

ADVANCED COMPUTER ARCHITECTURE

PARALLELISM

SCALABILITY

PROGRAMMABILITY



TATA McGRAW-HILL
EDITION

Limited preview ! Not for commercial use



Tata McGraw-Hill

ADVANCED COMPUTER ARCHITECTURE

Parallelism. Scalability. Programmability

Copyright C 2000 by The McGraw-Hill Companies, Inc.

All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication

Tata McGraw-Hill Edition 2001

Eighteenth reprint 2008

RACYCDRXRBZQD

Reprinted in India by arrangement with The McGraw-Hill Companies Inc.,
New York

Sales territories: India, Pakistan, Nepal, Bangladesh, Sri Lanka and Bhutan

Library of Congress Cataloging-in-Publication Data

Hwang, Kai, 1943 -

Advanced Computer Architecture: Parallelism, Scalability, Programmability/

Kai Hwang

p cm. -(McGraw-Hill computer science series. Computer organization and architecture. Network parallel and distributed computing. McGraw-Hill computer engineering series)

Includes bibliographical references (p.) and index.

ISBN 0-07-031622-8

I. Computer Architecture I. Title II Series

QA76.9.A7H87 1991

004135-dc20

92-44944

ISBN-13: 978-0-07-053070-6

ISBN-10: 0-07-053070-X

Published by Tata McGraw-Hill Publishing Company Limited.
7 West Patel Nagar, New Delhi 110 008, and printed at
Gopaljee Enterprises, Delhi 110 053

The McGraw-Hill Companies

to-

Contents

<u>Foreword</u>	xvii
<u>Preface</u>	xix
<u>PART I THEORY OF PARALLELISM</u>	1
<u>Chapter 1 Parallel Computer Models</u>	3
<u>1.1 The State of Computing</u>	3
<u>1.1.1 Computer Development Milestones</u>	3
<u>1.1.2 Elements of Modern Computers</u>	6
<u>1.1.3 Evolution of Computer Architecture</u>	9
<u>1.1.4 System Attributes to Performance</u>	14
<u>1.2 Multiprocessors and Multicomputers</u>	19
<u>1.2.1 Shared-Memory Multiprocessors</u>	19
<u>1.2.2 Distributed-Memory Multicomputers</u>	24
<u>1.2.3 A Taxonomy of MIMD Computers</u>	27
<u>1.3 Multivector and SIMD Computers</u>	27
<u>1.3.1 Vector Supercomputers</u>	27
<u>1.3.2 SIMD Supercomputers</u>	30
<u>1.4 PRAM and VLSI Models</u>	32
<u>1.4.1 Parallel Random-Access Machines</u>	33
<u>1.4.2 VLSI Complexity Model</u>	38
<u>1.5 Architectural Development Tracks</u>	41
<u>1.5.1 Multiple-Processor Tracks</u>	41
<u>1.5.2 Multivector and SIMD Tracks</u>	43
<u>1.5.3 Multithreaded and Dataflow Tracks</u>	44
<u>1.6 Bibliographic Notes and Exercises</u>	45

Chapter 2 Program and Network Properties	51
2.1 Conditions of Parallelism	51
2.1.1 Data and Resource Dependences	51
2.1.2 Hardware and Software Parallelism	57
2.1.3 The Role of Compilers	60
2.2 Program Partitioning and Scheduling	61
2.2.1 Grain Sizes and Latency	61
2.2.2 Grain Packing and Scheduling	64
2.2.3 Static Multiprocessor Scheduling	67
2.3 Program Flow Mechanisms	70
2.3.1 Control Flow Versus Data Flow	71
2.3.2 Demand-Driven Mechanisms	74
2.3.3 Comparison of Flow Mechanisms	75
2.4 System Interconnect Architectures	76
2.4.1 Network Properties and Routing	77
2.4.2 Static Connection Networks	fiO
2.4.3 Dynamic Connection Networks	89
2.5 Bibliographic Notes and Exercises	96
Chapter 3 Principles of Scalable Performance	105
3.1 Performance Metrics and Measures	105
3.1.1 Parallelism Profile in Programs	105
3.1.2 Harmonic Mean Performance	108
3.1.3 Efficiency, Utilization, and Quality	112
3.1.4 Standard Performance Measures	115
3.2 Parallel Processing Applications	118
3.2.1 Massive Parallelism for Grand Challenges	118
3.2.2 Application Models of Parallel Computers	122
3.2.3 Scalability of Parallel Algorithms	125
3.3 Speedup Performance Laws	129
3.3.1 Amdahl's Law for a Fixed Workload	129
3.3.2 Gustafson's Law for Scaled Problems	131
3.3.3 Memory-Bounded Speedup Model	134
3.4 Scalability Analysis and Approaches	138
3.4.1 Scalability Metrics and Goals	138
3.4.2 Evolution of Scalable Computers	143
3.4.3 Research Issues and Solutions	147
3.5 Bibliographic Notes and Exercises	149

Chapter 4 Processors and Memory Hierarchy	157
4.1 Advanced Processor Technology	157
4.1.1 Design Space of Processors	157
4.1.2 Instruction-Set Architectures	162
4.1.3 CISC Scalar Processors	165
4.1.4 RISC Scalar Processors	169
4.2 Superscalar and Vector Processors	177
4.2.1 Superscalar Processors	178
4.2.2 The VLIW Architecture	182
4.2.3 Vector and Symbolic Processors	184
4.3 Memory Hierarchy Technology	188
4.3.1 Hierarchical Memory Technology	188
4.3.2 Inclusion, Coherence, and Locality	190
4.3.3 Memory Capacity Planning	194
4.4 Virtual Memory Technology	196
4.4.1 Virtual Memory Models	196
4.4.2 TLB, Paging, and Segmentation	198
4.4.3 Memory Replacement Policies	205
4.5 Bibliographic Notes and Exercises	208
Chapter 5 Bus, Cache, and Shared Memory	213
5.1 Backplane Bus Systems	213
5.1.1 Backplane Bus Specification	213
5.1.2 Addressing and Timing Protocols	216
5.1.3 Arbitration, Transaction, and Interrupt	218
5.1.4 The IEEE Futurebus+ Standards	221
5.2 Cache Memory Organizations	224
5.2.1 Cache Addressing Models	225
5.2.2 Direct Mapping and Associative Caches	228
5.2.3 Set-Associative and Sector Caches	232
5.2.4 Cache Performance Issues	236
5.3 Shared-Memory Organizations	238
5.3.1 Interleaved Memory Organization	239
5.3.2 Bandwidth and Fault Tolerance	242
5.3.3 Memory Allocation Schemes	244
5.4 Sequential and Weak Consistency Models	248
5.4.1 Atomicity and Event Ordering	248
5.4.2 Sequential Consistency Model	252
5.4.3 Weak Consistency Models	253
5.5 Bibliographic Notes and Exercises	256

6.1	Linear Pipeline Processors	265
6.1.1	Asynchronous and Synchronous Models	265
6.1.2	Clocking and Timing Control	267
6.1.3	Speedup, Efficiency, and Throughput	268
6.2	Nonlinear Pipeline Processors	270
6.2.1	Reservation and Latency Analysis	270
6.2.2	Collision-Free Scheduling	274
6.2.3	Pipeline Schedule Optimization	276
6.3	Instruction Pipeline Design	280
6.3.1	Instruction Execution Phases	280
6.3.2	Mechanisms for Instruction Pipelining	283
6.3.3	Dynamic Instruction Scheduling	288
6.3.4	Branch Handling Techniques	291
6.4	Arithmetic Pipeline Design	297
6.4.1	Computer Arithmetic Principles	297
6.4.2	Static Arithmetic Pipelines	299
6.4.3	Multifunctional Arithmetic Pipelines	307
6.5	Superscalar and Superpipeline Design	308
6.5.1	Superscalar Pipeline Design	310
6.5.2	Superpipelined Design	316
6.5.3	Supersymmetry and Design Tradeoffs	320
6.6	Bibliographic Notes and Exercises	322

PART HI PARALLEL AND SCALABLE ARCHITECTURES 329

Chapter 7	Multiprocessors and Multicomputers	331
7.1	Multiprocessor System Interconnects	331
7.1.1	Hierarchical Bus Systems	333
7.1.2	Crossbar Switch and Multiport Memory	336
7.1.3	Multistage and Combining Networks	341
7.2	Cache Coherence and Synchronization Mechanisms	348
7.2.1	The Cache Coherence Problem	348
7.2.2	Snoopy Bus Protocols	351
7.2.3	Directory-Based Protocols	358
7.2.4	Hardware Synchronization Mechanisms	364
7.3	Three Generations of Multicomputers	368
7.3.1	Design Choices in the Past	368
7.3.2	Present and Future Development	370
7.3.3	The Intel Paragon System	372
7.4	Message-Passing Mechanisms	375
7.4.1	Message-Routing Schemes	375

7.4.2 Deadlock and Virtual Channels	379
<u>7.4.3 Flow Control Strategies</u>	<u>383</u>
<u>7.4.4 Multicast Routing Algorithms</u>	<u>387</u>
7.5 Bibliographic Notes and Exercises	393
 Chapter 8 Multivector and SIMD Computers	 403
8.1 Vector Processing Principles	-103
<u>8.1.1 Vector Instruction Types</u>	<u>403</u>
<u>8.1.2 Vector-Access Memory Schemes</u>	<u>408</u>
<u>8.1.3 Past and Present Supercomputers</u>	<u>410</u>
8.2 Multivector Multiprocessors	415
8.2.1 Performance-Directed Design Rules	415
<u>8.2.2 Cray Y-MP, C-90, and MPP</u>	<u>419</u>
8.2.3 Fujitsu VP2000 and VPP500	425
<u>8.2.4 Mainframes and Minisupercomputers</u>	<u>429</u>
8.3 Compound Vector Processing	435
8.3.1 Compound Vector Operations	436
8.3.2 Vector Loops and Chaining	437
8.3.3 Multipipeline Networking	442
8.4 SIMD Computer Organizations	447
8.4.1 Implementation Models	447
ft 4.2 The CM-2 Architecture	<u>449</u>
fi.4.3 The MasPar MP-1 tohStflCtfflB	<u>453</u>
8.5 The Connection Machine CM-5	457
8.5.1 * A Synchronized MIMD Machine	457
8.5.2 The CM-5 Network Architecture	460
8.5.3 Control Processors and Processing Nodes	462
8.5.4 Interprocessor Communications	465
8.6 Bibliographic Notes and Exercises	468
 Chapter 9 Scalable, Multithreaded, and Dataflow Architectures	 475
9.1 Latency-Hiding Techniques	475
<u>9.1.1 Shared Virtual Memory</u>	<u>476</u>
9.1.2 Prefetching Techniques	480
9.1.3 Distributed Coherent Caches	482
9.1.4 Scalable Coherence Interface	483
<u>9.1.5 Relaxed Memory Consistency</u>	<u>486</u>
9.2 Principles of Multithreading	490
<u>9.2.1 Multithreading Issues and Solutions</u>	<u>490</u>
9.2.2 Multiple-Context Processors	495
9.2.3 Multidimensional Architectures	499
9.3 Fine-Grain Multicomputers	504

9.3.1	Fine-Grain Parallelism	505
9.3.2	The MIT J-Machine	506
9.3.3	The Caltech Mosaic (!)	514
9.4	Scalable and Multithreaded Architectures	516
9.4.1	The Stanford Dash Multiprocessor	516
9.4.2	The Kendall Square Research KSR-1	521
9.4.3	The Tera Multiprocessor System	524
9.5	Dataflow and Hybrid Architectures	531
9.5.1	The Evolution of Dataflow Computers	531
9.5.2	The ETL/EM-4 in Japan	534
9.5.3	The MIT/Motorola *T Prototype	536
9.6	Bibliographic Notes and Exercises	539

PART **TV** **SOFTWARE FOR PARALLEL PROGRAMMING** **545**

Chapter 10	Parallel Models, Languages, and Compilers	547
10.1	Parallel Programming Models	547
10.1.1	Shared-Variable Model	547
10.1.2	Message-Passing Model	551
10.1.3	Data-Parallel Model	554
10.1.4	Object-Oriented Model	556
10.1.5	Functional and Logic Models	559
10.2	Parallel Languages and Compilers	560
10.2.1	Language Features for Parallelism	560
10.2.2	Parallel Language Constructs	562
10.2.3	Optimizing Compilers for Parallelism	564
10.3	Dependence Analysis of Data Arrays	567
10.3.1	Iteration Space and Dependence Analysis	567
10.3.2	Subscript Separability and Partitioning	570
10.3.3	Categorized Dependence Tests	573
10.4	Code Optimization and Scheduling	578
10.4.1	Scalar Optimization with Basic Blocks	578
10.4.2	Local and Global Optimizations	581
10.4.3	Vectorization and Parallelization Methods	585
10.4.4	Code Generation and Scheduling	592
10.4.5	Trace Scheduling Compilation	596
10.5	Loop Parallelization and Pipelining	599
10.5.1	Loop Transformation Theory	599
10.5.2	Parallelization and Wavefronting	602
10.5.3	Tiling and Localization	605
10.54	Software Pipelining	610

10.6	Bibliographic Notes and Exercises	612
Chapter 11	Parallel Program Development and Environments	617
11.1	Parallel Programming Environments	617
11.1.1	Software Tools and Environments	617
11.1.2	<u>Y-MP, Paragon, and CM-5 Environments</u>	<u>621</u>
11.1.3	<u>Visualization and Performance Tuning</u>	<u>623</u>
11.2	Synchronization and Multiprocessing Modes	625
11.2.1	Principles of Synchronization	625
11.2.2	Multiprocessor Execution Modes	628
11.2.3	<u>Multitasking on Cray Multiprocessors</u>	<u>629</u>
11.3	Shared-Variable Program Structures	634
11.3.1	Locks for Protected Access	634
11.3.2	Semaphores and Applications	637
11.3.3	Monitors and Applications	640
11.4	Message-Passing Program Development	644
11.4.1	Distributing the Computation	644
11.4.2	Synchronous Message Passing	645
11.4.3	Asynchronous Message Passing	647
11.5	Mapping Programs onto Multicomputers	648
11.5.1	Domain Decomposition Techniques	648
11.5.2	Control Decomposition Techniques	652
11.5.3	<u>Heterogeneous Processing</u>	<u>, 656</u>
11.6	<u>Bibliographic Notes and Exercises</u>	<u>661</u>

Chapter 12 UNIX, Mach, and OSF/1 for Parallel Computers **667**

12.1	Multiprocessor UNIX Design Goals	667
12.1.1	<u>Conventional UNIX Limitations</u>	<u>668</u>
12.1.2	<u>Compatibility and Portability</u>	<u>670</u>
12.1.3	Address Space and Load Balancing	671
12.1.4	<u>Parallel I/O and Network Services</u>	<u>671</u>
12.2	2 Master-Slave and Multithreaded UNIX	fiI2
12.2.1	<u>Master-Slave Kernels</u>	<u>fiI2</u>
12.2.2	Floating-Executive Kernels	674
12.2.3	<u>Multithreaded UNIX Kernel</u>	<u>fiZS</u>
12.3	Multicomputer UNIX Extensions	683
12.3.1	<u>Message-Passing OS Models</u>	<u>683</u>
12.3.2	<u>Cosmic Environment and Reactive Kernel</u>	<u>683</u>
12.3.3	<u>Intel NX/2 Kernel and Extensions</u>	<u>685</u>
12.4	Mach/OS Kernel Architecture	686
12.4.1	<u>Mach/OS Kernel Functions</u>	<u>687</u>
12.4.2	<u>Multithreaded Multitasking</u>	<u>688</u>

<u>12.4.3</u>	<u>Message-Based Communications</u>	<u>694</u>
<u>12.4.4</u>	<u>Virtual Memory Management</u>	<u>697</u>
12.5	OSF/1 Architecture and Applications	701
12.5.1	The OSF/1 Architecture	702
<u>12.5.2</u>	<u>The OSF/1 Programming Environment</u>	<u>707</u>
12.5.3	Improving Performance with Threads	709
12.6	Bibliographic Notes and Exercises	712
<u>Bibliography</u>		<u>717</u>
<u>Index</u>		<u>739</u>
<u>Answers to Selected Problems</u>		<u>765</u>

Foreword

by Gordon Bell

Kai Hwang has introduced the issues in **designing** and using high **performance parallel** computers at a time when a plethora of scalable computers utilizing commodity microprocessors offer higher peak performance than traditional vector **supercomputers**. These new machines, their operating environments including the operating system and languages, and the programs to effectively utilize them are introducing more rapid changes for researchers, builders, and users than at any time in the history of computer structures.

For the first time since the introduction of Cray 1 vector processor in 1975, it may again be **necessary** to change and evolve the programming paradigm — provided that massively parallel computers can be shown to be **useful** outside of research on massive parallelism. Vector processors required modest data parallelism and these operations have been reflected either explicitly in Fortran programs or implicitly with the need to evolve Fortran (e.g., Fortran 90) to build in vector operations.

So far, the main line of **supercomputing** as measured by the usage (hours, jobs, number of programs, program portability) has been the shared memory, vector multi-processor as pioneered by Cray Research. Fujitsu, IBM, Hitachi, and NEC all produce computers of this type. In 1993- the Cray C90 supercomputer delivers a peak of 16 billion floating-point operations per second (a Gigaflops) with 16 processors and costs about \$30 million, providing roughly 500 floating-point operations per second per dollar.

In contrast, massively parallel computers introduced in the early 1990s are nearly all based on utilizing the same powerful, RISC-based, CMOS microprocessors that are used in workstations. These scalar processors provide a peak of ≈ 100 million floating-point operations per second and cost \$20 thousand, providing an order of magnitude more peak per dollar (5000 flops per dollar). Unfortunately, to obtain peak power requires large-scale problems that can require $O(n^3)$ operations over supers, and this significantly increases the running time when peak power is the goal.

The multicomputer approach interconnects computers built from microprocessors through **high-bandwidth** switches that introduce latency. Programs are written in either an evolved parallel data model utilizing Fortran or as independent programs that communicate by passing messages. The book describes a variety of **multicomputers** including Thinking Machines' CM5, the first computer announced that could reach a teraflops using 8K independent computer nodes, each of which can deliver 128 **Mflops** utilizing four **32-Mflops** floating-point units.

The architecture research trend is toward scalable, shared-memory multiprocessors in order to handle general workloads ranging from technical to commercial tasks and workloads, negate the need to explicitly pass messages for communication, and provide memory addressed accessing. KSR's scalable multiprocessor and Stanford's Dash prototype have proven that such machines are possible.

The author starts by positing a framework based on evolution that outlines the main approaches to designing computer structures. He covers both the scaling of computers and workloads, various multiprocessors, vector processing, multicomputers, and emerging scalable or multithreaded multiprocessors. The final three chapters describe parallel programming techniques and discuss the host operating environment necessary to utilize these new computers.

The book provides case studies of both industrial and research computers, including the Illinois Cedar, Intel Paragon, TMC CM-2, MasPar M1, TMC CM-5, Cray **Y-MP**, C-90, and Cray MPP, Fujitsu VP2000 and **VPP500**, NEC SX, Stanford Dash, **KSR-1**, MIT J-Machine, MIT *T, ETL **EM-4**, Caltech Mosaic C, and Tera Computer.

The book presents a balanced treatment of the theory, technology, architecture, and software of advanced **computer** systems. The emphasis on parallelism, scalability, and **programmability** makes this book rather unique and educational.

I highly recommend Dr. Hwang's timely book. I believe it will benefit many readers and be a fine reference.

C. Gordon Bell

Preface

The Aims

This book provides a comprehensive study of scalable and parallel computer architectures for achieving a proportional increase in performance with increasing system resources. System resources are scaled by the number of processors used, the memory capacity enlarged, the access latency tolerated, the I/O bandwidth required, the performance level desired, etc.

Scalable architectures delivering a sustained performance are desired in both sequential and parallel computers. Parallel architecture has a higher potential to deliver scalable performance. The scalability varies with different **architecture-algorithm** combinations. Both hardware and software issues need to be studied in building scalable computer systems.

It is my intent to put the reader in a position to design **scalable** computer systems. Scalability is defined in a broader sense to reflect the interplay among architectures, **algorithms**, software, and environments. The integration between hardware and software is emphasized for building cost-effective computers.

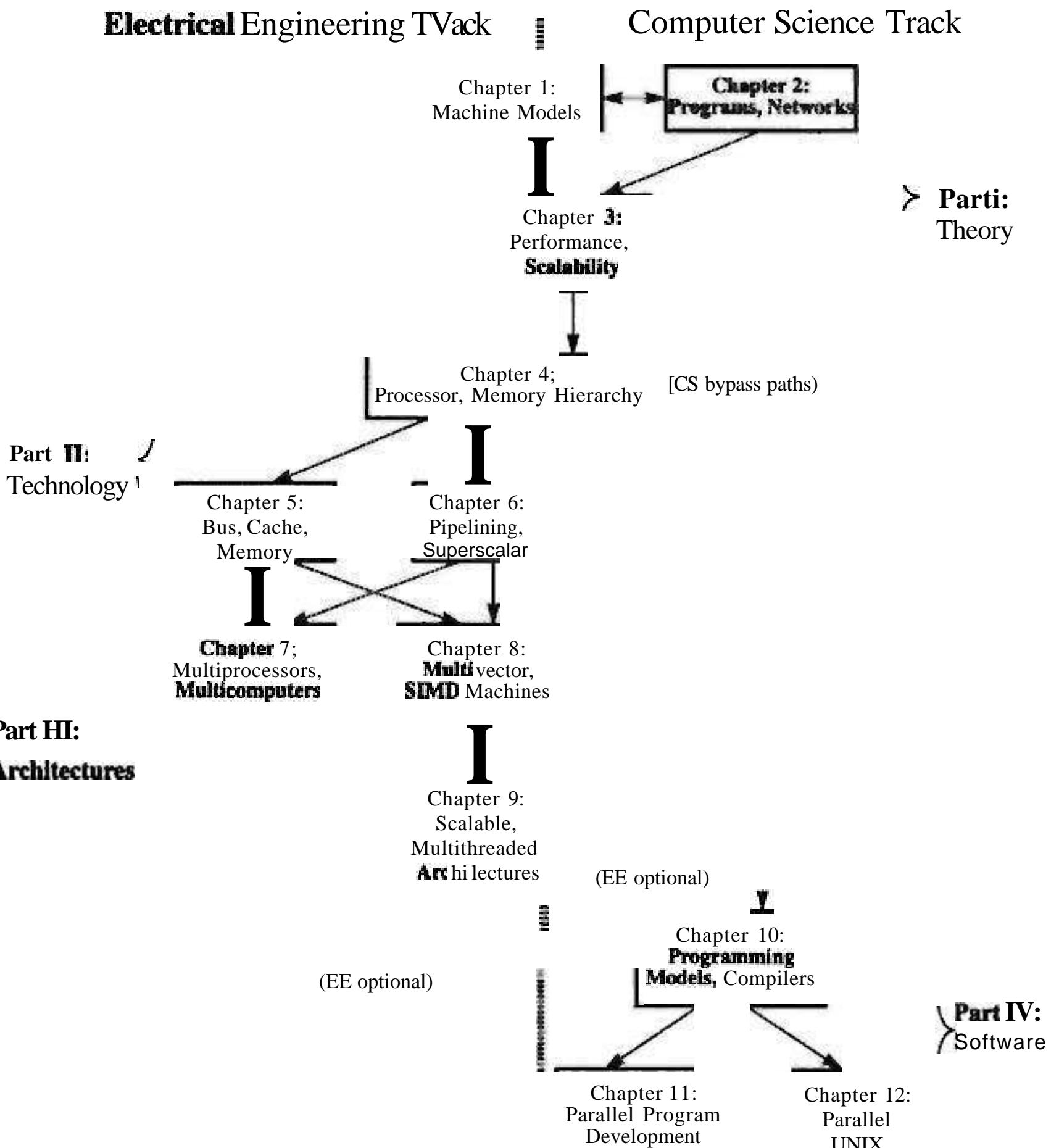
We should explore cutting-edge technologies in scalable parallel computing. Systems architecture is thus studied with **generality**, scalability, **programmability**, and **performability** in mind.

Since high technology changes so rapidly, I have presented the material in a generic manner, unbiased toward particular machine implementations. Representative processors and systems are presented only if they contain important features which may last into the future.

Every author faces the same dilemma in writing a **technology-dependent** book which may become obsolete quickly. To cope with the problem, frequent updates with newer editions become a necessity, and I plan to make revisions every few years in the future.

The Contents

This book consists of twelve chapters divided into four parts covering **theory**, **technology**, **architecture**, and **software** aspects of parallel and vector computers as shown in the flowchart:



Readers' Guide

Part I presents principles of parallel processing in three chapters. These include parallel computer **models**, scalability analysis, theory of parallelism, data dependences, program flow mechanisms, network topologies, benchmark measures, performance laws, and program behaviors. These chapters lay the necessary foundations for readers to study hardware and software in subsequent chapters.

In Part II, three chapters **are** devoted to studying advanced processors, cache and memory technology, and pipelining techniques. Technological bases touched include RISC, CISC, superscalar, superpipelining, and **VLIW** architectures. Shared memory, consistency models, cache architecture, and coherence protocols are studied.

Pipelining is extensively applied in memory-access, instruction execution, scalar, superscalar, and vector arithmetic operations. Instruction prefetch, data forwarding, software interlocking, scoreboarding, branch handling, and out-of-order issue and completion are studied for designing advanced processors.

In Part III, three chapters are provided to cover shared-memory multiprocessors, vector and SIMD supercomputers, **message-passing** multicomputers, and scalable or multithreaded architectures. Since we emphasize scalable architectures, special treatment is given to the IEEE **Futurebus+** standards, multistage networks, cache coherence, latency tolerance, fast synchronization, and hierarchical and multidimensional structures for building shared-memory systems.

Massive parallelism *is* addressed in message-passing systems as well as in synchronous SIMD computers. Shared virtual memory and multithreaded architectures are the important topics, in addition to compound vector processing on pipelined supercomputers and coordinated data parallelism on the **CM-5**.

Part IV consists of three chapters dealing with parallel programming models, multiprocessor UNIX, software environments, and compiler development for parallel/vector computers. Both shared variables and **message-passing** schemes are studied for interprocessor communications. Languages, compilers, and software tools for program and benchmark development and performance monitoring are studied.

Among various UNIX extensions, we discuss master-slave, floating-executive, and multithreaded kernels for resource management in a network of heterogeneous computer systems. The Mach/OS and **OSF/1** are studied as example systems.

This book has been completely newly written based on recent material. The contents are rather different from my earlier book coauthored with Dr. Faye Briggs in 1983. The two books, separated by 10 years, have very little in **common**.

The Audience

The material included in this text is an outgrowth of two **graduate-level** courses: Computer Systems Architecture (EE 557) and Parallel Processing (EE 657) that I have taught at the University of Southern California, the University of Minnesota (**Spring** 1989), and National Taiwan University (Fall 1991) during the last eight years.

The book can be adopted as a textbook in senior- or **graduate-level** courses offered by Computer Science (**CS**), Computer Engineering (CE), Electrical Engineering (EE), or Computational Science programs. The flowchart guides the students and instructors in reading this book.

The first four chapters should be taught to all disciplines. The three technology chapters are necessary for EE and CE students. The three architecture chapters can be selectively taught to CE and CS students, depending on the instructor's interest and the computing facilities available to teach the course. The three software chapters are written for CS students and are optional to EE **students**.

Five course outlines are suggested below for different audiences. The first three outlines are for **45-hour**, one-semester courses. The **last** two outlines are for **two-quarter** courses in a sequence.

- (1) For a Computer Science course on *Parallel Computers and Programming*, the minimum coverage should include Chapters 1-4, **7**, and 9-12.
- (2) For an exclusive Electrical Engineering course on *Advanced Computer Architecture*, the minimum coverage should include Chapters 1-9.
- (3) For a joint CS and EE **course** on *Parallel Processing Computer Systems*, the minimum coverage should include Chapters 1-4 and 7-12.
- (4) Chapters 1 through 6 can be taught to a senior or first-year graduate course under the title *Computer Architecture* in one quarter (10 weeks / 30 hours).
- (5) Chapters 7 through 12 can be taught to a graduate course on *Parallel Computer Architecture and Programming* in a **one-quarter** course with course (4) as the prerequisite.

Instructors may wish to include some advanced research topics treated in Sections **1.4**, **2.3**, 3.4, 5.4, **6.2**, 6.5, 7.2, 7.3, 8.3, 10.4, **11.1**, 12.5, and selected sections from Chapter 9 in each of the above course options. The architecture chapters present four different families of commercially available computers. Instructors may choose to teach a subset of these machine families based on the accessibility of corresponding machines on campus or via a public network. Students are encouraged to learn through hands-on programming experience on parallel computers.

A *Solutions Manual* is available to instructors only from the Computer Science Editor, College Division, McGraw-Hill, Inc., 1221 Avenue of the Americas, New York, NY 10020. Answers to a few selected exercise problems are given at the end of the book.

The Prerequisites

This is an advanced text on computer architecture and parallel programming. The reader should have been exposed to some basic computer organization and programming courses at the undergraduate level. Some of the required background material can be found in *Computer Architecture: A Quantitative Approach* by John Hennessy and David Patterson (Morgan Kaufman, 1990) or in *Machine and Assembly Language Programming* by Arthur Gill (Prentice-Hall, 1978).

Students should have some knowledge and experience in logic **design**, computer hardware, **system** operations, assembly languages, and Fortran or C programming. Because of the emphasis on scalable architectures and the exploitation of parallelism in practical applications, readers will find it useful to **have** some background in probability, discrete mathematics, matrix algebra, and optimization theory.

Acknowledgments

I have tried to identify all sources of information in the bibliographic notes. As the subject area evolves rapidly, omissions are almost unavoidable. I apologize to those whose valuable work has not been included in this edition. I am responsible for all omissions and for any errors found in the book. Readers are encouraged to contact me directly regarding error correction or suggestions for future editions.

The writing of this book was inspired, taught, or assisted by numerous scholars or specialists working in the area. I would like to thank each of them for intellectual exchanges, valuable suggestions, critical reviews, and technical assistance.

First of all, I want to thank a number of my former and current Ph.D. students. Hwang-Cheng Wang has assisted me in producing the entire manuscript in LATEX. Besides, he has coauthored the Solutions Manual with Jung-Gen Wu, who visited USC during 1992. Weihua Mao has drawn almost all the figure illustrations using FrameMaker, based on my original sketches. I want to thank D.K. Panda, Joydeep Ghosh, Ahmed Louri, Dongseung Kim, Zhi-Wei Xu, Sugih Jamin, Chien-Ming Cheng, Santosh Rao, Shisheng Shang, Jih-Cheng Liu, Scott Toborg, Stanley Wang, and Myungho Lee for their assistance in collecting material, proofreading, and contributing some of the homework problems. The errata from Teerapon Jungwiwattanaporn were also useful. The Index was compiled by H.C. Wang and J.G. Wu jointly.

I want to thank Gordon Bell for sharing his insights on supercomputing with me and for writing the Foreword to motivate my readers. John Hennessy and Anoop Gupta provided the Dash multiprocessor-related results from Stanford University. Charles Seitz has taught me through his work on Cosmic Cube, Mosaic, and multicomputers. From MIT, I received valuable inputs from the works of Charles Leiserson, William Dally, Anant Agarwal, and Rishiyur Nikhil. From University of Illinois, I received the Cedar and Perfect benchmark information from Pen Yew.

Jack Dongarra of the University of Tennessee provided me the Linpack benchmark results. James Smith of Cray Research provided up-to-date information on the C-90 clusters and on the Cray/MPP. Ken Miura provided the information on Fujitsu VPP500. Lionel Ni of Michigan State University helped me in the areas of performance laws and adaptive wormhole routing. Justin Ratter provided information on the Intel Delta and Paragon systems. Burton Smith provided information on the Tera computer development.

Harold Stone and John Hayes suggested corrections and ways to improve the presentation. H.C. Torng of Cornell University, Andrew Chien of University of Illinois, and Daniel Tobak of George-Mason University made useful suggestions. Among my colleagues at the University of Southern California, Jean-Luc Gaudiot, Michel Dubois, Rafael Saavedra, Monte Ung, and Viktor Prasanna have made concrete suggestions to improve the manuscript. I appreciate the careful proofreading of an earlier version of the manuscript by D.K. Panda of the Ohio State University. The inputs from Vipin Kumar of the University of Minnesota, Xian-He Sun of NASA Langley Research Center, and Alok Choudhary of Syracuse University are also appreciated.

In addition to the above individuals, my understanding on computer architecture and parallel processing has been influenced by the works of David Kuck, Ken Kennedy,

Jack Dennis, Michael **Flynn**, Arvind, T.C. Chen, Wolfgang **Giloi**, Hairy Jordan, **H.T. Kung**, John Rice, H.J. Siegel, Allan Gottlieb, Philips Treleaven, Faye **Briggs**, Peter **Kogge**, Steve Chen, Ben Wah, Edward Davidson, **Alvin Despain**, James Goodman. Robert Keller, Duncan **Lawrie**, C.V. **Ramamoorthy**, Sartaj Sahni, **Jean-Loup Baer**, **Milos Ercegovac**, Doug DeGroot, Janak **Patel**, **Dharma Agrawal**, Lenart Johnsson, John Gustafson, **Tse-Yun Feng**, Herbert Schewetman, and Ken Batcher. I want to thank all of them for sharing their vast knowledge with me.

I want to acknowledge the research support I have received from the National Science **Foundation**, the Office of Naval Research, the Air Force Office of Scientific Research, International Business Machines. Intel Corporation, **Alliant** Computer Systems, and American Telephone and Telegraph Laboratories.

Technical exchanges with the **Electrotechnical** Laboratory (ETL) in Japan, the German National Center for Computer Research (GMD) in Germany, and the Industrial Technology Research Institute (**ITRI**) in Taiwan are always rewarding experiences to the author.

I appreciate the staff and facility support provided by Purdue University, the University of Southern California, the University of Minnesota, the University of Tokyo, National Taiwan University, and Academia Sinica during the past twenty years. In particular, I appreciate the encouragement and professional advices received from Henry Yang, Lofti Zadeh, Richard Karp, George Bekey, Arthur Gill, Ben Coates, Melvin **Breuer**, Jerry Mendel, Len Silverman, Solomon **Golomb**, and Irving Reed over the years.

Excellent work by the McGraw-Hill editorial and production **staff** has greatly improved the readability of this book. In particular, I want to thank Eric Munson for his continuous sponsorship of my book projects. I appreciate Joe Murphy and his coworkers for excellent copy editing and production jobs. Suggestions from reviewers listed below have greatly helped improve the contents and presentation.

The book was completed at the expense of cutting back many aspects of my life, spending many long hours, evenings, and weekends in seclusion during the last several years. I appreciate the patience and understanding of my friends, my students, and my family members during those ten.se periods. Finally, the book has been completed and I hope you enjoy reading it.

Kai Hwang

Reviewers:

Andrew A. Chien, *University of Illinois*;
David Culler, *University of California, Berkeley*,
Ratan K. **Guha**, *University of Central Florida*;
John P. Hayes, *University of Michigan*;
John Hennessy, *Stanford University*;
Dhamir Mannai, *Northeastern University*,
Michael **Quinn**, *Oregon State University*,
H. J. Siegel, *Purdue University*;
Daniel Tabak. *George-Mason University*.

ADVANCED COMPUTER ARCHITECTURE:

Parallelism, Scalability, Programmability

Part I

Theory of Parallelism

Chapter 1
Parallel Computer Models

Chapter 2
Program and Network Properties

Chapter 3
Principles of Scalable Performance

Summary

This theoretical part presents computer models, program **behavior**, architectural choices, scalability, programmability, and performance issues related to parallel processing. These topics form the foundations for designing high-performance computers and for the development of supporting software and applications.

Physical computers modeled include **shared-memory multiprocessors**, message-passing multicomputers, vector supercomputers, synchronous processor arrays, and massively parallel processors. The theoretical parallel random-access machine (PRAM) model is also presented. Differences between the PRAM model and physical architectural models are discussed. The VLSI complexity model is presented for **implementing** parallel algorithms directly in integrated circuits.

Network design principles and parallel program characteristics are introduced. These include dependence theory, computing granularity, communication latency, program flow mechanisms, network properties, performance laws, and scalability studies. This evolving theory of parallelism consolidates our understanding of parallel computers, from abstract models to hardware machines, software systems, and performance evaluation.

Chapter 1

Parallel Computer Models

Parallel processing has emerged as a key enabling technology in modern computers, driven by the ever-increasing **demand** for higher performance, lower costs, and sustained productivity in real-life applications. Concurrent events are taking place in today's high-performance computers due to the common practice of multiprogramming, multiprocessing, or multicompacting.

Parallelism appears in various forms, such as lookahead, pipelining, vectorization, concurrency, simultaneity, data **parallelism**, partitioning, interleaving, overlapping, multiplicity, replication, time sharing, space sharing, multitasking, multiprogramming, multithreading, and distributed computing at different processing levels.

In this chapter, we model physical architectures of parallel computers, vector supercomputers, multiprocessors, multicompacting, and massively parallel processors. Theoretical machine models are also presented, including the *parallel random-access machines (PRAMs)* and the complexity model of VLSI (*very large-scale integration*) circuits. Architectural development tracks are identified with case studies in the book. Hardware and software subsystems are introduced to pave the way for detailed studies in subsequent chapters.

1.1 The State of Computing

Modern computers are equipped with powerful hardware facilities driven by extensive software packages. To assess **state-of-the-art** computing, we first review historical milestones in the development of computers. Then we take a grand tour of the crucial hardware and software elements built into modern computer systems. We then examine the evolutional relations in milestone architectural development. Basic hardware and software factors are identified in analyzing the performance of computers.

1.1.1 Computer Development Milestones

Computers have gone through two major stages of development: mechanical and electronic. Prior to 1945, computers were made with mechanical or electromechanical

parts. The earliest mechanical computer can be traced back to 500 BC in the form of the abacus used in China. The abacus is manually operated to perform decimal arithmetic with carry propagation digit by digit.

Blaise Pascal built a mechanical adder/subtractor in France in 1642. Charles Babbage designed a difference engine in England for polynomial evaluation in 1827. Konrad Zuse built the first binary mechanical computer in Germany in 1941. Howard Aiken proposed the very first electromechanical decimal computer, which was built as the Harvard Mark I by IBM in 1944. Both Zuse's and Aiken's machines were designed for **general-purpose** computations.

Obviously, the fact that computing and communication were carried out with **moving** mechanical parts greatly limited the computing **speed** and reliability of mechanical computers. Modern computers were marked by the introduction of electronic components. The moving parts in mechanical computers were replaced by high-mobility electrons in electronic computers. Information transmission by mechanical gears or levers was replaced by electric signals traveling almost at the speed of light.

Computer Generations Over the past five decades, electronic computers have gone through five generations of development. Table 1.1 provides a summary of the five generations of electronic computer development. Each of the first three generations lasted about 10 years. The fourth generation covered a time span of 15 years. We have just entered the fifth generation with the use of processors and memory devices with more than 1 million transistors on a single silicon chip.

The division of generations is marked primarily by sharp changes in hardware and software technologies. The entries in Table 1.1 indicate the new hardware and software features introduced with each generation. Most features introduced in earlier generations have been passed to later generations. In other words, the latest generation **computers** have inherited all the nice features and eliminated all the bad ones found in previous generations.

Progress in Hardware As far as hardware technology is concerned, the first generation (1945-1954) used vacuum tubes and relay memories interconnected by insulated wires. The second generation (1955-1964) was marked by the use of discrete transistors, diodes, and magnetic ferrite **cores**, interconnected by printed circuits.

The third generation (1965-1974) began to use *integrated circuits (ICs)* for both logic and memory in *small-scale* or *medium-scale integration* (SSI or MSI) and multi-layered printed circuits. The fourth generation (**1974-1991**) used *large-scale* or *very-large-scale integration* (LSI or VLSI). Semiconductor memory replaced core memory as computers moved from the third to the fourth generation.

The fifth generation (1991 present) is highlighted by the use of high-density and high-speed processor and memory chips based on even more improved VLSI technology. For example, 64-bit 150-MHz microprocessors are now available on a single chip with over one million transistors. Four-megabit dynamic *random-access memory* (RAM) and 256K-bit static **RAM** are now in widespread use in **today's** high-performance computers.

It has been projected that four microprocessors will be built on a single CMOS chip with more than 50 million transistors, and 64M-bit dynamic RAM will become

Table 1-1 Five **Generations** of Electronic Computers

Generation	Technology and Architecture	Software and Applications	Representative Systems
First (1945-54)	Vacuum tubes and relay memories, CPU driven by PC and accumulator, fixed-point arithmetic.	Machine/assembly languages, single user, no subroutine linkage , programmed I/O using CPU.	ENIAC , Princeton IAS, IBM 701.
Second (1955–64)	Discrete transistors and core memories, floating-point arithmetic, I/O processors, multiplexed memory access.	HLL used with compilers, subroutine libraries, batch processing monitor .	IBM 7090, CDC 1604, Univac LARC .
Third (1965-74)	Integrated circuits (SSI/-MSI), microprogramming, pipelining, cache, and lookahead processors.	Multiprogramming and time-sharing OS, multiuser applications.	IBM 360/370, CDC 6600, TI-ASC, PDP-8.
Fourth (1975-90)	LSI/VLSI and semiconductor memory, multiprocessors, vector supercomputers, multic平computers.	Multiprocessor OS, languages, compilers, and environments for parallel processing.	VAX 9000, Cray X-MP, IBM 3090, BBN TC2000.
Fifth (1991-present)	ULSI/VHSIC processors, memory, and switches, high-density packaging, scalable architectures.	Massively parallel processing, grand challenge applications , heterogeneous processing.	Fujitsu VPP500, Cray/MPP , TMC/CM-5, Intel Paragon.

available in large quantities **within** the next decade.

The First Generation From the architectural and software points of view, first-generation computers were built with a single *central processing unit* (CPU) which performed serial fixed-point arithmetic using a program counter, branch instructions, and an accumulator. The CPU must be involved in **all** memory access and *input/output* (**I/O**) operations. Machine or assembly languages were used.

Representative systems include the ENIAC (Electronic Numerical Integrator and Calculator) built at the Moore School of the University of Pennsylvania in 1950; the IAS (Institute for Advanced Studies) computer based on a design proposed by John von Neumann, Arthur Burks, and Herman Goldstine at Princeton in 1946; and the IBM 701, the first electronic **stored-program** commercial computer built by IBM in 1953. Subroutine linkage was not implemented in early computers.

The Second Generation Index registers, floating-point arithmetic, multiplexed memory, and **I/O** processors were introduced with second-generation computers. *High-level languages* (**HLLs**), such as Fortran, Algol, and **Cobol**, were introduced along with compilers, subroutine libraries, and batch processing monitors. Register transfer language was developed by Irving Reed (1957) for systematic design of digital computers.

Representative systems include the IBM 7030 (the Stretch computer) featuring

instruction **lookahead** and error-correcting memories built in 1962, the **Univac LARC (Livermore** Atomic Research Computer) built in 1959, and the CDC 1604 built in the 1960s.

The Third Generation The third generation was **represented** by the IBM/360-370 Series, the CDC 6600/7600 **Series**, Texas Instruments ASC (Advanced Scientific Computer), and Digital Equipment's **PDP-8** Series from the mid-1960s to the mid-1970s.

Microprogrammed control became popular with this generation. Pipelining and cache memory were introduced to close up the speed gap between the CPU and main memory. The idea of multiprogramming was implemented to interleave CPU and I/O activities across multiple user programs. This led to the development of **time-sharing operating systems** (OS) using virtual memory with greater sharing or multiplexing of resources.

The Fourth Generation Parallel computers in various architectures appeared in the fourth generation of computers using shared or distributed memory or optional vector hardware. Multiprocessing OS, special languages, and compilers were developed for parallelism. Software tools and environments were created for parallel processing or distributed computing.

Representative systems include the VAX 9000, Cray **X-MP**, IBM/3090 **VF**, BBN TC-2000, etc. During these 15 years (1975-1990), the technology of parallel processing gradually became mature and entered the production mainstream.

The Fifth Generation Fifth-generation computers have just begun to appear. These machines emphasize *massively parallel processing* (MPP). Scalable and latency-tolerant architectures are being adopted in MPP systems using VLSI silicon, GaAs technologies, high-density packaging, and optical technologies.

Fifth-generation computers are targeted to achieve **Teraflops** (10^{12} floating-point operations per second) performance by the mid-1990s. *Heterogeneous processing* is emerging to solve large scale problems using a network of heterogeneous computers with shared virtual memories. The fifth-generation MPP systems are represented by several recently announced projects at Fujitsu (VPP500), Cray Research (MPP), Thinking Machines Corporation (the CM-5), and Intel Supercomputer Systems (the Paragon).

1.1.2 Elements of Modern Computers

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

Computing Problems It has been long recognized that the concept of computer architecture is no longer restricted to the structure of the bare machine hardware. A modern computer is an integrated system consisting of machine hardware, an instruction set, system software, application programs, and user interfaces. These system elements are depicted in Fig. 1.1. The use of a computer is **driven** by **real-life** problems demanding

fast and accurate solutions. Depending on the nature of the problems, the solutions may require different computing resources.

For numerical problems in science and technology, the solutions demand complex mathematical formulations and tedious integer or floating-point computations. For alphanumerical problems in business and government, the solutions demand accurate transactions, large database management, and information retrieval operations.

For artificial intelligence (**AI**) problems, the solutions demand logic inferences and symbolic manipulations. These computing problems have been labeled *numerical computing*, *transaction processing*, and *logical reasoning*. Some complex problems may demand a combination of these processing modes.

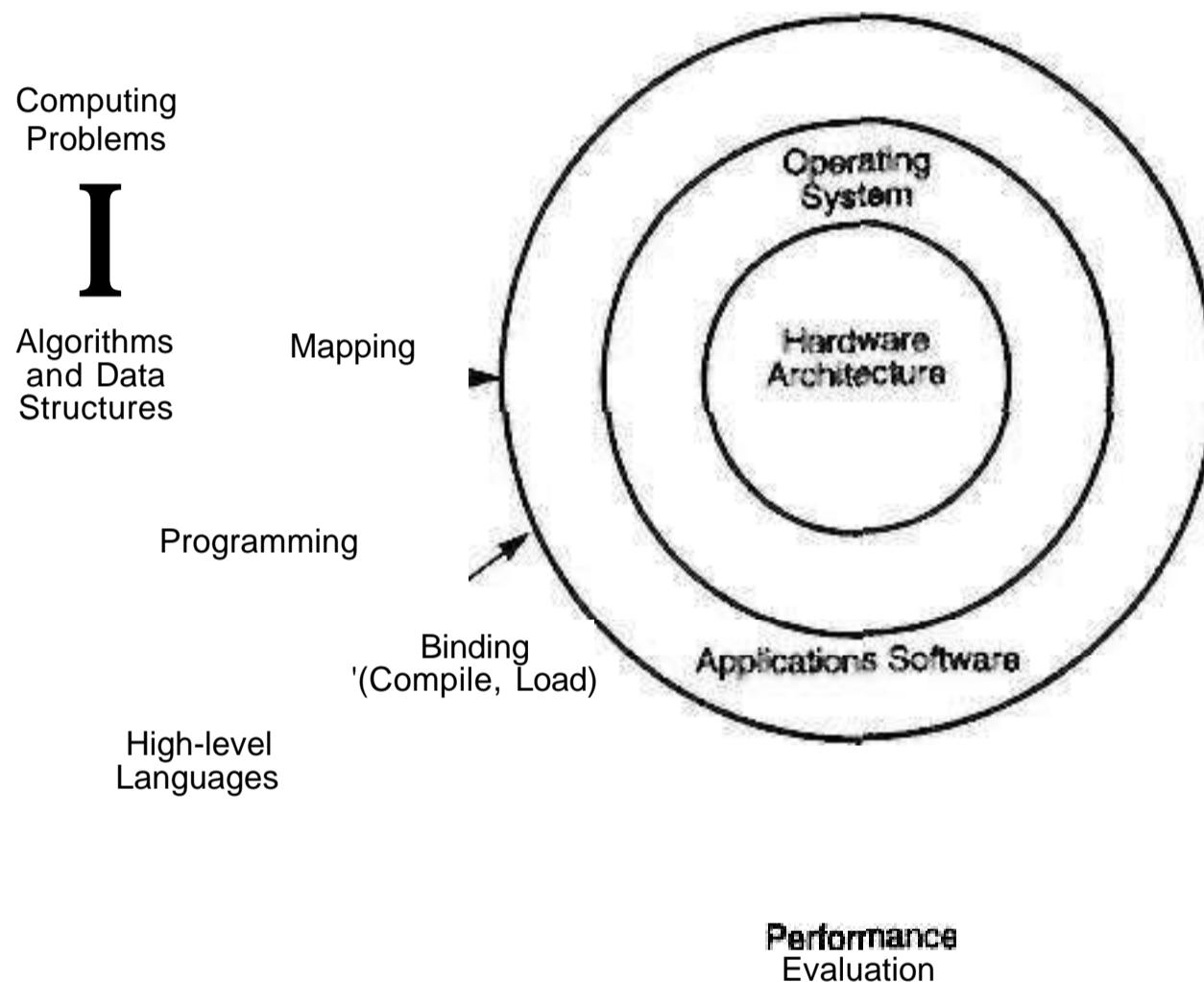


Figure 1.1 Elements of a modern computer system.

Algorithms and Data Structures Special algorithms and data structures are needed to specify the computations and communications involved in computing problems. Most numerical algorithms are deterministic, using regularly structured data. Symbolic processing may use heuristics or nondeterministic searches over large knowledge bases.

Problem formulation and the development of parallel algorithms often require interdisciplinary interactions among theoreticians, experimentalists, and computer programmers. There are many books dealing with the design and mapping of algorithms or heuristics onto parallel computers. In this book, we are more concerned about the

resources mapping problem than about the design and analysis of parallel **algorithms**

Hardware Resources The system architecture of a computer is represented by three nested circles on the right in Fig. 1.1. A modern computer system demonstrates its power through coordinated efforts by hardware resources, an operating system, and **application** software. Processors, memory, and peripheral devices form the hardware core of a computer system. We will study instruction-set processors, memory organization, multiprocessors, supercomputers, **multicomputers**, and massively parallel computers.

Special hardware interfaces are often built into **I/O** devices, such as terminals, workstations, optical page scanners, magnetic ink character recognizers, modems, file servers, voice data entry, printers, and plotters. These peripherals are connected to mainframe computers directly or through local or wide-area networks.

In addition, software interface programs are needed. These software interfaces include file transfer systems, editors, word processors, device drivers, interrupt handlers, network communication programs, etc. These programs greatly facilitate the portability of user programs on different machine architectures.

Operating System An effective operating system manages the allocation and deallocation of resources during the execution of user programs. We will study UNIX extensions for multiprocessors and multicomputers in Chapter 12. Mach/OS kernel and **OSF/1** will be specially studied for multithreaded kernel functions, virtual memory management, file subsystems, and network communication services. Beyond the OS, application software must be developed to benefit the users. Standard benchmark* programs are needed for performance evaluation.

Mapping is a bidirectional process matching algorithmic structure with hardware architecture, and vice versa. Efficient mapping will benefit the programmer and produce better source codes. The mapping of algorithmic and data structures onto the machine architecture includes processor scheduling, memory maps, interprocessor communications, etc. These activities are usually architecture-dependent.

Optimal mappings are sought for various computer architectures. The **implementation** of these mappings relies on efficient compiler and operating system support. Parallelism can be exploited at algorithm design time, at program time, at compile time, and at run time. Techniques for exploiting parallelism at these levels form the core of parallel processing technology.

System Software Support Software support is needed for the development of efficient programs in high-level languages. The source code written in a **HLL** must be **first** translated into object code by an optimizing compiler. The *compiler* assigns variables to registers or to memory words and reserves functional units for operators. An *assembler* is used to translate the compiled object code into machine code which can be recognized by the machine hardware. A *loader* is used to initiate the program execution through the OS kernel.

Resource binding demands the **use** of the compiler, assembler, loader, and OS kernel to commit physical machine resources to program execution. The effectiveness of this process determines the efficiency of hardware utilization and the **programmability** of the

computer. Today, programming parallelism is still very difficult for most programmers due to the fact that existing languages were originally developed for sequential computers. Programmers are often forced to program **hardware-dependent** features instead of programming parallelism in a generic and portable way. Ideally, we need to develop a parallel programming environment with **architecture-independent** languages, compilers, and software tools.

To develop a parallel language, we aim for efficiency in its implementation, portability across **different** machines, compatibility with existing sequential languages, expressiveness of parallelism, and ease of programming. One can attempt a new language approach or try to extend existing sequential languages gradually. A new language approach has the advantage of using explicit high-level constructs for specifying parallelism. However, new languages are often incompatible with existing languages and require new compilers or new passes to existing compilers. Most systems choose the language extension approach.

Compiler Support There are three compiler upgrade approaches: *preprocessor*, *precompiler*, and *parallelizing compiler*. A preprocessor uses a sequential compiler and a low-level library of the target computer to implement high-level parallel constructs. The precompiler approach requires some program flow analysis, dependence checking, and limited optimizations toward parallelism detection. The third approach demands a fully developed parallelizing or vectorizing compiler which can automatically detect parallelism in source code and transform sequential codes into parallel constructs. These approaches will be studied in Chapter 10.

The efficiency of the binding process depends on the effectiveness of the preprocessor, the precompiler, the parallelizing compiler, the loader, and the OS support. Due to unpredictable program behavior, none of the existing compilers can be considered fully automatic or fully intelligent in detecting all types of **parallelism**. Very often *compiler directives* are inserted into the source code to help the compiler do a better job. Users may interact with the compiler to restructure the programs. This has been proven useful in enhancing the performance of parallel computers.

1.1.3 Evolution of Computer Architecture

The study of computer architecture involves both hardware organization and **programming/software requirements**. As **seen** by an **assembly** language programmer, computer architecture is abstracted by its instruction set, which includes opcode (operation codes), addressing modes, registers, virtual memory, etc.

From the hardware implementation point of view, the abstract machine is organized with CPUs, caches, buses, microcode, pipelines, physical memory, etc. Therefore, the study of architecture covers both instruction-set architectures and machine implementation organizations.

Over the past four decades, computer architecture has gone through evolutional rather than revolutionary changes. Sustaining features are those that were proven performance deliverers. As depicted in Fig. 1.2, we started with the von Neumann architecture built as a sequential machine executing scalar data. The sequential computer was

improved from bit-serial to word-parallel operations, and from fixed-point to **floating-point** operations. The von Neumann architecture is slow due to **sequential** execution of instructions in programs.

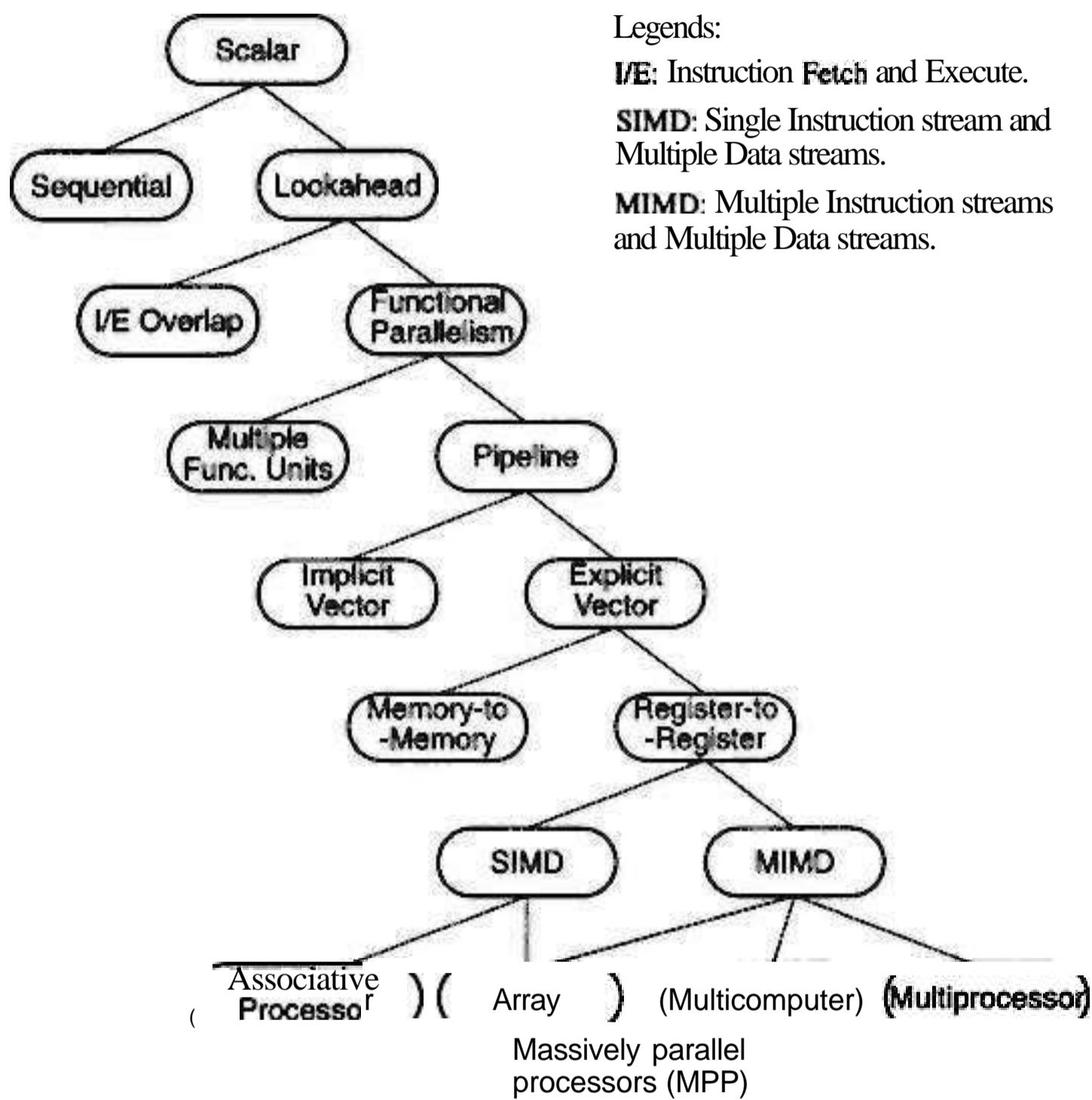


Figure 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers.

Lookahead, Parallelism, and Pipelining were introduced to prefetch instructions in order to overlap I/E (instruction fetch/decode and execution) operations and to enable functional parallelism. Functional parallelism was supported by two approaches: One is to use multiple functional units simultaneously, and the other is to practice pipelining at **various** processing levels.

The latter includes pipelined instruction execution, pipelined arithmetic computations, and memory-access operations. Pipelining has proven especially attractive in

performing identical operations repeatedly over vector data strings. Vector operations were originally carried out implicitly by **software-controlled** looping using scalar pipeline processors.

Flynn's Classification Michael **Flynn** (1972) introduced a classification of various computer architectures based on notions of instruction and data streams. As illustrated in Fig. 1.3a, conventional sequential machines are called SISD (*single instruction stream over a single data stream*) computers. Vector computers are equipped with scalar and vector hardware or appear as SIMD (*single instruction stream over multiple data streams*) machines (Fig. 1.3b). Parallel computers are reserved for MIMD (*multiple instruction streams over multiple data streams*) machines.

An MISD (*multiple instruction streams and a single data stream*) machines are modeled in Fig. 1.3d. The same data stream flows through a linear array of processors executing different instruction streams. This architecture is also known as *systolic arrays* (Kung and Leiserson, 1978) for pipelined execution of specific algorithms.

Of the four machine models, most parallel computers built in the past assumed the MIMD model for general-purpose computations. The SIMD and MISD models are more suitable for special-purpose computations. For this reason, MIMD is the most popular model, SIMD next, and MISD the least popular model being applied in commercial machines.

Parallel/Vector Computers Intrinsic parallel **computers** are those that execute programs in MIMD mode. There are two major classes of parallel computers, namely, *shared-memory multiprocessors* and *message-passing multicomputers*. The major distinction between multiprocessors and multicomputers lies in memory sharing and the mechanisms used for interprocessor communication.

The processors in a multiprocessor system communicate with each other through *shared variables* in a common memory. Each computer node in a multicomputer system has a local memory, unshared with other nodes. Interprocessor communication is done through *message passing* among the nodes.

Explicit vector **instructions** were introduced with the appearance of *vector processors*. A vector processor is equipped with multiple vector pipelines that can be concurrently used under hardware or firmware control. There are two families of pipelined vector processors:

Memory-to-memory architecture supports the pipelined flow of vector operands directly from the memory to pipelines and then back to the memory. *Register-to-register* architecture uses vector registers to interface between the memory and functional pipelines. Vector processor architectures will be studied in Chapter 8.

Another important branch of the architecture tree consists of the SIMD computers for synchronized vector processing. An SIMD computer exploits *spatial parallelism* rather than *temporal parallelism* as in a pipelined computer. SIMD computing is achieved through the use of an array of *processing elements* (PEs) synchronized by the same controller. Associative memory can be used to build SIMD associative processors. SIMD machines will be treated in Chapter 8 along with pipelined vector computers.

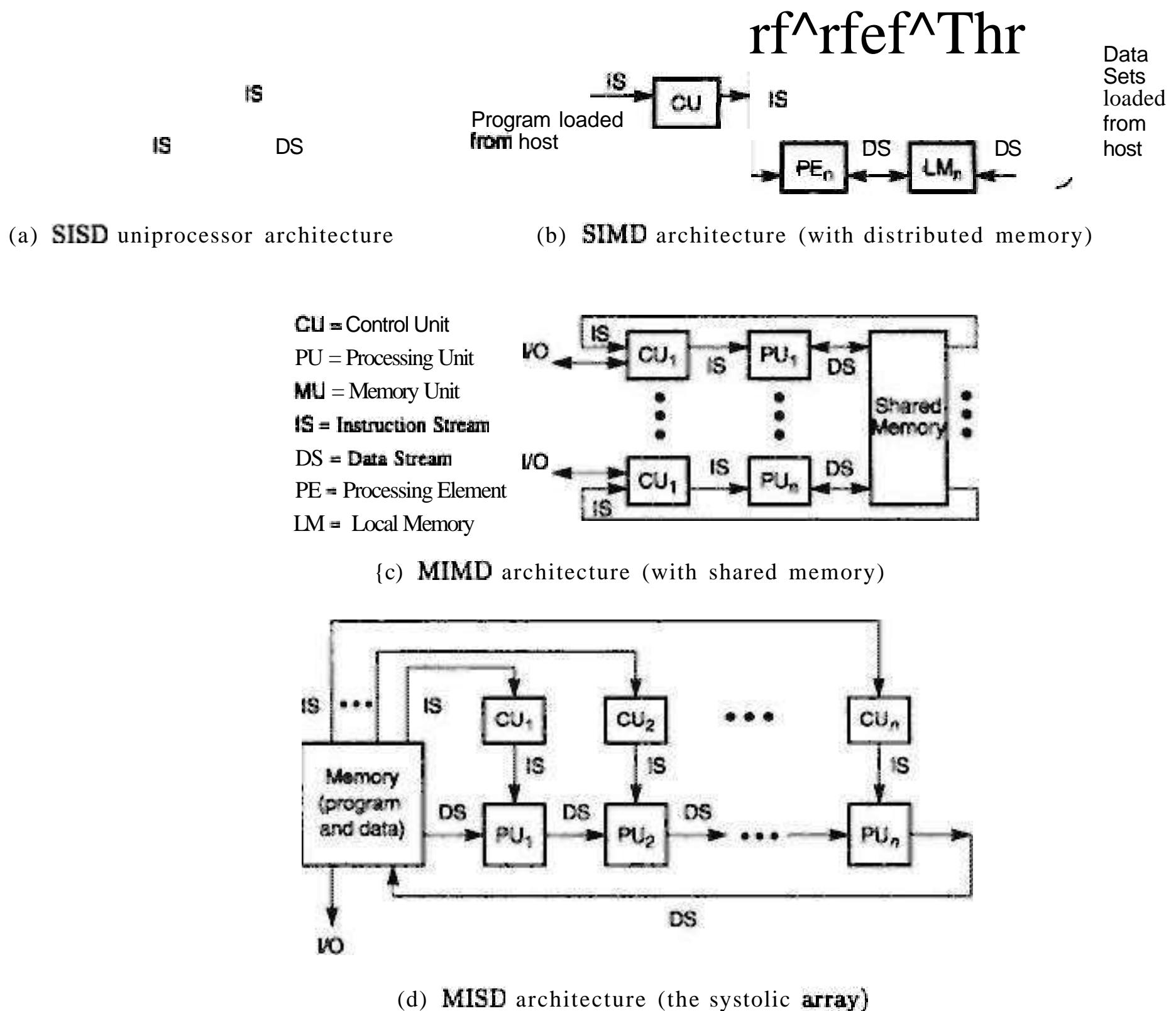


Figure 1.3 **Flynn's** classification of computer architectures. (Derived from Michael Flynn, 1972)

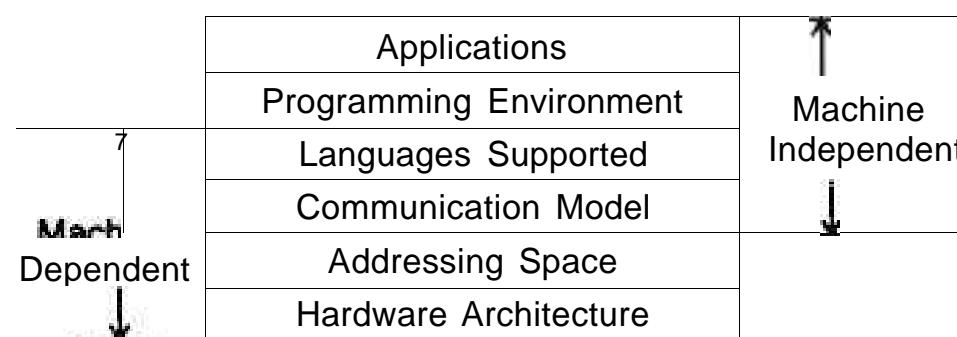


Figure 1.4 **Six** layers for computer system development. (Courtesy of Lionel Mi, 1990)

Development Layers A layered **development** of parallel computers is illustrated in Fig. 1.4, based on a recent classification by Lionel Ni (1990). Hardware configurations differ from machine to machine, even those of the same model. The address space of a processor in a computer system varies among different architectures. It depends on the memory organization, which is machine-dependent. These features are up to the designer and should match the target application domains.

On the other hand, we want to develop application programs and programming environments which are **machine-independent**. Independent of machine architecture, the user programs can be ported to many computers with minimum conversion costs. High-level languages and communication models depend on the architectural choices made in a computer system. From a programmer's viewpoint, these two layers should be **architecture-transparent**.

At present, Fortran, C, Pascal, Ada, and Lisp are supported by most computers. However, the communication models, shared variables versus message passing, are mostly **machine-dependent**. The Linda approach using *tuple spaces* offers an architecture-transparent communication model for parallel computers. These language features will be studied in Chapter 10.

Application programmers prefer more architectural transparency. However, kernel programmers have to explore the opportunities supported by hardware. As a good **computer architect**, one has to approach the problem from both ends. The compilers and OS support should be designed to remove as many architectural constraints as possible from the programmer.

New Challenges The technology of parallel processing is the outgrowth of four decades of research and industrial advances in microelectronics, printed circuits, high-density packaging, advanced processors, memory systems, peripheral devices, communication channels, language evolution, compiler sophistication, operating systems, programming environments, and application challenges.

The rapid progress made in hardware technology has significantly increased the economical feasibility of building a new generation of computers adopting parallel processing. However, the major barrier preventing parallel processing from entering the production mainstream is on the software and application side.

To date, it is still very difficult and painful to program parallel and vector computers. We need to strive for major progress in the software area in order to create a user-friendly environment for high-power computers. A whole new generation of programmers need to be trained to program parallelism effectively. High-performance computers provide fast and accurate solutions to scientific, engineering, business, social, and defense problems.

Representative real-life problems include weather forecast modeling, computer-aided design of VLSI circuits, large-scale database management, artificial intelligence, crime control, and strategic defense initiatives, just to name a few. The application domains of parallel processing computers are expanding steadily. With a good understanding of scalable computer architectures and mastery of parallel programming **techniques**, the reader will be better prepared to face future computing challenges.

1.1.4 System Attributes to Performance

The ideal performance of a computer system demands a perfect match between machine capability and program behavior. Machine capability can be enhanced with better hardware technology, innovative architectural features, and efficient resources management. **However**, program behavior is difficult to predict due to its heavy dependence on application and run-time conditions.

There are also many other factors affecting program behavior, including algorithm design, data structures, language efficiency, programmer skill, **and compiler** technology. It is impossible to achieve a perfect match between hardware and software by merely improving only a few factors without touching other factors.

Besides, machine performance may vary from program to program. This makes *peak performance* an impossible target to achieve in real-life applications. On the other **hand**, a machine cannot be said to have an average performance either. All performance **indices** or benchmarking results must be tied to a program mix. **For this reason**, the **performance** should be described as a range or as a harmonic distribution.

We introduce below fundamental factors for projecting the performance of a computer. These performance indicators are by no means conclusive in **all** applications. However, they can be used to guide system architects in designing better machines or to educate programmers or compiler writers in optimizing the codes for more efficient execution by the hardware.

Consider the execution of a given program on a given computer. The simplest measure of program performance is the *turnaround time*, which includes disk and memory accesses, input and output activities, compilation time, OS overhead, and CPU time. In order to shorten the turnaround time, one must reduce all these time factors.

In a **multiprogrammed** computer, the **I/O** and system overheads of a given program may overlap with the CPU times required in other programs. Therefore, it is fair to compare just the total CPU time needed for program execution. The CPU is used to execute both system programs and user programs. It is the user CPU time that concerns the user most.

Clock Rate and CPI The CPU (or simply the *processor*) of today's digital computer is driven by a clock with a constant *cycle time* (T in nanoseconds). The inverse of the cycle time is the *clock rate* ($f = 1/T$ in megahertz). The size of a program is determined by its *instruction count* (I_c), in terms of the number of machine instructions to be executed in the program. Different machine instructions may require different numbers of clock cycles to execute. Therefore, the *cycles per instruction* (CPI) becomes an important parameter for measuring the time needed to execute each instruction.

For a given instruction set, we can calculate an *average* CPI over all instruction types, provided we know their frequencies of appearance in the program. An accurate estimate of the average CPI requires a large amount of program code to be traced over a long period of time. Unless specifically focusing on a single instruction type, we simply use the term CPI to mean the average value with respect to a given instruction set and a given program mix.

Performance Factors Let I_c be the number of instructions in a given program, or the instruction count. The CPU time (T in seconds/program) needed to execute the program is estimated by finding the product of three contributing factors:

$$r = I_c \times CPI \times \tau \quad (1.1)$$

The execution of an instruction requires going through a cycle of events involving the instruction fetch, decode, operand(s) fetch, execution, and store results. In this cycle, only the instruction decode and execution phases are carried out in the CPU. The remaining three operations may be required to access the memory. We define a *memory cycle* as the time **needed** to complete one memory reference. Usually, a memory cycle is k times the processor cycle τ . The value of k depends on the speed of the memory technology and processor-memory interconnection scheme used.

The **CPI** of an instruction type can be divided into two component terms corresponding to the total processor cycles and memory cycles needed to complete the execution of the instruction. Depending on the instruction type, the complete instruction cycle may involve one to four memory references (one for instruction fetch, two for operand fetch, and one for store results). Therefore we can rewrite Eq. 1.1 as follows;

$$T = I_c \times \{p + m \times k\} \times \tau \quad (1.2)$$

where p is the number of processor cycles **needed** for the instruction decode and execution, m is the number of memory references needed, k is the ratio between memory cycle and processor cycle, I_c is the instruction **count**, and r is the processor cycle time. Equation 1.2 can be further refined once the **CPI** components (p, m, k) are weighted over the entire instruction set.

System Attributes The above five performance factors (I_c, p, m, k, T) are influenced by four system attributes: instruction-set architecture, compiler technology, CPU implementation and control, and cache and memory hierarchy, as specified in Table 1.2.

The instruction-set architecture affects the program length (I_c) and processor cycle needed (p). The compiler technology affects the values of I_c , p , and the memory reference count (m). The CPU implementation and control determine the total processor time ($p \cdot \tau$) needed. Finally, the memory technology and hierarchy design affect the memory access latency ($k \cdot r$). The above CPU time can be used as a basis in estimating the execution rate of a processor.

MIPS Rate Let C be the total number of clock cycles needed to execute a given program. Then the CPU time in Eq. 1.2 can be estimated as $T = C \times r = C/f$. Furthermore, $CPI = C/I_c$ and $T = I_c \times CPI \times r = I_c \times CPI \times f$. The processor speed is often measured in terms of *million instructions per second* (MIPS). We simply **call** it the MIPS rate of a given processor. It should be emphasized that the MIPS rate varies with respect to a number of factors, including the clock rate (/), the instruction count (I_c), and the CPI of a given machine, as defined below:

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

Table 1.2 Performance Factors Versus System Attributes

System Attributes	Instr. Count,	Performance Factors			Processor Cycle Time, T
		Average Cycles per Instruction, CPI	Processor Cycles per Instruction, p	Memory References per Instruction, m	
Instruction-set Architecture	X	X			
Compiler Technology	X	X	X		
Processor Implementation and Control		X			X
Cache and Memory Hierarchy				X	X

Based on Eq. 1.3, the CPU time in Eq. 1.2 can also be written as $T = I_c \times 10^{-6}/\text{MIPS}$. Based on the system attributes identified in Table 1.2 and the above derived expressions, we conclude by indicating the fact that the MIPS rate of a given computer is directly proportional to the clock rate and inversely proportional to the CPI. All four system attributes, instruction set, compiler, processor, and memory technologies, affect the MIPS rate, which varies also from program to program.

Throughput Rate Another important concept is related to how many programs a system can execute per unit time, called the *system throughput* W_s (in programs/second). In a **multiprogrammed** system, the system throughput is often lower than the *CPU throughput* W_p defined by:

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

Note that $W_p = (\text{MIPS}) \times 10^6/I_c$ from Eq. 1.3. The unit for W_p is programs/second. The CPU throughput is a measure of how many programs can be executed per second, based on the MIPS rate and average program length (I_c). The reason why $W_s < W_p$ is due to the additional system overheads caused by the I/O, compiler, and OS when multiple programs are interleaved for CPU execution by multiprogramming or time-sharing operations. If the CPU is kept busy in a perfect program-interleaving fashion, then $W_s = W_p$. This will probably never happen, since the system overhead often causes an extra delay and the CPU may be left idle for some cycles.

Example 1.1 MIPS ratings and performance measurement

Consider the use of a VAX/780 and an IBM RS/6000-based workstation to execute a hypothetical benchmark program. Machine characteristics and claimed performance are given below:

Machine	Clock	Performance	CPU Time
VAX 11/780	5 MHz	1 MIPS	12x seconds
IBM RS/6000	25 MHz	18 MIPS	x seconds

These data indicate that the measured CPU time on the VAX 11/780 is 12 times longer than that measured on the RS/6000. The object codes running on the two machines have different lengths due to the differences in the machines and compilers used. All other overhead times are ignored.

Based on Eq. 1.3, the instruction count of the object code running on the RS/6000 is 1.5 times longer than that of the code running on the VAX machine. Furthermore, the average CPI on the VAX/780 is assumed to be 5, while that on the RS/6000 is 1.39 executing the same benchmark program.

The VAX 11/780 has a typical CISC (*complex instruction set computing*) architecture, while the IBM machine has a typical RISC (*reduced instruction set computing*) architecture to be characterized in Chapter 4. This example offers a simple comparison between the two types of computers based on a single program run. When a different program is run, the conclusion may not be the same.

We cannot calculate the CPU throughput W_p unless we know the program length and the average CPI of each code. The system throughput W_s should be measured across a large number of programs over a long observation period. The message being conveyed is that one should not draw a sweeping conclusion about the performance of a machine based on one or a few program runs.

Programming Environments The **programmability** of a computer depends on the programming environment provided to the users. Most computer environments are not **user-friendly**. In fact, the marketability of any new computer system depends on the creation of a user-friendly environment in which programming becomes a joyful undertaking rather than a nuisance. We briefly introduce below the environmental features desired in modern computers.

Conventional uniprocessor computers are programmed in a *sequential environment* in which instructions are executed one after another in a sequential manner. In fact, the original UNIX/OS kernel was designed to respond to one system call from the user process at a time. Successive system calls must be serialized through the kernel.

Most existing compilers are designed to generate sequential object codes to run on a sequential computer. In other words, conventional computers are being used in a sequential programming environment using languages, compilers, and operating systems all developed for a uniprocessor computer.

When using a parallel computer, one desires a *parallel environment* where parallelism is automatically exploited. Language extensions or new constructs must be developed to specify parallelism or to facilitate easy detection of parallelism at various granularity levels by more intelligent compilers.

Besides parallel languages and compilers, the operating systems must be also extended to support parallel activities. The OS must be able to manage the resources

behind parallelism. **Important** issues include parallel scheduling of concurrent events, shared memory allocation, and shared peripheral and communication links.

Implicit Parallelism An implicit approach uses a conventional language, such as C, Fortran, Lisp, or Pascal, to write the source program. The sequentially coded source program is translated into parallel object code by a parallelizing compiler. As illustrated in Fig. 1.5a, this compiler must be able to detect parallelism and **assign** target machine resources. This compiler approach has been applied in programming shared-memory multiprocessors.

With parallelism being implicit, success relies heavily on the “**intelligence**” of a parallelizing compiler. This approach requires less effort on the part of the programmer. David Kuck of the University of Illinois and Ken Kennedy of Rice University and their associates have adopted this implicit-parallelism approach.

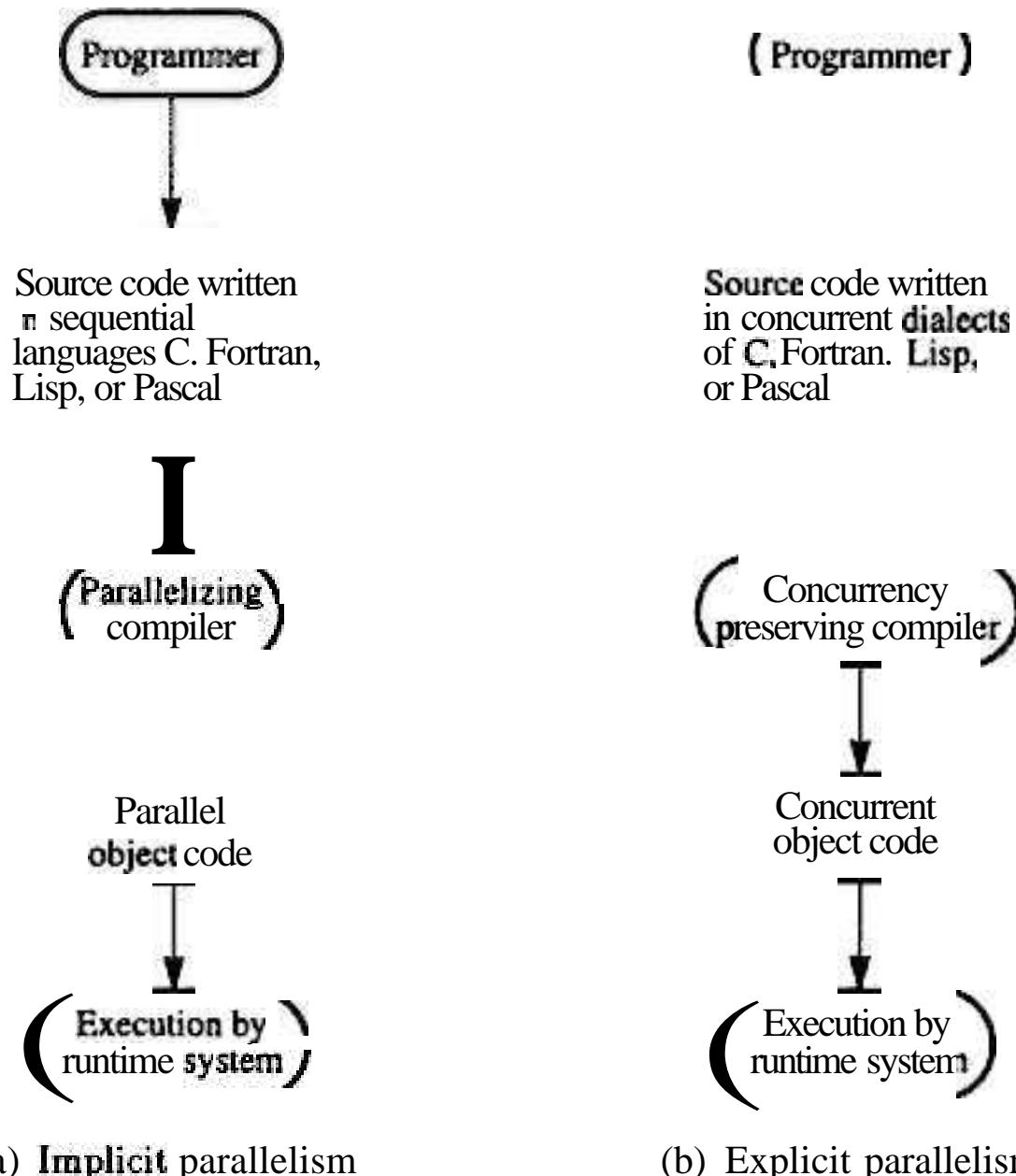


Figure 1.5 Two approaches to parallel programming. (Courtesy of Charles Seitz; reprinted with permission from "Concurrent Architectures", p. 51 and p. 53, *VLSI and Parallel Computation*, edited by Suaya and Birtwistle, Morgan Kaufmann Publishers, 1990)

Explicit Parallelism The second approach (Fig. 1.5b) requires more effort by the programmer to **develop** a source program **using parallel** dialects of C, Fortran, Lisp, or Pascal. **Parallelism** is explicitly specified in the user programs. This will significantly reduce the burden on the compiler to detect parallelism. Instead, the compiler needs to preserve parallelism and, where **possible**, assigns target machine resources. Charles Seitz of California Institute of Technology and William Dally of Massachusetts Institute of Technology adopted this explicit approach in multicomputer development.

