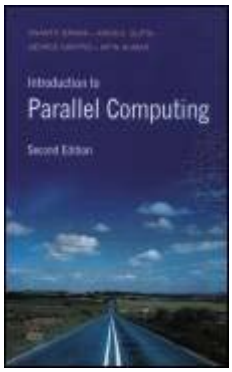


• Table of Contents

Introduction to Parallel Computing, Second Edition

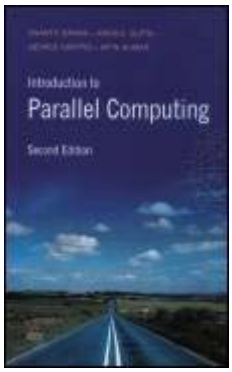
By Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar

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

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.

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-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 programming models and the evolving application mix of parallel computing is reflected in various examples throughout the book.



- Table of Contents

Introduction to Parallel Computing, Second Edition

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

Start Reading

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

PREVIOUS NEXT

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

PREVIOUS NEXT

Pearson Education

We work with leading authors to develop the strongest educational materials in computing, bringing cutting-edge thinking and best learning practice to a global market.

Under a range of well-known imprints, including Addison-Wesley, we craft high-quality print and electronic publications which help readers to understand and apply their content, whether studying or at work.

To find out more about the complete range of our publishing, please visit us on the World Wide Web at: www.pearsoneduc.com

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.

PREVIOUS NEXT

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.

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.

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.

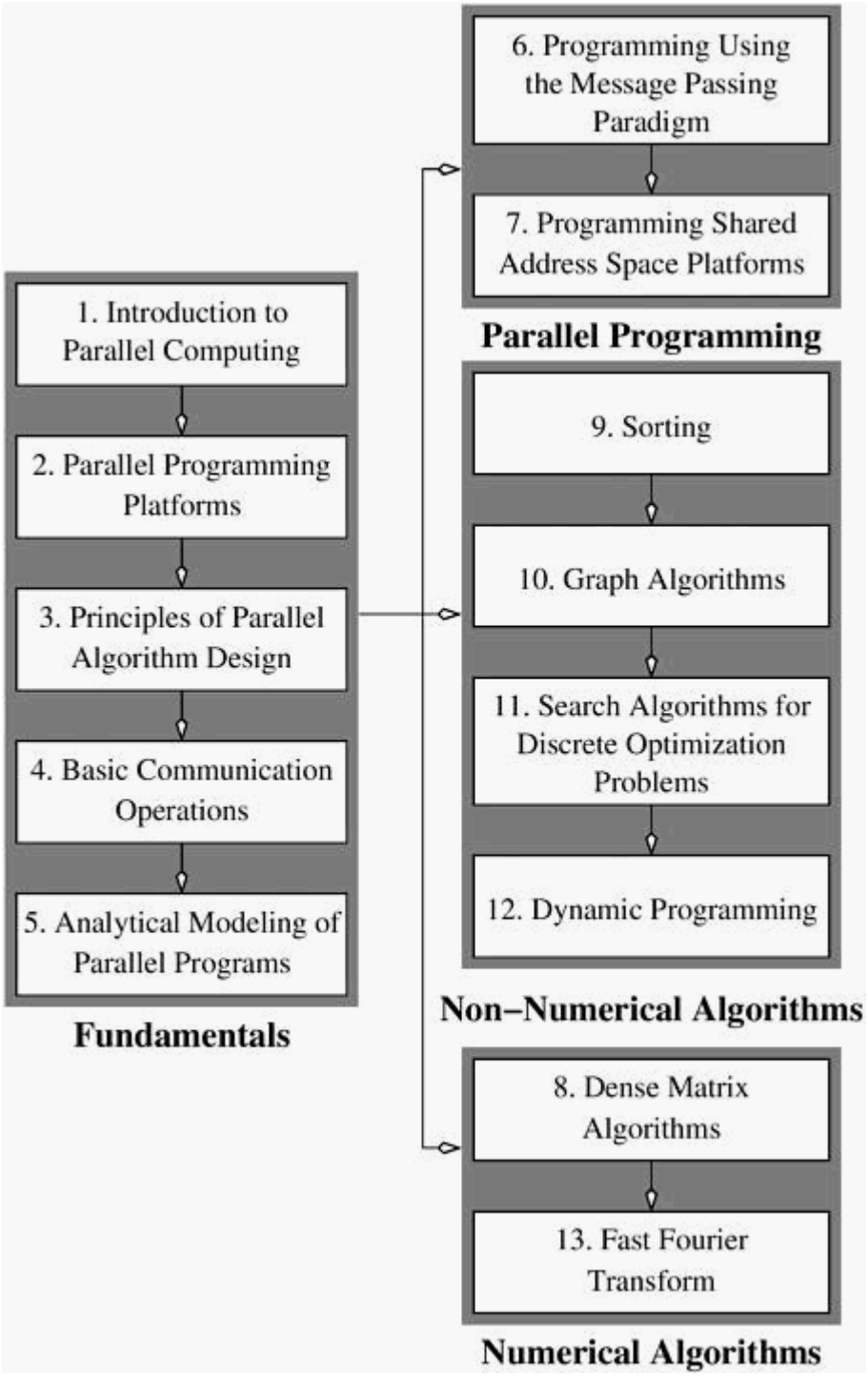
While parallel computing has traditionally confined itself to platforms with well behaved compute and network elements in which faults and errors do not play a significant role, there are valuable lessons that extend to computations on ad-hoc, mobile, or faulty environments.

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.

PREVIOUS NEXT

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.

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.

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

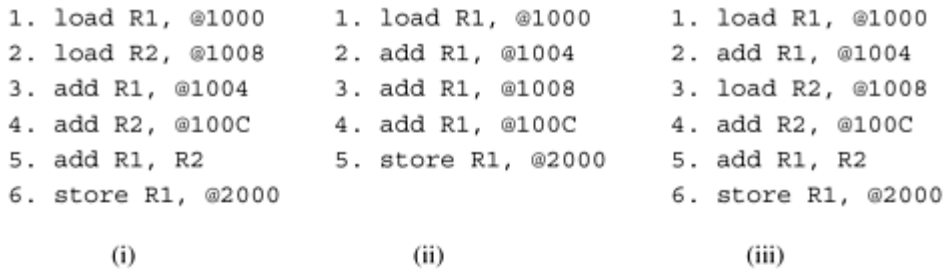
Processors have long relied on pipelines for improving execution rates. By overlapping various stages in instruction execution (fetch, schedule, decode, operand fetch, execute, store, among others), pipelining enables faster execution. The assembly-line analogy works well for understanding pipelines. 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.

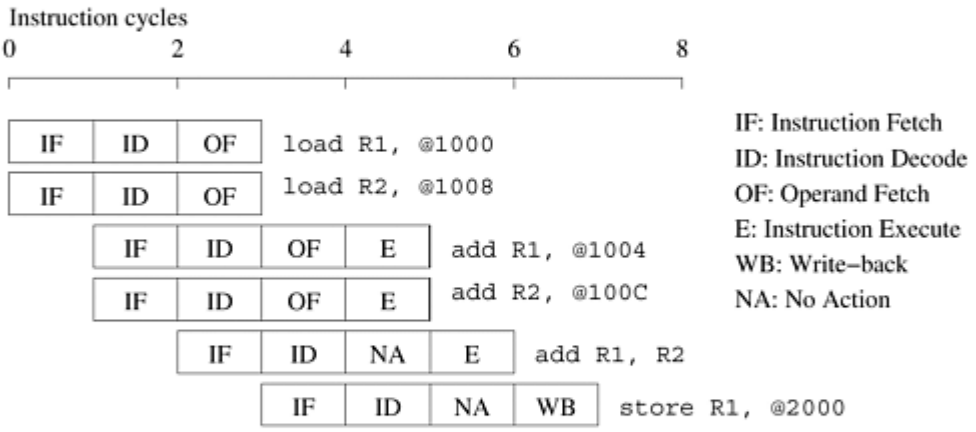
Example 2.1 Superscalar execution

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.

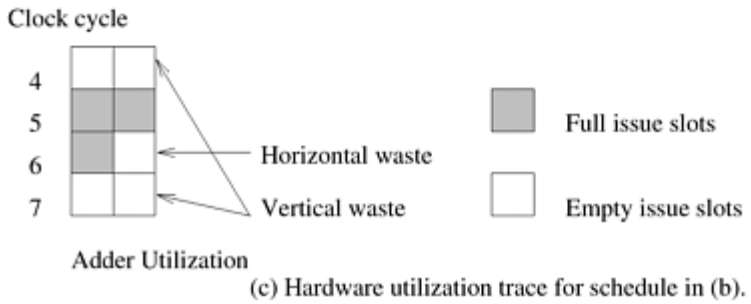
Figure 2.1. Example of a two-way superscalar execution of instructions.



(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

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 x 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.

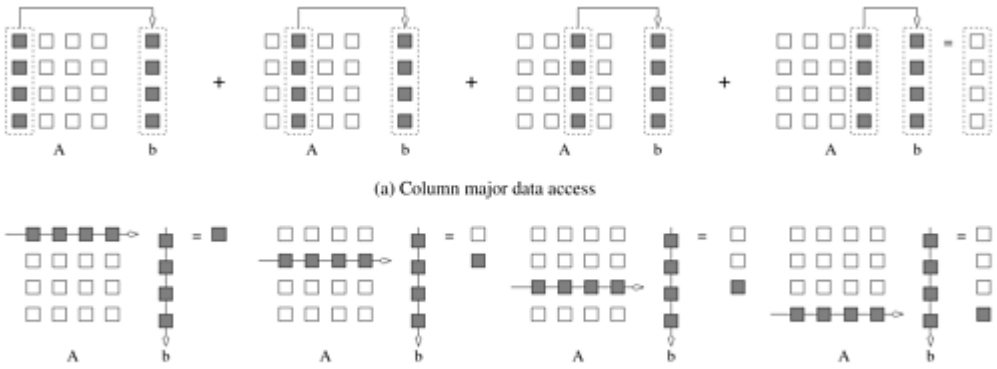
Example 2.5 Impact of strided access

Consider the following code fragment:

```
1  for (i = 0; i < 1000; i++)
2          column_sum[i] = 0.0;
3          for (j = 0; j < 1000; j++)
4                  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:

```
1  for (i = 0; i < 1000; i++)
2          column_sum[i] = 0.0;
3  for (j = 0; j < 1000; j++)
4          for (i = 0; i < 1000; i++)
5                  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* x *n* matrix `a` by a vector `b` to get vector `c`.

```
1   for(i=0;i<n;i++)
2      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:

```
1  for(i=0;i<n;i++)
2     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.

PREVIOUS NEXT

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:

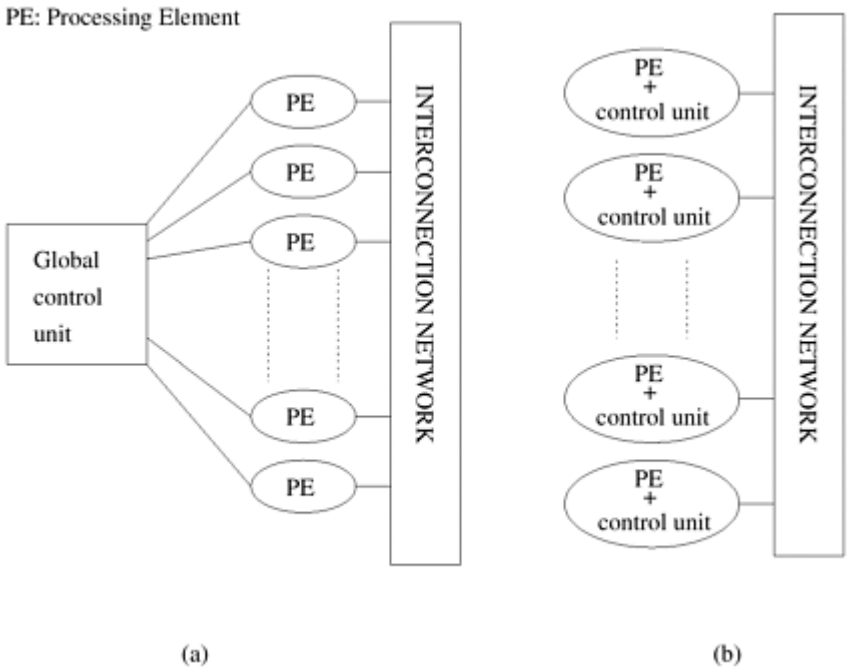
```
1   for (i = 0; i < 1000; i++)
2           c[i] = a[i] + b[i];
```

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.

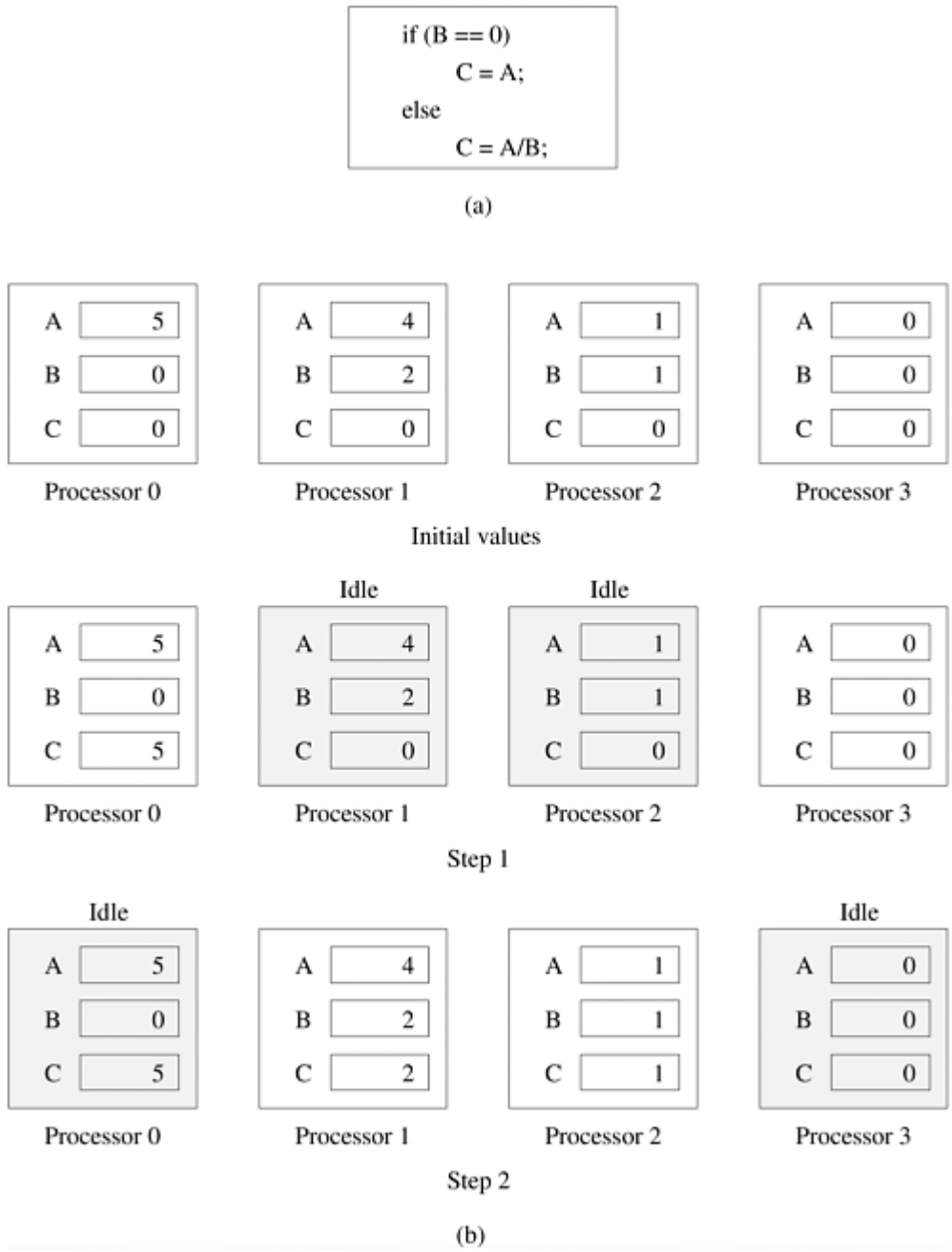
SIMD computers require less hardware than MIMD computers because they have only one global control unit. Furthermore, SIMD computers require less memory because only one copy of the program needs to be stored. In contrast, MIMD computers store the program and operating system at each processor. However, 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

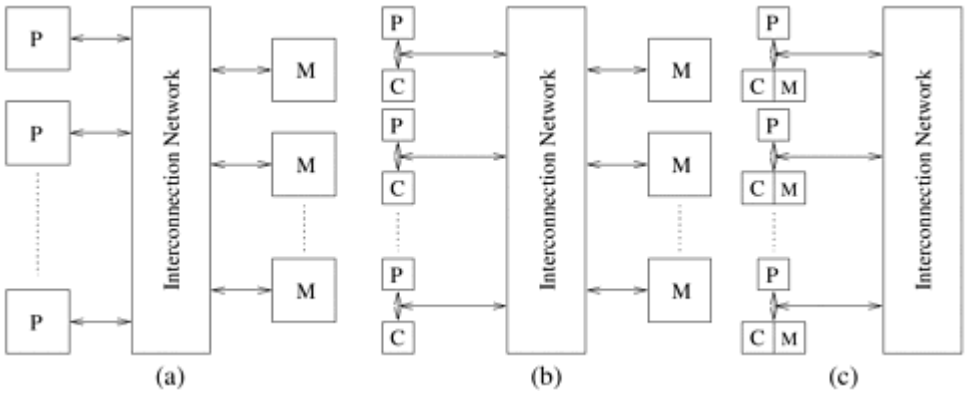
There are two primary forms of data exchange between parallel tasks – accessing a shared data space and exchanging messages.

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.

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

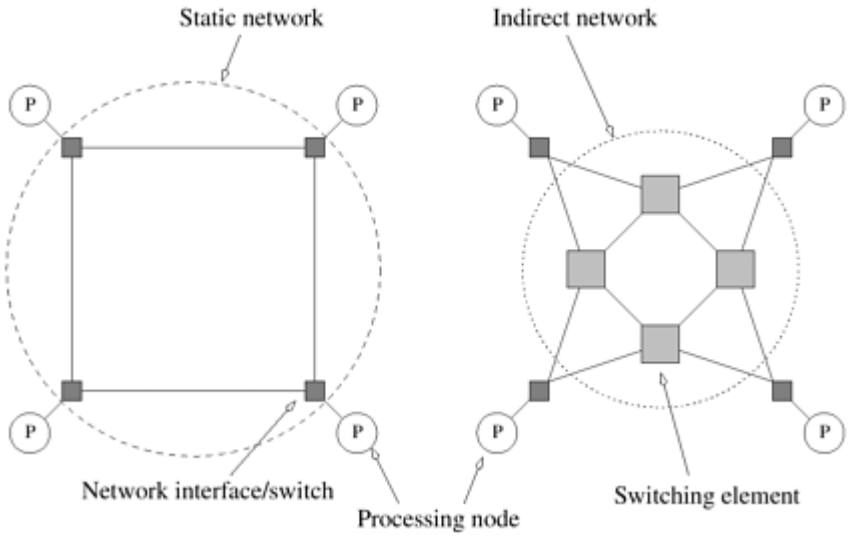
Consider the implementation of an EREW PRAM as a shared-memory computer with p processors and a global memory of m words. The processors are connected to the memory through a set of switches. These switches determine the memory word being accessed by each processor. In an EREW PRAM, each of the p processors 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 $\Theta(mp)$. (See the Appendix for an explanation of the Θ 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.

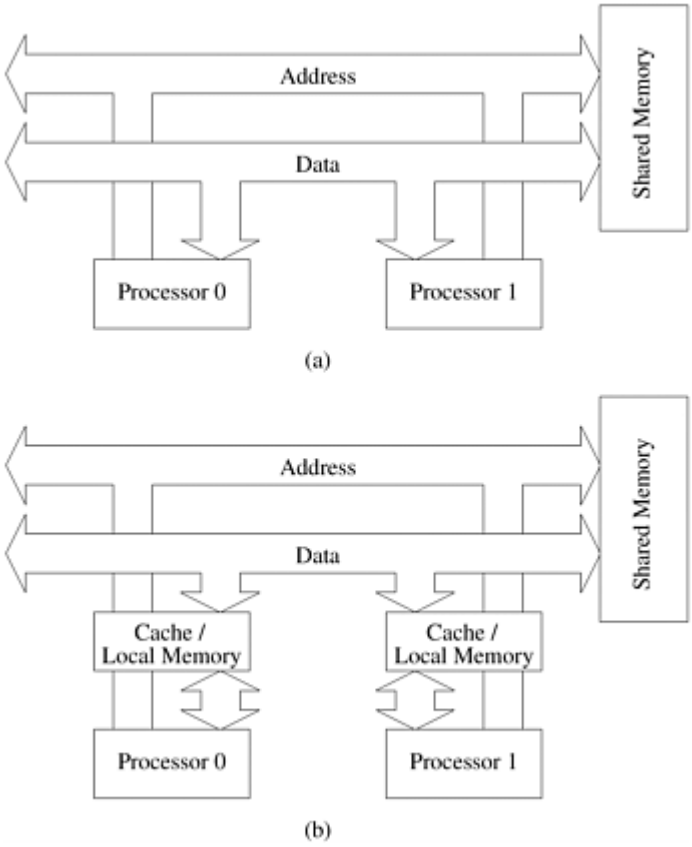
The demands on bus bandwidth can be reduced by making use of the property that in typical programs, a majority of the data accessed is local to the node. For such programs, it is possible to provide a cache for each node. Private data is cached at the node and only remote data is accessed through the 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} x 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 x t_{cycle} x k + 0.5 x t_{cycle} x 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 x t_{cycle} x 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.

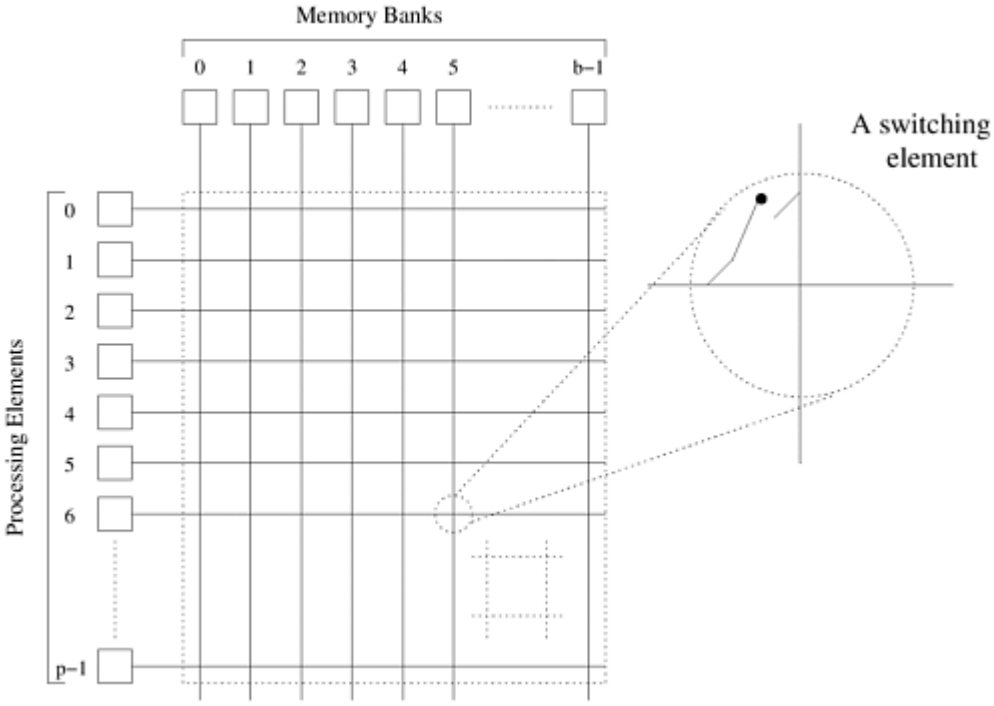


In practice, shared and private data is handled in a more sophisticated manner. This is briefly addressed with cache coherence issues in Section 2.4.6.

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 $\Theta(pb)$. 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 $\Omega(p^2)$. (See the Appendix for an explanation of the Ω 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

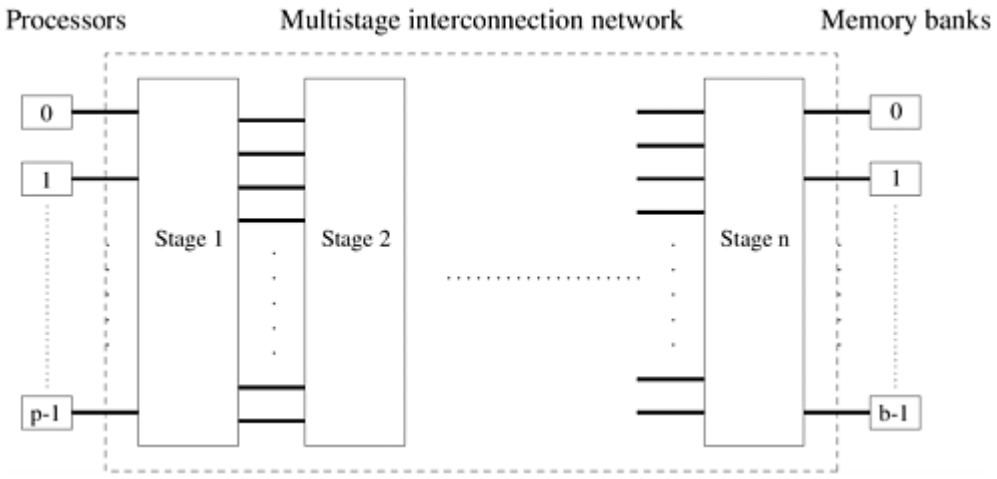
The crossbar interconnection network is scalable in terms of performance but unscalable in terms of cost. Conversely, the shared bus network is scalable in terms of cost but unscalable in terms of performance. An intermediate class of networks called *multistage interconnection networks* lies between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

$$j = \begin{cases} 2i, & 0 \le i \le p/2 - 1 \\ 2i + 1 - p, & p/2 \le i \le p - 1 \end{cases}$$

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.

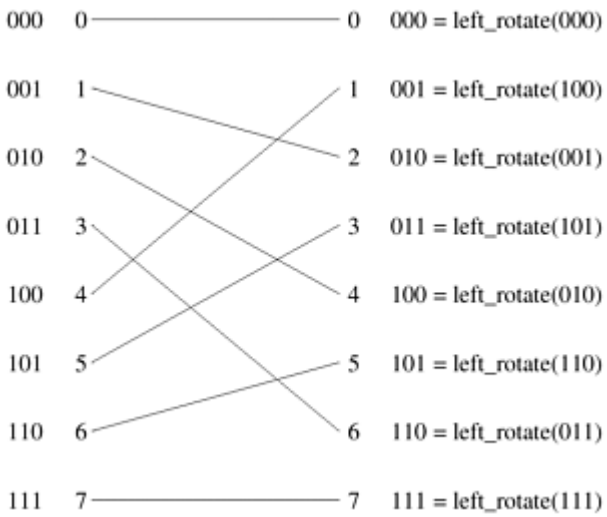


Figure 2.11. Two switching configurations of the 2 x 2 switch: (a) Pass-through; (b) Cross-over.

(a) (b)

An omega network has $p/2$ x log p switching nodes, and the cost of such a network grows as $\Theta(p \log p)$. Note that this cost is less than the $\Theta(p^2)$ 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.

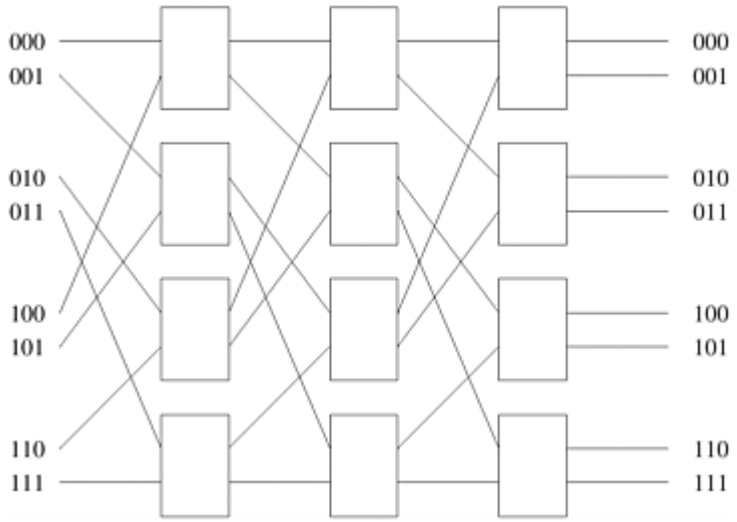
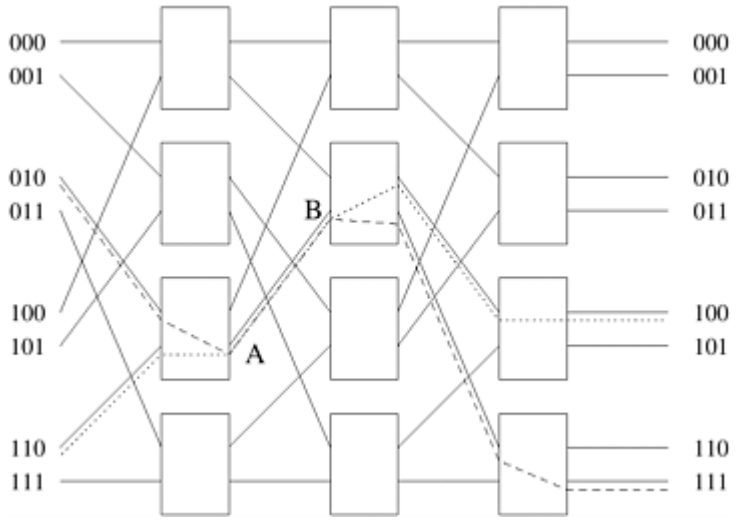


Figure 2.13 shows data routing over an omega network from processor two (010) to memory bank seven (111) and from processor six (110) to memory bank four (100). This figure also illustrates an important property of this network. When processor two (010) is communicating with memory bank seven (111), 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*.

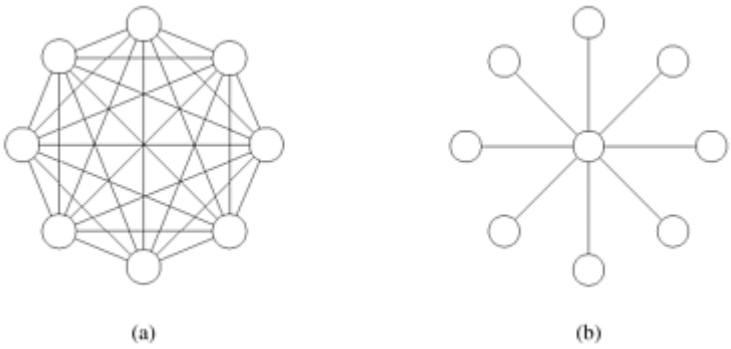
Figure 2.13. An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.



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.



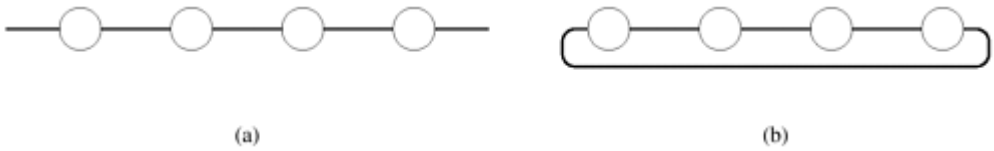
Star-Connected Network

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

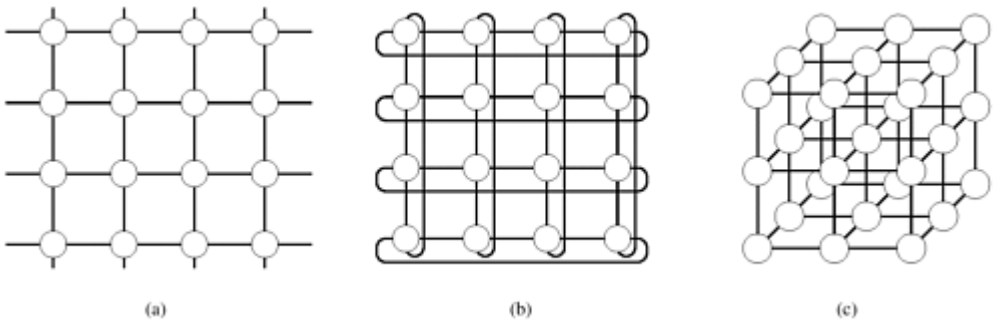
Due to the large number of links in completely connected networks, sparser networks are typically used to build parallel computers. A family of such networks spans the space of linear arrays and hypercubes. A linear array is a static network in which each node (except the two nodes at the ends) has two 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 $\sqrt{p}$ 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).

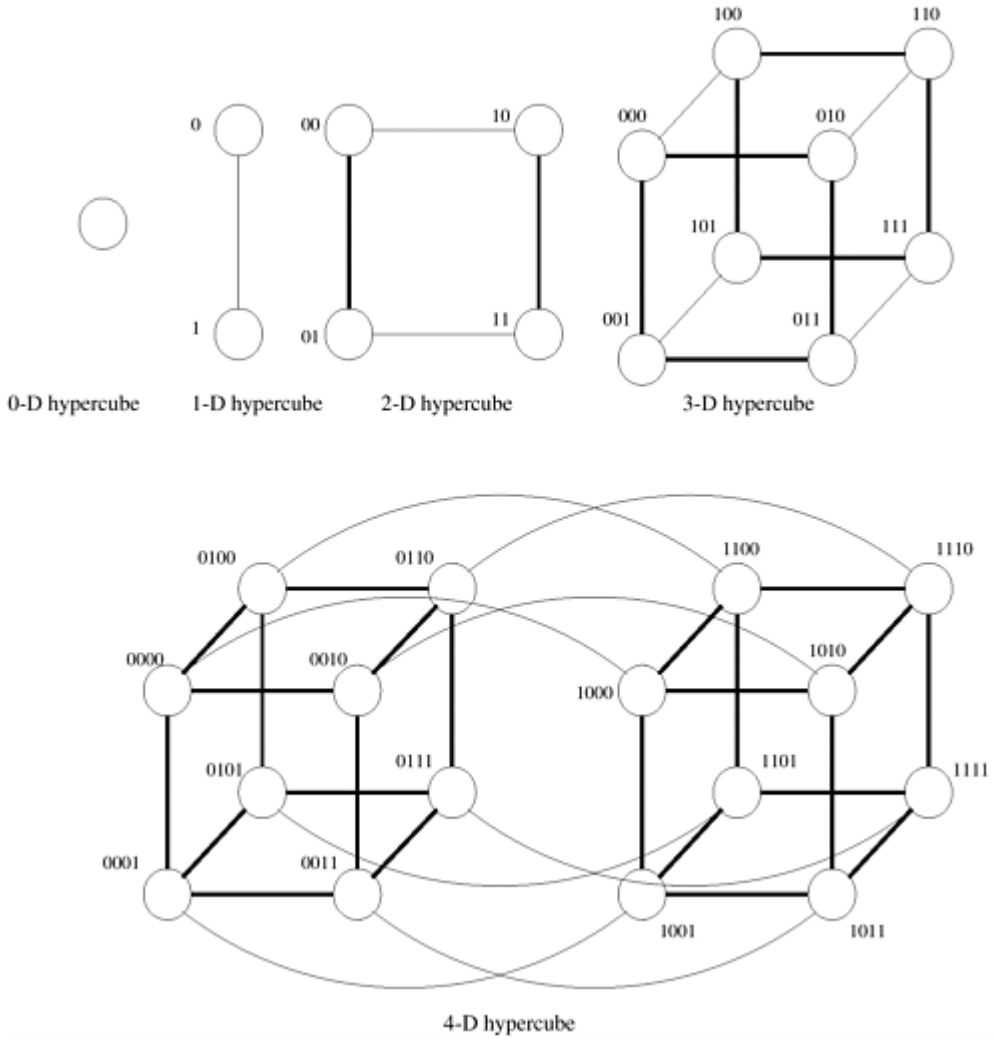
Figure 2.16. Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.



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.

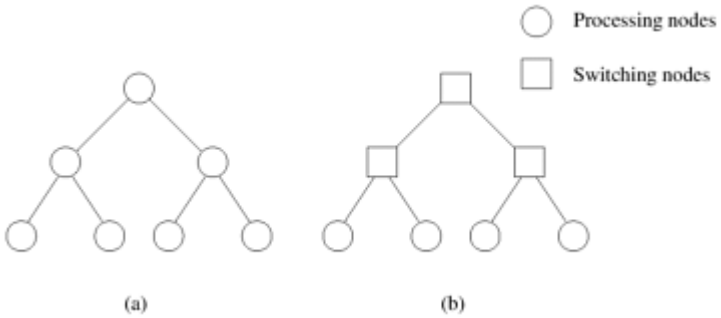


It is useful to derive a numbering scheme for nodes in a hypercube. A simple numbering scheme can be derived from the construction of a hypercube. As illustrated in Figure 2.17, if we have a numbering of two subcubes of $p/2$ nodes, we can derive a numbering scheme for the cube of p nodes by prefixing 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)).

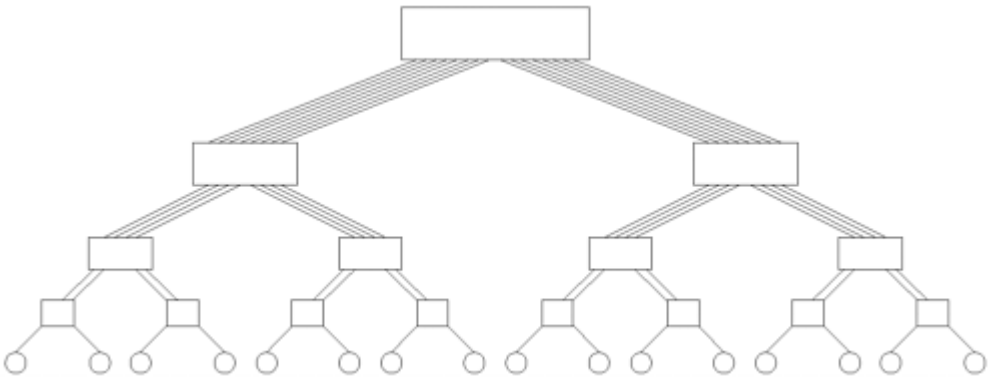
Figure 2.18. Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.



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 $\lfloor p/2 \rfloor$. The diameter of a two-dimensional mesh without wraparound connections is $2(\sqrt{p}-1)$ for the two nodes at diagonally opposed corners, and that of a wraparound mesh is $2\lfloor \sqrt{p}/2 \rfloor$. 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 $\sqrt{p}$ and with wraparound connections is $2\sqrt{p}$. 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
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	4	2p

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Hypercube	$\log p$	$p/2$	$\log p$	$(p \log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2 \rfloor$	$2k^{d-1}$	$2d$	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 $\Theta(w^2)$, and the lower bound on the volume in a three-dimensional packaging is $\Theta(w^{3/2})$. 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.

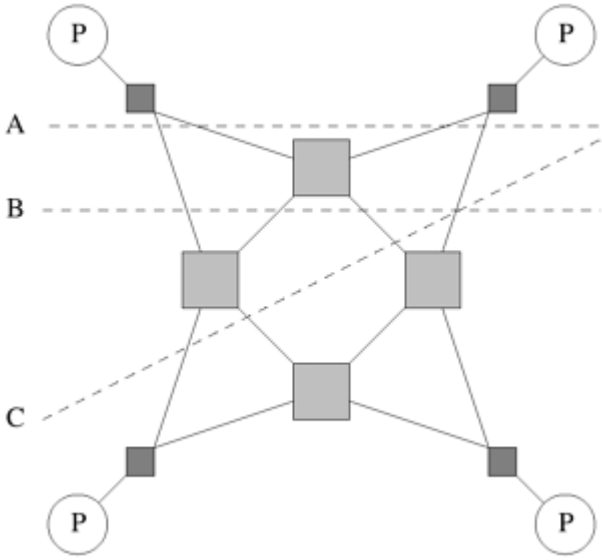
The bisection width of a dynamic network must be defined more precisely than diameter and connectivity. In the case of bisection width, we consider any possible partitioning of the p processing nodes into two equal parts. Note that this does not restrict the partitioning of the switching nodes. For each 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

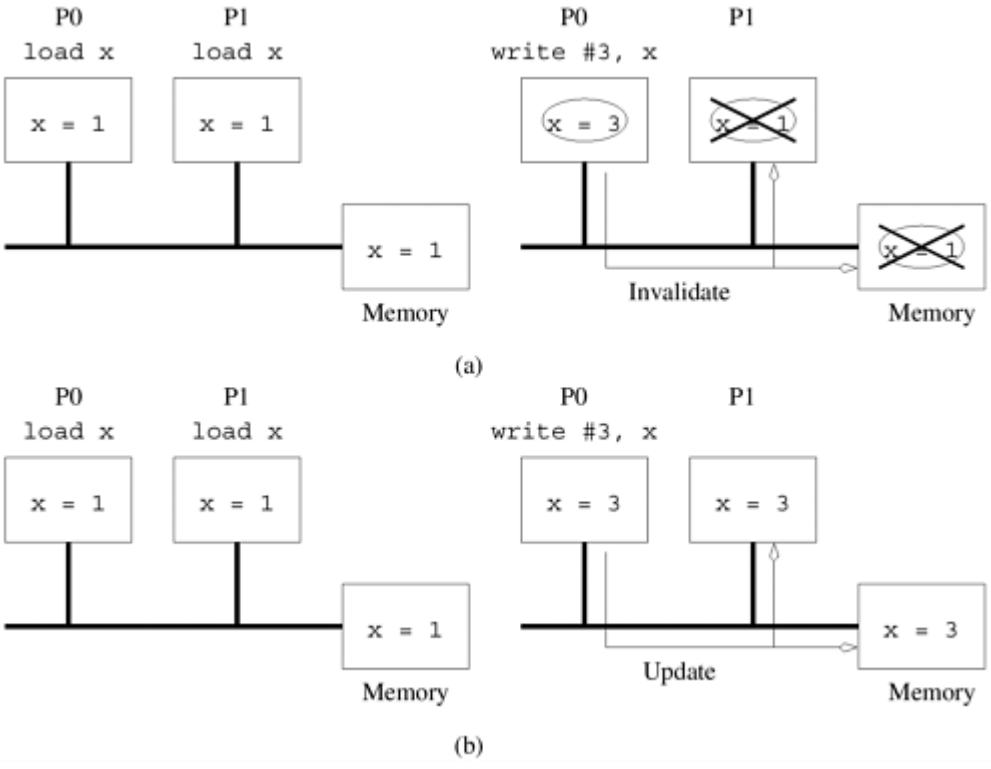
different caches/memory elements), how do we ensure that different processors operate on these in a manner that follows predefined semantics?

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.

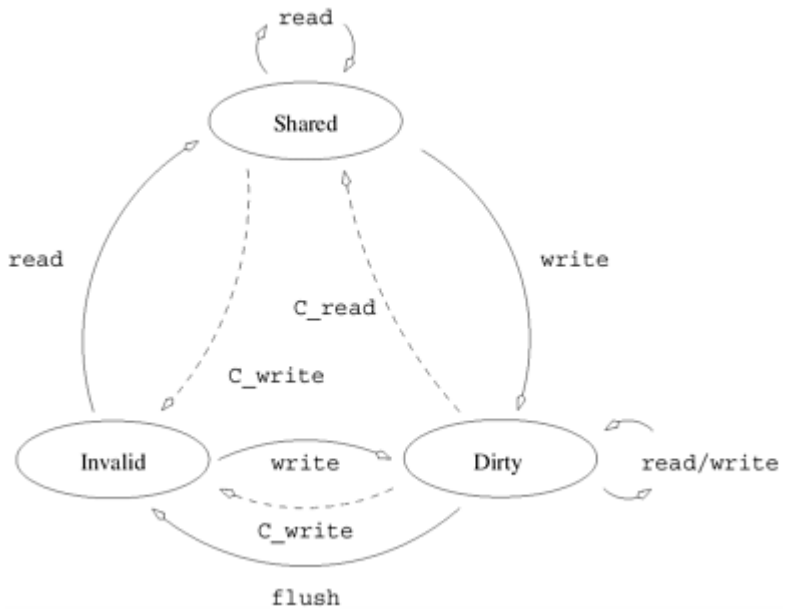
The tradeoff between invalidate and update schemes is the classic tradeoff between communication overhead (updates) and idling (stalling in invalidates). Current generation cache coherent machines typically rely on invalidate protocols. The rest of our discussion of multiprocessor cache systems therefore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

Time ↓

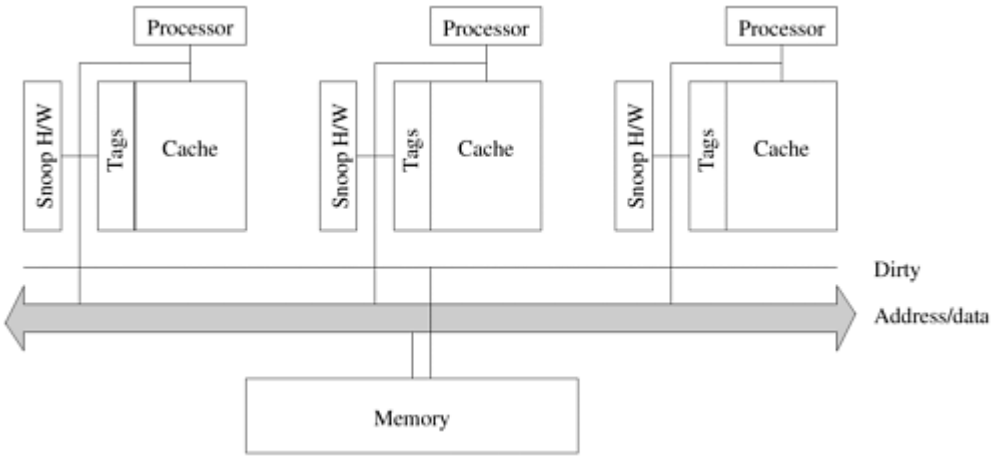
Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
				x = 5, D y = 12, D
read x		x = 5, S		x = 5, S
	read y		y = 12, S	y = 12, S
x = x + 1		x = 6, D		x = 5, I
	y = y + 1		y = 13, D	y = 12, I
read y		y = 13, S	y = 13, S	y = 13, S
	read x	x = 6, S	x = 6, S	x = 6, S
x = x + y		x = 19, D	x = 6, I	x = 6, I
	y = x + y	y = 13, I	y = 19, D	y = 13, I
x = x + 1		x = 20, D		x = 6, I
	y = y + 1		y = 20, D	y = 13, I

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.



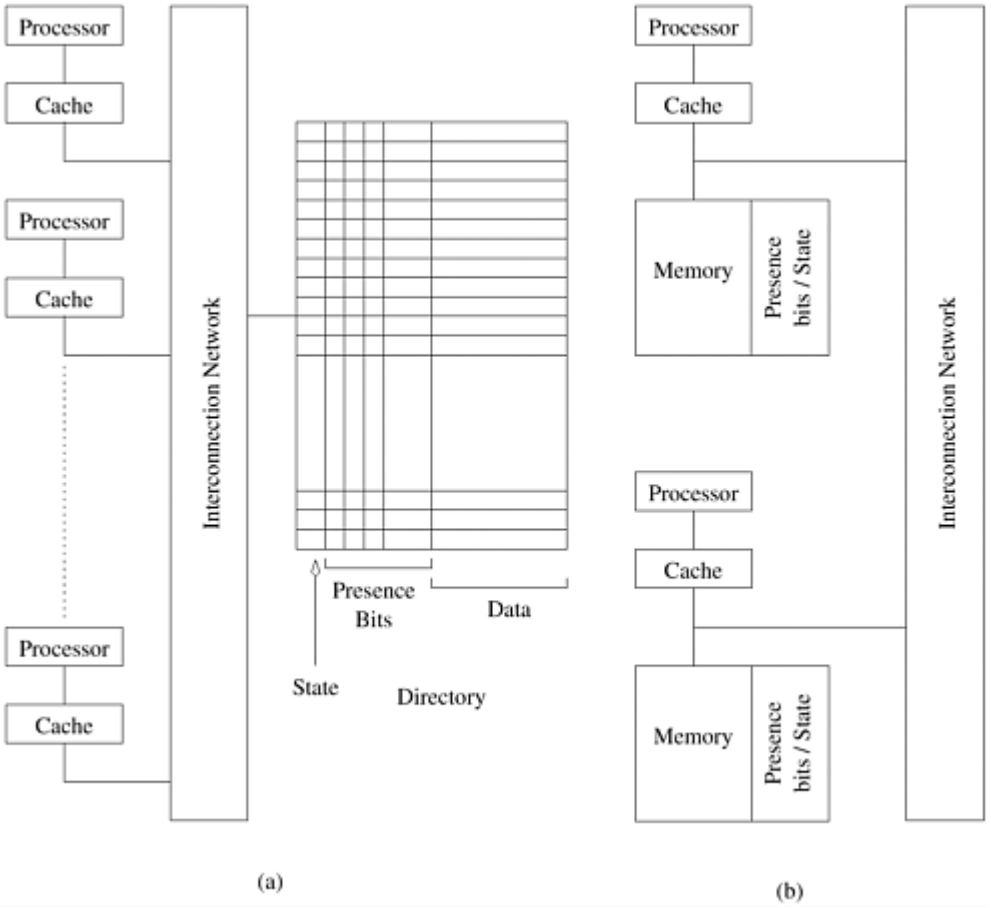
Performance of Snoopy Caches Snoopy protocols have been extensively studied and used in commercial systems. This is largely because of their simplicity and the fact that existing bus based systems can be upgraded to accommodate snoopy protocols. The performance gains of snoopy systems are derived from 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.

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.

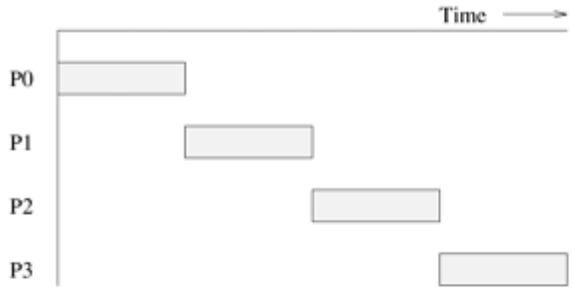
3. *Per-word transfer time* (t_w): If the channel bandwidth is r words per second, then each word takes time $t_w = 1/r$ to traverse the link. This time is called the per-word transfer time. This time includes network as well as buffering overheads.

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

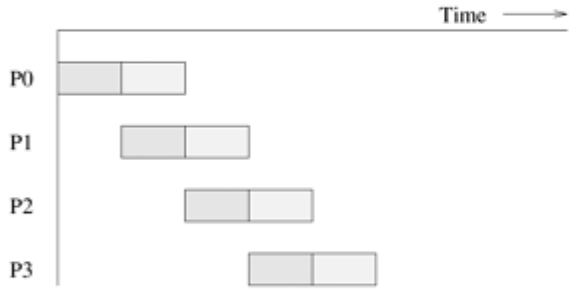
Store-and-Forward Routing

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.

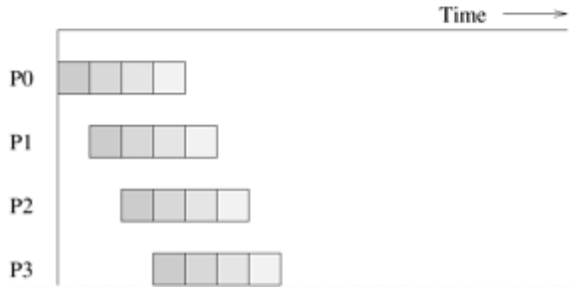
Figure 2.26. Passing a message from node P_0 to P_3 (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed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



(a) A single message sent over a store-and-forward network



(b) The same message broken into two parts and sent over the network.



(c) The same message broken into four parts and sent over the network.

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

$$t_{comm} = t_s + (mt_w + t_h)l.$$

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

$$t_{comm} = t_s + mlt_w.$$

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:

$$\begin{aligned} t_{comm} &= t_s + t_{w1}m + t_h l + t_{w2}(r+s) + \left(\frac{m}{r} - 1\right) t_{w2}(r+s) \\ &= t_s + t_{w1}m + t_h l + t_{w2}m + t_{w2}\frac{s}{r}m \\ &= t_s + t_h l + t_w m, \end{aligned}$$

where

$$t_w = t_{w1} + t_{w2}\left(1 + \frac{s}{r}\right).$$

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 $t_w m$ after the arrival of the header of the message. Therefore, the total communication time for cut-through routing is

Equation 2.3

$$t_{comm} = t_s + lt_h + t_w m.$$

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.

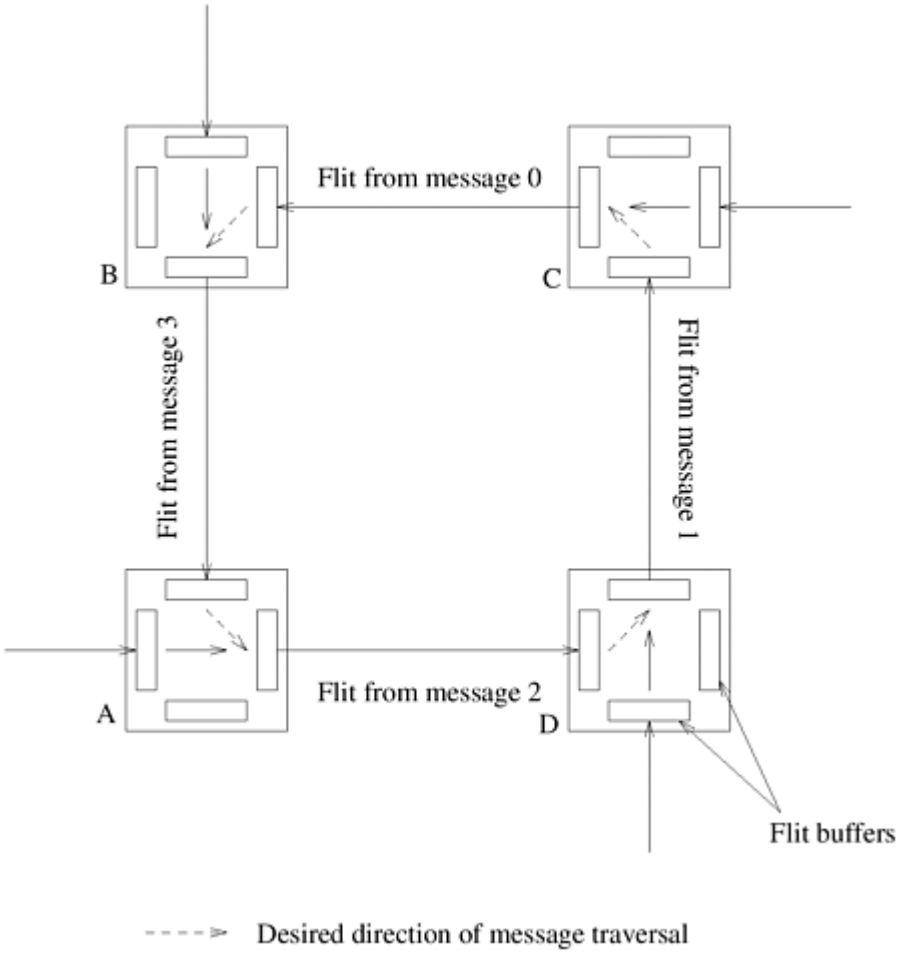
Most current parallel computers and many local area networks support cut-through routing. The size of a flit is determined by a variety of network parameters. The control circuitry must operate at the flit rate. Therefore, if we select a very small flit size, for a given link bandwidth, the required 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

$$t_{comm} = t_s + lt_h + t_w m.$$

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

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

$$t_{comm} = t_s + t_w m$$

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 $\sqrt{p} \times \sqrt{p}$ 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 $\sqrt{p}$. From these two, we can infer that some links would now have to carry at least $\frac{p/4}{\sqrt{p}} = \sqrt{p}/4$ 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 $t_s + t_w m \times \sqrt{p}/4$. 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.

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*.

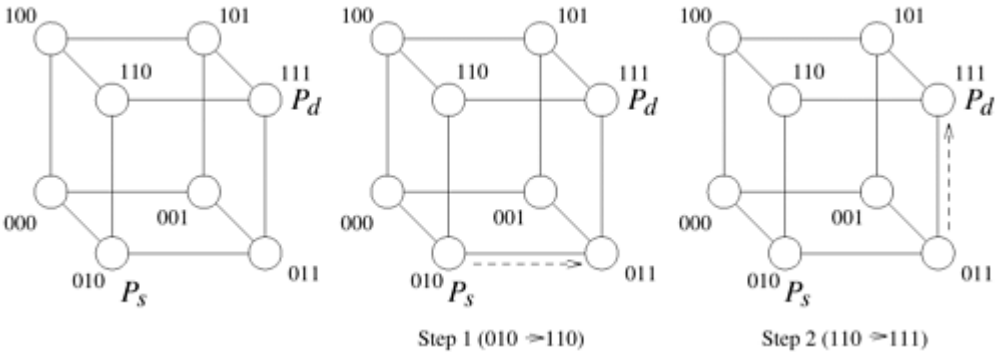
Consider a two-dimensional mesh without wraparound connections. In the XY-routing scheme, a message is sent first along the X dimension until it reaches the column of the destination node and then along the Y dimension until it reaches its destination. Let $P_{Sy,Sx}$ represent the position of the source 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 $Dx \geq Sx$ and $Dy \geq Sy$. 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 $P_s \oplus P_d$ (where $\oplus$ represents the bitwise exclusive-OR operation). In the E-cube algorithm, node P_s computes $P_s \oplus P_d$ and sends the message along dimension k, where k is the position of the least significant nonzero bit in $P_s \oplus P_d$. At each intermediate step, node P_i , which receives the message, computes $P_i \oplus P_d$ 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 $\oplus$ 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 $\oplus$ 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.

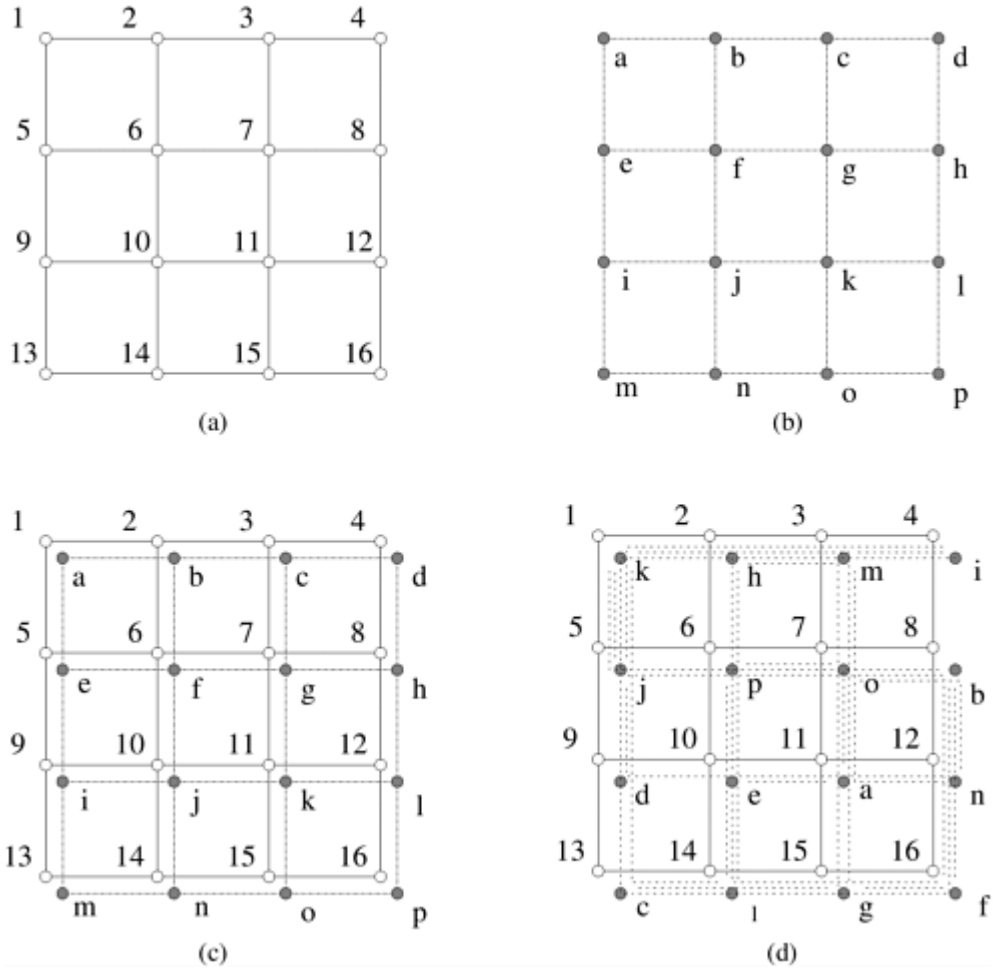
2.7 Impact of Process-Processor Mapping and Mapping Techniques

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).

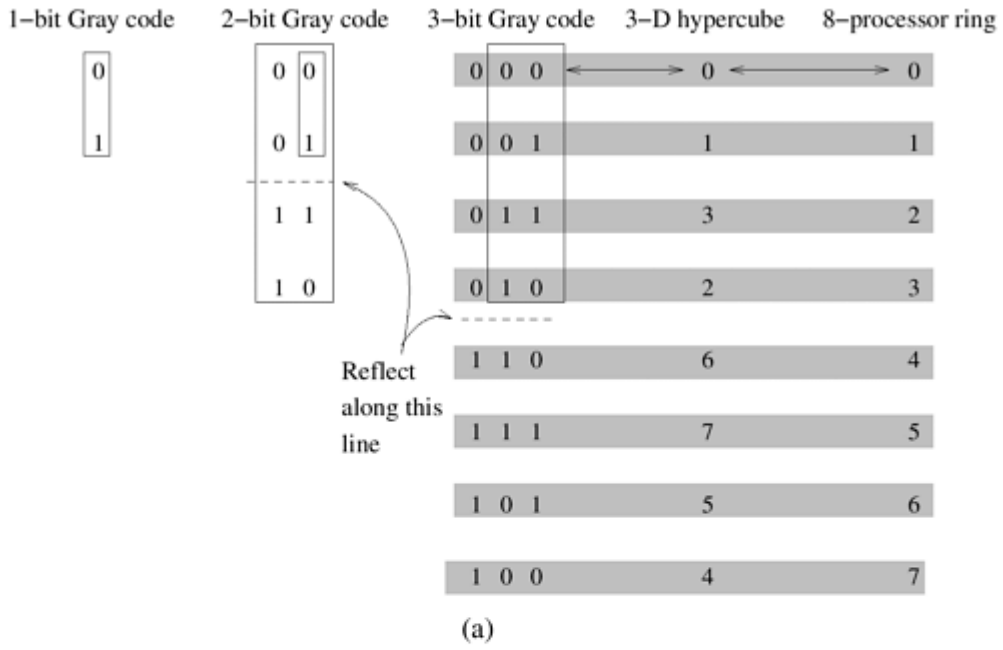
Embedding a Linear Array into a Hypercube

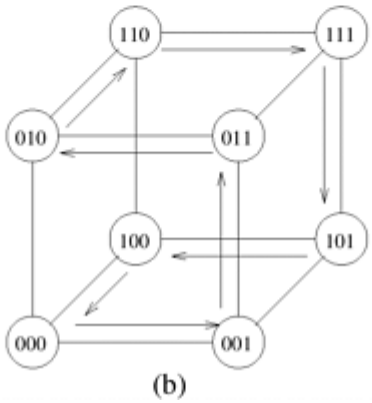
A linear array (or a ring) composed of 2^d nodes (labeled 0 through 2^d -1) can be embedded into a d-dimensional hypercube by mapping node i of the linear array onto node $G(i, d)$ of the hypercube. The function $G(i, x)$ is defined as follows:

$$
\begin{aligned}
G(0,1) &= 0 \\
G(1,1) &= 1 \\
G(i,x+1) &= \begin{cases} G(i,x), & i < 2^x \\ 2^x + G(2^{x+1} - 1 - i, x), & i \geq 2^x \end{cases}
\end{aligned}
$$

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

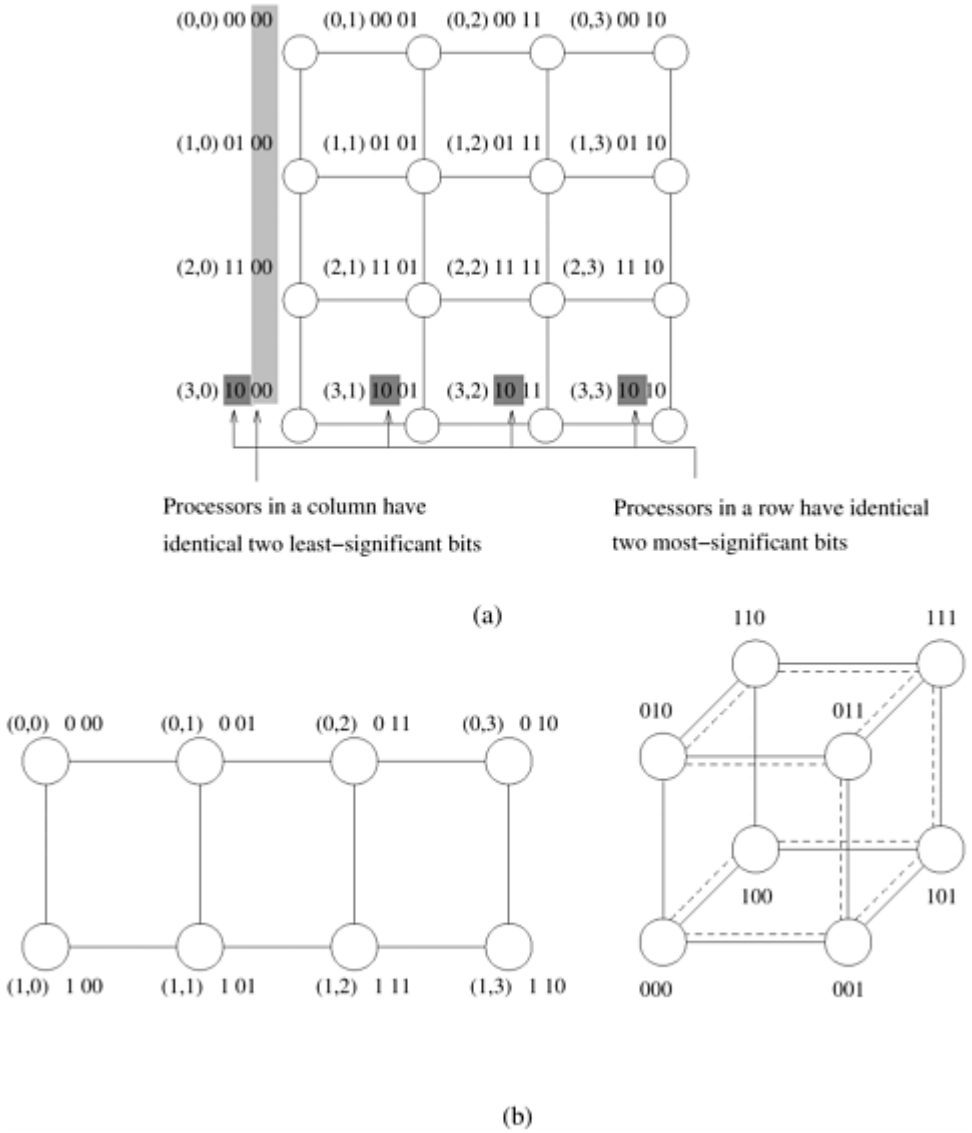
Embedding a mesh into a hypercube is a natural extension of embedding a ring into a hypercube. We can embed a 2^r x 2^s wraparound mesh into a 2^{r+s} -node hypercube by

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

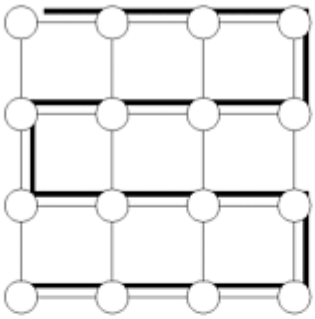
the same row of the mesh are mapped to hypercube nodes whose labels have *r* identical most significant bits. We know from Section 2.4.3 that fixing any *r* bits in the node label of an (*r* + *s*)-dimensional hypercube yields a subcube of dimension *s* with 2^s nodes. Since each mesh node is mapped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

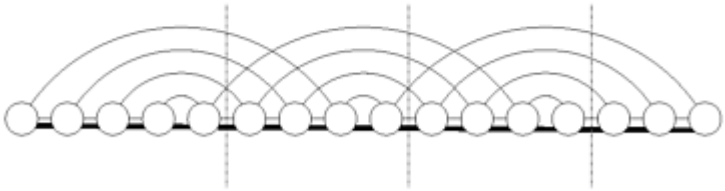
We have, up until this point, considered embeddings of sparser networks into denser networks. A 2-D mesh has 2 x *p* links. In contrast, a *p*-node linear array has *p* links. Consequently, there must be a congestion associated with this mapping.

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.



(a) Mapping a linear array into a 2D mesh (congestion 1).



(b) Inverting the mapping – mapping a 2D mesh into a linear array (congestion 5)

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 $\sqrt{p}+1$ for a general p-node mapping (one for each of the $\sqrt{p}$ 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 $\sqrt{p}$, 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 $\sqrt{p}$ 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 $\sqrt{p}$ mesh links. Therefore, the congestion of any mapping is lower bounded by $\sqrt{p}$. This is almost identical to the simple (inverse) mapping we have illustrated in Figure 2.32(b).

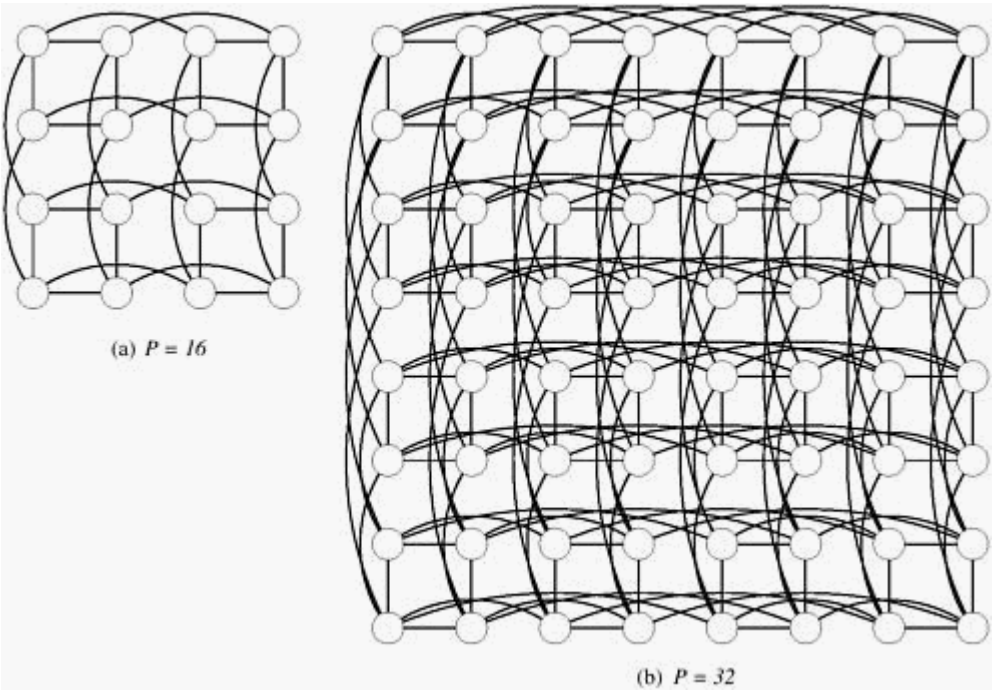
The lower bound established above has a more general applicability when mapping denser networks to sparser ones. One may reasonably believe that the lower bound on congestion of a mapping of network S with x links into network Q with y links is x/y. In the case of the mapping from a mesh to a linear array, this would be $2p/p$, 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 $\sqrt{p}$ subcubes, each with $\sqrt{p}$ 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 $\sqrt{p}$ 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 $\sqrt{p}$ node subcube is mapped to a $\sqrt{p}$ node row of the mesh. We do this by simply inverting the linear-array to hypercube mapping. The bisection width of the $\sqrt{p}$ node hypercube is $\sqrt{p}/2$. The corresponding bisection width of a $\sqrt{p}$ node row is 1. Therefore the congestion of this subcube-to-row mapping is $\sqrt{p}/2$ (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 $\sqrt{p}$ nodes. Using the same argument as in the case of subcube-to-row mapping, this results in a congestion of $\sqrt{p}/2$. Since the congestion from the row and column mappings affects disjoint sets of edges, the total congestion of this mapping is $\sqrt{p}/2$.

