8

Multivector and SIMD Computers

By definition, supercomputers are the fastest computers available at any specific time. The value of supercomputing was originally identified by Buzbee (1983) in three areas: *knowledge acquisition*, *computational tractability*, and *promotion of productivity*. Computing demand, however, is always ahead of computer capability. Today's supercomputers are still one generation behind the computing requirements in most application areas, which have expanded enormously over the last two decades.

In this chapter, we study the architectures of pipelined multivector supercomputers and of SIMD array processors. Both types of machines perform vector processing over large volumes of data. Besides discussing basic vector processors, we describe compound vector functions and multipipeline chaining and networking techniques for developing higher-performance vector multiprocessors.

The evolution from SIMD and MIMD computers to hybrid SIMD/MIMD computer systems is also considered. The Connection Machine CM-5 reflected this architectural trend. This hybrid approach to designing reconfigurable computers opened up new opportunities for exploiting coordinated parallelism in complex application problems. Recent trends in this direction will be discussed in Chapter 13.

8.1 VECTOR PROCESSING PRINCIPLES

Vector instruction types, memory-access schemes for vector operands, and an overview of supercomputer families are given in this section.

8.1.1 Vector Instruction Types

Basic concepts behind vector processing are defined below. Then we discuss major types of vector instructions encountered in a typical vector processor. The intent is to acquaint the reader with the instruction-set architectures of typical vector processors.

Vector Processing Definitions A *vector* is an ordered set of scalar data items, all of the same type, stored in memory. Usually, the vector elements are ordered to have a fixed addressing increment between successive elements, called the *stride*.

A *vector processor* is an ensemble of hardware resources, including vector registers, functional pipelines, processing elements, and register counters, for performing vector operations. *Vector processing* occurs when arithmetic or logical operations are applied to vectors. It is distinguished from scalar processing which operates on one datum or one pair of data. The conversion from scalar code to vector code is called *vectorization*.

In general, vector processing is faster and more efficient than scalar processing. Both pipelined processors and SIMD computers can perform vector operations. Vector processing reduces software overhead incurred in the maintenance of looping control, reduces memory-access conflicts, and above all matches nicely with the pipelining and segmentation concepts to generate one result per clock cycle continuously.

Depending on the *speed ratio* between vector and scalar operations (including startup delays and other overheads) and on the *vectorization ratio* in user programs, a vector processor executing a well-vectorized code can easily achieve a speedup of 10 to 20 times, as compared with scalar processing on conventional machines.

Of course, the enhanced performance comes with increased hardware and compiler costs, as expected. A compiler capable of vectorization is called a *vectorizing compiler* or simply a *vectorizer.* For successful vector processing, one needs to make improvements in vector hardware, vectorizing compilers, and programming skills specially targeted at vector machines.

Vector Instruction Types We briefly introduced basic vector instructions in Chapter 4. What are characterized below are vector instructions for register-based, pipelined vector machines. Six types of vector instructions are illustrated in Figs. 8.1 and 8.2. We define these vector instruction types by mathematical mappings between their working registers or memory where vector operands are stored.

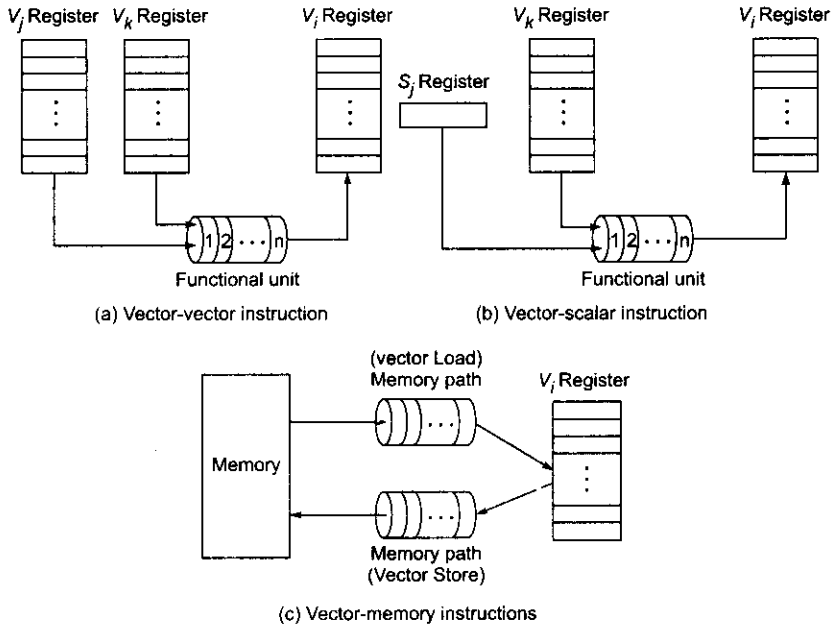


Fig. 8.1 Vector instruction types in Cray-like computers

(1) *Vector-vector instructions* As shown in Fig. 8.1a, one or two vector operands are fetched from the respective vector registers, enter through a functional pipeline unit, and produce results in another vector register. These instructions are defined by the following two mappings:

$$f_1 : V_j \rightarrow V_i \tag{8.1}$$

$$f_2 : V_j \times V_k \rightarrow V_i \tag{8.2}$$

Examples are $V_1 = \sin(V_2)$ and $V_3 = V_1 + V_2$ for the mappings f_1 and f_2, respectively, where V_i for $i = 1, 2$, and 3 are vector registers.

(2) *Vector-scalar instructions* Figure 8.1b shows a vector-scalar instruction corresponding to the following mapping:

$$f_3 : s \times V_k \rightarrow V_i \tag{8.3}$$

An example is a scalar product $s \times V_1 = V_2$, in which the elements of V_1 are each multiplied by a scalar s to produce vector V_2 of equal length.

(3) *Vector-memory instructions* This corresponds to vector load or vector store (Fig. 8.1c), element by element, between the vector register (V) and the memory (M) as defined below:

$$f_4 : M \rightarrow V \qquad \textit{Vector load} \tag{8.4}$$
$$f_5 : V \rightarrow M \qquad \textit{Vector store} \tag{8.5}$$

(4) *Vector reduction instructions* These correspond to the following mappings:

$$f_6 : V_i \rightarrow s \tag{8.6}$$
$$f_7 : V_i \times V_j \rightarrow s \tag{8.7}$$

Examples of f_6 include finding the *maximum*, *minimum*, *sum*, and *mean value* of all elements in a vector. A good example of f_7 is the *dot product*, which performs $s = \sum_{i=1}^{n} a_i \times b_i$ from two vectors $A = (a_i)$ and $B = (b_i)$.

(5) *Gather and scatter instructions* These instructions use two vector registers to gather or to scatter vector elements randomly throughout the memory, corresponding to the following mappings:

$$f_8 : M \rightarrow V_1 \times V_0 \qquad \textit{Gather} \tag{8.8}$$
$$f_9 : V_1 \times V_0 \rightarrow M \qquad \textit{Scatter} \tag{8.9}$$

Gather is an operation that fetches from memory the nonzero elements of a sparse vector using indices that themselves are indexed. *Scatter* does the opposite, storing into memory a vector in a sparse vector whose nonzero entries are indexed. The vector register V_1 contains the data, and the vector register V_0 is used as an index to gather or scatter data from or to random memory locations as illustrated in Figs. 8.2a and 8.2b, respectively.

(6) *Masking instructions* This type of instruction uses a *mask vector* to compress or to expand a vector to a shorter or longer index vector, respectively, corresponding to the following mappings:

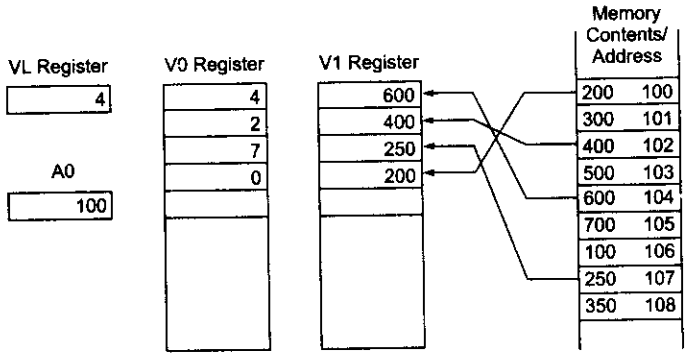
$$f_{10} : V_0 \times V_m \rightarrow V_1 \tag{8.10}$$

The following example will clarify the meaning of *gather*, *scatter*, and *masking* instructions.

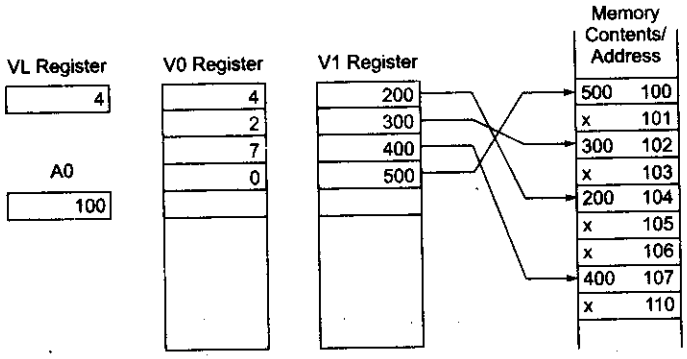
Example 8.1 Gather, scatter, and masking instructions in the Cray Y-MP (Cray Research, 1990)

The gather instruction (Fig. 8.2a) transfers the contents (600, 400, 250, 200) of nonsequential memory locations (104, 102, 107, 100) to four elements of a vector register $V1$. The base address (100) of the memory is indicated by an address register $A0$. The number of elements being transferred is indicated by the contents (4) of a *vector length* register VL.

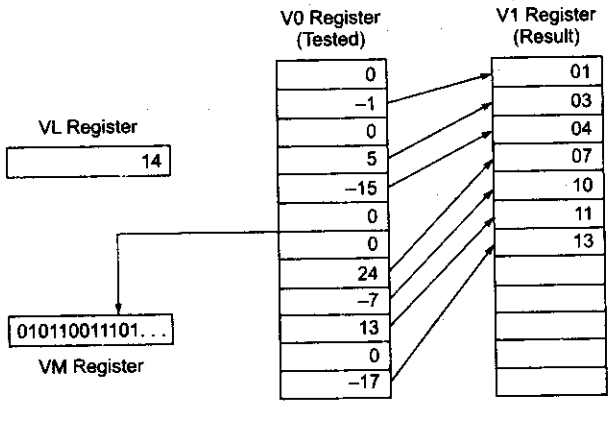
The offsets (indices) from the base address are retrieved from the vector register *V0*. The effective memory addresses are obtained by adding the base address to the indices.



(a) Gather instruction



(b) Scatter instruction



(c) Masking instruction

Fig. 8.2 Gather, scatter and masking operations on the Cray Y-MP (Courtesy of Cray Research, 1990)

The scatter instruction reverses the mapping operations, as illustrated in Fig. 8.2b. Both the *VL* and *A0* registers are embedded in the instruction.

The masking instruction is shown in Fig. 8.2c for compressing a long vector into a short index vector. The contents of vector register *V0* are tested for zero or nonzero elements. A *masking register* (*VM*) is used to store the test results. After testing and forming the *masking vector* in *VM*, the corresponding nonzero indices are stored in the *V1* register. The *VL* register indicates the length of the vector being tested.

The *gather*, *scatter*, and *masking* instructions are very useful in handling sparse vectors or sparse matrices often encountered in practical vector processing applications. Sparse matrices are those in which most of the entries are zeros. Advanced vector processors implement these instructions directly in hardware.

The above instruction types cover the most important ones. A given specific vector processor may implement an instruction set containing only a subset or even a superset of the above instructions.

8.1.2 Vector-Access Memory Schemes

The flow of vector operands between the main memory and vector registers is usually pipelined with multiple access paths. In this section, we specify vector operands and describe three vector-access schemes from interleaved memory modules allowing overlapped memory accesses.

Vector Operand Specifications Vector operands may have arbitrary length. Vector elements are not necessarily stored in contiguous memory locations. For example, the entries in a matrix may be stored in row major or in column major order. Each row, column, or diagonal of the matrix can be used as a vector.

When row elements are stored in contiguous locations with a unit stride, the column elements are stored with a stride of n, where n is the matrix order. Similarly, the diagonal elements are also separated by a stride of $n + 1$.

To access a vector in memory, one must specify its *base address*, *stride*, and *length*. Since each vector register has a fixed number of component registers, only a segment of the vector can be loaded into the vector register in a fixed number of cycles. Long vectors must be segmented and processed one segment at a time.

Vector operands should be stored in memory to allow pipelined or parallel access. The memory system for a vector processor must be specifically designed to enable fast vector access. The access rate should match the pipeline rate. In fact, the access path is often itself pipelined and is called an *access pipe*. These vector-access memory organizations are described below.

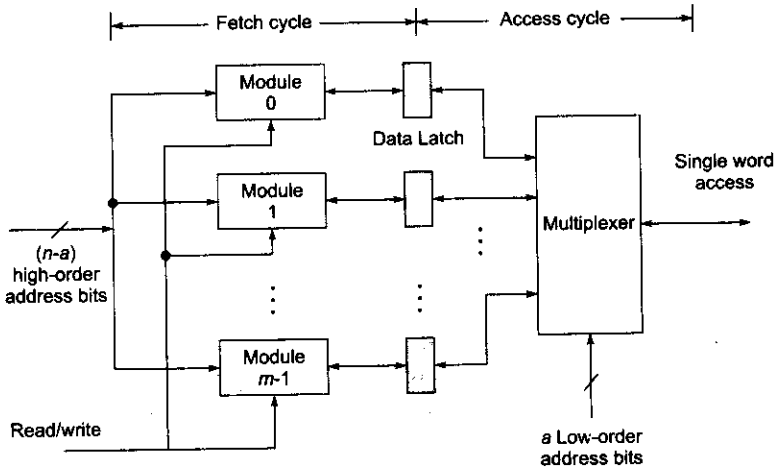
C-Access Memory Organization The m-way low-order interleaved memory structure shown in Figs. 5.15a and 5.16 allows m memory words to be accessed concurrently in an overlapped manner. This *concurrent access* has been called *C-access* as illustrated in Fig. 5.16b.

The access cycles in different memory modules are staggered. The low-order a bits select the modules, and the high-order b bits select the word within each module, where $m = 2^a$ and $a + b = n$ is the address length.

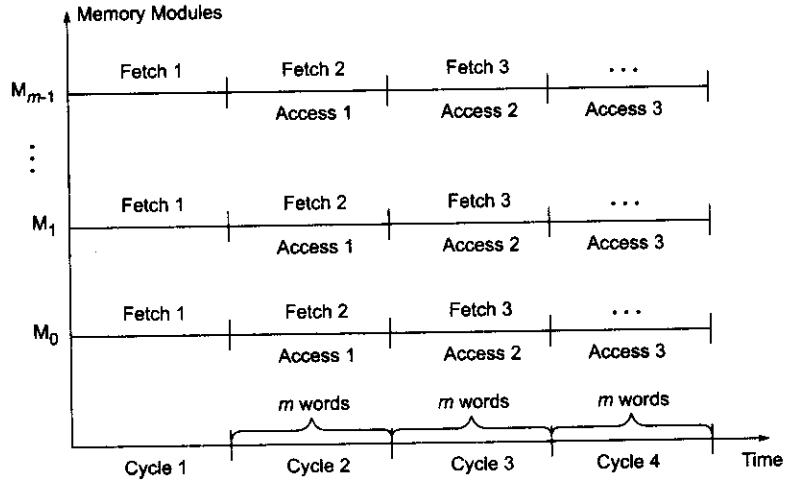
To access a vector with a stride of 1, successive addresses are latched in the address buffer at the rate of one per cycle. Effectively it takes m minor cycles to fetch m words, which equals one (major) memory cycle as stated in Eq. 5.4 and Fig. 5.16b.

If the stride is 2, the successive accesses must be separated by two minor cycles in order to avoid access conflicts. This reduces the memory throughput by one-half. If the stride is 3, there is no module conflict and the maximum throughput (m words) results. In general, C-access will yield the maximum throughput of m words per memory cycle if the stride is relatively prime to m, the number of interleaved memory modules.

S-Access Memory Organization The low-order interleaved memory can be rearranged to allow *simultaneous access*, or *S-access*, as illustrated in Fig. 8.3a. In this case, all memory modules are accessed simultaneously in a synchronized manner. Again the high-order $(n - a)$ bits select the same offset word from each module.



(a) S-access organization for an m-way interleaved memory



(b) Successive vector accesses using overlapped fetch and access cycles

Fig. 8.3 The S-access interleaved memory for vector operands access

At the end of each memory cycle (Fig. 8.3b), $m = 2^a$ consecutive words are latched in the data buffers simultaneously. The low-order a bits are then used to multiplex the m words out, one per each minor cycle.

If the minor cycle is chosen to be $1/m$ of the major memory cycle (Eq. 5.4), then it takes two memory cycles to access m consecutive words.

However, if the access phase of the last access is overlapped with the fetch phase of the current access (Fig. 8.3b), effectively m words take only one memory cycle to access. If the stride is greater than 1, then the throughput decreases, roughly proportionally to the stride.

C/S-Access Memory Organization A memory organization in which the C-access and S-access are combined is called *C/S-access*. This scheme is shown in Fig. 8.4, where n access buses are used with m interleaved memory modules attached to each bus. The m modules on each bus are m-way interleaved to allow C-access. The n buses operate in parallel to allow S-access. In each memory cycle, at most $m \cdot n$ words are fetched if the n buses are fully used with pipelined memory accesses.

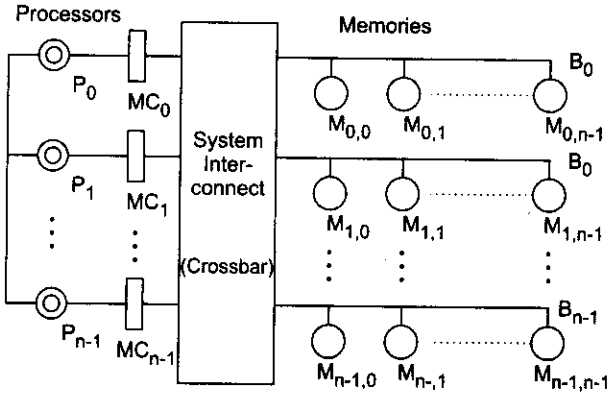


Fig. 8.4 The C/S memory organization with $m = n$. (Courtesy of D.K. Panda, 1990)

The C/S-access memory is suitable for use in vector multiprocessor configurations. It provides parallel pipelined access of a vector data set with high bandwidth. A special *vector cache* design is needed within each processor in order to guarantee smooth data movement between the memory and multiple vector processors.

8.1.3 Early Supercomputers

This section introduces five early supercomputer families, including the Cray Research series, the CDC/ETA series, the Fujitsu VP series, the NEC SX series, and the Hitachi 820 series (Table 8.1). The relative performance of these machines for vector processing are compared with scalar processing at the end.

The Cray Research Series Seymour Cray founded Cray Research, Inc. in 1972. Since then, hundreds units of Cray supercomputers have been produced and installed worldwide. As we shall see in Chapter 13, the company has gone through a change of name and evolution of product line.

The Cray 1 was introduced in 1975. An enhanced version, the Cray 1S, was produced in 1979. It was the first ECL-based supercomputer with a 12.5-ns clock cycle. High degrees of pipelining and vector processing were the major features of these machines.

Table 8.1 *Summary of Early Supercomputers*

System model	*Maximum configuration, clock rate, OS/Compiler*	*Unique features and remarks*
Cray 1S	Uniprocessor with 10 pipelines, 12.5 ns, COS/CF77 2.1.	First ECL-based super, introduced in 1976.
Cray 2S /4-256	4 processors with 256M-word memory, 4.1 ns, COS or UNIX/CF77 3.0.	16K-word local memory, ported UNIX V introduced in 1985.
Cray X-MP 416	4 processors with 16M-word memory, and 128M-word SSD, 8.5 ns, COS CF77 5.0.	Using shared register clusters for IPC, introduced in 1983.
Cray Y-MP 832	8 processors with 128M-word memory, 6 ns, CF77 5.0.	Enhanced from X-MP, introduced in 1988.
Cray Y-MP C-90	16 processors with 2 vector pipes per processor, 4.2 ns, UNICOS/CF 77 5.0.	The largest Cray machine, introduced in 1991.
CDC Cyber 205	Uniprocessor with 4 pipelines, 20 ns, virtual OS/FTN 200.	Memory-to-memory architecture, introduced in 1982.
ETA 10 E	Uniprocessor with 10.5 ns, ETAV/FTN 200	Successor to Cyber 205, introduced in 1985.
NEC SX-X/44	4 processors with 4 sets of pipelines per processor, 2.9 ns, F77SX.	Succeeded by SX-X Series, introduced in 1991.
Fujitsu VP2600/10	Uniprocessor with 5 vector pipes and dual scalar processors, 3.2 ns, MSP-EX/F77 EX/VP.	Used reconfigurable vector registers and masking, introduced in 1991.
Hitachi 820/80	18 functional pipelines in a uniprocessor with 512 Mbytes memory 4 ns, FORT 77/HAP V23-OC.	Introduced in 1987 with 64 I/O channels providing a maximum of 288 Mbytes/s transfer.

Ten functional pipelines could run simultaneously in the Cray 1S to achieve a computing power equivalent to that of 10 IBM 3033's or CDC Cyber 7600's. Only batch processing with a single user was allowed when the Cray 1 was initially introduced using the Cray Operating System (COS) with a Fortran 77 compiler (CF 77 Version 2.1).

The Cray X-MP Series introduced multiprocessor configurations in 1983. Steve Chen led the effort at Cray Research in developing this series using one to four Cray 1-equivalent CPUs with shared memory. A unique feature introduced with the X-MP models was shared register clusters for fast interprocessor communications without going through the shared memory.

Besides 128 Mbytes of shared memory, the X-MP system had 1 Gbyte of *solid-state storage* (SSD) as extended shared memory. The clock rate was also reduced to 8.5 ns. The peak performance of the X-MP-416 was 840 Mflops when eight vector pipelines for add and multiply were used simultaneously across four processors.

The successor to the Cray X-MP was the Cray Y-MP introduced in 1988 with up to eight processors in a single system using a 6-ns clock rate and 256 Mbytes of shared memory.

The Cray Y-MP C-90 was introduced in 1990 to offer an integrated system with 16 processors using a 4.2-ns clock. We will study models Y-MP 816 and C-90 in detail in the next section.

Another product line was the Cray 2S introduced in 1985. The system allowed up to four processors with 2 Gbytes of shared memory and a 4.1-ns clock. A major contribution of the Cray 2 was to switch from the batch processing COS to multiuser UNIX System V on a supercomputer. This led to the UNICOS operating system, derived from the UNIX/V and Berkeley 4.3 BSD, variants of which are currently in use in some Cray computer systems.

The Cyber/ETA Series Control Data Corporation (CDC) introduced its first supercomputer, the STAR-100, in 1973. Cyber 205 was the successor produced in 1982. The Cyber 205 ran at a 20-ns clock rate, using up to four vector pipelines in a uniprocessor configuration.

Different from the register-to-register architecture used in Cray and other supercomputers, the Cyber 205 and its successor, the ETA 10, had memory-to-memory architecture with longer vector instructions containing memory addresses.

The largest ETA 10 consisted of 8 CPUs sharing memory and 18 I/O processors. The peak performance of the ETA 10 was targeted for 10 Gflops. Both the Cyber and the ETA Series are no longer in production but were in use for many years at several supercomputer centers.

Japanese Supercomputers NEC produced the SX-X Series with a claimed peak performance of 22 Gflops in 1991. Fujitsu produced the VP-2000 Series with a 5-Gflops peak performance at the same time. These two machines used 2.9- and 3.2-ns clocks, respectively.

