**Sengamala Thayaar Educational Trust Women's College**
(Affiliated to Bharathidasan University, Tiruchirapalli)
(Accredited with 'A' Grade {3.45/4.00} By NAAC)
(An ISO 9001: 2015 Certified Institution)
SUNDARAKKOTTAI,MANNARGUDI-614016
Thiruvarur (Dt.), Tamil Nadu, India.

**EC-III**

# ADVANCED COMPUTER ARCHITECTURE

## II M.Sc., Computer Science
## Semester – III

**N.SUBHALAKSHMI M.Sc., M.Phil., (Ph.D.,)**
**ASSISTANT PROFESSOR,**
**PG & RESEARCH DEPARTMENT OF COMPUTER SCIENCE**

# SYLLABUS

## ELECTIVE COURSE III

## ADVANCED COMPUTER ARCHITECTURE

**Objectives:** To study the advanced computer Architecture, theories of parallel computing, network properties and applications of cost effective computer systems to meet the above requirements.

**UNIT I**

Parallel computer models :- The state of computing - Multiprocessors and multicomputers – Multivector and SIMD computers.

**UNIT II**

Program and Network properties:- Conditions of parallelism – Program partitioning and scheduling – program flow mechanisms – system interconnect architectures.

**UNIT III**

Processors and memory hierarchy :- Advanced processor Technology – Super scalar and vector processors – Linear Pipeline Processors – Nonlinear pipeline Processors.

**UNIT IV**

Multiprocessors and Multicomputers:- Multiprocessor System interconnects – Message Passing Mechanisms – SIMD Computer Organizations – The Connection Machine CM 5 – Fine-Grain Multicomputers.

**UNIT V**

Software for Parallel Programming:- Parallel Programming Models – Parallel Languages and Compilers – Dependence Analysis of Data Arrays.

**Text Book**

1. Kai Hwang, "Advanced Computer Architecture "McGraw-Hill International Edn., Singapore , 1993. Chapters 1.1-1.3, 2, 4.1, 4.2, 6.2, 7.1, 7.4, 8 4, 8.5, 10.1, 10.2, 10.3

**Reference Books:**

1. Kai Hwang and Faye A.Briggs, "Computer Architecture and Parallel Processing", McGraw-Hill International Editions, Singapore , 1985.
2. Michael J.Quinn, "Parallel Computing, Theory and Practice", McGraw-Hill International Edn., Singapore , 1994. *****

# UNIT – I

## CHAPTER -I
## PARALLEL COMPUTER MODELS

# I. The state of computing

### 1.1. Computer Development Milestones

### 1.2. Elements of Computers

### 1.3. Evolution of Computer architecture

### 1.4. System Attributes to Performance

## Introduction

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 computer, vector supercomputers, Multiprocessors, multicomputer and massively parallel processors.

# I. The state of computing

Modern computers arc 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. We then examine the crucial hardware and software elements built into modern computer systems, the evolutional relations in milestone architectural development. Basic hardware and software factors are identified in analyzing the performance of computers.

## 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. Modem 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/0 processors, multiplexed memory access. | HLL used with compilers, subroutine libraries, batch processing monitor. | **IBM 7090,** CDC 1604, Univac LARC. |
| Third (1965-74) | Integrated circuits (SSV-MSI), microprogramming, pipclining, cache, and lookahead processors. | Multiprogramming and time-sharing OS, multiuser applications. | IBM 360/370, CDC 6600. Tl-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* ([Cs) 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 EN1AC (Electronic Numerical Integrator and Calculator) built at the Moore School of the University of Pennsylvania in 1950; the 1AS (Institute for Advanced Studies) computer based on a design proposed by John von Neumann, Arthur Burks, and Hen tan 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 1/0 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 arc 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.

