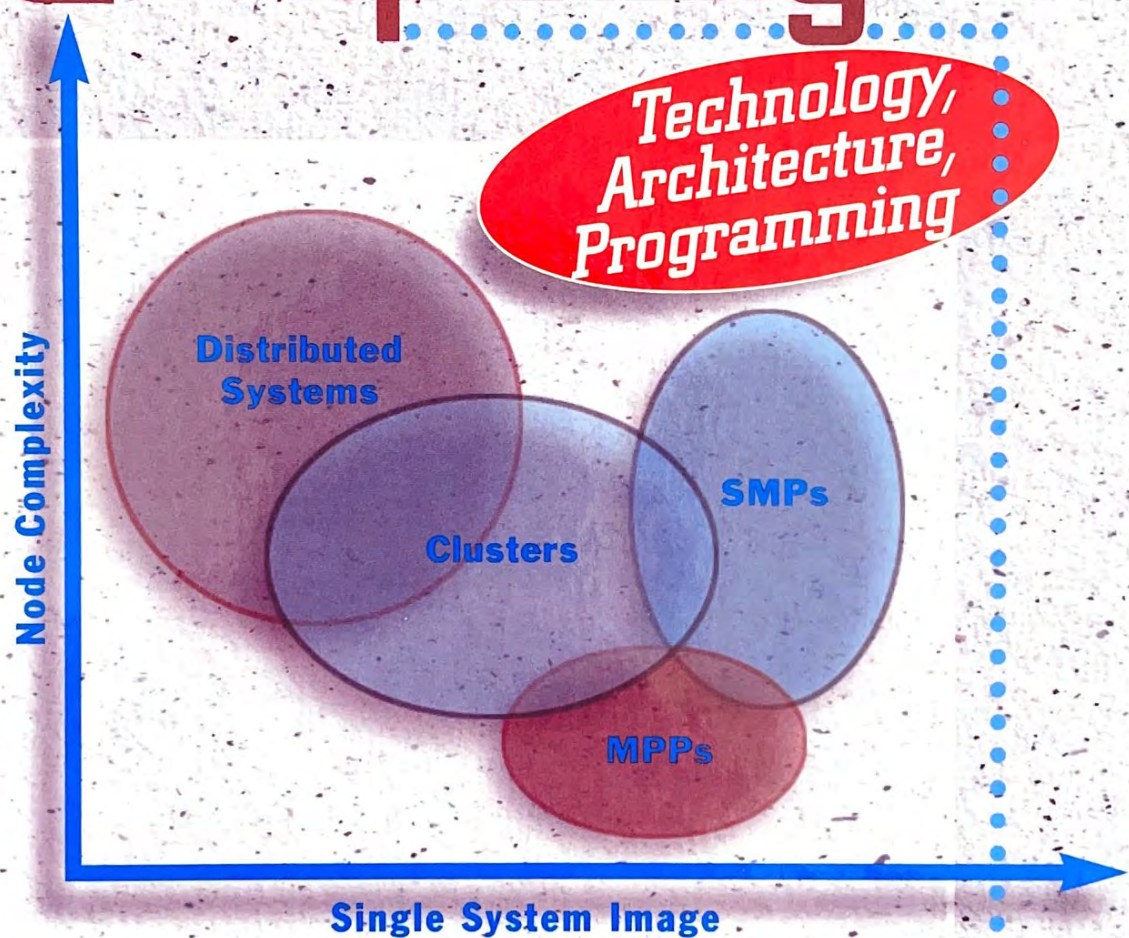


Scalable Parallel Computing



Kai Hwang • Zhiwei Xu

Scalable Parallel Computing

*Technology,
Architecture,
Programming*

Kai Hwang

*University of Hong Kong
University of Southern California*

Zhiwei Xu

*Chinese Academy of Sciences
University of Hong Kong*

**McGraw-Hill WCB
McGraw-Hill**

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Lisbon London Madrid
Mexico City Milan New Delhi Seoul Singapore Sydney Taipei Toronto

WCB/McGraw-Hill
A Division of The McGraw-Hill Companies

Scalable Parallel Computing
Technology, Architecture, Programming

Copyright © 1998, by The McGraw-Hill Companies, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC DOC 9 0 0 9 8

ISBN 0-07-031798-4

Editorial Director: Kevin Kane
Publisher: Tom Casson
Sponsoring editor: Lynn Cox
Marketing manager: John Wannemacher
Project manager: Richard DeVitto
Production supervisor: Richard DeVitto
Cover designer: Francis Owens
Editorial Assistant: Nina Kreiden
Printer: Quebecor Printing Dubuque

Library of Congress Cataloging-in-Publication Data

Hwang, Kai.

Scalable parallel computing : technology, architecture,
Programming / Kai Hwang, Zhiwei Xu.
p. cm.

Includes index.

ISBN 0-07-031798-4

1. Parallel processing (Electronic computers) 2. Computer
architecture. 3. Computer networks. I. Xu, Zhiwei. II. Title.
QA76.58H85 1997
004'.358—dc21

97-41663
CIP

<http://www.mhhe.com>

Trademark Notice

ANSI Standards X3T11, X3T9, and X3H5 are trademarks of the American National Standards Institute.

ALLICACHE and KSR-1 are trademarks of Kendall Square Research.

X Window, Alewife, J-Machine, and *T are trademarks of Massachusetts Institute of Technology.

CM-2, CM-5, C*, *Lisp, CM Fortran, and CMOST are trademarks of Thinking Machines Corporation.

HP/Convex Exemplar is a trademarks of Hewlett-Packard Company and Convex Computer Systems.

Cray 1, Cray 2, Y-MP, C90, CRAFT, T3D, T3E, UNICOS are trademarks of Cray Research, Inc.

Dash and SPLASH are trademarks of Stanford University.

Ethernet is a trademark of Xerox Corporation.

Gigabit Ethernet is trademark of the Gigabit Ethernet Alliance.

SCI, POSIX Threads, and Futurebus+ are trademarks of the Institute of Electrical and Electronic Engineering, Inc.

Illiac IV, Cedar, and Parafrase are trademarks of University of Illinois.

Intel 80x86, i960, i860, SHV, Paragon, Intel Hypercube, Pentium, Pentium Pro, and Pentium-II are trademarks of Intel Corporation.

MIPS R2000, R4000, and R10000 are trademarks of SGI/MIPS Technology, Inc.

Mach/OS, C.mmp, and Cm* are trademarks of Carnegie-Mellon University.

NYU Ultracomputer is a trademark of New York University.

SX/4 is a trademark of NEC Information Systems.

Symmetry and NUMA-Q are trademarks of Sequent Computers.

Gigaplane, Gigaplane-XB, S-Bus, TSO, PSO, NFS, SPARC, Ultra Enterprise, SPARCcluster, SPARCstation, and Solaris MC are trademarks

of Sun Microsystems, Inc.

Tera is a trademark of Tera Computer Systems.

Cosmic Cube and Mosaic C are trademarks of California Institute of Technology.

UNIX 4.3 BSD, NOW, GLUnix, IRAM, xFS, and 4.4BSD are trademarks of the University of California at Berkeley.

DEC, DECsafe, VAX, VAXcluster, VMS, Digital Unix, TruCluster, Digital NT Cluster, Alpha 21164, and Alpha AXP are trademarks of Digital Equipment Corporation.

Apollo Domain, HP, Series 9000, PA-RISC, and UX are trademarks of the Hewlett-Packard Company.

IBM PC, AIX, ESSL, 801, RP3, EUI, HACMP, IBM, LoadLeveller, NetView, POWER, PowerPC, RS/6000, S/370, S/390, SP2, SP, and Sysplex are trademarks of International Business Machines Corporation.

Lifekeeper, NCR and Teradata are trademarks of National Cash Register.

M680x0 is trademark of Motorola Corporation.

OSF/1, DCE, and DFS are trademarks of the Open Software Foundation, Inc.

Oracle Parallel Server is a trademark of Oracle Corporation.

Express is a trademark of Parasoft Corporation.

Load Sharing Facility is a trademark of Platform Computing Corporation.

Origin 2000, Power C, POWER CHALLENGEarray, and Cellular IRIX are trademarks of Silicon Graphics Computer Systems.

Sybase is a trademark of Sybase, Inc.

UNIX and SVR5 are trademarks of X/Open Company, Ltd.

Microsoft, Wolfpack, and Windows NT are trademarks of Microsoft Corporation.

OpenMP is a trademark of OpenMP Standards Evaluation Board.

Myrinet is a trademark of Myricom, Inc.

About the Authors

Kai Hwang presently holds a Chair Professor of Computer Engineering at the University of Hong Kong (HKU), while taking a research leave from the University of Southern California (USC). He has been engaged in higher education and computer research for 26 years, after earning the Ph.D. in Electrical Engineering and Computer Science from the University of California at Berkeley. His work on this book started at the USC and was mostly completed at the HKU.

An IEEE Fellow, he has published extensively in the areas of computer architecture, digital arithmetic, parallel processing, and distributed computing. He is the founding Editor-in-Chief of the *Journal of Parallel and Distributed Computing*. He has chaired the international conferences: ICPP86, ARITH-7, IPPS96, ICAPP 96, and HPCA-4 in 1998. He has received several achievement awards for outstanding research and academic contributions to the field of parallel computing.

He has lectured worldwide and performed consulting and advisory work for US National Academy of Sciences, MIT Lincoln Laboratory, IBM Fishkill, TriTech in Singapore, Fujitsu and ETL in Japan, GMD in Germany, CERN School of Computing, and Academia Sinica in China. Presently, he leads a research group at HKU in developing an ATM-based multicomputer cluster for high-performance computing and distributed multimedia, Intranet, and Internet applications.

Zhiwei Xu is a Professor and Chief Architect at the National Center for Intelligent Computing Systems (NCIC), Chinese Academy of Sciences, Beijing, China. He is also a Honorary Research Fellow of HKU. He received a Ph.D. in Computer Engineering from the University of Southern California. He participated in the STAP/MPP benchmark projects led by Dr. Hwang at USC and HKU during the past 5 years.

He has taught at the Rutgers University and New York Polytechnic University. He has published in the areas of parallel languages, pipelined vector processing, and benchmark evaluation of massively parallel processors. Presently, he leads a design group at NCIC in building a series of cluster-based superservers in China. His current research interest lies mainly in network-based cluster computing and the software environments for parallel programming.

Dedication

To my family, for their love and support -- k.h.

**To my high-school teacher,
Ms. Zhang Dahua, who taught me the
serenity of science at a tumultuous time -- z.x.**

Table of Contents

About the Authors viii

Foreword xix

Preface xx

Guide to Instructors/Readers xxiii

Part I Scalability and Clustering 1

Chapter 1 Scalable Computer Platforms and Models 3

- 1.1 Evolution of Computer Architecture 5
 - 1.1.1 Computer Generations 5
 - 1.1.2 Scalable Computer Architectures 6
 - 1.1.3 Converging System Architectures 8
- 1.2 Dimensions of Scalability 9
 - 1.2.1 Resource Scalability 9
 - 1.2.2 Application Scalability 11
 - 1.2.3 Technology Scalability 12
- 1.3 Parallel Computer Models 13
 - 1.3.1 Semantic Attributes 14
 - 1.3.2 Performance Attributes 17
 - 1.3.3 Abstract Machine Models 18
 - 1.3.4 Physical Machine Models 26
- 1.4 Basic Concepts of Clustering 30
 - 1.4.1 Cluster Characteristics 30
 - 1.4.2 Architectural Comparisons 31
 - 1.4.3 Benefits and Difficulties of Clusters 32
- 1.5 Scalable Design Principles 37
 - 1.5.1 Principle of Independence 37
 - 1.5.2 Principle of Balanced Design 39
 - 1.5.3 Design for Scalability 44
- 1.6 Bibliographic Notes and Problems 47

Chapter 2 Basics of Parallel Programming 51

- 2.1 Parallel Programming Overview 51
 - 2.1.1 Why Is Parallel Programming Difficult? 52
 - 2.1.2 Parallel Programming Environments 55
 - 2.1.3 Parallel Programming Approaches 56
- 2.2 Processes, Tasks, and Threads 59

2.2.1	Definitions of an Abstract Process	59
2.2.2	Execution Mode	62
2.2.3	Address Space	63
2.2.4	Process Context	65
2.2.5	Process Descriptor	66
2.2.6	Process Control	67
2.2.7	Variations of Process	70
2.3	Parallelism Issues	71
2.3.1	Homogeneity in Processes	72
2.3.2	Static versus Dynamic Parallelism	74
2.3.3	Process Grouping	75
2.3.4	Allocation Issues	76
2.4	Interaction/Communication Issues	77
2.4.1	Interaction Operations	77
2.4.2	Interaction Modes	80
2.4.3	Interaction Patterns	82
2.4.4	Cooperative versus Competitive Interactions	84
2.5	Semantic Issues in Parallel Programs	85
2.5.1	Program Termination	85
2.5.2	Determinacy of Programs	86
2.6	Bibliographic Notes and Problems	87
Chapter 3	Performance Metrics and Benchmarks	91
3.1	System and Application Benchmarks	91
3.1.1	Micro Benchmarks	92
3.1.2	Parallel Computing Benchmarks	96
3.1.3	Business and TPC Benchmarks	98
3.1.4	SPEC Benchmark Family	100
3.2	Performance versus Cost	102
3.2.1	Execution Time and Throughput	103
3.2.2	Utilization and Cost-Effectiveness	104
3.3	Basic Performance Metrics	108
3.3.1	Workload and Speed Metrics	108
3.3.2	Caveats in Sequential Performance	111
3.4	Performance of Parallel Computers	113
3.4.1	Computational Characteristics	113
3.4.2	Parallelism and Interaction Overheads	115
3.4.3	Overhead Quantification	118
3.5	Performance of Parallel Programs	126
3.5.1	Performance Metrics	126
3.5.2	Available Parallelism in Benchmarks	131
3.6	Scalability and Speedup Analysis	134
3.6.1	Amdahl's Law: Fixed Problem Size	134
3.6.2	Gustafson's Law: Fixed Time	136
3.6.3	Sun and Ni's Law: Memory Bounding	139
3.6.4	Isoperformance Models	144
3.7	Bibliographic Notes and Problems	148

Part II Enabling Technologies 153**Chapter 4 Microprocessors as Building Blocks 155**

- 4.1 System Development Trends 155**
 - 4.1.1 Advances in Hardware 156
 - 4.1.2 Advances in Software 159
 - 4.1.3 Advances in Applications 160
- 4.2 Principles of Processor Design 164**
 - 4.2.1 Basics of Instruction Pipeline 164
 - 4.2.2 From CISC to RISC and Beyond 169
 - 4.2.3 Architectural Enhancement Approaches 172
- 4.3 Microprocessor Architecture Families 174**
 - 4.3.1 Major Architecture Families 174
 - 4.3.2 Superscalar versus Superpipelined Processors 175
 - 4.3.3 Embedded Microprocessors 180
- 4.4 Case Studies of Microprocessors 182**
 - 4.4.1 Digital's Alpha 21164 Microprocessor 182
 - 4.4.2 Intel Pentium Pro Processor 186
- 4.5 Post-RISC, Multimedia, and VLIW 191**
 - 4.5.1 Post-RISC Processor Features 191
 - 4.5.2 Multimedia Extensions 195
 - 4.5.3 The VLIW Architecture 199
- 4.6 The Future of Microprocessors 201**
 - 4.6.1 Hardware Trends and Physical Limits 201
 - 4.6.2 Future Workloads and Challenges 203
 - 4.6.3 Future Microprocessor Architectures 204
- 4.7 Bibliographic Notes and Problems 206**

Chapter 5 Distributed Memory and Latency Tolerance 211

- 5.1 Hierarchical Memory Technology 211**
 - 5.1.1 Characteristics of Storage Devices 211
 - 5.1.2 Memory Hierarchy Properties 214
 - 5.1.3 Memory Capacity Planning 217
- 5.2 Cache Coherence Protocols 220**
 - 5.2.1 Cache Coherency Problem 220
 - 5.2.2 Snoopy Coherency Protocols 222
 - 5.2.3 The MESI Snoopy Protocol 224
- 5.3 Shared-Memory Consistency 228**
 - 5.3.1 Memory Event Ordering 228
 - 5.3.2 Memory Consistency Models 231
 - 5.3.3 Relaxed Memory Models 234
- 5.4 Distributed Cache/Memory Architecture 237**
 - 5.4.1 NORMA, NUMA, COMA, and DSM Models 237
 - 5.4.2 Directory-Based Coherency Protocol 243
 - 5.4.3 The Stanford Dash Multiprocessor 245
 - 5.4.4 Directory-Based Protocol in Dash 248

5.5	Latency Tolerance Techniques	250
5.5.1	Latency Avoidance, Reduction, and Hiding	250
5.5.2	Distributed Coherent Caches	253
5.5.3	Data Prefetching Strategies	255
5.5.4	Effects of Relaxed Memory Consistency	257
5.6	Multithreaded Latency Hiding	257
5.6.1	Multithreaded Processor Model	258
5.6.2	Context-Switching Policies	260
5.6.3	Combining Latency Hiding Mechanisms	265
5.7	Bibliographic Notes and Problems	266
Chapter 6 System Interconnects and Gigabit Networks		273
6.1	Basics of Interconnection Network	273
6.1.1	Interconnection Environments	273
6.1.2	Network Components	276
6.1.3	Network Characteristics	277
6.1.4	Network Performance Metrics	280
6.2	Network Topologies and Properties	281
6.2.1	Topological and Functional Properties	281
6.2.2	Routing Schemes and Functions	283
6.2.3	Networking Topologies	286
6.3	Buses, Crossbar, and Multistage Switches	294
6.3.1	Multiprocessor Buses	294
6.3.2	Crossbar Switches	298
6.3.3	Multistage Interconnection Networks	301
6.3.4	Comparison of Switched Interconnects	305
6.4	Gigabit Network Technologies	307
6.4.1	Fiber Channel and FDDI Rings	307
6.4.2	Fast Ethernet and Gigabit Ethernet	310
6.4.3	Myrinet for SAN/LAN Construction	313
6.4.4	HiPPI and SuperHiPPI	314
6.5	ATM Switches and Networks	318
6.5.1	ATM Technology	318
6.5.2	ATM Network Interfaces	320
6.5.3	Four Layers of ATM Architecture	321
6.5.4	ATM Internetwork Connectivity	324
6.6	Scalable Coherence Interface	326
6.6.1	SCI Interconnects	327
6.6.2	Implementation Issues	329
6.6.3	SCI Coherence Protocol	332
6.7	Comparison of Network Technologies	334
6.7.1	Standard Networks and Perspectives	334
6.7.2	Network Performance and Applications	335
6.8	Bibliographic Notes and Problems	337
Chapter 7 Threading, Synchronization, and Communication		343
7.1	Software Multithreading	343

7.1.1	The Thread Concept	344
7.1.2	Threads Management	346
7.1.3	Thread Synchronization	348
7.2	Synchronization Mechanisms	349
7.2.1	Atomicity versus Mutual Exclusion	349
7.2.2	High-Level Synchronization Constructs	355
7.2.3	Low-Level Synchronization Primitives	360
7.2.4	Fast Locking Mechanisms	364
7.3	The TCP/IP Communication Protocol Suite	366
7.3.1	Features of The TCP/IP Suite	367
7.3.2	UDP, TCP, and IP	371
7.3.3	The Sockets Interface	375
7.4	Fast and Efficient Communication	376
7.4.1	Key Problems in Communication	377
7.4.2	The Log P Communication Model	384
7.4.3	Low-Level Communications Support	386
7.4.4	Communication Algorithms	396
7.5	Bibliographic Notes and Problems	398

Part III Systems Architecture 403

Chapter 8 Symmetric and CC-NUMA Multiprocessors 407

8.1	SMP and CC-NUMA Technology	407
8.1.1	Multiprocessor Architecture	407
8.1.2	Commercial SMP Servers	412
8.1.3	The Intel SHV Server Board	413
8.2	Sun Ultra Enterprise 10000 System	416
8.2.1	The Ultra E-10000 Architecture	416
8.2.2	System Board Architecture	418
8.2.3	Scalability and Availability Support	418
8.2.4	Dynamic Domains and Performance	420
8.3	HP/Convex Exemplar X-Class	421
8.3.1	The Exemplar X System Architecture	421
8.3.2	Exemplar Software Environment	424
8.4	The Sequent NUMA-Q 2000	425
8.4.1	The NUMA-Q 2000 Architecture	426
8.4.2	Software Environment of NUMA-Q	430
8.4.3	Performance of the NUMA-Q	431
8.5	The SGI/Cray Origin 2000 Superserver	434
8.5.1	Design Goals of Origin 2000 Series	434
8.5.2	The Origin 2000 Architecture	435
8.5.3	The Cellular IRIX Environment	443
8.5.4	Performance of the Origin 2000	447
8.6	Comparison of CC-NUMA Architectures	447
8.7	Bibliographic Notes and Problems	451

Chapter 9 Support of Clustering and Availability 453

- 9.1 Challenges in Clustering 453**
 - 9.1.1 Classification of Clusters 453
 - 9.1.2 Cluster Architectures 456
 - 9.1.3 Cluster Design Issues 457
- 9.2 Availability Support for Clustering 459**
 - 9.2.1 The Availability Concept 460
 - 9.2.2 Availability Techniques 463
 - 9.2.3 Checkpointing and Failure Recovery 468
- 9.3 Support for Single System Image 473**
 - 9.3.1 Single System Image Layers 473
 - 9.3.2 Single Entry and Single File Hierarchy 475
 - 9.3.3 Single I/O, Networking, and Memory Space 479
- 9.4 Single System Image in Solaris MC 482**
 - 9.4.1 Global File System 482
 - 9.4.2 Global Process Management 484
 - 9.4.3 Single I/O System Image 485
- 9.5 Job Management in Clusters 486**
 - 9.5.1 Job Management System 486
 - 9.5.2 Survey of Job Management Systems 492
 - 9.5.3 Load-Sharing Facility (LSF) 494
- 9.6 Bibliographic Notes and Problems 501**

Chapter 10 Clusters of Servers and Workstations 505

- 10.1 Cluster Products and Research Projects 505**
 - 10.1.1 Supporting Trend of Cluster Products 506
 - 10.1.2 Cluster of SMP Servers 508
 - 10.1.3 Cluster Research Projects 509
- 10.2 Microsoft Wolfpack for NT Clusters 511**
 - 10.2.1 Microsoft Wolfpack Configurations 512
 - 10.2.2 Hot Standby Multiserver Clusters 513
 - 10.2.3 Active Availability Clusters 514
 - 10.2.4 Fault-Tolerant Multiserver Cluster 516
- 10.3 The IBM SP System 518**
 - 10.3.1 Design Goals and Strategies 518
 - 10.3.2 The SP2 System Architecture 521
 - 10.3.3 I/O and Internetworking. 523
 - 10.3.4 The SP System Software 526
 - 10.3.5 The SP2 and Beyond 530
- 10.4 The Digital TruCluster 531**
 - 10.4.1 The TruCluster Architecture 531
 - 10.4.2 The Memory Channel Interconnect 534
 - 10.4.3 Programming the TruCluster 537
 - 10.4.4 The TruCluster System Software 540
- 10.5 The Berkeley NOW Project 541**
 - 10.5.1 Active Messages for Fast Communication 541

10.5.2	GLUnix for Global Resource Management	547
10.5.3	The xFS Serverless Network File System	549
10.6	TreadMarks: A Software-Implemented DSM Cluster	556
10.6.1	Boundary Conditions	556
10.6.2	User Interface for DSM	557
10.6.3	Implementation Issues	559
10.7	Bibliographic Notes and Problems	561
Chapter 11	MPP Architecture and Performance	565
11.1	An Overview of MPP Technology	565
11.1.1	MPP Characteristics and Issues	565
11.1.2	MPP Systems – An Overview	569
11.2	The Cray T3E System	570
11.2.1	The System Architecture of T3E	571
11.2.2	The System Software in T3E	573
11.3	New Generation of ASCI/MPPs	574
11.3.1	ASCI Scalable Design Strategy	574
11.3.2	Hardware and Software Requirements	576
11.3.3	Contracted ASCI/MPP Platforms	577
11.4	Intel/Sandia ASCI Option Red	579
11.4.1	The Option Red Architecture	579
11.4.2	Option Red System Software	582
11.5	Parallel NAS Benchmark Results	584
11.5.1	The NAS Parallel Benchmarks	585
11.5.2	Superstep Structure and Granularity	586
11.5.3	Memory, I/O, and Communications	587
11.6	MPI and STAP Benchmark Results	590
11.6.1	MPI Performance Measurements	590
11.6.2	MPI Latency and Aggregate Bandwidth	592
11.6.3	STAP Benchmark Evaluation of MPPs	594
11.6.4	MPP Architectural Implications	600
11.7	Bibliographic Notes and Problems	603
Part IV	Parallel Programming	607
Chapter 12	Parallel Paradigms and Programming Models	609
12.1	Paradigms and Programmability	609
12.1.1	Algorithmic Paradigms	609
12.1.2	Programmability Issues	612
12.1.3	Parallel Programming Examples	614
12.2	Parallel Programming Models	617
12.2.1	Implicit Parallelism	617
12.2.2	Explicit Parallel Models	621
12.2.3	Comparison of Four Models	624
12.2.4	Other Parallel Programming Models	627

12.3	Shared-Memory Programming	629
12.3.1	The ANSI X3H5 Shared-Memory Model	629
12.3.2	The POSIX Threads (Pthreads) Model	634
12.3.3	The OpenMP Standard	636
12.3.4	The SGI Power C Model	640
12.3.5	C//: A Structured Parallel C Language	643
12.4	Bibliographic Notes and Problems	649
Chapter 13	Message-Passing Programming	653
13.1	The Message-Passing Paradigm	653
13.1.1	Message-Passing Libraries	653
13.1.2	Message-Passing Modes	655
13.2	Message-Passing Interface (MPI)	658
13.2.1	MPI Messages	661
13.2.2	Message Envelope in MPI	668
13.2.3	Point-to-Point Communications	674
13.2.4	Collective MPI Communications	678
13.2.5	The MPI-2 Extensions	682
13.3	Parallel Virtual Machine (PVM)	686
13.3.1	Virtual Machine Construction	687
13.3.2	Process Management in PVM	689
13.3.3	Communication with PVM	693
13.4	Bibliographic Notes and Problems	699
Chapter 14	Data-Parallel Programming	705
14.1	The Data-Parallel Model	705
14.2	The Fortran 90 Approach	706
14.2.1	Parallel Array Operations	706
14.2.2	Intrinsic Functions in Fortran 90	708
14.3	High-Performance Fortran	711
14.3.1	Support for Data Parallelism	712
14.3.2	Data Mapping in HPF	715
14.3.3	Summary of Fortran 90 and HPF	721
14.4	Other Data-Parallel Approaches	725
14.4.1	Fortran 95 and Fortran 2001	725
14.4.2	The pC++ and Nesl Approaches	728
14.5	Bibliographic Notes and Problems	733
Bibliography		737
Web Resources List		765
Subject Index		787
Author Index		799

Chapter 1

Scalable Computer Platforms and Models

This chapter presents basic models of parallel and cluster computers. Fundamental design issues and operational principles of scalable computer platforms are introduced. We review the computer technology over the last 50 years. Scalable and cluster computer systems are modeled with key architectural distinctions. Scalability will be introduced in three orthogonal dimensions: *resource*, *application*, and *technology*.

