher-dimensional hypercubes, the hypercube architecture was gradually being replaced by other architectures. For example, the CM-5 employed the fat tree over the hypercube implemented in the CM-2. The Intel Paragon employed a two-dimensional mesh over its hypercube predecessors. Topological equivalence has been established among a number of network architectures. The bottom line for an architecture to survive in future systems is packaging efficiency and scalability to allow modular growth.

Cube-Connected Cycles This architecture is modified from the hypercube. As illustrated in Fig. 2.19c, a 3-cube is modified to form *3-cube-connected cycles* (CCC). The idea is to cut off the corner nodes (vertices) of the 3-cube and replace each by a ring (cycle) of 3 nodes.

In general, one can construct *k-cube-connected cycles* from a *k-cube* with $n = 2^k$ cycles nodes as illustrated in Fig. 2.19d. The idea is to replace each vertex of the k dimensional hypercube by a ring of k nodes. A k-cube can be thus transformed to a k-CCC with $k \times 2^k$ nodes.

The 3-CCC shown in Fig. 2.19b has a diameter of 6, twice that of the original 3-cube. In general, the network diameter of a k-CCC equals $2k$. The major improvement of a CCC lies in its constant node degree of 3, which is independent of the dimension of the underlying hypercube.



(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

Fig. 2.19 Hypercubes and cube-connected cycles

Consider a hypercube with $N = 2^n$ nodes. A CCC with an equal number of N nodes must be built from a lower-dimension k-cube such that $2^n = k \cdot 2^k$ for some $k < n$.

For example, a 64-node CCC can be formed by replacing the corner nodes of a 4-cube with cycles of four nodes, corresponding to the case $n = 6$ and $k = 4$. The CCC has a diameter of $2k = 8$, longer than 6 in a 6-cube. But the CCC has a node degree of 3, smaller than the node degree of 6 in a 6-cube. In this sense, the CCC is a better architecture for building scalable systems if latency can be tolerated in some way.

k-ary n-Cube Networks Rings, meshes, tori, binary n-cubes (hypercubes), and Omega networks are topologically isomorphic to a family of *k-ary n-cube* networks. Figure 2.20 shows a 4-ary 3-cube network.

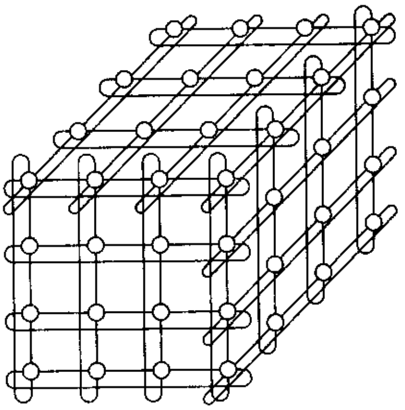


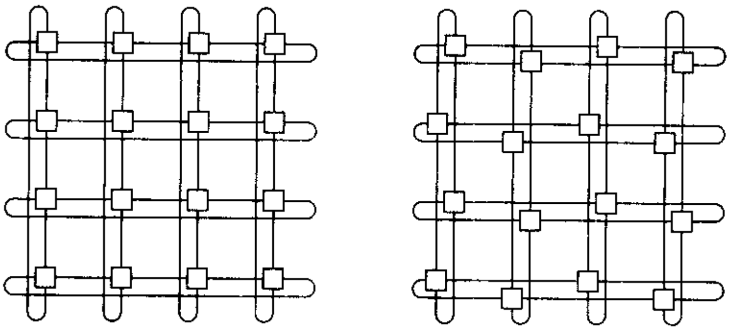
Fig. 2.20 The k-ary n-cube network shown with $k = 4$ and $n = 3$; hidden nodes or connections are not shown

The parameter n is the dimension of the cube and k is the *radix*, or the number of nodes (multiplicity) along each dimension. These two numbers are related to the number of nodes, N, in the network by:

$$N = k^n, (k = \sqrt[n]{N}, n = \log_k N) \tag{2.3}$$

A node in the k-ary n-cube can be identified by an n-digit radix-k address $A = a_1 a_2 \ldots a_n$, where a_i represents the node's position in the ith dimension. For simplicity, all links are assumed bidirectional. Each line in the network represents two communication channels, one in each direction. In Fig. 2.20, the lines between nodes are bidirectional links.

Traditionally, low-dimensional k-ary n-cubes are called *tori*, and high-dimensional binary n-cubes are called *hypercubes*. The long end-around connections in a torus can be avoided by folding the network as shown in Fig. 2.21. In this case, all links along the ring in each dimension have equal wire length when the multidimensional network is embedded in a plane.



(a) Traditional torus (a 4-ary 2-cube)

(b) A torus with folded connections

Fig. 2.21 Folded connections to equalize the wire length in a torus network (Courtesy of W. Dally; reprinted with permission from *IEEE Trans. Computers*, June 1990)

William Dally (1990) has revealed a number of interesting properties of k-ary n cube networks. The cost of such a network is dominated by the amount of wire, rather by the number of switches required. Under the assumption of constant wire bisection, low-dimensional networks with wide channels provide lower latency, less contention, and higher hot-spot throughput than higher-dimensional networks with narrow channels.

Network Throughput The *network throughput* is defined as the total number of messages the network can handle per unit time. One method of estimating throughput is to calculate the capacity of a network, the total number of messages that can be in the network at once. Typically, the maximum throughput of a network 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. Hot-spot traffic can degrade performance of the entire network by causing congestion. The *hot-spot throughput* of a network is the maximum rate at which mèssages can be sent from one specific node P_i to another specific node P_j.