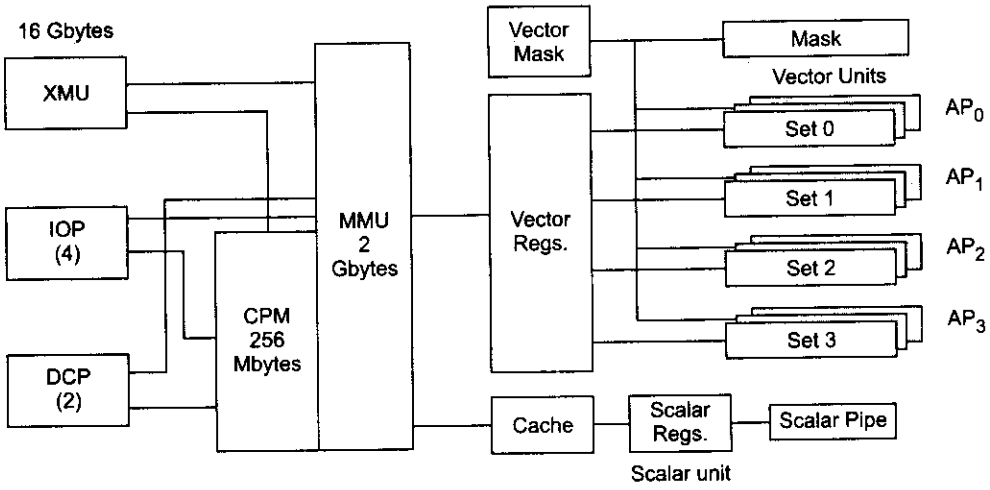
Shared communication registers and reconfigurable vector registers were special features in these machines. Hitachi offered the 820 Series providing a 3-Gflops peak performance. Japanese supercomputers were at one time strong in high-speed hardware and interactive vectorizing compilers.

The NEC SX-X 44 NEC claimed that this machine was the fastest vector supercomputer (22 Gflops peak) ever built up to 1992. The architecture is shown in Fig. 8.5. One of the major contributions to this performance was the use of a 2.9-ns clock cycle based on VLSI and high-density packaging.

There were four arithmetic processors communicating through either the shared registers or via the shared memory of 2 Gbytes. There were four sets of vector pipelines per processor, each set consisting of two add/shift and two multiply/logical pipelines. Therefore, 64-way parallelism was obtained with four processors, similar to that in the C-90.

Besides the vector unit, a high-speed scalar unit employed RISC architecture with 128 scalar registers. Instruction reordering was supported to exploit higher parallelism. The main memory was 1024-way interleaved. The extended memory of up to 16 Gbytes provided a maximum transfer rate of 2.75 Gbytes/s.

A maximum of four I/O processors could be configured to accommodate a 1-Gbyte/s data transfer rate per I/O processor. The system could provide a maximum of 256 channels for high-speed network, graphics, and peripheral operations. The support included 100-Mbytes/s channels.



Captions:
XMU: Extended memory unit
IOP: I/O processors (4)
DCP: Data central processors (2)
AP: Arithmetic processors (4)
MMU: Main memory unit
CPM: Data central processor memory
Each set consists of 4 pipelines for add/shift and multiply/logical vector operations

Fig. 8.5 The NEC SX-X 44 vector supercomputer architecture (Courtesy of NEC, 1991)

Relative Vector/Scalar Performance Let r be the vector/scalar speed ratio and f the vectorization ratio. By Amdahl's law in Section 3.3.1, the following *relative performance* can be defined:

$$P = \frac{1}{(1-f)+f/r} = \frac{r}{(1-f)r+f} \tag{8.11}$$

This relative performance indicates the speedup performance of vector processing over scalar processing. The hardware speed ratio r is the designer's choice. The vectorization ratio f reflects the percentage of code in a user program which is vectorized.

The relative performance is rather sensitive to the value of f. This value can be increased by using a better vectorizing compiler or through user program transformations. The following example shows the IBM experience in vector processing with the 3090/VF computer system.

Example 8.2 The vector/scalar relative performance of the IBM 3090/VF

Figure 8.6 plots the relative performance P as a function of r with f as a running parameter. The higher the

value of f, the higher the relative speedup. The IBM 3090 with vector facility (VF) was a high-end mainframe with add-on vector hardware.

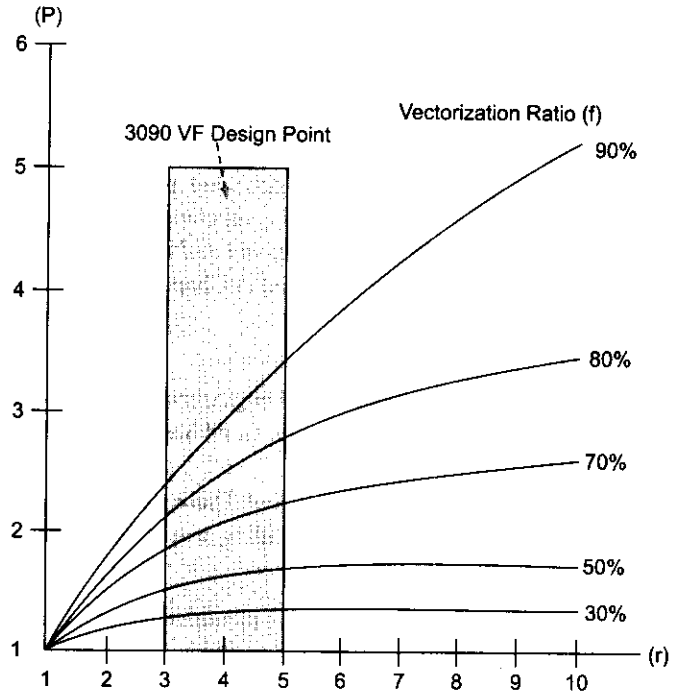


Fig. 8.6 Speedup performance of vector processing over scalar processing in the IBM 3090/VF (Courtesy of IBM Corporation, 1986)

The designers of the 3090/VF chose a speed ratio in the range $3 \leq r \leq 5$ because IBM wanted a balance between business and scientific applications. When the program is 70% vectorized, one expects a maximum speedup of 2.2. However, for $f \leq 30\%$, the speedup is reduced to less than 1.3.

The IBM designers did not choose a high speed ratio because they did not expect user programs to be highly vectorizable. When f is low, the speedup cannot be high, even with a very high r. In fact, the limiting case is $P \rightarrow 1$ if $f \rightarrow 0$.

On the other hand, $P \rightarrow r$ when $f \rightarrow 1$. Scientific supercomputer designers like Cray and Japanese manufacturers often chose a much higher speed ratio, say, $10 \leq r \leq 25$, because they expected a higher vectorization ratio f in user programs, or they used better vectorizers to increase the ratio to a desired level.

Huge advances have taken place in the underlying technologies—and especially in VLSI technology—over the last two decades. We shall see that these advances, summarized in brief in Chapter 13, have defined the direction of advances in computer architecture over this period. Powerful single-chip processors—as also multi-core *systems-on-a-chip*—provide *High Performance Computing* (HPC) today. Such HPC systems typically make use of MIMD and/or SPMD configurations with a large number of processors.

Advent of superscalar processors has resulted in vector processing instructions being built into powerful processors, rather than as specialized processors. Thus the ideas we have studied in this section have made

their appearance in capabilities such as *Streaming SIMD Extensions* (SSE) in processors (see Chapter 13). We may say that the *concepts* of vector processing remain valid today, but their *implementation* varies with advances in technology.

8.2 MULTIVECTOR MULTIPROCESSORS

The architectural design of supercomputers continues to be upgraded based on advances in technology and past experience. Design rules are provided for high performance, and we review these rules in case studies of well-known early supercomputers, high-end mainframes, and minisupercomputers. The trends toward scalable architectures in building MPP systems for supercomputing are also assessed, while recent developments will be discussed in Chapter 13.

8.2.1 Performance-Directed Design Rules

Supercomputers are targeted toward large-scale scientific and engineering problems. They should provide the highest performance constrained only by current technology. In addition, they must be programmable and accessible in a multiuser environment.

Supercomputer architecture design rules are presented below. These rules are driven by the desire to offer the highest available performance in a variety of respects, including processor, memory, and I/O performance, capacities, and bandwidths in all subsystems.

Architecture Design Goals Smith, Hsu, and Hsiung (1990) identified the following four major challenges in the development of future general-purpose supercomputers:

- Maintaining a good vector/scalar performance balance.
- Supporting scalability with an increasing number of processors.
- Increasing memory system capacity and performance.
- Providing high-performance I/O and an easy-access network.

Balanced Vector/Scalar Ratio In a supercomputer, separate hardware resources with different speeds are dedicated to concurrent vector and scalar operations. Scalar processing is indispensable for general-purpose architectures. Vector processing is needed for regularly structured parallelism in scientific and engineering computations. These two types of computations must be balanced.

The *vector balance point* is defined as the percentage of vector code in a program required to achieve equal utilization of vector and scalar hardware. In other words, we expect equal time spent in vector and scalar hardware so that no resources will be idle.

Example 8.3 Vector/scalar balance point in supercomputer design (Smith, Hsu, and Hsiung, 1990)

If a system is capable of 9 Mflops in vector mode and 1 Mflops in scalar mode, equal time will be spent in each mode if the code is 90% vector and 10% scalar, resulting in a vector balance point of 0.9.

It may not be optimal for a system to spend equal time in vector and scalar modes. However, the vector balance point should be maintained sufficiently high, matching the level of vectorization in user programs.

Vector performance can be enhanced with replicated functional unit pipelines in each processor. Another approach is to apply deeper pipelining on vector units with a double or triple clock rate with respect to scalar pipeline operations. Longer vectors are required to really achieve the target performance.

Vector/Scalar Performance In Figs. 8.7a and 8.7b, the single-processor vector performance and scalar performance are shown, based on running Livermore Fortran loops on Cray Research and Japanese supercomputers of the 1980s and early 1990s. The scalar performance of these supercomputers increases along the dashed lines in the figure.

One of the contributing factors to vector capability is the high clock rate, and other factors include use of a better compiler and the optimization support provided.

Table 8.2 compares the vector and scalar performances in seven supercomputers of that period. Note that these supercomputers have a 90% or higher vector balance point. The higher the vector/scalar ratio, the heavier the dependence on a high degree of vectorization in the object code.

Table 8.2 *Vector and Scalar Performance of Various Early Supercomputers*

Machine	*Cray 1S*	*Cray 2S*	*Cray X-MP*	*Cray Y-MP*	*Hitachi S820*	*NEC SX2*	*Fujitsu VP400*
Vector performance (Mflops)	85.0	151.5	143.3	201.6	737.3	424.2	207.1
Scalar performance (Mflops)	9.8	11.2	13.1	17.0	17.8	9.5	6.6
Vector balance point	0.90	0.93	0.92	0.92	0.98	0.98	0.97

Source: J.E. Smith et al., Future General-Purpose Supercomputing Conference, *IEEE Supercomputing Conference*, 1990.

The above approach is quite different from the design in comparable IBM vector machines which maintained a low vector/scalar ratio between 3 and 5. The idea was to make a good compromise between the demands of scalar and vector processing for general-purpose applications.

I/O and Networking Performance With the aggregate speed of supercomputers increasing at least three to five times each generation, problem size has been increasing accordingly, as have I/O bandwidth requirements. Figure 8.7c illustrates the aggregate I/O bandwidths supported by supercomputer systems of the period up to the early 1990s.

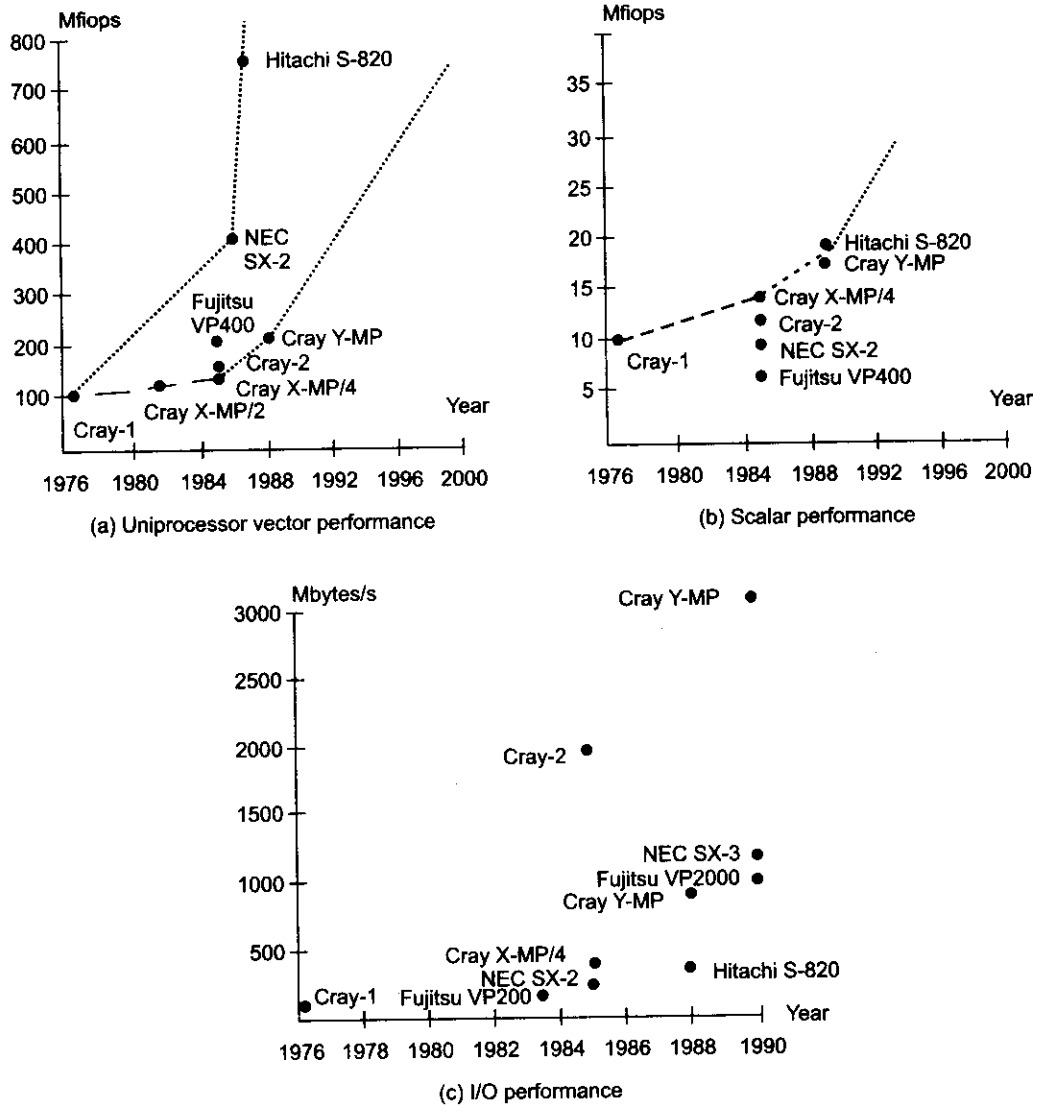


Fig. 8.7 Some reported supercomputer performance data (Source: Smith, Hsu, and Hsiung, IEEE Supercomputing Conference, 1990)

The I/O is defined as the transfer of data between the processor/memory and peripherals or a network. In the earlier generation of supercomputers, I/O bandwidths were not always well correlated with computational performance. I/O processor architectures were implemented by Cray Research with two different approaches.

The first approach is exemplified by the Cray Y-MP I/O subsystem, which used I/O processors that were flexible and could do complex processing. The second approach was used in the Cray 2, where a simple front-end processor controlled high-speed channels with most of the I/O management being done by the mainframe's operating system.

Today more than aggregate 100-Gbytes/s I/O transfer rate are needed in supercomputers connected to high-speed disk arrays and networks. Support for high-speed networking has become a major component of the I/O architecture in supercomputers.

Memory Demand The main memory sizes and extended memory sizes of supercomputers of 1980s and early 1990s are shown in Fig. 8.8. A large-scale memory system must provide a low latency for scalar processing, a high bandwidth for vector and parallel processing, and a large size for grand challenge problems and throughput.

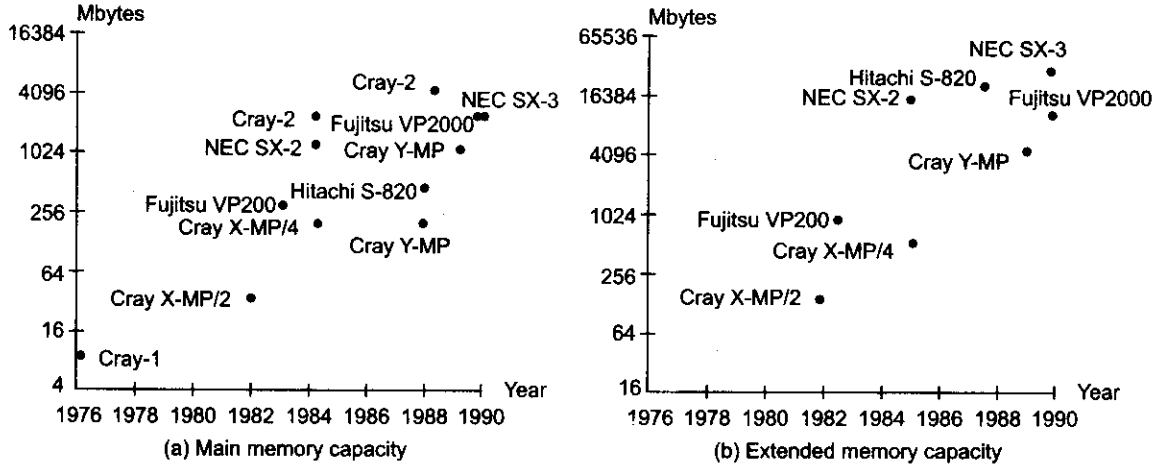


Fig. 8.8 Supercomputer memory capacities (Source: Smith, Hsu, and Hsiung, *IEEE Supercomputing Conference*, 1990)

To achieve the above goals, an effective memory hierarchy is necessary. A typical hierarchy may consist of data files or disks, extended memory in dynamic RAMs, a fast shared memory in static RAMs, and a cache/local memory using RAM on arrays.

Over the last two decades, with advances in VLSI technology, the processing power available on a chip has tended to double every two years or so. Memory sizes available on a chip have also grown rapidly; however, as we shall see in Chapter 13, the *memory speeds* achievable—i.e. read and write cycle times—have grown much less rapidly than processor performance. Therefore the *relative* speed mismatch between processors and memory, which has been a feature of computer systems from their earliest days, has widened much further over the last two decades. This has necessitated the development of more sophisticated memory latency hiding techniques, such as wider memory access paths and multi-level cache memories.

Supporting Scalability Multiprocessor supercomputers must be designed to support the triad of scalar, vector, and parallel processing. The dominant scalability problem involves support of shared memory with an increasing number of processors and memory ports. Increasing memory-access latency and interprocessor communication overhead impose additional constraints on scalability.

Scalable architectures include multistage interconnection networks in flat systems, hierarchical clustered systems, and multidimensional spanning buses, ring, mesh, or torus networks with a distributed shared memory. Table 8.3 summarizes the key features of three representative multivector supercomputers of 1990s.

8.2.2 Cray Y-MP, C-90, and MPP

We study below the architectures of the Cray Research Y-MP, C-90, and MPP. Besides architectural features, we examine the operating systems, languages/compilers, and target performance of these machines.

Table 8.3 *Architectural Characteristics of Three Supercomputers of the 1990s*

Machine Characteristics	*Cray Y-MP C90/16256*	*NEC SX-X Series*	*Fujitsu VP-2000 Series*
Number of processors	16 CPUs	4 arithmetic processors	1 for VP2600/10, 2 for VP2400/40
Machine cycle time	4.2 ns	2.9 ns	3.2 ns
Max. memory	256M words (2 Gbytes).	2 Gbytes, 1024-way interleaving.	1 or 3 Gbytes of SRAM.
Optional SSD memory	512M, 1024M, or 2048M words (16 Gbytes).	16 Gbytes with 2.75 Gbytes/s transfer rate.	32 Gbytes of extended memory.
Processor architecture: vector pipelines, functional and scalar units	Two vector pipes and two functional units per CPU, delivering 64 vector results per clock period.	Four sets of vector pipelines per processor, each set with two adder/shift and two multiply/logical pipelines. A separate scalar pipeline.	Two load/store pipes and 5 functional pipes per vector unit, 1 or 2 vector units, 2 scalar units could be attached to each vector unit.
Operating system	UNICOS derived from UNIX/V and BSD.	Super-UX based on UNIX System V and 4.3 BSD.	UXP/M and MSP/EX enhanced for vector processing.
Front-ends	IBM, CDC, DEC, Univac, Apollo, Honeywell.	Built-in control processor and 4 I/O processors.	IBM-compatible hosts.
Vectorizing languages / compilers	Fortran 77, C, CF77 5.0, Cray C release 3.0	Fortran 77/SX, Vectorizer/XS, Analyzer/SX.	Fortran 77 EX/VP, C/VP compiler with interactive vectorizer.
Peak performance and I/O bandwidth	16 Gflops, 13.6 Gbytes/s.	22 Gflops, 1 Gbyte/s per I/O processor.	5 Gflops, 2 Gbyte/s with 256 channels.

The Cray Y-MP 816 A schematic block diagram of the Y-MP 8 is shown in Fig. 8.9. The system could be configured to have one, two, four, or eight processors. The eight CPUs of the Y-MP shared the central memory, the I/O section, the interprocessor communication section, and the real-time clock.

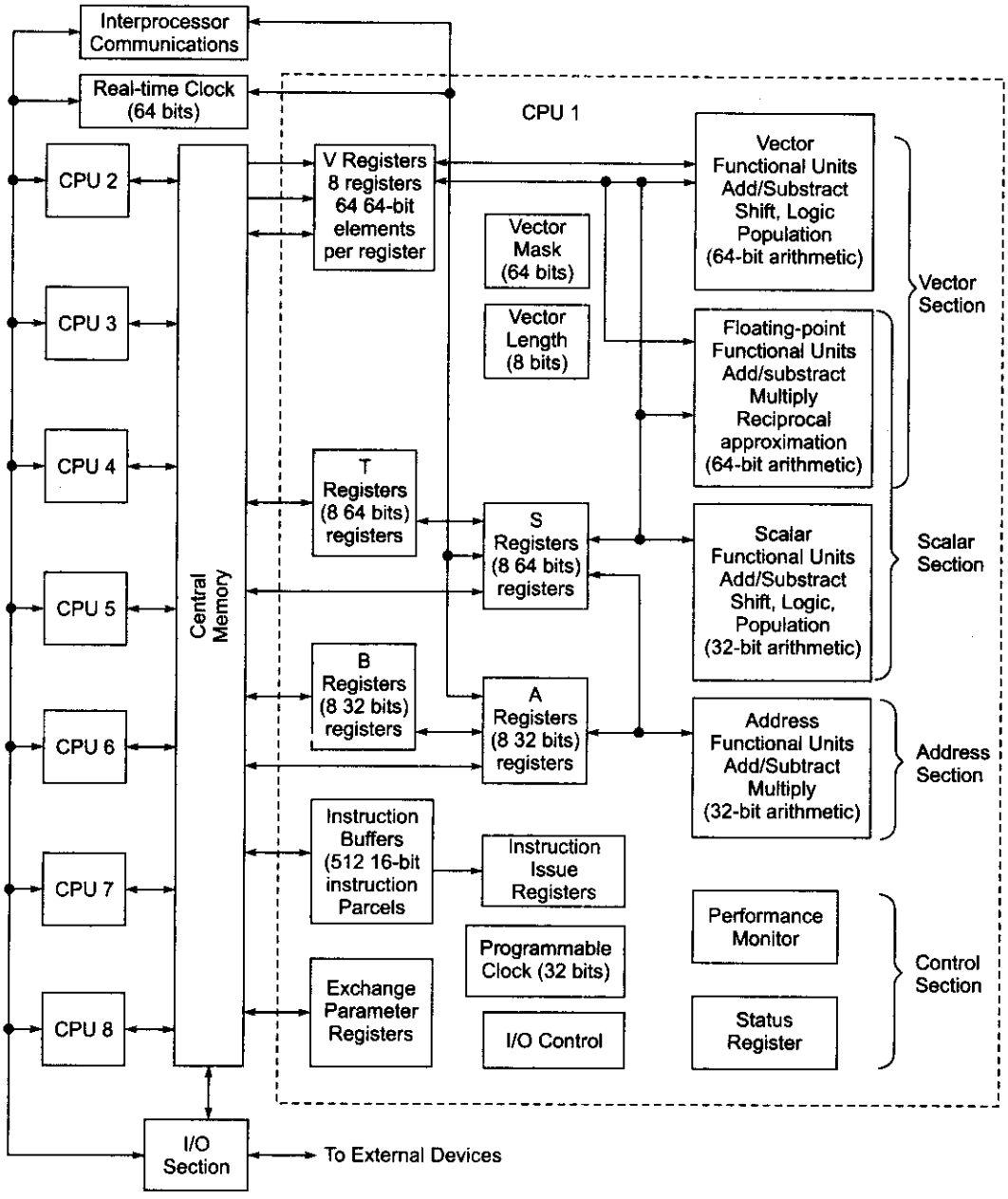


Fig. 8.9 Cray Y-MP 816 system organization (Courtesy of Cray Research, 1991)

The central memory was divided into 256 interleaved banks. Overlapping memory access was made possible through memory interleaving via four memory-access ports per CPU. A 6-ns clock period was used in the CPU design.

