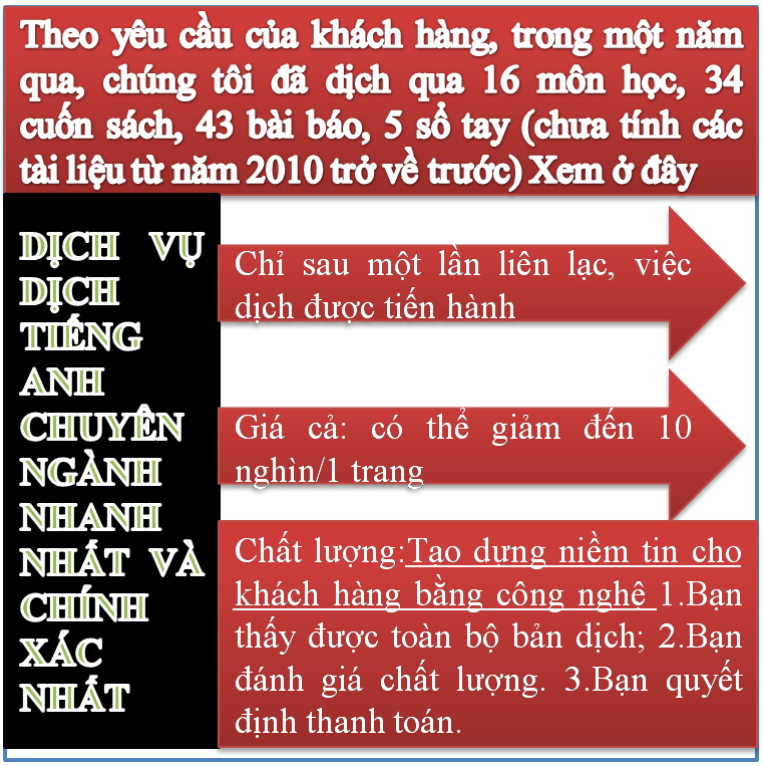


Tài liệu này được dịch sang tiếng việt bởi:

www.mientayvn.com

Tìm bản gốc tại thư mục này (copy link và dán hoặc nhấn Ctrl+Click):

https://drive.google.com/folderview?id=0B4rAPqlxIMRDSFE2RXQ2N3FtdDA&usp=sharing

Liên hệ để mua:

thanhlam1910_2006@yahoo.com hoặc frbwrthes@gmail.com hoặc số 0168 8557 403 (gặp Lâm)

Giá tiền: 1 nghìn /trang đơn (trang không chia cột); 500 VND/trang song ngữ

Dịch tài liệu của bạn: http://www.mientayvn.com/dich_tieng_anh_chuyen_nghanh.html

Introduction to Advanced Computer Architecture and Parallel Processing Computer architects have always strived to increase the performance of their computer architectures. High performance may come from fast dense circuitry, packaging technology, and parallelism. Single-processor supercomputers have achieved unheard of speeds and have been pushing hardware technology to the physical limit of chip manufacturing. However, this trend will soon come to an end, because there are physical and architectural bounds that limit the computational power that can be achieved with a single-processor system. In this book we will study advanced computer architectures that utilize parallelism via multiple processing units. Parallel processors are computer systems consisting of multiple processing units connected via some interconnection network plus the software needed to make the processing units work together. There are two major factors used to categorize such systems: the processing units themselves, and the interconnection network that ties them together. The processing units can communicate and interact with each other using either shared memory or message passing methods. The interconnection network for shared memory systems can be classified as bus-based versus switch-based. In	Giới thiệu kiến trúc máy tính tiên tiến và xử lý song song (đã sửa) Các kỹ sư máy tính luôn cố gắng tăng hiệu suất kiến trúc máy tính của họ (Những nhà thiết kế máy tính luôn cố gắng tăng hiệu năng kiến trúc máy tính của họ). Hiệu suất cao có thể nhờ vào mức độ tích hợp mạch nhanh, công nghệ đóng gói, và công nghệ xử lý song song. Siêu máy tính đơn xử lý (một bộ vi xử lý) đã đạt được tốc độ chưa từng có và đã đẩy công nghệ phần cứng tới giới hạn vật lý của nó ở phương diện sản xuất chip (đã đẩy công nghệ sản xuất chip đến giới hạn của nó). Tuy nhiên, xu hướng này sẽ sớm kết thúc, bởi vì các giới hạn vật lý và kiến trúc làm hạn chế công suất tính toán đạt được với hệ đơn xử lý. Trong sách này, chúng ta sẽ nghiên cứu các kiến trúc máy tính tiên tiến sử dụng cơ chế song song thông qua các bộ đa xử lý. Các bộ xử lý song song là các hệ thống máy tính bao gồm nhiều đơn vị xử lý được kết nối thông qua mạng liên thông và các phần mềm cần thiết để các đơn vị xử lý làm việc cùng nhau" Có hai yếu tố chính được sử dụng để phân loại các hệ thống là: chính các đơn vị xử lý này, và mạng liên thông gắn kết chúng với nhau. Các đơn vị xử lý có thể giao tiếp và tương tác với nhau bằng cách sử dụng bộ nhớ chung hoặc các phương pháp truyền

message passing systems, the interconnection network is divided into static and dynamic. Static connections have a fixed topology that does not change while programs are running. Dynamic connections create links on the fly as the program executes. The main argument for using multiprocessors is to create powerful computers by simply connecting multiple processors. A multiprocessor is expected to reach faster speed than the fastest single-processor system. In addition, a multiprocessor consisting of a number of single processors is expected to be more cost-effective than building a high-performance single processor. Another advantage of a multiprocessor is fault tolerance. If a processor fails, the remaining processors should be able to provide continued service, albeit with degraded performance. 1.1 FOUR DECADES OF COMPUTING Most computer scientists agree that there have been four distinct paradigms or eras of computing. These are: batch, time-sharing, desktop, and network. Table 1.1 is modified from a table proposed by Lawrence Tesler. In this table, major characteristics of the different computing paradigms are associated with each decade of computing, starting from 1960.	tin. Mạng liên thông của các hệ thống bộ nhớ dùng chung có thể thuộc loại dựa trên bus và dựa trên chuyển mạch. Trong các hệ thống truyền tin, mạng liên thông được chia thành tĩnh và động. Các kết nối tĩnh có một tôpô cố định không thay đổi khi chương trình đang chạy. Các kết nối động tạo ra các liên kết tuỳ biến khi chương trình thực thi. Mục đính chính của việc dùng các bộ đa xử lý là tạo ra các máy tính mạnh bằng sự kết nối đơn giản nhiều bộ xử lý. Người ta cho rằng một bộ đa xử lý sẽ đạt tốc độ nhanh hơn so với hệ thống đơn xử lý nhanh nhất. Ngoài ra, việc phát triển một bộ đa xử lý bao gồm nhiều bộ đơn xử lý cũng có thể hiệu quả hơn về chi phí so với việc phát triển một hệ đơn xử lý hiệu suất cao. Một ưu điểm khác của bộ đa xử lý là khả năng kháng lỗi. Nếu một bộ xử lý trong hệ gặp sự cố, các bộ xử lý còn lại vẫn sẽ hoạt động bình thường, mặc dù hiệu suất suy giảm. 1.1 Tổng quan về máy tính qua 4 thập kỷ Hầu hết các nhà khoa học máy tính đều nhất trí rằng đã có bốn mô hình hay kỷ nguyên điện toán khác nhau. Đó là: máy tính (xử lý) bó, máy tính phân hưởng thời gian, destop, và mạng. Bảng 1.1 được sửa đổi từ bảng được Lawrence Tesler đề xuất. Trong bảng này, các đặc điểm chính của các mô hình điện toán khác nhau được trình bày ứng với mỗi

	thập kỷ điện toán, bắt đầu từ năm 1960.
	1.1.1 Kỷ nguyên máy tính (xử lý) bó
1.1.1 Batch Era	Năm 1965, máy tính lớn thuộc hệ IBM/360 thống trị trong các công ty. Đây là một loại máy tính xử lý theo lô điển hình với các bộ đọc thẻ đục lỗ, băng và ổ đĩa, nhưng không thể trao đổi dữ liệu vượt ra ngoài phạm vi phòng máy. Chúng hình thành nên các máy tính tập trung lớn và là một dạng tiêu chuẩn của máy tính trong nhiều thập kỷ. Hệ IBM/360 có một hệ điều hành, nhiều ngôn ngữ lập trình, và dung lượng đĩa là 10 MB. Các hộp kim loại của Hệ IBM/360 và những người vận hành chúng chiếm trọn một căn phòng. Tốc độ hoạt động của các mạch transistor bên trong cũng không nhanh lắm. Những người sử dụng thành thạo có thể đặt mua các bộ nhớ lõi từ lên đến 1 MB từ 32-bit. Máy tính này đã đủ lớn để hỗ trợ cùng lúc nhiều chương trình trong bộ nhớ, mặc dù bộ xử lý trung tâm phải chuyển từ một chương trình này sang chương trình khác.
By 1965 the IBM System/360 mainframe dominated the corporate computer centers. It was the typical batch processing machine with punched card readers, tapes and disk drives, but no connection beyond the computer room. This single mainframe established large centralized computers as the standard form of computing for decades. The IBM System/360 had an operating system, multiple programming languages, and 10 megabytes of disk storage. The System/360 filled a room with metal boxes and people to run them. Its transistor circuits were reasonably fast. Power users could order magnetic core memories with up to one megabyte of 32-bit words. This machine was large enough to support many programs in memory at the same time, even though the central processing unit had to switch from one program to another.	
	1.1.2 Kỷ nguyên máy tính phân hưởng thời gian
1.1.1 Time-Sharing Era	Các máy tính lớn của thời kỳ máy tính xử lý theo khối đã được đặt nền móng vững chắc vào cuối những năm 1960. Cũng tại thời điểm đó, những tiến bộ trong công nghệ bán dẫn đã làm cho bộ
The mainframes of the batch era were firmly established by the late 1960s when advances in semiconductor technology made the solid-state	

memory and integrated circuit feasible. These advances in hardware technology spawned the minicomputer era. They were small, fast, and inexpensive enough to be spread throughout the company at the divisional level. However, they were still too expensive and difficult	nhớ trạng thái rắn và mạch tích hợp trở nên khả thi. Những tiến bộ trong công nghệ phần cứng đã sinh ra kỷ nguyên máy tính mini. Chúng nhỏ, nhanh, và giá cả vừa phải nên được sử dụng rộng rãi trong công ty ở mọi phòng ban. Tuy nhiên, đối với người dùng cuối, chúng vẫn còn quá đắt và khó chuyển giao.
TABLE 1.1 Four Decades of Computing	BẢNG 1.1 Tổng quan về máy tính qua bốn thập kỷ
to use to hand over to end-users. Minicomputers made by DEC, Prime, and Data General led the way in defining a new kind of computing: time-sharing. By the 1970s it was clear that there existed two kinds of commercial or business computing:	Các máy tính mini được chế tạo bởi tập đoàn DEC, Prime, và Data General dẫn đến việc định nghĩa một loại công nghệ điện toán mới: phân hưởng thời gian. Vào những năm 1970, rõ ràng, trên thị trường đã tồn tại hai loại máy tính thương mại hoặc kinh doanh:
(1) centralized data processing mainframes, and (2) time-sharing minicomputers. In parallel with small-scale machines, supercomputers were coming into play. The first such supercomputer, the CDC 6600, was introduced in 1961 by Control Data Corporation. Cray Research Corporation introduced the best cost/performance supercomputer, the Cray-1, in 1976.	(1) Các máy tính lớn xử lý dữ liệu tập trung, và (2) các máy tính mini phân hưởng thời gian. Song song với các máy tính quy mô nhỏ, các siêu máy tính đã ra đời và tham gia vào cuộc chơi. Siêu máy tính đầu tiên là CDC 6600, được giới thiệu vào năm 1961 bởi Control Data Corporation. Cray Research Corporation đã giới thiệu siêu máy tính chi phí / hiệu suất tốt nhất, Cray-1, vào năm 1976.
1.1.3 Desktop Era	1.1.3 Kỷ nguyên desktop
Personal computers (PCs), which were introduced in 1977 by Altair, Processor	Các máy tính cá nhân (PC), được giới thiệu vào năm 1977 bởi tập đoàn Processor Technology, North Star, Tandy, Commodore, Apple, và nhiều tập

Technology, North Star, Tandy, Commodore, Apple, and many others, enhanced the productivity of end-users in numerous departments. Personal computers from Compaq, Apple, IBM, Dell, and many others soon became pervasive, and changed the face of computing.	đoàn khác, tăng năng suất làm việc của người dùng cuối trong nhiều công ty. Các máy tính cá nhân từ tập đoàn Compaq, Apple, IBM, Dell, và nhiều tập đoàn khác nhanh chóng trở nên phổ biến, và đã làm thay đổi bộ mặt của ngành máy tính.
Local area networks (LAN) of powerful personal computers and workstations began to replace mainframes and minis by 1990. The power of the most capable big machine could be had in a desktop model for one-tenth of the cost. However, these individual desktop computers were soon to be connected into larger complexes of computing by wide area networks (WAN).	Mạng máy tính cục bộ (LAN) các máy tính cá nhân và các máy trạm mạnh bắt đầu thay thế các máy tính lớn và mini vào năm 1990. Máy desktop có thể có khả năng tính toán ngang với các máy tính lớn mạnh nhất nhưng giá thành chỉ bằng một phần mười. Tuy nhiên, các destop cá nhân đã sớm được kết nối vào các phức hệ điện toán lớn hơn qua mạng diện rộng (WAN).
1.1.4 Network Era The fourth era, or network paradigm of computing, is in full swing because of rapid advances in network technology. Network technology outstripped processor technology throughout most of the 1990s. This explains the rise of the network paradigm listed in Table 1.1. The surge of network capacity tipped the balance from a processor-centric view of computing to a network-centric view.	1.1.4 Kỷ nguyên mạng máy tính Kỷ nguyên thứ tư, hay còn gọi là mô hình mạng máy tính (mô hình mạng trong kĩ thuật điện toán), đang phát triển hết sức nhanh chóng do những tiến bộ trong công nghệ mạng. Công nghệ mạng vượt xa công nghệ xử lý (công nghệ vi xử lý) trong suốt những năm 1990. Điều này là nguyên nhân của sự xuất hiện các mô hình mạng được đề cập trong Bảng 1.1. Sự tăng đột biến công suất mạng đã làm chúng ta chuyển từ quan điểm lấy bộ xử lý làm trung tâm sang quan điểm lấy mạng làm trung tâm. Trong những năm 1980 và 1990, thế giới

The 1980s and 1990s witnessed the introduction of many commercial parallel computers with multiple processors. They can generally be classified into two main categories: (1) shared memory, and (2) distributed memory systems. The number of processors in a single machine ranged from several in a shared memory computer to hundreds of thousands in a massively parallel system. Examples of parallel computers during this era include Sequent Symmetry, Intel iPSC, nCUBE, Intel Paragon, Thinking Machines (CM-2, CM-5), MsPar (MP), Fujitsu (VPP500), and others.	đã chứng kiến sự ra đời của nhiều máy tính song song thương mại có nhiều bộ xử lý. Chúng được phân thành hai loại chính: (1) hệ thống bộ nhớ dùng chung, và (2) hệ thống bộ nhớ phân tán. Số lượng các bộ vi xử lý trong một máy dao động từ một vài bộ trong máy tính bộ nhớ dùng chung cho đến hàng trăm ngàn bộ vi xử lý trong một hệ thống song song cực lớn. Ví dụ về các máy tính song song trong thời kỳ này bao gồm Sequent Symmetry, Intel iPSC, nCUBE, Intel Paragon, Thinking Machines (CM-2, CM-5), MsPar (MP), Fujitsu (VPP500), và những dòng khác.
1.1.5 Current Trends One of the clear trends in computing is the substitution of expensive and specialized parallel machines by the more cost-effective clusters of workstations. A cluster is a collection of stand-alone computers connected using some interconnection network. Additionally, the pervasiveness of the Internet created interest in network computing and more recently in grid computing. Grids are geographically distributed platforms of computation. They should provide dependable, consistent, pervasive, and inexpensive access to high-end computational facilities.	1.1.5 Các xu hướng hiện tại Một trong những xu hướng rõ rệt trong máy tính là sự thay thế các máy song song đắt tiền và chuyên biệt bằng các cụm máy trạm giá thành rẻ hơn. Một cụm là một tập hợp các máy tính độc lập được kết nối bằng mạng liên thông. Ngoài ra, sự phổ biến rộng rãi của Internet thúc đẩy sự quan tâm đến tính toán mạng (điện toán mạng) và gần đây hơn là điện toán mạng lưới. Lưới là các nền tính toán phân tán về mặt địa lý. Chúng cung cấp cho chúng ta những ta khả năng truy cập đáng tin cậy, phù hợp, phổ biến, và giá thành rẻ vào các phương tiện tính toán cao cấp. 1.2 PHÂN LOẠI KIẾN TRÚC MÁY

1.2 FLYNN'S TAXONOMY OF COMPUTER ARCHITECTURE The most popular taxonomy of computer architecture was defined by Flynn in 1966. Flynn's classification scheme is based on the notion of a stream of information. Two types of information flow into a processor: instructions and data. The instruction stream is defined as the sequence of instructions performed by the processing unit. The data stream is defined as the data traffic exchanged between the memory and the processing unit. According to Flynn's classification, either of the instruction or data streams can be single or multiple. Computer architecture can be classified into the following four distinct categories: • single-instruction single-data streams (SISD); • single-instruction multiple-data streams (SIMD); • multiple-instruction single-data streams (MISD); and • multiple-instruction multiple-data streams (MIMD). Conventional single-processor von Neumann computers are classified as SISD systems. Parallel computers are either SIMD or MIMD. When there is only one control unit and all processors execute the same instruction in a synchronized fashion, the parallel machine is classified as SIMD. In a MIMD machine, each processor has its	TÍNH CỦA FLYNN Phân loại cấu trúc máy tính phổ biến nhất được Flynn định nghĩa vào năm 1966. Phương pháp phân loại của Flynn dựa trên khái niệm về luồng thông tin. Hai loại luồng thông tin đi vào bộ xử lý là: các lệnh và dữ liệu. Luồng lệnh là chuỗi các lệnh được thực hiện bởi các đơn vị xử lý. Các luồng dữ liệu là lưu lượng dữ liệu trao đổi giữa bộ nhớ và các đơn vị xử lý. Theo phân loại Flynn, các luồng lệnh hoặc các luồng dữ liệu có thể là một hoặc nhiều (đơn hoặc đa). Kiến trúc máy tính có thể được phân loại thành bốn loại riêng biệt sau đây: • Máy tính một dòng lệnh-một dòng dữ liệu (SISD); • Máy tính một dòng lệnh-nhiều dòng dữ liệu (SIMD); • Máy tính nhiều dòng lệnh-một dòng dữ liệu (MISD), và • Máy tính nhiều dòng lệnh-nhiều dòng dữ liệu (MIMD). Máy tính đơn xử lý von Neumann truyền thống thuộc hệ SISD. Các máy tính song song hoặc có thể thuộc loại SIMD hoặc MIMD. Khi chỉ có một bộ điều khiển và tất cả các bộ xử lý thực hiện lệnh giống nhau theo kiểu đồng bộ thì máy song song được xếp vào loại SIMD. Trong máy MIMD, mỗi bộ xử lý có bộ điều

own control unit and can execute different instructions on different data. In the MISD category, the same stream of data flows through a linear array of processors executing different instruction streams. In practice, there is no viable MISD machine; however, some authors have considered pipelined machines (and perhaps systolic-array computers) as examples for MISD. Figures 1.1, 1.2, and 1.3 depict the block diagrams of SISD, SIMD, and MIMD, respectively. An extension of Flynn's taxonomy was introduced by D. J. Kuck in 1978. In his classification, Kuck extended the instruction stream further to single (scalar and array) and multiple (scalar and array) streams. The data stream in Kuck's classification is called the execution stream and is also extended to include single ……	khiển riêng và có thể thực hiện các lệnh khác nhau trên các dữ liệu khác nhau. Trong MISD, cùng một dòng dữ liệu chạy qua một mảng tuyến tính các bộ xử lý thực hiện các dòng lệnh khác nhau. Trong thực tế, máy MISD không tồn tại, tuy nhiên, một số tác giả đã xem các máy cấu trúc ống (và có thể là các máy tính mảng systolic) là các MISD. Hình 1.1, 1.2, và 1.3 mô tả sơ đồ khối tương ứng của SISD, SIMD, và MIMD. Phân loại Flynn đã được DJ Kuck bổ sung vào năm 1978. Trong phân loại của mình, Kuck mở rộng thêm luồng lệnh thành luồng lệnh đơn(vô hướng và mảng) và đa luồng lệnh (vô hướng và mảng). Luồng dữ liệu trong phân loại Kuck được gọi là luồng thực thi và cũng được mở rộng để gộp vào ……
Figure 1.1 SISD architecture. Figure 1.2 SIMD architecture. (scalar and array) and multiple (scalar and array) streams. The combination of these streams results in a total of 16 categories of architectures.	Hình 1.1 Kiến trúc trúc SISD. Hình 1.2 Kiến trúc SIMD. luồng đơn (Vô hướng và mảng) và đa luồng(vô hướng và mảng). Sự kết hợp của những luồng này dẫn đến tổng cộng 16 loại kiến trúc.
1.3 SIMD ARCHITECTURE The SIMD model of parallel computing consists of two parts: a front-end computer of the usual von Neumann style, and a processor array as shown in	1.3 KIẾN TRÚC SIMD Mô hình tính toán song song SIMD bao gồm hai phần: một máy tính phụ trợ kiểu von Neumann thông thường, và một mảng bộ xử lý như miêu tả ở hình 1.4.

Figure 1.4. The processor array is a set of identical synchronized processing elements capable of simultaneously performing the same operation on different data. Each processor in the array has a small amount of local memory where the distributed data resides while it is being processed in parallel. The processor array is connected to the memory bus of the front end so that the front end can randomly access the local …… ……	Mảng xử lý là tập hợp của các bộ xử lý đồng bộ giống hệt nhau có khả năng thực hiện đồng thời cùng một hoạt động trên các dữ liệu khác nhau. Mỗi bộ vi xử lý trong mảng có một lượng bộ nhớ riêng nhỏ để lưu dữ liệu phân tán trong khi nó đang được xử lý song song. Mảng xử lý được kết nối với bus nhớ của máy tính phụ trợ để nó có thể truy cập dữ liệu ngẫu nhiên vào bộ nhớ xử lý cục bộ (bộ nhớ riêng) ……
Figure 1.3 MIMD architecture. Virtual Processors SIMD architecture model.	Hình 1.3 Kiến trúc MIMD. Mô hình kiến trúc SIMD bộ xử lý ảo.
processor memories as if it were another memory. Thus, the front end can issue special commands that cause parts of the memory to be operated on simultaneously or cause data to move around in the memory. A program can be developed and executed on the front end using a traditional serial programming language. The application program is executed by the front end in the usual serial way, but issues commands to the processor array to carry out SIMD operations in parallel. The similarity between serial and data parallel programming is one of the strong points of data parallelism. Synchronization is made irrelevant by the lock-step syn-chronization of the processors. Processors either do nothing or exactly the same operations at the same time. In SIMD architecture,	với chức năng như một bộ nhớ khác. Như vậy, máy tính phụ trợ có thể đưa ra những lệnh đặc biệt làm cho các bộ phận của bộ nhớ được vận hành cùng lúc (đồng thời) hoặc làm cho dữ liệu di chuyển trong bộ nhớ. Một chương trình có thể được phát triển và thực thi ở máy tính phụ trợ dùng một ngôn ngữ lập trình kiểu nối tiếp truyền thống. Chương trình ứng dụng được thực thi bằng máy tính phụ trợ theo phương thức nối tiếp thông thường, nhưng truyền lệnh đến các mảng xử lý để thực hiện các phép toán SIMD song song. Sự giống nhau giữa lập trình dữ liệu song song và nối tiếp chính là một trong những điểm mạnh của xử lý dữ liệu song song. Đồng bộ hóa trở nên không thích hợp qua việc đồng bộ hoá

parallelism is exploited by applying simultaneous operations across large sets of data. This paradigm is most useful for solving problems that have lots of data that need to be updated on a wholesale basis. It is especially powerful in many regular numerical calculations.	nhịp xung của các bộ xử lý. Bộ vi xử lý không làm gì hoặc thực hiện các hoạt động giống hệt nhau cùng một lúc. Ở kiến trúc SIMD, phương pháp song song được khai thác bằng cách áp dụng đồng thời các phép toán cho các tập dữ liệu lớn. Mô hình này phát huy hiệu quả tốt nhất khi giải những bài toán có nhiều dữ liệu cần phải được cập nhật hàng loạt. Nó rất hiệu quả trong các tính toán số thông thường.
There are two main configurations that have been used in SIMD machines (see Fig. 1.5). In the first scheme, each processor has its own local memory. Processors can communicate with each other through the interconnection network. If the interconnection network does not provide direct connection between a given pair of processors, then this pair can exchange data via an intermediate processor. The ILLIAC IV used such an interconnection scheme. The interconnection network in the ILLIAC IV allowed each processor to communicate directly with four neighboring processors in an 8 x 8 matrix pattern such that the ith processor can communicate directly with the (i — 1)th, (i + 1)th, (i — 8)th, and (i + 8)th processors. In the second SIMD scheme, processors and memory modules communicate with each other via the interconnection network. Two processors can transfer data between each other via intermediate memory module(s) or possibly via intermediate	Có hai cấu hình chính được sử dụng trong các máy SIMD (xem hình. 1.5). Trong sơ đồ đầu tiên, mỗi bộ xử lý có bộ nhớ cục bộ riêng của nó. Các bộ vi xử lý có thể giao tiếp với nhau thông qua mạng liên thông. Nếu mạng liên thông không kết nối trực tiếp giữa hai bộ xử lý xác định, thì cặp này có thể trao đổi dữ liệu thông qua một bộ xử lý trung gian. ILLIAC IV đã sử dụng một sơ đồ kết nối này. Các kết nối mạng trong ILLIAC IV cho phép mỗi bộ xử lý giao tiếp trực tiếp với bốn bộ vi xử lý lân cận theo mô hình ma trận 8 x 8 sao cho bộ vi xử lý thứ i có thể giao tiếp trực tiếp với bộ (i - 1), (i + 1), (i - 8), và bộ vi xử lý thứ (i + 8). Ở sơ đồ SIMD thứ hai, bộ vi xử lý và các mô-đun bộ nhớ giao tiếp với nhau thông qua mạng liên thông. Hai bộ vi xử lý có thể truyền dữ liệu cho nhau thông qua một hay nhiều mô-đun bộ nhớ trung gian hoặc qua một hoặc nhiều bộ xử lý trung gian. BSP (Bộ xử lý khoa học

processor(s). The BSP (Burroughs' Scientific Processor) used the second SIMD scheme. 1.4 MIMD ARCHITECTURE Multiple-instruction multiple-data streams (MIMD) parallel architectures are made of multiple processors and multiple memory modules connected together via some Figure 1.5 Two SIMD schemes. interconnection network. They fall into two broad categories: shared memory or message passing. Figure 1.6 illustrates the general architecture of these two categories. Processors exchange information through their central shared memory in shared memory systems, and exchange information through their interconnection network in message passing systems. A shared memory system typically accomplishes interprocessor coordination through a global memory shared by all processors. These are typically server systems that communicate through a bus and cache memory controller. The bus/ cache architecture alleviates the need for expensive multiported memories and interface circuitry as well as the need to adopt a message-passing paradigm when developing application software. Because access to shared memory is balanced, these systems are also called SMP (symmetric multiprocessor) systems. Each processor has equal opportunity to read/write to memory, including equal access speed.	Burroughs) sử dụng sơ đồ SIMD thứ hai. 1.4 CẤU TRÚC MIMD Kiến trúc song song nhiều dòng lệnh-nhiều dòng dữ liệu (MIMD) được tạo thành từ nhiều bộ xử lý và nhiều mô-đun bộ nhớ kết nối với nhau thông qua một số kết nối mạng. Chúng thuộc hai loại chính: bộ nhớ dùng chung hoặc truyền tin. Hình 1.6 minh họa cấu trúc chung của hai loại này. Các bộ vi xử lý trao đổi thông tin thông qua bộ nhớ dùng chung trung tâm của chúng trong các hệ thống bộ nhớ dùng chung, và trao đổi thông tin thông qua kết nối mạng của chúng trong hệ thống truyền tin. Một hệ thống bộ nhớ dùng chung thường phối hợp các bộ vi xử lý với nhau thông qua bộ nhớ toàn cục được tất cả các bộ xử lý dùng chung (chia sẻ). Đây là những hệ thống máy chủ điển hình giao tiếp thông qua bus và bộ điều khiển bộ nhớ đệm. Kiến trúc bus / bộ nhớ đệm làm giảm nhu cầu sử dụng các bộ nhớ nhiều cổng và mạch giao tiếp đắt tiền cũng như nhu cầu áp dụng một mô hình truyền tin khi phát triển phần mềm ứng dụng. Do việc truy cập vào bộ nhớ dùng chung được cân bằng, các hệ thống này còn được gọi là các hệ SMP (đa xử lý đối xứng). Mỗi bộ xử lý có cơ hội đọc/viết như nhau vào bộ nhớ, thậm chí cả tốc độ truy cập cũng bằng nhau.

Shared Memory MIMD Architecture Message Passing MIMD Architecture Figure 1.6 Shared memory versus message passing architecture. Commercial examples of SMPs are Sequent Computer's Balance and Symmetry, Sun Microsystems multiprocessor servers, and Silicon Graphics Inc. multiprocessor servers.	Kiến trúc MIMD bộ nhớ dùng chung Kiến trúc MIMD truyền tin Hình 1.6 Kiến trúc bộ nhớ dùng chung và truyền tin Các ví dụ thương mại về SMP là Sequent Computer's Balance và Symmetry, các máy chủ đa xử lý Sun Microsystems, và máy chủ đa xử lý Silicon Graphics Inc.
A message passing system (also referred to as distributed memory) typically combines the local memory and processor at each node of the interconnection network. There is no global memory, so it is necessary to move data from one local memory to another by means of message passing. This is typically done by a Send/Receive pair of commands, which must be written into the application software by a programmer. Thus, programmers must learn the message-passing paradigm, which involves data copying and dealing with consistency issues. Commercial examples of message passing architectures c. 1990 were the nCUBE, iPSC/2, and various Transputer-based systems. These systems eventually gave way to Internet connected systems whereby the processor/memory nodes were either Internet servers or clients on individuals' desktop. It was also apparent that distributed memory is the only way efficiently to increase the number of processors	Hệ thống truyền tin (còn được gọi là bộ nhớ phân tán) thường kết hợp với bộ nhớ riêng và bộ vi xử lý tại mỗi nút mạng. Vì không có bộ nhớ toàn cục nên bắt buộc phải chuyển dữ liệu từ bộ nhớ riêng này sang bộ nhớ khác bằng cách truyền tin. Điều này thường được thực hiện bằng cặp lệnh gửi / nhận, chúng phải được một lập trình viên viết vào các phần mềm ứng dụng. Do đó, các lập trình viên phải tìm hiểu các mô hình truyền tin, bao gồm sao chép dữ liệu và xử lý các vấn đề nhất quán. Một số ví dụ thương mại của kiến trúc truyền tin năm 1990 là Ncube, IPSC / 2, và các hệ thống dựa trên phần mềm trung gian khác nhau. Cuối cùng, các hệ thống này cũng nhường chỗ cho các hệ thống kết nối Internet trong đó các nút vi xử lý / bộ nhớ hoặc là máy chủ Internet hoặc là các client trên desktop cá nhân. Rõ ràng, bộ nhớ phân tán là phương thức duy nhất có hiệu quả để tăng số lượng các bộ vi xử lý của hệ thống song song và phân tán. Nếu khả năng mở rộng hệ

managed by a parallel and distributed system. If scalability to larger and larger systems (as measured by the number of processors) was to continue, systems had to use distributed memory techniques. These two forces created a conflict: programming in the shared memory model was easier, and designing systems in the message passing model provided scalability. The distributed-shared memory (DSM) architecture began to appear in systems like the SGI Origin2000, and others. In such systems, memory is physically distributed; for example, the hardware architecture follows the message passing school of design, but the programming model follows the shared memory school of thought. In effect, software covers up the hardware. As far as a programmer is concerned, the architecture looks and behaves like a shared memory machine, but a message passing architecture lives underneath the software. Thus, the DSM machine is a hybrid that takes advantage of both design schools.	thống ngày càng lớn (được đo bằng số lượng các bộ vi xử lý) vẫn tiếp tục, các hệ thống phải sử dụng các kỹ thuật bộ nhớ phân tán. Hai ràng buộc này tạo ra mâu thuẫn: lập trình theo mô hình bộ nhớ dùng chung dễ dàng hơn, và thiết kế các hệ thống theo mô hình truyền tin có khả năng mở rộng. Kiến trúc bộ nhớ phân tán-dùng chung (DSM) bắt đầu xuất hiện trong các hệ thống như SGI Origin2000, và những hệ thống khác. Với những hệ thống như vậy, bộ nhớ phân tán về mặt vật lý, ví dụ, kiến trúc phần cứng theo trường phái thiết kế truyền tin, nhưng mô hình lập trình lại theo trường phái bộ nhớ dùng chung. Trong thực tế, phần mềm chi phối các phần cứng. Theo những gì một lập trình viên biết, kiến trúc nhìn bề ngoài có vẻ như một máy bộ nhớ dùng chung, nhưng kiến trúc truyền tin lại hoạt động bên dưới phần mềm. Như vậy, máy DSM là một dạng lai hóa tận dụng cả hai trường phái thiết kế.
1.4.1 Shared Memory Organization A shared memory model is one in which processors communicate by reading and writing locations in a shared memory that is equally accessible by all processors. Each processor may have registers, buffers,	1.4.1 Tổ chức bộ nhớ dùng chung Mô hình bộ nhớ dùng chung là một mô hình mà bộ vi xử lý giao tiếp bằng cách đọc và ghi lại vị trí trong bộ nhớ dùng chung, cái mà tất cả các bộ vi xử lý đều có thể truy cập vào nó với khả năng như nhau. Mỗi bộ xử lý đều có thanh ghi, bộ đệm, bộ nhớ đệm, và các ngân hàng bộ nhớ riêng được xem như nguồn nhớ bổ

caches, and local memory banks as additional memory resources. A number of basic issues in the design of shared memory systems have to be taken into consideration. These include access control, synchronization, protection, and security. Access control determines which process accesses are possible to which resources. Access control models make the required check for every access request issued by the processors to the shared memory, against the contents of the access control table. The latter contains flags that determine the legality of each access attempt. If there are access attempts to resources, then until the desired access is completed, all disallowed access attempts and illegal processes are blocked. Requests from sharing processes may change the contents of the access control table during execution. The flags of the access control with the synchronization rules determine the system's functionality. Synchronization constraints limit the time of accesses from sharing processes to shared resources. Appropriate synchronization ensures that the information flows properly and ensures system functionality. Protection is a system feature that prevents pro-cesses from making arbitrary access to resources belonging to other processes. Sharing and protection are incompatible; sharing allows access, whereas protection restricts it.	sung. Một số vấn đề cơ bản trong việc thiết kế hệ thống bộ nhớ dùng chung phải được xem xét. Bao gồm: kiểm soát truy cập, đồng bộ hóa, bảo vệ và bảo mật. Kiểm soát truy cập xác định quá trình truy cập nào có thể dùng cho nguồn tài nguyên nào. Mô hình điều khiển truy cập thực hiện việc kiểm tra bắt buộc đối với mỗi yêu cầu truy cập của bộ vi xử lý đến bộ nhớ dùng chung, dựa vào nội dung của bảng điều khiển truy cập. Bản này chứa cờ xác định tính hợp lệ của mỗi nổ lực truy cập (lần thử truy cập). Nếu có những nổ lực truy cập vào các nguồn tài nguyên, sau quá trình xem xét các truy cập, những truy cập nào không được phép và các quá trình không hợp lệ bị chặn. Các yêu cầu của quá trình dùng chung có thể thay đổi nội dung của bảng điều khiển truy cập trong quá trình thực thi. Cờ điều khiển truy cập với những quy tắc đồng bộ hóa xác định chức năng của hệ thống. Có những ràng buộc đồng bộ hóa hạn chế thời gian truy cập của quá trình chia sẻ nguồn tài nguyên dùng chung. Đồng bộ hóa thích hợp đảm bảo lượng thông tin lưu thông đúng và đảm bảo chức năng hệ thống. Bảo vệ là một tính năng hệ thống ngăn chặn các quá trình truy cập tùy ý vào các nguồn tài nguyên của các quá trình khác. Quá trình dùng chung (chia sẻ) và bảo vệ không tương thích với nhau, dùng chung (chia sẻ) cho phép truy cập, còn bảo vệ lại hạn

