FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

The McGraw-Hill Companies

ADVANCED COMPUTER ARCHITECTURE

Parallelism, Scalability, Programmability

Second Edition

Kai Hwang

Professor of Electrical Engineering and Computer Science
University of Southern California, USA

Naresh Jotwani

Director, School of Solar Energy
Pandit Deendayal Petroleum University
Gandhinagar, Gujarat

Tata McGraw Hill Education Private Limited
NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

Contents

Foreword to the First Edition *xv*
Preface to the Second Edition *xvii*
Preface to the First Edition *xxiii*

Part I Theory of Parallelism **1**

1. Parallel Computer Models **3**

1.1 The State of Computing *3*
1.1.1 Computer Development Milestones *3*
1.1.2 Elements of Modern Computers *6*
1.1.3 Evolution of Computer Architecture *8*
1.1.4 System Attributes to Performance *12*
1.2 Multiprocessors and Multicomputers *17*
1.2.1 Shared-Memory Multiprocessors *17*
1.2.2 Distributed-Memory Multicomputers *22*
1.2.3 A Taxonomy of MIMD Computers *24*
1.3 Multivector and SIMD Computers *25*
1.3.1 Vector Supercomputers *25*
1.3.2 SIMD Supercomputers *27*
1.4 PRAM and VLSI Models *29*
1.4.1 Parallel Random-Access Machines *30*
1.4.2 VLSI Complexity Model *33*
1.5 Architectural Development Tracks *36*
1.5.1 Multiple-Processor Tracks *36*
1.5.2 Multivector and SIMD Tracks *38*
1.5.3 Multithreaded and Dataflow Tracks *39*
Summary *40*
Exercises *41*

2. Program and Network Properties 44

2.1 Conditions of Parallelism *44*
2.1.1 Data and Resource Dependences *44*
2.1.2 Hardware and Software Parallelism *49*
2.1.3 The Role of Compilers *52*
2.2 Program Partitioning and Scheduling *52*
2.2.1 Grain Sizes and Latency *52*
2.2.2 Grain Packing and Scheduling *55*
2.2.3 Static Multiprocessor Scheduling *58*
2.3 Program Flow Mechanisms *61*
2.3.1 Control Flow Versus Data Flow *61*
2.3.2 Demand-Driven Mechanisms *65*
2.3.3 Comparison of Flow Mechanisms *65*
2.4 System Interconnect Architectures *66*
2.4.1 Network Properties and Routing *67*
2.4.2 Static Connection Networks *70*
2.4.3 Dynamic Connection Networks *77*
Summary *83*
Exercises *84*

3. Principles of Scalable Performance 89

3.1 Performance Metrics and Measures *89*
3.1.1 Parallelism Profile in Programs *89*
3.1.2 Mean Performance *92*
3.1.3 Efficiency, Utilization, and Quality *95*
3.1.4 Benchmarks and Performance Measures *97*
3.2 Parallel Processing Applications *99*
3.2.1 Massive Parallelism for Grand Challenges *99*
3.2.2 Application Models of Parallel Computers *102*
3.2.3 Scalability of Parallel Algorithms *104*
3.3 Speedup Performance Laws *108*
3.3.1 Amdahl's Law for a Fixed Workload *108*
3.3.2 Gustafson's Law for Scaled Problems *111*
3.3.3 Memory-Bounded Speedup Model *112*
3.4 Scalability Analysis and Approaches *116*
3.4.1 Scalability Metrics and Goals *116*
3.4.2 Evolution of Scalable Computers *120*
3.4.3 Research Issues and Solutions *123*
Summary *125*
Exercises *125*

Part I

Theory of Parallelism

Chapter 1
Parallel Computer Models

Chapter 2
Program and Network Properties

Chapter 3
Principles of Scalable Performance

Summary

This theoretical part presents computer models, program behavior, architectural choices, scalability, programmability, and performance issues related to parallel processing. These topics form the foundations for designing high-performance computers and for the development of supporting software and applications.

Physical computers modeled include shared-memory multiprocessors, message-passing multicomputers, vector supercomputers, synchronous processor arrays, and massively parallel processors. The theoretical parallel random-access machine (PRAM) model is also presented. Differences between the PRAM model and physical architectural models are discussed. The VLSI complexity model is presented for implementing parallel algorithms directly in integrated circuits.

Network design principles and parallel program characteristics are introduced. These include dependence theory, computing granularity, communication latency, program flow mechanisms, network properties, performance laws, and scalability studies. This evolving theory of parallelism consolidates our understanding of parallel computers, from abstract models to hardware machines, software systems, and performance evaluation.

The McGraw-Hill Companies

1

Parallel Computer Models

Over the last two decades, computer and communication technologies have literally transformed the world we live in. Parallel processing has emerged as the key enabling technology in modern computers, driven by the ever-increasing demand for higher performance, lower costs, and sustained productivity in real-life applications.

Parallelism appears in various forms, such as lookahead, pipelining, vectorization, concurrency, simultaneity, data parallelism, partitioning, interleaving, overlapping, multiplicity, replication, time sharing, space sharing, multitasking, multiprogramming, multithreading, and distributed computing at different processing levels.

In this chapter, we model physical architectures of parallel computers, vector supercomputers, multiprocessors, multicomputers, and massively parallel processors. Theoretical machine models are also presented, including the parallel random-access machines (PRAMs) and the complexity model of VLSI (very large-scale integration) circuits. Architectural development tracks are identified with case studies in the book. Hardware and software subsystems are introduced to pave the way for detailed studies in subsequent chapters.

1.1 THE STATE OF COMPUTING

Modern computers are equipped with powerful hardware facilities driven by extensive software packages. To assess state-of-the-art computing, we first review historical milestones in the development of computers. Then we take a grand tour of the crucial hardware and software elements built into modern computer systems. We then examine the evolutional relations in milestone architectural development. Basic hardware and software factors are identified in analyzing the performance of computers.

1.1.1 Computer Development Milestones

Prior to 1945, computers were made with mechanical or electromechanical parts. The earliest mechanical computer can be traced back to 500 BC in the form of the abacus used in China. The abacus is manually operated to perform decimal arithmetic with carry propagation digit by digit.

Blaise Pascal built a mechanical adder/subtractor in France in 1642. Charles Babbage designed a difference engine in England for polynomial evaluation in 1827. Konrad Zuse built the first binary mechanical computer in Germany in 1941. Howard Aiken proposed the very first electromechanical decimal computer, which was built as the Harvard Mark I by IBM in 1944. Both Zuse's and Aiken's machines were designed for general-purpose computations.

Obviously, the fact that computing and communication were carried out with moving mechanical parts greatly limited the computing speed and reliability of mechanical computers. Modern computers were marked by the introduction of electronic components. The moving parts in mechanical computers were replaced by high-mobility electrons in electronic computers. Information transmission by mechanical gears or levers was replaced by electric signals traveling almost at the speed of light.

Computer Generations Over the past several decades, electronic computers have gone through roughly five generations of development. Table 1.1 provides a summary of the five generations of electronic computer development. Each of the first three generations lasted about 10 years. The fourth generation covered a time span of 15 years. The fifth generation today has processors and memory devices with more than 1 billion transistors on a single silicon chip.

The division of generations is marked primarily by major changes in hardware and software technologies. The entries in Table 1.1 indicate the new hardware and software features introduced with each generation. Most features introduced in earlier generations have been passed to later generations.

Table 1.1 *Five Generations of Electronic Computers*

Generation	*Technology and Architecture*	*Software and Applications*	*Representative Systems*
First (1945–54)	Vacuum tubes and relay memories, CPU driven by PC and accumulator, fixed-point arithmetic.	Machine/assembly languages, single user, no subroutine linkage, programmed I/O using CPU.	ENIAC, Princeton IAS, IBM 701.
Second (1955–64)	Discrete transistors and core memories, floating-point arithmetic, I/O processors, multiplexed memory access.	HLL used with compilers, subroutine libraries, batch processing monitor.	IBM 7090, CDC 1604, Univac LARC.
Third (1965–74)	Integrated circuits (SSI/-MSI), microprogramming, pipelining, cache, and lookahead processors.	Multiprogramming and time-sharing OS, multiuser applications.	IBM 360/370, CDC 6600, TI-ASC, PDP-8.
Fourth (1975–90)	LSI/VLSI and semiconductor memory, multiprocessors, vector supercomputers, multicomputers.	Multiprocessor OS, languages, compilers, and environments for parallel processing.	VAX 9000, Cray X-MP, IBM 3090, BBN TC2000.
Fifth (1991–present)	Advanced VLSI processors, memory, and switches, high-density packaging, scalable architectures.	Superscalar processors, systems on a chip, massively parallel processing, grand challenge applications, heterogeneous processing.	See Tables 1.3–1.6 and Chapter 13.

Progress in Hardware As far as hardware technology is concerned, the first generation (1945–1954) used vacuum tubes and relay memories interconnected by insulated wires. The second generation (1955–1964)

was marked by the use of discrete transistors, diodes, and magnetic ferrite cores, interconnected by printed circuits.

The third generation (1965–1974) began to use *integrated circuits* (ICs) for both logic and memory in *small-scale* or *medium-scale integration* (SSI or MSI) and multilayered printed circuits. The fourth generation (1974-1991) used *large*-scale or *very large-scale integration* (LSI or VLSI). Semiconductor memory replaced core memory as computers moved from the third to the fourth generation.

The fifth generation (1991-present) is highlighted by the use of high-density and high-speed processor and memory chips based on advanced VLSI technology. For example, 64-bit GHz range processors are now available on a single chip with over one billion transistors.

The First Generation From the architectural and software points of view, first generation computers were built with a single *central processing unit* (CPU) which performed serial fixed-point arithmetic using a program counter, branch instructions, and an accumulator. The CPU must be involved in all memory access and *input/output* (I/O) operations. Machine or assembly languages were used.

Representative systems include the ENIAC (Electronic Numerical Integrator and Calculator) built at the Moore School of the University of Pennsylvania in 1950; the IAS (Institute for Advanced Studies) computer based on a design proposed by John von Neumann, Arthur Burks, and Herman Goldstine at Princeton in 1946; and the IBM 701, the first electronic stored-program commercial computer built by IBM in 1953. Subroutine linkage was not implemented in early computers.

The Second Generation Index registers, floating-point arithmetic, multiplexed memory, and I/O processors were introduced with second-generation computers. *High level languages* (HLLs), such as Fortran, Algol, and Cobol, were introduced along with compilers, subroutine libraries, and batch processing monitors. Register transfer language was developed by Irving Reed (1957) for systematic design of digital computers.

Representative systems include the IBM 7030 (the Stretch computer) featuring instruction lookahead and error-correcting memories built in 1962, the Univac LARC (Livermore Atomic Research Computer) built in 1959, and the CDC 1604 built in the 1960s.

The Third Generation The third generation was represented by the IBM/360–370 Series, the CDC 6600/7600 Series, Texas Instruments ASC (Advanced Scientific Computer), and Digital Equipment's PDP-8 Series from the mid-1960s to the mid 1970s.

Microprogrammed control became popular with this generation. Pipelining and cache memory were introduced to close up the speed gap between the CPU and main memory. The idea of multiprogramming was implemented to interleave CPU and I/O activities across multiple user programs. This led to the development of time-sharing *operating systems* (OS) using virtual memory with greater sharing or multiplexing of resources.

The Fourth Generation Parallel computers in various architectures appeared in the fourth generation of computers using shared or distributed memory or optional vector hardware. Multiprocessing OS, special languages, and compilers were developed for parallelism. Software tools and environments were created for parallel processing or distributed computing.

Representative systems include the VAX 9000, Cray X-MP, IBM/3090 VF, BBN TC-2000, etc. During these 15 years (1975–1990), the technology of parallel processing gradually became mature and entered the production mainstream.

The Fifth Generation These systems emphasize superscalar processors, cluster computers, and *massively parallel processing* (MPP). Scalable and latency tolerant architectures are being adopted in MPP systems using advanced VLSI technologies, high-density packaging, and optical technologies.

Fifth-generation computers achieved Teraflops (10^{12} floating-point operations per second) performance by the mid-1990s, and have now crossed the Petaflop (10^{15} floating point operations per second) range. *Heterogeneous processing* is emerging to solve large-scale problems using a network of heterogeneous computers. Early fifth-generation MPP systems were represented by several projects at Fujitsu (VPP500), Cray Research (MPP), Thinking Machines Corporation (the CM-5), and Intel (the Paragon). For present-day examples of advanced processors and systems; see Chapter 13.

1.1.2 Elements of Modern Computers

Hardware, software, and programming elements of a modern computer system are briefly introduced below in the context of parallel processing.

Computing Problems It has been long recognized that the concept of computer architecture is no longer restricted to the structure of the bare machine hardware. A modern computer is an integrated system consisting of machine hardware, an instruction set, system software, application programs, and user interfaces. These system elements are depicted in Fig. 1.1. The use of a computer is driven by real-life problems demanding cost effective solutions. Depending on the nature of the problems, the solutions may require different computing resources.

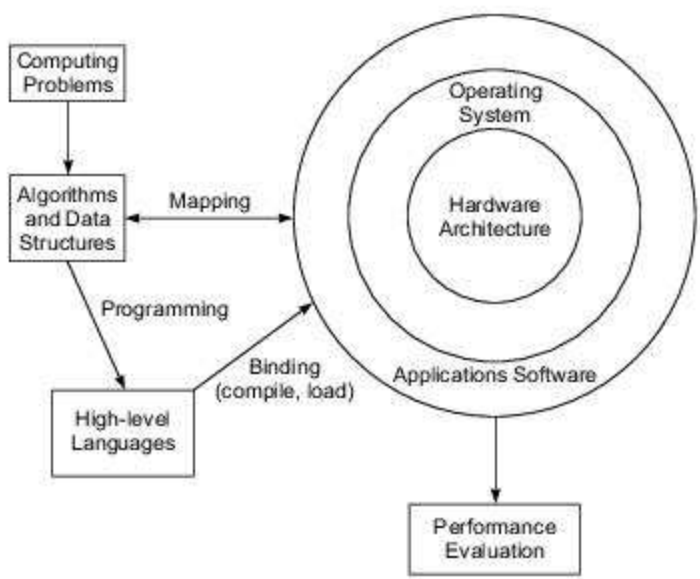


Fig. 1.1 Elements of a modern computer system

For numerical problems in science and technology, the solutions demand complex mathematical formulations and intensive integer or floating-point computations. For alphanumerical problems in business

and government, the solutions demand efficient transaction processing, large database management, and information retrieval operations.

For artificial intelligence (AI) problems, the solutions demand logic inferences and symbolic manipulations. These computing problems have been labeled *numerical computing*, *transaction processing*, and *logical* reasoning. Some complex problems may demand a combination of these processing modes.

Algorithms and Data Structures Special algorithms and data structures are needed to specify the computations and communications involved in computing problems. Most numerical algorithms are deterministic, using regularly structured data. Symbolic processing may use heuristics or nondeterministic searches over large knowledge bases.

Problem formulation and the development of parallel algorithms often require interdisciplinary interactions among theoreticians, experimentalists, and computer programmers. There are many books dealing with the design and mapping of algorithms or heuristics onto parallel computers. In this book, we are more concerned about the resources mapping problem than about the design and analysis of parallel algorithms.

Hardware Resources The system architecture of a computer is represented by three nested circles on the right in Fig. 1.1. A modern computer system demonstrates its power through coordinated efforts by hardware resources, an operating system, and application software. Processors, memory, and peripheral devices form the hardware core of a computer system. We will study instruction-set processors, memory organization, multiprocessors, supercomputers, multicomputers, and massively parallel computers.

Special hardware interfaces are often built into I/O devices such as display terminals, workstations, optical page scanners, magnetic ink character recognizers, modems, network adaptors, voice data entry, printers, and plotters. These peripherals are connected to mainframe computers directly or through local or wide-area networks.

In addition, software interface programs are needed. These software interfaces include file transfer systems, editors, word processors, device drivers, interrupt handlers, network communication programs, etc. These programs greatly facilitate the portability of user programs on different machine architectures.

Operating System An effective operating system manages the allocation and deallocation of resources during the execution of user programs. Beyond the OS, application software must be developed to benefit the users. Standard benchmark programs are needed for performance evaluation.

Mapping is a bidirectional process matching algorithmic structure with hardware architecture, and vice versa. Efficient mapping will benefit the programmer and produce better source codes. The mapping of algorithmic and data structures onto the machine architecture includes processor scheduling, memory maps, interprocessor communications, etc. These activities are usually architecture-dependent.

Optimal mappings are sought for various computer architectures. The implementation of these mappings relies on efficient compiler and operating system support. Parallelism can be exploited at algorithm design time, at program time, at compile time, and at run time. Techniques for exploiting parallelism at these levels form the core of parallel processing technology.

System Software Support Software support is needed for the development of efficient programs in high-level languages. The source code written in a HLL must be first translated into object code by an optimizing compiler. The *compiler* assigns variables to registers or to memory words, and generates machine operations corresponding to HLL operators, to produce machine code which can be recognized by the machine hardware. A *loader* is used to initiate the program execution through the OS kernel.

Resource binding demands the use of the compiler, assembler, loader, and OS kernel to commit physical machine resources to program execution. The effectiveness of this process determines the efficiency of hardware utilization and the programmability of the computer. Today, programming parallelism is still difficult for most programmers due to the fact that existing languages were originally developed for sequential computers. Programmers are sometimes forced to program hardware-dependent features instead of programming parallelism in a generic and portable way. Ideally, we need to develop a parallel programming environment with architecture-independent languages, compilers, and software tools.

To develop a parallel language, we aim for efficiency in its implementation, portability across different machines, compatibility with existing sequential languages, expressiveness of parallelism, and ease of programming. One can attempt a new language approach or try to extend existing sequential languages gradually. A new language approach has the advantage of using explicit high-level constructs for specifying parallelism. However, new languages are often incompatible with existing languages and require new compilers or new passes to existing compilers. Most systems choose the language extension approach; one way to achieve this is by providing appropriate function libraries.

Compiler Support There are three compiler upgrade approaches: *preprocessor*, *precompiler*, and *parallelizing compiler*. A preprocessor uses a sequential compiler and a low-level library of the target computer to implement high-level parallel constructs. The precompiler approach requires some program flow analysis, dependence checking, and limited optimizations toward parallelism detection. The third approach demands a fully developed parallelizing or vectorizing compiler which can automatically detect parallelism in source code and transform sequential codes into parallel constructs. These approaches will be studied in Chapter 10.

The efficiency of the binding process depends on the effectiveness of the preprocessor, the precompiler, the parallelizing compiler, the loader, and the OS support. Due to unpredictable program behavior, none of the existing compilers can be considered fully automatic or fully intelligent in detecting all types of parallelism. Very often *compiler directives* are inserted into the source code to help the compiler do a better job. Users may interact with the compiler to restructure the programs. This has been proven useful in enhancing the performance of parallel computers.

1.1.3 Evolution of Computer Architecture

The study of computer architecture involves both hardware organization and programming/software requirements. As seen by an assembly language programmer, computer architecture is abstracted by its instruction set, which includes opcode (operation codes), addressing modes, registers, virtual memory, etc.

From the hardware implementation point of view, the abstract machine is organized with CPUs, caches, buses, microcode, pipelines, physical memory, etc. Therefore, the study of architecture covers both instruction-set architectures and machine implementation organizations.

Over the past decades, computer architecture has gone through evolutional rather than revolutional changes. Sustaining features are those that were proven performance deliverers. As depicted in Fig. 1.2, we started with the von Neumann architecture built as a sequential machine executing scalar data . The sequential computer was improved from bit-serial to word-parallel operations, and from fixed-point to floating point operations. The von Neumann architecture is slow due to sequential execution of instructions in programs.

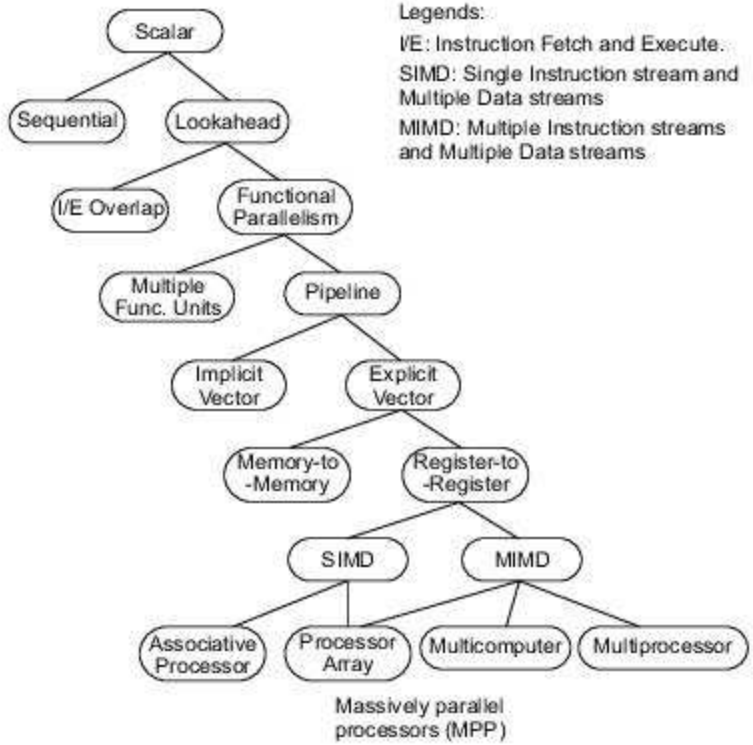


Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers

Lookahead, Parallelism, and Pipelining Lookahead techniques were introduced to prefetch instructions in order to overlap I/E (instruction fetch/decode and execution) operations and to enable functional parallelism. Functional parallelism was supported by two approaches: One is to use multiple functional units simultaneously, and the other is to practice pipelining at various processing levels.

The latter includes pipelined instruction execution, pipelined arithmetic computations, and memory-access operations. Pipelining has proven especially attractive in performing identical operations repeatedly over vector data strings. Vector operations were originally carried out implicitly by software-controlled looping using scalar pipeline processors.

Flynn's Classification Michael Flynn (1972) introduced a classification of various computer architectures based on notions of instruction and data streams. As illustrated in Fig. 1.3a, conventional sequential machines are called SISD (*single instruction stream over a single data stream*) computers. Vector computers are equipped with scalar and vector hardware or appear as SIMD (*single instruction stream over multiple data streams*) machines (Fig. 1.3b). Parallel computers are reserved for MIMD (*multiple instruction streams over multiple data streams*) machines.

An MISD (*multiple instruction streams and a single data stream*) machine is modeled in Fig. 1.3d. The same data stream flows through a linear array of processors executing different instruction streams. This

architecture is also known as *systolic arrays* (Kung and Leiserson, 1978) for pipelined execution of specific algorithms.

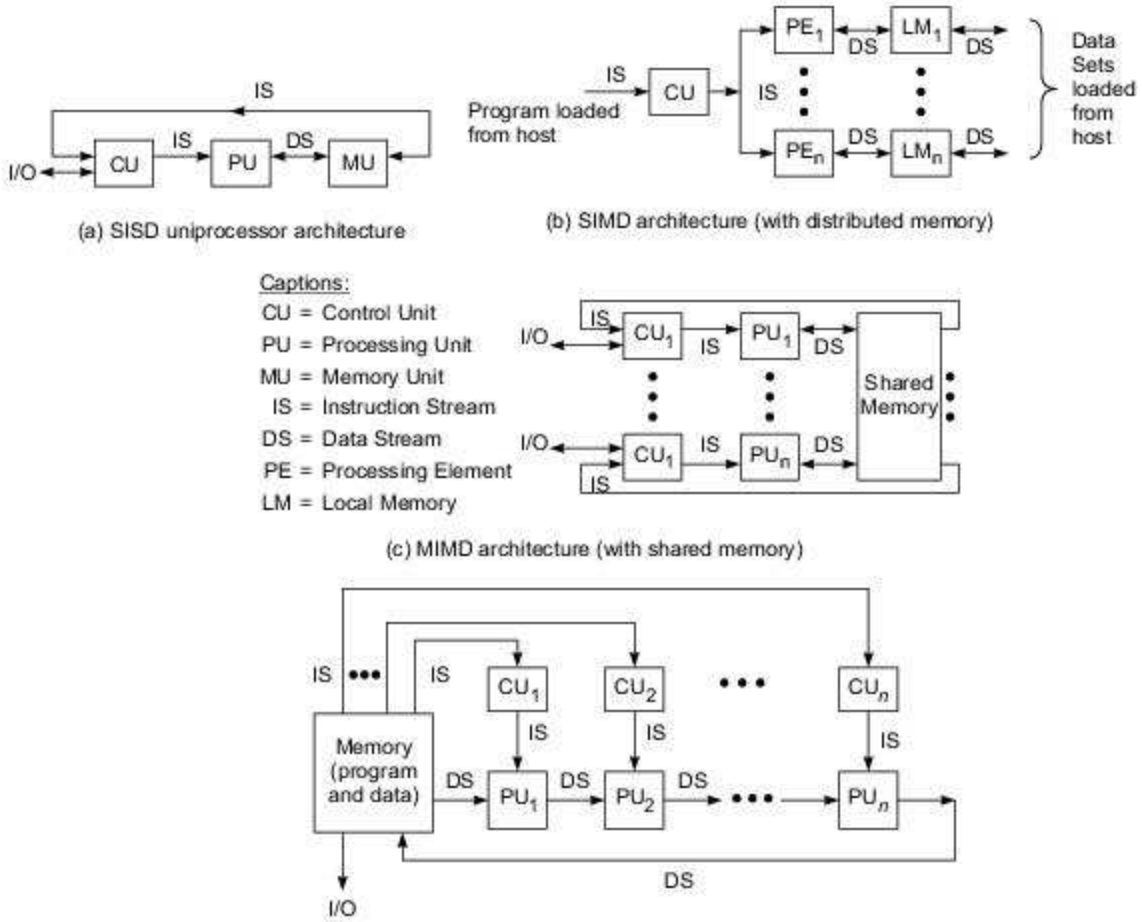


Fig. 1.3 Flynn's classification of computer architectures (Derived from Michael Flynn, 1972)

Of the four machine models, most parallel computers built in the past assumed the MIMD model for general-purpose computations. The SIMD and MISD models are more suitable for special-purpose computations. For this reason, MIMD is the most popular model, SIMD next, and MISD the least popular model being applied in commercial machines.

Parallel/Vector Computers Intrinsic parallel computers are those that execute programs in MIMD mode. There are two major classes of parallel computers, namely, *shared-memory multiprocessors* and *message-passing multicomputers*. The major distinction between multiprocessors and multicomputers lies in memory sharing and the mechanisms used for interprocessor communication.

The processors in a multiprocessor system communicate with each other through *shared variables* in a common memory. Each computer node in a multicomputer system has a local memory, unshared with other nodes. Interprocessor communication is done through *message passing* among the nodes.

Explicit vector instructions were introduced with the appearance of *vector processors*. A vector processor is equipped with multiple vector pipelines that can be concurrently used under hardware or firmware control. There are two families of pipelined vector processors:

Memory-to-memory architecture supports the pipelined flow of vector operands directly from the memory to pipelines and then back to the memory. *Register-to-register* architecture uses vector registers to interface between the memory and functional pipelines. Vector processor architectures will be studied in Chapter 8.

Another important branch of the architecture tree consists of the SIMD computers for synchronized vector processing. An SIMD computer exploits spatial parallelism rather than *temporal parallelism* as in a pipelined computer. SIMD computing is achieved through the use of an array of *processing elements* (PEs) synchronized by the same controller. Associative memory can be used to build SIMD associative processors. SIMD machines will be treated in Chapter 8 along with pipelined vector computers.

Development Layers A layered development of parallel computers is illustrated in Fig. 1.4, based on a classification by Lionel Ni (1990). Hardware configurations differ from machine to machine, even those of the same model. The address space of a processor in a computer system varies among different architectures. It depends on the memory organization, which is machine-dependent. These features are up to the designer and should match the target application domains.

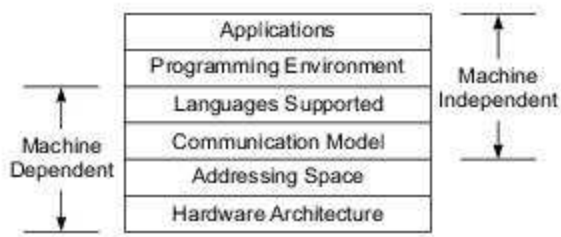


Fig. 1.4 Six layers for computer system development (Courtesy of Lionel Ni, 1990)

On the other hand, we want to develop application programs and programming environments which are machine-independent. Independent of machine architecture, the user programs can be ported to many computers with minimum conversion costs. High-level languages and communication models depend on the architectural choices made in a computer system. From a programmer's viewpoint, these two layers should be architecture-transparent.

Programming languages such as Fortran, C, C++, Pascal, Ada, Lisp and others can be supported by most computers. However, the communication models, shared variables versus message passing, are mostly machine-dependent. The Linda approach using *tuple spaces* offers an architecture-transparent communication model for parallel computers. These language features will be studied in Chapter 10.

Application programmers prefer more architectural transparency. However, kernel programmers have to explore the opportunities supported by hardware. As a good computer architect, one has to approach the problem from both ends. The compilers and OS support should be designed to remove as many architectural constraints as possible from the programmer.

New Challenges The technology of parallel processing is the outgrowth of several decades of research and industrial advances in microelectronics, printed circuits, high density packaging, advanced processors, memory systems, peripheral devices, communication channels, language evolution, compiler sophistication, operating systems, programming environments, and application challenges.

The rapid progress made in hardware technology has significantly increased the economical feasibility of building a new generation of computers adopting parallel processing. However, the major barrier preventing parallel processing from entering the production mainstream is on the software and application side.

To date, it is still fairly difficult to program parallel and vector computers. We need to strive for major progress in the software area in order to create a user-friendly environment for high-power computers. A whole new generation of programmers need to be trained to program parallelism effectively. High-performance computers provide fast and accurate solutions to scientific, engineering, business, social, and defense problems.

Representative real-life problems include weather forecast modeling, modeling of physical, chemical and biological processes, computer aided design, large-scale database management, artificial intelligence, crime control, and strategic defense initiatives, just to name a few. The application domains of parallel processing computers are expanding steadily. With a good understanding of scalable computer architectures and mastery of parallel programming techniques, the reader will be better prepared to face future computing challenges.

1.1.4 System Attributes to Performance

The ideal performance of a computer system demands a perfect match between machine capability and program behavior. Machine capability can be enhanced with better hardware technology, innovative architectural features, and efficient resources management. However, program behavior is difficult to predict due to its heavy dependence on application and run-time conditions.

There are also many other factors affecting program behavior, including algorithm design, data structures, language efficiency, programmer skill, and compiler technology. It is impossible to achieve a perfect match between hardware and software by merely improving only a few factors without touching other factors.

Besides, machine performance may vary from program to program. This makes *peak performance* an impossible target to achieve in real-life applications. On the other hand, a machine cannot be said to have an average performance either. All performance indices or benchmarking results must be tied to a program mix. For this reason, the performance should be described as a range or a distribution.

We introduce below fundamental factors for projecting the performance of a computer. These performance indicators are by no means conclusive in all applications. However, they can be used to guide system architects in designing better machines or to educate programmers or compiler writers in optimizing the codes for more efficient execution by the hardware.

Consider the execution of a given program on a given computer. The simplest measure of program performance is the *turnaround time*, which includes disk and memory accesses, input and output activities, compilation time, OS overhead, and CPU time. In order to shorten the turnaround time, one must reduce all these time factors.

In a multiprogrammed computer, the I/O and system overheads of a given program may overlap with the CPU times required in other programs. Therefore, it is fair to compare just the total CPU time needed for program execution. The CPU is used to execute both system programs and user programs, although often it is the user CPU time that concerns the user most.

Clock Rate and CPI The CPU (or simply the *processor*) of today's digital computer is driven by a clock with a constant *cycle time* τ. The inverse of the cycle time is the *clock rate* ($f = 1/\tau$). The size of a program is determined by its *instruction count* (I_c), in terms of the number of machine instructions to be executed in the program. Different machine instructions may require different numbers of clock cycles to execute. Therefore, the *cycles per instruction* (CPI) becomes an important parameter for measuring the time needed to execute each instruction.

For a given instruction set, we can calculate an *average* CPI over all instruction types, provided we know their frequencies of appearance in the program. An accurate estimate of the average CPI requires a large amount of program code to be traced over a long period of time. Unless specifically focusing on a single instruction type, we simply use the term CPI to mean the average value with respect to a given instruction set and a given program mix.

Performance Factors Let I_c be the number of instructions in a given program, or the instruction count. The CPU time (T in seconds/program) needed to execute the program is estimated by finding the product of three contributing factors:

$$T = I_c \times \text{CPI} \times \tau \tag{1.1}$$

The execution of an instruction requires going through a cycle of events involving the instruction fetch, decode, operand(s) fetch, execution, and store results. In this cycle, only the instruction decode and execution phases are carried out in the CPU. The remaining three operations may require access to the memory. We define a *memory cycle* as the time needed to complete one memory reference. Usually, a memory cycle is k times the processor cycle τ. The value of k depends on the speed of the cache and memory technology and processor-memory interconnection scheme used.

The CPI of an instruction type can be divided into two component terms corresponding to the total processor cycles and memory cycles needed to complete the execution of the instruction. Depending on the instruction type, the complete instruction cycle may involve one to as many as four memory references (one for instruction fetch, two for operand fetch, and one for store results). Therefore we can rewrite Eq. 1.1 as follows:

$$T = I_c \times (p + m \times k) \times \tau \tag{1.2}$$

where p is the number of processor cycles needed for the instruction decode and execution, m is the number of memory references needed, k is the ratio between memory cycle and processor cycle, I_c is the instruction count, and τ is the processor cycle time. Equation 1.2 can be further refined once the CPI components (p, m, k) are weighted over the entire instruction set.