The central memory offered 16M-, 32M-, 64M-, and 128M-word options with a maximum size of 1 Gbyte. The SSD options were from 32M to 512M words or up to 4 Gbytes.

The four memory-access ports allowed each CPU to perform two scalar and vector *fetches*, one *store*, and one independent *I/O* simultaneously. These parallel memory accesses were also pipelined to make the *vector read* and *vector write* possible.

The system had built-in resolution hardware to minimize the delays caused by memory conflicts. To protect data, single-error correction/double-error detection (SECDED) logic was used in central memory and on the data channels to and from central memory.

The CPU computation section consisted of 14 functional units divided into vector, scalar, address, and control sections (Fig. 8.9). Both scalar and vector instructions could be executed in parallel. All arithmetic was register-to-register. Eight out of the 14 functional units could be used by vector instructions.

Large numbers of address, scalar, vector, intermediate, and temporary registers were used. Flexible chaining of functional pipelines was made possible through the use of registers and multiple memory-access and arithmetic/logic pipelines. Both 64-bit floating-point and 64-bit integer arithmetic were performed. Large instruction caches (buffers) were used to hold 512 16-bit instruction parcels at the same time.

The interprocessor communication section of the mainframe contained clusters of shared registers for fast synchronization purposes. Each cluster consisted of shared address, shared scalar, and semaphore registers. Note that vector data communication among the CPUs was done through the shared memory.

The real-time clock consisted of a 64-bit counter that advanced one count each clock period. Because the clock advanced synchronously with program execution, it could be used to time the execution to an exact clock count.

The I/O section supported three channel types with transfer rates of 6 Mbytes/s, 100 Mbytes/s, and 1 Gbyte/s. The IOS and SSD were high-speed data transfer devices designed to support the mainframe processing by eight caches.

Example 8.4 The multistage crossbar network in the Cray Y-MP 816

The interconnections between the 8 CPUs and 256 memory banks in the Cray Y-MP 816 were implemented with a multistage crossbar network, logically depicted in Fig. 8.10. The building blocks were 4×4 and 8×8 crossbar switches and 1×8 demultiplexers.

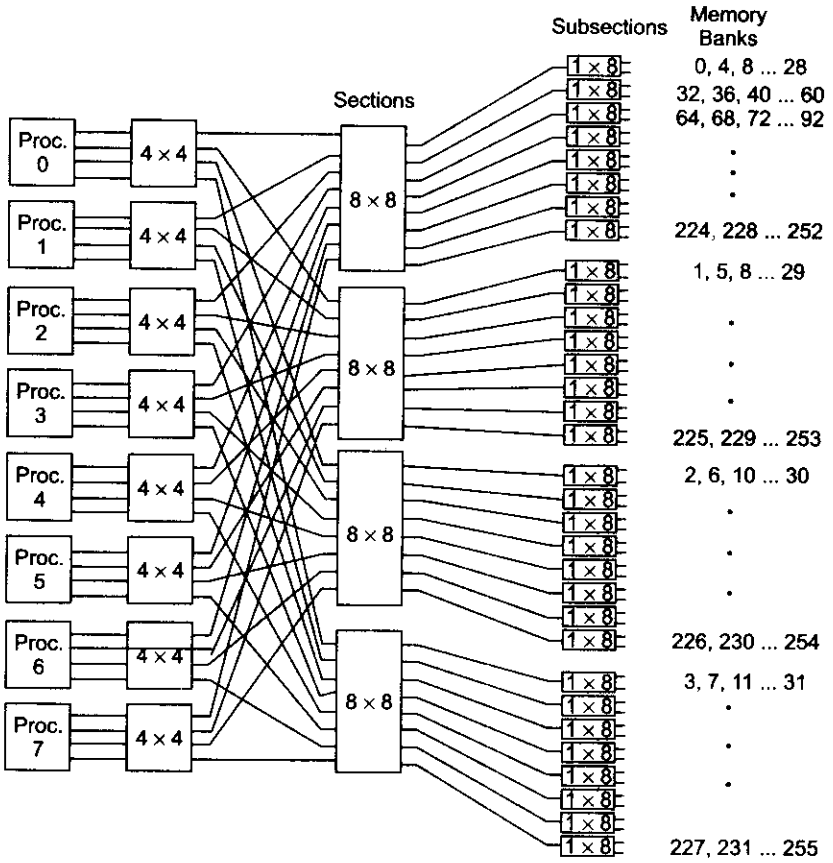


Fig. 8.10 Schematic logic diagram of the crossbar network between 8 processors and 256 memory banks in the Cray Y-MP 816

The network was controlled by a form of circuit switching where all conflicts were worked out early in the memory-access process and all requests from a given port returned to the port in order.

The use of a multistage network instead of a single-stage crossbar for interprocessor memory connections was aimed at enhancing scalability in the building of even larger systems with 64 or 1024 processors.

However, crossbar networks work only for small systems. To enhance scalability, emphasis should be given to data routing, heavier reliance on processor-based local memory (as in the Cray 2), or the use of clustered structures (as in the Cedar multiprocessor) to offset any increased latency when system size increases.

The C-90 and Clusters The C-90 was further enhanced in technology and scaled in size from the Y-MP Series. The architectural features of C-90/16256 are summarized in Table 8.3. The system was built with 16 CPUs, each of which was similar to that used in the Y-MP. The system used up to 256 megawords (2 Gbytes) of shared main memory among the 16 processors. Up to 16 Gbytes of SSD memory was available

as optional secondary main memory. In each cycle, two vector pipes and two functional units could operate in parallel, producing four vector results per clock. This implied a four-way parallelism within each processor. Thus 16 processors could deliver a maximum of 64 vector results per clock cycle.

The C-90 used the UNICOS operating system, which was extended from the UNIX system V and Berkeley BSD 4.3. The C-90 could be driven by a number of host machines. Vectorizing compilers were available for Fortran 77 and C on the system. The 64-way parallelism, coupled with a 4.2-ns clock cycle, lead to a peak performance of 16 Gflops. The system had a maximum I/O bandwidth of 13.6 Gbytes/s.

Multiple C-90's could be used in a clustered configuration in order to solve large-scale problems. As illustrated in Fig. 8.11, four C-90 clusters were connected to a group of SSDs via 1000 Mbytes/s channels. Each C-90 cluster was allowed to access only its own main memory. However, they shared the access of the SSDs. In other words, large data sets in the SSD could be shared by four clusters of C-90's. The clusters could also communicate with each other through a shared semaphore unit. Only synchronization and control information was passed via the semaphore unit. In this sense, the C-90 clusters were loosely coupled, but collectively they could provide a maximum of 256-way parallelism. For computations which were well partitioned and balanced among the clusters, a maximum peak performance of 64 Gflops was possible for a four-cluster configuration.

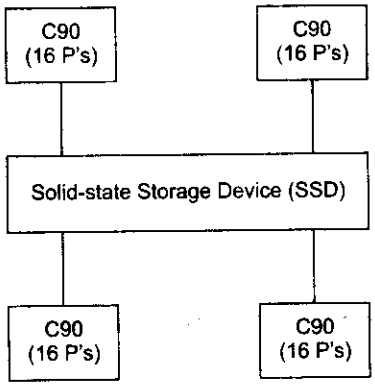


Fig. 8.11 Four Cray Y-MP C-90's connected to a common SSD forming a loosely coupled 64-way parallel system

The Cray/MPP System Massively parallel processing (MPP) systems have the potential for tackling highly parallel problems. Standard off-the-shelf microprocessors may have deficiencies when used as building blocks of an MPP system. What is needed is a balanced system that matches fast processor speed with fast I/O, fast memory access, and capable software. Cray Research announced its MPP development in October 1992. The development plan sheds some light on the trend towards MPP from the standpoint of a major supercomputer manufacturer.

Most of the early RISC microprocessors lacked the communication, memory, and synchronization features needed for efficient MPP systems. Cray Research planned to circumvent these shortcomings by surrounding the RISC chip with powerful communications hardware, besides exploiting Cray's expertise in supercomputer packaging and cooling. In this way, thousands of commodity RISC processors would be transformed into a supercomputer-class MPP system that could address terabytes of memory, minimize communication overhead, and provide flexible, lightweight synchronization in a UNIX environment.

Cray's first MPP system was code-named T3D because a three-dimensional, dense torus network was used to interconnect the machine resources. The heart of Cray's T3D was a scalable macroarchitecture that combined the DEC Alpha microprocessors through a low-latency interconnect network that had a bisection bandwidth an order of magnitude greater than that of existing MPP systems. The T3D system was designed to work jointly with the Cray Y-MP C-90 or the large-memory M-90 in a closely coupled fashion. Specific features of the MPP macroarchitecture are summarized below:

(1) The T3D was an MIMD machine that could be dynamically partitioned to emulate SIMD or multicomputer MIMD operations. The 3-D torus operated at a 150-MHz clock matching that of the Alpha chips. High-speed bidirectional switching nodes were built into the T3D network so that interprocessor communications could be handled without interrupting the PEs attached to the nodes. The T3D network was designed to be scalable from tens to thousands of PEs.

(2) The system used a globally addressable, physically distributed memory. Because the memory was logically shared, any PE could access the memory of any other processing element without explicit message passing and without involving the remote PE. As a result, the system could be scaled to address terabytes of memory. Latency hiding (to be studied in Chapter 9) was supported by data prefetching, fast synchronization, and parallel I/O. These were supported by dedicated hardware. For example, special remote-access hardware was provided to hide the long latency in memory accesses. Fast synchronization support included special primitives for data-parallel and message-passing programming paradigms.

(3) The Cray/MPP used a Mach-based microkernel operating system. Each PE had a microkernel that managed communications with other PEs and with the closely coupled Y-MP vector processors. Software portability was a major design goal in the Cray/MPP Series. Software-configurable redundant hardware was included so that processing could continue in the event of a PE failure.

(4) The Cray CFT77 compiler was modified with extended directives for MPP applications. Program debugging and performance tools were developed.

Cray/MPP Development Phases The original Cray/MPP program was planned to have three phases as illustrated in Fig. 8.12. The T3D/MPP was attached to the Cray Y-MP as a back-end accelerator engine. Besides hardware development, the biggest challenge in any MPP development is the software environment and availability. The Cray T3D programming model was based on an MIMD-oriented concept. Both the Connection Machine CM-5 (to be described in Section 8.5) and the Cray T3D emphasized this model, in order to broaden the application spectrum for their machines. More recent developments in Cray supercomputer systems are reviewed in Chapter 13.

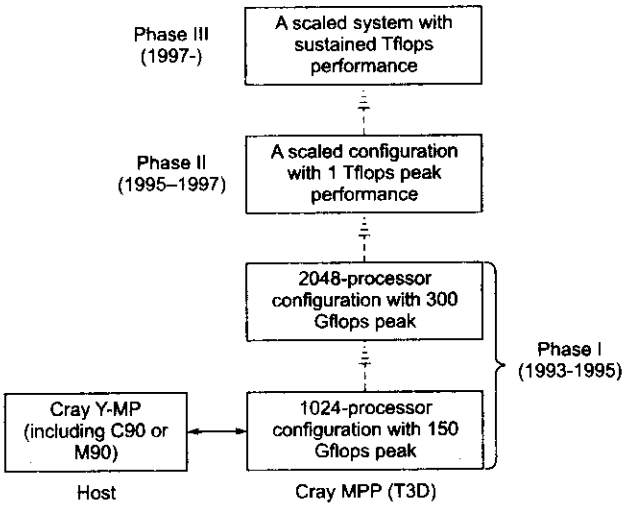


Fig. 8.12 The development phases of the original Cray/MPP system (Courtesy of Cray Research, 1992)

8.2.3 Fujitsu VP2000 and VPP500

Multivector multiprocessors from Fujitsu Ltd. are reviewed in this section as supercomputer design examples. The VP2000 Series offered one- or two-processor configurations. The VPP500 Series offered from 7 to 222 processing elements (PEs) in a single MPP system. The two systems could be used jointly in solving large-scale problems. We describe below the functional specifications and technology bases of the Fujitsu supercomputers.

The Fujitsu VP2000 Figure 8.13 shows the architecture of the VP-2600/10 uniprocessor system. The system could be expanded to have dual processors (the VP-2400/40). The system clock was 3.2 ns, the main memory unit was of 1 or 2 Gbytes, and the system storage unit provided up to 32 Gbytes of extended memory.

Each vector processing unit consisted of two load/store pipelines, three functional pipelines, and two mask pipelines. Two scalar units could be attached to each vector unit, making a maximum of four scalar units in the dual-processor configuration. The maximum vector performance ranged from 0.5 to 5 Gflops across 10 different models of the VP2000 Series.

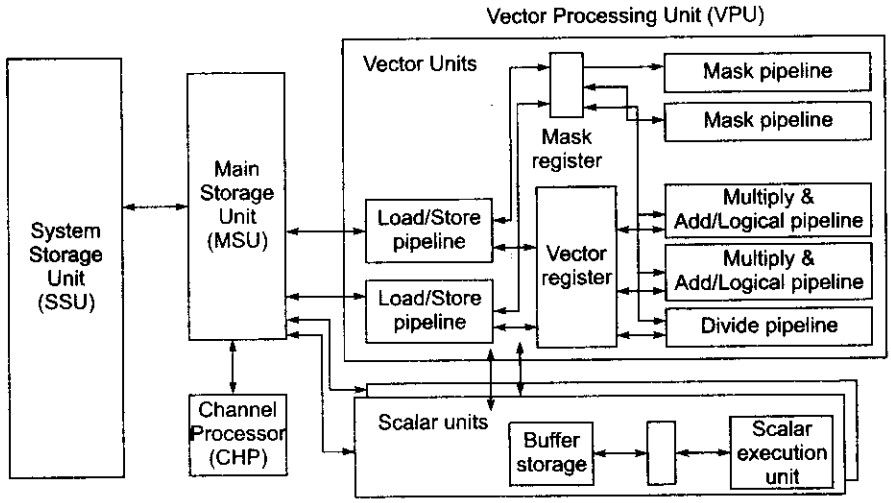
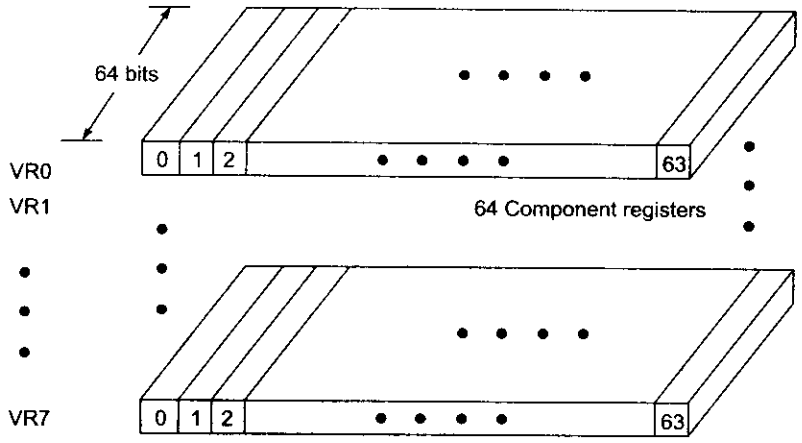


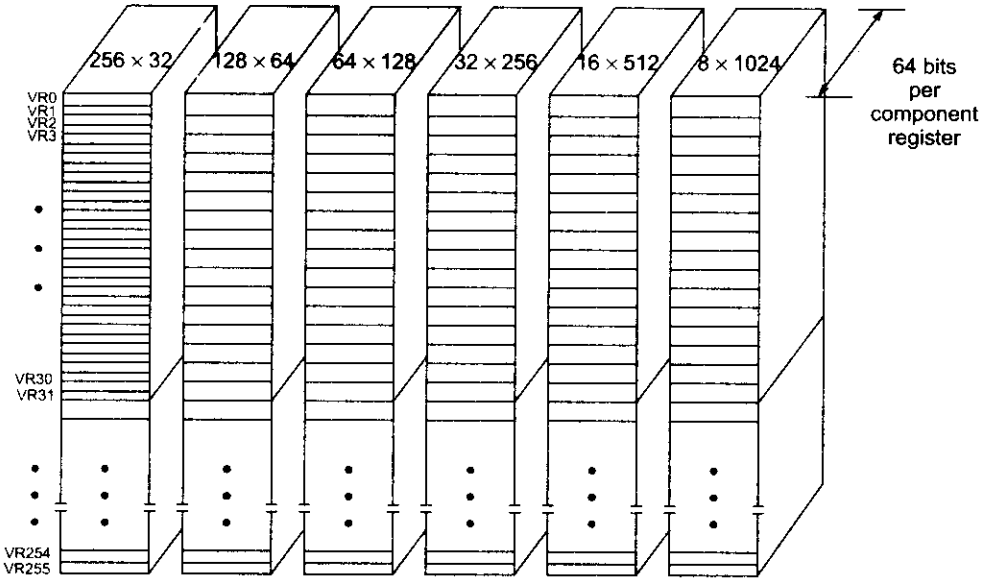
Fig. 8.13 The Fujitsu VP2000 Series supercomputer architecture (Courtesy of Fujitsu, 1991)

Example 8.5 Reconfigurable vector register file in the Fujitsu VP2000

Vector registers in Cray and Fujitsu machines are illustrated in Fig. 8.14. Cray machines used 8 vector registers, and each had a fixed length of 64 component registers. Each component register was 64 bits wide as shown in Fig. 8.14a.



(a) Eight vector registers (8 × 64 × 64 bits) on Cray machines



(b) Vector registers configurations in the Fujitsu VP2000

Fig. 8.14 Vector register file in Cray and Fujitsu supercomputers

A component counter was built within each Cray vector register to keep track of the number of vector elements fetched or processed. A segment of a 64-element subvector was held as a package in each vector register. Long vectors had to be divided into 64-element segments before they could be processed in a pipelined fashion.

In an early model of the Fujitsu VP2000, the vector registers were reconfigurable to have variable lengths. The purpose was to dynamically match the register length with the vector length being processed.

As illustrated in Fig. 8.14b, a total of 64 Kbytes in the register file could be configured into 8, 16, 32, 64, 128, and 256 vector registers with 1024, 512, 256, 128, 64, and 32 component registers, respectively. All component registers were 64 bits in length.

In the following Fortran Do loop operations, the three-dimensional vectors are indexed by I with constant values of J and K in the second and third dimensions.

```
Do  10 I = 0, 31
    ZZ0(I) = U(I,J,K) – U(I,J – 1,K)
    ZZ1(I) = V(I,J,K) – V(I,J – 1,K)
                ⋮
    ZZ84(I) = W(I,J,K) – W(I,J – 1,K)
10  Continue
```

The program can be vectorized to have 170 input vectors and 85 output vectors with a vector length of 32 elements (I = 0 to 31). Therefore, the optimal partition is to configure the register file as 256 vector registers with 32 components each.

Software support for parallel and vector processing in such supercomputers will be treated in Part IV. This includes multitasking, macrotasking, microtasking, autotasking, and interactive compiler optimization techniques for vectorization or parallelization.

The VPP 500 This was a latter supercomputer series from Fujitsu, called *vector parallel processor*. The architecture of the VPP500 was scalable from 7 to 222 PEs, offering a highly parallel MIMD multivector system. The peak performance was targeted for 335 Gflops. Figure 8.15 shows the architecture of the VPP500 used as a back-end machine attached to a VP2000 or a VPX 200 host.

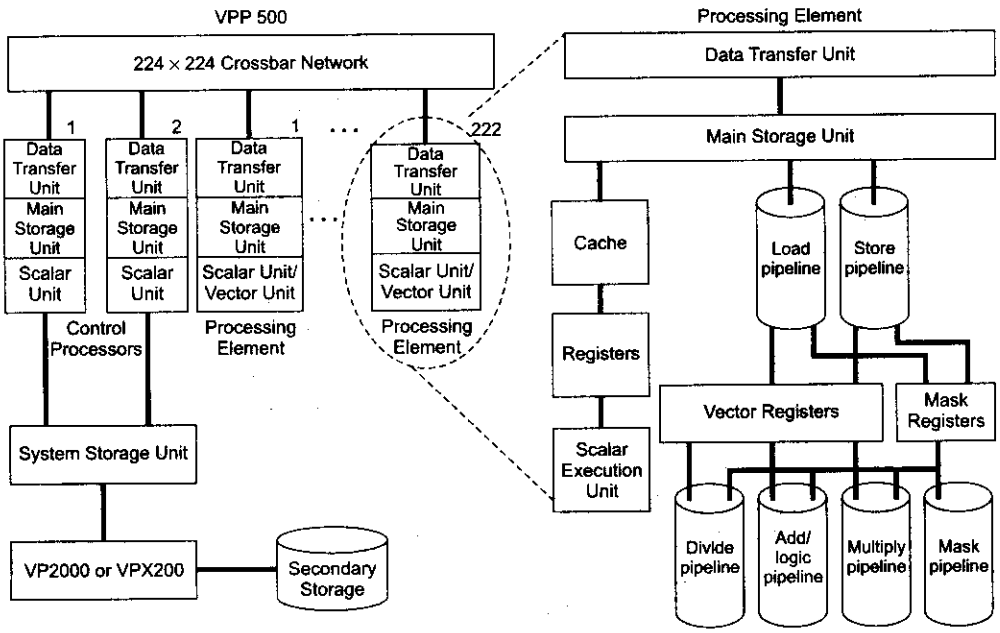


Fig. 8.15 The Fujitsu VPP500 architecture (Courtesy of Fujitsu, 1992)

Each PE had a peak processing speed of 1.6 Gflops, implemented with 256K-gate GaAs and BiCMOS LSI circuits. Up to two control processors coordinated the activities of the PEs through a crossbar network. The data transfer units in each PE handled inter-PE communications. Each PE had its own memory with up to 256 Mbytes of static RAM. The system applied the global shared virtual memory concept. In other words, the collection of local memories physically distributed over the PEs formed a single address space. The entire system could have up to 55 Gbytes of main memory collectively.

Each PE had a scalar unit and a vector unit operating in parallel. These functional pipelines were very similar to those built into the VP2000 (Fig. 8.13), but the pipeline functions were modified. We have seen the 224 × 224 crossbar design in Fig. 2.26b. This was by far the largest crossbar built into a commercial MPP system. The crossbar network is conflict-free, since only one crosspoint switch is on in each row or column of the crossbar switch array.

The VPP500 ran jointly with its host the UNIX System V Release 4-based UXP/VPP operating system with support for closely coupled MIMD operations. The optimization functions of the Fortran 77 compiler worked with the parallel scheduling function of the UNIX-based OS to exploit the maximum capability of the vector parallel architecture.

The data transfer unit in each PE provided 400 Mbytes/s unidirectional and 800 Mbytes/s bidirectional data exchange among PEs. The unit translated logical addresses to physical addresses to facilitate access to the virtual global memory. The unit was also equipped with special hardware for fast barrier synchronization. We will further review the software environment for the VPP500 in Chapter 11.

The system was scalable with an incremental control structure. A single control processor was sufficient to control up to 9 PEs. Two control processors were used to coordinate a VPP with 30 to 222 PEs. The system performance was scalable with the number of PEs spanning a peak performance range from 11 to 335 Gflops and a memory capacity of 1.8 to 55 Gbytes.

8.2.4 Mainframes and Minisupercomputers

In the early 1990s, several high-end mainframes, minisupercomputers, and supercomputing workstations were introduced. Besides summarizing these systems, we examine the architecture designs of the VAX 9000 and Stardent 3000 as case studies. The LINPACK results compiled by Dongarra (1992) are presented to compare a range of these computers for solving linear systems of equations.

High-End Mainframe Supercomputers This class of supercomputers have been called *near-supercomputers*. In the early 1990s, they offered a peak performance of several hundreds of Mflops to 2.4 Gflops as listed in Table 8.4. These machines were not designed entirely for number crunching. Their main applications were in business and transaction processing. The floating-point capability was only an add-on optional feature of these mainframe machines.

The number of CPUs ranged from one to six in a single system among the IBM ES/9000, VAX 9000, and Cyber 2000 listed in Table 8.4. The main memory was between 32 Mbytes and 1 Gbyte. Extended memory could be as large as 8 Gbytes in the ES/9000.

