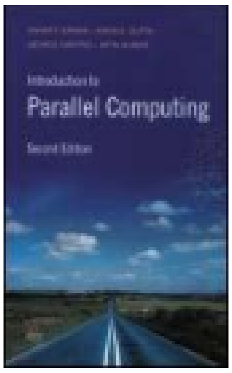
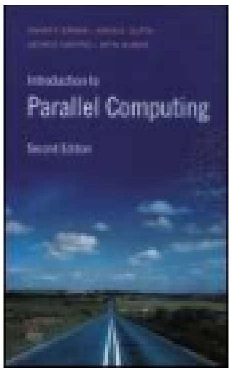
[Team LiB]

Pub Date: January 16, 2003

ISBN: 0-201-64865-2
ISBN: 0-201-64865-2

Pages: 856
Pages: 856

NEXT NEXT

• Table of Contents
• Table of Contents

Introduction to Parallel Computing, Second Edition

Introduction to Parallel Computing, Second Edition

By Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar
By Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar

Start Reading Start Reading

Publisher: Addison Wesley
Publisher: Addison Wesley

Pub Date: January 16, 2003

Increasingly, parallel processing is being seen as the only cost-effective method for the fast solution of computationally large and data-intensive problems. The emergence of inexpensive parallel computers such as commodity desktop multiprocessors and clusters of workstations or PCs has made such parallel methods generally applicable, as have software standards for portable parallel programming. This sets the stage for substantial growth in parallel software.

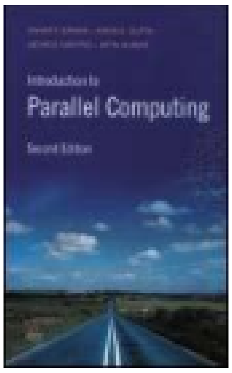
Increasingly, parallel processing is being seen as the only cost-effective method for the fast solution of computationally large and data-intensive problems. The emergence of inexpensive parallel computers such as commodity desktop multiprocessors and clusters of workstations or PCs has made such parallel methods generally applicable, as have software standards for portable parallel programming. This sets the stage for substantial growth in parallel software.

Data-intensive applications such as transaction processing and information retrieval, data mining and analysis and multimedia services have provided a new challenge for the modern generation of parallel platforms. Emerging areas such as computational biology and nanotechnology have implications for algorithms and systems development, while changes in architectures, programming models and applications have implications for how parallel platforms are made available to users in the form of grid-based services.

Data-intensive applications such as transaction processing and information retrieval, data mining and analysis and multimedia services have provided a new challenge for the modern generation of parallel platforms. Emerging areas such as computational biology and nanotechnology have implications for algorithms and systems development, while changes in architectures, programming models and applications have implications for how parallel platforms are made available to users in the form of grid-based services. This book takes into

account these new developments as well as covering the more traditional

This book takes into account these new developments as well as covering the more traditional problems addressed by parallel computers.Where possible it employs an architecture
problems addressed by parallel computers.Where possible it employs an architecture
independent view of the underlying platforms and designs algorithms for an abstract model.
independent view of the underlying platforms and designs algorithms for an abstract model.
Message Passing Interface (MPI), POSIX threads and OpenMP have been selected as
Message Passing Interface (MPI), POSIX threads and OpenMP have been selected as
programming models and the evolving application mix of parallel computing is reflected in
programming models and the evolving application mix of parallel computing is reflected in
various examples throughout the book.
various examples throughout the book.



• Table of Contents

Introduction to Parallel Computing, Second Edition By Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar

Publisher: Addison Wesley
Pub Date: January 16, 2003
ISBN: 0-201-64865-2
Pages: 856

Copyright
Pearson Education
Preface
Acknowledgments
Chapter 1. Introduction to Parallel Computing Section 1.1. Motivating Parallelism
Section 1.2. Scope of Parallel Computing Section 1.3. Organization and Contents of the Text Section 1.4. Bibliographic Remarks
Problems

Chapter 2. Parallel Programming Platforms
Section 2.1. Implicit Parallelism: Trends in Microprocessor Architectures* Section 2.2. Limitations of Memory System Performance* Section 2.3. Dichotomy of Parallel Computing Platforms Section 2.4. Physical Organization of Parallel Platforms Section 2.5. Communication Costs in Parallel Machines Section 2.6. Routing Mechanisms for Interconnection Networks Section 2.7. Impact of Process-Processor Mapping and Mapping Techniques Section 2.8. Bibliographic Remarks
Problems

Chapter 3. Principles of Parallel Algorithm Design
Section 3.1. Preliminaries
Section 3.2. Decomposition Techniques

Section 3.3. Characteristics of Tasks and Interactions
Section 3.4. Mapping Techniques for Load Balancing
Section 3.5. Methods for Containing Interaction Overheads Section 3.6. Parallel Algorithm Models
Section 3.7. Bibliographic Remarks
Problems

Chapter 4. Basic Communication Operations
Section 4.1. One-to-All Broadcast and All-to-One Reduction Section 4.2. All-to-All Broadcast and Reduction
Section 4.3. All-Reduce and Prefix-Sum Operations
Section 4.4. Scatter and Gather
Section 4.5. All-to-All Personalized Communication
Section 4.6. Circular Shift
Section 4.7. Improving the Speed of Some Communication Operations Section 4.8. Summary
Section 4.9. Bibliographic Remarks
Problems

Chapter 5. Analytical Modeling of Parallel Programs
Section 5.1. Sources of Overhead in Parallel Programs
Section 5.2. Performance Metrics for Parallel Systems
Section 5.3. The Effect of Granularity on Performance
Section 5.4. Scalability of Parallel Systems
Section 5.5. Minimum Execution Time and Minimum Cost-Optimal Execution Time Section 5.6. Asymptotic Analysis of Parallel Programs
Section 5.7. Other Scalability Metrics
Section 5.8. Bibliographic Remarks
Problems

Chapter 6. Programming Using the Message-Passing Paradigm
Section 6.1. Principles of Message-Passing Programming
Section 6.2. The Building Blocks: Send and Receive Operations Section 6.3. MPI: the Message Passing Interface
Section 6.4. Topologies and Embedding
Section 6.5. Overlapping Communication with Computation Section 6.6. Collective Communication and Computation Operations Section 6.7. Groups and Communicators
Section 6.8. Bibliographic Remarks
Problems

Chapter 7. Programming Shared Address Space Platforms
Section 7.1. Thread Basics
Section 7.2. Why Threads?
Section 7.3. The POSIX Thread API
Section 7.4. Thread Basics: Creation and Termination
Section 7.5. Synchronization Primitives in Pthreads
Section 7.6. Controlling Thread and Synchronization Attributes Section 7.7. Thread Cancellation
Section 7.8. Composite Synchronization Constructs
Section 7.9. Tips for Designing Asynchronous Programs
Section 7.10. OpenMP: a Standard for Directive Based Parallel Programming Section 7.11. Bibliographic Remarks
Problems

Chapter 8. Dense Matrix Algorithms
Section 8.1. Matrix-Vector Multiplication
Section 8.2. Matrix-Matrix Multiplication

Section 8.3. Solving a System of Linear Equations
Section 8.4. Bibliographic Remarks
Problems

Chapter 9. Sorting
Section 9.1. Issues in Sorting on Parallel Computers
Section 9.2. Sorting Networks
Section 9.3. Bubble Sort and its Variants
Section 9.4. Quicksort
Section 9.5. Bucket and Sample Sort
Section 9.6. Other Sorting Algorithms
Section 9.7. Bibliographic Remarks
Problems

Chapter 10. Graph Algorithms
Section 10.1. Definitions and Representation
Section 10.2. Minimum Spanning Tree: Prim's Algorithm Section 10.3.
Single-Source Shortest Paths: Dijkstra's Algorithm Section 10.4. All-Pairs
Shortest Paths
Section 10.5. Transitive Closure
Section 10.6. Connected Components
Section 10.7. Algorithms for Sparse Graphs
Section 10.8. Bibliographic Remarks
Problems

Chapter 11. Search Algorithms for Discrete Optimization Problems
Section 11.1. Definitions and Examples
Section 11.2. Sequential Search Algorithms
Section 11.3. Search Overhead Factor
Section 11.4. Parallel Depth-First Search
Section 11.5. Parallel Best-First Search
Section 11.6. Speedup Anomalies in Parallel Search Algorithms Section
11.7. Bibliographic Remarks
Problems

Chapter 12. Dynamic Programming
Section 12.1. Overview of Dynamic Programming
Section 12.2. Serial Monadic DP Formulations
Section 12.3. Nonserial Monadic DP Formulations
Section 12.4. Serial Polyadic DP Formulations
Section 12.5. Nonserial Polyadic DP Formulations
Section 12.6. Summary and Discussion
Section 12.7. Bibliographic Remarks
Problems

Chapter 13. Fast Fourier Transform
Section 13.1. The Serial Algorithm
Section 13.2. The Binary-Exchange Algorithm Section
13.3. The Transpose Algorithm Section 13.4.
Bibliographic Remarks
Problems

Appendix A. Complexity of Functions and Order Analysis
Section A.1. Complexity of Functions
Section A.2. Order Analysis of Functions

Bibliography

Copyright

Pearson Education Limited

Edinburgh Gate

Harlow

Essex CM20 2JE

England

and Associated Companies throughout the world

Visit us on the World Wide Web at: www.pearsoneduc.com

First published by The Benjamin/Cummings Publishing Company, Inc. 1994

Second edition published 2003

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

Library of Congress Cataloging-in-Publication Data

A catalog record for this book is available from the Library of Congress

10 9 8 7 6 5 4 3 2 1
07 06 05 04 03

Printed and bound in the United States of America

Dedication

To Joanna, Rinku, Krista, and Renu

[Team LiB]

[Team LiB]

Pearson Education

[Team LiB]

[Team LiB]

Preface

Since the 1994 release of the text "Introduction to Parallel Computing: Design and Analysis of Algorithms" by the same authors, the field of parallel computing has undergone significant changes. Whereas tightly coupled scalable message-passing platforms were the norm a decade ago, a significant portion of the current generation of platforms consists of inexpensive clusters of workstations, and multiprocessor workstations and servers. Programming models for these platforms have also evolved over this time. Whereas most machines a decade back relied on custom APIs for messaging and loop-based parallelism, current models standardize these APIs across platforms. Message passing libraries such as PVM and MPI, thread libraries such as POSIX threads, and directive based models such as OpenMP are widely accepted as standards, and have been ported to a variety of platforms.

With respect to applications, fluid dynamics, structural mechanics, and signal processing formed dominant applications a decade back. These applications continue to challenge the current generation of parallel platforms. However, a variety of new applications have also become important. These include data-intensive applications such as transaction processing and information retrieval, data mining and analysis, and multimedia services. Applications in emerging areas of computational biology and nanotechnology pose tremendous challenges for algorithms and systems development. Changes in architectures, programming models, and applications are also being accompanied by changes in how parallel platforms are made available to the users in the form of grid-based services.

This evolution has a profound impact on the process of design, analysis, and implementation of parallel algorithms. Whereas the emphasis of parallel algorithm design a decade back was on precise mapping of tasks to specific topologies such as meshes and hypercubes, current emphasis is on programmability and portability, both from points of view of algorithm design and implementation. To this effect, where possible, this book employs an architecture independent view of the underlying platforms and designs algorithms for an abstract model. With respect to programming models, Message Passing Interface (MPI), POSIX threads, and OpenMP have been selected. The evolving application mix for parallel computing is also reflected in various examples in the book.

This book forms the basis for a single concentrated course on parallel computing or a two-part sequence. Some suggestions for such a two-part sequence are:

1. Introduction to Parallel Computing: Chapters 1–6. This course would provide the basics of algorithm design and parallel programming.

2. Design and Analysis of Parallel Algorithms: Chapters 2 and 3 followed by Chapters 8–12. This course would provide an in-depth coverage of design and analysis of various parallel algorithms.

The material in this book has been tested in Parallel Algorithms and Parallel Computing courses at the University of Minnesota and Purdue University. These courses are taken primarily by graduate students and senior-level undergraduate students in Computer Science. In addition, related courses in Scientific Computation, for which this material has also been tested, are taken by graduate students in science and engineering, who are interested in solving computationally intensive problems.
Most chapters of the book include (i) examples and illustrations; (ii) problems that supplement the text and test students' understanding of the material; and (iii) bibliographic remarks to aid researchers and students interested in learning more about related and advanced topics. The comprehensive subject index helps the reader locate terms they might be interested in. The page number on which a term is defined is highlighted in boldface in the index. Furthermore, the term itself appears in bold italics where it is defined. The sections that deal with relatively complex material are preceded by a '*'. An instructors' manual containing slides of the figures and solutions to selected problems is also available from the publisher (http://www.booksites.net/kumar).

As with our previous book, we view this book as a continually evolving resource. We thank all the readers who have kindly shared critiques, opinions, problems, code, and other information relating to our first book. It is our sincere hope that we can continue this interaction centered around this new book. We encourage readers to address communication relating to this book to book-vk@cs.umn.edu. All relevant reader input will be added to the information archived at the site http://www.cs.umn.edu/~parbook with due credit to (and permission of) the sender(s). An on-line errata of the book will also be maintained at the site. We believe that in a highly dynamic field such as ours, a lot is to be gained from a healthy exchange of ideas and material in this manner.

[Team LiB]

[Team LiB]

Acknowledgments

We would like to begin by acknowledging our spouses, Joanna, Rinku, Krista, and Renu to whom this book is dedicated. Without their sacrifices this project would not have been seen completion. We also thank our parents, and family members, Akash, Avi, Chethan, Eleni, Larry, Mary-Jo, Naina, Petros, Samir, Subhasish, Varun, Vibhav, and Vipasha for their affectionate support and encouragement throughout this project.

Our respective institutions, Computer Sciences and Computing Research Institute (CRI) at Purdue University, Department of Computer Science & Engineering, the Army High Performance Computing Research Center (AHPCRC), and the Digital Technology Center (DTC) at the University of Minnesota, and the IBM T. J. Watson Research Center at Yorktown Heights, provided computing resources and active and nurturing environments for the completion of this project.

This project evolved from our first book. We would therefore like to acknowledge all of the people who helped us with both editions. Many people contributed to this project in different ways. We would like to thank Ahmed Sameh for his constant encouragement and support, and Dan Challou, Michael Heath, Dinesh Mehta, Tom Nurkkala, Paul Saylor, and Shang-Hua Teng for the valuable input they provided to the various versions of the book. We thank the students of the introduction to parallel computing classes at the University of Minnesota and Purdue university for identifying and working through the errors in the early drafts of the book. In particular, we acknowledge the patience and help of Jim Diehl and Rasit Eskicioglu, who worked through several early drafts of the manuscript to identify numerous errors. Ramesh Agarwal, David Bailey, Rupak Biswas, Jim Bottum, Thomas Downar, Rudolf Eigenmann, Sonia Fahmy, Greg Frederickson, John Gunnels, Fred Gustavson, Susanne Hambrusch, Bruce Hendrickson, Christoph Hoffmann, Kai Hwang, Ioannis Ioannidis, Chandrika Kamath, David Keyes, Mehmet Koyuturk, Piyush Mehrotra, Zhiyuan Li, Jens Palsberg, Voicu Popescu, Alex Pothen, Viktor Prasanna, Sanjay Ranka, Naren Ramakrishnan, Elisha Sacks, Vineet Singh, Sartaj Sahni, Vivek Sarin, Wojciech Szpankowski, Srikanth Thirumalai, Jan Vitek, and David Yau have been great technical resources. It was a pleasure working with the cooperative and helpful staff at Pearson Education. In particular, we would like to thank Keith Mansfield and Mary Lince for their professional handling of the project.

The Army Research Laboratory, ARO, DOE, NASA, and NSF provided parallel computing research support for Ananth Grama, George Karypis, and Vipin Kumar. In particular, Kamal Abdali, Michael Coyle, Jagdish Chandra, Frederica Darema, Stephen Davis, Wm Randolph Franklin, Richard Hirsch, Charles Koelbel, Raju Namburu, N. Radhakrishnan, John Van Rosendale, Subhash Saini, and Xiaodong Zhang have been supportive of our research programs in the area of parallel computing. Andrew Conn, Brenda Dietrich, John Forrest, David Jensen, and Bill Pulleyblank at IBM supported the work of Anshul Gupta over the years.

[Team LiB]
[Team LiB]

Chapter 1. Introduction to Parallel Computing

The past decade has seen tremendous advances in microprocessor technology. Clock rates of processors have increased from about 40 MHz (e.g., a MIPS R3000, circa 1988) to over 2.0 GHz (e.g., a Pentium 4, circa 2002). At the same time, processors are now capable of executing multiple instructions in the same cycle. The average number of cycles per instruction (CPI) of high end processors has improved by roughly an order of magnitude over the past 10 years. All this translates to an increase in the peak floating point operation execution rate (floating point operations per second, or FLOPS) of several orders of magnitude. A variety of other issues have also become important over the same period. Perhaps the most prominent of these is the ability (or lack thereof) of the memory system to feed data to the processor at the required rate. Significant innovations in architecture and software have addressed the alleviation of bottlenecks posed by the datapath and the memory.

The role of concurrency in accelerating computing elements has been recognized for several decades. However, their role in providing multiplicity of datapaths, increased access to storage elements (both memory and disk), scalable performance, and lower costs is reflected in the wide variety of applications of parallel computing. Desktop machines, engineering workstations, and compute servers with two, four, or even eight processors connected together are becoming common platforms for design applications. Large scale applications in science and engineering rely on larger configurations of parallel computers, often comprising hundreds of processors. Data intensive platforms such as database or web servers and applications such as transaction processing and data mining often use clusters of workstations that provide high aggregate disk bandwidth. Applications in graphics and visualization use multiple rendering pipes and processing elements to compute and render realistic environments with millions of polygons in real time. Applications requiring high availability rely on parallel and distributed platforms for redundancy. It is therefore extremely important, from the point of view of cost, performance, and application requirements, to understand the principles, tools, and techniques for programming the wide variety of parallel platforms currently available.

[Team LiB]
[Team LiB]

1.1 Motivating Parallelism

Development of parallel software has traditionally been thought of as time and effort intensive. This can be largely attributed to the inherent complexity of specifying and coordinating concurrent tasks, a lack of portable algorithms, standardized environments, and software development toolkits. When viewed in the context of the brisk rate of development of microprocessors, one is tempted to question the need for devoting significant effort towards exploiting parallelism as a means of accelerating applications. After all, if it takes two years to develop a parallel application, during which time the underlying hardware and/or software platform has become obsolete, the development effort is clearly wasted. However, there are some unmistakable trends in hardware design, which indicate that uniprocessor (or implicitly parallel) architectures may not be able to sustain the rate of *realizable* performance increments in the future. This is a result of lack of implicit parallelism as well as other bottlenecks such as the datapath and the memory. At the same time, standardized hardware interfaces have reduced the turnaround time from the development of a microprocessor to a parallel machine based on the microprocessor. Furthermore, considerable progress has been made in standardization of programming environments to ensure a longer life-cycle for parallel applications. All of these present compelling arguments in favor of parallel computing platforms.

