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.

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.

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.

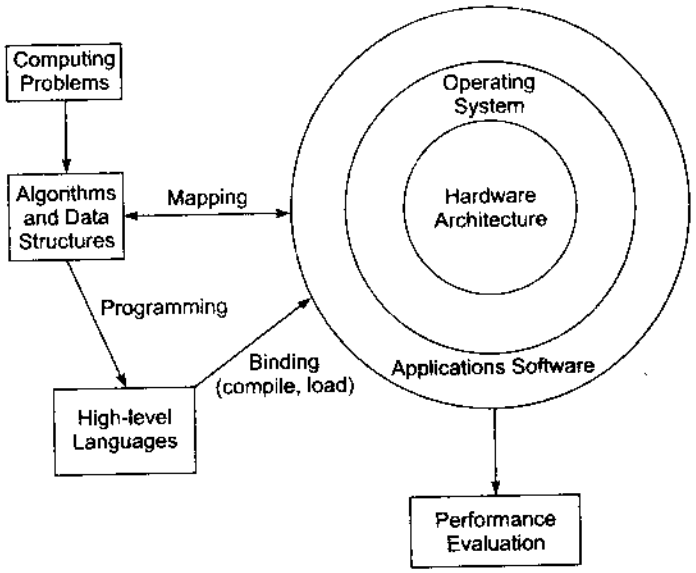


Fig. 1.1 Elements of a modern computer system

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.

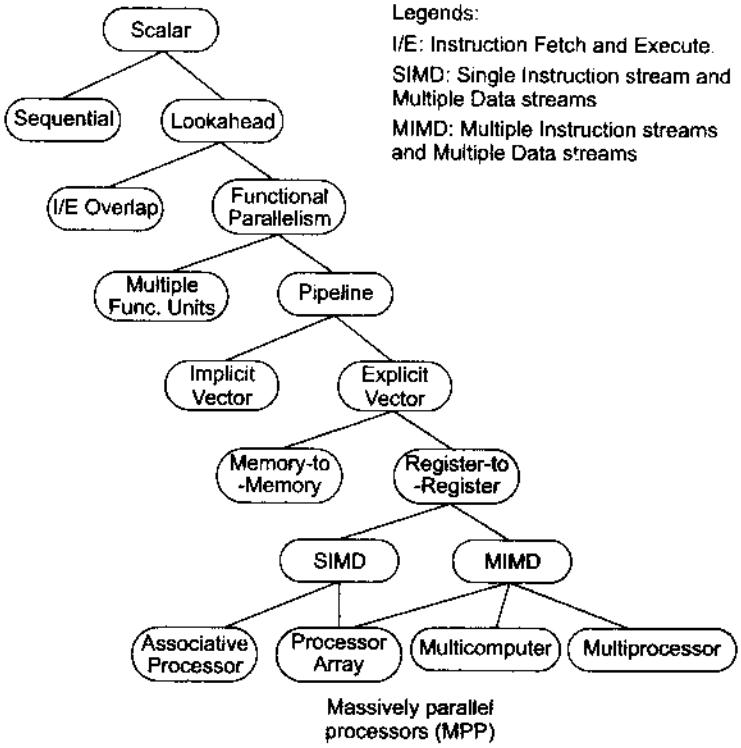


Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers

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.