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



To establish a lower bound on the congestion, we follow the same argument as in Section 2.7.1. Since the bisection width of a hypercube is $p/2$ and that of a mesh is $\sqrt{p}$, the lower bound on congestion is the ratio of these, i.e., $\sqrt{p}/2$. We notice that our mapping yields this lower bound on congestion.

Process-Processor Mapping and Design of Interconnection Networks

Our analysis in previous sections reveals that it is possible to map denser networks into sparser networks with associated congestion overheads. This implies that a sparser network whose link bandwidth is increased to compensate for the congestion can be expected to perform as well as the denser network (modulo dilation effects). For example, a mesh whose links are faster by a factor of $\sqrt{p}/2$ will yield comparable performance to a hypercube. We call such a mesh a fat-mesh. A fat-mesh has the same bisection-bandwidth as a hypercube; however it has a higher diameter. As we have seen in Section 2.5.1, by using appropriate message routing techniques, the effect of node distance can be minimized. It is important to note that higher dimensional networks involve more complicated layouts, wire crossings, and variable wire-lengths. For these reasons, fattened lower dimensional networks provide attractive alternate approaches to designing interconnects. We now do a more formal examination of the cost-performance tradeoffs of parallel architectures.

2.7.2 Cost-Performance Tradeoffs

We now examine how various cost metrics can be used to investigate cost-performance tradeoffs in interconnection networks. We illustrate this by analyzing the performance of a mesh and a hypercube network with identical costs.

If the cost of a network is proportional to the number of wires, then a square p-node wraparound mesh with $(\log p)/4$ wires per channel costs as much as a p-node hypercube with one wire per channel. Let us compare the average communication times of these two networks. The average distance l_{av} between any two nodes in a two-dimensional wraparound mesh is

$\sqrt{p}/2$ and that in a hypercube is (log p)/2. The time for sending a message of size m between nodes that are l_{av} hops apart is given by $t_s + t_h l_{av} + t_w m$ in networks that use cut-through routing. Since the channel width of the mesh is scaled up by a factor of (log p)/4, the per-word transfer time is reduced by the same factor. Hence, if the per-word transfer time on the hypercube is t_w, then the same time on a mesh with fattened channels is given by $4t_w/(\log p)$. Hence, the average communication latency for a hypercube is given by $t_s + t_h (\log p)/2 + t_w m$ and that for a wraparound mesh of the same cost is $t_s + t_h\sqrt{p}/2 + 4t_w m/(\log p)$.

Let us now investigate the behavior of these expressions. For a fixed number of nodes, as the message size is increased, the communication term due to t_w dominates. Comparing t_w for the two networks, we see that the time for a wraparound mesh ($4t_w m/(\log p)$)is less than the time for a hypercube ($t_w m$)if p is greater than 16 and the message size m is sufficiently large. Under these circumstances, point-to-point communication of large messages between random pairs of nodes takes less time on a wraparound mesh with cut-through routing than on a hypercube of the same cost. Furthermore, for algorithms in which communication is suited to a mesh, the extra bandwidth of each channel results in better performance. Note that, with store-and-forward routing, the mesh is no longer more cost-efficient than a hypercube. Similar cost-performance tradeoffs can be analyzed for the general case of k-ary d-cubes (Problems 2.25–2.29).

The communication times above are computed under light load conditions in the network. As the number of messages increases, there is contention on the network. Contention affects the mesh network more adversely than the hypercube network. Therefore, if the network is heavily loaded, the hypercube will outperform the mesh.

If the cost of a network is proportional to its bisection width, then a p-node wraparound mesh with $\sqrt{p}/4$ wires per channel has a cost equal to a p-node hypercube with one wire per channel. Let us perform an analysis similar to the one above to investigate cost-performance tradeoffs using this cost metric. Since the mesh channels are wider by a factor of $\sqrt{p}/4$, the per-word transfer time will be lower by an identical factor. Therefore, the communication times for the hypercube and the mesh networks of the same cost are given by $t_s + t_h (\log p)/2 + t_w m$ and $t_s + t_h\sqrt{p}/2 + 4t_w m/\sqrt{p}$, respectively. Once again, as the message size m becomes large for a given number of nodes, the t_w term dominates. Comparing this term for the two networks, we see that for $p > 16$ and sufficiently large message sizes, a mesh outperforms a hypercube of the same cost. Therefore, for large enough messages, a mesh is always better than a hypercube of the same cost, provided the network is lightly loaded. Even when the network is heavily loaded, the performance of a mesh is similar to that of a hypercube of the same cost.

2.8 Bibliographic Remarks

Several textbooks discuss various aspects of high-performance architectures [PH90, PH96, Sto93]. Parallel architectures and interconnection networks have been well described [CSG98, LW95, HX98, Fly95, AG94, DeC89, HB84, Lil92, Sie85, Sto93]. Historically, the classification of parallel computers as SISD, SIMD, and MIMD was introduced by Flynn [Fly72]. He also proposed the MISD (multiple instruction stream, single data stream) model. MISD is less natural than the other classes, although it can be viewed as a model for pipelining. Darema [DRGNP] introduced the Single Program Multiple Data (SPMD) paradigm. Ni [Ni91] provides a layered classification of parallel computers based on hardware architecture, address space, communication model, language, programming environment, and applications.

Interconnection networks have been an area of active interest for decades. Feng [Fen81] provides a tutorial on static and dynamic interconnection networks. The perfect shuffle interconnection pattern was introduced by Stone [Sto71]. Omega networks were introduced by Lawrie [Law75]. Other multistage networks have also been proposed. These include the Flip network [Bat76] and the Baseline network [WF80]. Mesh of trees and pyramidal mesh are discussed by Leighton [Lei92]. Leighton [Lei92] also provides a detailed discussion of many related networks.

The C.mmp was an early research prototype MIMD shared-address-space parallel computer based on the Crossbar switch [WB72]. The Sun Ultra HPC Server and Fujitsu VPP 500 are examples of crossbar-based parallel computers or their variants. Several parallel computers were based on multistage interconnection networks including the BBN Butterfly [BBN89], the NYU Ultracomputer [GGK+83], and the IBM RP-3 [PBG+85]. The SGI Origin 2000, Stanford Dash [LLG+92] and the KSR-1 [Ken90] are NUMA shared-address-space computers.

The Cosmic Cube [Sei85] was among the first message-passing parallel computers based on a hypercube-connected network. These were followed by the nCUBE 2 [nCU90] and the Intel iPSC-1, iPSC-2, and iPSC/860. More recently, the SGI Origin 2000 uses a network similar to a hypercube. Saad and Shultz [SS88, SS89a] derive interesting properties of the hypercube-connected network and a variety of other static networks [SS89b]. Many parallel computers, such as the Cray T3E, are based on the mesh network. The Intel Paragon XP/S [Sup91] and the Mosaic C [Sei92] are earlier examples of two-dimensional mesh-based computers. The MIT J-Machine [D+92] was based on a three-dimensional mesh network. The performance of mesh-connected computers can be improved by augmenting the mesh network with broadcast buses [KR87a]. The reconfigurable mesh architecture (Figure 2.35 in Problem 2.16) was introduced by Miller et al. [MKRS88]. Other examples of reconfigurable meshes include the TRAC and PCHIP.

The DADO parallel computer was based on a tree network [SM86]. It used a complete binary tree of depth 10. Leiserson [Lei85b] introduced the fat-tree interconnection network and proved several interesting characteristics of it. He showed that for a given volume of hardware, no network has much better performance than a fat tree. The Thinking Machines CM-5 [Thi91] parallel computer was based on a fat tree interconnection network.

The Illiac IV [Bar68] was among the first SIMD parallel computers. Other SIMD computers include the Goodyear MPP [Bat80], the DAP 610, and the CM-2 [Thi90], MasPar MP-1, and MasPar MP-2 [Nic90]. The CM-5 and DADO incorporate both SIMD and MIMD features. Both are MIMD computers but have extra hardware for fast synchronization, which enables them to operate in SIMD mode. The CM-5 had a control network to augment the data network. The control network provides such functions as broadcast, reduction, combining, and other global operations.

Leighton [Lei92] and Ranka and Sahni [RS90b] discuss embedding one interconnection network into another. Gray codes, used in embedding linear array and mesh topologies, are discussed by Reingold [RND77]. Ranka and Sahni [RS90b] discuss the concepts of congestion, dilation, and expansion.

A comprehensive survey of cut-through routing techniques is provided by Ni and McKinley [NM93]. The wormhole routing technique was proposed by Dally and Seitz [DS86]. A related technique called *virtual cut-through*, in which communication buffers are provided at intermediate nodes, was described by Kermani and Kleinrock [KK79]. Dally and Seitz [DS87] discuss deadlock-free wormhole routing based on channel dependence graphs. Deterministic routing schemes based on dimension ordering are often used to avoid deadlocks. Cut-through routing has been used in several parallel computers. The E-cube routing scheme for hypercubes was proposed by [SB77].

Dally [Dal90b] discusses cost-performance tradeoffs of networks for message-passing computers. Using the bisection bandwidth of a network as a measure of the cost of the network, he shows that low-dimensional networks (such as two-dimensional meshes) are more cost-effective than high-dimensional networks (such as hypercubes) [Dal87, Dal90b, Dal90a]. Kreeger and Vempaty [KV92] derive the bandwidth equalization factor for a mesh with respect to a hypercube-connected computer for all-to-all personalized communication (Section 4.5). Gupta and Kumar [GK93b] analyze the cost-performance tradeoffs of FFT computations on mesh and hypercube networks.

The properties of PRAMs have been studied extensively [FW78, KR88, LY86, Sni82, Sni85]. Books by Akl [Akl89], Gibbons [GR90], and Jaja [Jaj92] address PRAM algorithms. Our discussion of PRAM is based upon the book by Jaja [Jaj92]. A number of processor networks have been proposed to simulate PRAM models [AHMP87, HP89, LPP88, LPP89, MV84, Upf84, UW84]. Mehlhorn and Vishkin [MV84] propose the *module parallel computer* (MPC) to simulate PRAM models. The MPC is a message-passing parallel computer composed of p processors, each with a fixed amount of memory and connected by a completely-connected network. The MPC is capable of probabilistically simulating T steps of a PRAM in $T \log p$ steps if the total memory is increased by a factor of $\log p$. The main drawback of the MPC model is that a completely-connected network is difficult to construct for a large number of processors. Alt *et al.* [AHMP87] propose another model called the *bounded-degree network* (BDN). In this network, each processor is connected to a fixed number of other processors. Karlin and Upfal [KU86] describe an $O(T \log p)$ time probabilistic simulation of a PRAM on a BDN. Hornick and Preparata [HP89] propose a bipartite network that connects sets of processors and memory pools. They investigate both the message-passing MPC and BDN based on a mesh of trees.

Many modifications of the PRAM model have been proposed that attempt to bring it closer to practical parallel computers. Aggarwal, Chandra, and Snir [ACS89b] propose the LPRAM (local-memory PRAM) model and the BPRAM (block PRAM) model [ACS89b]. They also introduce a hierarchical memory model of computation [ACS89a]. In this model, memory units at different levels are accessed in different times. Parallel algorithms for this model induce locality by bringing data into faster memory units before using them and returning them to the slower memory units. Other PRAM models such as phase PRAM [Gib89], XPRAM [Val90b], and the delay model [PY88] have also been proposed. Many researchers have investigated abstract universal models for parallel computers [CKP+93a, Sny86, Val90a]. Models such as BSP [Val90a], Postal model [BNK92], LogP [CKP+93b], A^3 [GKRS96], C^3 [HK96], CGM [DFRC96], and QSM [Ram97] have been proposed with similar objectives.

Problems

2.1 Design an experiment (i.e., design and write programs and take measurements) to determine the memory bandwidth of your computer and to estimate the caches at various levels of the hierarchy. Use this experiment to estimate the bandwidth and L1 cache of your computer. Justify your answer. (Hint: To test bandwidth, you do not want reuse. To test cache size, you want reuse to see the effect of the cache and to increase this size until the reuse decreases sharply.)

2.2 Consider a memory system with a level 1 cache of 32 KB and DRAM of 512 MB with the processor operating at 1 GHz. The latency to L1 cache is one cycle and the latency to DRAM is 100 cycles. In each memory cycle, the processor fetches four words (cache line size is four words). What is the peak achievable performance of a dot product of two vectors? Note: Where necessary, assume an optimal cache placement policy.

```
1         /* dot product loop */
2         for (i = 0; i < dim; i++)
3                 dot_prod += a[i] * b[i];
```

2.3 Now consider the problem of multiplying a dense matrix with a vector using a two-loop dot-product formulation. The matrix is of dimension 4*K* x 4*K*. (Each row of the matrix takes 16 KB of storage.) What is the peak achievable performance of this technique using a two-loop dot-product based matrix-vector product?

```
1         /* matrix-vector product loop */
2         for (i = 0; i < dim; i++)
3                 for (j = 0; i < dim; j++)
4                         c[i] += a[i][j] * b[j];
```

2.4 Extending this further, consider the problem of multiplying two dense matrices of dimension 4*K* x 4*K*. What is the peak achievable performance using a three-loop dot-product based formulation? (Assume that matrices are laid out in a row-major fashion.)

```
1         /* matrix-matrix product loop */
2         for (i = 0; i < dim; i++)
3                 for (j = 0; i < dim; j++)
4                         for (k = 0; k < dim; k++)
5                                 c[i][j] += a[i][k] * b[k][j];
```

2.5 Restructure the matrix multiplication algorithm to achieve better cache performance. The most obvious cause of the poor performance of matrix multiplication was the absence of spatial locality. In some cases, we were wasting three of the four words fetched from memory. To fix this problem, we compute the elements of the result matrix four at a time. Using this approach, we can increase our FLOP count with a simple restructuring of the program. However, it is possible to achieve much higher performance from this problem. This is possible by viewing the matrix multiplication problem as a cube in which each internal grid point corresponds to a multiply-add operation. Matrix multiplication algorithms traverse this cube in different ways, which induce different partitions of the cube. The data required for computing a partition grows as the surface area of the input

faces of the partition and the computation as the volume of the partition. For the algorithms discussed above, we were slicing thin partitions of the cube for which the area and volume were comparable (thus achieving poor cache performance). To remedy this, we restructure the computation by partitioning the cube into subcubes of size k x k x k. The data associated with this is 3 x k^2 (k^2 data for each of the three matrices) and the computation is k^3. To maximize performance, we would like 3 x k^2 to be equal to 8K since that is the amount of cache available (assuming the same machine parameters as in Problem 2.2). This corresponds to k = 51. The computation associated with a cube of this dimension is 132651 multiply-add operations or 265302 FLOPs. To perform this computation, we needed to fetch two submatrices of size 51 x 51. This corresponds to 5202 words or 1301 cache lines. Accessing these cache lines takes 130100 ns. Since 265302 FLOPs are performed in 130100 ns, the peak computation rate of this formulation is 2.04 GFLOPS. Code this example and plot the performance as a function of k. (Code on any conventional microprocessor. Make sure you note the clock speed, the microprocessor and the cache available at each level.)

2.6 Consider an SMP with a distributed shared-address-space. Consider a simple cost model in which it takes 10 ns to access local cache, 100 ns to access local memory, and 400 ns to access remote memory. A parallel program is running on this machine. The program is perfectly load balanced with 80% of all accesses going to local cache, 10% to local memory, and 10% to remote memory. What is the effective memory access time for this computation? If the computation is memory bound, what is the peak computation rate?

Now consider the same computation running on one processor. Here, the processor hits the cache 70% of the time and local memory 30% of the time. What is the effective peak computation rate for one processor? What is the fractional computation rate of a processor in a parallel configuration as compared to the serial configuration?

Hint: Notice that the cache hit for multiple processors is higher than that for one processor. This is typically because the aggregate cache available on multiprocessors is larger than on single processor systems.

2.7 What are the major differences between message-passing and shared-address-space computers? Also outline the advantages and disadvantages of the two.

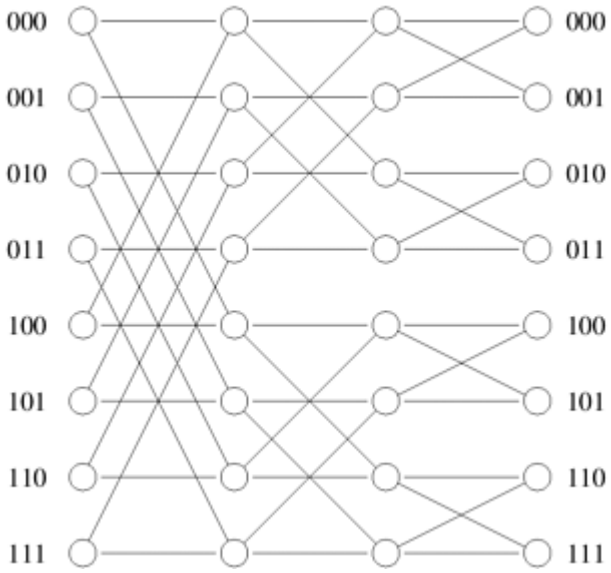
2.8 Why is it difficult to construct a true shared-memory computer? What is the minimum number of switches for connecting p processors to a shared memory with b words (where each word can be accessed independently)?

2.9 Of the four PRAM models (EREW, CREW, ERCW, and CRCW), which model is the most powerful? Why?

2.10 [Lei92] The *Butterfly network* is an interconnection network composed of log p levels (as the omega network). In a Butterfly network, each switching node i at a level l is connected to the identically numbered element at level l + 1 and to a switching node whose number differs from itself only at the lth most significant bit. Therefore, switching node S_i is connected to element S_j at level l if $j = i$ or $j = i \oplus (2^{\log p - l})$.

Figure 2.34 illustrates a Butterfly network with eight processing nodes. Show the equivalence of a Butterfly network and an omega network.

Figure 2.34. A Butterfly network with eight processing nodes.



Hint: Rearrange the switches of an omega network so that it looks like a Butterfly network.

2.11 Consider the omega network described in Section 2.4.3. As shown there, this network is a blocking network (that is, a processor that uses the network to access a memory location might prevent another processor from accessing another memory location). Consider an omega network that connects p processors. Define a function f that maps $P = [0, 1, \ldots, p - 1]$ onto a permutation P' of P (that is, $P'[i] = f(P[i])$ and $P'[i] \in P$ for all $0 \leq i < p$). Think of this function as mapping communication requests by the processors so that processor $P[i]$ requests communication with processor $P'[i]$.

1. How many distinct permutation functions exist?
2. How many of these functions result in non-blocking communication?
3. What is the probability that an arbitrary function will result in non-blocking communication?

2.12 A cycle in a graph is defined as a path originating and terminating at the same node. The length of a cycle is the number of edges in the cycle. Show that there are no odd-length cycles in a d-dimensional hypercube.

2.13 The labels in a d-dimensional hypercube use d bits. Fixing any k of these bits, show that the nodes whose labels differ in the remaining $d - k$ bit positions form a $(d - k)$-dimensional subcube composed of $2^{(d-k)}$ nodes.

2.14 Let A and B be two nodes in a d-dimensional hypercube. Define $H(A, B)$ to be the Hamming distance between A and B, and $P(A, B)$ to be the number of distinct paths connecting A and B. These paths are called *parallel paths* and have no common nodes other than A and B. Prove the following:

1. The minimum distance in terms of communication links between A and B is given by $H(A, B)$.
2. The total number of parallel paths between any two nodes is $P(A, B) = d$.

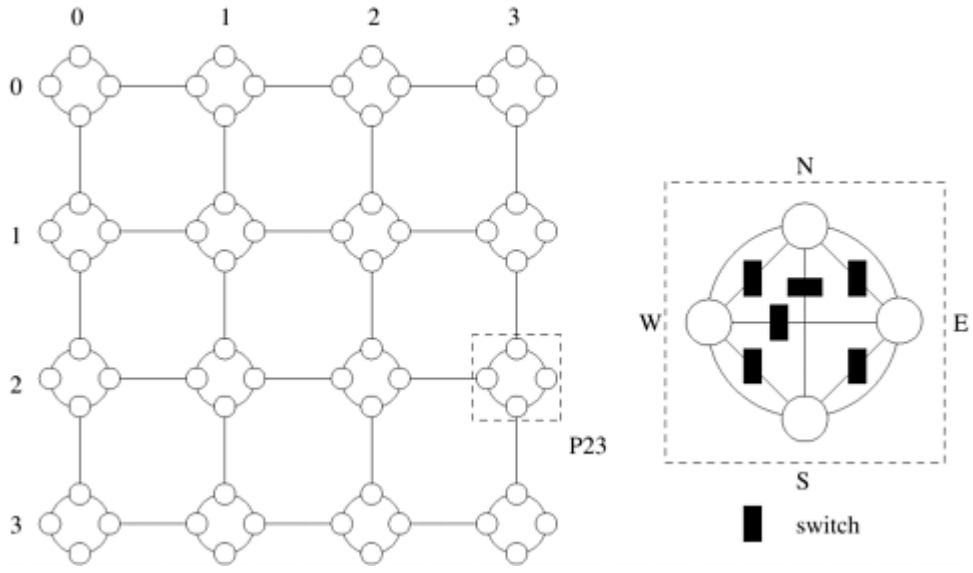
3. The number of parallel paths between A and B of length $H(A, B)$ is $P_{length=H(A,B)}(A, B) = H(A, B)$.

4. The length of the remaining $d - H(A, B)$ parallel paths is $H(A, B) + 2$.

2.15 In the informal derivation of the bisection width of a hypercube, we used the construction of a hypercube to show that a d-dimensional hypercube is formed from two $(d - 1)$-dimensional hypercubes. We argued that because corresponding nodes in each of these subcubes have a direct communication link, there are $2^d - 1$ links across the partition. However, it is possible to partition a hypercube into two parts such that neither of the partitions is a hypercube. Show that any such partitions will have more than $2^d - 1$ direct links between them.

2.16 [MKRS88] A $\sqrt{p} \times \sqrt{p}$ *reconfigurable mesh* consists of a $\sqrt{p} \times \sqrt{p}$ array of processing nodes connected to a grid-shaped reconfigurable broadcast bus. A 4 x 4 reconfigurable mesh is shown in Figure 2.35. Each node has locally-controllable bus switches. The internal connections among the four ports, north (N), east (E), west (W), and south (S), of a node can be configured during the execution of an algorithm. Note that there are 15 connection patterns. For example, {SW, EN} represents the configuration in which port S is connected to port W and port N is connected to port E. Each bit of the bus carries one of *1-signal* or *0-signal* at any time. The switches allow the broadcast bus to be divided into subbuses, providing smaller reconfigurable meshes. For a given set of switch settings, a *subbus* is a maximally-connected subset of the nodes. Other than the buses and the switches, the reconfigurable mesh is similar to the standard two-dimensional mesh. Assume that only one node is allowed to broadcast on a *subbus* shared by multiple nodes at any time.

Figure 2.35. Switch connection patterns in a reconfigurable mesh.

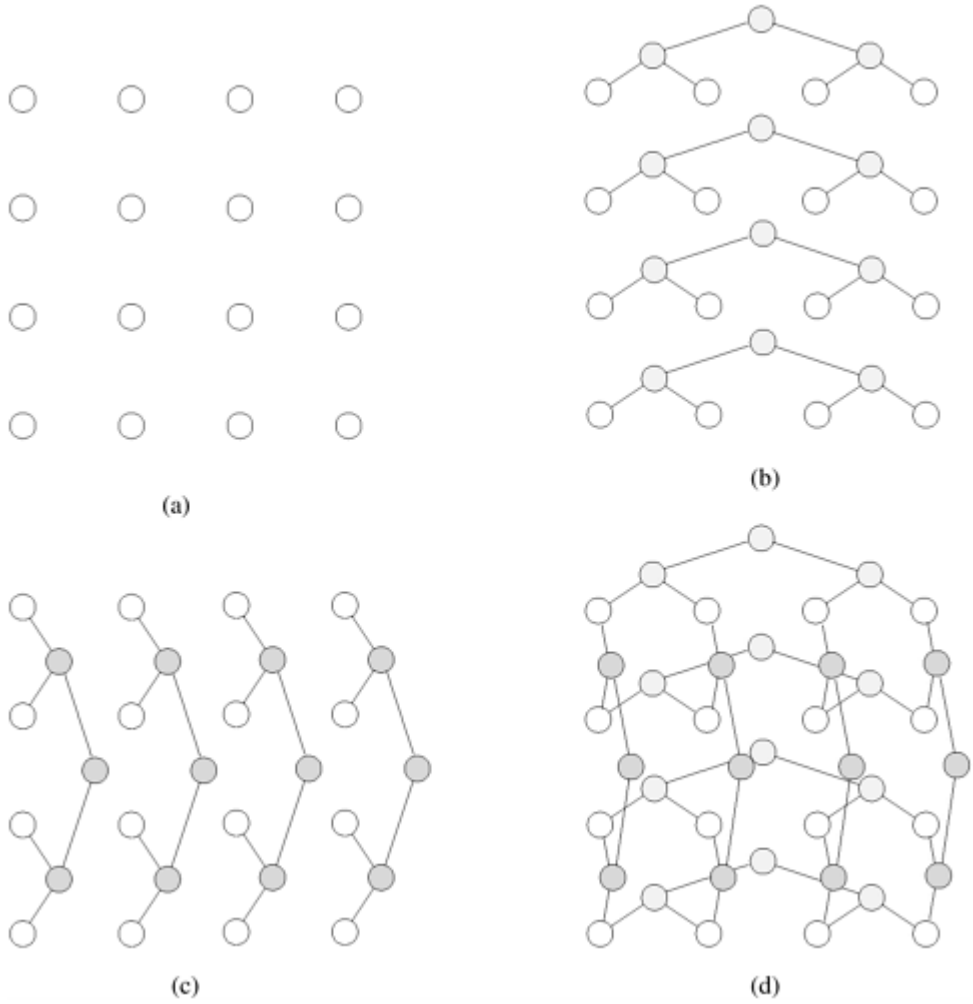


Determine the bisection width, the diameter, and the number of switching nodes and communication links for a reconfigurable mesh of $\sqrt{p} \times \sqrt{p}$ processing nodes. What are the advantages and disadvantages of a reconfigurable mesh as compared to a wraparound mesh?

2.17 [Lei92] A *mesh of trees* is a network that imposes a tree interconnection on a grid of processing nodes. A $\sqrt{p} \times \sqrt{p}$ mesh of trees is constructed as follows. Starting with a $\sqrt{p} \times \sqrt{p}$ grid, a complete binary tree is imposed on each row of the grid. Then a

complete binary tree is imposed on each column of the grid. Figure 2.36 illustrates the construction of a 4 x 4 mesh of trees. Assume that the nodes at intermediate levels are switching nodes. Determine the bisection width, diameter, and total number of switching nodes in a $\sqrt{p} \times \sqrt{p}$ mesh.

Figure 2.36. The construction of a 4 x 4 mesh of trees: (a) a 4 x 4 grid, (b) complete binary trees imposed over individual rows, (c) complete binary trees imposed over each column, and (d) the complete 4 x 4 mesh of trees.



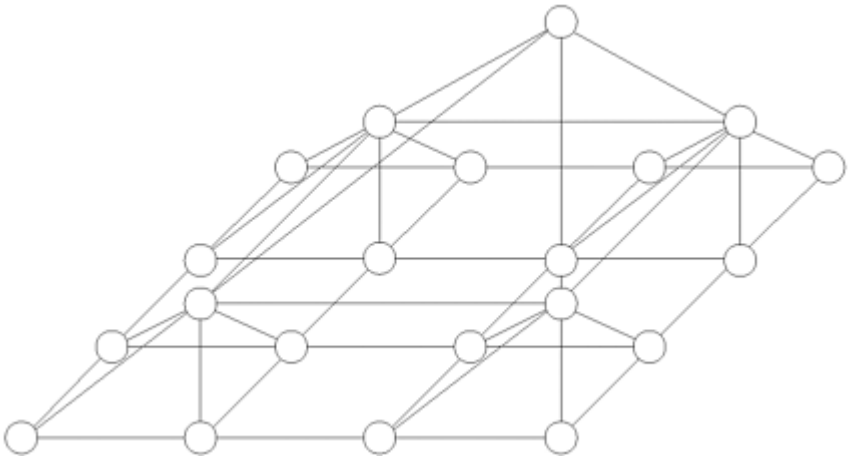
2.18 [Lei92] Extend the two-dimensional mesh of trees (Problem 2.17) to *d* dimensions to construct a $p^{1/d}$ x $p^{1/d}$ x ⋯ x $p^{1/d}$ mesh of trees. We can do this by fixing grid positions in all dimensions to different values and imposing a complete binary tree on the one dimension that is being varied.

Derive the total number of switching nodes in a $p^{1/d}$ x $p^{1/d}$ x ⋯ x $p^{1/d}$ mesh of trees. Calculate the diameter, bisection width, and wiring cost in terms of the total number of wires. What are the advantages and disadvantages of a mesh of trees as compared to a wraparound mesh?

2.19 [Lei92] A network related to the mesh of trees is the *d*-dimensional *pyramidal mesh*. A *d*-dimensional pyramidal mesh imposes a pyramid on the underlying grid of processing nodes (as opposed to a complete tree in the mesh of trees). The generalization

is as follows. In the mesh of trees, all dimensions except one are fixed and a tree is imposed on the remaining dimension. In a pyramid, all but two dimensions are fixed and a pyramid is imposed on the mesh formed by these two dimensions. In a tree, each node i at level k is connected to node $i/2$ at level k - 1. Similarly, in a pyramid, a node (i, j) at level k is connected to a node $(i/2, j/2)$ at level k - 1. Furthermore, the nodes at each level are connected in a mesh. A two-dimensional pyramidal mesh is illustrated in Figure 2.37.

Figure 2.37. A 4 x 4 pyramidal mesh.



For a $\sqrt{p} \times \sqrt{p}$ pyramidal mesh, assume that the intermediate nodes are switching nodes, and derive the diameter, bisection width, arc connectivity, and cost in terms of the number of communication links and switching nodes. What are the advantages and disadvantages of a pyramidal mesh as compared to a mesh of trees?

2.20 [Lei92] One of the drawbacks of a hypercube-connected network is that different wires in the network are of different lengths. This implies that data takes different times to traverse different communication links. It appears that two-dimensional mesh networks with wraparound connections suffer from this drawback too. However, it is possible to fabricate a two-dimensional wraparound mesh using wires of fixed length. Illustrate this layout by drawing such a 4 x 4 wraparound mesh.

2.21 Show how to embed a p-node three-dimensional mesh into a p-node hypercube. What are the allowable values of p for your embedding?

2.22 Show how to embed a p-node mesh of trees into a p-node hypercube.

2.23 Consider a complete binary tree of 2^d - 1 nodes in which each node is a processing node. What is the minimum-dilation mapping of such a tree onto a d-dimensional hypercube?

2.24 The concept of a *minimum congestion mapping* is very useful. Consider two parallel computers with different interconnection networks such that a congestion-r mapping of the first into the second exists. Ignoring the dilation of the mapping, if each communication link in the second computer is more than r times faster than the first computer, the second computer is strictly superior to the first.

Now consider mapping a d-dimensional hypercube onto a 2^d-node mesh. Ignoring the dilation of the mapping, what is the minimum-congestion mapping of the hypercube onto the mesh? Use this result to determine whether a 1024-node mesh with communication links operating at 25 million bytes per second is strictly better than a 1024-node hypercube (whose nodes are identical to those used in the mesh) with communication links

operating at two million bytes per second.

2.25 Derive the diameter, number of links, and bisection width of a k-ary d-cube with p nodes. Define l_{av} to be the average distance between any two nodes in the network. Derive l_{av} for a k-ary d-cube.

2.26 Consider the routing of messages in a parallel computer that uses store-and-forward routing. In such a network, the cost of sending a single message of size m from P_{source} to $P_{destination}$ via a path of length d is $t_s + t_w \times d \times m$. An alternate way of sending a message of size m is as follows. The user breaks the message into k parts each of size m/k, and then sends these k distinct messages one by one from P_{source} to $P_{destination}$. For this new method, derive the expression for time to transfer a message of size m to a node d hops away under the following two cases:

1. Assume that another message can be sent from P_{source} as soon as the previous message has reached the next node in the path.
2. Assume that another message can be sent from P_{source} only after the previous message has reached $P_{destination}$.

For each case, comment on the value of this expression as the value of k varies between 1 and m. Also, what is the optimal value of k if t_s is very large, or if $t_s = 0$?

2.27 Consider a hypercube network of p nodes. Assume that the channel width of each communication link is one. The channel width of the links in a k-ary d-cube (for $d < \log p$) can be increased by equating the cost of this network with that of a hypercube. Two distinct measures can be used to evaluate the cost of a network.

1. The cost can be expressed in terms of the total number of wires in the network (the total number of wires is a product of the number of communication links and the channel width).
2. The bisection bandwidth can be used as a measure of cost.

Using each of these cost metrics and equating the cost of a k-ary d-cube with a hypercube, what is the channel width of a k-ary d-cube with an identical number of nodes, channel rate, and cost?

2.28 The results from Problems 2.25 and 2.27 can be used in a cost-performance analysis of static interconnection networks. Consider a k-ary d-cube network of p nodes with cut-through routing. Assume a hypercube-connected network of p nodes with channel width one. The channel width of other networks in the family is scaled up so that their cost is identical to that of the hypercube. Let s and s' be the scaling factors for the channel width derived by equating the costs specified by the two cost metrics in Problem 2.27.

For each of the two scaling factors s and s', express the average communication time between any two nodes as a function of the dimensionality (d)of a k-ary d-cube and the number of nodes. Plot the communication time as a function of the dimensionality for p = 256, 512, and 1024, message size m = 512 bytes, t_s = 50.0μs, and $t_h = t_w$ = 0.5μs (for the hypercube). For these values of p and m, what is the dimensionality of the network that yields the best performance for a given cost?

2.29 Repeat Problem 2.28 for a k-ary d-cube with store-and-forward routing.

Chapter 3. Principles of Parallel Algorithm Design

Algorithm development is a critical component of problem solving using computers. A sequential algorithm is essentially a recipe or a sequence of basic steps for solving a given problem using a serial computer. Similarly, a parallel algorithm is a recipe that tells us how to solve a given problem using multiple processors. However, specifying a parallel algorithm involves more than just specifying the steps. At the very least, a parallel algorithm has the added dimension of concurrency and the algorithm designer must specify sets of steps that can be executed simultaneously. This is essential for obtaining any performance benefit from the use of a parallel computer. In practice, specifying a nontrivial parallel algorithm may include some or all of the following:

- Identifying portions of the work that can be performed concurrently.
- Mapping the concurrent pieces of work onto multiple processes running in parallel.
- Distributing the input, output, and intermediate data associated with the program.
- Managing accesses to data shared by multiple processors.
- Synchronizing the processors at various stages of the parallel program execution.

Typically, there are several choices for each of the above steps, but usually, relatively few combinations of choices lead to a parallel algorithm that yields performance commensurate with the computational and storage resources employed to solve the problem. Often, different choices yield the best performance on different parallel architectures or under different parallel programming paradigms.

In this chapter, we methodically discuss the process of designing and implementing parallel algorithms. We shall assume that the onus of providing a complete description of a parallel algorithm or program lies on the programmer or the algorithm designer. Tools and compilers for automatic parallelization at the current state of the art seem to work well only for highly structured programs or portions of programs. Therefore, we do not consider these in this chapter or elsewhere in this book.

3.1 Preliminaries

Dividing a computation into smaller computations and assigning them to different processors for parallel execution are the two key steps in the design of parallel algorithms. In this section, we present some basic terminology and introduce these two key steps in parallel algorithm design using matrix-vector multiplication and database query processing as examples.

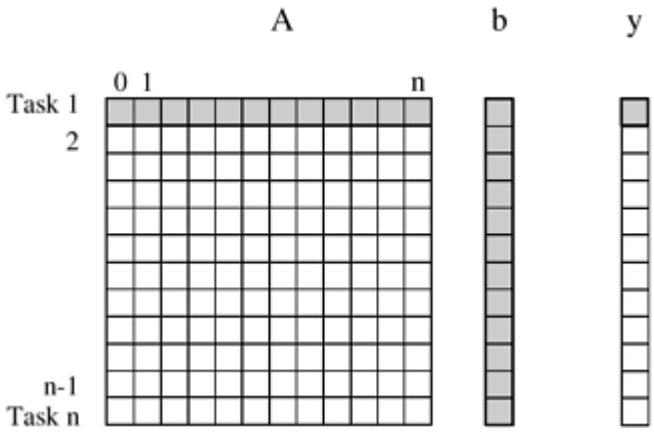
3.1.1 Decomposition, Tasks, and Dependency Graphs

The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel, is called *decomposition*. *Tasks* are programmer-defined units of computation into which the main computation is subdivided by means of decomposition. Simultaneous execution of multiple tasks is the key to reducing the time required to solve the entire problem. Tasks can be of arbitrary size, but once defined, they are regarded as indivisible units of computation. The tasks into which a problem is decomposed may not all be of the same size.

Example 3.1 Dense matrix-vector multiplication

Consider the multiplication of a dense $n \times n$ matrix A with a vector b to yield another vector y. The ith element $y[i]$ of the product vector is the dot-product of the ith row of A with the input vector b; i.e., $y[i] = \sum_{j=1}^{n} A[i, j].b[j]$. As shown later in Figure 3.1, the computation of each $y[i]$ can be regarded as a task. Alternatively, as shown later in Figure 3.4, the computation could be decomposed into fewer, say four, tasks where each task computes roughly $n/4$ of the entries of the vector y. ■

Figure 3.1. Decomposition of dense matrix-vector multiplication into n tasks, where n is the number of rows in the matrix. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.



Note that all tasks in Figure 3.1 are independent and can be performed all together or in any sequence. However, in general, some tasks may use data produced by other tasks and thus may need to wait for these tasks to finish execution. An abstraction used to express such dependencies among tasks and their relative order of execution is known as a *task-dependency graph*. A task-dependency graph is a directed acyclic graph in which the nodes represent tasks and the directed edges indicate the dependencies amongst them. The task corresponding to a node can be executed when all tasks connected to this node by incoming edges have completed. Note that task-dependency graphs can be disconnected and the edge-set of a task-dependency graph can be empty. This is the case for matrix-vector multiplication, where each task computes a subset of the entries of the product vector. To see a more interesting task-dependency graph, consider the following database query processing example.

Example 3.2 Database query processing

Table 3.1 shows a relational database of vehicles. Each row of the table is a record that contains data corresponding to a particular vehicle, such as its ID, model, year, color, etc. in various fields. Consider the computations performed in processing the following query:

```
MODEL="Civic" AND YEAR="2001" AND (COLOR="Green" OR COLOR="White")
```

This query looks for all 2001 Civics whose color is either Green or White. On a relational database, this query is processed by creating a number of intermediate tables. One possible way is to first create the following four tables: a table containing all Civics, a table containing all 2001-model cars, a table containing all green-colored cars, and a table containing all white-colored cars. Next, the computation proceeds by combining these tables by computing their pairwise intersections or unions. In particular, it computes the intersection of the Civic-table with the 2001-model year table, to construct a table of all 2001-model Civics. Similarly, it computes the union of the green- and white-colored tables to compute a table storing all cars whose color is either green or white. Finally, it computes the intersection of the table containing all the 2001 Civics with the table containing all the green or white vehicles, and returns the desired list. ■

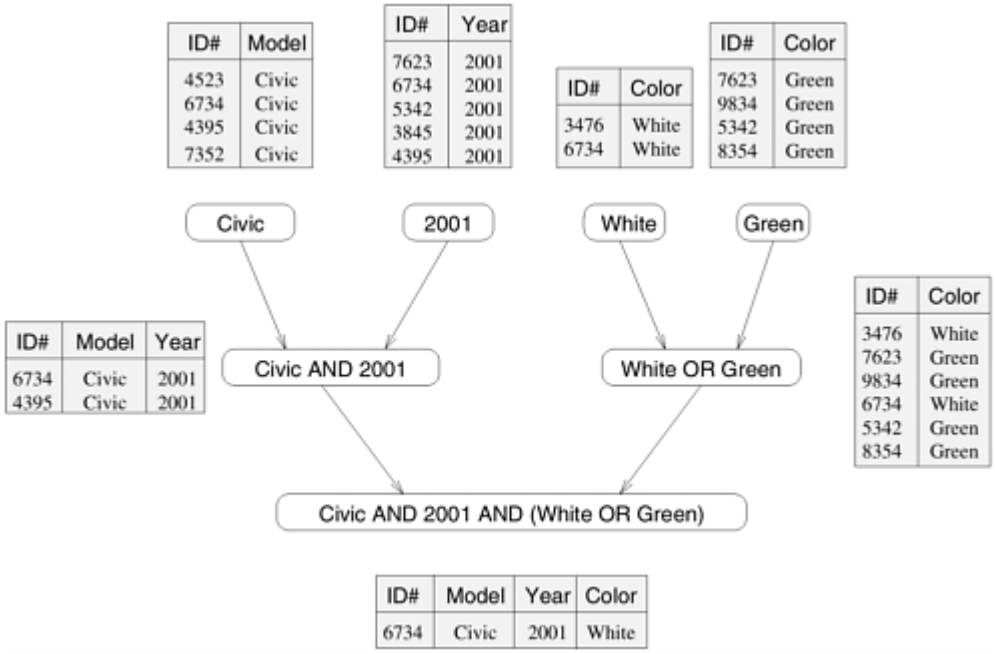
Table 3.1. A database storing information about used vehicles.

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000

ID#	Model	Year	Color	Dealer	Price
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

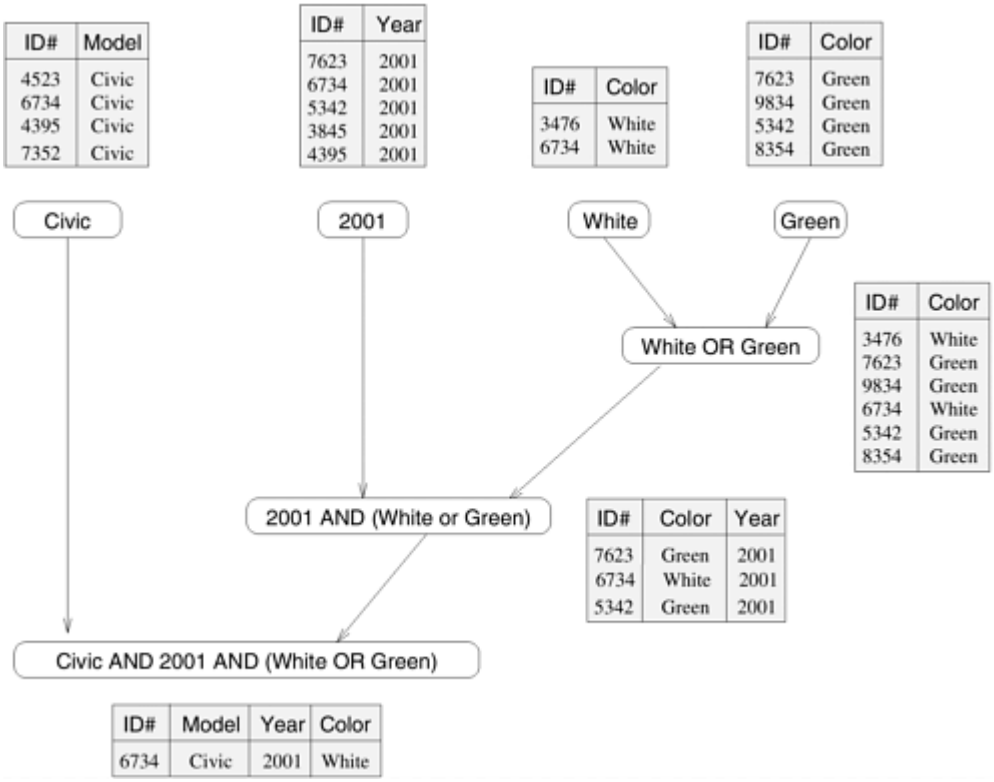
The various computations involved in processing the query in Example 3.2 can be visualized by the task-dependency graph shown in Figure 3.2. Each node in this figure is a task that corresponds to an intermediate table that needs to be computed and the arrows between nodes indicate dependencies between the tasks. For example, before we can compute the table that corresponds to the 2001 Civics, we must first compute the table of all the Civics and a table of all the 2001-model cars.

Figure 3.2. The different tables and their dependencies in a query processing operation.



Note that often there are multiple ways of expressing certain computations, especially those involving associative operators such as addition, multiplication, and logical AND or OR. Different ways of arranging computations can lead to different task-dependency graphs with different characteristics. For instance, the database query in Example 3.2 can be solved by first computing a table of all green or white cars, then performing an intersection with a table of all 2001 model cars, and finally combining the results with the table of all Civics. This sequence of computation results in the task-dependency graph shown in Figure 3.3.

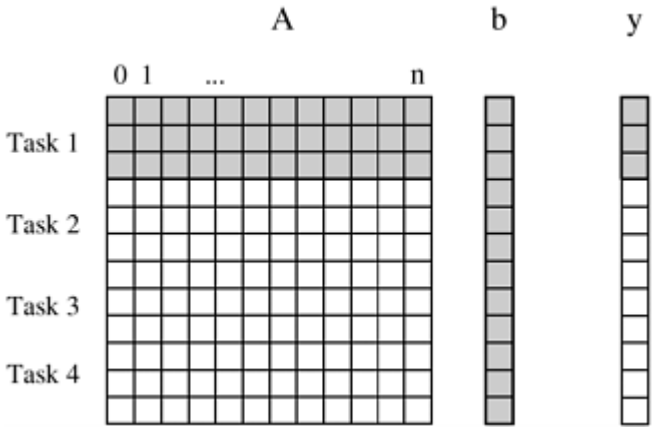
Figure 3.3. An alternate data-dependency graph for the query processing operation.



3.1.2 Granularity, Concurrency, and Task-Interaction

The number and size of tasks into which a problem is decomposed determines the *granularity* of the decomposition. A decomposition into a large number of small tasks is called *fine-grained* and a decomposition into a small number of large tasks is called *coarse-grained*. For example, the decomposition for matrix-vector multiplication shown in Figure 3.1 would usually be considered fine-grained because each of a large number of tasks performs a single dot-product. Figure 3.4 shows a coarse-grained decomposition of the same problem into four tasks, where each tasks computes $n/4$ of the entries of the output vector of length n.

Figure 3.4. Decomposition of dense matrix-vector multiplication into four tasks. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.



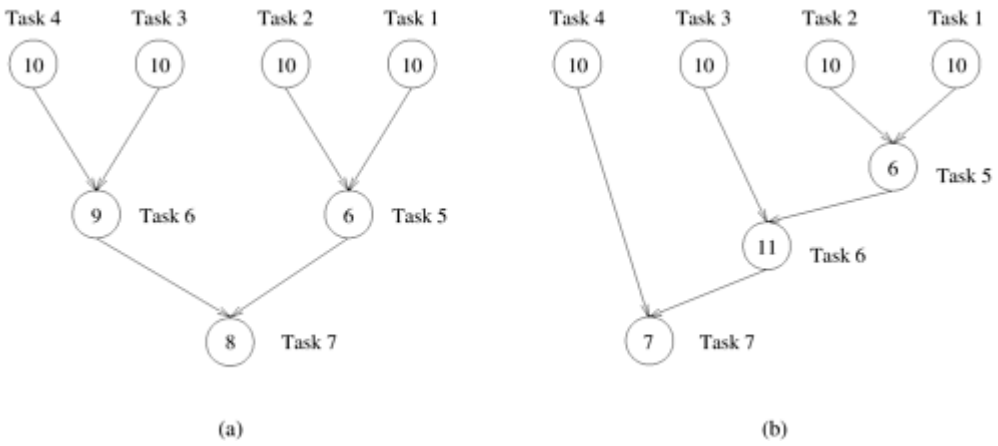
A concept related to granularity is that of *degree of concurrency*. The maximum number of tasks that can be executed simultaneously in a parallel program at any given time is known as its *maximum degree of concurrency*. In most cases, the maximum degree of concurrency is less than the total number of tasks due to dependencies among the tasks. For example, the maximum degree of concurrency in the task-graphs of Figures 3.2 and 3.3 is four. In these task-graphs, maximum concurrency is available right at the beginning when tables for Model, Year, Color Green, and Color White can be computed simultaneously. In general, for task-dependency graphs that are trees, the maximum degree of concurrency is always equal to the number of leaves in the tree.

A more useful indicator of a parallel program's performance is the *average degree of concurrency*, which is the average number of tasks that can run concurrently over the entire duration of execution of the program.

Both the maximum and the average degrees of concurrency usually increase as the granularity of tasks becomes smaller (finer). For example, the decomposition of matrix-vector multiplication shown in Figure 3.1 has a fairly small granularity and a large degree of concurrency. The decomposition for the same problem shown in Figure 3.4 has a larger granularity and a smaller degree of concurrency.

The degree of concurrency also depends on the shape of the task-dependency graph and the same granularity, in general, does not guarantee the same degree of concurrency. For example, consider the two task graphs in Figure 3.5, which are abstractions of the task graphs of Figures 3.2 and 3.3, respectively (Problem 3.1). The number inside each node represents the amount of work required to complete the task corresponding to that node. The average degree of concurrency of the task graph in Figure 3.5(a) is 2.33 and that of the task graph in Figure 3.5(b) is 1.88 (Problem 3.1), although both task-dependency graphs are based on the same decomposition.

Figure 3.5. Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.



A feature of a task-dependency graph that determines the average degree of concurrency for a given granularity is its *critical path*. In a task-dependency graph, let us refer to the nodes with no incoming edges by *start nodes* and the nodes with no outgoing edges by *finish nodes*. The longest directed path between any pair of start and finish nodes is known as the critical path. The sum of the weights of nodes along this path is known as the *critical path length*, where the weight of a node is the size or the amount of work associated with the corresponding task. The ratio of the total amount of work to the critical-path length is the average degree of concurrency. Therefore, a shorter critical path favors a higher degree of concurrency. For example, the critical path length is 27 in the task-dependency graph shown in Figure 3.5(a) and

is 34 in the task-dependency graph shown in Figure 3.5(b). Since the total amount of work required to solve the problems using the two decompositions is 63 and 64, respectively, the average degree of concurrency of the two task-dependency graphs is 2.33 and 1.88, respectively.

Although it may appear that the time required to solve a problem can be reduced simply by increasing the granularity of decomposition and utilizing the resulting concurrency to perform more and more tasks in parallel, this is not the case in most practical scenarios. Usually, there is an inherent bound on how fine-grained a decomposition a problem permits. For instance, there are n^2 multiplications and additions in matrix-vector multiplication considered in Example 3.1 and the problem cannot be decomposed into more than $O(n^2)$ tasks even by using the most fine-grained decomposition.

Other than limited granularity and degree of concurrency, there is another important practical factor that limits our ability to obtain unbounded speedup (ratio of serial to parallel execution time) from parallelization. This factor is the *interaction* among tasks running on different physical processors. The tasks that a problem is decomposed into often share input, output, or intermediate data. The dependencies in a task-dependency graph usually result from the fact that the output of one task is the input for another. For example, in the database query example, tasks share intermediate data; the table generated by one task is often used by another task as input. Depending on the definition of the tasks and the parallel programming paradigm, there may be interactions among tasks that appear to be independent in a task-dependency graph. For example, in the decomposition for matrix-vector multiplication, although all tasks are independent, they all need access to the entire input vector *b*. Since originally there is only one copy of the vector *b*, tasks may have to send and receive messages for all of them to access the entire vector in the distributed-memory paradigm.

The pattern of interaction among tasks is captured by what is known as a *task-interaction graph*. The nodes in a task-interaction graph represent tasks and the edges connect tasks that interact with each other. The nodes and edges of a task-interaction graph can be assigned weights proportional to the amount of computation a task performs and the amount of interaction that occurs along an edge, if this information is known. The edges in a task-interaction graph are usually undirected, but directed edges can be used to indicate the direction of flow of data, if it is unidirectional. The edge-set of a task-interaction graph is usually a superset of the edge-set of the task-dependency graph. In the database query example discussed earlier, the task-interaction graph is the same as the task-dependency graph. We now give an example of a more interesting task-interaction graph that results from the problem of sparse matrix-vector multiplication.

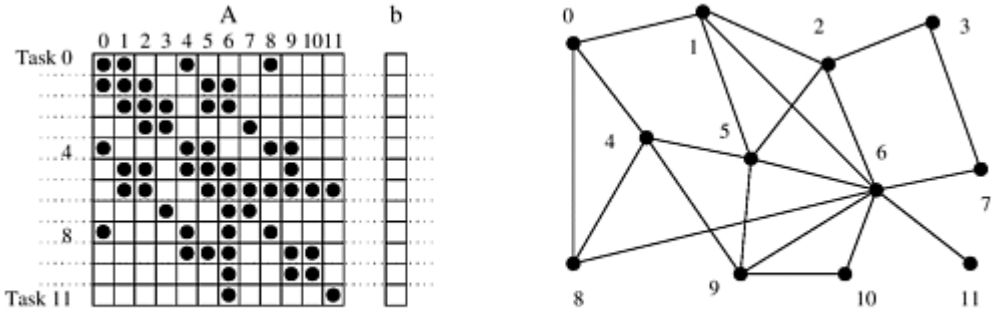
Example 3.3 Sparse matrix-vector multiplication

Consider the problem of computing the product $y = Ab$ of a sparse n x n matrix A with a dense n x 1 vector b. A matrix is considered sparse when a significant number of entries in it are zero and the locations of the non-zero entries do not conform to a predefined structure or pattern. Arithmetic operations involving sparse matrices can often be optimized significantly by avoiding computations involving the zeros. For instance, while computing the ith entry $y[i] = \sum_{j=1}^{n} (A[i, j] \times b[j])$ of the product vector, we need to compute the products $A[i, j] \times b[j]$ for only those values of j for which $A[i, j] \neq 0$. For example, $y[0] = A[0, 0].b[0] + A[0, 1].b[1] + A[0, 4].b[4] + A[0, 8].b[8]$.

One possible way of decomposing this computation is to partition the output vector y and have each task compute an entry in it. Figure 3.6(a) illustrates this

decomposition. In addition to assigning the computation of the element $y[i]$ of the output vector to Task i, we also make it the "owner" of row $A[i, *]$ of the matrix and the element $b[i]$ of the input vector. Note that the computation of $y[i]$ requires access to many elements of b that are owned by other tasks. So Task i must get these elements from the appropriate locations. In the message-passing paradigm, with the ownership of $b[i]$,Task i also inherits the responsibility of sending $b[i]$ to all the other tasks that need it for their computation. For example, Task 4 must send $b[4]$ to Tasks 0, 5, 8, and 9 and must get $b[0]$, $b[5]$, $b[8]$, and $b[9]$ to perform its own computation. The resulting task-interaction graph is shown in Figure 3.6(b). ■

Figure 3.6. A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task i computes $\sum_{0 \leq j \leq 11, A[i,j] \neq 0} A[i, j].b[j]$.



Chapter 5 contains detailed quantitative analysis of overheads due to interaction and limited concurrency and their effect on the performance and scalability of parallel algorithm-architecture combinations. In this section, we have provided a basic introduction to these factors because they require important consideration in designing parallel algorithms.

3.1.3 Processes and Mapping

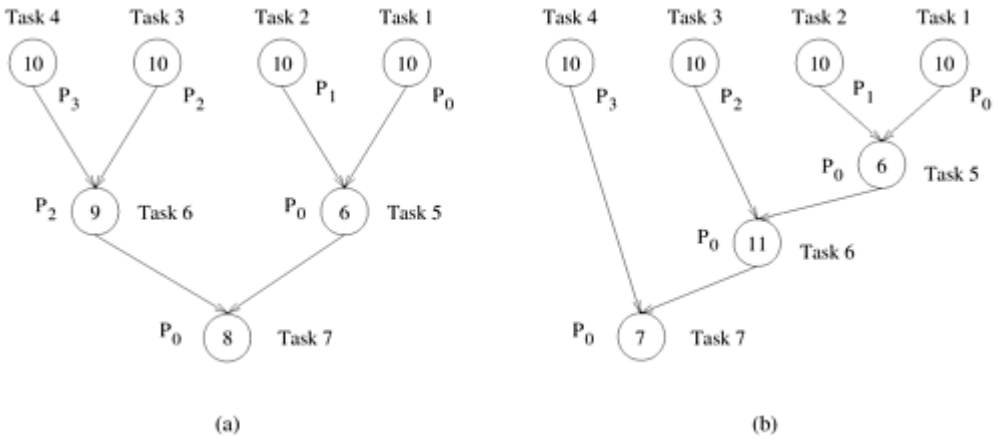
The tasks, into which a problem is decomposed, run on physical processors. However, for reasons that we shall soon discuss, we will use the term *process* in this chapter to refer to a processing or computing agent that performs tasks. In this context, the term process does not adhere to the rigorous operating system definition of a process. Instead, it is an abstract entity that uses the code and data corresponding to a task to produce the output of that task within a finite amount of time after the task is activated by the parallel program. During this time, in addition to performing computations, a process may synchronize or communicate with other processes, if needed. In order to obtain any speedup over a sequential implementation, a parallel program must have several processes active simultaneously, working on different tasks. The mechanism by which tasks are assigned to processes for execution is called *mapping*. For example, four processes could be assigned the task of computing one submatrix of C each in the matrix-multiplication computation of Example 3.5.

The task-dependency and task-interaction graphs that result from a choice of decomposition play an important role in the selection of a good mapping for a parallel algorithm. A good mapping should seek to maximize the use of concurrency by mapping independent tasks onto different processes, it should seek to minimize the total completion time by ensuring that processes are available to execute the tasks on the critical path as soon as such tasks become

executable, and it should seek to minimize interaction among processes by mapping tasks with a high degree of mutual interaction onto the same process. In most nontrivial parallel algorithms, these tend to be conflicting goals. For instance, the most efficient decomposition-mapping combination is a single task mapped onto a single process. It wastes no time in idling or interacting, but achieves no speedup either. Finding a balance that optimizes the overall parallel performance is the key to a successful parallel algorithm. Therefore, mapping of tasks onto processes plays an important role in determining how efficient the resulting parallel algorithm is. Even though the degree of concurrency is determined by the decomposition, it is the mapping that determines how much of that concurrency is actually utilized, and how efficiently.

For example, Figure 3.7 shows efficient mappings for the decompositions and the task-interaction graphs of Figure 3.5 onto four processes. Note that, in this case, a maximum of four processes can be employed usefully, although the total number of tasks is seven. This is because the maximum degree of concurrency is only four. The last three tasks can be mapped arbitrarily among the processes to satisfy the constraints of the task-dependency graph. However, it makes more sense to map the tasks connected by an edge onto the same process because this prevents an inter-task interaction from becoming an inter-processes interaction. For example, in Figure 3.7(b), if Task 5 is mapped onto process P_2, then both processes P_0 and P_1 will need to interact with P_2. In the current mapping, only a single interaction between P_0 and P_1 suffices.

Figure 3.7. Mappings of the task graphs of Figure 3.5 onto four processes.



3.1.4 Processes versus Processors

In the context of parallel algorithm design, processes are logical computing agents that perform tasks. Processors are the hardware units that physically perform computations. In this text, we choose to express parallel algorithms and programs in terms of processes. In most cases, when we refer to processes in the context of a parallel algorithm, there is a one-to-one correspondence between processes and processors and it is appropriate to assume that there are as many processes as the number of physical CPUs on the parallel computer. However, sometimes a higher level of abstraction may be required to express a parallel algorithm, especially if it is a complex algorithm with multiple stages or with different forms of parallelism.

Treating processes and processors separately is also useful when designing parallel programs for hardware that supports multiple programming paradigms. For instance, consider a parallel computer that consists of multiple computing nodes that communicate with each other via message passing. Now each of these nodes could be a shared-address-space module with

multiple CPUs. Consider implementing matrix multiplication on such a parallel computer. The best way to design a parallel algorithm is to do so in two stages. First, develop a decomposition and mapping strategy suitable for the message-passing paradigm and use this to exploit parallelism among the nodes. Each task that the original matrix multiplication problem decomposes into is a matrix multiplication computation itself. The next step is to develop a decomposition and mapping strategy suitable for the shared-memory paradigm and use this to implement each task on the multiple CPUs of a node.

PREVIOUS NEXT

3.2 Decomposition Techniques

As mentioned earlier, one of the fundamental steps that we need to undertake to solve a problem in parallel is to split the computations to be performed into a set of tasks for concurrent execution defined by the task-dependency graph. In this section, we describe some commonly used decomposition techniques for achieving concurrency. This is not an exhaustive set of possible decomposition techniques. Also, a given decomposition is not always guaranteed to lead to the best parallel algorithm for a given problem. Despite these shortcomings, the decomposition techniques described in this section often provide a good starting point for many problems and one or a combination of these techniques can be used to obtain effective decompositions for a large variety of problems.

These techniques are broadly classified as *recursive decomposition*, *data-decomposition*, *exploratory decomposition*, and *speculative decomposition*. The recursive- and data-decomposition techniques are relatively *general purpose* as they can be used to decompose a wide variety of problems. On the other hand, speculative- and exploratory-decomposition techniques are more of a *special purpose* nature because they apply to specific classes of problems.

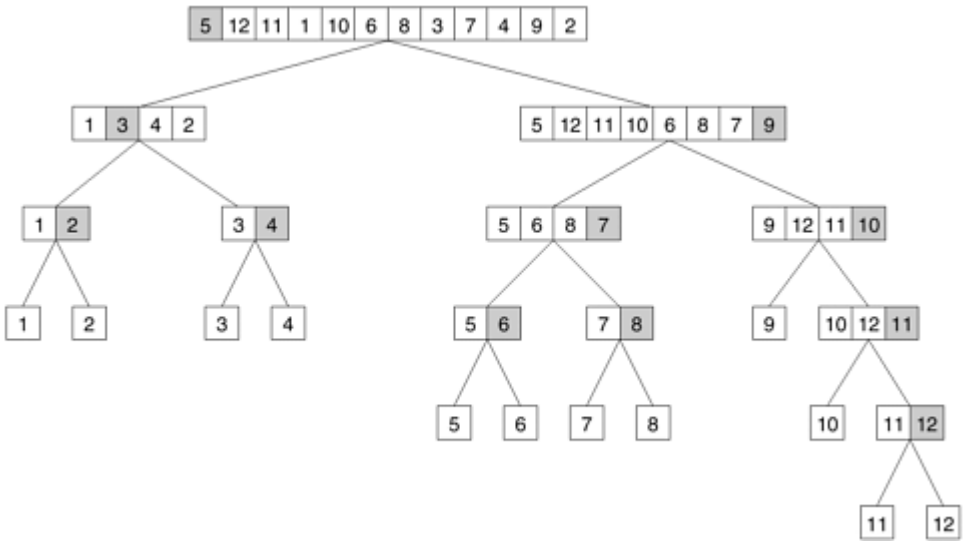
3.2.1 Recursive Decomposition

Recursive decomposition is a method for inducing concurrency in problems that can be solved using the divide-and-conquer strategy. In this technique, a problem is solved by first dividing it into a set of independent subproblems. Each one of these subproblems is solved by recursively applying a similar division into smaller subproblems followed by a combination of their results. The divide-and-conquer strategy results in natural concurrency, as different subproblems can be solved concurrently.

Example 3.4 Quicksort

Consider the problem of sorting a sequence A of n elements using the commonly used quicksort algorithm. Quicksort is a divide and conquer algorithm that starts by selecting a pivot element x and then partitions the sequence A into two subsequences A_0 and A_1 such that all the elements in A_0 are smaller than x and all the elements in A_1 are greater than or equal to x. This partitioning step forms the *divide* step of the algorithm. Each one of the subsequences A_0 and A_1 is sorted by recursively calling quicksort. Each one of these recursive calls further partitions the sequences. This is illustrated in Figure 3.8 for a sequence of 12 numbers. The recursion terminates when each subsequence contains only a single element. ■

Figure 3.8. The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.



In Figure 3.8, we define a task as the work of partitioning a given subsequence. Therefore, Figure 3.8 also represents the task graph for the problem. Initially, there is only one sequence (i.e., the root of the tree), and we can use only a single process to partition it. The completion of the root task results in two subsequences (A_0 and A_1, corresponding to the two nodes at the first level of the tree) and each one can be partitioned in parallel. Similarly, the concurrency continues to increase as we move down the tree.

Sometimes, it is possible to restructure a computation to make it amenable to recursive decomposition even if the commonly used algorithm for the problem is not based on the divide-and-conquer strategy. For example, consider the problem of finding the minimum element in an unordered sequence A of n elements. The serial algorithm for solving this problem scans the entire sequence A, recording at each step the minimum element found so far as illustrated in Algorithm 3.1. It is easy to see that this serial algorithm exhibits no concurrency.

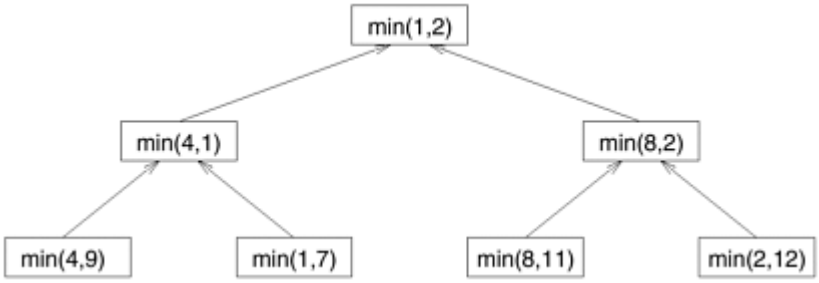
Algorithm 3.1 A serial program for finding the minimum in an array of numbers A of length n.

```
1.    procedure SERIAL_MIN (A, n)
2.    begin
3.    min = A[0];
4.    for i := 1 to n - 1 do
5.        if (A[i] < min) min := A[i];
6.    endfor;
7.    return min;
8.    end SERIAL_MIN
```

Once we restructure this computation as a divide-and-conquer algorithm, we can use recursive decomposition to extract concurrency. Algorithm 3.2 is a divide-and-conquer algorithm for finding the minimum element in an array. In this algorithm, we split the sequence A into two subsequences, each of size $n/2$, and we find the minimum for each of these subsequences by performing a recursive call. Now the overall minimum element is found by selecting the minimum of these two subsequences. The recursion terminates when there is only one element left in each subsequence. Having restructured the serial computation in this manner, it is easy to construct a task-dependency graph for this problem. Figure 3.9 illustrates such a task-dependency graph for finding the minimum of eight numbers where each task is assigned the

work of finding the minimum of two numbers.

Figure 3.9. The task-dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}. Each node in the tree represents the task of finding the minimum of a pair of numbers.



Algorithm 3.2 A recursive program for finding the minimum in an array of numbers *A* of length *n*.

```
1.   procedure RECURSIVE_MIN (A, n)
2.   begin
3.   if (n = 1) then
4.       min := A[0];
5.   else
6.       lmin := RECURSIVE_MIN (A, n/2);
7.       rmin := RECURSIVE_MIN (&(A[n/2]), n - n/2);
8.       if (lmin < rmin) then
9.           min := lmin;
10.      else
11.          min := rmin;
12.      endelse;
13.  endelse;
14.  return min;
15.  end RECURSIVE_MIN
```

3.2.2 Data Decomposition

Data decomposition is a powerful and commonly used method for deriving concurrency in algorithms that operate on large data structures. In this method, the decomposition of computations is done in two steps. In the first step, the data on which the computations are performed is partitioned, and in the second step, this data partitioning is used to induce a partitioning of the computations into tasks. The operations that these tasks perform on different data partitions are usually similar (e.g., matrix multiplication introduced in Example 3.5) or are chosen from a small set of operations (e.g., LU factorization introduced in Example 3.10).

The partitioning of data can be performed in many possible ways as discussed next. In general, one must explore and evaluate all possible ways of partitioning the data and determine which one yields a natural and efficient computational decomposition.

Partitioning Output Data In many computations, each element of the output can be computed independently of others as a function of the input. In such computations, a partitioning of the output data automatically induces a decomposition of the problems into tasks, where each task is assigned the work of computing a portion of the output. We introduce the problem of matrix-

multiplication in Example 3.5 to illustrate a decomposition based on partitioning output data.

Example 3.5 Matrix multiplication

Consider the problem of multiplying two $n \times n$ matrices A and B to yield a matrix C. Figure 3.10 shows a decomposition of this problem into four tasks. Each matrix is considered to be composed of four blocks or submatrices defined by splitting each dimension of the matrix into half. The four submatrices of C, roughly of size $n/2 \times n/2$ each, are then independently computed by four tasks as the sums of the appropriate products of submatrices of A and B. ■

Figure 3.10. (a) Partitioning of input and output matrices into 2 x 2 submatrices. (b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices in (a).

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

$$\begin{aligned} &\text{Task 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ &\text{Task 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ &\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ &\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

(b)

Most matrix algorithms, including matrix-vector and matrix-matrix multiplication, can be formulated in terms of block matrix operations. In such a formulation, the matrix is viewed as composed of blocks or submatrices and the scalar arithmetic operations on its elements are replaced by the equivalent matrix operations on the blocks. The results of the element and the block versions of the algorithm are mathematically equivalent (Problem 3.10). Block versions of matrix algorithms are often used to aid decomposition.

The decomposition shown in Figure 3.10 is based on partitioning the output matrix C into four submatrices and each of the four tasks computes one of these submatrices. The reader must note that data-decomposition is distinct from the decomposition of the computation into tasks. Although the two are often related and the former often aids the latter, a given data-decomposition does not result in a unique decomposition into tasks. For example, Figure 3.11 shows two other decompositions of matrix multiplication, each into eight tasks, corresponding to the same data-decomposition as used in Figure 3.10(a).

Figure 3.11. Two examples of decomposition of matrix multiplication into eight tasks.

Decomposition I	Decomposition II
Task 1: $C_{1,1} = A_{1,1}B_{1,1}$	Task 1: $C_{1,1} = A_{1,1}B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$
Task 3: $C_{1,2} = A_{1,1}B_{1,2}$	Task 3: $C_{1,2} = A_{1,2}B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1}B_{1,2}$
Task 5: $C_{2,1} = A_{2,1}B_{1,1}$	Task 5: $C_{2,1} = A_{2,2}B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1}B_{1,1}$
Task 7: $C_{2,2} = A_{2,1}B_{1,2}$	Task 7: $C_{2,2} = A_{2,1}B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

We now introduce another example to illustrate decompositions based on data partitioning. Example 3.6 describes the problem of computing the frequency of a set of itemsets in a transaction database, which can be decomposed based on the partitioning of output data.

Example 3.6 Computing frequencies of itemsets in a transaction database

Consider the problem of computing the frequency of a set of itemsets in a transaction database. In this problem we are given a set T containing n transactions and a set I containing m itemsets. Each transaction and itemset contains a small number of items, out of a possible set of items. For example, T could be a grocery stores database of customer sales with each transaction being an individual grocery list of a shopper and each itemset could be a group of items in the store. If the store desires to find out how many customers bought each of the designated groups of items, then it would need to find the number of times that each itemset in I appears in all the transactions; i.e., the number of transactions of which each itemset is a subset of. Figure 3.12(a) shows an example of this type of computation. The database shown in Figure 3.12 consists of 10 transactions, and we are interested in computing the frequency of the eight itemsets shown in the second column. The actual frequencies of these itemsets in the database, which are the output of the frequency-computing program, are shown in the third column. For instance, itemset {D, K} appears twice, once in the second and once in the ninth transaction. ■

Figure 3.12. Computing itemset frequencies in a transaction database.