1.1.1 The Computational Power Argument – from Transistors to FLOPS

In 1965, Gordon Moore made the following simple observation:

> *"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000."*

His reasoning was based on an empirical log-linear relationship between device complexity and time, observed over three data points. He used this to justify that by 1975, devices with as many as 65,000 components would become feasible on a single silicon chip occupying an area of only about one-fourth of a square inch. This projection turned out to be accurate with the fabrication of a 16K CCD memory with about 65,000 components in 1975. In a subsequent paper in 1975, Moore attributed the log-linear relationship to exponential behavior of die sizes, finer minimum dimensions, and "circuit and device cleverness". He went on to state that:

> *"There is no room left to squeeze anything out by being clever. Going forward from here we have to depend on the two size factors - bigger dies and finer dimensions."*

He revised his rate of circuit complexity doubling to 18 months and projected from 1975 onwards at this reduced rate. This curve came to be known as "Moore's Law". Formally, Moore's Law states that circuit complexity doubles every eighteen months. This empirical relationship has been amazingly resilient over the years both for microprocessors as well as for DRAMs. By relating component density and increases in die-size to the computing power of a device, Moore's law has been extrapolated to state that the amount of computing power available at a given cost doubles approximately every 18 months.

The limits of Moore's law have been the subject of extensive debate in the past few years. Staying clear of this debate, the issue of translating transistors into useful OPS (operations per second) is the critical one. It is possible to fabricate devices with very large transistor counts. How we use these transistors to achieve increasing rates of computation is the key architectural challenge. A logical recourse to this is to rely on parallelism – both implicit and explicit. We will briefly discuss implicit parallelism in Section 2.1 and devote the rest of this book to exploiting explicit parallelism.

1.1.2 The Memory/Disk Speed Argument

The overall speed of computation is determined not just by the speed of the processor, but also by the ability of the memory system to feed data to it. While clock rates of high-end processors have increased at roughly 40% per year over the past decade, DRAM access times have only improved at the rate of roughly 10% per year over this interval. Coupled with increases in instructions executed per clock cycle, this gap between processor speed and memory presents a tremendous performance bottleneck. This growing mismatch between processor speed and DRAM latency is typically bridged by a hierarchy of successively faster memory devices called caches that rely on locality of data reference to deliver higher memory system performance. In addition to the latency, the net effective bandwidth between DRAM and the processor poses other problems for sustained computation rates.

The overall performance of the memory system is determined by the fraction of the total memory requests that can be satisfied from the cache. Memory system performance is addressed in greater detail in Section 2.2. Parallel platforms typically yield better memory system performance because they provide (i) larger aggregate caches, and (ii) higher aggregate bandwidth to the memory system (both typically linear in the number of processors). Furthermore, the principles that are at the heart of parallel algorithms, namely locality of data reference, also lend themselves to cache-friendly serial algorithms. This argument can be extended to disks where parallel platforms can be used to achieve high aggregate bandwidth to secondary storage. Here, parallel algorithms yield insights into the development of out-of-core computations. Indeed, some of the fastest growing application areas of parallel computing in data servers (database servers, web servers) rely not so much on their high aggregate computation rates but rather on the ability to pump data out at a faster rate.

1.1.3 The Data Communication Argument

As the networking infrastructure evolves, the vision of using the Internet as one large heterogeneous parallel/distributed computing environment has begun to take shape. Many applications lend themselves naturally to such computing paradigms. Some of the most impressive applications of massively parallel computing have been in the context of wide-area distributed platforms. The SETI (Search for Extra Terrestrial Intelligence) project utilizes the power of a large number of home computers to analyze electromagnetic signals from outer space. Other such efforts have attempted to factor extremely large integers and to solve large discrete optimization problems.

In many applications there are constraints on the location of data and/or resources across the Internet. An example of such an application is mining of large commercial datasets distributed over a relatively low bandwidth network. In such applications, even if the computing power is available to accomplish the required task without resorting to parallel computing, it is infeasible to collect the data at a central location. In these cases, the motivation for parallelism comes not just from the need for computing resources but also from the infeasibility or undesirability of alternate (centralized) approaches.

1.2 Scope of Parallel Computing

Parallel computing has made a tremendous impact on a variety of areas ranging from computational simulations for scientific and engineering applications to commercial applications in data mining and transaction processing. The cost benefits of parallelism coupled with the performance requirements of applications present compelling arguments in favor of parallel computing. We present a small sample of the diverse applications of parallel computing.

1.2.1 Applications in Engineering and Design

Parallel computing has traditionally been employed with great success in the design of airfoils (optimizing lift, drag, stability), internal combustion engines (optimizing charge distribution, burn), high-speed circuits (layouts for delays and capacitive and inductive effects), and structures (optimizing structural integrity, design parameters, cost, etc.), among others. More recently, design of microelectromechanical and nanoelectromechanical systems (MEMS and NEMS) has attracted significant attention. While most applications in engineering and design pose problems of multiple spatial and temporal scales and coupled physical phenomena, in the case of MEMS/NEMS design these problems are particularly acute. Here, we often deal with a mix of quantum phenomena, molecular dynamics, and stochastic and continuum models with physical processes such as conduction, convection, radiation, and structural mechanics, all in a single system. This presents formidable challenges for geometric modeling, mathematical modeling, and algorithm development, all in the context of parallel computers.

Other applications in engineering and design focus on optimization of a variety of processes. Parallel computers have been used to solve a variety of discrete and continuous optimization problems. Algorithms such as Simplex, Interior Point Method for linear optimization and Branch-and-bound, and Genetic programming for discrete optimization have been efficiently parallelized and are frequently used.

1.2.2 Scientific Applications

The past few years have seen a revolution in high performance scientific computing applications. The sequencing of the human genome by the International Human Genome Sequencing Consortium and Celera, Inc. has opened exciting new frontiers in bioinformatics. Functional and structural characterization of genes and proteins hold the promise of understanding and fundamentally influencing biological processes. Analyzing biological sequences with a view to developing new drugs and cures for diseases and medical conditions requires innovative algorithms as well as large-scale computational power. Indeed, some of the newest parallel computing technologies are targeted specifically towards applications in bioinformatics.

Advances in computational physics and chemistry have focused on understanding processes ranging in scale from quantum phenomena to macromolecular structures. These have resulted in design of new materials, understanding of chemical pathways, and more efficient processes. Applications in astrophysics have explored the evolution of galaxies, thermonuclear processes, and the analysis of extremely large datasets from telescopes. Weather modeling, mineral prospecting, flood prediction, etc., rely heavily on parallel computers and have very significant impact on day-to-day life.
Bioinformatics and astrophysics also present some of the most challenging problems with

respect to analyzing extremely large datasets. Protein and gene databases (such as PDB, SwissProt, and ENTREZ and NDB) along with Sky Survey datasets (such as the Sloan Digital Sky Surveys) represent some of the largest scientific datasets. Effectively analyzing these datasets requires tremendous computational power and holds the key to significant scientific discoveries.

1.2.3 Commercial Applications

With the widespread use of the web and associated static and dynamic content, there is increasing emphasis on cost-effective servers capable of providing scalable performance. Parallel platforms ranging from multiprocessors to linux clusters are frequently used as web and database servers. For instance, on heavy volume days, large brokerage houses on Wall Street handle hundreds of thousands of simultaneous user sessions and millions of orders. Platforms such as IBMs SP supercomputers and Sun Ultra HPC servers power these business-critical sites. While not highly visible, some of the largest supercomputing networks are housed on Wall Street.

The availability of large-scale transaction data has also sparked considerable interest in data mining and analysis for optimizing business and marketing decisions. The sheer volume and geographically distributed nature of this data require the use of effective parallel algorithms for such problems as association rule mining, clustering, classification, and time-series analysis.

1.2.4 Applications in Computer Systems

As computer systems become more pervasive and computation spreads over the network, parallel processing issues become engrained into a variety of applications. In computer security, intrusion detection is an outstanding challenge. In the case of network intrusion detection, data is collected at distributed sites and must be analyzed rapidly for signaling intrusion. The infeasibility of collecting this data at a central location for analysis requires effective parallel and distributed algorithms. In the area of cryptography, some of the most spectacular applications of Internet-based parallel computing have focused on factoring extremely large integers.

Embedded systems increasingly rely on distributed control algorithms for accomplishing a variety of tasks. A modern automobile consists of tens of processors communicating to perform complex tasks for optimizing handling and performance. In such systems, traditional parallel and distributed algorithms for leader selection, maximal independent set, etc., are frequently used.

While parallel computing has traditionally confined itself to platforms with well behaved compute and network elements in which faults and errors do not play a significant role, there are valuable lessons that extend to computations on ad-hoc, mobile, or faulty environments.

[Team LiB]
[Team LiB]

1.3 Organization and Contents of the Text

This book provides a comprehensive and self-contained exposition of problem solving using parallel computers. Algorithms and metrics focus on practical and portable models of parallel machines. Principles of algorithm design focus on desirable attributes of parallel algorithms and techniques for achieving these in the contest of a large class of applications and architectures. Programming techniques cover standard paradigms such as MPI and POSIX threads that are available across a range of parallel platforms.

Chapters in this book can be grouped into four main parts as illustrated in Figure 1.1. These parts are as follows:

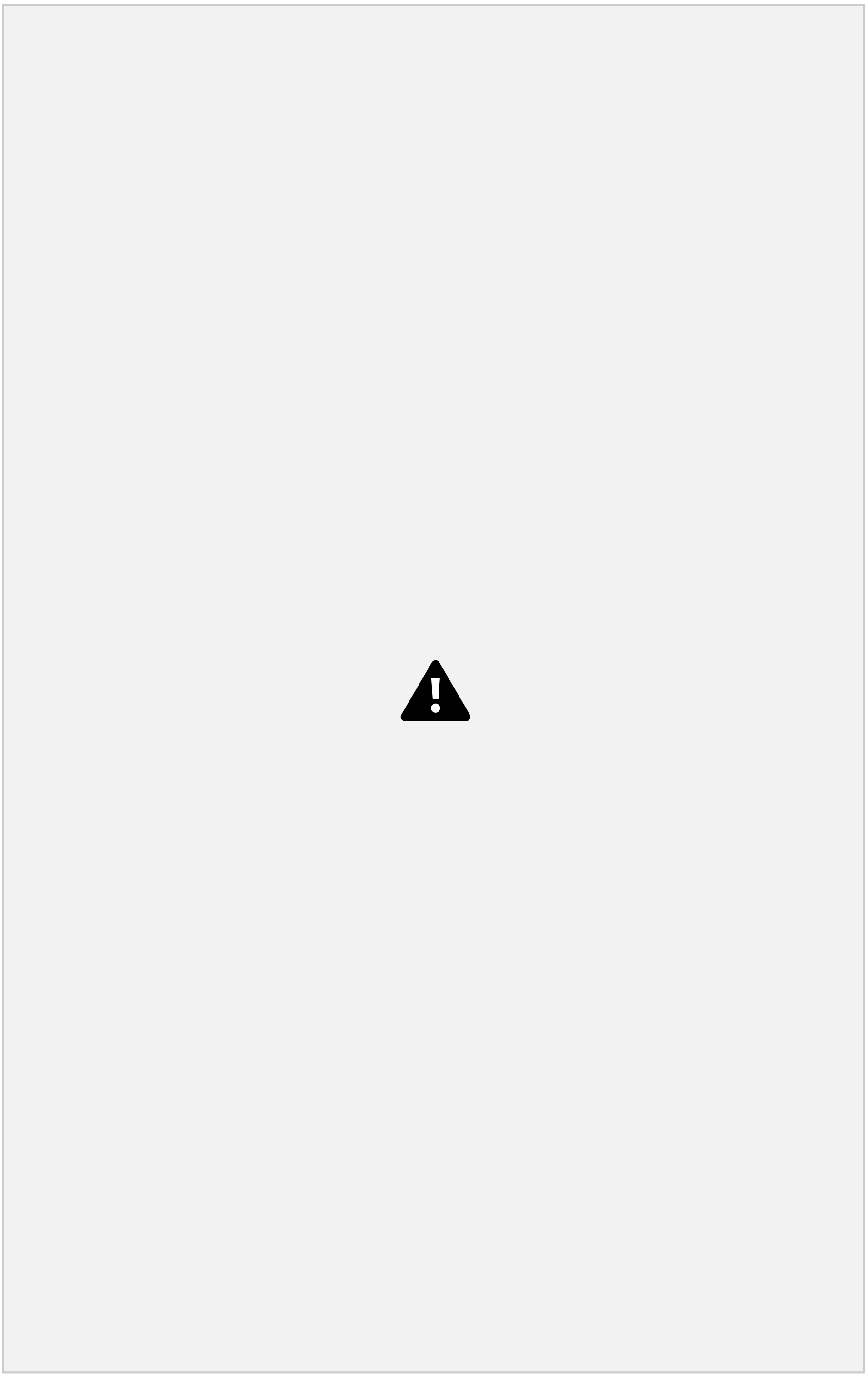


Figure 1.1. Recommended sequence for reading the chapters.

Fundamentals This section spans Chapters 2 through 4 of the book. Chapter 2, Parallel Programming Platforms, discusses the physical organization of parallel platforms. It establishes cost metrics that can be used for algorithm design. The objective of this chapter is not to provide an exhaustive treatment of parallel architectures; rather, it aims to provide sufficient detail required to use these machines efficiently. Chapter 3, Principles of Parallel Algorithm Design, addresses key factors that contribute to efficient parallel algorithms and presents a suite of techniques that can be applied across a wide range of applications. Chapter 4, Basic Communication Operations, presents a core set of operations that are used throughout the book for facilitating efficient data transfer in parallel algorithms. Finally, Chapter 5, Analytical Modeling of Parallel Programs, deals with metrics for quantifying the performance of a parallel algorithm.

Parallel Programming This section includes Chapters 6 and 7 of the book. Chapter 6, Programming Using the Message-Passing Paradigm, focuses on the Message Passing Interface (MPI) for programming message passing platforms, including clusters. Chapter 7, Programming Shared Address Space Platforms, deals with programming paradigms such as threads and directive based approaches. Using paradigms such as POSIX threads and OpenMP, it describes various features necessary for programming shared-address-space parallel machines. Both of these chapters illustrate various programming concepts using a variety of examples of parallel programs.

Non-numerical Algorithms Chapters 9–12 present parallel non-numerical algorithms. Chapter 9 addresses sorting algorithms such as bitonic sort, bubble sort and its variants, quicksort, sample sort, and shellsort. Chapter 10 describes algorithms for various graph theory problems such as minimum spanning tree, shortest paths, and connected components. Algorithms for sparse graphs are also discussed. Chapter 11 addresses search-based methods such as branch and-bound and heuristic search for combinatorial problems. Chapter 12 classifies and presents parallel formulations for a variety of dynamic programming algorithms.

Numerical Algorithms Chapters 8 and 13 present parallel numerical algorithms. Chapter 8 covers basic operations on dense matrices such as matrix multiplication, matrix-vector multiplication, and Gaussian elimination. This chapter is included before non-numerical algorithms, as the techniques for partitioning and assigning matrices to processors are common to many non-numerical algorithms. Furthermore, matrix-vector and matrix-matrix multiplication algorithms form the kernels of many graph algorithms. Chapter 13 describes algorithms for computing Fast Fourier Transforms.

1.4 Bibliographic Remarks

Many books discuss aspects of parallel processing at varying levels of detail. Hardware aspects of parallel computers have been discussed extensively in several textbooks and monographs [CSG98, LW95, HX98, AG94, Fly95, AG94, Sto93, DeC89, HB84, RF89, Sie85, Tab90, Tab91, WF84, Woo86]. A number of texts discuss paradigms and languages for programming parallel computers [LB98, Pac98, GLS99, GSNL98, CDK $^+$00, WA98, And91, BA82, Bab88, Ble90, Con89, CT92, Les93, Per87, Wal91]. Akl [Akl97], Cole [Col89], Gibbons and Rytter [GR90], Foster [Fos95], Leighton [Lei92], Miller and Stout [MS96], and Quinn [Qui94] discuss various aspects of parallel algorithm design and analysis. Buyya (Editor) [Buy99] and Pfister [Pfi98] discuss various aspects of parallel computing using clusters. Jaja [Jaj92] covers parallel algorithms for the PRAM model of computation. Hillis [Hil85, HS86] and Hatcher and Quinn [HQ91] discuss data-parallel programming. Agha [Agh86] discusses a model of concurrent computation based on *actors*. Sharp [Sha85] addresses data-flow computing. Some books provide a general overview of topics in parallel computing [CL93, Fou94, Zom96, JGD87, LER92, Mol93, Qui94]. Many books address parallel processing applications in numerical analysis and scientific computing [DDSV99, FJDS96, GO93, Car89]. Fox *et al.* [FJL$^+$88] and Angus *et al.* [AFKW90] provide an application-oriented view of algorithm design for problems in scientific computing. Bertsekas and Tsitsiklis [BT97] discuss parallel algorithms, with emphasis on numerical applications.

Akl and Lyons [AL93] discuss parallel algorithms in computational geometry. Ranka and Sahni [RS90b] and Dew, Earnshaw, and Heywood [DEH89] address parallel algorithms for use in computer vision. Green [Gre91] covers parallel algorithms for graphics applications. Many books address the use of parallel processing in artificial intelligence applications [Gup87, HD89b, KGK90, KKKS94, Kow88, RZ89].

A useful collection of reviews, bibliographies and indexes has been put together by the

Association for Computing Machinery [ACM91]. Messina and Murli [MM91] present a collection of papers on various aspects of the application and potential of parallel computing. The scope of parallel processing and various aspects of US government support have also been discussed in National Science Foundation reports [NSF91, GOV99].

A number of conferences address various aspects of parallel computing. A few important ones are the Supercomputing Conference, ACM Symposium on Parallel Algorithms and Architectures, the International Conference on Parallel Processing, the International Parallel and Distributed Processing Symposium, Parallel Computing, and the SIAM Conference on Parallel Processing. Important journals in parallel processing include IEEE Transactions on Parallel and Distributed Systems, International Journal of Parallel Programming, Journal of Parallel and Distributed Computing, Parallel Computing, IEEE Concurrency, and Parallel Processing Letters. These proceedings and journals provide a rich source of information on the state of the art in parallel processing.

[Team LiB]
[Team LiB]

Problems

1.1 Go to the Top 500 Supercomputers site (http://www.top500.org/) and list the five most powerful supercomputers along with their FLOPS rating.

1.2 List three major problems requiring the use of supercomputing in the following domains:

1. Structural Mechanics.
2. Computational Biology.
3. Commercial Applications.

1.3 Collect statistics on the number of components in state of the art integrated circuits over the years. Plot the number of components as a function of time and compare the growth rate to that dictated by Moore's law.

1.4 Repeat the above experiment for the peak FLOPS rate of processors and compare the speed to that inferred from Moore's law.