## 1.2 Elements of Modern Computers

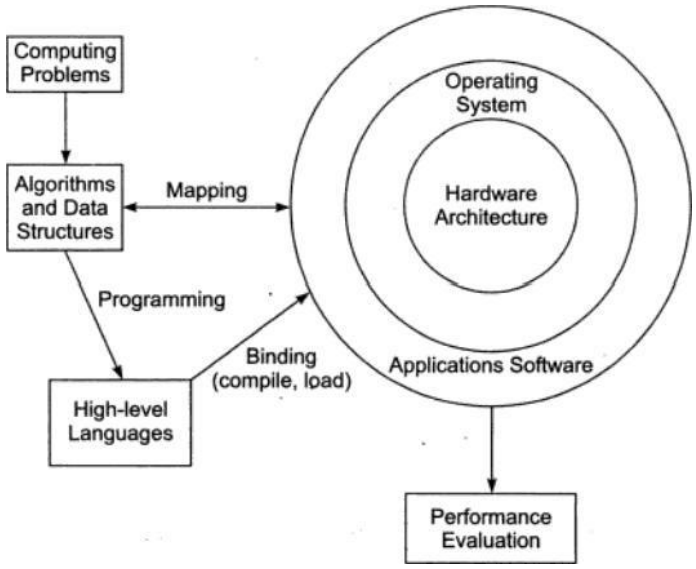
Hardware, software, and programming elements of a modern computer system are briefly introduced below in the context of parallel processing.

**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.

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

**Fig.1.1 Elements of a modern computer system**



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

For **artificial intelligence (Al) problems**, the solutions demand logic inferences and symbolic mampulatIons. 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.

***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.

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.