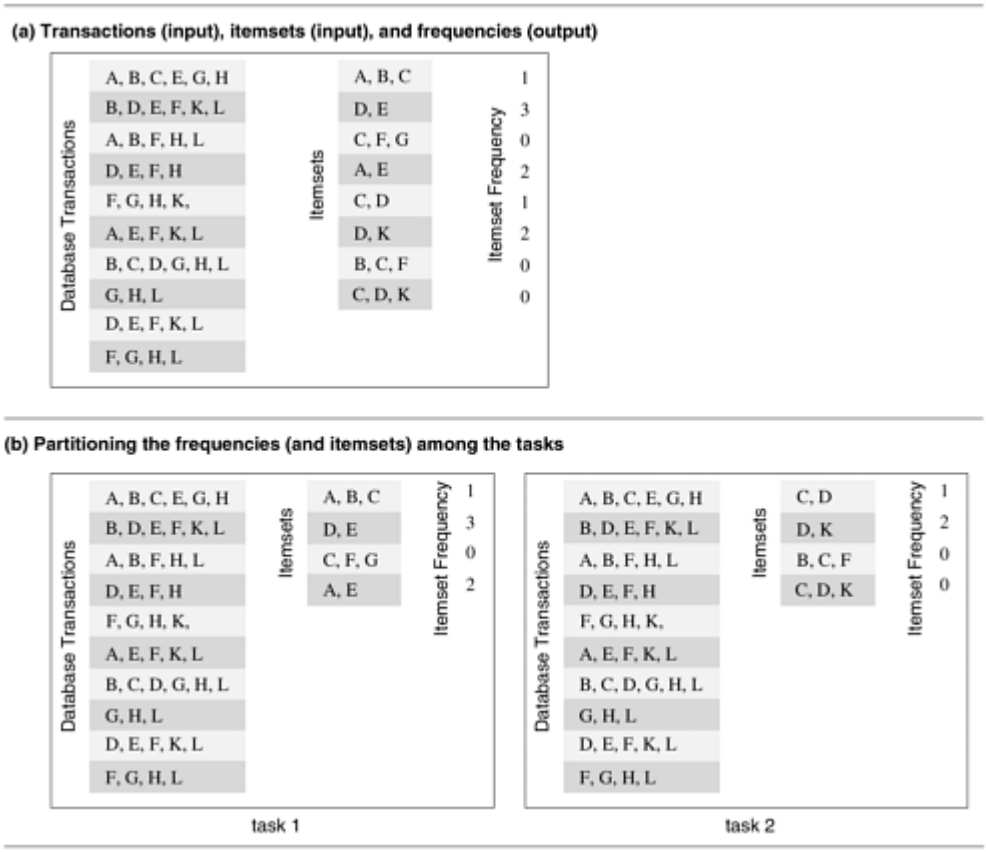
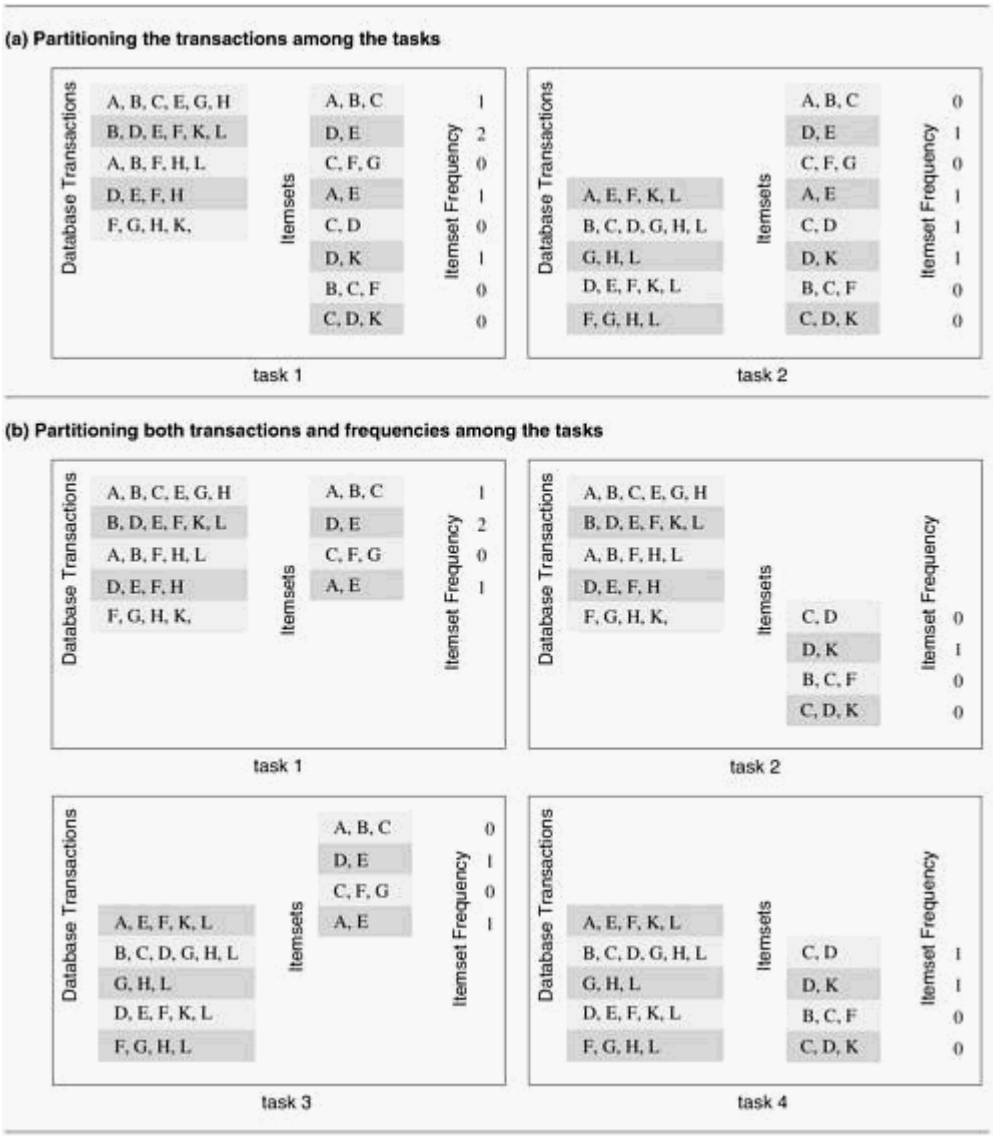


Figure 3.12(b) shows how the computation of frequencies of the itemsets can be decomposed into two tasks by partitioning the output into two parts and having each task compute its half of the frequencies. Note that, in the process, the itemsets input has also been partitioned, but the primary motivation for the decomposition of Figure 3.12(b) is to have each task independently compute the subset of frequencies assigned to it.

Partitioning Input Data Partitioning of output data can be performed only if each output can be naturally computed as a function of the input. In many algorithms, it is not possible or desirable to partition the output data. For example, while finding the minimum, maximum, or the sum of a set of numbers, the output is a single unknown value. In a sorting algorithm, the individual elements of the output cannot be efficiently determined in isolation. In such cases, it is sometimes possible to partition the input data, and then use this partitioning to induce concurrency. A task is created for each partition of the input data and this task performs as much computation as possible using these local data. Note that the solutions to tasks induced by input partitions may not directly solve the original problem. In such cases, a follow-up computation is needed to combine the results. For example, while finding the sum of a sequence of N numbers using p processes ($N > p$), we can partition the input into p subsets of nearly equal sizes. Each task then computes the sum of the numbers in one of the subsets. Finally, the p partial results can be added up to yield the final result.

The problem of computing the frequency of a set of itemsets in a transaction database described in Example 3.6 can also be decomposed based on a partitioning of input data. Figure 3.13(a) shows a decomposition based on a partitioning of the input set of transactions. Each of the two tasks computes the frequencies of all the itemsets in its respective subset of transactions. The two sets of frequencies, which are the independent outputs of the two tasks, represent intermediate results. Combining the intermediate results by pairwise addition yields the final result.

Figure 3.13. Some decompositions for computing itemset frequencies in a transaction database.

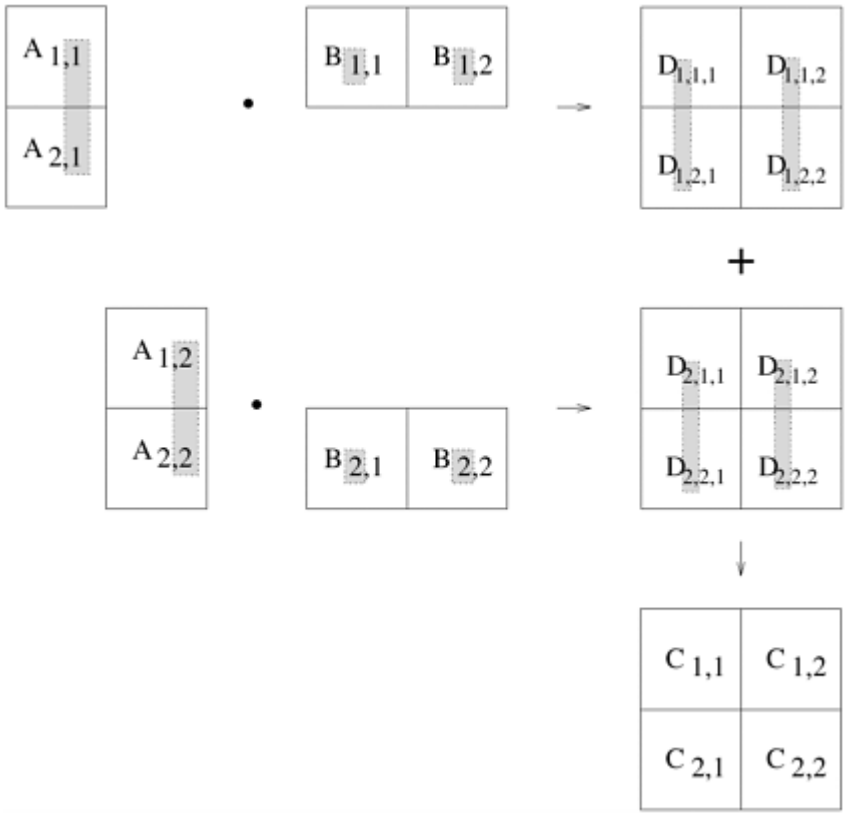


Partitioning both Input and Output Data In some cases, in which it is possible to partition the output data, partitioning of input data can offer additional concurrency. For example, consider the 4-way decomposition shown in Figure 3.13(b) for computing itemset frequencies. Here, both the transaction set and the frequencies are divided into two parts and a different one of the four possible combinations is assigned to each of the four tasks. Each task then computes a local set of frequencies. Finally, the outputs of Tasks 1 and 3 are added together, as are the outputs of Tasks 2 and 4.

Partitioning Intermediate Data Algorithms are often structured as multi-stage computations such that the output of one stage is the input to the subsequent stage. A decomposition of such an algorithm can be derived by partitioning the input or the output data of an intermediate stage of the algorithm. Partitioning intermediate data can sometimes lead to higher concurrency than partitioning input or output data. Often, the intermediate data are not generated explicitly in the serial algorithm for solving the problem and some restructuring of the original algorithm may be required to use intermediate data partitioning to induce a decomposition.

Let us revisit matrix multiplication to illustrate a decomposition based on partitioning intermediate data. Recall that the decompositions induced by a 2 x 2 partitioning of the output matrix C, as shown in Figures 3.10 and 3.11, have a maximum degree of concurrency of four. We can increase the degree of concurrency by introducing an intermediate stage in which eight tasks compute their respective product submatrices and store the results in a temporary three-dimensional matrix D, as shown in Figure 3.14. The submatrix $D_{k,i,j}$ is the product of $A_{i,k}$ and $B_{k,j}$.

Figure 3.14. Multiplication of matrices A and B with partitioning of the three-dimensional intermediate matrix D.



A partitioning of the intermediate matrix D induces a decomposition into eight tasks. Figure 3.15 shows this decomposition. After the multiplication phase, a relatively inexpensive matrix addition step can compute the result matrix C. All submatrices $D_{*,i,j}$ with the same second and third dimensions i and j are added to yield $C_{i,j}$. The eight tasks numbered 1 through 8 in Figure 3.15 perform $O(n^3/8)$ work each in multiplying $n/2$ x $n/2$ submatrices of A and B. Then, four tasks numbered 9 through 12 spend $O(n^2/4)$ time each in adding the appropriate $n/2$ x $n/2$ submatrices of the intermediate matrix D to yield the final result matrix C. Figure 3.16 shows the task-dependency graph corresponding to the decomposition shown in Figure 3.15.

Figure 3.15. A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix.

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

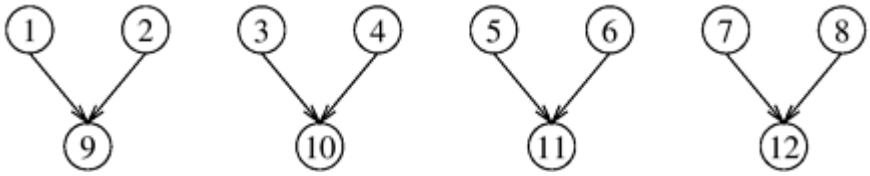
Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A decomposition induced by a partitioning of D

$$\begin{aligned}
\text{Task 01:} \quad & D_{1,1,1} = A_{1,1}B_{1,1} \\
\text{Task 02:} \quad & D_{2,1,1} = A_{1,2}B_{2,1} \\
\text{Task 03:} \quad & D_{1,1,2} = A_{1,1}B_{1,2} \\
\text{Task 04:} \quad & D_{2,1,2} = A_{1,2}B_{2,2} \\
\text{Task 05:} \quad & D_{1,2,1} = A_{2,1}B_{1,1} \\
\text{Task 06:} \quad & D_{2,2,1} = A_{2,2}B_{2,1} \\
\text{Task 07:} \quad & D_{1,2,2} = A_{2,1}B_{1,2} \\
\text{Task 08:} \quad & D_{2,2,2} = A_{2,2}B_{2,2} \\
\text{Task 09:} \quad & C_{1,1} = D_{1,1,1} + D_{2,1,1} \\
\text{Task 10:} \quad & C_{1,2} = D_{1,1,2} + D_{2,1,2} \\
\text{Task 11:} \quad & C_{2,1} = D_{1,2,1} + D_{2,2,1} \\
\text{Task 12:} \quad & C_{2,2} = D_{1,2,2} + D_{2,2,2}
\end{aligned}$$

Figure 3.16. The task-dependency graph of the decomposition shown in Figure 3.15.



Note that all elements of D are computed implicitly in the original decomposition shown in Figure 3.11, but are not explicitly stored. By restructuring the original algorithm and by explicitly storing D, we have been able to devise a decomposition with higher concurrency. This, however, has been achieved at the cost of extra aggregate memory usage.

The Owner-Computes Rule A decomposition based on partitioning output or input data is also widely referred to as the *owner-computes* rule. The idea behind this rule is that each partition performs all the computations involving data that it owns. Depending on the nature of the data or the type of data-partitioning, the owner-computes rule may mean different things. For instance, when we assign partitions of the input data to tasks, then the owner-computes rule means that a task performs all the computations that can be done using these data. On the other hand, if we partition the output data, then the owner-computes rule means that a task computes all the data in the partition assigned to it.

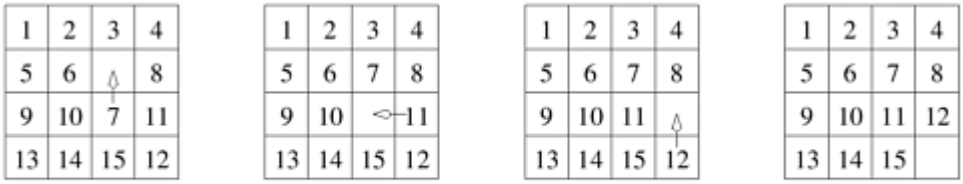
3.2.3 Exploratory Decomposition

Exploratory decomposition is used to decompose problems whose underlying computations correspond to a search of a space for solutions. In exploratory decomposition, we partition the search space into smaller parts, and search each one of these parts concurrently, until the desired solutions are found. For an example of exploratory decomposition, consider the 15-puzzle problem.

Example 3.7 The 15-puzzle problem

The 15-puzzle consists of 15 tiles numbered 1 through 15 and one blank tile placed in a 4 x 4 grid. A tile can be moved into the blank position from a position adjacent to it, thus creating a blank in the tile's original position. Depending on the configuration of the grid, up to four moves are possible: up, down, left, and right. The initial and final configurations of the tiles are specified. The objective is to determine any sequence or a shortest sequence of moves that transforms the initial configuration to the final configuration. Figure 3.17 illustrates sample initial and final configurations and a sequence of moves leading from the initial configuration to the final configuration. ■

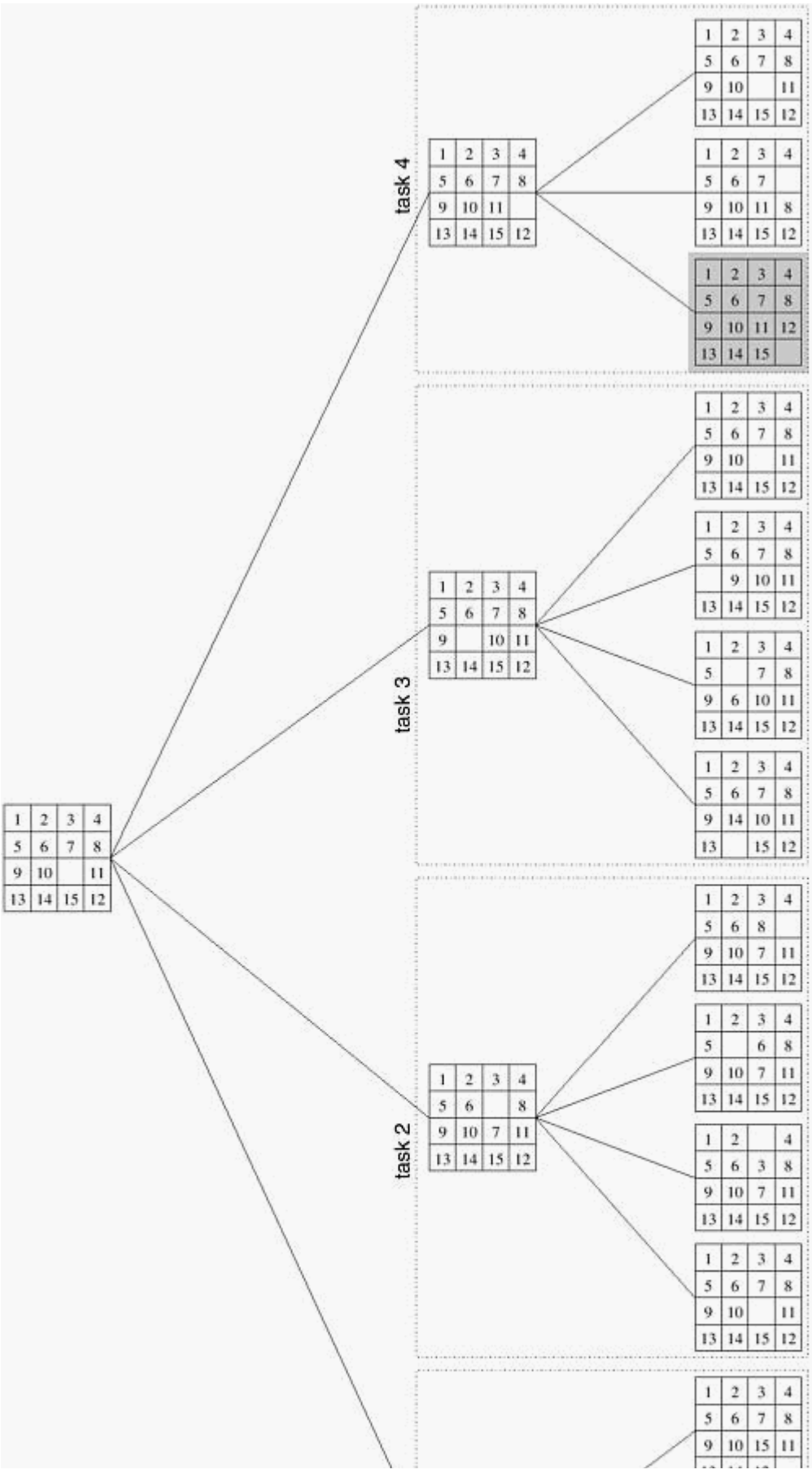
Figure 3.17. A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration.

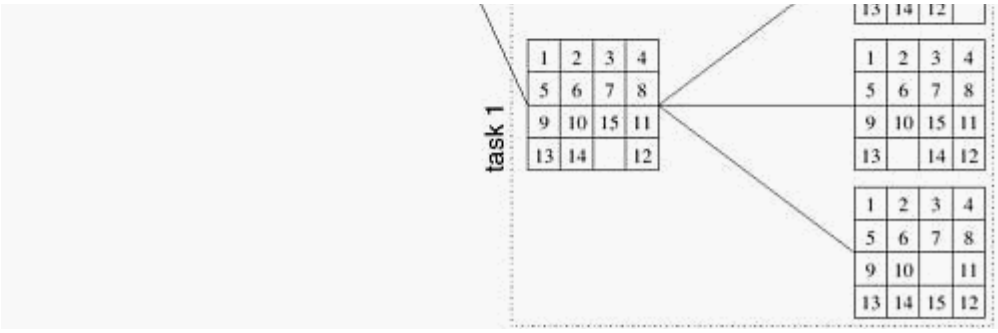


The 15-puzzle is typically solved using tree-search techniques. Starting from the initial configuration, all possible successor configurations are generated. A configuration may have 2, 3, or 4 possible successor configurations, each corresponding to the occupation of the empty slot by one of its neighbors. The task of finding a path from initial to final configuration now translates to finding a path from one of these newly generated configurations to the final configuration. Since one of these newly generated configurations must be closer to the solution by one move (if a solution exists), we have made some progress towards finding the solution. The configuration space generated by the tree search is often referred to as a state space graph. Each node of the graph is a configuration and each edge of the graph connects configurations that can be reached from one another by a single move of a tile.

One method for solving this problem in parallel is as follows. First, a few levels of configurations starting from the initial configuration are generated serially until the search tree has a sufficient number of leaf nodes (i.e., configurations of the 15-puzzle). Now each node is assigned to a task to explore further until at least one of them finds a solution. As soon as one of the concurrent tasks finds a solution it can inform the others to terminate their searches. Figure 3.18 illustrates one such decomposition into four tasks in which task 4 finds the solution.

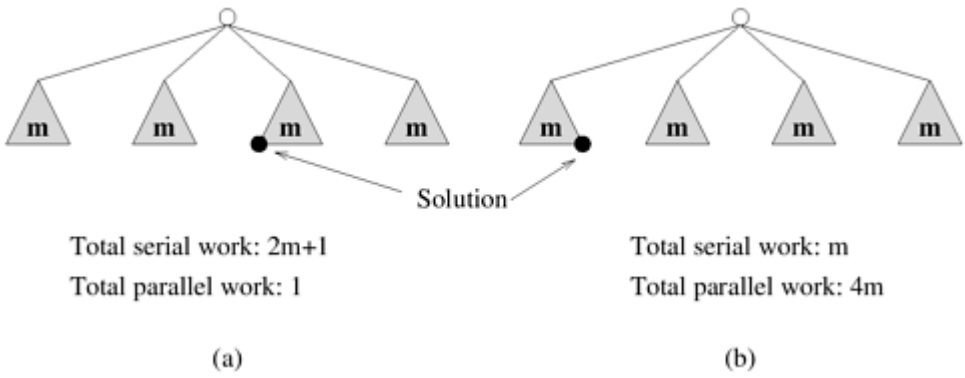
Figure 3.18. The states generated by an instance of the 15-puzzle problem.


task 4
task 3
task 2



Note that even though exploratory decomposition may appear similar to data-decomposition (the search space can be thought of as being the data that get partitioned) it is fundamentally different in the following way. The tasks induced by data-decomposition are performed in their entirety and each task performs useful computations towards the solution of the problem. On the other hand, in exploratory decomposition, unfinished tasks can be terminated as soon as an overall solution is found. Hence, the portion of the search space searched (and the aggregate amount of work performed) by a parallel formulation can be very different from that searched by a serial algorithm. The work performed by the parallel formulation can be either smaller or greater than that performed by the serial algorithm. For example, consider a search space that has been partitioned into four concurrent tasks as shown in Figure 3.19. If the solution lies right at the beginning of the search space corresponding to task 3 (Figure 3.19(a)), then it will be found almost immediately by the parallel formulation. The serial algorithm would have found the solution only after performing work equivalent to searching the entire space corresponding to tasks 1 and 2. On the other hand, if the solution lies towards the end of the search space corresponding to task 1 (Figure 3.19(b)), then the parallel formulation will perform almost four times the work of the serial algorithm and will yield no speedup.

Figure 3.19. An illustration of anomalous speedups resulting from exploratory decomposition.



3.2.4 Speculative Decomposition

Speculative decomposition is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it. In this situation, while one task is performing the computation whose output is used in deciding the next computation, other tasks can concurrently start the computations of the next stage. This scenario is similar to evaluating one or more of the branches of a *switch* statement in C in parallel before the input for the *switch* is available. While one task is performing the computation that will eventually resolve the switch, other tasks could pick up the multiple branches of the switch in parallel. When the input for the *switch* has finally been

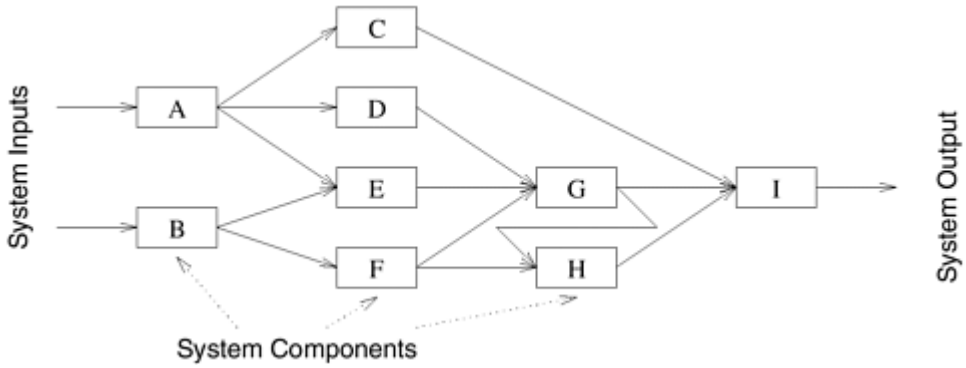
computed, the computation corresponding to the correct branch would be used while that corresponding to the other branches would be discarded. The parallel run time is smaller than the serial run time by the amount of time required to evaluate the condition on which the next task depends because this time is utilized to perform a useful computation for the next stage in parallel. However, this parallel formulation of a switch guarantees at least some wasteful computation. In order to minimize the wasted computation, a slightly different formulation of speculative decomposition could be used, especially in situations where one of the outcomes of the switch is more likely than the others. In this case, only the most promising branch is taken up a task in parallel with the preceding computation. In case the outcome of the switch is different from what was anticipated, the computation is rolled back and the correct branch of the switch is taken.

The speedup due to speculative decomposition can add up if there are multiple speculative stages. An example of an application in which speculative decomposition is useful is *discrete event simulation*. A detailed description of discrete event simulation is beyond the scope of this chapter; however, we give a simplified description of the problem.

Example 3.8 Parallel discrete event simulation

Consider the simulation of a system that is represented as a network or a directed graph. The nodes of this network represent components. Each component has an input buffer of jobs. The initial state of each component or node is idle. An idle component picks up a job from its input queue, if there is one, processes that job in some finite amount of time, and puts it in the input buffer of the components which are connected to it by outgoing edges. A component has to wait if the input buffer of one of its outgoing neighbors if full, until that neighbor picks up a job to create space in the buffer. There is a finite number of input job types. The output of a component (and hence the input to the components connected to it) and the time it takes to process a job is a function of the input job. The problem is to simulate the functioning of the network for a given sequence or a set of sequences of input jobs and compute the total completion time and possibly other aspects of system behavior. Figure 3.20 shows a simple network for a discrete event solution problem. ■

Figure 3.20. A simple network for discrete event simulation.



The problem of simulating a sequence of input jobs on the network described in Example 3.8 appears inherently sequential because the input of a typical component is the output of another. However, we can define speculative tasks that start simulating a subpart of the network, each

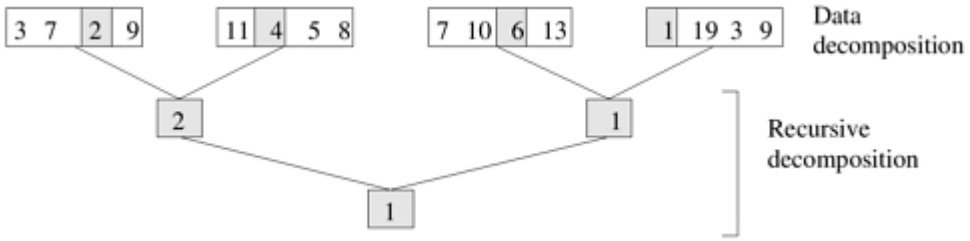
assuming one of several possible inputs to that stage. When an actual input to a certain stage becomes available (as a result of the completion of another selector task from a previous stage), then all or part of the work required to simulate this input would have already been finished if the speculation was correct, or the simulation of this stage is restarted with the most recent correct input if the speculation was incorrect.

Speculative decomposition is different from exploratory decomposition in the following way. In speculative decomposition, the input at a branch leading to multiple parallel tasks is unknown, whereas in exploratory decomposition, the output of the multiple tasks originating at a branch is unknown. In speculative decomposition, the serial algorithm would strictly perform only one of the tasks at a speculative stage because when it reaches the beginning of that stage, it knows exactly which branch to take. Therefore, by preemptively computing for multiple possibilities out of which only one materializes, a parallel program employing speculative decomposition performs more aggregate work than its serial counterpart. Even if only one of the possibilities is explored speculatively, the parallel algorithm may perform more or the same amount of work as the serial algorithm. On the other hand, in exploratory decomposition, the serial algorithm too may explore different alternatives one after the other, because the branch that may lead to the solution is not known beforehand. Therefore, the parallel program may perform more, less, or the same amount of aggregate work compared to the serial algorithm depending on the location of the solution in the search space.

3.2.5 Hybrid Decompositions

So far we have discussed a number of decomposition methods that can be used to derive concurrent formulations of many algorithms. These decomposition techniques are not exclusive, and can often be combined together. Often, a computation is structured into multiple stages and it is sometimes necessary to apply different types of decomposition in different stages. For example, while finding the minimum of a large set of n numbers, a purely recursive decomposition may result in far more tasks than the number of processes, P, available. An efficient decomposition would partition the input into P roughly equal parts and have each task compute the minimum of the sequence assigned to it. The final result can be obtained by finding the minimum of the P intermediate results by using the recursive decomposition shown in Figure 3.21.

Figure 3.21. Hybrid decomposition for finding the minimum of an array of size 16 using four tasks.



As another example of an application of hybrid decomposition, consider performing quicksort in parallel. In Example 3.4, we used a recursive decomposition to derive a concurrent formulation of quicksort. This formulation results in $O(n)$ tasks for the problem of sorting a sequence of size n. But due to the dependencies among these tasks and due to uneven sizes of the tasks, the effective concurrency is quite limited. For example, the first task for splitting the input list into two parts takes $O(n)$ time, which puts an upper limit on the performance gain possible via parallelization. But the step of splitting lists performed by tasks in parallel quicksort can also be decomposed using the input decomposition technique discussed in Section 9.4.1. The resulting hybrid decomposition that combines recursive decomposition and the input data-decomposition

leads to a highly concurrent formulation of quicksort.

3.3 Characteristics of Tasks and Interactions

The various decomposition techniques described in the previous section allow us to identify the concurrency that is available in a problem and decompose it into tasks that can be executed in parallel. The next step in the process of designing a parallel algorithm is to take these tasks and assign (i.e., map) them onto the available processes. While devising a mapping scheme to construct a good parallel algorithm, we often take a cue from the decomposition. The nature of the tasks and the interactions among them has a bearing on the mapping. In this section, we shall discuss the various properties of tasks and inter-task interactions that affect the choice of a good mapping.

3.3.1 Characteristics of Tasks

The following four characteristics of the tasks have a large influence on the suitability of a mapping scheme.

Task Generation The tasks that constitute a parallel algorithm may be generated either statically or dynamically. *Static task generation* refers to the scenario where all the tasks are known before the algorithm starts execution. Data decomposition usually leads to static task generation. Examples of data-decomposition leading to a static task generation include matrix-multiplication and LU factorization (Problem 3.5). Recursive decomposition can also lead to a static task-dependency graph. Finding the minimum of a list of numbers (Figure 3.9) is an example of a static recursive task-dependency graph.

Certain decompositions lead to a *dynamic task generation* during the execution of the algorithm. In such decompositions, the actual tasks and the task-dependency graph are not explicitly available *a priori*, although the high level rules or guidelines governing task generation are known as a part of the algorithm. Recursive decomposition can lead to dynamic task generation. For example, consider the recursive decomposition in quicksort (Figure 3.8). The tasks are generated dynamically, and the size and shape of the task tree is determined by the values in the input array to be sorted. An array of the same size can lead to task-dependency graphs of different shapes and with a different total number of tasks.

Exploratory decomposition can be formulated to generate tasks either statically or dynamically. For example, consider the 15-puzzle problem discussed in Section 3.2.3. One way to generate a static task-dependency graph using exploratory decomposition is as follows. First, a preprocessing task starts with the initial configuration and expands the search tree in a breadth-first manner until a predefined number of configurations are generated. These configuration now represent independent tasks, which can be mapped onto different processes and run independently. A different decomposition that generates tasks dynamically would be one in which a task takes a state as input, expands it through a predefined number of steps of breadth-first search and spawns new tasks to perform the same computation on each of the resulting states (unless it has found the solution, in which case the algorithm terminates).

Task Sizes The size of a task is the relative amount of time required to complete it. The complexity of mapping schemes often depends on whether or not the tasks are *uniform*; i.e., whether or not they require roughly the same amount of time. If the amount of time required by the tasks varies significantly, then they are said to be *non-uniform*. For example, the tasks in the decompositions for matrix multiplication shown in Figures 3.10 and 3.11 would be considered uniform. On the other hand, the tasks in quicksort in Figure 3.8 are non-uniform.

Knowledge of Task Sizes The third characteristic that influences the choice of mapping scheme is knowledge of the task size. If the size of all the tasks is known, then this information can often be used in mapping of tasks to processes. For example, in the various decompositions for matrix multiplication discussed so far, the computation time for each task is known before the parallel program starts. On the other hand, the size of a typical task in the 15-puzzle problem is unknown. We do not know *a priori* how many moves will lead to the solution from a given state.

Size of Data Associated with Tasks Another important characteristic of a task is the size of data associated with it. The reason this is an important consideration for mapping is that the data associated with a task must be available to the process performing that task, and the size and the location of these data may determine the process that can perform the task without incurring excessive data-movement overheads.

Different types of data associated with a task may have different sizes. For instance, the input data may be small but the output may be large, or vice versa. For example, the input to a task in the 15-puzzle problem may be just one state of the puzzle. This is a small input relative to the amount of computation that may be required to find a sequence of moves from this state to a solution state. In the problem of computing the minimum of a sequence, the size of the input is proportional to the amount of computation, but the output is just one number. In the parallel formulation of the quick sort, the size of both the input and the output data is of the same order as the sequential time needed to solve the task.

3.3.2 Characteristics of Inter-Task Interactions

In any nontrivial parallel algorithm, tasks need to interact with each other to share data, work, or synchronization information. Different parallel algorithms require different types of interactions among concurrent tasks. The nature of these interactions makes them more suitable for certain programming paradigms and mapping schemes, and less suitable for others. The types of inter-task interactions can be described along different dimensions, each corresponding to a distinct characteristic of the underlying computations.

Static versus Dynamic One way of classifying the type of interactions that take place among concurrent tasks is to consider whether or not these interactions have a *static* or *dynamic* pattern. An interaction pattern is static if for each task, the interactions happen at predetermined times, and if the set of tasks to interact with at these times is known prior to the execution of the algorithm. In other words, in a static interaction pattern, not only is the task-interaction graph known *a priori*, but the stage of the computation at which each interaction occurs is also known. An interaction pattern is dynamic if the timing of interactions or the set of tasks to interact with cannot be determined prior to the execution of the algorithm.

Static interactions can be programmed easily in the message-passing paradigm, but dynamic interactions are harder to program. The reason is that interactions in message-passing require active involvement of both interacting tasks – the sender and the receiver of information. The unpredictable nature of dynamic iterations makes it hard for both the sender and the receiver to participate in the interaction at the same time. Therefore, when implementing a parallel algorithm with dynamic interactions in the message-passing paradigm, the tasks must be assigned additional synchronization or polling responsibility. Shared-address space programming can code both types of interactions equally easily.

The decompositions for parallel matrix multiplication presented earlier in this chapter exhibit static inter-task interactions. For an example of dynamic interactions, consider solving the 15-puzzle problem in which tasks are assigned different states to explore after an initial step that generates the desirable number of states by applying breadth-first search on the initial state. It

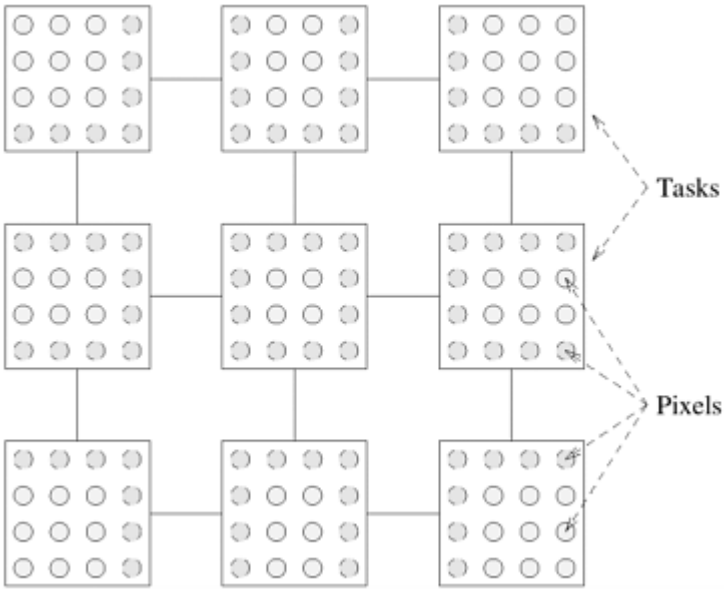
is possible that a certain state leads to all dead ends and a task exhausts its search space without reaching the goal state, while other tasks are still busy trying to find a solution. The task that has exhausted its work can pick up an unexplored state from the queue of another busy task and start exploring it. The interactions involved in such a transfer of work from one task to another are dynamic.

Regular versus Irregular Another way of classifying the interactions is based upon their spatial structure. An interaction pattern is considered to be *regular* if it has some structure that can be exploited for efficient implementation. On the other hand, an interaction pattern is called *irregular* if no such regular pattern exists. Irregular and dynamic communications are harder to handle, particularly in the message-passing programming paradigm. An example of a decomposition with a regular interaction pattern is the problem of image dithering.

Example 3.9 Image dithering

In image dithering, the color of each pixel in the image is determined as the weighted average of its original value and the values of its neighboring pixels. We can easily decompose this computation, by breaking the image into square regions and using a different task to dither each one of these regions. Note that each task needs to access the pixel values of the region assigned to it as well as the values of the image surrounding its region. Thus, if we regard the tasks as nodes of a graph with an edge linking a pair of interacting tasks, the resulting pattern is a two-dimensional mesh, as shown in Figure 3.22. ■

Figure 3.22. The regular two-dimensional task-interaction graph for image dithering. The pixels with dotted outline require color values from the boundary pixels of the neighboring tasks.



Sparse matrix-vector multiplication discussed in Section 3.1.2 provides a good example of irregular interaction, which is shown in Figure 3.6. In this decomposition, even though each

task, by virtue of the decomposition, knows *a priori* which rows of matrix *A* it needs to access, without scanning the row(s) of *A* assigned to it, a task cannot know which entries of vector *b* it requires. The reason is that the access pattern for *b* depends on the structure of the sparse matrix *A*.

Read-only versus Read-Write We have already learned that sharing of data among tasks leads to inter-task interaction. However, the type of sharing may impact the choice of the mapping. Data sharing interactions can be categorized as either *read-only* or *read-write* interactions. As the name suggests, in read-only interactions, tasks require only a read-access to the data shared among many concurrent tasks. For example, in the decomposition for parallel matrix multiplication shown in Figure 3.10, the tasks only need to read the shared input matrices *A* and *B*. In read-write interactions, multiple tasks need to read and write on some shared data. For example, consider the problem of solving the 15-puzzle. The parallel formulation method proposed in Section 3.2.3 uses an exhaustive search to find a solution. In this formulation, each state is considered an equally suitable candidate for further expansion. The search can be made more efficient if the states that appear to be closer to the solution are given a priority for further exploration. An alternative search technique known as heuristic search implements such a strategy. In heuristic search, we use a heuristic to provide a relative approximate indication of the distance of each state from the solution (i.e. the potential number of moves required to reach the solution). In the case of the 15-puzzle, the number of tiles that are out of place in a given state could serve as such a heuristic. The states that need to be expanded further are stored in a priority queue based on the value of this heuristic. While choosing the states to expand, we give preference to more promising states, i.e. the ones that have fewer out-of-place tiles and hence, are more likely to lead to a quick solution. In this situation, the priority queue constitutes shared data and tasks need both read and write access to it; they need to put the states resulting from an expansion into the queue and they need to pick up the next most promising state for the next expansion.

One-way versus Two-way In some interactions, the data or work needed by a task or a subset of tasks is explicitly supplied by another task or subset of tasks. Such interactions are called *two-way* interactions and usually involve predefined producer and consumer tasks. In other interactions, only one of a pair of communicating tasks initiates the interaction and completes it without interrupting the other one. Such an interaction is called a *one-way* interaction. All read-only interactions can be formulated as one-way interactions. Read-write interactions can be either one-way or two-way.

The shared-address-space programming paradigms can handle both one-way and two-way interactions equally easily. However, one-way interactions cannot be directly programmed in the message-passing paradigm because the source of the data to be transferred must explicitly send the data to the recipient. In the message-passing paradigm, all one-way interactions must be converted to two-way interactions via program restructuring. Static one-way interactions can be easily converted to two-way communications. Since the time and the location in the program of a static one-way interaction is known *a priori*, introducing a matching interaction operation in the partner task is enough to convert a one-way static interaction to a two-way static interaction. On the other hand, dynamic one-way interactions can require some nontrivial program restructuring to be converted to two-way interactions. The most common such restructuring involves polling. Each task checks for pending requests from other tasks after regular intervals, and services such requests, if any.

3.4 Mapping Techniques for Load Balancing

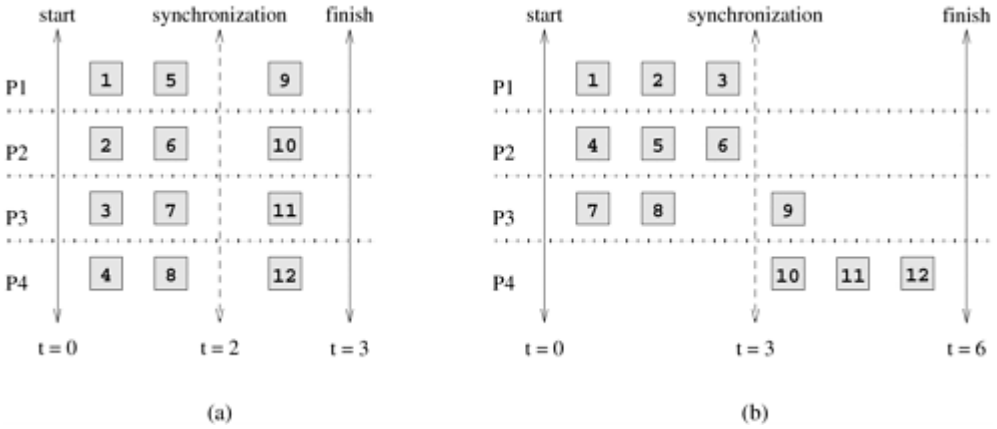
Once a computation has been decomposed into tasks, these tasks are mapped onto processes with the objective that all tasks complete in the shortest amount of elapsed time. In order to achieve a small execution time, the *overheads* of executing the tasks in parallel must be minimized. For a given decomposition, there are two key sources of overhead. The time spent in inter-process interaction is one source of overhead. Another important source of overhead is the time that some processes may spend being idle. Some processes can be idle even before the overall computation is finished for a variety of reasons. Uneven load distribution may cause some processes to finish earlier than others. At times, all the unfinished tasks mapped onto a process may be waiting for tasks mapped onto other processes to finish in order to satisfy the constraints imposed by the task-dependency graph. Both interaction and idling are often a function of mapping. Therefore, a good mapping of tasks onto processes must strive to achieve the twin objectives of (1) reducing the amount of time processes spend in interacting with each other, and (2) reducing the total amount of time some processes are idle while the others are engaged in performing some tasks.

These two objectives often conflict with each other. For example, the objective of minimizing the interactions can be easily achieved by assigning sets of tasks that need to interact with each other onto the same process. In most cases, such a mapping will result in a highly unbalanced workload among the processes. In fact, following this strategy to the limit will often map all tasks onto a single process. As a result, the processes with a lighter load will be idle when those with a heavier load are trying to finish their tasks. Similarly, to balance the load among processes, it may be necessary to assign tasks that interact heavily to different processes. Due to the conflicts between these objectives, finding a good mapping is a nontrivial problem.

In this section, we will discuss various schemes for mapping tasks onto processes with the primary view of balancing the task workload of processes and minimizing their idle time. Reducing inter-process interaction is the topic of Section 3.5. The reader should be aware that assigning a balanced aggregate load of tasks to each process is a necessary but not sufficient condition for reducing process idling. Recall that the tasks resulting from a decomposition are not all ready for execution at the same time. A task-dependency graph determines which tasks can execute in parallel and which must wait for some others to

finish at a given stage in the execution of a parallel algorithm. Therefore, it is possible in a certain parallel formulation that although all processes perform the same aggregate amount of work, at different times, only a fraction of the processes are active while the remainder contain tasks that must wait for other tasks to finish. Similarly, poor synchronization among interacting tasks can lead to idling if one of the tasks has to wait to send or receive data from another task. A good mapping must ensure that the computations and interactions among processes at each stage of the execution of the parallel algorithm are well balanced. Figure 3.23 shows two mappings of 12-task decomposition in which the last four tasks can be started only after the first eight are finished due to dependencies among tasks. As the figure shows, two mappings, each with an overall balanced workload, can result in different completion times.

Figure 3.23. Two mappings of a hypothetical decomposition with a synchronization.



Mapping techniques used in parallel algorithms can be broadly classified into two categories: *static* and *dynamic*. The parallel programming paradigm and the characteristics of tasks and the interactions among them determine whether a static or a dynamic mapping is more suitable.

- Static Mapping: Static mapping techniques distribute the tasks among processes prior to the execution of the algorithm. For statically generated tasks, either static or dynamic mapping can be used. The choice of a good mapping in this case depends on several factors, including the knowledge of task sizes, the size of data associated with tasks, the characteristics of inter-task interactions, and even the parallel programming paradigm. Even when task sizes are known, in general, the problem of obtaining an optimal mapping is an NP-complete problem for non-uniform tasks. However, for many practical cases, relatively inexpensive heuristics provide fairly acceptable approximate solutions to the optimal static mapping problem.

  Algorithms that make use of static mapping are in general easier to design and program.

- Dynamic Mapping: Dynamic mapping techniques distribute the work among processes during the execution of the algorithm. If tasks are generated dynamically, then they must be mapped dynamically too. If task sizes are unknown, then a static mapping can potentially lead to serious load-imbalances and dynamic mappings are usually more effective. If the amount of data associated with tasks is large relative to the computation, then a dynamic mapping may entail moving this data among processes. The cost of this data movement may outweigh some other advantages of dynamic mapping and may render a static mapping more suitable. However, in a shared-address-space paradigm, dynamic mapping may work well even with large data associated with tasks if the interaction is read-only. The reader should be aware that the shared-address-space programming paradigm does not automatically provide immunity against data-movement costs. If the underlying hardware is NUMA (Section 2.3.2), then the data may physically move from a distant memory. Even in a cc-UMA architecture, the data may have to move from one cache to another.

  Algorithms that require dynamic mapping are usually more complicated, particularly in the message-passing programming paradigm.

Having discussed the guidelines for choosing between static and dynamic mappings, we now describe various schemes of these two types of mappings in detail.

3.4.1 Schemes for Static Mapping

Static mapping is often, though not exclusively, used in conjunction with a decomposition based on data partitioning. Static mapping is also used for mapping certain problems that are expressed naturally by a static task-dependency graph. In the following subsections, we will discuss mapping schemes based on data partitioning and task partitioning.

Mappings Based on Data Partitioning

In this section, we will discuss mappings based on partitioning two of the most common ways of representing data in algorithms, namely, arrays and graphs. The data-partitioning actually induces a decomposition, but the partitioning or the decomposition is selected with the final mapping in mind.

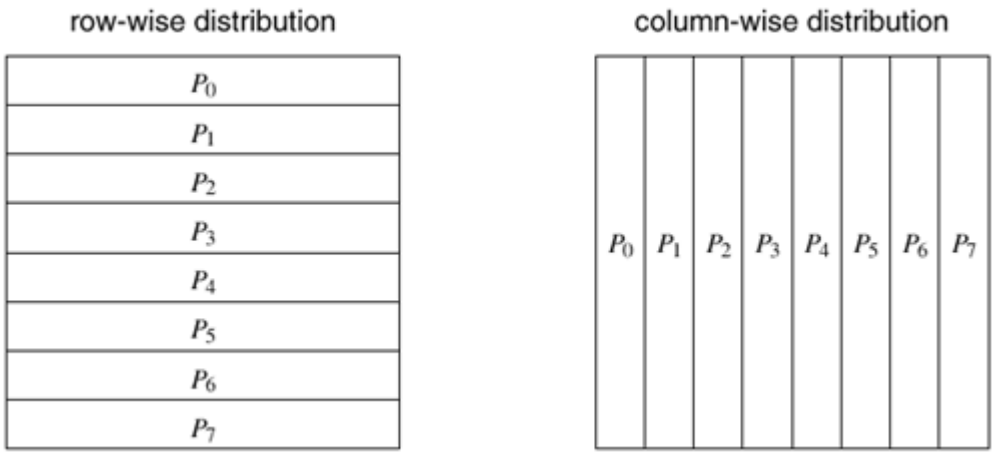
Array Distribution Schemes In a decomposition based on partitioning data, the tasks are closely associated with portions of data by the owner-computes rule. Therefore, mapping the relevant data onto the processes is equivalent to mapping tasks onto processes. We now study some commonly used techniques of distributing arrays or matrices among processes.

Block Distributions

Block distributions are some of the simplest ways to distribute an array and assign uniform contiguous portions of the array to different processes. In these distributions, a *d*-dimensional array is distributed among the processes such that each process receives a contiguous block of array entries along a specified subset of array dimensions. Block distributions of arrays are particularly suitable when there is a locality of interaction, i.e., computation of an element of an array requires other nearby elements in the array.

For example, consider an n x n two-dimensional array A with n rows and n columns. We can now select one of these dimensions, e.g., the first dimension, and partition the array into p parts such that the kth part contains rows $kn/p \ldots (k + 1)n/p - 1$, where $0 \leq k < p$. That is, each partition contains a block of n/p consecutive rows of A. Similarly, if we partition A along the second dimension, then each partition contains a block of n/p consecutive columns. These row- and column-wise array distributions are illustrated in Figure 3.24.

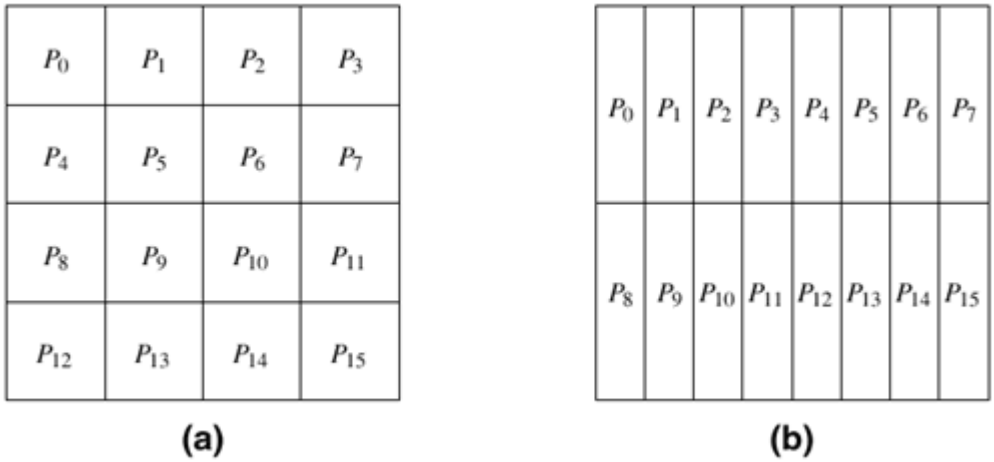
Figure 3.24. Examples of one-dimensional partitioning of an array among eight processes.



Similarly, instead of selecting a single dimension, we can select multiple dimensions to

partition. For instance, in the case of array A we can select both dimensions and partition the matrix into blocks such that each block corresponds to a n/p_1 x n/p_2 section of the matrix, with $p = p_1$ x p_2 being the number of processes. Figure 3.25 illustrates two different two-dimensional distributions, on a 4 x 4 and 2x 8 process grid, respectively. In general, given a d-dimensional array, we can distribute it using up to a d-dimensional block distribution.

Figure 3.25. Examples of two-dimensional distributions of an array, (a) on a 4 x 4 process grid, and (b) on a 2 x 8 process grid.



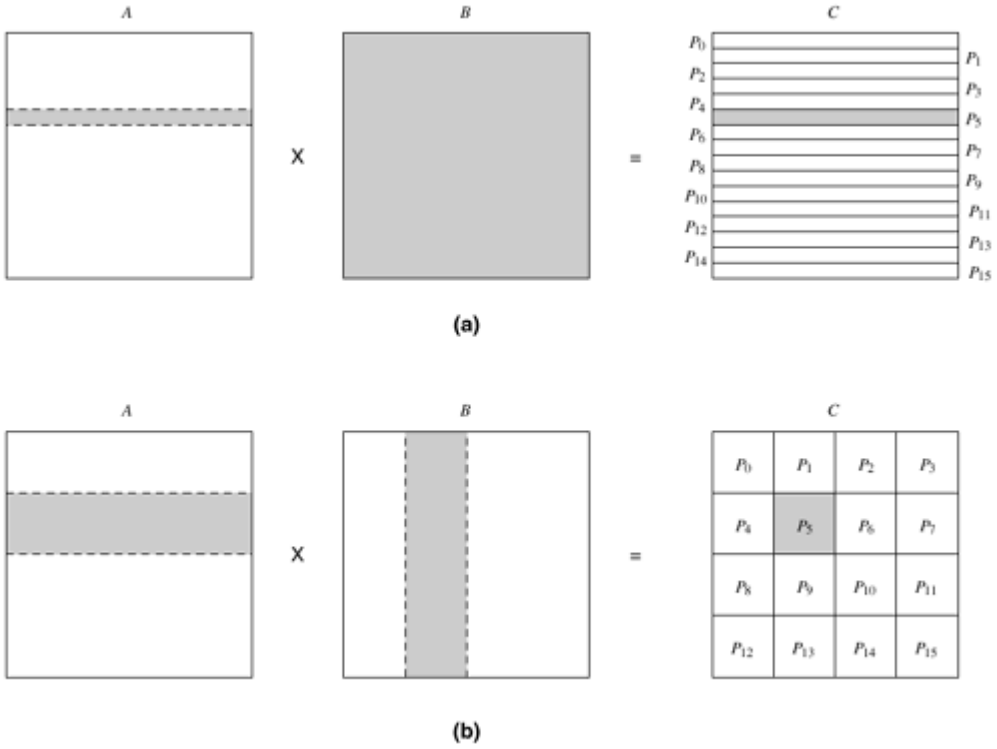
Using these block distributions we can load-balance a variety of parallel computations that operate on multi-dimensional arrays. For example, consider the n x n matrix multiplication $C = A$ x B , as discussed in Section 3.2.2. One way of decomposing this computation is to partition the output matrix C . Since each entry of C requires the same amount of computation, we can balance the computations by using either a one- or two-dimensional block distribution to partition C uniformly among the p available processes. In the first case, each process will get a block of n/p rows (or columns) of C, whereas in the second case, each process will get a block of size $n/\sqrt{p} \times n/\sqrt{p}$. In either case, the process will be responsible for computing the entries of the partition of C assigned to it.

As the matrix-multiplication example illustrates, quite often we have the choice of mapping the computations using either a one- or a two-dimensional distribution (and even more choices in the case of higher dimensional arrays). In general, higher dimensional distributions allow us to use more processes. For example, in the case of matrix-matrix multiplication, a one-dimensional distribution will allow us to use up to n processes by assigning a single row of C to each process. On the other hand, a two-dimensional distribution will allow us to use up to n^2 processes by assigning a single element of C to each process.

In addition to allowing a higher degree of concurrency, higher dimensional distributions also sometimes help in reducing the amount of interactions among the different processes for many problems. Figure 3.26 illustrates this in the case of dense matrix-multiplication. With a one-dimensional partitioning along the rows, each process needs to access the corresponding n/p rows of matrix A and the entire matrix B, as shown in Figure 3.26(a) for process P_5. Thus the total amount of data that needs to be accessed is $n^2/p + n^2$. However, with a two-dimensional distribution, each process needs to access $n/\sqrt{p}$ rows of matrix A and $n/\sqrt{p}$ columns of matrix B, as shown in Figure 3.26(b) for process P_5. In the two-dimensional case, the total amount of shared data that each process needs to access is $O(n^2/\sqrt{p})$, which is significantly smaller compared to $O(n^2)$ shared data in the one-dimensional case.

Figure 3.26. Data sharing needed for matrix multiplication with (a)

one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices *A* and *B* are required by the process that computes the shaded portion of the output matrix *C*.



Cyclic and Block-Cyclic Distributions

If the amount of work differs for different elements of a matrix, a block distribution can potentially lead to load imbalances. A classic example of this phenomenon is LU factorization of a matrix, in which the amount of computation increases from the top left to the bottom right of the matrix.

Example 3.10 Dense LU factorization

In its simplest form,the LU factorization algorithm factors a nonsingular square matrix *A* into the product of a lower triangular matrix *L* with a unit diagonal and an upper triangular matrix *U*. Algorithm 3.3 shows the serial algorithm. Let *A* be an *n* x *n* matrix with rows and columns numbered from 1 to *n*. The factorization process consists of *n* major steps – each consisting of an iteration of the outer loop starting at Line 3 in Algorithm 3.3. In step *k*, first, the partial column *A*[*k* + 1 : *n*, *k*] is divided by *A*[*k*, *k*]. Then, the outer product *A*[*k* + 1 : *n*, *k*] x *A*[*k*, *k* + 1 : *n*] is subtracted from the (*n* - *k*) x (*n* - *k*) submatrix *A*[*k* + 1 : *n*, *k* + 1 : *n*]. In a practical implementation of LU factorization, separate arrays are not used for *L* and *U* and *A* is modified to store *L* and *U* in its lower and upper triangular parts, respectively. The 1's on the principal diagonal of *L* are implicit and the diagonal entries actually belong to *U* after factorization.

Algorithm 3.3 A serial column-based algorithm to factor a

nonsingular matrix A into a lower-triangular matrix L and an upper-triangular matrix U. Matrices L and U share space with A. On Line 9, $A[i, j]$ on the left side of the assignment is equivalent to $L[i, j]$ if $i > j$; otherwise, it is equivalent to $U[i, j]$.

```
1.    procedure COL_LU (A)
2.    begin
3.       for k := 1 to n do
4.           for j := k to n do
5.               A[j, k]:= A[j, k]/A[k, k];
6.           endfor;
7.           for j := k + 1 to n do
8.               for i := k + 1 to n do
9.                   A[i, j] := A[i, j] - A[i, k] x A[k, j];
10.              endfor;
11.          endfor;
    /*
After this iteration, column A[k + 1 : n, k] is logically the kth
column of L and row A[k, k : n] is logically the kth row of U.
    */
12.      endfor;
13.   end COL_LU
```

Figure 3.27 shows a possible decomposition of LU factorization into 14 tasks using a 3 x 3 block partitioning of the matrix and using a block version of Algorithm 3.3. ■

Figure 3.27. A decomposition of LU factorization into 14 tasks.

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} \cdot \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

1: $A_{1,1} \rightarrow L_{1,1}U_{1,1}$
2: $L_{2,1} = A_{2,1}U_{1,1}^{-1}$
3: $L_{3,1} = A_{3,1}U_{1,1}^{-1}$
4: $U_{1,2} = L_{1,1}^{-1}A_{1,2}$
5: $U_{1,3} = L_{1,1}^{-1}A_{1,3}$
6: $A_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$
7: $A_{3,2} = A_{3,2} - L_{3,1}U_{1,2}$
8: $A_{2,3} = A_{2,3} - L_{2,1}U_{1,3}$
9: $A_{3,3} = A_{3,3} - L_{3,1}U_{1,3}$
10: $A_{2,2} \rightarrow L_{2,2}U_{2,2}$
11: $L_{3,2} = A_{3,2}U_{2,2}^{-1}$
12: $U_{2,3} = L_{2,2}^{-1}A_{2,3}$
13: $A_{3,3} = A_{3,3} - L_{3,2}U_{2,3}$
14: $A_{3,3} \rightarrow L_{3,3}U_{3,3}$

For each iteration of the outer loop $k := 1$ to n, the next nested loop in Algorithm 3.3 goes from $k + 1$ to n. In other words, the active part of the matrix, as shown in Figure 3.28, shrinks towards the bottom right corner of the matrix as the computation proceeds. Therefore, in a block distribution, the processes assigned to the beginning rows and columns (i.e., left rows and top columns) would perform far less work than those assigned to the later rows and columns. For example, consider the decomposition for LU factorization shown in Figure 3.27 with a 3 x 3 two-dimensional block partitioning of the matrix. If we map all tasks associated with a certain block onto a process in a 9-process ensemble, then a significant amount of idle time will result. First, computing different blocks of the matrix requires different amounts of work. This is illustrated in Figure 3.29. For example, computing the final value of $A_{1,1}$ (which is actually $L_{1,1}$ $U_{1,1}$) requires only one task – Task 1. On the other hand, computing the final value of $A_{3,3}$

requires three tasks – Task 9, Task 13, and Task 14. Secondly, the process working on a block may idle even when there are unfinished tasks associated with that block. This idling can occur if the constraints imposed by the task-dependency graph do not allow the remaining tasks on this process to proceed until one or more tasks mapped onto other processes are completed.

Figure 3.28. A typical computation in Gaussian elimination and the active part of the coefficient matrix during the *k*th iteration of the outer loop.

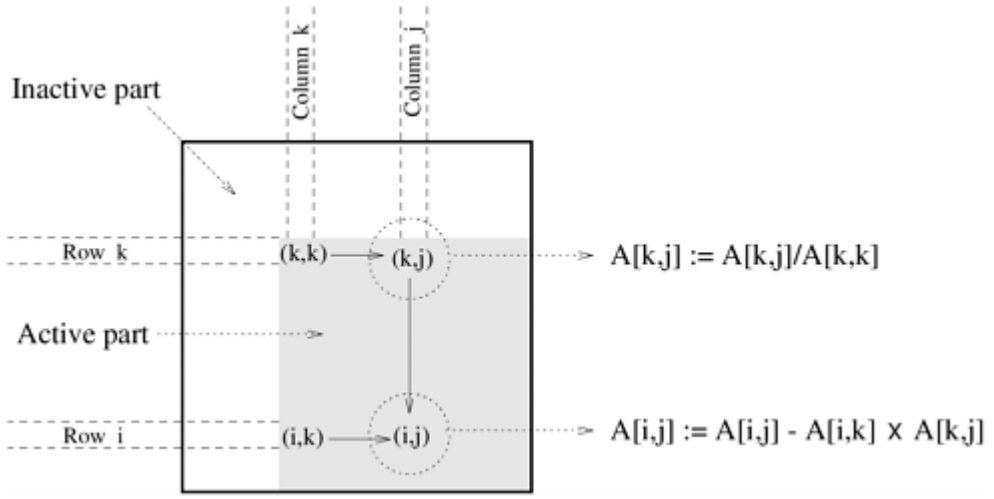


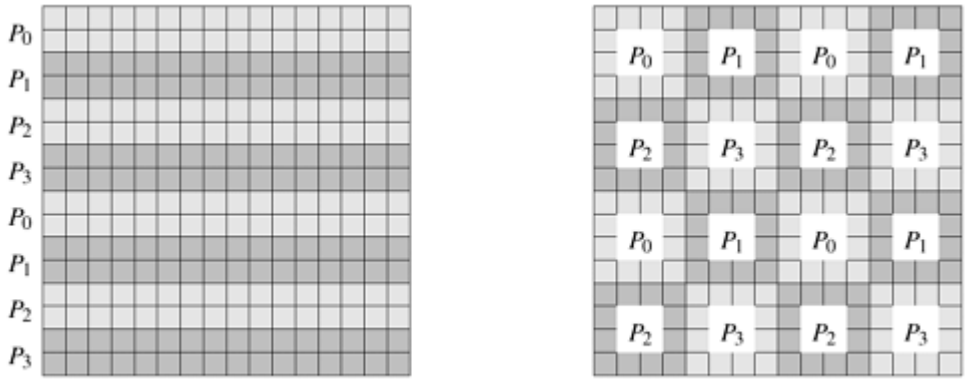
Figure 3.29. A naive mapping of LU factorization tasks onto processes based on a two-dimensional block distribution.

P_0 T_1	**P_3** T_4	**P_6** T_5
P_1 T_2	**P_4** T_6 T_{10}	**P_7** T_8 T_{12}
P_2 T_3	**P_5** T_7 T_{11}	**P_8** T_9 T_{13} T_{14}

The *block-cyclic distribution* is a variation of the block distribution scheme that can be used to alleviate the load-imbalance and idling problems. A detailed description of LU factorization with block-cyclic mapping is covered in Chapter 8, where it is shown how a block-cyclic mapping leads to a substantially more balanced work distribution than in Figure 3.29. The central idea behind a block-cyclic distribution is to partition an array into many more blocks than the number of available processes. Then we assign the partitions (and the associated tasks) to processes in a round-robin manner so that each process gets several non-adjacent blocks. More precisely, in a one-dimensional block-cyclic distribution of a matrix among p processes, the rows (columns) of an $n \times n$ matrix are divided into αp groups of $n/(\alpha p)$ consecutive rows (columns), where $1 \leq \alpha \leq n/p$. Now, these blocks are distributed among the p processes in a wraparound fashion such that block b_i is assigned to process $P_{i \% p}$ ('%' is the

modulo operator). This distribution assigns α blocks of the matrix to each process, but each subsequent block that gets assigned to the same process is p blocks away. We can obtain a two-dimensional block-cyclic distribution of an n x n array by partitioning it into square blocks of size $\alpha\sqrt{p} \times \alpha\sqrt{p}$ and distributing them on a hypothetical $\sqrt{p} \times \sqrt{p}$ array of processes in a wraparound fashion. Similarly, the block-cyclic distribution can be extended to arrays of higher dimensions. Figure 3.30 illustrates one- and two-dimensional block cyclic distributions of a two-dimensional array.

Figure 3.30. Examples of one- and two-dimensional block-cyclic distributions among four processes. (a) The rows of the array are grouped into blocks each consisting of two rows, resulting in eight blocks of rows. These blocks are distributed to four processes in a wraparound fashion. (b) The matrix is blocked into 16 blocks each of size 4 x 4, and it is mapped onto a 2 x 2 grid of processes in a wraparound fashion.



The reason why a block-cyclic distribution is able to significantly reduce the amount of idling is that all processes have a sampling of tasks from all parts of the matrix. As a result, even if different parts of the matrix require different amounts of work, the overall work on each process balances out. Also, since the tasks assigned to a process belong to different parts of the matrix, there is a good chance that at least some of them are ready for execution at any given time.

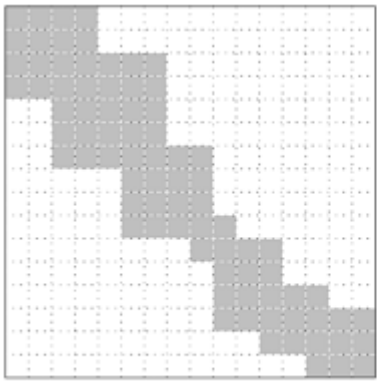
Note that if we increase α to its upper limit of n/p, then each block is a single row (column) of the matrix in a one-dimensional block-cyclic distribution and a single element of the matrix in a two-dimensional block-cyclic distribution. Such a distribution is known as a *cyclic distribution*. A cyclic distribution is an extreme case of a block-cyclic distribution and can result in an almost perfect load balance due to the extreme fine-grained underlying decomposition. However, since a process does not have any contiguous data to work on, the resulting lack of locality may result in serious performance penalties. Additionally, such a decomposition usually leads to a high degree of interaction relative to the amount computation in each task. The lower limit of 1 for the value of α results in maximum locality and interaction optimality, but the distribution degenerates to a block distribution. Therefore, an appropriate value of α must be used to strike a balance between interaction conservation and load balance.

As in the case of block-distributions, higher dimensional block-cyclic distributions are usually preferable as they tend to incur a lower volume of inter-task interaction.

Randomized Block Distributions

A block-cyclic distribution may not always be able to balance computations when the distribution of work has some special patterns. For example, consider the sparse matrix shown in Figure 3.31(a) in which the shaded areas correspond to regions containing nonzero elements. If this matrix is distributed using a two-dimensional block-cyclic distribution, as illustrated in Figure 3.31(b), then we will end up assigning more non-zero blocks to the diagonal processes P_0, P_5, P_{10}, and P_{15} than on any other processes. In fact some processes, like P_{12}, will not get any work.

Figure 3.31. Using the block-cyclic distribution shown in (b) to distribute the computations performed in array (a) will lead to load imbalances.



(a)

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}

(b)

Randomized block distribution, a more general form of the block distribution, can be used in situations illustrated in Figure 3.31. Just like a block-cyclic distribution, load balance is sought by partitioning the array into many more blocks than the number of available processes. However, the blocks are uniformly and randomly distributed among the processes. A one-dimensional randomized block distribution can be achieved as follows. A vector V of length ap (which is equal to the number of blocks) is used and $V[j]$ is set to j for $0 \leq j < ap$. Now, V is randomly permuted, and process P_i is assigned the blocks stored in $V[ia \ldots (i + 1)a - 1]$. Figure 3.32 illustrates this for $p = 4$ and $a = 3$. A two-dimensional randomized block distribution of an n x n array can be computed similarly by randomly permuting two vectors of length $\alpha\sqrt{p}$ each and using them to choose the row and column indices of the blocks to be assigned to each process. As illustrated in Figure 3.33, the random block distribution is more effective in load balancing the computations performed in Figure 3.31.

Figure 3.32. A one-dimensional randomized block mapping of 12 blocks onto four process (i.e., $a = 3$).

V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

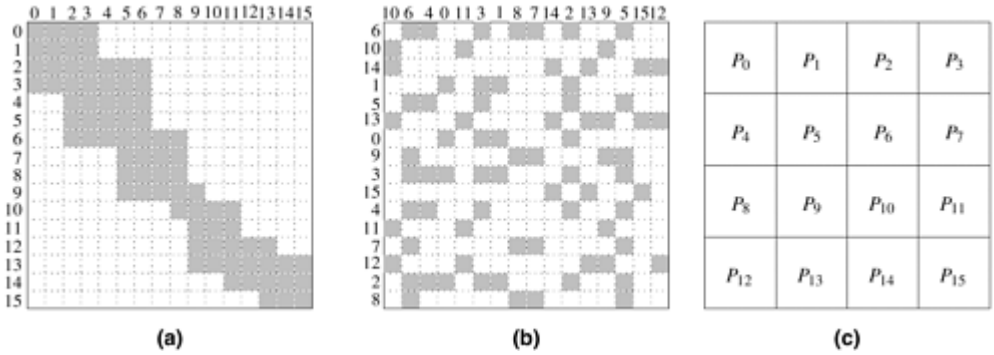
random(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]

mapping = 8 2 6 0 3 7 11 1 9 5 4 10

P_0 P_1 P_2 P_3

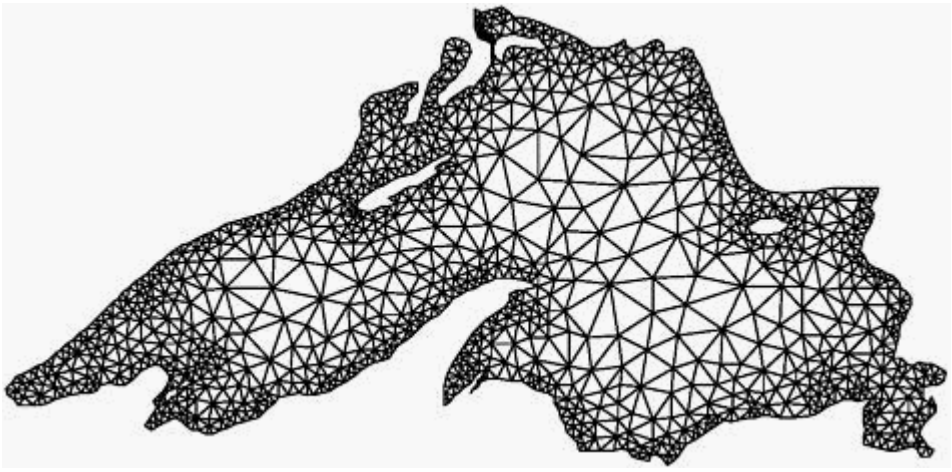
Figure 3.33. Using a two-dimensional random block distribution shown

in (b) to distribute the computations performed in array (a), as shown in (c).