[Team LiB]
[Team LiB]

Chapter 2. Parallel Programming Platforms

The traditional logical view of a sequential computer consists of a memory connected to a processor via a datapath. All three components – processor, memory, and datapath – present bottlenecks to the overall processing rate of a computer system. A number of architectural innovations over the years have addressed these bottlenecks. One of the most important innovations is multiplicity – in processing units, datapaths, and memory units. This multiplicity is either entirely hidden from the programmer, as in the case of implicit parallelism, or exposed to the programmer in different forms. In this chapter, we present an overview of important architectural concepts as they relate to parallel processing. The objective is to provide sufficient detail for programmers to be able to write efficient code on a variety of platforms. We develop cost models and abstractions for quantifying the performance of various parallel algorithms, and identify bottlenecks resulting from various programming constructs.

We start our discussion of parallel platforms with an overview of serial and implicitly parallel architectures. This is necessitated by the fact that it is often possible to re-engineer codes to achieve significant speedups (2 x to 5 x unoptimized speed) using simple program transformations. Parallelizing sub-optimal serial codes often has undesirable effects of unreliable speedups and misleading runtimes. For this reason, we advocate optimizing serial performance of codes before attempting parallelization. As we shall demonstrate through this chapter, the tasks of serial and parallel optimization often have very similar characteristics. After discussing serial and implicitly parallel architectures, we devote the rest of this chapter to organization of parallel platforms, underlying cost models for algorithms, and platform abstractions for portable algorithm design. Readers wishing to delve directly into parallel architectures may choose to skip Sections 2.1 and 2.2.

2.1 Implicit Parallelism: Trends in Microprocessor Architectures*

While microprocessor technology has delivered significant improvements in clock speeds over the past decade, it has also exposed a variety of other performance bottlenecks. To alleviate these bottlenecks, microprocessor designers have explored alternate routes to cost-effective performance gains. In this section, we will outline some of these trends with a view to understanding their limitations and how they impact algorithm and code development. The objective here is not to provide a comprehensive description of processor architectures. There are several excellent texts referenced in the bibliography that address this topic.

Clock speeds of microprocessors have posted impressive gains - two to three orders of magnitude over the past 20 years. However, these increments in clock speed are severely diluted by the limitations of memory technology. At the same time, higher levels of device integration have also resulted in a very large transistor count, raising the obvious issue of how best to utilize them. Consequently, techniques that enable execution of multiple instructions in a single clock cycle have become popular. Indeed, this trend is evident in the current generation of microprocessors such as the Itanium, Sparc Ultra, MIPS, and Power4. In this section, we briefly explore mechanisms used by various processors for supporting multiple instruction execution.

2.1.1 Pipelining and Superscalar Execution

Processors have long relied on pipelines for improving execution rates. By overlapping various stages in instruction execution (fetch, schedule, decode, operand fetch, execute, store, among others), pipelining enables faster execution. The assembly-line analogy works well for understanding pipelines. If the assembly of a car, taking 100 time units, can be broken into 10 pipelined stages of 10 units each, a single assembly line can produce a car every 10 time units! This represents a 10-fold speedup over producing cars entirely serially, one after the other. It is also evident from this example that to increase the speed of a single pipeline, one would break down the tasks into smaller and smaller units, thus lengthening the pipeline and increasing overlap in execution. In the context of processors, this enables faster clock rates since the tasks are now smaller. For example, the Pentium 4, which operates at 2.0 GHz, has a 20 stage pipeline. Note that the speed of a single pipeline is ultimately limited by the largest atomic task in the pipeline. Furthermore, in typical instruction traces, every fifth to sixth instruction is a branch instruction. Long instruction pipelines therefore need effective techniques for predicting branch destinations so that pipelines can be speculatively filled. The penalty of a misprediction increases as the pipelines become deeper since a larger number of instructions need to be flushed. These factors place limitations on the depth of a processor pipeline and the resulting performance gains.

An obvious way to improve instruction execution rate beyond this level is to use multiple pipelines. During each clock cycle, multiple instructions are piped into the processor in parallel. These instructions are executed on multiple functional units. We illustrate this process with the help of an example.

Example 2.1 Superscalar execution

Consider a processor with two pipelines and the ability to simultaneously issue two instructions. These processors are sometimes also referred to as super-pipelined processors. The ability of a processor to issue multiple instructions in the same cycle is referred to as superscalar execution. Since the architecture illustrated in Figure 2.1 allows two issues per clock cycle, it is also referred to as two-way superscalar or dual issue execution.

Figure 2.1. Example of a two-way superscalar execution of instructions.

Consider the execution of the first code fragment in Figure 2.1 for adding four numbers. The first and second instructions are independent and therefore can be issued concurrently. This is illustrated in the simultaneous issue of the instructions `load R1, @1000` and `load R2, @1008` at *t* = 0. The instructions are fetched, decoded, and the operands are fetched. The next two instructions, `add R1, @1004` and `add R2, @100C` are also mutually independent, although they must be executed after the first two instructions. Consequently, they can be issued concurrently at *t* = 1 since the processors are pipelined. These instructions terminate at *t* = 5. The next two instructions, `add R1, R2` and `store R1, @2000` cannot be executed concurrently since the result of the former (contents of register `R1`) is used by the latter. Therefore, only the `add` instruction is issued at *t* = 2 and the `store` instruction at *t* = 3. Note that the instruction `add R1, R2` can be executed only after the previous two instructions have been executed. The instruction schedule is illustrated in Figure 2.1(b). The schedule assumes that each memory access takes a single cycle. In reality, this may not be the case. The implications of this assumption are discussed in Section 2.2 on memory system performance.

In principle, superscalar execution seems natural, even simple. However, a number of issues need to be resolved. First, as illustrated in Example 2.1, instructions in a program may be related to each other. The results of an instruction may be required for subsequent instructions. This is referred to as ***true data dependency***. For instance, consider the second code fragment in Figure 2.1 for adding four numbers. There is a true data dependency between `load R1, @1000` and `add R1, @1004`, and similarly between subsequent instructions. Dependencies of this type must be resolved before simultaneous issue of instructions. This has two implications. First, since the resolution is done at runtime, it must be supported in hardware. The complexity of this hardware can be high. Second, the amount of instruction level parallelism in a program is often limited and is a function of coding technique. In the second code fragment, there can be no simultaneous issue, leading to poor resource utilization. The three code fragments in Figure 2.1(a) also illustrate that in many cases it is possible to extract more parallelism by reordering the instructions and by altering the code. Notice that in this example the code reorganization corresponds to exposing parallelism in a form that can be used by the instruction issue mechanism.

Another source of dependency between instructions results from the finite resources shared by various pipelines. As an example, consider the co-scheduling of two floating point operations on a dual issue machine with a single floating point unit. Although there might be no data dependencies between the instructions, they cannot be scheduled together since both need the floating point unit. This form of dependency in which two instructions compete for a single processor resource is referred to as ***resource dependency***.

The flow of control through a program enforces a third form of dependency between instructions. Consider the execution of a conditional branch instruction. Since the branch destination is known only at the point of execution, scheduling instructions *a priori* across branches may lead to errors. These dependencies are referred to as ***branch dependencies*** or ***procedural dependencies*** and are typically handled by speculatively scheduling across branches and rolling back in case of errors. Studies of typical traces have shown that on average, a branch instruction is encountered between every five to six instructions. Therefore, just as in populating instruction pipelines, accurate branch prediction is critical for efficient superscalar execution.

The ability of a processor to detect and schedule concurrent instructions is critical to superscalar performance. For instance, consider the third code fragment in Figure 2.1 which also computes the sum of four numbers. The reader will note that this is merely a semantically equivalent reordering of the first code fragment. However, in this case, there is a data dependency between the first two instructions – `load R1, @1000` and `add R1, @1004`. Therefore, these instructions cannot be issued simultaneously. However, if the processor had the ability to look ahead, it would realize that it is possible to schedule the third instruction – `load R2, @1008` – with the first instruction. In the next issue cycle, instructions two and four can be scheduled, and so on. In this way, the same execution schedule can be derived for the first and third code fragments. However, the processor needs the ability to issue instructions ***out-of-order*** to accomplish desired reordering. The parallelism available in ***in-order*** issue of instructions can be highly limited as illustrated by this example. Most current microprocessors are capable of out of-order issue and completion. This model, also referred to as ***dynamic instruction issue***, exploits maximum instruction level parallelism. The processor uses a window of instructions from which it selects instructions for simultaneous issue. This window corresponds to the look ahead of the scheduler.

The performance of superscalar architectures is limited by the available instruction level parallelism. Consider the example in Figure 2.1. For simplicity of discussion, let us ignore the pipelining aspects of the example and focus on the execution aspects of the program. Assuming two execution units (multiply-add units), the figure illustrates that there are several zero-issue cycles (cycles in which the floating point unit is idle). These are essentially wasted cycles from the point of view of the execution unit. If, during a particular cycle, no instructions are issued on the execution units, it is referred to as ***vertical waste***; if only part of the execution units are used during a cycle, it is termed ***horizontal waste***. In the example, we have two cycles of vertical waste and one cycle with horizontal waste. In all, only three of the eight available cycles are used for computation. This implies that the code fragment will yield no more than three eighths of the peak rated FLOP count of the processor. Often, due to limited parallelism, resource dependencies, or the inability of a processor to extract parallelism, the resources of superscalar processors are heavily under-utilized. Current microprocessors typically support up to four-issue superscalar execution.

2.1.2 Very Long Instruction Word Processors

The parallelism extracted by superscalar processors is often limited by the instruction look ahead. The hardware logic for dynamic dependency analysis is typically in the range of 5-10% of the total logic on conventional microprocessors (about 5% on the four-way superscalar Sun UltraSPARC). This complexity grows roughly quadratically with the number of issues and can become a bottleneck. An alternate concept for exploiting instruction-level parallelism used in very long instruction word (VLIW) processors relies on the compiler to resolve dependencies and resource availability at compile time. Instructions that can be executed concurrently are packed into groups and parceled off to the processor as a single long instruction word (thus the name) to be executed on multiple functional units at the same time.

The VLIW concept, first used in Multiflow Trace (circa 1984) and subsequently as a variant in the Intel IA64 architecture, has both advantages and disadvantages compared to superscalar processors. Since scheduling is done in software, the decoding and instruction issue mechanisms are simpler in VLIW processors. The compiler has a larger context from which to select instructions and can use a variety of transformations to optimize parallelism when compared to a hardware issue unit. Additional parallel instructions are typically made available to the compiler to control parallel execution. However, compilers do not have the dynamic program state (e.g., the branch history buffer) available to make scheduling decisions. This reduces the accuracy of branch and memory prediction, but allows the use of more sophisticated static prediction schemes. Other runtime situations such as stalls on data fetch because of cache misses are extremely difficult to predict accurately. This limits the scope and performance of static compiler-based scheduling.

Finally, the performance of VLIW processors is very sensitive to the compilers' ability to detect data and resource dependencies and read and write hazards, and to schedule instructions for maximum parallelism. Loop unrolling, branch prediction and speculative execution all play important roles in the performance of VLIW processors. While superscalar and VLIW processors have been successful in exploiting implicit parallelism, they are generally limited to smaller scales of concurrency in the range of four- to eight-way parallelism.

2.2 Limitations of Memory System Performance*

The effective performance of a program on a computer relies not just on the speed of the processor but also on the ability of the memory system to feed data to the processor. At the logical level, a memory system, possibly consisting of multiple levels of caches, takes in a request for a memory word and returns a block of data of size b containing the requested word after l nanoseconds. Here, l is referred to as the ***latency*** of the memory. The rate at which data can be pumped from the memory to the processor determines the ***bandwidth*** of the memory system.

It is very important to understand the difference between latency and bandwidth since different, often competing, techniques are required for addressing these. As an analogy, if water comes out of the end of a fire hose 2 seconds after a hydrant is turned on, then the latency of the system is 2 seconds. Once the flow starts, if the hose pumps water at 1 gallon/second then the 'bandwidth' of the hose is 1 gallon/second. If we need to put out a fire immediately, we might desire a lower latency. This would typically require higher water pressure from the hydrant. On the other hand, if we wish to fight bigger fires, we might desire a higher flow rate, necessitating a wider hose and hydrant. As we shall see here, this analogy works well for memory systems as well. Latency and bandwidth both play critical roles in determining memory system performance. We examine these separately in greater detail using a few examples.

To study the effect of memory system latency, we assume in the following examples that a memory block consists of one word. We later relax this assumption while examining the role of memory bandwidth. Since we are primarily interested in maximum achievable performance, we also assume the best case cache-replacement policy. We refer the reader to the bibliography for a detailed discussion of memory system design.

Example 2.2 Effect of memory latency on performance

Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The peak processor rating is therefore 4 GFLOPS. Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data. Consider the problem of computing the dot product of two vectors on such a platform. A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch. It is easy to see that the peak speed of this computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS, a very small fraction of the peak processor rating. This example highlights the need for effective memory system performance in achieving high computation rates. □

2.2.1 Improving Effective Memory Latency Using Caches

Handling the mismatch in processor and DRAM speeds has motivated a number of architectural innovations in memory system design. One such innovation addresses the speed mismatch by placing a smaller and faster memory between the processor and the DRAM. This memory, referred to as the cache, acts as a low-latency high-bandwidth storage. The data needed by the processor is first fetched into the cache. All subsequent accesses to data items residing in the cache are serviced by the cache. Thus, in principle, if a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache. The fraction of data references satisfied by the cache is called the cache ***hit ratio*** of the computation on the system. The effective computation rate of many applications is bounded not by the processing rate of the CPU, but by the rate at which data can be pumped into the CPU. Such computations are referred to as being ***memory bound***. The performance of memory bound programs is critically impacted by the cache hit ratio.

Example 2.3 Impact of caches on memory system performance

As in the previous example, consider a 1 GHz processor with a 100 ns latency DRAM. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle (typically on the processor itself). We use this setup to multiply two matrices *A* and *B* of dimensions 32 x 32. We have carefully chosen these numbers so that the cache is large enough to store matrices *A* and *B*, as well as the result matrix *C*. Once again, we assume an ideal cache placement strategy in which none of the data items are overwritten by others. Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 µs. We know from elementary algorithmics that multiplying two $n \times n$ matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16 µs) at four instructions per cycle. The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., 200+16 µs. This corresponds to a peak computation rate of 64K/216 or 303 MFLOPS. Note that this is a thirty-fold improvement over the previous example, although it is still less than 10% of the peak processor performance. We see in this example that by placing a small cache memory, we are able to improve processor utilization considerably.

The improvement in performance resulting from the presence of the cache is based on the assumption that there is repeated reference to the same data item. This notion of repeated reference to a data item in a small time window is called ***temporal locality*** of reference. In our example, we had $O(n^2)$ data accesses and $O(n^3)$ computation. (See the Appendix for an explanation of the *O* notation.) Data reuse is critical for cache performance because if each data item is used only once, it would still have to be fetched once per use from the DRAM, and therefore the DRAM latency would be paid for each operation.

2.2.2 Impact of Memory Bandwidth

Memory bandwidth refers to the rate at which data can be moved between the processor and memory. It is determined by the bandwidth of the memory bus as well as the memory units. One commonly used technique to improve memory bandwidth is to increase the size of the memory blocks. For an illustration, let us relax our simplifying restriction on the size of the memory block and assume that a single memory request returns a contiguous block of four words. The single unit of four words in this case is also referred to as a ***cache line***. Conventional computers typically fetch two to eight words together into the cache. We will see how this helps the performance of applications for which data reuse is limited.

Example 2.4 Effect of block size: dot-product of two vectors

Consider again a memory system with a single cycle cache and 100 cycle latency DRAM with the processor operating at 1 GHz. If the block size is one word, the processor takes 100 cycles to fetch each word. For each pair of words, the dot-product performs one multiply-add, i.e., two FLOPs. Therefore, the algorithm performs one FLOP every 100 cycles for a peak speed of 10 MFLOPS as illustrated in Example 2.2.

Now let us consider what happens if the block size is increased to four words, i.e., the processor can fetch a four-word cache line every 100 cycles. Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles. This is because a single memory access fetches four consecutive words in the vector. Therefore, two accesses can fetch four elements of each of the vectors. This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS. Note that increasing the block size from one to four words did not change the latency of the memory system. However, it increased the bandwidth four-fold. In this case, the increased bandwidth of the memory system enabled us to accelerate the dot-product algorithm which has no data reuse at all.

Another way of quickly estimating performance bounds is to estimate the cache hit ratio, using it to compute mean access time per word, and relating this to the FLOP rate via the underlying algorithm. For example, in this example, there are two DRAM accesses (cache misses) for every eight data accesses required by the algorithm. This corresponds to a cache hit ratio of 75%. Assuming that the dominant overhead is posed by the cache misses, the average memory access time contributed by the misses is 25% at 100 ns (or 25 ns/word). Since the dot-product has one operation/word, this corresponds to a computation rate of 40 MFLOPS as before. A more accurate estimate of this rate would compute the average memory access time as 0.75 x 1 + 0.25 x 100 or 25.75 ns/word. The corresponding computation rate is 38.8 MFLOPS.

Physically, the scenario illustrated in Example 2.4 corresponds to a wide data bus (4 words or 128 bits) connected to multiple memory banks. In practice, such wide buses are expensive to construct. In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved. For example, with a 32 bit data bus, the first word is put on the bus after 100 ns (the associated latency) and one word is put on each subsequent bus cycle. This changes our calculations above slightly since the entire cache line becomes available only after 100 + 3 x (memory bus cycle) ns. Assuming a data bus operating at 200 MHz, this adds 15 ns to the cache line access time. This does not change our bound on the execution rate significantly.

The above examples clearly illustrate how increased bandwidth results in higher peak computation rates. They also make certain assumptions that have significance for the programmer. The data layouts were assumed to be such that consecutive data words in memory were used by successive instructions. In other words, if we take a computation-centric view, there is a ***spatial locality*** of memory access. If we take a data-layout centric point of view, the computation is ordered so that successive computations require contiguous data. If the computation (or access pattern) does not have spatial locality, then effective bandwidth can be much smaller than the peak bandwidth.
An example of such an access pattern is in reading a dense matrix column-wise when the matrix has been stored in a row-major fashion in memory. Compilers can often be relied on to do a good job of restructuring computation to take advantage of spatial locality.