Abstract and physical machine models are specified in Section 1.3. In Section 1.4, we introduce basic concepts of multicomputer clustering. The differences among symmetric multiprocessors, clusters of computers, and distributed computer systems are clarified. Three basic principles are studied in Section 1.5 to guide the design and application of scalable parallel computers.

Bits, Bytes, and Words The following units are widely used in the computer field, but sometimes were wrongly used with confusing notations and ambiguous meanings. To cope with this problem, we present below a set of notations that will be used throughout the book. In particular, readers should not be confused with the shorthand notations for basic units of *time*, *byte*, and *bit* respectively.

The basic unit in time is *second*, abbreviated as s. The two basic information units are *byte* and *bit*. One byte (1 B) is 8 bits (8 b). Byte is always abbreviated as B and bit as b. Other information units are *word* (16 b or 2 B), *doubleword* (32 b or 4 B), and *quadword* (64 b or 8 B). This is based on convention used by Intel, Motorola, and Digital Equipment.

Mainframe vendors consider a word to have 32 b. Some supercomputer designers consider 64 b in a word. A frequently used workload unit is the number of *floating-point operations*, abbreviated as *flop*. A unit for computing speed is the number of *floating-point operations per second* (flop/s). A unit for information transfer rate is the number of *bytes per second* (B/s). The execution rate of a processor is often measured as *million instructions per second* (MIPS), which is equivalent to the notation Mi/s used in Europe.

Notations and Conventions We will use the conventions and notations specified in Table 1.1. Note that decimal and binary interpretations differ to some extent in the high-order units. Readers should be aware of their differences.

For instance, a *kilo* could mean either $10^3 = 1000$ or $2^{10} = 1024$. In computer science, K is usually used to mean 1024; in other scientific disciplines, k denotes 1000. This ambiguity must be clarified, even the relative difference is small. In general, the binary interpretation is applied to information units such as bits, bytes, or words. The decimal interpretation is often used in reference to other units such as execution time, power, weight, pin count, and computational workload, etc.

The convention for plurals is as follows: The unit will be pluralized when the whole word is used, and not pluralized when an abbreviation is used. For instance, we say 2 KB, 2 kilobytes, or even 2 Kbytes. They all mean the same thing: 2048 bytes. But we do not use 2 KBs or 2 Kbyte. Note that 2 Kb means a completely different thing: 2048 bits, or 256 bytes.

Similarly, 2 Mflop/s is a proper notation, meaning a speed of executing 2 million floating-point operations per second. We should avoid using 2 Mflops/s, 2 MFLOPS, or 2 Mflops, even some people use them. Throughout this book, the logarithmic function is always base 2. That is, $\log n$ means $\log_2 n$, unless otherwise specified.

Table 1.1 Notations and Conventions for Numbers

Prefix	Abbreviation	Meaning	Numeric Value
milli	m	One thousandth	10^{-3}
micro	μ	One millionth	10^{-6}
nano	n	One billionth	10^{-9}
pico	p	One trillionth	10^{-12}
femto	f	One quadrillionth	10^{-15}
atta	a	One quintillionth	10^{-18}
kilo	K (or k)	Thousand	10^3 (or 2^{10})
mega	M	Million	10^6 or 2^{20}
giga	G	Billion	10^9 or 2^{30}
tera	T	Trillion	10^{12} or 2^{40}
peta	P	Quadrillion	10^{15} or 2^{50}
exa	E	Quintillion	10^{18} or 2^{60}

1.1 Evolution of Computer Architecture

Computers have gone through 50 years of development. We summarize below five generations of development, approximately 10 to 15 years per generation, since the 1940s. A formal definition of a scalable computer is given below. Then we describe the converging architecture for future parallel computers.

1.1.1 Computer Generations

Over the past 50 years, digital electronic computers have gone through five generations of development. Table 1.2 provides a summary of the technology, architecture, software, application, and representative systems in each generation of computers. Each generation has improved from the previous generations in hardware and software technologies used and in application levels.

Table 1.2 Computer Generations in 50 Years

Generation Period	Technology and Architecture	Software and Operating System	Representative Systems
First 1946-1956	Vacuum tubes and relay memory, single-bit CPU with accumulator-based instruction set	Machine/assembly languages, programs without subroutines	ENIAC, IBM 701, Princeton IAS
Second 1956-1967	Discrete transistors, core memory, floating-point accelerator, I/O channels	Algol and Fortran with compilers, batch processing OS	IBM 7030, CDC 1604, Univac LARC
Third 1967-1978	Integrated circuits, pipelined CPU, microprogrammed control unit	C language, multiprogramming, timesharing OS	PDP-11, IBM 360/370, CDC 6600
Fourth 1978-1989	VLSI microprocessors, solid-state memory, multiprocessors, vector supercomputers	Symmetric multiprocessing, parallelizing compilers, message-passing libraries	IBM PC, VAX 9000, Cray X/MP
Fifth 1990- present	ULSI circuits, scalable parallel computers, workstation clusters, Intranet, Internet	Java, microkernels, Multithreading, distributed OS, World-Wide Web	IBM SP2, SGI Origin 2000, Digital TruCluster

In hardware technology, the first generation used vacuum tubes and relay memory. The second generation was marked by the use of discrete transistors and core memory.

The third generation began to use *small-scale integrated* (SSI) circuits. The fourth generation began with the use of *very large-scale integrated* (VLSI) microprocessors. The fifth generation uses *ultra large-scale integrated* (ULSI) circuits. For example, in 1997, 10M-transistor microprocessors and 256M-transistor DRAMs (*dynamic random-access memories*) are in use.

In the software area, the first generation used machine/assembly languages. The second generation began to use *high-level languages* (HLLs), such as Algol and Fortran. The third generation was marked by the use of C language, multiprogramming and time-sharing operating systems. The fourth generation emphasized parallel and vector processing. The fifth generation emphasizes scalable and clustered computing as covered in this book. As we cross into the 21st century, future generations of computers will evolve with even higher capability and user friendliness.

1.1.2 Scalable Computer Architectures

A common feature of modern computers is parallelism. Even within a single processor, parallelism has been exploited in many ways. This will be discussed in detail in Chapter 4. More recently, the concept of *scalability* has been reemphasized, which includes parallelism. The scalability concept is central to this book and will be elaborated on later. Given below is a short definition of scalable systems.

Definition 1.1 A computer system, including all its hardware and software resources, is called *scalable* if it can *scale up* (i.e., improve its resources) to accommodate ever-increasing performance and functionality demand and/or *scale down* (i.e., decrease its resources) to reduce cost. ■

Although we will focus on the scaling-up aspect most of the time, scalability is not equivalent to being big. Scalability involves the ability to scale down. Scaling issues arise even on a uniprocessor system. More specifically, saying that a system is scalable implies the following:

- **Functionality and Performance:** The scaled-up system should provide more functionality or better performance. The total computing power of the system should increase proportionally to the increase in resources. Ideally, the users would like to see computing power increase close to n times when the system resource is improved n times.
- **Scaling in Cost:** The cost paid for scaling up must be reasonable. A rule of thumb is that scaling up n times should incur a cost of no more than n or $n \log n$ times.
- **Compatibility:** The same components, including hardware, system software, and application software, should still be usable with little change. It is unreasonable to

expect the users to pay for a completely new operating system and to redevelop their application codes. Often scaling involves only a part of a system, e.g., adding more processors or upgrading a processor to the next-generation. The improvement should be compatible with the rest of the system. That is, the old memory, disk, interconnect, and peripherals should still be usable.

The Computer Pyramid The main motivation for a scalable system is to provide a flexible, cost-effective information processing tool. As illustrated in Fig. 1.1, the computer classes form a pyramid with respect to sales volume and performance and cost. At the bottom of the pyramid is the class of personal computers (PCs), which have a quite affordable price and a huge annual sales volume. Their performance is also the lowest.

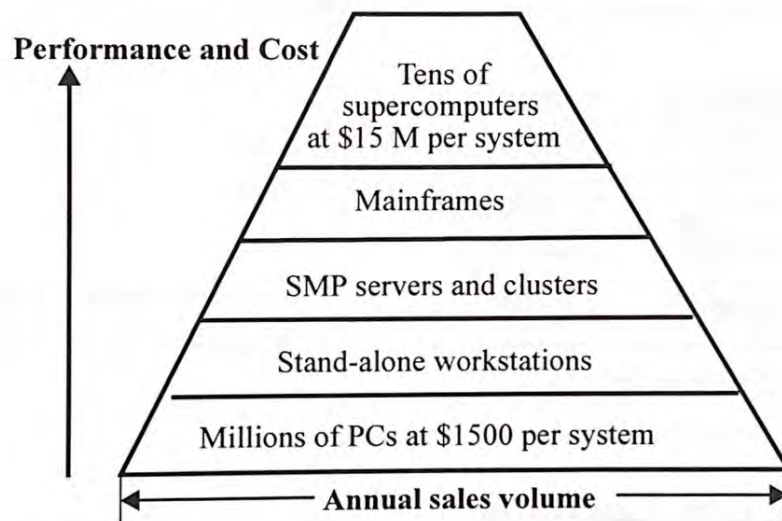


Figure 1.1 The pyramid of computer classes.

At the top is the class of supercomputers. Their sales volume is very low because of the expensive price. However, a supercomputer integrates many resources into a single system, employs cutting-edge technology, and has the highest performance.

The scalability concept advocates a common scalable architecture across all computer classes. This has several advantages:

- It satisfies different performance and cost requirements. For instance, a user could first purchase a low-end system. When his performance requirement increases, he could scale up the system, knowing that the original software and hardware components can still be used in the new system, and his investment is protected.

- High-end machines may use low-end components to cut cost. For example, with high volume, PCs have low-cost, off-the-shelf components. With a scalable architecture, supercomputers can use such components to reduce system cost. In fact, using commodity components (processors, memory chips, disks, I/O controllers, etc.) has become a trend in high-performance systems development.
- The cutting-edge technology developed for high-end systems may eventually migrate down the pyramid to improve the performance of low-end systems, if cost-effectiveness can be increased with improved production technology.

1.1.3 Converging System Architectures

Scalable parallel computers are converging toward three architectures, which share much commonality with increasing levels of resources sharing as shown in Fig. 1.2.

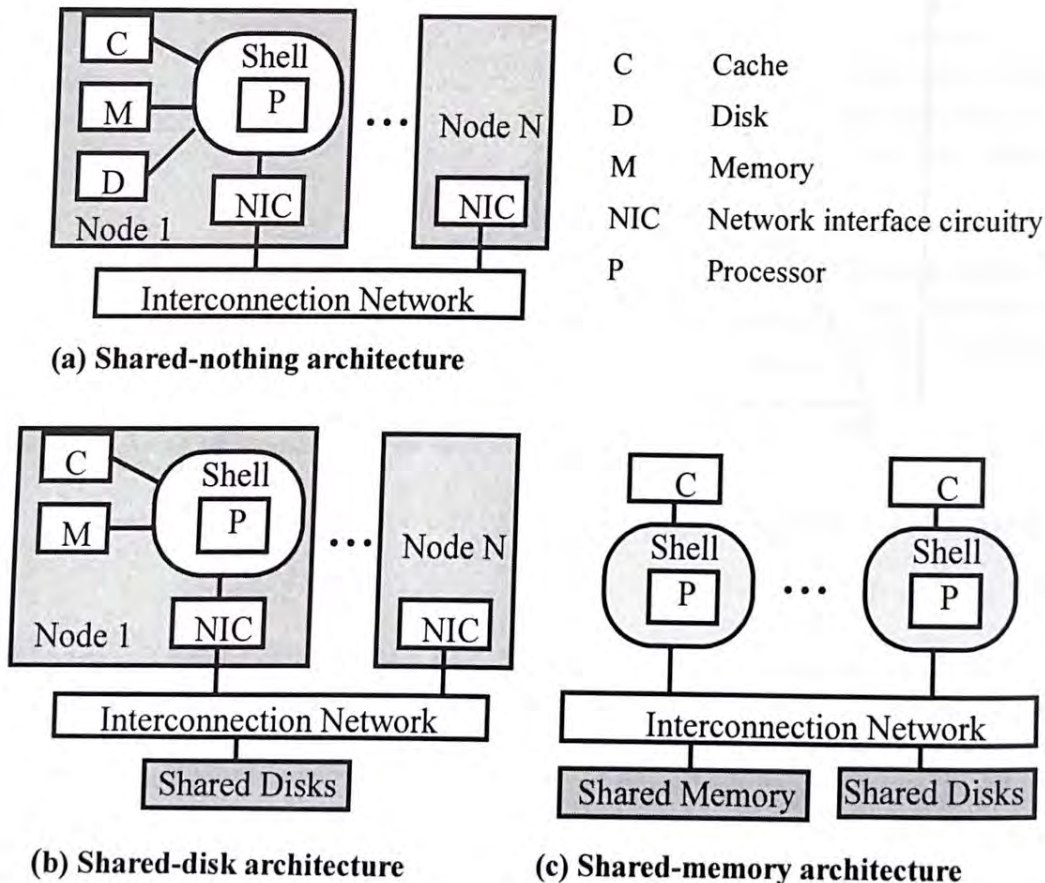


Figure 1.2 Common architecture for scalable parallel computers.

The *shared-nothing* architecture consists of a number of *nodes* connected by an *inter-connection network*. The node usually follows a *shell* architecture [571], where a custom-designed circuitry (called the *shell*) interfaces a commodity microprocessor to the rest of the node, including a (board-level) cache, a local memory, a *network interface circuitry* (NIC), and a disk. There may be more than one processor in a node.

The *shared-disk* architecture differs from the shared-nothing architecture in that the disk modules are moved out of the nodes to be shared among the nodes. In the *shared-memory* architecture, even the main memory becomes shared.

Macro-Architecture vs. Micro-Architecture The overall structure of a computer system (e.g., as shown in Fig. 1.2) is called the *macro-architecture*, while the processor and the shell surrounding it are called the *microarchitecture*. An advantage of the shell architecture is that when the processor is upgraded to the next generation or changed to a different architecture, only the shell (or the microarchitecture) needs to be changed.

All of the above architectural concepts are simplifications of real parallel computers. For instance, shared memory or shared disks can be attached to the interconnect of a shared-nothing architecture [Fig. 1.2(a)]. This will lead to the architectures with both local and shared disk and/or shared memory as shown in Fig. 1.2(b-c). More clarification will be given in later chapters.

1.2 Dimensions of Scalability

There are different ways of scaling (improving or reducing system resources), which leads to different aspects and definitions of scalability [69, 513]. In the following, we will focus on scaling up. We first need to define several basic terms:

A *processor* refers to a CPU, including one or more instruction pipelines, floating-point units, and caches, among other things. A processor may be implemented on a single chip (die), on multiple dies mounted on a single multichip module, or as multiple, separate chips. A *node* may have one or more processors, a local memory, and other circuitry. For simplicity, we will assume single-processor nodes in the models shown in Fig. 1.2, unless otherwise specified.

1.2.1 Resource Scalability

Resource scalability refers to gaining higher performance or functionality by increasing the *machine size* (i.e., the number of processors), investing in more storage (cache, main memory, disks), improving the software, etc.

Size Scalability The most obvious way to scale up a computer system is to increase the *machine size*, i.e., the number of processors. Size scalability measures the maximum

number of processors a system can accommodate. Not all parallel computers are equally size-scalable. For instance, as of 1997, a *symmetric multiprocessor* (SMP) system can scale up to about 64 processors, while an IBM SP2 can scale up to 512 processors.

In current parallel computers, size scaling is often not just a simple matter of adding processors. The communication subsystem, including interconnect, interface, and communication software, may also need to be improved. How to effectively utilize the greater hardware parallelism, i.e., how to program the scaled-up system, is another concern.

Two main factors limiting the scalability of parallel systems are *programming* and *communication*. We will study these issues extensively in later chapters and learn how current commercial and research systems address them effectively.

Scaling Up in Resources While adding processors is probably the most obvious way to scale up a system, it is not the only way. One can keep the same number of processors, but invest in more memory, bigger off-chip caches, bigger disks, etc.

Example 1.1 Memory requirement in the IBM SP2

When the Maui High-Performance Computing Center (MHPCC) decided to upgrade its 400-node SP2 system, it opted to increase the memory and disk capacity, instead of adding more nodes. The scaled-up storage capacity is summarized in Table 1.3. This table says that, for instance, there are eight wide nodes in group 1, each having a 1-GB main memory and a 4.5-GB local disk. Also 1000 MB is used as the paging space for swapping, and another 2000 MB is used as a local scratch space.

Table 1.3 Memory and Disk Capacity of the IBM SP2 at MHPCC

Group	Node Type	Nodes	Memory (MB)	Disk (GB)	Paging Space (MB)	Scratch Space (MB)
1	Wide	8	1024	4.5	1000	2000
2	Wide	56	256	4.5	500	2000
3	Wide	16	256	2.0	500	1000
4	Thin	160	128	2.0	256	1000
5	Thin	160	64	1.0	180	250

Scaling up the memory capacity is more than just buying more memory chips. The system must be designed to allow for this expanded memory capacity. Real systems always have an upper limit on the maximum memory capacity. For instance, the IBM SP2 can accommodate at most 2-GB memory per node, and the Cray T3D only 64 MB. These capacities are mainly limited by cost and technology tradeoffs.



Software Scalability The software of a scalable computer system can be improved in many ways, as suggested below:

- A newer version of the operating system, which has more functionalities such as multithreading, supports for more user processes and larger address space, more efficient kernel functions, etc.
- A better compiler with more efficient optimizations
- More efficient mathematical and engineering libraries
- More efficient and easy-to-use applications software
- More user-friendly programming environment

1.2.2 Application Scalability

To fully exploit the power of scalable parallel computers, the application programs must also be scalable. This means that the same program should run with proportionally better performance on a scaled-up system. Two most interesting measures are scalability in *machine size* and that in *problem size*.

Scalability in Machine Size This indicates how well the performance will improve with additional processors. For instance, suppose an n -processor computer system is used as a database server. It holds the U.S. population database, is normally queried by 100 U.S. scientists, and gives a performance of 1000 *transactions-per-second* (TPS).

Now, if we double the number of processors to $2n$, how much speed improvement can we expect? If the system's speed can increase to close to 2000 TPS, we are justified in saying that the system scales well over machine size. Note that the resources increased are most frequently processors, but they could also be memory capacity and I/O capability.

Scalability in Problem Size This indicates how well the system can handle larger problems with larger data size and workload. Consider the database server example again. How well will the server behave if the server now holds the population database of China? Note that the size of the database is increased 5 times. What will happen if the number of users is increased to 200 (100 U.S. and 100 Chinese scientists in a joint study)?

There are three points worth noting when studying application scalability:

- Many practical parallel applications have built-in limitations regarding machine size and problem size. For instance, a parallel radar signal processing program may use at most 256 processors and handle at most 100 radar channels. These limitations cannot be exceeded by simply increasing machine resources. The program has to be significantly modified to handle more processors and radar channels.
- We should consider an application on a specific machine as a combination. This application/machine pair is sometimes referred to as a *system*.

- Application scalability does not depend on just machine size and problem size. It also depends on the memory capacity, I/O capability, and communication capability of the machine. All these factors jointly affect the scalability. We will treat these issues in Chapter 3.

1.2.3 Technology Scalability

Technology scalability applies to a scalable system that can adapt to changes in technology. It can be further divided into three categories: *generation scalability*, *space scalability*, and *heterogeneity scalability*.

Generation (Time) Scalability A system can be scaled up using the next-generation components, such as a faster processor, a faster memory, a newer version of operating system, a more powerful compiler, etc. Again, the computing power should increase when the system migrates to the next generation. Furthermore, the rest of the system should be usable with as few modifications as possible. It is unreasonable to require an overhaul of the entire system when the user just wants to upgrade the processor or the operating system.

The fastest evolving component of a computer system is the processor, which is expected to double its performance approximately every 18 months to 2 years. The slowest evolving is probably programming languages (Fortran 77 is still widely used), although more powerful compilers come out every few years.

A performance-conscious user may want to upgrade the processor of the computer system every 2 years and upgrade other components at much slower rates. Unfortunately, this is not what has happened with parallel computers in the past, where every generation often required new investment in system software and redevelopment of user applications.

Example 1.2 Generation scalability of IBM personal computers

The most generation-scalable computers are probably the IBM PCs. Many PC users are frequently upgrading their system components such as processor, RAM capacity, and multimedia extensions, knowing that the computing power will increase and the rest of the system can stay the same. The rest system include the monitor, the hard drive, and the printer, etc.

Most importantly, they know that the *binary code* of their existing system and application programs (DOS, Windows, database, spreadsheet, word processing software, etc.) can run faster on the upgraded system without change. This is because the PC system (from processor to motherboard, I/O cards, and software) is designed to be generation-scalable.

Space Scalability This phrase was coined by Gordon Bell to refer to the ability of a system to scale from multiple processors in a box, in a room, or in a building to multiple buildings and geographic regions. In this sense, SMPs and MPPs have limited space scalability, while the Internet has excellent space scalability.

Heterogeneity Scalability This property refers to how well a system can scale up by integrating hardware and software components supplied from different designers or vendors. This calls for using components with a standard, open architecture and interface. In the software area, this is called *portability*.

Example 1.3 Software portability in scalable parallel computers

The IBM *Parallel Operating Environment* (POE) is scalable over any size of RS6000 system. With POE, a parallel program can run without change on any network of RS6000 nodes, where each node can be a low-end PowerPC workstation or a high-end SP2 wide node.

These nodes can be connected by any popular interconnect, from the slow Ethernet to the *high-performance switch* (HPS) of SP2. The nodes can even reside at geographically distanced locations, one in the University of Southern California and the rest in Maui High-Performance Computing Center.

Another example is the *parallel virtual machine* (PVM), which is also heterogeneously scalable: It allows a parallel program to run on a network of nodes from different vendors.



1.3 Parallel Computer Models

A *parallel machine model* (also known as *programming model*, *type architecture*, *conceptual model*, or *idealized model*) is an abstract parallel computer from the programmer's viewpoint, analogous to the von Neumann model for sequential computing. According to a Purdue Report [556], such a model must characterize those capabilities of a parallel computer that are fundamental to parallel computation. All but the most specialized parallel computers can be expected to provide them.

The abstraction need not imply any structural information, such as the number of processors and interprocessor communication structure, but it should capture implicitly the relative costs of parallel computation. Such a model should be precise enough about performance without being too explicit about the implementation details. Such an abstract model provides many benefits to computer architects, software developers, programmers, and algorithm designers. Every parallel computer has a *native model* that closely reflects its own architecture.

1.3.1 Semantic Attributes

A parallel machine model can be characterized by five *semantic attributes* and several *performance attributes*. These attributes can be understood by first looking at a theoretical machine model. A *parallel random-access machine* (PRAM) of size n , as shown in Fig. 1.3, consists of n processors, all accessing a shared memory.

A parallel program on a PRAM consists of n processes, where the i -th process resides in the i -th processor and is composed of a sequence of instructions. Each processor executes one instruction at each basic time step (called a *cycle*). The instructions include data transfer, arithmetic/logic, control flow, and I/O instructions, as can be found in a typical sequential computer.

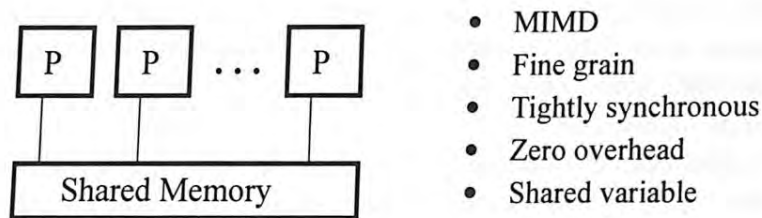


Figure 1.3 The PRAM (parallel random-access machine) model.

Homogeneity This attribute characterizes how alike the processors of a parallel computer behave when executing a parallel program. A PRAM of size 1 degenerates to the traditional *random access machine* [16], where the program executes a single instruction stream and accesses a single data stream. Based on Flynn's classification [244], such a machine is a *single-instruction* (stream) *single-data* (stream) (SISD) machine, which models conventional sequential computers.

When there is more than one processor, a PRAM will access multiple data streams and, in general, may execute multiple instruction streams. Thus a PRAM is a *multiple-instruction* (streams) *multiple-data* (streams) (MIMD) machine. However, if at each cycle all processors must execute the same instruction, there is only one instruction stream, and we have a *single-instruction* (stream) *multiple-data* (streams) (SIMD) machine.

A special case of MIMD is the *single-program multiple-data* (SPMD) computation [363], where all the processes execute the same program, parameterized by the process index. A difference between SIMD and SPMD is that in an SPMD computation, different instructions can be executed at the same cycle.

Synchrony This attribute states how tightly synchronized the processes are. A central feature of the PRAM model is that it is *synchronous*. At each cycle, all memory read operations from all the n instructions must be performed before any processor can perform a memory write or a branch.

We say that the PRAM is *synchronous at instruction level*. Such strong synchrony is absent in most real parallel computers, except SIMD machines. Rather, real MIMD parallel computers are *asynchronous*: Each process executes at its own pace, independent of the speed of other processes. If a process has to wait for other processes to enforce correct semantics, additional *synchronization operations* must be executed.

There are other synchrony schemes between these two extremes. An interesting alternative is suggested by Valiant in his *bulk synchronous parallel* (BSP) model [623]. Instead of synchronizing at every instruction, the processes synchronize with one another only at the end of every *superstep*, where a superstep can be considered as a bulk of L instructions. Within a superstep, the processors execute their instructions asynchronously. Again, within a superstep, all memory read operations from all processors must be performed before any processor can perform a memory write operation.