Graph Partitioning The array-based distribution schemes that we described so far are quite effective in balancing the computations and minimizing the interactions for a wide range of algorithms that use dense matrices and have structured and regular interaction patterns. However, there are many algorithms that operate on sparse data structures and for which the pattern of interaction among data elements is data dependent and highly irregular. Numerical simulations of physical phenomena provide a large source of such type of computations. In these computations, the physical domain is discretized and represented by a mesh of elements. The simulation of the physical phenomenon being modeled then involves computing the values of certain physical quantities at each mesh point. The computation at a mesh point usually requires data corresponding to that mesh point and to the points that are adjacent to it in the mesh. For example, Figure 3.34 shows a mesh imposed on Lake Superior. The simulation of a physical phenomenon such the dispersion of a water contaminant in the lake would now involve computing the level of contamination at each vertex of this mesh at various intervals of time.

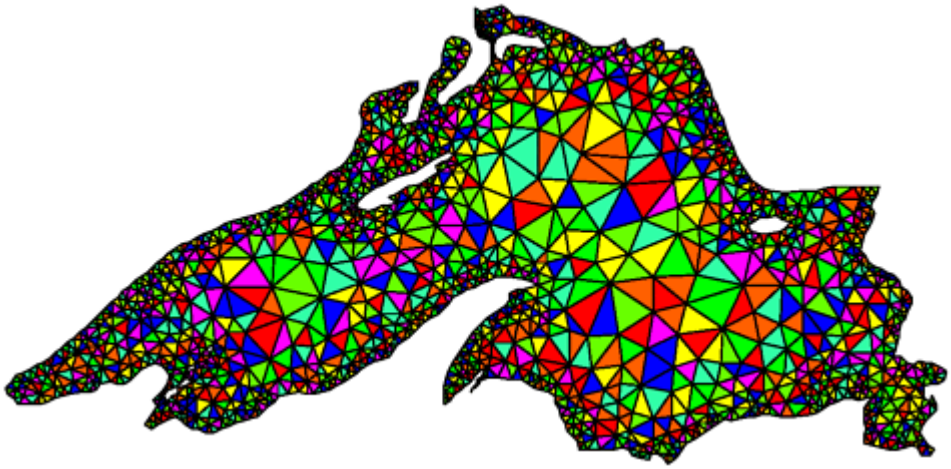
Figure 3.34. A mesh used to model Lake Superior.



Since, in general, the amount of computation at each point is the same, the load can be easily balanced by simply assigning the same number of mesh points to each process. However, if a distribution of the mesh points to processes does not strive to keep nearby mesh points together, then it may lead to high interaction overheads due to excessive data sharing. For example, if each process receives a random set of points as illustrated in Figure 3.35, then each process will need to access a large set of points belonging to other processes to complete computations for its assigned portion of the mesh.

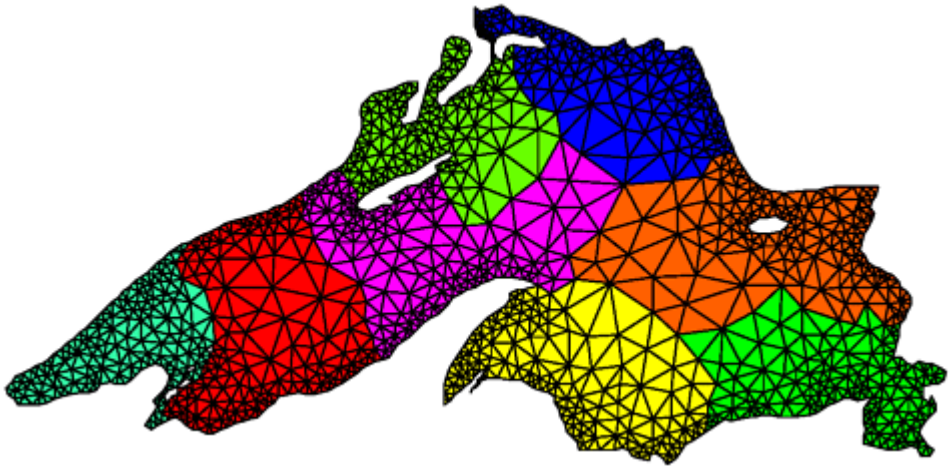
Figure 3.35. A random distribution of the mesh elements to eight

processes.



Ideally, we would like to distribute the mesh points in a way that balances the load and at the same time minimizes the amount of data that each process needs to access in order to complete its computations. Therefore, we need to partition the mesh into *p* parts such that each part contains roughly the same number of mesh-points or vertices, and the number of edges that cross partition boundaries (i.e., those edges that connect points belonging to two different partitions) is minimized. Finding an exact optimal partition is an NP-complete problem. However, algorithms that employ powerful heuristics are available to compute reasonable partitions. After partitioning the mesh in this manner, each one of these *p* partitions is assigned to one of the *p* processes. As a result, each process is assigned a contiguous region of the mesh such that the total number of mesh points that needs to be accessed across partition boundaries is minimized. Figure 3.36 shows a good partitioning of the Lake Superior mesh – the kind that a typical graph partitioning software would generate.

Figure 3.36. A distribution of the mesh elements to eight processes, by using a graph-partitioning algorithm.



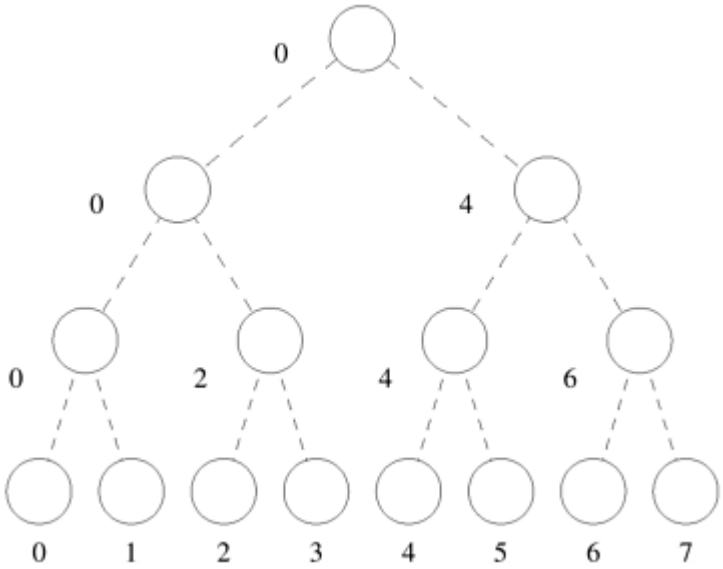
Mappings Based on Task Partitioning

A mapping based on partitioning a task-dependency graph and mapping its nodes onto processes can be used when the computation is naturally expressible in the form of a static task-dependency graph with tasks of known sizes. As usual, this mapping must seek to achieve

the often conflicting objectives of minimizing idle time and minimizing the interaction time of the parallel algorithm. Determining an optimal mapping for a general task-dependency graph is an NP-complete problem. However, specific situations often lend themselves to a simpler optimal or acceptable approximate solution.

As a simple example of a mapping based on task partitioning, consider a task-dependency graph that is a perfect binary tree. Such a task-dependency graph can occur in practical problems with recursive decomposition, such as the decomposition for finding the minimum of a list of numbers (Figure 3.9). Figure 3.37 shows a mapping of this task-dependency graph onto eight processes. It is easy to see that this mapping minimizes the interaction overhead by mapping many interdependent tasks onto the same process (i.e., the tasks along a straight branch of the tree) and others on processes only one communication link away from each other. Although there is some inevitable idling (e.g., when process 0 works on the root task, all other processes are idle), this idling is inherent in the task-dependency graph. The mapping shown in Figure 3.37 does not introduce any further idling and all tasks that are permitted to be concurrently active by the task-dependency graph are mapped onto different processes for parallel execution.

Figure 3.37. Mapping of a binary tree task-dependency graph onto a hypercube of processes.



For some problems, an approximate solution to the problem of finding a good mapping can be obtained by partitioning the task-interaction graph. In the problem of modeling contaminant dispersion in Lake Superior discussed earlier in the context of data partitioning, we can define tasks such that each one of them is responsible for the computations associated with a certain mesh point. Now the mesh used to discretize the lake also acts as a task-interaction graph. Therefore, for this problem, using graph-partitioning to find a good mapping can also be viewed as task partitioning.

Another similar problem where task partitioning is applicable is that of sparse matrix-vector multiplication discussed in Section 3.1.2. A simple mapping of the task-interaction graph of Figure 3.6 is shown in Figure 3.38. This mapping assigns tasks corresponding to four consecutive entries of b to each process. Figure 3.39 shows another partitioning for the task-interaction graph of the sparse matrix vector multiplication problem shown in Figure 3.6 for three processes. The list Ci contains the indices of b that the tasks on Process i need to access from tasks mapped onto other processes. A quick comparison of the lists $C0$, $C1$, and $C2$ in the two cases readily reveals that the mapping based on partitioning the task interaction graph

entails fewer exchanges of elements of *b* between processes than the naive mapping.

Figure 3.38. A mapping for sparse matrix-vector multiplication onto three processes. The list C*i* contains the indices of *b* that Process *i* needs to access from other processes.

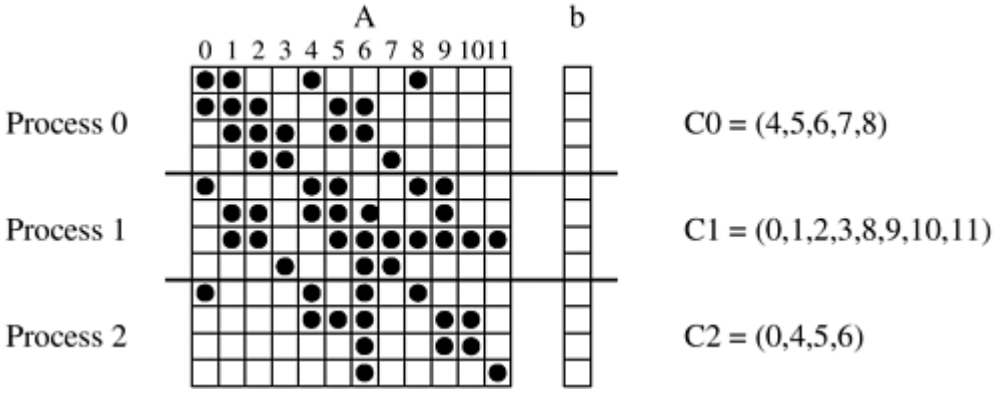
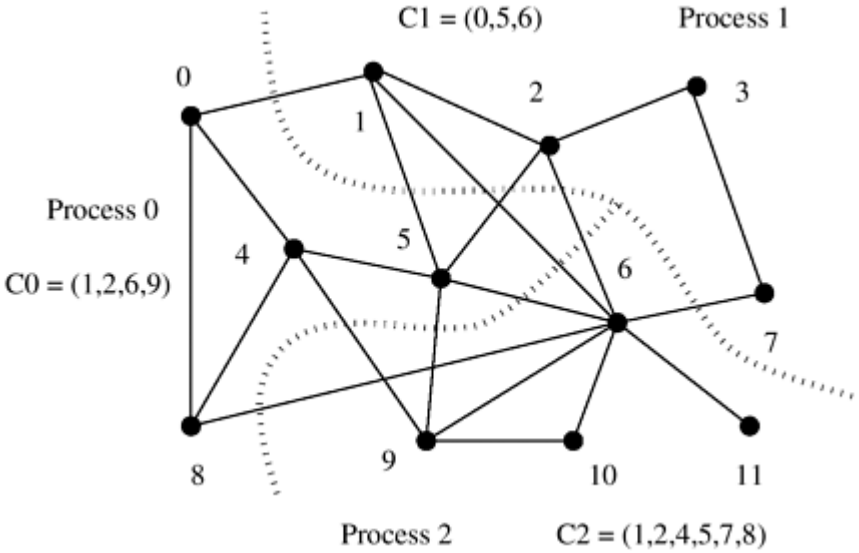


Figure 3.39. Reducing interaction overhead in sparse matrix-vector multiplication by partitioning the task-interaction graph.

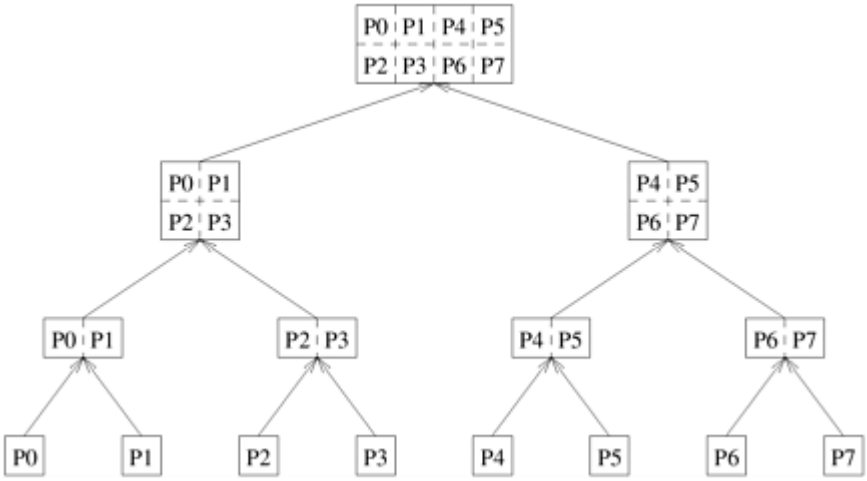


Hierarchical Mappings

Certain algorithms are naturally expressed as task-dependency graphs; however, a mapping based solely on the task-dependency graph may suffer from load-imbalance or inadequate concurrency. For example, in the binary-tree task-dependency graph of Figure 3.37, only a few tasks are available for concurrent execution in the top part of the tree. If the tasks are large enough, then a better mapping can be obtained by a further decomposition of the tasks into smaller subtasks. In the case where the task-dependency graph is a binary tree with four levels, the root task can be partitioned among eight processes, the tasks at the next level can be partitioned among four processes each, followed by tasks partitioned among two processes each at the next level. The eight leaf tasks can have a one-to-one mapping onto the processes. Figure 3.40 illustrates such a hierarchical mapping. Parallel quicksort introduced in Example 3.4 has a task-dependency graph similar to the one shown in Figure 3.37, and hence is an ideal

candidate for a hierarchical mapping of the type shown in Figure 3.40.

Figure 3.40. An example of hierarchical mapping of a task-dependency graph. Each node represented by an array is a supertask. The partitioning of the arrays represents subtasks, which are mapped onto eight processes.



An important practical problem to which the hierarchical mapping example discussed above applies directly is that of sparse matrix factorization. The high-level computations in sparse matrix factorization are guided by a task-dependency graph which is known as an *elimination graph* (*elimination tree* if the matrix is symmetric). However, the tasks in the elimination graph (especially the ones closer to the root) usually involve substantial computations and are further decomposed into subtasks using data-decomposition. A hierarchical mapping, using task partitioning at the top layer and array partitioning at the bottom layer, is then applied to this hybrid decomposition. In general, a hierarchical mapping can have many layers and different decomposition and mapping techniques may be suitable for different layers.

3.4.2 Schemes for Dynamic Mapping

Dynamic mapping is necessary in situations where a static mapping may result in a highly imbalanced distribution of work among processes or where the task-dependency graph itself if dynamic, thus precluding a static mapping. Since the primary reason for using a dynamic mapping is balancing the workload among processes, dynamic mapping is often referred to as dynamic load-balancing. Dynamic mapping techniques are usually classified as either *centralized* or *distributed*.

Centralized Schemes

In a centralized dynamic load balancing scheme, all executable tasks are maintained in a common central data structure or they are maintained by a special process or a subset of processes. If a special process is designated to manage the pool of available tasks, then it is often referred to as the *master* and the other processes that depend on the master to obtain work are referred to as *slaves*. Whenever a process has no work, it takes a portion of available work from the central data structure or the master process. Whenever a new task is generated, it is added to this centralized data structure or reported to the master process. Centralized load-balancing schemes are usually easier to implement than distributed schemes, but may have

limited scalability. As more and more processes are used, the large number of accesses to the common data structure or the master process tends to become a bottleneck.

As an example of a scenario where centralized mapping may be applicable, consider the problem of sorting the entries in each row of an n x n matrix A. Serially, this can be accomplished by the following simple program segment:

```
1    for (i=1; i<n; i++)
2      sort(A[i],n);
```

Recall that the time to sort an array using some of the commonly used sorting algorithms, such as quicksort, can vary significantly depending on the initial order of the elements to be sorted. Therefore, each iteration of the loop in the program shown above can take different amounts of time. A naive mapping of the task of sorting an equal number of rows to each process may lead to load-imbalance. A possible solution to the potential load-imbalance problem in this case would be to maintain a central pool of indices of the rows that have yet to be sorted. Whenever a process is idle, it picks up an available index, deletes it, and sorts the row with that index, as long as the pool of indices is not empty. This method of scheduling the independent iterations of a loop among parallel processes is known as *self scheduling*.

The assignment of a single task to a process at a time is quite effective in balancing the computation; however, it may lead to bottlenecks in accessing the shared work queue, especially if each task (i.e., each loop iteration in this case) does not require a large enough amount of computation. If the average size of each task is M, and it takes Δ time to assign work to a process, then at most M/Δ processes can be kept busy effectively. The bottleneck can be eased by getting more than one task at a time. In *chunk scheduling*, every time a process runs out of work it gets a group of tasks. The potential problem with such a scheme is that it may lead to load-imbalances if the number of tasks (i.e., chunk) assigned in a single step is large. The danger of load-imbalance due to large chunk sizes can be reduced by decreasing the chunk-size as the program progresses. That is, initially the chunk size is large, but as the number of iterations left to be executed decreases, the chunk size also decreases. A variety of schemes have been developed for gradually adjusting the chunk size, that decrease the chunk size either linearly or non-linearly.

Distributed Schemes

In a distributed dynamic load balancing scheme, the set of executable tasks are distributed among processes which exchange tasks at run time to balance work. Each process can send work to or receive work from any other process. These methods do not suffer from the bottleneck associated with the centralized schemes. Some of the critical parameters of a distributed load balancing scheme are as follows:

- How are the sending and receiving processes paired together?
- Is the work transfer initiated by the sender or the receiver?
- How much work is transferred in each exchange? If too little work is transferred, then the receiver may not receive enough work and frequent transfers resulting in excessive interaction may be required. If too much work is transferred, then the sender may be out of work soon, again resulting in frequent transfers.
- When is the work transfer performed? For example, in receiver initiated load balancing, work may be requested when the process has actually run out of work or when the receiver has too little work left and anticipates being out of work soon.

A detailed study of each of these parameters is beyond the scope of this chapter. These load balancing schemes will be revisited in the context of parallel algorithms to which they apply when we discuss these algorithms in the later chapters – in particular, Chapter 11 in the context of parallel search algorithms.

Suitability to Parallel Architectures

Note that, in principle, both centralized and distributed mapping schemes can be implemented in both message-passing and shared-address-space paradigms. However, by its very nature any dynamic load balancing scheme requires movement of tasks from one process to another. Hence, for such schemes to be effective on message-passing computers, the size of the tasks in terms of computation should be much higher than the size of the data associated with the tasks. In a shared-address-space paradigm, the tasks don't need to be moved explicitly, although there is some implied movement of data to local caches or memory banks of processes. In general, the computational granularity of tasks to be moved can be much smaller on shared-address architecture than on message-passing architectures.

3.5 Methods for Containing Interaction Overheads

As noted earlier, reducing the interaction overhead among concurrent tasks is important for an efficient parallel program. The overhead that a parallel program incurs due to interaction among its processes depends on many factors, such as the volume of data exchanged during interactions, the frequency of interaction, the spatial and temporal pattern of interactions, etc.

In this section, we will discuss some general techniques that can be used to reduce the interaction overheads incurred by parallel programs. These techniques manipulate one or more of the three factors above in order to reduce the interaction overhead. Some of these are applicable while devising the decomposition and mapping schemes for the algorithms and some are applicable while programming the algorithm in a given paradigm. All techniques may not be applicable in all parallel programming paradigms and some of them may require support from the underlying hardware.

3.5.1 Maximizing Data Locality

In most nontrivial parallel programs, the tasks executed by different processes require access to some common data. For example, in sparse matrix-vector multiplication $y = Ab$, in which tasks correspond to computing individual elements of vector y (Figure 3.6), all elements of the input vector b need to be accessed by multiple tasks. In addition to sharing the original input data, interaction may result if processes require data generated by other processes. The interaction overheads can be reduced by using techniques that promote the use of local data or data that have been recently fetched. Data locality enhancing techniques encompass a wide range of schemes that try to minimize the volume of nonlocal data that are accessed, maximize the reuse of recently accessed data, and minimize the frequency of accesses. In many cases, these schemes are similar in nature to the data reuse optimizations often performed in modern cache based microprocessors.

Minimize Volume of Data-Exchange A fundamental technique for reducing the interaction overhead is to minimize the overall volume of shared data that needs to be accessed by concurrent processes. This is akin to maximizing the temporal data locality, i.e., making as many of the consecutive references to the same data as possible. Clearly, performing as much of the computation as possible using locally available data obviates the need for bringing in more data into local memory or cache for a process to perform its tasks. As discussed previously, one way of achieving this is by using appropriate decomposition and mapping schemes. For example, in the case of matrix multiplication, we saw that by using a two-dimensional mapping of the computations to the processes we were able to reduce the amount of shared data (i.e., matrices A and B) that needs to be accessed by each task to $2n^2/\sqrt{p}$ as opposed to $n^2/p + n^2$ required by a one-dimensional mapping (Figure 3.26). In general, using higher dimensional distribution often helps in reducing the volume of nonlocal data that needs to be accessed.

Another way of decreasing the amount of shared data that are accessed by multiple processes is to use local data to store intermediate results, and perform the shared data access to only place the final results of the computation. For example, consider computing the dot product of two vectors of length n in parallel such that each of the p tasks multiplies n/p pairs of elements. Rather than adding each individual product of a pair of numbers to the final result, each task can first create a partial dot product of its assigned portion of the vectors of length n/p in its

own local location, and only access the final shared location once to add this partial result. This will reduce the number of accesses to the shared location where the result is stored to p from n.

Minimize Frequency of Interactions Minimizing interaction frequency is important in reducing the interaction overheads in parallel programs because there is a relatively high startup cost associated with each interaction on many architectures. Interaction frequency can be reduced by restructuring the algorithm such that shared data are accessed and used in large pieces. Thus, by amortizing the startup cost over large accesses, we can reduce the overall interaction overhead, even if such restructuring does not necessarily reduce the overall volume of shared data that need to be accessed. This is akin to increasing the spatial locality of data access, i.e., ensuring the proximity of consecutively accessed data locations. On a shared-address-space architecture, each time a word is accessed, an entire cache line containing many words is fetched. If the program is structured to have spatial locality, then fewer cache lines are accessed. On a message-passing system, spatial locality leads to fewer message-transfers over the network because each message can transfer larger amounts of useful data. The number of messages can sometimes be reduced further on a message-passing system by combining messages between the same source-destination pair into larger messages if the interaction pattern permits and if the data for multiple messages are available at the same time, albeit in separate data structures.

Sparse matrix-vector multiplication is a problem whose parallel formulation can use this technique to reduce interaction overhead. In typical applications, repeated sparse matrix-vector multiplication is performed with matrices of the same nonzero pattern but different numerical nonzero values. While solving this problem in parallel, a process interacts with others to access elements of the input vector that it may need for its local computation. Through a one-time scanning of the nonzero pattern of the rows of the sparse matrix that a process is responsible for, it can determine exactly which elements of the input vector it needs and from which processes. Then, before starting each multiplication, a process can first collect all the nonlocal entries of the input vector that it requires, and then perform an interaction-free multiplication. This strategy is far superior than trying to access a nonlocal element of the input vector as and when required in the computation.

3.5.2 Minimizing Contention and Hot Spots

Our discussion so far has been largely focused on reducing interaction overheads by directly or indirectly reducing the frequency and volume of data transfers. However, the data-access and inter-task interaction patterns can often lead to contention that can increase the overall interaction overhead. In general, contention occurs when multiple tasks try to access the same resources concurrently. Multiple simultaneous transmissions of data over the same interconnection link, multiple simultaneous accesses to the same memory block, or multiple processes sending messages to the same process at the same time, can all lead to contention. This is because only one of the multiple operations can proceed at a time and the others are queued and proceed sequentially.

Consider the problem of multiplying two matrices $C = AB$, using the two-dimensional partitioning shown in Figure 3.26(b). Let p be the number of tasks with a one-to-one mapping of tasks onto processes. Let each task be responsible for computing a unique $C_{i,j}$, for $0 \leq i, j < \sqrt{p}$. The straightforward way of performing this computation is for $C_{i,j}$ to be computed according to the following formula (written in matrix-block notation):

Equation 3.1

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} * B_{k,j}$$

Looking at the memory access patterns of the above equation, we see that at any one of the $\sqrt{p}$ steps, $\sqrt{p}$ tasks will be accessing the same block of A and B. In particular, all the tasks that work on the same row of C will be accessing the same block of A. For example, all $\sqrt{p}$ processes computing $C_{0,0}, C_{0,1}, \ldots, C_{0,\sqrt{p}-1}$ will attempt to read $A_{0,0}$ at once. Similarly, all the tasks working on the same column of C will be accessing the same block of B. The need to concurrently access these blocks of matrices A and B will create contention on both NUMA shared-address-space and message-passing parallel architectures.

One way of reducing contention is to redesign the parallel algorithm to access data in contention-free patterns. For the matrix multiplication algorithm, this contention can be eliminated by modifying the order in which the block multiplications are performed in Equation 3.1. A contention-free way of performing these block-multiplications is to compute $C_{i,j}$ by using the formula

Equation 3.2

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} * B_{(i+j+k)\%\sqrt{p},j}$$

where '%' denotes the modulo operation. By using this formula, all the tasks $P_{*,j}$ that work on the same row of C will be accessing block $A_{*,(*+j+k)\%\sqrt{p}}$, which is different for each task. Similarly, all the tasks $P_{i,*}$ that work on the same column of C will be accessing block $B_{(i+*+k)\%\sqrt{p},*}$, which is also different for each task. Thus, by simply rearranging the order in which the block-multiplications are performed, we can completely eliminate the contention. For example, among the processes computing the first block row of C, the process computing $C_{0,j}$ will access $A_{0,j}$ from the first block row of A instead of $A_{0,0}$.

Centralized schemes for dynamic mapping (Section 3.4.2) are a frequent source of contention for shared data structures or communication channels leading to the master process. The contention may be reduced by choosing a distributed mapping scheme over a centralized one, even though the former may be harder to implement.

3.5.3 Overlapping Computations with Interactions

The amount of time that processes spend waiting for shared data to arrive or to receive additional work after an interaction has been initiated can be reduced, often substantially, by doing some useful computations during this waiting time. There are a number of techniques that can be used to overlap computations with interactions.

A simple way of overlapping is to initiate an interaction early enough so that it is completed before it is needed for computation. To achieve this, we must be able to identify computations that can be performed before the interaction and do not depend on it. Then the parallel program must be structured to initiate the interaction at an earlier point in the execution than it is needed in the original algorithm. Typically, this is possible if the interaction pattern is spatially and temporally static (and therefore, predictable) or if multiple tasks that are ready for execution are available on the same process so that if one blocks to wait for an interaction to complete, the process can work on another task. The reader should note that by increasing the

number of parallel tasks to promote computation-interaction overlap, we are reducing the granularity of the tasks, which in general tends to increase overheads. Therefore, this technique must be used judiciously.

In certain dynamic mapping schemes, as soon as a process runs out of work, it requests and gets additional work from another process. It then waits for the request to be serviced. If the process can anticipate that it is going to run out of work and initiate a work transfer interaction in advance, then it may continue towards finishing the tasks at hand while the request for more work is being serviced. Depending on the problem, estimating the amount of remaining work may be easy or hard.

In most cases, overlapping computations with interaction requires support from the programming paradigm, the operating system, and the hardware. The programming paradigm must provide a mechanism to allow interactions and computations to proceed concurrently. This mechanism should be supported by the underlying hardware. Disjoint address-space paradigms and architectures usually provide this support via non-blocking message passing primitives. The programming paradigm provides functions for sending and receiving messages that return control to the user's program before they have actually completed. Thus, the program can use these primitives to initiate the interactions, and then proceed with the computations. If the hardware permits computation to proceed concurrently with message transfers, then the interaction overhead can be reduced significantly.

On a shared-address-space architecture, the overlapping of computations and interaction is often assisted by prefetching hardware. In this case, an access to shared data is nothing more than a regular load or store instruction. The prefetch hardware can anticipate the memory addresses that will need to be accessed in the immediate future, and can initiate the access in advance of when they are needed. In the absence of prefetching hardware, the same effect can be achieved by a compiler that detects the access pattern and places pseudo-references to certain key memory locations before these locations are actually utilized by the computation. The degree of success of this scheme is dependent upon the available structure in the program that can be inferred by the prefetch hardware and by the degree of independence with which the prefetch hardware can function while computation is in progress.

3.5.4 Replicating Data or Computations

Replication of data or computations is another technique that may be useful in reducing interaction overheads.

In some parallel algorithms, multiple processes may require frequent read-only access to shared data structure, such as a hash-table, in an irregular pattern. Unless the additional memory requirements are prohibitive, it may be best in a situation like this to replicate a copy of the shared data structure on each process so that after the initial interaction during replication, all subsequent accesses to this data structure are free of any interaction overhead.

In the shared-address-space paradigm, replication of frequently accessed read-only data is often affected by the caches without explicit programmer intervention. Explicit data replication is particularly suited for architectures and programming paradigms in which read-only access to shared data is significantly more expensive or harder to express than local data accesses. Therefore, the message-passing programming paradigm benefits the most from data replication, which may reduce interaction overhead and also significantly simplify the writing of the parallel program.

Data replication, however, does not come without its own cost. Data replication increases the memory requirements of a parallel program. The aggregate amount of memory required to store the replicated data increases linearly with the number of concurrent processes. This may

limit the size of the problem that can be solved on a given parallel computer. For this reason, data replication must be used selectively to replicate relatively small amounts of data.

In addition to input data, the processes in a parallel program often share intermediate results. In some situations, it may be more cost-effective for a process to compute these intermediate results than to get them from another process that generates them. In such situations, interaction overhead can be traded for replicated computation. For example, while performing the Fast Fourier Transform (see Section 13.2.3 for more details), on an N-point series, N distinct powers of w or "twiddle factors" are computed and used at various points in the computation. In a parallel implementation of FFT, different processes require overlapping subsets of these N twiddle factors. In a message-passing paradigm, it is best for each process to locally compute all the twiddle factors it needs. Although the parallel algorithm may perform many more twiddle factor computations than the serial algorithm, it may still be faster than sharing the twiddle factors.

3.5.5 Using Optimized Collective Interaction Operations

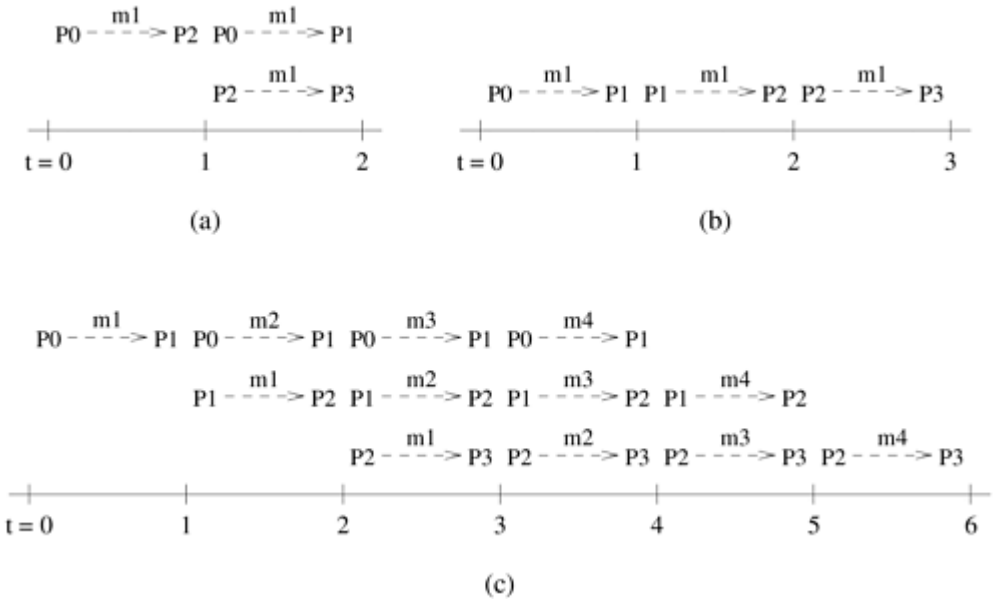
As discussed in Section 3.3.2, often the interaction patterns among concurrent activities are static and regular. A class of such static and regular interaction patterns are those that are performed by groups of tasks, and they are used to achieve regular data accesses or to perform certain type of computations on distributed data. A number of key such *collective* interaction operations have been identified that appear frequently in many parallel algorithms. Broadcasting some data to all the processes or adding up numbers, each belonging to a different process, are examples of such collective operations. The collective data-sharing operations can be classified into three categories. The first category contains operations that are used by the tasks to access data, the second category of operations are used to perform some communication-intensive computations, and finally, the third category is used for synchronization.

Highly optimized implementations of these collective operations have been developed that minimize the overheads due to data transfer as well as contention. Chapter 4 describes algorithms for implementing some of the commonly used collective interaction operations. Optimized implementations of these operations are available in library form from the vendors of most parallel computers, e.g., MPI (message passing interface). As a result, the algorithm designer does not need to think about how these operations are implemented and needs to focus only on the functionality achieved by these operations. However, as discussed in Section 3.5.6, sometimes the interaction pattern may make it worthwhile for the parallel programmer to implement one's own collective communication procedure.

3.5.6 Overlapping Interactions with Other Interactions

If the data-transfer capacity of the underlying hardware permits, then overlapping interactions between multiple pairs of processes can reduce the effective volume of communication. As an example of overlapping interactions, consider the commonly used collective communication operation of one-to-all broadcast in a message-passing paradigm with four processes P_0, P_1, P_2, and P_3. A commonly used algorithm to broadcast some data from P_0 to all other processes works as follows. In the first step, P_0 sends the data to P_2. In the second step, P_0 sends the data to P_1, and concurrently, P_2 sends the same data that it had received from P_0 to P_3. The entire operation is thus complete in two steps because the two interactions of the second step require only one time step. This operation is illustrated in Figure 3.41(a). On the other hand, a naive broadcast algorithm would send the data from P_0 to P_1 to P_2 to P_3, thereby consuming three steps as illustrated in Figure 3.41(b).

Figure 3.41. Illustration of overlapping interactions in broadcasting data from one to four processes.



Interestingly, however, there are situations when the naive broadcast algorithm shown in Figure 3.41(b) may be adapted to actually increase the amount of overlap. Assume that a parallel algorithm needs to broadcast four data structures one after the other. The entire interaction would require eight steps using the first two-step broadcast algorithm. However, using the naive algorithm accomplishes the interaction in only six steps as shown in Figure 3.41(c). In the first step, P_0 sends the first message to P_1. In the second step P_0 sends the second message to P_1 while P_1 simultaneously sends the first message to P_2. In the third step, P_0 sends the third message to P_1, P_1 sends the second message to P_2, and P_2 sends the first message to P_3. Proceeding similarly in a pipelined fashion, the last of the four messages is sent out of P_0 after four steps and reaches P_3 in six. Since this method is rather expensive for a single broadcast operation, it is unlikely to be included in a collective communication library. However, the programmer must infer from the interaction pattern of the algorithm that in this scenario, it is better to make an exception to the suggestion of Section 3.5.5 and write your own collective communication function.

3.6 Parallel Algorithm Models

Having discussed the techniques for decomposition, mapping, and minimizing interaction overheads, we now present some of the commonly used parallel algorithm models. An algorithm model is typically a way of structuring a parallel algorithm by selecting a decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

3.6.1 The Data-Parallel Model

The *data-parallel model* is one of the simplest algorithm models. In this model, the tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data. This type of parallelism that is a result of identical operations being applied concurrently on different data items is called *data parallelism*. The work may be done in phases and the data operated upon in different phases may be different. Typically, data-parallel computation phases are interspersed with interactions to synchronize the tasks or to get fresh data to the tasks. Since all tasks perform similar computations, the decomposition of the problem into tasks is usually based on data partitioning because a uniform partitioning of data followed by a static mapping is sufficient to guarantee load balance.

Data-parallel algorithms can be implemented in both shared-address-space and message-passing paradigms. However, the partitioned address-space in a message-passing paradigm may allow better control of placement, and thus may offer a better handle on locality. On the other hand, shared-address space can ease the programming effort, especially if the distribution of data is different in different phases of the algorithm.

Interaction overheads in the data-parallel model can be minimized by choosing a locality preserving decomposition and, if applicable, by overlapping computation and interaction and by using optimized collective interaction routines. A key characteristic of data-parallel problems is that for most problems, the degree of data parallelism increases with the size of the problem, making it possible to use more processes to effectively solve larger problems.

An example of a data-parallel algorithm is dense matrix multiplication described in Section 3.1.1. In the decomposition shown in Figure 3.10, all tasks are identical; they are applied to different data.

3.6.2 The Task Graph Model

As discussed in Section 3.1, the computations in any parallel algorithm can be viewed as a task-dependency graph. The task-dependency graph may be either trivial, as in the case of matrix multiplication, or nontrivial (Problem 3.5). However, in certain parallel algorithms, the task-dependency graph is explicitly used in mapping. In the *task graph model*, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs. This model is typically employed to solve problems in which the amount of data associated with the tasks is large relative to the amount of computation associated with them. Usually, tasks are mapped statically to help optimize the cost of data movement among tasks. Sometimes a decentralized dynamic mapping may be used, but even then, the mapping uses the information about the task-dependency graph structure and the interaction pattern of tasks to minimize interaction

overhead. Work is more easily shared in paradigms with globally addressable space, but mechanisms are available to share work in disjoint address space.

Typical interaction-reducing techniques applicable to this model include reducing the volume and frequency of interaction by promoting locality while mapping the tasks based on the interaction pattern of tasks, and using asynchronous interaction methods to overlap the interaction with computation.

Examples of algorithms based on the task graph model include parallel quicksort (Section 9.4.1), sparse matrix factorization, and many parallel algorithms derived via divide-and-conquer decomposition. This type of parallelism that is naturally expressed by independent tasks in a task-dependency graph is called *task parallelism*.

3.6.3 The Work Pool Model

The *work pool* or the *task pool* model is characterized by a dynamic mapping of tasks onto processes for load balancing in which any task may potentially be performed by any process. There is no desired premapping of tasks onto processes. The mapping may be centralized or decentralized. Pointers to the tasks may be stored in a physically shared list, priority queue, hash table, or tree, or they could be stored in a physically distributed data structure. The work may be statically available in the beginning, or could be dynamically generated; i.e., the processes may generate work and add it to the global (possibly distributed) work pool. If the work is generated dynamically and a decentralized mapping is used, then a termination detection algorithm (Section 11.4.4) would be required so that all processes can actually detect the completion of the entire program (i.e., exhaustion of all potential tasks) and stop looking for more work.

In the message-passing paradigm, the work pool model is typically used when the amount of data associated with tasks is relatively small compared to the computation associated with the tasks. As a result, tasks can be readily moved around without causing too much data interaction overhead. The granularity of the tasks can be adjusted to attain the desired level of tradeoff between load-imbalance and the overhead of accessing the work pool for adding and extracting tasks.

Parallelization of loops by chunk scheduling (Section 3.4.2) or related methods is an example of the use of the work pool model with centralized mapping when the tasks are statically available. Parallel tree search where the work is represented by a centralized or distributed data structure is an example of the use of the work pool model where the tasks are generated dynamically.

3.6.4 The Master-Slave Model

In the *master-slave* or the *manager-worker* model, one or more master processes generate work and allocate it to worker processes. The tasks may be allocated *a priori* if the manager can estimate the size of the tasks or if a random mapping can do an adequate job of load balancing. In another scenario, workers are assigned smaller pieces of work at different times. The latter scheme is preferred if it is time consuming for the master to generate work and hence it is not desirable to make all workers wait until the master has generated all work pieces. In some cases, work may need to be performed in phases, and work in each phase must finish before work in the next phases can be generated. In this case, the manager may cause all workers to synchronize after each phase. Usually, there is no desired premapping of work to processes, and any worker can do any job assigned to it. The manager-worker model can be generalized to the hierarchical or multi-level manager-worker model in which the top-level manager feeds large chunks of tasks to second-level managers, who further subdivide the tasks among their

own workers and may perform part of the work themselves. This model is generally equally suitable to shared-address-space or message-passing paradigms since the interaction is naturally two-way; i.e., the manager knows that it needs to give out work and workers know that they need to get work from the manager.

While using the master-slave model, care should be taken to ensure that the master does not become a bottleneck, which may happen if the tasks are too small (or the workers are relatively fast). The granularity of tasks should be chosen such that the cost of doing work dominates the cost of transferring work and the cost of synchronization. Asynchronous interaction may help overlap interaction and the computation associated with work generation by the master. It may also reduce waiting times if the nature of requests from workers is non-deterministic.

3.6.5 The Pipeline or Producer-Consumer Model

In the *pipeline model*, a stream of data is passed on through a succession of processes, each of which perform some task on it. This simultaneous execution of different programs on a data stream is called *stream parallelism*. With the exception of the process initiating the pipeline, the arrival of new data triggers the execution of a new task by a process in the pipeline. The processes could form such pipelines in the shape of linear or multidimensional arrays, trees, or general graphs with or without cycles. A pipeline is a chain of producers and consumers. Each process in the pipeline can be viewed as a consumer of a sequence of data items for the process preceding it in the pipeline and as a producer of data for the process following it in the pipeline. The pipeline does not need to be a linear chain; it can be a directed graph. The pipeline model usually involves a static mapping of tasks onto processes.

Load balancing is a function of task granularity. The larger the granularity, the longer it takes to fill up the pipeline, i.e. for the trigger produced by the first process in the chain to propagate to the last process, thereby keeping some of the processes waiting. However, too fine a granularity may increase interaction overheads because processes will need to interact to receive fresh data after smaller pieces of computation. The most common interaction reduction technique applicable to this model is overlapping interaction with computation.

An example of a two-dimensional pipeline is the parallel LU factorization algorithm, which is discussed in detail in Section 8.3.1.

3.6.6 Hybrid Models

In some cases, more than one model may be applicable to the problem at hand, resulting in a hybrid algorithm model. A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm. In some cases, an algorithm formulation may have characteristics of more than one algorithm model. For instance, data may flow in a pipelined manner in a pattern guided by a task-dependency graph. In another scenario, the major computation may be described by a task-dependency graph, but each node of the graph may represent a supertask comprising multiple subtasks that may be suitable for data-parallel or pipelined parallelism. Parallel quicksort (Sections 3.2.5 and 9.4.1) is one of the applications for which a hybrid model is ideally suited.

3.7 Bibliographic Remarks

Various texts, such as those by Wilson [Wil95], Akl [Akl97], Hwang and Xu [HX98], Wilkinson and Allen [WA99], and Culler and Singh [CSG98], among others, present similar or slightly varying models for parallel programs and steps in developing parallel algorithms. The book by Goedecker and Hoisie [GH01] is among the relatively few textbooks that focus on the practical aspects of writing high-performance parallel programs. Kwok and Ahmad [KA99a, KA99b] present comprehensive surveys of techniques for mapping tasks onto processes.

Most of the algorithms used in this chapter as examples are discussed in detail in other chapters in this book dedicated to the respective class of problems. Please refer to the bibliographic remarks in those chapters for further references on these algorithms.

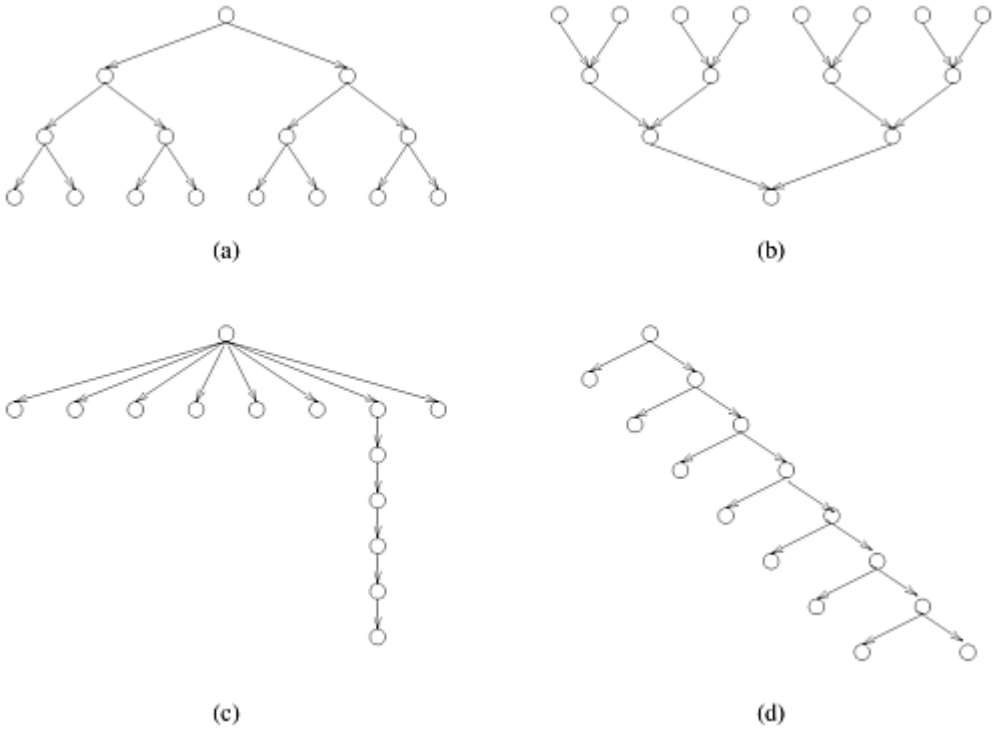
Problems

3.1 In Example 3.2, each union and intersection operation can be performed in time proportional to the sum of the number of records in the two input tables. Based on this, construct the weighted task-dependency graphs corresponding to Figures 3.2 and 3.3, where the weight of each node is equivalent to the amount of work required by the corresponding task. What is the average degree of concurrency of each graph?

3.2 For the task graphs given in Figure 3.42, determine the following:

1. Maximum degree of concurrency.
2. Critical path length.
3. Maximum achievable speedup over one process assuming that an arbitrarily large number of processes is available.
4. The minimum number of processes needed to obtain the maximum possible speedup.
5. The maximum achievable speedup if the number of processes is limited to (a) 2, (b) 4, and (c) 8.

Figure 3.42. Task-dependency graphs for Problem 3.2.



3.3 What are the average degrees of concurrency and critical-path lengths of task-dependency graphs corresponding to the decompositions for matrix multiplication shown in Figures 3.10 and 3.11?

3.4 Let d be the maximum degree of concurrency in a task-dependency graph with t tasks and a critical-path length l. Prove that $\lceil \frac{t}{l} \rceil \leq d \leq t - l + 1$.

3.5 Consider LU factorization of a dense matrix shown in Algorithm 3.3. Figure 3.27 shows the decomposition of LU factorization into 14 tasks based on a two-dimensional partitioning of the matrix A into nine blocks $A_{i,j}$, $1 \leq i, j \leq 3$. The blocks of A are modified into corresponding blocks of L and U as a result of factorization. The diagonal blocks of L are lower triangular submatrices with unit diagonals and the diagonal blocks of U are upper triangular submatrices. Task 1 factors the submatrix $A_{1,1}$ using Algorithm 3.3. Tasks 2 and 3 implement the block versions of the loop on Lines 4–6 of Algorithm 3.3. Tasks 4 and 5 are the upper-triangular counterparts of tasks 2 and 3. The element version of LU factorization in Algorithm 3.3 does not show these steps because the diagonal entries of L are 1; however, a block version must compute a block-row of U as a product of the inverse of the corresponding diagonal block of L with the block-row of A. Tasks 6–9 implement the block version of the loops on Lines 7–11 of Algorithm 3.3. Thus, Tasks 1–9 correspond to the block version of the first iteration of the outermost loop of Algorithm 3.3. The remainder of the tasks complete the factorization of A. Draw a task-dependency graph corresponding to the decomposition shown in Figure 3.27.

3.6 Enumerate the critical paths in the decomposition of LU factorization shown in Figure 3.27.

3.7 Show an efficient mapping of the task-dependency graph of the decomposition shown in Figure 3.27 onto three processes. Prove informally that your mapping is the best possible mapping for three processes.

3.8 Describe and draw an efficient mapping of the task-dependency graph of the decomposition shown in Figure 3.27 onto four processes and prove that your mapping is the best possible mapping for four processes.

3.9 Assuming that each task takes a unit amount of time, [1] which of the two mappings – the one onto three processes or the one onto four processes – solves the problem faster?

[1] In practice, for a block size $b \gg 1$, Tasks 1, 10, and 14 require about $2/3b^3$ arithmetic operations; Tasks 2, 3, 4, 5, 11, and 12 require about b^3 operations; and Tasks 6, 7, 8, 9, and 13 require about $2b^3$ operations.

3.10 Prove that block steps 1 through 14 in Figure 3.27 with block size b (i.e., each $A_{i,j}$, $L_{i,j}$, and $U_{i,j}$ is a b x b submatrix) are mathematically equivalent to running the algorithm of Algorithm 3.3 on an n x n matrix A, where $n = 3b$.

Hint: Using induction on b is one possible approach.

3.11 Figure 3.27 shows the decomposition into 14 tasks of LU factorization of a matrix split into blocks using a 3 x 3 two-dimensional partitioning. If an m x m partitioning is used, derive an expression for the number of tasks $t(m)$ as a function of m in a decomposition along similar lines.

Hint: Show that $t(m) = t(m - 1) + m^2$.

3.12 In the context of Problem 3.11, derive an expression for the maximum degree of concurrency $d(m)$ as a function of m.

3.13 In the context of Problem 3.11, derive an expression for the critical-path length $l(m)$ as a function of m.

3.14 Show efficient mappings for the decompositions for the database query problem shown in Figures 3.2 and 3.3. What is the maximum number of processes that you would use in each case?

3.15 In the algorithm shown in Algorithm 3.4, assume a decomposition such that each execution of Line 7 is a task. Draw a task-dependency graph and a task-interaction graph.

Algorithm 3.4 A sample serial program to be parallelized.

```
1.   procedure FFT_like_pattern(A, n)
2.   begin
3.      m := log2 n;
4.      for j := 0 to m - 1 do
5.          k := 2^j;
6.          for i := 0 to n - 1 do
7.              A[i] := A[i] + A[i XOR 2^j];
8.      endfor
9.   end FFT_like_pattern
```

3.16 In Algorithm 3.4, if $n = 16$, devise a good mapping for 16 processes.

3.17 In Algorithm 3.4, if $n = 16$, devise a good mapping for 8 processes.

3.18 Repeat Problems 3.15, 3.16, and 3.17 if the statement of Line 3 in Algorithm 3.4 is changed to $m = (\log_2 n) - 1$.

3.19 Consider a simplified version of bucket-sort. You are given an array A of n random integers in the range $[1...r]$ as input. The output data consist of r buckets, such that at the end of the algorithm, Bucket i contains indices of all the elements in A that are equal to i.

- Describe a decomposition based on partitioning the input data (i.e., the array A) and an appropriate mapping onto p processes. Describe briefly how the resulting parallel algorithm would work.
- Describe a decomposition based on partitioning the output data (i.e., the set of r buckets) and an appropriate mapping onto p processes. Describe briefly how the resulting parallel algorithm would work.

3.20 In the context of Problem 3.19, which of the two decompositions leads to a better parallel algorithm? Should the relative values of n and r have a bearing on the selection of one of the two decomposition schemes?

3.21 Consider seven tasks with running times of 1, 2, 3, 4, 5, 5, and 10 units, respectively. Assuming that it does not take any time to assign work to a process, compute the best- and worst-case speedup for a centralized scheme for dynamic mapping with two processes.

3.22 Suppose there are M tasks that are being mapped using a centralized dynamic load balancing scheme and we have the following information about these tasks:

- Average task size is 1.
- Minimum task size is 0.

- Maximum task size is m.
- It takes a process Δ time to pick up a task.

Compute the best- and worst-case speedups for self-scheduling and chunk-scheduling assuming that tasks are available in batches of l ($l < M$). What are the actual values of the best- and worst-case speedups for the two scheduling methods when $p = 10$, $\Delta = 0.2$, $m = 20$, $M = 100$, and $l = 2$?

PREVIOUS NEXT

Chapter 4. Basic Communication Operations

In most parallel algorithms, processes need to exchange data with other processes. This exchange of data can significantly impact the efficiency of parallel programs by introducing interaction delays during their execution. For instance, recall from Section 2.5 that it takes roughly $t_s + mt_w$ time for a simple exchange of an m-word message between two processes running on different nodes of an interconnection network with cut-through routing. Here t_s is the latency or the startup time for the data transfer and t_w is the per-word transfer time, which is inversely proportional to the available bandwidth between the nodes. Many interactions in practical parallel programs occur in well-defined patterns involving more than two processes. Often either all processes participate together in a single global interaction operation, or subsets of processes participate in interactions local to each subset. These common basic patterns of interprocess interaction or communication are frequently used as building blocks in a variety of parallel algorithms. Proper implementation of these basic communication operations on various parallel architectures is a key to the efficient execution of the parallel algorithms that use them.

In this chapter, we present algorithms to implement some commonly used communication patterns on simple interconnection networks, such as the linear array, two-dimensional mesh, and the hypercube. The choice of these interconnection networks is motivated primarily by pedagogical reasons. For instance, although it is unlikely that large scale parallel computers will be based on the linear array or ring topology, it is important to understand various communication operations in the context of linear arrays because the rows and columns of meshes are linear arrays. Parallel algorithms that perform rowwise or columnwise communication on meshes use linear array algorithms. The algorithms for a number of communication operations on a mesh are simple extensions of the corresponding linear array algorithms to two dimensions. Furthermore, parallel algorithms using regular data structures such as arrays often map naturally onto one- or two-dimensional arrays of processes. This too makes it important to study interprocess interaction on a linear array or mesh interconnection network. The hypercube architecture, on the other hand, is interesting because many algorithms with recursive interaction patterns map naturally onto a hypercube topology. Most of these algorithms may perform equally well on interconnection networks other than the hypercube, but it is simpler to visualize their communication patterns on a hypercube.

The algorithms presented in this chapter in the context of simple network topologies are practical and are highly suitable for modern parallel computers, even though most such computers are unlikely to have an interconnection network that exactly matches one of the networks considered in this chapter. The reason is that on a modern parallel computer, the time to transfer data of a certain size between two nodes is often independent of the relative location of the nodes in the interconnection network. This homogeneity is afforded by a variety of firmware and hardware features such as randomized routing algorithms and cut-through routing, etc. Furthermore, the end user usually does not have explicit control over mapping processes onto physical processors. Therefore, we assume that the transfer of m words of data between *any* pair of nodes in an interconnection network incurs a cost of $t_s + mt_w$. On most architectures, this assumption is reasonably accurate as long as a free link is available between the source and destination nodes for the data to traverse. However, if many pairs of nodes are communicating simultaneously, then the messages may take longer. This can happen if the number of messages passing through a cross-section of the network exceeds the cross-section bandwidth (Section 2.4.4) of the network. In such situations, we need to adjust the value of t_w

to reflect the slowdown due to congestion. As discussed in Section 2.5.1, we refer to the adjusted value of t_w as effective t_w. We will make a note in the text when we come across communication operations that may cause congestion on certain networks.

As discussed in Section 2.5.2, the cost of data-sharing among processors in the shared-address-space paradigm can be modeled using the same expression $t_s + mt_w$, usually with different values of t_s and t_w relative to each other as well as relative to the computation speed of the processors of the parallel computer. Therefore, parallel algorithms requiring one or more of the interaction patterns discussed in this chapter can be assumed to incur costs whose expression is close to one derived in the context of message-passing.

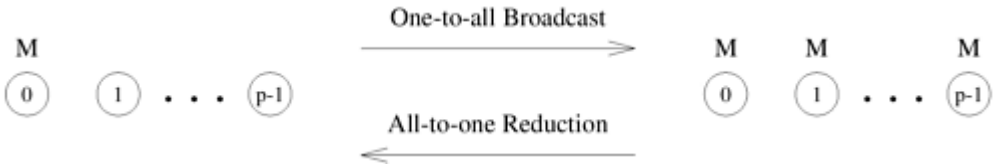
In the following sections we describe various communication operations and derive expressions for their time complexity. We assume that the interconnection network supports cut-through routing (Section 2.5.1) and that the communication time between any pair of nodes is practically independent of of the number of intermediate nodes along the paths between them. We also assume that the communication links are bidirectional; that is, two directly-connected nodes can send messages of size m to each other simultaneously in time $t_s + t_w m$. We assume a single-port communication model, in which a node can send a message on only one of its links at a time. Similarly, it can receive a message on only one link at a time. However, a node can receive a message while sending another message at the same time on the same or a different link.

Many of the operations described here have duals and other related operations that we can perform by using procedures very similar to those for the original operations. The *dual* of a communication operation is the opposite of the original operation and can be performed by reversing the direction and sequence of messages in the original operation. We will mention such operations wherever applicable.

4.1 One-to-All Broadcast and All-to-One Reduction

Parallel algorithms often require a single process to send identical data to all other processes or to a subset of them. This operation is known as *one-to-all broadcast* . Initially, only the source process has the data of size *m* that needs to be broadcast. At the termination of the procedure, there are *p* copies of the initial data – one belonging to each process. The dual of one-to-all broadcast is *all-to-one reduction* . In an all-to-one reduction operation, each of the *p* participating processes starts with a buffer *M* containing *m* words. The data from all processes are combined through an associative operator and accumulated at a single destination process into one buffer of size *m* . Reduction can be used to find the sum, product, maximum, or minimum of sets of numbers – the *i* th word of the accumulated *M* is the sum, product, maximum, or minimum of the *i* th words of each of the original buffers. Figure 4.1 shows one-to-all broadcast and all-to-one reduction among *p* processes.

Figure 4.1. One-to-all broadcast and all-to-one reduction.



One-to-all broadcast and all-to-one reduction are used in several important parallel algorithms including matrix-vector multiplication, Gaussian elimination, shortest paths, and vector inner product. In the following subsections, we consider the implementation of one-to-all broadcast in detail on a variety of interconnection topologies.

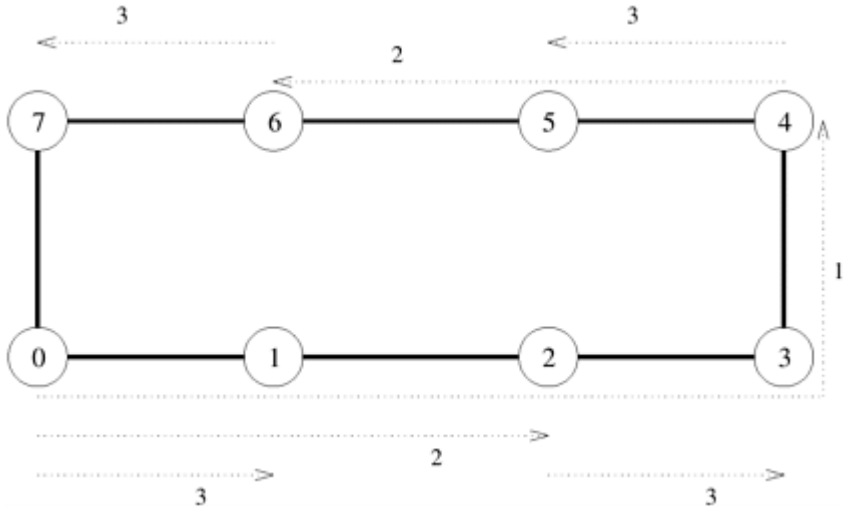
4.1.1 Ring or Linear Array

A naive way to perform one-to-all broadcast is to sequentially send *p* - 1 messages from the source to the other *p* - 1 processes. However, this is inefficient because the source process becomes a bottleneck. Moreover, the communication network is underutilized because only the connection between a single pair of nodes is used at a time. A better broadcast algorithm can be devised using a technique commonly known as *recursive doubling* . The source process first sends the message to another process. Now both these processes can simultaneously send the message to two other processes that are still waiting for the message. By continuing this procedure until all the processes have received the data, the message can be broadcast in log *p* steps.

The steps in a one-to-all broadcast on an eight-node linear array or ring are shown in Figure 4.2 . The nodes are labeled from 0 to 7. Each message transmission step is shown by a numbered, dotted arrow from the source of the message to its destination. Arrows indicating messages sent during the same time step have the same number.

Figure 4.2. One-to-all broadcast on an eight-node ring. Node 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during

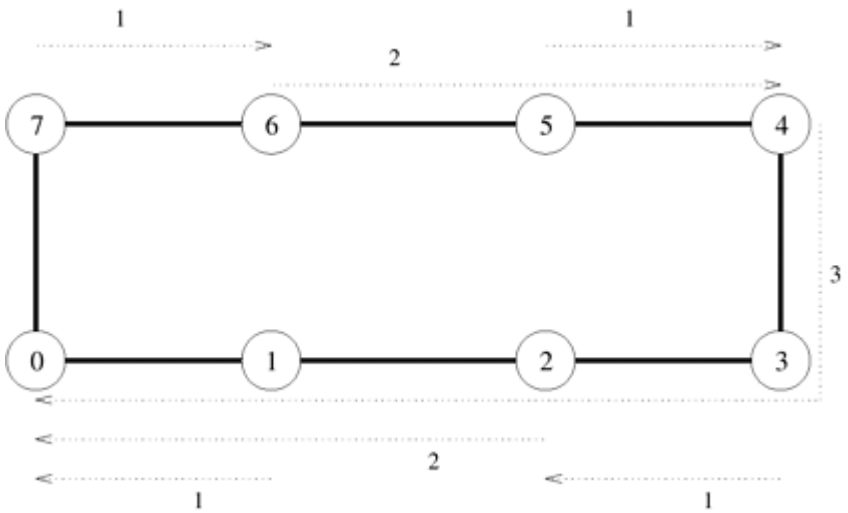
which the message is transferred.



Note that on a linear array, the destination node to which the message is sent in each step must be carefully chosen. In Figure 4.2 , the message is first sent to the farthest node (4) from the source (0). In the second step, the distance between the sending and receiving nodes is halved, and so on. The message recipients are selected in this manner at each step to avoid congestion on the network. For example, if node 0 sent the message to node 1 in the first step and then nodes 0 and 1 attempted to send messages to nodes 2 and 3, respectively, in the second step, the link between nodes 1 and 2 would be congested as it would be a part of the shortest route for both the messages in the second step.

Reduction on a linear array can be performed by simply reversing the direction and the sequence of communication, as shown in Figure 4.3 . In the first step, each odd numbered node sends its buffer to the even numbered node just before itself, where the contents of the two buffers are combined into one. After the first step, there are four buffers left to be reduced on nodes 0, 2, 4, and 6, respectively. In the second step, the contents of the buffers on nodes 0 and 2 are accumulated on node 0 and those on nodes 6 and 4 are accumulated on node 4. Finally, node 4 sends its buffer to node 0, which computes the final result of the reduction.

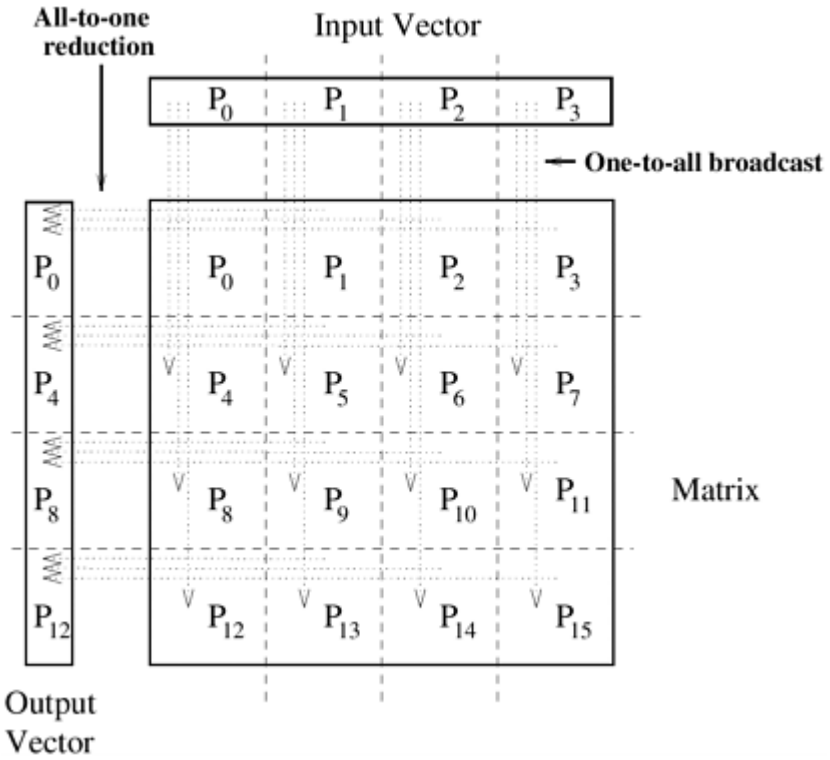
Figure 4.3. Reduction on an eight-node ring with node 0 as the destination of the reduction.



Example 4.1 Matrix-vector multiplication

Consider the problem of multiplying an n x n matrix A with an n x 1 vector x on an n x n mesh of nodes to yield an n x 1 result vector y . Algorithm 8.1 shows a serial algorithm for this problem. Figure 4.4 shows one possible mapping of the matrix and the vectors in which each element of the matrix belongs to a different process, and the vector is distributed among the processes in the topmost row of the mesh and the result vector is generated on the leftmost column of processes.

Figure 4.4. One-to-all broadcast and all-to-one reduction in the multiplication of a 4 x 4 matrix with a 4 x 1 vector.



Since all the rows of the matrix must be multiplied with the vector, each process needs the element of the vector residing in the topmost process of its column. Hence, before computing the matrix-vector product, each column of nodes performs a one-to-all broadcast of the vector elements with the topmost process of the column as the source. This is done by treating each column of the n x n mesh as an n -node linear array, and simultaneously applying the linear array broadcast procedure described previously to all columns.

After the broadcast, each process multiplies its matrix element with the result of the broadcast. Now, each row of processes needs to add its result to generate the corresponding element of the product vector. This is accomplished by performing all-to-one reduction on each row of the process mesh with the first process of each row as the destination of the reduction operation.

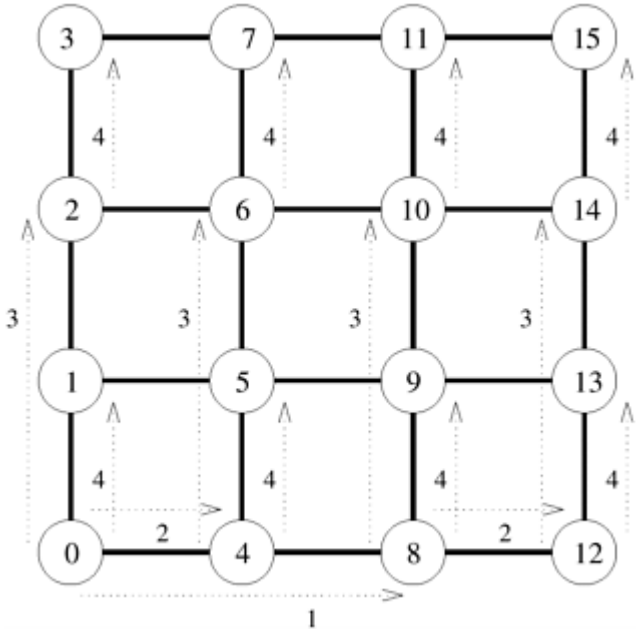
For example, P_9 will receive x [1] from P_1 as a result of the broadcast, will multiply it with A [2, 1] and will participate in an all-to-one reduction with P_8 , P_{10} , and P_{11} to accumulate y [2] on P_8 . ■

4.1.2 Mesh

We can regard each row and column of a square mesh of p nodes as a linear array of $\sqrt{p}$ nodes. So a number of communication algorithms on the mesh are simple extensions of their linear array counterparts. A linear array communication operation can be performed in two phases on a mesh. In the first phase, the operation is performed along one or all rows by treating the rows as linear arrays. In the second phase, the columns are treated similarly.

Consider the problem of one-to-all broadcast on a two-dimensional square mesh with $\sqrt{p}$ rows and $\sqrt{p}$ columns. First, a one-to-all broadcast is performed from the source to the remaining ($\sqrt{p}-1$) nodes of the same row. Once all the nodes in a row of the mesh have acquired the data, they initiate a one-to-all broadcast in their respective columns. At the end of the second phase, every node in the mesh has a copy of the initial message. The communication steps for one-to-all broadcast on a mesh are illustrated in Figure 4.5 for p = 16, with node 0 at the bottom-left corner as the source. Steps 1 and 2 correspond to the first phase, and steps 3 and 4 correspond to the second phase.

Figure 4.5. One-to-all broadcast on a 16-node mesh.



We can use a similar procedure for one-to-all broadcast on a three-dimensional mesh as well. In this case, rows of $p^{1/3}$ nodes in each of the three dimensions of the mesh would be treated as linear arrays. As in the case of a linear array, reduction can be performed on two- and three-dimensional meshes by simply reversing the direction and the order of messages.

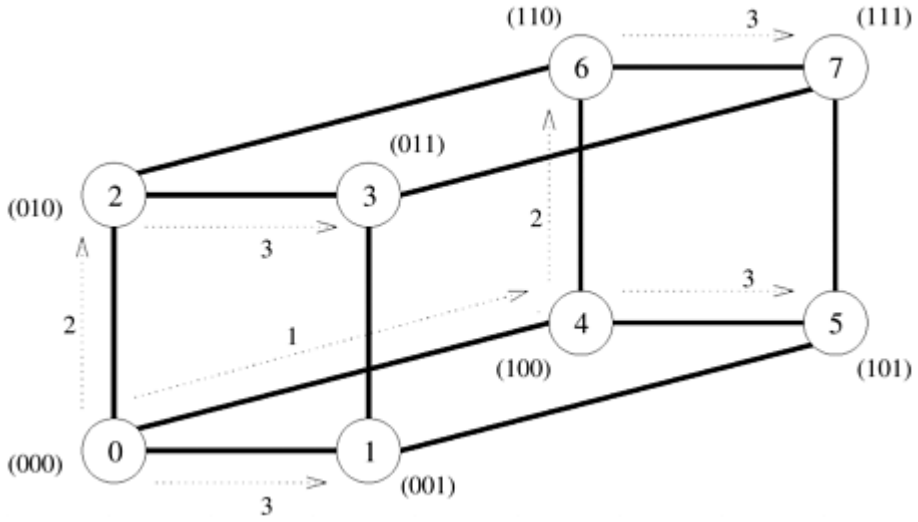
4.1.3 Hypercube

The previous subsection showed that one-to-all broadcast is performed in two phases on a two-dimensional mesh, with the communication taking place along a different dimension in each phase. Similarly, the process is carried out in three phases on a three-dimensional mesh. A hypercube with 2^d nodes can be regarded as a d -dimensional mesh with two nodes in each

dimension. Hence, the mesh algorithm can be extended to the hypercube, except that the process is now carried out in *d* steps – one in each dimension.

Figure 4.6 shows a one-to-all broadcast on an eight-node (three-dimensional) hypercube with node 0 as the source. In this figure, communication starts along the highest dimension (that is, the dimension specified by the most significant bit of the binary representation of a node label) and proceeds along successively lower dimensions in subsequent steps. Note that the source and the destination nodes in three communication steps of the algorithm shown in Figure 4.6 are identical to the ones in the broadcast algorithm on a linear array shown in Figure 4.2 . However, on a hypercube, the order in which the dimensions are chosen for communication does not affect the outcome of the procedure. Figure 4.6 shows only one such order. Unlike a linear array, the hypercube broadcast would not suffer from congestion if node 0 started out by sending the message to node 1 in the first step, followed by nodes 0 and 1 sending messages to nodes 2 and 3, respectively, and finally nodes 0, 1, 2, and 3 sending messages to nodes 4, 5, 6, and 7, respectively.