## 1.3 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.

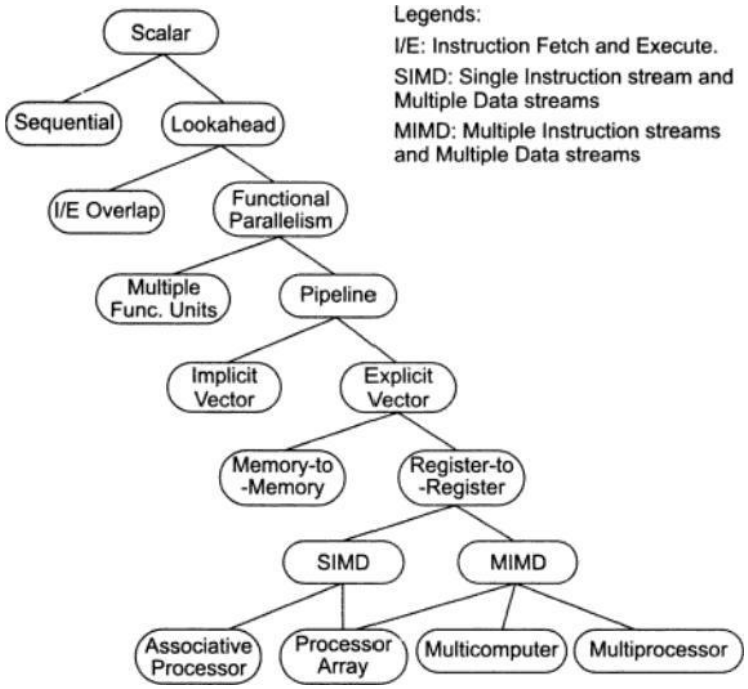



**Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers**

### *Lookahead, Parallelism, and Pipelining*

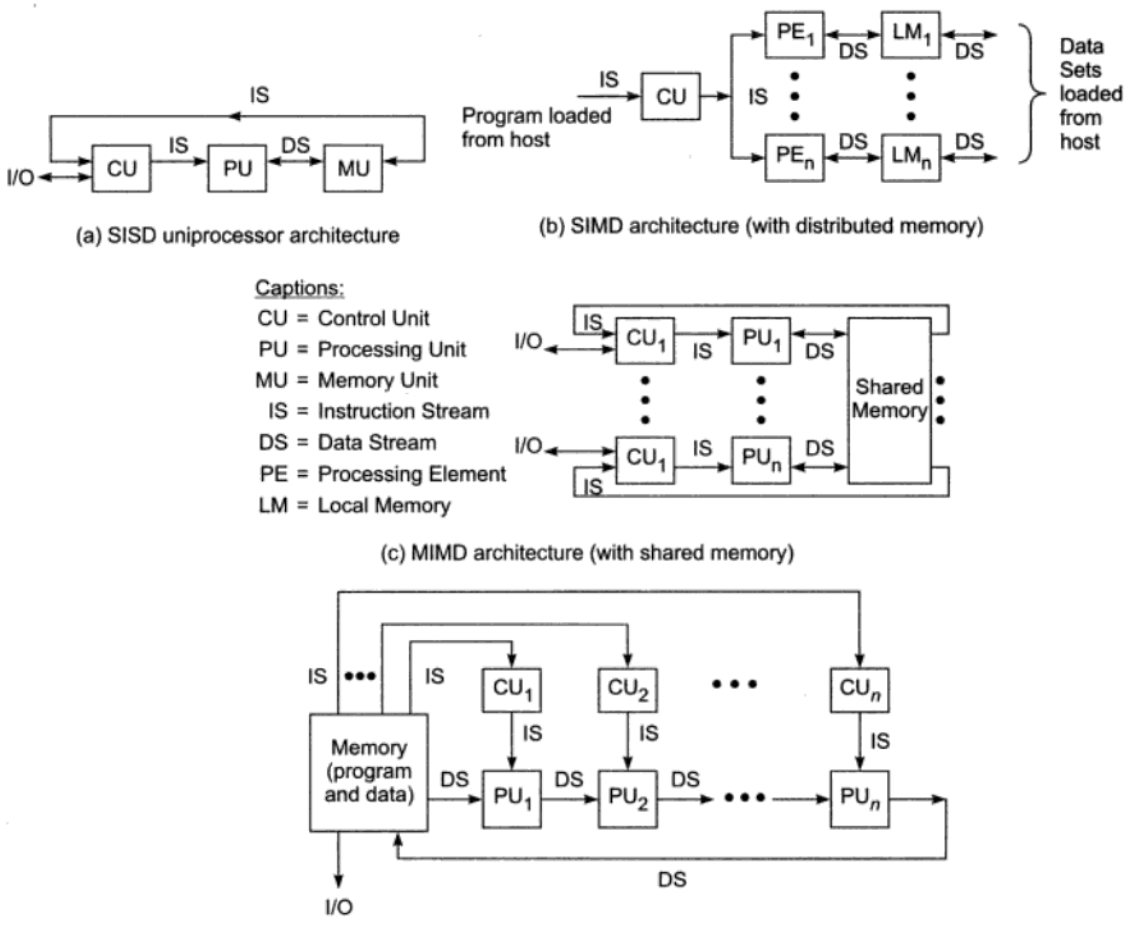
Lookahead techniques were introduced to prefetch instructions in order to overlap I/E (instruction fetch/decode and execution) operations and to enable functional parallelism. Functional parallelism was supported by two approaches: One is to use multiple functional units simultaneously, and the other 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 MI MD *(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.




(a) SISD uniprocessor architecture

(b) SIMD architecture (with distributed memory)

(c) MIMD architecture (with shared memory)

(d) MISD architecture (the systolic array)

**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

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.

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. 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.

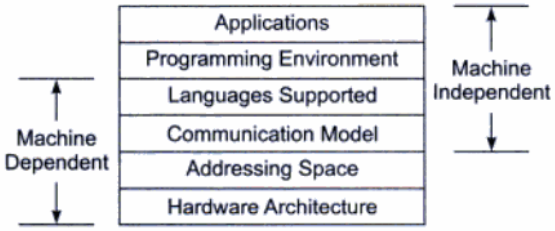



**Fig. 1.4** Six layers for computer system development (Courtesy of Lionel Ni, 1990)

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.

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.

## 1.4 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* r. The inverse of the cycle time is the *clock rate (f=* lir). The size of a program is determined by its *instruction count (4), 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 1. 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:

$$T = \text{x CPI x r} \qquad (1.1)$$

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 *T.* 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:

$$T = 4x(p + mxk)xr \qquad (1.2)$$

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, is the instruction count, and r 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 (4, *p, m, k,* 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 (4) and processor cycles needed *(p).* The compiler technology affects the values of/$_{L}$., *p,* and the memory reference count (m). The CPU implementation and control determine the total processor time *(p • T)* needed. Finally, the memory technology and hierarchy design affect the memory access latency *(k* • r). The above CPU time can be used as a basis in estimating the execution rate of a processor.

