FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

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

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

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

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

The McGraw-Hill Companies

ADVANCED COMPUTER ARCHITECTURE

Parallelism, Scalability, Programmability

Second Edition

Kai Hwang

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

Naresh Jotwani

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

Tata McGraw Hill Education Private Limited
NEW DELHI

McGraw-Hill Offices

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

Part III

Parallel and Scalable Architectures

Chapter 7
Multiprocessors and Multicomputers

Chapter 8
Multivector and SIMD Computers

Chapter 9
Scalable, Multithreaded, and Dataflow Architectures

Summary

Part III consists of three chapters dealing with parallel, vector, and scalable architectures for building high-performance computers. The multiprocessor system interconnects studied include crossbar switches, multistage networks, hierarchical buses, and multidimensional ring, mesh, and torus architectures. Three generations of multicomputer developments are reviewed. Then we consider message-passing mechanisms.

Vector supercomputers appear either as pipelined multiprocessors or as SIMD data-parallel computers. We study the architectures of the Cray Y-MP, C-90, Cray/MPP, NEC SX, Fujitsu VP-2000, VPP500, VAX 9000, Hitachi S-820, Stardent 3000, CM-2, MasPar MP-1, and CM-5 for concurrent scalar/vector processing.

Chapter 9 introduces scalable architectures for massively parallel processing applications. These include both von Neumann, fine-grain, multithreaded, and dataflow architectures. Various latency-hiding techniques are described, including the principles of multithreading. Case studies include the Intel Paragon, Stanford Dash, MIT Alewife, J-Machine and *T, Tera computer, KSR-I, Wisconsin Multicube, USC/OMP, ETL EM4, etc.

The McGraw-Hill Companies

7

Multiprocessors and Multicomputers

In this chapter, we study system architectures of multiprocessors and multicomputers. Various cache coherence protocols, synchronization methods, crossbar switches, multiport memory, and multistage networks are described for building multiprocessor systems. Then we discuss multicomputers with distributed memories which are not globally shared. The Intel Paragon is used as a case study. Message-passing mechanisms required with multicomputers are also reviewed. Single-address-space multicomputers will be studied in Chapter 9.

7.1 MULTIPROCESSOR SYSTEM INTERCONNECTS

Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory, I/O, and peripheral devices. Hierarchical buses, crossbar switches, and multistage networks are often used for this purpose.

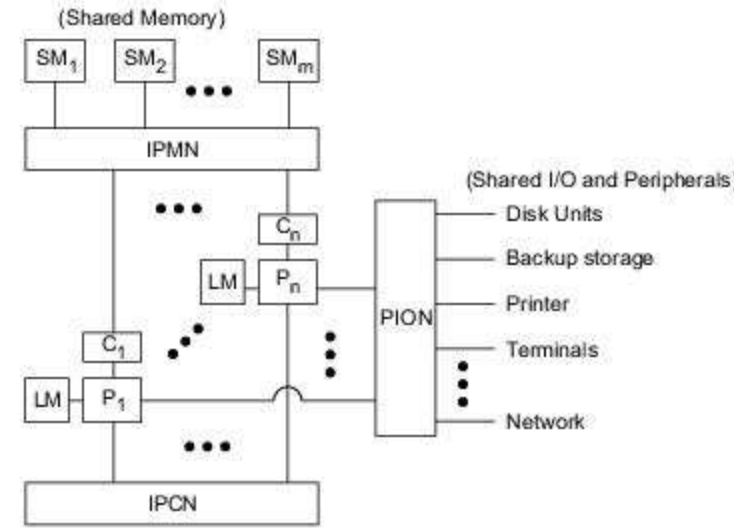
A generalized multiprocessor system is depicted in Fig. 7.1. This architecture combines features from the UMA, NUMA, and COMA models introduced in Section 1.4.1. Each processor P_i is attached to its own local memory and private cache. Multiple processors are connected to shared-memory modules through an interprocessor-memory network (IPMN).

The processors share the access of I/O and peripheral devices through a processor I/O network (PION). Both IPMN and PION are necessary in a shared-resource multiprocessor. Direct interprocessor communications are supported by an optional interprocessor communication network (IPCN) instead of through the shared memory.

Network Characteristics Each of the above types of networks can be designed with many choices. The choices are based on the topology, timing protocol, switching method, and control strategy. Dynamic networks are used in multiprocessors in which the interconnections are under program control. Timing, switching, and control are three major operational characteristics of an interconnection network. The timing control can be either *synchronous* or *asynchronous*. Synchronous networks are controlled by a global clock that synchronizes all network activities. Asynchronous networks use handshaking or interlocking mechanisms to coordinate fast and slow devices requesting use of the same network.

A network can transfer data using either *circuit switching* or *packet switching*. In circuit switching, once a device is granted a path in the network, it occupies the path for the entire duration of the data transfer.

In packet switching, the information is broken into small packets individually competing for a path in the network.



Legends: IPMN (Inter-Processor-Memory Network)
PION (Processor- I/O Network)
IPCN (Inter-Processor Communication Network)
P (Processor)
C (Cache)
SM (Shared Memory)
LM (Local Memory)

Fig. 7.1 Interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory, and shared peripherals

Network control strategy is classified as *centralized* or *distributed*. With centralized control, a global controller receives requests from all devices attached to the network and grants the network access to one or more requesters. In a distributed system, requests are handled by local devices independently.

7.1.1 Hierarchical Bus Systems

A *bus system* consists of a hierarchy of buses connecting various system and subsystem components in a computer. Each bus is formed with a number of signal, control, and power lines. Different buses are used to perform different interconnection functions.

In general, the hierarchy of bus systems are packaged at different levels as depicted in Fig. 7.2, including local buses on boards, backplane buses, and I/O buses.

Local Bus Buses implemented within processor chips or on *printed-circuit* boards are called *local buses*. On a processor board one may find a local bus which provides a common communication path among major components (chips) mounted on the board. A memory board uses a *memory bus* to connect the memory with

the interface logic. An I/O or network interface chip or board uses a *data bus*. Each of these local buses consists of signal and utility lines.

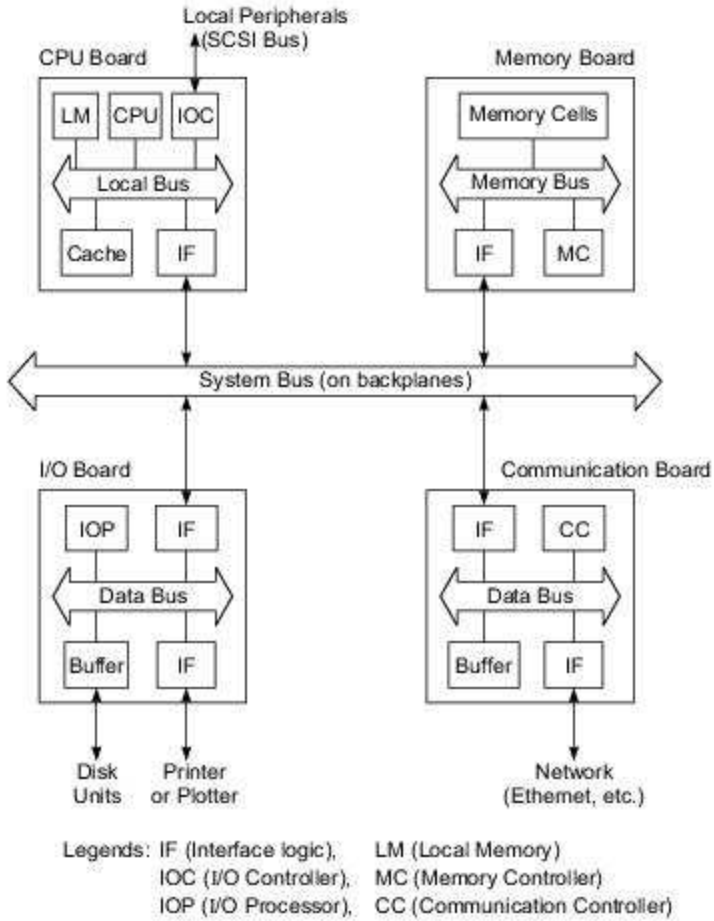


Fig. 7.2 Bus systems at board level, backplane level, and I/O level

Backplane Bus A *backplane* is a printed circuit on which many connectors are used to plug in functional boards. A *system bus*, consisting of shared signal paths and utility lines, is built on the backplane. This system bus provides a common communication path among all plug-in boards.

Several backplane bus standards have been developed over time such as the VME bus (IEEE Standard 1014-1987), Multibus II (IEEE Standard 1296-1987), and Futurebus+ (IEEE Standard 896.1-1991) as introduced in Chapter 5. However, point to-point switched interconnects have emerged as more efficient alternatives, as discussed in Chapters 5 and 13.

I/O Bus Input/output devices are connected to a computer system through an *I/O bus* such as the SCSI (Small Computer Systems Interface) bus. This bus is made of coaxial cables with taps connecting disks,

printer, and other devices to a processor through an I/O controller (Fig. 7.2). Special interface logic is used to connect various board types to the backplane bus.

Complete specifications for a bus system include logical, electrical, and mechanical properties, various application profiles, and interface requirements. Our study will be confined to the logical and application aspects of system buses. Emphasis will be placed on the scalability and bus support for cache coherence and fast synchronization.

For example, the core of the Encore Multimax multiprocessor was the Nanobus, consisting of 20 slots, a 32-bit address, a 64-bit data path, and a 14-bit vector bus, and operating at a clock rate of 12.5 MHz with a total memory bandwidth of 100 Mbytes/s. The Sequent multiprocessor bus had a 64-bit data path, a 10-MHz clock rate, and a 32-bit address, for a channel bandwidth of 80 Mbytes/s. A write-back private cache was used to reduce the bus traffic by 50%.

Digital bus interconnects can be adopted in commercial systems ranging from workstations to minicomputers, mainframes, and multiprocessors. Hierarchical bus systems can be used to build medium-sized multiprocessors with less than 100 processors. However, the bus approach is limited by bandwidth scalability and the packaging technology employed.

Hierarchical Buses and Caches Wilson (1987) proposed a hierarchical cache/ bus architecture as shown in Fig. 7.3. This is a multilevel tree structure in which the leaf nodes are processors and their private caches (denoted P_j and C_{1j} in Fig. 7.3). These are divided into several clusters, each of which is connected through a cluster bus.

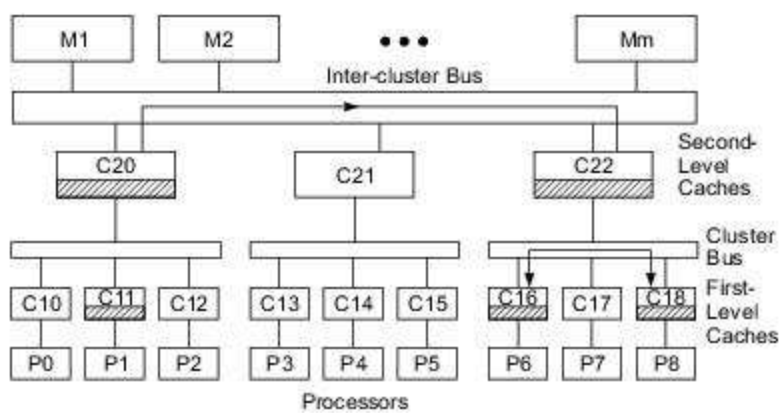


Fig. 7.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor (Courtesy of Wilson; reprinted from *Proc. of Annual Int. Symp. on Computer Architecture, 1987*)

An intercluster bus is used to provide communications among the clusters. Second level caches (denoted as C_{2j}) are used between each cluster bus and the intercluster bus. Each second-level cache must have a capacity that is at least an order of magnitude larger than the sum of the capacities of all first-level caches connected beneath it.

Each single cluster operates as a single-bus system. Snoopy bus coherence protocols can be used to establish consistency among first-level caches belonging to the same cluster. Second-level caches are used to extend consistency from each local cluster to the upper level.

The upper-level caches form another level of shared memory between each cluster and the main memory modules connected to the intercluster bus. Most memory requests should be satisfied at the lower-level caches. Intercluster cache coherence is controlled among the second-level caches and the resulting effects are passed to the lower level.

Example 7.1 Encore Ultramax multiprocessor architecture

The Ultramax had a two-level hierarchical-bus architecture as depicted in Fig. 7.4. The Ultramax architecture was very similar to that characterized by Wilson, except that the global Nanobus was used only for intercluster communications.

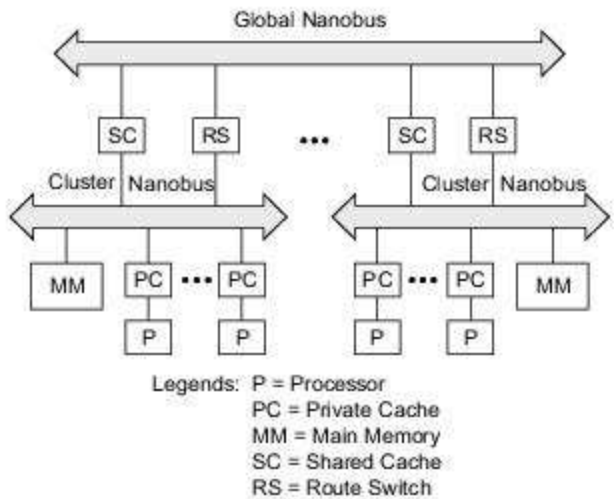


Fig. 7.4 The Ultramax multiprocessor architecture using hierarchical buses with multiple clusters (Courtesy of Encore Computer Corporation, 1987)

The shared memories were distributed to all clusters instead of being connected to the intercluster bus. The cluster caches formed the second-level caches and performed the same filtering and cache coherence control for remote accesses as in Wilson's scheme. When an access request reached the top bus, it would be routed down to the cluster memory that matched it with the reference address.

The idea of using *bridges* between multiprocessor clusters is to allow transactions initiated on a local bus to be completed on a remote bus. As exemplified in Fig. 7.5, multiple buses are used to build a very large system consisting of three multiprocessor clusters. The bus used in this example is Futurebus+, but the basic idea is more general. Bridges are used to interface the clusters. The main functions of a bridge include communication protocol conversion, interrupt handling in split transactions, and serving as cache and memory agents.

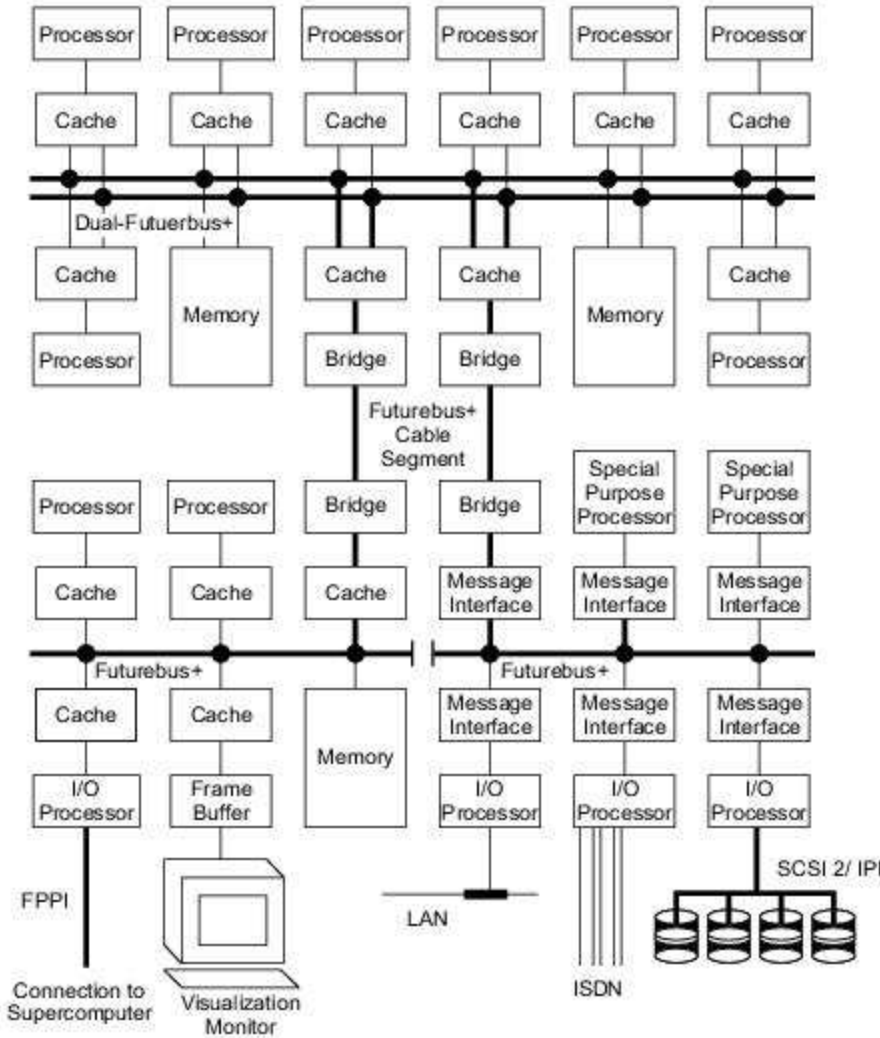


Fig. 7.5 A multiprocessor system using multiple Futurebus+ segments (Reprinted with permission from IEEE Standard 896.1-1991, copyright © 1991 by IEEE, Inc.)

7.1.2 Crossbar Switch and Multiport Memory

Switched networks provide dynamic interconnections between the inputs and outputs. Major classes of switched networks are specified below, based on the number of stages and blocking or nonblocking. We describe the crossbar networks and multiport memory structures first and then the multistage networks. Crossbar networks are mostly used in small or medium-size systems. The multistage networks can be extended to larger systems if the increased latency problem can be suitably addressed.

Network Stages Depending on the interstage connections used, a *single-stage network* is also called a *recirculating network* because data items may have to recirculate through the single stage many times before

reaching their destination. A single-stage network is cheaper to build, but multiple passes may be needed to establish certain connections. The crossbar switch and multiport memory organization are both single-stage networks.

A multistage network consists of more than one stage of switch boxes. Such a network should be able to connect from any input to any output. We will study unidirectional multistage networks in Section 7.1.3. The choice of interstage connection patterns determines the network connectivity. These patterns may be the same or different at different stages, depending the class of networks to be designed. The Omega network, Flip network, and Baseline networks are all multistage networks.

Blocking versus Nonblocking Networks A multistage network is called *blocking* if the simultaneous connections of some multiple input-output pairs may result in conflicts in the use of switches or communication links.

Examples of blocking networks include the Omega (Lawrie, 1975), Baseline (Wu and Feng, 1980), Banyan (Goke and Lipovski, 1973), and Delta networks (Patel, 1979). Some blocking networks are equivalent after graph transformations. In fact, most multistage networks are blocking in nature. In a blocking network, multiple passes through the network may be needed to achieve certain input-output connections.

A multistage network is called *nonblocking* if it can perform all possible connections between inputs and outputs by rearranging its connections. In such a network, a connection path can always be established between any input-output pair. The Benes networks (Benes, 1965) have such a capability. However, Benes networks require almost twice the number of stages to achieve the nonblocking connections. The Clos networks (Clos, 1953) can also perform all permutations in a single pass without blocking. Certain subclasses of blocking networks can also be made nonblocking if extra stages are added or connections are restricted. The blocking problem can be avoided by using combining networks to be described in the next section.

Crossbar Networks In a *crossbar network*, every input port is connected to a free output port through a crosspoint switch (circles in Fig. 2.26a) without blocking. A crossbar network is a single-stage network built with unary switches at the crosspoints.

Once the data is read from the memory, its value is returned to the requesting processor along the same crosspoint switch. In general, such a crossbar network requires the use of $n \times m$ crosspoint switches. A square crossbar ($n = m$) can implement any of the $n!$ permutations without blocking.

As introduced earlier, a crossbar switch network is a single-stage, nonblocking, permutation network. Each crosspoint in a crossbar network is a unary switch which can be set open or closed, providing a point-to-point connection path between the source and destination.

All processors can send memory requests independently and asynchronously. This poses the problem of multiple requests destined for the same memory module at the same time. In such cases, only one of the requests is serviced at a time. Let us characterize below the crosspoint switching operations.

Crosspoint Switch Design Out of n crosspoint switches in each column of an $n \times m$ crossbar mesh, only one can be connected at a time. To resolve the contention for each memory module, each crosspoint switch must be designed with extra hardware.

Furthermore, each crosspoint switch requires the use of a large number of connecting lines accommodating address, data path, and control signals. This means that each crosspoint has a complexity matching that of a bus of the same width.

For an $n \times n$ crossbar network, this implies that n^2 sets of crosspoint switches and a large number of lines are needed. What this amounts to is a crossbar network requiring extensive hardware when n is very large. So far only relatively small crossbar networks with $n \leq 16$ have been built into commercial machines.

On each row of the crossbar mesh, multiple crosspoint switches can be connected simultaneously. Simultaneous data transfers can take place in a crossbar between n pairs of processors and memories.

Figure 7.6 shows the schematic design of a row of crosspoint switches in a single crossbar network. Multiplexer modules are used to select one of n *read* or *write* requests for service. Each processor sends in an independent request, and the arbitration logic makes the selection based on certain fairness or priority rules.

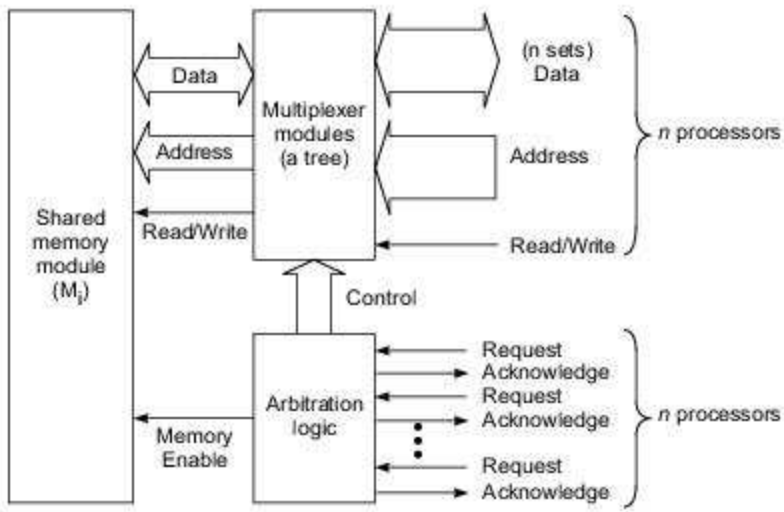


Fig. 7.6 Schematic design of a row of crosspoint switches in a crossbar network

For example, a 4-bit control signal will be generated for $n = 16$ processors. Note that n sets of data, address, and read/write lines are connected to the input of the multiplexer tree. Based on the control signal received, only one out of n sets of information lines is selected as the output of the multiplexer tree.

The memory address is entered for both *read* and *write* access. In the case of *read*, the data fetched from memory are returned to the selected processor in the reverse direction using the data path established. In the case of *write*, the data on the data path are stored in memory.

Acknowledge signals are used to indicate the arbitration result to all requesting processors. These signals initiate data transfer and are used to avoid conflicts. Note that the data path established is bidirectional, in order to serve both *read* and *write* requests for different memory cycles.

Crossbar Limitations A single processor can send many requests to multiple memory modules. For an $n \times n$ crossbar network, at most n memory words can be delivered to at most n processors in each cycle.

The crossbar network offers the highest bandwidth of n data transfers per cycle, as compared with only one data transfer per bus cycle. Since all necessary switching and conflict resolution logic are built into the crosspoint switch, the processor interface and memory port logic are much simplified and cheaper. A crossbar network is cost-effective only for small multiprocessors with a few processors accessing a few memory modules. A single-stage crossbar network is not expandable once it is built.

Redundancy or parity-check lines can be built into each crosspoint switch to enhance the fault tolerance and reliability of the crossbar network.

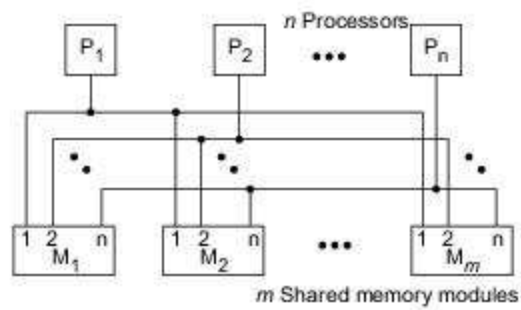
Multiport Memory Because building a crossbar network into a large system is cost prohibitive, some mainframe multiprocessors used a multiport memory organization. The idea is to move all crosspoint arbitration and switching functions associated with each memory module into the memory controller.

Thus the memory module becomes more expensive due to the added access ports and associated logic as demonstrated in Fig. 7.7a. The circles in the diagram represent n switches tied to n input ports of a memory module. Only one of n processor requests can be honored at a time.

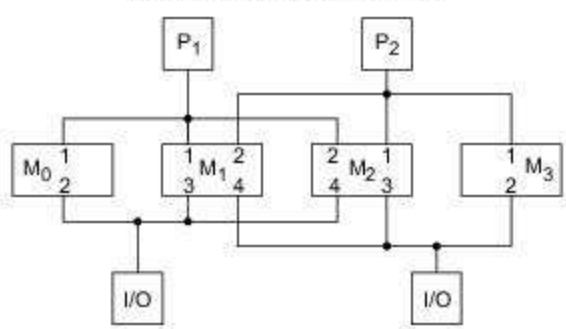
The multiport memory organization is a compromise solution between a low-cost, low-performance bus system and a high-cost, high-bandwidth crossbar system. The contention bus is time-shared by all processors and device modules attached. The multiport memory must resolve conflicts among processors.

This memory structure becomes expensive when m and n become large. A typical mainframe multiprocessor configuration may have $n = 4$ processors and $m = 16$ memory modules. A multiport memory multiprocessor is not scalable because once the ports are fixed, no more processors can be added without redesigning the memory controller.

Another drawback is the need for a large number of interconnection cables and connectors when the configuration becomes large. The ports of each memory module in Fig. 7.7b are prioritized. Some of the processors are CPUs, some are I/O processors, and some are connected to dedicated processors.



(a) n-port memory modules used



(b) Memory ports prioritized or privileged in each module by numbers

Fig. 7.7 Multiport memory organizations for multiprocessor systems (Courtesy of P. H. Enslow, *ACM Computing Surveys*, March 1977)

For example, the Univac 1100/94 multiprocessor consisted of four CPUs, four I/O processors, and two scientific vector processors connected to four shared-memory modules, each of which was 10-way ported. The access to these ports was prioritized under operating system control. In other multiprocessors, part of the memory module can be made private with ports accessible only to the owner processors.

7.1.3 Multistage and Combining Networks

Multistage networks are used to build larger multiprocessor systems. We describe two multistage networks, the Omega network and the Butterfly network, that have been built into commercial machines. We will study a special class of multistage networks, called combining networks, for resolving access conflicts automatically through the network. The combining network was built into the NYU's Ultracomputer.

Routing in Omega Network We have defined the Omega network in Chapter 2. In what follows, we describe the message-routing algorithm and broadcast capability of Omega network. This class of network was built into the Illinois Cedar multiprocessor (Kuck et al., 1987), into the IBM RP3 (Pfister et al., 1985), and into the NYU Ultracomputer (Gottlieb et al., 1983). An 8-input Omega network is shown in Fig. 7.8.

In general, an n-input Omega network has $\log_2 n$ stages. The stages are labeled from 0 to $\log_2 n - 1$ from the input end to the output end. Data routing is controlled by inspecting the destination code in binary. When the ith high-order bit of the destination code is a 0, a 2×2 switch at stage i connects the input to the upper output. Otherwise, the input is directed to the lower output.

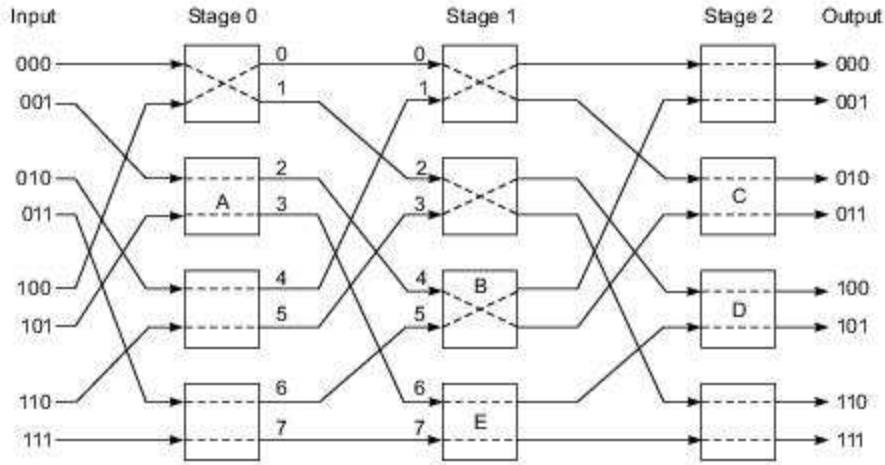
Two switch settings are shown in Figs. 7.8a and b with respect to permutations $\pi_1 = (0, 7, 6, 4, 2)\ (1, 3)\ (5)$ and $\pi_2 = (0, 6, 4, 7, 3)\ (1, 5)\ (2)$, respectively.

The switch settings in Fig. 7.8a are for the implementation of π_1, which maps $0 \rightarrow 7$, $7 \rightarrow 6$, $6 \rightarrow 4$, $4 \rightarrow 2$, $2 \rightarrow 0$, $1 \rightarrow 3$, $3 \rightarrow 1$, $5 \rightarrow 5$. Consider the routing of a message from input 001 to output 011. This involves the use of switches A, B, and C. Since the most significant bit of the destination 011 is a "zero", switch A must be set straight so that the input 001 is connected to the upper output (labeled 2). The middle bit in 011 is a "one", thus input 4 to switch B is connected to the lower output with a "crossover" connection. The least significant bit in 011 is a "one", implying a flat connection in switch C. Similarly, the switches A, E, and D are set for routing a message from input 101 to output 101. There exists no conflict in all the switch settings needed to implement the permutation π_1 in Fig. 7.8a.

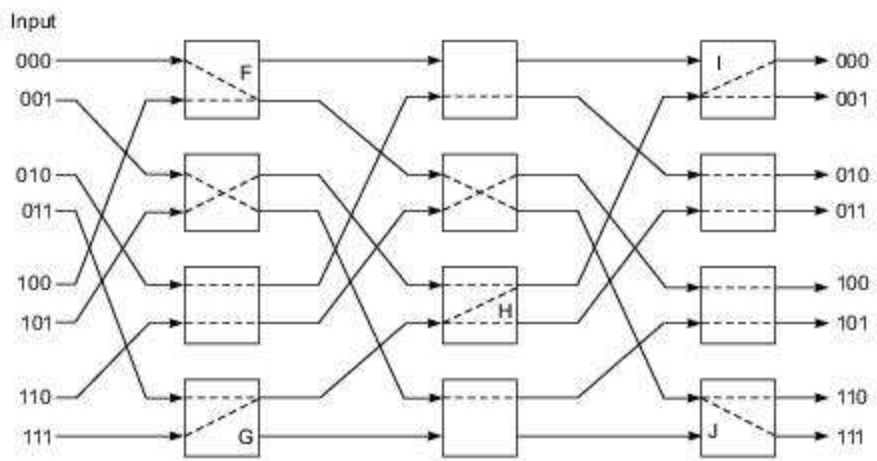
Now consider implementing the permutation π_2 in the 8-input Omega network (Fig. 7.8b). Conflicts in switch settings do exist in three switches identified as F, G, and H. The conflicts occurring at F are caused by the desired routings $000 \rightarrow 110$ and $100 \rightarrow 111$. Since both destination addresses have a leading bit 1, both inputs to switch F must be connected to the lower output. To resolve the conflicts, one request must be blocked.

Similarly, we see conflicts at switch G between $011 \rightarrow 000$ and $111 \rightarrow 011$, and at switch H between $101 \rightarrow 001$ and $011 \rightarrow 000$. At switches I and J, broadcast is used from one input to two outputs, which is allowed if the hardware is built to have four legitimate states as shown in Fig. 2.24a. The above example indicates the fact that not all permutations can be implemented in one pass through the Omega network.

The Omega network is a blocking network. In case of blocking, one can establish the conflicting connections in several passes. For the example π_2, we can connect $000 \rightarrow 110$, $001 \rightarrow 101$, $010 \rightarrow 010$, $101 \rightarrow 001$, $110 \rightarrow 100$ in the first pass and $011 \rightarrow 000$, $100 \rightarrow 111$, $111 \rightarrow 011$ in the second pass. In general, if 2×2 switch boxes are used, an n-input Omega network can implement $n^{n/2}$ permutations in a single pass. There are $n!$ permutations in total.



(a) Permutation π_1 = (0, 7, 6, 4, 2) (1, 3) (5) implemented on an Omega network without blocking



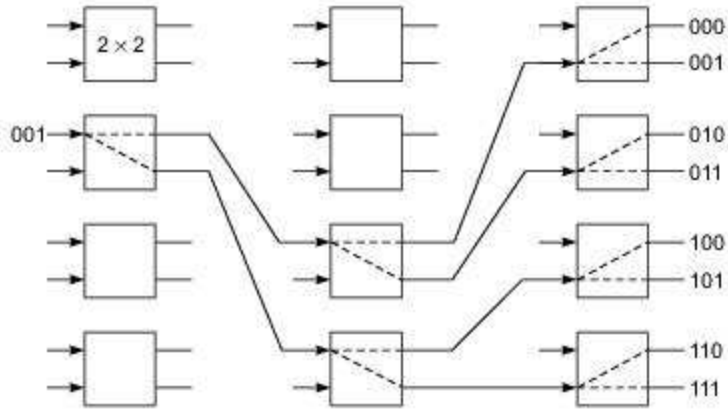
(b) Permutation π_2 = (0, 6, 4, 7, 3) (1, 5) (2) blocked at switches marked F, G, and H

Fig. 7.8 Two switch settings of an 8 × 8 Omega network built with 2 × 2 switches

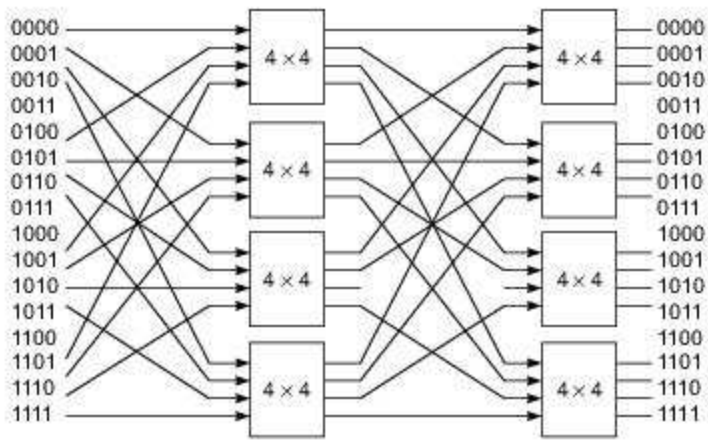
For $n = 8$, this implies that only $8^4/8! = 4096/40320 = 0.1016 = 10.16\%$ of all permutations are implementable in a single pass through an 8-input Omega network. All others will cause blocking and demand up to three passes to be realized. In general, a maximum of $\log_2 n$ passes are needed for an n-input Omega. Blocking is not a desired feature in any multistage network, since it lowers the effective bandwidth.

The Omega network can also be used to broadcast data from one source to many destinations, as exemplified in Fig. 7.9a, using the upper broadcast or lower broadcast switch settings. In Fig. 7.9a, the message at input 001 is being broadcast to all eight outputs through a binary tree connection.

The two-way shuffle interstage connections can be replaced by four-way shuffle interstage connections when 4 × 4 switch boxes are used as building blocks, as exemplified in Fig. 7.9b for a 16-input Omega network with $\log_4 16 = 2$ stages.



(a) Broadcast connections



(b) Using four-way shuffle interstage connections

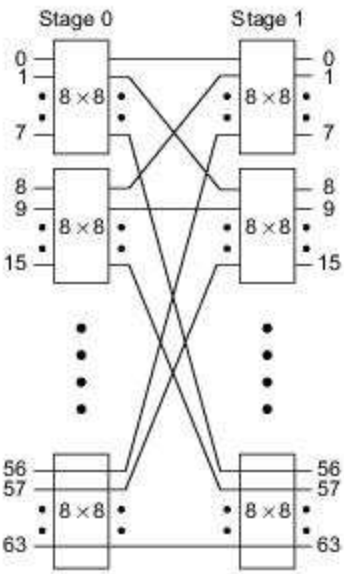
Fig. 7.9 Broadcast capability of an Omega network built with 4 × 4 switches

Note that a four-way shuffle corresponds to dividing the 16 inputs into four equal subsets and then shuffling them evenly among the four subsets. When $k \times k$ switch boxes are used, one can define a k-way shuffle function to build an even larger Omega network with $\log_k n$ stages.

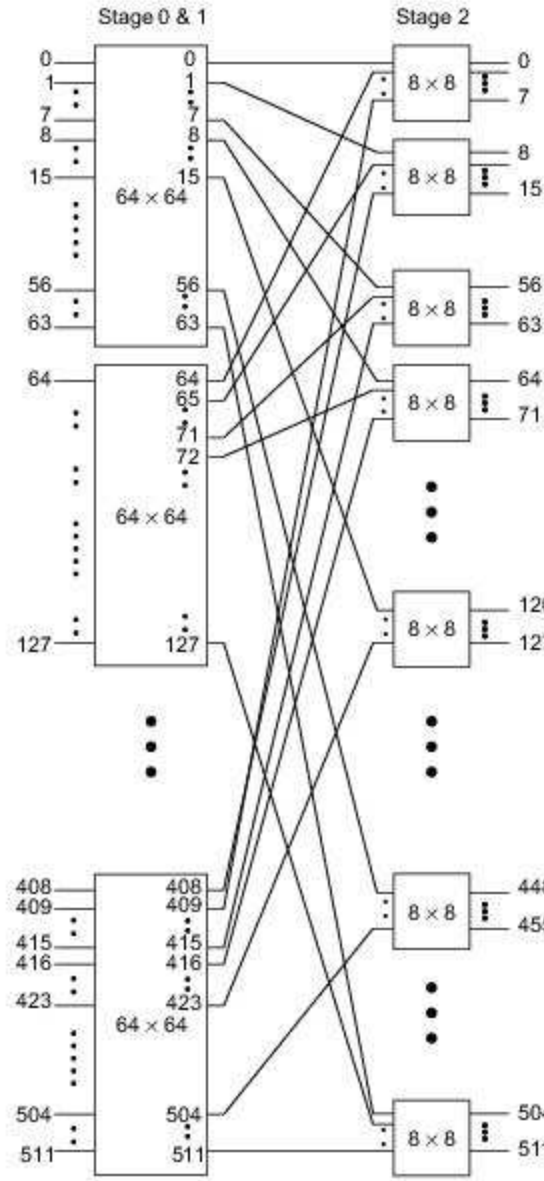
Routing in Butterfly Networks This class of networks is constructed with crossbar switches as building blocks. Figure 7.10 shows two Butterfly networks of different sizes. Figure 7.10a shows a 64-input Butterfly network built with two stages ($2 = \log_8 64$) of 8×8 crossbar switches. The eight-way shuffle function is used to establish the interstage connections between stage 0 and stage 1. In Fig. 7.10b, a three-stage Butterfly network is constructed for 512 inputs, again with 8×8 crossbar switches. Each of the 64×64 boxes in Fig. 7.10b is identical to the two-stage Butterfly network in Fig. 7.10a.

In total, sixteen 8×8 crossbar switches are used in Fig. 7.10a and $16 \times 8 + 8 \times 8 = 192$ are used in Fig. 7.10b. Larger Butterfly networks can be modularly constructed using more stages. Note that no broadcast

connections are allowed in a Butterfly network, making these networks a restricted subclass of Omega networks.



(a) A two-stage 64 × 64 Butterfly switch network built with 16 8 × 8 crossbar switches and eight-way shuffle interstage connections



(b) A three-stage 512 × 512 Butterfly switch network built with 192 8 × 8 crossbar switches

Fig. 7.10 Modular construction of Butterfly switch networks with 8 × 8 crossbar switches (Courtesy of BBN Advanced Computers, Inc., 1990)

The Hot-Spot Problem When the network traffic is nonuniform, a *hot spot* may appear corresponding to a certain memory module being excessively accessed by many processors at the same time. For example, a semaphore variable being used as a synchronization barrier may become a hot spot since it is shared by many processors.

Hot spots may degrade the network performance significantly. In the NYU Ultracomputer and the IBM RP3 multiprocessor, a combining mechanism has been added to the Omega network. The purpose was to combine multiple requests heading for the same destination at switch points where conflicts are taking place.

An atomic read-modify-write primitive Fetch&Add(x, e), has been developed to perform parallel memory updates using the combining network.

Fetch&Add This atomic memory operation is effective in implementing an N-way synchronization with a complexity independent of N. In a Fetch&Add(x, e) operation, x is an integer variable in shared memory and e is an integer increment. When a single processor executes this operation, the semantics is

$$\begin{aligned} &\text{Fetch\&Add } (x, e) \\ &\quad \{temp \leftarrow x; \\ &\quad\ x \leftarrow temp + e; \\ &\quad \text{return } temp\} \end{aligned} \tag{7.1}$$

When N processes attempt Fetch&Add(x, e) at the same memory word simultaneously, the memory is updated only once following a *serialization principle*. The sum of the N increments, $e_1 + e_2 + \ldots + e_N$, is produced in any arbitrary serialization of the N requests.

This sum is added to the memory word x, resulting in a new value $x + e_1 + e_2 + \ldots + e_N$. The values returned to the N requests are all unique, depending on the serialization order followed. The net result is similar to a sequential execution of N Fetch&Adds but is performed in one indivisible operation. Two simultaneous requests are combined in a switch as illustrated in Fig. 7.11.

One of the following operations will be performed if processor P_1 executes $\text{Ans}_1 \leftarrow$ Fetch&Add(x, e_1) and P_2 executes $\text{Ans}_2 \leftarrow$ Fetch&Add(x, e_2) simultaneously on the shared variable x. If the request from P_1 is executed ahead of that from P_2 , the following values are returned:

$$\begin{aligned} \text{Ans}_1 &\leftarrow x \\ \text{Ans}_2 &\leftarrow x + e_1 \end{aligned} \tag{7.2}$$

If the execution order is reversed, the following values are returned:

$$\begin{aligned} \text{Ans}_1 &\leftarrow x + e_2 \\ \text{Ans}_2 &\leftarrow x \end{aligned} \tag{7.3}$$

Regardless of the executing order, the value $x + e_1 + e_2$ is stored in memory. It is the responsibility of the switch box to form the sum $e_1 + e_2$, transmit the combined request Fetch&Add(x, $e_1 + e_2$), store the value e_1 (or e_2) in a wait buffer of the switch, and return the values x and $x + e_1$ to satisfy the original requests Fetch&Add(x, e_1) and Fetch&Add(x, e_2), respectively, as illustrated in Fig. 7.11 in four steps.

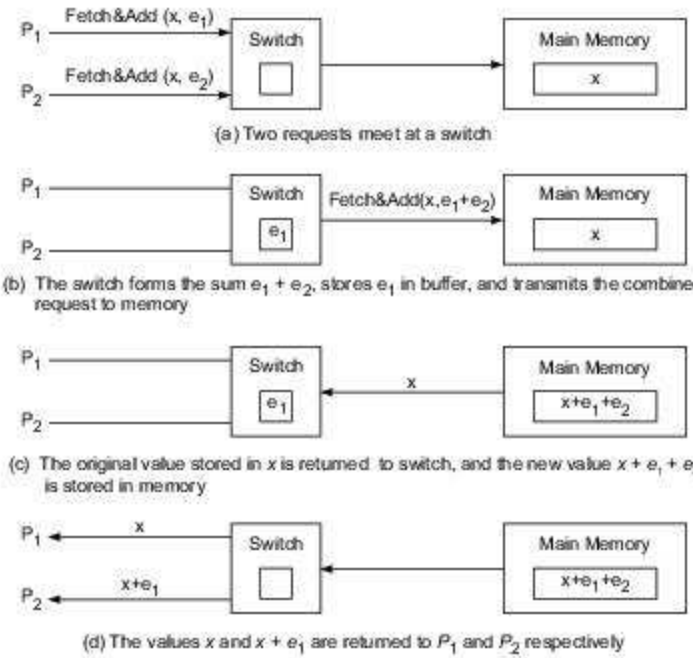


Fig. 7.11 Two Fetch&Add operations are combined to access a shared variable simultaneously via a combining network

Applications and Drawbacks The Fetch&Add primitive is very effective in accessing sequentially allocated queue structures in parallel, or in forking out parallel processes with identical code that operate on different data sets.

Consider the parallel execution of N independent iterations of the following Do loop by p processors:

Doall $N = 1$ to 100
<Code using N>
Endall

Each processor executes a Fetch&Add on N before working on a specific iteration of the loop. In this case, a unique value of N is returned to each processor, which is used in the code segment. The code for each processor is written as follows, with N being initialized as 1:

$n \leftarrow$ Fetch&Add (N, 1)
While ($n \leq 100$) **Doall**
{Code using n}
$n \leftarrow$ Fetch&Add(N, 1)
Endall

The advantage of using a combining network to implement the Fetch&Add operation is achieved at a significant increase in network cost. According to NYU Ultracomputer experience, message queueing and combining in each bidirectional 2×2 switch box increased the network cost by a factor of at least 6 or more.

Additional switch cycles are also needed to make the entire operation an atomic memory operation. This may increase the network latency significantly. Multistage combining networks have the potential of supporting large-scale multiprocessors with thousands of processors. The problem of increased cost and latency may be alleviated with the use of faster and cheaper switching technology in the future.

Multistage Networks in Real Systems The IBM RP3 was designed to include 512 processors using a high-speed Omega network for reads or writes and a combining network for synchronization using Fetch&Adds. A 128-port Omega network in the RP3 had a bandwidth of 13 Gbytes/s using a 50-MHz clock.

Multistage Omega networks were also built into the Cedar multiprocessor (Kuck et al., 1986) at the University of Illinois and in the Ultracomputer (Gottlieb et al., 1983) at New York University.

The BBN Butterfly processor (TC2000) used 8×8 crossbar switch modules to build a two-stage 64×64 Butterfly network for a 64-processor system, and a three-stage 512×512 Butterfly switch (see Fig. 7.10) for a 512-processor system in the TC2000 Series. The switch hardware was clocked at 38 MHz with a 1-byte data path. The maximum interprocessor bandwidth for a 64-processor TC2000 was designed at 2.4 Gbytes/s.

The Cray Y-MP multiprocessor used 64-, 128-, or 256-way interleaved memory banks, each of which could be accessed via four ports. Crossbar networks were used between the processors and memory banks in all Cray multiprocessors. The Alliant FX/2800 used crossbar interconnects between seven four-processor (i860) boards plus one I/0 board and eight shared, interleaved cache boards which were connected to the physical memory via a memory bus.

7.2 CACHE COHERENCE AND SYNCHRONIZATION MECHANISMS

Cache coherence protocols for coping with the multicache inconsistency problem are considered below. Snoopy protocols are designed for bus-connected systems. Directory–based protocols apply to network-connected systems. Finally, we study hardware support for fast synchronization. Software-implemented synchronization will be discussed in Chapter 11.

7.2.1 The Cache Coherence Problem

In a memory hierarchy for a multiprocessor system, data inconsistency may occur between adjacent levels or within the same level. For example, the cache and main memory may contain inconsistent copies of the same data object. Multiple caches may possess different copies of the same memory block because multiple processors operate asynchronously and independently.

Caches in a multiprocessing environment introduce the *cache coherence problem*. When multiple processors maintain locally cached copies of a unique shared-memory location, any local modification of the location can result in a globally inconsistent view of memory. Cache coherence schemes prevent this problem by maintaining a uniform state for each cached block of data. Cache inconsistencies caused by data sharing, process migration, or I/O are explained below.

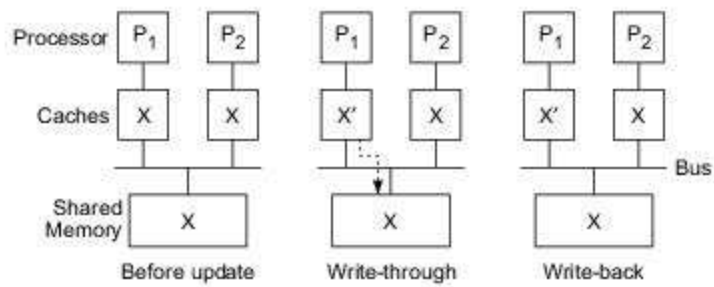
Inconsistency in Data Sharing The cache inconsistency problem occurs only when multiple private caches are used. In general, three sources of the problem are identified: *sharing of writable data, process migration*, and *I/O activity*. Figure 7.12 illustrates the problems caused by the first two sources. Consider a multiprocessor with two processors, each using a private cache and both sharing the main memory. Let X be

a shared data element which has been referenced by both processors. Before update, the three copies of X are consistent.

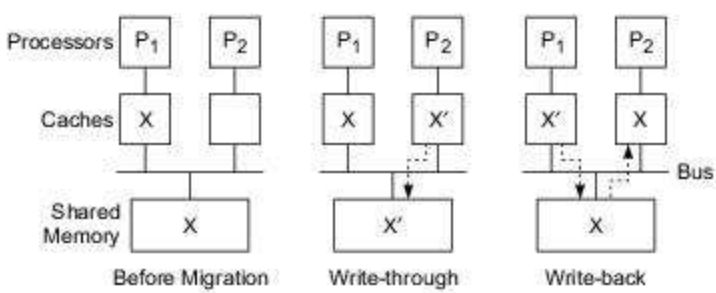
If processor P_1 writes new data X' into the cache, the same copy will be written immediately into the shared memory under a *write-through* policy. In this case, inconsistency occurs between the two copies (X' and X) in the two caches (Fig. 7.12a).

On the other hand, inconsistency may also occur when a *write-back* policy is used, as shown on the right in Fig. 7.12a. The main memory will be eventually updated when the modified data in the cache are replaced or invalidated.

Process Migration and I/O Figure 7.12b shows the occurrence of inconsistency after a process containing a shared variable X migrates from processor 1 to processor 2 using the write-back cache on the right. In the middle, a process migrates from processor 2 to processor 1 when using write-through caches.



(a) Inconsistency in sharing of writable data



(b) Inconsistency after process migration

Fig. 7.12 Cache coherence problems in data sharing and in process migration (Adapted from Dubois, Scheurich, and Briggs 1988)

In both cases, inconsistency appears between the two cache copies, labeled X and X'. Special precautions must be exercised to avoid such inconsistencies. A coherence protocol must be established before processes can safely migrate from one processor to another.

Inconsistency problems may occur during I/O operations that bypass the caches.

When the I/O processor *loads* a new data X' into the main memory, bypassing the write through caches (middle diagram in Fig. 7.13a), inconsistency occurs between cache 1 and the shared memory. When outputting a data directly from the shared memory (bypassing the caches), the write-back caches also create inconsistency.

One possible solution to the I/O inconsistency problem is to attach the 1/O processors (IOP_1 and IOP_2) to the private caches (C_1 and C_2), respectively, as shown in Fig. 7.13b. This way I/O processors share caches with the CPU. The I/O consistency can be maintained if cache-to-cache consistency is maintained via the bus. An obvious shortcoming of this scheme is the likely increase in cache perturbations and the poor locality of I/O data, which may result in higher miss ratios.

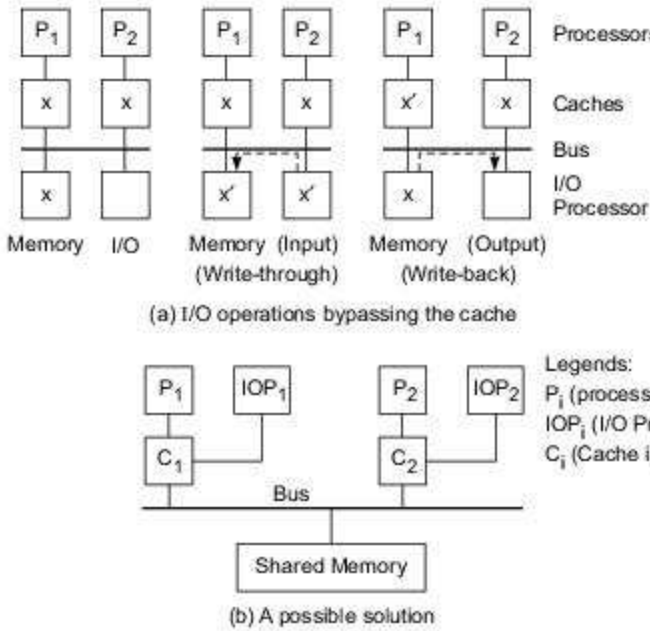


Fig. 7.13 Cache inconsistency after an I/O operation and a possible solution (Adapted from Dubois, Scheurich, and Briggs, 1988)

Two Protocol Approaches Many of the early commercially available multiprocessors used bus-based memory systems. A bus is a convenient device for ensuring cache coherence because it allows all processors in the system to observe ongoing memory transactions. If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take appropriate actions to invalidate the local copy. Protocols using this mechanism to ensure coherence are called *snoopy protocols* because each cache snoops on the transactions of other caches.

On the other hand, scalable multiprocessor systems interconnect processors using short point-to-point links in direct or multistage networks. Unlike the situation in buses, the bandwidth of these networks increases as more processors are added to the system. However, such networks do not have a convenient snooping mechanism and do not provide an efficient broadcast capability. In such systems, the cache coherence problem can be solved using some variant of directory schemes.

In general, a cache coherence protocol consists of the set of possible states in the local caches, the state in the shared memory, and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. In what follows, we first describe the snoopy protocols and then the directory-based protocols. Other approaches to designing a scalable cache coherence interface will be studied in Chapter 9.

7.2.2 Snoopy Bus Protocols

In using private caches associated with processors tied to a common bus, two approaches have been practiced for maintaining cache consistency: *write-invalidate* and *write-update* policies. Essentially, the write-invalidate policy will invalidate all remote copies when a local cache block is updated. The write-update policy will broadcast the new data block to all caches containing a copy of the block.

Snoopy protocols achieve data consistency among the caches and shared memory through a bus watching mechanism. As illustrated in Fig. 7.14, two snoopy bus protocols create different results. Consider three processors (P_1, P_2, and P_n) maintaining consistent copies of block X in their local caches (Fig. 7.14a) and in the shared-memory module marked X.

Using a *write-invalidate protocol*, the processor P_1 modifies (writes) its cache from X to X', and all other copies are invalidated via the bus (denoted I in Fig. 7.14b). Invalidated blocks are sometimes called *dirty*, meaning they should not be used. The *write-update protocol* (Fig. 7.14c) demands the new block content X' be broadcast to all cache copies via the bus. The memory copy is also updated if write-through caches are used. In using write-back caches, the memory copy is updated later at block replacement time.