Vector hardware was an optional feature which could be used concurrently with the scalar units. Most vector units consisted of an add pipeline and a multiply pipeline. The clock rates were between 9 and 30 ns in these machines. The I/O subsystems were rather sophisticated due to the need to support large database processing applications in a network environment.

DEC VAX 9000 Even though the VAX 9000 did not provide Gflop performance, the design represented a typical mainframe approach to high-performance computing. The architecture is shown in Fig. 8.16a.

Multichip packing technology was used to build the VAX 9000. It offered 40 times the VAX/780 performance per processor. With a four-processor configuration, this implied 157 times the 11/780 performance. When used for transaction processing, 70 TPS was reported on a uniprocessor. The peak vector processing rate ranged from 125 to 500 Mflops.

The system control unit utilized a crossbar switch providing four simultaneous 500-Mbytes/s data transfers. Besides incorporating interconnect logic, the crossbar was designed to monitor the contents of cache memories, tracking the most up-to-date cache content to maintain coherence.

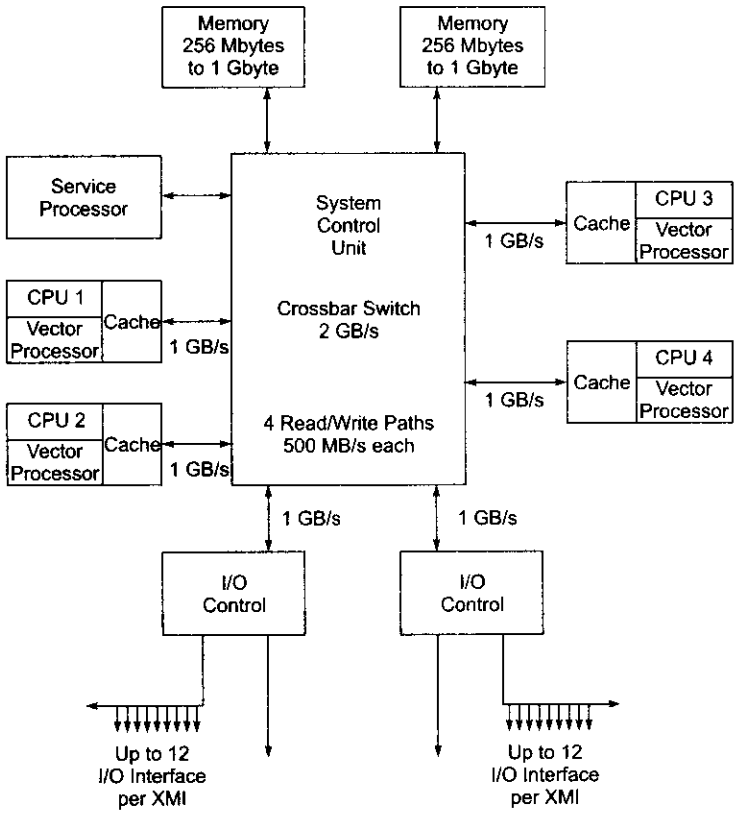
Up to 512 Mbytes of main memory were available using 1-Mbit DRAMs on 64-Mbyte arrays. Up to 2 Gbytes of extended memory were available using 4-Mbit DRAMs. Various I/O channels provided an aggregate data transfer rate of 320 Mbytes/s. The crossbar had eight ports to four processors, two memory modules, and two I/O controllers. Each port had a maximum transfer rate of 1 Gbyte/s, much higher than in bus-connected systems.

Table 8.4 *High-end Mainframe Supercomputers*

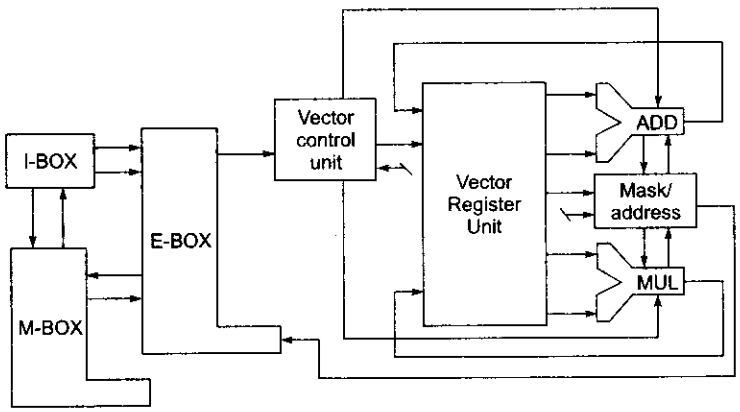
Machine Characteristics	*IBM ES/9000 -900 VF*	*DEC VAX 9000/440 VP*	*CDC Cyber 2000V*
Number of processors	6 processors each attached to a vector facility	4 processors with vector boxes	2 central processors with vector hardware
Machine cycle time	9 ns	16 ns	9 ns
Maximum memory	1 Gbyte	512 Mbytes	512 Mbytes
Extended memory	8 Gbytes	2 Gbytes	N/A
Processor architecture: vector, scalar, and other functional units	Vector facility (VF) attached to each processor, delivering 4 floating-point results per cycle.	Vector processor (VBOX) connected to a scalar CPU. Two vector pipelines per VBOX. Four functional units in scalar CPU.	FPU for add and multiply, scalar unit with divide and multiply, integer unit and business data handler per processor.
I/O subsystem	256 ESCON fiber optic channels.	4 XMI I/O buses and 14 VAXBI I/O buses.	18 I/O processors with optional 18 additional I/O processors.
Operating system	MVS/ESA, VM/ESA, VSE/ESA	VMS or ULTRIX	NOS/VE
Vectorizing languages/ compilers	Fortran V2 with interactive vectorization.	VAX Fortran compiler supporting concurrent scalar and vector processing.	Cyber 2000 Fortran V2.
Peak performance and remarks	2.4 Gflops	500 Mflops peak.	210 Mflops per processor.

Each vector processor (VBOX) was equipped with an add and a multiply pipeline using vector registers and a mask/address generator as shown in Fig. 8.16b. Vector instructions were fetched through the memory

unit (MBOX), decoded in the IBOX, and issued to the VBOX by the EBOX. Scalar operations were directly executed in the EBOX.



(a) The VAX 9000 multiprocessor system



(b) The vector processor (VBOX)

Fig. 8.16 The DEC VAX 9000 system architecture and vector processor design (Courtesy of Digital Equipment Corporation, 1991)

The vector register file consisted of $16 \times 64 \times 64$ bits, divided into sixteen 64-element vector registers. No instruction took more than five cycles. The vector processor generated two 64-bit results per cycle, and the vector pipelines could be chained for dot-product operations.

The VAX 9000 could run with either VMS or ULTRIX operating system. The service processor in Fig. 8.16a used four MicroVAX processors devoted to system, disk/tape, and user interface control and to monitoring 20,000 scan points throughout the system for reliable operation and fault diagnosis.

Minisupercomputers These were a class of low-cost supercomputer systems with a performance of about 5 to 15% and a cost of 3 to 10% of that of a full-scale supercomputer. Representative systems of the early 1990s include the Convex C series, Alliant FX series, Encore Multimax series, and Sequent Symmetry series.

Some of these minisupercomputers have been introduced in Chapters 1 and 7. Most of them had an open architecture using standard off-the-shelf processors and UNIX systems.

Both scalar and vector processing was supported in these multiprocessor systems with shared memory and peripherals. Most of these systems were built with a graphics subsystem for visualization and performance-tuning purposes.

Supercomputing Workstations In the early 1990s, high-performance workstations were being produced by Sun Microsystems, IBM, DEC, HP, Silicon Graphics, and Stardent using the state-of-the-art superscalar RISC processors introduced in Chapters 4 and 6. Most of these workstations had a uniprocessor configuration with built-in graphics support but no vector hardware.

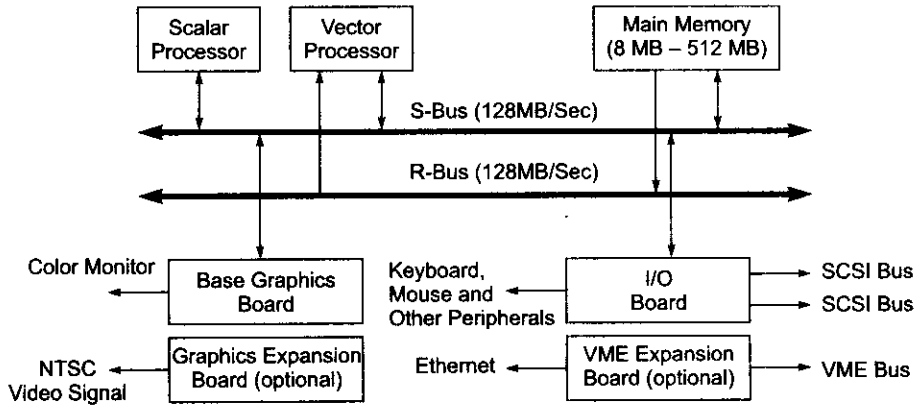
Silicon Graphics produced the 4-D Series using four R3000 CPUs in a single workstation without vector hardware. Stardent Computer Systems produced a departmental supercomputer, called the Stardent 3000, with custom-designed vector hardware.

The Stardent 3000 The Stardent 3000 was a multiprocessor workstation that evolved from the TITAN architecture developed by Ardent Computer Corporation. The architecture and graphics subsystem of the Stardent 3000 are depicted in Fig. 8.17. Two buses were used for communication between the four CPUs, memory, I/O, and graphics subsystems (Fig. 8.17a).

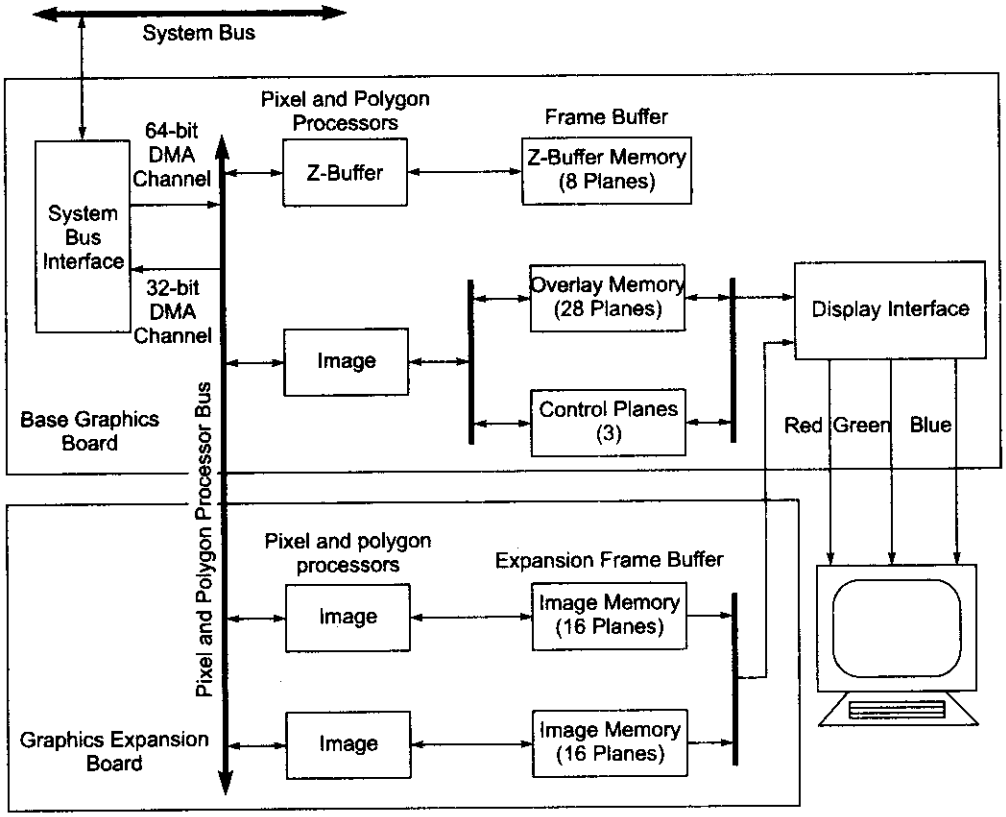
The system featured R3000/R3010 processors/floating-point units. The vector processors were custom-designed. A 32-MHz clock was used. There were 128 Kbytes of cache; one half was used for instructions and the other half for data.

The buses carried 32-bit addresses and 64-bit data and operated at 16 MHz. They were rated at 128 Mbytes/s each. The R-bus was dedicated to data transfers from memory to the vector processor, and the S-bus handled all other transfers. The system could support a maximum of 512 Mbytes of memory.

A full graphics subsystem is shown in Fig. 8.17b. It consisted of two boards that were tightly coupled to both the CPUs and memory. These boards incorporated rasterizers (pixel and polygon processors), frame buffers, Z-buffers, and additional overlay and control planes.



(a) The Stardent 3000 system architecture



(b) The graphics subsystem architecture

Fig. 8.17 The Stardent 3000 visualization departmental supercomputer (Courtesy of Stardent Computer, 1990)

The Stardent system was designed for numerically intensive computing with two- and three-dimensional rendering graphics. One to two I/O processors were connected to SCSI or VME buses and other peripherals or Ethernet connections. The peak performance was estimated at 32 to 128 MIPS, 16 to 64 scalar Mflops, and 32 to 128 vector Mflops. Scoreboard, crossbar switch, and arithmetic pipelines were implemented in each vector processor.

Gordon Bell, chief architect of the VAX Series and of the TITAN/Stardent architecture, identified 11 rules of minisupercomputer design in 1989. These rules require performance-directed design, balanced scalar/vector operations, avoiding holes in the performance space, achieving peaks in performance even on a single program, providing a decade of addressing space, making a computer easy to use, building on others' work, always looking ahead to the next generation, and expecting the unexpected with slack resources.

The LINPACK Results This is a general-purpose Fortran library of mathematical software for solving dense linear systems of equations of order 100 or higher. LINPACK is very sensitive to vector operations and the degree of vectorization by the compiler. It has been used to predict computer performance in scientific and engineering areas.

Many published Mflops and Gflops results are based on running the LINPACK code with prespecified compilers. LINPACK programs can be characterized as having a high percentage of floating-point arithmetic operations.

In solving a linear system of n equations, the total number of arithmetic operations involved is estimated as $2n^3/3 + 2n^2$, where $n = 1000$ in the LINPACK experiments.

Over many years, Dongarra compared the performance of various computer systems in solving dense systems of linear equations. His performance experiments involved about 100 computers.

The timing information presented in this report reflects the floating-point, parallel, and vector processing capabilities of the machines tested. Since the original reports are quite long, only brief excerpts are quoted in Table 8.5.

The second column reports LINPACK performance results based on a matrix of order $n = 100$ in a Fortran environment. The third column shows the results of solving a system of equations of order $n = 1000$ with no restriction on the method or its implementation. The last column lists the theoretical peak performance of the machines.

The LINPACK results reported in the second column of Table 8.5 were for a small problem size of 100 unknowns. No changes were made in the LINPACK software to exploit vector capabilities on multiple processors in the machines being evaluated. The compilers of some machines might generate optimized code that itself accessed special hardware features.

The third column corresponds to a much larger problem size of 1000 unknowns. All possible optimization means, including user optimizations of the software, were allowed to achieve as high an execution rate as possible, called the *best-effort Mflops*.

The theoretical peak can easily be calculated by counting the maximum number of floating-point additions and multiplications that can be completed during a period of time, usually the cycle time of the machine.

Table 8.5 *Performance in Solving a System of Linear Equations*

Computer Model	*LINPACK Benchmark n = 100 OS/Compiler, Mflops*	*Best-effort (Mflops) n = 1000*	*Theoretic Peak (Mflops)*
Cray Y-MP C90 (16 proc., 4.2 ns)	CF77 5.0 –Zp –Wd-e68 479	9715	16000
NEC SX-3/14 (1 proc., 2.9 ns)	f77SX020 R1.l3 –pi* 314	4511	5500
Fujitsu VP2400/10 (4 ns)	Fortran77 EX/VP V11 L10 170	1688	2000
Convex C3840 (4 proc., 16.7 ns)	fc7.0 –tm c38 –O3 –ep –ds –is 75	425	480
IBM ES/9000-520VF (1 proc., 9 ns)	VAST-2/VS Fortran V2R4 60	338	444
FPS 510S MCP707 (7 proc., 25 ns)	Pgf77 –O4 –Minline 33	184	280
Alliant FX/2800-200 (14 processors)	fortran 1.1.27 –O –inline 31	325	560
DEC VAX9000/410VP (1 proc., 16 ns)	HPO V1.3–163V, DXML 22	89	125
CDC Cyber 205 (4 pipes)	FTN 17	195	400
Stardent 3040	3.0 -inline –nmax = 300 12	77	128
SUN SPARCstation 2	177 1.4 -03 –cg89 -dalign 4	N/A	N/A
IBM PC/AT with 80287	Microsoft 3.2 0.0091	N/A N/A	N/A N/A

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

Example 8.6 Peak performance calculation for the Cray Y-MP/8

The Cray Y-MP/8 had a cycle time of 6 ns. During a cycle, the results of both an addition and a multiplication could be completed. Furthermore, there were eight processors operating simultaneously without interference in the best case. Thus, we calculate the peak performance of the Cray Y-MP/8 as follows:

$$\frac{2 \text{ operations}}{1 \text{ cycle}} \times \frac{1 \text{ cycle}}{6 \times 10^{-9} \text{ s}} \times 8 \text{ processors} = 2667 \text{ Mflops} = 2.6 \text{ Gflops} \tag{8.12}$$

The peak performance is often cited by manufacturers. It provides an upper bound on the real performance. Comparing the results in the second and third columns with the peak values, only 2.9 to 86.3% of the peak was achieved in these runs. This implies that the peak performance cannot represent sustained real performance in most cases. Often, only about 10% of the peak performance is achievable.

8.3 COMPOUND VECTOR PROCESSING

In this section, we study compound vector operations. Multipipeline chaining and networking techniques are described and design examples given. A graph transformation approach is presented for setting up pipeline networks to implement compound vector functions, which are either specified by the programmer or detected by an intelligent compiler.

8.3.1 Compound Vector Operations

A *compound vector function* (CVF) is defined as a composite function of vector operations converted from a looping structure of linked scalar operations. The following example clarifies the concept.

Example 8.7 A compound vector function called the SAXPY code

Consider the following Fortran type loop of a sequence of five scalar operations to be executed N times:

```
Do 10 I = 1, N
    Load        R1, X(I)
    Load        R2, Y(I)
    Multiply    R1, S
    Add         R2, R1
    Store       Y(I), R2
10  Continue
```
(8.13)

where X(I) and Y(I), I =1, 2, ..., N, are two source vectors originally residing in the memory. After the computation, the resulting vector is stored back to the memory. S is an immediate constant supplied to the multiply instruction.

After vectorization, the above scalar SAXPY code is converted to a sequence of five vector instructions:

$$\begin{aligned} M(x : x + N - 1) &\rightarrow V1 && \textit{Vector load} \\ M(y : y + N - 1) &\rightarrow V2 && \textit{Vector load} \\ S \times V1 &\rightarrow V1 && \textit{Vector multiply} \end{aligned} \tag{8.14}$$

$$\begin{aligned} V2 + V1 &\rightarrow V2 && \textit{Vector add} \\ V2 &\rightarrow M(y : y + N - 1) && \textit{Vector store} \end{aligned}$$

The same vector notation used in Eq. 4.1 is applied here, where x and y are the starting memory addresses of the X and Y vectors, respectively; V1 and V2 are two N-element vector registers in the vector processor.

The vector code in Eq. 8.14 can be expressed as a CVF as follows, using Fortran 90 notation:

$$Y(1 : N) = S \times X(1 : N) + Y(1 : N) \tag{8.15}$$

For simplicity, we write the above expression for a CVF as follows:

$$Y(I) = S \times X(I) + Y(I) \tag{8.16}$$

where the index I implies that all vector operations involve N elements.

Compound Vector Functions Table 8.6 lists a number of example CVFs involving one-dimensional vectors indexed by *I*. The same concept can be generalized to multidimensional vectors with multiple indexes. For simplicity, we discuss only CVFs defined over one-dimensional vectors. Typical operations appearing in these CVFs include *load*, *store*, *multiply*, *divide*, *logical*, and *shifting* vector operations. We use "slash" to represent the *divide* operations. All vector operations are defined on a component-wise basis unless otherwise specified.

The purpose of studying CVFs is to explore opportunities for concurrent processing of linked vector operations. The numbers of available vector registers and functional pipelines impose some limitations on how many CVFs can be executed simultaneously.

Table 8.6 *Representative Compound Vector Functions*

One-dimensional compound vector functions	*Maximum chaining degree*
V1 (I) = V2(I) + V3(I) × V4(I)	2
V1(I) = B(I) + C(I)	3
A(I) = V1(I) × S + B(I)	4
A(I) = V1(I) + B(I) + C(I)	5
A(I) = B(I) + S × C(I)	5
A(I) = B(I) + C(I) + D(I)	6
A(I) = Q × V1(I) (R × B(I) + C(I))	7
A(I) = B(I) × C(I) + D(I) × V1(I)	7
A(I) = V1(I) + (1 / A(I) + 1 / B(I) + Log(V2(I))	8
A(I) = $\sqrt{V2(I)}$ + Sin(B(I) + C(I)) + V3(I)	8
A(I) = B(I) × C(I) + D(I) × E(I) × S	9
A(I) = (A(I) + B(I) × C(I) + D(I)) × (I)	10

Note: Vi(I) are vector registers in the processor. A(I), B(I), C(I), D(I), and E(I) are vectors in memory. Scalars indicated as Q, R, and S are available from scalar registers in the processor. The chaining degrees include both memory-access and functional pipeline operations.

8.3.2 Vector Loops and Chaining

Vector pipelining and chaining are an integral part of all vector processors. Concurrent processing of several vector arithmetic, logic, shift, and memory-access operations require the chaining of multiple pipelines in a linear cascade. The idea of chaining is an extension of the technique of internal data forwarding practiced in a scalar processor (Fig. 5.15), and also leads to Stream Processing (see Chapter 13).

Chaining affects the speed of vector processors. Each of the CVFs listed in Table 8.6 is potentially a candidate for chaining. However, the implementation may hinge on the particular architecture of a vector machine. Principal concepts and implementations of vector looping, chaining, and recursion are described below.

Vector Loops or Strip-mining When a vector has a length greater than that of the vector registers, segmentation of the long vector into fixed-length *segments* is necessary. This technique has been called *strip-mining*. One vector segment (one surface of the mine field) is processed at a time. In the case of Cray computers, the vector segment length is 64 elements.

Until all the vector elements in each segment are processed, the vector register cannot be assigned to another vector operation. Strip-mining is restricted by the number of available vector registers and so is vector chaining. In the Fujitsu VP Series, the vector registers can be reconfigured to match the vector length. This allows strip-mining to be done more dynamically with a different "depth" in different applications.

The program construct for processing long vectors is called a *vector loop*. Vector segmentation is done by machine hardware under software control and should be transparent to the programmer. The loop count is determined at compile time or at run time, depending on the index value. Inside a loop, all vector operands have equal length, equal to that of the vector register.

Functional Units Independence In order for vector operations to be linked, they must follow a linear data flow pattern, and all functional pipeline units employed must be independent of each other. The same unit cannot be assigned to execute more than one instruction in the same chain.

Furthermore, vector registers must be lined up as interfaces between functional pipelines. The successive output results of a pipeline unit are fed into a vector register, one element per cycle. This vector register is then used as an input register for the next pipeline unit in the chain.

With the requirement of continuous data flow in the successive pipelines, the interface registers must be able to pass one vector element per cycle between adjacent pipelines. There may be transition time delays between loading the successive vector segments into the interface registers.