Example 2.5 Impact of strided access

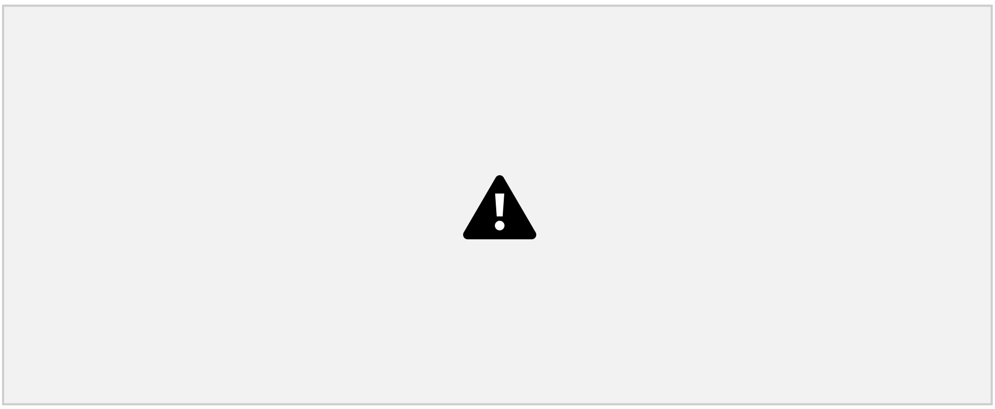
Consider the following code fragment:

```
for (i = 0; i < 1000; i++)
column_sum[i] = 0.0;
```

```
for (j = 0; j < 1000; j++)
column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix `b` into a vector `column_sum`. There are two observations that can be made: (i) the vector `column_sum` is small and easily fits into the cache; and (ii) the matrix `b` is accessed in a column order as illustrated in Figure 2.2(a). For a matrix of size 1000 x 1000, stored in a row-major order, this corresponds to accessing every 1000*th* entry. Therefore, it is likely that only one word in each cache line fetched from memory will be used. Consequently, the code fragment as written above is likely to yield poor performance.

Figure 2.2. Multiplying a matrix with a vector: (a) multiplying column-by-column, keeping a running sum; (b) computing each element of the result as a dot product of a row of the matrix with the vector.



The above example illustrates problems with strided access (with strides greater than one). The lack of spatial locality in computation causes poor memory system performance. Often it is possible to restructure the computation to remove strided access. In the case of our example, a simple rewrite of the loops is possible as follows:

Example 2.6 Eliminating strided access

Consider the following restructuring of the column-sum fragment:

```
for (i = 0; i < 1000; i++)
column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
for (i = 0; i < 1000; i++)
column_sum[i] += b[j][i];
```

In this case, the matrix is traversed in a row-order as illustrated in Figure 2.2(b). However, the reader will note that this code fragment relies on the fact that the vector `column_sum` can be retained in the cache through the loops. Indeed, for this particular example, our assumption is reasonable. If the vector is larger, we would have to break the iteration space into blocks and compute the product one block at a time. This concept is also called ***tiling*** an iteration space. The improved performance of this loop is left as an exercise for the reader.

So the next question is whether we have effectively solved the problems posed by memory latency and bandwidth. While peak processor rates have grown significantly over the past decades, memory latency and bandwidth have not kept pace with this increase. Consequently, for typical computers, the ratio of peak FLOPS rate to peak memory bandwidth is anywhere between 1 MFLOPS/MBs (the ratio signifies FLOPS per megabyte/second of bandwidth) to 100 MFLOPS/MBs. The lower figure typically corresponds to large scale vector supercomputers and the higher figure to fast microprocessor based computers. This figure is very revealing in that it tells us that on average, a word must be reused 100 times after being fetched into the full bandwidth storage (typically L1 cache) to be able to achieve full processor utilization. Here, we define full-bandwidth as the rate of data transfer required by a computation to make it processor bound.

The series of examples presented in this section illustrate the following concepts:

Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth.

Certain applications have inherently greater temporal locality than others, and thus have greater tolerance to low memory bandwidth. The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth.

Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality.

2.2.3 Alternate Approaches for Hiding Memory Latency

Imagine sitting at your computer browsing the web during peak network traffic hours. The lack of response from your browser can be alleviated using one of three simple approaches:

(i) we anticipate which pages we are going to browse ahead of time and issue requests for them in advance; (ii) we open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or (iii) we access a whole bunch of pages in one go – amortizing the latency across various accesses. The first approach is called ***prefetching***, the second ***multithreading***, and the third one corresponds to spatial locality in accessing memory words. Of these three approaches, spatial locality of memory accesses has been discussed before. We focus on prefetching and multithreading as techniques for latency hiding in this section.

Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program. We illustrate threads with a simple example:

Example 2.7 Threaded execution of matrix multiplication

Consider the following code segment for multiplying an $n \times n$ matrix `a` by a vector `b` to get vector `c`.

```
for(i=0;i<n;i++)
c[i] = dot_product(get_row(a, i), b);
```

This code computes each element of `c` as the dot product of the corresponding row of `a` with the vector `b`. Notice that each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

```
for(i=0;i<n;i++)
c[i] = create_thread(dot_product, get_row(a, i), b);
```

The only difference between the two code segments is that we have explicitly specified each instance of the dot-product computation as being a thread. (As we shall learn in Chapter 7, there are a number of APIs for specifying threads. We have simply chosen an intuitive name for a function to create threads.) Now, consider the execution of each instance of the function `dot_product`. The first instance of this function accesses a pair of vector elements and waits for them. In the meantime, the second instance of this function can access two other vector elements in the next cycle, and so on. After *l* units of time, where *l* is the latency of the memory system, the first function instance gets the requested data from memory and can perform the required computation. In the next cycle, the data items for the next function instance arrive, and so on. In this way, in every clock cycle, we can perform a computation.

The execution schedule in Example 2.7 is predicated upon two assumptions: the memory system is capable of servicing multiple outstanding requests, and the processor is capable of switching threads at every cycle. In addition, it also requires the program to have an explicit specification of concurrency in the form of threads. Multithreaded processors are capable of maintaining the context of a number of threads of computation with outstanding requests (memory accesses, I/O, or communication requests) and execute them as the requests are satisfied. Machines such as the HEP and Tera rely on multithreaded processors that can switch the context of execution in every cycle. Consequently, they are able to hide latency effectively, provided there is enough concurrency (threads) to keep the processor from idling. The tradeoffs between concurrency and latency will be a recurring theme through many chapters of this text.

Prefetching for Latency Hiding

In a typical program, a data item is loaded and used by a processor in a small time window. If the load results in a cache miss, then the use stalls. A simple solution to this problem is to advance the load operation so that even if there is a cache miss, the data is likely to have arrived by the time it is used. However, if the data item has been overwritten between load and use, a fresh load is issued. Note that this is no worse than the situation in which the load had not been advanced. A careful examination of this technique reveals that prefetching works for much the same reason as multithreading. In advancing the loads, we are trying to identify independent threads of execution that have no resource dependency (i.e., use the same registers) with respect to other threads. Many compilers aggressively try to advance loads to mask memory system latency.

Example 2.8 Hiding latency by prefetching

Consider the problem of adding two vectors `a` and `b` using a single `for` loop. In the first iteration of the loop, the processor requests `a[0]` and `b[0]`. Since these are not in the cache, the processor must pay the memory latency. While these requests are being serviced, the processor also requests `a[1]` and `b[1]`. Assuming that each request is generated in one cycle (1 ns) and memory requests are satisfied in 100 ns, after 100 such requests the first set of data items is returned by the memory system. Subsequently, one pair of vector components will be returned every cycle. In this way, in each subsequent cycle, one addition can be performed and processor cycles are not wasted.

2.2.4 Tradeoffs of Multithreading and Prefetching

While it might seem that multithreading and prefetching solve all the problems related to memory system performance, they are critically impacted by the memory bandwidth.

Example 2.9 Impact of bandwidth on multithreaded programs

Consider a computation running on a machine with a 1 GHz clock, 4-word cache line, single cycle access to the cache, and 100 ns latency to DRAM. The computation has a cache hit ratio at 1 KB of 25% and at 32 KB of 90%. Consider two cases: first, a single threaded execution in which the entire cache is available to the serial context, and second, a multithreaded execution with 32 threads where each thread has a cache residency of 1 KB. If the computation makes one data request in every cycle of 1 ns, in the first case the bandwidth requirement to DRAM is one word every 10 ns since the other words come from the cache (90% cache hit ratio). This corresponds to a bandwidth of 400 MB/s. In the second case, the bandwidth requirement to DRAM increases to three words every four cycles of each thread (25% cache hit ratio). Assuming that all threads exhibit similar cache behavior, this corresponds to 0.75 words/ns, or 3 GB/s.

Example 2.9 illustrates a very important issue, namely that the bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread. In the example, while a sustained DRAM bandwidth of 400 MB/s is reasonable, 3.0 GB/s is more than most systems currently offer. At this point, multithreaded systems become bandwidth bound instead of latency bound. It is important to realize that multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem.

Another issue relates to the additional hardware resources required to effectively use prefetching and multithreading. Consider a situation in which we have advanced 10 loads into registers. These loads require 10 registers to be free for the duration. If an intervening instruction overwrites the registers, we would have to load the data again. This would not increase the latency of the fetch any more than the case in which there was no prefetching. However, now we are fetching the same data item twice, resulting in doubling of the bandwidth requirement from the memory system. This situation is similar to the one due to cache constraints as illustrated in Example 2.9. It can be alleviated by supporting prefetching and multithreading with larger register files and caches.

[Team LiB]

[Team LiB]

2.3 Dichotomy of Parallel Computing Platforms

In the preceding sections, we pointed out various factors that impact the performance of a serial or implicitly parallel program. The increasing gap in peak and sustainable performance of current microprocessors, the impact of memory system performance, and the distributed nature of many problems present overarching motivations for parallelism. We now introduce, at a high level, the elements of parallel computing platforms that are critical for performance oriented and portable parallel programming. To facilitate our discussion of parallel platforms, we first explore a dichotomy based on the logical and physical organization of parallel platforms. The logical organization refers to a programmer's view of the platform while the physical organization refers to the actual hardware organization of the platform. The two critical components of parallel computing from a programmer's perspective are ways of expressing parallel tasks and mechanisms for specifying interaction between these tasks. The former is sometimes also referred to as the control structure and the latter as the communication model.

2.3.1 Control Structure of Parallel Platforms

Parallel tasks can be specified at various levels of granularity. At one extreme, each program in a set of programs can be viewed as one parallel task. At the other extreme, individual instructions within a program can be viewed as parallel tasks. Between these extremes lie a range of models for specifying the control structure of programs and the corresponding architectural support for them.

Example 2.10 Parallelism from single instruction on multiple processors

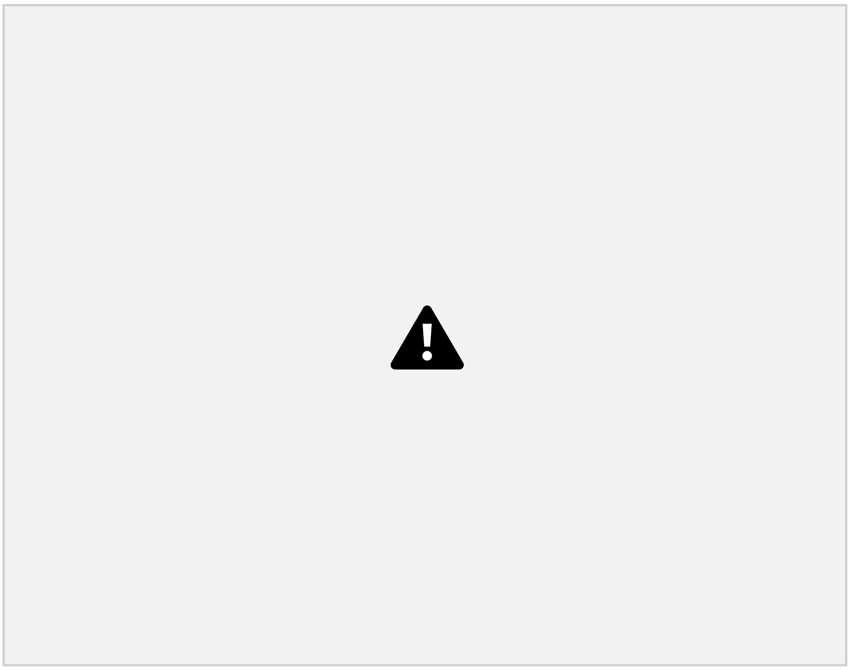
Consider the following code segment that adds two vectors:

```
for (i = 0; i < 1000; i++)
c[i] = a[i] + b[i];
```

In this example, various iterations of the loop are independent of each other; i.e., `c[0] = a[0] + b[0]; c[1] = a[1] + b[1];`, etc., can all be executed independently of each other. Consequently, if there is a mechanism for executing the same instruction, in this case `add` on all the processors with appropriate data, we could execute this loop much faster.

Processing units in parallel computers either operate under the centralized control of a single control unit or work independently. In architectures referred to as ***single instruction stream, multiple data stream*** (SIMD), a single control unit dispatches instructions to each processing unit. Figure 2.3(a) illustrates a typical SIMD architecture. In an SIMD parallel computer, the same instruction is executed synchronously by all processing units. In Example 2.10, the `add` instruction is dispatched to all processors and executed concurrently by them. Some of the earliest parallel computers such as the Illiac IV, MPP, DAP, CM-2, and MasPar MP-1 belonged to this class of machines. More recently, variants of this concept have found use in co-processing units such as the MMX units in Intel processors and DSP chips such as the Sharc. The Intel Pentium processor with its SSE (Streaming SIMD Extensions) provides a number of instructions that execute the same instruction on multiple data items. These architectural enhancements rely on the highly structured (regular) nature of the underlying computations, for example in image processing and graphics, to deliver improved performance.

Figure 2.3. A typical SIMD architecture (a) and a typical MIMD architecture (b).



While the SIMD concept works well for structured computations on parallel data structures such as arrays, often it is necessary to selectively turn off operations on certain data items. For this reason, most SIMD programming paradigms allow for an "activity mask". This is a binary mask associated with each data item and operation that specifies whether it should participate in the operation or not. Primitives such as `where (condition) then <stmnt> <elsewhere stmnt>` are used to support selective execution. Conditional execution can be detrimental to the performance of SIMD processors and therefore must be used with care.

In contrast to SIMD architectures, computers in which each processing element is capable of executing a different program independent of the other processing elements are called ***multiple instruction stream, multiple data stream*** (MIMD) computers. Figure 2.3(b) depicts a typical MIMD computer. A simple variant of this model, called the ***single program multiple data*** (SPMD) model, relies on multiple instances of the same program executing on different data. It is easy to see that the SPMD model has the same expressiveness as the MIMD model since each of the multiple programs can be inserted into one large `if-else` block with conditions specified by the task identifiers. The SPMD model is widely used by many parallel platforms and requires minimal architectural support. Examples of such platforms include the Sun Ultra Servers, multiprocessor PCs, workstation clusters, and the IBM SP.

SIMD computers require less hardware than MIMD computers because they have only one global control unit. Furthermore, SIMD computers require less memory because only one copy of the program needs to be stored. In contrast, MIMD computers store the program and operating system at each processor. However, the relative unpopularity of SIMD processors as general purpose compute engines can be attributed to their specialized hardware architectures, economic factors, design constraints, product life-cycle, and application characteristics. In contrast, platforms supporting the SPMD paradigm can be built from inexpensive off-the-shelf components with relatively little effort in a short amount of time. SIMD computers require extensive design effort resulting in longer product development times. Since the underlying serial processors change so rapidly, SIMD computers suffer from fast obsolescence. The irregular nature of many applications also makes SIMD architectures less suitable. Example 2.11 illustrates a case in which SIMD architectures yield poor resource utilization in the case of conditional execution.

Example 2.11 Execution of conditional statements on a SIMD architecture

Consider the execution of a conditional statement illustrated in Figure 2.4. The conditional statement in Figure 2.4(a) is executed in two steps. In the first step, all processors that have *B* equal to zero execute the instruction *C* = *A*. All other processors are idle. In the second step, the 'else' part of the instruction (*C* = *A*/*B*) is executed. The processors that were active in the first step now become idle. This illustrates one of the drawbacks of SIMD architectures.

Figure 2.4. Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.

2.3.2 Communication Model of Parallel Platforms

There are two primary forms of data exchange between parallel tasks – accessing a shared data space and exchanging messages.

Shared-Address-Space Platforms

The "shared-address-space" view of a parallel platform supports a common data space that is accessible to all processors. Processors interact by modifying data objects stored in this shared address-space. Shared-address-space platforms supporting SPMD programming are also referred to as ***multiprocessors***. Memory in shared-address-space platforms can be local (exclusive to a processor) or global (common to all processors). If the time taken by a processor to access any memory word in the system (global or local) is identical, the platform is classified as a ***uniform memory access*** (UMA) multicomputer. On the other hand, if the time taken to access certain memory words is longer than others, the platform is called a ***non uniform memory access*** (NUMA) multicomputer. Figures 2.5(a) and (b) illustrate UMA platforms, whereas Figure 2.5(c) illustrates a NUMA platform. An interesting case is illustrated in Figure 2.5(b). Here, it is faster to access a memory word in cache than a location in memory. However, we still classify this as a UMA architecture. The reason for this is that all current microprocessors have cache hierarchies. Consequently, even a uniprocessor would not be termed UMA if cache access times are considered. For this reason, we define NUMA and UMA architectures only in terms of memory access times and not cache access times. Machines such as the SGI Origin 2000 and Sun Ultra HPC servers belong to the class of NUMA multiprocessors. The distinction between UMA and NUMA platforms is important. If accessing local memory is cheaper than accessing global memory, algorithms must build locality and structure data and computation accordingly.

Figure 2.5. Typical shared-address-space architectures: (a) Uniform memory-access shared-address-space computer; (b) Uniform memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

The presence of a global memory space makes programming such platforms much easier. All read-only interactions are invisible to the programmer, as they are coded no differently than in a serial program. This greatly eases the burden of writing parallel programs. Read/write interactions are, however, harder to program than the read-only interactions, as these operations require mutual exclusion for concurrent accesses. Shared-address-space programming paradigms such as threads (POSIX, NT) and directives (OpenMP) therefore support synchronization using `locks` and related mechanisms.

The presence of caches on processors also raises the issue of multiple copies of a single memory word being manipulated by two or more processors at the same time. Supporting a shared address-space in this context involves two major tasks: providing an address translation mechanism that locates a memory word in the system, and ensuring that concurrent operations on multiple copies of the same memory word have well-defined semantics. The latter is also referred to as the ***cache coherence*** mechanism. This mechanism and its implementation are discussed in greater detail in Section 2.4.6. Supporting cache coherence requires considerable hardware support. Consequently, some shared-address-space machines only support an address translation mechanism and leave the task of ensuring coherence to the programmer. The native programming model for such platforms consists of primitives such as `get` and `put`. These primitives allow a processor to get (and put) variables stored at a remote processor. However, if one of the copies of this variable is changed, the other copies are not automatically updated or invalidated.

It is important to note the difference between two commonly used and often misunderstood terms – shared-address-space and shared-memory computers. The term shared-memory computer is historically used for architectures in which the memory is physically shared among various processors, i.e., each processor has equal access to any memory segment. This is identical to the UMA model we just discussed. This is in contrast to a distributed-memory computer, in which different segments of the memory are physically associated with different processing elements. The dichotomy of shared- versus distributed-memory computers pertains to the physical organization of the machine and is discussed in greater detail in Section 2.4. Either of these physical models, shared or distributed memory, can present the logical view of a disjoint or shared-address-space platform. A distributed-memory shared-address-space computer is identical to a NUMA machine.

Message-Passing Platforms

The logical machine view of a message-passing platform consists of *p* processing nodes, each with its own exclusive address space. Each of these processing nodes can either be single processors or a shared-address-space multiprocessor – a trend that is fast gaining momentum in modern message-passing parallel computers. Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers. On such platforms, interactions between processes running on different nodes must be accomplished using messages, hence the name ***message passing***. This exchange of messages is used to transfer data, work, and to synchronize actions among the processes. In its most general form, message-passing paradigms support execution of a different program on each of the *p* nodes.

Since interactions are accomplished by sending and receiving messages, the basic operations in this programming paradigm are `send` and `receive` (the corresponding calls may differ across APIs but the semantics are largely identical). In addition, since the send and receive operations must specify target addresses, there must be a mechanism to assign a unique identification or ID to each of the multiple processes executing a parallel program. This ID is typically made available to the program using a function such as `whoami`, which returns to a calling process its ID. There is one other function that is typically needed to complete the basic set of message passing operations – `numprocs`, which specifies the number of processes participating in the ensemble. With these four basic operations, it is possible to write any message-passing program. Different message-passing APIs, such as the Message Passing Interface (MPI) and Parallel Virtual Machine (PVM), support these basic operations and a variety of higher level functionality under different function names. Examples of parallel platforms that support the message-passing paradigm include the IBM SP, SGI Origin 2000, and workstation clusters.

It is easy to emulate a message-passing architecture containing *p* nodes on a shared-address space computer with an identical number of nodes. Assuming uniprocessor nodes, this can be done by partitioning the shared-address-space into *p* disjoint parts and assigning one such partition exclusively to each processor. A processor can then "send" or "receive" messages by writing to or reading from another processor's partition while using appropriate synchronization primitives to inform its communication partner when it has finished reading or writing the data. However, emulating a shared-address-space architecture on a message-passing computer is costly, since accessing another node's memory requires sending and receiving messages.

[Team LiB]
[Team LiB]

2.4 Physical Organization of Parallel Platforms

In this section, we discuss the physical architecture of parallel machines. We start with an ideal architecture, outline practical difficulties associated with realizing this model, and discuss some conventional architectures.

2.4.1 Architecture of an Ideal Parallel Computer

A natural extension of the serial model of computation (the Random Access Machine, or RAM) consists of *p* processors and a global memory of unbounded size that is uniformly accessible to all processors. All processors access the same address space. Processors share a common clock but may execute different instructions in each cycle. This ideal model is also referred to as a ***parallel random access machine (PRAM)***. Since PRAMs allow concurrent access to various memory locations, depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.

1. ***Exclusive-read, exclusive-write (EREW) PRAM.*** In this class, access to a memory location is exclusive. No concurrent read or write operations are allowed. This is the weakest PRAM model, affording minimum concurrency in memory access.

2. ***Concurrent-read, exclusive-write (CREW) PRAM.*** In this class, multiple read accesses to a memory location are allowed. However, multiple write accesses to a memory location are serialized.

3. ***Exclusive-read, concurrent-write (ERCW) PRAM.*** Multiple write accesses are allowed to a memory location, but multiple read accesses are serialized.

4. ***Concurrent-read, concurrent-write (CRCW) PRAM.*** This class allows multiple read and write accesses to a common memory location. This is the most powerful PRAM model.

Allowing concurrent read access does not create any semantic discrepancies in the program. However, concurrent write access to a memory location requires arbitration. Several protocols are used to resolve concurrent writes. The most frequently used protocols are as follows:

Common, in which the concurrent write is allowed if all the values that the processors are attempting to write are identical.

Arbitrary, in which an arbitrary processor is allowed to proceed with the write operation and the rest fail.

Priority, in which all processors are organized into a predefined prioritized list, and the processor with the highest priority succeeds and the rest fail.

Sum, in which the sum of all the quantities is written (the sum-based write conflict resolution model can be extended to any associative operator defined on the quantities being written).

Architectural Complexity of the Ideal Model

Consider the implementation of an EREW PRAM as a shared-memory computer with *p* processors and a global memory of *m* words. The processors are connected to the memory through a set of switches. These switches determine the memory word being accessed by each processor. In an EREW PRAM, each of the *p* processors in the ensemble can access any of the memory words, provided that a word is not accessed by more than one processor simultaneously. To ensure such connectivity, the total number of switches must be (*mp*). (See the Appendix for an explanation of the notation.) For a reasonable memory size, constructing a switching network of this complexity is very expensive. Thus, PRAM models of computation are impossible to realize in practice.

2.4.2 Interconnection Networks for Parallel Computers

Interconnection networks provide mechanisms for data transfer between processing nodes or between processors and memory modules. A blackbox view of an interconnection network consists of *n* inputs and *m* outputs. The outputs may or may not be distinct from the inputs. Typical interconnection networks are built using links and switches. A link corresponds to physical media such as a set of wires or fibers capable of carrying information. A variety of factors influence link characteristics. For links based on conducting media, the capacitive coupling between wires limits the speed of signal propagation. This capacitive coupling and attenuation of signal strength are functions of the length of the link.

Interconnection networks can be classified as ***static*** or ***dynamic***. Static networks consist of point-to-point communication links among processing nodes and are also referred to as ***direct*** networks. Dynamic networks, on the other hand, are built using switches and communication links. Communication links are connected to one another dynamically by the switches to establish paths among processing nodes and memory banks. Dynamic networks are also referred to as ***indirect*** networks. Figure 2.6(a) illustrates a simple static network of four processing elements or nodes. Each processing node is connected via a network interface to two other nodes in a mesh configuration. Figure 2.6(b) illustrates a dynamic network of four nodes connected via a network of switches to other nodes.

Figure 2.6. Classification of interconnection networks: (a) a static network; and (b) a dynamic network.

A single switch in an interconnection network consists of a set of input ports and a set of output ports. Switches provide a range of functionality. The minimal functionality provided by a switch is a mapping from the input to the output ports. The total number of ports on a switch is also called the ***degree*** of the switch. Switches may also provide support for internal buffering (when the requested output port is busy), routing (to alleviate congestion on the network), and multicast (same output on multiple ports). The mapping from input to output ports can be provided using a variety of mechanisms based on physical crossbars, multi-ported memories, multiplexor-demultiplexors, and multiplexed buses. The cost of a switch is influenced by the cost of the mapping hardware, the peripheral hardware and packaging costs. The mapping hardware typically grows as the square of the degree of the switch, the peripheral hardware linearly as the degree, and the packaging costs linearly as the number of pins.

The connectivity between the nodes and the network is provided by a network interface. The network interface has input and output ports that pipe data into and out of the network. It typically has the responsibility of packetizing data, computing routing information, buffering incoming and outgoing data for matching speeds of network and processing elements, and error checking. The position of the interface between the processing element and the network is also important. While conventional network interfaces hang off the I/O buses, interfaces in tightly coupled parallel machines hang off the memory bus. Since I/O buses are typically slower than memory buses, the latter can support higher bandwidth.

2.4.3 Network Topologies

A wide variety of network topologies have been used in interconnection networks. These topologies try to trade off cost and scalability with performance. While pure topologies have attractive mathematical properties, in practice interconnection networks tend to be combinations or modifications of the pure topologies discussed in this section.

Bus-Based Networks

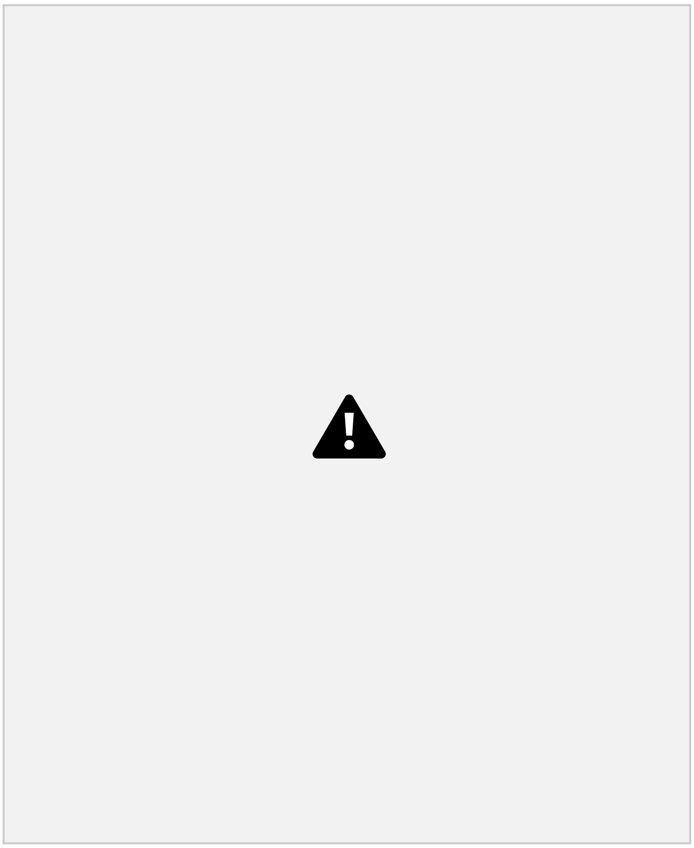
A bus-based network is perhaps the simplest network consisting of a shared medium that is common to all the nodes. A bus has the desirable property that the cost of the network scales linearly as the number of nodes, p. This cost is typically associated with bus interfaces. Furthermore, the distance between any two nodes in the network is constant ($O(1)$). Buses are also ideal for broadcasting information among nodes. Since the transmission medium is shared, there is little overhead associated with broadcast compared to point-to-point message transfer. However, the bounded bandwidth of a bus places limitations on the overall performance of the network as the number of nodes increases. Typical bus based machines are limited to dozens of nodes. Sun Enterprise servers and Intel Pentium based shared-bus multiprocessors are examples of such architectures.

The demands on bus bandwidth can be reduced by making use of the property that in typical programs, a majority of the data accessed is local to the node. For such programs, it is possible to provide a cache for each node. Private data is cached at the node and only remote data is accessed through the bus.

Example 2.12 Reducing shared-bus bandwidth using caches

Figure 2.7(a) illustrates p processors sharing a bus to the memory. Assuming that each processor accesses k data items, and each data access takes time t_{cycle}, the execution time is lower bounded by $t_{cycle} \times kp$ seconds. Now consider the hardware organization of Figure 2.7(b). Let us assume that 50% of the memory accesses (0.5k) are made to local data. This local data resides in the private memory of the processor. We assume that access time to the private memory is identical to the global memory, i.e., t_{cycle}. In this case, the total execution time is lower bounded by $0.5 \times t_{cycle} \times k + 0.5 \times t_{cycle} \times kp$. Here, the first term results from accesses to local data and the second term from access to shared data. It is easy to see that as p becomes large, the organization of Figure 2.7(b) results in a lower bound that approaches $0.5 \times t_{cycle} \times kp$. This time is a 50% improvement in lower bound on execution time compared to the organization of Figure 2.7(a).

Figure 2.7. Bus-based interconnects (a) with no local caches; (b) with local memory/caches.



In practice, shared and private data is handled in a more sophisticated manner. This is briefly addressed with cache coherence issues in Section 2.4.6.

Crossbar Networks

A simple way to connect *p* processors to *b* memory banks is to use a crossbar network. A crossbar network employs a grid of switches or switching nodes as shown in Figure 2.8. The crossbar network is a non-blocking network in the sense that the connection of a processing node to a memory bank does not block the connection of any other processing nodes to other memory banks.

Figure 2.8. A completely non-blocking crossbar network connecting *p* processors to *b* memory banks.

The total number of switching nodes required to implement such a network is (*pb*). It is reasonable to assume that the number of memory banks *b* is at least *p*; otherwise, at any given time, there will be some processing nodes that will be unable to access any memory banks. Therefore, as the value of *p* is increased, the complexity (component count) of the switching network grows as (p^2). (See the Appendix for an explanation of the notation.) As the number of processing nodes becomes large, this switch complexity is difficult to realize at high data rates. Consequently, crossbar networks are not very scalable in terms of cost.

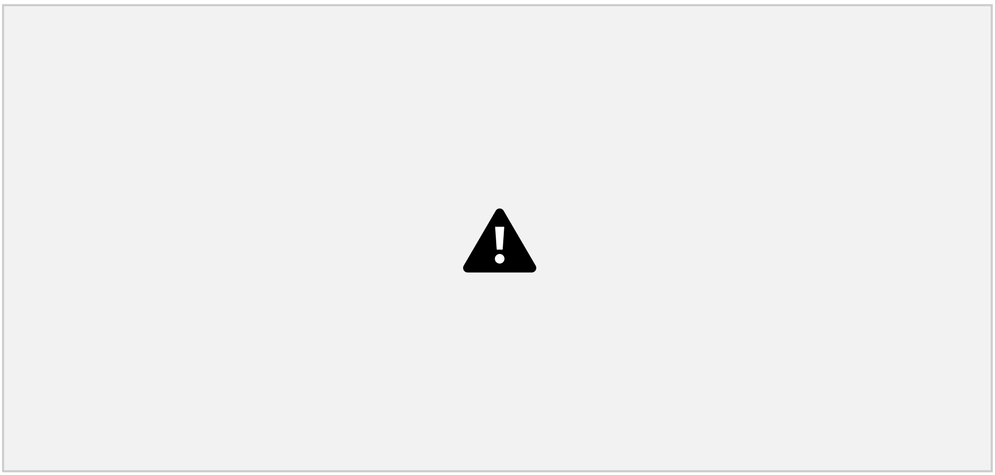
Multistage Networks

The crossbar interconnection network is scalable in terms of performance but unscalable in terms of cost. Conversely, the shared bus network is scalable in terms of cost but unscalable in terms of performance. An intermediate class of networks called ***multistage interconnection networks*** lies between these two extremes. It is more scalable than the bus in terms of performance and more scalable than the crossbar in terms of cost.

The general schematic of a multistage network consisting of *p* processing nodes and *b* memory banks is shown in Figure 2.9. A commonly used multistage connection network is the ***omega network***. This network consists of log *p* stages, where *p* is the number of inputs (processing nodes) and also the number of outputs (memory banks). Each stage of the omega network consists of an interconnection pattern that connects *p* inputs and *p* outputs; a link exists between input *i* and output *j* if the following is true:

Equation 2.1

Figure 2.9. The schematic of a typical multistage interconnection network.



Equation 2.1 represents a left-rotation operation on the binary representation of *i* to obtain *j*. This interconnection pattern is called a ***perfect shuffle***. Figure 2.10 shows a perfect shuffle interconnection pattern for eight inputs and outputs. At each stage of an omega network, a perfect shuffle interconnection pattern feeds into a set of *p*/2 switches or switching nodes. Each switch is in one of two connection modes. In one mode, the inputs are sent straight through to the outputs, as shown in Figure 2.11(a). This is called the ***pass-through*** connection. In the other mode, the inputs to the switching node are crossed over and then sent out, as shown in Figure 2.11(b). This is called the ***cross-over*** connection.

Figure 2.10. A perfect shuffle interconnection for eight inputs and outputs.