The simplest shared memory system consists of one memory module that can be accessed from two processors. Requests arrive at the memory module through its two ports. An arbitration unit within the memory module passes requests through to a memory controller. If the memory module is not busy and a single request arrives, then the arbitration unit passes that request to the memory controller and the request is granted. The module is placed in the busy state while a request is being serviced. If a new request arrives while the memory is busy servicing a previous request, the requesting processor may hold its request on the line until the memory becomes free or it may repeat its request sometime later. Depending on the interconnection network, a shared memory system leads to systems can be classified as: uniform memory access (UMA), nonuniform memory access (NUMA), and cache-only memory architecture (COMA). In the UMA system, a shared memory is accessible by all processors through an interconnection network in the same way a single processor accesses its memory. Therefore, all processors have equal access time to any memory location. The interconnection network used in the UMA can be a single bus, multiple buses, a crossbar, or a multiport	chế nó. Hệ thống bộ nhớ dùng chung đơn giản nhất gồm một mô-đun bộ nhớ có thể được truy cập từ hai bộ vi xử lý. Mô-đun bộ nhớ tiếp nhận những yêu cầu thông qua hai cổng của nó. Bộ xử lý lệnh trong mô-đun bộ nhớ chuyển các yêu cầu thông qua một bộ điều khiển. Nếu mô-đun bộ nhớ không bận trong quá trình xử lý mà có một yêu cầu đến, thì bộ xử lý lệnh chuyển yêu cầu đó đến bộ điều khiển và yêu cầu được chấp nhận. Mô-đun được đặt trong trạng thái bận trong khi có một yêu cầu đang được xử lý. Nếu một yêu cầu mới đến trong khi bộ nhớ đang bận xử lý một yêu cầu trước đó, bộ xử lý yêu cầu có thể giữ yêu cầu đó trên hàng cho đến khi bộ nhớ rãnh hoặc nó có thể lặp lại các yêu cầu vài lần sau đó. Tùy thuộc vào mạng liên thông, một hệ thống bộ nhớ dùng chung dẫn đến các hệ có thể chia thành: truy cập bộ nhớ đồng nhất (UMA), truy cập bộ nhớ không đồng nhất (Numa), và kiến trúc bộ nhớ chỉ dùng Cache (COMA). Trong hệ thống UMA, tất cả các bộ vi xử lý có thể truy cập vào bộ nhớ dùng chung thông qua mạng liên thông giống như (theo cách giống như) một bộ xử lý truy cập vào bộ nhớ của nó. Vì vậy, tất cả các bộ vi xử lý có thời gian truy cập như nhau tại bất kỳ vị trí nhớ. Mạng liên thông được sử dụng trong UMA có thể là một

memory. In the NUMA system, each processor has part of the shared memory attached. The memory has a single address space. Therefore, any processor could access any memory location directly using its real address. However, the access time to modules depends on the distance to the processor. This results in a nonuniform memory access time. A number of architectures are used to interconnect processors to memory modules in a NUMA. Similar to the NUMA, each processor has part of the shared memory in the COMA. However, in this case the shared memory consists of cache memory. A COMA system requires that data be migrated to the processor requesting it. Shared memory systems will be discussed in more detail in Chapter 4.	bus, nhiều bus, bộ chuyển mạch điểm chéo, hay một bộ nhớ đa cổng. Trong hệ thống Numa, mỗi bộ vi xử lý có kèm theo một phần của bộ nhớ dùng chung. Bộ nhớ này chỉ có một không gian địa chỉ. Vì vậy, bất kỳ bộ vi xử lý nào cũng có thể truy cập trực tiếp vào bất kỳ vị trí nhớ nào khi sử dụng địa chỉ thực của nó. Tuy nhiên, thời gian truy cập vào các module phụ thuộc vào khoảng cách đến bộ xử lý. Điều này làm cho thời gian truy cập vào bộ nhớ không đồng đều (khống bằng nhau). Một số kiến trúc được sử dụng để liên kết các bộ vi xử lý với mô-đun trong bộ nhớ Numa. Tương tự như Numa, trong bộ nhớ COMA, mỗi bộ vi xử lý có một phần của bộ nhớ dùng chung. Tuy nhiên, trong trường hợp này, bộ nhớ dùng chung có bộ nhớ Cache. Một hệ thống bộ nhớ COMA yêu cầu dữ liệu được di chuyển đến bộ xử lý đang yêu cầu nó. Trong chương 4, chúng sẽ sẽ thảo luận chi tiết hơn về bộ nhớ dùng chung.
1.4.2 Message Passing Organization Message passing systems are a class of multiprocessors in which each processor has access to its own local memory. Unlike shared memory systems, communications in message passing systems are performed via send and receive operations. A node in such	1.4.2 Bộ phận truyền tin Các hệ thống truyền tin là một loại đa xử lý trong đó mỗi bộ xử lý có thể truy cập vào bộ nhớ riêng của nó. Không giống như các hệ thống bộ nhớ dùng chung, truyền thông trong các hệ thống truyền tin được thực hiện thông qua các hoạt động gửi và nhận. Một nút trong một hệ thống như vậy bao gồm một bộ

a system consists of a processor and its local memory. Nodes are typically able to store messages in buffers (temporary memory locations where messages wait until they can be sent or received), and perform send/receive operations at the same time as processing. Simultaneous message processing and problem calculating are handled by the underlying operating system. Processors do not share a global memory and each processor has access to its own address space. The processing units of a message passing system may be connected in a variety of ways ranging from architecture-specific interconnection structures to geographically dispersed networks. The message passing approach is, in principle, scalable to large proportions. By scalable, it is meant that the number of processors can be increased without significant decrease in efficiency of operation.	xử lý và bộ nhớ riêng của nó. Các nút có thể lưu trữ tin trong những vùng đệm (các vị trí nhớ tạm thời, nơi các thông tin chờ cho đến khi chúng có thể gửi hoặc nhận), và thực hiện những hoạt động gửi / nhận đồng thời với việc xử lý. Việc xử lý tin và việc tính toán đồng thời được xử lý bởi hệ điều hành cơ bản. Các bộ xử lý của hệ thống truyền tin không sử dụng chung một bộ nhớ toàn cục và mỗi bộ xử lý có quyền truy cập vào vùng địa chỉ riêng của mình. Các đơn vị xử lý của một hệ thống truyền tin có thể được kết nối theo nhiều cách khác nhau, từ cấu trúc nối kết đặc trưng tới các mạng phân tán về mặt địa lý. Về nguyên tắc, phương pháp truyền tin có khả năng mở rộng sang quy mô lớn. Khả năng có thể mở rộng dẫn đến một lợi thế là chúng ta có thể tăng số lượng bộ xử lý mà không làm giảm đáng kể hiệu suất tính toán.
Message passing multiprocessors employ a variety of static networks in local communication. Of importance are hypercube networks, which have received special attention for many years. The nearest neighbor two-dimensional and three-dimensional mesh networks have been used in message passing systems as well. Two important design factors must be considered in designing	Các bộ đa xử lý truyền tin sử dụng một loạt mạng tĩnh trong truyền thông cục bộ. Quan trọng trong số đó là mạng hình siêu khối, một loại mạng đã thu hút sự quan tâm trong thời gian dài. Mạng mắt lưới 2 và 3 chiều lân cận gần nhất cũng được sử dụng trong hệ thống truyền tin. Cần phải xem xét hai yếu tố quan trọng trong việc thiết kế các mạng liên thông cho hệ thống truyền tin. Đó là băng thông liên kết và thời gian trì hoãn của mạng. Băng thông liên kết được định nghĩa là số bit có thể đượctruyền đi trong một đơn vị

interconnection networks for message passing systems. These are the link bandwidth and the network latency. The link bandwidth is defined as the number of bits that can be transmitted per unit time (bits/ s). The network latency is defined as the time to complete a message transfer. Wormhole routing in message passing was introduced in 1987 as an alternative to the traditional store-and-forward routing in order to reduce the size of the required buffers and to decrease the message latency. In wormhole routing, a packet is divided into smaller units that are called flits (flow control bits) such that flits move in a pipeline fashion with the header flit of the packet leading the way to the destination node. When the header flit is blocked due to network congestion, the remaining flits are blocked as well. More details on message passing will be introduced in Chapter 5.

thời gian (bit/ s). Thời gian trì hoãn của mạng được định nghĩa là thời gian để hoàn thành một quá trình truyền tin. Cơ chế điều khiển luồng Wormhole trong truyền tin đã được đưa ra vào năm 1987 như một sự thay thế cho cơ chế điều khiển luồng lưu trữ-và-chuyển tiếp truyền thống để làm giảm kích thước của bộ đệm cần thiết và giảm độ trễ truyền tin. Trong cơ chế điều khiển luồng Wormhole, một gói tin được chia thành các đơn vị nhỏ hơn được gọi là các flit (bit điều khiển lưu lượng) để các flit di chuyển theo kiểu đường ống cùng với flit đầu của gói tin dẫn đến nút đích. Khi flit đầu bị chặn do tắc nghẽn mạng, các flit còn lại cũng bị chặn. Trong chương 5, chúng ta sẽ nghiên cứu chi tiết hơn về quá trình truyền tin.

1.5 INTERCONNECTION NETWORKS

Multiprocessors interconnection networks (INs) can be classified based on a number of criteria. These include (1) mode of operation (synchronous versus asynchronous), (2) control strategy (centralized versus decentralized), (3) switching techniques (circuit versus packet), and (4) topology (static versus dynamic).

1.5.1 Mode of Operation

1.5 CÁC MẠNG LIÊN THÔNG

Các mạng liên thông đa xử lý (INS) có thể được phân loại dựa trên một số tiêu chí. Chúng bao gồm (1) phương thức hoạt động (đồng bộ hay bất đồng bộ), (2) chiến lược kiểm soát (tập trung hay không tập trung), (3) các kỹ thuật chuyển mạch (mạch hay gói tin), và (4) tô pô (tĩnh hay động).

1.5.1 PHƯƠNG THỨC HOẠT ĐỘNG

Theo phương thức hoạt động, INs được phân loại thành đồng bộ và bất đồng bộ. Trong phương thức hoạt động đồng bộ, tất cả các thành phần trong hệ thống sử

According to the mode of operation, INs are classified as synchronous versus asynchronous. In synchronous mode of operation, a single global clock is used by all components in the system such that the whole system is operating in a lock-step manner. Asynchronous mode of operation, on the other hand, does not require a global clock. Handshaking signals are used instead in order to coordinate the operation of asynchronous systems. While synchronous systems tend to be slower compared to asynchronous systems, they are race and hazard-free.	dụng chung một xung đồng hồ để toàn bộ hệ hoạt động theo kiểu lock-step (xung nhịp). Mặt khác, phương thức hoạt động không đồng bộ không đòi hỏi một xung đồng hồ chung. Thay vào đó, tín hiệu bắt tay được sử dụng để phối hợp hoạt động của các hệ thống không đồng bộ. Trong khi hệ thống đồng bộ có xu hướng chậm hơn so với các hệ thống không đồng bộ, chúng không cạnh tranh và ảnh hưởng nhau.
1.5.2 Control Strategy According to the control strategy, INs can be classified as centralized versus decentralized. In centralized control systems, a single central control unit is used to oversee and control the operation of the components of the system. In decentralized control, the control function is distributed among different components in the system. The function and reliability of the central control unit can become the bottleneck in a centralized control system. While the crossbar is a centralized system, the multistage interconnection networks are decentralized.	1.5.2 CHIẾN LƯỢC KIỂM SOÁT Theo chiến lược kiểm soát, INs có thể được phân loại thành tập trung và phi tập trung. Trong các hệ thống điều khiển tập trung, một đơn vị điều khiển trung tâm duy nhất được sử dụng để giám sát và kiểm soát các thành phần của hệ thống. Trong điều khiển phi tập trung, chức năng điều khiển được phân bổ cho các thành phần khác nhau trong hệ. Chức năng và độ tin cậy của các đơn vị điều khiển trung tâm có thể trở thành một trở ngại lớn trong hệ thống điều khiển tập trung. Trong khi mạng phân bố là một hệ thống tập trung, thì mạng liên thông nhiều tầng là phi tập trung. PHẦN BÊN DƯỚI KHOẢNG 5 TRANG KÉP (2.5 TRANG ĐƠN) do bạn bên CNTT dịch
1.5.3 Switching Techniques Interconnection networks can be classified according to the switching	1.5.3 Các kỹ thuật chuyển mạch Theo cơ chế chuyển mạch, mạng liên

mechanism as circuit versus packet switching networks. In the circuit switching mechanism, a complete path has to be established prior to the start of communication between a source and a destination. The established path will remain in existence during the whole communication period. In a packet switching mechanism, communication between a source and destination takes place via messages that are divided into smaller entities, called packets. On their way to the destination, packets can be sent from a node to another in a store-and-forward manner until they reach their destination. While packet switching tends to use the network resources more efficiently compared to circuit switching, it suffers from variable packet delays.	thông có thể được phân loại thành chuyển mạch-mạch và chuyển mạch-gói. Trong cơ chế chuyển mạch-mạch, một đường dẫn hoàn chỉnh phải được thiết lập trước khi bắt đầu giao tiếp giữa nguồn và đích. Đường dẫn đã thiết lập sẽ vẫn tồn tại trong suốt khoảng thời gian truyền thông. Trong cơ chế chuyển mạch gói, truyền thông giữa nguồn và đích thực hiện thông qua tin nhắn được chia thành các đơn vị nhỏ hơn, gọi là các gói tin. Trên đường đến đích, các gói tin có thể được gửi từ một nút tới nút khác bằng cách lưu trữ và chuyển tiếp cho đến khi tới được điểm đến của chúng. Ưu điểm của chuyển mạch gói là sử dụng các tài nguyên mạng hiệu quả hơn so với chuyển mạch mạch, nhược điểm của nó là độ trễ các gói tin biến đổi.
1.5.4 Topology An interconnection network topology is a mapping function from the set of processors and memories onto the same set of processors and memories. In other words, the topology describes how to connect processors and memories to other processors and memories. A fully connected topology, for example, is a mapping in which each processor is connected to all other processors in the computer. A ring topology is a mapping that connects processor k to its neighbors, processors (k - 1) and (k + 1).	1.5.4 Tô pô Tô pô mạng liên thông là một hàm ánh xạ từ các bộ vi xử lý và bộ nhớ vào cùng một bộ vi xử lý và bộ nhớ. Nói cách khác, các tô pô mô tả cách thức kết nối bộ vi xử lý và các bộ nhớ với bộ vi xử lý và các bộ nhớ khác. Ví dụ, một tô pô kết nối hoàn chỉnh là một quá trình ánh xạ, trong đó mỗi bộ vi xử lý được kết nối với tất cả các bộ xử lý khác trong máy tính. Tô pô vòng là một ánh xạ kết nối vi xử lý k với các vi xử lý lân cận của nó, các bộ vi xử lý (k - 1) và (k + 1).
In general, interconnection networks	Nhìn chung, mạng liên thông có thể được

can be classified as static versus dynamic networks. In static networks, direct fixed links are established among nodes to form a fixed network, while in dynamic networks, connections are established as needed. Switching elements are used to establish connections among inputs and outputs. Depending on the switch settings, different interconnections can be established. Nearly all multiprocessor systems can be distinguished by their interconnection network topology. Therefore, we devote Chapter 2 of this book to study a variety of topologies and how they are used in constructing a multiprocessor system. However, in this section, we give a brief introduction to interconnection networks for shared memory and message passing systems.	phân loại thành các mạng tĩnh và động. Trong mạng tĩnh, liên kết cố định trực tiếp được thiết lập giữa các nút để tạo thành một mạng cố định, trong khi trong mạng động, kết nối được thiết lập khi cần thiết. Các phần tử chuyển mạch được sử dụng để thiết lập kết nối giữa đầu vào và đầu ra. Tùy thuộc vào các thiết lập chuyển mạch, các liên kết khác nhau có thể được thiết lập. Gần như tất cả các hệ thống đa xử lý có thể được phân biệt theo tô pô mạng liên thông của chúng. Vì vậy, chúng tôi dành trọn Chương 2 của cuốn sách này để nghiên cứu một loạt các tô pô và cách sử dụng chúng trong việc xây dựng một hệ thống đa xử lý. Tuy nhiên, trong phần này, chúng tôi sẽ đưa ra một giới thiệu ngắn gọn về các mạng liên thông của hệ thống bộ nhớ dùng chung và hệ thống truyền tin.
Shared memory systems can be designed using bus-based or switch-based INs. The simplest IN for shared memory systems is the bus. However, the bus may get saturated if multiple processors are trying to access the shared memory (via the bus) simultaneously. A typical bus-based design uses caches to solve the bus contention problem. Other shared memory designs rely on switches for interconnection. For example, a crossbar switch can be used to connect multiple processors to multiple memory modules. A crossbar switch, which will be discussed further in	Việc thiết kế hệ thống bộ nhớ dùng chung có thể sử dụng các IN dựa trên bus hoặc dựa trên chuyển mạch. IN đơn giản nhất đối với các bộ nhớ dùng chung là bus. Tuy nhiên, bus có thể bị bão hòa nếu có quá nhiều bộ xử lý truy cập đồng thời vào bộ dùng chung (thông qua bus). Thông thường, thiết kế dựa trên bus sử dụng bộ nhớ đệm để giải quyết vấn đề tranh chấp bus. Các thiết kế bộ nhớ dùng chung khác lệ thuộc vào các chuyển mạch để kết nối. Ví dụ, một chuyển mạch crossbar có thể được sử dụng để kết nối nhiều bộ xử lý cho nhiều mô-đun

Chapter 2, can be visualized as a mesh of wires with switches at the points of intersection. Figure 1.7 shows (a) bus-based and (b) switch-based shared memory systems. Figure 1.8 shows bus-based systems when a single bus is used versus the case when multiple buses are used.	bộ nhớ. Chuyển mạch crossbar, sẽ được thảo luận trong Chương 2, có thể được hình dung như một mạng lưới dây cùng với các chuyển mạch tại các điểm giao nhau. Hình 1.7 biểu diễn các hệ thống bộ nhớ dùng chung (a) dựa trên bus và (b) dựa trên chuyển mạch. Hình 1.8 minh họa hệ thống dựa trên bus, bao gồm đơn bus và đa bus.
Message passing INs can be divided into static and dynamic. Static networks form all connections when the system is designed rather than when the connection is needed. In a static network, messages must be routed along established links. Figure 1.8 Single bus and multiple bus systems. Dynamic INs establish a connection between two or more nodes on the fly as messages are routed along the links. The number of hops in a path from source to destination node is equal to the number of point-to-point links a message must traverse to reach its destination. In either static or dynamic networks, a single message may have to hop through intermediate processors on its way to its destination. Therefore, the ultimate performance of an interconnection network is greatly influenced by the number of hops taken to traverse the network. Figure 1.9 shows a number of popular static topologies: (a) linear array, (b) ring, (c) mesh, (d) tree, (e) hypercube.	Các mạng liên thông truyền tin có thể được chia thành tĩnh và động. Mạng tĩnh hình thành tất cả các kết nối khi hệ thống được thiết kế chứ không phải khi cần kết nối. Trong mạng tĩnh, thông điệp phải được chuyển cùng với các liên kết được thiết lập. Hình 1.8 Hệ thống bus đơn và nhiều bus. Các mạng liên thông động thiết lập kết nối giữa hai hay nhiều nút tùy biến khi tin nhắn được chuyển dọc theo các liên kết. Số lượng hops trong một đường dẫn từ nút nguồn đến nút đích bằng với số liên kết điểm đến điểm mà một tin phải đi qua để đến đích của nó. Trong cả mạng tĩnh hoặc động, một tin đơn có thể phải nhảy qua các bộ vi xử lý trung gian trên đường đến đích của nó. Vì vậy, hoạt động cuối cùng của mạng liên thông bị ảnh hưởng rất nhiều bởi số lượng bước nhảy thực hiện để đi qua mạng. Hình 1.9 biểu diễn một số tô pô tĩnh phổ biến: (a) mảng tuyến tính, (b) vòng, (c) lưới, (d)

	cây, (e) siêu lập phương.
Figure 1.10 shows examples of dynamic networks. The single-stage interconnection network of Figure 1.10a is a simple dynamic network that connects each of the inputs on the left side to some, but not all, outputs on the right side through a single layer of binary switches represented by the rectangles. The binary switches can direct the message on the left-side input to one of two possible outputs on the right side. If we cascade enough single-stage networks together, they form a completely connected multistage interconnection network (MIN), as shown in Figure 1.10b. The omega MIN connects eight sources to eight destinations. The con-nection from the source 010 to the destination 010 is shown as a bold path in Figure 1.10b. These are dynamic INs because the connection is made on the fly, as needed. In order to connect a source to a destination, we simply use a function of the bits of the source and destination addresses as instructions for dynamically selecting a path through the switches. For example, to connect source 111 to destination 001 in the omega network, the switches in the first and second stage must be set to connect to the upper output port, while the switch at the third stage must be set TABLE 1.2 Performance Comparison of Some Dynamic INs MIN, on the other hand requires log N clocks to make a connection. The diameter of the omega MIN is therefore	Hình 1.10 đưa ra ví dụ về mạng động. Mạng liên thông một tầng ở hình 1.10a là một mạng động đơn giản, kết nối mỗi đầu vào ở phía bên trái với những phần khác, nhưng không phải tất cả, các đầu ra ở phía bên phải qua một lớp thiết bị chuyển mạch nhị phân được biểu diễn bằng các hình chữ nhật. Các chuyển mạch nhị phân có thể hướng tin ở đầu vào bên trái đến một trong hai đầu ra khả dĩ ở bên phải. Nếu chúng ta ghép các mạng một tầng với nhau, chúng sẽ hình thành nên một mạng liên thông nhiều tầng được kết nối hoàn chỉnh (MIN), như biểu diễn trong hình 1.10b. MIN omega kết nối tám nguồn đến tám đích. Sự kết nối từ nguồn 010 đến đích 010 được biểu diễn bằng một đường đậm trong hình 1.10b. Đây là các IN động vì kết nối được thực hiện tùy biến khi cần thiết. Để kết nối một nguồn đến đích, chúng ta chỉ cần sử dụng một hàm theo bit địa chỉ nguồn và đích như hướng dẫn để lựa chọn động một đường đi qua các chuyển mạch. Ví dụ, để kết nối nguồn 111 đến đích 001 trong mạng omega, các chuyển mạch ở tầng đầu tiên và thứ hai buộc phải kết nối với cổng ra phía trên, trong khi chuyển mạch ở tầng thứ ba phải được đặt BẢNG 1.2 So sánh tính năng của một số IN động

log N. Both networks limit the number of alternate paths between any source/destination pair. This leads to limited fault tolerance and network traffic congestion. If the single path between pairs becomes faulty, that pair cannot communicate. If two pairs attempt to communicate at the same time along a shared path, one pair must wait for the other. This is called blocking, and such MINs are called blocking networks. A network that can handle all possible connections without blocking is called a nonblocking network.	MIN, mặt khác cần logN đồng hồ để thực hiện kết nối. Do đó, đường kính của MIN omega là N. Cả hai mạng đều hạn chế số lượng đường luân phiên giữa bất kỳ nguồn / đích nào. Điều này dẫn đến hạn chế khả năng chịu lỗi và tắc nghẽn lưu thông mạng. Nếu con đường duy nhất giữa các cặp bị lỗi, cặp đó không thể giao tiếp. Nếu hai cặp cố gắng giao tiếp đồng thời trên cùng một đường, một cặp phải đợi cặp kia. Điều này được gọi là chặn, và các MIN như vậy được gọi là mạng chặn. Một mạng có thể xử lý tất cả các kết nối có thể có (các kết nối khả dĩ) mà không ngăn chặn được gọi là mạng không chặn.
Table 1.2 shows a performance comparison among a number of different dynamic INs. In this table, m represents the number of multiple buses used, while N represents the number of processors (memory modules) or input/output of the network.	Bảng 1.2 so sánh hiệu năng của các IN động khác nhau. Trong bảng này, m biểu diễn cho số bus sử dụng, trong khi N biểu diễn số bộ vi xử lý (các mô đun bộ nhớ) hoặc đầu vào / đầu ra của mạng.
Table 1.3 shows a performance comparison among a number of static INs. In this table, the degree of a network is defined as the maximum number of links (channels) connected to any node in the network. The diameter of a network is defined as the maximum path, p, of the shortest paths between any two nodes. Degree of a node, d, is defined as the number of channels incident on the node. Performance measures will be discussed in more detail in Chapter 3.	Bảng 1.3 so sánh hiệu năng của một số IN tĩnh. Trong bảng này, mức độ của một mạng (bậc của mạng) được định nghĩa là số lượng tối đa các liên kết (kênh) kết nối với bất kỳ nút nào trong mạng. Đường kính của mạng được định nghĩa là đường dẫn cực đại, p, trong các đường dẫn ngắn nhất giữa bất kỳ hai nút nào. Mức độ của nút (bậc của nút), d, được định nghĩa là số lượng kênh đến trên nút. Trong chương 3, chúng ta sẽ đề cập đến việc

	định lượng hiệu năng.

CHAPTER 2

Multiprocessors Interconnection Networks

As we have seen in Chapter 1, a multiprocessor system consists of multiple processing units connected via some interconnection network plus the software needed to make the processing units work together. There are two major factors used to categorize such systems: the processing units themselves, and the interconnection network that ties them together. A number of communication styles exist for multiprocessing networks. These can be broadly classified according to the communication model as shared memory (single address space) versus message passing (multiple address spaces). Communication in shared memory systems is performed by writing to and reading from the global memory, while communication in message passing systems is accomplished via send and receive commands. In both cases, the interconnection network plays a major role in determining the communication speed. In this chapter, we introduce the different topologies used for interconnecting multiple processors and memory modules. Two schemes are introduced, namely static and dynamic interconnection networks. Static networks form all connections when the system is designed rather than when the connection is needed. In a static network, messages must be routed along established links. Dynamic interconnection networks establish connections between two or more nodes on the fly as messages are routed along the links. The hypercube, mesh, and k-ary n-cube topologies are introduced as examples for static networks. The bus, crossbar, and

[redacted]

chứ không phải lúc cần kết nối

định tuyến, phân luồng) dọc theo

khi

dọc theo

multistage inter-connection topologies are introduced as examples for dynamic interconnection networks. Our coverage in this chapter will conclude with a section on performance evaluation and analysis of the different interconnection networks.

2.1 INTERCONNECTION NETWORKS TAXONOMY

In this section, we introduce a topology-based taxonomy for interconnection networks (INs). An interconnection network could be either static or dynamic. Connections in a static network are fixed links, while connections in a dynamic network

……..

Figure 2.1 A topology-based taxonomy for interconnection networks.

are established on the fly as needed. Static networks can be further classified according to their interconnection pattern as one-dimension (1D), two-dimension (2D), or hypercube (HC). Dynamic networks, on the other hand, can be classified based on interconnection scheme as bus-based versus switch-based. Bus-based networks can further be classified as single bus or multiple buses. Switch-based dynamic networks can be classified according to the structure of the interconnection network as single-stage (SS), multistage (MS), or crossbar networks. Figure 2.1 illustrate this taxonomy. In the following sections, we study the different types of dynamic and static interconnection networks.

2.2 BUS-BASED DYNAMIC INTERCONNECTION NETWORKS

Trong khi đó

khác nhau

2.2.1 Single Bus Systems

A single bus is considered the simplest way to connect multiprocessor systems. Figure 2.2 shows an illustration of a single bus system. In its general form, such a system consists of N processors, each having its own cache, connected by a

……
…..

Figure 2.2 Example single bus system.

shared bus. The use of local caches reduces the processor-memory traffic. All processors communicate with a single shared memory. The typical size of such a system varies between 2 and 50 processors. The actual size is determined by the traffic per processor and the bus bandwidth (defined as the maximum rate at which the bus can propagate data once transmission has started). The single bus network complexity, measured in terms of the number of buses used, is O(1), while the time complexity, measured in terms of the amount of input to output delay is O(N).

Although simple and easy to expand, single bus multiprocessors are inherently limited by the bandwidth of the bus and the fact that only one processor can access the bus, and in turn only one memory access can take place at any given time. The characteristics of some commercially available single bus computers are summarized in Table 2.1.

2.2.2 Multiple Bus Systems

The use of multiple buses to connect multiple processors is a natural extension to

the single shared bus system. A multiple bus multiprocessor system uses several parallel buses to interconnect multiple processors and multiple memory modules. A number of connection schemes are possible in this case. Among the possibilities are the multiple bus with full bus-memory connection (MBFBMC), multiple bus with single bus memory connection (MBSBMC), multiple bus with partial bus- memory connection (MBPBMC), and multiple bus with class-based memory connection (MBCBMC). Illustrations of these connection schemes for the case of N = 6 processors, M = 4 memory modules, and B = 4 buses are shown in Figure 2.3. The multiple bus with full bus-memory connection has all memory modules connected to all buses. The multiple bus with single bus-memory connection has each memory module connected to a specific bus. The multiple bus with partial bus -memory connection has each memory module connected to a subset of buses. The multiple bus with class-based memory connection has memory modules grouped into classes whereby each class is connected to a specific subset of buses. A class is just an arbitrary collection of memory modules.

One can characterize those connections using the number of connections required and the load on each bus as shown in Table 2.2. In this table, k represents the number of classes; g represents the number of buses per group, and Mj represents the number of memory modules in class j.

TABLE 2.1 Characteristics of Some Commercially Available Single Bus Systems

In general, multiple bus multiprocessor

Một trong số đó có thể là kết nối

kết nối

kết nối

Hình 2.3

kết nối

kết nối

kết nối

kết nối

mô đu bộ nhớ tùy ý

biểu diễn

biểu diễn

số lớp biểu diễn

organization offers a number of desirable features such as high reliability and ease of incremental growth. A single bus failure will leave (B — 1) distinct fault-free paths between the processors and the memory modules. On the other hand, when the number of buses is less than the number of memory modules (or the number of processors), bus contention is expected to increase.

Figure 2.3 (a) Multiple bus with full bus-memory connection (MBFBMC); (b) multiple bus with single bus-memory connection (MBSBMC); (c) multiple bus with partial bus- memory connection (MBPBMC); and (d) multiple bus with class-based memory connection (MBCBMC).

Figure 2.3 Continued.

2.2.3 Bus Synchronization

A bus can be classified as synchronous or asynchronous. The time for any transaction over a synchronous bus is known in advance. In accepting and/or generating information over the bus, devices take the transaction time into account. Asynchronous bus, on the other hand, depends on the availability of data and the readiness of devices to initiate bus transactions.

In a single bus multiprocessor system, bus arbitration is required in order to resolve the bus contention that takes place when more than one processor competes to access the bus. In this case, processors that want to use the bus submit their requests to bus arbitration logic. The latter decides, using a certain priority scheme, which processor will be granted access to the bus during a certain time interval (bus master). The

có

bị đường không bị lỗi

Kết nối

kết nối

kết nối

kết nối

2.2.3 Bus Đồng Bộ

của mọi

(giao tác) đã được biết trước Trong quá trình

xét đến thời gian trao đổi (giao tác) Trái lại

(giao tác bus)

(điều phối kênh)

logic phân xử bus Hệ thống này đưa ra quyết định

cấp quyền truy cập vào bus cho bộ vi xử lý

process of passing bus mastership from one processor to another is called handshaking and requires the use of two control signals: bus request and bus grant. The first indicates that a given processor is requesting mastership of the bus, while the second indicates that bus mastership is granted. A third signal, called bus busy, is usually used to indicate whether or not the bus is currently being used. Figure 2.4 illustrates such a system.

In deciding which processor gains control of the bus, the bus arbitration logic uses a predefined priority scheme. Among the priority schemes used are random

TABLE 2.2 Characteristics of Multiple Bus Architectures

Figure 2.4 Bus handshaking mechanism (a) the scheme; and (b) the timing.

priority, simple rotating priority, equal priority, and least recently used (LRU) priority. After each arbitration cycle, in simple rotating priority, all priority levels are reduced one place, with the lowest priority processor taking the highest priority. In equal priority, when two or more requests are made, there is equal chance of any one request being processed. In the LRU algorithm, the highest priority is given to the processor that has not used the bus for the longest time.

2.3 SWITCH-BASED INTERCONNECTION NETWORKS

In this type of network, connections among

processors and memory modules are made using simple switches. Three basic interconnection topologies exist: crossbar, single-stage, and multistage.

2.3.1 Crossbar Networks

A crossbar network represents the other extreme to the limited single bus network. While the single bus can provide only a single connection, the crossbar can provide simultaneous connections among all its inputs and all its outputs. The crossbar contains a switching element (SE) at the intersection of any two lines extended horizontally or vertically inside the switch. Consider, for example the 8 x 8 crossbar network shown in Figure 2.5. In this case, an SE (also called a cross-point) is provided at each of the 64 intersection points (shown as small squares in Fig. 2.5). The figure illustrates the case of setting the SEs such that simultaneous connections between p and M8_i+1 for 1 < i < 8 are made. The two possible settings of an SE in the crossbar (straight and diagonal) are also shown in the figure.

As can be seen from the figure, the number of SEs (switching points) required is 64 and the message delay to traverse from the input to the output is constant, regardless of which input/output are communicating. In general for an N x N crossbar, the network complexity, measured in terms of the number of switching points, is O(N 2) while the time complexity, measured in terms of the input to output delay, is O(1). It should be noted that the complexity of the crossbar

network pays off in the form of reduction in the time complexity. Notice also that the crossbar is a nonblocking network that allows a multiple input-output connection pattern (permutation) to be achieved simultaneously. However, for a large multiprocessor system the complexity of the crossbar can become a dominant financial factor.