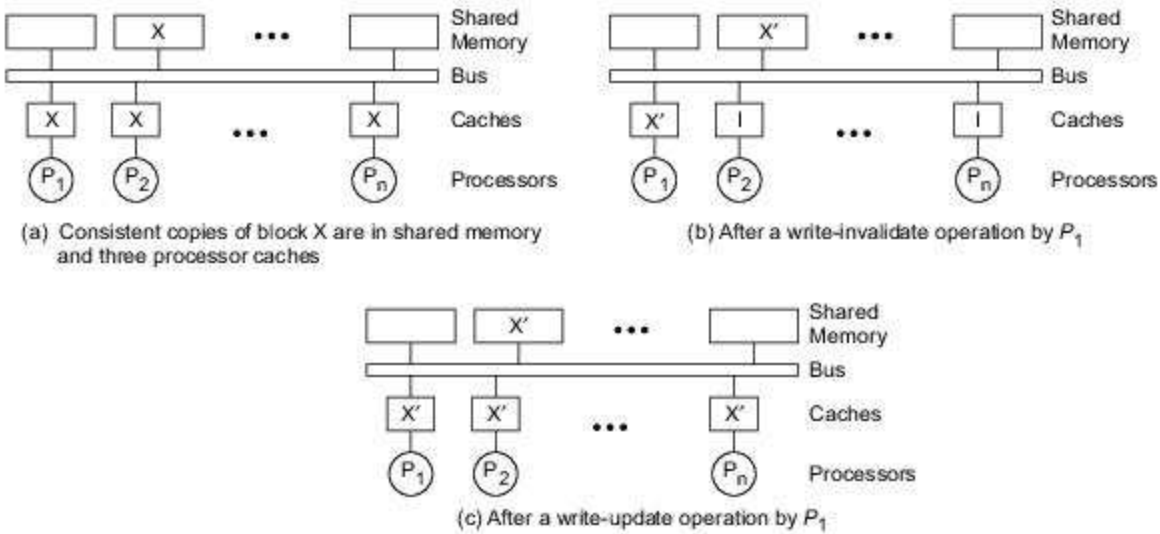


Fig. 7.14 Write-invalidate and write-update coherence protocols for write through caches (1: invalidate)

Write-Through Caches The states of a cache block copy change with respect to *read*, *write*, and *replacement* operations in the cache. Figure 7.15 shows the state transitions for two basic write-invalidate snoopy protocols developed for write-through and write-back caches, respectively. A block copy of a write-through cache i attached to processor i can assume one of two possible cache states: *valid* or *invalid* (Fig. 7.15a).

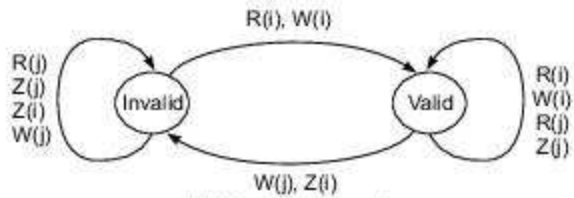
A remote processor is denoted j, where $j \neq i$. For each of the two cache states, six possible events may take place. Note that all cache copies of the same block use the same transition graph in making state changes.

In a *valid* state (Fig. 7.15a), all processors can *read* ($R(i)$, $R(j)$) safely. Local processor i can also *write* ($W(i)$) safely in a *valid* state. The *invalid* state corresponds to the case of the block either being invalidated or being replaced ($Z(i)$ or $Z(j)$).

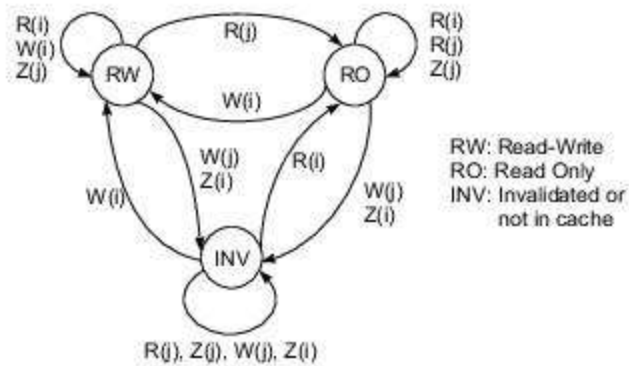
Wherever a remote processor *writes* ($W(j)$) into its cache copy, all other cache copies become invalidated. The cache block in cache i becomes valid whenever a successful read ($R(i)$) or write ($W(i)$) is carried out by a local processor i.

The fraction of *write cycles* on the bus is higher than the fraction of *read cycles* in a write-through cache, due to the need for request invalidations. The *cache directory* (registration of cache states) can be made in dual copies or dual-ported to filter out most invalidations. In case locks are cached, an atomic Test&Set must be enforced.

Write-Back Caches The *valid* state of a write-back cache can be further split into two cache states, labeled RW (*read-write*) and RO (*read-only*) as shown in Fig. 7.15b. The INV (invalidated or not-in-cache) cache state is equivalent to the *invalid* state mentioned before. This three-state coherence scheme corresponds to an *ownership protocol*.



(a) Write-through cache



$W(i)$ = Write to block by processor i.
$R(i)$ = Read block by processor i.
$Z(i)$ = Replace block in cache i.

$W(j)$ = Write to block copy in cache j by processor $j \neq i$.
$R(j)$ = Read block copy in cache j by processor $j \neq i$.
$Z(j)$ = Replace block copy in cache $j \neq i$.

(b) Write-back cache

Fig. 7.15 Two state-transition graphs for a cache block using write-invalidate snoopy protocols (Adapted from Dubois, Scheurich, and Briggs, 1988)

When the memory owns a block, caches can contain only the RO copies of the block. In other words, multiple copies may exist in the RO state and every processor having a copy (called a *keeper* of the copy) can *read* ($R(i)$, $R(j)$) the copy safely.

The INV state is entered whenever a remote processor *writes* ($W(j)$) its local copy or the local processor replaces ($Z(i)$) its own block copy. The RW state corresponds to only one cache copy existing in the entire system owned by the local processor i. *Read* ($R(i)$) and *write* ($W(i)$) can be safely performed in the RW state. From either the RO state or the INV state, the cache block becomes uniquely owned when a local *write* ($W(i)$) takes place.

Other state transitions in Fig. 7.15b can be similarly figured out. Before a block is modified, ownership for exclusive access must first be obtained by a *read-only* bus transaction which is broadcast to all caches and memory. If a modified block copy exists in a remote cache, memory must first be updated, the copy invalidated, and ownership transferred to the requesting cache.

Write-once Protocol James Goodman (1983) proposed a cache coherence protocol for bus-based multiprocessors. This scheme combines the advantages of both write-through and write-back invalidations. In order to reduce bus traffic, the very first *write* of a cache block uses a write-through policy.

This will result in a consistent memory copy while all other cache copies are invalidated. After the first *write*, shared memory is updated using a write-back policy. This scheme can be described by the four-state transition graph shown in Fig. 7.16. The four cache states are defined below:

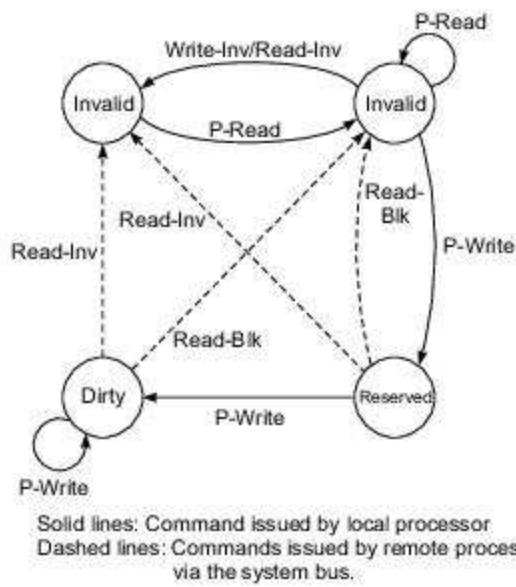


Fig. 7.16 Goodman's write-once cache coherence protocol using the write invalidate policy on write-back caches (Adapted from James Goodman 1983, reprinted from Stenstrom, *IEEE Computer*, June 1990)

- *Valid:* The cache block, which is consistent with the memory copy, has been *read* from shared memory and has not been modified.
- *Invalid:* The block is not found in the cache or is inconsistent with the memory copy.
- *Reserved:* Data has been *written* exactly *once* since being *read* from shared memory. The cache copy is consistent with the memory copy, which is the only other copy.

- *Dirty:* The cache block has been modified (*written*) more than once, and the cache copy is the only one in the system (thus inconsistent with all other copies).

To maintain consistency, the protocol requires two different sets of commands. The solid lines in Fig. 7.16 correspond to access commands issued by a local processor labeled *read-miss*, *write-hit*, and *write-miss*. Whenever a *read-miss* occurs, the *valid* state is entered.

The first *write-hit* leads to the *reserved state*. The second *write-hit* leads to the *dirty* state, and all future *write-hits* stay in the *dirty* state. Whenever a *write-miss* occurs, the cache block enters the *dirty* state.

The dashed lines correspond to invalidation commands issued by remote processors via the snoopy bus. The *read-invalidate* command reads a block and invalidates all other copies. The *write-invalidate* command invalidates all other copies of a block. The *bus–read* command corresponds to a normal memory *read* by a remote processor via the bus

Cache Events and Actions The memory-access and invalidation commands trigger the following events and actions:

- *Read-miss:* When a processor wants to read a block that is not in the cache, a *read-miss* occurs. A *bus-read* operation will be initiated. If no *dirty* copy exists, then main memory has a consistent copy and supplies a copy to the requesting cache. If a *dirty* copy does exist in a remote cache, that cache will inhibit the main memory and send a copy to the requesting cache. In all cases, the cache copy will enter the *valid* state after a read-miss.
- *Write-hit:* If the copy is in the *dirty* or *reserved* state, the *write* can be carried out locally and the new state is *dirty*. If the new state is *valid*, a *write-invalidate* command is broadcast to all caches, invalidating their copies. The shared memory is *written through*, and the resulting state is *reserved* after this first *write*.
- *Write-miss*: When a processor fails to write in a local cache, the copy must come either from the main memory or from a remote cache with a dirty block. This is accomplished by sending a *read-invalidate* command which will invalidate all cache copies. The local copy is thus updated and ends up in a *dirty* state.
- *Read-hit*: Read-hits can always be performed in a local cache without causing a state transition or using the snoopy bus for invalidation.
- *Block Replacement:* If a copy is *dirty*, it has to be written back to main memory by block replacement. If the copy is *clean* (i.e., in either the *valid*, *reserved*, or *invalid* state), no replacement will take place.

Goodman's write-once protocol demands special bus lines to inhibit the main memory when the memory copy is invalid, and a *bus-read* operation is needed after a *read miss*. Most standard buses cannot support this inhibition operation.

The IEEE Futurebus+ proposed to include this special bus provision. Using a write-through policy after the first *write* and using a write-back policy in all additional *writes* eliminates unnecessary invalidations.

Snoopy cache protocols are popular in bus-based multiprocessors because of their simplicity of implementation. The write-invalidate policies were implemented on the Sequent Symmetry multiprocessor and on the Alliant FX multiprocessor.

Besides the DEC Firefly multiprocessor, the Xerox Palo Alto Research Center implemented another write-update protocol for its Dragon multiprocessor workstation. The Dragon protocol avoids updating memory until replacement, in order to improve the efficiency of intercache transfers.

Multilevel Cache Coherence To maintain consistency among cache copies at various levels, Wilson proposed an extension to the write-invalidate protocol used on a single bus. Consistency among cache copies at the same level is maintained in the same way as described above. Consistency of caches at different levels is illustrated in Fig. 7.3.

An invalidation must propagate vertically up and down in order to invalidate all copies in the shared caches at level 2. Suppose processor P_1 issues a *write* request. The *write* request propagates up to the highest level and invalidates copies in C_{20}, C_{22}, C_{16}, and C_{18}, as shown by the arrows to all the shaded copies.

High-level caches such as C_{20} keep track of dirty blocks beneath them. A subsequent *read* request issued by P_7 will propagate up the hierarchy because no copies exist. When it reaches the top level, cache C_{20} issues a flush request down to cache C_{11} and the dirty copy is supplied to the private cache associated with processor P_7. Note that higher-level caches act as filters for consistency control. An invalidation command or a read request will not propagate down to clusters that do not contain a copy of the corresponding block. The cache C_{21} acts in this manner.

Protocol Performance Issues The performance of any snoopy protocol depends heavily on the workload patterns and implementation efficiency. The main motivation for using the snooping mechanism is to reduce bus traffic, with a secondary goal of reducing the effective memory-access time. The block size is very sensitive to cache performance in write-invalidate protocols, but not in write-update protocols.

For a uniprocessor system, bus traffic and memory-access time are mainly contributed by cache misses. The miss ratio decreases when block size increases. However, as the block size increases to a *data pollution* point, the miss ratio starts to increase. For larger caches, the data pollution point appears at a larger block size.

For a system requiring extensive process migration or synchronization, the write-invalidate protocol will perform better. However, a cache miss can result for an invalidation initiated by another processor prior to the cache access. Such *invalidation misses* may increase bus traffic and thus should be reduced.

Extensive simulation results have suggested that bus traffic in a multiprocessor may increase when the block size increases. Write-invalidate also facilitates the implementation of synchronization primitives. Typically, the average number of invalidated cache copies is rather small (one or two) in a small multiprocessor.

The write-update protocol requires a bus broadcast capability. This protocol also can avoid the ping-pong effect on data shared between multiple caches. Reducing the sharing of data will lessen bus traffic in a write-update multiprocessor. However, write–update cannot be used with long write bursts. Only through extensive program traces (trace-driven simulation) can one reveal the cache behavior, hit ratio, bus traffic, and effective memory-access time.

7.2.3 Directory-Based Protocols

A write-invalidate protocol may lead to heavy bus traffic caused by *read-misses*, resulting from the processor updating a variable and other processors trying to read the same variable. On the other hand, the write-update protocol may update data items in remote caches which will never be used by other processors. In fact, these problems pose additional limitations in using buses to build large multiprocessors.

When a multistage or packet switched network is used to build a large multiprocessor with hundreds of processors, the snoopy cache protocols must be modified to suit the network capabilities. Since broadcasting is expensive to perform in such a network, consistency commands will be sent only to those caches that keep a copy of the block. This leads to *directory-based protocols* for network-connected multiprocessors.

Directory Structures In a multistage or packet switched network, cache coherence is supported by using cache directories to store information on where copies of cache blocks reside. Various directory-based protocols differ mainly in how the directory maintains information and what information it stores.

Tang (1976) proposed the first directory scheme, which used a *central directory* containing duplicates of all cache directories. This central directory, providing all the information needed to enforce consistency, is usually very large and must be associatively searched, like the individual cache directories. Contention and long search times are two drawbacks in using a central directory for a large multiprocessor.

A distributed-directory scheme was proposed by Censier and Feautrier (1978). Each memory module maintains a separate directory which records the state and presence information for each memory block. The state information is local, but the presence information indicates which caches have a copy of the block.

In Fig. 7.17, a *read-miss* (thin lines) in cache 2 results in a request sent to the memory module. The memory controller retransmits the request to the dirty copy in cache 1. This cache *writes back* its copy. The memory module can supply a copy to the requesting cache. In the case of a *write-hit* at cache 1 (bold lines), a command is sent to the memory controller, which sends invalidations to all caches (cache 2) marked in the presence vector residing in the directory D_1.

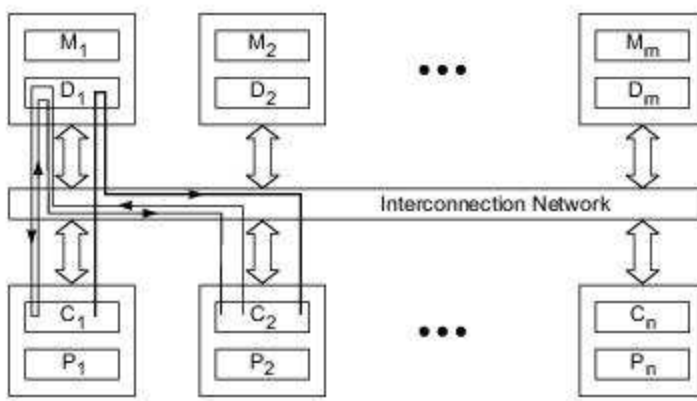


Fig. 7.17 Basic concept of a directory-based cache coherence scheme (Courtesy of Censier and Feautrier, *IEEE Trans. Computers*, Dec. 1978)

A cache-coherence protocol that does not use broadcasts must store the locations of all cached copies of each block of shared data. This list of cached locations, whether centralized or distributed, is called a *cache directory*. A directory entry for each block of data contains a number of *pointers* to specify the locations of copies of the block. Each directory entry also contains a dirty bit to specify whether a particular cache has permission to write the associated block of data.

Different types of directory protocols fall under three primary categories: *full map directories*, *limited directories*, and *chained directories*. Full-map directories store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data. That is, each directory entry contains N pointers, where N is the number of processors in the system.

Limited directories differ from full-map directories in that they have a fixed number of pointers per entry, regardless of the system size. Chained directories emulate the full-map schemes by distributing the directory

among the caches. The following descriptions of the three classes of cache directories are based on the original classification by Chaiken, Fields, Kwihara, and Agarwal (1990):

Full-Map Directories The full-map protocol implements directory entries with one bit per processor and a dirty bit. Each bit represents the status of the block in the corresponding processor's cache (present or absent). If the dirty bit is set, then one and only one processor's bit is set and that processor can write into the block.

A cache maintains two bits of state per block. One bit indicates whether a block is valid, and the other indicates whether a valid block may be written. The cache coherence protocol must keep the state bits in the memory directory and those in the cache consistent.

Figure 7.18a illustrates three different states of a full-map directory. In the first state, location X is missing in all of the caches in the system. The second state results from three caches (C1, C2, and C3) requesting copies of location X. Three pointers (processor bits) are set in the entry to indicate the caches that have copies of the block of data. In the first two states, the dirty bit on the left side of the directory entry is set to clean (C), indicating that no processor has permission to write to the block of data. The third state results from cache C3 requesting write permission for the block. In the final state, the dirty bit is set to dirty (D), and there is a single pointer to the block of data in cache C3.

Let us examine the transition from the second state to the third state in more detail. Once processor P3 issues the write to cache C3, the following events will take place:

(1) Cache C3 detects that the block containing location X is valid but that the processor does not have permission to write to the block, indicated by the block's write-permission bit in the cache.
(2) Cache C3 issues a write request to the memory module containing location X and stalls processor P3.
(3) The memory module issues invalidate requests to caches C1 and C2.
(4) Caches C1 and C2 receive the invalidate requests, set the appropriate bit to indicate that the block containing location X is invalid, and send acknowledgments back to the memory module.
(5) The memory module receives the acknowledgments, sets the dirty bit, clears the pointers to caches C1 and C2, and sends write permission to cache C3.
(6) Cache C3 receives the write permission message, updates the state in the cache, and reactivates processor P3.

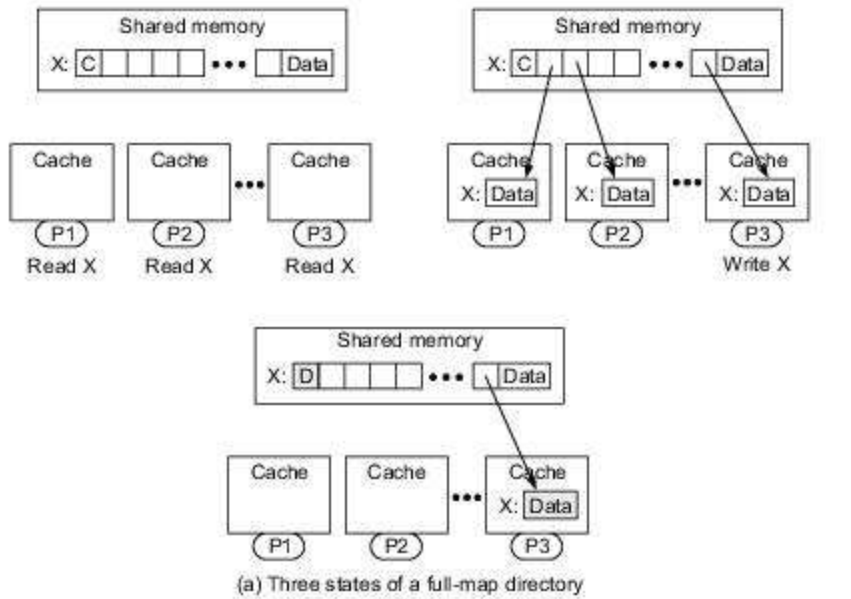
The memory module waits to receive the acknowledgments before allowing processor P3 to complete its write transaction. By waiting for acknowledgments, the protocol guarantees that the memory system ensures sequential consistency. The full-map protocol provides a useful upper bound for the performance of centralized directory-based cache coherence. However, it is not scalable due to excessive memory overhead.

Because the size of the directory entry associated with each block of memory is proportional to the number of processors, the memory consumed by the directory is proportional to the size of memory $O(N)$ multiplied by the size of the directory $O(N)$. Thus, the total memory overhead scales as the square of the number of processors $O(N^2)$.

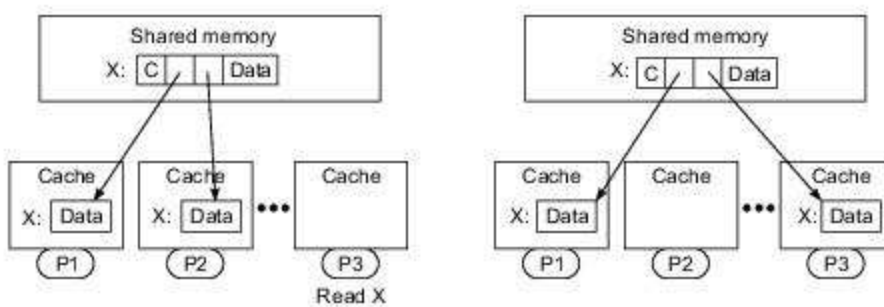
Limited Directories Limited directory protocols are designed to solve the directory size problem. Restricting the number of simultaneously cached copies of any particular block of data limits the growth of the directory to a constant factor.

A directory protocol can be classified as $Dir_i\ X$ using the notation from Agarwal et al (1988). The symbol i stands for the number of pointers, and X is NB for a scheme with no broadcast. A full-map scheme without

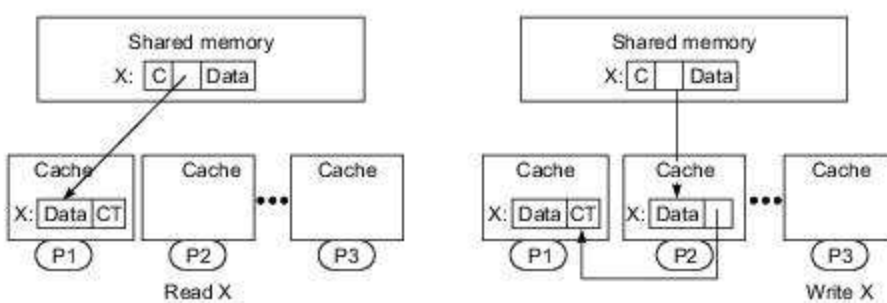
broadcast is represented as $Dir_N NB$. A limited directory protocol that uses $i < N$ pointers is denoted $Dir_i NB$. The limited directory protocol is similar to the full-map directory, except in the case when more than i caches request read copies of a particular block of data.



(a) Three states of a full-map directory



(b) Eviction in a limited directory



(c) The chained directory

Fig. 7.18 Three types of cache directory protocols (Courtesy of Chaiken et al., *IEEE Computer*, June 1990)

Figure 7.18b shows the situation when three caches request read copies in a memory system with a $Dir_2\,NB$ protocol. In this case, we can view the two-pointer directory as a two-way set-associative cache of pointers to shared copies. When cache C3 requests a copy of location X, the memory module must invalidate the copy in either cache Cl or cache C2. This process of pointer replacement is called *eviction*. Since the directory acts as a set-associative cache, it must have a pointer replacement policy.

If the multiprocessor exhibits processor locality in the sense that in any given interval of time only a small subset of all the processors access a given memory word, then a limited directory is sufficient to capture this small worker set of processors.

Directory pointers in a $Dir_i\,NB$ protocol encode binary processor identifiers, so each pointer requires $\log_2 N$ bits of memory, where N is the number of processors in the system. Given the same assumptions as for the full-map protocol, the memory overhead of limited directory schemes grows as $O(N \log_2 N)$.

These protocols are considered scalable with respect to memory overhead because the resource required to implement them grows approximately linearly with the number of processors in the system. $Dir_i\,B$ protocols allow more than i copies of each block of data to exist, but they resort to a broadcast mechanism when more than i cached copies of a block need to be invalidated. However, point-to-point interconnection networks do not provide an efficient systemwide broadcast capability. In such networks, it is difficult to determine the completion of a broadcast to ensure sequential consistency.

Chained Directories Chained directories realize the scalability of limited directories without restricting the number of shared copies of data blocks. This type of cache coherence scheme is called a *chained* scheme because it keeps track of shared copies of data by maintaining a chain of directory pointers.

The simpler of the two schemes implements a singly linked chain, which is best described by example (Fig. 7.18c). Suppose there are no shared copies of location X. If processor P1 reads location X, the memory sends a copy to cache C1, along with a *chain termination* (CT) pointer. The memory also keeps a pointer to cache C1. Subsequently, when processor P2 reads location X, the memory sends a copy to cache C2, along with the pointer to cache C1. The memory then keeps a pointer to cache C2.

By repeating the above step, all of the caches can cache a copy of the location X. If processor P3 writes to location X, it is necessary to send a data invalidation message down the chain. To ensure sequential consistency, the memory module denies processor P3 write permission until the processor with the chain termination pointer acknowledges the invalidation of the chain. Perhaps this scheme should be called a *gossip* protocol (as opposed to a snoopy protocol) because information is passed from individual to individual rather than being spread by covert observation.

The possibility of cache block replacement complicates chained-directory protocols.

Suppose that caches C1 through CN all have copies of location X and that location X and location Y map to the same (direct-mapped) cache line. If processor P_i reads location Y, it must first evict location X from its cache with the following possibilities:

(1) Send a message down the chain to cache C_{i-1} with a pointer to cache C_{i+1} and splice C_i out of the chain, or

(2) Invalidate location X in cache C_{i+1} through cache C_N.

The second scheme can be implemented by a less complex protocol than the first. In either case, sequential consistency is maintained by locking the memory location while invalidations are in progress. Another solution to the replacement problem is to use a doubly linked chain. This scheme maintains forward and backward chain pointers for each cached copy so that the protocol does not have to traverse the chain when

there is a cache replacement. The doubly linked directory optimizes the replacement condition at the cost of a larger average message block size (due to the transmission of extra directory pointers), twice the pointer memory in the caches, and a more complex coherence protocol.

Although the chained protocols are more complex than the limited directory protocols, they are still scalable in terms of the amount of memory used for the directories. The pointer sizes grow as the logarithm of the number of processors, and the number of pointers per cache or memory block is independent of the number of processors.

Cache Design Alternatives The relative merits of physical address caches and virtual address caches have to be judged based on the access time, the aliasing problem, the flushing problem, OS kernel overhead, special tagging at the process level, and cost/performance considerations. Beyond the use of private caches, three design alternatives are suggested below.

Each of the design alternatives has its own advantages and shortcomings. There exists insufficient evidence to determine whether any of the alternatives is always better or worse than the use of private caches. More research and trace data are needed to apply these cache architectures in designing high-performance multiprocessors.

Shared Caches An alternative approach to maintaining cache coherence is to completely eliminate the problem by using *shared caches* attached to shared-memory modules. No private caches are allowed in this case. This approach will reduce the main memory access time but contributes very little to reducing the overall memory-access time and to resolving access conflicts.

Shared caches can be built as second-level caches. Sometimes, one can make the second-level caches partially shared by different clusters of processors. Various cache architectures are possible if private and shared caches are both used in a memory hierarchy. The use of shared cache alone may be against the scalability of the entire system. Tradeoffs between using private caches, caches shared by multiprocessor clusters, and shared main memory are interesting topics for further research.

Noncacheable Data Another approach is not to cache shared writable data. Shared data are *noncacheable*, and only instructions or private data are *cacheable* in local caches. Shared data include locks, process queues, and any other data structures protected by critical sections.

The compiler must tag data as either *cacheable* or *noncacheable*. Special hardware tagging must be used to distinguish them. Cache systems with cacheable and noncacheable blocks demand more support from hardware and compilers.

Cache Flushing A third approach is to use *cache flushing* every time a synchronization primitive is executed. This may work well with transaction processing multiprocessor systems. Cache flushes are slow unless special hardware is used. This approach does not solve I/O and process migration problems.

Flushing can be made very selective by the compiler in order to increase efficiency. Cache flushing at synchronization, I/O, and process migration may be carried out unconditionally or selectively. Cache flushing is more often used with virtual address caches.

7.2.4 Hardware Synchronization Mechanisms

Synchronization is a special form of communication in which control information is exchanged, instead of data, between communicating processes residing in the same or different processors. Synchronization

enforces correct sequencing of processors and ensures mutually exclusive access to shared writable data. Synchronization can be implemented in software, firmware, and hardware through controlled sharing of data and control information in memory.

Multiprocessor systems use hardware mechanisms to implement low-level or primitive synchronization operations, or use software (operating system) level synchronization mechanisms such as *semaphores* or *monitors*. Only hardware synchronization mechanisms are studied below. Software approaches to synchronization will be treated in Chapter 10.

Atomic Operations Most multiprocessors are equipped with hardware mechanisms for enforcing atomic operations such as memory *read*, *write*, or *read-modify-write* operations which can be used to implement some synchronization primitives. Besides atomic memory operations, some interprocessor interrupts can be used for synchronization purposes. For example, the synchronization primitives, Test&Set (*lock*) and Reset (*lock*), are defined below:

$$\begin{aligned}&\text{Test\&Set} \quad (lock)\\ &\qquad\quad temp \leftarrow lock; \quad lock \leftarrow 1;\\ &\qquad\quad \text{return } temp\\ &\text{Reset} \quad (lock)\\ &\qquad\quad lock \leftarrow 0\end{aligned} \tag{7.4}$$

Test&Set is implemented with atomic *read-modify-write* memory operations. To synchronize concurrent processes, the software may repeat Test&Set until the returned value (*temp*) becomes 0. This synchronization primitive may tie up some bus cycles while a processor enters busy-waiting on the *spin lock*. To avoid spinning, interprocessor interrupts can be used.

A lock tied to an interrupt is called a *suspend lock*. Using such a lock, a process does not relinquish the processor while it is waiting. Whenever the process fails to open the lock, it records its status and disables all interrupts aiming at the lock. When the lock is open, it signals all waiting processors through an interrupt. A similar primitive, Compare&Swap, was implemented in IBM 370 mainframes.

Concurrent processes residing in different processors can be synchronized using *barriers*. A barrier can be implemented by a shared-memory word which keeps counting the number of processes reaching the barrier. After all processes have updated the barrier counter, the synchronization point has been reached. No processor can execute beyond the barrier until the synchronization process is complete.

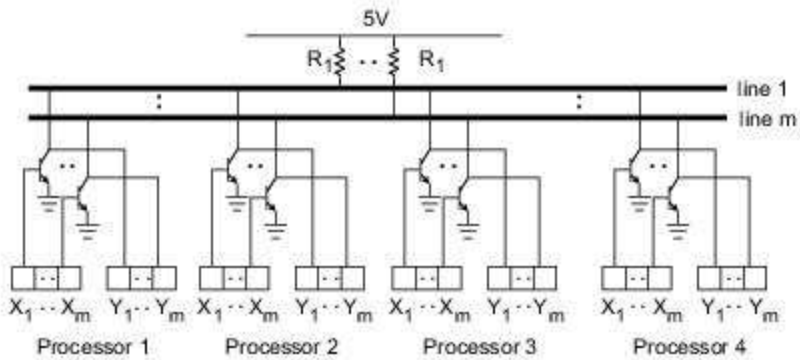
Wired Barrier Synchronization A wired-NOR logic is shown in Fig. 7.19 for implementing a barrier mechanism for fast synchronization. Each processor uses a dedicated control vector $X = (X_1, X_2, ..., X_m)$ and accesses a common monitor vector $Y = (Y_1, Y_2, ..., Y_m)$ in shared memory, where m corresponds to the barrier lines used.

The number of barrier lines needed for synchronization depends on the multiprogramming degree and the size of the multiprocessor system. Each control bit X_i is connected to the base (input) of a probing transistor. The monitor bit Y_i checks the collector voltage (output) of the transistor.

Each barrier line is wired-NOR to n transistors from n processors. Whenever bit X_i is raised to high (1), the corresponding transistor is closed, pulling down (0) the level of barrier line i. The wired-NOR connection implies that line i will be high (1) only if control bits X_i from all processors are low (0).

This demonstrates the ability to use the control bit X_i to signal the completion of a process on processor i. The bit X_i is set to 1 when a process is initiated and reset to 0 when the process finishes its execution.

When all processes finish their jobs, the X_i bits from the participating processors are all set to 0; and the barrier line is then raised to high (1), signaling the synchronization barrier has been crossed. This timing is watched by all processors through snooping on the Y_i bits. Thus only one barrier line is needed to monitor the initiation and completion of a single synchronization involving many concurrent processes.



(a) Barrier lines and interface logic

Step 1: Forking (use of one barrier line)

	Processor 1	Processor 2	Processor 3	Processor 4
Line	1			
X	1	1	1	1
Y	0	0	0	0

Step 2: Process 1 and Process 3 reach the synchronization point

	Process 1	Process 2	Process 3	Process 4
X	0	1	0	1
Y	0	0	0	0

Step 3: All processes reach the synchronization point

	Process 1	Process 2	Process 3	Process 4
X	0	0	0	0
Y	1	1	1	1

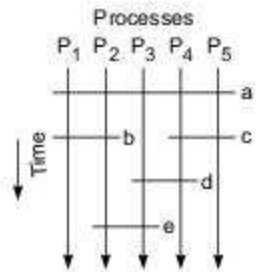
(b) Synchronization steps

Fig. 7.19 The synchronization of four independent processes on four processors using one wired-NOR barrier line (Adapted from Hwang and Shang, *Proc. Int. Conf. Parallel Processing*, 1991)

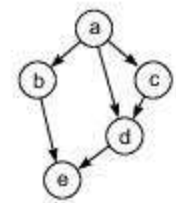
Multiple barrier lines can be used simultaneously to monitor several synchronization points. Figure 7.19 shows the synchronization of four processes residing on four processors using one barrier line. Note that other barrier lines can be used to synchronize other processes at the same time in a multiprogrammed multiprocessor environment.

Example 7.2 Wired barrier synchronization of five partially ordered processes (Hwang and Shang, 1991)

If the synchronization pattern is predicted after compile time, then one can follow the precedence graph of a partially ordered set of processes to perform multiple synchronization as demonstrated in Fig. 7.20.



(a) Synchronization patterns



(b) Precedence graph

Step 0: Initializing the control vectors (use 5 barrier lines)

	Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
X	1 1 0 0 0	1 1 0 0 1	1 0 0 1 1	1 0 1 1 0	1 0 1 0 0
Y	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
Step 1: Synchronization at barrier a					
X	0 1 0 0 0	0 1 0 0 1	0 0 0 1 1	0 0 1 1 0	0 0 1 0 0
Y	1 0 0 0 0	1 0 0 0 0	1 0 0 0 0	1 0 0 0 0	1 0 0 0 0
Step 2a: Synchronization at barrier b					
X	0 0 0 0 0	0 0 0 0 1	0 0 0 1 1	0 0 1 1 0	0 0 1 0 0
Y	1 1 0 0 0	1 1 0 0 0	1 1 0 0 0	1 1 0 0 0	1 1 0 0 0
Step 2b: Synchronization at barrier c					
X	0 0 0 0 0	0 0 0 0 1	0 0 0 1 1	0 0 0 1 0	0 0 0 0 0
Y	1 1 1 0 0	1 1 1 0 0	1 1 1 0 0	1 1 1 0 0	1 1 1 0 0
Step 3: Synchronization at barrier d					
X	0 0 0 0 0	0 0 0 0 1	0 0 0 0 1	0 0 0 0 0	0 0 0 0 0
Y	1 1 1 1 0	1 1 1 1 0	1 1 1 1 0	1 1 1 1 0	1 1 1 1 0
Step 4: Synchronization at barrier e					
X	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
Y	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1

(c) Synchronization steps

Fig. 7.20 The synchronization of five partially ordered processes using wired-NOR barrier lines (Adapted from Hwang and Shang, *Proc. Int. Conf. Parallel Processing*, 1991)

Here five processes (P_1, P_2, …, P_5) are synchronized by snooping on five barrier lines corresponding to five synchronization points labeled a, b, c, d, e. At step 0 the control vectors need to be initialized. All five processes are synchronized at point a. The crossing of barrier a is signaled by monitor bit Y_1, which is observable by all processors.

Barriers b and c can be monitored simultaneously using two lines as shown in steps $2a$ and $2b$. Only four steps are needed to complete the entire process. Note that only one copy of the monitor vector Y is maintained in the shared memory. The bus interface logic of each processor module has a copy of Y for local monitoring purposes as shown in Fig. 7.20c.

Separate control vectors are used in local processors. The above dynamic barrier synchronization is possible only if the synchronization pattern is predicted at compile time and process preemption is not allowed. One can also use the barrier wires along with counting semaphores in memory to support multiprogrammed multiprocessors in which preemption is allowed.

7.3 THREE GENERATIONS OF MULTICOMPUTERS

Three early generations of multicomputers are reviewed in this section, which have contributed to the development of modern systems. Experiences from Intel, nCUBE, MIT, and Caltech are examined. In particular, we present the Intel Paragon system in some detail. The generic multicomputer model shown in Fig. 1.9 and various network topologies presented in Section 2.3 form the background needed for reading this section. Further discussion on related topics and current advances can be found in Chapter 13.

7.3.1 Design Choices in the Past

Before we examine these developments, let us identify the major design choices made so far in building multicomputers, as compared with the development of other types of parallel computers. As illustrated in Fig. 7.21, the choices made involve the selection of processors, memory structure, interconnection schemes, and control strategy.

Design Choices In selecting a processor technology, a multicomputer designer typically chooses low-cost so-called commodity processors as building blocks. In fact, the majority of parallel computers have been built with standard off-the-shelf processors. Even the custom-designed processors used in the AMT DAP, nCUBE, TMC/CM-2, and IBM RP3 computers were low-cost processors.

The next step was to choose distributed memory for multicomputers rather than using shared memory which would limit the scalability. Each processor has its own local memory to address. Scalability becomes more feasible without shared resources. With distributed memory, a new programming model and tools are needed for multicomputers.

Multicomputers have message-passing, point-to-point, direct networks as an interconnection scheme rather than the address-switching networks used in NUMA multiprocessors like the IBM RP3 and BBN Butterfly. A message-passing network routes messages between nodes. Any node can send a message to another. Send/receive semantics must be incorporated to guarantee consistent programming with or without uniform messaging speeds.

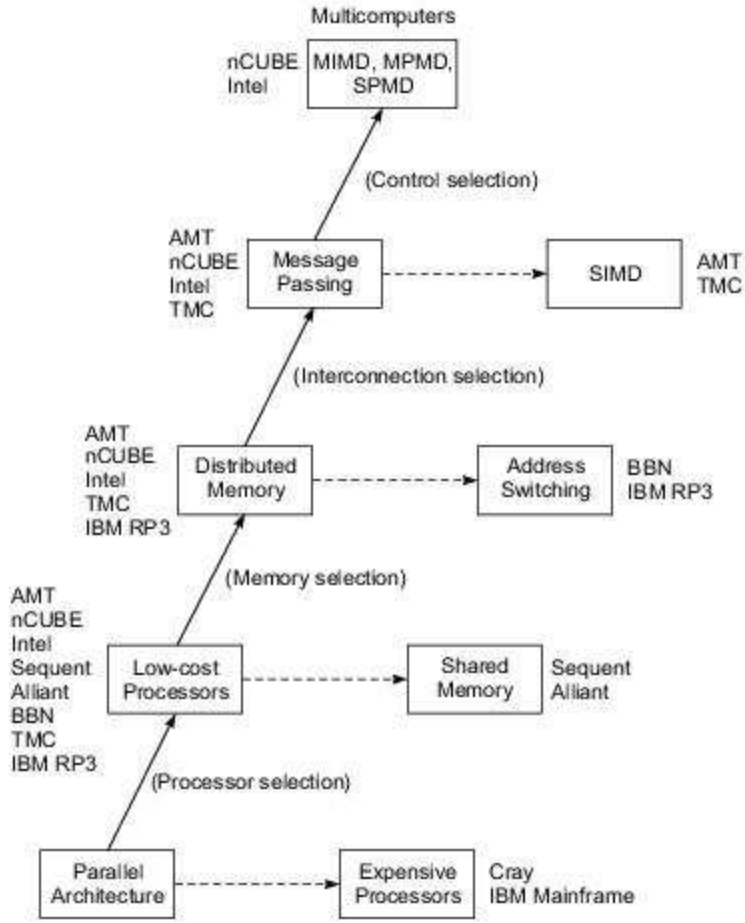


Fig. 7.21 Design choices made in the past for developing message-passing multicomputers compared to those made for other parallel computers (Courtesy of Intel Scientific Computers, 1988)

In selecting a control strategy, designers of multicomputers choose the asynchronous MIMD, MPMD, and SPMD operations, rather than the SIMD lockstep operations as in the CM-2 and DAP. Even though both support massive parallelism, the SIMD approach offers little or no opportunity to utilize existing multiprocessor code because radical changes must be made in the programming style.

On the other hand, multicomputers allow the use of existing software with minor changes from that developed for multiprocessors or for other types of parallel computers.

First Generation Caltech's Cosmic Cube (Seitz, 1983) was the first of the first generation multicomputers. The Intel iPSC/1, Ametek S/14, and nCUBE/10 were various evolutions of the original Cosmic Cube.

For example, the iPSC/1 used i80286 processors with 512 Kbytes of local memory per node. Each node was implemented on a single printed-circuit board with eight I/O ports. Seven I/O ports were used to form a seven-dimensional hypercube. The eighth port was used for an Ethernet connection from each node to the host.

Table 7.1 summarizes the important parameters used in designing the early three generations of multicomputers. The communication latency (for a 100-byte message) was rather long in the early 1980s. The 3-to-1 ratio between remote and local communication latencies was caused by the use of a *store-and-forward* routing scheme where the latency is proportional to the number of hops between two communicating nodes.

Table 7.1 *Three Early Generations of Multicomputer Development*

Generation	*First*	*Second*	*Third*
Years	*1983–87*	*1988–92*	*1993–97*
Typical node			
MIPS	1	10	100
Mflops scalar	0.1	2	40
Mflops vector	10	40	200
Memory (Mbytes)	0.5	4	32
Typical system			
N (nodes)	64	256	1024
MIPS	64	2560	100K
Mflops scalar	6.4	512	40K
Mflops vector	640	10K	200K
Memory (Mbytes)	32	1K	32K
Communication latency (100-byte message)			
Neighbor (microseconds)	2000	5	0.5
Nonlocal (microseconds)	6000	5	0.5

(Modified from Athas and Seitz, "Multicomputers: Message-Passing Concurrent Computers", *IEEE Computer*, August 1988).

Vector hardware was added on a separate board attached to each processing node board. Or one could use the second board to hold extended local memory. The host used in the iPSC/1 was an Intel 310 microprocessor. All I/O must be done through the host.

7.3.2 Present and Future Development

The second and third generations of multicomputers are introduced below. The Intel Paragon is presented as a case study. More recent advances in high-performance computing are discussed in Chapter 13.

The Second Generation A major improvement of the second generation included the use of better processors, such as i386 in the iPSC/2 and i860 in the iPSC/860 and in the Delta. The nCUBE/2 implemented 64 custom-designed VLSI processors on a single PC board. The memory per node was also increased to 10 times that of the first generation.

Most importantly, hardware-supported routing, such as *wormhole routing*, reduced the communication latency significantly from 6000 μs to less than 5 μs. In fact, the latency for remote and local communications became almost the same, independent of the number of hops between any two nodes.

The architecture of a typical second-generation multicomputer is shown in Fig. 7.22. This corresponds to a 16-node mesh-connected architecture. Mesh routing chips (MRCs) are used to establish the four-neighbor mesh network. All the mesh communication channels and MRCs are built on a backplane.

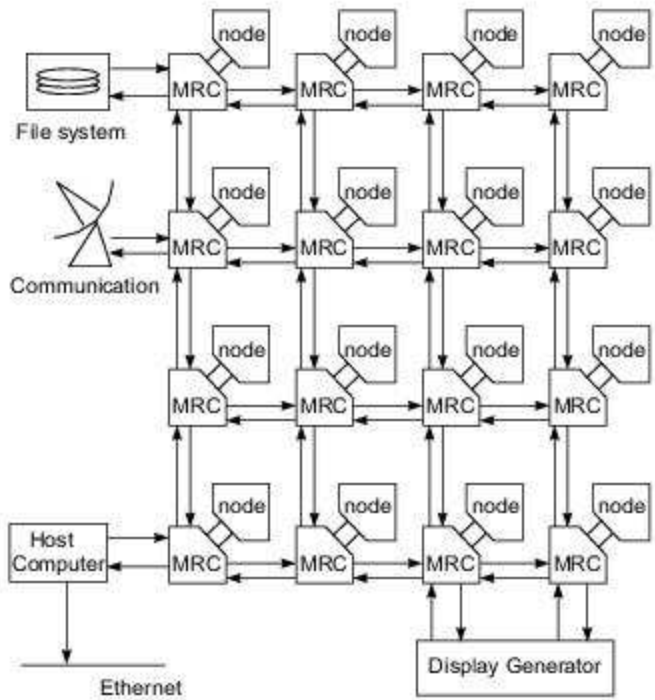


Fig. 7.22 The architecture of a second-generation multicomputer using a hardware-routed mesh interconnect (Courtesy of Charles Seitz; reprinted with permission from "Concurrent Architectures", *VLSI and Parallel Computation*, edited by Suaya and Birtwistle, Morgan Kaufmann Publishers, 1990)

Each node is implemented on a PC board plugged into the backplane at the proper MRC position. All I/O devices, graphics, and the host are connected to the periphery (boundary) of the mesh. The Intel Delta system had such a mesh architecture.

Another representative system was the nCUBE/2 which implemented a hypercube with up to 8192 nodes with a total of 512 Gbytes of distributed memory. Note that some parameters in Table 7.1 have been updated from the conservative estimates made by Atlas and Seitz in 1988. Typical figures representative of current systems can be found in Chapter 13.

The SuperNode 1000 was a Transputer-based multicomputer produced by Parsystem Ltd., England. Another second-generation system was Ametek's Series 2010, made with 25-MHz M68020 processors using a mesh-routed architecture with 225-Mbytes/s channels.

The Third Generation These designs laid the foundation for the current generation of multicomputers. Caltech had the Mosaic C project designed to use VLSI-implemented nodes, each containing a 14-MIPS processor, 20-Mbytes/s routing channels, and 16 Kbytes of RAM integrated on a single chip.

The full size of the Mosaic was targeted to have a total of 16,384 nodes organized in a three-dimensional mesh architecture. MIT built the J-machine which it planned to extend to a 65K-node multicomputer with VLSI nodes interconnected by a three-dimensional mesh network. We will study the J-machine experience in Section 9.3.2.

The J-machine planned to use message-driven processors to reduce the message handling overhead to less than 1 μs. Each processor chip would contain a 512-Kbit DRAM, a 32-bit processor, a floating-point unit, and a communication controller. The communication latency in systems was later reduced to a few ns using high-speed links and sophisticated communication protocols.

The significant reduction of overhead in communication and synchronization would permit the execution of much shorter tasks with grain sizes of 5 μs per processor in the J-machine, as opposed to executing tasks of 100 μs in the iPSc/1.This implies that concurrency may increase from 10^2 in the iPSc/1 to 10^5 in the J-machine.

The first two generations of multicomputers have been called *medium-grain systems*. With a significant reduction in communication latency, the third generation systems may be called *fine-grain multicomputers*.

Research is also underway to combine the private virtual address spaces distributed over the nodes into a globally shared virtual memory in MPP multicomputers. Instead of page-oriented message passing, the fine-grain system may require block-level cache communications. This fine-grain and shared virtual memory approach can in theory combine the relative merits of multiprocessors and multicomputers in a *heterogeneous processing* (HP) environment.

7.3.3 The Intel Paragon System

In the 1980s, hypercube multicomputers were made with homogeneous nodes because all I/O functions were given to the host. This limited the I/O bandwidth, and thus these computers could not be used in solving large-scale problems with efficiency or high throughput. The Intel Paragon was designed to overcome this difficulty. The usage model turned the multicomputer into an applications server with multiuser access in a network environment.

Ever since the introduction of the iPSC/2 CFS, parallel I/O has been possible with dedicated disk nodes in addition to the computing nodes. The iPSC/860 further pushed the idea of using heterogeneous node types. The Paragon system went further by making it a host-free multicomputer. We explain below the various node types used in the Paragon and present the hardware router design.

The architecture of the Intel Paragon system is shown in Fig. 7.23. This system was driven by applications which require solving general sparse matrix problems, performing parallel data manipulation, or making scientific predictions through simulation modeling.

These difficult problems demand heterogeneous node types for numeric, service, I/O, and network gateways, as demonstrated in the schematic diagram of the Paragon system. The mesh architecture of the Paragon was divided into three sections.

The middle section, called the compute partition, is a mesh of numeric nodes implemented with Intel i860XP microprocessors. This array had an aggregate of 8.8 Gbytes of distributed memory.

The system had a potential performance of 5 to 300 Gflops collectively. This mesh architecture eliminated the power-of-2 upgrade requirement of a hypercube architecture. All I/O was handled by the two disk I/O columns at the left and right edges of the mesh. Each column was a 16×1 array of 16 disk nodes. The aggregate I/O bandwidth reached 48 Mbytes/s with a total of 27.4 Gbytes per disk I/O column.

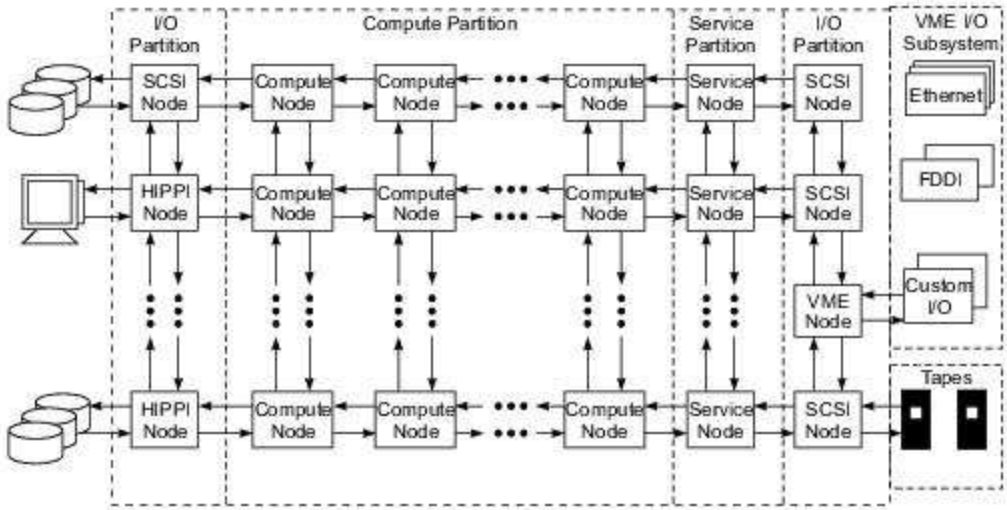


Fig. 7.23 The Intel Paragon system architecture (Courtesy of Intel Supercomputer Systems Division, 1991)

The processors used in the I/O columns were Intel i386's which supervised the massive data transfers between the disk arrays and the computational array during I/O operations. The system I/O column was made up of six *service nodes*, two tape nodes, two Ethernet nodes, and a HIPPI node. The service nodes were used for system diagnosis and handling of interrupts. The tape nodes were used for backup storage.

The Ethernet and HIPPI nodes were used for fast gateway connections with the outside world. Collectively, a 17,000-MIPS performance was claimed possible on the 570 numeric and disk I/O nodes involved in program execution. The system was designed to run iPSC/860-compatible software.

Node and Router Architecture The Paragon was designed as an experimental system. One unit was built and delivered to Caltech in May 1991 for research use by a consortium of 13 national laboratories and universities. The typical node architecture is shown in Fig. 7.24.

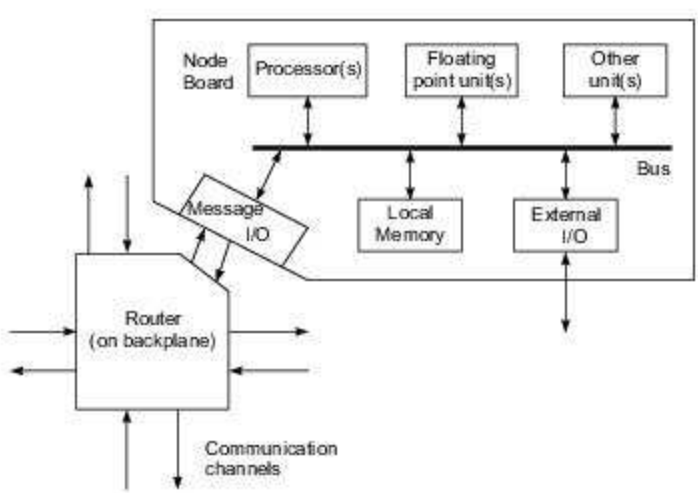


Fig. 7.24 Node architecture of the Paragon multicomputer

Each node was on a separate board. For numeric nodes, the processor and floating-point units were on the same i860 chip. The local memory took up most of the board space. The external I/O interface was implemented only on the boundary nodes with a computational array. The message I/O interface was required for message passing between local nodes and the mesh network. The *mesh-connected router* is shown in Fig. 7.25.

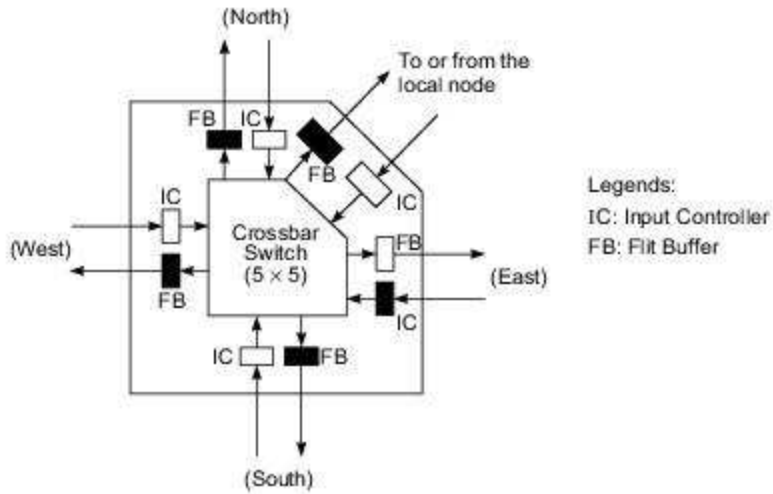


Fig. 7.25 The structure of a mesh-connected router with four pairs of I/O channels connected to neighboring routers

Each router had 10 I/O ports, 5 for input and 5 for output. Four pairs of I/O channels were used for mesh connection to the four neighbors at the north, south, east, and west nodes.