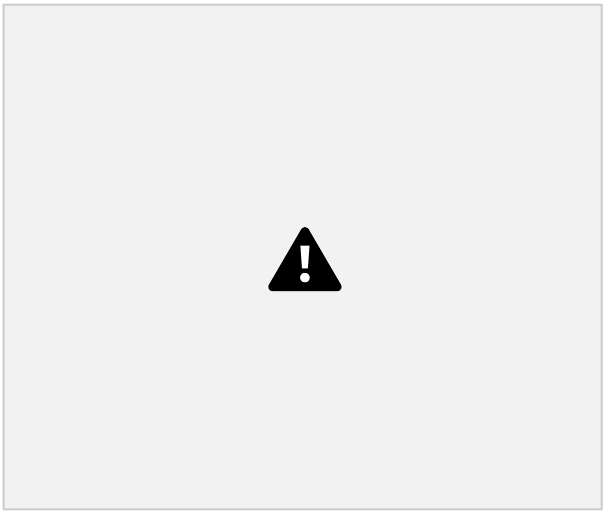


Figure 2.11. Two switching configurations of the 2 x 2 switch: (a) Pass-through; (b) Cross-over.

An omega network has *p*/2 x log *p* switching nodes, and the cost of such a network grows as (*p* log *p*). Note that this cost is less than the (p^2) cost of a complete crossbar network. Figure 2.12 shows an omega network for eight processors (denoted by the binary numbers on the left) and eight memory banks (denoted by the binary numbers on the right). Routing data in an omega network is accomplished using a simple scheme. Let *s* be the binary representation of a processor that needs to write some data into memory bank *t*. The data traverses the link to the first switching node. If the most significant bits of *s* and *t* are the same, then the data is routed in pass-through mode by the switch. If these bits are different, then the data is routed through in crossover mode. This scheme is repeated at the next switching stage using the next most significant bit. Traversing log *p* stages uses all log *p* bits in the binary representations of *s* and *t*.

Figure 2.12. A complete omega network connecting eight inputs and eight outputs.

Figure 2.13 shows data routing over an omega network from processor two (010) to memory bank seven (111) and from processor six (110) to memory bank four (100). This figure also illustrates an important property of this network. When processor two (010) is communicating with memory bank seven (111), it blocks the path from processor six (110) to memory bank four (100). Communication link AB is used by both communication paths. Thus, in an omega network, access to a memory bank by a processor may disallow access to another memory bank by another processor. Networks with this property are referred to as ***blocking networks***.

Figure 2.13. An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.

Completely-Connected Network

In a ***completely-connected network***, each node has a direct communication link to every other node in the network. Figure 2.14(a) illustrates a completely-connected network of eight nodes. This network is ideal in the sense that a node can send a message to another node in a single step, since a communication link exists between them. Completely-connected networks are the static counterparts of crossbar switching networks, since in both networks, the communication between any input/output pair does not block communication between any other pair.

Figure 2.14. (a) A completely-connected network of eight nodes; (b) a star connected network of nine nodes.

Star-Connected Network

In a ***star-connected network***, one processor acts as the central processor. Every other processor has a communication link connecting it to this processor. Figure 2.14(b) shows a star connected network of nine processors. The star-connected network is similar to bus-based networks. Communication between any pair of processors is routed through the central processor, just as the shared bus forms the medium for all communication in a bus-based network. The central processor is the bottleneck in the star topology.

Linear Arrays, Meshes, and *k-d* Meshes

Due to the large number of links in completely connected networks, sparser networks are typically used to build parallel computers. A family of such networks spans the space of linear arrays and hypercubes. A linear array is a static network in which each node (except the two nodes at the ends) has two neighbors, one each to its left and right. A simple extension of the linear array (Figure 2.15(a)) is the ring or a 1-D torus (Figure 2.15(b)). The ring has a wraparound connection between the extremities of the linear array. In this case, each node has two neighbors.

Figure 2.15. Linear arrays: (a) with no wraparound links; (b) with wraparound link.

A two-dimensional mesh illustrated in Figure 2.16(a) is an extension of the linear array to two dimensions. Each dimension has nodes with a node identified by a two-tuple (*i*, *j*). Every node (except those on the periphery) is connected to four other nodes whose indices differ in any dimension by one. A 2-D mesh has the property that it can be laid out in 2-D space, making it attractive from a wiring standpoint. Furthermore, a variety of regularly structured computations map very naturally to a 2-D mesh. For this reason, 2-D meshes were often used as interconnects in parallel machines. Two dimensional meshes can be augmented with wraparound links to form two dimensional tori illustrated in Figure 2.16(b). The three dimensional cube is a generalization of the 2-D mesh to three dimensions, as illustrated in Figure 2.16(c). Each node element in a 3-D cube, with the exception of those on the periphery, is connected to six other nodes, two along each of the three dimensions. A variety of physical simulations commonly executed on parallel computers (for example, 3-D weather modeling, structural modeling, etc.) can be mapped naturally to 3-D network topologies. For this reason, 3-D cubes are used commonly in interconnection networks for parallel computers (for example, in the Cray T3E).

Figure 2.16. Two and three dimensional meshes: (a) 2-D mesh with no wraparound;

(b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

The general class of *k*-*d* meshes refers to the class of topologies consisting of *d* dimensions with *k* nodes along each dimension. Just as a linear array forms one extreme of the *k*-*d* mesh family, the other extreme is formed by an interesting topology called the hypercube. The hypercube topology has two nodes along each dimension and log *p* dimensions. The construction of a hypercube is illustrated in Figure 2.17. A zero-dimensional hypercube consists of 2^0, i.e., one node. A one-dimensional hypercube is constructed from two zero-dimensional hypercubes by connecting them. A two-dimensional hypercube of four nodes is constructed from two one dimensional hypercubes by connecting corresponding nodes. In general a *d*-dimensional hypercube is constructed by connecting corresponding nodes of two (*d* - 1) dimensional hypercubes. Figure 2.17 illustrates this for up to 16 nodes in a 4-D hypercube.

Figure 2.17. Construction of hypercubes from hypercubes of lower dimension.

It is useful to derive a numbering scheme for nodes in a hypercube. A simple numbering scheme can be derived from the construction of a hypercube. As illustrated in Figure 2.17, if we have a numbering of two subcubes of $p/2$ nodes, we can derive a numbering scheme for the cube of p nodes by prefixing the labels of one of the subcubes with a "0" and the labels of the other subcube with a "1". This numbering scheme has the useful property that the minimum distance between two nodes is given by the number of bits that are different in the two labels. For example, nodes labeled 0110 and 0101 are two links apart, since they differ at two bit positions. This property is useful for deriving a number of parallel algorithms for the hypercube architecture.

Tree-Based Networks

A ***tree network*** is one in which there is only one path between any pair of nodes. Both linear arrays and star-connected networks are special cases of tree networks. Figure 2.18 shows networks based on complete binary trees. Static tree networks have a processing element at each node of the tree (Figure 2.18(a)). Tree networks also have a dynamic counterpart. In a dynamic tree network, nodes at intermediate levels are switching nodes and the leaf nodes are processing elements (Figure 2.18(b)).

Figure 2.18. Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

To route a message in a tree, the source node sends the message up the tree until it reaches the node at the root of the smallest subtree containing both the source and destination nodes. Then the message is routed down the tree towards the destination node.

Tree networks suffer from a communication bottleneck at higher levels of the tree. For example, when many nodes in the left subtree of a node communicate with nodes in the right subtree, the root node must handle all the messages. This problem can be alleviated in dynamic tree networks by increasing the number of communication links and switching nodes closer to the root. This network, also called a ***fat tree***, is illustrated in Figure 2.19.

Figure 2.19. A fat tree network of 16 processing nodes.

2.4.4 Evaluating Static Interconnection Networks

We now discuss various criteria used to characterize the cost and performance of static interconnection networks. We use these criteria to evaluate static networks introduced in the previous subsection.

Diameter The ***diameter*** of a network is the maximum distance between any two processing nodes in the network. The distance between two processing nodes is defined as the shortest path (in terms of number of links) between them. The diameter of a completely-connected network is one, and that of a star-connected network is two. The diameter of a ring network is . The diameter of a two-dimensional mesh without wraparound connections is for the two nodes at diagonally opposed corners, and that of a wraparound mesh is . The diameter of a hypercube-connected network is log p since two node labels can differ in at most log *p* positions. The diameter of a complete binary tree is 2 log((*p* + 1)/2) because the two communicating nodes may be in separate subtrees of the root node, and a message might have to travel all the way to the root and then down the other subtree.

Connectivity The ***connectivity*** of a network is a measure of the multiplicity of paths between any two processing nodes. A network with high connectivity is desirable, because it lowers contention for communication resources. One measure of connectivity is the minimum number of arcs that must be removed from the network to break it into two disconnected networks. This is called the ***arc connectivity*** of the network. The arc connectivity is one for linear arrays, as well as tree and star networks. It is two for rings and 2-D meshes without wraparound, four for 2-D wraparound meshes, and *d* for *d*-dimensional hypercubes.

Bisection Width and Bisection Bandwidth The ***bisection width*** of a network is defined as the minimum number of communication links that must be removed to partition the network into two equal halves. The bisection width of a ring is two, since any partition cuts across only two communication links. Similarly, the bisection width of a two-dimensional *p*-node mesh without wraparound connections is and with wraparound connections is . The bisection width of a tree and a star is one, and that of a completely-connected network of *p* nodes is $p^2/4$. The bisection width of a hypercube can be derived from its construction. We construct a *d* dimensional hypercube by connecting corresponding links of two (*d* - 1)-dimensional hypercubes. Since each of these subcubes contains $2^{(d-1)}$ or *p*/2 nodes, at least *p*/2 communication links must cross any partition of a hypercube into two subcubes (Problem 2.15).

The number of bits that can be communicated simultaneously over a link connecting two nodes is called the ***channel width***. Channel width is equal to the number of physical wires in each communication link. The peak rate at which a single physical wire can deliver bits is called the ***channel rate***. The peak rate at which data can be communicated between the ends of a communication link is called ***channel bandwidth***. Channel bandwidth is the product of channel rate and channel width.

Table 2.1. A summary of the characteristics of various static network topologies connecting *p* nodes.

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p - 1$	$p(p - 1)/2$
Star	2	1	1	$p - 1$
Complete binary tree	2 log((*p* + 1)/2)	1	1	$p - 1$
Linear array	$p - 1$	1	1	$p - 1$

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
2-D mesh, no wraparound			2	
2-D wraparound mesh			4	2p
Hypercube	log p	$p/2$	log$p$	(p log p)/2
Wraparound k-ary d cube		$2k^{d-1}$	$2d$	dp

The ***bisection bandwidth*** of a network is defined as the minimum volume of communication allowed between any two halves of the network. It is the product of the bisection width and the channel bandwidth. Bisection bandwidth of a network is also sometimes referred to as ***cross section bandwidth***.

Cost Many criteria can be used to evaluate the cost of a network. One way of defining the cost of a network is in terms of the number of communication links or the number of wires required by the network. Linear arrays and trees use only p - 1 links to connect p nodes. A d-dimensional wraparound mesh has dp links. A hypercube-connected network has (p log p)/2 links.

The bisection bandwidth of a network can also be used as a measure of its cost, as it provides a lower bound on the area in a two-dimensional packaging or the volume in a three-dimensional packaging. If the bisection width of a network is w, the lower bound on the area in a two dimensional packaging is (w^2), and the lower bound on the volume in a three-dimensional packaging is ($w^{3/2}$). According to this criterion, hypercubes and completely connected networks are more expensive than the other networks.

We summarize the characteristics of various static networks in Table 2.1, which highlights the various cost-performance tradeoffs.

2.4.5 Evaluating Dynamic Interconnection Networks

A number of evaluation metrics for dynamic networks follow from the corresponding metrics for static networks. Since a message traversing a switch must pay an overhead, it is logical to think of each switch as a node in the network, in addition to the processing nodes. The diameter of the network can now be defined as the maximum distance between any two nodes in the network. This is indicative of the maximum delay that a message will encounter in being communicated between the selected pair of nodes. In reality, we would like the metric to be the maximum distance between any two processing nodes; however, for all networks of interest, this is equivalent to the maximum distance between any (processing or switching) pair of nodes.

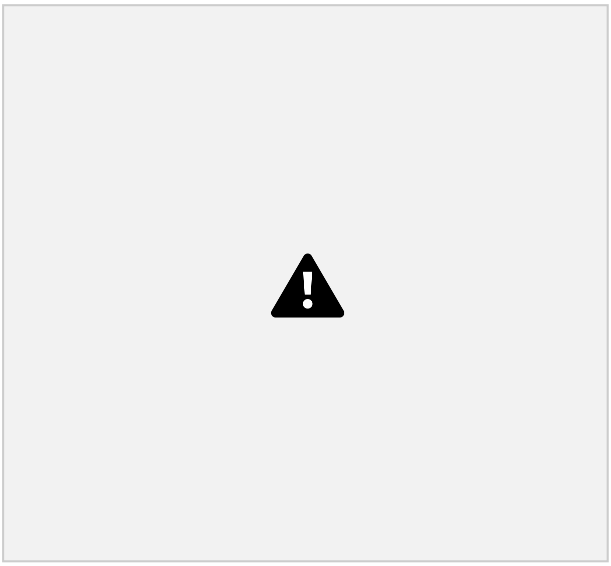
The connectivity of a dynamic network can be defined in terms of node or edge connectivity. The node connectivity is the minimum number of nodes that must fail (be removed from the network) to fragment the network into two parts. As before, we should consider only switching nodes (as opposed to all nodes). However, considering all nodes gives a good approximation to the multiplicity of paths in a dynamic network. The arc connectivity of the network can be similarly defined as the minimum number of edges that must fail (be removed from the network) to fragment the network into two unreachable parts.

The bisection width of a dynamic network must be defined more precisely than diameter and connectivity. In the case of bisection width, we consider any possible partitioning of the p processing nodes into two equal parts. Note that this does not restrict the partitioning of the switching nodes. For each such partition, we select an induced partitioning of the switching nodes such that the number of edges crossing this partition is minimized. The minimum number of edges for any such partition is the bisection width of the dynamic network. Another intuitive way of thinking of bisection width is in terms of the minimum number of edges that must be removed from the network so as to partition the network into two halves with identical number of processing nodes. We illustrate this concept further in the following example:

Example 2.13 Bisection width of dynamic networks

Consider the network illustrated in Figure 2.20. We illustrate here three bisections, A, B, and C, each of which partitions the network into two groups of two processing nodes each. Notice that these partitions need not partition the network nodes equally. In the example, each partition results in an edge cut of four. We conclude that the bisection width of this graph is four.

Figure 2.20. Bisection width of a dynamic network is computed by examining various equi-partitions of the processing nodes and selecting the minimum number of edges crossing the partition. In this case, each partition yields an edge cut of four. Therefore, the bisection width of this graph is four.



The cost of a dynamic network is determined by the link cost, as is the case with static networks, as well as the switch cost. In typical dynamic networks, the degree of a switch is constant. Therefore, the number of links and switches is asymptotically identical. Furthermore, in typical networks, switch cost exceeds link cost. For this reason, the cost of dynamic networks is often determined by the number of switching nodes in the network.

We summarize the characteristics of various dynamic networks in Table 2.2.

2.4.6 Cache Coherence in Multiprocessor Systems

While interconnection networks provide basic mechanisms for communicating messages (data), in the case of shared-address-space computers additional hardware is required to keep multiple copies of data consistent with each other. Specifically, if there exist two copies of the data (in different caches/memory elements), how do we ensure that different processors operate on these in a manner that follows predefined semantics?

Table 2.2. A summary of the characteristics of various dynamic network

topologies connecting *p* processing nodes.

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Crossbar	1	*p*	1	p^2
Omega Network	log *p*	*p*/2	2	*p*/2
Dynamic Tree	2 log *p*	1	2	*p* - 1

The problem of keeping caches in multiprocessor systems coherent is significantly more complex than in uniprocessor systems. This is because in addition to multiple copies as in uniprocessor systems, there may also be multiple processors modifying these copies. Consider a simple scenario illustrated in Figure 2.21. Two processors P_0 and P_1 are connected over a shared bus to a globally accessible memory. Both processors load the same variable. There are now three copies of the variable. The coherence mechanism must now ensure that all operations performed on these copies are serializable (i.e., there exists some serial order of instruction execution that corresponds to the parallel schedule). When a processor changes the value of its copy of the variable, one of two things must happen: the other copies must be invalidated, or the other copies must be updated. Failing this, other processors may potentially work with incorrect (stale) values of the variable. These two protocols are referred to as ***invalidate*** and ***update*** protocols and are illustrated in Figure 2.21(a) and (b).

Figure 2.21. Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.

In an update protocol, whenever a data item is written, all of its copies in the system are updated. For this reason, if a processor simply reads a data item once and never uses it, subsequent updates to this item at other processors cause excess overhead in terms of latency at source and bandwidth on the network. On the other hand, in this situation, an invalidate protocol invalidates the data item on the first update at a remote processor and subsequent updates need not be performed on this copy.

Another important factor affecting the performance of these protocols is ***false sharing***. False sharing refers to the situation in which different processors update different parts of of the same cache-line. Thus, although the updates are not performed on shared variables, the system does not detect this. In an invalidate protocol, when a processor updates its part of the cache-line, the other copies of this line are invalidated. When other processors try to update their parts of the cache-line, the line must actually be fetched from the remote processor. It is easy to see that false-sharing can cause a cache-line to be ping-ponged between various processors. In an update protocol, this situation is slightly better since all reads can be performed locally and the writes must be updated. This saves an invalidate operation that is otherwise wasted.

The tradeoff between invalidate and update schemes is the classic tradeoff between communication overhead (updates) and idling (stalling in invalidates). Current generation cache coherent machines typically rely on invalidate protocols. The rest of our discussion of multiprocessor cache systems therefore assumes invalidate protocols.

Maintaining Coherence Using Invalidate Protocols Multiple copies of a single data item are kept consistent by keeping track of the number of copies and the state of each of these copies. We discuss here one possible set of states associated with data items and events that trigger transitions among these states. Note that this set of states and transitions is not unique. It is possible to define other states and associated transitions as well.

Let us revisit the example in Figure 2.21. Initially the variable *x* resides in the global memory. The first step executed by both processors is a load operation on this variable. At this point, the state of the variable is said to be ***shared***, since it is shared by multiple processors. When processor P_0 executes a store on this variable, it marks all other copies of this variable as ***invalid***. It must also mark its own copy as modified or ***dirty***. This is done to ensure that all subsequent accesses to this variable at other processors will be serviced by processor P_0 and not from the memory. At this point, say, processor P_1 executes another load operation on *x* . Processor P_1 attempts to fetch this variable and, since the variable was marked dirty by processor P_0, processor P_0 services the request. Copies of this variable at processor P_1 and the global memory are updated and the variable re-enters the shared state. Thus, in this simple model, there are three states - ***shared***, ***invalid***, and ***dirty*** - that a cache line goes through.

The complete state diagram of a simple three-state protocol is illustrated in Figure 2.22. The solid lines depict processor actions and the dashed lines coherence actions. For example, when a processor executes a read on an invalid block, the block is fetched and a transition is made from invalid to shared. Similarly, if a processor does a write on a shared block, the coherence protocol propagates a C_write (a coherence write) on the block. This triggers a transition from shared to invalid at all the other blocks.

Figure 2.22. State diagram of a simple three-state coherence protocol.

Example 2.14 Maintaining coherence using a simple three-state protocol

Consider an example of two program segments being executed by processor P_0 and P_1

as illustrated in Figure 2.23. The system consists of local memories (or caches) at processors P_0 and P_1, and a global memory. The three-state protocol assumed in this example corresponds to the state diagram illustrated in Figure 2.22. Cache lines in this system can be either shared, invalid, or dirty. Each data item (variable) is assumed to be on a different cache line. Initially, the two variables *x* and *y* are tagged dirty and the only copies of these variables exist in the global memory. Figure 2.23 illustrates state transitions along with values of copies of the variables with each instruction execution.

Figure 2.23. Example of parallel program execution with the simple three-state coherence protocol discussed in Section 2.4.6.

The implementation of coherence protocols can be carried out using a variety of hardware mechanisms – snoopy systems, directory based systems, or combinations thereof.

Snoopy Cache Systems

Snoopy caches are typically associated with multiprocessor systems based on broadcast interconnection networks such as a bus or a ring. In such systems, all processors snoop on (monitor) the bus for transactions. This allows the processor to make state transitions for its cache-blocks. Figure 2.24 illustrates a typical snoopy bus based system. Each processor's cache has a set of tag bits associated with it that determine the state of the cache blocks. These tags are updated according to the state diagram associated with the coherence protocol. For instance, when the snoop hardware detects that a read has been issued to a cache block that it has a dirty copy of, it asserts control of the bus and puts the data out. Similarly, when the snoop hardware detects that a write operation has been issued on a cache block that it has a copy of, it invalidates the block. Other state transitions are made in this fashion locally.

Figure 2.24. A simple snoopy bus based cache coherence system.