2.3.2 Single-Stage Networks

In this case, a single stage of switching elements (SEs) exists between the inputs and the outputs of the network. The simplest switching element that can be used is the

Figure 2.5 An 8 x 8 crossbar network (a) straight switch setting; and (b) diagonal switch setting.

Figure 2.6 The different settings of the 2 x 2 SE.

2 x 2 switching element (SE). Figure 2.6 illustrates the four possible settings that an SE can assume. These settings are called straight, exchange, upper-broadcast, and lower-broadcast. In the straight setting, the upper input is transferred to the upper output and the lower input is transferred to the lower output. In the exchange setting the upper input is transferred to the lower output and the lower input is transferred to the upper output. In the upper-broadcast setting the upper input is broadcast to both the upper and the lower outputs. In the lower-broadcast the lower input is broadcast to both the upper and the lower outputs.

To establish communication between a given input (source) to a given output (destination), data has to be circulated a

Và

cho phép đạt được đồng thời

Các

nhất

cách thiết lập hoán đổi

quá trình

tới

number of times around the network. A well-known connection pattern for interconnecting the inputs and the outputs of a single-stage network is the Shuffle-Exchange. Two operations are used. These can be defined using an m bit-wise address pattern of the inputs, Pm-1Pm-2 • • -p^0, as follows:

With shuffle (S) and exchange (E) operations, data is circulated from input to output until it reaches its destination. If the number of inputs, for example, processors, in a single-stage IN is N and the number of outputs, for example, memories, is N, the number of SEs in a stage is N/2. The maximum length of a path from an input to an output in the network, measured by the number of SEs along the path, is log2 N.

Example In an 8-input single stage Shuffle-Exchange if the source is 0 (000) and the destination is 6 (110), then the following is the required sequence of Shuffle/ Exchange operations and circulation of data:
……

The network complexity of the single-stage interconnection network is O(N) and the time complexity is O(N).
In addition to the shuffle and the exchange functions, there exist a number of other interconnection patterns that are used in forming the interconnections among stages in interconnection networks. Among these are the Cube and the Plus-Minus 2'(PM2I) networks. These are introduced below.

The Cube Network The interconnection pattern used in the cube network is defined

phổ biến

thao tác

dạng thức địa chỉ từng bít của

thao tác di chuyển

Chẳng hạn như,

được

Hóan vị-Di chuyển

as follows:

……

Consider a 3-bit address (N = 8), then we have C2(6) = 2, C1 (7) = 5 and C0(4) = 5. Figure 2.7 shows the cube interconnection patterns for a network with N = 8.

The network is called the cube network due to the fact that it resembles the interconnection among the corners of an n-dimensional cube (n = log2 N) (see Fig. 2.16e, later).

The Plus-Minus 2i (PM2I) Network The PM2I network consists of 2k interconnection functions defined as follows:

……

For example, consider the case N = 8, PM2+1(4) = 4 + 21 mod 8 = 6. Figure 2.8 shows the PM2I for N = 8. It should be noted that PM2+(k-1)(P) = PM2_(k-1)(P)VP, 0 < P < N. It should also be noted that PM2+2 = C2. This last observation indicates that it should be possible to use the PM2I network to perform at least part of the connections that are parts of the Cube network (simulating the Cube network using the PM2I network) and the reverse is also possible. Table 2.3 provides the lower and the upper bounds on network simulation times for the three networks PM2I, Cube, and Shuffle-Exchange. In this table the entries at the intersection of a given row and a given column are the lower and the upper

……..

Figure 2.7 The cube network for N = 8 (a) C0; (b) C1; and (c) C2

[redacted]

bounds on the time required for the network in the row to simulate the network in the column (see the exercise at the end of the chapter).

The Butterfly Function The interconnection pattern used in the butterfly network is defined as follows:

……..

Consider a 3-bit address (N = 8), the following is the butterfly mapping:

……..

TABLE 2.3 Network Simulation Time for Three Networks

……..

2.3.3 Multistage Networks

Multistage interconnection networks (MINs) were introduced as a means to improve some of the limitations of the single bus system while keeping the cost within an affordable limit. The most undesirable single bus limitation that MINs is set to improve is the availability of only one single path between the processors and the memory modules. Such MINs provide a number of simultaneous paths between the processors and the memory modules.

As shown in Figure 2.9, a general MIN consists of a number of stages each consisting of a set of 2 x 2 switching elements. Stages are connected to each other using Inter-stage Connection (ISC) Pattern. These patterns may follow any of the routing functions such as Shuffle-Exchange, Butterfly, Cube, and so on.

Figure 2.10 shows an example of an 8 x 8 MIN that uses the 2 x 2 SEs described before. This network is known in the literature as the Shuffle-Exchange network (SEN). The settings of the SEs in the figure illustrate how a number of paths can be

trong khoảng

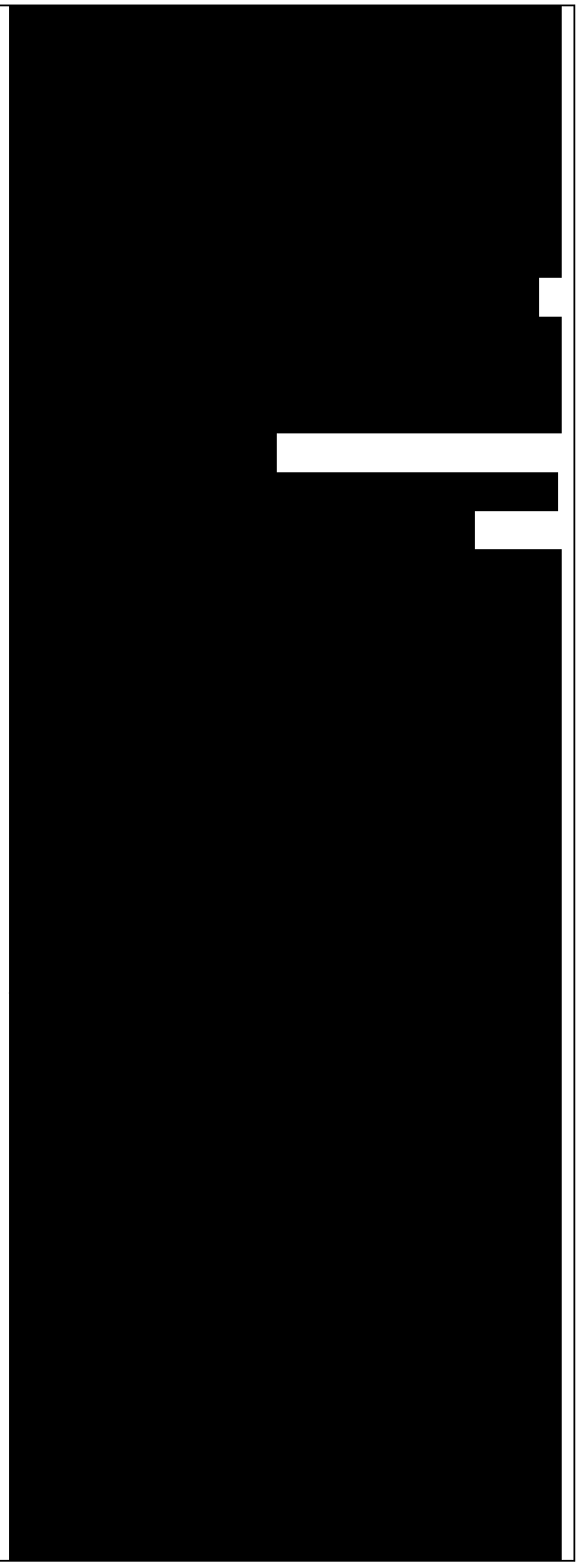
quá trình ánh xạ

mô phỏng mạng

đối với

established simultaneously in the network. For example, the figure shows how three simultaneous paths connecting the three pairs of input/output 000 ! 101, 101 ! 011, and 110 ! 010 can be established. It should be noted that the interconnection pattern among stages follows the shuffle operation.

In MINs, the routing of a message from a given source to a given destination is based on the destination address (self-routing). There exist log2 N stages in an



……

Figure 2.9 Multistage interconnection network.

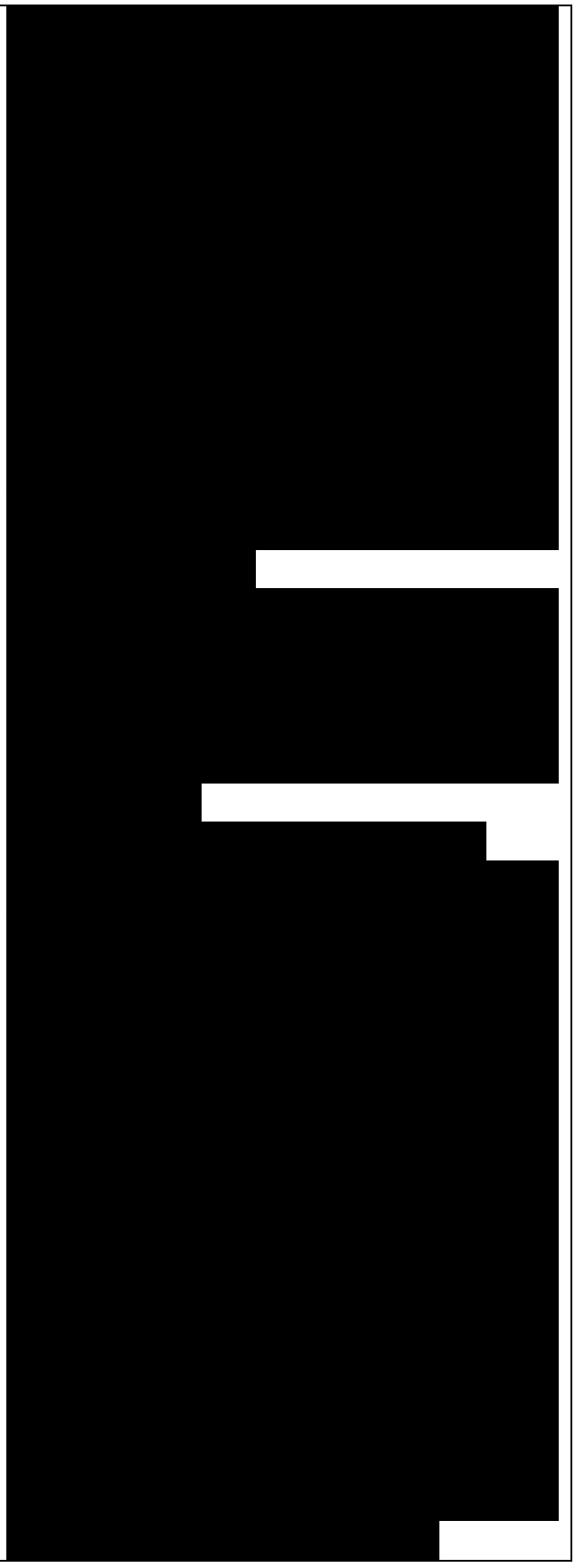
N x N MIN. The number of bits in any destination address in the network is log2 N. Each bit in the destination address can be used to route the message through one stage. The destination address bits are scanned from left to right and the stages are traversed from left to right. The first (most significant bit) is used to control the routing in the first stage; the next bit is used to control the routing in the next stage, and so on. The convention used in routing messages is that if the bit in the destination address controlling the routing in a given stage is 0, then the message is routed to the upper output of the switch. On the other hand if the bit is 1, the mess-age is routed to the lower output of the switch. Consider, for example, the routing of a message from source input 101 to destination output 011 in the 8 x 8 SEN shown in Figure 2.10. Since the first bit of the destination address is 0, therefore the message is first routed to the upper output of the switch in the first (leftmost) stage. Now, the next bit in the destination address is 1, thus the message is routed to the lower output of the switch in

the middle stage. Finally, the last bit is 1, causing the message to be routed to the lower output in the switch in the last stage. This sequence causes the message to arrive at the correct output (see Fig. 2.10). Ease of message routing in MINs is one of the most desirable features of these networks.

The Banyan Network A number of other MINs exist, among these the Banyan network is well known. Figure 2.11 shows an example of an 8 x 8 Banyan network. The reader is encouraged to identify the basic features of the Banyan network.

Figure 2.11 An 8 x 8 Banyan network.

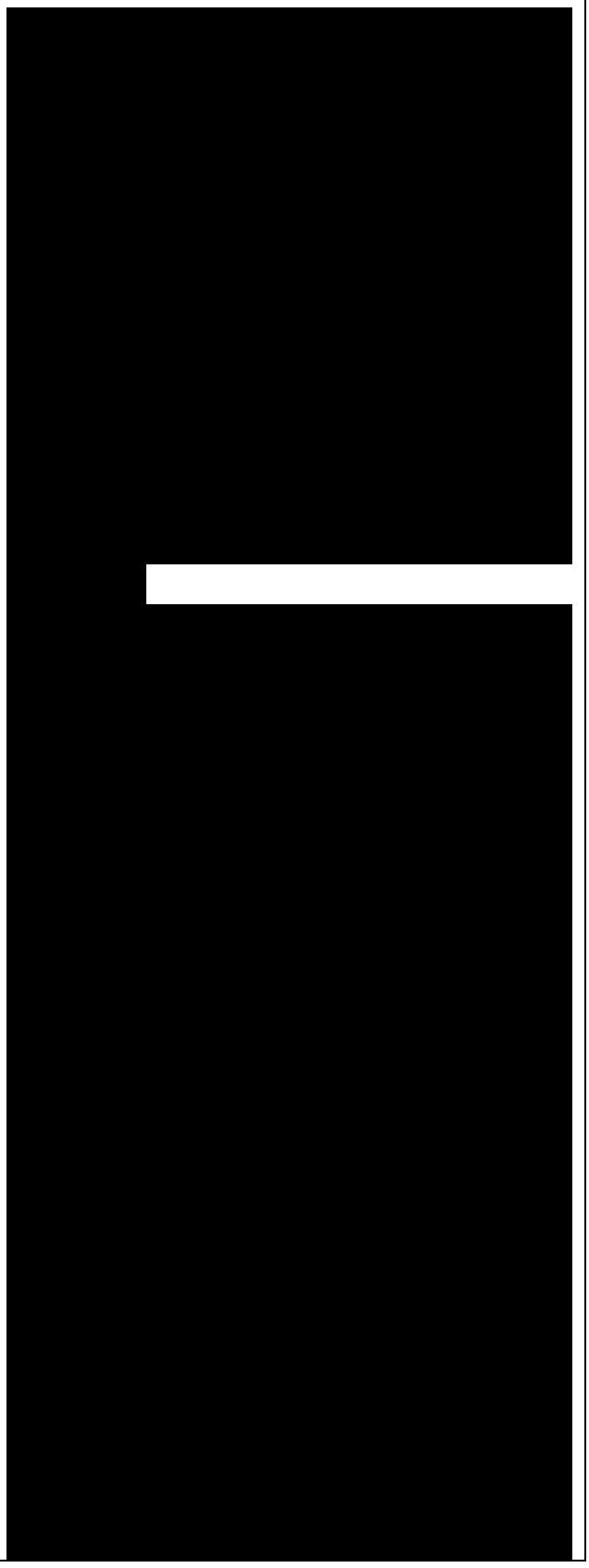
If the number of inputs, for example, processors, in an MIN is N and the number of outputs, for example, memory modules, is N, the number of MIN stages is log2 N and the number of SEs per stage is N/2, and hence the network complexity, measured in terms of the total number of SEs is O(N x log2 N). The number of SEs along the path is usually taken as a measure of the delay a message has to encounter as it finds its way from a source input to a destination output. The time complexity, measured by the number of SEs along the path from input to output, is O(log2 N). For example, in a 16 x 16 MIN, the length of the path from input to output is 4. The total number of SEs in the network is usually taken as a measure for the total area of the network. The total area



of a 16 x 16 MIN is 32 SEs.

The Omega Network The Omega Network represents another well-known type of MINs. A size N omega network consists of n (n = log2 N single-stage) Shuffle-Exchange networks. Each stage consists of a column of N/2, two-input switching elements whose input is a shuffle connection. Figure 2.12 illustrates the case of an N = 8 Omega network. As can be seen from the figure, the inputs to each stage follow the shuffle interconnection pattern. Notice that the connections are identical to those used in the 8 x 8 Shuffle-Exchange network (SEN) shown in Figure 2.10.

Owing to its versatility, a number of university projects as well as commercial MINs have been built. These include the Texas Reconfigurable Array Computer (TRAC) at the University of Texas at Austin, the Cedar at the University of Illinois at Urbana-Champaign, the RP3 at IBM, the Butterfly by BBN Laboratories, and the NYU Ultracomputer at New York University. The NYU Ultracomputer is an experimental shared memory MIMD architecture that could have as many as 4096 processors connected through an Omega MIN to 4096 memory modules. The MIN is an enhanced network that can combine two or more requests bound for the same memory address. The network interleaves consecutive memory addresses across the memory modules in order to reduce conflicts in accessing different data elements. The switch nodes in the NYU Ultracomputer are provided with queues (queue lengths of 8 to 10 messages) to handle messages collision at the switch. The system achieves one-cycle processor to



memory access.

2.3.4 Blockage in Multistage Interconnection Networks

A number of classification criteria exist for MINs. Among these criteria is the criterion of blockage. According to this criterion, MINs are classified as follows.

.....

Rearrangeable Networks Rearrangeable networks are characterized by the property that it is always possible to rearrange already established connections in order to make allowance for other connections to be established simultaneously. The Benes is a well-known example of rearrangeable networks. Figure 2.13 shows an example 8 x 8 Benes network. Two simultaneous connections are shown established in the network. These are 110 ! 100 and 010 ! 110. In the presence of the

……..

Figure 2.13 Illustration of the rearrangeability of the Benes network (a) Benes network with two simultaneously established paths; and (b) the rearrangement of connection 110 ! 100 in order to satisfy connection 101 ! 001.

connection 110 ! 100, it will not be possible to establish the connection 101 ! 001 unless the connection 110 ! 100 is rearranged as shown in part (b) of the figure.

Nonblocking Networks Nonblocking networks are characterized by the property that in the presence of a currently established connection between any pair of input/output, it will always be possible to establish a connection between any arbitrary unused pair of input/output. The Clos is a

well-known example of nonblocking networks. It consists of r1n1 x m input crossbar switches (r1 is the number of input crossbars, and n1 x m is the size of each input crossbar), mr1 x r2 middle crossbar switches (m is the number of middle crossbars, and r1 x r2 is the size of each middle crossbar), and r2m x n2 output crossbar switches (r2 is the number of output crossbars and m x n2 is the size of each output crossbar). The Clos network is not blocking if the following inequality is satisfied m > n1 + n2 — 1.

A three-stage Clos network is shown in Figure 2.14. The network has the following parameters: r1 = 4, n1 = 2, m = 4, r2 = 4, and n2 = 2. The reader is encouraged to ascertain the nonblocking feature of the network shown in Figure 2.14 by working out some example simultaneous connections. For example show that in the presence of a connection such as 110 to 010, any other connection will be possible. Note that Clos networks will be discussed again in Chapter 7.

2.4 STATIC INTERCONNECTION NETWORKS

Static (fixed) interconnection networks are characterized by having fixed paths, unidirectional or bidirectional, between processors. Two types of static networks can be identified. These are completely connected networks (CCNs) and limited connection networks (LCNs).

……..

Figure 2.14 A three-stage Clos network.

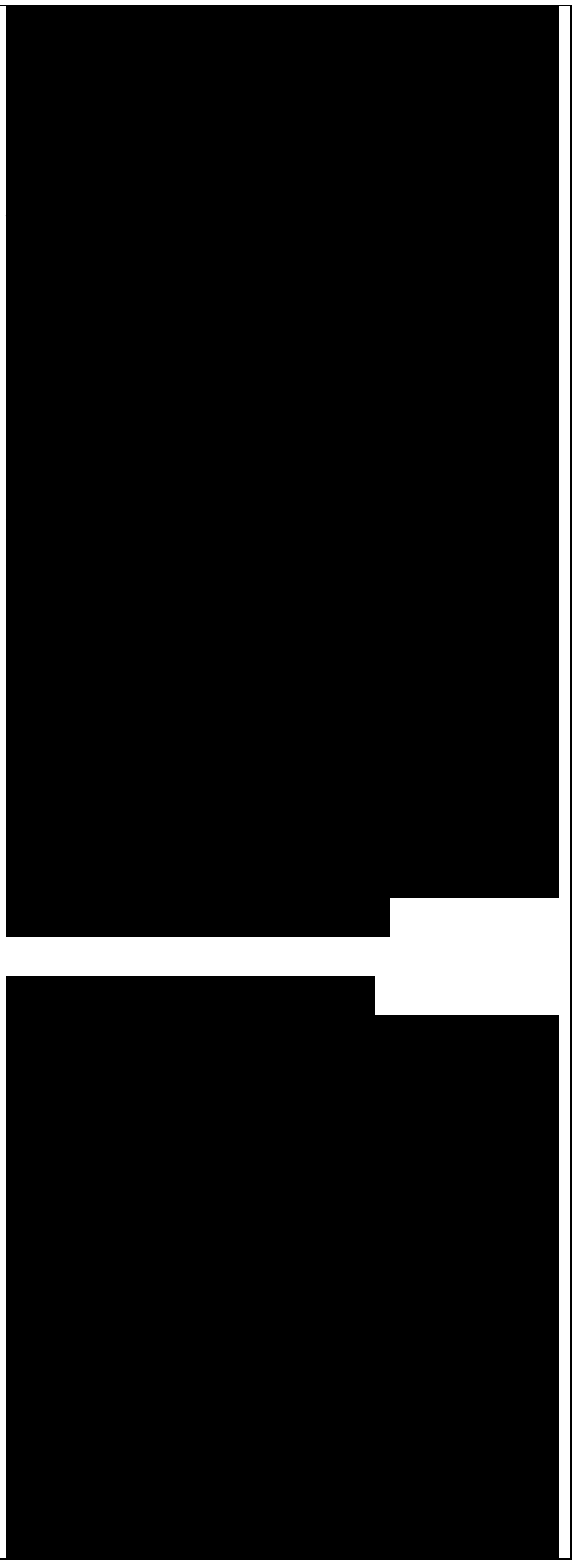
2.4.1 Completely Connected Networks

In a completely connected network (CCN) each node is connected to all other nodes in the network. Completely connected

networks guarantee fast delivery of messages from any source node to any destination node (only one link has to be traversed). Notice also that since every node is connected to every other node in the network, routing of messages between nodes becomes a straightforward task. Completely connected networks are, however, expensive in terms of the number of links needed for their construction. This disadvantage becomes more and more apparent for higher values of N. It should be noted that the number of links in a completely connected network is given by N(N — 1)/2, that is, O(N2). The delay complexity of CCNs, measured in terms of the number of links traversed as messages are routed from any source to any destination is constant, that is, O(1). An example having N = 6 nodes is shown in Figure 2.15. A total of 15 links are required in order to satisfy the complete interconnectivity of the network.

2.4.2 Limited Connection Networks

Limited connection networks (LCNs) do not provide a direct link from every node to every other node in the network. Instead, communications between some nodes have to be routed through other nodes in the network. The length of the path between nodes, measured in terms of the number of links that have to be traversed, is expected to be longer compared to the case of CCNs. Two other conditions seem to have been imposed by the existence of limited interconnectivity in LCNs. These are: the need for a pattern of interconnection among nodes and the need for a mechanism for routing messages around the network until they reach their destinations. These two



items are discussed below.

A number of regular interconnection patterns have evolved over the years for LCNs These patterns include:

- linear arrays;
- ring (loop) networks;
- two-dimensional arrays (nearest-neighbor mesh);
- tree networks; and
- cube networks.

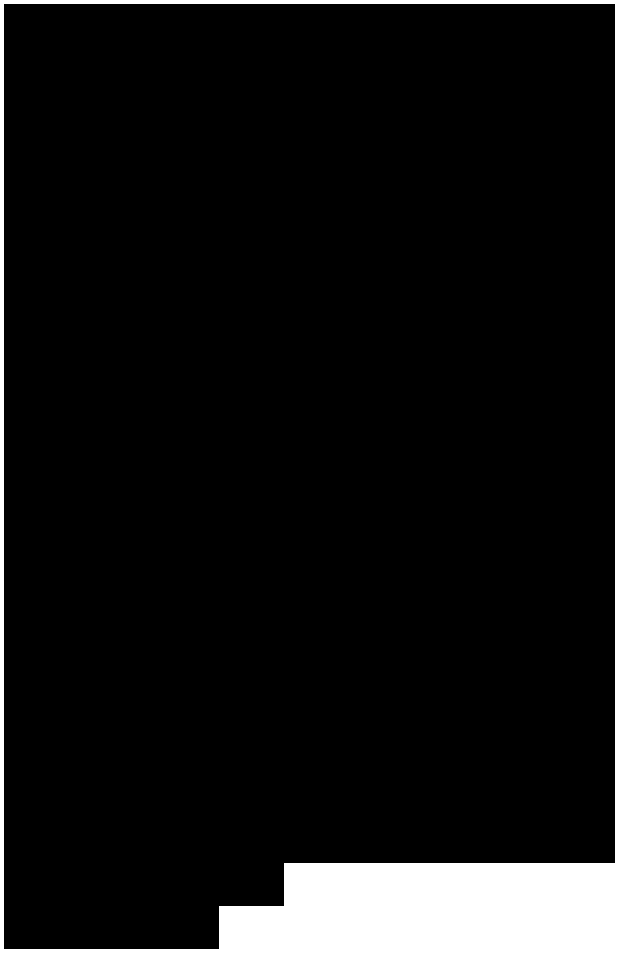
Simple examples for these networks are shown in Figure 2.16.

In a linear array, each node is connected to its two immediate neighboring nodes. The two nodes at the extreme ends of the array are connected to their single immediate neighbor. If node i needs to communicate with node j, j > i, then the message from node i has to traverse nodes i + 1, i + 2,..., j — i. Similarly, when node i needs to communicate with node j, where i > j, then the message from node i has to traverse nodes i — 1, i — 2,..., i — j. In the worst possible case, when node 1 has to send a message to node N, the message has to traverse a total of N— 1 nodes before it can reach its destination. Therefore, although linear arrays are simple in their architecture and have simple routing mechanisms, they tend to be slow. This is particularly true when the number of nodes N is large. The network complexity of the linear array is O(N) and its time complexity is O(N). If the two nodes at the extreme ends of a linear array network are connected, then the resultant network has ring (loop) architecture.

In a tree network, of which the binary tree (shown in Fig. 2.16d) is a special case, if a

node at level i (assuming that the root node is at level 0) needs to communicate with a node at level j, where i > j and the destination node belongs to the same root's child subtree, then it will have to send its message up the tree traversing nodes at levels i — 1, i — 2,..., j + 1 until it reaches the destination node. If a node at level i needs to communicate with another node at the same level i (or with node at level j = i where the destination node belongs to a different root's child subtree), it will have to send its message up the tree until the message reaches the root node at level 0. The message will have to be then sent down from the root nodes until it reaches its destination. It should be noted that the number of nodes (processors) in a binary tree system having k levels can be calculated as:

……..



Notice also that the maximum depth of a binary tree system is |log2 N], where N is the number of nodes (processors) in the network. Therefore, the network complexity is O(2k) and the time complexity is O(log2 N).

The cube-connected and the mesh-connected networks have been receiving increasing interest and, therefore, we discuss them in more detail in the following subsections.

2.4.3 Cube-Connected Networks

Cube-connected networks are patterned after the n-cube structure. An n-cube (hypercube

of order n) is defined as an undirected graph having 2n vertices labeled 0 to 2n — 1 such that there is an edge between a given pair of vertices if and only if the binary representation of their addresses differs by one and only one bit. A 4-cube is shown in Figure 2.17. In a cube-based multiprocessor system, processing elements are positioned at the vertices of the graph. Edges of the graph represent the point-to- point communication links between processors. As can be seen from the figure, each processor in a 4-cube is connected to four other processors. In an n-cube, each processor has communication links to n other processors. Recall that in a hypercube, there is an edge between a given pair of nodes if and only if the binary representation of their addresses differs by one and only one bit. This property allows for a simple message routing mechanism. The route of a message originating at node i and destined for node j can be found by XOR-ing the binary address representation of i and j. If the XOR-ing operation results in a 1 in a given bit position, then the message has to be sent along the link that spans the corresponding dimension. For example, if a message is sent from source (S) node 0101 to destination (D) node 1011, then the XOR operation results in 1110. That will mean that the message will be sent only along dimensions 2, 3, and 4 (counting from right to left) in order to arrive at the destination. The order in which the message traverses the three dimensions is not important. Once the message traverses the three dimensions in any order it will reach its destination. The three possible disjoint routes that can be taken by the message in

this example are shown in bold in Figure 2.17. Disjoint routes do not share any common links among them.

In an n-cube, each node has a degree n. The degree of a node is defined as the number of links incident on the node. The upper limit on the number of disjoint paths in an n-cube is n. The hypercube is referred to as a logarithmic architecture. This is because the maximum number of links a message has to traverse in order to reach its destination in an n-cube containing N = 2n nodes is log2 N = n links. One of the desirable features of hypercube networks is the recursive nature of their constructions. An n-cube can be constructed from two subcubes each having an (n — 1) degree by connecting nodes of similar addresses in both subcubes. Notice that the 4-cube shown in Figure 2.17 is constructed from two subcubes each of degree three. Notice that the construction of the 4-cube out of the two 3-cubes requires an increase in the degree of each node. It is worth mentioning that the Intel iPSC is an example of hypercube-based commercial multiprocessor systems. A number of performance issues of hypercube multiprocessors will be discussed in Section 2.5.

A number of variations to the basic hypercube interconnection have been proposed. Among these is the cube-connected cycle architecture. In this architecture, 2n+r nodes are connected in an n-cube fashion such that groups of r nodes

each form cycles (loops) at the vertices of the cube. For example, a 3-cube connected cycle network with r = 3 will have three nodes (processors) forming a loop (ring) at each vertex of the 3-cube. The idea of cube-connected cycles has not been widely used.

2.4.4 Mesh-Connected Networks

An n-dimensional mesh can be defined as an interconnection structure that has K0 x K1 x---x K,,_i nodes where n is the number of dimensions of the network and K is the radix of dimension i. Figure 2.18 shows an example of a 3 x 3 x 2 mesh network. A node whose position is (i, j, k) is connected to its neighbors at dimensions i + 1, j + 1, and k + 1. Mesh architecture with wrap around connections forms a torus. A number of routing mechanisms have been used to route messages around meshes. One such routing mechanism is known as the dimension-ordering routing. Using this technique, a message is routed in one given dimension at a time, arriving at the proper coordinate in each dimension before proceeding to the next dimension. Consider, for example, a 3D mesh. Since each node is represented by its position (i, j, k), then messages are first sent along the i dimension, then along the j dimension, and finally along the k dimension. At most two turns will be allowed and these turns will be from i to j and then from j to k. In Figure 2.18 we show the route of a message sent from node S at position (0, 0, 0) to node D at position (2, 1, 1). Other routing mechanisms in meshes have been proposed. These include dimension reversal routing, the turn model routing, and node labeling routing. Readers are referred to the bibliography for more information on those,

and other routing mechanisms. It should be noted that for a mesh interconnection network with N nodes, the longest distance traveled between any two arbitrary nodes is 0(VN).

Multiprocessors with mesh interconnection networks are able to support many scientific computations very efficiently. It is also known that n-dimensional meshes can be laid out in n dimensions using only short wires and built using identical boards, each requiring only a small number of pins for connections to other boards. Another advantage of mesh interconnection networks is that they are scalable. Larger meshes can be obtained from smaller ones without changing the node degree (a node degree is defined as the number of links incident on the node). Because of these features, a large number of distributed memory parallel computers utilize mesh interconnection networks. Examples include MPP from Goodyear Aerospace, Paragon from Intel, and J-Machine from MIT.

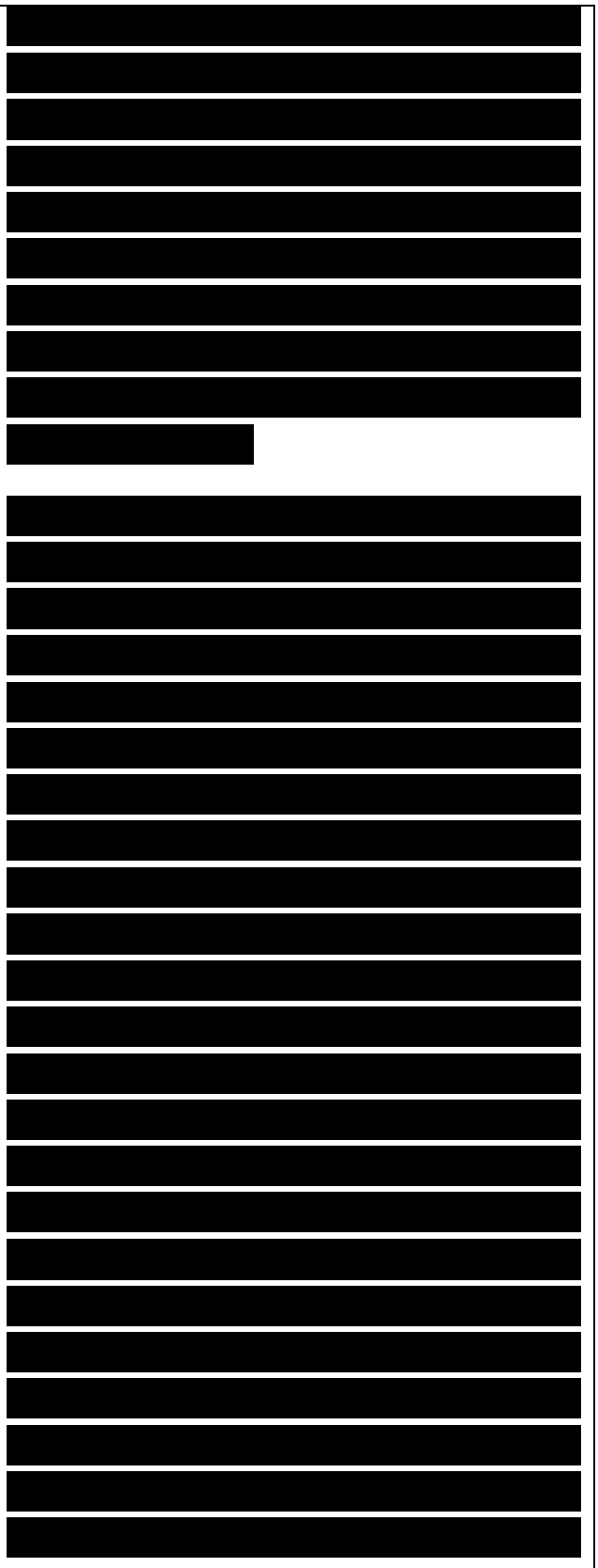
2.4.5 The k-ary n-Cube Networks

The k-ary n-cube network is a radix k cube with n dimensions. The radix implies that there are k nodes in each dimension. An 8-ary 1-cube is simply an eight node ring, while an 8-ary 2-cube is eight 8-node rings connected such that nodes are connected to all nodes with an address that differs in only one digit (see Figure 2.19 Examples of k-ary n-cube networks (a) 8-ary 1-cube (8 nodes ring) network; and (b) 8-ary 2-cube (eight 8-node rings) network.