Flow control digits (flits) buffers were used at the end of input channels to hold the incoming flits. The concept of flits will be clarified in the next section. Besides four pairs of external channels, a fifth pair was used for internal connection between the router and the local node. A 5×5 crossbar switch was used to establish a connection between any input channel and any output channel.

The functions of the hardware router included pipelined message routing at the flit level and resolving buffer or channel deadlock situations to achieve deadlock-free routing. In the next section, we will explain various routing mechanisms and deadlock avoidance schemes.

All the I/O channels shown in Figs. 7.24 and 7.25 are *physical channels* which allow only one message (flit) to pass at a time. Through time-sharing, one can also implement *virtual channels* to multiplex the use of physical channels as described in the next section.

7.4 MESSAGE-PASSING MECHANISMS

Message passing in a multicomputer network demands special hardware and software support. In this section, we study the store-and-forward and wormhole routing schemes and analyze their communication latencies. We introduce the concept of virtual channels. Deadlock situations in a message-passing network are examined. We show how to avoid deadlocks using virtual channels.

Both deterministic and adaptive routing algorithms are presented for achieving deadlock-free message routing. We first study deterministic dimension-order routing schemes such as E-cube routing for hypercubes and X-Y routing for two-dimensional meshes. Then we discuss adaptive routing using virtual channels or virtual subnets. Besides one-to-one unicast routing, we will consider one-to-many multicast and one-to-all broadcast operations using virtual subnets and greedy routing algorithms.

7.4.1 Message-Routing Schemes

Message formats are introduced below. Refined formats led to the improvement from store-and-forward to wormhole routing in two generations of multicomputers. A handshaking protocol is described for asynchronous pipelining of successive routers along a communication path. Finally, latency analysis is conducted to show the time difference between the two routing schemes presented.

Message Formats Information units used in message routing are specified in Fig. 7.26. A *message* is the logical unit for internode communication. It is often assembled from an arbitrary number of fixed-length packets, thus it may have a variable length.

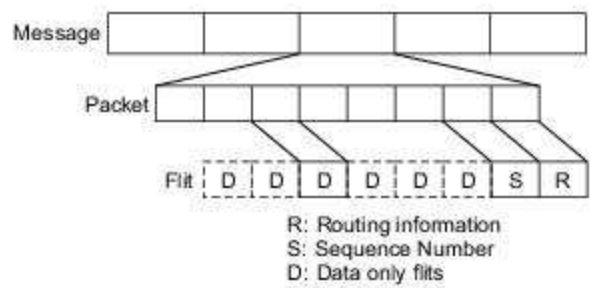


Fig. 7.26 The format of message, packets, and flits (control flow digits) used as information units of communication in a message-passing network

A *packet* is the basic unit containing the destination address for routing purposes. Because different packets may arrive at the destination asynchronously, a sequence number is needed in each packet to allow reassembly of the message transmitted.

A packet can be further divided into a number of fixed-length *flits* (flow control digits). Routing information (destination) and sequence number occupy the header flits. The remaining flits are the data elements of a packet.

In multicomputers with store-and-forward routing, packets are the smallest unit of information transmission. In wormhole-routed networks, packets are further subdivided into flits. The flit length is often affected by the network size.

The packet length is determined by the routing scheme and network implementation. Typical packet lengths range from 64 to 512 bits. The sequence number may occupy one to two flits depending on the message length. Other factors affecting the choice of packet and flit sizes include channel bandwidth, router design, network traffic intensity, etc.

Store-and-Forward Routing Packets are the basic unit of information flow in a *store-and-forward* network. The concept is illustrated in Fig. 7.27a. Each node is required to use a packet buffer. A packet is transmitted from a source node to a destination node through a sequence of intermediate nodes.

When a packet reaches an intermediate node, it is first stored in the buffer. Then it is forwarded to the next node if the desired output channel and a packet buffer in the receiving node are both available.

The latency in store-and-forward networks is directly proportional to the distance (the number of hops) between the source and the destination. This routing scheme was implemented in the first generation of multicomputers.

Wormhole Routing By subdividing the packet into smaller flits, latter generations of multicomputers implement the *wormhole routing* scheme, as illustrated in Fig. 7.27b. Flit buffers are used in the hardware routers attached to nodes. The transmission from the source node to the destination node is done through a sequence of routers.

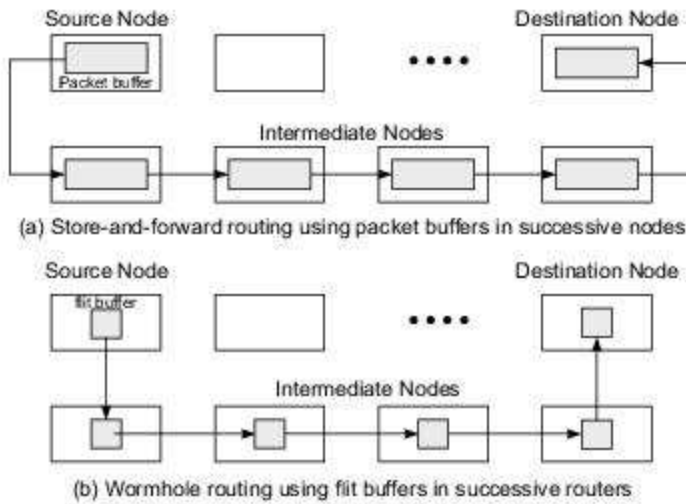


Fig. 7.27 Store-and-forward routing and wormhole routing (Courtesy of Lionel Ni, 1991)

All the flits in the same packet are transmitted in order as inseparable companions in a pipelined fashion. The packet can be visualized as a railroad train with an engine car (the header flit) towing a long sequence of box cars (data flits).

Only the header flit knows where the train (packet) is going. All the data flits (box cars) must follow the header flit. Different packets can be interleaved during transmission. However, the flits from different packets cannot be mixed up. Otherwise they may be towed to the wrong destinations.

We prove below that wormhole routing has a latency almost independent of the distance between the source and the destination.

Asynchronous Pipelining The pipelining of successive flits in a packet is done asynchronously using a handshaking protocol as shown in Fig. 7.28. Along the path, a 1-bit *ready/request* (R/A) line is used between adjacent routers.

When the receiving router (D) is ready (Fig. 7.28a) to receive a flit (i.e. the flit buffer is available), it pulls the R/A line low. When the sending router (S) is ready (Fig. 7.28b), it raises the line high and transmits flit i through the channel.

While the flit is being received by D (Fig. 7.28c), the R/A line is kept high. After flit i is removed from D's buffer (i.e. is transmitted to the next node) (Fig. 7.28d), the cycle repeats itself for the transmission of the next flit $i + 1$ until the entire packet is transmitted.

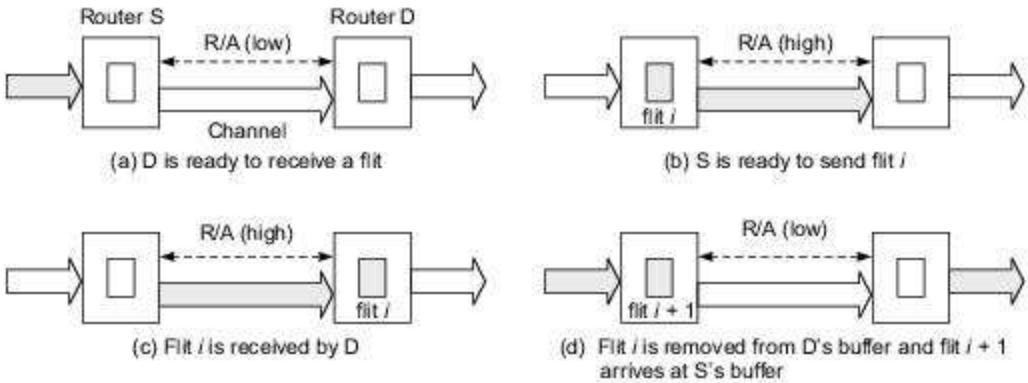


Fig. 7.28 Handshaking protocol between two wormhole routers (Courtesy of Lionel Ni, 1991)

Asynchronous pipelining can be very efficient, and the clock used can be faster than that used in a synchronous pipeline. However, the pipeline can be stalled if flit buffers or successive channels along the path are not available during certain cycles. Should that happen, the packet can be buffered, blocked, dragged, or detoured. We will discuss these flow control methods in Section 7.4.3.

Latency Analysis A time comparison between store-and-forward and wormhole-routed networks is given in Fig. 7.29. Let L be the packet length (in bits), W the channel bandwidth (in bits/s), D the distance (number of nodes traversed minus 1), and F the flit length (in bits).

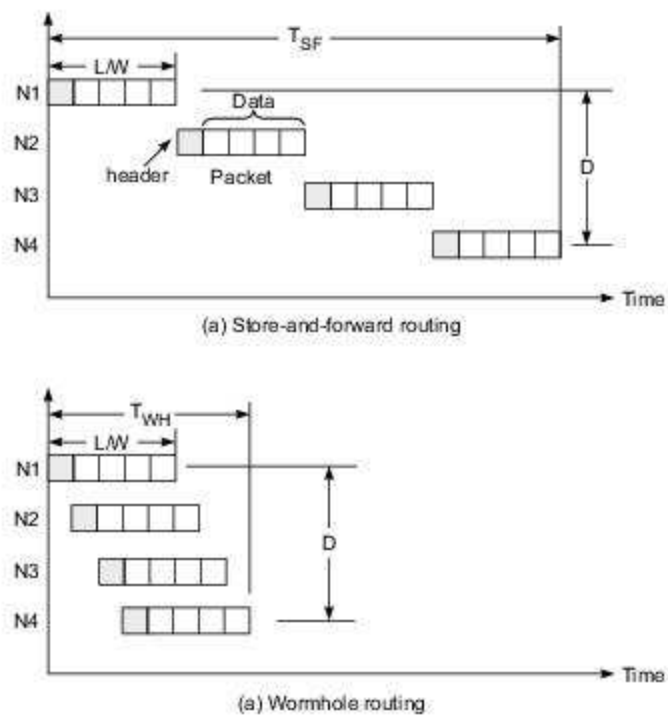


Fig. 7.29 Time comparison between the two routing techniques

The communication latency T_{SF} for a store-and-forward network is expressed by

$$T_{SF} = \frac{L}{W}(D+1) \tag{7.5}$$

The latency T_{WH} for a wormhole-routed network is expressed by

$$T_{WH} = \frac{L}{W} + \frac{F}{W} \times D \tag{7.6}$$

Equation 7.5 implies that T_{SF} is directly proportional to D. In Eq. 7.6, $T_{WH} = L/W$ if $L >> F$. Thus the distance D has a negligible effect on the routing latency.

We have ignored the network startup latency and block time due to resource shortage (such as channels being busy or buffers being full, etc.) The channel propagation delay has also been ignored because it is much smaller than the terms in T_{SF} or T_{WH}.

According to the estimate given in Table 7.1, a typical first generation value of T_{SF} is between 2000 and 6000 μs, while a typical value of T_{WH} is 5 μs or less. Current systems employ much faster processors, data links and routers. Both the latency figures above would therefore be smaller, but wormhole routing would still have much lower latency than packet store-and-forward routing.

7.4.2 Deadlock and Virtual Channels

The communication channels between nodes in a wormhole-routed multicomputer network are actually shared by many possible source and destination pairs. The sharing of a physical channel leads to the concept of virtual channels.

We introduce below the concept and explain its applications in avoiding deadlocks in this section and in facilitating network partitioning for multicasting in Section 7.4.4.

Virtual Channels A virtual channel is a logical link between two nodes. It is formed by a flit buffer in the source node, a physical channel between them, and a flit buffer in the receiver node. Figure 7.30 shows the concept of four virtual channels sharing a single physical channel.

Four flit buffers are used at the source node and receiver node, respectively. One source buffer is paired with one receiver buffer to form a virtual channel when the physical channel is allocated for the pair.

In other words, the physical channel is time-shared by all the virtual channels. Besides the buffers and channel involved, some channel states must be identified with different virtual channels. The source buffers hold flits awaiting use of the channel. The receiver buffers hold flits just transmitted over the channel. The channel (wires or fibers) provides a communication medium between them.

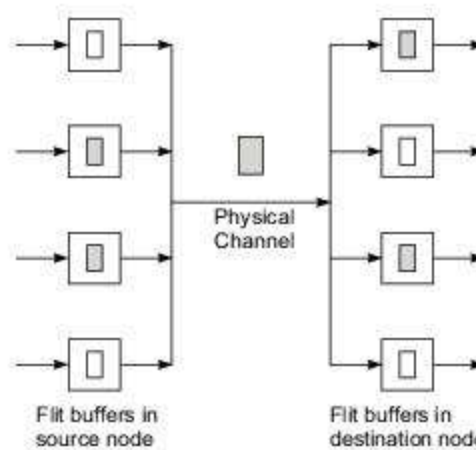
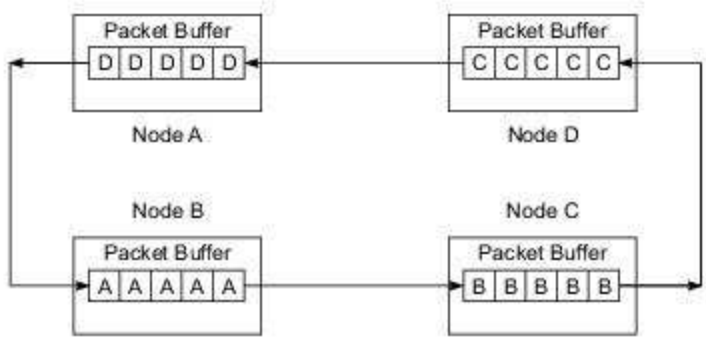


Fig. 7.30 Four virtual channels sharing a physical channel with time multiplexing on a flit-by-flit basis

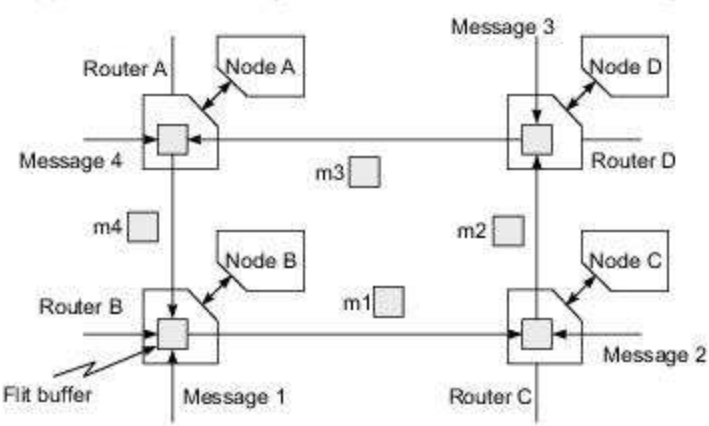
Comparing the setup in Fig. 7.30 with that in Fig. 7.28, the difference lies in the added buffers at both ends. The sharing of a physical channel by a set of virtual channels is conducted by time-multiplexing on a flit-by-flit basis.

Example 7.3 The deadlock situations caused by circular waits at buffers or at channels

As illustrated in Fig. 7.31, two types of deadlock situations are caused by a circular wait at buffers or channels. A *buffer deadlock* is shown in Fig. 7.31a for a store-and-forward network. A circular wait situation results from four packets occupying four buffers in four nodes. Unless one packet is discarded or misrouted, the deadlock cannot be broken. In Fig. 7.31b, a *channel deadlock* results from four messages being simultaneously transmitted along four channels in a mesh-connected network using wormhole routing.



(a) Buffer deadlock among four nodes with store-and-forward routing



(b) Channel deadlock among four nodes with wormhole routing; shaded boxes are flit buffers

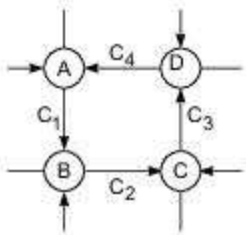
Fig. 7.31 Deadlock situations caused by a circular wait at buffers or at communication channels

Four flits from four messages occupy the four channels simultaneously. If none of the channels in the cycle is freed, the deadlock situation will continue. Circular waits are further illustrated in Fig. 7.32 using a *channel-dependence graph*.

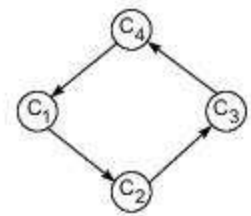
The channels involved are represented by nodes, and directed arrows are used to show the dependence relations among them. A deadlock avoidance scheme is presented using virtual channels.

Deadlock Avoidance By adding two virtual channels, V_3 and V_4 in Fig. 7.32c, one can break the deadlock cycle. A modified channel-dependence graph is obtained by using the virtual channels V_3 and V_4, after the use of channel C_2, instead of reusing C_3 and C_4.

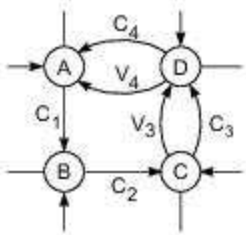
The cycle in Fig. 7.32b is being converted to a spiral, thus avoiding a deadlock. Channel multiplexing can be done at the flit level or at the packet level if the packet length is sufficiently short. Virtual channels can be implemented with either *unidirectional channels* or *bidirectional channels*.



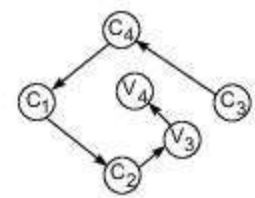
(a) Channel deadlock



(b) Channel-dependence graph containing a cycle



(c) Adding two virtual channels (V_3, V_4)



(d) A modified channel-dependence graph using the virtual channels

Fig. 7.32 Deadlock avoidance using virtual channels to convert a cycle to a spiral on a channel-dependence graph

The use of virtual channels may reduce the effective channel bandwidth available to each request. There exists a tradeoff between network throughput and communication latency in determining the degree of using virtual channels. High-speed multiplexing is required for implementing a large number of virtual channels.

7.4.3 Flow Control Strategies

In this section, we examine various strategies developed to control smooth network traffic flow without causing congestion or deadlock situations. When two or more packets collide at a node when competing for buffer or channel resources, policies must be set regarding how to resolve the conflict.

Based on these policies, we describe below deterministic and adaptive routing algorithms developed for one-to-one i.e. unicast communication.

Packet Collision Resolution In order to move a flit between adjacent nodes in a pipeline of channels, three elements must be present: (1) the source buffer holding the flit, (2) the channel being allocated, and (3) the receiver buffer accepting the flit.

When two packets reach the same node, they may request the same receiver buffer or the same outgoing channel. Two arbitration decisions must be made: (i) Which packet will be allocated the channel? and (ii) What will be done with the packet being denied the channel? These decisions lead to the four methods illustrated in Fig. 7.33 for coping with the packet collision problem.

Figure 7.33 illustrates four methods for resolving the conflict between two packets competing for the use of the same outgoing channel at an intermediate node. Packet 1 is being allocated the channel, and packet 2 being denied. A *buffering* method has been proposed with the *virtual cut-through routing* scheme devised by Kermani and Kleinrock (1979).

Packet 2 is temporarily stored in a packet buffer. When the channel becomes available later, it will be transmitted then. This buffering approach has the advantage of not wasting the resources already allocated. However, it requires the use of a large buffer to hold the entire packet.

Furthermore, the packet buffers along the communication path should not form a cycle as shown in Fig. 7.31a. The packet buffer however may cause significant storage delay. The virtual cut-through method offers a compromise by combining the store-and-forward and wormhole routing schemes. When collisions do not occur, the scheme should perform as well as wormhole routing. In the worst case, it will behave like a store-and-forward network.

Pure wormhole routing uses a blocking policy in case of packet collision, as illustrated in Fig. 7.33b. The second packet is being blocked from advancing; however, it is not being abandoned. Figure 7.33c shows the *discard* policy, which simply drops the packet being blocked from passing through.

The fourth policy is called *detour* (Fig. 7.33d). The blocked packet is routed to a detour channel. The blocking policy is economical to implement but may result in the idling of resources allocated to the blocked packet.

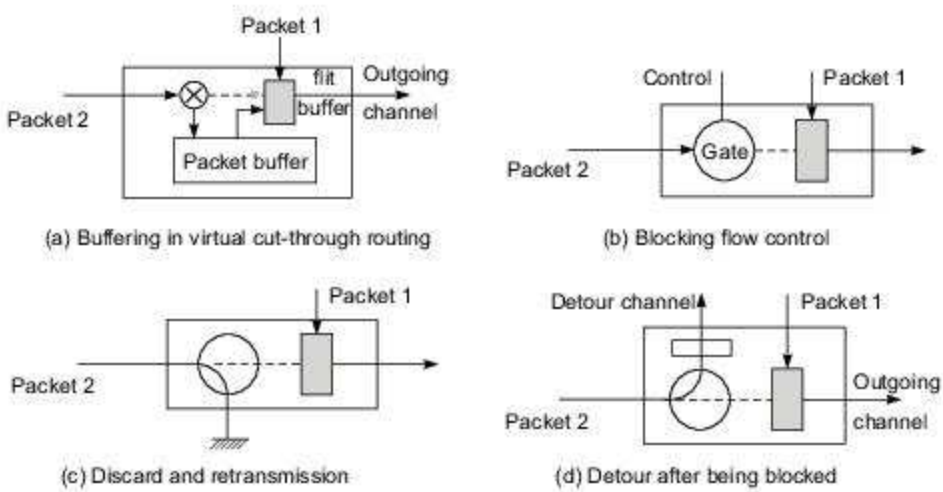


Fig. 7.33 Flow control methods for resolving a collision between two packets requesting the same outgoing channel (packet 1 being allocated the channel and packet 2 being denied)

The discard policy may result in a severe waste of resources, and it demands packet retransmission and acknowledgment. Otherwise, a packet may be lost after discarding. This policy is rarely used now because of its unstable packet delivery rate. The BBN Butterfly network had used this discard policy.

Detour routing offers more flexibility in packet routing. However, the detour may waste more channel resources than necessary to reach the destination. Furthermore, a re-routed packet may enter a cycle of *livelock*, which wastes network resources. Both the Connection Machine and the Denelcor HEP had used this detour policy.

In practice, some multicomputer networks use hybrid policies which may combine the advantages of some of the above flow control policies.

Dimension-Order Routing Packet routing can be conducted deterministically or adaptively. In *deterministic routing*, the communication path is completely determined by the source and destination addresses. In other words, the routing path is uniquely predetermined in advance, independent of network condition.

Adaptive routing may depend on network conditions, and alternate paths are possible. In both types of routing, deadlock-free algorithms are desired. Two such deterministic routing algorithms are given below, based on a concept called *dimension order routing*.

Dimension-order routing requires the selection of successive channels to follow a specific order based on the dimensions of a multidimensional network. In the case of a two-dimensional mesh network, the scheme is called *X-Y routing* because a routing path along the X-dimension is decided first before choosing a path along the Y-dimension. For hypercube (or n-cube) networks, the scheme is called *E-cube routing* as originally proposed by Sullivan and Bashkow (1977). These two routing algorithms are described below by presenting examples.

E-cube Routing on Hypercube Consider an n-cube with $N = 2^n$ nodes. Each node b is binary-coded as $b = b_{n-1}b_{n-2} \ldots b_1b_0$. Thus the source node is $s = s_{n-1} \ldots s_1s_0$ and the destination node is $d = d_{n-1} \ldots d_1d_0$. We want to determine a route from s to d with a minimum number of steps.

We denote the n dimensions as $i = 1,2, \ldots, n$, where the ith dimension corresponds to the $(i-1)$st bit in the node address. Let $v = v_{n-1} \ldots v_1v_0$ be any node along the route. The route is uniquely determined as follows:

1. Compute the direction bit $r_i = s_{i-1} \oplus d_{i-1}$ for all n dimensions ($i = 1, \ldots, n$). Start the following with dimension $i = 1$ and $v = s$.
2. Route from the current node v to the next node $v \oplus 2^{i-1}$ if $r_i = 1$. Skip this step if $r_i = 0$.
3. Move to dimension $i + 1$ (i.e. $i \leftarrow i + 1$). If $i \leq n$, go to step 2, else done.

Example 7.4 E-cube routing on a four-dimensional hypercube

The above E-cube routing algorithm is illustrated with the example in Fig. 7.34. Now $n = 4$, $s = 0110$, and $d = 1101$. Thus $r = r_4r_3r_2r_1 = 1011$. Route from s to $s \oplus 2^0 = 0111$ since $r_1 = 0 \oplus 1 = 1$. Route from $v = 0111$ to $v \oplus 2^1 = 0101$ since $r_2 = 1 \oplus 0 = 1$. Skip dimension $i = 3$ because $r_3 = 1 \oplus 1 = 0$. Route from $v = 0101$ to $v \oplus 2^3 = 1101 = d$ since $r_4 = 1$.

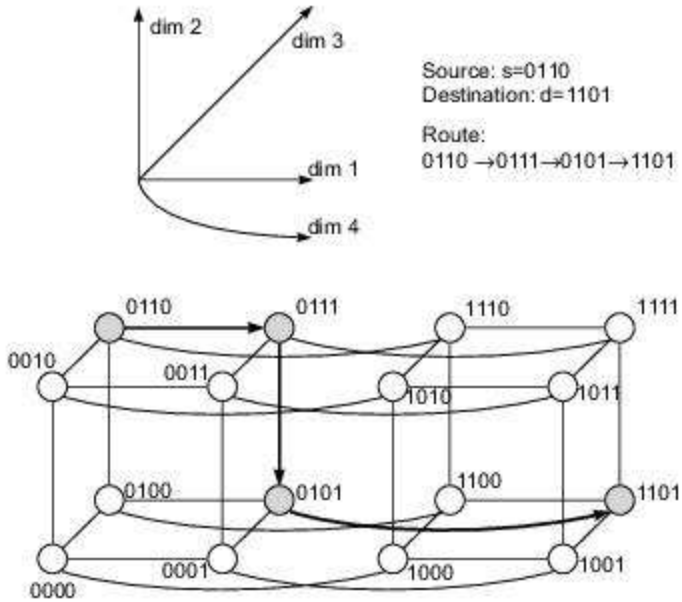


Fig. 7.34 E-cube routing on a hypercube computer with 16 nodes

The route selected is shown in Fig. 7.34 by arrows. Note that the route is determined from dimension 1 to dimension 4 in order. If the ith bit of s and d agree, no routing is needed along dimension i. Otherwise, move from the current node to the other node along the same dimension. The procedure is repeated until the destination is reached.

X-Y Routing on a 2D Mesh The same idea is applicable to mesh-connected networks. X-Y routing is illustrated by the example in Fig. 7.35. From any source node $s = (x_1y_1)$ to any destination node $d = (x_2y_2)$, route from s along the X-axis first until it reaches the column Y_2, where d is located. Then route to d along the Y-axis.

There are four possible X-Y routing patterns corresponding to the east-north, east-south, west-north, and west-south paths chosen.

Example 7.5 X-Y routing on a 2D mesh-connected multicomputer

Four (source, destination) pairs are shown in Fig. 7.35 to illustrate the four possible routing patterns on a two-dimensional mesh.

An east-north route is needed from node (2,1) to node (7,6). An east-south route is set up from node (0,7) to node (4,2). A west-south route is needed from node (5,4) to (2,0). The fourth route is west-north bound from node (6,3) to node (1,5). If the X-dimension is always routed first and then the Y-dimension, a deadlock or circular wait situation will not exist.

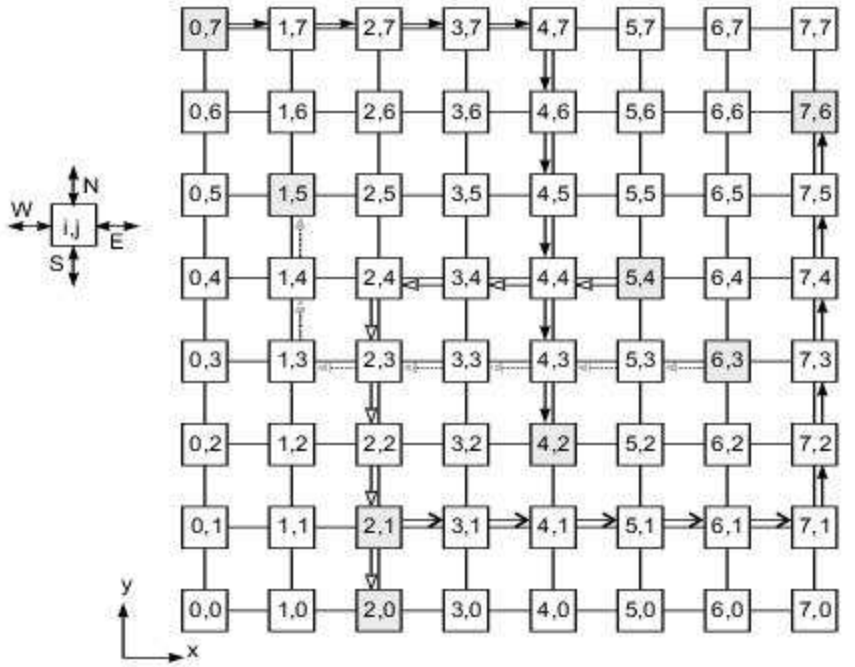
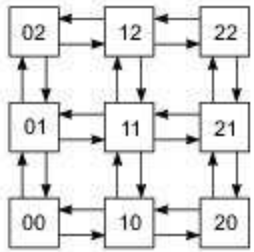


Fig. 7.35 X-Y routing on a 2D mesh computer with 8 × 8 = 64 nodes

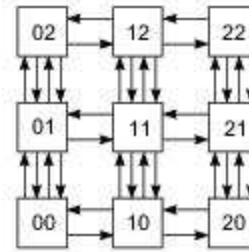
It is left as an exercise for the reader to prove that both E-cube and X-Y schemes result in deadlock-free routing. Both can be applied in either store-and-forward or wormhole-routed networks, resulting in a minimal route with the shortest distance between source and destination.

However, the same dimension order routing scheme cannot produce minimal routes for torus networks. Nonminimal routing algorithms, producing deadlock-free routes, allow packets to traverse through longer paths, sometimes to reduce network traffic or for other reasons.

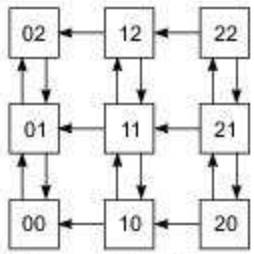
Adaptive Routing The main purpose of using adaptive routing is to achieve efficiency and avoid deadlock. The concept of virtual channels makes adaptive routing more economical and feasible to implement. We have shown in Fig. 7.32 how to apply virtual channels for this purpose. The idea can be further extended by having virtual channels in all connections along the same dimension of a mesh-connected network (Fig. 7.36).



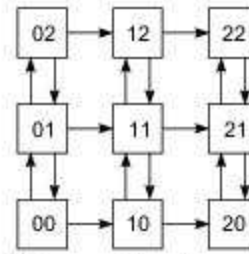
(a) Original mesh without virtual channel



(b) Two pairs of virtual channels in Y-dimension



(c) For a westbound message



(d) For an eastbound message

Fig. 7.36 Adaptive X-Y routing using virtual channels to avoid deadlock; only westbound and eastbound traffic are deadlock-free (Courtesy of Lionel Ni, 1991)

Example 7.6 Adaptive X-Y routing using virtual channels

This example uses two pairs of virtual channels in the Y-dimension of a mesh using X-Y routing.

For westbound traffic, the *virtual network* in Fig. 7.36c can be used to avoid deadlock because all eastbound X-channels are not in use. Similarly, the virtual network in Fig. 7.36d supports only eastbound traffic using a different set of virtual Y-channels.

The two virtual networks are used at different times; thus deadlock can be adaptively avoided. This concept will be further elaborated for achieving deadlockfree multicast routing in the next section.

7.4.4 Multicast Routing Algorithms

Various communication patterns are specified below. Routing efficiency is defined. The concept of virtual networks and network partitioning are applied to realize the complex communication patterns with efficiency.

Communication Patterns Four types of communication patterns may appear in multicomputer networks. What we have implemented in previous sections is the one-to-one unicast pattern with one source and one destination.

A *multicast* pattern corresponds to one-to-many communication in which one source sends the same message to multiple destinations.

A *broadcast* pattern corresponds to the case of one-to-all communication. The most generalized pattern is the many-to-many *conference* communication.

In what follows, we consider the requirements for implementing multicast, broadcast, and conference communication patterns. Of course, all patterns can be implemented with multiple unicasts sequentially, or even simultaneously if resource conflicts can be avoided. Special routing schemes must be used to implement these multi-destination patterns.

Routing Efficiency Two commonly used efficiency parameters are *channel bandwidth* and *communication latency*. The channel bandwidth at any time instant (or during any time period) indicates the effective data transmission rate achieved to deliver the messages. The latency is indicated by the packet transmission delay involved.

An optimally routed network should achieve both maximum bandwidth and minimum latency for the communication patterns involved. However, these two parameters are not totally independent. Achieving maximum bandwidth may not necessarily achieve minimum latency at the same time, and vice versa.

Depending on the switching technology used, latency is the more important issue in a store-and-forward network, while in general the bandwidth affects efficiency more in a wormhole-routed network.

Example 7.7 Multicast and broadcast on a mesh-connected computer

Multicast routing is implemented on a 3×3 mesh in Fig. 7.37. The source node is identified as S, which transmits a packet to five destinations labeled D_i for $i = 1, 2, ..., 5$.

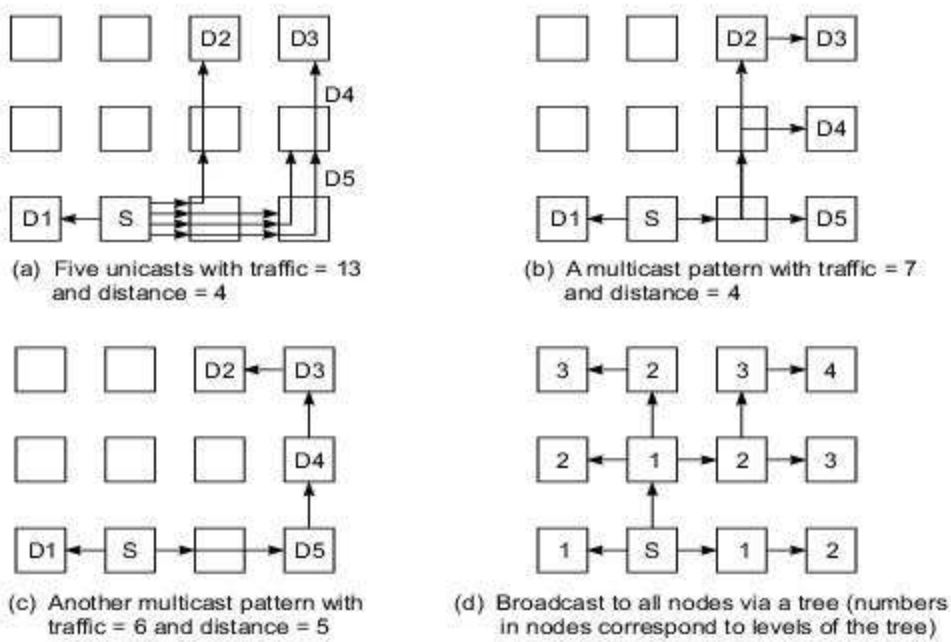


Fig. 7.37 Multiple unicasts, multicast patterns, and a broadcast tree on a 3×4 mesh computer

This five-destination multicast can be implemented by five unicasts, as shown in Fig. 7.37a. The X-Y routing traffic requires the use of 1 + 3 + 4 + 3 + 2 = 13 channels, and the latency is 4 for the longest path leading to D3.

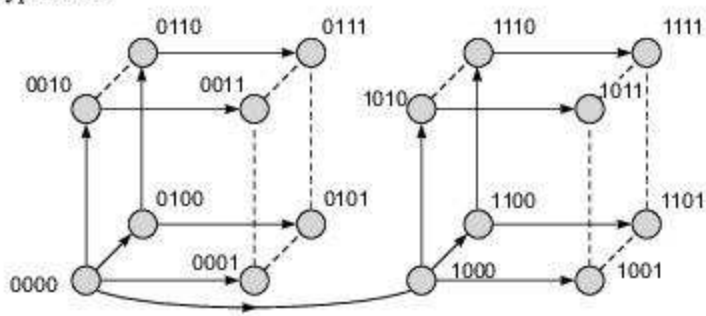
A multicast can be implemented by replicating the packet at an intermediate node, and multiple copies of the packet reach their destinations with significantly reduced channel traffic.

Two multicast routes are given in Figs. 7.37b and 7.37c, resulting in traffic of 7 and 6, respectively. On a wormhole-routed network, the multicast route in Fig. 7.37c is better. For a store-and-forward network, the route in Fig. 7.37b is better and has a shorter latency.

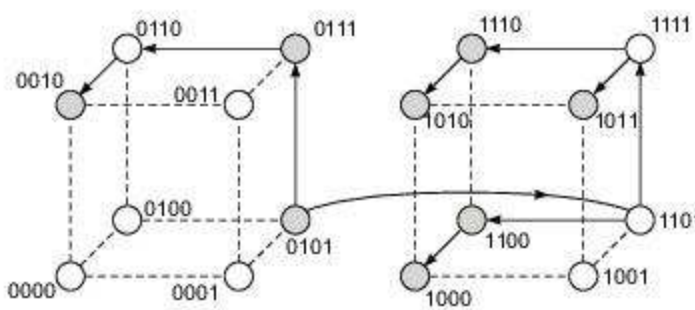
A four-level spanning tree is used from node S to broadcast a packet to all the mesh nodes in Fig. 7.37d. Nodes reached at level i of the tree have latency i. This broadcast tree should result in minimum latency as well as in minimum traffic.

Example 7.8 Multicast and broadcast on a hypercube computer

To broadcast on an n-cube, a similar spanning tree is used to reach all nodes within a latency of n. This is illustrated in Fig. 7.38a for a 4-cube rooted at node 0000. Again, minimum traffic should result with a broadcast tree for a hypercube.



(a) Broadcast tree for a 4-cube rooted at node 0000



(b) A multicast tree from node 0101 to seven destination nodes 1100, 0111, 1010, 1110, 1011, 1000, and 0010

Fig. 7.38 Broadcast tree and multicast tree on a 4-cube using a greedy algorithm (Lan, Esfahnian, and Ni, 1990)

A greedy multicast tree is shown in Fig. 7.38b for sending a packet from node 0101 to seven destination nodes. The greedy multicast algorithm is based on sending the packet through the dimension(s) which can reach the most number of remaining destinations.

Starting from the source node $S = 0101$, there are two destinations via dimension 2 and five destinations via dimension 4. Therefore, the first-level channels used are $0101 \rightarrow 0111$ and $0101 \rightarrow 1101$.

From node 1101, there are three destinations reachable in dimension 2 and four destinations via dimension 1. Thus the second-level channels used include $1101 \rightarrow 1111$, $1101 \rightarrow 1100$, and $0111 \rightarrow 0110$.

Similarly, the remaining destinations can be reached with third-level channels $1111 \rightarrow 1110$, $1111 \rightarrow 1011$, $1100 \rightarrow 1000$, and $0110 \rightarrow 0010$, and fourth-level channel $1110 \rightarrow 1010$.

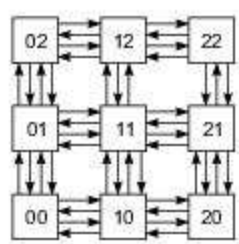
Extending the multicast tree, one should compare the reachability via all dimensions before selecting certain dimensions to obtain a minimum cover set for the remaining nodes. In case of a tie between two dimensions, selecting any one of them is sufficient. Therefore, the tree may not be uniquely generated.

It has been proved that this greedy multicast algorithm requires the least number of traffic channels compared with multiple unicasts or a broadcast tree. To implement multicast operations on wormhole-routed networks, the router in each node should be able to replicate the data in the flit buffer.

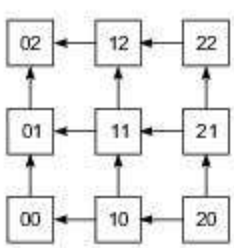
In order to synchronize the growth of a multicast tree or a broadcast tree, all outgoing channels at the same level of the tree must be ready before transmission can be pushed one level down. Otherwise, additional buffering is needed at intermediate nodes.

Virtual Networks Consider a mesh with dual virtual channels along both dimensions as shown in Fig. 7.39a.

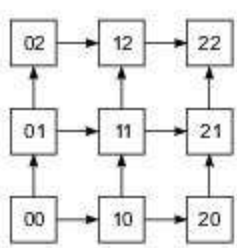
These virtual channels can be used to generate four possible virtual networks. For west-north traffic, the virtual network in Fig. 7.39b should be used.



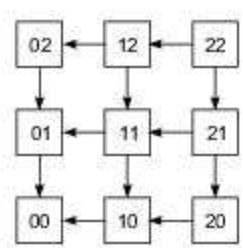
(a) A dual-channel 3 × 3 mesh



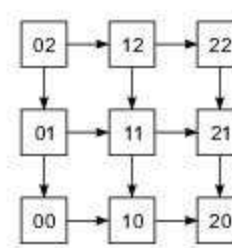
(b) West-north subnet



(c) East-north subnet



(d) West-south subnet



(e) East-south subnet

Fig. 7.39 Four virtual networks implementable from a dual-channel mesh

Similarly, one can construct three other virtual nets for other traffic orientations. Note that no cycle is possible on any of the virtual networks. Thus deadlock can be completely avoided when X-Y routing is implemented on these networks.

If both pairs between adjacent nodes are physical channels, then any two of the four virtual networks can be simultaneously used without conflict. If only one pair of physical channels is shared by the dual virtual channels between adjacent nodes, then only (b) and (e) or (c) and (d) can be used simultaneously.

Other combinations, such as (b) and (c), or (b) and (d), or (c) and (e), or (d) and (e), cannot coexist at the same time due to a shortage of channels.

Obviously, adding channels to the network will increase the adaptivity in making routing decisions. However, the increased cost can be appreciable and thus prevent the use of redundancy.

Network Partitioning The concept of virtual networks leads to the partitioning of a given physical network into logical subnetworks for multicast communications. The idea is illustrated in Fig. 7.40.

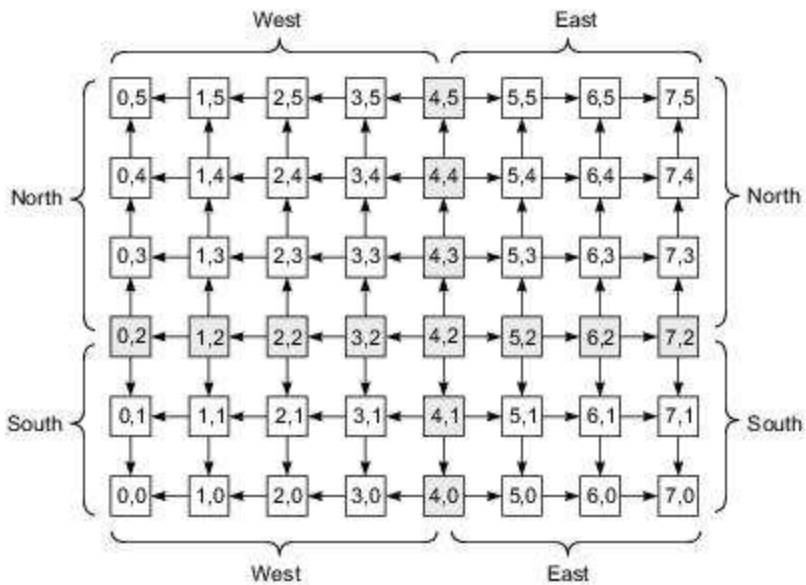


Fig. 7.40 Partitioning of a 6×8 mesh into four subnets for a multicast from source node (4,2). Shaded nodes are along the boundary of adjacent subnets (Courtesy of Lin, McKinly, and Ni, 1991)

Suppose source node (4, 2) wants to transmit to a subset of nodes in the 6×8 mesh. The mesh is partitioned into four logical subnets. All traffic heading for east and north uses the subnet at the upper right corner. Similarly, one constructs three other subnets at the remaining corners of the mesh.

Nodes in the fifth column and third row are along the boundary between subnets. Essentially, the traffic is being directed outward from the center node (4, 2). There is no deadlock if an X-Y multicast is performed in this partitioned mesh.

Similarly, one can partition a binary n-cube into 2^{n-1} subcubes to provide deadlock-free adaptive routing. Each subcube has $n + 1$ levels with 2^n virtual channels per level for the bidirectional network. The number

of required virtual channels increases rapidly with n. It has been shown that for low-dimensional cubes ($n = 2$ to 4), this method is best for general-purpose routing.

Summary

In a multiprocessor system, interconnects between sub-systems such as processors, memories and network controllers play a crucial role in determining system performance. The earliest multiprocessor systems were bus-based, with shared main memory. The bus is a simple interconnect, but it has limitations in scalability. Hierarchical bus systems can address the problem to a limited extent, but as systems grow larger, more sophisticated and scalable system interconnects are needed.

A network may be of blocking or non-blocking type. We studied the crossbar network and the basic design of a row of crosspoint switches, with its arbitration and multiplexer modules. While it has better aggregate bandwidth than the bus, the crossbar network also has limitations of scalability. Multi-port memory can be used to enhance the aggregate bandwidth of a memory module.

We studied Omega and Butterfly multistage networks. Larger Omega networks can be built using 2x2 and 4x4 basic switches, while the Butterfly network is built from modules of crossbar switches. When network traffic is non-uniform, so-called 'hot-spots' may develop which may degrade network performance. The concept of combining networks was developed in an attempt to address this performance limitation.

We studied the related issues of maintaining cache coherence and synchronization. Write operations on shared cache data, process migration and I/O operations can cause loss of cache coherence. If all the caches are on a common bus, then the snoopy bus protocol can be used to maintain cache coherence. Directory-based cache coherence protocols—using full map, limited or chained directories—can be used on more general types of system interconnects. Details of the schemes vary between write-back and write-through types of cache.

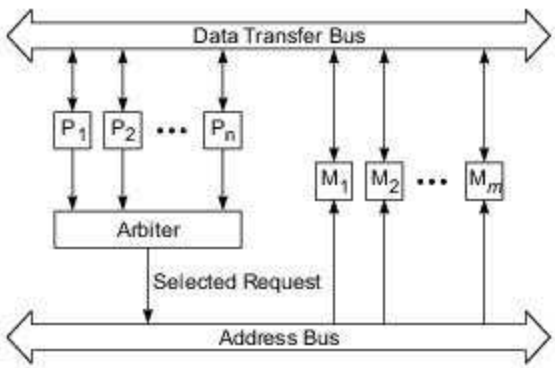
Hardware synchronization mechanisms between processors make use of atomic operations typified by Test&Set. However, at a still lower level of hardware, in theory wired barrier synchronization can also be used, of which we saw examples.

Three early generations of multicomputer systems were studied, providing a picture of how multicomputer architecture has evolved over time. Broadly, the trend has been from expensive to low cost processors, from shared to distributed memory, and (with higher speed processors) to higher speed interconnects. We studied the Intel Paragon system as a specific example, laying the basis to review more recent advances in Chapter 13.

Message-passing communication uses networks of point-to-point links, the basic aim of routing protocols being to achieve low network latency and high bandwidth. We studied the typical formats of messages, packets, and flits (flow control digits); routing schemes were studied from the points of view of latency analysis and the avoidance of deadlocks. We examined the important concepts of virtual channels, wormhole routing, flow control, collision resolution, dimension order routing, and multicast communication.

Exercises

Problem 7.1 Consider a multiprocessor with n processors and m shared-memory modules, all connected to the same backplane bus with a central arbiter as depicted below:



Assume $m > n$ and all memory modules are equally accessible to each processor. In other words, each processor generates a request for any module with probability $1/m$. The address bus and the DTB can be used at the same time to serve different requests. Both buses take one cycle to pass the address of a request or to transfer one word of 4 bytes between memory and processor. At each bus cycle (τ), the arbiter randomly selects one of the requests from the processors.

Once a memory module is identified at the end of the address cycle (one bus cycle), it takes a memory cycle (which equals c bus cycles) to retrieve the addressed word from the memory module, and another bus cycle to transfer the word to the requesting processor via the data transfer bus.

Until a memory cycle is completed, the arbiter will not issue another request to the same module. All rejected requests are ignored and resubmitted in subsequent bus cycles until being selected.

(a) Calculate the memory bandwidth defined as the average number of memory words transferred per second over the DTB if $n = 8$, $m = 16$, $\tau = 10$ ns, and $c \cdot \tau = 8\tau = 80$ ns.

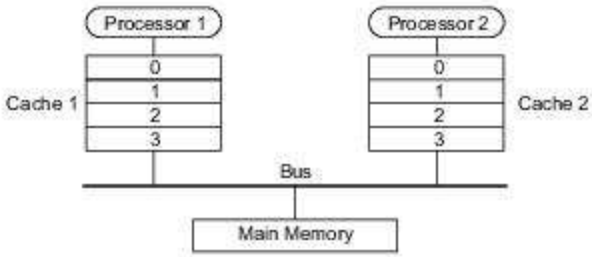
(b) Calculate the memory utilization defined as the average number of requests accepted by all memory modules per memory cycle using the same set of parameters used in part (a).

Problem 7.2 Use two-input AND and OR gates (no wired-OR) to construct an $n \times n$ crossbar switch network between n processors and n memory modules. Let the width of each crosspoint be w bits (or a word) in each direction.

(a) Prepare a schematic design of a typical crosspoint switch using c_{ij} as the enable signal for the switch in the ith row and jth column. Estimate the total number of AND and OR gates needed as a function of n and w.

(b) Assume that processor P_i has higher priority over processor P_j if $i < j$ when they are competing for access to the same memory module. Let $k = \log_2 n$ be the address width. Design an arbiter which generates all the crosspoint enable signals c_{ij}, again using only two-input AND and OR gates and some inverters if needed. The memory address decoder is assumed available from each processor and thus is not included in the arbiter design. Indicate the complexity of the arbiter design as a function of n and k.

Problem 7.3 Consider a dual-processor (P1 and P2) system using write-back private caches and a shared memory, all connected to a common contention bus. Each cache has four block frames labeled below as 0, 1, 2, 3.



The shared memory is divided into eight cache blocks as 0, 1, …, 7. To maintain cache coherence, the system uses a three-state (RO, RW, and invalid) snoopy protocol based on the write-invalidate policy described in Fig. 7.12b.

Assume the same clock drives the processors and the memory bus. Within each cycle, any processor can submit a request to access the bus. In case of simultaneous bus requests from both processors, the request from P1 is granted and P2 must wait one or more cycles to access the bus.

In all cases, the bus allows only one transaction per cycle. Once a bus access is granted, the transaction must be completed before the next request is granted. When there is no bus contention, memory-access events from each processor may require one to two cycles to complete, as specified below separately:

- Read-hit in cache requires one cycle and no bus request at all.
- Read-miss in cache requires two cycles without contention: one for block fetch and one for CPU read from cache.
- Write-hit requires one cycle for CPU write and bus invalidation simultaneously.
- Write-miss requires two cycles: one for block fetch and bus invalidation, and one for CPU write.
- Replacement of a dirty block requires one cycle to update memory via the bus.

(a) In the case of bus contention, one additional cycle is needed for bus arbitration in all the above cases except a read-hit.

(i) Show how to map the eight cache blocks to four cache block frames using a direct-mapping cache organization.

(ii) Show how to map the eight cache block frames using a two-way set-associative cache organization.

(b) Consider the following two asynchronous sequences of memory-access events, where boldface numbers are for write and the remaining are for read.

Processor #1 : 0,**0**,0,**1**,1,4,3,3,5,**5**,5
Processor #2 : 2,**2**,0,0,**7**,5,5,5,**7**,7,0

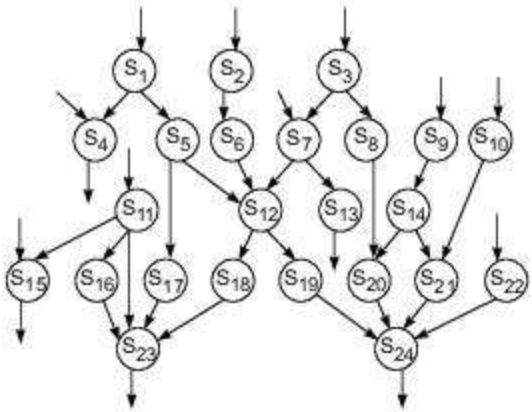
(i) Trace the execution of these two sequences on the two processors by executing the successive blocks. Both caches are initially flushed (empty). Assume a direct-mapping organization in both caches. Indicate the state (RO or RW) of each valid cache block and mark cache miss and bus utilization (busy or idle) in the block trace for each cycle. Assume that the very first memory-access events from both processors take place in cycle 1 simultaneously. Calculate the hit ratio of cache 1 and cache 2, respectively.

(ii) Assume a two-way set-associative cache organization and a LRU cache block replacement policy.

Problem 7.4 Consider the execution of 24 code segments, S_1 through S_{24}, following a given precedence graph on a multiprocessor with four processors and six memory modules as shown below. Assume all segments have the same gain size and execute with equal time. When two or more processors try to access the same memory module at the same time, the request of the lowest numbered processor is granted and the rest of the requests are deferred to later segment time steps.

A processor waiting from an earlier memory-access rejection has seniority priority over new requests to access the same memory module. No processor should wait for more than three steps

to access any given memory module. Each code segment takes a fixed unit time to access a memory and to execute. Assume that the four processors are synchronized in each segment execution instruction cycle.



In some cases, a single segment may require access to several memory modules simultaneously. Ignore the contention problem in the interconnection network. The four processors operate in MIMD mode, and different instructions can be executed by different processors during the same cycle.

What is the average memory bandwidth in words per unit time? Try to achieve the minimum execution time by maximizing the degree of parallelism at all steps.

Note that at each step some of the memory modules may be idle. The highest possible memory bandwidth is six words per step. Some segments may require a wait of no more than three steps before granting of the memory access requested. But such a waiting period should be minimized.