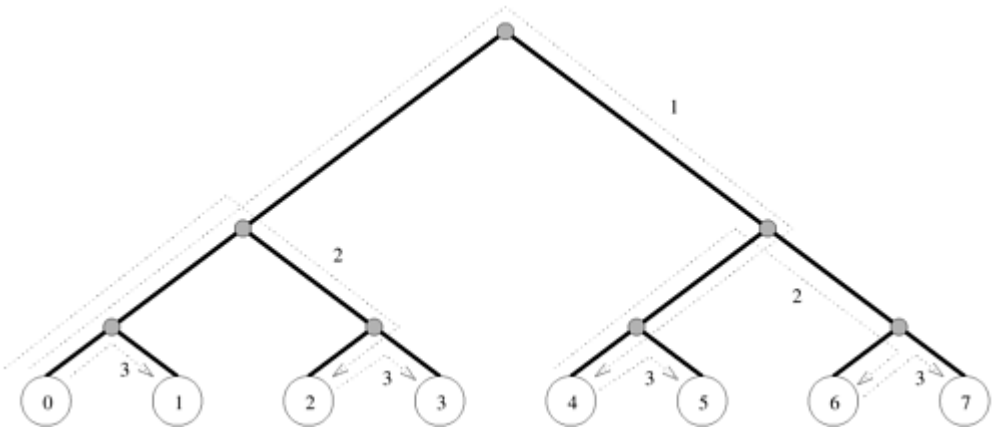
Figure 4.6. One-to-all broadcast on a three-dimensional hypercube. The binary representations of node labels are shown in parentheses.



4.1.4 Balanced Binary Tree

The hypercube algorithm for one-to-all broadcast maps naturally onto a balanced binary tree in which each leaf is a processing node and intermediate nodes serve only as switching units. This is illustrated in Figure 4.7 for eight nodes. In this figure, the communicating nodes have the same labels as in the hypercube algorithm illustrated in Figure 4.6 . Figure 4.7 shows that there is no congestion on any of the communication links at any time. The difference between the communication on a hypercube and the tree shown in Figure 4.7 is that there is a different number of switching nodes along different paths on the tree.

Figure 4.7. One-to-all broadcast on an eight-node tree.



4.1.5 Detailed Algorithms

A careful look at Figures 4.2 , 4.5 , 4.6 , and 4.7 would reveal that the basic communication pattern for one-to-all broadcast is identical on all the four interconnection networks considered in this section. We now describe procedures to implement the broadcast and reduction operations. For the sake of simplicity, the algorithms are described here in the context of a hypercube and assume that the number of communicating processes is a power of 2. However, they apply to any network topology, and can be easily extended to work for any number of processes (Problem 4.1).

Algorithm 4.1 shows a one-to-all broadcast procedure on a 2^d -node network when node 0 is the source of the broadcast. The procedure is executed at all the nodes. At any node, the value of *my_id* is the label of that node. Let *X* be the message to be broadcast, which initially resides at the source node 0. The procedure performs *d* communication steps, one along each dimension of a hypothetical hypercube. In Algorithm 4.1 , communication proceeds from the highest to the lowest dimension (although the order in which dimensions are chosen does not matter). The loop counter *i* indicates the current dimension of the hypercube in which communication is taking place. Only the nodes with zero in the *i* least significant bits of their labels participate in communication along dimension *i* . For instance, on the three-dimensional hypercube shown in Figure 4.6 , *i* is equal to 2 in the first time step. Therefore, only nodes 0 and 4 communicate, since their two least significant bits are zero. In the next time step, when *i* = 1, all nodes (that is, 0, 2, 4, and 6) with zero in their least significant bits participate in communication. The procedure terminates after communication has taken place along all dimensions.

The variable *mask* helps determine which nodes communicate in a particular iteration of the loop. The variable *mask* has *d* (= log *p*) bits, all of which are initially set to one (Line 3). At the beginning of each iteration, the most significant nonzero bit of *mask* is reset to zero (Line 5). Line 6 determines which nodes communicate in the current iteration of the outer loop. For instance, for the hypercube of Figure 4.6 , *mask* is initially set to 111, and it would be 011 during the iteration corresponding to *i* = 2 (the *i* least significant bits of *mask* are ones). The AND operation on Line 6 selects only those nodes that have zeros in their *i* least significant bits.

Among the nodes selected for communication along dimension *i* , the nodes with a zero at bit position *i* send the data, and the nodes with a one at bit position *i* receive it. The test to determine the sending and receiving nodes is performed on Line 7. For example, in Figure 4.6 , node 0 (000) is the sender and node 4 (100) is the receiver in the iteration corresponding to *i* = 2. Similarly, for *i* = 1, nodes 0 (000) and 4 (100) are senders while nodes 2 (010) and 6 (110) are receivers.

Algorithm 4.1 works only if node 0 is the source of the broadcast. For an arbitrary source, we

must relabel the nodes of the hypothetical hypercube by XORing the label of each node with the label of the source node before we apply this procedure. A modified one-to-all broadcast procedure that works for any value of *source* between 0 and p - 1 is shown in Algorithm 4.2 . By performing the XOR operation at Line 3, Algorithm 4.2 relabels the source node to 0, and relabels the other nodes relative to the source. After this relabeling, the algorithm of Algorithm 4.1 can be applied to perform the broadcast.

Algorithm 4.3 gives a procedure to perform an all-to-one reduction on a hypothetical d -dimensional hypercube such that the final result is accumulated on node 0. Single node-accumulation is the dual of one-to-all broadcast. Therefore, we obtain the communication pattern required to implement reduction by reversing the order and the direction of messages in one-to-all broadcast. Procedure ALL_TO_ONE_REDUCE(d , *my_id* , m , X , *sum*) shown in Algorithm 4.3 is very similar to procedure ONE_TO_ALL_BC(d , *my_id* , X) shown in Algorithm 4.1 . One difference is that the communication in all-to-one reduction proceeds from the lowest to the highest dimension. This change is reflected in the way that variables *mask* and i are manipulated in Algorithm 4.3 . The criterion for determining the source and the destination among a pair of communicating nodes is also reversed (Line 7). Apart from these differences, procedure ALL_TO_ONE_REDUCE has extra instructions (Lines 13 and 14) to add the contents of the messages received by a node in each iteration (any associative operation can be used in place of addition).

Algorithm 4.1 One-to-all broadcast of a message X from node 0 of a d -dimensional p -node hypercube (d = log p). AND and XOR are bitwise logical-and and exclusive-or operations, respectively.

```
1.    procedure ONE_TO_ALL_BC(d, my_id, X)
2.    begin
3.       mask := 2^d - 1;                  /* Set all d bits of mask to 1 */
4.       for i := d - 1 downto 0 do        /* Outer loop */
5.           mask := mask XOR 2^i;         /* Set bit i of mask to 0 */
6.           if (my_id AND mask) = 0 then /* If lower i bits of my_id are 0 */
7.               if (my_id AND 2^i) = 0 then
8.                   msg_destination := my_id XOR 2^i;
9.                   send X to msg_destination;
10.              else
11.                  msg_source := my_id XOR 2^i;
12.                  receive X from msg_source;
13.              endelse;
14.          endif;
15.      endfor;
16.   end ONE_TO_ALL_BC
```

Algorithm 4.2 One-to-all broadcast of a message X initiated by *source* on a d -dimensional hypothetical hypercube. The AND and XOR operations are bitwise logical operations.

```
1.   procedure GENERAL_ONE_TO_ALL_BC(d, my_id, source, X)
2.   begin
3.      my_virtual_id := my_id XOR source;
4.      mask := 2^d - 1;
5.      for i := d - 1 downto 0 do   /* Outer loop */
6.          mask := mask XOR 2^i;    /* Set bit i of mask to 0 */
```

```
7.          if (my_virtual_id AND mask) = 0 then
8.              if (my_virtual_id AND 2^i) = 0 then
9.                  virtual_dest := my_virtual_id XOR 2^i;
10.                 send X to (virtual_dest XOR source);
        /* Convert virtual_dest to the label of the physical destination */
11.             else
12.                 virtual_source := my_virtual_id XOR 2^i;
13.                 receive X from (virtual_source XOR source);
        /* Convert virtual_source to the label of the physical source */
14.             endelse;
15.     endfor;
16.  end GENERAL_ONE_TO_ALL_BC
```

Algorithm 4.3 Single-node accumulation on a d -dimensional hypercube. Each node contributes a message X containing m words, and node 0 is the destination of the sum. The AND and XOR operations are bitwise logical operations.

```
1.   procedure ALL_TO_ONE_REDUCE(d, my_id, m, X, sum)
2.   begin
3.      for j := 0 to m - 1 do sum[j] := X[j];
4.      mask := 0;
5.      for i := 0 to d - 1 do
            /* Select nodes whose lower i bits are 0 */
6.          if (my_id AND mask) = 0 then
7.              if (my_id AND 2^i) ≠ 0 then
8.                  msg_destination := my_id XOR 2^i;
9.                  send sum to msg_destination;
10.             else
11.                 msg_source := my_id XOR 2^i;
12.                 receive X from msg_source;
13.                 for j := 0 to m - 1 do
14.                     sum[j] :=sum[j] + X[j];
15.             endelse;
16.         mask := mask XOR 2^i; /* Set bit i of mask to 1 */
17.     endfor;
18.  end ALL_TO_ONE_REDUCE
```

4.1.6 Cost Analysis

Analyzing the cost of one-to-all broadcast and all-to-one reduction is fairly straightforward. Assume that p processes participate in the operation and the data to be broadcast or reduced contains m words. The broadcast or reduction procedure involves log p point-to-point simple message transfers, each at a time cost of $t_s + t_w m$. Therefore, the total time taken by the procedure is

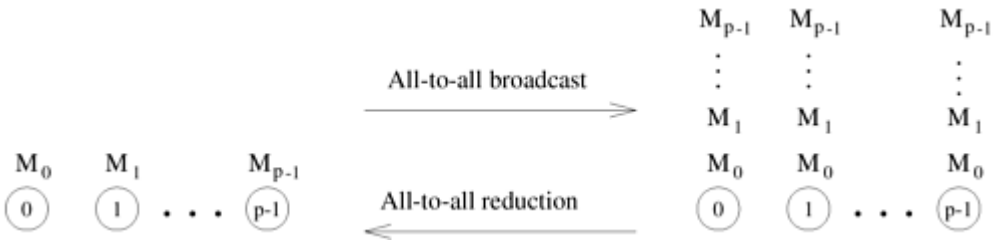
Equation 4.1

$$T = (t_s + t_w m) \log p.$$

4.2 All-to-All Broadcast and Reduction

All-to-all broadcast is a generalization of one-to-all broadcast in which all *p* nodes simultaneously initiate a broadcast. A process sends the same *m*-word message to every other process, but different processes may broadcast different messages. All-to-all broadcast is used in matrix operations, including matrix multiplication and matrix-vector multiplication. The dual of all-to-all broadcast is *all-to-all reduction*, in which every node is the destination of an all-to-one reduction (Problem 4.8). Figure 4.8 illustrates all-to-all broadcast and all-to-all reduction.

Figure 4.8. All-to-all broadcast and all-to-all reduction.



One way to perform an all-to-all broadcast is to perform *p* one-to-all broadcasts, one starting at each node. If performed naively, on some architectures this approach may take up to *p* times as long as a one-to-all broadcast. It is possible to use the communication links in the interconnection network more efficiently by performing all *p* one-to-all broadcasts simultaneously so that all messages traversing the same path at the same time are concatenated into a single message whose size is the sum of the sizes of individual messages.

The following sections describe all-to-all broadcast on linear array, mesh, and hypercube topologies.

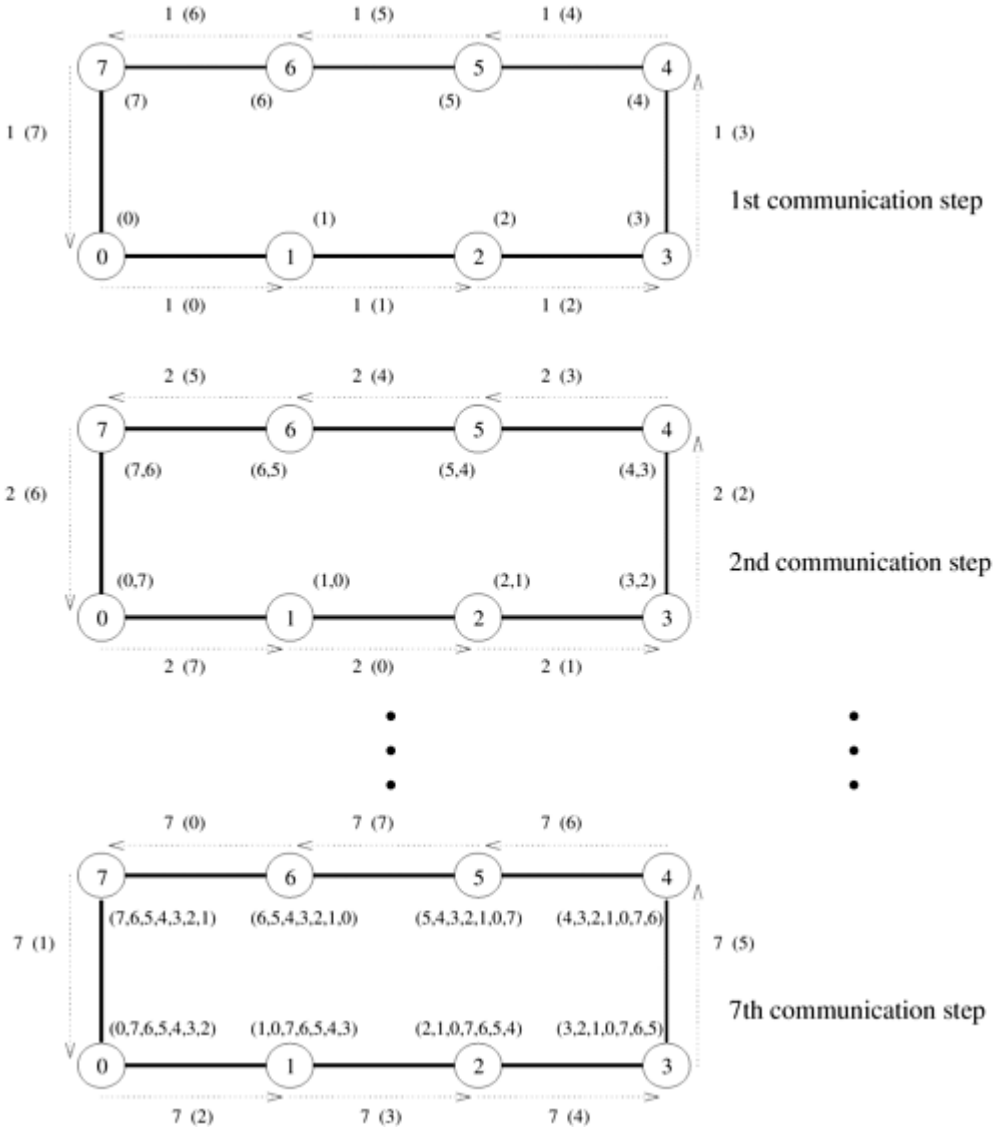
4.2.1 Linear Array and Ring

While performing all-to-all broadcast on a linear array or a ring, all communication links can be kept busy simultaneously until the operation is complete because each node always has some information that it can pass along to its neighbor. Each node first sends to one of its neighbors the data it needs to broadcast. In subsequent steps, it forwards the data received from one of its neighbors to its other neighbor.

Figure 4.9 illustrates all-to-all broadcast for an eight-node ring. The same procedure would also work on a linear array with bidirectional links. As with the previous figures, the integer label of an arrow indicates the time step during which the message is sent. In all-to-all broadcast, *p* different messages circulate in the *p*-node ensemble. In Figure 4.9, each message is identified by its initial source, whose label appears in parentheses along with the time step. For instance, the arc labeled 2 (7) between nodes 0 and 1 represents the data communicated in time step 2 that node 0 received from node 7 in the preceding step. As Figure 4.9 shows, if communication is performed circularly in a single direction, then each node receives all (*p* - 1) pieces of information from all other nodes in (*p* - 1) steps.

Figure 4.9. All-to-all broadcast on an eight-node ring. The label of each

arrow shows the time step and, within parentheses, the label of the node that owned the current message being transferred before the beginning of the broadcast. The number(s) in parentheses next to each node are the labels of nodes from which data has been received prior to the current communication step. Only the first, second, and last communication steps are shown.



Algorithm 4.4 gives a procedure for all-to-all broadcast on a p-node ring. The initial message to be broadcast is known locally as *my_msg* at each node. At the end of the procedure, each node stores the collection of all p messages in *result*. As the program shows, all-to-all broadcast on a mesh applies the linear array procedure twice, once along the rows and once along the columns.

Algorithm 4.4 All-to-all broadcast on a p-node ring.

```
1.    procedure ALL_TO_ALL_BC_RING(my_id, my_msg, p, result)
2.    begin
```

```
3.      left := (my_id - 1) mod p;
4.      right := (my_id + 1) mod p;
5.      result := my_msg;
6.      msg := result;
7.      for i := 1 to p - 1 do
8.          send msg to right;
9.          receive msg from left;
10.         result := result ∪ msg;
11.     endfor;
12.  end ALL_TO_ALL_BC_RING
```

In all-to-all reduction, the dual of all-to-all broadcast, each node starts with p messages, each one destined to be accumulated at a distinct node. All-to-all reduction can be performed by reversing the direction and sequence of the messages. For example, the first communication step for all-to-all reduction on an 8-node ring would correspond to the last step of Figure 4.9 with node 0 sending $msg[1]$ to 7 instead of receiving it. The only additional step required is that upon receiving a message, a node must combine it with the local copy of the message that has the same destination as the received message before forwarding the combined message to the next neighbor. Algorithm 4.5 gives a procedure for all-to-all reduction on a p-node ring.

Algorithm 4.5 All-to-all reduction on a p-node ring.

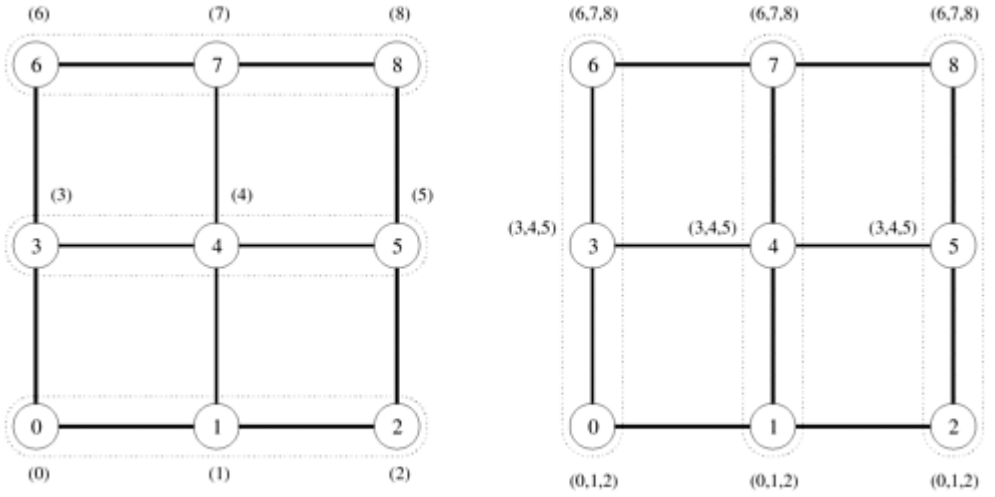
```
1.   procedure ALL_TO_ALL_RED_RING(my_id, my_msg, p, result)
2.   begin
3.      left := (my_id - 1) mod p;
4.      right := (my_id + 1) mod p;
5.      recv := 0;
6.      for i := 1 to p - 1 do
7.          j := (my_id + i) mod p;
8.          temp := msg[j] + recv;
9.          send temp to left;
10.         receive recv from right;
11.     endfor;
12.     result := msg[my_id] + recv;
13.  end ALL_TO_ALL_RED_RING
```

4.2.2 Mesh

Just like one-to-all broadcast, the all-to-all broadcast algorithm for the 2-D mesh is based on the linear array algorithm, treating rows and columns of the mesh as linear arrays. Once again, communication takes place in two phases. In the first phase, each row of the mesh performs an all-to-all broadcast using the procedure for the linear array. In this phase, all nodes collect $\sqrt{p}$ messages corresponding to the $\sqrt{p}$ nodes of their respective rows. Each node consolidates this information into a single message of size $m\sqrt{p}$, and proceeds to the second communication phase of the algorithm. The second communication phase is a columnwise all-to-all broadcast of the consolidated messages. By the end of this phase, each node obtains all p pieces of m-word data that originally resided on different nodes. The distribution of data among the nodes of a 3 x 3 mesh at the beginning of the first and the second phases of the algorithm is shown in Figure 4.10.

Figure 4.10. All-to-all broadcast on a 3 x 3 mesh. The groups of nodes

communicating with each other in each phase are enclosed by dotted boundaries. By the end of the second phase, all nodes get (0,1,2,3,4,5,6,7) (that is, a message from each node).



(a) Initial data distribution (b) Data distribution after rowwise broadcast

Algorithm 4.6 gives a procedure for all-to-all broadcast on a $\sqrt{p} \times \sqrt{p}$ mesh. The mesh procedure for all-to-all reduction is left as an exercise for the reader (Problem 4.4).

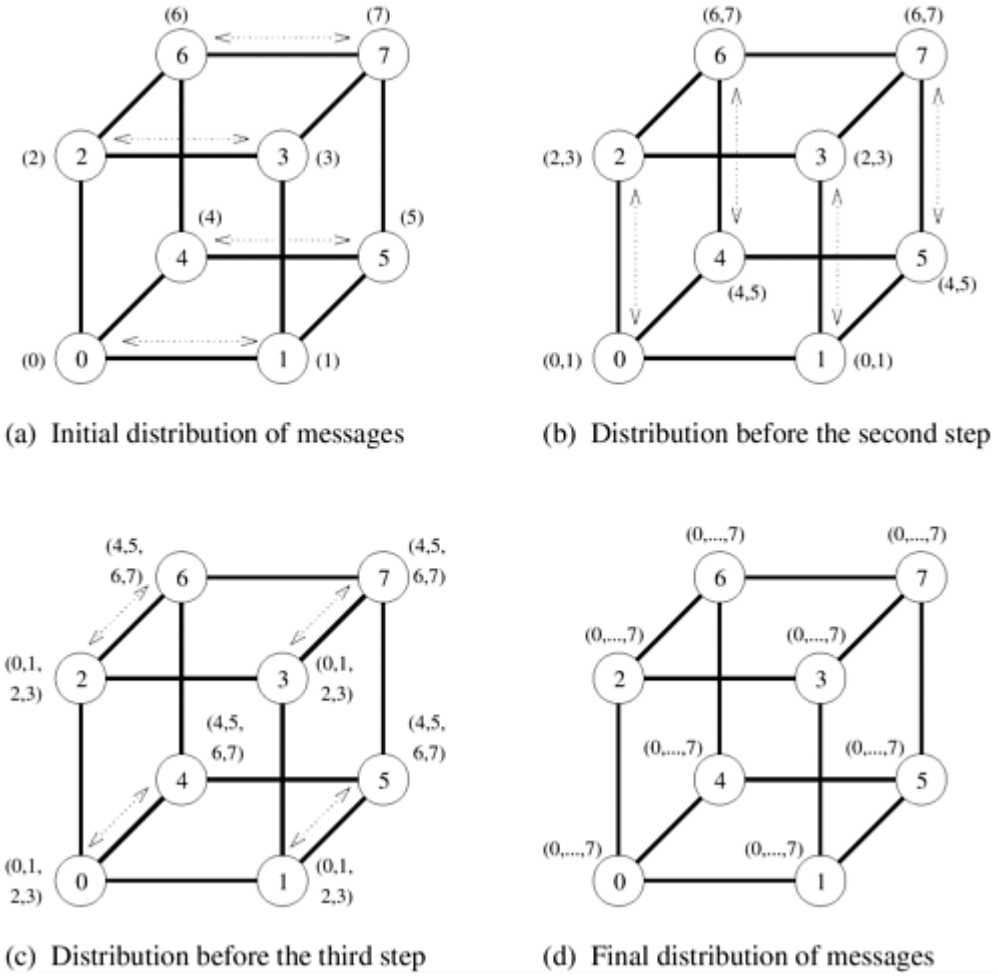
Algorithm 4.6 All-to-all broadcast on a square mesh of *p* nodes.

```
1.   procedure ALL_TO_ALL_BC_MESH(my_id, my_msg, p, result)
2.   begin

/* Communication along rows */
3.       left := my_id - (my_id mod √p) + (my_id - 1)mod √p;
4.       right := my_id - (my_id mod √p) + (my_id + 1) mod √p;
5.       result := my_msg;
6.       msg := result;
7.       for i := 1 to √p - 1 do
8.           send msg to right;
9.           receive msg from left;
10.          result := result ∪ msg;
11.      endfor;

/* Communication along columns */
12.      up := (my_id - √p) mod p;
13.      down := (my_id + √p) mod p;
14.      msg := result;
15.      for i := 1 to √p - 1 do
16.          send msg to down;
17.          receive msg from up;
18.          result := result ∪ msg;
19.      endfor;
20.  end ALL_TO_ALL_BC_MESH
```

4.2.3 Hypercube

The hypercube algorithm for all-to-all broadcast is an extension of the mesh algorithm to log p dimensions. The procedure requires log p steps. Communication takes place along a different dimension of the p-node hypercube in each step. In every step, pairs of nodes exchange their data and double the size of the message to be transmitted in the next step by concatenating the received message with their current data. Figure 4.11 shows these steps for an eight-node hypercube with bidirectional communication channels.

Figure 4.11. All-to-all broadcast on an eight-node hypercube.



Algorithm 4.7 gives a procedure for implementing all-to-all broadcast on a d-dimensional hypercube. Communication starts from the lowest dimension of the hypercube and then proceeds along successively higher dimensions (Line 4). In each iteration, nodes communicate in pairs so that the labels of the nodes communicating with each other in the i th iteration differ in the i th least significant bit of their binary representations (Line 5). After an iteration's communication steps, each node concatenates the data it receives during that iteration with its resident data (Line 8). This concatenated message is transmitted in the following iteration.

Algorithm 4.7 All-to-all broadcast on a d-dimensional hypercube.

```
1.   procedure ALL_TO_ALL_BC_HCUBE(my_id, my_msg, d, result)
2.   begin
3.      result := my_msg;
4.      for i := 0 to d - 1 do
5.          partner := my id XOR 2^i;
6.          send result to partner;
7.          receive msg from partner;
8.          result := result ∪ msg;
9.      endfor;
10.  end ALL_TO_ALL_BC_HCUBE
```

As usual, the algorithm for all-to-all reduction can be derived by reversing the order and direction of messages in all-to-all broadcast. Furthermore, instead of concatenating the messages, the reduction operation needs to select the appropriate subsets of the buffer to send out and accumulate received messages in each iteration. Algorithm 4.8 gives a procedure for all-to-all reduction on a d-dimensional hypercube. It uses *senloc* to index into the starting location of the outgoing message and *recloc* to index into the location where the incoming message is added in each iteration.

Algorithm 4.8 All-to-all broadcast on a d-dimensional hypercube. AND and XOR are bitwise logical-and and exclusive-or operations, respectively.

```
1.   procedure ALL_TO_ALL_RED_HCUBE(my_id, msg, d, result)
2.   begin
3.      recloc := 0;
4.      for i := d - 1 to 0 do
5.          partner := my_id XOR 2^i;
6.          j := my_id AND 2^i;
7.          k := (my_id XOR 2^i) AND 2^i;
8.          senloc := recloc + k;
9.          recloc := recloc + j;
10.         send msg[senloc .. senloc + 2^i - 1] to partner;
11.         receive temp[0 .. 2^i - 1] from partner;
12.         for j := 0 to 2^i - 1 do
13.             msg[recloc + j] := msg[recloc + j] + temp[j];
14.         endfor;
15.     endfor;
16.     result := msg[my_id];
17.  end ALL_TO_ALL_RED_HCUBE
```

4.2.4 Cost Analysis

On a ring or a linear array, all-to-all broadcast involves p - 1 steps of communication between nearest neighbors. Each step, involving a message of size m, takes time $t_s + t_w m$. Therefore, the time taken by the entire operation is

Equation 4.2

$$T = (t_s + t_w m)(p - 1).$$

Similarly, on a mesh, the first phase of $\sqrt{p}$ simultaneous all-to-all broadcasts (each among $\sqrt{p}$ nodes) concludes in time $(t_s + t_w m)(\sqrt{p} - 1)$. The number of nodes participating in each all-to-all broadcast in the second phase is also $\sqrt{p}$, but the size of each message is now $m\sqrt{p}$. Therefore, this phase takes time $(t_s + t_w m\sqrt{p})(\sqrt{p} - 1)$ to complete. The time for the entire all-to-all broadcast on a p-node two-dimensional square mesh is the sum of the times spent in the individual phases, which is

Equation 4.3

$$T = 2t_s(\sqrt{p} - 1) + t_w m(p - 1).$$

On a p-node hypercube, the size of each message exchanged in the i th of the log p steps is $2^{i-1}m$. It takes a pair of nodes time $t_s + 2^{i-1}t_w m$ to send and receive messages from each other during the i th step. Hence, the time to complete the entire procedure is

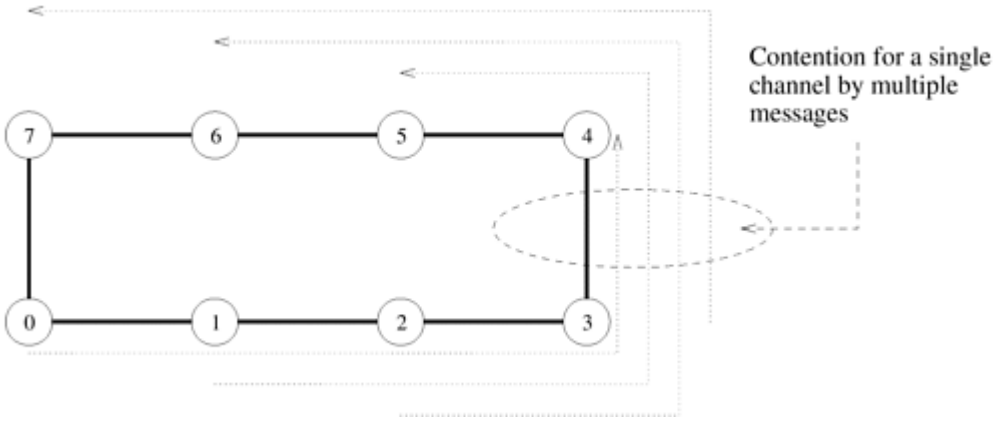
Equation 4.4

$$\begin{aligned} T &= \sum_{i=1}^{\log p} (t_s + 2^{i-1} t_w m) \\ &= t_s \log p + t_w m(p - 1). \end{aligned}$$

Equations 4.2, 4.3, and 4.4 show that the term associated with t_w in the expressions for the communication time of all-to-all broadcast is $t_w m(p - 1)$ for all the architectures. This term also serves as a lower bound for the communication time of all-to-all broadcast for parallel computers on which a node can communicate on only one of its ports at a time. This is because each node receives at least $m(p - 1)$ words of data, regardless of the architecture. Thus, for large messages, a highly connected network like a hypercube is no better than a simple ring in performing all-to-all broadcast or all-to-all reduction. In fact, the straightforward all-to-all broadcast algorithm for a simple architecture like a ring has great practical importance. A close look at the algorithm reveals that it is a sequence of p one-to-all broadcasts, each with a different source. These broadcasts are pipelined so that all of them are complete in a total of p nearest-neighbor communication steps. Many parallel algorithms involve a series of one-to-all broadcasts with different sources, often interspersed with some computation. If each one-to-all broadcast is performed using the hypercube algorithm of Section 4.1.3, then n broadcasts would require time $n(t_s + t_w m) \log p$. On the other hand, by pipelining the broadcasts as shown in Figure 4.9, all of them can be performed spending no more than time $(t_s + t_w m)(p - 1)$ in communication, provided that the sources of all broadcasts are different and $n \leq p$. In later chapters, we show how such pipelined broadcast improves the performance of some parallel algorithms such as Gaussian elimination (Section 8.3.1), back substitution (Section 8.3.3), and Floyd's algorithm for finding the shortest paths in a graph (Section 10.4.2).

Another noteworthy property of all-to-all broadcast is that, unlike one-to-all broadcast, the hypercube algorithm cannot be applied unaltered to mesh and ring architectures. The reason is that the hypercube procedure for all-to-all broadcast would cause congestion on the communication channels of a smaller-dimensional network with the same number of nodes. For instance, Figure 4.12 shows the result of performing the third step (Figure 4.11(c)) of the hypercube all-to-all broadcast procedure on a ring. One of the links of the ring is traversed by all four messages and would take four times as much time to complete the communication step.

Figure 4.12. Contention for a channel when the communication step of Figure 4.11(c) for the hypercube is mapped onto a ring.



PREVIOUS NEXT

4.3 All-Reduce and Prefix-Sum Operations

The communication pattern of all-to-all broadcast can be used to perform some other operations as well. One of these operations is a third variation of reduction, in which each node starts with a buffer of size *m* and the final results of the operation are identical buffers of size *m* on each node that are formed by combining the original *p* buffers using an associative operator. Semantically, this operation, often referred to as the *all-reduce* operation, is identical to performing an all-to-one reduction followed by a one-to-all broadcast of the result. This operation is different from all-to-all reduction, in which *p* simultaneous all-to-one reductions take place, each with a different destination for the result.

An all-reduce operation with a single-word message on each node is often used to implement barrier synchronization on a message-passing computer. The semantics of the reduction operation are such that, while executing a parallel program, no node can finish the reduction before each node has contributed a value.

A simple method to perform all-reduce is to perform an all-to-one reduction followed by a one-to-all broadcast. However, there is a faster way to perform all-reduce by using the communication pattern of all-to-all broadcast. Figure 4.11 illustrates this algorithm for an eight-node hypercube. Assume that each integer in parentheses in the figure, instead of denoting a message, denotes a number to be added that originally resided at the node with that integer label. To perform reduction, we follow the communication steps of the all-to-all broadcast procedure, but at the end of each step, add two numbers instead of concatenating two messages. At the termination of the reduction procedure, each node holds the sum (0 + 1 + 2 + ⋯ + 7) (rather than eight messages numbered from 0 to 7, as in the case of all-to-all broadcast). Unlike all-to-all broadcast, each message transferred in the reduction operation has only one word. The size of the messages does not double in each step because the numbers are added instead of being concatenated. Therefore, the total communication time for all log *p* steps is

Equation 4.5

$$T = (t_s + t_w m) \log p.$$

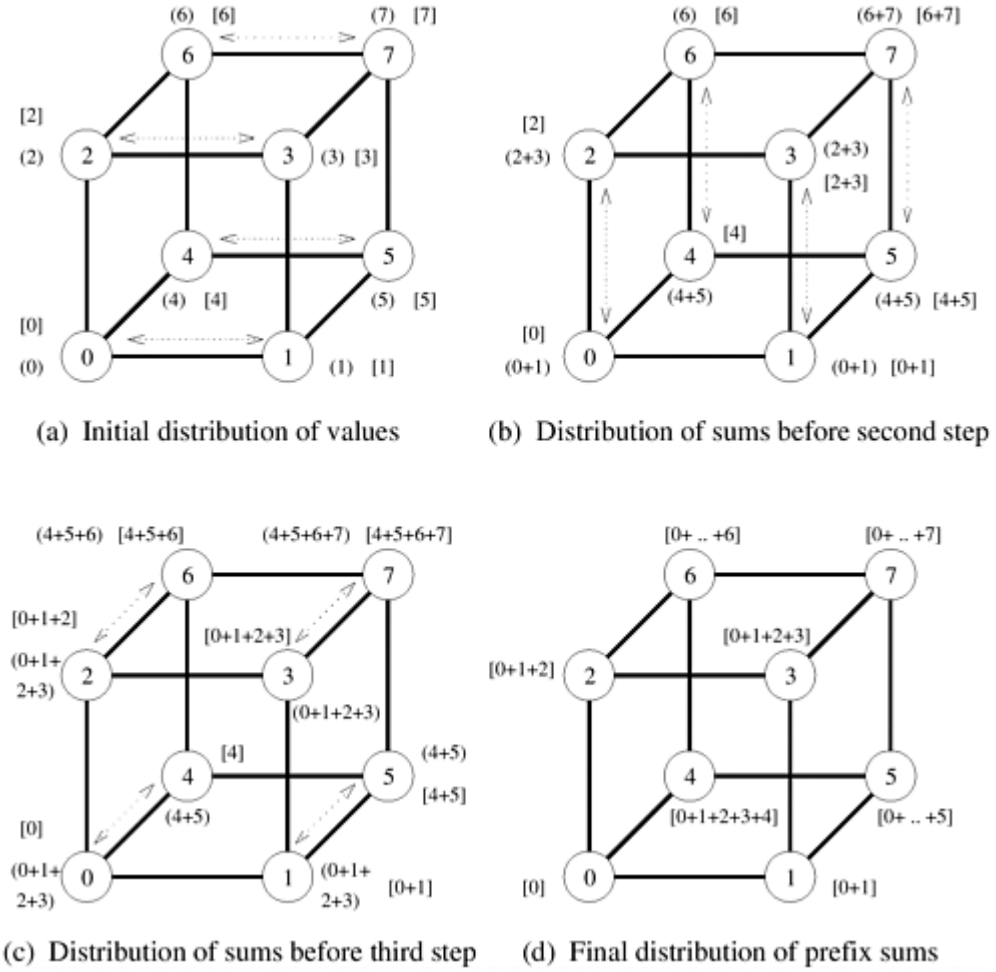
Algorithm 4.7 can be used to perform a sum of *p* numbers if *my_msg*, *msg*, and *result* are numbers (rather than messages), and the union operation ('$\cup$') on Line 8 is replaced by addition.

Finding *prefix sums* (also known as the *scan* operation) is another important problem that can be solved by using a communication pattern similar to that used in all-to-all broadcast and all-reduce operations. Given *p* numbers $n_0, n_1, \ldots, n_{p-1}$ (one on each node), the problem is to compute the sums $s_k = \Sigma_{i=0}^{k} n_i$ for all *k* between 0 and *p* - 1. For example, if the original sequence of numbers is <3, 1, 4, 0, 2>, then the sequence of prefix sums is <3, 4, 8, 8, 10>. Initially, n_k resides on the node labeled *k*, and at the end of the procedure, the same node holds s_k . Instead of starting with a single numbers, each node could start with a buffer or vector of size *m* and the *m*-word result would be the sum of the corresponding elements of buffers.

Figure 4.13 illustrates the prefix sums procedure for an eight-node hypercube. This figure is a

modification of Figure 4.11. The modification is required to accommodate the fact that in prefix sums the node with label k uses information from only the k-node subset of those nodes whose labels are less than or equal to k. To accumulate the correct prefix sum, every node maintains an additional result buffer. This buffer is denoted by square brackets in Figure 4.13. At the end of a communication step, the content of an incoming message is added to the result buffer only if the message comes from a node with a smaller label than that of the recipient node. The contents of the outgoing message (denoted by parentheses in the figure) are updated with every incoming message, just as in the case of the all-reduce operation. For instance, after the first communication step, nodes 0, 2, and 4 do not add the data received from nodes 1, 3, and 5 to their result buffers. However, the contents of the outgoing messages for the next step are updated.

Figure 4.13. Computing prefix sums on an eight-node hypercube. At each node, square brackets show the local prefix sum accumulated in the result buffer and parentheses enclose the contents of the outgoing message buffer for the next step.



Since not all of the messages received by a node contribute to its final result, some of the messages it receives may be redundant. We have omitted these steps of the standard all-to-all broadcast communication pattern from Figure 4.13, although the presence or absence of these messages does not affect the results of the algorithm. Algorithm 4.9 gives a procedure to solve the prefix sums problem on a d-dimensional hypercube.

Algorithm 4.9 Prefix sums on a d-dimensional hypercube.

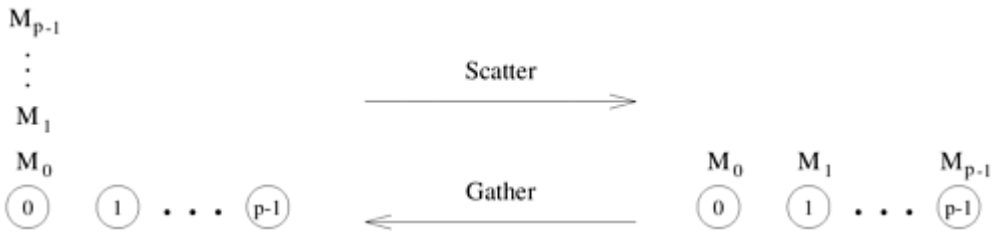
```
1.   procedure PREFIX_SUMS_HCUBE(my_id, my number, d, result)
2.   begin
3.      result := my_number;
4.      msg := result;
5.      for i := 0 to d - 1 do
6.          partner := my_id XOR 2^i;
7.          send msg to partner;
8.          receive number from partner;
9.          msg := msg + number;
10.         if (partner < my_id) then result := result + number;
11.     endfor;
12.  end PREFIX_SUMS_HCUBE
```


PREVIOUS NEXT

4.4 Scatter and Gather

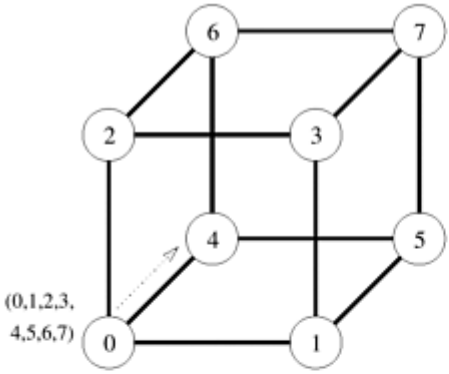
In the *scatter* operation, a single node sends a unique message of size *m* to every other node. This operation is also known as *one-to-all personalized communication*. One-to-all personalized communication is different from one-to-all broadcast in that the source node starts with *p* unique messages, one destined for each node. Unlike one-to-all broadcast, one-to-all personalized communication does not involve any duplication of data. The dual of one-to-all personalized communication or the scatter operation is the *gather* operation, or *concatenation*, in which a single node collects a unique message from each node. A gather operation is different from an all-to-one reduce operation in that it does not involve any combination or reduction of data. Figure 4.14 illustrates the scatter and gather operations.

Figure 4.14. Scatter and gather operations.

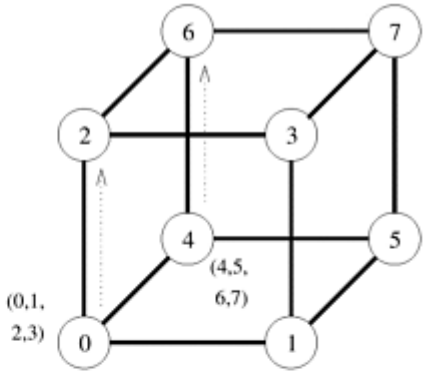


Although the scatter operation is semantically different from one-to-all broadcast, the scatter algorithm is quite similar to that of the broadcast. Figure 4.15 shows the communication steps for the scatter operation on an eight-node hypercube. The communication patterns of one-to-all broadcast (Figure 4.6) and scatter (Figure 4.15) are identical. Only the size and the contents of messages are different. In Figure 4.15, the source node (node 0) contains all the messages. The messages are identified by the labels of their destination nodes. In the first communication step, the source transfers half of the messages to one of its neighbors. In subsequent steps, each node that has some data transfers half of it to a neighbor that has yet to receive any data. There is a total of log *p* communication steps corresponding to the log *p* dimensions of the hypercube.

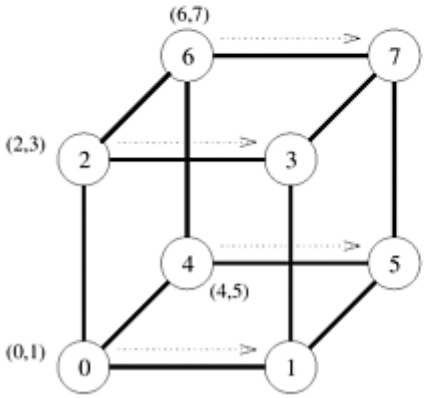
Figure 4.15. The scatter operation on an eight-node hypercube.



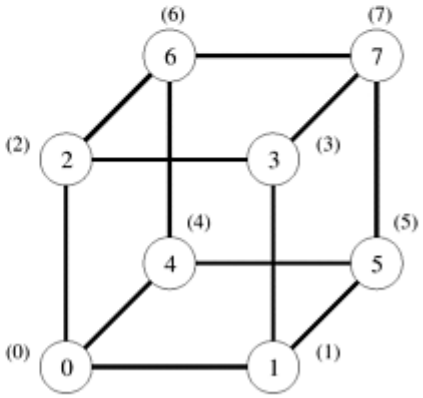
(a) Initial distribution of messages



(b) Distribution before the second step



(c) Distribution before the third step



(d) Final distribution of messages

The gather operation is simply the reverse of scatter. Each node starts with an m word message. In the first step, every odd numbered node sends its buffer to an even numbered neighbor behind it, which concatenates the received message with its own buffer. Only the even numbered nodes participate in the next communication step which results in nodes with multiples of four labels gathering more data and doubling the sizes of their data. The process continues similarly, until node 0 has gathered the entire data.

Just like one-to-all broadcast and all-to-one reduction, the hypercube algorithms for scatter and gather can be applied unaltered to linear array and mesh interconnection topologies without any increase in the communication time.

Cost Analysis All links of a p-node hypercube along a certain dimension join two $p/2$-node subcubes (Section 2.4.3). As Figure 4.15 illustrates, in each communication step of the scatter operations, data flow from one subcube to another. The data that a node owns before starting communication in a certain dimension are such that half of them need to be sent to a node in the other subcube. In every step, a communicating node keeps half of its data, meant for the nodes in its subcube, and sends the other half to its neighbor in the other subcube. The time in which all data are distributed to their respective destinations is

Equation 4.6

$$T = t_s \log p + t_w m(p-1).$$

The scatter and gather operations can also be performed on a linear array and on a 2-D square

mesh in time $t_s \log p + t_w m(p - 1)$ (Problem 4.7). Note that disregarding the term due to message-startup time, the cost of scatter and gather operations for large messages on any k-d mesh interconnection network (Section 2.4.3) is similar. In the scatter operation, at least $m(p - 1)$ words of data must be transmitted out of the source node, and in the gather operation, at least $m(p - 1)$ words of data must be received by the destination node. Therefore, as in the case of all-to-all broadcast, $t_w m(p - 1)$ is a lower bound on the communication time of scatter and gather operations. This lower bound is independent of the interconnection network.

PREVIOUS NEXT

4.5 All-to-All Personalized Communication

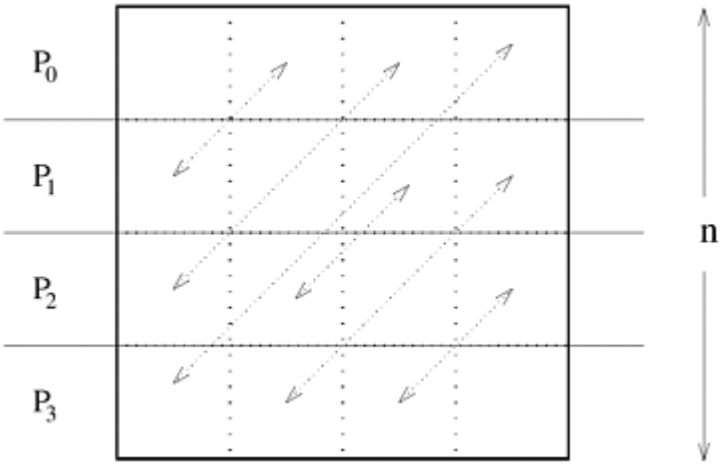
In *all-to-all personalized communication*, each node sends a distinct message of size *m* to every other node. Each node sends different messages to different nodes, unlike all-to-all broadcast, in which each node sends the same message to all other nodes. Figure 4.16 illustrates the all-to-all personalized communication operation. A careful observation of this figure would reveal that this operation is equivalent to transposing a two-dimensional array of data distributed among *p* processes using one-dimensional array partitioning (Figure 3.24). All-to-all personalized communication is also known as *total exchange*. This operation is used in a variety of parallel algorithms such as fast Fourier transform, matrix transpose, sample sort, and some parallel database join operations.

Figure 4.16. All-to-all personalized communication.

Example 4.2 Matrix transposition

The transpose of an *n* x *n* matrix *A* is a matrix A^T of the same size, such that $A^T[i, j] = A[j, i]$ for $0 \leq i, j < n$. Consider an *n* x *n* matrix mapped onto *n* processors such that each processor contains one full row of the matrix. With this mapping, processor P_i initially contains the elements of the matrix with indices [*i*, 0], [*i*, 1], ..., [*i*, *n* - 1]. After the transposition, element [*i*, 0] belongs to P_0, element [*i*, 1] belongs to P_1, and so on. In general, element [*i*, *j*] initially resides on P_i , but moves to P_j during the transposition. The data-communication pattern of this procedure is shown in Figure 4.17 for a 4 x 4 matrix mapped onto four processes using one-dimensional rowwise partitioning. Note that in this figure every processor sends a distinct element of the matrix to every other processor. This is an example of all-to-all personalized communication.

Figure 4.17. All-to-all personalized communication in transposing a 4 x 4 matrix using four processes.



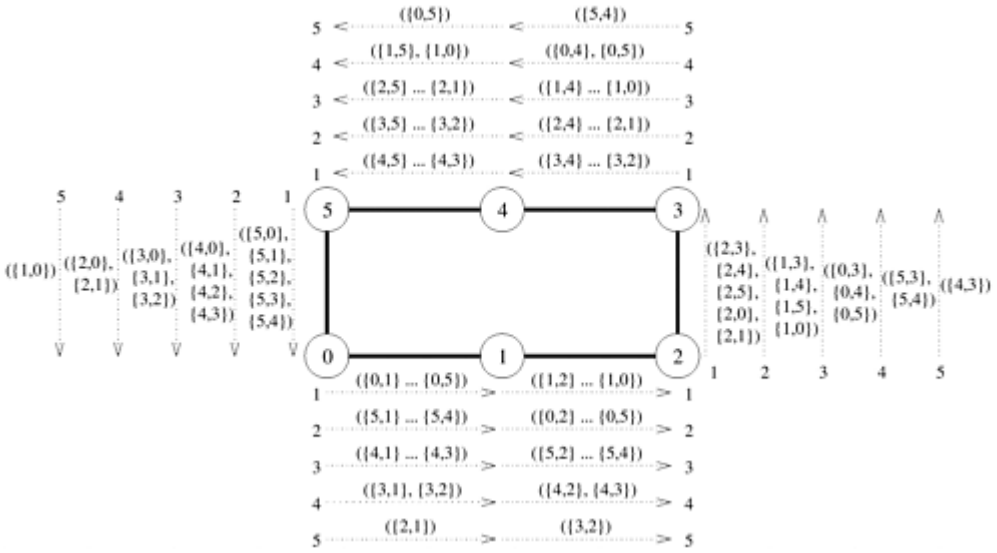
In general, if we use p processes such that $p \leq n$, then each process initially holds n/p rows (that is, n^2/p elements) of the matrix. Performing the transposition now involves an all-to-all personalized communication of matrix blocks of size n/p x n/p, instead of individual elements. ■

We now discuss the implementation of all-to-all personalized communication on parallel computers with linear array, mesh, and hypercube interconnection networks. The communication patterns of all-to-all personalized communication are identical to those of all-to-all broadcast on all three architectures. Only the size and the contents of messages are different.

4.5.1 Ring

Figure 4.18 shows the steps in an all-to-all personalized communication on a six-node linear array. To perform this operation, every node sends p - 1 pieces of data, each of size m. In the figure, these pieces of data are identified by pairs of integers of the form $\{i, j\}$, where i is the source of the message and j is its final destination. First, each node sends all pieces of data as one consolidated message of size $m(p - 1)$ to one of its neighbors (all nodes communicate in the same direction). Of the $m(p - 1)$ words of data received by a node in this step, one m-word packet belongs to it. Therefore, each node extracts the information meant for it from the data received, and forwards the remaining $(p - 2)$ pieces of size m each to the next node. This process continues for p - 1 steps. The total size of data being transferred between nodes decreases by m words in each successive step. In every step, each node adds to its collection one m-word packet originating from a different node. Hence, in p - 1 steps, every node receives the information from all other nodes in the ensemble.

Figure 4.18. All-to-all personalized communication on a six-node ring. The label of each message is of the form $\{x, y\}$, where x is the label of the node that originally owned the message, and y is the label of the node that is the final destination of the message. The label $(\{x_1, y_1\}, \{x_2, y_2\}, \ldots, \{x_n, y_n\})$ indicates a message that is formed by concatenating n individual messages.



In the above procedure, all messages are sent in the same direction. If half of the messages are sent in one direction and the remaining half are sent in the other direction, then the communication cost due to the t_w can be reduced by a factor of two. For the sake of simplicity, we ignore this constant-factor improvement.

Cost Analysis On a ring or a bidirectional linear array, all-to-all personalized communication involves p - 1 communication steps. Since the size of the messages transferred in the i th step is $m(p - i)$, the total time taken by this operation is

Equation 4.7

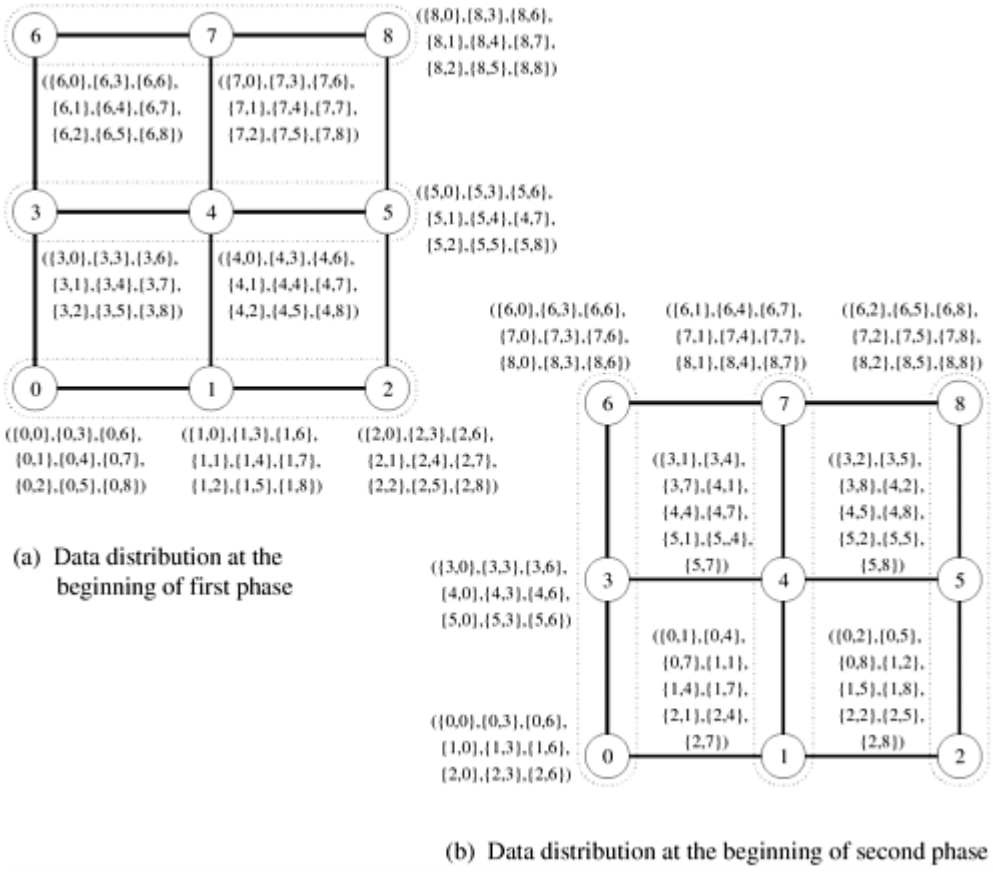
$$\begin{aligned} T &= \sum_{i=1}^{p-1}(t_s + t_w m(p-i)) \\ &= t_s(p-1) + \sum_{i=1}^{p-1} i t_w m \\ &= (t_s + t_w mp/2)(p-1). \end{aligned}$$

In the all-to-all personalized communication procedure described above, each node sends $m(p - 1)$ words of data because it has an m-word packet for every other node. Assume that all messages are sent either clockwise or counterclockwise. The average distance that an m-word packet travels is $(\Sigma_{i=1}^{p-1} i)/(p-1)$, which is equal to $p/2$. Since there are p nodes, each performing the same type of communication, the total traffic (the total number of data words transferred between directly-connected nodes) on the network is $m(p - 1) \times p/2 \times p$. The total number of inter-node links in the network to share this load is p. Hence, the communication time for this operation is at least $(t_w \times m(p - 1)p^2/2)/p$, which is equal to $t_w m(p - 1)p/2$. Disregarding the message startup time t_s, this is exactly the time taken by the linear array procedure. Therefore, the all-to-all personalized communication algorithm described in this section is optimal.

4.5.2 Mesh

In all-to-all personalized communication on a $\sqrt{p} \times \sqrt{p}$ mesh, each node first groups its p messages according to the columns of their destination nodes. Figure 4.19 shows a 3 x 3 mesh, in which every node initially has nine m-word messages, one meant for each node. Each node assembles its data into three groups of three messages each (in general, $\sqrt{p}$ groups of $\sqrt{p}$ messages each). The first group contains the messages destined for nodes labeled 0, 3, and 6; the second group contains the messages for nodes labeled 1, 4, and 7; and the last group has messages for nodes labeled 2, 5, and 8.

Figure 4.19. The distribution of messages at the beginning of each phase of all-to-all personalized communication on a 3 x 3 mesh. At the end of the second phase, node i has messages ($\{0, i\}$, ..., $\{8, i\}$), where $0 \leq i \leq 8$. The groups of nodes communicating together in each phase are enclosed in dotted boundaries.



After the messages are grouped, all-to-all personalized communication is performed independently in each row with clustered messages of size $m\sqrt{p}$. One cluster contains the information for all $\sqrt{p}$ nodes of a particular column. Figure 4.19(b) shows the distribution of data among the nodes at the end of this phase of communication.

Before the second communication phase, the messages in each node are sorted again, this time according to the rows of their destination nodes; then communication similar to the first phase takes place in all the columns of the mesh. By the end of this phase, each node receives a message from every other node.

Cost Analysis We can compute the time spent in the first phase by substituting $\sqrt{p}$ for the

number of nodes, and $m\sqrt{p}$ for the message size in Equation 4.7. The result of this substitution is $(t_s + t_w mp/2)(\sqrt{p} - 1)$. The time spent in the second phase is the same as that in the first phase. Therefore, the total time for all-to-all personalized communication of messages of size m on a p-node two-dimensional square mesh is

Equation 4.8

$$T = (2t_s + t_w mp)(\sqrt{p} - 1).$$

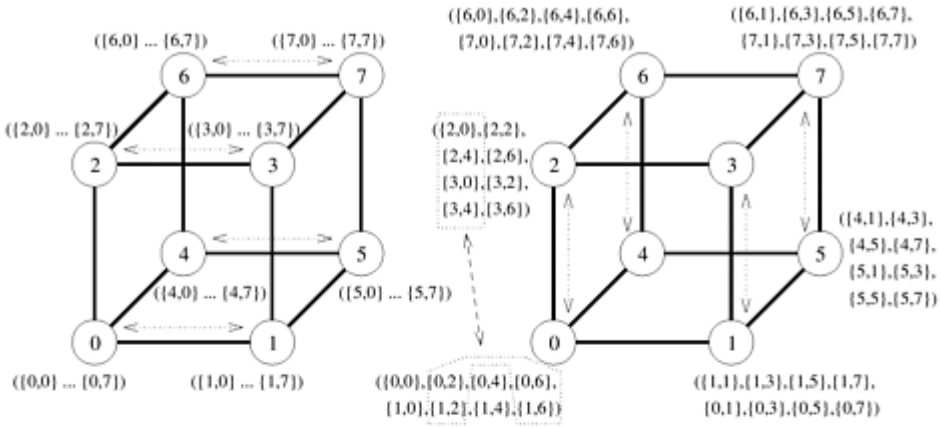
The expression for the communication time of all-to-all personalized communication in Equation 4.8 does not take into account the time required for the local rearrangement of data (that is, sorting the messages by rows or columns). Assuming that initially the data is ready for the first communication phase, the second communication phase requires the rearrangement of mp words of data. If t_r is the time to perform a read and a write operation on a single word of data in a node's local memory, then the total time spent in data rearrangement by a node during the entire procedure is $t_r mp$ (Problem 4.21). This time is much smaller than the time spent by each node in communication.

An analysis along the lines of that for the linear array would show that the communication time given by Equation 4.8 for all-to-all personalized communication on a square mesh is optimal within a small constant factor (Problem 4.11).

4.5.3 Hypercube

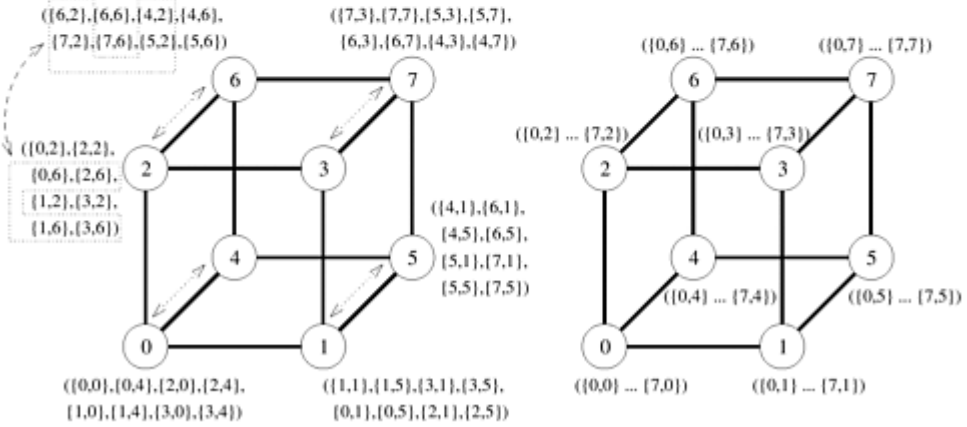
One way of performing all-to-all personalized communication on a p-node hypercube is to simply extend the two-dimensional mesh algorithm to log p dimensions. Figure 4.20 shows the communication steps required to perform this operation on a three-dimensional hypercube. As shown in the figure, communication takes place in log p steps. Pairs of nodes exchange data in a different dimension in each step. Recall that in a p-node hypercube, a set of $p/2$ links in the same dimension connects two subcubes of $p/2$ nodes each (Section 2.4.3). At any stage in all-to-all personalized communication, every node holds p packets of size m each. While communicating in a particular dimension, every node sends $p/2$ of these packets (consolidated as one message). The destinations of these packets are the nodes of the other subcube connected by the links in current dimension.

Figure 4.20. An all-to-all personalized communication algorithm on a three-dimensional hypercube.



(a) Initial distribution of messages

(b) Distribution before the second step



(c) Distribution before the third step

(d) Final distribution of messages

In the preceding procedure, a node must rearrange its messages locally before each of the log p communication steps. This is necessary to make sure that all $p/2$ messages destined for the same node in a communication step occupy contiguous memory locations so that they can be transmitted as a single consolidated message.

Cost Analysis In the above hypercube algorithm for all-to-all personalized communication, $mp/2$ words of data are exchanged along the bidirectional channels in each of the log p iterations. The resulting total communication time is

Equation 4.9

$$T = (t_s + t_w mp/2) \log p.$$

Before each of the log p communication steps, a node rearranges mp words of data. Hence, a total time of $t_r mp \log p$ is spent by each node in local rearrangement of data during the entire procedure. Here t_r is the time needed to perform a read and a write operation on a single word of data in a node's local memory. For most practical computers, t_r is much smaller than t_w; hence, the time to perform an all-to-all personalized communication is dominated by the communication time.

Interestingly, unlike the linear array and mesh algorithms described in this section, the

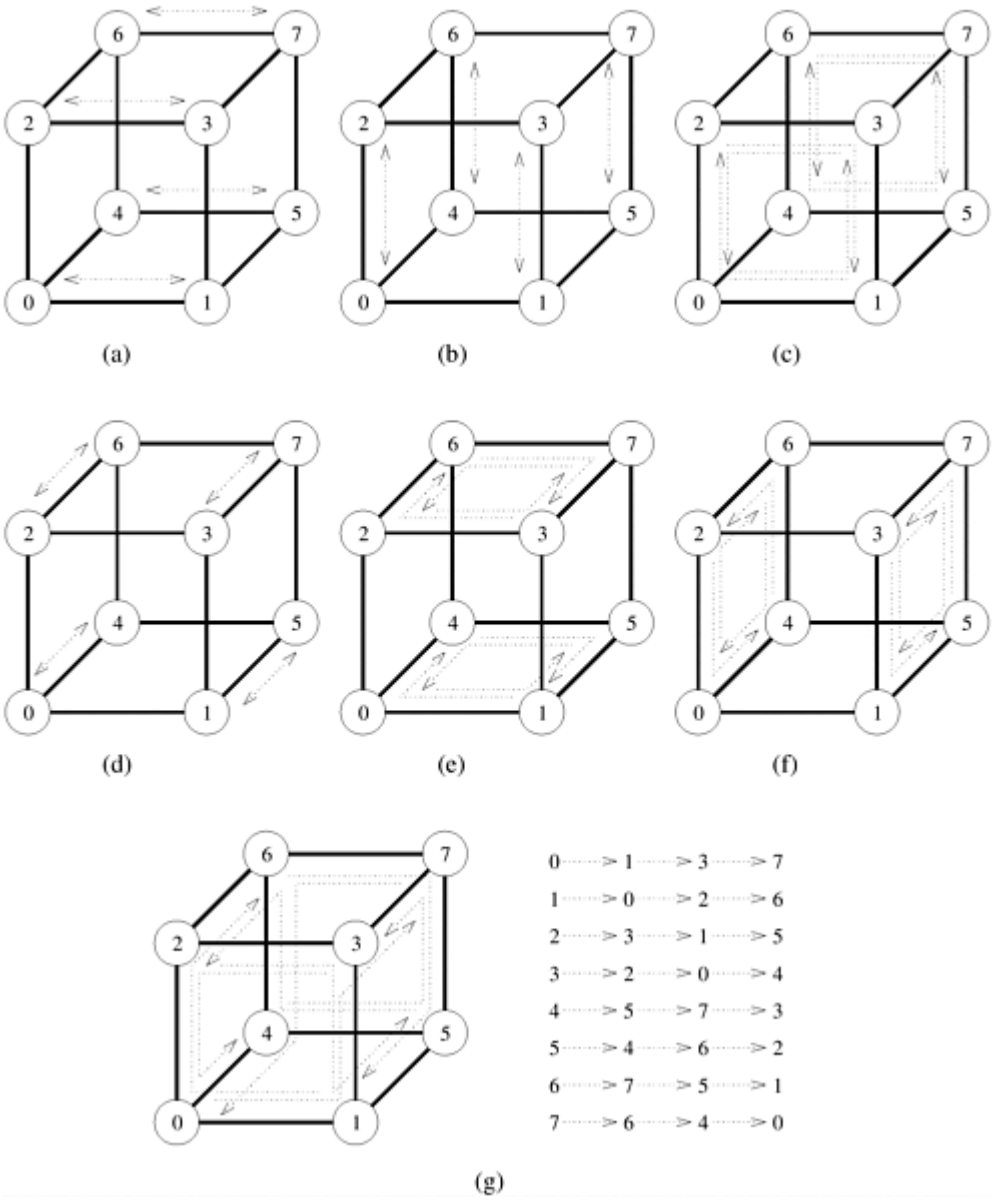
hypercube algorithm is not optimal. Each of the p nodes sends and receives $m(p - 1)$ words of data and the average distance between any two nodes on a hypercube is $(\log p)/2$. Therefore, the total data traffic on the network is $p \times m(p - 1) \times (\log p)/2$. Since there is a total of $(p \log p)/2$ links in the hypercube network, the lower bound on the all-to-all personalized communication time is

$$\begin{aligned} T &= \frac{t_w pm(p-1)(\log p)/2}{(p \log p)/2} \\ &= t_w m(p-1). \end{aligned}$$

An Optimal Algorithm

An all-to-all personalized communication effectively results in all pairs of nodes exchanging some data. On a hypercube, the best way to perform this exchange is to have every pair of nodes communicate directly with each other. Thus, each node simply performs $p - 1$ communication steps, exchanging m words of data with a different node in every step. A node must choose its communication partner in each step so that the hypercube links do not suffer congestion. Figure 4.21 shows one such congestion-free schedule for pairwise exchange of data in a three-dimensional hypercube. As the figure shows, in the j th communication step, node i exchanges data with node (i XOR j). For example, in part (a) of the figure (step 1), the labels of communicating partners differ in the least significant bit. In part (g) (step 7), the labels of communicating partners differ in all the bits, as the binary representation of seven is 111. In this figure, all the paths in every communication step are congestion-free, and none of the bidirectional links carry more than one message in the same direction. This is true in general for a hypercube of any dimension. If the messages are routed appropriately, a congestion-free schedule exists for the $p - 1$ communication steps of all-to-all personalized communication on a p-node hypercube. Recall from Section 2.4.3 that a message traveling from node i to node j on a hypercube must pass through at least l links, where l is the Hamming distance between i and j (that is, the number of nonzero bits in the binary representation of (i XOR j)). A message traveling from node i to node j traverses links in l dimensions (corresponding to the nonzero bits in the binary representation of (i XOR j)). Although the message can follow one of the several paths of length l that exist between i and j (assuming $l > 1$), a distinct path is obtained by sorting the dimensions along which the message travels in ascending order. According to this strategy, the first link is chosen in the dimension corresponding to the least significant nonzero bit of (i XOR j), and so on. This routing scheme is known as *E-cube routing*.

Figure 4.21. Seven steps in all-to-all personalized communication on an eight-node hypercube.



Algorithm 4.10 for all-to-all personalized communication on a *d*-dimensional hypercube is based on this strategy.

Algorithm 4.10 A procedure to perform all-to-all personalized communication on a *d*-dimensional hypercube. The message $M_{i,j}$ initially resides on node i and is destined for node *j*.

```
1.   procedure ALL_TO_ALL_PERSONAL(d, my_id)
2.   begin
3.      for i := 1 to 2^d - 1 do
4.      begin
5.         partner := my_id XOR i;
6.         send M_{my_id, partner} to partner;
7.         receive M_{partner,my_id} from partner;
8.      endfor;
9.   end ALL_TO_ALL_PERSONAL
```

Cost Analysis E-cube routing ensures that by choosing communication pairs according to Algorithm 4.10, a communication time of $t_s + t_w m$ is guaranteed for a message transfer between node i and node j because there is no contention with any other message traveling in the same direction along the link between nodes i and j. The total communication time for the entire operation is

Equation 4.10

$$T = (t_s + t_w m)(p - 1).$$

A comparison of Equations 4.9 and 4.10 shows the term associated with t_s is higher for the second hypercube algorithm, while the term associated with t_w is higher for the first algorithm. Therefore, for small messages, the startup time may dominate, and the first algorithm may still be useful.

PREVIOUS NEXT

4.6 Circular Shift

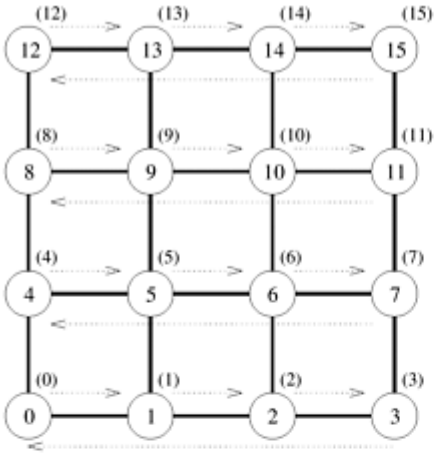
Circular shift is a member of a broader class of global communication operations known as *permutation*. A permutation is a simultaneous, one-to-one data redistribution operation in which each node sends a packet of m words to a unique node. We define a *circular q-shift* as the operation in which node i sends a data packet to node $(i + q) \bmod p$ in a p-node ensemble $(0 < q < p)$. The shift operation finds application in some matrix computations and in string and image pattern matching.

4.6.1 Mesh

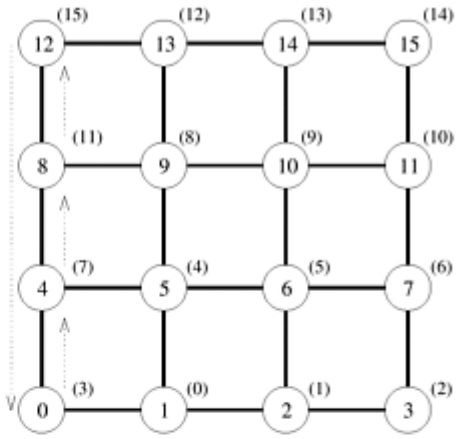
The implementation of a circular q-shift is fairly intuitive on a ring or a bidirectional linear array. It can be performed by $\min\{q , p - q\}$ neighbor-to-neighbor communications in one direction. Mesh algorithms for circular shift can be derived by using the ring algorithm.