System Attributes The above five performance factors (I_c, p, m, k, τ) are influenced by four system attributes: instruction-set architecture, compiler technology, CPU implementation and control, and cache and memory hierarchy, as specified in Table 1.2.

The instruction-set architecture affects the program length (I_c) and processor cycles needed (p). The compiler technology affects the values of I_c, p, and the memory reference count (m). The CPU implementation and control determine the total processor time ($p \cdot \tau$) needed. Finally, the memory technology and hierarchy design affect the memory access latency ($k \cdot \tau$). The above CPU time can be used as a basis in estimating the execution rate of a processor.

Table 1.2 *Performance Factors versus System Attributes*

System Attributes	*Performance Factors*				
	Instr. Count, I_c	*Average Cycles per Instruction, CPI*			*Processor Cycle Time, τ*
		Processor Cycles per Instruction, p	*Memory References per Instruction, m*	*Memory-Access Latency, k*	
Instruction-set Architecture	✓	✓			
Compiler Technology	✓	✓	✓		
Processor Implementation and Control		✓			✓
Cache and Memory Hierarchy				✓	✓

MIPS Rate Let C be the total number of clock cycles needed to execute a given program. Then the CPU time in Eq. 1.2 can be estimated as $T = C \times \tau = C/f$. Furthermore, $\text{CPI} = C/I_c$ and $T = I_c \times \text{CPI} \times \tau = I_c \times \text{CPI}/f$. The processor speed is often measured in terms of *million instructions per second* (MIPS). We simply call it the MIPS rate of a given processor. It should be emphasized that the MIPS rate varies with respect to a number of factors, including the clock rate (f), the instruction count (I_c), and the CPI of a given machine, as defined below:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{\text{CPI} \times 10^6} = \frac{f \times I_c}{C \times 10^6} \tag{1.3}$$

Based on Eq. 1.3, the CPU time in Eq. 1.2 can also be written as $T = I_c \times 10^{-6}/\text{MIPS}$. Based on the system attributes identified in Table 1.2 and the above derived expressions, we conclude by indicating the fact that the MIPS rate of a given computer is directly proportional to the clock rate and inversely proportional to the CPI. All four system attributes, instruction set, compiler, processor, and memory technologies, affect the MIPS rate, which varies also from program to program because of variations in the instruction mix.

Floating Point Operations per Second Most compute-intensive applications in science and engineering make heavy use of floating point operations. Compared to instructions per second, for such applications a more relevant measure of performance is floating point operations per second, which is abbreviated as flops. With prefix mega (10^6), giga (10^9), tera (10^{12}) or peta (10^{15}), this is written as megaflops (mflops), gigaflops (gflops), teraflops or petaflops.

Throughput Rate Another important concept is related to how many programs a system can execute per unit time, called the *system throughput* W_s (in programs/second). In a multiprogrammed system, the system throughput is often lower than the CPU *throughput* W_p defined by:

$$W_p = \frac{f}{I_c \times \text{CPI}} \tag{1.4}$$

Note that $W_p = (\text{MIPS}) \times 10^6 / I_c$ from Eq. 1.3. The unit for W_p is also programs/second. The CPU throughput is a measure of how many programs can be executed per second, based only on the MIPS rate and average program length (I_c). Usually $W_s < W_p$ due to the additional system overheads caused by the I/O, compiler, and OS when multiple programs are interleaved for CPU execution by multiprogramming or time-sharing operations. If the CPU is kept busy in a perfect program-interleaving fashion, then $W_s = W_p$. This will probably never happen, since the system overhead often causes an extra delay and the CPU may be left idle for some cycles.

Example 1.1 MIPS ratings and performance measurement

Consider the use of two systems S_1 and S_2 to execute a hypothetical benchmark program. Machine characteristics and claimed performance are given below:

Machine	*Clock*	*Performance*	*CPU Time*
S_1	500 MHz	100 MIPS	12x seconds
S_2	2.5 GHz	1800 MIPS	x seconds

These data indicate that the measured CPU time on S_1 is 12 times longer than that measured on S_2. The object codes running on the two machines have different lengths due to the differences in the machines and compilers used. All other overhead times are ignored.

Based on Eq. 1.3, we can see that the instruction count of the object code running on S_2 must be 1.5 times longer than that of the code running on S_1. Furthermore, the average CPI on S_1 is seen to be 5, while that on S_2 is 1.39 executing the same benchmark program.

S_1 has a typical CISC (*complex instruction set computing*) architecture, while S_2 has a typical RISC (*reduced instruction set computing*) architecture to be characterized in Chapter 4. This example offers a simple comparison between the two types of computers based on a single program run. When a different program is run, the conclusion may not be the same.

We cannot calculate the CPU throughput W_p unless we know the program length and the average CPI of each code. The system throughput W_s should be measured across a large number of programs over a long observation period. The message being conveyed is that one should not draw a sweeping conclusion about the performance of a machine based on one or a few program runs.

Programming Environments The programmability of a computer depends on the programming environment provided to the users. In fact, the marketability of any new computer system depends on the creation of a user-friendly environment in which programming becomes a productive undertaking rather than a challenge. We briefly introduce below the environmental features desired in modern computers.

Conventional uniprocessor computers are programmed in a *sequential environment* in which instructions are executed one after another in a sequential manner. In fact, the original UNIX/OS kernel was designed to respond to one system call from the user process at a time. Successive system calls must be serialized through the kernel.

When using a parallel computer, one desires a *parallel environment* where parallelism is automatically exploited. Language extensions or new constructs must be developed to specify parallelism or to facilitate easy detection of parallelism at various granularity levels by more intelligent compilers.

Besides parallel languages and compilers, the operating systems must be also extended to support parallel processing. The OS must be able to manage the resources behind parallelism. Important issues include parallel scheduling of concurrent processes, inter-process communication and synchronization, shared memory allocation, and shared peripheral and communication links.

Implicit Parallelism An implicit approach uses a conventional language, such as C, C++, Fortran, or Pascal, to write the source program. The sequentially coded source program is translated into parallel object code by a parallelizing compiler. As illustrated in Fig. 1.5a, this compiler must be able to detect parallelism and assign target machine resources. This compiler approach has been applied in programming shared-memory multiprocessors.

With parallelism being implicit, success relies heavily on the "intelligence" of a parallelizing compiler. This approach requires less effort on the part of the programmer.

Explicit Parallelism The second approach (Fig. 1.5b) requires more effort by the programmer to develop a source program using parallel dialects of C, C++, Fortran, or Pascal. Parallelism is explicitly specified in the user programs. This reduces the burden on the compiler to detect parallelism. Instead, the compiler needs to preserve parallelism and, where possible, assigns target machine resources. New programming language Chapel (see Chapter 13) is in this category.

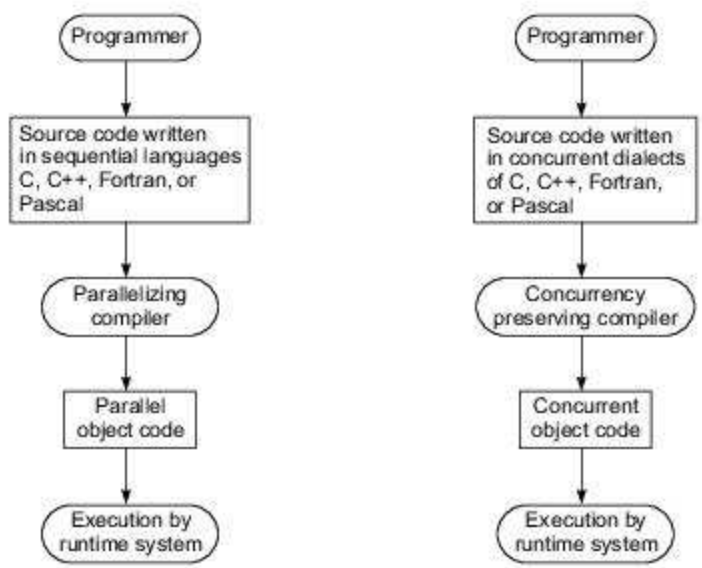


Fig. 1.5 Two approaches to parallel programming (Courtesy of Charles Seitz; adapted with permission from "Concurrent Architectures", p. 51 and p. 53, *VLSI and Parallel Computation*, edited by Suaya and Birtwistle, Morgan Kaufmann Publishers, 1990)

Special software tools are needed to make an environment more friendly to user groups. Some of the tools are parallel extensions of conventional high-level languages. Others are integrated environments which include tools providing different levels of program abstraction, validation, testing, debugging, and tuning; performance prediction and monitoring; and visualization support to aid program development, performance measurement, and graphics display and animation of computational results.

1.2 MULTIPROCESSORS AND MULTICOMPUTERS

Two categories of parallel computers are architecturally modeled below. These physical models are distinguished by having a shared common memory or unshared distributed memories. Only architectural organization models are described in Sections 1.2 and 1.3. Theoretical and complexity models for parallel computers are presented in Section 1.4.

1.2.1 Shared-Memory Multiprocessors

We describe below three shared-memory multiprocessor models: the *uniform memory-access* (UMA) model, the *nonuniform-memory-access* (NUMA) model, and the *cache-only memory architecture* (COMA) model. These models differ in how the memory and peripheral resources are shared or distributed.

The UMA Model In a UMA multiprocessor model (Fig. 1.6), the physical memory is uniformly shared by all the processors. All processors have equal access time to all memory words, which is why it is called uniform memory access. Each processor may use a private cache. Peripherals are also shared in some fashion.

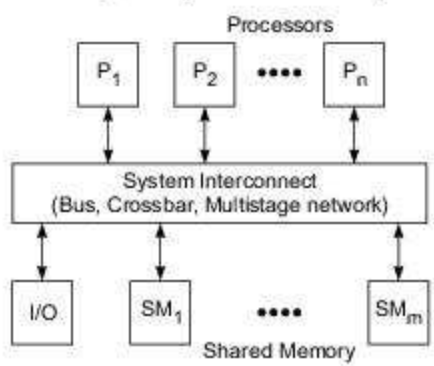


Fig. 1.6 The UMA multiprocessor model

Multiprocessors are called *tightly coupled systems* due to the high degree of resource sharing. The system interconnect takes the form of a common bus, a crossbar switch, or a multistage network to be studied in Chapter 7.

Some computer manufacturers have *multiprocessor* (MP) extensions of their *uniprocessor* (UP) product line. The UMA model is suitable for general-purpose and timesharing applications by multiple users. It can be used to speed up the execution of a single large program in time-critical applications. To coordinate parallel events, synchronization and communication among processors are done through using shared variables in the common memory.

When all processors have equal access to all peripheral devices, the system is called a *symmetric* multiprocessor. In this case, all the processors are equally capable of running the executive programs, such as the OS kernel and I/O service routines.

In an *asymmetric* multiprocessor, only one or a subset of processors are executive-capable. An executive or a master processor can execute the operating system and handle I/O. The remaining processors have no I/O capability and thus are called *attached processors* (APs). Attached processors execute user codes under the supervision of the master processor. In both MP and AP configurations, memory sharing among master and attached processors is still in place.

Example 1.2 Approximated performance of a multiprocessor

This example exposes the reader to parallel program execution on a shared memory multiprocessor system. Consider the following Fortran program written for sequential execution on a uniprocessor system. All the arrays, A(I), B(I), and C(I), are assumed to have N elements.

```
L1:             Do 10 I = 1, N
L2:             A(I) = B(I) + C(I)
L3:       10    Continue
L4:             SUM = 0
L5:             Do 20 J = 1, N
L6:             SUM = SUM + A(J)
L7:       20    Continue
```

Suppose each line of code L2, L4, and L6 takes 1 machine cycle to execute. The time required to execute the program control statements L1, L3, L5, and L7 is ignored to simplify the analysis. Assume that k cycles are needed for each interprocessor communication operation via the shared memory.

Initially, all arrays are assumed already loaded in the main memory and the short program fragment already loaded in the instruction cache. In other words, instruction fetch and data loading overhead is ignored. Also, we ignore bus contention or memory access conflicts problems. In this way, we can concentrate on the analysis of CPU demand.

The above program can be executed on a sequential machine in $2N$ cycles under the above assumptions. N cycles are needed to execute the N independent iterations in the I loop. Similarly, N cycles are needed for the J loop, which contains N recursive iterations.

To execute the program on an M-processor system, we partition the looping operations into M sections with $L = N/M$ elements per section. In the following parallel code, **Doall** declares that all M sections be executed by M processors in parallel.

For M-way parallel execution, the sectioned I loop can be done in L cycles.

The sectioned J loop produces M partial sums in L cycles. Thus $2L$ cycles are consumed to produce all M partial sums. Still, we need to merge these M partial sums to produce the final sum of N elements.

```
Doall K = 1, M
      Do 10 I = (K – 1) * L + 1, K * L
            A(I) = B(I) + C(I)
10    Continue
      SUM(K) = 0
      Do 20 J = 1, L
            SUM(K) = SUM(K) + A((K – 1) * L + J)
20    Continue
Endall
```

The addition of each pair of partial sums requires k cycles through the shared memory. An l-level binary adder tree can be constructed to merge all the partial sums, where $l = \log_2 M$. The adder tree takes $l(k + 1)$ cycles to merge the M partial sums sequentially from the leaves to the root of the tree. Therefore, the multiprocessor requires $2L + l(k + 1) = 2N/M + (k + 1)\log_2 M$ cycles to produce the final sum.

Suppose $N = 2^{20}$ elements in the array. Sequential execution of the original program takes $2N = 2^{21}$ machine cycles. Assume that each IPC synchronization overhead has an average value of $k = 200$ cycles. Parallel execution on $M = 256$ processors requires $2^{13} + 1608 = 9800$ machine cycles.

Comparing the above timing results, the multiprocessor shows a speedup factor of 214 out of the maximum value of 256. Therefore, an efficiency of 214/256 = 83.6% has been achieved. We will study the speedup and efficiency issues in Chapter 3.

The above result was obtained under favorable assumptions about overhead. In reality, the resulting speedup might be lower after considering all software overhead and potential resource conflicts. Nevertheless, the example shows the promising side of parallel processing if the interprocessor communication overhead can be maintained to a sufficiently low level, represented here in the value of k.

The NUMA Model A NUMA multiprocessor is a shared-memory system in which the access time varies with the location of the memory word. Two NUMA machine models are depicted in Fig. 1.7. The shared memory is physically distributed to all processors, called local memories. The collection of all *local memories* forms a global address space accessible by all processors.

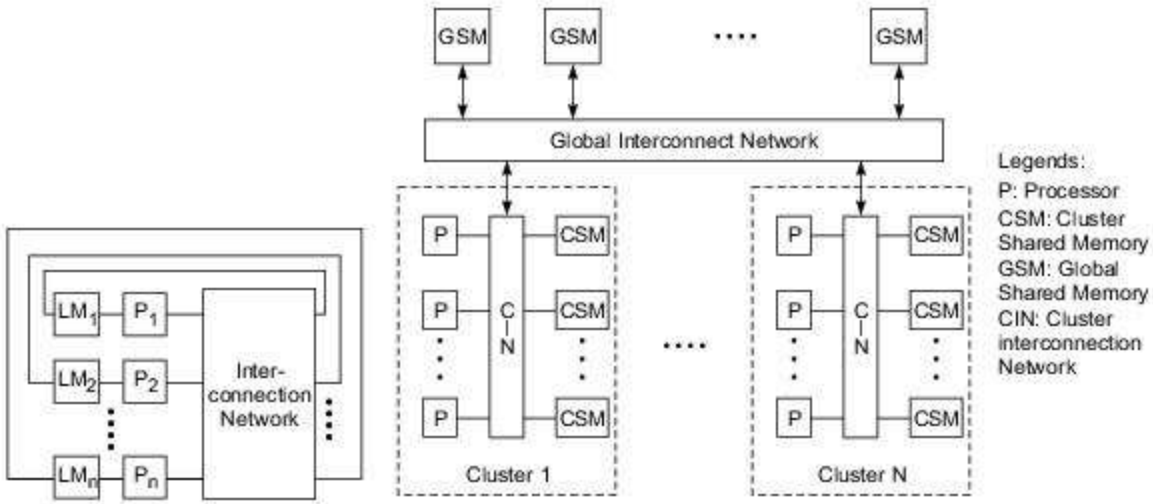
It is faster to access a local memory with a local processor. The access of remote memory attached to other processors takes longer due to the added delay through the interconnection network. The BBN TC-2000 Butterfly multiprocessor had the configuration shown in Fig. 1.7a.

Besides distributed memories, globally shared memory can be added to a multiprocessor system. In this case, there are three memory-access patterns: The fastest is local memory access. The next is global memory access. The slowest is access of remote memory as illustrated in Fig. 1.7b. As a matter of fact, the models shown in Figs. 1.6 and 1.7 can be easily modified to allow a mixture of shared memory and private memory with prespecified access rights.

A hierarchically structured multiprocessor is modeled in Fig. 1.7b. The processors are divided into several *clusters**. Each cluster is itself an UMA or a NUMA multiprocessor. The clusters are connected to *global shared-memory* modules. The entire system is considered a NUMA multiprocessor. All processors belonging to the same cluster are allowed to uniformly access the *cluster shared-memory* modules.

*The word 'cluster' is used in a different sense in cluster computing, as we shall see later.

All clusters have equal access to the global memory. However, the access time to the cluster memory is shorter than that to the global memory. One can specify the access rights among intercluster memories in various ways. The Cedar multiprocessor, built at the University of Illinois, had such a structure in which each cluster was an Alliant FX/80 multiprocessor.



(a) Shared local memories (e.g. the N Butterfly)

(b) A hierarchical cluster model (e.g. the Cedar system at the University of Illinois)

Fig. 1.7 Two NUMA models for multiprocessor systems

The COMA Model A multiprocessor using cache-only memory assumes the COMA model. Early examples of COMA machines include the Swedish Institute of Computer Science's Data Diffusion Machine (DDM, Hagersten et al., 1990) and Kendall Square Research's KSR-1 machine (Burkhardt et al., 1992). The COMA model is depicted in Fig. 1.8. Details of KSR-1 are given in Chapter 9.

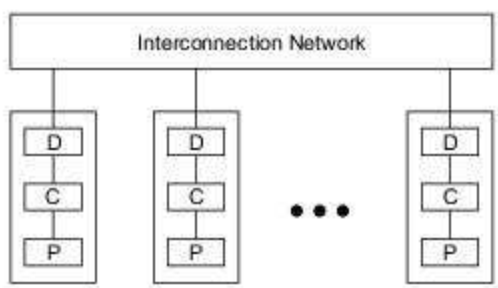


Fig. 1.8 The COMA model of a multiprocessor (P: Processor, C: Cache, D: Directory; e.g. the KSR-1)

The COMA model is a special case of a NUMA machine, in which the distributed main memories are converted to caches. There is no memory hierarchy at each processor node. All the caches form a global

address space. Remote cache access is assisted by the distributed cache directories (D in Fig. 1.8). Depending on the interconnection network used, sometimes hierarchical directories may be used to help locate copies of cache blocks. Initial data placement is not critical because data will eventually migrate to where it will be used.

Besides the UMA, NUMA, and COMA models specified above, other variations exist for multiprocessors. For example, a *cache-coherent non-uniform memory access* (CC-NUMA) model can be specified with distributed shared memory and cache directories. Early examples of the CC-NUMA model include the Stanford Dash (Lenoski et al., 1990) and the MIT Alewife (Agarwal et al., 1990) to be studied in Chapter 9. A cache-coherent COMA machine is one in which all cache copies must be kept consistent.

Representative Multiprocessors Several early commercially available multiprocessors are summarized in Table 1.3. They represent four classes of multiprocessors. The Sequent Symmetry S81 belonged to a class called minisupercomputers. The IBM System/390 models were high-end mainframes, sometimes called near-supercomputers. The BBN TC-2000 represented the MPP class.

Table 1.3 *Some Early Commercial Multiprocessor Systems*

Company and Model	*Hardware and Architecture*	*Software and Applications*	*Remarks*
Sequent Symmetry S-81	Bus-connected with 30 i386 processors, IPC via SLIC bus; Weitek floating-point accelerator.	DYNIX/OS, KAP/Sequent preprocessor, transaction multiprocessing.	Latter models designed with faster processors of the family.
IBM ES/9000 Model 900/VF	6 ES/9000 processors with vector facilities, crossbar connected to I/O channels and shared memory.	OS support: MVS, VM KMS, AIX/370, parallel Fortran, VSF V2.5 compiler.	Fiber optic channels, integrated cryptographic architecture.
BBN TC-2000	512 M88100 processors with local memory connected by a Butterfly switch, a NUMA machine.	Ported Mach/OS with multiclustering, parallel Fortran, time-critical applications.	Latter models designed with faster processors of the family.

The S-81 was a transaction processing multiprocessor consisting of 30 i386/i486 microprocessors tied to a common backplane bus. The IBM ES/9000 models were the latest IBM mainframes having up to 6 processors with attached vector facilities. The TC-2000 could be configured to have 512 M88100 processors interconnected by a multistage Butterfly network. This was designed as a NUMA machine for real-time or time-critical applications.

Multiprocessor systems are suitable for general-purpose multiuser applications where programmability is the major concern. A major shortcoming of multiprocessors is the lack of scalability. It is rather difficult to build MPP machines using centralized shared memory model. Latency tolerance for remote memory access is also a major limitation.

Packaging and cooling impose additional constraints on scalability. We will study scalability and programmability in subsequent chapters.

1.2.2 Distributed-Memory Multicomputers

A distributed-memory multicomputer system is modeled in Fig. 1.9. The system consists of multiple computers, often called *nodes*, interconnected by a message-passing network. Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or I/O peripherals.

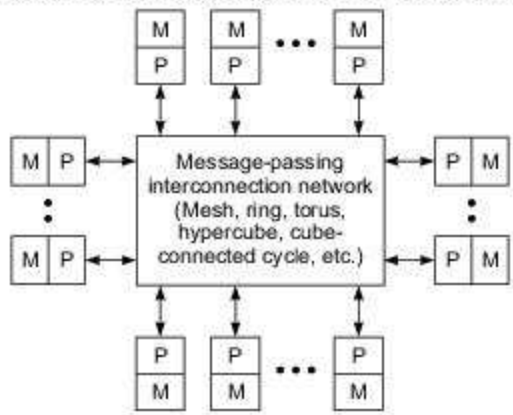


Fig. 1.9 Generic model of a message-passing multicomputer

The message-passing network provides point-to-point static connections among the nodes. All local memories are private and are accessible only by local processors. For this reason, traditional multicomputers have also been called *no-remote-memory-access* (NORMA) machines. Internode communication is carried out by passing messages through the static connection network. With advances in interconnection and network technologies, this model of computing has gained importance, because of its suitability for certain applications, scalability, and fault-tolerance.

Multicomputer Generations Modern multicomputers use hardware routers to pass messages. A computer node is attached to each router. The boundary router may be connected to I/O and peripheral devices. Message passing between any two nodes involves a sequence of routers and channels. Mixed types of nodes are allowed in a heterogeneous multicomputer. The internode communication in a heterogeneous multicomputer is achieved through compatible data representations and message-passing protocols.

Early message-passing multicomputers were based on processor board technology using hypercube architecture and software-controlled message switching. The Caltech Cosmic and Intel iPSC/1 represented this early development.

The second generation was implemented with mesh-connected architecture, hardware message routing, and a software environment for medium-grain distributed computing, as represented by the Intel Paragon and the Parsys SuperNode 1000.

Subsequent systems of this type are fine-grain multicomputers, early examples being the MIT J-Machine and Caltech Mosaic, implemented with both processor and communication gears on the same VLSI chip. For further discussion; see Chapter 13.

In Section 2.4, we will study various static network topologies used to construct multicomputers. Commonly used topologies include the *ring, tree, mesh, torus, hypercube, cube-connected cycle*, etc. Various communication patterns are demanded among the nodes, such as one-to-one, broadcasting, permutations, and multicast patterns.

Important issues for multicomputers include message-routing schemes, network flow control strategies, deadlock avoidance, virtual channels, message-passing primitives, and program decomposition techniques.

Representative Multicomputers Three early message-passing multicomputers are summarized in Table 1.4. With distributed processor/memory nodes, such machines are better in achieving a scalable performance. However, message passing imposes a requirement on programmers to distribute the computations and data sets over the nodes or to establish efficient communication among nodes.

Table 1.4 *Some Early Commercial Multicomputer Systems*

System Features	*Intel Paragon XP/S*	*nCUBE/2 6480*	*Parsys Ltd. SuperNode1000*
Node Types and Memory	50 MHz i860 XP computing nodes with 16–128 Mbytes per node, special I/O service nodes.	Each node contains a CISC 64-bit CPU, with FPU, 14 DMA ports, with 1–64 Mbytes /node.	EC-funded Esprit supernode built with multiple T-800 Transputers per node.
Network and I/O	2-D mesh with SCSI, HIPPI, VME, Ethernet, and custom I/O.	13-dimensional hypercube of 8192 nodes, 512-Gbyte memory, 64 I/O boards.	Reconfigurable interconnect, expandable to have 1024 processors.
OS and Software Task Parallelism Support	OSF conformance with 4.3 BSD, visualization and programming support.	Vertex/OS or UNIX supporting message passing using wormhole routing.	IDRIS/OS UNIX-compatible.
Application Drivers	General sparse matrix methods, parallel data manipulation, strategic computing.	Scientific number crunching with scalar nodes, database processing.	Scientific and academic applications.
Performance Remarks	5–300 Gflops peak 64-bit results, 2.8–160 GIPS peak integer performance.	27 Gflops peak, 36 Gbytes/s I/O	200 MIPS to 13 GIPS peak.

The Paragon system had a mesh architecture, and the nCUBE/2 had a hypercube architecture. The Intel i860s and some custom-designed VLSI processors were used as building blocks in these machines. All three OSs were UNIX-compatible with extended functions to support message passing.

Most multicomputers can be upgraded to yield a higher degree of parallelism with enhanced processors. We will study various massively parallel systems in Part III where the tradeoffs between scalability and programmability are analyzed.

1.2.3 A Taxonomy of MIMD Computers

Parallel computers appear as either SIMD or MIMD configurations. The SIMDs appeal more to special-purpose applications. It is clear that SIMDs are not size-scalable, but unclear whether large SIMDs are generation-scalable. The fact that CM-5 had an MIMD architecture, away from the SIMD architecture in CM-2, represents the architectural trend (see Chapter 8). Furthermore, the boundary between multiprocessors and multicomputers has become blurred in recent years.

The architectural trend for general-purpose parallel computers is in favor of MIMD configurations with various memory configurations (see Chapter 13). Gordon Bell (1992) has provided a taxonomy of MIMD machines, reprinted in Fig. 1.10. He considers shared-memory multiprocessors as having a single address space. Scalable multiprocessors or multicomputers must use distributed memory. Multiprocessors using centrally shared memory have limited scalability.

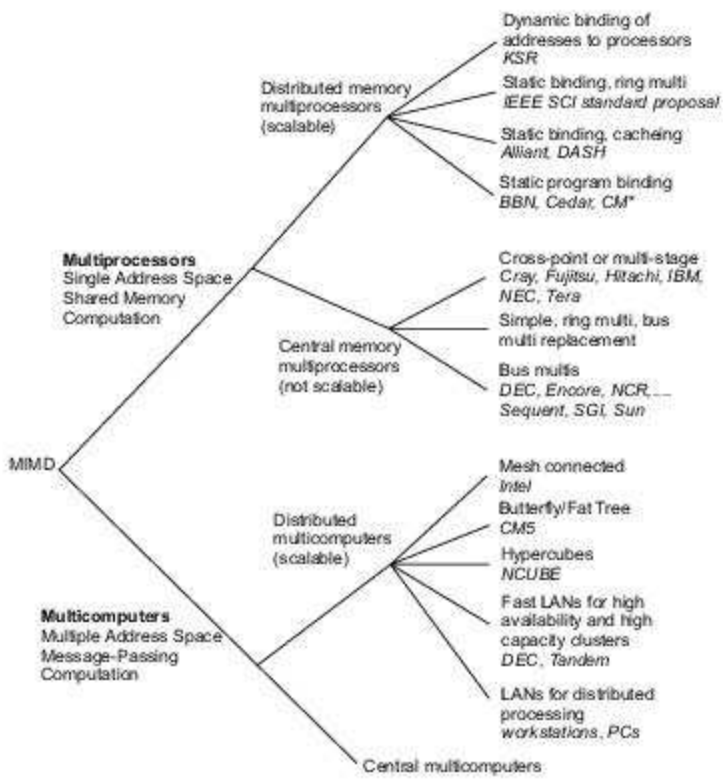


Fig. 1.10 Bell's taxonomy of MIMD computers (Courtesy of Gordon Bell; reprinted with permission from the *Communications of ACM*, August 1992)

Multicomputers use distributed memories with multiple address spaces. They are scalable with distributed memory. The evolution of fast LAN (*local area network*)-connected workstations has created "commodity supercomputing". Bell was the first to advocate high-speed workstation clusters interconnected by high-speed switches in lieu of special-purpose multicomputers. The CM-5 development was an early move in that direction.

The scalability of MIMD computers will be further studied in Section 3.4 and Chapter 9. In Part III, we will study distributed-memory multiprocessors (KSR-1, SCI, etc.); central-memory multiprocessors (Cray, IBM, DEC, Fujitsu, Encore, etc.); multicomputers by Intel, TMC, and nCUBE; fast LAN-based workstation clusters, and other exploratory research systems.

1.3 MULTIVECTOR AND SIMD COMPUTERS

In this section, we introduce supercomputers and parallel processors for vector processing and data parallelism. We classify supercomputers either as pipelined vector machines using a few powerful processors equipped with vector hardware, or as SIMD computers emphasizing massive data parallelism.

1.3.1 Vector Supercomputers

A vector computer is often built on top of a scalar processor. As shown in Fig. 1.11, the vector processor is attached to the scalar processor as an optional feature. Program and data are first loaded into the main memory through a host computer. All instructions are first decoded by the scalar control unit. If the decoded instruction is a scalar operation or a program control operation, it will be directly executed by the scalar processor using the scalar functional pipelines.

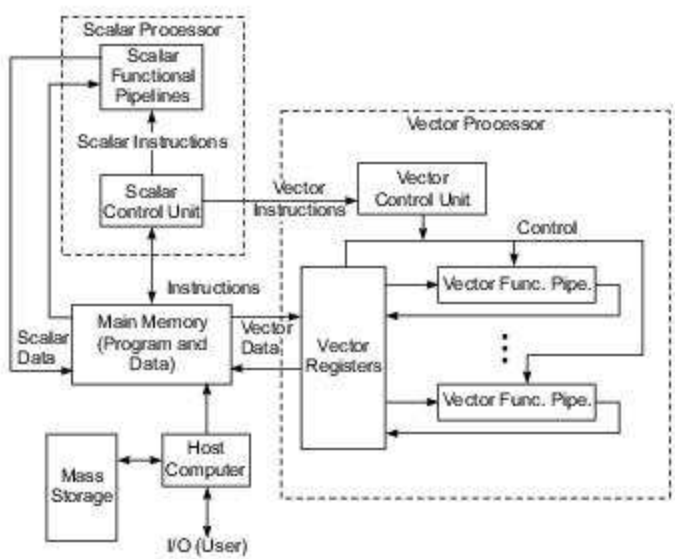


Fig. 1.11 The architecture of a vector supercomputer

If the instruction is decoded as a vector operation, it will be sent to the vector control unit. This control unit will supervise the flow of vector data between the main memory and vector functional pipelines. The vector data flow is coordinated by the control unit. A number of vector functional pipelines may be built into a vector processor. Two pipeline vector supercomputer models are described below.

Vector Processor Models Figure 1.11 shows a *register-to-register* architecture. Vector registers are used to hold the vector operands, intermediate and final vector results. The vector functional pipelines retrieve operands from and put results into the vector registers. All vector registers are programmable in user instructions. Each vector register is equipped with a component counter which keeps track of the component registers used in successive pipeline cycles.

The length of each vector register is usually fixed, say, sixty-four 64-bit component registers in a vector register in a Cray Series supercomputer. Other machines, like the Fujitsu VP2000 Series, use reconfigurable vector registers to dynamically match the register length with that of the vector operands.

In general, there are fixed numbers of vector registers and functional pipelines in a vector processor. Therefore, both resources must be reserved in advance to avoid resource conflicts between different vector operations. Some early vector-register based supercomputers are summarized in Table 1.5.

Table 1.5 *Some Early Commercial Vector Supercomputers*

System Model	*Vector Hardware Architecture and Capabilities*	*Compiler and Software Support*
Convex C3800 family	GaAs-based multiprocessor with 8 processors and 500-Mbyte/s access port. 4 Gbytes main memory. 2 Gflops peak performance with concurrent scalar/vector operations.	Advanced C, Fortran, and Ada vectorizing and parallelizing compilers. Also supported interprocedural optimization, POSIX 1003.1/OS plus I/O interfaces and visualization system
Digital VAX 9000 System	Integrated vector processing in the VAX environment, 125–500 Mflops peak performance. 63 vector instructions. $16 \times 64 \times 64$ vector registers. Pipeline chaining possible.	MS or ULTRIX/OS, VAX Fortran and VAX Vector Instruction Emulator (VVIEF) for vectorized program debugging.
Cray Research Y-MP and C-90	Y-MP ran with 2, 4, or 8 processors, 2.67 Gflop peak with Y-MP8256. C-90 had 2 vector pipes/CPU built with 10K gate ECL with 16 Gflops peak performance.	CF77 compiler for automatic vectorization, scalar optimization, and parallel processing. UNICOS improved from UNIX/V and Berkeley BSD/OS.

A *memory-to-memory* architecture differs from a register-to-register architecture in the use of a vector stream unit to replace the vector registers. Vector operands and results are directly retrieved from and stored into the main memory in superwords, say, 512 bits as in the Cyber 205.