Special software tools are needed to make an environment more friendly to user groups. Some of the tools are parallel extensions of conventional high-level languages. Others are integrated environments which include tools providing different levels of program abstraction, **validation**, **testing**, **debugging**, and **tuning**; **performance** prediction and monitoring; and visualization support to aid program development, performance measurement, and graphics display and animation of computational results.

1.2 Multiprocessors and Multicomputers

Two categories of parallel computers are architecturally modeled below. These physical models are distinguished by having a shared common memory or unshared distributed memories. Only architectural organization models are described in Sections 1.2 and 1.3. Theoretical **and** complexity models for parallel **computers** are presented in Section 1.4.

1.2.1 Shared-Memory Multiprocessors

We describe below three shared-memory multiprocessor models: the *uniform-memory-access (UMA)* model, the *nonuniform-memory-access (NUMA)* model, and the *cache-only memory architecture (COMA)* model. These models differ in how the memory and peripheral resources are **shared** or distributed.

The UMA Model In a UMA multiprocessor model (Fig. 1.6), the physical memory is uniformly shared by all the processors. All processors have equal access time to all memory words, which is why it is called uniform memory access. Each processor may use a private cache. Peripherals are also shared in some fashion.

Multiprocessors **are** called *tightly coupled systems* due to the high degree of resource sharing. The system interconnect takes the form of a common **bus**, a crossbar switch, or a multistage network to be studied in Chapter 7.

Most computer manufacturers have *multiprocessor* (MP) extensions of their *uniprocessor* (UP) product line. The UMA model is suitable for general-purpose and time-sharing applications by **multiple** users. It can be used to speed up the execution of a single large program in time-critical applications. To coordinate parallel events, synchronization and communication among processors are done through using shared variables in the **common** memory.

When all processors have equal access to all peripheral devices, the system is called a *symmetric multiprocessor*. In this case, all the processors are equally capable of running the executive programs, such as the OS kernel and **I/O** service routines.

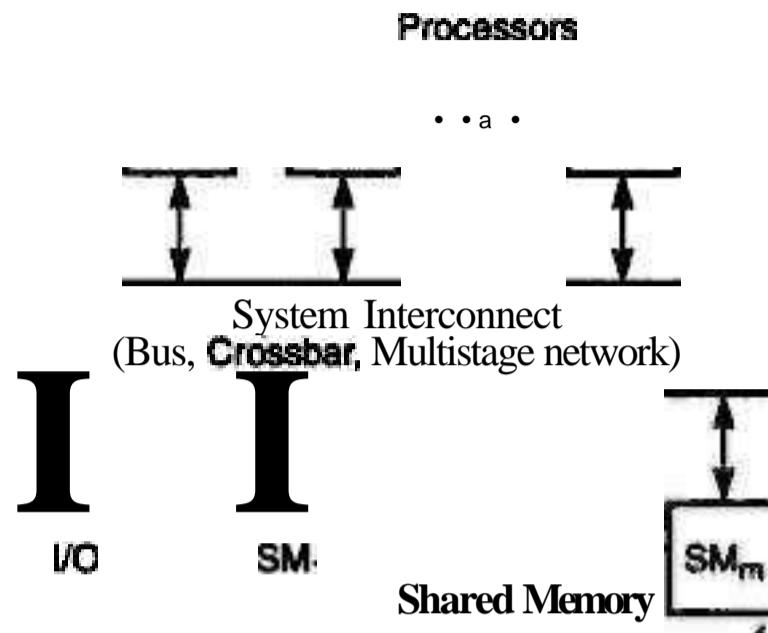


Figure 1.6 The UMA multiprocessor model (e.g., the Sequent Symmetry S-81).

In an *asymmetric* multiprocessor, only one or a subset of processors are **executive-capable**. An executive or a master processor can execute the operating system and handle **I/O**. The remaining processors have no **I/O** capability and thus are called *attached processors (APs)*. Attached processors execute user codes under the supervision of the master processor. In both MP and AP configurations, memory sharing among master and attached processors is still in place.

Example 1.2 Approximated performance of a multiprocessor

This example exposes the reader to parallel program execution on a shared-memory multiprocessor system. Consider the following Fortran program written for sequential execution on a uniprocessor system. All the arrays, A(I), B(I), and C(I), are assumed to have TV elements.

```

L1:      Do 10 I = 1, N
L2:      A(I) = B(I) + C(I)
L3:      10 Continue
L4:      SUM = 0
L5:      Do 20 J = 1, N
L6:      SUM = SUM + A(J)
L7:      20 Continue
  
```

Suppose **each line** of code L2, L4, and L6 takes 1 machine cycle to execute. The time required to execute the program control statements L1, L3, L5, and L7 is ignored to simplify the analysis. Assume that **k** cycles are needed for each interprocessor communication operation via the shared memory.

Initially, **all** arrays are **assumed** already loaded in the main memory and the short program fragment already loaded in the instruction cache. In other words, instruction fetch and data loading overhead is ignored. Also, we ignore bus con-

tention or memory access conflicts problems. In this way, we can concentrate on the analysis of CPU demand.

The above program can be executed on a sequential machine in $2N$ cycles under the above assumptions. N cycles are needed to execute the N independent iterations in the **I** loop. Similarly, TV cycles are needed for **the J** loop, which contains N recursive iterations.

To execute the program on an **M-processor** system, we partition **the looping** operations into **M** sections with $L = N/M$ elements per section. In the following parallel code, **Doall** declares that all **M** sections be executed by **M** processors in parallel:

```

Doall K=1,M
  Do 10 I = L(K-1) + 1, KL
    A(I) = B(I) + C(I)
  10 Continue
    SUM(K) = 0
    Do 20 J = 1, L
      SUM(K) = SUM(K) + A(L(K-1) + J)
    20 Continue
Endall

```

For **M-way** parallel execution, the sectioned **I** loop can be done in **L** cycles. The sectioned **J** loop produces **M** partial sums in **L** cycles. Thus $2L$ cycles are consumed to produce all **M** partial sums. Still, we need to merge these **M** partial sums to produce the final sum of **N** elements.

The addition of each pair of partial sums requires k cycles through the shared memory. An $/$ -level binary adder tree can be constructed to merge all the partial sums, where $l = \log_2 M$. The adder tree takes $l(k+1)$ cycles to merge the **M** partial sums sequentially from the leaves to the root of the tree. Therefore, the multiprocessor requires $2L + t(k+1) = 2N/M + (k+1)\log_2 M$ cycles to produce the final sum.

Suppose $N = 2^{20}$ elements in the array. Sequential execution of the original program takes $2N = 2^{21}$ machine cycles. Assume that each IPC synchronization overhead has an average value of $k = 200$ cycles. Parallel execution on $M = 256$ processors requires $2^{13} + 1608 = 9800$ machine cycles.

Comparing the above timing results, the multiprocessor shows a speedup factor of 214 out of the maximum value of 256. Therefore, an efficiency of $214/256 = 83.6\%$ has been achieved. We will study the speedup and efficiency issues in Chapter 3.

The above result was obtained under favorable assumptions about overhead. In reality, the resulting speedup might be lower after considering all software overhead and potential resource conflicts. Nevertheless, the example shows the promising side of **parallel** processing if the interprocessor communication overhead can be reduced to a sufficiently low level.

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

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

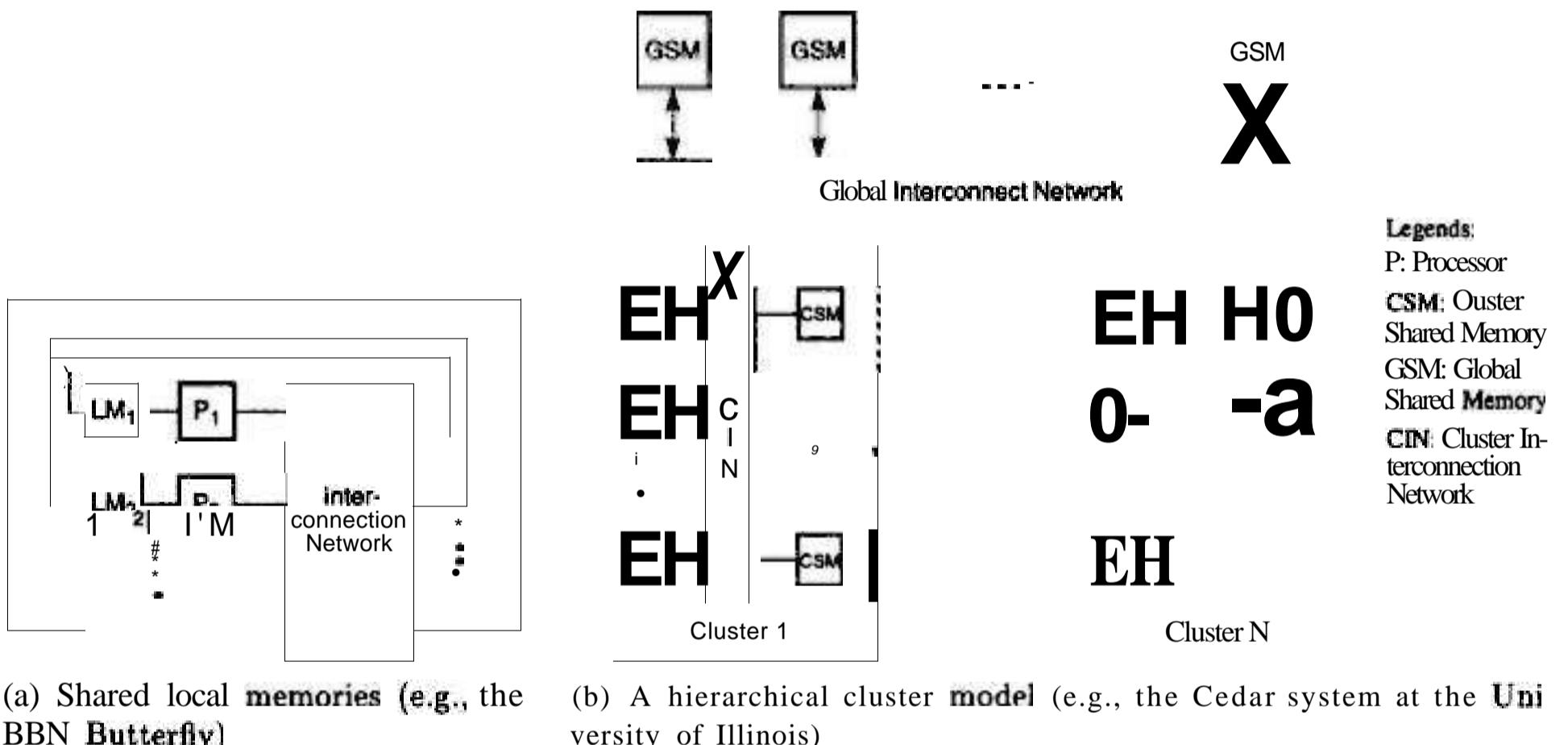


Figure 1.7 Two NUMA models for multiprocessor systems.

Besides distributed memories, globally shared memory can be added to a multiprocessor system. In this case, there are three memory-access patterns: The fastest is local memory access. The next is global memory access. The slowest is access of remote memory as illustrated in Fig. 1.7b. As a matter of fact, the models shown in Figs. 1.6 and 1.7 can be easily modified to allow a mixture of shared memory and **private** memory with **prespecified** access rights.

A hierarchically structured multiprocessor is modeled in Fig. 1.7b. The processors are divided into several **clusters**. Each cluster is itself an UMA or a NUMA multiprocessor. The clusters are connected to *global shared-memory* modules. The entire system is considered a NUMA multiprocessor. All processors belonging to the same cluster are allowed to uniformly access the *cluster shared-memory* modules.

AH clusters have equal access to the global memory. However, the access time to the cluster memory is shorter than that to the global memory. One can specify the access right among **intercluster** memories in various ways. The Cedar multiprocessor,

built at the University of Illinois, assumes such a structure in which each cluster is an Alliant FX/80 multiprocessor.

Table 1.3 Representative Multiprocessor Systems

Company and Model	Hardware and Architecture	Software and Applications	Remarks
Sequent Symmetry S-81	Bus-connected with 30 i386 processors, IPC via SLIC bus; Weitek floating-point accelerator.	DYNIX/OS, KAP/Sequent preprocessor, transaction multiprocessing.	i486-based multiprocessor available 1991. i586-based systems to appear.
IBM ES/9000 Model 900/VF	Model 900/VF has 6 ES/9000 processors with vector facilities, crossbar connected to I/O channels and shared memory.	OS support: MVS, VM KMS, AIX/370, parallel Fortran, VSF V2.5 compiler.	Fiber optic channels, integrated cryptographic architecture.
BBN TC-2000	512 M88100 processors with local memory connected by a Butterfly switch, a NUMA machine.	Ported Mach/OS with multiclustering, use parallel Fortran, time-critical applications.	Shooting for higher performance using faster processors in future models.

The COMA Model A multiprocessor using cache-only memory assumes the COMA model. Examples of COMA machines include the Swedish Institute of Computer Science's Data Diffusion Machine (DDM, Hagersten et al., 1990) and Kendall Square Research's KSR-1 machine (Burkhardt et al., 1992). The COMA model is depicted in Fig. 1.8. Details of KSR-1 are given in Chapter 9.

The COMA model is a special case of a NUMA machine, in which the distributed main memories are converted to caches. There is no memory hierarchy at each processor node. All the caches form a global address space. Remote cache access is assisted by the distributed cache directories (D in Fig. 1.8). Depending on the interconnection network used, sometimes hierarchical directories may be used to help locate copies of cache blocks. Initial data placement is not critical because data will eventually migrate to where it will be used.

Besides the UMA, NUMA, and COMA models specified above, other variations exist for multiprocessors. For example, a *cache-coherent non-uniform memory access* (CC-NUMA) model can be specified with distributed shared memory and cache directories. Examples of the CC-NUMA model include the Stanford Dash (Lenoski et al., 1990) and the MIT Alewife (Agarwal et al., 1990) to be studied in Chapter 9. One can also insist on a cache-coherent COMA machine in which all cache copies must be kept consistent,

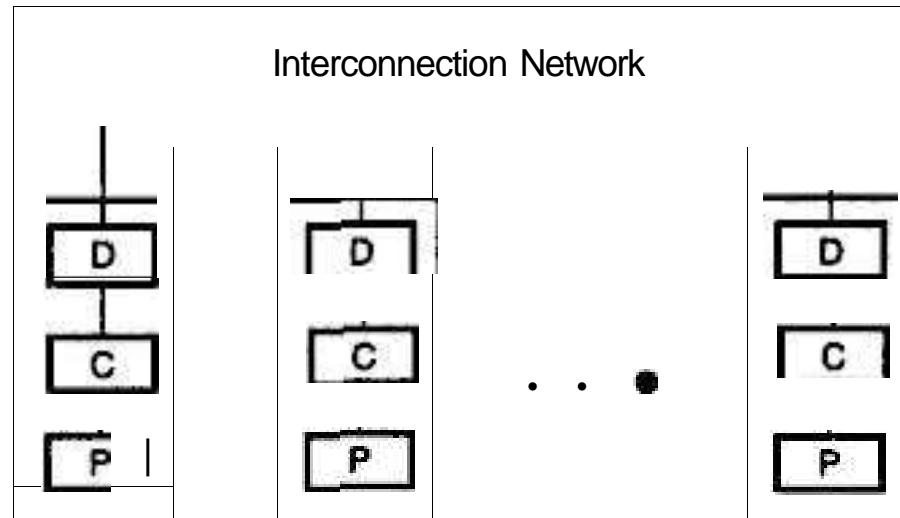


Figure 1.8 The COMA model of a multiprocessor. (P; Processor, C: Cache, D: Directory; e.g., the **KSR-1**)

Representative Multiprocessors Several commercially available multiprocessors are summarized in Table 1.3. They represent four classes of multiprocessors. The Sequent Symmetry S81 belongs to a class called **minisupercomputers**. The IBM System/390 models are **high-end** mainframes, sometimes called near-supercomputers. The BBN TC-2000 represents the MPP class.

The **S-81** is a transaction processing multiprocessor consisting of 30 i386/i486 microprocessors tied to a common backplane bus. The IBM **ES/9000** models are the latest IBM mainframes having up to 6 processors with attached vector facilities. The TC-2000 can be configured to have 512 M88100 processors interconnected by a multistage Butterfly network. This is designed as a **NUMA** machine for real-time or time-critical applications.

Multiprocessor systems are suitable for general-purpose multiuser applications where programmability is the major concern. A major shortcoming of multiprocessors is the lack of scalability. It is rather difficult to build MPP machines using centralized shared-memory model. Latency tolerance for remote memory access is also a major limitation.

Packaging and cooling impose additional constraints on scalability. We will study scalability and programmability **in subsequent** chapters. In building MPP systems, distributed-memory multicompilers are more scalable but less programmable due to added communication protocols.

1.2.2 Distributed-Memory Multicomputers

A distributed-memory multicomputer system is modeled in Fig. 1.9. The system consists of multiple computers, often called **nodes**, interconnected by a message-passing network. Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or **I/O** peripherals.

The **message-passing** network provides point-to-point static connections among the nodes. All local memories are private and are accessible only by local processors. For this reason, traditional multicomputers have been called **no-remote-memory-access** (NORMA) machines. However, this restriction **will** gradually be removed in future **mul-**

multicomputers with distributed shared memories. Internode communication is carried out by passing messages through the static connection network.

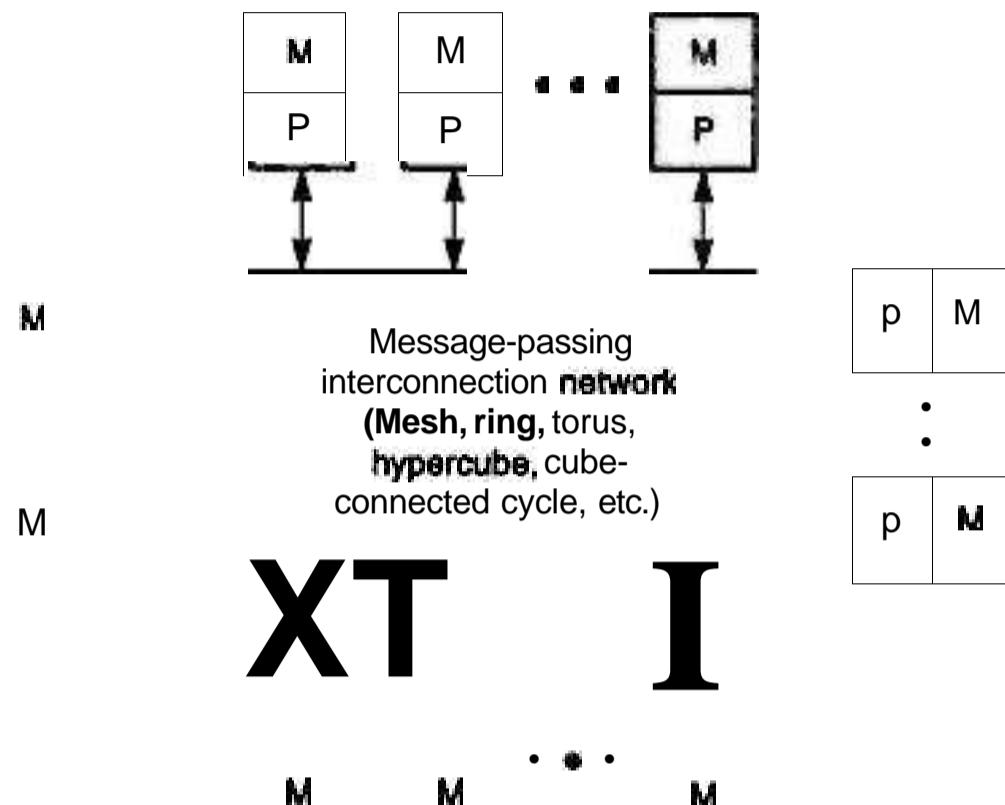


Figure 1.9 Generic model of a message-passing multicomputer.

Multicomputer Generations Modern multicomputers use hardware routers to pass messages. A computer node is attached to each router. The boundary router may be connected to **I/O** and peripheral devices. Message passing between any two nodes involves a sequence of routers and channels. Mixed types of nodes are allowed in a heterogeneous multicomputer. The internode communications in a heterogeneous multicomputer are achieved through compatible data representations and **message-passing** protocols.

Message-passing multicomputers have gone through two generations of development, and a new generation is emerging.

The *first generation (1983-1987)* was based on processor board technology using hypercube architecture and **software-controlled** message switching. The Caltech Cosmic and Intel **iPSC/1** represented the first-generation development.

The *second generation (1988-1992)* was implemented with mesh-connected architecture, hardware message routing, and a software environment for medium-grain **distributed** computing, as represented by the Intel Paragon and the Parsys SuperNode 1000.

The emerging *third generation (1993-1997)* is expected to be fine-grain multicomputers, like the MIT J-Machine and Caltech Mosaic, implemented with both processor and communication gears on the same VLSI chip.

In Section 2.4, we will study various static network topologies used to construct

multicomputers. Famous topologies include the *ring*, *tree*, *mesh*, *torus*, **hypercube**, *cube-connected cycle*, etc. Various communication patterns **are** demanded among the nodes, such as **one-to-one**, broadcasting, permutations, and multicast patterns.

Important issues **for** multicomputers include **message-routing** schemes, network flow control strategies, deadlock **avoidance**, virtual channels, message-passing primitives, and program decomposition techniques. In Part IV, we will study the programming issues of three generations of multicomputers.

Representative Multicomputers Three **message-passing** multicomputers are summarized in Table 1.4. With distributed processor/memory nodes, these machines are better in achieving a scalable performance. However, message passing imposes a hardship on programmers to distribute the computations and data sets over the nodes or to establish efficient communication among nodes. Until intelligent compilers and efficient distributed OSs **become available**, multicomputers will continue to lack programmability.

Table 1-4 Representative Multicomputer Systems

System Features	Intel Paragon XP/S	nCUBE/2 6480	Parsys Ltd. SuperNode1000
Node Types and Memory	50 MHz i860 XP computing nodes with 16-128 Mbytes per node, special I/O service nodes.	Each node contains a CISC 64-bit CPU , with FPU, 14 DMA ports , with 1-64 Mbytes/node.	EC-funded Esprit supernode built with multiple T-800 Transputers per node.
Network and I/O	2-D mesh with SCSI, HIPPI, VME, Ethernet, and custom I/O .	13-dimensional hypercube of 8192 nodes, 512-Gbyte memory, 64 I/O boards.	Reconfigurable interconnect, expandable to have 1024 processors*
OS and Software task parallelism Support	OSF conformance with 4.3 BSD , visualization and programming support.	Vertex/OS or UNIX supporting message passing using wormhole routing.	IDRIS/OS is UNIX-compatible , supported.
Application Drivers	General sparse matrix methods , parallel data manipulation, strategic computing .	Scientific number crunching with scalar nodes , database processing .	Scientific and academic applications.
Performance Remarks	5-300 Gflops peak 64-bit results , 2.8-160 GIPS peak integer performance .	27 Gflops peak, 36 Gbytes/s I/O , challenge to build even larger machine .	200 MIPS to 13 GIPS peak, largest supernode in use contains 256 Transputers,

The Paragon system assumes a mesh architecture, and the nCUBE/2 has a hy-

percube architecture. The Intel i860s and some custom-designed VLSI processors are used as building blocks in these machines. All three OSs are UNIX-compatible with extended functions to support message passing.

Most multicomputers are being upgraded to yield a higher degree of parallelism with enhanced processors. We will study various massively parallel systems in Part III where the tradeoffs between scalability and programmability are analyzed.

1.2.3 A Taxonomy of MIMD Computers

Parallel computers appear as either SIMD or MIMD configurations. The SIMDs appeal more to special-purpose applications. It is clear that SIMDs are not size-scalable, but unclear whether large SIMDs are generation-scalable. The fact that CM-5 has an MIMD architecture, away from the SIMD architecture in CM-2, may shed some light on the architectural trend. Furthermore, the boundary between multiprocessors and multicomputers has become blurred in recent years. Eventually, the distinctions may vanish.

The architectural trend for future general-purpose computers is in favor of MIMD configurations with distributed memories having a globally shared virtual address space. For this reason, Gordon Bell (1992) has provided a taxonomy of MIMD machines, reprinted in Fig. 1.10. He considers shared-memory multiprocessors as having a single address space. Scalable multiprocessors or multicomputer must use distributed shared memory. Unscalable multiprocessors use centrally shared memory.

Multicomputer use distributed memories with multiple address spaces. They are scalable with distributed memory. Centralized multicomputer are yet to appear. Many of the identified example systems will be treated in subsequent chapters. The evolution of fast LAN (*local area network*)-connected workstations will create "commodity supercomputing". Bell advocates high-speed workstation clusters interconnected by high-speed switches in lieu of special-purpose multicomputer. The CM-5 development has already moved in this direction.

The scalability of MIMD computers will be further studied in Section 3.4 and Chapter 9. In Part III, we will study distributed-memory multiprocessors (KSR-1, SCI, etc.); central-memory multiprocessor (Cray, IBM, DEC, Fujitsu, Encore, etc.); multicomputer by Intel, TMC, and nCUBE; fast LAN-based workstation clusters; and other exploratory research systems.

1-3 Multivector and SIMD Computers

In this section, we introduce supercomputers and parallel processors for vector processing and data parallelism. We classify supercomputers either as pipelined vector machines using a few powerful processors equipped with vector hardware, or as SIMD computers emphasizing massive data parallelism.

1.3.1 Vector Supercomputers

A vector computer is often built on top of a scalar processor. As shown in Fig. 1.11, the vector processor is attached to the scalar processor as an optional feature. Program

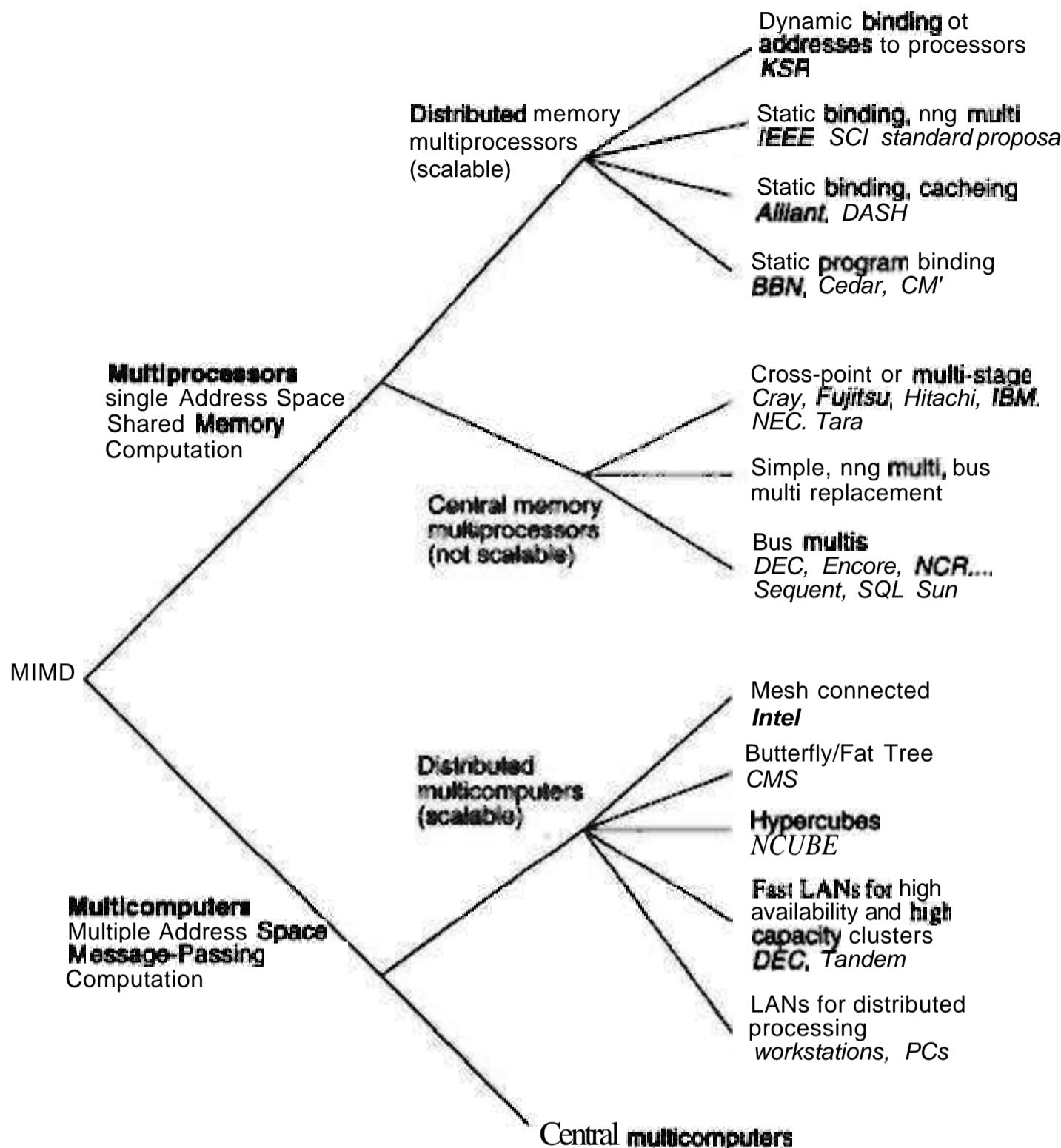


Figure 1.10 Bell's taxonomy of MIMD computers. (Courtesy of Gordon Bell; reprinted with permission from the *Communications of ACM*, August 1992)

and data are first loaded into the **main** memory through a host **computer**. All **instructions** are first decoded by the scalar control **unit**. If the decoded instruction is a scalar operation or a program control operation, it will be directly executed by the scalar processor using the scalar functional pipelines.

If the instruction is decoded as a vector operation, it will be sent to the vector control unit. This control unit will supervise the flow of vector data between the main memory and vector functional pipelines. The vector data flow is coordinated by the control unit. A number of vector functional pipelines may be built into a vector processor. Two pipeline vector supercomputer models are described below.

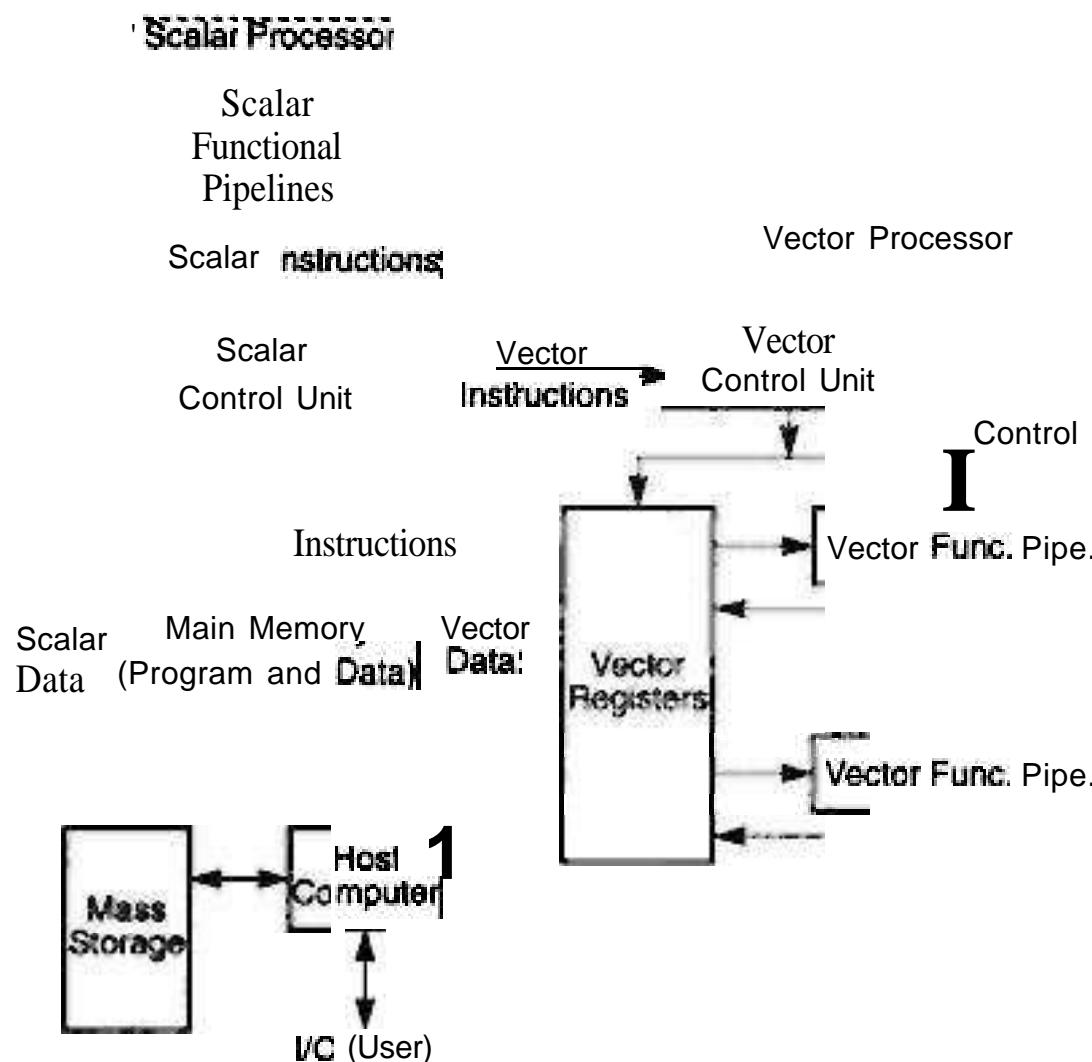


Figure 1.11 The architecture of a vector supercomputer.

Vector Processor Models Figure 1.11 shows a **register-to-register** architecture. Vector registers are used to hold the vector operands, intermediate and **final** vector results. The vector **functional** pipelines retrieve operands from and put results into the vector registers. All vector registers are programmable in user instructions. Each vector register is equipped with a component counter which keeps track of the component registers used in successive pipeline cycles.

The length of each vector register is usually fixed, say, **sixty-four** 64-bit component registers in a vector register in a Cray Series supercomputer. Other machines, like the Fujitsu **VP2000** Series, use reconfigurable vector registers to dynamically match the register length with that of the vector operands.

In general, there are fixed numbers of vector registers and functional pipelines in a vector processor. Therefore, both resources must be reserved in advance to avoid resource conflicts between different vector operations. Several vector-register based supercomputers are summarized in Table 1.5.

A **memory-to-memory** architecture differs from a **register-to-register** architecture in the use of a vector stream unit to replace the vector registers. Vector operands and results are directly retrieved from the main memory in superwords, say, 512 bits as in the Cyber 205.

Pipelined vector supercomputers started with uniprocessor models such as the Cray 1 in 1976. Recent supercomputer systems offer both uniprocessor and multiprocessor models such as the Cray Y-MP Series. Most high-end mainframes offer multiprocessor

models with add-on vector hardware, such as the VAX 9000 and IBM 390/VF models.

Representative Supercomputers Over a dozen pipelined vector computers have been manufactured, ranging from workstations to mini- and supercomputers. Notable ones include the Stardent 3000 multiprocessor equipped with vector pipelines, the Convex C3 Series, the DEC VAX 9000, the IBM 390/VF, the Cray Research Y-MP family, the NEC SX Series, the Fujitsu VP2000, and the Hitachi S-810/20.

Table 1.5 Representative Vector Supercomputers

System Model	Vector Hardware Architecture and Capabilities	Compiler and Software Support
Convex C3800 family	GaAs-based multiprocessor with 8 processors and 500-Mbyte/s access port- 4 Gbytes main memory 2 Gflops peak performance with concurrent scalar/vector operations .	Advanced C , Fortran, and Ada vectorizing and parallelizing compilers . Also support inter-procedural optimization , POSIX 1003.1/OS plus I/O interfaces and visualization system
Digital VAX 9000 System	Integrated vector processing in the VAX environment, 125–500 Mflops peak performance. 63 vector instructions. 16 x 64 x 64 vector registers- Pipeline chaining possible.	MS or ULTRIX/OS , VAX Fortran and VAX Vector Instruction Emulator (VVIEF) for vectorized program debugging.
Cray Research Y-MP and C-90	Y-MP runs with 2, 4, or 8 processors, 2.67 Gflop peak with Y-MP8256 . C-90 has 2 vector pipes/CPU built with 10K gate ECL with 16 Gflops peak performance .	CF77 compiler for automatic vectorization , scalar optimization , and parallel processing . UNICOS improved from UNIX/V and Berkeley BSD/OS.

The Convex C1 and C2 Series were made with ECL/CMOS technologies. The latest C3 Series is based on GaAs technology.

The DEC VAX 9000 is Digital's largest mainframe system providing concurrent **scalar/vector** and multiprocessing capabilities. The VAX 9000 processors use a hybrid architecture. The vector unit is an optional feature attached to the VAX 9000 CPU. The Cray Y-MP family offers both vector and multiprocessing capabilities.

1.3.2 SIMD Supercomputers

In Fig. 1.3b, we have shown an abstract model of SIMD computers having a single instruction stream over multiple data streams. An operational model of SIMD

computers is presented below (Fig. 1.12) based on the work of H. J. Siegel (1979). Implementation models and case studies of SIMD machines are given in Chapter 8.

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

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

where

- (1) N is the number of *processing elements* (PEs) in the machine. For example, the **Illiac IV** has 64 PEs and the Connection Machine CM-2 uses 65,536 PEs.
- (2) C is the set of instructions directly executed by the *control unit* (CU), including scalar and program flow control instructions.
- (3) I is the set of instructions broadcast by the CU to all PEs for parallel execution. These include arithmetic, logic, data routing, masking, and other local operations executed by each active PE over data within that PE.
- (4) M is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets.
- (5) R is the set of data-routing functions, specifying various patterns to be set up in the interconnection network for inter-PE communications.

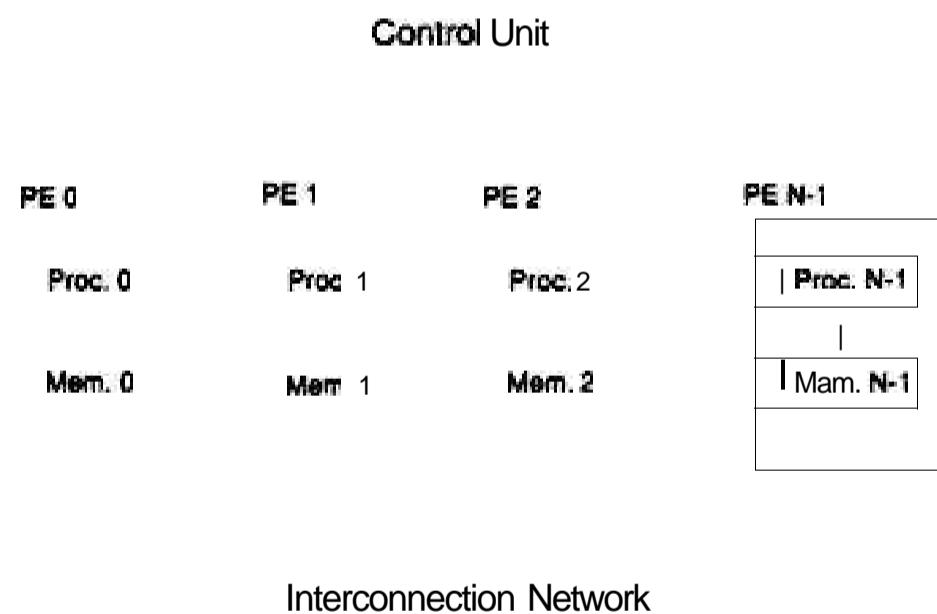


Figure 1.12 Operational model of SIMD computers.

One can **describe** a particular SIMD machine architecture by specifying the 5-tuple. An example SIMD machine is partially specified below:

Example 1.3 Operational specification of the **MasPar MP-1** computer

We will study the detailed architecture of the MasPar MP-1 in Chapter 7. Listed below is a partial specification of the 5-tuple for this machine:

- (1) The **MP-1** is an **SIMD** machine with $N = 1024$ to **16,384** PEs, depending on which configuration is considered.
- (2) The **CU** executes scalar instructions, broadcasts decoded vector instructions to the PE **array**, and controls the inter-PE communications.
- (3) Each PE is a register-based load/store RISC processor capable of executing integer operations over various data sizes and standard floating-point operations. The PEs receive instructions from the CU.
- (4) The masking scheme is built within each PE and continuously monitored by the CU which can set and reset the status of each PE dynamically at run time.
- (5) The MP-1 has an **X-Net** mesh network plus a global multistage crossbar router for inter-CU-PE, X-Net nearest 8-neighbor, and global router communications.

Representative SIMD Computers Three SIMD supercomputers are summarized in Table 1.6. The number of PEs in these systems ranges from 4096 in the DAP610 to 16,384 in the MasPar MP-1 and 65,536 in the CM-2. Both the CM-2 and DAP610 are **fine-grain**, bit-slice SIMD computers with attached floating-point accelerators for blocks of PEs.

Each PE of the MP-1 is equipped with a 1-bit logic unit, 4-bit integer ALU, 64-bit mantissa **unit**, and 16-bit exponent unit. **Therefore**, the MP-1 is a medium-grain SIMD machine. Multiple PEs can be built on a single chip due to the simplicity of each PE. The MP-1 implements 32 PEs per chip with forty 32-bit registers per PE. The 32 PEs are interconnected by an **X-Net mesh**, which is a 4-neighbor mesh augmented with diagonal dual-stage links.

The CM-2 implements 16 PEs as a mesh on a single chip. Each **16-PE** mesh chip is placed at one vertex of a **12-dimensional** hypercube. Thus $16 \times 2^{12} = 2^{16} = 65,536$ PEs form the entire SIMD array.

The DAP610 implements 64 PEs as a mesh on a chip. Globally, a large mesh (64×64) is formed by interconnecting these small meshes on chips. Fortran 90 and modified versions of C, Lisp, and other synchronous programming languages have been developed to program SIMD machines.

1.4 PRAM and VLSI Models

Theoretical models of parallel computers are abstracted from the physical models studied in previous sections. These models are often used by algorithm designers and VLSI device/chip developers. The ideal models provide a convenient framework for developing parallel algorithms without worry about the implementation details or physical constraints.

The models can be applied to obtain theoretical performance bounds on parallel computers or to estimate VLSI complexity on chip area and execution time before **the**

Table 1.6 Representative SIMD Supercomputers

System Model	SIMD Machine Architecture and Capabilities	Languages, Compilers and Software Support
MasPar Computer Corporation MP-1 Family	Available in configurations from 1024 to 16,384 processors with 26,000 MIPS or 1.3 Gflops . Each PE is a RISC processor , with 16 Kbytes local memory. An X-Net mesh plus a multistage crossbar interconnect.	Fortran 77, MasPar Fortran (MPF) , and MasPar Parallel Application Language; UNIX/OS with X-window, symbolic debugger, visualizers and animators .
Thinking Machines Corporation, CM-2	A bit-slice array of up to 65,536 PEs arranged as a 10-dimensional hypercube with 4×4 mesh on each vertex, up to 1M bits of memory per PE, with optional FPU shared between blocks of 32 PEs . 28 Gflops peak and 5.6 Gflops sustained .	Driven by a host of VAX, Sun, or Symbolics 3600, Lisp compiler, Fortran 90, C* , and *Lisp supported by PARIS
Active Memory Technology DAP600 Family	A fine-grain, bit-slice SIMD array of up to 4096 PEs interconnected by a square mesh with 1K bits per PE , orthogonal and 4-neighbor links, 20 GIPS and 560 Mflops peak performance .	Provided by host VAX/VMS or UNIX Fortran-plus or APAL on DAP, Fortran 77 or C on host. Fortran-plus affected Fortran 90 standards .

chip is fabricated. The abstract models are also useful in scalability and programmability analysis, when real machines are compared with an idealized parallel machine without worrying about communication overhead among processing nodes.

1.4.1 Parallel Random-Access Machines

Theoretical models of parallel computers are presented below. We define first the time and space complexities. Computational tractability is reviewed for solving difficult problems on computers. Then we introduce the *random-access machine* (RAM), *parallel random-access machine* (PRAM), and variants of PRAMs. These complexity models facilitate the study of asymptotic behavior of algorithms implementable on parallel computers.

Time and Space Complexities The complexity of an algorithm for solving a **problem** of size s on a computer is determined by the execution time and the storage space **required**. The *time complexity* is a function of the problem size. The time complexity function in order notation is the *asymptotic time complexity* of the algorithm. Usually, the worst-case time complexity is considered. For example, a time complexity $g(s)$ is said to be $O(f(s))$, read "order $f(s)$ ", if there exist positive constants c and s_0 such that $g(s) < cf(s)$ for all **nonnegative** values of $s > s_0$.

The *space complexity* can be similarly defined as a function of the problem size s . The *asymptotic space complexity* refers to the data storage of large **problems**. Note that the program (code) storage requirement and the storage for input data are not considered in this.

The time complexity of a **serial** algorithm is simply called *serial complexity*. The time complexity of a parallel algorithm is called *parallel complexity*. Intuitively, the parallel complexity should be lower than the serial complexity, at least asymptotically. We consider only *deterministic algorithms*, in which every operational step is uniquely defined in agreement with the way programs are executed on real computers.

A *nondeterministic algorithm* contains operations resulting in one outcome in a set of possible outcomes. There exist no real computers that can execute nondeterministic algorithms. Therefore, all algorithms (or machines) considered in this book are deterministic, unless otherwise noted.

NP-Completeness An algorithm has a *polynomial complexity* if there exists a polynomial $p(s)$ such that the time complexity is $O(p(s))$ for any problem size s . The set of problems having polynomial-complexity algorithms is called **P-class** (for polynomial class). The set of problems solvable by nondeterministic algorithms in polynomial time is called **NP-class** (for nondeterministic polynomial class).

Since deterministic algorithms are special cases of the nondeterministic ones, we know that $P \subseteq NP$. The **P-class** problems are computationally **tractable**, while the $NP - P$ -class problems are **intractable**. But we do not know whether $P = NP$ or $P \neq NP$. This is still an open problem in computer science.

To simulate a nondeterministic algorithm with a deterministic algorithm may require exponential time. Therefore, intractable NP-class problems are also said to have exponential-time complexity.

Example 1.4 Polynomial- and exponential-complexity algorithms

Polynomial-complexity algorithms are known for sorting n numbers in $O(n \log n)$ time and for multiplication of two $n \times m$ matrices in $O(n^3)$ time. Therefore, both problems belong to the P-class.

Nonpolynomial algorithms have been developed for the traveling salesperson problem with complexity $O(n^2 2^n)$ and for the knapsack problem with complexity $O(2^{n/2})$. These complexities are **exponential**, greater than the polynomial complexities. So far, deterministic polynomial algorithms have not been found for these problems. Therefore, **exponential-complexity** problems belong to the NP-class.

Most computer scientists believe that $P \neq NP$. This leads to the conjecture that there exists a subclass, called **NP-complete (NPC)** problems, such that $NPC \subseteq NP$ but $NPC \cap P = \emptyset$ (Fig. 1.13). In fact, it has been proved that if any NP-complete problem is **polynomial-time** solvable, then one can conclude $P = NP$. Thus NP-complete problems are considered the hardest ones to solve. Only approximation algorithms were derived for solving some of the NP-complete problems.

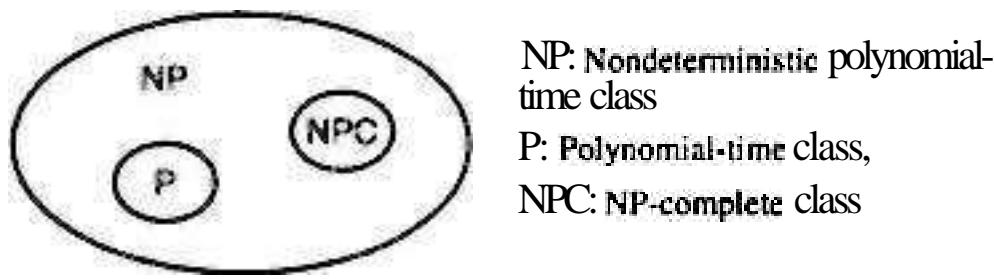


Figure 1.13 The relationships conjectured among the NP, P, and NPC **classes** of computational **problems**.

PRAM Models Conventional uniprocessor computers have been modeled as **random-access machines** (RAM) by Sheperdson and Sturgis (1963). A **parallel random-access machine** (PRAM) model has been developed by Fortune and Wyllie (1978) for modeling idealized parallel computers with zero synchronization or memory access overhead. This PRAM model will be used for parallel algorithm development and for scalability and complexity analysis.

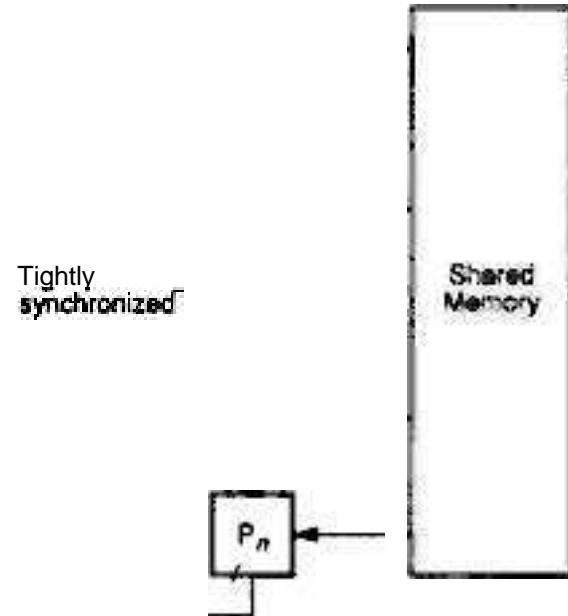


Figure 1.14 PRAM model of a multiprocessor system with shared memory, on which all n processors operate in **lockstep** in memory access and program execution operations. **Each** processor can access any memory location in unit time.

An **n-processor** PRAM (Fig. 1.14) has a globally addressable memory. The shared memory can be distributed among the processors or centralized in one place. The n processors [also called *processing elements* (PEs) by other authors] operate on a synchronized read-memory, **compute**, and **write-memory** cycle. With shared memory, **the** model must specify how concurrent read and concurrent write of memory are handled. Four memory-update options are possible:

Exclusive read (ER) — This allows at most one processor to read from any memory location in each cycle, a rather restrictive policy.

- *Exclusive write* (EW) — This allows at most one processor to write into a memory location at a time.
- *Concurrent read* (CR) — This allows multiple processors to read the same information from the same memory cell in the same cycle.
- *Concurrent write* (CW) — This allows simultaneous writes to the same memory location. In order to avoid confusion, some policy must be set up to resolve the write conflicts.

Various combinations of the above options lead to several variants of the PRAM model as specified below. Since CR does not create a **conflict** problem, variants differ mainly in how they handle the CW conflicts.

PRAM Variants Described below are four variants of the PRAM model, depending on how the memory reads and writes are handled.

- (1) The *EREW-PRAM model* — This model forbids more than one processor from reading or writing the same memory cell simultaneously (Snir, 1982; Karp and Ramachandran. 1988). This is the most restrictive PRAM model proposed.
- (2) The *CREW-PRAM model* — The write conflicts are avoided by mutual exclusion. Concurrent reads to the same memory location are allowed.
- { 3) The *ERCW-PRAM model* — This allows exclusive read or concurrent writes to the same memory location.
- (4) The *CRCW-PRAM model* — This model allows either concurrent reads or concurrent writes at the same time. The conflicting writes are resolved by one of the following four policies (Fortune and Wyllie, 1978):
 - *Common* — All simultaneous writes store the same value to the hot-spot memory location.
 - *Arbitrary* — Any one of the values written may remain; the others are ignored.
 - *Minimum* — The value written by the processor with the minimum index will remain.
 - *Priority* — The values being written are combined using some associative functions, such as summation or maximum.

Example 1.5 Multiplication of two $n \times n$ matrices in $O(\log n)$ time on a PRAM with $n^3/\log n$ processors (Viktor Prasanna, 1992)

Let A and B be the input matrices. Assume n^3 PEs are available initially. We later reduce the number of PEs to $n^3/\log n$. To visualize the algorithm, assume the memory is organized as a **three-dimensional** array with inputs A and B stored in two planes. Also, for sake of explanation, assume a three-dimensional indexing of the PEs. PE(i,j,k), $0 \leq k \leq n - 1$ are used for computing the (i,j)th entry of the output **matrix**, $0 < i, j < n - 1$, and n is a power of 2.

In step 1, n product terms corresponding to each output are computed using n PEs in $O(1)$ time. In step 2, these are added to produce an output in $O(\log n)$ time.

The total number of PEs used is n^3 . The result is available in $C(i,j,0), 0 < i,j < n-1$. Listed below are programs for each $PE(i,j,k)$ to execute. All n^3 PEs operate in parallel for n^3 multiplications. But at most $n^3/2$ PEs are busy for $(n^3 - n^2)$ additions. Also, the PRAM is assumed to be CREW.

Step 1:

1. Read $A(i,k)$
2. Read $B(k,j)$
3. Compute $A(i,k) \times B(k,j)$
4. Store in $C(i,j,k)$

Step 2:

1. $\ell \leftarrow n$
2. Repeat
 - $\ell \leftarrow \ell/2$
 - if ($k < t$) then
 - begin
 - Read $C(i,j,k)$
 - Read $C(i,j,k + \ell)$
 - Compute $C(i,j,k) + C(i,j,k + \ell)$
 - Store in $C(i,j,k)$
 - end
 - until ($\ell = 1$)

To reduce the number of PEs to $n^3/\log n$, use a PE array of size $n \times n \times n/\log n$. Each PE is responsible for computing $\log n$ product terms and summing them up. Step 1 can be easily modified to produce $n/\log n$ partial sums, each consisting of $\log n$ multiplications and $(\log n - 1)$ additions. Now we have an array $C(i,j,k)$, $0 < i,j < n-1, 0 < k < n/\log n - 1$, which can be summed up in $\log(n/\log n)$ time. Combining the time spent in step 1 and step 2, we have a total execution time $2\log n - 1 + \log(n/\log n) \approx O(\log n)$ for large n .

Discrepancy with Physical Models PRAM models idealized parallel computers, in which all memory references and program executions by multiple processors are synchronized without extra cost. In reality, such parallel machines do not exist. An **SIMD** machine with shared memory is the closest architecture modeled by PRAM. However, PRAM allows different instructions to be executed on different processors simultaneously. Therefore, PRAM really operates in synchronized MIMD mode with a shared memory.

Among the four PRAM variants, the EREW and CRCW are the most popular models used. In fact, every CRCW algorithm can be simulated by an EREW **algorithm**. The CRCW algorithm runs faster than an equivalent EREW algorithm. It has been proved that the best n -processor EREW algorithm can be no more than $O(\log n)$ times slower than any n -processor CRCW algorithm.

The CREW model has received more attention in the literature than the **ERCW** model. The CREW models **MISD** machines, which have attracted little attention. For our purposes, we will use the CRCW-PRAM model unless otherwise stated. This particular model will be used in defining scalability in Chapter 3.

For complexity analysis or performance comparison, various PRAM variants offer an ideal model of parallel computers. Therefore, computer scientists use the PRAM model more often than computer engineers. In this book, we design **parallel/vector computers** using physical architectural models rather than PRAM models.

The PRAM model will be used for scalability and performance studies in Chapter 3 as a theoretical reference machine. For regularly **structured** parallelism, the PRAM can model much better than practical machine models. Therefore, sometimes PRAMs can indicate an upper bound on the performance of real parallel computers.

1.4.2 VLSI Complexity Model

Parallel computers rely on the use of VLSI chips to fabricate the major components such as processor arrays, memory arrays, and large-scale switching networks. An AT^2 model for two-dimensional VLSI chips is presented below, based on the work of Clark Thompson (1980). Three lower bounds on VLSI circuits are interpreted by Jeffrey Ullman (1984). The bounds are obtained by setting limits on memory, **I/O**, and communication for implementing parallel algorithms with VLSI chips.

The AT^2 Model Let A be the chip area and T be the latency for completing a given computation using a VLSI circuit chip. Let s by the problem size involved in the computation. Thompson stated in his doctoral thesis that for certain computations, there exists a lower bound $f(s)$ such that

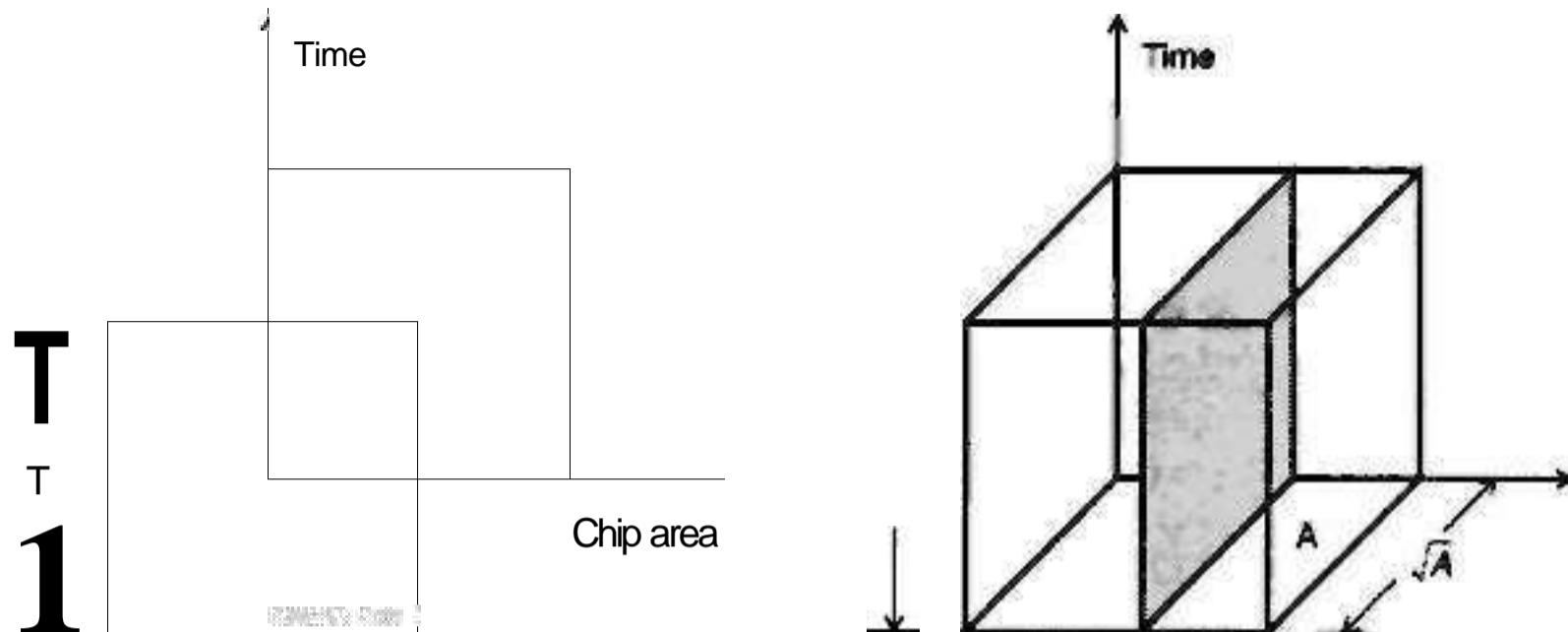
$$A \times T^2 > O(f(s)) \quad (1.6)$$

The chip area A is a measure of the chip's complexity. The latency T is the time required from when inputs are applied until all outputs are produced for a single problem instance. Figure 1.15 shows how to interpret the AT^2 complexity results in VLSI chip development. The chip is represented by the base area in the two horizontal dimensions. The vertical **dimension** corresponds to time. **Therefore**, the three-dimensional solid represents the history of the computation performed by the chip.

Memory Bound on Chip Area A There are many computations which are memory-bound, due to the need to process large data sets. To implement this type of computation in silicon, one is limited by how densely information (**bit** cells) can be placed on the chip. As depicted in Fig. 1.15a. the memory requirement of a computation sets a lower bound on the chip area A .

The amount of information processed by the chip can be visualized as information flow upward across the chip area. Each bit can flow through a unit area of the horizontal chip slice. Thus, the chip area bounds the amount of memory bits stored on the chip.

I/O Bound on Volume AT The volume of the rectangular cube is represented by



(a) Memory-limited bound on chip area A and I/O-limited bound on chip history represented by the volume AT

(b) Communication-limited bound on the bisection VAT

Figure 1.15 The **AT** complexity model of **two-dimensional** VLSI chips.

the product AT . As **information** flows through the chip for a period of time T , the number of input bits cannot exceed the volume. This provides an I/O-limited lower bound on the product AT , as demonstrated in Fig. 1.15a.

The area A corresponds to data into and out of the entire surface of the silicon chip. This **areal** measure sets the maximum **I/O** limit rather than using the peripheral **I/O** pads as seen in conventional chips. The height T of the volume can be visualized as a number of snapshots on the chip, as computing time elapses. The volume represents the amount of information flowing through the chip during the entire course of the computation.

Bisection Communication Bound, \sqrt{AT} Figure 1.15b depicts a communication limited lower bound on the bisection area \sqrt{AT} . The bisection is represented by the vertical slice cutting across the shorter dimension of the chip **area**. The distance of this dimension is at most \sqrt{A} for a square chip. The height of the cross section is T .

The bisection area represents the maximum amount of information exchange between the two halves of the chip circuit during the time period T . The **cross-section** area vAT limits the communication bandwidth of a computation. VLSI complexity theoreticians have used the square of this measure, AT^2 , as the lower bound.

Charles Seitz (1990) has given another interpretation of the AT^2 result. He considers the area-time product AT the cost of a computation, which can be expected to vary as $1/T$. This implies that the cost of computation for a two-dimensional chip decreases with the execution time allowed.

When **three-dimensional** (multilayer) silicon chips are used, Seitz asserted that the

cost of computation, as limited by volume-time product, would vary as $1/\sqrt{T}$. This is due to the fact that the bisection will vary as $(AT)^{2/3}$ for 3-D chips instead of as \sqrt{AT} for 2-D chips.

Example 1.6 VLSI chip implementation of a matrix multiplication algorithm (Viktor Prasanna, 1992)

This example shows how to estimate the chip area A and compute time T for $n \times n$ matrix multiplication $C = A \times B$ on a mesh of processing elements (PEs) with a broadcast bus on each row and each column. The 2-D mesh architecture is shown in Fig. 1.16. **Inter-PE communication** is done through the broadcast buses. We want to prove the bound $AT^7 = O(n^4)$ by developing a parallel matrix multiplication algorithm with time $T = O(n)$ in using the mesh with broadcast buses. Therefore, we need to prove that the chip area is bounded by $A = O(n^2)$.

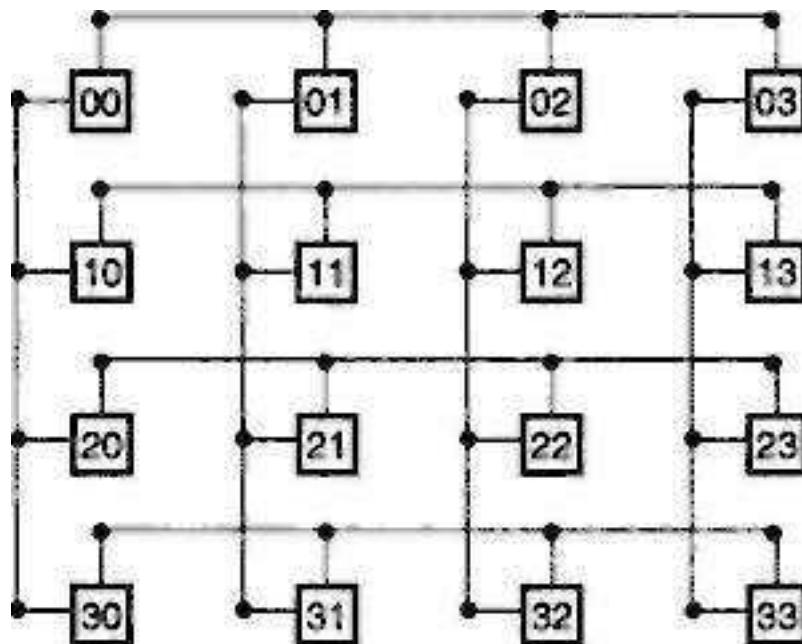


Figure 1.16 A 4×4 mesh of processing elements (PEs) with broadcast buses on each row and on each column. (Courtesy of Prasanna Kumar and Raghavendra; reprinted from *Journal of Parallel and Distributed Computing*, April 1987)

Each PE occupies a unit area, and the broadcast buses require $O(n^2)$ wire area. Thus the total chip area needed is $O(n^2)$ for an $n \times n$ mesh with broadcast buses. We show next that the $n \times n$ matrix multiplication can be performed on this mesh chip in $T = O(n)$ time. Denote the PEs as $PE(i,j)$, $0 < i, j < n - 1$.

Initially the input matrix elements $A(i,j)$ and $B(i,j)$ are stored in $PE(i,j)$ with no duplicated data. The memory is distributed among all the PEs. Each PE can access only its own local memory. The following parallel algorithm shows how to perform the dot-product operations in generating all the output elements $C(i,j) = \sum_{k=0}^{n-1} A(i,k) \times B(k,j)$ for $0 < i, j < n - 1$.

Doall 10 for $0 \leq i, j < n - 1$
10 $PE(i,j)$ sets $C(i,j)$ to 0 /Initialization/



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

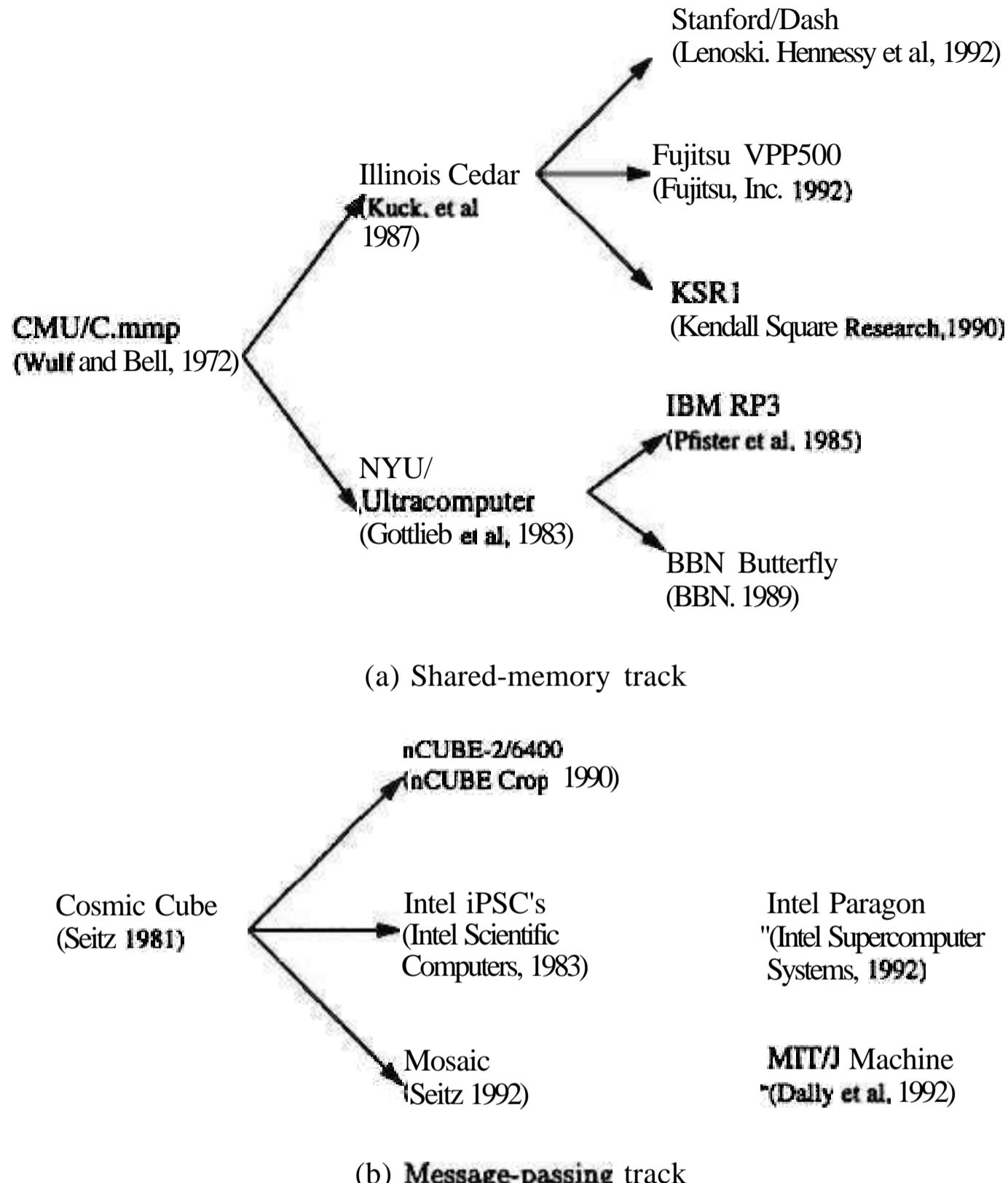


Figure 1.17 Two **multiple-processor** tracks with and without shared memory.

The **Stanford Dash** (Lenoski, Hennessy et al., 1992) is a **NUMA** multiprocessor with distributed memories forming a global address space. Cache coherence is enforced with distributed directories. The KSR-1 is a typical COMA model. The Fujitsu VPP 500 is a 222-processor system with a crossbar interconnect. The shared memories are **distributed** to all processor nodes- We will study the Dash and the KSR-1 in Chapter 9 and the VPP500 in Chapter 8.

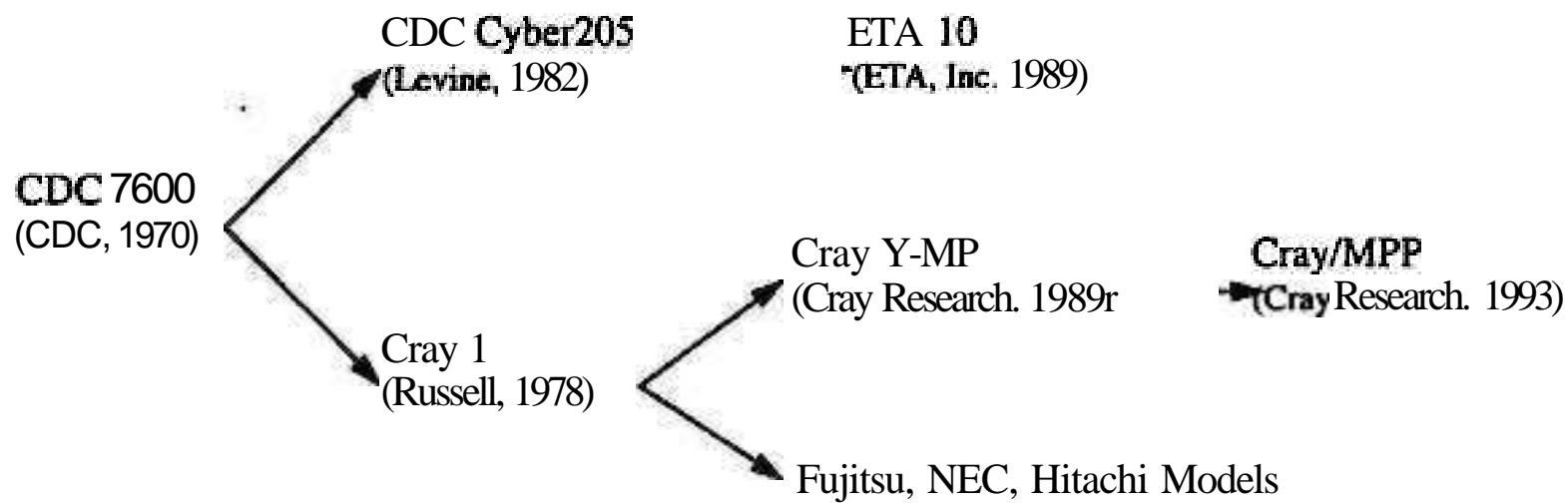
Following the Ultracomputer are two large-scale multiprocessors, both using multistage networks but with different interstage connections to be studied in Chapters 2 and 7. Among the systems listed in Fig. 1.17a, only the KSR-1, VPP500, and BBN Butterfly (BBN Advanced Computers, 1989) are commercial products. The rest are

research systems; only prototypes have been built in laboratories.

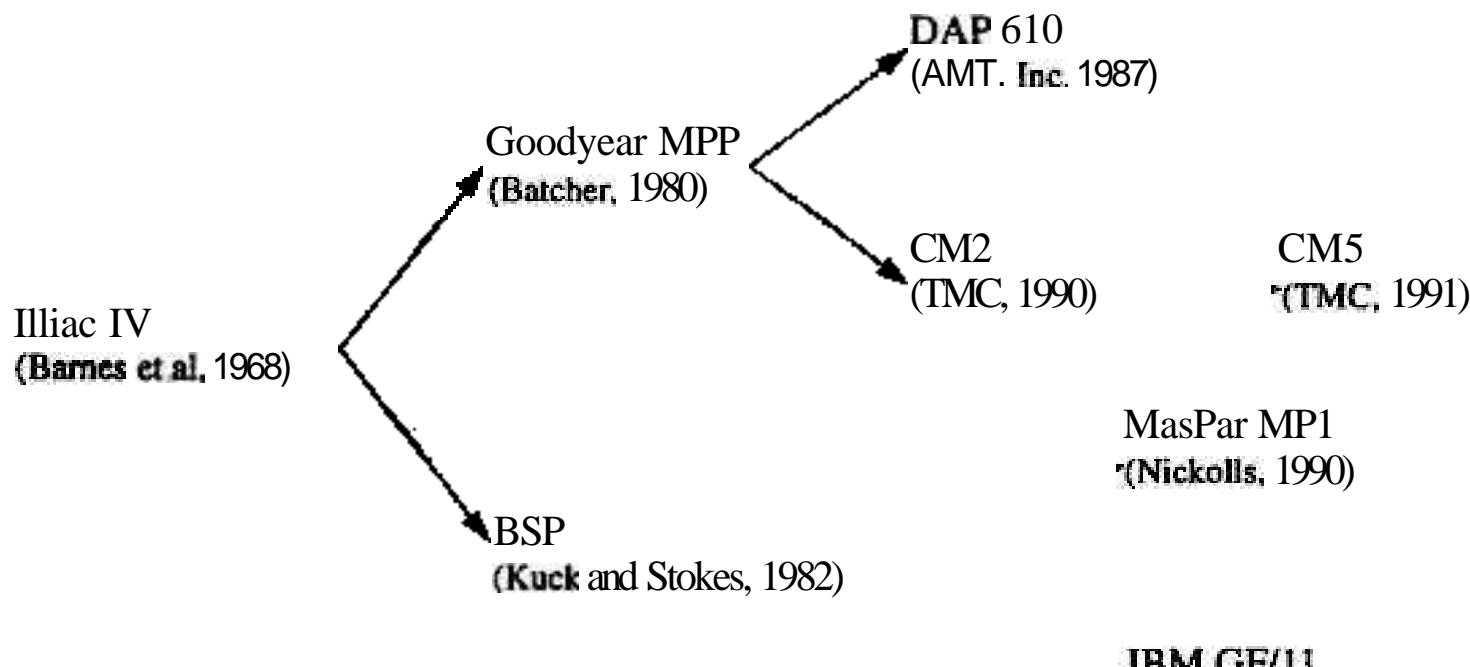
Message-Passing Track The Cosmic Cube (Seitz et al., 1981) pioneered the development of message-passing multicomputers (Fig. 1.17b). Since then, Intel has produced a series of medium-grain hypercube computers (the iPSCs). The nCUBE 2 also assumes a hypercube configuration. The latest Intel system is the Paragon (1992) to be studied in Chapter 7. On the research track, the Mosaic C (Seitz, 1992) and the **MIT J-Machine** (Dally et al., 1992) are two **fine-grain** multicomputers to be studied in Chapter 9.

1.5.2 Multivector and **SIMD** Tracks

The **multivector** track is shown in Fig. 1.18a, and the SIMD track in Fig. 1.18b.



(a) Multivector track



(b) SIMD track

Figure 1.18 Multivector and SIMD tracks.

Both tracks are used for concurrent scalar/vector processing. Detailed studies can be found in Chapter 8.

Multivector Track These are traditional vector supercomputers. The CDC 7600 was the first vector **dual-processor** system. Two subtracks were derived from the CDC 7600. The Cray and Japanese supercomputers all followed the register-to-register architecture. Cray 1 pioneered the multivector development in 1978. The latest Cray/MPP is a massively parallel system with distributed shared memory. It is supposed to work as a back-end accelerator engine **compatible** with the existing Cray Y-MP Series.

The other subtrack used **memory-to-memory** architecture in building vector supercomputers. We have identified only the CDC Cyber 205 and its successor the **ETA** 10 here. Since the production of both machines has been discontinued now, we list them here simply for completeness in tracking different supercomputer architectures.

The SIMD Track The Illiac IV pioneered the construction of **SIMD** computers, even the array processor concept can be traced back far earlier to the 1960s. The subtrack, consisting of the Goodyear MPP, the AMT/DAP610, and the TMC/CM-2, are all SIMD **machines** built with bit-slice PEs. The CM-5 is a synchronized **MIMD** machine executing in a **multiple-SIMD** mode.

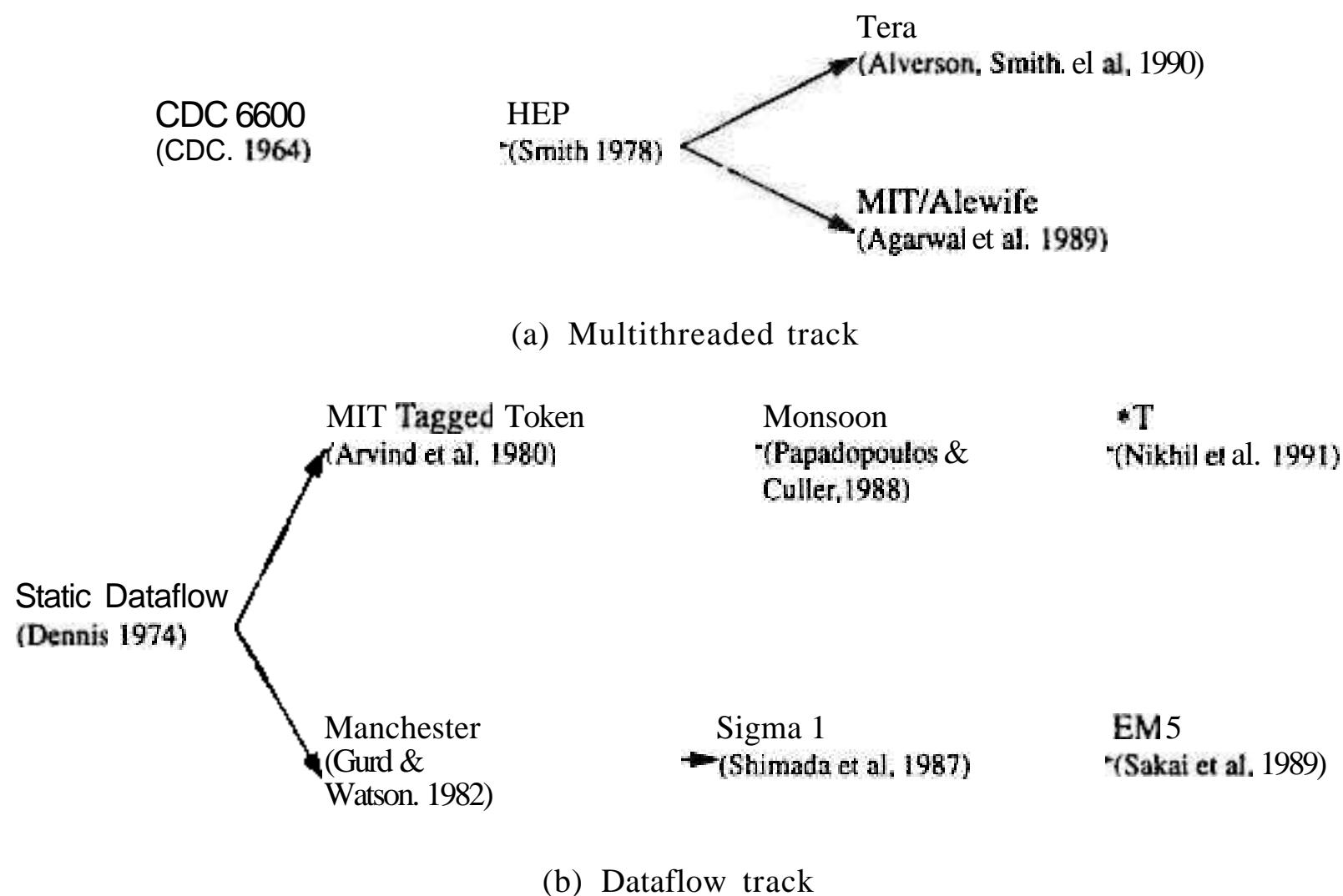
The other subtrack corresponds to medium-grain SIMD computers using word-wide PEs. The BSP (Kuck and Stokes, 1982) was a shared-memory SIMD machine built with 16 processors updating a group of 17 memory modules synchronously. The **GF11** (Beetem et al., 1985) was developed at the IBM Watson Laboratory for **scientific** simulation research use. The MasPar MP1 is the only medium-grain SIMD computer currently in production use. We will describe the CM-2, MasPar MP1, and CM-5 in Chapter 8.

1.5.3 Multithreaded and **Dataflow** Tracks

These are two research tracks (Fig. 1.19) that have been mainly experimented with in laboratories. Both tracks will be studied in Chapter 9. The following introduction covers only basic definitions and milestone systems built today.

The conventional von Neumann machines are built with processors that execute a single context by each processor at a time. In **other** words, each processor maintains a single thread of control with limited hardware resources. In a multithreaded **architecture**, each processor can execute multiple contexts at the same time. The term **multithreading** implies that there are multiple threads of control in each processor. Multithreading offers an effective **mechanism** for hiding long latency in building large-scale multiprocessors.