Another *loosely synchronous* scheme [253, 254] is very similar to the BSP model. A parallel program is divided into a sequence of *phases*. A phase could be as simple as a single assignment statement, or as complex as a sequence of arbitrary statements. Multiple processes are executed within each phase asynchronously. They synchronize at the end of each phase.

Interaction Mechanism This attribute characterizes how parallel processes may *interact* to affect the behavior of one another. In the PRAM model, processes interact through *shared variables* (or shared memory). Another important interaction mechanism is *message passing*. With this scheme, processes do not have shared variables (i.e., all variables accessible by a process are invisible to other processes).

Instead, processes interact by sending and receiving messages. An asynchronous MIMD machine interacting through shared variables is often called a *multiprocessor*. An asynchronous MIMD machine with message passing is often called a *multicomputer*. These interaction mechanisms may be combined in various ways. Multiprocessors and multicomputers will be further distinguished in Section 5.4.1 and Section 8.1.1.

Address Space The *address space* of a process is the set of memory locations accessible by the process. In some machines (e.g., the PRAM model), all memory locations reside in a *single address space*, from the programmer's viewpoint. Not all parallel computers have a single address space. For instance, in a multicomputer, each processor has its own separate address space. Such machines are said to have *multiple address spaces*. It is for this reason that processors of a multicomputer communicate not through shared variables, but by passing messages.

One important feature of the theoretical PRAM model is that all processes have equal access time to all memory locations. We say that such a machine has *uniform memory access* (UMA). On the other hand, if it takes different amounts of time for processes to access different word locations, we say the machine has *nonuniform memory access* (NUMA). These memory models will be studied in Chapters 5 and 8 in details.

For instance, in a *distributed shared memory* (DSM) multiprocessor, the programmer sees a logically shared memory, which is physically distributed to all processors, called *local memories*. One processor's local memory may be another's *remote memory*. Some computers also have a hierarchical memory organization. Besides local memories, the processors have access to a global memory. The time to access a local word is usually less than that of accessing a remote or global word, sometimes by 1 or 2 orders of magnitude. Thus for NUMA machines, it is important to exploit both *program-* and *data locality*.

Memory Model This attribute specifies how the machine model handles shared-memory access conflicts. Generally, a number of *consistency rules* are used to resolve such conflicts. For the PRAM model, one can use the *exclusive read exclusive write* (EREW) rule, by which a memory cell can be read or written by at most one processor per cycle.

The *concurrent read exclusive write* (CREW) rule says that at each cycle, a memory cell can be read by multiple processors, but can be written by at most one processor. With the *concurrent read concurrent write* (CRCW) rule, at each cycle, multiple processors can both read from and write to the same memory location. In case of conflicts, concurrent writes are resolved by some prespecified access policy.

A shared memory in a real multiprocessor is a complex system that incorporates many performance-enhancing techniques. As a consequence, the order of memory accesses in a parallel program may not directly match what is supported by a multiprocessor. The behavior of a shared memory is characterized by various memory *consistency rules*, commonly known as the *memory consistency model*, to be treated in Chapter 5.

Definition 1.2 An *atomic operation* is one that is

- (1) **Indivisible:** Once it starts, it cannot be interrupted in the middle, meaning other processes cannot see an intermediate state.
- (2) **Finite:** Once it starts, it will finish in a finite amount of time.

Definition 1.3 A stronger definition of an atomic operation requires it to satisfy the following four properties. Such an atomic operation is known as a *transaction*.

- **Atomicity:** A transaction satisfies Definition 1.2. Putting it in another way, either all or none of the suboperations of the transaction are performed.
- **Consistency:** A transaction always transfers a program from one consistent state to another. Exactly what is consistent depends on program semantics. For instance, a write-a-word operation should not write just half of a word.
- **Isolation:** The effect (result) of a transaction is not revealed to other transactions until it is *committed*.
- **Durability:** Once committed, a transaction's effect persists when system fails.

1.3.2 Performance Attributes

The semantic attributes are important in that a programmer must understand them in order to write correct parallel programs. These attributes are relatively long-lasting, usually staying the same from one generation of a parallel computer to the next. There are other attributes of a parallel computer, which a programmer should know in order to develop *efficient* parallel programs. These *performance attributes* are highly platform-dependent and are usually improved with each new generation.

Basic performance attributes are summarized in Table 1.4. Machine size n is the number of processors in a parallel computer. Workload W is the number of computation operations in a program. P_{peak} is the peak speed of a processor.

Table 1.4 Performance Attributes of Parallel Systems

Terminology	Notation	Unit
Machine size	n	Dimensionless
Clock rate	f	MHz
Workload	W	Mflop
Sequential execution time	T_1	s
Parallel execution time	T_n	s
Speed	$P_n = W / T_n$	Mflop/s
Speedup	$S_n = T_1 / T_n$	Dimensionless
Efficiency	$E_n = S_n / n$	Dimensionless
Utilization	$U_n = P_n / (nP_{peak})$	Dimensionless
Startup time	t_0	μ s
Asymptotic bandwidth	r_∞	MB/s

Definition 1.4 There are three types of operations in a parallel program: *Computation* operations include arithmetic/logic, data transfer, and control flow operations that can be found in a traditional sequential program; *parallelism* operations are needed to manage processes, such as creation and termination, context switching, and grouping; *interaction* operations are needed to communicate and to synchronize processes. ■

The parallelism and interaction operations can be either *explicit* or *implicit*. An explicit operation is one that appears in the parallel program text. For instance, a process can be created by explicitly calling the UNIX function *fork()*. An implicit operation does not appear in the program, but is done silently by the system.

For example, when the time slice of a process runs out, the operating system may perform a *process switch* to put the current process into a waiting queue and schedule another process to run. Such a process switch is not explicitly specified in the parallel program. Both explicit and implicit operations take time to execute.

Definition 1.5 The parallelism and the interaction operations are the sources of *overhead*, in that they need extra time to carry out in addition to the time needed to execute pure computational workload. More specifically, these overheads can be divided into the following four types:

- *Parallelism overhead* caused by process management
- *Communication overhead* caused by processors exchanging information
- *Synchronization overhead* in executing synchronization operations
- *Load imbalance overhead* incurred, when some processors are idle while the others are busy

■

If there were no overheads, the speedup of using n processors would always be n . These overheads will be treated in detail in Chapter 3. For now, we briefly define two overhead metrics (first introduced by Hockney [313]) for one node to communicate a message to another node. Such a communication is called *point-to-point*. There is another type, called *collective communication*, where more than one messages are communicated among multiple nodes simultaneously.

The *start-up time*, denoted by t_0 , is the time (in μ s) to communicate a 0-Byte or a short (e.g., one-word) message. The start-up time is also known as the *communication latency*, or simply *latency*. The *asymptotic bandwidth*, denoted by r_∞ , is the rate (in MB/s) for communicating a long message. These terms will be used in estimating the total communication time in point-to-point and collective message passing operations.

1.3.3 Abstract Machine Models

An abstract machine model is mainly used in the design and analysis of parallel algorithms without worrying about the details of physical machines. We show three abstract models below. Each has many variations under different assumptions.

The PRAM Model We have already described the semantic attributes of the PRAM model in Section 1.3.1. Its performance attributes are summarized below:

- The machine size n can be arbitrarily large.
- The basic time step is called a *cycle*.
- Within a cycle, each processor executes exactly one instruction. The instruction could be a *null* instruction (i.e., do nothing), in which case we say the processor is idle at that cycle.
- All processors implicitly synchronize at each cycle, and the synchronization overhead is assumed to be zero. Communication is done through reading and writing of shared variables. The communication overhead is ignored. The parallelism overhead is also ignored. Thus the only overhead accounted for in a PRAM program is the load imbalance overhead.
- An instruction can be any *random-access machine* instruction [16]. For instance, an instruction could perform the following three operations in one cycle: (1) fetch one or two words from the memory as operands, (2) perform an arithmetic/logic operation, and (3) store the result back in memory.

Example 1.4 parallel execution on the PRAM machine

To compute the inner product s of two N -dimensional vectors A and B on an n -processor EREW PRAM computer, one can assign each processor to perform $2N/n$ additions and multiplications to generate a local result in $2N/n$ cycles, and then one can add all the n local sums to a final sum s by a treelike reduction method in $\log n$ cycles. The total execution time is $2N/n + \log n$ cycles. Compared with the sequential algorithm which takes $2N$ cycles, the parallel PRAM algorithm has a speedup of $n / \{1 + [n / (2N)] \log n\} \rightarrow n$, when $N \gg n$.

■

Note that the time complexity of the sequential algorithm is a function of just one parameter: the problem size N . The parallel PRAM algorithm introduces another parameter: the machine size n . The time complexity of most PRAM algorithms is expressed as a function of N and n . The PRAM model does account for the load imbalance overhead. For instance, in the last step of the parallel summation algorithm above, only one processor is busy computing the final sum, and all other processors are idle.

Due to its simplicity and clean semantics, the PRAM model has been favored by many computer scientists. The majority of theoretical parallel algorithms are specified with the PRAM model or a variant thereof. This model is also widely used in analyzing the complexity of parallel algorithms. The main shortcoming of the PRAM model lies in its unrealistic assumptions of zero communication overhead and instruction-level synchrony.

Another problem with the PRAM model is that the time complexity of a PRAM algorithm is often expressed in the big- O notation, which is often misleading, because the

machine size n is usually small in existing parallel computers. One should not rely the big- O complexity in estimating real machine performance at all.

Example 1.5 Computational Complexity in PRAM steps

Suppose three PRAM algorithms A , B , and C have time complexities of $7n$, $(n \log n)/4$, and $n \log \log n$, respectively, when executing on an n -processor PRAM computer. According to the big- O notation, algorithm A is the fastest ($O(n)$), followed by C ($O(n \log \log n)$), with B being the slowest ($O(n \log n)$).

In reality, on machines with no more than 1024 processors, we have $\log n \leq \log 1024 = 10$ and $\log \log n \leq \log \log 1024 < 4$. Thus the fastest algorithm is really B , followed by C , while A is the slowest for systems with less than 1024 processors. ■

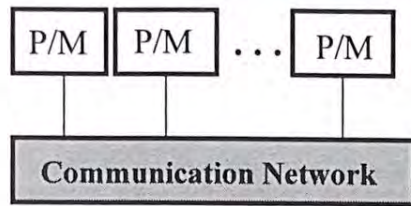
A parallel sorting algorithm based on a PRAM model can sort $O(n)$ items in $O(\log n)$ time [18]. However, sorting on real parallel computers does not use such an “optimal” algorithm, because the big- O notation hides a big constant factor. Furthermore, even when this $O(\log n)$ algorithm is improved, it may still not be suitable for practical use, since it ignored all the overheads.

Despite the inaccuracy, the PRAM model abstracts away many details, and the resulting simplicity makes it an excellent model for high-level parallel algorithm development. Many parallel algorithms, developed with the use of the PRAM model, are practically good algorithms. Sometimes an unrealistic PRAM algorithm can be improved to generate a practical parallel algorithm.

With unrealistic assumptions, the PRAM model has not been used as a machine model for *real-life* parallel computers. Other abstract models were proposed to amend this inaccuracy. They all introduce additional parameters to capture the overheads and are more complex than the PRAM model. A common feature of all these abstract models is that they are independent of the communication network topology.

The BSP Model The *bulk synchronous parallel* (BSP) model was proposed by Leslie Valiant of Harvard University to overcome the shortcomings of the PRAM model, while keeping its simplicity. A *BSP computer* consists of a set of n processor/memory pairs (*nodes*) that are interconnected by a communication network as shown in Fig. 1.4.

A BSP program has n processes, each residing on a node. The basic time unit is a *cycle* (or *time step*). The program executes as a strict sequence of *supersteps*. In each superstep, a process executes the computation operations in at most w cycles, a communication operation that takes gh cycles, and a *barrier synchronization* that takes l cycles. The barrier forces the processes to wait so that all processes have to finish the current superstep before any of them can begin the next superstep.



w : Maximum computation time within each superstep

l : barrier synchronization overhead

g : h relation coefficient

- MIMD
- Superstep:
 - Computation
 - Communication
 - Barrier
- Variable grain
- Loosely synchronous
- Nonzero overhead
- Message passing or shared variable

Figure 1.4 The bulk synchronous parallel (BSP) model.

The BSP computer is an MIMD system, because the processes can execute different instructions simultaneously. It is loosely synchronous at the superstep level, compared to the instruction-level tight synchrony in the PRAM model. Within a superstep, different processes execute asynchronously at their own pace. The BSP model does not require any specific interaction mechanism: It could be either shared variable or message passing. There is a single address space, and a processor can access not only its local memory but also any remote memory in another node.

Within a superstep, each computation operation uses only data in its local memory. These data are put into the local memory either at the program start-up time or by the communication operations of previous supersteps. Therefore, the computation operations of a process are independent of other processes. A communication is always realized in a point-to-point manner. Thus it is not allowed for multiple processes to read or write the same memory location in the same cycle.

Because of the barrier synchronization, all memory and communication operations in a superstep must completely finish before any operation of the next superstep begins. All of these imply that a BSP computer has a sequential consistency memory model.

In a real parallel computer, there are many component communication operations demanding different communication patterns among them. For simplicity, the BSP model abstracts the communication operations in a BSP superstep by the h relation concept. This greatly simplifies performance prediction and algorithm complexity analysis.

Definition 1.6 An h relation is an abstraction of any communication operation, where each node sends at most h words to various nodes and each node receives at most h words. On a BSP computer, the time to realize any h relation is no more than gh cycles, where g is a constant decided by the machine platform.

■

The BSP model is more realistic than the PRAM model because it accounts for all overheads except the *parallelism overhead* for process management. The execution time of a superstep is determined as follows:

- To account for load imbalance, the computation time w is the maximum number of cycles spent on computation operations by any processor.
- The synchronization overhead is l , which has a lower bound of the communication network latency (i.e., the time for a word to propagate through the physical network) and is always greater than zero.
- The communication overhead is gh cycles, where g is the proportional coefficient for realizing an h relation. The value of g is platform-dependent, but independent of the communication pattern. In other words, gh is the time to execute the most time-consuming h relation. The parameter g has a small value on a computer with more efficient communication support.
- The time for a superstep is estimated by the sum $w + gh + l$.

The BSP model does not disallows the overlapping of the computation, the communication, and the synchronization operations within a superstep. If all three types of operations are fully overlapped, the time for a superstep becomes $\max(w, gh, l)$. However, we will use only the more conservative $w + gh + l$.

Example 1.6 Parallel execution using the BSP machine model

Consider the inner-product problem in Example 1.4. We can solve this problem on an 8-processor BSP computer in 4 supersteps:

Superstep 1

Computation: Each processor computes its local sum in $w = 2N/8$ cycles.

Communication: Processors 0, 2, 4, 6 send their local sums to processors 1, 3, 5, 7. Apply 1 relation here.

Barrier synchronization.

Superstep 2

Computation: Processors 1, 3, 5, 7 each perform one addition ($w = 1$).

Communication: Processors 1 and 5 send their intermediate results to processors 3 and 7. A 1 relation is applied here.

Barrier synchronization.

Superstep 3

Computation: Processors 3 and 7 each perform one addition. ($w = 1$).

Communication: Processor 3 sends its intermediate result to processor 7. Apply 1 relation here.

Barrier synchronization.

Superstep 4

Computation: Processor 7 performs one addition ($w = 1$) to generate the final sum.

No more communication or synchronization is needed.

The total execution time is $2N/8 + 3g + 3l + 3$ cycles. In general, the execution time is $2N/n + \log n (g + l + 1)$ cycles on an n -processor BSP. This is in contrast to the $2N/n + \log n$ time on a PRAM computer. The two extra terms, $g \log n$ and $l \log n$, correspond to *communication* and *synchronization overheads*, respectively. ■

Phase Parallel Model The authors have proposed a *phase parallel model* [657] for parallel computation, that is further refined from the above two abstract machine models. This model is similar to the BSP model with the following distinctions:

- A parallel program is executed as a sequence of phases. The next phase cannot begin until all operations in the current phase have finished. There are three types of phases as defined below:
 - (1) **Parallelism phase:** This refers to the overhead work involved in process management, such as process creation and grouping for parallel processing.
 - (2) **Computation phase:** One or more processors execute a number of local computation operations. Here local means that all data needed by a node are available in its local memory.
 - (3) **Interaction phase:** This refers to tasks needed to perform interaction operation: a communication, a synchronization, or an aggregation (e.g., *reduction* and *scan*) operation.
- For a given a computation phase, denote the workload (e.g., in Mflop) by w and t_f is the average time (in μs) to execute a computation operation. Different computation phases may execute different workloads at different speeds, corresponding to different w and t_f values. This is in contrast to the previous two models, where every computation operation is assumed to take a uniform *cycle*.
- Different interaction operations may take different times. However, there is a general form for the time to execute an interaction operation:

$$T_{interact}(m, n) = t_0(n) + \frac{m}{r_\infty(n)} = t_0(n) + m \cdot t_c(n) \quad (1.1)$$

Here m is called the *message length* in bytes, the *start-up time* $t_0(n)$ and the *asymptotic bandwidth* $r_\infty(n)$ are functions of machine size n . The parameter

$t_c(n) = 1/r_\infty(n)$ is called the *per-byte messaging time*. We explain below how to sequence parallelism, computation, and interaction phases. Analogous to the BSP model, we call such a sequence a *superstep*. We focus on the case when the interaction is a communication with a message length of m B. Note that some, but not all, phases may be missing in a superstep. For example, a superstep may contain just a computation phase, while another superstep may contain just an interaction phase. ■

Example 1.7 Parallel execution using the phase parallel model

The inner-product problem in Example 1.4 is solved by the following three-phase parallel algorithm:

Parallelism phase:	parfor ($i=0; i<n; i++$)
Computation phase:	$\{ \quad \quad \quad \sum_{j=i}^{(i+1)n} A[j] \cdot B[j]$
Interaction phase:	$s = \text{sum_reduction}(\text{LocalSum}[i]);$
	$\}$

Figure 1.5 Parallel inner-product computation using the phase parallel model

The first phase creates n processes. In the second phase, process i computes $\text{LocalSum}[i]$ for all $0 \leq i < n$. In the third phase, all processes participate in a reduction operation to generate the final result s .

In the computation phase, each process computes a workload of $2N/n$ flop at a speed of $1/t_f$, where t_f is the average time to execute a flop by a processor, and is equal to the reciprocal of the single-processor speed. That is, $t_f = 1/P$.

Assume that the parallelism overhead is $t_p(n)$ and the interaction overhead for the reduction operation is $t_0(n)$ (the reduction operation has a constant message length $m = 4$ B). Then the total execution time is $2Nt_f/n + t_p(n) + t_0(n)$. ■

Suppose the total computational workload performed by the n processors in a superstep is $W = nw$, which can be divided into n independent computations. The execution times of the n independent computations are assumed to be random variables, with a mean value of wt_f and a standard deviation of σt_f . Therefore, the parameters w and σ represent the mean and the standard deviation, respectively, of the per-processor

workload. Intuitively, w indicates *granularity* and σ indicates *load imbalance*, respectively. A small σ indicates a nearly balanced parallel computation phase. When $\sigma = 0$, the total workload W is assumed evenly divided among the processors.

Since the n processors perform independent computations, the execution time of the computation phase, denoted by T_{comp} , should be the maximum of the n individual execution times of the processors. Kruskal and Weiss [379] have shown that this maximum value can be approximated by

$$T_{\text{comp}} = (w + \sigma\sqrt{2\log n}) t_f \quad (1.2)$$

Definition 1.7 Assume the message length is $m = \alpha w B$ in the interaction phase of a superstep. The parameter α , called the *communication-to-computation ratio* (CCR) of the superstep, measures how many bytes need to be communicated per flop computation. ■

From Eq.(1.1), the interaction overhead is expressed by

$$T_{\text{interact}} = t_0(n) + m/r_\infty(n) = t_0(n) + \alpha \cdot w \cdot t_c(n) \quad (1.3)$$

The total execution time of the superstep on n processors is expressed by:

$$\begin{aligned} T_n &= T_{\text{comp}} + T_{\text{interact}} + T_{\text{par}} \\ &= (w + \sigma\sqrt{2\log n}) t_f + t_0(n) + \alpha \cdot w \cdot t_c(n) + t_p(n) \end{aligned} \quad (1.4)$$

Improved from the PRAM and the BSP models, the phase parallel model is closer to covering real machine/ program behavior. All types of overheads are accounted for, as shown in Eq.(1.4): the load imbalance overhead (the σ term), the interaction overhead (the t_0 and the t_c terms), and the parallelism overhead (the t_p term). Different expressions are used for different interactions.

These expressions can be obtained by measuring the interaction operations on real machines. Once obtained, they can be used many times by different applications. This point will be treated again in Chapter 3. The parameters used in the phase parallel model are summarized in Table 1.5.

Table 1.5 Parameters of the Phase Parallel Model

Parameter	Unit	Meaning	Depending on
t_0	μs	Start-up time	System
t_c	$\mu s/B$	Time to communicate a byte	System
t_f	μs	Time to execute a flop	System and application
t_p	μs	Parallelism overhead	System
w	Mflop	Mean workload	Application
σ	Mflop	Standard deviation of workload	Application
α	B/flop	Communication-to-computation ratio	Application

1.3.4 Physical Machine Models

Large-scale computer systems are generally classified into six practical machine models: the *single-instruction multiple-data* (SIMD) machines, the *parallel vector processor* (PVP), the *symmetric multiprocessor* (SMP), the *massively parallel processor* (MPP), the *cluster of workstations* (COW), and the *distributed shared memory* (DSM) multiprocessors. SIMD computers are used mostly for special-purpose applications. The remaining models are all MIMD machines, as summarized in Table 1.6.

Table 1.6 Semantic Attributes of Parallel Machine Models

Attributes	PRAM	PVP/SMP	DSM	MPP/COW
Homogeneity	MIMD	MIMD	MIMD	MIMD
Synchrony	Instruction-level synchronous	Asynchronous or loosely synchronous	Asynchronous or loosely synchronous	Asynchronous or loosely synchronous
Interaction mechanism	Shared variable	Shared variable	Shared variable	Message passing
Address space	Single	Single	Single	Multiple
Access cost	UMA	UMA	NUMA	NORMA
Memory model	EREW, CREW or CRCW	Sequential consistency	Weak ordering is widely used	N/A
Example machines	Theoretical model	IBM R50, Cray T-90	Stanford DASH, SGI Origin 2000	Cray T3E, Berkeley NOW

Five MIMD parallel machine models are shown in Fig. 1.6. Most modern parallel computers are built with commercially available, off-the-shelf commodity hardware and software components. The only exception is the PVP machines, where many building blocks are custom-made.

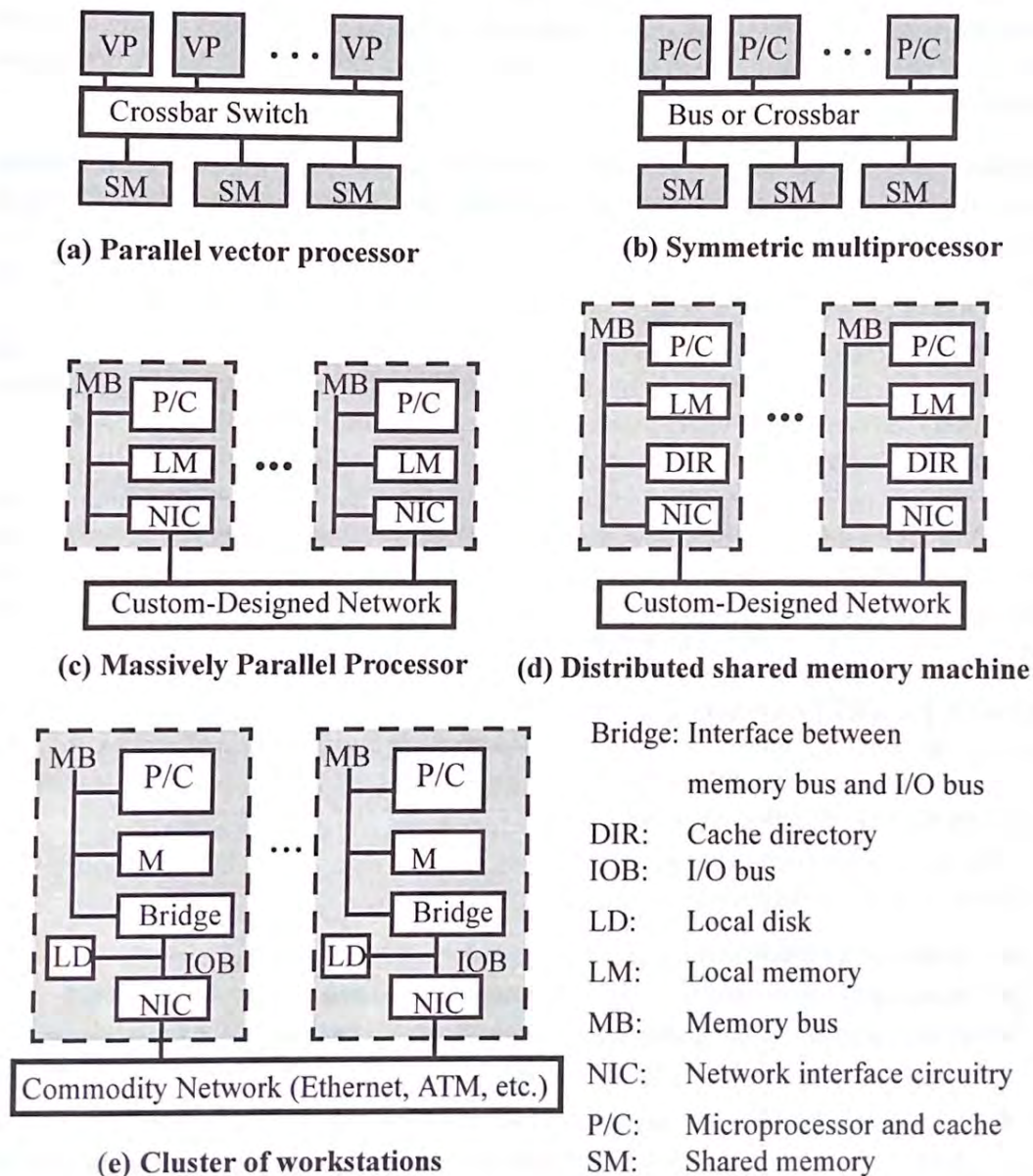


Figure 1.6 Five physical parallel computer models: the PVP, SMP, MPP, DSM, and COW.