Low-dimensional networks operate better under nonuniform loads because they allow better resource sharing. In a high-dimensional network, wires are assigned to particular dimensions and cannot be shared between dimensions. For example, in a binary n-cube, it is possible for a wire to be saturated while a physically adjacent wire assigned to a different dimension remains idle. In a torus, all physically adjacent wires are combined into a single channel which is shared by all messages.

As a rule of thumb, minimum network latency is achieved when the network radix k and dimension n are chosen to make the components of communication latency due to distance D (the number of hops between nodes) and the message aspect ratio L/W (message length L normalized to the channel width W) approximately equal.

Low-dimensional networks reduce contention because having a few high-bandwidth channels results in more resource sharing and thus a better queueing performance than having many low-bandwidth channels. While network capacity and worst-case blocking latency are independent of dimension, low-dimensional networks have a higher maximum throughput and lower average block latency than do high-dimensional networks.

Both fat tree networks and k-ary n-cube networks are considered universal in the sense that they can efficiently simulate any other network of the same volume. Dally claimed that any point-to-point network can be embedded in a 3-D mesh with no more than a constant increase in wiring length.

Summary of Static Networks In Table 2.2, we summarize the important characteristics of static connection networks. The node degrees of most networks are less than 4, which is rather desirable. For example, the INMOS Transputer chip was a compute communication microprocessor with four ports for communication. See also the TILE64 system-on-a-chip described in Chapter 13.

Table 2.2 *Summary of Static Network Characteristics*

Network type	*Node degree, d*	*Network diameter,*	*No. of links, l*	*Bisection width, B*	*Symmetry*	*Remarks on network size*
Linear Array	2	$N-1$	$N-1$	1	No	N nodes
Ring	2	$\lfloor N/2 \rfloor$	N	2	Yes	N nodes
Completely Connected	$N-1$	1	$N(N-1)/2$	$(N/2)^2$	Yes	N nodes
Binary Tree	3	$2(h-1)$	$N-1$	1	No	Tree height $h = \lceil \log_2 N \rceil$
Star	$N-1$	2	$N-1$	$\lfloor N/2 \rfloor$	No	N nodes
2D-Mesh	4	$2(r-1)$	$2N-2r$	r	No	$r \times r$ mesh where $r = \sqrt{N}$
Illiac Mesh	4	$r-1$	$2N$	$2r$	No	Equivalent to a chordal ring of $r = \sqrt{N}$
2D-Torus	4	$2\lfloor r/2 \rfloor$	$2N$	$2r$	Yes	$r \times r$ torus where $r = \sqrt{N}$
Hypercube	n	n	$nN/2$	$N/2$	Yes	N nodes, $n = \log_2 N$ (dimension)
CCC	3	$2k-1+\lfloor k/2 \rfloor$	$3N/2$	$N/(2k)$	Yes	$N = k \times 2^k$ nodes with a cycle length $k \geq 3$
k-ary n-cube	$2n$	$n\lfloor k/2 \rfloor$	nN	$2k^{n-1}$	Yes	$N = k^n$ nodes

With a constant node degree of 4, a Transputer (such as the T800) becomes applicable as a building block. The node degrees for the completely connected and star networks are both bad. The hypercube node degree increases with $\log_2 N$ and is also bad when the value of N becomes large.

Network diameters vary over a wide range. With the invention of hardware routing (wormhole routing), the diameter has become less critical an issue because the communication delay between any two nodes becomes almost a constant with a high degree of pipelining. The number of links affects the network cost. The bisection width affects the network bandwidth.

The property of symmetry affects scalability and routing efficiency. It is fair to say that the total network cost increases with d and l. A smaller diameter is still a virtue. But the average distance between nodes may be a better measure. The bisection width can be enhanced by a wider channel width. Based on the above analysis, ring, mesh, torus, k-ary n-cube, and CCC all have some desirable features for building MPP systems.

2.4.3 Dynamic Connection Networks

For multipurpose or general-purpose applications, we may need to use dynamic connections which can implement all communication patterns based on program demands. Instead of using fixed connections, switches or arbiters must be used along the connecting paths to provide the dynamic connectivity. In increasing order of cost and performance, dynamic connection networks include *bus systems, multistage interconnection networks* (MIN), and *crossbar switch networks.*

The price tags of these networks are attributed to the cost of the wires, switches, arbiters, and connectors required. The performance is indicated by the network bandwidth, data transfer rate, network latency, and communication patterns supported. A brief introduction to dynamic connection networks is given below. Details can be found in subsequent chapters.

Digital Buses A *bus system* is essentially a collection of wires and connectors for data transactions among processors, memory modules, and peripheral devices attached to the bus. The bus is used for only one transaction at.a time between source and destination. In case of multiple requests, the bus arbitration logic must be able to allocate or deallocate the bus, servicing the requests one at a time.

For this reason, the digital bus has been called *contention bus* or a *time-sharing bus* among multiple functional modules. A bus system has a lower cost and provides a limited bandwidth compared to the other two dynamic connection networks. Many industrial and IEEE bus standards are available.

Figure 2.22 shows a bus-connected multiprocessor system. The system bus provides a common communication path between the processors, I/O subsystem, and the memory modules, secondary storage devices, network adaptors, etc. The system bus is often implemented on a backplane of a printed circuit board. Other boards for processors, memories, or device interfaces are plugged into the backplane board via connectors or cables.