	Processor			
Instr.	P_1	P_2	P_3	P_4
S_1	M_1		M_5	M_1
S_2	M_1	M_2	M_2	M_2
S_3			M_3	M_3
S_4	M_5	M_3	M_2	M_4
S_5	M_1	M_6	M_2	
S_6		M_2	M_1	M_3
S_7			M_6	M_5
S_8			M_2	M_3
S_9	M_3		M_4	M_4
S_{10}	M_1		M_3	M_4	M_4
S_{11}	M_2		M_4	M_5	M_1
S_{12}			M_2	M_6	M_5
S_{13}			M_1		M_6
S_{14}			M_4	M_5	M_4
S_{15}	M_3		M_3	M_3	
S_{16}	M_2		M_2	M_2	M_4
S_{17}	M_1				
S_{18}			M_2	M_5	
S_{19}	M_2		M_2	M_2	M_1
S_{20}			M_3	M_3	M_4
S_{21}	M_2				M_4
S_{22}	M_3		M_1		M_6
S_{23}	M_1		M_2	M_5	M_3
S_{24}				M_3	M_4

Problem 7.5 This problem is based on Fig. 7.11 which combines multiple Fetch&Add requests to the same shared variable in a common memory.

(a) Show the necessary combining network components needed to combine four Fetch&Add (x, e_i) for $i = 1, 2, 3, 4$.

(b) Show the successive snapshots and variations in switch and memory contents, as in Fig. 7.11, for combining the four requests.

Problem 7.6 You have learned about a two-way shuffle (perfect shuffle) in Fig. 2.14 and a four-way shuffle in Fig. 7.9. Generalize the mappings to an m-way shuffle over n objects, where $m \times k = n$ for some integer $k \geq 2$, for the construction of the class of Delta networks introduced by Patel (1980).

(a) Show how to perform a four-way shuffle over 12 objects.

(b) Use a minimum number of 4×3 switch modules and a four-way shuffle mapping as an interstage connection pattern to build a 64-input, 27-output Delta network in three stages.

(c) In general, an n-stage $a^n \times b^n$ Delta network is implemented with $a \times b$ switch modules as shown in Fig. 2.23. Calculate the total number of switch modules needed and specify the

interstage connection pattern from b^n inputs to a^n outputs.

(d) Figure out a simple routing scheme to control the switch settings from stage to stage in an $a^n \times b^n$ Delta network with n stages.

(e) What is the relationship between Omega networks and Delta networks?

Problem 7.7 Prove the following properties associated with multistage Omega networks using different-sized building blocks:

(a) Prove that the number of legitimate states (connections) in a $k \times k$ switch module equals k^k.

(b) Determine the percentage of permutations that can be realized in one pass through a 64-input Omega network built with 2×2 switch modules.

(c) Repeat part (b) for a 64-input Omega network built with 8×8 switch modules.

(d) Repeat part (b) for a 512-input Omega network built with 8×8 switch modules.

Problem 7.8 Consider the interleaved execution of k programs in a multiprogrammed multiprocessor using m wired-NOR synchronization lines on n processors as described in Fig. 7.19a.

In general, the number m_i of barrier lines needed for a program i is estimated as $m_i = b_i [q_i/P_i] + 1$, where b_i = the number of barriers demanded in program i, q_i = the number of processes created in program i, and P_i = the number of processors allocated to program i.

Thus $m = m_1 + m_2 + \ldots + m_k$. For simplicity, assume $b_i = b$ and $q_i = q$ for $i = 1, 2, \ldots, k$, and $P_i = \min(n/k, q)$ processors are allocated to each program i.

Prove that m can be approximated by $b \cdot q \cdot k^2/n + k$, or that the degree of multiprogramming is $k \leq \left(-n + \sqrt{n^2 + 4bqmn}\right)/(2bq)$ in such a multiprocessor system. Note that bq represents the number of required synchronization points, which depends on the parallelism profiles in user programs. For fixed values of bq and n, the maximally allowed multiprogramming degree k increases with respect to $\sqrt{m}$.

Problem 7.9 Wilson (1987) proposed a hierarchical cache/bus architecture (Fig. 7.3) and outlined how multilevel cache coherence can be enforced by extending the write-invalidate protocol. Can you figure out a write-broadcast protocol for achieving multilevel cache coherence on the same hardware platform? Comment on the relative merits of the two protocols. Feel free to modify the hardware in Fig. 7.3 if needed to implement the write-broadcast protocol on the hierarchical bus/cache architecture.

Problem 7.10 Answer the following questions on design choices of multicomputers made in the past;

(a) Why were low-cost processors chosen over expensive processors as processing nodes?

(b) Why was distributed memory chosen over shared memory?

(c) Why was message passing chosen over address switching?

(d) Why was MIMD, MPMD, or SPMD control chosen over SIMD data parallelism?

Problem 7.11 Explain the following terms associated with multicomputer networks and message-passing mechanisms:

(a) Message, packets, and flits.

(b) Store-and-forward routing at packet level.

(c) Wormhole routing at flit level.

(d) Virtual channels versus physical channels.

(e) Buffer deadlock versus channel deadlock.

(f) Buffering flow control using virtual cut-through routing.

(g) Blocking flow control in wormhole routing.

(h) Discard and retransmission flow control.

(i) Detour flow control after being blocked.

(j) Virtual networks and subnetworks.

Problem 7.12

(a) Draw a 16-input Omega network using 2×2 switches as building blocks.

(b) Show the switch settings for routing a message from node 1011 to node 0101 and from node 0111 too node 1001 simultaneously. Does blocking exist in this case?

(c) Determine how many permutations can be implemented in one pass through this Omega network. What is the percentage of one-pass permutations among all permutations?

(d) What is the maximum number of passes needed to implement any permutation through the network?

Problem 7.13 Explain the following terms as applied to communication patterns in a message-passing network:

(a) Unicast versus multicast

(b) Broadcast versus conference

(c) Channel bandwidth

(d) Communication latency

(e) Network partitioning for multicasting communications

Problem 7.14 Determine the optimal routing paths in the following mesh and hypercube multicomputers.

(a) Consider a 64-node hypercube network. Based on the E-cube routing algorithm, show how to route a message from node (101101) to node (011010). All intermediate nodes must be identified on the routing path.

(b) Determine two optimal routes for multicast on an 8×8 mesh, subject to the following constraints separately. The source node is (3, 5), and there are 10 destination nodes (1, 1), (1,2), (1,6), (2, 1), (4, 1), (5,5), (5,7), (6, 1), (7, 1), (7,5). (i) The first multicast route should be implemented with a minimum number of channels. (ii) The second multicast route should result in minimum distances from the source to each of the 10 destinations.

(c) Based on the greedy algorithm (Fig. 7.38), determine a suboptimal multicast route, with minimum distances from the source to all destinations using as few traffic channels as possible, on a 16-node hypercube network. The source node is (1010), and there are 9 destination nodes (0000), (0001), (0011), (0100), (0101), (0111), (1111), (1101), and (1001).

Problem 7.15 Prove the following statements with reasoning or analysis or counter-examples:

(a) Prove that E-cube routing is deadlock-free on a wormhole-routed hypercube with a pair of opposite unidirectional channels between adjacent nodes.

(b) Prove that X-Y routing is deadlock-free on a 2D mesh.

(c) Prove that E-cube routing on the 3D mesh (k-ary n-cube) used in the J-Machine is deadlock-free with wormhole routing and blocking flow control.

Problem 7.16 Study the Turn model for adaptive routing proposed by Glass and Ni (1992) in the 1992 *Annual International Symposium on Computer Architecture*. Answer the following questions:

(a) Why is the Turn model deadlock-free from having cycles?

(b) How can the Turn model be applied on an n-dimensional mesh to prevent deadlock?

(c) How can the Turn model be applied on a k-ary n-cube to prevent deadlock?

Problem 7.17 The following assignments are related to the greedy algorithm for multicast routing on a wormhole-routed hypercube network.

(a) Formulate the successive steps of the greedy algorithm (Example 7.8) as a minimum cover problem, similar to that practiced in Karnaugh maps.

(b) Prove that the greedy algorithm always yields the minimum network traffic and minimum distance from the source to any of the destinations.

Problem 7.18 Consider the implementation of Goodman's write-once cache coherence protocol in a bus-connected multiprocessor system. Specify the use of additional bus lines to inhibit the main memory when the memory copy is invalid. Also specify all other hardware mechanisms and software support needed for an economical and fast implementation of the Goodman protocol.

Explain why this protocol will reduce bus traffic and how unnecessary invalidations can be eliminated. Consult if necessary the two related papers published by Goodman in 1983 and 1990.

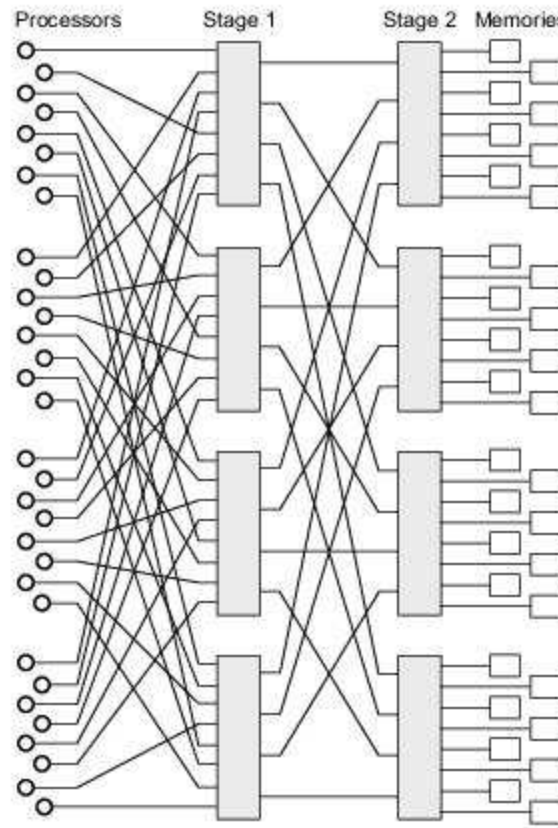
Problem 7.19 Study the paper by Archibald and Baer (1986) which evaluated various cache coherence protocols using a multiprocessor simulation model. Explain the Dragon protocol implemented in the Dragon multiprocessor workstation at the Xerox Palo Alto Research Center. Compare the relative merits of the Goodman protocol, the Firefly protocol, and the Dragon protocol in the context of implementation requirements and expected performance.

Problem 7.20 The Cedar multiprocessor at Illinois was built with a clustered Omega network as shown below. Four 8 × 4 crossbar switches were used in the first stage and four 4 × 8 crossbar switches were used in the second stage. There were 32 processors and 32 memory modules, divided into four clusters with eight of each per cluster.

(a) Figure out a fixed priority scheme to avoid conflicts in using the crossbar switches for nonblocking connections. For simplicity, consider only the forward connections from the processors to the memory modules.

(b) Suppose both stages use 8 × 8 crossbar switches. Design a two-stage Cedar network to provide switched connections between 64 processors and 64 memory modules, again in a clustered manner similar to the above Cedar network design.

(c) Further expand the Cedar network to three stages using 8 × 8 crossbar switches as building blocks to connect 512 processors and 512 memory modules. Show the schematic interconnections in all three stages from the input end to the output end.



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.

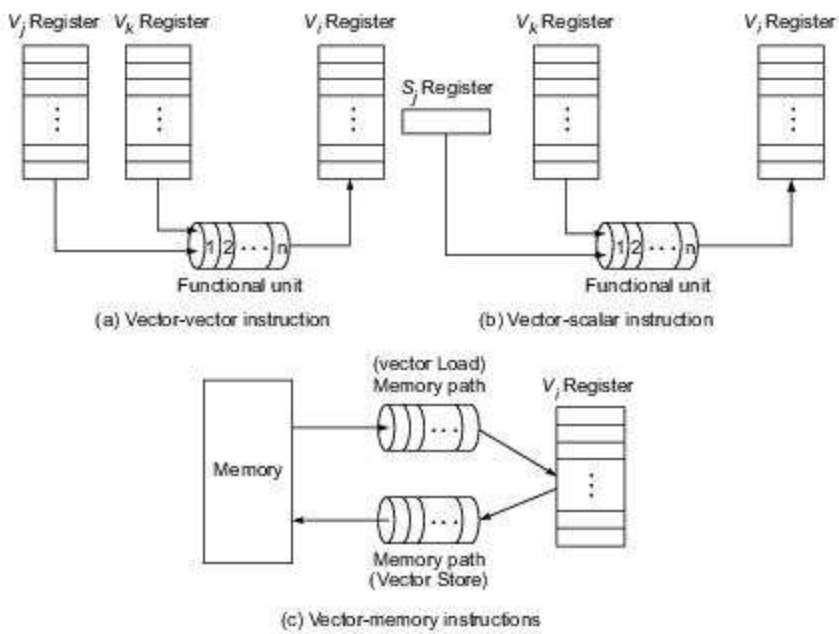


Fig. 8.1 Vector instruction types in Cray-like computers

(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:

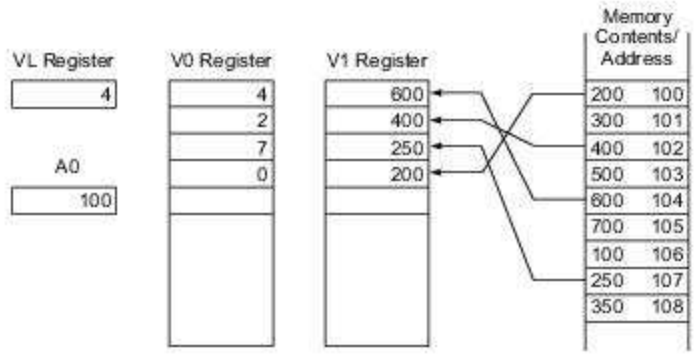
$$f_{10} : V_0 \times V_m \rightarrow V_1 \tag{8.10}$$

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

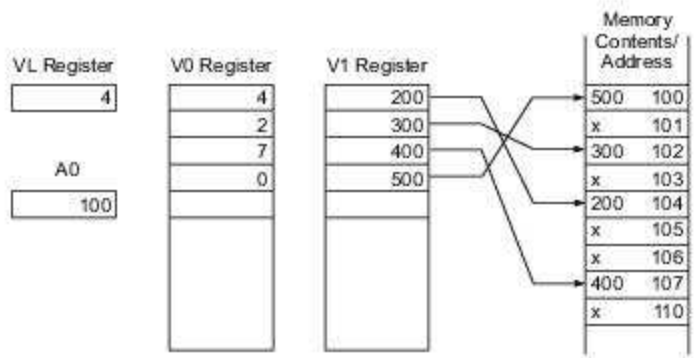
Example 8.1 Gather, scatter, and masking instructions in the Cray Y-MP (Cray Research, 1990)

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.

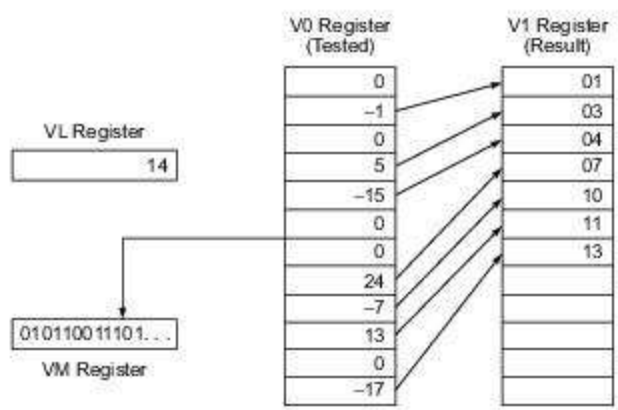
The offsets (indices) from the base address are retrieved from the vector register *V0*. The effective memory addresses are obtained by adding the base address to the indices.



(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.

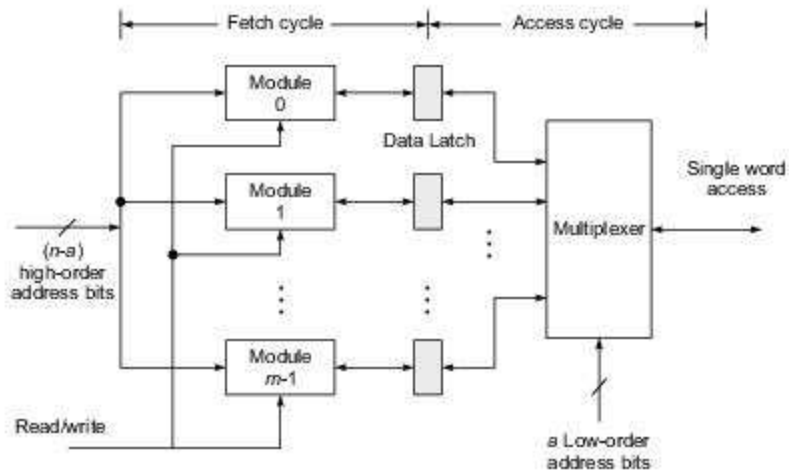
C-Access Memory Organization The m-way low-order interleaved memory structure shown in Figs. 5.15a and 5.16 allows m memory words to be accessed concurrently in an overlapped manner. This *concurrent access* has been called *C-access* as illustrated in Fig. 5.16b.

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.

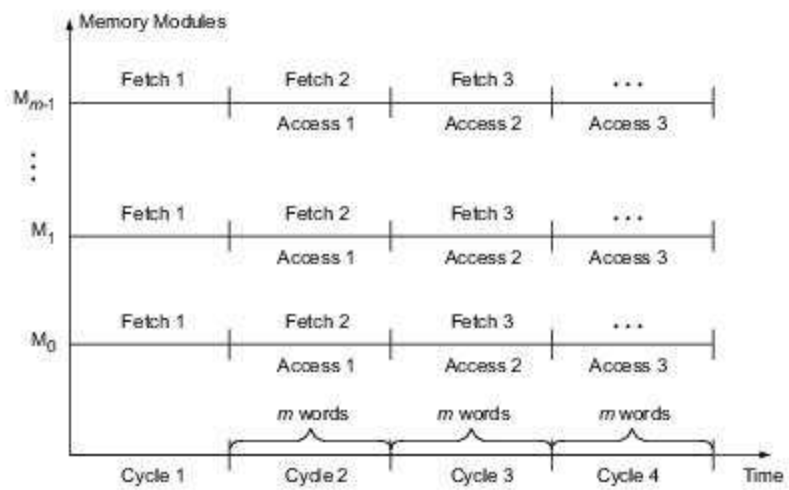
To access a vector with a stride of 1, successive addresses are latched in the address buffer at the rate of one per cycle. Effectively it takes m minor cycles to fetch m words, which equals one (major) memory cycle as stated in Eq. 5.4 and Fig. 5.16b.

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.

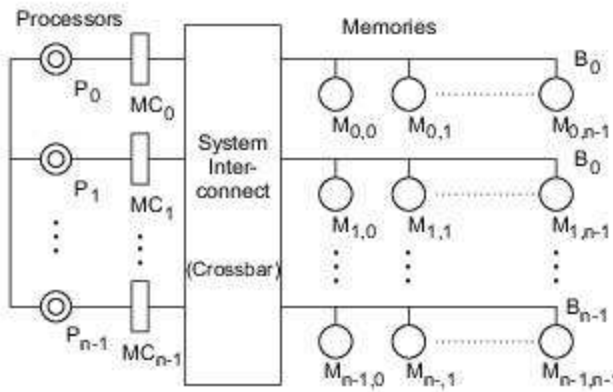


Fig. 8.4 The C/S memory organization with $m = n$. (Courtesy of D.K. Panda, 1990)

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.

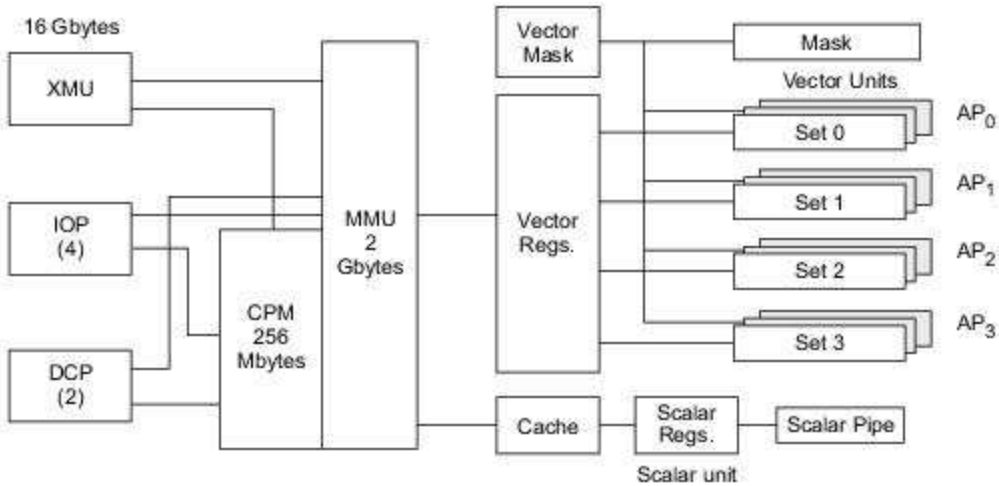
Shared communication registers and reconfigurable vector registers were special features in these machines. Hitachi offered the 820 Series providing a 3-Gflops peak performance. Japanese supercomputers were at one time strong in high-speed hardware and interactive vectorizing compilers.

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.

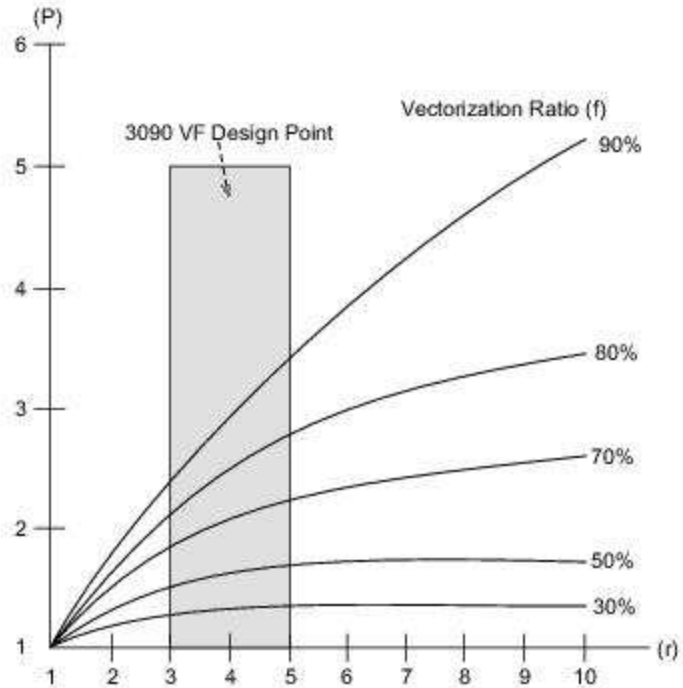


Fig. 8.6 Speedup performance of vector processing over scalar processing in the IBM 3090/VF (Courtesy of IBM Corporation, 1986)

The designers of the 3090/VF chose a speed ratio in the range $3 \le r \le 5$ 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 $f \le 30\%$, 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 $P \to 1$ if $f \to 0$.

On the other hand, $P \to r$ when $f \to 1$. Scientific supercomputer designers like Cray and Japanese manufacturers often chose a much higher speed ratio, say, $10 \le r \le 25$, 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.

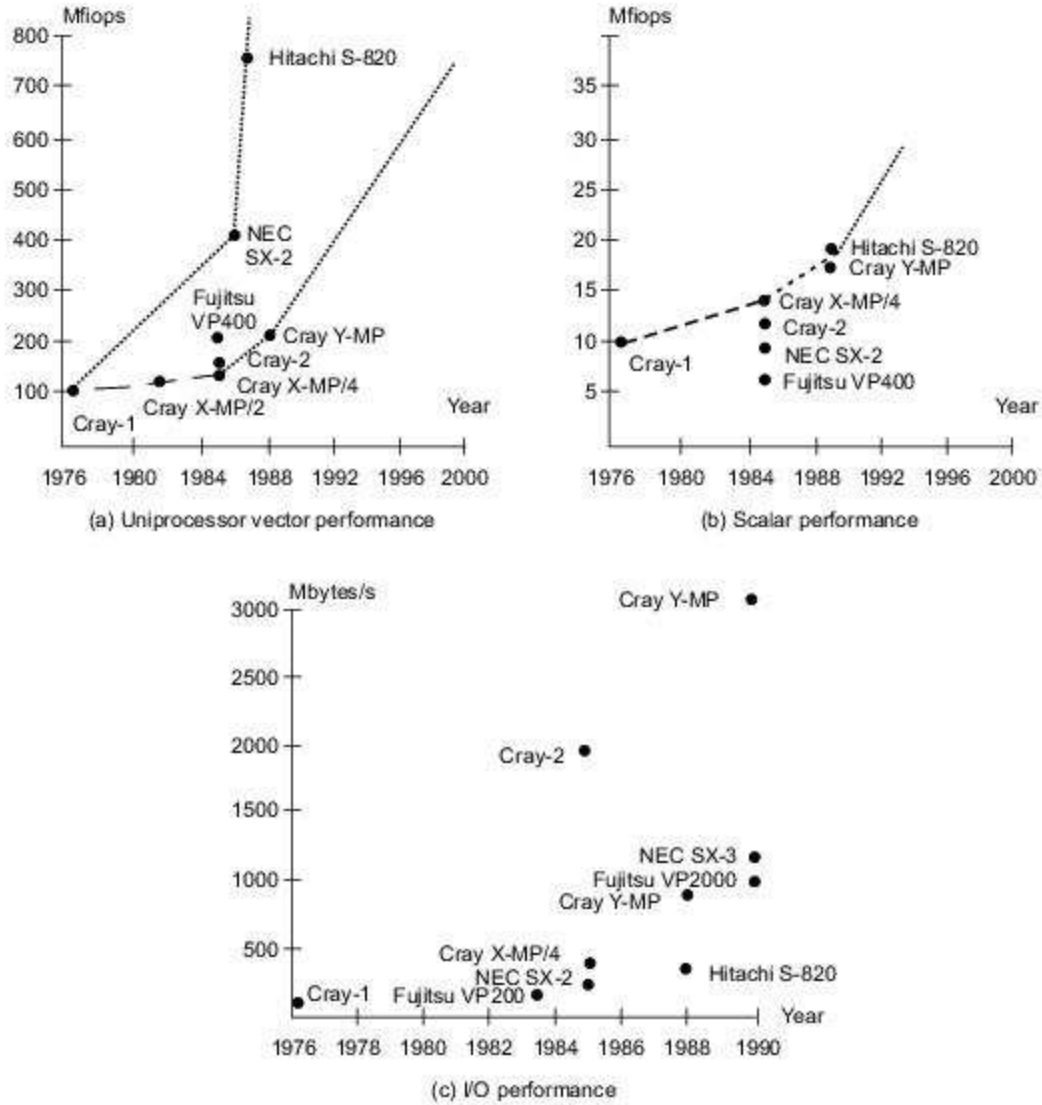


Fig. 8.7 Some reported supercomputer performance data (Source: Smith, Hsu, and Hsiung, IEEE Supercomputing Conference, 1990)

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.

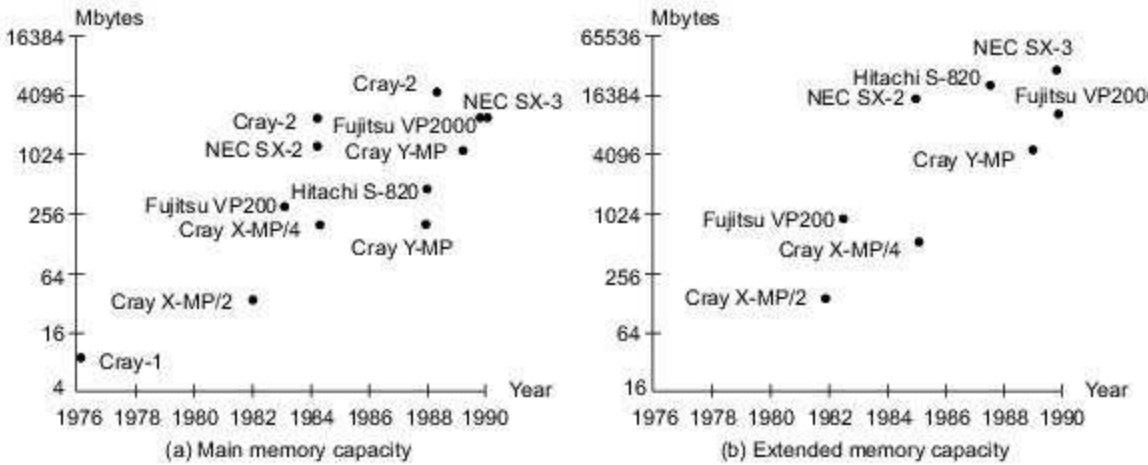


Fig. 8.8 Supercomputer memory capacities (Source: Smith, Hsu, and Hsiung, *IEEE Supercomputing Conference*, 1990)

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.

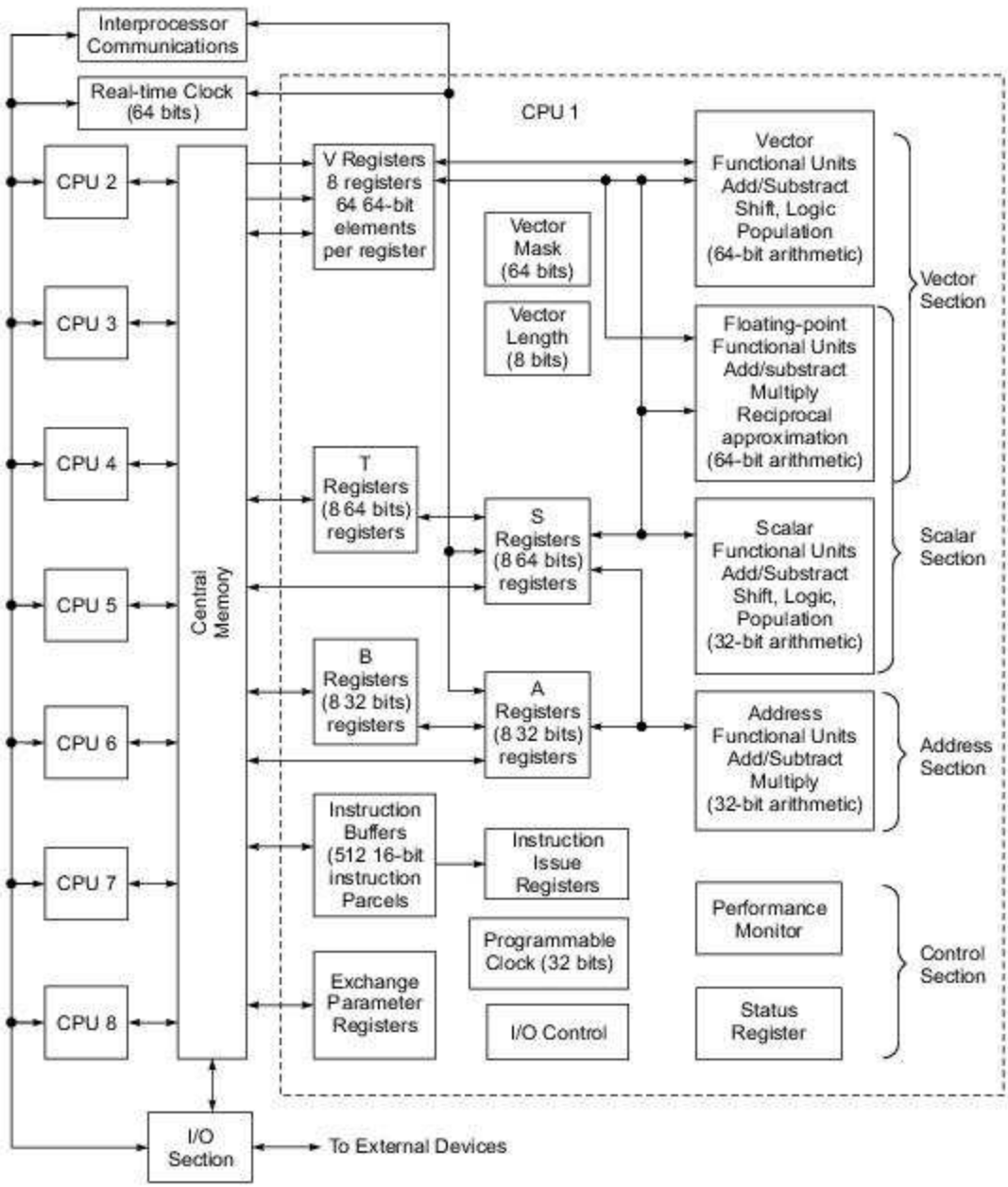


Fig. 8.9 Cray Y-MP 816 system organization (Courtesy of Cray Research, 1991)

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.

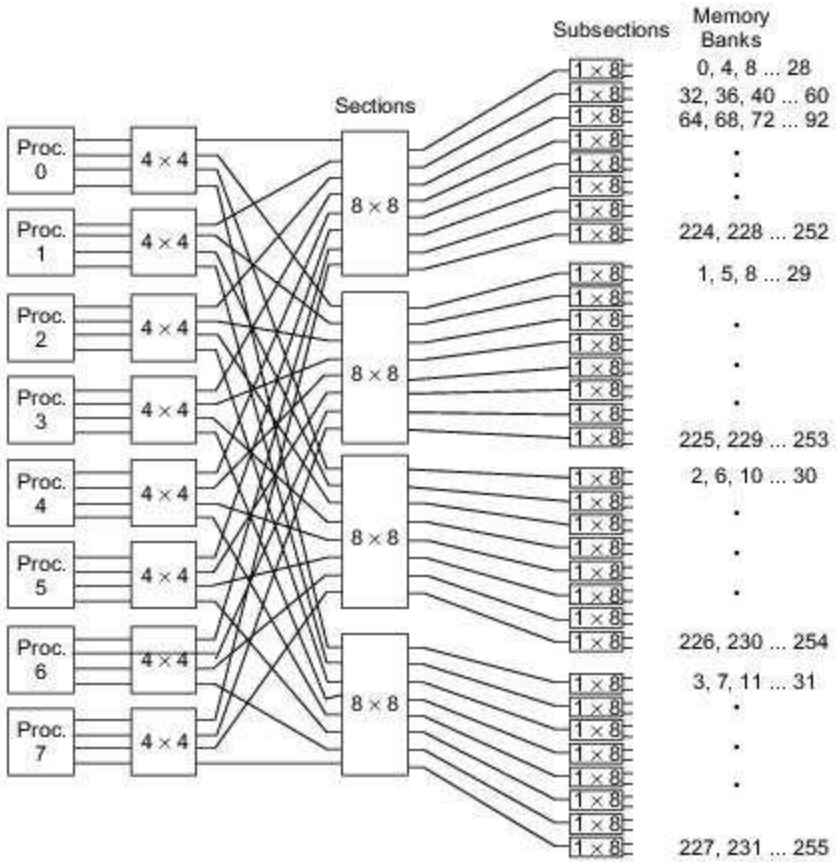


Fig. 8.10 Schematic logic diagram of the crossbar network between 8 processors and 256 memory banks in the Cray Y-MP 816

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.

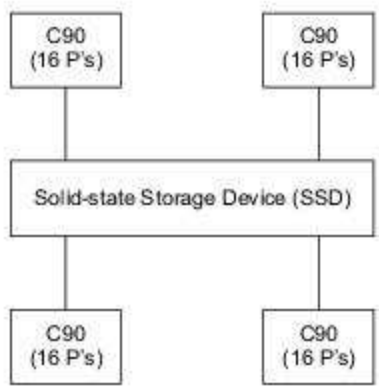


Fig. 8.11 Four Cray Y-MP C-90's connected to a common SSD forming a loosely coupled 64-way parallel system

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.

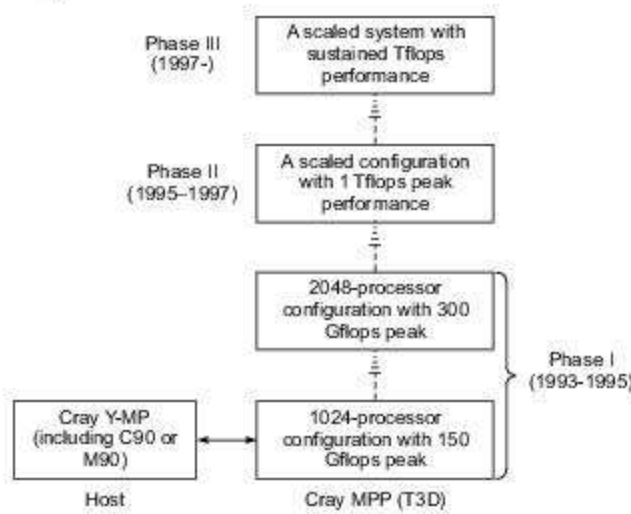


Fig. 8.12 The development phases of the original Cray/MPP system (Courtesy of Cray Research, 1992)

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.

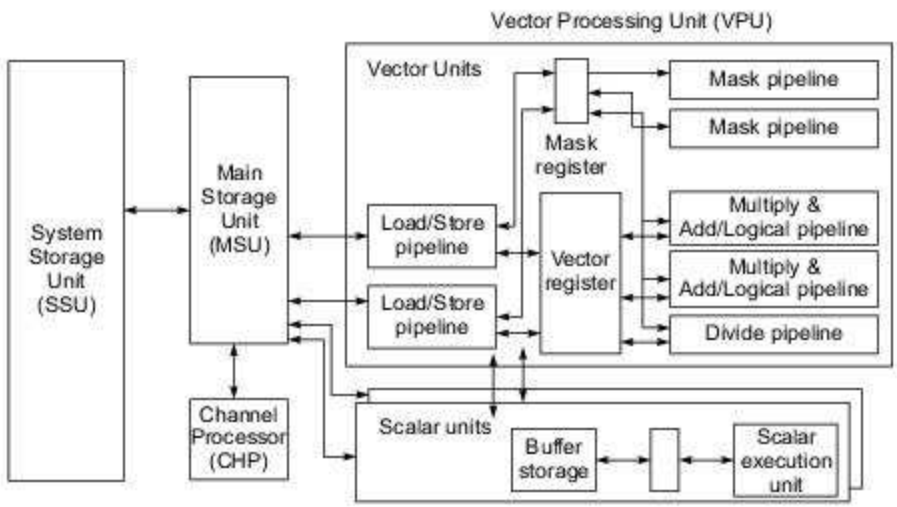
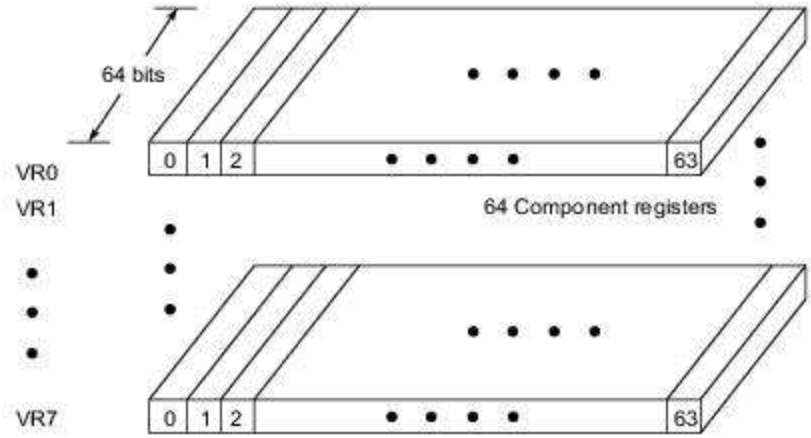


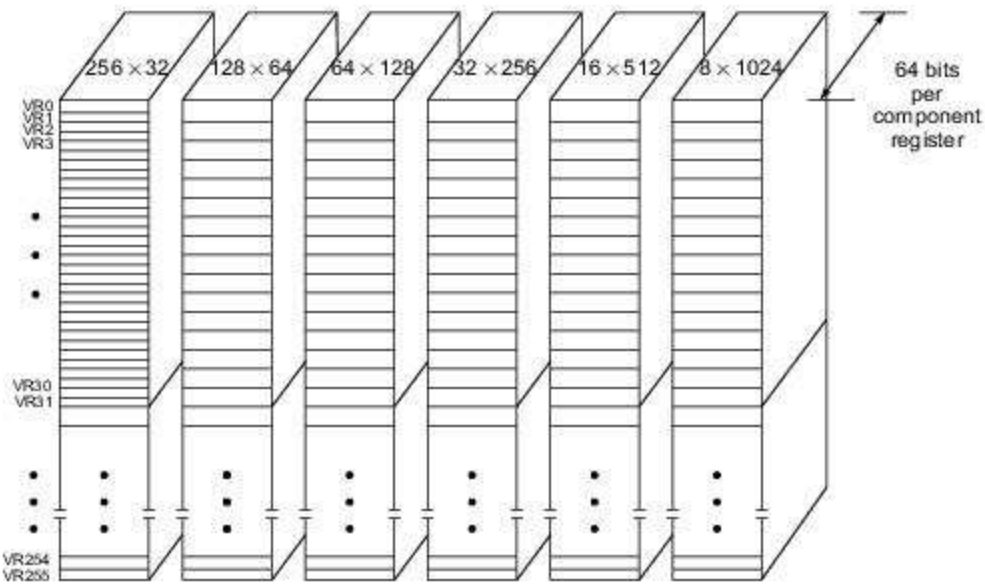
Fig. 8.13 The Fujitsu VP2000 Series supercomputer architecture (Courtesy of Fujitsu, 1991)

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 ($8 \times 64 \times 64$ 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)
    ZZl(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.

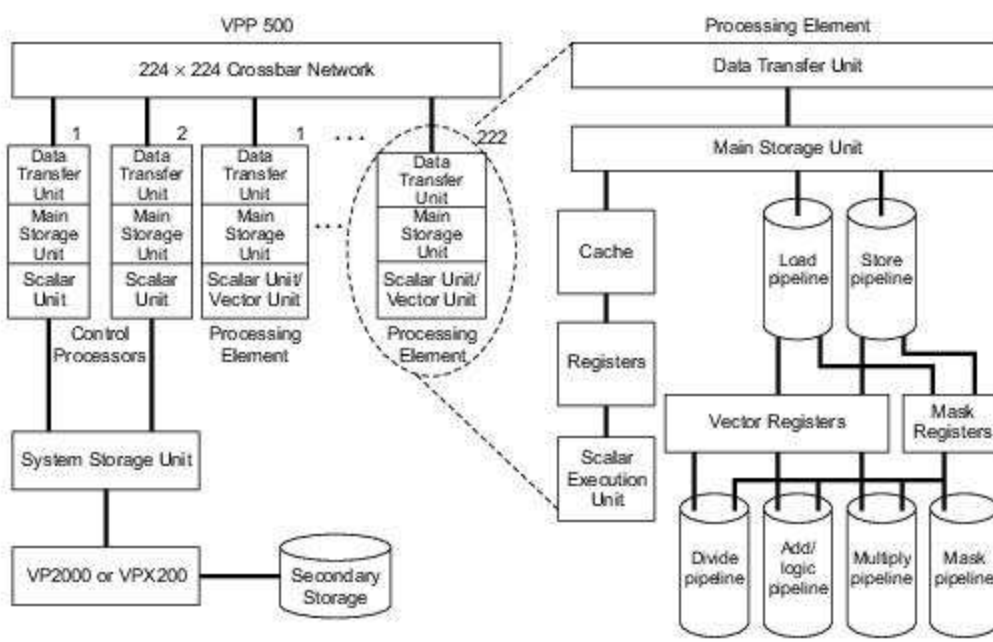


Fig. 8.15 The Fujitsu VPP500 architecture (Courtesy of Fujitsu, 1992)

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.

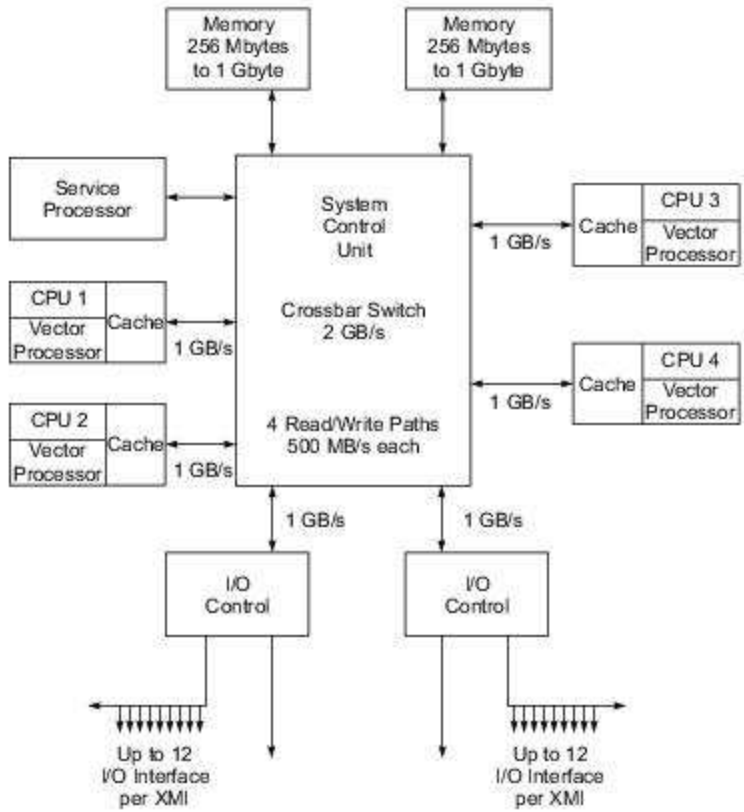
Up to 512 Mbytes of main memory were available using 1-Mbit DRAMs on 64-Mbyte arrays. Up to 2 Gbytes of extended memory were available using 4-Mbit DRAMs. Various I/O channels provided an aggregate data transfer rate of 320 Mbytes/s. The crossbar had eight ports to four processors, two memory modules, 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*

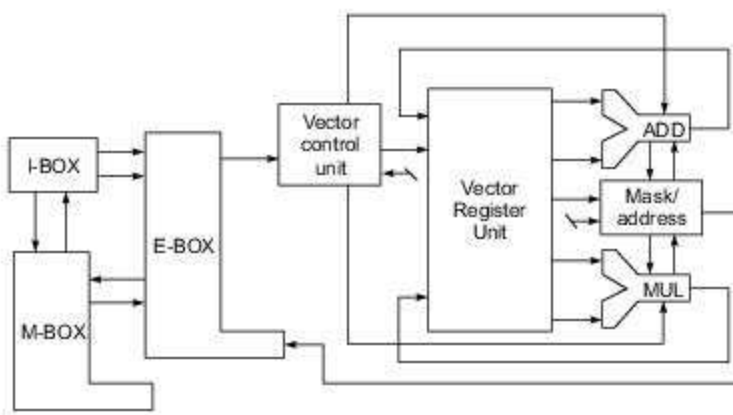
Machine Characteristics	*IBM ES/9000 –900 VF*	*DEC VAX 9000/440 VP*	*CDC Cyber 2000V*
Number of processors	6 processors each attached to a vector facility	4 processors with vector boxes	2 central processors with vector hardware
Machine cycle time	9 ns	16 ns	9 ns
Maximum memory	1 Gbyte	512 Mbytes	512 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.

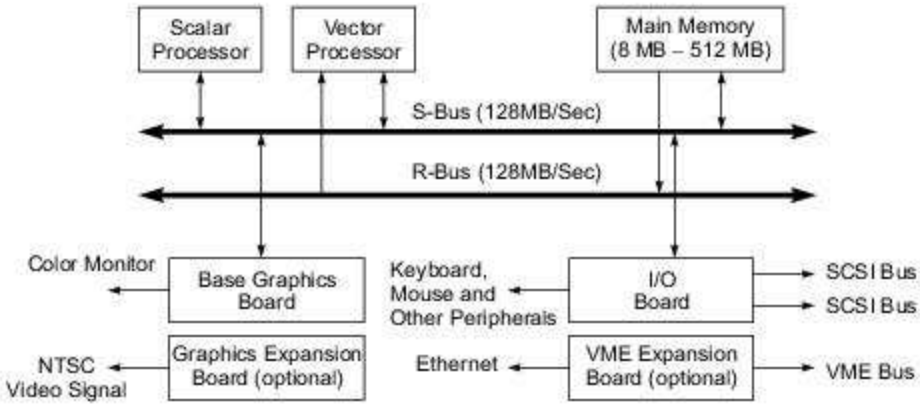
Silicon Graphics produced the 4-D Series using four R3000 CPUs in a single workstation without vector hardware. Stardent Computer Systems produced a departmental supercomputer, called the Stardent 3000, with custom-designed vector hardware.

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).

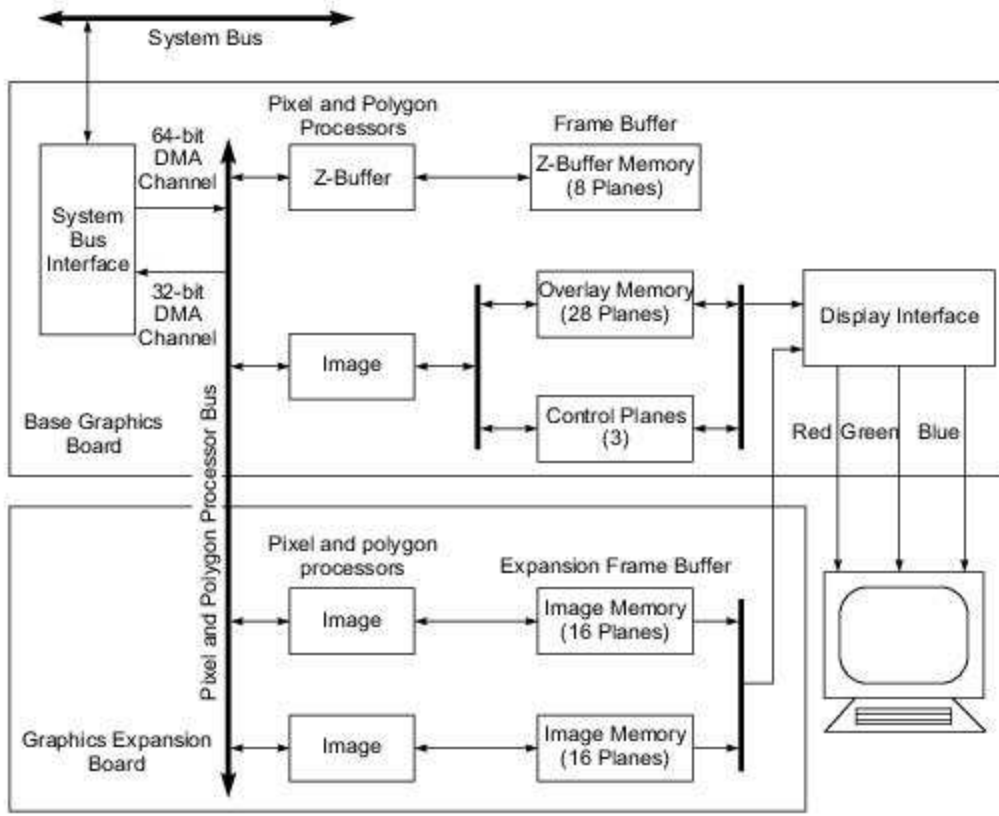
The system featured R3000/R3010 processors/floating-point units. The vector processors were custom-designed. A 32-MHz clock was used. There were 128 Kbytes of cache; one half was used for instructions and the other half for data.