If the nodes of the mesh have row-major labels, a circular q-shift can be performed on a p-node square wraparound mesh in two stages. This is illustrated in Figure 4.22 for a circular 5-shift on a 4 x 4 mesh. First, the entire set of data is shifted simultaneously by ($q \bmod \sqrt{p}$) steps along the rows. Then it is shifted by $\lfloor q/\sqrt{p} \rfloor$ steps along the columns. During the circular row shifts, some of the data traverse the wraparound connection from the highest to the lowest labeled nodes of the rows. All such data packets must shift an additional step forward along the columns to compensate for the $\sqrt{p}$ distance that they lost while traversing the backward edge in their respective rows. For example, the 5-shift in Figure 4.22 requires one row shift, a compensatory column shift, and finally one column shift.

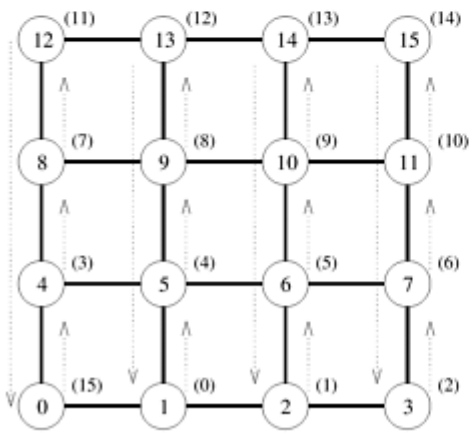
Figure 4.22. The communication steps in a circular 5-shift on a 4 x 4 mesh.



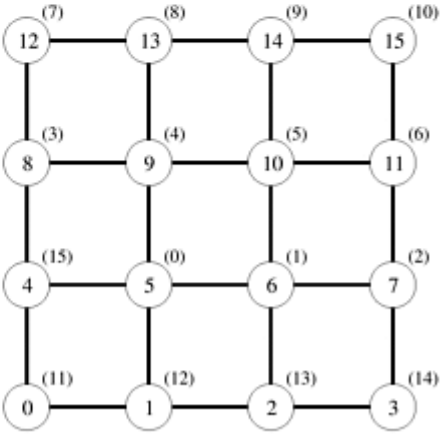
(a) Initial data distribution and the first communication step



(b) Step to compensate for backward row shifts



(c) Column shifts in the third communication step



(d) Final distribution of the data

In practice, we can choose the direction of the shifts in both the rows and the columns to minimize the number of steps in a circular shift. For instance, a 3-shift on a 4 x 4 mesh can be performed by a single backward row shift. Using this strategy, the number of unit shifts in a direction cannot exceed $\lfloor \sqrt{p}/2 \rfloor$.

Cost Analysis Taking into account the compensating column shift for some packets, the total time for any circular q-shift on a p-node mesh using packets of size m has an upper bound of

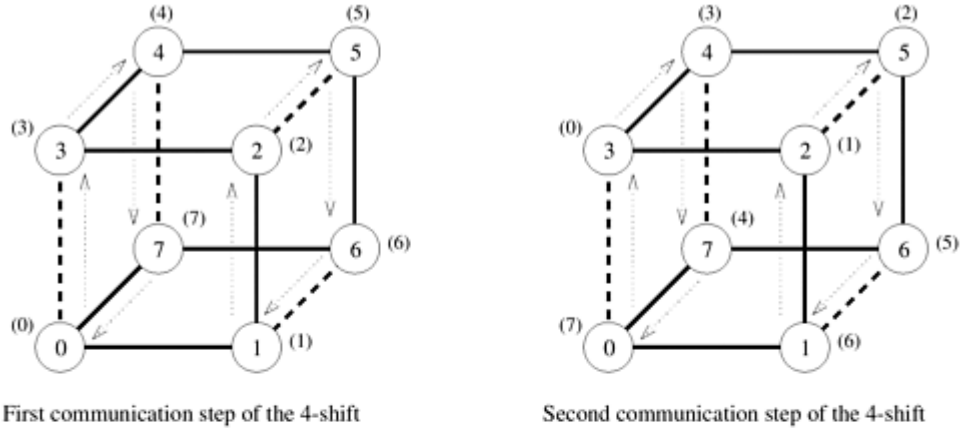
$$T = (t_s + t_w m)(\sqrt{p} + 1)$$

4.6.2 Hypercube

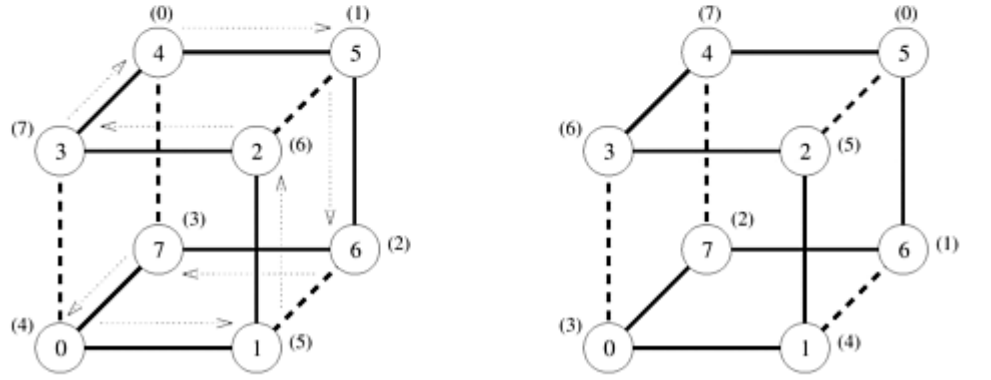
In developing a hypercube algorithm for the shift operation, we map a linear array with 2^d nodes onto a d-dimensional hypercube. We do this by assigning node i of the linear array to node j of the hypercube such that j is the d-bit binary reflected Gray code (RGC) of i. Figure 4.23 illustrates this mapping for eight nodes. A property of this mapping is that any two nodes at a distance of 2^i on the linear array are separated by exactly two links on the hypercube. An

exception is $i = 0$ (that is, directly-connected nodes on the linear array) when only one hypercube link separates the two nodes.

Figure 4.23. The mapping of an eight-node linear array onto a three-dimensional hypercube to perform a circular 5-shift as a combination of a 4-shift and a 1-shift.



(a) The first phase (a 4-shift)



(b) The second phase (a 1-shift) (c) Final data distribution after the 5-shift

To perform a q-shift, we expand q as a sum of distinct powers of 2. The number of terms in the sum is the same as the number of ones in the binary representation of q. For example, the number 5 can be expressed as $2^2 + 2^0$. These two terms correspond to bit positions 0 and 2 in the binary representation of 5, which is 101. If q is the sum of s distinct powers of 2, then the circular q-shift on a hypercube is performed in s phases.

In each phase of communication, all data packets move closer to their respective destinations by short cutting the linear array (mapped onto the hypercube) in leaps of the powers of 2. For example, as Figure 4.23 shows, a 5-shift is performed by a 4-shift followed by a 1-shift. The number of communication phases in a q-shift is exactly equal to the number of ones in the binary representation of q. Each phase consists of two communication steps, except the 1-shift, which, if required (that is, if the least significant bit of q is 1), consists of a single step. For example, in a 5-shift, the first phase of a 4-shift (Figure 4.23(a)) consists of two steps and the second phase of a 1-shift (Figure 4.23(b)) consists of one step. Thus, the total number of steps for any q in a p-node hypercube is at most $2 \log p - 1$.

All communications in a given time step are congestion-free. This is ensured by the property of the linear array mapping that all nodes whose mutual distance on the linear array is a power of 2 are arranged in disjoint subarrays on the hypercube. Thus, all nodes can freely communicate in a circular fashion in their respective subarrays. This is shown in Figure 4.23(a), in which nodes labeled 0, 3, 4, and 7 form one subarray and nodes labeled 1, 2, 5, and 6 form another subarray.

The upper bound on the total communication time for any shift of m-word packets on a p-node hypercube is

Equation 4.11

$$T = (t_s + t_w m)(2 \log p - 1).$$

We can reduce this upper bound to $(t_s + t_w m) \log p$ by performing both forward and backward shifts. For example, on eight nodes, a 6-shift can be performed by a single backward 2-shift instead of a forward 4-shift followed by a forward 2-shift.

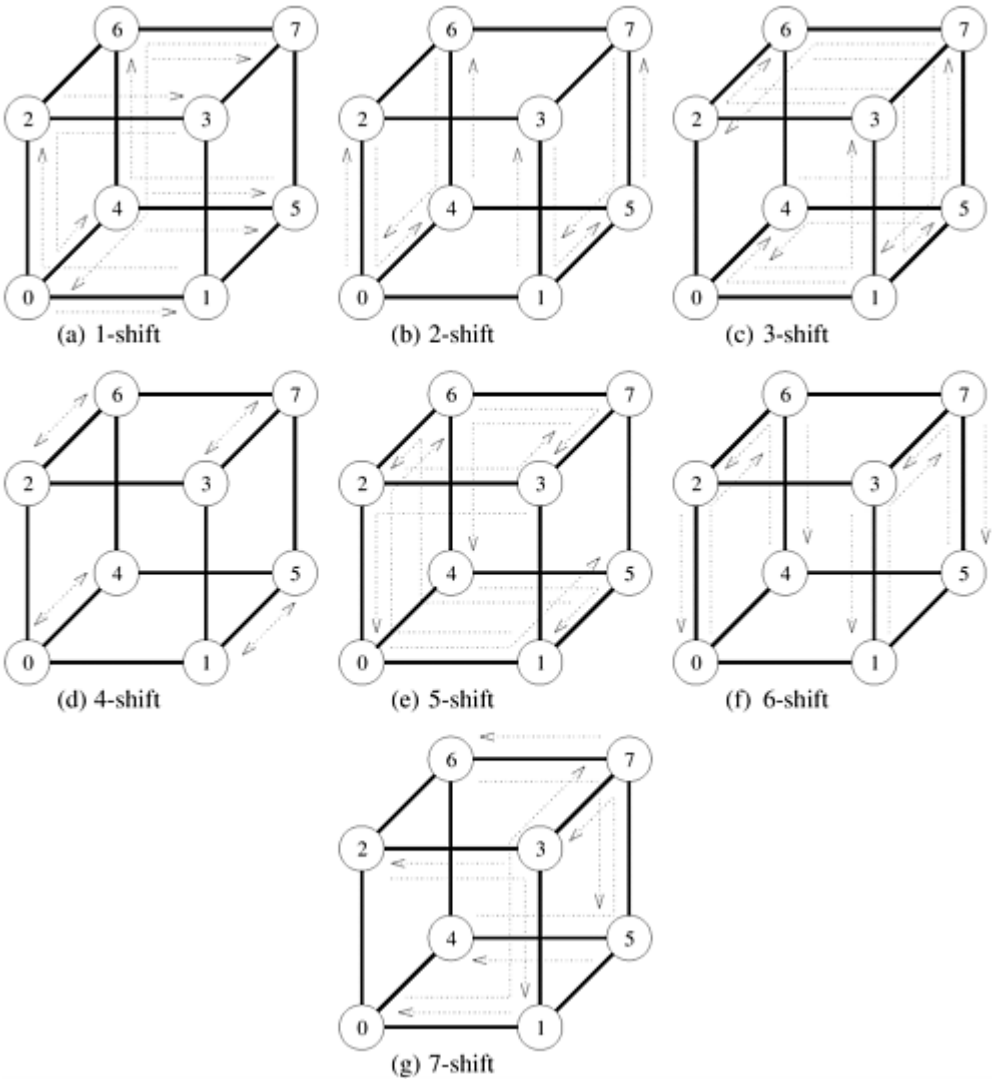
We now show that if the E-cube routing introduced in Section 4.5 is used, then the time for circular shift on a hypercube can be improved by almost a factor of log p for large messages.

This is because with E-cube routing, each pair of nodes with a constant distance l ($i \leq l < p$) has a congestion-free path (Problem 4.22) in a p-node hypercube with bidirectional channels. Figure 4.24 illustrates the non-conflicting paths of all the messages in circular q -shift operations for $1 \leq q < 8$ on an eight-node hypercube. In a circular q-shift on a p-node hypercube, the longest path contains $\log p - \gamma(q)$ links, where $\gamma(q)$ is the highest integer j such that q is divisible by 2^j (Problem 4.23). Thus, the total communication time for messages of length m is

Equation 4.12

$$T = t_s + t_w m.$$

Figure 4.24. Circular q-shifts on an 8-node hypercube for $1 \leq q < 8$.



(a) 1-shift (b) 2-shift (c) 3-shift

(d) 4-shift (e) 5-shift (f) 6-shift

(g) 7-shift

4.7 Improving the Speed of Some Communication Operations

So far in this chapter, we have derived procedures for various communication operations and their communication times under the assumptions that the original messages could not be split into smaller parts and that each node had a single port for sending and receiving data. In this section, we briefly discuss the impact of relaxing these assumptions on some of the communication operations.

4.7.1 Splitting and Routing Messages in Parts

In the procedures described in Sections 4.1–4.6, we assumed that an entire m-word packet of data travels between the source and the destination nodes along the same path. If we split large messages into smaller parts and then route these parts through different paths, we can sometimes utilize the communication network better. We have already shown that, with a few exceptions like one-to-all broadcast, all-to-one reduction, all-reduce, etc., the communication operations discussed in this chapter are asymptotically optimal for large messages; that is, the terms associated with t_w in the costs of these operations cannot be reduced asymptotically. In this section, we present asymptotically optimal algorithms for three global communication operations.

Note that the algorithms of this section rely on m being large enough to be split into p roughly equal parts. Therefore, the earlier algorithms are still useful for shorter messages. A comparison of the cost of the algorithms in this section with those presented earlier in this chapter for the same operations would reveal that the term associated with t_s increases and the term associated with t_w decreases when the messages are split. Therefore, depending on the actual values of t_s, t_w, and p, there is a cut-off value for the message size m and only the messages longer than the cut-off would benefit from the algorithms in this section.

One-to-All Broadcast

Consider broadcasting a single message M of size m from one source node to all the nodes in a p-node ensemble. If m is large enough so that M can be split into p parts $M_0, M_1, \ldots, M_{p-1}$ of size m/p each, then a scatter operation (Section 4.4) can place M_i on node i in time $t_s \log p + t_w(m/p)(p - 1)$. Note that the desired result of the one-to-all broadcast is to place $M = M_0 \cup M_1 \cup \cdots \cup M_{p-1}$ on all nodes. This can be accomplished by an all-to-all broadcast of the messages of size m/p residing on each node after the scatter operation. This all-to-all broadcast can be completed in time $t_s \log p + t_w(m/p)(p - 1)$ on a hypercube. Thus, on a hypercube, one-to-all broadcast can be performed in time

Equation 4.13

$$\begin{aligned} T &= 2 \times \left(t_s \log p + t_w (p-1)\frac{m}{p}\right) \\ &\approx 2 \times (t_s \log p + t_w m). \end{aligned}$$

Compared to Equation 4.1, this algorithm has double the startup cost, but the cost due to the t_w term has been reduced by a factor of (log p)/2. Similarly, one-to-all broadcast can be improved on linear array and mesh interconnection networks as well.

All-to-One Reduction

All-to-one reduction is a dual of one-to-all broadcast. Therefore, an algorithm for all-to-one reduction can be obtained by reversing the direction and the sequence of communication in one-to-all broadcast. We showed above how an optimal one-to-all broadcast algorithm can be obtained by performing a scatter operation followed by an all-to-all broadcast. Therefore, using the notion of duality, we should be able to perform an all-to-one reduction by performing all-to-all reduction (dual of all-to-all broadcast) followed by a gather operation (dual of scatter). We leave the details of such an algorithm as an exercise for the reader (Problem 4.17).

All-Reduce

Since an all-reduce operation is semantically equivalent to an all-to-one reduction followed by a one-to-all broadcast, the asymptotically optimal algorithms for these two operations presented above can be used to construct a similar algorithm for the all-reduce operation. Breaking all-to-one reduction and one-to-all broadcast into their component operations, it can be shown that an all-reduce operation can be accomplished by an all-to-all reduction followed by a gather followed by a scatter followed by an all-to-all broadcast. Since the intermediate gather and scatter would simply nullify each other's effect, all-reduce just requires an all-to-all reduction and an all-to-all broadcast. First, the m-word messages on each of the p nodes are logically split into p components of size roughly m/p words. Then, an all-to-all reduction combines all the i th components on p_i. After this step, each node is left with a distinct m/p-word component of the final result. An all-to-all broadcast can construct the concatenation of these components on each node.

A p-node hypercube interconnection network allows all-to-one reduction and one-to-all broadcast involving messages of size m/p in time $t_s \log p + t_w(m/p)(p - 1)$ each. Therefore, the all-reduce operation can be completed in time

Equation 4.14

$$\begin{aligned} T &= 2 \times (t_s \log p + t_w(p-1)\frac{m}{p}) \\ &\approx 2 \times (t_s \log p + t_w m). \end{aligned}$$

4.7.2 All-Port Communication

In a parallel architecture, a single node may have multiple communication ports with links to other nodes in the ensemble. For example, each node in a two-dimensional wraparound mesh has four ports, and each node in a d-dimensional hypercube has d ports. In this book, we generally assume what is known as the *single-port communication* model. In single-port communication, a node can send data on only one of its ports at a time. Similarly, a node can receive data on only one port at a time. However, a node can send and receive data simultaneously, either on the same port or on separate ports. In contrast to the single-port

model, an *all-port communication* model permits simultaneous communication on all the channels connected to a node.

On a p-node hypercube with all-port communication, the coefficients of t_w in the expressions for the communication times of one-to-all and all-to-all broadcast and personalized communication are all smaller than their single-port counterparts by a factor of $\log p$. Since the number of channels per node for a linear array or a mesh is constant, all-port communication does not provide any asymptotic improvement in communication time on these architectures.

Despite the apparent speedup, the all-port communication model has certain limitations. For instance, not only is it difficult to program, but it requires that the messages are large enough to be split efficiently among different channels. In several parallel algorithms, an increase in the size of messages means a corresponding increase in the granularity of computation at the nodes. When the nodes are working with large data sets, the internode communication time is dominated by the computation time if the computational complexity of the algorithm is higher than the communication complexity. For example, in the case of matrix multiplication, there are n^3 computations for n^2 words of data transferred among the nodes. If the communication time is a small fraction of the total parallel run time, then improving the communication by using sophisticated techniques is not very advantageous in terms of the overall run time of the parallel algorithm.

Another limitation of all-port communication is that it can be effective only if data can be fetched and stored in memory at a rate sufficient to sustain all the parallel communication. For example, to utilize all-port communication effectively on a p-node hypercube, the memory bandwidth must be greater than the communication bandwidth of a single channel by a factor of at least $\log p$; that is, the memory bandwidth must increase with the number of nodes to support simultaneous communication on all ports. Some modern parallel computers, like the IBM SP, have a very natural solution for this problem. Each node of the distributed-memory parallel computer is a NUMA shared-memory multiprocessor. Multiple ports are then served by separate memory banks and full memory and communication bandwidth can be utilized if the buffers for sending and receiving data are placed appropriately across different memory banks.

4.8 Summary

Table 4.1 summarizes the communication times for various collective communications operations discussed in this chapter. The time for one-to-all broadcast, all-to-one reduction, and the all-reduce operations is the minimum of two expressions. This is because, depending on the message size m, either the algorithms described in Sections 4.1 and 4.3 or the ones described in Section 4.7 are faster. Table 4.1 assumes that the algorithm most suitable for the given message size is chosen. The communication-time expressions in Table 4.1 have been derived in the earlier sections of this chapter in the context of a hypercube interconnection network with cut-through routing. However, these expressions and the corresponding algorithms are valid for any architecture with a $\Theta(p)$ cross-section bandwidth (Section 2.4.4). In fact, the terms associated with t_w for the expressions for all operations listed in Table 4.1, except all-to-all personalized communication and circular shift, would remain unchanged even on ring and mesh networks (or any k-d mesh network) provided that the logical processes are mapped onto the physical nodes of the network appropriately. The last column of Table 4.1 gives the asymptotic cross-section bandwidth required to perform an operation in the time given by the second column of the table, assuming an optimal mapping of processes to nodes. For large messages, only all-to-all personalized communication and circular shift require the full $\Theta(p)$ cross-section bandwidth. Therefore, as discussed in Section 2.5.1, when applying the expressions for the time of these operations on a network with a smaller cross-section bandwidth, the t_w term must reflect the effective bandwidth. For example, the bisection width of a p-node square mesh is $\Theta(\sqrt{p})$ and that of a p-node ring is $\Theta(1)$. Therefore, while performing all-to-all personalized communication on a square mesh, the effective per-word transfer time would be $\Theta(\sqrt{p})$ times the t_w of individual links, and on a ring, it would be $\Theta(p)$ times the t_w of individual links.

Table 4.1. Summary of communication times of various operations discussed in Sections 4.1–4.7 on a hypercube interconnection network. The message size for each operation is m and the number of nodes is p.

Operation	Hypercube Time	B/W Requirement
One-to-all broadcast, All-to-one reduction	$\min((t_s + t_w m) \log p, 2(t_s \log p + t_w m))$	$\Theta(1)$
All-to-all broadcast, All-to-all reduction	$t_s \log p + t_w m(p - 1)$	$\Theta(1)$
All-reduce	$\min((t_s + t_w m) \log p, 2(t_s \log p + t_w m))$	$\Theta(1)$
Scatter, Gather	$t_s \log p + t_w m(p - 1)$	$\Theta(1)$
All-to-all personalized	$(t_s + t_w m)(p - 1)$	$\Theta(p)$
Circular shift	$t_s + t_w m$	$\Theta(p)$

Table 4.2. MPI names of the various operations discussed in this chapter.

Operation	MPI Name
One-to-all broadcast	MPI_Bcast
All-to-one reduction	MPI_Reduce
All-to-all broadcast	MPI_Allgather
All-to-all reduction	MPI_Reduce_scatter
All-reduce	MPI_Allreduce
Gather	MPI_Gather
Scatter	MPI_Scatter
All-to-all personalized	MPI_Alltoall

The collective communications operations discussed in this chapter occur frequently in many parallel algorithms. In order to facilitate speedy and portable design of efficient parallel programs, most parallel computer vendors provide pre-packaged software for performing these collective communications operations. The most commonly used standard API for these operations is known as the Message Passing Interface, or MPI. Table 4.2 gives the names of the MPI functions that correspond to the communications operations described in this chapter.

PREVIOUS NEXT

4.9 Bibliographic Remarks

In this chapter, we studied a variety of data communication operations for the linear array, mesh, and hypercube interconnection topologies. Saad and Schultz [SS89b] discuss implementation issues for these operations on these and other architectures, such as shared-memory and a switch or bus interconnect. Most parallel computer vendors provide standard APIs for inter-process communications via message-passing. Two of the most common APIs are the message passing interface (MPI) [SOHL+96] and the parallel virtual machine (PVM) [GBD+94].

The hypercube algorithm for a certain communication operation is often the best algorithm for other less-connected architectures too, if they support cut-through routing. Due to the versatility of the hypercube architecture and the wide applicability of its algorithms, extensive work has been done on implementing various communication operations on hypercubes [BOS+91, BR90, BT97, FF86, JH89, Joh90, MdV87, RS90b, SS89a, SW87]. The properties of a hypercube network that are used in deriving the algorithms for various communication operations on it are described by Saad and Schultz [SS88].

The all-to-all personalized communication problem in particular has been analyzed for the hypercube architecture by Boppana and Raghavendra [BR90], Johnsson and Ho [JH91], Seidel [Sei89], and Take [Tak87]. E-cube routing that guarantees congestion-free communication in Algorithm 4.10 for all-to-all personalized communication is described by Nugent [Nug88].

The all-reduce and the prefix sums algorithms of Section 4.3 are described by Ranka and Sahni [RS90b]. Our discussion of the circular shift operation is adapted from Bertsekas and Tsitsiklis [BT97]. A generalized form of prefix sums, often referred to as *scan*, has been used by some researchers as a basic primitive in data-parallel programming. Blelloch [Ble90] defines a *scan vector model*, and describes how a wide variety of parallel programs can be expressed in terms of the scan primitive and its variations.

The hypercube algorithm for one-to-all broadcast using spanning binomial trees is described by Bertsekas and Tsitsiklis [BT97] and Johnsson and Ho [JH89]. In the spanning tree algorithm described in Section 4.7.1, we split the m-word message to be broadcast into $\log p$ parts of size $m/\log p$ for ease of presenting the algorithm. Johnsson and Ho [JH89] show that the optimal size of the parts is $\lceil (\sqrt{t_s m / t_w \log p}) \rceil$. In this case, the number of messages may be greater than $\log p$. These smaller messages are sent from the root of the spanning binomial tree to its $\log p$ subtrees in a circular fashion. With this strategy, one-to-all broadcast on a p-node hypercube can be performed in time $t_s \log p + t_w m + 2 t_w \lceil (\sqrt{t_s m / t_w \log p}) \rceil \log p$.

Algorithms using the all-port communication model have been described for a variety of communication operations on the hypercube architecture by Bertsekas and Tsitsiklis [BT97], Johnsson and Ho [JH89], Ho and Johnsson [HJ87], Saad and Schultz [SS89a], and Stout and Wagar [SW87]. Johnsson and Ho [JH89] show that on a p-node hypercube with all-port communication, the coefficients of t_w in the expressions for the communication times of one-to-all and all-to-all broadcast and personalized communication are all smaller than those of their single-port counterparts by a factor of $\log p$. Gupta and Kumar [GK91] show that all-port communication may not improve the scalability of an algorithm on a parallel architecture over single-port communication.

The elementary operations described in this chapter are not the only ones used in parallel applications. A variety of other useful operations for parallel computers have been described in

literature, including selection [Akl89], pointer jumping [HS86, Jaj92], BPC permutations [Joh90, RS90b], fetch-and-op [GGK+83], packing [Lev87, Sch80], bit reversal [Loa92], and keyed-scan or multi-prefix [Ble90, Ran89].

Sometimes data communication does not follow any predefined pattern, but is arbitrary, depending on the application. In such cases, a simplistic approach of routing the messages along the shortest data paths between their respective sources and destinations leads to contention and imbalanced communication. Leighton, Maggs, and Rao [LMR88], Valiant [Val82], and Valiant and Brebner [VB81] discuss efficient routing methods for arbitrary permutations of messages.

PREVIOUS NEXT

Problems

4.1 Modify Algorithms 4.1, 4.2, and 4.3 so that they work for any number of processes, not just the powers of 2.

4.2 Section 4.1 presents the recursive doubling algorithm for one-to-all broadcast, for all three networks (ring, mesh, hypercube). Note that in the hypercube algorithm of Figure 4.6, a message is sent along the highest dimension first, and then sent to lower dimensions (in Algorithm 4.1, line 4, i goes down from d - 1 to 0). The same algorithm can be used for mesh and ring and ensures that messages sent in different time steps do not interfere with each other.

Let's now change the algorithm so that the message is sent along the lowest dimension first (i.e., in Algorithm 3.1, line 4, i goes up from 0 to d - 1). So in the first time step, processor 0 will communicate with processor 1; in the second time step, processors 0 and 1 will communicate with 2 and 3, respectively; and so on.

1. What is the run time of this revised algorithm on hypercube?
2. What is the run time of this revised algorithm on ring?

For these derivations, if k messages have to traverse the same link at the same time, then assume that the effective per-word-transfer time for these messages is kt_w.

4.3 On a ring, all-to-all broadcast can be implemented in two different ways: (a) the standard ring algorithm as shown in Figure 4.9 and (b) the hypercube algorithm as shown in Figure 4.11.

1. What is the run time for case (a)?
2. What is the run time for case (b)?

If k messages have to traverse the same link at the same time, then assume that the effective per-word-transfer time for these messages is kt_w. Also assume that $t_s = 100 \times t_w$.

1. Which of the two methods, (a) or (b), above is better if the message size m is very large?
2. Which method is better if m is very small (may be one word)?

4.4 Write a procedure along the lines of Algorithm 4.6 for performing all-to-all reduction on a mesh.

4.5 (All-to-all broadcast on a tree) Given a balanced binary tree as shown in Figure 4.7, describe a procedure to perform all-to-all broadcast that takes time $(t_s + t_w mp/2) \log p$ for m-word messages on p nodes. Assume that only the leaves of the tree contain nodes, and that an exchange of two m-word messages between any two nodes connected by bidirectional channels takes time $t_s + t_w mk$ if the communication channel (or a part of it) is shared by k simultaneous messages.

4.6 Consider the all-reduce operation in which each processor starts with an array of m words, and needs to get the global sum of the respective words in the array at each processor. This operation can be implemented on a ring using one of the following three alternatives:

i. All-to-all broadcast of all the arrays followed by a local computation of the sum of the respective elements of the array.

ii. Single node accumulation of the elements of the array, followed by a one-to-all broadcast of the result array.

iii. An algorithm that uses the pattern of the all-to-all broadcast, but simply adds numbers rather than concatenating messages.

1. For each of the above cases, compute the run time in terms of m, t_s, and t_w.
2. Assume that t_s = 100, t_w = 1, and m is very large. Which of the three alternatives (among (i), (ii) or (iii)) is better?
3. Assume that t_s = 100, t_w = 1, and m is very small (say 1). Which of the three alternatives (among (i), (ii) or (iii)) is better?

4.7 (One-to-all personalized communication on a linear array and a mesh) Give the procedures and their communication times for one-to-all personalized communication of m-word messages on p nodes for the linear array and the mesh architectures.

Hint: For the mesh, the algorithm proceeds in two phases as usual and starts with the source distributing pieces of $m\sqrt{p}$ words among the $\sqrt{p}$ nodes in its row such that each of these nodes receives the data meant for all the $\sqrt{p}$ nodes in its column.

4.8 (All-to-all reduction) The dual of all-to-all broadcast is all-to-all reduction, in which each node is the destination of an all-to-one reduction. For example, consider the scenario where p nodes have a vector of p elements each, and the i th node (for all i such that $0 \leq i < p$) gets the sum of the i th elements of all the vectors. Describe an algorithm to perform all-to-all reduction on a hypercube with addition as the associative operator. If each message contains m words and t_{add} is the time to perform one addition, how much time does your algorithm take (in terms of m, p, t_{add}, t_s and t_w)?

Hint: In all-to-all broadcast, each node starts with a single message and collects p such messages by the end of the operation. In all-to-all reduction, each node starts with p distinct messages (one meant for each node) but ends up with a single message.

4.9 Parts (c), (e), and (f) of Figure 4.21 show that for any node in a three-dimensional hypercube, there are exactly three nodes whose shortest distance from the node is two links. Derive an exact expression for the number of nodes (in terms of p and l) whose shortest distance from any given node in a p-node hypercube is l.

4.10 Give a hypercube algorithm to compute prefix sums of n numbers if p is the number of nodes and n/p is an integer greater than 1. Assuming that it takes time t_{add} to add two numbers and time t_s to send a message of unit length between two directly-connected nodes, give an exact expression for the total time taken by the algorithm.

4.11 Show that if the message startup time t_s is zero, then the expression $t_w mp(\sqrt{p}-1)$ for the time taken by all-to-all personalized communication on a $\sqrt{p} \times \sqrt{p}$ mesh is

optimal within a small ($\leq$ 4) constant factor.

4.12 Modify the linear array and the mesh algorithms in Sections 4.1–4.5 to work without the end-to-end wraparound connections. Compare the new communication times with those of the unmodified procedures. What is the maximum factor by which the time for any of the operations increases on either the linear array or the mesh?

4.13 (3-D mesh) Give optimal (within a small constant) algorithms for one-to-all and all-to-all broadcasts and personalized communications on a $p^{1/3} \times p^{1/3} \times p^{1/3}$ three-dimensional mesh of p nodes with store-and-forward routing. Derive expressions for the total communication times of these procedures.

4.14 Assume that the cost of building a parallel computer with p nodes is proportional to the total number of communication links within it. Let the cost effectiveness of an architecture be inversely proportional to the product of the cost of a p-node ensemble of this architecture and the communication time of a certain operation on it. Assuming t_s to be zero, which architecture is more cost effective for each of the operations discussed in this chapter – a standard 3-D mesh or a sparse 3-D mesh?

4.15 Repeat Problem 4.14 when t_s is a nonzero constant but $t_w = 0$. Under this model of communication, the message transfer time between two directly-connected nodes is fixed, regardless of the size of the message. Also, if two packets are combined and transmitted as one message, the communication latency is still t_s.

4.16 (k-to-all broadcast) Let k-to-all broadcast be an operation in which k nodes simultaneously perform a one-to-all broadcast of m-word messages. Give an algorithm for this operation that has a total communication time of $t_s \log p + t_w m(k \log(p/k) + k - 1)$ on a p-node hypercube. Assume that the m-word messages cannot be split, k is a power of 2, and $1 \leq k \leq p$.

4.17 Give a detailed description of an algorithm for performing all-to-one reduction in time $2(t_s \log p + t_w m(p - 1)/ p)$ on a p-node hypercube by splitting the original messages of size m into p nearly equal parts of size m/p each.

4.18 If messages can be split and their parts can be routed independently, then derive an algorithm for k-to-all broadcast such that its communication time is less than that of the algorithm in Problem 4.16 for a p-node hypercube.

4.19 Show that, if $m \geq p$, then all-to-one reduction with message size m can be performed on a p-node hypercube spending time $2(t_s \log p + t_w m)$ in communication.

Hint: Express all-to-one reduction as a combination of all-to-all reduction and gather.

4.20 (k-to-all personalized communication) In k-to-all personalized communication, k nodes simultaneously perform a one-to-all personalized communication ($1 \leq k \leq p$) in a p-node ensemble with individual packets of size m. Show that, if k is a power of 2, then this operation can be performed on a hypercube in time $t_s (\log(p/k) + k - 1) + t_w m(p - 1)$.

4.21 Assuming that it takes time t_r to perform a read and a write operation on a single word of data in a node's local memory, show that all-to-all personalized communication on a p-node mesh (Section 4.5.2) spends a total of time $t_r mp$ in internal data movement on the nodes, where m is the size of an individual message.

Hint: The internal data movement is equivalent to transposing a $\sqrt{p} \times \sqrt{p}$ array of messages of size m.

4.22 Show that in a p-node hypercube, all the p data paths in a circular q-shift are congestion-free if E-cube routing (Section 4.5) is used.

Hint: (1) If $q > p/2$, then a q-shift is isomorphic to a $(p - q)$-shift on a p-node hypercube. (2) Prove by induction on hypercube dimension. If all paths are congestion-free for a q-shift ($1 \leq q < p$) on a p-node hypercube, then all these paths are congestion-free on a 2 p-node hypercube also.

4.23 Show that the length of the longest path of any message in a circular q-shift on a p-node hypercube is $\log p - \gamma(q)$, where $\gamma(q)$ is the highest integer j such that q is divisible by 2^j.

Hint: (1) If $q = p/2$, then $\gamma(q) = \log p - 1$ on a p-node hypercube. (2) Prove by induction on hypercube dimension. For a given q, $\gamma(q)$ increases by one each time the number of nodes is doubled.

4.24 Derive an expression for the parallel run time of the hypercube algorithms for one-to-all broadcast, all-to-all broadcast, one-to-all personalized communication, and all-to-all personalized communication adapted unaltered for a mesh with identical communication links (same channel width and channel rate). Compare the performance of these adaptations with that of the best mesh algorithms.

4.25 As discussed in Section 2.4.4, two common measures of the cost of a network are (1) the total number of wires in a parallel computer (which is a product of number of communication links and channel width); and (2) the bisection bandwidth. Consider a hypercube in which the channel width of each link is one, that is $t_w = 1$. The channel width of a mesh-connected computer with equal number of nodes and identical cost is higher, and is determined by the cost metric used. Let s and s' represent the factors by which the channel width of the mesh is increased in accordance with the two cost metrics. Derive the values of s and s'. Using these, derive the communication time of the following operations on a mesh:

1. One-to-all broadcast
2. All-to-all broadcast
3. One-to-all personalized communication
4. All-to-all personalized communication

Compare these times with the time taken by the same operations on a hypercube with equal cost.

4.26 Consider a completely-connected network of p nodes. For the four communication operations in Problem 4.25 derive an expression for the parallel run time of the hypercube algorithms on the completely-connected network. Comment on whether the added connectivity of the network yields improved performance for these operations.

Chapter 5. Analytical Modeling of Parallel Programs

A sequential algorithm is usually evaluated in terms of its execution time, expressed as a function of the size of its input. The execution time of a parallel algorithm depends not only on input size but also on the number of processing elements used, and their relative computation and interprocess communication speeds. Hence, a parallel algorithm cannot be evaluated in isolation from a parallel architecture without some loss in accuracy. A *parallel system* is the combination of an algorithm and the parallel architecture on which it is implemented. In this chapter, we study various metrics for evaluating the performance of parallel systems.

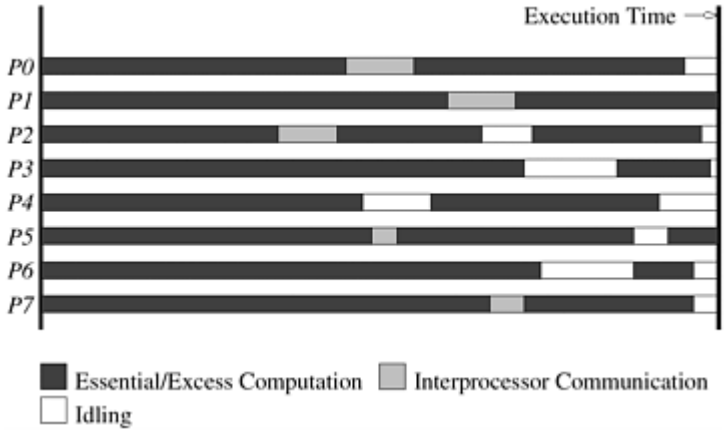
A number of measures of performance are intuitive. Perhaps the simplest of these is the wall-clock time taken to solve a given problem on a given parallel platform. However, as we shall see, a single figure of merit of this nature cannot be extrapolated to other problem instances or larger machine configurations. Other intuitive measures quantify the benefit of parallelism, i.e., how much faster the parallel program runs with respect to the serial program. However, this characterization suffers from other drawbacks, in addition to those mentioned above. For instance, what is the impact of using a poorer serial algorithm that is more amenable to parallel processing? For these reasons, more complex measures for extrapolating performance to larger machine configurations or problems are often necessary. With these objectives in mind, this chapter focuses on metrics for quantifying the performance of parallel programs.

5.1 Sources of Overhead in Parallel Programs

Using twice as many hardware resources, one can reasonably expect a program to run twice as fast. However, in typical parallel programs, this is rarely the case, due to a variety of overheads associated with parallelism. An accurate quantification of these overheads is critical to the understanding of parallel program performance.

A typical execution profile of a parallel program is illustrated in Figure 5.1. In addition to performing essential computation (i.e., computation that would be performed by the serial program for solving the same problem instance), a parallel program may also spend time in interprocess communication, idling, and excess computation (computation not performed by the serial formulation).

Figure 5.1. The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.



Interprocess Interaction Any nontrivial parallel system requires its processing elements to interact and communicate data (e.g., intermediate results). The time spent communicating data between processing elements is usually the most significant source of parallel processing overhead.

Idling Processing elements in a parallel system may become idle due to many reasons such as load imbalance, synchronization, and presence of serial components in a program. In many parallel applications (for example, when task generation is dynamic), it is impossible (or at least difficult) to predict the size of the subtasks assigned to various processing elements. Hence, the problem cannot be subdivided statically among the processing elements while maintaining uniform workload. If different processing elements have different workloads, some processing elements may be idle during part of the time that others are working on the problem. In some parallel programs, processing elements must synchronize at certain points during parallel program execution. If all processing elements are not ready for synchronization at the same time, then the ones that are ready sooner will be idle until all the rest are ready. Parts of an algorithm may be unparallelizable, allowing only a single processing element to work on it. While one processing element works on the serial part, all the other processing elements must wait.

Excess Computation The fastest known sequential algorithm for a problem may be difficult or impossible to parallelize, forcing us to use a parallel algorithm based on a poorer but easily parallelizable (that is, one with a higher degree of concurrency) sequential algorithm. The difference in computation performed by the parallel program and the best serial program is the excess computation overhead incurred by the parallel program.

A parallel algorithm based on the best serial algorithm may still perform more aggregate computation than the serial algorithm. An example of such a computation is the Fast Fourier Transform algorithm. In its serial version, the results of certain computations can be reused. However, in the parallel version, these results cannot be reused because they are generated by different processing elements. Therefore, some computations are performed multiple times on different processing elements. Chapter 13 discusses these algorithms in detail.

Since different parallel algorithms for solving the same problem incur varying overheads, it is important to quantify these overheads with a view to establishing a figure of merit for each algorithm.

5.2 Performance Metrics for Parallel Systems

It is important to study the performance of parallel programs with a view to determining the best algorithm, evaluating hardware platforms, and examining the benefits from parallelism. A number of metrics have been used based on the desired outcome of performance analysis.

5.2.1 Execution Time

The serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The *parallel runtime* is the time that elapses from the moment a parallel computation starts to the moment the last processing element finishes execution. We denote the serial runtime by T_S and the parallel runtime by T_P.

5.2.2 Total Parallel Overhead

The overheads incurred by a parallel program are encapsulated into a single expression referred to as the *overhead function*. We define overhead function or *total overhead* of a parallel system as the total time collectively spent by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element. We denote the overhead function of a parallel system by the symbol T_o.

The total time spent in solving a problem summed over all processing elements is pT_P . T_S units of this time are spent performing useful work, and the remainder is overhead. Therefore, the overhead function (T_o) is given by

Equation 5.1

$$T_o = pT_P - T_S.$$

5.2.3 Speedup

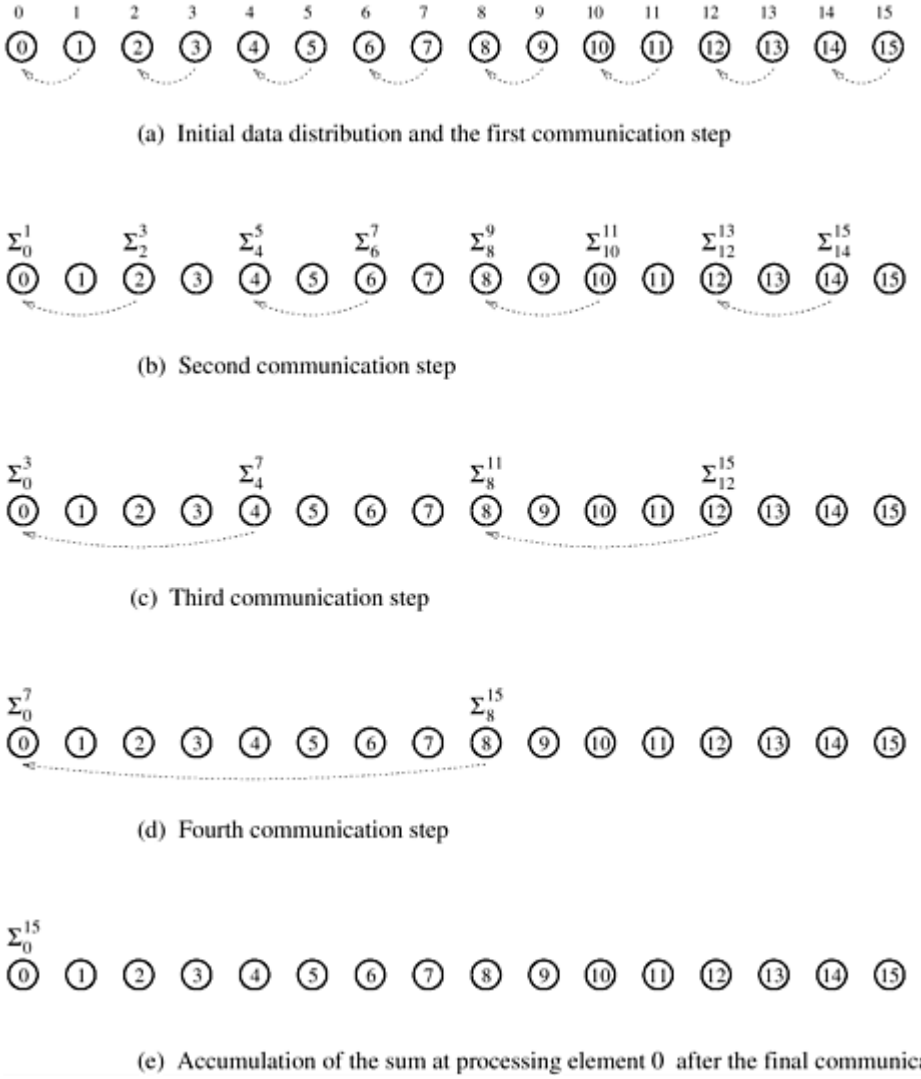
When evaluating a parallel system, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with p identical processing elements. We denote speedup by the symbol S.

Example 5.1 Adding n numbers using n processing elements

Consider the problem of adding n numbers by using n processing elements. Initially, each processing element is assigned one of the numbers to be added and, at the end

of the computation, one of the processing elements stores the sum of all the numbers. Assuming that n is a power of two, we can perform this operation in log n steps by propagating partial sums up a logical binary tree of processing elements. Figure 5.2 illustrates the procedure for $n = 16$. The processing elements are labeled from 0 to 15. Similarly, the 16 numbers to be added are labeled from 0 to 15. The sum of the numbers with consecutive labels from i to j is denoted by Σ_i^j.

Figure 5.2. Computing the globalsum of 16 partial sums using 16 processing elements. Σ_i^j denotes the sum of numbers with consecutive labels from i to j.



Each step shown in Figure 5.2 consists of one addition and the communication of a single word. The addition can be performed in some constant time, say t_c, and the communication of a single word can be performed in time $t_s + t_w$. Therefore, the addition and communication operations take a constant amount of time. Thus,

Equation 5.2

$$T_P = \Theta(\log n).$$

Since the problem can be solved in $\Theta(n)$ time on a single processing element, its speedup is

Equation 5.3

$$S = \Theta\left(\frac{n}{\log n}\right).$$

■

For a given problem, more than one sequential algorithm may be available, but all of these may not be equally suitable for parallelization. When a serial computer is used, it is natural to use the sequential algorithm that solves the problem in the least amount of time. Given a parallel algorithm, it is fair to judge its performance with respect to the fastest sequential algorithm for solving the same problem on a single processing element. Sometimes, the asymptotically fastest sequential algorithm to solve a problem is not known, or its runtime has a large constant that makes it impractical to implement. In such cases, we take the fastest known algorithm that would be a practical choice for a serial computer to be the best sequential algorithm. We compare the performance of a parallel algorithm to solve a problem with that of the best sequential algorithm to solve the same problem. We formally define the *speedup S* as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processing elements. The p processing elements used by the parallel algorithm are assumed to be identical to the one used by the sequential algorithm.

Example 5.2 Computing speedups of parallel programs

Consider the example of parallelizing bubble sort (Section 9.3.1). Assume that a serial version of bubble sort of 10^5 records takes 150 seconds and a serial quicksort can sort the same list in 30 seconds. If a parallel version of bubble sort, also called odd-even sort, takes 40 seconds on four processing elements, it would appear that the parallel odd-even sort algorithm results in a speedup of 150/40 or 3.75. However, this conclusion is misleading, as in reality the parallel algorithm results in a speedup of 30/40 or 0.75 with respect to the best serial algorithm. ■

Theoretically, speedup can never exceed the number of processing elements, p. If the best sequential algorithm takes T_S units of time to solve a given problem on a single processing element, then a speedup of p can be obtained on p processing elements if none of the processing elements spends more than time T_S/p. A speedup greater than p is possible only if each processing element spends less than time T_S/p solving the problem. In this case, a single processing element could emulate the p processing elements and solve the problem in fewer than T_S units of time. This is a contradiction because speedup, by definition, is computed with respect to the best sequential algorithm. If T_S is the serial runtime of the algorithm, then the problem cannot be solved in less than time T_S on a single processing element.

In practice, a speedup greater than p is sometimes observed (a phenomenon known as *superlinear speedup*). This usually happens when the work performed by a serial algorithm is greater than its parallel formulation or due to hardware features that put the serial implementation at a disadvantage. For example, the data for a problem might be too large to fit into the cache of a single processing element, thereby degrading its performance due to the use of slower memory elements. But when partitioned among several processing elements, the individual data-partitions would be small enough to fit into their respective processing elements' caches. In the remainder of this book, we disregard superlinear speedup due to hierarchical memory.

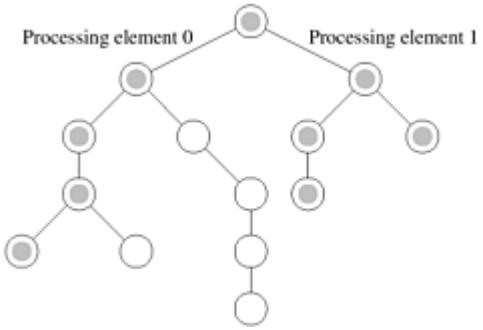
Example 5.3 Superlinearity effects from caches

Consider the execution of a parallel program on a two-processor parallel system. The program attempts to solve a problem instance of size W. With this size and available cache of 64 KB on one processor, the program has a cache hit rate of 80%. Assuming the latency to cache of 2 ns and latency to DRAM of 100 ns, the effective memory access time is 2 x 0.8 + 100 x 0.2, or 21.6 ns. If the computation is memory bound and performs one FLOP/memory access, this corresponds to a processing rate of 46.3 MFLOPS. Now consider a situation when each of the two processors is effectively executing half of the problem instance (i.e., size $W/2$). At this problem size, the cache hit ratio is expected to be higher, since the effective problem size is smaller. Let us assume that the cache hit ratio is 90%, 8% of the remaining data comes from local DRAM, and the other 2% comes from the remote DRAM (communication overhead). Assuming that remote data access takes 400 ns, this corresponds to an overall access time of 2 x 0.9 + 100 x 0.08 + 400 x 0.02, or 17.8 ns. The corresponding execution rate at each processor is therefore 56.18, for a total execution rate of 112.36 MFLOPS. The speedup in this case is given by the increase in speed over serial formulation, i.e., 112.36/46.3 or 2.43! Here, because of increased cache hit ratio resulting from lower problem size per processor, we notice superlinear speedup. ■

Example 5.4 Superlinearity effects due to exploratory decomposition

Consider an algorithm for exploring leaf nodes of an unstructured tree. Each leaf has a label associated with it and the objective is to find a node with a specified label, in this case 'S'. Such computations are often used to solve combinatorial problems, where the label 'S' could imply the solution to the problem (Section 11.6). In Figure 5.3, we illustrate such a tree. The solution node is the rightmost leaf in the tree. A serial formulation of this problem based on depth-first tree traversal explores the entire tree, i.e., all 14 nodes. If it takes time t_c to visit a node, the time for this traversal is $14t_c$. Now consider a parallel formulation in which the left subtree is explored by processing element 0 and the right subtree by processing element 1. If both processing elements explore the tree at the same speed, the parallel formulation explores only the shaded nodes before the solution is found. Notice that the total work done by the parallel algorithm is only nine node expansions, i.e., $9t_c$. The corresponding parallel time, assuming the root node expansion is serial, is $5t_c$ (one root node expansion, followed by four node expansions by each processing element). The speedup of this two-processor execution is therefore $14t_c/5t_c$, or 2.8!

Figure 5.3. Searching an unstructured tree for a node with a given label, 'S', on two processing elements using depth-first traversal. The two-processor version with processor 0 searching the left subtree and processor 1 searching the right subtree expands only the shaded nodes before the solution is found. The corresponding serial formulation expands the entire tree. It is clear that the serial algorithm does more work than the parallel algorithm.



The cause for this superlinearity is that the work performed by parallel and serial algorithms is different. Indeed, if the two-processor algorithm was implemented as two processes on the same processing element, the algorithmic superlinearity would disappear for this problem instance. Note that when exploratory decomposition is used, the relative amount of work performed by serial and parallel algorithms is dependent upon the location of the solution, and it is often not possible to find a serial algorithm that is optimal for all instances. Such effects are further analyzed in greater detail in Chapter 11. ■

5.2.4 Efficiency

Only an ideal parallel system containing *p* processing elements can deliver a speedup equal to *p*. In practice, ideal behavior is not achieved because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computations of the algorithm. As we saw in Example 5.1, part of the time required by the processing elements to compute the sum of *n* numbers is spent idling (and communicating in real systems). *Efficiency* is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. In an ideal parallel system, speedup is equal to *p* and efficiency is equal to one. In practice, speedup is less than *p* and efficiency is between zero and one, depending on the effectiveness with which the processing elements are utilized. We denote efficiency by the symbol *E*. Mathematically, it is given by

Equation 5.4

$$E = \frac{S}{p}.$$

Example 5.5 Efficiency of adding n numbers on n processing

elements

From Equation 5.3 and the preceding definition, the efficiency of the algorithm for adding n numbers on n processing elements is

$$E = \frac{\Theta\left(\frac{n}{\log n}\right)}{n} = \Theta\left(\frac{1}{\log n}\right)$$

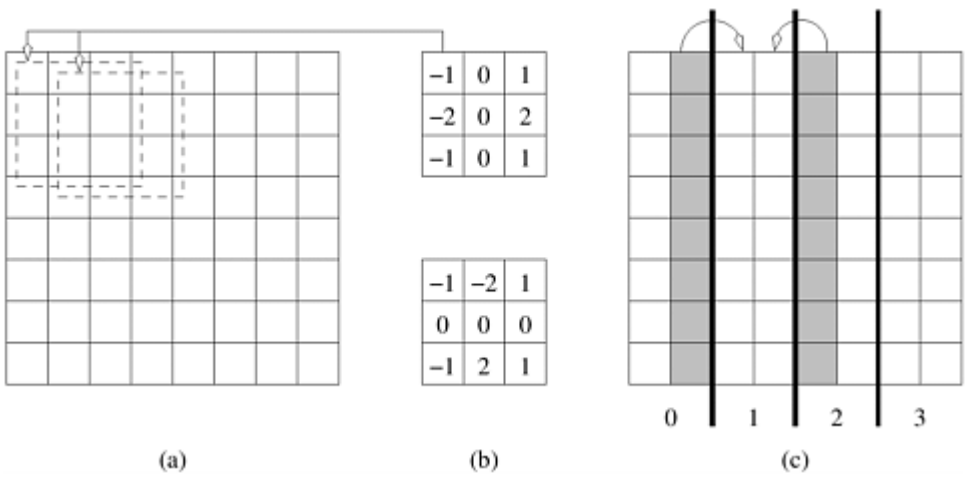
■

We also illustrate the process of deriving the parallel runtime, speedup, and efficiency while preserving various constants associated with the parallel platform.

Example 5.6 Edge detection on images

Given an n x n pixel image, the problem of detecting edges corresponds to applying a3x 3 template to each pixel. The process of applying the template corresponds to multiplying pixel values with corresponding template values and summing across the template (a convolution operation). This process is illustrated in Figure 5.4(a) along with typical templates (Figure 5.4(b)). Since we have nine multiply-add operations for each pixel, if each multiply-add takes time t_c, the entire operation takes time $9t_cn^2$ on a serial computer.

Figure 5.4. Example of edge detection: (a) an 8 x 8 image; (b) typical templates for detecting edges; and (c) partitioning of the image across four processors with shaded regions indicating image data that must be communicated from neighboring processors to processor 1.



A simple parallel algorithm for this problem partitions the image equally across the

processing elements and each processing element applies the template to its own subimage. Note that for applying the template to the boundary pixels, a processing element must get data that is assigned to the adjoining processing element. Specifically, if a processing element is assigned a vertically sliced subimage of dimension n x (n/p), it must access a single layer of n pixels from the processing element to the left and a single layer of n pixels from the processing element to the right (note that one of these accesses is redundant for the two processing elements assigned the subimages at the extremities). This is illustrated in Figure 5.4(c).

On a message passing machine, the algorithm executes in two steps: (i) exchange a layer of n pixels with each of the two adjoining processing elements; and (ii) apply template on local subimage. The first step involves two n-word messages (assuming each pixel takes a word to communicate RGB data). This takes time $2(t_s + t_w n)$. The second step takes time $9t_c n^2/p$. The total time for the algorithm is therefore given by:

$$T_P = 9t_c \frac{n^2}{p} + 2(t_s + t_w n)$$

The corresponding values of speedup and efficiency are given by:

$$S = \frac{9t_c n^2}{9t_c \frac{n^2}{p} + 2(t_s + t_w n)}$$

and

$$E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}.$$

■

5.2.5 Cost

We define the *cost* of solving a problem on a parallel system as the product of parallel runtime and the number of processing elements used. Cost reflects the sum of the time that each processing element spends solving the problem. Efficiency can also be expressed as the ratio of the execution time of the fastest known sequential algorithm for solving a problem to the cost of solving the same problem on p processing elements.

The cost of solving a problem on a single processing element is the execution time of the fastest known sequential algorithm. A parallel system is said to be *cost-optimal* if the cost of solving a problem on a parallel computer has the same asymptotic growth (in Θ terms) as a function of the input size as the fastest-known sequential algorithm on a single processing element. Since efficiency is the ratio of sequential cost to parallel cost, a cost-optimal parallel system has an efficiency of $\Theta(1)$.

Cost is sometimes referred to as *work* or *processor-time product*, and a cost-optimal system is also known as a pT_P -optimal system.

Example 5.7 Cost of adding n numbers on n processing elements

The algorithm given in Example 5.1 for adding n numbers on n processing elements has a processor-time product of $\Theta(n \log n)$. Since the serial runtime of this operation is $\Theta(n)$, the algorithm is not cost optimal. ■

Cost optimality is a very important practical concept although it is defined in terms of asymptotics. We illustrate this using the following example.

Example 5.8 Performance of non-cost optimal algorithms

Consider a sorting algorithm that uses n processing elements to sort the list in time $(\log n)^2$. Since the serial runtime of a (comparison-based) sort is $n \log n$, the speedup and efficiency of this algorithm are given by $n/\log n$ and $1/\log n$, respectively. The pT_P product of this algorithm is $n(\log n)^2$. Therefore, this algorithm is not cost optimal but only by a factor of $\log n$. Let us consider a realistic scenario in which the number of processing elements p is much less than n. An assignment of these n tasks to $p < n$ processing elements gives us a parallel time less than $n(\log n)^2/p$. This follows from the fact that if n processing elements take time $(\log n)^2$, then one processing element would take time $n(\log n)^2$; and p processing elements would take time $n(\log n)^2/p$. The corresponding speedup of this formulation is $p/\log n$. Consider the problem of sorting 1024 numbers ($n = 1024$, $\log n = 10$) on 32 processing elements. The speedup expected is only $p/\log n$ or 3.2. This number gets worse as n increases. For $n = 10^6$, $\log n = 20$ and the speedup is only 1.6. Clearly, there is a significant cost associated with not being cost-optimal even by a very small factor (note that a factor of $\log p$ is smaller than even $\sqrt{p}$). This emphasizes the practical importance of cost-optimality. ■

5.3 The Effect of Granularity on Performance

Example 5.7 illustrated an instance of an algorithm that is not cost-optimal. The algorithm discussed in this example uses as many processing elements as the number of inputs, which is excessive in terms of the number of processing elements. In practice, we assign larger pieces of input data to processing elements. This corresponds to increasing the granularity of computation on the processing elements. Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called *scaling down* a parallel system in terms of the number of processing elements. A naive way to scale down a parallel system is to design a parallel algorithm for one input element per processing element, and then use fewer processing elements to simulate a large number of processing elements. If there are n inputs and only p processing elements ($p < n$), we can use the parallel algorithm designed for n processing elements by assuming n virtual processing elements and having each of the p physical processing elements simulate n/p virtual processing elements.

As the number of processing elements decreases by a factor of n/p, the computation at each processing element increases by a factor of n/p because each processing element now performs the work of n/p processing elements. If virtual processing elements are mapped appropriately onto physical processing elements, the overall communication time does not grow by more than a factor of n/p. The total parallel runtime increases, at most, by a factor of n/p, and the processor-time product does not increase. Therefore, if a parallel system with n processing elements is cost-optimal, using p processing elements (where $p < n$)to simulate n processing elements preserves cost-optimality.

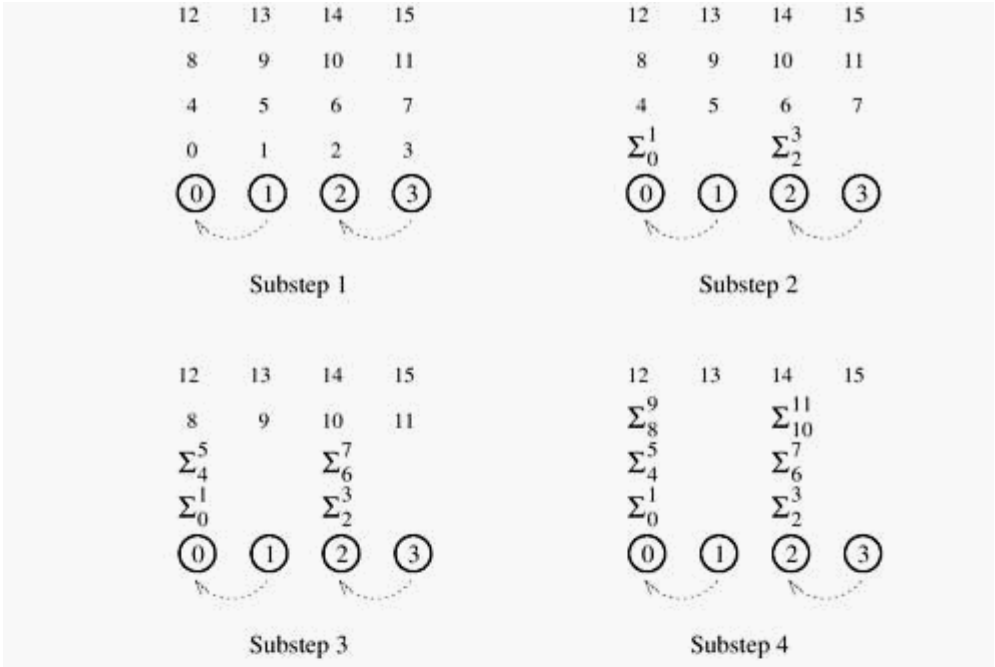
A drawback of this naive method of increasing computational granularity is that if a parallel system is not cost-optimal to begin with, it may still not be cost-optimal after the granularity of computation increases. This is illustrated by the following example for the problem of adding n numbers.

Example 5.9 Adding n numbers on p processing elements

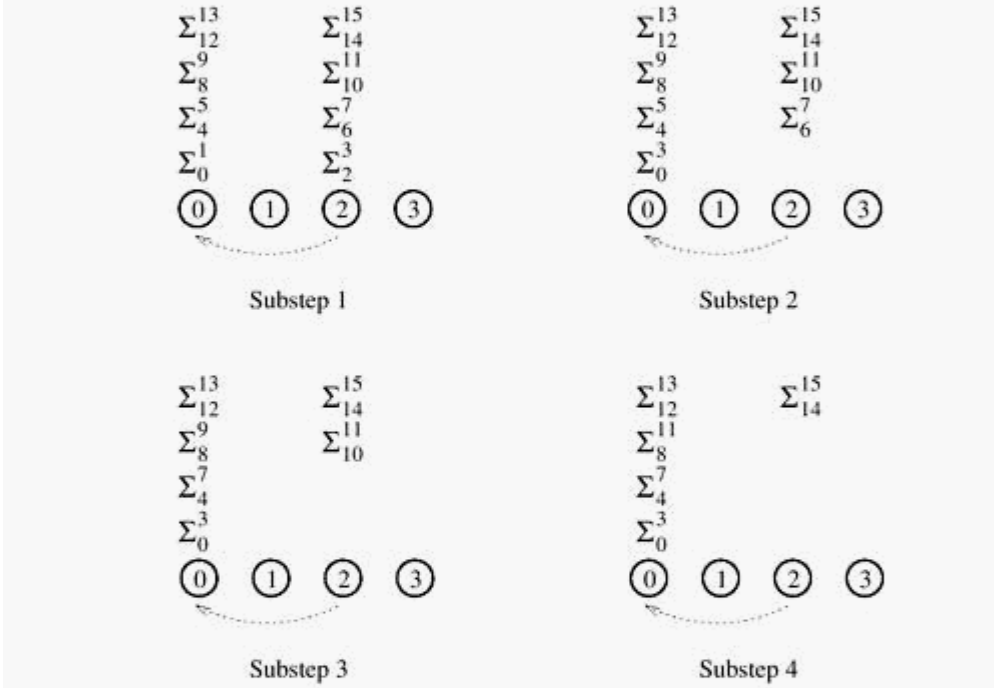
Consider the problem of adding n numbers on p processing elements such that $p < n$ and both n and p are powers of 2. We use the same algorithm as in Example 5.1 and simulate n processing elements on p processing elements. The steps leading to the solution are shown in Figure 5.5 for $n = 16$ and $p = 4$. Virtual processing element i is simulated by the physical processing element labeled $i \bmod p$; the numbers to be added are distributed similarly. The first $\log p$ of the $\log n$ steps of the original algorithm are simulated in $(n/p) \log p$ steps on p processing elements. In the remaining steps, no communication is required because the processing elements that communicate in the original algorithm are simulated by the same processing element; hence, the remaining numbers are added locally. The algorithm takes $\Theta((n/p) \log p)$ time in the steps that require communication, after which a single processing element is left with n/p numbers to add, taking time $\Theta(n/p)$. Thus, the overall parallel execution time of this parallel system is $\Theta((n/p) \log p)$. Consequently, its cost is $\Theta(n \log p)$, which is asymptotically higher than the $\Theta(n)$ cost of adding n numbers sequentially. Therefore, the parallel system is not cost-optimal. ■

Figure 5.5. Four processing elements simulating 16 processing

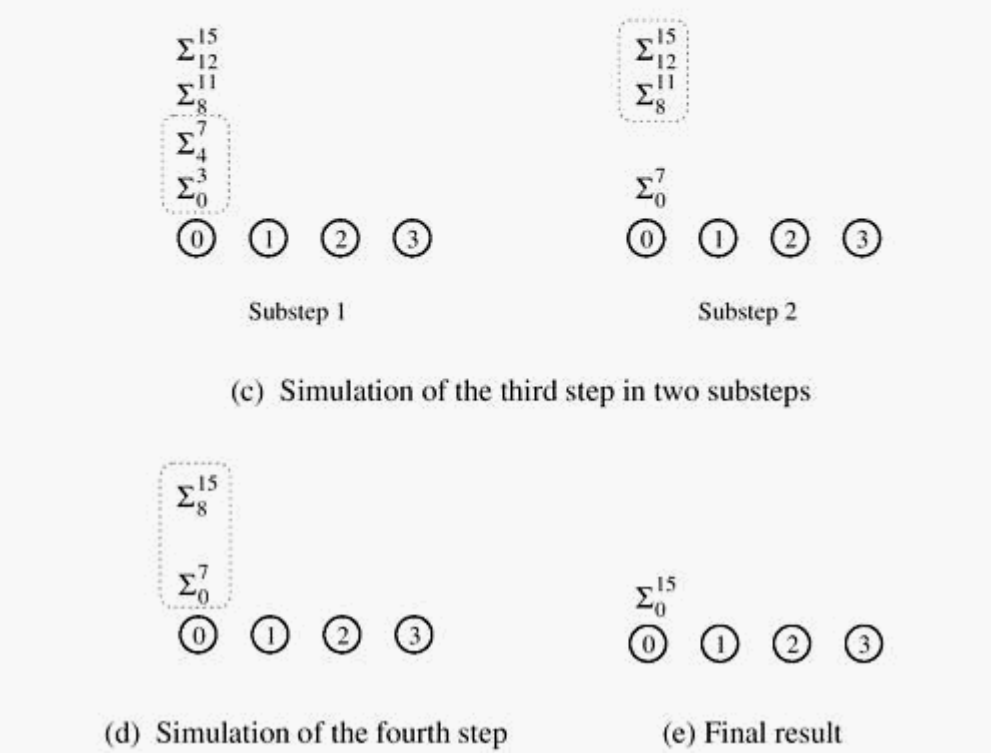
elements to compute the sum of 16 numbers (first two steps). Σ_i^j denotes the sum of numbers with consecutive labels from i to j. Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (last three steps).



(a) Four processors simulating the first communication step of 16 processors



(b) Four processors simulating the second communication step of 16 processors



(c) Simulation of the third step in two substeps

(d) Simulation of the fourth step

(e) Final result

Example 5.1 showed that n numbers can be added on an n-processor machine in time $\Theta(\log n)$. When using p processing elements to simulate n virtual processing elements ($p < n$), the expected parallel runtime is $\Theta((n/p) \log n)$. However, in Example 5.9 this task was performed in time $\Theta((n/p) \log p)$ instead. The reason is that every communication step of the original algorithm does not have to be simulated; at times, communication takes place between virtual processing elements that are simulated by the same physical processing element. For these operations, there is no associated overhead. For example, the simulation of the third and fourth steps (Figure 5.5(c) and (d)) did not require any communication. However, this reduction in communication was not enough to make the algorithm cost-optimal. Example 5.10 illustrates that the same problem (adding n numbers on p processing elements) can be performed cost-optimally with a smarter assignment of data to processing elements.

Example 5.10 Adding n numbers cost-optimally

An alternate method for adding n numbers using p processing elements is illustrated in Figure 5.6 for $n = 16$ and $p = 4$. In the first step of this algorithm, each processing element locally adds its n/p numbers in time $\Theta(n/p)$. Now the problem is reduced to adding the p partial sums on p processing elements, which can be done in time $\Theta(\log p)$ by the method described in Example 5.1. The parallel runtime of this algorithm is

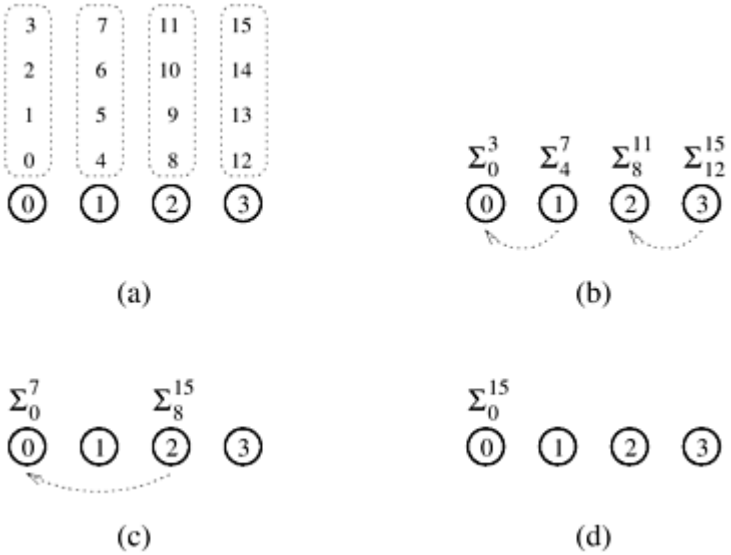
Equation 5.5

$$T_P = \Theta(n/p + \log p),$$

and its cost is $\Theta(n + p \log p)$. As long as $n = \Omega(p \log p)$, the cost is $\Theta(n)$, which is the

same as the serial runtime. Hence, this parallel system is cost-optimal. ■

Figure 5.6. A cost-optimal way of computing the sum of 16 numbers using four processing elements.



These simple examples demonstrate that the manner in which the computation is mapped onto processing elements may determine whether a parallel system is cost-optimal. Note, however, that we cannot make all non-cost-optimal systems cost-optimal by scaling down the number of processing elements.