The active or master devices (processors or I/O subsystem) generate requests to address the memory. The passive or slave devices (memories or peripherals) respond to the requests. The common bus is used on a time-sharing basis, and important busing issues include the bus arbitration, interrupts handling, coherence protocols, and transaction processing. We will study typical bus systems, such as the VME bus and others, in Chapter 5. Hierarchical bus structures for building larger multiprocessor systems are studied in Chapter 7.

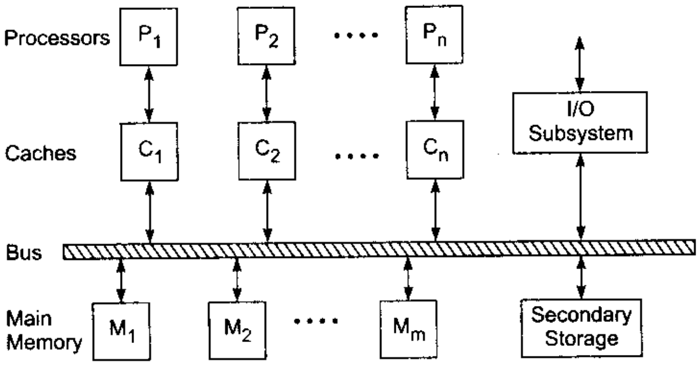


Fig. 2.22 A bus-connected multiprocessor system, such as the Sequent Symmetry S1

Switch Modules An $a \times b$ *switch module* has a inputs and b outputs. A *binary switch* corresponds to a 2×2 switch module in which $a = b = 2$. In theory, a and b do not have to be equal. However, in practice, a and b are often chosen as integer powers of 2; that is, $a = b = 2^k$ for some $k \geq 1$.

Table 2.3 lists several commonly used switch module sizes: 2×2, 4×4, and 8×8. Each input can be connected to one or more of the outputs. However, conflicts must be avoided at the output terminals. In other words, one-to-one and one-to-many mappings are allowed; but many-to-one mappings are not allowed due to conflicts at the output terminal.

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	$n!$

When only one-to-one mappings (permutations) are allowed, we call the module an $n \times n$ crossbar switch. For example, a 2×2 crossbar switch can connect two possible patterns: *straight* or *crossover*. In general, an $n \times n$ crossbar can achieve $n!$ permutations. The numbers of legitimate connection patterns for switch modules of various sizes are listed in Table 2.3.

Multistage Interconnection Networks MINs have been used in both MIMD and SIMD computers. A generalized multistage network is illustrated in Fig. 2.23. A number of $a \times b$ switches are used in each stage. Fixed interstage connections are used between the switches in adjacent stages. The switches can be dynamically set to establish the desired connections between the inputs and outputs.

Different classes of MINs differ in the switch modules used and in the kind of *interstage connection* (ISC) patterns used. The simplest switch module would be the 2×2 switches ($a = b = 2$ in Fig. 2.23). The ISC patterns often used include *perfect shuffle, butterfly, multiway shuffle, crossbar, cube connection,* etc. Some of these ISC patterns are shown below with examples.

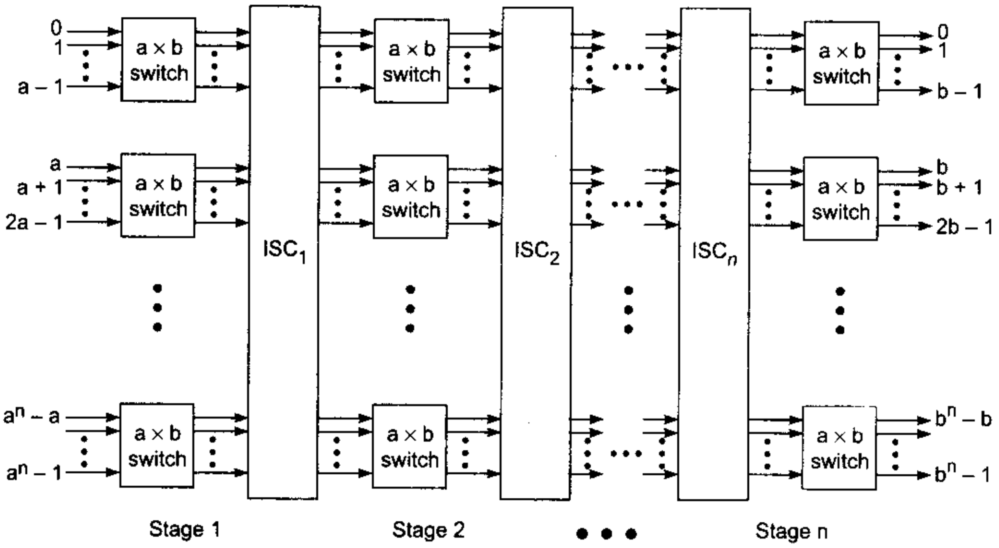


Fig. 2.23 A generalized structure of a multistage interconnection network (MIN) built with a × b switch modules and interstage connection patterns $ISC_1, ISC_2, \ldots, ISC_n$

Omega Network Figures 2.24a to 2.24d show four possible connections of 2 × 2 switches used in constructing the Omega network. A 16 × 16 Omega network is shown in Fig. 2.24e. Four stages of 2 × 2 switches are needed. There are 16 inputs on the left and 16 outputs on the right. The ISC pattern is the perfect shuffle over 16 objects.