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

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:

$$W_p = \frac{f}{I_c \times \text{CPI}} \quad (1.4)$$

Note that $W_p$ = (MIPS) x $10^6$/4 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 (4). 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.

---

### *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. I .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.

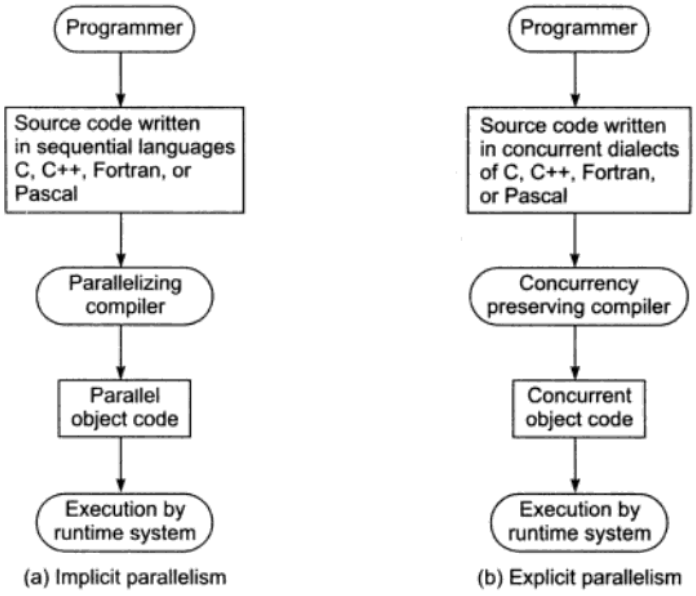



**Fig. 1.5** Two approaches to parallel programming (Courtesy of Charles Seitz; adapted with permission from "Concurrent Architectures", p. 51 and p. 53, *VLSI and Parallel Computation*, edited by Suaya and Birtwistle, Morgan Kaufmann Publishers, 1990)

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.

# II . MULTIPROCESSORS AND MULTICOMPUTERS

## 2.1 Shared-Memory Multiprocessors

## 2.2 Distributed-Memory Multicomputers

## 2.3 A Taxonomy of MIMD Computers

# II . MULTIPROCESSORS AND MULTICOMPUTERS

Two categories of parallel computers arc architecturally modeled below. These physical models are distinguished by having a shared common memory or unshared distributed memories

## 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.

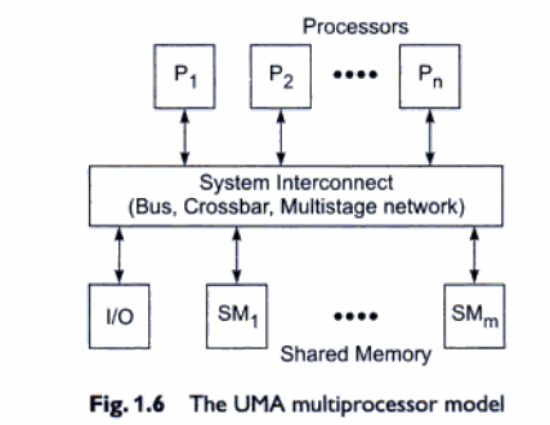



**Fig. 1.6** The UMA multiprocessor model

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.

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.