Parallel Vector Processors The structure of a typical PVP is shown in Fig. 1.6(a). Examples of PVPs include the Cray C-90, Cray T-90, and NEC SX-4. Such systems contain a small number of powerful custom-designed *vector processors* (VPs), each capable of at least 1 Gflop/s performance.

A custom-designed, high-bandwidth crossbar switch network connects these vector processors to a number of *shared-memory* (SM) modules, which provide high-speed data access. For instance, in the T-90, the shared memory can pump data to a processor at 14 GB/s. Such machines normally do not use caches, rather they use a large number of vector registers and an instruction buffer.

Symmetric Multiprocessors The SMP architecture is shown in Fig. 1.6(b). Examples include the IBM R50, the SGI Power Challenge, and the DEC Alpha server 8400. Unlike a PVP, an SMP system uses commodity microprocessors with on-chip and off-chip caches. These processors are connected to a shared memory through a high-speed snoop bus. On some SMPs, a crossbar switch is used in addition to the bus.

SMP systems are heavily used in commercial applications, such as databases, on-line transaction systems, and data warehouses. It is important for the system to be *symmetric*, in that every processor has equal access to the shared memory, the I/O devices, and the operating system services.

Being symmetric, a higher degree of parallelism can be released, which is not possible in an *asymmetric* (or *master-slave*) multiprocessor system. In 1997, most PVPs and SMPs have at most 64 processors, such as the Sun Ultra Enterprise 10000. The limitation is mainly caused by using a centralized shared memory and a bus or crossbar system interconnect, which are both difficult to scale once built.

Massively Parallel Processors To take advantage of higher parallelism available in applications such as scientific computing, engineering simulation, signal processing, and data warehousing, we need to use even higher scalability computer platforms by exploiting the distributed memory architectures, such as MPPs, DSMs, and COWs.

The term MPP (*massively parallel processor*) generally refers to a very large-scale computer system having the following features:

- It uses commodity microprocessors in processing nodes.
- It uses physically distributed memory over processing nodes.
- It uses an interconnect with high communication bandwidth and low latency.
- It can be scaled up to hundreds or even thousands of processors.
- Like the multiprocessor model, it is an asynchronous MIMD machine. However, processes are synchronized through blocking message-passing operations, not shared-variable synchronization operations.
- The program consists of multiple processes, each having its private address space. Processes interact by passing messages.

The MPP modeled in Fig. 1.6(c) is more restricted, representing machines such as the Intel Paragon and TFLOP. Such a machine consists a number of *processing nodes*, each containing one or more microprocessors interconnected by a high-speed memory bus to a local memory and a *network interface circuitry* (NIC). The nodes are interconnected by a high-speed, proprietary communication network. The nodes are said to be *tightly coupled*.

Distributed Shared-Memory (DSM) Machines The DSM machines are modeled in Fig. 1.6(d), based on the Stanford DASH architecture [405]. *Cache directory* (DIR) is used to support distributed coherent caches. The Cray T3D is also a DSM machine [3]. But it does not use the DIR to implement coherent caches. Instead, the T3D relies on special hardware and software extensions to achieve the DSM at arbitrary block-size level, ranging from words to large pages of shared data.

The main difference between DSM machines and SMP is that the memory is physically distributed among different nodes. However, the system hardware and software create an illusion of a single address space to application users. A DSM machine can be also implemented with software extensions on a network of workstations such as the TreadMarks, to be studied in Chapter 10.

Clusters of Computers The cluster concept is shown in Fig. 1.6(e). Examples include Digital's TruCluster [197], IBM SP2 [14], and the Berkeley NOW [178]. Clusters are the low-cost variation of MPPs in some cases. Important cluster distinctions are listed below:

- Each node of a COW is a complete workstation, minus some peripherals (e.g., monitor, keyboard, mouse, etc.). Such a node is sometimes called a "headless workstation". A node may also be an SMP or a PC.
- The nodes are connected through a low-cost commodity network, such as Ethernet, FDDI, Fiber-Channel, and ATM switch, although proprietary networks are also used in some commercial clusters.
- The network interface is *loosely coupled* to the I/O bus in a node. This is in contrast to the *tightly coupled* network interface of an MPP [Fig. 1.6(c)], which is connected to the memory bus of a processing node.
- There is always a local disk, which may be absent in an MPP node.
- A complete operating system resides on each node, whereas in some MPPs only a microkernel exists. The operating system of a COW is the same workstation UNIX, plus an add-on software layer to support single system image, availability, parallelism, communication, and load balancing.

The boundary between MPPs and COWs is becoming fuzzy these days. The IBM SP2 is considered an MPP. But it has a cluster architecture, except that a proprietary *high-performance switch* is used as the communication network. Clusters have many cost/performance advantages over the MPPs. Clustering is becoming a trend in developing scalable parallel computers as highlighted in the next section.

1.4 Basic Concepts of Clustering

In this section, we elaborate further on basic concepts concerning clusters of computers. Multicomputer clusters are distinguished below from SMPs, MPPs, and distributed systems. Benefits and challenges are identified in building clusters. Detailed network technologies will be treated in Chapter 6. Principles of clustering are treated in Chapter 9, and example cluster systems are studied in Chapter 10.

1.4.1 Cluster Characteristics

A *cluster* is a collection of complete computers (nodes), that are physically interconnected by a high-performance network or a *local-area network* (LAN). Typically, each computer node is an SMP server, or a workstation, or a personal computer. More importantly, all cluster nodes must be able to work together collectively as a single, integrated computing resource; in addition to filling the conventional role of using each node by interactive users individually.

Five architectural concepts (indicated in *italic*) are merged into a cluster as an *inter-connected* set of whole computers (*nodes*) that work collectively as a *single system* to provide uninterrupted (*availability*) and efficient (*performance*) services. A conceptual architecture of a cluster is shown in Fig. 1.7.

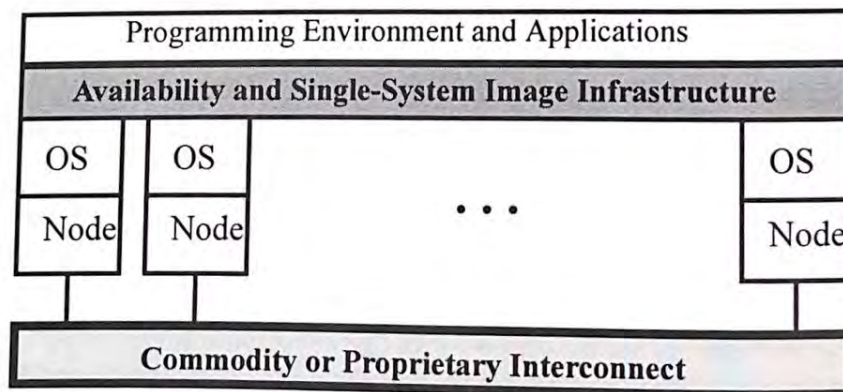


Figure 1.7 Typical architecture of a cluster of multiple computers.

The five architectural concepts of clusters are characterized below. More can be found in Chapters 9 and 10.

- **Cluster Nodes:** Each node is a complete computer. This implies that each node has its processor(s), cache, memory, disk, and some I/O adapters. Furthermore, a complete, standard operating system resides on each node. A node can have more

than one processor, but only one copy of the OS image. Chapter 10 will describe the details of some cluster nodes.

- **Single-System Image:** A cluster is a single computing resource. This is in contrast to a *distributed system* (e.g., a local computer network), where the nodes are used only as individual resources. A cluster realizes this single-resource concept by a number of *single-system image* (SSI) techniques. SSI makes a cluster easier to use and manage. Most commercially available clusters have not achieved the status of full SSI operations.
- **Internode Connection:** The nodes of a cluster are usually connected through a commodity network, such as Ethernet, FDDI, Fiber-Channel, and ATM switch. However, standard protocols are used to smooth internode communication. Different networks for cluster construction are treated in Chapter 6.
- **Enhanced Availability:** Clustering offers a cost-effective way to enhance the availability of a system, which is the percentage of time a system remains available to the user. The availability techniques will be discussed in Section 9.2.
- **Better Performance:** A cluster should offer higher performance in a number of service areas. One area is to treat a cluster as a *superserver*. If each node of an n -node cluster can serve m clients, the cluster as a whole can serve mn clients simultaneously. Another area is to use a cluster to minimize the time to execute a single large job by distributed parallel processing.

1.4.2 Architectural Comparisons

Clusters, SMPs, MPPs, and distributed systems are four overlapped architectural concepts, as shown in Fig. 1.8. An good example of a distributed system is a LAN. A node in a LAN can be a PC, a workstation, or an SMP server. The *node complexity* refers to the hardware and software capability. A cluster node is more complex than an MPP node because the former has a disk and a full-fledged operating system, while the latter may not have a disk and may use only a microkernel.

An SMP server is more complex than a cluster node because it has more peripherals, such as terminals, printers, external *redundant array of inexpensive disks* (RAID), and tape units. Some of these peripherals may be absent in cluster nodes. We project the cluster boundary will be continuously expanded to overlap even more with the MPPs and SMPs in the future.

The horizontal axis shows the degree of SSI at various abstraction levels: ranging from a single application, to subsystem, runtime system, OS kernel, and hardware (memory and/or I/O) levels. In other words, the SSI is a relative concept, depending on the boundary of the SSI from a user's perspective. The SMP always offers SSI at all levels. The MPP supports SSI only in some application and system levels. The clusters offer SSI at a reduced level from SMPs to a comparable level of MPPs.

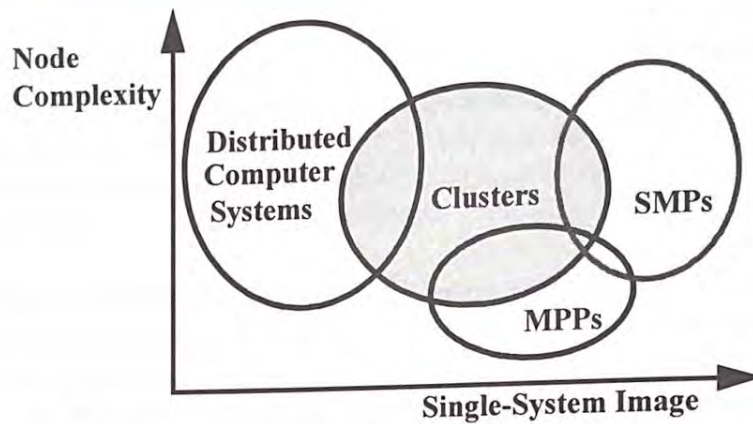


Figure 1.8 Overlapped design space of clusters, MPPs, SMPs, and distributed computer systems.

A cluster or an MPP can be used as a single resource (e.g., one giant workstation), while a distributed system always produces multiple system images due to the autonomy of individual computers in the network domain. We compare other characteristics of all four system types in Table 1.7. Current MPPs and distributed systems have a maximum size of thousands of nodes. Most clusters have tens of nodes, only as a few clusters have exceeded 100 nodes.

MPP nodes often run a microkernel, while nodes in other architectures use a full-size OS. The operating systems on a distributed system are usually heterogenous, but homogenous OS are preferred for the others. Internode communication is often done by message passing in an MPP or cluster; whereas shared files in the server are often used in a distributed system. The processors in an SMP communicate through the shared memory. Only SMPs maintain a single address space. The MPP has a single address space only if DSM is supported by hardware.

SMP has a single run queue for all processors. Multiple run queues are used in cluster and MPP nodes, but they are coordinated to balance the workload. Multiple queues in a distributed system are mostly independent. Distributed systems must use standard communication networks and protocols to accommodate heterogeneous node platforms. A distributed system often spans multiple organizations, thus requires internode security measures. These are not required for MPPs and clusters.

1.4.3 Benefits and Difficulties of Clusters

The cluster concept brings many benefits as well as challenges. Among them the most important ones are *usability*, *availability*, *scalability*, *available utilization*, and

Table 1.7 Comparison of Cluster, MPP, SMP, and Distributed Systems

System Characteristic	MPP	SMP	Cluster	Distributed System
Number of nodes (N)	$O(100)$ - $O(1000)$	$O(10)$ or less	$O(100)$ or less	$O(10)$ - $O(1000)$
Node complexity	Fine or medium grain	Medium or coarse grain	Medium grain	Wide range
Internode communication	Message passing or shared variables for DSM	Shared memory	Message passing	Shared files, RPC, message passing
Job scheduling	Single run queue at host	Single run queue	Multiple queues but coordinated	Independent multiple queues
SSI support	Partially	Always	Desired	No
Node OS copies and Type	N (microkernel) and 1 host OS (monolithic)	One (monolithic)	N (homogeneous desired)	N (heterogenous)
Address space	Multiple (single if DSM)	Single	Multiple	Multiple
Internode security	Unnecessary	Unnecessary	Required if exposed	Required
Ownership	One organization	One organization	One or more organizations	Many organizations
Network protocol	Nonstandard	Nonstandard	Standard or nonstandard	Standard
System availability	Low to medium	Often low	Highly available or fault-tolerant	Medium
Performance metric	Throughput and turnaround time	Turnaround time	Throughput and turnaround time	Response time
SSI: Single-system image, RPC: Remote procedure call, DSM: Distributed shared memory				

performance/cost ratio. Simply connecting a bunch of workstations through an Ethernet and dubbing them a cluster will hardly lead to a high-availability and high-performance system. A number of techniques (mostly software) must be in place to realize these potentials. In Chapter 5, 9, and 10, we will discuss these technological challenges in detail. Only basic concepts are given below for the time being.

Usability Since individual nodes of a cluster are a traditional platform, users can develop and run their applications in a familiar, mature environment. The platform provides all the

powerful workstation programming environment tools and allows the thousands of existing (sequential) applications to run without change. Thus a cluster can be viewed as a huge workstation, providing much increased throughput and reduced response time for multiple sequential user jobs.

For parallel applications, a cluster is no more difficult to program than a message-passing MPP. In addition, many database vendors have modified their products to run on clusters, such as the IBM DB2 Parallel Edition, Oracle, and Sybase.

Availability The term *availability* refers to the percentage of time a system is available for productive use. Traditional monolithic systems, such as mainframes and fault-tolerant systems, achieve high availability through expensive, customized design. Instead of using custom components, clusters utilize inexpensive commodity components to provide higher availability, with multitudes of redundancy:

- **Processors and Memories:** A cluster has multiple memory and processor components. When one fails, the others can still be used to keep the cluster going. In contrast, when the shared memory of an SMP machine fails, the entire system will be brought down.
- **Disk Arrays:** A cluster has multiple local disks. Thus, the failure of one does not crash the entire system. In fact, a node with a faulty disk can still function, by using remote disks.
- **Operating System:** A cluster has multiple OS images, each residing in a separate node. When one system image crashes, other nodes can still work. This is in contrast to SMP, which has a single OS image residing in the shared memory. The failure of this image crashes the entire system.

The key to realizing the availability potential of clusters is a number of software techniques. Techniques are also needed to make shared components (e.g., the interconnect) highly available; otherwise they may become a *single point of failure*. These techniques will be discussed in Chapter 9.

Scalable Performance A cluster's computing power can increase with added nodes. Again, clusters' scalability is a multitude scalability. This can be best seen by comparing clusters to SMPs. SMPs are processor-scalable systems, while clusters scale in many components, including processor, memory, disks, and even I/O devices. Being loosely coupled, clusters can scale to hundreds of nodes, while it is extremely difficult to build an SMP of more than tens of processors.

In an SMP, the shared memory (and the memory bus) is a bottleneck. When several sequential programs are running on an SMP, they have to compete for the memory, even though these programs are totally independent. When the same set of programs is executed in a cluster, there is no memory bottleneck. Each program can be executed in one node, utilizing the local memory.

For such applications, clusters can provide much higher aggregate memory bandwidth and reduced memory latency. The local disks of a cluster also aggregate to large disk space, which can easily surpass that of a centralized RAID disk. The enhanced processing, storage, and I/O capability enables a cluster to solve large-scale problems by using some of the well-developed parallel software packages such as PVM or MPI, to be discussed in Chapter 13.

Clusters are compared with UP (uniprocessor), SMP, MPP, and fault-tolerant systems in Fig. 1.9, with respect to scalable performance and availability. SMPs are not highly scalable, due to the use of contention bus and centralized shared memory. The single operating system image and shared memory are potentially two single points of failure, which reduces the availability of an SMP.

Fault-tolerant systems have extremely high availability, but it is expensive to scale them up. MPPs are just the opposite. They can be easily scaled up and yet preserve a single-system image. Clusters are currently in the middle ground, moving toward higher performance and higher availability in future upgrades.

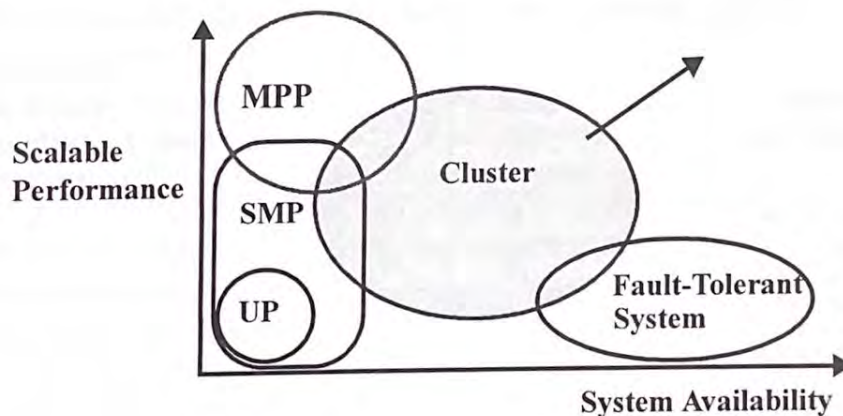


Figure 1.9 Comparison of uniprocessor, SMP, MPP, cluster, and fault-tolerant systems in system availability.

Performance/Cost Ratio Clusters can achieve the above benefits cost-effectively. Traditional PVP supercomputers and MPPs can easily cost tens of millions of dollars. On the hand, the price of a cluster with the same peak performance is one or two orders of magnitude lower. Clusters are largely made of commodity components, and their performance and price follow Moore's law, which makes the performance/cost ratio of clusters grow faster than those of PVPs and MPPs.

Example 1.8 Performance/cost ratios of parallel computers and clusters

The Berkeley NOW group has compared the performance/cost ratios of two parallel computers and four NOW configurations, as reported in Table 1.8. The six configurations are a 16-processor Cray C90 PVP, a 256-node Intel Paragon MPP, and four versions of a network of 256 RS6000 workstations, when running an atmospheric chemistry application called Gator.

The program has three main steps. The computation step needs to perform 36 Gflop to solve *ordinary differential equations* (ODEs) in parallel. The transport step is communication-intensive. The I/O step is dominated by inputting 3.9 GB of data.

The execution times for C90 and Paragon are measured data, while those for the four versions of NOW are predicted by the Berkeley group. Just connecting 256 workstations using Ethernet and using PVM over TCP/IP for communication lead to a dreadful system, which is 1000 times slower than C90 and whose performance/cost ratio is 138 times lower. The main degrading factor is the communication overhead, which is 23,340 s versus 4 s on the C90.

Table 1.8 Performance/Cost Ratios of C90, Paragon, and Four NOW Systems (Courtesy: Anderson et al. *IEEE Micro*, [33], 1995)

System Configuration	ODE (s)	Transport (s)	Input/Output (s)	Total (s)	Cost (\$M)	Performance/Cost Ratio (Mflop/s per \$M)
Cray C90	7	4	16	27	30	44
Intel Paragon	12	24	10	46	10	78
NOW	4	23,340	4030	27,347	4	0.32
NOW+ATM	4	192	2015	2211	5	3.3
NOW+ATM+PIO	4	192	10	205	5	35
NOW+ATM+PIO+AM	4	8	10	21	5	342

Replacing the Ethernet with a high-bandwidth ATM switch improved the performance 10 times, adding 25% cost. Implementing parallel I/O capability improves the performance by another order. Finally, using active message to speed up communication further reduces the execution time by an order of magnitude. With all improvements in place, the cluster is faster than C90, and the performance/cost ratio becomes 7 times better than that of the C90. In Chapters 9 and 10, we will learn how to develop SSI clusters with all the above hardware and software improvements. ■

1.5 Scalable Design Principles

Designing a scalable high-performance computer is a complex engineering process. Over the past two decades, four scalable design principles have emerged, based on the experiences and lessons learned from both successful and failed system designs:

- Principle of independence
- Principle of balanced design
- Principle of design for scalability
- Principle of latency hiding

These principles come into play again and again in subsequent chapters. In this section, we explain the first three principles. Latency hiding issues will be discussed in Chapter 5.

1.5.1 Principle of Independence

This principle says that we should try to make the components of a system independent of one another. If complete independence cannot be achieved, we should make the dependence as small and clear as possible. Here the components include both hardware and software components.

One obvious benefit is the possibility of *independent scaling*: We can scale up a system along one dimension by improving one component, independent of others. The user is not required to simultaneously upgrade all components. This is also called *incremental scaling*. For instance, if the user wants to scale the size of a parallel computer system by adding nodes, the user should not be required to upgrade the operating system or the programming environment.

When the processor is upgraded to the next generation, the system can operate at higher performance without upgrading the remaining components. Another benefit is to enable heterogeneity scalability. Since a component is not tied to a particular architecture or system, it can be used in building many other systems. This can result in cost cutting. Ideally, the component becomes a commodity, with the following features:

- It has an open architecture with a standard interface to the rest of the system.
- It is an off-the-shelf product; or better yet, it is in the public domain.
- It has multiple vendors in the open market with a large volume.
- It is relatively mature, as it is used by many people for a sufficiently long time period and is reasonably debugged.
- Because of all the above, the commodity component has low cost and high availability and reliability.

We elaborate below the *independence principle* by listing some specific applications along with some concrete examples.

- The algorithm should be independent of the architecture.
- The application should be independent of the platform.
- The programming language should be independent of the machine.
- The language should be modular and have *orthogonal* features.
- The node should be independent of the network, and the network interface should be independent of the network topology.

Example 1.9 Independence spirit in developing the Internet and IBM SP2.

The success of the Internet is a perfect example of the benefits of the independence principle. The Internet is independent of the type of hosts, interconnection hardware, and application software. Different types of hosts are connected, ranging from PCs to supercomputers from different vendors. The interconnect hardware can be Ethernet, FDDI, etc. Users can use different software to browse the Web.

The IBM SP2 design incorporates the independence principle. The node architecture is designed to allow the use of different communication architectures (e.g., Ethernet or HPS). The communication protocol is independent of the communication hardware: on either an Ethernet or HPS, either the standard IP protocol or an IBM proprietary *user space* protocol can be used.

■

Example 1.10 The MPI story and the legend of hypercube computers

Message passing interface (MPI) is a good example of using a few independent (orthogonal) language features. MPI is based on four main concepts that are orthogonal to one another: *data type*, *communication operations*, *communicator*, and *virtual topology*. Any combination of the four is valid. This orthogonal independence brings a *multiplicative effect*: A few simple concepts, when combined together, can offer a lot of functionalities.

Many parallel algorithms developed for earlier hypercube computers explicitly used the hypercube topology of the interconnect, which may not perform well on a mesh-collected system. Learning from this lesson, current MPPs employ communication algorithms that are independent of the interconnect topology. A good example is the collective communication library in IBM SP system[59].

■

Two common techniques are employed to realize the independence principle: separating architecture from implementation, and using standard components.

Architectures vs. Implementations The concept of architecture should be separated from that of implementation. An *architecture* is a precise model (or specification) of the

common behavior or functionalities of a family of computer systems or system components. An *implementation* is a specific realization of the model. The architecture model is used by both users and designers. An architecture can have many different implementations, which may have different performances, but all realize the same functionalities.

The architecture concept is an old one, which was first proposed and successfully applied in the design of the IBM 360 computers. Closely related to the architecture concept is the *family* concept. All members of a family share the same architecture. But different members may use different implementations and configurations to suit the different performance/cost requirements of users.

Another related concept that has gained popularity is the *open architecture* (or *open system*) concept, which means that the owner of the architecture (usually the vendor) allows users or third parties to see the architecture so that they can make components that are compatible with the architecture, or even modify or reengineer it. The success of IBM PCs testifies that this is indeed a technically powerful and commercially viable concept.

Using Standard Components The above discussion about architecture already alludes to the importance of using standard components. There are two types of standard:

- The first is an *industry standard* (also called a *de facto* standard), which is usually pioneered by a company, then becomes widely used by the end users and accepted by the majority of the industry.
- The second type of standard is established by a national or international standards organization, such as the *International Standards Organization* (ISO), the *American National Standards Institute* (ANSI), and the IEEE Standard Committee.

Caveats • We must be aware of two caveats when applying the independence principle:

- First, parallel computers are at the cutting-edge of computing technology. In any such system, there is usually some key component/technique that is novel, thus not quite a standard yet.
- Second, we cannot build an efficient system by simply scaling up one or a few component alone.

For example, using faster processors but slow memory modules or slow communication subsystems will not work. We must design a balanced system among all subsystems, as discussed in the next subsection.

1.5.2 Principle of Balanced Design

This principle states that we should try to minimize any performance bottleneck. We should avoid an unbalanced system design, where a slow component degrades the performance of the entire system, even though the remaining components are fast.

Furthermore, we should avoid *single point of failure*; i.e., the failure of one component breaks down the entire system. There are several theoretical and experimental results that can help achieve a balanced design.

Amdahl's Law Suppose an application program is grouped into two types of computational structure: part X and part Y . The two parts take $X\%$ and $Y\%$ of the total execution time, respectively:



If the part X is improved to run n times as fast, the speedup S is defined

$$S = \frac{\text{Original time}}{\text{Improved time}} = \frac{1}{(X/n)\% + Y\%} \rightarrow \frac{1}{Y} \quad n \rightarrow \infty \quad (1.5)$$

This equation is known as *Amdahl's law*, which has the following implications:

- We should optimize the larger part X , i.e., speed up the common cases.
- The best speedup is upper bounded by the value $1/Y$.
- The slow part Y is called the *bottleneck*. We should minimize Y , i.e., make Y as small as possible.

Amdahl's Rule This is also known as *Amdahl's other law*. It states that the processing speed should be balanced against the memory capacity and the I/O speed. More specifically, a rule of thumb is that a 1 million instructions per second (MIPS) computational speed should be balanced with 1 MB of memory capacity and 1 Mb/s of I/O rate.

Amdahl's rule is a rule of thumb. As such it is not necessarily always observed as far as the *peak* speed is concerned. Current processors have a peak speed in the range of 400-1200 Mi/s, but not many machines have installed with more than 400 MB local memory per processor. Amdahl's rule is more observed in recent systems when the sustained computational speed (e.g., sustained Mflop/s) is used, instead of the peak speed.