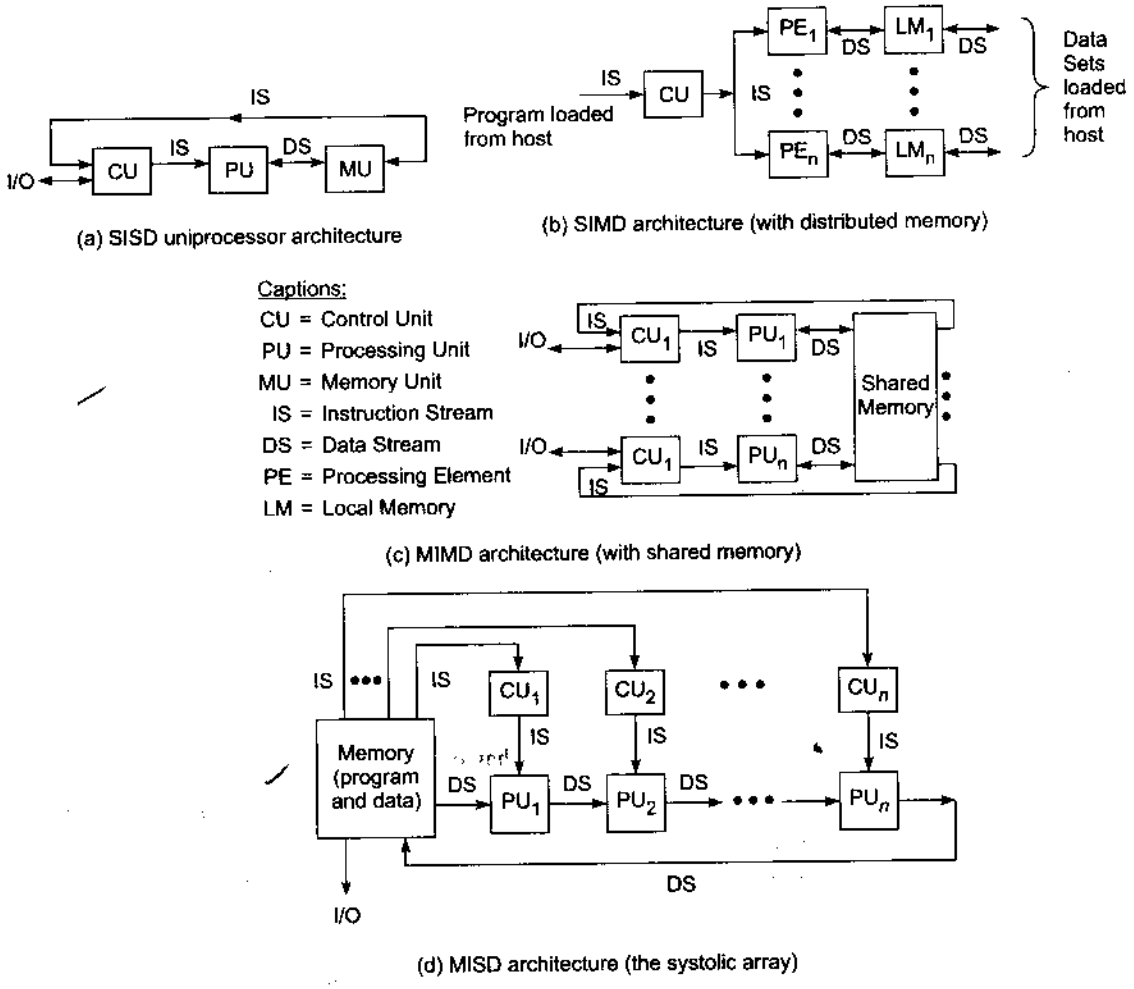


Fig. 1.3 Flynn's classification of computer architectures (Derived from Michael Flynn, 1972)

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.

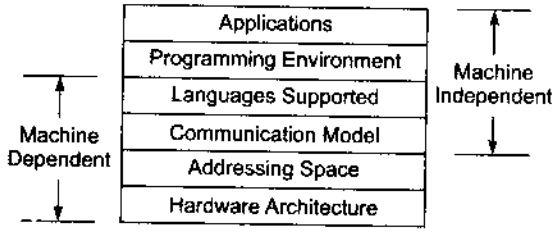


Fig. 1.4 Six layers for computer system development (Courtesy of Lionel Ni, 1990)

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, 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 = (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.