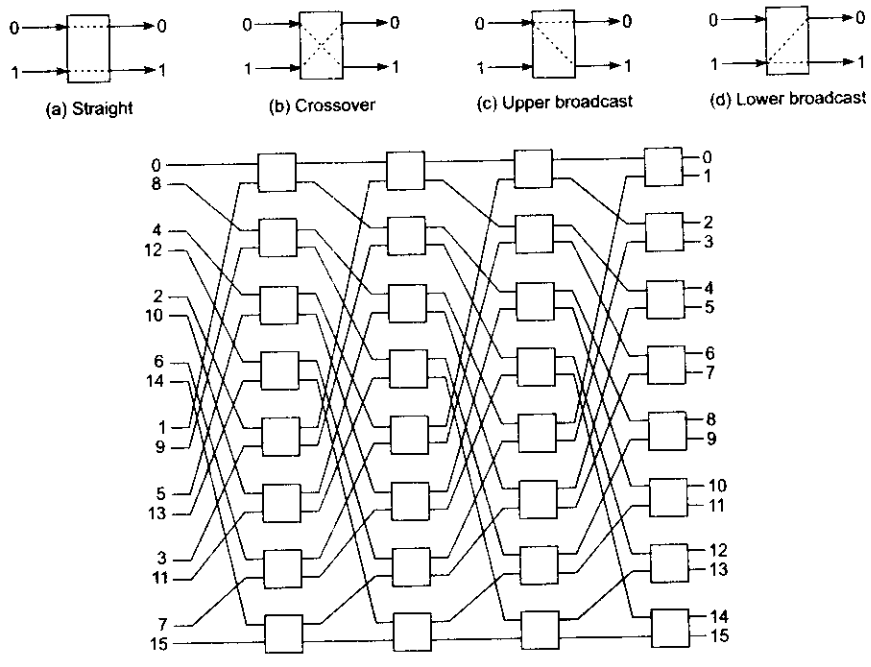
In general, an n-input Omega network requires $\log_2 n$ stages of 2 × 2 switches. Each stage requires $n/2$ switch modules. In total, the network uses $n \log_2 n/2$ switches. Each switch module is individually controlled.

Various combinations of the switch states implement different permutations, broadcast, or other connections from the inputs to the outputs. The interconnection capabilities of the Omega and other networks will be further studied in Chapter 7.

Baseline Network Wu and Feng (1980) have studied the relationship among a class of multistage interconnection networks. A *Baseline network* can be generated recursively as shown in Fig. 2.25a.

The first stage contains one $N \times N$ block, and the second stage contains two $(N/2) \times (N/2)$ subblocks, labeled C_0 and C_1. The construction process can be recursively applied to the subblocks until the $N/2$ subblocks of size 2 × 2 are reached.

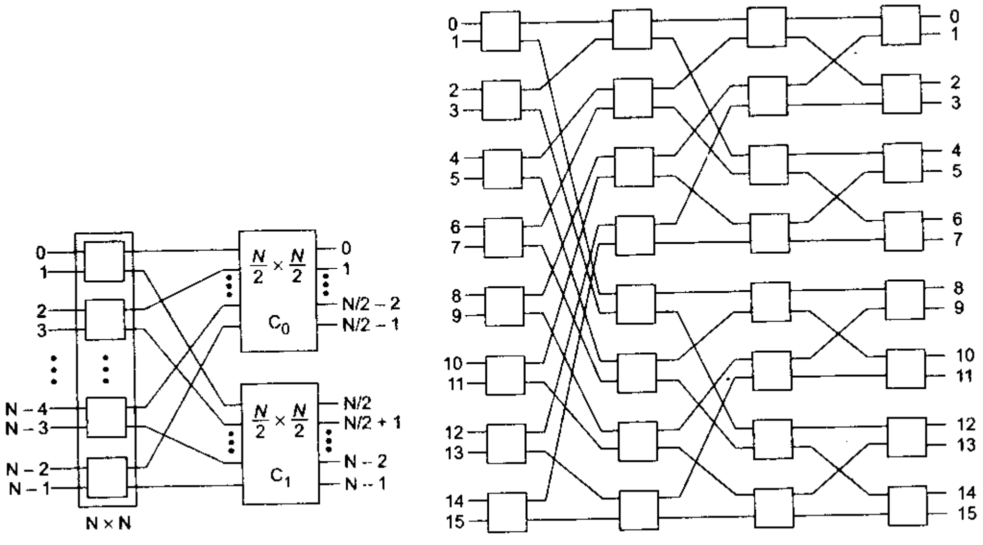
The small boxes and the ultimate building blocks of the subblocks are the 2 × 2 switches, each with two legitimate connection states: *straight* and *crossover* between the two inputs and two outputs. A 16 × 16 Baseline network is shown in Fig. 2.25b. In Problem 2.15, readers are asked to prove the topological equivalence between the Baseline and other networks.



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

(e) 16 × 16 Omega network

Fig. 2.24 The use of 2 × 2 switches and perfect shuffle as an interstage connection pattern to construct a 16 × 16 Omega network (Courtesy of Duncan Lawrie; reprinted with permission from *IEEE Trans. Computers*, Dec. 1975)