Example 1.11 The PetaFLOPS Project

A recent prediction from the PetaFLOPS Project [587] indicates that for a wide range of scientific/engineering simulation problems, the memory requirement (in GB) and the speed demand (in Gflop/s) have the following relationship:

$$\text{Memory} \sim \text{Speed}^{3/4} \quad (1.6)$$

Thus a memory size of about 30 TB is appropriate for a Pflop/s machine. Note that 1 Pflop/s = 1,000,000 Gflop/s. ■

Example 1.12 I/O and checkpoint problems

To understand the I/O speed requirement, consider the *checkpoint* problem: The system needs to periodically dump the memory contents to the disk so that in case of a system crash, users can restart their work from the last checkpoint, instead of from the very beginning. Assume we want the dumping to be finished in 90 s. Then for a 1-MB memory, we need a disk bandwidth of $1/90$ MB/s = 0.9 Mb/s, which is close to Amdahl's rule.

For larger systems, the checkpoint time is even longer. Assume a dumping time of 900 s. Then for a machine with 1-GB memory, we need a disk bandwidth of $1000/900 = 1.1$ MB/s. For 100-GB memory, the disk I/O requirement increases to more than 100 MB/s! ■

The 50% Rule From Eq.(1.4), the performance of a parallel program can be degraded by *load imbalance*, *parallelism overhead*, *communication start-up overhead*, and *per-byte communication overhead*. Another rule of thumb is that a parallel system is balanced if each of the four overhead factor may degrade the performance by a factor of no more than 50%. Using this rule, we can estimated desired bounds on various overhead factors. This will be treated in Chapter 3.

For instance, Table 1.9 lists the desired values of the *communication start-up overhead* t_0 under different granularity and speed conditions. Current message-passing computers have too large a start-up overhead to support fine-grained or even medium-grained parallel applications. The exception is some Cray vector supercomputers (PVPs), which have a small t_0 of only 2 μ s for point-to-point communication.

When the message size $\alpha \cdot w$ or the granularity w is large, the bandwidth $r_\infty = 1/t_c$ becomes a more important factor. Table 1.10 lists the desired values of r_∞ under different assumptions of speed P_1 and *communication-to-computation* ratio α .

Example 1.13 Grid lattice in PDE solvers

Many numerical parallel *partial differential equation* (PDE) solvers for two-dimensional (2D) problems use the following scheme: The data domain is a 2D grid with $N \times N$ data points. Each data point needs X B of memory, and the total memory

Table 1.9 Desired Communication Startup Overhead Values

Grain Size w	Single-Node Speed P_1	t_0 should be less than
100 flop (Fine Grain)	4 Mflop/s	25 μ s
	20 Mflop/s	5 μ s
	100 Mflop/s	1 μ s
1000 flop (Medium Grain)	10 Mflop/s	100 μ s
	50 Mflop/s	20 μ s
	250 Mflop/s	4 μ s
10000 flop (Coarse Grain)	10 Mflop/s	1000 μ s
	100 Mflop/s	100 μ s
	1000 Mflop/s	10 μ s

Table 1.10 Desired Communication Bandwidth Values

Communication-to-Computation Ratio α	Single-Node Speed P_1	r_∞ should be greater than
0.01 B/flop	4 Mflop/s	0.04 MB/s
	20 Mflop/s	0.2 MB/s
	100 Mflop/s	1 MB/s
0.1 B/flop	10 Mflop/s	1 MB/s
	50 Mflop/s	5 MB/s
	250 Mflop/s	25 MB/s
1 B/flop	10 Mflop/s	10 MB/s
	100 Mflop/s	100 MB/s
	1000 Mflop/s	1000 MB/s

requirement is $N^2 X B$. The algorithm performs a number (e.g., 10,000) of time steps. In each step, a grid point needs Y -flop computation and access to its four neighbor points. The execution time using a single processor is

$$T_1 = YN^2 t_f \quad (1.7)$$

On an n -processor MPP, the data domain can be decomposed into n square regions, each an $(N/\sqrt{n}) \times (N/\sqrt{n})$ subdomain containing N^2/n grid points. In each time step, the computational workload per processor is YN^2/n . Each processor needs to get $XN/\sqrt{n}$ B from each of the four neighbors. Thus on an n -processor machine, the execution time is approximately

$$T_n = \frac{YN^2 t_f}{n} + 8 \left(t_0 + \frac{NX t_c}{\sqrt{n}} \right) \quad (1.8)$$

The constant factor 8 comes from the fact that each processor needs to send a message to and receive one from each of the four neighbors. The *speedup factor* is

$$S = \frac{T_1}{T_n} = \frac{n}{1 + \frac{8t_0}{YN^2 t_f} + \frac{8X t_c}{YN t_f \sqrt{n}}} \quad (1.9)$$

In Fig. 1.10, we draw the speedup curves for four different parallel computers:

- (1) The first machine has 50-Mflop/s processors connected by a slow communication network with $t_0 = 550 \mu\text{s}$ and $r_\infty = 1 \text{ MB/s}$. Such a network is comparable to Ethernet for point-to-point communication. When the problem size $N = 102$, machine A shows a poor speedup (denoted by the square curve at the lower part of Fig. 1.10), less than 10 over 128 processors.
- (2) However, the speedup improves drastically when the problem size N increases 8 times to 8K (denoted by the diamond curve in Fig. 1.10). This increases granularity w and reduces communication-to-computation ratio α .
- (3) The third machine (denoted by the triangle curve in Fig. 1.10) has 100-Mflop/s processors (twice as fast as the first machine) connected by the same slow network. Due to this unbalanced design, the speedup drops back for $N = 809$.
- (4) The last machine (denoted by the circled curve at the top of Fig. 1.10) is more balanced than machine B, with 100-Mflop/s processors connected by a faster network similar to that of IBM SP2, with $t_0 = 46 \mu\text{s}$ and $r_\infty = 35 \text{ MB/s}$. This balanced design results in the best performance among the four parallel computers.

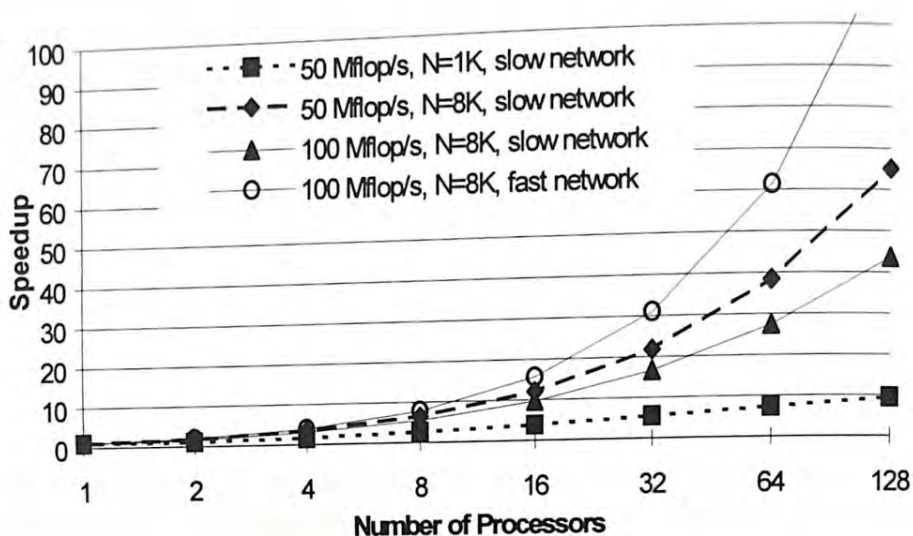


Figure 1.10 Speedup of a 2D PDE solver on four parallel computers.

1.5.3 Design for Scalability

This principle states that when designing a scalable system, one should consider scalability as a main objective right from the start, instead of as an afterthought. Provisions must be made so that the system can either scale up to provide higher performance or scale down to allow affordability or greater cost-effectiveness. Two popular approaches to designing for scalability are *overdesign* and *backward compatibility*.

Overdesign Using the overdesign technique, a system is designed not to barely satisfy the minimal requirements of a current-generation system. The design must include additional features that anticipate improvement from scaled-up systems in the future. These features might be wasted on the current system, but they allow smooth migration into future, improved systems.

Example 1.14 Address space in modern processor design

One of the most important considerations in designing a processor is the *size of its address space*, i.e., the number of byte locations the processor is able to access directly. To quote Gordon Bell, a small address space is the only architecture design mistake that cannot be easily corrected later.

Modern processors support a 64-bit address space, or $2^{64} = 11.8 \times 10^{19}$ B. This huge address space may not be fully utilized by Unix supporting only a 32-b (4-GB) address space. This overdesign in address space will create a much easier transition when the operating system is scaled up to a 64-b Unix. ■

As a counterexample, the processor used in the original IBM PCs, the Intel 8086/8088 microprocessor, has a limited 20-bit or 1-MB address space. The DOS puts a limit of 640-KB for kernel and user software. This 640-KB limit caused many problems for compiler-writers and application software developers.

Since no DOS program (including Windows) can exceed this 640-KB limit, software designers had to invent elaborate techniques (e.g., high memory, expanded memory, and extended memory) to take advantage of the larger address spaces provided by later generations of processors (Intel 286, 386, 486, Pentium, and Pentium Pro).

Example 1.15 Overdesign in the IBM RS6000 SMP servers

The IBM RS6000 SMP is overdesigned for generation scalability. For instance, the first generation of SMPs is based on the PowerPC 601 processor. But the rest of system components, from memory, I/O, power, fan, and the clocking circuitry, are designed to accommodate the next two generations of PowerPC 604 and 620 processors.

Each processor port of the switch has a bandwidth of 600 MB/s, larger than needed by the 601. These overdesigned features make it easy to scale up in a future generation of SMP system by simply upgrading the processors. ■

Backward Compatibility The overdesign technique says that one must consider requirements for *scaled-up* systems when designing a current system. A complementary technique, called *backward compatibility*, says that in designing a hardware or software component, one must consider the requirements of *scaled-down* systems. The scaled-up component should be usable in the old or in a scaled-down system. For instance:

- A new processor should be able to execute binary codes for old processors.
- A parallel program designed to run on n nodes should be able to run on a single node ($n = 1$), maybe with a reduced input data set.
- A program for a supercomputer should be able to run on a workstation. Again, the input data set must fit the limited main memory of a workstation.
- A new version of the operating system should preserve all old functionalities, except obsolete ones, which must be clearly documented.

Example 1.16 Backward compatibility of a data bus width in processors

The CPU bus of modern processors has a 64-b data bus. Some processors support backward compatibility by allowing a 64-b processor to be used in a motherboard that has a smaller data bus width (32, 16, or 8 b).

This is achieved by setting a few control pins of the processor. Suppose the processor needs two bus cycles for a memory *read* (one cycle to send the address and another to read the data from memory). When the *read* is realized in an 8-b motherboard, the processor setting will automatically read the 1 B data from memory in 9 bus cycles. ■

Example 1.17 The parallel operating environment in IBM SP systems

The IBM *parallel operating environment* (POE) was initially designed for IBM ScalableParallel computers (SP1 and SP2), where a user application executes on multiple SP nodes. The POE supports backward compatibility as follows:

- The POE can be executed completely on one node, which allows multiple user tasks to run on one node.
- The node does not have to be an SP node. It can be any RS6000 workstation.
- The POE is implemented on top of the IBM Unix (AIX). Thus existing Unix programs can run in the POE. ■

Caveats Backward compatibility does not mean all features of an old system should be kept. We should weed out obsolete features. In applying the overdesign technique, we should consider the cost incurred and not anticipate into the future too much. Sometimes, though, the overdesign can actually decrease the total development and production costs. This is explained by the following example from Intel experience.

Example 1.18 The Intel 486 DX and 486 SX

When Intel announced the 80486 microprocessor chips, two models were provided: the 486 DX and the 486 SX. Internally, both chips are the same. The only difference is that the *floating-point unit* (FPU) of the DX chip was disabled in the SX chip, which was marketed for users who did not need floating-point capability.

Although the chip area for the FPU is wasted on the SX chip, this overdesign saved Intel from designing two chips. The combined sales of the two chips showed much reduced design and production costs and thus increased profit. ■

1.6 Bibliographic Notes and Problems

Parallel processors are also treated in Flynn [245] and Hennessy and Patterson [305]. Hwang [327], Lenoski and Weber [406], and Culler, et al. [181] have covered scalable multiprocessors, but with different scopes and emphases in their treatment of the subject matters. Discussions of system scalability are given by Hill [308], Rattner [513] and Smith [569]. Ni [464] has given a layered classification of parallel computers. Almasi and Gottlieb [26] cover a broad range of highly parallel computing systems up to 1994.

Basic concepts of clusters of computers are introduced in Pfister [497]. Two special issues on network-based cluster computing appeared in the *Journal of Parallel and Distributed Computing* in February and June of 1997. Recent advances in parallel and cluster computing are also discussed in Bell and Gray [71] and Davy and Dew [190]. Scalable design principles are treated in the articles by Hwang and Xu [333] and Harris and Topham [297].

The PRAM model was proposed by Fortune and Wyllie [246]. The BSP model is described in Valiant [623]. The phase parallel model was presented in Xu and Hwang [657]. Amdahl's law was originally presented in [29].

Gordon Bell has assessed supercomputers and MPPs against modern technologies [67] to [71]. Scalable computer platforms are reported by IBM [335] to [341], DEC [191] to [198], Cray Research [3] and [174], SGI [547] to [550], Sun Microsystems [598] to [600], Microsoft's Wolfpack [573], and Convex/HP [165] and [166], etc. These systems will be studied in subsequent chapters.

Journals and Conferences We recommend the following archival *journals* and *conference proceedings* for additional information on parallel processing, distributed and cluster computing, computer architecture, and scalable computing:

- (1) *IEEE Concurrency: Parallel, Distributed and Mobile Computing*, a dedicated magazine specialized in applications of parallel computers, published first as *IEEE Parallel and Distributed Technology* since 1993.
- (2) *IEEE Computer Magazine*, a general magazine for all subgroups of the IEEE Computer Society, published since 1967. This magazine often carries special issues that are related to parallelism or networking.
- (3) *Communication of The ACM*, a general magazine for all subgroups of the Association for Computing Machinery, published since 1957.
- (4) *Journal of Parallel and Distributed Computing* (JPDC), the very first journal devoted to this area, published by Academic Press, Inc. since 1983.
- (5) *IEEE Transactions on Parallel and Distributed Systems*, a research journal published by IEEE Computer Society since 1990.
- (6) *Parallel Computing*, an international journal on parallel computers and their programming and applications, published by Elsevier Science since 1984.

- (7) *International Conference on Parallel Processing (ICPP)*, the oldest conference in this field since 1972, often held in the Midwest United States.
- (8) *International Parallel Processing Symposium (IPPS)*, an annual symposium sponsored by the IEEE/Computer Society Technical Committee on Parallel Processing since 1986.
- (9) *International Symposium on Computer Architecture (ISCA)*, an annual symposium jointly sponsored by the IEEE/CS Technical Committee on Computer Architecture and by the ACM SIGARCH since 1972.
- (10) *International Symposium on Architectural Support for Programming Languages and Operating Systems*, ACM annual conference since 1975.
- (11) *Scalable High-Performance Computing Conference*, a conference sponsored by the IEEE/CS Tech. Committee on Supercomputer Applications since 1985.
- (12) *International Symposium on High-Performance Computer Architecture (HPCA)*, sponsored by IEEE Computer Society since 1995.
- (13) *International Conference on Algorithms and Architectures for Parallel Processing*, an Asia-based annual conference sponsored by IEEE Singapore and Australia sections since 1995.
- (14) The *COMPAR*, *VAPP*, and *EUROPAR*, three Europe-based conference series on parallel processing. Conference proceedings have been published by North-Holland or Springer-Verlag since 1981.
- (15) *The ACM Symposium on Parallel Algorithms and Architecture*, an annual symposium sponsored by the ACM since 1989.

Homework Problems

Problem 1.1 Define and distinguish among the following terms on scalability:

- (a) Scalability over machine size
- (b) Scalability over problem size
- (c) Resource scalability
- (d) Generation scalability
- (e) Heterogeneity scalability

Problem 1.2 Compare the following abstract parallel computer models. Comment on their differences, relative merits, and limitations in modeling real-life parallel computers and applications:

Chapter 6

System Interconnects and Gigabit Networks

In this chapter, we assess available network technologies and system interconnects. The focus is placed on standard interconnects, commodity networks, and their design and application principles. We study the topologies, protocols, and connectivity of buses, crossbar switches, multistage switches, shared-media networks, cell switched and packet switched networks. These topics pave the way to studying parallel or cluster computer architectures in subsequent chapters.

6.1 Basics of Interconnection Network

Switching fabrics are the core of a modern computer network. We introduce below basic network terms, environments, classifications, standard protocols, and performance metrics of today's high-bandwidth networks. In this book, *networks* refer to the connections among multiple computers. The *system interconnect* glues together subsystems of an integrated computer system. A multicomputer cluster needs to use both types of interconnection structures.

6.1.1 Interconnection Environments

In a scalable multiprocessor, multicomputer cluster, or a distributed system, the building blocks (desktop hosts, servers, mainframes, switches, networks, adapter cards, and peripheral devices etc.) are interconnected at five networking environments: *System or I/O buses or crossbar/multistage switches* are system interconnects often found on the backplane of a single rack of computer platform, or confined within a few interconnected racks in the same room. A dozen network technologies are mapped into a two-dimension space in Fig. 6.1.

A *system-area network* (SAN) connects processors, memory, and interface boards with microstrip cables within a short distance. A *local-area network* (LAN) is confined within a building, a campus, or an enterprise. A *metropolitan-area network* (MAN) covers a whole district or within a city limit. A *wide-area network* (WAN) often appears as an internetwork of a number of smaller networks, thus it can be extended to long distance.

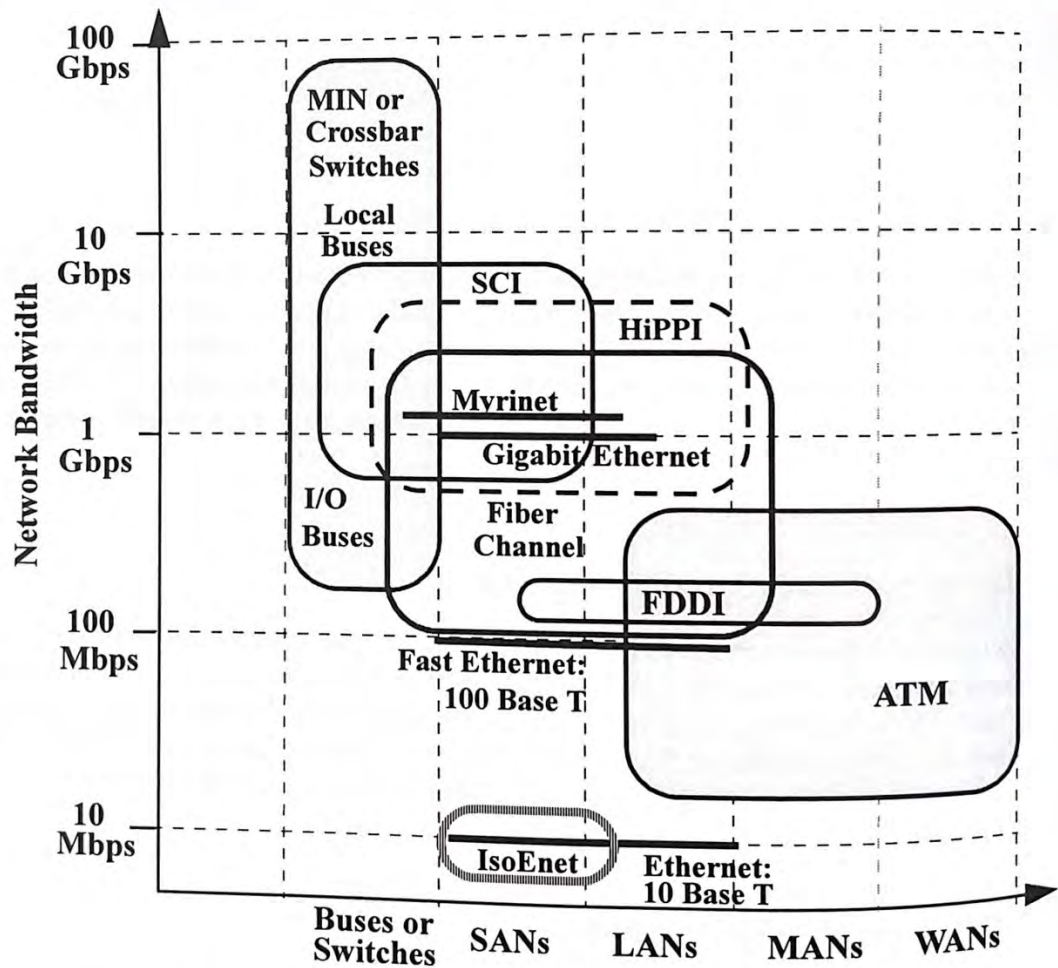


Figure 6.1 System interconnects and network technologies for building scalable parallel and cluster platforms.

These network technologies will be described in subsequent sections. The mapping is based on 1997 technology. The maximum network distance increases from left to right along the horizontal axis. The interconnection environments do overlap in operating

distance. We did not mark the distance scale in Fig. 6.1 due to the overlapping situation. We use Fig. 6.2 to define a set of terms for buses and networks that are widely used in industry and academic. The pins of a processor chip form the *processor bus*. The bus that connects the processor and the memory modules is called the *local bus*, or *memory bus*, the length of which is within tens of centimeters.

A *system bus* (also called *I/O bus*) provides *slots* to connect I/O devices, such as disk drives, tape drives, network interface cards, etc. It is connected to the local bus through an I/O bridge circuit. A system bus is often built on a motherboard, a backplane or center-plane of a system rack, thus limited to 2 m.

Examples of system bus include the AT bus (ISA), the EISA bus, the PCI bus, and the Micro Channel used in IBM PCs, the Sbus in Sun workstations. Note that the term *I/O bus* is often used to refer to the SCSI (*small computer system interface*) bus. Depending on the cabling technology, an internal I/O bus rarely exceeds 3 m and an external SCSI cable can be extended to almost 20 m.

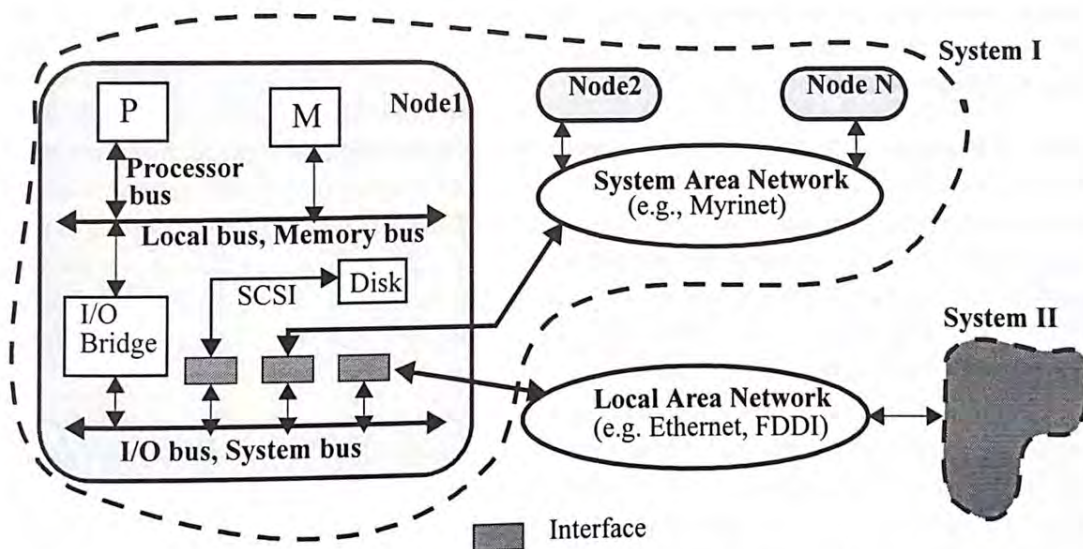


Figure 6.2 Illustration of local bus (memory bus), system bus (I/O bus), system-area network, and local-area network.

Many multiprocessor or multicomputer systems divide their hardware into a number of *nodes*. The network connecting these nodes together to form a single system is called a SAN. In contrast, a LAN is used to connect multiple systems. Of course, a technology (e.g., Ethernet) can be used in SANs and LANs.

What constitutes a system will be made clear in Chapter 9, when we discuss clusters and single system image in detail. A SAN spans 3 to 25 m. A LAN can extend from 25 m to 500 m or 2 km. A MAN normally extends to 25 km or longer.

A WAN is essentially unlimited in distance. It can extend beyond international boundaries. The vertical axis corresponds to increasing *network bandwidth*, defined as the maximum amount of information transferable through the interconnect or network per unit time. Some professionals call it simply the *speed* of the network, often measured as Mbps (*Million bits per second*) or Gbps (*Giga bits per second*). The interconnection technologies shown in Fig. 6.1 yield a bandwidth from 10 Mbps to 100 Gbps.

We shall examine the speed of buses, crossbar switch, and various network technologies in subsequent sections. Specific technology may be upgraded further with higher bandwidth or wider cabling distance. Therefore, some of the circled ranges may move toward the upper right corner of Fig. 6.1. A Tbps (*Tera bits per second*) network speed is entirely possible beyond the year 2000.

6.1.2 Network Components

All switched networks are basically built with three components: *links* (*channels* or *cables*), *switches* (*routers*), and *network interface cards*. Shared-media networks will not use switches. We briefly introduce these network components below before getting into other characteristics of networks.

Links, Channels, or Cables All three terms refer to the physical connections between two hardware units in a computer system. A *link* can be implemented with copper wires or fiber-optic cables. The simplest link is an *unshielded twisted pair* (UTP) of copper wires. Using copper links is cheaper but limited by a short cabling distance due to a signaling problem. The cabling distance can be extended to some extent, if *shielded twisted pairs* (STPs) are used. Fiber-optic cables are more expensive, but do offer a much higher bandwidth and a longer cabling distance.

Some authors call a link a *channel* or a *cable*. A link may connect two switches or connect a switch and a network interface attached to a host node. A *short link* may contain only one logic signal at a time, while a *long link* behaves as a transmission line, allowing a string of logical signals to propagate along the line at the same time. A *serial link* or a *thin link* has one bit line to be shared by data, and control signals in a multiplexed manner.