Fig. 2.19). It should be noted that the number of nodes in a k-ary n-cube is N = kn nodes and that when k = 2, the architecture becomes a binary n-cube. Routing of messages in a k-ary n-cube network can be done in a similar way to that used in mesh networks. Figure 2.19 illustrates a possible path for a message sent from a source node (S) to a destination node (D). Notice that, depending on the directionality of links among nodes the possible route(s) will be decided. Another factor involved in the selection of route in a k-ary n-cube network is the minimality of the route, measured in terms of the number of hops (links) traversed by a message before reaching its destination. The length of the route between S and D in Figure 2.19b is 6. Notice that other routes exist between S and D but they are longer than the indicated route. The longest distance traveled between any two arbitrary nodes in a k-ary n-cube network is O(n + k).

2.5 ANALYSIS AND PERFORMANCE METRICS

Having introduced the main architecture of multiprocessors, we now turn our attention to a discussion on the analysis and performance issues related to those architectures. We provide an introduction to the basic performance issues and performance metrics related to both static and dynamic interconnection networks. For dynamic networks, we discuss the performance issues related to cost, measured in terms of the number of cross points (switching elements), the delay (latency), the blocking characteristics, and the fault tolerance. For static networks, we discuss the performance issues related to degree,



diameter, and fault tolerance. A more detailed discussion on assessing the performance of these networks will be given in Chapter 3.

2.5.1 Dynamic Networks

The Crossbar The cost of the crossbar system can be measured in terms of the number of switching elements (cross points) required inside the crossbar. Recall that for an N x N crossbar, the network cost, measured in terms of the number of switching points, is N 2. This is because in an N x N crossbar a cross point is needed at the intersection of every two lines extended horizontally and vertically inside the switch. We, therefore, say that the crossbar possesses a quadratic rate of cost (complexity) given by O(N2). The delay (latency) within a crossbar switch, measured in terms of the amount of the input to output delay, is constant. This is because the delay from any input to any output is bounded. We, therefore, say that the crossbar possesses a constant rate of delay (latency) given by O(1). It should be noted that the high cost (complexity) of the crossbar network pays off in the form of reduction in the time (latency). However, for a large multiprocessor system the cost (complexity) of the crossbar can become a dominant financial burden. The crossbar is however a nonblocking network; that is, it allows multiple output connection pattern (permutation) to be achieved (see Fig. 2.5). The nonblocking property of the crossbar is a highly desirable feature that allows concurrent (simultaneous) processor-memory accesses to take place.

A fault-tolerant system can be simply defined as a system that can still function

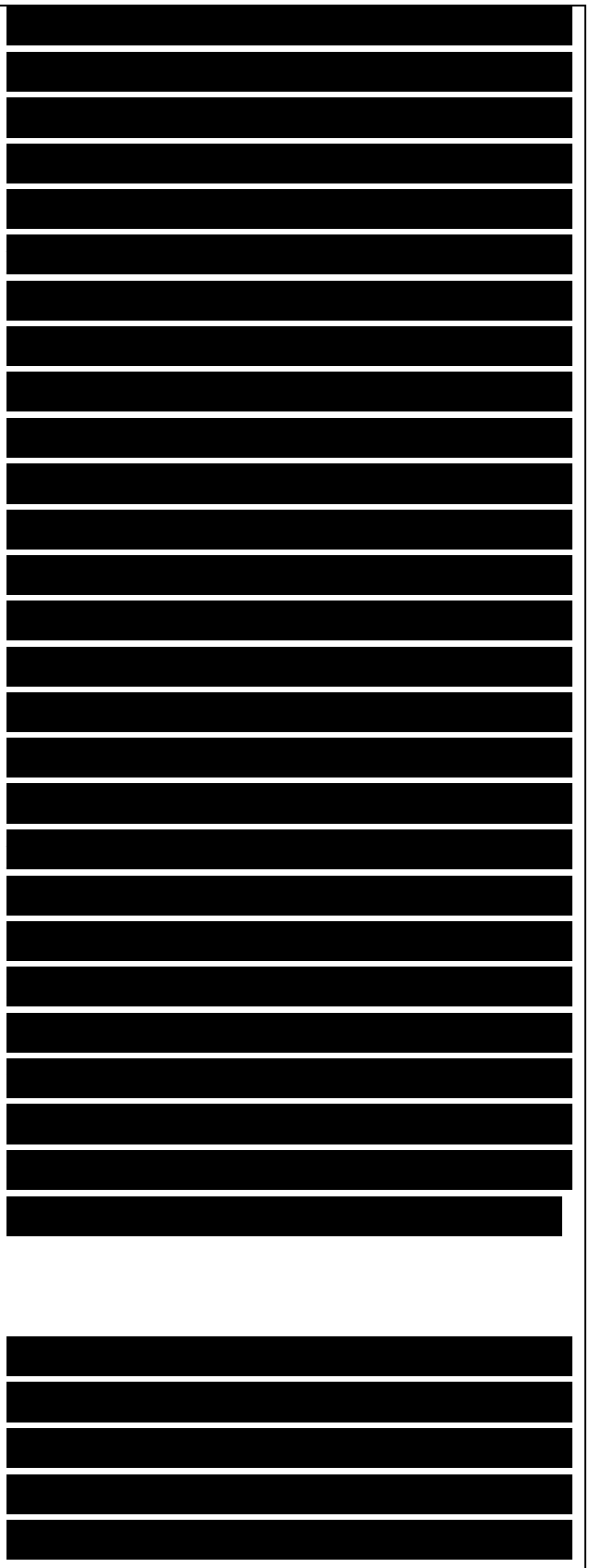
even in the presence of faulty components inside the system. Fault tolerance is a desirable feature that allows a system to continue functioning despite the fact that it contains some faulty elements. The crossbar can be affected by a single-point failure. This is because a failure of a single cross point inside the switch can lead to the crossbar being unable to provide simultaneous connections among all its inputs and all its outputs. Consider, for example the cross-point failure shown in Figure 2.20. In this case, a number of simultaneous connections are possible to make within the switch. However, a connection between P5 and M4 cannot be made. Nevertheless, segmenting the crossbar and realizing each segment independently can reduce the effect of a single-point failure in a crossbar. It may also be possible to introduce routing algorithms such that more than one path exists for the establishment of a connection between any processor-memory pair. Therefore, the existence of a faulty cross point and/or link along one path will not cause the total elimination of a connection between the processor-memory pair.

Multiple Bus In Section 2.2.2 we considered a number of different multiple bus arrangements. A general multiple bus arrangement is shown in Figure 2.21. It consists of M memory modules, Nprocessors, and B buses. A given bus is dedicated

……..

Figure 2.21 Example multiple bus system.

to a particular processor for the duration of a bus transaction. A processor-memory transfer can use any of the available buses.



Given B buses in the system, then up to B requests for memory use can be served simultaneously. The cost of a multiple-bus system such as the ones shown in Figure 2.3 is measured in terms of the number of buses used, B. We therefore say that a multiple bus possesses an O(B) rate of cost (complexity) growth. The delay (latency) of a multiple bus, measured in terms of the amount of the input to output delay, is proportional to B x N. We therefore say that the multiple bus possesses an O(B x N) rate of delay (latency) growth.

Multiple bus multiprocessor organization offers the desirable feature of being highly reliable and fault-tolerant. This is because a single bus failure in a B bus system will leave (B — 1) distinct fault-free paths between the processors and the memory modules. On the other hand, when the number of buses is less than the number of memory modules (or the number of processors), bus contention is expected to increase.

Multistage Interconnection Networks As mentioned before, the number of stages in an N x N MIN is log2 N. Each stage consists of N/2, 2 x 2 switching elements (SEs). The network cost (complexity), measured in terms of the total number of SEs, is O(N x log2 N). The number of SEs along a path from a given input to a given output is usually taken as a measure for the delay a message has to encounter as it finds its way from a source to a destination. The latency (time) complexity, measured by the number of SEs along the path from input to output, is O(log2 N).

Simplicity of message routing inside a MIN is a desirable feature of such networks. There exists a unique path between a given input-output pair. However, this feature, while simplifying the routing mechanism, causes the MIN to be vulnerable to single-point failure. The failure of a component (a switch or a link) along a given path will render the corresponding path inoperable, thus causing the disconnection of the corresponding input-output pair. Therefore, MINs are characterized as being 0-fault tolerant; that is, a MIN cannot tolerate the failure of a single component. A number of solutions have been suggested in order to improve the fault- tolerance characteristics of MINs. One such solution has been to add an extra stage of SEs such that the number of stages becomes (log2 N + 1). The addition of such a stage leads to the creation of two paths between an input-output pair and requires a minor modification in the routing strategy.

Based on the above discussion, Table 2.4 provides an overall performance comparison among different dynamic interconnection networks. Notice that in

TABLE 2.4 Performance Comparison of Dynamic Networks

…….

this table N represent the number of inputs (outputs) while m represents the number of buses.

2.5.2 Static Networks

Before discussing performance issues related to static interconnection networks, we need to introduce a number of definitions and topological characteristics:

• Degree of a node, d, is defined as the number of channels incident on the node. The number of channels into the node is the in-degree, din. The number of channels out of a node is the out-degree, dout. The total degree, d, is the sum,
d — din ^ dout.

• Diameter, D, of a network having N nodes is defined as the longest path, p, of the shortest paths between any two nodes D — max (minp[pj length(p)). In this equation, pj is the length of the path between nodes i and j and length (p) is a procedure that returns the length of the path, p. For example, the diameter of a
4 x 4 Mesh D = 6.

• A network is said to be symmetric if it is isomorphic to itself with any node labeled as the origin; that is, the network looks the same from any node. Rings and Tori networks are symmetric while linear arrays and mesh networks are not.

Having introduced the above definitions, we now proceed to introduce the basic issues related to the performance of a number of static networks.

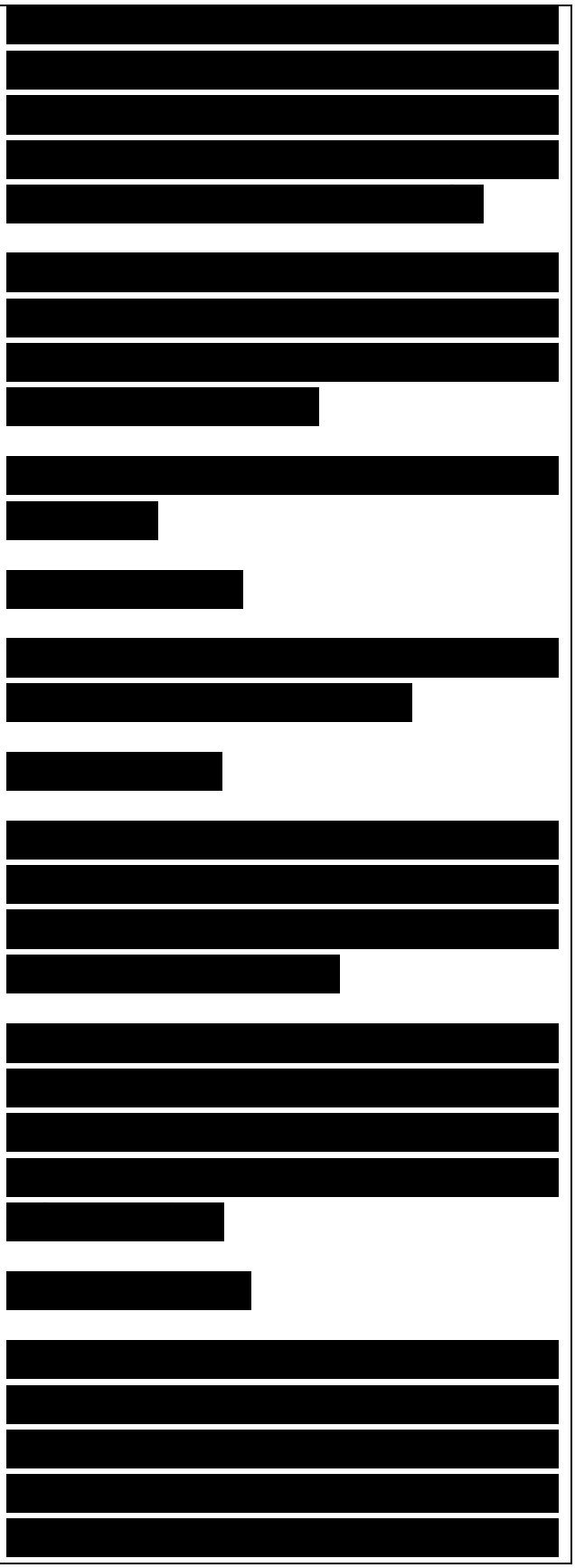
Completely Connected Networks (CCNs)
As mentioned before, in a completely connected network each node is connected to all other nodes in the network. Thus, the cost of a completely connected network having N nodes, measured in terms of the number of links in the network, is given by N(N — l)/2, that is, O(N2). The delay (latency) complexity of CCNs, measured in terms of the number of links traversed as messages are routed from any source to any destination, is constant, that is, O(1). Notice

also that the degree of a node in CCN is N — 1, that is, O(N), while the diameter is O(1).

Linear Array Networks In this network architecture, each node is connected to its two immediate neighboring nodes. Each of the two nodes at the extreme ends of the network is connected only to its single immediate neighbor. The network cost (complexity) measured in terms of the number of nodes of the linear array is O(N). The delay (latency) complexity measured in terms of the average number of nodes that must be traversed to reach from a source node to a destination node is N/2, that is, O(N). The node degree in the linear array is 2, that is, O(1) and the diameter is (N — 1), that is, O(N).

Tree Networks In a tree-connected network, a given node is connected to both its parent node and to its children nodes. In a k-level complete binary tree network, the network cost (complexity) measured in terms of the number of nodes in the network is O(2k) and the delay (latency) complexity is O(log2 N). The degree of a node in a binary tree is 3, that is, O(1), while the diameter is O(log2 N).

Cube-Connected Networks An n-cube network has 2n nodes where two nodes are connected if the binary representation of their addresses differs by one and only one bit. The cost (complexity) of an n-cube measured in terms of the number of nodes in the cube is O(2n) while the delay (latency) measured in terms of the number of nodes traversed while going from a source node to a destination node is O(log2 N). The node degree in an n-cube is O(log2 N) and the diameter of an n-cube is O(log2 N).



Mesh-Connected Networks A 2D mesh architecture connects n x n nodes in a 2D manner such that a node whose position is (i, j) is connected to its neighbors at positions (i + 1, j + 1). The cost (complexity) of a 2D mesh measured in terms of the number of nodes is O(n2), while the delay (latency) measured in terms of the number of nodes traversed while going from a source to a destination is O(n). The node degree in a 2D mesh is 4 and the diameter is O(n).

The k-ary n-Cube Networks The k-ary n-cube architecture is a radix k cube with n dimensions. The number of nodes in a k-ary n-cube is N = kn. The cost (complexity) measured in terms of the number of nodes is O(kn) and the delay (latency) measured in terms of the number of nodes traversed while going from a source to a destination is O(n + k). The node degree of a k-ary n-cube is 2n and the diameter is O(n x k). The relationship among the topological characteristics introduced above for a k-ary n-cube network is summarized below.

……

Having briefly discussed the basic performance characteristics of a number of static interconnection networks, Table 2.5 summarizes those topological characteristics. In this table, N is the number of nodes and n is the number of dimensions.

III. Performance Analysis of Multiprocessor Architecture

In the previous chapter, we introduced the fundamental concepts related to the design and analysis of multiple-processor systems. We have also touched upon some of the basic issues in the performance analysis of

static and dynamic interconnection networks. In this Chapter, we will build on this foundation by providing an in-depth analysis of the performance measures of parallel architectures. Our coverage in this chapter starts by introducing the concept of computational models as related to multiprocessors. The emphasis here is on the computational aspects of the processing elements (processors). Two computational models are studied, namely the equal duration processes and the parallel computation with serial sections models. In studying these models, we discuss two measures. These are the speedup factor and the efficiency. The impact of the communication overhead on the overall speed performance of multiprocessors is emphasized in these models. Having introduced the computational models, we move on to present a number of arguments in support of parallel architectures. Following that, we study a number of performance measures (metrics) of interconnection networks. We define performance metrics such as the bandwidth, worst-case delay, utilization, average distance traveled by a message, cost, and interconnectivity. We will show how to compute those measures for sample dynamic and static networks. Our coverage continues with a discussion on the scalability of parallel systems. A discussion on the important topic of benchmark performance concludes our coverage in this chapter.

[redacted]

[redacted]

	[redacted]

3.1 COMPUTATIONAL MODELS

In developing a computational model for multiprocessors, we assume that a given computation can be divided into concurrent tasks for execution on the multiprocessor. Two computational models, thus, arise. These are discussed below.

3.1.1 Equal Duration Model

In this model, it is assumed that a given task -an be divided into n equal subtasks, each of which can be executed by one processor. If fs is the execution time of the whole task using a single processor, then the time taken by each processor to execute its subtask is fm = fs/n. Since, according to this model, all processors are executing their subtasks simultaneously, then the time taken to execute the whole task is fm = fs/n. The speedup factor of a parallel system can be defined as the ratio between the time taken by a single processor to solve a given problem instance to the time taken by a parallel system consisting of n processors to solve the same problem instance.

……

The above equation indicates that, according to the equal duration model, the speedup factor resulting from using n processors is equal to the number of processors used, n.

One important factor has been overlooked in the above derivation. This factor is the

communication overhead, which results from the time needed for processors to communicate and possibly exchange data while executing their subtasks. Assume that the time incurred due to the communication overhead is called fc then the actual time taken by each processor to execute its subtask is given by

……..

S(n) = speedup factor with communication overhead

……..

The above equation indicates that the relative values of fs and fc affect the achieved speedup factor. A number of cases can then be contemplated: (1) if fc ^ fs then the potential speedup factor is approximately n; (2) if fc ^ fs then the potential speedup factor is ^/^ ^ 1; (3) if fc = fs then the potential speedup factor is n/n + 1 ffi 1, for n » 1.

In order to scale the speedup factor to a value between 0 and 1, we divide it by the number of processors, n. The resulting measure is called the efficiency, j. The efficiency is a measure of the speedup achieved per processor. According to the simple equal duration model, the efficiency j is equal to 1 if the communication overhead is ignored. However if the communication overhead is taken into consideration, the efficiency can be expressed as

……

Although simple, the equal duration model is however unrealistic. This is because it is based on the assumption that a given task can be divided into a number of equal subtasks that can be executed by a number of processors in parallel.

However, it is sufficient here to indicate that real algorithms contain some (serial) parts that cannot be divided among processors. These (serial) parts must be executed on a single processor. Consider, for example, the program segments given in Figure 3.1. In these program segments, we assume that we start with a value from each of the two arrays (vectors) a and b stored in a processor of the available n processors. The first program block (enclosed in a square) can be done in parallel; that is, each processor can compute an element from the array (vector) c. The elements of array c are now distributed among processors, and each processor has an element. The next program segment cannot be executed in parallel. This block will require that the elements of array c be communicated to one processor and are added up there. The last program segment can be done in parallel. Each processor can update its elements of a and b.

This illustrative example shows that a realistic computational model should assume the existence of (serial) parts in the given task (program) that cannot be divided. This is the basis for the following model.

3.1.2 Parallel Computation with Serial Sections Model

In this computational model, it is assumed that a fraction f of the given task (computation) is not dividable into concurrent subtasks. The remaining part (1 — f) is assumed to be dividable into concurrent subtasks. Performing similar

……

Example program segments.
~~derivations to those done in the case of the equal duration model will result in the following.~~

The time required to execute the task on n processors is tm = fts + (1 — f)(ts/n). The speedup factor is therefore given by
……
According to this equation, the potential speedup due to the use of n processors is determined primarily by the fraction of code that cannot be divided. If the task (program) is completely serial, that is, f = 1, then no speedup can be achieved regardless of the number of processors used. This principle is known as Amdahl's law. It is interesting to note that according to this law, the maximum speedup factor is given by limn!1 S(n) = 1/f. Therefore, according to Amdahl's law the improvement in performance (speed) of a parallel algorithm over a sequential one is limited not by the number of processors employed but rather by the fraction of the algorithm that cannot be parallelized. A first glance at Amdahl's law indicates that regardless of the number of processors used, there exists an intrinsic limit on the potential usefulness of using parallel architectures. For some time and according to Amdahl's law, researchers were led to believe that a substantial increase in speedup factor would not be possible by using parallel architectures. We will discuss the validity of that and similar postulates in the next section. However, let us show the effect of the

communication overhead on the speedup factor, given that a fraction, f, of the computation is not parallelizable. As stated earlier, the communication overhead should be included in the processing time. Considering the time incurred due to this communication overhead, the speedup factor is given by

……..

The maximum speedup factor under such conditions is given by

……..

The above formula indicates that the maximum speedup factor is determined not by the number of parallel processors employed but by the fraction of the computation that is not parallelized and the communication overhead.

Having considered the speedup factor, we now touch on the efficiency measure. Recall that the efficiency is defined as the ratio between the speedup factor and the number of processors, n. The efficiency can be computed as:

……

As a last observation, one has to notice that in a parallel architecture, processors must maintain a certain level of efficiency. However, as the number of processors increases, it may become difficult to use those processors efficiently. In order to maintain a certain level of processor efficiency, there should exist a relationship between the fraction of serial computation, f, and the number of processor employed (see Problem 6).

After introducing the above two computational models, we now turn our attention to a discussion on some performance laws (postulates) that were

hypothesized regarding the potential gain of parallel architectures. Among these are Grosch's, Amdahl's and Gustafson-Brasis's laws.

3.2 AN ARGUMENT FOR PARALLEL ARCHITECTURES

In this section, we introduce a number of postulates that were introduced by some well-known computer architects expressing skepticism about the usefulness of parallel architectures. We will also provide rebuttal to those concerns.

3.2.1 Grosch's Law

It was as early as the late 1940s that H. Grosch studied the relationship between the power of a computer system, P, and its cost, C. He postulated that $P = K \times C^s$, where s and K are positive constants. Grosch postulated further that the value of s would be close to 2. Simply stated, Grosch's law implies that the power of a computer system increases in proportion to the square of its cost. Alternatively, one can express the cost of a system as $C = V(P/K)$ assuming that $s = 2$. The relation between computing power and cost according to Grosch's law is shown in Figure 3.2.

……

Figure 3.2 Power-cost relationship according to Grosch's law.

According to Grosch's law, in order to sell a computer for twice as much, it must be four times as fast. Alternatively, to do a computation twice as cheaply, one has to do it four times slower. With the advances in computing, it is easy to see that Grosch's law is repealed, and it is possible to build faster and less expensive

computers over time.

3.2.2 Amdahl's Law

Recall that in Section 3.1.2 we defined the speedup factor of a parallel system as the ratio between the time taken by a single processor to solve a given problem instance to the time taken by a parallel system consisting of n processors to solve the same problem instance.

……

Similar to Grosch's law, Amdahl's law made it so pessimistic to build parallel computer systems due to the intrinsic limit set on the performance improvement (speed) regardless of the number of processors used. An interesting observation to make here is that according to Amdahl's law,f is fixed and does not scale with the problem size, n. However, it has been practically observed that some real parallel algorithms have a fraction that is a function of n. Let us assume thatf is a function of n such that limn!1 f(n) = 0. Hence,

……..

This is clearly in contradiction to Amdahl's law. It is therefore possible to achieve a linear speedup factor for large-sized problems, given that limn!1 f (n) = 0, a condition that has been practically observed. For example, researchers at the Sandia National Laboratories have shown that using a 1024-processor hypercube multiprocessor system for a number of engineering problems, a linear speedup factor can be achieved.

Consider, for example, the well-known engineering problem of multiplying a large square matrix A(m x m) by a vector

X(m) to obtain a vector, that is, C(m) — A(m x m) * X(m). Assume further that the solution of such a problem is performed on a binary tree architecture consisting of n nodes (processors). Initially, the root node stores the vector X(m) and the matrix A(m x m) is distributed row-wise among the n processors such that the maximum number of rows in any processor is [jnJ + 1. A simple algorithm to perform such computation consists of the following three steps:

……

The indivisible part of the computation (steps 1 and 3) is equal to O(m) + O(m log n). Therefore, the fraction of computation that is indivisible f (m) = (O(m) + O(m log n))/ O(m2) = O((1 + log n)/m). Notice that limm!1 f(m) = 0. Hence, contrary to Amdahl's law, a linear speedup can be achieved for such a large-sized problem.

It should be noted that in presenting the above scenario for solving the matrix vector multiplication problem, we have assumed that the memory size of each processor is large enough to store the maximum number of rows expected. This assumption amounts to us saying that with n processors, the memory is n times larger. Naturally, this argument is more applicable to message passing parallel architectures than it is to shared memory ones (shared memory and message passing parallel architectures are introduced in Chapters 4 and 5, respectively). The Gustafson- Barsis law makes use of this argument and is presented below.

3.2.3 Gustafson-Barsis's Law

In 1988, Gustafson and Barsis at Sandia Laboratories studied the paradox created by Amdahl's law and the fact that parallel architectures comprised of hundreds of processors were built with substantial improvement in performance. In introducing their law, Gustafson recognized that the fraction of indivisible tasks in a given algorithm might not be known a priori. They argued that in practice, the problem size scales with the number of processors, n. This contradicts the basis of Amdahl's law. Recall that Amdahl's law assumes that the amount of time spent on the parts of the program that can be done in parallel, (1 — f), is independent of the number of processors, n. Gustafson and Brasis postulated that when using a more powerful processor, the problem tends to make use of the increased resources. They found that to a first approximation the parallel part of the program, not the serial part, scales up with the problem size. They postulated that if s and p represent respectively the serial and the parallel time spent on a parallel system, then s + p X n represents the time needed by a serial processor to perform the computation. They therefore, introduced a new factor, called the scaled speedup factor, SS(n), which can be computed as:

……..

This equation shows that the resulting function is a straight line with a slope = (1 — n). This shows clearly that it is possible, even easier, to achieve efficient parallel performance than is implied by Amdahl's speedup formula. Speedup

should be measured by scaling the problem to the number of processors, not by fixing the problem size.

Having considered computational models and rebutted some of the criticism set forth by a number of computer architects in the face of using parallel architectures, we now move to consider some performance issues in dynamic and static interconnection networks. The emphasis will be on the performance of the interconnection networks rather than the computational aspects of the processors (the latter was considered in Section 3.1).

3.3 INTERCONNECTION NETWORKS PERFORMANCE ISSUES

In this section, we introduce a number of metrics for assessing the performance of dynamic and static interconnection networks. In introducing the metrics, we will show how to compute them for sample networks chosen from those introduced in Chapter 2. The reader is reminded to review the definitions given in Chapter 2 before proceeding with this section. In particular, the reader should review the definitions given about the diameter D, the degree d, and the symmetry of a network. In addition to those definitions, we provide the following definition.

- Channel bisection width of a network, B, is defined as the minimum number of wires that, when cut, divide the network into equal halves with respect to the number of nodes. The wire bisection is defined as the number of wires crossing this cut of the network. For example, the bisection width of a 4-cube is B = 8.

Table 3.1 provides some numerical values of the above topological characteristics for sample static networks. General expressions for the topological characteristics of a number of static interconnection networks are summarized in Table 3.2. It should be noted that in this table, N is the number of nodes and n is the number of dimensions. In presenting these expressions, we assume that the reader is familiar with their topologies as given in Chapter 2.

TABLE 3.1 Topological Characteristics of Static Networks

Network Configuration Bisection Width (B) Node Degree (d) Diameter

……

TABLE 3.2 Topological Characteristics of a Number of Static Networks

……

• Bandwidth The bandwidth of a network can be defined as the data transfer rate of the network. In a more formal way, the bandwidth is defined as the asymptotic traffic load supported by the network as its utilization approaches unity.

3.3.1 Bandwidth of a Crossbar

We will define the bandwidth for the crossbar as the average number of requests that can be accepted by a crossbar in a given cycle. As processors make requests for memory modules in a crossbar, contention can take place when two or more processors request access to the same memory module. Consider, for example, the case of a crossbar consisting of three processors p1, p2, and p3 and

three memory modules M1, M2, and M3. As processors make requests for accessing memory modules, the following cases may take place:

1. All three processors request access to the same memory module: In this case, only one request can be accepted. Since there are three memory modules, then there are three ways (three accepted requests) in which such a case can arise.

2. All three processors request access to two different memory modules: In this case two requests can be granted. There are 18 ways (36 accepted requests) in which such a case can arise.

3. All three processors request access to three different memory modules: In this case all three requests can be granted. There are six ways (18 accepted requests) in which such a case can arise.

From the above enumeration, it is clear that of the 27 combinations of 3 requests taken from 3 possible requests, there are 57 requests that can be accepted (causing no memory contention). Therefore, we say that the bandwidth of such a crossbar is BW = 57/27 = 2.11. It should be noted that in computing the bandwidth in this simple example, we made a simplified assumption that all processors make requests for memory module access in every cycle.

In general, for M memory modules and n processors, if a processor generates a request with probability p in a cycle

directed to each memory with equal probability, then the expression for the bandwidth can be computed as follows. The probability that a processor requests a particular memory module is p/M. The probability that a processor does not request that memory module during a given cycle is (1 — p/M). The probability that none of the P processors request that memory module during a cycle is (1 — (p/M))n. The probability that at least one request is made to that memory module is (1 — (1 — (p/M))n). Therefore, the expected number of distinct memory modules with at least one request (the bandwidth) is BW = M(1 — (1 — (p/M))n).

Notice that in case there is equal probability that any module be requested by a processor, then the term p/M in the above equation will become 1/M. Now, considering the case M = 3 and n = 3, the BW = 19/9 = 2.11, the same as before.

In deriving the above expression, we have assumed that all processors generate requests for memory modules during a given cycle. A similar expression can be derived for the case whereby only a fraction of processors generate requests during a given cycle (see the exercise at the end of the chapter).

3.3.2 Bandwidth of a Multiple Bus

We will develop an expression for the bandwidth of the general multiple bus arrangement shown in Figure 3.3. It consists of M memory modules, n processors, and B buses. A given bus is dedicated to a particular processor for the

duration of a bus transaction. A processor-memory transfer can use any of the available buses. Given B buses in the system, then up to B requests for memory use can be served simultaneously. In order to resolve possible conflicts in accessing a given memory module out of the available M modules, M arbiters, one for each memory module, are used to arbitrate among the requests made for a given memory module. The set of M arbiters accepts only one request for each memory module at any given time. Let us assume that a processor generates a request with probability p in a cycle directed to each memory with equal probability. Therefore, out of all possible memory requests, only up to M memory requests can be accepted. The probability that a memory module has at least one request is given by (see the crossbar analysis) b = 1 — (1 — (p/M))n. Owing to the availability of only B buses, then of all memory requests, only B request can be satisfied. The

Figure 3.3 A multiple bus system.

probability that exactly k different memory modules are requested during a given cycle can be expressed as ……..

Two cases have to be considered. These are the case where fewer than B different requests being made while fewer than B buses are being used and the case where B or more different requests are made while all B buses are in use. Given these two cases, the bandwidth of the B buses system can be expressed as

……

3.3.3 Bandwidth of a Multistage Interconnection Network (MIN)

In this subsection, we compute the bandwidth of a MIN. A simplifying assumption that we make is that the MIN consists of stages of a x b crossbar switches. One such MIN is the Delta network. This assumption is made such that the results we obtained for the bandwidth of the crossbar network can be utilized.

Let us assume that the request rate at the input of the first stage is given by r0. The number of requests accepted by the first stage and passed on to the next stage is R1 = (1 — (1 — (r0/b))a). The number of requests at any of the b output lines of the first stage is r1 = 1 — (1 — (r0/b))a. Since these requests become the input to the next stage, then by analogy the number of requests at the output of the second stage is given by r2 = 1 — (1 — (r1/b))a. This recursive relation can be extended to compute the number of requests at the output of stage j in terms of the rate of input requests passed on from stage j — 1 as follows: rj = 1 — (1 — (rj—1/b))a for 1 < j < n where n is the number of stages. Based on this, the bandwidth of the MIN is given by BW = bn x rn.

- Latency is defined as the total time required to transmit a message from a source node to a destination node in a parallel architecture machine.

It should be noted that parallel machines attempt to minimize the communication latency by increasing the

interconnectivity. In our discussion, we will show the latency caused by the time spent in switching elements. Latency caused by software overhead, routing delay, and connection delay are overlooked in this discussion.

The latency of a k-ary n-cube is k x log2 N, that of binary hypercube is given by (log2 N), while that of a 2D mesh is given by */N.

• Average distance, da, traveled by a message in a static network, is a measure of the typical number of links (hops) a message has to traverse as it makes its way from any source to any destination in the network. In a network consisting of N nodes, the average distance can be computed using the following relation:

……..

TABLE 3.3 Distance from Node 0000 to all Other Nodes

……..

In the above relation Nd is the number of nodes separated by d links and max is the maximum distance necessary to interconnect two nodes in the network. Consider, for example, a 4-cube network. The average distance between two nodes in such a network can be computed as follows. We compute the distance between node (0) and all other 15 nodes in the cube. These are shown in Table 3.3. From these, therefore, the average distance for a 4-cube is (32/15) ffi 2.13.

• Complexity (Cost) of a static network can be measured in terms of the number of links needed to realize the topology of the network.

The cost of a k-ary n-cube measure in terms of the number of links is given by n x N, that of a hypercube is given by (n x N)/2, that of a 2D mesh (having N nodes) is given by 2(N — 2), and that of a binary tree is given by (N — 1).

• Interconnectivity of a network is a measure of the existence of alternate paths between each source-destination pair. The importance of network connectivity is that it shows the resistance of the network to node and link failures. Network

TABLE 3.4 Performance Measure for a Number of Dynamic Networks

……..

TABLE 3.5 Performance Measure for a Number of Static Networks

……

connectivity can be represented by the two components: node connectivity and link connectivity.

Consider, for example, the binary tree architecture. The failure of a node, for example, the root node, can lead to the

partitioning of the network into two disjoint halves. Similarly, the failure of a link can lead to the partitioning of the network. We therefore say that the binary tree network has a node connectivity of 1 and a link connectivity of 1.

Based on the above discussion and the information provided in Chapter 2, the following two tables, Tables 3.4 and 3.5, provide overall performance comparison among different dynamic interconnection networks and different static networks, respectively. Having presented a number of performance measures for static and dynamic networks, we now turn our attention to the important issue of parallel architecture scalability.

3.4 SCALABILITY OF PARALLEL ARCHITECTURES

A parallel architecture is said to be scalable if it can be expanded (reduced) to a larger (smaller) system with a linear increase (decrease) in its performance (cost). This general definition indicates the desirability for providing equal chance for scaling up a system for improved performance and for scaling down a system for greater cost- effectiveness and/or affordability. Unless otherwise mentioned, our discussion in this section will assume the scaling up of systems. In this context, scalability is used as a measure of the system's ability to provide increased performance, for example, speed as its size is increased. In other words, scalability is a reflection of the system's ability to efficiently utilize the

increased processing resources. In practice, the scalability of a system can be manifested in a number of forms. These forms include speed, efficiency, size, applications, generation, and heterogeneity.