Pipelined vector supercomputers started with uniprocessor models such as the Cray 1 in 1976. Subsequent supercomputer systems offered both uniprocessor and multiprocessor models such as the Cray Y-MP Series.

Representative Supercomputers Over a dozen pipelined vector computers have been manufactured, ranging from workstations to mini- and supercomputers. Notable early examples include the Stardent 3000 multiprocessor equipped with vector pipelines, the Convex C3 Series, the DEC VAX 9000, the IBM 390/VF, the Cray Research Y-MP family, the NEC SX Series, the Fujitsu VP2000, and the Hitachi S-810/20. For further discussion, see Chapters 8 and 13.

The Convex C1 and C2 Series were made with ECL/CMOS technologies. The latter C3 Series was based on GaAs technology.

The DEC VAX 9000 was Digital's largest mainframe system providing concurrent scalar/vector and multiprocessing capabilities. The VAX 9000 processors used a hybrid architecture. The vector unit was an optional feature attached to the VAX 9000 CPU. The Cray Y-MP family offered both vector and multiprocessing capabilities.

1.3.2 SIMD Supercomputers

In Fig. 1.3b, we have shown an abstract model of SIMD computers having a single instruction stream over multiple data streams. An operational model of SIMD computers is presented below (Fig. 1.12) based on the work of H. J. Siegel (1979). Implementation models and case studies of SIMD machines are given in Chapter 8.

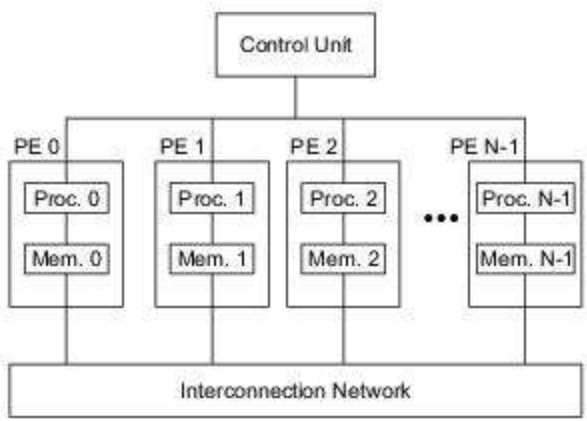


Fig. 1.12 Operational model of SIMD computers

SIMD Machine Model An operational model of an SIMD computer is specified by a 5-tuple:

$$M = (N, C, I, M, R) \tag{1.5}$$

where

(1) N is the number of *processing elements* (PEs) in the machine. For example, the Illiac IV had 64 PEs and the Connection Machine CM-2 had 65,536 PEs.

(2) C is the set of instructions directly executed by the *control unit* (CU), including scalar and program flow control instructions.

(3) I is the set of instructions broadcast by the CU to all PEs for parallel execution. These include arithmetic, logic, data routing, masking, and other local operations executed by each active PE over data within that PE.

(4) M is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets.

(5) R is the set of data-routing functions, specifying various patterns to be set up in the interconnection network for inter-PE communications.

One can describe a particular SIMD machine architecture by specifying the 5-tuple. An example SIMD machine is partially specified below.

Example 1.3 Operational specification of the MasPar MP-1 computer

We will study the detailed architecture of the MasPar MP-1 in Chapter 7. Listed below is a partial specification of the 5-tuple for this machine:

(1) The MP-1 was an SIMD machine with N = 1024 to 16,384 PEs, depending on which configuration is considered.

(2) The CU executed scalar instructions, broadcast decoded vector instructions to the PE array, and controlled inter-PE communications.

(3) Each PE was a register-based load/store RISC processor capable of executing integer operations over various data sizes and standard floating-point operations. The PEs received instructions from the CU.

(4) The masking scheme was built within each PE and continuously monitored by the CU which could set and reset the status of each PE dynamically at run time.

(5) The MP-1 had an X-Net mesh network plus a global multistage crossbar router for inter-CU-PE, X-Net nearest 8-neighbor, and global router communications.

Representative SIMD Computers Three early commercial SIMD supercomputers are summarized in Table 1.6. The number of PEs in these systems ranges from 4096 in the DAP610 to 16,384 in the MasPar MP-1 and 65,536 in the CM-2. Both the CM-2 and DAP610 were fine-grain, bit-slice SIMD computers with attached floating-point accelerators for blocks of PEs*.

Each PE of the MP-1 was equipped with a 1-bit logic unit, 4-bit integer ALU, 64-bit mantissa unit, and 16-bit exponent unit. Multiple PEs could be built on a single chip due to the simplicity of each PE. The MP-1

* With rapid advances in VLSI technology, use of bit-slice processors in systems has disappeared.

implemented 32 PEs per chip with forty 32-bit registers per PE. The 32 PEs were interconnected by an *X-Net* mesh, which was a 4-neighbor mesh augmented with diagonal dual-stage links.

The CM-2 implemented 16 PEs as a mesh on a single chip. Each 16-PE mesh chip was placed at one vertex of a 12-dimensional hypercube. Thus $16 \times 2^{12} = 2^{16} = 65{,}536$ PEs formed the entire SIMD array.

The DAP610 implemented 64 PEs as a mesh on a chip. Globally, a large mesh (64×64) was formed by interconnecting these small meshes on chips. Fortran 90 and modified versions of C, Lisp, and other sequential programming languages have been developed to program SIMD machines.

Table 1.6 *Some Early Commercial SIMD Supercomputers*

System Model	*SIMD Machine Architecture and Capabilities*	*Languages, Compilers and Software Support*
MasPar Computer Corporation MP-1 Family	Designed for configurations from 1024 to 16,384 processors with 26,000 MIPS or 1.3 Gflops. Each PE was a RISC processor, with 16 Kbytes local memory. An X-Net mesh plus a multistage crossbar interconnect.	Fortran 77, MasPar Fortran (MPF), and MasPar Parallel Application Language; UNIX/OS with X-window, symbolic debugger, visualizers and animators.
Thinking Machines Corporation, CM-2	A bit-slice array of up to 65,536 PEs arranged as a 10-dimensional hypercube with 4×4 mesh on each vertex, up to 1M bits of memory per PE, with optional FPU shared between blocks of 32 PEs. 28 Gflops peak and 5.6 Gflops sustained.	Driven by a host of VAX, Sun, or Symbolics 3600, Lisp compiler, Fortran 90, C*, and *Lisp supported by PARIS
Active Memory Technology DAP600 Family	A fine-grain, bit-slice SIMD array of up to 4096 PEs interconnected by a square mesh with 1 K bits per PE, orthogonal and 4-neighbor links, 20 GIPS and 560 Mflops peak performance.	Provided by host VAX/VMS or UNIX Fortran-plus or APAL on DAP, Fortran 77 or C on host.

1.4 PRAM AND VLSI MODELS

Theoretical models of parallel computers are abstracted from the physical models studied in previous sections. These models are often used by algorithm designers and VLSI device/chip developers. The ideal models provide a convenient framework for developing parallel algorithms without worry about the implementation details or physical constraints.

The models can be applied to obtain theoretical performance bounds on parallel computers or to estimate VLSI complexity on chip area and execution time before the chip is fabricated. The abstract models are

also useful in scalability and programmability analysis, when real machines are compared with an idealized parallel machine without worrying about communication overhead among processing nodes.

1.4.1 Parallel Random-Access Machines

Theoretical models of parallel computers are presented below. We define first the time and space complexities. Computational tractability is reviewed for solving difficult problems on computers. Then we introduce the *random-access machine* (RAM), *parallel random-access machine* (PRAM), and variants of PRAMs. These complexity models facilitate the study of asymptotic behavior of algorithms implementable on parallel computers.

Time and Space Complexities The complexity of an algorithm for solving a problem of size s on a computer is determined by the execution time and the storage space required. The time complexity is a function of the problem size. The *time complexity* function in order notation is the *asymptotic time complexity* of the algorithm. Usually, the worst-case time complexity is considered. For example, a time complexity $g(s)$ is said to be $O(f(s))$, read "order $f(s)$", if there exist positive constants c_1, c_2 and s_0 such that $c_1 f(s) \leq g(s) \leq c_2 f(s)$, for all nonnegative values of $s > s_0$.

The *space complexity* can be similarly defined as a function of the problem size s. The *asymptotic space complexity* refers to the data storage of large problems. Note that the program (code) storage requirement and the storage for input data are not considered in this.

The time complexity of a serial algorithm is simply called *serial complexity*. The time complexity of a parallel algorithm is called *parallel complexity*. Intuitively, the parallel complexity should be lower than the serial complexity, at least asymptotically. We consider only *deterministic algorithms*, in which every operational step is uniquely defined in agreement with the way programs are executed on real computers.

A *nondeterministic algorithm* contains operations resulting in one outcome from a set of possible outcomes. There exist no real computers that can execute nondeterministic algorithms. Therefore, all algorithms (or machines) considered in this book are deterministic, unless otherwise noted.

NP-Completeness An algorithm has a *polynomial complexity* if there exists a polynomial $p(s)$ such that the time complexity is $O(p(s))$ for problem size s. The set of problems having polynomial-complexity algorithms is called *P-class* (for polynomial class). The set of problems solvable by nondeterministic algorithms in polynomial time is called *NP-class* (for nondeterministic polynomial class).

Since deterministic algorithms are special cases of the nondeterministic ones, we know that P $\subset$ NP. The P-class problems are computationally *tractable*, while the NP – P-class problems are *intractable*. But we do not know whether P = NP or P $\neq$ NP. This is still an open problem in computer science.

To simulate a nondeterministic algorithm with a deterministic algorithm may require exponential time. Therefore, intractable NP-class problems are also said to have exponential-time complexity.

Example 1.4 Polynomial- and exponential-complexity algorithms

Polynomial-complexity algorithms are known for sorting n numbers in $O(n \log n)$ time and for multiplication of two $n \times n$ matrices in $O(n^3)$ time. Therefore, both problems belong to the P-class.

Nonpolynomial algorithms have been developed for the traveling salesperson problem with complexity $O(n^2 2^n)$ and for the knapsack problem with complexity $O(2^{n/2})$. These complexities are *exponential*, greater than the polynomial complexities. So far, deterministic polynomial algorithms have not been found for these problems. Therefore, these exponential-complexity problems belong to the NP-class.

Most computer scientists believe that $P \neq NP$. This leads to the conjecture that there exists a subclass, called *NP-complete* (NPC) problems, such that $NPC \subset NP$ but $NPC \cap P = \phi$ (Fig. 1.13). In fact, it has been proved that if any NP-complete problem is polynomial-time solvable, then one can conclude $P = NP$. Thus NP-complete problems are considered the hardest ones to solve. Only approximation algorithms can be derived for solving the NP-complete problems in polynomial time.

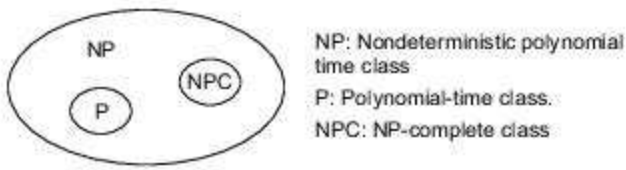


Fig. 1.13 The relationships conjectured among the NP, P, and NPC classes of computational problems

PRAM Models Conventional uniprocessor computers have been modeled as *random access machines* (RAM) by Sheperdson and Sturgis (1963). A *parallel random-access machine* (PRAM) model has been developed by Fortune and Wyllie (1978) for modeling idealized parallel computers with zero synchronization or memory access overhead. This PRAM model will be used for parallel algorithm development and for scalability and complexity analysis.

An n-processor PRAM (Fig. 1.14) has a globally addressable memory. The shared memory can be distributed among the processors or centralized in one place. The n processors — also called *processing elements* (PEs)—operate on a synchronized read-memory, compute, and write-memory cycle. With shared memory, the model must specify how concurrent read and concurrent write of memory are handled. Four memory-update options are possible:

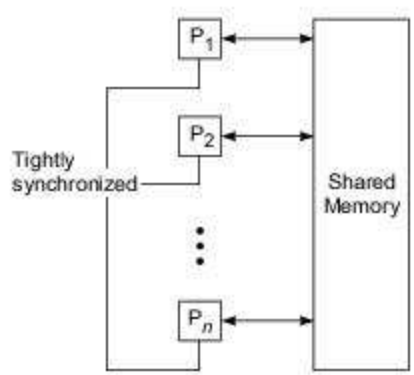


Fig. 1.14 PRAM model of a multiprocessor system with shared memory, on which all n processors operate in lockstep in memory access and program execution operations. Each processor can access any memory location in unit time

- *Exclusive read* (ER)—This allows at most one processor to read from any memory location in each cycle, a rather restrictive policy.
- *Exclusive write* (EW)—This allows at most one processor to write into a memory location at a time.
- *Concurrent read* (CR)—This allows multiple processors to read the same information from the same memory cell in the same cycle.
- *Concurrent write* (CW)—This allows simultaneous writes to the same memory location. In order to avoid confusion, some policy must be set up to resolve the write conflicts.

Various combinations of the above options lead to several variants of the PRAM model as specified below. Since CR does not create a conflict problem, variants differ mainly in how they handle the CW conflicts.

PRAM Variants Described below are four variants of the PRAM model, depending on how the memory reads and writes are handled.

(1) *EREW-PRAM model*—This model forbids more than one processor from reading or writing the same memory cell simultaneously (Snir, 1982; Karp and Ramachandran, 1988). This is the most restrictive PRAM model proposed.
(2) *CREW-PRAM model*—The write conflicts are avoided by mutual exclusion. Concurrent reads to the same memory location are allowed.
(3) *ERCW-PRAM model*—This allows exclusive read or concurrent writes to the same memory location.
(4) *CRCW-PRAM model*—This model allows either concurrent reads or concurrent writes to the same memory location.

Example 1.5 Multiplication of two $n \times n$ matrices in $O(\log n)$ time on a PRAM with $n^3/\log n$ processors (Viktor Prasanna, 1992)

Let A and B be the input matrices. Assume n^3 PEs are available initially. We later reduce the number of PEs to $n^3/\log n$. To visualize the algorithm, assume the memory is organized as a three-dimensional array with inputs A and B stored in two planes. Also, for the sake of explanation, assume a three-dimensional indexing of the PEs. PE(i, j, k), $0 \le k \le n-1$ are used for computing the (i, j)th entry of the output matrix, $0 \le i, j \le n-1$, and n is a power of 2.

In step 1, n product terms corresponding to each output are computed using n PEs in O(1) time. In step 2, these are added to produce an output in $O(\log n)$ time.

The total number of PEs used is n^3. The result is available in $C(i, j, 0)$, $0 \le i, j \le n-1$. Listed below are programs for each PE(i, j, k) to execute. All n^3 PEs operate in parallel for n^3 multiplications. But at most $n^3/2$ PEs are busy for $(n^3 - n^2)$ additions. Also, the PRAM is assumed to be CREW.

Step 1

1. Read $A(i, k)$
2. Read $B(k, j)$

3. Compute $A(i, k) \times B(k, j)$
4. Store in $C(i, j, k)$

Step 2

1. $\ell \leftarrow n$
2. Repeat
   $\ell \leftarrow \ell/2$
   if $(k < \ell)$ then
   begin
   Read $C(i, j, k)$
   Read $C(i, j, k + \ell)$
   Compute $C(i, j, k) + C(i, j, k + \ell)$
   Store in $C(i, j, k)$
   end

until $(\ell = 1)$

To reduce the number of PEs to $n^3/\log n$, use a PE array of size $n \times n \times n/\log n$. Each PE is responsible for computing $\log n$ product terms and summing them up. Step 1 can be easily modified to produce $n/\log n$ partial sums, each consisting of $\log n$ multiplications and $(\log n - 1)$ additions. Now we have an array $C(i, j, k)$, $0 \le i, j \le n - 1$, $0 \le k \le n/\log n - 1$, which can be summed up in $\log(n/\log n)$ time. Combining the time spent in step 1 and step 2, we have a total execution time $2 \log n - 1 + \log(n/\log n) = O(\log n)$ for large n.

Discrepancy with Physical Models PRAM models idealize parallel computers, in which all memory references and program executions by multiple processors are synchronized without extra cost. In reality, such parallel machines do not exist. An SIMD machine with shared memory is the closest architecture modeled by PRAM. However, PRAM allows different instructions to be executed on different processors simultaneously. Therefore, PRAM really operates in synchronized MIMD mode with a shared memory.

Among the four PRAM variants, the EREW and CRCW are the most popular models used. In fact, every CRCW algorithm can be simulated by an EREW algorithm. The CRCW algorithm runs faster than an equivalent EREW algorithm. It has been proved that the best n-processor EREW algorithm can be no more than $O(\log n)$ times slower than any n-processor CRCW algorithm.

The CREW model has received more attention in the literature than the ERCW model. For our purposes, we will use the CRCW-PRAM model unless otherwise stated. This particular model will be used in defining scalability in Chapter 3.

For complexity analysis or performance comparison, various PRAM variants offer an ideal model of parallel computers. Therefore, computer scientists use the PRAM model more often than computer engineers. In this book, we design parallel/vector computers using physical architectural models rather than PRAM models.

The PRAM model will be used for scalability and performance studies in Chapter 3 as a theoretical reference machine. PRAM models can indicate upper and lower bounds on the performance of real parallel computers.

1.4.2 VLSI Complexity Model

Parallel computers rely on the use of VLSI chips to fabricate the major components such as processor arrays, memory arrays, and large-scale switching networks. An AT^2 model for two-dimensional VLSI chips

is presented below, based on the work of Clark Thompson (1980). Three lower bounds on VLSI circuits are interpreted by Jeffrey Ullman (1984). The bounds are obtained by setting limits on memory, I/O, and communication for implementing parallel algorithms with VLSI chips.

The AT^2 Model Let A be the chip area and T be the latency for completing a given computation using a VLSI circuit chip. Let s by the problem size involved in the computation. Thompson stated in his doctoral thesis that for certain computations, there exists a lower bound $f(s)$ such that

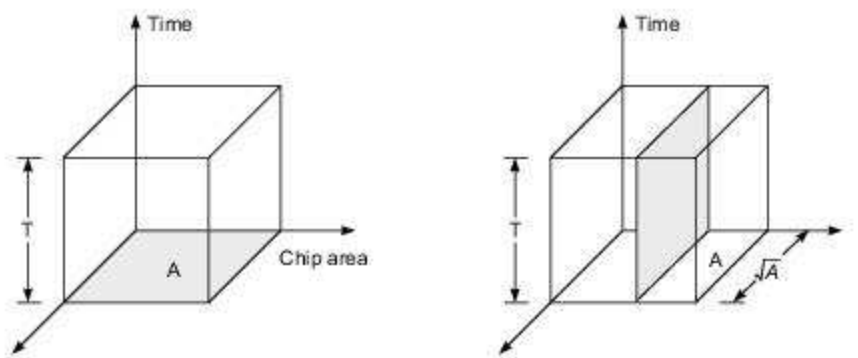
$$A \times T^2 \geq O(f(s)) \tag{1.6}$$

The chip area A is a measure of the chip's complexity. The latency T is the time required from when inputs are applied until all outputs are produced for a single problem instance. Figure 1.15 shows how to interpret the AT^2 complexity results in VLSI chip development. The chip is represented by the base area in the two horizontal dimensions. The vertical dimension corresponds to time. Therefore, the three-dimensional solid represents the history of the computation performed by the chip.

Memory Bound on Chip Area There are many computations which are memory-bound, due to the need to process large data sets. To implement this type of computation in silicon, one is limited by how densely information (bit cells) can be placed on the chip. As depicted in Fig. 1.15a, the memory requirement of a computation sets a lower bound on the chip area A.

The amount of information processed by the chip can be visualized as information flow upward across the chip area. Each bit can flow through a unit area of the horizontal chip slice. Thus, the chip area bounds the amount of memory bits stored on the chip.

I/O Bound on Volume AT The volume of the rectangular cube is represented by the product AT. As information flows through the chip for a period of time T, the number of input bits cannot exceed the volume AT, as demonstrated in Fig. 1.15a.



(a) Memory-limited bound on chip area A and I/O-limited bound on chip history represented by the volume AT

(b) Communication-limited bound on the bisection $\sqrt{A}T$

Fig. 1.15 The AT^2 complexity model of two-dimensional VLSI chips

The area A corresponds to data into and out of the entire surface of the silicon chip. This areal measure sets the maximum I/O limit rather than using the peripheral I/O pads as seen in conventional chips. The height T of the volume can be visualized as a number of snapshots on the chip, as computing time elapses. The volume represents the amount of information flowing through the chip during the entire course of the computation.

***Bisection Communication Bound,* $\sqrt{AT}$** Figure 1.15b depicts a communication limited lower bound on the bisection area $\sqrt{AT}$. The bisection is represented by the vertical slice cutting across the shorter dimension of the chip area. The distance of this dimension is $\sqrt{A}$ for a square chip. The height of the cross section is T.

The bisection area represents the maximum amount of information exchange between the two halves of the chip circuit during the time period T. The cross-section area $\sqrt{AT}$ limits the communication bandwidth of a computation. VLSI complexity theoreticians have used the square of this measure, AT^2, to which the lower bound applies, as seen in Eq. 1.6.

Example 1.6 VLSI chip implementation of a matrix multiplication algorithm (Viktor Prasanna, 1992)

This example shows how to estimate the chip area A and compute time T for $n \times n$ matrix multiplication $C = A \times B$ on a mesh of processing elements (PEs) with a broadcast bus on each row and each column. The 2-D mesh architecture is shown in Fig. 1.16. Inter-PE communication is done through the broadcast buses. We want to prove the bound $AT^2 = O(n^4)$ by developing a parallel matrix multiplication algorithm with time $T = O(n)$ in using the mesh with broadcast buses. Therefore, we need to prove that the chip area is bounded by $A = O(n^2)$.

Each PE occupies a unit area, and the broadcast buses require $O(n^2)$ wire area. Thus the total chip area needed is $O(n^2)$ for an $n \times n$ mesh with broadcast buses. We show next that the $n \times n$ matrix multiplication can be performed on this mesh chip in $T = O(n)$ time. Denote the PEs as PE(i, j), $0 \le i, j \le n - 1$.

Initially the input matrix elements $A(i, j)$ and $B(i, j)$ are stored in PE(i, j) with no duplicated data. The memory is distributed among all the PEs. Each PE can access only its own local memory. The following parallel algorithm shows how to perform the dot-product operations in generating all the output elements $C(i, j) = \sum_{k=0}^{n-1} A(i, k) \times B(k, j)$ for $0 \le i, j \le n - 1$.

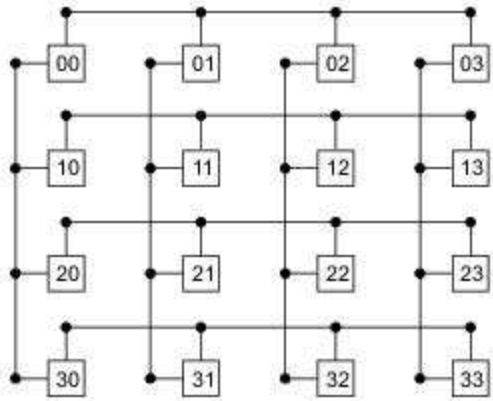


Fig. 1.16 A 4 × 4 mesh of processing elements (PEs) with broadcast buses on each row and on each column (Courtesy of Prasanna Kumar and Raghavendra; reprinted from *Journal of Parallel and Distributed Computing*, April 1987)

```
        Doall 10 for 0 ≤ i, j ≤ n − 1
10          PE(i, j) sets C(i, j) to 0 /Initialization/
        Do 50 for 0 ≤ k ≤ n − 1
            Doall 20 for 0 ≤ i ≤ n − 1
20              PE(i, k) broadcasts A(i, k) along its row bus
            Doall 30 for 0 ≤ j ≤ n − 1
30              PE(k, j) broadcasts B(k, j) along its column bus
                /PE(i, j) now has A(i, k) and B(k, j), 0 ≤ i, j ≤ n − 1/
            Doall 40 for 0 ≤ i, j ≤ n − 1
40              PE(i, j) computes C(i, j) ← C(i, j) + A(i, k) × B(k, j)
50      Continue
```

The above algorithm has a sequential loop along the dimension indexed by k. It takes n time units (iterations) in this k-loop. Thus, we have $T = O(n)$. Therefore, $AT^2 = O(n^2)$. $(O(n))^2 = O(n^4)$.

1.5 ARCHITECTURAL DEVELOPMENT TRACKS

The architectures of most existing computers follow certain development tracks. Understanding features of various tracks provides insights for new architectural development. We look into six tracks to be studied in later chapters. These tracks are distinguished by similarity in computational models and technological bases. We also review a few early representative systems in each track.

1.5.1 Multiple-Processor Tracks

Generally speaking, a multiple-processor system can be either a shared-memory multiprocessor or a distributed-memory multicomputer as modeled in Section 1.2. Bell listed these machines at the leaf nodes of

the taxonomy tree (Fig. 1.10). Instead of a horizontal listing, we show a historical development along each important track of the taxonomy.

Shared-Memory Track Figure 1.17a shows a track of multiprocessor development employing a single address space in the entire system. The track started with the C.mmp system developed at Carnegie-Mellon University (Wulf and Bell, 1972). The C.mmp was an UMA multiprocessor. Sixteen PDP 11/40 processors were interconnected to 16 shared-memory modules via a crossbar switch. A special interprocessor interrupt bus was provided for fast interprocess communication, besides the shared memory. The C.mmp project pioneered shared-memory multiprocessor development, not only in the crossbar architecture but also in the multiprocessor operating system (Hydra) development.

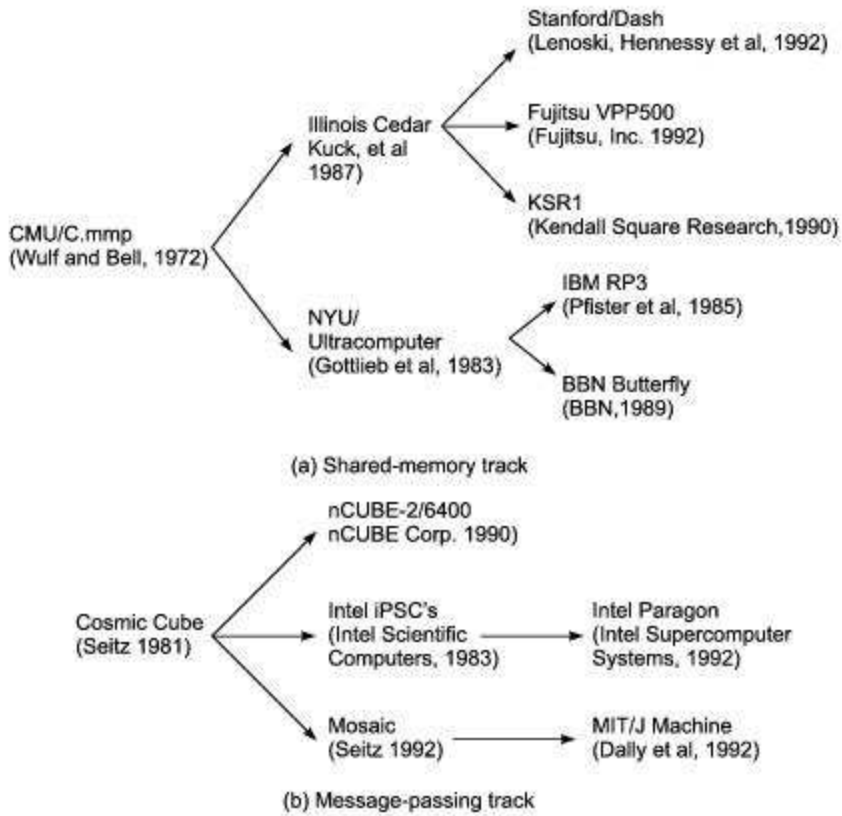


Fig. 1.17 Two multiple-processor tracks with and without shared memory

Both the NYU Ultracomputer project (Gottlieb et al., 1983) and the Illinois Cedar project (Kuck et al., 1987) were developed with a single address space. Both systems used multistage networks as a system interconnect. The major achievements in the Cedar project were in parallel compilers and performance benchmarking experiments. The Ultracomputer developed the combining network for fast synchronization among multiple processors, to be studied in Chapter 7.

The Stanford Dash (Lenoski, Hennessy et al., 1992) was a NUMA multiprocessor with distributed memories forming a global address space. Cache coherence was enforced with distributed directories. The KSR-1 was a typical COMA model. The Fujitsu VPP 500 was a 222-processor system with a crossbar interconnect. The shared memories were distributed to all processor nodes. We will study the Dash and the KSR-1 in Chapter 9 and the VPP500 in Chapter 8.

Following the Ultracomputer are two large-scale multiprocessors, both using multistage networks but with different interstage connections to be studied in Chapters 2 and 7. Among the systems listed in Fig. 1.17a, only the KSR-1, VPP500, and BBN Butterfly (BBN Advanced Computers, 1989) were commercial products. The rest were research systems; only prototypes were built in laboratories, with a view to validate specific architectural concepts.

Message-Passing Track The Cosmic Cube (Seitz et al., 1981) pioneered the development of message-passing multicomputers (Fig. 1.17b). Since then, Intel produced a series of medium-grain hypercube computers (the iPSCs). The nCUBE 2 also assumed a hypercube configuration. A subsequent Intel system was the Paragon (1992) to be studied in Chapter 7. On the research track, the Mosaic C (Seitz, 1992) and the MIT J-Machine (Dally et al., 1992) were two fine-grain multicomputers, to be studied in Chapter 9.

1.5.2 Multivector and SIMD Tracks

The multivector track is shown in Fig. 1.18a, and the SIMD track in Fig. 1.18b, with corresponding early representative systems of each type.

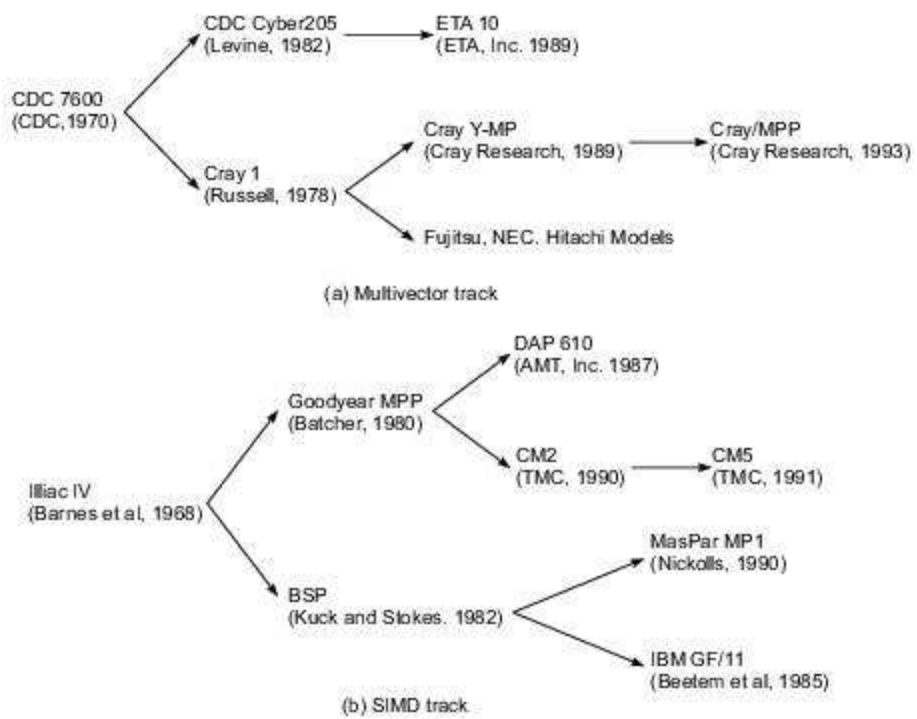


Fig. 1.18 Multivector and SIMD tracks

Both tracks are useful for concurrent scalar/vector processing. Detailed studies can be found in Chapter 8, with further discussion in Chapter 13.

Multivector Track These are traditional vector supercomputers. The CDC 7600 was the first vector dual-processor system. Two subtracks were derived from the CDC 7600. The Cray and Japanese supercomputers all followed the register-to-register architecture. Cray 1 pioneered the multivector development in 1978. The Cray/MPP was a massively parallel system with distributed shared memory, to work as a back-end accelerator engine compatible with the Cray Y-MP Series.

The other subtrack used memory-to-memory architecture in building vector supercomputers. We have identified only the CDC Cyber 205 and its successor the ETA10 here, for completeness in tracking different supercomputer architectures.

The SIMD Track The Illiac IV pioneered the construction of SIMD computers, although the array processor concept can be traced back far earlier to the 1960s. The subtrack, consisting of the Goodyear MPP, the AMT/DAP610, and the TMC/CM-2, were all SIMD machines built with bit-slice PEs. The CM-5 was a synchronized MIMD machine executing in a multiple-SIMD mode.

The other subtrack corresponds to medium-grain SIMD computers using word-wide PEs. The BSP (Kuck and Stokes, 1982) was a shared-memory SIMD machine built with 16 processors updating a group of 17 memory modules synchronously. The GF11 (Beetem et al., 1985) was developed at the IBM Watson Laboratory for scientific simulation research use. The MasPar MP1 was the only medium-grain SIMD computer to achieve production use in that time period. We will describe the CM-2, MasPar MP1, and CM-5 in Chapter 8.

1.5.3 Multithreaded and Dataflow Tracks

These two architectural tracks (Fig. 1.19) will be studied in Chapter 9. The following introduction covers only basic definitions and milestone systems built in the early years.