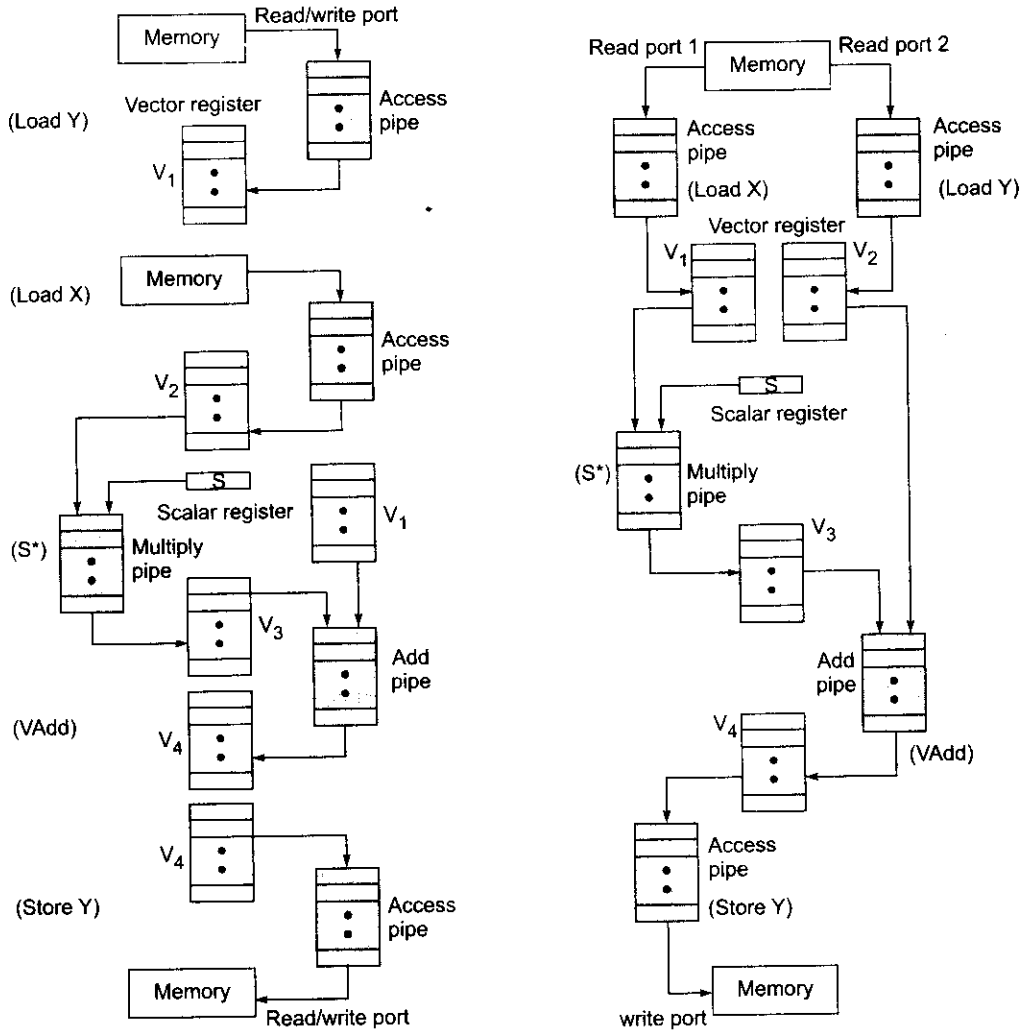
To avoid conflicts among different vector operations, the vector registers and functional pipelines must be reserved before a vector chain can be established. The vector chaining and the timing relationship are illustrated in Figs. 8.18 and 8.19 for executing the vectorized SAXPY code specified in Eq. 8.14.

Example 8.8 Pipeline chaining on Cray Supercomputers and on the Cray X-MP (Courtesy of Cray Research, Inc., 1985)

The Cray 1 had one memory-access pipe for either load or store but not for both at the same time. The Cray X-MP had three memory-access pipes, two for *vector load* and one for *vector store*. These three access pipes could be used simultaneously.

To implement the SAXPY code on the Cray 1, the five vector operations are divided into three chains: The first chain has only one vector operation, *load Y*. The second chain links the *load X* to scalar-vector *multiply* (S ×) operations and then to the *vector add* operation. The last chain is for *store Y* as illustrated in Fig. 8.18a.

The same set of vector operations was implemented on the Cray X-MP in a single chain, as shown in Fig. 8.18b, because three memory-access pipes are used simultaneously. The chain links five vector operations in a single connected cascade.

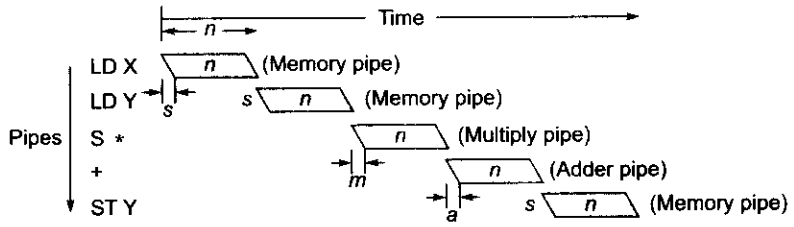


(a) Limited chaining using only one memory-access pipe in the Cray 1

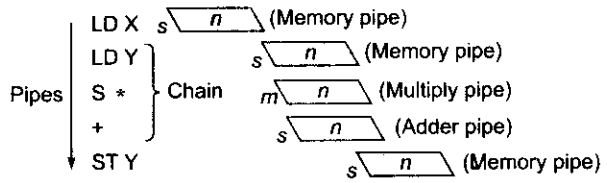
(b) Complete chaining using three memory-access pipes in the Cray X-MP

Fig. 8.18 Multipipeline chaining on Cray 1 and Cray X-MP for executing the SAXPY code: Y(1:N) = S × X(1:N) + Y(1:N) (Courtesy of Cray Research, 1985)

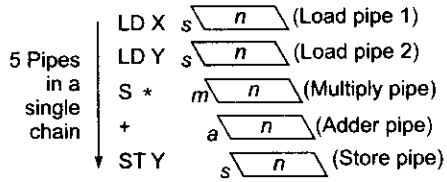
To compare the time required for chaining these pipelines, Fig. 8.19a shows that roughly $5n$ cycles are needed to perform the vector operations sequentially without any overlapping or any chaining. The Cray 1 requires about $3n$ cycles to execute, corresponding to about n cycles for each vector chain. The Cray X-MP requires about n cycles to execute.



(a) Sequential execution without chaining



(b) Chaining of arithmetic pipes with only one memory pipe



Captions:
s = Memory access latency
m, a = Multiply/add latencies
n = Time to process n elements, one per cycle

(c) Chaining in two load pipes, two arithmetic pipes, and one store pipe

Fig. 8.19 Timing for chaining the SAXPY code Y(1 : N) = S × X(1 : N) + Y(1 : N) under different memory-access capabilities (Courtesy of Cray Research, 1985)

In Fig. 8.19, the pipeline flow-through latencies (startup delays) are denoted as s, m, and a for the memory-access pipe, the multiply pipe, and the add pipe, respectively. These latencies equal the lengths of individual pipelines. The exact cycle counts can be slightly greater than the counts of $5n$, $3n$, and n due to these extra delays.

The above example clearly demonstrates the advantages of vector chaining. A meaningful chain must link two or more pipelines. As far as the amount of time is concerned, the longer the chaining, the better the

performance. The *degree of chaining* is indicated by the number of distinct pipeline units that can be linked together.

Vector chaining effectively increases the overall pipeline length by adding the pipeline stages of all functional units in the chain to form a single long pipeline. The potential speedup of this long pipeline is certainly greater according to Eq. 5.5.

Chaining Limitations The number of vector operations in a CVF must be small enough to make chaining possible. Vector chaining is limited by the small number of functional pipelines available in a vector processor. Furthermore, the limited number of vector registers also imposes an additional limit on chaining.

For example, the Cray Y-MP had only eight vector registers. Suppose all memory pipes are used in a vector chain. These require that three vector registers (two for *vector read* and one for *vector store*) be reserved at the beginning and end of the chaining operations. The remaining five vector registers are used for arithmetic, logic, and shift operations.

The number of interface registers required between two adjacent pipeline units is at least one and sometimes two for two source vectors. Thus, the number of non-memory-access vector operations implementable with the remaining five vector registers cannot be greater than five. In practice, this number is between two and three.

The actual degree of chaining depends on how many of the vector operations involved are binary or unary and how many use scalar or vector registers. If they are all binary operations, each requiring two source vector registers, then only two or three vector operations can be sandwiched between the memory-access operations. Thus a single chain on the Cray/Y-MP could link at most five or six vector operations including the memory-access operations.

Vector Recurrence These are a special class of vector loops in which the outputs of a functional pipeline may feed back into one of its own source vector registers. In other words, a vector register is used for holding the source operands and the result elements simultaneously.

This has been done on Cray machines using a *component counter* associated with each vector register. In each pipeline cycle, the vector register is used like a shift register at the component level. When a component operand is "shifted" out of the vector register and enters the functional pipeline, a result component can enter the vacated component register during the same cycle. The component counter must keep track of the shifting operations until all 64 components of the result are loaded into the vector register.

Recursive vector summation is often needed in scientific and statistical computations. For example, the dot product of two vectors, $A \cdot B = \sum_{i=1}^{n} a_i \times b_i$, can be implemented using recursion. Another example is polynomial evaluation over vector operands.

Summary Our discussion of vector and pipeline chaining is based on a load-store architecture using vector registers in all vector instructions. The number of functional units increases steadily in supercomputers; both the Cray C-90 and the NEC SX-X offered 16-way parallelism within each processor.

The degree of chaining can certainly increase if the vector register file becomes larger and scoreboarding techniques are applied to ensure functional unit independence and to resolve data dependence or resource dependence problems. The use of multiport memory is crucial to enabling large vector chains.

Vector looping, chaining, and recursion represent the state of the art in extending pipelining for vector processing. Furthermore, one can use *masking*, *scatter*, and *gather* instructions to manipulate sparse vectors or sparse matrices containing a large number of dummy zero entries. A vector processor cannot be considered versatile unless it is designed to handle both dense and sparse vectors effectively.

8.3.3 Multipipeline Networking

The idea of linking vector operations can be generalized to a multipipeline networking concept. Instead of linking vector operations into a linear chain, one can build a *pipenet* by introducing multiple functional pipelines with inserted delays to achieve *systolic computation* of CVFs.

In 1978, Kung and Leiserson introduced systolic arrays for special-purpose computing. Their idea was to map a specific algorithm into a fixed architecture. A *systolic array* is formed with a network of functional units which are locally connected and operates synchronously with multidimensional pipelining. We explain below how a pipeline net can be extended from the systolic array concept to build a dynamic vector processor for efficient execution of various CVFs.

Pipeline Net (Pipenet) Systolic arrays are built with fixed connectivity among the processing cells. This restriction is removed in a pipeline net. A pipenet has programmable connectivity as illustrated in Fig. 8.20. It is constructed from interconnecting multiple *functional pipelines* (FPs) through two *buffered crossbar networks* (BCNs) which are themselves pipelined.

A two-level pipeline architecture is seen in a pipeline net. The lower level corresponds to pipelining within each functional unit. The higher level is the pipelining of FPs through the BCNs. A generic model of a pipeline net is shown in Fig. 8.20d. The register file includes scalar and vector registers, as found in a typical vector processor.

The set of functional pipelines should be able to handle important vector arithmetic, logic, shifting, and masking operations. Each FP_i is pipelined with k_i stages. The output terminals of each BCN are buffered with programmable delays. BCN1 is used to establish the dynamic connections between the register file and the FPs. BCN2 sets up the dynamic connections among the FPs.

For simplicity, we call a pipeline network a *pipenet*. Conventional pipelines or pipeline chains are special cases of pipenets. Note that a pipenet is programmable with dynamic connectivity. This represents the fundamental difference between a static systolic array and a dynamic pipenet. In a way, one can visualize pipenets as programmable systolic arrays. The programmability sets up the dynamic connections, as well as the number of delays along some connection paths.

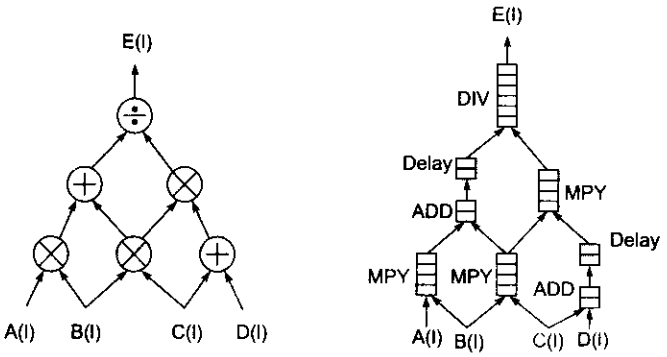
Setup of the Pipenet Figures 8.20a through 8.20d show how to convert from a program graph to a pipenet. Whenever a CVF is to be evaluated, the crossbar networks are programmed to set up a connectivity pattern among the FPs that matches the data flow pattern in the CVF.

The *program graph* represents the data flow pattern in a given CVF. Nodes on the graph correspond to vector operators, and edges show the data dependence, with delays properly labeled, among the operators.

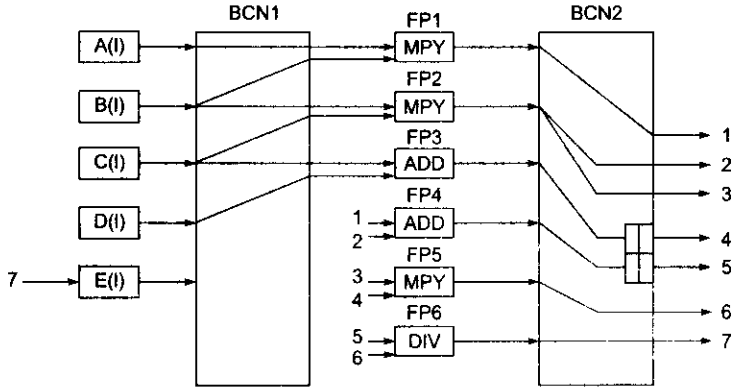
The program graph in Fig. 8.20a corresponds to the following CVF:

$$E(I) = [A(I) \times B(I) + B(I) \times C(I)] / [B(I) \times C(I) \times [C(I) + D(I)]] \tag{8.17}$$

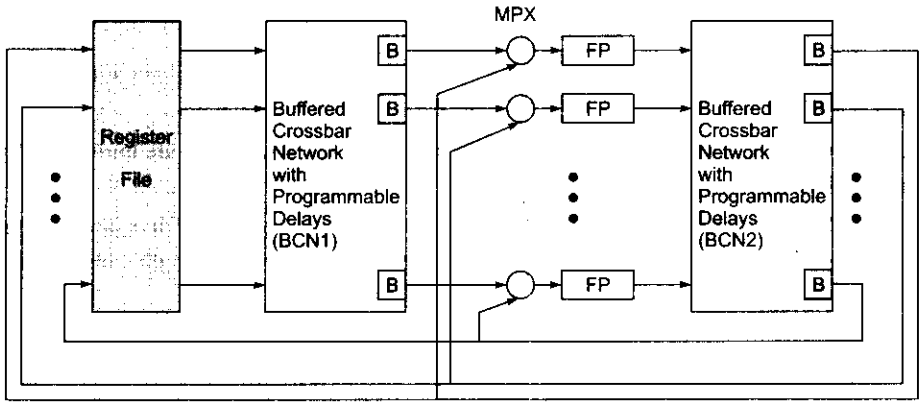
for $I = 1, 2, \ldots, n$. This CVF has four input vectors A(I), B(I), C(I), and D(I) and one output vector E(I) which demand five memory-access operations. In addition, there are seven vector arithmetic operations involved.



(a) A program graph (b) The pipenet



(c) A crossbar implementation



(d) A generalized pipenet model

Fig. 8.20 The concept of a pipenet and its implementation model (Reprinted from Hwang and Xu, *IEEE Transactions on Computers*, Jan. 1988)

In other words, the above CVF demands a chaining degree of 11 if one considers implementing it with a chain of memory-access and arithmetic pipelines. This high degree of chaining is very difficult to implement with a limited number of FPs and vector registers. However, the CVF can be easily implemented with a pipenet as shown in Fig. 8.20b.

Six FPs are employed to implement the seven vector operations because the product vector B(I) × C(I), once generated, can be used in both the denominator and the numerator. We assume two, four, and six pipeline stages in the ADD, MPY, and DIV units, respectively. Two noncompute delays are being inserted, each with two clock delays, along two of the connecting paths. The purpose is to equalize all the path delays from the input end to the output end.

The connections among the FPs and the two inserted delays are shown in Fig. 8.20c for a crossbar-connected vector processor. The feedback connections are identified by numbers. The delays are set up in the appropriate buffers at the output terminals identified as 4 and 5. Usually, these buffers allow a range of delays to be set up at the time the resources are scheduled.

The program graph can be specified either by the programmer or by a compiler. Various connection patterns in the crossbar networks can be prestored for implementing each CVF type. Once the CVF is decoded, the connect pattern is enabled for setup dynamically.

Program Graph Transformations The program in Fig. 8.20a is acyclic or loopfree without feedback connections. An almost trivial mapping is used to establish the pipenet (Fig. 8.20b). In general, the mapping cannot be obtained directly without some graph transformations. We describe these transformations below with a concrete example CVF, corresponding to a cyclic graph shown in Fig. 8.21a.

On a directed program graph, nodal delays correspond to the appropriate FPs, and edge delays are the signal flow delays along the connecting path between FPs. For simplicity, each delay is counted as one pipeline clock cycle.

A *cycle* in a graph is a sequence of nodes and edges whish starts and ends with the same node. We will consider a k-graph, a *synchronous program graph* in which all nodes have a delay of k cycles. A 0-graph is called a *systolic program graph.*

The following two lemmas provide basic tools for converting a given program graph into an equivalent graph. The equivalence is defined up to graph isomorphism and with the same input/output behaviors.

Lemma 1: Adding k delays to any node in a systolic program graph and then subtracting k delays from all incoming edges to that node will produce an equivalent program graph.

Lemma 2: An equivalent program graph is generated if all nodal and edge delays are multiplied by the same positive integer, called the *scaling constant.*

To implement a CVF by setting up a pipenet in a vector processor, one needs first to represent the CVF as a systolic graph with zero delays and positive edge delays. Only a systolic graph can be converted to a pipenet as exemplified below.

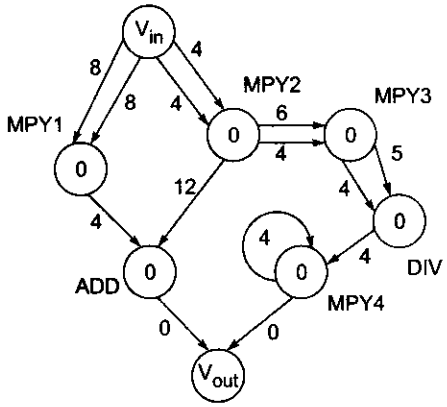
Example 8.9 Program graph transformation to set up a pipenet (Hwang and Xu, 1988)

Consider the systolic program graph in Fig. 8.21a. This graph represents the following set of CVFs:

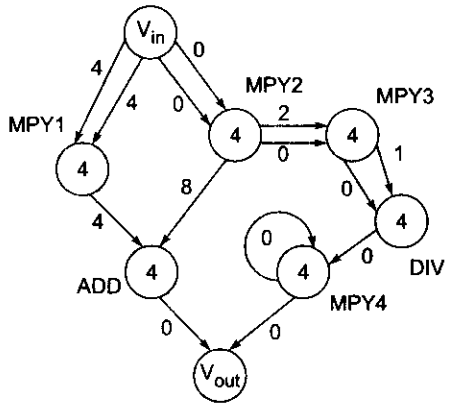
$$
\begin{aligned}
E(I) &= [B(I) \times C(I)] + [C(I) \times D(I)] \\
F(I) &= [C(I) \times D(I)] \times [C(I-2) \times D(I-2)] \\
G(I) &= [F(I)/F(I-1)] \times G(I-4)
\end{aligned} \tag{8.18}
$$

Two multiply operators (MPY1 and MPY2) and one add operator (ADD) are applied to evaluate the vector E(I) from the input end (V_{in}) to the output end (V_{out}) in Fig. 8.21a. The same operator MP2 is applied twice, with different delays (four and six cycles), before it is multiplied by MPY3 to generate the output vector F(I). Finally, the divide (DIV) and multiply (MPY 4) operators are applied to generate the output vector G(I).

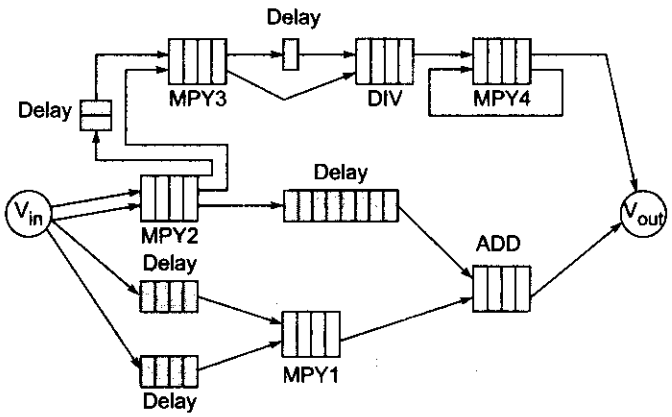
Applying Lemma 1, we add four-cycle delays to each operator node and subtract four-cycle delays from all incoming edges. The transformed graph is obtained in Fig. 8.21b. This is a 4-graph with all nodal delays equal to four cycles. Therefore, one can construct a pipenet with all FPs having four pipeline stages as shown in Fig. 8.21c. The two graphs shown in Figs. 8.21b and 8.21c are indeed isomorphic.



(a) A systolic program graph



(b) After graph transformation



(c) Pipenet implementation with inserted delays between pipelines

Fig. 8.21 From synchronous program graph to pipenet implementation (Reprinted from Hwang and Xu, *IEEE Transactions on Computers*, Jan. 1988)

The inserted delays correspond to the edge delays on the transformed graph. These delays can be implemented with programmable delays in the buffered crossbar networks shown in Fig. 8.20a. Note that the only self-reflecting cycle at node MPY4 represents the recursion defined in the equation for vector G(I). No scaling is applied in this graph transformation.

The systolic program graph in Fig. 8.21a can be obtained by intuitive reasoning and delay analysis as shown above. Systematic procedures needed to convert any set of CVFs into systolic program graphs were reported in the original paper by Hwang and Xu (1988).

If the systolic graph so obtained does not have enough edge delays to be transferred into the operator nodes, we have to multiply the edge delays by a scaling constant s, applying Lemma 2. Then the pipenet clock rate must be reduced by s times. This means that successive vector elements entering the pipenet must be separated by s cycles to avoid collisions in the respective pipelines.

Performance Evaluation The above graph transformation technique has been applied in developing various pipenets for implementing CVFs embedded in Livermore loops. Speedup improvements of between 2 and 12 were obtained, as compared with implementing them on vector hardware without chaining or networking.

In order to build into future vector processors the capabilities of multipipeline networking described above, Fortran and other vector languages must be extended to represent CVFs under various conditions.

Automatic compiler techniques need to be developed to convert from vector expressions to systolic graphs and then to pipeline nets. Therefore, new hardware and software mechanisms are needed to support compound vector processing. This hardware approach can be one or two orders of magnitude faster than the software implementation.

8.4 SIMD COMPUTER ORGANIZATIONS

Vector processing can also be carried out by SIMD computers as introduced in Section 1.3. Implementation models and two example SIMD machines are presented below. We examine their interconnection networks, processing elements, memory, and I/O structures.

Note 8.1 Current status of the SIMD system model

Huge advances in processor technology and processor interconnect technology have taken place over the last two decades. These advances have resulted in the dominance of MIMD and SPMD architectures for high performance systems, rather than the SIMD architecture which was developed at an earlier stage. As a case study, this shift can be seen in how the erstwhile Thinking Machines Corporation changed its architectural model as it went from CM-2 to CM-5 (see Sub-section 8.4.2 and Section 8.5).

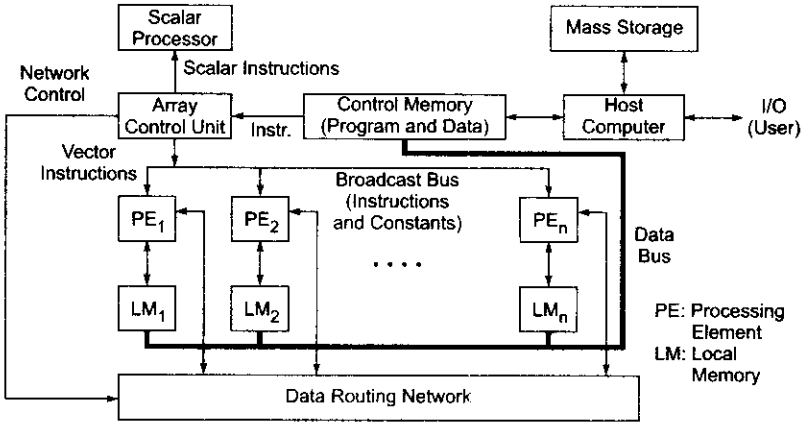
Possibly other than in specialized research platforms, no computer system of the original SIMD model is in operation today. However, a study of this model of computer system can still serve the twin purpose of bringing out (i) the basic SIMD concept and its related issues, and (ii) an important historical perspective on the development of computer architecture. Of course, in a specific course on the subject of computer architecture, the teacher must make the final decision on the amount of time to be devoted to this particular model of computation.