In terms of speed, a scalable system is capable of increasing its speed in proportion to the increase in the number of processors. Consider, for example, the case of adding m numbers on a 4-cube (n = 16 processors) parallel system. Assume for simplicity that m is a multiple of n. Assume also that originally each processor has (m/n) numbers stored in its local memory. The addition can then proceed as follows. First, each processor can add its own numbers sequentially in (m/n) steps. The addition operation is performed simultaneously in all processors. Secondly, each pair of neighboring processors can communicate their results to one of them whereby the communicated result is added to the local result. The second step can be repeated (log2 n) times, until the final result of the addition process is stored in one of the processors. Assuming that each computation and the communication takes one unit time then the time needed to perform the addition of these m numbers is Tp = (m/n) + 2 x log2 n. Recall that the time required to perform the same operation on a single processor is Ts = m.

Therefore, the speedup is given by

……

[redacted]

Table 3.6 provides the speedup S for different values of m and n. It is interesting to notice from the table that for the same number of processors, n, a larger instance of the same problem, m, results in an increase in the speedup, S. This is a property of a scalable parallel system.

[redacted]

In terms of efficiency, a parallel system is said to be scalable if its efficiency can be kept fixed as the number of processors is increased, provided that the problem size is also increased. Consider, for example, the above problem of adding m numbers on an n-cube. The efficiency of such a system is defined as the ratio between the actual speedup, S, and the ideal speedup, n. Therefore, $j = (S/n) = m/(m + 2n \times \log_2 n)$. Table 3.7 shows the values of the efficiency, j, for different values of m and n. The values in the table indicate that for the same number of processors, n, higher efficiency is achieved as the size of the problem, m, is increased. However, as the number of processors, n, increases, the efficiency continues to decrease. Given these two observations, it should be possible to keep the efficiency fixed by increasing simultaneously both the size of the problem, m, and the number of

[redacted]

processors, n. This is a property of a scalable parallel system.

It should be noted that the degree of scalability of a parallel system is determined by the rate at which the problem size must increase with respect to n in order to maintain a fixed efficiency as the number of processors increases. For example, in a highly scalable parallel system the size of the problem needs to grow linearly

TABLE 3.6 The Possible Speedup for Different m and n

……

TABLE 3.7 Efficiency for Different Values of m and n

……

with respect to n to maintain a fixed efficiency. However, in a poorly scalable system, the size of the problem needs to grow exponentially with respect to n to maintain a fixed efficiency.

Recall that the time spent by each processor in performing parallel execution in solving the problem of adding m numbers on an n-cube is given by (m/n) + 2 x log2 n. Of this time, approximately (m/n) is spent performing the actual execution, while the remaining portion of the time, Toh, is an overhead

incurred in performing tasks such as interprocessor communication. The following relationship applies: Toh = n x Tp — Ts. For example, the overall overhead for the addition problem considered above is given by Toh = 2n x log2 n. It is interesting to note that a sequential algorithm running on a single processor does not suffer from such overhead. Now, we can rewrite the expression for the efficiency as j = m/(m + Toh), which leads to the equation m = Z/(1 — Z)Toh. Consider again the problem of adding m numbers using an n-cube. For this problem the problem size m = 2 x Z/(1 — Z) x n x log2 n = Kn x log2 n = Q(n x log2 n). The rate at which the problem size, m, is required to grow with respect to the number of processors, n, to keep the efficiency, j, fixed is called the isoefficiency of a parallel system and can be used as a measure of the scalability of the system. A highly scalable parallel system has a small isoefficiency, while a poor parallel system has a large isoefficiency. Theoretically speaking, a parallel system is considered scalable if its isoefficiency function exists; otherwise the system is considered not scalable. Recall that Gustafson has shown that by scaling up the problem size, m, it is possible to obtain near-linear speedup on as many as 1024 processors (see Section 3.2).

Having discussed the issues of speedup and efficiency of scalable parallel systems, we now conduct a discussion on their relationship. It is useful to indicate at the outset that typically an increase in the speedup of a parallel system (benefit), due to an increase in the number of processors, comes at the expense of a decrease in the efficiency (cost). In order to study the actual behavior of speedup and efficiency, we need first to introduce a new parameter, called the average parallelism (Q). It is defined as the average number of processors that are busy during the execution of given parallel software (program), provided that an unbounded number of processors are available. The average parallelism can equivalently be defined as the speedup achieved assuming the availability of an unbounded number of processors. A number of other equivalent definitions exist for the average parallelism. It has been shown that once Q is determined, then the following bounds are attainable for the speedup and the efficiency on an n-processor system:

……..

The above two bounds show that the sum of the attained fraction of the maximum possible speedup, S(n)/Q, and attained efficiency, must always exceed 1. Notice

also that, given a certain average parallelism, Q, the efficiency (cost) incurred to achieve a given speedup is given by Z(n) > (Q — S(n))/(Q — 1). It is therefore fair to say that the average parallelism of a parallel system, Q, determines the associated speedup versus efficiency tradeoff.	[redacted]
In addition to the above scalability metrics, there has been a number of other unconventional metrics used by some researchers. A number of these are explained below.	[redacted]
Size scalability measures the maximum number of processors a system can accommodate. For example, the size scalability of the IBM SP2 is 512, while that of the symmetric multiprocessor (SMP) is 64.	[redacted]
Application scalability refers to the ability of running application software with improved performance on a scaled-up version of the system. Consider, for example, an n-processor system used as a database server, which can handle 10,000 transactions per second. This system is said to possess application scalability if the number of transactions can be increased to 20,000 using double the number of processors.	[redacted]

Generation scalability refers to the ability of a system to scale up by using next-generation (fast) components. The most obvious example for generation scalability is the IBM PCs. A user can upgrade his/her system (hardware or software) while being able to run their code generated on their existing system without change on the upgraded one.

Heterogeneous scalability refers to the ability of a system to scale up by using hardware and software components supplied by different vendors. For example, under the IBM Parallel Operating Environment (POE) a parallel program can run without change on any network of RS6000 nodes; each can be a low-end PowerPC or a high-end SP2 node.

In his vision on the scalability of parallel systems, Gordon Bell has indicated that in order for a parallel system to survive, it has to satisfy five requirements. These are size scalability, generation scalability, space scalability, compatibility, and competitiveness. As can be seen, three of these long-term survivability requirements have to do with different forms of scalability.

As can be seen from the above introduction, scalability, regardless of its form, is a desirable feature of any parallel

system. This is because it guarantees that with sufficient parallelism in a program, the performance, for example, speedup, can be improved by including additional hardware resources without requiring program change. Owing to its importance, there has been an evolving design trend, called design for scalability (DFS), which promotes the use of scalability as a major design objective. Two different approaches have evolved as DFS. These are overdesign and backward compatibility. Using the first approach, systems are designed with additional features in anticipation for future system scale-up. An illustrative example for such approach is the design of modern processors with 64-bit address, that is, 264 bytes address space. It should be noted that the current UNIX operating system supports only 32-bit address space. With memory space overdesign, future transition to 64-bit UNIX can be performed with minimum system changes. The other form of DFS is the backward compatibility. This approach considers the requirements for scaled-down systems. Backward compatibility allows scaled-up components (hardware or software) to be usable with both the original and the scaled-down systems. As an example, a new processor should be able to execute code generated by old processors. Similarly, a new version of an operating system should preserve all useful functionality of its predecessor such that application software that runs under the old version must be able to run on the new version.

Having introduced a number of scalability metrics for parallel systems, we now turn our attention to the important issue of benchmark performance measurement.

3.5 BENCHMARK PERFORMANCE

Benchmark performance refers to the use of a set of integer and floating-point programs (known collectively as a benchmark) that are designed to test different performance aspects of the computing system(s) under test. Benchmark programs should be designed to provide fair and effective comparisons among high- performance computing systems. For a benchmark to be meaningful, it should evaluate faithfully the performance for the intended use of the system. Whenever advertising for their new computer systems, companies usually quote the benchmark ratings of their systems as a trusted measure. These ratings are usually used for performance comparison purposes among different competing systems.

Among the first known examples of benchmarks are the Dhrystone and Whetstone benchmarks. These are

synthetic (not real) benchmarks intended to measure performance of real machines. The Dhrystone benchmark addresses integer performance. It consists of 100 statements and does not use floating-point operations or data. The rate obtained from Dhrystone is used to compute the MIPS index as a performance measure. This makes the Dhrystone rather unreliable as a source for performance measure. The Whetstone, on the other hand, is a kernel program that addresses floating-point performance for arithmetic operations, array indexing, conditional branch, and subroutine calls. The execution speed obtained using Whetstone is used solely to determine the system performance. This leads to a single figure measure for performance, which makes it unreliable. Synthetic benchmarks were superseded by a number of application software segments that reflect real engineering and scientific applications. These include PERFECT (Performance Evaluation for Cost-Effective Transformations), TPC

TABLE 3.8 SPEC Integer Programs

……

measure for database I/O performance and SPEC (Standard Performance Evaluation Corporation) measure.

The SPEC is a nonprofit corporation formed to "establish, maintain, and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers" . The first SPEC benchmark suite was released in 1989 (SPEC89). It consisted of ten engineering/scientific programs. Two measures were derived from SPEC89. The SPECmark measures the ten programs' execution rates and SPECthruput, which examines the system's throughput. Owing to its unsatisfactory results, SPEC89 was replaced by SPEC92 in 1992.

The SPEC92 consists of two suites: CINT92, which consists of six integer intensive C programs (see Table 3.8), and CFP92, which consists of 14 floating-point intensive C and Fortran programs (see Table 3.9).

In SPEC92, the measure SPECratio represents the ratio of the actual execution time to the predetermined reference time for a given program. In addition, SPEC92 uses the measure SPECint92 as the geometric mean of the SPECratio for the programs in CINT92. Similarly, the measure SPECfp92 is the geometric mean of the SPECratio for the programs in CFP92. In using SPEC for performance measures, three major steps have to be taken: building the tools, preparing auxiliary files, and running the benchmark suites. The tools are used to compile, run, and evaluate the benchmarks. Compilation information

such as the optimization flags and references to alternate source code is kept in what is called makefile wrappers and configuration files. The tools and the auxiliary files are then used to compile and execute the code and compute the SPEC metrics.

The use of the geometric mean to obtain the average time ratio for all programs in the SPEC92 has been subject to a number of criticisms. The premise for these criticisms is that the geometric mean is bound to cause distortion in the obtained results. For example, Table 3.10 shows the execution times (in seconds) obtained using the 14 floating-point programs in SPEC92 for two systems: Silicon Graphics' Challenger XL/Onyx and the Sun Sparc Center with eight CPUs.

As can be observed from Table 3.10 the SG XL/Onyx runs the SPEC92 benchmarks 13.8% (1772.1 — 1557.3/1557.3) faster than the Sun Sparc. However, the Sun Sparc is ranked as 12.5% (109.2 — 97.1/97.1) higher on the SPECrate using the geometric mean. It is such a drawback that causes skepticism among computer architects for the use of the geometric mean in SPEC92. This is because a large improvement of only one program can boost the geometric mean significantly. It was because of this observation that Giladi and Ahituv have suggested that the geometric mean be replaced by the harmonic mean.

TABLE 3.10 SPEC92 Execution Time (in Seconds) for Two Systems

……

Subsequently, there arose a concern about the sensitivity of SPEC metrics to compiler flags. For example, Mirghafori and others have computed the average improvement of SPECpeak with respect to PSECbase for CINT92 and CFP92 on a number of platforms. Recall that PSECpeaks are those ratings that are reported by vendors in their advertisement of new products. The SPECbase is a new measurement to the SPEC92, which has been designed to accurately reflect the typical usage of compiler technology (introduced by PSEC in 1994).

The Mirghafori study revealed that compiler flag tunings have brought about 11% increase in the SPEC ratings. In addition, it has been reported that a number of tuning parameters are usually used by vendors in obtaining their reported SPECpeak and SPECbase ratings and that reproducibility of those ratings is sometimes impossible. To show the discrepancy between the reported SPECbase and SPECpeak performance by a number of vendors, Table 3.11 shows a

sample of eight CFP92 results reported in the SPEC newsletter (the June and September 1994 issues). As can be seen from the table, while some machines show superior performance to other machines based on the reported SPECbase, they show inferior performance using the SPECpeak, and vice versa.

For the abovementioned observations, it became apparent to a number of computer architects that SPEC92 does not predict faithfully the performance of computers on random software for a typical user.

In October 1995, SPEC announced the release of the SPEC95 suite, which replaced the SPEC92 suite fully in September 1996. SPEC95 consists of two CPU-intensive applications: CINT95, a set of eight integer programs and CFP95, a set of 10 floating-point programs. According to SPEC, all SPEC95 performance results published consider the SUN SPARC station 10/40 as the reference machine. Performance results are therefore shown as ratios compared to that machine. Each metric used by SPEC95 is the aggregate overall benchmark of a given suite by taking the geometric mean of the ratios of the individual benchmarks. In presenting the performance results, SPEC takes the speed metrics to measure the ratios to exe-cute a single copy of the benchmark,

while the throughput metrics measure the ratios to execute multiple copies of the benchmark. For example, the SPEC95 performance results of a Digital AlphaStation 500, which uses a 500 MHz Alpha 21164 processor with 8 MB cache and 128 MB memory, are shown in the Table 3.12. In this table,

TABLE 3.11 Five Misleading Reported CFP92

……

the SPECint_rate_base95 is obtained by taking the geometric mean of the rates of the eight benchmarks of the CIT95, where each benchmark is compiled with a low optimization. The rate of each benchmark is measured by running multiple copies of the benchmark for a week, and normalizing the execution time with respect to the SUN SPARCstation 10/40. Therefore, the number 113 means that the AlphaStation executes 112 times more copies of the CINT95 than the SUN in a week.

The SPECfp is obtained by taking the geometric mean of the ratios of the ten benchmarks of the CFP95, where each benchmark is compiled with aggressive optimization. The rate of each benchmark is measured by running a single copy of the benchmark for a week, and normalizing the execution time with respect to the SUN SPARCstation 10/40. Therefore, the number 20.4 means that

the Alpha- Station is 19.4 times faster than the SUN in executing a single copy of the CFP95.

On June 30, 2000, SPEC retired the SPEC95 and replaced it with SPEC CPU2000. The new benchmark suite consists of 26 benchmarks in total (12 integer and 14 floating-point benchmarks). It has 19 applications that have never been in a SPEC CPU suite. The CPU2000 integer and floating-point benchmark suites are shown in Tables 3.13 and 3.14, respectively. Three subjective criteria are achieved

TABLE 3.13 The CPU2000 Integer Benchmark Suite

……..

TABLE 3.14 The CPU2000 Floating-Point Benchmark Suite

……..

in the CPU2000. These are confidence in the benchmark maintainability, transparency, and vendor interest.

Performance results of the 26 CPU2000 benchmarks (both integer and floatingpoint) were reported for three different configured systems using the Alpha 21164 chip. These systems are the AlphaStation 500/500 (System #1), the Personal Workstation 500au (System #2), and the AlphaServer 4100 5/533 (System

#3). The performance is stated relative to a reference machine, a 300 MHz Sun Ultra5_10, which gets a score of 100. It was reported that the performance of the 26 benchmarks on the 21164 systems ranges from 92.3 (for the 172.mgrid) to 331 (for the 179.art). It was also found that the 500 MHz Systems # and System #2 differ by more than 5% on 17 of the 26 benchmarks. The 533 MHz (system #3), with a 7% megahertz advantage, wins by more than 10% three times (176.gcc, 253.perlbmk, 199.art), by less than 3% three times (197.parser, 253.eon, 256.bzip2), and loses to the 500 MHz three times (181.mcf, 172.mgrid, 188.ammp). …… …… Shared Memory Architecture	[redacted]

Shared Memory Architecture

Shared memory systems form a major category of multiprocessors. In this category, all processors share a global memory. Communication between tasks running on different processors is performed through writing to and reading from the global memory. All interprocessor coordination and synchronization is also accomplished via the global memory. A shared memory computer system consists of a set of independent processors, a set of memory modules, and an interconnection network as shown in Figure 4.1.

Two main problems need to be addressed when designing a shared memory system: performance degradation due to contention, and coherence problems. Performance degradation might happen when multiple processors are trying to access the shared memory simultaneously. A typical design might use caches to solve the contention problem. However, having multiple copies of data, spread throughout the caches, might lead to a coherence problem. The copies in the caches are coherent if they are all equal to the same value. However, if one of the processors writes over the value of one of the copies, then the copy becomes inconsistent because it no longer equals the value of the other copies. In this chapter we study

a variety of shared memory systems and their solutions of the cache coherence problem.

……
…..

Figure 4.1 Shared memory systems.

4.1 CLASSIFICATION OF SHARED MEMORY SYSTEMS

The simplest shared memory system consists of one memory module (M) that can be accessed from two processors (P1 and P2); see Figure 4.2. Requests arrive at the memory module through its two ports. An arbitration unit within the memory module passes requests through to a memory controller. If the memory module is not busy and a single request arrives, then the arbitration unit passes that request to the memory controller and the request is satisfied. The module is placed in the busy state while a request is being serviced. If a new request arrives while the memory is busy servicing a previous request, the memory module sends a wait signal, through the memory controller, to the processor making the new request. In response, the requesting processor may hold its request on the line until the memory becomes free or it may repeat its request some time later.

If the arbitration unit receives two requests, it selects one of them and passes it to the memory controller. Again, the denied request can be either held to be served next or it may be repeated some time later. Based on the interconnection network used, shared memory systems can be categorized in the following categories.

4.1.1 Uniform Memory Access (UMA)

In the UMA system a shared memory is accessible by all processors through an interconnection network in the same way a single processor accesses its memory. All processors have equal access time to any memory location. The interconnection network used in the UMA can be a single bus, multiple buses, or a crossbar switch. Because access to shared memory is balanced, these systems are also called SMP (symmetric multiprocessor) systems. Each processor has equal opportunity to read/write to memory, including equal access speed. Commercial examples of SMPs are Sun Microsystems multiprocessor servers and Silicon Graphics Inc. multiprocessor servers. A typical bus-structured SMP computer, as shown in Figure 4.3, attempts to reduce contention for the bus by fetching instructions and data directly from each individual cache, as much as possible. In the extreme, the bus contention might be reduced to zero after the cache memories are loaded from the global memory, because it is possible for all instructions and data to be completely

contained within the cache. This memory organization is the most popular among
……

Figure 4.2 Shared memory via two ports.

Figure 4.3 Bus-based UMA (SMP) shared memory system.

shared memory systems. Examples of this architecture are Sun Starfire servers, HP V series, and Compaq AlphaServer GS.

4.1.2 Nonuniform Memory Access (NUMA)

In the NUMA system, each processor has part of the shared memory attached. The memory has a single address space. Therefore, any processor could access any memory location directly using its real address. However, the access time to modules depends on the distance to the processor. This results in a nonuniform memory access time. A number of architectures are used to interconnect processors to memory modules in a NUMA. Among these are the tree and the hierarchical bus networks. Examples of NUMA architecture are BBN TC-2000, SGI Origin 3000, and Cray T3E. Figure 4.4 shows the NUMA system organization.
…….

Figure 4.4 NUMA shared memory system.

4.1.3 Cache-Only Memory Architecture (COMA)

Similar to the NUMA, each processor has part of the shared memory in the COMA.

However, in this case the shared memory consists of cache memory. A COMA system requires that data be migrated to the processor requesting it. There is no memory hierarchy and the address space is made of all the caches. There is a cache directory (D) that helps in remote cache access. The Kendall Square Research's KSR-1 machine is an example of such architecture. Figure 4.5 shows the organization of COMA.

4.2 BUS-BASED SYMMETRIC MULTIPROCESSORS

Shared memory systems can be designed using bus-based or switch-based interconnection networks. The simplest network for shared memory systems is the bus. The bus/cache architecture alleviates the need for expensive multiported memories and interface circuitry as well as the need to adopt a message-passing paradigm when developing application software. However, the bus may get saturated if multiple processors are trying to access the shared memory (via the bus)

simultaneously. A typical bus-based design uses caches to solve the bus contention problem. Highspeed caches connected to each processor on one side and the bus on the other side mean that local copies of instructions and data can be supplied at the highest possible rate.

If the local processor finds all of its instructions and data in the local cache, we say the hit rate is 100%. The miss rate of a cache is the fraction of the references that cannot be satisfied by the cache, and so must be copied from the global memory, across the bus, into the cache, and then passed on to the local processor. One of the goals of the cache is to maintain a high hit rate, or low miss rate under high processor loads. A high hit rate means the processors are not using the bus as much. Hit rates are determined by a number of factors, ranging from the application programs being run to the manner in which cache hardware is implemented.

Miss rate (Tỉ lệ phần trăm truy cập bộ nhớ không được tìm thấy trong 1 cấp độ bộ nhớ)

A processor goes through a duty cycle, where it executes instructions a certain number of times per clock cycle. Typically, individual processors execute less than one instruction per cycle, thus reducing the number of times it needs to access memory. Subscalar processors execute less than one instruction per cycle, and superscalar processors execute more than one instruction per cycle. In

any case, we want to minimize the number of times each local processor tries to use the central bus. Otherwise, processor speed will be limited by bus bandwidth.

We define the variables for hit rate, number of processors, processor speed, bus speed, and processor duty cycle rates as follows:

- N = number of processors;
- h = hit rate of each cache, assumed to be the same for all caches;
- (1 — h) = miss rate of all caches;
- B = bandwidth of the bus, measured in cycles/second;
- I = processor duty cycle, assumed to be identical for all processors, in fetches/ cycle; and
- V = peak processor speed, in fetches/second.

The effective bandwidth of the bus is BI fetches/second. If each processor is running at a speed of V, then misses are being generated at a rate of V(1 — h). For an N-processor system, misses are simultaneously being generated at a rate of N(1 — h) V. This leads to saturation of the bus when N processors simultaneously try to access the bus. That is, N(1 — h)V < BI. The maximum

number of processors with cache memories that the bus can support is given by the relation,

……..

Example 1 Suppose a shared memory system is constructed from processors that can execute V = 107 instructions/s and the processor duty cycle I = 1. The caches are designed to support a hit rate of 97%, and the bus supports a peak bandwidth of B = 106 cycles/s. Then, (1 — h) = 0.03, and the maximum number of processors N is N < 106/(0.03 * 107) = 3.33. Thus, the system we have in mind can support only three processors!

We might ask what hit rate is needed to support a 30-processor system. In this case, h = 1 — BI/NV = 1 — (106(1))/((30)(107)) = 1 — 1/300, so for the system we have in mind, h = 0.9967. Increasing h by 2.8% results in supporting a factor of ten more processors.

4.3 BASIC CACHE COHERENCY METHODS

Multiple copies of data, spread throughout the caches, lead to a coherence problem among the caches. The copies in the caches are coherent if they all equal the same value. However, if one of the processors writes over the value of one of the copies, then the copy becomes inconsistent because it no longer equals the value of the other copies.

If data are allowed to become inconsistent (incoherent), incorrect results will be propagated through the system, leading to incorrect final results. Cache coherence algorithms are needed to maintain a level of consistency throughout the parallel system.

4.3.1 Cache-Memory Coherence

In a single cache system, coherence between memory and the cache is maintained using one of two policies: (1) write-through, and (2) write-back. When a task running on a processor P requests the data in memory location X, for example, the contents of X are copied to the cache, where it is passed on to P. When P updates the value of X in the cache, the other copy in memory also needs to be updated in order to maintain consistency. In write-through, the memory is updated every time the cache is updated, while in write-back, the memory is updated only when the block in the cache is being replaced. Table 4.1 shows the write-through versus write-back policies.

4.3.2 Cache-Cache Coherence

In multiprocessing system, when a task running on processor P requests the data in global memory location X, for example, the contents of X are copied to processor P's local cache, where it is

passed on to P. Now, suppose processor Q also accesses X. What happens if Q wants to write a new value over the old value of X?

There are two fundamental cache coherence policies: (1) write-invalidate, and (2) write-update. Write-invalidate maintains consistency by reading from local caches until a write occurs. When any processor updates the value of X through a write, posting a dirty bit for X invalidates all other copies.

For example, processor Q invalidates all other copies of X when it writes a new value into its cache. This sets the dirty bit for X. Q can continue to change X without further notifications to other caches because Q has the only valid copy of X. However, when processor P wants to read X, it must wait until X is updated and the dirty bit is cleared. Write-update maintains consistency by immediately updating all copies in all caches. All dirty bits are set during each write operation. After all copies have been updated, all

TABLE 4.1 Write-Through vs. Write-Back

……..

dirty bits are cleared. Table 4.2 shows the write-update versus write-invalidate

policies.

4.3.3 Shared Memory System Coherence

The four combinations to maintain coherence among all caches and global memory are:

- Write-update and write-through;
- Write-update and write-back;
- Write-invalidate and write-through; and
- Write-invalidate and write-back.

If we permit a write-update and write-through directly on global memory location X, the bus would start to get busy and ultimately all processors would be idle while waiting for writes to complete. In write-update and write-back, only copies in all caches are updated. On the contrary, if the write is limited to the copy of X in cache Q, the caches become inconsistent on X. Setting the dirty bit prevents the spread of inconsistent values of X, but at some point, the inconsistent copies must be updated.

4.4 SNOOPING PROTOCOLS

Snooping protocols are based on watching bus activities and carry out the appropriate coherency commands when necessary. Global memory is moved in blocks, and each block has a state associated with it, which determines what happens to the entire contents of the block. The state of a block might change as a result of the operations Read-Miss,

Read-Hit, Write-Miss, and Write-Hit. A cache miss means that the requested block is not in the cache or it is in the cache but has been invalidated. Snooping protocols differ in whether they update or invalidate shared copies in remote caches in case of a write operation. They also differ as to where to obtain the new data in the case of a cache miss. In what follows we go over some examples of snooping protocols that maintain cache coherence.

TABLE 4.3 Write-Invalidate Write-Through Protocol

……

Read-Hit Use the local copy from the cache.

Read-Miss Fetch a copy from global memory. Set the state of this copy to Valid.

Write-Hit Perform the write locally. Broadcast an Invalid command to all caches. Update the global memory.

Write-Miss Get a copy from global memory. Broadcast an invalid command to all caches. Update the global memory. Update the local copy and set its state to Valid.

Block replacement Since memory is always consistent, no write-back is needed when a block is replaced.

4.4.1 Write-Invalidate and Write-Through

In this simple protocol the memory is always consistent with the most recently updated cache copy. Multiple processors can read block copies from main memory safely until one processor updates its copy. At this time, all cache copies are invalidated and the memory is updated to remain consistent. The block states and protocol are summarized in Table 4.3.

Example 2 Consider a bus-based shared memory with two processors P and Q as shown in Figure 4.6. Let us see how the cache coherence is maintained using Write- Invalidate Write-Through protocol. Assume that that X in memory was originally set to 5 and the following operations were performed in the order given: (1) P reads X; (2) Q reads X; (3) Q updates X; (4) Q reads X; (5) Q updates X; (6) P updates X; (7) Q reads X. Table 4.4 shows the contents of memory and the

Figure 4.6 A bus-based shared memory system with two processors P and Q.

…….

two caches after the execution of each

operation when Write-Invalidate Write-Through was used for cache coherence. The table also shows the state of the block containing X in P's cache and Q's cache.

4.4.2 Write-Invalidate and Write-Back (Ownership Protocol)

In this protocol a valid block can be owned by memory and shared in multiple caches that can contain only the shared copies of the block. Multiple processors can safely read these blocks from their caches until one processor updates its copy. At this time, the writer becomes the only owner of the valid block and all other copies are invalidated. The block states and protocol are summarized in Table 4.5.

Example 3 Consider the shared memory system of Figure 4.6 and the following operations: (1) P reads X; (2) Q reads X; (3) Q updates X; (4) Q reads X; (5) Q updates X; (6) P updates X; (7) Q reads X. Table 4.6 shows the contents of memory and the two caches after the execution of each operation when Write-Invalidate Write-Back was used for cache coherence. The table also shows the state of the block containing X in P's cache and Q's cache.

4.4.3 Write-Once

This write-invalidate protocol, which was proposed by Goodman in 1983, uses a combination of write-through and write-back. Write-through is used the very first	[redacted]
TABLE 4.5 Write-Invalidate Write-Back Protocol …….	[redacted]
Read-Miss If no Exclusive (Read-Write) copy exists, then supply a copy from global memory. Set the state of this copy to Shared (Read-Only). If an Exclusive (Read-Write) copy exists, make a copy from the cache that set the state to Exclusive (Read-Write), update global memory and local cache with the copy. Set the state to Shared (ReadOnly) in both caches.	[redacted]
Write-Hit If the copy is Exclusive (Read-Write), perform the write locally. If the state is Shared (Read-Only), then broadcast an Invalid to all caches. Set the state to Exclusive (Read-Write).	[redacted]
Write-Miss Get a copy from either a cache with an Exclusive (Read- Write) copy, or from global memory itself. Broadcast an Invalid command to all caches. Update the local copy and set its state to Exclusive (Read-Write).	[redacted]

Block replacement If a copy is in an Exclusive (Read-Write) state, it has to be written back to main memory if the block is being replaced. If the copy is in Invalid or Shared (Read-Only) states, no write-back is needed when a block is replaced.

time a block is written. Subsequent writes are performed using write-back. The block states and protocol are summarized in Table 4.7.

Example 4 Consider the shared memory system of Figure 4.6 and the following operations: (1) P reads X; (2) Q reads X; (3) Q updates X; (4) Q reads X; (5) Q updates X; (6) P updates X; (7) Q reads X. Table 4.8 shows the contents of memory and the two caches after the execution of each operation when Write-Once was used for cache coherence. The table also shows the state of the block containing X in P's cache and Q's cache.

4.4.4 Write-Update and Partial Write-Through

In this protocol an update to one cache is written to memory at the same time it is broadcast to other caches sharing the updated block. These caches snoop on the bus

…….

and perform updates to their local copies. There is also a special bus line, which is asserted to indicate that at least one other cache is sharing the block. The block states and protocol are summarized in Table 4.9.

Example 5 Consider the shared memory system of Figure 4.6 and the following operations: (1) P reads X; (2) P updates X; (3) Q reads X; (4) Q updates X; (5) Q reads X; (6) Block X is replaced in P's cache; (7) Q updates X; (8) P updates X. Table 4.10 shows the contents of memory and the two caches after the execution of each operation when Write-Update Partial Write-Through was used for cache coherence. The table also shows the state of the block containing X in P's cache and Q's cache.

4.4.5 Write-Update and Write-Back

This protocol is similar to the previous one except that instead of writing through to the memory whenever a shared block is updated, memory updates are done only when the block is being replaced. The block states and protocol are summarized in Table 4.11.

Example 6 Consider the shared memory system of Figure 4.6 and the following operations: (1) P reads X; (2) P updates X; (3) Q reads X; (4) Q updates X; (5) Q reads X; (6) Block X is replaced in Q's cache; (7) P updates X; (8) Q updates X. Table 4.12 shows the contents of memory

and the two caches after the execution

TABLE 4.7 Write-Once Protocol

State	Description
Invalid [INV]	The copy is inconsistent.
Valid [VALID]	The copy is consistent with global memory.
Reserved [RES]	Data have been written exactly once and the copy is consistent with global memory. There is only one copy of the global memory block in one local cache.
Dirty [DIRTY]	Data have been updated more than once and there is only one copy in one local cache. When a copy is dirty, it must be written back to global memory.
Event	Actions
Read-Hit	Use the local copy from the cache.
Read-Miss	If no Dirty copy exists, then supply a copy from global memory. Set the state of this copy to Valid. If a dirty copy exists, make a copy from the cache that set the state to Dirty, update global memory and local cache with the copy. Set the state to VALID in both caches.

Write-Hit If the copy is Dirty or Reserved, perform the write locally, and set the state to Dirty. If the state is Valid, then broadcast an Invalid command to all caches. Update the global memory and set the state to Reserved.	[redacted]
Write-Miss Get a copy from either a cache with a Dirty copy or from global memory itself. Broadcast an Invalid command to all caches. Update the local copy and set its state to Dirty.	[redacted]
Block replacement If a copy is in a Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid, Reserved, or Invalid states, no write-back is needed when a block is replaced.	[redacted]
TABLE 4.8 Example 4 (Write-Once Protocol)	[redacted]
State Description	[redacted]
Use the local copy from the cache. State does not change.	[redacted]
If no other cache copy exists, then supply a copy from global memory. Set the state of this copy to Valid Exclusive. If a cache copy exists, make a copy from the cache. Set the state to Shared in both caches. If the cache copy was in a Dirty state, the value must also be written to memory.	[redacted]

Perform the write locally and set the state to Dirty. If the state is Shared, then broadcast data to memory and to all caches and set the state to Shared. If other caches no longer share the block, the state changes from Shared to Valid Exclusive.	
The block copy comes from either another cache or from global memory. If the block comes from another cache, perform the update and update all other caches that share the block and global memory. Set the state to Shared. If the copy comes from memory, perform the write and set the state to Dirty.	
If a copy is in a Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid Exclusive or Shared states, no write-back is needed when a block is replaced.	
of each operation when Write-Update Write-Back was used for cache coherence. The table also shows the state of the block containing X in P's cache and Q's cache.	
4.5 DIRECTORY BASED PROTOCOLS	

Owing to the nature of some interconnection networks and the size of the shared memory system, updating or invalidating caches using snoopy protocols might become unpractical. For example, when a multistage network is used to build a large shared memory system, the broadcasting techniques used in the snoopy protocols becomes very expensive. In such situations, coherence commands need to be sent to only those caches that might be affected by an update. This is the idea behind directory-based protocols. Cache coherence protocols that somehow store information on where copies of blocks reside are called directory schemes. A directory is a data structure that maintains information on the processors that share a memory block and on its state. The information maintained in the directory could

TABLE 4.10 Example 5 (Write-Update Partial Write-Through)

be either centralized or distributed. A Central directory maintains information about all blocks in a central data structure. While Central directory includes everything in one location, it becomes a bottleneck and suffers from large search time. To alleviate this problem, the same information can be handled in a distributed fashion by allowing each memory module to maintain a separate directory. In a distributed directory, the

entry associated with a memory block has only one pointer one of the cache that requested the block.

4.5.1 Protocol Categorization

A directory entry for each block of data should contain a number of pointers to specify the locations of copies of the block. Each entry might also contain a dirty bit to specify whether or not a unique cache has permission to write this memory block. Most directory-based protocols can be categorized under three categories: full-map directories, limited directories, and chained directories.

Full-Map Directories In a full-map setting, each directory entry contains N pointers, where N is the number of processors. Therefore, there could be N cached copies of a particular block shared by all processors. For every memory block, an N-bit vector is maintained, where N equals the number of processors in

Read-Miss If no other cache copy exists, then supply a copy from global memory. Set the state of this copy to Valid Exclusive. If a cache copy exists, make a copy from the cache. Set the state to Shared Clean. If the supplying cache copy was in a Valid Exclusion or Shared Clean, its new state becomes Shared Clean. If the supplying cache copy was in a Dirty or Shared Dirty state, its new state becomes Shared Dirty.