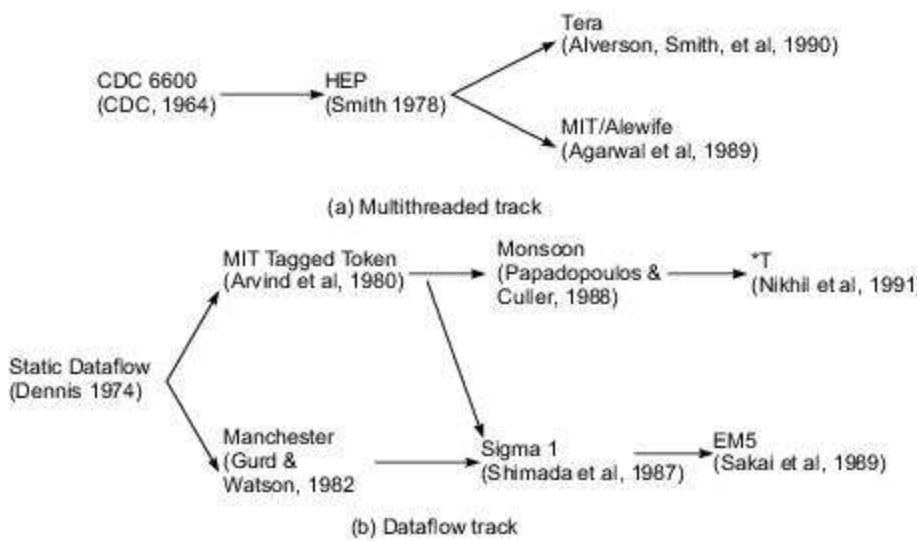


Fig. 1.19 Multithreaded and dataflow tracks

The conventional von Neumann machines are built with processors that execute a single context by each processor at a time. In other words, each processor maintains a single thread of control with its hardware resources. In a multithreaded architecture, each processor can execute multiple contexts at the same time. The term *multithreading* implies that there are multiple threads of control in each processor. Multithreading offers an effective mechanism for hiding long latency in building large-scale multiprocessors and is today a mature technology.

As shown in Fig. 1.19a, the multithreading idea was pioneered by Burton Smith (1978) in the HEP system which extended the concept of scoreboarding of multiple functional units in the CDC 6400. Subsequent multithreaded multiprocessor projects were the Tera computer (Alverson, Smith et al., 1990) and the MIT Alewife (Agarwal et al., 1989) to be studied in Section 9.4. In Chapters 12 and 13, we shall discuss the present technological factors which have led to the design of multi-threaded processors.

The Dataflow Track We will introduce the basic concepts of dataflow computers in Section 2.3. Some experimental dataflow systems are described in Section 9.5. The key idea is to use a dataflow mechanism, instead of a control-flow mechanism as in von Neumann machines, to direct the program flow. Fine-grain, instruction-level parallelism is exploited in dataflow computers.

As listed in Fig. 1.19b, the dataflow concept was pioneered by Jack Dennis (1974) with a "static" architecture. The concept later inspired the development of "dynamic" dataflow computers. A series of tagged-token architectures was developed at MIT by Arvind and coworkers. We will describe the tagged-token architecture in Section 2.3.1 and then the *T prototype (Nikhil et al., 1991) in Section 9.5.3.

Another subtrack of dynamic dataflow computer was represented by the Manchester machine (Gurd and Watson, 1982). The ETL Sigma 1 (Shimada et al., 1987) and EM5 evolved from the MIT and Manchester machines. We will study the EM5 (Sakai et al., 1989) in Section 9.5.2. These dataflow machines represent research concepts which have not had a major impact in terms of widespread use.

Summary

In science and in engineering, theory and practice go hand-in-hand, and any significant achievement invariably relies on a judicious blend of the two. In this chapter, as the first step towards a conceptual understanding of parallelism in computer architecture, we have looked at the models of parallel computer systems which have emerged over the years. We started our study with a brief look at the development of modern computers and computer architecture, including the means of classification of computer architecture, and in particular Flynn's scheme of classification.

The performance of any engineering system must be quantifiable. In the case of computer systems, we have performance measures such as processor clock rate, cycles per instruction (CPI), word size, and throughput in MIPs and/or MFLOPs, These measures have been defined, and basic relationships between them have been examined. Thus the ground has been prepared for our study in subsequent chapters of how processor architecture, system architecture, and software determine performance.

Next we looked at the architecture of shared memory multiprocessors and distributed memory multicomputers, laying the foundation for a taxonomy of MIMD computers. A key system characteristic is whether different processors in the system have access to a common shared memory and—if they do—

whether the access is uniform or non-uniform. Vector computers and SIMD computers were examined, which address the needs of highly compute-intensive scientific and engineering applications.

Over the last two or three decades, advances in VLSI technology have resulted in huge advances in computer system performance; however, the basic architectural concepts which were developed prior to the 'VLSI revolution' continue to remain valid.

Parallel random access machine (PRAM) is a theoretical model of a parallel computer. No real computer system can behave exactly like the PRAM, but at the same time the PRAM model provides us with a basis for the study of parallel algorithms and their performance in terms of time and/or space complexity. Different sub-types of the PRAM model emerge, depending on whether or not multiple processors can perform concurrent read or write operations to the shared memory.

Towards the end of the chapter, we could discern the separate architectural development tracks which have emerged over the years in computer systems. We looked at multiple-processor systems, vector processing, SIMD systems, and multi-threaded and dataflow systems. We shall see in Chapters 12 and 13 that, due to various technological factors, multi-threaded processors have gained in importance over the last decade or so.

Exercises

Problem 1.1 A 400-MHz processor was used to execute a benchmark program with the following instruction mix and clock cycle counts:

Instruction type	Instruction count	Clock cycle count
Integer arithmetic	450000	1
Data transfer	320000	2
Floating point	150000	2
Control transfer	80000	2

Determine the effective CPI, MIPS rate, and execution time for this program.

Problem 1.2 Explain how instruction set, compiler technology, CPU implementation and control, and cache and memory hierarchy affect the CPU performance and justify the effects in terms of program length, clock rate, and effective CPI.

Problem 1.3 A workstation uses a 1.5 GHz processor with a claimed 1000-MIPS rating to execute a given program mix. Assume a one-cycle delay for each memory access.

(a) What is the effective CPI of this computer?

(b) Suppose the processor is being upgraded with a 3.0 GHz clock. However, even with faster cache, two clock cycles are needed per memory access. If 30% of the instructions require one memory access and another 5% require two memory accesses per instruction, what is the performance of the upgraded processor with a compatible instruction set and equal instruction counts in the given program mix?

Problem 1.4 Consider the execution of an object code with 2×10^6 instructions on a 400-MHz processor. The program consists of four major types of instructions. The instruction mix and the number of cycles (CPI) needed for each instruction type are given below based on the result of a program trace experiment:

Instruction type	CPI	Instruction mix
Arithmetic and logic	1	60%
Load/store with cache hit	2	18%
Branch	4	12%
Memory reference with cache miss	8	10%

(a) Calculate the average CPI when the program is executed on a uniprocessor with the above trace results.

(b) Calculate the corresponding MIPS rate based on the CPI obtained in part (a).

Problem 1.5 Indicate whether each of the following statements is *true* or *false* and justify your answer with reasoning and supportive or counter examples:

(a) The CPU computations and I/O operations cannot be overlapped in a multiprogrammed computer.

(b) Synchronization of all PEs in an SIMD computer is done by hardware rather than by software as is often done in most MIMD computers.

(c) As far as programmability is concerned, shared-memory multiprocessors offer simpler interprocessor communication support than that offered by a message-passing multicomputer.

(d) In an MIMD computer, all processors must execute the same instruction at the same time synchronously.

(e) As far as scalability is concerned, multicomputers with distributed memory are more scalable than shared-memory multiprocessors.

Problem 1.6 The execution times (in seconds) of four programs on three computers are given below:

Assume that 10^9 instructions were executed in each of the four programs. Calculate the MIPS rating of each program on each of the three machines. Based on these ratings, can you draw a clear conclusion regarding the relative performance of the three computers? Give reasons if you find a way to rank them statistically.

	Execution Time (in seconds)		
Program	Computer A	Computer B	Computer C
Program 1	1	10	20
Program 2	1000	100	20
Program 3	500	1000	50
Program 4	100	800	100

Problem 1.7 Characterize the architectural operations of SIMD and MIMD computers. Distinguish between multiprocessors and multicomputers based on their structures, resource sharing, and interprocessor communications. Also, explain the differences among UMA, NUMA, and COMA, and NORMA computers.

Problem 1.8 The following code segment, consisting of six instructions, needs to be executed 64 times for the evaluation of vector arithmetic expression: $D(I) = A(I) + B(I) \times C(I)$ for $0 \le I \le 63$.

Load R1, B(I)	/R1 ← Memory (α + I)/
Load R2, C(I)	/R2 ← Memory (β + I)/
Multiply R1, R2	/R1 ← (R1) × (R2)/
Load R3, A(I)	/R3 ← Memory (γ + I)/
Add R3, R1	/R3 ← (R3) + (R1)/
Store D(I), R3	/Memory (θ + I) ← (R3)/

where R1, R2, and R3 are CPU registers, (R1) is the content of R1, α, β, γ, and θ are the starting memory addresses of arrays B(I), C(I), A(I), and D(I), respectively. Assume four clock cycles for each Load or Store, two cycles for the Add, and eight cycles for the Multiply on either a uniprocessor or a single PE in an SIMD machine.

(a) Calculate the total number of CPU cycles needed to execute the above code segment repeatedly 64 times on an SISD uniprocessor computer sequentially, ignoring all other time delays.

(b) Consider the use of an SIMD computer with 64 PEs to execute the above vector operations

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 $1 \le p \le n^3/\log n$. 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 +→ S2 dicates antidependence from S1 to S2.

(3) *Output dependence*: Two statements are *output-dependent* if they produce (write) the same output variable. S1 o→ 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:

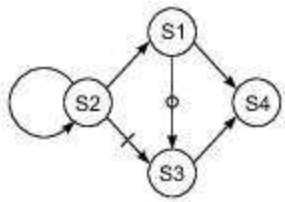
S1:	Load R1, A	/R1 ← Memory(A)/
S2:	Add R2, R1	/R2 ← (R1) + (R2)/
S3:	Move R1, R3	/R1 ← (R3)/
S4:	Store B, R1	/Memory(B) ← (R1)/