8.4.1 Implementation Models

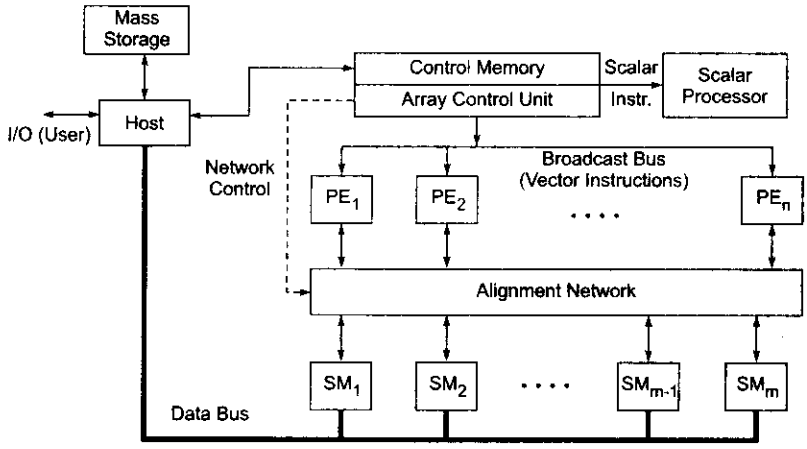
Two SIMD computer models are described below based on the memory distribution and addressing scheme used. Most SIMD computers use a single control unit and distributed memories, except for a few that use associative memories.

The instruction set of an SIMD computer is decoded by the array control unit. The *processing elements* (PEs) in the SIMD array are passive ALUs executing instructions broadcast from the control unit. All PEs must operate in lockstep, synchronized by the same array controller.

Distributed-Memory Model Spatial parallelism is exploited among the PEs in an SIMD computer. A distributed-memory SIMD computer consists of an array of PEs which are controlled by the same array control unit, as shown in Fig. 8.22a. Program and data are loaded into the control memory through the host computer.



(a) Using distributed local memories (e.g. the Illiac IV)



(b) Using shared-memory modules (e.g. the BSP)

Fig. 8.22 Two models for constructing SIMD supercomputers

An instruction is sent to the control unit for decoding. If it is a scalar or program control operation, it will be directly executed by a scalar processor attached to the control unit. If the decoded instruction is a vector operation, it will be broadcast to all the PEs for parallel execution.

Partitioned data sets are distributed to all the local memories attached to the PEs through a vector data bus. The PEs are interconnected by a data-routing network which performs inter-PE data communications such as shifting, permutation, and other routing operations. The data-routing network is under program control through the control unit. The PEs are synchronized in hardware by the control unit.

In other words, the same instruction is executed by all the PEs in the same cycle. However, masking logic is provided to enable or disable any PE from participation in a given instruction cycle. The Illiac IV was such an early SIMD machine consisting of 64 PEs with local memories interconnected by an 8×8 mesh with wraparound connections (Fig. 2.18b).

Almost all SIMD machines built have been based on the distributed-memory model. Various SIMD machines differ mainly in the data-routing network chosen for inter-PE communications. The four-neighbor mesh architecture has been the most popular choice in the past. Besides Illiac IV, the Goodyear MPP and AMT DAP610 were also implemented with the two-dimensional mesh. Variations from the mesh are the hypercube embedded in a mesh implemented in the CM-2, and the X-Net plus a multistage crossbar router implemented in the MasPar MP-1.

Shared-Memory Model In Fig. 8.22b, we show a variation of the SIMD computer using shared memory among the PEs. An alignment network is used as the inter-PE memory communication network. Again this network is controlled by the control unit.

The Burroughs Scientific Processor (BSP) had adopted this architecture, with $n = 16$ PEs updating $m = 17$ shared-memory modules through a 16×17 alignment network. It should be noted that the value m is often chosen to be relatively prime with respect to n, so that parallel memory access can be achieved through skewing without conflicts.

The alignment network must be properly set to avoid access conflicts. Most SIMD computers were built with distributed memories. Some SIMD computers used bit-slice PEs, such as the DAP610 and CM/200. Both bit-slice and word-parallel SIMD computers are studied below.

SIMD Instructions SIMD computers execute vector instructions for arithmetic, logic, data-routing, and masking operations over vector quantities. In bit-slice SIMD machines, the vectors are nothing but binary vectors. In word-parallel SIMD machines, the vector components are 4- or 8-byte numerical values.

All SIMD instructions must use vector operands of equal length n, where n is the number of PEs. SIMD instructions are similar to those used in pipelined vector processors, except that temporal parallelism in pipelines is replaced by spatial parallelism in multiple PEs.

The data-routing instructions include permutations, broadcasts, multicasts, and various rotate and shift operations. Masking operations are used to enable or disable a subset of PEs in any instruction cycle.

Host and I/O All I/O activities are handled by the host computer in the above SIMD organizations. A special control memory is used between the host and the array control unit. This is a staging memory for holding programs and data.

Divided data sets are distributed to the local memories (Fig. 8.22a) or to the shared memory modules (Fig. 8.22b) before starting the program execution. The host manages the mass storage and graphics display of computational results. The scalar processor operates concurrently with the PE array under the coordination of the control unit.

8.4.2 The CM-2 Architecture

The Connection Machine CM-2 produced by Thinking Machines Corporation was a fine-grain MPP computer using thousands of bit-slice PEs in parallel to achieve a peak processing speed of above 10 Gflops. We describe the parallel architecture built into the CM-2. Parallel software developed with the CM-2 will be discussed in Chapter 10.

Program Execution Paradigm All programs started execution on a *front-end*, which issued microinstructions to the back-end processing array when data-parallel operations were desired. The *sequencer* broke down these microinstructions and broadcast them to all *data processors* in the array.

Data sets and results could be exchanged between the front-end and the processing array in one of three ways: *broadcasting*, *global combining*, and *scalar memory bus* as depicted in Fig. 8.23. Broadcasting was carried out through the broadcast bus to all data processors at once.

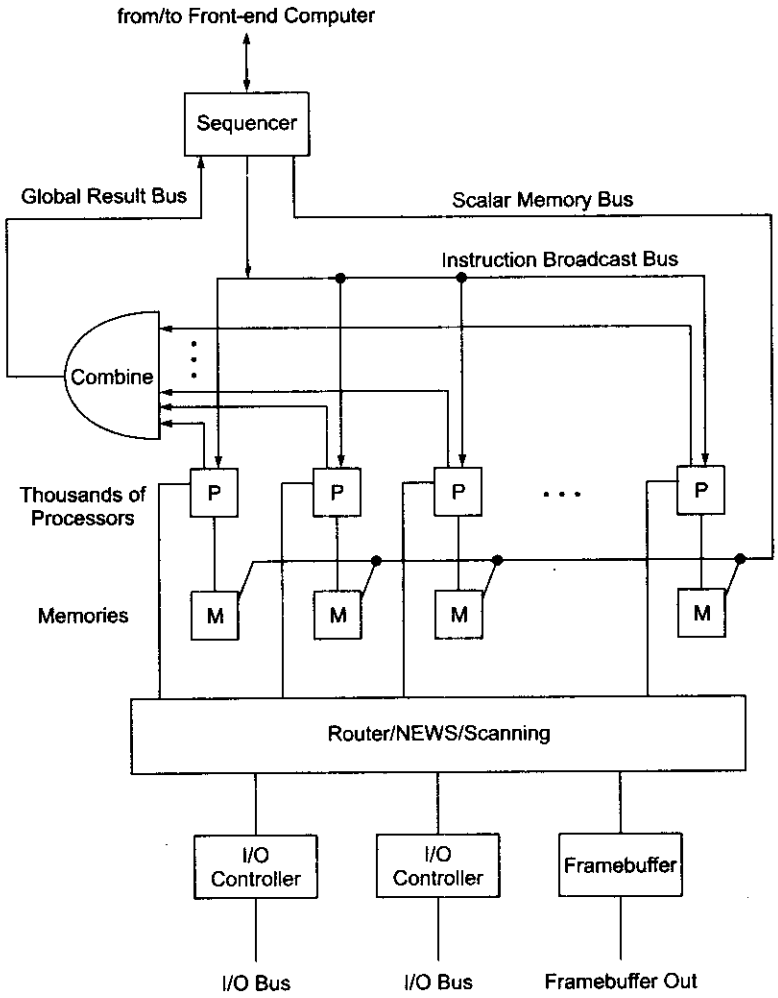


Fig. 8.23 The architecture of the Connection Machine CM-2 (Courtesy of Thinking Machines Corporation, 1990)

Global combining allowed the front-end to obtain the sum, largest value, logical OR, etc., of values, one from each processor. The scalar bus allowed the front-end to read or to write one 32-bit value at a time from or to the memories attached to the data processors. Both VAX and Symbolics Machines were used as the front-end and as hosts.

The Processing Array The CM-2 was a back-end machine for data-parallel computation. The processing array contained from 4K to 64K bit-slice data processors (or PEs), all of which were controlled by a sequencer as shown in Fig. 8.23.

The sequencer decoded microinstructions from the front-end and broadcast nanoinstructions to the processors in the array. All processors could access their memories simultaneously. All processors executed the broadcast instructions in a lockstep manner.

The processors exchanged data among themselves in parallel through the *router*, *NEWS grids*, or a *scanning* mechanism. These network elements were also connected to I/O interfaces. A mass storage subsystem, called the *data vault*, was connected through the I/O for storing up to 60 Gbytes of data.

Processing Nodes Figure 8.24 shows the CM-2 processor chips with memory and floating-point chips. Each data processing node contained 32 bit-slice data processors, an optional floating-point accelerator, and interfaces for interprocessor communication. Each data processor was implemented with a 3-input and 2-output bit-slice ALU and associated latches and a memory interface. This ALU could perform bit-serial full-adder and Boolean logic operations.

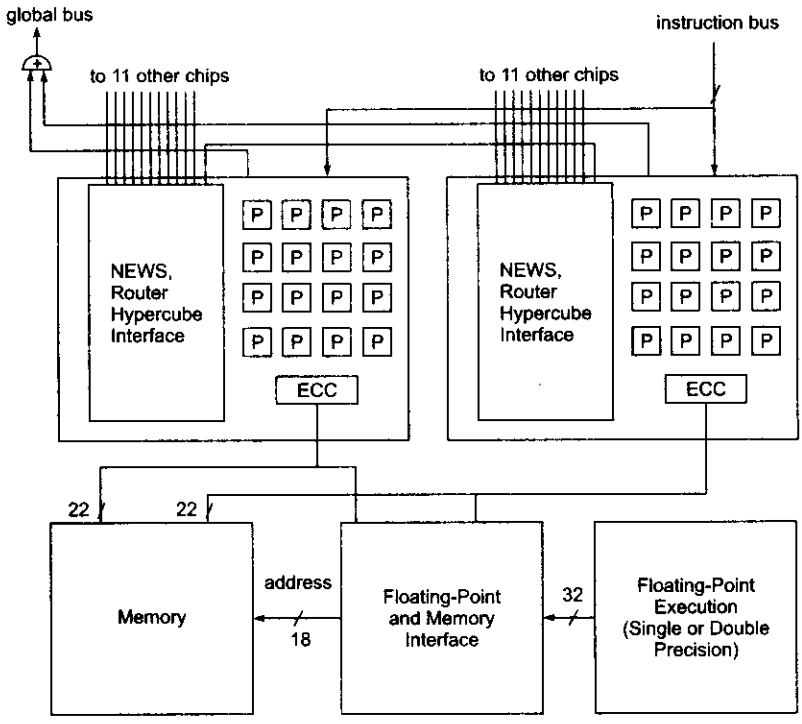


Fig. 8.24 A CM-2 processing node consisting of two processor chips and some memory and floating-point chips (Courtesy of Thinking Machines Corporation, 1990)

The processor chips were paired in each node sharing a group of memory chips. Each processor chip contained 16 processors. The parallel instruction set, called *Paris*, included nanoinstructions for memory load and store, arithmetic and logical, and control of the router, NEWS grid, and hypercube interface, floating-point, I/O, and diagnostic operations.

The memory data path was 22 bits (16 data and 6 ECC) per processor chip. The 18-bit memory address allowed 2^{18} = 256K memory words (512 Kbytes of data) shared by 32 processors. The floating-point chip handled 32-bit operations at a time. Intermediate computational results could be stored back into the memory for subsequent use. Note that integer arithmetic was carried out directly by the processors in a bit-serial fashion.

Hypercube Routers Special hardware was built on each processor chip for data routing among the processors. The router nodes on all processor chips were wired together to form a Boolean n-cube. A full configuration of CM-2 had 4096 router nodes on processor chips interconnected as a 12-dimensional hypercube.

Each router node was connected to 12 other router nodes, including its paired node (Fig. 8.24). All 16 processors belonging to the same node were equally capable of sending a message from one vertex to any other processor at another vertex of the 12-cube. The following example clarifies this message-passing concept.

Example 8.10 Message routing on the CM-2 hypercube (Thinking Machines Corporation, 1990)

On each vertex of the 12-cube, the processors are numbered 0 through 15. The hypercube routers are numbered 0 through 4095 at the 4096 vertices. Processor 5 on router node 7 is thus identified as the 117th processor in the entire system because $16 \times 7 + 5 = 117$.

Suppose processor 117 wants to send a message to processor 361, which is located at processor 9 on router node 22 ($16 \times 22 + 9 = 361$). Since router node 7 = $(000000000111)_2$ and router node 22 = $(000000010110)_2$, they differ at dimension 0 and dimension 4.

This message must traverse dimensions 0 and 4 to reach its destination. From router node 7, the message is first directed to router node 6 = $(00000000110)_2$ through dimension 0 and then to router node 22 through dimension 4, if there is no contention for hypercube wires. On the other hand, if router 7 has another message using the dimension 0 wire, the message can be routed first through dimension 4 to router 23 = $(000000010111)_2$ and then to the final destination through dimension 0 to avoid channel conflicts.

The NEWS Grid Within each processor chip, the 16 physical processors could be arranged as an 8×2, 1×16, 4×4, $4 \times 2 \times 2$, or $2 \times 2 \times 2 \times 2$ grid, and so on. Sixty four *virtual processors* could be assigned to each physical processor. These 64 virtual processors could be imagined to form a 8×8 grid within the chip.

The "NEWS" grid was based on the fact that each processor has a north, east, west, and south neighbor in the various grid configurations. Furthermore, a subset of the hypercube wires could be chosen to connect the 2^{12} nodes (chips) as a two-dimensional grid of any shape, 64×64 being one of the possible grid configurations.

By coupling the internal grid configuration within each node with the global grid configuration, one could arrange the processors in NEWS grids of any shape involving any number of dimensions. These flexible interconnections among the processors made it very efficient to route data on dedicated grid configurations based on the application requirements.

Scanning and Spread Mechanisms Besides dynamic reconfiguration in NEWS grids through the hypercube routers, the CM-2 had been built with special hardware support for scanning or spreading across NEWS grids. These were very powerful parallel operations for fast data combining or spreading throughout the entire array.

Scanning on NEWS grids combined communication and computation. The operation could simultaneously scan in every row of a grid along a particular dimension for the partial sum of that row, the largest or smallest value, or bitwise OR, AND, or exclusive OR. Scanning operations could be expanded to cover all elements of an array.

Spreading could send a value to all other processors across the chips. A single-bit value could be spread from one chip to all other chips along the hypercube wires in only 75 steps. Variants of scans and spreads were built into the Paris instructions for ease of access.

I/O and Data Vault The Connection Machine emphasized massive parallelism in computing as well as in visualization of computational results. High-speed I/O channels were available from 2 to 16 channels for data and/or image I/O operations. Peripheral devices attached to I/O channels included a data vault, CM-HIPPI system, CM-IOP system, and VMEbus interface controller as illustrated in Fig. 8.23. The data vault was a disk-based mass storage system for storing program files and large data bases.

Major Applications The CM-2 was applied in almost all the MPP and grand challenge applications introduced in Chapter 3. Specifically, the Connection Machine Series was applied in document retrieval using relevance feedback, in memory-based reasoning as in the medical diagnostic system called QUACK for simulating the diagnosis of a disease, and in bulk processing of natural languages.

Other applications of the CM-2 included SPICE-like VLSI circuit analysis and layout, computational fluid dynamics, signal/image/vision processing and integration, neural network simulation and connectionist modeling, dynamic programming, context-free parsing, ray tracing graphics, and computational geometry problems. As the CM-2 was upgraded to the CM-5, the applications domain was expected to expand accordingly.

8.4.3 The MasPar MP-1 Architecture

This was a medium-grain SIMD computer, quite different from the CM-2. Parallel architecture and MP-1 hardware design are described below. Special attention is paid to its interprocessor communication mechanisms.

The MasPar MP-1 The MP-1 architecture consisted of four subsystems: the *PE array*, the *array control unit* (ACU), a *UNIX subsystem* with standard I/O, and a *highspeed I/O subsystem* as depicted in Fig. 8.25a. The UNIX subsystem handled traditional serial processing. The high-speed I/O, working together with the PE array, handled massively parallel computing.

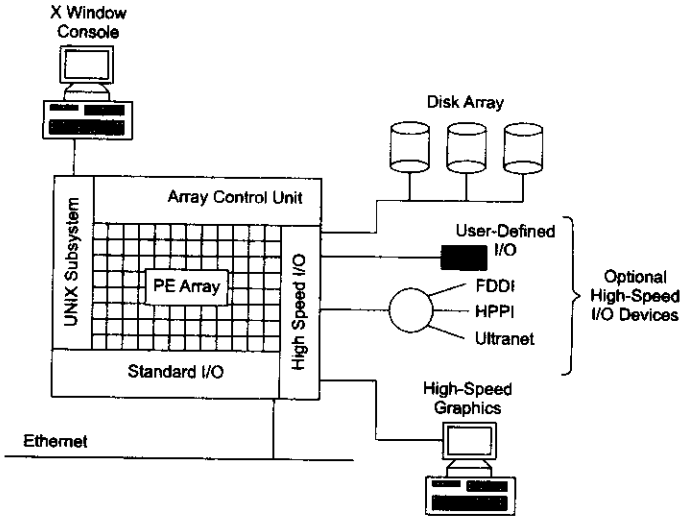
The MP-1 family included configurations with 1024, 4096, and up to 16,384 processors. The peak performance of the 16K-processor configuration was 26,000 MIPS in 32-bit RISC integer operations. The

system also had a peak floating-point capability of 1.5 Gflops in single-precision and 650 Mflops in double-precision operations.

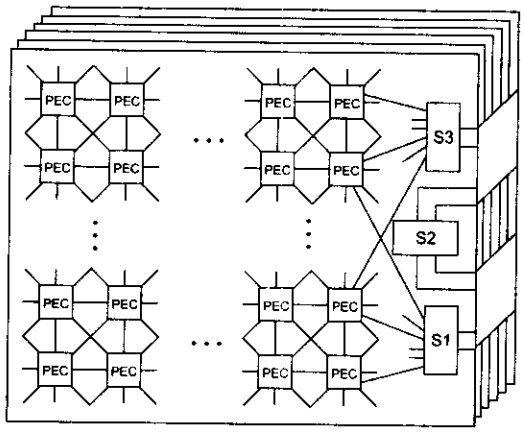
Array Control Unit The ACU was a 14-MIPS scalar RISC processor using a demand-paging instruction memory. The ACU fetched and decoded MP-1 instructions, computed addresses and scalar data values, issued control signals to the PE array, and monitored the status of the PE array.

Like the sequencer in CM-2, the ACU was microcoded to achieve horizontal control of the PE array. Most scalar ACU instructions executed in one 70-ns clock. The whole ACU was implemented on one PC board.

An implemented functional unit, called a *memory machine*, was used in parallel with the ACU. The memory machine performed PE array load and store operations, while the ACU broadcast arithmetic, logic, and routing instructions to the PEs for parallel execution.



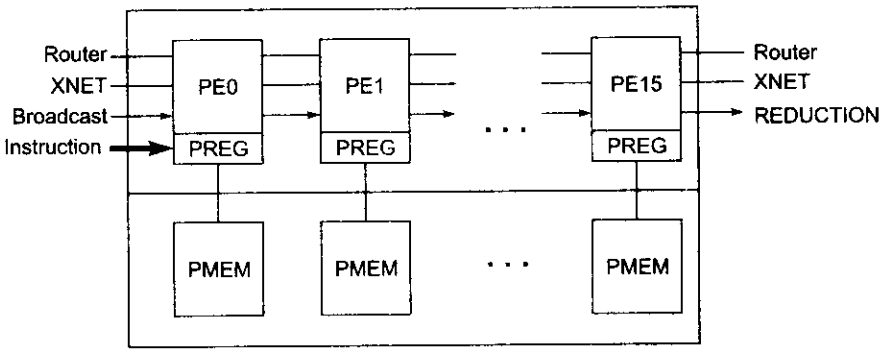
(a) MP-1 System Block Diagram



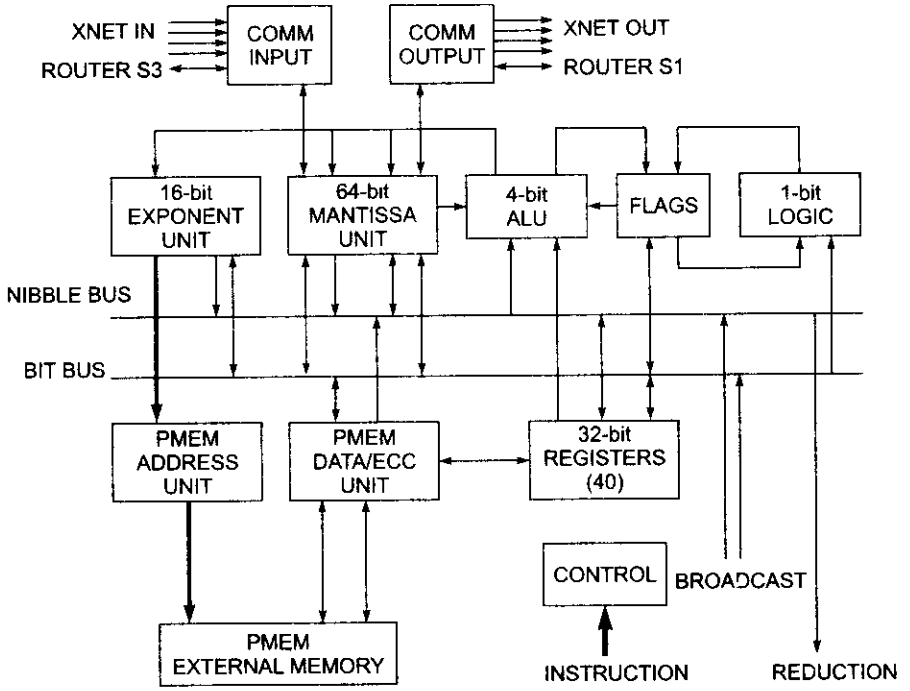
(b) Array of PE clusters

Fig. 8.25 The MasPar MP-1 architecture (Courtesy of MasPar Computer Corporation, 1990)

The PE Array Each processor board had 1024 PEs and associated memory arranged as 64 *PE clusters* (PEC) with 16 PEs per cluster. Figure 8.25b shows the inter-PEC connections on each processor board. Each PEC chip was connected to eight neighbors via the X-Net mesh and a global multistage crossbar router network, labeled S1, S2, and S3 in Fig. 8.25b.



(a) A PE cluster



(b) Processor element and memory

Fig. 8.26 Processing element and memory design in the MasPar MP-1 (Courtesy of MasPar Computer Corporation, 1990)

Each PE cluster (Fig. 8.26a) was composed of 16 PEs and 16 processor memories (PEMs). The PEs were logically arranged as a 4×4 array for the X-Net two-dimensional mesh interconnections. The 16 PEs in a cluster shared an access port to the multistage crossbar router. Interprocessor communications were carried out via three mechanisms:

(1) ACU-PE array communications.
(2) X-Net nearest-neighbor communications.
(3) Global crossbar router communications.

The first mechanism supported ACU instruction/data broadcasts to all PEs in the array simultaneously and performed global reductions on parallel data to recover scalar values from the array. The other two IPC mechanisms are described separately below.