	[redacted]
Write-Hit If the sate was Valid Exclusive or Dirty, perform the write locally and set the state to Dirty. If the state is Shared Clean or Shared Dirty, perform update and change state to Shared Dirty. Broadcast the updated block to all other caches. These caches snoop the bus and update their copies and set their state to Shared Clean.	[redacted]
Write-Miss The block copy comes from either another cache or from global memory. If the block comes from another cache, perform the update, set the state to Shared Dirty, and broadcast the updated block to all other caches. Other caches snoop the bus, update their copies, and change their state to Shared Clean. If the copy comes from memory, perform the write and set the state to Dirty.	[redacted]
Block replacement If a copy is in a Dirty or Shared Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid Exclusive, no write back is needed when a block is replaced.	[redacted]

the shared memory system. Each bit in the vector corresponds to one processor. If the i * bit is set to one, it means that processor i has a copy of this block in its cache. Figure 4.7 illustrates the fully mapped scheme. In the figure the vector associated with block X in memory indicates that X is in Cache C0 and Cache C2. Clearly the space is not utilized efficiently in this scheme, in particular if not many processors share the same block.

Limited Directories Limited directories have a fixed number of pointers per directory entry regardless of the number of processors. Restricting the number of

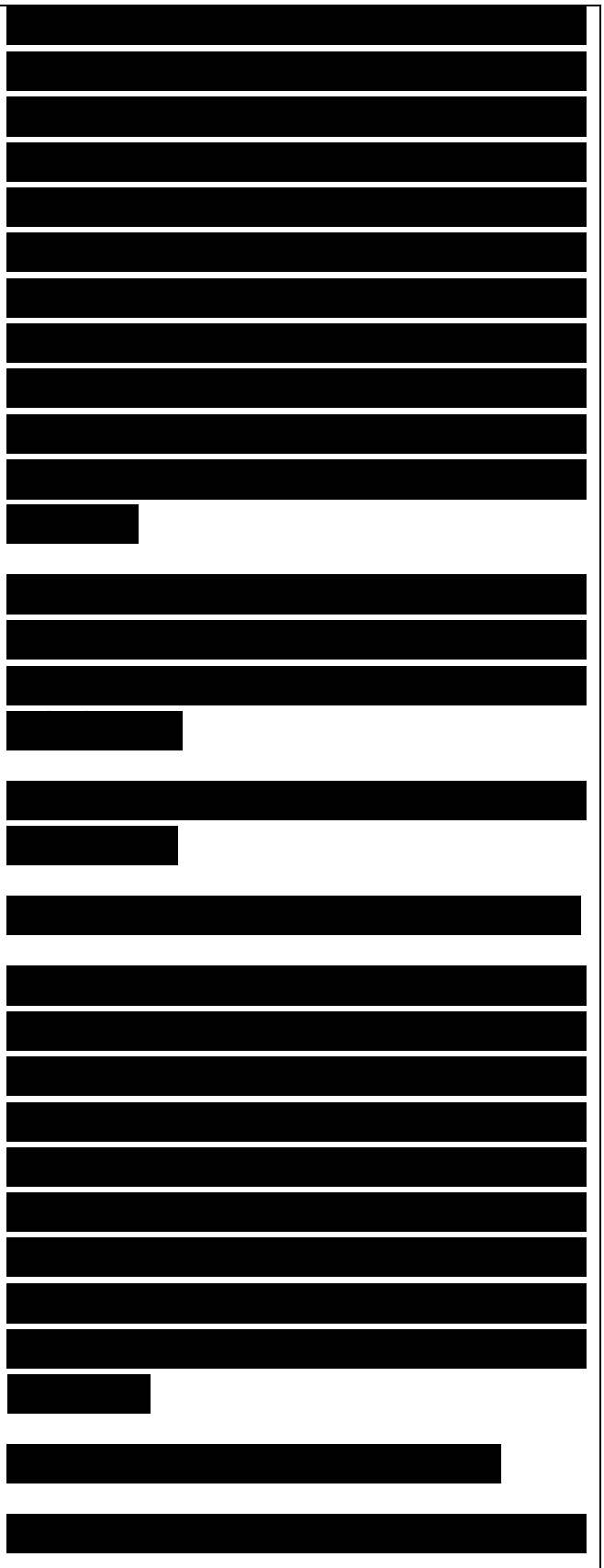
TABLE 4.12 Example 6 (Write-Update Write-Back)

……

simultaneously cached copies of any block should solve the directory size problem that might exist in full-map directories. Figure 4.8 illustrates the limited directory scheme. In this example, the number of copies that can be shared is restricted to two. This is why the vector associated with block X in memory has only two locations. The vector indicates that X is in Cache C0 and Cache C2.

Figure 4.7 Fully mapped directory.

Figure 4.8 Limited directory (maximum sharing = 2).



Chained Directories Chained directories emulate full-map by distributing the directory among the caches. They are designed to solve the directory size problem without restricting the number of shared block copies. Chained directories keep track of shared copies of a particular block by maintaining a chain of directory pointers. Figure 4.9 shows that the directory entry associated with X has a pointer to Cache C2, which in turn has a pointer to Cache C0. That is, block X exists in the two Caches C0 and Cache C2. The pointer from Cache C0 is pointing to terminator (CT), indicating the end of the list.

4.5.2 Invalidate Protocols

Centralized Directory Invalidate When a write request is issued, the central directory is used to determine which processors have a copy of the block.

……..

Invalidating signals and a pointer to the requesting processor are forwarded to all processors that have a copy of the block. Each invalidated cache sends an acknowledgment to the requesting processor. After the invalidation is complete, only the writing processor will have a cache with a copy of the block.

Figure 4.10 shows a write-miss request from Cache C3. Upon receiving the request, the memory sends invalidating signals and a pointer to the Cache C3 to Cache C0 and Cache C2. These caches invalidate themselves and send invalidation acknowl-edgment to Cache C3. After the invalidation is done, Cache C3 will have exclusive read-write access to X.

Scalable Coherent Interface (SCI) The scalable coherent interface (SCI) protocols are based on a doubly linked list of distributed directories. Each cached block is entered into a list of processors sharing that block. For every block address, the memory and cache entries have additional tag bits. Part of the memory tag identifies the first processor in the sharing list (the head). Part of each cache tag identifies the previous and following sharing list entries. Without counting the number of bits needed in the local caches for the pointers, the directory size in memory equals the number of memory blocks times log2 (number of caches).

Initially memory is in the uncached state and cached copies are invalid. A read request is directed from a processor to the memory controller. The requested data is returned to the requester's cache and its entry state is changed from invalid to the head state. This changes the memory state from uncached to cached. When a new

requester directs its read request to memory, the memory returns a pointer to the head. A cache-to-cache read request (called Prepend) is sent from the requester to the head cache. On receiving the request, the head cache sets its backward pointer to point to the requester's cache. The requested data is returned to the requester's cache and its entry state is changed to the head state. The head of the list has the authority to purge other entries in the list to obtain an exclusive (read-write) entry. The initial transaction to the second sharing list entry purges that entry and

……..

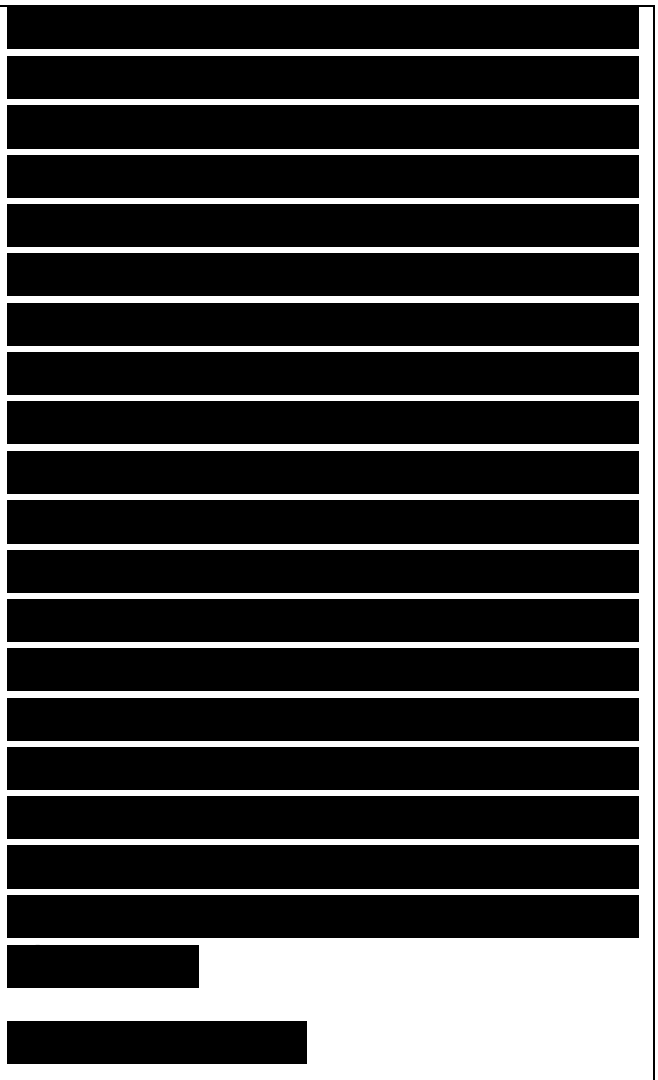


Figure 4.10 Centralized directory invalidation.

returns its forward pointer. The forward pointer is used to purge the next entry and so on. Entries can also delete themselves from the list when they are needed to cache other block addresses. Figure 4.11 shows the sharing list addition and removal operations in SCI.

Stanford Distributed Directory (SDD) The Stanford distributed directory (SDD) protocol is based on a singly linked list of distributed directories. Similar to the SCI protocol, memory points to the head of the sharing list. Each processor points only to its predecessor. The sharing list additions and removals are handled differently from the SCI protocol.

On a read-miss, a new requester sends a read-miss message to memory. The memory updates its head pointers to point to the requester and send a read-miss-forward signal to the old head. On receiving the request, the old head returns the requested data along with its address as a read-miss-reply. When the reply is received, at the requester's cache, the data is copied and the pointer is made to point to the old head.

On a write-miss, a requester sends a write-miss message to memory. The memory updates its head pointers to point to the requester and sends a write-miss-forward signal to the old head. The old head invalidates itself, returns the requested data

……

as a write-miss-reply-data signal, and send a write-miss-forward to the next cache in the list. When the next cache

receives the write-miss-forward signal, it invalidates itself and sends a write-miss-forward to the next cache in the list. When the write-miss-forward signal is received by the tail or by a cache that no longer has a copy of the block, a write-miss-reply is sent to the requester. The write is complete when the requester receives both write-miss-reply-data and write-miss-reply. Figure 4.12 shows the sharing list addition and removal operations in SDD.

4.6 SHARED MEMORY PROGRAMMING

Shared memory parallel programming is perhaps the easiest model to understand because of its similarity with operating systems programming and general multiprogramming. Shared memory programming is done through some extensions to existing programming languages, operating systems, and code libraries. In a shared memory parallel program, there must exist three main programming constructs:

Figure 4.12 Stanford distributed directory (a) sharing list addition (SDD); and (b) write miss sharing list removal (SDD).

4.6.1 Task Creation

At the large-grained level, a shared memory system can provide traditional

timesharing. Each time a new process is initiated, idle processors are supplied to run the new process. If the system is loaded, the processor with least amount of work is assigned the new process. These large-grained processes are often called heavy weight tasks because of their high overhead. A heavy weight task in a multitasking system like UNIX consists of page tables, memory, and file description in addition to program code and data. These tasks are created in UNIX by invocation of fork, exec, and other related UNIX commands. This level is best suited for heterogeneous tasks.

At the fine-grained level, lightweight processes makes parallelism within a single application practical, where it is best suited for homogeneous tasks. At this level, an application is a series of fork-join constructs. This pattern of task creation is called the supervisor-workers model, as shown in Figure 4.13.

4.6.2 Communication

In general, the address space on an executing process has three segments called the text, data, and stack. The text is where the binary code to be executed is stored; the data segment is where the program's data are stored; and the stack is where activation records and dynamic data are stored. The data and stack segments expand and contract as the

program executes. Therefore, a gap is purposely left in between the data and stack segments. Serial processes are assumed to be mutually independent and do not share addresses. The code of each serial process is allowed to access data in its own data and stack segments only. A parallel process is similar to the serial process plus an additional shared data segment. This shared area is allowed to grow and is placed in the gap between private data and stack segments. Figure 4.14 shows the difference between a serial process and a parallel process.

……..

Figure 4.13 Supervisor-workers model used in most parallel applications on shared memory systems.

…….

Communication among parallel processes can be performed by writing to and reading from shared variables in the shared data segments as shown in Figure 4.15.

4.6.3 Synchronization

Synchronization is needed to protect shared variables by ensuring that they are accessed by only one process at a given time (mutual exclusion). They can also be used to coordinate the execution of

parallel processes and synchronize at certain points in execution. There are two main synchronization constructs in shared memory systems: (1) locks and (2) barriers. Figure 4.16a shows three parallel processes using locks to ensure mutual exclusion. Process P2 has to wait until P1 unlocks the critical section; similarly P3 has to wait until P2 issues the unlock statement. In Figure 4.16b, P3 and P1 reach their barrier statement before P2, and

…….

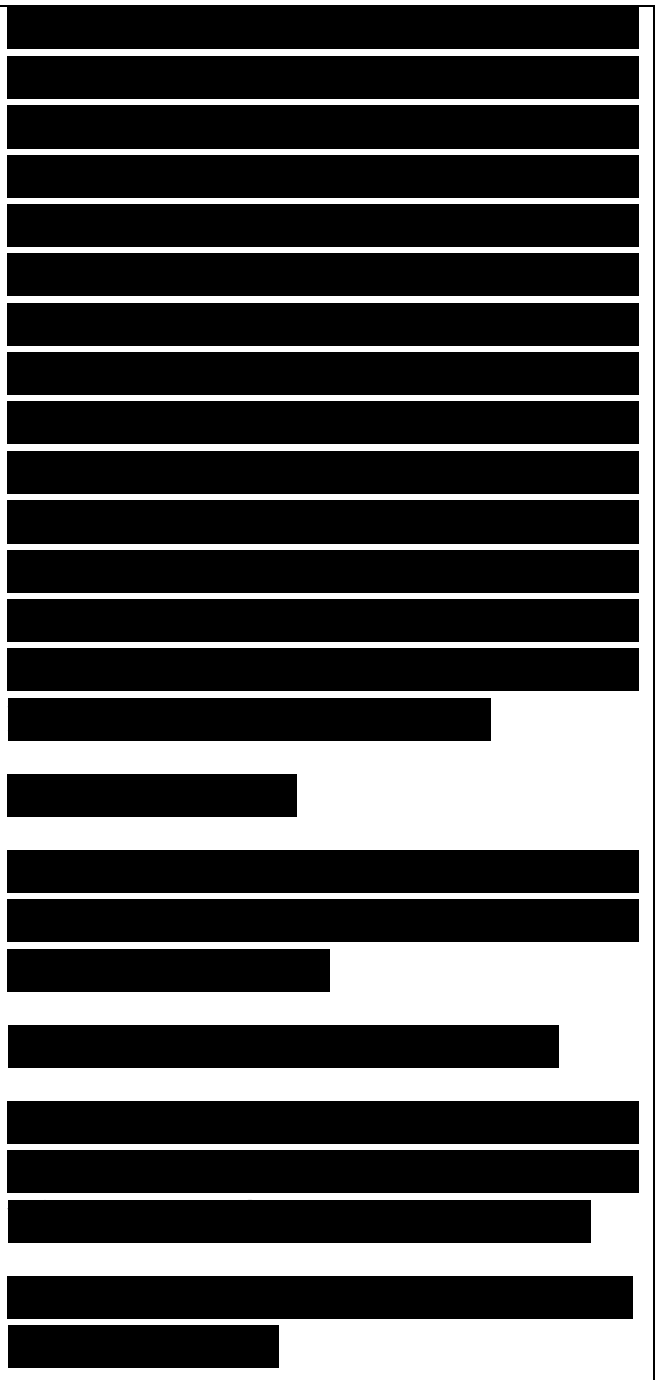
Figure 4.15 Two parallel processes communicate using the shared data segment.

Figure 4.16 Locks and barriers.

they have to wait until P2 reaches its barrier. When all three reach the barrier statement, they all can proceed.

……

……



Parallel Programming in the Parallel Virtual Machine

The Parallel Virtual Machine (PVM) was originally developed at Oak Ridge National Laboratory and the University of Tennessee. It makes it possible to develop applications on a set of heterogeneous computers connected by a network that appears logically to the users as a single parallel computer. The PVM offers a powerful set of process control and dynamic resource management functions. It provides programmers with a library of routines for the initiation and termination of tasks, synchronization, and the alteration of the virtual machine configuration. It also facilitates message passing via a number of simple constructs. Interoperability among different heterogeneous computers is a major advantage in PVM. Programs written for some architecture can be copied to another architecture, compiled and executed without modification. Additionally, these PVM executables can still communicate with each other. A PVM application is made from a number of tasks that cooperate to jointly provide a solution to a single problem. A task may alternate between computation and communication with other tasks. The programming model is a network of communicating sequential tasks in which each task has its own locus of control, and sequential tasks communicate by exchanging messages.

8.1 PVM ENVIRONMENT AND APPLICATION STRUCTURE

The computing environment in PVM is the virtual machine, which is a dynamic set of heterogeneous computer systems connected via a network and managed as a single parallel computer. The computer nodes in the network are called hosts, which could be uniprocessor, multiprocessor systems, or clusters running the PVM software. PVM has two components: a library of PVM routines, and a daemon that should reside on all the hosts in the virtual machine. Before running a PVM application, a user should start up PVM and configure a virtual machine. The PVM console allows the user to interactively start and then alter the virtual machine at any time during system operation. The details of how to set up the PVM software, how to configure a virtual machine, and how to compile and run PVM programs can be found at http://www.epm.ornl.gov/pvm and in Geist et al. (1994).

The PVM application is composed of a number of sequential programs, each of which will correspond to one or more processes in a parallel program. These pro-grams are compiled individually for each host in the virtual machine. The object files are placed in locations

accessible from other hosts. One of these sequential programs, which is called the initiating task, has to be started manually on one of the hosts.

The tasks on the other hosts are activated automatically by the initiating task. The tasks comprising a PVM application can all be identical but work on different ranges of data. This model of parallel programming is called SPMD, which stands for Single Program Multiple Data. Although SPMD is common in most PVM applications, it is still possible to have the tasks perform different functions. A pipeline of parallel tasks that perform input, processing, and output is an example of parallel tasks that are performing different functions.

Parallel virtual machine parallel applications can take different structures. One of the most common structures is the star graph in which the middle node in the star is called the supervisor and the rest of the nodes are workers. The star structure is often referred to as a supervisor-workers or a master-slaves model.

In this model, the supervisor is the initiating task that activates all the workers. A tree structure is another form of a PVM application. The root of the tree is the top supervisor and underneath there

are several levels in the hierarchy. We will use the terms supervisor-workers and hierarchy to refer to the star and the tree structures, respectively.

[redacted]

8.1.1 Supervisor-Workers Structure

[redacted]

There is only one level of hierarchy in this structure: one supervisor and many workers. The supervisor serves as the initiating task that is activated manually on one of the hosts. The supervisor, which is also called the master, has a number of special responsibilities. It normally interacts with the user, activates the workers on the virtual machine, assigns work to the workers, and collects results from the workers.

[redacted]

The workers, which are also called slaves, are activated by the supervisor to perform calculations. The workers may or may not be independent. If they are not independent, they may communicate with each other before sending the result of the computation back to the supervisor.

[redacted]

For example, a simple idea to sort an array of numbers using the supervisor-workers structure can be described as follows. The supervisor creates a number

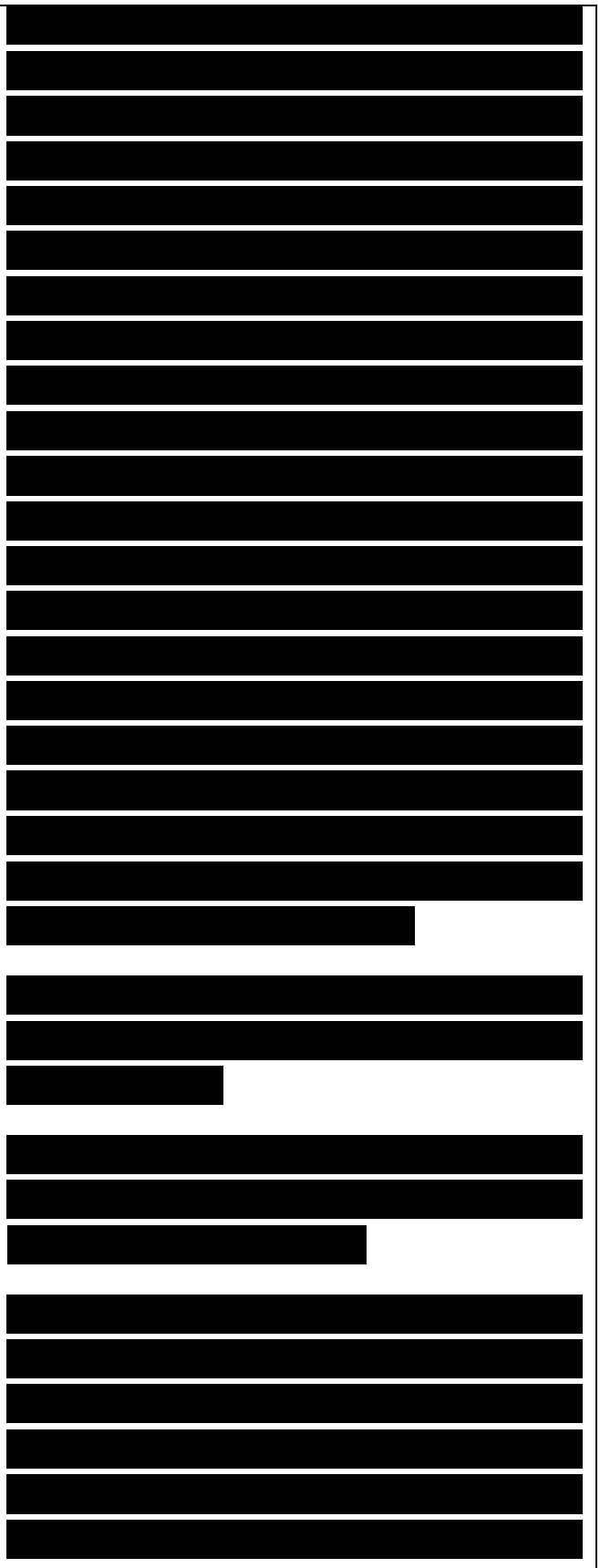
[redacted]

of workers and divides the array elements among them such that each worker gets an almost equal number of elements. Each worker independently sorts its share of the array and sends the sorted list back to the supervisor. The supervisor collects the sorted lists from the workers and merges them into one sorted list. Figure 8.1 shows an example of sorting an array of elements using one supervisor (S) and four workers (W1, W2, W3, and W4). Note that in this example the workers are entirely independent and communicate only with the supervisor that performs the merge procedure on the four sorted sublists while the workers remain idle.

(b) The supervisor is idle and the four workers are sorting their sublists

(d) The supervisor is merging the four sorted sublists and the four workers are idle

Figure 8.1 Supervisor-workers structure of sorting using a supervisor S and four independent workers W1, W2, W3, and W4. The solid edges indicate message passing. The dashed edges between S and W1, W2, W3, and W4 indicate that the workers were created by S.



Another way of sorting a list using the supervisor-workers structure is to make the workers help in the merge process and let the supervisor eventually merge only two sorted sublists. Figure 8.2 illustrates how this procedure works using one super-visor (S) and four workers (W1, W2, W3, and W4). First, the supervisor divides the list among the four workers. Each worker sorts its sublist independently. Workers W2 and W4 then send their sorted sublists to W1 and W3, respectively. Worker W1 will merge its sorted sublist with the one received from W2. Similarly, W3 will merge its sorted sublist with the one received from W4. Eventually the supervisor receives two sorted sublists from W1 and W3 to perform the final merge.

8.1.2 Hierarchy Structure

Unlike the supervisor-workers structure, the hierarchy structure allows the workers to create new levels of workers. The top-level supervisor is the initiating task, which creates a set of workers at the second level. These workers may create

other sets of ……	[redacted]
(b) The supervisor is idle and the four workers are sorting their sublists	[redacted]
(c) Workers W2 and W4 send their sorted sublists to W1 and W3, respectively mergemerge	[redacted]
(d) Workers W1 and W3 are merging two sublists each and W2 and W4 are idle	[redacted]
(e) Workers W1 and W3 send two sorted sublists to the supervisor workers at the next level, and so on. (A task that creates another task is also called its parent.) This task creation process can continue to any number of levels, forming a tree structure. The leaves of the tree are the workers at the lowest level. This structure matches very well with the organization of divide and conquer applications.	[redacted]
For example, sorting an array of elements using the hierarchy structure can be performed as follows. The top supervisor creates two workers and passes to each of them one-half of the array to sort. Each	[redacted]

worker will in turn create two new workers and send to each of them one-half of the already halved array to sort. This process will continue until the leaf workers have an appropriate number of elements to sort. These leaf workers will independently sort their lists and send them up to their parent to perform the merge operation. This process will continue upward until finally the top supervisor merges two sorted lists into the final sorted array.

Figure 8.3 illustrates the sorting algorithm using the hierarchy structure when eight leaf workers are used for sorting. Note that dashed edges in the tree signify a parent-child relationship between the tasks.

8.2 TASK CREATION

A task in PVM can be started manually or can be spawned from another task. The initiating task is always activated manually by simply running its executable code on one of the hosts. Other PVM tasks can be created dynamically from within other tasks. The function pvm_spawn () is used for dynamic task creation. The task that calls the function pvm_spawn() is referred to as the parent and the newly created tasks are called children. To create a child from a running parent, a programmer must at

least specify the following:

1. The machine on which the child will be started.
2. A path to the executable file on the specified machine.
3. The number of copies of the child to be created.
4. An array of arguments to the child task(s).

As all PVM tasks are identified by an integer task identifier, when a task is created it is assigned a unique identifier (TID). Task identification can be used to identify senders and receivers during communication. They can also be used to assign functions to different tasks based on their TIDs.

8.2.1 Task Identifier Retrieval

Parallel virtual machine provides a number of functions to retrieve TID values so that a particular task can identify itself, its parent, and other tasks in the system.

Task's TID A running task can retrieve its own TID by calling the PVM function pvm_myid() as follows:

(c) W3, W4, W5, and W6 merge two sublists each

……..

Child's TID When a task calls the function pvm_spawn(), an array containing the TIDs of the children created by this call will be returned. For example, the array tid in the following pvm_spawn() call will have the TIDs of all the children.

……

/* The TIDs of the children created by this call are saved in the array tid */

Parent's TID A task can retrieve the TID of its parent (the task from which it was spawned) by calling the function pvm_parent() as follows:

…….

The value PvmNoParent will be returned if the calling task is the one that was created manually and does not have a parent. This is an easy way to distinguish the supervisor from the workers in an application.

Daemon's TID A task can retrieve the TID of the daemon running on the same host as another task whose TID is id by calling the function pvm_tidtohost() as follows:

……

This function is useful for determining on which host a given task is running.

8.2.2 Dynamic Task Creation

The pvm_spawn() function is used to create one task or more on the same or a different machine in the PVM configuration. The format of this function is given as follows:

……

This function has six parameters and returns the actual number of the successfully created tasks in the variable num. The first two parameters are the executable file name of the program to be activated and the arguments to be passed to the execu-table (in standard argv format, terminated with a NULL).

The next two parameters specify where to start the process. The Flag parameter controls the target of the spawn operation. A value of zero lets PVM decides on the appropriate machine on which to start the task. Other values specify that the where parameter signifies a machine name, or an architecture type. Specifying a machine name gives the programmer ultimate control over the task allocation process. Spe-cifying an architecture type may be more appropriate in some cases, especially when the virtual machine is configured from a widely dispersed set of architectures. One of the requirements of the spawn command is that the executable must already exist on whatever machine it

is to run on.

TABLE 8.1 Parameters for Dynamic Task Creation

Parameter	Meaning
Child	The executable file name of the program to be started. The executable must reside on the host on which it will run.
Arguments	A pointer to an array of arguments to the program. If the program takes no arguments this pointer should be NULL.
Flag	A flag value of zero lets the PVM system decides what machine will run the spawned task(s). Other values signify that a particular host name or architecture type will be specified to run the spawned tasks.
Where	A host name or an architecture type to run the created tasks depending on the value of the above flag.
HowMany	The number of identical children to be started.
Tids	The TIDs of the children created by this call.

The final two parameters contain control information, such as the number of processes to spawn with this call, and an array in which to return information, such as task identifiers and error codes. The different parameters and their meanings are summarized in Table 8.1.

Example 1 Suppose that we want to create two and four copies of the program "worker" on the two hosts: homer and fermi, respectively. Assume that the execu-table file "worker" resides in the directory "/user/rewini" in both machines. The following two statements in the initiating task should create the required tasks:

……

The second parameter is 0 when there is no arguments to "worker". The third par-ameter is the spawn type flag, which was set to 1 so that we can specify homer and fermi as our target hosts. The TID values of the created tasks are returned in tid1 and tid2 . Finally n1 and n2 are the actual number of created tasks on homer and fermi, respectively.

8.3 TASK GROUPS

PVM allows running tasks to belong to named groups, which can change at any time during computation. Groups are useful in cases when a collective operation is performed on only a subset of the tasks. For example, a broadcast operation, which sends a message to all tasks in a system, can use a named group to send a message to only the members of this group. A task may join or leave a group at any time without informing other tasks in the group. A task may also belong to multiple groups. PVM provides several functions for tasks to join and leave a group, and retrieve information about

other groups.

A task can join a group by calling the function pvm_joingroup() as follows: i = pvm_joingroup(group_name)

This function adds the task that calls it to the group named group_name. It returns the instance number of the task that just joined the group. The group itself is created when pvm_j oingroup is called for the first time. In this case, the first caller gets 0 as instance number. The returned instance number starts at 0 and is incremented by 1 every time a new task joins the group. However, this set of instance numbers may have gaps as a result of having one or more tasks leave the group. When a task joins a group with gaps in the set of instance numbers, this new member will get the lowest available instance number. Maintaining a set of instance numbers without gaps is the programmer's responsibility.

A member of a group may leave the group by calling the function pvm_lvgroup () as follows:

……..

The task that successfully calls this function will leave the group group_name. In case of an error, info will have a negative value. If this task decides to rejoin this group at a later time, it may get a different instance number because the old number may have been assigned to another task that may have joined.

There are a number of other functions that can be called by any task to retrieve information without having to be a member of the specified group. For example, the function pvm_gsize() can be used to retrieve the size of a group. It takes as input the group name and returns the number of members in the group. The function pvm_gettid () is provided to retrieve the TID of a task given its instance number and its group name. Similarly, the function pvm_getinst () retrieves the instance number of a task given its TID and the name of a group to which it belongs.

Example 2 Suppose that tasks T0, T1, T2, and T3 have TIDs 200, 100, 300, and 400, respectively. Let us see what happens after the execution of each of the following statements.

1. Task T0 calls the function i1 = pvm_joingroup("slave")
The group "slave" is created, T0 joins this group and T0 is assigned the instance number 0 (i1 = 0).

2. Task T1 calls the function i2 = pvm_joingroup("slave")
T1 joins the group "slave" and is assigned instance number 1 (i2 = 1).

3. Task T2 calls the function i3 = pvm_joingroup("slave")
T2 joins the group "slave" and is assigned the instance number 2 (i3 = 2).

4. Task T1 calls the function info = pvm_lvgroup("slave")

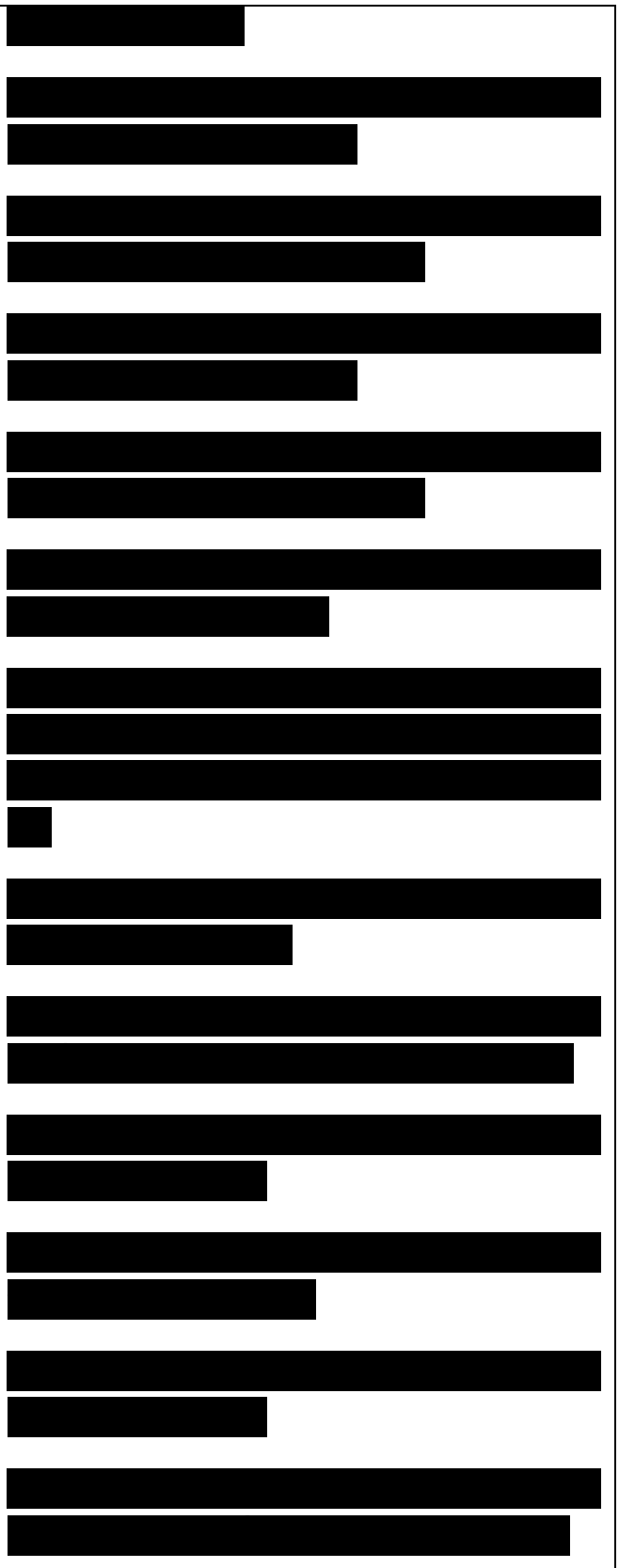
T1 leaves the group "slave" and the instance number 1 becomes available to other tasks that may wish to join the group "slave" in the future.