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

The following loop has *control-dependent* iterations:

```
      Do    10I = 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:

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

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.

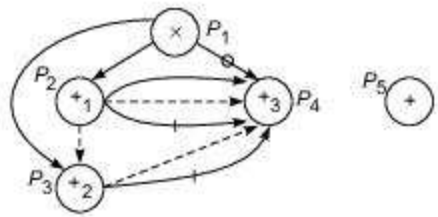
In general, a set of processes, $P_1, P_2, \ldots, P_k$, can execute in parallel if Bernstein's conditions are satisfied on a pairwise basis; that is, $P_1 \parallel P_2 \parallel P_3 \parallel \ldots \parallel P_k$ if and only if $P_i \parallel P_j$ for all $i \neq j$. This is exemplified by the following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

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

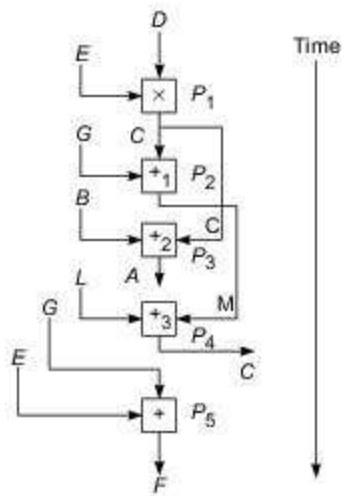
Consider the simple case in which each process is a single HLL statement. We want to detect the parallelism embedded in the following five statements labeled P_1, P_2, P_3, P_4, and P_5 in program order.

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

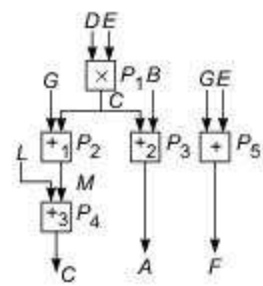
Assume that each statement requires one step to execute. No pipelining is considered here. The dependence graph shown in Fig. 2.2a demonstrates flow dependence as well as resource dependence. In sequential execution, five steps are needed (Fig. 2.2b).



(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, $P_1 \parallel P_5$, $P_2 \parallel P_3$, $P_2 \parallel P_5$, $P_5 \parallel P_3$, and $P_4 \parallel P_5$, 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.

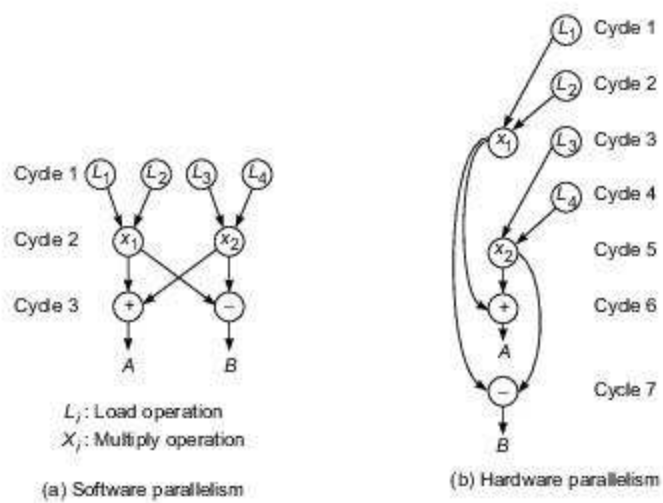


Fig. 2.3 Executing an example program by a two-issue superscalar processor

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.

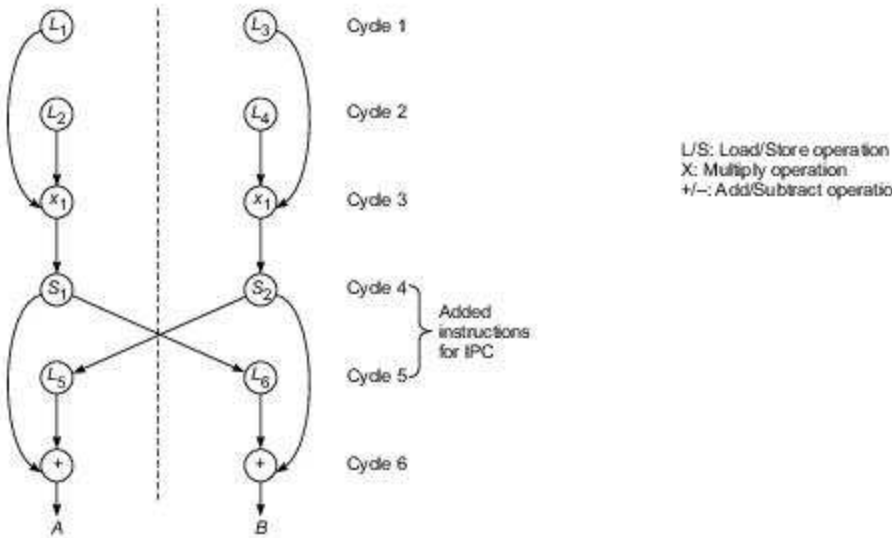


Fig. 2.4 Dual-processor execution of the program in Fig. 2.3a

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.

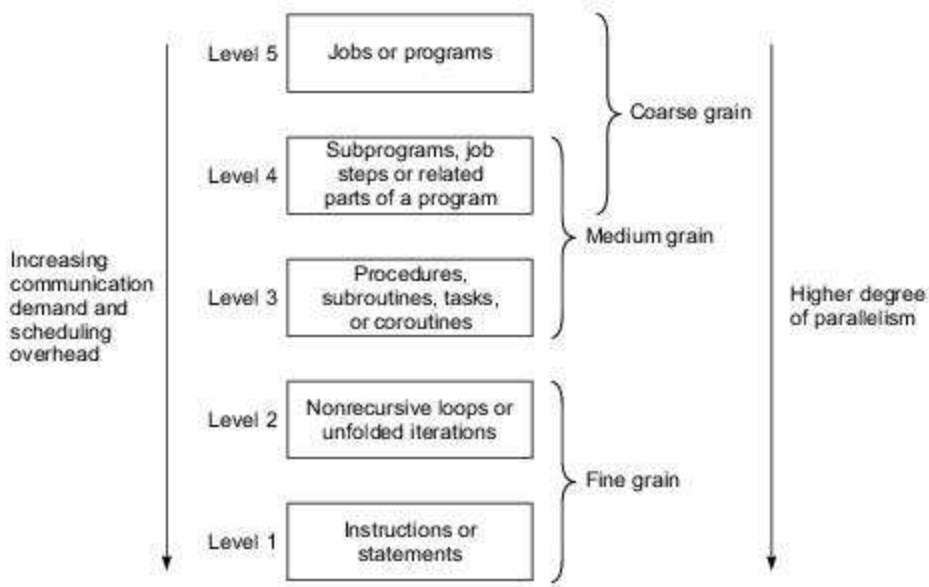


Fig. 2.5 Levels of parallelism in program execution on modern computers (Reprinted from Hwang, *Proc. IEEE*, October 1987)

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.

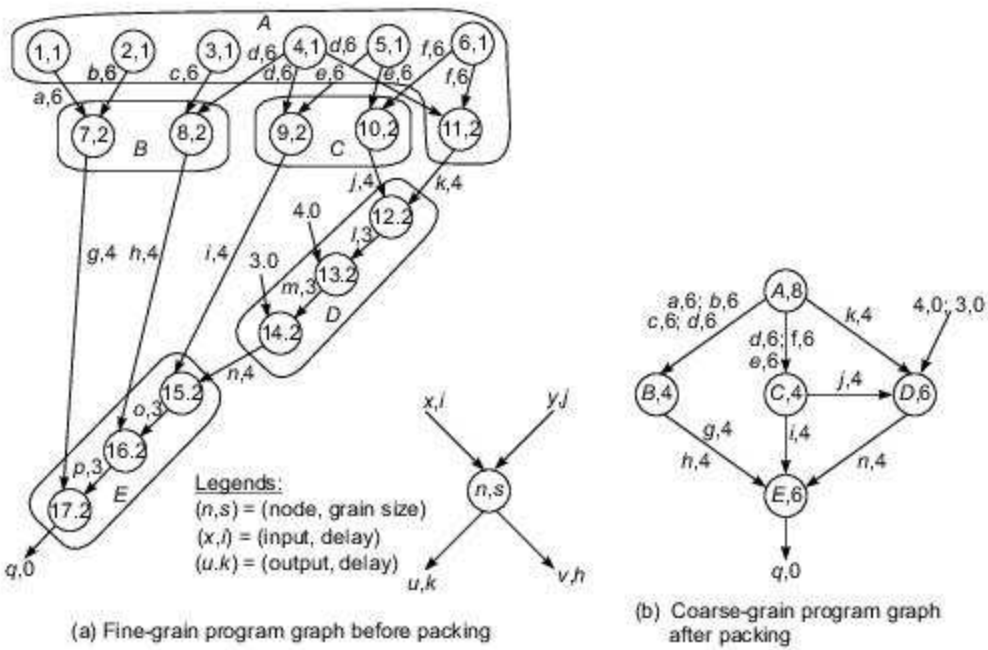
Example 2.4 Program graph before and after grain packing (Kruatrachue and Lewis, 1988)

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

End

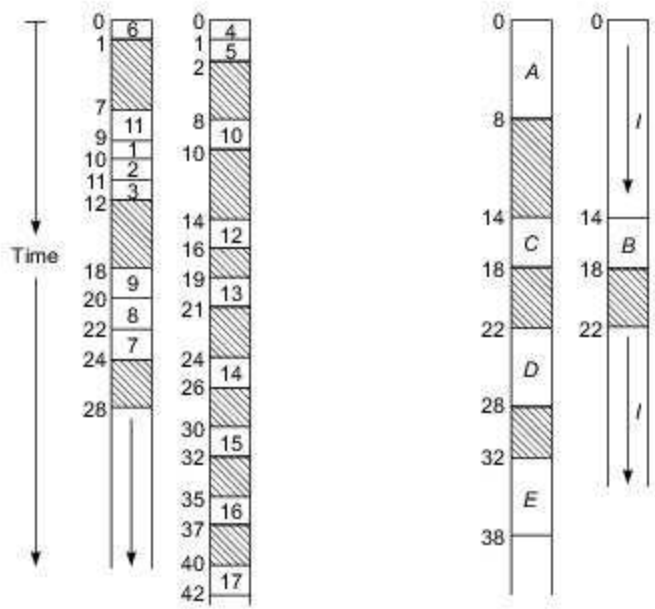
Nodes 1, 2, 3, 4, 5, and 6 are memory reference (data fetch) operations. Each takes one cycle to address and six cycles to fetch from memory. All remaining nodes (7 to 17) are CPU operations, each requiring two cycles to complete. After packing, the coarse-grain nodes have larger grain sizes ranging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

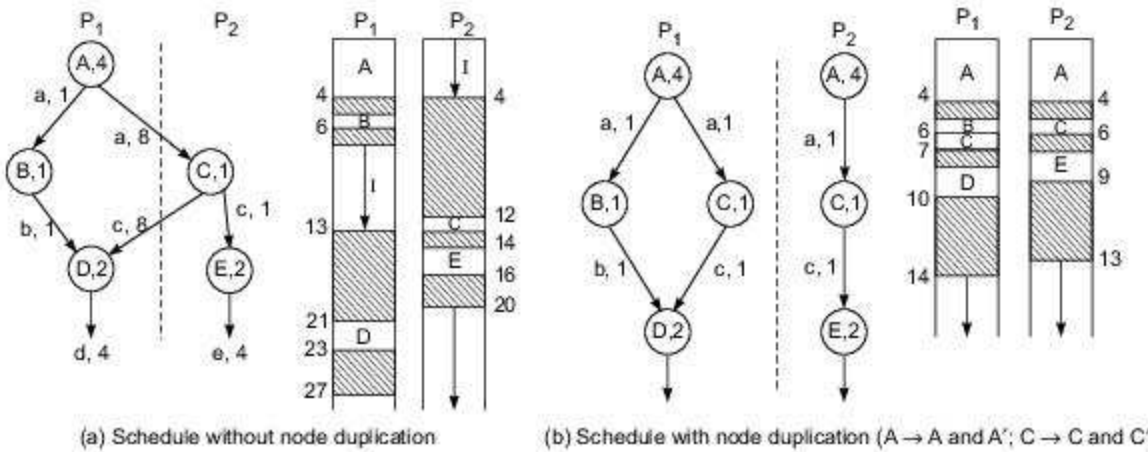
With respect to the fine-grain versus coarse-grain program graphs in Fig. 2.6, two multiprocessor schedules are shown in Fig. 2.7. The fine-grain schedule is longer (42 time units) because more communication delays were included as shown by the shaded area. The coarse-grain schedule is shorter (38 time 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:

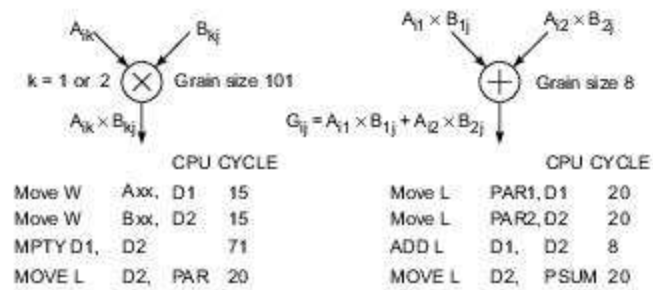
Step 1. Construct a fine-grain program graph.
Step 2. Schedule the fine-grain computation.
Step 3. Perform grain packing to produce the coarse grains.
Step 4. Generate a parallel schedule based on the packed graph.

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

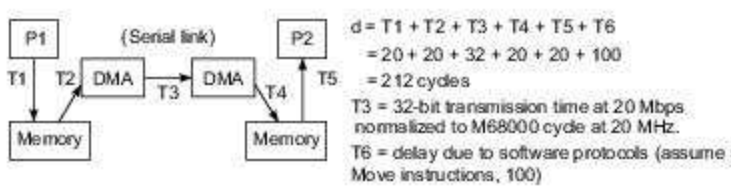
Example 2.5 Program decomposition for static multiprocessor scheduling (Kruatrachue and Lewis, 1988)

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:

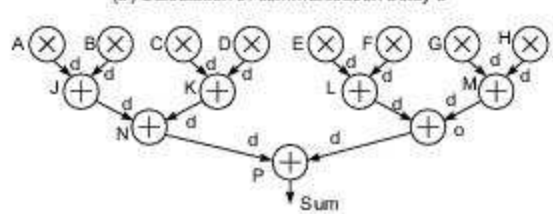
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$



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



(b) Calculation of communication delay d



(c) Fine-grain program graph

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

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

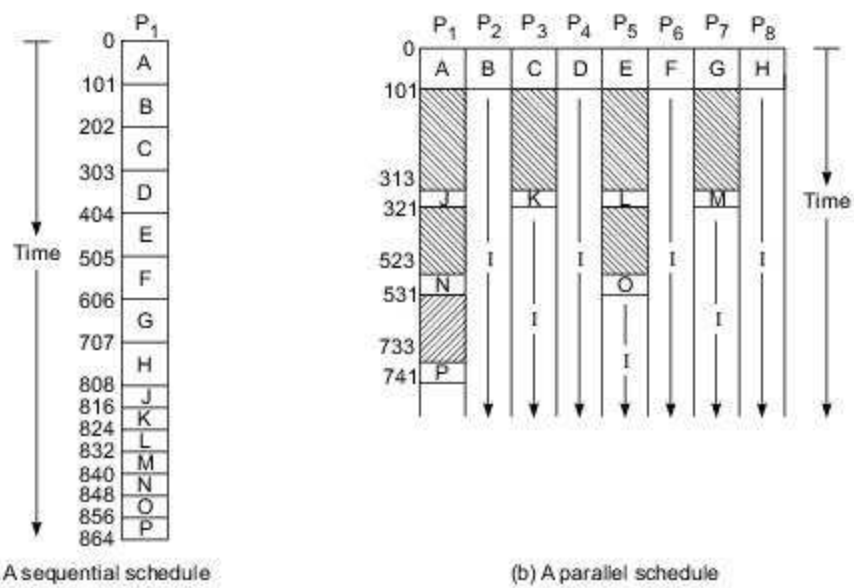
As shown in Fig. 2.9a, the eight multiplications are performed in eight ⊗ nodes, each of which has a grain size of 101 CPU cycles. The remaining seven additions are performed in a 3-level binary tree consisting of seven ⊕ nodes. Each additional node requires 8 CPU cycles.

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.

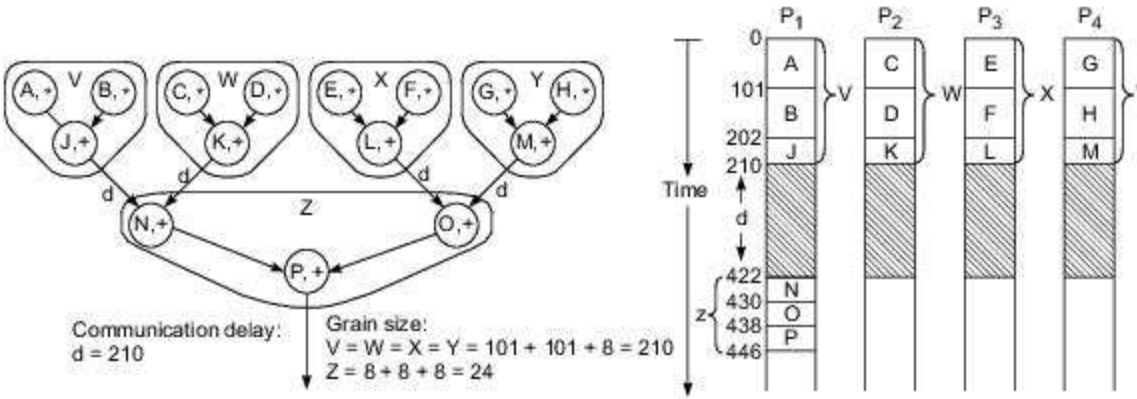


(a) A sequential schedule (b) A parallel schedule

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 $n \times n$ routing network. The entire system supports pipelined dataflow operations in all n PEs. Inter-PE communications are done through the pipelined routing network.

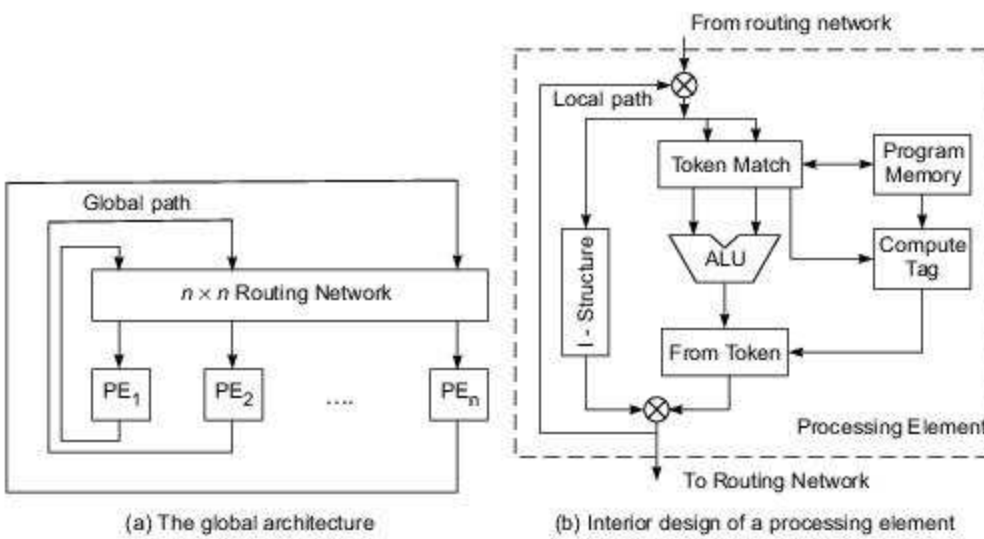


Fig. 2.12 The MIT tagged-token dataflow computer (adapted from Arvind and Iannucci, 1986 with permission)

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.

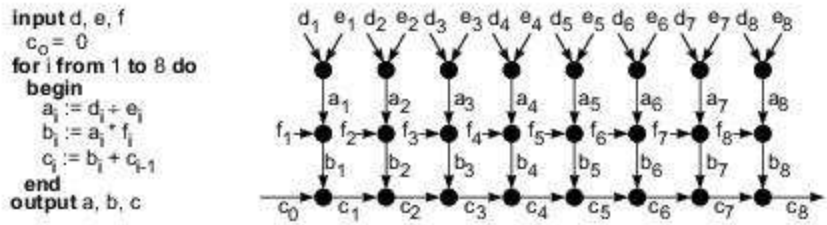
Example 2.6 Comparison of dataflow and control-flow computers (Gajski, Padua, Kuck, and Kuhn, 1982)

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.

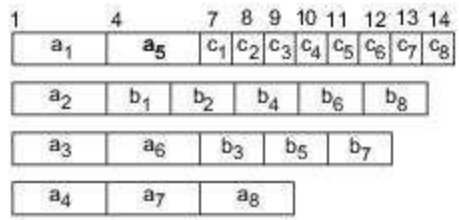
Figure 2.13d shows the execution of the same set of computations on a conventional multiprocessor using shared memory to hold the intermediate results (s_i and t_i for $i = 1, 2, 3, 4$). Note that no shared memory is used in the dataflow implementation. The example does not show any time advantage of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

The theoretical minimum time is 13 cycles along the critical path $a_1 b_1 c_1 c_2 \ldots c_8$. 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.

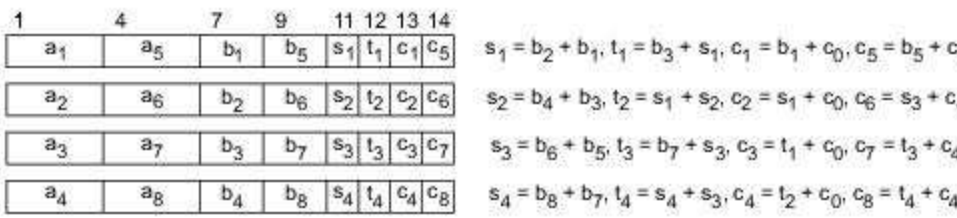


(a) A sample program and its dataflow graph

(b) Sequential execution on a uniprocessor in 48 cycles



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



(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 $\times$ 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: $a \rightarrow b$, $b \rightarrow c$, $c \rightarrow a$, $d \rightarrow e$, and $e \rightarrow d$ 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).

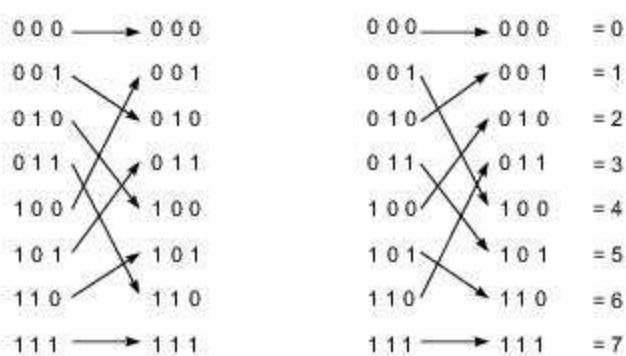


Fig. 2.14 Perfect shuffle and its inverse mapping over eight objects (Courtesy of H. Stone; reprinted with permission from *IEEE Trans. Computers*, 1971)

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.

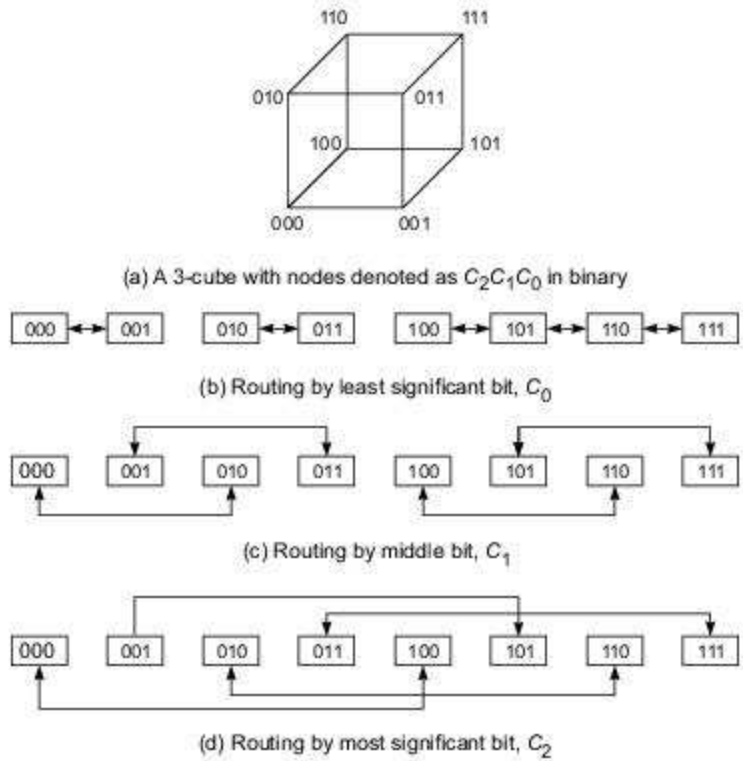


Fig. 2.15 Three routing functions defined by a binary 3-cube

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 $D=n/2$.

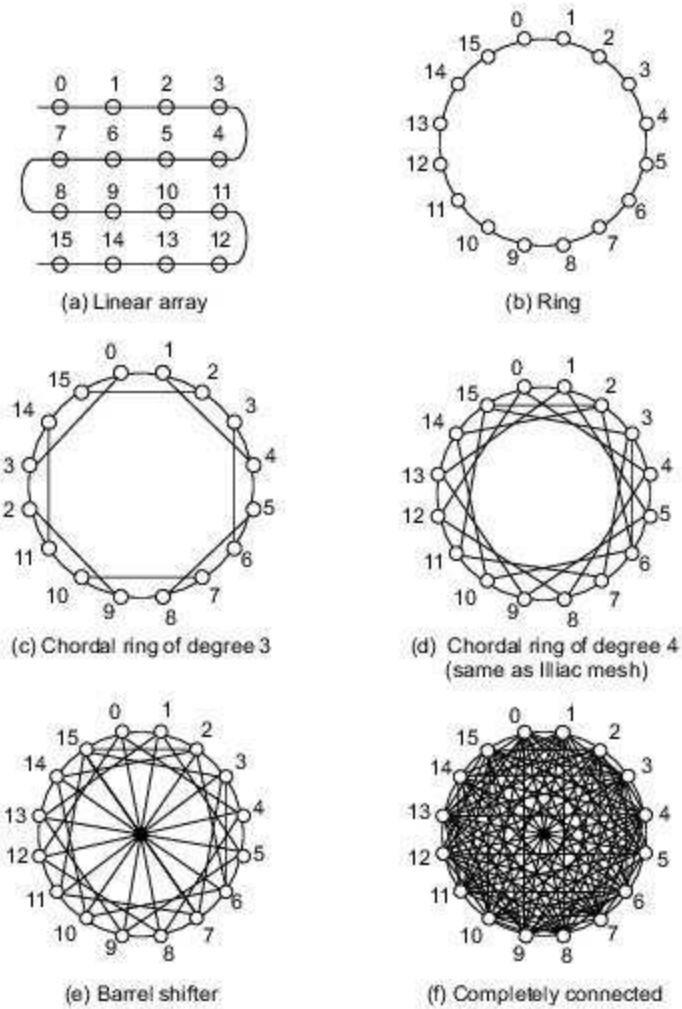


Fig. 2.16 Linear array, ring, chordal rings of degrees 3 and 4, barrel shifter, and completely connected network

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.

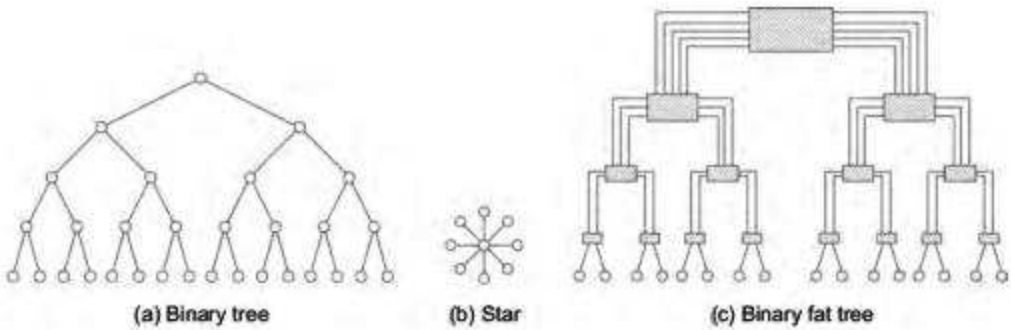


Fig. 2.17 Tree, star, and fat tree

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.

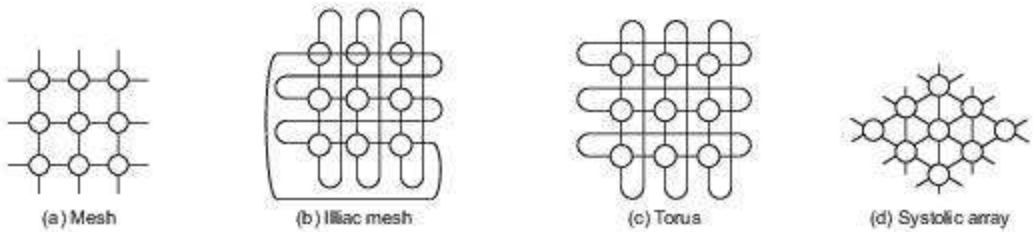


Fig. 2.18 Mesh, Illiac mesh, torus, and systolic array

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.

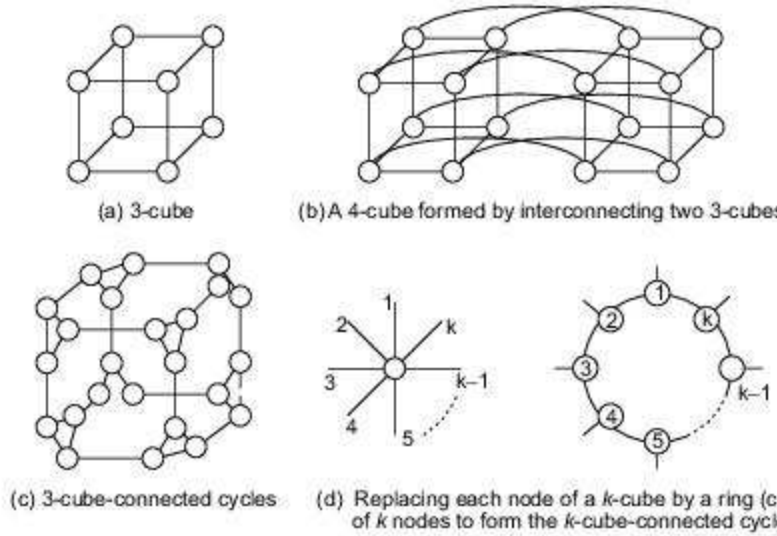


Fig. 2.19 Hypercubes and cube-connected cycles

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.

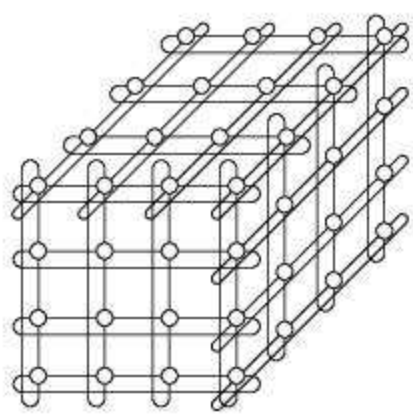


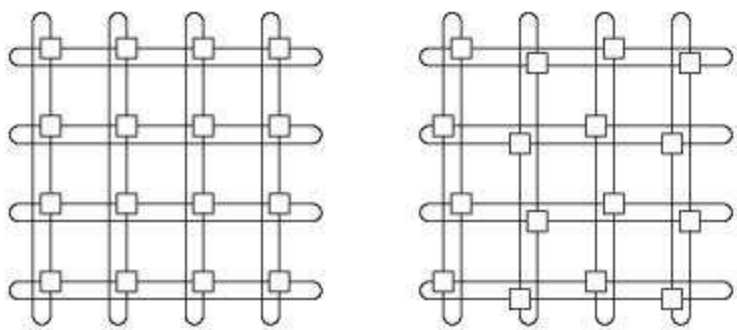
Fig. 2.20 The k-ary n-cube network shown with $k = 4$ and $n = 3$; hidden nodes or connections are not shown

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 messages 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, I*	*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 I. 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.

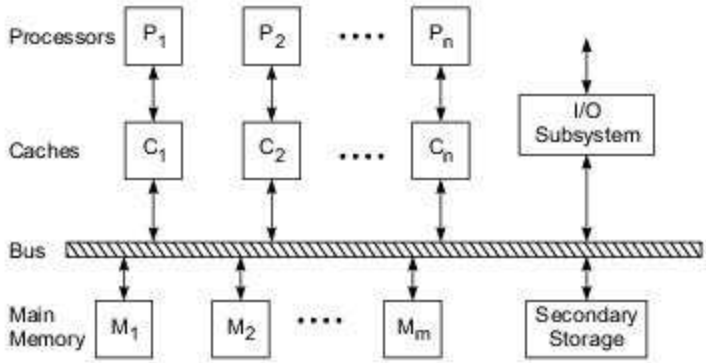


Fig. 2.22 A bus-connected multiprocessor system, such as the Sequent Symmetry S1

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.

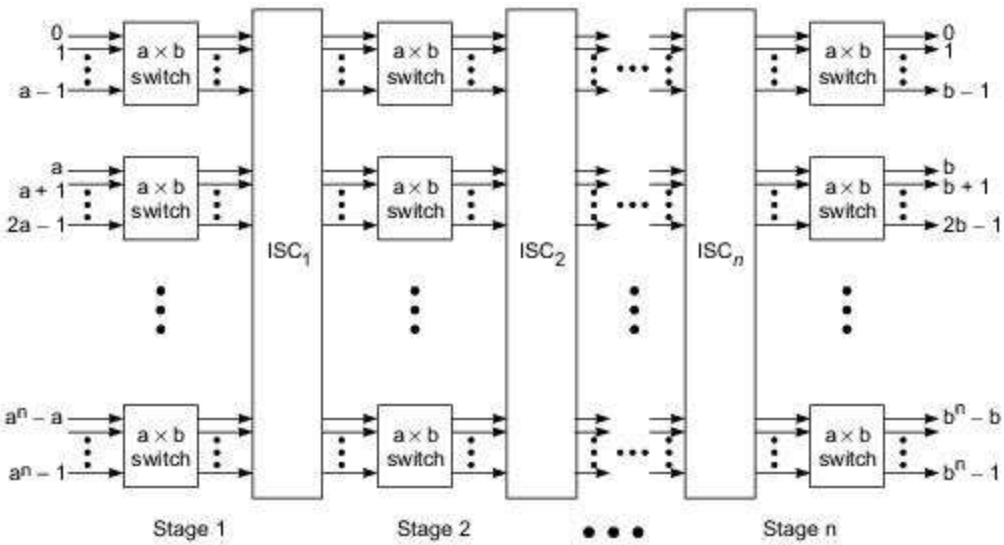


Fig. 2.23 A generalized structure of a multistage interconnection network (MIN) built with a × b switch modules and interstage connection patterns $ISC_1, ISC_2, \ldots, ISC_n$

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.

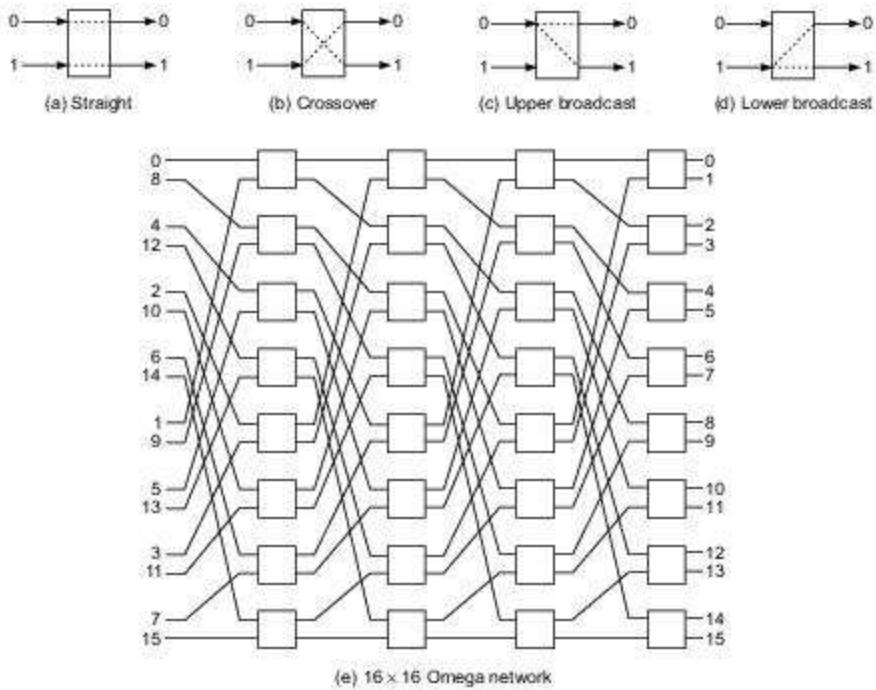


Fig. 2.24 The use of 2 × 2 switches and perfect shuffle as an interstage connection pattern to construct a 16 × 16 Omega network (Courtesy of Duncan Lawrie; reprinted with permission from *IEEE Trans. Computers*, Dec. 1975)

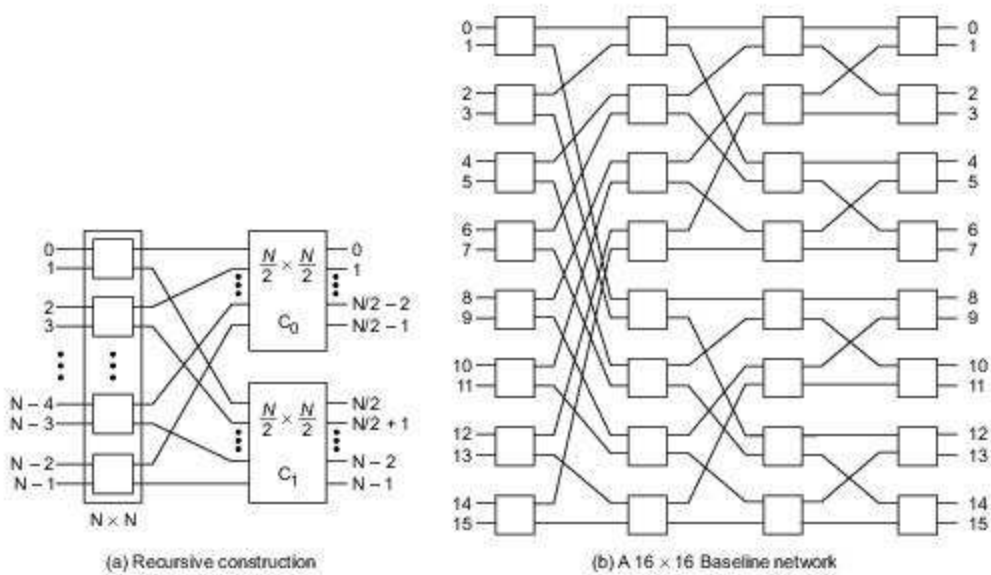
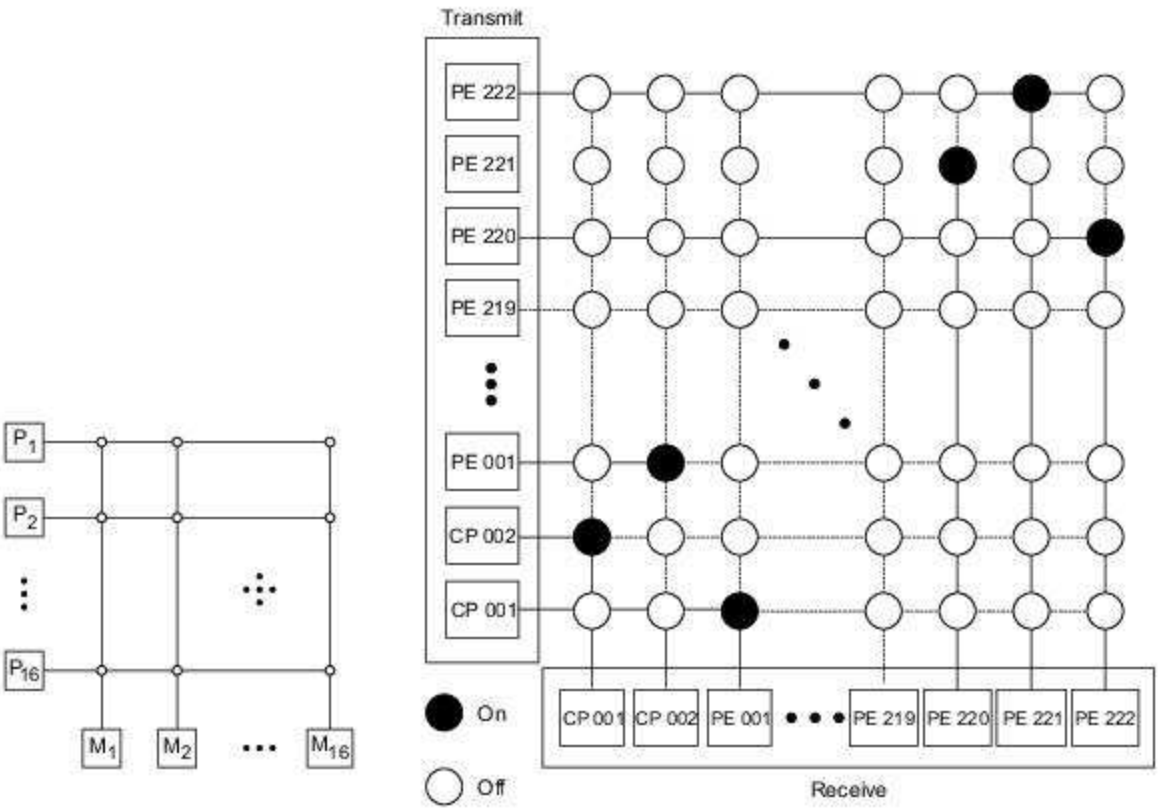


Fig. 2.25 Recursive construction of a Baseline network (Courtesy of Wu and Feng; reprinted with permission from *IEEE Trans. Computers, August 1980*)

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 VPP 500 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 ← (A + B) × (A – B)
P2:  Y ← (C + D) / (C – D)
P3:  Z ← X + Y
P4:  A ← E × F
P5:  Y ← E – Z
P6:  B ← (X – F) × 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 I the total number of links in a network. Suppose the quality of a network is measured by $(d \times D \times I)^{-1}$. 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 $(D - i + 1) / \sum_{k=1}^{D} k$, 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.

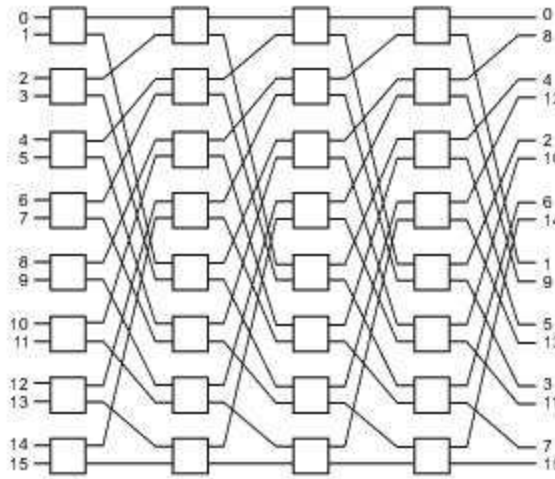


Fig. 2.27 A 16×16 Flip network (Courtesy of Ken Batcher; reprinted from *Proc. Int. Conf. Parallel Processing*, 1976)

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

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

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.

3

Principles of Scalable Performance

We study performance measures, speedup laws, and scalability principles in this chapter. Three speedup models are presented under different computing objectives and resource constraints. These include Amdahl's law (1967), Gustafson's scaled speedup (1988), and the memory-bounded speedup by Sun and Ni (1993).

The efficiency, redundancy, utilization, and quality of a parallel computation are defined, involving the interplay between architectures and algorithms. Standard performance measures and several benchmark kernels are introduced with relevant performance data.

The performance of parallel computers relies on a design that balances hardware and software. The system architects and programmers must exploit parallelism, pipelining, and networking in a balanced approach. Toward building massively parallel systems, the scalability issues must be resolved first. Fundamental concepts of scalable systems are introduced in this chapter. Case studies can be found in subsequent chapters, especially in Chapters 9 and 13.

3.1 PERFORMANCE METRICS AND MEASURES

In this section, we first study parallelism profiles and define the asymptotic speedup factor, ignoring communication latency and resource limitations. Then we introduce the concepts of system efficiency, utilization, redundancy, and quality of parallel computations. Possible tradeoffs among these performance metrics are examined in the context of cost-effectiveness. Several commonly used performance measures, MIPS, Mflops, and TPS, are formally defined.

3.1.1 Parallelism Profile in Programs

The degree of parallelism reflects the extent to which software parallelism matches hardware parallelism. We characterize below parallelism profiles, introduce the concept of average parallelism, and define an ideal speedup with infinite machine resources. Variations on the ideal speedup factor will be presented in subsequent sections from various application viewpoints and under different system limitations.

Degree of Parallelism The execution of a program on a parallel computer may use different numbers of processors at different time periods during the execution cycle. For each time period, the number of processors used to execute a program is defined as the *degree of parallelism* (DOP). This is a discrete time function, assuming only nonnegative integer values.

The plot of the DOP as a function of time is called the *parallelism profile* of a given program. For simplicity, we concentrate on the analysis of single-program profiles. Some software tools are available to trace the parallelism profile. The profiling of multiple programs in an interleaved fashion can in theory be extended from this study.

Fluctuation of the profile during an observation period depends on the algorithmic structure, program optimization, resource utilization, and run-time conditions of a computer system. The DOP was defined under the assumption of having an unbounded number of available processors and other necessary resources. The DOP may not always be achievable on a real computer with limited resources.

When the DOP exceeds the maximum number of available processors in a system, some parallel branches must be executed in chunks sequentially. However, parallelism still exists within each chunk, limited by the machine size. The DOP may be also limited by memory and by other nonprocessor resources. We consider only the limit imposed by processors in our discussions on speedup models.

Average Parallelism In what follows, we consider a parallel computer consisting of n homogeneous processors. The maximum parallelism in a profile is m. In the ideal case, $n >> m$. The *computing capacity* Δ of a single processor is approximated by the execution rate, such as MIPS or Mflops, without considering the penalties from memory access, communication latency, or system overhead. When i processors are busy during an observation period, we have DOP = i.

The total amount of work W (instructions or computations) performed is proportional to the area under the profile curve:

$$W = \Delta \int_{t_1}^{t_2} DOP(t)\, dt \tag{3.1}$$

This integral is often computed with the following discrete summation:

$$W = \Delta \sum_{i=1}^{m} i \cdot t_i \tag{3.2}$$

where t_i is the total amount of time that DOP = i and $\sum_{i=1}^{m} t_i = t_2 - t_1$ is the total elapsed time.

The *average parallelism* A is computed by

$$A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOP(t)\, dt \tag{3.3}$$

In discrete form, we have

$$A = \left(\sum_{i=1}^{m} i \cdot t_i \right) \Big/ \left(\sum_{i=1}^{m} t_i \right) \tag{3.4}$$

Example 3.1 Parallelism profile and average parallelism of a divide-and-conquer algorithm (Sun and Ni, 1993)

As illustrated in Fig. 3.1, the parallelism profile of a divide-and-conquer algorithm increases from 1 to its peak value at $m = 8$ and then decreases to 0 during the observation period (t_1, t_2).

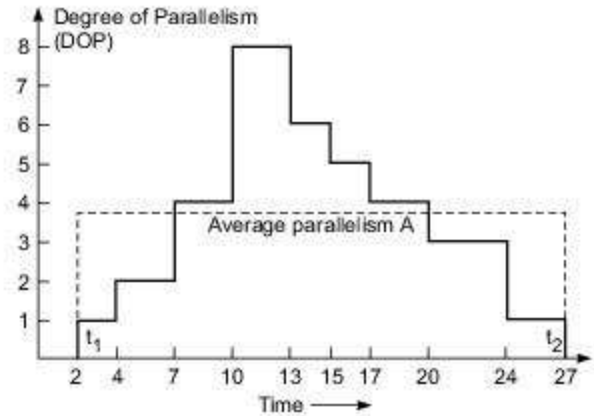


Fig. 3.1 Parallelism profile of a divide-and-conquer algorithm

In Fig. 3.1, the average parallelism $A = (1 \times 5 + 2 \times 3 + 3 \times 4 + 4 \times 6 + 5 \times 2 + 6 \times 2 + 8 \times 3)/(5 + 3 + 4 + 6 + 2 + 2 + 3) = 93/25 = 3.72$. In fact, the total workload $W = A\Delta\,(t_2 - t_1)$, and A is an upper bound of the asymptotic speedup to be defined below.

Available Parallelism There is a wide range of potential parallelism in application programs. Engineering and scientific codes exhibit a high DOP due to data parallelism. Manoj Kumar (1988) has reported that computation-intensive codes may execute 500 to 3500 arithmetic operations concurrently in each clock cycle in an idealized environment. Nicolau and Fisher (1984) reported that standard Fortran programs averaged about a factor of 90 parallelism available for very-long-instruction word architectures. These numbers show the optimistic side of available parallelism.

However, David Wall (1991) indicated that limits of instruction-level parallelism is around 5, rarely exceeding 7. Bulter et al. (1991) reported that when all constraints are removed, the DOP in programs may exceed 17 instructions per .cycle. If the hardware is perfectly balanced, one can sustain from 2.0 to 5.8 instructions per cycle on a superscalar processor that is reasonably designed. These numbers show the pessimistic side of available parallelism.

The above measures of available parallelism show that computation that is less numeric than that in scientific codes has relatively little parallelism even when basic block boundaries are ignored. A *basic block* is a sequence or block of instructions in a program that has a single entry and a single exit points. While compiler optimization and algorithm redesign may increase the available parallelism in an application, limiting parallelism extraction to a basic block limits the potential instruction-level parallelism to a factor of about 2 to 5 in ordinary programs. However, the DOP may be pushed to thousands in some scientific codes when multiple processors are used to exploit parallelism beyond the boundary of basic blocks.

Asymptotic Speedup Denote the amount of work executed with DOP = i as $W_i = i\Delta t_i$ or we can write $W = \sum_{i=1}^{m} W_i$. The execution time of W_i on a single processor (sequentially) is $t_i(1) = W_i/\Delta$. The execution time of W_i on k processors is $t_i(k) = W_i/k\Delta$. With an infinite number of available processors, $t_i(\infty) = W_i/i\Delta$ for $1 \le i \le m$. Thus we can write the *response time* as

$$T(1) = \sum_{i}^{m} t_i\ (1) = \sum_{i}^{m} \frac{W_i}{\Delta} \tag{3.5}$$

$$T(\infty) = \sum_{i=1}^{m} t_i \quad (\infty) = \sum_{i=1}^{m} \frac{W_i}{i\Delta} \tag{3.6}$$

The *asymptotic speedup* S_∞ is defined as the ratio of $T(1)$ to $T(\infty)$:

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} W_i / i} \tag{3.7}$$

Comparing Eqs. 3.4 and 3.7, we realize that $S_\infty = A$ in the ideal case. In general, $S_\infty \leq A$ if communication latency and other system overhead are considered. Note that both S_∞ and A are defined under the assumption $n = \infty$ or $n >> m$.

3.1.2 Mean Performance

Consider a parallel computer with n processors executing m programs in various modes with different performance levels. We want to define the mean performance of such multimode computers. With a weight distribution we can define a meaningful performance expression.

Different execution modes may correspond to scalar, vector, sequential, or parallel processing with different program parts. Each program may be executed with a combination of these modes. *Harmonic mean* performance provides an average performance across a large number of programs running in various modes.

Before we derive the harmonic mean performance expression, let us study the *arithmetic mean* performance expression first derived by James Smith (1988). The execution rate R_i for program i is measured in MIPS rate or Mflops rate, and so are the various performance expressions to be derived below.

Arithmetic Mean Performance Let $\{R_i\}$ be the execution rates of programs $i = 1, 2, \ldots, m$. The *arithmetic mean execution rate* is defined as

$$R_a = \sum_{i=1}^{m} R_i / m \tag{3.8}$$

The expression R_a assumes equal weighting $(1/m)$ on all m programs. If the programs are weighted with a distribution $\pi = \{f_i | i = 1, 2, \ldots, m\}$, we define a *weighted arithmetic mean execution rate* as follows:

$$R_a^* = \sum_{i=1}^{m} (f_i R_i) \tag{3.9}$$

Arithmetic mean execution rate is proportional to the sum of the inverses of execution times; it is not inversely proportional to the sum of execution times. Consequently, the arithmetic mean execution rate fails to represent the real times consumed by the benchmarks when they are actually executed.

Harmonic Mean Performance With the weakness of arithmetic mean performance measure, we need to develop a mean performance expression based on arithmetic mean execution time. In fact, $T_i = 1/R_i$ is the mean execution time per instruction for program i. The *arithmetic mean execution time* per instruction is defined by

$$T_a = \frac{1}{m}\sum_{i=1}^{m} T_i = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{R_i} \tag{3.10}$$

The *harmonic mean execution rate* across m benchmark programs is thus defined by the fact $R_h = 1/T_a$:

$$R_h = \frac{m}{\sum_{i=1}^{m}(1/R_i)} \tag{3.11}$$

Therefore, the harmonic mean performance is indeed related to the average execution time. With a weight distribution $\pi = \{f_i | i = 1, 2, \ldots, m\}$, we can define the *weighted harmonic mean execution rate as*:

$$R_h^* = \frac{1}{\sum_{i=1}^{m}(f_i/R_i)} \tag{3.12}$$

The above harmonic mean performance expressions correspond to the total number of operations divided by the total time. Compared to arithmetic mean, the harmonic mean execution rate is closer to the real performance.

Harmonic Mean Speedup Another way to apply the harmonic mean concept is to tie the various modes of a program to the number of processors used. Suppose a program (or a workload of multiple programs combined) is to be executed on an n-processor system. During the executing period, the program may use $i = 1, 2, \ldots, n$ processors in different time periods.

We say the program is executed in *mode i*, if i processors are used. The corresponding execution rate R_i is used to reflect the collective speed of i processors. Assume that $T_1 = 1/R_1 = 1$ is the sequential execution time on a uniprocessor with an execution rate $R_1 = 1$. Then $T_i = 1/R_i = 1/i$ is the execution time of using i processors with a combined execution rate of $R_i = i$ in the ideal case.

Suppose the given program is executed in n execution modes with a weight distribution $w = \{f_i | i = 1, 2, \ldots, n\}$. A *weighted harmonic mean speedup* is defined as follows:

$$S = T_1/T^* = \frac{1}{(\sum_{i=1}^{n} f_i/R_i)} \tag{3.13}$$

where $T^* = 1/R_h^*$ is the *weighted arithmetic mean execution time* across the n execution modes, similar to that derived in Eq. 3.12.

Example 3.2 Harmonic mean speedup for a multiprocessor operating in *n* execution modes (Hwang and Briggs, 1984)

In Fig. 3.2, we plot Eq. 3.13 based on the assumption that $T_i = 1/i$ for all $i = 1, 2, \ldots, n$. This corresponds to the ideal case in which a unit-time job is done by i processors in minimum time. The assumption can also be interpreted as $R_i = i$ because the execution rate increases i times from $R_1 = 1$ when i processors are fully utilized without waste.

The three probability distributions π_1, π_2, and π_3 correspond to three processor utilization patterns. Let $s = \sum_{i=1}^{n} i$. $\pi_1 = (1/n, 1/n, \ldots, 1/n)$ corresponds to a uniform distribution over the n execution modes, $\pi_2 = (1/s, 2/s, \ldots, n/s)$ favors using more processors, and $\pi_3 = (n/s, (n-1)/s, \ldots, 2/s, 1/s)$ favors using fewer processors.

The ideal case corresponds to the 45° dashed line. Obviously, π_2 produces a higher speedup than π_1 does. The distribution π_1 is superior to the distribution π_3 in Fig. 3.2.

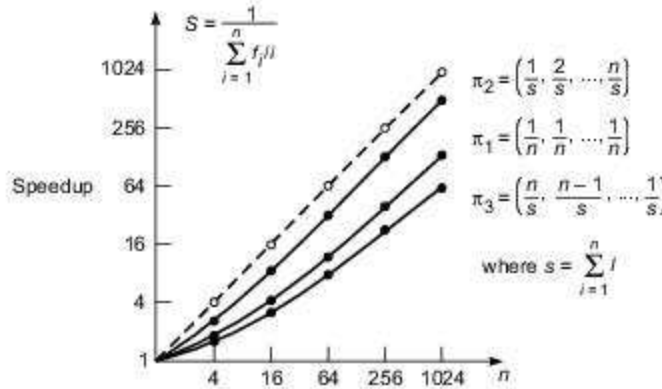


Fig. 3.2 Harmonic mean speedup performance with respect to three probability distributions: π_1 for uniform distribution, π_2 in favor of using more processors, and π_3 in favor of using fewer processors

Amdahl's Law Using Eq. 3.13, one can derive Amdahl's law as follows: First, assume $R_i = i$, $w = (\alpha, 0, 0, \ldots, 0, 1-\alpha)$; i.e., $w_1 = \alpha$, $w_n = 1 - \alpha$, and $w_i = 0$ for $i \neq 1$ and $i \neq n$. This implies that the system is used either in a pure sequential mode on one processor with a probability α, or in a fully parallel mode using n processors with a probability $1 - \alpha$. Substituting $R_1 = 1$ and $R_n = n$ and w into Eq. 3.13, we obtain the following speedup expression:

$$S_n = \frac{n}{1 + (n-1)\alpha} \tag{3.14}$$

This is known as Amdahl's law. The implication is that $S \to 1/\alpha$ as $n \to \infty$. In other words, under the above probability assumption, the best speedup one can expect is upper-bounded by $1/\alpha$, regardless of how many processors are employed.

In Fig. 3.3, we plot Eq. 3.14 as a function of n for four values of α. When $\alpha = 0$, the ideal speedup is achieved. As the value of α increases from 0.01 to 0.1 to 0.9, the speedup performance drops sharply.

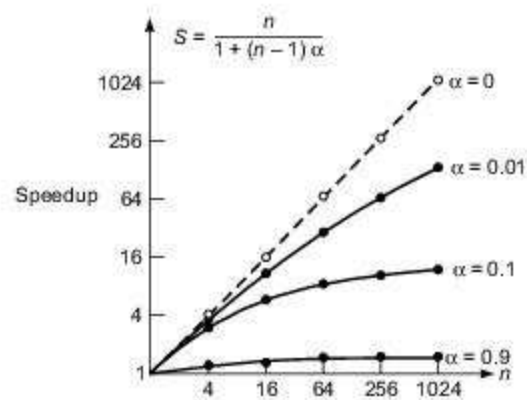


Fig. 3.3 Speedup performance with respect to the probability distribution $\pi = (\alpha, 0, \ldots, 0, 1-\alpha)$ where α is the fraction of sequential bottleneck

For many years, Amdahl's law has painted a pessimistic picture for parallel processing. That is, the system performance cannot be high as long as the serial fraction α exists. We will further examine Amdahl's law in Section 3.3.1 from the perspective of workload growth.

3.1.3 Efficiency, Utilization, and Quality

Ruby Lee (1980) has defined several parameters for evaluating parallel computations. These are fundamental concepts in parallel processing. Tradeoffs among these performance factors are often encountered in real-life applications.

System Efficiency Let $O(n)$ be the total number of unit operations performed by an n-processor system and $T(n)$ be the execution time in unit time steps. In general, $T(n) < O(n)$ if more than one operation is performed by n processors per unit time, where $n \geq 2$. Assume $T(1) = O(1)$ in a uniprocessor system. The *speedup factor* is defined as

$$S(n) = T(1)/T(n) \tag{3.15}$$

The *system efficiency* for an n-processor system is defined by

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)} \tag{3.16}$$

Efficiency is an indication of the actual degree of speedup performance achieved as compared with the maximum value. Since $1 \leq S(n) \leq n$, we have $1/n \leq E(n) \leq 1$.

The lowest efficiency corresponds to the case of the entire program code being executed sequentially on a single processor, the other processors remaining idle. The maximum efficiency is achieved when all n processors are fully utilized throughout the execution period.

Redundancy and Utilization The *redundancy* in a parallel computation is defined as the ratio of $O(n)$ to $O(1)$:

$$R(n) = O(n)/O(1) \tag{3.17}$$

This ratio signifies the extent of matching between software parallelism and hardware parallelism. Obviously $1 \leq R(n) \leq n$. The *system utilization* in a parallel computation is defined as

$$U(n) = R(n)E(n) = \frac{O(n)}{nT(n)} \tag{3.18}$$

The system utilization indicates the percentage of resources (processors, memories, etc.) that was kept busy during the execution of a parallel program. It is interesting to note the following relationships: $1/n \leq E(n) \leq U(n) \leq 1$ and $1 \leq R(n) \leq 1/E(n) \leq n$.

Quality of Parallelism The *quality* of a parallel computation is directly proportional to the speedup and efficiency and inversely related to the redundancy. Thus, we have

$$Q(n) = \frac{S(n)E(n)}{R(n)} = \frac{T^3(1)}{nT^2(n)O(n)} \tag{3.19}$$

Since $E(n)$ is always a fraction and $R(n)$ is a number between 1 and n, the quality $Q(n)$ is always upper-bounded by the speedup $S(n)$.

Example 3.3 A hypothetical workload and performance plots

In Fig. 3.4, we compare the relative magnitudes of $S(n)$, $E(n)$, $R(n)$, $U(n)$, and $Q(n)$ as a function of machine size n, with respect to a hypothetical workload characterized by $O(1) = T(1) = n^3$, $O(n) = n^3 + n^2 \log_2 n$, and $T(n) = 4n^3/(n + 3)$.

Substituting these measures into Eqs. 3.15 to 3.19, we obtain the following performance expressions:

$$\begin{aligned} S(n) &= (n + 3)/4 \\ E(n) &= (n + 3)/(4n) \\ R(n) &= (n + \log_2 n)/n \\ U(n) &= (n + 3)(n + \log_2 n)/(4n^2) \\ Q(n) &= (n + 3)^2/(16(n + \log_2 n)) \end{aligned}$$

The relationships $1/n \le E(n) \le U(n) \le 1$ and $0 \le Q(n) \le S(n) \le n$ are observed where the linear speedup corresponds to the ideal case of 100% efficiency.

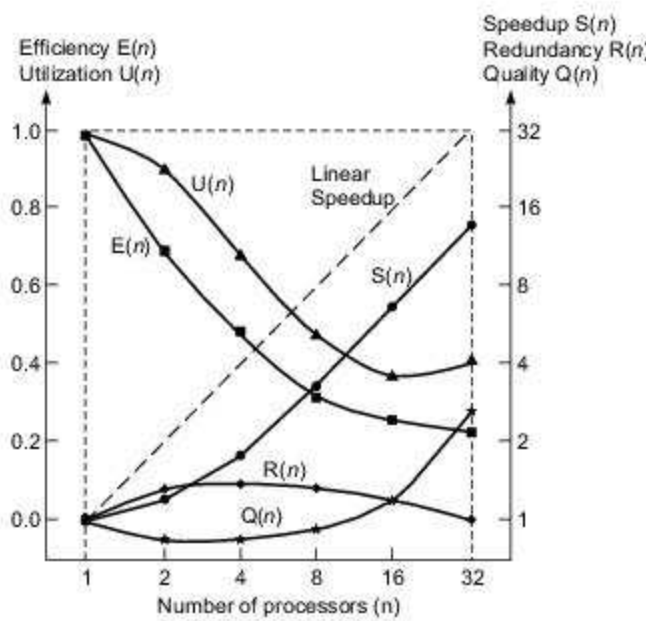


Fig. 3.4 Performance measures for Example 3.3 on a parallel computer with up to 32 processors

To summarize the above discussion on performance indices, we use the speedup $S(n)$ to indicate the degree of speed gain in a parallel computation. The efficiency $E(n)$ measures the useful portion of the total work performed by n processors. The redundancy $R(n)$ measures the extent of workload increase.

The utilization $U(n)$ indicates the extent to which resources are utilized during a parallel computation.

Finally, the quality $Q(n)$ combines the effects of speedup, efficiency, and redundancy into a single expression to assess the relative merit of a parallel computation on a computer system.

The speedup and efficiency of 10 parallel computers are reported in Table 3.1 for solving a linear system of 1000 equations. The table entries are excerpts from Table 2 in Dongarra's report (1992) on LINPACK benchmark performance over a large number of computers.

Either the standard LINPACK algorithm or an algorithm based on matrix-matrix multiplication was used in these experiments. A high degree of parallelism is embedded in these experiments. Thus high efficiency (such as 0.94 for the IBM 3090/6008 VF and 0.95 for the Convex C3240) was achieved. The low efficiency reported on the Intel Delta was based on some initial data.

Table 3.1 *Speedup and Efficiency of Parallel Computers for Solving a Linear System with 1000 Unknowns*

Computer Model	*No. of Processors* n	*Uni-processor Timing* T_1 (s)	*Multi-processor Timing* T_n (s)	*Speedup* $S = T_1/T_n$	*Efficiency* $E = S/n$
Cray Y-MP C90	16	0.77	0.069	11.12	0.69
NEC SX-3	2	0.15	0.082	1.82	0.91
Cray Y-MP/8	8	2.17	0.312	6.96	0.87
Fujitsu AP 1000	512	160.0	1.10	147.0	0.29
IBM 3090/600S VF	6	7.27	1.29	5.64	0.94
Intel Delta	512	22.0	1.50	14.7	0.03
Alliant FX/2800-200	14	22.9	2.06	11.1	0.79
nCUBE/2	1024	331.0	2.59	128.0	0.12
Convex C3240	4	14.9	3.92	3.81	0.95
Parsytec FT-400	400	1075.0	4.90	219.0	0.55

Source: Jack Dongarra, "Performance of Various Computers Using Standard Linear Equations Software," Computer Science Dept., Univ. of Tennessee, Knoxville, TN 37996-1301, March 11, 1992.

3.1.4 Benchmarks and Performance Measures

We have used MIPS and Mflops to describe the *instruction execution rate* and *floating-point capability* of a parallel computer. The MIPS rate defined in Eq. 1.3 is calculated from clock frequency and average CPI. In practice, the MIPS and Mflops ratings and other performance indicators to be introduced below should be measured from running benchmarks or real programs on real machines.

In this section, we introduce standard measures adopted by the industry to compare various computer performance, including *Mflops, MIPS, KLIPS, Dhrystone*, and *Whestone*, as often encountered in reported computer ratings.

Most computer manufacturers state peak or sustained performance in terms of MIPS or Mflops. These ratings are by no means conclusive. The real performance is always program-dependent or application-driven. In general, the MIPS rating depends on the instruction set, varies between programs, and even varies inversely with respect to performance, as observed by Hennessy and Patterson (1990).

To compare processors with different clock cycles and different instruction sets is not totally fair. Besides the native MIPS, one can define a *relative MIPS* with respect to a reference machine. We will discuss relative MIPS rating against the VAX/780 when Dhrystone performance is introduced below. For numerical computing, the LINPACK results on a large number of computers are reported in Chapter 8.

Similarly, the Mflops rating depends on the machine hardware design and on the program behavior. MIPS and Mflops ratings are not convertible because they measure different ranges of operations. The conventional rating is called the *native Mflops*, which does not distinguish unnormalized from normalized floating-point operations.

For example, a *real floating-point divide* operation may correspond to four *normalized floating-point divide* operations. One needs to use a conversion table between real and normalized floating-point operations to convert a native Mflops rating to a normalized Mflops rating.

The Dhrystone Results This is a CPU-intensive benchmark consisting of a mix of about 100 high-level language instructions and data types found in system programming applications where floating-point operations are not used (Weicker, 1984). The Dhrystone statements are balanced with respect to statement type, data type, and locality of reference, with no operating system calls and making no use of library functions or subroutines. Thus the Dhrystone rating should be a measure of the integer performance of modern processors. The unit *KDhrystones/s* is often used in reporting Dhrystone results.

The Dhrystone benchmark version 1.1 was applied to a number of processors. DEC VAX 11/780 scored 1.7 KDhrystones/s performance. This machine has been used as a reference computer with a 1 MIPS performance. The relative VAX/MIPS rating is commonly accepted by the computer industry.

The Whetstone Results This is a Fortran-based synthetic benchmark assessing the floating-point performance, measured in the number of *KWhetstones/s* that a system can perform. The benchmark includes both integer and floating-point operations involving array indexing, subroutine calls, parameter passing, conditional branching, and trigonometric/transcendental functions.

The Whetstone benchmark does not contain any vectorizable code and shows dependence on the system's mathematics library and efficiency of the code generated by a compiler.

The Whetstone performance is not equivalent to the Mflops performance, although the Whetstone contains a large number of scalar floating-point operations.

Both the Dhrystone and Whetstone are synthetic benchmarks whose performance results depend heavily on the compilers used. As a matter of fact, the Dhrystone benchmark program was originally written to test the CPU and compiler performance for a typical program. Compiler techniques, especially procedure in-lining, can significantly affect the Dhrystone performance.

Both benchmarks were criticized for being unable to predict the performance of user programs. The sensitivity to compilers is a major drawback of these benchmarks. In real-life problems, only application-oriented benchmarks will do the trick. We will examine the SPEC and other benchmark suites in Chapter 9.

The TPS and KLIPS Ratings On-line transaction processing applications demand rapid, interactive processing for a large number of relatively simple transactions. They are typically supported by very large databases. Automated teller machines and airline reservation systems are familiar examples. Today many such applications one web-based.

The throughput of computers for on-line transaction processing is often measured in *transactions per second* (TPS). Each transaction may involve a database search, query answering, and database update operations. Business computers and servers should be designed to deliver a high TPS rate. The TP1 benchmark was originally proposed in 1985 for measuring the transaction processing of business application computers. This benchmark also became a standard for gauging relational database performances.

Over the last couple of decades, there has been an enormous increase both in the diversity and the scale of computer applications deployed around the world. The world-wide web, web-based applications, multi-media applications and search engines did not exist in the early 19990s. Such scale and diversity have been made possible by huge advances in processing, storage, graphics display and networking capabilities over this period, which have been reviewed in Chapter 13.

For such applications, application-specific benchmarks have become more important than general purpose benchmarks such as Whetstone. For web servers providing 24 × 7 service for example, we may wish to benchmark—under simulated but realistic load conditions—performance parameters such as: *throughput* (in number of requests served and/or amount of data delivered) and *average response time.*

In artificial intelligence applications, the measure KLIPS (*kilo logic inferences per second*) was used at one time to indicate the reasoning power of an AI machine. For example, the high-speed inference machine developed under Japan's Fifth-Generation Computer System Project claimed a performance of 400 KLIPS.

Assuming that each logic inference operation involves about 100 assembly instructions, 400 KLIPS implies approximately 40 MIPS in this sense. The conversion ratio is by no means fixed. Logic inference demands symbolic manipulations rather than numeric computations. Interested readers are referred to the book edited by Wah and Ramamoorthy (1990).

3.2 PARALLEL PROCESSING APPLICATIONS

Massively parallel processing has become one of the frontier challenges in supercomputer applications. We introduce grand challenges in high-performance computing and communications and then assess the speed, memory, and I/O requirements to meet these challenges. Characteristics of parallel algorithms are also discussed in this context.

3.2.1 Massive Parallelism for Grand Challenges

The definition of massive parallelism varies with respect to time. Based on today's standards, any machine having hundreds or thousands of processors is a *massively parallel processing* (MPP) system. As computer technology advances rapidly, the demand for a higher degree of parallelism becomes more obvious.

The performance of most commercial computers is marked by their peak MIPS rate or peak Mflops rate. In reality, only a fraction of the peak performance is achievable in real benchmark or evaluation runs. Observing the sustained performance makes more sense in evaluating computer performance.