X-Net Mesh Interconnect The X-Net interconnect directly connected each PE with its eight neighbors in the two-dimensional mesh. Each PE had four connections at its diagonal corners, forming an X pattern similar to the BLITZEN X grid network (Davis and Reif, 1986). A tri-state node at each X intersection permitted communication with any of eight neighbors using only four wires per PE.

The connections to the PE array edges were wrapped around to form a 2-D torus. The torus structure is symmetric and facilitates several important matrix algorithms and can emulate a one-dimensional ring with two-X-Net steps. The aggregate X-Net communication bandwidth was 18 Gbytes/s in the largest MP-1 configuration.

Multistage Crossbar Interconnect The network provided global communication between all PEs and formed the basis for the MP-1 I/O system. The three router stages implemented the function of a 1024×1024 crossbar switch. Three router chips were used on each processor board.

Each PE cluster shared an originating port connected to router stage S1 and a target port connected to router stage S3. Connections were established from an originating PE through stages S1, S2, and S3 and then to the target PE. The full MP-1 configuration had 1024 PE clusters, so each stage had 1024 router ports. The router supported up to 1024 simultaneous connections with an aggregate bandwidth of 1.3 Gbytes/s.

Processor Elements and Memory The PE design had mostly data path logic and no instruction fetch or decode logic. The design is detailed in Fig. 8.26b. Both integer and floating-point computations executed in each PE with a register-based RISC architecture. Load and store instructions moved data between the PEM and the register set.

Each PE had forty 32-bit registers available to the programmer and eight 32-bit registers for system use. The registers were bit and byte addressable. Each PE had a 4-bit integer ALU, a 1-bit logic unit, a 64-bit mantissa unit, a 16-bit exponent unit, and a flag unit. The NIBBLE bus was four bits wide and the BIT bus was one bit wide. The PEM could be directly or indirectly addressed with a maximum aggregated memory bandwidth of 12 Gbytes/s.

Most data movement with each PE occurred on the NIBBLE bus and the BIT bus. Different functional units within the PE could be simultaneously active during each microstep. In other words, integer, Boolean, and floating-point operations could all perform at the same time. Each PE ran with a slow clock, while the system speed was obtained through massive parallelism like that implemented in the CM-2.

Parallel Disk Arrays Another feature worthy of mention is the massively parallel I/O architecture implemented in the MP-1. The PE array (Fig. 8.25a) communicated with a parallel disk array through the high-speed I/O subsystem, which was essentially implemented by the 1.3 Gbytes/s global router network.

The disk array provided up to 17.3 Gbytes of formatted capacity with a 9-Mbytes/s sustained disk I/O rate. The parallel disk array was a necessity to support data-parallel computation and provide file system transparency and multilevel fault tolerance.

8.5 THE CONNECTION MACHINE CM-5

Note 8.2 Thinking Machines Corporation

Thinking Machines Corporation (TMC), of Cambridge, Massachusettes, developed its initial SIMD systems CM-1 and CM-2 on the basis of ideas originally developed at MIT and aimed at *artificial intelligence* (AI) applications. The company went out of operation in the mid-1990s. Two innovative computer systems developed by this company are reviewed in this chapter: CM-2 (in Sub-section 8.4.2) and CM-5 (in Section 8.5). From a commercial point of view, none of these systems can be considered successful. However, it would be worthwhile studying the architecture from the point of view of learning about (i) innovative system ideas, (ii) the shift from SIMD to the MIMD system architecture of CM-5, and (iii) the use of a standard RISC processor in an MIMD system with a large number of processors. Many key designers who worked at TMC later worked for other companies, including Sun Microsystems.

The grand challenge applications drive the development of present and future MPP systems to achieve higher and higher performance goals. The Connection Machine model CM-5 was the most innovative effort of Thinking Machines Corporation toward this end. We describe below the innovations surrounding the CM-5 architectural development, its building blocks, and the application paradigms.

8.5.1 A Synchronized MIMD Machine

The CM-2 and its predecessors were criticized for having a rigid SIMD architecture, limiting general-purpose applications. The CM-5 designers liberated themselves by choosing a universal architecture, which combines the advantages of both SIMD and MIMD machines.

Traditionally, supercomputer programmers were forced to choose between MIMD and SIMD computers. An MIMD machine is good at independent branching but bad at synchronization and communication. On the other hand, an SIMD machine is good at synchronization and communication but poor at branching. The CM-5 was designed with a synchronized MIMD structure to support both styles of parallel computation.

The Building Blocks The CM-5 architecture is shown in Fig. 8.27. The machine was designed to contain from 32 to 16,384 *processing nodes*, each of which could have a 32-MHz SPARC processor, 32-Mbytes of memory, and a 128-Mflops vector processing unit capable of performing 64-bit floating-point and integer operations.

Instead of using a single sequencer (as in the CM-2), the system used a number of *control processors*, which were Sun Microsystems workstation computers. The number of control processors, varying with different configurations, ranged from one to several tens. Each control processor was configured with memory and disk based on the needs.

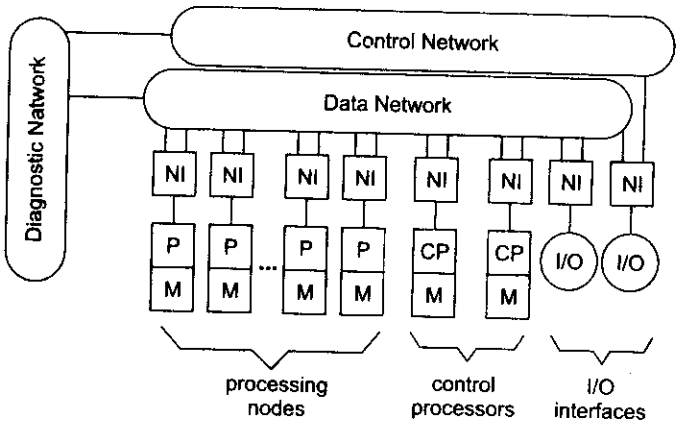


Fig. 8.27 The network architecture of the Connection Machine CM-5 (Courtesy of Leiserson et al, Thinking Machines Corporation, 1992)

Input and output were provided via high-bandwidth *I/O interfaces* to graphics devices, mass secondary storage such as a data vault, and high-performance networks. Additional low-speed I/O was provided by Ethernet connections to the control processors. The largest configuration was expected to occupy a space of 30 m × 30 m, and was designed for a peak performance of over 1 Tflops.

The Network Functions The building blocks were interconnected by three networks: a *data network*, a *control network*, and a *diagnostic network*. The data network provided high-performance, point-to-point data communications between the processing nodes. The control network provided cooperative operations, including broadcast, synchronization, and scans, as well as system management functions.

The diagnostic network allowed "back-door" access to all system hardware to test system integrity and to detect and isolate errors. The data and control networks were connected to processing nodes, control processors, and I/O channels via *network interfaces*.

The CM-5 architecture was considered universal because it was optimized for data-parallel processing of large and complex problems. The data parallelism could be implemented in either SIMD mode, multiple SIMD mode, or synchronized MIMD mode.

The data and control networks were designed to have good *scalability*, making the machine size limited by the affordable cost but not by any architectural or engineering constraint. In other words, the networks depended on no specific types of processors. When new technological advances arrived, they could be easily incorporated into the architecture. The network interfaces were designed to provide an abstract view of the networks.

The System Operations The system operated one or more *user partitions*. Each partition consisted of a control processor, a collection of processing nodes, and dedicated portions of the data and control networks. Figure 8.28 illustrates the distributed control on the CM-5 obtained through the dynamic use of the two interprocessor communication networks. Major system management functions, services, and data distribution are summarized in this diagram.

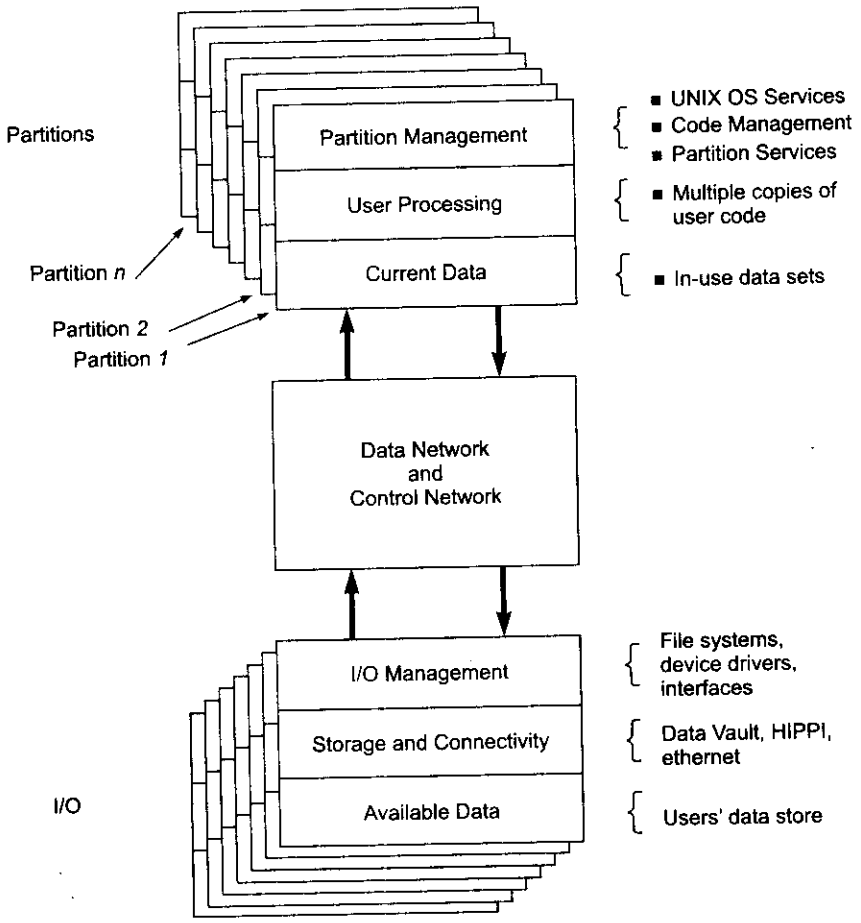


Fig. 8.28 Distributed control on the CM-5 with concurrent user partitions and I/O activities (Courtesy of Thinking Machines Corporation, 1992)

The partitioning of resources was managed by a system executive. The control processor assigned to each partition behaved like a *partition manager*. Each user process executed on a single partition but could exchange data with processes on other partitions. Since all partitions utilized UNIX time-sharing and security features, each allowed multiple users to access the partition, while ensuring no conflicts or interferences.

Access to system functions was classified as either *privileged* or *nonprivileged*. Privileged system functions included access to data and control networks. These accesses could be executed directly by user code without system calls. Thus, OS kernel overhead could be eliminated in network communication within a user task. Access to the diagnostic network, to shared I/O resources, and to other partitions was also privileged and could only be accomplished via system calls.

Some control processors in the CM-5 were assigned to manage the I/O devices and interfaces. This organization allowed a process on any partition to access any I/O device, and ensured that access to one device does not impede access to other devices. Functionally, the system operations, as depicted in Fig. 8.28,

were divided into user-oriented partitions, I/O services based upon system calls, dynamic control of the data and control networks, and system management and diagnostics.

The two networks could download user code from a control processor to the processing nodes, pass I/O requests, transfer messages of all sorts between control processors, and transfer data among nodes and I/O devices, either in a single partition or among different partitions. The I/O capacity could be scaled with increasing numbers of processing nodes or of control partitions. The CM-5 embodied the features of hardware modularity, distributed control, latency tolerance, and user abstraction; all of these are needed for *scalable computing*.

8.5.2 The CM-5 Network Architecture

The data network was based on the *fat-tree* concept introduced by Leiserson (1985). We explain below how it is applied in CM-5 construction. Then we describe the major operations on the control network. Finally, the structure of the diagnostic network is discussed.

Fat Trees A fat tree is more like a real tree in that it becomes thicker as it acquires more leaves. Processing nodes, control processors, and I/O channels are located at the leaves of a fat tree. A *binary fat tree* was illustrated in Fig. 2.17c. The internal nodes are switches. Unlike an ordinary binary tree, the channel capacities of a fat tree increase as we ascend from leaves to root.

The hierarchical nature of a fat tree can be exploited to give each user partition a dedicated subtree, which cannot be interfered with by any other partition's message traffic. The CM-5 data network was actually implemented with a 4-ary fat tree as shown in Fig. 8.29. Each of the internal switch nodes was made up of several router chips. Each router chip was connected to four child chips and either two or four parent chips.

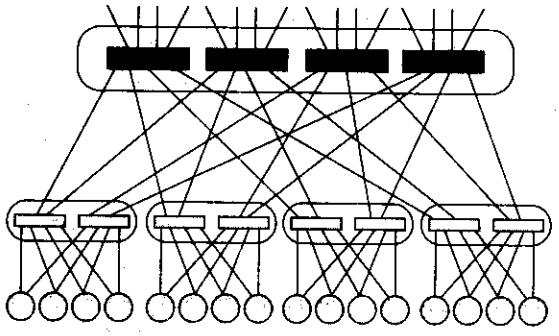


Fig. 8.29 CM-5 data network implemented with a 4-ary fat tree (Courtesy of Leiserson et al, Thinking Machines Corporation, 1992)

To implement the partitions, one could allocate different subtrees to handle different partitions. The size of the subtrees varied with different partition demands. The I/O channels were assigned to another subtree, which was not devoted to any user partition. The I/O subtree was accessed as shared system resource. In many ways, the data network functioned like a hierarchical system bus, except that there was no interference among partitioned subtrees. All leaf nodes had unique physical addresses.

The Data Network To route a message from one processor node to another, the message was sent up the tree to the least common ancestor of the two processors and then down to the destination.

In the 4-ary fat-tree implementation (Fig. 8.29) of the data network, each connection provided a link to another chip with a raw bandwidth of 20 Mbytes/s in each direction. By selecting at each level of the tree whether two or four parent links are used, the bandwidths between nodes in the fat tree could be adjusted. Flow control was provided on each link.

Each processor had two connections to the data network, corresponding to a raw bandwidth of 40 Mbytes/s in and out of each leaf node. In the first two levels, each router chip used only two parent connections to the next higher level, yielding an aggregate bandwidth of 160 Mbytes/s out of a subtree with 16 leaf nodes. All router chips higher than the second level used four parent connections, which yielded an aggregate bandwidth of 10 Gbytes/s in each direction, from one half of a 2K-node system to the other.

The bandwidth continued to scale linearly up to 16,384 nodes, the largest CM-5 configuration planned. In larger machines, transmission-line techniques were to be used to pipeline bits across long wires, thereby overcoming the bandwidth limitation that would otherwise be imposed by wire latency.

As a message went up the tree, it would have several choices as to which parent connection to take. The decision was resolved by pseudo-randomly selecting from among those links that were unobstructed by other messages. After reaching the least common ancestor of the source and destination nodes, the message took a single available path of links down to the destination. The pseudo-random choice at each level automatically balanced the load on the network and avoided undue congestion caused by pathological message sets.

The data network chips were driven by a 40-MHz clock. The first two levels were routed through backplanes. The wires on higher levels were routed through cables, which could be either 9 or 26 ft in length. Message routing was based on the wormhole concept discussed in Section 7.4.

Faulty processor nodes or connection links could be mapped out of the system and quarantined. This allowed the system to remain functional while servicing and testing the mapped-out portion. The data network was acyclic from input to output, which precluded deadlock from occurring if the network promised to eventually deliver all messages injected into it and the processors promised to eventually remove all messages from the network after they were successfully delivered.

The Control Network The architecture of the control network was that of a complete binary tree with all system components at the leaves. Each user partition was assigned to a subtree of the network. Processing nodes were located at leaves of the subtree, and a control processor was mapped into the partition at an additional leaf. The control processor executed scalar part of the code, while the processing nodes executed the data-parallel part.

Unlike the variable-length messages transmitted by the data network, control network packets had a fixed length of 65 bits. There were three major types of operations on the control network: *broadcasting*, *combining*, and *global operations*. These operations provided interprocessor communications. Separate FIFOs in the network interface were assigned to each type of control operations.

The control network provided the mechanisms allowing data-parallel code to be executed efficiently and supported MIMD execution for general-purpose applications. The binary tree architecture made the control network simpler to implement than the fat tree used in the data network. The control network had the additional switching capability to map around faults and to connect any of the control processors to any user partition using an off-line routing strategy.

The Diagnostic Network This network was needed for upgrading system availability. Built-in testability was achieved with scan-based diagnostics. Again, this network was organized as a (not necessarily complete) binary tree for its simplicity in addressing. One or more *diagnostic processors* were at the root. The leaves were *pods*, and each pod was a physical system, such as a board or a backplane. There was a unique path from the root to each pod being tested.

The diagnostic network allowed groups of pods to be addressed according to a "hypercube-address" scheme. A special diagnostic interface was designed to form an in-system check of the integrity of all CM-5 chips that supported the JTAG (Joint Test Action Group) standard and all networks. It provided scan access to all chips supporting the JTAG standard and programmable ad hoc access to non-JTAG chips. The network itself was completely testable and diagnosable. It was able to map out and ignore faulty or power-down parts of the machine.

8.5.3 Control Processors and Processing Nodes

The functional architecture of the control processors and of the processing nodes is described in this subsection.

Control Processor As shown in Fig. 8.30, the basic control processor consisted of a RISC microprocessor (CPU), memory subsystem, I/O with local disks and Ethernet connections, and a CM-5 network interface. This was equivalent to a standard off-the-shelf workstation-class computer system. The network interface connected the control processor to the rest of the system through the control network and the data network.

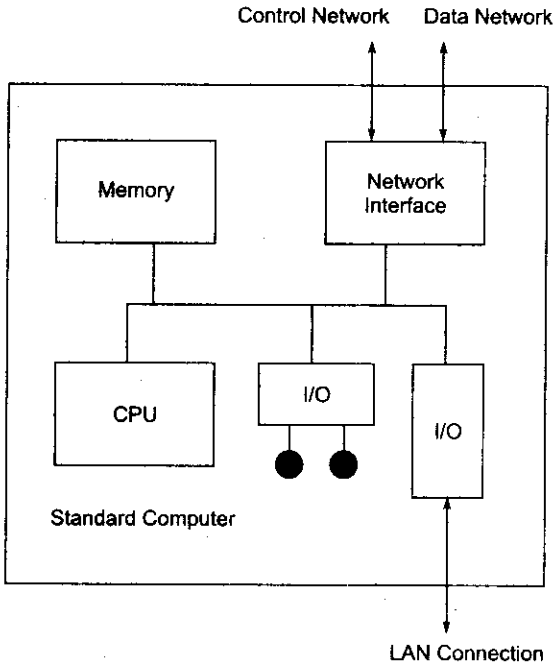


Fig. 8.30 The control processor in the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

Each control processor ran CMOST, a UNIX-based OS with extensions for managing the parallel processing resources of the CM-5. Some control processors managed computational resources in user partitions. Others were used to manage I/O resources. Control processors specialized in managerial functions rather than computational functions. For this reason, high-performance arithmetic accelerators were not needed. Instead, additional I/O connections were provided in control processors.

Processing Nodes Figure 8.31 shows the basic structure of a processing node. It was a SPARC-based processor with a memory subsystem, consisting of a memory controller and 8, 16, or 32 Mbytes of DRAM memory. The internal bus was 64 bits wide.

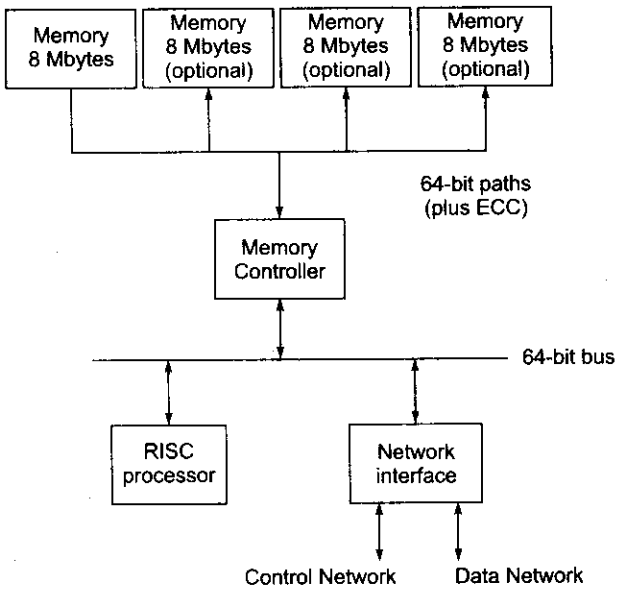


Fig. 8.31 The processing node in the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

The SPARC processor was chosen for its multiwindow feature to facilitate fast context switching. This was very crucial to the dynamic use of the processing nodes in different user partitions at different times. The network interface connected the node to the rest of the system through the control and data networks. The use of a hardware arithmetic accelerator to augment the processor was optional.

Vector Units As illustrated in Fig. 8.32a, vector units could be added between the memory bank and the system bus as an optional feature. The vector units would replace the memory controller in Fig. 8.31. Each vector unit had a dedicated 72-bit path to its attached memory bank, providing a peak memory bandwidth of 128 Mbytes/s per vector unit.

The vector unit executed vector instructions issued by the scalar processor and performed all functions of a memory controller, including generation and check of ECC (error correcting code) bits. As detailed in Fig. 8.32b, each vector unit had a vector instruction decoder, a pipelined ALU, and sixty-four 64-bit registers like a conventional vector processor.

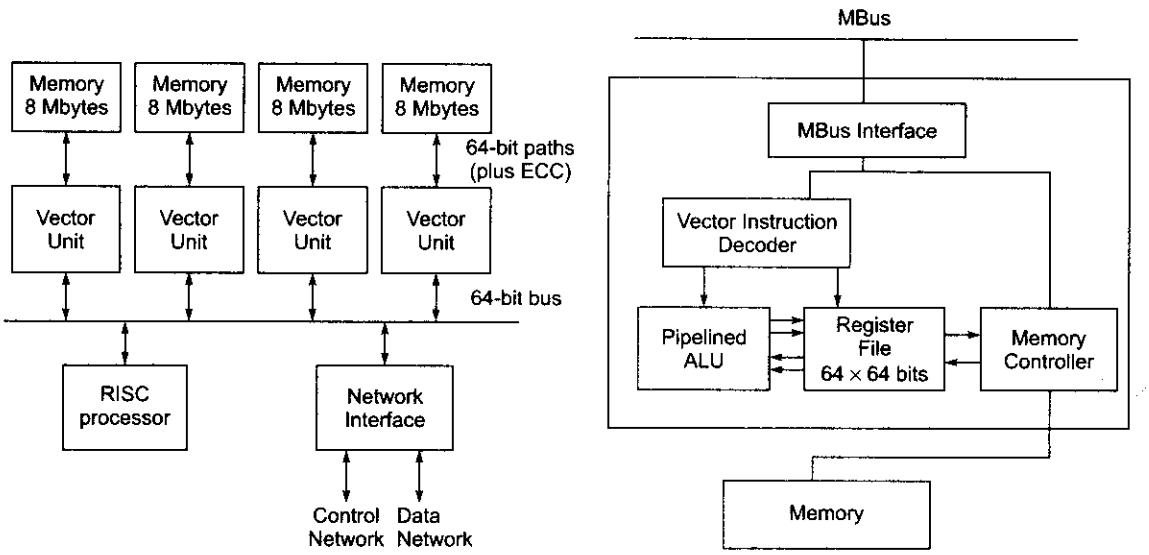


Fig. 8.32 The processing node with vector units in the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

Each vector instruction could be issued to a specific vector unit or pairs of units or broadcast to all four units at once. The scalar processor took care of address translation and loop control, overlapping them with vector unit operations. Together, the vector units provided 512 Mbytes/s memory bandwidth and 128 Mflops 64-bit peak performance per node. In this sense, each processing node of the CM-5 was itself a supercomputer. Collectively, 16K processing nodes would yield a peak performance of $2^{14} \times 2^7 = 2^{21}$ Mflops = 2 Tflops.

Initially, SPARC processors were being used in implementing the control processors and processing nodes. As processor technology advanced, other new processors could be also combined in the system. The network architecture was designed to be independent of the processors chosen except for the network interfaces which would need some minor modifications when new processors were used.

8.5.4 Interprocessor Communications

We have described the high-speed scanning and spreading mechanisms built into the CM-2. In the CM-5, these mechanisms were designed to be further upgraded into four categories of interprocessor communication: *replication, reduction, permutation, parallel prefix.*

These operations could be applied to regular or irregular data sets including vectors, matrices, multidimensional arrays, variable-length vectors, linked lists, and completely irregular patterns. In this section, we describe the key concepts behind these IPC operations. The role of the control network is also identified in these operations.