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:

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

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          (8.13)
    Add         R2, R1
    Store       Y(I), R2
10  Continue
```

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:

```
M(x : x + N − 1)  →  V1      Vector load
M(y : y + N − 1)  →  V2      Vector load
S × V1            →  V1      Vector multiply   (8.14)
```

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

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*
$V1(I) = V2(I) + V3(I) \times V4(I)$	2
$V1(I) = B(I) + C(I)$	3
$A(I) = V1(I) \times S + B(I)$	4
$A(I) = V1(I) + B(I) + C(I)$	5
$A(I) = B(I) + S \times C(I)$	5
$A(I) = B(I) + C(I) + D(I)$	6
$A(I) = Q \times V1(I) (R \times B(I) + C(I))$	7
$A(I) = B(I) \times C(I) + D(I) \times V1(I)$	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
$A(I) = B(I) \times C(I) + D(I) \times E(I) \times S$	9
$A(I) = (A(I) + B(I) \times C(I) + D(I)) \times (I)$	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.

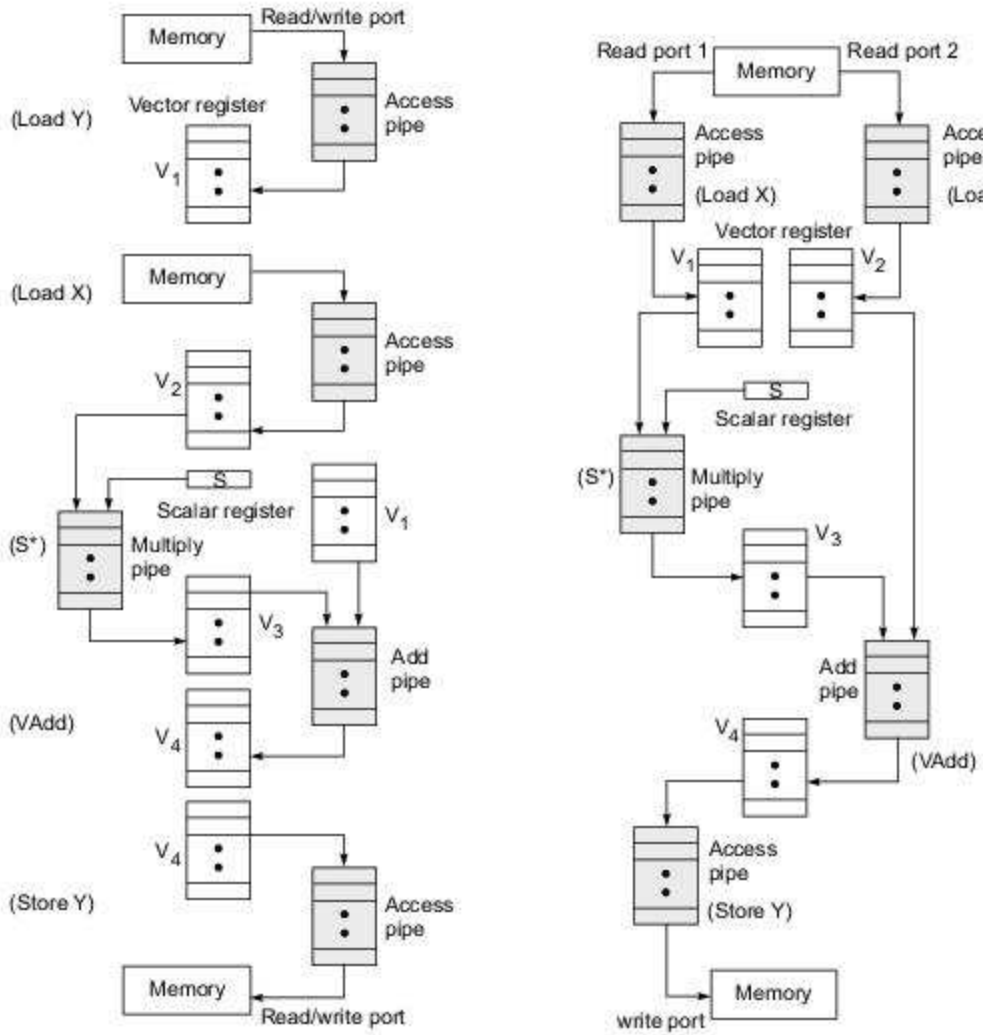
To avoid conflicts among different vector operations, the vector registers and functional pipelines must be reserved before a vector chain can be established. The vector chaining and the timing relationship are illustrated in Figs. 8.18 and 8.19 for executing the vectorized SAXPY code specified in Eq.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

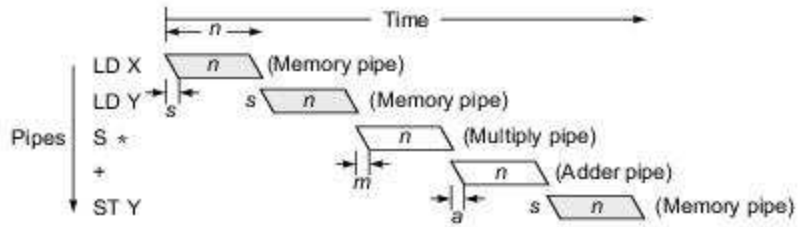


(a) Limited chaining using only one memory-access pipe in the Cray 1

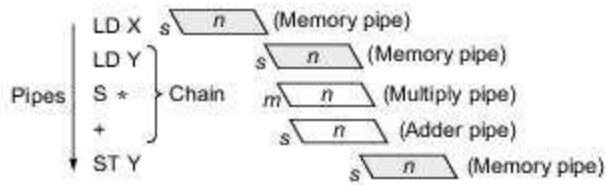
(b) Complete chaining using three memory-access pipes in the Cray X-MP

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)

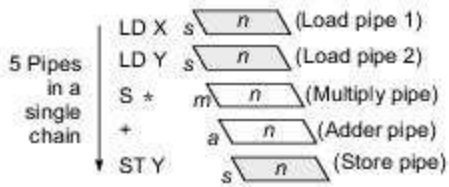
To compare the time required for chaining these pipelines, Fig. 8.19a shows that roughly $5n$ cycles are needed to perform the vector operations sequentially without any overlapping or any chaining. The Cray 1 requires about $3n$ cycles to execute, corresponding to about n cycles for each vector chain.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

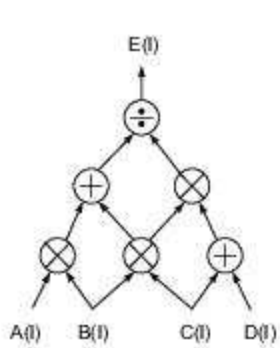
Setup of the Pipenet Figures 8.20a through 8.20d show how to convert from a program graph to a pipenet. Whenever a CVF is to be evaluated, the crossbar networks are programmed to set up a connectivity pattern among the FPs that matches the data flow pattern in the CVF.

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.

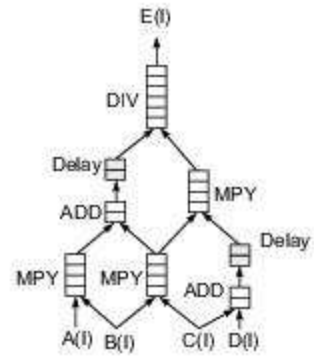
The program graph in Fig. 8.20a corresponds to the following CVF:

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

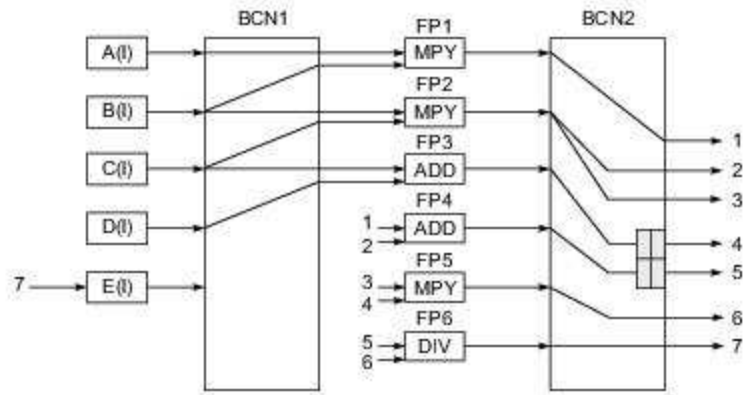
for $I = 1, 2, \ldots, n$. This CVF has four input vectors A(I), B(I), C(I), and D(I) and one output vector E(I) which demand five memory-access operations. In addition, there are seven vector arithmetic operations involved.



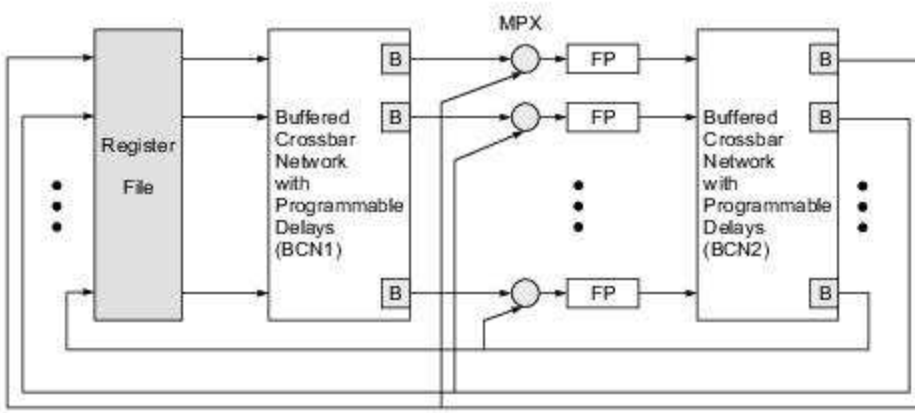
(a) A program graph



(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.

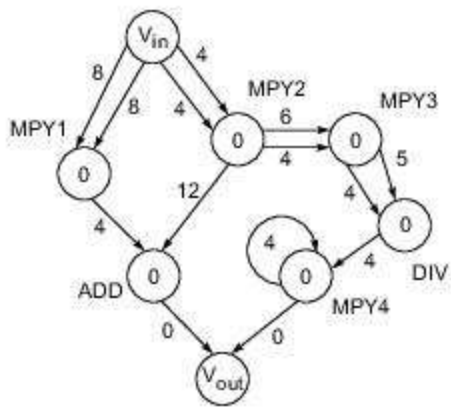
Example 8.9 Program graph transformation to set up a pipenet (Hwang and Xu, 1988)

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

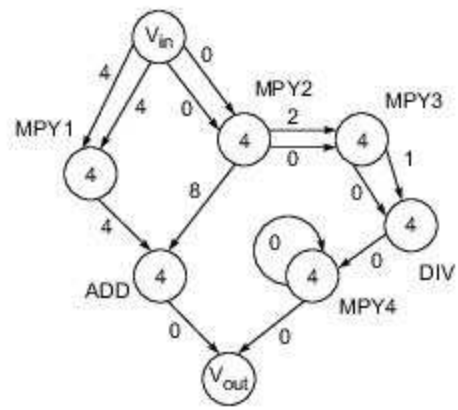
$$E(I) = [B(I) \times C(I)] + [C(I) \times D(I)]$$
$$F(I) = [C(I) \times D(I)] \times [C(I-2) \times D(I-2)] \quad (8.18)$$
$$G(I) = [F(I)/F(I-1)] \times G(I-4)$$

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).

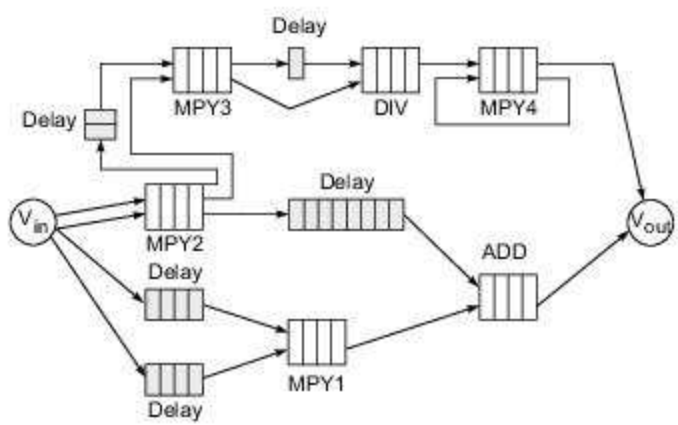
Applying Lemma 1, we add four-cycle delays to each operator node and subtract four-cycle delays from all incoming edges. The transformed graph is obtained in Fig. 8.21b. This is a 4-graph with all nodal delays equal to four cycles. Therefore, one can construct a pipenet with all FPs having four pipeline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

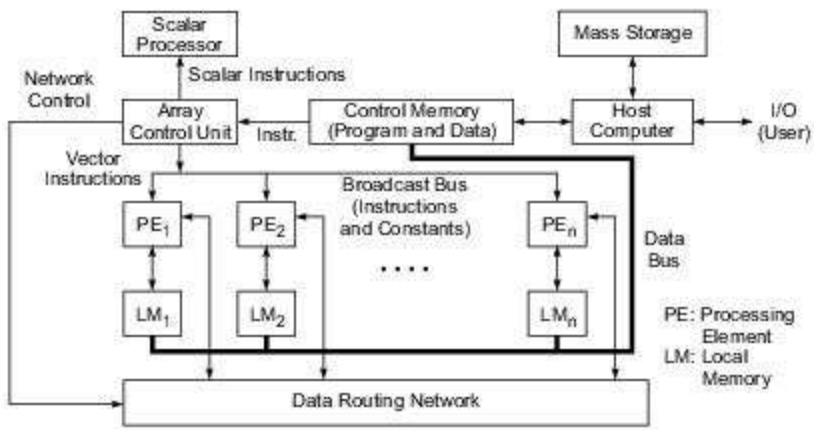
Possibly other than in specialized research platforms, no computer system of the original SIMD model is in operation today. However, a study of this model of computer system can still serve the twin purpose of bringing out (i) the basic SIMD concept and its related issues, and (ii) an important historical 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

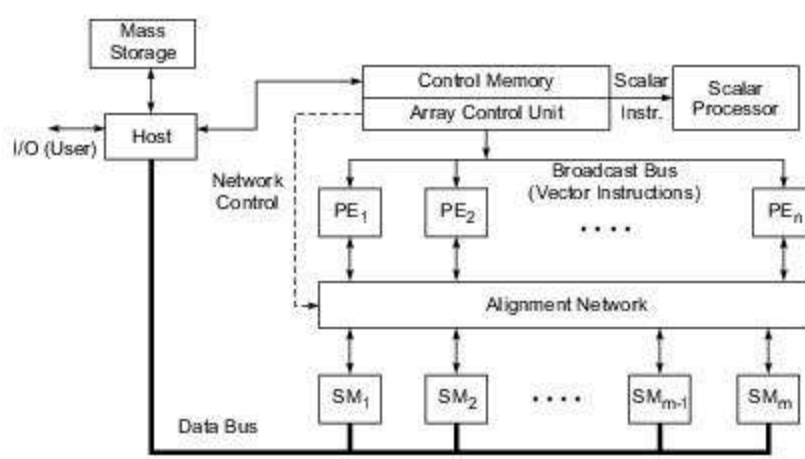
Two SIMD computer models are described below based on the memory distribution and addressing scheme used. Most SIMD computers use a single control unit and distributed memories, except for a few that use associative memories.

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-l.

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.

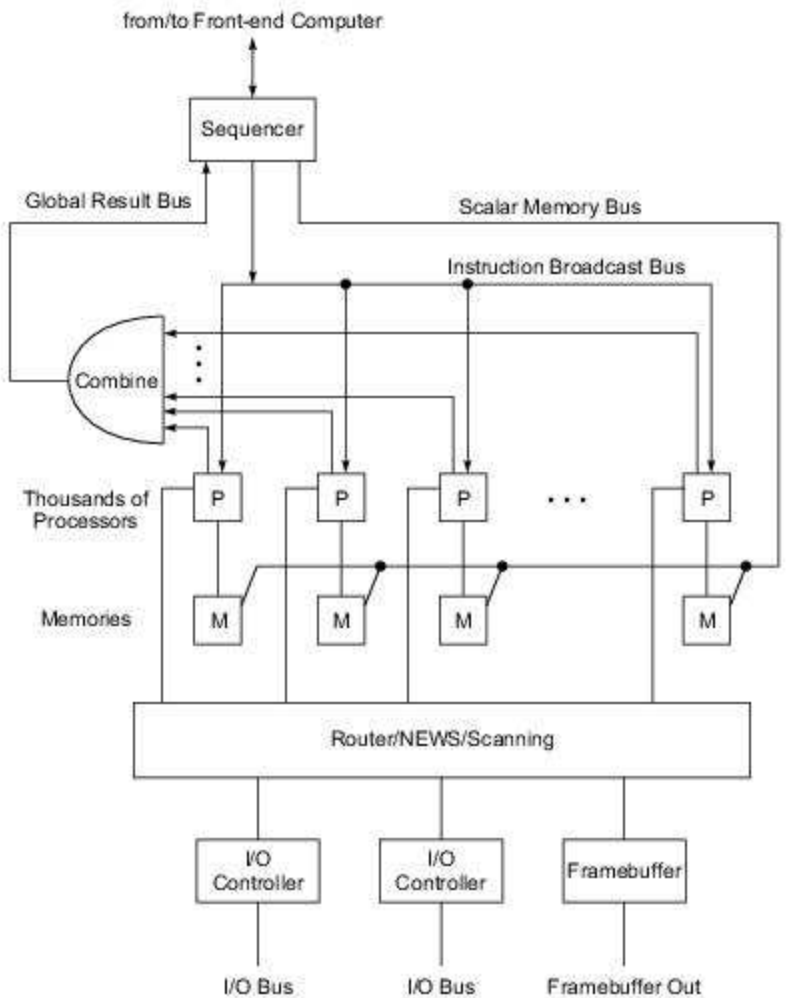


Fig. 8.23 The architecture of the Connection Machine CM-2 (Courtesy of Thinking Machines Corporation, 1990)

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.

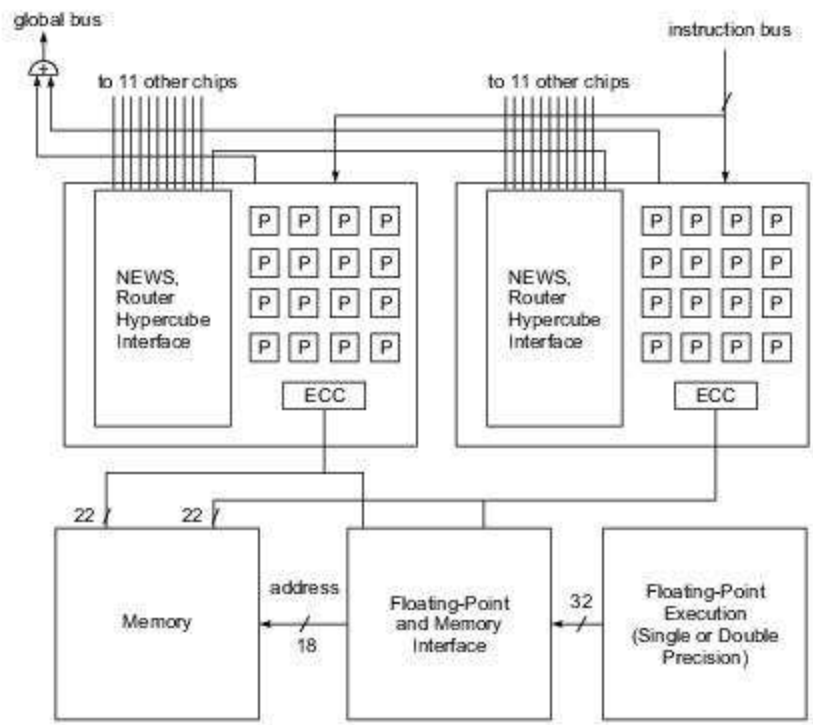


Fig. 8.24 A CM-2 processing node consisting of two processor chips and some memory and floating-point chips (Courtesy of Thinking Machines Corporation, 1990)

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 $2^{18} = 256$K 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.

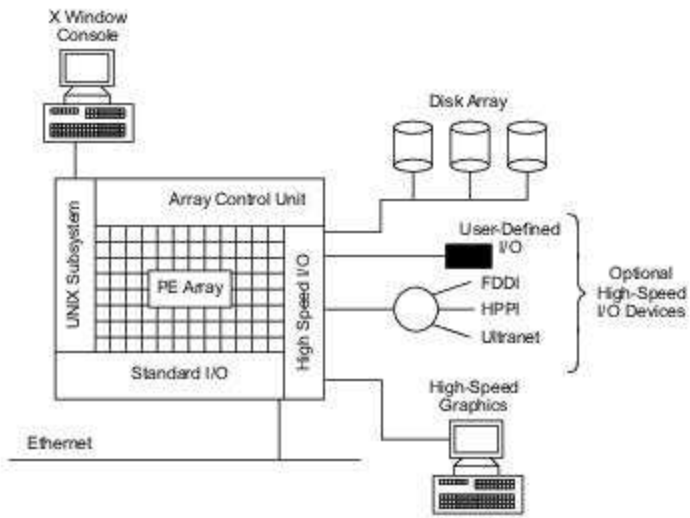
The MP-1 family included configurations with 1024, 4096, and up to 16,384 processors. The peak performance of the 16K-processor configuration was 26,000 MIPS in 32-bit RISC integer operations. The

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

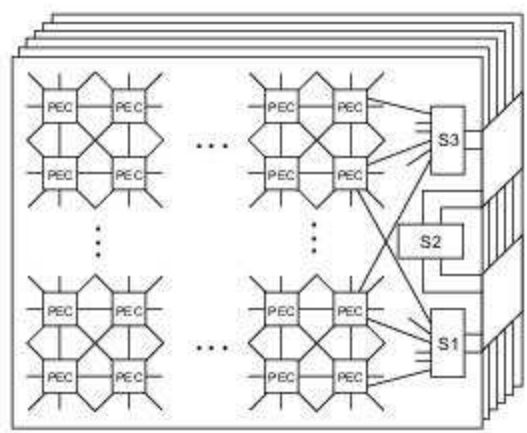
Array Control Unit The ACU was a 14-MIPS scalar RISC processor using a demand-paging instruction memory. The ACU fetched and decoded MP-1 instructions, computed addresses and scalar data values, issued control signals to the PE array, and monitored the status of the PE array.

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.



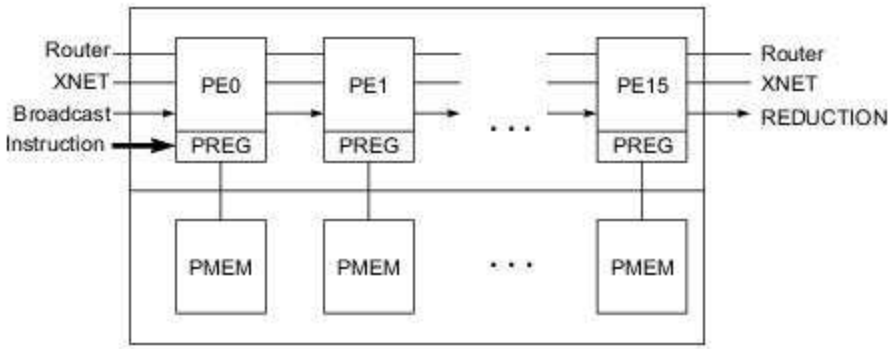
(a) MP-1 System Block Diagram



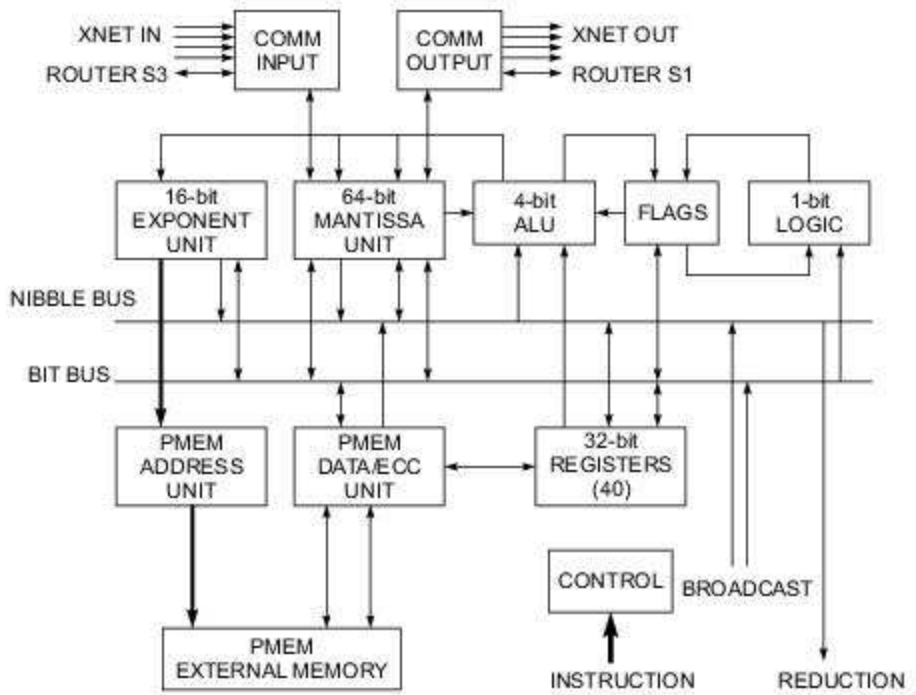
(b) Array of PE clusters

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-I (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-l. 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.

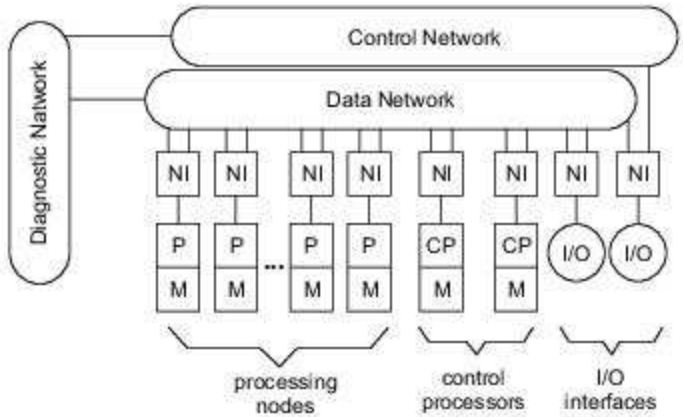


Fig. 8.27 The network architecture of the Connection Machine CM-5 (Courtesy of Leiserson et al, Thinking Machines Corporation, 1992)

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.

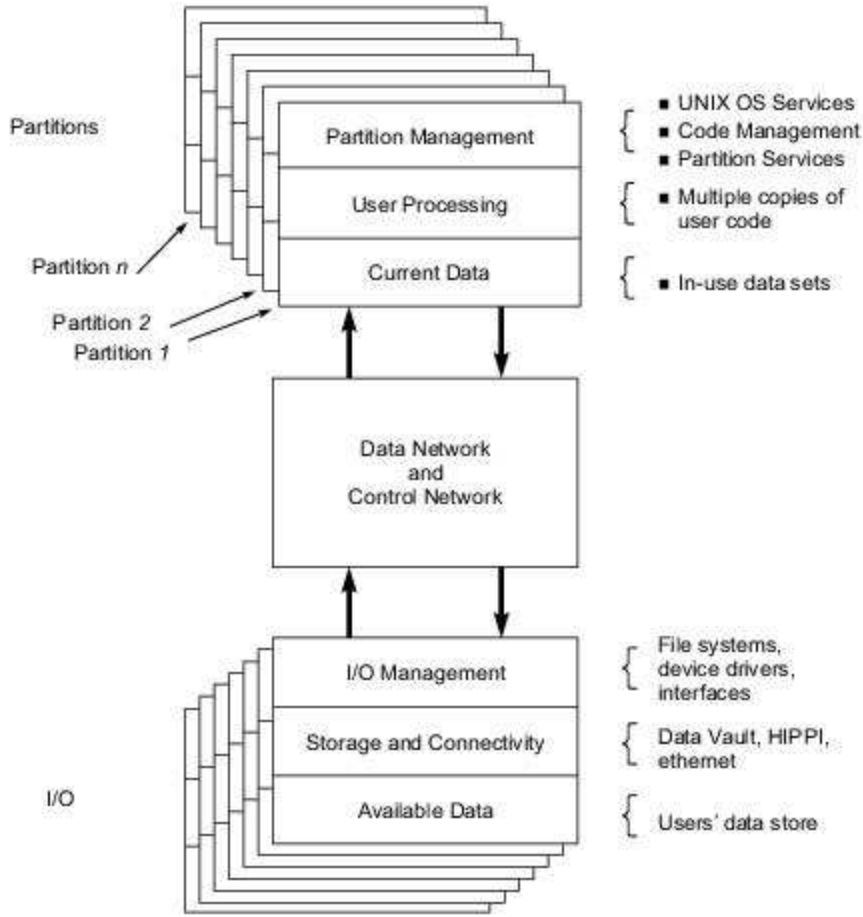


Fig. 8.28 Distributed control on the CM-5 with concurrent user partitions and I/O activities (Courtesy of Thinking Machines Corporation, 1992)

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.

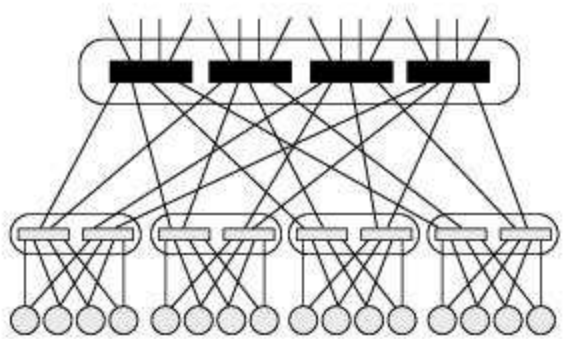


Fig. 8.29 CM-5 data network implemented with a 4-ary fat tree (Courtesy of Leiserson et al, Thinking Machines Corporation, 1992)

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.

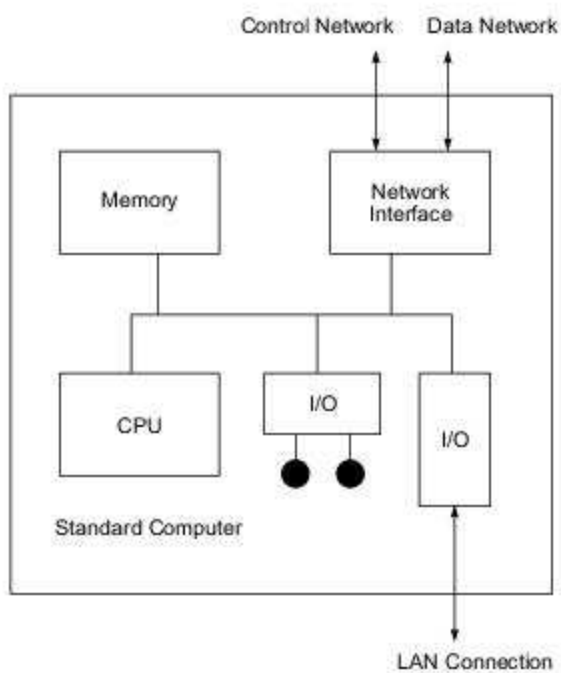


Fig. 8.30 The control processor in the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

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.

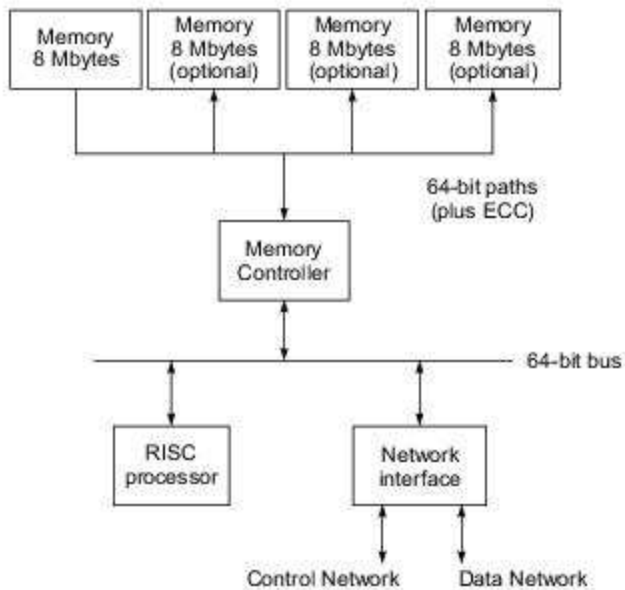


Fig. 8.31 The processing node in the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

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.

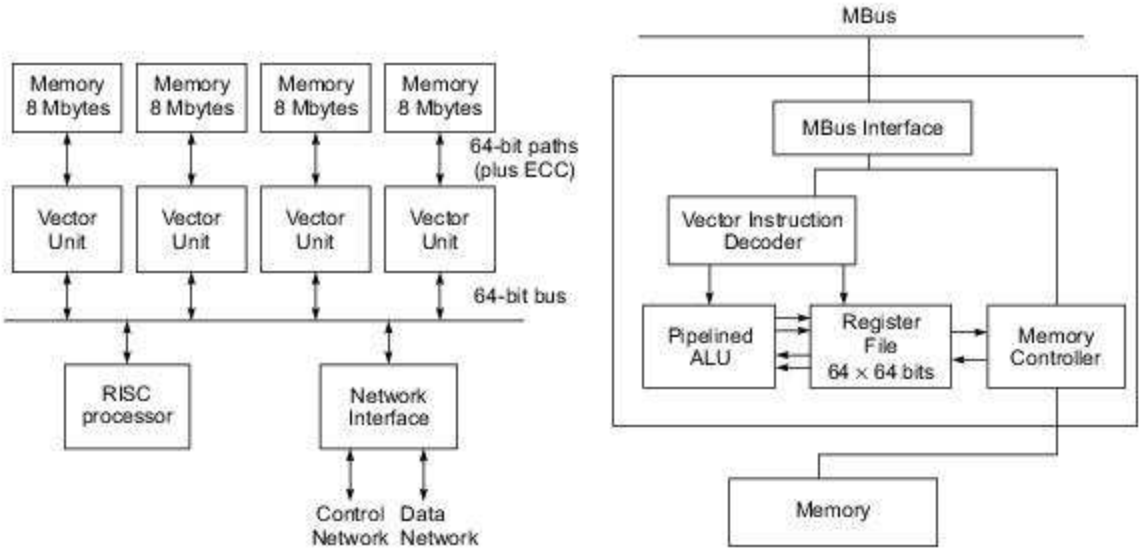


Fig. 8.32 The processing node with vector units in the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

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).

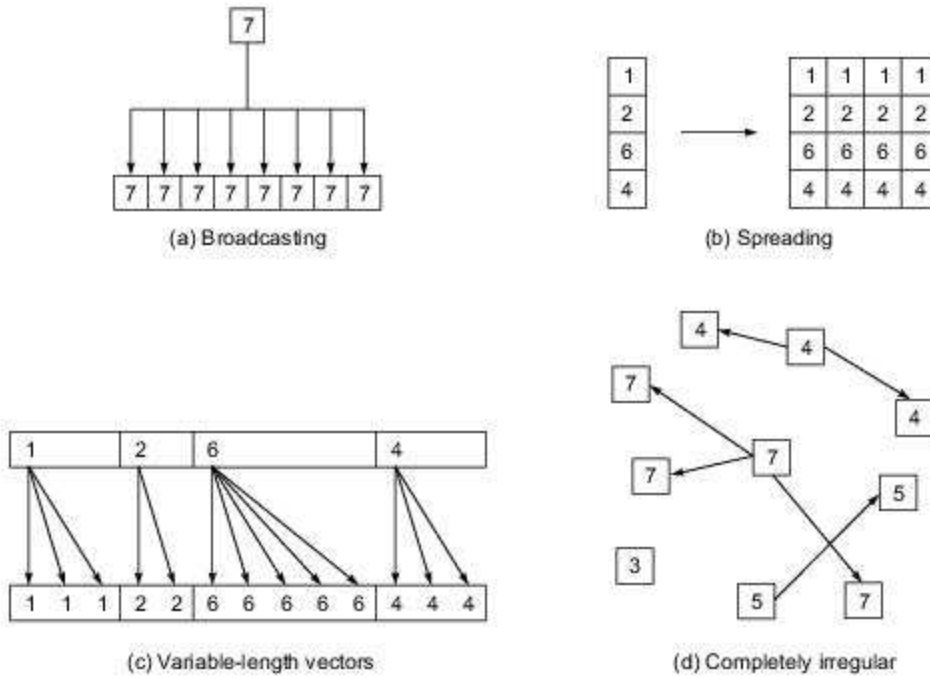


Fig. 8.33 Replication operations for interprocessor communications on CM-5 (Courtesy of Thinking Machines Corporation, 1992)

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.

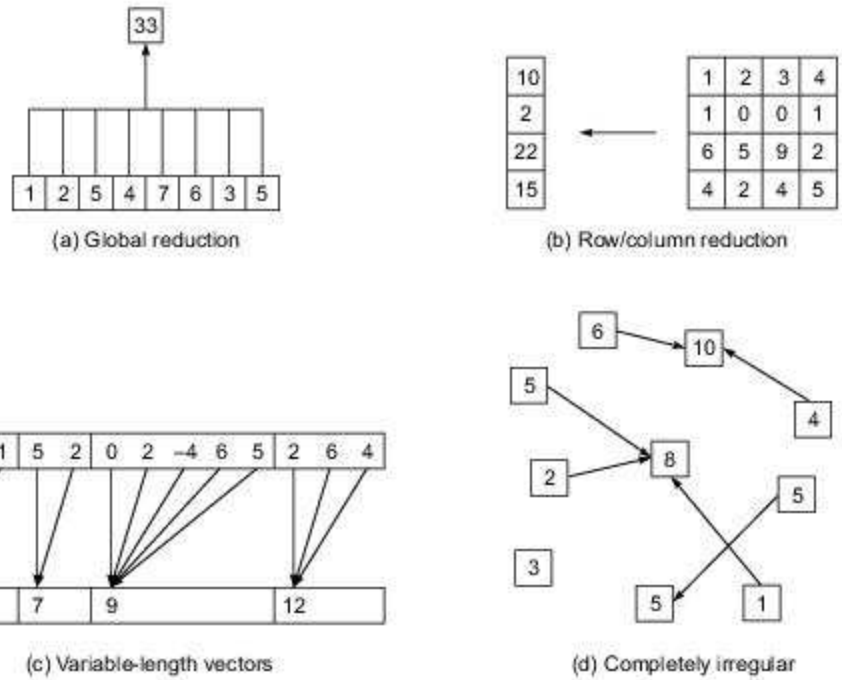


Fig. 8.34 Reduction operations on the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

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.

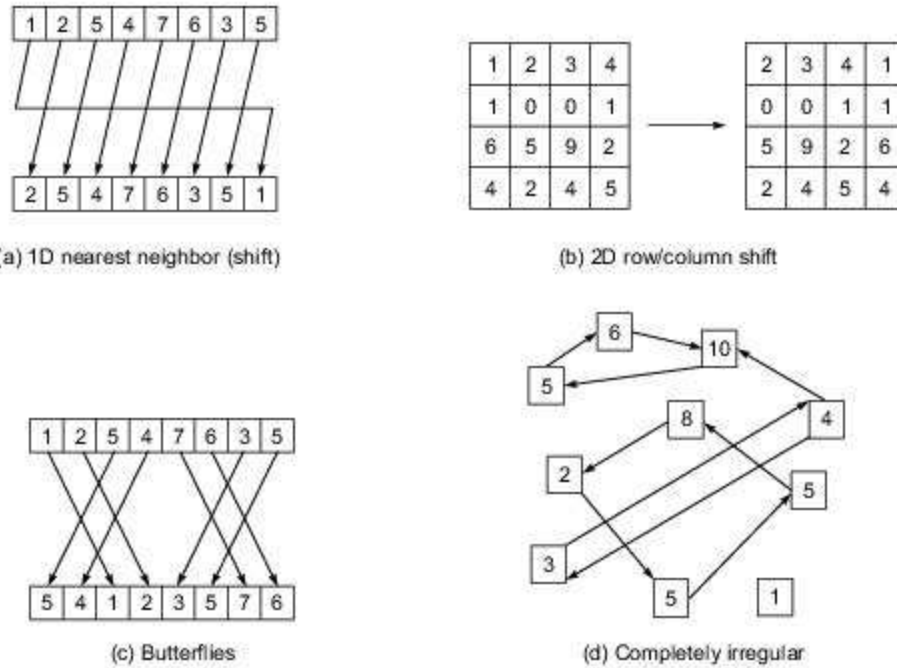


Fig. 8.35 Permutation operations for interprocessor communications on the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

Parallel Prefix This is a kind of combining operation supported by the control network. A *parallel prefix* operation delivers to the *i*th 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.

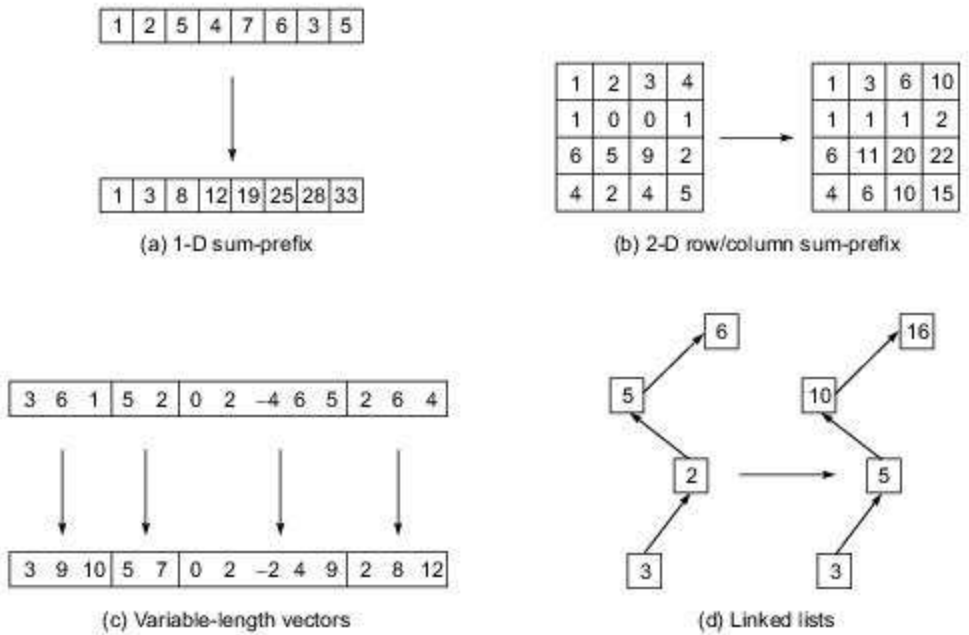


Fig. 8.36 Parallel prefix operations on the CM-5 (Courtesy of Thinking Machines Corporation, 1992)

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 $0 \le j \le 1023$ are column vectors; $\{a_j | 0 \le j \le 1023\}$ 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.

9.1 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.

The Architecture Environment The Dash architecture was a large-scale, cache-coherent, NUMA multiprocessor system, as depicted in Fig. 9.1. It consisted of multiple multiprocessor clusters connected through a scalable, low-latency interconnection network. Physical memory was distributed among the processing nodes in various clusters. The distributed memory formed a global address space.

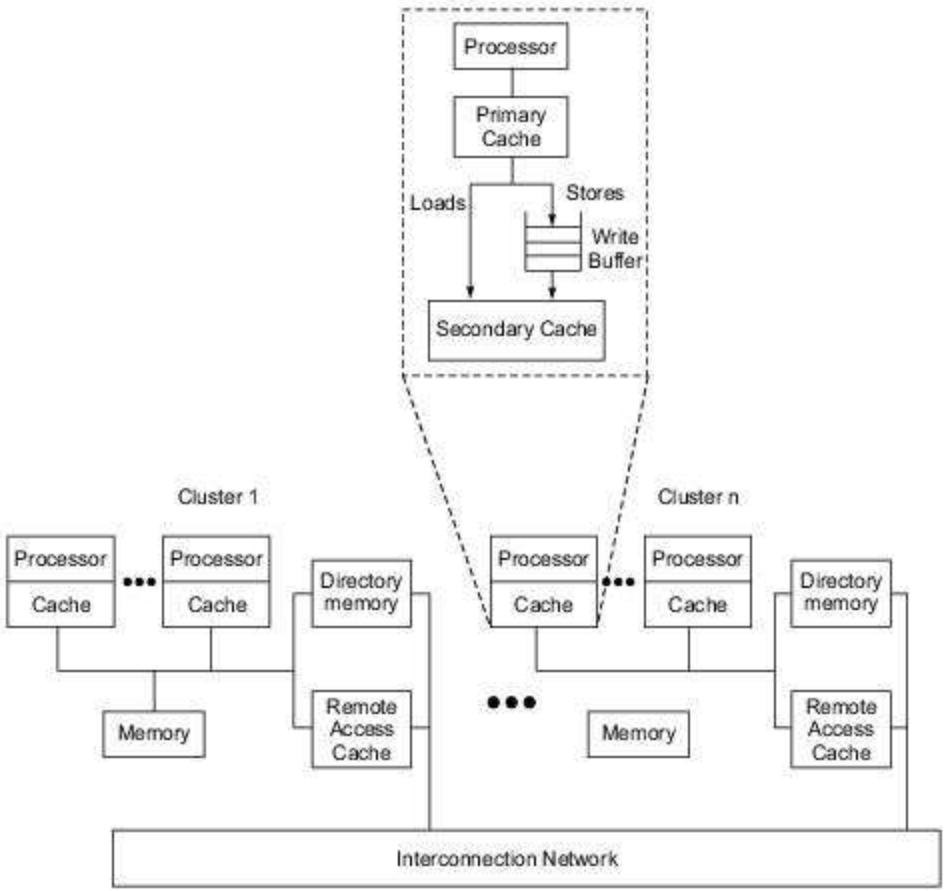


Fig. 9.1 A scalable coherent cache multiprocessor with distributed shared memory modeled after the Stanford Dash (Courtesy of Anoop Gupta et al, *Proc. 1991 Ann. Int. Symp. Computer Arch.*)

Cache coherence was maintained using an invalidating, distributed directory-based protocol (Section 7.2.3). For each memory block, the directory kept track of remote nodes cacheing it. When a write occurred, point-to-point messages were sent to invalidate remote copies of the block. Acknowledgment messages were used to inform the originating node when an invalidation was completed.

Two levels of local cache were used per processing node. Loads and writes were separated with the use of *write buffers* for implementing weaker memory consistency models. The main memory was shared by all processing nodes in the same cluster. To facilitate prefetching and the directory-based coherence protocol, directory memory and remote-access caches were used for each cluster. The remote-access cache was shared by all processors in the same cluster.

The SVM Concept Figure 9.2 shows the structure of a distributed shared memory. A global virtual address space is shared among processors residing at a large number of loosely coupled processing nodes. This *shared virtual memory* (SVM) concept was introduced in Section 4.4.1. Implementation and management issues of SVM are discussed below.

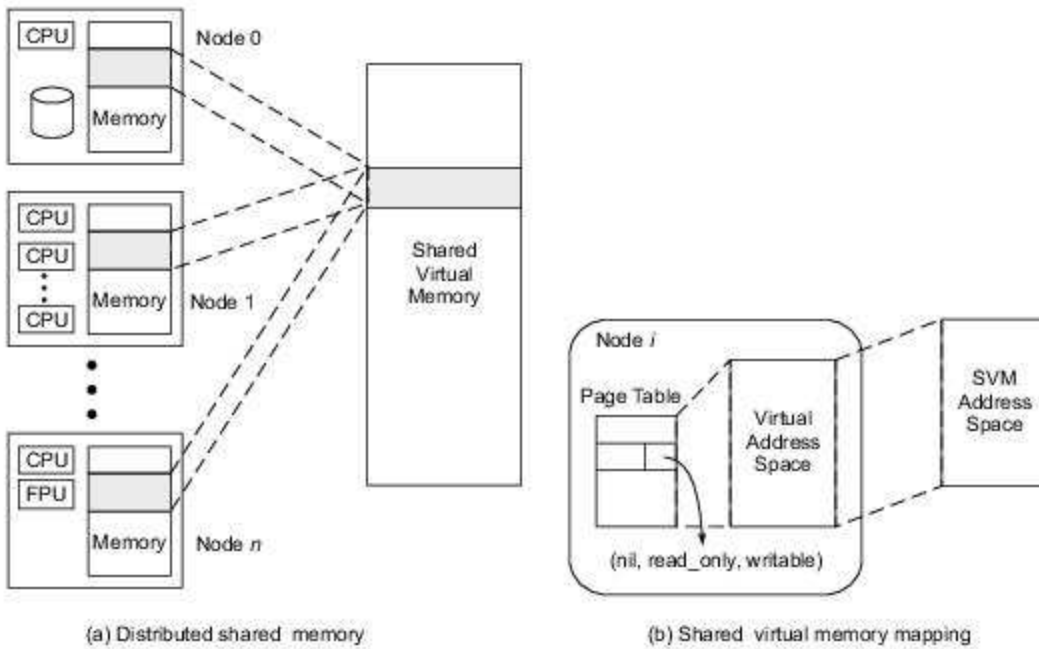


Fig. 9.2 The concept of distributed shared memory with a global virtual address space shared among all processors on loosely coupled processing nodes in a massively parallel architecture (Courtesy of Kai Li, 1992)

Shared virtual memory was first developed in a Ph.D. thesis by Li (1986) at Yale University. The idea is to implement coherent shared memory on a network of processors without physically shared memory. The coherent mapping of SVM on a message-passing multicomputer architecture is shown in Fig. 9.2b. The system uses virtual addresses instead of physical addresses for memory references.

Each virtual address space can be as large as a single node can provide and is shared by all nodes in the system. Li (1988) implemented the first SVM system, IVY, on a network of Apollo workstations. The SVM address space is organized in pages which can be accessed by any node in the system. A memory-mapping manager on each node views its local memory as a large cache of pages for its associated processor.

Page Swapping According to Kai Li (1992), pages that are marked read-only can have copies residing in the physical memories of other processors. A page currently being written may reside in only one local memory. When a processor writes a page that is also on other processors, it must update the page and then invalidate all copies on the other processors. Li described the page swapping as follows:

A memory reference causes a page fault when the page containing the memory location is not in a processor's local memory. When a page fault occurs, the memory manager retrieves the missing page from the memory of another processor. If there is a page frame available on the receiving node, the page is moved

in. Otherwise, the SVM system uses page replacement policies to find an available page frame, swapping its contents to the sending node.

A hardware MMU can set the access rights (*nil, read-only, writable*) so that a memory access violating memory coherence will cause a page fault. The memory coherence problem is solved in IVY through distributed fault handlers and their servers. To client programs, this mechanism is completely transparent.

The large virtual address space allows programs to be larger in code and data space than the physical memory on a single node. This SVM approach offers the ease of shared-variable programming in a message-passing environment. In addition, it improves software portability and enhances system scalability through modular memory growth.

Example SVM Systems Nitzberg and Lo (1991) conducted a survey of SVM research systems. Excerpted from their survey, descriptions of four representative SVM systems are summarized in Table 9.1. Dash implemented SVM with a directory-based coherence protocol. Linda offered a shared associative object memory with access functions. Plus used a write-update coherence protocol and performed replication only by program request. Shiva extended the IVY system for the Intel iPSC/2 hypercube. In using SVM systems, there exists a tendency to use large block (page) sizes as units of coherence. This tends to increase false-sharing activity.

Table 9.1 *Representative SVM Research Systems (Excerpts from Nitzberg and Lo, IEEE Comput., August 1991)*

System and Developer	*Implementation and Structure*	*Coherence Semantics and Protocols*	*Special Mechanics for Performance and Synchronization*
Stanford Dash (Lenoski, Laudon, Gharachorloo, Gupta, and Hennessy, 1988–).	Mesh-connected network of Silicon Graphics 4D/340 workstations with added hardware for coherent caches and prefetching.	Release memory consistency with write-invalidate protocol.	Relaxed coherence, prefetching, and queued locks for synchronization.
Yale Linda (Carriero and Gelernter, 1982–).	Software-implemented system based on the concepts of tuple space with access functions to achieve coherence via virtual memory management.	Coherence varied with environment; hashing used in associative search; no mutable data.	Linda could be implemented for many languages and machines using C-Linda or Fortran-Linda interfaces.
CMU Plus (Bisiani and Ravishankar, 1988–).	A hardware implementation using MC 88000, Caltech mesh, and Plus kernel.	Used processor consistency, nondemand write-update coherence, delayed operations.	Pages for sharing, words for coherence, complex synchronization instructions.
Princeton Shiva (Li and Schaefer, 1988).	Software-based system for Intel iPSC/2 with a Shiva/native operating system.	Sequential consistency, write-invalidate protocol, 4-Kbyte page swapping.	Used data structure compaction, messages for semaphores and signal-wait, distributed memory as backing store.

Scalability issues of SVM architectures include determining the sizes of data structures for maintaining memory coherence and how to take advantage of the fast data transmission among distributed memories in order to implement large SVM address spaces. Data structure compaction and page swapping can simplify the design of a large SVM address space without using disks as backing stores. A number of alternative choices are given in Li (1992).

9.1.2 Prefetching Techniques

Prefetching techniques are studied below. These involve both hardware and software approaches. Some benchmark results for prefetching on the Stanford Dash system are presented to illustrate the benefits.

Prefetching Techniques Prefetching uses knowledge about the expected misses in a program to move the corresponding data close to the processor before it is actually needed. Prefetching can be classified based on whether it is *binding* or *nonbinding*, and whether it is controlled by *hardware* or *software*.

With binding prefetching, the value of a later reference (e.g. a register load) is bound at the time when the prefetch completes. This places restrictions on when a binding prefetch can be issued, since the value will become stale if another processor modifies the same location during the interval between prefetch and reference. Binding prefetching may result in a significant loss in performance due to such limitations.

In contrast, nonbinding prefetching also brings the data close to the processor, but the data remains visible to the cache coherence protocol and is thus kept consistent until the processor actually reads the value.

Hardware-controlled prefetching includes schemes such as long cache lines and instruction lookahead. The effectiveness of long cache lines is limited by the reduced spatial locality in multiprocessor applications, while instruction lookahead is limited by branches and the finite lookahead buffer size.

With software-controlled prefetching, explicit prefetch instructions are issued. Software control allows the prefetching to be done selectively (thus reducing bandwidth requirements) and extends the possible interval between prefetch issue and actual reference, which is very important when latencies are large.

The disadvantages of software control include the extra instruction overhead required to generate the prefetches, as well as the need for sophisticated software intervention. In our study, we concentrate on *nonbinding software controlled prefetching*.

Benefits of Prefetching The benefits of prefetching come from several sources. The most obvious benefit occurs when a prefetch is issued early enough in the code so that the line is already in the cache by the time it is referenced. However, prefetching can improve performance even when this is not possible (e.g. when the address of a data structure cannot be determined until immediately before it is referenced). If multiple prefetches are issued back to back to fetch the data structure, the latency of all but the first prefetched reference can be hidden due to the pipelining of the memory accesses.

Prefetching offers another benefit in multiprocessors that use an ownership-based cache coherence protocol. If a cache block line is to be modified, prefetching it directly with ownership can significantly reduce the write latencies and the ensuing network traffic for obtaining ownership. Network traffic is reduced in read-modify-write instructions, since prefetching with ownership avoids first fetching a read-shared copy.

Benchmark Results Stanford researchers (Gupta, Hennessy, Gharachorloo, Mowry, and Weber, 1991) reported some benchmark results for evaluating various latency-hiding mechanisms. Benchmark programs included a particle-based three-dimensional simulator used in aeronautics (MP3D), an LU-decomposition program (LU), and a digital logic simulation program (PTHOR). The effect of prefetching is illustrated in Fig. 9.3 for running the MP3D code on a simulated Dash multiprocessor (Fig. 9.1).

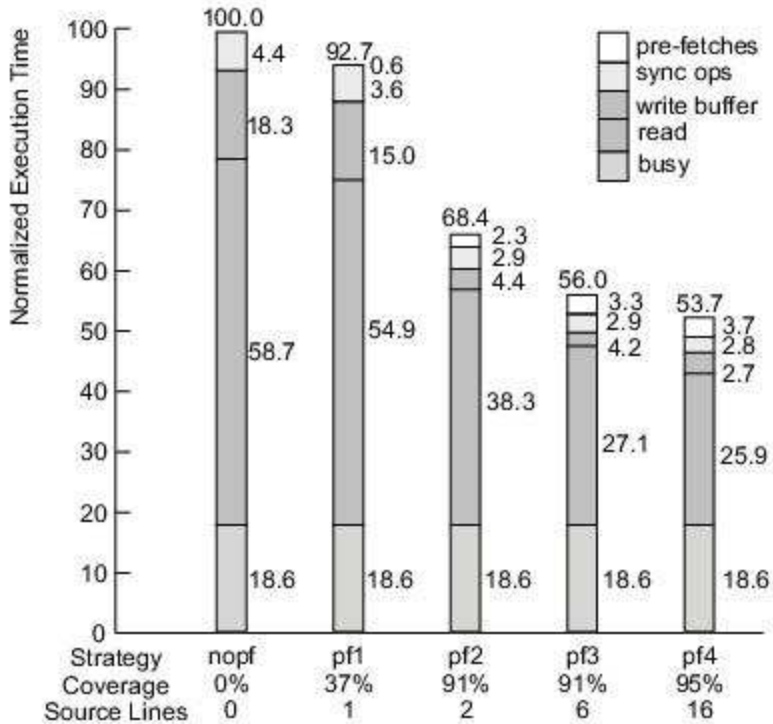


Fig. 9.3 Effect of various pre-fetching strategies for running the MP3D benchmark on a simulated Dash multiprocessor (Courtesy of Anoop Gupta et al, 1992)

The simulation runs involved 10,000 particles in a 64 × 8 × 8 space array with five time steps. Five prefetching strategies were tested (*nof*, *pf1*, *pf2*, *pf3*, and *pf4* in Fig. 9.3). These strategies range from no prefetching (*nopf*) to prefetching of the particle record in the same iteration or pipelined across increasing numbers of iterations (*pf1* through *pf4*). The bar diagrams in Fig. 9.3 show the execution times normalized with respect to the *nopf* strategy. Each bar shows a breakdown of the times required for prefetches, synchronization operations, using write buffers, reads, and busy in computing.

The end result was that prefetches were issued for up to 95% of the misses that occurred in the case without prefetching (referred to as the *coverage factor* in Fig. 9.3). Prefetching yielded significant time reduction in synchronization operations, using write buffers, and performing read operations. The best speedup achieved in Fig. 9.3 is 1.86, when the *pf4* prefetching strategy is compared with the *nopf* strategy. Still the prefetching benefits would be application-dependent. To introduce the pre-fetches in the MP3D code, only 16 lines of extra code were added to the source code.

9.1.3 Distributed Coherent Caches

While the coherence problem is easily solved for small bus-based multiprocessors through the use of snoopy cache coherence protocols, the problem is much more complicated for large-scale multiprocessors that use general interconnection networks. As a result, some large-scale multiprocessors did not provide caches (e.g. BBN Butterfly), others provided caches that must be kept coherent by software (e.g. IBM RP3), and still others provided full hardware support for coherent caches (e.g. Stanford Dash).

Dash Experience We evaluate the benefits when both private and shared read-write data are cacheable, as allowed by the Dash hardware coherent caches, versus the case where only private data are cacheable. Figure 9.4 presents a breakdown of the normalized execution times with and without cacheing of shared data for each of the applications. Private data are cached in both caches.

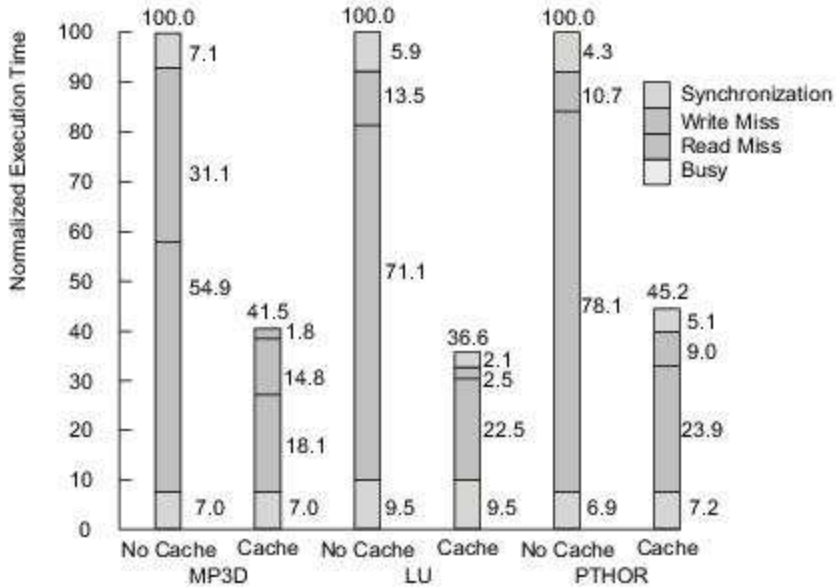


Fig. 9.4 Effect of cacheing shared data in simulated Dash benchmark experiments (Courtesy of Gupta et al, *Proc. Int. Symp. Comput. Archit.*, Toronto, Canada, May 1991)

The execution time of each application is normalized to the execution time of the case where shared data is not cached. The bottom section of each bar represents the busy time or useful cycles executed by the processor. The section above it represents the time that the processor is stalled waiting for reads. The section above that is the amount of time the processor is stalled waiting for writes to be completed. The top section, labeled "synchronization," accounts for the time processor is stalled due to locks and barriers.

Benefits of Cacheing As expected, the cacheing of shared read-write data provided substantial gains in performance, with benefits ranging from 2.2- to 2.7-fold improvement for the three Stanford benchmark programs. The largest benefit came from a reduction in the number of cycles wasted due to read misses. The cycles wasted due to write misses were also reduced, although the magnitude of the benefits varied across the three programs due to different write-hit ratios.

The cache-hit ratios achieved by MP3D, LU, and PTHOR were 80, 66, and 77%, respectively, for shared-read references, and 75, 97, and 47% for shared-write references. It is interesting to note that these hit ratios are substantially lower than the usual uniprocessor hit ratios.

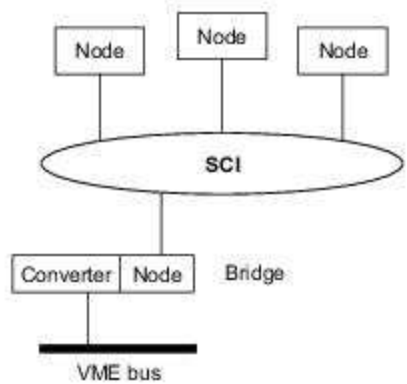
The low hit ratios arise from several factors: The data set size for engineering applications is large, parallelism decreases spatial locality in the application, and communication among processors results in invalidation misses. Still, hardware cache coherence is an effective technique for substantially increasing the performance with no assistance from the compiler or programmer.

9.1.4 Scalable Coherence Interface

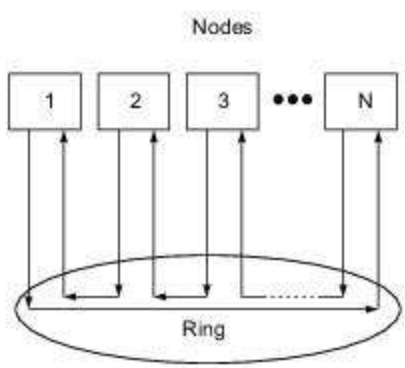
A scalable coherence interconnect structure with low latency is needed to extend from conventional bused backplanes to a fully duplex, point-to-point interface specification. The *scalable coherence interface* (SCI), which was introduced in Chapter 5, is specified in IEEE Standard 1596-1992. SCI supports unidirectional point-to-point connections, with two such links between each pair of nodes; packet-based communication is used, with routing.

Up to 64K processors, memory modules, or I/O nodes can effectively interface with a shared SCI interconnect. The cache coherence protocols used in SCI are directory-based. A sharing list is used to chain the distributed directories together for reference purposes.

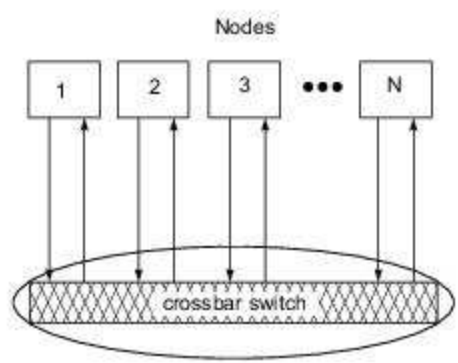
SCI Interconnect Models SCI defines the interface between nodes and the external interconnect, using 16-bit links with a bandwidth of up to 1 Gbyte/s per link. As a result, backplane buses have been replaced by unidirectional point-to-point links. A typical SCI configuration is shown in Fig. 9.5a. Each SCI node can be a processor with attached memory and I/O devices. The SCI interconnect can assume a ring structure or a crossbar switch as depicted in Figs. 9.5b and 9.5c, respectively, among other configurations.



(a) Typical SCI configuration with bridge to other bus



(b) A ring for point-to-point transactions



(b) A crossbar multiprocessor

Fig. 9.5 SCI interconnection configurations (Reprinted with permission from the IEEE Standard 1596-1992, copyright © 1992 by IEEE, Inc.)

Each node has an input link and an output link which are connected from or to the SCI ring or crossbar. The bandwidth of SCI links depends on the physical standard chosen to implement the links and interfaces.

In such an environment, the concept of broadcast bus-based transactions is abandoned. Coherence protocols are based on point-to-point transactions initiated by a requester and completed by a responder. A ring interconnect provides the simplest feedback connections among the nodes.

The converter in Fig. 9.5a is used to bridge the SCI ring to the VME bus as shown. A mesh of rings can also be considered using some bridging modules. The bandwidth, arbitration, and addressing mechanisms of an SCI ring significantly outperform backplane buses. By eliminating the snoopy cache controllers, the SCI is also less expensive per node, but the main advantage lies in its low latency and scalability.

Although SCI is scalable, the amount of memory used in the cache directories also scales up well. The performance of the SCI protocol does not scale, since when the sharing list is long, invalidations take proportionately longer time.

Sharing-List Structures Sharing lists are used in SCI to build chained directories for cache coherence use. The length of the sharing lists is effectively unbounded. Sharing lists are dynamically created, pruned, and destroyed. Each coherently cached block is entered onto a list of processors sharing the block.

Processors have the option of bypassing the coherence protocols for locally cached data. Cache blocks of 64 bytes are assumed. By distributing the directories among the sharing processors, SCI avoids scaling limitations imposed by using a central directory. Communications among sharing processors are supported by heavily shared memory controllers, as shown in Fig. 9.6.

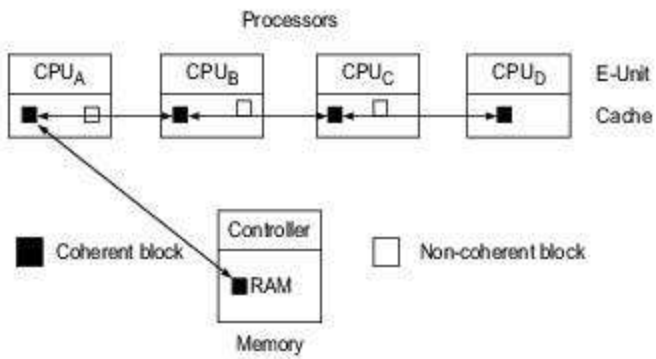


Fig. 9.6 SCI cache coherence protocol with distributed directories (Courtesy of D.V. James et al, *IEEE Computer*, 1990)

Other blocks may be locally cached and are not visible to the coherence protocols. For every block address, the memory and cache entries have additional tag bits which are used to identify the first processor (head) in the sharing list and to link the previous and following nodes.

Doubly linked lists are maintained between processors in the sharing list, with forward and backward pointers as shown by the double arrows in each link. Noncoherent copies may also be made coherent by page-level control. However, such higher-level software coherence protocols are beyond the scope of the SCI standard.

Sharing-List Creation The states of the sharing list are defined by the state of the memory and the states of list entries. Normally, the shared memory is either in a home (uncached) or a cached (sharing-list) state. The sharing-list entries specify the location of the entry in a multiple-entry sharing list, identify the only entry in the list, or specify the entry's cache properties, such as clean, dirty, valid, or stale.

The head processor is always responsible for list management. The stable and legal combinations of the memory and entry states can specify uncached data, clean or dirty data at various locations, and cached writable or stale data.

The memory is initially in the home state (uncached), and all cache copies are invalid. Sharing-list creation begins at the cache where an entry is changed from an invalid to a pending state. When a read-cache transaction is directed from a processor to the memory controller, the memory state is changed from uncached to cached and the requested data is returned.

The requester's cache entry state is then changed from a pending state to an only-clean state. Sharing-list creation is illustrated in Fig. 9.7a. Multiple requests can be simultaneously generated, but they are processed sequentially by the memory controller.

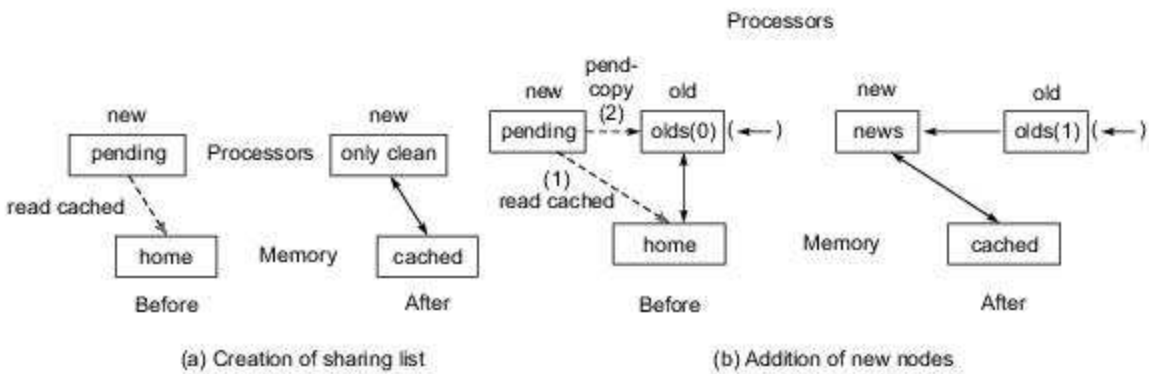


Fig. 9.7 Sharing-list creation and update examples (Courtesy of D.V. James et al, *IEEE Computer*, 1990)

Sharing-List Updates For subsequent memory access, the memory state is cached, and the cache head of the sharing list has possibly dirty data. As illustrated in Fig. 9.7b, a new requester (cache A) first directs its read-cache transaction to memory but receives a pointer to cache B instead of the requested data.

A second cache-to-cache transaction, called *prepend*, is directed from cache A to cache B. Cache B then sets its backward pointer to point to cache A and returns the requested data. The dashed lines correspond to transactions between a processor and memory or another processor. The solid lines are sharing-list pointers.

After the transaction, the inserted cache A becomes the new head, and the old head, cache B, is in the middle as shown by the new sharing list on the right in Fig. 9.7b.

Any sharing-list entry may delete itself from the list. Details of entry deletions are left as an exercise for the reader. Simultaneous deletions never generate deadlocks or starvation. However, the addition of new sharing-list entries must be performed in first-in-first-out order in order to avoid potential deadlocking dependences.

The head of the sharing list has the authority to purge other entries from the list to obtain an exclusive entry. Others may reenter as a new list head. Purges are performed sequentially. The chained-directory coherence protocols are fault-tolerant in that dirty data is never lost when transactions are discarded.

Implementation Issues SCI was developed to support multiprocessor systems with thousands of processors by providing a coherent distributed-cache image of distributed shared memory and bridges that interface with existing or future buses. It can support various multiprocessor topologies using Omega or crossbar networks.

Differential emitter coupled logic (ECL) signaling works well at SCI clock rates. The original SCI implementation uses a 16-bit data path at 2 ns per word. The interface is synchronously clocked. Several models of clock distribution are supported. With distributed shared-memory and distributed cache coherence protocols, the boundary between multiprocessors and multicomputers has become blurred in MIMD systems of this class.

9.1.5 Relaxed Memory Consistency

We have studied *weak consistency* (WC) (Sindhu et al, 1992) and *sequential consistency* (SC) in Section 5.4. Two additional memory models are introduced below for building scalable multiprocessors with distributed shared memory.

Processor Consistency Goodman (1989) introduced the *processor consistency* (PC) model in which writes issued by each individual processor are always in program order. However, the order of writes from two different processors can be out of program order. In other words, consistency in writes is observed in each processor, but the order of reads from each processor is not restricted as long as they do not involve other processors.

The PC model relaxes from the SC model by removing some restrictions on writes from different processors. This opens up more opportunities for write buffering and pipelining. Two conditions related to other processors are required for ensuring processor consistency:

(1) Before a *read* is allowed to perform with respect to any other processor, all previous *read* accesses must be performed.
(2) Before a write is allowed to perform with respect to any other processor, all previous *read* or *write* accesses must be performed.

These conditions allow *reads* following a *write* to bypass the *write*. To avoid deadlock, the implementation should guarantee that a *write* that appears previously in program order will eventually be performed.

Release Consistency One of the most relaxed memory models is the *release consistency* (RC) model introduced by Gharachorloo et al (1990). Release consistency requires that synchronization accesses in the program be identified and classified as either *acquires* (e.g. locks) or *releases* (e.g. unlocks). An acquire is a read operation (which can be part of a read-modify-write) that gains permission to access a set of data, while a release is a write operation that gives away such permission. This information is used to provide flexibility in buffering and pipelining of accesses between synchronization points.

The main advantage of the relaxed models is the potential for increased performance by hiding as much write latency as possible. The main disadvantage is increased hardware complexity and a more complex programming model. Three conditions ensure release consistency:

(1) Before an ordinary *read* or *write* access is allowed to perform with respect to any other processor, all previous *acquire* accesses must be performed.

(2) Before a *release* access is allowed to perform with respect to any other processor, all previous ordinary *read* and *store* accesses must be performed.

(3) *Special accesses* are processor-consistent with one another. The ordering restrictions imposed by weak consistency are not present in release consistency. Instead, release consistency requires processor consistency and not sequential consistency.

Release consistency can be satisfied by (i) stalling the processor on an acquire access until it completes, and (ii) delaying the completion of release access until all previous memory accesses complete. Intuitive definitions of the four memory consistency models, the SC, WC, PC, and RC, are summarized in Fig. 9.8.

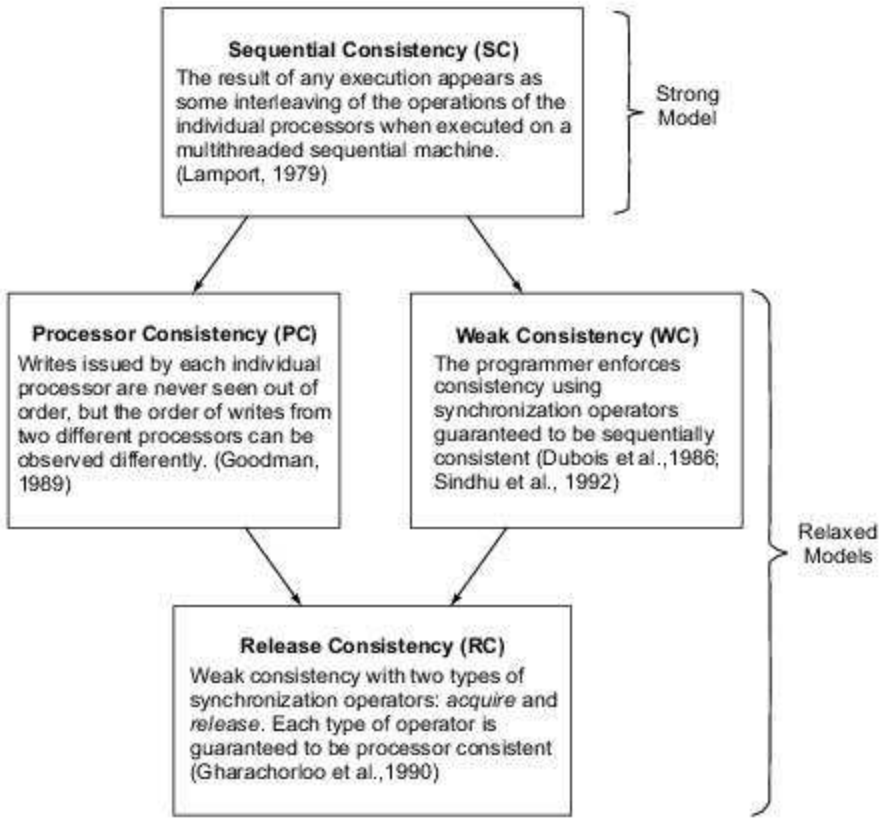


Fig. 9.8 Intuitive definitions of four memory consistency models. The arrows point from strong to relaxed consistencies (Courtesy of Nitzberg and Lo, *IEEE Computer*, August 1991)

The cost of implementing RC over that for SC arises from the extra hardware cost of providing a lockup-free cache and keeping track of multiple outstanding requests. Although this cost is not negligible, the same hardware features are also required to support prefetching and multiple contexts.

Effect of Release Consistency Figure 9.9 presents the breakdown of execution times under SC and RC for the three applications. The execution times are normalized to those shown in Fig. 9.3 with shared data cached. As can be seen from the results, RC removes all idle time due to write-miss latency.

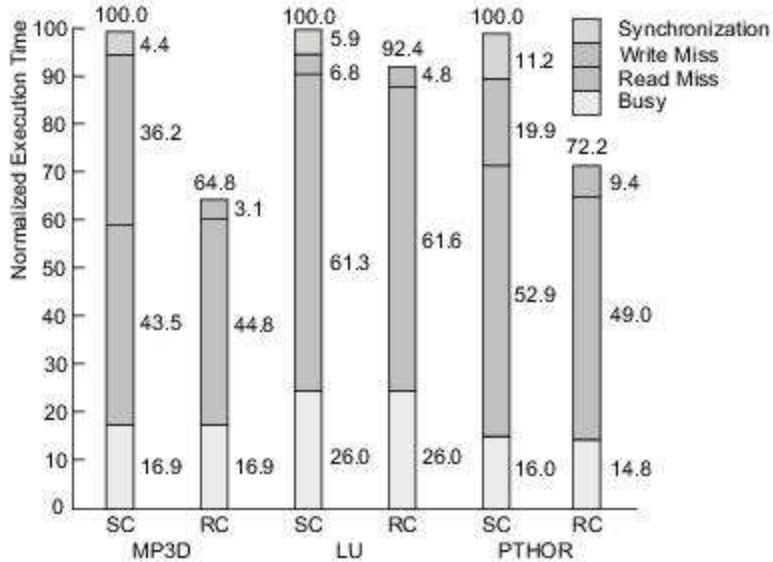


Fig. 9.9 Effect of relaxing the shared-memory model from sequential consistency (SC) to release consistency (RC) (Courtesy of Gupta et al, *Proc. Int. Symp. Comput. Archit.*, Toronto, Canada, May 1991)

The gains are large in MP3D and PTHOR since the write-miss time constitutes a large portion of the execution time under SC (35 and 20%, respectively), while the gain is small in LU due to the relatively small write-miss time under SC (7%).

Effect of Combining Mechanisms The effect of combining various latency-hiding mechanisms is illustrated by Fig. 9.10 based on the MP3D benchmark results obtained at Stanford University. The idea of using *multiple-context* processors will be described in Section 9.2. However, the effect of integrating MC with other latency-hiding mechanisms is presented below.

The busy parts of the execution times in Fig. 9.10 are equal in all combinations. This is the CPU busy time for executing the MP3D program. The idle part in the bar diagram corresponds to memory latency and includes all cache-miss penalties. All the times are normalized with respect to the execution time (100 units) required in a *cache-coherent* system. The leftmost time bar (with 241 units) corresponds to the worst case of using a private cache exclusively without shared reads or writes. Long overhead is experienced in this case due to excessive cache misses. The use of a cache-coherent system shows a 2.41-fold improvement over the private case. All the remaining cases are assumed to use hardware coherent caches.