---

### *The NUMA Model*

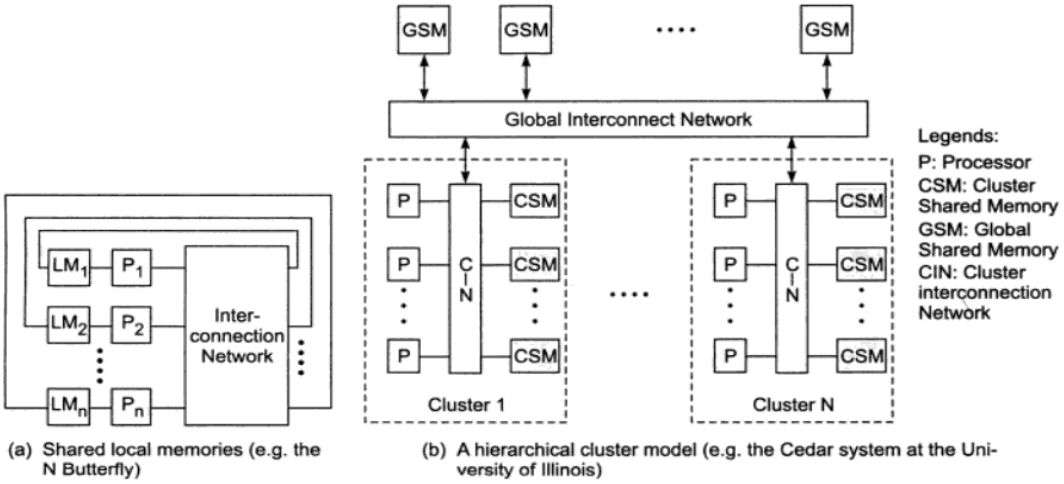
A NUMA multiprocessor is a shared-memory system in which the access time varies with the location of the memory word. Two NUMA machine models are depicted in Fig. 1.7. The shared memory is physically distributed to all processors, called local memories. The collection of all *local memories* forms a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

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 module*.

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.

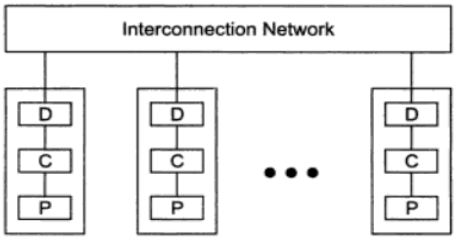



**Fig. 1.8** The COMA model of a multiprocessor (P: Processor, C: Cache, D: Directory; e.g. the KSR-1)

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.

***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 mini supercomputers. 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/Sequcnt preprocessor, transaction multiprocessing.** | **Latter models designed with faster processors of the family.** |
| **IBM ES/9000 Model 900NF** | **6 ES/9000 processors with vector facilities, crossbar connected to I/0 channels and shared memory.** | **OS support: MVS, VM KMS, ADC/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.** |

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.

## 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.

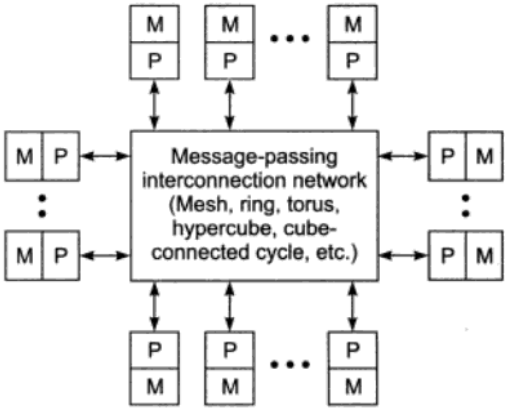



**Fig. 1.9** Generic model of a message-passing multicomputer

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 1/0 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 1PSC/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.

we will study various static network topologies used to construct multicomputers. Commonly used topologies include the *ring, tree, mesh, torus, hypetrube, 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 Muhicomputers*

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 Mutticomputer* Systems

| *System Features* | *Intel Paragon XP/S* | *nCUBE/2 6480* | *Parsys Ltd. SuperNode 1000* |
|---|---|---|---|
| Node Types and Memory | 50 MHz i860 XP computing nodes with 16-128 Mbytes per node, special 1/0 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 multiple1-800 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, 5I2-Gbyte memory, 64 I/0 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 Ill where the tradeoffs between scalability and programmability are analyzed.

## 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 M1MD architecture, away from the SIMD architecture in CM-2, represents the architectural trend 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. 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.