5. Some task calls the function size = pvm_gsize ("slave")

The variable size will be assigned the value 2, which is the number of tasks that currently belong to the group "slave".
6. Task T3 calls the function i4 = pvm_j oingroup("slave")
T3 joins the group "slave" and is assigned instance number 1 (i4 = 1).
7. Task T1 calls the function i5 = pvm_j oingroup("slave")

T1 rejoins the group "slave" and is now assigned the instance number 3 (i5 = 3).

8. Some task calls the function tid = pvm_gettid ("slave",1)



The variable tid will be assigned the value 400, which is the TID of the task T3 whose instance number is 1.

9. Some task calls the function inst = pvm_getinst("slave",100) The variable inst will be assigned the value 3, which is the instance number of the task T1 whose TID is 100.

8.4 COMMUNICATION AMONG TASKS

Communication among PVM tasks is performed using the message passing approach, which is achieved using a library of routines and a daemon. During program execution, the user program communicates with the PVM daemon through the library routines.

The daemon, which runs on each machine in the PVM environment, determines the destination of each message. If the message is sent to a task on the local machine, the daemon routes the message directly. If the message is for a task on a remote host, the daemon sends the message to the corresponding daemon on the remote machine. The remote daemon then routes the message to the right receiving task.

The operations Send and Receive are the heart of this communication scheme, which is generally asynchronous. A message can be sent to one or more destinations by calling one of the PVM

send functions. A message can be received by calling either a blocking or nonblocking receive function. Figure 8.4 schematically illustrates communication in PVM. The arrows from the user applications to the daemons represent communication calls (crossing the API boundary). The arrows from the daemons back to the user applications represent the return from the API calls. The thread of control of the user task briefly blocks on the daemon.

Using standard PVM asynchronous communication, a sending process issues a send command (point 1 in Fig. 8.4). The message is transferred to the daemon (point 2), then control is returned to the user application (points 3 and 4). The daemon will transmit the message on the physical link sometime after returning control to the user application (point 3). At some other time, either before or after the send command, the receiving task issues a receive command (point 5 in Fig. 8.4). In the case of a blocking receive, the receiving task blocks on the daemon waiting for a message (point 6). After the message arrives, control is returned to the user application (points 7 and 8). In the case of nonblocking receive, control is returned to the user application immediately (points 7 and 8) even if the message has not yet arrived.

Sending Task Receiving Task

…….

Figure 8.4 Communication in PVM.

A sender task can send a message to one or more receivers in three steps as follows:

1. A send buffer must be initialized.

2. The message is packed into this buffer.

3. The completed message is sent to its destination(s).

Similarly, receiving a message is done in two steps as follows:

1. The message is received.

2. The received items are unpacked from the receive buffer.

8.4.1 Message Buffers

Before packing a message for transmission, a send buffer must be created and prepared for data to be assembled into it. PVM provides two functions for buffer creation; pvm_initsend() and pvm_mkbuf(). These two functions agree on the input and output parameters. They take as input an integer value to specify the next message's encoding scheme, and they return an integer value specifying the message buffer identifier. The two functions are listed below.

bufid = pvm_initsend(encoding_option)
bufid = pvm_mkbuf(encoding_option)

There are three encoding options in creating the buffer. The default encoding option is XDR, which is useful when a message is sent to a different machine that may not be able to read the message native format. However, if this is not the case, another option is to skip the encoding step and a message is sent using its original format. A third option is to leave data in place and to make the send operation copy items directly from the user's memory. The buffer is used only to store the message size and pointers to the data items in this case. Clearly, the third option saves time by reducing the copying time but it requires that the user does not modify

TABLE 8.2 Encoding Options for Buffer Creation

.....

data before they are sent. The different values and the meanings of the different encoding options are summarized in Table 8.2.

If the user is using only one send buffer, pvm_initsend() should be the only required function. It clears the send buffer

and prepares it for packing a new message. The function pvm_mkbuf(), on the other hand, is useful when multiple message buffers are required in an application. It creates a new empty send buffer every time it is called. In PVM 3, there is only one active send buffer and one active receive buffer at any time. All packing, sending, receiving, and unpacking functions affect only the active buffer. PVM provides the following functions to set the active send (or receive) buffers to bufid. They save the state of the previous buffer and return its identifier in oldbuf.

oldbuf = pvm_setsbuf(bufid) oldbuf = pvm_setrbuf(bufid)

PVM also provides the functions pvm_getsbuf() and pvm_getrbuf() to retrieve the identifier of the active send buffer and the active receive buffer, respectively.

8.4.2 Data Packing

PVM provides a variety of packing functions pvm_pk* () to pack an array of a given data type into the active send buffer. Each of the packing functions takes three arguments as input. The first argument is a pointer to where the first item is, and the second argument specifies the number of items to be packed in an array. The third argument is the stride to use when packing (that is, how many

items to skip between two packed items). For example, a stride of 1 means a contiguous array is packed, a stride of 2 means every other item is packed, and so on. The packing functions return a status code, which will have a negative value in case of an error.

There are several packing functions for all kinds of data types such as byte, double, string, and so on. All the functions have the same number of arguments except the string packing function pvm_pkstr(), which takes only one argument (a pointer to the string). PVM also provides the function pvm_packf() that uses a printf like format expression to specify what to pack in the buffer before sending. Packing functions can be called multiple times to pack data into a single message. Other packing functions for the different data types include: pvm_pkbyte(), pvm_pkcplx(), pvm_pkdcplx(), pvm_pkdouble(), pvm_pkfloat(), pvm_pkint(), pvm_pklong(), pvm_pkshort(), pvm_pkuint(), pvm_pkushort(), pvm_pkulong().

Example 3 The following function calls pack a string followed by an array, called my_array, of n items into the message buffer:

info = pvm_pkstr("This is my data"); info=pvm pkint(my array, n, 1)

First, the string is packed and then n integers from the array list are packed into the send buffer. Note that there is no limit to the complexity of the packed message, but it should be unpacked exactly the same way at the receiving end.

8.4.3 Sending a Message

Sending messages in PVM is done in an asynchronous fashion. The sending task will resume its execution once the message is sent (points 3 and 4 in Fig. 8.4). It will not wait for the receiving task to execute the matching receive operation as in synchronous communication. Note that synchronous communication constructs for PVM were suggested in Lundell et al. (1996).

After the buffer is initialized and the packing process is completed, the message is now ready to be sent. A message can be sent to one or multiple receivers. All we need to specify at this point are an identifier for each task that should receive the message and a label (tag) for the message.

Sending to One Receiver The function pvm_s end() performs a point-to- point send operation. It takes two arguments: the TID of the destination task and an integer message identifier (tag). For

example, the function call

info=pvm_send(tid, tag)

will label the message packed in the send buffer with the label tag that is supplied by the programmer and send it to the task whose TID is tid. The call returns integer status code info. A negative value of info indicates an error.

Sending to Multiple Receivers To send the message to multiple destinations, the function pvm_mcast() should be used. The TIDs of the tasks that will receive the message should be saved in an array. A pointer to the TIDs array, the number of recipient tasks, and the message label are the arguments to pvm_mcast (). For example, the function call
info=pvm_mcast(tids, n, tag)

will label the message with the integer tag and send it to the n tasks whose TIDs are specified in the array tids. Again the status code info indicates whether the call was successful. Note that the message will never be sent to the caller task even if its TID was included in the array tids.

Sending to a Group A message can be broadcast to all members of a group using the function pvm_bcast(). Any task can call this function without having to be a member of the group. The arguments of this function are the group name and the

message tag. It first determines the TIDs of the group members and then uses pvm_mcast () to broadcast the message. For example, the function call

info = pvm_bcast(group_name, tag)

will label the message with the integer tag and send it to all members of the group group_name. Note that if the group changes during the broadcast, the change will not be reflected. Since group changes are not collective operations over the group, the result of collective operations cannot be predicted unless synchronization is done by hand.

Packing and Sending in One Step PVM also provides another function to send messages without the need to prepare and pack the buffer manually. The operation pvm_psend() does the packing automatically for the programmer. In addition to the destination TID and the message label, pvm_psend() takes a pointer to a buffer, its length, its data type as arguments. For example, the call

info=pvm_psend(tid, tag, my_array, n, int)

packs an array of n integers called my_array into a message labeled tag, and sends it to the task whose TID is tid.

8.4.4 Receiving a Message

PVM supports three types of message receiving functions: blocking, nonblocking, and timeout. When calling a blocking receive function, the receiving task must wait until the expected message arrives in the receive buffer. A nonblocking receive immediately returns with either the expected data or a flag that the data have not arrived. Timeout receive allows the programmer to specify a period of time for which the receive function should wait before it returns. If the timeout period is very large, this function will act like the blocking receive. On the other hand, if the timeout period is set to zero, it acts exactly like the nonblocking case. Figure 8.5 illustrates the three types of receive operations.

Blocking Receive

bufid=pvm_recv(tid, tag)

This function will wait until a message with label tag is received from a task with TID = tid. A value of — 1 can be used as a wild card to match anything in either one of the arguments: tid or tag. A successful receive will create a receive buffer and return the buffer identifier to be used in unpacking the message.

Time is expired

Figure 8.5 The three types of receive operations.

Nonblocking Receive

bufid=pvm_nrecv(tid, tag)

If the message has arrived successfully when this function is called, it will return a buffer identifier similar to the case of blocking receive. However, if the expected message has not arrived, the function will return immediately with bufid = 0.

Timeout Receive

bufid=pvm_trecv(tid, tag, timeout)

This function blocks the execution of its caller task until a message with a label tag has arrived from tid within a specified waiting period of time. If there is no match-ing message arriving within the specified waiting time, this function will return with bufid = 0, which indicates that no message was received. The waiting time argu-ment (timeout) is a structure with two integer fields tv_sec and tv_usec. With both fields set to zero, this function will act as a nonblocking receive. Passing a null pointer as timeout makes the function act like a blocking receive. If pvm_trecv() is successful, bufid will have the value of the new active receive buffer

identifier.

Receive and Unpack in One Step Similar to the pvm_psend() function, PVM provides the function pvm_precv(), which combines the functions of blocking receive and unpacking in one routine. It does not return a buffer identifier; instead it returns the actual values. For example, the following call …….

will block until a matching message is received. The contents of the message will be saved in my_array up to length len. In addition to the status code info, the actual TID of the sender, actual message tag, and the actual message length are returned in src, atag, and alen, respectively. Again the value — 1 can be used as a wild card for the arguments: tag or tid.

8.4.5 Data Unpacking

When messages are received, they need to be unpacked in the same way they were packed in the sending task. Unpacking functions must match their corresponding packing functions in type, number of items, and stride.

PVM provides many unpacking functions pvm_upk* (), each of which corresponds to a particular packing function. Similar

to packing functions, each of the unpacking functions takes three arguments as input. These arguments are address of the first item, number of items, and stride. PVM also provides the two functions pvm_upkstr() and pvm_unpackf() to unpack the messages packed by pvm_pkstr () and pvm_packf(), respectively.

Other unpacking functions for the different data types include: pvm_ upkbyte(), pvm_upkcplx(), pvm_upkdcplx(), pvm_upkdouble(), pvm_upkfloat(), pvm_upkint(), pvm_upklong(), pvm_upkshort(), pvm_upkuint(), pvm_upkushort(), pvm_upkulong().

Example 4 The following function calls unpack a string followed by an array of n items from the receive buffer:
info = pvm_upkstr(string)
info=pvm_upkint(my_array, n, 1)

Note that the string and the array must have been packed using the corresponding packing functions.

8.5 TASK SYNCHRONIZATION

Synchronization constructs can be used to force a certain order of execution among the activities in a parallel program. For example, a task that uses certain variables in its computation must wait until these variables are computed (possibly by other tasks) before it resumes its execution.

Even without data dependence involvement, parallel tasks may need to synchronize with each other at a given point in the execution. For example, members of a group that finish their work early may need to wait at a synchronization point until those tasks that take a longer time reach the same point. Synchronization in PVM can be achieved using several constructs, most notably blocking receive and barrier operations.

[redacted]

8.5.1 Precedence Synchronization

Message passing can be used effectively to force precedence constraints among tasks. Using the blocking receive operation (pvm_recv()) forces the receiving task to wait until a matching message is received. The sender of this matching message may hold its message as long as it wants the receiver to wait. For example, con-sider the two tasks; T0 and T1 in Figure 8.6. Suppose that we want to make sure that the function g() in T1 is not executed until T0 has completed the execution of the function f(). This particular order of execution can be guaranteed using a send operation after calling f() in T0, and a matching blocking receive operation before calling g() in T1.

[redacted]

8.5.2 Barriers

Parallel tasks can be synchronized through the use of synchronization points called barriers. No task may proceed beyond a barrier until all participating tasks have reached that barrier. Members of a group can choose to wait at a barrier until a specified number of group members check in at that barrier. PVM provides barrier synchronization through the use of the function pvm_barrier (). This function takes two inputs: the group name, and the number of group members that should call this function before any of them can proceed beyond the barrier as follows.

info =pvm_barrier(group_name, ntasks)

Again the status code info will return a negative integer value in case of an error. The number of members specified could be set to any number less than or equal to ……

Figure 8.6 Precedence synchronization using message passing. The function f() in T0 is guaranteed to be executed before the function g() in T1.

the total number of members. However, it is typically the total number of members in the group. In any case, the value of this argument should match across a given barrier call. If this argument is set to — 1,

PVM will use the value of pvm_gsize (), which returns the total number of members. Since it is possible for tasks to join the group after other tasks have already called pvm_barrier (), it is necessary to specify the number of tasks that should check in at the barrier. It is not allowed for a task to call pvm_barrier () with a group to which it does not belong.

Example 5 Figure 8.7 shows three members of the group slave (T0, T1, T2) using a barrier to synchronize at a certain point in their execution. Each of the three tasks should call the following function: info=pvm_barrier("slave", 3)

The execution will block until three members of the group slave have issued the call to function pvm_barrier () as shown in the figure. Task T1 calls the function first, followed by T0, and then finally T2. Tasks T0 and T1 wait at the barrier until T2 reaches the barrier before they can all proceed.

8.6 REDUCTION OPERATIONS

Reduction is an operation by which multiple values are reduced into a single value. This single value could be the maximum (minimum) value, the summation (product) of all elements, or the result of applying an associative binary operator that yields a single result. PVM supports reduction through the use of the function pvm_ reduce(). The format

of this function is given as follows:
info=pvm_reduce(func, data, n, datatype, tag, group_name, root)

Parameter	Meaning
func	The function that defines the operation to be performed.
data	An array of data elements.
n	The number of elements in the data array.
datatype	The type of entries in the data array.
tag	Message tag.
group_name	The name of an existing group.
root	Instance number of a group member who gets the result.

The function returns an integer status code (info). The different parameters and their meanings are summarized in Table 8.3.

The reduction operation is performed on the corresponding elements in the data array across the group. The reduced value for each element in the array across the group will be returned to the root specified in the parameters. In fact, the data array on the root will be overwritten with the result of the reduction operation over the group. Users can define their own functions or can use several PVM

predefined functions such as PvmMin, PvmMax, PvmSum, and PvmProduct for the minimum, maximum, summation, and product, respectively.

Example 6 Figure 8.8 shows an example of a reduction summation of the entries of data_array over the group "slave", which has three members: T0, T1, and T2. The reduced values are returned to the root, which is assumed to be task T1 in this example. The following function must be called by the three tasks.

……

8.7 WORK ASSIGNMENT

Assigning work to workers can be done either by writing a separate program for each worker or writing a single program for all workers. If the workers perform the same computation on different sets of data concurrently, it is appropriate to use a single program for all workers. On the other hand, if the workers perform different functions, it is possible to do it either way. In this section, we show how to assign work to the parallel tasks.

8.7.1 Using Different Programs

If the workers forming the parallel application perform completely different oper-ations, they can be written as different programs. These different

workers can be activated by the initiating task (supervisor) using pvm_spawn(). The supervisor can communicate with the workers since it knows their TIDs, which are returned when pvm_spawn() is called. To communicate with the supervisor, the workers need to know the supervisor's TID. The function pvm_parent() returns the supervisor's TID when called by the workers.

Example 7 Suppose that we want to activate four different tasks: "workerl", "worker2", "worker3", and "worker4" on the hosts cselabOl, cselab02, cselab03, and cselab04, respectively. Assume that the executable files reside in the directory "/user/rewini" in all machines. The following statements in the initiating task will create the required tasks.

……..

8.7.2 Using the Same Program

Assigning work to parallel tasks running the same program can be done easily if we know in advance the identification numbers assigned by the system. For example, if we know that the identification numbers of n — 1 workers running the same program are 1,2, ... , n — 1, we can assign work to these tasks as follows:

8.7.2 Sử dụng cùng một chương trình

Có thể dễ dàng phân công công việc cho các tác vụ song song chạy cùng một chương trình nếu chúng ta biết trước các mã định danh (các số nhận dạng) được chỉ định bởi hệ thống. Ví dụ, nếu chúng ta biết các mã định danh (các số nhận dạng) của n — 1 người thi hành (công nhân) đang chạy cùng một chương trình lần lượt

…… Unfortunately, a task in PVM is assigned any integer as its identification number. Tasks are not necessarily identified by the integers 1, 2, 3, and so on, as shown in the above example. In what follows, we show how to overcome this problem. Using Task Groups In this method, all the tasks join one group and the instance numbers are used as the new task identifiers. The supervisor is the first one to join the group and gets 0 as its instance number. The workers will get instance numbers in the range from 1 to n — 1. Using Task ID Array In this method, the supervisor sends an array containing the TIDs of all the tasks to all the workers. The supervisor TID is saved in the zeroth element of the array, and the workers' TIDs are saved in elements 1 to n — 1. Each worker searches for its own TID in the array received from the supervisor and the index can be used to identify the corresponding worker as shown in Figure 8.9.	là 1, 2,..., n — 1, thì chúng ta có thể phân công công việc cho các tác vụ này như sau: …… Tuy nhiên, một tác vụ trong PVM (máy ảo song song) được gán một số nguyên bất kỳ làm mã định danh (số nhận dạng) của nó. Các tác vụ không nhất thiết phải được nhận dạng bởi các số nguyên 1, 2, 3 v.v., như được thể hiện trong ví dụ nêu trên. Trong phần tiếp theo, chúng ta sẽ chỉ ra cách để khắc phục vấn đề này. Sử dụng các nhóm tác vụ Trong phương pháp này, tất cả các tác vụ tham gia vào một nhóm và các số trường hợp (số phiên bản) được sử dụng như là bộ nhận dạng tác vụ mới. Giám sát viên là phần tử đầu tiên tham gia vào nhóm và nhận số 0 làm số trường hợp (số phiên bản) của nó. Các người thi hành (công nhân) sẽ được nhận được các số trường hợp (số phiên bản) trong phạm vi từ 1 đến n-1. Dùng mảng Task ID (mảng nhận dạng tác vụ) trong phương pháp này, giám sát viên gửi một mảng chứa các TID của tất cả các tác vụ cho tất cả các người thi hành (công nhân). TID của giám sát viên được lưu vào các phần tử 0 của mảng, và các TID của các người thi hành (công nhân) được lưu vào các phần tử từ 1 đến n — 1. Mỗi người thi hành (công nhân) tìm kiếm TID cho mình trong mảng nhận được từ giám sát viên và có thể sử dụng chỉ số để nhận dạng người thi hành (công nhân) tương ứng như được chỉ ra trên hình 8.9.

1. What has been the trend in computing from the following points of views: (a) cost of hardware; (b) size of memory; (c) speed of hardware; (d) number of processing elements; and (e) geographical locations of system components.	1. Xu hướng của máy tính sẽ là gì theo các quan điểm sau đây: a. Chi phí của phần cứng b. Dung lượng của bộ nhớ c. Tốc độ của phần cứng d. Số lượng bộ xử lý e. Vị trí địa lý của các thành phần hệ thống
2. Given the trend in computing in the last 20 years, what are your predictions for the future of computing?	2. Biết được xu hướng của máy tính 20 năm về trước, hãy dự đoán tương lai của máy tính?
3. What is the difference between cluster computing and grid computing?	3. Sự khác biệt giữa điện toán cụm và điện toán lưới là gì?
4. Assume that a switching component such as a transistor can switch in zerotime. We propose to construct a disk-shaped computer chip with such a component. The only limitation is the time it takes to send electronic signals from one edge of the chip to the other. Make the simplifying assumption that electronic signals can travel at 300,000 km/s. What is the limitation on the diameter of a round chip so that any computation result can by used anywhere on the chip at a clock rate of 1 GHz? What are the diameter restrictions if the whole chip should operate at 1 THz = 1012 Hz? Is such a chip feasible?	4. Giả sử rằng linh kiện chuyển mạch như transistor có thể chuyển mạch ngay tức khắc. Chúng tôi đề nghị nên chế tạo chip máy tính có dạng đĩa bằng linh kiện loại này. Nhược điểm duy nhất là thời gian cần để gửi tín hiệu điện từ cạnh này đến cạnh kia của chip. Cứ cho một giả định đơn giản rằng (Để đơn giản chúng ta giả sử rằng) tín hiệu điện có thể truyền với tốc độ 300,000 km/s. Đường kính của chip tròn phải như thế nào để bất kỳ mọi kết quả tính toán có thể sử dụng ở bất cứ nơi đâu trên con chip có xung nhịp 1 GHz? Đường kính giới hạn là gì nếu toàn bộ chip phải vận hành ở 1 THz = 1012 Hz? Đó có phải là chip khả thi không?
5. Compare uniprocessor systems	5. So sánh giữa hệ thống bộ xử lý đơn

with multiprocessor systems for the following aspects: (a) ease of programming; (b) the need for synchronization; (c) performance evaluation; and (d) run time system. 6. Provide a list of the main advantages and disadvantages of SIMD and MIMD machines.	và hệ thống đa xử lý ở các khía cạnh sau: ~~a.~~ Dễ lập trình b. Nhu cầu đồng bộ hóa c. Đánh giá hiệu suất; và d. Hệ thời gian chạy. 6. Cung cấp bảng liệt kê về các ưu và nhược điểm chính của máy tính SIMD và MIMD.
MỘT BẢN DỊCH KHÁC 1. What has been the trend in computing from the following points of views: (a) cost of hardware; (b) size of memory; (c) speed of hardware; (d) number of processing elements; and (e) geographical locations of system components.	1. Hãy nêu xu hướng tin học từ những quan điểm dưới đây: (a) hao tổn phần cứng; (b) dung lượng bộ nhớ; (c) tốc độ phần cứng; (d) số lượng phần tử xử lý; và (e) vị trí địa lý của các thành phần hệ thống.
2. Given the trend in computing in the last 20 years, what are your predictions for the future of computing?	2. Với xu hướng trong tin học 20 năm gần đây, bạn dự đoán gì về tương lai của ngành tin học?
3. What is the difference between cluster computing and grid computing?	3. Đâu là sự khác nhau giữa điện toán cụm và điện toán lưới?
4. Assume that a switching component such as a transistor can switch in zerotime. We propose to construct a disk-shaped computer chip with such a component. The only limitation is the time it takes to send electronic signals from one edge of the chip to the other. Make the simplifying assumption that electronic signals can travel at 300,000 km/s. What is the limitation on the diameter of a round chip so that any computation result can by used anywhere on the chip at a clock rate of 1 GHz?	4. Giả sử 1 thiết bị đóng ngắt mạch như tran-zit-tơ có thể đóng ngắt ngay tức thì. Ta dự định nối 1 vi mạch máy tính hình đĩa với 1 thiết bị như vậy. Hạn chế duy nhất là thời gian truyền tín hiệu điện từ cạnh này tới cạnh khác của vi mạch. Lấy giả định đơn giản là tín hiệu điện truyền đi với tốc độ 300.000km/s. Giới hạn đường kính của một vi mạch tròn là bao nhiêu để bất kỳ kết quả xử lý nào cũng có thể được sử dụng ở bất cứ vị trí nào trên vi mạch với tốc độ xử lý 1 GHz? Giới hạn đường kính là bao nhiêu nếu

What are the diameter restrictions if the whole chip should operate at 1 THz = 1012 Hz? Is such a chip feasible?	toàn bộ vi mạch xử lý với tốc độ 1 THz = 1012 Hz? Vi mạch như vậy có khả thi không?
5. Compare uniprocessor systems with multiprocessor systems for the following aspects: (a) ease of programming; (b) the need for synchronization; (c) performance evaluation; and (d) run time system. 6. Provide a list of the main advantages and disadvantages of SIMD and MIMD machines.	5. So sánh hệ thống đơn xử lý với hệ thống đa xử lý trên các phương diện sau: (a) Dễ lập trình; (b) yêu cầu đồng bộ hóa; (c) đánh giá sự hiệu suất; và (d) hệ thống thời gian chạy. 6. Hãy đưa ra danh sách những ưu và nhược điểm chính của các máy SIMD và MIMD.
7. Provide a list of the main advantages and disadvantages of shared-memory and message-passing paradigm.	7. Cung cấp một danh sách các ưu và khuyết điểm chính của mô hình bộ nhớ dùng chung và truyền thông báo;
8. List three engineering applications, with which you are familiar, for which SIMD is most efficient to use, and another three for which MIMD is most efficient to use.	8. Liệt kê 3 ứng dụng kỹ thuật, mà bạn biết trong đó SIMD thích hợp nhất để sử dụng, và 3 ứng dụng khác thích hợp với MIMD;
9. Assume that a simple addition of two elements requires a unit time. You are required to compute the execution time needed to perform the addition of a 40 x 40 elements array using each of the following arrangements:	9. Giả sử rằng một phép cộng đơn giản 2 phần tử yêu cầu 1 đơn vị thời gian. Bạn cần tính thời gian cần thiết để thực hiện cộng mảng 40x40 phần tử sử dụng một trong các cách sắp xếp sau đây :
(a) A SIMD system having 64 processing elements connected in nearest-neighbor fashion. Consider that each processor has only its local memory.	a) Một hệ thống SIMD có 64 phần tử xử lý được kết nối với theo kiểu lân cận gần nhất. Giả sử rằng mỗi bộ xử lý chỉ có bộ nhớ riêng của nó;
(b) A SIMD system having 64 processing elements connected to a shared memory through an interconnection network. Ignore the communication time.	b) Một hệ thống SIMD có 64 phần tử xử lý được kết nối với bộ nhớ dùng chung qua mạng liên thông. Bỏ qua thời gian

	trao đổi thông tin.
(c) A MIMD computer system having 64 independent elements accessing a shared memory through an interconnection network. Ignore the communication time. (d) Repeat (b) and (c) above if the communication time takes two time units.	c) Một hệ thống máy tính MIMD có 64 phần tử độc lập truy cập một bộ nhớ dùng chung qua mạng liên thông. Bỏ qua thời gian trao đổi thông tin; d) Lặp lại b) và c) ở trên nếu thời gian trao đổi thông tin có tới 2 đơn vị thời gian.
10. Conduct a comparative study between the following interconnection networks in their cost, performance, and fault tolerance: (a) bus; (b) hypercube; (c) mesh; (d) fully connected; (e) multistage dynamic network;	10. Tiến hành nghiên cứu so sánh giữa các mạng liên thông theo giá thành, hiệu suất, khả năng kháng lỗi: a) bus; b) siêu lập phương (siêu khối); c) lưới; d) Kết nối hoàn toàn; e) Mạng động đa tầng;
7. Provide a list of the main advantages and disadvantages of shared-memory and message-passing paradigm.	7. Liệt kê danh sách các điểm ưu và nhược điểm chính của bộ nhớ dùng chung và mô hình truyền thông điệp.
8. List three engineering applications, with which you are familiar, for which SIMD is most efficient to use, and another three for which MIMD is most efficient to use.	8. Liệt kê ba ứng dụng kỹ thuật quen thuộc với bạn sử dụng SIMD hiệu quả nhất và ba ứng dụng kỹ thuật khác sử dụng MIMD hiệu quả nhất.
9. Assume that a simple addition of two elements requires a unit time. You are required to compute the execution time needed to perform the addition of a 40 x 40 elements array using each of the	9. Giả sử một phép cộng đơn giản gồm hai phần tử cần một đơn vị thời gian. Bạn được yêu cầu tính thời gian cần thiết để thực hiện một phép cộng mảng 40x40

following arrangements:	phần tử sử dụng một trong các cấu hình dưới đây:
(a) A SIMD system having 64 processing elements connected in nearest-neighbor fashion. Consider that each processor has only its local memory.	(a) Một hệ thống đơn lệnh đa dữ liệu (SIMD) có 64 đơn vị xử lý được kết nối theo kiến trúc lân cận gần nhất. Coi mỗi bộ xử lý chỉ sử dụng một bộ nhớ riêng.
(b) A SIMD system having 64 processing elements connected to a shared memory through an interconnection network. Ignore the communication time.	(b)Một hệ thống đơn lệnh đa dữ liệu (SIMD) có 64 đơn vị xử lý được kết nối với một bộ nhớ dùng chung qua một mạch liên kết. Bỏ qua thời gian truyền thông.
(c) A MIMD computer system having 64 independent elements accessing a shared memory through an interconnection network. Ignore the communication time.	(c) Một hệ thống máy tính đa lệnh đa dữ liệu (MIMD) có 64 đơn vị xử lý độc lập kết nối với một bộ nhớ dùng chung qua một mạng liên thông. Bỏ qua thời gian truyền thông.
(d) Repeat (b) and (c) above if the communication time takes two time units.	(d) Làm như (b) và (c) nếu thời gian kết nối là 2 đơn vị thời gian.
10. Conduct a comparative study between the following interconnection networks in their cost, performance, and fault tolerance:	10. Tiến hành một nghiên cứu so sánh giữa các mạng liên kết dưới đây trên các phương diện chi phí, hiệu suất và khả năng kháng lỗi của chúng:
(a) bus; (b) hypercube; (c) mesh; (d) fully connected; (e) multistage dynamic network;	(a) kênh truyền (bus) (b) siêu lập phương (hypercube) (c) lưới (mesh) (d) kết nối hoàn toàn (fully connected) (e) mạng động đa tầng (multistage dynamic network)

(f) crossbar switch.

.....

.....

.....

PROBLEMS

1. Design a nonblocking Clos network that connects 16 processors and 16 memory modules. Show clearly the number of crossbar switches needed, together with their interconnection pattern.
2. Consider the case of an 8 x 8 single-stage recirculating Shuffle-Exchange network. Determine all input-output combinations that require the maximum number of passes through the network.
3. Consider the case of an 8x8 Banyan multistage interconnection network similar to the one shown in Figure 2.8. Determine whether it is possible to connect input #I to output (i mod 8) for all I simultaneously. If it is possible show the routing in each case.
4. Consider a simple cost comparison between an n x n crossbar and an n x n Shuffle-Exchange MIN. While the crossbar uses cross points, the Shuffle network uses 2 x 2 switching elements (SEs). Assume that the cost of a 2 x 2SE is four times that of a cross point. What is the relative cost of an n x n Shuffle-Exchange network with respect to that of a crossbar of the same size? Determine the smallest value of n for which the cost of the crossbar is four times that of the Shuffle-Exchange.