As shown in Fig. 1.19a, the multithreading idea was pioneered by Burton Smith (1978) in the HEP system which extended the concept of scoreboardng of multiple functional units in the CDC 6400. The latest multithreaded multiprocessor projects are the Tera computer (Alverson, Smith et al., 1990) and the MIT **Alewife** (Agarwal et al., 1989) to be studied in Section 9.4. Until then, all multiprocessors studied use single-threaded processors as building blocks.

Figure 1.19 Multithreaded and **dataflow** tracks.

The Dataflow Track We will introduce the basic concepts of **dataflow** computers in Section 2.3. Some experimental dataflow systems are described in Section 9.5. The key idea is to use a dataflow **mechanism**, instead of a control-flow **mechanism** as in von Neumann machines, to direct the program flow. **Fine-grain**, instruction-level parallelism is exploited in dataflow computers.

As listed in Fig. 1.19b, the dataflow concept was pioneered by Jack Dennis (1974) with a "static" architecture. The concept later inspired the development of "dynamic" dataflow computers. A series of tagged-token architectures was developed at MIT by Arvind and coworkers. We will describe the tagged-token architecture in Section 2v3.1 and then the *T prototype (Nikhil et al., 1991) in Section 9.5.3.

Another important subtrack of dynamic dataflow computer is **represented** by the Manchester machine (Gurd and Watson, 1982). The ETL **Sigma 1** (Shimada et al., 1987) and EM5 have evolved from the MIT and Manchester machines. We will study the EM5 (Sakai et al., 1989) in Section 9.5.2. These dataflow machines are still in the research stage.

1.6 Bibliographic Notes and Exercises

Various machine architectures were classified by [Flynn72j]. A recent collection of papers on architectural alternatives for exploiting parallelism can be found in the

tutorial [Lilja92]. [Almasi89] and Gottlieb provided a thorough survey of research done on parallel computing up to 1989. [Hwang89a] and DeGroot presented critical reviews on parallel processing techniques developed for supercomputers and artificial intelligence.

[Hennessy90] and Patterson provided an excellent treatment of uniprocessor computer design. For a treatment of earlier parallel processing computers, readers are referred to [Hwang84] and Briggs. The layered classification of parallel computers was proposed in [Ni91j]. [Bell92] introduced the **MIMD** taxonomy. Systolic array was introduced by [Kung78] and Leiserson. [Prasanna Kumar87] and Raghavendra proposed the mesh architecture with broadcast buses for parallel computing.

Multiprocessor issues were characterized by [Gajski85] and Pier and by [Dubois88], Scheurich, and Briggs. Multicomputer technology was assessed by [Athas88] and Seitz. An introduction to existing parallel computers can be found in [Trew91] and Wilson. Key references of various computer systems, listed in Bell's taxonomy (Fig. 1.10) and in different architectural development tracks (Figs. 1.17 through 1.19), are identified in the figures as well as in the bibliography. Additional references to these case-study machines can be found in later chapters. A collection of reviews, a bibliography, and indexes of resources in parallel systems can be found in [ACM91] with an introduction by Charles Seitz.

A comparative study of **NUMA** and COMA multiprocessor models can be found in [Stenström92], Joe, and Gupta. **SIMD** machines were modeled by [Siegel79]. The RAM model was proposed by [Sheperdson63] and Sturgis. The PRAM model and variations were studied in [Fortune78] and Wyllie, [Snir82], and [Karp88]. The theory of **NP**-completeness is covered in the book by [Cormen90], Leiserson, and Rivest. The VLSI complexity model was introduced in [Thompson80] and interpreted by [Ullman84] and by [Seitz90] subsequently.

Readers are referred to the following journals and conference records for information on recent developments:

- *Journal of Parallel and Distributed Computing* (Academic Press, since 1983).
- *Journal of Parallel Computing* (North Holland, Amsterdam, since 1984).
- *IEEE Transactions on Parallel and Distributed Systems* (IEEE Computer Society, since 1990).
- *International Conference on Parallel Processing* (Pennsylvania State University, since 1972).
- *International Symposium on Computer Architecture* (IEEE Computer Society, since 1972).
- *Symposium on the Frontiers of Massively Parallel Computation* (IEEE Computer Society, since 1986).
- *International Conference on Supercomputing* (ACM, since 1987).
- *Symposium on Architectural Support for Programming Languages and Operating Systems* (ACM, since 1975).
- *Symposium on Parallel Algorithms and Architectures* (ACM, since 1989).
- » *International Parallel Processing Symposium* (IEEE Computer Society, since 1986).

- *IEEE Symposium on Parallel and Distributed Processing* (IEEE Computer Society, since 1989).

Exercises

Problem 1.1 A 40-MHz processor was used to execute a benchmark program with the following instruction mix and clock cycle counts:

Instruction type	Instruction count	Clock cycle count
Integer arithmetic j	45000	1
Data transfer	32000	2
Floating point	15000	2
Control transfer	8000	2

Determine the effective **CPI**, MIPS rate, and execution time for this program.

Problem 1.2 Explain how instruction set, compiler technology, CPU implementation and control, and cache and memory hierarchy affect the CPU performance and justify the effects in terms of program length, clock **rate**, and effective **CPI**.

Problem 1.3 A workstation uses a 15-MHz processor with a claimed **10-MIPS** rating to execute a given program mix. Assume a **one-cycle** delay for each memory access.

- What is the effective CPI of this computer?
- Suppose the processor is being upgraded with a 30-MHz clock. However, the speed of the memory subsystem remains unchanged, and consequently two clock cycles are needed per memory access. If **30%** of the instructions require one memory access and another **5%** require two memory accesses per instruction, what is the performance of the upgraded processor with a compatible instruction set and equal instruction counts in the given program mix?

Problem 1.4 Consider the execution of an object code with 200,000 instructions on a **40-MHz** processor. The program consists of four major types of instructions. The instruction mix and the number of cycles (CPI) needed for each instruction type are given below based on the result of a program trace experiment:

Instruction type	CPI	Instruction mix
Arithmetic and logic	1	60%
Load/store with cache hit	2	18%
Branch	4	12%
Memory reference with cache miss	8	10%



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Problem 1.13 Design an algorithm to find the maximum of n numbers in $O(\log n)$ time on an EREW-PRAM model. Assume that initially each location holds one input value. Explain how you would make the algorithm processor time optimal.

Problem 1.14 Develop two **algorithms** for fast multiplication of two $n \times n$ matrices with a system of p processors, where $1 < p < n^3 / \log n$. Choose an appropriate PRAM machine model to prove that the matrix multiplication can be done in $T = O(n^3/p)$ time.

- (a) Prove that $T = O(n^2)$ if $p = n$. The corresponding **algorithm** must be shown, similar to that in Example 1.5.
- (b) Show the parallel algorithm with $X = O(n)$ if $p = n^2$.

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

- (a) A massively parallel system built with **multiple-context** processors and a 3-D torus architecture.
- (b) A data-parallel computer built with bit-slice PEs interconnected by a hypercube/mesh network.
- (c) A ring-connected multiprocessor using a cache-only memory architecture.
- (d) An experimental multiprocessor built with a dynamic dataflow architecture.
- (e) A **crossbar-connected** multiprocessor built with distributed processor/memory node** forming a single address space.
- (f) A multicomputer built with commercial microprocessors with multiple address spaces.
- (g) A scalable multiprocessor built with distributed shared memory and coherent caches.
- (h) An **MIMD** computer built with a large multistage switching network.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

```

Do 10 I = 1, N
    IF (A(I - 1) .EQ. 0) A(I) = 0
10 Continue

```

Control dependence often prohibits parallelism from being exploited. Compiler techniques are needed to get around the control dependence in order to exploit more parallelism.

Resource Dependence This is different from data or control dependence, which demands the independence of the work to be done. *Resource dependence* is concerned with the conflicts in using shared resources, such as integer units, floating-point units, registers, and memory areas, among parallel events. When the conflicting resource is an **ALU**, we call it *ALU dependence*.

If the conflicts involve workplace storage, we call it *storage dependence*. In the case of storage dependence, each task must work on independent storage locations or use protected access (such as locks or monitors to be described in Chapter 11) to shared writable data.

The transformation of a sequentially coded program into a parallel executable form can be done manually by the programmer using explicit parallelism, or by a compiler detecting implicit parallelism automatically. In both approaches, the decomposition of programs is the primary objective.

Program partitioning determines whether a given program can be partitioned or split into pieces that can execute in parallel or follow a certain **prespecified** order of execution. Some programs are inherently sequential in nature and thus cannot be decomposed into parallel branches. The detection of parallelism in programs requires a check of the various dependence relations.

Bernstein's Conditions In 1966, Bernstein revealed a set of conditions based on which two processes can execute in parallel. A *process* is a software entity corresponding to the abstraction of a program fragment defined at various processing levels. We define the *input set* I_i of a process P_i as the set of all input variables needed to execute the process.

Similarly, the *output set* O_i consists of all output variables generated after execution of the process P_i . Input variables are essentially operands which can be fetched from memory or registers, and output variables are the results to be stored in working registers or memory locations.

Now, consider two processes P_1 and P_2 with their input sets I_1 and I_2 and output sets O_1 and O_2 , respectively. These two processes can execute in parallel and are denoted $P_1 \parallel P_2$ if they are independent and do not create confusing results.

Formally, these conditions are stated as follows:

$$\begin{aligned}
 I_1 \cap O_2 &= \emptyset \\
 I_2 \cap O_1 &= \emptyset \\
 O_1 \cap O_2 &= \emptyset
 \end{aligned} \tag{2.1}$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

ming style, and compiler optimization. The program flow graph displays the patterns of simultaneously executable operations. Parallelism in a program varies during the execution period. It often limits the sustained performance of the processor.

Example 2.3 Mismatch between software parallelism and hardware parallelism (Wen-Mei Hwu, 1991)

Consider the example program graph in Fig. 2.3a. There are eight instructions (four *loads* and four *arithmetic* operations) to be executed in three consecutive machine cycles. Four *load* operations are performed in the first cycle, followed by two *multiply* operations in the second cycle and two *add/subtract* operations in the third cycle. Therefore, the parallelism varies from 4 to 2 in three cycles. The average software parallelism is equal to $8/3 = 2.67$ instructions per cycle in this example program.

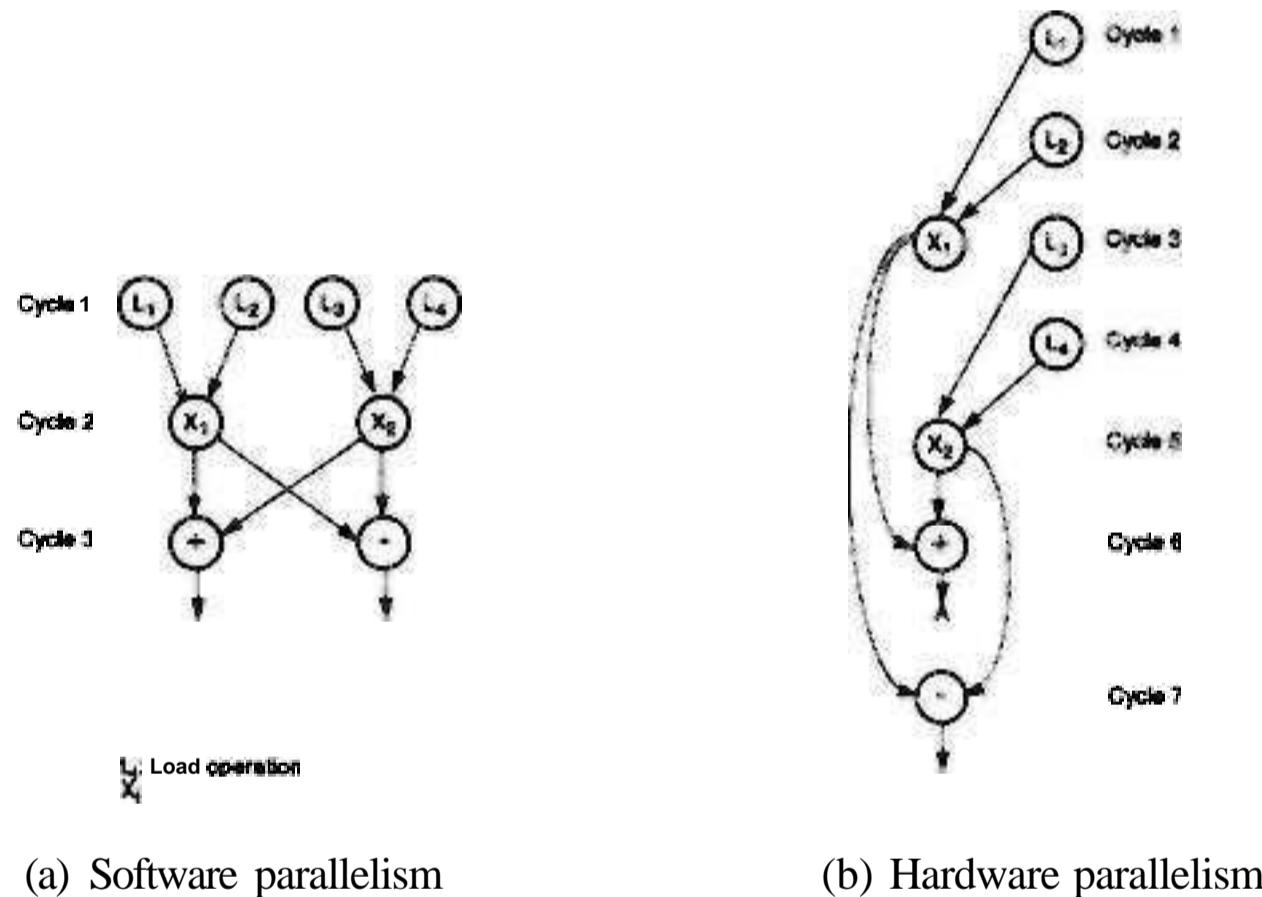


Figure 2.3 Executing an example program by a two-issue superscalar processor.

Now consider execution of the same program by a **two-issue** processor which can execute one memory access (*load* or write) and one arithmetic (*add*, *subtract*, *multiply*, etc.) operation simultaneously. With this hardware restriction, the program must execute in seven machine cycles as shown in Fig. 2.3b. **Therefore, the hardware parallelism** displays an average value of $8/7 = 1.14$ instructions executed per cycle. This demonstrates a mismatch between the software parallelism and the hardware parallelism.

Let us try to match the software parallelism shown in Fig. 2.3a in a hardware platform of a dual-processor system, where **single-issue** processors are used.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

2.2 Program Partitioning and Scheduling

This section introduces the basic definitions of computational granularity or level of parallelism in programs. Communication latency and scheduling issues are illustrated with programming examples.

2.2.1 Grain Sizes and Latency

Grain size or *granularity* is a measure of the amount of computation involved in a software process. The simplest measure is to count the number of instructions in a grain (program segment). Grain size determines the basic program segment chosen for parallel processing. Grain sizes are commonly described as *fine*, *medium*, or *coarse*, depending on the processing levels involved.

Latency is a time measure of the communication overhead incurred between machine subsystems. For example, the *memory latency* is the time required by a processor to access the memory. The time required for two processes to synchronize with each other is called the *synchronization latency*. Computational granularity and communication latency are closely related. We reveal their relationship below.

Parallelism has been exploited at various processing levels. As illustrated in Fig. 2.5, five levels of program execution represent different computational grain sizes and changing communication and control requirements. The lower the level, the finer the granularity of the software processes.

In general, the execution of a program may involve a combination of these levels. The actual combination depends on the application, formulation, algorithm, language, program, compilation support, and hardware limitations. We characterize below the parallelism levels and review their implementation issues from the viewpoints of a programmer and of a compiler writer.

Instruction Level At instruction or statement level, a typical grain contains less than 20 instructions, called *fine grain* in Fig. 2.5. Depending on individual programs, fine-grain parallelism at this level may range from two to thousands. Butler et al. (1991) has shown that single-instruction-stream parallelism is greater than two. Wall (1991) finds that the average parallelism at instruction level is around five, rarely exceeding seven, in an ordinary program. For scientific applications, Kumar (1988) has measured the average parallelism in the range of 500 to 3000 Fortran statements executing concurrently in an idealized environment.

The advantage of fine-grain computation lies in the abundance of parallelism. The exploitation of fine-grain parallelism can be assisted by an optimizing compiler which should be able to automatically detect parallelism and translate the source code to a parallel form which can be recognized by the run-time system. Instruction-level parallelism is rather tedious for an ordinary programmer to detect in a source code.

Loop Level This corresponds to the iterative loop operations. A typical loop contains less than 500 instructions. Some loop operations, if independent in successive iterations, can be vectorized for pipelined execution or for lock-step execution on SIMD machines.

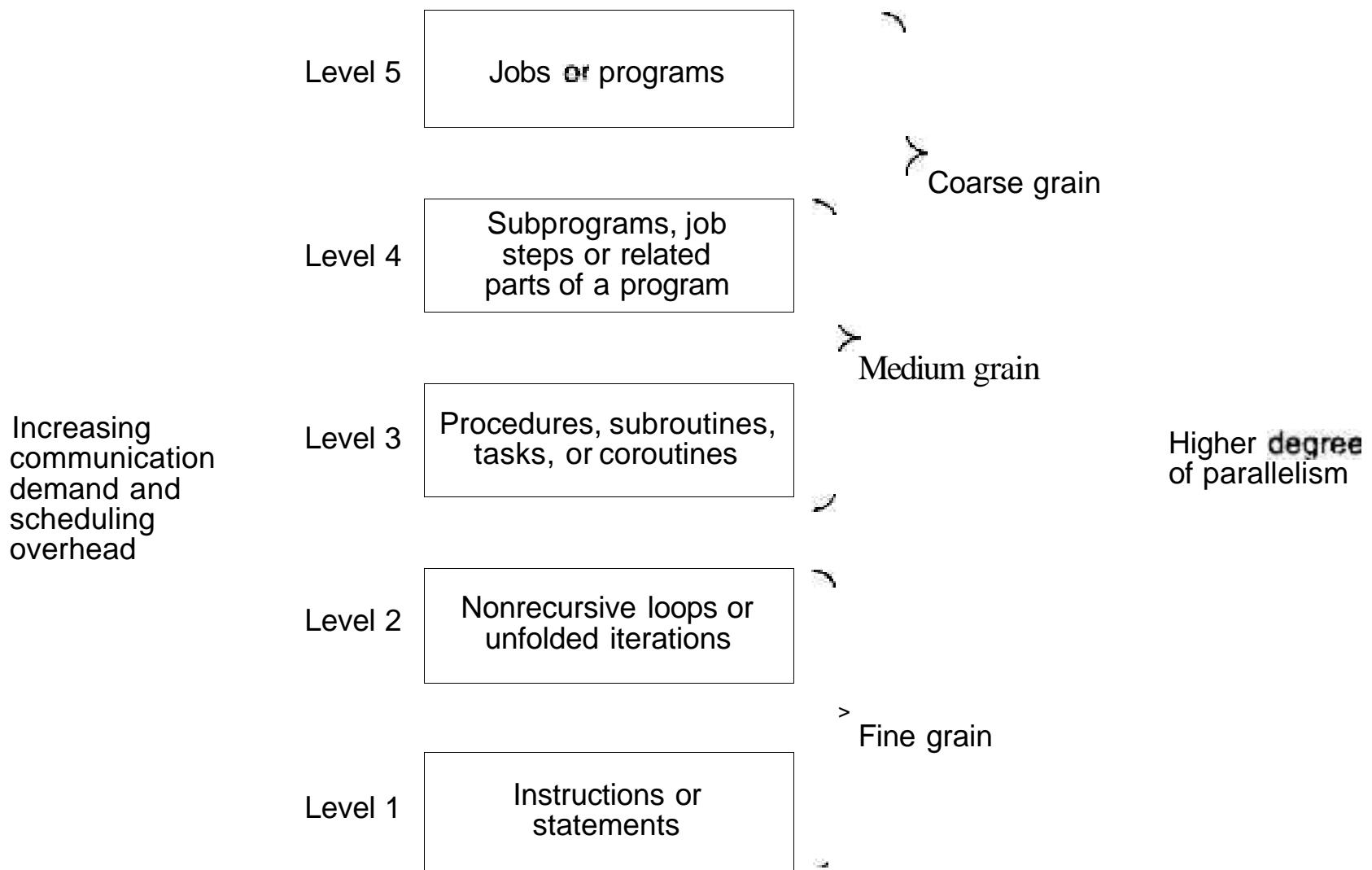


Figure 2.5 Levels of parallelism in program execution on modern computers.
(Reprinted from Hwang, *Proc. IEEE*, October 1987)

Some loop operations can be self-scheduled for parallel execution on MIMD machines.

Loop-level parallelism is the most optimized program construct to execute on a parallel or vector computer. However, recursive loops are rather difficult to parallelize. Vector processing is mostly exploited at the loop level (level 2 in Fig. 2.5) by a vectorizing compiler. The loop level is still considered a fine grain of computation.

Procedure Level This level corresponds to medium-grain size at the task, procedural, subroutine, and coroutine levels. A typical grain at this level contains less than 2000 instructions. Detection of parallelism at this level is much more difficult than at the finer-grain levels. Interprocedural dependence analysis is much more involved and history-sensitive.

The communication requirement is often less compared with that required in MIMD execution mode. SPMD execution mode is a special case at this level. Multitasking also belongs in this category. Significant efforts by programmers may be needed to restructure a program at this level, and some compiler assistance is also needed.

Subprogram Level This corresponds to the level of job steps and related subprograms. The grain size may typically contain thousands of instructions. Job steps can overlap across different jobs. Subprograms can be scheduled for different processors in

SPMD or MPMD mode, often on message-passing multicomputers.

Multiprogramming on a uniprocessor or on a multiprocessor is conducted at this level. In the past, parallelism at this level has been exploited by algorithm designers or programmers, rather than by compilers. We do not have good compilers for exploiting medium- or coarse-grain parallelism at present.

Job (Program) Level This corresponds to the parallel execution of essentially independent jobs (programs) on a parallel computer. The grain size can be as high as tens of thousands of instructions in a single program. For supercomputers with a small number of very powerful processors, such coarse-grain parallelism is practical. Job-level parallelism is handled by the program loader and by the operating system in general. Time-sharing or space-sharing multiprocessors explore this level of parallelism. In fact, both time and space sharing are extensions of multiprogramming.

To summarize, fine-grain parallelism is often exploited at instruction or loop levels, preferably assisted by a parallelizing or vectorizing compiler. Medium-grain parallelism at the task or job step demands significant roles for the programmer as well as compilers. Coarse-grain parallelism at the program level relies heavily on an effective OS and on the efficiency of the algorithm used. Shared-variable communication is often used to support fine-grain and medium-grain computations.

Message-passing multicomputers have been used for medium- and coarse-grain computations. In general, the finer the grain size, the higher the potential for parallelism and the higher the communication and scheduling overhead. Fine grain provides a higher degree of parallelism, but heavier communication overhead, as compared with coarse-grain computations. Massive parallelism is often explored at the fine-grain level, such as data parallelism on SIMD or MIMD computers.

Communication Latency By balancing granularity and latency, one can achieve better performance of a computer system. Various latencies are attributed to machine architecture, implementing technology, and communication patterns involved. The architecture and technology affect the design choices for latency tolerance between subsystems. In fact, latency imposes a limiting factor on the scalability of the machine size. For example, memory latency increases with respect to memory capacity. Thus memory cannot be increased indefinitely without exceeding the tolerance level of the access latency. Various latency hiding or tolerating techniques will be studied in Chapter 9.

The latency incurred with interprocessor communication is another important parameter for a system designer to minimize. Besides signal delays in the data path, IPC latency is also affected by the communication patterns involved. In general, n tasks communicating with each other may require $n(n - 1)/2$ communication links among them. Thus the complexity grows quadratically. This leads to a communication bound which limits the number of processors allowed in a large computer system.

Communication patterns are determined by the algorithms used as well as by the architectural support provided. Frequently encountered patterns include permutations and broadcast, multicast, and conference (many-to-many) communications. The communication demand may limit the granularity or parallelism. Very often tradeoffs do exist between the two.

The **communication** issue thus involves the reduction of latency or complexity, the prevention of **deadlock**, minimizing blocking in communication patterns, and the trade-off between parallelism and communication overhead. We **will** study techniques that minimize communication latency, prevent deadlock, and optimize grain size throughout the hook.

2.2.2 Grain Packing and Scheduling

Two fundamental **questions** to ask in parallel programming are: (i) How can we partition a program into parallel **branches**, program modules, microtasks, or grains to yield the shortest possible execution time? and (ii) What is the optimal size of concurrent grains in a computation?

This grain-size problem demands determination of both the number and the size of grains (or microtasks) in a parallel program. Of course, the solution is both **problem-dependent** and **machine-dependent**. The goal is to produce a short schedule for fast execution of subdivided program modules.

There exists a tradeoff between parallelism and scheduling/synchronization overhead. The time complexity involves both computation and communication overheads

The program partitioning involves the **algorithm** designer, programmer, compiler, operating system support, etc. We describe below a **grain packing** approach introduced by Kruatrachue and Lewis (1988) for parallel programming applications.

Example 2.4 Program graph before and after grain packing (Kruatrachue and Lewis, 1988)

The basic concept of program partitioning is introduced below. In Fig. 2.6, we show an example *program graph* in two different grain sizes. A program graph shows the structure of a program. It is very similar to the dependence graph introduced in Section 2.1.1. Each node in the program graph corresponds to a computational unit in the program. The *grain size* is measured by the number of basic machine cycles (including both processor and memory cycles) needed to execute all the operations within the node.

We denote each node in Fig. 2G by a pair (n, s) , where n is the *node name* (id) and s is the grain size of the node. Thus grain size **reflects** the number of computations involved in a program segment. **Fine-grain** nodes have a smaller grain size, and coarse-grain nodes have a larger grain size.

The edge label (v, d) between two end nodes specifies the output variable v from the source node or the input variable to the destination node, and the **communication delay** d between them. This delay includes all the path delays and memory latency involved.

There are 17 nodes in the fine-grain program graph (Fig. 2.6a) and 5 in the coarse-grain program graph (Fig. 2.6b). The **coarse-grain** node is obtained by combining (grouping) multiple fine-grain nodes. The fine grain corresponds to the following program:

Var $a, b, c, d, e, f, g, h, i, j, \text{fc}, l, m, n, o, p, q$

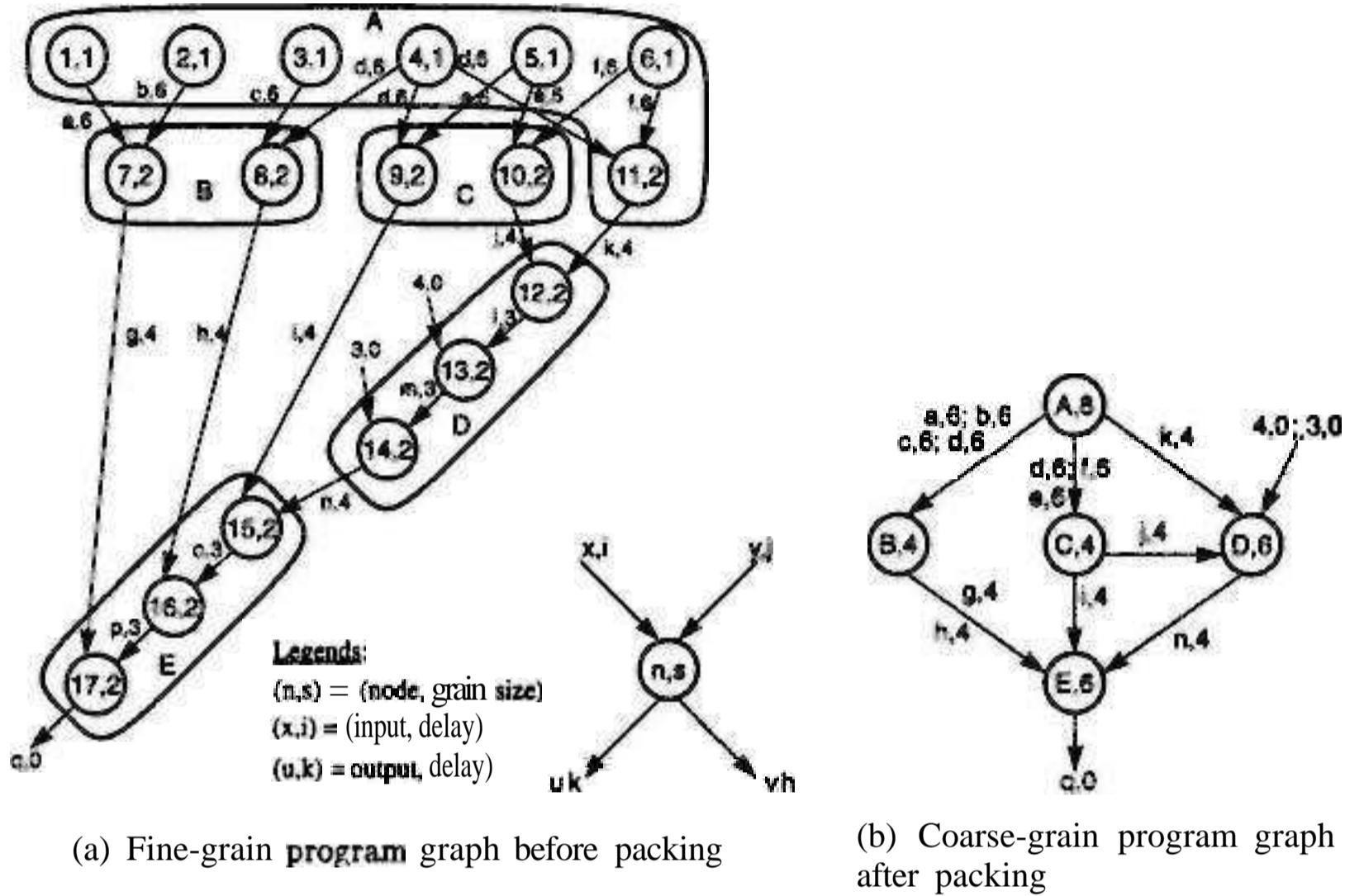


Figure 2.6 A program graph before and after **grain** packing in Example 2.4. (Modified from Kruatrachue and Lewis, *IEEE Software*, Jan. 1988)

Begin

1.	a := 1	10.	j := e × f
2.	fr := 2	11.	k := d × f
3.	c := 3	12.	l := j × k
4.	d := 4	13.	m := <i>Ax l</i>
5.	e := 5	14.	n := 3 × m
6.	/ := 6	15.	o := n × i
7.	g := a × 6	16.	p := o × h
8.	h := c × d	17.	q := p × g
9.	i := d × e		

End

Nodes 1, **2**, **3**, 4, 5, and 6 are memory reference (data fetch) operations. Each takes one cycle to address **and** six cycles to fetch from memory. All remaining nodes (7 to 17) are CPU operations, each **requiring** two cycles to complete. After packing, the coarse-grain nodes have larger **grain** sizes **ranging** from 4 to 8 as shown.

The **node** (A,8) in Fig. 2.6b is obtained by combining the nodes (1,1)* (2,1), (3,1), (**4,1**), (**5,1**), (6,1), and (11,2) in Fig. 2.6a. The grain size, **8**, of node A is the **summation** of all grain sizes ($1 + 1 + 1 + 1 + 1 + 2 = 8$) **being** combined.

The idea of grain packing is to apply fine grain first in order to achieve a higher degree of parallelism. Then one combines (packs) multiple fine-grain nodes into a coarse-grain node if it can eliminate unnecessary communications delays or reduce the overall scheduling overhead.

Usually, all fine-grain operations within a single coarse-grain node are assigned to the same processor for execution. **Fine-grain** partition of a program often demands more interprocessor communication than that required in a **coarse-grain partition**. Thus grain packing offers a tradeoff between parallelism and scheduling/communication overhead.

Internal delays among fine-grain operations within the same coarse-grain node are negligible because the communication delay is **contributed** mainly by interprocessor delays rather than by delays within the same processor. The choice of the optimal grain size is meant to achieve the shortest schedule for the nodes on a parallel computer system.

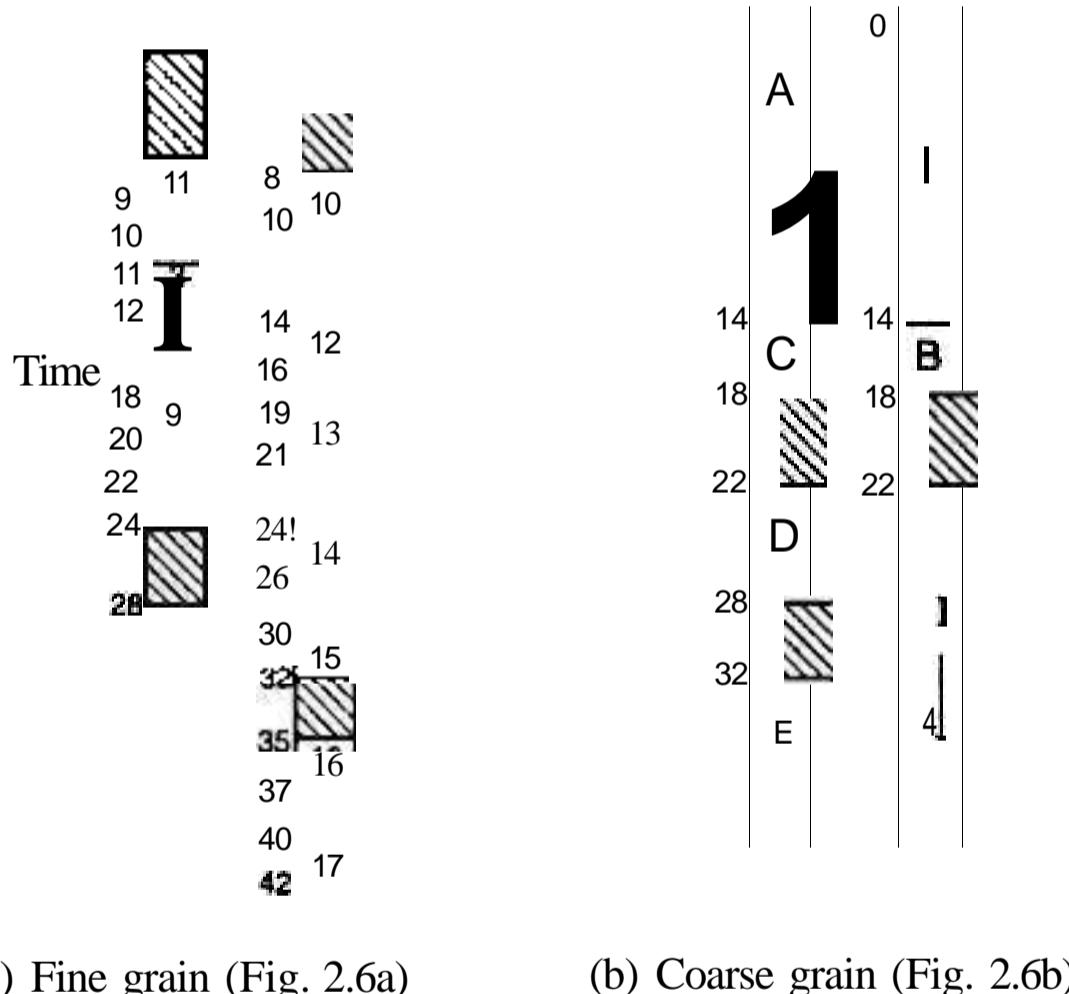


Figure 2.7 Scheduling of the fine-grain and coarse-grain **programs**. (I: idle time; shaded areas: communication delays)

With respect to the fine-grain versus coarse-grain program graphs in Fig. 2.6, two multiprocessor schedules are shown in Fig. 2.7. The fine-grain schedule is longer (42 time units) because more communication delays were included as shown by the shaded area. The coarse-grain schedule is shorter (38 time units) because communication delays among nodes 12, 13, and 14 within the same node D (and also the delays among 15, 16, and 17 within the node E) are eliminated after grain packing.

2.2.3 Static Multiprocessor Scheduling

Grain packing may not always produce a shorter schedule. In general, dynamic multiprocessor scheduling is an NP-hard problem. Very often heuristics are used to yield suboptimal solutions. We introduce below the basic concepts behind multiprocessor scheduling using static schemes.

Node Duplication In order to eliminate the idle time and to further reduce the communication delays among processors, one can duplicate some of the nodes in more than one processor.

Figure 2.8a shows a schedule without duplicating any of the five nodes. This schedule contains idle time as well as long interprocessor delays (8 units) between P₁ and P₂. In Fig. 2.8b, node A is duplicated into A' and assigned to P₂ besides retaining the original copy A in P₁. Similarly, a duplicated node C' is copied into P₁ besides the original node C in P₂. The new schedule shown in Fig. 2.8b is almost 50% shorter than that in Fig. 2.8a. The reduction in schedule time is caused by elimination of the (a, 8) and (c, 8) delays between the two processors.

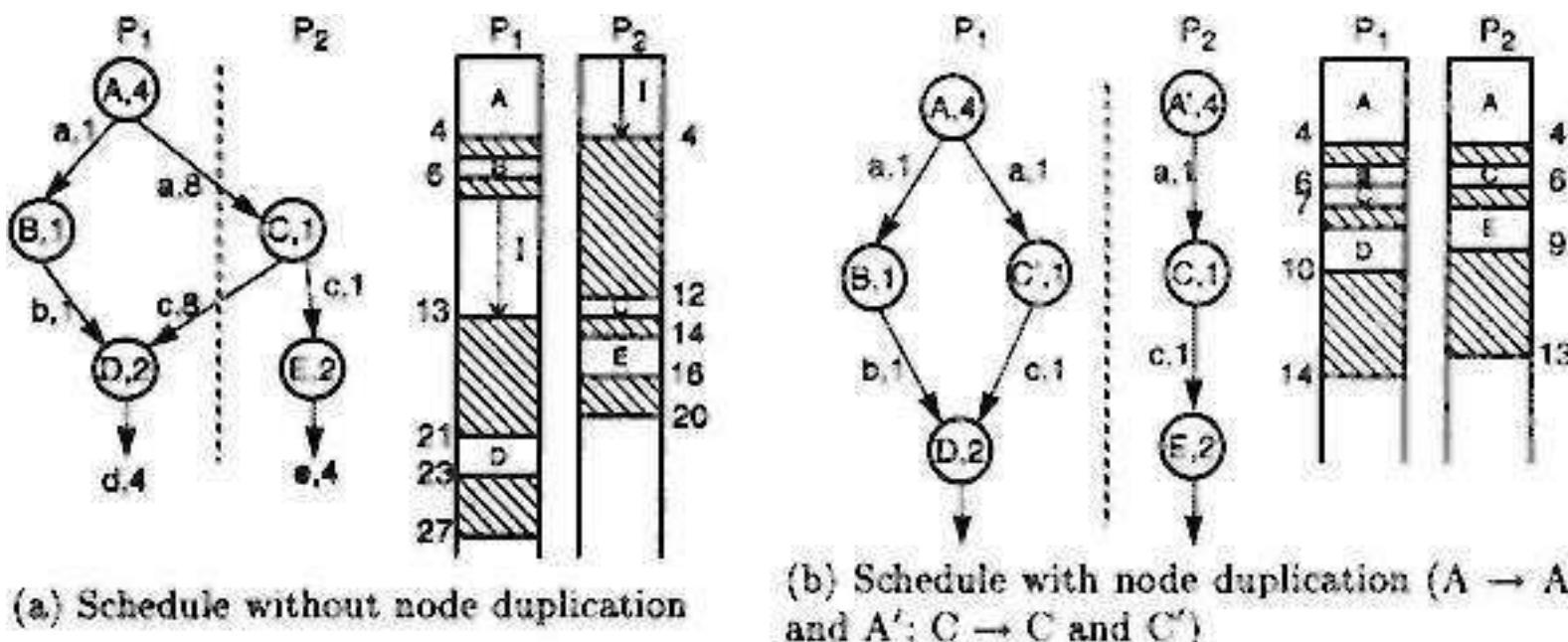


Figure 2.8 Node-duplication scheduling to eliminate communication delays between processors. (I: idle time; shaded areas: communication delays)

Grain packing and node duplication are often used jointly to determine the best grain size and corresponding schedule. Four major steps are involved in the grain determination and the process of scheduling optimization:

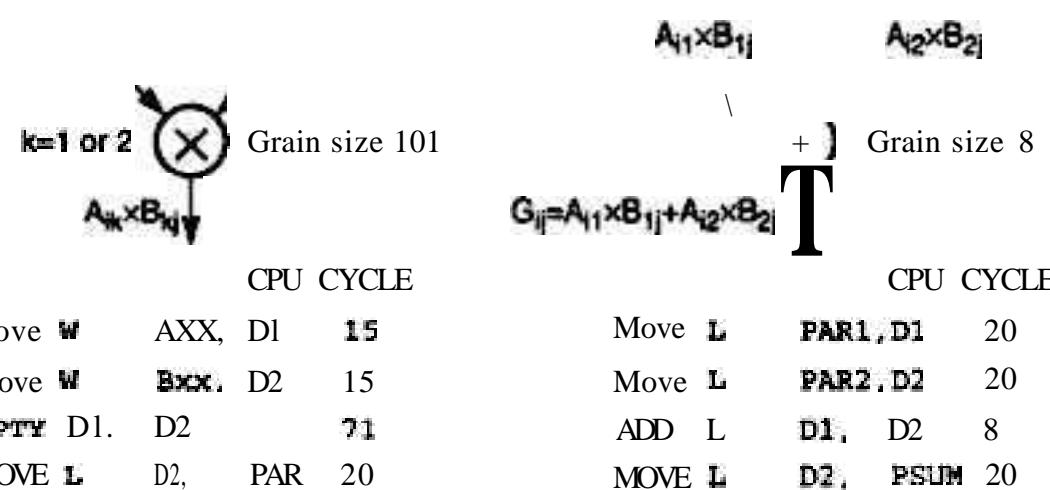
- Step 1. Construct a fine-grain program graph.
- Step 2. Schedule the fine-grain computation.
- Step 3. Grain packing to produce the coarse grains-
- Step 4. Generate a parallel schedule based on the packed graph.

The purpose of multiprocessor scheduling is to obtain a minimal time schedule for the computations involved. The following example clarifies this concept.

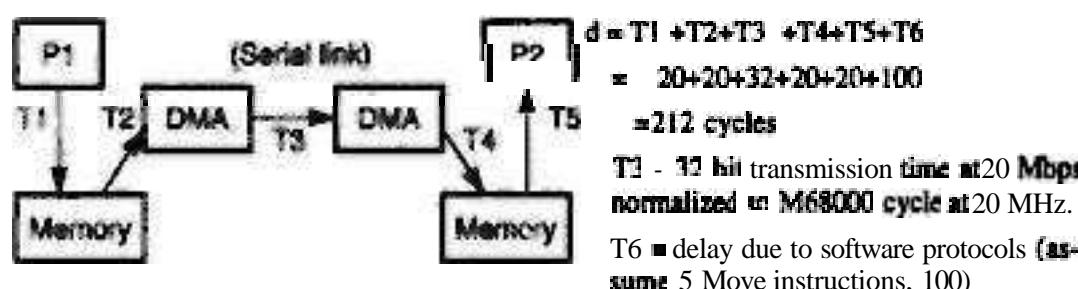
Example 2.5 Program decomposition for static multiprocessor scheduling (Kruatrachue and Lewis, 1988)

Figure 2.9 shows an example of how to calculate the grain size and communication latency. In this example, two 2×2 matrices A and B are multiplied to compute the sum of the four elements in the resulting product matrix $C = A \times B$. There are eight multiplications and seven additions to be performed in this program, as written below:

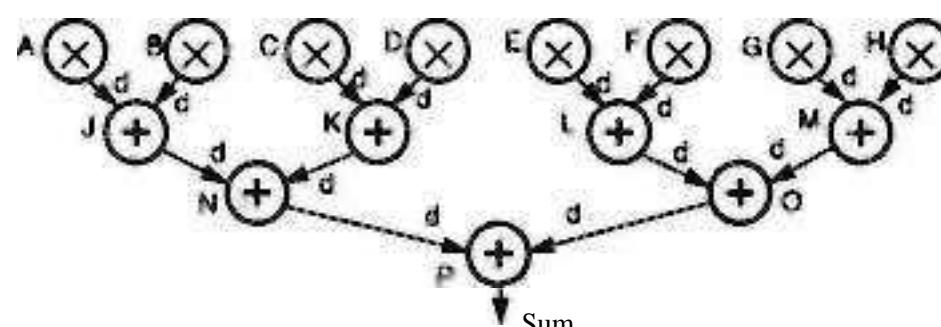
$$\begin{array}{cc|c|cc|cc} A_{11} & A_{12} & \times & B_{11} & B_{12} & | & C_{11} & C_{12} \\ A_{21} & A_{22} & & B_{21} & B_{22} & & C_{21} & C_{22} \end{array}$$



(a) Grain size calculation in **M68000** assembly code at **20-MHz** cycle



(b) Calculation of communication delay d



(c) Fine-grain program graph

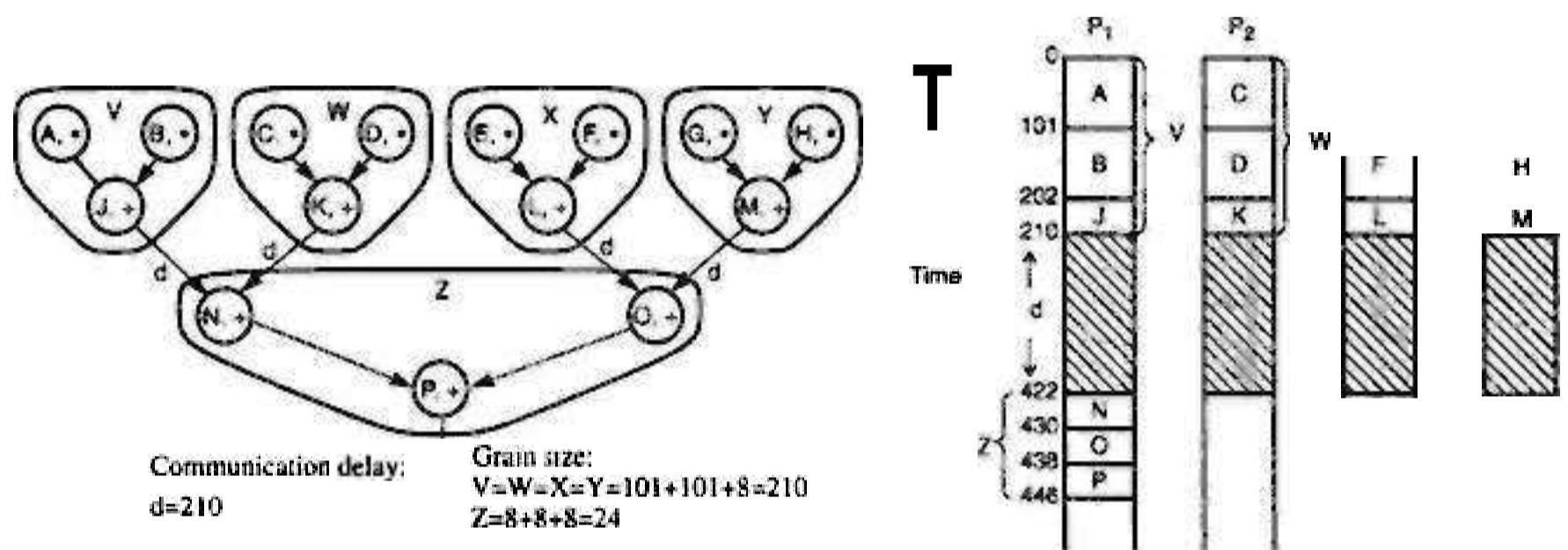
Figure 2.9 Calculation of grain size and communication delay for the program graph in Example 2.5. (Courtesy of Kruatrachue and Lewis; reprinted with permission from *IEEE Software*, 1988)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

nodes on 8 processors with incurred communication delays (shaded areas). Note that the communication delays have slowed down the parallel execution significantly, resulting in many processors idling (indicated by I), except for **P1** which produces the **final** sum. A speedup factor of $864/741 = 1.16$ is observed.

Next we show how to use grain packing (Step 3) to reduce the communication overhead. As shown in Fig. 2.11, we group the nodes in the top two levels into four coarse-grain nodes labeled **V**, **W**, **X**, and **Y**. The remaining three nodes (**N**, **O**, **P**) then form the fifth node **Z**. Note that there is only one level of interprocessor communication required as marked by **d** in Fig. 2.11a.



(a) Grain packing of 15 small nodes into 5 bigger nodes (b) Parallel schedule for the packed program

Figure 2.11 Parallel scheduling for Example 2.5 after **grain** packing to reduce communication delays.

Since the maximum degree of parallelism is now reduced to 4 in the program graph, we use only four processors to execute this coarse-grain **program**. A parallel schedule is worked out (Fig. 2.11) for this program in 446 cycles, resulting in an improved speedup of $864/446 = 1.94$.

2.3 Program Flow Mechanisms

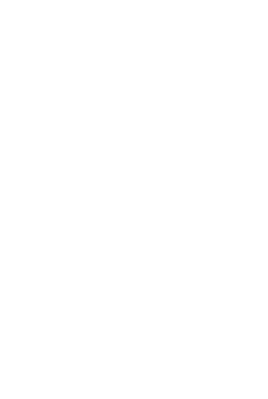
Conventional computers are based on a control flow mechanism by which the order of program execution is explicitly stated in the user **programs**. Dataflow computers are based on a data-driven mechanism which allows the execution of any instruction to be driven by data (operand) availability. Dataflow computers emphasize a high degree of parallelism at the fine-grain instructional level. Reduction computers are based on a demand-driven mechanism which initiates an operation based on the demand for its results by other computations.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

produced then trigger the execution of a_5, b_1, a_6 , and a_7 starting from cycle 4. The data-driven chain reactions are shown in Fig. 2.13c. The output c_8 is the last one to produce, due to its dependence on all the previous c_i 's.

Figure 2.13d shows the execution of the same set of computations on a conventional multiprocessor using shared memory to hold the intermediate results (s_i and t_i for $i = 1, 2, 3, 4$). Note that no shared memory is used in the dataflow implementation. The example does not show any time advantage of dataflow execution over control flow execution.

The theoretical minimum time is 13 cycles along the critical path $a_1 b_1 c_1 c_2 \dots c_8$. The chain reaction control in dataflow is more difficult to implement and may result in longer overhead, as compared with the uniform operations performed by all the processors in Fig. 2.13d.

•

One advantage of tagging each datum is that data from different contexts can be mixed freely in the instruction execution pipeline. Thus, **instruction-level** parallelism of dataflow graphs can absorb the communication latency and minimize the losses due to synchronization waits. Besides token matching and **I-structure**, compiler technology is also needed to generate dataflow graphs for tagged-token dataflow computers. The dataflow architecture offers an ideal model for massively parallel computations because all far-reaching side effects are removed. Side effects refer to the modification of some shared variables by unrelated operations.

2.3.2 Demand-Driven Mechanisms

In a *reduction machine*, the **computation** is triggered by the demand for an **operation's result**. Consider the evaluation of a nested arithmetic expression $a = ((6 + 1) \times c - (d \div e))$. The data-driven computation chooses a bottom-up approach, starting from the innermost operations $6 + 1$ and $d \div e$, then proceeding to the \times operation, and finally to the outermost operation $-$. Such a computation has been called *eager evaluation* because operations are carried out immediately after all their operands become available.

A *demand-driven* computation chooses a **top-down** approach by first demanding **the value** of a , which triggers the demand for evaluating the next-level expressions $(b+1) \times c$ and $r_f \cdot i \cdot e$, which in turn triggers the demand for evaluating $6 + 1$ at the innermost level. The results are then returned to the nested demander in the reverse order before a is evaluated.

A demand-driven computation corresponds to *lazy evaluation*, because operations are executed only when their results are required by another instruction. The demand-driven approach matches naturally with the functional programming concept. The removal of side effects in functional programming makes programs easier to parallelize. There are two types of reduction machine models, both having a recursive control mechanism as characterized below.

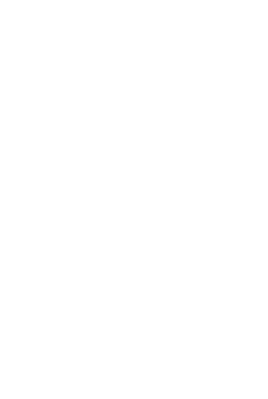
Reduction Machine Models In a *string reduction* model, each demander gets a separate copy of the expression for its own evaluation. A long string expression is



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

permutation (one-to-one), *broadcast* (one-to-all), *multicast* (many-to-many), *personalized communication (one-to-many)*, *shuffle*, *exchange*, etc. These routing functions can be implemented on ring, mesh, hypercube, or multistage networks.

Permutations For n objects, there are $n!$ permutations by which the n objects can be reordered. The set of all permutations form a *permutation group* with respect to a composition operation. One can use cycle notation to specify a permutation function.

For example, the permutation $\pi = (a, b, c)(d, e)$ stands for the bijection mapping: $a \rightarrow b, b \rightarrow c, c \rightarrow a, d \rightarrow e$ and $e \rightarrow d$ in a circular fashion. The cycle (a, b, c) has a period of 3, and the cycle (d, e) a period of 2. Combining the two cycles, the permutation π has a **period** of $2 \times 3 = 6$. If one applies the permutation π six times, the identity mapping $I = (a), (b), (c), (d), (e)$ is obtained.

One can use a crossbar switch to implement the permutation. Multistage networks can implement some of the permutations in one or multiple passes through the network. Permutations can also be implemented with shifting or broadcast operations. The permutation capability of a network is often used to indicate the data routing capability. When n is large, the permutation speed often dominates the performance of a data-routing network.

Perfect Shuffle and Exchange Perfect shuffle is a special permutation function suggested by Harold Stone (1971) for parallel processing applications. The mapping corresponding to a perfect shuffle is shown in Fig. 2.14a. Its inverse is shown on the right-hand side (Fig. 2.14b).

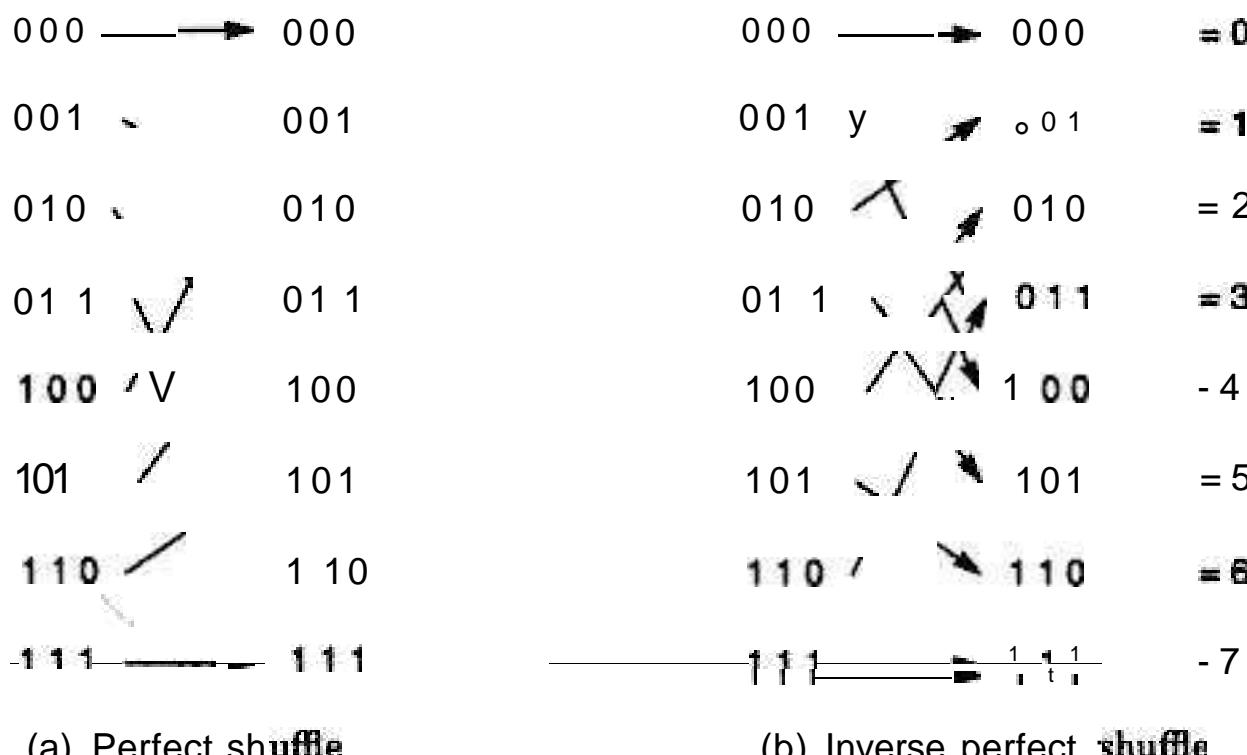


Figure 2.14 Perfect shuffle and its inverse mapping over eight objects- (Courtesy of H. Stone; reprinted with permission from *IEEE Trans. Computers*, 1971)

In **general**, to shuffle $n = 2^k$ objects **evenly**, one can express each object in the



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

bidirectional. It is **symmetric** with a constant node degree of 2. The diameter is $\lceil N/2 \rceil$ for a bidirectional ring, and N for unidirectional ring.

The **IBM token ring** has this topology, in which messages circulate along the ring until they reach the destination with a matching token. Pipelined or packet-switched, rings have been implemented in the CDC Cyberplus multiprocessor (1985) and in the **KSR-1** computer system (1992) for interprocessor communications.

By increasing the node degree from 2 to 3 or 4, we obtain two **chordal rings** as shown in Figs. 2.16c and 2.16d, respectively. One and two extra links are added to produce the two chordal rings, respectively. In general, the more links added, the higher the node degree and the shorter the network diameter.

Comparing the **16-node ring** (Fig. 2.16b) with the two chordal rings (Figs. 2.16c and 2.16d), the network diameter drops from 8 to 5 and to 3, respectively. In the extreme, the *completely connected network* in Fig. 2.16f has a node degree of 15 with the shortest possible diameter of 1.

Barrel Shifter As shown in Fig. 2.16e for a network of $N = 16$ nodes, the *barrel shifter* is obtained from the ring by adding extra links from each node to those nodes having a distance equal to an integer power of 2. This implies that node i is connected to node j if $|j - i| = 2^r$ for some $r = 0, 1, 2, \dots, n - 1$ and the network size is $N = 2^n$. Such a barrel shifter has a node degree of $d = 2n - 1$ and a diameter $D = n/2$.

Obviously, the connectivity in the barrel shifter is increased over that of any chordal ring of lower node degree. For $N = 16$, the barrel shifter has a node degree of 7 with a diameter of 2. But the barrel shifter complexity is still much lower than that of the completely connected network (Fig. 2.16f).

Tree and Star A *binary tree* of 31 nodes in five levels is shown in Fig. 2.17a. In general, a k -level, completely balanced binary tree should have $N = 2^k - 1$ nodes. The maximum node degree is 3 and the diameter is $2(k - 1)$. With a constant node degree, the binary tree is a scalable architecture. However, the diameter is rather long.

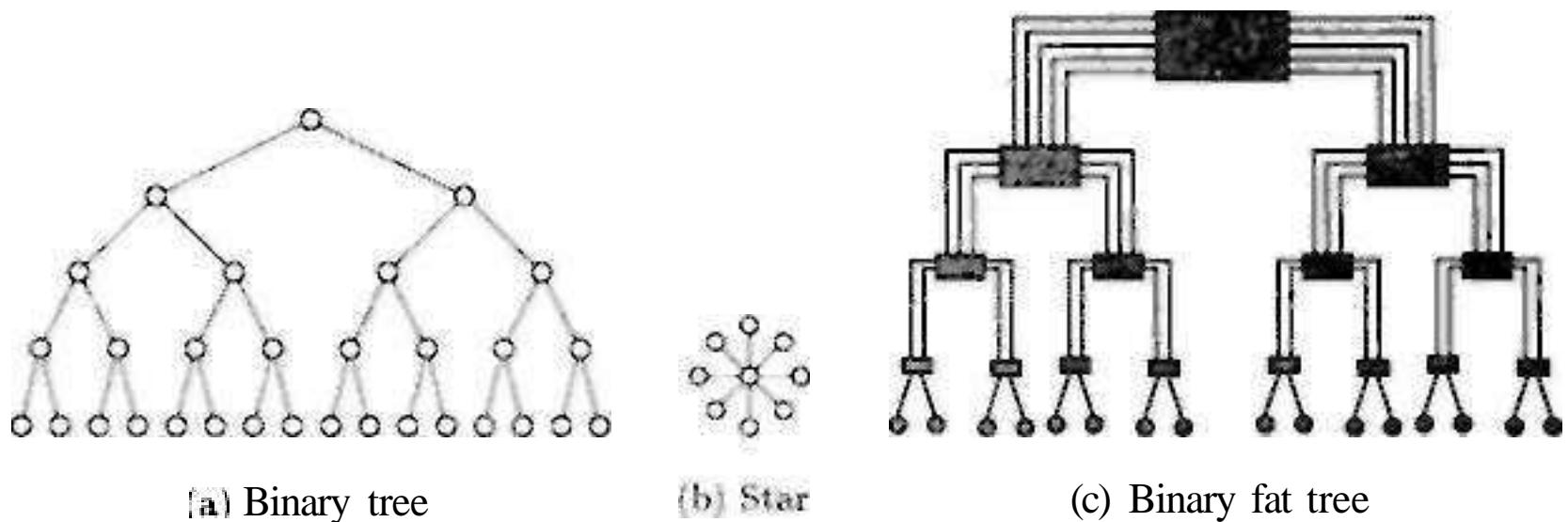


Figure 2.17 Tree, star, and fat tree.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$2k$ and the network diameter is $k(n - 1)$. Note that the pure mesh as shown in Fig. 2.17c is not symmetric. The node degrees at the boundary and corner nodes are 3 or 2.

Figure 2.18b shows a variation of the mesh by allowing wraparound connections. The Illiac IV assumed an 8×8 Illiac mesh with a constant node degree of 4 and a diameter of 7. The Illiac mesh is topologically equivalent to a chordal ring of degree 4 as shown in Fig. 2.16d for an $N = 9 = 3 \times 3$ configuration.

In general, an $n \times n$ Illiac mesh should have a diameter of $d = n - 1$, which is only half of the diameter for a pure mesh. The *torus* shown in Fig. 2.18c can be viewed as another variant of the mesh with an even shorter diameter. This topology combines the ring and mesh and extends to higher dimensions.

The torus has ring connections along each row and along each column of the array. In general, a $n \times n \times n$ binary torus has a node degree of 4 and a diameter of $2[n/2]$. The torus is a symmetric topology. All added wraparound connections help reduce the diameter by **one-half** from that of the mesh.

Systolic Arrays This is a class of multidimensional pipelined array architectures designed for implementing fixed algorithms. What is shown in Fig. 2.18d is a systolic array specially designed for performing matrix-matrix multiplication. The interior node degree is 6 in this example.

In general, static systolic arrays are pipelined with multidirectional flow of data streams. The commercial machine Intel iWarp system (Anaratone et al., 1986) was designed with a systolic architecture. The systolic array has become a popular research area ever since its introduction by Kung and Leiserson in 1978.

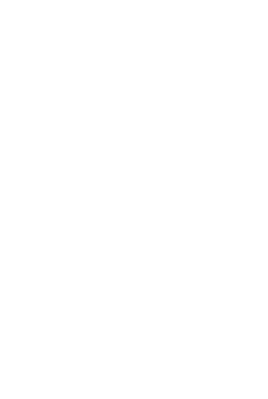
With fixed interconnection and synchronous operation, a systolic array matches the communication structure of the algorithm. For special applications like signal/image processing, systolic arrays may offer a better performance/cost ratio. However, the structure has limited applicability and can be very difficult to program. Since this book emphasizes **general-purpose** computing, we will not study systolic arrays further. Interested readers may refer to the book by S.Y. Kung (1988) for using systolic and waveform architectures in building VLSI array processors.

Hypercubes This is a binary n -cube architecture which has been implemented in the iPSC, nCUBE, and CM-2 systems. In general, an n -cube consists of $N = 2^n$ nodes spanning along n dimensions, with two nodes per dimension. A **3-cube** with 8 nodes is shown in Fig. 2.19a.

A 4-cube can be formed by interconnecting the corresponding nodes of two 3-cubes, as illustrated in Fig. 2.19b. The node degree of an n -cube equals n and so does the network diameter. In fact, the node degree increases linearly with respect to the dimension, making it difficult to consider the hypercube a scalable architecture.

Binary hypercube has been a very popular architecture for research and development in the 1980s. Both Intel iPSC/1, iPSC/2, and nCUBE machines were built with the hypercube architecture. The architecture has dense connections. Many other architectures, such as binary trees, meshes, etc., can be embedded in the hypercube.

With poor scalability and difficulty in packaging **higher-dimensional** hypercubes, the hypercube architecture is gradually being replaced by other architectures. For



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

For example, a **64-node** CCC can be formed by replacing the corner nodes of a **4-cube** with cycles of four nodes, corresponding to the case $n = 6$ and $fc = 4$. The CCC has a diameter of $2fc = 8$, longer than 6 in a 6-cube. But the CCC has a node degree of 3, smaller than the node degree of 6 in a 6-cube. In this sense, the CCC is a better architecture for building scalable systems if latency can be tolerated in some way.

fc-ary n-Cube Networks Rings, meshes, tori, binary n-cubes (hypercubes), and Omega networks are topologically isomorphic to a family of **k-ary n-cube** networks. Figure 2.20 shows a **4-ary** 3-cube network.

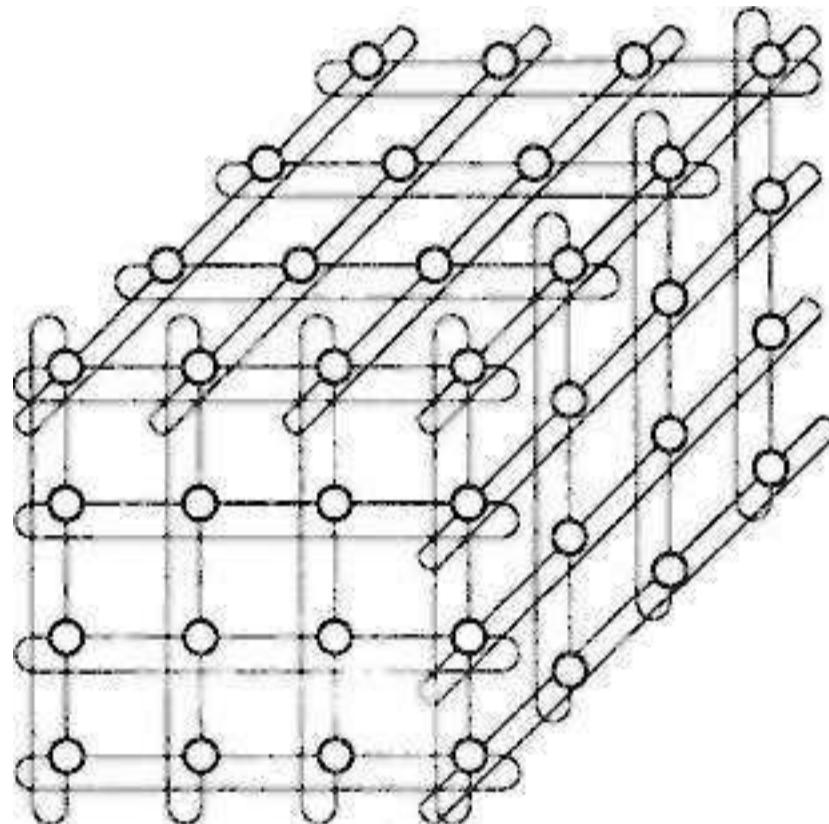


Figure 2.20 The k -ary n -cube network shown with $k = 4$ and $n = 3$; hidden nodes or connections are not shown.

The parameter n is the dimension of the cube and k is the *radix*, or the number of nodes (multiplicity) along each dimension. These two numbers are related to the number of nodes, N , in the network by:

$$N = k^n, \quad (k = \sqrt[n]{N}, \quad n = \log_k N) \quad (2.3)$$

A node in the fc-ary n-cube can be identified by an n -digit radix-fc address $A = a_0 a_1 a_2 \dots a_n$, where a_i represents the node's position in the i th dimension. For simplicity, all links are assumed bidirectional. Each line in the network represents two communication channels, one in each direction. In Fig. 2.20, the lines between nodes are bidirectional links.

Traditionally, low-dimensional fc-ary n-cubes are called **tori**, and high-dimensional binary n-cubes are called **hypercubes**. The long end-around connections in a torus can be avoided by folding the network as shown in Fig. 2.21. In this case, all links along the ring in each dimension have equal wire length when the multidimensional network



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

memory modules or secondary storage devices (disks, tape units, etc.). The system bus is often implemented on a backplane of a printed circuit board. Other boards for processors, memories, or device interfaces are plugged into the backplane board via connectors or cables.

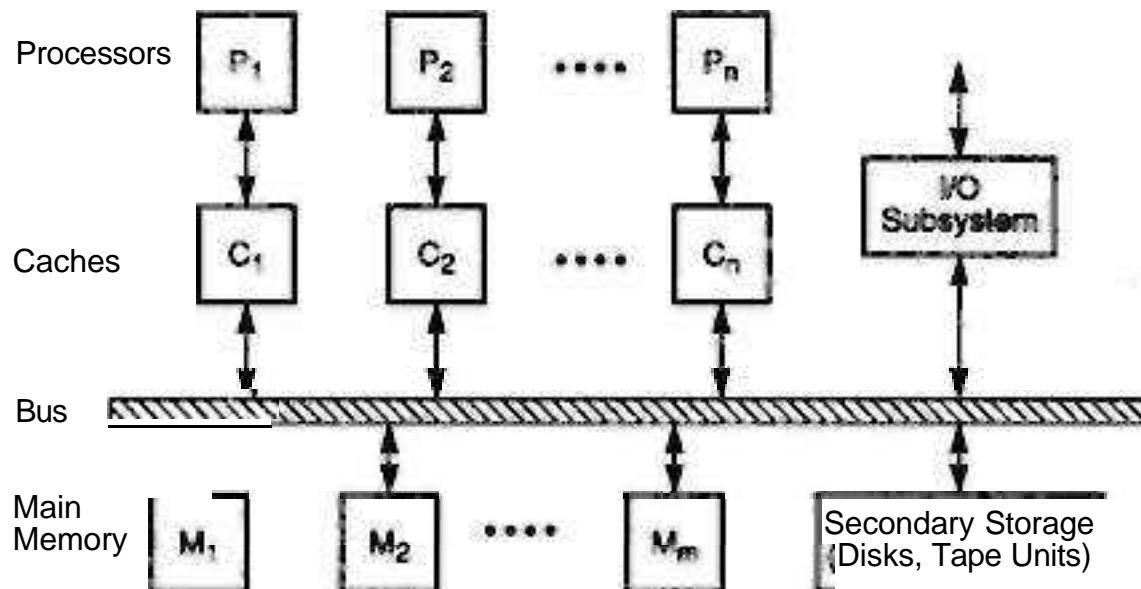


Figure 2.22 A bus-connected multiprocessor system, such as the Sequent Symmetry S1.

The active or master devices (processors or I/O subsystem) generate requests to address the memory. The passive or slave devices (memories or peripherals) respond to the requests. The common bus is used on a time-sharing basis, and important busing issues include the bus arbitration, interrupts handling, coherence protocols, and transaction processing. We will study various bus systems, such as the VME bus and the IEEE Futurebus+, in Chapter 5. Hierarchical bus structures for building larger multiprocessor systems are studied in Chapter 7.

Switch Modules A $a \times b$ switch module has a inputs and b outputs. A *binary switch* corresponds to a 2×2 switch module in which $a = b = 2$. In theory, a and b do not have to be equal. However, in practice, a and b are often chosen as integer powers of 2; that is, $a = b = 2^k$ for some $k > 1$.

Table 2.3 lists several commonly used switch module sizes: 2×2 , 4×4 , and 8×8 . Each input can be connected to one or more of the outputs. However, conflicts must be avoided at the output terminals. In other words, one-to-one and one-to-many mappings are allowed; but many-to-one mappings are not allowed due to conflicts at the output terminal.

When only one-to-one mappings (permutations) are allowed, we call the module an $n \times n$ crossbar switch. For example, a 2×2 crossbar switch can connect two possible patterns: *straight* or *crossover*. In general, an $n \times n$ crossbar can achieve $n!$ permutations. The numbers of legitimate connection patterns for switch modules of various sizes are listed in Table 2.3.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

to the **subblocks** until the $N/2$ subblocks of size 2×2 are reached.

The small boxes and the ultimate building blocks of the subblocks are the 2×2 switches, each with two legitimate connection states: *straight* and *crossover* between the two inputs and two outputs. A 16×16 Baseline network is shown in Fig. 2.25b. In Problem 2.15, readers are asked to prove the topological equivalence between the Baseline and other networks.

Crossbar Network The highest bandwidth and interconnection capability are provided by crossbar networks. A crossbar network can be visualized as a single-stage switch network. Like a telephone switchboard, the crosspoint switches provide dynamic connections between (**source**, destination) pairs. Each crosspoint switch **can** provide a dedicated connection path between a pair. The switch can be set on or off dynamically upon program demand. Two types of crossbar networks are illustrated in Fig. 2.26.

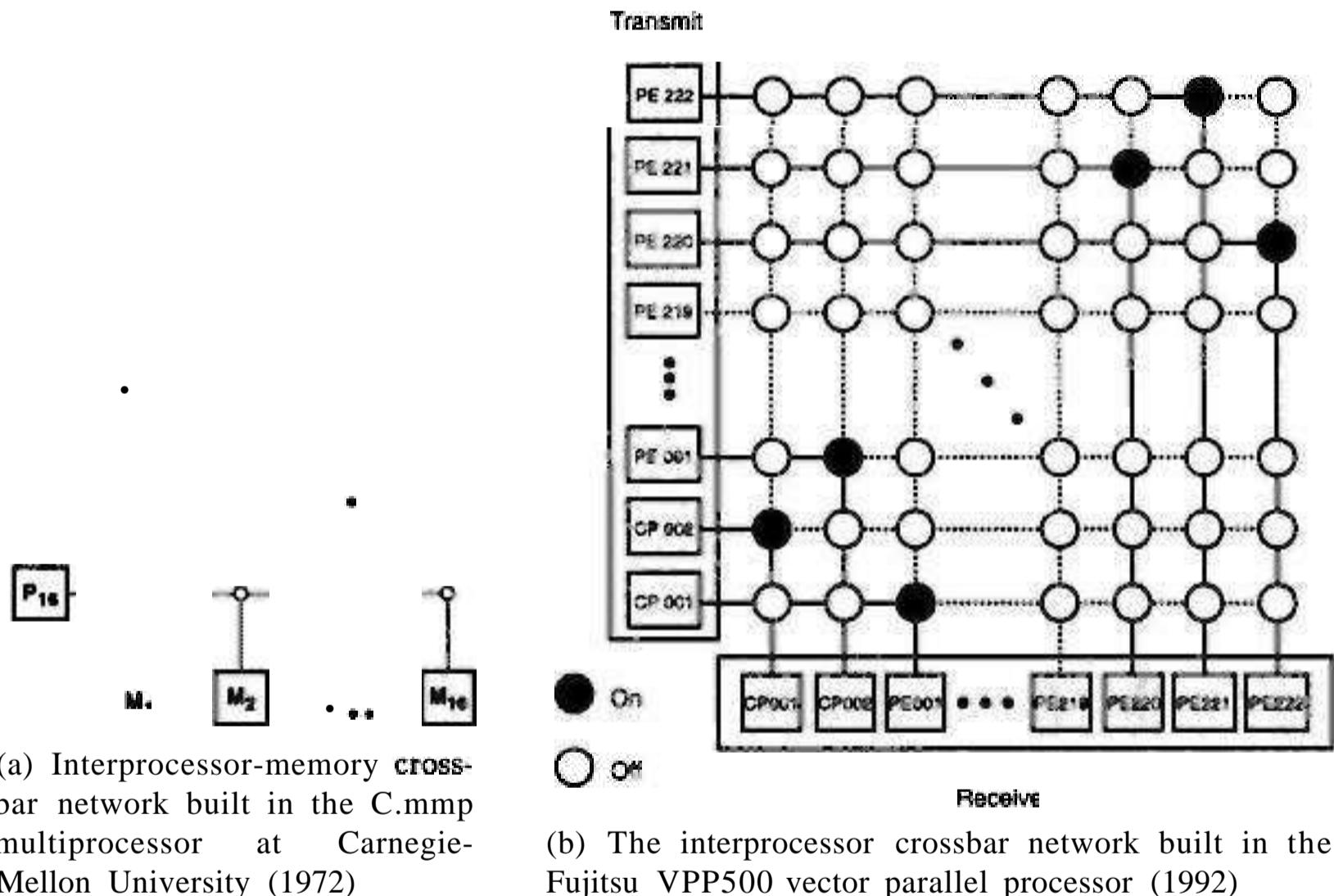


Figure 2.26 Two crossbar switch network **configurations**.

To build a **shared-memory** multiprocessor, one can use a crossbar network between the processors and memory modules (Fig. 2.26a). This is essentially a memory-access network. The **C.mmp** multiprocessor (**Wulf** and Bell. 1972) has implemented a 16×16 crossbar network which connects 16 PDP 11 processors to 16 memory modules, each of which has a capability of 1 million words of memory cells. The 16 memory modules can



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

        Do 10 I = 1, N
S1:      A(I+1) = B(I-1) + C(I)
S2:      B(I) = A(I) x K
S3:      C(I) = B(I) - 1
10      Continue

```

Problem 2.5 Analyze the data dependences among the following statements in a given program:

S1:	Load R1, 1024	/R1 \leftarrow 1024/
S2:	Load R2, M(10)	/R2 \leftarrow Memory(10) /
S3:	Add R1, R2	/R1 \leftarrow (R1) + (R2) /
S4:	Store M(1024), R1	/Memory(1024) \leftarrow (R1) /
S5:	Store M((R2)), 1024	/Memory(64) \leftarrow 1024 /

where (R_i) means the content of register R_i and Memory(lO) contains 64 initially.

- (a) Draw a dependence graph to show all the dependences.
- (b) Are there any resource dependences if only one copy of each functional unit is available in the CPU?
- (c) Repeat the above for the following program statements:

S1:	Load R1, M(100)	/R1 \leftarrow Memory(100) /
S2:	Move R2, R1	/R2 \leftarrow (R1) /
S3:	Inc R1	/R1 \leftarrow (R1) + 1 /
S4:	Add R2, R1	/R2 \leftarrow (R2) + (R1) /
S5:	Store M(100), R1	/Memory(100) \leftarrow (R1) /

Problem 2.6 A sequential program consists of the following five statements, S₁ through S₅. Considering each statement as a separate process, clearly **identify input set I_i** and **output set O_i** of each process. Restructure the program using Bernstein's conditions in order to achieve maximum parallelism between processes. If any pair of processes cannot be executed concurrently, specify which of the three conditions is not satisfied.

```

S1:      A = B + C
S2:      C = B * D
S3:      S = 0
S4:      Do I = A, 100
           S = S + X(I)
           End Do
S5:      IF (S .GT. 1000) C = C x 2

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

equivalence among the Omega, Flip, and Baseline networks.

- Prove that the Omega network (Fig. 2.24) is **topologically equivalent** to the Baseline network (Fig. 2.25b).
- The *Flip network* (Fig. 2.27) is constructed using inverse perfect shuffle (Fig. 2.14b) for interstage connections. Prove that the Flip network is topologically equivalent to the Baseline network.
- Based on the results obtained in (a) and (b). prove the topological **equivalence** between the Flip network and the Omega network.

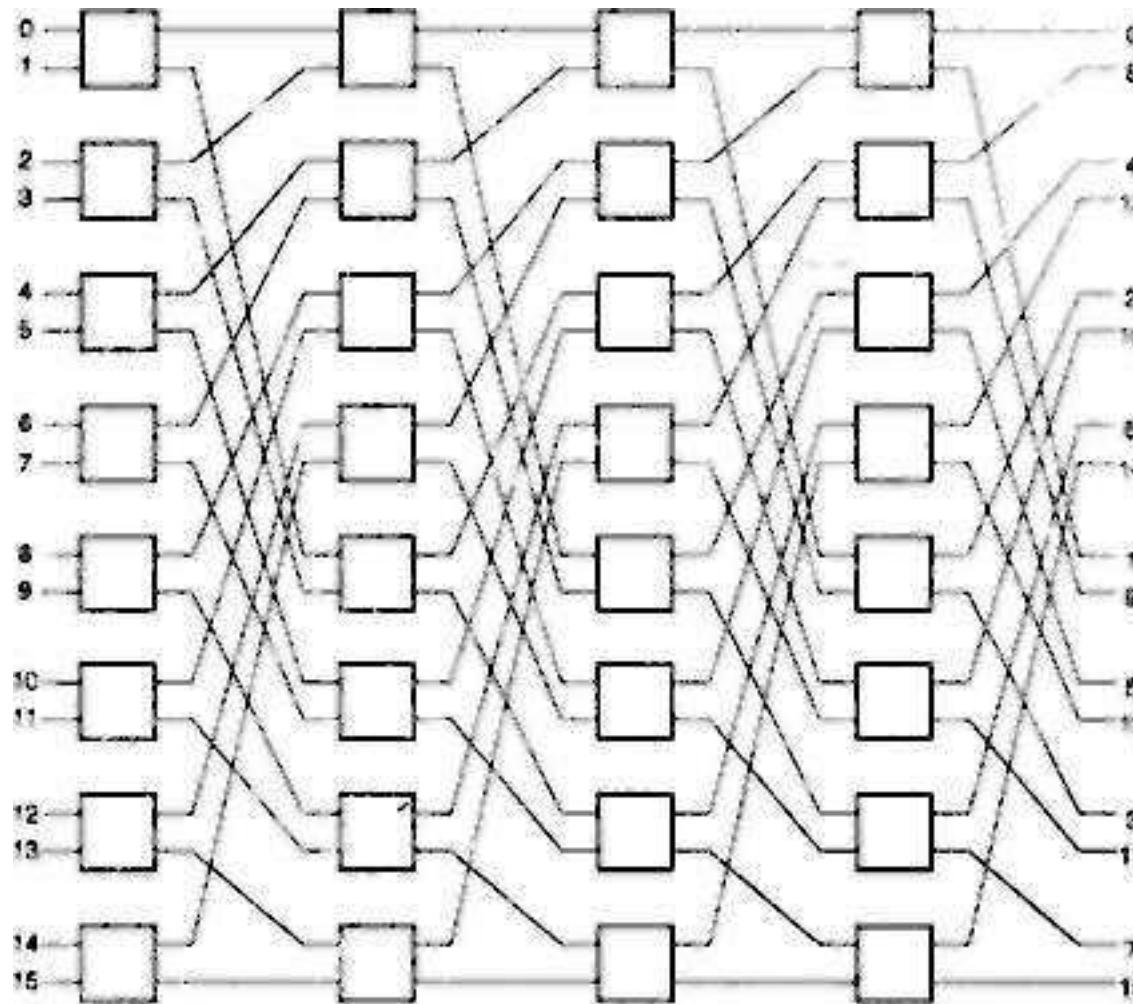


Figure 2.27 A 16 x 16 **Flip** network. (Courtesy of Ken Batcher; reprinted from *Proc. Int. Conf. Parallel Processing*, 1976)

Problem 2.16 Answer the following questions for the fc-ary n-cube network:

- How many nodes are there?
- What is the network diameter?
- What is the bisection bandwidth?
- What is the node degree?
- Explain the graph-theoretic relationship among fc-ary n-cube networks and rings, meshes, tori, binary **n-cubes**, and Omega networks.
- Explain the difference between a conventional torus and a folded torus.
- Under the assumption of constant wire bisection, why do **low-dimensional** networks



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

various application viewpoints and under different system limitations.