A *wide link* or a *parallel link* has multiple bit lines, allowing parallel transmission of data and control information. The links are driven by one of two clocking mechanisms: synchronous or asynchronous. *Synchronous clocking* means the source and destination ends operate with the same global clock. *Asynchronous links* use some embedded clock encoding mechanism to allow both ends to handshake with different clock rates.

Switches *Switches* are needed in building switched networks. In general, a switch has multiple input and output ports. Each input port has a receiver and input *buffer* to handle the arriving packets or cells. Each output port has a transmitter to pass the outgoing data signal to a communication link connected to another switch or a network interface. A four-input and four-output switch is shown in Fig. 6.3.

An internal *crossbar* is used to establish n connections between n inputs and n outputs simultaneously. Each *crosspoint* can be switched on (as marked by dots in Fig. 6.3) or off under program control. This number n is often called the *degree* of the switch. Multiple switches and links are often used to build a large switched network following a chosen topology.

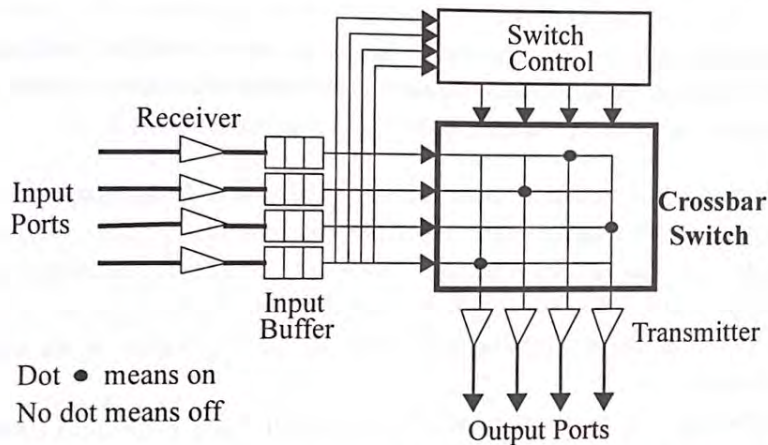


Figure 6.3 A four-port crossbar switch providing any permutation connection between 4 inputs and 4 outputs.

Network Interface Circuitry A *network interface circuitry* (NIC) is often used to connect a host computer to a LAN or to another network. Thus some professionals call it a *host interface*. The NIC must be able to handle two-way traffic between the host and the network. Therefore, the architecture of a NIC depends on the network and the host. Different hosts may require different interface cards, even connected to the same network.

A typical NIC contains an embedded processor, some input and output buffering, and some control memory and logic. Its functions include the formatting of packets or cells, some control memory and logic. Its functions include the formatting of packets or cells, routing path selection, coherence checking, flow and error control, etc. Therefore, the cost of a NIC is determined by the port size, storage capacity, processing power, and control circuitry; often resulting in a higher complexity than that of a routing switch.

6.1.3 Network Characteristics

We distinguish below between basic network terms; often these terms appear as pairs with two opposite characteristics. Or one can consider each characteristic a dimensionality of the entire network design space.

Static versus Dynamic Networks Based on connectivity and control strategy, we classify interconnection networks into several categories. *Static networks* are built with

point-to-point links that will not change during program execution. In other words, static networks have fixed connections among processing nodes. Static networks are also known as *direct networks*, because only one host computer is connected to each node switch. On the other hand, a *dynamic network* is implemented with switched channels, which are dynamically configured to meet the communication requirements in user programs.

Dynamic networks include system buses, crossbar switches, multistage networks, and various SANs, LANs, MANs, and WANs. Dynamic networks are also called *indirect networks*, where multiple hosts are connected to specific node switches and routing is done by a sequence of switching decisions. Some hybrid networks having both types of nodes support either static or dynamic connections upon program control.

Shared-Media Networks In a *shared-media network*, physical links (copper or fiber) are accessible by all computers connected to the network. For this reason, shared-media networks are also known as *multiaccess networks*. However, only one requester is granted access at a time. In other words, the bandwidth is shared by multiple requesters. The most challenging problem in designing a shared-media network is to allocate equal bandwidth to all competing requesters.

Shared-media networks can be implemented with FDDI, Fast Ethernet (100Base-T), Fiber Channel technologies. Since no switches are used, shared-media networks often cost less than the corresponding switched networks. The allocation of available bandwidth to all requesters is known as the *bandwidth management* problem. The selection of a network access protocol will greatly affect the efficiency of a bandwidth management system.

Switched Networks In contrast, a *switched network* uses switching fabric to allocate or deallocate the media resources to one requester at a time. In other words, the access to a media link is not shared by more than one user at a time. Many shared-media networks can be converted to a switched design at an increased cost. For example, most Ethernet and FDDI rings share the media, but switched Ethernet or switched FDDI rings are also available. On the other hand, all ATM networks are inherently switched networks.

Switched networks are often designed with three options: *circuit switching*, *packet switching*, and *cell switching*, as characterized below.

Circuit Switched Networks Switching is needed to determine the routing path between a pair of nodes for point-to-point communication or among multiple nodes for collective communications. In *circuit switching*, the entire path (of links and buffers) from the source node to the receiving node is reserved for the entire period of transmission.

This is similar to the railroad transportation by a long train on a reserved track, where no other train is allowed to share the same track at the same time. If the train is very long, the reserved track may be held for a long period before the next train can pass through.

Packet Switched Networks For a *packet network*, a long message is broken into a sequence of small packets. Each packet contains the routing information and a segment of

data payload. Packets from the same message can be routed separately on different paths. This is similar to use smaller trucks (or cars) in dividing a whole trainload of goods.

Packet switching requires disassembly of the message into packets before the transmission and message reassembly upon arrival of all packets. Packet switching is supposed to yield better utilization of network resources, because many packets from different messages can share the same network links and buffers more efficiently.

Cell Switched Networks In a packet network, packets from different messages may differ in length. In other words, long and short packets may be serialized in a long string of packets. A small packet following a long packet may experience excessive delays, until the long packet is removed from the shared path. The *cell switching* is improved over *packet switching* by partitioning a long packet into fixed-size small *cells*.

The purpose is to alleviate the blocking imposed by long packets as depicted in Fig. 6.4. In practice, a long packet may be as long as 4096 B and a short packet may have only a few bytes. The waiting time is large when the small packet of an isochronous traffic such as voice traffic is waiting behind a large packet. In a *cell switched network*, a typical cell is 56 B. Constant transmission delay is possible in a cell network.

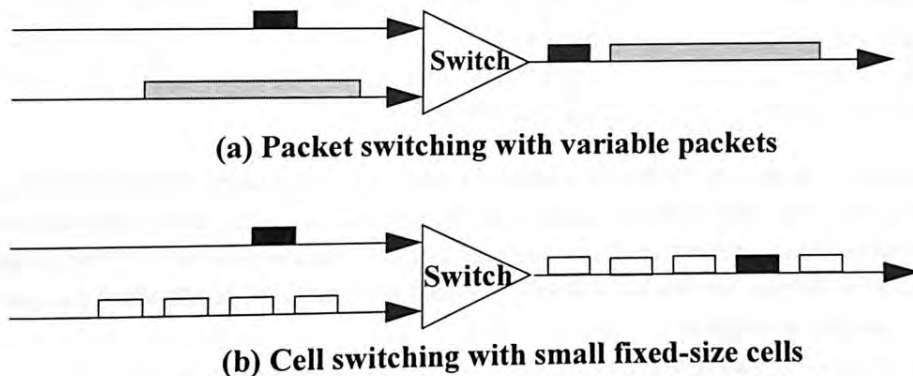


Figure 6.4 **Serialization of packets and cells in two types of switched networks.** (Courtesy of C. Patridge, *Gigabit Networking*, (Figs. 3.2 and 3.3 on p. 45), ©1994 Addison-Wesley Pub. Co. [489], Reprinted by permission of Addison Wesley Longman).

Another advantage of cell networks over packet networks lies in the simplified hardware design of the cell switches, because of handling small, fixed-size cells instead of variable-length packets. The cell switching became popular with the introduction of ATM networks. Fragmenting data messages into many cells may lead to very low network performance, since the cells could be lost in transit and retransmission is often not allowed. We shall deal with these problems when ATM cell switches and ATM networks are discussed in Section 6.5.

6.1.4 Network Performance Metrics

Latency and bandwidth are two fundamental metrics used to assess the performance of a network or of a system interconnect. We describe below various aspects of these two performance metrics.

Communication Latency We consider the total time needed to transmit a message from a source node to a destination node in a multicomputer system. This latency consists of four time components:

- (1) The *software overhead* associated with sending and receiving the message at both ends of the network
- (2) The *channel delay* caused by the channel occupancy (or the total message length divided by the channel bandwidth)
- (3) The *routing delay* caused by the time spent in the successive switches in making a sequence of routing decisions along the routing path
- (4) The *contention delay* caused by traffic contentions in the network

The software overhead is largely contributed by the host kernels in handling the message at both ends. The channel delay is often determined by the bottleneck link or channel. The routing delay is proportional to the *routing distance* (or the *path length* or the number of *hops* between the ends). The contention delay is most difficult to predict, because it depends on the traffic conditions in the network.

Network Latency Both the software overhead and the contention delay depend on the program behavior. For this reason, hardware designers consider only the sum of the middle two terms, channel delay and switch delay, as the *network latency*. This quantity is totally determined by the network hardware characteristics, independent of the program behavior and traffic conditions.

In a light-traffic (contention-free) message-passing networked system, the network latency (usually around 1 μ s) is much smaller than the software overheads or contention delay (in tens or hundreds of microseconds). Overall, the communication latency is a function of the *kernel overheads* at both ends, the *message length* (including the header), the *channel bandwidth*, the *switch delay* (or the routing algorithm used), the *path length*, and the *network traffic* (or program behavior).

In order to reduce the latency, a major effort should be conducted at reducing or hiding the software overhead time. Various latency reduction or hiding techniques will be discussed at the end of this chapter.

Per-Port Bandwidth We have already defined the *per-port bandwidth* earlier as the maximum number of bits (or bytes) that can be transmitted per second from any port to any other port in a network. For example, the IBM SP2 *high-performance switch* (HPS) has a per-port bandwidth of 40 MB/s.

In a *symmetric network*, all the ports are topologically equivalent. In other words, the per-port bandwidth is independent of the port location in a symmetric network. Otherwise, we have an *asymmetric network*. The per-port bandwidth for an asymmetric network is defined as the minimum of all per-port bandwidths.

Aggregate Bandwidth The *aggregate bandwidth* of a given network is defined as the maximum number of bits (or bytes) that can be transmitted from one half of the nodes to another half of the nodes per second. For example, the HPS is a symmetric network consisting of n nodes (ports), where n has an upper bound of 512.

With a per-port bandwidth of 40 MB/s. The aggregated bandwidth of a 512-node HPS is calculated as: $(40 \times 512) / 2 = 10.24$ GB/s. The division by 2 is attributed to the fact that bidirectional traffic should be counted only once.

Bisection Bandwidth The aggregate bandwidth is also related to a topological term, called *bisection bandwidth*. This is defined as the maximum number of bits (or bytes) per second crossing all the wires on the bisection plane dividing the network into two equal halves.

Let b be the number of links crossing the bisection plane and w be the number of wires per link (called the *link width* or *channel width*). The product bw is called *bisection width*, representing the total number of wires crossing the bisection plane. If each wire transmits r b/s, the bisection bandwidth is defined as $B = bwr$ b/s. The time to move M B across a network has as a lower bound the ratio M/B .

In general, the network bandwidth is sensitive to the *network topology*, the *channel width*, the *network size* (or the port number), the *channel number*, the *switch degree*, and *network clock rate*. Only hardware architecture affects the network bandwidth, which is independent of program behavior and traffic patterns. This is also true for network latency. The communication latency is affected by both machine and program behaviors.

6.2 Network Topologies and Properties

Basic network properties are revealed along with their architectural implications. Before examining network topology, we define parameters often used to estimate the complexity, communication efficiency, and cost of a network.

6.2.1 Topological and Functional Properties

Basically, an interconnection network is represented by a *directed* or an *undirected graph* of a finite number of *nodes* linked by *edges*. A node could be a switch, or a host, or a device. An edge may refer to a link or channel. The number of nodes in the graph corresponds to the *network size*.

Node Degree The number of edges (links or channels) incident on a node is called the *node degree* d . In case of unidirectional channels, the number of channels into a node is the *in degree*, and that out of a node is the *out degree*. The node degree is the sum of the two. It reflects the number of I/O ports required per node, and thus the cost of a node. The node degree should be kept as small as possible, to reduce the implied cost.

Network Diameter The *diameter* D of a network is the maximum path between any two nodes. The path length is measured by the number of links traversed. The network diameter indicates the maximum number of distinct hops between any two nodes, thus providing a simple figure to estimate the communication delay on a data network.

Blocking versus Nonblocking Networks Message passing in a network can be designed to be blocking or nonblocking.

A *blocking network* allows a message to be buffered for later transmission in case of channel conflicts among multiple messages. Therefore, blocking networks and *buffered networks* are used interchangeably.

In a *nonblocking network*, no buffers are used. Alternate paths are used to resolve conflicts among messages. Most nonblocking networks use switches to route messages.

Synchronous versus Asynchronous Networks In a *synchronous network*, both the sender and receiver must be synchronized in time and space. This is similar to a telephone network requiring synchronization between caller and callee. Most synchronous networks do use buffers and thus are blocking networks.

Asynchronous networks do not use buffers and are nonblocking in nature. No synchronization of sender and receiver is needed in an asynchronous network. This is similar to the postal delivery network. Some communication networks are built to support both synchronous and asynchronous communications.

Functionality and Complexity - *Functionality* refers to special features or mechanisms built inside a network to support packet routing, interrupt handling, fast synchronization, message combining, data coherence, switch conflicts, channel conflicts, flow control, fault tolerance, lost packet handling, data prefetching, latency tolerance, bandwidth management, etc.

The complexity affects the network construction cost. Complexity is often determined by ports, cables, switches, buffers, connectors, arbitrators, and interface logic required in a network construction. The control circuitry, buffer memory, and power supply are also part of the network complexity.

Symmetry and Scalability We call a network *symmetric* if the topology looks the same from any node in the network. Otherwise, the network is called *asymmetric*. Symmetric networks are easier to lay out on a VLSI circuit or on a PC board. It is also easier to route messages in a symmetric communication network.

Network *scalability* refers to the ability of a network to be modularly expandable. The expansion must be accompanied by proportional increase in expected performance. Network scalability is limited not only by topological properties such as node degree, diameters, symmetry, bisection bandwidth, and complexity, but also by physical technology and management requirements such as packaging, power dissipation, cooling, programmed control, traffic management, etc.

6.2.2 Routing Schemes and Functions

Two different ways of routing packets in a switched network are introduced below: The *store-and-forward* scheme was used in the first generation of message-passing multi-computers; the *cut-through* or *wormhole routing* scheme is implemented in more of today's multicomputer or cluster systems.

Store-and-Forward Routing The entire packet must be stored in a *packet buffer* in a node before it is forwarded to an outgoing link. Thus, successive packets are transmitted sequentially without overlapping in time. In Fig. 6.5(a), we show the transmission of a packet consisting of four cells from node 1 to node 4 via two intermediate nodes (node 2 and node 3). Each packet consists of four cells. The header is labeled as **h**, followed by three data cells labeled as **a**, **b**, and **c**.

The header is used to determine the routing path. Each node consumes 4 time units (equal to the packet length) to pass the packet to the next node. Therefore, it takes 16 time units (cycles) to pass the packet from N1 to N4 if there are no resource conflicts or deadlock situation along the routing path. When the network is jammed, the routing time could be even longer.

Cut-Through Routing Each node uses a *flit buffer* to hold one *flit*, one cell of the packet, at a time. The flit is automatically forwarded to an outgoing link, once the header is decoded. All data flits in the same packet follow the same path that the header traverses. Therefore, each node forwards the packet one flit down the path per unit time. Successive packets are forwarded in a pipelined fashion.

In Fig. 6.5(b), it takes only 7 time units for node 4 to receive the entire packet of 4 flits, compared with 16 cycles needed in the store-and-forward routing. Let L be the packet length and N be the number of nodes from source to destination. In general, the store-and-forward scheme takes NL time units, while the wormhole routing takes only $L + N - 1$ time units to pass through a path of $N - 1$ hops. A speedup of $NL / L + N - 1$ is expected. For a sufficient long packet, the limiting speedup approaches N times faster for wormhole routing over the traditional routing scheme.

A wormhole router overlaps the transmission of successive packets by using the network resources (links, switches, and network interfaces) along the selected path as a transmission pipeline (Fig. 6.5). The packet bypasses an intermediate node without copy and storage delays.

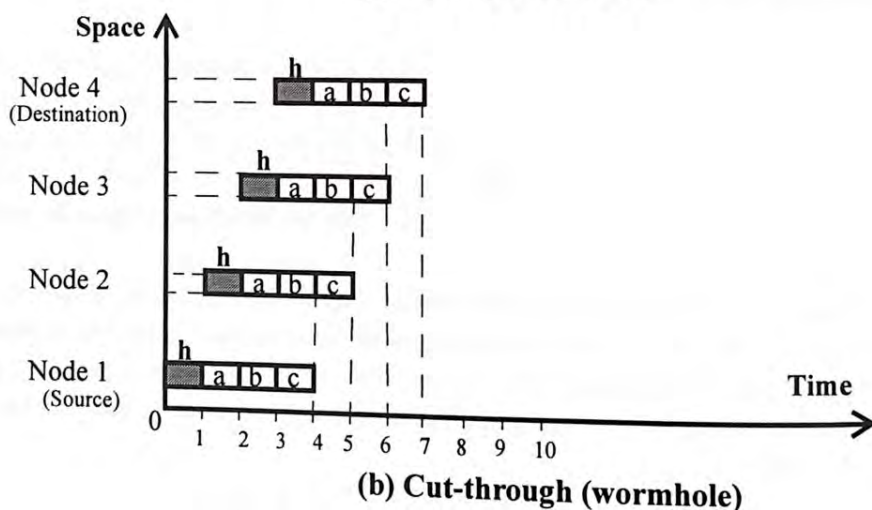
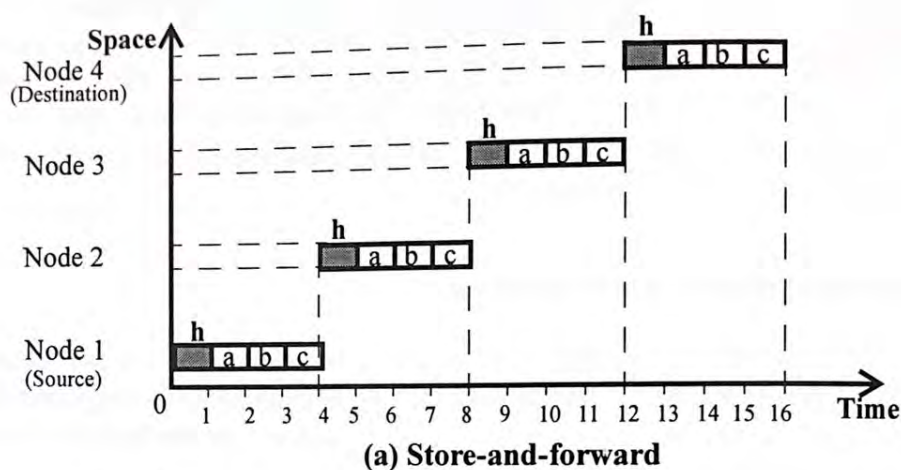


Figure 6.5 Store-and-forward versus cut-through routing in a packet switched network (h: header; a, b, and c are data elements).

The wormhole routing reduces the connection latency from 500 μs in store-and-forward scheme to 20 μs in commercial switch designs. Interested readers may refer to Dally and Seitz [185] for details of wormhole routing.

Data Routing Functions A *data network* is used for internode data communications. We specify below primitive data routing functions implemented on a routing network. Commonly seen data-routing functions include *shifting*, *rotation*, *permutation* (one-to-one), *broadcast* (one-to-all), *multicast* (many-to-many), *personalized communication* (one-to-many), *shuffle*, *exchange*, etc.

Permutations For n objects, there are $n!$ permutations by which n objects can be reordered. The set of all permutations forms a *permutation group* with respect to a compo-

sition operation. One can use the cycle notation in group theory to specify a permutation function.

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

One can use a crossbar switch to implement any of the permutation patterns. Multi-stage 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. When n is large, the permutation speed may decide the basic performance of a data routing network.

Perfect Shuffle *Perfect shuffle* is a special permutation function suggested by Harold Stone [588] for parallel processing applications. The mapping from a perfect shuffle is shown in Fig. 6.6a. The *inverse perfect shuffle* is defined in Fig. 6.6.

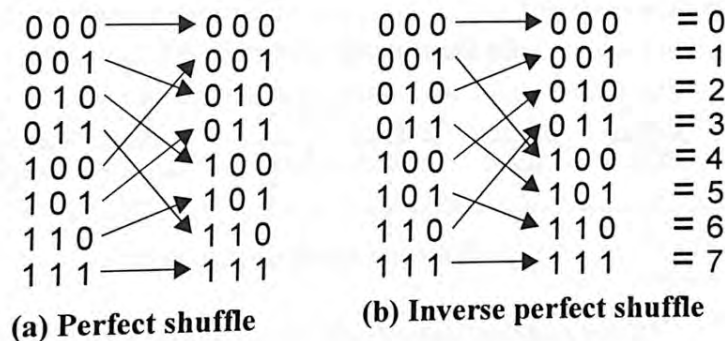


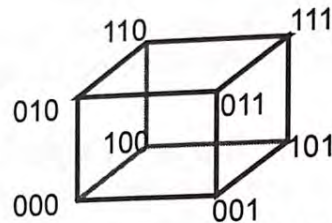
Figure 6.6 Perfect shuffle and its inverse mapping over eight objects.

(Courtesy of H. Stone; reprinted with permission from *IEEE Trans. Computers* [588], 1971)

Shuffling n cards (objects) evenly is performed by a game dealer. One can represent each card in a deck of $n = 2^k$ cards by a k -b binary number $x = (x_{k-1}, \dots, x_1, x_0)$. The perfect shuffle maps card x to card y , where the representation $y = (x_{k-2}, \dots, x_1, x_0, x_{k-1})$ is obtained by shifting x one bit to the left and wrapping around the most to the least significant position. This gives a binary method to generate the perfect shuffle pattern.

Hypercube Routing Functions A three-dimensional binary cube is shown in Fig. 6.7(a). Three routing functions are defined by three bits in the node address. For example, one can exchange the data between adjacent nodes differing in the least significant bit C_0 , as shown in Fig. 6.7(b). Similarly, two other routing patterns can be obtained by checking

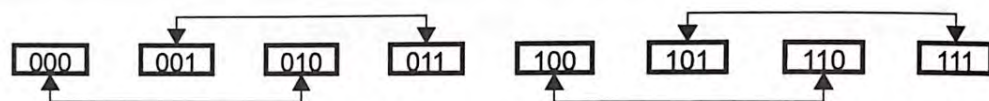
the middle bit C_1 [Fig. 6.7(c)] and the most significant bit C_2 [Fig. 6.7(d)], respectively. In general, an n -dimensional hypercube has n routing functions, defined by each bit of the n -bit address.



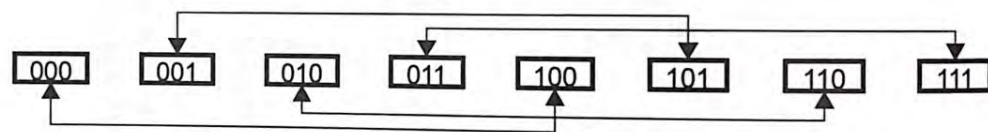
(a) A binary cube with 8 nodes coded in binary $C_2C_1C_0$



(b) Routing by inverting bit C_0



(c) Routing by inverting bit C_1



(d) Routing by inverting bit C_2

Figure 6.7 Three routing functions on a binary cube.

Broadcast and Multicast *Broadcast* is a one-to-all mapping. This can be easily achieved in an SIMD computer using a broadcast bus. A message-passing multicomputer uses a *multicast* mechanism to broadcast messages. Multicast corresponds to mapping from one subset to another subset of nodes (many-to-many communication). *Personalized broadcast* sends personalized messages to only selected receivers. Broadcast is often treated as a global operation in a multicomputer.

6.2.3 Networking Topologies

Static networks use point-to-point links that are fixed once connected. This type of network often shares the media and is more suitable for predictable traffic patterns. These topologies can be also applied to build switched networks.

Linear Array This is the simplest network in which N nodes are connected by $N - 1$ links forming a single cascade [Fig. 6.8(a)]. Internal nodes have degree 2. The diameter is $N - 1$, which is rather long for a large N . The bisection width $b = 1$. The structure is not symmetric and poses a communication inefficiency when N becomes very large.

For the degenerate case of $N = 2$, it is rather economical to implement a linear array. As N increases, another topology may be better. It should be noted that a *linear array* is very different from a *bus*, which is timeshared through switching among many nodes attached to it. A linear array allows concurrent use of different sections (links) of the cascade by different users.

Ring A *ring* is obtained by connecting the two terminal nodes of a linear array with one more link to form a closed loop [Fig. 6.8(b)]. A ring can be unidirectional or bidirectional. It is symmetric with a constant node degree of 2. The diameter equals $N/2$ for a bidirectional ring, and $N - 1$ for a 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.

Chordal Ring By increasing the node degree from 2 to 3 or to 4, we obtain two *chordal rings* shown in Fig. 6.8(c) and Fig. 6.8(d), respectively. One and two extra links are added to produce these two chordal rings, respectively. In general, the more links added, the higher the node degree and the shorter the network diameter. Chordal rings with higher node degree require a lot of more links to implement.

Comparing the 16-node ring with the two chordal rings, we see the network diameter drops from 8 to 5 and to 3, respectively. In the extreme, a *completely connected network* in Fig. 6.8(f) has the highest node degree of $N - 1$ with the shortest possible diameter of 1. The tradeoffs between node degree and network diameter affect the cost/performance ratio, as well as the scalability and fault tolerance of a given interconnection network.

Barrel Shifter As shown in Fig. 6.8(e), the *barrel shifter* is modified from the ring by adding extra links from each node to distance nodes with integer power of 2 hops away. 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 node degree is 7 and a diameter of 2. All the topologies presented have lower complexity than that of a completely connected network, which is used only if the network size is rather small.

Tree A *binary tree* of 7 nodes in three levels is shown in Fig. 6.9(a). The node degree of a binary tree is 3. A k -level binary tree has $N = 2^k - 1$ nodes. A binary tree can be extended to a *multiway tree* with more than two offsprings per node. In general, an m -way tree has $N = m^k - 1$ nodes. Each nonleaf (parent) node may have up to m child nodes.