Replication Recall the *broadcast* operation, where a single value may be replicated to as many copies and distributed to all processors, as illustrated in Fig. 8.33a. Other duplication operations include the *spreading* of a column vector into all the columns of a matrix (Fig. 8.33b), the *expansion* of a short vector into a long vector (Fig. 8.33c), and a completely irregular duplication (Fig. 8.33d).

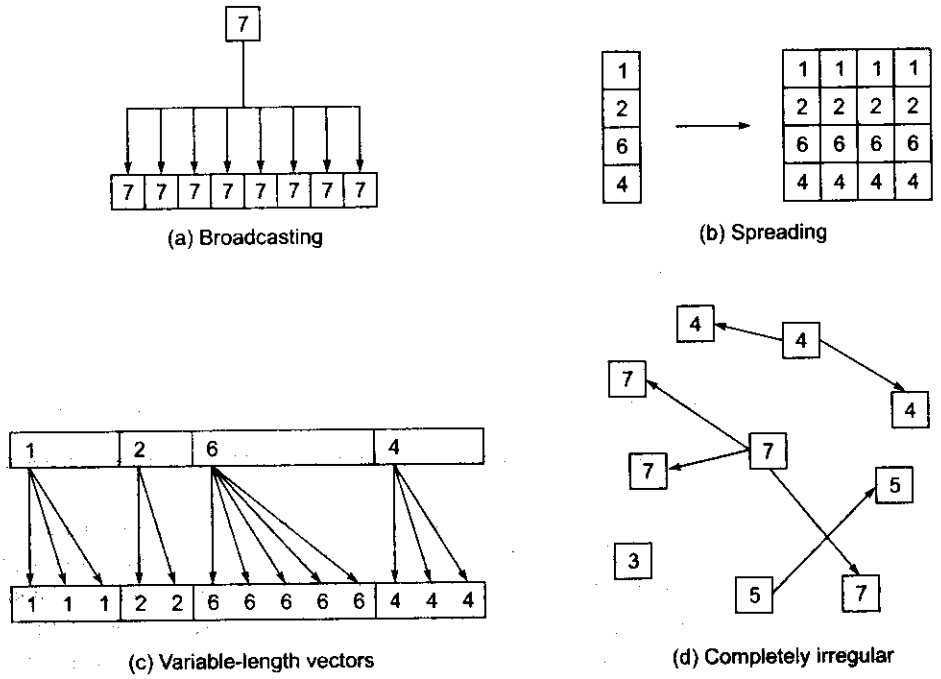


Fig. 8.33 Replication operations for interprocessor communications on CM-5 (Courtesy of Thinking Machines Corporation, 1992)

Replication plays a fundamental role in matrix arithmetic and vector processing, especially on a data-parallel machine. Replication is carried out through the control network in four kinds of broadcasting schemes: *user broadcast*, *supervisor broadcast*, *interrupt broadcast*, and *utility broadcast*. These operations can be used to download code and to distribute data, to implement fast barrier synchronization, and to configure partitions through the OS.

Reduction Vector reduction was implemented on the CM-2 by fast scanning, and on the CM-5 the mechanism was further generalized as the opposite of replication. As illustrated in Fig. 8.34, *global reduce* produces the sum of vector components (Fig. 8.34a). Similarly, the row/column reductions produce the sums per each row or column of a matrix (Fig. 8.34b).

Variable-length vectors were reduced in chunks of a long vector (Fig. 8.34c). The same idea was applied to a completely irregular set as well (Fig. 8.34d). In general, reduction functions include the maximum, the minimum, the average, the dot product, the sum, logical AND, logical OR, etc. Fast scanning and combining are necessities in implementing these operations.

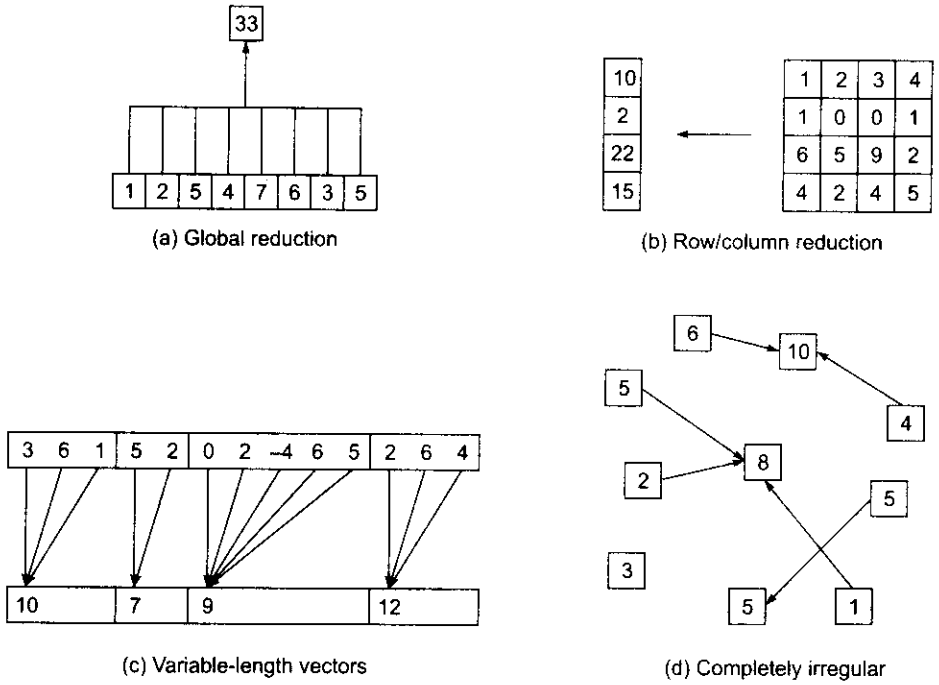


Fig. 8.34 Reduction operations on the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

Four types of combining operations, *reduction*, *forward scan* (parallel prefix), *backward scan* (parallel suffix), and *router done*, were supported by the control network. We will describe parallel prefix shortly. *Router done* refers to the detection of completion of a message-routing cycle, based on Kirchoff's current law, in that the network interfaces keep track of the number of messages entering and leaving the data network. When a round of message sending and acknowledging is complete, the net "current" (messages) in and out of a port should be zero.

Permutation Data-parallel computing relies on permutation for fast exchange of data among processing nodes. Figure 8.35 illustrates four cases of permutations performed on the CM-5. These permutation operations are often needed in matrix transpose, reversing a vector, shifting a multidimensional grid, and FFT butterfly operations.

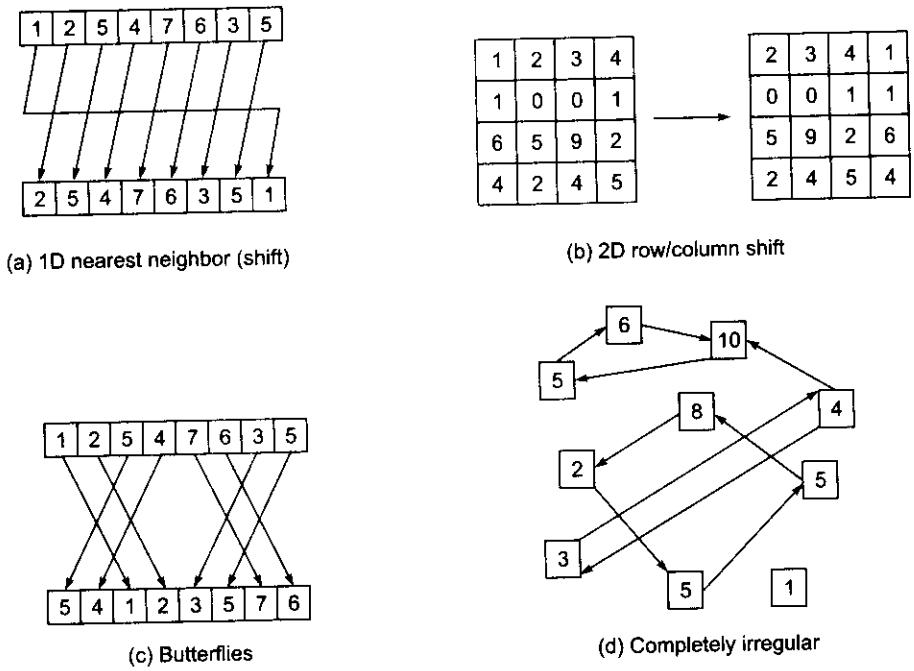


Fig. 8.35 Permutation operations for interprocessor communications on the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

Parallel Prefix This is a kind of combining operation supported by the control network. A *parallel prefix* operation delivers to the ith processor the result of applying one of the five reduction operators to the values in the preceding $i-1$ processors, in the linear order given by data address.

The idea is illustrated in Fig. 8.36 with four examples. Figure 8.36a shows the one-dimensional sum-prefix, in which for example the fourth output 12 is the sum of the first four input elements ($1+2+5+4=12$). The two-dimensional row/column sum-prefix (Fig. 8.36b) can be similarly performed using the forward-scanning mechanism.

Figure 8.36c computes the one-dimensional prefix-sum on sections of a long vector independently. Figure 8.36d shows the forward scanning along linked lists to produce the prefix-sums as outputs.

Many prefix and suffix scanning operations appear to be inherently sequential processes. But the scanning and combining mechanisms on the CM-5 could make the process approximately $\log_2 n$ faster, where n is the array length involved. For example, on the CM-5 a parallel prefix operation on a vector of 1000 entries could be finished in 10 steps instead of 1000 steps.

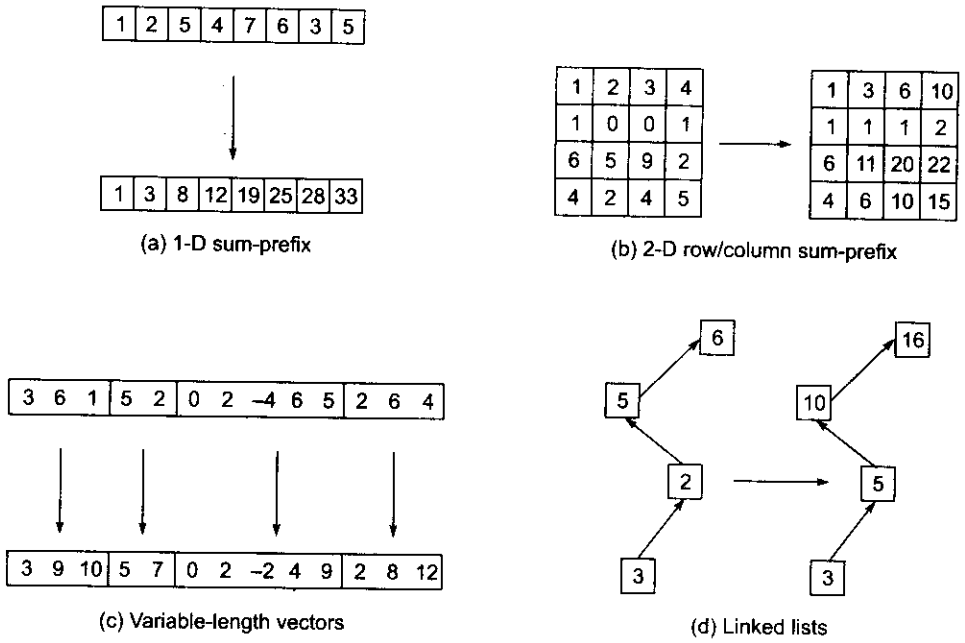


Fig. 8.36 Parallel prefix operations on the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

Summary

By around 1970, computer systems based on the basic single-processor von Neumann architecture had become fairly well established, with products from several computer companies available in the market. In the search for higher processing power, especially for scientific and engineering applications, the earliest supercomputers made heavy use of vector processing concepts, while the concepts of shared-bus multiprocessors and SIMD systems were also beginning to emerge at around that time.

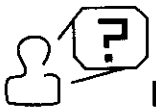
We started this chapter with a study of the basic vector processing concepts, vector instruction types, and interleaved vector memory access schemes. Vector instruction types include vector-vector, vector-scalar, vector-memory, vector reduction, gather and scatter, and masking operations. Examples were studied of the early supercomputers based on vector processing concepts, including systems produced by the two pioneer supercomputer companies Cray and CDC.

Our study of multivector computers—i.e. systems based on multiple vector processors—began with the basic system design rules for achieving the target performance. These design rules can be related to processing power, I/O and networking, memory bandwidth, and scalability. As specific examples, multivector systems and early massively parallel processing (MPP) systems introduced by Cray were studied, as were Fujitsu multivector systems. Also reviewed in brief were mainframe systems provided with vector processing capability, and the so-called mini-supercomputers which emerged with advances in electronic technology.

The concept of compound vector processing arises from the search for more efficient processing of vector data. Scientific and engineering applications make use of such vector operations, and therefore system architects have always looked for ways to map them efficiently onto the underlying vector processing hardware. The concepts of vector loops and chaining, and of multi-pipeline networking, have also been developed with the aim of providing efficient support for compound vector processing.

SIMD computer systems may be of one of two basic types—with distributed memory modules and with shared memory modules. Specific examples were discussed of two innovative SIMD systems: Connection Machine 2 (CM-2), with processors based on bit-slice technology, and MasPar MP-1, with its specially designed processors. Both systems used sophisticated system interconnects and had the capability to connect thousands of processors. However, for good technological reasons, the architectural trend later turned away from SIMD systems and towards massively parallel MIMD (or SPMD) systems.

Connection Machine 5 (CM-5) represents the shift towards massively parallel MIMD architecture which occurred in the mid-1990s. The main factor behind this shift was the availability of low-cost but powerful processors, made possible by rapid advances in the underlying VLSI technology. CM-5 innovations included the use of a large number of RISC processors, a sophisticated data network (using a fat tree), and special hardware features to support efficient and versatile interprocessor communication—which included useful operations such as replication, reduction and permutation.



Exercises

Problem 8.1 Explain the structural and operational differences between register-to-register and memory-to-memory architectures in building multipipelined supercomputers for vector processing. Comment on the advantages and disadvantages in using SIMD computers as compared with the use of pipelined supercomputers for vector processing.

Problem 8.2 Explain the following terms related to vector processing:

(a) Vector and scalar balance point.
(b) Vectorization ratio in user code.
(c) Vectorization compiler or vectorizer.
(d) Vector reduction instructions.
(e) Gather and scatter instructions.
(f) Sparse matrix and masking instruction.

Problem 8.3 Explain the following memory organizations for vector accesses:

(a) S-access memory organization.
(b) C-access memory organization.
(c) C/S-access memory organization.

Problem 8.4 Distinguish among the following vector processing machines in terms of architecture, performance range, and cost-effectiveness:

(a) Full-scale vector supercomputers.
(b) High-end mainframes or near-supercomputers.
(c) Minisupercomputers or supercomputing workstations.

Problem 8.5 Explain the following terms associated with compound vector processing:

(a) Compound vector functions.
(b) Vector loops and pipeline chaining.
(c) Systolic program graphs.
(d) Pipeline network or pipenets.

Problem 8.6 Answer the following questions related to the architecture and operations of the Connection Machine CM-2:

(a) Describe the processing node architecture, including the processor, memory, floating-point unit, and network interface.
(b) Describe the hypercube router and the NEWS grid and explain their uses.
(c) Explain the scanning and spread mechanisms and their applications on the CM-2.
(d) Explain the concepts of broadcasting, global combining, and virtual processors in the use of the CM-2.

Problem 8.7 Answer the following questions about the MasPar MP-1:

(a) Explain the X-Net mesh interconnect (the PE array) built into the MP-1.
(b) Explain how the multistage crossbar router works for global communication between all PEs.
(c) Explain the computing granularity on PEs and how fast I/O is performed on the MP-1.

Problem 8.8 Answer the following questions about the Connection Machine CM-5:

(a) What is a fat tree and its application in constructing the data network in the CM-5?
(b) What are user partitions and their resources requirements?
(c) Explain the functions of the control processors of the control network and of the diagnostic network.
(d) Explain how vector processing is supported in each processing node.

Problem 8.9 Give examples, different from those in Figs. 8.33 through 8.36, to explain the concepts of replication, reduction, permutation, and parallel prefix operations on the CM-5. Check the *Technical Summary of CM-5* published by Thinking Machines Corporation if additional reading is needed.

Problem 8.10 On a Fujitsu VP2000, the vector processing unit was equipped with two load/store pipelines plus five functional pipelines as shown in Fig. 8.13. Consider the execution of the following compound vector function:

$$A(I) = B(I) \times C(I) + D(I) \times E(I) + F(I) \times G(I)$$

for $I = 1, 2, \ldots, N$. Initially, all vector operands are in memory, and the final vector result must be stored in memory.

(a) Show a pipeline-chaining diagram, similar to Fig. 8.18, for executing this CVF.
(b) Show a space-time diagram, similar to Fig. 8.19, for pipelined execution of the CVF. Note that two *vector loads* can be carried out simultaneously on the two vector-access pipes. At the end of computation, one of the two access pipes is used for storing the A array.

Problem 8.11 The following sequence of compound vector function is to be executed on a Cray X-MP type vector processor:

$$A(I) = B(I) + s \times C(I)$$
$$D(I) = s \times B(I) \times C(I)$$
$$E(I) = C(I) \times (C(I) - B(I))$$

where B(I) and C(I) are each 64-element vectors originally stored in memory. The resulting vectors A(I), D(I), and E(I) must be stored back into memory after the computation.

(a) Write 11 vector instructions in proper order to execute the above CVFs on a Cray X-MP type vector processor with two vector-load pipes and one vector-store pipe which can be used simultaneously with the remaining functional pipelines.
(b) Show a space-time diagram, similar to Fig. 8.19, for achieving maximally chained vector operations for executing the above CVFs in minimum time.
(c) Show the potential speedup of the above vector chaining operations over the chaining operations on the Cray 1, which had only one memory-access pipe.

Problem 8.12 Consider a vector computer which can operate in one of two execution modes at a time: one is the *vector mode* with an execution rate of R_v = 2000 Mflops, and the other is the *scalar*

mode with an execution rate of R_s = 200 Mflops. Let α be the percentage of code that is vectorizable in a typical program mix for this computer.

(a) Derive an expression for the *average execution rate* R_a for this computer.
(b) Plot R_a as a function of α in the range (0, 1).
(c) Determine the vectorization ratio α needed in order to achieve an average execution rate of R_a = 1500 Mflops.
(d) Suppose α = 0.7. What value of R_v is needed to achieve R_a = 400 Mflops?

Problem 8.13 Describe an algorithm using *add*, *multiply*, and *data-routing* operations to compute the expression $s = A_1 \times B_1 + A_2 \times B_2 + \ldots + A_{32} \times B_{32}$ with minimum time in each of the following two computer systems. It is assumed that add and multiply require two and four time units, respectively. The time required for instruction/data fetches from memory and decoding delays are ignored. All instructions and data are assumed already loaded into the relevant PEs. Determine the minimum compute time in each case.

(a) A serial computer with a processor equipped with one adder and one multiplier, only one of which can be used at a time. No data-routing operation is needed in this uniprocessor machine.
(b) An SIMD computer with eight PEs (PE_0, PE_1, ..., PE_7), which are connected by a bidirectional circular ring. Each PE can directly *route* its data to its neighbors in one time unit. The operands A_i and B_i are initially stored in $PE_{i \bmod 8}$ for i = 1, 2, ..., 32. Each PE can *add* or *multiply* at different times.

Problem 8.14 Calculate the peak performance in Gflops with reasoning in each of the following two vector supercomputers.

(a) The Cray Y-MP C-90 with 16 vector processors.
(b) The NEC SX-X with 4 vector processors.
(c) Explain why both machines offered a maximum 64-way parallelism in their vector operations.

Problem 8.15 Devise a minimum-time algorithm to multiply two 64 × 64 matrices, $A = (a_{ij})$ and $B = (b_{ij})$, on an SIMD machine consisting of 64 PEs with local memory. The 64 PEs are interconnected by a 2D 8 × 8 torus with bidirectional links.

(a) Show the initial distribution of the input matrix elements (a_{ij}) and (b_{ij}) on the PE memories.
(b) Specify the SIMD instructions needed to carry out the matrix multiplication. Assume that each PE can perform one *multiply*, one *add*, or one *shift* (shifting data to one of its four neighbors) operation per cycle.
You should first compute all the *multiply* and *add* operations on local data before starting to route data to neighboring PEs. The SIMD *shift* operations can be either east, west, south, or north with wraparound connections on the torus.
(c) Estimate the total number of SIMD instruction cycles needed to compute the matrix multiplication. The time includes all arithmetic and data-routing operations. The final product elements $C = A \times B = (c_{ij})$ end up in various PE memories without duplication.
(d) Suppose data duplication is allowed initially by loading the same data element into multiple PE memories. Devise a new algorithm to further reduce the SIMD execution time. The initial data duplication time, using either data *broadcast* instructions or data *routing* (shifting) instructions, must be counted. Again, each result element c_{ij} ends up in only one PE memory.

Problem 8.16 Compare the Connection Machines CM-2 and CM-5 in their architectures, operation modes, functional capabilities, and potential performance, from the viewpoints of a computer architect and of a machine programmer.

Problem 8.17 Consider the use of a multivector multiprocessor system for computing the following linear combination of n vectors:

$$\mathbf{y} = \sum_{j=0}^{1023} a_j \times \mathbf{x}_j$$

where $\mathbf{y} = (y_0, y_1, \ldots, y_{1023})^T$ and $\mathbf{x}_j = (x_{0j}, x_{1j}, \ldots, x_{1023,j})^T$ for $0 \leq j \leq 1023$ are column vectors; $\{a_j | 0 \leq j \leq 1023\}$ are scalar constants. You are asked to implement the above computations on a four-processor system with shared memory. Each processor is equipped with a vector-add pipeline and a vector-multiply pipeline. Assume four pipeline stages in each functional pipeline.

(a) Design a minimum-time parallel algorithm to perform concurrent vector operations on the given multiprocessor, ignoring all memory-access and I/O operations.

(b) Compare the performance of the multiprocessor algorithm with that of a sequential algorithm on a uniprocessor without the pipelined vector hardware.

Problem 8.18 The Burroughs Scientific Processor (BSP) was built as an SIMD computer consisting of 16 PEs accessing 17 shared memory modules. Prove that conflict-free memory access can be achieved on the BSP for vectors of an arbitrary length with a stride which is not a multiple of 17.

9

Scalable, Multithreaded, and Dataflow Architectures

This chapter discusses innovative computers built with scalable, multithreaded, or dataflow architectures. These architectures generated and validated many research ideas which led to the latter development of massively parallel processing (MPP) systems. Therefore, the material is presented with a strong research flavor benefiting mostly researchers, designers, and graduate students. More recent developments of these ideas are presented in Chapter 13.

Major research issues covered include latency-hiding techniques, principles of multithreading, multidimensional scalability, multithreaded architectures, fine-grain multicomputers, dataflow, and hybrid architectures. Example systems studied include the Stanford Dash, Wisconsin Multicube, USC/OMP, KSR-1, Tera, MIT Alewife and J-Machine, Caltech Mosaic C, ETL EM-4, and MIT/Motorola *T.

LATENCY-HIDING TECHNIQUES

Massively parallel and scalable systems may typically use distributed shared memory. The access of remote memory significantly increases memory latency. Furthermore, the processor speed has been increasing at a much faster rate than memory speeds. Thus any scalable multiprocessor or large-scale multicomputer must rely on the use of latency-reducing, -tolerating, or -hiding mechanisms. Four latency-hiding mechanisms are studied below for enhancing scalability and programmability.

Latency hiding can be accomplished through four complementary approaches: (i) using *prfetching techniques* which bring instructions or data close to the processor before they are actually needed; (ii) using *coherent caches* supported by hardware to reduce cache misses; (iii) using *relaxed memory consistency* models by allowing buffering and pipelining of memory references; and (iv) using *multiple-contexts* support to allow a processor to switch from one context to another when a long-latency operation is encountered.

The first three mechanisms are described in this section, supported by simulation results obtained by Stanford researchers. Multiple contexts will be treated with multithreaded processors and system architectures in Sections 9.2 and 9.4. However, the effect of multiple contexts is shown here in combination with other latency-hiding mechanisms.

9.1.1 Shared Virtual Memory

Single-address-space multiprocessors/multicomputers must use shared virtual memory. We present a model of such an architectural environment based on the Stanford Dash experience. Then we examine several shared-virtual-memory systems developed at Stanford, Yale, Carnegie-Mellon, and Princeton universities.