Degree of Parallelism The execution of a program on a parallel computer may use different numbers of processors at different time periods during the execution cycle. For each time period, the number of processors used to execute a program is defined as the *degree of parallelism* (DOP). This is a discrete time **function**, assuming only **nonnegative** integer values.

The plot of the DOP as a function of time is called the *parallelism profile* of a given program. For simplicity, we concentrate on the analysis of **single-program profiles**. Some software tools are available to trace the parallelism profile. The profiling of multiple programs in an interleaved fashion can be easily **extended** from this study.

Fluctuation of the profile **during** an observation period depends on the algorithmic structure, program optimization, resource utilization, and run-time conditions of a computer system. The DOP was defined under the assumption of having an unbounded number of available processors and other necessary resources. The DOP may not always be achievable on a real computer with limited resources.

When the DOP exceeds the maximum number of available processors in a system, some parallel branches must be executed in **chunks** sequentially. However, parallelism **still** exists within each chunk, limited by the machine size. The DOP may be also limited by memory and by other nonprocessor resources. We consider only the limit imposed by processors in future discussions on speedup models.

Average Parallelism In what follows, we consider a parallel computer consisting of n homogeneous processors. The maximum parallelism in a profile is m . In the ideal **case**, $n \gg T_0$. The *computing capacity* A of a single processor is approximated by the execution **rate**, such as MIPS or Mflops, without considering the penalties from memory access, communication latency, or system overhead. When i processors are busy during an observation period, we have DOP = i .

The total amount of work W (instructions or computations) performed is proportional to the area under the profile curve:

$$W = \Delta \int_{t_1}^{t_2} DOP(t) dt \quad (3.1)$$

This integral is often computed with the following discrete summation:

$$W = \Delta \sum_{i=1}^m i \cdot t_i \quad (3.2)$$

where t_i is the total amount of time that DOP = i and $\sum_{i=1}^m t_i = t_2 - t_1$ is the total elapsed time.

The *average parallelism* A is computed by

$$A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOP(t) dt \quad (3.3)$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

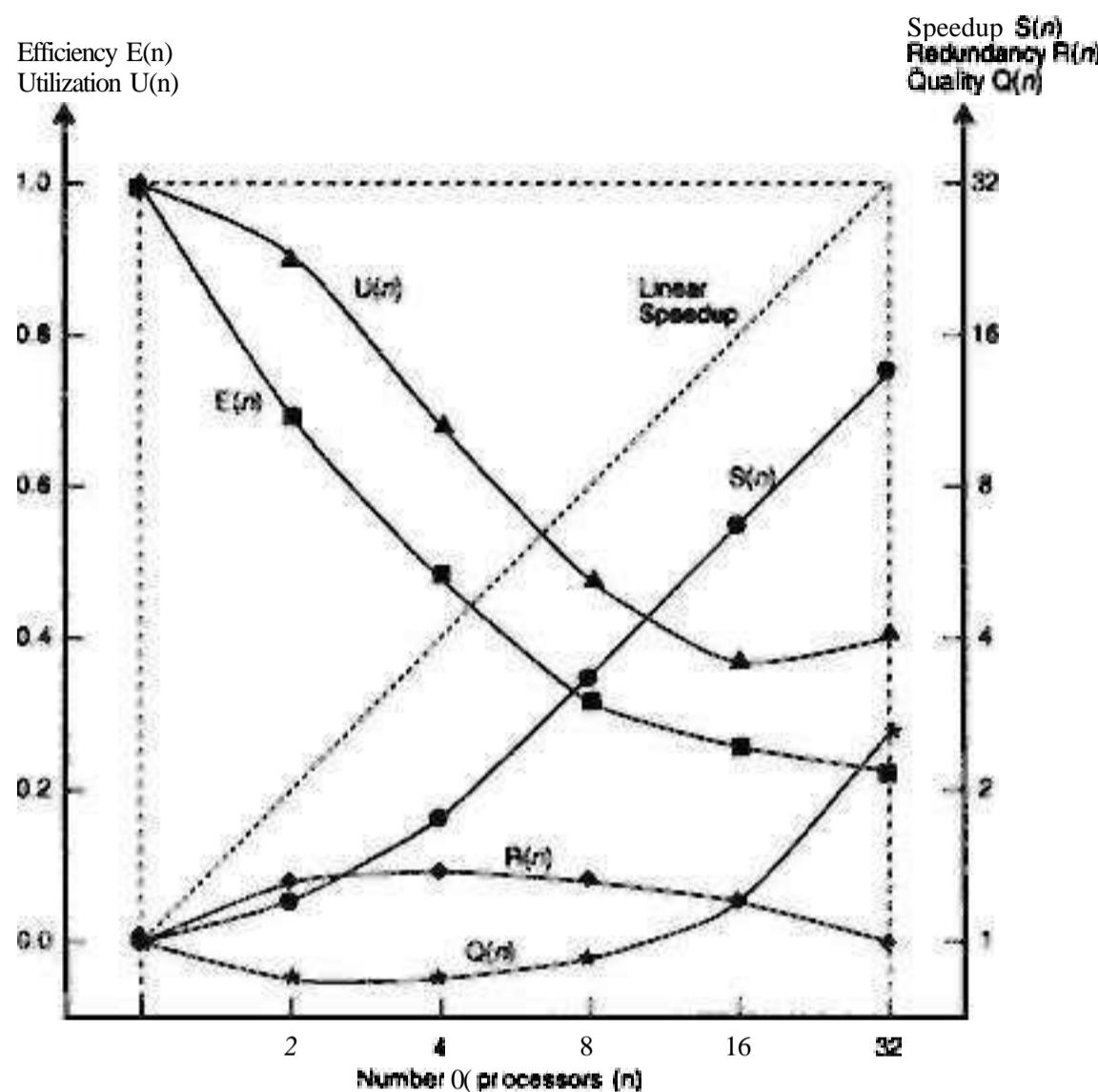


Figure 3.4 Performance measures for Example 3.3 on a parallel computer with up to 32 processors.

To summarize the above discussion on performance indices, we use the speedup $S(n)$ to indicate the degree of speed gain in a parallel computation. The efficiency $E(n)$ measures the useful portion of the total work performed by n processors. The redundancy $R(n)$ measures the extent of workload increase.

The utilization $U(n)$ indicates the extent to which resources are utilized during a parallel computation. Finally, the quality $Q(n)$ combines the effects of speedup, efficiency, and redundancy into a single expression to assess the relative merit of a parallel computation on a computer system.

The speedup and efficiency of 10 parallel computers are reported in Table 3.1 for solving a linear system of 1000 equations. The table entries are excerpts from Table 2 in Dongarra's report (1992) on LINPACK benchmark performance over a large number of computers.

Either the standard LINPACK algorithm or an algorithm based on matrix-matrix multiplication was used in these experiments. A high degree of parallelism is embedded in these experiments. Thus high efficiency (such as 0.94 for the IBM 3090/600S VF and 0.95 for the Convex C3240) was achieved. The low efficiency on the Intel Delta was based on some initial data.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the overall execution time in an optimization process.

Exploiting parallelism for higher **performance** demands both scalable architectures and scalable algorithms. The architectural scalability can be limited by long communication latency, **bounded** memory capacity, bounded I/O bandwidth, and limited CPU speed. How to achieve a balanced design among these practical constraints is the major challenge of today's MPP designers. On the other hand, parallel algorithms and efficient data structures also need to be scalable.

3.2.3 Scalability of Parallel Algorithms

In this subsection, we analyze the scalability of parallel algorithms with respect to key machine classes. An **isoefficiency** concept is introduced for scalability analysis of parallel algorithms. Two examples are used to illustrate the idea. Further studies of scalability are given in Section 3.4 after we study the speedup performance laws in Section 3.3.

Algorithmic Characteristics Computational algorithms are traditionally executed sequentially on uniprocessors. **Parallel algorithms** are those specially devised for parallel computers. The idealized parallel algorithms are those written for the PRAM models if no physical constraints or communication overheads **are** imposed. In the real world, an algorithm is considered efficient only if it can be cost effectively implemented on physical machines. In this sense, all **machine-implementable algorithms** must be **architecture-dependent**. This means the effects of **communicaiton** overhead and architectural constraints cannot be ignored.

We summarize below important characteristics of parallel algorithms which **are** machine implementable:

- (1) *Deterministic versus nondeterministic*: As defined in Section 1.4.1, only deterministic algorithms are implementable on real machines. Our study is **confined** to deterministic algorithms with polynomial time complexity
- (2) *Computational granularity*: As introduced in Section 2.2.1, granularity decides the size of data items and program modules used in computation. In this sense, we also classify algorithms as **fine-grain**, **medium-grain**, or **coarse-grain**.
- (3) *Parallelism profile*: The distribution of the degree of parallelism in an algorithm reveals the opportunity for parallel processing. This often affects the effectiveness of the parallel algorithms.
- (4) *Communication patterns and synchronization requirements*: Communication patterns address both memory access and interprocessor **com muni cat ions**. The patterns can be *static* or *dynamic*, depending on the algorithms. Static algorithms are more suitable for **SIMD** or pipelined machines, while dynamic algorithms are for **MIMD** machines. The *synchronization frequency* often affects the efficiency of an algorithm.
- (5) *Uniformity of the operations*: This refers to the types of fundamental operations to be performed. Obviously, if the operations are uniform across the data set, the SIMD processing or pipelining may be more desirable. In other words, randomly

structured algorithms are more suitable for **MIMD processing**. Other related issues include data types and precision desired.

(6) *Memory requirement and data structures*: In solving large-scale problems, the data sets may require huge memory space. *Memory efficiency* is affected by data structures chosen and data movement patterns in the algorithms. Both time and space complexities are key measures of the granularity of a parallel algorithm.

The Isoefficiency Concept The workload w of an algorithm often grows in the order $O(s)$, where s is the problem size. Thus, we denote the workload to $= w(s)$ as a function of s . Kumar and Rao (1987) have introduced an *isoefficiency* concept relating workload to machine size n needed to maintain a fixed efficiency E when implementing a parallel algorithm on a parallel computer. Let k be the total communication overhead involved in the algorithm implementation. This overhead is usually a function of both machine size and problem size, thus denoted $k = h(s, n)$.

The efficiency of a parallel algorithm implemented on a given parallel computer is thus defined as

$$E = \frac{w(s)}{w(s) + h(s, n)} \quad (3.22)$$

The workload $w(s)$ corresponds to useful computations while the overhead $h(s, n)$ are useless computations attributed to synchronization and data communications delays. In general, the overhead increases with respect to both increasing values of s and n . Thus, the **efficiency** is always less than 1. The question is hinged on relative growth rates between $w(s)$ and $h(s, n)$.

With a fixed problem size (or fixed workload), the efficiency decreases as n increase. The reason is that the overhead $h(s, n)$ increases with n . With a fixed machine size, the overhead h grows slower than the workload w . Thus the efficiency increases with increasing problem size for a fixed-size machine. Therefore, one can expect to maintain a constant efficiency if the workload w is allowed to grow properly with increasing machine size.

For a given algorithm, the workload w might need to grow polynomially or exponentially with respect to n in order to maintain a fixed efficiency. Different algorithms may require different workload growth rates to keep the efficiency from dropping, as n is increased. The isoefficiency functions of common parallel algorithms are polynomial functions of n ; i.e., they are $O(n^k)$ for some $k > 1$. The smaller the power of n in the isoefficiency function, the more scalable the parallel system. Here, the system includes the algorithm and architecture combination.

Isoefficiency Function We can rewrite Eq. 3.22 as $E = 1/(1 + h(s, n)/w(s))$. In order to maintain a constant E , the workload $w(s)$ should grow in proportion to the overhead $h(s, n)$. This leads to the following condition:

$$w(s) = \frac{x}{1 - E} \quad ft(a, n) \quad (3.23)$$

The factor $C = E/(1 - E)$ is a constant for a fixed **efficiency** E . Thus we can define the



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

in a parallel computer, the fixed load is distributed to more processors for parallel execution. Therefore, the main objective is to produce the results as soon as possible. In other words, minimal turnaround time is the primary goal. Speedup obtained for **time-critical** applications is called fixed-load speedup.

Fixed-Load Speedup The ideal speedup formula given in Eq. 3.7 is based on a fixed workload, regardless of the machine size. **Traditional** formulations for speedup, including Amdahl's law, are all based on a fixed problem size and thus on a fixed load. The speedup factor is upper-bounded by a sequential bottleneck in this case.

We consider below both the cases of $DOP < n$ and of $DOP > n$. We use the ceiling function $\lceil x \rceil$ to represent the smallest integer that is greater than or equal to the positive real number x . When x is a fraction, $\lceil x \rceil$ equals 1. Consider the case where $DOP = i > n$. Assume all n processors are used to execute W_i exclusively. The execution time of W_i is

$$t_i(n) = g \left\lceil \frac{i}{n} \right\rceil \quad (3.25)$$

Thus the response time is

$$T(n) = \sum_{i=1}^m \frac{W_i}{i \Delta} \left\lceil \frac{i}{n} \right\rceil \quad (3.26)$$

Note that if $i < n$, then $t_i(n) = t_i(\infty) = W_i/i\Delta$. Now, we define the *fixed-load speedup factor* as the ratio of $T(1)$ to $T(n)$:

$$S_n = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^m \frac{W_i}{i}}{\sum_{i=1}^m \frac{W_i}{i} \left\lceil \frac{i}{n} \right\rceil} \quad (3.27)$$

Note that $S_n < S_\infty < A$, by comparing Eqs. 3.4, 3.7, and 3.27.

A number of factors we have ignored may lower the speedup performance. These include communication latencies caused by delayed memory access, interprocessor communications over a bus or a network, or operating system overhead and delay caused by interrupts. Let $Q(n)$ be the lumped sum of all system overheads on an n -processor system. We can rewrite Eq. 3.27 as follows:

$$S_n = \frac{T(1)}{T(n) + Q(n)} = \frac{\sum_{i=1}^m W_i}{\sum_{i=1}^m \frac{W_i}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} \quad (3.28)$$

The overhead delay $Q(n)$ is certainly application-dependent as well as **machine-dependent**. It is very difficult to obtain a closed form for $Q(n)$. Unless otherwise specified, we assume $Q(n) = 0$ to simplify the discussion.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

can rewrite Eq. 3.31 as follows, assuming $Q(n) = 0$,

$$\mathbf{E} W'_i$$

where $W'_n = nW_n$ and $W_1 + W_n = W'_1 + W'_n/n$, corresponding to the fixed-time condition. From Eq. 3.32, the parallel workload W'_n has been scaled to n times W_n in a linear fashion.

The relationship of a scaled workload to Gustafson's scaled speedup is depicted in Fig. 3.9. In fact, Gustafson's law can be restated as follows in terms of $\alpha = W_1$ and $1 - \alpha = W_n$ under the same assumption $W_1 + W_n = 1$ that we have made for Amdahl's

In Fig. 3.9a, we demonstrate the workload scaling situation. Figure 3.9b shows the fixed-time execution style. Figure 3.9c plots S'_n as a function of the sequential portion α of a program running on a system with $n = 1024$ processors.

Note that the slope of the S_n curve in Fig. 3.9c is much flatter than that in Fig. 3.8c. This implies that Gustafson's law does support scalable performance as the machine size increases. The idea is to keep all processors busy by increasing the problem size. When the problem can scale to match available computing power, the sequential fraction is no longer a bottleneck.

3.3.3 Memory-Bounded Speedup Model

Xian-He Sun and Lionel Ni (1993) have developed a **memory-bounded** speedup model which generalizes Amdahl's law and Gustafson's law to maximize the use of both CPU and memory capacities. The idea is to solve the largest possible problem, limited by memory space. This also demands a scaled workload, providing higher speedup, higher accuracy, and better resource utilization.

Memory-Bound Problems Large-scale scientific or engineering computations often require larger memory space. In fact, many applications of parallel computers are memory-bound rather than CPU-bound or **I/O-bound**. This is especially true in a multicomputer system using distributed memory. The local memory attached to each node is relatively small. Therefore, each node can handle only a small subproblem.

When a large number of nodes are used collectively to solve a single large problem, the total memory capacity increases proportionally. This enables the system to solve a scaled problem through program partitioning or replication and domain decomposition of the data set.

- Instead of keeping the execution time fixed, one may want to use up all the increased memory by scaling the problem size further. In other words, if you have adequate



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

under the *global computation model* illustrated in Fig. 3.11a., where all the distributed memories are used as a common memory shared by all processor nodes.

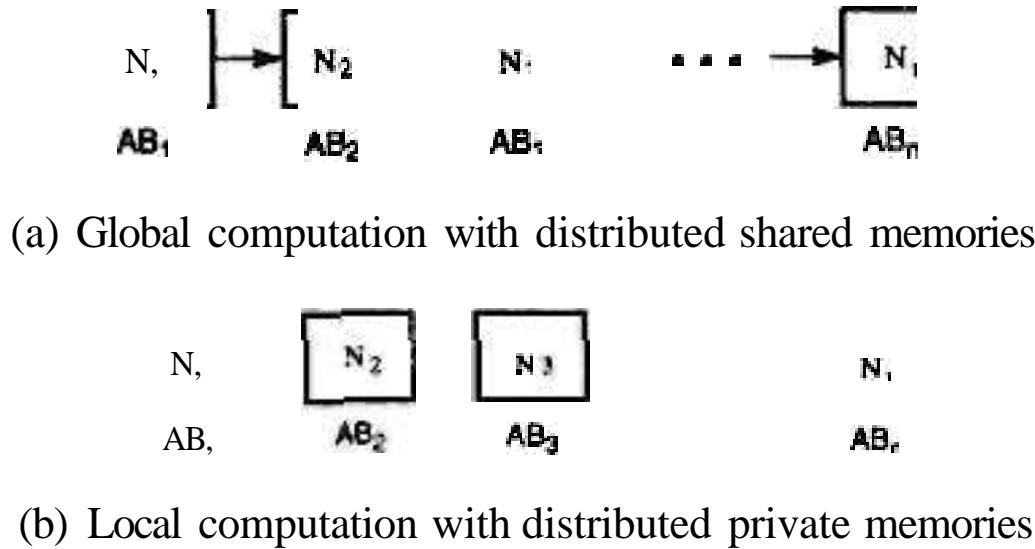


Figure 3.11 Two models for the distributed matrix multiplication.

As illustrated in Fig. 3.11b, the node memories are used locally without sharing. In such a *local computation model*, $G(n) = n$, and we obtain the following speedup:

$$n \cdot \frac{W_1 + nW_n}{W_1 + W_n} \quad (3.37)$$

The above example illustrates Gustafson's scaled speedup for local computation. Comparing the above two speedup expressions, we realize that the fixed-memory speedup (Eq. 3.36) may be higher than the fixed-time speedup (Eq. 3.37). In general, many applications demand the use of a combination of local and global addressing spaces. Data may be distributed in some nodes and duplicated in other nodes.

Data duplication is added deliberately to reduce communication demand. Speedup factors for these applications depend on the ratio between the global and local computations. This reinforces the belief that $G(n)$ increases linearly in general applications. The higher the value of $G(n)$, the better the performance.

3.4 Scalability Analysis and Approaches

The performance of a computer system depends on a large number of factors, all affecting the scalability of the computer architecture and the application program involved. The simplest definition of *scalability* is that the performance of a computer system increases linearly with respect to the number of processors used for a given application.

Scalability analysis of a given computer must be conducted for a given application program/algorithm. The analysis can be performed under different constraints on the



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

where $T_I(s, n)$ is the parallel execution time on the PRAM ignoring all communication overhead. The scalability is defined as follows:

$$\Phi(s, n) = \frac{S(s, n)}{S_I(s, n)} = \frac{T_I(s, n)}{T(s, n)} \quad (3.40)$$

Intuitively, the larger the scalability, the better the performance that the given architecture can yield running the given algorithm. In the ideal case, $S_I(s, n) = n$, the scalability definition in Eq. 3.40 becomes identical to the efficiency definition given in Eq. 3.39.

Example 3.7 Scalability of various machine architectures for parity calculation (Nussbaum and Agarwal, 1991)

Table 3.7 shows the execution times, asymptotic speedups, and scalabilities (with respect to the EREW-PRAM model) of five representative interconnection architectures: linear array, meshes, hypercube, and Omega network, for running a parallel parity calculation.

Table 3.7 Scalability of Various Network-Based Architectures for the Parity Calculation

Metrics	Machine Architecture				
	Linear array	2-D mesh	3-D mesh	Hypercube	Omega Network
$T(s, n)$	$s^{1/2}$	$s^{1/3}$	$s^{1/4}$	$\log s$	$\log^2 s$
$S(s, n)$	$s^{1/2}$	$s^{2/3}$	$s^{3/4}$	$8 / \log 3$	$s / \log^2 s$
$\Phi(s, n)$	$\log s / s^{1/2}$	$\log s / s^{1/3}$	$\log s / s^{1/4}$	1	$1 / \log s$

This calculation examines s bits, determining whether the number of bits set is even or odd using a balanced binary tree. For this algorithm, $T(s, 1) = s$, $T_I(s, n) = \log s$, and $S_I(s, n) = s / \log s$ for the ideal PRAM machine.

On real architectures, the parity algorithm's performance is limited by network diameter. For example, the linear array has a network diameter equal to $n - 1$, yielding a total parallel running time of $s/n + n$. The optimal partition of the problem is to use $n = \sqrt{s}$ processors so that each processor performs the parity check on \sqrt{s} bits locally. This partition gives the best match between computation costs and communication costs with $T(s, n) = s^{1/2}$, $S(s, n) = s^{1/2}$ and thus scalability $\Phi(s, n) = \log s / s^{1/2}$.

The 2D and 3D mesh architectures use a similar partition to match their own communication structure with the computational loads, yielding even better scalability.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

nodes. Based on finite-difference method, solving Eq. 3.41 requires performing the following averaging operation **iteratively** across a very large grid Fig. 3.14:

$$u_{i,j,k}^{(m)} = \frac{1}{7} u_{i-1,j,k}^{(m-1)} + u_{i+1,j,k}^{(m-1)} + u_{i,j-1,k}^{(m-1)} + u_{i,j+1,k}^{(m-1)} + u_{i,j,k-1}^{(m-1)} + u_{i,j,k+1}^{(m-1)} \quad (3.42)$$

where $1 < i, j, k < N$ and N is the number of grid points along each dimension. In total, there are N^3 grid points in the problem domain to be evaluated **during** each iteration m for $1 < m < M$.

The three-dimensional domain can be partitioned into p subdomains, each having n^3 grid points such that $pn^3 = N^3$, where p is the machine **size**. The computations involved in each **subdomain** are assigned to one node of a multicomputer. Therefore, in each iteration, each node is required to perform **7n** computations as specified in Eq. 3.42.

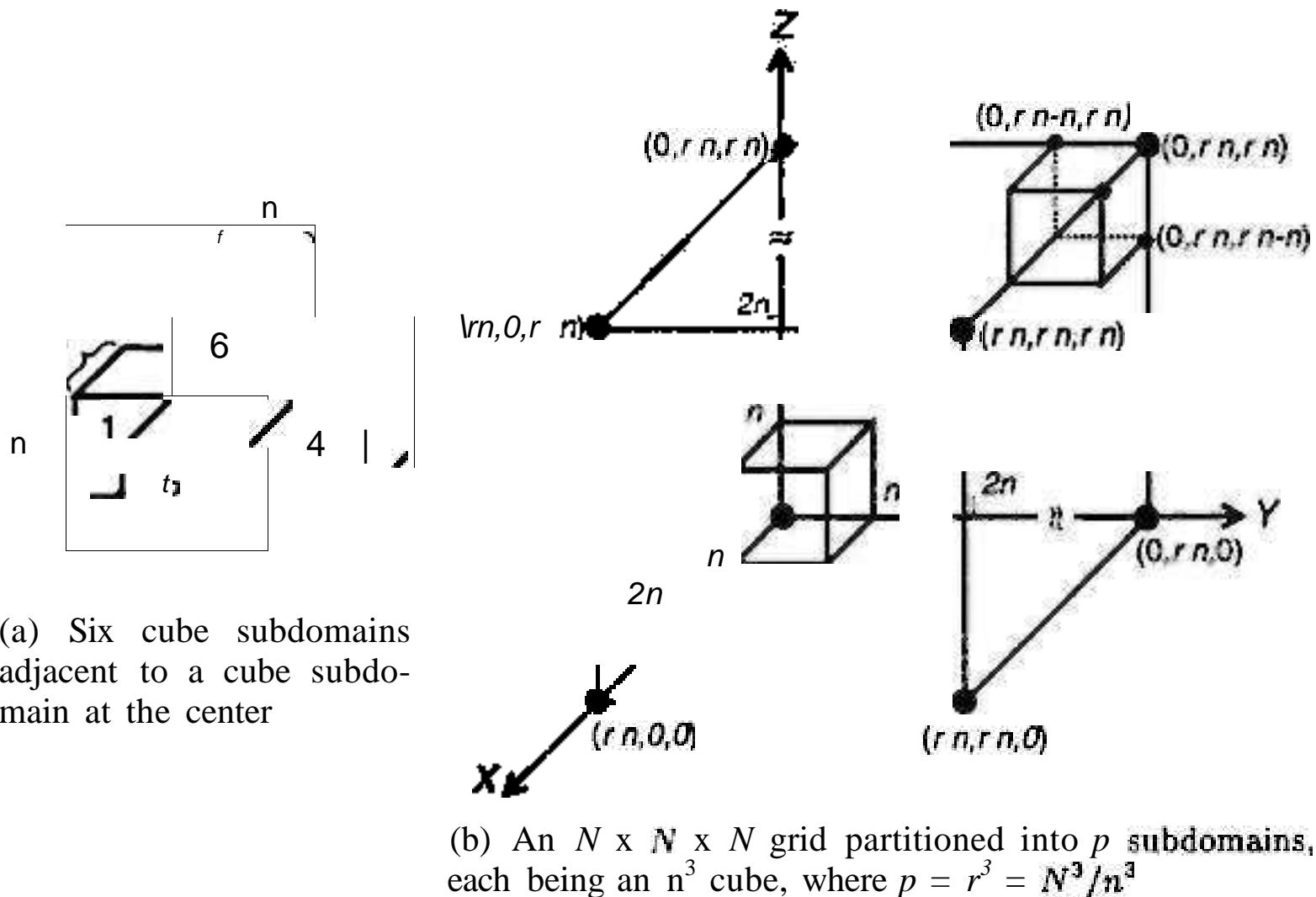


Figure 3.14 Partitioning of a 3D domain for solving the Laplace equation.

Each subdomain is adjacent to six other subdomains (Fig. 3.14a). Therefore, in each iteration, each node needs to exchange (send or receive) a total of $6n^2$ words of floating-point numbers with its neighbors. Assume each floating-point number is **double-precision** (64bits, or 8 bytes). Each processing node has the capability of performing 100 Mflops (or 0.01 μ s per floating-point operation). The internode communication latency is assumed to be 1 μ s (or 1 megaword/s) for transferring a



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



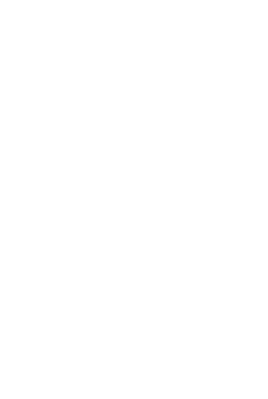
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

cessor. The clock rate of superscalar processors matches that of scalar RISC processors.

The *very long instruction word (VLIW)* architecture uses even more functional units than that of a superscalar processor. Thus the CPI of a VLIW processor can be further lowered. Due to the use of very long instructions (256 to 1024 bits per instruction), VLIW processors have been mostly implemented with microprogrammed control. Thus the clock rate is slow with the use of *read-only memory* (ROM). A large number of microcode access cycles may be needed for some instructions.

Superpipelined processors are those that use multiphase clocks with a much increased clock rate ranging from 100 to 500 MHz. However, the CPI rate is rather high unless superpipelining is practiced jointly with *multiclock* issue. The processors in *vector supercomputers* are mostly superpipelined and use multiple functional units for concurrent scalar and vector operations.

The effective CPI of a processor used in a supercomputer should be very low, positioned at the lower right corner of the design space. However, the cost increases appreciably if a processor design is restricted to the lower right corner. The *subspaces* in which the various processor families will be designed in the future are by no means fixed.

Instruction Pipelines The execution cycle of a typical instruction includes four phases: *fetch*, *decode*, *execute*, and *write-back*. These instruction phases are often executed by an *instruction pipeline* as demonstrated in Fig. 4.2a. In other words, we can simply model an *instruction processor* by such a pipeline structure.

For the time being, we will use an abstract pipeline model for an intuitive explanation of various processor classes. The *pipeline*, like an industrial assembly line, receives successive instructions from its input end and executes them in a streamlined, overlapped fashion as they flow through.

A *pipeline cycle* is intuitively **defined** as the time required for each phase to complete its operation, assuming equal delay in all phases (pipeline stages). Introduced below are the basic definitions associated with instruction pipeline operations:

- (1) *Instruction pipeline cycle* — the clock period of the instruction pipeline.
- (2) *Instruction issue latency* — the time (in cycles) required between the issuing of two adjacent instructions.
- (3) *Instruction issue rate* — the number of instructions issued per cycle, also called the *degree* of a superscalar processor.
- (4) *Simple operation latency* — Simple operations make up the vast majority of instructions executed by the machine, such as *integer adds*, *loads*, *stores*, *branches*, *moves*, etc. On the contrary, complex operations are those requiring an order-of-magnitude longer latency, such as *divides*, *cache misses*, etc. These latencies are **measured** in number of cycles.
- (5) *Resource conflicts* — This refers to the situation where two or more instructions demand use of the same functional unit at the same time.

A *base scalar processor* is defined as a machine with *one* instruction issued per cycle, a one-cycle latency for a simple operation, and a one-cycle latency between instruction issues. The instruction pipeline can be fully utilized if successive instructions can enter



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

making the instruction set very large and very complex. The growth of instruction sets was **also** encouraged by the popularity of microprogrammed control in the 1960s and 1970s. Even user-defined instruction sets were implemented using microcodes in some processors for special-purpose applications.

A typical CISC instruction set contains approximately 120 to 350 instructions using **variable** instruction/data formats, uses a small set of 8 to 24 *general-purpose registers (GPRs)*, and executes a large number of memory reference operations based on more than a dozen addressing modes. Many **HLL** statements are directly implemented in **hardware/firmware** in a CISC architecture. This may simplify the compiler development, improve execution efficiency, and allow an extension from scalar instructions to vector and symbolic instructions.

Reduced Instruction Sets We started with RISC instruction sets and gradually moved to CISC instruction sets **during** the 1980s. After two decades of using CISC processors, computer users began to reevaluate the performance relationship between instruction-set architecture and available hardware/software technology.

Through many years of program tracing, computer scientists realized that only 25% of the instructions of a complex instruction set are frequently used about 95% of the time. This implies that about 75% of **hardware-supported** instructions often are not used at all. A natural question then popped up: Why should we waste valuable chip area for rarely used instructions?

With low-frequency elaborate instructions demanding long microcodes to execute them, it may be more advantageous to remove them completely from **the** hardware and rely on software to implement them. Even if the software implementation is slow, the net result will be still a plus due to their low frequency of appearance. Pushing rarely used instructions into software will vacate chip areas for building more powerful RISC or superscalar processors, even with on-chip caches or floating-point units.

A RISC instruction set typically contains less than 100 **instructions** with a fixed instruction format (32 bits). Only three to five simple addressing modes are used. **Most** instructions are register-based. Memory access is done by load/store instructions only. A large register file (at least 32) is used to improve fast context switching among multiple users, and most instructions **execute** in one cycle with hardwired control.

Because of the reduction in instruction-set **complexity**, the entire processor is **implementable** on a single VLSI chip. The resulting benefits include a higher clock rate and a lower **CPI**, which lead to higher MIPS ratings as reported on commercially available **RISC/superscalar** processors.

Architectural Distinctions Hardware features built into CISC and RISC processors are compared below. Figure 4.4 shows the architectural distinctions between modern CISC and traditional RISC. Some of the distinctions may disappear, because future processors may be designed with features from both types.

Conventional CISC architecture uses a unified cache for holding both instructions and data. Therefore, they must share the same data/instruction path. In a RISC **processor**, separate **instruction** and **data caches** are used with different access paths. However, exceptions do exist. In other words, CISC processors may also use split codes.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

A *translation lookaside buffer* (TLB) is used in the memory control unit for fast generation of a physical address from a virtual address. Both integer and floating-point units are pipelined. The performance of the processor pipelines relies heavily on the cache hit ratio and on minimal branching damage to the pipeline flow.

The CPI of a VAX 8600 instruction varies within a wide range from 2 cycles to as high as 20 cycles. For example, both *multiply* and *divide* may tie up the execution unit for a large number of cycles. This is caused by the use of long sequences of microinstructions to control hardware operations.

The general philosophy of designing a CISC processor is to implement useful instructions in hardware/firmware which may result in a shorter program length with a lower software overhead. However, this advantage has been obtained at the expense of a lower clock rate and a higher CPI, which may not pay off at all.

The VAX 8600 was improved from the earlier VAX/11 Series. The system was later further upgraded to the VAX 9000 Series offering both vector hardware and multiprocessor options. All the VAX Series have used a paging technique to allocate the physical memory to user programs.

CISC Microprocessor Families In 1971, the Intel 4004 appeared as the first microprocessor based on a 4-bit ALU. Since then, Intel has produced the 8-bit 8008, 8080, and 8085. Intel's 16-bit processors appeared in 1978 as the 8086, 8088, 80186, and 80286. In 1985, the 80386 appeared as a 32-bit machine. The 80486 and 80586 are the latest 32-bit micros in the Intel 80x86 family.

Motorola produced its first 8-bit micro, the MC6800, in 1974, then moved to the 16-bit 68000 in 1979, and then to the 32-bit 68020 in 1984. The latest are the MC68030 and MC68040 in the Motorola MC680x0 family. National Semiconductor's latest 32-bit micro is the NS 32332 introduced in 1988. These CISC microprocessor families are widely used in the *personal computer* (PC) industry.

In recent years, the parallel computer industry has begun to build experimental systems with a large number of open-architecture microprocessors. Both CISC and RISC microprocessors have been employed in these efforts. One thing worthy of mention is the compatibility of new models with the old ones in each of the families. This makes it easier to port software along the series of models.

Example 4.2 The Motorola MC68040 microprocessor architecture

The MC68040 is a 0.8- μm HCMOS microprocessor containing more than 1.2 million transistors, comparable to the i80486. Figure 4.6 shows the MC68040 architecture. The processor implements over 100 instructions using 16 general-purpose registers, a 4-Kbyte data cache, and a 4-Kbyte instruction cache, with separate *memory management units (MMUs)* supported by an *address translation cache (ATC)*, which is equivalent to the TLB used in other systems. The data formats range from 8 to 80 bits, based on the IEEE floating-point standard.

Eighteen addressing modes are supported, including register direct and indirect, indexing, memory indirect, program counter indirect, absolute, and immediate



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

Representative RISC Processors Four RISC-based processors, the Sun SPARC, Intel i860, Motorola M88100, and AMD 29000, are summarized in Table 4.4. All of these processors use 32-bit instructions. The instruction sets consist of 51 to 124 basic instructions. **On-chip** floating-point units are built into the i860 and **M88100**, while the SPARC and AMD use off-chip floating-point units. We consider these four processors generic scalar RISC issuing essentially only one instruction per pipeline cycle.

Among the four scalar RISC processors, we choose to examine the Sun SPARC and i860 architectures below. SPARC stands for *scalable processor architecture*. The scalability of the SPARC architecture refers to the use of a different number of **register windows** in different SPARC implementations.

This is different from the M88100, where scalability refers to the number of *special-function units* (SFUs) **implementable** on different versions of the M88000 processor. The Sun SPARC is derived from the original Berkeley RISC design.

Example 4.3 The Sun Microsystems SPARC architecture

The SPARC has been implemented by a number of licensed manufacturers as summarized in Table 4.5. Different technologies and window numbers are used by different SPARC **manufacturers**.

Table 4.5 SPARC Implementations by Licensed Manufacturers

SPARC Chip	Technology	Clock Rate (MHz)	Claimed VAX MIPS	Remarks
Cypress CY7C601 IU	0.8-μm CMOS IV, 207 pins.	33	24	CY7C602 FPU delivers 4.5 Mflops DP Linpack, CY7C604 Cache/MMC, CY7C157 Cache.
Fujitsu MB 86901 IU	1.2-μm CMOS, 179 pins.	25	15	MB 86911 FPC FPC and TI 8847 FPP, MB86920 MMU, 2.7 Mflops DP Linpack by FPU.
LSI Logic L64811	1.0-μm HCMOS, 179 pins.	33	20	L64814 FPU, L64815 MMU.
TI 8846	0.8-μm CMOS	33	24	42 Mflops DP Linpack on TI-8847 FPP.
BIT IU B-3100	ECL family.	80	50	15 Mflops DP Linpack on FPUs: B-3120 ALU, B-3611 FP Multiply/Divide.

All of these manufacturers implement the *floating-point unit* (FPU) on a separate coprocessor chip. The SPARC processor architecture contains essentially a



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

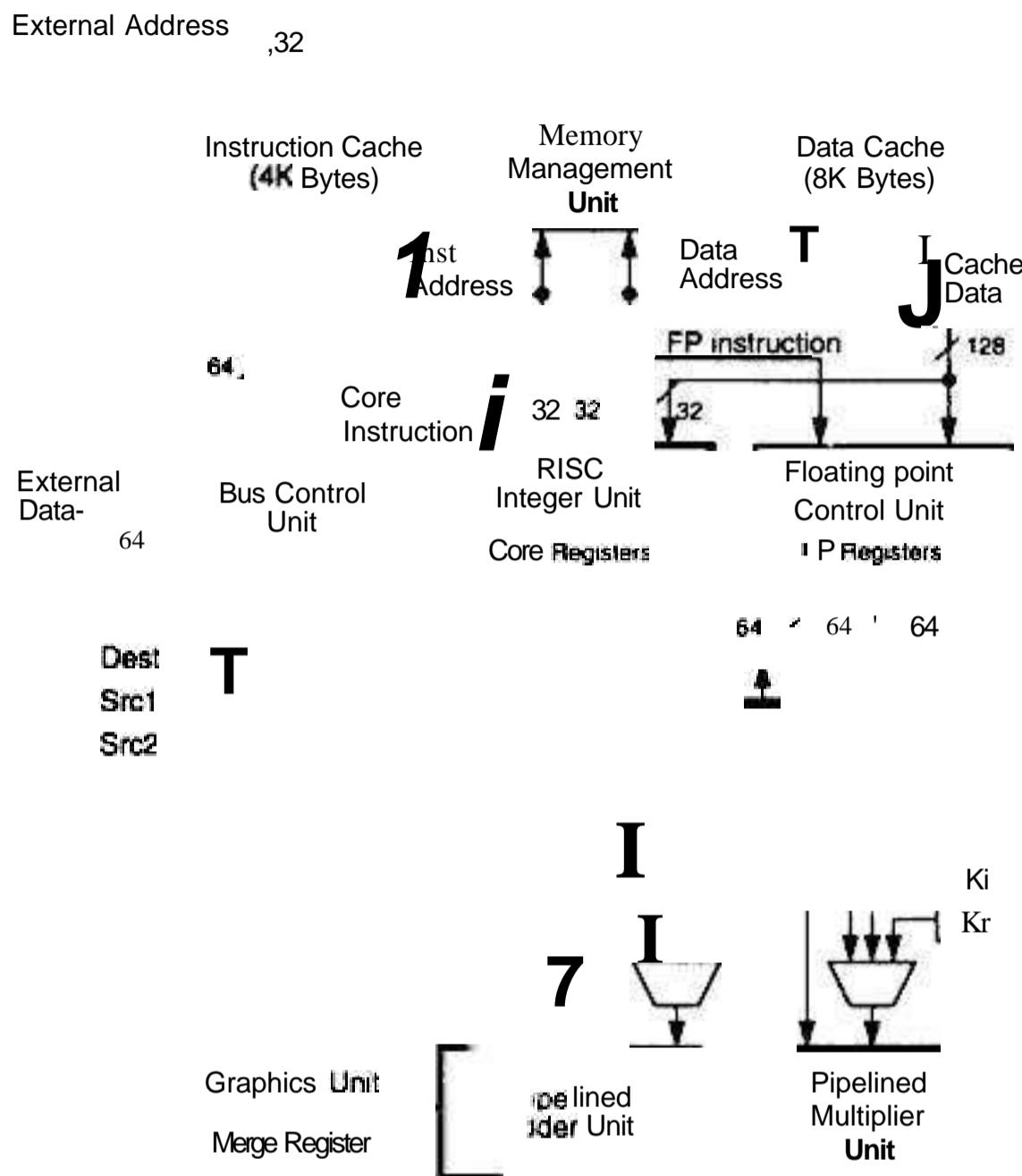


Figure 4.0 Functional units and data paths of the Intel i860 RISC microprocessor.
(Courtesy of Intel Corporation, 1990)

multiprocessor system, permitting the development of compatible OS kernels. The RISC integer unit executes *load*, *store*, *integer*, *bit*, and *control* instructions and fetches instructions for the floating-point control unit as well.

There are two floating-point units, namely, the *multiplier unit* and the *adder unit*, which can be used separately or simultaneously under the coordination of the floating-point control unit. Special dual-operation floating-point instructions such as *add-and-multiply* and *subtract-and-multiply* use both the multiplier and adder units in parallel (Fig. 4.10).

Furthermore, both the integer unit and the floating-point control unit can execute concurrently. In this sense, the i860 is also a superscalar RISC processor capable of executing two instructions, one integer and one floating-point, at the same time. The floating-point unit conforms to the IEEE 754 floating-point standard, operating with *single-precision* (32-bit) and *double-precision* (64-bit) operands.

The graphics unit executes integer operations corresponding to 8-, 16-, or 32-bit pixel data types. This unit supports three-dimensional drawing in a graphics frame



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Table 4.6 Representative Superscalar Processors

Feature	Intel i960CA	IBM RS/6000	DEC 21064
Technology, clock rate, year	25 MHz 1986.	1- μ m CMOS technology, 30 MHz, 1990.	0.75- μ m CMOS, 150 MHz, 431 pins, 1992.
Functional units and multiple instruction issues	Issue up to 3 instructions (register, memory, and control) per cycle, seven functional units available for concurrent use.	POWER architecture, issue 4 instructions (1 FXU, 1 FPU, and 2 ICUs) per cycle.	Alpha architecture, issue 2 instructions per cycle, 64-bit IU and FPU, 128-bit data bus, and 34-bit address bus implemented in initial version.
Registers, caches, MMU, address space	1-KB I-cache, 1.5-KB RAM, 4-channel I/O with DMA, parallel decode, multiported registers.	32 32-bit GPRs, 8-KB I-cache, 64-KB D-cache with separate TLBs.	32 64-bit GPRs, 8-KB I-cache, 8-KB D-cache, 64-bit virtual space designed, 43-bit address space implemented in initial version.
Floating-point unit and functions	On-chip FPU, fast multimode interrupt, multitask control.	On-chip FPU 64-bit multiply, add, divide, subtract, IEEE 754 standard.	On-chip FPU, 32 64-bit FP registers, 10-stage pipeline, IEEE and VAX FP standards.
Claimed performance and remarks	30 VAX/MIPS peak at 25 MHz, real-time embedded system control, and multiprocessor applications.	34 MIPS and 11 Mflops at 25 MHz on POWERstation 530.	300 MIPS peak and 150 Mflops peak at 150 MHz, multiprocessor and cache coherence support.

Note: KB = Kbytes, FP = floating point.

for a RISC processor is shown in Fig. 4.12.

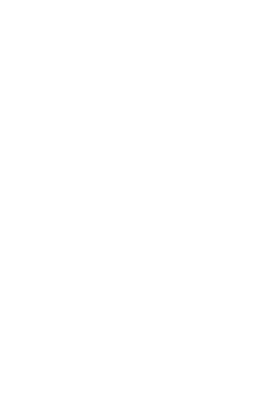
Multiple instruction pipelines are used. The instruction cache supplies multiple instructions per fetch. However, the actual number of instructions issued to various functional units may vary in each cycle. The number is constrained by data dependences and resource conflicts among instructions that are simultaneously decoded. Multiple functional units are built into the integer unit and into the floating-point unit.

Multiple data buses exist among the functional units. In theory, all functional units can be simultaneously used if conflicts and dependences do not exist among them during a given cycle.

Representative Superscalar Processors A number of commercially available processors have been implemented with the superscalar architecture. Notable ones include the IBM RS/6000, DEC 21064, and Intel i960CA processors as summarized in Table 4.6.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

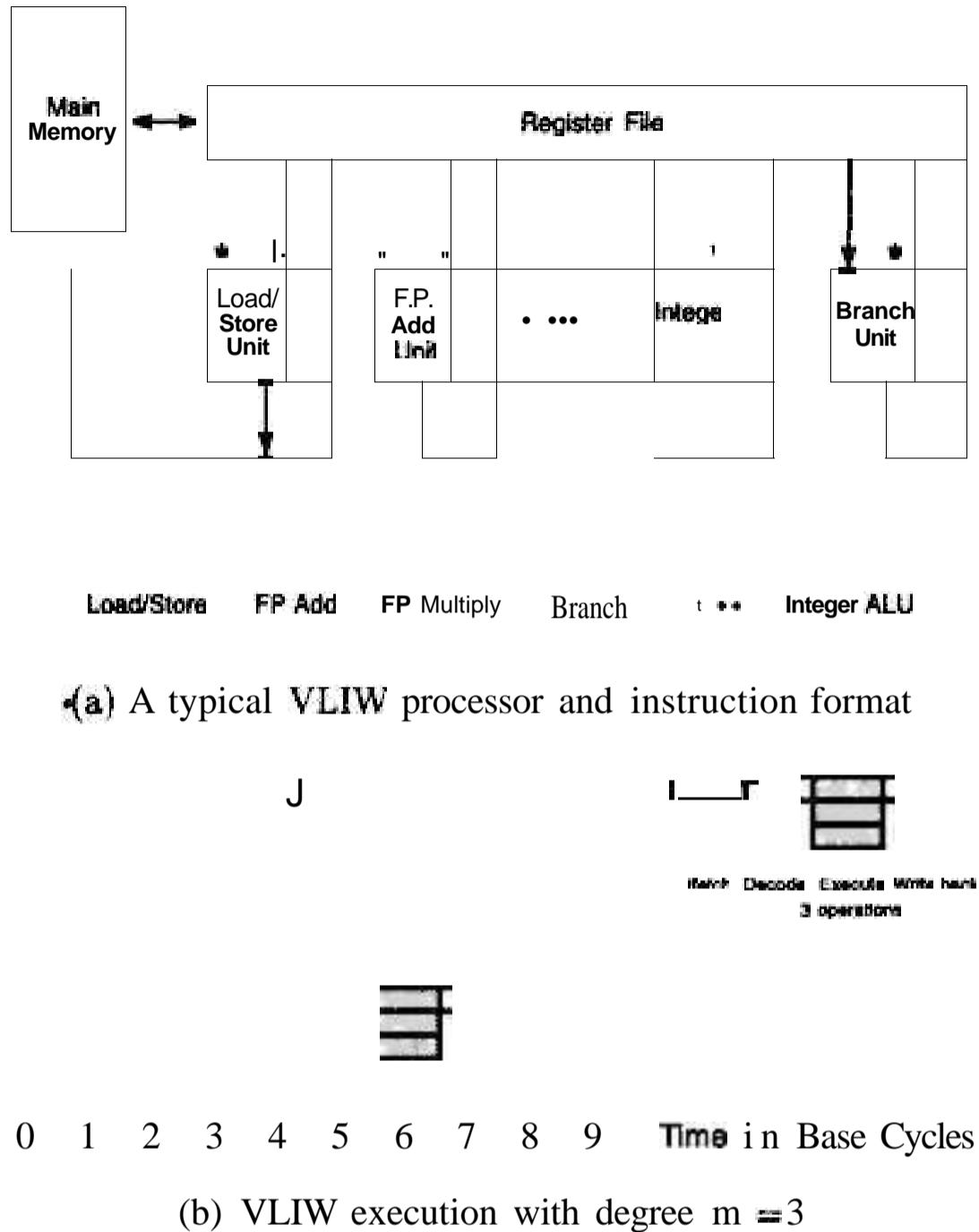


Figure 4.14 The architecture of a very long instruction word (VLIW) processor and its pipeline operations. (Courtesy of Multiflow Computer, Inc., 1987)

to seven operations to be executed concurrently with 256 bits per VLIW instruction.

VLIW Opportunities In a VLIW architecture, random **parallelism** among scalar operations is exploited instead of regular or synchronous parallelism as in a **vectorized** supercomputer or in an **SIMD** computer. The success of a **VLIW** processor depends heavily on the efficiency in code compaction. The architecture is totally incompatible with that of any conventional **general-purpose** processor.

Furthermore, the instruction parallelism embedded in the compacted code **may** require a different latency to be executed by different functional units even though the instructions are issued at the same time. Therefore, different implementations of the same VLIW architecture may not be binary-compatible with each other.

By explicitly encoding parallelism in the long instruction, a VLIW processor can eliminate the hardware or software needed to detect parallelism. The main advantage



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

of linked lists as the basic data structure makes it possible to implement an automatic garbage collection mechanism.

Table 4.2.3 summarizes the major characteristics of symbolic processing. Instead of dealing with numerical data, symbolic processing deals with logic programs, symbolic lists, objects, scripts, blackboards, production systems, semantic networks, frames, and artificial neural networks.

Primitive operations for artificial intelligence include *search*, *compare*, *logic inference*, *pattern matching*, *unification*, *filtering*, *context*, *retrieval*, *set operations*, *transitive closure*, and *reasoning operations*. These operations demand a special instruction set containing *compare*, *matching*, *logic*, and *symbolic manipulation* operations. Floating-point operations are not often used in these machines.

Example 4.6 The Symbolics 3600 Lisp processor

The processor architecture of the Symbolics 3600 is shown in Fig. 4.16. This is a stack-oriented machine. The division of the overall machine architecture into layers allows the use of a pure stack model to simplify instruction-set design, while implementation is carried out with a stack-oriented machine. Nevertheless most operands are fetched from the stack, so the stack buffer and scratch-pad memories are implemented as fast caches to main memory.

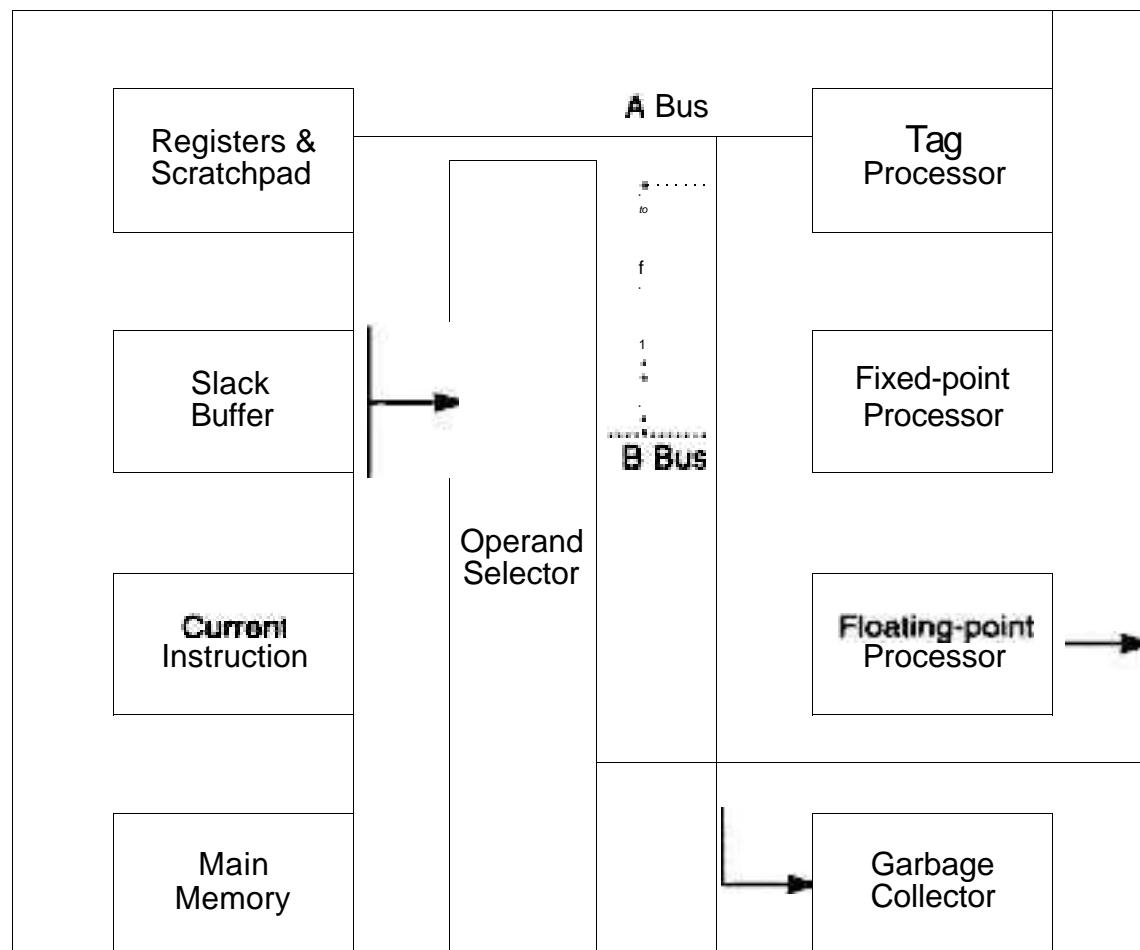


Figure 4.16 The architecture of the Symbolics 3600 Lisp processor. (Courtesy of Symbolics, Inc., 1985)



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

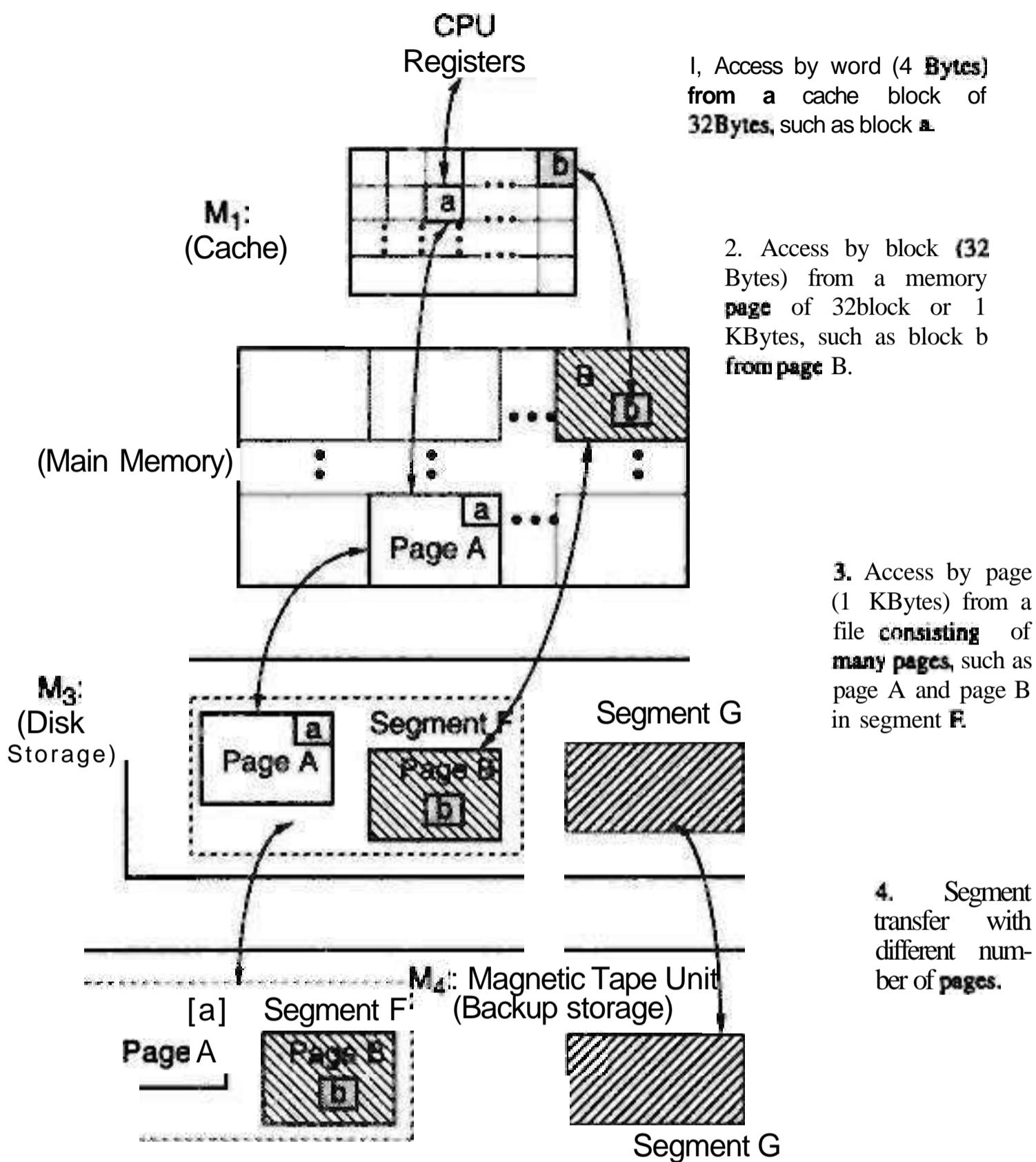


Figure 4.18 The inclusion **property** and data transfers between adjacent levels of a **memory** hierarchy.

The set inclusion relationship implies that all information items are originally stored in the outermost level M_n . During the processing, subsets of M_n are copied into M_{n-1} . Similarly, subsets of M_{n-1} are copied into M_{n-2} , and so on.

In other words, if an information word is found in M_i , then copies of the same word can be also found in all upper levels $M_{i+1}, M_{i+2}, \dots, M_n$. However, a word stored in M_{i+1} may not be found in M_i . A word **miss** in M_i implies that it is also missing from all lower levels $M_{i-1}, M_{i-2}, \dots, M_1$. The highest level is the backup storage, where everything can be found.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

memory are accessed more often than the outer levels.

Effective Access Time In practice, we wish to achieve as high a hit ratio as possible at M_1 . Every time a miss occurs, a penalty must be paid to access the next higher level of memory. The misses have been called *block misses* in the cache and *page faults* in the main memory because blocks and pages are the units of transfer between these levels.

The time penalty for a page fault is much longer than that for a block miss due to the fact that $t_1 < t_2 < t_3$. Stone (1990) has pointed out that a cache miss is 2 to 4 times as costly as a cache hit, but a page fault is 1000 to 10,000 times as costly as a page hit.

Using the access frequencies f_i , for $i = 1, 2, \dots, n$, we can formally define the *effective access time* of a memory hierarchy as follows:

$$\begin{aligned} T_{\text{eff}} &= \sum_{i=1}^n f_i \cdot t_i \\ &= h_1 t_1 + (1 - h_1) t_2 + (1 - h_1)(1 - h_2) t_3 + \dots + \\ &\quad (1 - h_1)(1 - h_2) \dots (1 - h_{n-1}) t_n \end{aligned} \quad (4.3)$$

The first several terms in Eq. 4.3 dominate. Still, the effective access time depends on the program behavior and memory design choices. Only after extensive program trace studies can one estimate the hit ratios and the value of T_{eff} more accurately.

Hierarchy Optimization The total cost of a memory hierarchy is estimated as follows:

$$C_{\text{total}} = \sum_{i=1}^n c_i \cdot s_i \quad (4.4)$$

This implies that the cost is distributed over n levels. Since $c_1 > c_2 > c_3 > \dots > c_n$, we have to choose $s_1 < s_2 < s_3 < \dots < s_n$. The optimal design of a memory hierarchy should result in a T_{eff} close to the t_1 of M_1 and a total cost close to the c_n of M_n . In reality, this is difficult to achieve due to the tradeoffs among n levels.

The optimization process can be formulated as a linear programming problem, given a ceiling C_0 on the total cost — that is, a problem to minimize

subject to the following constraints:

$$s_i > 0, t_i > 0 \quad \text{for } i = 1, 2, \dots, n$$

$$C_{\text{total}} = \sum_{i=1}^n c_i \cdot s_i < C_0 \quad (4.6)$$

As shown in Table 4.7, the unit cost c_i and capacity s_i at each level M_i depend on the speed t_i required. Therefore, the above optimization involves tradeoffs among t_i , c_i ,



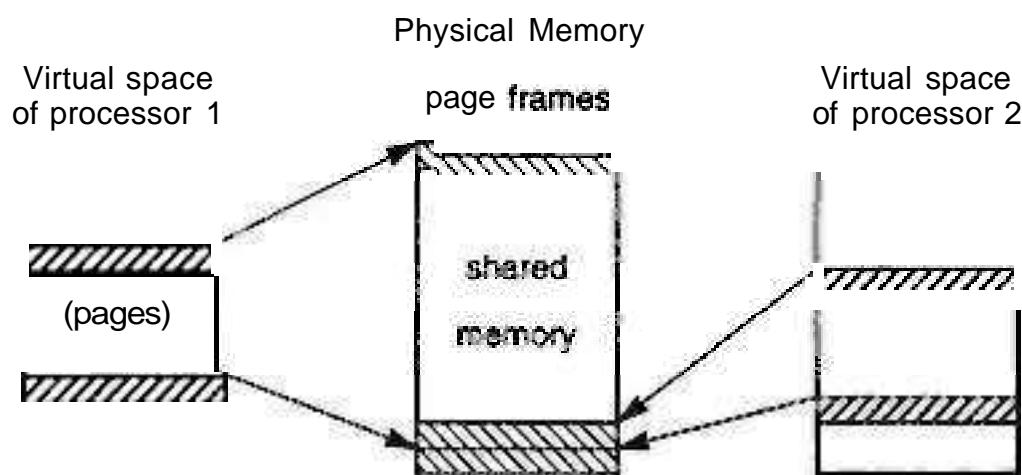
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



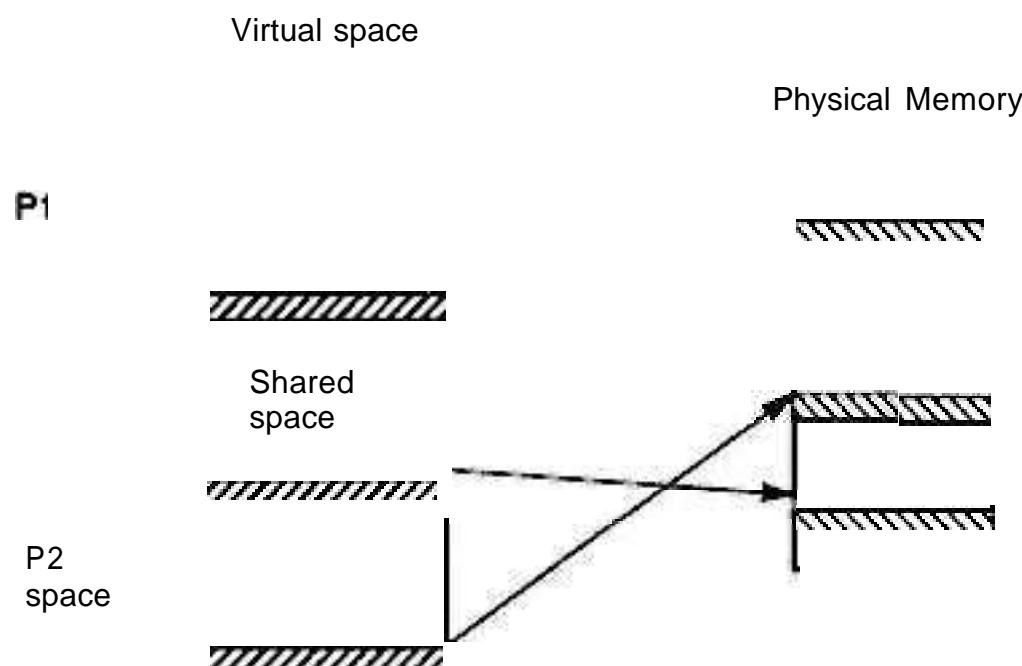
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



(a) Private virtual memory spaces in different processors



(b) Globally shared virtual memory space

Figure 4.20 Two virtual memory models for multiprocessor systems. (Courtesy of Dubois and Briggs, tutorial, *Annual Symposium on Computer Architecture*, 1990)

Translation maps are stored in the cache, in associative memory, or in the main memory. To access these maps, a mapping function is applied to the virtual address. This function generates a pointer to the desired translation map. This mapping can be implemented with a *hashing* or *congruence* function.

Hashing is a simple computer **technique** for converting a long page number into a short one with fewer bits. The hashing function should randomize the virtual page number and produce a unique hashed number to be used as the pointer. The congruence function provides hashing into a linked list.

Translation Lookaside Buffer Translation maps appear in the form of a **translation**



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

that address and, **if it is found**, uses the table index of the matching entry as the address **of** the desired page frame. A hashing table is used to search through the inverted PT. The size of an inverted PT is governed by the size of the physical space, while that of traditional PTs is determined by the size of the **virtual** space. Because of limited **physical** space, no multiple levels are needed for the inverted page table.

Example 4.8 Paging and segmentation in the Intel i486 processor

As with its predecessor in the 80x86 family, the i486 features both segmentation and paging compatibilities. Protected mode increases the linear address from 4 Gbytes (2^{32} bytes) to 64 Tbytes (2^{48} bytes) with four levels of protection. The maximal memory size in real mode is 1 Mbyte (2^{20} bytes). Protected mode allows the i486 to run all software from existing 8086, 80286, and 80386 processors. A segment can have any length from 1 byte to 4 Gbytes, the maximal physical memory size.

A segment can start at any base address, and storage overlapping between segments is allowed. The virtual address (Fig. 4.22a) has a 16-bit segment selector to determine the base address of the *linear address space* to be used with the i486 paging system.

The " 32-bit offset specifies the internal address within **a** segment. The segment descriptor is used to specify access rights and segment size besides selection of the address of **the first byte of the** segment.

The paging feature is optional on the i486. It can be enabled or disabled by **software** control. When paging is enabled, the virtual address is first translated into a linear address and then into the physical address. When paging is disabled, the linear address and physical address are **identical**. When a 4-Gbyte segment is selected, the entire physical memory becomes one large segment, which means the segmentation mechanism is essentially disabled.

In this sense, the i486 can be used with four different memory organizations, *pure paging*, *pure segmentation*, *segmented paging*, or *pure physical addressing* without paging and segmentation.

A 32-entry TLB (Fig 4.22b) is used to convert the linear **address directly** into the physical address without resorting to the two-level paging scheme (Fig 4.22c). The standard page size on the i486 is 4 Kbytes = 2^{12} bytes. Four control registers are used to select between regular paging and page fault handling.

The page table directory (4 Kbytes) allows 1024 page directory entries. Each page table at the second level is 4 Kbytes and holds up to 1024 PTEs. The upper 20 linear address bits are compared to determine if there is a hit. The hit ratios of the **TLB and** of the page tables depend on program behavior and the efficiency of the update (page replacement) policies. A 98% **hit** ratio has been observed in **TLB** operations in the past.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Page Tracing experiments are described below for three page replacement policies: LRU, OPT, and FIFO, respectively. The successive pages loaded in the page frames (PFs) form the trace entries. Initially, all PFs are empty.

		PF	0	•	1	2	4	2	3	7	2	1	3	1	Hit Ratio
LRU	a	0	0	0	0	0	4	4	4	7	7	7	3	3	$\frac{3}{11}$
	b			1	1	1	1	1	3	3	3	1	j	1	
	c				2	2	2	2	2	2	2	2	2	2	
	Faults	*	*	*	*	*	*	*	*	*	*	*	*	*	
OPT	a	0	0	0	0	4	1	3	1	1	7	3	3	3	$\frac{4}{11}$
	b			1	1	1	1	1	1	1	1	1	1	1	
	c				2	2	2	2	2	2	9	2	2	9	
	Faults	*	*	*	*	*	#	*	*	*	*	*	*	*	
FIFO	a	0	!	0	0	0	4	4	4	4	0	2	2	2	$\frac{2}{11}$
	b	1	1	1	1	1	1	3	3	3	1	1	1	1	
	c				2	2	2	2	7	7	7	3	3	3	
	Faults	*	;	*	*	*	#	*	*	*	!	*	i	*	

The above results indicate the superiority of the OPT policy over the others. However, the OPT cannot be implemented in practice. The LRU policy performs better than the FIFO due to the locality of references. From these results, we realize that the LRU is generally better than the FIFO. However, exceptions still exist due to the dependence on program behavior.

Relative Performance The performance of a page replacement algorithm depends on the page trace (program behavior) encountered. The best policy is the OPT algorithm. However, the OPT replacement is not realizable because no one can predict the future page demand in a program.

The LRU algorithm is a popular policy and often results in a high hit ratio. The FIFO and random policies may perform badly because of violation of the program locality.

The circular FIFO policy attempts to approximate the LRU with a simple circular queue implementation. The LFU policy may perform between the LRU and the FIFO policies. However, there is no fixed superiority of any policy over the others because of the dependence on program behavior and run-time status of the page frames.

In general, the page fault rate is a monotonic decreasing function of the size of the resident set $R(t)$ at time t because more resident pages result in a higher hit ratio in the main memory.

Block Replacement Policies The relationship between the cache block frames and cache blocks is similar to that between page frames and pages on a disk. Therefore, those page replacement policies can be modified for *block replacement* when a cache miss occurs.

Different cache organizations (Section 5.1) may offer different flexibilities in implementing some of the replacement algorithms. The cache memory is often associatively searched, while the main memory is randomly addressed.

Due to the difference between page allocation in main memory and block allocation in the cache, the cache hit ratio and memory page hit ratio are affected by the replacement policies differently. *Cache traces* are often needed to evaluate the cache performance. These considerations will be further discussed in Chapter 5.

4.5 Bibliographic Notes and Exercises

Advanced microprocessors were surveyed in the book by [Tabak91]. A tutorial on RISC computers was given by [Stallings90]. Superscalar and superpipelined machines were characterized by Jouppi and Wall [Jouppi89]. [Johnson91] provided an excellent book on superscalar processor design. The **VLIW** architecture was first developed by [Fisher83].

The Berkeley RISC was reported by Patterson and Sequin in [Patterson82]. A MIPS **R2000** overview can be found in [Kane88]. The HP precision architecture has been assessed in [Lee89]. Optimizing compilers for SPARC appears in [Muchnick88]. A further description of the M68040 can be found in [Edenfield90].

A good source of information on the i860 can be found in [Margulis90]. The DEC Alpha architecture is described in [DEC92]. The latest MIPS R4000 was reported by Mirapuri, **Woodacre**, and Vasseghi [Mirapuri92]. The IBM RS/6000 was discussed in [IBM90]. The Hot-Chips Symposium Series [Hotchips91] usually present the latest developments in high-performance processor chips.

The virtual memory models were based on the tutorial by Dubois and Briggs [Dubois90c]. The book by [Hennessy90] and Patterson has treated the memory hierarchy design based on extensive program trace data. Distributed shared virtual memory was surveyed in [Nitzberg91] and **Lo**.

The concept of a working set was introduced by [Denning68]. A linear programming optimization of the memory hierarchy was reported in [Chow74]. [Crawford90] explained the paging and segmentation schemes in the i486. Inverted paging was described by [Chang88] and Mergen. [Cragon92b] has discussed memory systems for pipeline processor design.

Exercises

Problem 4.1 Define the following basic terms related to modern processor technology:

- | | |
|--------------------------------|---|
| (a) Processor design space. | (f) Processor versus coprocessor. |
| (b) Instruction issue latency. | (g) General-purpose registers. |
| (c) Instruction issue rate. | (h) Addressing modes. |
| (d) Simple operation latency. | (i) Unified versus split caches. |
| (e) Resource conflicts. | (j) Hardwired versus microcoded control. |



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Problem 4.11 Consider a **two-level** memory hierarchy, M_1 and M_2 . Denote the hit ratio of M_1 as h . Let c_1 and c_2 be the costs per kilobyte, s_1 and s_2 the memory capacities, and t_1 and t_2 the access times, respectively.

- (a) Under what conditions will the average cost of the entire memory system approach
- (b) What is the effective memory-access time t_a of this hierarchy?
- (c) Let $r = t_2/t_1$ be the speed ratio of the two memories. Let $E = t_1/t_a$ be the *access efficiency* of the memory system. Express E in terms of r and h .
- (d) Plot E against h for $r = 5, 20$, and 100. **respectively**, on grid **paper**.
- (e) What is the required hit ratio h to make $E > 0.95$ if $r = 100$?

Problem 4.12 You are asked to perform capacity planning for a **two-level** memory system. The first level, M_1 , is a cache with three capacity choices of 64 Kbytes, 128 Kbytes, and 256 Kbytes. The second level, M_2 , is a main memory with a **4-Mbyte** capacity. Let c_1 and c_2 be the costs per byte and t_1 and t_2 the access times for M_1 and M_2 , respectively. Assume $c_1 = 20c_2$ and $t_2 = 10t_1$. The cache hit ratios for the three capacities are assumed to be 0.7, **0.9**, and 0.98, respectively.

- (a) What is the average access time t_a in terms of $t_1 = 20$ ns in the three cache designs? (Note that t_1 is the time from CPU to M_1 and t_2 is that from CPU to M_2 , not from M_1 to M_2).
- (b) Express the average byte cost of the entire memory hierarchy if $c_2 = \$0.2/\text{Kbyte}$.
- (c) Compare the three memory designs and indicate the order of merit in terms of average costs and average access times, respectively. Choose the optimal design based on the product of average cost and average access time.

Problem 4.13 Compare the advantages and shortcomings in implementing private virtual memories and a **globally** shared virtual memory in a **multicomputer** system. This comparative study should consider the latency, coherence, page **migration**, protection, implementation, and application problems in the context of building a scalable multicomputer system with distributed shared **memories**.

Problem 4.14 Explain the *inclusion property* and **memory coherence** requirements in a multilevel memory hierarchy. Distinguish between *write-through* and *write-back* policies in maintaining the coherence in adjacent levels. Also explain the basic concepts of *paging* and *segmentation* in managing the physical and virtual memories in a hierarchy.

Problem 4.15 A two-level memory system has eight virtual pages on a disk to be mapped into four page frames (PFs) in the main memory. A certain program generated the following page trace:

1, 0, 2, 2, 1, 7, 6, 7, 0, 1, 2, 0, 3, 0, 4, 5, 1, 5, 2, 4, 5, 6, 7, 6, 7, 2, 4, 2, 7, 3, 3, 2, 3

- (a) Show the successive virtual pages residing in the four page frames with respect to



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The data lines are often proportional to the memory word length.

For example, the revised VME bus specification has 32 address lines and 32 (or 64) data lines. Besides being used in addressing, the 32 address lines can be multiplexed to serve as the lower half of the 64-bit data during data transfer cycles. The DTB control lines are used to indicate read/write, timing control, and bus error conditions.

Bus Arbitration and Control The process of assigning control of the DTB to a requester is called *arbitration*. Dedicated lines are reserved to coordinate the arbitration process among several requesters. The requester is called a *master*, and the receiving end is called a *slave*.

Interrupt lines are used to handle interrupts, which are often **prioritized**. Dedicated lines may be used to synchronize parallel activities among the processor modules. Utility lines include signals that provide periodic timing (clocking) and coordinate the power-up and power-down sequences of the system.

The backplane is made of signal lines and connectors. A special bus controller board is used to house the backplane control logic, such as the system clock driver, arbiter, bus timer, and power driver.

Functional Modules A *functional module* is a collection of electronic circuitry that **resides** on one functional board (Fig. 5.1) and works to achieve special bus control functions. Special **functional** modules are introduced below:

An *arbiter* is a functional module that accepts bus requests from the requester module and grants control of the DTB to one requester at a time.

A *bus timer* measures the time each data transfer takes on the DTB and terminates the DTB cycle if a transfer takes too long.

An *interrupter* module generates an interrupt request and provides status/ID information when an *interrupt handler* module requests it.

A *location monitor* is a functional module that monitors data transfers over the DTB. A *power monitor* watches the status of the power source and signals when power becomes unstable.

A *system clock driver* is a module that provides a clock timing signal on the utility bus. In **addition**, *board interface logic* is needed to match the signal line impedance, the propagation time, and termination values between the backplane and the plug-in boards.

Physical Limitations Due to electrical, mechanical, and packaging limitations, only a limited number of boards can be plugged into a single backplane. Multiple backplane buses can be mounted on the same backplane chassis.

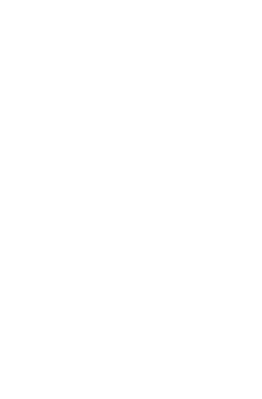
For **example**, the VME chassis can house one to three backplane buses. Two can be used as a shared bus among all processors and memory boards, and the third as a local bus connecting a host processor to additional memory and **I/O** boards. Means of extending a **single-bus** system to build larger multiprocessors will be studied in Section 7.1.1. The bus system is difficult to scale, mainly limited by packaging constraints.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

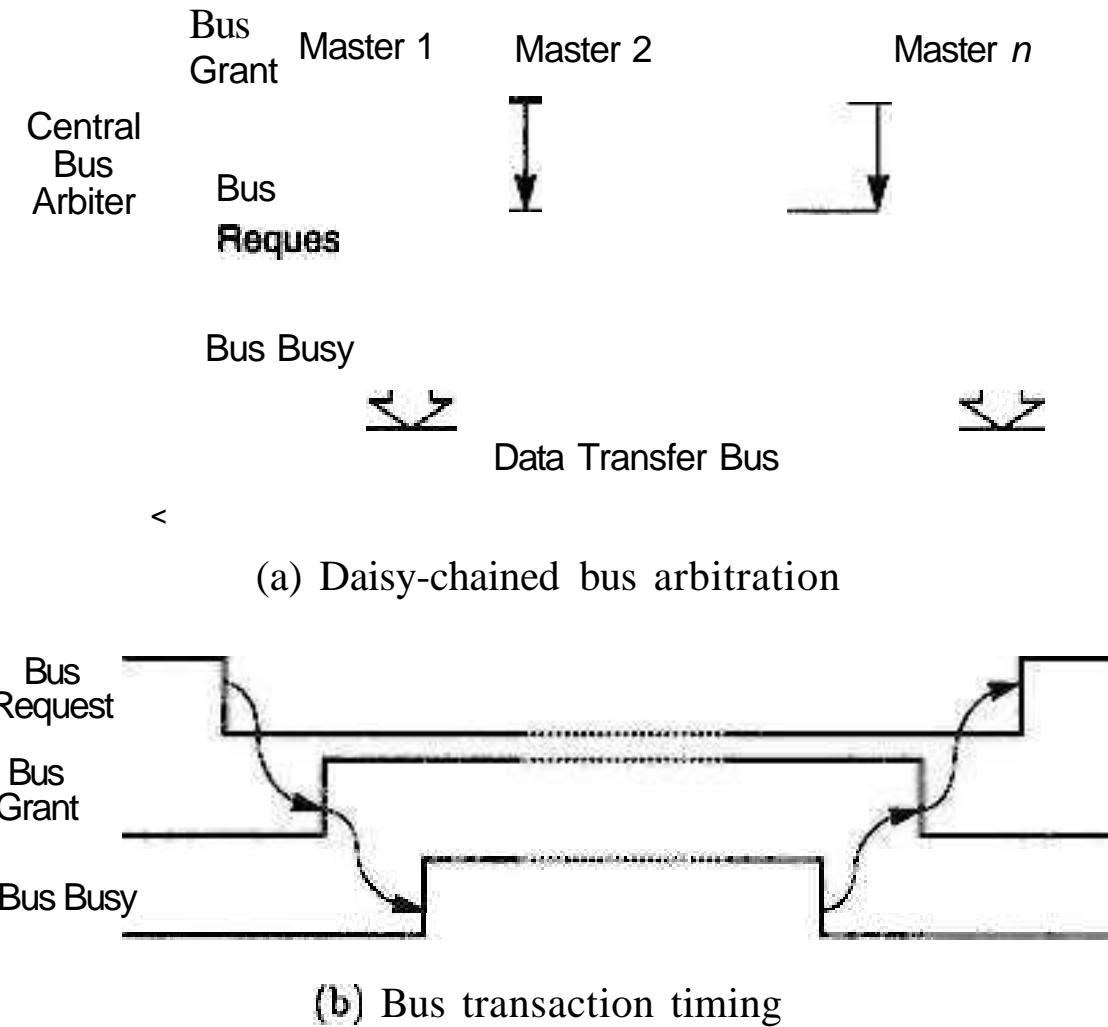


Figure 5.4 Central bus arbitration using shared requests and daisy-chained bus grants with a fixed priority.

be added anywhere in the **daisy chain** by sharing the **same** set of **arbitration** lines. The disadvantage is a fixed-priority sequence violating the fairness practice. Another drawback is its slowness in propagating the *bus-grant* signal along the daisy chain.