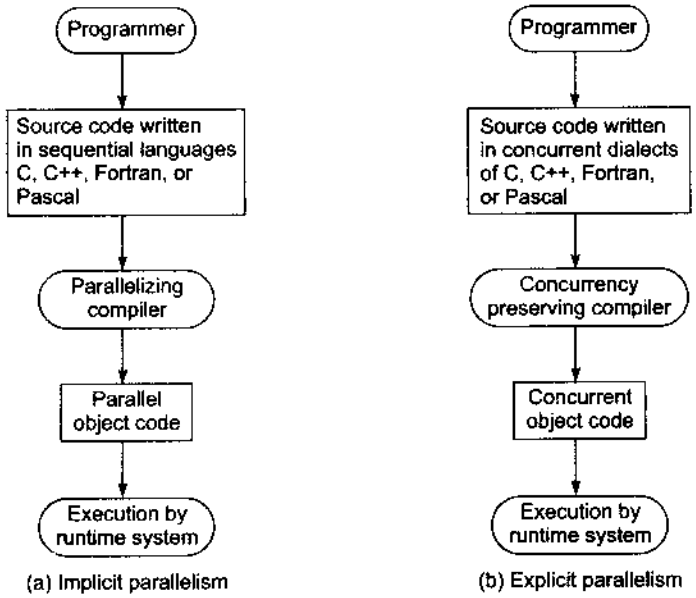


Fig. 1.5 Two approaches to parallel programming (Courtesy of Charles Seitz; adapted with permission from "Concurrent Architectures", p. 51 and p. 53, *VLSI and Parallel Computation*, edited by Suaya and Birtwistle, Morgan Kaufmann Publishers, 1990)

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.

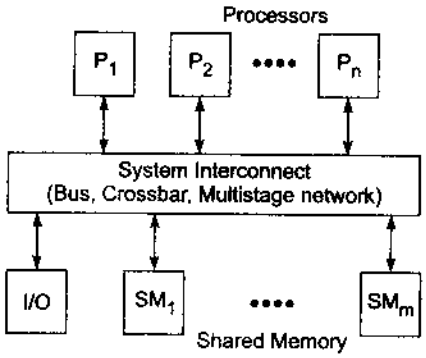


Fig. 1.6 The UMA multiprocessor model

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.

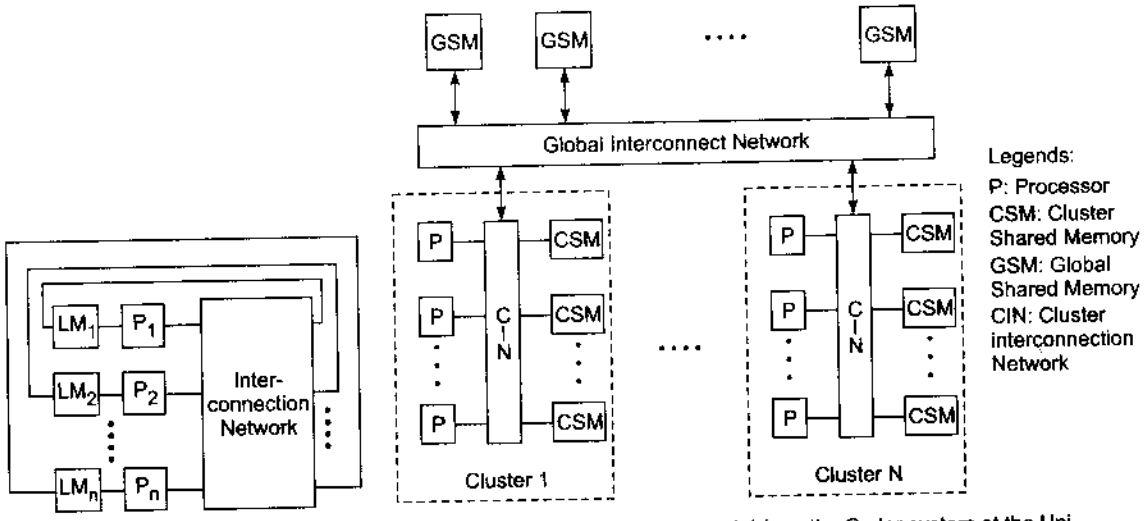
It is faster to access a local memory with a local processor. The access of remote memory attached to other processors takes longer due to the added delay through the interconnection network. The BBN TC-2000 Butterfly multiprocessor had the configuration shown in Fig. 1.7a.

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.