PREVIOUS NEXT

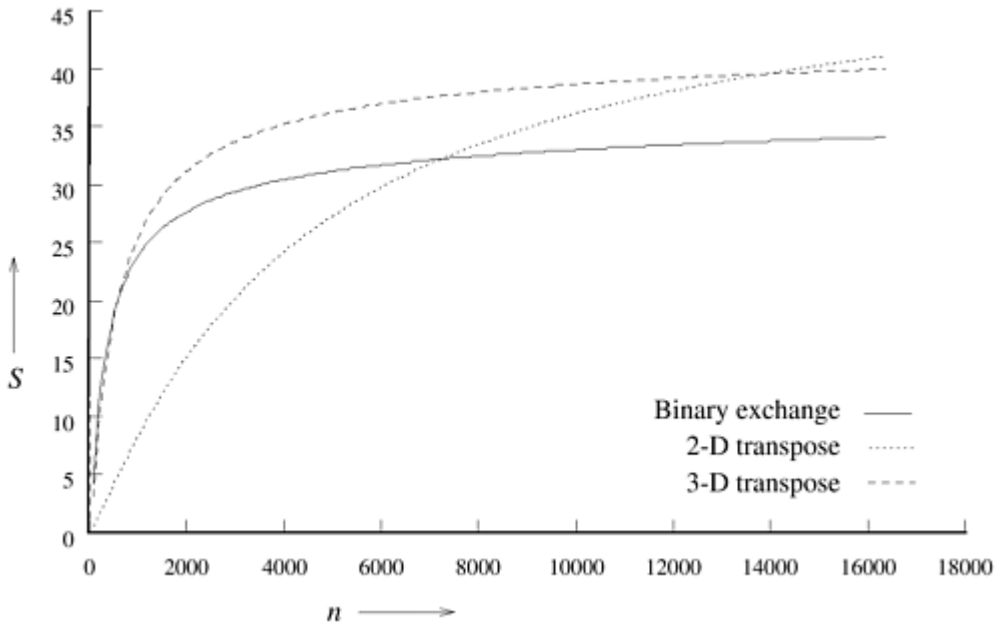
5.4 Scalability of Parallel Systems

Very often, programs are designed and tested for smaller problems on fewer processing elements. However, the real problems these programs are intended to solve are much larger, and the machines contain larger number of processing elements. Whereas code development is simplified by using scaled-down versions of the machine and the problem, their performance and correctness (of programs) is much more difficult to establish based on scaled-down systems. In this section, we will investigate techniques for evaluating the scalability of parallel programs using analytical tools.

Example 5.11 Why is performance extrapolation so difficult?

Consider three parallel algorithms for computing an n-point Fast Fourier Transform (FFT) on 64 processing elements. Figure 5.7 illustrates speedup as the value of n is increased to 18 K. Keeping the number of processing elements constant, at smaller values of n, one would infer from observed speedups that binary exchange and 3-D transpose algorithms are the best. However, as the problem is scaled up to 18 K points or more, it is evident from Figure 5.7 that the 2-D transpose algorithm yields best speedup. (These algorithms are discussed in greater detail in Chapter 13.) ■

Figure 5.7. A comparison of the speedups obtained by the binary-exchange, 2-D transpose and 3-D transpose algorithms on 64 processing elements with $t_c = 2$, $t_w = 4$, $t_s = 25$, and $t_h = 2$ (see Chapter 13 for details).



Similar results can be shown relating to the variation in number of processing elements as the problem size is held constant. Unfortunately, such parallel performance traces are the norm as opposed to the exception, making performance prediction based on limited observed data very difficult.

5.4.1 Scaling Characteristics of Parallel Programs

The efficiency of a parallel program can be written as:

$$E = \frac{S}{p} = \frac{T_S}{pT_P}$$

Using the expression for parallel overhead (Equation 5.1), we can rewrite this expression as

Equation 5.6

$$E = \frac{1}{1 + \frac{T_o}{T_S}}.$$

The total overhead function T_o is an increasing function of p. This is because every program must contain some serial component. If this serial component of the program takes time t_{serial}, then during this time all the other processing elements must be idle. This corresponds to a total overhead function of (p - 1) x t_{serial}. Therefore, the total overhead function T_o grows at least linearly with p. In addition, due to communication, idling, and excess computation, this function may grow superlinearly in the number of processing elements. Equation 5.6 gives us several interesting insights into the scaling of parallel programs. First, for a given problem size (i.e. the value of T_S remains constant), as we increase the number of processing elements, T_o increases. In such a scenario, it is clear from Equation 5.6 that the overall efficiency of the parallel program goes down. This characteristic of decreasing efficiency with increasing number of processing elements for a given problem size is common to all parallel programs.

Example 5.12 Speedup and efficiency as functions of the number of processing elements

Consider the problem of adding n numbers on p processing elements. We use the same algorithm as in Example 5.10. However, to illustrate actual speedups, we work with constants here instead of asymptotics. Assuming unit time for adding two numbers, the first phase (local summations) of the algorithm takes roughly n/p time. The second phase involves log p steps with a communication and an addition at each step. If a single communication takes unit time as well, the time for this phase is 2 log p. Therefore, we can derive parallel time, speedup, and efficiency as:

Equation 5.7

$$T_P = \frac{n}{p} + 2\log p$$

Equation 5.8

$$S = \frac{n}{\frac{n}{p} + 2\log p}$$

Equation 5.9

$$E = \frac{1}{1 + \frac{2p\log p}{n}}$$

These expressions can be used to calculate the speedup and efficiency for any pair of n and p. Figure 5.8 shows the S versus p curves for a few different values of n and p. Table 5.1 shows the corresponding efficiencies.

Figure 5.8. Speedup versus the number of processing elements for adding a list of numbers.

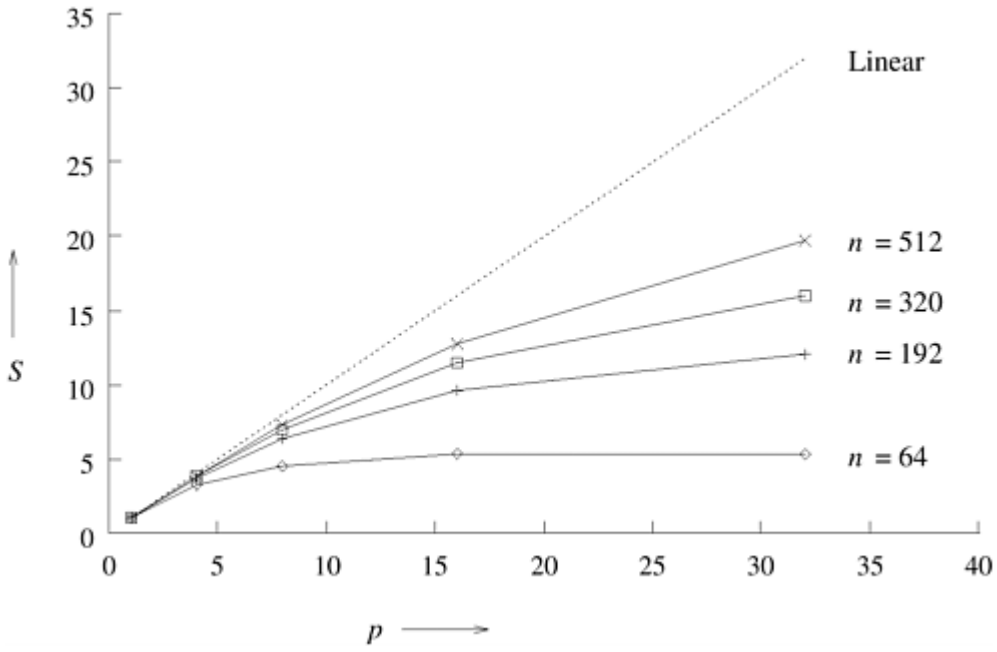


Figure 5.8 and Table 5.1 illustrate that the speedup tends to saturate and efficiency drops as a consequence of *Amdahl's law* (Problem 5.1). Furthermore, a larger instance of the same problem yields higher speedup and efficiency for the same number of processing elements, although both speedup and efficiency continue to drop with increasing p. ■

Let us investigate the effect of increasing the problem size keeping the number of processing elements constant. We know that the total overhead function T_o is a function of both problem size T_S and the number of processing elements p. In many cases, T_o grows sublinearly with

respect to T_S . In such cases, we can see that efficiency increases if the problem size is increased keeping the number of processing elements constant. For such algorithms, it should be possible to keep the efficiency fixed by increasing both the size of the problem and the number of processing elements simultaneously. For instance, in Table 5.1, the efficiency of adding 64 numbers using four processing elements is 0.80. If the number of processing elements is increased to 8 and the size of the problem is scaled up to add 192 numbers, the efficiency remains 0.80. Increasing p to 16 and n to 512 results in the same efficiency. This ability to maintain efficiency at a fixed value by simultaneously increasing the number of processing elements and the size of the problem is exhibited by many parallel systems. We call such systems *scalable* parallel systems. The *scalability* of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processing elements. It reflects a parallel system's ability to utilize increasing processing resources effectively.

Table 5.1. Efficiency as a function of n and p for adding n numbers on p processing elements.

n	$p = 1$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
64	1.0	*0.80*	0.57	0.33	0.17
192	1.0	0.92	*0.80*	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	*0.80*	0.62

Recall from Section 5.2.5 that a cost-optimal parallel system has an efficiency of $\Theta(1)$. Therefore, scalability and cost-optimality of parallel systems are related. A scalable parallel system can always be made cost-optimal if the number of processing elements and the size of the computation are chosen appropriately. For instance, Example 5.10 shows that the parallel system for adding n numbers on p processing elements is cost-optimal when $n = \Omega(p \log p)$. Example 5.13 shows that the same parallel system is scalable if n is increased in proportion to $\Theta(p \log p)$ as p is increased.

Example 5.13 Scalability of adding n numbers

For the cost-optimal addition of n numbers on p processing elements $n = \Omega(p \log p)$. As shown in Table 5.1, the efficiency is 0.80 for $n = 64$ and $p = 4$. At this point, the relation between n and p is $n = 8\ p \log p$. If the number of processing elements is increased to eight, then $8\ p \log p = 192$. Table 5.1 shows that the efficiency is indeed 0.80 with $n = 192$ for eight processing elements. Similarly, for $p = 16$, the efficiency is 0.80 for $n = 8\ p \log p = 512$. Thus, this parallel system remains cost-optimal at an efficiency of 0.80 if n is increased as $8\ p \log p$. ■

5.4.2 The Isoefficiency Metric of Scalability

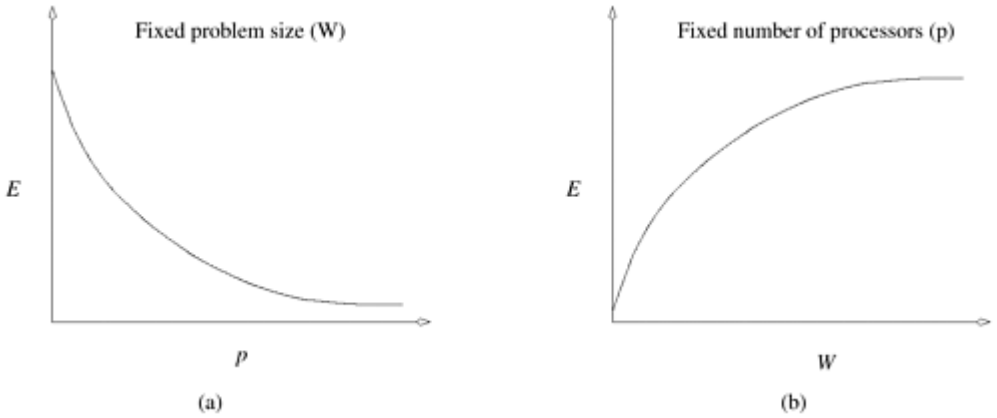
We summarize the discussion in the section above with the following two observations:

1. For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down. This phenomenon is common to all parallel systems.

2. In many cases, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.

These two phenomena are illustrated in Figure 5.9(a) and (b), respectively. Following from these two observations, we define a scalable parallel system as one in which the efficiency can be kept constant as the number of processing elements is increased, provided that the problem size is also increased. It is useful to determine the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed. For different parallel systems, the problem size must increase at different rates in order to maintain a fixed efficiency as the number of processing elements is increased. This rate determines the degree of scalability of the parallel system. As we shall show, a lower rate is more desirable than a higher growth rate in problem size. Let us now investigate metrics for quantitatively determining the degree of scalability of a parallel system. However, before we do that, we must define the notion of *problem size* precisely.

Figure 5.9. Variation of efficiency: (a) as the number of processing elements is increased for a given problem size; and (b) as the problem size is increased for a given number of processing elements. The phenomenon illustrated in graph (b) is not common to all parallel systems.



Problem Size When analyzing parallel systems, we frequently encounter the notion of the size of the problem being solved. Thus far, we have used the term *problem size* informally, without giving a precise definition. A naive way to express problem size is as a parameter of the input size; for instance, n in case of a matrix operation involving n x n matrices. A drawback of this definition is that the interpretation of problem size changes from one problem to another. For example, doubling the input size results in an eight-fold increase in the execution time for matrix multiplication and a four-fold increase for matrix addition (assuming that the conventional $\Theta(n^3)$ algorithm is the best matrix multiplication algorithm, and disregarding more complicated algorithms with better asymptotic complexities).

A consistent definition of the size or the magnitude of the problem should be such that, regardless of the problem, doubling the problem size always means performing twice the amount of computation. Therefore, we choose to express problem size in terms of the total number of basic operations required to solve the problem. By this definition, the problem size is $\Theta(n^3)$ for n x n matrix multiplication (assuming the conventional algorithm) and $\Theta(n^2)$ for n x n matrix addition. In order to keep it unique for a given problem, we define *problem size* as the number of basic computation steps in the best sequential algorithm to solve the problem on a

single processing element, where the best sequential algorithm is defined as in Section 5.2.3. Because it is defined in terms of sequential time complexity, the problem size is a function of the size of the input. The symbol we use to denote problem size is W.

In the remainder of this chapter, we assume that it takes unit time to perform one basic computation step of an algorithm. This assumption does not impact the analysis of any parallel system because the other hardware-related constants, such as message startup time, per-word transfer time, and per-hop time, can be normalized with respect to the time taken by a basic computation step. With this assumption, the problem size W is equal to the serial runtime T_S of the fastest known algorithm to solve the problem on a sequential computer.

The Isoefficiency Function

Parallel execution time can be expressed as a function of problem size, overhead function, and the number of processing elements. We can write parallel runtime as:

Equation 5.10

$$T_P = \frac{W + T_o(W, p)}{p}$$

The resulting expression for speedup is

Equation 5.11

$$\begin{aligned} S &= \frac{W}{T_P} \\ &= \frac{Wp}{W + T_o(W, p)}. \end{aligned}$$

Finally, we write the expression for efficiency as

Equation 5.12

$$\begin{aligned} E &= \frac{S}{p} \\ &= \frac{W}{W + T_o(W, p)} \\ &= \frac{1}{1 + T_o(W, p)/W}. \end{aligned}$$

In Equation 5.12, if the problem size is kept constant and p is increased, the efficiency decreases because the total overhead T_o increases with p. If W is increased keeping the number of processing elements fixed, then for scalable parallel systems, the efficiency increases. This is because T_o grows slower than $\Theta(W)$ for a fixed p. For these parallel systems, efficiency can be maintained at a desired value (between 0 and 1) for increasing p, provided W is also increased.

For different parallel systems, W must be increased at different rates with respect to p in order to maintain a fixed efficiency. For instance, in some cases, W might need to grow as an exponential function of p to keep the efficiency from dropping as p increases. Such parallel systems are poorly scalable. The reason is that on these parallel systems it is difficult to obtain good speedups for a large number of processing elements unless the problem size is enormous. On the other hand, if W needs to grow only linearly with respect to p, then the parallel system is highly scalable. That is because it can easily deliver speedups proportional to the number of processing elements for reasonable problem sizes.

For scalable parallel systems, efficiency can be maintained at a fixed value (between 0 and 1) if the ratio T_o/W in Equation 5.12 is maintained at a constant value. For a desired value E of efficiency,

Equation 5.13

$$\begin{aligned} E &= \frac{1}{1 + T_o(W, p)/W}, \\ \frac{T_o(W, p)}{W} &= \frac{1 - E}{E}, \\ W &= \frac{E}{1 - E} T_o(W, p). \end{aligned}$$

Let $K = E/(1 - E)$ be a constant depending on the efficiency to be maintained. Since T_o is a function of W and p, Equation 5.13 can be rewritten as

Equation 5.14

$$W = K T_o(W, p).$$

From Equation 5.14, the problem size W can usually be obtained as a function of p by algebraic manipulations. This function dictates the growth rate of W required to keep the efficiency fixed as p increases. We call this function the *isoefficiency function* of the parallel system. The isoefficiency function determines the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processing elements. A small isoefficiency function means that small increments in the problem size are sufficient for the efficient utilization of an increasing number of processing elements, indicating that the parallel system is highly scalable. However, a large isoefficiency function indicates a poorly scalable parallel system. The isoefficiency function does not exist for unscalable parallel systems, because in such systems the efficiency cannot be kept at any constant value as p increases, no matter how fast the problem size is increased.

Example 5.14 Isoefficiency function of adding numbers

The overhead function for the problem of adding n numbers on p processing elements is approximately $2p \log p$, as given by Equations 5.9 and 5.1. Substituting T_o by $2p \log p$ in Equation 5.14, we get

Equation 5.15

$$W = K2p \log p.$$

Thus, the asymptotic isoefficiency function for this parallel system is $\Theta(p \log p)$. This means that, if the number of processing elements is increased from p to p', the problem size (in this case, n) must be increased by a factor of $(p' \log p')/(p \log p)$ to get the same efficiency as on p processing elements. In other words, increasing the number of processing elements by a factor of p'/p requires that n be increased by a factor of $(p' \log p')/(p \log p)$ to increase the speedup by a factor of p'/p. ■

In the simple example of adding n numbers, the overhead due to communication (hereafter referred to as the *communication overhead*) is a function of p only. In general, communication overhead can depend on both the problem size and the number of processing elements. A typical overhead function can have several distinct terms of different orders of magnitude with respect to p and W. In such a case, it can be cumbersome (or even impossible) to obtain the isoefficiency function as a closed function of p. For example, consider a hypothetical parallel system for which $T_o = p^{3/2} + p^{3/4} W^{3/4}$. For this overhead function, Equation 5.14 can be rewritten as $W = Kp^{3/2} + Kp^{3/4} W^{3/4}$. It is hard to solve this equation for W in terms of p.

Recall that the condition for constant efficiency is that the ratio T_o/W remains fixed. As p and W increase, the efficiency is nondecreasing as long as none of the terms of T_o grow faster than W. If T_o has multiple terms, we balance W against each term of T_o and compute the respective isoefficiency functions for individual terms. The component of T_o that requires the problem size to grow at the highest rate with respect to p determines the overall asymptotic isoefficiency function of the parallel system. Example 5.15 further illustrates this technique of isoefficiency analysis.

Example 5.15 Isoefficiency function of a parallel system with a complex overhead function

Consider a parallel system for which $T_o = p^{3/2} + p^{3/4} W^{3/4}$. Using only the first term of T_o in Equation 5.14, we get

Equation 5.16

$$W = Kp^{3/2}.$$

Using only the second term, Equation 5.14 yields the following relation between W and p:

Equation 5.17

$$\begin{aligned} W &= Kp^{3/4}W^{3/4} \\ W^{1/4} &= Kp^{3/4} \\ W &= K^4p^3 \end{aligned}$$

To ensure that the efficiency does not decrease as the number of processing elements increases, the first and second terms of the overhead function require the problem size to grow as $\Theta(p^{3/2})$ and $\Theta(p^3)$, respectively. The asymptotically higher of the two rates, $\Theta(p^3)$, gives the overall asymptotic isoefficiency function of this parallel system, since it subsumes the rate dictated by the other term. The reader may indeed verify that if the problem size is increased at this rate, the efficiency is $\Theta(1)$ and that any rate lower than this causes the efficiency to fall with increasing p. ■

In a single expression, the isoefficiency function captures the characteristics of a parallel algorithm as well as the parallel architecture on which it is implemented. After performing isoefficiency analysis, we can test the performance of a parallel program on a few processing elements and then predict its performance on a larger number of processing elements. However, the utility of isoefficiency analysis is not limited to predicting the impact on performance of an increasing number of processing elements. Section 5.4.5 shows how the isoefficiency function characterizes the amount of parallelism inherent in a parallel algorithm. We will see in Chapter 13 that isoefficiency analysis can also be used to study the behavior of a parallel system with respect to changes in hardware parameters such as the speed of processing elements and communication channels. Chapter 11 illustrates how isoefficiency analysis can be used even for parallel algorithms for which we cannot derive a value of parallel runtime.

5.4.3 Cost-Optimality and the Isoefficiency Function

In Section 5.2.5, we stated that a parallel system is cost-optimal if the product of the number of processing elements and the parallel execution time is proportional to the execution time of the fastest known sequential algorithm on a single processing element. In other words, a parallel system is cost-optimal if and only if

Equation 5.18

$$pT_P = \Theta(W).$$

Substituting the expression for T_P from the right-hand side of Equation 5.10, we get the following:

Equation 5.19

$$\begin{aligned} W + T_o(W, p) &= \Theta(W) \\ T_o(W, p) &= O(W) \end{aligned}$$

Equation 5.20

$$W = \Omega(T_o(W, p))$$

Equations 5.19 and 5.20 suggest that a parallel system is cost-optimal if and only if its

overhead function does not asymptotically exceed the problem size. This is very similar to the condition given by Equation 5.14 for maintaining a fixed efficiency while increasing the number of processing elements in a parallel system. If Equation 5.14 yields an isoefficiency function $f(p)$, then it follows from Equation 5.20 that the relation $W = \Omega(f(p))$ must be satisfied to ensure the cost-optimality of a parallel system as it is scaled up. The following example further illustrates the relationship between cost-optimality and the isoefficiency function.

Example 5.16 Relationship between cost-optimality and isoefficiency

Consider the cost-optimal solution to the problem of adding n numbers on p processing elements, presented in Example 5.10. For this parallel system, $W \approx n$, and $T_o = \Theta(p \log p)$. From Equation 5.14, its isoefficiency function is $\Theta(p \log p)$; that is, the problem size must increase as $\Theta(p \log p)$ to maintain a constant efficiency. In Example 5.10 we also derived the condition for cost-optimality as $W = \Omega(p \log p)$. ■

5.4.4 A Lower Bound on the Isoefficiency Function

We discussed earlier that a smaller isoefficiency function indicates higher scalability. Accordingly, an ideally-scalable parallel system must have the lowest possible isoefficiency function. For a problem consisting of W units of work, no more than W processing elements can be used cost-optimally; additional processing elements will be idle. If the problem size grows at a rate slower than $\Theta(p)$ as the number of processing elements increases, then the number of processing elements will eventually exceed W. Even for an ideal parallel system with no communication, or other overhead, the efficiency will drop because processing elements added beyond $p = W$ will be idle. Thus, asymptotically, the problem size must increase at least as fast as $\Theta(p)$ to maintain fixed efficiency; hence, $\Omega(p)$ is the asymptotic lower bound on the isoefficiency function. It follows that the isoefficiency function of an ideally scalable parallel system is $\Theta(p)$.

5.4.5 The Degree of Concurrency and the Isoefficiency Function

A lower bound of $\Omega(p)$ is imposed on the isoefficiency function of a parallel system by the number of operations that can be performed concurrently. The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its *degree of concurrency*. The degree of concurrency is a measure of the number of operations that an algorithm can perform in parallel for a problem of size W; it is independent of the parallel architecture. If $C(W)$ is the degree of concurrency of a parallel algorithm, then for a problem of size W, no more than $C(W)$ processing elements can be employed effectively.

Example 5.17 Effect of concurrency on isoefficiency function

Consider solving a system of n equations in n variables by using Gaussian elimination (Section 8.3.1). The total amount of computation is $\Theta(n^3)$. But then variables must be eliminated one after the other, and eliminating each variable requires $\Theta(n^2)$

computations. Thus, at most $\Theta(n^2)$ processing elements can be kept busy at any time. Since $W = \Theta(n^3)$ for this problem, the degree of concurrency $C(W)$ is $\Theta(W^{2/3})$ and at most $\Theta(W^{2/3})$ processing elements can be used efficiently. On the other hand, given p processing elements, the problem size should be at least $\Omega(p^{3/2})$ to use them all. Thus, the isoefficiency function of this computation due to concurrency is $\Theta(p^{3/2})$. ■

The isoefficiency function due to concurrency is optimal (that is, $\Theta(p)$) only if the degree of concurrency of the parallel algorithm is $\Theta(W)$. If the degree of concurrency of an algorithm is less than $\Theta(W)$, then the isoefficiency function due to concurrency is worse (that is, greater) than $\Theta(p)$. In such cases, the overall isoefficiency function of a parallel system is given by the maximum of the isoefficiency functions due to concurrency, communication, and other overheads.

◄ PREVIOUS NEXT ►

5.5 Minimum Execution Time and Minimum Cost-Optimal Execution Time

We are often interested in knowing how fast a problem can be solved, or what the minimum possible execution time of a parallel algorithm is, provided that the number of processing elements is not a constraint. As we increase the number of processing elements for a given problem size, either the parallel runtime continues to decrease and asymptotically approaches a minimum value, or it starts rising after attaining a minimum value (Problem 5.12). We can determine the minimum parallel runtime T_P^{min} for a given W by differentiating the expression for T_P with respect to p and equating the derivative to zero (assuming that the function T_P (W, p) is differentiable with respect to p). The number of processing elements for which T_P is minimum is determined by the following equation:

Equation 5.21

$$\frac{\mathrm{d}}{\mathrm{d}p}T_P = 0$$

Let p_0 be the value of the number of processing elements that satisfies Equation 5.21. The value of T_P^{min} can be determined by substituting p_0 for p in the expression for T_P . In the following example, we derive the expression for T_P^{min} for the problem of adding n numbers.

Example 5.18 Minimum execution time for adding n numbers

Under the assumptions of Example 5.12, the parallel run time for the problem of adding n numbers on p processing elements can be approximated by

Equation 5.22

$$T_P = \frac{n}{p} + 2\log p.$$

Equating the derivative with respect to p of the right-hand side of Equation 5.22 to zero we get the solutions for p as follows:

Equation 5.23

$$\begin{aligned} -\frac{n}{p^2} + \frac{2}{p} &= 0 \\ -n + 2p &= 0 \\ p &= \frac{n}{2} \end{aligned}$$

Substituting $p = n/2$ in Equation 5.22, we get

Equation 5.24

$$T_P^{min} = 2 \log n.$$

■

In Example 5.18, the processor-time product for $p = p_0$ is $\Theta(n \log n)$, which is higher than the $\Theta(n)$ serial complexity of the problem. Hence, the parallel system is not cost-optimal for the value of p that yields minimum parallel runtime. We now derive an important result that gives a lower bound on parallel runtime if the problem is solved cost-optimally.

Let $T_P^{cost\_opt}$ be the minimum time in which a problem can be solved by a cost-optimal parallel system. From the discussion regarding the equivalence of cost-optimality and the isoefficiency function in Section 5.4.3, we conclude that if the isoefficiency function of a parallel system is $\Theta(f(p))$, then a problem of size W can be solved cost-optimally if and only if $W = \Omega(f(p))$. In other words, given a problem of size W, a cost-optimal solution requires that $p = O(f^{-1}(W))$. Since the parallel runtime is $\Theta(W/p)$ for a cost-optimal parallel system (Equation 5.18), the lower bound on the parallel runtime for solving a problem of size W cost-optimally is

Equation 5.25

$$T_P^{cost\_opt} = \Omega\left(\frac{W}{f^{-1}(W)}\right).$$

Example 5.19 Minimum cost-optimal execution time for adding n numbers

As derived in Example 5.14, the isoefficiency function $f(p)$ of this parallel system is $\Theta(p \log p)$. If $W = n = f(p) = p \log p$, then $\log n = \log p + \log \log p$. Ignoring the double logarithmic term, $\log n \approx \log p$. If $n = f(p) = p \log p$, then $p = f^{-1}(n) = n/\log p \approx n/\log n$. Hence, $f^{-1}(W) = \Theta(n/\log n)$. As a consequence of the relation between cost-optimality and the isoefficiency function, the maximum number of processing elements that can be used to solve this problem cost-optimally is $\Theta(n/\log n)$. Using $p = n/\log n$ in Equation 5.2, we get

Equation 5.26

$$\begin{aligned} T_P^{cost\_opt} &= \log n + \log\left(\frac{n}{\log n}\right) \\ &= 2 \log n - \log \log n. \end{aligned}$$

■

It is interesting to observe that both T_P^{min} and $T_P^{cost\_opt}$ for adding n numbers are $\Theta(\log n)$ (Equations 5.24 and 5.26). Thus, for this problem, a cost-optimal solution is also the asymptotically fastest solution. The parallel execution time cannot be reduced asymptotically by using a value of p greater than that suggested by the isoefficiency function for a given problem size (due to the equivalence between cost-optimality and the isoefficiency function). This is not true for parallel systems in general, however, and it is quite possible that $T_P^{cost\_opt} > \Theta(T_P^{min})$. The following example illustrates such a parallel system.

Example 5.20 A parallel system with

Consider the hypothetical parallel system of Example 5.15, for which

Equation 5.27

$$T_o = p^{3/2} + p^{3/4}W^{3/4}.$$

From Equation 5.10, the parallel runtime for this system is

Equation 5.28

$$T_P = \frac{W}{p} + p^{1/2} + \frac{W^{3/4}}{p^{1/4}}.$$

Using the methodology of Example 5.18,

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}p}T_P = -\frac{W}{p^2} + \frac{1}{2p^{1/2}} - \frac{W^{3/4}}{4p^{5/4}} &= 0, \\
-W + \frac{1}{2}p^{3/2} - \frac{1}{4}W^{3/4}p^{3/4} &= 0, \\
p^{3/4} &= \frac{1}{4}W^{3/4} \pm (\frac{1}{16}W^{3/2} + 2W)^{1/2} \\
&= \Theta(W^{3/4}), \\
p &= \Theta(W).
\end{aligned}$$

From the preceding analysis, $p_0 = \Theta(W)$. Substituting p by the value of p_0 in Equation 5.28, we get

Equation 5.29

$$T_P^{min} = \Theta(W^{1/2}).$$

According to Example 5.15, the overall isoefficiency function for this parallel system is $\Theta(p^3)$, which implies that the maximum number of processing elements that can be used cost-optimally is $\Theta(W^{1/3})$. Substituting $p = \Theta(W^{1/3})$ in Equation 5.28, we get

Equation 5.30

$$T_P^{cost\_opt} = \Theta(W^{2/3}).$$

A comparison of Equations 5.29 and 5.30 shows that $T_P^{cost\_opt}$ is asymptotically greater than T_P^{min}. ■

In this section, we have seen examples of both types of parallel systems: those for which $T_P^{cost\_opt}$ is asymptotically equal to T_P^{min}, and those for which $T_P^{cost\_opt}$ is asymptotically greater than T_P^{min}. Most parallel systems presented in this book are of the first type. Parallel systems for which the runtime can be reduced by an order of magnitude by using an asymptotically higher number of processing elements than indicated by the isoefficiency function are rare.

While deriving the minimum execution time for any parallel system, it is important to be aware that the maximum number of processing elements that can be utilized is bounded by the degree of concurrency $C(W)$ of the parallel algorithm. It is quite possible that p_0 is greater than $C(W)$ for a parallel system (Problems 5.13 and 5.14). In such cases, the value of p_0 is meaningless, and T_P^{min} is given by

Equation 5.31

$$T_P^{min} = \frac{W + T_o(W, C(W))}{C(W)}.$$

5.6 Asymptotic Analysis of Parallel Programs

At this point, we have accumulated an arsenal of powerful tools for quantifying the performance and scalability of an algorithm. Let us illustrate the use of these tools for evaluating a set of parallel programs for solving a given problem. Often, we ignore constants and concern ourselves with the asymptotic behavior of quantities. In many cases, this can yield a clearer picture of relative merits and demerits of various parallel programs.

Table 5.2. Comparison of four different algorithms for sorting a given list of numbers. The table shows number of processing elements, parallel runtime, speedup, efficiency and the pT_P product.

Algorithm	A1	A2	A3	A4
p	n^2	$\log n$	n	$\sqrt{n}$
T_P	1	n	$\sqrt{n}$	$\sqrt{n}\log n$
S	$n \log n$	$\log n$	$\sqrt{n}\log n$	$\sqrt{n}$
E	$\frac{\log n}{n}$	1	$\frac{\log n}{\sqrt{n}}$	1
pT_P	n^2	$n \log n$	$n^{1.5}$	$n \log n$

Consider the problem of sorting a list of n numbers. The fastest serial programs for this problem run in time $O(n \log n)$. Let us look at four different parallel algorithms A1, A2, A3, and A4, for sorting a given list. The parallel runtime of the four algorithms along with the number of processing elements they can use is given in Table 5.2. The objective of this exercise is to determine which of these four algorithms is the best. Perhaps the simplest metric is one of speed; the algorithm with the lowest T_P is the best. By this metric, algorithm A1 is the best, followed by A3, A4, and A2. This is also reflected in the fact that the speedups of the set of algorithms are also in this order.

However, in practical situations, we will rarely have n^2 processing elements as are required by algorithm A1. Furthermore, resource utilization is an important aspect of practical program design. So let us look at how efficient each of these algorithms are. This metric of evaluating the algorithm presents a starkly different image. Algorithms A2 and A4 are the best, followed by A3 and A1. The last row of Table 5.2 presents the cost of the four algorithms. From this row, it is evident that the costs of algorithms A1 and A3 are higher than the serial runtime of $n \log n$ and therefore neither of these algorithms is cost optimal. However, algorithms A2 and A4 are cost optimal.

This set of algorithms illustrate that it is important to first understand the objectives of parallel algorithm analysis and to use appropriate metrics. This is because use of different metrics may often result in contradictory outcomes.

5.7 Other Scalability Metrics

A number of other metrics of scalability of parallel systems have been proposed. These metrics are specifically suited to different system requirements. For example, in real time applications, the objective is to scale up a system to accomplish a task in a specified time bound. One such application is multimedia decompression, where MPEG streams must be decompressed at the rate of 25 frames/second. Consequently, a parallel system must decode a single frame in 40 ms (or with buffering, at an average of 1 frame in 40 ms over the buffered frames). Other such applications arise in real-time control, where a control vector must be generated in real-time. Several scalability metrics consider constraints on physical architectures. In many applications, the maximum size of a problem is constrained not by time, efficiency, or underlying models, but by the memory available on the machine. In such cases, metrics make assumptions on the growth function of available memory (with number of processing elements) and estimate how the performance of the parallel system changes with such scaling. In this section, we examine some of the related metrics and how they can be used in various parallel applications.

Scaled Speedup This metric is defined as the speedup obtained when the problem size is increased linearly with the number of processing elements. If the scaled-speedup curve is close to linear with respect to the number of processing elements, then the parallel system is considered scalable. This metric is related to isoefficiency if the parallel algorithm under consideration has linear or near-linear isoefficiency function. In this case the scaled-speedup metric provides results very close to those of isoefficiency analysis, and the scaled-speedup is linear or near-linear with respect to the number of processing elements. For parallel systems with much worse isoefficiencies, the results provided by the two metrics may be quite different. In this case, the scaled-speedup versus number of processing elements curve is sublinear.

Two generalized notions of scaled speedup have been examined. They differ in the methods by which the problem size is scaled up with the number of processing elements. In one method, the size of the problem is increased to fill the available memory on the parallel computer. The assumption here is that aggregate memory of the system increases with the number of processing elements. In the other method, the size of the problem grows with p subject to an upper-bound on execution time.

Example 5.21 Memory and time-constrained scaled speedup for matrix-vector products

The serial runtime of multiplying a matrix of dimension n x n with a vector is $t_c n^2$, where t_c is the time for a single multiply-add operation. The corresponding parallel runtime using a simple parallel algorithm is given by:

$$T_P = t_c \frac{n^2}{p} + t_s \log p + t_w n$$

and the speedup S is given by:

Equation 5.32

$$S = \frac{t_c n^2}{t_c \frac{n^2}{p} + t_s \log p + t_w n}$$

The total memory requirement of the algorithm is $\Theta(n^2)$. Let us consider the two cases of problem scaling. In the case of memory constrained scaling, we assume that the memory of the parallel system grows linearly with the number of processing elements, i.e., $m = \Theta(p)$. This is a reasonable assumption for most current parallel platforms. Since $m = \Theta(n^2)$, we have $n^2 = c \times p$, for some constant c. Therefore, the scaled speedup S' is given by:

$$S' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + t_w \sqrt{c \times p}}$$

or

$$S' = \frac{c_1 p}{c_2 + c_3 \log p + c_4 \sqrt{p}}.$$

In the limiting case, $S' = O(\sqrt{p})$.

In the case of time constrained scaling, we have $T_P = O(n^2/p)$. Since this is constrained to be constant, $n^2 = O(p)$. We notice that this case is identical to the memory constrained case. This happened because the memory and runtime of the algorithm are asymptotically identical. ■

Example 5.22 Memory and time-constrained scaled speedup for matrix-matrix products

The serial runtime of multiplying two matrices of dimension $n \times n$ is $t_c n^3$, where t_c, as before, is the time for a single multiply-add operation. The corresponding parallel runtime using a simple parallel algorithm is given by:

$$T_P = t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}$$

and the speedup S is given by:

Equation 5.33

$$S = \frac{t_c n^3}{t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}}$$

The total memory requirement of the algorithm is $\Theta(n^2)$. Let us consider the two cases of problem scaling. In the case of memory constrained scaling, as before, we assume that the memory of the parallel system grows linearly with the number of processing elements, i.e., $m = \Theta(p)$. Since $m = \Theta(n^2)$, we have $n^2 = c \times p$, for some constant c. Therefore, the scaled speedup S' is given by:

$$S' = \frac{t_c(c \times p)^{1.5}}{t_c \frac{(c \times p)^{1.5}}{p} + t_s \log p + 2t_w \frac{c \times p}{\sqrt{p}}} = O(p)$$

In the case of time constrained scaling, we have $T_P = O(n^3/p)$. Since this is constrained to be constant, $n^3 = O(p)$, or $n^3 = c \times p$ (for some constant c).

Therefore, the time-constrained speedup S'' is given by:

$$S'' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + 2t_w \frac{(c \times p)^{2/3}}{\sqrt{p}}} = O(p^{5/6})$$

This example illustrates that memory-constrained scaling yields linear speedup, whereas time-constrained speedup yields sublinear speedup in the case of matrix multiplication. ■

Serial Fraction f The experimentally determined serial fraction f can be used to quantify the performance of a parallel system on a fixed-size problem. Consider a case when the serial runtime of a computation can be divided into a totally parallel and a totally serial component, i.e.,

$$W = T_{ser} + T_{par}.$$

Here, T_{ser} and T_{par} correspond to totally serial and totally parallel components. From this, we can write:

$$T_P = T_{ser} + \frac{T_{par}}{p}.$$

Here, we have assumed that all of the other parallel overheads such as excess computation and communication are captured in the serial component T_{ser}. From these equations, it follows that:

Equation 5.34

$$T_P = T_{ser} + \frac{W - T_{ser}}{p}$$

The serial fraction f of a parallel program is defined as:

$$f = \frac{T_{ser}}{W}.$$

Therefore, from Equation 5.34, we have:

$$\begin{aligned} T_P &= f \times W + \frac{W - f \times W}{p} \\ \frac{T_P}{W} &= f + \frac{1 - f}{p} \end{aligned}$$

Since $S = W/T_P$, we have

$$\frac{1}{S} = f + \frac{1 - f}{p}.$$

Solving for f , we get:

Equation 5.35

$$f = \frac{1/S - 1/p}{1 - 1/p}.$$

It is easy to see that smaller values of f are better since they result in higher efficiencies. If f increases with the number of processing elements, then it is considered as an indicator of rising communication overhead, and thus an indicator of poor scalability.

Example 5.23 Serial component of the matrix-vector product

From Equations 5.35 and 5.32, we have

Equation 5.36

$$f = \frac{\frac{t_c n^2}{p} + t_s \log p + t_w n}{t_c n^2} \Big/ \left(1 - 1/p\right)$$

Simplifying the above expression, we get

$$f = \frac{t_s p \log p + t_w n p}{t_c n^2} \times \frac{1}{p-1}$$
$$f \approx \frac{t_s \log p + t_w n}{t_c n^2}$$

It is useful to note that the denominator of this equation is the serial runtime of the algorithm and the numerator corresponds to the overhead in parallel execution. ■

In addition to these metrics, a number of other metrics of performance have been proposed in the literature. We refer interested readers to the bibliography for references to these.

PREVIOUS NEXT

5.8 Bibliographic Remarks

To use today's massively parallel computers effectively, larger problems must be solved as more processing elements are added. However, when the problem size is fixed, the objective is to attain the best compromise between efficiency and parallel runtime. Performance issues for fixed-size problems have been addressed by several researchers [FK89, GK93a, KF90, NW88, TL90, Wor90]. In most situations, additional computing power derived from increasing the number of processing elements can be used to solve bigger problems. In some situations, however, different ways of increasing the problem size may apply, and a variety of constraints may guide the scaling up of the workload with respect to the number of processing elements [SHG93]. Time-constrained scaling and memory-constrained scaling have been explored by Gustafson *et al.* [GMB88, Gus88, Gus92], Sun and Ni [SN90, SN93], and Worley [Wor90, Wor88, Wor91] (Problem 5.9).

An important scenario is one in which we want to make the most efficient use of the parallel system; in other words, we want the overall performance of the parallel system to increase linearly with p. This is possible only for scalable parallel systems, which are exactly those for which a fixed efficiency can be maintained for arbitrarily large p by simply increasing the problem size. For such systems, it is natural to use the isoefficiency function or related metrics [GGK93, CD87, KR87b, KRS88]. Isoefficiency analysis has been found to be very useful in characterizing the scalability of a variety of parallel algorithms [GK91, GK93b, GKS92, HX98, KN91, KR87b, KR89, KS91b, RS90b, SKAT91b, TL90, WS89, WS91]. Gupta and Kumar [GK93a, KG94] have demonstrated the relevance of the isoefficiency function in the fixed time case as well. They have shown that if the isoefficiency function is greater than $\Theta(p)$, then the problem size cannot be increased indefinitely while maintaining a fixed execution time, no matter how many processing elements are used. A number of other researchers have analyzed the performance of parallel systems with concern for overall efficiency [EZL89, FK89, MS88, NW88, TL90, Zho89, ZRV89].

Kruskal, Rudolph, and Snir [KRS88] define the concept of *parallel efficient (PE)* problems. Their definition is related to the concept of isoefficiency function. Problems in the class PE have algorithms with a polynomial isoefficiency function at some efficiency. The class PE makes an important distinction between algorithms with polynomial isoefficiency functions and those with worse isoefficiency functions. Kruskal *et al.* proved the invariance of the class PE over a variety of parallel computational models and interconnection schemes. An important consequence of this result is that an algorithm with a polynomial isoefficiency on one architecture will have a polynomial isoefficiency on many other architectures as well. There can be exceptions, however; for instance, Gupta and Kumar [GK93b] show that the fast Fourier transform algorithm has a polynomial isoefficiency on a hypercube but an exponential isoefficiency on a mesh.

Vitter and Simons [VS86] define a class of problems called *PC**. PC* includes problems with efficient parallel algorithms on a PRAM. A problem in class *P* (the polynomial-time class) is in PC* if it has a parallel algorithm on a PRAM that can use a polynomial (in terms of input size) number of processing elements and achieve a minimal efficiency ϵ. Any problem in PC* has at least one parallel algorithm such that, for an efficiency ϵ, its isoefficiency function exists and is a polynomial.

A discussion of various scalability and performance measures can be found in the survey by Kumar and Gupta [KG94]. Besides the ones cited so far, a number of other metrics of performance and scalability of parallel systems have been proposed [BW89, CR89, CR91, Fla90, Hil90, Kun86, Mol87, MR, NA91, SG91, SR91, SZ96, VC89].

Flatt and Kennedy [FK89, Fla90] show that if the overhead function satisfies certain mathematical properties, then there exists a unique value p_0 of the number of processing elements for which T_P is minimum for a given W. A property of T_o on which their analysis depends heavily is that $T_o > \Theta(p)$. Gupta and Kumar [GK93a] show that there exist parallel systems that do not obey this condition, and in such cases the point of peak performance is determined by the degree of concurrency of the algorithm being used.

Marinescu and Rice [MR] develop a model to describe and analyze a parallel computation on an MIMD computer in terms of the number of threads of control p into which the computation is divided and the number of events $g(p)$ as a function of p. They consider the case where each event is of a fixed duration q and hence $T_o = qg(p)$. Under these assumptions on T_o, they conclude that with increasing number of processing elements, the speedup saturates at some value if $T_o = \Theta(p)$, and it asymptotically approaches zero if $T_o = \Theta(p^m)$,where $m \geq 2$. Gupta and Kumar [GK93a] generalize these results for a wider class of overhead functions. They show that the speedup saturates at some maximum value if $T_o \leq \Theta(p)$, and the speedup attains a maximum value and then drops monotonically with p if $T_o > \Theta(p)$.

Eager *et al.* [EZL89] and Tang and Li [TL90] have proposed a criterion of optimality of a parallel system so that a balance is struck between efficiency and speedup. They propose that a good choice of operating point on the execution time versus efficiency curve is that where the incremental benefit of adding processing elements is roughly $\frac{1}{2}$ per processing element or, in other words, efficiency is 0.5. They conclude that for $T_o = \Theta(p)$, this is also equivalent to operating at a point where the ES product is maximum or $p(T_P)^2$ is minimum. This conclusion is a special case of the more general case presented by Gupta and Kumar [GK93a].

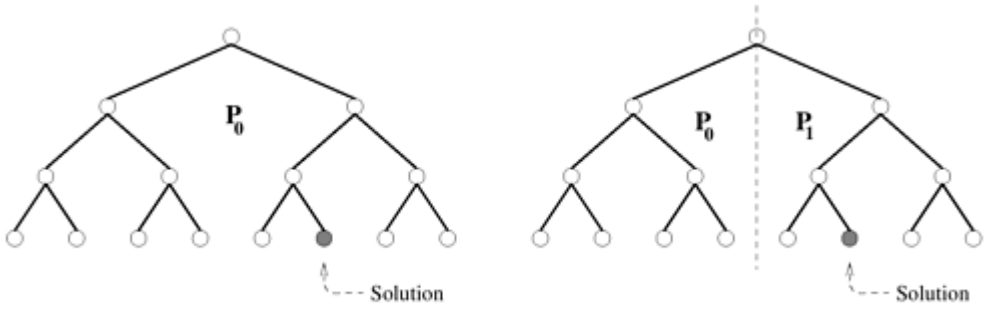
Belkhale and Banerjee [BB90], Leuze *et al.* [LDP89], Ma and Shea [MS88], and Park and Dowdy [PD89] address the important problem of optimal partitioning of the processing elements of a parallel computer among several applications of different scalabilities executing simultaneously.

Problems

5.1 (Amdahl's law [Amd67]) If a problem of size W has a serial component W_S, prove that W/W_S is an upper bound on its speedup, no matter how many processing elements are used.

5.2 (Superlinear speedup) Consider the search tree shown in Figure 5.10(a), in which the dark node represents the solution.

Figure 5.10. Superlinear(?) speedup in parallel depth first search.



(a) DFS with one processing element (b) DFS with two processing elements

a. If a sequential search of the tree is performed using the standard depth-first search (DFS) algorithm (Section 11.2.1), how much time does it take to find the solution if traversing each arc of the tree takes one unit of time?

b. Assume that the tree is partitioned between two processing elements that are assigned to do the search job, as shown in Figure 5.10(b). If both processing elements perform a DFS on their respective halves of the tree, how much time does it take for the solution to be found? What is the speedup? Is there a speedup anomaly? If so, can you explain the anomaly?

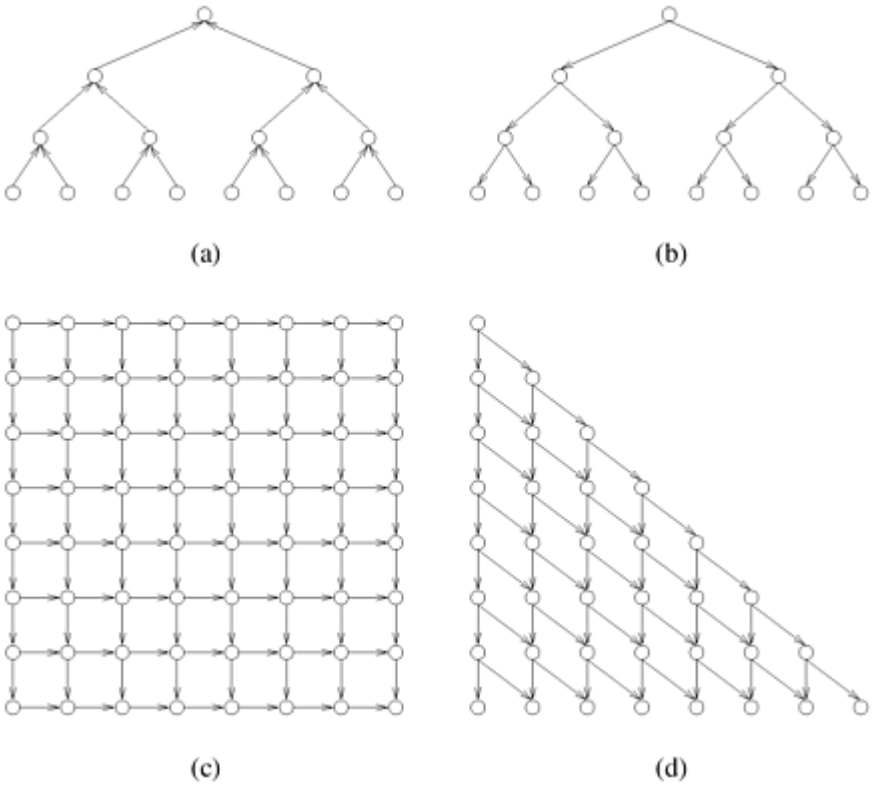
5.3 (The DAG model of parallel computation) Parallel algorithms can often be represented by dependency graphs. Four such dependency graphs are shown in Figure 5.11. If a program can be broken into several tasks, then each node of the graph represents one task. The directed edges of the graph represent the dependencies between the tasks or the order in which they must be performed to yield correct results. A node of the dependency graph can be scheduled for execution as soon as the tasks at all the nodes that have incoming edges to that node have finished execution. For example, in Figure 5.11(b), the nodes on the second level from the root can begin execution only after the task at the root is finished. Any deadlock-free dependency graph must be a *directed acyclic graph* (DAG); that is, it is devoid of any cycles. All the nodes that are scheduled for execution can be worked on in parallel provided enough processing elements are available. If N is the number of nodes in a graph, and n is an integer, then $N = 2^n - 1$ for graphs (a) and (b), $N = n^2$ for graph (c), and $N = n(n + 1)/2$ for graph (d) (graphs (a) and (b) are drawn for $n = 4$ and graphs (c) and (d) are drawn for $n = 8$). Assuming that each task takes one unit of time and that interprocessor communication time is zero, for the algorithms represented by each of these graphs:

1. Compute the degree of concurrency.

2. Compute the maximum possible speedup if an unlimited number of processing elements is available.

3. Compute the values of speedup, efficiency, and the overhead function if the number of processing elements is (i) the same as the degree of concurrency and (ii) equal to half of the degree of concurrency.

Figure 5.11. Dependency graphs for Problem 5.3.



5.4 Consider a parallel system containing p processing elements solving a problem consisting of W units of work. Prove that if the isoefficiency function of the system is worse (greater) than $\Theta(p)$, then the problem cannot be solved cost-optimally with $p = (W)$. Also prove the converse that if the problem can be solved cost-optimally only for $p < \Theta(W)$, then the isoefficiency function of the parallel system is worse than linear.

5.5 (Scaled speedup) *Scaled speedup* is defined as the speedup obtained when the problem size is increased linearly with the number of processing elements; that is, if W is chosen as a base problem size for a single processing element, then

Equation 5.37

$$\text{Scaled speedup} = \frac{pW}{T_P(pW, p)}.$$

For the problem of adding n numbers on p processing elements (Example 5.1), plot the speedup curves, assuming that the base problem for $p = 1$ is that of adding 256 numbers.

Use p = 1, 4, 16, 64, and 256. Assume that it takes 10 time units to communicate a number between two processing elements, and that it takes one unit of time to add two numbers. Now plot the standard speedup curve for the base problem size and compare it with the scaled speedup curve.

Hint: The parallel runtime is $(n/p - 1) + 11 \log p$.

5.6 Plot a third speedup curve for Problem 5.5, in which the problem size is scaled up according to the isoefficiency function, which is $\Theta(p \log p)$. Use the same expression for T_P .

Hint: The scaled speedup under this method of scaling is given by the following equation:

$$\textit{Isoefficient scaled speedup} = \frac{pW \log p}{T_P(pW \log p, p)}$$

5.7 Plot the efficiency curves for the problem of adding n numbers on p processing elements corresponding to the standard speedup curve (Problem 5.5), the scaled speedup curve (Problem 5.5), and the speedup curve when the problem size is increased according to the isoefficiency function (Problem 5.6).

5.8 A drawback of increasing the number of processing elements without increasing the total workload is that the speedup does not increase linearly with the number of processing elements, and the efficiency drops monotonically. Based on your experience with Problems 5.5 and 5.7, discuss whether or not scaled speedup increases linearly with the number of processing elements in general. What can you say about the isoefficiency function of a parallel system whose scaled speedup curve matches the speedup curve determined by increasing the problem size according to the isoefficiency function?

5.9 (Time-constrained scaling) Using the expression for T_P from Problem 5.5 for p = 1, 4, 16, 64, 256, 1024, and 4096, what is the largest problem that can be solved if the total execution time is not to exceed 512 time units? In general, is it possible to solve an arbitrarily large problem in a fixed amount of time, provided that an unlimited number of processing elements is available? Why?

5.10 (Prefix sums) Consider the problem of computing the prefix sums (Example 5.1) of n numbers on n processing elements. What is the parallel runtime, speedup, and efficiency of this algorithm? Assume that adding two numbers takes one unit of time and that communicating one number between two processing elements takes 10 units of time. Is the algorithm cost-optimal?

5.11 Design a cost-optimal version of the prefix sums algorithm (Problem 5.10) for computing all prefix-sums of n numbers on p processing elements where $p < n$. Assuming that adding two numbers takes one unit of time and that communicating one number between two processing elements takes 10 units of time, derive expressions for T_P , S, E , cost, and the isoefficiency function.

5.12 [GK93a] Prove that if $T_o \leq (p)$ for a given problem size, then the parallel execution time will continue to decrease as p is increased and will asymptotically approach a constant value. Also prove that if $T_o > \Theta(p)$, then TP first decreases and then increases with p; hence, it has a distinct minimum.

5.13 The parallel runtime of a parallel implementation of the FFT algorithm with p processing elements is given by $T_P = (n/p) \log n + t_w(n/p) \log p$ for an input sequence of

length n (Equation 13.4 with $t_s = 0$). The maximum number of processing elements that the algorithm can use for an n-point FFT is n. What are the values of p_0 (the value of p that satisfies Equation 5.21) and T_P^{min} for $t_w = 10$?

5.14 [GK93a] Consider two parallel systems with the same overhead function, but with different degrees of concurrency. Let the overhead function of both parallel systems be $W^{1/3} p^{3/2} + 0.1W^{2/3} p$. Plot the T_P versus p curve for $W = 10^6$, and $1 \leq p \leq 2048$. If the degree of concurrency is $W^{1/3}$ for the first algorithm and $W^{2/3}$ for the second algorithm, compute the values of T_P^{min} for both parallel systems. Also compute the cost and efficiency for both the parallel systems at the point on the T_P versus p curve where their respective minimum runtimes are achieved.

Chapter 6. Programming Using the Message-Passing Paradigm

Numerous programming languages and libraries have been developed for explicit parallel programming. These differ in their view of the address space that they make available to the programmer, the degree of synchronization imposed on concurrent activities, and the multiplicity of programs. The *message-passing programming paradigm* is one of the oldest and most widely used approaches for programming parallel computers. Its roots can be traced back in the early days of parallel processing and its wide-spread adoption can be attributed to the fact that it imposes minimal requirements on the underlying hardware.

In this chapter, we first describe some of the basic concepts of the message-passing programming paradigm and then explore various message-passing programming techniques using the standard and widely-used Message Passing Interface.

6.1 Principles of Message-Passing Programming

There are two key attributes that characterize the message-passing programming paradigm. The first is that it assumes a partitioned address space and the second is that it supports only explicit parallelization.

The logical view of a machine supporting the message-passing paradigm consists of *p* processes, each with its own exclusive address space. Instances of such a view come naturally from clustered workstations and non-shared address space multicomputers. There are two immediate implications of a partitioned address space. First, each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed. This adds complexity to programming, but encourages locality of access that is critical for achieving high performance on non-UMA architecture, since a processor can access its local data much faster than non-local data on such architectures. The second implication is that all interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data. This requirement for cooperation adds a great deal of complexity for a number of reasons. The process that has the data must participate in the interaction even if it has no logical connection to the events at the requesting process. In certain circumstances, this requirement leads to unnatural programs. In particular, for dynamic and/or unstructured interactions the complexity of the code written for this type of paradigm can be very high for this reason. However, a primary advantage of explicit two-way interactions is that the programmer is fully aware of all the costs of non-local interactions, and is more likely to think about algorithms (and mappings) that minimize interactions. Another major advantage of this type of programming paradigm is that it can be efficiently implemented on a wide variety of architectures.

The message-passing programming paradigm requires that the parallelism is coded explicitly by the programmer. That is, the programmer is responsible for analyzing the underlying serial algorithm/application and identifying ways by which he or she can decompose the computations and extract concurrency. As a result, programming using the message-passing paradigm tends to be hard and intellectually demanding. However, on the other hand, properly written message-passing programs can often achieve very high performance and scale to a very large number of processes.

Structure of Message-Passing Programs Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms. In the asynchronous paradigm, all concurrent tasks execute asynchronously. This makes it possible to implement any parallel algorithm. However, such programs can be harder to reason about, and can have non-deterministic behavior due to race conditions. Loosely synchronous programs are a good compromise between these two extremes. In such programs, tasks or subsets of tasks synchronize to perform interactions. However, between these interactions, tasks execute completely asynchronously. Since the interaction happens synchronously, it is still quite easy to reason about the program. Many of the known parallel algorithms can be naturally implemented using loosely synchronous programs.

In its most general form, the message-passing paradigm supports execution of a different program on each of the *p* processes. This provides the ultimate flexibility in parallel programming, but makes the job of writing parallel programs effectively unscalable. For this reason, most message-passing programs are written using the *single program multiple data* (SPMD) approach. In SPMD programs the code executed by different processes is identical except for a small number of processes (e.g., the "root" process). This does not mean that the

processes work in lock-step. In an extreme case, even in an SPMD program, each process could execute a different code (the program contains a large case statement with code for each process). But except for this degenerate case, most processes execute the same code. SPMD programs can be loosely synchronous or completely asynchronous.

PREVIOUS NEXT

6.2 The Building Blocks: Send and Receive Operations

Since interactions are accomplished by sending and receiving messages, the basic operations in the message-passing programming paradigm are `send` and `receive` . In their simplest form, the prototypes of these operations are defined as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

The `sendbuf` points to a buffer that stores the data to be sent, `recvbuf` points to a buffer that stores the data to be received, `nelems` is the number of data units to be sent and received, `dest` is the identifier of the process that receives the data, and `source` is the identifier of the process that sends the data.

However, to stop at this point would be grossly simplifying the programming and performance ramifications of how these functions are implemented. To motivate the need for further investigation, let us start with a simple example of a process sending a piece of data to another process as illustrated in the following code-fragment:

```
1          P0                                P1
2
3          a = 100;                           receive(&a, 1, 0)
4          send(&a, 1, 1);                    printf("%d\n", a);
5          a=0;
```

In this simple example, process P0 sends a message to process P1 which receives and prints the message. The important thing to note is that process P0 changes the value of `a` to 0 immediately following the `send` . The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0. That is, the value of `a` at the time of the send operation must be the value that is received by process `P1` .

It may seem that it is quite straightforward to ensure the semantics of the send and receive operations. However, based on how the send and receive operations are implemented this may not be the case. Most message passing platforms have additional hardware support for sending and receiving messages. They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware. Network interfaces allow the transfer of messages from buffer memory to desired location without CPU intervention. Similarly, DMA allows copying of data from one memory location to another (e.g., communication buffers) without CPU support (once they have been programmed). As a result, if the send operation programs the communication hardware and returns before the communication operation has been accomplished, process `P1` might receive the value 0 in `a` instead of 100!

While this is undesirable, there are in fact reasons for supporting such send operations for performance reasons. In the rest of this section, we will discuss send and receive operations in the context of such a hardware environment, and motivate various implementation details and message-passing protocols that help in ensuring the semantics of the send and receive operations.

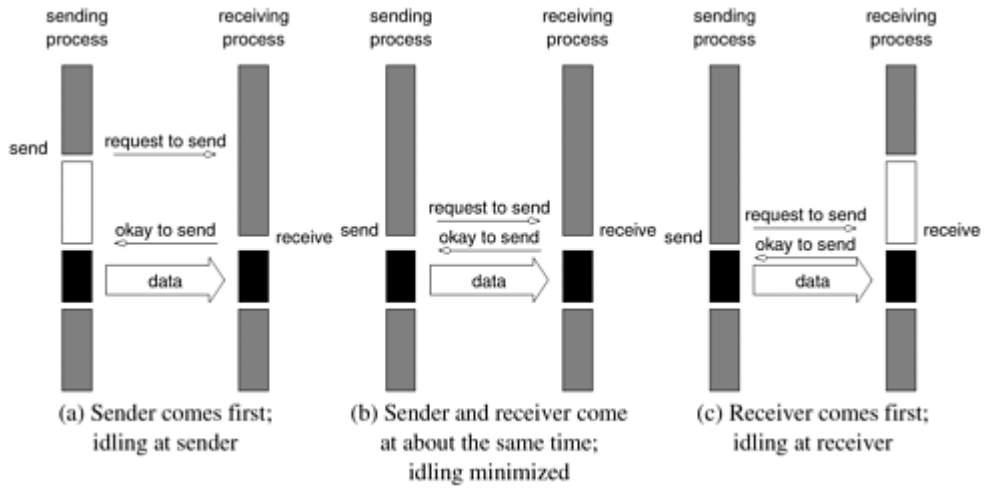
6.2.1 Blocking Message Passing Operations

A simple solution to the dilemma presented in the code fragment above is for the send operation to return only when it is semantically safe to do so. Note that this is not the same as saying that the send operation returns only after the receiver has received the data. It simply means that the sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently. There are two mechanisms by which this can be achieved.

Blocking Non-Buffered Send/Receive

In the first case, the send operation does not return until the matching receive has been encountered at the receiving process. When this happens, the message is sent and the send operation returns upon completion of the communication operation. Typically, this process involves a handshake between the sending and receiving processes. The sending process sends a request to communicate to the receiving process. When the receiving process encounters the target receive, it responds to the request. The sending process upon receiving this response initiates a transfer operation. The operation is illustrated in Figure 6.1 . Since there are no buffers used at either sending or receiving ends, this is also referred to as a *non-buffered blocking operation* .

Figure 6.1. Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.



Idling Overheads in Blocking Non-Buffered Operations In Figure 6.1 , we illustrate three scenarios in which the send is reached before the receive is posted, the send and receive are posted around the same time, and the receive is posted before the send is reached. In cases (a) and (c), we notice that there is considerable idling at the sending and receiving process. It is also clear from the figures that a blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time. However, in an asynchronous environment, this may be impossible to predict. This idling overhead is one of the major drawbacks of this protocol.

Deadlocks in Blocking Non-Buffered Operations Consider the following simple exchange of messages that can lead to a deadlock:

```
1          P0                                P1
2
3          send(&a, 1, 1);                   send(&a, 1, 0);
4          receive(&b, 1, 1);                receive(&b, 1, 0);
```

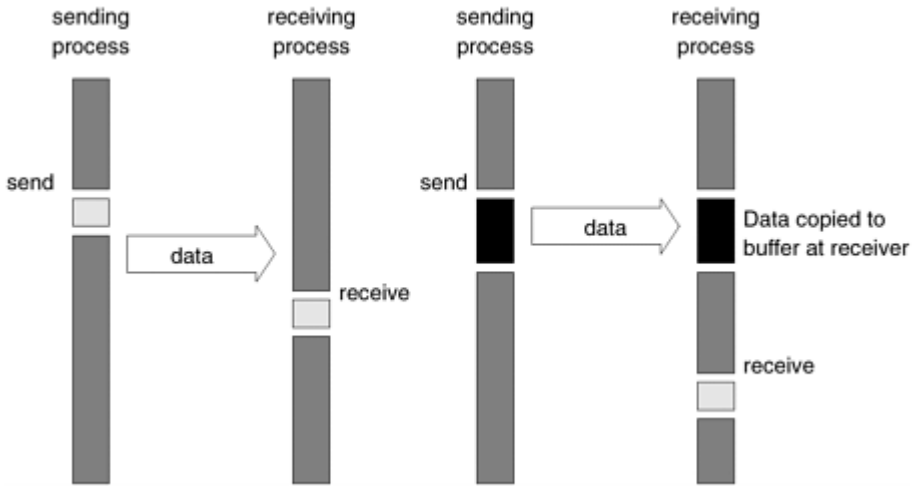
The code fragment makes the values of a available to both processes P0 and P1. However, if the send and receive operations are implemented using a blocking non-buffered protocol, the send at P0 waits for the matching receive at P1 whereas the send at process P1 waits for the corresponding receive at P0, resulting in an infinite wait.

As can be inferred, deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits of the nature outlined. In the above example, this can be corrected by replacing the operation sequence of one of the processes by a `receive` and a `send` as opposed to the other way around. This often makes the code more cumbersome and buggy.

Blocking Buffered Send/Receive

A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends. We start with a simple case in which the sender has a buffer pre-allocated for communicating messages. On encountering a send operation, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The sender process can now continue with the program knowing that any changes to the data will not impact program semantics. The actual communication can be accomplished in many ways depending on the available hardware resources. If the hardware supports asynchronous communication (independent of the CPU), then a network transfer can be initiated after the message has been copied into the buffer. Note that at the receiving end, the data cannot be stored directly at the target location since this would violate program semantics. Instead, the data is copied into a buffer at the receiver as well. When the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer. If so, the data is copied into the target location. This operation is illustrated in Figure 6.2(a) .

Figure 6.2. Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.



In the protocol illustrated above, buffers are used at both sender and receiver and communication is handled by dedicated hardware. Sometimes machines do not have such communication hardware. In this case, some of the overhead can be saved by buffering only on

one side. For example, on encountering a send operation, the sender interrupts the receiver, both processes participate in a communication operation and the message is deposited in a buffer at the receiver end. When the receiver eventually encounters a receive operation, the message is copied from the buffer into the target location. This protocol is illustrated in Figure 6.2(b) . It is not difficult to conceive a protocol in which the buffering is done only at the sender and the receiver initiates a transfer by interrupting the sender.

It is easy to see that buffered protocols alleviate idling overheads at the cost of adding buffer management overheads. In general, if the parallel program is highly synchronous (i.e., sends and receives are posted around the same time), non-buffered sends may perform better than buffered sends. However, in general applications, this is not the case and buffered sends are desirable unless buffer capacity becomes an issue.

Example 6.1 Impact of finite buffers in message passing

Consider the following code fragment:

```
1          P0                                  P1
2
3          for (i = 0; i < 1000; i++) {        for (i = 0; i < 1000; i++) {
4            produce_data(&a);                   receive(&a, 1, 0);
5            send(&a, 1, 1);                     consume_data(&a);
6          }                                   }
```

In this code fragment, process P0 produces 1000 data items and process P1 consumes them. However, if process P1 was slow getting to this loop, process P0 might have sent all of its data. If there is enough buffer space, then both processes can proceed; however, if the buffer is not sufficient (i.e., buffer overflow), the sender would have to be blocked until some of the corresponding receive operations had been posted, thus freeing up buffer space. This can often lead to unforeseen overheads and performance degradation. In general, it is a good idea to write programs that have bounded buffer requirements. ■

Deadlocks in Buffered Send and Receive Operations While buffering alleviates many of the deadlock situations, it is still possible to write code that deadlocks. This is due to the fact that as in the non-buffered case, receive calls are always blocking (to ensure semantic consistency). Thus, a simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

```
1          P0                                  P1
2
3          receive(&a, 1, 1);                  receive(&a, 1, 0);
4          send(&b, 1, 1);                     send(&b, 1, 0);
```

Once again, such circular waits have to be broken. However, deadlocks are caused only by waits on receive operations in this case.

6.2.2 Non-Blocking Message Passing Operations

In blocking protocols, the overhead of guaranteeing semantic correctness was paid in the form of idling (non-buffered) or buffer management (buffered). Often, it is possible to require the

programmer to ensure semantic correctness and provide a fast send/receive operation that incurs little overhead. This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so. Consequently, the user must be careful not to alter data that may be potentially participating in a communication operation. Non-blocking operations are generally accompanied by a `check-status` operation, which indicates whether the semantics of a previously initiated transfer may be violated or not. Upon return from a non-blocking send or receive operation, the process is free to perform any computation that does not depend upon the completion of the operation. Later in the program, the process can check whether or not the non-blocking operation has completed, and, if necessary, wait for its completion.

As illustrated in Figure 6.3 , non-blocking operations can themselves be buffered or non-buffered. In the non-buffered case, a process wishing to send data to another simply posts a pending message and returns to the user program. The program can then do other useful work. At some point in the future, when the corresponding receive is posted, the communication operation is initiated. When this operation is completed, the check-status operation indicates that it is safe for the programmer to touch this data. This transfer is indicated in Figure 6.4(a) .

Figure 6.3. Space of possible protocols for send and receive operations.

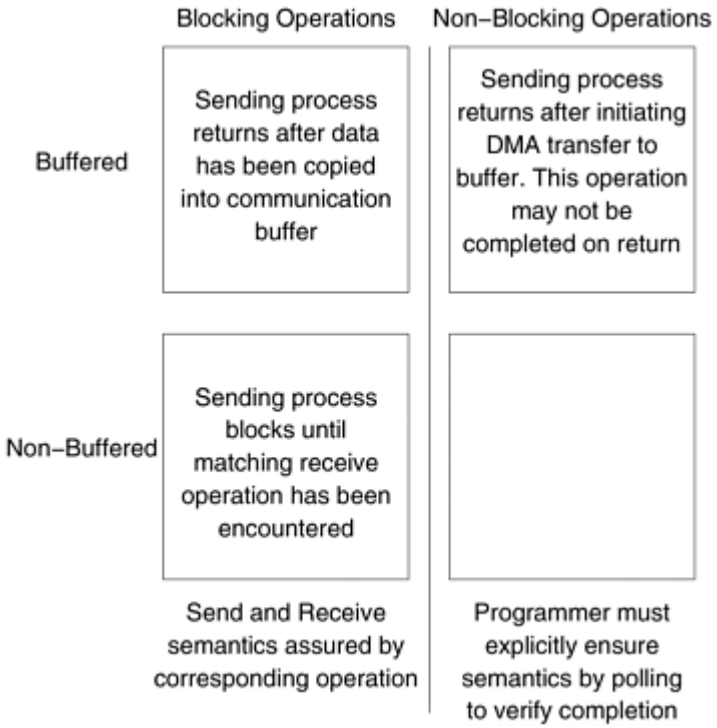
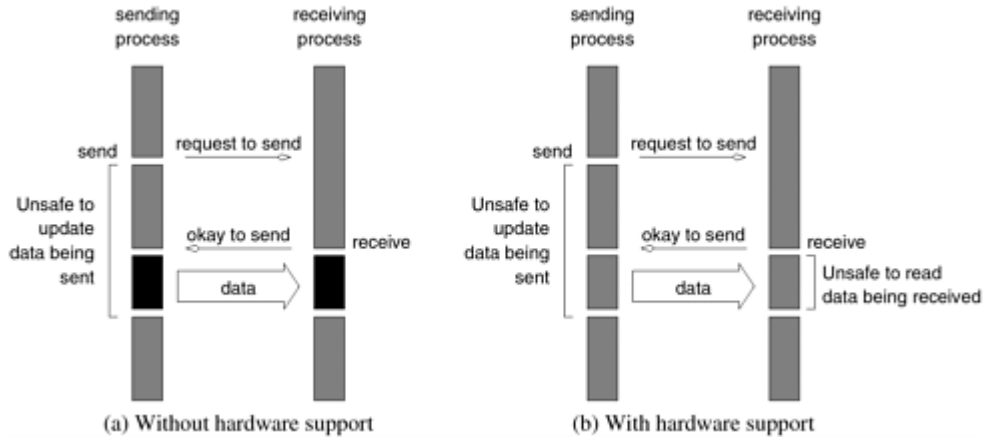


Figure 6.4. Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.



Comparing Figures 6.4(a) and 6.1(a) , it is easy to see that the idling time when the process is waiting for the corresponding receive in a blocking operation can now be utilized for computation, provided it does not update the data being sent. This alleviates the major bottleneck associated with the former at the expense of some program restructuring. The benefits of non-blocking operations are further enhanced by the presence of dedicated communication hardware. This is illustrated in Figure 6.4(b) . In this case, the communication overhead can be almost entirely masked by non-blocking operations. In this case, however, the data being received is unsafe for the duration of the receive operation.