Whenever a higher-priority device fails, all the lower-priority devices on the right of the daisy chain cannot use the bus. Bypassing a failing device or a removed device on the daisy chain is desirable. Some new bus standards are specified with such a capability.

Independent Requests and Grants Instead of using shared request and grant lines as in Fig. 5.4, multiple *bus-request* and *bus-grant* signal lines can be independently provided for each potential master as in Fig. 5.5a. No daisy-chaining is used in this scheme.

The arbitration **among** potential masters is still carried out by a central arbiter. However, any priority-based or fairness-based bus allocation policy can be implemented. A multiprocessor system usually uses a priority-based policy for **I/O** transactions and a fairness-based policy among the processors.

In some **asymmetric** multiprocessor architectures, the processors may be assigned different functions, such as serving as a front-end host, an executive processor, or a back-end slave processor. In such cases, a priority policy can also be used among the **processors**.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- (10) Direct support of snoopy **cache-based** multiprocessors with recursive protocols to support large systems interconnected by multiple buses.
- (11) Compatible message-passing protocols with multicomputer connections and **special** application profiles and interface design guides **provided**.

Example 5.1 Signal lines in the proposed Futurebus+ standard

As illustrated in Fig. 5.6, the Futurebus+ consists of **information**, synchronization, bus **arbitration**, and handshake lines that can match different system designs. The 64-bit *address lines* are multiplexed with the lower-order 64-bit *data lines*. Additional data lines can be added to form a data path up to 256 bits wide. The *tag lines* are optional for extending the **addressing/data modes**.

The *command lines* carry command information from the master to one or more slaves. The *status lines* are used by slaves to respond to the master's **commands**. The *capability lines* are used to declare special bus **transactions**. Every byte of lines is protected by at least one **parity-check line**.

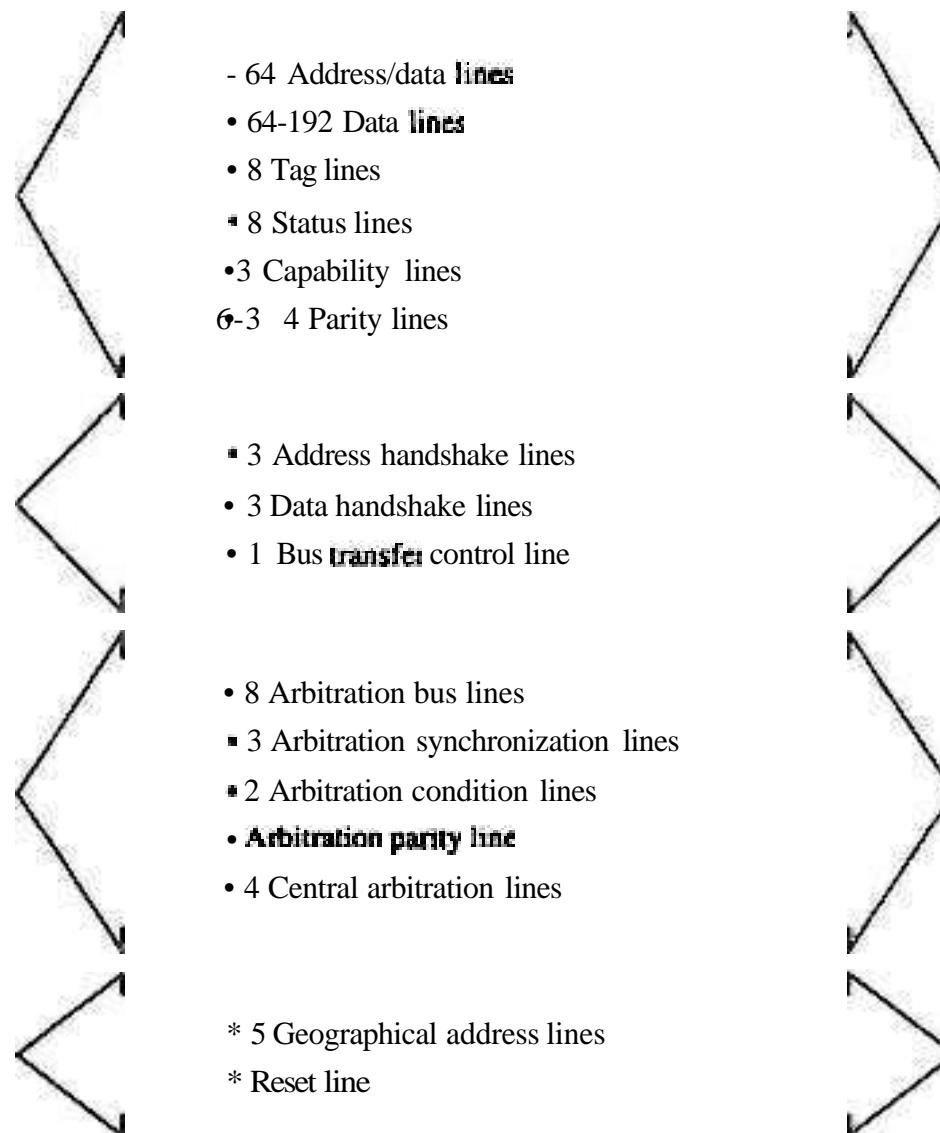


Figure 5.6 The Futurebus+ organization. (Reprinted with permission from IEEE Standard 896.1-1991, copyright © 1991 by IEEE, Inc.)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

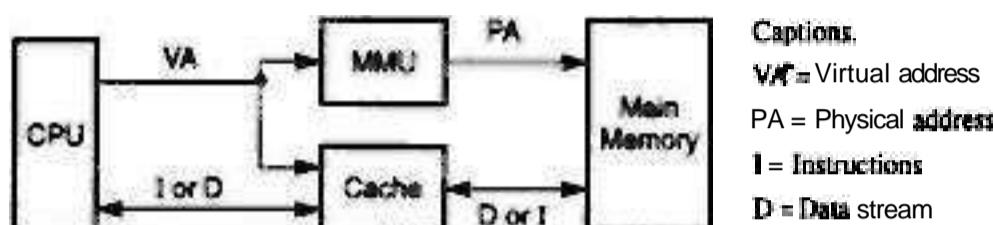


You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

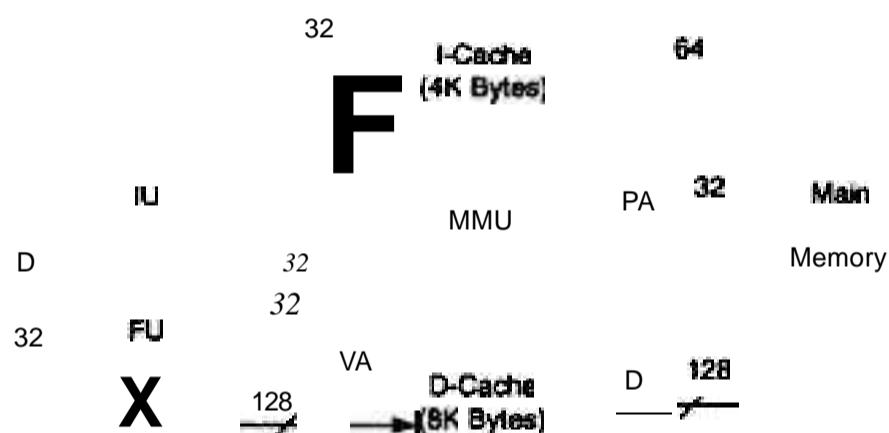


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Virtual Address Caches When a cache is indexed or tagged with a virtual address as shown in Fig. 5.9, it is called a *virtual address cache*. In this model, both cache and **MMU** translation or validation are done in parallel. The physical address generated by the MMU can be saved in tags for later write-back but is not used during the cache lookup operations. The virtual address cache is motivated with its enhanced efficiency to access the cache faster, overlapping with the MMU translation as exemplified below.



(a) A unified cache accessed by virtual address



(b) A split cache accessed by virtual address as in the Intel i860 processor

Figure 5.9 **Virtual address** models for **unified** and split caches. (Courtesy of Intel Corporation, 1989)

Example 5.3 The virtual addressed split cache design in Intel i860

Figure 5.9b shows the virtual address design in the Intel i860 using split caches for data and instructions. Instructions are 32 bits wide. Virtual addresses generated by the **integer** unit (IU) **are** 32 bits wide, and so are the physical addresses generated by the MMU. The data cache is 8 Kbytes with a block size of 32 bytes. A **two-way, set-associative** cache organization (Section 5.2.3) is implemented with 128 sets in the D-cache and 64 sets in the I-cache.

The Aliasing Problem The major problem associated with a virtual address cache is *aliasing*, when different logically addressed data have the same index/tag in the cache. Multiple processes may use the same range of virtual addresses. This aliasing problem may create confusion if two or more processes access the same physical cache location.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

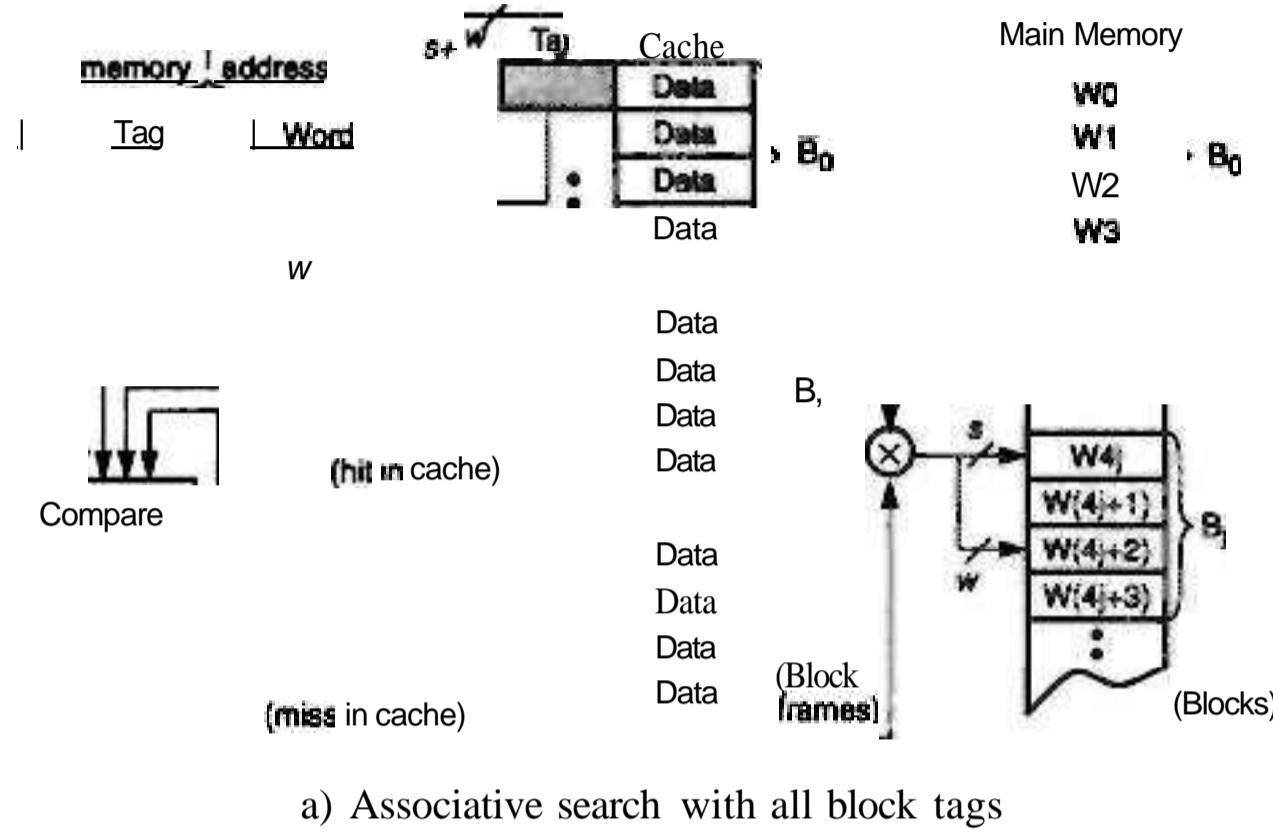


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

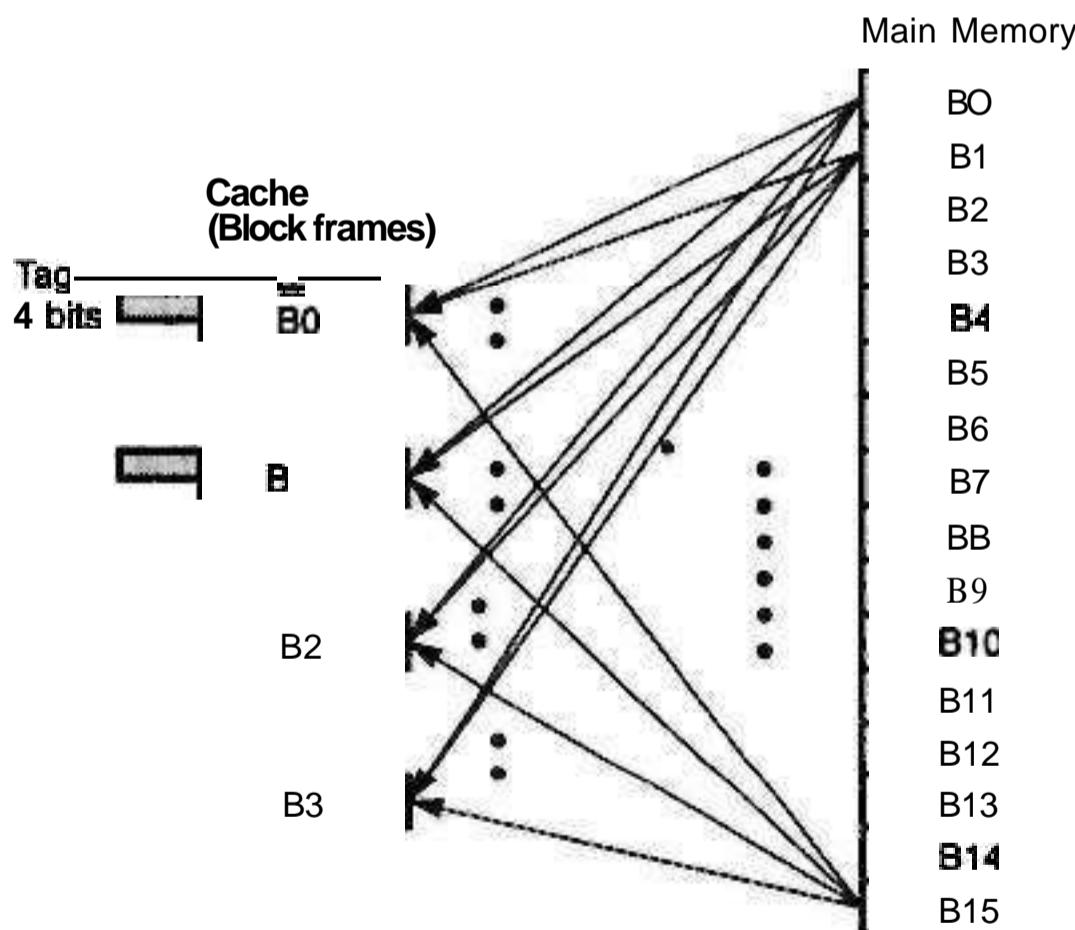


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

offers the greatest flexibility in implementing block replacement policies **for a higher hit ratio**.



a) Associative search with all block tags



b) Every block is mapped to any of the four block frames identified by the tag

Figure 5.11 Fully associative cache organization and a mapping example.

The **m -way** comparison of all tags is very **time-consuming** if the tags are compared



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

5.3.1 Interleaved Memory Organization

Various organizations of the physical **memory** are studied in this section. In order to close up the speed gap between the CPU/cache and main memory built with RAM modules, an *interleaving* technique is presented below which allows pipelined access of the parallel memory modules.

The memory design goal is to broaden the **effective memory bandwidth** so that more memory words can be accessed per unit time. The ultimate purpose is to match the memory bandwidth with the bus bandwidth and with the processor **bandwidth**.

Memory Interleaving The main memory is built with multiple modules. These memory modules are connected to a system bus or a switching network to which other resources such as processors or I/O devices are also connected.

Once presented with a memory address, each memory module returns with one word per cycle. It is possible to present different addresses to different memory modules so that parallel access of multiple words can be done simultaneously or in a pipelined fashion. Both parallel access and pipelined access are forms of parallelism practiced in a parallel memory organization.

Consider a main memory formed with $m = 2^a$ memory modules, each containing $w = 2^b$ words of memory cells. The total memory capacity is $m \cdot w = 2^{a+b}$ words. These memory words are assigned linear addresses. Different ways of assigning linear addresses result in different memory organizations.

Besides random access, the main memory is often block-accessed at consecutive addresses. Block access is needed for fetching a sequence of instructions or for accessing a linearly ordered data structure. Each block access may correspond to the size of a cache block (cache line) or to that of several cache blocks. Therefore, it is desirable to design the memory to facilitate block access of contiguous words.

Figure 5.15a shows two address formats for memory interleaving. *Low-order interleaving* spreads contiguous memory locations across the m modules horizontally (Fig. 5.15a). This implies that the low-order a bits of the memory address are used to identify the memory module. The high-order b bits are the word addresses (displacement) within each module. Note that the same word address is applied to all memory modules simultaneously. A module address decoder is used to distribute module addresses.

High-order interleaving (Fig. 5.15b) uses the high-order a bits as the module address and the low-order b bits as the word address within each module. Contiguous memory locations are thus assigned to the same memory module. In each memory cycle, only one word is accessed from each module. Thus the high-order interleaving cannot support block access of contiguous locations.

On the other hand, the low-order m -way interleaving does support block access in a pipelined fashion. Unless otherwise specified, we consider only low-order memory interleaving in subsequent discussions.

Pipelined Memory Access Access of the m memory modules can be overlapped in a pipelined fashion. For this purpose, the memory cycle (called the *major cycle*) is



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

5.3.2 Bandwidth and Fault Tolerance

Hellerman (1967) has **derived** an equation to estimate the effective increase in memory bandwidth through **multiway** interleaving. A single memory module is assumed to deliver one word per memory cycle and thus has a bandwidth of 1.

Memory Bandwidth The *memory bandwidth* B of an **m -way** interleaved memory is upper-bounded by **m** and lower-bounded by **1**. The Hellerman estimate of B is

$$B = m^{0.56} \sim \sqrt{m} \quad (5.5)$$

where **m** is the number of interleaved memory modules. This equation implies **that if** 16 memory modules are used, then the effective memory bandwidth is approximately four times that of a single module.

This pessimistic estimate is due to the fact that block access of various lengths and access of single words are randomly mixed in user programs. Hellerman's estimate was based on a **single-processor** system. If memory-access conflicts from multiple processors (such as the hot spot problem) are considered, the effective memory bandwidth will be further reduced.

In a vector processing computer, the access time of a long vector with **n** elements and stride distance 1 has been estimated by Cragon (1992) as follows: It is assumed that the n elements are stored in contiguous memory locations in an m -way interleaved memory system. The average time t_1 required to access one element in a vector is estimated by

$$t_1 = \frac{\theta}{m} \left(1 + \frac{m-1}{n}\right) \quad (5.6)$$

When $n \rightarrow \infty$ (very **long** vector), $t_1 \rightarrow \theta/m = r$ as derived in Eq. 5.4. As $n \rightarrow 1$ (scalar access), $t_1 \rightarrow B$. Equation 5.6 conveys the message that interleaved memory appeals to pipelined access of long vectors; the longer the better.

Fault Tolerance High- and low-order interleaving can be combined to yield many different interleaved memory organizations. Sequential addresses are assigned in the high-order interleaved memory in each memory module.

This makes it easier to isolate faulty memory modules in a *memory bank* of **m** memory modules. When one module failure is detected, the remaining modules can still be used by opening a window in the address space. This fault isolation cannot be carried out in a low-order interleaved memory, in which a module failure may paralyze the entire memory bank. Thus low-order interleaving memory is not fault-tolerant.

Example 5.7 Memory banks, fault tolerance, and bandwidth tradeoffs

In Fig. 5.17, two alternative memory addressing schemes are shown which combines the high- and low-order interleaving concepts. These alternatives offer a better bandwidth in case of module failure. A four-way low-order interleaving is organized in each of two memory banks in Fig. 5.17a.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The *event ordering* can be used to declare whether a memory event is legal or illegal, when several processes are accessing a common set of memory locations. A *program order* is the order by which memory accesses occur for the execution of a single process, provided that no program **reordering** has taken place. Dubois et al. (1986) have defined three primitive memory operations for the purpose of specifying memory **consistency** models:

- (1) A *load* by processor P_i is considered *performed* with respect to processor P_k at a point of time when the issuing of a *store* to the same location by P_k cannot affect the value returned by the *load*.
- (2) A *store* by P_i is considered *performed* with respect to P_k at one time when an issued *load* to the same address by P_k returns the value by this *store*.
- (3) A *load* is *globally performed* if it is performed with respect to all processors and if the *store* that is the source of the returned value has been performed with respect to all processors.

As illustrated in Fig. 5.19a, a processor can execute instructions out of program order using a compiler to resequence instructions in order to boost performance. A uniprocessor system allows these out-of-sequence executions provided that hardware interlock mechanisms exist to check data and control dependences between instructions.

When a processor in a multiprocessor system executes a concurrent program as illustrated in Fig. 5.19b, local dependence checking is necessary but may not be sufficient to preserve the intended outcome of a concurrent execution.

Maintaining the correctness and predictability of the execution results is rather complex on an **MIMD** system for the following reasons:

- (a) The order in which instructions belonging to different streams are executed is not fixed in a parallel program. If no synchronization among the instruction streams exists, then a large number of different instruction interleavings is possible.
- (b) If for performance reasons the order of execution of instructions belonging to the same stream is different from the program order, then an even larger number of instruction interleavings is possible.
- (c) If accesses are not atomic with multiple copies of the same data coexisting as in a **cache-based** system, then different processors can individually observe different interleavings during the same execution. In this case, the total number of possible execution instantiations of a program becomes even larger.

Example 5.9 Event ordering in a three-processor system (Dubois, Scheurich, and Briggs, 1988)

To illustrate the possible ways of interleaving concurrent program executions among multiple processors updating the same memory, we examine the **simultaneous** and asynchronous executions of three program segments on the three processors in Fig. 5.19c.

The shared variables are initially set as zeros, and we assume a **Print** statement reads both variables indivisibly during the same cycle to avoid confusion. If the



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



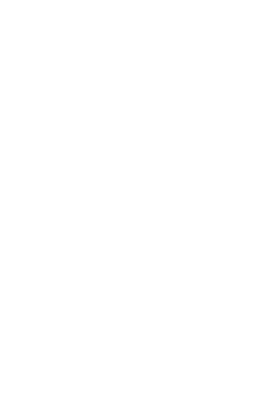
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



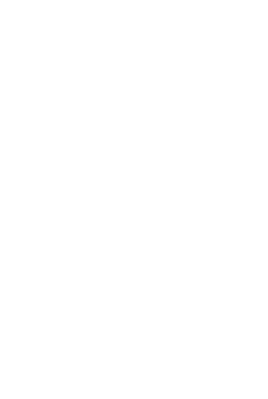
You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

Problem 5.19 Explain the following terms associated with memory management:

- (a) The role of a memory **manager** in an OS kernel.
- (b) Preemptive versus nonpreemptive memory allocation policies.
- (c) Swapping memory system and examples.
- (d) Demand paging memory system and examples.
- (e) Hybrid memory system and examples.

Problem 5.20 Compare the memory-access constraints in the following memory consistency models:

- (a) Determine the similarities and subtle differences among the conditions on sequential consistency imposed by Lamport (1979), by **Dubois et al.** (1986), and by **Sindhu et al.** (1992), respectively.
- (b) Repeat question (a) between the DSB model and the TSO model for weak consistency memory systems.
- (c) A PSO (partial store order) model for weak consistency has been refined from the TSO model. Study the PSO **specification** in the paper by Sindhu et al. (1992) and compare the relative merits between the TSO and the PSO memory models.

Chapter 6

Pipelining and Superscalar Techniques

This chapter deals with advanced pipelining and superscalar **design** in processor development. We begin with a discussion of conventional linear pipelines and analyze their performance. A generalized pipeline model is introduced to include nonlinear interstage connections. Collision-free scheduling techniques are described for performing dynamic **functions**.

Specific techniques for building instruction pipelines, arithmetic pipelines, and memory-access pipelines are presented. The discussion includes instruction prefetching, internal data forwarding, software interlocking, hardware scoreboarding, hazard avoidance, branch handling, and instruction-issuing techniques. Both static and multifunctional **arithmetic** pipelines are designed. Superpipelining and superscalar design techniques are studied along with a performance analysis.

6.1 Linear Pipeline Processors

A *linear pipeline processor* is a cascade of processing stages which are linearly connected to perform a fixed function over a stream of data flowing from one end to the other. In modern computers, linear pipelines are applied for instruction execution, arithmetic computation, and memory-access operations.

6.1.1 Asynchronous and Synchronous Models

A linear pipeline processor is constructed with k processing stages. External inputs (operands) are fed into the pipeline at the first stage S_1 . The processed results are passed from stage S_i to stage S_{i+1} , for all $i = 1, 2, \dots, k - 1$. The final result emerges from the pipeline at the last stage S_k .

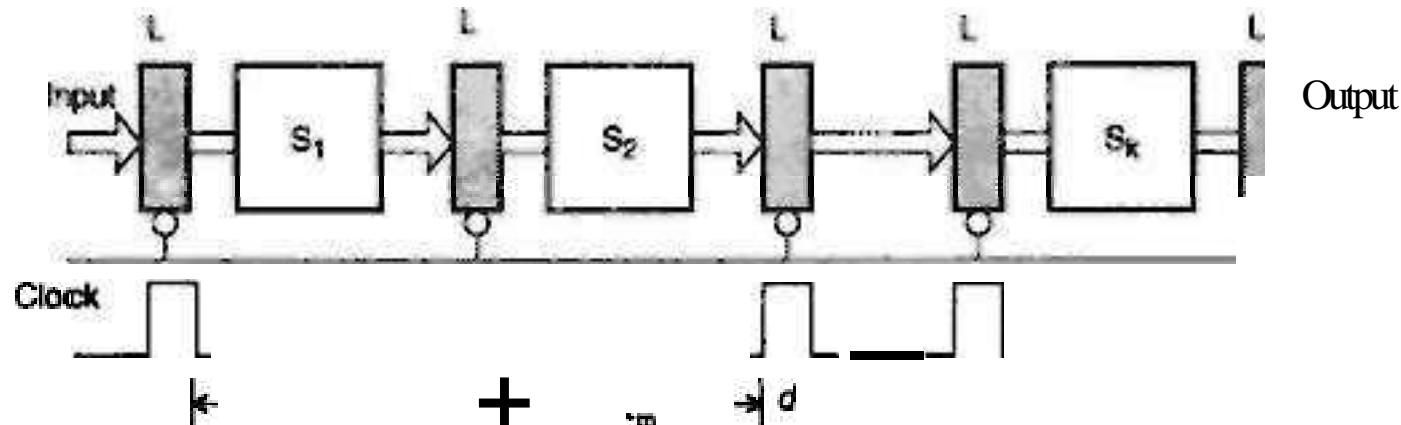
Depending on the control of data flow along the pipeline, we model linear pipelines in two categories: *asynchronous* and *synchronous*.

Asynchronous Model As shown in Fig. 6.1a, data flow between adjacent stages in an asynchronous pipeline is controlled by a handshaking protocol. When stage S_i is ready to transmit, it sends a *ready* signal to stage S_{i+1} . After stage S_{i+1} receives the incoming data, it returns an *acknowledge* signal to S_i .

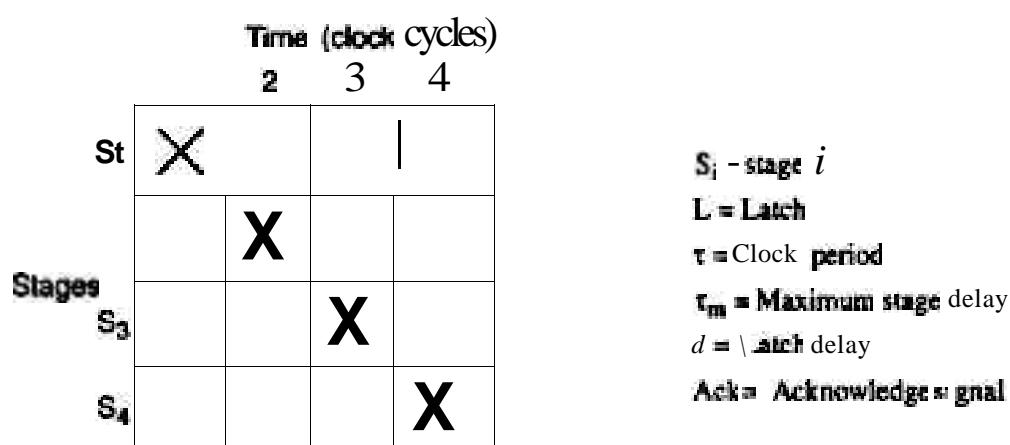
Asynchronous pipelines are useful in designing communication channels in message-passing multicomputers where pipelined wormhole routing is practiced (see Chapter 9). Asynchronous pipelines may have a variable throughput rate. Different amounts of delay may be experienced in different stages.



(a) An asynchronous pipeline model



(b) A synchronous pipeline model



(c) Reservation table of a four-stage linear pipeline

Figure 6.1 Two models of linear pipeline units and the corresponding reservation table.

Synchronous Model Synchronous pipelines are illustrated in Fig. 6.1b. Clocked latches are used to interface between stages. The latches are made with master-slave flip-flops, which can isolate inputs from outputs. Upon the arrival of a clock pulse, all

latches transfer data to the next stage simultaneously.

The pipeline stages are combinational logic circuits. It is desired to have approximately equal delays in all stages. These delays determine the clock period and thus the speed of the pipeline. Unless otherwise specified, only synchronous pipelines are studied in this book.

The utilization pattern of successive stages in a synchronous pipeline is specified by a *reservation table*. For a linear pipeline, the utilization follows the diagonal streamline pattern shown in Fig. 6.1c. This table is essentially a **space-time** diagram depicting the precedence relationship in using the pipeline stages. For a k -stage linear pipeline, k clock cycles are needed to flow through the pipeline.

Successive tasks or operations are **initiated** one per cycle to enter the pipeline. Once the pipeline is **filled** up, one result emerges from the pipeline for each additional cycle. This throughput is sustained only if the **successive** tasks are independent of each other.

6.1.2 Clocking and Timing Control

The *clock cycle* T of a **pipeline** is determined below. Let τ_i be the time delay of the circuitry in stage S_i and d the time delay of a latch, as shown in Fig. 6.1b.

Clock Cycle and Throughput Denote the *maximum stage delay* as τ_m , and we can write T as

$$T = \max_i \{\tau_i\} + d = \tau_m + d \quad (6.1)$$

At the rising edge of the clock pulse, the data is latched to the master **flip-flops** of each latch register. The clock pulse has a width equal to d . In general, $\tau_m \gg d$ for one to two orders of magnitude. This implies that the maximum stage delay τ_m dominates the clock period.

The *pipeline frequency* is defined as the inverse of the clock period:

$$f = \frac{1}{T} \quad (82)$$

If one result is expected to come out of the pipeline per cycle, f represents the *maximum throughput* of the pipeline. Depending on the initiation rate of successive tasks entering the pipeline, the *actual throughput* of the pipeline may be lower than f . This is because more than one clock cycle has elapsed between successive task initiations.

Clock Skewing Ideally, we expect the clock pulses to arrive at all stages (latches) at the same time. However, due to a problem known as *clock skewing*, the same clock pulse may arrive at different stages with a time offset of s . Let t_{\max} be the time delay of the longest logic path within a stage and t_{\min} that of the shortest logic path within a stage.

To avoid a race in two successive stages, we must choose $\tau_m > t_{\max} + s$ and $d < t_{\min} - s$. These constraints translate into the following bounds on the clock period when clock skew takes effect:

$$d + t_{\max} + s \leq T \leq \tau_m + t_{\min} - s \quad (6.3)$$

In the ideal case $\delta = 0$, $t_{max} = \tau_m$, and $t_{min} = d$. Thus, we have $r = \tau_m + d$, consistent with the definition in Eq. 6.1 without the effect of clock skewing.

6.1.3 Speedup, Efficiency, and Throughput

Ideally, a linear pipeline of k stages can process n tasks in $k + (n - 1)$ clock cycles, where k cycles are needed to complete the execution of the very first task and the remaining $n - 1$ tasks require $n - 1$ cycles. Thus the total time required is

$$T_k = [k + (n - 1)]r \quad (6.4)$$

where r is the clock period. Consider an equivalent-function nonpipelined processor which has a *flow-through delay* of kr . The amount of time it takes to execute n tasks on this nonpipelined processor is $T_1 = nkr$.

Speedup Factor The speedup factor of a k -stage pipeline over an equivalent non-pipelined processor is defined as

$$S_k = \frac{T_1}{T_k} = \frac{nkr}{kr + (n - 1)r} = \frac{nk}{k + (n - 1)}$$

Example 6.1 Pipeline speedup versus stream length

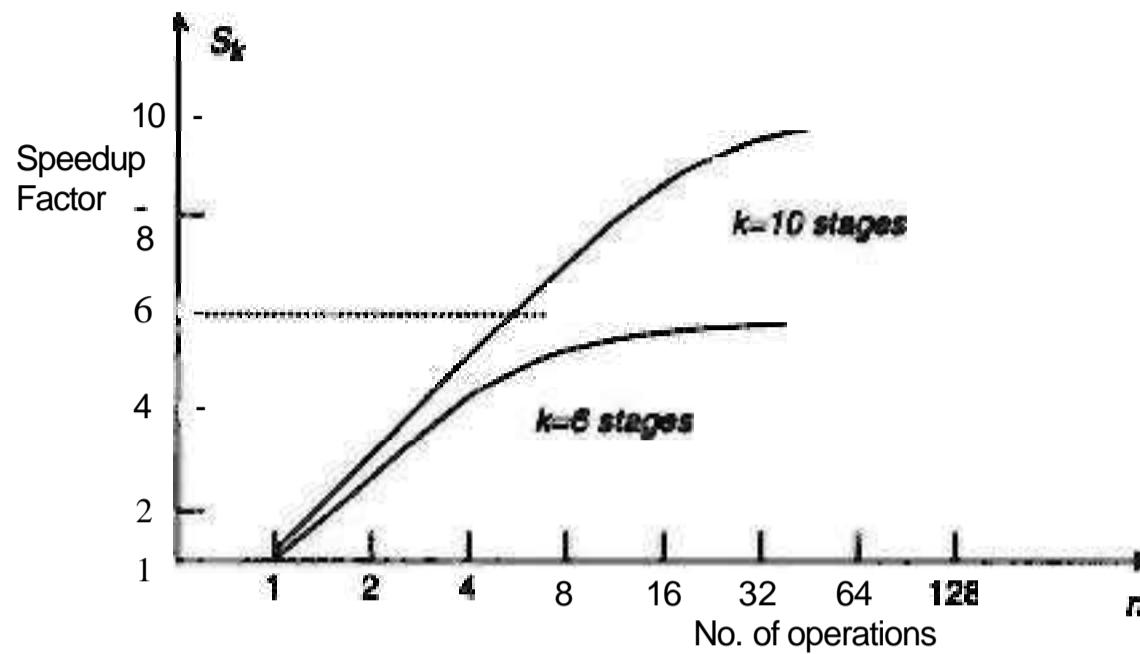
The **maximum** speedup is $S_k \rightarrow k$ as $n \rightarrow \infty$. This maximum speedup is very difficult to achieve because of data dependences between successive tasks (instructions), program **branches**, interrupts, and other factors to be studied in subsequent sections.

Figure 6.2a plots the speedup factor as a function of n , the number of tasks (operations or instructions) performed by the pipeline. For small values of n , the speedup can be very poor. The smallest value of S_k is 1 when $n = 1$.

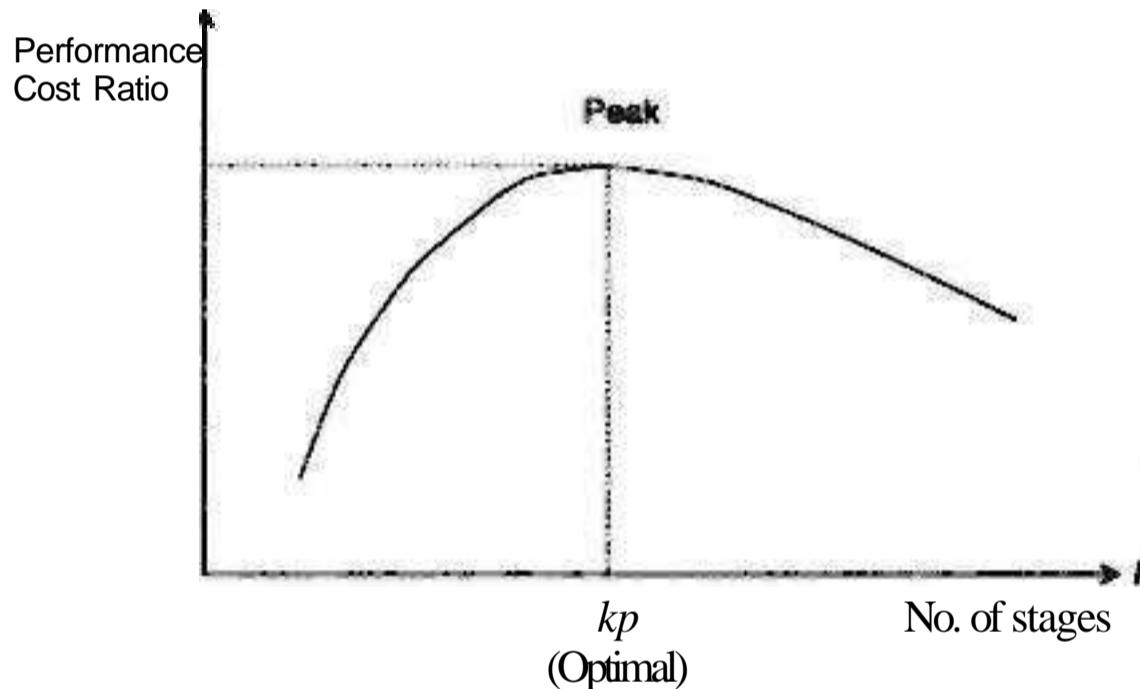
The larger the **number** A of subdivided pipeline stages, the higher the potential speedup performance. When $n = 64$, an eight-stage pipeline has a speedup value of 7.1 and a four-stage pipeline has a speedup of 3.7. However, the number of pipeline stages cannot increase indefinitely due to practical constraints on costs, control complexity, circuit implementation, and packaging limitations. Furthermore, the stream length n also affects the speedup; the longer the better in using a pipeline.

Optimal Number of Stages The finest level of pipelining is called **micropipelining**, with a subdivision of pipeline stages at the logic gate level. In practice, most pipelining is staged at the functional level with $2 < k < 15$. Very few pipelines are designed to exceed 10 stages in real computers.

On the other hand, the coarse level for pipeline stages can be conducted at the processor level, called **macropipelining**. The optimal choice of the number of pipeline stages should be able to maximize a performance/cost ratio.



(a) Speedup factor as a function of the number of operations (Eq. 6.5)



(b) Optimal number of pipeline stages (Eqs. 6.6 and 6.7)

Figure 6.2 Speedup factors and the optimal number of pipeline stages for a linear pipeline unit.

Let t be the total time required for a nonpipelined sequential program of a given function. To execute the same program on a **k-stage** pipeline with an equal flow-through delay t , one needs a clock period of $p = t/k + d$, where d is the latch delay. **Thus, the** pipeline has a maximum throughput of $\lambda = 1/p = 1/(t/k + d)$. The total pipeline cost is roughly estimated by $c + kh$, where c covers the cost of all logic stages and h represents the cost of each latch. A pipeline *performance/cost ratio* (PCR) has been **defined** by Larson (1973):

$$\text{PCR} = \frac{f}{c + fe/i(t/k + d)(c + kh)} \quad (6.6)$$

Figure 6.2b plots the PCR as a function of fc . The peak of the PCR curve corre-

sponds to an optimal choice for the number of desired pipeline stages:

$$f_{co} = \sqrt{\frac{t + h}{c + h}} \quad (6.7)$$

where t is the total flow-through delay of the pipeline. The total stage cost c , the latch delay a , and the latch cost h can be adjusted to achieve the optimal value k_0 .

Efficiency and Throughput The *efficiency* E_k of a linear k -stage pipeline is defined as

$$E_k = \frac{k}{k + (n - 1)} \quad (6.8)$$

Obviously, the efficiency approaches 1 when $n \rightarrow \infty$, and a lower bound on E_k is X/k when $n = 1$. The *pipeline throughput* H_k is defined as the number of tasks (operations) performed per unit time:

$$H_k = \frac{n}{k + (n - 1)} \quad (6.9)$$

The *maximum throughput* f occurs when $E_k \rightarrow 1$ as $n \rightarrow \infty$. This coincides with the speedup definition given in Chapter 3. Note that $H_k = E_k \cdot f = E_k / \tau = S_k / k\tau$.

6.2 Nonlinear Pipeline Processors

A *dynamic pipeline* can be reconfigured to perform variable functions at different times. The traditional linear pipelines are static pipelines because they are used to perform fixed functions.

A dynamic pipeline allows feedforward and feedback connections in addition to the streamline connections. For this reason, some authors call such a structure a *nonlinear pipeline*.

6.2.1 Reservation and Latency Analysis

In a static pipeline, it is easy to partition a given function into a sequence of linearly ordered subfunctions. However, function partitioning in a dynamic pipeline becomes quite involved because the pipeline stages are interconnected with loops in addition to streamline connections.

A multifunction dynamic pipeline is shown in Fig. 6.3a. This pipeline has three stages. Besides the *streamline connections* from S_1 to S_2 and from S_2 to S_3 , there is a *feedforward connection* from S_1 to S_3 and two *feedback connections* from S_3 to S_2 and from S_3 to S_1 .

These feedforward and feedback connections make the scheduling of successive events into the pipeline a nontrivial task. With these connections, the output of the pipeline is not necessarily from the last stage. In fact, following different dataflow patterns, one can use the same pipeline to evaluate different functions.

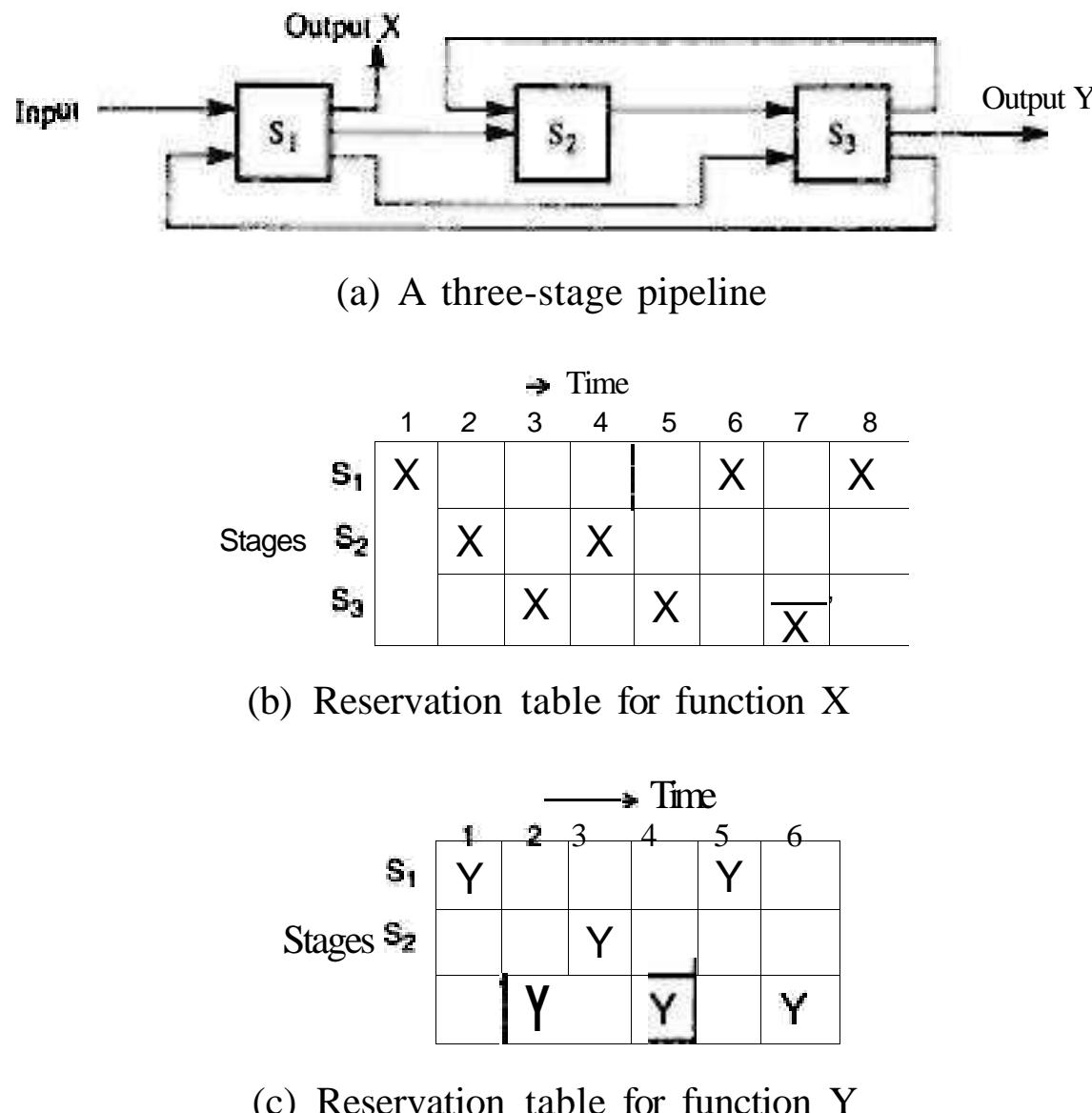


Figure 6.3 A dynamic pipeline with feedforward and feedback connections for two different functions.

Reservation Tables The reservation table for a static linear pipeline is trivial in the sense that dataflow follows a linear streamline. The *reservation table* for a dynamic pipeline becomes more interesting because a nonlinear pattern is followed. Given a pipeline configuration, multiple reservation tables can be generated for the evaluation of **different** functions.

Two reservation tables are given in Figs. 6.3b and 6.3c, corresponding to a function X and a function Y, respectively. Each function evaluation is specified by one reservation table. A static pipeline is specified by a single reservation table. A dynamic pipeline may be specified by more than one reservation table.

Each reservation table displays the time-space flow of data through the pipeline for one function evaluation. Different functions may follow different paths on the reservation table. A number of pipeline configurations may be represented by the same reservation table. There is a many-to-many mapping between various pipeline configurations and different reservation tables.

The number of columns in a reservation table is called the *evaluation time* of a given function. For example, the function X requires eight clock cycles to evaluate, and function Y requires six cycles, as shown in Figs. 6.3b and 6.3c, respectively.

A pipeline *initiation* table corresponds to each function evaluation. All initiations

to a static pipeline use the same reservation table. On the other hand, a dynamic pipeline may allow different initiations to follow a mix of reservation tables. The checkmarks in each row of the reservation table correspond to the time instants (cycles) that a particular stage will be used.

There may be multiple checkmarks in a row, which means repeated usage of the same stage in different cycles. Contiguous checkmarks in a row simply imply the extended usage of a stage **over more** than one cycle. Multiple checkmarks in a column mean that multiple stages are used in parallel during a particular clock cycle.

Latency Analysis The number of time units (clock cycles) between two initiations of a pipeline is *the latency* between them. Latency values must be nonnegative integers. A latency of k means that two initiations are separated by k clock cycles. Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a **collision**.

A collision implies resource conflicts between two initiations in the pipeline. Therefore, **all** collisions must be avoided in scheduling a sequence of pipeline initiations. Some **latencies** will cause collisions, and some **will not**. Latencies that cause collisions are called *forbidden latencies*. In using the pipeline in Fig. 6.3 to evaluate the function X , latencies 2 and 5 are forbidden, as illustrated in Fig. 6.4.

		Time →									
		1	2	3	4	5	6	7	9	10	11
Stages	S ₁	X ₁		X ₂		X ₃	X ₁	X ₄	X _{1, X₂}	X _{2, X₃}	
	S ₂		X ₁		X _{1, X₂}		X _{2, X₃}		X _{3, X₄}		X ₄
	S ₃			X ₁	1	X _{1, X₂}		X _{2, X₃, X₄}		X _{3, X₂, X₄}	

(a) Collision with scheduling latency 2

		Time →										
		1	2	3	4	5	6	7	8	9	10	11
Stages	S ₁	X ₁					X _{1, X₂}		X ₁			
	S ₂		X ₁		X ₁			X ₂		X ₂		• • •
	S ₃			X ₁		X ₁		X ₁	X ₂		X ₂	

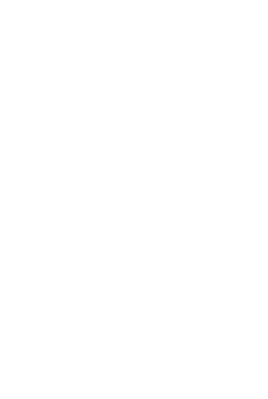
(b) Collision with scheduling latency 5

Figure 6.4 Collisions with forbidden latencies 2 and 5 in using the pipeline in Fig. 6.3 to evaluate the function X .

The **i**th initiation is denoted as X_i in Fig. 6.4. With latency **2**, initiations X_1 and X_2 collide in stage 2 at time 4. At time 7, these initiations collide in stage 3. Similarly,



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

may result in the same or different initial collision vectors(s).

This implies that even different reservation tables may produce the same state diagram. However, different reservation tables may produce different collision vectors and thus different state diagrams.



The **0's** and **1's** in the present **state**, say at time **t**, of a state diagram indicate the permissible and forbidden latencies, respectively, at time *t*. The bitwise ORing of the shifted version of the present state with the initial collision vector is meant to prevent collisions from future initiations starting at time $t + 1$ and onward.

Thus the state diagram covers all permissible state transitions that avoid collisions. All latencies equal to or greater than m are permissible. This implies that collisions can always be avoided if events are scheduled far apart (with latencies of m^+). However, such long latencies are not tolerable from the viewpoint of pipeline throughput.

Greedy Cycles From the state diagram, we can determine optimal latency cycles which result in the **MAL**. There are infinitely many latency cycles one can trace from the state diagram. For example, (1, 8), (1, 8, 6, 8), (3), (6), (3, 8), (3, 6, 3) are legitimate cycles traced from the state diagram in Fig. 6.6b. Among these cycles, only *simple cycles* are of interest.

A simple cycle is a latency cycle in which each state appears only once. In the state diagram in Fig. 6.6b, only (3), (6), (8), (1, 8), (3, 8), and (6, 8) are simple cycles. The cycle (1, 8, 6, 8) is not simple because it travels through the state (1011010) twice. Similarly, the cycle (3, 6, 3, 8, 6) is not simple because it repeats the state (1011011) three times.

Some of the simple cycles are *greedy cycles*. A greedy cycle is one whose edges are all made with minimum latencies from their respective starting states. For example, in Fig. 6.6b the cycles (1, 8) and (3) are greedy cycles. Greedy cycles in Fig. 6.6c are (1, 5) and (3). Such cycles must first be simple, and their average latencies must be lower than those of other simple cycles. The greedy cycle (1, 8) in Fig. 6.6b has an average latency of $(1 + 8)/2 = 4.5$, which is lower than that of the simple cycle (6, 8) = $(6 + 8)/2 = 7$. The greedy cycle (3) has a constant latency which equals the MAL for evaluating function X without causing a collision.

The MAL in Fig. 6.6c is 3, corresponding to either of the two greedy cycles. The **minimum-latency** edges in the state diagrams are marked with **asterisks**.

At least one of the greedy cycles will lead to the MAL. The collision-free scheduling of pipeline events is thus reduced to finding greedy cycles from the set of simple cycles. The greedy cycle yielding the MAL is the final choice.

6.2.3 Pipeline Schedule Optimization

An optimization technique based on the MAL is given below. The idea is to insert noncompute delay stages into the original pipeline. This will modify the reservation table, resulting in a new collision vector and an improved state diagram. The purpose is to yield an optimal latency cycle, which is absolutely the shortest.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

6.3 Instruction Pipeline Design

A stream of instructions can be executed by a pipeline in an overlapped manner. We describe below instruction pipelines for CISC and RISC scalar processors. Topics to be studied include instruction prefetching, data forwarding, hazard avoidance, interlocking for resolving data dependences, dynamic instruction scheduling, and branch handling **techniques for improving pipelined processor performance.**

6.3.1 Instruction Execution Phases

A typical instruction execution consists of a sequence of operations, including instruction fetch, decode, operand fetch, execute, and **write-back** phases. These phases are ideal for **overlapped** execution on a **linear pipeline**. Each phase may **require** one or more clock cycles to execute, depending on the instruction type and processor/memory architecture used.

Pipelined Instruction Processing A typical instruction pipeline is depicted in Fig. 6.9. The *fetch stage* (F) fetches instructions from a cache memory, presumably one per cycle.. The *decode stage* (D) reveals the instruction function to be performed and identifies the resources needed. Resources include general-purpose registers, buses, and functional units. The *issue stage* (I) reserves resources. Pipeline control interlocks are maintained at this stage. The operands are also read from registers during the issue stage.

The instructions are executed in one or several *execute stages* (E). Three execute stages are shown in Fig. 6.9a. The last *writeback stage* (W) is used to write results into the registers. Memory load or store operations are treated as part of execution. Figure 6.9 shows the flow of machine instructions through a typical pipeline. These eight instructions are for pipelined execution of the high-level language statements $X = Y + Z$ and $A = B \times C$. Assume *load* and *store* instructions take four execution clock cycles, while floating-point *add* and *multiply* operations take three cycles.

The above timing assumptions represent typical values used in a CISC processor. In many RISC processors, fewer clock cycles are needed. On the other hand, Cray 1 requires 11 cycles for a load and a floating-point addition takes six. With in-order instruction issuing, if an instruction is blocked from issuing due to a data or resource dependence, all instructions **following** it are blocked.

Figure 6.9b illustrates the issue of instructions following the original program order. The shaded boxes correspond to idle cycles when instruction issues are blocked due to resources latency or conflicts or due to data dependences. The first two *load* instructions issue on consecutive cycles. The *add* is dependent on both *loads* and must wait three cycles before the data (Y and Z) are loaded in.

Similarly, the *store* of the sum to memory location X must wait three cycles for the *add* to finish due to a flow dependence. There are similar blockages **during** the calculation of A . The total time required is 17 cluck cycles. This time is measured beginning at cycle 4 when the first instruction starts execution until cycle 20 the last **instruction** starts execution. This **timing** measure **eliminates** the unduly **effects** of the



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

it contains instructions sequentially ahead of the current instruction. This saves the instruction fetch time from memory. Second, it recognizes when the target of a branch falls within the loop boundary. In this case, unnecessary memory accesses can be avoided if the target instruction is already in the loop buffer. The CDC 6600 and Cray 1 have used loop buffers.

Multiple Functional Units Sometimes a certain pipeline stage becomes the bottleneck. This stage corresponds to the row with the maximum number of checkmarks in the reservation table. This bottleneck problem can be alleviated by using multiple copies of the same stage simultaneously. This leads to the use of multiple execution units in a pipelined processor design (Fig. 6.12).

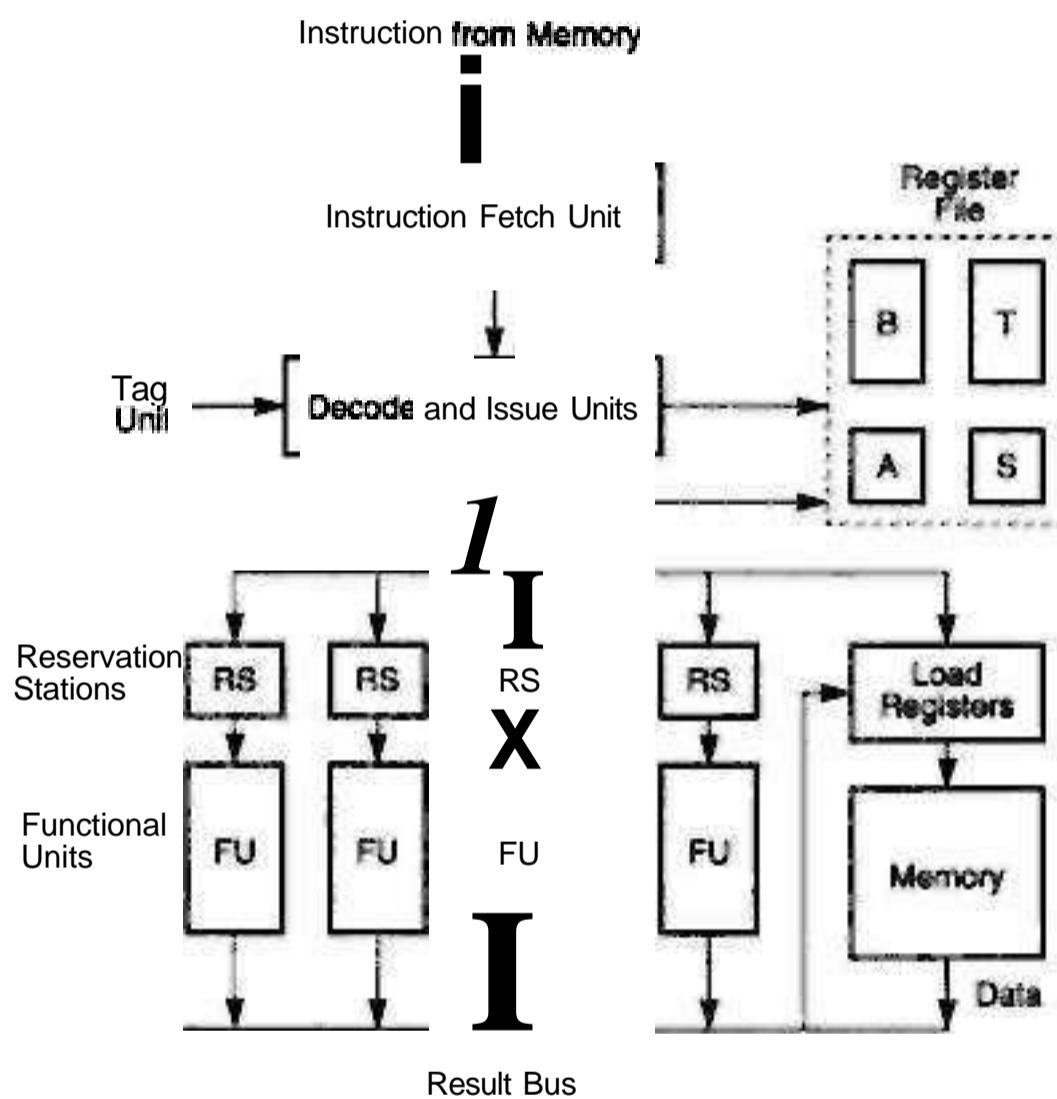
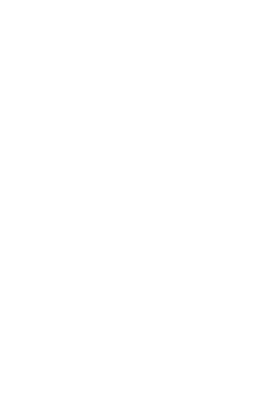


Figure 6.12 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging. (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

Sohi (1990) used a model architecture for a pipelined scalar processor containing multiple functional units (Fig. 6.12). In order to resolve data or resources dependences among the successive instructions entering the pipeline, the *reservation stations* (RS) are used with each functional unit. Operands can wait in the RS until its data dependences have been resolved. Each RS is uniquely identified by a *tag*, which is monitored by a tag unit.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

These conditions are necessary but not sufficient. This means the hazard may not appear even if one or more of the conditions exist. The RAW hazard corresponds to the flow dependence, WAR to the antidependence, and WAW to the output dependence introduced in Section 2.1. The occurrence of a logic hazard depends on the order in which the two instructions are executed. As long as the order is right, the hazard will not occur.

The resolution of hazard conditions can be checked by special hardware while instructions are being loaded into the prefetch buffer. A special *tag bit* can be used with each operand register to indicate safe or hazard-prone. Successive *read* or *write* operations are allowed to set or reset the tag bit to avoid hazards.

6.3.3 Dynamic Instruction Scheduling

In this section, we described three methods for scheduling instructions through an instruction pipeline. The static *scheduling* scheme is supported by an optimizing compiler. *Dynamic scheduling* is achieved with **Tomasulo's register-tagging** scheme built in the IBM **360/91**, or using the *scoreboarding* scheme built in the CDC 6600 processor.

Static Scheduling Data dependences in a sequence of instructions create interlocked relationships among them. Interlocking can be resolved through a compiler-based static scheduling approach. A compiler or a postprocessor can be used to increase the separation between interlocked instructions.

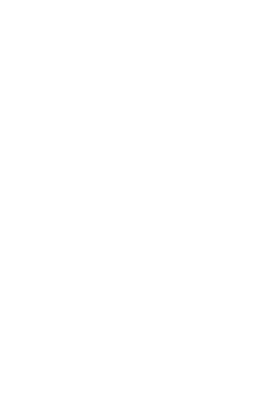
Consider the execution of the following **code fragment**. The *multiply* instruction cannot be initiated until the preceding *load* is complete. This data dependence will stall the pipeline for three clock cycles since the two *loads* overlap by one cycle.

Stage delay:	Instruction:	
2 cycles	Add	R0, R1 / $R0 \leftarrow (R0)+(R1)$ /
1 cycle	Move	R1, R5 / $R1 \leftarrow (R5)$ /
2 cycles	Load	R2, M(α) / $R2 \leftarrow (\text{Memory } (\alpha))$ /
2 cycles	Load	R3, M(β) / $R3 \leftarrow (\text{Memory } (\beta))$ /
<u>3 cycles</u>	<u>Multiply</u>	<u>R2, R3'</u> / $R2 \leftarrow (R2) \times (R3)$ /

The two *loads*, since they are independent of the *add* and *move*, can be moved ahead to increase the spacing between them and the *multiply* instruction. The following program is obtained after this modification:

Load R2, M(α)	2 to 3 cycles
Load R3, M(β)	2 cycles due to overlapping
Add R0, R1	2 cycles
Move R1, R5	1 cycle
Multiply R2, R3	3 cycles

Through this code rearrangement, the data dependences and program semantics are preserved, and the *multiply* can be initiated without delay. While the operands are being loaded from memory cells α and β into registers R2 and R3, the two instructions *add* and *move* consume three cycles and thus pipeline stalling is avoided.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

reviewed. The evaluation of **branching** strategies can be performed either on specific pipeline architecture using trace data, or by applying analytic models. We provide below a simple performance analysis. For an in-depth treatment of the subject, readers are referred to the book *Branch Strategy Taxonomy and Performance Models* by Harvey Cragon (1992).

Effect of Branching Three basic terms are introduced below for the analysis of **branching** effects: The action of fetching a nonsequential or remote instruction after a branch instruction is called *branch taken*. The instruction to be executed after a branch taken is called a *branch target*. The number of pipeline cycles wasted between a branch taken and its branch target is called the *delay slot*, denoted by b . In general, $0 < b < f_c - 1$, where f_c is the number of pipeline stages.

When a branch taken occurs, all the instructions following the branch in the pipeline become useless and will be drained from the pipeline. This implies that a branch taken causes the pipeline to be flushed, losing a number of useful cycles.

These terms are illustrated in Fig. 6.18, where a branch taken causes I_{b+1} through I_{b+k-1} to be drained from the pipeline. Let p be the probability of a conditional branch instruction in a typical instruction stream and q the probability of a successfully executed conditional branch instruction (a branch taken). Typical values of $p = 20\%$ and $q = 60\%$ have been observed in some programs.

The penalty paid by branching is equal to $pqnbr$ because each branch taken costs br extra pipeline cycles. Based on Eq. 6.4, we thus obtain the total execution time of n instructions, including the effect of branching, as follows:

$$T_{\text{eff}} = k\tau + (n-1)\tau + pqnbr$$

Modifying Eq. 6.9, we define the following *effective pipeline throughput* with the influence of branching:

$$H_{\text{eff}} = \frac{n}{T_{\text{eff}}} = \frac{n}{k\tau + (n-1)\tau + pqnbr} \quad (6.12)$$

When $n \rightarrow \infty$, the tightest upper bound on the effective pipeline throughput is obtained when $k = n = 1$:

When $p = q = 0$ (no branching), the above bound approaches the maximum throughput $\lambda = 1/\tau$, same as in Eq. 6.2. Suppose $p = 0.2$, $q = 0.6$, and $b = f_c - 1 = 7$. We define the following *performance degradation factor*:

$$D = \frac{1 - H_{\text{eff}}^*}{1 - \frac{1}{pq(k-1) + 1}} = \frac{1 - \frac{pq(k-1)}{pq(k-1) + 1}}{1 - \frac{1}{pq(7-1) + 1}} = \frac{1 - \frac{0.84}{1.84}}{1 - \frac{1}{1.84}} = 0.46 \quad (6.14)$$

The above analysis implies that the pipeline performance can be degraded by 46% with branching when the instruction stream is sufficiently long. This analysis demonstrates the degree of performance degradation caused by **branching** in an instruction pipeline.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

would be reduced significantly if the delay slot could be shortened or minimized to a zero penalty. The purpose of delayed branches is to make this possible, as illustrated in Fig. 6.20.

The idea was originally used to reduce the branching penalty in coding microinstructions. A *delayed branch* of d cycles allows at most $d - 1$ useful instructions to be executed following the branch taken. The execution of these instructions should be **independent** of the outcome of the branch instruction. **Otherwise**, a zero branching penalty cannot be achieved.

The technique is similar to that used for software interlocking. NOPs can be used as fillers if needed. The probability of moving one instruction ($d = 2$ in Fig. 6.20a) into the delay slot is greater than 0.6, that of moving two instructions ($d = 3$ in Fig. 6.20b) is about 0.2, and that of moving three instructions ($d = 4$ in Fig. 6.20c) is less than 0.1, according to some program trace results.

Example 6.8 A delayed branch with code **motion** into a **delay slot**

Code motion across branches can be used to achieve a delayed branch, as illustrated in Fig. 6.21. Consider the execution of a code fragment in Fig. 6.21a. The original program is modified by moving the useful instruction **I1** into the delay slot after the branch instruction **I3**. By so doing, instructions **I1**, **I4**, and **I5** are executed regardless of the branch outcome.

•	•
I1. Load R1,A	I2. Dec R3,1
I2. Dec R3,1	I3. BrZero R3,I5
I3. BrZero R3,I5	I1. Load R1,A
I4. Add R2,R4	I4. Add R2,R4
I5. Sub R5,R6	I5. Sub R5,R6
I6. Store R5,B	I6. Store R5,B

(a) Original program

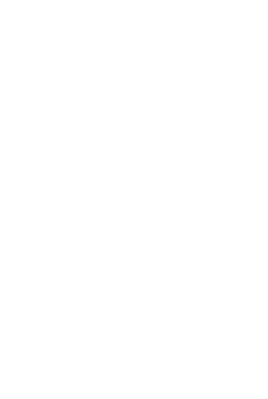
(b) Moving three useful instructions into the delay slot

Figure 6.21 Code motion across a branch to achieve a delayed branch with a **reduced** penalty to pipeline performance.

In case the branch is not **taken**, the execution of the **modified** program produces the same results as the original program. In case the branch is taken in the modified **program**, execution of the delayed instructions **I1** and **I5** is needed anyway.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

rately. The fixed-point unit is also called the integer unit. The floating-point unit can be built either as part of the central processor or on a separate coprocessor.

These arithmetic units perform scalar **operations** involving one pair of **operands** at a time. The pipelining in scalar arithmetic pipelines is controlled by software loops. Vector arithmetic units can be designed with **pipeline** hardware directly under firmware or **hardwired** control.

Scalar and vector arithmetic pipelines differ mainly in the areas of register files and control mechanisms involved. Vector hardware pipelines are often built as add-on options to a scalar processor or as an attached processor driven by a control processor. Both scalar and vector processors **are** used in **modern** supercomputers.

Arithmetic Pipeline Stages Depending on the function to be **implemented**, **different** pipeline stages in an arithmetic unit require different hardware logic. Since all arithmetic operations (such as *add*, *subtract*, *multiply*, *divide*, *squaring*, *square rooting*, *logarithm*, etc.) can be implemented with the basic add and shifting operations, the core arithmetic stages require some form of hardware to add or to shift.

For **example**, a typical three-stage floating-point adder includes a first stage for exponent comparison and equalization which is implemented with an integer adder and some shifting logic; a second stage for fraction addition using a high-speed carry **lookahead** adder; and a third stage for fraction normalization and exponent readjustment using a shifter and another addition logic.

Arithmetic or logical shifts can be easily implemented with *shift registers*. **High**-speed addition requires either the use of a *carry-propagation adder* (CPA) which adds two numbers and produces an arithmetic sum as shown in Fig. 6.22a. or the use of a *carry-save adder* (CSA) to **"add"** three input numbers and produce one sum output **and** a carry output as exemplified in Fig. 6.22b.

In a CPA, the carries generated in successive digits are allowed to propagate from the low end to the high end, using either ripple carry propagation or some carry lookahead technique.

In a CSA, the carries are not allowed to propagate but instead are saved in a carry vector. In general, an n-bit CSA is specified as follows: Let X , Y , and Z be three n-bit input numbers, expressed as $X = (x_{n-1}, x_{n-2}, \dots, x_1, x_0)$. The CSA performs bitwise operations simultaneously on all columns of digits to produce two n-bit output numbers, denoted as $S^b = (0, S_{n-1}, S_{n-2}, \dots, S_1, S_0)$ and $C = (C_n, C_{n-1}, \dots, C_1, 0)$.

Note that the leading bit of the *bitwise sum* S^b is always a 0. and the tail bit of the carry *vector* C is always a 0. The input-output relationships are expressed below:

$$\begin{aligned} S_i &= x_i \odot y_i \oplus z_i \\ C_{i+1} &= x_i y_i \vee y_i z_i \vee z_i x_i \end{aligned} \quad (6.21)$$

for $i = 0, 1, 2, \dots, n - 1$. where \oplus is the exclusive OR and \vee is the logical OR operation. Note that the arithmetic sum of three input numbers, i.e., $S = X + Y + Z$, is obtained by adding the two output numbers, i.e., $S = S^b + C$, using a CPA. We use the CPA and CSAs to implement the pipeline stages of a fixed-point multiply unit as follows.

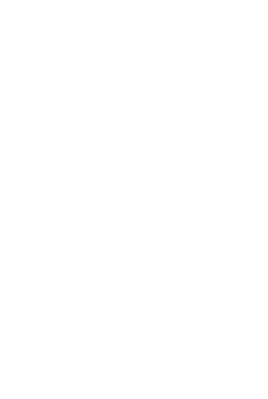
Multiply Pipeline Design Consider the multiplication of two 8-bit integers $A \times B =$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

This arithmetic pipeline has three stages. The mantissa section and exponent section are essentially two separate pipelines. The mantissa section can perform floatingpoint add or multiply operations, either single-precision (32 bits) or double-precision (64 bits).

In the mantissa section, stage 1 receives input operands and returns with computation results; 64-bit registers are used in this stage. Note that all three stages are connected to two 64-bit data buses. Stage 2 contains the array multiplier (64 x 8) which must be repeatedly used to carry out a long multiplication of the two mantissas.

The 67-bit adder performs the addition/subtraction of two mantissas, the barrel shifter is used for normalization. Stage 3 contains registers for holding results before they are loaded into the register file in stage 1 for subsequent use by other instructions.

On the exponent side, a 16-bit bus is used between stages. Stage 1 has an exponent adder for comparing the relative magnitude of two exponents. The result of stage 1 is used to equalize the exponents before mantissa addition can be performed. Therefore, a shift count (from the output of the exponent adder) is sent to the barrel shifter for mantissa alignment.

After normalization of the final result (getting rid of leading zeros), the exponent needs to be readjusted in stage 3 using another adder. The final value of the resulting exponent is fed from the register in stage 3 to the register file in stage 1, ready for subsequent usage.

Convergence Division Division can be carried out by repeated multiplications. Mantissa division is carried out by a convergence method. This convergence division obtains the quotient $Q = M/D$ of two normalized fractions $0.5 < M < D < 1$ in two's complement notation by performing two sequences of chain multiplications as follows:

$$Q = \frac{M \times R_1 \times R_2 \times \dots \times R_k}{D} \quad (6.22)$$

where the successive multipliers

$$R_i = 1 + \delta^{2^{i-1}} = 2 - D^{(i)} \quad \text{for } i = 1, 2, \dots, k \quad \text{and} \quad D = 1 - \delta$$

The purpose is to choose R_i such that the denominator $D^{(k)} = D \times R_1 \times R_2 \times \dots \times R_k \rightarrow 1$ for a sufficient number of k iterations, and then the resulting numerator $M \times R_1 \times R_2 \times \dots \times R_k \rightarrow Q$.

Note that the multiplier R_i can be obtained by finding the two's complement of the previous chain product $D^{(i)} = D \times R_1 \times \dots \times R_{i-1} = 1 - \delta^2$ because $2 - D^{(i)} = R_i$. The reason why $D^{(k)} \rightarrow 1$ for large k is that

$$\begin{aligned} D^{(i)} &= (1 - \delta)(1 + \delta)(1 + \delta^2)(1 + \delta^4) \dots (1 + \delta^{2^{i-1}}) \\ &= (1 - \delta^2)(1 + \delta^2)(1 + \delta^4) \dots (1 + \delta^{2^{i-1}}) \\ &= (1 - \delta^{2^i}) \quad \text{for } i = 1, 2, \dots, k \end{aligned} \quad (6.23)$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

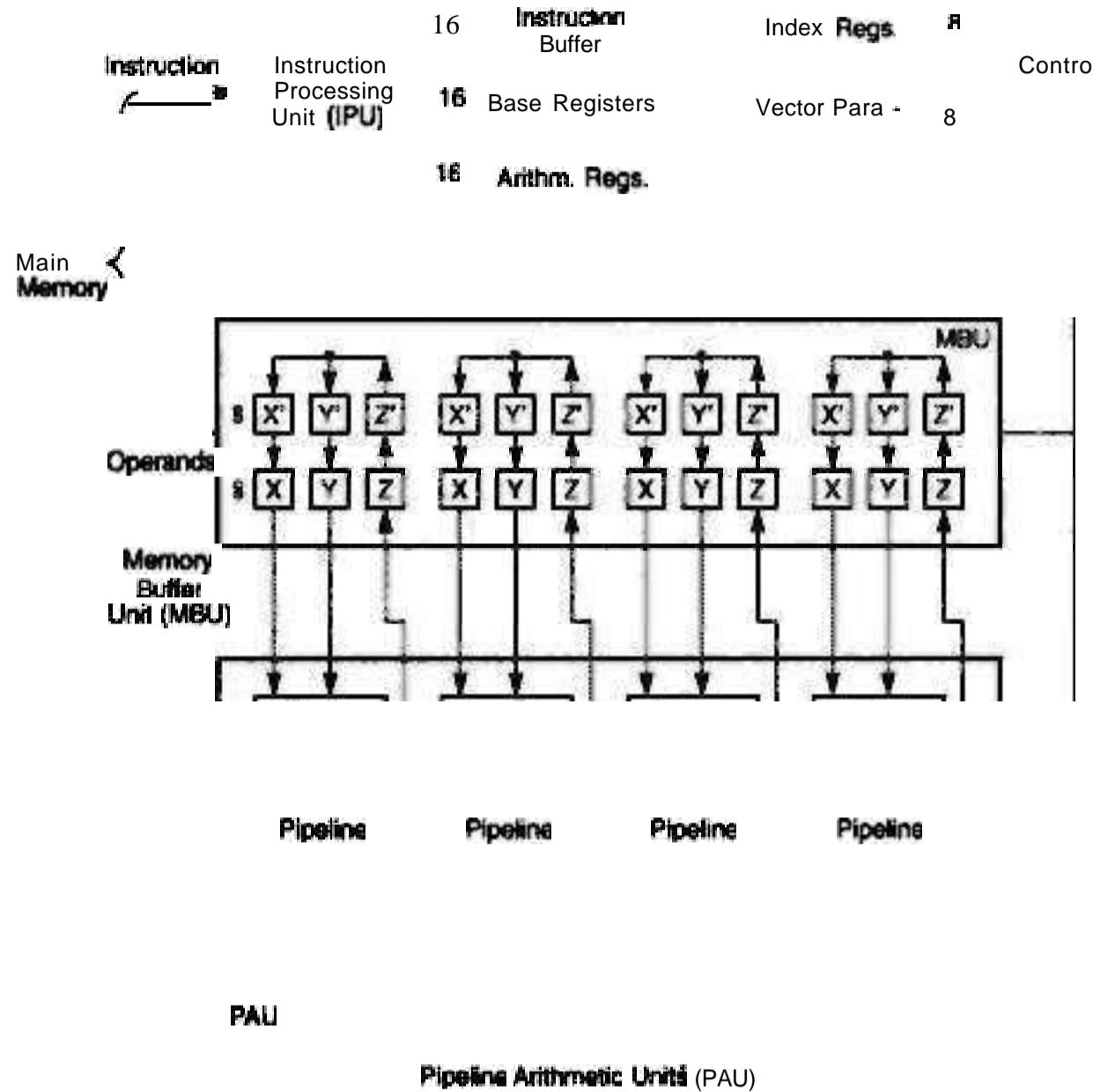


Figure 6.26 The architecture of the TI Advanced Scientific Computer (ASC) (Courtesy of Texas Instruments, Inc.)

The entire pipeline can perform the *multiply* (\times) and the *add* ($+$) in a single flow through the pipeline. The two levels of buffer registers will isolate the loading and fetching of operands to or from the PAU, respectively, as in the concept of using a pair in the prefetch buffers described in Fig. 6.11.

Even though the **TI-ASC** is no longer in production, the system provided a unique design for multifunction arithmetic pipelines. Today, most supercomputers implement arithmetic pipelines with **dedicated** functions for much simplified control circuitry.

6.5 Superscalar and Superpipeline Design

We present two architectural approaches to improving pipeline performance using the base scalar pipeline as a reference machine. There is a duality of symmetry between the two architectures. Practical design constraints are discussed in regard to



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Data Dependences Consider the example program in Fig. 6.28b. A dependence graph is drawn to indicate the relationship among the **instructions**. Because the register content in **R1** is loaded by **I1** and then used by **I2**, we have flow dependence: $I1 \rightarrow I2$.

Because the result in register R4 after executing **I4** may affect the operand register R4 used by **I3**, we have **antidependence**: $I3 \rightarrow I4$. Since both **I5** and **I6** modify the register R6 and R6 supplies an operand for **I6**, we have both flow and output dependence: $I5 \rightarrow I6$ and $I5 \rightarrow I6$ as shown in the dependence graph.

To schedule instructions **through** one or more pipelines, these data dependences must not be violated. Otherwise, erroneous' results may be produced. These data dependences are detected by a compiler and made available at pipeline scheduling time.

Pipeline Stalling This is a problem which may seriously lower pipeline utilization. Proper scheduling avoids pipeline stalling. The problem exists in both scalar and superscalar processors. However, it is more serious in a superscalar **pipeline**. Stalling can be caused by data dependences or by resource conflicts among instructions already in the pipeline or about to enter the pipeline. We use an example to illustrate the conditions causing **pipeline** stalling.

Consider the scheduling of two instruction **pipelines in a two-issue superscalar processor**. Figure 6.29a shows the case of no data dependence on the left and flow dependence ($I1 \rightarrow I2$) on the right. Without data dependence, all pipeline stages are utilized without idling.

With dependence, instruction **I2** entering the second pipeline must wait for two cycles (shaded time slots) before entering the execution stages. This delay may also pass to the next instruction **I4** entering the pipeline.

In Fig. 6.29b, we show the effect of branching (instruction **I2**). A delay slot of four cycles results from a branch taken by **I2** at cycle 5. Therefore, both pipelines must be flushed before the target instructions **I3** and **I4** can enter the pipelines from cycle 6. Here, delayed branch or other amending operations are not **taken**.

In Fig. 6.29c, we show a combined problem involving both resource conflicts and data dependence. Instructions **I1** and **I2** need to use the same functional unit, and $I2 \rightarrow I4$ exists.

The net effect is that **I2** must be scheduled one cycle behind because the two pipeline stages (e_1 and e_2) of the same functional unit must be used by **I1** and **I2** in an overlapped fashion. For the same reason, **I3** is also delayed by one cycle. Instruction **I4** is delayed for two cycles due to the flow dependence on **I2**. The shaded boxes in **all** the timing charts correspond to idle stages.

Multipipeline Scheduling Instruction issue and completion policies are critical to superscalar processor performance. Three scheduling policies are introduced below. When instructions are issued in **program order**, we call it *in-order issue*. When program order is violated, *out-of-order issue* is being practiced.

Similarly, if the instructions must be completed in program order, it is called *in-order completion*. Otherwise, *out-of-order completion* may result. In-order issue is easier to implement but may not yield the optimal performance. In-order issue may result in either in-order or out-of-order completion.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

associative with 128 sets, two blocks for each set, and 32 bytes (8 instructions) per block. The data cache resembles that of the instruction set.

The **88110** employs the **MESI** cache coherence protocol. A **write-invalidate** procedure guarantees that one processor on the bus has a modified copy of any cache block at the same time. The 88110 is implemented with 1.3 million transistors in a 299-pin package and driven by a **50-MHz** clock. Interested readers may refer to **Diefendorff and Allen (1992)** for details.