The use of *release consistency* shows a 35% further improvement over the coherent system. The adding of prefetching reduces the time further to 44 units. The best case is the combination of using coherent caches, RC, and *multiple contexts* (MC). The rightmost time bar is obtained from applying all four mechanisms. The combined results show an overall speedup of 4 to 7 over the case of using private caches.

The above and other uncited benchmark results reported at Stanford suggest that a coherent cache and relaxed consistency uniformly improve performance. The improvements due to prefetching and multiple

contexts are sizable but are much more application-dependent. Combinations of the various latency-hiding mechanisms generally attain a better performance than each one on its own.

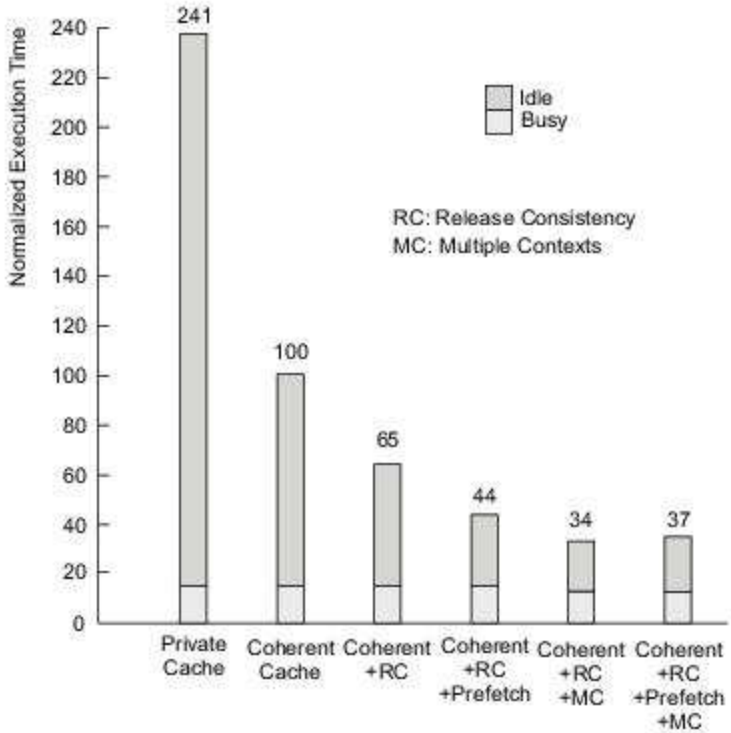


Fig. 9.10 Effect of combining various latency-hiding mechanisms from the MP3D benchmark on a simulated Dash multiprocessor (Courtesy of Gupta, 1992)

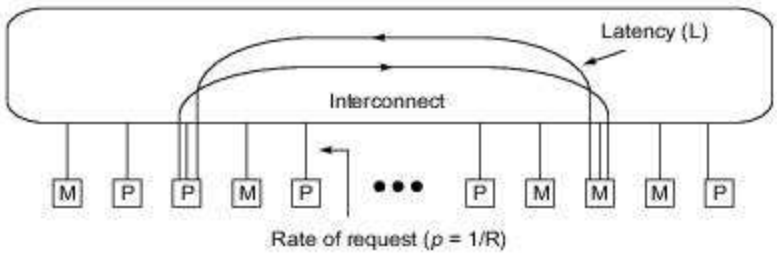
9.2 PRINCIPLES OF MULTITHREADING

This section considers multithreaded processors and multidimensional system architectures. Only control-flow approaches are described here. Fine-grain machines are studied in Section 9.3, von Neumann multithreading in Section 9.4, and dataflow multithreading in Section 9.5. Recent developments in multithreading support by processor hardware are discussed in Chapters 12 and 13.

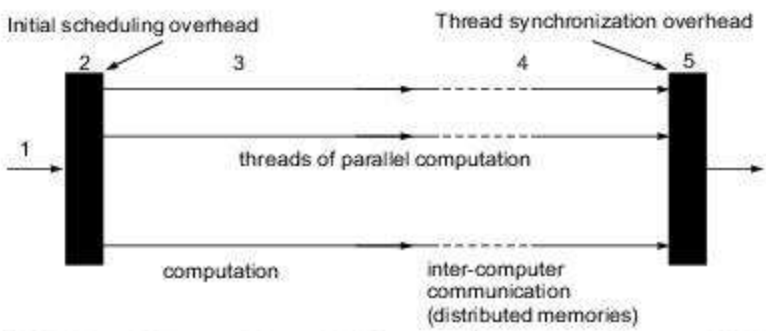
9.2.1 Multithreading Issues and Solutions

Multithreading demands that the processor be designed to handle multiple contexts simultaneously on a context-switching basis. We first specify the typical architecture environment using multiple-context processors. Next we present a multithreaded computation model. Then we look further into the latency and synchronization problems and discuss their solutions in this environment.

Architecture Environment One possible multithreaded MPP system is modeled by a network of processor (P) and memory (M) nodes as depicted in Fig. 9.11a. The distributed memories form a global address space. Four machine parameters are defined below to analyze the performance of this network:



(a) The architecture environment. (Courtesy of Rafael Saavedra, 1992)



(b) Multithreaded computation model. (Courtesy of Gordon Bell, *Commun. ACM*, August 1992)

Fig. 9.11 Multithreaded architecture and its computation model for a massively parallel processing system

(1) *The latency* (L): This is the communication latency on a remote memory access. The value of L includes the network delays, cache-miss penalty, and delays caused by contentions in split transactions.

(2) *The number of threads* (N): This is the number of threads that can be interleaved in each processor. A *thread* is represented by a *context* consisting of a program counter, a register set, and the required context status words.

(3) *The context-switching overhead* (C): This refers to the cycles lost in performing context switching in a processor. This time depends on the switch mechanism and the amount of processor states devoted to maintaining active threads.

(4) *The interval between switches* (R): This refers to the cycles between switches triggered by remote reference. The inverse $p = 1/R$ is called the *rate of requests* for remote accesses. This reflects a combination of program behavior and memory system design.

In order to increase efficiency, one approach is to reduce the rate of requests by using distributed coherent caches. Another is to eliminate processor waiting through multithreading. The basic concept of multithreading is described below.

Multithreaded Computations Bell (1992) has described the structure of the multithreaded parallel computations model shown in Fig. 9.11b. The computation starts with a sequential thread (1), followed

by supervisory scheduling (2) where the processors begin threads of computation (3), by intercomputer messages that update variables among the nodes when the computer has a distributed memory (4), and finally by synchronization prior to beginning the next unit of parallel work (5).

The communication overhead period (4) inherent in distributed memory structures is usually distributed throughout the computation and is possibly completely overlapped. Message-passing overhead (send and receive calls) in multicomputers can be reduced by specialized hardware operating in parallel with computation.

Communication bandwidth limits granularity, since a certain amount of data has to be transferred with other nodes in order to complete a computational grain. Message-passing calls (4) and synchronization (5) are nonproductive. Fast mechanisms to reduce or to hide these delays are therefore needed. Multithreading is not capable of speedup in the execution of single threads, while weak ordering or relaxed consistency models are capable of doing this.

Problems of Asynchrony Massively parallel processors operate asynchronously in a network environment. The asynchrony triggers two fundamental latency problems: *remote loads* and *synchronizing loads*, as observed by Nikhil (1992). These two problems are explained by the following example:

Example 9.1 Latency problems for remote loads or synchronizing loads (Rishiyun Nikhil, 1992).

The remote load situation is illustrated in Fig. 9.12a. Variables A and B are located on nodes N2 and N3, respectively. They need to be brought to node N1 to compute the difference $A - B$ in variable C. The basic computation demands the execution of two remote loads (rload) and then the subtraction.

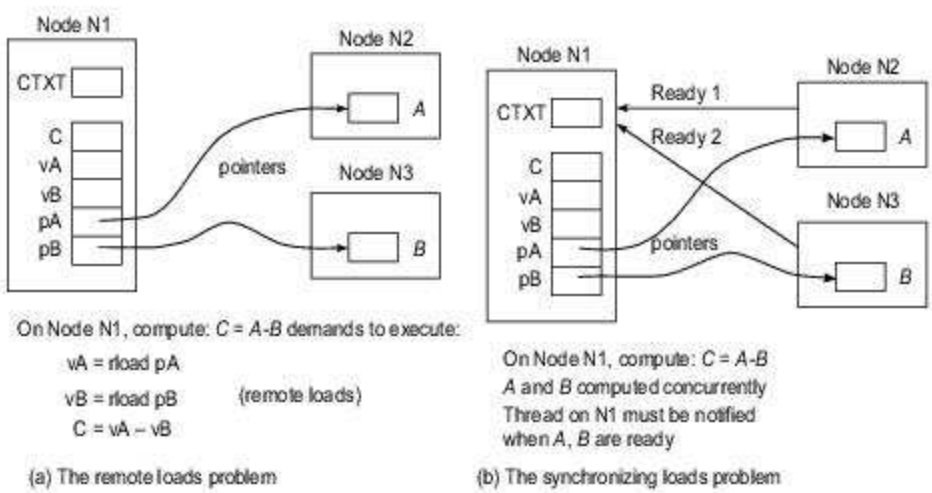


Fig. 9.12 Two common problems caused by asynchrony and communication latency in massively parallel processors (Courtesy of R.S. Nikhil, Digital Equipment Corporation, 1992)

Let pA and pB be the pointers to A and B, respectively. The two rloads can be issued from the same thread or from two different threads. The *context* of the computation on N1 is represented by the variable CTXT. It can be a stack pointer, a frame pointer, a current-object pointer, a process identifier, etc. In general, variable names like vA, vB, and C are interpreted relative to CTXT.

In Fig. 9.12b, the idling due to synchronizing loads is illustrated. In this case, A and B are computed by concurrent processes, and we are not sure exactly when they will be ready for node N1 to read. The ready signals (Ready1 and Ready2) may reach node N1 asynchronously. This is a typical situation in the producer-consumer problem. Busy-waiting may result.

The key issue involved in remote loads is how to avoid idling in node N1 during the load operations. The latency caused by remote loads is an architectural property. The latency caused by synchronizing loads also depends on scheduling and the time it takes to compute A and B, which may be much longer than the transit latency. The synchronization latency is often unpredictable, while the remote-load latencies are often predictable.

Multithreading Solutions This solution to asynchrony problems is to multiplex among many threads: When one thread issues a remote-load request, the processor begins work on another thread, and so on (Fig. 9.13a). Clearly, the cost of thread switching should be much smaller than that of the latency of the remote load, or else the processor might as well wait for the remote load's response.

As the internode latency increases, more threads are needed to hide it effectively. Another concern is to make sure that messages carry continuations. Suppose, after issuing a remote load from thread T_1 (Fig. 9.13a), we switch to thread T_2, which also issues a remote load. The responses may not return in the same order. This may be caused by requests traveling different distances, through varying degrees of congestion, to destination nodes whose loads differ greatly, etc.

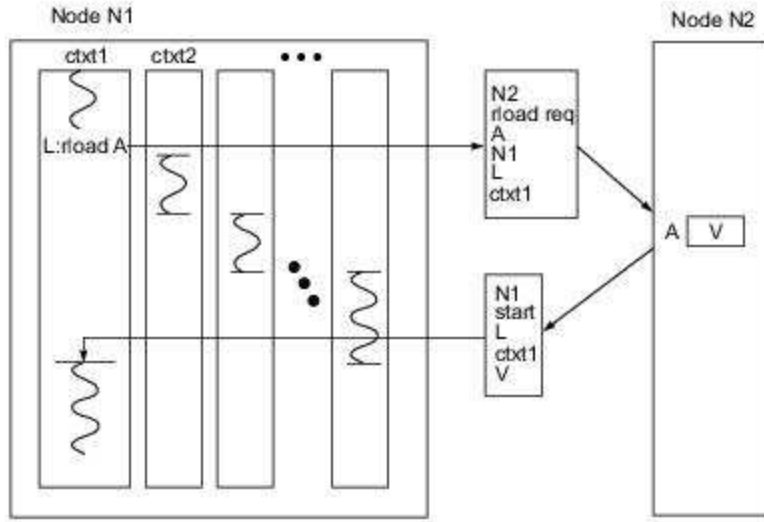
One way to cope with the problem is to associate each remote load and response with an identifier for the appropriate thread, so that it can be reenabled on the arrival of a response. These thread identifiers are referred to as *continuations* on messages. A large *continuation name space* should be provided to name an adequate number of threads waiting for remote responses.

The size of the hardware-supported continuation in a name space varies greatly in different system designs: from 1 in the Dash, 4 in the Alewife, 64 in the HEP, and 1024 in the Tera (Section 9.4) to the local memory address space in the Monsoon, Hybrid Dataflow/von Neumann, MDP (Section 9.3), and *T (Section 9.5). Of course, if the hardware-supported name space is small, one can always virtualize it by multiplexing in software, but this has an associated overhead.

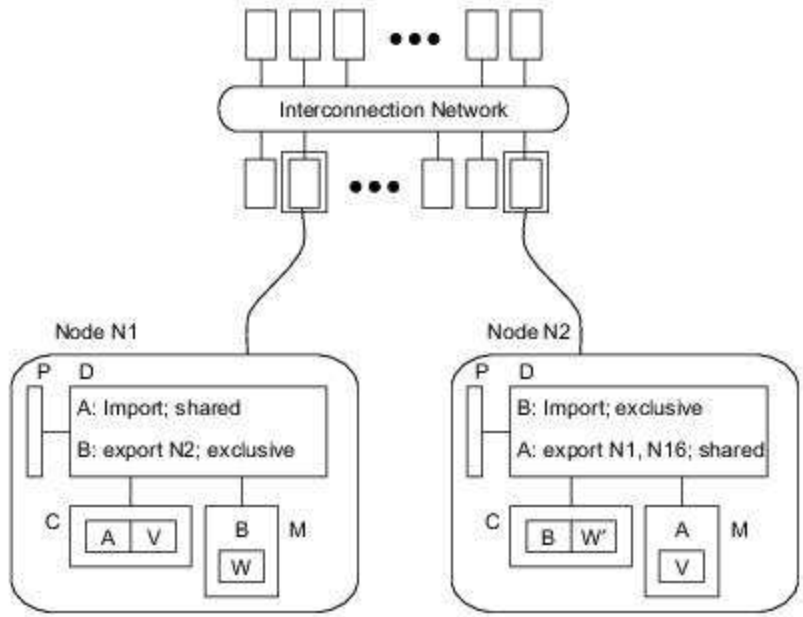
Distributed Cacheing The concept of distributed cacheing is shown in Fig. 9.13b. Every memory location has an owner node. For example, N1 owns B and N2 owns A. The directories are used to contain import-export lists and state whether the data is *shared* (for reads, many caches may hold copies) or *exclusive* (for writes, one cache holds the current value).

The directories multiplex among a small number of contexts to cover the cache loading effects. The MIT Alewife, KSR-1, and Stanford Dash have implemented directory-based coherence protocols. It should be noted that distributed cacheing offers a solution for the remote-loads problem, but not for the synchronizing-

loads problem. Multithreading offers a solution for remote loads and possibly for synchronizing loads. However, the two approaches can be combined to solve both types of remote-access problems.



(a) Multithreading solution



P = Processor; D = Directory; C = Cache; M = Memory

(b) Distributed cacheing

Fig. 9.13 Two solutions for overcoming the asynchrony problems (Courtesy of R. S. Nikhil, Digital Equipment Corporation, 1992)

9.2.2 Multiple-Context Processors

Multithreaded systems are constructed with *multiple-context* (or *multithreaded)* processors. In this section, we study an abstract model based on the work of Saavedra et al (1990). We then present an example of this type of processor. We discuss the processor efficiency issue as a function of memory latency (L), the number of contexts (N), and context-switching overhead (C).

The Enhanced Processor Model A conventional single-thread processor will *wait* during a remote reference, so we may say it is idle for a period of time L. A multithreaded processor, as modeled in Fig. 9.14a, will suspend the current context and switch to another, so after some fixed number of cycles it will again be busy doing useful work, even though the remote reference is outstanding. Only if all the contexts are suspended (blocked) will the processor be idle.

Clearly, the objective is to maximize the fraction of time that the processor is busy, so we will use the *efficiency* of the processor as our performance index, given by

$$Efficiency = \frac{busy}{busy + switching + idle} \tag{9.1}$$

where *busy*, *switching*, and *idle* represent the amount of time, measured over some large interval, that the processor is in the corresponding state. The basic idea behind a multithreaded machine is to interleave the execution of several contexts in order to dramatically reduce the value of *idle*, but without overly increasing the magnitude of *switching*.

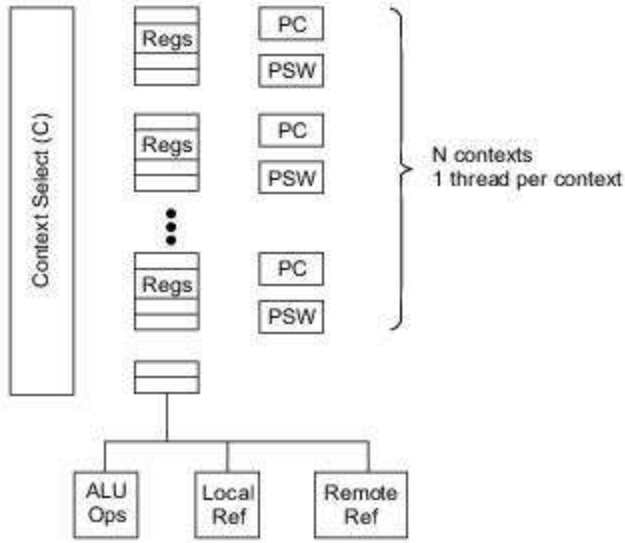
The state of a processor is determined by the disposition of the various contexts on the processor. During its lifetime, a context cycles through the following states: *ready*, *running*, *leaving*, and *blocked*. There can be at most one context running or leaving. A processor is *busy* if there is a context in the running state; it is *switching* while making the transition from one context to another, i.e. when a context is leaving. Otherwise, all contexts are blocked and we say the processor is *idle*.

A running context keeps the processor busy until it issues an operation that requires a context switch. The context then spends C cycles in the *leaving* state, then goes into the *blocked* state for L cycles, and finally reenters the *ready* state. Eventually the processor will choose it and the cycle will start again.

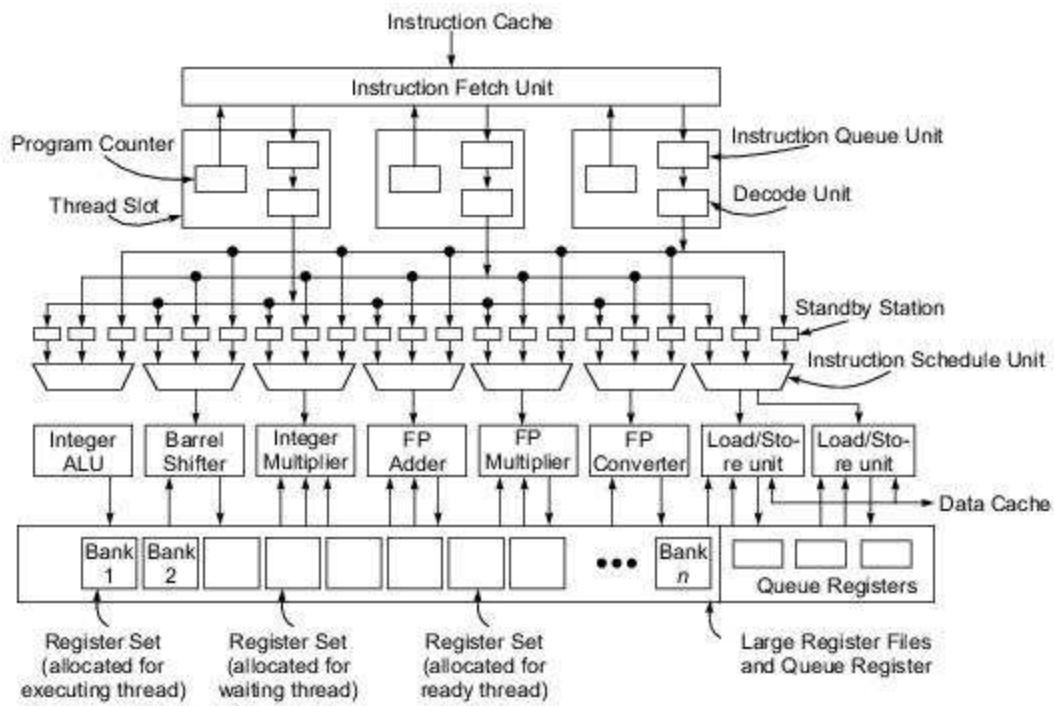
The abstract model shown in Fig. 9.14a assumes one thread per context, and each context is represented by its own program counter (PC), register set, and process status word (PSW). An example multithreaded processor in which three thread slots ($N = 3$) are provided is shown in Fig. 9.14b.

Example 9.2 A multithreaded processor with three thread slots (Hiroaki Hirata et al., 1992).

As shown in Fig. 9.14b, the processor is provided with several instruction queue unit and decode unit pairs, called *thread slots*. Each thread slot, associated with a program counter, makes up a *logical processor*, while an instruction fetch unit and all functional units are physically shared among logical processors.



(a) Multithreaded model. (Courtesy of Rafael Saavedra, 1992)



(b) A three-thread processor example (Courtesy of H. Hirata et al, *Proc 19th Int. Symp. Comput. Archit.*, Australia, May 1992)

Fig. 9.14 Multiple-context processor model and an example design

An instruction queue unit has a buffer which saves some instructions succeeding the instruction indicated by the program counter. The buffer size needs to be at least $B = N \times C$ words, where N is the number of thread slots and C is the number of cycles required to access the instruction cache.

An instruction fetch unit fetches at most B instructions for one thread every C cycles from the instruction cache and attempts to fill the buffers in the instruction queue unit. This fetching operation is done in an interleaved fashion for multiple threads. So, on the average, the buffer in one instruction queue unit is filled once in B cycles.

When one of the threads encounters a branch instruction, however, that thread can preempt the prefetching operation. The instruction cache and fetch unit might become a bottleneck for a processor with many thread slots. In such cases, a bigger and/or faster cache and another fetch unit would be needed.

Context-Switching Policies Different multithreaded architectures are distinguished by the context-switching policies adopted. Specified below are four switching policies:

(1) *Switch on cache miss*—This policy corresponds to the case where a context is preempted when it causes a cache miss. In this case, R is taken to be the average interval between misses (in cycles), and L the time required to satisfy the miss. Here, the processor switches contexts only when it is certain that the current one will be delayed for a significant number of cycles.

(2) *Switch on every load*—This policy allows switching on every load, independent of whether it will cause a miss or not. In this case, R represents the average interval between loads. A general multithreading model assumes that a context is blocked for L cycles after every switch; but in the case of a switch-on-load processor, this happens only if the load causes a cache miss.

The general model can be employed if it is postulated that there are two sources of latency (L_1 and L_2), each having a particular probability (p_1 and p_2) of occurring on every switch. If L_1 represents the latency on a cache miss, then p_1 corresponds to what is normally referred to as the miss ratio. L_2 is a zero-cycle memory latency with probability p_2.

(3) *Switch on every instruction*—This policy allows switching on every instruction, independent of whether it is a load or not. In other words, it interleaves the instructions from different threads on a cycle-by-cycle basis. Successive instructions become independent, which will benefit pipelined execution. However, the cache miss may increase due to breaking of locality. It has been verified by some trace-driven experiments at Stanford that cycle-by-cycle interleaving of contexts provides a performance advantage over switching on a cache miss in that the context interleaving could hide pipeline dependences and reduce the context switch cost.

(4) *Switch on block of instruction*—Blocks of instructions from different threads are interleaved. This will improve the cache-hit ratio due to locality. It will also benefit single-context performance.

Processor Efficiencies A single-thread processor executes a context until a remote reference is issued (R cycles) and then is idle until the reference completes (L cycles). There is no context switch and obviously no switch overhead. We can model this behavior as an alternating renewal process having a cycle of $R + L$. In terms of Eq. 9.1, R and L correspond to the amount of time during a cycle that the processor is *busy* and *idle*, respectively. Thus the efficiency of a single-threaded machine is given by

$$E_1 = \frac{R}{R+L} = \frac{1}{1+L/R} \tag{9.2}$$

This shows clearly the performance degradation of such a processor in a parallel system with a large memory latency.

With multiple contexts, memory latency can be hidden by switching to a new context, but we assume that the switch takes C cycles of overhead. Assuming the run length between switches is constant with a sufficient number of contexts, there is always a context ready to execute when a switch occurs, so the processor is never idle. The processor efficiency is analyzed below under two different conditions as illustrated in Fig. 9.15.

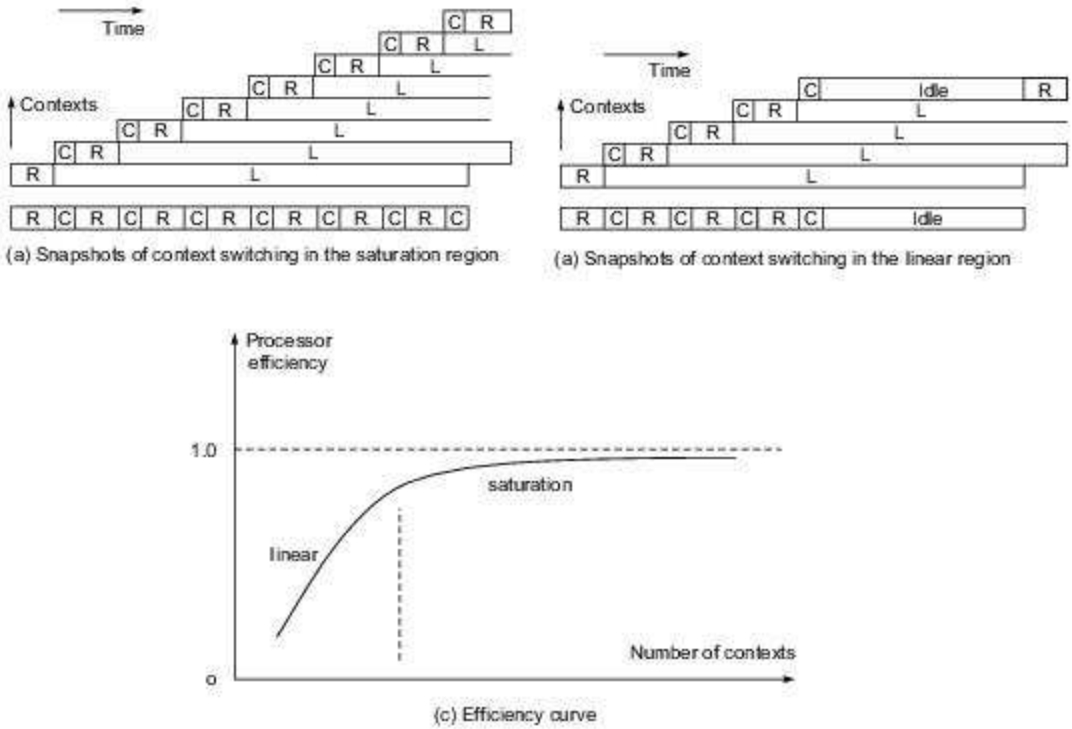


Fig. 9.15 Context switching and processor efficiency as a function of the number of contexts (Courtesy of Rafael Saavedra, 1992)

(1) *Saturation region*—In this saturated region, the processor operates with maximum utilization. The cycle of the renewal process in this case is $R + C$, and the efficiency is simply

$$E_{sat} = \frac{R}{R+C} = \frac{1}{1+C/R} \tag{9.3}$$

Observe that the efficiency in saturation is independent of the latency and also does not change with a further increase in the number of contexts.

Saturation is achieved when the time the processor spends servicing the other threads exceeds the time required to process a request, i.e., when $(N-1)(R+C) > L$. This gives the saturation point, under constant run length, as

$$N_d = \frac{L}{R+C} + 1 \tag{9.4}$$

(2) *Linear region*—When the number of contexts is below the saturation point, there may be no ready contexts after a context switch, so the processor will experience idle cycles. The time required to switch to a ready context, execute it until a remote reference is issued, and process the reference is equal to $R + C + L$. Assuming N is below the saturation point, during this time all the other contexts have a turn in the processor. Thus, the efficiency is given by

$$E_{lin} = \frac{NR}{R+C+L} \tag{9.5}$$

Observe that the efficiency increases linearly with the number of contexts until the saturation point is reached and beyond that remains constant. The equation for E_{sat} gives the fundamental limit on the efficiency of a multithreaded processor and underlines the importance of the ratio C/R. Unless the context switch is extremely cheap, the remote reference rate must be kept low.

Figures 9.15a and 9.15b show snapshots of context switching in the saturation and linear regions, respectively. The processor efficiency is plotted as a function of the number of contexts in Fig. 9.15c.

In Fig. 9.16, the processor efficiency is plotted as a function of the memory latency L with an average run length $R = 16$ cycles. The $C = 0$ curve corresponds to zero switching overhead. With $C = 16$ cycles, about 50% efficiency can be achieved. These results are based on a Markov model of multithreaded architecture by Saavedra (1992). It should be noted that multithreading increases both processor efficiency and network traffic. Tradeoffs do exist between these two opposing goals, and this has been discussed in a paper by Agarwal (1992).

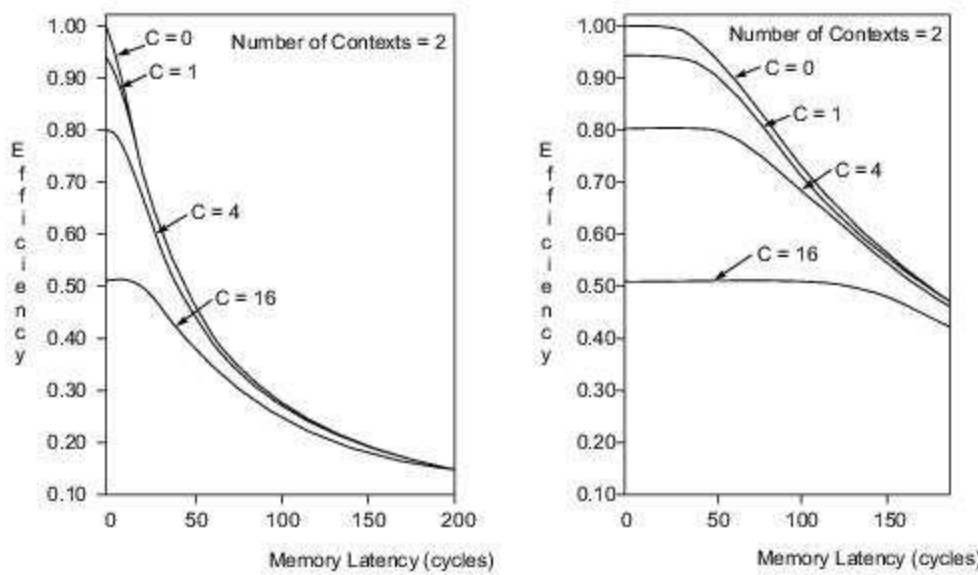


Fig. 9.16 Processor efficiency of a multithreaded architecture (Courtesy of R. Saavedra, D. E. Culler, and T. von Eicken, 1992)

9.2.3 Multidimensional Architectures

In order to enhance the scalability of multiprocessor systems, many research groups have explored economical and multidimensional architectures that support fast communication, coherence extension, distributed shared memory, and modular packaging.

The architecture of massively parallel processors has evolved from one-dimensional *rings* to two-dimensional and three-dimensional meshes or tori as illustrated in Fig. 9.17. The Maryland Zmob experimented on a *slotted token ring* for building a multiprocessor. Both the CDC Cyberplus and KSR-1 used hierarchical (two-level) ring architectures. The ring is the simplest architecture to implement from the viewpoint of backplane packaging.

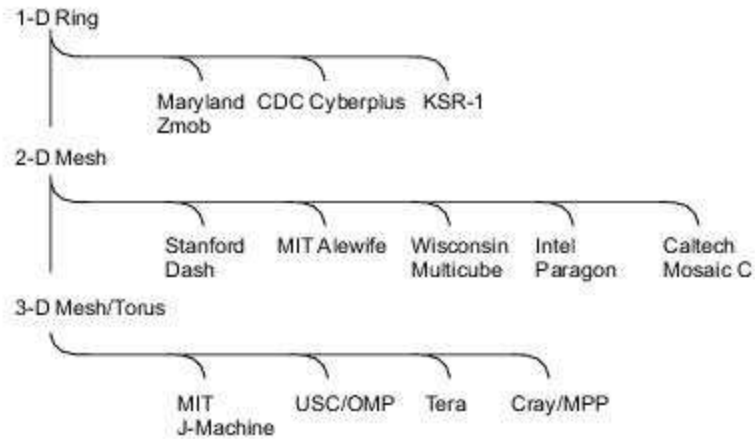
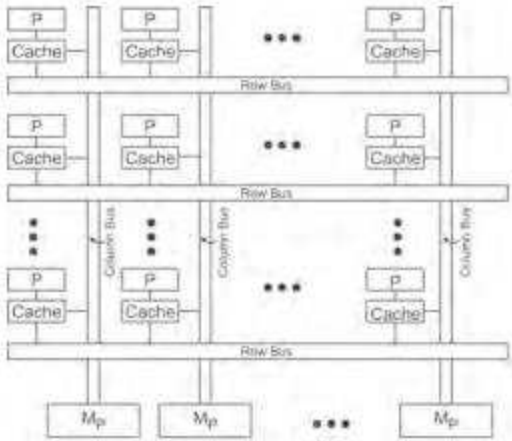


Fig. 9.17 The evolution from one-dimensional ring to two-dimensional mesh and then to three-dimensional mesh/torus architecture for building massively parallel processors.

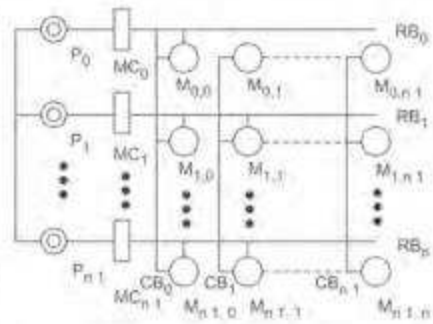
Two-dimensional meshes were adopted in the Stanford Dash, the MIT Alewife, the Wisconsin Multicube, the Intel Paragon, and the Caltech Mosaic C. A three-dimensional mesh/torus was implemented in the MIT J-Machine, the Tera computer, and in the Cray/MPP architecture, called T3D. The USC *orthogonal multiprocessor* (OMP) could be extended to higher dimensions. However, it becomes more difficult to build higher-dimensional architectures with conventional two-dimensional circuit boards.

Instead of using hierarchical buses or switched network architectures in one dimension, multiprocessor architectures can be extended to a higher *dimensionality* or *multiplicity* along each dimension. The concepts are described below for two- and three-dimensional meshes proposed for the Multicube and OMP architectures, respectively.

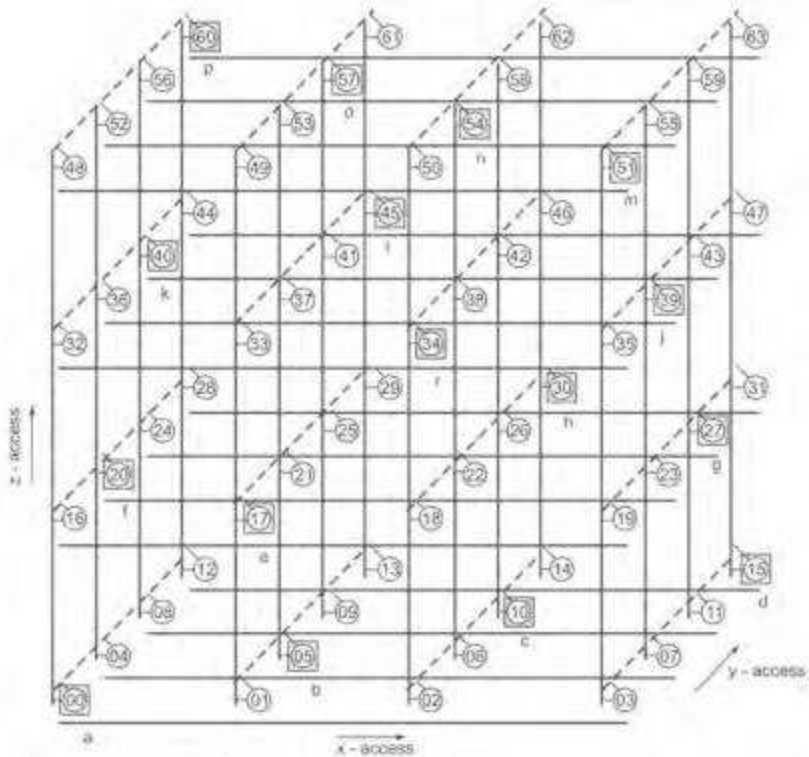
The Wisconsin Multicube This architecture was proposed by Goodman and Woest (1988) at the University of Wisconsin. It employed a snooping cache system over a grid of buses, as shown in Fig. 9.18a. Each processor was connected to a multilevel cache.



(a) The Wisconsin Multicube



(b) The two-dimensional OMP(2,n). (Pi: Processors; Mij: memory modules; RB_i: row buses; CB_j: column buses)



(c) The 3-D OMP (3,4) architecture. (Processors are labeled a, b, ..., p; memory modules are labeled 00,01,... 63)

Fig. 9.18 The Multicube and orthogonal multiprocessor architectures (Courtesy of Goodman and Woest, 1988, and of Hwang et al, 1989)

The first-level cache, called the *processor cache*, was a high-performance SRAM cache designed with the traditional goal of minimizing memory latency. A second-level cache, referred to as the *snooping cache*, was a very large cache designed to minimize bus traffic.

Each snooping cache monitored two buses, a row bus and a column bus, in order to maintain data consistency among the snooping caches. Consistency between the two cache levels was maintatined by using a write-through strategy to ensure that the processor cache is always a strict subset of the snooping cache. The main memory was divided up among the column buses. All processors tied to the same column shared the same home memory. The row buses were used for intercolumn communication and cache coherence control.

The proposed architecture was an example of a new class of interconnection topologies, the *multicube*, consisting of $N = n^k$ processors, where each processor was connected to k buses and each bus was connected to n processors. The hypercube is a special case where $n = 2$. The Wisconsin Multicube was a two-dimensional multicube ($k = 2$), where n scaled to about 32, resulting in a proposed system of over 1000 processors.

The Orthogonal Multiprocessor In the proposed OMP architecture (Fig. 9.18b), n processors simultaneously access n rows or n columns of interleaved memory modules. The $n \times n$ *memory mesh* is interleaved in both dimensions. In other words, each row is n-way interleaved and so is each column of memory modules. There are $2n$ logical buses spanning in two orthogonal directions.

The synchronized row access or column access must be performed exclusively. In fact, the row bus R_i and the column bus C_i can be the same physical bus because only one of the two will be used at a time. The memory controller (MC) in Fig. 9.18b synchronizes the row access and column access of the shared memory.

The OMP architecture supports special-purpose computations in which data sets can be regularly arranged as matrices. Simulated performance results obtained at USC verified the effectiveness of using an OMP in matrix algebraic computations or in image processing operations.

In Fig. 9.18b, each of the memory modules M_{ij} is shared by two processors P_i and P_j. In other words, the physical address space of processor P_i covers only the ith row or the ith column of the memory mesh. The OMP is well suited for SPMD operations, in which n processors are synchronized at the memory-access level when data sets are vectorized in matrix format.

Multidimensional Extensions The above OMP architecture can be generalized to higher dimensions. A generalized orthogonal multiprocessor is denoted as an OMP(n, k), where n is the *dimension* and k is the *multiplicity*. There are $p = k^{n-1}$ processors and $m = k^n$ memory modules in the system, where $p \gg n$ and $p \gg k$.

The system uses p memory buses, each spanning into n dimensions. But only one dimension is used in a given memory cycle. There are k memory modules attached to each spanning bus.

Each module is connected to n out of p buses through an n-way switch. It should be noted that the dimension n corresponds to the number of accessible ports that each memory module has. This implies that each module is shared by n out of $p = k^{n-1}$ processors. For example, the architecture of an OMP(3,4) is shown in Fig. 9.18c, where the circles represent memory modules, the squares processor modules, and the circles inside squares computer modules.

The 16 processors orthogonally access 64 memory modules via 16 buses, each spanning into three directions, called the *x-access*, *y-access*, and *z-access*, respectively. Various sizes of OMP architecture for different values of n and k are given in Table 9.2. A five-dimensional OMP with multiplicity $k = 16$ has 64K processors.

Table 9.2 *Orthogonal Multiprocessor of Dimension n and Multiplicity k*

OMP(n, k)	$p = k^{n-1}$	$m = k^n$
OMP(2, 8)	8	64
OMP(2, 16)	16	256
OMP(3, 8)	64	512
OMP(3, 16)	256	4096
OMP(4, 8)	512	4096
OMP(4, 16)	4096	65,536
OMP(5,16)	65,536	1,048,576

Note: p = number of processors; m = number of memory modules.

9.3 FINE-GRAIN MULTICOMPUTERS

Traditionally, shared-memory multiprocessors like the Cray Y-MP were used to perform coarse-grain computations in which each processor executed programs having tasks of a few seconds or longer. Message-passing multicomputers are used to execute medium-grain programs with approximately 10-ms task size as in the iPSC/1. In order to build MPP systems, we may have to explore a higher degree of parallelism by making the task grain size even smaller.

Fine-grain parallelism was utilized in SIMD or data-parallel computers like the CM-2 or on the message-driven J-Machine and Mosaic C to be described below. We first characterize fine-grain parallelism and discuss the network architectures proposed for such systems. Special attention is paid to the efficient hardware or software mechanisms developed for achieving fine-grain MIMD computation.

9.3.1 Fine-Grain Parallelism

We compare below the grain sizes, communication latencies, and concurrency in four classes of parallel computers. This comparison leads to the rationales for developing fine-grain multicomputers. In Chapter 13 we shall review recent developments.

Latency Analysis The computing granularity and communication latency of leading early examples of multiprocessors, data-parallel computers, and medium-and fine-grain multicomputers are summarized in Table 9.3. These table entries summarize what we have learned in Chapters 7 and 8. Four attributes are identified to characterize these machines. Only typical values for a typical program mix are shown. The intention is to show the order of magnitude in these entries.

The *communication latency* T_c measures the data or message transfer time on a system interconnect. This corresponds to the shared-memory access time on the Cray Y-MP, the time required to send a 32-bit value across the hypercube network in the CM-2, and the network latency on the iPSC/1 or J-Machine. The *synchronization overhead* T_s is the processing time required on a processor, or on a PE, or on a processing node of a multicomputer for the purpose of synchronization.

The sum $T_c + T_s$ gives the total time required for IPC. The shared-memory Cray Y-MP had a short T_c but a long T_s. The SIMD machine CM-2 had a short T_s but a long T_c. The long latency of the iPSC/1 made it unattractive based on fast advancing standards. The MIT J-Machine was designed to make a major improvement in both of these communication delays.

Fine-Grain Parallelism The *grain size* T_g is measured by the execution time of a typical program, including both computing time and communication time involved. Supercomputers handle large grain. Both the CM-2 and the J-Machine were designed as fine-grain machines. The iPSC/1 was a relatively medium-grain machine compared with the rest.

Large grain implies lower concurrency or a lower DOP (degree of parallelism). Fine grain leads to a much higher DOP and also to higher communication overhead. SIMD machines used hardwired synchronization and massive parallelism to overcome the problems of long network latency and slow processor speed. *Fine-grain multicomputers*, like the J-Machine *and* Caltech Mosaic, were designed to lower both the grain size and the communication overhead compared to those of traditional multicomputers.

Table 9.3 *Fine-Grain, Medium-Grain, and Coarse-Grain Machine Characteristics of Some Example Systems*

Characteristics	*Machine*			
	Cray Y-MP	*Connection Machine CM-2*	*Intel iPSC/1*	*MIT J-Machine*
Communication latency, T_c	40 ns via shared memory	600 μs per 32-bit *send* operation	5 ms	2 μs
Synchronization overhead, T_s	20 μs	125 ns per bit-slice operation in lock step	500 μs	1 μs
Grain size, T_g	20 s	4 μs per 32-bit result per PE instruction	10 ms	5 μs
Concurrency (DOP)	2–16	8K–64K	8–128	1K–64K
Remark	Coarse-grain supercomputer	Fine-grain data parallelism	Medium-grain multicomputer	Fine-grain multicomputer

9.3.2 The MIT J-Machine

The architecture and building block of the MIT J-Machine, its instruction set, and system design considerations are described below based on the paper by Dally et al (1992). The building block was the *message-driven processor* (MDP), a 36-bit microprocessor custom-designed for a fine-grain multicomputer.

The J-Machine Architecture The k-ary n-*cube* networks were applied in the MIT J-Machine. The initial prototype J-Machine used a 1024-node network ($8 \times 8 \times 16$), which was a reduced 16-ary 3-cube with 8 nodes along the x- and y-dimensions and 16 nodes along the z-dimension. A 4096-node J-Machine would use a full 16-ary 3-cube with $16 \times 16 \times 16$ nodes. The J-Machine designers called their network a three-dimensional mesh.

Network addressing limited the size of the J-Machine to a maximum configuration of 65,536 nodes, corresponding to a three-dimensional mesh with $32 \times 32 \times 64$ nodes. The architecture of the three-dimensional mesh or a general k-ary n-*cube* was shown in Fig. 2.20 for the case of $k = 4$. All hidden parts (nodes and links) are not shown for purposes of clarity. Clearly, every node has a constant node degree of 6, and there are three rings crossing each node along the three dimensions. The end-around connections can be folded (Fig. 2.21b) to balance the wire length on all channels.

The MDP Design The MDP chip included a processor, a 4096-word by 36-bit memory, and a built-in router with network ports as shown in Fig. 9.19. An on-chip memory controller with error checking and correction (ECC) capability permitted local memory to be expanded to 1 million words by adding external DRAM chips. The processor was message-driven in the sense that it executed functions in response to messages, via the dispatch mechanism. No receive instruction was needed.

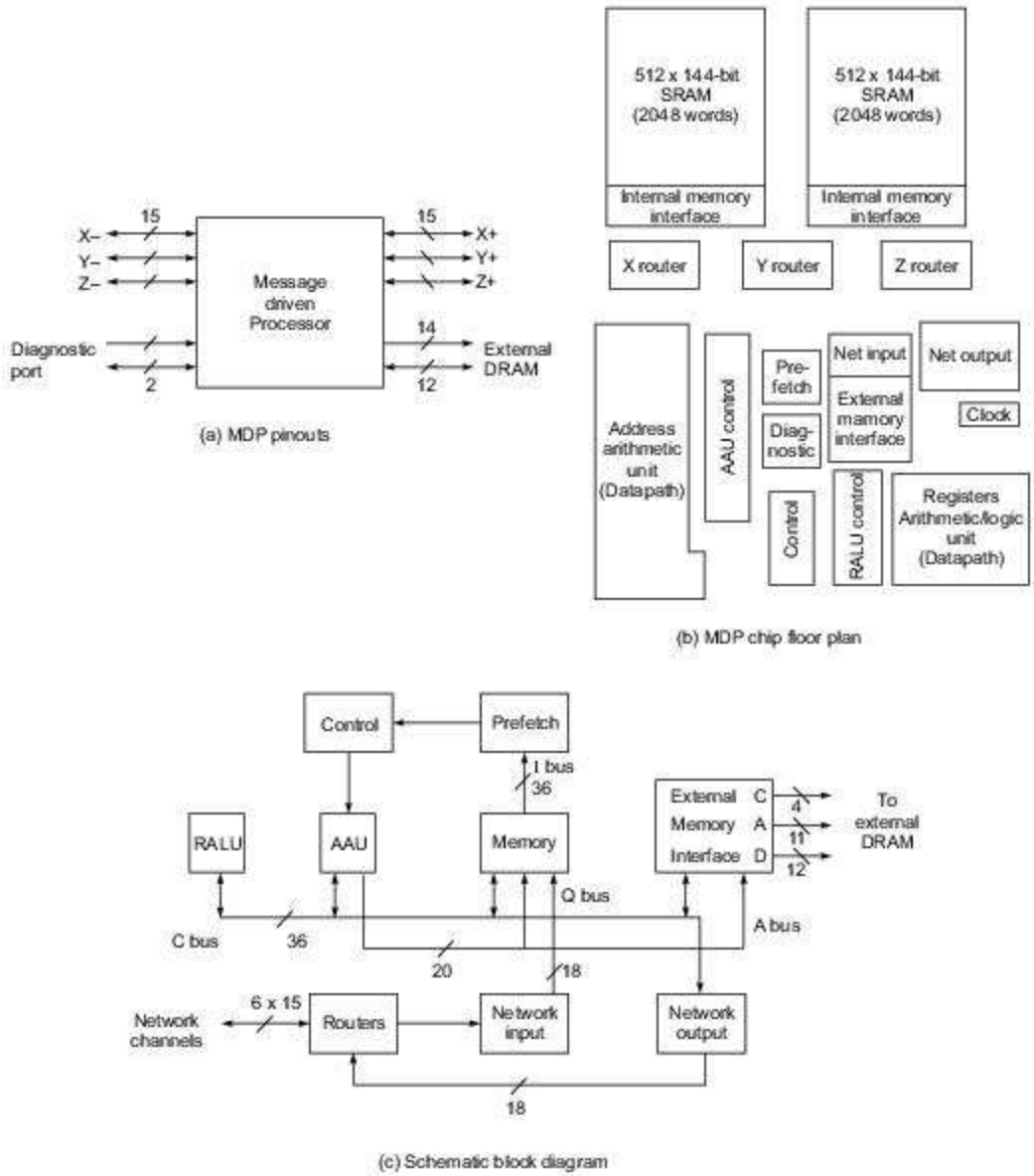


Fig. 9.19 The message-driven processor (MDP) architecture (Courtesy of W. Dally et al; reprinted with permission from *IEEE Micro*, April 1992)

The MDP created a task to handle each arriving message. Messages carrying these tasks drove each computation. MDP was a general-purpose multicomputer processing node that provided the communication, synchronization, and global naming mechanisms required to efficiently support fine-grain, concurrent programming models. The grain size was as small as 8-word objects or 20-instruction tasks. As we have seen, fine-grain programs typically execute from 10 to 100 instructions between communication and synchronization actions.

MDP chips provided inexpensive processing nodes with plentiful VLSI commodity parts to construct the Jellybean Machine (J-Machine) multicomputer. As shown in Fig. 9.19a, the MDP appeared as a component with a memory port, six two-way network ports, and a diagnostic port.

The memory port provided a direct interface to up to 1M words of ECC DRAM, consisting of 11 multiplexed address lines, a 12-bit data bus, and 3 control signals. Prototype J-Machines used three 1M × 4 static-column DRAMs to form a four-chip processing node with 262,144 words of memory. The DRAMs cycled three times to access a 36-bit data word and a fourth time to check or update the ECC check bits.

The network ports connected MDPs together in a three-dimensional mesh network. Each of the six ports corresponded to one of the six cardinal directions (+x, –x, +y, –y, +z, –z) and consisted of nine data and six control lines. Each port connected directly to the opposite port on an adjacent MDP.

The diagnostic port could issue supervisory commands and read and write MDP memory from a console processor (host). Using this port, a host could read or write at any location in the MDP's address space, as well as reset, interrupt, halt, or single-step the processor. The MDP chip floor plan is shown Fig. 9.19b.

Figure 9.19c shows the components built inside the MDP chip. The chip included a conventional microprocessor with prefetch, control, register file and ALU (RALU), and memory blocks. The network communication subsystem comprised the routers and network input and output interfaces. The *address arithmetic unit* (AAU) provided addressing functions. The MDP also included a DRAM interface, control clock, and diagnostic interface.

Instruction-Set Architecture The MDP extended a conventional microprocessor instruction-set architecture with instructions to support parallel processing. The instruction set contained fixed-format, three-address instructions. Two 17-bit instructions fit into each 36-bit word with 2 bits reserved for type checking.

Separate register sets were provided to support rapid switching among three execution levels: background, priority 0 (P0), and priority 1 (P1). The MDP executed at the background level while no message created a task, and initiated execution upon message arrival at P0 or P1 level depending on the message priority.

P1 level had higher priority than P0 level. The register set at each priority level included four GPRs, four address registers, four ID registers, and one instruction pointer (IP). The ID registers were not used in the background register set.

Communication Support The MDP provided hardware support for end-to-end message delivery including formatting, injection, delivery, buffer allocation, buffering, and task scheduling. An MDP transmitted a message using a series of SEND instructions, each of which injected one or two words into the network at either priority 0 or 1.

Consider the following MDP assembly code for sending a four-word message using three variants of the SEND instruction.