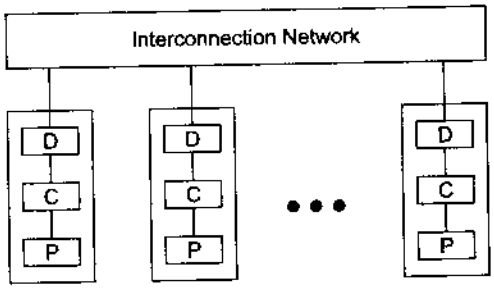


Fig. 1.8 The COMA model of a multiprocessor (P: Processor, C: Cache, D: Directory; e.g. the KSR-1)

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.

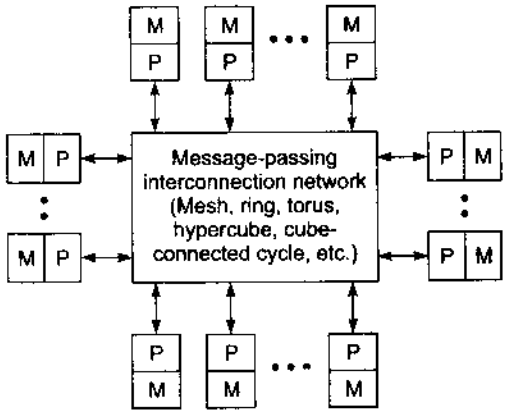


Fig. 1.9 Generic model of a message-passing multicomputer

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.

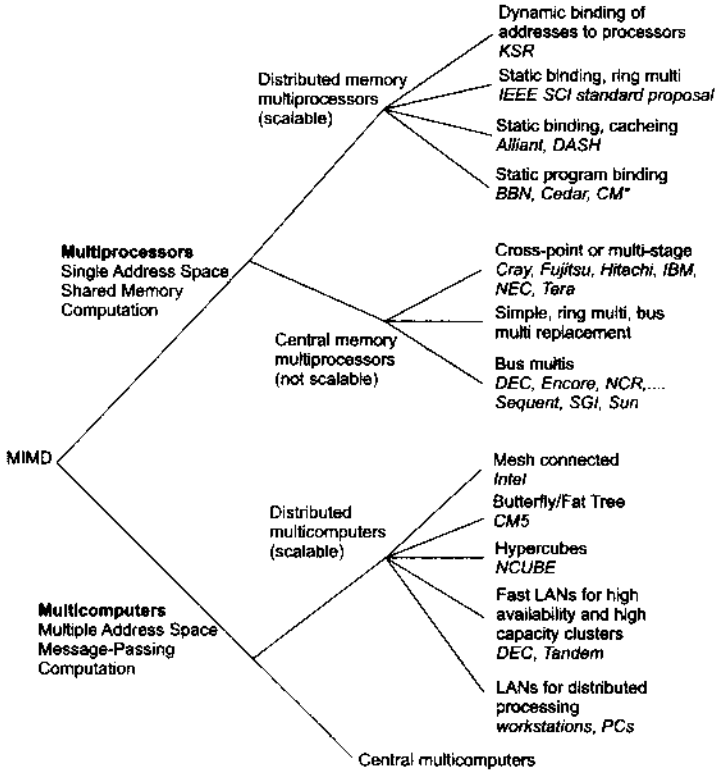


Fig. 1.10 Bell's taxonomy of MIMD computers (Courtesy of Gordon Bell; reprinted with permission from the *Communications of ACM*, August 1992)

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.

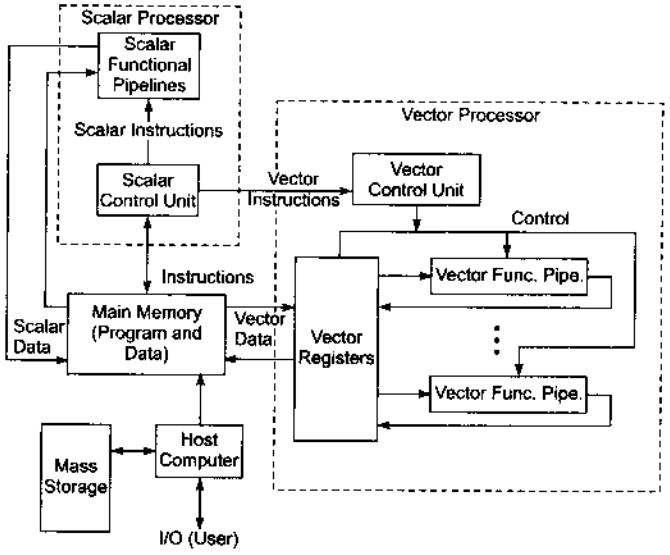


Fig. 1.11 The architecture of a vector supercomputer

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 × 64 × 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.