Superscalar Performance To compare the relative performance of a superscalar processor with that of a scalar base **machine**, we estimate the ideal execution time of N independent instructions through the pipeline.

The time required by the scalar base machine is

$$T(1,1) = k + N - 1 \quad (\text{base cycles}) \quad (6.26)$$

The ideal execution time required by an **m-issue** superscalar machine is

$$T(m, i) = k + \frac{N}{m} - 1 \quad (\text{base cycles}) \quad (6.27)$$

where k is the time **required** to execute the first m instructions through the m pipelines simultaneously, and the second term corresponds to the time required to execute the remaining $N - m$ instructions, T_0 per cycle, through m pipelines.

The ideal speedup of the superscalar machine over the base machine is

$$S(m, 1) = \frac{T(1, 1)}{T(m, 1)} = \frac{N + k - 1}{N/m + k - 1} = \frac{m(N + k - 1)}{N + m(k - 1)} \quad (6.28)$$

As $N \rightarrow \infty$, the speedup limit $S(m, 1) \rightarrow T_0$, as expected.

6.5.2 Superpipelined Design

In a **superpipelined** processor of degree n , the pipeline cycle time is $1/n$ of the base cycle. As a comparison, while a fixed-point add takes one cycle in the base scalar processor, the same operation takes n short cycles in a superpipelined processor implemented with the same technology.

Figure 6.31a shows the execution of instructions with a superpipelined machine of degree $n = 3$. In this case, only one instruction is issued per **cycle**, but the cycle time is **one-third** of the base cycle. Single-operation latency is n pipeline cycles, equivalent to one base cycle. The **ILP** required to fully utilize the machine is n instructions, as shown in the third column of Table 6.1.

Superpipelined machines have been around for a long time. Cray has built both the CDC 760C and the Cray 1 as superpipelined machines with a latency of $n - 3$ cycles for a fixed-point add. Superpipelining is not possible without a high-speed clocking mechanism.

Superpipeline Performance The minimum time required to execute A^* instructions



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



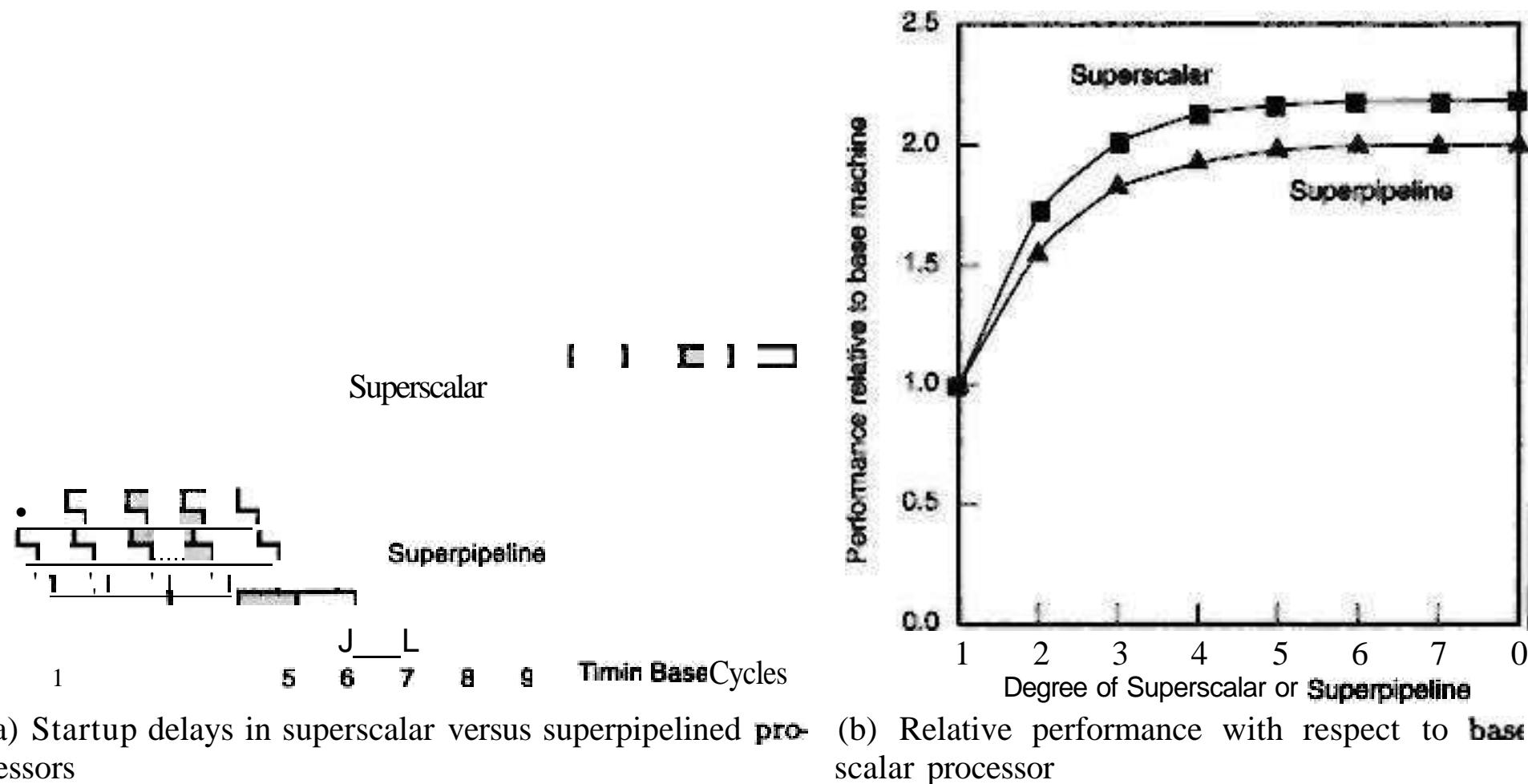
You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

6.5.3 Supersymmetry and Design Tradeoffs

Figure 6.33 compares a superscalar processor with a superpipelined processor, both of degree 3 (i.e., $m = n = 3$), issuing a basic block of six independent instructions. The issue of three instructions per base cycle is the rate for the superscalar machine, while the superpipelined machine takes only **one-third** of the base cycle to issue each instruction. In the steady state, both machines will execute the same number of instructions during the same time interval.



(a) Startup delays in superscalar versus superpipelined processors

(b) Relative performance with respect to base scalar processor

Figure 6.33 Relative performance of superscalar and superpipelined processors. (Courtesy of Jouppi and Wall; reprinted from *Proc. ASPLOS*, ACM press, 1989)

Supersymmetry The superpipelined machine has a longer startup delay and lags behind the superscalar machine at the start of the program (Fig. 6.33a). Furthermore, a branch may create more damage on the superpipelined machine than on the superscalar machine. This effect diminishes as the degree of superpipelining increases, and all issuable instructions are issued closer together.

Figure 6.33b shows the performance of superscalar and of superpipelined processors relative to the base scalar processor. There is a duality of latency and a parallel issue of instructions.

These two curves were obtained by Jouppi and Wall (1989) after simulation runs of eight benchmarks on an ideal base machine, on a superscalar machine of degree m , and on a superpipelined machine of degree n , where $2 \leq m, n \leq 8$.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

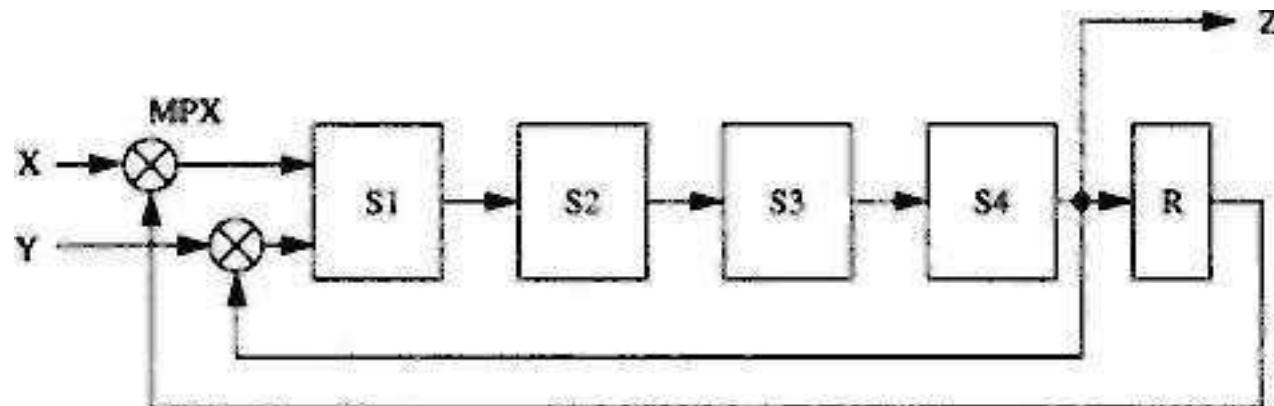


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

purpose is to yield a new reservation table leading to an optimal latency equal to the lower bound.

- Show the modified reservation table with five rows and seven columns.
- Draw the new state transition diagram for obtaining the optimal cycle.
- List all the simple cycles and greedy cycles from the state diagram.
- Prove that the new MAL equals the lower bound.
- What is the optimal throughput of this pipeline? Indicate the percentage of throughput improvement compared with that obtained in part (d) of Problem G.6.

Problem 6.8 Consider an adder pipeline with four stages as shown below. The pipeline consists of input lines X and Y and output line Z. The pipeline has a register R at its output where the temporary result can be stored and fed back to S1 at a later point in time. The inputs X and Y are multiplexed with the outputs R and Z.



- Assume the elements of the vector A are fed into the pipeline through input X , one element per cycle. What is the minimum number of clock cycles required to compute the sum of an N -element vector A : $s = \sum_{I=1}^N A(I)$? In the absence of an operand, a value of 0 is input into the pipeline by default. Neglect the setup time for the pipeline.
- Let τ be the clock period of the pipelined adder. Consider an equivalent non-pipelined adder with a flow-through delay of $4T$. Find the actual speedup $S_4(64)$ and the efficiency $\eta_4(64)$ of using the above pipeline adder for $N = 64$.
- Find the maximum speedup $S_4(\infty)$ and the efficiency $\eta_4(\infty)$ when N tends to infinity.
- Find $N_{1/2}$, the minimum vector length required to achieve half of the maximum speedup.

Problem 6.9 Consider the following pipeline reservation table.

S1	X		X
S2		X	
S3			X



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

7.1.1 Hierarchical Bus Systems

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

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

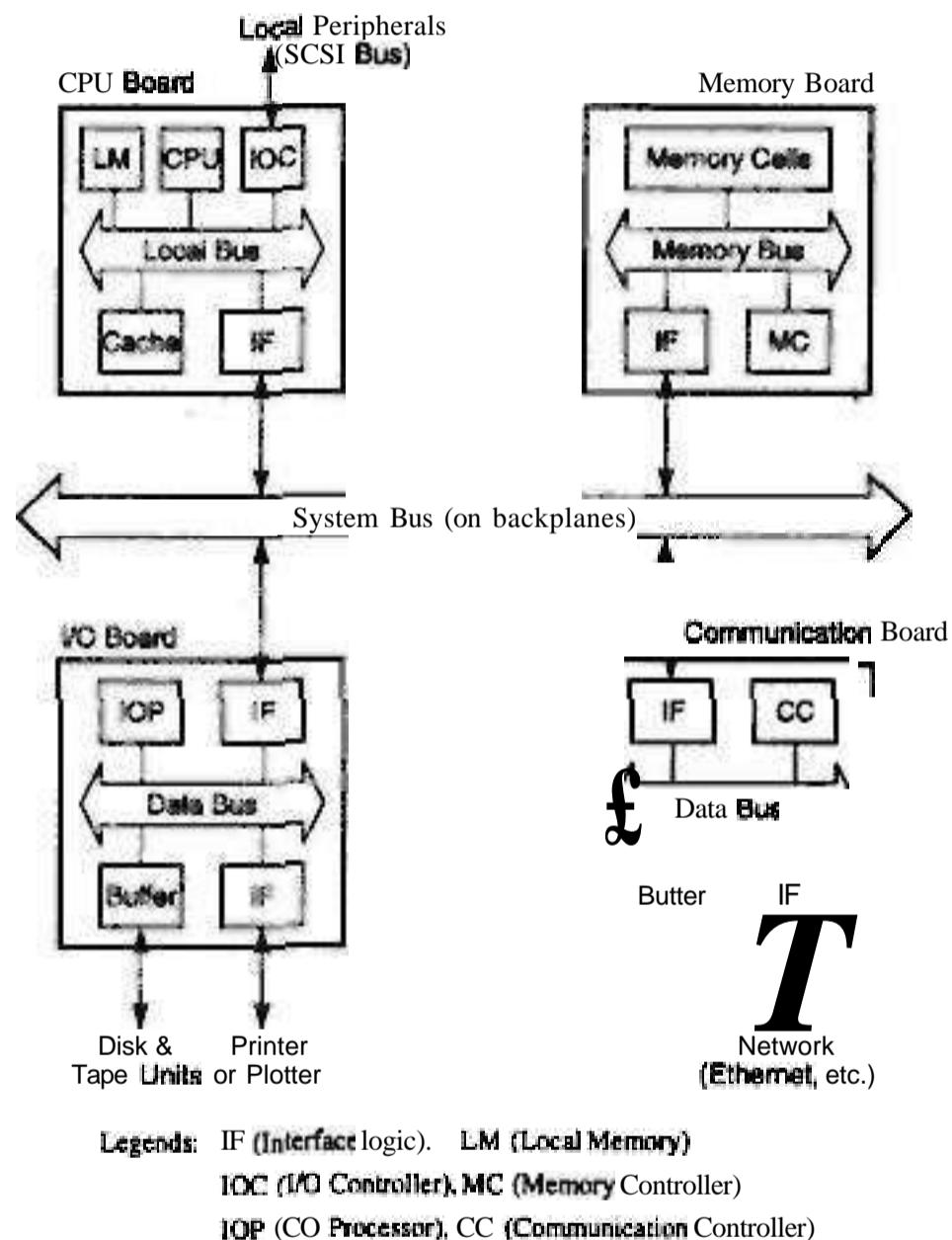


Figure 7.2 Bus systems at board level, backplane level, and I/O level.

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

An **I/O** board or network interface board uses a *data bus*. Each of these board buses consists of signal and utility **lines**. **With** the sharing of the lines by many **I/O** devices, the layout of these **lines** may be at different layers of the PC board.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

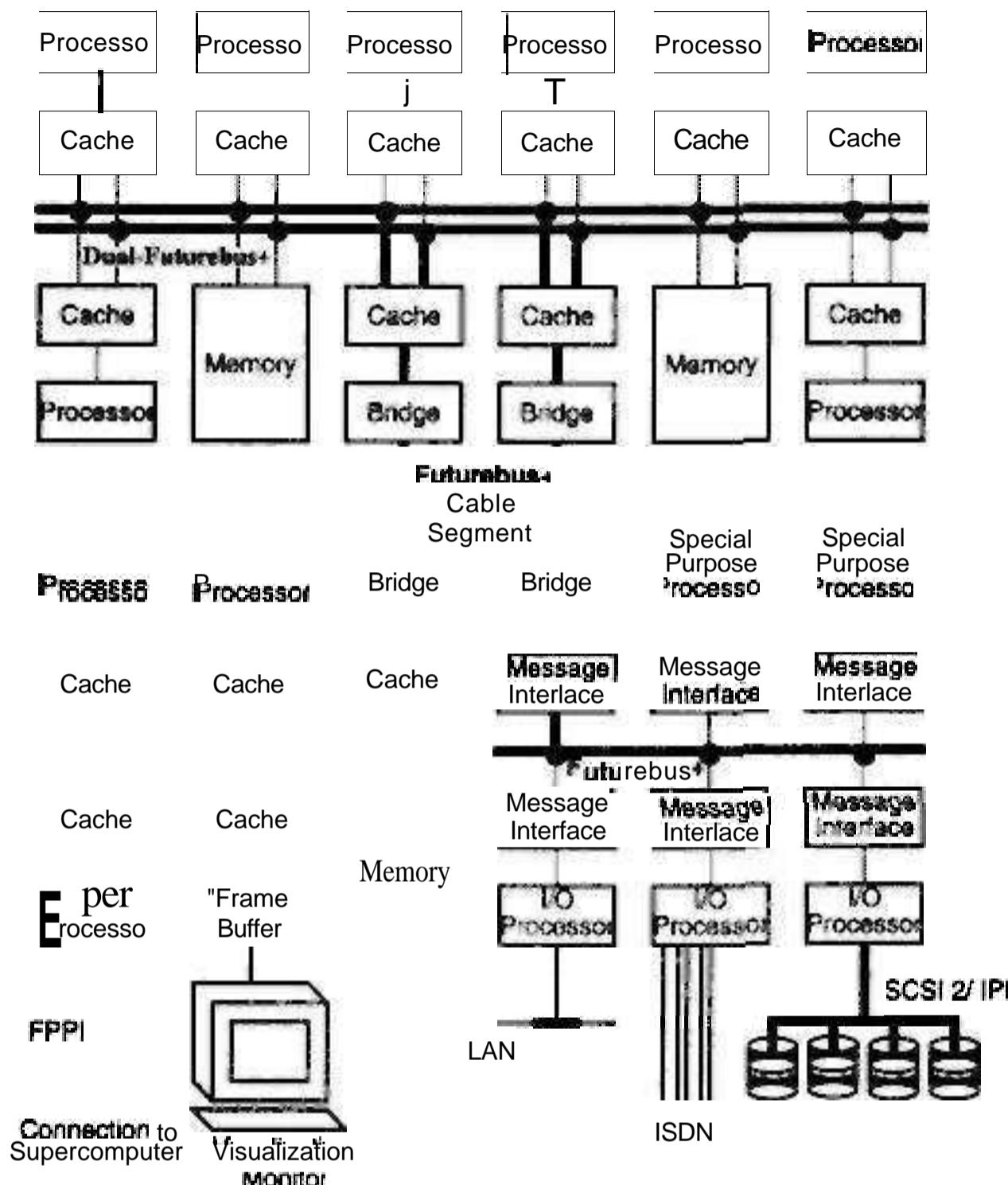


Figure 7.5 A large multiprocessor system using multiple **Futurebus+** segments.
 (Reprinted with permission from IEEE Standard 896.1-1991, copyright © 1991 by IEEE, Inc.)

the simultaneous connections of some multiple input-output pairs may result in conflicts in the use of switches or communication links.

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

A multistage network is called **nonblocking** if it can perform all possible connections between inputs and outputs by rearranging its connections. In such a network, a connection path can always be established between any input-output pair. The Benes



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

typical **mainframe** multiprocessor configuration uses $n = 4$ processors and $m = 16$ memory modules. A **multiport** memory multiprocessor is not scalable because once the ports are fixed, no more processors can be added without redesigning the memory controller.

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

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

7.1.3 Multistage and Combining Networks

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

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

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

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

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

Now consider implementing the permutation π_2 in the 8-input Omega network



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

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

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

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

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

```
Fetch&Add( $x, e$ )
  {  $temp \leftarrow x;$ 
     $x \leftarrow temp + e;$ 
  return  $temp\}$ 
```

(7-1)

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

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

One of the following operations will be performed if processor P_1 executes $Ans_1 \leftarrow \text{Fetch\&Add}(x, e_1)$ and P_2 executes $Ans_2 \leftarrow \text{Fetch\&Add}(x, e_2)$ simultaneously on the shared variable x . If the request from P_1 is executed ahead of that from P_2 , the following values are returned:

$$\begin{aligned} Ans_1 &\leftarrow x \\ Ans_2 &\leftarrow x + e_1 \end{aligned} \tag{7.2}$$

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



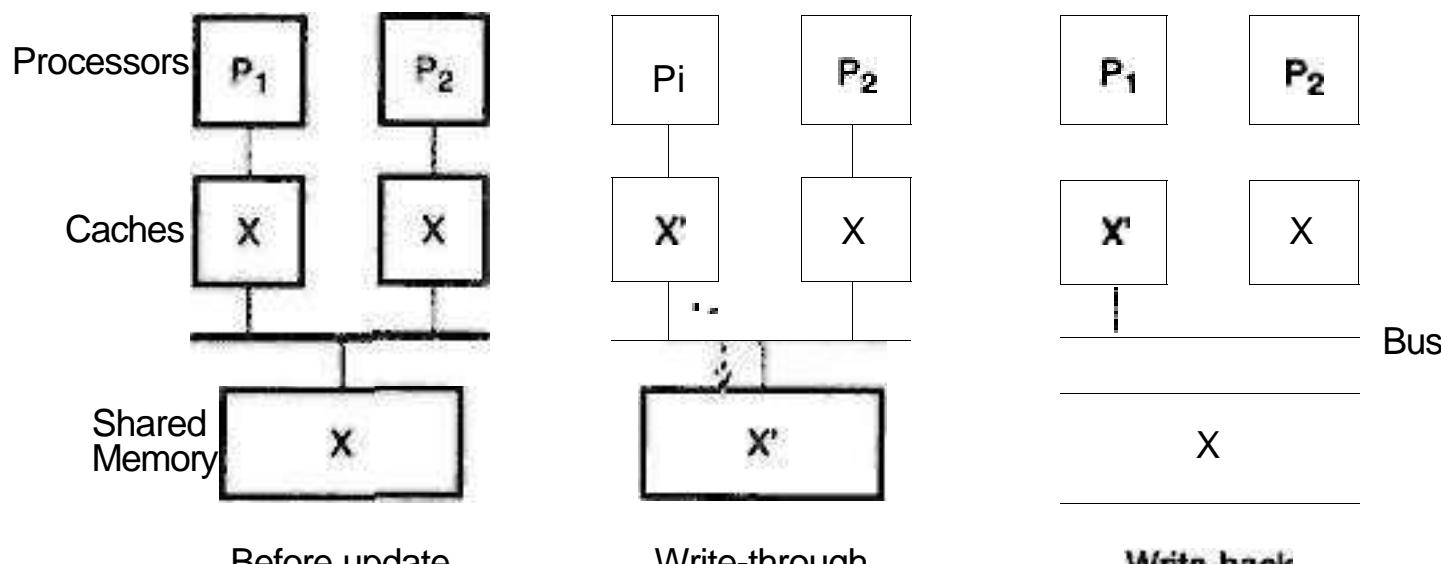
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



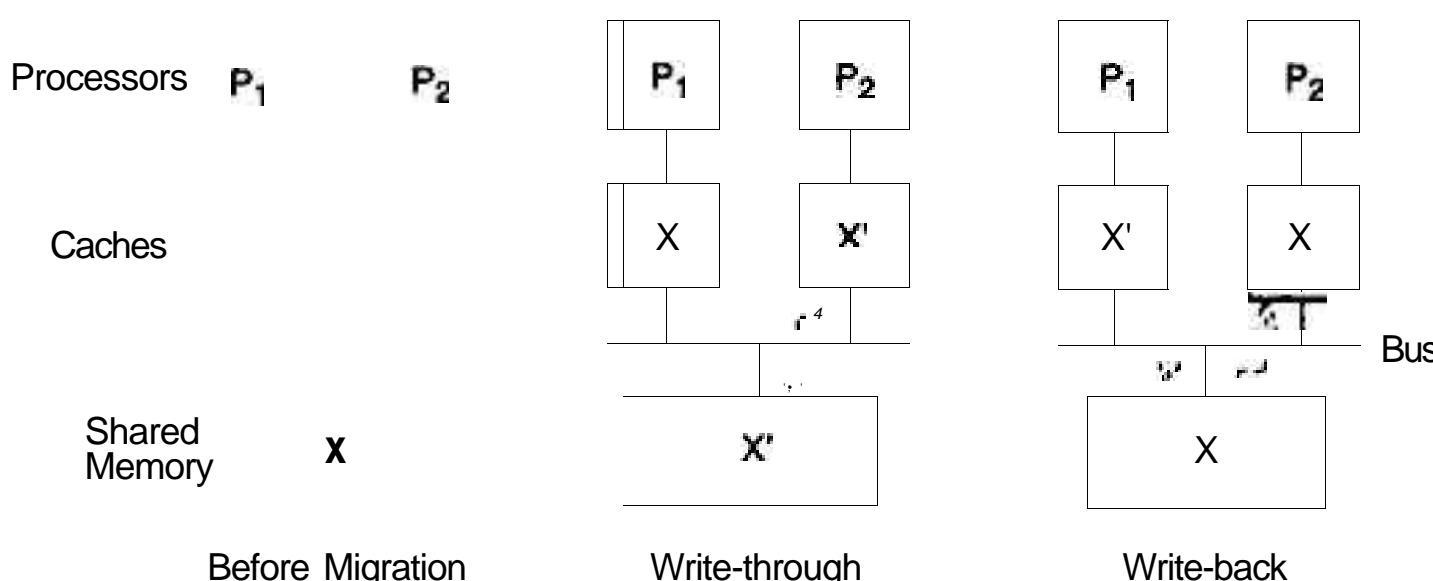
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



(a) Inconsistency in sharing of writable data



(b) Inconsistency after process migration

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

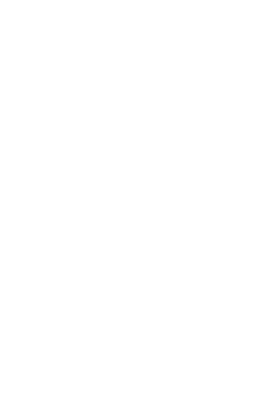
When the **I/O** processor *loads* a new data X' into the main memory, bypassing the **write-through** caches (middle diagram in Fig. 7.13a), inconsistency occurs between cache 1 and the shared memory. When **outputting** a data directly from the shared memory (bypassing the caches), the **write-back**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

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

Two Protocol Approaches Many of today's commercially available multiprocessors use bus-based memory systems. A bus is a convenient device for ensuring cache coherence because it allows all processors in the system to observe ongoing memory



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

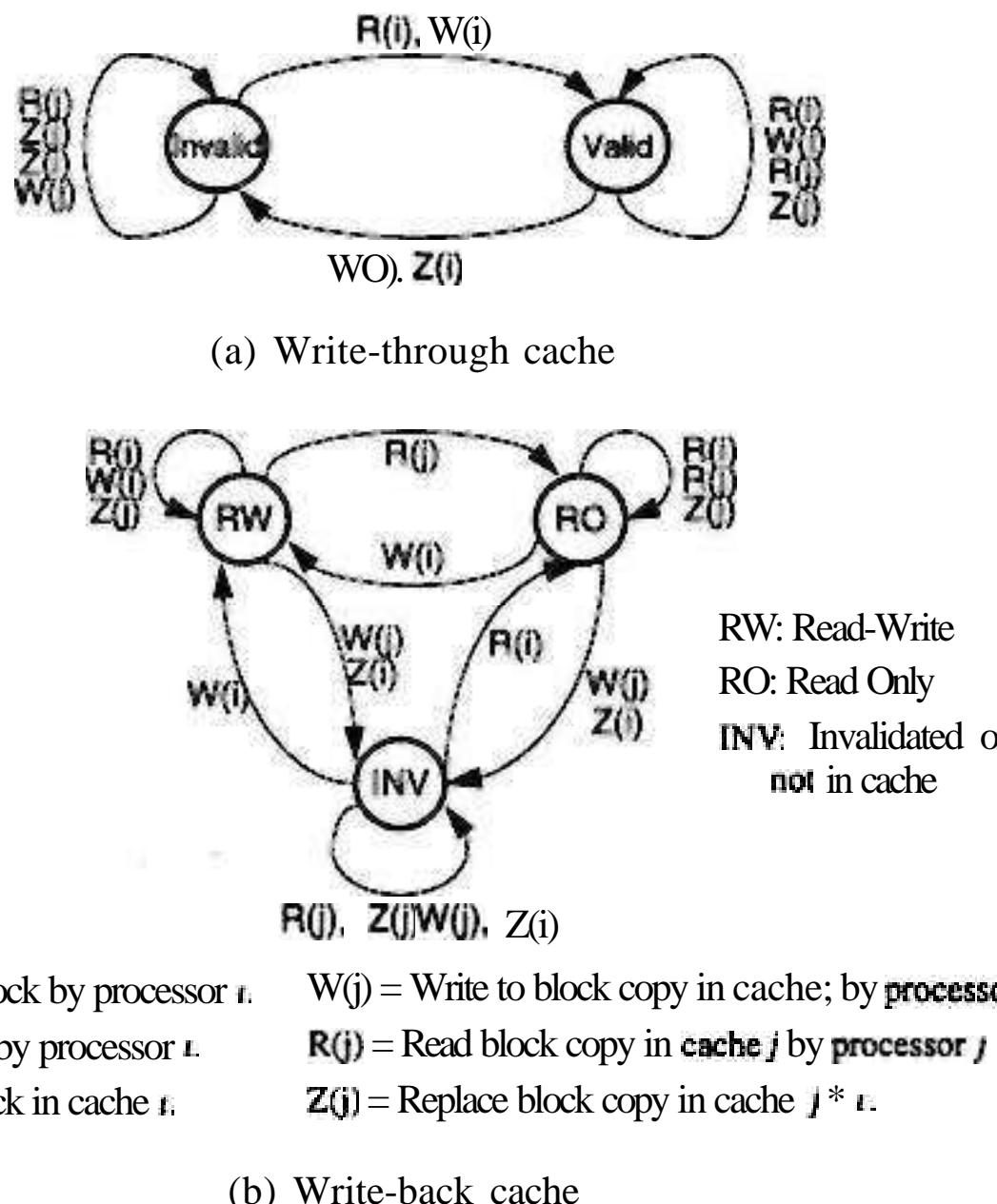


Figure 7.15 Two **state-transit ion** graphs for a cache block using **write-invalidate** snoopy protocols. (Adapted from Dubois, Scheurich, and Briggs, 1988)

transaction which is broadcast to all caches and memory. If a modified block copy exists in a remote cache, memory must first be updated, the copy invalidated, and ownership transferred to the requesting cache.

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

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

- *Valid*: The cache block, which is consistent with the memory copy, has been *read* from shared memory and has not been modified.
- *Invalid*: The block is not found in the cache or is inconsistent with the memory



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

responsibilities of some remote memory modules. A memory agent splits a transaction to assign *read* and *write* permissions to the location it is handling for the bus segment it is on. It should also pass external invalidations or interventions to the bus segment. Split transactions have been used to delay invalidation completions. Various cache coherence commands are needed on the buses.

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

An invalidation must propagate vertically up and down in order to invalidate all copies in the shared caches at level 2. Suppose processor P_1 issues a *write* request. The *write* request propagates up to the highest level and invalidates copies in C_{20} , C_{22} , C_{16} , and C_{18} , as shown by the arrows from C_{11} to all the shaded copies.

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

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

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

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

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

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

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

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

Test&Set	<i>(lock)</i>	
	<i>temp</i> \leftarrow <i>lock</i> ;	<i>lock</i> \leftarrow 1;
	return <i>temp</i>	
Reset	<i>(lock)</i>	
	<i>lock</i> \leftarrow 0	

(7.4)

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

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

Synchronization on Futurebus+ The asynchronous (**source-synchronized**) protocols used in **Futurebus+** allow the synchronization domain of the *sender* to extend along the bus segment, presenting only one synchronization interface between the *bus* and the receiver.

A centrally synchronized bus requires three separate synchronization domains (*bus*, *sender*, *receiver*). Since Futurebus+ uses only one synchronization interface, it eliminates unnecessary delays and spatial skews caused by resynchronization in other interfaces. Thus higher performance is expected using a sender-synchronized protocol for synchronization.

Concurrent processes residing in different processors can be synchronized using *barriers*. A barrier can be implemented by a shared-memory word which keeps counting the number of processes reaching the barrier. After all processes have updated the



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



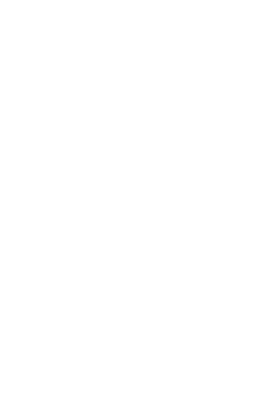
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

- (g) Blocking How **control** in wormhole **routing**.
- (h) Discard and retransmission flow control.
- (i) **Detour** flow control after being blocked.
- (j) Virtual networks and subnetworks.

Problem 7.12

- (a) Draw a **16-input** Omega network using **2×2** switches as building blocks.
- (b) Show the switch settings for routing a message from node 1011 to node 0101 and from node **0111** to node 1001 simultaneously. Does blocking exist in this case?
- (c) Determine how many permutations can be implemented in one pass through this Omega network. What is the percentage of **one-pass** permutations among all permutations?
- (d) What is the maximum number of passes needed to implement any permutation through the network?

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

- (a) **Unicast** versus multicast
- (b) Broadcast versus conference
- (c) Channel traffic or network traffic
- (d) Network communication latency
- (e) Network partitioning for multicasting communications

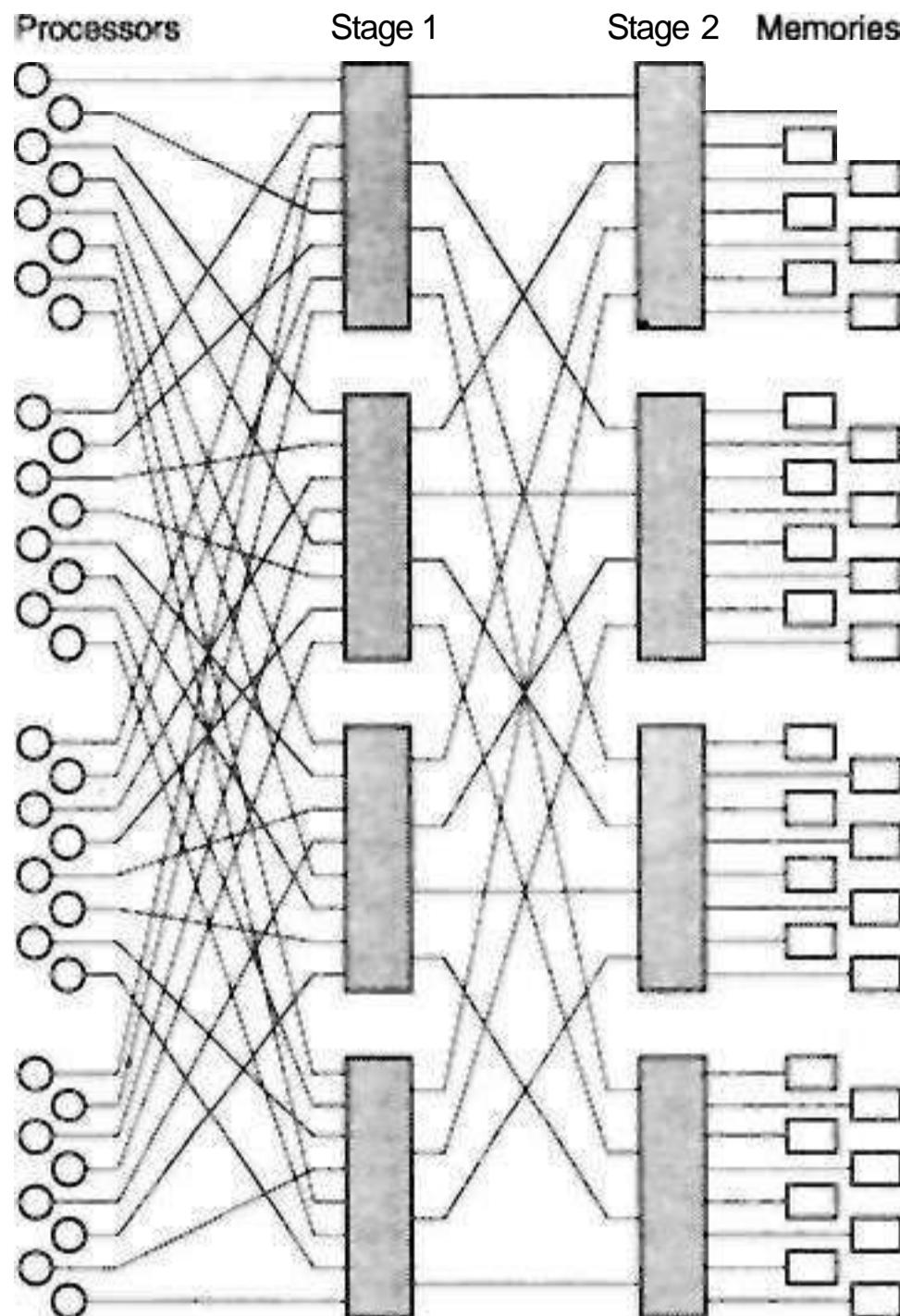
Problem 7.14 Determine the optimal routing paths in the following mesh and hypercube multicomputers.

- (a) Consider a **64-node** hypercube network. Based on the E-cube routing algorithm, show how to route a message from node **(101101)** to node (011010). All intermediate nodes must be **identified** on the routing path.
- (b) Determine two optimal routes for multicast on an 8×8 mesh, subject to the following constraints **separately**. The source node is (3, 5), and there are 10 destination nodes (1,1), (1, 2), **(1, 6)**, (2, 1), (4, 1), (5, **5**), (5, 7), (6, 1), (7, 1), (7, 5). (i) The first multicast route should be implemented with a minimum number of channels. (ii) The second multicast route should result in minimum distances from the source to each of the 10 destinations.
- (c) Based on the greedy algorithm (Fig. 7.39), **determine a suboptimal multicast** route, with minimum distances from the source to all destinations using as few traffic channels as possible, on a **16-node** hypercube network. The source node is (1010), and there are 9 destination nodes (0000), (0001), **(0011)**, (0100), (0101), (0111), **(1111)**, **(1101)**, and **(1001)**.

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



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



- (a) Figure out a fixed priority scheme to avoid conflicts in using the crossbar switches for nonblocking connections. For simplicity, consider only the forward connections from the processors to the memory modules.
- (b) Suppose both stages use 8×8 crossbar switches- Design a **two-stage** Cedar network to provide switched connections between 64 processors and 64 memory modules, again in a clustered manner similar to the above Cedar network design.
- (c) Further expand the Cedar network to three stages using 8×8 crossbar switches as building blocks to connect 512 processors and 512 memory modules. Show the schematic interconnections in all three stages from the input end to the output end.

Chapter 8

Multivector and SIMD Computers

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

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

The evolution from SIMD and MIMD computers to hybrid SIMD/MIMD computer systems is also considered. The Connection Machine CM-5 reflects this architectural trend. This hybrid approach to designing **reconfigurable** computers opens up new opportunities for exploiting coordinated parallelism in complex application problems.

8.1 Vector Processing Principles

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

8.1.1 Vector Instruction Types

Basic concepts behind vector processing **are** defined below. Then we discuss major types of vector instructions encountered in a modern vector machine. The intent is to acquaint the reader with the instruction-set architectures of available vector supercomputers.

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

addressing increment between successive elements called the *stride*.

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

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

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

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

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

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

$$f_1 : V_i \rightarrow V_j \quad (8.1)$$

$$f_2 : V_j \times V_k \rightarrow V_i \quad (8.2)$$

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

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

$$f_3 : s \times V_i \rightarrow V_j \quad (8.3)$$

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

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

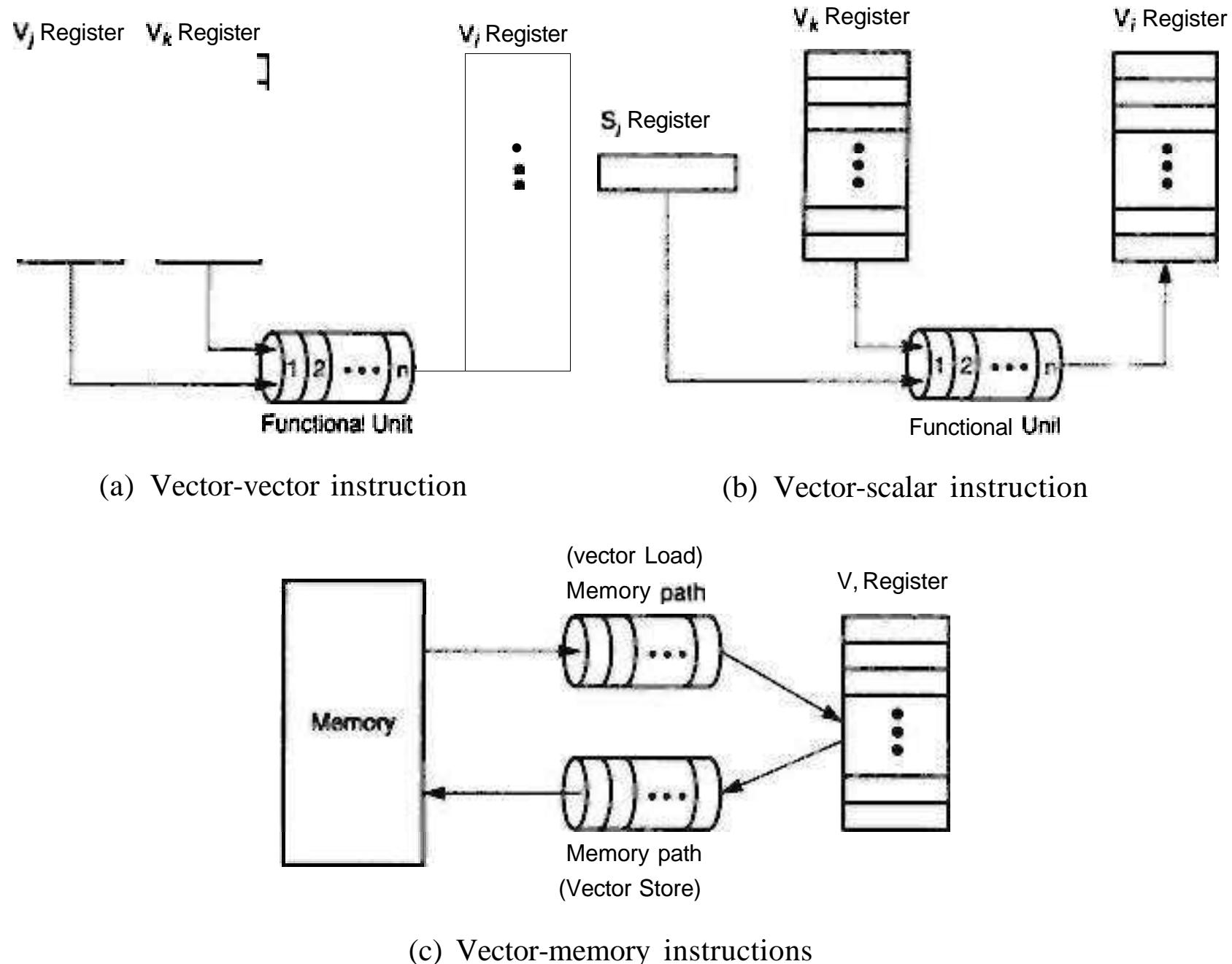


Figure 8.1 Vector instruction types in Cray-like computers.

(M) as defined below:

$$f_4 : M \rightarrow V \quad \text{Vector load} \quad (8.4)$$

$$f_5 : V \rightarrow M \quad \text{Vector store} \quad (8.5)$$

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

$$f_6 : V_i \rightarrow s_j \quad (8.6)$$

$$f_7 : V_i \times V_j \rightarrow s_k \quad (8.7)$$

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

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

$$/a : M \rightarrow V_1 \times V_0 \quad \text{Gather} \quad (8.8)$$

$$fa : V_1 \times V_0 \rightarrow M \quad Scatter \quad (8.9)$$

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

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

$$f_{10} : V_0 \times V_m \rightarrow V_1 \quad (8.10)$$

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

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

The gather instruction (Fig. 8.2a) transfers the contents (600, 400, 250, 200) of nonsequential memory locations (104, 102, 107, 100) to four **elements** of a vector register V_1 . The base address (100) of the memory is indicated by an address register A_0 . The number of elements being transferred is indicated by the contents (4) of a *vector length* register VL .

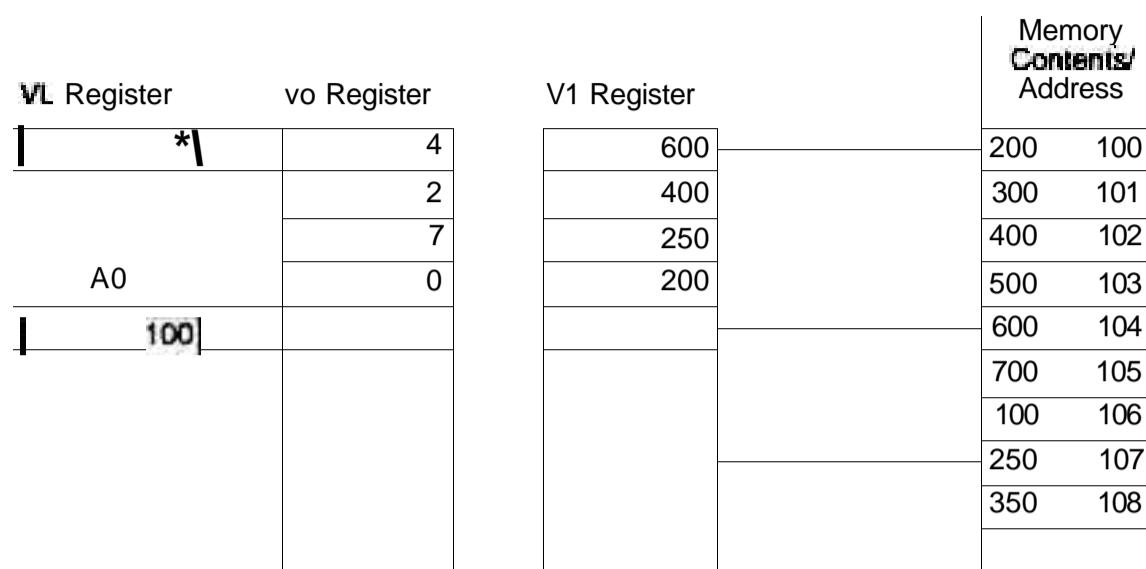
The offsets (indices) from the base address are retrieved from the vector register V_0 . The effective memory addresses are obtained by adding the base address to **the** indices.

The scatter instruction reverses the mapping operations, as illustrated in Fig. 8.2b. Both the VL and A_0 registers are embedded in the instruction.

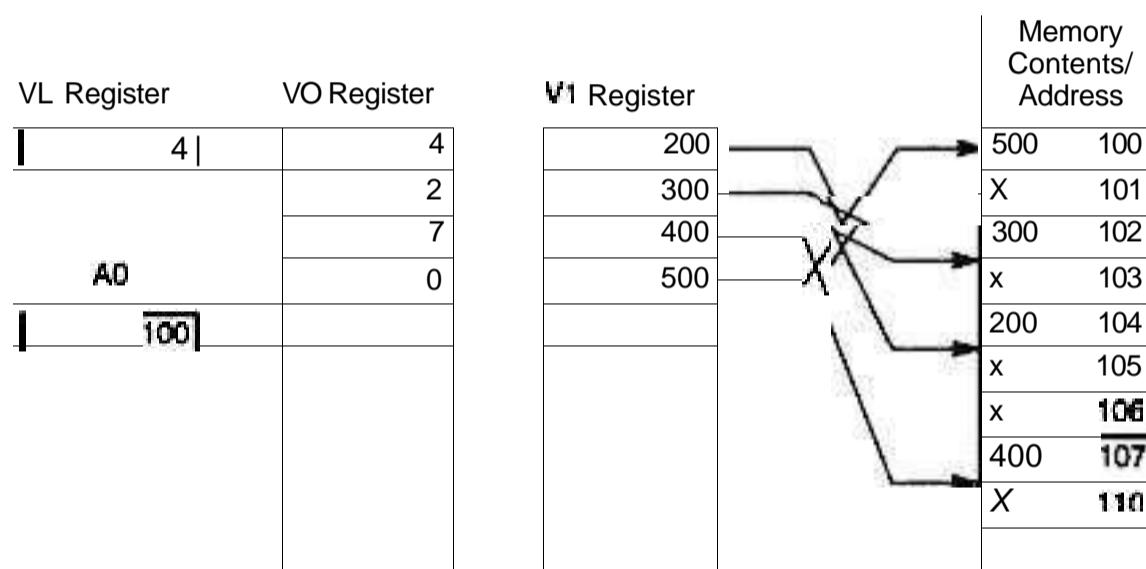
The masking instruction is shown in Fig. 8.2c for compressing a long vector into a short index vector. The contents of vector register V_0 are tested for zero or nonzero elements. A *masking register* (VM) is used to store the test results. After testing and forming the *masking vector* in VM , the corresponding nonzero indices are stored in the V_1 register. The VL register indicates the length of the vector being tested.

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

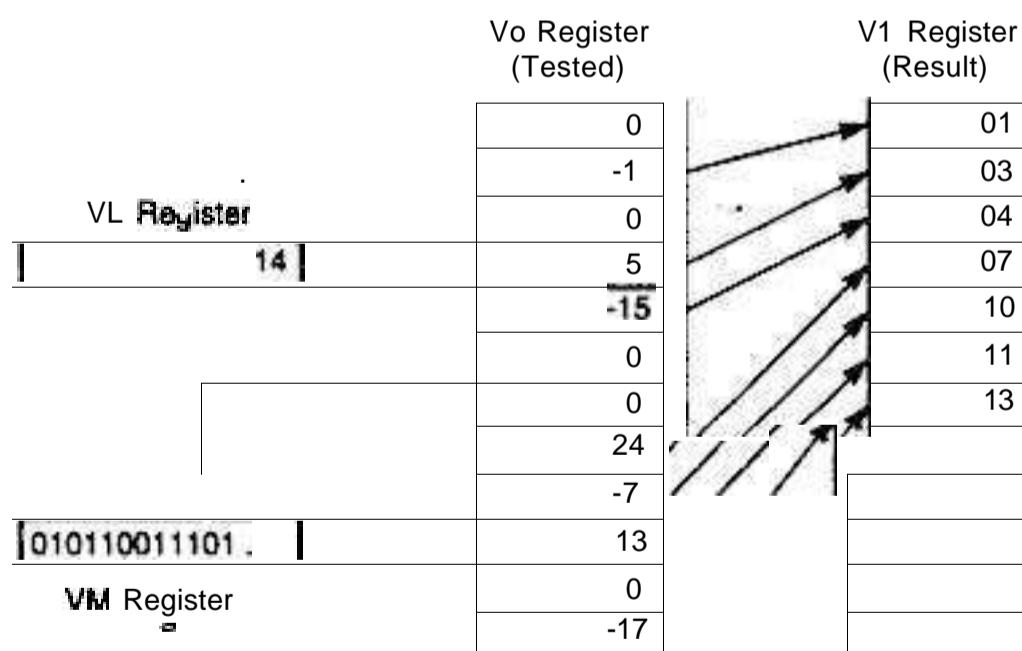
The above instruction types cover the most important ones. A given specific machine may implement an instruction set containing only a subset or even a superset of the above instructions.



a) Gather instruction



b) Scatter instruction



c) Masking instruction

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

8.1.2 Vector-Access Memory Schemes

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

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

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

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

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

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

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

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

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

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

At the end of each memory cycle (Fig. 8.3b), $m = 2^a$ consecutive words are latched

If the stride is greater than 1, then the throughput decreases, roughly proportionally to the **stride**.

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

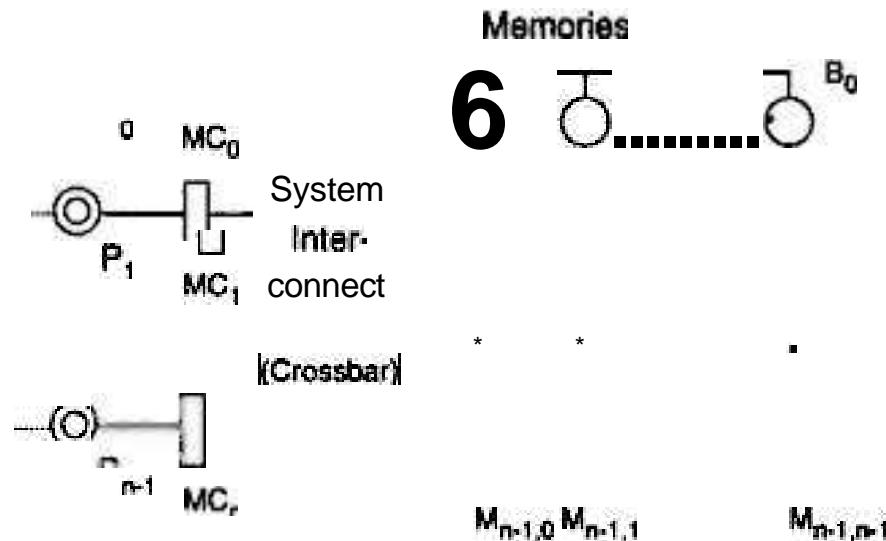


Figure 8.4 The C/S memory organization. (Courtesy of D.K. Panda, 1990)

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

8.1.3 Past and Present Supercomputers

This section introduces full-scale supercomputers manufactured recently in the United States and in Japan. Five supercomputer families are reviewed, including the Cray Research Series, the CDC/ETA Series, the Fujitsu VP Series, the NEC SX **Series**, and the Hitachi 820 Series (Table 8.1). The relative performance of these machines for vector processing are compared with scalar processing at the end.

The Cray Research Series Seymour Cray founded Cray Research, Inc. in 1972. Since then, about 400 units of Cray supercomputers have been produced and installed **worldwide**.

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

to eight processors in a single system using a 6-ns clock rate and 256 Mbytes of shared memory.

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

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

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

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

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

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

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

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

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

Besides the vector unit, a high-speed scalar unit employs RISC architecture with 128 scalar registers. Instruction **reordering** is supported to exploit higher parallelism. The main memory is 1024-way interleaved. The extended memory of up to 16 Gbytes

Figure 8.6 plots the relative performance V as a function of r with $/$ as a running parameter. The higher the value of $/$, the higher the relative speedup. The IBM 3090 with vector facility (VF) was a high-end mainframe with add-on vector hardware.

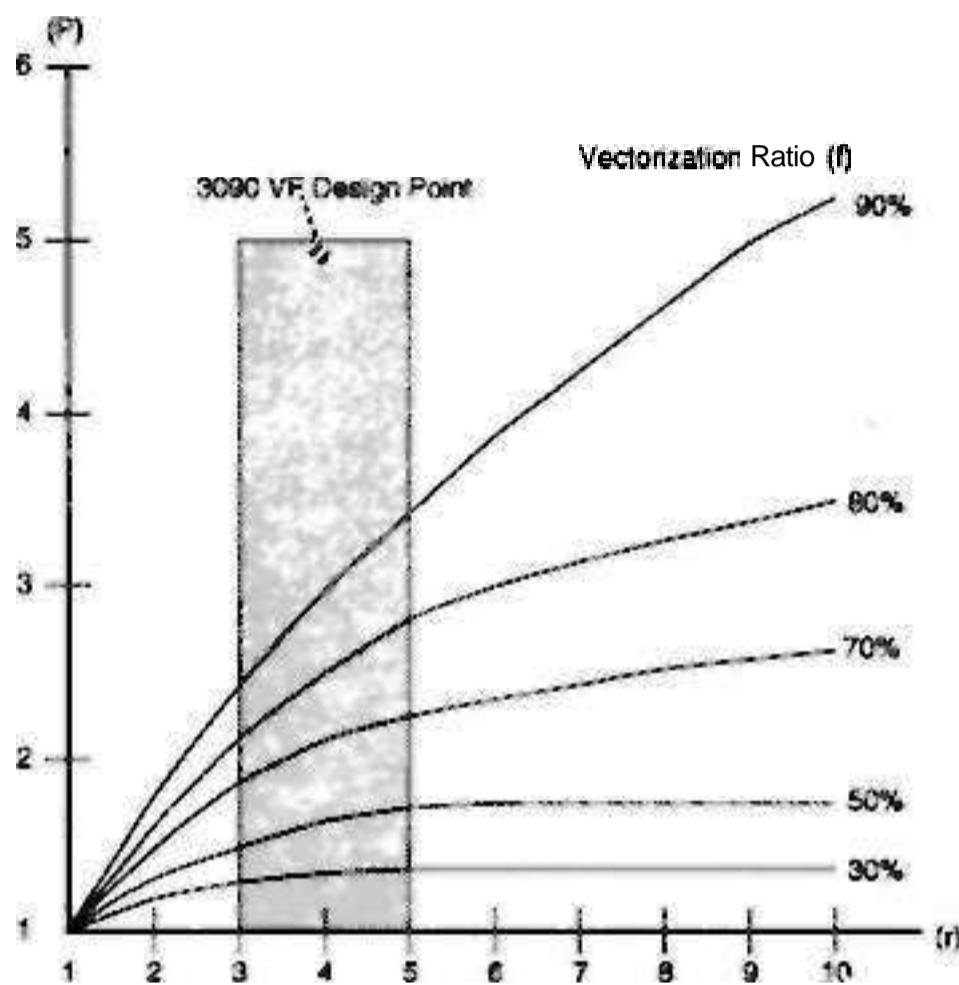


Figure 8.6 Speedup performance of vector processing over scalar processing in the IBM 3090/VF. (Courtesy of IBM Corporation, 1986)

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

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

On the other hand, $P \rightarrow r$ when $/ \rightarrow 1$. Scientific supercomputer designers like Cray and Japanese manufacturers often choose a much higher **speed** ratio, say, $10 < r < 25$, because they expect a higher vectorization ratio $/$ in user programs, or they use better vectorizers to increase the ratio to a desired level.

Vector performance can be enhanced with replicated functional unit pipelines in each processor. Another approach is to apply superpipelining on vector units with a double or triple clock rate with respect to scalar pipeline operations.

Longer vectors are required to really achieve the target performance. It was projected that by the year 2000, an 8-Gflops peak will be achievable with multiple functional units running simultaneously in a processor.

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

Japanese supercomputers certainly outperform Cray Research machines in vector capability. One of the contributing factors is the high clock rate, and other factors include use of a better compiler and the optimization support provided.

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

Table 8.2 Vector and Scalar Performance of Various Supercomputers

Machine	Cray 1S	Cray 25	Cray X-MP	Cray Y-MP	Hitachi S820	NEC SX2	Fujitsu VP400
Vector performance (Mflops)	85.0	151.5	143.3	201.6	737.3	424.2	207.1
Scalar performance (Mflops)	9.8	11.2	13.1	17.0	17.8	9.5	6.6
Vector balance point	0.90	0.93	0.92	0.92	0.98	0.98	0.97

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

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

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

The I/O is defined as the transfer of data between the mainframe and peripherals

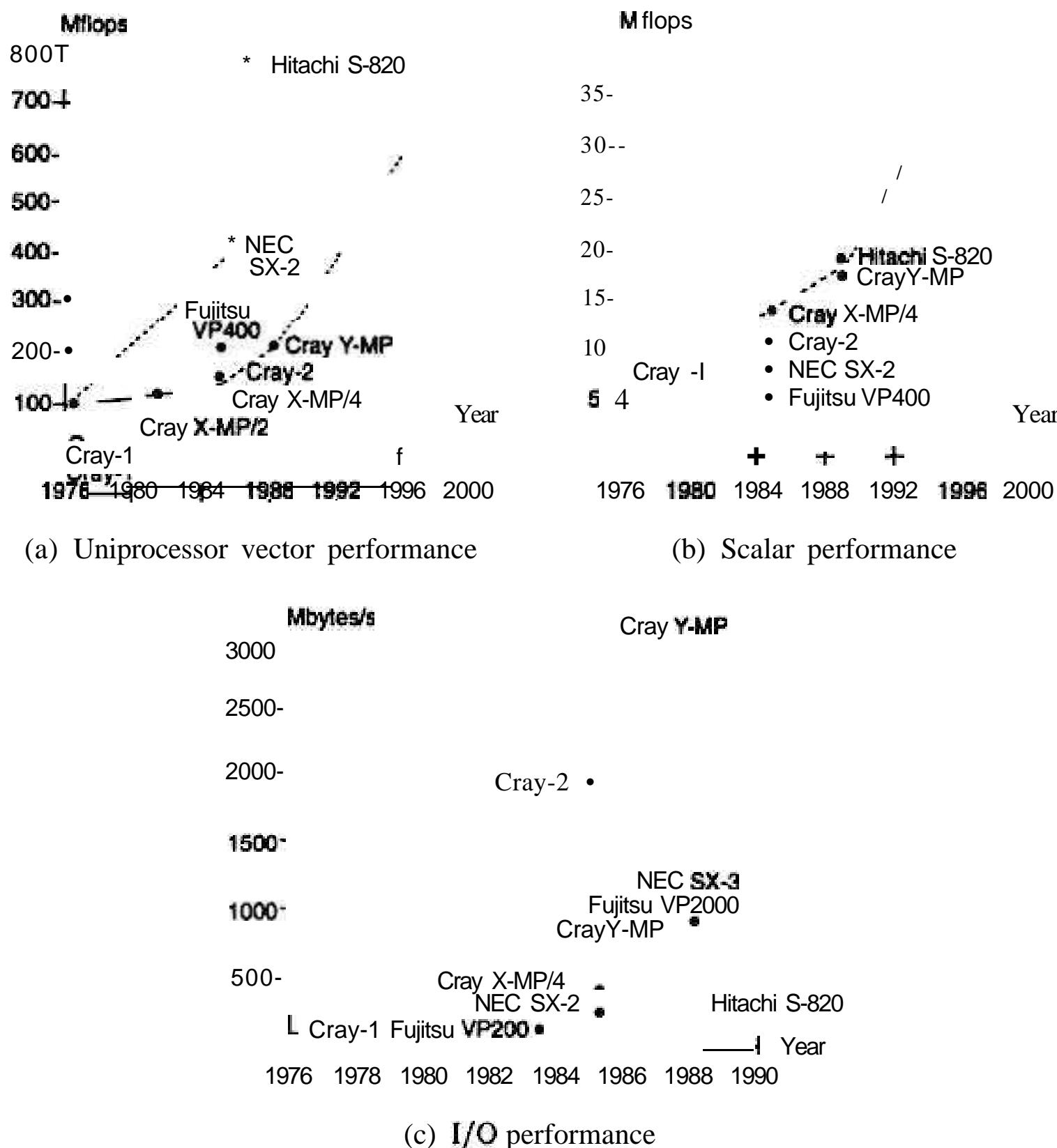


Figure 8.7 Some reported supercomputer performance data (Source: Smith, Hsu, and Hsiung, *IEEE Supercomputing Conference*, 1990)

or a network. Over the past generation, **I/O bandwidths** have not always been well correlated with computational performance. **I/O** processor architectures were implemented by Cray Research with two different approaches.

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

It is expected that in the next 5 to 10 years, more than a 50-Gbytes/s **I/O** transfer rate will be needed in supercomputers connected to high-speed disk arrays and net-

works. Support for high-speed networking will become a major component of the I/O architecture in supercomputers.

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

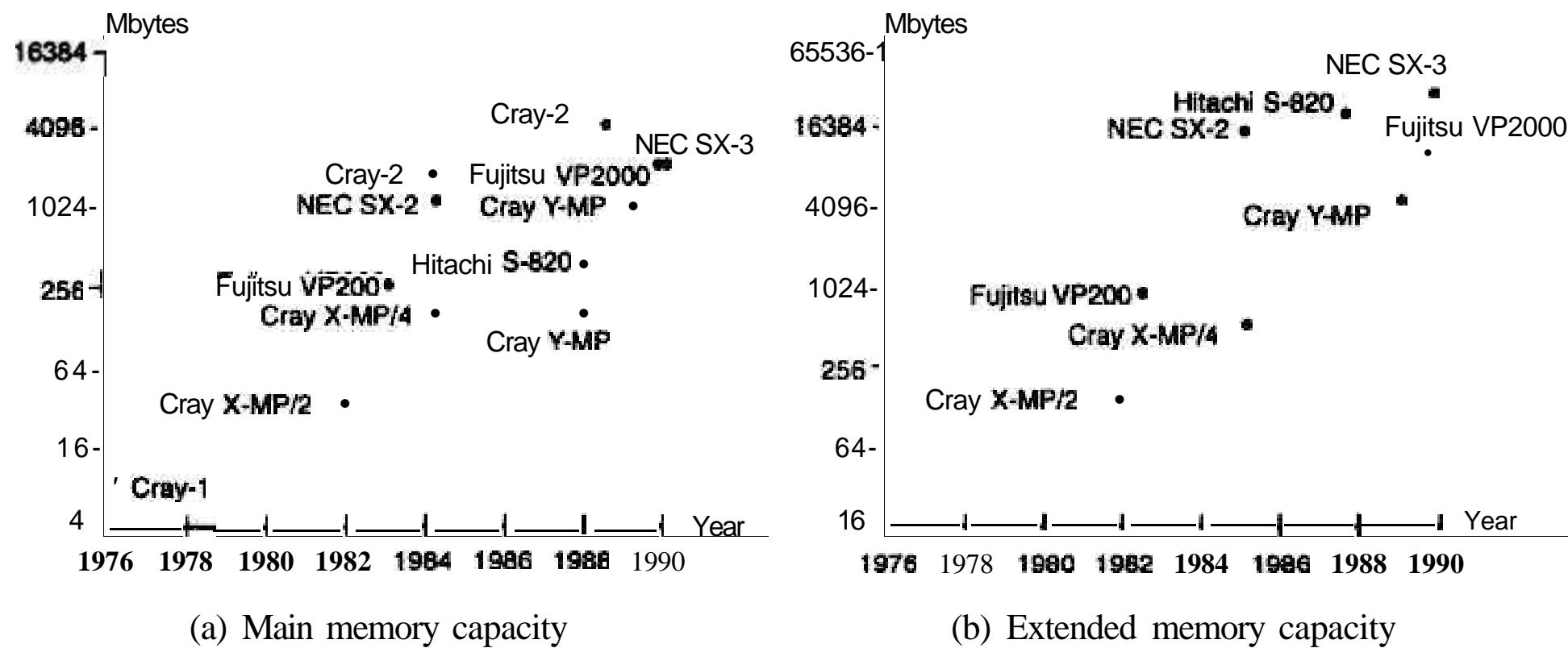


Figure 8.8 Supercomputer memory capacities (Source: Smith, Hsu, and Hsiung, *IEEE Supercomputing Conference*, 1990)

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

It is expected that main memory will reach 32 Gbytes with an extended memory four times as large in the next decade. This will match the expected growth of supercomputer architectures of 64 processors by the mid-1990s and 1024 processors by the **end of the century**.

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

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

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

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

Table 8.3 Architectural Characteristics of Three Current Supercomputers

Machine Characteristics	Cray Y-MP C90/16256	NEC SX-X Series	Fujitsu VP-2000 Series
Number of processors	16 CPUs	4 arithmetic processors	1 for VP2600/10, 2 for VP2400/40
Machine cycle time	4.2 ns	2-9 ns	3.2 ns
Max. memory	256M words (2 Gbytes).	2 Gbytes, 1024-way interleaving.	1 or 3 Gbytes of SRAM.
Optional SSD memory	512M, 1024M, or 2048M words (16 Gbytes).	16 Gbytes with 2.75 * Gbytes/s transfer rate.	32 Gbytes of extended memory.
Processor architecture: vector pipelines, functional and scalar units	Two vector pipes and two functional units per CPU, delivering 64 vector results per clock period.	Four sets of vector pipelines per processor, each set consists of two adder/shift and two multiply/logical pipelines. A separate scalar pipeline.	Two load/store pipes and 5 functional pipes per vector unit, 1 or 2 vector units, 2 scalar units can be attached to each vector unit.
Operating system	UNICOS derived from UNIX/V and BSD.	Super-UX based on UNIX System V and 4.3 BSD.	UXP/M and MSP/EX enhanced for vector processing.
Front-ends	IBM, CDC, DEC, Univac , Apollo, Honeywell.	Built-in control processor and 4 I/O processors.	IBM-compatible hosts.
Vectorizing languages / compilers	Fortran 77, C, CF77 5.0, Cray C release 3.0	Fortran 77/SX , Vectorizer/XS , Analyzer/SX .	Fortran 77 EX/VP , C/VP compiler with interactive vectorizer .
Peak performance and I/O bandwidth	16 Gflops , 13.6 Gbytes/s.	22 Gflops, 1 Gbyte/s per I/O processor.	5 Gflops, 2 Gbyte/s with 256 channels.

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

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

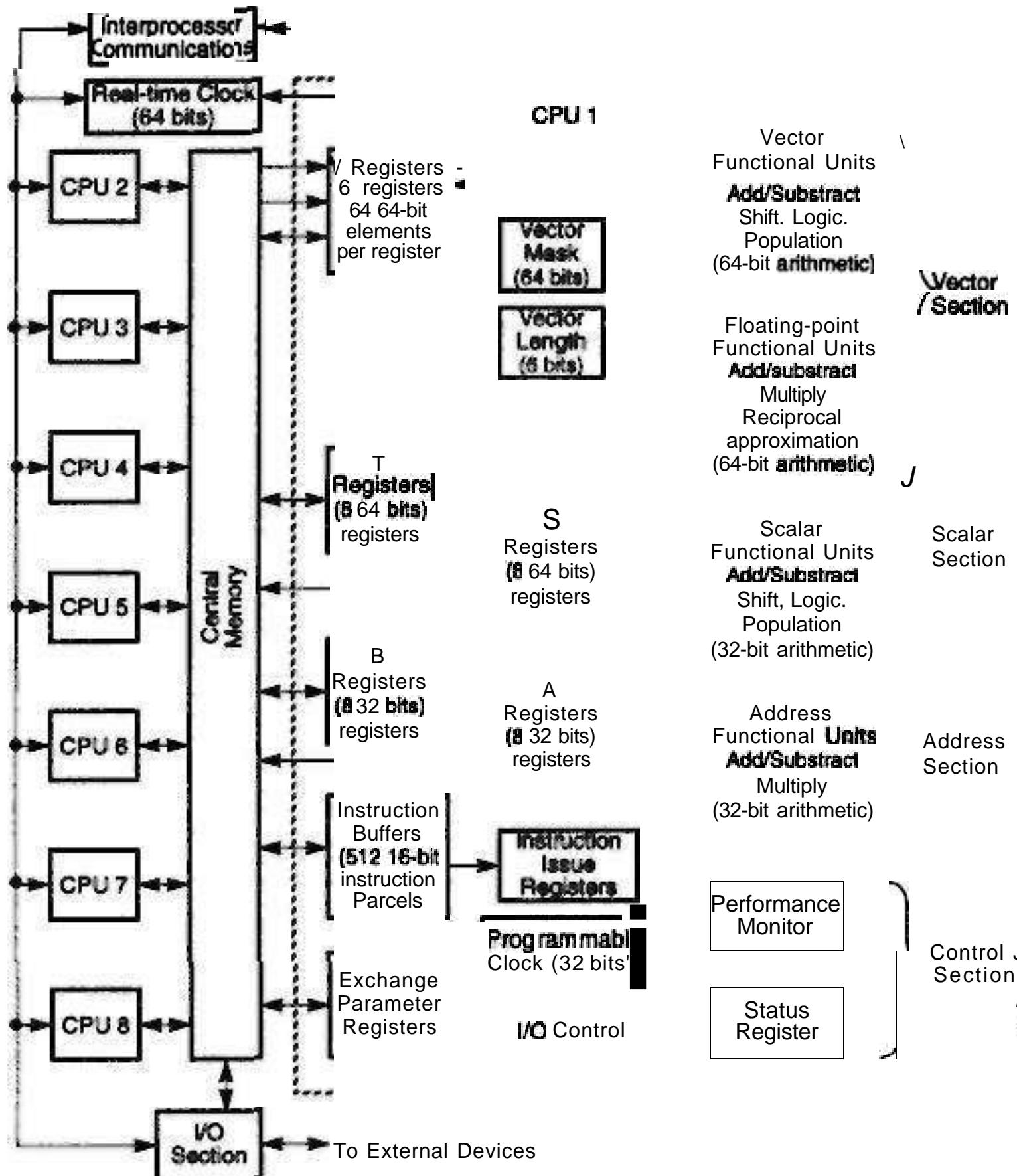


Figure 8.9 Cray Y-MP 816 system organization. (Courtesy of Cray Research, Inc., 1991)

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

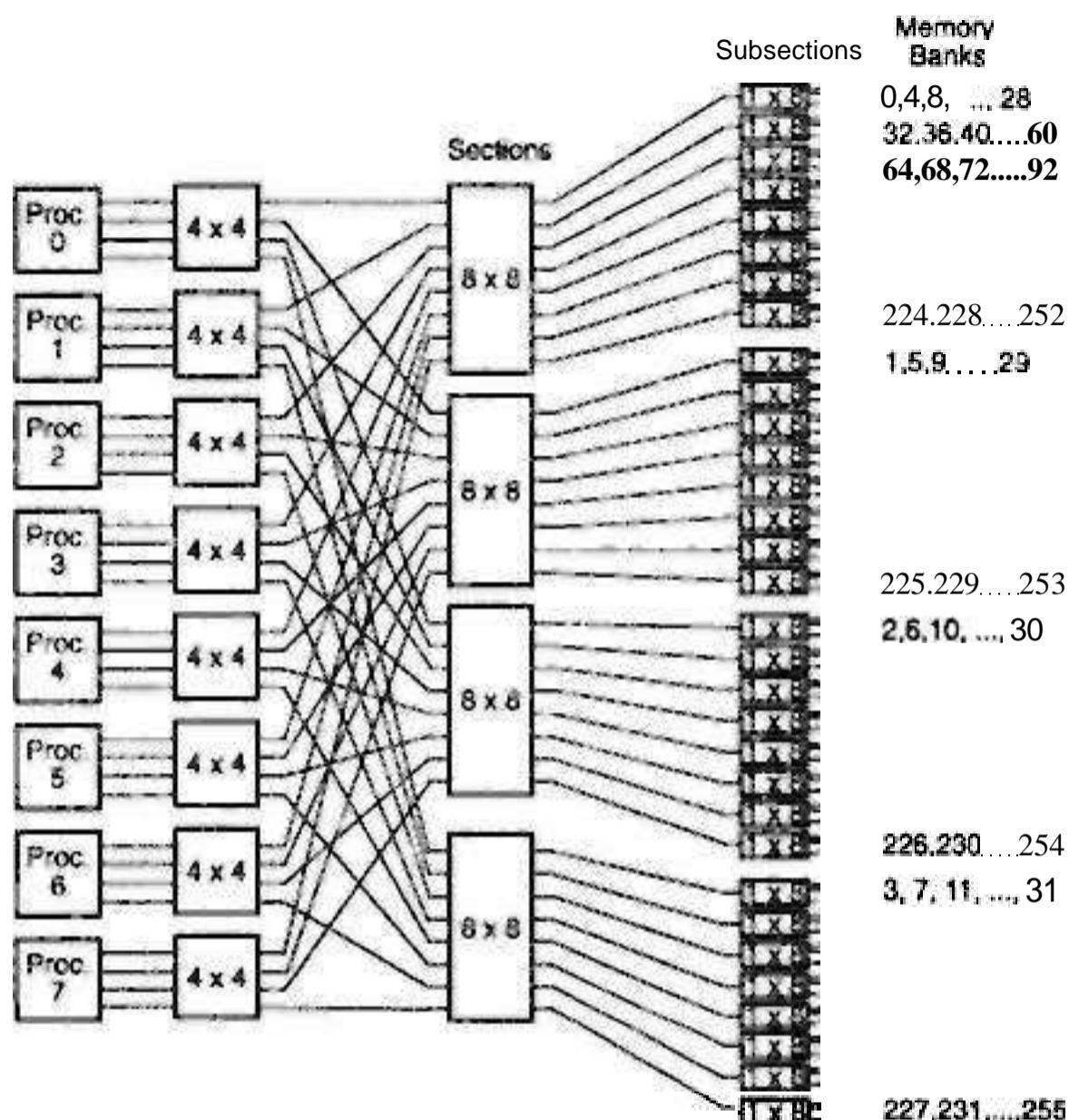


Figure 8.10 Schematic logic diagram of the crossbar network **between** 8 processors and 256 memory banks in the Cray Y-MP 816.

The C-90 and Clusters The C-90 is further enhanced in technology and scaled in size from the Y-MP Series. The architectural features of **C-90/16256** are summarized in Table 8.3. The system is built with 16 CPUs, each of which is similar to that used in the Y-MP. The system uses up to 256 **megawords** (2 Gbytes) of shared main memory among the 16 processors. Up to 16 Gbytes of SSD memory is available as optional secondary **main** memory. In each cycle, two vector pipes and two functional units can operate in parallel, producing four vector results per clock. This implies a four-way parallelism within each processor. Thus 16 processors can deliver a maximum of 64 vector results per clock cycle.

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

Multiple C-90's can be used in a clustered configuration in order to solve **large-scale** problems. As illustrated in Fig. 8.11, four C-90 clusters are connected to a group of SSDs



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

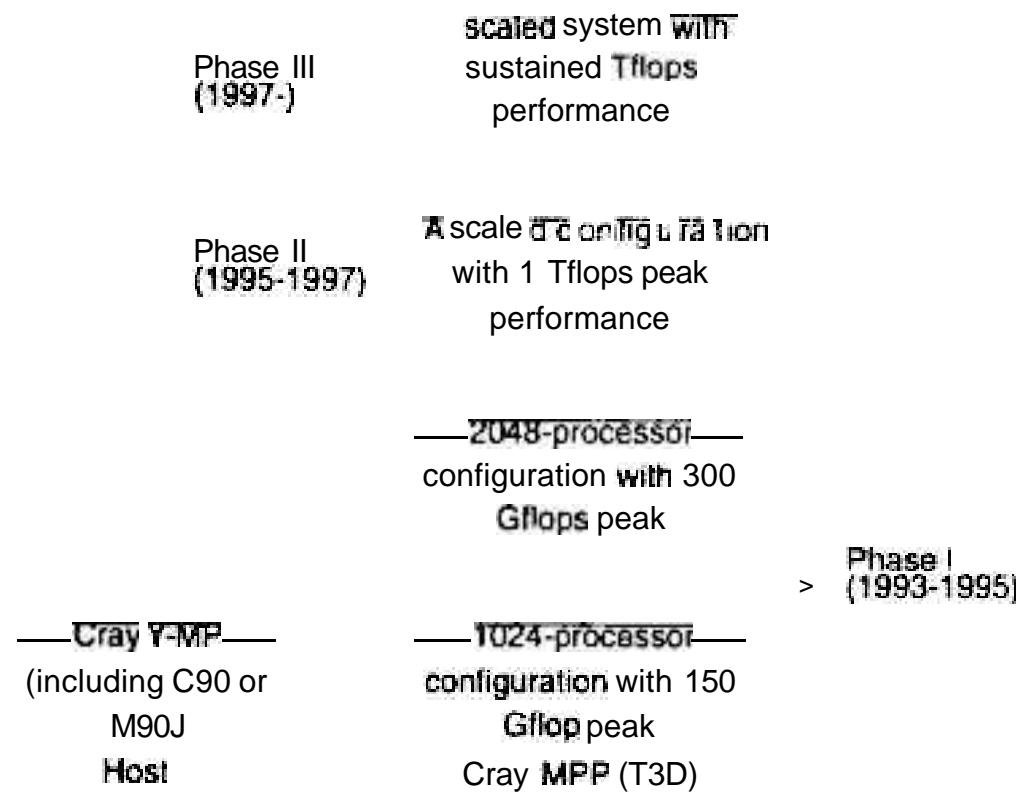


Figure 8.12 The development phases of the Cray/MPP system. (Courtesy of Cray Research, Inc. 1992)

8.2.3 Fujitsu VP2000 and VPP500

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

The Fujitsu VP2000 Figure 8.13 shows the architecture of the VP-2G00/10 uniprocessor system. The system can be expanded to have dual processors (the VP-2400/40). The system clock is 3.2 ns. the main memory unit has 1 or 2 Gbytes, and the system storage unit provides up to 32 Gbytes of extended memory.

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

Example 8.5 Reconfigurable vector register file in the Fujitsu VP2000

Vector registers in Cray and Fujitsu machines are illustrated in Fig. 8.14. Cray machines use 8 vector registers, and each has a fixed length of 64 component registers. Each component register is 64 bits wide as demonstrated in Fig. 8.14a.

A component counter is built within each Cray vector register to keep track

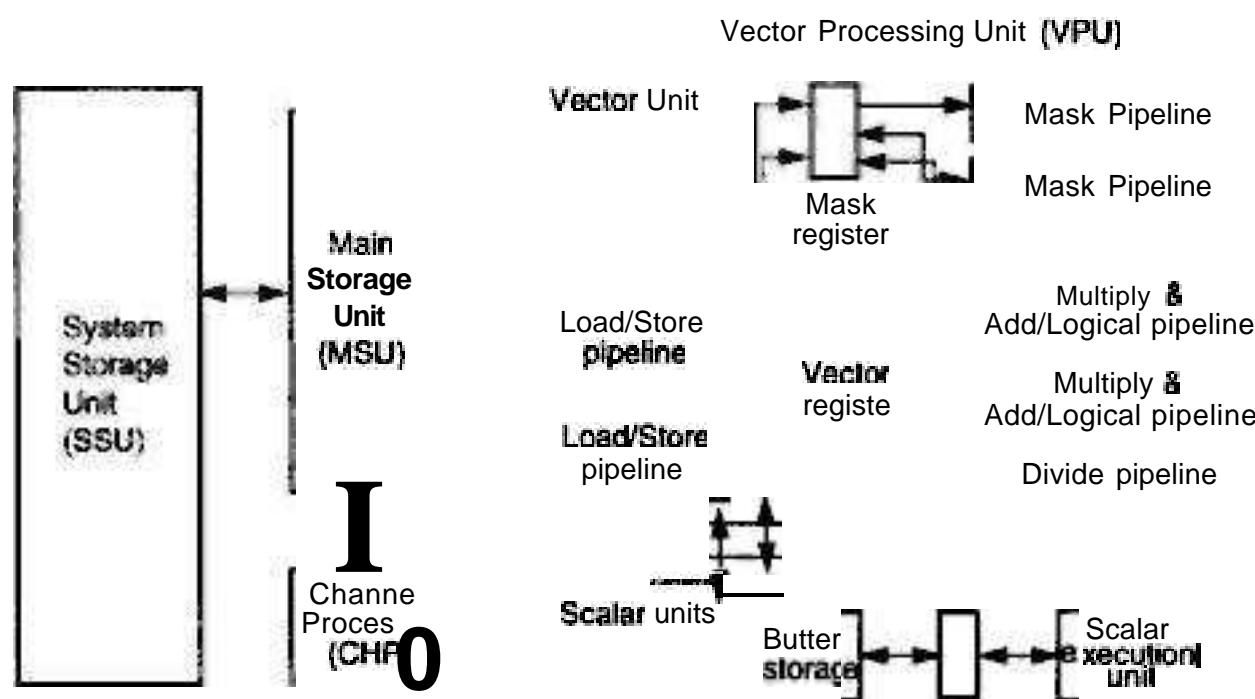


Figure 8.13 The Fujitsu VP2000 Series supercomputer architecture. (Courtesy of Fujitsu Ltd., 1991)

of the number of vector elements fetched or processed. A segment of a 64-element **subvector** is held as a package in each vector register. Long vectors must be divided into 64-element segments before they can be processed in a pipelined fashion.