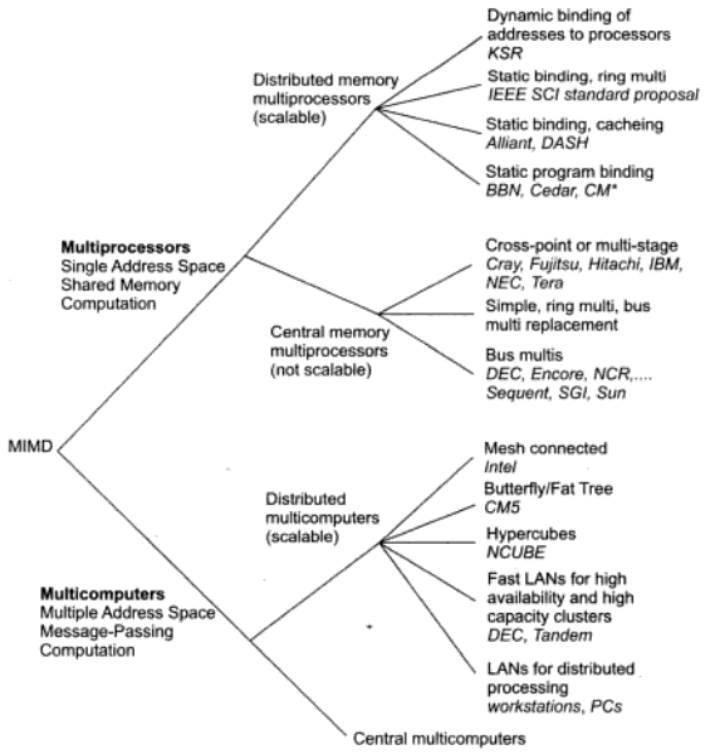



**Fig. 1.10** Bell's taxonomy of MIMD computers (Courtesy of Gordon Bell; reprinted with permission from the *Communications of ACM*, August 1992)



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.

# III. MULTIVECTOR AND SIMD COMPUTERS

3.1 Vector Supercomputers

3.2 SIMD Supercomputers

# III. 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.

## 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.

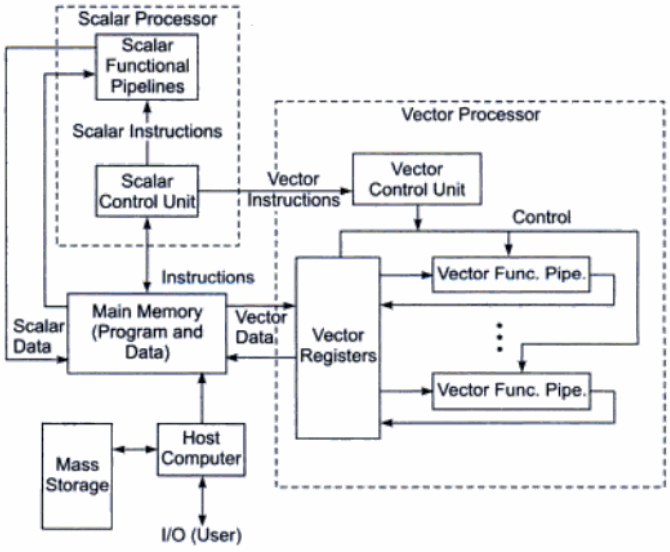



**Fig. 1.11** The architecture of a vector supercomputer

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.

### *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 Han!ware 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 x 64 x 64 vector registers. Pipeline chaining possible.** | **MS or ULTRDUOS. 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.

The Convex CI 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.

## 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).

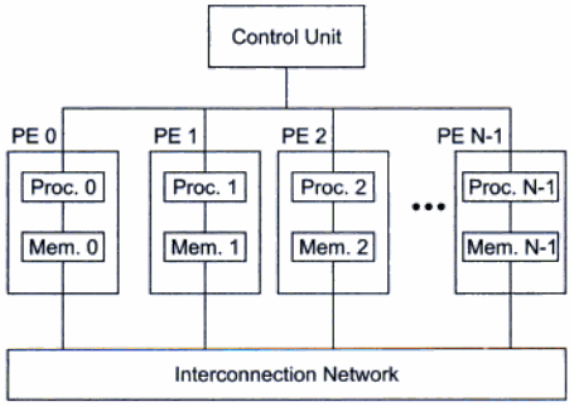



**Fig. 1.12** Operational model of SIMD computers

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

$$M = (N, C, I, M, R) \quad (1.5)$$

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) / 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.

---

***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 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 x $2^{12}$ = $2^{16}$ = 65,536 PEs formed the entire SIMD array.

The DAP610 implemented 64 PEs as a mesh on a chip. Globally, a large mesh (64 x 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 SAID 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 x 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.** |