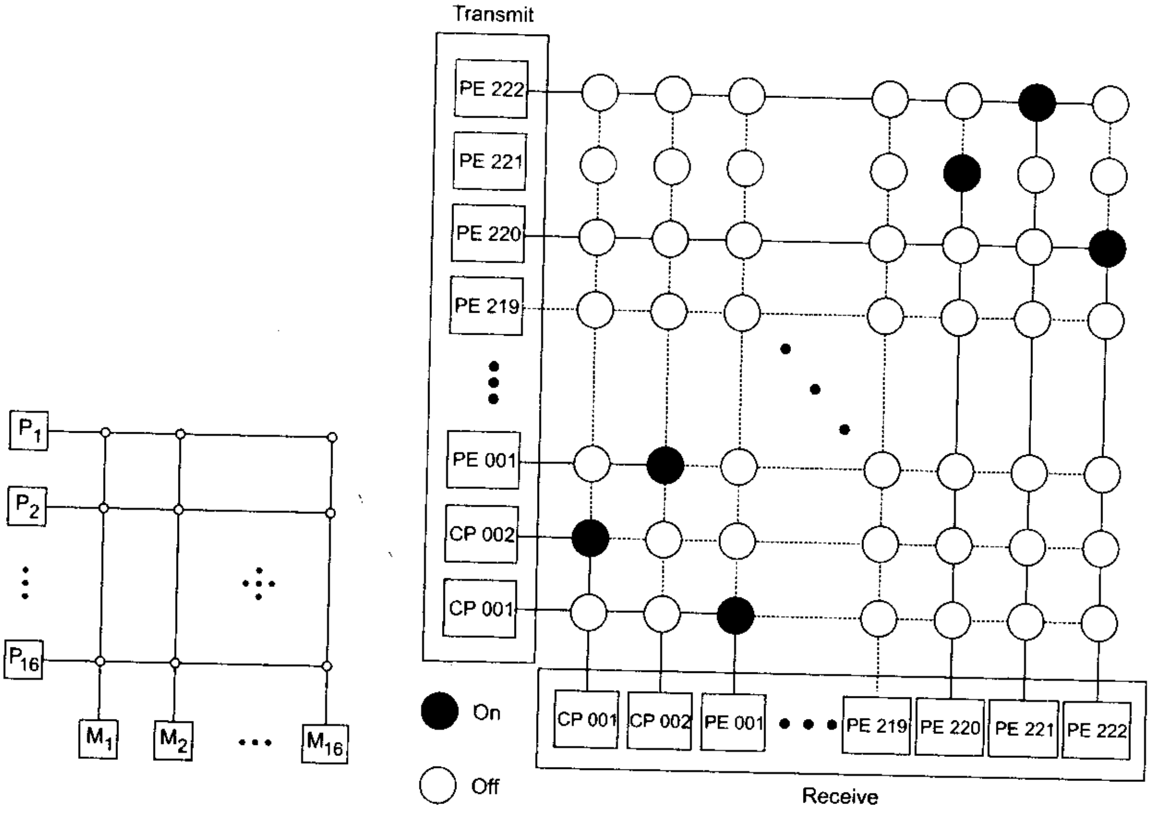


(a) Recursive construction (b) A 16 × 16 Baseline network

Fig. 2.25 Recursive construction of a Baseline network (Courtesy of Wu and Feng; reprinted with permission from *IEEE Trans. Computers, August 1980*)

Crossbar Network The highest bandwidth and interconnection capability are provided by crossbar networks. A crossbar network can be visualized as a single-stage switch network. Like a telephone switchboard, the crosspoint switches provide dynamic connections between source, destination pairs. Each crosspoint switch can provide a dedicated connection path between a pair. The switch can be set on or off dynamically upon program demand. Two types of crossbar networks are illustrated in Fig. 2.26.

To build a shared-memory multiprocessor, one can use a crossbar network between the processors and memory modules (Fig. 2.26a). This is essentially a memory-access network. The pioneering C.mmp multiprocessor (Wulf and Bell, 1972) implemented a 16 × 16 crossbar network which connected 16 PDP 11 processors to 16 memory modules, each of which had a capability of 1 million words of memory cells. The 16 memory modules could be accessed by the processors in parallel.



(a) Interprocessor-memory crossbar network built in the C.mmp multiprocessor at Carnegie-Mellon University (1972)

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

Fig. 2.26 Two crossbar switch network configurations

Note that each memory module can satisfy only one processor request at a time. When multiple requests arrive at the same memory module simultaneously, the crossbar must resolve the conflicts. The behavior of each crossbar switch is very similar to that of a bus. However, each processor can generate a sequence of addresses to access multiple memory modules simultaneously. Thus, in Fig. 2.26a, only one crosspoint switch can be set on in each column. However, several crosspoint switches can be set on simultaneously in order to support parallel (or interleaved) memory accesses.

Another type of crossbar network is for interprocessor communication and is depicted in Fig. 2.26b. This large crossbar (224 × 224) was actually built in a vector parallel processor (VPP500) by Fujitsu Inc. (1992). The PEs are processors with attached memory. The CPs stand for control processors which are used to supervise the entire system operation, including the crossbar networks. In this crossbar, at one time only one crosspoint switch can be set on in each row and each column.

The interprocessor crossbar provides permutation connections among the processors. Only one-to-one connections are provided. Therefore, the $n \times n$ crossbar connects at most n source, destination pairs at a time. We will further study crossbar networks in Chapters 7 and 8.

Summary In Table 2.4, we summarize the important features of buses, multistage networks, and crossbar switches in building dynamic networks. Obviously. the bus is the cheapest to build, but its drawback lies in the low bandwidth available to each processor.

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.

Another problem with the bus is that it is prone to failure. Some fault-tolerant systems, like the Tandem multiprocessor for transaction processing, used dual buses to protect the system from single failures.

The crossbar switch is the most expensive one to build, due to the fact that its hardware complexity increases as n^2. However, the crossbar has the highest bandwidth and routing capability. For a small network size, it is the desired choice.

Multistage networks provide a compromise between the two extremes. The major advantage of MINs lies in their scalability with modular construction. However, the latency increases with log n, the number of stages in the network. Also, costs due to increased wiring and switching complexity are another constraint.