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

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

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

```

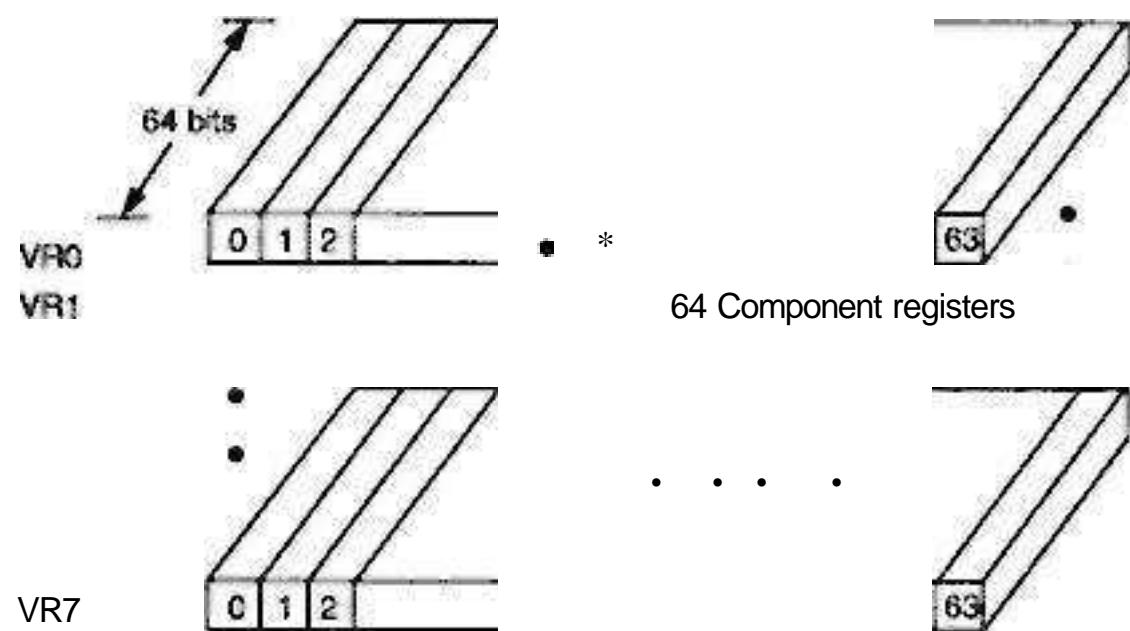
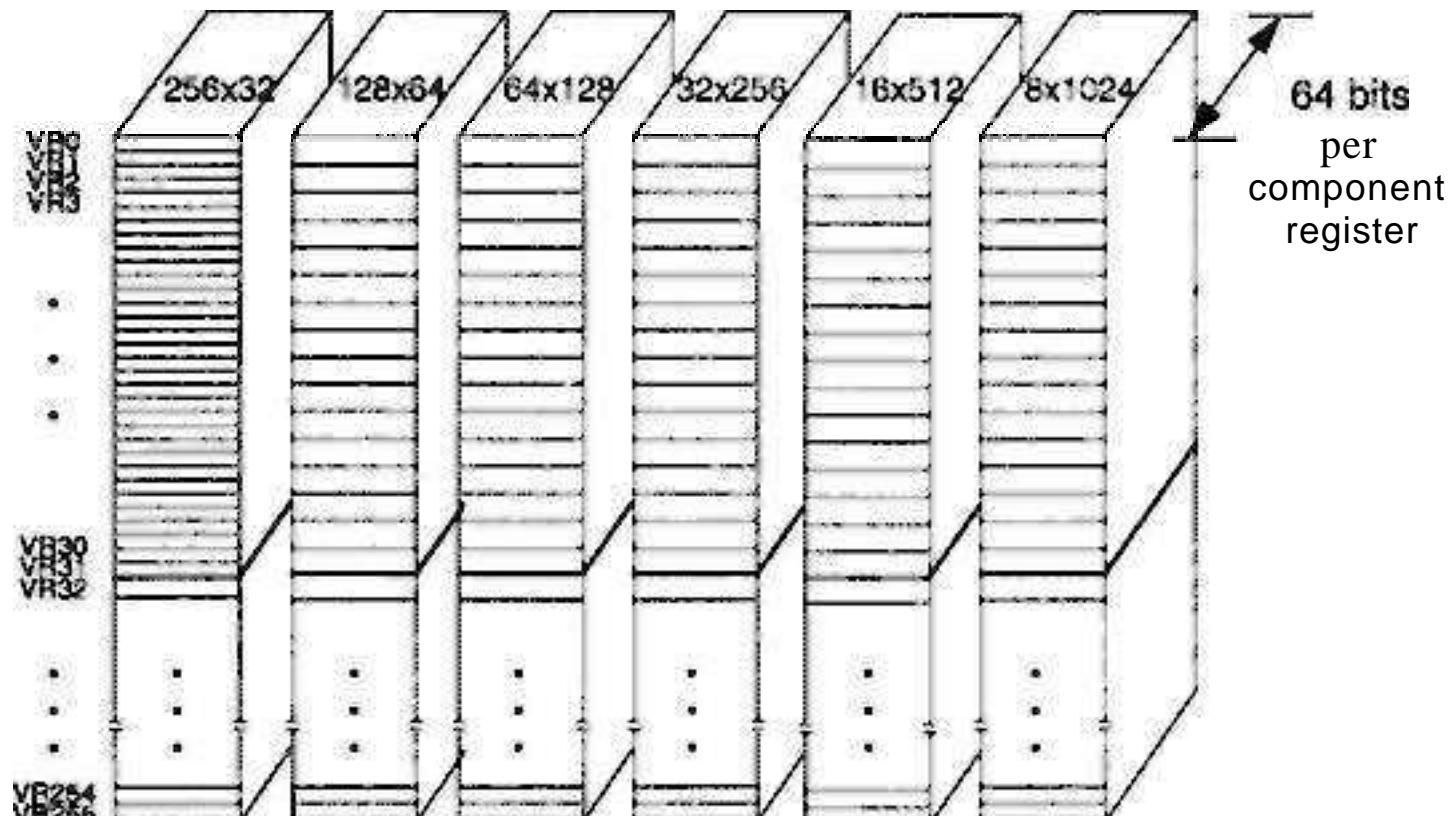
Do 10 I = 0, 31
ZZ0(I) = U(I,J,K)   U(I,J-1,K)
ZZ1(I) = V(I,J,K)   V(I,J-1,K)

ZZ84(I) = W(I,J,K) - W(I,J-1,K)
10 Continue

```

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

Software support for parallel and vector processing in the above supercomputers will be treated in Part IV. This includes multitasking, **macrotasking**, microtasking,

(a) Eight vector registers ($8 \times 64 \times 64$ bits) on Cray machines

(b) Vector register configurations in the Fujitsu VP2000

Figure 8.14 Vector register file in Cray and Fujitsu supercomputers.

autotasking, and interactive compiler optimization techniques for vectorization or parallelization.

The VPP 500 This is a new supercomputer series from Fujitsu, called *vector parallel processor*. The architecture of the VPP500 is scalable from 7 to 222 PEs, offering a highly parallel MIMD multivector system. The peak performance is targeted for 335 Gflops with the first shipment in September 1993. Figure 8.15 shows the architecture of the VPP500 used as a back-end machine attached to a VP2000 or a VPX 200 host.

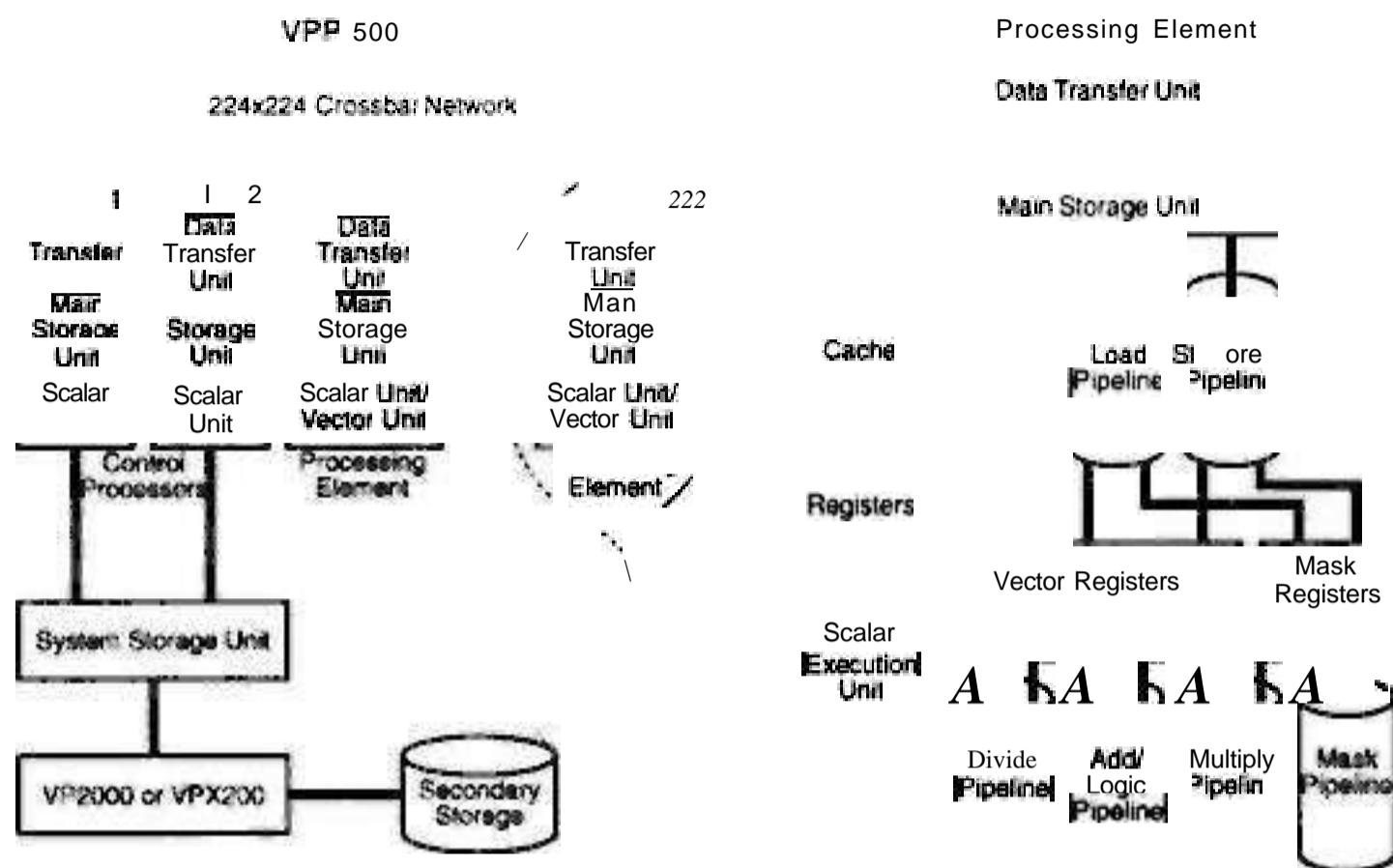


Figure 8.15 The Fujitsu VPP500 architecture. (Courtesy of Fujitsu Ltd., 1992)

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

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

The VPP500 runs jointly with its host the UNIX System V Release 4-based **UXP/VPP** operating system that supports closely coupled **MIMD** operations. The optimization functions of the Fortran 77 compiler work with the parallel scheduling function of the UNIX-based OS to exploit the maximum capability of the vector parallel architecture. The IEEE 754 floating-point standard has been adopted.

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

The system is scalable with an incremental control structure. A single control

processor is sufficient to control up to 9 PEs. Two control processors are used to coordinate a VPP with 30 to 222 PEs. The system performance is expected to scale with the number of PEs spanning a peak performance range from 11 to 335 Gflops and a memory capacity of 1.8 to 55 Gbytes. The real performance of both the Cray/MPP and Fujitsu VPP500 remain to be seen when the systems become operational.

8.2.4 Mainframes and Minisupercomputers

High-end mainframes, minisupercomputers, and supercomputing workstations are studied in this section. Besides summarizing current systems, we examine the architecture designs in the VAX 9000 and Stardent 3000 as case studies. The UNPACK results compiled by Dongarra (1992) are presented to compare a wide range of computers for solving linear systems of equations.

High-End Mainframe Supercomputers This class of supercomputers have been called near-supercomputers. They offer a peak performance of several hundreds of Mflops to 2.4 Gflops as listed in Table 8.4. These machines are not designed entirely for number crunching. Their main applications are still in business and transaction processing. The floating-point capability is only an add-on optional feature of these mainframe machines.

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

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

DEC VAX 9000 Even though the VAX 9000 cannot provide a Gflops performance, the design represents a typical mainframe approach to high-performance computing. The architecture is shown in Fig. 8.16a.

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

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

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

Table 8-4 High-end Mainframe Supercomputers

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

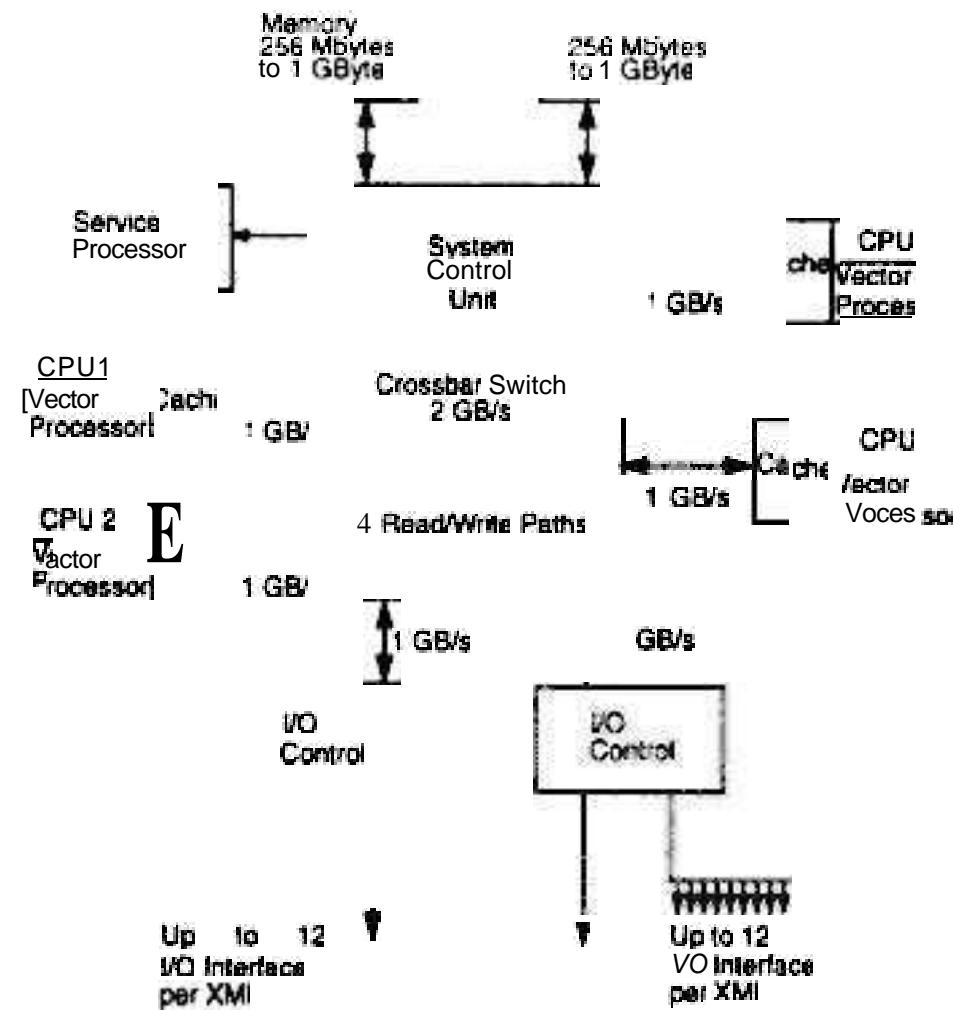
systems.

Each vector processor (VBOX) is equipped with an add and a multiply pipeline using vector registers and a mask/address generator as shown in Fig. 8.16b. Vector instructions are fetched through the memory unit (MBOX), decoded in the IBOX, and issued to the VBOX by the EBOX. Scalar operations are directly executed in the EE BOX.

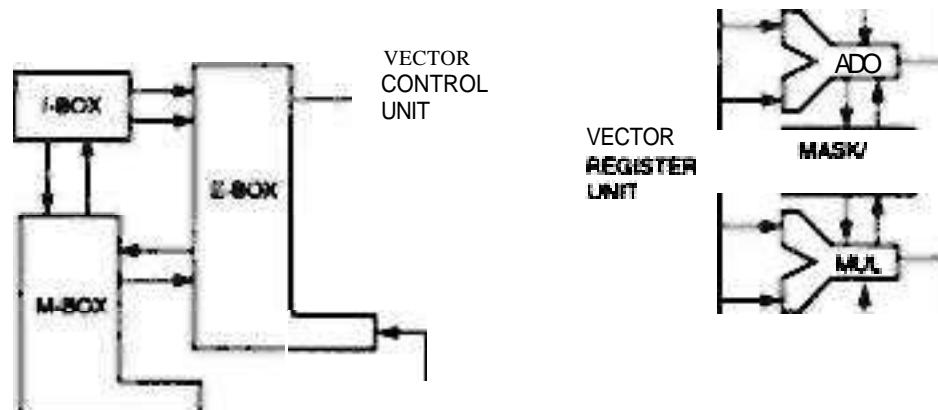
The vector register file consists of $16 \times 64 \times 64$ bits, divided into sixteen 64-element vector registers. No instruction takes more than five cycles. The vector processor generates two 64-bit results per cycle, and the vector pipelines can be chained for dot-product operation/..

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

Minisupercomputers These are a class of low-cost supercomputer systems with a performance of about 5 to 15% and a cost of 3 to 10% of that of a full-scale supercomputer. Representative systems include the Convex C Series, Alliant FX Series, Encore



(a) The VAX 9000 multiprocessor system



(b) The vector processor (VBOX)

Figure 8*16 The DEC VAX 9000 system architecture and vector processor **design**.
 (Courtesy of Digital Equipment Corporation, 1991)

Multimax Series, and Sequent Symmetry Series.

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

Both scalar and vector processing is supported in these multiprocessor systems with shared memory and **peripherals**. Most of these systems are built with a graphic subsystem for visualization and performance-tuning purposes.

Supercomputing Workstations High-performance workstations are being produced by Sun Microsystems, IBM, DEC, HP, Silicon Graphics, and Stardent using the state-of-the-art superscalar or superpipelined RISC microprocessors introduced in Chapters 4 and 6. Most of these workstations have a uniprocessor configuration with built-in graphics support but no vector hardware.

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

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

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

The buses carry 32-bit addresses and 64-bit data and operate at 16 MHz. They are rated at 128 Mbytes/s each. The R-bus is dedicated to data transfers from memory to the vector processor, and the **S-bus** handles all other transfers. The system can support a maximum of 512 Mbytes of memory.

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

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

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

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

Many published **Mflops** and **Gflops** results are based on running the LINPACK code with **prespecified** compilers. LINPACK programs can be characterized as having



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

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

The program graph can be specified either by the programmer or by a compiler. The pipeline can be set up by using a sequence of microinstructions or by using hardwired control. Various connection patterns in the crossbar networks can be prestored for implementing each CVF type. Once the CVF is decoded, the connect pattern is enabled for setup dynamically.

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

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

A *cycle* is a sequence of nodes and edges which starts and ends with the same node. We will concentrate on *synchronous* program graphs in which all cycles have positive delays. In particular, we consider a fc-graph, a synchronous program graph in which all nodes have a delay of k cycles. A 0-graph is called a *systolic program graph*,

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

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

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

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

Example 8.9 Program graph transformation to set up a pipenet (Hwang



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

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

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

8.4.3 The MasPar MP-1 Architecture

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

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

The MP-1 family includes configurations with 1024, 4096, and up to 16,384 processors. The peak performance of the 16K-processor configuration is 26,000 MIPS in 32-bit RISC integer operations. The system also has a peak floating-point capability of 1.5 Gflops in single-precision and 650 Mflops in double-precision operations.

Array Control Unit The ACU is a 14-MIPS scalar RISC processor using a demand-paging instruction memory. The ACU fetches and decodes MP-1 instructions, computes addresses and scalar data values, issues control signals to the PE array, and monitors the status of the PE array.

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

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You should first compute all the *multiply* and *add* operations on local data before starting to route data to neighboring PEs. The SIMD *shift* operations can be either east, west, south, or north with wraparound connections on the torus.

- (c) Estimate the total number of SIMD instruction cycles needed to compute the matrix multiplication. The time includes all arithmetic and data-routing operations. The final product elements $C = A \times B = (c_{ij})$ end up in various PE memories without duplication.
- (d) Suppose data duplication is allowed initially by loading the same data element into multiple PE memories. Devise a new algorithm to further reduce the SIMD execution **time**. The initial data duplication time, using either data *broadcast* instructions or data *routing* (shifting) instructions, must be counted. Again, each result element c_{ij} ends up in only one PE memory.

Problem 8.16 Compare the Connection Machines CM-2 and CM-5 in their architectures, operation modes, functional capabilities, and potential performance based on technical data available in the literature. Write a report summarizing your comparative studies and comment on the improvement made in the CM-5 over the CM-2 from the viewpoints of a computer architect and of a machine programmer.

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

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

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

- (a) Design a minimum-time parallel algorithm to perform concurrent vector operations on the given multiprocessor, ignoring all memory-access and **I/O** operations.
- (b) Compare the performance of the multiprocessor algorithm with that of a sequential algorithm on a uniprocessor without the pipelined vector hardware.

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

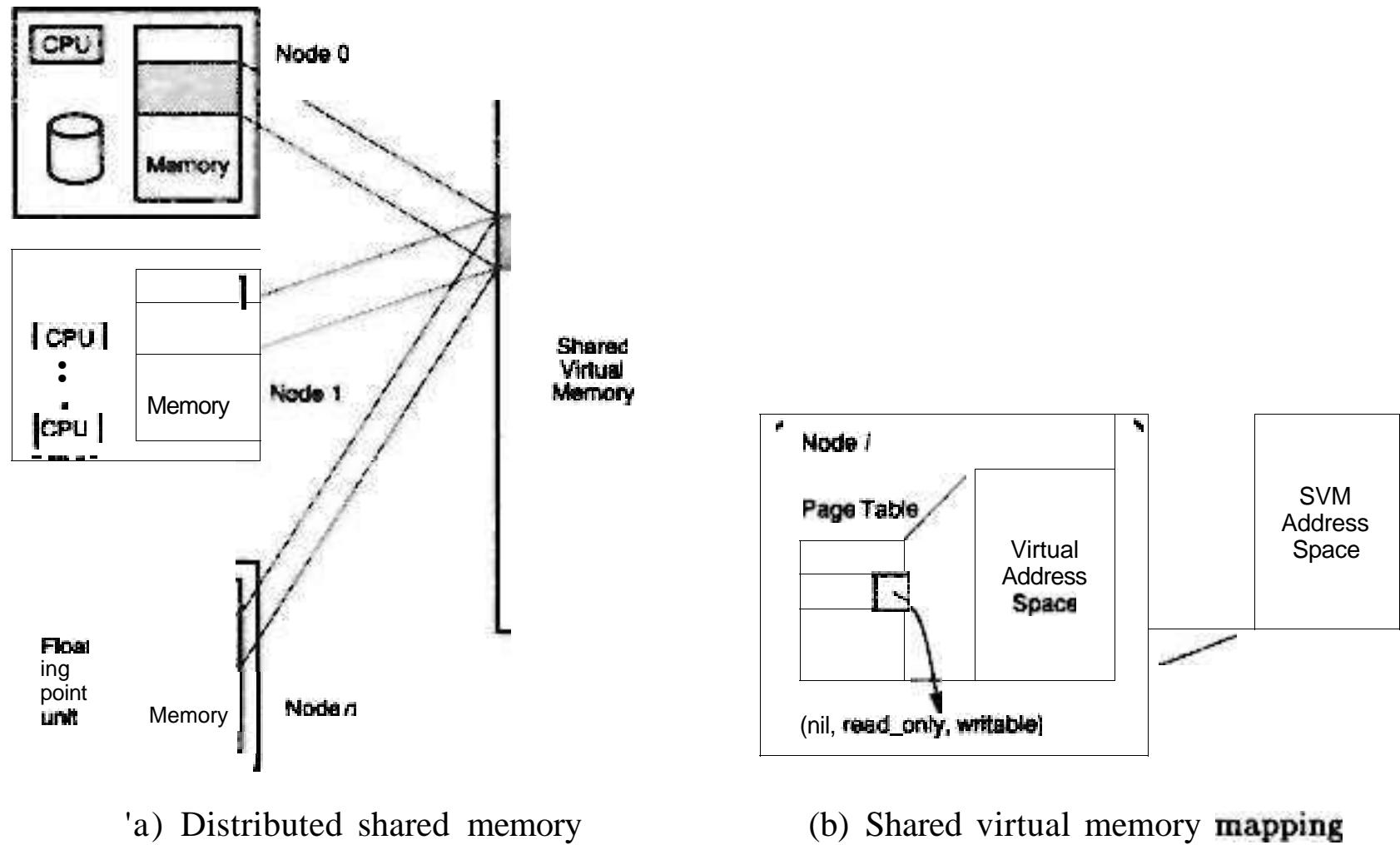


Figure 9.2 The concept of distributed shared memory with a global virtual address space shared among all processors on loosely coupled processing nodes in a massively parallel architecture. (Courtesy of Kai Li, 1992)

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

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

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

Example SVM Systems Nitzberg and Lo (1991) have conducted a survey of SVM systems. Excerpts from their survey, descriptions of four representative SVM systems are summarized in Table 9.1. The Dash implements SVM with a **directory-based** coherence protocol. The Linda offers a shared associative object memory with access functions. The Plus uses a **write-update** coherence protocol and performs replication only by program request. The Shiva extends the IVY system for the Intel iPSC/2 **hypercube**. In using SVM systems, there exists a tendency to use large block (page) sizes



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

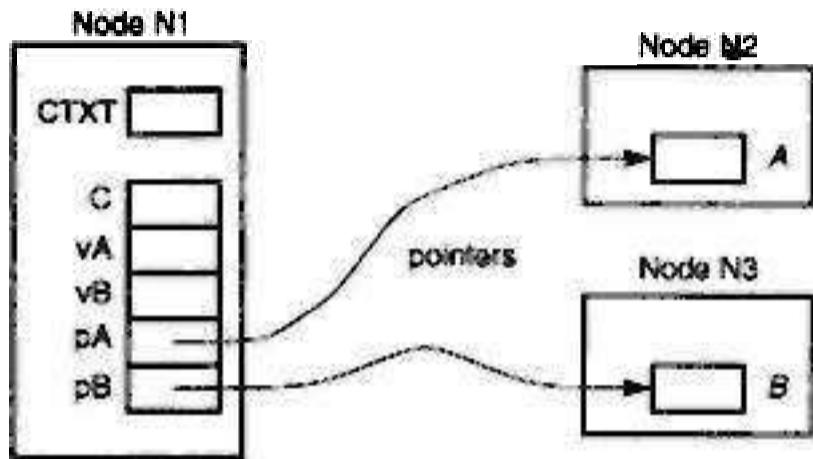


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

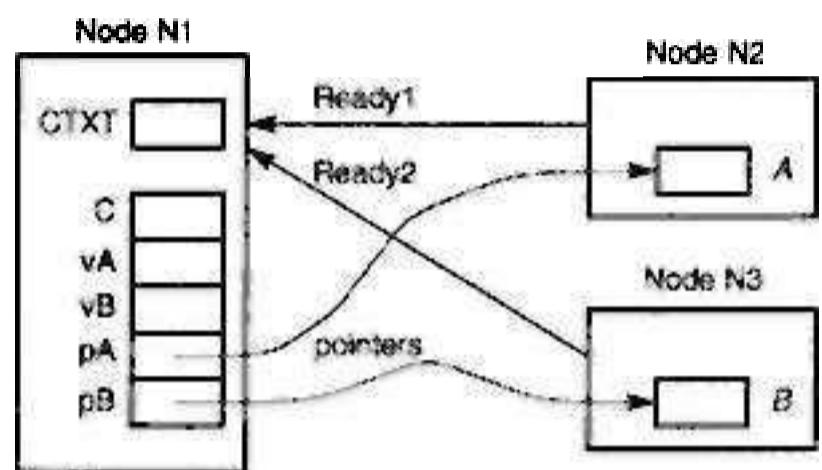
The remote load situation is illustrated in Fig. 9.12a. Variables A and B are located on nodes N2 and N3, respectively. They need to be brought to node N1 to compute the difference $A - B$ in variable C . The basic computation demands the execution of two remote loads (`rload`) and then the subtraction.



On **Node N1**, compute: $C = A - B$ demands to execute:

$$\begin{aligned} vA &= \text{rload } pA \\ vB &= \text{rload } pB \quad \} \text{"remote" loads} \\ C &= vA - vB \end{aligned}$$

(a) The remote loads problem



On Node N1, compute: $C = A - B$
 A and B computed concurrently
 Thread on **N1** must be notified when A, B are ready

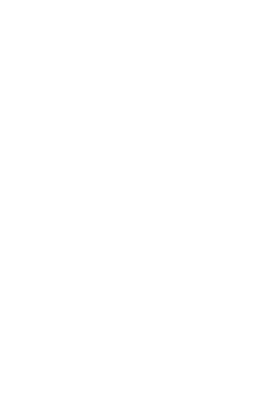
(b) The synchronizing loads problem

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

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

In Fig. 9.12b, the **idling** due to synchronizing loads is illustrated. In this case, A and B are computed by concurrent processes, and we are not sure exactly when they will be ready for node N1 to read. The ready signals (ready1 and ready2) may reach node N1 asynchronously. This is a typical situation in the producer-consumer problem. **Busy-waiting** may result.

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



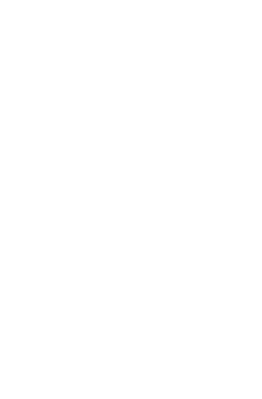
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

are the same, and the hierarchy collapses to three levels. In **general**, however, a request will travel through the interconnection network to the home cluster.

The home cluster can usually satisfy the request immediately, but if the directory entry is in a dirty state, or in a shared state when the requesting processor requests exclusive access, the fourth level must also be accessed. The *remote cluster* level for a memory block consists of the clusters marked by the directory as holding a copy of the block. It takes 135 processor clocks to access processor caches in remote clusters in the prototype design.

The Directory Protocol The directory memory relieves the processor caches of snooping on memory requests by keeping track of which caches hold each memory block. In the home node, there is a directory entry per block frame. Each entry contains one *presence bit* per processor cache. In addition, a *state bit* indicates whether the block is uncached, shared in **multiple** caches, or held exclusively by one cache (i.e., whether the block is dirty).

Using the state and presence bits, the memory can tell which caches need to be invalidated when a location is written. Likewise, the directory indicates whether memory's copy of the block is **up-to-date** or which cache holds the most recent copy. If the memory and directory are partitioned into independent units and connected to the processors by a scalable interconnect, the memory system can provide a scalable memory bandwidth.

By using the directory memory, a node writing a location can send point-to-point invalidation or update messages to those processors that are actually cacheing that block. This is in contrast to the invalidating broadcast required by the snoopy protocol. The scalability of the Dash depends on this ability to avoid broadcasts.

Another **important** attribute of the directory-based protocol is that it does not depend on any specific interconnection network topology. As a result, one can readily use any of the low-latency scalable networks, such as meshes or hypercubes, that were **originally** developed for **message-passing** machines.

Example 9.5 Cache coherence protocol using distributed directories in the Dash multiprocessor (Daniel Lenoski **and John Hennessy et al.**, 1992.)

Figure 9.26a illustrates the flow of a read request to remote memory with the directory in a dirty remote state. The read request is forwarded to the owning dirty cluster. The owning cluster sends out two messages in response to the read. A message containing the data is **sent** directly to the requesting cluster, and a sharing writeback request is sent to the home cluster. The sharing writeback request writes the cache block back to memory and also updates the directory.

This protocol reduces latency by permitting the dirty cluster to respond directly to the requesting cluster. In addition, this forwarding strategy allows the directory controller to simultaneously process many requests (i.e., to be multithreaded) without the added complexity of maintaining the state of outstanding requests. Serialization is reduced to the time of a single intercluster bus transaction. The only resource held while intercluster messages are being sent is a single entry **in** the



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

points on an **n -processor OMP**, where $N = n \cdot k$ for some integer $k > 2$. The idea of performing a **two-dimensional** FFT on an OMP is to perform a **one-dimensional** FFT along one dimension in a row-access mode.

All n processors then synchronize, switch to a column-access mode, and perform another one-dimensional FFT along the second dimension. First try the case where $N = 8$, $n = 4$, and $k = 2$ and then work out the general case for large $N \gg n$.

Problem 9.6 The following questions are related to shared virtual memory:

- (a) Why has shared virtual memory (SVM) become a necessity in building a scalable system with memories physically distributed over a large number of processing nodes?
- (b) What are the major differences in implementing SVM at the cache block level and the page level?

Problem 9.7 The release consistency (**RC**) model has combined the advantages of both the processor consistency (PC) and the weak consistency (WC) models. Answer the following questions related to these consistency models:

- (a) Compare the implementation requirements in the three consistency models.
- (b) Comment on the advantages and shortcomings of each consistency model.

Problem 9.8 Answer the following questions involving the MIT J-Machine:

- (a) What are the unique features of the **message-driven** processors (MDP) making it suitable for building **fine-grain** multicomputers?
- (b) Explain the **E-cube** routing mechanism built into the MDP.
- (c) What are the functions of the address arithmetic unit (**AAU**)?
- (d) Explain the concept of using a combining tree for synchronization of events on various nodes in the J-Machine.

Problem 9.9 Explain the following architectural trends in building multicomputers or scalable MPP systems:

- (a) Why are hypercube networks (binary n -cube networks), which were very popular in first-generation multicomputers, being replaced by 2D or 3D meshes or tori in the second and third generations of multicomputers?
- (b) Can you relate **first-level** wormhole routing to the asynchronous superpipelining practiced in processor designs? Identify their similarities and differences.

Problem 9.10 Answer the following questions on the SCI standards:

- (a) Explain the **sharing-list** creation and update methods used in the IEEE Scalable Coherence Interface (SCI) standard.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

OOP include the development of CAD (computer-aided design) tools and word processors with graphics capabilities.

Objects are program entities which encapsulate data and operations into single computational units. It turns out that concurrency is a natural consequence of the concept of objects. In fact, the concurrent use of coroutines in conventional programming is very similar to the concurrent manipulation of objects in OOP.

The development of *concurrent object-oriented programming* (COOP) provides an alternative model for concurrent computing on multiprocessors or on multicomputers. Various object models differ in the internal behavior of objects and in how they interact with each other.

An Actor Model COOP must support patterns of reuse and classification, for example, through the use of inheritance which allows all instances of a particular class to share the same property. An *actor model* developed at MIT is presented as one framework for COOP.

Actors are self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing. In an actor model, message passing is **attached** with semantics. Basic actor primitives **include**:

- (1) *Create*: Creating an actor from a behavior description and a set of parameters.
- (2) *Send-to*: Sending a message to another actor.
- (3) *Become*: An actor replacing its own behavior by a new behavior.

State changes are specified by behavior replacement. The replacement mechanism allows one to aggregate changes and to avoid unnecessary control-flow dependences. Concurrent computations are visualized in terms of concurrent actor creations, simultaneous communication events, and behavior replacements. Each message may cause an object (actor) to modify its state, create new objects, and send new messages.

Concurrency control structures represent particular patterns of message passing. The actor primitives provide a low-level description of concurrent systems. High-level constructs are also needed for raising the granularity of descriptions and for encapsulating faults. The actor model is particularly suitable for multicomputer **implementations**.

Parallelism in COOP Three common patterns of parallelism have been found in the practice of COOP. **First.** *pipeline concurrency* involves the overlapped enumeration of successive solutions and concurrent testing of the solutions as they emerge from an evaluation pipeline.

Second, *divide-and-conquer concurrency* involves the **concurrent** elaboration of different subprograms and the combining of their solutions to produce a solution to the overall **problem**. **In this case**, there is no interaction between the procedures solving the subproblems. These two patterns **are** illustrated by the following examples taken from the paper by Agha (1990).

Example 10.2 Concurrency **in** object-oriented programming (Gul Agha, 1990)



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

- Scalability — The language is scalable to the number of processors available and independent of hardware topology.
- Compatibility — The language is compatible with an established sequential language.
- Portability — The language is portable to **shared-memory** multiprocessors, message-passing **multicomputers**, or both.

Synchronization/Communication Features Listed below are desirable language features for synchronization or for communication purposes:

- **Single-assignment** languages
- Shared variables (locks) for IPC
- Logically shared memory such as the tuple space in Linda
- Send/receive for message passing
- Rendezvous in Ada
- Remote procedure call
- Dataflow languages such as Id
- Barriers, mailbox, semaphores, monitors

Control of Parallelism Listed below are features involving control constructs for specifying parallelism in various forms:

- Coarse, medium, or fine grain
- Explicit versus implicit parallelism
- Global parallelism in the entire program
- Loop parallelism in iterations
- **Task-split** parallelism
- Shared task queue
- **Divide-and-conquer** paradigm
- Shared abstract data types
- Task dependency specification

Data Parallelism Features Data parallelism is used to specify how data are accessed and distributed in either **SIMD** or **MIMD** computers.

- Run-time automatic decomposition — Data are automatically distributed with no user intervention as in Express.
- Mapping specification — Provides a facility for users to specify communication patterns or how data and processes are mapped onto the hardware, as in **DINO**.
- Virtual processor support — The compiler maps the virtual processors dynamically or statically onto the physical processors, as in PISCES 2 and DINO.
- Direct access to shared data — Shared data can be directly accessed without monitor **control**, as in Linda.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



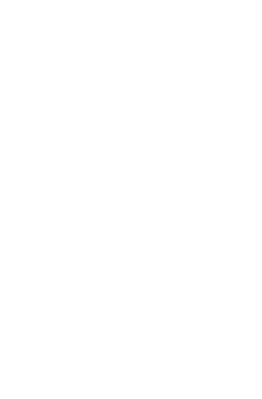
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



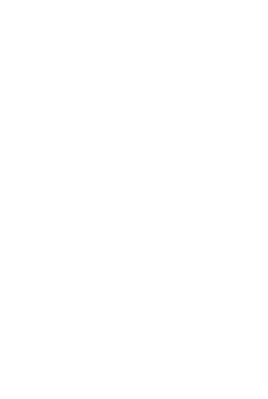
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.

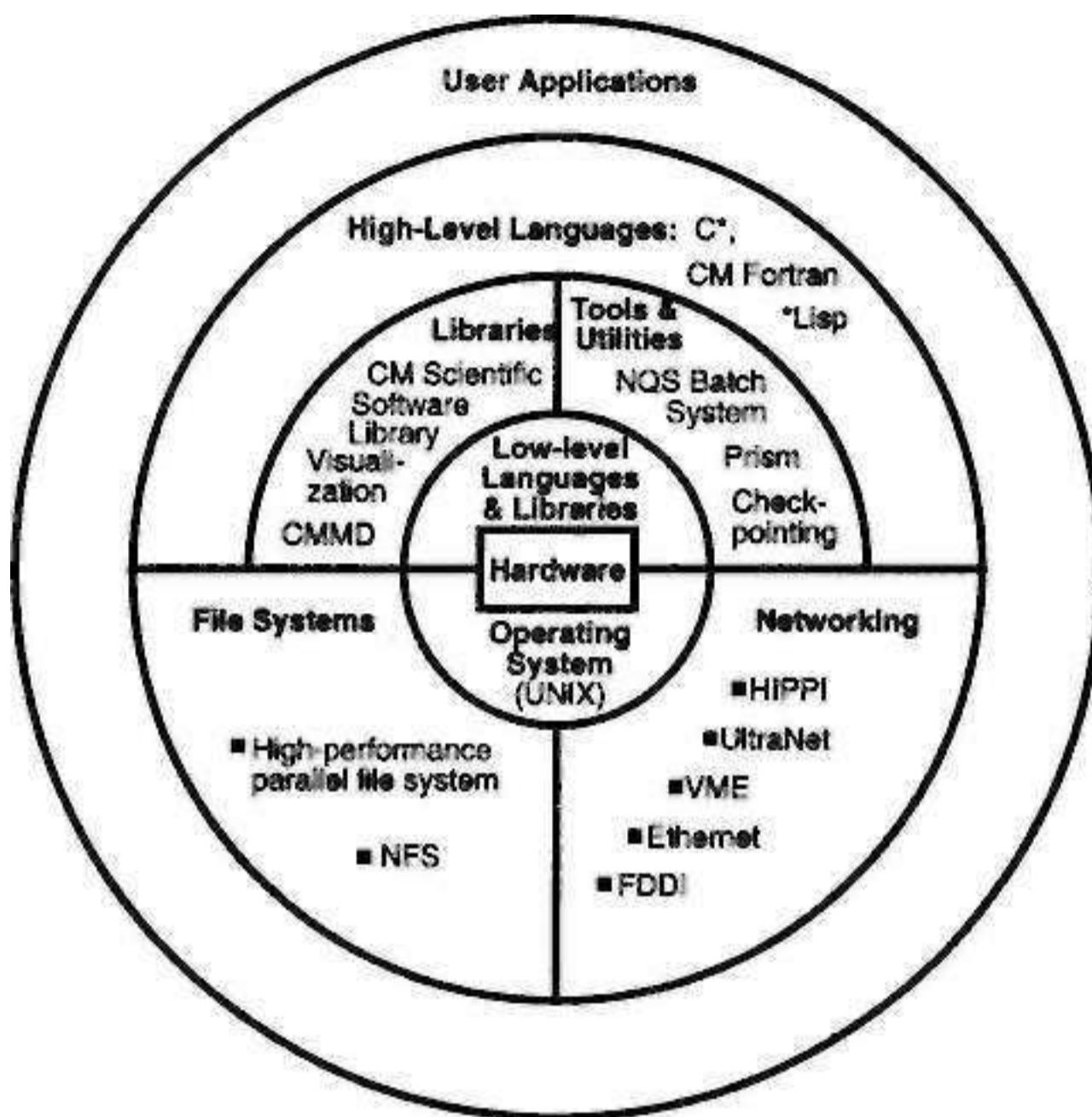


Figure 11.3 Software layers of the Connection Machine system. (Courtesy of Thinking Machines Corporation, 1992)

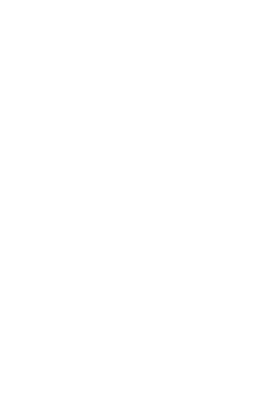
a FORALL statement and some additional **intrinsic functions**.

C* is an extension of the C programming language which supports data-parallel programming. Similarly, the *Lisp is an extension of the Common Lisp programming language for data-parallel programming. Both can be used to structure parallel data and to compute, communicate, and transfer data in parallel.

The CM Scientific Software Library includes linear algebra routines, fast Fourier transforms, random number **generators**, and **some** statistical analysis packages. Data visualization is aided by **an integrated environment including** the CMX11 library. The CMMD is a CM **message-passing** library permitting concurrent processing in which synchronization occurs only between matched sending and receiving nodes.

11.1.3 Visualization and Performance Tuning

The performance of a parallel computer can be enhanced at five processing stages: At **the** machine design stage, the architecture and OS should be optimized to yield high resource utilization and maximum system throughput. At the algorithm design/data



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Macrotasking When multitasking is conducted at the level of subroutine calls, it is called *macrotasking* with medium to coarse grains. Macrotasking has been implemented ever since the introduction of Cray **X-MP** systems. The concept of macrotasking is depicted in Fig. 11.4a.

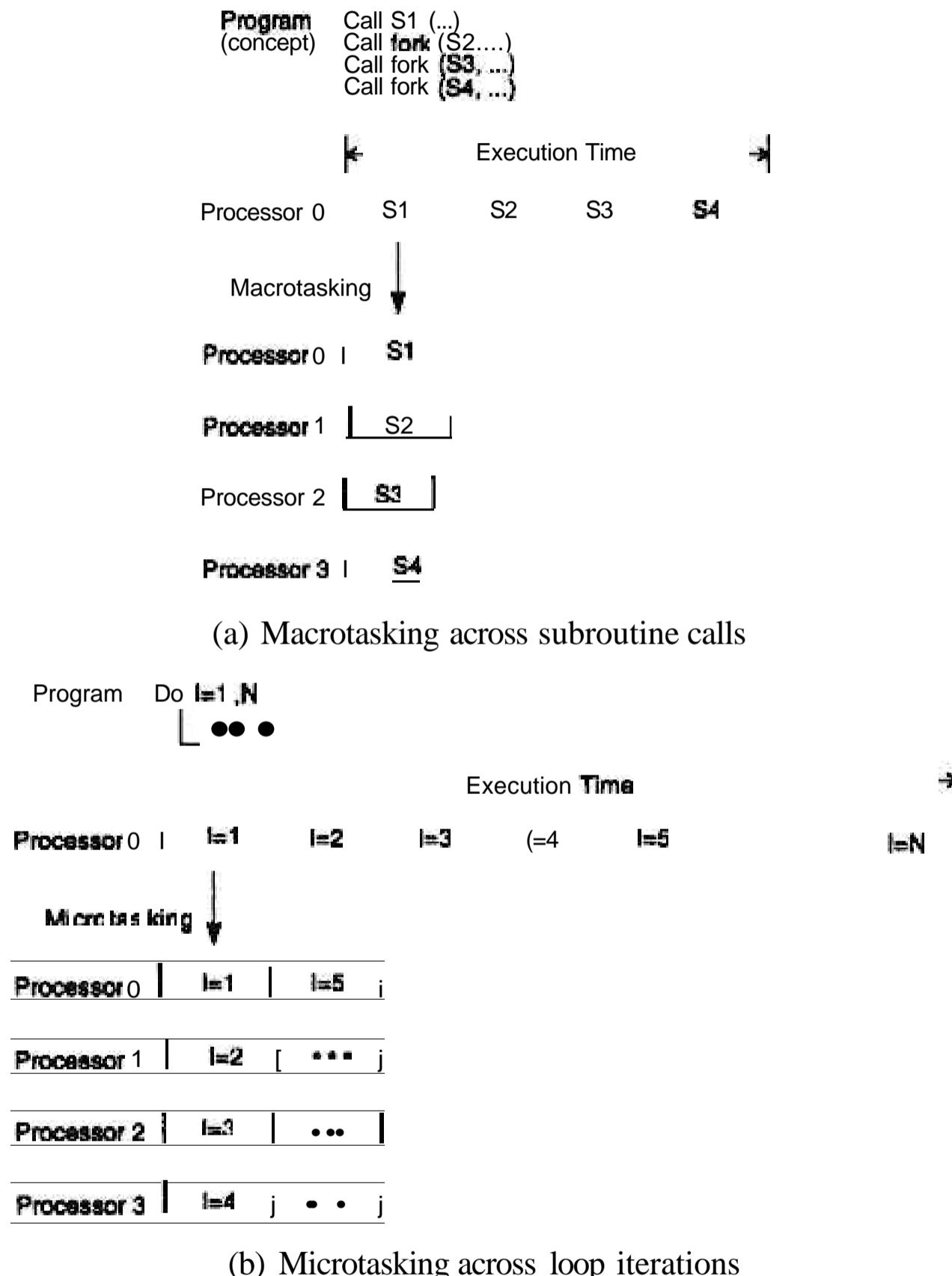


Figure 11.4 Multitasking at two different processing levels.

A **main** program calls a subroutine **S1** and then forks out three additional calls of the **same** subroutine. Macrotasking is best suited to programs with larger, longer-running tasks. The user interface with the Cray **X-MP** system's macrotasking capability



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



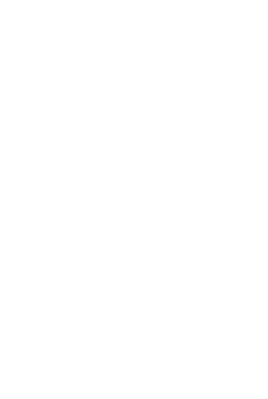
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

11.6 Bibliographic Notes and Exercises

The material on shared-variable synchronization and access order is based on [Bitar91] and [Bitar92]. Various synchronization methods were surveyed in [Dinning89]. [Graunke90] and Thakkar have compared synchronization algorithms on shared-memory multiprocessors.

Binary semaphores were originally proposed by [Dijkstra68]. [Hoare74] developed the use of *monitors* for operating system structuring. [Wirth77] introduced *Modula* for modular multiprogramming. [Perrott79] developed the *Actus* language for **array/vector** processing. [Brinch Hansen75] introduced the *Concurrent Pascal* language for shared-variable programming.

For message-passing programming, readers are referred to the *Cosmic C Programming Manual* by [Seitz89] et al. and the *Intel Parallel Programming Primer* edited by [Ragsdale90]. [Hoare74] developed CSP, which was later modified to *Occam* by [Pountain87] and May.

[Gelernter85a] proposed the *Linda* programming system for asynchronous communications. The material on domain, control, functional, and object decomposition techniques is based on Intel's iPSC experience. Additional programming examples using these techniques can be found in the Intel iPSC publication [Ragsdale90]. A parallel program for solving the *N-queens* problem can be found in [Athas88 and Seitz].

Surveys of parallel programming tools were reported in [Chang90 and Smith and in [Cheng91]. The MIMDizer was introduced in [Harrison90]. A user's guide for MIMDizer is also directly available from Pacific-Sierra Research Corporation [PSR90]. The Express is documented in [Parasoft90]. Linda was first introduced by [Gelernter85b] et al. C-Linda was discussed in [Ahuja86] et al., and SCHEDULE was reported by [Dongarra86] and Sorensen.

Cray Y-MP software overview can be found in [Cray89]. The Paragon programming environment is summarized in [Intel91]. The CM-5 software discussion is based on the technical summary [TMC91J]. Additional information on **CM-2** software can be found in [TMC90]. The supercompilers for iPSC/860 are described in [Intel90].

Exercises

Problem 11.1 Explain the following terms associated with fast and efficient synchronization schemes on a shared-memory multiprocessor:

- (a) Busy-wait versus sleep-wait protocols for sole access of a critical section.
- (b) Fairness policies for reviving one of the suspended processes waiting in a queue.
- (c) Lock mechanisms for presynchronization to achieve sole access to a critical section.
- (d) Optimistic concurrency or the postsynchronization method.
- (e) Server synchronization and the corresponding synchronization environment.

Problem 11.2 Distinguish between spin locks and suspend locks for sole access to



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the time required to conduct the sequential search. A sequential search involves backtracking once an impossible configuration is exposed. The backtracking step systematically removes the current emplacements of the queens and then continues with a new emplacement.

- (b) Develop a parallel program to run on a **message** passing multicomputer if one is available. For a concurrent search for solutions to the **N-queens** problem, backtracking is not necessary because all solutions are equally pursued. A detected impossible configuration is simply discarded by the node. Observe the dynamics of the concurrent search activities and record the total execution time. Compare the measured execution time data and comment on speedup gain and other performance issues.

Note that an 8-queens program written in an object-oriented programming language Cantor is given in the paper by Athas and Seitz (1988). Interested readers may want to compare their results with those reported by the Caltech researchers.

Problem 11.11 The *traveling salesperson problem* is to find the shortest route connecting a set of cities, visiting each city only once. The difficulty is that as the number of cities grows, the number of possible paths connecting them grows exponentially. In fact, $(n - 1)!/2$ paths are possible for n cities. A parallel program, based on *simulated annealing*, has been developed by Caltech researchers Felten, Karlin, and Otto [Felten85] for solving the problem for 64 cities grouped in 4 clusters of 16 each on a multicomputer.

- (a) Kallstrom and Thakkar (1988) have implemented the Caltech program in C language on an **iPSC/1** hypercube computer with 32 nodes. Study this C program for solving the traveling salesperson problem using a simulated annealing technique. Describe the concurrency opportunities in performing the large number of iterations (such as 60,000) per temperature drop.
- (b) Rerun the code on a modern **message-passing** multicomputer. Check the execution time and performance results and compare them with those reported by Kallstrom and Thakkar. Contact these authors for original codes. You may need to modify the code in order to run on a different machine.

Problem 11.12 Choose an example program to demonstrate the concepts of **macro-tasking**, **microtasking**, and autotasking on a Cray-like multiprocessor supercomputer. Perform a tradeoff study on the relative performance of the three multitasking schemes based on the example program execution.

Problem 11.13 Write a multitasked vectorized code in Fortran 90 for **matrix** multiplication using four processors with a shared memory. Assume square matrices of order $n = 4k$. The entire data set is available from the shared memory

Problem 11.14 Design a **message-passing** program for performing fast Fourier transform (FFT) over 1024 sample points on a **32-node** hypercube computer. Both host and



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

is to support uniform communication between the UNIX host and node processes. A set of daemon **processes**, utility programs, and libraries constitute the CE system.

Besides being able to interface with a single multicomputer, the CE can be used as a stand-alone system for running **message-passing** C programs on collections of network-connected UNIX hosts. It can also be used to handle the allocation of, and interfaces with, more than one multicomputer as depicted in Fig. 12.8.

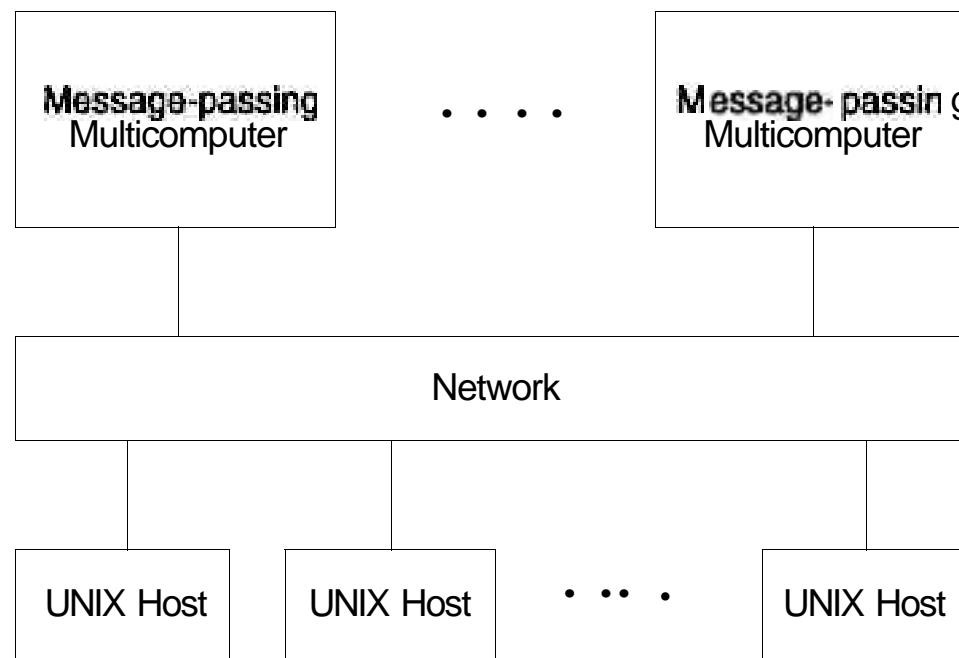


Figure 12.8 A **network** of UNIX hosts and several multicomputers.

Node Operating System The RK is a node operating system for a multicomputer. It was originally developed as the *Cosmic kernel* for first-generation Cosmic Cubes and later modified and ported to second-generation multicomputers like the **Symult Series 2010**. The same C programming environment is also supported on Intel **iPSC/1** and **iPSC/2** multicomputers with an NX node OS through the use of a compatibility library.

The difference between running message-passing C programs on UNIX hosts under the CE system RK and on multicomputer nodes is in their process creation functions. Every node runs a copy of the RK system. It supports multiprogramming within a single node. Based on 1991 standards, message passing is about **100** times faster in a multicomputer network than in a local area network.

The CE provides the same message-passing functions for UNIX processes that the RK provides for node processes. Thus the CE and the RK together provide an integrated **programming** environment in which uniform communication between host and node processes can be achieved independently of the physical allocation of these processes.

The entire set of processes created for a given computation is called a *process group*. The allocation of a process group and a set of multicomputer nodes is controlled by the user with a host facility called [getcube](#) in **first-generation** binary n-cube multicomputers.

The n-cube nodes are shared by recursive partitioning into **subcubes** of lower dimensions. A subcube is specified by its "cube" size as characterized by its dimension, $n = \log_2 N$, where N is the total number of nodes involved in subcube operations.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your limited preview limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

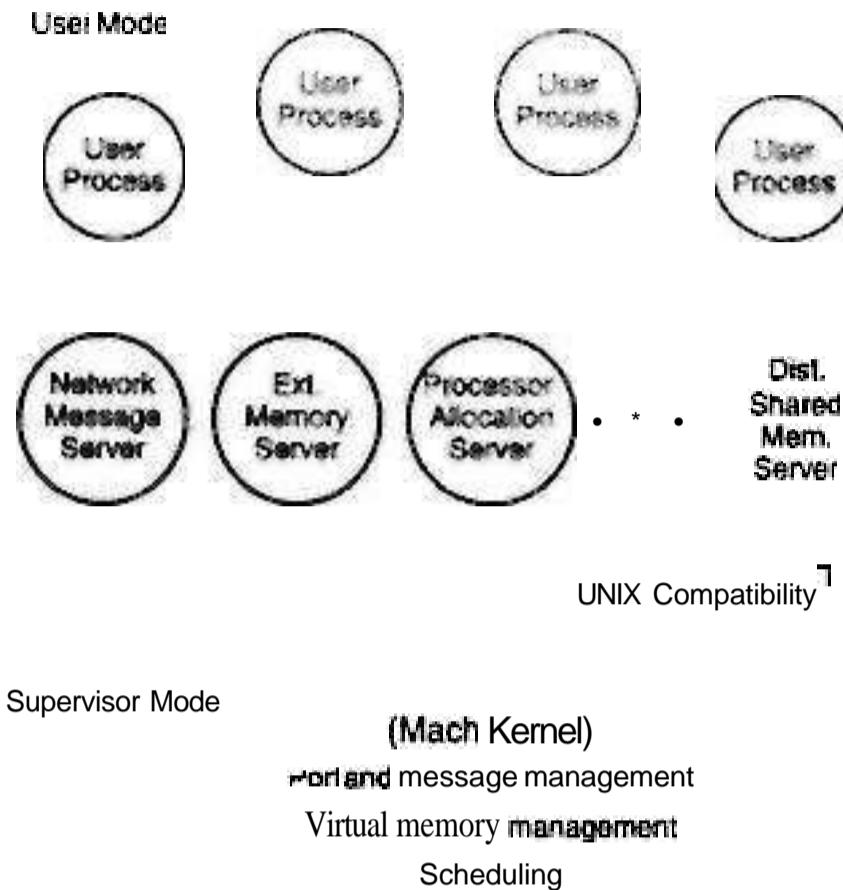


Figure 12.10 The Mach/OS kernel and various **user-level** servers with **UNIX compatibility**,

is a basic unit of resource **allocation**. A task includes protected access and control of all system **resources**, including CPUs, physical **I/O** ports, and either virtual or real **memory**.

A task address space uses a structured **map** of memory objects. The UNIX notion of a process is a task with a single thread of control. The task itself performs no computation; rather, it is a framework for running threads.

Threads A *thread* is the basic unit of CPU utilization. It is **equivalent** to a program stream with an independent program counter operating within a task. Usually, threads are lightweight processes that run within the environment defined by a task. A task may have multiple threads with all threads under the same **task-sharing** capabilities and resources.

As a matter of fact, the Mach kernel itself can be thought of as a task spawning multiple threads to do its work. Spawning of a new thread is much faster and more efficient than forking a traditional UNIX process because processes copy the parent's address space, whereas threads simply share the address space of the parent task. A traditional UNIX process can be **thought** of as a task with a single thread.

A thread can also be treated as an *object* capable of performing computations with low overhead **and minimal** state representation. In this sense, a task is generally a high-overhead object (much like a traditional UNIX process), whereas a thread is a relatively low-overhead object.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

An IPC interface generates **RPC code** for communication between a client and a server process. The IPC interface is language-independent. A *Mach interface generator* is used to generate procedures in C to *pack* and *send* or *receive* and *unpack* the IPC **messages** -

12.4.4 Virtual Memory Management

The Mach virtual memory design provides functions for managing a large, sparse virtual address space. Each Mach task can use up to 4 Gbytes of virtual memory for the execution of its threads. A restriction imposed by Mach is that memory regions must be aligned on system **page boundaries**. The size of a system page is defined at boot time and can be any multiple of the hardware page size.

Task Memory Allocation The *task address space* consists of a series of mappings of memory addresses to the task and its associated *memory objects*. The virtual memory design allows tasks to:

- Allocate or deallocate regions of virtual memory,
- Set protections on regions of virtual memory, and
- Specify the inheritance of regions of virtual memory.

Virtual Memory Operations Table 12.2 lists some typical virtual memory operations that can be performed on a task. Mach supports **copy-on-write** virtual copy operations, and **copy-on-write** and *read-write* memory sharing between tasks. It also supports *memory-mapped files* and user-provided **wacking-store objects and pagers**.

An example of **Mach's** virtual memory operations is the *fork* command in UNIX. When a *fork* is invoked, the child's address map is created based on the parent's address map's inheritance values. Inheritance may be specified as *shared*, *copy*, or *none* and **may** be specified on a **per-page** basis.

Shared pages are for *read* and *write* access by both the parent and the child. Copy pages are marked for copying onto the child's map. When a *write* request is made to the marked **address**, it is copied into the child's address space. A none page is not passed to the child at all.

Memory Mapping Mechanisms The Mach virtual memory management hardware has been implemented with minimal **machine** architecture dependence. The machine-dependent portion of the virtual memory subsystem consists of a single code module and its related header file.

The machine-dependent data structures contain mappings necessary for the execution of a specific program on a particular machine. Most other virtual memory management functions are maintained in machine-independent data structures. Several **important data structures** used are specified below.

- *Address maps* — A doubly linked list of map entries, each of which describes a mapping from a range of addresses to a region of virtual memory. There is a single address map associated with each task.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

making code sharing easier. In summary, **OSF/I** was designed to construct new applications that are **shareable**, dynamically linkable, position-independent, high performing, and fully compatible with existing **applications**. **OSF/I** encourages **distributing** applications through networking resources and activities.

12.5.3 improving Performance with Threads

Both Mach and OSF/I are highly motivated with multithreading or multitasking, as seen in previous sections. In this final subsection, we further elaborate on the use of threads to improve application performance. A thread implies a single sequential flow of control. Therefore, the conventional program execution follows a single **thread**. On a parallel computer or in a distributed system, an application may create multiple threads, all of which execute simultaneously.

Creating threads is relatively inexpensive in terms of execution time and OS resources. Therefore, modern systems encourage the use of a high degree of multithreading. One must realize that all threads within a task (or process) share the same memory-resident variables or use a common virtual address space. Threads are created to cooperate with each other in order to solve a large problem. Furthermore, data access must be synchronized. IEEE **POSIX** Standard P1003.4a is a standard for the behavior and programming interfaces relevant to **threads**, often referred to as the **Pthreads** for POSIX threads.

Basic Concept of Pthreads As noted in Chapter 9, even on a uniprocessor system, multiple threads may bring benefits in **performance**. Pthreads can cross-develop multiprocessor applications even on a uniprocessor. Many applications can be more efficiently executed with threads. In its simplest form, one can think of threads as **the ability** to apply many programs to solve a common problem. Each program has a small, self-contained job to perform. A more sophisticated approach is a single program being divided into many threads for parallel processing.

One example of the use of threads is to build a *multithreaded server*, which contains a simple main program which creates a thread upon each request. The threads can block. Each thread can wait for input on a separate input channel and then handle each service request. While some threads are blocked, others continue to execute. The requests are serviced simultaneously on multiprocessors. Other advantages of multithreading come from parallel processing, which overlaps computations, communications, and I/O activities.

A third motivation for threads is to handle asynchronous events more efficiently. This prevents inconvenient interrupts and avoids complex control flows. The following example shows some Pthreads codes in the main program and in a typical thread handling a request.

Example 12.10 Pthreads code for a multithreaded server (Open Software Foundation, 1990).

The Multithreaded server operates as an appointment manager keeping data in



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Index

- Accetta, M., 687, 695, 698, 712
Acosta, R. D., 322
Active Memory Technology (AMT)
 DAP600 Series, [33](#), [83](#), 368, [555](#)
 DAP610, [32](#), [44](#), 121, 447, 448
Actor model, [552](#)
Actus, [661](#)
Ada, 13, 646, 657
Adaptive routing, [375](#), 382, 384, 387, 392, 394, 401
Address arithmetic unit (**AAU**), 507, 512
Address translation cache (ATC), [167](#), 169
Address-only transfer, 221
Addressing mode, 162-164, 166, [167](#), [208](#), 209
Advanced Micro Device (AMD) [29000](#), [170](#), 171, 315
Adve, S. V., 256, 257
Agarwal, A., [23](#), [44](#), 141, [142](#), 149, [500](#), [539](#)
Agha, G., [557](#), 558, 612
Aho, A., 149
Ahuja, S., 661
Aiken, H., [4](#)
Alewife, [23](#), [44](#), 475, 494, 501, [538](#), 539
Algebraic optimization, 581, 583
Algol, [5](#)
Algorithm, [7](#), [33](#), [34](#)
 CRCW, [37](#)
 deterministic, [34](#)
 EREW, [37](#)
 nondeterministic, [34](#)
 nonpolynomial, [34](#)
 NP-complete, [315](#)
 parallel, [7](#), [8](#), [32](#), [34](#), [35](#), [38](#), [49](#)
 polynomial-complexity, [34](#).
Allan, S. J., 612
ALLCACHE, 522, 523
Allen, J. R., 567, 612
Allen, M., 322
Alliant Computer Systems Corporation
 FX, 355
 FX Fortran compiler, [560](#), 567, [619](#)
 FX/2800, 347
 FX/2800-200, 115
 FX/80, [23](#), 589
Almasi, G. S., [46](#)
ALU dependence, [54](#)
Alverson, R., [44](#), 539
Amdahl Corporation
 470/V6, 234
 470/V7, 234
 470V/8, 280
Amdahl's law, 105, [111](#), 112, 124, [129](#)-[134](#), [136](#), 149, 151
Amdahl, G., 131, 149
Ametek 2010, 371, 394
Ametek S/14, 368
Analytical modeling, 236
Anaratone, M., [84](#), 96
Anderson, D. W., 322
Andrews, G. R., 612
Antidependence, 52, 289, [312](#), 315
A PAL, [33](#)
Apollo workstation, 476
Arbiter, [215](#)
Arbitration competition, 218
Archibald, J., 257
Architecture-independent language, [9](#)
Architecture-neutral distribution format (ANDF), 702
Arden, B. W., 96
Ardent Computer Corporation, [432](#)
Arithmetic and logic unit (ALU), [161](#), [167](#), 174, 177, 281, 299, 318, 507
Arithmetic mean, 108
Array processing language, 555
Array processor, 120, 555
Artificial intelligence (**AI**), [7](#), [13](#), [46](#), 157,

- 161, 187**, 559, 560, **612**, 652, 659
 Artificial neural network, **187**
Arvind, 45, 71, **96**, 531-534, 536, 537, **539**, 540
Assembler, 8
Assembly language, 5, 645
Associative memory, **11**, 232
Associative processor, **11**
Asynchronous bus timing, 217
Asynchrony problem, 492, 494
Athas, W. C., **46**, **369-370**, **394**, **fifil**
Atomic memory operation, **345**, 347, 551
Atomic operation, 549, 625, 627, 636, 637, 640, 662
Atomicity, 248, **249**, 253, 255, 260
Attached processor, 20, 161, **162**, **185**
Autotasking, **427**, 621, 631
Available processor set, **678**, 679
Average latency, 273, 274, **276**, 277
- Babb, R. G.**, 612
Babbage, C., **4**
Bach, M. J., 712
Back-end processor, 162
Backplane bus, 213-224
 effective bandwidth, 213
Baer, J. L., 257
Balbo, G., 393
 Banerjee's inequalities, 567, 570, 578
Banerjee, U., 570, **612**
Banyan network, **337**
Barrel shifter, **82**
Barrier, **364-366**
 Barrier synchronization, 365, 366, 394
Base scalar processor, **159**, 165, 169, 170, 178, 210, **320**, 325
Baseline network, 93, 96, **102**, 336, **33Z**
Bashkow, T. R., 384, 394
Basic block, 566, **579-581**, 583, 588, 589, 592, 594, 596, 597
Batch processing, 411, **412**, **621**, 622
Batcher, K. E., 96
BBN Advanced Computers, Inc.
 . Butterfly, 368, 383, 482, 617, 686, 701
 Butterfly network, **42**
TC-2000, 5, 6, **22-24**, 95, 121, 347
Beetem, M., **44**
Bell, C. G., **27**, 41, **46**, **94**, 96, 143, **145**, 149, **432**, **491**, **523**, **539**
- Ben-Ari, M.**, 612
Benes network, 338
Benes, V. E., 338
Berkeley 4.3BSD, 622, 687
Berkeley 4.3BSD, 621
Berkeley RISC, 60, **171-173**, **208**
Berkeley UNIX 4.3BSD, 688
Berkeley UNIX 4.4 BSD, 671, 702, 704, **705**, 708
Bernstein's conditions, **54-57**
Bernstein, A. J., **54**, 96
Berntsen, J., 149
Best-effort Mflops, 435
Bhandarkar, D. P., 469
Bic, L., 75, 96, 539
Billion floating-point instructions per second (Gflops), **26**, **30**, **33**, **122**, **143**
Billion instructions per second (GIPS), **26**, **33**
Binary n -cube, **86**, 87, 103
Binary n -cube, **84**, 392
 see also Hypercube
Binary lock, 634
Bisection bandwidth, **76-77**, 80, 89, 527, 537
Bisection communication bound, **39**
Bisiani, R., 479, 539
BIT IU B-3100, **171**
Bit-slice, **32**, **44**, 120
Bitar, P., 625, **661**
Black, D., 712, 713
Blelloch, G. E., 469
Blevins, D. W., 469
BLITZEN architecture, 455, 469
Blocking flow control, 383, 509, 511
Blocking network, 336-338, 342
Board interface logic, **215**
Borkar, S., 394
Brainerd, W. S., 613
Branch folding, 297
Branch handling technique, **265**, **280**, 282, 592, 595
Branch target buffer (BTB), 294
Branch prediction, 60, 184, **292**, **293**, 315
 dynamic, 293, 294
 static, 293
Brawer, S., **612**
Bridge, 335

- Briggs, F. A., [46](#), [110](#), 149, [208](#), 251, [256](#), 348, [349](#), 351, 393, 612
- Brinch Hansen**, P., 643, [661](#)
- Broadcast**, 216, 222
- Broadcast, [26](#), [31](#), [32](#), 49, [63](#), [78](#), 80, [92](#), 101, 214, [216](#), [222](#), 224, 343, [345](#), [351](#), [353](#), [355](#), [357](#), [358](#), [375](#), [388](#)-391, [400](#), [447](#), [449](#), [453](#), 455, 457, [461](#), [464](#), [465](#), [470](#), [473](#)
- Brunner**, R. A., 469
- Bubble sort, 580, 581, 583, 592
- Buffered crossbar network (BCN), 442
- Buffering flow control, 383
- Bulldog, 60
- Burkhardt**, H., [23](#)
- Burroughs Scientific Processor (BSP), [44](#)
- Bus addressing, 216
- Bus arbitration, [215](#)-[221](#), [223](#), 224, 257
- Bus cycle, [216](#), 221
- Bus system, [89](#)-[90](#), 257, [333](#), 334, 340, 393 hierarchy, [333](#)
- Bus tenure, 218
- Bus timer, [215](#)
- Bus transaction, 218
- Bus-connected system, 348
- Busy waiting**, [364](#), [493](#), 538
- Busy-wait protocol, *see* Wait protocol
- Butler, M., [61](#), 149
- Butterfly, 91
- Butterfly network, [341](#), 343, [345](#), 347, 394
- Butterfly switch, [23](#)
- Buzbee, B., [403](#)
- C**, 13, [18](#), [19](#), [30](#), [618](#)-[621](#), 623, 644, 645, [665](#)
- C***, 618, [623](#)
- C++**, [514](#), 657
- C+-** 514
- C-acceas memory, 240, [408](#)
- C.mmp**, 41, 49, [94](#), 96
- C/S-access** memory, [410](#)
- C2/B1 security system, [703](#)-[704](#)
- Cache, 224-238
- k-way** associative, 232
 - access time, 261, [263](#)
 - addressing model, 213, 225
 - aliasing problem, 225-228
 - associative mapping, 228
- block, 174, 192, [195](#), [201](#), [207](#), 225, [228](#), 230, 232, 237, 261
- block frame, [207](#), 228-230, [232](#), 234, [260](#)-[262](#)
- block replacement, 225, 229, [231](#), 232, 234, 236
- block size, [227](#), 232, 236-238, [357](#)
- capacity, [236](#), [238](#)
- coherence, [216](#), [221](#), 236, 257
- coherence protocol, 257
- cycle count, 236
- data (D-cache), [163](#), 164, 167, 174, 177, 182, 209, 226, [227](#), 234, 238, [261](#), [262](#), [281](#), [316](#)
- design parameter, 230
- direct-mapping, [228](#)-[230](#), [232](#), 260
- directory, 23, 351, 358
- directory-based** coherence, 257, [259](#)
- flush, [226](#), 228, 363
- fully associative mapping, 230, 232, 234, 260
- hit, [47](#), [195](#), 225, 230, 234
- hit ratio, [167](#), 196, [208](#), [211](#), 212, 228, [230](#), [231](#), 233, 236-238, [259](#), 261, 483
- implementation scheme, 213
- instruction (**I-cache**), [60](#), [163](#), [164](#), [167](#), [174](#), [177](#), [179](#), 182, 226, [227](#), 234, [238](#), [261](#)-[263](#), [281](#), [310](#), [315](#)
- line, 192, 235
- lookup, 225, [227](#)
- miss, [47](#), 159, [195](#), 207, 225, 230, 234, 262, [349](#), [357](#), 396
- multilevel, 226, 238
- performance, 224, 228, 236, 237
- physical address, [224](#)-[227](#)
- private, 225, 257-259, 331, 334, [348](#), 349, 351, 357, 363, 490
- sector mapping, 224, 234-236, 260
- set number, 232, 236, 260
- set size, 232, 261
- set-associative mapping, 224, [227](#), 232-235, 238, 396, 397
- shared, [357](#), 363
- size, [232](#), 237, 238
- speed, 236
- split**, 164, [208](#)
- unified, [163](#), 166, [208](#), 225, 355
- virtual address, 224-227

write-back (WB), 225-228, 236, 238, 334, 348, 351, 353, 355, 395, 476
write-through (WT), 225, 236, 238, 258, 348, 351, 353, 355, 476, 502
 Cache agent, 356
Cache-coherent nonuniform-memory-access model (CC-NUMA), 23
Cache-only memory architecture (COMA), 19, 23, 42, 46, 48, 331, 521
 Callahan, D., 612
 Carriero, N., 479
 Carry-propagation adder (CPA), 300, 302, 303
 Carry-save adder (CSA), 300, 302, 303, 322, 327
 Caswell, D., 712, 713
 Cedar, 22, 41, 341, 347, 421, 628
Cekleov, M.. 248, 252, 256, 257
Censier, L. M., 257, 358
 Central arbitration, 218-219
 Central directory, 358
 Central processing unit (CPU), 5
Chaiken, D., 257, 539
 Chained-directory coherence protocol, 486
Chang, A., 208
Chang, L. C., 661
 Channel bisection width, 77
 Channel width, 77, 83, 88, 89
 Channel-addressed model, 683, 714
 Channel-addressing scheme, 647
 Channel-dependence graph, 380, 381
Chapman, B., 612
Chen, S., 411
Cheng, D. Y., 661
Cheng, H., 469
 Chin, C. Y., 393
Chordal ring, 80-82, 84, 96
Chow, C. K., 208
 Christy, P., 612
 Circuit switching, 332
 Circular FIFO replacement algorithm, 206, 207, 212
 CISC processor, 157, 158, 162-167, 169, 170, 177, 178, 209, 280, 283
 Clark, K. L., 612
 Clark, R. S., 469
 Clock cycle, 267, 268, 271, 272, 279, 281, 290, 321-324, 326

Clock rate, 14-16, 47, 157-160, 163, 164, 167, 170, 174, 176, 177, 180, 182
 Clock skew, 267, 268, 318, 327
Clos network, 338
Clos, C., 338
 Cluster shared memory, 22
CM-5
 control network, 457-462, 465, 466
 data network, 457-462, 466
 diagnostic network, 457, 459, 460, 462, 471
Coarse-grain supercomputer, 506
 Cobegin, 563-564
Cobol, 5
 Cocke, J., 117, 149
 Code compaction, 182-184, 596, 597
 Code generation, 564, 566, 567, 578, 585, 592, 595, 596
 Code hoisting, 584
 Code optimization, 60, 578
 Code optimizer, 60
 Code **parallelization**, 562, 589, 591, 620
 Code scheduling, 60, 579
Coend, 563, 564
Cohen, D., 516
 Coherent cache, 475, 482-487, 516, 518, 538
Colley, S., 394
Collision vector, 274-277, 279, 284, 323, 327
 Combining network, 338, 341, 345, 347, 394, 398
 Combining tree, 511-513
 Common bus, 19, 24
 Communicating sequential process (CSP), 646
 Communication bandwidth, 455, 460, 492, 516
 Communication latency, 60, 61, 63-64, 68, 74, 88, 105, 106, 108, 125, 143, 146, 147, 368, 370, 371, 378, 382, 388, 400, 491, 493, 505, 506
 Communication overhead, 61, 148, 372, 491, 505, 552, 625, 648
Communication/computation ratio, 645, 652, 656
Compare&Swap, 364, 636
 Compatibility, 670, 682, 684, 686-688, 714
 Compelled data transfer, 221

- Compensation code, 596-598
 Compiler
 intelligent, 151
 optimizing, 8, 60, 61, 148, 185, 208, 315, 525, 556, 566, 567, 585, 619
 parallelizing, 9, 18, 30, 131, 178, 564, 567, 570, 578, 585, 619
 sequential, 9
 vectorizing, 9, 30, 62, 63, 150, 404, 412, 413, 469, 585
 Compiler directive, 9
 Compiler preprocessor, 566, 619
 Compiler support, 525, 531
 Complementary MOS (CMOS), 168, 170, 171-129
 Complex instruction-set computer (**CISC**), 17, 26, 158, 166, 668
 Complexity
 hardware, 80, 96, 101
 space, 51
 time, 51, 64
 Component counter, 441
 Compound vector function (**CVF**), 435-437, 439, 442, 444-446, 471
 Computation model, 51
 Computational tractability, 403
Compute-bound, 228
 Computer-aided design (CAD), 120
 Computing capacity, 106
 Computing granularity, 492, 505
 Concurrent object-oriented programming (COOP), 556, 552
 Concurrent outer/vector inner (**COVI**), 589-591
 Concurrent Pascal, 618, 643, 661
 Concurrent Prolog, 559, 612
 Concurrent Smalltalk, 514
 Conference communication, 63, 388, 400
 Connected transaction, 221, 355, 356
 Constant cycle, 274, 325
 Conte, G., 393
 Content-addressable memory (CAM), *see* Associative memory
 Contention bus, 89
 Context switching, 228, 496, 498, 499, 510, 531, 491, 499, 514, 529, 668, 672, 676, 682, 686, 691-693
 overhead, 491, 494
 policy, 498
 Continuation name space, 494
Continuation stack, 538
 Control Data Corporation
 6600/7600, 5, fi
 1604, 5, 6
 6400, 44
 6600, 291, 306, 322, 525
 7600, 44, 306, 316, 411
 Cyber 205, 29, 44, 412
 Cyberplus, 82, 501, 539
 ETA, 410, 412
 ETA10, 44
 STACKLIB, 60
 STAR-100, 412
 Control decomposition, 652, 664
 Control dependence, 53-54, 57, 564, 567, 578, 579, 626
 Control flow, 491
 Control memory, 164
 Control token, 75
 Control unit (CU), 28, 31, 32
 scalar, 28
 vector, 28
 Control-driven, 71, 76
 Control-flow computer, 71-73, 75
 Control-flow optimization, 584
 Convergence division, 304, 307
 Convex Computer Corporation
 C1, 30
 C2, 30
 C3, 30
 C3240, 114, 115
 Cooperative problem solving, 558
 Coprocessor, 161, 162, 171, 184, 208
Copy-on-write, 228, 669, 688, 690, 697, 698, 700
Cormen, T. If. 46
 Cosmic cube, 25, 368, 394, 514, 645, 665, 667, 684
 Cosmic environment, 644, 645, 647, 683, 684, 714
 Cosmic kernel, 644
 Coverage factor, 481
 CPU time, 14-17
CPU-bound, 134
 Cragon, H. G., 208, 322
 Crawford, J. H., 208
 Cray Research, Inc.
 assembler CAL, 621