```
SEND     R0,0       ;   send net address (priority 0)
SEND2    R1,R2,0    ;   header and receiver (priority 0)
SEND2E   R3,[3,A3],0 ;  selector and continuation end message (priority 0)
```

The first SEND instruction reads the absolute address of the destination node in $< X, Y, Z >$ format from R0 and forwards it to the network hardware. The SEND2 instruction reads the first two words of the message out of registers R1 and R2 and enqueues them for transmission. The final instruction enqueues two additional words of data, one from R3 and one from memory. The use of the SEND2E instruction marks the end of the message and causes it to be transmitted into the network.

The J-Machine was a three-dimensional mesh with two-way channels, dimension-order routing, and blocking flow control (Fig. 9.20). The faces of the network cube were open for use as I/O ports to the machine. Each channel could sustain a data rate of 288 Mbps (million bits per second). All three dimensions could operate simultaneously for an aggregate data rate of 864 Mbps per node.

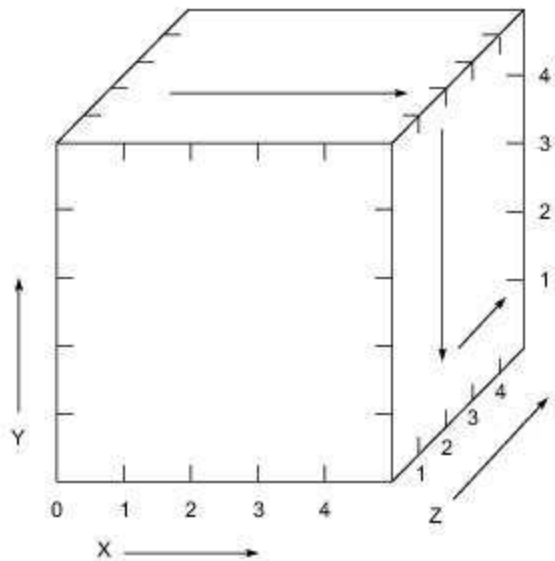


Fig. 9.20 E-cube routing from node (1, 5, 2) to node (5, 1, 4) on a 6-ary 3-cube

Message Format and Routing The J-Machine used deterministic dimension-order E-cube routing. As shown in Fig. 9.20, all messages routed first in the x-dimension, then in the y-dimension, and then in the z-dimension. Since messages routed in dimension order and messages running in opposite directions along the same dimension cannot block, resource cycles were thus avoided, making the network provably deadlock-free.

Example 9.3 A typical message in the MIT J-Machine (W. Dally et al, 1992)

The following message consists of nine flits. The first three flits of the message contain the x-, y-, and z-addresses. Each node along the path compares the address in the head flit of the message. If the two indices match, the node routes the rest to the next dimension. The final flit in the message is marked as the tail.

Flit	Contents		Remarks
1	5:+		x-address
2	1:–		y-address
3	4:+		z-address
4	Msg: 00		Method to call
5	00440		
6	INT: 00		Argument to method
7	0023		
8	INT: 0 0		Reply address
9	< 1 : 5 : 2 >	T	

The MDP supported a broad range of parallel programming models, including shared memory, data-parallel, dataflow, actor, and explicit message passing, by providing a low-overhead primitive mechanism for communication, synchronization, and naming.

Its communication mechanisms permitted a user-level task on one node to send a message to any other node in a 4096-node machine in less than 2 μs. This process did not consume any processing resources on intermediate nodes, and it automatically allocated buffer memory on the receiving node. On message arrival, the receiving node created and dispatched a task in less than 1 μs.

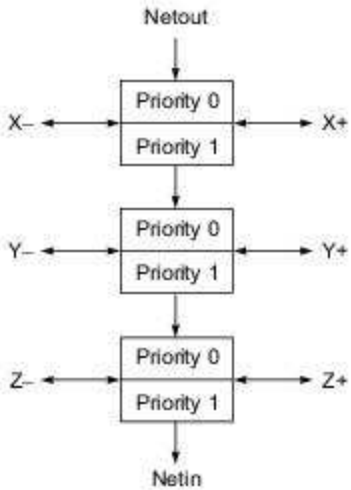
Presence tags provided synchronization on all storage locations. Three separate register sets allowed fast context switching. A translation mechanism maintained bindings between arbitrary names and values and supported a global virtual address space. These mechanisms were selected to be general and amenable to efficient hardware implementation. The J-Machine used wormhole routing and blocking flow control. A combining-tree approach was used for synchronization.

The Router Design The routers formed the switches in a J-Machine network and delivered messages to their destinations. As shown in Fig. 9.21a, the MDP contained three independent routers, one for each bidirectional dimension of the network.

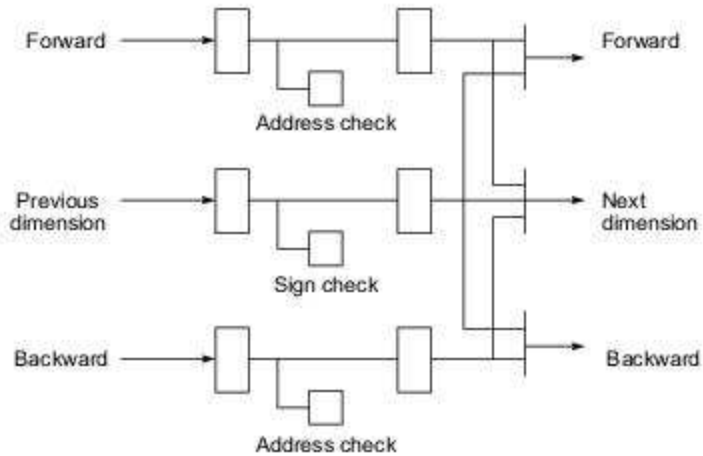
Each router contained two separate virtual networks with different priorities that shared the same physical channels. The priority-1 network could preempt the wires even if the priority-0 network was congested or jammed. The priority levels supported multi-threaded operations.

Each of the 18 router paths contained buffers, comparators, and output arbitration (Fig. 9.21). On each data path, a comparator compared the lead flit, which contained the destination address in that dimension, to the node coordinate. If the head flit did not match, the message continued in the current direction. Otherwise the message was routed to the next dimension.

A message entering the dimension competed with messages continuing in the dimension at a two-to-one switch. Once a message was granted this switch, all other input was locked out for the duration of the message. Once the head flit of the message had set up the route, subsequent flits followed directly behind it.



(a) Dual-priority levels per dimension in the router



(a) Each priority with forward, reverse, and previous data paths to the next dimension.

Fig. 9.21 Priority control and dimension-order router design in the MDP chip (Courtesy of W. Dally et al; reprinted with permission from *IEEE Micro*, April 1992)

Two priorities of messages shared the physical wires but used completely separate buffers and routing logic. This allowed priority-1 messages to proceed through blockages at priority 0. Without this ability, the system would not be able to redistribute data that caused hot spots in the network.

Synchronization The MDP synchronized using message dispatch and presence tags on all states. Because each message arrival dispatched a process, messages could signal events on remote nodes. For example, in the following combining-tree example, each COMBINE message signals its own arrival and initiates the COMBINE routine.

In response to an arriving message, the processor may set presence tags for task synchronization. For example, access to the value produced by the combining tree may be synchronized by initially tagging as empty the location that will hold this value. An attempt to read this location before the combining tree has written it will raise an exception and suspend the reading task until the root of the tree writes the value.

Example 9.4 Using a combining tree for synchronization of events (W. Dally et al, 1992)

A combining tree is shown in Fig. 9.22. This tree sums results produced by a distributed computation. Each node sums the input values as they arrive and then passes a result message to its parent.

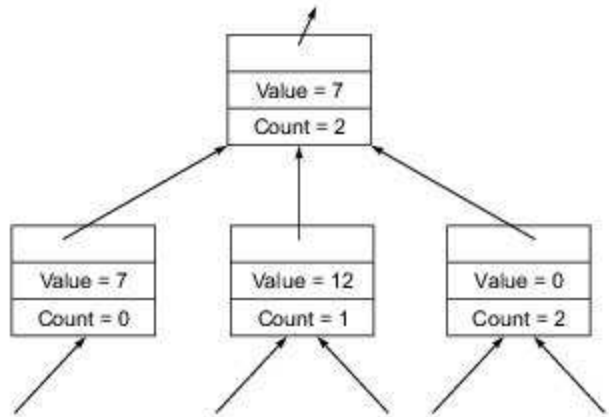


Fig. 9.22 A combining tree for internode communication or synchronization (Courtesy of W. Dally et al, 1992)

A pair of SEND instructions was used to send the COMBINE message to a node. Upon message arrival, the MDP buffered the message and created a task to execute the following COMBINE routine written in MDP assembly code:

```
COMBINE:  MOVE    [1, A3], COMB          ;  get node pointer from message
          MOVE    [2, A3], R1            ;  get value from message
          ADD     R1, COMB.VALUE, R1
          MOVE    R1, COMB.VALUE         ;  store result
          MOVE    COMB.COUNT, R2         ;  get Count
          ADD     R2, -1, R2
          MOVE    R2, COMB.COUNT         ;  store decremented Count
          BNZ     R2, DONE
          MOVE    HEADER, R0             ;  get message header
          SEND2   COMB.PARENT_NODE, R0   ;  send message to parent
          SEND2E  COMB.PARENT, R1        ;  with value
DONE:     SUSPEND
```

If the node was idle, execution of this routine began three cycles after message arrival. The routine loaded the combining-node pointer and value from the message, performed the required add and decrement, and, if Count reached zero, sent a message to its parent.

Research Issues The J-Machine was an exploratory research project. Rather than being specialized for a single model of computation, the MDP incorporated primitive mechanisms for efficient communication, synchronization, and naming. The machine was used as a platform for software experiments in fine-grain parallel programming.

Reducing the grain size of a program increases both the potential speedup due to parallel execution and the potential overhead associated with parallelism. Special hardware mechanisms for reducing the overhead

due to communication, process switching, synchronization, and multi-threading were therefore central to the design of the MDP. Software issues such as load balancing, scheduling, and locality also remained open questions.

The MIT research group led by Dally implemented two languages on the J-Machine: the actor language Concurrent Smalltalk and the dataflow language Id. The machine's mechanism also supported dataflow and object-oriented programming models using a global name space. The use of a few simple mechanisms provided orders of magnitude lower communication and synchronization overhead than was possible with multicomputers built from then available off-the-shelf microprocessors.

9.3.3 The Caltech Mosaic C

The Caltech Mosaic C was an experimental fine-grain multicomputer that employed single-chip nodes and advanced packaging technology to demonstrate the performance/cost advantages of fine-grain multicomputer architecture. We describe below the architecture of the Mosaic C and review its application potentials, based on a report by Seitz (1992), the project leader at Caltech.

From Cosmic Cube to Mosaic C The evolution from the Cosmic Cube to the Mosaic is an example of one type of *scaling track* in which advances in technology are employed to reimplement nodes of a similar logical complexity but which are faster and smaller, have lower power, and are less expensive. The progress in microelectronics over the preceding decade was such that Mosaic nodes were = 60 times faster, used = 20 times less power, were = 100 times smaller, and were (in constant dollars) = 25 times less expensive to manufacture than Cosmic Cube nodes.

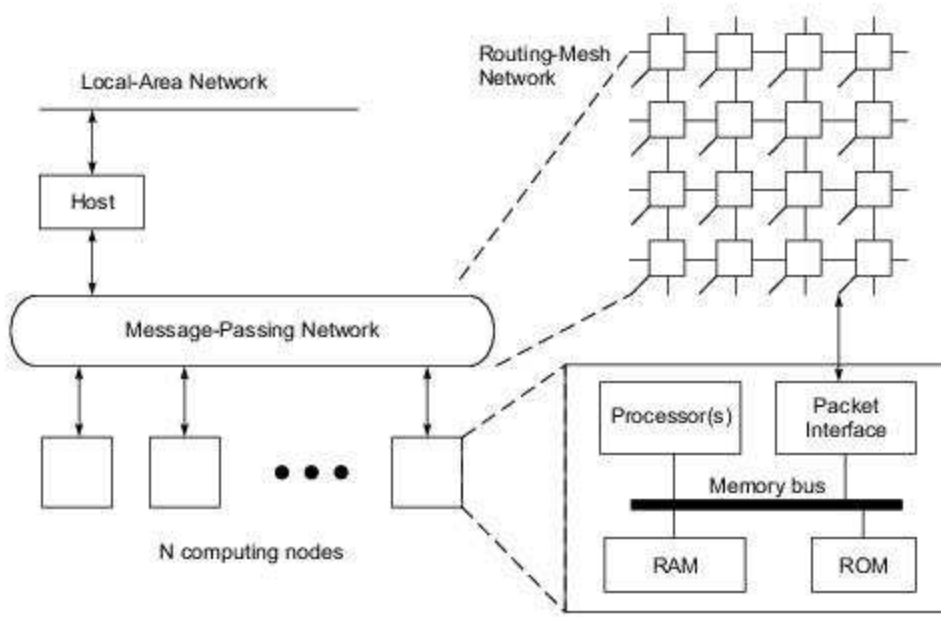


Fig. 9.23 The Caltech Mosaic architecture (Courtesy of C. Seitz, 1992)

Each Mosaic node included 64 Mbytes of memory and an 11-MIPS processor, a packet interface, and a router. The nodes were tied together with a 60-Mbytes/s, two-dimensional routing-mesh network (Fig. 9.23).

The compilation-based programming system allowed fine-grain reactive-process message-passing programs to be expressed in C+−, an extension of C++, and the run-time system performed automatic distributed management of system resources.

Mosaic C Node The Mosaic C multicomputer node was a single 9.25 mm × 10.00 mm chip fabricated in a 1.2-μm-feature-size, two-level-metal CMOS process. At 5-V operation, the synchronous parts of the chip operated with large margins at a 30-MHz clock rate, and the chip dissipated = 0.5 W.

The processor also included two program counters and two sets of general-purpose registers to allow zero-time context switching between user programs and message handling. Thus, when the packet interface received a complete packet, received the header of a packet, completed the sending of a packet, exhausted the allocated space for receiving packets, or any of several other events that could be selected, it could interrupt the processor by switching it instantly to the message-handling context.

Instead of several hundred instructions for handling a packet, the Mosaic typically required only about 10 instructions. The number of clock cycles for the message-handling routines could be reduced to insignificance by placing them in hardware, but the Caltech group chose the more flexible software mechanism so that they could experiment with different message-handling strategies.

Mosaic C 8 × 8 Mesh Boards The choice of a two-dimensional mesh for the Mosaic was based on a 1989 engineering analysis; originally, a three-dimensional mesh network was planned. But the mutual fit of the two-dimensional mesh network and the circuit board medium provided high packaging density and allowed the high-speed signals between the routers to be conveyed on shorter wires.

Sixty-four Mosaic chips were packaged by tape-automated bonding (TAB) in an 8 × 8 array on a circuit board. These boards allowed the construction of arbitrarily large, two-dimensional arrays of nodes using stacking connectors. This style of packaging was meant to demonstrate some of the density, scaling, and testing advantages of mesh-connected systems. Host-interface boards were also used to connect the Mosaic arrays and workstations.

Applications and Future Trends Charles Seitz determined that the most profitable niche and scaling track for the multicomputer, a highly scalable and economical MIMD architecture, was the fine-grain multicomputer. The Mosaic C demonstrated many of the advantages of this architecture, but the major part of the Mosaic experiment was to explore the programmability and application span of this class of machine.

The Mosaic may be taken as the origin of two scaling tracks: (1) Single-chip nodes are a technologically attractive point in the design space of multicomputers. Constant-node-size scaling results in single-chip nodes of increasing memory size, processing capability, and communication bandwidth in larger systems than centralized shared-memory multiprocessors. (2) It was also forecasts that constant-node-complexity scaling would allow a Mosaic 8 × 8 board to be implemented as a single chip, with about 20 times the performance per node, within 10 years. In this context, see also the discussion in Chapter 13.

A 16K-node machine was constructed at Caltech to explore the programmability and application span of the Mosaic C architecture for large-scale computing problems. For the loosely coupled computations in which it excels, a multicomputer can be more economically implemented as a network of high-performance workstations connected by a high-bandwidth local-area network. In fact, the Mosaic components and programming tools were used by a USC Information Science Institute project (led by Danny Cohen, 1992) to implement a 400-Mbits/s ATOMIC local-area network for this purpose.

9.4 SCALABLE AND MULTITHREADED ARCHITECTURES

Three pioneering and landmark scalable multiprocessor systems are discussed in this section. The Stanford Dash combined several latency-hiding mechanisms. The Kendall Square Research KSR-1 offered the first commercial attempt to produce a multiprocessor with cache-only memory. The Tera computer evolved from the HEP/Horizon series developed by Burton Smith. Only the main architectural features are described below. All three systems were extensions of the traditional von Neumann model. By far, the Tera system represented the most aggressive attempt to build a multi-threaded multiprocessor.

9.4.1 The Stanford Dash Multiprocessor

This was an experimental multiprocessor system developed by John Hennessy and coworkers at Stanford University beginning in 1988. The name Dash is an abbreviation for *Directory Architecture for Shared Memory*. The fundamental premise behind Dash was that it is possible to build a scalable high-performance machine with a single address space, coherent caches, and distributed memories. The directory-based coherence gave Dash the ease of use of shared-memory architectures, while maintaining the scalability of message-passing machines.

The Prototype Architecture A high-level organization of the Dash architecture was illustrated in Fig. 9.1 when we studied the various latency-hiding techniques. The Dash prototype is illustrated in Fig. 9.24. It incorporated up to 64 MIPS R3000/R3010 microprocessors with 16 clusters of 4 PEs each. The cluster hardware was modified from Silicon Graphics 4D/340 nodes with new directory and reply controller boards as depicted in Fig. 9.24a.

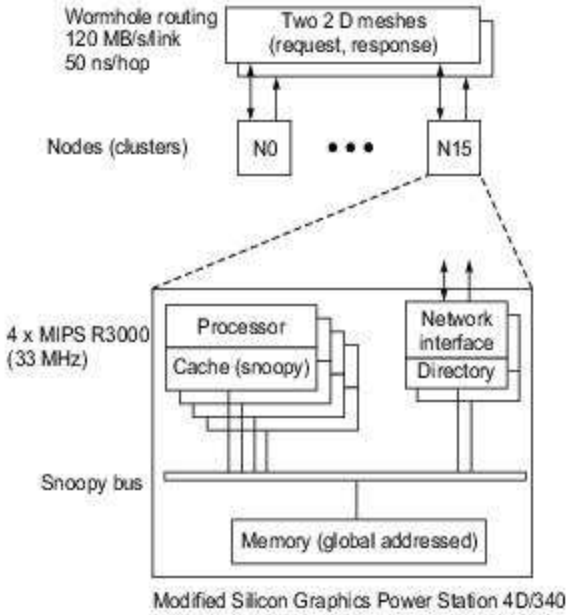
The interconnection network among the 16 multiprocessor clusters was a pair of wormhole-routed mesh networks. The channel width was 16 bits with a 50-ns fall-through time and a 35-ns cycle time. One mesh network was used to *request* remote memory, and the other was a *reply* mesh as depicted in Fig. 9.24b, where the small squares at mesh intersections are the 5×5 mesh routers.

The Dash designers claimed scalability for the Dash approach. Although the prototype was limited to at most 16 clusters (a 4×4 mesh), due to the limited physical memory addressability (256 Mbytes) of the 4D/340 system, the system was scalable to support hundreds to thousands of processors.

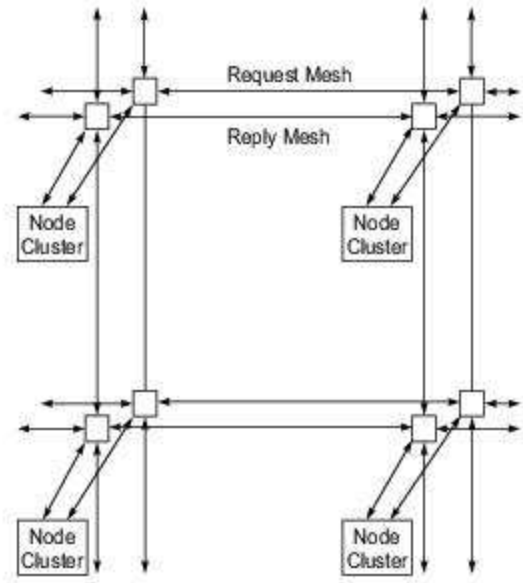
To use the 4D/340 in the Dash, the Stanford team made minor modifications to the existing system boards and designed a pair of new boards to support the directory memory and intercluster interface. The main modification to the existing boards was to add a bus retry signal, to be used when a request required service from a remote cluster.

The central bus arbiter was modified to accept a mask from the directory. The mask held off a processor's retry until the remote request was serviced. This effectively created a split-transaction bus protocol for requests requiring remote service.

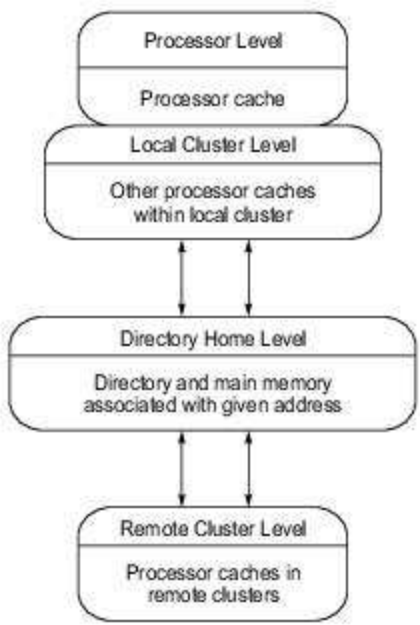
The new directory controller boards contained the directory memory, the intercluster coherence state machines and buffers, and a local section of the global interconnection network. The directory logic was split between the two logic boards along the lines of the logic used for outbound and inbound portions of intercluster transactions.



(a) The prototype node implementation



(a) Block diagram of 2 × 2 mesh interconnect



(c) Logic memory hierarchy

Fig. 9.24 The Stanford Dash prototype system (Courtesy of D. Lenoski et al, *Proc. 19th Int. Symp. Comput. Archit.*, Australia, May 1992)

The mesh networks supported a scalable local and global memory bandwidth. The single-address space with coherent caches permitted incremental porting or tuning of applications, and exploited temporal and spatial locality. Other factors contributing to improved performance included mechanisms for reducing and tolerating latency, and well-designed I/O capabilities.

Dash Memory Hierarchy Dash implemented an invalidation-based cache coherence protocol. A memory location could be in one of three states:

- *Uncached*—not cached by any cluster;
- *Shared*—in an unmodified state in the caches of one or more clusters; or
- *Dirty*—modified in a single cache of some cluster.

The directory kept the summary information for each memory block, specifying its state and the clusters cacheing it. The Dash memory system could be logically broken into four levels of hierarchy, as illustrated in Fig. 9.25c.

The first level was the processor cache which was designed to match the processor speed and support snooping from the bus. It took only one clock to access the processor cache. A request that could not be serviced by the processor cache was sent to the *local cluster*. The prototype allowed 30 processor clocks to access the local cluster. This level included the other processors' caches within the requesting processor's cluster.

Otherwise, the request was sent to the *home cluster* level. The home level consisted of the cluster that contained the directory and physical memory for a given memory address. It took 100 processor clocks to access the directory at the home level. For many accesses (for instance, most private data references), the local and home cluster were the same, and the hierarchy collapsed to three levels. In general, however, a request would travel through the interconnection network to the home cluster.

The home cluster could usually satisfy the request immediately, but if the directory entry was in a dirty state, or in a shared state when the requesting processor requested exclusive access, the fourth level had to be accessed. The *remote cluster* level for a memory block consisted of the clusters marked by the directory as holding a copy of the block. It took 135 processor clocks to access processor caches in remote clusters in the prototype design.

The Directory Protocol The directory memory relieved the processor caches of snooping on memory requests by keeping track of which caches held each memory block. In the home node, there was a directory entry per block frame. Each entry contained one *presence bit* per processor cache. In addition, a *state bit* indicated whether the block was uncached, shared in multiple caches, or held exclusively by one cache (i.e. whether the block was dirty).

Using the state and presence bits, the memory could tell which caches needed to be invalidated when a location was written. Likewise, the directory indicated whether the memory copy of the block was up-to-date or which cache held the most recent copy.

By using the directory memory, a node writing a location could send point-to-point invalidation or update messages to the processors actually cacheing that block. This is in contrast to the invalidating broadcast required by the snoopy protocol. The scalability of the Dash depended on this ability to avoid broadcasts.

Another important attribute of a directory-based protocol is that it does not depend on any specific interconnection network topology. As a result, the designer can readily use any of the low-latency scalable networks, such as meshes or hypercubes, that were originally developed for message-passing machines.

Example 9.5 Cache coherence protocol using distributed directories in the Dash multiprocessor (Daniel Lenoski and John Hennessy et al, 1992.)

Figure 9.25a illustrates the flow of a read request to remote memory with the directory in a dirty remote state. The read request is forwarded to the owning dirty cluster. The owning cluster sends out two messages in response to the read. A message containing the data is sent directly to the requesting cluster, and a sharing writeback request is sent to the home cluster. The sharing writeback request writes the cache block back to memory and also updates the directory.

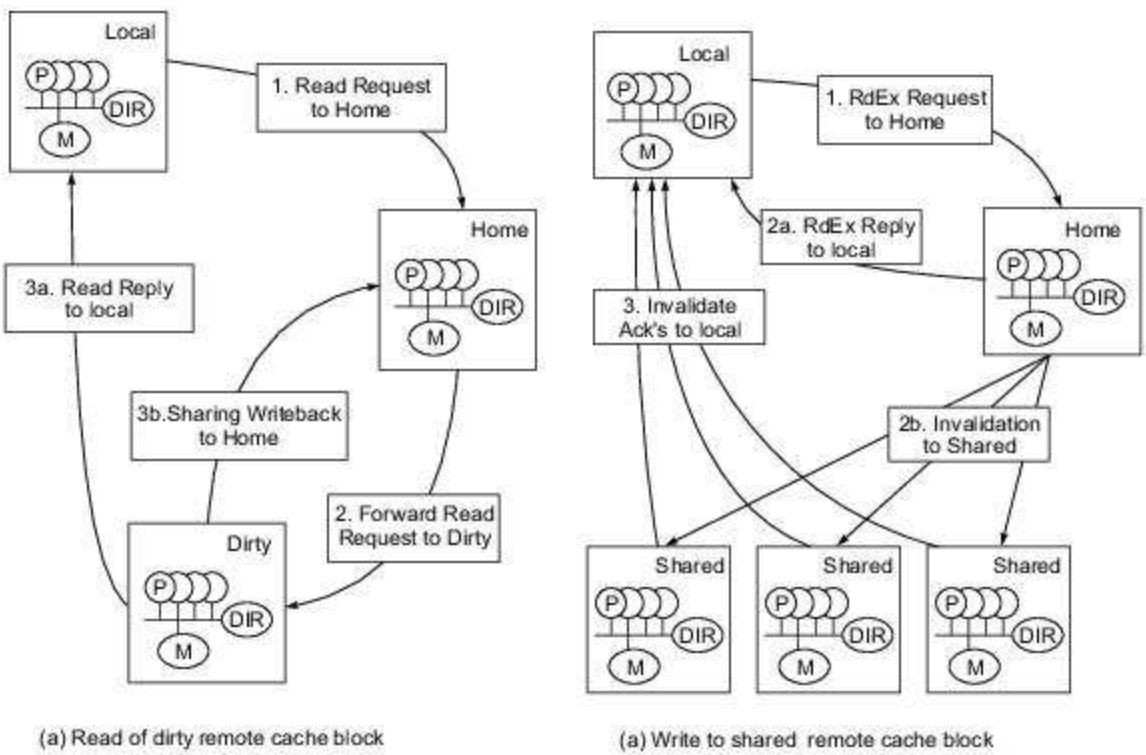


Fig. 9.25 Two examples of a directory-based cache coherence protocol in the Dash (Courtesy of Lenoski and Hennessy, 1992)

This protocol reduces latency by permitting the dirty cluster to respond directly to the requesting cluster. In addition, this forwarding strategy allows the directory controller to simultaneously process many requests (i.e. to be multithreaded) without the added complexity of maintaining the state of outstanding requests. Serialization is reduced to the time of a single intercluster bus transaction. The only resource held while intercluster messages are being sent is a single entry in the originating cluster's remote-access cache.

Figure 9.25b shows the corresponding sequence for a write operation that requires remote service. The invalidation-based protocol requires the processor (actually the write buffer) to acquire exclusive ownership of the cache block before completing the store. Thus, if a write is made to a block that the processor does not have cached, or only has cached in a shared state, the processor issues a read-exclusive request on the local bus.

In this case, no other cache holds the block entry dirty in the local cluster, so a RdEx Request (message 1) is sent to the home cluster. As before, a remote-access cache entry is allocated in the local cluster. At the home cluster, the pseudo-CPU issues the read-exclusive request to the bus. The directory indicates that the line is in the shared state. This results in the directory controller sending a RdEx Reply (message 2a) to the local cluster and invalidation requests (Inv-Req, message 2b) to the sharing cluster.

The home cluster owns the block, so it can immediately update the directory to the dirty state, indicating that the local cluster now holds an exclusive copy of the memory line. The RdEx Reply message is received in the local cluster by the reply controller, which can then satisfy the read-exclusive request.

To ensure consistency at release points, however the remote-access cache entry is deallocated only when it receives the number of invalidate acknowledgments (Inv-Ack, message 3) equal to an invalidation count sent in the original reply message.

The Dash prototype with 64 nodes was rather small in size. If each processor had a five-issue superscalar operation with a l00-MHz clock, an extended machine with 2K nodes would have the potential to become a system with 1 tera operations per second, with higher performance at higher clock rates.

This demands an integrated implementation with lower overhead in the scalable directory structure. A three-dimensional torus network was considered with 16-bit data paths, a 20-ns fall-through delay, and a 4-ns cycle time. The access time ratio among the four levels of memory hierarchy was to be approximately 1:5:16:80:120, where 1 corresponds to one processor clock. The larger version of DASH was not implemented; however, the concept of distributed directory-based cache coherence was validated.

9.4.2 The Kendall Square Research KSR-1

This was the first commercial attempt to build a scalable multiprocessor with *cache-only memory architecture* (COMA). The Kendall Square Research KSR-1 was a size- and generation-scalable shared-memory multiprocessor computer. It was formed as a hierarchy of "ring multis" as depicted in Fig. 9.26.

The KSR-I Architecture Scalability in the KSR-1 was achieved by connecting 32 processors to form a ring multi (search engine 0 in Fig. 9.26) operating at 1 Gbyte/s (128 million accesses per second). Interconnection bandwidth within a ring scales linearly, since every ring slot has roughly the capacity of a typical crosspoint switch found in a supercomputer that interconnects eight to sixteen 100-Mbytes/s HIPPI channels.

The KSR-1 used a two-level hierarchy to interconnect 34 Ring:0s by a top-level Ring:1 (1088 processors) and was therefore massive. The ring design supported an arbitrary number of levels, permitting ultras to be built (Fig. 9.27).

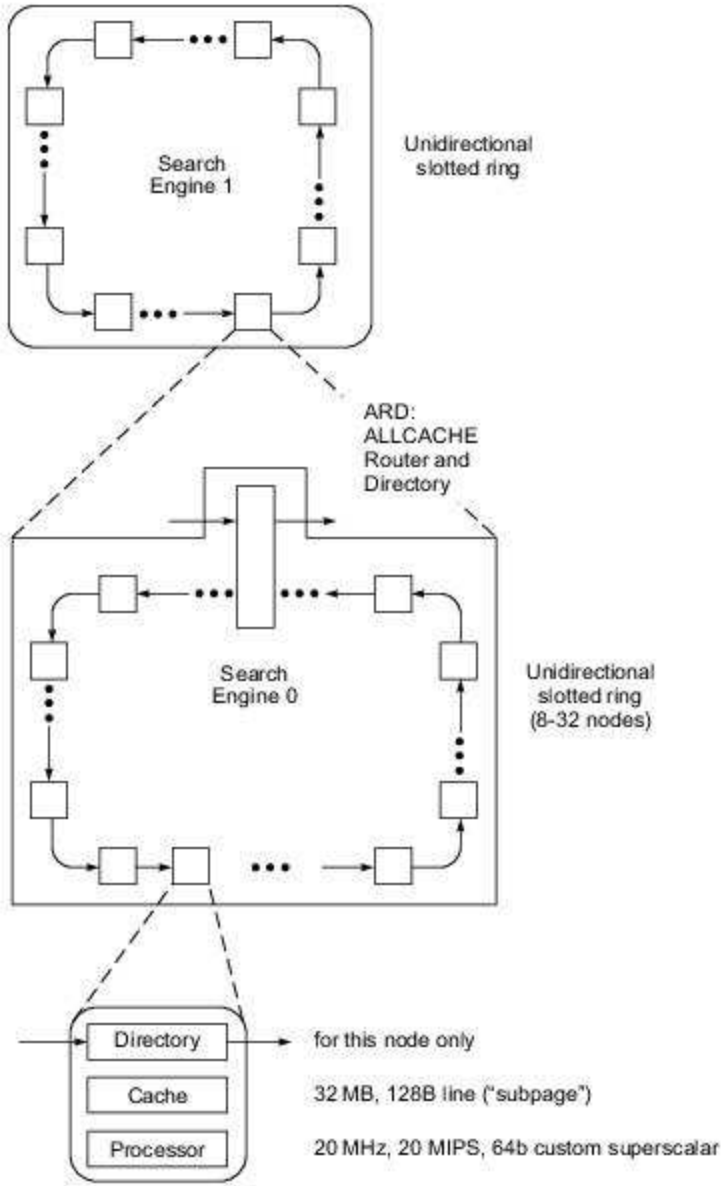


Fig. 9.26 The KSR-1 architecture with a slotted ring for communication (Courtesy of Kendall Square Research Corporation, 1991)

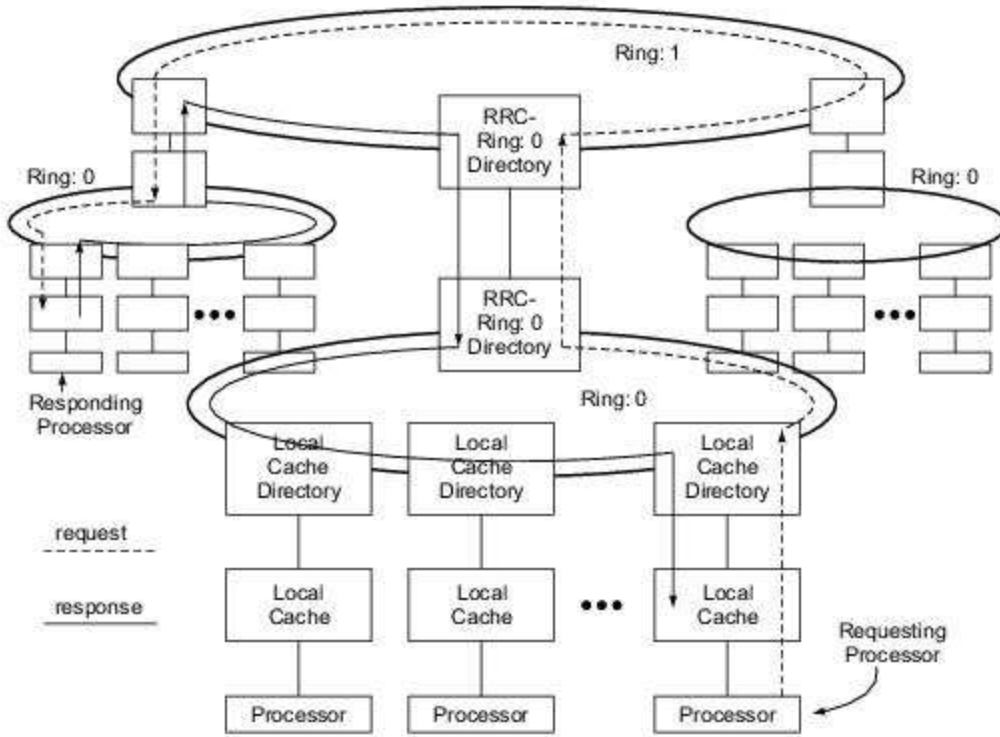


Fig. 9.27 Remote cache (memory) access through two levels of communication rings in the KSR-1 (Courtesy of Kendall Square Research Corporation, 1991)

Each node comprised a primary cache, acting as a 32-Mbyte primary memory, and a 64-bit superscalar processor with roughly the same performance as an IBM RS/6000 operating at the same clock rate. The superscalar processors containing 64 floating-point and 32 fixed-point registers of 64 bits were designed for both scalar and vector operations.

For example, 16 elements could be prefetched at one time. A processor also had a 0.5-Mbyte subcache supplying 20 million accesses per second to the processor (a computational efficiency of 0.5). A processor operated at 20 MHz and was fabricated in 1.2-μm CMOS.

The processor, without caches, contained 3.9 million transistors on 6 types of 12 custom chips. Three-quarters of each processor consisted of the search engine responsible for migrating data to and from other nodes, for maintaining memory coherence throughout the system using distributed directories, and for ring control.

The ALLCACHE Memory The KSR-1 eliminated the memory hierarchy found in conventional computers and the corresponding physical memory addressing overhead. Instead, it offered a single-level memory, called *ALLCACHE* by KSR designers. This ALLCACHE design represented the confluence of cache and shared virtual memory concepts that exploit locality required by scalable distributed computing. Each local cache had a capacity of 32 Mbytes (2^{25} bytes). The global virtual address space had 2^{40} bytes.

Bell (1992) considered the KSR machine the most likely blueprint for future scalable MPP systems. This was a revolutionary architecture and thus was more controversial when it was first introduced in 1991. The architecture provided size (including I/O) and generation scalability in that every node was identical, and it offered an efficient environment for both arbitrary workloads and sequential to parallel processing through a large hardware-supported address space with an unlimited number of processors.

Programming Model The KSR machine provided a strict sequentially consistent programming model and dynamic management of memory through hardware migration and replication of data throughout the distributed processor memory nodes using its ALLCACHE mechanism.

With sequential consistency, every processor returns the latest value of a written value, and results of an execution on multiple processors appear as some interleaving of operations of individual nodes when executed on a multithreaded machine. With ALLCACHE, an address became a name, and this name automatically migrated throughout the system and was associated with a processor in a cache-like fashion as needed.

Copies of a given cell were made by the hardware and sent to other nodes to reduce access time. A processor could prefetch data into a local cache and post-store data for other cells. The hardware was designed to exploit spatial and temporal locality.

For example, in the SPMD programming model, copies of the program moved dynamically and were cached in each of the operating nodes' primary and processor caches. Data such as elements of a matrix moved to the nodes as required simply by accessing the data, and the processor had instructions to prefetch data to the processor's registers. When a processor wrote to an address, all cells were updated and thus memory coherence was maintained. Data movement occurred in subpages of 128 bytes of the 16K pages.

Example 9.6 Multi-ring searching with requesting and responding processors on different Ring: Os (Courtesy of Kendall Square Research Corporation, 1991).

Internode communication for remote memory access was achieved through a searching process. When the requester and responder were in the same Ring:0, the searching was restricted to a single connected Ring:0. Local cache directories showed what addresses could be found in the connected local cache. Each Ring:0 was a unidirectional slotted ring for pipelined searching until the destination was reached.

Figure 9.27 illustrates the situation when the requester and responder resided in different Ring:0s. The top level, Ring:1, consisted entirely of *ring routing cells* (RRCs), each containing a directory for the Ring:0 to which it was connected. Each RRC directory on Ring:1 was essentially a duplicate of the RRC directory on the corresponding Ring:0.

When a packet reached an RRC on Ring:1, it was moved to the next RRC on the ring if the RRC directory indicated that the requested data was not on the corresponding ring. Otherwise, the packet was routed down to the RRC on Ring:0. The packet-passing speed of a Ring:0 was 8 million packets per second. Ring:1 could be configured to handle 8, 16, 32, or 64 million packets per second.

Environment and Performance Every known form of parallelism was supported via the KSR's Mach-based operating system. Multiple users could run multiple sessions comprising multiple applications or multiple processes (each with independent address space), each of which might consist of multiple threads of control running and simultaneously sharing a common address space. Message passing was supported by pointer passing in the shared memory to avoid data copying and enhance performance.

The KSR also provided a commercial programming environment for transaction processing that accessed relational databases in parallel with unlimited scalability as an alternative to multicomputers formed from multiprocessor mainframes. A 1K-node system provided almost two orders of magnitude more processing power, primary memory, I/O bandwidth, and mass storage capacity than a multiprocessor mainframe available at that time.

For example, unlike other contemporary candidates, a 1088-node system could be configured with 15.3 terabytes of disk memory, providing 500 times the capacity of its main memory. The 32- and 320-node systems were designed to deliver over 1000 and 10,000 transactions per second, respectively, giving them over 100 times the throughput of a multiprocessor mainframe available at the time.

With rapid advances in VLSI and interconnect technologies, the mid-1990s saw a major shakeout in the supercomputer business. Kendall Square Research, the developers of KSR-1 and its sequel KSR-2 systems, were forced to exit from hardware business during that period. As in the case of other innovative and pioneering attempts at the development of parallel computer architectures, knowledge gained from the KSR development was also useful in the design and development of MPP computer systems of subsequent generations. Our next case study on MPP system will also bring out clearly this important point.

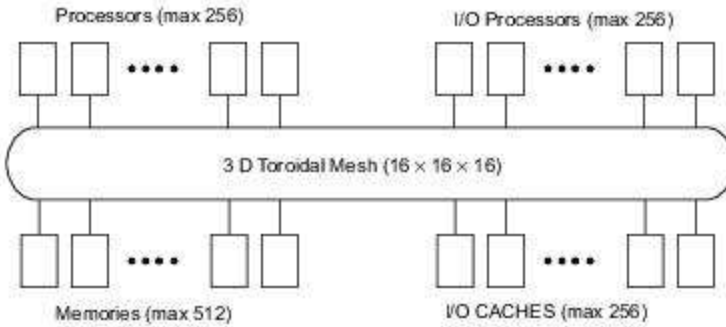
9.4.3 The Tera Multiprocessor System

Multithreaded von Neumann architecture can be traced back to the CDC 6600 manufactured in the mid-1960s. Multiple functional units in the 6600 CPU could execute different operations simultaneously using a score-boarding control. The very first multithreaded multiprocessor was the Denelcor HEP designed by Burton Smith in 1978. The HEP was built with 16 processors driven by a 10-MHz clock, and each processor could execute 128 *threads* (called *processes* in HEP terminology) simultaneously.

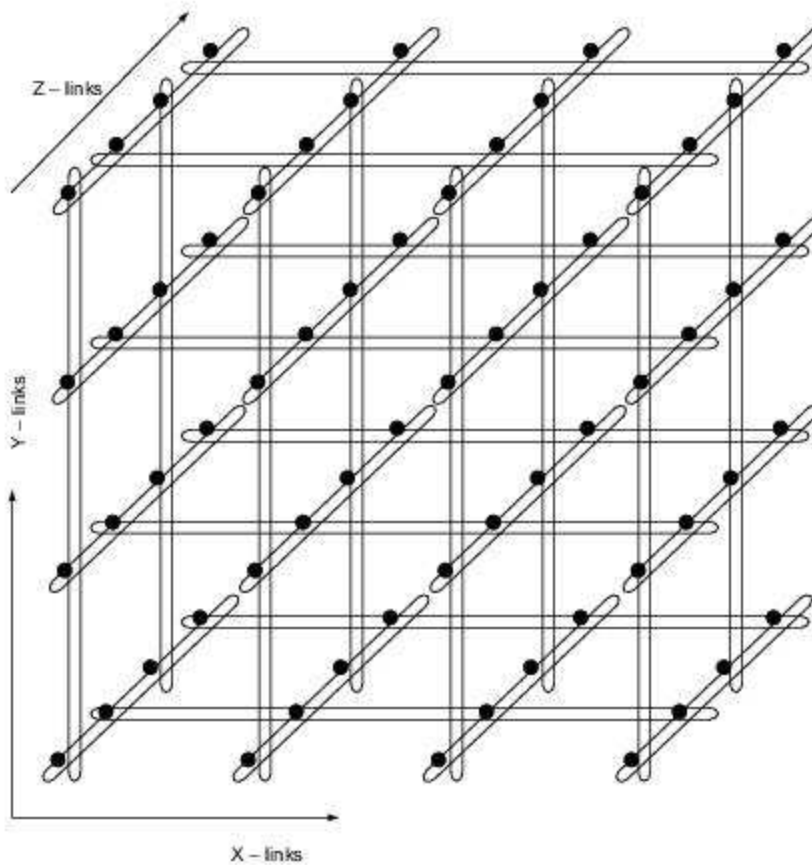
The HEP failed to survive due to inadequate software and compiler support. The Tera was very much a HEP descendant but was implemented with VLSI circuits and packaging technology. A 400-MHz clock was proposed for use in the Tera system, again with a maximum of 128 threads (*i-streams* in Tera terminology) per processor.

In this section, we describe the Tera architecture, its processors and thread state, and the tagged memory/registers. The unique features of the Tera included not only the high degree of multithreading but also the explicit-dependence lookahead and the high degree of pipelining in its processor-network-memory operations. These advanced features were mutually supportive. The first Tera Multithreaded Architecture (MTA) system was delivered in 1998.

The Tera Design Goals The Tera architecture was designed with several major goals in mind. First, it needed to be suitable for very high-speed implementations, i.e. have a short clock period and be scalable to many processors. A maximum configuration of the first implementation of the architecture (Fig. 9.28a) was 256 processors, 512 memory units, 256 I/O cache units, 256 I/O processors, 4096 interconnection network nodes, and a clock period of less than 3 ns.



(a) The Tera computer system



(b) A sparse 4 × 4 × 4 torus with X-links and Y-links missing on alternate Z-layers, respectively

Fig. 9.28 The Tera multiprocessor and its three-dimensional sparse torus architecture shown with a 4 × 4 × 4 configuration (Courtesy of Tera Computer Company, 1992)

Second, it was important that the architecture be applicable to a wide spectrum of problems. Programs that do not vectorize well, perhaps because of a preponderance of scalar operations or too frequent conditional branches, will execute efficiently as long as there is sufficient parallelism to keep the processors busy. Virtually any parallelism applicable in the total computational workload can be turned into speed, from operation-level parallelism within program basic blocks to multiuser time and space sharing.

A third goal was ease of compiler implementation. Although the instruction set did have a few unusual features, they did not pose unduly difficult problems for the code generator. There were no register or memory addressing constraints and only three addressing modes. Condition code setting was consistent and orthogonal.

Because the architecture permitted free exchange of spatial and temporal locality for parallelism, a highly optimizing compiler could improve locality and trade the parallelism thereby saved for more speed. On the other hand, if there was sufficient parallelism, the compiler could exploit it efficiently.

The Sparse Three-Dimensional Torus The interconnection network was a three-dimensional sparsely populated torus (Fig. 9.28b) of pipelined packet-switching nodes, each of which was linked to some of its neighbors. Each link could transport a packet containing source and destination addresses, an operation, and 64 data bits in both directions simultaneously on every clock tick. Some of the nodes were also linked to *resources,* i.e. processors, data memory units, I/O processors, and I/O cache units.

Instead of locating the processors on one side of the network and the memories on the other (a "dance hall" configuration), the resources were distributed more-or-less uniformly throughout the network. This permitted data to be placed in memory units near the appropriate processor when possible, and otherwise generally maximized the distance between possibly interfering resources.

The interconnection network of one 256-processor Tera system contained 4096 nodes arranged in a $16 \times 16 \times 16$ toroidal mesh; i.e. the mesh "wrapped around" in all three dimensions. Of the 4096 nodes, 1280 were attached to the resources comprising 256 cache units and 256 I/O processors. The 2816 remaining nodes did not have resources attached but still provided message bandwidth.

To increase node performance, some of the links were omitted. If the three directions are named x, y, and z, then x-links and y-links were omitted on alternate z-layers (Fig. 9.28b). This reduces the node degree from 6 to 4, or from 7 to 5, counting the resource link. In spite of its missing links, the bandwidth of the network was very large.

Any plane bisecting the network crossed at least 256 links, giving the network a data bisection bandwidth of one 64-bit data word per processor per tick in each direction. This bandwidth was needed to support shared-memory addressing in the event that all 256 processors addressed memory on the other side of some bisecting plane simultaneously.

As the Tera architecture scaled to larger numbers of processors p, the number of network nodes grew as $p^{3/2}$ rather than as the $p \log p$ associated with the more commonly used multistage networks. To see this, we first assume that memory latency is fully masked by parallelism only when the number of messages being routed by the network is at least $p \times l$, where l is the (round-trip) latency. Since messages occupy volume, the network must have a volume proportional to $p \times l$; since the speed of light is finite, the volume is also proportional to l^3 and therefore l is proportional to $p^{1/2}$ rather than $\log p$.

Pipelined Support Each processor in a Tera computer could execute multiple instruction streams (threads) simultaneously. In the initial implementation, as few as 1 or as many as 128 program counters could be active

at once. On every tick of the clock, the processor logic selected a ready-to-execute thread and allowed it to issue its next instruction. Since instruction interpretation was completely pipelined by the processor and by the network and memories as well (Fig. 9.29), a new instruction from a different thread could be issued during each tick without interfering with its predecessors.

When an instruction finished, the thread to which it belonged became ready to execute the next instruction. As long as there were enough threads in the processor so that the average instruction latency was filled with instructions from other threads, the processor was fully utilized. Thus, it was only necessary to have enough threads to hide the expected latency (perhaps 70 ticks on average); once latency was hidden, the processor would run at peak performance and additional threads would not speed the result.

If a thread were not allowed to issue its next instruetion until the previous instruction completed, then approximately 70 different threads would be required on each processor to hide the expected latency. The lookahead described later allowed threads to issue multiple instructions in parallel, thereby reducing the number of threads needed to achieve peak performance.

As seen in Fig. 9.29, three operations could be executed simultaneously per instruction per processor. The *M-pipeline* was for memory-access operations, the *A-pipeline* for arithmetic operations, and the *C-pipeline* for control or arithmetic operations. The instructions were 64 bits wide. If more than one operation in an instruction specified the same register or setting of condition codes, the priority was $M > A > C$.

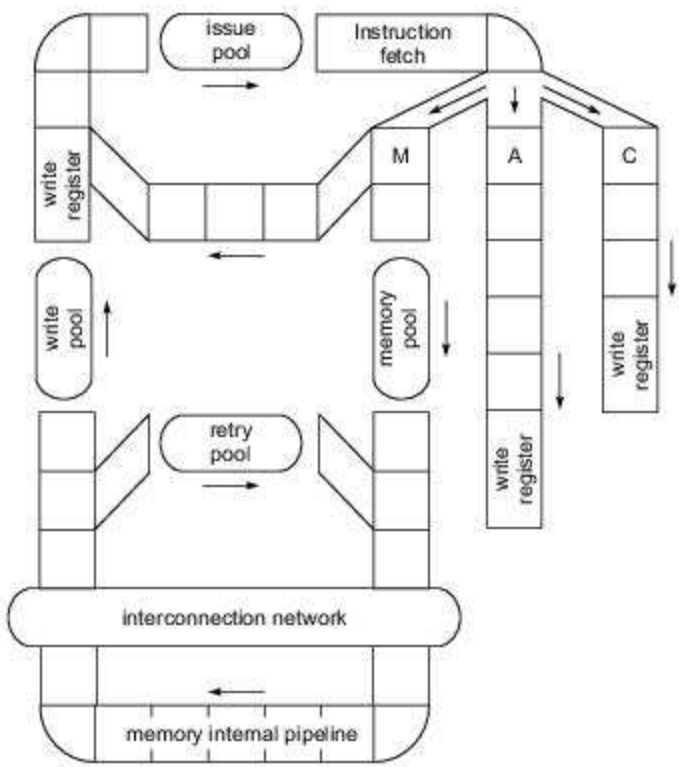
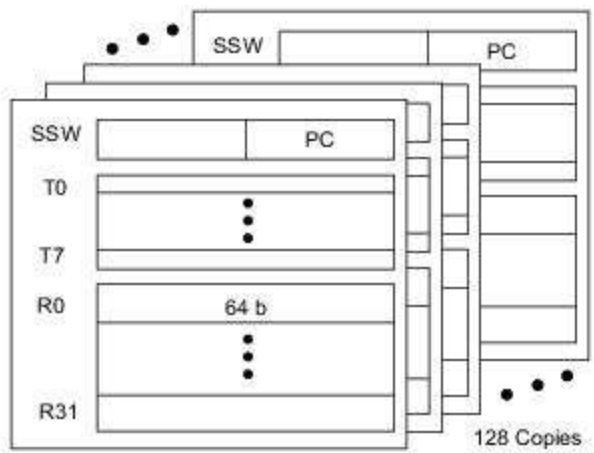


Fig. 9.29 Pipelined processor-network-memory structure (Courtesy of Tera Computer Company, 1992)

It was estimated that a peak speed of 1G operations per second could be achieved per processor if driven by a 333-MHz clock. However, a particular thread would not exceed about 100M operations per second because of interleaved execution. The processor pipeline was rather deep, about 70 ticks, as compared with 8 ticks in the earlier HEP pipeline.

Thread State and Management Figure 9.30 shows that each thread had the following state associated with it:

- One 64-bit stream status word (SSW);
- Thirty-two 64-bit general-purpose registers (R0-R31);
- Eight 64-bit target registers (T0–T7).



Stream Status Word (SSW)
- 32 bit PC (Program Counter)
- Modes (e.g. rounding, lookahead disable)
- Trap disable mask (e.g. data alignment, overflow)
- Condition codes (last four emitted)

No synchronization bits on R0-R31
Target Registers (T0-T7) look like SSWs

Fig. 9.30 The thread management scheme used in the Tera computer (Courtesy of Tera Computer Company, 1992)

Context switching was so rapid that the processor had no time to swap the processor-resident thread state. Instead, it had 128 of everything, i.e. 128 SSWs, 4096 general purpose registers, and 1024 target registers. It is appropriate to compare these registers in both quantity and function to vector registers or words of caches in other architectures. In all three cases, the objective is to improve locality and avoid reloading data.

Program addresses were 32 bits in length. Each thread's current program counter (PC) was located in the lower half of its SSW. The upper half described various modes (e.g. floating-point rounding, lookahead disable), the trap disable mask (e.g. data alignment, floating overflow), and the four most recently generated condition codes.

Most operations had a _TEST variant which emitted a condition code; and branch operations could examine any subset of the last four condition codes emitted and branch appropriately. Also associated with each thread were thirty-two 64-bit general-purpose registers. Register R0 was special in that it read as 0 and output to it was discarded. Otherwise, all general-purpose registers were identical.

The target registers were used as branch targets. The format of the target registers was identical to that of the SSW, though most control transfer operations used only the low 32 bits to determine a new PC. Separating the determination of the branch target address from the decision to branch allowed the hardware to prefetch instructions at the branch targets, thus avoiding delay when the branch decision was made. Using target registers also made branch operations smaller, resulting in tighter loops. There were also skip operations which obviated the need to set targets for short forward branches.

One target register (T0) pointed to the trap handler which was nominally an unprivileged program. When a trap occurred, the effect was as if a coroutine call to a T0 had been executed. This made trap handling extremely lightweight and independent of the operating system. Trap handlers could be changed by the user to achieve specific trap capabilities and priorities without loss of efficiency.

Explicit-Dependence Lookahead If there were enough threads executing on each processor to hide the pipeline latency (about 70 ticks), then the machine would run at peak performance. However, if each thread could execute some of its instructions in parallel (e.g. two successive loads), then fewer threads and parallel activities would be required to achieve peak performance.

The obvious solution was to introduce instruction lookahead; the difficulty was that the traditional register reservation approach requires far too much scoreboard bandwidth in this kind of architecture. Either multithreading or horizontal instruction alone would preclude scoreboarding.

The Tera architecture used a new technique called *explicit-dependence lookahead*. Each instruction contained a 3-bit lookahead field that explicitly specified how many instructions from this thread would be issued before encountering an instruction that depended on the current one. Since seven was the maximum possible lookahead value, at most 8 instructions and 24 operations could be concurrently executing from each thread.

A thread was ready to issue a new instruction when all instructions with lookahead values referring to the new instruction had completed. Thus, if each thread maintained a lookahead of seven, then nine threads were needed to hide 72 ticks of latency.

Lookahead across one or more branch operations was handled by specifying the minimum of all distances involved. The variant branch operations `JUMP_OFTEN` and `JUMP_SELDOM`, for high-and low-probability branches, respectively, facilitated optimization by providing a barrier to lookahead along the less likely path. There were also `SKIP_OFTEN` and `SKIP_SELDOM` operations. The overall approach was conceptually similar to exposed-pipeline lookahead except that the quanta were instructions instead of ticks.

Advantages and Drawbacks The Tera used multiple contexts to hide latency. The machine performed a context switch every clock cycle. Both pipeline latency and memory latency were hidden in the HEP/Tera approach. The major focus was on latency tolerance rather than latency reduction.

With 128 contexts per processor, a large number (2K) of registers must be shared finely between threads. The thread creation must be very cheap (a few clock cycles). Tagged memory and registers with full/empty bits were used for synchronization. As long as there was plenty of parallelism in user programs to hide latency and plenty of compiler support, the performance was potentially very high.

However, these Tera advantages were embedded in a number of potential drawbacks. The performance must be bad for limited parallelism, such as guaranteed low single-context performance. A large number of contexts (threads) demanded lots of registers and other hardware resources which in turn implied higher cost and complexity. Finally, the limited focus on latency reduction and cacheing entailed lots of slack parallelism to hide latency as well as lots of memory bandwidth; both required a higher cost for building the machine.

In the year 1996, the independent company Cray Research, Inc. founded by Seymour Cray merged with the high-performance graphics workstation producer Silicon Graphics, Inc. (SGI); Cray Research then became a business division of SGI. In the year 2000, Tera Computer Company, originators and developers of the Tera MTA massively parallel system which we have studied in this section, took over Cray Research. The merged company was named Cray, Inc., and it is in active operation today (see www.cray.com). Cray has continued with the development of the MTA architecture, as we shall review in Chapter 13.

9.5 DATAFLOW AND HYBRID ARCHITECTURES

Multithreaded architectures can in theory be designed with a pure dataflow approach or with a hybrid approach combining von Neumann and data-driven mechanisms. In this final section, we briefly review the historical development of dataflow computers. Then we consider the design of the ETL/EM-4 in Japan and the prototype design of the MIT/Motorola *T project.

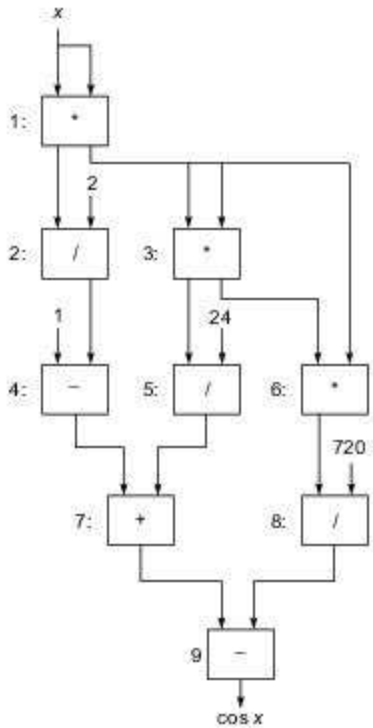
9.5.1 The Evolution of Dataflow Computers

As introduced in Section 2.3, dataflow computers have the potential for exploiting all the parallelism available in a program. Since execution is driven only by the availability of operands at the inputs to the functional units, there is no need for a program counter in this architecture, and its parallelism is limited only by the actual data dependences in the application program. While the dataflow concept offers the potential of high performance, the performance of an actual dataflow implementation can be restricted by a limited number of functional units, limited memory bandwidth, and the need to associatively match pending operations with available functional units.

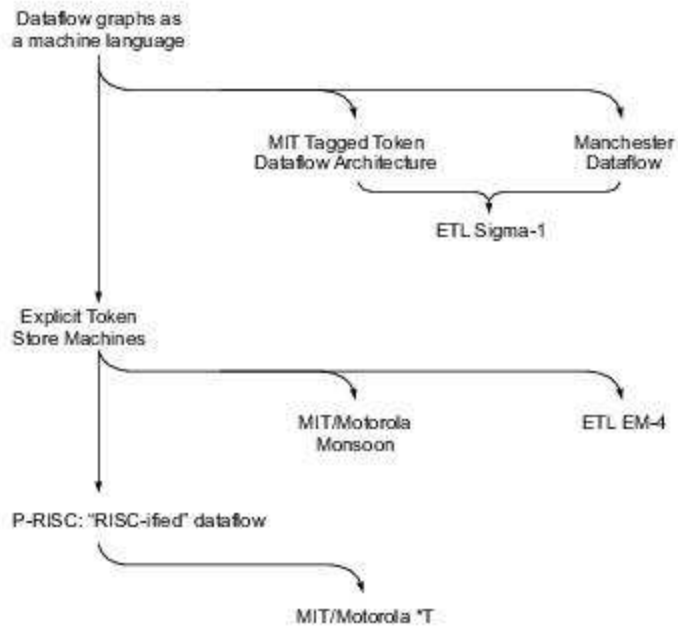
Arvind and Iannucci (1987) identified *memory latency* and *synchronization overhead* as two fundamental issues in multiprocessing. Scalable multiprocessors must address the loss in processor efficiency in these cases. Using various latency-hiding mechanisms and multiple contexts per processor can make the conventional von Neumann architecture relatively expensive to implement, and only certain types of parallelism can be exploited efficiently.

HEP/Tera computers offered an evolutionary step beyond the von Neumann architectures. Dataflow architectures represent a radical alternative to von Neumann architectures because they use dataflow graphs as their machine languages. Dataflow graphs, as opposed to conventional machine languages, specify only a partial order for the execution of instructions and thus provide opportunities for parallel and pipelined execution at the level of individual instructions.

Dataflow Graphs We have seen a dataflow graph in Fig. 2.13. Dataflow graphs can be used as a machine language in dataflow computers. Another example of a dataflow graph (Fig. 9.31a) is given below.



(a) Dataflow graph for computing cos *x* (Courtesy of Arvind)



(b) Evolution tree of dynamic dataflow machines (Courtesy of R. Nikhil)

Fig. 9.31 An example dataflow graph and dataflow machine projects

Example 9.7 The dataflow graph for the calculation of cos*x* (Arvind, 1991).

This dataflow graph shows how to obtain an approximation of cos*x* by the following power series computation:

$$\cos x \simeq 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} \tag{9.6}$$

The corresponding dataflow graph consists of nine operators (actors or nodes). The edges in the graph interconnect the operator nodes. The successive powers of x are obtained by repeated multiplications. The constants (divisors) are fed into the nodes directly. All intermediate results are forwarded among the nodes.

Static versus Dynamic Dataflow *Static dataflow computers* simply disallow more than one token to reside on any one arc, which is enforced by the firing rule: A node is enabled as soon as tokens are present on all input arcs and there is no token on any of its output arcs. Jack Dennis proposed the very first static dataflow computer in 1974.

The static firing rule is difficult to implement in hardware. Special feedback *acknowledge signals* are needed to secure the correct token passing between producing nodes and consuming nodes. Also, the static rule makes it very inefficient to process arrays of data. The number of acknowledge signals can grow too fast to be supported by hardware.

However, static dataflow inspired the development of *dynamic dataflow computers,* which were researched vigorously at MIT and in Japan. In a dynamic architecture, each data token is tagged with a context descriptor, called a *tagged token.* The firing rule of tagged-token dataflow is changed to: A node is enabled as soon as tokens with identical tags are present at each of its input arcs.

With tagged tokens, tag matching becomes necessary. Special hardware mechanisms are needed to achieve this. In the rest of this section, we discuss only dynamic dataflow computers. Arvind of MIT pioneered the development of tagged-token architecture for dynamic dataflow computers.

Although data dependence does exist in dataflow graphs, it does not force unnecessary sequentialization, and dataflow computers schedule instructions according to the availability of the operands. Conceptually, "token"-carrying values flow along the edges of the graph. Values or tokens may be memory locations.

Each instruction waits for tokens on all inputs, consumes input tokens, computes output values based on input values, and produces tokens on outputs. No further restriction on instruction ordering is imposed. No side effects are produced with the execution of instructions in a dataflow computer. Both dataflow graphs and machines implement only functional languages.

Pure Dataflow Machines Figure 9.31b shows the evolution of dataflow computers. The MIT *tagged-token dataflow architecture* (TTDA) (Arvind et al, 1983), the Manchester Dataflow Computer (Gurd and Watson, 1982), and the ETL Sigma-1 (Hiraki and Shimada, 1987) were all pure dataflow computers. The TTDA was simulated but never built. The Manchester machine was actually built and became operational in mid-1982. It operated asynchronously using a separate clock for each processing element with a performance comparable to that of the VAX/780.

The ETL Sigma-1 was developed at the Electrotechnical Laboratory, Tsukuba, Japan. It consisted of 128 PEs fully synchronous with a 10-MHz clock. It implemented the I-structure memory proposed by Arvind. The full configuration became operational in 1987 and achieved a 170-Mflops performance. The major problem in using the Sigma-1 was lack of high-level language for users.

Explicit Token Store Machines These were successors to the pure dataflow machines. The basic idea is to eliminate associative token matching. The waiting token memory is directly addressed, with the use of full/empty bits. This idea was used in the MIT/Motorola Monsoon (Papadopoulos and Culler, 1988) and in the ETL EM-4 system (Sakai et al, 1989).

Multithreading was supported in Monsoon using multiple register sets. Thread-based programming was conceptually introduced in Monsoon. The maximum configuration built consisted of eight processors and eight I-structure memory modules using an 8×8 crossbar network. It became operational in 1991.

EM-4 was an extension of the Sigrna-1. It was designed for 1024 nodes, but only an 80-node prototype became operational in 1990. The prototype achieved 815 MIPS in an 80×80 matrix multiplication benchmark. We will study the details of EM-4 in Section 9.5.2.

Hybrid and Unified Architectures These are architectures combining positive features from the von Neumann and dataflow architectures. The best research examples include the MIT P-RISC (Nikhil and Arvind, 1988), the IBM Empire (Iannucci et al., 1991), and the MIT/Motorola *T (Nikhil, Papadopoulos, Arvind, and Greiner, 1991).

P-RISC was a "RISC-ified" dataflow architecture. It allowed tighter encodings of the dataflow graphs and produced longer threads for better performance. This was achieved by splitting "complex" dataflow instructions into separate "simple" component instructions that could be composed by the compiler. It used traditional instruction sequencing. It performed all intraprocessor communication via memory and implemented "joins" explicitly using memory locations.

P-RISC replaced some of the dataflow synchronization with conventional program counter-based synchronization. IBM Empire was a von Neumann/dataflow hybrid architecture under development at IBM based on the thesis of Iannucci (1988). The *T was a latter effort at MIT joining both the dataflow and von Neumann ideas, to be discussed in Section 9.5.3.

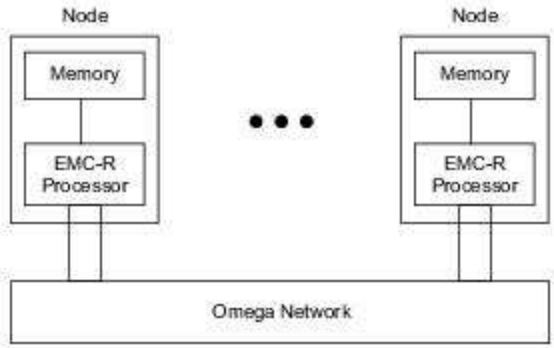
9.5.2 ETL/EM-4 in Japan

EM-4 had the overall system organization as shown in Fig. 9.32a. Each EMC-R node was a single-chip processor without floating-point hardware but including a switch of the network. Each node played the role of I-structure memory and had 1.31 Mbytes of static RAM. An Omega network was used to provide interconnections among the nodes.

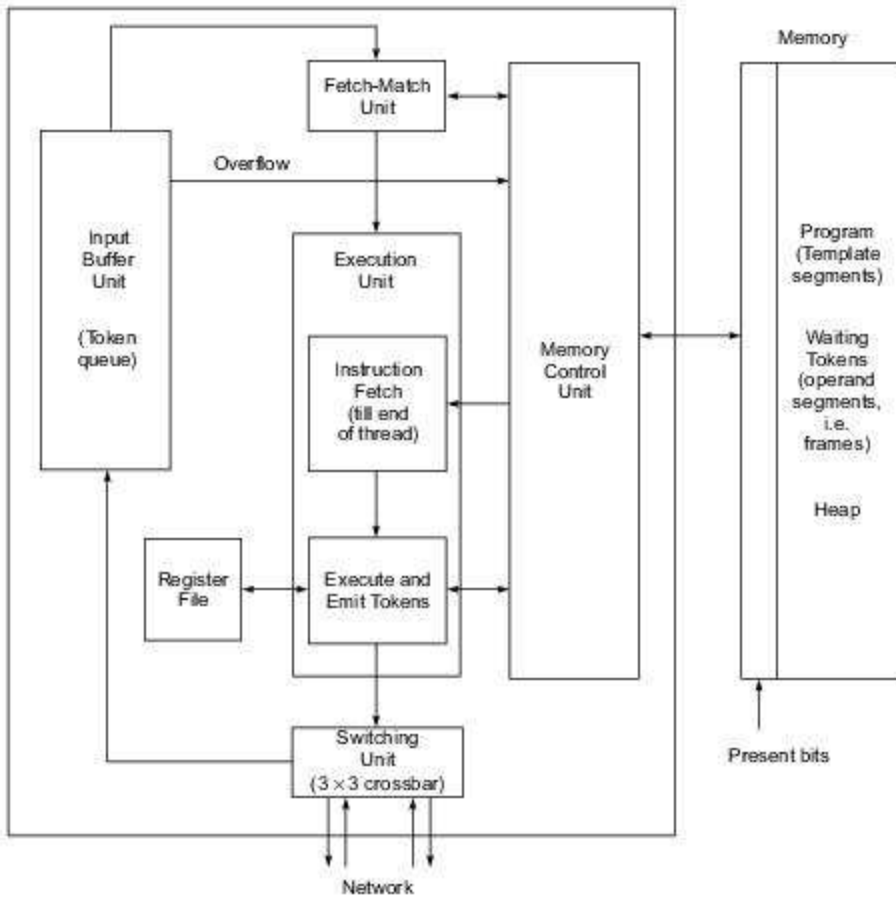
The Node Architecture The internal design of the processor chip and of the node memory are shown in Fig. 9.32b. The processor chip communicated with the network through a 3×3 crossbar *switch unit.* The processor and its memory were interfaced with *a memory control unit.* The memory was used to hold programs (template segments) as well as tokens (operand segments, heaps, or frames) waiting to be fetched.

The processor consisted of six component units. The *input buffer* was used as a token store with a capacity of 32 words. The *fetch-match unit* fetched tokens from the memory and performed tag-matching operations among the tokens fetched in. Instructions were directly fetched from the memory through the memory controller.

The heart of the processor was the *execution unit,* which fetched instructions until the end of a thread. Instructions with matching tokens were executed. Instructions could emit tokens or write to registers. Instructions were fetched continually using traditional sequencing (PC + 1 or branch) until a "stop" flag was raised to indicate the end of a thread. Then another pair of tokens was accepted. Each instruction in a thread specified the two sources for the next instruction in the thread.



(a) Global organization



(b) The EMC-R processor design

Fig. 9.32 The ETL EM-4 dataflow architecture (Courtesy of Sakai, Yamaguchi et al, Electrotechnical Laboratory, Tsukuba, Japan, 1991)

The same idea was used as in Monsoon for token matching, but with different encoding. All data tokens were 32 bits, and instruction words were 38 bits. EM-4 supported remote loads and synchronizing loads. The *full/empty* bits present in memory words were used to synchronize remote loads associated with different threads.

9.5.3 The MIT/Motorola *T Prototype

The *T project was a direct descendant of a series of MIT dynamic dataflow architectures unifying with the von Neumann architectures. In this final section, we describe *T, a prototype multithreaded MPP system based on the work of Nikhil, Papadopoulos, and Arvind of MIT in collaboration with Greiner and Traub of Motorola. Finally, we compare the dataflow and von Neumann perspectives in building fine-grain, massively parallel systems.

The Prototype Architecture The *T prototype was a single-address-space system. A "brick" of 16 nodes was packaged in a 9-in cube (Fig. 9.33a). The local network was built with 8×8 crossbar switching chips. A brick had the potential to achieve 3200 MIPS or 3.2 Gflops. The memory was distributed to the nodes. One gigabyte of RAM was used per brick. With 200-Mbytes/s links, the I/O bandwidth was 6.4 Gbytes/s per brick.

A 256-node machine could be built with 16 bricks as illustrated in Fig. 9.33b. The 16 bricks were interconnected by four switching boards. Each board implemented a 16×16 crossbar switch. The entire system could be packaged into a 1.5-m cube. No cables were used between the boards. The package was limited by connector-pin density. The 256-node machine had the potential to achieve 50,000 MIPS or 50 Gflops. The bisection bandwidth was 50 Gbytes/s.

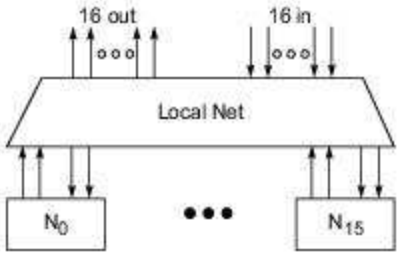
The *T Node Design Each node was designed to be implemented with four component units. A Motorola superscalar RISC microprocessor (MC88110) was modified as a *data processor* (dP). This dP was optimized for long threads. Concurrent integer and floating-point operations were performed within each dP.

A *synchronization coprocessor* (sP) was implemented as an 88000 special-function unit (SFU), which was optimized for simple, short threads. Both the dP and the sP could handle fast loads. The dP handled incoming continuation, while the sP handled incoming messages, rload/rstore responses, and joins for messaging or synchronization purposes. In other words, the sP off-loaded simple message-handling tasks from the main processor (the dP). Thus the dP would not be disrupted by short messages.

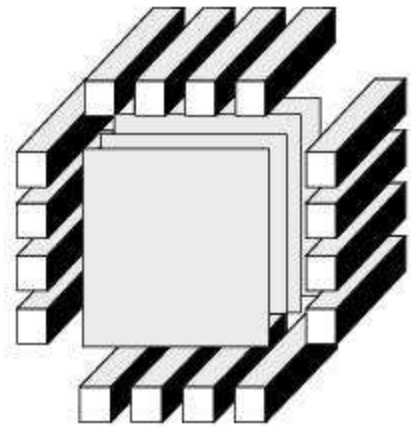
The *memory controller* handled requests for remote memory load or store, as well as the management of node memory (64 Mbytes). The *network interface unit* received or transmitted messages from or to the network, respectively, as illustrated in Fig. 9.33c. It should be noted that the sP was built as an on-chip SFU of the dP.

The MC 88110 family allowed additional on-chip SFUs, with reserved opcode space, common instruction-issue logic and caches, etc., and direct access to processor registers. Example SFUs included the floating-point unit, graphics unit, coprocessor, etc. The MC 88110 was itself a two-way superscalar processor driven by a 50-MHz clock.

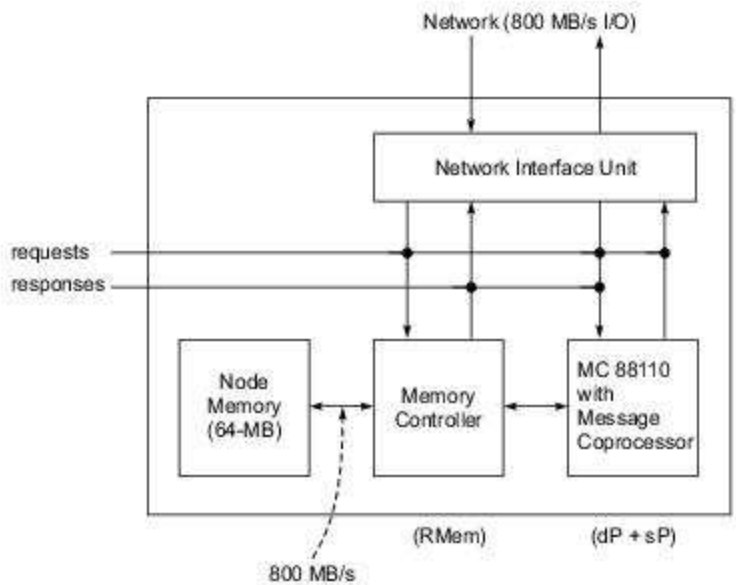
New SFUs were added into the MC 88110 to provide 16 buffers for incoming messages and 4 buffers for outgoing messages. Other SFUs included a *continuation stack* with 64 entries and a *microthreaded scheduler*, which supplied continuations from messages and the continuation stack, etc. Special instructions were available for packing or unpacking continuations.



(a) A brick of 16 nodes with 3.2-Gflops and 3200-MIPS peak performance, packaged in a 9-in cube



(b) A 256-node machine consisting of 16 bricks interconnected by 4 boards of 16 × 16 switches and packaged in a 1.5-m cube



(c) Interior node architecture with data processor (MC 88110) and synchronization coprocessor (sP)

Fig. 9.33 The MIT/Motorola *T prototype multithreaded architecture (Courtesy of Nikhil, Papadopoulos, and Arvind, *Proc. 19th Int. Symp. Computer Arch.*, Australia, May 1992)

Research Experiments The *T prototype was used to test the effectiveness of the unified architecture in supporting multithreading operations. The development of *T was influenced by other multithreaded architectures, including Tera, Alewife, and J-Machine.

The I-structure semantics was also implemented in *T. Full/empty bits were used on producer-consumer variables. *T treated messages as virtual continuations. Thus busy-waiting was eliminated. Other optimizations in *T included speculative avoidance of the extra loads and stores through multithreading and coherent cacheing.

The *T designers wanted to provide a superset of the capabilities of Tera, J-Machine, and EM-4. Compiler techniques developed for these machines were expected to be applicable to *T. To achieve these goals, a promising approach was to start with declarative languages while the compiler could aim to extract a large amount of fine-grain parallelism.

Multithreading: A Perspective The Dash, KSR-1, and Alewife leveraged existing processor technology. The advantages of these directory-based cacheing systems include compatibility with existing hardware and software. But they offer a less aggressive pursuit of parallelism and depend heavily on compilers to obtain locality. The synchronizing loads are still problematic in these distributed cacheing solutions.

In von Neumann multithreading approaches, the HEP/Tera replicated the conventional instruction stream. Synchronizing-loads problems were solved by a hardware trap and software. Hybrid architectures, such as Empire, replicated conventional instruction streams, but they did not preserve registers across threads. The synchronizing loads were entirely supported in hardware. J-Machine supported three instruction streams (priorities). It grew out of message-passing machines but added support for global addressing. Remote synchronizing loads were supported by software convention.

In the dataflow approaches, the system-level view has stayed constant from the Tagged-Token Dataflow Architecture to the *T. The various designs differ in internal node architecture, with trends toward the removal of intra-node synchronization, using longer threads, high-speed registers, and compatibility with existing machine codes. The *T designers claimed that the unification of dataflow and von Neumann ideas would support a scalable shared-memory programming model using existing SIMD/SPMD codes.

Summary

Computer systems have always operated with processors having much faster cycle times than main memories. With steady advances in VLSI technology over the years, both processors and main memories have become faster, but the relative speed mismatch between them has in fact widened over the years. *Latency hiding techniques* are therefore devised to allow processors to operate at high efficiency in spite of having to access slower memories from time to time; use of cache memories is a common latency hiding technique. In the context of Massively Parallel Processing (MPP) systems, other technical challenges also confront system designers in minimizing the impact of memory access latencies.

In this chapter, we studied some basic latency hiding techniques applicable to such systems, namely: shared virtual memory with some specific examples; prefetching techniques and their effectiveness; and the use of distributed coherent caches. Scalable Coherent Interface (SCI) provides cache coherence with distributed directories and sharing lists. We studied several relaxed memory consistency models which can permit greater exploitation of parallelism in applications; the impact of relaxed consistency models while running three specific applications was presented.

Principles of multi-threading were introduced, with specific attention paid to the technical factors relevant to system design, namely: communication latency on remote access, number of threads, context-switching overhead, and the interval between context switches. Multiple context processors have been designed to provide hardware support for single cycle context switching. Possible context-switching policies were studied, along with their impact on system efficiency. Multidimensional architectures were reviewed as a possible platform for multi-threaded systems.

Fine-grain multicomputers are specially designed to provide efficient support for fine-grain parallelism in applications. The MIT J-machine was studied from the points of view of its overall system design, its Message-Driven Processor (MDP) and instruction set architecture, and the message format and routing employed in its 3-dimensional mesh. The design goal of Caltech Mosaic C system was to exploit the advances which had taken place in VLSI and packaging technologies; we studied the basic node design with its two contexts (for user program and message handler), and basic 8×8 mesh design employed in the system.

In the category of scalable multithreaded architectures, the Stanford Dash multiprocessor system utilized directory-based cache coherence in a single address-space distributed memory system. Kendall Square Research KSR-1 system employed a cache-only memory design with a ring-based interconnect. The Tera multiprocessor system relied for its performance on a large degree of multi-threading and aggressive use of pipelining throughout the system, with a sparse 3-dimensional torus interconnect.

We also studied the basic concepts and evolution of dataflow and hybrid architectures, from the first introduction of the concept in 1974 by Jack Dennis at MIT. Specific dataflow and hybrid systems studied in this context were the ETL/EM-4 system developed in Japan, and the MIT/Motorola *T prototype system.

Exercises

Problem 9.1 Consider a scalable multiprocessor with p processing nodes and distributed shared memory. Let R be the rate of each processing node generating a request to access remote memory through the interconnection network. Let L be the average latency for remote memory access. Derive expressions for the processor efficiency E under each of the following conditions:

(a) The processor is single-threaded, uses only a private cache, and has no other latency-hiding mechanisms. Express E as a function of R and L.

(b) Suppose a coherent cache is supported by hardware with proper data sharing and h is the probability that a remote request can be satisfied by a local cache. Express E as a function of R, L, and h.

(c) Now assume each processor is multithreaded to handle N contexts simultaneously. Assume a context-switching overhead of C. Express E as a function of N, R, L, h, and C.

(d) Now consider the use of a 2-D $r \times r$ torus with $r^2 = p$ and bidirectional links. Let t_d be the time delay between adjacent nodes and t_m be the local memory-access time. Assume that the network is fast enough to respond to each request without buffering. Express the latency L as a function of p, t_d and t_m. Then express the efficiency E as a function of N, R, h, C, p, t_d and t_m.

Problem 9.2 The following two questions are related to the effect of prefetching on latency tolerance:

(a) Perform an analytical study of the effects of data prefetching on the performance (efficiency) of processors in a scalable multiprocessor system without multithreading.

(b) Repeat part (a) for a multithreaded multiprocessor system under reasonable assumptions.

Problem 9.3 The following questions are related to the effects of memory consistency models:

(a) Perform an analytical study of the effects of using a relaxed consistency memory model in a scalable multiprocessor without multithreading.

(b) Repeat part (a) for a multithreaded multiprocessor system under reasonable assumptions.

(c) Can you derive an efficiency expression for a multiple-context processor supported by both prefetching and release memory consistency?

Problem 9.4 Consider a two-dimensional multicube architecture with m row buses and m column buses (Fig: 9.18a). Each bus has a bandwidth of B bits/s. The bus is considered active when it is actually in progress. The bus utilization rate a ($0 < a \leq 1$) is defined as the number of active bus cycles over the total cycles elapsed. The per-processor request rate r is defined as the number of requests that a processor sends on either of the two buses (for the purpose of memory access, cache coherence, synchronization, etc.) per second.

(a) Consider a single-column bus with associated processors and memory module and express the bus bandwidth as a function of m, a, and r.

(b) What is the total bus bandwidth available in the entire system?

(c) If r is kept constant as the number of processors increases, how many requests can be sent to the system without exceeding the limit?

(d) Each request goes through a maximum of two buses in the multicube. What bus bandwidth will be needed to satisfy all the requests?

(e) In parts (b) and (d), does the multicube provide enough bus bandwidth? Justify the answer with reasoning.

Problem 9.5 Consider the use of an orthogonal multiprocessor consisting of 4 processors and 16 orthogonally shared memory modules (Fig. 9.18b) to perform an unfolded multiplication of two 8×8 matrices in a partitioned SPMD mode.

(a) Show how to distribute the 2×2 submatrices of the input matrix $A = (a_{ij})$ and $B = (b_{ij})$ to the 16 orthogonally shared memory modules.

(b) Specify the SPMD algorithm by involving all four processors in a synchronized manner to access either the row memories or the column memories. Synchronization is handled at the loop level.

You can assume the use of a pipeline-read to fetch either one column or one row vector of the input matrix A or B at a time, and a pipeline-write to store the product matrix $C = A \times B = (c_{ij})$ elements in a similar fashion. Assume that sufficient large register windows are available within each processor to hold all 2×2 submatrix elements. Each processor can perform inner product operations.

(c) Let $N \times N$ be the matrix size and $k = N/n$ the partitioned block size in mapping a large matrix in the orthogonal memory. Estimate the number of orthogonal memory accesses and the number of synchronizations needed in an SPMD algorithm for multiplying two $N \times N$ matrices on an n-processor OMP.

(d) Repeat the above for a two-dimensional fast Fourier transform over $N \times N$ sample points on an n-processor OMP, where $N = n \cdot k$ for some integer $k \geq 2$. The idea of performing a two-dimensional FFT on an OMP is to perform a one-dimensional FFT along one dimension in a row-access mode.

All n processors then synchronize, switch to a column-access mode, and perform another one-dimensional FFT along the second dimension. First try the case where $N = 8$, $n = 4$, and $k = 2$ and then work out the general case for large $N \gg n$.

Problem 9.6 The following questions are related to shared virtual memory:

(a) Why has shared virtual memory (SVM) become a necessity in building a scalable system with memories physically distributed over a large number of processing nodes?

(b) What are the major differences in implementing SVM at the cache block level and the page level?

Problem 9.7 The release consistency (RC) model has combined the advantages of both the processor consistency (PC) and the weak consistency (WC) models. Answer the following questions related to these consistency models:

(a) Compare the implementation requirements in the three consistency models.

(b) Comment on the advantages and shortcomings of each consistency model.

Problem 9.8 Answer the following questions involving the MIT J-Machine:

(a) What were the unique features of the message-driven processors (MDP) making it suitable for building fine-grain multicomputers?

(b) Explain the E-cube routing mechanism built into the MDP.

(c) Explain the concept of using a combining tree for synchronization of events on various nodes in the J-Machine.

Problem 9.9 Why are hypercube networks (binary *n-cube* networks), which were very popular in first-generation multicomputers, being replaced by 2D or 3D meshes or tori in the second and third generations of multicomputers?

Problem 9.10 Answer the following questions on the SCI standard:

(a) Explain the sharing-list creation and update methods used in the IEEE Scalable Coherence Interface (SCI) standard.

(b) Comment on the advantages and disadvantages of chained directories for cache coherence control in large-scale multiprocessor systems.

Problem 9.11 Compare the four context-switching policies: switch on cache miss, switch on every load, switch on every instruction (cycle by cycle), and switch on block of instructions.

(a) What are the advantages and shortcomings of each policy?

(b) What additional research would be needed to make an optimal choice among these policies?

Problem 9.12 After studying the Dash memory hierarchy and directory protocol, answer the following questions with an analysis of potential performance:

(a) Define the cache states used in Dash.

(b) How were the cache directories implemented in the memory hierarchy?

(c) Explain the Dash directory-based coherence protocol when reading a remote cache block that is dirty in a remote cluster.

(d) Repeat part (c) for the case of writing to a shared remote cache block.

Problem 9.13 Answer the following questions on multiprocessors:

(a) Describe the ALLCACHE architecture implemented in the Kendall Square Research KSR-1.

(b) Explain how cache coherence can be maintained in the KSR-1.

(c) Study the papers on COMA architectures by Stenström et al (1992) and Hagersten et al (1990). Compare the differences between KSR-1 and the Data Diffusion Machine (DDM) architecture.

Problem 9.14 Answer the following questions on the development of the Tera computer.

(a) What were the design goals of the Tera computer?

(b) Explain the sparse 3D torus used in Tera. What are the advantages of the sparse structure?

(c) Explain how pipelining is applied in supporting the multithreaded operations in each Tera processor.

(d) Explain the thread state and management scheme used in Tera.

(e) Explain the idea of explicit-dependence lookahead and its effects on multithreading in Tera.

(f) What are the contributions of the Tera architecture and software development? Compare the advantages and potential drawbacks of the Tera computer.

Problem 9.15 Answer the following questions related to dataflow computers:

(a) Distinguish between static dataflow computers and dynamic dataflow computers.

(b) Draw a dataflow graph showing the computations of the roots of a sequence of quadratic equations $A_i x_i^2 + B_i x_i + C_i = 0$ for $i = 1, 2, ..., N$.

(c) Consider the parallel execution of the successive root computations with a four-PE tagged-token dataflow computer (Fig. 2.12). Show a minimum-time schedule for using the four PEs to compute the N pairs of roots.

Problem 9.16 Consider the mapping of a one-dimensional circular convolution computation on a multiprocessor with 4 processors and 32 memory modules which are 32-way interleaved for pipelined access of vector data. Assume no contention between processors and memories in the interconnection network. The one-dimensional convolution is defined over a $1 \times n$ image and a $1 \times m$ kernel as follows:

$$Y(i) = \sum_{j=0}^{m-1} W(j) \cdot X((i-j) \bmod n) \text{ for } 0 \le i \le n-1$$

(a) How many multiplications and additions are involved in the above computations? Map the image pixels $X(i)$ to memory module M_j if $j = i \pmod{32}$ and assume $n = 256$. The output image $Y(i)$ is also stored in module M_j if $j = i \pmod{32}$ for $0 \le i \le 255$. The kernel is also stored in a similar manner. Assume $m = 4$ and each processor handles the computation of one output image.

(b) Show how to partition the computations among the four processors such that minimum time is spent in both memory-access and CPU executions. Assume no memory conflicts and up to four fetch or store operations (but not mixed) performed at the same time. The interleaved memory can be accessed by one or more processors at the same time.

(c) What is the minimum execution time (including both memory and CPU operations) if each multiply and add and each interleaved memory access is considered one time unit. Assume enough working registers are available in each CPU.

(d) What is the speedup factor of the above multiprocessor solution over a uniprocessor solution? You can make similar assumptions about the use of the 32-way interleaved memory for both uniprocessor and multiprocessor configurations.

Problem 9.17 Answer the following questions on fine-grain multicomputers and massive parallelism:

(a) Why are fine-grain processors chosen for

research-oriented multicomputers and MPP systems over medium-grain processors used in the past?

(b) Why is a single global addressing space desired over distributed address spaces?

(c) From scalability point of view, why is fine-grain parallelism more appealing than medium-grain or coarse-grain parallelism for building MPP systems?