For building MPP systems, some of the static topologies are more scalable in specific applications. Advances in VLSI and interconnect technologies have had a major impact on multiprocessor system architecture, as we shall see in Chapter 13, and there has been a clear shift towards the use of packet-based switched-media interconnects.

Summary

In this chapter, we have focused on basic program properties which make parallelism possible and determine the amount and type of parallelism which can be exploited. With increasing degree of multiprocessing, the rate at which data must be communicated between subsystems also increases, and therefore the system interconnect architecture becomes important in determining system performance.

We started this chapter with a study of the basic conditions which must be satisfied for parallel computations to be possible. In essence, it is dependences between operations which limit the amount of parallelism which can be exploited. After all, any set of N fully independent operations can always be performed in parallel.

The three basic data dependences between operations are *flow dependence*, *anti-dependence* and *output dependence*. *Resource dependence* refers to a limitation in available hardware and/or software resources which limits the achievable degree of parallelism. Bernstein's conditions—which apply to input and output sets of processes—must be satisfied for parallel execution of processes to be possible.

Parallelism may be exploited at the level of software or hardware. For software parallelism, program design, and the program development and runtime environments play the key role. For hardware parallelism, availability of the right mix of hardware resources plays the key role. Program partitioning, grain size, communication latency and scheduling are important concepts; scheduling may be static or dynamic.

Program flow may be control-driven, data-driven or demand-driven. Of these, control-driven program flow, as exemplified in the von Neumann model, is the only one that has proved commercially successful over the last six decades. Other program flow models have been tried out on research-oriented systems, but in general these models have not found acceptance on a broader basis.

When computer systems consist of multiple processors—and several other sub-systems such as memory modules and network adapters—the system interconnect architecture plays a very important role in determining final system performance. We studied basic network properties, including topology and routing functionality. Network performance can be characterized in terms of bandwidth, latency, functionality and scalability.

We studied static network topologies such as the linear array, ring, tree, fat tree, torus and hypercube; we also looked at dynamic network topologies which involve switching and/or routing of data. With higher degree of multiprocessing, bus-based systems are unable to meet aggregate bandwidth requirements of the system; multistage inter-connection networks and crossbar switches can provide better alternatives.

Exercises

Problem 2.1 Define the following terms related to parallelism and dependence relations:

(a) Computational granularity.
(b) Communication latency.
(c) Flow dependence.
(d) Antidependence.
(e) Output dependence.
(f) I/O dependence.
(g) Control dependence.
(h) Resource dependence.
(i) Bernstein conditions.
(j) Degree of parallelism.

Problem 2.2 Define the following terms for various system interconnect architectures:

(a) Node degree.
(b) Network diameter.
(c) Bisection bandwidth.
(d) Static connection networks.
(e) Dynamic connection networks.
(f) Nonblocking networks.
(g) Multicast and broadcast.
(h) Mesh versus torus.
(i) Symmetry in networks.
(j) Multistage networks.
(k) Crossbar networks.
(l) Digital buses.

Problem 2.3 Answer the following questions on program flow mechanisms and computer models:

(a) Compare control-flow, dataflow, and reduction computers in terms of the program flow mechanism used.
(b) Comment on the advantages and disadvantages in control complexity, potential for parallelism, and cost-effectiveness of the above computer models.
(c) What are the differences between string reduction and graph reduction machines?

Problem 2.4 Perform a data dependence analysis on each of the following Fortran program fragments. Show the dependence graphs among the statements with justification.

```
(a)  S1:  A = B + D
     S2:  C = A × 3
     S3:  A = A + C
     S4:  E = A / 2

(b)  S1:  X = SIN(Y)
     S2:  Z = X + W
     S3:  Y = −2.5 × W
     S4:  X = COS(Z)
```

(c) Determine the data dependences in the same and adjacent iterations of the following Do-loop.

```
                Do  10 I = 1, N
     S1:            A(I + 1) = B(I − 1) + C(I)
     S2:            B(I) = A(I) × K
     S3:            C(I) = B(I) − 1
              10  Continue
```

Problem 2.5 Analyze the data dependences among the following statements in a given program:

```
S1:  Load R1, 1024        /R1 ← 1024/
S2:  Load R2, M(10)       /R2 ← Memory(10)/
S3:  Add R1, R2           /R1 ← (R1) + (R2)/
S4:  Store M(1024), R1    /Memory(1024) ← (R1)/
S5:  Store M((R2)), 1024  /Memory(64) ← 1024/
```

where (Ri) means the content of register Ri and Memory(10) contains 64 initially.

(a) Draw a dependence graph to show all the dependences.
(b) Are there any resource dependences if only

one copy of each functional unit is available in the CPU?

(c) Repeat the above for the following program statements:

```
S1:  Load R1, M(100)   /R1 ← Memory(100)/
S2:  Move R2, R1       /R2 ← (R1)/
S3:  Inc R1            /R1 ← (R1) + 1/
S4:  Add R2, R1        /R2 ← (R2) + (R1)/
S5:  Store M(100), R1  /Memory(100) ← (R1)/
```

Problem 2.6 A sequential program consists of the following five statements, S1 through S5. Considering each statement as a separate process, clearly identify *input set* I_i and *output set* O_i of each process. Restructure the program using Bernstein's conditions in order to achieve maximum parallelism between processes. If any pair of processes cannot be executed concurrently, specify which of the three conditions is not satisfied.

```
S1:  A = B + C
S2:  C = B × D
S3:  S = 0
S4:  Do I = A, 100
         S = S + X(I)
     End Do
S5:  IF (S .GT. 1000) C = C × 2
```