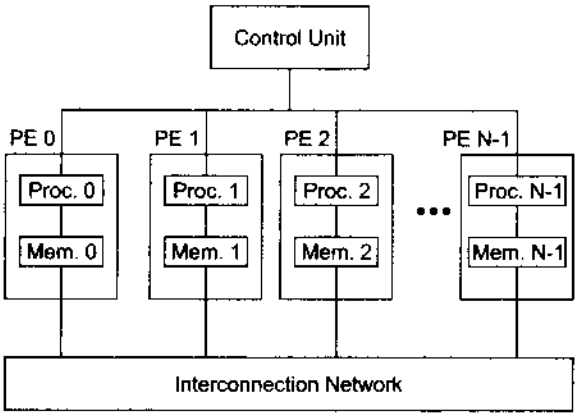


Fig. 1.12 Operational model of SIMD computers

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 ≠ NP. This leads to the conjecture that there exists a subclass, called *NP-complete* (NPC) problems, such that NPC ⊂ NP but NPC ∩ P = ϕ (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.

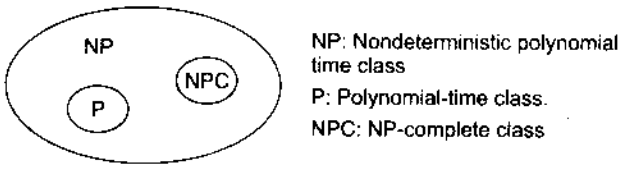


Fig. 1.13 The relationships conjectured among the NP, P, and NPC classes of computational problems

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:

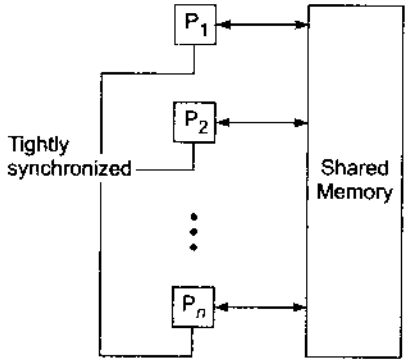


Fig. 1.14 PRAM model of a multiprocessor system with shared memory, on which all n processors operate in lockstep in memory access and program execution operations. Each processor can access any memory location in unit time

- *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 $2 \log n - 1 + \log(n/\log n) \approx O(\log n)$ 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

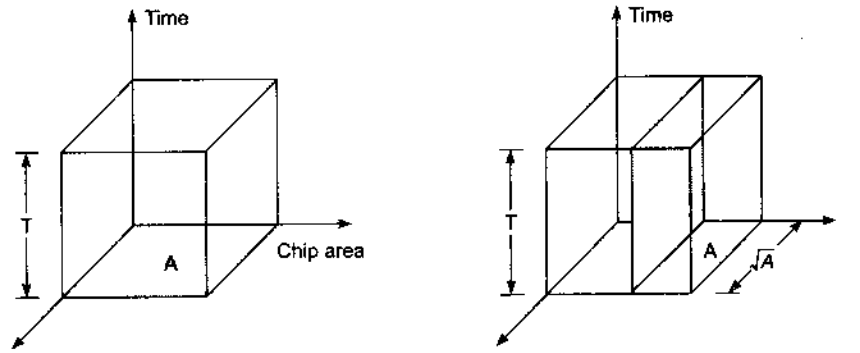
$$A \times T^2 \geq O(f(s)) \tag{1.6}$$

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 $\sqrt{AT}$

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, $\sqrt{A}T$ Figure 1.15b depicts a communication limited lower bound on the bisection area $\sqrt{A}T$. 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 $\sqrt{A}T$ 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 $0 \le i, j \le n - 1$.