Grand Challenges We review below same of the grand challenges identified in the U.S. High-Performance Computing and Communication (HPCC) program, reveal opportunities for massive parallelism, assess past developments, and comment on future trends in MPP.

(1) The magnetic recording industry relies on the use of computers to study megmetostatic and exchange

interactions in order to reduce noise in metallic thin films used to coat high-density disks. In general, all research in science and engineering makes heavy demands on computing power.

(2) Rational drug design is being aided by computers in the search for a cure for cancer, acquired immunodeficiency syndrome and other diseases. Using a high-performance computer, new potential agents have been identified that block the action of human immunodeficiency virus protease.

(3) Design of high-speed transport aircraft is being aided by computational fluid dynamics running on supercomputers. Fuel combustion can be made more efficient by designing better engine models through chemical kinetics calculations.

(4) Catalysts for chemical reactions are being designed with computers for many biological processes which are catalytically controlled by enzymes. Massively parallel quantum models demand large simulations to reduce the time required to design catalysts and to optimize their properties.

(5) Ocean modeling cannot be accurate without supercomputing MPP systems. Ozone depletion and climate research demands the use of computers in analyzing the complex thermal, chemical and fluid-dynamic mechanisms involved.

(6) Other important areas demanding computational support include digital anatomy in real-time medical diagnosis, air pollution reduction through computational modeling, the design of protein structures by computational biologists, image processing and understanding, and technology linking research to education.

Besides computer science and computer engineering, the above challenges also encourage the emerging discipline of computational science and engineering. This demands systematic application of computer systems and computational solution techniques to mathematical models formulated to describe and to simulate phenomena of scientific and engineering interest.

The HPCC Program also identified some grand challenge computing requirements of the time, as shown in Fig. 3.5. This diagram shows the levels of processing speed and memory size required to support scientific simulation modeling, advanced *computer-aided design* (CAD), and real-time processing of large-scale database and information retrieval operations. In the period since the early 1990s, there have been huge advances in the processing, storage and networking capabilities of computer systems. Some MPP systems have reached petaflop performance, while even PCs have gigabytes of memory. At the same time, computing requirements in science and engineering have also grown enormously.

Exploiting Massive Parallelism The parallelism embedded in the instruction level or procedural level is rather limited. Very few parallel computers can successfully execute more than two instructions per machine cycle from the same program. *Instruction parallelism* is often constrained by program behavior, compiler/OS incapabilities, and program flow and execution mechanisms built into modern computers.

On the other hand, *data parallelism* is much higher than instruction parallelism. Data parallelism refers to the situation where the same operation (instruction or program) executes over a large array of data (operands). Data parallelism has been implemented on pipelined vector processors, SIMD array processors, and SPMD or MPMD multicomputer systems.

In Table 1.6, we find SIMD data parallelism over 65,536 PEs in the CM-2. One may argue that the CM-2 was a bit-slice machine. Even if we divide the number by 64 (the word length of a typical supercomputer), we still end up with a DOP on the order of thousands in CM-2.

The vector length can be used to determine the parallelism implementable on a vector supercomputer.

In the case of the Cray Y/MP C-90, 32 pipelines in 16 processors could potentially achieve a DOP of $32 \times 5 = 160$ if the average pipeline has five stages. Thus a pipelined processor can support a lower degree of data parallelism than an SIMD computer.

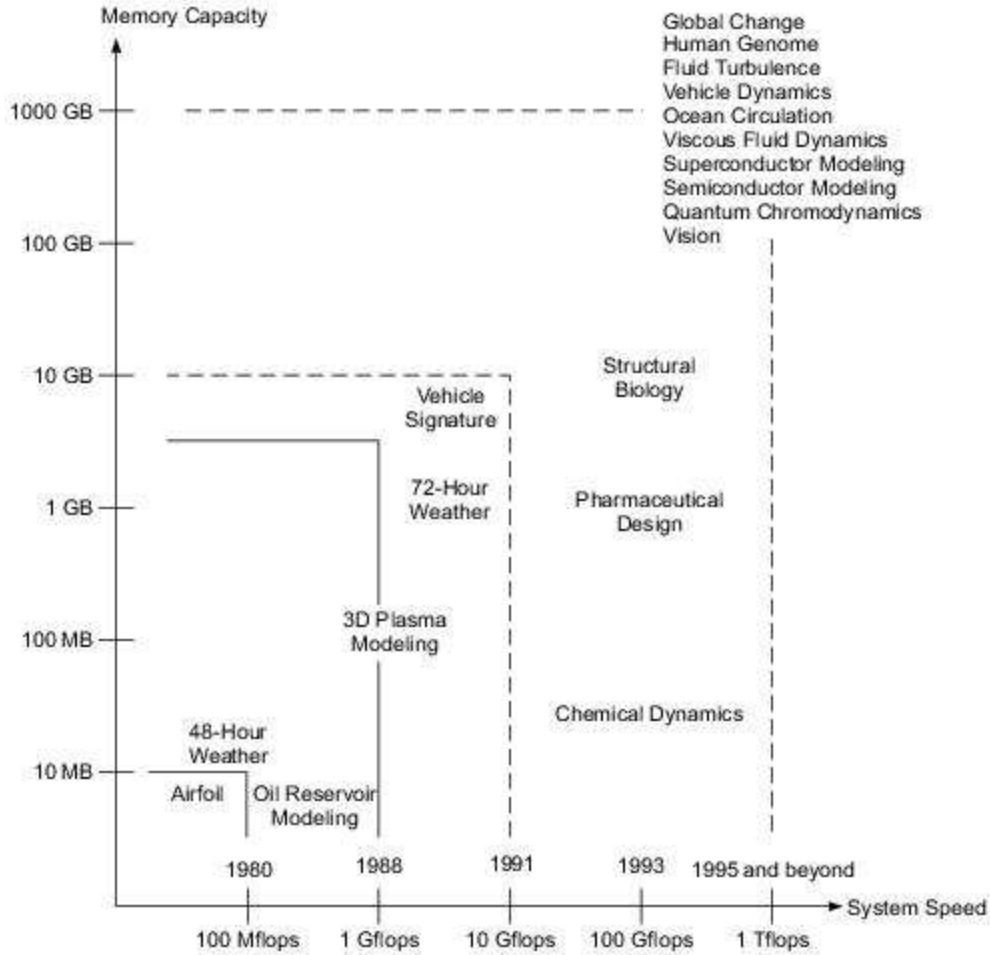


Fig. 3.5 Grand challenge requirements in computing and communications (Courtesy of U.S. High-Performance Computing and Communication Program, 1992)

On a message-passing multicomputer, the parallelism is more scalable than a shared-memory multiprocessor. As revealed in Table 1.4, the nCUBE/2 could achieve a maximum parallelism on the order of thousands if all the node processors were kept busy simultaneously.

The Past and the Future MPP systems started in 1968 with the introduction of the Illiac IV computer with 64 PEs under one controller. Subsequently, a massively parallel processor, called MPP, was built by Goodyear with 16,384 PEs. IBM built a GF11 machine with 576 PEs. The MasPar MP-1, AMT DAP610, and CM-2 were all early examples of SIMD computers.

Early MPP systems operating in MIMD mode included the BBN TC-2000 with a maximum configuration of 512 processors. The IBM RP-3 was designed to have 512 processors (only a 64-processor version was built). The Intel Touchstone Delta was a system with 570 processors.

Several subsequent MPP projects included the Paragon by Intel Supercomputer Systems, the CM-5 by Thinking Machine Corporation, the KSR-1 by Kendall Square Research, the Fujitsu VPP500 System, the Tera computer, and the MIT *T system.

IBM announced MPP projects using thousands of IBM RS/6000 and later Power processors, while Cray developed MPP systems using Digital's Alpha processors and later AMD Opteron processors as building blocks. Some early MPP projects are summarized in Table 3.2. We will study some of these systems in later chapters, and more recent advances in Chapter 13.

Table 3.2 *Early Representative Massively Parallel Processing Systems*

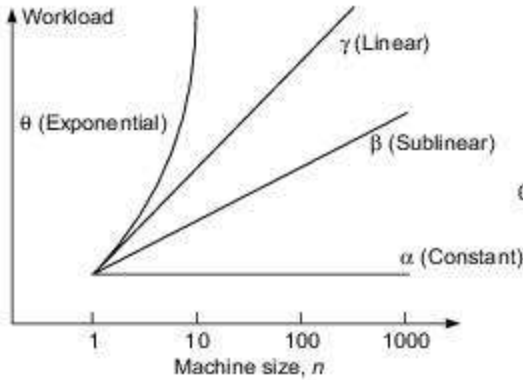
MPP System	*Architecture, Technology, and Operational Features*
Intel Paragon	A 2-D mesh-connected multicomputer, built with i860 XP processors and wormhole routers, targeted for 300 Gflops in peak performance.
IBM MPP Model	Use IBM RISC/6000 processors as building blocks, 50 Gflops peak expected for a 1024-processor configuration.
TMC CM-5	A universal architecture for SIMD/MIMD computing using SPARC PEs and custom-designed FPUs, control and data networks, 2 Tflops peak for 16K nodes.
Cray Research MPP Model	A 3D torus heterogeneous architecture using DEC Alpha chips with special communication support, global address space over physically distributed memory; first system offered 150 Gflops in a 1024-processor configuration in 1993; capable of growing to Tflops with larger configurations.
Kendall Square Research KSR-1	An ALLCACHE ring-connected multiprocessor with custom-designed processors, 43 Gflops peak performance for a 1088-processor configuration.
Fujitsu VPP500	A crossbar-connected 222-PE MIMD vector system, with shared distributed memories using VP2000 as a host; peak performance = 355 Gflops.

3.2.2 Application Models of Parallel Computers

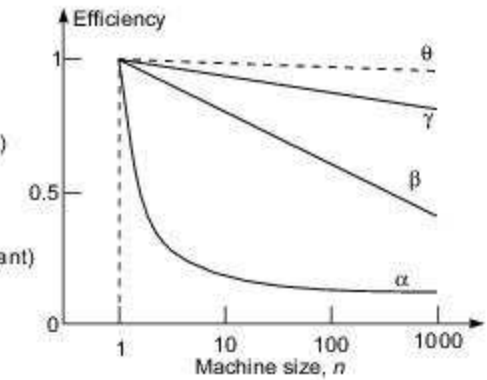
In general, if the workload is kept unchanged as shown by curve α in Fig. 3.6a, then the efficiency E decreases rapidly as the machine size n increases. The reason is that the overhead h increases faster than the machine size. To maintain the efficiency at a desired level, one has to increase the machine size and problem size proportionally. Such a system is known as a *scalable computer* for solving *scaled problems*.

In the ideal case, we like to see a workload curve which is a linear function of n (curve γ in Fig. 3.6a). This implies linear scalability in problem size. If the linear workload curve is not achievable, the second choice is to achieve a sublinear scalability as close to linearity as possible, as illustrated by curve β in Fig. 3.6a, which has a smaller constant of proportionality than the curve γ.

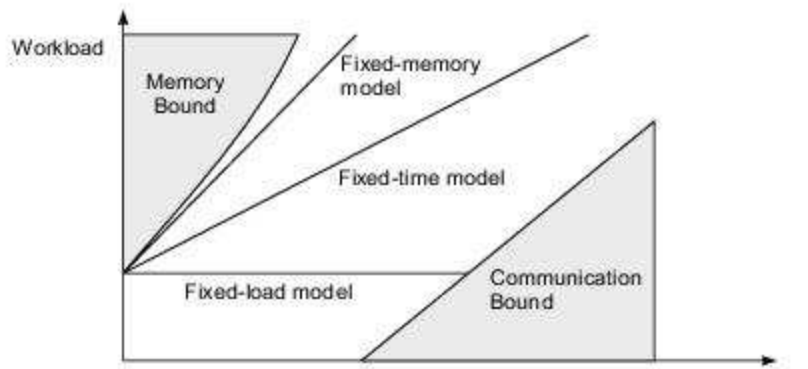
Suppose that the workload follows an exponential growth pattern and becomes enormously large, as shown by curve θ in Fig. 3.6a. The system is considered poorly scalable in this case. The reason is that to keep a constant efficiency or a good speedup, the increase in workload with problem size becomes explosive and exceeds the memory or I/O limits.



(a) Four workload growth patterns



(b) Corresponding efficiency curves



(c) Application models for parallel computers

Fig. 3.6 Workload growth, efficiency curves, and application models of parallel computers under resources constraints

The Efficiency Curves Corresponding to the four workload patterns specified in Fig. 3.6a, four efficiency curves are shown in Fig. 3.6b, respectively. With a constant workload, the efficiency curve (α) drops rapidly. In fact, curve α corresponds to the famous Amdahl's law. For a linear workload, the efficiency curve (γ) is almost flat, as observed by Gustafson in 1988.

The exponential workload (θ) may not be implementable due to memory shortage or I/O bounds (if real-time application is considered). Thus the θ efficiency (dashed lines) is achievable only with exponentially increased memory (or I/O) capacity. The sublinear efficiency curve (β) lies somewhere between curves α and γ.

Scalability analysis determines whether parallel processing of a given problem can offer the desired improvement in performance. The analysis should help guide the design of a massively parallel processor. It is clear that no single scalability metric suffices to cover all possible cases. Different measures will be useful in different contexts, and further analysis is needed along multiple dimensions for any specific application.

A parallel system can be used to solve arbitrarily large problems in a fixed time if and only if its workload

pattern is allowed to grow linearly. Sometimes, even if minimum time is achieved with more processors, the system utilization (or efficiency) may be very poor.

Application Models The workload patterns shown in Fig. 3.6a are not allowed to grow unbounded. In Fig. 3.6c, we show three models for the application of parallel computers. These models are bounded by limited memory, limited tolerance of IPC latency, or limited I/O bandwidth. These models are briefly introduced below. They lead to three speedup performance models to be formulated in Section 3.3.

The *fixed-load model* corresponds to a constant workload (curve α in Fig. 3.6a). The use of this model is eventually limited by the communication bound shown by the shaded area in Fig. 3.6c.

The *fixed-time model* demands a constant program execution time, regardless of how the workload scales up with machine size. The linear workload growth (curve γ in Fig. 3.6a) corresponds to this model. The *fixed-memory model* is limited by the memory bound, corresponding to a workload curve between γ and θ in Fig. 3.6a.

From the application point of view, the shaded areas are forbidden. The communication bound includes not only the increasing IPC overhead but also the increasing I/O demands. The memory bound is determined by main memory and disk capacities.

In practice, an algorithm designer or a parallel computer programmer may choose an application model within the above resource constraints, as shown in the unshaded application region in Fig. 3.6c.

Tradeoffs in Scalability Analysis Computer cost c and programming overhead p (in addition to speedup and efficiency) are equally important in scalability analysis. After all, cost-effectiveness may impose the ultimate constraint on computing with a limited budget. What we have studied above was concentrated on system efficiency and fast execution of a single algorithm/program on a given parallel computer.

It would be interesting to extend the scalability analysis to multiuser environments in which multiple programs are executed concurrently by sharing the available resources. Sometimes one problem is poorly scalable, while another has good scalability characteristics. Tradeoffs exist in increasing resource utilization but not necessarily to minimize the overall execution time in an optimization process.

Exploiting parallelism for higher performance demands both scalable architectures and scalable algorithms. The architectural scalability can be limited by long communication latency, bounded memory capacity, bounded I/O bandwidth, and limited processing speed. How to achieve a balanced design among these practical constraints is the major challenge of today's MPP system designers. On the other hand, parallel algorithms and efficient data structures also need to be scalable.

3.2.3 Scalability of Parallel Algorithms

In this subsection, we analyze the scalability of parallel algorithms with respect to key machine classes. An isoefficiency concept is introduced for scalability analysis of parallel algorithms. Two examples are used to illustrate the idea. Further studies of scalability are given in Section 3.4 after we study the speedup performance laws in Section 3.3.

Algorithmic Characteristics Computational algorithms are traditionally executed sequentially on uniprocessors. *Parallel algorithms* are those specially devised for parallel computers. The idealized parallel algorithms are those written for the PRAM models if no physical constraints or communication overheads are imposed. In the real world, an algorithm is considered efficient only if it can be cost effectively implemented on physical machines. In this sense, all machine-implementable algorithms must be architecture-dependent. This means the effects of communication overhead and architectural constraints cannot be ignored.

We summarize below important characteristics of parallel algorithms which are machine implementable:

(1) *Deterministic versus nondeterministic*: As defined in Section 1.4.1, only deterministic algorithms are implementable on real machines. Our study is confined to deterministic algorithms with polynomial time complexity

(2) *Computational granularity*: As introduced in Section 2.2.1, granularity decides the size of data items and program modules used in computation. In this sense, we also classify algorithms as fine-grain, medium-grain, or coarse-grain.

(3) *Parallelism profile*: The distribution of the degree of parallelism in an algorithm reveals the opportunity for parallel processing. This often affects the effectiveness of the parallel algorithms.

(4) *Communication patterns and synchronization requirements*: Communication patterns address both memory access and interprocessor communications. The patterns can be *static* or *dynamic*, depending on the algorithms. Static algorithms are more suitable for SIMD or pipelined machines, while dynamic algorithms are for MIMD machines. The *synchronization frequency* often affects the efficiency of an algorithm.

(5) *Uniformity of the operations*: This refers to the types of fundamental operations to be performed. Obviously, if the operations are uniform across the data set, the SIMD processing or pipelining may be more desirable. In other words, randomly structured algorithms are more suitable for MIMD processing. Other related issues include data types and precision desired.

(6) *Memory requirement and data structures*: In solving large-scale problems, the data sets may require huge memory space. *Memory efficiency* is affected by data structures chosen and data movement patterns in the algorithms. Both time and space complexities are key measures of the granularity of a parallel algorithm.

The Isoefficiency Concept The workload w of an algorithm grows with s, the problem size. Thus, we denote the workload $w = w(s)$ as a function of s. Kumar and Rao (1987) have introduced an *isoefficiency* concept relating workload to machine size n needed to maintain a fixed efficiency E when implementing a parallel algorithm on a parallel computer. Let h be the total communication overhead involved in the algorithm implementation. This overhead is usually a function of both machine size and problem size, thus denoted $h = h(s, n)$.

The efficiency of a parallel algorithm implemented on a given parallel computer is thus defined as

$$E = \frac{w(s)}{w(s) + h(s,n)} \tag{3.20}$$

The workload $w(s)$ corresponds to useful computations while the overhead $h(s, n)$ are computations attributed to synchronization and data communication delays. In general, the overhead increases with respect to both increasing values of s and n. Thus, the efficiency is always less than 1. The question is hinged on relative growth rates between $w(s)$ and $h(s, n)$.

With a fixed problem size (or fixed workload), the efficiency decreases as n increase. The reason is that the overhead $h(s, n)$ increases with n. With a fixed machine size, the overhead h grows slower than the workload w. Thus the efficiency increases with increasing problem size for a fixed-size machine. Therefore, one can expect to maintain a constant efficiency if the workload w is allowed to grow properly with increasing machine size.

For a given algorithm, the workload w might need to grow polynomially or exponentially with respect to n in order to maintain a fixed efficiency. Different algorithms may require different workload growth rates to keep the efficiency from dropping, as n is increased. The isoefficiency functions of common parallel algorithms are polynomial functions of n; i.e., they are $O(n^k)$ for some $k \geq 1$. The smaller the power of n in the isoefficiency function, the more scalable the parallel system. Here, the system includes the algorithm and architecture combination.

Isoefficiency Function We can rewrite Eq. 3.20 as $E = 1/(1 + h(s, n)/w(s))$. In order to maintain a constant E, the workload $w(s)$ should grow in proportion to the overhead $h(s, n)$. This leads to the following condition:

$$w(s) = \frac{E}{1-E} \times h(s, n) \tag{3.21}$$

The factor $C = E/(1 - E)$ is a constant for a fixed efficiency E. Thus we can define the *isoefficiency function* as follows:

$$f_E(n) = C \times h(s, n) \tag{3.22}$$

If the workload $w(s)$ grows as fast as $f_E(n)$ in Eq. 3.21, then a constant efficiency can be maintained for a given algorithm-architecture combination. Two examples are given below to illustrate the use of isoefficiency functions for scalability analysis.

Example 3.4 Scalability of matrix multiplication algorithms (Gupta and Kumar, 1992)

Four algorithms for matrix multiplication are compared below. The problem size s is represented by the matrix order. In other words, we consider the multiplication of two $s \times s$ matrices A and B to produce an output matrix C = A × B. The total workload involved is $w = O(s^3)$. The number of processors used is confined within $1 \leq n \leq s^3$. Some of the algorithms may use less than s^3 processors.

The isoefficiency functions of the four algorithms are derived below based on equating the workload with the communication overhead (Eq. 3.21) in each algorithm. Details of these algorithms and corresponding architectures can be found in the original papers identified in Table 3.3 as well as in the paper by Gupta and Kumar (1992). The derivation of the communication overheads is left as an exercise in Problem 3.14.

The Fox-Otto-Hey algorithm has a total overhead $h(s, n) = O(n \log n + s^2 \sqrt{n})$. The workload $w = O(s^3) = O(n \log n + s^2 \sqrt{n})$. Thus we must have $O(s^3) = O(n \log n)$ and $O(s) = O(\sqrt{n})$. Combining the two, we obtain the isoefficiency function $O(s^3) = O(n^{3/2})$, where $1 \leq n \leq s^2$ as shown in the first row of Table 3.3.

Although this algorithm is written for the torus architecture, the torus can be easily embedded in a hypercube architecture. Thus we can conduct a fair comparison of the four algorithms against the hypercube architecture.

Berntsen's algorithm restricts the use of $n \leq s^{3/2}$ processors. The total overhead is $O(n^{4/3} + n \log n + s^2 n^{1/3})$. To match this with $O(s^3)$, we must have $O(s^3) = O(n^{4/3})$ and $O(s^3) = O(n)$. Thus, $O(s^3)$ must be chosen to yield the isoefficiency function $O(n^2)$.

The Gupta-Kumar algorithm has an overhead $O(n \log n + s^2 n^{1/3} \log n)$. Thus we must have $O(s^3) = O(n \log n)$ and $O(s^3) = O(s^2 n^{1/3} \log n)$. This leads to the isoefficiency function $O(n(\log n)^3)$ in the third row of Table 3.3.

The Dekel-Nassimi-Sahni algorithm has a total overhead $O(n \log n + s^3)$ besides a useful computation time of $O(s^3/n)$ for $s^2 \le n \le s^3$. Thus the workload growth $O(s^3) = O(n \log n)$ will yield the isoefficiency listed in the last row of Table 3.3.

Table 3.3 *Asymptotic Isoefficiency Functions of Four Matrix Multiplication Algorithms (Gupta and Kumar, 1992)*

Matrix Multiplication Algorithm	*Isoefficiency Function $f_E(n)$*	*Range of Applicability*	*Target Machine Architecture*
Fox, Otto, and Hey (1987)	$O(n^3/2)$	$1 \le n \le s^2$	$A\ \sqrt{n} \times \sqrt{n}$ torus
Berntsen (1989)	$O(n^2)$	$1 \le n \le s^{3/2}$	A hypercube with $n = 2^{3k}$ nodes
Gupta and Kumar (1992)	$O(n(\log n)^3)$	$1 \le n \le s^3$	A hypercube with $n = 2^{3k}$ nodes and $k < \frac{1}{3} \log s$
Dekel, Nassimi, and Sahni (1981)	$O(n \log n)$	$s^2 \le n \le s^3$	A hypercube with $n = s^3 = 2^{3k}$ nodes

Note: Two $s \times s$ matrices are multiplied.

The above isoefficiency functions indicate the asymptotic scalabilities of the four algorithms. In practice, none of the algorithms is strictly better than the others for all possible problem sizes and machine sizes. For example, when these algorithms are implemented on a multicomputer with a long communication latency (as in Intel iPSC1), Berntsen's algorithm is superior to the others.

To map the algorithms on an SIMD computer with an extremely low synchronization overhead, the algorithm by Gupta and Kumar is inferior to the others. Hence, it is best to use the Dekel-Nassimi-Sahni algorithm for $s^2 \le n \le s^3$, the Fox-Otto-Hey algorithm for $s^{3/2} \le n \le s^2$, and Berntsen's algorithm for $n \le s^{3/2}$ for SIMD hypercube machines.

Example 3.5 Fast Fourier transform on mesh and hypercube computers (Gupta and Kumar, 1993)

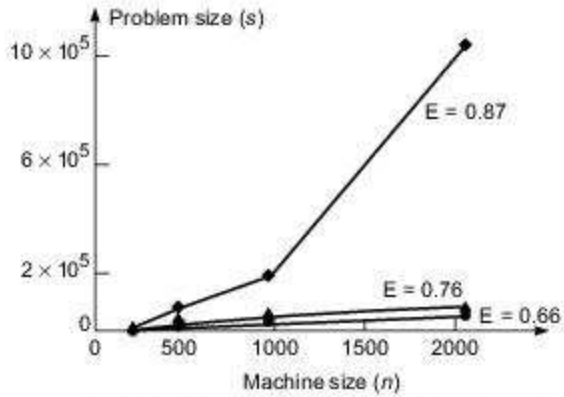
This example demonstrates the sensitivity of machine architecture on the scalability of the FFT on two different parallel computers: *mesh and hypercube*. We consider the Cooley-Tukey algorithm for one-dimensional s-point fast Fourier transform.

Gupta and Kumar have established the overheads: $h_1(s, n) = O(n \log n + s \log n)$ for FFT on a hypercube machine with n processors, and $h_2(s, n) = O(n \log n + s\sqrt{n})$ on a $\sqrt{n} \times \sqrt{n}$ mesh with n processors.

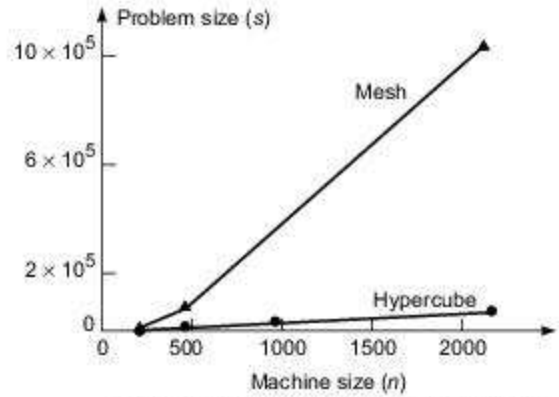
For an s-point FFT, the total workload involved is $w(s) = O(s \log s)$. Equating the workload with overheads, we must satisfy $O(s \log s) = O(n \log n)$ and $O(s \log s) = O(s \log n)$, leading to the isoefficiency function $f_1 = O(n \log n)$ for the hypercube machine.

Similarly, we must satisfy $O(s \log s) = O(n \log n)$ and $O(s \log s) = O(s\sqrt{n})$ by equating $w(s) = h_2(s, n)$. This leads to the isoefficiency function $f_2 = O(\sqrt{n}k^{\sqrt{n}})$ for some constant $k \leq 2$.

The above analysis leads to the conclusion that FFT is indeed very scalable on a hypercube computer. The result is plotted in Fig. 3.7a for three efficiency values.



(a) Hypercube under three operating efficiencies



(b) Comparison between mesh and hypercube

Fig. 3.7 Isoefficiency curves for FFT on two parallel computers (Courtesy of Gupta and Kumar, 1993)

To maintain the same efficiency, the mesh is rather poorly scalable as demonstrated in Fig. 3.7b.

This is predictable by the fact that the workload must grow exponentially in $O(\sqrt{n}k^{\sqrt{n}})$ for the mesh architecture, while the hypercube demands only $O(n \log n)$ workload increase as the machine size increases. Thus, we conclude that the FFT is scalable on a hypercube but not so on a mesh architecture.

If the bandwidth of the communication channels in a mesh architecture increases proportional to the increase of machine size, the above conclusion will change. For the design and analysis of FFT on parallel machines, readers are referred to the books by Aho, Hopcroft and Ullman (1974) and by Quinn (1987). We will further address scalability issues.from the architecture standpoint in Section 3.4.

3.3 SPEEDUP PERFORMANCE LAWS

Three speedup performance models are defined below. Amdahl's law (1967) is based on a fixed workload or a fixed problem size. Gustafson's law (1987) is applied to scaled problems, where the problem size increases with the increase in machine size. The speedup model by Sun and Ni (1993) is for scaled problems bounded by memory capacity.

3.3.1 Amdahl's Law for a Fixed Workload

In many practical applications that demand a real-time response, the computational workload is often fixed with a fixed problem size. As the number of processors increases in a parallel computer, the fixed load is distributed to more processors for parallel execution. Therefore, the main objective is to produce the results

as soon as possible. In other words, minimal turnaround time is the primary goal. Speedup obtained for time-critical applications is called fixed-load speedup.

Fixed-Load Speedup The ideal speedup formula given in Eq. 3.7 is based on a fixed workload, regardless of the machine size. Traditional formulations for speedup, including Amdahl's law, are all based on a fixed problem size and thus on a fixed load. The speedup factor is upper-bounded by a sequential bottleneck in this case.

We consider below both the cases of DOP $< n$ and of DOP $\geq n$. We use the ceiling function $\lceil x \rceil$ to represent the smallest integer that is greater than or equal to the positive real number x. When x is a fraction, $\lceil x \rceil$ equals 1. Consider the case where DOP $= i > n$. Assume all n processors are used to execute W_i exclusively. The execution time of W_i is

$$t_i(n) = \frac{W_i}{i\Delta}\left\lceil \frac{i}{n} \right\rceil \tag{3.23}$$

Thus the response time is

$$T(n) = \sum_{i=1}^{m} \frac{W_i}{i\Delta}\left\lceil \frac{i}{n} \right\rceil \tag{3.24}$$

Note that if $i < n$, then $t_i(n) = t_i(\infty) = W_i/i\Delta$. Now, we define the *fixed-load speedup factor* as the ratio of $T(1)$ to $T(n)$:

$$S_n = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{n} \right\rceil} \tag{3.25}$$

Note that $S_n \leq S_\infty \leq A$, by comparing Eqs. 3.4, 3.7, and 3.25.

A number of factors we have ignored may lower the speedup performance. These include communication latencies caused by delayed memory access, interprocessor communication over a bus or a network, or operating system overhead and delay caused by interrupts. Let $Q(n)$ be the lumped sum of all system overheads on an n-processor system. We can rewrite Eq, 3.25 as follows:

$$S_n = \frac{T(1)}{T(n) + Q(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{n} \right\rceil + Q(n)} \tag{3.26}$$

The overhead delay $Q(n)$ is certainly application-dependent as well as machine-dependent. It is very difficult to obtain a closed form for $Q(n)$. Unless otherwise specified, we assume $Q(n) = 0$ to simplify the discussion.

Amdahl's Law Revisited In 1967, Gene Amdahl derived a fixed-load speedup for the special case where the computer operates either in sequential mode (with DOP = 1) or in perfectly parallel mode (with DOP $= n$). That is, $W_i = 0$ if $i \neq 1$ or $i \neq n$ in the profile. Equation 3.25 is then simplified to

$$S_n = \frac{W_1 + W_n}{W_1 + W_n/n} \tag{3.27}$$

Amdahl's law implies that the sequential portion of the program W_i does not change with respect to the machine size n. However, the parallel portion is evenly executed by n processors, resulting in a reduced time.

Consider a normalized situation in which $W_1 + W_n = \alpha + (1 - \alpha) = 1$, with $\alpha = W_1$ and $\alpha = W_n$. Equation 3.27 is reduced to Eq. 3.14, where α represents the percentage of a program that must be executed sequentially and $1 - \alpha$ corresponds to the portion of the code that can be executed in parallel.

Amdahl's law is illustrated in Fig. 3.8. When the number of processors increases, the load on each processor decreases. However, the total amount of work (workload) $W_1 + W_n$ is kept constant as shown in Fig. 3.8a. In Fig. 3.8b, the total execution time decreases because $T_n = W_n/n$. Eventually, the sequential part will dominate the performance because $T_n \to 0$ as n becomes very large and T_1 is kept unchanged.

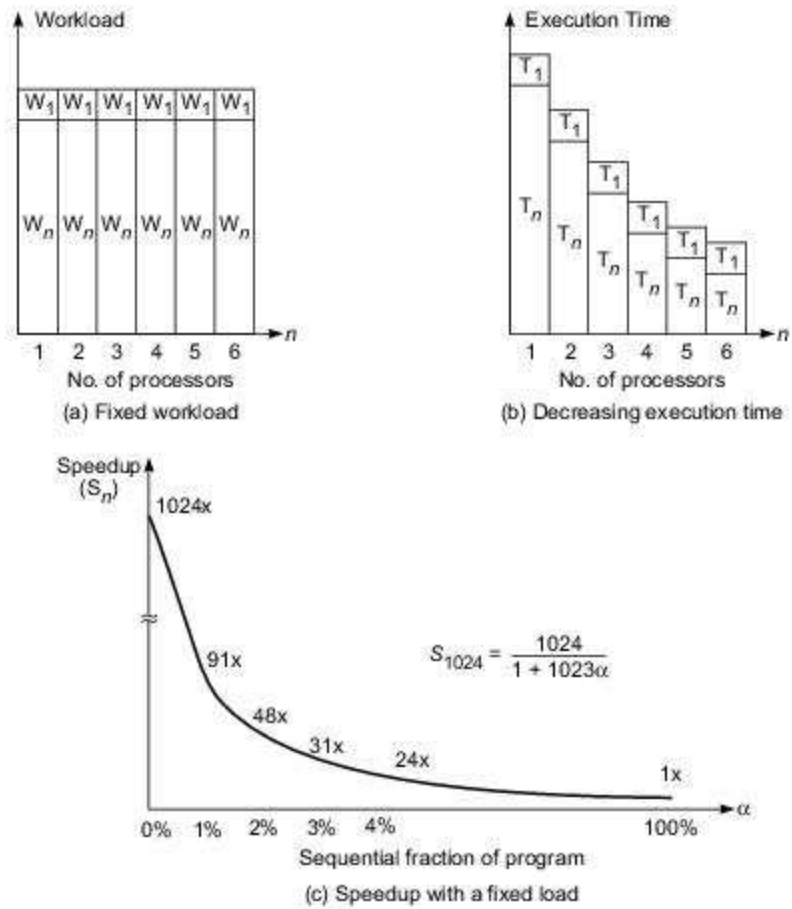


Fig. 3.8 Fixed-load speedup model and Amdahl's law

Sequential Bottleneck Figure 3.8c plots Amdahl's law using Eq. 3.14 over the range $0 \le \alpha \le 1$. The maximum speedup $S_n = n$ if $\alpha = 0$. The minimum speedup $S_n = 1$ if $\alpha = 1$. As $n \to \infty$, the limiting value of $S_n \to 1/\alpha$. This implies that the speedup is upper-bounded by $1/\alpha$, as the machine size becomes very large.

The speedup curve in Fig. 3.8c drops very rapidly as α increases. This means that with a small percentage of the sequential code, the entire performance cannot go higher than $1/\alpha$. This α has been called *the sequential bottleneck* in a program.

The problem of a sequential bottleneck cannot be solved just by increasing the number of processors in a system. The real problem lies in the existence of a sequential fraction of the code. This property has imposed a pessimistic view on parallel processing over the past two decades.

In fact, two major impacts on the parallel computer industry were observed. First, manufacturers were discouraged from making large-scale parallel computers. Second, more research attention was shifted toward developing parallelizing compilers which would reduce the value of α and in turn boost the performance.

3.3.2 Gustafson's Law for Scaled Problems

One of the major shortcomings in applying Amdahl's law is that the problem (workload) cannot scale to match the available computing power as the machine size increases. In other words, the fixed load prevents scalability in performance. Although the sequential bottleneck is a serious problem, the problem can be greatly alleviated by removing the fixed-load (or fixed-problem-size) restriction. John Gustafson (1988) has proposed a fixed-time concept which leads to a scaled speedup model.

Scaling for Higher Accuracy Time-critical applications provided the major motivation leading to the development of the fixed-load speedup model and Amdahl's law. There are many other applications that emphasize accuracy more than minimum turnaround time. As the machine size is upgraded to obtain more computing power, we may want to increase the problem size in order to create a greater workload, producing more accurate solution and yet keeping the execution time unchanged.

Many scientific modeling and engineering simulation applications demand the solution of very large-scale matrix problems based on some partial differential equation (PDE) formulations discretized with a huge number of grid points. Representative examples include the use of finite-element method to perform structural analysis or the use of finite-difference method to solve computational fluid dynamics problems in weather forecasting.

Coarse grids require less computation, but finer grids require many more computations, yielding greater accuracy. The weather forecasting simulation often demands the solution of four-dimensional PDEs. If one reduces the grid spacing in each physical dimension (X, Y, and Z) by a factor of 10 and increases the time steps by the same magnitude, then we are talking about an increase of 10^4 times more grid points. The workload thus increases to at least 10,000 times greater.

With such a problem scaling, of course, we demand more computing power to yield the same execution time. The main advantage is not in saving time but in producing much more accurate weather forecasting. This problem scaling for accuracy has motivated Gustafson to develop a fixed-time speedup model. The scaled problem keeps all the increased resources busy, resulting in a better system utilization ratio.

Fixed-Time Speedup In accuracy-critical applications, we wish to solve the largest problem size possible on a larger machine with about the same execution time as for solving a smaller problem on a smaller machine. As the machine size increases, we have to deal with an increased workload and thus a new parallelism profile. Let m' be the maximum DOP with respect to the scaled problem and W_i' be the scaled workload with DOP = i.

Note that in general $W_i' > W_i$ for $2 \le i \le m'$ and $W_1' = W_1$. The fixed-time speedup is defined under the assumption that $T(1) = T'(n)$, where $T'(n)$ is the execution time of the scaled problem and $T(1)$ corresponds to the original problem without scaling. We thus obtain

$$\sum_{i=1}^{m} W_i = \sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n) \tag{3.28}$$

A general formula for fixed-time speedup is defined by $S_n' = T(1)/T'(n)$, modified from Eq. 3.26:

$$S_n' = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i} \tag{3.29}$$

Gustafson's Law Fixed-time speedup was originally developed by Gustafson for a special parallelism profile with $W_i = 0$ if $i \neq 1$ and $i \neq n$. Similar to Amdahl's law, we can rewrite Eq. 3.29 as follows, assuming $Q(n) = 0$,

$$S_n' = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i} = \frac{W_1' + W_n'}{W_1 + W_n} = \frac{W_1 + nW_n}{W_1 + W_n} \tag{3.30}$$

where $W_n' = nW_n$ and $W_1 + W_n = W_1' + W_n'/n$, corresponding to the fixed-time condition. From Eq. 3.30, the parallel workload W_n' has been scaled to n times W_n in a linear fashion.

The relationship of a scaled workload to Gustafson's scaled speedup is depicted in Fig. 3.9. In fact, Gustafson's law can be restated as follows in terms of $\alpha = W_1$ and $1 - \alpha = W_n$ under the same assumption $W_1 + W_n = 1$ that we have made for Amdahl's law:

$$S_n' = \frac{\alpha + n(1-\alpha)}{\alpha + (1-\alpha)} = n - \alpha(n-1) \tag{3.31}$$

In Fig. 3.9a, we demonstrate the workload scaling situation. Figure 3.9b shows the fixed-time execution style. Figure 3.9c plots S_n' as a function of the sequential portion α of a program running on a system with $n = 1024$ processors.

Note that the slope of the S_n curve in Fig. 3.9c is much flatter than that in Fig. 3.8c. This implies that Gustafson's law does support scalable performance as the machine size increases. The idea is to keep all processors busy by increasing the problem size. When the problem can scale to match available computing power, the sequential fraction is no longer a bottleneck.

3.3.3 Memory-Bounded Speedup Model

Xian-He Sun and Lionel Ni (1993) have developed a memory-bounded speedup model which generalizes Amdahl's law and Gustafson's law to maximize the use of both CPU and memory capacities. The idea is to solve the largest possible problem, limited by memory space. This also demands a scaled workload, providing higher speedup, higher accuracy, and better resource utilization.

Memory-Bound Problems Large-scale scientific or engineering computations often require larger memory space. In fact, many applications of parallel computers are memory-bound rather than CPU-bound

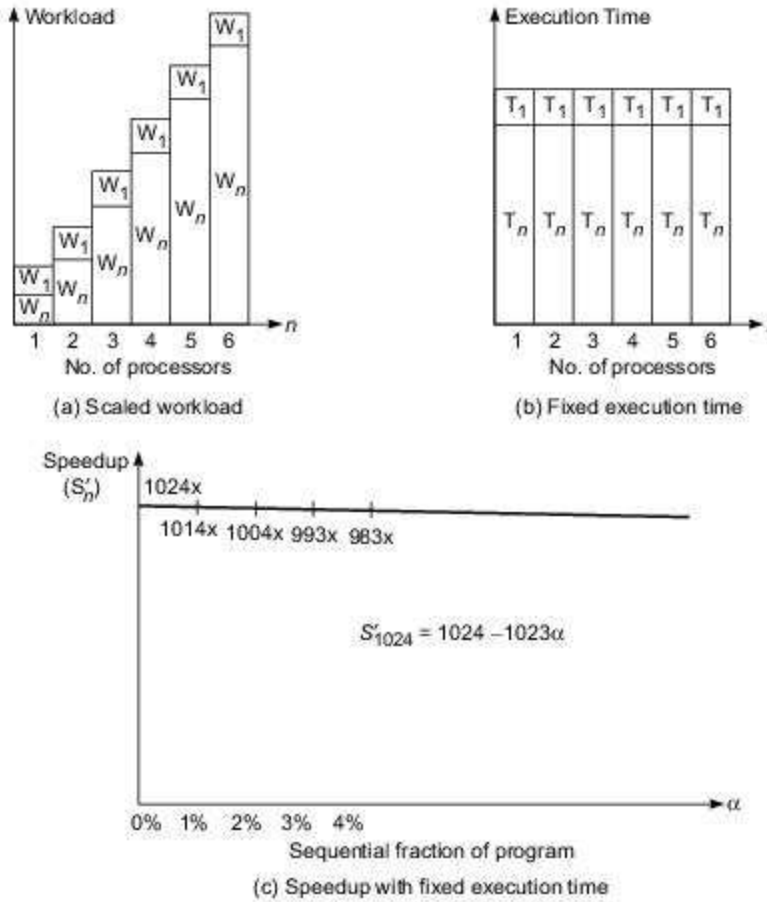


Fig. 3.9 Fixed-time speedup model and Gustafson's law

or I/O-bound. This is especially true in a multicomputer system using distributed memory. The local memory attached to each node may be relatively small. Therefore, each node can handle only a small subproblem.

When a large number of nodes are used collectively to solve a single large problem, the total memory capacity increases proportionally. This enables the system to solve a scaled problem through program partitioning or replication and domain decomposition of the data set.

Instead of keeping the execution time fixed, one may want to use up all the increased memory by scaling the problem size further. In other words, if you have adequate memory space and the scaled problem meets the time limit imposed by Gustafson's law, you can further increase the problem size, yielding an even better or more accurate solution.

A memory-bounded model was developed under this philosophy. The idea is to solve the largest possible problem, limited only by the available memory capacity. This model may result in an increase in execution time to achieve scalable performance.

Fixed-Memory Speedup Let M be the memory requirement of a given problem and W be the computational workload. These two factors are related to each other in various ways, depending on the address space and architectural constraints. Let us write $W = g(M)$ or $M = g^{-1}(W)$, where g^{-1} is the inverse of g.