Problem 2.7 Consider the execution of the following code segment consisting of seven statements. Use Bernstein's conditions to detect the maximum parallelism embedded in this code. Justify the portions that can be executed in parallel and the remaining portions that must be executed sequentially. Rewrite the code using parallel constructs such as *Cobegin* and *Coend*. No variable substitution is allowed. All statements can be executed in parallel if they are declared within the same block of a (*Cobegin, Coend*) pair.

```
S1:  A = B + C
S2:  C = D + E
S3:  F = G + E
S4:  C = A + F
S5:  M = G + C
S6:  A = L + C
S7:  A = E + A
```

Problem 2.8 According to program order, the following six arithmetic expressions need to be executed in minimum time. Assume that all are integer operands already loaded into working registers. No memory reference is needed for the operand fetch. Also, all intermediate or final results are written back to working registers without conflicts.

P1: $X \leftarrow (A + B) \times (A - B)$

P2: $Y \leftarrow (C + D) / (C - D)$

P3: $Z \leftarrow X + Y$

P4: $A \leftarrow E \times F$

P5: $Y \leftarrow E - Z$

P6: $B \leftarrow (X - F) \times A$

(a) Use the minimum number of working registers to rewrite the above HLL program into a minimum-length assembly language code using arithmetic opcodes *add, subtract, multiply,* and *divide* exclusively. Assume a fixed instruction format with three register fields: two for sources and one for destinations.

(b) Perform a flow analysis of the assembly code obtained in part (a) to reveal all data dependences with a dependence graph.

(c) The CPU is assumed to have two *add units,* one *multiply unit,* and one *divide unit.* Work out an optimal schedule to execute the assembly code in minimum time, assuming 1 cycle for the add unit, 3 cycles for the multiply unit, and 18 cycles for the divide unit to complete the execution of one instruction. Ignore all overhead caused by instruction fetch, decode, and writeback. No pipelining is assumed here.

Problem 2.9 Consider the following assembly language code. Exploit the maximum degree of parallelism among the 16 instructions, assuming no resource conflicts and multiple functional units are available simultaneously. For simplicity, no pipelining is assumed. All instructions take one machine cycle to execute. Ignore all other overhead.

```
 1:  Load R1, A          /R1 ← Mem(A)/
 2:  Load R2, B          /R2 ← Mem(B)/
 3:  Mul R3, R1, R2      /R3 ← (R1) × (R2)/
 4:  Load R4, D          /R4 ← Mem(D)/
 5:  Mul R5, R1, R4      /R5 ← (R1) × (R4)/
 6:  Add R6, R3, R5      /R6 ← (R3) + (R5)/
 7:  Store X, R6         /Mem(X) ← (R6)/
 8:  Load R7, C          /R7 ← Mem(C)/
 9:  Mul R8, R7, R4      /R8 ← (R7) × (R4)/
10:  Load R9, E          /R9 ← Mem(E)/
11:  Add R10, R8, R9     /R10 ← (R8) + (R9)/
12:  Store Y, R10        /Mem(Y) ← (R10)/
13:  Add R11, R6, R10    /R11 ← (R6) + (R10)/
14:  Store U, R11        /Mem(U) ← (R11)/
15:  Sub R12, R6, R10    /R12 ← (R6) – (R10)/
16:  Store V, R12        /Mem(V) ← (R12)/
```

(a) Draw a program graph with 16 nodes to show the flow relationships among the 16 instructions.

(b) Consider the use of a three-issue superscalar processor to execute this program fragment in minimum time. The processor can issue one memory-access instruction (Load or Store but not both), one Add/Sub instruction, and one Mul (multiply) instruction per cycle. The Add unit, Load/Store unit, and Multiply unit can be used simultaneously if there is no data dependence.

Problem 2.10 Repeat part (b) of Problem 2.9 on a dual-processor system with shared memory. Assume that the same superscalar processors are used and that all instructions take one cycle to execute.

(a) Partition the given program into two balanced halves. You may want to insert some load or store instructions to pass intermediate results generated by the two processors to each other. Show the divided program flow graph with the final output U and V generated by the two processors separately.

(b) Work out an optimal schedule for parallel execution of the above divided program by the two processors in minimum time.

Problem 2.11 You are asked to design a direct network for a multicomputer with 64 nodes using a three-dimensional torus, a six-dimensional binary hypercube, and cube-connected-cycles (CCC) with a minimum diameter. The following questions are related to the relative merits of these network topologies:

(a) Let d be the node degree, D the network diameter, and l the total number of links in a network. Suppose the quality of a network is measured by $(d \times D \times l)^{-1}$. Rank the three architectures according to this quality measure.

(b) A *mean internode distance* is defined as the average number of hops (links) along the shortest path for a message to travel from one node to another. The average is calculated for all (source, destination) pairs. Order the three architectures based on their mean internode distances, assuming that the probability that a node will send a message to all other nodes with distance i is $(D - i + 1)/\Sigma_{k=1}^{D}$ k, where D is the network diameter.

Problem 2.12 Consider an Illiac mesh (8 × 8), a binary hypercube, and a barrel shifter, all with 64 nodes labeled $N_0, N_1, \ldots, N_{63}$. All network links are bidirectional.

(a) List all the nodes reachable from node N_0 in exactly three steps for each of the three networks.

(b) Indicate in each case the tightest upper bound on the minimum number of routing steps needed to send data from any node N_i to another node N_j.

(c) Repeat part (b) for a larger network with 1024 nodes.