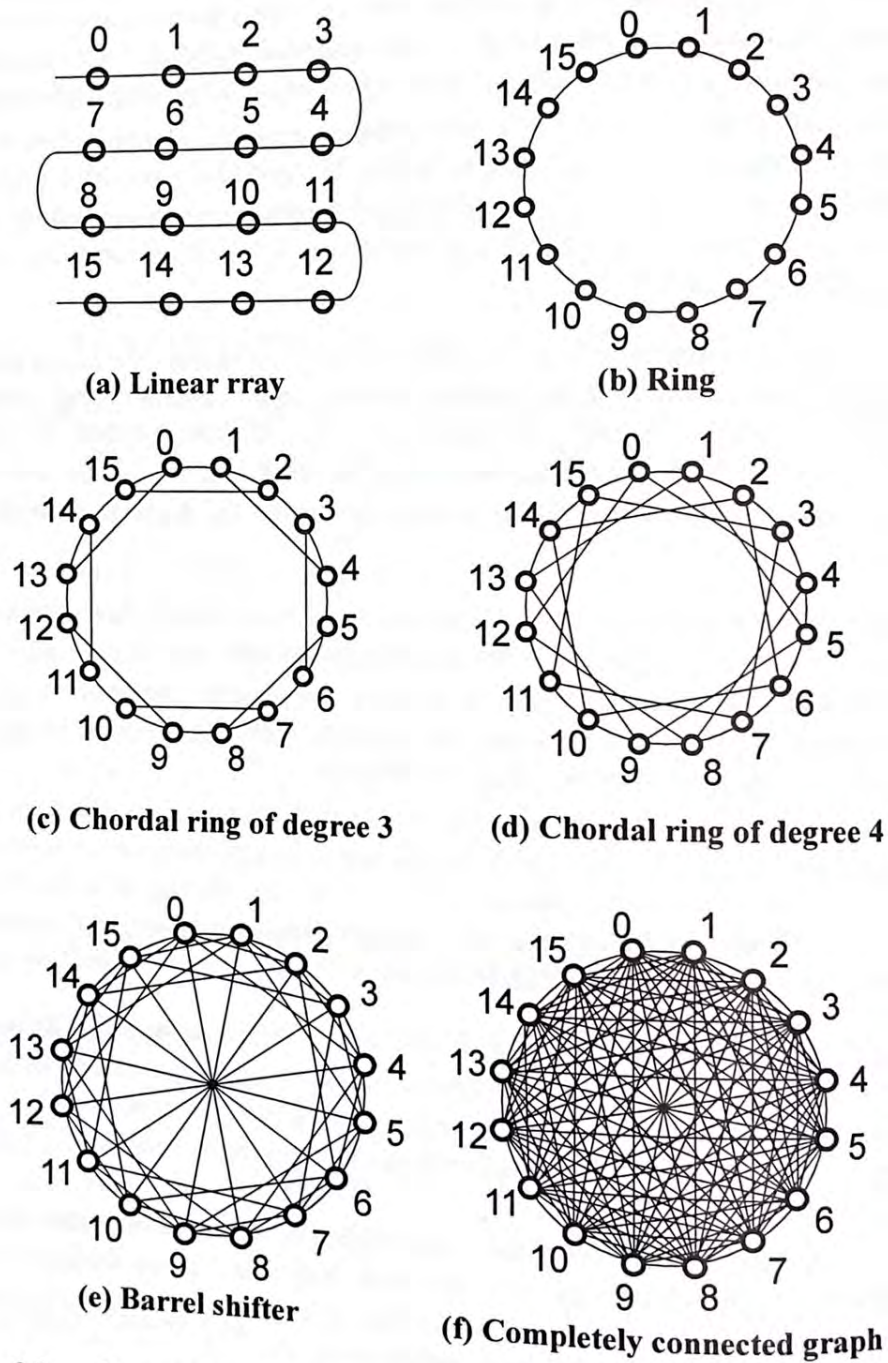


Figure 6.8 Six network topologies with increasing node degree and connectivity from a linear array to a ring, two chordal rings, a barrel shifter, and a completely connected graph.

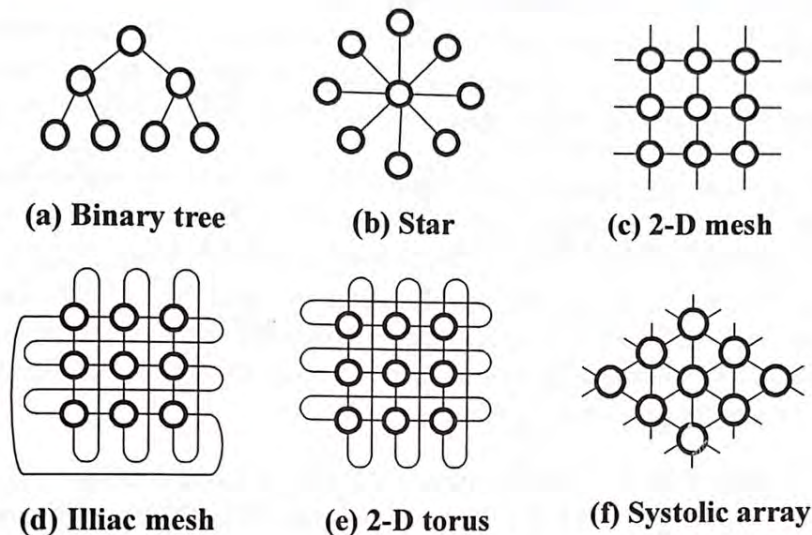


Figure 6.9 Six additional topologies for network construction.

Star The *star* [Fig. 6.9(b)] is a two-level, multiway tree with the highest node degree equal to $d = N - 1$ and the smallest diameter of 2. One problem in using a tree or a star network is the bottleneck effect towards the root of a tree or towards the center node of a star network. Traffic becomes much heavier at the root or the center node, when leaf nodes communicate with each other.

Mesh A two-dimensional, 3×3 *mesh* network is shown in Fig. 6.9(c). This is a popular architecture which has been implemented in a number of systems in the past, such as the Goodyear MPP and Intel Paragon.

In general, a k -dimensional mesh with n nodes on each dimension has an interior node degree of $2k$, and the network diameter is $k(n - 1)$. Note that a mesh is not a symmetric network, because the boundary nodes are different from the interior nodes.

Illiac Mesh In Fig. 6.9(d) we show a variation of the mesh by allowing wraparound connections vertically and a snakelike loop horizontally. This drawing is a reduced version of the *Illiac network* [105]. The original Illiac network assumed an 8×8 mesh with a node degree of 4 and a diameter of 7.

In general, an $n \times n$ Illiac mesh should have a diameter of $d = n - 1$, which is only half of the diameter of a 2-D mesh. This Illiac mesh shown in Fig. 6.9(d) is topologically equivalent to a chordal ring of degree 4 shown in Fig. 6.8(d).

Torus A 2-dimensional *torus* is shown in Fig. 6.9(e). This architecture extends from the mesh by having wraparound connections. A torus would have about half of the diameter

of a mesh. Or one can say that this topology combines the advantages of ring and mesh. Both mesh and torus can be extended to higher dimensions.

For example, Cray Research Inc. chooses to implement a 3D torus network in their T3D/T3E series. The torus is a symmetric topology, whereas a mesh is not. All added wraparound connections help reduce the torus diameter and restore the symmetry.

Systolic Array This is a special class of application-driven array architectures designed for VLSI implementation of certain data flow algorithms. What is shown in Fig. 6.9(f) is a systolic array specially designed for matrix-to-matrix multiplication.

Static systolic arrays are pipelined with multidirectional flow of data streams. With a fixed structure, a systolic array is supposed to match the communication structure of a given algorithm. The structure cannot implement other algorithms efficiently once it is optimized for a given application.

Fat Tree To alleviate the bottleneck towards the root of a tree structure, Leiserson [403] introduced the multiway *fat tree* in 1985. The number of links between parent and child nodes in a fat tree increases as we traverse from a leaf node toward the root.

The fat tree is more like a natural tree in that the trunks get thicker toward the root. Fat tree architecture has been implemented in Connection Machine CM5 system. A four-way fat tree was adapted in building the data network in CM5. The fat tree built in CM5 allows each node to have 4 child nodes and 2 or 4 parent nodes as illustrated by the internode connections in Fig. 6.10.

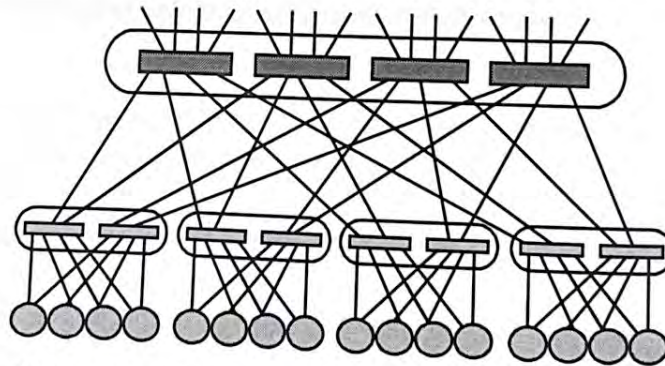


Figure 6.10 Four-way fat tree implemented in the CM5 (Courtesy C. Leiserson et al.[403], *IEEE Trans. Computers*, Oct. 1985),

Hypercube Formally, a hypercube should be called a *binary n -cube*. Binary refers to the fact that each dimension of the cube has two nodes, where n is called the *dimensionality* of the hypercube. In general, an n -cube consists of $N = 2^n$ nodes spanning along n dimensions. A 3-cube with 8 nodes is shown in Fig. 6.11(a). A 4-cube is formed by interconnecting the corresponding nodes of two 3-cubes as illustrated in Fig. 6.11(b). The node degree of an n -cube equals n , and so does the network diameter. In fact, the node degree

increases linearly with the dimensionality, making it difficult to scale up the size of a hypercube architecture.

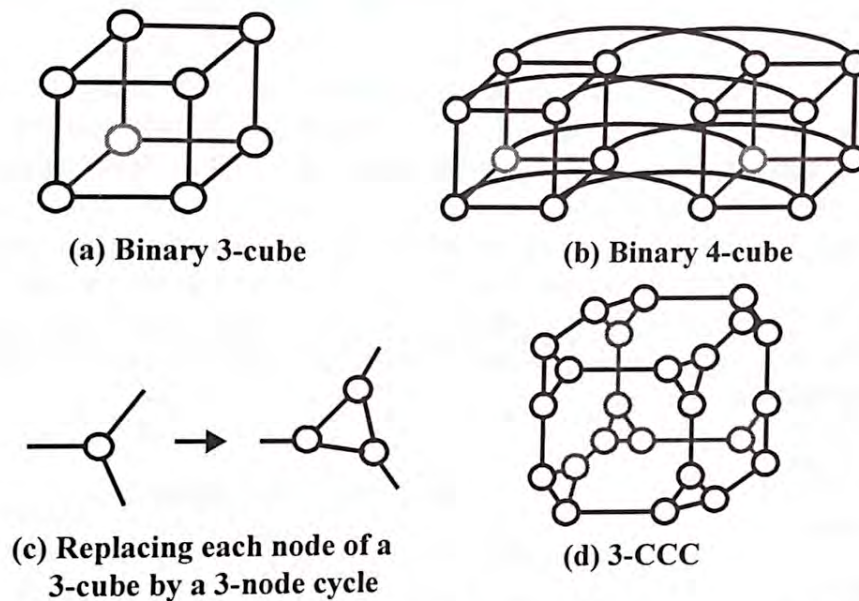


Figure 6.11 Binary hypercube and cube-connected cycles (CCCs).

Binary hypercube has been a very popular architecture for research and development in the 1980s. Intel iPSC/1, iPSC/2, nCUBE, and Connection Machine's CM2 machines were built with the hypercube architecture. The architecture has dense connections. Binary trees, meshes, and many other lower-dimension topologies can be embedded in a hypercube. With poor scalability and difficulty in packaging high-dimension cubes, the hypercube architecture is gradually replaced by lower-dimension architectures.

For example, Connection Machine CM5 chooses the fat tree over the hypercube implemented in CM2. Intel Paragon chooses a two-dimensional mesh over its hypercube predecessors. Topological equivalence has been established among a number of network architectures.

The bottom line for an architecture to survive lies primarily in cost/performance trade-offs in packaging, efficiency, scalability, and performance. With the introduction of wormhole routing and pipelined network operations, the choice of a regular topology becomes less important than before. In other words, architectural topology becomes less sensitive or less critical to determine the performance of an interconnection network. It is the media and switching technology that dominate.

Cube-Connected Cycles This architecture is modified from the hypercube. As illustrated in Fig. 6.11(c), a 3-cube is modified to form *3-cube-connected cycles* (3-CCCs)

shown in Fig. 6.11(d). The idea is to cut off each corner node of the 3-cube and replace it by a cycle of 3 nodes.

In general, one can construct a *k*-cube-connected cycle from a *k*-cube with $n = 2^k$ cycle nodes. The idea is to replace each vertex of the *k*-dimensional hypercube by a ring of *k* nodes. A *k*-cube can be thus transformed to a *k*-CCC with $k \times 2^k$ nodes.

The 3-CCC shown in Fig. 6.11(d) has a diameter of 6, twice that of the original 3-cube. In general, the network diameter of a *k*-CCC equals $2k$. The major improvement of a CCC over a hypercube lies in a fixed node degree of 3, which is independent of the dimension of the underlying hypercube.

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

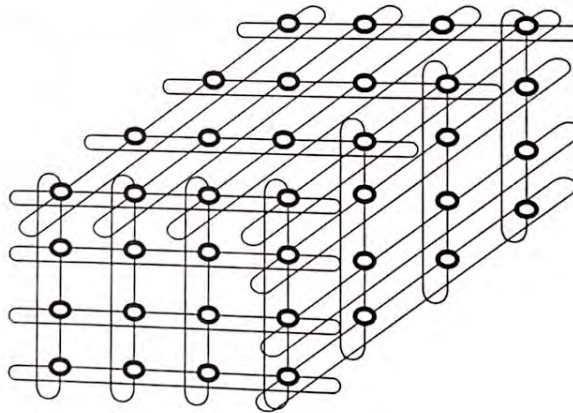


Figure 6.12 A 4-ary 3-cube network, hidden parts not shown.

***k*-ary *n*-Cube Networks** Rings, meshes, tori, binary *n*-cubes (hypercube), and Omega networks are all topologically isomorphic to a family of *k*-ary *n*-cube networks. In Fig. 6.12 we show a 4-ary 3-cube network with $k = 4$ nodes at each dimension. For clarity, the hidden nodes and hidden links are not shown.

There are $N = k^n$, ($k = \sqrt[n]{N}$, $n = \log_k N$) nodes in this network. A node in a *k*-ary *n*-cube is identifiable by an *n*-digit radix-*k* number $A = a_0 a_1 a_2 \dots a_n$, where a_i represents the node's position in the *i*-th dimension. Traditionally, low-dimensional *k*-ary *n*-cubes are called *tori*, such as for $n = 2$ or 3. Higher-dimension binary *n*-cubes are called *hypercubes*. Both fat trees and *k*-ary *n*-cube networks are universal in simulating other networks.

Dally [183] has revealed some interesting properties of *k*-ary *n*-cube networks. The cost of most networks is dominated by the amount of wires, rather than by the number of

switches required. If limited by a constant wire bisection, low-dimension networks with wider channels provide lower latency, less contention, and higher throughput than high-dimension networks with narrower channels.

Summary of Network Topologies In Table 6.1, we summarize important characteristics of the above network topologies. Those networks with node degree of 4 or less are more desirable. For example, the INMOS Transputer chip is a microprocessor with four ports linking to four channels in a 2D mesh architecture. The completely connected and star networks are both very bad for having a high node degree. The hypercube node degree increases with $\log_2 N$, which is also bad when N becomes very large.

Table 6.1 Topological Properties of Static-Connection Networks

Network Type	Node Degree d	Network Diameter D	No. of Links l	Bisection Width B	Symmetry	Network Size and Remarks
Linear array	2	$N - 1$	$N - 1$	1	No	N nodes
Ring	2	$N/2$	N	2	Yes	N nodes
Completely connected	$N - 1$	1	$N(N-1)/2$	$(N/2)^2$	Yes	N nodes
Binary tree	3	$2(h - 1)$	$N - 1$	1	No	Tree height $h = \lceil \log_2 N \rceil$
Star	$N - 1$	2	$N - 1$	$N/2$	No	N nodes
2D Mesh	4	$2(r - 1)$	$2N - 2r$	r	No	$r \times r$ mesh for $r = \sqrt{N}$
Illiac Mesh	4	$r - 1$	$2N$	$2r$	No	Chordal ring with $r = \sqrt{N}$
2D Torus	4	$2\lceil r/2 \rceil$	$2N$	$2r$	Yes	$r \times r$ torus for $r = \sqrt{N}$
Hypercube	n	n	$nN/2$	$N/2$	Yes	$N = 2^n$ nodes,
CCC	3	$2k - 1 + \lceil k/2 \rceil$	$3N/2$	$N/(2k)$	Yes	$N = k \cdot 2k$ nodes with cycle $k \geq 3$
k -ary n -cube	$2n$	$n\lceil k/2 \rceil$	nN	$2k^{n-1}$	Yes	$N = k^n$ nodes

With pipelined wormhole routing, the network diameter becomes less critical than before, because the hardware delay between any two nodes is almost independent of the path length. The number of links used affects the network cost most. The bisection width and channel width dominate the network bandwidth. Network symmetry is often desired to enhance scalability and routing efficiency.

6.3 Buses, Crossbar, and Multistage Switches

Dynamic connections are implemented by placing electronic switches, routers, concentrators, distributors, arbitrators at the crosspoint of multiple communication paths. Three families of dynamic connections are *buses*, *crossbar switches*, and *multistage switches*. As seen from Fig. 6.1, these interconnects currently yield a wide range of data transfer rates (bandwidth) between 200 Mbps and 100 Gbps.

6.3.1 Multiprocessor Buses

A *bus* is essentially a collection of wires and connectors for data transactions among processors, memory modules, and peripheral devices. A *system bus* is used for data transfer between a *master* device such as a processor and a *slave device* such as a memory board. The bus arbitration logic grants the bus access one request at a time, thus the name *contention bus*.

Many bus standards have been established, such as the PCI, VME, Multibus, SBus, MicroChannel, and IEEE Futurebus. Most standard buses offer the lowest cost in building uniprocessor systems. We will focus on *multiprocessor buses* and *hierarchical buses* for building large SMP, NUMA, or DSM machines. These scalable buses are often equipped with hardware support for cache coherence, fast microprocessor synchronization, and interrupt handling in split transactions, etc.

In Fig. 6.13, we show the structure of a typical multiprocessor bus system. The system bus is often implemented on the backplane or centerplane. Each processor (P) or each *I/O processor* (IOP) can be a master generating a request to access a particular slave device (memory or disk drive, etc.). The system bus consists of data path, address lines and control lines, providing a common medium for communication among all plug-in functional boards. Specialized *interface logic* (IF) and special functional *controllers* (C) are used in different plug-in boards.

Local buses are found on CPU, I/O, or network interface boards. The local bus on a memory board is particularly called a *memory bus*. A typical *I/O bus* can be the SCSI or other I/O channels connecting local disks, printers, and other peripheral devices attached to host machines.

Important issues of designing a multiprocessor bus system include bus arbitration, interrupts handling, protocol conversion, fast synchronization, cache coherence protocol,

split transactions, bus bridging, hierarchical bus extensions, etc. Often, snoopy protocols are built in a multiprocessor bus as discussed in Section 5.2.2.

Hardwired barrier synchronization lines can be added to a multiprocessor cluster bus as proposed in [552]. These hardwired lines can reduce the software-based synchronization time from several thousands of microseconds to a few hundreds of nanoseconds.

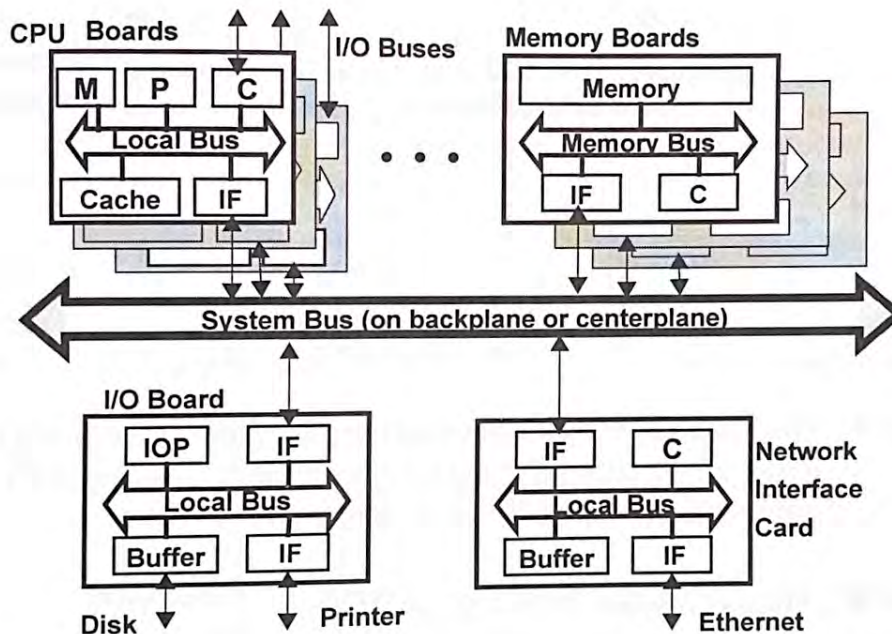


Figure 6.13 A multiprocessor bus consisting of a system bus, memory buses, and local buses on various functional boards.

We consider below two case studies of system buses developed in commercial SMP servers: the Sun Microsystems *Gigaplane bus* and the SGI *POWERpath-2 bus*.

Example 6.1 Sun Microsystems 2.5 Gbps Gigaplane bus interconnect

As illustrated in Fig. 6.14, Sun's Ultra Enterprise server (SunFire) uses a new system bus, called Gigaplane, to interconnect CPU/memory boards and SBus I/O boards, yielding a maximum bandwidth of 2.5 GB/s, 6 times faster than Sun's XDBus used in many servers. The Gigaplane bus data-path has a 256-b packet switching design with 42-b address lines to address 2 TB of shared memory.

The bus clock rate is 83.4 MHz. With a cache line of 64 B transferred per 2 bus cycles, the total burst data transfer rate is 2.6 GB/s. Instead of connecting processors, memory, and I/O buses directly to the Gigaplane, each board uses a UPA (*ultra port architecture*) interconnect switch to fan out the Gigaplane to individual units.

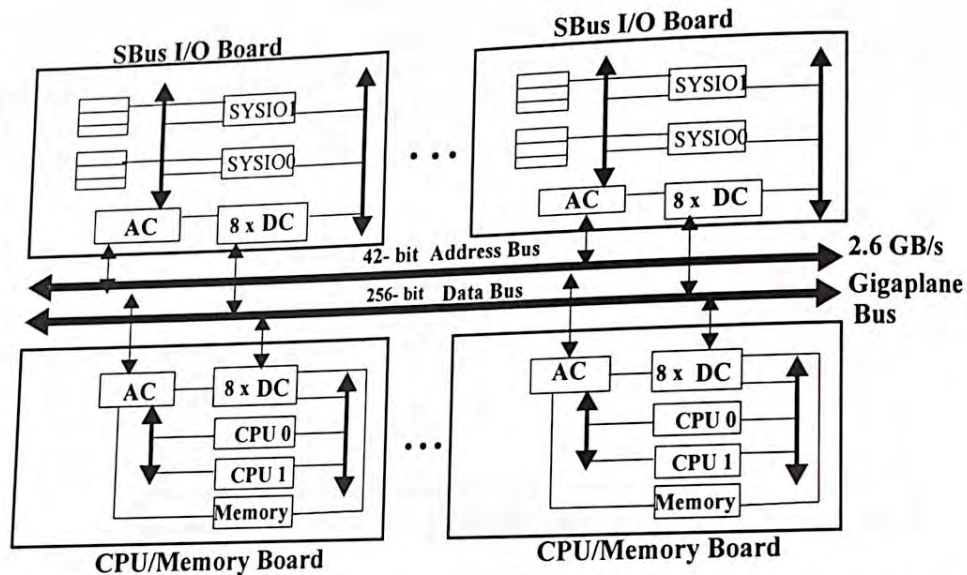


Figure 6.14 The Gigaplane bus in Sun Enterprise x000 servers. (AD: Address Controller, DC: Data Controller, SYSIO: SBus-to-UPA interconnect, Courtesy Sun Microsystems [600], 1997)

Example 6.2 Silicon Graphics POWERpath-2™ Bus

This is a 256-b synchronous bus with a maximum memory bandwidth of 1.2 GB/s. Up to 36 MIPS® R10000® processors, 16 GB of memory, graphics, and six I/O buses can be connected to the POWERpath-2 bus shown in Fig. 6.15.

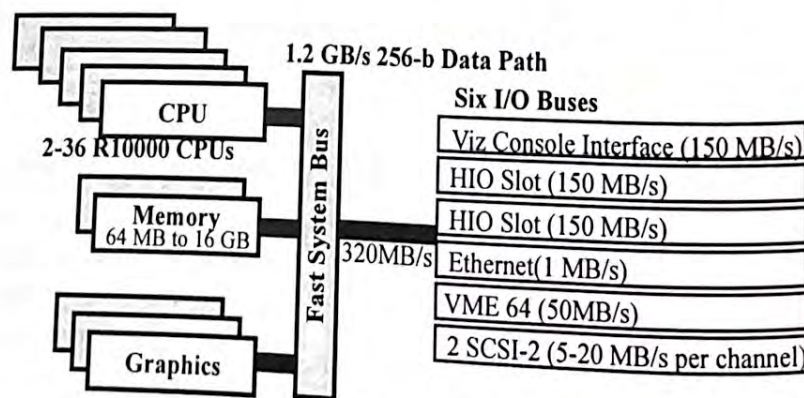


Figure 6.15 The POWERpath-2™ bus architecture in Silicon Graphics servers. (Courtesy Silicon Graphics Computer Systems, 1996)

The system bus uses a split transaction protocol. Four R10000s are mounted on each CPU board. With 320 MB/s per I/O board, up to 500-MB/s sustained data transfer rate can be yielded by the I/O subsystem. High-performance graphics support visualization in supercomputing applications.