In a multicomputer, the total memory capacity increases linearly with the number of nodes available. We write $W = \sum_{i=1}^{m^*} W_i$ as the workload for sequential execution of the program on a single node, and $W^* = \sum_{i=1}^{m^*} W_i^*$ as the scaled workload for execution on n nodes, where m^* is the maximum DOP of the scaled problem. The memory requirement for an active node is thus bounded by $g^{-1}\left(\sum_{i=1}^{m} W_i\right)$.

A *fixed-memory speedup* is defined below similarly to that in Eq. 3.29.

$$S_n^* = \frac{\sum_{i=1}^{m^*} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} \tag{3.32}$$

The workload for sequential execution on a single processor is independent of the problem size or system size. Therefore, we can write $W_1 = W_1' = W_i^*$ in all three speedup models. Let us consider the special case of two operational modes: *sequential* versus *perfectly parallel* execution. The enhanced memory is related to the scaled workload by $W_n^* = g^*(nM)$, where nM is the increased memory capacity for an n-node multicomputer.

Furthermore, we assume $g^*(nM) = G(n)g(M) = G(n)W_n$, where $W_n = g(M)$ and g^* is a homogeneous function. The factor $G(n)$ reflects the increase in workload as memory increases n times. Now we are ready to rewrite Eq. 3.32 under the assumption that $W_i = 0$ if $i \neq 1$ or n and $Q(n) = 0$:

$$S_n^* = \frac{W_1^* + W_n^*}{W_1^* + W_n^*/n} = \frac{W_1 + G(n)W_n}{W_1 + G(n)W_n/n} \tag{3.33}$$

Rigorously speaking, the above speedup model is valid under two assumptions: (1) The collection of all memory forms a global address space (in other words, we assume a shared distributed memory space); and (2) All available memory areas are used up for the scaled problem. There are three special cases where Eq. 3.33 can apply:

Case 1: $G(n) = 1$. This corresponds to the case where the problem size is fixed. Thus, the fixed-memory speedup becomes equivalent to Amdahl's law; i.e. Eqs. 3.27 and 3.33 are equivalent when a fixed workload is given.

Case 2: $G(n) = n$. This applies to the case where the workload increases n times when the memory is increased n times. Thus, Eq. 3.33 is identical to Gustafson's law (Eq. 3.30) with a fixed execution time.

Case 3: $G(n) > n$. This corresponds to the situation where the computational workload increases faster than the memory requirement. Thus, the fixed-memory model (Eq. 3.33) will likely give a higher speedup than the fixed-time speedup (Eq. 3.30).

The above analysis leads to the following conclusions: Amdahl's law and Gustafson's law are special cases of the fixed-memory model. When computation grows faster than the memory requirement, as is often true in some scientific simulation and engineering applications, the fixed-memory model (Fig. 3.10) may yield an even higher speedup (i.e., $S_n^* \geq S_n' \geq S_n$) and better resource utilization.

The fixed-memory model also assumes a scaled workload and allows an increase in execution time. The increase in workload (problem size) is memory-bound. The growth in machine size is limited by increasing

communication demands as the number of processors becomes large. The fixed-time model can be moved very close to the fixed-memory model if available memory is fully utilized.

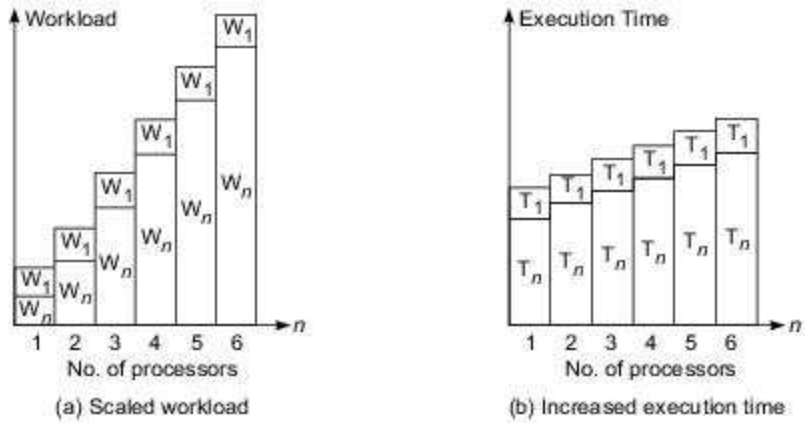


Fig. 3.10 Scaled speedup model using fixed memory (Courtesy of Sun and Ni; reprinted with permission from *ACM Supercomputing*, 1990)

Example 3.6 Scaled matrix multiplication using global versus local computation models (Sun and Ni, 1993)

In scientific computations, a matrix often represents some discretized data continuum. Enlarging the matrix size generally leads to a more accurate solution for the continuum. For matrices with dimension n, the number of computations involved in matrix multiplication is $2n^3$ and the memory requirement is roughly $M = 3n^2$.

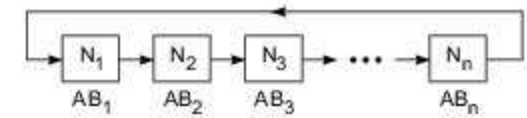
As the memory increases n times in an n-processor multicomputer system, $nM = n \times 3n^2 = 3n^3$. If the enlarged matrix has a dimension of N, then $3n^3 = 3N^2$. Therefore, $N = n^{1.5}$. Thus $G(n) = n^{1.5}$, and the scaled workload $W_n^* = G(n)W_n = n^{1.5}W$. Using Eq. 3.33, we have

$$S^* = \frac{W_1 + n^{1.5}W_n}{W_1 + \dfrac{n^{1.5}W_n}{n}} = \frac{W_1 + n^{1.5}W_n}{W_1 + n^{0.5}W_n} \tag{3.34}$$

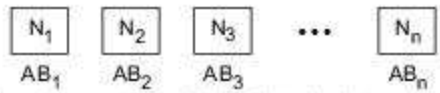
under the *global computation model* illustrated in Fig. 3.11a,. where all the distributed memories are used as. a common memory shared by all processor nodes.

As illustrated in Fig. 3.11b, the node memories are used locally without sharing. In such a *local computatien model*, $G(n) = n$, and we obtain the following speedup:

$$S_n^* = \frac{W_1 + nW_n}{W_1 + W_n} \tag{3.35}$$



(a) Global computation with distributed shared memories



(b) Local computation with distributed private memories

Fig. 3.11 Two models for the distributed matrix multiplication

The above example illustrates Gustafson's scaled speedup for local computation. Comparing the above two speedup expressions, we realize that the fixed-memory speedup (Eq. 3.34) may be higher than the fixed-time speedup (Eq. 3.35). In general, many applications demand the use of a combination of local and global addressing spaces. Data may be distributed in some nodes and duplicated in other nodes. Data duplication is added deliberately to reduce communication demand. Speedup factors for these applications depend on the ratio between the global and local computations.

3.4 SCALABILITY ANALYSIS AND APPROACHES

The performance of a computer system depends on a large number of factors, all affecting the scalability of the computer architecture and the application program involved. The simplest definition of *scalability* is that the performance of a computer system increases linearly with respect to the number of processors used for a given application,

Scalability analysis of a given computer system must be conducted for a given application program/algorithm. The analysis can be performed under different constraints on the growth of the problem size (workload) and on the machine size (number of processors). A good understanding of scalability will help evaluate the performance of parallel computer architectures for large-scale applications.

3.4.1 Scalability Metrics and Goals

Scalability studies determine the degree of matching between a computer architecture and an application algorithm. For different (architecture, algorithm) pairs, the analysis may end up with different conclusions. A machine can be very efficient for one algorithm but bad for another, and vice versa.

Thus, a good computer architecture should be efficient in implementing a large class of application algorithms. In the ideal case, the computer performance should be linearly scalable with an increasing number of processors employed in implementing the algorithms.

Scalability metrics Identified below are the basic metrics (Fig. 3.12) affecting the scalability of a computer system for a given application:

- *Machine size* (n)—the number of processors employed in a parallel computer system. A large machine size implies more resources and more computing power.

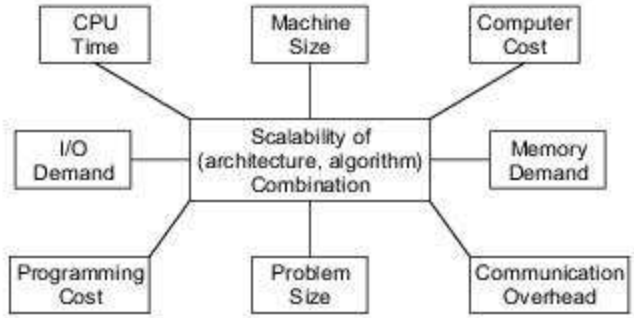


Fig. 3.12 Scalability metrics

- *Clock rate* (f)—the clock rate determines the basic machine cycle. We hope to build a machine with components (processors, memory, bus or network, etc.) driven by a clock which can scale up with better technology.
- *Problem size* (s)—the amount of computational workload or the number of data points used to solve a given problem. The problem size is directly proportional to the *sequential execution time* $T(s, 1)$ for a uniprocessor system because each data point may demand one or more operations.
- *CPU time* (T)—the actual CPU time (in seconds) elapsed in executing a given program on a parallel machine with n processors collectively. This is the *parallel execution time*, denoted as $T(s, n)$ and is a function of both s and n.
- *I/O demand* (d)—the input/output demand in moving the program, data, and results associated with a given application run. The I/O operations may overlap with the CPU operations in a multiprogrammed environment.
- *Memory capacity* (m)—the amount of main memory (in bytes or words) used in a program execution. Note that the memory demand is affected by the problem size, the program size, the algorithms, and the data structures used.

  The memory demand varies dynamically during program execution. Here, we refer to the maximum number of memory words demanded. Virtual memory is almost unlimited with a 64-bit address space. It is the physical memory which may be limited in capacity.
- *Communication overhead* (h)—the amount of time spent for interprocessor communication, synchronization, remote memory access, etc. This overhead also includes all noncompute operations which do not involve the CPUs or I/O devices. This overhead $h(s, n)$ is a function of s and n and is not part of $T(s, n)$. For a uniprocessor system, the overhead $h(s, 1) = 0$.
- *Computer cost* (c)—the total cost of hardware and software resources required to carry out the execution of a program.
- *Programming overhead* (p)—the development overhead associated with an application program. Programming overhead may slow down software productivity and thus implies a high cost. Unless otherwise stated, both computer cost and programming cost are ignored in our scalability analysis.

Depending on the computational objectives and resource constraints imposed, one can fix some of the above parameters and optimize the remaining ones to achieve the highest performance with the lowest cost.

The notion of scalability is tied to the notions of speedup and efficiency. A sound definition of scalability must be able to express the effects of the architecture's interconnection network, of the communication patterns

inherent to algorithms, of the physical constraints imposed by technology, and of the cost effectiveness or system efficiency. We introduce first the notion of speedup and efficiency. Then we define scalability based on the relative performance of a real machine compared with that of an idealized theoretical machine.

Speedup and Efficiency Revisited For a given architecture, algorithm, and problem size s, the *asymptotic speedup* $S(s, n)$ is the best speedup that is attainable, varying only the number (n) of processors. Let $T(s, 1)$ be the sequential execution time on a uniprocessor, $T(s, n)$ be the minimum parallel execution time on an n-processor system, and $h(s, n)$ be the lump sum of all communication and I/O overheads. The asymptotic speedup is formally defined as follows:

$$S(s, n) = \frac{T(s,1)}{T(s,n) + h(s, n)} \tag{3.36}$$

The problem size is the independent parameter, upon which all other metrics are based. A meaningful measurement of asymptotic speedup mandates the use of a good sequential algorithm, even it is different from the structure of the corresponding parallel algorithm. The $T(s, n)$ is minimal in the sense that the problem is solved using as many processors as necessary to achieve the minimum runtime for the given problem size.

In scalability analysis, we are mainly interested in results obtained from solving large problems. Therefore, the run times $T(s, n)$ and $T(s, 1)$ should be expressed using order-of-magnitude notations, reflecting the asymptotic behavior.

The *system efficiency* of using the machine to solve a given problem is defined by the following ratio:

$$E(s, n) = \frac{S(s, n)}{n} \tag{3.37}$$

In general, the best possible efficiency is one, implying that the best speedup is linear, or $S(s, n) = n$. Therefore, an intuitive definition of scalability is: *A system is scalable if the system efficiency* $E(s, n) = 1$ *for all algorithms with any number of n processors and any problem size s.*

Mark Hill (1990) has indicated that this definition is too restrictive to be useful because it precludes any system from being called scalable. For this reason, a more practical efficiency or scalability definition is needed, comparing the performance of the real machine with respect to the theoretical PRAM model.

Scalability Definition Nussbaum and Agarwal (1991) have given the following scalability definition based on a PRAM model. The scalability $\Phi(s, n)$ of a machine for a given algorithm is defined as the ratio of the asymptotic speedup $S(s, n)$ on the real machine to the asymptotic speedup $S_I(s, n)$ on the ideal realization of an EREW PRAM.

$$S_I(s, n) = \frac{T(s,1)}{T_I(s, n)}$$

where $T_I(s, n)$ is the parallel execution time on the PRAM, ignoring all communication overhead. The scalability is defined as follows:

$$\Phi(s, n) = \frac{S(s, n)}{S_I(s, n)} = \frac{T_I(s, n)}{T(s, n)} \tag{3.38}$$

Intuitively, the larger the scalability, the better the performance that the given architecture can yield running the given algorithm. In the ideal case, $S_I(s, n) = n$, the scalability definition in Eq. 3.38 becomes identical to the efficiency definition given in Eq. 3.37.

Example 3.7 Scalability of various machine architectures for parity calculation (Nussbaum and Agarwal, 1991)

Table 3.4 shows the execution times, asymptotic speedups, and scalabilities (with respect to the EREW-PRAM model) of five representative interconnection architectures: linear array, 2-D and 3-D meshes, hypercube, and Omega network, for running a parallel parity calculation.

Table 3.4 *Scalability of Various Network-Based Architectures for the Parity Calculation*

Metrics	*Machine Architecture*				
	Linear array	*2-D mesh*	*3-D mesh*	*Hypercube*	*Omega Network*
$T(s, n)$	$s^{1/2}$	$s^{1/3}$	$s^{1/4}$	$\log s$	$\log^2 s$
$S(s, n)$	$s^{1/2}$	$s^{2/3}$	$s^{3/4}$	$s/\log s$	$s/\log^2 s$
$\Phi(s, n)$	$\log s/s^{1/2}$	$\log s/s^{1/3}$	$\log s/s^{1/4}$	1	$1/\log s$

This calculation examines s bits, determining whether the number of bits set is even or odd using a balanced binary tree. For this algorithm, $T(s, 1) = s$, $T_I(s, n) = \log s$, and $S_I(s, n) = s/\log s$ for the ideal PRAM machine.

On real architectures, the parity algorithm's performance is limited by network diameter. For example, the linear array has a network diameter equal to $n - 1$, yielding a total parallel running time of $s/n + n$. The optimal partition of the problem is to use $n = \sqrt{s}$ processors so that each processor performs the parity check on $\sqrt{s}$ bits locally. This partition gives the best match between computation costs and communication costs with $T(s, n) = s^{1/2}$, $S(s, n) = s^{1/2}$ and thus scalability $\Phi(s, n) = \log s/s^{1/2}$.

The 2D and 3D mesh architectures use a similar partition to match their own communication structure with the computational loads, yielding even better scalability results. It is interesting to note that the scalability increases as the cornmunication latency decreases in a network with a smaller diameter.

The hypercube and the Omega network provide richer communication structures (and lower diameters) than meshes of lower dimensionality. The hypercube does as well as a PRAM for this algorithm, yielding $\Phi(s, n) = 1$.

The Omega network (Fig. 2.24) does not exploit locality: communication with all processors takes the same amount of time. This loss of locality hurts its performance when compared to the hypercube, but its lower diameter gives it better scalability than any of the meshes.

Although performance is limited by network diameter for the above parity algorithm, for many other algorithms the network bandwidth is the performance-limiting factor. The above analysis assumed unit communication time between directly connected communication nodes. An architecture may be scalable for one algorithm but unscalable for another. One must examine a large class of useful algorithms before drawing a scalability conclusion on a given architecture.

3.4.2 Evolution of Scalable Computers

The idea of massive parallelism is rather old, the technology is advancing steadily, and the software is relatively unexplored, as was observed by Cybenko and Kuck (1992). One evolutional trend is to build scalable supercomputers with distributed shared memory and standardized UNIX/LINUX for parallel processing. In this section, we present the evolutional path and some scalable computer design concepts; recent advances in this direction are discussed in Chapter 13.

The Evolutional Path Figure 3.13 shows the early evolution of supercomputers with four-to-five-year gestation and of micro-based scalable computers with three-year gestation. This plot shows the peak performance of Cray and NEC supercomputers and of Cray, Intel, and Thinking Machines scalable computers versus the introduction year. The marked nodes correspond to machine models with increasing size and cost.

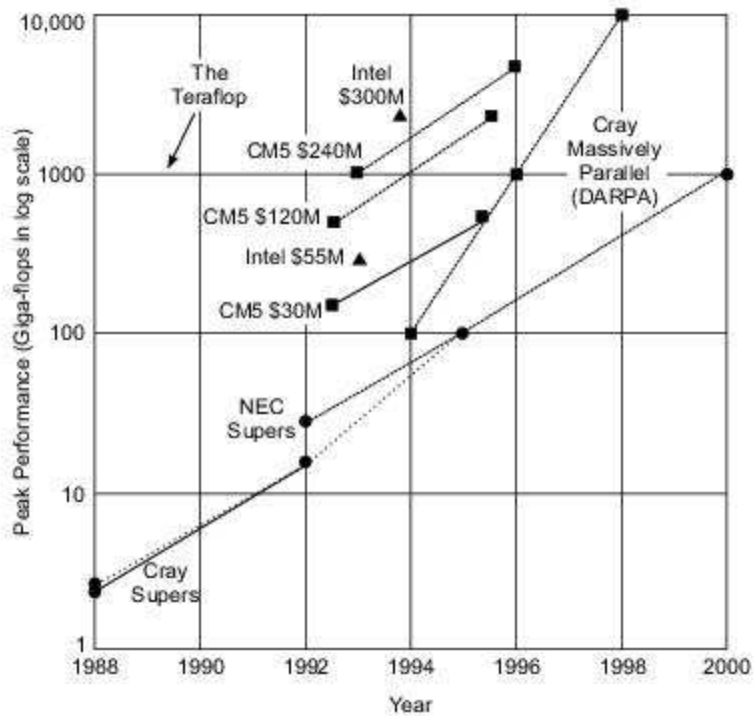


Fig. 3.13 The performance (in Gftops) of various computers manufactured during 1990s by Cray Research, Inc., NEC, Intel, and Thinking Machines Corporation (Courtesy of Gordon Bell; reprinted with permission from the *Communications of ACM*, August 1992)[1]

In 1988, the Cray Y-MP 8 delivered a peak of 2.8 Gflops. By 1991, the Intel Touchstone Delta, a 672-node multicomputer, and the Thinking Machines CM-2, a 2K PE SIMD machine, both began to supply an order-of-magnitude greater peak power (20 Gflops) than conventional supercomputers. By mid-1992, a completely new generation of computers were introduced, including the CM-5 and Paragon.

[1] Thinking Machines Corporation has since gone out of business.

In the past, the IBM System/360 provided a 100:1 range of growth for its various models. DEC VAX machines spanned a range of 1000:1 over their lifetime. Based on past experiences, Gordon Bell has identified three objectives for designing scalable computers. Implications and case studies of these challenges will be further discussed in subsequent chapters.

Size Scalability The study of system scalability started with the desire to increase the machine size. A size-scalable computer is designed to have a scaling range from a small to a large number of resource components. The expectation is to achieve linearly increased performance with incremental expansion for a well-defined set of applications. The components include computers, processors or processing elements, memories, interconnects, switches, cabinets, etc.

Size scalability depends on spatial and temporal locality as well as component bottleneck. Since very large systems have inherently longer latencies than small and centralized systems, the locality behavior of program execution will help tolerate the increased latency. Locality will be characterized in Chapter 4. The bottleneck-free condition demands a balanced design among processing, storage, and I/O bandwidth.

For example, since MPPs are mostly interconnected by large networks or switches, the bandwidth of the switch should increase linearly with processor power. The I/O demand may exceed the processing bandwidth in some real-time and large-scale applications.

The Cray Y-MP series scaled over a range of 16 processors (the C-90 model) and the current range of Cray supercomputers offer a much larger range of scalability (see Chapter 13). The CM-2 was designed to scale between 8K and 64K processing elements. The CM-5 scaling range was 1024 to 16K computers. The KSR-1 had a range of 8 to 1088 processor-memory pairs. Size-scalability cannot be achieved alone without considering cost, efficiency, and programmability on reasonable time scale.

Generation (Time) Scalability Since the basic processor nodes become obsolete every three years, the time scalability is equally important as the size scalability. Not only should the hardware technology be scalable, such as the CMOS circuits and packaging technologies in building processors and memory chips, but also the software/algorithm which demands software compatibility and portability with new hardware systems.

DEC claimed that the Alpha microprocessor was generation-scalable for 25 years. In general, all computer characteristics must scale proportionally: processing speed, memory speed and size, interconnect bandwidth and latency, I/O, and software overhead, in order to be useful for a given application.

Problem Scalability The problem size corresponds to the data set size. This is the key to achieving scalable performance as the program granularity changes. A problem scalable computer should be able to perform well as the problem size increases. The problem size can be scaled to be sufficiently large in order to operate efficiently on a computer with a given granularity.

Problems such as Monte Carlo simulation and ray tracing are "perfectly parallel", since their threads of computation do not come together over long spells of computation. Such an independence among threads is very much desired in using a scalable MPP system. In general, the *problem granularity* (operations on a grid point/data required from adjacent grid points) must be greater than a *machine's granularity* (node operation rate/node-to-node communication data rate) in order for a multicomputer to be effective.

Example 3.8 Problem scaling for solving Laplace equation on a distributed memory multicomputer (Gordon Bell, 1992)

Laplace equations are often used to model physical structures. A 3-D Laplace equation is specified by

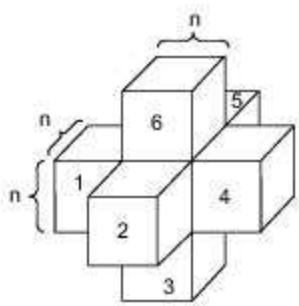
$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0 \tag{3.39}$$

We want to determine the problem scalability of the Laplace equation solver on a distributed-memory multicomputer with a sufficiently large number of processing nodes. Based on finite-difference method, solving Eq. 3.39 requires performing the following averaging operation iteratively across a very large grid, as shown in Fig. 3.14:

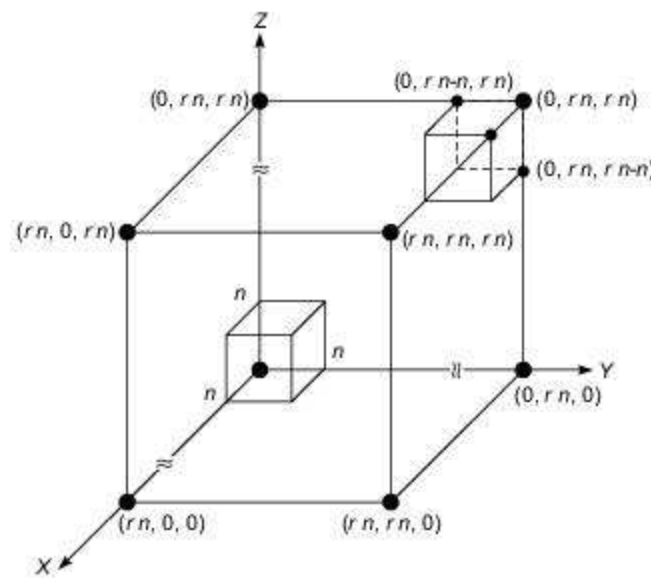
$$u_{i,j,k}^{(m)} = \frac{1}{6}\left[u_{i-1,j,k}^{(m-1)} + u_{i+1,j,k}^{(m-1)} + u_{i,j-1,k}^{(m-1)} + u_{i,j+1,k}^{(m-1)} + u_{i,j,k-1}^{(m-1)} + u_{i,j,k+1}^{(m-1)}\right] \tag{3.40}$$

where $1 \le i, j, k \le N$ and N is the number of grid points along each dimension. In total, there are N^3 grid points in the problem domain to be evaluated during each iteration m for $1 \le m \le M$.

The three-dimensional domain can be partitioned into p subdomains, each having n^3 grid points such that $pn^3 = N^3$, where p is the machine size. The computations involved in each subdomain are assigned to one node of a multicomputer. Therefore, in each iteration, each node is required to perform $7n^3$ computations as specified in Eq. 3.40.



(a) Six cube subdomains adjacent to a cube subdomain at the center



(b) An $N \times N \times N$ grid partitioned into p subdomains, each being an n^3 cube, where $p = r^3 = N^3/n^3$

Fig. 3.14 Partitioning of a 3D domain for solving the Laplace equation

Each subdomain is adjacent to six other subdomains (Fig. 3.14a). Therefore, in each iteration, each node needs to exchange (send or receive) a total of $6n^2$ words of floating-point numbers with its neighbors. Assume each floating-point number is double-precision (64 bits, or 8 bytes). Each processing node has the capability of performing 100 Mflops (or 0.01 μs per floating-point operation). The internode communication latency is assumed to be 1 μs (or 1 megaword/s) for transferring a floating-point number.

For a balanced multicomputer, the computation time within each node and inter-node communication latency should be equal. Thus $0.07n^3\mu$s equals $6n^2$ μs communication latency, implying that n has to be at least as large as 86. A node memory of capacity $86^3 \times 8 = 640\text{K} \times 8 = 5120$ Kwords = 5 megabytes is needed to hold each subdomain of data.

On the other hand, suppose each message exchange takes 2 μs (one receive and one send) per word. The communication latency is doubled. We desire to scale up the problem size with an enlarged local memory of 32 megabytes. The subdomain dimension size n can be extended to at most 160, because $160^3 \times 8 = 32$ megabytes. This size problem requires 0.3 s of computation time and 2×0.15 s of send and receive time. Thus each iteration takes 0.6 (0.3 + 0.3) s, resulting in a computation rate of 50 Mflops, which is only 50% of the peak speed of each node.

If the problem size n is further increased, the effective Mflops rate and efficiency will be improved. But this cannot be achieved unless the memory capacity is further enlarged. For a fixed memory capacity, the situation corresponds to the memorybound region shown in Fig. 3.6c. Another risk of problem scaling is to exacerbate the limited I/O capability which is not demonstrated in this example.

To summarize the above studies on scalability, we realize that the machine size, problem size, and technology scalabilities are not necessarily orthogonal to each other. They must be considered jointly. In the next section, we will identify additional issues relating scalability studies to software compatibility, latency tolerance, machine programmability, and cost-effectiveness.

3.4.3 Research Issues and Solutions

Toward the development of truly scalable computers, much research is being done. In this section, we briefly identify several frontier research problems. Partial solutions to these problems will be studied in subsequent chapters.

The Problems When a computer is scaled up to become an MPP system, the following difficulties can arise:

- Memory-access latency becomes too long and too nonuniformly distributed to be considered tolerable.
- The IPC complexity or synchronization overhead becomes too high to be useful.
- The multicache inconsistency problem becomes out of control.
- The processor utilization rate deteriorates as the system size becomes large.
- Message passing (or page migration) becomes too time-consuming to benefit resource sharing in a large distributed system.
- Overall system performance becomes saturated with diminishing return as system size increases further.

Some Approaches In order to overcome the above difficulties, listed below are some approaches being pursued by researchers:

- Searching for latency reducing and fast synchronization techniques.
- Using weaker memory consistency models.
- Developing scalable cache coherence protocols.
- Realizing shared virtual memory system.
- Integrating multithreaded architectures for improved processor utilization and system throughput.
- Expanding software portability and standardizing parallel and distributed UNIX/LINUX systems.

Scalability analysis can be carried out either by analytical methods or through trace-driven simulation experiments. In Chapter 9, we will study both approaches toward the development of scalable computer architectures that match program/ algorithmic behaviors. Analytical tools include the use of Markov chains, Petri nets, or queueing models. A number of simulation packages have already been developed at Stanford University and at MIT.

Supporting Issues Besides the emphases of scalability on machine size, problem size and technology, we identify below several extended areas for continued research and development:

(1) *Software scalability*: As problem size scales in proportion to the increase in machine size, the algorithms can be optimized to match the architectural constraints. Software tools are being developed to help programmers in mapping algorithms onto a target architecture.

A perfect match between architecture and algorithm requires matching both computational and communication patterns through performance-tuning experiments in addition to simple numerical analysis. Optimizing compilers and visualization tools should be designed to reveal opportunities for algorithm/program restructuring to match with the architectural growth.

(2) *Reducing communication overhead:* Scalability analysis should concern both useful computations and available parallelism in programs. The most difficult part of the analysis is to estimate the communication overhead accurately. Excessive communication overhead, such as the time required to synchronize a large number of processors, wastes system resources. This overhead grows rapidly as machine size and problem size increase.

Furthermore, the run time conditions are often difficult to capture. How to reduce the growth of communication overhead and how to tolerate the growth of memory-access latency in very large systems are still wide-open research problems.

(3) *Enhancing programmability*: The computing community generally agrees that multicomputers are more scalable; multiprocessors may be more easily programmed but are less scalable than multicomputers. It is the centralized-memory versus distributed private-memory organization that makes the difference. In the ideal case, we want to build machines which will retain the advantages of both architectures. This implies a system with shared distributed memory and simplified message communication among processor nodes. Hsterogencous programming paradigms are needed for future systems.

(4) *Providing longevity and generality*: Other scalability issues include *longevity*, which requires an architecture with sufficiently large address space, and *generality*, which supports a wide variety of languages and binary migration of software.

Performance, scalability, programmability, and generality will be studied throughout the book for general-purpose parallel processing applications, unless otherwise noted.

Summary

With rapid advances in technology, scalability becomes an important criterion for any modern computer system—and especially so for a parallel processing system. However, system scalability can only be defined in terms of system performance, and therefore issues of scalability and system performance are very closely interrelated. In this chapter, we have studied some basic issues related to the performance and scalability of parallel processing systems.

The main performance metric considered is the execution time of a parallel program which has a specific parallelism profile. As a program executes, the degree of parallelism in it varies with time, and therefore we can calculate the average degree of parallelism in the program. The parallelism profile also allows us to estimate the speedup achievable on the system as the number of processors is increased.

Apart from speedup, we also defined system efficiency and system utilization as asymptotic functions of the number of processors. On an n processor system, efficiency is defined as the speedup achieved divided by n (which is the ideal case speedup). System utilization, on the other hand, indicates the fraction of processor cycles which was actually utilized during program execution on the n processor system.

Benchmark programs are very useful tools in measuring the performance of computer systems. We looked at certain well-known benchmark programs, although it is also true that no two applications are identical and that therefore, in the final analysis, application specific benchmark programs are more useful.

We took a brief look at so-called 'grand challenge' applications of high performance computer systems; these are applications which are likely to have major impact in science and technology. Massively parallel processing (MPP) systems are increasingly being applied to such problems; clearly performance and scalability are important criteria for all such applications.

We then looked at some speedup performance laws governing parallel applications. Amdahl's law states in essence that, for a problem of a given fixed size, as the number of processors is increased, the speedup achievable is limited by the program fraction which must necessarily run as a sequential program, i.e. on one processor. Gustafson's law, on the other hand, studies also the effect of increasing the problem size as the system size is increased, resulting in the so-called fixed time speedup model. The third model studied was the memory-bounded speedup model proposed by Sun and Ni.

The specific metrics which affect the scalability of a computer system for a given application are—machine size in number of processors, processor clock rate, problem size, processor time consumed, I/O requirement, memory requirement, communication requirement, system cost, and programming cost of the application. Open research issues related to scalability in massively parallel systems were reviewed.

Exercises

Problem 3.1 Consider the parallel execution of the same program in Problem 1.4 on a four-processor system with shared memory. The program can be partitioned into four equal parts for balanced execution by the four processors. Due to the need for synchronization among the four program parts, 50000 extra instructions are added to each divided program part.

Assume the same instruction mix as in Problem 1.4 for each divided program part.

The CPI for the memory reference (with cache miss) instructions has been increased from 8 to 12 cycles due to contentions. The CPIs for the remaining instruction types do not change.

(a) Repeat part (a) in Problem 1.4 when the program is executed on the four-processor system.

(b) Repeat part (b) in Problem 1.4 when the program is executed on the four-processor system.

(c) Calculate the speedup factor of the four-processor system over the uniprocessor system in Problem 1.4 under the respective trace statistics.

(d) Calculate the efficiency of the four-processor system by comparing the speedup factor in part (c) with the ideal case.

Problem 3.2 A uniprocessor computer can operate in either scalar or vector mode. In vector mode, computations can be performed nine times faster than in scalar mode. A certain benchmark program took time T to run on this computer. Further, it was found that 25% of T was attributed to the vector mode. In the remaining time, the machine operated in the scalar mode.

(a) Calculate the effective speedup under the above condition as compared with the condition when the vector mode is not used at all. Also calculate α, the percentage of code that has been vectorized in the above program.

(b) Suppose we double the speed ratio between the vector mode and the scalar mode by hardware improvements. Calculate the effective speedup that can be achieved.

(c) Suppose the same speedup obtained in part (b) must be obtained by compiler improvements instead of hardware improvements. What would be the new vectorization ratio α that should be supported by the vectorizing compiler for the same benchmark program?

Problem 3.3 Let α be the percentage of a program code which can be executed simultaneously by n processors in a computer system. Assume that the remaining code must be executed sequentially by a single processor. Each processor has an execution rate of x MIPS, and all the processors are assumed equally capable.

(a) Derive an expression for the effective MIPS rate when using the system for exclusive execution of this program, in terms of the parameters n, α, and x.

(b) If $n = 16$ and $x = 400$ MIPS, determine the value of α which will yield a system performance of 4000 MIPS.

Problem 3.4 Consider a computer which can execute a program in two operational modes: *regular mode* versus *enhanced mode*, with a probability distribution of $\{\alpha, 1 - \alpha\}$, respectively.

(a) If α varies between a and b and $0 \le a < b \le 1$, derive an expression for the *average speedup factor* using the harmonic mean concept.

(b) Calculate the speedup factor when $a \to 0$ and $b \to 1$.

Problem 3.5 Consider the use of a four-processor, shared-memory computer for the execution of a program mix. The multiprocessor can be used in four execution modes corresponding to the active use of one, two, three, and four processors. Assume that each processor has a peak execution rate of 500 MIPS.

Let f_i be the percentage of time that i processors will be used in the above program execution and $f_1 + f_2 + f_3 + f_4 = 1$. You can assume the execution rates R_1, R_2, R_3, and R_4, corresponding to the distribution (f_1, f_2, f_3, f_4), respectively.

(a) Derive an expression to show the harmonic mean execution rate R of the multiprocessor in terms of f_i and R_i for $i = 1, 2, 3, 4$. Also show an expression for the harmonic mean execution time T in terms of R.

(b) What would be the value of the harmonic mean execution time T of the above program mix given $f_1 = 0.4$, $f_2 = 0.3$, $f_3 = 0.2$, $f_4 = 0.1$ and $R_1 = 400$ MIPS, $R_2 = 800$ MIPS, $R_3 = 1100$ MIPS, $R_4 = 1500$ MIPS? Explain the possible causes of observed R_i values in the above program execution.

(c) Suppose an intelligent compiler is used to enhance the degree of parallelization in the above program mix with a new distribution $f_1 = 0.1$, $f_2 = 0.2$, $f_3 = 0.3$, $f_4 = 0.4$. What would be the harmonic mean execution time of the same program under the same assumption on $\{R_i\}$ as in part (b)?

Problem 3.6 Explain the applicability and the restrictions involved in using Amdahl's law, Gustafson's law, and Sun and Ni's law to estimate the speedup performance of an n-processor system compared with that of a single-processor system. Ignore all communication overheads.

Problem 3.7 The following Fortran program is to be executed on a uniprocessor, and a parallel version is to be executed on a shared-memory multiprocessor.