Performance of Snoopy Caches Snoopy protocols have been extensively studied and used in commercial systems. This is largely because of their simplicity and the fact that existing bus based systems can be upgraded to accommodate snoopy protocols. The performance gains of snoopy systems are derived from the fact that if different processors operate on different data items, these items can be cached. Once these items are tagged dirty, all subsequent operations can be performed locally on the cache without generating external traffic. Similarly, if a data item is read by a number of processors, it transitions to the shared state in the cache and all subsequent read operations become local. In both cases, the coherence protocol does not add any overhead. On the other hand, if multiple processors read and update the same data item, they generate coherence functions across processors. Since a shared bus has a finite bandwidth, only a constant number of such coherence operations can execute in unit time. This presents a fundamental bottleneck for snoopy bus based systems.

Snoopy protocols are intimately tied to multicomputers based on broadcast networks such as buses. This is because all processors must snoop all the messages. Clearly, broadcasting all of a processor's memory operations to all the processors is not a scalable solution. An obvious solution to this problem is to propagate coherence operations only to those processors that must participate in the operation (i.e., processors that have relevant copies of the data). This solution requires us to keep track of which processors have copies of various data items and also the relevant state information for these data items. This information is stored in a directory, and the coherence mechanism based on such information is called a directory-based system.

Directory Based Systems

Consider a simple system in which the global memory is augmented with a directory that maintains a bitmap representing cache-blocks and the processors at which they are cached (Figure 2.25). These bitmap entries are sometimes referred to as the ***presence bits***. As before, we assume a three-state protocol with the states labeled ***invalid***, ***dirty***, and ***shared***. The key to the performance of directory based schemes is the simple observation that only processors that hold a particular block (or are reading it) participate in the state transitions due to coherence operations. Note that there may be other state transitions triggered by processor read, write, or flush (retiring a line from cache) but these transitions can be handled locally with the operation reflected in the presence bits and state in the directory.

Figure 2.25. Architecture of typical directory based systems: (a) a centralized directory; and (b) a distributed directory.

Revisiting the code segment in Figure 2.21, when processors P_0 and P_1 access the block corresponding to variable x, the state of the block is changed to shared, and the presence bits updated to indicate that processors P_0 and P_1 share the block. When P_0 executes a store on the variable, the state in the directory is changed to dirty and the presence bit of P_1 is reset. All subsequent operations on this variable performed at processor P_0 can proceed locally. If another processor reads the value, the directory notices the dirty tag and uses the presence bits to direct the request to the appropriate processor. Processor P_0 updates the block in the memory, and sends it to the requesting processor. The presence bits are modified to reflect this and the state transitions to shared.

Performance of Directory Based Schemes As is the case with snoopy protocols, if different processors operate on distinct data blocks, these blocks become dirty in the respective caches and all operations after the first one can be performed locally. Furthermore, if multiple processors read (but do not update) a single data block, the data block gets replicated in the caches in the shared state and subsequent reads can happen without triggering any coherence overheads.

Coherence actions are initiated when multiple processors attempt to update the same data item. In this case, in addition to the necessary data movement, coherence operations add to the overhead in the form of propagation of state updates (invalidates or updates) and generation of state information from the directory. The former takes the form of communication overhead and the latter adds contention. The communication overhead is a function of the number of processors requiring state updates and the algorithm for propagating state information. The contention overhead is more fundamental in nature. Since the directory is in memory and the memory system can only service a bounded number of read/write operations in unit time, the number of state updates is ultimately bounded by the directory. If a parallel program requires a large number of coherence actions (large number of read/write shared data blocks) the directory will ultimately bound its parallel performance.

Finally, from the point of view of cost, the amount of memory required to store the directory may itself become a bottleneck as the number of processors increases. Recall that the directory size grows as $O(mp)$, where m is the number of memory blocks and p the number of processors. One solution would be to make the memory block larger (thus reducing m for a given memory size). However, this adds to other overheads such as false sharing, where two processors update distinct data items in a program but the data items happen to lie in the same memory block. This phenomenon is discussed in greater detail in Chapter 7.

Since the directory forms a central point of contention, it is natural to break up the task of maintaining coherence across multiple processors. The basic principle is to let each processor maintain coherence of its own memory blocks, assuming a physical (or logical) partitioning of the memory blocks across processors. This is the principle of a distributed directory system.

Distributed Directory Schemes In scalable architectures, memory is physically distributed across processors. The corresponding presence bits of the blocks are also distributed. Each processor is responsible for maintaining the coherence of its own memory blocks. The architecture of such a system is illustrated in Figure 2.25(b). Since each memory block has an owner (which can typically be computed from the block address), its directory location is implicitly known to all processors. When a processor attempts to read a block for the first time, it requests the owner for the block. The owner suitably directs this request based on presence and state information locally available. Similarly, when a processor writes into a memory block, it propagates an invalidate to the owner, which in turn forwards the invalidate to all processors that have a cached copy of the block. In this way, the directory is decentralized and the contention associated with the central directory is alleviated. Note that the communication overhead associated with state update messages is not reduced.

Performance of Distributed Directory Schemes As is evident, distributed directories permit $O(p)$ simultaneous coherence operations, provided the underlying network can sustain the associated state update messages. From this point of view, distributed directories are inherently more scalable than snoopy systems or centralized directory systems. The latency and bandwidth of the network become fundamental performance bottlenecks for such systems.

[Team LiB]

[Team LiB]

2.5 Communication Costs in Parallel Machines

One of the major overheads in the execution of parallel programs arises from communication of information between processing elements. The cost of communication is dependent on a variety of features including the programming model semantics, the network topology, data handling and routing, and associated software protocols. These issues form the focus of our discussion here.

2.5.1 Message Passing Costs in Parallel Computers

The time taken to communicate a message between two nodes in a network is the sum of the time to prepare a message for transmission and the time taken by the message to traverse the network to its destination. The principal parameters that determine the communication latency are as follows:

1. ***Startup time*** (t_s): The startup time is the time required to handle a message at the sending and receiving nodes. This includes the time to prepare the message (adding header, trailer, and error correction information), the time to execute the routing algorithm, and the time to establish an interface between the local node and the router. This delay is incurred only once for a single message transfer.

2. ***Per-hop time*** (t_h): After a message leaves a node, it takes a finite amount of time to reach the next node in its path. The time taken by the header of a message to travel between two directly-connected nodes in the network is called the per-hop time. It is also known as ***node latency***. The per-hop time is directly related to the latency within the routing switch for determining which output buffer or channel the message should be forwarded to.

3. ***Per-word transfer time*** (t_w): If the channel bandwidth is r words per second, then each word takes time $t_w = 1/r$ to traverse the link. This time is called the per-word transfer time. This time includes network as well as buffering overheads.

We now discuss two routing techniques that have been used in parallel computers – store-and forward routing and cut-through routing.

Store-and-Forward Routing

In store-and-forward routing, when a message is traversing a path with multiple links, each intermediate node on the path forwards the message to the next node after it has received and stored the entire message. Figure 2.26(a) shows the communication of a message through a store-and-forward network.

Figure 2.26. Passing a message from node P_0 to P_3 (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.

Suppose that a message of size m is being transmitted through such a network. Assume that it traverses *l* links. At each link, the message incurs a cost t_h for the header and $t_w m$ for the rest of the message to traverse the link. Since there are *l* such links, the total time is $(t_h + t_w m)l$. Therefore, for store-and-forward routing, the total communication cost for a message of size *m* words to traverse *l* communication links is

Equation 2.2

In current parallel computers, the per-hop time t_h is quite small. For most parallel algorithms, it is less than $t_w m$ even for small values of *m* and thus can be ignored. For parallel platforms using store-and-forward routing, the time given by Equation 2.2 can be simplified to

Packet Routing

Store-and-forward routing makes poor use of communication resources. A message is sent from one node to the next only after the entire message has been received (Figure 2.26(a)). Consider the scenario shown in Figure 2.26(b), in which the original message is broken into two equal sized parts before it is sent. In this case, an intermediate node waits for only half of the original message to arrive before passing it on. The increased utilization of communication resources and reduced communication time is apparent from Figure 2.26(b). Figure 2.26(c) goes a step further and breaks the message into four parts. In addition to better utilization of communication resources, this principle offers other advantages – lower overhead from packet loss (errors), possibility of packets taking different paths, and better error correction capability. For these reasons, this technique is the basis for long-haul communication networks such as the Internet, where error rates, number of hops, and variation in network state can be higher. Of course, the overhead here is that each packet must carry routing, error correction, and sequencing information.

Consider the transfer of an *m* word message through the network. The time taken for programming the network interfaces and computing the routing information, etc., is independent of the message length. This is aggregated into the startup time t_s of the message transfer. We assume a scenario in which routing tables are static over the time of message transfer (i.e., all packets traverse the same path). While this is not a valid assumption under all circumstances, it serves the purpose of motivating a cost model for message transfer. The message is broken into packets, and packets are assembled with their error, routing, and sequencing fields. The size of a packet is now given by $r + s$, where *r* is the original message and *s* is the additional information carried in the packet. The time for packetizing the message is proportional to the length of the message. We denote this time by mt_{w1}. If the network is capable of communicating one word every t_{w2} seconds, incurs a delay of t_h on each hop, and if the first packet traverses *l* hops, then this packet takes time $t_h l + t_{w2}(r + s)$ to reach the destination. After this time, the destination node receives an additional packet every $t_{w2}(r + s)$ seconds. Since there are $m/r - 1$ additional packets, the total communication time is given by:

where

Packet routing is suited to networks with highly dynamic states and higher error rates, such as local- and wide-area networks. This is because individual packets may take different routes and retransmissions can be localized to lost packets.

Cut-Through Routing

In interconnection networks for parallel computers, additional restrictions can be imposed on message transfers to further reduce the overheads associated with packet switching. By forcing all packets to take the same path, we can eliminate the overhead of transmitting routing information with each packet. By forcing in-sequence delivery, sequencing information can be eliminated. By associating error information at message level rather than packet level, the overhead associated with error detection and correction can be reduced. Finally, since error rates in interconnection networks for parallel machines are extremely low, lean error detection mechanisms can be used instead of expensive error correction schemes.

The routing scheme resulting from these optimizations is called cut-through routing. In cut through routing, a message is broken into fixed size units called ***flow control digits*** or ***flits***. Since flits do not contain the overheads of packets, they can be much smaller than packets. A tracer is first sent from the source to the destination node to establish a connection. Once a connection has been established, the flits are sent one after the other. All flits follow the same path in a dovetailed fashion. An intermediate node does not wait for the entire message to arrive before forwarding it. As soon as a flit is received at an intermediate node, the flit is passed on to the next node. Unlike store-and-forward routing, it is no longer necessary to have buffer space at each intermediate node to store the entire message. Therefore, cut-through routing uses less memory and memory bandwidth at intermediate nodes, and is faster.

Consider a message that is traversing such a network. If the message traverses l links, and t_h is the per-hop time, then the header of the message takes time lt_h to reach the destination. If the message is m words long, then the entire message arrives in time t_wm after the arrival of the header of the message. Therefore, the total communication time for cut-through routing is

Equation 2.3

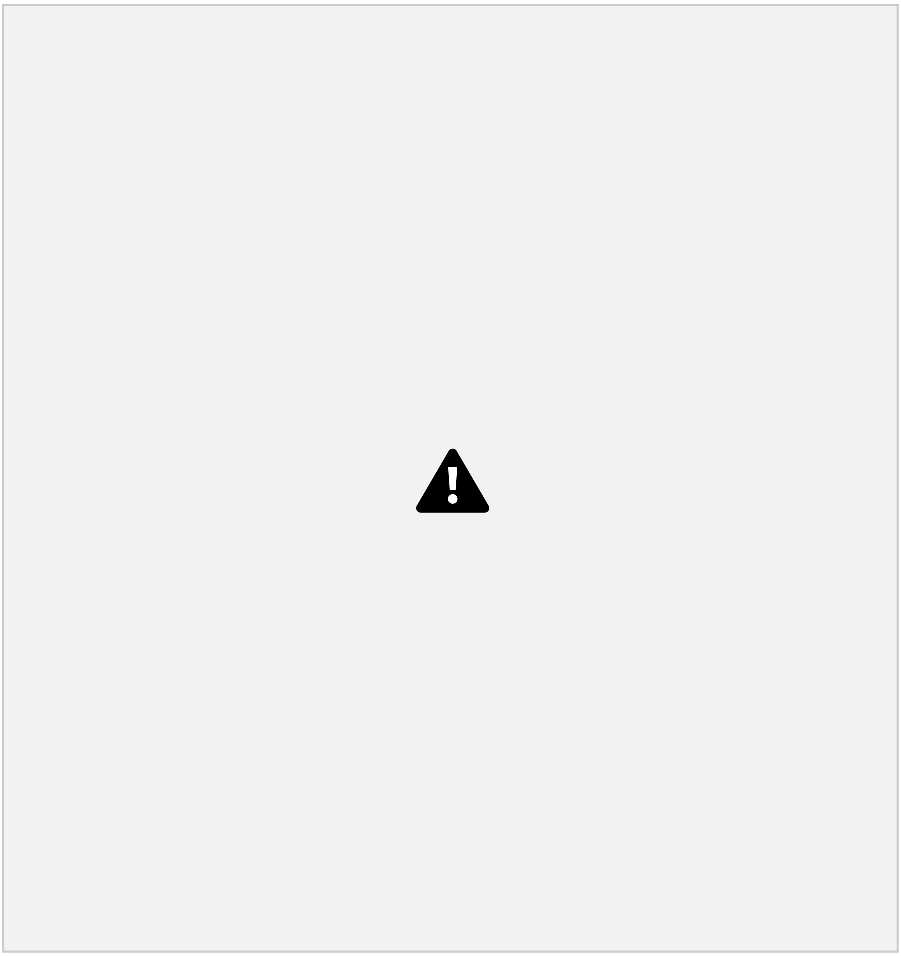
This time is an improvement over store-and-forward routing since terms corresponding to number of hops and number of words are additive as opposed to multiplicative in the former. Note that if the communication is between nearest neighbors (that is, $l = 1$), or if the message size is small, then the communication time is similar for store-and-forward and cut-through routing schemes.

Most current parallel computers and many local area networks support cut-through routing. The size of a flit is determined by a variety of network parameters. The control circuitry must operate at the flit rate. Therefore, if we select a very small flit size, for a given link bandwidth, the required flit rate becomes large. This poses considerable challenges for designing routers as it requires the control circuitry to operate at a very high speed. On the other hand, as flit sizes become large, internal buffer sizes increase, so does the latency of message transfer. Both of these are undesirable. Flit sizes in recent cut-through interconnection networks range from four bits to 32 bytes. In many parallel programming paradigms that rely predominantly on short messages (such as cache lines), the latency of messages is critical. For these, it is unreasonable for a long message traversing a link to hold up a short message. Such scenarios are addressed in routers using multilane cut-through routing. In multilane cut-through routing, a single physical channel is split into a number of virtual channels.

Messaging constants t_s, t_w, and t_h are determined by hardware characteristics, software layers, and messaging semantics. Messaging semantics associated with paradigms such as message passing are best served by variable length messages, others by fixed length short messages. While effective bandwidth may be critical for the former, reducing latency is more important for the latter. Messaging layers for these paradigms are tuned to reflect these requirements.

While traversing the network, if a message needs to use a link that is currently in use, then the message is blocked. This may lead to deadlock. Figure 2.27 illustrates deadlock in a cut-through routing network. The destinations of messages 0, 1, 2, and 3 are *A*, *B*, *C*, and *D*, respectively. A flit from message 0 occupies the link *CB* (and the associated buffers). However, since link *BA* is occupied by a flit from message 3, the flit from message 0 is blocked. Similarly, the flit from message 3 is blocked since link *AD* is in use. We can see that no messages can progress in the network and the network is deadlocked. Deadlocks can be avoided in cut-through networks by using appropriate routing techniques and message buffers. These are discussed in Section 2.6.

Figure 2.27. An example of deadlock in a cut-through routing network.



A Simplified Cost Model for Communicating Messages

As we have just seen in Section 2.5.1, the cost of communicating a message between two nodes *l* hops away using cut-through routing is given by

This equation implies that in order to optimize the cost of message transfers, we would need to:

1. **Communicate in bulk.** That is, instead of sending small messages and paying a startup cost t_s for each, we want to aggregate small messages into a single large message and amortize the startup latency across a larger message. This is because on typical platforms such as clusters and message-passing machines, the value of t_s is much larger than those of t_h or t_w.
2. **Minimize the volume of data.** To minimize the overhead paid in terms of per-word transfer time t_w, it is desirable to reduce the volume of data communicated as much as

3.
2.

possible.

3. **Minimize distance of data transfer.** Minimize the number of hops l that a message must traverse.

While the first two objectives are relatively easy to achieve, the task of minimizing distance of communicating nodes is difficult, and in many cases an unnecessary burden on the algorithm designer. This is a direct consequence of the following characteristics of parallel platforms and paradigms:

In many message-passing libraries such as MPI, the programmer has little control on the mapping of processes onto physical processors. In such paradigms, while tasks might have well defined topologies and may communicate only among neighbors in the task topology, the mapping of processes to nodes might destroy this structure.

Many architectures rely on randomized (two-step) routing, in which a message is first sent to a random node from source and from this intermediate node to the destination. This alleviates hot-spots and contention on the network. Minimizing number of hops in a randomized routing network yields no benefits.

The per-hop time (t_h) is typically dominated either by the startup latency (t_s)for small messages or by per-word component ($t_w m$) for large messages. Since the maximum number of hops (l) in most networks is relatively small, the per-hop time can be ignored with little loss in accuracy.

All of these point to a simpler cost model in which the cost of transferring a message between two nodes on a network is given by:

Equation 2.4

This expression has significant implications for architecture-independent algorithm design as well as for the accuracy of runtime predictions. Since this cost model implies that it takes the same amount of time to communicate between any pair of nodes, it corresponds to a completely connected network. Instead of designing algorithms for each specific architecture (for example, a mesh, hypercube, or tree), we can design algorithms with this cost model in mind and port it to any target parallel computer.

This raises the important issue of loss of accuracy (or fidelity) of prediction when the algorithm is ported from our simplified model (which assumes a completely connected network) to an actual machine architecture. If our initial assumption that the t_h term is typically dominated by the t_s or t_w terms is valid, then the loss in accuracy should be minimal.

However, it is important to note that our basic cost model is valid only for uncongested networks. Architectures have varying thresholds for when they get congested; i.e., a linear array has a much lower threshold for congestion than a hypercube. Furthermore, different communication patterns congest a given network to different extents. Consequently, our simplified cost model is valid only as long as the underlying communication pattern does not congest the network.

Example 2.15 Effect of congestion on communication cost

Consider a mesh in which each node is only communicating with its nearest neighbor. Since no links in the network are used for more than one communication, the time for this operation is $t_s + t_w m$, where m is the number of words communicated. This time is consistent with our simplified model.

Consider an alternate scenario in which each node is communicating with a randomly selected node. This randomness implies that there are $p/2$ communications (or $p/4$ bi directional communications) occurring across any equi-partition of the machine (since the node being communicated with could be in either half with equal probability). From our discussion of bisection width, we know that a 2-D mesh has a bisection width of . From these two, we can infer that some links would now have to carry at least messages, assuming bi-directional communication channels. These messages must be serialized over the link. If each message is of size m, the time for this operation is at least . This time is not in conformity with our simplified model.

The above example illustrates that for a given architecture, some communication patterns can be non-congesting and others may be congesting. This makes the task of modeling communication costs dependent not just on the architecture, but also on the communication pattern. To address this, we introduce the notion of ***effective bandwidth***. For communication patterns that do not congest the network, the effective bandwidth is identical to the link bandwidth. However, for communication operations that congest the network, the effective bandwidth is the link bandwidth scaled down by the degree of congestion on the most congested link. This is often difficult to estimate since it is a function of process to node mapping, routing algorithms, and communication schedule. Therefore, we use a lower bound on the message communication time. The associated link bandwidth is scaled down by a factor p/b, where b is the bisection width of the network.

In the rest of this text, we will work with the simplified communication model for message passing with effective per-word time t_w because it allows us to design algorithms in an architecture-independent manner. We will also make specific notes on when a communication operation within an algorithm congests the network and how its impact is factored into parallel runtime. The communication times in the book apply to the general class of *k-d* meshes. While these times may be realizable on other architectures as well, this is a function of the underlying architecture.

2.5.2 Communication Costs in Shared-Address-Space Machines

The primary goal of associating communication costs with parallel programs is to associate a figure of merit with a program to guide program development. This task is much more difficult for cache-coherent shared-address-space machines than for message-passing or non-cache coherent architectures. The reasons for this are as follows:

- Memory layout is typically determined by the system. The programmer has minimal control on the location of specific data items over and above permuting data structures to optimize access. This is particularly important in distributed memory shared-address space architectures because it is difficult to identify local and remote accesses. If the access times for local and remote data items are significantly different, then the cost of communication can vary greatly depending on the data layout.