(f) Bộ chuyển mạch thanh ngang (chuyển mạch ngang dọc)

CÁC BÀI TẬP

1. Thiết kết hệ thống mạng Clos không chờ kết nối 16 bộ xử lý với 16 mô dun bộ nhớ. Biểu diễn rõ số bộ chuyển mạch thanh chéo cần thiết, cùng với mô hình liên kết.
2. Xét mạng Shuffle-Exchange xoay vòng đơn tầng 8 x 8. Xác định tất cả các kết hợp vào ra cần số lần đi qua mạng nhiều nhất.
3. Xét một hệ thống mạng liên thông đa tầng 8 x 8 tương tự như hệ thống ở hình 2.8. Xem xét khả năng kết nối đầu vào #I với đầu ra(I mod 8) cùng lúc đối với tất cả I. Nếu có thể, hãy phát họa sự định tuyến cho từng trường hợp.
4. Xét một phép so sánh chi phí đơn giản giữa một hệ thống crossbar n x n và một hệ thống Shuffle-Exchange n x n MIN. Trong khi, crossbar sử dụng các điểm chéo, Shuffle sử dụng các phần tử chuyển mạch 2 x 2 (PTCM). Giả sử chi phí của một PTCM 2 x 2 gấp 4 lần chi phí của điểm chéo. Chi phí tương đối của một hệ thống Shuffle-Exchange n x n so với hệ thống crossbar cùng kích cỡ nhau là bao nhiêu? Xác định giá trị nhỏ nhất của n để chi phí của hệ thống crossbar gấp bốn lần hệ thống

	Shuffle-Exchange. 5. Trong quá trình tính toán số liên kết cho các hệ thống nhiều bus khác nhau, cần phải lưu ý rằng tất cả hệ thống nhiều bus cần phải có ít nhất BN liên kết. Tuy nhiên, số lượng liên kết bổ sung sẽ khác nhau. Ví dụ, MBFBMC cần thêm BM liên kết bổ sung trong khi MBSBMC chỉ cần M liên kết bổ sung. Bạn cần phải so sánh bốn hệ thống đa bus dựa trên số lượng liên kết bổ sung của từng hệ thống. Bạn có thể giả định một số giá trị cụ thể cho B, N, M, g, và k. Xét trường hợp nối N=100 bộ xử lý với M=400 mô-dun bộ nhớ sử dụng B= 40 bus. Xác định giá trị tối ưu của g và k để hệ thống MBCBMC luôn tốt hơn hệ thống MBPBMC dựa trên (theo) số lượng liên kết bổ sung.
5. In computing the number of connections for different multiple-bus systems, it is noticed that all multiple-bus systems require at least BN connections. However, they differ in the number of additional connections required. For example, while the MBFBMC requires BM additional connections, the MBSBMC requires only M additional connections. You are required to compare the four multiple-bus systems in terms of the additional number of connections required for each. You may assume some numerical values for B, N, M, g, and k. Consider the case of connecting N = 100 processors to M = 400 memory modules using B = 40 buses. Determine the optimal values for g and k such that the MBCBMC system is always better that the MBPBMC in terms of the number of additional connections.	
(f) crossbar switch. …… …… ……	Bộ chuyển mạch thanh ngang (chuyển mạch ngang dọc) …… …… ……
PROBLEMS 1. Design a nonblocking Clos network that connects 16 processors and 16 memory modules. Show clearly the number of crossbar switches needed, together with their interconnection pattern. 2. Consider the case of an 8 x 8	CÁC BÀI TẬP Thiết kế một mạng Closs không chờ ~~mà~~ liên kết 16 bộ vi xử lý và 16 modun bộ nhớ. Chỉ ra một cách rõ ràng số bộ chuyển mạch thanh ngang cần dùng, cùng với mô hình kết nối của chúng.

single-stage recirculating Shuffle-Exchange network. Determine all input-output combinations that require the maximum number of passes through the network.	Xét trường hợp ~~của~~ mạng Shuffle-Exchange một tầng quay vòng (luân chuyển) 8x8. Xác định tất cả những kết hợp đầu vào- đầu ra cần số lần qua mạng nhiều nhất.
3. Consider the case of an 8x8 Banyan multistage interconnection network similar to the one shown in Figure 2.8. Determine whether it is possible to connect input #I to output (i mod 8) for all I simultaneously. If it is possible show the routing in each case.	Xét trường hợp mạng liên thông đa tầng Banyan 8x8 tương tự như hệ thống được nêu ra trong hình 2.8. Xác định xem có thể kết nối đầu vào # I với đầu ra (I mod 8) cho tất cả I cùng một lúc không. Nếu có thể chỉ ra định tuyến trong mỗi trường hợp
4. Consider a simple cost comparison between an n x n crossbar and an n x n Shuffle-Exchange MIN. While the crossbar uses cross points, the Shuffle network uses 2 x 2 switching elements (SEs). Assume that the cost of a 2 x 2SE is four times that of a cross point. What is the relative cost of an n x n Shuffle-Exchange network with respect to that of a crossbar of the same size? Determine the smallest value of n for which the cost of the crossbar is four times that of the Shuffle-Exchange.	Xét một sự so sánh chi phí đơn giản giữa một crossbar n x n và mạng liên kết đa tầng Shuffle-Exchange. Trong khi crossbar sử dụng các giao điểm, thì hệ thống Shuffle-Exchange sử dụng những phần tử chuyển mạch 2x2(SEs). Giả sử rằng chi phí của một SE gấp 4 lần chi phí của một giao điểm. Chi phí tương đối của một hệ thống Shuffle-Exchange n x n đối với crossbar cùng kích cỡ là gì? Xác định giá trị nhỏ nhất của n trong đó chi phí của crossbar ~~là~~ bằng 4 lần so với của Shuffle-Exchange.
5. In computing the number of connections for different multiple-bus systems, it is noticed that all multiple-bus systems require at least BN connections. However, they differ in the number of additional connections required. For example, while the MBFBMC requires	Trong quá trình tính toán số lượng kết nối cho những hệ thống nhiều bus khác nhau, chúng ta thấy rằng tất cả những hệ thống đa kênh yêu cầu ít nhất BN kết nối. Tuy nhiên, chúng khác nhau về số kết nối bổ sung cần thiết. Ví dụ như, trong khi

BM additional connections, the MBSBMC requires only M additional connections. You are required to compare the four multiple-bus systems in terms of the additional number of connections required for each. You may assume some numerical values for B, N, M, g, and k. Consider the case of connecting N = 100 processors to M = 400 memory modules using B = 40 buses. Determine the optimal values for g and k such that the MBCBMC system is always better that the MBPBMC in terms of the number of additional connections.	MBFBMC yêu cầu BM kết nối bổ sung, thì MBSBMC chỉ yêu cầu M kết nối bổ sung. Bạn được yêu cầu so sánh 4 hệ thống đa kênh theo số kết nối bổ sung cần thiết cho mỗi loại. Bạn có thể giả định một vài giá trị số cho B,N,M,g và k. Xét trường hợp kết nối N=100 bộ vi xử lý với M=400 modun bộ nhớ sử dụng B=40 bus. Xác định những giá trị tối ưu của g và k để hệ thống MBCBMC luôn luôn tốt hơn MBPBMC theo số lượng kết nối bổ sung.
6. Consider the two MINs shown in Figures 2.10 and 2.11. At first glance one can notice the difference between these two networks. In particular, while the first one (the Shuffle-Exchange) uses straight connections between the input processors and the network inputs and straight connections between the output of the network and the output memory modules, the second network (the Banyan network) uses straight connections at the inputs but a shuffle connection at the output. If we generalize that principle such that at the input and the output we can have either straight or shuffle connections while keeping the connection among stages as shown, how many different types of networks will result? Characterize the resulting networks in terms of their ability to interconnect all inputs to all outputs simultaneously.	6.Xét hai MIN trong hình 2.10 và 2.11. Thoạt nhìn, chúng ta có thể thấy sự khác nhau giữa hai mạng này. Đặc biệt, trong khi mạng đầu tiên (the Shuffle-Exchange) sử dụng các kết nối thẳng giữa giữa các bộ xử lý đầu vào và các đầu vào mạng và kết nối thẳng giữa đầu ra mạng và các mô đun bộ nhớ đầu ra, mạng thứ hai (mạng Banyan) sử dụng các kết nối thẳng tại các đầu vào nhưng sử dụng kết nối shuffle tại đầu ra. Nếu chúng ta tổng quát hóa nguyên tắc đó sao cho ở các đầu vào và đầu ra chúng ta có kết có các kết nối thẳng hoặc kết nối shuffle trong khi vẫn giữ kết nối giữa các tầng như đã biểu diễn, có bao nhiêu loại khác nhau được hình thành? Xác định tính chất đặc trưng của các loại mạng này theo khả năng kết nối đồng thời tất cả các đầu vào với tất cả các đầu ra của chúng.
7. Repeat Problem 6 above for the cases whereby the interstage connection	7.Làm lại bài tập 6 ở trên cho trường hợp

patterns can be either straight or shuffle.	mô hình kết nối giữa các tầng có thể thẳng hoặc shuffle.
8. Assume that we define a new operation, call it inverse shuffle (IS), which is defined as …… Repeat Problems 7 and 8 above if the IS is used instead of the shuffle operation.	8.Giả sử rằng chúng ta định nghĩa một phép toán mới. gọi là shuffle ngược (IS), nó được định nghĩa là Làm lại bài tập 7 và 8 ở trên nếu dùng phép toán IS thay cho shuffle.
9. Determine the maximum speedup of a single-bus multiprocessor system having N processors if each processor uses the bus for a fraction f of every cycle.	9.Xác định độ tăng tốc cực đại của hệ đa xử lý đơn bus có N bộ xử lý nếu mỗi bộ xử lý dùng bus trong khoảng một phần f của mỗi chu kỳ.
10. Discuss in some details the fault-tolerance features of dynamic INs such as multiple-bus, MINs, and crossbar. In particular, discuss the effect of failure of nodes and/or links on the ability of routing in each network. Repeat the same for static networks such as hypercubes, meshes, and tree networks.	10. Thảo luận tính chất khánh lỗi của các IN động chẳng hạn như nhiều bus, các MIN, và crossbar. Đặc biệt, thảo luận ảnh hưởng của sự hư hỏng của các nút và/hoặc link đến khả năng định tuyến trong mỗi mạng. Lặp lại điều tương tự cho các mạng tĩnh chẳng hạn như các mạng siêu khối lập phương, lưới và cây.
11. Determine the condition under which a binary tree of height h has a larger diameter and larger number of links than each of the followings: (a) an n-dimensional hypercube, (b) an r x r 2D mesh with r — \N, and (c) a k-ary n-cube.	Xác định điều kiện cần thiết để một cây nhị phân chiều cao h có đường kính lớn hơn và số liên kết lớn hơn mỗi đối tượng sau đây: (a) 1 siêu lập phương n chiều, (b) Một lưới 2D rxr với r = VN, và khối lập phương n chiều k-ary
12. What are the minimum and the maximum distances a message has to travel in an n-dimensional hypercube? Can you use such information to compute the average distance a message has to travel in such cube? Show how?	Khoảng cách tối thiểu và tối đa mà một thông điệp phải di chuyển trong một siêu lập phương n chiều là bao nhiêu? Chúng ta có thể sử dụng thông tin đó để tính khoảng cách trung bình mà một thông điệp phải si chuyển trong một siêu lập

	phương (siêu khối) đó? Trình bày cụ thể.
13. Repeat Problem 12 for the case of an r x r 2D mesh with r = VN.	Làm lại bài tập 12 cho trường hợp mạng lưới 2D rxr với r= VN
14. Repeat Problem 12 for the case of a binary tree whose height is h and assuming that all possible source/destination pairs are equally likely.	Làm lại bài tập 12 cho trường hợp một cây nhị phân có chiều cao là h và giả sử rằng tất cả các cặp nguồn / đích đều có khả năng như nhau.
15. Routing of messages between two nodes A and B in a binary tree has been described in general terms in Section 2.4 of this chapter. You are required to obtain a step-by-step algorithm for routing messages between any two nodes in a binary tree given the following information:	Sự định tuyến tin nhắn giữa hai nút A và B trong một cây nhị phân đã được mô tả một cách tổng quát trong mục 2.4 của chương này. Bạn được yêu cầu xây dựng một thuật toán từng bước để định tuyến thông điệp giữa bất kỳ hai nút trong một cây nhị phân, với các thông tin sau:
(a) the root node is numbered as 1 and is considered at level 1;	Nút gốc được đánh số 1 và được coi là ở mức 1;
(b) the left and right nodes of a node whose number is x are respectively 2x and 2x + 1;	Các nút bên trái và bên phải của một nút có số là x lần lượt là 2x và 2x + 1
(c) the binary representation of the numbers of nodes at level i are i bits long; and	biểu diễn nhị phân của số nút ở cấp i dài i bit, và
(d) the left and right children of a node are having a 0 or a 1 appended to their parent's number, respectively.	Nhánh con bên trái và bên phải của một nút đang có số 0 hoặc 1 nối vào số nhánh cha tương ứng của chúng,
Show how to apply your algorithm to route messages between node number 8 and node number 13 in a 4 level binary tree.	Chỉ rõ cách thức áp dụng thuật toán của bạn để định tuyến tin giữa nút số 8 và số 13 trong một cây nhị phân 4 cấp.
....	....
1. Consider the case of a multiple-bus system consisting of 50	Xét trường hợp hệ thống nhiều bus bao

processors, 50 memory modules, and 10 buses. Assume that a processor generates a memory request with probability p in a given cycle. Compute the bandwidth of such system for p = 0.2, 0.5, and 1.0. Show also the effect on the bandwidth if the number of buses is increased to B = 20, 30, and 40 for the same request probability values.	gồm 50 bộ vi xử lý, 50 bộ nhớ và 10 bus. Giả sử rằng một bộ xử lý tạo ra một yêu cầu bộ nhớ với xác suất p trong một chu kỳ nhất định. Tính toán băng thông của hệ thống như vậy khi p = 0.2, 0.5, và 1.0. Đồng thời hãy chứng tỏ ảnh hưởng đến băng thông nếu số bus tăng lên B = 20, 30, và 40 đối với các giá trị xác suất yêu cầu như nhau .
11. Determine the condition under which a binary tree of height h has a larger diameter and larger number of links than each of the followings: (a) an n-dimensional hypercube, (b) an r x r 2D mesh with r — \N, and (c) a k-ary n-cube.	11.Xác định điều kiện để một cây nhị phân có chiều cao là h có đường kính và số lượng liên kết lớn hơn các đường kính và số lượng liên kết của các trường hợp sau đây :(a) mạng siêu lập phương n chiều, (b) một 2D r x r với r= VN, (c) một khối lập phương n chiều k phương.
12. What are the minimum and the maximum distances a message has to travel in an n-dimensional hypercube? Can you use such information to compute the average distance a message has to travel in such cube? Show how?	12.Khoảng cách lớn nhất và nhỏ nhất mà một tin nhắn phải di chuyển trong một siêu lập phương n chiều là bao nhiêu ? Có thể dùng thông tin vừa rồi để ước tính khoảng cách trung bình mà tin nhắn phải di chuyển trong siêu lập phương n chiều đó được không? Giải thích rõ cách thực hiện?
13. Repeat Problem 12 for the case of an r x r 2D mesh with r = VN.	13.Làm lại bài tập 12 đối với trường hợp một lưới 2D r x r với r=VN
14. Repeat Problem 12 for the case of a binary tree whose height is h and assuming that all possible source/destination pairs are equally likely.	14.Làm lại bài tập 12 đối với trường hợp một cây nhị phân có chiều cao là h và giả thiết rằng các cặp nguồn/đích đến đều có khả năng như nhau.

15. Routing of messages between two nodes A and B in a binary tree has been described in general terms in Section 2.4 of this chapter. You are required to obtain a step-by-step algorithm for routing messages between any two nodes in a binary tree given the following information:	15.Quá trình định tuyến tin giữa hai nút A&B trong một cây nhị phân đã được mô tả trong phần Thuật ngữ chung ở Phần 2.4 của chương này. Yêu cầu bạn tạo ra thuật toán từng bước một để định tuyến tin giữa hai nút trong một cây nhị phân với các thông tin cho trước sau đây:
(a) the root node is numbered as 1 and is considered at level 1; (b) the left and right nodes of a node whose number is x are respectively 2x and 2x + 1; (c) the binary representation of the numbers of nodes at level i are i bits long; and (d) the left and right children of a node are having a 0 or a 1 appended to their parent's number, respectively.	(a) Nút gốc được đánh số là 1 và được coi là cấp 1. (b) Nút trái và nút phải của một nút ~~mà~~ được đánh số là x lần lượt là 2x và 2x+1. (c) Biểu diễn nhị phân của số nút ở cấp i dài I bit; và (d)Nút con trái và nút con phải của một nút được lần lượt cộng thêm một số 0 và 1 vào số nút bố (mẹ) chúng.
Show how to apply your algorithm to route messages between node number 8 and node number 13 in a 4 level binary tree.	Trình bày rõ cách ứng dụng thuật toán của bạn vào việc truyền thông tin giữa nút số 8 và nút số 13 ở một cây nhị phân 4 mức.
…… …… ……	
1. Consider the case of a multiple-bus system consisting of 50 processors, 50 memory modules, and 10 buses. Assume that a processor generates a memory request with probability p in a given cycle. Compute the bandwidth of such system for p = 0.2, 0.5, and 1.0. Show also the effect on the	1.Xét trường hợp một hệ thống nhiều bus gồm 50 bộ xử lý, 50 mô-đun bộ nhớ, và 50 bus. Giả sử rằng một bộ xử lý tạo ra một yêu cầu bộ nhớ với xác suất p trong một chu trình cho trước. Ước tính băng thông của hệ thống này khi p= 0.2, 0.5, 1.0. Ngoài ra, chứng tỏ ảnh hưởng đến băng thông nếu số bus được tăng lên đến

bandwidth if the number of buses is increased to B = 20, 30, and 40 for the same request probability values.	mức B=20, 30 và 40 với các giá trị xác suất yêu cầu như trên.
2. In deriving the expression for the bandwidth of a crossbar system, we have assumed that all processors generate requests for memory modules during a given cycle. Derive a similar expression for the case whereby only a fraction of processors, f, generate requests during a given cycle. Consider the two cases whereby a processor generates a memory request with probability p in a given cycle and whereby a processor can request any memory module.	2. Trong quá trình rút ra biểu thức băng thông của hệ thống crossbar, ta coi mọi bộ xử lý đều tạo ra các yêu cầu tới các môđun bộ nhớ trong một chu trình nhất định. Rút ra biểu thức tương tự cho trường hợp trong đó chỉ một phần bộ xử lý f tạo ra các yêu cầu trong một chu trình nhất định. Xét hai trường hợp- khi một bộ xử lý tạo ra một yêu cầu bộ nhớ với xác suất p trong một chu trình nhất định và khi một bộ xử lý có thể yêu cầu bất kỳ môđun bộ nhớ nào.
3. Consider the recursive expression developed for the bandwidth of a Delta MIN network consisting of stages of a x b crossbar switches. Assuming that a = 2, b = 4, and ra = 0.5, compute the bandwidth of such a network.	3. Xét biểu thức hồi quy của băng thông đối với mạng Delta MIN bao gồm các tầng có các bộ chuyển mạch thanh ngang a x b. Giả sử a=2, b=4 và ra=5, tính băng thông của mạng này.
4. Consider the case of a binary n-cube having N nodes. Compute the bandwidth of such a cube given that p is the probability that a node receives an external request and n is the probability that a node generates a request (either internally or passes on an external request). Assume that a fraction f of the external requests received by a node is passed onwards to another node.	4. Xét trường hợp mạng lập phương n chiều nhị phân có N nút. Tính băng thông của mạng lập phương này khi p là xác suất nút mạng nhận yêu cầu bên ngoài và n là xác suất nút mạng tạo yêu cầu (cả truyền nội bộ lẫn truyền sang yêu cầu bên ngoài). Giả sử một phần yêu cầu bên ngoài f do nút mạng này nhận được và truyền sang nút mạng khác.
5. Consider the expressions obtained for efficiency under the two	5. Xét các biểu thức hiệu suất thu được theo hai mô hình điện toán nêu ở

computational models presented in the chapter. Compute the expected efficiency values for different values of tc and ts .	trong chương này. Tính các giá trị hiệu suất kỳ vọng đối với các giá trị tc và ts khác nhau.
6. Starting from the equation for the speedup factor given by S(n) = 1	6. Bắt đầu từ phương trình hệ số tăng tốc S(n)=1
show the inequality that relates the fraction of serial computation, f, and the number of processors employed, n, if a 50% efficiency is to be achieved.	Thiết lập một hệ thức biểu diễn mối quan hệ giữa một phần thuật toán chuỗi f và số lượng bộ xử lý được sử dụng n nếu hiệu suất đạt 50%.
7. Contrast the following two approaches for building a parallel system. In this first approach, a small number of powerful processors is used in which each processor is capable of performing serial computations at a given rate, C. In the second approach, a large number of simple processors are used in which each processor is capable of performing serial computations at a lower rate, F < C. What is the condition under which the second system will execute a given computation more slowly than a single processor of the first system?	7. Đối chiếu (so sánh) hai pháp xây dựng hệ thống song song sau đây. Trong phương thức đầu tiên, một số lượng nhỏ bộ xử lý mạnh được sử dụng trong đó mỗi bộ xử lý có thể thực hiện các phép toán nối tiếp với một tốc độ nhất định là C. Trong phương thức thứ hai, một số lượng lớn bộ xử lý đơn giản được sử dụng trong đó mỗi bộ xử lý có thể thực hiện các phép toán nối tiếp với tốc độ thấp hơn F<C. Trong trường hợp nào hệ thống thứ hai sẽ thực hiện một phép toán cho trước chậm hơn một bộ xử lý đơn trong hệ thống đầu tiên?
8. Consider a parallel architecture built using processors each capable of sustaining 0.5 megaflop. Consider a supercomputer capable of sustaining 100 megaflops. What is the condition (in terms of f) under which the parallel architecture can exceed the performance of the supercomputer?	8. Xét một kiến trúc song song sử dụng các bộ xử lý mà mỗi bộ xử lý có khả năng đạt tốc độ 0,5 megaflop. Xét một siêu máy tính có khả năng đạt tốc độ 100 megaflop. Trong điều kiện nào (theo f) kiến trúc song song song này có thể có hiệu suất cao hơn hiệu suất của siêu máy

	tính.
9. Consider an algorithm in which (1/a) th of the time is spent executing computations that must be done in a serial fashion. What is the maximum speedup achievable by a parallel form of the algorithm?	9. Xét một thuật toán trong đó (1/a) thời gian dùng thực hiện phép toán theo phương thức nối tiếp. Sự tăng tốc tối đa đạt được theo phương thức song song của thuật toán là bao nhiêu?
10. Show that the lower bound on the isoefficiency function of a parallel system is given by Q(n). Hint: If the problem size m grows at a rate slower than Q(n) as the number of processors increases, then the number of processors can exceed the problem size m.	10. Chứng minh cận dưới của hàm đẳng suất của hệ thống song song phụ thuộc vào Q (n). Gợi ý: Nếu ksich thước bài toán m tăng với tốc độ chậm hơn Q (n) khi số bộ xử lý tăng thì số bộ xử lý có thể vượt quá kích thướt bài toán m.
11. Compute the isoefficiency of a parallel system having an overhead Toh = n4/3 + m3/4 x n3/2.	11. Tính đẳng suất của hệ thống song song có tổng chi phí Toh= n4/3 + m3/4 x n3/2.
12. In addition to the two definitions offered in Section 3.4, one can also define the average parallelism, Q, as the intersection point of the hardware bound and the software bound on speedup. Show that the three definitions are equivalent. …… …… …	12. Ngoài hai định nghĩa nêu trong Mục 3.4, chúng ta cũng có thể định nghĩa điểm song song trung bình Q là giao điểm giữa biên phần cứng và biên phần mềm khi tăng tốc. Điều này cho thấy 3 định nghĩa tương đương nhau. …… …… …
1. Explain mutual exclusion and its relation to the cache coherence problem.	1. Giải thích hiện tượng loại trừ lẫn nhau và mối liên quan giữa nó với vấn đề tương hợp cache.
2. Discuss the advantages and disadvantages of using the following interconnection networks in the design of a shared memory system.	2. Thảo luận ưu và nhược điểm khi sử cụng các mạng liên thông sau trong thiết kế hệ thống bộ nhớ dùng chung

(a) Bus;
(b) Crossbar switch;
(c) Multistage network.
Kết thúc phần dịch mẫu của Nguyễn Hương Thơm

3. Some machines provide special hardware instructions that allows one to swap the contents of two words in one memory cycle. Show how the swap instruction can be used to implement mutual exclusion.
4. Consider a bus-based shared memory multiprocessor system. It is constructed using processors with speed of 106 instructions/s, and a bus with a peak bandwidth of 105 fetches/s. The caches are designed to support a hit rate of 90%.
(a) What is the maximum number of processors that can be supported by this system?
(b) What hit rate is needed to support a 20-processor system?
5. Determine the maximum speedup of a single-bus multiprocessor system having N processors if each processor uses the bus for a fraction f of every cycle.

6. Consider the two tasks T0 and T1 that are executed in parallel on processors P1 and P2 , respectively, in a shared memory system. Assume that the print statement is uninterruptible, and A, B, C, D are initialized to 0.
To Ti
A = 1;C = 3;
B = 2;D = 4;
Print A, D; Print B, C;

(a) Bus

(b) Bộ chuyển mạch thanh ngang;

(c) Mạng nhiều tầng.

3. Một số loại máy có hướng dẫn sử dụng phần cứng đặc biệt cho phép hoán đổi nội dung của hai từ trong cùng một chu kỳ bộ nhớ. Hãy trình bày cách sử dụng hướng dẫn hoán đổi nhằm thực hiện phương pháp loại trừ lẫn nhau.
4. Xem một hệ đa xử lý với bộ nhớ dùng chung dựa trên bus. Nó được hình thành trên cơ sở các bộ xử lý với tốc độ 106 lệnh/s và bus với băng thông tối đa 105 fetch/s. Thiết kế bộ nhớ cache sẽ hỗ trợ hit ra lên đến 90 %.
(a) Hãy cho biết hệ thống này có thể hỗ trợ tối đa bao nhiêu bộ xử lý?
(b) Cần hit rate bằng bao nhiêu để hỗ trợ một hệ thống với 20 bộ xử lý?
5. Trong trường hợp mỗi một bộ xử lý sử dụng bus cho một phân số f của mọi chu kỳ, hãy xác định độ tăng tốc tối đa của một hệ đa xử lý dạng bus đơn (single-bus) có N bộ xử lý.

6. Xét 2 tác vụ T0 và T1 chạy song song trên các bộ xử lý P1 và P2 trong cùng 1 hệ thống bộ nhớ dùng chung. Giả sử rằng phát biểu in không thể dừng và A, B, C, D đều bắt đầu từ 0.
To Ti

A = 1; C = 3;

B = 2; D = 4;

	In A và D; in B và C;
Show four different possible outputs of the parallel execution of these two tasks.	Hãy trình bày 4 đầu ra khác nhau có thể xảy ra trong trường hợp chạy song song 2 tác vụ này.
7. Consider a bus-based shared memory system consisting of three processors. The shared memory is divided into four blocks x, y, z, w. Each processor has a cache that can fit only one block at any given time. Each block can be in one of two states: valid (V) or invalid (I). Assume that caches are initially flushed (empty) and that the contents of the memory are as follows: Memory block x y z w Contents 10 30 80 20	7. Xét 1 hệ thống bộ nhớ dùng chung dạng bus gồm có 3 bộ xử lý. Chia bộ nhớ này thành 4 khối x, y, z, w. Mỗi bộ xử lý đều có 1 cache riêng chỉ có thể tương thích với 1 khối tại bất kỳ thời điểm nhất định. Mỗi khối có thể ở 1 trong 2 trạng thái: Valid và Invalid. Giả sử các cache lúc ban đầu đều rỗng và bộ nhớ có nội dung như sau: Khối bộ nhớ x y z w Nội dung 10 30 80 20
Consider, the following sequence of memory access events given in order: Kết thúc phần dịch mẫu của Minh Hiếu i) Pi: Read(x), 2) P2: Read(x), 3) P3: Read(x), 4) Pi: x = x + 25, 5) Pi: Read(z), 6) P2: Read(x), 7) P3:x = 15, 8) Pi: z = z + 1	Xét trình tự sự kiện truy cập bộ nhớ sau theo thứ tự: i) Pi: Đọc(x), 2) P2: Đọc(x), 3) P3: Đọc(x), 4) Pi: x = x + 25, 5) Pi: Đọc(z), 6) P2: Đọc(x), 7) P3:x = 15, 8) Pi: z = z + 1
Show the contents of the caches and memory and the state of cache blocks after each of the above operations in the following cases: (1) write-through and write-invalidate and (2) write-back and write-invalidate.	cho biết nội dung của bộ nhớ cache và trạng thái của khối cache mỗi phép toán ở trên trong các trường hợp sau: (1) write-through và write-invalidate và (2) write-back và write-invalidate.
8. Repeat Problem 7 assuming the following: (a) Each processor has a cache that has four block frames labeled 0, 1, 2, 3. The shared memory is divided into eight blocks 0, 1,... ,7. Assume that the contents	8. Làm lại bài tập 7 với giả thuyết sau: (a) Mỗi bộ vi xử lý có một bộ nhớ cache gồm 4 khung dữ liệu khối được đánh số 0, 1, 2, 3. Bộ nhớ chung được chia thành

of the shared memory are as follows: Block number 01234567 Contents 10 30 80 20 70 60 50 40	8 khối từ 0,1,…, 7. Giả định rằng nội dung của bộ nhớ chung như sau: Số khối 01234567 Nội dung 10 30 80 20 70 60 50 40
(b) To maintain cache coherence, the system uses the write-once protocol.	(b) Để đảm bảo tính tương hợp cache, hệ thống sử dụng giao thức ghi một lần (write-once)
(c) Memory access events are as follows: 1) Pi: Read(0), 2) P2: Read(0), 3) P3: Read(0), 4) P2: Read(2), 5) Pi: Write (i5 in 0), 6) P3: Read (2), 7) Pi: Write(25 in 0), 8) Pi: Read(2), 9) P3: Write (85 in 2), i0) P2: Read(7), ii) P3: Read (7), i2) Pi: Read(7)	(c) Các lượt (sự kiện) truy cập bộ nhớ như sau: 1) Pi: Đọc (0), 2) P2: Đọc (0), 3) P3: Đọc (0), 4) P2: Đọc (2), 5) Pi: Đọc (i5 trong 0), 6) P3: Đọc (2), 7) Pi: Ghi (25 trong 0), 8) Pi: Đọc (2), 9) P3: Ghi (85 trong 2), i0) P2: Đọc (7)
(Note that Write(x in i) means the value x is written in block i.) …… …… … 1. What are the differences between the functions pvm_initsend() and pvm_mkbuf()? 2. Discuss some situations in which nonblocking receive is preferred over blocking receive? 3. Consider the precedence constraints in Figure 8.10 among the tasks T0, T1, T2, T3, T4. Note that an arc from T; to T, implies that T; must be completed before Tj can start. Show how to enforce these precedence in PVM.	Lưu ý: Ghi (x trong i) tức là giá trị x được ghi trong khối i 1. Nêu điểm khác biệt giữa hàm pvm_initsend() và pvm_mkbuf()? 2. Thảo luận một số tình huống sử dụng chức năng nhận không chờ tốt hơn nhận chờ 3. Xem xét những ràng buộc ưu tiên (tiền đề) tại mục 8.10 trong các tác vụ T0, T1, T2, T3, T4. Chú ý rằng một arc từ T; đến T đồng nghĩa với việc T phải được hoàn tất trước khi Tj khởi động. Chỉ ra

	cách để áp đặt ưu tiên (tiền đề, điều kiện tiên quyết) này trong PVM.
i) Pi: Read(x), 2) P2: Read(x), 3) P3: Read(x), 4) Pi: x = x + 25, 5) Pi: Read(z), 6) P2: Read(x), 7) P3:x = 15, 8) Pi: z = z + 1	i) Pi: Đọc (x), 2) sP2: Đọc (x), 3) P3: Đọc (x), 4) Pi: x = x + 25, 5) Pi: Đọc (z), 6) P2: đọc (x), 7) P3: x = 15, 8) Pi: z = z + 1
Show the contents of the caches and memory and the state of cache blocks after each of the above operations in the following cases: (1) write-through and write-invalidate and (2) write-back and write-invalidate. 8. Repeat Problem 7 assuming the following: (a) Each processor has a cache that has four block frames labeled 0, 1, 2, 3. The shared memory is divided into eight blocks 0, 1,... ,7. Assume that the contents of the shared memory are as follows: Block number 01234567 Contents 10 30 80 20 70 60 50 40	Hiển thị các nội dung của bộ nhớ cache và bộ nhớ và trạng thái của các khối bộ nhớ cache sau mỗi phép toán trên trong các trường hợp sau đây: (1) write-through và write-invalidate and (2) write-back và write-invalidate. 8. Làm lại bài toán 7 với giả thuyết như sau: (a) Mỗi bộ vi xử lý có một bộ nhớ cache với bốn khung khối có nhãn 0, 1, 2, 3. Bộ nhớ dùng chung được chia thành tám khối 0, 1, ... , 7. Giả sử rằng rằng các nội dung của bộ nhớ dùng chung như sau: Mã số khối 01234567 Nội dung 10 30 80 20 70 60 50 40
(b) To maintain cache coherence, the system uses the write-once protocol.	(b) Để duy trì sự tương hợp cache, hệ thống sử dụng giao thức ghi một lần.
(c) Memory access events are as follows: 1) Pi: Read(0), 2) P2: Read(0), 3) P3: Read(0), 4) P2: Read(2), 5) Pi: Write (i5 in 0), 6) P3: Read (2), 7) Pi: Write(25 in 0), 8) Pi: Read(2), 9) P3: Write (85 in 2), i0) P2: Read(7), ii) P3: Read (7), i2) Pi: Read(7)	(c) Các sự kiện truy cập bộ nhớ như sau: 1) Pi: Đọc (0), 2) P2: Đọc (0), 3) P3: Đọc (0), 4) P2: đọc (2), 5) Pi: Viết (i5 trong 0), 6) P3: đọc (2), 7) Pi: Viết (25 0), 8) Pi: đọc (2), 9) P3: Viết (85 2), i0) P2: Đọc (7), ii) P3: Đọc (7), i2) Pi: Đọc (7)

(Note that Write(x in i) means the value x is written in block i.) …… …… … 1. What are the differences between the functions pvm_initsend() and pvm_mkbuf()? 2. Discuss some situations in which nonblocking receive is preferred over blocking receive? 3. Consider the precedence constraints in Figure 8.10 among the tasks T0, T1, T2, T3, T4. Note that an arc from T; to T, implies that T; must be completed before Tj can start. Show how to enforce these precedence in PVM.	(Lưu ý rằng Write(x in i) có nghĩa là giá trị x được viết trong khối i.) .... 1. Sự khác biệt giữa các hàm pvm_initsend () và pvm_mkbuf () là gì? 2. Thảo luận về một số trường hợp trong đó nhận không chờ chiếm ưu thế hơn so với nhận chờ? 3. Xét các ràng buộc ưu tiên trong hình 8.10 trong các tác vụ T0, T1, T2, T3, T4. Lưu ý rằng một cung từ T, T, T có nghĩa là T; phải hoàn thành trước Tj bắt đầu. Chứng tỏ cách thức để áp đặt những điều kiện tiên quyết này trong PVM.
4. Consider the four tasks in Figure 8.11, which are synchronized using barriers corresponding to the synchronization points shown. Show how to implement the given barrier structure in PVM. Figure 8.10 Precedence constraints for Problem 3. 5. Suppose that we want to extend PVM to support fully synchronized communication among processes. What parts of PVM should be altered to provide a fully synchronous send operation? Discuss all possible methods to achieve this goal. 6. Suppose that you were hired to develop techniques for assigning tasks to machines in a PVM environment. What performance measures should you optimize? What parameters should be	4.Xét bốn tác vụ trong hình 8.11, chúng được đồng bộ hóa bằng các hàng rào tương ứng với các điểm đồng bộ hóa được biểu diễn. Chứng tỏ cách thực thi cấu trúc hàng rào như thế trong PVM. Hình 8.10 Các ràng buộc ưu tiên đối với Bài toán 3. 5.Giả sử chúng ta muốn mở rộng PVM để hỗ trợ sự truyền thông được đồng bộ hóa hoàn toàn giữa các quá trình. Phần nào của PVM nên được thay thế để cho ra một hoạt động (thao tác, phép toán) gửi hoàn toàn đồng bộ? Thảo luận một số phương pháp tiềm năng để đạt được mục tiêu này. 6.Giả sử bạn được thuê xây dựng các kỹ thuật để ấn định các tác vụ cho các máy trong môi trường PVM. Bạn nên tối ưu tham số hiệu suất nào? Các tham số nào

considered? Should the assignment be done statically or dynamically? Why? 7. Devise a static algorithm for task allocation that can be used to schedule a PVM application on a given virtual machine. Devise another dynamic method to balance the load among the PVM hosts. 8. A task can be partitioned at different levels of granularity: fine-grain, medium-grain, and large-grain. Which level of granularity fits the PVM programming approach the most? Justify your answer. 9. Develop a matrix multiplication program in PVM. This program multiplies two n x n matrices in parallel (C = A x B). The program consists of a supervisor and n — 1 workers. The supervisor sends each worker one row of the first matrix and the entire second matrix. Each worker calculates one row in the resulting matrix and sends it to the supervisor. 10. Rewrite the program of Problem 9 such that each task calculates (a) exactly one cell in the matrix C, (b) part of a row in C, (c) more than one row in C. Contrast all the methods. Discuss the advantages and disadvantages of each method. Kết thúc phần dịch mẫu của Thoại Ân	cần xem xét? Việc phân công nên được thực hiện tĩnh hay động? Tại sao? 7. Hãy nghĩ ra một thuật toán tĩnh để phân công tác vụ có thể được dùng để xây dựng một ứng dụng PVM trên một máy ảo nhất định. Nghĩ ra một phương pháp động khác để cân bằng tải giữa các host PVM. 9.Một tác vụ có thể được phân vùng ở các mức hạt khác nhau: hạt mịn, hạt trung bình, hạt lớn. Mức hạt nào khớp với mô hình lập trình PVM nhất? Chứng minh câu trả lời của bạn. 9.Xây dựng một chường trình nhân ma trận trong PVM. Chương trình này nhân hai ma trận nxn song song (C = A x B). Chương trình bao gồm một giám sát viên và n-1 người thi hành (công nhân). Giám sát viên gửi cho mỗi công nhân một hàng của ma trận đầu tiên và toàn bộ ma trận thứ hai. Mỗi người thi hành (công nhân) tính một hàng trong ma trận cuối cùng và gửi nó cho giám sát viên. 10.Viết lại chương trình của Bài toán 9 sao cho mỗi tác vụ tính (a) đúng một ô trong ma trận C, (b) một phần của một hàng trong C, (c) hơn một hàng trong C. Phân biệt tất cả các phương pháp. Thảo luận ưu và nhược điểm của mỗi phương pháp.