```
L1:      Do 10 I = 1, 1024
L2:           SUM(I) = 0
L3:           Do 20 J = 1, I
L4:  20            SUM (I) = SUM (I) + I
L5:  10  Continue
```

Suppose statements 2 and 4 each take two machine cycle times, including all CPU and memory-access activities. Ignore the overhead caused by the software loop control (statements L1, L3, and L5) and all other system overhead and resource conflicts.

(a) What is the total execution time of the program on a uniprocessor?

(b) Divide the outer loop iterations among 32 processors with prescheduling as follows: Processor 1 executes the first 32 iterations (I = 1 to 32), processor 2 executes the next 32 iterations (I = 33 to 64), and so on. What are the execution time and speedup factors compared with part (a)? (Note that the computational workload, dictated by the J-loop, is unbalanced among the processors.)

(c) Modify the given program to facilitate a balanced parallel execution of all the computational workload over 32 processors. By a balanced load, we mean an equal number of additions assigned to each processor with respect to both loops.

(d) What is the minimum execution time resulting from the balanced parallel execution on 32 processors? What is the new speedup over the uniprocessor?

Problem 3.8 Consider the multiplications of two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$ on a scalar uniprocessor and on a multiprocessor, respectively. The matrix elements are floating-point numbers, initially stored in the main memory in row-major order. The resulting product matrix $C = (c_{ij})$ where $C = A \times B$, should be stored back to memory in contiguous locations.

Assume a 2-address instruction format and an instruction set of your choice. Each load/store instruction takes, on the average, 4 cycles to complete. All ALU operations must be done sequentially on the processor with 2 cycles if no memory reference is required in the instruction. Otherwise, 4 cycles are added for each memory reference to fetch an operand. Branch-type instructions require, on the average, 2 cycles.

(a) Write a minimal-length assembly-language program to perform the matrix multiplication on a scalar processor with a load-store architecture and floating-point hardware.

(b) Calculate the total instruction count, the total number of cycles needed for the program execution, and the average cycles per instruction (CPI).

(c) What is the MIPS rate of this scalar machine, if the processor is driven by a 400-MHz clock?

(d) Suggest a partition of the above program to execute the divided program parts on an N-processor shared-memory system with minimum time. Assume $n = 1000N$. Estimate the potential speedup of the multiprocessor over the uniprocessor, assuming the same type of processors are used in both systems. Ignore the memory-access conflicts, synchronization and other overheads.

(e) Sketch a scheme to perform distributed matrix computations with distributed data sets on an N-node multicomputer with distributed memory. Each node has a computer equivalent to the scalar processor used in part (a).

(f) Specify the message-passing operations required in part (e). Suppose that, on the average, each message passing requires 100 processor cycles to complete. Estimate the total execution time on the multicomputer for the distributed matrix multiplication. Make appropriate assumptions if needed in your timing analysis.

Problem 3.9 Consider the interleaved execution of the four programs in Problem 1.6 on each of the three machines. Each program is executed in a particular mode with the measured MIPS rating.

(a) Determine the arithmetic mean execution time per instruction for each machine executing the combined workload, assuming equal weights for the four programs.

(b) Determine the harmonic mean MIPS rate of each machine.

(c) Rank the machines based on the harmonic mean performance. Compare this ranking with that obtained in Problem 1.6.

Problem 3.10 Answer or prove the following statements related to speedup performance law:

(a) Derive the fixed-memory speedup expression S_n^* in Eq. 3.33 under reasonable assumptions.

(b) Derive Amdahl's law (S_n in Eq. 3.14) as a special case of the S_n^* expression.

(c) Derive Gustafson's law (S'_n in Eq. 3.34) as a special case of the S_n^* expression.

(d) Prove the relation $S_n^* \geq S'_n \geq S_n$ for solving the same problem on the same machine under different assumptions.

Problem 3.11 Prove the following relations among the speedup $S(n)$, efficiency $E(n)$, utilization $U(n)$, redundancy $R(n)$, and quality $Q(n)$ of a parallel computation, based on the definitions given by Lee (1980):

(a) Prove $1/n \leq E(n) \leq U(n) \leq 1$, where n is the number of processors used in the parallel computation.

(b) Prove $1 \leq R(n) \leq 1/E(n) \leq n$.

(c) Prove the expression for $Q(n)$ in Eq. 3.19.

(d) Verify the above relations using the hypothetical workload in Example 3.3.

Problem 3.12 Repeat Example 3.7 for sorting s numbers on five different n-processor machines using the linear array, 2D-mesh, 3D-mesh, hypercube, and Omega network as interprocessor communication architectures, respectively.

(a) Show the scalability of the five architectures as compared with the EREW-PRAM model.

(b) Compare the results obtained in part (a) with those in Example 3.7. Based on these two benchmark results, rank the relative scalability of the five architectures. Can the results be generalized to the performance of other algorithms?

Problem 3.13 Consider the execution of two benchmark programs. The performance of three computers running these two benchmarks are given below:

Benchmark	*Millions of floating-point operations*	*Computer 1* T_1 *(sec.)*	*Computer 2* T_2 *(sec.)*	*Computer 3* T_3 *(sec.)*
Problem 1	100	1	10	20
Problem 2	100	1000	100	20
Total time		1001	110	40

(a) Calculate R_a and R_h for each computer under the equal-weight assumption $f_1 = f_2 = 0.5$.

(b) When benchmark 1 has a constant R_1 = 10 Mflops performance across the three computers, plot R_a and R_h as a function of R_2, which varies from 1 to 100 Mflops under the assumption $f_1 = 0.8$ and $f_2 = 0.2$.

(c) Repeat part (b) for the case $f_1 = 0.2$ and $f_2 = 0.8$.

(d) From the above performance results under different conditions, can you draw a conclusion regarding the relative performance of the three machines?

Problem 3.14 In Example 3.5, four parallel algorithms are mentioned for multiplication of $s \times s$ matrices. After reading the original papers describing these algorithms, prove the following communication overheads on the target machine architectures:

(a) Prove that $h(s, n) = O(n \log n + s^2\sqrt{n})$ when mapping the Fox-Otto-Hey algorithm on a $\sqrt{n} \times \sqrt{n}$ torus.

(b) Prove that $h(s, n) = O(n^{4/3} + n \log n + s^2 n^{1/3})$ when mapping Berntsen's algorithm on a hypercube with $n = 2^{3k}$ nodes, where $k \leq \frac{1}{2} \log s$.

(c) Prove that $h(s, n) = O(n \log n + s^3)$ when mapping the Dekel-Nassimi-Sahni algorithm on a hypercube with $n = s^3 = 2^{3k}$ nodes.

Problem 3.15 Xian-He Sun (1992) has introduced an *isospeed* concept for scalability analysis. The concept is to maintain a fixed speed for each processor while increasing the problem size. Let W and W' be two workloads corresponding to two problem sizes. Let N and N' be two machine sizes (in terms of the number of processors). Let T_N and $T_{N'}$ be the parallel execution times using N and N' processors, respectively.

The isospeed is achieved when $W/(NT_N) = W'/(N'T_{N'})$. The *isoefficiency* concept defined by Kumar and Rao (1987) is achieved by maintaining a fixed efficiency through $S_N(W)/N = S_{N'}(W')/N'$, where $S_N(W)$ and $S_{N'}(W')$ are the corresponding speedup factors.

Prove that the two concepts are indeed equivalent if (i) the speedup factors are defined as the ratio of *parallel speed* R_N to *sequential speed* R_1 (rather than as the ratio of sequential execution time to parallel execution time), and (ii) $R_1(W) = R_1(W')$. In other words, isoefficiency is identical to isospeed when the sequential speed is fixed as the problem size is increased.

The McGraw-Hill Companies