- Finite cache sizes can result in cache thrashing. Consider a scenario in which a node needs a certain fraction of the total data to compute its results. If this fraction is smaller than locally available cache, the data can be fetched on first access and computed on. However, if the fraction exceeds available cache, then certain portions of this data might get overwritten, and consequently accessed several times. This overhead can cause sharp degradation in program performance as the problem size is increased. To remedy this, the programmer must alter execution schedules (e.g., blocking loops as illustrated in serial matrix multiplication in Problem 2.5) for minimizing working set size. While this problem is common to both serial and multiprocessor platforms, the penalty is much higher in the case of multiprocessors since each miss might now involve coherence operations and interprocessor communication.

- Overheads associated with invalidate and update operations are difficult to quantify. After a data item has been fetched by a processor into cache, it may be subject to a variety of operations at another processor. For example, in an invalidate protocol, the cache line might be invalidated by a write operation at a remote processor. In this case, the next read operation on the data item must pay a remote access latency cost again. Similarly, the overhead associated with an update protocol might vary significantly depending on the number of copies of a data item. The number of concurrent copies of a data item and the schedule of instruction execution are typically beyond the control of the programmer.

- Spatial locality is difficult to model. Since cache lines are generally longer than one word (anywhere from four to 128 words), different words might have different access latencies associated with them even for the first access. Accessing a neighbor of a previously fetched word might be extremely fast, if the cache line has not yet been overwritten. Once again, the programmer has minimal control over this, other than to permute data structures to maximize spatial locality of data reference.

- Prefetching can play a role in reducing the overhead associated with data access. Compilers can advance loads and, if sufficient resources exist, the overhead associated with these loads may be completely masked. Since this is a function of the compiler, the underlying program, and availability of resources (registers/cache), it is very difficult to model accurately.

- False sharing is often an important overhead in many programs. Two words used by (threads executing on) different processor may reside on the same cache line. This may cause coherence actions and communication overheads, even though none of the data might be shared. The programmer must adequately pad data structures used by various processors to minimize false sharing.

- Contention in shared accesses is often a major contributing overhead in shared address space machines. Unfortunately, contention is a function of execution schedule and consequently very difficult to model accurately (independent of the scheduling algorithm). While it is possible to get loose asymptotic estimates by counting the number of shared accesses, such a bound is often not very meaningful.

Any cost model for shared-address-space machines must account for all of these overheads. Building these into a single cost model results in a model that is too cumbersome to design programs for and too specific to individual machines to be generally applicable.

As a first-order model, it is easy to see that accessing a remote word results in a cache line being fetched into the local cache. The time associated with this includes the coherence overheads, network overheads, and memory overheads. The coherence and network overheads are functions of the underlying interconnect (since a coherence operation must be potentially propagated to remote processors and the data item must be fetched). In the absence of knowledge of what coherence operations are associated with a specific access and where the word is coming from, we associate a constant overhead to accessing a cache line of the shared data. For the sake of uniformity with the message-passing model, we refer to this cost as t_s. Because of various latency-hiding protocols, such as prefetching, implemented in modern processor architectures, we assume that a constant cost of t_s is associated with initiating access to a contiguous chunk of m words of shared data, even if m is greater than the cache line size. We further assume that accessing shared data is costlier than accessing local data (for instance, on a NUMA machine, local data is likely to reside in a local memory module, while data shared by p processors will need to be fetched from a nonlocal module for at least p - 1 processors). Therefore, we assign a per-word access cost of t_w to shared data.

From the above discussion, it follows that we can use the same expression $t_s + t_w m$ to account for the cost of sharing a single chunk of m words between a pair of processors in both shared memory and message-passing paradigms (Equation 2.4) with the difference that the value of the constant t_s relative to t_w is likely to be much smaller on a shared-memory machine than on a distributed memory machine (t_w is likely to be near zero for a UMA machine). Note that the cost $t_s + t_w m$ assumes read-only access without contention. If multiple processes access the same data, then the cost is multiplied by the number of processes, just as in the message passing where the process that owns the data will need to send a message to each receiving process. If the access is read-write, then the cost will be incurred again for subsequent access by processors other than the one writing. Once again, there is an equivalence with the message-passing model. If a process modifies the contents of a message that it receives, then it must send it back to processes that subsequently need access to the refreshed data. While this model seems overly simplified in the context of shared-address-space machines, we note that the model provides a good estimate of the cost of sharing an array of m words between a pair of processors.

The simplified model presented above accounts primarily for remote data access but does not model a variety of other overheads. Contention for shared data access must be explicitly accounted for by counting the number of accesses to shared data between co-scheduled tasks. The model does not explicitly include many of the other overheads. Since different machines have caches of varying sizes, it is difficult to identify the point at which working set size exceeds the cache size resulting in cache thrashing, in an architecture independent manner. For this reason, effects arising from finite caches are ignored in this cost model. Maximizing spatial locality (cache line effects) is not explicitly included in the cost. False sharing is a function of the instruction schedules as well as data layouts. The cost model assumes that shared data structures are suitably padded and, therefore, does not include false sharing costs. Finally, the cost model does not account for overlapping communication and computation. Other models have been proposed to model overlapped communication. However, designing even simple algorithms for these models is cumbersome. The related issue of multiple concurrent computations (threads) on a single processor is not modeled in the expression. Instead, each processor is assumed to execute a single concurrent unit of computation.

[Team LiB]
[Team LiB]

2.6 Routing Mechanisms for Interconnection Networks

Efficient algorithms for routing a message to its destination are critical to the performance of parallel computers. A ***routing mechanism*** determines the path a message takes through the network to get from the source to the destination node. It takes as input a message's source and destination nodes. It may also use information about the state of the network. It returns one or more paths through the network from the source to the destination node.

Routing mechanisms can be classified as ***minimal*** or ***non-minimal***. A minimal routing mechanism always selects one of the shortest paths between the source and the destination. In a minimal routing scheme, each link brings a message closer to its destination, but the scheme can lead to congestion in parts of the network. A non-minimal routing scheme, in contrast, may route the message along a longer path to avoid network congestion.

Routing mechanisms can also be classified on the basis of how they use information regarding the state of the network. A ***deterministic routing*** scheme determines a unique path for a message, based on its source and destination. It does not use any information regarding the state of the network. Deterministic schemes may result in uneven use of the communication resources in a network. In contrast, an ***adaptive routing*** scheme uses information regarding the current state of the network to determine the path of the message. Adaptive routing detects congestion in the network and routes messages around it.

One commonly used deterministic minimal routing technique is called ***dimension-ordered routing***. Dimension-ordered routing assigns successive channels for traversal by a message based on a numbering scheme determined by the dimension of the channel. The dimension ordered routing technique for a two-dimensional mesh is called ***XY-routing*** and that for a hypercube is called ***E-cube routing***.

Consider a two-dimensional mesh without wraparound connections. In the XY-routing scheme, a message is sent first along the X dimension until it reaches the column of the destination node and then along the Y dimension until it reaches its destination. Let $P_{Sy,Sx}$ represent the position of the source node and $P_{Dy,Dx}$ represent that of the destination node. Any minimal routing scheme should return a path of length $|Sx - Dx| + |Sy - Dy|$. Assume that Dx □ Sx and Dy □ Sy. In the XY-routing scheme, the message is passed through intermediate nodes $P_{Sy,Sx+1}$, $P_{Sy,Sx+2}$, ..., $P_{Sy,Dx}$ along the X dimension and then through nodes $P_{Sy+1,Dx}$, $P_{Sy+2,Dx}$, ..., $P_{Dy,Dx}$ along the Y dimension to reach the destination. Note that the length of this path is indeed $|Sx - Dx| + |Sy - Dy|$.

E-cube routing for hypercube-connected networks works similarly. Consider a d-dimensional hypercube of p nodes. Let P_s and P_d be the labels of the source and destination nodes. We know from Section 2.4.3 that the binary representations of these labels are d bits long. Furthermore, the minimum distance between these nodes is given by the number of ones in P_s □ P_d (where □ represents the bitwise exclusive-OR operation). In the E-cube algorithm, node P_s computes P_s □ P_d and sends the message along dimension k, where k is the position of the least significant nonzero bit in P_s □ P_d. At each intermediate step, node P_i, which receives the message, computes P_i □ P_d and forwards the message along the dimension corresponding to the least significant nonzero bit. This process continues until the message reaches its destination. Example 2.16 illustrates E-cube routing in a three-dimensional hypercube network.

Example 2.16 E-cube routing in a hypercube network

Consider the three-dimensional hypercube shown in Figure 2.28. Let P_s = 010 and P_d = 111 represent the source and destination nodes for a message. Node P_s computes

010 111 = 101. In the first step, P_s forwards the message along the dimension corresponding to the least significant bit to node 011. Node 011 sends the message along the dimension corresponding to the most significant bit (011 111 = 100). The message reaches node 111, which is the destination of the message.

Figure 2.28. Routing a message from node P_s (010) to node P_d (111) in a three-dimensional hypercube using E-cube routing.

In the rest of this book we assume deterministic and minimal message routing for analyzing parallel algorithms.

[Team LiB]

[Team LiB]

2.7 Impact of Process-Processor Mapping and Mapping Techniques

As we have discussed in Section 2.5.1, a programmer often does not have control over how logical processes are mapped to physical nodes in a network. For this reason, even communication patterns that are not inherently congesting may congest the network. We illustrate this with the following example:

Example 2.17 Impact of process mapping

Consider the scenario illustrated in Figure 2.29. The underlying architecture is a 16-node mesh with nodes labeled from 1 to 16 (Figure 2.29(a)) and the algorithm has been implemented as 16 processes, labeled 'a' through 'p' (Figure 2.29(b)). The algorithm has been tuned for execution on a mesh in such a way that there are no congesting communication operations. We now consider two mappings of the processes to nodes as illustrated in Figures 2.29(c) and (d). Figure 2.29(c) is an intuitive mapping and is such that a single link in the underlying architecture only carries data corresponding to a single communication channel between processes. Figure 2.29(d), on the other hand, corresponds to a situation in which processes have been mapped randomly to processing nodes. In this case, it is easy to see that each link in the machine carries up to six channels of data between processes. This may potentially result in considerably larger communication times if the required data rates on communication channels between processes is high.

Figure 2.29. Impact of process mapping on performance: (a) underlying architecture; (b) processes and their interactions; (c) an intuitive mapping of processes to nodes; and (d) a random mapping of processes to nodes.

It is evident from the above example that while an algorithm may be fashioned out of non congesting communication operations, the mapping of processes to nodes may in fact induce congestion on the network and cause degradation in performance.

2.7.1 Mapping Techniques for Graphs

While the programmer generally does not have control over process-processor mapping, it is important to understand algorithms for such mappings. This is because these mappings can be used to determine degradation in the performance of an algorithm. Given two graphs, *G*(*V*, *E*) and *G*'(*V*', *E*'), mapping graph *G* into graph *G*' maps each vertex in the set *V* onto a vertex (or a set of vertices) in set *V*' and each edge in the set *E* onto an edge (or a set of edges) in *E*'. When mapping graph *G*(*V*, *E*) into *G*'(*V*', *E*'), three parameters are important. First, it is possible that more than one edge in *E* is mapped onto a single edge in *E*'. The maximum number of edges mapped onto any edge in *E*' is called the ***congestion*** of the mapping. In Example 2.17, the mapping in Figure 2.29(c) has a congestion of one and that in Figure 2.29(d) has a congestion of six. Second, an edge in *E* may be mapped onto multiple contiguous edges in *E*'. This is significant because traffic on the corresponding communication link must traverse more than one link, possibly contributing to congestion on the network. The maximum number of links in *E*' that any edge in *E* is mapped onto is called the ***dilation*** of the mapping. Third, the sets *V* and *V*' may contain different numbers of vertices. In this case, a node in *V* corresponds to more than one node in *V*'. The ratio of the number of nodes in the set *V*' to that in set *V* is called the ***expansion*** of the mapping. In the context of process-processor mapping, we want the expansion of the mapping to be identical to the ratio of virtual and physical processors.

In this section, we discuss embeddings of some commonly encountered graphs such as 2-D meshes (matrix operations illustrated in Chapter 8), hypercubes (sorting and FFT algorithms in Chapters 9 and 13, respectively), and trees (broadcast, barriers in Chapter 4). We limit the scope of the discussion to cases in which sets *V* and *V'* contain an equal number of nodes (i.e., an expansion of one).

Embedding a Linear Array into a Hypercube

A linear array (or a ring) composed of 2^d nodes (labeled 0 through 2^d -1) can be embedded into a *d*-dimensional hypercube by mapping node *i* of the linear array onto node *G*(*i*, *d*) of the hypercube. The function *G*(*i*, *x*) is defined as follows:

The function *G* is called the ***binary reflected Gray code*** (RGC). The entry *G*(*i*, *d*) denotes the *i* th entry in the sequence of Gray codes of *d* bits. Gray codes of *d* + 1 bits are derived from a table of Gray codes of *d* bits by reflecting the table and prefixing the reflected entries with a 1 and the original entries with a 0. This process is illustrated in Figure 2.30(a).

Figure 2.30. (a) A three-bit reflected Gray code ring; and (b) its embedding into a three-dimensional hypercube.

A careful look at the Gray code table reveals that two adjoining entries ($G(i, d)$ and $G(i + 1, d)$) differ from each other at only one bit position. Since node i in the linear array is mapped to node $G(i, d)$, and node $i + 1$ is mapped to $G(i + 1, d)$, there is a direct link in the hypercube that corresponds to each direct link in the linear array. (Recall that two nodes whose labels differ at only one bit position have a direct link in a hypercube.) Therefore, the mapping specified by the function G has a dilation of one and a congestion of one. Figure 2.30(b) illustrates the embedding of an eight-node ring into a three-dimensional hypercube.

Embedding a Mesh into a Hypercube

Embedding a mesh into a hypercube is a natural extension of embedding a ring into a hypercube. We can embed a 2^r x 2^s wraparound mesh into a 2^{r+s} -node hypercube by

mapping node (i, j) of the mesh onto node $G(i, r - 1)||G(j, s - 1)$ of the hypercube (where || denotes concatenation of the two Gray codes). Note that immediate neighbors in the mesh are mapped to hypercube nodes whose labels differ in exactly one bit position. Therefore, this mapping has a dilation of one and a congestion of one.

For example, consider embedding a 2 x 4 mesh into an eight-node hypercube. The values of r and s are 1 and 2, respectively. Node (i, j) of the mesh is mapped to node $G(i, 1)||G(j, 2)$ of the hypercube. Therefore, node (0, 0) of the mesh is mapped to node 000 of the hypercube, because $G(0, 1)$ is 0 and $G(0, 2)$ is 00; concatenating the two yields the label 000 for the hypercube node. Similarly, node (0, 1) of the mesh is mapped to node 001 of the hypercube, and so on. Figure 2.31 illustrates embedding meshes into hypercubes.

Figure 2.31. (a) A 4 x 4 mesh illustrating the mapping of mesh nodes to the nodes in a four-dimensional hypercube; and (b) a 2 x 4 mesh embedded into a three-dimensional hypercube.

This mapping of a mesh into a hypercube has certain useful properties. All nodes in

the same row of the mesh are mapped to hypercube nodes whose labels have *r* identical most significant bits. We know from Section 2.4.3 that fixing any *r* bits in the node label of an (*r* + *s*)-dimensional hypercube yields a subcube of dimension *s* with 2^s nodes. Since each mesh node is mapped onto a unique node in the hypercube, and each row in the mesh has 2^s nodes, every row in the mesh is mapped to a distinct subcube in the hypercube. Similarly, each column in the mesh is mapped to a distinct subcube in the hypercube.

Embedding a Mesh into a Linear Array

We have, up until this point, considered embeddings of sparser networks into denser networks. A 2-D mesh has 2 x *p* links. In contrast, a *p*-node linear array has *p* links. Consequently, there must be a congestion associated with this mapping.

Consider first the mapping of a linear array into a mesh. We assume that neither the mesh nor the linear array has wraparound connections. An intuitive mapping of a linear array into a mesh is illustrated in Figure 2.32. Here, the solid lines correspond to links in the linear array and normal lines to links in the mesh. It is easy to see from Figure 2.32(a) that a congestion-one, dilation-one mapping of a linear array to a mesh is possible.

Figure 2.32. (a) Embedding a 16 node linear array into a 2-D mesh; and (b) the inverse of the mapping. Solid lines correspond to links in the linear array and normal lines to links in the mesh.

Consider now the inverse of this mapping, i.e., we are given a mesh and we map vertices of the mesh to those in a linear array using the inverse of the same mapping function. This mapping is illustrated in Figure 2.32(b). As before, the solid lines correspond to edges in the linear array and normal lines to edges in the mesh. As is evident from the figure, the congestion of the mapping in this case is five – i.e., no solid line carries more than five normal lines. In general, it is easy to show that the congestion of this (inverse) mapping is for a general p-node mapping (one for each of the edges to the next row, and one additional edge).

While this is a simple mapping, the question at this point is whether we can do better. To answer this question, we use the bisection width of the two networks. We know that the bisection width of a 2-D mesh without wraparound links is , and that of a linear array is 1. Assume that the best mapping of a 2-D mesh into a linear array has a congestion of *r*. This implies that if we take the linear array and cut it in half (at the middle), we will cut only one linear array link, or no more than *r* mesh links. We claim that *r* must be at least equal to the bisection width of the mesh. This follows from the fact that an equi-partition of the linear array into two also partitions the mesh into two. Therefore, at least mesh links must cross the partition, by definition of bisection width. Consequently, the one linear array link connecting the two halves must carry at least mesh links. Therefore, the congestion of any mapping is lower bounded by . This is almost identical to the simple (inverse) mapping we have illustrated in Figure 2.32(b).

The lower bound established above has a more general applicability when mapping denser networks to sparser ones. One may reasonably believe that the lower bound on congestion of a mapping of network *S* with *x* links into network *Q* with *y* links is *x*/*y*. In the case of the mapping from a mesh to a linear array, this would be 2*p*/*p*, or 2. However, this lower bound is overly conservative. A tighter lower bound is in fact possible by examining the bisection width of the two networks. We illustrate this further in the next section.

Embedding a Hypercube into a 2-D Mesh

Consider the embedding of a *p*-node hypercube into a *p*-node 2-D mesh. For the sake of convenience, we assume that *p* is an even power of two. In this scenario, it is possible to visualize the hypercube as subcubes, each with nodes. We do this as follows: let *d* = log *p* be the dimension of the hypercube. From our assumption, we know that *d* is even. We take the *d*/2 least significant bits and use them to define individual subcubes of nodes. For example, in the case of a 4D hypercube, we use the lower two bits to define the subcubes as (0000, 0001, 0011, 0010), (0100, 0101, 0111, 0110), (1100, 1101, 1111, 1110), and (1000, 1001, 1011, 1010). Note at this point that if we fix the *d*/2 least significant bits across all of these subcubes, we will have another subcube as defined by the *d*/2 most significant bits. For example, if we fix the lower two bits across the subcubes to 10, we get the nodes (0010, 0110, 1110, 1010). The reader can verify that this corresponds to a 2-D subcube.

The mapping from a hypercube to a mesh can now be defined as follows: each node subcube is mapped to a node row of the mesh. We do this by simply inverting the linear array to hypercube mapping. The bisection width of the node hypercube is . The corresponding bisection width of a node row is 1. Therefore the congestion of this subcube to-row mapping is (at the edge that connects the two halves of the row). This is illustrated for the cases of *p* = 16 and *p* = 32 in Figure 2.33(a) and (b). In this fashion, we can map each subcube to a different row in the mesh. Note that while we have computed the congestion resulting from the subcube-to-row mapping, we have not addressed the congestion resulting from the column mapping. We map the hypercube nodes into the mesh in such a way that nodes with identical *d*/2 least significant bits in the hypercube are mapped to the same column. This results in a subcube-to-column mapping, where each subcube/column has nodes. Using the same argument as in the case of subcube-to-row mapping, this results in a congestion of . Since the congestion from the row and column mappings affects disjoint sets of edges, the total congestion of this mapping is .

Figure 2.33. Embedding a hypercube into a 2-D mesh.