Many proprietary multiprocessor buses were also built in IBM mainframes and Digital and Hewlett-Packard multiprocessor servers. IEEE has also developed VME bus (IEEE Standard 1014-1987), Multibus II (IEEE Standard 1296-1987), and Futurebus (IEEE Standard 896.1-1991) for multiprocessors. Readers may refer to published bus standards for specifications of electrical, mechanical, and functional protocols.

Hierarchical Buses The single SMP bus has limited scalability to accommodate large-scale systems. A hierarchical bus structure can alleviate this problem to some extent. We show in Fig. 6.16 a conceptual bus hierarchy designed for interconnecting multiple multiprocessor clusters to build a CC-NUMA machine.

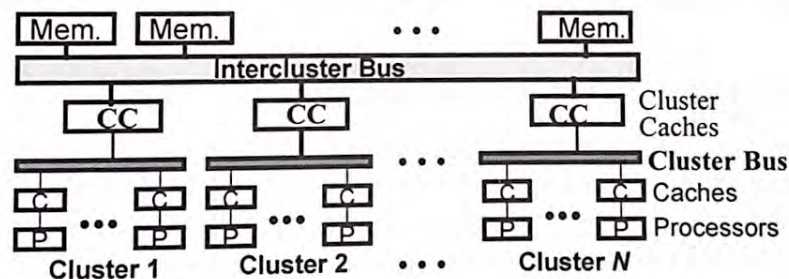


Figure 6.16 A hierarchy of buses for connecting multiprocessor clusters to build a CC-NUMA machine (Courtesy A.W. Wilson, *Proc. Int'l Symp. Computer Architecture*, [644] 1987)

All processors belonging to the same cluster (such as a SMP server) are tied to a common *cluster bus*. A *cluster cache* (CC) can be used as a second-level cache to be shared by all processors in the same cluster. Multiprocessor clusters communicate with each other through an *intercluster bus* connecting all globally shared-memory modules.

Multiple levels of buses must be equipped with a bridging mechanism to maintain cache coherence among all private and shared caches and to interface among the clusters. The IEEE Futurebus [344] has developed special bridges, cache and memory agents, message interfaces, and cable segments for building hierarchical bus systems.

Shortcomings in Bus Interconnects Bus interconnects are for time-shared use by many processors. Even when the bus bandwidth is high, *per-processor bandwidth* is only a

fraction of the total bus bandwidth. Furthermore, bus is prone to failure for lack of redundancy. Bus also has limited scalability. These shortcomings are primarily constrained by packaging technology and the cost involved.

Buses are often confined within a small rack. Clock skewing and global clocking are difficult to overcome in a hierarchical bus extension beyond several racks. Upgrading bus architecture with crossbar switches or multistage networks will overcome these shortcomings to some extent, as studied in the next two subsections.

6.3.2 Crossbar Switches

Much higher bandwidth can be provided by a crossbar switched network for the same datapath width per port and equal number of connecting ports. A crossbar is a single-stage switched network. Like a telephone switchboard, the crosspoint switches provide dynamic connections between all (source, destination) pairs. The cross-point switch can be set on or off dynamically upon program control.

Two ways of using crossbar switches are illustrated below: one for interprocessor communication in symmetric multiprocessors or in multicomputer clusters; and another for *inter-processor-memory* accesses in SMP servers or in vector supercomputers.

Example 6.3 Digital's GIGAswitch/FDDI Crossbar Switch

In Fig. 6.17, we show the crossbar design in Digital's GIGAswitch/FDDI. This crossbar was designed as a cluster interconnect among multiple FDDI rings of Alpha workstations and servers. Using a *FDDI full-duplex technology* (FFDT), it can connect up to 22 FDDI ports in 100 Mbps each way.

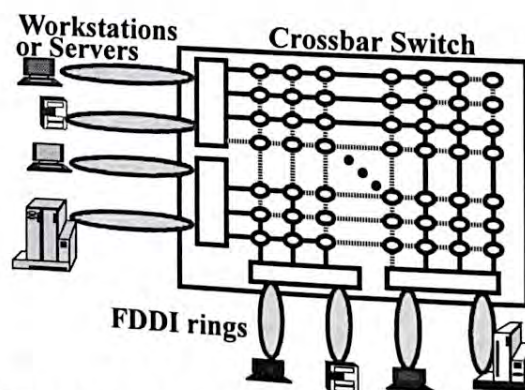


Figure 6.17 GIGAswitch/FDDI: A crossbar switch built for Alpha workstation/server farms (Courtesy Digital Equipment Co., [194], Sept. 1994).

With cut-through routing, latency is reduced to less than 20 μ s. Two or four port line cards are used to yield the 22-port crossbar switch. As a whole, the GIGAswitch/FDDI delivers up to 3.6 Gbps of bandwidth, 360 times that of an Ethernet. The GIGAswitch/FDDI has been used in Digital's TruCluster.

The GIGAswitch/FDDI is general networking product that can be used in any FDDI ring-connected system. This is a good example of an internode communication system based on a crossbar design. ■

In general, the crossbar complexity increases directly with N^2 , where N is the number of ports. Limited by cost, one cannot expect to build very large-scale crossbar switches in the future. The interprocessor crossbar provides permutation connections among N processors. Recently, Sun Microsystems upgraded the Gigaplane bus to a *Gigaplane-XB interconnect* in the Ultra Enterprise 10000 (StarFire) SMP server.

This is a crossbar interconnect using a packet switched scheme with separate address and data paths. Up to 64 processors are interconnected by four snoopy address buses and one 16×16 data crossbar switch. The crossbar is dedicated for point-to-point data communication among processors, while address distribution is handled by broadcast routers on the snoopy buses. We shall further study the SunFire system in Chapter 8.

Crossbar Chip Design Crossbar networks are built with crossbar switch chips. These crossbar chips can be also used to build multistage networks. We study below a crossbar chip design in IBM SP2 multicomputer system.

Example 6.4 IBM Vulcan crossbar chip design

The IBM Vulcan crossbar switch chip design is shown in Fig. 6.18. This switch has been used as the building block of the multistage network in the IBMSp2. Each chip has eight input ports and eight output ports.

The buffered wormhole routing is used. An 8×8 crossbar enables eight packet cells (called *flits*) to pass through the switch in every 40-MHz cycle if there is no conflict. When there is a *hot-spot* of memory access conflict, only one crosspoint switch will be enabled at a time. The blocked flits are buffered in a central queue. This buffering frees up the input ports to receive subsequent flits from the previous switch stage.

The central queue is implemented by a dual-port RAM that can perform one read and one write in a clock cycle. To match the maximum bandwidth, each input port first deserializes eight flits from its FIFO into a chunk and writes the entire 64-b chunk to the central queue in one cycle.

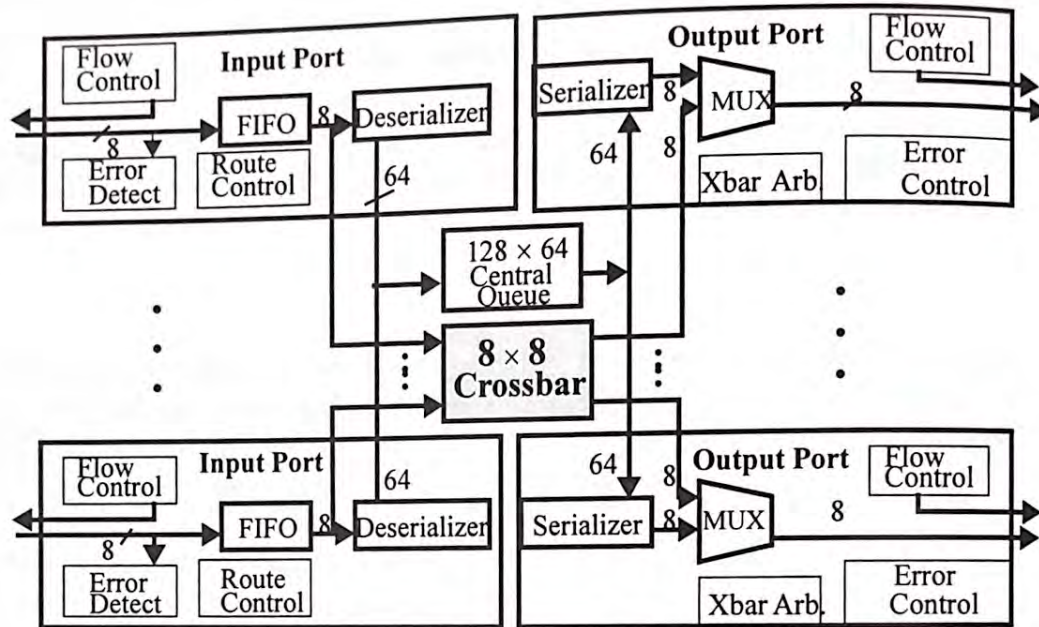


Figure 6.18 An 8×8 crossbar (Vulcan) chip design for SP2 HPS
(Copyright 1995, International Business Machines Corp. Reprinted with permission from C.B. Stunkel et al. *IBM Systems Journal*, Vol. 34, No.2, [592], 1995)

Processor-Memory Crossbar Switch A bus-connected multiprocessor is limited by its bus bandwidth. Very often, the bus becomes the bottleneck in accessing the shared memory by a large number of processors. A better approach is to replace the interprocessor memory bus by a crossbar switch as illustrated in Fig. 6.19. This opens up the possibility of parallel memory accesses of multiple memory banks.

This is essentially a memory access network. The major benefit comes from a sharp increase of memory bandwidth or data transfer rate between processors and shared-memory modules. Each memory module can be accessed by only one processor at a time. When multiple requests collide at a module, the crossbar must resolve the conflicts.

The behavior of each crosspoint switch is very similar to that of a contention bus. However, the processor can generate a sequence of addresses to access different memory modules in a pipelined fashion or simultaneously. In traditional SMP servers, almost all IBM and Sun Microsystems among other SMP suppliers have pushed for crossbar switched interconnects in their scalable SMP servers.

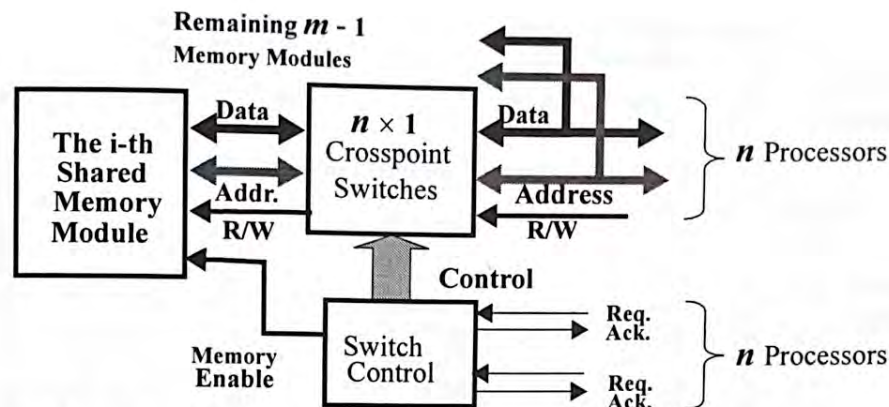


Figure 6.19 Partial circuitry of a $n \times m$ crossbar switch between n processors and m memory modules.

6.3.3 Multistage Interconnection Networks

To build a larger switched network, the single-stage crossbar must be extended to multiple stages, often called a *multistage interconnection network* (MIN). MINs have been built in both MIMD and SIMD computers. Multiple switch modules are used in each stage. Fixed interstage connections are used between switches in adjacent stages. The switch connections can be dynamically set on or off to establish the desired connection patterns between inputs and outputs.

Switch Modules An $n \times n$ switch module has n inputs and n outputs. A binary switch corresponds to a 2×2 switch module as shown in Fig. 6.20(a). Each input can be connected to any of the output ports. But only one-to-one or one-to-many mappings are allowed; no many-to-one mappings are allowed due to output conflicts.

Omega Network In general, an $n \times n$ crossbar switch can achieve $n!$ permutation connections. Different classes of MINs differ in the switch modules and *interstage connection* (ISC) patterns used. The ISC patterns often used include *perfect shuffle*, *butterfly*, *multiway shuffle*, *crossbar*, *cube connection*, etc.

In Fig. 6.20(a), we show four possible connections of a 2×2 switch used in constructing the Omega network. A 8×8 Omega network is shown in Fig. 6.20(b) using three stages of 2×2 switches.

An n -input Omega network requires $\log_2 n$ stages of 2×2 switches. Omega network has been implemented in the Cedar multiprocessors system at the University of Illinois [382]. Next, we show two MIN design examples in Cray Y/MP, a vector multiprocessor, and in IBM SP2 multicomputer.

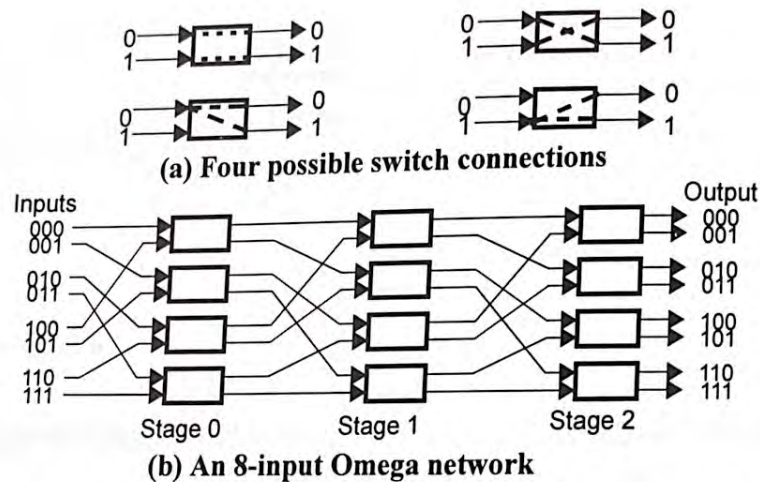


Figure 6.20 Use of 2×2 switches and perfect shuffle to construct a 8×8 Omega network (Courtesy D.H. Lawrie, *IEEE Trans. Computers*, [395] 1975).

Example 6.5 IBM high performance switch (HPS)

All the switch hardware in the node and switch frames forms the *high-performance switch* (HPS). Each *frame* consists of 16 processing nodes (N0 to N15) connected by a 16-way *switchboard*. Eight frames are interconnected by an extra stage of switchboards. Each thin line represents an 8-b bidirectional link. Each thick line represents four 8-b links. There are two stages of switch chips on each switch board. In total, this MIN has four switch stages.

The HPS is a packet switching, multistage Omega network with buffered wormhole routing, driven by a 40-MHz clock. An example HPS connecting 128 nodes is shown in Fig. 6.21. Each switchboard has two stages of switch chips. The left links of a switchboard are used to connect nodes within a frame. The right links provide interframe connections. Each link is 8-b, bidirectional.

Messages are divided into packets, which are in turn divided into 8-b *flow control digits* (flits). When there is no contention, it takes only 5 clocks, or 125 ns for a flit to travel through one stage (i.e., one switch chip) of the HPS. Thus the contention-free hardware latency of the HPS is small, only 875 ns or less for up to 512 nodes.

The actual latency seen by application processes is much higher: It takes at least 40 μ s for a process to send an empty message to another process. A large portion of this message-passing latency is due to software overhead. Between a pair of nodes, the HPS provides a per-port bandwidth of 40 MB/s.

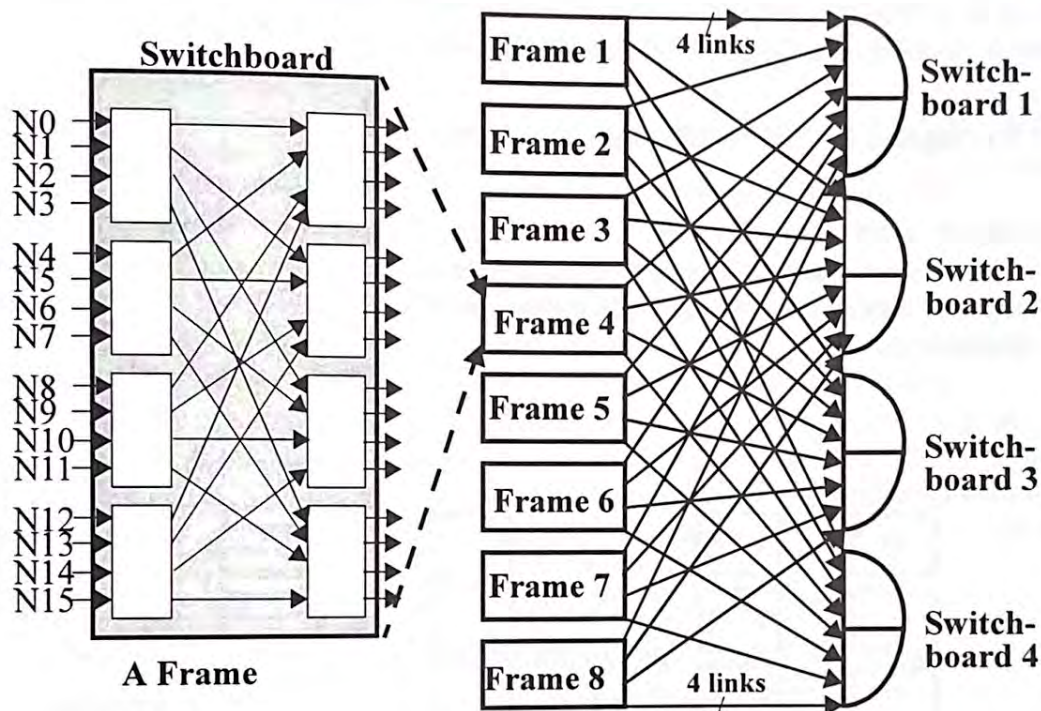


Figure 6.21 A 128-way high-performance switch built with four stages of 16-way switches in the IBM SP2 (Copyright 1995, International Business Machines Corp. Reprinted with permission from C. B. Stunkel et al. *IBM Systems Journal*, Vol. 34, No.2, [592], 1995)

Two important problems need to be solved in designing any high-performance cluster systems. It is desirable for the clocks in all nodes to be synchronized. For instance, when two nodes call the Unix function *gettimeofday()* at the same time, the two calls should return the same value. Of course, perfect synchronization is impossible. How to align the clocks as closely as possible (e.g., within a small drift range of a few microseconds) is the *clock synchronization* problem.

The second problem is the *long-wire* problem. The nodes of a cluster cannot be packed as closely as those of a traditional MPP, because a cluster node is more complex than an MPP node. It is not uncommon that internode communication wires are more than one m long. Long wire length can adversely affect both latency and bandwidth.

The SP2 solves the clock synchronization problem by using a synchronous communication network. The entire HPS is synchronized by a clock from a common 40-MHz oscillation network.

lator. The synchronous network enables each node to maintain a local time-of-day clock that is in synchrony with the clocks in other nodes. IBM developed a software called *Worm* to tune the node clocks to be synchronized within a few microseconds.

Example 6.6 Cray Y/MP Multistage Network

This network is built with 4×4 and 8×8 crossbar switches and 1×8 demultiplexors in three stages shown in Fig. 6.22.

The network is designed to support data streaming between 8 vector processors and 256 memory banks. The network can avoid memory-access conflicts by the eight processors.

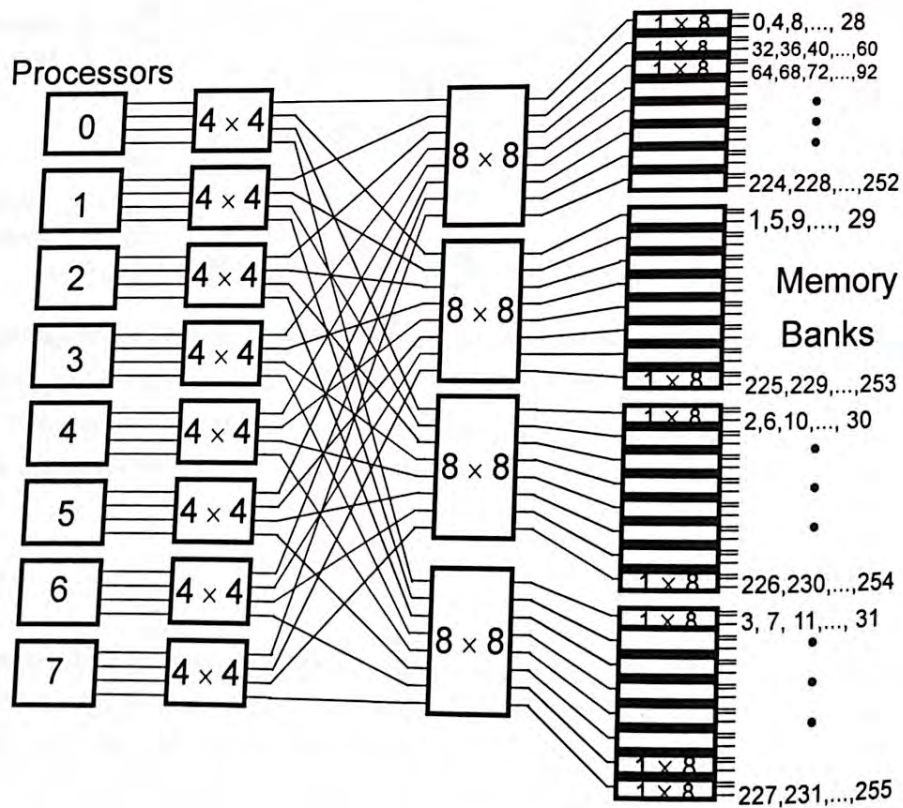


Figure 6.22 The multistage crossbar network for inter-processor-memory connections in Cray Y-MP/816 (Courtesy Cray Research, Inc. [174], 1992)

6.3.4 Comparison of Switched Interconnects

We compare below hardware requirements and performance potentials of system buses, multistage networks, and crossbar switches for building dynamic system interconnects in scalable platforms or cluster systems. Table 6.2 summarizes the wiring/switching complexity, per-processor bandwidth, and aggregate bandwidth of switched interconnects.

Hardware Complexity A bus interconnect costs the lowest among the three with limited wiring and switching complexities. The wiring complexity is primarily determined by the data path width and by the address width of the bus design. 256 data lines and 42 address lines represent state of the art in today's buses. The address lines can be hidden as part of the data path, such as 64 shared address/data lines in the 256-b Futurebus.

The bus switching complexity is determined by the number n of connection taps on the bus, which is also limited by a small number of processors, memory and I/O boards attached to a system bus. Let w be the data path width on the bus. The hardware complexity of a bus interconnect increases linearly with respect to both n and w , as shown by $O(n + w)$ in Table 6.2.

Table 6.2 Complexity and Bandwidth of Switched Interconnects

Interconnect Characteristics	System Bus	Multistage Network	Crossbar Switch
Hardware complexity	$O(n + w)$	$O(nw \log_k n)$	$O(n^2 w)$
Per-processor bandwidth	$O(wf/n)$ to $O(wf)$	$O(wf)$	$O(wf)$
Reported aggregate bandwidth	2.67 GB/s for Gigaplane bus in SunFire server	10.24 GB/s for 512-node HPS in IBM SP2	3.4 Gb/s for Digital's GIGAswitch

The crossbar switch is the most expensive one to build, because its hardware complexity increases with respect to the product $n^2 w$, where n^2 corresponds to the number of cross-points in the crossbar switch and w is the channel width in the crosspoint switch design. For the same data path width, an $n \times n$ crossbar switch is almost n^2 more expensive than that of a bus interconnect.

An n -input multistage network has a hardware complexity of $O[(n \log_k n)w]$, where $n \log_k n$ crosspoints to the number of switches used, assuming $k \times k$ switches are used as building blocks. Here, w is the link width in the MIN design. This complexity lies between the two extremes of buses and crossbar switches. One can roughly estimate that MIN is $n/\log n$ less expensive than that of a crossbar switch for the same channel width.

Per-Processor Bandwidth A bus is time-shared by n processors in an SMP server. Therefore, n processors compete for the bus bandwidth. Assuming the same clock rate f , each unit data transfer takes only 1 cycle on all three interconnects. The *per-processor bandwidth* of the bus varies within the range $O(wf/n)$ and $O(wf)$. Both MIN and crossbar have a wider per-processor bandwidth varying linearly with respect to the function $O(wf)$. This clearly indicates the bottleneck effect in the bus structure.

Even with the same clock rate f , the bus takes less time (usually 1 or 2 cycles) to transfer a unit piece of data, compared with many cycles needed to transfer a piece of data through the multiple switch stages in a MIN. Therefore, the actual per bus bandwidth may not be much lower than that of a MIN. In all cases, the crossbar has the highest per-processor bandwidth, because of a short latency (1 or 2 cycles as well) and conflict-free connections between the input and output ports.

Aggregate Bandwidth In the last row of Table 6.2, we show the aggregate bandwidths of three representative bus, MIN, and crossbar interconnects available from the commodity market. In this case, the Gigaplane bus has an aggregate bandwidth of 2.67 GB/s = 21.36 Gbps. Assuming $n = 24$ processors sharing the bandwidth, the per-processor bandwidth can be as low as $21.36/24 = 0.89$ Gbps.

On the other hand, the GIGAswitch crossbar has an aggregate bandwidth of 3.4 Gbps. This means that the per-processor bandwidth of GIGAswitch is potentially 3.8 times higher than that of the Gigaplane bus. Among the three, the IBM HPS has a much larger configuration of $n = 512$ ports, resulting in an aggregate bandwidth of 10.24 Gbps = 8192 Gbps. With this reasoning, we conclude that the MIN is more scalable than either the bus or the crossbar interconnect.

Switch Selection Among commercial systems, there is no doubt that the bus is still the most cost-effective system interconnect for an SMP system with less than 40 processors. The crossbar interconnect should be chosen if performance outweighs price tag, or when the system is rather small, say, having less than 16 processors. This is evidenced by the fact that Sun's StarFire SMP server is built with an interconnect having four address buses and a 16×16 crossbar switch among 64 processors.

The major advantage of MINs lies in their scalability with modular construction. However, the latency increases with $\log n$, the number of stages in the network. Also, costs due to increased wiring and switching complexity are another constraint in building large MINs in commodity networks. For this reason, the commercial SP2 system goes only to a maximum size of 128 nodes.

To build future MPP or clustered systems, point-to-point topologies are more flexible and even more scalable than regularly structured networks. With advances in optical interconnects and digital signal processing microelectronics, large-scale MINs or crossbar networks may become economically feasible in larger multiprocessors or multicomputers. Based on 1997 technology, 64×64 crossbar and 512-input MIN have been built in commercial systems.