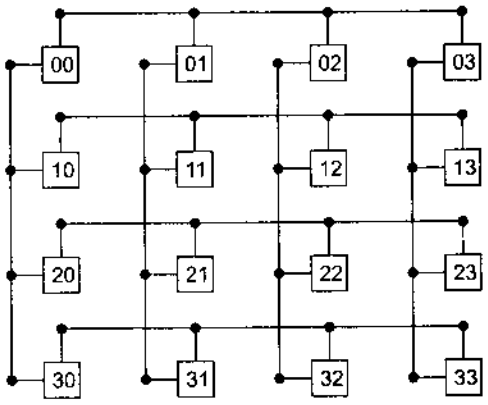


Fig. 1.16 A 4 × 4 mesh of processing elements (PEs) with broadcast buses on each row and on each column (Courtesy of Prasanna Kumar and Raghavendra; reprinted from *Journal of Parallel and Distributed Computing*, April 1987)

```
      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.

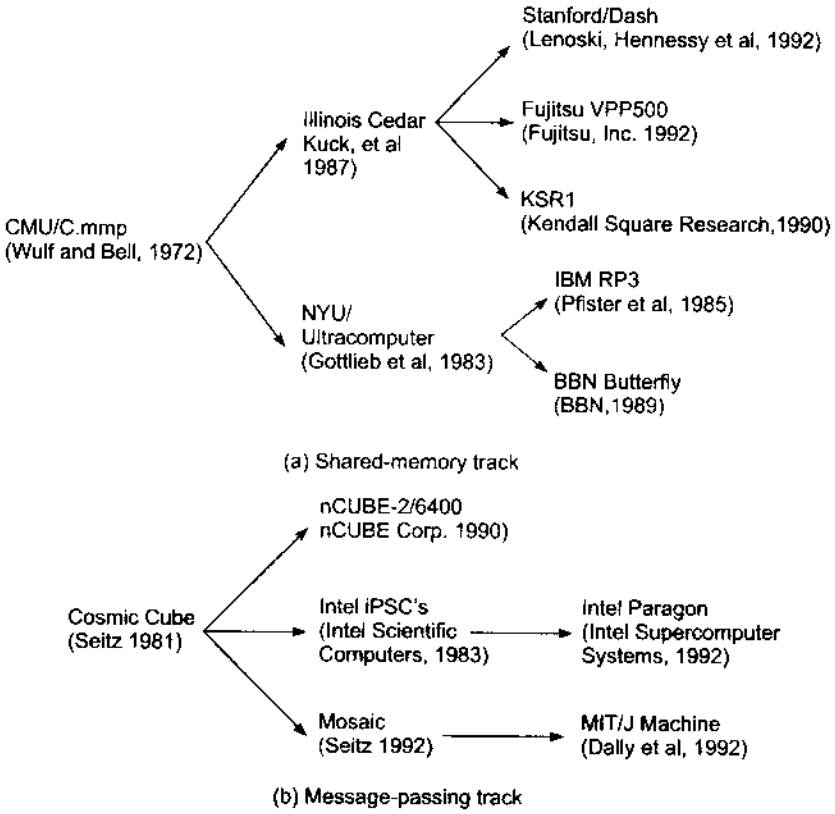


Fig. 1.17 Two multiple-processor tracks with and without shared memory

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.

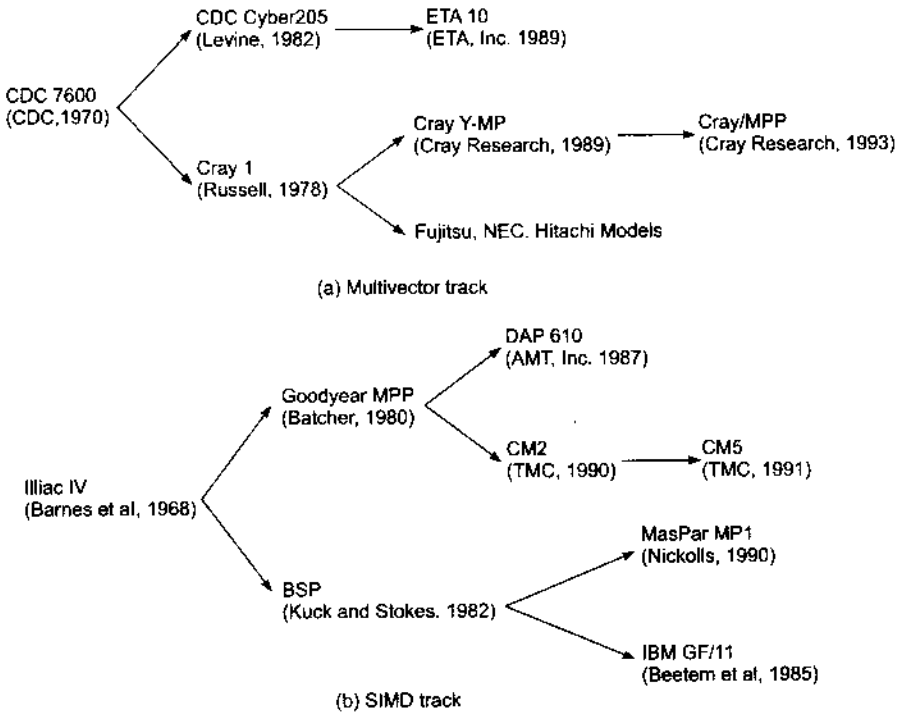


Fig. 1.18 Multivector and SIMD tracks

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.

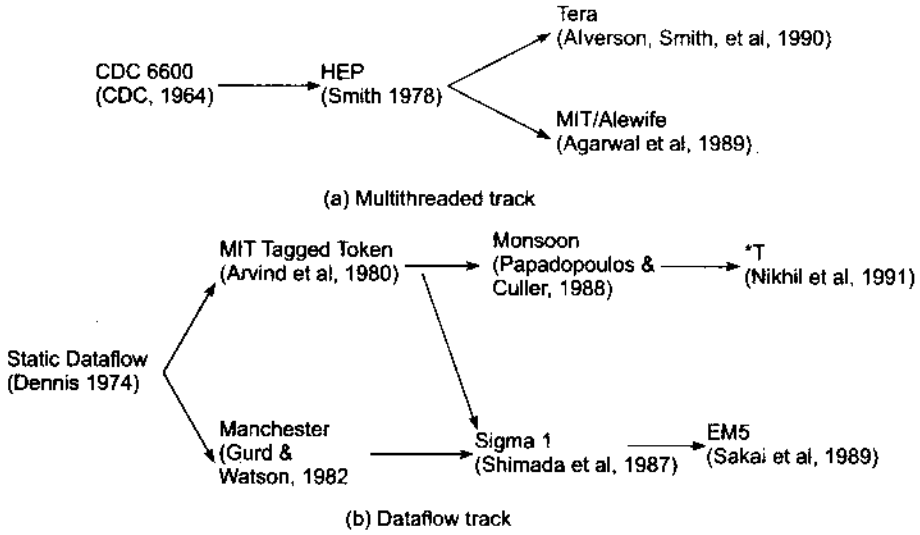


Fig. 1.19 Multithreaded and dataflow tracks

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 $1 \leq p \leq n^3/\log n$. Choose an appropriate PRAM machine model to prove that the matrix multiplication can be done in $T = O(n^3/p)$ time.

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

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

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

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

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

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

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

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

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

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

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

2

Program and Network Properties

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

2.1 CONDITIONS OF PARALLELISM

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

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

2.1.1 Data and Resource Dependences

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

We use a *dependence graph* to describe the relations. The nodes of a dependence graph correspond to the program statements (instructions), and the directed edges with different labels show the ordered relations among the statements. The analysis of dependence graphs shows where opportunity exists for parallelization and vectorization.

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

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

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

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

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

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

- The subscript of a variable is itself subscribed.
- The subscript does not contain the loop index variable.
- A variable appears more than once with subscripts having different coefficients of the loop variable.
- The subscript is nonlinear in the loop index variable.

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

Example 2.1 Data dependence in programs

Consider the following code fragment of four instructions:

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