- C-90**, [422-424](#)
 CFT, 60
 CFT compiler, 567
 Cray [1](#), [29](#), [44](#), [316](#), [410](#), [411](#), [438](#), [471](#)
[Cray 1S](#), [410](#), [411](#)
[Cray 2](#), [412](#), [417](#), [421](#), 620
[Cray 2S](#), [412](#)
 MPP, [5](#), [6](#), [44](#), 423-424
 T3D, 423, 424
 X-MP, [5](#), [6](#), [411](#), 438, 439, 469, [471](#),
 591, 620, 630, 632, 633
Y-MP, [29](#), [30](#), [44](#), 95, [115](#), 143, 145,
 162, 347, [406](#), [411](#), [412](#), [417](#), [419](#),
 421, [422](#), [424](#), 440, [469](#), [472](#), 505,
 617, 621, 628, [630](#), [633](#), [661](#)
- Cray, S. [410](#)
 Critical section (**CS**), 548, 549, 629, 634-
[637](#), [640](#), [641](#), [661](#), [662](#)
 Crossbar switch network, 76, [78](#), [101](#), 89,
[94-96](#), 336, 339, 340, 347, 380,
 395, 421, [429](#), [432](#), 455, 486, 534
 Crossing dependence, 577
Crosspoint switch, 338, 339, 395
 Cube-connected cycle (CCC), [26](#), [85-86](#),
 89, [96](#), 100
 Culler, D. E., 494, 534, 539
 Cybenko, G., [143](#)
 Cycle scheduling, 594
 Cycle time, [14](#), [15](#)
 Cycles per instruction (**CPI**), [14-17](#), [47](#), [48](#),
 115, [150](#), [152](#), 157-160, [163](#), [164](#),
[167](#), 170, 177, 180, 182, 209
 Cypress
 CY7C157, [171](#)
 CY7C601, 170-172
 CY7C602, [170-172](#)
 CY7C604, [170](#), [171](#)
- DADO, 83
 Daisy chain, 218, [219](#)
 Dally, W. J., [19](#), [43](#), 87, 88, 96, [103](#), 394,
 507, 511, 514, 539
 Dash, [23](#), [42](#), 49, 475, 476, [478](#), 480-482,
 489, 494, 501, 516-521, 538, 539
 Data dependence, 51-53, 57, [268](#), [280-282](#),
[285](#), [288](#), 290, 291, [297](#), [305](#), 306,
 310, [312](#), 313, 322, 564, 568, 570,
[579](#), 588, 592, 626
 Data Diffusion Machine (DDM), [23](#)
- Data parallelism 120
 Data pollution point, [357](#)
 Data structure, [7](#), [8](#), [14](#), 72, 76, [187](#), 629,
 643, 644, 648, 649, 652, 654, 656
 dynamic, 649
 shared, 625, 657
 static, 648, 649
 Data token, [71](#), [72](#)
 Data transfer bus (DTB), 214, [215](#), 221
Data-driven, [70](#), 71, [74-76](#)
Data-parallel programming model, [554-556](#),
 612, 623
Data-routing function, 76-78
 Database management, [7](#), [13](#), [26](#), 117, [118](#),
 120
 Dataflow architecture, 534, 539
 Dataflow computer, [70-72](#), 75, 475, 531-
 534, 536, [540](#), [559](#), 617
 dynamic, [45](#), 533, 540
 static, [45](#), 533, 540
 Dataflow graph, 72, [74](#), [531-534](#), 559, 560
 Dataflow language, 559, [561](#)
 Dataflow multithreading, 491
 Davis, E. W., 455, 469
 Davison, E. S., 274, [277](#), 322
 Deadlock, [222](#), [375](#), [379-382](#), [388](#), [391](#), [392](#),
 548, 626, [638-640](#), 663, 664
 buffer, 375, 380, 399
 channel, 375, 380, 399
 static prevention, 640
 Deadlock avoidance, [26](#), [375](#), [387](#), [401](#), 549,
 640
 dynamic, 640
 Deadlock recovery, 640
 Deadlock-free allocation, 640
 Debit credit benchmark, 118
 Degree of chaining, 439, 441, 442
 Degree of parallelism (DOP), 291
 Degree of superscalar processor, [159](#), 178,
 180, 210
 DeGroot, D., [46](#), 612
Dekel, E., 149
Dekker's protocol, 635-637, 662
 Delay insertion, 279
 Delay slot, [292](#), 295-297, [312](#), 319
 Delayed branch, [292](#), [295-297](#), [312](#)
 Delta network, [337](#), 393, 398
Demand-driven, [70](#), 71, [74-76](#)

Denelcor HEP, [44](#), 383, 494, **516**, **525**, 529, 531, 538, 539
Denning, P. J., 194, [208](#)
Dennis, J., [45](#), 533, 540
 Dependence analysis, 564, 568, 570, 585
 Dependence equation, 569, 570, 576, 577
 Dependence graph, 51, 52, 55, [64](#), 72
 Dependence testing, 567-570, **573**, **574**, **578**
 Deterministic routing, 375, 382, 384
 Detour flow control, 383
Dhrystone, [115](#), [116](#), [149](#)
Diefendorff, K., 322
 Digital Equipment Corporation
[21064](#), [179](#), [318](#)
 VAX [8600](#), 225
 Alpha, [122](#), 145, [179](#), [208](#), [317-319](#), 322, 424
 Firefly, 355
 PDP [11](#), [94](#)
PDP 11/40, 41
PDP-11, 246
PDP-8, 5, fi
 VAX, [33](#), [143](#), 674, 698-701, 712
 VAX [11/780](#), [16](#), [17](#), **115-117** [234](#), [429](#), [674](#)
 VAX [9000](#), [5](#), [6](#), [30](#), [118](#), [167](#), [177](#), [429](#), [430](#), 469
 VAX/8600, **158**, **165-167**
 VAX/8800, 230
 Digital logic simulation, 481
 Digital signal processor (DSP), 161
Dijkstra, E. W., [637](#), [661](#), 662
 Dimension-order routing, 375, 384, 509
 Dinning, A., [g61](#)
DINO, [560-562](#), 619
 Direct memory access (DMA), 226
 Directed acyclic graph (DAG), 592, 593, 595
 Direction vector, **570-574**, 576, 578, 586, 607
Directory-based cache coherence protocol, [348](#), [350](#), [357-358](#), 476, [478](#), [494](#), [519](#), 539
 Discard flow control, 383
 Discouragement hint, 694, 713
 Disk drive, 189
 Distance vector, 570, 571, 573, 586, 587, 601, 602, 605, 607
 Distributed arbitration, 220-221

Distributed cache, [491](#), 494, 538
 Distributed cache coherence protocol, 487
 Distributed computing environment (DCE), 702
 Distributed directory, 358
Distributed management environment (DME), 702
 Distributed memory, [6](#), [19](#), [22](#), [27](#), 48
 Distributed **shared** memory, [23](#), [27](#), 143, 475, 476, 486, 487, 500
Divide-and-conquer, 107, [557](#), 613, 654
 Domain, 55, 289
 Domain decomposition, 553, **648-650**, 652, 659, 660
 Dongarra, J., [114](#), [115](#), 149, [429](#), [433](#), 434, 469, 620, [661](#)
 DSB memory model, **254-256**, [263](#), 488
 Dubois, M., [46](#), 208, 251, 256, 257, 348, 349, 351, 393, 487
 Dynamic graphical animation, 620
 Dynamic instruction scheduling, 60, 288-291, 305, 315, 578-579
E-cube routing, 384, **385**, **394**, **401**, 509, [542](#)
 Eager evaluation, [74](#), 75
 Eager, D. L., 149
Earle, J. G., 322
Edenfield, R., [208](#)
 Effective MIPS rate, 150
 Efficiency, [8](#), [9](#), [21](#), [105](#), 112-114, 116, 119, [122-126](#), 141, [142](#), 147, 149, 150
Eicken, 539
 Elementary function, 299
 Emitter-coupled logic (**ECL**), [170-171](#), [174](#), 190
 Emma, P. G., 322
 Encore Computer Corporation
 Multimax, 95, 701
Multimax series, 334, [411](#)
Ultramax, 335
ENIAC, [5](#)
Enslow, P. H., 393
 Error checking and correction, 507
 Error-correcting code (ECC), 450, 464
 Error-correcting memory, [6](#)
 ETL (Japan)
[EM-4](#) 475, 531, 534-536, 538, 540
[EM-4+](#). 536
[EM-5](#), 45, 536

Sigma [1, 45](#), 533, 534, 540
 Event ordering, 248, 251, 257
 Event-based elevation, 690
 Exact test, 569, [575](#), 576
 Exception handling, 711, 712, 714
 Exchange, [78](#)
 Execution unit, 283, [284](#), 305
 Explicit-dependence **lookahead**, 525, 530
 Express, 617
 Expression tree, 595

 Fairness policy, 625, 626, 636, **fifcl**
 Fast Fourier transform (FFT), 614
 Fastbus, 222, 393, 487
 Fat tree, 83, 85, 88, 96, 103, [460](#), 462, 469, 471
 universal, 103
 Fault tolerance, 339, 676
 Fault-tolerant system, 96, 671
 FAUST, 617
 Feautrier, P., 257, 358
Felten, E., 612
 Feng, T. Y., 92, [96](#), [337](#), 393
Ferrante, M. W., 539
Fetch&Add, [345-346](#), 398, [551](#), 636
 Fetch-match unit, 536
 FFT
 Cooley-Tukey algorithm, 128
Fier, J., 394
Fifth-Generation Computer System Project
 (FGCS), [118](#), 149, 559, [560](#), 612
 File server, 669
 File subsystem, 668
Fine-grain multicomputer 475
Fine-grain program, 507
 First-in-first-out (FIFO) replacement algorithm, 206, [207](#)
 Fisher J. A., 149, [208](#), 612, 613
 Fixed master kernel, 676
 Fixed scheduling, 693
 Fixed-load speedup, 124, [130-132](#)
 Fixed-memory speedup, [124](#), 136-138
 Fixed-point operation, 297-298
 Fixed-point unit (FXU), [179](#), [181](#)
 Fixed-time speedup, [124](#), [133](#), [134](#), [136-138](#)
 Flip network, 96, [102](#), 336
 Floating-executive kernel, [667](#), [672](#), [674](#), [676](#), [677](#), 682, 712
 Floating-point accelerator, [23](#), [32](#), [161](#), [176](#)

Floating-point arithmetic processor, [5](#)
 Floating-point capability, 115
 Floating-point number, 298-299
 Floating-point operation, 299
 Floating-point unit (FPU), [54](#), [122](#), 170-172, [174](#), [179-182](#), 297, 299, [300](#), [303](#), [305](#), [372](#), [373](#)
Flow analysis, 564, 565
 Flow control digit (Hit), 375, 376, 378-381, 383, 391, 399, 509-512
 Flow control strategy, 382
 Flow dependence, 52, 55, 289, [312](#)
 Flow graph, 581, 585, 620
 Flow-through delay, [268-270](#), 322, [324](#)
Flynn's classification, [11](#), [45](#)
 Forbidden latency, [272-274](#), [276](#), 277, 323, 325, 326
FORCE, 619
FORGE, 622
 Fork, 564, 669, **fiSZ**
 Fortran, [13](#), [18-20](#), [30](#), 77, 555, 560, 618-622, 631, 644, 659, 660
 Fortran [90](#), [618](#), 620, 622
 Fortran-plus, [33](#)
 Fortune, S., [35](#), [36](#), [46](#)
 Fox, G. C., 149 "
Frailong, J. M., 248, 252, 256
 Free Software Foundation (FSF), 707
 Fujitsu Ltd.
 AP 1000, 115
 MB86901 **IU**, [171](#)
 MB86911, [171](#)
 MB86920 MMU, [171](#)
 VP/2000, [410](#), [412](#), [419](#), [425](#), [426](#), [437](#), 469, 471
 VP2000, 29, 30, 122, 469
 VPP500, [5](#), [6](#), [42](#), 95, [96](#), [122](#)
 Full-scale supercomputer, 415
 Full-service protocol, 626
Function programming model, 559-560
 Functional decomposition, 652, 659
 Functional module, [215](#)
 Functional pipeline, [184](#), [186](#), [404](#), [411](#), [419](#), 421, [425](#), [436-438](#), 441, 471
Futurebus+, 483, 487

Gajski, D. D., [46](#), 72, 96, [393](#)
 Gallium Arsenide (**GaAs**), [6](#), [30](#), 174
 Gang Scheduling, [678](#)-[680](#), 692

Garbage collection mechanism, 186, 188
 Gaudiot, J.-L., 75, 96, 539
 Gaussian elimination, 613, 650
 GCD tests, 567, 570
 Gelernter, D., 479, 613, 661
 General-purpose register <GPR>, 162-164, 166, 167, 179, 208, 209
 Generality, 149
 Generations, computer, 4 fi
 Generations, multicomputer, 25-26, 367
 Geometric mean, 108
Gharachorloo, K., 256, 476, 479, 480, 482, 487, 489, 539
 Ghosh, J., 96
Giant lock, 672, 674, 676, 678, 682
 Gjessing, S., 257
 Glass, C.J., 394, 401
 Global address space, 476, 491
 Global combining, 449, 470
 Global computation model, 137, 138
 Global name space, 514
 Global naming, 507, 512
 Global optimization, 565, 566, 581, 583-585
 Global register, 173
 Global register allocator, 60
 Global shared memory, 22
Goble, G. H., 674, 712
Goff, G., 568, 573, 575, 612
 Goke, R., 337
Goldschmidt, R. E., 322
 Goodman, J. R., 256, 257, 353-356, 487, 501, 539
 Goodyear Aerospace MPP, 44, 83, 121, 447
 Gottlieb, A., 41, 46, 341, 347, 394
 Graham, S. L., 592, 593, 612
 Grain packing, 64, 66, 67, 70, 96
 Grain size, 61-63, 65, 505-507, 514
 Grand challenge, 5, 118, 120, 418, 453, 457
 Granularity, 51, 60, 63
 Graph reduction, 75
 Graunke, G., 661
 Greedy cycle, 276, 277, 279, 323-327
 Greedy routing, 375
 Greenberg, R. L., 469
 Gross, T. R., 322
 Gupta, A., 46, 149, 256, 257, 476, 479, 480, 482, 489, 539
Gurd, J. R., 45, 533, 540

Gustafson's law, 129, 131, 133-136, 151
 Gustafson, J. L., 124, 131, 133, 149, 394
Hagersten, E., 23, 256
Handoff hint, 694, 713
 Hard atom, 625, 627
 Harden, J. C., 394
 Hardware interlocking, 282, 291
 Hardwired control, 163, 164, 177
 Harmonic mean performance, 108-112
 Harrison, W., fi61
 Hatcher, P. J., 612
 Hayes, J. P., 393, 394
 Hazard avoidance, 265, 280, 282, 288, 290, 291
 Heath, M. T., 394
 Heaton, R., 469
Hellerman, H., 242
 Hennessy, J. L., 42, 46, 115, 193, 208, 256, 476, 479, 480, 482, 489, 516, 519, 539
 Hertzberger, L. O., 612
 Heterogeneous processing, 5, 6, 372, 552, 656, 659, 660
 Hewlett Packard (HP), 432
 precision architecture, 208
 Spectrum, 198
 Hey, A. J., 149
 Hierarchical cache/bus, 334
 Hierarchical ring, 501
 High-end mainframe, 24, 414, 415, 429, 470
 High-level language (HLL), 5, 8, 13, 19
High-Performance Computing and Communication (HPCC) program, 118, 120, 143, 149, 671
 High-performance parallel interface (**HIPPI**), 521
 High-speed CMOS (HCMOS), 167, 168, 170, 171
 Hill, M. D., 141, 256, 257
 Hillis, W. D., 469
 Hinds, A., 469
Hiraki, H., 540
Hirata, H., 496, 539
 Hitachi 820 series, 410, 412
 Hoare, C. A. R., 640, 646, 661
 Holt, R. C., 612
Homewood, M., 394
 Honeywell 66/60, 234

- Hopcroft, J. E.**, 149
Hord, R. M., 469
Horizontal microcoding, 182
Horowitz, M., 539
Hot spot, 87, 345
Hot-spot throughput, 87, 103
Hsiung, C., 415, 469
Hsu, W. C., 415, 469
Hudak, P., 257
Hwang, K., 4F, 61, 96, 110, 149, 322, 359, 365, 366, 393, 394, 446, 469, 539, 612, 613, 712
Hwu, W. M., 58, 96
Hybrid dataflow/von Neumann architecture, 75, 494, 534
Hydra, 41
Hypercube, 25-27, 32, 33, 43, 77-79, 84-85, 89, 100, 101, 103, 142, 143, 153, 368, 370, 373, 375, 384, 385, 390, 394, 400, 401, 478, 502, 505, 519, 644, 665
Hypernet, 96, 104
Hypertasking, 562
- I-structure**, 72, 74, 534, 538
I/O bandwidth, 416, 417, 419
I/O dependence, 52
I/O processor, 5, 341, 349, 525, 527
I/O-bound, 134
Iannucci, R. A., 96, 531, 534, 539
IAS, 5
IBM Corporation
360/370, 5, 6
360/85, 235
360/91, 305, 307, 322
370/158, 230
370/168, 234
360, 143, 306
370, 280, 306, 364
390, 677
701, 5
801, 60, 198, 297
3033, 234, 411
3090, 5, 6, 115, 414, 469
7030, 5
Empire, 534, 538
ES/9000, 23, 24, 429
GF11, 44, 77, 121
1X/370, 667, 677
- Mark I**, 4
PL8, 60
POWERstation 530, 179, 182
RP-3, 121
RP3, 95, 198, 341, 345, 347, 368, 482, 701
RS/6000, 16, 17, 57, 122, 158, 179, 181, 182, 202, 208, 521
RT, 198
RT/PC, 699, 701
System 38, 198
System/390, 24, 30, 158
Id, 514, 561
Ideal performance, 14
IEEE floating-point standard, 167, 175, 297-299, 322
IEEE Futurebus+, 90, 221-224, 334, 335, 355, 356, 364, 393
synchronization, 364
IEEE Microcomputer Standards Committee, 222
Illiac IV, 31, 44, 83, 84, 120, 447, 554, 555
Illiac mesh, 84, 101, 472
Image understanding, 653, 659
IMPACT, 60
In-order completion, 312, 314
In-order issue, 312, 314, 315
Independent requests and grants, 219-220
Information retrieval, 120
Initiation latency, 612
INMOS Ltd.
Transputer, 89, 647
IMS T9000, 394
Input set, 54, 55, 98, 289
Input/output (I/O), 5, 6, 14, 16, 19, 20, 30, 38, 39, 48, 215, 219, 226, 246, 256
channel, 23
device, 8, 24, 25, 190, 213, 216-219, 221, 226, 239
read, 228
write, 228
Input/output overhead, 14
Instruction count, 14, 15, 17, 47
Instruction cycle, 280
Instruction decode, 280, 310, 315
Instruction execute, 280, 281, 284, 288, 290, 292, 295, 296, 309-311, 316-318, 320, 321, 325

- Instruction execution rate, [106](#), [111](#), [115](#), [150](#), [151](#)
- Instruction fetch, [280](#), [281](#), [283](#), [310](#), [311](#), [325](#)
- Instruction format, [162](#), [163](#), [174](#), [209](#)
- Instruction issue, [57](#), [265](#), [311](#), [312](#), [315](#)-[317](#), [321](#)-[323](#)
- Instruction issue rate, [159](#), [208](#), [309](#), [317](#), [321](#)
- Instruction issue latency, [159](#), [160](#), [208](#), [309](#)
- Instruction **lookahead**, [fi](#)
- Instruction prefetching, [265](#), [280](#), [281](#), [283](#), [284](#)
- Instruction reordering, [412](#), [583](#), [585](#)
- Instruction set, [579](#), [598](#)
- Instruction window, [181](#)
- Instruction **write-back**, [280](#), [284](#), [325](#)
- Instruction-level parallelism, [120](#)
- Instruction-set** architecture (ISA), [15](#), [162](#), [209](#), [507](#), [622](#)
- Integer unit (**IU**), [54](#), [172](#), [179](#), [180](#), [222](#)
- Integrated circuit, [4](#)
- large-scale** integration (LSI), [4](#)
 - medium-scale integration (MSI), [4](#), [5](#)
 - small-scale integration (SSI), [4](#), [5](#)
 - very large-scale integration (VLSI), [4](#), [6](#), [157](#), [163](#), [412](#), [506](#), [507](#), [525](#)
- Intel Corporation
- 8086/8088*, [162](#), [167](#), [203](#)
 - 8008, [167](#)
 - 8080, [167](#)
 - 8085, [167](#)
 - 8087, [162](#)
 - 80186*, [167](#)
 - 80286, [162](#), [167](#), [203](#), [368](#)
 - 80287, [162](#)
 - 80387, [162](#)
 - i386*, [24](#), [162](#), [167](#), [203](#)
 - i486, [24](#), [158](#), [162](#), [165](#), [167](#), [168](#), [174](#), [203](#), [208-210](#), [225](#)
 - i586*, [164](#), [167](#), [177](#), [210](#)
 - i860, [26](#), [27](#), [158](#), [170](#), [171](#), [174](#), [175](#), [208](#), [210](#), [227](#), [234](#), [347](#), [370](#), [373](#), [621](#), [622](#)
 - i860XP, [372](#)
 - i960, [57](#), [297](#)
 - IPSC*, [84](#), [620](#), [645-647](#), [656](#), [661](#), [685](#), [701](#)
 - IPSC-VX* compiler, [567](#)
- iPSC/1**, [25](#), [84](#), [368](#), [369](#), [372](#), [505](#), [506](#), [684](#), [685](#)
- iPSC/2**, [84](#), [370](#), [372](#), [478](#), [479](#), [684](#), [685](#)
- iPSC/860**, [370](#), [372](#), [373](#), [617](#), [620](#), [621](#), [661](#).[685](#)
- iWarp, [84](#), [96](#), [394](#)
- NX/2, [667](#), [685](#)
- Paragon, [5](#), [6](#), [25](#), [26](#), [43](#), [83](#), [85](#), [121](#), [143](#), [367](#), [369](#), [372](#), [373](#), [394](#), [501](#), [686](#)
- Paragon XP/S, [621](#), [661](#)
- Touchstone Delta, [114](#), [115](#), [121](#), [143](#), [370](#), [621](#), [686](#)
- Inter-PE** communication, [71](#), [77](#)
- Interactive compiler optimization, [42Z](#)
- Interactive scheduling, [693](#)
- Interconnection network, [75](#)-[95](#), [476](#), [482](#), [518](#), [519](#), [525](#), [527](#)
- direct, [75](#)
 - dynamic, [75](#), [76](#), [89](#)-[95](#)
 - indirect, [75](#)
 - performance, [80](#)
 - static, [75](#)-[76](#), [80](#)-[88](#)
- Interdomain socket, [695](#)
- Internal data forwarding, [265](#), [280](#), [282](#), [283](#), [286](#), [287](#), [291](#), [292](#), [305](#)
- Interprocess** communication (IPC), [547](#)-[549](#), [668](#), [683](#), [686](#)-[688](#), [694](#)-[697](#), [702](#), [703](#), [711](#), [712](#)
- Interprocessor communication latency, [21](#), [33](#), [124](#)
- Interprocessor-memory** network (IPMN), [331](#)
- Interrupt, [215](#), [218](#), [221](#), [228](#), [335](#), [364](#), [373](#)
- interprocessor, [364](#)
 - priority, [221](#)
- Interrupt handler, [215](#), [221](#)
- Interrupt mechanism, [221](#)
- Interrupt message, [648](#)
- Interrupter, [215](#), [221](#)
- Interstage connection (ISC), [91](#)
- Invalidation-based cache coherence protocol, [518](#), [520](#)
- Isoefficiency**, [126](#)
- Isoefficiency function, [149](#)
- Iteration space, [568](#), [575](#), [577](#), [601](#), [603](#), [605](#), [607](#)-[608](#), [610](#)
- IVY, [476](#), [478](#)

- J-Machine**, 25, 43, 475, 501, 505–514, 538, 539
- James, D., 257
- Jermoluk**, T., 669, 712
- Joe, T., 46
- Johnson, M., 179, 208, 322
- Johnsson, L., 96
- Join, 564
- Jones, A. K., 393
- Jones, M. B., 712
- Jordan, H. F., 539
- Jouppi**, N. P., 208, 210, 320, 322, 323
- Kallstrom**, M., 612
- Kane, G., 2118
- KAP vectorizer, 567
- Karlin**, S., 612
- Karp, R. M.**, 36, 46
- Kendall Square Research **KSR-1**, 23, 27, 42, 82, 122, 145, 475, 494, 501, 516, 521–525, 538
- Kennedy, K., 18, 567, 568, 570, 573, 577, 612
- Kermani**, P., 394
- Kernighan**, B., 667
- Kilo logic inferences per second (KLIPS), 115, 117, 118, 149
- Kjelstrup, J., 322
- Kleinrock**, L., 394
- Knowledge acquisition, 403
- Kodama**, Y., 534, 540
- Kogge, P. M.**, 277, 322
- Koopman, P. J., 469
- Kowalik, J. S., 539
- Kruatrachue, B., 64, 96
- Kuck**, D. J., 18, 41, 44, 72, 143, 341, 347, 566, 567, 612
- Kuhn, R. H., 72
- Kumar, V., 61, 126, 149
- Kung, H. T., 11, 46, 51, 84, 96, 441
- Kung, S. Y.**, 84, 96
- fc-ary n-cube network, 86–89, 96, 103
- k-way** shuffle, 343
- Lam, M., 322, 568, 595, 599–601, 603, 605, 608, 610, 613
- Lamport, L., 248, 257
- Lan**, Y., 394
- Latency cycle, 273, 274, 276, 279, 325–327
- optimal, 276, 277
- Latency sequence, 273
- Latency tolerance, 147
- Latency-hiding techniques, 475–490, 516, 531
- Latency-tolerant architecture, 6
- Laudon, J., 479, 539
- Lawrie, D. H., 96, 337
- Lazy evaluation, 74, 75
- Least frequently used (**LFU**), 206, 2QZ
- Least recently used (LRU), 194, 206, 207, 212, 236
- Leasure, B., 612
- Lee, H., 96
- Lee, R. B., 112, 149, 208
- Leiserson**, C. E., 11, 46, 83, 84, 96, 103, 441, 460, 469
- Lenoski, D., 23, 42, 479, 519, 539
- Lewis, T., 64, 96
- Lexicographic order, 568, 569
- Li, K., 257, 476, 479, 539
- Lilja**, D. J., 46, 322
- Lin, X., 394
- Linda, 13, 478, 479, 647, 648, 661
- Linder, D. H., 394
- Linear array, 80, 142, 153
- Linked list, 187, 199
- LINPACK, 114–115, 429, 432–435, 650
- Lipovski, G. J., 337
- Lisp, 13, 18, 19, 32, 186, 559, 613, 618, 644
- ***Lisp**, 33
- Lisp processor, 161, 186, 187, 660
- List scheduling, 594, 595
- Livelock**, 222, 383
- Livermore loops, 416, 446
- Lo, V., 208, 478, 479, 489, 539
- Load balancing, 51, 514, 538, 617, 620, 644, 652, 659, 671, 674, 683, 712, 714
- dynamic, 654
- Load-load forwarding, 286
- Loader, 8, 9
- Local area network (LAN), 27, 516
- Local bus, 333
- Local computation model, 137, 138
- Local memory, 11, 22–24, 33, 331, 368, 373, 383
- Local network, 8
- Local optimization, 566, 579, 581, 583
- Locality, 236, 247, 565, 599, 605, 607, 609

- references, 193, 200, 207, 212, 225, 238
sequential, 193, 212
 spatial, 144, 193, 212, 480, 483, 518, 524, 525
 temporal, 144, 193, 212, 237, 238, 518, 524, 608
 Locality optimization, 605, 607, 608
 Localization, 605
 Location monitor, 215
 Lock, 551, 626–628, 635, 636, 638
 Dekker's, 634
 generalized spin, 634
 spin, 634–637, 640, 661, 662
 suspend, 634, 636, 661
 Lock synchronization, 626, 627
 Lockstep operation, 368, 554, 555
 Logic inference machine, 118
 Logic programming model, 559–560
 Logical processor, 496
 Logical reasoning, X
 Logical volume manager (**LVM**), 703, 706
 Longevity, 149
 Lookahead, 10, 528–530
 Loop buffer, 283
 Loop distribution, 587
 Loop interchanging, 570, 586, 599, 602, 607
 Loop **parallelization**, 599, 602, 603, 605
 Loop reversal, 599, 607
 Loop sectioning, 588
 Loop skewing, 599, 602, 603, 605
 Loop splitting, 577
 Loop transformation, 60, 584, 599, 601, 602, 607
Loop-carried dependence, 563, 570, 577, 589, 591
Loop-in dependent dependence, 589
 Lost message, 648
 LSI Logic Corporation
 L64811, 171
 L64814 FPU, 171
 L64815 MMU, 171
 LU factorization, 650
 LU decomposition, 481
 Mach/OS kernel, 545, 667, 678, 680, 683, 686–691, 694–697, 699–703, 712–714
 Mach/OS scheduler, 667, 686, 688, 690–692, 694
 Machine language, 5
 Macintosh, 620
 Macropipelining, 268
 Macroprocessor, 619
Macrotasking, 426, 621, 630–632
 Main **memory**, 188
 Mainframe computer, 157, 190, 334, 364
 Manager-worker approach, 652, 654, 656, 664
 Manchester **dataflow** machine, 45, 533, 540, 559
Margulis, N., 208
 Markov chain, 148
 Markov model, 393, 500
 Markstein, V., 117, 149
 Marsh, M. H., 712
 Marson, M. A., 393
 Masking register, 406
 Masking scheme, 31, 32
 MasPar **MP-1**, 31–33, 44, 121, 447, 453–470
 Massively parallel processing (MPP), 5, 8, 24, 27, 59, 89, 96, 121, 125, 143–145, 147, 372, 415, 423, 475, 492, 500–502, 505, 523, 544, 671
 Master processor, 211
 Master-slave kernel, 667, 672–674, 676, 677, 681, 682, 712, 714
 Matrix multiplication, 36, 40, 50, 114, 127, 137, 152
 Matrix-multiplication algorithms
 Berntsen's algorithm, 128
 Dekel-Nassimi-Sahni algorithm, 128
 Fox-Otto-Hey algorithm, 127
 Gupta-Kumar algorithm, 128
 May, D., 661
 McKeen, F. X., 469
McKinley, P., 394
 Medium-grain multicomputer, 505, 506
 Memory
 allocation scheme, 244
 bandwidth, 188, 209, 238, 239, 242, 243, 263, 410, 418, 455, 464, 518, 519, 531
 bank, 242–244, 263
 demand paging system, 244, 247, 258
 fault tolerance, 238, 242, 244
 global allocation policy, 245
 hit ratio, 205
 hybrid system, 247

- inverted paging, 202
 local allocation **policy**, 245
 manager, 244, 246
nonpreemptive allocation, 244
 preemptive allocation, 244
 segmented, 201, 202
 size, 188, 194, 203
 swapping, 244-247, 258
- Memory access, 248, 251, 254, 258, 262
 atomic, 251, 254, 258, 259, 264
nonatomic, 251, 252, 258, 259, 264
- Memory agent, 335, 356, 357
- Memory controller, 485, 486, 502, 507, 536, 538
- Memory cycle, 15, 35, 239, 242, 244, 263
- Memory hierarchy, 23, 157, 188, 190, 193-196, 201, 208, 209, 211, 212, 348, 363, 418, 518, 521, 522
 access frequency, 194, 125
 coherence property, 192
 hit ratio, 194, 195, 207, 212
 inclusion property, 190, 211
write-back (WB), 193
write-through <WT>, 193
- Memory interleaving, 238-240, 242-244, 263, 544
 degree of, 240, 244
 high-order, 239, 242
 low-order, 239, 242, 243, 258, 408
- Memory management unit (**MMU**), 167, 168, 174, 188, 189, 209, 225-227, 478
- Memory object, 683, 687, 689, 697, 699, 700, 703, 712
- Memory-access latency, 15, 61, 106, 147, 418, 475, 490, 494, 498-502, 527, 530, 531
- Memory-access order, 248, 252, 254
- Memory-bounded speedup model, 105, 134, 149
- Memory-mapped file, 688, 697, 700, 711, 712
- Memory-to-memory** architecture, 11, 29, 44, 49, 184, 412, 469
- Mergen, M. F., 208
- Mesh, 25, 26, 32, 33, 78, 83-86, 88, 89, 103, 122, 142, 153, 370-373, 375, 380, 386-392, 400, 479, 483, 501, 506, 509, 515, 518, 527
- Mesh routing chip (MRC), 370
- Mesh-connected** router, 373
- MESI** cache coherence protocol, 316, 355
- Message passing, 11, 13, 25-27, 547, 552, 683-685, 701
 asynchronous, 552
 synchronous, 550
- Message passing network, 24
- Message-driven** processor (MDP), 494, 506-507, 509-514
- Message-passing multitasking, 690
- Message-passing** OS model, 683
- Message-passing** overhead, 492
- Message-passing programming, 25, 551-554, 613, 644, 646, 664
- Message-routing** scheme, 26
- Micropipelining**, 268
- Microprocessor, 4, 24, 145, 157, 165, 167, 169, 174, 208, 315, 368, 369, 372, 423, 431, 432, 462, 464, 507, 514, 516
- Microprogram, 5, 6
- Microprogrammed control, 158, 159, 163, 164, 166
- Microtasking**, 426, 621, 631
- Microthreaded scheduler, 538
- Million floating-point operations per second (**Mflops**), 30, 33, 105, 106, 115, 118, 146, 147
- Million instructions per second (MIPS), 15, 105, 115, 118, 150, 152
- MIMDizer, 560, 619, 620, 661
- Minicomputer, 162, 334
- Minimal average latency (MAL), 274, 276, 277, 323, 325, 326
- Minisupercomputer**, 24, 30, 415, 429-432, 470
- MIPS Corporation
 R2000, 208
 R3000, 158, 226, 297, 318, 516, 608, 668
 R4000, 280, 281, 317, 318, 322
- MIPS rate, 15, 47, 174, 258, 262, 263
- Mirapuri**, S., 208, 322
- Mismatch problem, 57, 58, 60
- Modula**, 661
- Molecule, 613
- Monitor, 364, 545, 549, 551, 556, 561, 628, 634, 640-643, 661-664
- Monitor bit, 366

- Monitor vector, 365, 366
MONMACS, 619
Monsoon, 494, 534, 536
Monte Carlo simulation, 145
Mosaic, 25, 43, 371, 475, 501, 505, 506, 514-516, 539
Motif, 702
Motorola Inc.
 MC6800, 167
 MC68000, 167, 668
 MC68020/68030, 162, 167
 MC68040, 158, 164, 165, 167, 169, 208, 297, 303
 MC68882, 162
 MC88000, 292, 297, 479, 538
 MC88110, 23, 169-171, 177, 234, 315-316, 322, 538
 MC88200, 315
Mowry, T., 480, 482, 489, 539
Muchnick, S. S., 208
Mudge, T. N., 393, 394
Multi-RISC, 222
Multibus II, 221, 222, 334, 335, 393
Multicast, 26, 63, 78, 80, 375, 379, 388-392, 394, 400, 401
Multicomputer
 centralized, 22
 distributed-meraory, 24, 41, 145, 648
 fine-grain, 25, 370, 372, 475, 491, 505, 506, 514, 516, 542, 544
 heterogeneous processing, 25
 hypercube, 372, 400, 684, 685
 medium-grain, 25, 370, 373
 message-passing, 11, 25, 26, 48, 63, 80, 120, 266, 550, 556, 561, 617, 627, 645, 658, 665
 special-purpose, 2Z
Multicomputer network, 383, 399
Multicomputer programming, 551, 644, 645, 648, 664
Multicomputer **UNIX**, 667-712
Multicube, 475
Multidimensional architecture, 500-504
Multiflow Computer Company
 Trace computer, 182
Multilevel cache coherence, 357
Multilisp, 559
Multiphase clock, 159, 321, 322
Multipipeline chaining, 435, 438, 439
Multiple contexts (MO), 475, 476, 489, 490, 499, 531
Multiple cooperative masters, 676, 714
Multiple index variable (**MIV**), 571, 574, 577, 578
Multiple instruction streams over single data stream (M1SD), 11, 38
Multiple instruction streams multiple data streams (**MIMD**), 399
Multiple instruction streams over multiple data streams (MIMD), 11, 27, 37, 44, 46, 59, 62, 91, 121, 368, 394, 457, 505, 516, 548, 554-556, 559, 625
Multiple programs over multiple data streams (MPMD). 63, 120, 368, 399, 549
Multiple **SIMD**, 44
Multiple-instruction issue, 60, 310, 322
Multiport memory, 331, 336, 340, 341
Multiported register, 283
Multiprocessing, 30, 549, 550, 621, 628, 629, 631
Multiprocessor
 asymmetric, 20, 219, 674
 atomic memory, 259
 bus-based, 353, 355
 bus-connected, 258
 cache-based, 223, 251
 central-memory, 21
 distributed-memory, 27
 GaAs-based, 30
 loosely coupled, 551
 multithreaded, 44
 network-based, 257
 scalable, 27, 48, 257
 shared-memory, 11, 18-20, 22, 27, 41, 48, 76, 94, 151, 248, 257, 355, 505, 516, 521, 549, 627, 648, **fifo**
 symmetric, 19, 706
 tightly coupled, 548, 552
 transaction processing, 24
 unscalable, 21
Multiprocessor programming, 548, 612
Multiprocessor scheduling, 67, fig
Multiprocessor UNIX, 667-712
Multiprogrammed computer, 14, 16, 48
Multiprogramming, 5, 6, 16, 549, 550, 621, 661
Multistage interconnection network (**MIN**),

19, 24, 41, 76-78, 89, 91-93, 96, 101, 331, 336, 358, 393, 418, 447, 454
Multitasked clustering, 701
 Multitasking, 254, 426, 545, 549, 562, 591, 621, 628-633, 656, 668-671, 678, 683, 709
 Multithreaded architecture, 75, 148, 475, 491, 498, 500, 516, 531, 537-539, 550
 Multithreaded kernel, 667, 672, 677, 678, 680-682, 712, 714
 Multithreaded MPP, 491, 536
 Multithreaded multitasking, 678, 686-688
 Multithreaded processor, 49, 494, 596
 Multithreaded server, 709
 Multithreading, 3, 44, 475, 491-500, 514, 525, 530, 534, 538, 550, 562, 671, 678, 680, 682, 690, 701, 709, 713
 Multiway shuffle, 91
Multiple-context processor, 491, 494, 496, 539
 Mutual exclusion, 198, 548, 549, 551, 552, 555

N-queens problem, 654
Nanobus, 222, 334, 335, 393
 Nassimi, D., 149
 National Semiconductor
NS32532, 167
NCUBE Corporation
 3200, 614
 6400, 394
nCUBE/10, 368
nCUBE/2, 26, 43, 115, 120, 370
 Near-supercomputer, 24, 429
 NEC, 143, 318
 SX series, 30, 410, 412, 441, 469, 472
 SX-3, 115
 Network control strategy, 332
 centralized, 332
 distributed, 332
Network diameter, 77, 80, 82-86, 89
 Network flow control strategy, 26
 Network interface unit, 538
 Network latency, 80, 87, 89
 Network partitioning, 391
 Network Queueing System (NQS), 621
 Network server, 696

Network throughput, 87-88
 Network UNIX host, 683
 Network-connected system, 348
 New Japanese national computer project (NIPT), 536
 NeXT Computer, 686, 688, 691, 694, 701, 713
 Ni, L. M., 13, 46, 105, 107, 129, 134, 137, 149, 394, 401
Nickolls, J. R., 469
 Nicolau, A., 149, 613
 Nikhil, R. S., 45, 492-494, 532, 534, 536, 537, 539, 540
 Nitzberg, B., 208, 478, 479, 489, 539
No-remote-memory-access (NORMA) model, 24, 48, 690, 714
 Node degree, 77, 82-86, 89
 Node duplication, 67
 Node name, 64
 Node splitting, 588
Node-addressed model, 683, 714
 Noncacheable data, 363
Nonuniform-memory-access (NUMA) model, 19, 22-24, 42, 46, 48, 331, 368, 476, 690, 701, 714
NP-class problem, 34
NP-complete problem, 34, 569
 NP-hard problem, 67
 NuBus, 222, 393
 Numerical computing, 1
Nussbaum, D, 141, 142, 149

Object decomposition, 644, 656, 657, 664
 Object-oriented model, 683, 686, 688, 695, 712, 714
 Object-oriented programming (OOP), 514, 556, 557, 643, 656, 657
 Occam, 646, 647, 648
 Oldehoeft, R., 612
 Omega network, 86, 91, 96, 101-103, 142, 143, 153, 336, 337, 341-343, 345, 347, 393, 398, 399, 486, 534
 Open Software Foundation **OSF/1**, 545, 621, 622, 667, 686, 701-703, 708, 712
 Operand fetch, 280, 281, 308, 325
 Operating system
 Cray operating system (COS), 411, 412
 UNICOS, 412, 419
 ULTRIX, 430

- VMS, [430](#)
CMOST, 622
COS, [621](#)
kernel, [8](#), [19](#), [226](#), 244, 246
Mach, 524
multiprocessing, 6
multiprocessor, [5](#)
time sharing, [5](#), 6
ULTRIX, [430](#)
UNICOS, 621
UNIX, [8](#), [17](#), [26](#), [27](#), [30](#), [33](#), 228, [244](#), 246, [247](#), [412](#), [419](#), [431](#), [453](#), 458, 462, [545](#), 550, 551, 556, 562, 612, 620-622, 640, 644, 667-715
VMS, 247, [430](#)
Optimal (OPT) replacement algorithm, 206
backward distance, 205
forward distance, 205
Optimistic concurrency, 627, [661](#)
Optimistic synchronization, 627
Optimization, 564-566, 570, 571
Optimizing compiler 566
Orthogonal multiprocessor (OMP), [475](#), [501](#)-
504
Otto, S. W., 149, 612
Ousterhout, J. K., 712
Out-of-order completion, [312](#)-[315](#)
Out-of-order issue, [312](#), 313, 315
Output dependence, [52](#), 289, [312](#)
Output set, [54](#)-[56](#), [98](#), 289
Ownership protocol, 352
Ownership-based cache coherence protocol, [480](#)
- P-class** problem, 34
P-RISC, 534, 540
Packet, 332, 375-378, 380-383, 387-390, 399
Packet data transfer, 221
Packet switching, 332
Padua, D. A., 72, 612
Page, 198, 201
Page fault, [195](#), 201, [203](#), [205](#), 206, 477
Page frame, 198, 201-203, [205](#)-[207](#), [211](#)
Page table, 198, 200-203
Page trace, 205
Paged segment, 202
Page swapping, 477-479
Palmer, J., 394
- Papadopoulos**, G. M., 534, 536, 537
Paraphrase, 60, 566, 567
Parallel complexity, [34](#)
Parallel computer, [6](#), [7](#), [9](#), [11](#), [13](#), 17, 19, [27](#), [32](#), [33](#), [37](#), [38](#), [46](#)
Parallel environment, [5](#), [17](#)
Parallel Fortran, [23](#)
Parallel Fortran Converter (PFC), 566-568
Parallel I/O, 671
Parallel prefix, 465, 466, 468, 471
Parallel programming, 510, 514, 545-623
Parallel programming environment, 562, 568, 617, [619](#)
ParaJlel programming model, 547-560
Parallel random-access **machine** (PRAM), [1](#), [3](#), [32](#), 33, [35](#)-[38](#), [46](#), 49, 50, [141](#)-[143](#)
CRCW-PRAM, [36](#), [38](#)
CREW-PRAM, [36](#), [38](#)
ERCW-PRAM, [36](#), [38](#)
EREW-PRAM, [36](#), 49, 141, [142](#), 153
Parallel suffix, 466
Parallelism
average, [105](#), [106](#)
coarse-grain, [63](#), 602, 605
control, 59, 554
data, [27](#), 59, [63](#), 120, 186, 554, 555, 561, 562
degree of (**DOP**), [27](#), 57, [63](#), [66](#), [70](#), 105, [106](#), 108, [114](#), [118](#), [130](#), [131](#), 133, 136, 151, 505, 554, 566, 595, 599, 602, 603, 605, 660, 672, [678](#)
detection, [9](#), [17](#)-[19](#), [54](#), 55, 57, [61](#), 62, [183](#), 550, [619](#)
explicit, [18](#), [54](#), 552, 561, 645
fine-grain, [6L](#) [63](#), 71, 505, 506, 538, 550, 565, 602
functional, [10](#)
global, [561](#)
hardware, 57-713
implicit, [18](#), [54](#), [561](#)
instruction-level (**ILP**), [45](#), [61](#), [120](#), [170](#), 178, 180, 182, [183](#), 809, 310, 315, [316](#), 321, 322, [549](#), 565, 578
job-level, [63](#)
layered, 656, 659
loop-level, [62](#), [561](#)
machine-level, 585
medium-grain, [63](#), 556

- perfect, 659
procedure-level, [62](#), 549
 software, [57–60](#), 96, 105, 113, 560
 spatial, [11](#), [318](#), 555
 stream, 559
 subprogram-level, [62](#)
 system-level, 585
 task, [26](#)
 task-split, [561](#)
 temporal, [11](#), [318](#)
unrolled-loop-level, 185
 Parallelism profile, 105–108, 133
 Parallelization, [545](#), [566](#), [584](#), [585](#), [589](#), [591](#),
 599, 602, 606, 620
 Parallelizing compiler, 567
 ParaScope, 567, 568
Par begin, 564
Parend, 564
 PARIS, [33](#)
 Parity check, 339
 Parsys Ltd.
 SuperNode 1000, [25](#), [26](#), 371
Parsytec FT-400, 115
 Partial differential **equation** (PDE), [132](#), [133](#)
 Partially **shared-memory** multitasking, 690
Particle-based three-dimensional simulator,
 481
 Pascal, [18](#), [19](#), 618, [621](#), 644
 Pascal, **B.**, [4](#)
 PAT, 619
Patel, B., 469
Patel, J. **H.**, 257, 277, 322, [337](#), 393, 398
 Patterson, D. A., [46](#), 115, 193, [208](#)
 Peak performance, [14](#)
Peir, J. **K.**, [46](#), 393
 Perfect Club benchmark, 116
 Perfect decomposition, 649, 664
 Perfect shuffle, [78–79](#), 91, 96
 Performance booster, [281](#), 282
 Performance evaluation, 8
 Performance factor, [15](#)
 Performance index, [14](#)
 Performance measurement, [16](#), 105–118
 Performance optimized with enhanced RISC
 (POWER), [179](#), [181](#)
 Performance tuning, [545](#), [617](#), [620](#), [624](#), [625](#),
 648, 652
 Performance visualization system (PVS), 622
 Performance/cost ratio (**PCR**), [269](#), 323
Peripheral technology, 190
 Permutation, [26](#), [63](#), [78](#), [90](#), 92, 95, 101,
 338, 341, [343](#), 398, 465, 467–468,
 471
Perrott, R. **H.**, 555, 612, **fifii**
 Personal computer (PC), [167](#), 620
 Personalized broadcast, 80
 Personalized communication, [78](#)
 Petri net, [148](#), 393
 PFC, 60
 Physical channel, 375, 379, 380, 391, 399
 PIE, 617 –
 Pipeline, [9–11](#), [28](#), [29](#)
 arithmetic, [265](#), 291, 297, 299, 300,
 [304](#), [307](#), [308](#), 421, [432](#), 442
 asynchronous, [265](#), 266
 dynamic, [270–272](#)
 efficiency, [270](#), 279, 315, 323
 feedback **connection**, [270](#), [284](#)
 feedforward connection, [270](#).
 fixed-point, 297
 floating-point, 319
 frequency, [267](#)
 instruction, [57](#), [159](#), 166, 168, 169, 177–
 [179](#), [265](#), [280–283](#), 285, [288](#), 289,
 291, 293, [296](#), 297, [312](#), [322](#), 327
 linear, [265](#), [267](#), [268](#), 270, 271, 280,
 322, 327
 memory-access, 184, [265](#), [281](#), [421](#), [437](#),
 438, 442, 471
 multifunction, [265](#), [270](#), 297, 307, 308
 nonlinear, [270](#), [284](#)
 optimal number of stages, [268](#), [270](#)
 scheduling, 57, 312, 322
 speedup, [268](#), [270](#)
 stage, [266–270](#), [272](#), [276](#), 277, 281, 283–
 285, [296](#), 297, [300](#), [303–305](#), 307,
 309–313, 317, 318, [321–323](#), 325–
 328
 stage delay, [266](#), [267](#)
 stage utilization, [267](#), 313
 stalling, 285, 290, 291, 310–312
 static, [270–272](#)
 streamline connection, [270](#)
 synchronous, [265–267](#)
 throughput, 325
 underpipelined, 160, 165, 210
 underutilized, 160
 vector, [28](#), [30](#), 185

- Pipeline chaining, 178, 588
 Pipeline concurrency, 557, 558
 Pipeline cycle, 159, 160, 171, 279, 292, 307, 309, 316
 Pipeline flow-through latency, 438
 Pipeline latency, 272–274, 277
 Pipelining, 5, 6, 10, 265–328
 asynchronous, 375, 377
 synchronous, 378
Pipenet, 441–444, 446, 470, 473
PISCES 2, 561
Polychronopoulos, C. D., 612
 Port set, 694, 695, 714
 Portability, 478, 556, 618, 670, 686, 688, 700, 703, 705, 714
 Position-independent code, 708
POSIX, 703–705, 708–710
Postoptimization, 592
 Postprocessor, 290
Postsynchronization, 627, *fifil*
 Pountain, D., 661
 Power monitor, 215
 Powere, D. M., 322
 Prasanna Kumar, V. K., 36, 40, 46
 Precompiler, 9, 556, 565, 619
 Prefetch buffer, 283, 290, 305, 308
 Prefetching techniques, 475, 480–481, 489, 490, 498, 507, 521, 524, 530, 607
 hardware-controlled, 480
 nonbinding software-controlled, 480
 software-controlled, 480, 539
 Preparata, F. P., 96
 Preprocessor, 9, 23
 Presence vector, 358, 359
Presynchronization, 627, 661
 Prism programming environment, 622
 Process key, 228
 Process migration, 348, 357, 363
 Processing element (PE), 11, 31, 33, 48, 49, 71, 72, 77, 80, 95, 120–122, 144, 447–449
 Processor allocation server, 679
 Processor cache, 502
 Processor consistency (PC), 479, 487, 488, 539
 Processor cycle, 15
 Processor design space, 157–159, 208
 Processor efficiency, 494, 498–500, 531
 Processor scheduling, 8
Processor-I/O network (PION), 331
Processor-network-memory operation, 525, 527
 Producer-consumer problem, 493, 538, 641–643
 Program counter (PC), 71, 72, 491, 496, 514, 527, 530, 536
 Program graph, 58, 64, 66, 67, 69, 70, 72, 100, 442, 444
 Program order, 250–255, 258–260, 264
 Program partitioning, 54, 61, 64, 96, 550
 Program profiling, 579
 Program replication, 550, 552, 562
 Program restructurer, 566
 Program restructuring, 550, 560
 Program scheduling, 61
 Program status word (PSW), 496
 Program **trace-driven** simulation, 236
Programmability, 8, 17, 24, 26, 27, 33, 48, 147–149, 475, 516
 Prolog processor, 186, 559
 Protected access, 634
Przybylski, S., 236, 256
 PSO memory model, 263
PTRAN, 567

 Quality of parallel computation, 105, 113, 114, 119, 149
 Queueing model, 148
 Quinn, M. J., 149, 612

Raghavendra, C. S., 40, 46
 Ragsdale, S., 661
 Ramachandran, V., 36, 46
Ramamoorthy, C. V., 118
 Random replacement algorithm, 206, 207
 Random-access machine (RAM), 33, 35
 Random-access memory (RAM), 189, 232, 236, 239, 299
 dynamic (DRAM), 4, 372, 418, 429, 462, 507
 static (SRAM), 4, 418, 502, 534
 Range, 55, 289
 Rao, N., 126, 149
 Rashid, R. F., 712
Ravishankar, M., 479, 539
 Reactive Kernel, 667, 683, 686, 714
 Read set, 55

Read-after-write (RAW) hazard, 289, 290, 626
Read-modify-write, 345, 364
 Read-only memory (ROM), 159, 164, 299
 Recirculating network, 336
Reconfigurable SIMD/MIMD computer, 403
 Reduced instruction-set computer (RISC), 17, 33, 158, 668
 Reduction, 465, 466, 468, 471
 Reduction computer, 70, 74, 75
 Redundancy, 105, 113, 114, 149, 339, 391
 Reed, I., 5
 Reentrant code, 629
 Register, 188
 floating-point, 305, 319
 Register allocation, 579, 592, 595
 Register file, 163, 164, 177, 180, 182, 184, 209, 507
 Register tagging, 283, 292, 305
 Register transfer language, 5
 Register window, 170–174, 177, 209, 210
Register-to-register architecture, 11, 44, 49, 184, 412, 421, 469
Reif, J. H., 455, 469
 Relative MIPS rate, 115
 Relative performance, 413
 Relaxed memory consistency, 475, 487-490
 Release consistency (RC), 487-490, 539
 Reliability, 222, 339
 Remote load, 492-494, 536
 Remote procedure call (RPC), 627, 646, 688, 695
 Reorder buffer, 180, 209
 Replication, 465, 466, 471
 Reservation station, 180, 209, 305, 306
 Reservation table, 267, 271-277, 279, 284
 285, 323-327
 Resource allocation graph, 639, 664
 Resource conflict, 159, 165, 179, 208
 Resource dependence, 54-57
 Response time, 108, 130
 Ring, 26, 78, 80-82, 84-86, 89, 483, 501, 521, 522
 Ring multi, 521
 Ring routing cell (RRC), 524
 RISC processor, 32, 157-159, 162, 164—
 171, 177-179, 208, 279, 283, 297, 315, 412, 453, 462, 566
 Ritchie, D. M., 667, 712

Rivest, R. L., 46
 Robertson, G. G., 469, 712
 Rothnie, J., 539
 Routing function, 31
 Run queue, 672, 673, 678, 692, 693
 Russell, C. H., 713

 S-access memory, 408-410
 Saavedra, R. H., 494, 500, 539
 Sahni, S., 149
Sakai, S., 45, 534, 540
Satyan, , 393
 SAXPY code, 435, 436, 438
 Scalability, 24, 27, 33, 35, 38, 48, 63, 76, 80, 84, 85, 89, 96, 105, 122, 125, 131, 138, 139, 141-144, 147-149, 334, 368, 415, 418, 421, 458, 475, 478, 479, 484, 487, 500, 516, 518, 519, 521, 524, 544, 703
 generation (time), 27, 145, 523
 metric, 124, 139
 problem, 145, 147
 size, 27, 144, 145, 147, 523
 software, 148
 technology, 147
 Scalability analysis, 105, 124, 138, 140, 141, 148
 Scalable architecture, 5, 6, 13, 84, 415, 418, 475, 487, 515, 516, 521, 531
 Scalable coherence interface (SCI), 27, 148, 257, 394, 483-487
 Scalable computer, 122, 143-145, 147, 148
 Scalable processor architecture (SPARC), 171, 172, 174, 208-210, 256, 259, 260, 457, 462, 464
 Scalar processor, 27, 28, 161, 162, 165, 170, 300, 312, 315
 Scalar-concurrent, 589-591
 Scaling track, 514, 316
 Schaefer, D., 479, 539
SCHEDULE, 617, 620, M1
 Scheduling overhead, 63, 64, 66, 67
 Scheurich, C., 46, 251, 256, 348, 349, 351, 393
 Schimmel, C, 681, 712
 Schwarz, P., 393
 Scoreboarding, 265, 282, 288, 291, 292, 315, 322
 Search tree, 652, 655

Seitz, C. L., [19](#), [39](#), [43](#), [46](#), 368-370, [394](#), [514](#), 515, 539, [661](#), 713
 Self-service protocol, 626
Semaphore, [345](#), [364](#), 545, 551, 556, [561](#), 628, 637, 663
 binary, 549, 637-640, [fifii-fifia](#)
 counting, 549, 551, 634, 638, 640
Semaphored kernel, 682
Sender-synchronized protocol, [364](#)
Sequent Computer Systems
 Balance 21000, 701
 Symmetry, 355
 Symmetry S81, [24](#), [90](#), 95
 Symmetry series, 118, [431](#)
Sequential bottleneck, 112, [130](#), [131](#)
Sequential buffer, 283
Sequential computer, [9](#), [17](#)
Sequential consistency (SC), 248, 251-256, 258, 479, 487-489, 523, 551
Sequential environment, [17](#)
Sequin, C., [208](#)
Serial complexity, [34](#)
Serial fraction, 112
Serialization principle, [345](#)
Serially reusable code, 629
Server synchronization, [627](#), 628
Session control, 705
Session leader, 705
Set-associative mapping, *see* Cache
Sevcik, K., 96
Shadow object, 698, 700
Shang, S., 359, 365, 366, 394
Shapiro, E., 612
Shar, L. E., 277, 322, 323
Shared distributed memory, 122, 136, 149
Shared memory, [6](#), [18-24](#), [27](#), [35](#), [37](#), [213](#), 228, 238, 248, 253, 258, 262, [331](#), 335, [341](#), [345](#), 348, [349](#), 351, 353-355, 365, 368, 395, 399
Shared variable, [11](#), [13](#), [19](#), [345](#), 348, 398, [548](#), [549](#), [551](#), [559](#), [561](#)
Shared virtual memory (SVM), [6](#). 148, 370, [372](#), [476](#), [671](#), [686](#), [711](#)
Shared-memory multiprocessor, *see* Multiprocessor
Shared-variable communication, 548
Shared-variable programming model, 547-551
Sharing list, 483-486

Sheperdson, J. C., [35](#), [46](#)
Shih, Y- L., 394
Shimada, T., [45](#)
Shiva, [478](#), [479](#)
Side-effect dependence, 626
Siegel, H. J., [31](#), [46](#), 96
Siewiorek, D. P., 469
Silicon Graphics, Inc.
 4-D, 226, [479](#), 517
Simple cycle, [276](#), [324](#), 325, 327
Simple operation latency, [159](#), [165](#), 166, 178, [208](#), 309, 317, 321
Sindhu, P. S., 248-256
Single floating master, 674
Single index variable (SIV), 571, 574-576, 578
Single instruction stream over multiple data streams (SIMD), [11](#), [27](#), [30-33](#), [37](#), [43](#), [48](#), [59](#), [63](#), [77](#), [80](#), [91](#), [120](#)-122, 143, [183](#), 186, 368, 399, [403](#), [446-457](#), [505](#), 539, 547, 554-556, [561](#), 614, 617, 625
Single instruction stream over single data stream (SISD), [11](#), [49](#)
Single master kernel, 672
Single program over multiple data streams (SPMD), [62](#), [63](#), 120, [368](#), 399, 504, 524, 539, 541, 554, 556, 562, 614, 649
Single-assignment language, 559
Single-error correction/double-error detection (SECDED), 421
Single-stage network, 336, 338
SISAL, 559, [612](#)
Slave processor, 162
Sleep-wait protocol, *see* Wait protocol
Slotted token ring, 500
Small computer systems interface (SCSI), 334, 393
Smith, A. J., 238, 256
Smith, B., 713
Smith, B. J., [44](#), 516, 525, 539
Smith, B. T., [661](#)
Smith, J. E., 281, 289-291, 322, 415, 469
Snir, M., [36](#), [46](#)
Snoopy bus, 351-355
Snoopy cache-coherence protocol, [351](#), [355](#)-358, 396, 482, [519](#)
Soft atom, 625, 627

- Software compatibility, **145, 147**
 Software interlocking, **265, 282, 288**, 290, 296
 Software overhead, **21**
 Software pipelining, 60, 599, 607, 610-612
 Software portability, **145**
Sole-access protocol, 625, 627
 Solid-state storage (SSD), 411, **420-423**
 Sohi, G. **S.**, 285, 322
 Sorensen, D. C., **661**
 Space complexity, **33, 34**
Space-sharing system, 685, **701**, 714
 Sparse **three-dimensional** torus, 527
 SPEC, 116
Special-function unit (**SFU**), **17L** 538
 Speculative execution, 60, 315, 578, 579
 Speedup, **21**, 49, **481**, 490, 498, 514
 asymptotic, **107, 108, 141, 142**
 harmonic mean, **110**
 ideal, **105, 112, 130**
 Speedup curve, 131
 Speedup factor, **105, 112, 130, 150-152**
 Spin lock, **364**, 673, 678, 682
 Split transaction, **221**, 222, 335, 355-357, 491
 Squashing, 297
 Stack pointer, **493**
 Stack processor, 186
 Stack-oriented machine, **187**
Stallings, W., **208**
 Stanford MIPS, 60
 Star, **82, 83**
STARAN computer, 96
 Stardent 3000, **30**
 State diagram, **274-277, 279, 295**, 323, **325**, 327
 Static network, **24, 25**
 Static scheduling, 290, 291, 550, 579
 Steele, G. **L.**, 469
Stenström, P., **46**, 257
 Stokes, It A., **44**
 Stone, **H. S.**, **78, 96**, 256
 Storage dependence, **54**
Store-and-forward routing, 368, 375, **376**, 378, 380, 383, 387-389, 399
Store-load forwarding, 286
 Store-store forwarding, 286
 Stout, Q. F., 394
 Strand 88, 559, 617
 Strength reduction, 585
 String reduction, **74**
 Stripmining, 437, 588
 Strong **SIV** subscript, 574
 Strong SIV test, 576
 Sturgis, H. E., **35, 46**
 Subscript partitioning, 572, 573
 Sullivan, H., **384, 394**
 Sun Microsystems, **432, 457**, 620
 SPARC, 158, **171**, 172, **174**, 254, 256, 259
 SUN **3/200**, 228
 Sun, X. **H.**, **105, 107, 129, 134**, 137, 149
SunView, 620
 Supercomputer, **8, 27, 29, 30, 32, 33, 44, 46**, 49, **63, 118-120**, 143, 149, 157, **159, 183, 184, 186, 403-473**
Supercomputing workstation, **429, 432**, 470
 Superpipeline degree, **316-318**, 320, 321, 323, 326
 Superpipelined processor, **157, 159, 165, 185**, **208, 316**, 585
 Superpipelined superscalar degree, **317, 318**, 325
 Superpipelined superscalar processor, 317-595
 Superpipelining, **60, 265, 281, 316-318, 322**, 416, 525
 Superscalar degree, **310, 320, 321, 323**, 326
 Superscalar processor, 100, 157-159, **163**, **165, 175**, 177-182, 185, **208-210**, **280, 310-316**, 318, 521, **537**, 565, 585, 596, 599, **603, 612, 668**, 678
 Supersymmetry, **320**
 Suspend lock, **364**
 Swap device, **245**, 246
 Swap space, 245
 Sweazey, P., 257
 Symbolic manipulator, 186
 Symbolic processing, **187**, 652
 Symbolic processor, **157, 186**, 212
 Symbolics 3600, **187**, 188
 Symmetric multiprocessing, 702, 703
 Synchronization
 hardware, **364**, 554
 shared-variable, **661**
 software, **364**, 551
 Synchronization space, 646
 Synchronization coprocessor (**sP**), 538

- Synchronization frequency, 624
 Synchronization latency, [61](#), [493](#)
 Synchronization overhead, [64](#), 147, [152](#), [262](#),
 372, 505, 514, 531
 Synchronization point, 551
 Synchronized **MIMD** machine, 457, 458
 Synchronizing load, [492](#), [493](#), 536, 539
 Synchronous bus timing, 216-217
 Synchronous program graph, [444](#)
 Synonym problem, 198
 System clock driver, [215](#)
 System integration, 51
 System overhead, [14](#), [16](#), [106](#), [108](#), [130](#), [151](#)
 System thrashing, 681
 Systolic array, [11](#), [46](#), [84](#), 441-442
 Systolic program graph, [444-446](#), [470](#), [473](#)
- *T, [45](#), [122](#), 475, 494, 531, 534, 536-540
Tabak, D., [208](#), 393
 Tagged token, 533
 Tagged-token dataflow architecture (TTDA),
 [45](#), [71](#), [74](#), 533, 539
 Tandem, 96
 Tape unit, 189
Tape-automated bonding (TAB), 515
 Target buffer, 283, [292](#)
 Task address space, [689](#), [697](#)
 Tera multiprocessor system, [44](#), [122](#), 475,
 494, [501](#), 516, [525-531](#), [538](#), 539
TeraDash, 521
Teraflops, [6](#), [118](#), [143](#), [145](#)
Test&Set, [364](#), 551, 625, 636, [637](#), 662
 Tevanian, A., 712, 713
 Texas **Instruments**
 8847 FPP, [171](#)
 8846, [121](#)
 8848, 170
 ASC, [5](#), [6](#), 297, 307, [308](#), 322
Thakkar, S. S., 256, 257, 393, 612, [661](#)
 Thinking Machines Corporation (**TMC**)
 CM-2, [27](#), [31-33](#), [44](#), 77, 83-85, [120](#),
 121, 143, [145](#), 368, 447-453, 457,
 465, 466, 469, 470, [473](#), 505, [506](#),
 555, [661](#)
 CM-5, [5](#), [6](#), [27](#), [44](#), 83, 85, [122](#), 143,
 145, [403](#), [453](#), 457-469, 471, [473](#),
 621, 622, [661](#)
 Thomas, D. P., 322
 Thompson, C. D., [38](#), [46](#)
- Thompson, K., 712
 Thornton, J. E., 322
 Thrashing, 244, 246, 247
 Thread management, [686](#), 688, 690, 715
 Thread slot, [496](#), 498
 Thread state, [525](#), 529
Three-party transaction, 222
 Throughput, [16](#)
 CPU, [16](#), [17](#)
 pipeline, [266-270](#), [276](#), [280](#), 282, 283,
 286, [292](#), 311, 323-328
 system, [16](#), [17](#)
 Tightly coupled **system**, [19](#)
 Tiling, 599, [602](#), [605-608](#), 610
 Time complexity, [33](#), [34](#)
 exponential, [34](#)
 Time sharing, 549, 621, 622
Time-sharing bus, 89
 Time-sharing system, 667, 679, 685, 690,
 [691](#), [693](#), [701](#), [712](#), [714](#)
 TITAN architecture, [432](#)
 Token ring, [82](#)
 Token matching, 71
Tomasulo's algorithm, 305, 306
Tomasulo, R. M., 322
 TOPSYS, 617
Torng, H. C., 322
 Torus, [26](#), [83-84](#), [86](#), 103, 500, 501, 521,
 525, 527
 Total store order (TSO) memory model,
 [252](#), 254-256, [263](#)
TP1 benchmark, [117](#), [118](#), 149
 Trace scheduling, 184, 596, 598, 612
 Transaction modes, 221
TVansaction processing, [7](#), [23](#), [90](#), [96](#), [363](#),
 [429](#), 668, 671
Transactions per second (TPS), 105, 117
Translation lookaside buffer (TLB), [167](#), [174](#),
 198, 200-203, 209, 225, 226, 233,
 668, 700
 Tree, [26](#), [82-84](#), 334
TVew, A., [46](#), 393, 469
 Tseng, C. W., 568, 573, 612
 Tseng, P. S., 539
 Tucker, L. W., 469
 Tuple space, [13](#)
 Turnaround time, [14](#)
- Ullman, J. D-, [38](#), [46](#), [149](#)

- Ultracomputer**, 41, [42](#), [341](#), [345](#), 347, 394
Unicast, 375, 388, 389, 391, [400](#)
Uniform-memory-access (UMA) model, [19](#), [22](#), [23](#), [41](#), [48](#), [331](#), [690](#), 714
Unimodular matrix, 601
Unimodular transformation, 599, 601, 605, 610
Uniprocessor, [17](#), [19](#), [20](#), [29](#), [35](#), [46](#), [48](#), 49, 60, [63](#), 69, 71, 73, 112, [139](#), 141, [150](#)–[152](#), [248](#)–[250](#), 254, 261, 262, [357](#), 544
Univac [1100](#)/[94](#), [341](#)
Univac LARC, [5](#), [6](#)
UNIX, *see* Operating system
UNIX System V, 671, 702, 704, 708
Unknown dependence, 52
Utilization, [105](#), [106](#), [111](#)–[114](#), 124, 133, [134](#), 137, 147–149
- Vasseghi**, N., [208](#), 322
VAST vectorizer, 567
VAX units of performance (VUP), [116](#)
Vector balance point, 415
Vector instruction, [11](#), [30](#), 32, 49, [403](#)–[406](#)
 gather, [405](#)
 masking, [406](#)
 scatter, [405](#)
 vector reduction, [405](#)
 vector-memory, [404](#)
 vector-scalar, [404](#)
 vector-vector, [404](#)
Vector length register, [406](#)
Vector loop, [437](#), 441
Vector processor, [5](#), [10](#), [11](#), [13](#), [27](#)–[30](#), 120, 157, [159](#), [161](#), 177, [178](#), 184–186, 210, [341](#), [403](#)–[446](#)
Vector recursion, 441
Vector reduction, 587
Vector register, [1L](#) [29](#), [30](#), [402](#), [404](#)–[406](#), [408](#), [412](#), [425](#), [426](#), [430](#), [436](#)–[438](#), 440–442, 588
Vector-access memory scheme, 408–410
Vector-concurrent, 589–591
Vectorization, [404](#), 415, [416](#), [427](#), [432](#), 436, 545, 562, 566, 567, 584–589, 591, 596, 599, 621
Vectorization inhibitor, 588
Vectorization ratio, [404](#), 413, [414](#), 472
Vectorizer, 567,
 see also Vectorizing compiler
Vectorizing compiler, 567
Very long instruction word (**VLIW**), [159](#)
Virtual address space, 197, 688, [697](#), [700](#), [709](#)
Virtual channel, [26](#), 375, [379](#)–[382](#), [387](#), [388](#), [391](#), [392](#), [394](#), [399](#)
Virtual cut-through routing, 383, 394, 399
Virtual file system (**VFS**), 703
Virtual interrupt, [221](#)
Virtual memory, 157, 194, [196](#)–[198](#), [201](#), 202, [208](#), [211](#), 667, 687, [688](#), [690](#), [697](#)–[700](#), 702–704, 715
 paged, 174, 206
 private, 198, [211](#)
 shared, [198](#), [208](#), [211](#)
Virtual network, 388, 391, [400](#)
Virtual processor, [452](#), 470, [561](#), 678, 679, [713](#)
Visualization, 618, 620–622, 660
 data, [623](#)
 program, 618, 620
VLIW processor, 157, [159](#), 182–184, [208](#), 209, 550, 578, 596, 598, 603
VLSI complexity model, [38](#)
VME bus, [90](#), [214](#), [215](#), 221, 334, 335, 393, 483, 487
VME International Trade Association (VITA), 214, 221, 222
Von Neumann architecture, [9](#), [10](#), [44](#), [45](#), 71, 75, 516, 531, 534, 536, 539
Von Neumann multithreading architecture, 491, 525, \$38
Vuillemin, J. E., 96
- Wah**, B., US
Wait protocol, 625, 626
 busy, 637
 sleep, [636](#)–[638](#), [661](#)
Wall, D. W., [61](#), 149, [208](#), 210, [320](#), 322, 323
Wallace tree, 302
Wallace, C. E., 322
Waltz, D. L., 469
Wang, H. C., 149
Waterman, P. J., 713
Watson, L., 533, 540
Watson, W. J., [45](#), [322](#)

- Wavefront transform, 602, 603, 605, 607
Weak consistency (WC), 213, 248, 254, 256, 487, 488, 551
Weak SIV test, 576
Weak-crossing SIV, 577
Weak-zero SIV, 576
Weber, W. D., 476, 480, 482, 489, 539
Weicker R. P., 117, 149
Weiser, W., 539
Weiss, S., 322
Weitek, 162
Whestone, 115, 116, 149
Wide-area network, 8
Wilson, A. W., 334, 357, 393, 394, 399
Wilson, G., 46, 393, 469
Wilson, T. L., 469
Window register, 173
Winsor, D. D., 393
Wire bisection width, 77
Wired-NOR barrier line, 366
Wirth, N., 643, 661
Woest, P., 501, 539
Wolf, M. E., 599, 601, 603, 605, 608, 613
Wolfe, M., 570, 605, 612
Woodacre, M., 208, 322
Word processor, 8
Working set, 194, 200, 208, 212
Workload, 126
Workload curve, 122, 124
Workstation, 8, 16, 27, 47, 157, 176, 182, 189, 334, 355, 432, 457, 515
Wormhole routing, 89, 266, 370, 375–380, 383, 387–389, 391, 394, 400, 401, 461, 511, 518
Write set, 55
Write-after-read (WAR) hazard, 289, 290, 626
Write-after-write (WAW) hazard, 289, 290, 626
Write-broadcast protocol, 399
Write-invalidate protocol, 351, 354, 355, 357, 358, 396, 399, 479
Write-once protocol, 353–355
Write-update coherence protocol, 351, 355, 357, 358, 478, 479
Wu, C. L., 92, 96, 337, 393
Wulf, W. A., 41, 94, 96
Wyllie, J., 35, 36, 46
X-net mesh, 32, 33, 447, 454, 455, 470
X-window, 620, 622, 645
X-Y routing, 375, 384, 386–389, 391, 401
Xerox PARC Dragon, 355
Xu, Z., 322, 446, 469, 613
Yamaguchi, Y., 534, 540
Young, M. W., 712, 714
Zero index variable (ZIV), 571, 572, 574
Zima, H., 612
Zmob, 500, 539
Zuse, K., 4



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



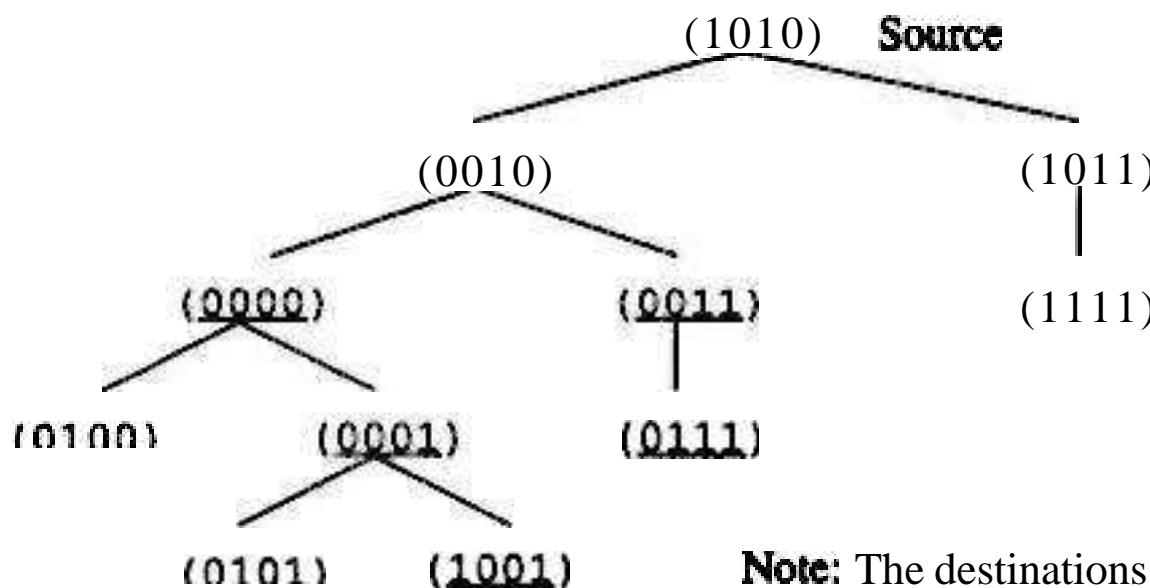
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Problem 6.17 (a) The four stages **perform**: Exponent subtract, Align, Fraction Add, and Normalize, **respectively**. (b) **111** cycles to add 100 floating-point numbers.

Problem 7.1 (a) Memory bandwidth = $\frac{mc}{(c+m)\tau}$ = 32 million words per second.
 (b) Memory utilization = $cm/(c + m)$ = 3.2 requests per memory cycle.

Problem 7.4 A minimum of 21 time steps are needed to schedule the 24 code segments on the 4 processors, which are updating the 6 memory modules simultaneously without conflicts. The average memory **bandwidth** is thus equal to $70/21=3.33$ words per time step, where 70 accounts for 70 memory accesses by four processors in 21 time steps without conflicts.

Problem 7.14 (a) **(101101) → (101100) → (101110) → (101010) → (111010) → (011010)** (b) Use either a route with a minimum of 20 channels and distance 9 or another route with a minimum distance of 8 and 22 **channels**. (c) The following tree shows multicast route on the hypercube:



Note: The destinations are underlined

Problem 8.12 (a) $R_s = 10/(10 - 9a)$ (in **Mflops**). (c) Vectorization ratio $a \cdot 26/27 = 0.963$. (d) $R_s = 3.5$ **Mflops**.

Problem 8.13 (a) Serial execution time = 190 time **units**. (b) **SIMD** execution time = 35 time units.

Problem 8.14 (a) C90 can execute 64 operations per cycle of 4.2ns, resulting a peak performance = $64/4.2 \times 10^{-9} = 15.2$ **Gflops**. (b) Similarly NEC has a peak performance - $64/2.9 \times 10^{-9} = 22$ **Gflops**.

Problem 9.1

(a) $E = 1/(1 + RL)$.

(b) $R' = (1 - h)R$, $E = 1/(1 + R'L) = 1/[1 + RL(1 - h)]$.

(c) While $N > N_d = \frac{L}{1/R'+C} 1$, $ft^* = \frac{1}{1+CR'}$ -

While $N < N_d$, $E_{\text{lin}} = \frac{N/R'}{1/R' + C + L} = \frac{1}{1 + R'C + R'L} = \frac{1}{1 + (1-h)(L+C)R}$.

(d) The mean **internode** distance $D = (r + 4)/3$.

Thus $L = 2Dt_d + t_m = \frac{2(r+4)}{3}t_d + t_m = \frac{2(\sqrt{p}+4)}{3}t_d + t_m$.

$$\frac{E_{\text{sat}}}{1} = \frac{1/R'}{1/R' + C} = \frac{1}{1 + (1-h)CR}$$

$$E_{\text{lin}} = \frac{1}{1 + (1-h)R(L+C)} = \frac{1}{1 + (1-h)R([\frac{2(\sqrt{p}+4)}{3}t_d + t_m] + C)}$$

Problem 10.5

(a) **A(5,8: V)** declares **A(5,8,1)**, **A(5,9,1)**, **A(5,10,1)**, **A(5,8,2)**, **A(5,9,2)**, **A(5,10,2)**, **A(5,8,3)**, **A(5,9,3)**, **A(5,10,3)**, **A(5,8,4)**, **A(5,9,4)**, **A(5,10,4)**, **A(5,8,5)**, **A(5,9,5)**, **A(5,10,5)**. **B(3:*:3,5:8)** corresponds to **B(3,5)**, **B(3,6)**, **B(3,7)**, **B(3,8)**, **B(6,5)**, **B(6,6)**, **B(6,7)**, **B(6,8)**, **B(9,5)**, **B(9,6)**, **B(9,7)**, **B(9,8)**. **C(*,3,4)** stands for **C(1,3,4)**, **C(2,3,4)**, **C(3,3,4)**.

(b) Yes, no, no, and yes respectively for the four array assignments.

Problem 10.7

(a) $S_1 = S_2 \leftarrow S_3$
 (b)

$$\begin{aligned} S_1 &: A(1:N) = B(1:N) \\ S_3 &: E(1:N) = C(2:N+1) \\ S_2 &: C(1:N) = A(1:N) + B(1:N) \end{aligned}$$

Problem 10.12

(a) Vectorized code:

$$\begin{aligned} \text{TEMP}(1:N) &= A(1:N) \\ A(2:N+1) &= \text{TEMP}(1:N) + 3.14159 \end{aligned}$$

(b) Parallelized code:

```

Doall I = 1, N
  If (A(I) .LE. 0.0) then
    S = S + B(I) * C(I)
    X = B(I)
  Endif
Enddo
```

Problem 11.15 (a) Suppose **the** image is partitioned **into** p segments, each consisting of $s - m/p$ rows. Vector **histog** is shared among the processors. Therefore its update has to be performed in a critical section to avoid race conditions. Assume it is possible

Kai Hwang has introduced the issues in designing and using high performance parallel computers at a time when a plethora of scalable computers utilizing commodity microprocessors offer higher peak performance than traditional vector supercomputers.. The book presents a balanced treatment of the theory, technology architecture and software of advanced computer systems. The emphasis on parallelism, scalability, and programmability makes this book rather unique and educational. I highly recommend Dr Hwang's timely book. I believe it will benefit many readers and be a fine reference.

C Gordon Bell

This book offers state-of-the-art principles and techniques for designing and programming parallel vector, and scalable computer systems. Written by a leading expert in the field, the authoritative text covers:

- Theory of parallelism — Parallel computer models, program and network properties, performance laws, and scalability analysis
- Advanced computer technology — RISC, CISC, Superscalar, VLIW and superpipelined processors, cache coherence, memory hierarchy, advanced pipelining, and system interconnects
- Parallel and scalable architectures — Multiprocessors, multicomputers, multivector and SIMD computers, and scalable multithreaded and dataflow architectures
- Software for parallel programming — Parallel models, languages, compilers, message passing, program development, synchronization, parallel UNIX extensions, and heterogeneous programming
- Illustrative examples and problems — Over 100 examples with solutions, 300 illustrations, and 200 homework problems involving designs, proofs, and analysis. Answers to selected problems are given. Solutions Manual available to instructors
- Case studies of real systems — Industrial computers from Cray, Intel, TMC, Fujitsu, NEC, Hitachi, IBM, DEC, MasPar, nCUBE, BBN, KSR, Tera, and Stardent, and experimental systems from Stanford, MIT, Caltech, Illinois, Wisconsin, USC, and ETL in Japan

The McGraw-Hill Companies



Tata McGraw-Hill
Publishing Company Limited
7 West Patel Nagar, New Delhi 110 008

ISBN-13

ISBN-10

978-0-07-053070-6

0-07-053070-X



Visit our website at : www.tatamcgrawhill.com

Limited preview ! Not for commercial use