Problem 2.13 Compare *buses*, *crossbar switches*, and *multistage networks* for building a multiprocessor system with n processors and m shared-memory modules. Assume a word length of w bits and that 2 × 2 switches are used in building the multistage networks. The comparison study is carried out separately in each of the following four categories:

(a) Hardware complexities such as switching, arbitration, wires, connector, or cable requirements.

(b) Minimum latency in unit data transfer between the processor and memory module.

(c) Bandwidth range available to each processor.

(d) Communication capabilities such as permutations, data broadcast, blocking handling, etc.

Problem 2.14 Answer the following questions related to multistage networks:

(a) How many legitimate states are there in a 4 × 4 switch module, including both broadcast and permutations? Justify your answer with reasoning.

(b) Construct a 64-input Omega network using 4 × 4 switch modules in multiple stages. How many permutations can be implemented directly in a single pass through the network without blocking?

(c) What is the percentage of one-pass permutations compared with the total number of permutations achievable in one or more passes through the network?

Problem 2.15 Topologically equivalent networks are those whose graph representations are isomorphic with the same interconnection capabilities. Prove the topological equivalence among the Omega, Flip, and Baseline networks.

(a) Prove that the Omega network (Fig. 2.24) is topologically equivalent to the Baseline network (Fig. 2.25b).

(b) The *Flip network* (Fig. 2.27) is constructed using inverse perfect shuffle (Fig. 2.14b) for interstage connections. Prove that the Flip network is topologically equivalent to the Baseline network.

(c) Based on the results obtained in (a) and (b), prove the topological equivalence between the Flip network and the Omega network.

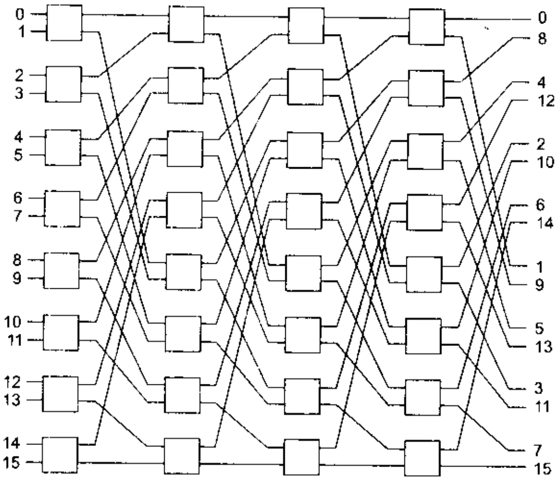


Fig. 2.27 A 16 × 16 Flip network (Courtesy of Ken Batcher; reprinted from *Proc. Int. Conf. Parallel Processing*, 1976)

Problem 2.16 Answer the following questions for the *k*-ary *n*-cube network:

(a) How many nodes does the network contain?

(b) What is the network diameter?

(c) What is the bisection bandwidth?

(d) What is the node degree?

(e) Explain the graph-theoretic relationship among *k*-ary *n*-cube networks and rings, meshes, tori, binary *n*-cubes, and Omega networks.

(f) Explain the difference between a conventional torus and a folded torus.

(g) Under the assumption of constant wire bisection, why do low-dimensional networks (tori) have lower latency and higher hot-spot throughput than high-dimensional networks (hypercubes)?

Problem 2.17 Read the paper on fat trees by Leiserson, which appeared in *IEEE Trans.* Computers, pp. 892–901, Oct. 1985. Answer the following questions related to the organization and application of fat trees:

(a) Explain the advantages of using binary fat trees over conventional binary trees as a multiprocessor interconnection network.

(b) A *universal fat tree* is defined as a fat tree of n nodes with root capacity w, where $n^{2/3} \le w \le n$, and for each channel c_k at level k of the tree, the capacity is

$$c_k = \min \left([n/2^k], [w/2^{2k/3}]\right)$$

Prove that the capacities of a universal fat tree grow exponentially as we go up the tree from the leaves. The channel capacity is defined here as the number of wires in a channel.

Problem 2.18 Read the paper on k-ary n-cube networks by Dally, which appeared in IEEE Trans. Computers, June 1990, pp. 775–785. Answer the following questions related to the network properties and applications as a VLSI communication network:

(a) Prove that the bisection width B of a k-ary n-cube with w-bit wide communication channels is

$$B(k, n) = 2w\sqrt{N}k^{n/2-1} = 2wN/k$$

where $N = k^n$ is the network size.

(b) Prove that the hot-spot throughput of a k-ary n-cube network with deterministic routing is equal to the bandwidth of a single channel $w = k - 1$, under the assumption of a constant wire cost.

Problem 2.19 Network embedding is a technique to implement a network A on a network B. Explain how to perform the following network embeddings:

(a) Embed a two-dimensional torus $r \times r$ on an n-dimensional hypercube with $N = 2^n$ nodes where $r^2 = 2^n$.

(b) Embed the largest ring on a CCC with $N = k \times 2^k$ nodes and $k \ge 3$.

(c) Embed a complete balanced binary tree with maximum height on a mesh of $r \times r$ nodes.

Problem 2.20 Read the paper on hypernets by Hwang and Ghosh, which appeared in *IEEE Trans. Computers*, Dec. 1989. Answer the following questions related to the network properties and applications of hypernets:

(a) Explain how hypernets integrate positive features of hypercube and tree-based topologies into one combined architecture.

(b) Prove that the average node degree of a hypernet can be maintained essentially constant when the network size is increased.

(c) Discuss the application potentials of hypernets in terms of message routing complexity, cost-effective support for global as well as localized communication, I/O capabilities, and fault tolerance.