Non-blocking operations can also be used with a buffered protocol. In this case, the sender initiates a DMA operation and returns immediately. The data becomes safe the moment the DMA operation has been completed. At the receiving end, the receive operation initiates a transfer from the sender's buffer to the receiver's target location. Using buffers with non-blocking operation has the effect of reducing the time during which the data is unsafe.

Typical message-passing libraries such as Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) implement both blocking and non-blocking operations. Blocking operations facilitate safe and easier programming and non-blocking operations are useful for performance optimization by masking communication overhead. One must, however, be careful using non-blocking protocols since errors can result from unsafe access to data that is in the process of being communicated.

PREVIOUS NEXT

6.3 MPI: the Message Passing Interface

Many early generation commercial parallel computers were based on the message-passing architecture due to its lower cost relative to shared-address-space architectures. Since message-passing is the natural programming paradigm for these machines, this resulted in the development of many different message-passing libraries. In fact, message-passing became the modern-age form of assembly language, in which every hardware vendor provided its own library, that performed very well on its own hardware, but was incompatible with the parallel computers offered by other vendors. Many of the differences between the various vendor-specific message-passing libraries were only syntactic; however, often enough there were some serious semantic differences that required significant re-engineering to port a message-passing program from one library to another.

The message-passing interface, or MPI as it is commonly known, was created to essentially solve this problem. MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran. The MPI standard defines both the syntax as well as the semantics of a core set of library routines that are very useful in writing message-passing programs. MPI was developed by a group of researchers from academia and industry, and has enjoyed wide support by almost all the hardware vendors. Vendor implementations of MPI are available on almost all commercial parallel computers.

The MPI library contains over 125 routines, but the number of key concepts is much smaller. In fact, it is possible to write fully-functional message-passing programs by using only the six routines shown in Table 6.1 . These routines are used to initialize and terminate the MPI library, to get information about the parallel computing environment, and to send and receive messages.

In this section we describe these routines as well as some basic concepts that are essential in writing correct and efficient message-passing programs using MPI.

`MPI_Init`

Initializes MPI.

`MPI_Finalize`

Terminates MPI.

`MPI_Comm_size`

Determines the number of processes.

`MPI_Comm_rank`

Determines the label of the calling process.

`MPI_Send`

Sends a message.

`MPI_Recv`

Receives a message.

Table 6.1. The minimal set of MPI routines.

6.3.1 Starting and Terminating the MPI Library

`MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment. Calling `MPI_Init` more than once during the execution of a program will lead to an error. `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment. No MPI calls may be performed after `MPI_Finalize` has been called, not even `MPI_Init` . Both `MPI_Init` and `MPI_Finalize` must be called by all the processes, otherwise MPI's behavior will be undefined. The exact calling sequences of these two routines for C are as follows:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

The arguments `argc` and `argv` of `MPI_Init` are the command-line arguments of the C program. An MPI implementation is expected to remove from the `argv` array any command-line arguments that should be processed by the implementation before returning back to the program, and to decrement `argc` accordingly. Thus, command-line processing should be performed only after `MPI_Init` has been called. Upon successful execution, `MPI_Init` and `MPI_Finalize` return `MPI_SUCCESS` ; otherwise they return an implementation-defined error code.

The bindings and calling sequences of these two functions are illustrative of the naming practices and argument conventions followed by MPI. All MPI routines, data-types, and constants are prefixed by "`MPI_` ". The return code for successful completion is `MPI_SUCCESS` . This and other MPI constants and data-structures are defined for C in the file `"mpi.h"` . This header file must be included in each MPI program.

6.3.2 Communicators

A key concept used throughout MPI is that of the *communication domain* . A communication domain is a set of processes that are allowed to communicate with each other. Information about communication domains is stored in variables of type `MPI_Comm` ,that are called *communicators* . These communicators are used as arguments to all message transfer MPI routines and they uniquely identify the processes participating in the message transfer operation. Note that each process can belong to many different (possibly overlapping) communication domains.

The communicator is used to define a set of processes that can communicate with each other. This set of processes form a *communication domain* . In general, all the processes may need to communicate with each other. For this reason, MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes involved in the parallel execution. However, in many cases we want to perform communication only within (possibly overlapping) groups of processes. By using a different communicator for each such group, we can ensure that no messages will ever interfere with messages destined to any other group. How to create and use such communicators is described at a later point in this chapter. For now, it suffices to use `MPI_COMM_WORLD` as the communicator argument to all the MPI functions that require a communicator.

6.3.3 Getting Information

The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively. The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

The function `MPI_Comm_size` returns in the variable `size` the number of processes that belong to the communicator `comm` . So, when there is a single process per processor, the call `MPI_Comm_size(MPI_COMM_WORLD, &size)` will return in `size` the number of processors used by the program. Every process that belongs to a communicator is uniquely identified by its *rank* . The rank of a process is an integer that ranges from zero up to the size of the communicator minus one. A process can determine its rank in a communicator by using the `MPI_Comm_rank` function that takes two arguments: the communicator and an integer variable `rank` . Up on return, the variable `rank` stores the rank of the process. Note that each process that calls either one of these functions must belong in the supplied communicator, otherwise an error will occur.

Example 6.2 Hello World

We can use the four MPI functions just described to write a program that prints out a "Hello World" message from each processor.

```
1   #include <mpi.h>
2
3   main(int argc, char *argv[])
4   {
5     int npes, myrank;
6
7     MPI_Init(&argc, &argv);
8      MPI_Comm_size(MPI_COMM_WORLD, &npes);
9      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10    printf("From process %d out of %d, Hello World!\n",
11            myrank, npes);
12    MPI_Finalize();
13  }
```

■

6.3.4 Sending and Receiving Messages

The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv` , respectively. The calling sequences of these routines are as follows:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
         int dest, int tag, MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
         int source, int tag, MPI_Comm comm, MPI_Status *status)
```

MPI_Send sends the data stored in the buffer pointed by buf . This buffer consists of consecutive entries of the type specified by the parameter datatype . The number of entries in the buffer is given by the parameter count . The correspondence between MPI datatypes and those provided by C is shown in Table 6.2 . Note that for all C datatypes, an equivalent MPI datatype is provided. However, MPI allows two additional datatypes that are not part of the C language. These are MPI_BYTE and MPI_PACKED .

MPI_BYTE corresponds to a byte (8 bits) and MPI_PACKED corresponds to a collection of data items that has been created by packing non-contiguous data. Note that the length of the message in MPI_Send , as well as in other MPI routines, is specified in terms of the number of entries being sent and not in terms of the number of bytes. Specifying the length in terms of the number of entries has the advantage of making the MPI code portable, since the number of bytes used to store various datatypes can be different for different architectures.

The destination of the message sent by MPI_Send is uniquely specified by the dest and comm arguments. The dest argument is the rank of the destination process in the communication domain specified by the communicator comm . Each message has an integer-valued tag associated with it. This is used to distinguish different types of messages. The message-tag can take values ranging from zero up to the MPI defined constant MPI_TAG_UB . Even though the value of MPI_TAG_UB is implementation specific, it is at least 32,767.

MPI_Recv receives a message sent by a process whose rank is given by the source in the communication domain specified by the comm argument. The tag of the sent message must be that specified by the tag argument. If there are many messages with identical tag from the same process, then any one of these messages is received. MPI allows specification of wildcard arguments for both source and tag . If source is set to MPI_ANY_SOURCE , then any process of the communication domain can be the source of the message. Similarly, if tag is set to MPI_ANY_TAG , then messages with any tag are accepted. The received message is stored in continuous locations in the buffer pointed to by buf . The count and datatype arguments of MPI_Recv are used to specify the length of the supplied buffer. The received message should be of length equal to or less than this length. This allows the receiving process to not know the exact size of the message being sent. If the received message is larger than the supplied buffer, then an overflow error will occur, and the routine will return the error MPI_ERR_TRUNCATE .

MPI_CHAR

signed char

MPI_SHORT

signed short int

MPI_INT

signed int

MPI_LONG

signed long int

MPI_UNSIGNED_CHAR

unsigned char

MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Table 6.2. Correspondence between the datatypes supported by MPI and those supported by C.

After a message has been received, the `status` variable can be used to get information about the `MPI_Recv` operation. In C, `status` is stored using the `MPI_Status` data-structure. This is implemented as a structure with three fields, as follows:

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
};
```

`MPI_SOURCE` and `MPI_TAG` store the source and the tag of the received message. They are particularly useful when `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are used for the `source` and `tag` arguments. `MPI_ERROR` stores the error-code of the received message.

The status argument also returns information about the length of the received message. This information is not directly accessible from the `status` variable, but it can be retrieved by calling the `MPI_Get_count` function. The calling sequence of this function is as follows:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
        int *count)
```

`MPI_Get_count` takes as arguments the `status` returned by `MPI_Recv` and the type of the received data in `datatype` , and returns the number of entries that were actually received in the `count` variable.

The `MPI_Recv` returns only after the requested message has been received and copied into the buffer. That is, `MPI_Recv` is a blocking receive operation. However, MPI allows two different implementations for `MPI_Send` . In the first implementation, `MPI_Send` returns only after the corresponding `MPI_Recv` have been issued and the message has been sent to the receiver. In the second implementation, `MPI_Send` first copies the message into a buffer and then returns, without waiting for the corresponding `MPI_Recv` to be executed. In either implementation, the buffer that is pointed by the `buf` argument of `MPI_Send` can be safely reused and overwritten. MPI programs must be able to run correctly regardless of which of the two methods is used for implementing `MPI_Send` . Such programs are called *safe* . In writing safe MPI programs, sometimes it is helpful to forget about the alternate implementation of `MPI_Send` and just think of it as being a blocking send operation.

Avoiding Deadlocks The semantics of `MPI_Send` and `MPI_Recv` place some restrictions on how we can mix and match send and receive operations. For example, consider the following piece of code in which process 0 sends two messages with different tags to process 1, and process 1 receives them in the reverse order.

```
1    int a[10], b[10], myrank;
2    MPI_Status status;
3    ...
4     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
5    if (myrank == 0) {
6      MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
7      MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
8    }
9    else if (myrank == 1) {
10     MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
11     MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
12   }
13   ...
```

If `MPI_Send` is implemented using buffering, then this code will run correctly provided that sufficient buffer space is available. However, if `MPI_Send` is implemented by blocking until the matching receive has been issued, then neither of the two processes will be able to proceed. This is because process zero (i.e., `myrank == 0`) will wait until process one issues the matching `MPI_Recv` (i.e., the one with `tag` equal to 1), and at the same time process one will wait until process zero performs the matching `MPI_Send` (i.e., the one with `tag` equal to 2). This code fragment is not safe, as its behavior is implementation dependent. It is up to the programmer to ensure that his or her program will run correctly on any MPI implementation. The problem in this program can be corrected by *matching the order in which the send and receive operations are issued* . Similar deadlock situations can also occur when a process sends a message to itself. Even though this is legal, its behavior is implementation dependent and must be avoided.

Improper use of `MPI_Send` and `MPI_Recv` can also lead to deadlocks in situations when each processor needs to send and receive a message in a circular fashion. Consider the following piece of code, in which process *i* sends a message to process *i* + 1 (modulo the number of processes) and receives a message from process *i* - 1 (module the number of processes).

```
1    int a[10], b[10], npes, myrank;
2    MPI_Status status;
3    ...
4     MPI_Comm_size(MPI_COMM_WORLD, &npes);
5     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
7    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
8    ...
```

When MPI_Send is implemented using buffering, the program will work correctly, since every call to MPI_Send will get buffered, allowing the call of the MPI_Recv to be performed, which will transfer the required data. However, if MPI_Send blocks until the matching receive has been issued, all processes will enter an infinite wait state, waiting for the neighboring process to issue a MPI_Recv operation. Note that the deadlock still remains even when we have only two processes. Thus, when pairs of processes need to exchange data, the above method leads to an unsafe program. The above example can be made safe, by rewriting it as follows:

```
1    int a[10], b[10], npes, myrank;
2    MPI_Status status;
3    ...
4     MPI_Comm_size(MPI_COMM_WORLD, &npes);
5     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6    if (myrank%2 == 1) {
7       MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
8       MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
9    }
10   else {
11      MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
12      MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
13   }
14   ...
```

This new implementation partitions the processes into two groups. One consists of the odd-numbered processes and the other of the even-numbered processes. The odd-numbered processes perform a send followed by a receive, and the even-numbered processes perform a receive followed by a send. Thus, when an odd-numbered process calls MPI_Send ,the target process (which has an even number) will call MPI_Recv to receive that message, before attempting to send its own message.

Sending and Receiving Messages Simultaneously The above communication pattern appears frequently in many message-passing programs, and for this reason MPI provides the MPI_Sendrecv function that both sends and receives a message.

MPI_Sendrecv does not suffer from the circular deadlock problems of MPI_Send and MPI_Recv . You can think of MPI_Sendrecv as allowing data to travel for both send and receive simultaneously. The calling sequence of MPI_Sendrecv is the following:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, int dest, int sendtag,
          void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
          int source, int recvtag, MPI_Comm comm,
          MPI_Status *status)
```

The arguments of MPI_Sendrecv are essentially the combination of the arguments of MPI_Send and MPI_Recv . The send and receive buffers must be disjoint, and the source and destination of

the messages can be the same or different. The safe version of our earlier example using MPI_Sendrecv is as follows.

```
1    int a[10], b[10], npes, myrank;
2    MPI_Status status;
3    ...
4     MPI_Comm_size(MPI_COMM_WORLD, &npes);
5     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6    MPI_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
7                  b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
8                 MPI_COMM_WORLD, &status);
9    ...
```

In many programs, the requirement for the send and receive buffers of MPI_Sendrecv be disjoint may force us to use a temporary buffer. This increases the amount of memory required by the program and also increases the overall run time due to the extra copy. This problem can be solved by using that MPI_Sendrecv_replace MPI function. This function performs a blocking send and receive, but it uses a single buffer for both the send and receive operation. That is, the received data replaces the data that was sent out of the buffer. The calling sequence of this function is the following:

```
int MPI_Sendrecv_replace(void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

Note that both the send and receive operations must transfer data of the same datatype.

6.3.5 Example: Odd-Even Sort

We will now use the MPI functions described in the previous sections to write a complete message-passing program that will sort a list of numbers using the odd-even sorting algorithm. Recall from Section 9.3.1 that the odd-even sorting algorithm sorts a sequence of n elements using p processes in a total of p phases. During each of these phases, the odd-or even-numbered processes perform a compare-split step with their right neighbors. The MPI program for performing the odd-even sort in parallel is shown in Program 6.1 . To simplify the presentation, this program assumes that n is divisible by p .

Program 6.1 Odd-Even Sorting

[View full width]

```
 1  #include <stdlib.h>
 2  #include <mpi.h> /* Include MPI's header file */
 3
 4  main(int argc, char *argv[])
 5  {
 6    int n;         /* The total number of elements to be sorted */
 7    int npes;      /* The total number of processes */
 8    int myrank;    /* The rank of the calling process */
 9    int nlocal;    /* The local number of elements, and the array that stores th
10    int *elmnts;   /* The array that stores the local elements */
11    int *relmnts;  /* The array that stores the received elements */
```

```
12    int oddrank;   /* The rank of the process during odd-phase communication */
13    int evenrank;  /* The rank of the process during even-phase communication */
14    int *wspace;   /* Working space during the compare-split operation */
15    int i;
16    MPI_Status status;
17
18    /* Initialize MPI and get system information */
19    MPI_Init(&argc, &argv);
20    MPI_Comm_size(MPI_COMM_WORLD, &npes);
21    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
22
23    n = atoi(argv[1]);
24    nlocal = n/npes; /* Compute the number of elements to be stored locally. */
25
26    /* Allocate memory for the various arrays */
27    elmnts  = (int *)malloc(nlocal*sizeof(int));
28    relmnts = (int *)malloc(nlocal*sizeof(int));
29    wspace  = (int *)malloc(nlocal*sizeof(int));
30
31    /* Fill-in the elmnts array with random elements */
32    srandom(myrank);
33    for (i=0; i<nlocal; i++)
34      elmnts[i] = random();
35
36    /* Sort the local elements using the built-in quicksort routine */
37    qsort(elmnts, nlocal, sizeof(int), IncOrder);
38
39    /* Determine the rank of the processors that myrank needs to communicate dur
➥ the */
40    /* odd and even phases of the algorithm */
41    if (myrank%2 == 0) {
42      oddrank  = myrank-1;
43      evenrank = myrank+1;
44    }
45    else {
46      oddrank  = myrank+1;
47      evenrank = myrank-1;
48    }
49
50    /* Set the ranks of the processors at the end of the linear */
51    if (oddrank == -1 || oddrank == npes)
52      oddrank = MPI_PROC_NULL;
53    if (evenrank == -1 || evenrank == npes)
54      evenrank = MPI_PROC_NULL;
55
56    /* Get into the main loop of the odd-even sorting algorithm */
57    for (i=0; i<npes-1; i++) {
58      if (i%2 == 1) /* Odd phase */
59        MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
60            nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
61      else /* Even phase */
62        MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
63            nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);
64
65      CompareSplit(nlocal, elmnts, relmnts, wspace,
```

```
66                    myrank < status.MPI_SOURCE);
67    }
68
69     free(elmnts); free(relmnts); free(wspace);
70    MPI_Finalize();
71  }
72
73  /* This is the CompareSplit function */
74   CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace,
75                int keepsmall)
76  {
77    int i, j, k;
78
79    for (i=0; i<nlocal; i++)
80      wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */
81
82    if (keepsmall) { /* Keep the nlocal smaller elements */
83      for (i=j=k=0; k<nlocal; k++) {
84        if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
85          elmnts[k] = wspace[i++];
86        else
87          elmnts[k] = relmnts[j++];
88      }
89    }
90    else { /* Keep the nlocal larger elements */
91       for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
92         if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
93           elmnts[k] = wspace[i--];
94         else
95           elmnts[k] = relmnts[j--];
96      }
97    }
98  }
99
100 /* The IncOrder function that is called by qsort is defined as follows */
101 int IncOrder(const void *e1, const void *e2)
102 {
103   return (*((int *)e1) - *((int *)e2));
104 }
```

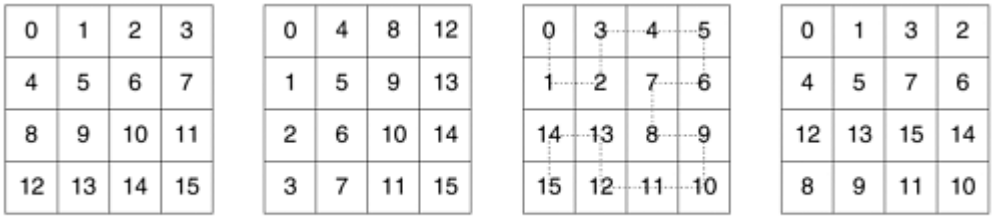

PREVIOUS NEXT

6.4 Topologies and Embedding

MPI views the processes as being arranged in a one-dimensional topology and uses a linear ordering to number the processes. However, in many parallel programs, processes are naturally arranged in higher-dimensional topologies (e.g., two- or three-dimensional). In such programs, both the computation and the set of interacting processes are naturally identified by their coordinates in that topology. For example, in a parallel program in which the processes are arranged in a two-dimensional topology, process (*i* , *j*) may need to send message to (or receive message from) process (*k* , *l*). To implement these programs in MPI, we need to map each MPI process to a process in that higher-dimensional topology.

Many such mappings are possible. Figure 6.5 illustrates some possible mappings of eight MPI processes onto a 4 x 4 two-dimensional topology. For example, for the mapping shown in Figure 6.5(a) , an MPI process with rank *rank* corresponds to process (*row* , *col*) in the grid such that *row* = *rank/4* and *col* = *rank%4* (where '%' is C's module operator). As an illustration, the process with rank 7 is mapped to process (1, 3) in the grid.

Figure 6.5. Different ways to map a set of processes to a two-dimensional grid. (a) and (b) show a row- and column-wise mapping of these processes, (c) shows a mapping that follows a space-filling curve (dotted line), and (d) shows a mapping in which neighboring processes are directly connected in a hypercube.



(a) Row-major mapping (b) Column-major mapping (c) Space-filling curve mapping (d) Hypercube mapping

In general, the goodness of a mapping is determined by the pattern of interaction among the processes in the higher-dimensional topology, the connectivity of physical processors, and the mapping of MPI processes to physical processors. For example, consider a program that uses a two-dimensional topology and each process needs to communicate with its neighboring processes along the *x* and *y* directions of this topology. Now, if the processors of the underlying parallel system are connected using a hypercube interconnection network, then the mapping shown in Figure 6.5(d) is better, since neighboring processes in the grid are also neighboring processors in the hypercube topology.

However, the mechanism used by MPI to assign ranks to the processes in a communication domain does not use any information about the interconnection network, making it impossible to perform topology embeddings in an intelligent manner. Furthermore, even if we had that information, we will need to specify different mappings for different interconnection networks, diminishing the architecture independent advantages of MPI. A better approach is to let the library itself compute the most appropriate embedding of a given topology to the processors of the underlying parallel computer. This is exactly the approach facilitated by MPI. MPI provides a set of routines that allows the programmer to arrange the processes in different topologies

without having to explicitly specify how these processes are mapped onto the processors. It is up to the MPI library to find the most appropriate mapping that reduces the cost of sending and receiving messages.

6.4.1 Creating and Using Cartesian Topologies

MPI provides routines that allow the specification of virtual process topologies of arbitrary connectivity in terms of a graph. Each node in the graph corresponds to a process and two nodes are connected if they communicate with each other. Graphs of processes can be used to specify any desired topology. However, most commonly used topologies in message-passing programs are one-, two-, or higher-dimensional grids, that are also referred to as *Cartesian topologies* . For this reason, MPI provides a set of specialized routines for specifying and manipulating this type of multi-dimensional grid topologies.

MPI's function for describing Cartesian topologies is called `MPI_Cart_create` . Its calling sequence is as follows.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
        int *periods, int reorder, MPI_Comm *comm_cart)
```

This function takes the group of processes that belong to the communicator `comm_old` and creates a virtual process topology. The topology information is attached to a new communicator `comm_cart` that is created by `MPI_Cart_create` . Any subsequent MPI routines that want to take advantage of this new Cartesian topology must use `comm_cart` as the communicator argument. Note that all the processes that belong to the `comm_old` communicator must call this function. The shape and properties of the topology are specified by the arguments `ndims` , `dims` , and `periods` . The argument `ndims` specifies the number of dimensions of the topology. The array `dims` specify the size along each dimension of the topology. The *i* th element of this array stores the size of the *i* th dimension of the topology. The array `periods` is used to specify whether or not the topology has wraparound connections. In particular, if `periods[i]` is true (non-zero in C), then the topology has wraparound connections along dimension *i* , otherwise it does not. Finally, the argument `reorder` is used to determine if the processes in the new group (i.e., communicator) are to be reordered or not. If `reorder` is false, then the rank of each process in the new group is identical to its rank in the old group. Otherwise, `MPI_Cart_create` may reorder the processes if that leads to a better embedding of the virtual topology onto the parallel computer. If the total number of processes specified in the `dims` array is smaller than the number of processes in the communicator specified by `comm_old` , then some processes will not be part of the Cartesian topology. For this set of processes, the value of `comm_cart` will be set to `MPI_COMM_NULL` (an MPI defined constant). Note that it will result in an error if the total number of processes specified by `dims` is greater than the number of processes in the `comm_old` communicator.

Process Naming When a Cartesian topology is used, each process is better identified by its coordinates in this topology. However, all MPI functions that we described for sending and receiving messages require that the source and the destination of each message be specified using the rank of the process. For this reason, MPI provides two functions, `MPI_Cart_rank` and `MPI_Cart_coord` , for performing coordinate-to-rank and rank-to-coordinate translations, respectively. The calling sequences of these routines are the following:

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims,
        int *coords)
```

The `MPI_Cart_rank` takes the coordinates of the process as argument in the `coords` array and

returns its rank in `rank` . The `MPI_Cart_coords` takes the rank of the process `rank` and returns its Cartesian coordinates in the array `coords` , of length `maxdims` . Note that `maxdims` should be at least as large as the number of dimensions in the Cartesian topology specified by the communicator `comm_cart` .

Frequently, the communication performed among processes in a Cartesian topology is that of shifting data along a dimension of the topology. MPI provides the function `MPI_Cart_shift` , that can be used to compute the rank of the source and destination processes for such operation. The calling sequence of this function is the following:

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir, int s_step,
        int *rank_source, int *rank_dest)
```

The direction of the shift is specified in the `dir` argument, and is one of the dimensions of the topology. The size of the shift step is specified in the `s_step` argument. The computed ranks are returned in `rank_source` and `rank_dest` . If the Cartesian topology was created with wraparound connections (i.e., the `periods[dir]` entry was set to true), then the shift wraps around. Otherwise, a `MPI_PROC_NULL` value is returned for `rank_source` and/or `rank_dest` for those processes that are outside the topology.

6.4.2 Example: Cannon's Matrix-Matrix Multiplication

To illustrate how the various topology functions are used we will implement Cannon's algorithm for multiplying two matrices *A* and *B* , described in Section 8.2.2 . Cannon's algorithm views the processes as being arranged in a virtual two-dimensional square array. It uses this array to distribute the matrices *A* , *B* , and the result matrix *C* in a block fashion. That is, if *n* x *n* is the size of each matrix and *p* is the total number of process, then each matrix is divided into square blocks of size $n/\sqrt{p} \times n/\sqrt{p}$ (assuming that *p* is a perfect square). Now, process $P_{i,j}$ in the grid is assigned the $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$ blocks of each matrix. After an initial data alignment phase, the algorithm proceeds in $\sqrt{p}$ steps. In each step, every process multiplies the locally available blocks of matrices *A* and *B* , and then sends the block of *A* to the leftward process, and the block of *B* to the upward process.

Program 6.2 shows the MPI function that implements Cannon's algorithm. The dimension of the matrices is supplied in the parameter `n` . The parameters `a` , `b` , and `c` point to the locally stored portions of the matrices *A* , *B* , and *C* , respectively. The size of these arrays is $n/\sqrt{p} \times n/\sqrt{p}$, where *p* is the number of processes. This routine assumes that *p* is a perfect square and that *n* is a multiple of $\sqrt{p}$. The parameter `comm` stores the communicator describing the processes that call the `MatrixMatrixMultiply` function. Note that the remaining programs in this chapter will be provided in the form of a function, as opposed to complete stand-alone programs.

Program 6.2 Cannon's Matrix-Matrix Multiplication with MPI's Topologies

```
1  MatrixMatrixMultiply(int n, double *a, double *b, double *c,
2                       MPI_Comm comm)
3  {
4    int i;
5    int nlocal;
6    int npes, dims[2], periods[2];
7    int myrank, my2drank, mycoords[2];
8    int uprank, downrank, leftrank, rightrank, coords[2];
```

```
 9     int shiftsource, shiftdest;
10     MPI_Status status;
11     MPI_Comm comm_2d;
12
13     /* Get the communicator related information */
14     MPI_Comm_size(comm, &npes);
15     MPI_Comm_rank(comm, &myrank);
16
17     /* Set up the Cartesian topology */
18     dims[0] = dims[1] = sqrt(npes);
19
20     /* Set the periods for wraparound connections */
21     periods[0] = periods[1] = 1;
22
23     /* Create the Cartesian topology, with rank reordering */
24     MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
25
26     /* Get the rank and coordinates with respect to the new topology */
27     MPI_Comm_rank(comm_2d, &my2drank);
28     MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
29
30     /* Compute ranks of the up and left shifts */
31     MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
32     MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
33
34     /* Determine the dimension of the local matrix block */
35     nlocal = n/dims[0];
36
37     /* Perform the initial matrix alignment. First for A and then for B */
38     MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
39     MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE, shiftdest,
40         1, shiftsource, 1, comm_2d, &status);
41
42     MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
43     MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
44         shiftdest, 1, shiftsource, 1, comm_2d, &status);
45
46     /* Get into the main computation loop */
47     for (i=0; i<dims[0]; i++) {
48       MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/
49
50       /* Shift matrix a left by one */
51       MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
52           leftrank, 1, rightrank, 1, comm_2d, &status);
53
54       /* Shift matrix b up by one */
55       MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
56           uprank, 1, downrank, 1, comm_2d, &status);
57     }
58
59     /* Restore the original distribution of a and b */
60     MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
61     MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
62         shiftdest, 1, shiftsource, 1, comm_2d, &status);
63
```

```
64      MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
65      MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
66          shiftdest, 1, shiftsource, 1, comm_2d, &status);
67
68      MPI_Comm_free(&comm_2d); /* Free up communicator */
69    }
70
71    /* This function performs a serial matrix-matrix multiplication c = a*b */
72    MatrixMultiply(int n, double *a, double *b, double *c)
73    {
74      int i, j, k;
75
76      for (i=0; i<n; i++)
77        for (j=0; j<n; j++)
78          for (k=0; k<n; k++)
79            c[i*n+j] += a[i*n+k]*b[k*n+j];
80    }
```


PREVIOUS NEXT

6.5 Overlapping Communication with Computation

The MPI programs we developed so far used blocking send and receive operations whenever they needed to perform point-to-point communication. Recall that a blocking send operation remains blocked until the message has been copied out of the send buffer (either into a system buffer at the source process or sent to the destination process). Similarly, a blocking receive operation returns only after the message has been received and copied into the receive buffer. For example, consider Cannon's matrix-matrix multiplication program described in Program 6.2. During each iteration of its main computational loop (lines 47– 57), it first computes the matrix multiplication of the sub-matrices stored in `a` and `b`, and then shifts the blocks of `a` and `b`, using `MPI_Sendrecv_replace` which blocks until the specified matrix block has been sent and received by the corresponding processes. In each iteration, each process spends $O(n^3/p^{1.5})$ time for performing the matrix-matrix multiplication and $O(n^2/p)$ time for shifting the blocks of matrices A and B. Now, since the blocks of matrices A and B do not change as they are shifted among the processors, it will be preferable if we can overlap the transmission of these blocks with the computation for the matrix-matrix multiplication, as many recent distributed-memory parallel computers have dedicated communication controllers that can perform the transmission of messages without interrupting the CPUs.

6.5.1 Non-Blocking Communication Operations

In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations. These functions are `MPI_Isend` and `MPI_Irecv`. `MPI_Isend` starts a send operation but does not complete, that is, it returns before the data is copied out of the buffer. Similarly, `MPI_Irecv` starts a receive operation but returns before the data has been received and copied into the buffer. With the support of appropriate hardware, the transmission and reception of messages can proceed concurrently with the computations performed by the program upon the return of the above functions.

However, at a later point in the program, a process that has started a non-blocking send or receive operation must make sure that this operation has completed before it proceeds with its computations. This is because a process that has started a non-blocking send operation may want to overwrite the buffer that stores the data that are being sent, or a process that has started a non-blocking receive operation may want to use the data it requested. To check the completion of non-blocking send and receive operations, MPI provides a pair of functions `MPI_Test` and `MPI_Wait`. The first tests whether or not a non-blocking operation has finished and the second waits (i.e., gets blocked) until a non-blocking operation actually finishes.

The calling sequences of `MPI_Isend` and `MPI_Irecv` are the following:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm comm, MPI_Request *request)
```

Note that these functions have similar arguments as the corresponding blocking send and receive functions. The main difference is that they take an additional argument `request`. `MPI_Isend` and `MPI_Irecv` functions allocate a *request object* and return a pointer to it in the `request` variable. This request object is used as an argument in the `MPI_Test` and `MPI_Wait`

functions to identify the operation whose status we want to query or to wait for its completion.

Note that the MPI_Irecv function does not take a status argument similar to the blocking receive function, but the status information associated with the receive operation is returned by the MPI_Test and MPI_Wait functions.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

MPI_Test tests whether or not the non-blocking send or receive operation identified by its request has finished. It returns flag = {true} (non-zero value in C) if it completed, otherwise it returns {false} (a zero value in C). In the case that the non-blocking operation has finished, the request object pointed to by request is deallocated and request is set to MPI_REQUEST_NULL. Also the status object is set to contain information about the operation. If the operation has not finished, request is not modified and the value of the status object is undefined. The MPI_Wait function blocks until the non-blocking operation identified by request completes. In that case it deal-locates the request object, sets it to MPI_REQUEST_NULL, and returns information about the completed operation in the status object.

For the cases that the programmer wants to explicitly deallocate a request object, MPI provides the following function.

```
int MPI_Request_free(MPI_Request *request)
```

Note that the deallocation of the request object does not have any effect on the associated non-blocking send or receive operation. That is, if it has not yet completed it will proceed until its completion. Hence, one must be careful before explicitly deallocating a request object, since without it, we cannot check whether or not the non-blocking operation has completed.

A non-blocking communication operation can be matched with a corresponding blocking operation. For example, a process can send a message using a non-blocking send operation and this message can be received by the other process using a blocking receive operation.

Avoiding Deadlocks By using non-blocking communication operations we can remove most of the deadlocks associated with their blocking counterparts. For example, as we discussed in Section 6.3 the following piece of code is not safe.

```
 1   int a[10], b[10], myrank;
 2   MPI_Status status;
 3   ...
 4    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 5   if (myrank == 0) {
 6     MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
 7     MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
 8   }
 9   else if (myrank == 1) {
10     MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
11     MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
12   }
13   ...
```

However, if we replace either the send or receive operations with their non-blocking counterparts, then the code will be safe, and will correctly run on any MPI implementation.

```
 1   int a[10], b[10], myrank;
 2   MPI_Status status;
 3   MPI_Request requests[2];
 4   ...
 5    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 6   if (myrank == 0) {
 7     MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
 8     MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
 9   }
10   else if (myrank == 1) {
11     MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
12     MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
13   }
14   ...
```

This example also illustrates that the non-blocking operations started by any process can finish in any order depending on the transmission or reception of the corresponding messages. For example, the second receive operation will finish before the first does.

Example: Cannon's Matrix-Matrix Multiplication (Using Non-Blocking Operations)

Program 6.3 shows the MPI program that implements Cannon's algorithm using non-blocking send and receive operations. The various parameters are identical to those of Program 6.2.

Program 6.3 Non-Blocking Cannon's Matrix-Matrix Multiplication

```
 1   MatrixMatrixMultiply_NonBlocking(int n, double *a, double *b,
 2                                    double *c, MPI_Comm comm)
 3   {
 4     int i, j, nlocal;
 5     double *a_buffers[2], *b_buffers[2];
 6     int npes, dims[2], periods[2];
 7     int myrank, my2drank, mycoords[2];
 8     int uprank, downrank, leftrank, rightrank, coords[2];
 9     int shiftsource, shiftdest;
10     MPI_Status status;
11     MPI_Comm comm_2d;
12     MPI_Request reqs[4];
13
14     /* Get the communicator related information */
15     MPI_Comm_size(comm, &npes);
16     MPI_Comm_rank(comm, &myrank);
17
18     /* Set up the Cartesian topology */
19     dims[0] = dims[1] = sqrt(npes);
20
21     /* Set the periods for wraparound connections */
22     periods[0] = periods[1] = 1;
23
24     /* Create the Cartesian topology, with rank reordering */
25     MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
26
27     /* Get the rank and coordinates with respect to the new topology */
```

```
28      MPI_Comm_rank(comm_2d, &my2drank);
29      MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
30
31     /* Compute ranks of the up and left shifts */
32      MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
33      MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
34
35     /* Determine the dimension of the local matrix block */
36     nlocal = n/dims[0];
37
38     /* Setup the a_buffers and b_buffers arrays */
39     a_buffers[0] = a;
40      a_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
41     b_buffers[0] = b;
42      b_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
43
44     /* Perform the initial matrix alignment. First for A and then for B */
45      MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
46      MPI_Sendrecv_replace(a_buffers[0], nlocal*nlocal, MPI_DOUBLE,
47          shiftdest, 1, shiftsource, 1, comm_2d, &status);
48
49      MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
50      MPI_Sendrecv_replace(b_buffers[0], nlocal*nlocal, MPI_DOUBLE,
51          shiftdest, 1, shiftsource, 1, comm_2d, &status);
52
53     /* Get into the main computation loop */
54     for (i=0; i<dims[0]; i++) {
55        MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
56            leftrank, 1, comm_2d, &reqs[0]);
57        MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
58            uprank, 1, comm_2d, &reqs[1]);
59        MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
60            rightrank, 1, comm_2d, &reqs[2]);
61        MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
62            downrank, 1, comm_2d, &reqs[3]);
63
64       /* c = c + a*b */
65        MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);
66
67       for (j=0; j<4; j++)
68         MPI_Wait(&reqs[j], &status);
69     }
70
71     /* Restore the original distribution of a and b */
72      MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
73      MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
74          shiftdest, 1, shiftsource, 1, comm_2d, &status);
75
76      MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
77      MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
78          shiftdest, 1, shiftsource, 1, comm_2d, &status);
79
80      MPI_Comm_free(&comm_2d); /* Free up communicator */
81
82      free(a_buffers[1]);
```

```
83      free(b_buffers[1]);
84  }
```

There are two main differences between the blocking program (Program 6.2) and this non-blocking one. The first difference is that the non-blocking program requires the use of the additional arrays *a_buffers* and *b_buffers*, that are used as the buffer of the blocks of *A* and *B* that are being received while the computation involving the previous blocks is performed. The second difference is that in the main computational loop, it first starts the non-blocking send operations to send the locally stored blocks of *A* and *B* to the processes left and up the grid, and then starts the non-blocking receive operations to receive the blocks for the next iteration from the processes right and down the grid. Having initiated these four non-blocking operations, it proceeds to perform the matrix-matrix multiplication of the blocks it currently stores. Finally, before it proceeds to the next iteration, it uses `MPI_Wait` to wait for the send and receive operations to complete.

Note that in order to overlap communication with computation we have to use two auxiliary arrays – one for *A* and one for *B*. This is to ensure that incoming messages never overwrite the blocks of *A* and *B* that are used in the computation, which proceeds concurrently with the data transfer. Thus, increased performance (by overlapping communication with computation) comes at the expense of increased memory requirements. This is a trade-off that is often made in message-passing programs, since communication overheads can be quite high for loosely coupled distributed memory parallel computers.

6.6 Collective Communication and Computation Operations

MPI provides an extensive set of functions for performing many commonly used collective communication operations. In particular, the majority of the basic communication operations described in Chapter 4 are supported by MPI. All of the collective communication functions provided by MPI take as an argument a communicator that defines the group of processes that participate in the collective operation. All the processes that belong to this communicator participate in the operation, and all of them must call the collective communication function. Even though collective communication operations do not act like barriers (i.e., it is possible for a processor to go past its call for the collective communication operation even before other processes have reached it), it acts like a *virtual* synchronization step in the following sense: the parallel program should be written such that it behaves correctly even if a global synchronization is performed before and after the collective call. Since the operations are virtually synchronous, they do not require tags. In some of the collective functions data is required to be sent from a single process (source-process) or to be received by a single process (target-process). In these functions, the source- or target-process is one of the arguments supplied to the routines. All the processes in the group (i.e., communicator) must specify the same source- or target-process. For most collective communication operations, MPI provides two different variants. The first transfers equal-size data to or from each process, and the second transfers data that can be of different sizes.

6.6.1 Barrier

The barrier synchronization operation is performed in MPI using the `MPI_Barrier` function.

```
int MPI_Barrier(MPI_Comm comm)
```

The only argument of `MPI_Barrier` is the communicator that defines the group of processes that are synchronized. The call to `MPI_Barrier` returns only after all the processes in the group have called this function.

6.6.2 Broadcast

The one-to-all broadcast operation described in Section 4.1 is performed in MPI using the `MPI_Bcast` function.

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
        int source, MPI_Comm comm)
```

`MPI_Bcast` sends the data stored in the buffer `buf` of process `source` to all the other processes in the group. The data received by each process is stored in the buffer `buf` . The data that is broadcast consist of `count` entries of type `datatype` . The amount of data sent by the `source` process must be equal to the amount of data that is being received by each process; i.e., the `count` and `datatype` fields must match on all processes.

6.6.3 Reduction

The all-to-one reduction operation described in Section 4.1 is performed in MPI using the `MPI_Reduce` function.

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
         MPI_Datatype datatype, MPI_Op op, int target,
         MPI_Comm comm)
```

`MPI_Reduce` combines the elements stored in the buffer `sendbuf` of each process in the group, using the operation specified in `op` , and returns the combined values in the buffer `recvbuf` of the process with rank `target` . Both the `sendbuf` and `recvbuf` must have the same number of `count` items of type `datatype` . Note that all processes must provide a `recvbuf` array, even if they are not the *target* of the reduction operation. When `count` is more than one, then the combine operation is applied element-wise on each entry of the sequence. All the processes must call `MPI_Reduce` with the same value for `count` , `datatype` , `op` , `target` , and `comm` .

MPI provides a list of predefined operations that can be used to combine the elements stored in `sendbuf` . MPI also allows programmers to define their own operations, which is not covered in this book. The predefined operations are shown in Table 6.3 . For example, in order to compute the maximum of the elements stored in `sendbuf` , the `MPI_MAX` value must be used for the `op` argument. Not all of these operations can be applied to all possible data-types supported by MPI. For example, a bit-wise OR operation (i.e., *op* = `MPI_BOR`) is not defined for real-valued data-types such as `MPI_FLOAT` and `MPI_REAL` . The last column of Table 6.3 shows the various data-types that can be used with each operation.

`MPI_MAX`

Maximum

C integers and floating point

`MPI_MIN`

Minimum

C integers and floating point

`MPI_SUM`

Sum

C integers and floating point

`MPI_PROD`

Product

C integers and floating point

`MPI_LAND`

Logical AND

C integers

MPI_BAND

Bit-wise AND

C integers and byte

MPI_LOR

Logical OR

C integers

MPI_BOR

Bit-wise OR

C integers and byte

MPI_LXOR

Logical XOR

C integers

MPI_BXOR

Bit-wise XOR

C integers and byte

MPI_MAXLOC

max-min value-location

Data-pairs

MPI_MINLOC

min-min value-location

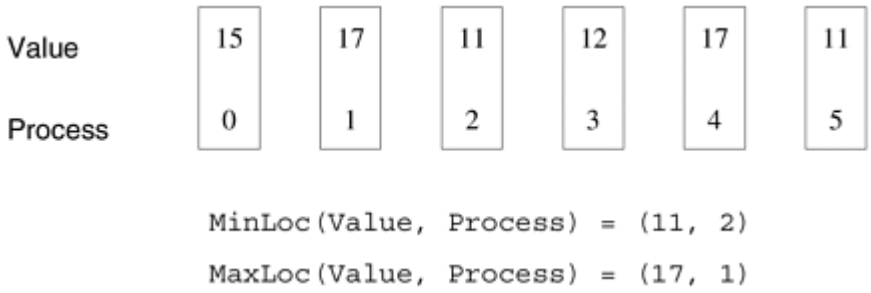
Data-pairs

Table 6.3. Predefined reduction operations.

Operation	Meaning	Datatypes

The operation MPI_MAXLOC combines pairs of values (v_i , l_i) and returns the pair (v , l) such that v is the maximum among all v_i 's and l is the smallest among all l_i 's such that $v = v_i$. Similarly, MPI_MINLOC combines pairs of values and returns the pair (v , l) such that v is the minimum among all v_i 's and l is the smallest among all l_i 's such that $v = v_i$. One possible application of MPI_MAXLOC or MPI_MINLOC is to compute the maximum or minimum of a list of numbers each residing on a different process and also the rank of the first process that stores this maximum or minimum, as illustrated in Figure 6.6 . Since both MPI_MAXLOC and MPI_MINLOC require datatypes that correspond to pairs of values, a new set of MPI datatypes have been defined as shown in Table 6.4 . In C, these datatypes are implemented as structures containing the corresponding types.

Figure 6.6. An example use of the **MPI_MINLOC** and **MPI_MAXLOC** operators.



When the result of the reduction operation is needed by all the processes, MPI provides the `MPI_Allreduce` operation that returns the result to all the processes. This function provides the functionality of the all-reduce operation described in Section 4.3 .

`MPI_2INT`

pair of `int` s

`MPI_SHORT_INT`

`short` and `int`

`MPI_LONG_INT`

`long` and `int`

`MPI_LONG_DOUBLE_INT`

`long double` and `int`

`MPI_FLOAT_INT`

`float` and `int`

`MPI_DOUBLE_INT`

`double` and `int`

Table 6.4. MPI datatypes for data-pairs used with the **MPI_MAXLOC** and **MPI_MINLOC** reduction operations.

MPI Datatype	C Datatype

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Note that there is no `target` argument since all processes receive the result of the operation.

6.6.4 Prefix

The prefix-sum operation described in Section 4.3 is performed in MPI using the `MPI_Scan` function.

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

MPI_Scan performs a prefix reduction of the data stored in the buffer sendbuf at each process and returns the result in the buffer recvbuf . The receive buffer of the process with rank *i* will store, at the end of the operation, the reduction of the send buffers of the processes whose ranks range from 0 up to and including *i* . The type of supported operations (i.e., op) as well as the restrictions on the various arguments of MPI_Scan are the same as those for the reduction operation MPI_Reduce .

6.6.5 Gather

The gather operation described in Section 4.4 is performed in MPI using the MPI_Gather function.

```
int MPI_Gather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int target, MPI_Comm comm)
```

Each process, including the target process, sends the data stored in the array sendbuf to the target process. As a result, if *p* is the number of processors in the communication comm , the target process receives a total of *p* buffers. The data is stored in the array recvbuf of the target process, in a rank order. That is, the data from process with rank *i* are stored in the recvbuf starting at location *i* * sendcount (assuming that the array recvbuf is of the same type as recvdatatype).

The data sent by each process must be of the same size and type. That is, MPI_Gather must be called with the sendcount and senddatatype arguments having the same values at each process. The information about the receive buffer, its length and type applies only for the target process and is ignored for all the other processes. The argument recvcount specifies the number of elements received by each process and not the total number of elements it receives. So, recvcount must be the same as sendcount and their datatypes must be matching.

MPI also provides the MPI_Allgather function in which the data are gathered to all the processes and not only at the target process.

```
int MPI_Allgather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

The meanings of the various parameters are similar to those for MPI_Gather ; however, each process must now supply a recvbuf array that will store the gathered data.

In addition to the above versions of the gather operation, in which the sizes of the arrays sent by each process are the same, MPI also provides versions in which the size of the arrays can be different. MPI refers to these operations as the *vector* variants. The vector variants of the MPI_Gather and MPI_Allgather operations are provided by the functions MPI_Gatherv and MPI_Allgatherv , respectively.

```
int MPI_Gatherv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int *recvcounts, int *displs,
```

```
        MPI_Datatype recvdatatype, int target, MPI_Comm comm)

int MPI_Allgatherv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int *recvcounts, int *displs, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```

These functions allow a different number of data elements to be sent by each process by replacing the `recvcount` parameter with the array `recvcounts` . The amount of data sent by process *i* is equal to `recvcounts[i]` . Note that the size of `recvcounts` is equal to the size of the communicator `comm` . The array parameter `displs` , which is also of the same size, is used to determine where in `recvbuf` the data sent by each process will be stored. In particular, the data sent by process *i* are stored in `recvbuf` starting at location `displs[i]` . Note that, as opposed to the non-vector variants, the `sendcount` parameter can be different for different processes.

6.6.6 Scatter

The scatter operation described in Section 4.4 is performed in MPI using the `MPI_Scatter` function.

```
int MPI_Scatter(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int source, MPI_Comm comm)
```

The `source` process sends a different part of the send buffer `sendbuf` to each processes, including itself. The data that are received are stored in `recvbuf` . Process *i* receives `sendcount` contiguous elements of type `senddatatype` starting from the *i* * `sendcount` location of the `sendbuf` of the source process (assuming that `sendbuf` is of the same type as `senddatatype`). `MPI_Scatter` must be called by all the processes with the same values for the `sendcount` , `senddatatype` , `recvcount` , `recvdatatype` , `source` , and `comm` arguments. Note again that `sendcount` is the number of elements sent to each individual process.

Similarly to the gather operation, MPI provides a vector variant of the scatter operation, called `MPI_Scatterv` , that allows different amounts of data to be sent to different processes.

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int source, MPI_Comm comm)
```

As we can see, the parameter `sendcount` has been replaced by the array `sendcounts` that determines the number of elements to be sent to each process. In particular, the `target` process sends `sendcounts[i]` elements to process *i* . Also, the array `displs` is used to determine where in `sendbuf` these elements will be sent from. In particular, if `sendbuf` is of the same type is `senddatatype` , the data sent to process *i* start at location `displs[i]` of array `sendbuf` . Both the `sendcounts` and `displs` arrays are of size equal to the number of processes in the communicator. Note that by appropriately setting the `displs` array we can use `MPI_Scatterv` to send overlapping regions of `sendbuf` .

6.6.7 All-to-All

The all-to-all personalized communication operation described in Section 4.5 is performed in MPI by using the `MPI_Alltoall` function.

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

Each process sends a different portion of the `sendbuf` array to each other process, including itself. Each process sends to process *i* `sendcount` contiguous elements of type `senddatatype` starting from the *i* * `sendcount` location of its `sendbuf` array. The data that are received are stored in the `recvbuf` array. Each process receives from process *i* `recvcount` elements of type `recvdatatype` and stores them in its `recvbuf` array starting at location *i* * `recvcount` . `MPI_Alltoall` must be called by all the processes with the same values for the `sendcount` , `senddatatype` , `recvcount` , `recvdatatype` , and `comm` arguments. Note that `sendcount` and `recvcount` are the number of elements sent to, and received from, each individual process.

MPI also provides a vector variant of the all-to-all personalized communication operation called `MPI_Alltoallv` that allows different amounts of data to be sent to and received from each process.

```
int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls
        MPI_Datatype senddatatype, void *recvbuf, int *recvcounts,
        int *rdispls, MPI_Datatype recvdatatype, MPI_Comm comm)
```

The parameter `sendcounts` is used to specify the number of elements sent to each process, and the parameter `sdispls` is used to specify the location in `sendbuf` in which these elements are stored. In particular, each process sends to process *i* , starting at location `sdispls[i]` of the array `sendbuf` , `sendcounts[i]` contiguous elements. The parameter `recvcounts` is used to specify the number of elements received by each process, and the parameter `rdispls` is used to specify the location in `recvbuf` in which these elements are stored. In particular, each process receives from process *i* `recvcounts[i]` elements that are stored in contiguous locations of `recvbuf` starting at location `rdispls[i]` . `MPI_Alltoallv` must be called by all the processes with the same values for the `senddatatype` , `recvdatatype` , and `comm` arguments.

6.6.8 Example: One-Dimensional Matrix-Vector Multiplication

Our first message-passing program using collective communications will be to multiply a dense *n* x *n* matrix *A* with a vector *b* , i.e., *x* = *Ab* . As discussed in Section 8.1 , one way of performing this multiplication in parallel is to have each process compute different portions of the product-vector *x* . In particular, each one of the *p* processes is responsible for computing *n* /*p* consecutive elements of *x* . This algorithm can be implemented in MPI by distributing the matrix *A* in a row-wise fashion, such that each process receives the *n* /*p* rows that correspond to the portion of the product-vector *x* it computes. Vector *b* is distributed in a fashion similar to *x* .

Program 6.4 shows the MPI program that uses a row-wise distribution of matrix *A* . The dimension of the matrices is supplied in the parameter `n` , the parameters `a` and `b` point to the locally stored portions of matrix *A* and vector *b* , respectively, and the parameter `x` points to the local portion of the output matrix-vector product. This program assumes that *n* is a multiple of the number of processors.

Program 6.4 Row-wise Matrix-Vector Multiplication

```
1   RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
2                           MPI_Comm comm)
```

```
{
  int i, j;
  int nlocal;         /* Number of locally stored rows of A */
  double *fb;         /* Will point to a buffer that stores the entire vector b
  int npes, myrank;
  MPI_Status status;

  /* Get information about the communicator */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Allocate the memory that will store the entire vector b */
  fb = (double *)malloc(n*sizeof(double));

  nlocal = n/npes;

  /* Gather the entire vector b on each processor using MPI's ALLGATHER operat
  MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
      comm);

  /* Perform the matrix-vector multiplication involving the locally stored sub
  for (i=0; i<nlocal; i++) {
    x[i] = 0.0;
    for (j=0; j<n; j++)
      x[i] += a[i*n+j]*fb[j];
  }

  free(fb);
}
```

An alternate way of computing x is to parallelize the task of performing the dot-product for each element of x . That is, for each element x_i , of vector x , all the processes will compute a part of it, and the result will be obtained by adding up these partial dot-products. This algorithm can be implemented in MPI by distributing matrix A in a column-wise fashion. Each process gets n/p consecutive columns of A , and the elements of vector b that correspond to these columns. Furthermore, at the end of the computation we want the product-vector x to be distributed in a fashion similar to vector b . Program 6.5 shows the MPI program that implements this column-wise distribution of the matrix.

Program 6.5 Column-wise Matrix-Vector Multiplication

```
ColMatrixVectorMultiply(int n, double *a, double *b, double *x,
                        MPI_Comm comm)
{
  int i, j;
  int nlocal;
  double *px;
  double *fx;
  int npes, myrank;
  MPI_Status status;

  /* Get identity and size information from the communicator */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);
```

```
14
15      nlocal = n/npes;
16
17      /* Allocate memory for arrays storing intermediate results. */
18      px = (double *)malloc(n*sizeof(double));
19      fx = (double *)malloc(n*sizeof(double));
20
21      /* Compute the partial-dot products that correspond to the local columns of
22      for (i=0; i<n; i++) {
23        px[i] = 0.0;
24        for (j=0; j<nlocal; j++)
25          px[i] += a[i*nlocal+j]*b[j];
26      }
27
28      /* Sum-up the results by performing an element-wise reduction operation */
29      MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);
30
31      /* Redistribute fx in a fashion similar to that of vector b */
32      MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0,
33          comm);
34
35      free(px); free(fx);
36    }
```

Comparing these two programs for performing matrix-vector multiplication we see that the row-wise version needs to perform only a `MPI_Allgather` operation whereas the column-wise program needs to perform a `MPI_Reduce` and a `MPI_Scatter` operation. In general, a row-wise distribution is preferable as it leads to small communication overhead (see Problem 6.6). However, many times, an application needs to compute not only Ax but also $A^T x$. In that case, the row-wise distribution can be used to compute Ax , but the computation of $A^T x$ requires the column-wise distribution (a row-wise distribution of A is a column-wise distribution of its transpose A^T). It is much cheaper to use the program for the column-wise distribution than to transpose the matrix and then use the row-wise program. We must also note that using a dual of the all-gather operation, it is possible to develop a parallel formulation for column-wise distribution that is as fast as the program using row-wise distribution (see Problem 6.7). However, this dual operation is not available in MPI.

6.6.9 Example: Single-Source Shortest-Path

Our second message-passing program that uses collective communication operations computes the shortest paths from a source-vertex s to all the other vertices in a graph using Dijkstra's single-source shortest-path algorithm described in Section 10.3 . This program is shown in Program 6.6.

The parameter `n` stores the total number of vertices in the graph, and the parameter `source` stores the vertex from which we want to compute the single-source shortest path. The parameter `wgt` points to the locally stored portion of the weighted adjacency matrix of the graph. The parameter `lengths` points to a vector that will store the length of the shortest paths from `source` to the locally stored vertices. Finally, the parameter `comm` is the communicator to be used by the MPI routines. Note that this routine assumes that the number of vertices is a multiple of the number of processors.

Program 6.6 Dijkstra's Single-Source Shortest-Path

[View full width]

```
 1   SingleSource(int n, int source, int *wgt, int *lengths, MPI_Comm comm)
 2   {
 3     int i, j;
 4     int nlocal;  /* The number of vertices stored locally */
 5     int *marker;  /* Used to mark the vertices belonging to V_o */
 6     int firstvtx;  /* The index number of the first vertex that is stored locall
 7     int lastvtx;  /* The index number of the last vertex that is stored locally
 8     int u, udist;
 9     int lminpair[2], gminpair[2];
10     int npes, myrank;
11     MPI_Status status;
12
13     MPI_Comm_size(comm, &npes);
14     MPI_Comm_rank(comm, &myrank);
15
16     nlocal   = n/npes;
17     firstvtx = myrank*nlocal;
18     lastvtx  = firstvtx+nlocal-1;
19
20     /* Set the initial distances from source to all the other vertices */
21     for (j=0; j<nlocal; j++)
22       lengths[j] = wgt[source*nlocal + j];
23
24     /* This array is used to indicate if the shortest part to a vertex has been
   or not. */
25     /* if marker [v] is one, then the shortest path to v has been found, otherwi
   has not. */
26     marker = (int *)malloc(nlocal*sizeof(int));
27     for (j=0; j<nlocal; j++)
28       marker[j] = 1;
29
30     /* The process that stores the source vertex, marks it as being seen */
31     if (source >= firstvtx && source <= lastvtx)
32       marker[source-firstvtx] = 0;
33
34     /* The main loop of Dijkstra's algorithm */
35     for (i=1; i<n; i++) {
36       /* Step 1: Find the local vertex that is at the smallest distance from sou
37       lminpair[0] = MAXINT; /* set it to an architecture dependent large number
38       lminpair[1] = -1;
39       for (j=0; j<nlocal; j++) {
40         if (marker[j] && lengths[j] < lminpair[0]) {
41           lminpair[0] = lengths[j];
42           lminpair[1] = firstvtx+j;
43         }
44       }
45
46       /* Step 2: Compute the global minimum vertex, and insert it into V_c */
47        MPI_Allreduce(lminpair, gminpair, 1, MPI_2INT, MPI_MINLOC,
48           comm);
49       udist = gminpair[0];
50       u = gminpair[1];
51
52       /* The process that stores the minimum vertex, marks it as being seen */
```

```
53        if (u == lminpair[1])
54          marker[u-firstvtx] = 0;
55
56        /* Step 3: Update the distances given that u got inserted */
57        for (j=0; j<nlocal; j++) {
58          if (marker[j] && udist + wgt[u*nlocal+j] < lengths[j])
59            lengths[j] = udist + wgt[u*nlocal+j];
60        }
61      }
62
63      free(marker);
64    }
```

The main computational loop of Dijkstra's parallel single-source shortest path algorithm performs three steps. First, each process finds the locally stored vertex in V_o that has the smallest distance from the source. Second, the vertex that has the smallest distance over all processes is determined, and it is included in V_c . Third, all processes update their distance arrays to reflect the inclusion of the new vertex in V_c .

The first step is performed by scanning the locally stored vertices in V_o and determining the one vertex *v* with the smaller *lengths* [*v*] value. The result of this computation is stored in the array *lminpair* . In particular, *lminpair* [0] stores the distance of the vertex, and *lminpair* [1] stores the vertex itself. The reason for using this storage scheme will become clear when we consider the next step, in which we must compute the vertex that has the smallest overall distance from the source. We can find the overall shortest distance by performing a min-reduction on the distance values stored in *lminpair* [0]. However, in addition to the shortest distance, we also need to know the vertex that is at that shortest distance. For this reason, the appropriate reduction operation is the `MPI_MINLOC` which returns both the minimum as well as an index value associated with that minimum. Because of `MPI_MINLOC` we use the two-element array *lminpair* to store the distance as well as the vertex that achieves this distance. Also, because the result of the reduction operation is needed by all the processes to perform the third step, we use the `MPI_Allreduce` operation to perform the reduction. The result of the reduction operation is returned in the *gminpair* array. The third and final step during each iteration is performed by scanning the local vertices that belong in V_o and updating their shortest distances from the source vertex.

Avoiding Load Imbalances Program 6.6 assigns *n* /*p* consecutive columns of *W* to each processor and in each iteration it uses the `MPI_MINLOC` reduction operation to select the vertex v to be included in V_c . Recall that the `MPI_MINLOC` operation for the pairs (*a* , *i*) and (*a* , *j*) will return the one that has the smaller index (since both of them have the same value). Consequently, among the vertices that are equally close to the source vertex, it favors the smaller numbered vertices. This may lead to load imbalances, because vertices stored in lower-ranked processes will tend to be included in V_c faster than vertices in higher-ranked processes (especially when many vertices in V_o are at the same minimum distance from the source). Consequently, the size of the set V_o will be larger in higher-ranked processes, dominating the overall runtime.

One way of correcting this problem is to distribute the columns of *W* using a cyclic distribution. In this distribution process *i* gets every *p* th vertex starting from vertex *i* . This scheme also assigns *n* /*p* vertices to each process but these vertices have indices that span almost the entire graph. Consequently, the preference given to lower-numbered vertices by `MPI_MINLOC` does not lead to load-imbalance problems.

6.6.10 Example: Sample Sort

The last problem requiring collective communications that we will consider is that of sorting a

sequence A of n elements using the sample sort algorithm described in Section 9.5 . The program is shown in Program 6.7 .

The `SampleSort` function takes as input the sequence of elements stored at each process and returns a pointer to an array that stores the sorted sequence as well as the number of elements in this sequence. The elements of this `SampleSort` function are integers and they are sorted in increasing order. The total number of elements to be sorted is specified by the parameter `n` and a pointer to the array that stores the local portion of these elements is specified by `elmnts` . On return, the parameter `nsorted` will store the number of elements in the returned sorted array. This routine assumes that `n` is a multiple of the number of processes.

Program 6.7 Samplesort

[View full width]

```
 1   int *SampleSort(int n, int *elmnts, int *nsorted, MPI_Comm comm)
 2   {
 3     int i, j, nlocal, npes, myrank;
 4     int *sorted_elmnts, *splitters, *allpicks;
 5     int *scounts, *sdispls, *rcounts, *rdispls;
 6
 7     /* Get communicator-related information */
 8     MPI_Comm_size(comm, &npes);
 9     MPI_Comm_rank(comm, &myrank);
10
11     nlocal = n/npes;
12
13     /* Allocate memory for the arrays that will store the splitters */
14     splitters = (int *)malloc(npes*sizeof(int));
15     allpicks = (int *)malloc(npes*(npes-1)*sizeof(int));
16
17     /* Sort local array */
18     qsort(elmnts, nlocal, sizeof(int), IncOrder);
19
20     /* Select local npes-1 equally spaced elements */
21     for (i=1; i<npes; i++)
22       splitters[i-1] = elmnts[i*nlocal/npes];
23
24     /* Gather the samples in the processors */
25     MPI_Allgather(splitters, npes-1, MPI_INT, allpicks, npes-1,
26         MPI_INT, comm);
27
28     /* sort these samples */
29     qsort(allpicks, npes*(npes-1), sizeof(int), IncOrder);
30
31     /* Select splitters */
32     for (i=1; i<npes; i++)
33       splitters[i-1] = allpicks[i*npes];
34     splitters[npes-1] = MAXINT;
35
36     /* Compute the number of elements that belong to each bucket */
37     scounts = (int *)malloc(npes*sizeof(int));
38     for (i=0; i<npes; i++)
39       scounts[i] = 0;
40
```

```
41      for (j=i=0; i<nlocal; i++) {
42        if (elmnts[i] < splitters[j])
43          scounts[j]++;
44        else
45          scounts[++j]++;
46      }
47
48      /* Determine the starting location of each bucket's elements in the elmnts a
49       sdispls = (int *)malloc(npes*sizeof(int));
50      sdispls[0] = 0;
51      for (i=1; i<npes; i++)
52         sdispls[i] = sdispls[i-1]+scounts[i-1];
53
54      /* Perform an all-to-all to inform the corresponding processes of the number
    elements */
55      /* they are going to receive. This information is stored in rcounts array */
56       rcounts = (int *)malloc(npes*sizeof(int));
57       MPI_Alltoall(scounts, 1, MPI_INT, rcounts, 1, MPI_INT, comm);
58
59      /* Based on rcounts determine where in the local array the data from each
    processor */
60      /* will be stored. This array will store the received elements as well as th
    final */
61      /* sorted sequence */
62       rdispls = (int *)malloc(npes*sizeof(int));
63      rdispls[0] = 0;
64      for (i=1; i<npes; i++)
65         rdispls[i] = rdispls[i-1]+rcounts[i-1];
66
67       *nsorted = rdispls[npes-1]+rcounts[i-1];
68       sorted_elmnts = (int *)malloc((*nsorted)*sizeof(int));
69
70      /* Each process sends and receives the corresponding elements, using the
    MPI_Alltoallv */
71      /* operation. The arrays scounts and sdispls are used to specify the number
    elements */
72      /* to be sent and where these elements are stored, respectively. The arrays
    rcounts */
73      /* and rdispls are used to specify the number of elements to be received, ar
    where these */
74      /* elements will be stored, respectively. */
75       MPI_Alltoallv(elmnts, scounts, sdispls, MPI_INT, sorted_elmnts,
76           rcounts, rdispls, MPI_INT, comm);
77
78      /* Perform the final local sort */
79       qsort(sorted_elmnts, *nsorted, sizeof(int), IncOrder);
80
81       free(splitters); free(allpicks); free(scounts); free(sdispls);
82       free(rcounts); free(rdispls);
83
84       return sorted_elmnts;
85    }
```

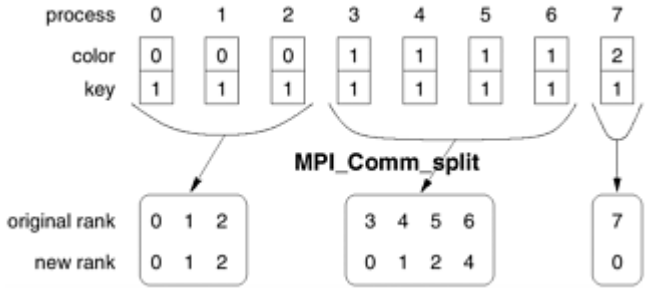
6.7 Groups and Communicators

In many parallel algorithms, communication operations need to be restricted to certain subsets of processes. MPI provides several mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator. A general method for partitioning a graph of processes is to use `MPI_Comm_split` that is defined as follows:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
        MPI_Comm *newcomm)
```

This function is a collective operation, and thus needs to be called by all the processes in the communicator `comm` . The function takes `color` and `key` as input parameters in addition to the communicator, and partitions the group of processes in the communicator `comm` into disjoint subgroups. Each subgroup contains all processes that have supplied the same value for the `color` parameter. Within each subgroup, the processes are ranked in the order defined by the value of the `key` parameter, with ties broken according to their rank in the old communicator (i.e., `comm`). A new communicator for each subgroup is returned in the `newcomm` parameter. Figure 6.7 shows an example of splitting a communicator using the `MPI_Comm_split` function. If each process called `MPI_Comm_split` using the values of parameters `color` and `key` as shown in Figure 6.7 , then three communicators will be created, containing processes {0, 1, 2}, {3, 4, 5, 6}, and {7}, respectively.

Figure 6.7. Using **`MPI_Comm_split`** to split a group of processes in a communicator into subgroups.



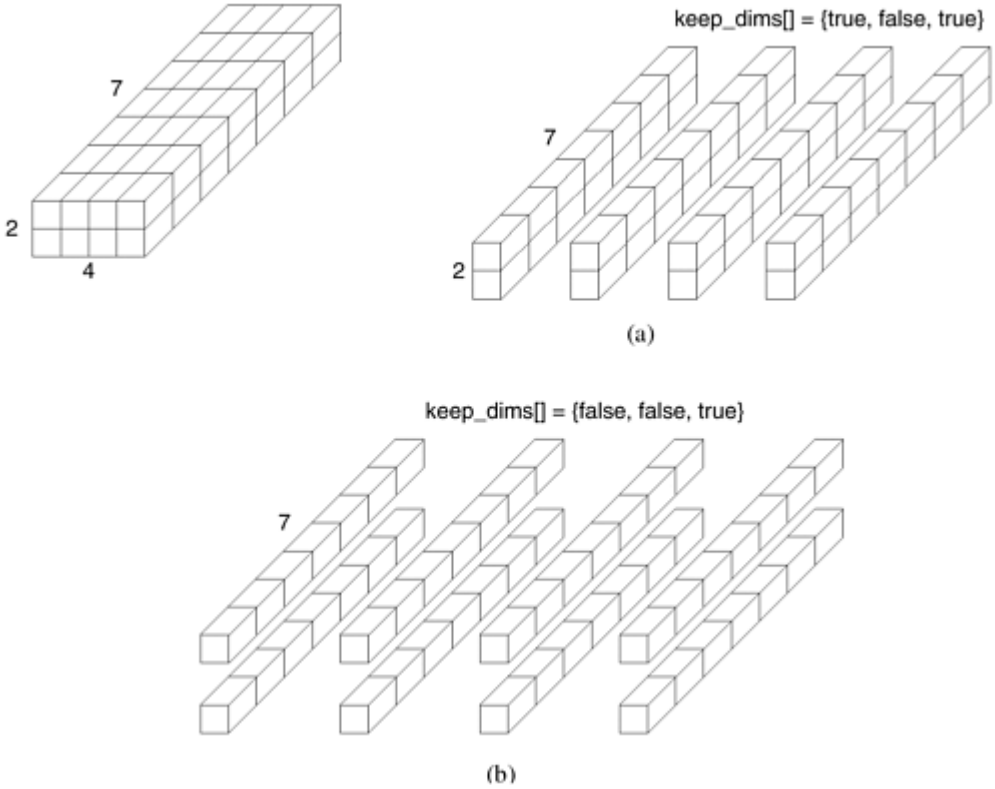
Splitting Cartesian Topologies In many parallel algorithms, processes are arranged in a virtual grid, and in different steps of the algorithm, communication needs to be restricted to a different subset of the grid. MPI provides a convenient way to partition a Cartesian topology to form lower-dimensional grids.

MPI provides the `MPI_Cart_sub` function that allows us to partition a Cartesian topology into sub-topologies that form lower-dimensional grids. For example, we can partition a two-dimensional topology into groups, each consisting of the processes along the row or column of the topology. The calling sequence of `MPI_Cart_sub` is the following:

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
        MPI_Comm *comm_subcart)
```

The array `keep_dims` is used to specify how the Cartesian topology is partitioned. In particular, if `keep_dims[i]` is true (non-zero value in C) then the `i` th dimension is retained in the new sub-topology. For example, consider a three-dimensional topology of size 2 x 4 x 7. If `keep_dims` is {true, false, true}, then the original topology is split into four two-dimensional sub-topologies of size 2 x 7, as illustrated in Figure 6.8(a) . If `keep_dims` is {false, false, true}, then the original topology is split into eight one-dimensional topologies of size seven, illustrated in Figure 6.8(b) . Note that the number of sub-topologies created is equal to the product of the number of processes along the dimensions that are not being retained. The original topology is specified by the communicator `comm_cart` , and the returned communicator `comm_subcart` stores information about the created sub-topology. Only a single communicator is returned to each process, and for processes that do not belong to the same sub-topology, the group specified by the returned communicator is different.

Figure 6.8. Splitting a Cartesian topology of size 2 x 4 x 7 into (a) four subgroups of size 2 x 1 x 7, and (b) eight subgroups of size 1 x 1 x 7.



The processes belonging to a given sub-topology can be determined as follows. Consider a three-dimensional topology of size d_1 x d_2 x d_3 , and assume that `keep_dims` is set to {true, false, true}. The group of processes that belong to the same sub-topology as the process with coordinates (x , y , z) is given by (*, y , *), where a '*' in a coordinate denotes all the possible values for this coordinate. Note also that since the second coordinate can take d_2 values, a total of d_2 sub-topologies are created.

Also, the coordinate of a process in a sub-topology created by `MPI_Cart_sub` can be obtained from its coordinate in the original topology by disregarding the coordinates that correspond to the dimensions that were not retained. For example, the coordinate of a process in the column-based sub-topology is equal to its row-coordinate in the two-dimensional topology. For instance, the process with coordinates (2, 3) has a coordinate of (2) in the sub-topology that corresponds to the third column of the grid.

6.7.1 Example: Two-Dimensional Matrix-Vector Multiplication

In Section 6.6.8 , we presented two programs for performing the matrix-vector multiplication $x = Ab$ using a row- and column-wise distribution of the matrix. As discussed in Section 8.1.2 , an alternative way of distributing matrix A is to use a two-dimensional distribution, giving rise to the two-dimensional parallel formulations of the matrix-vector multiplication algorithm.

Program 6.8 shows how these topologies and their partitioning are used to implement the two-dimensional matrix-vector multiplication. The dimension of the matrix is supplied in the parameter n , the parameters a and b point to the locally stored portions of matrix A and vector b , respectively, and the parameter x points to the local portion of the output matrix-vector product. Note that only the processes along the first column of the process grid will store b initially, and that upon return, the same set of processes will store the result x . For simplicity, the program assumes that the number of processes p is a perfect square and that n is a multiple of $\sqrt{p}$.

Program 6.8 Two-Dimensional Matrix-Vector Multiplication

[View full width]

```
 1   MatrixVectorMultiply_2D(int n, double *a, double *b, double *x,
 2                           MPI_Comm comm)
 3   {
 4     int ROW=0, COL=1; /* Improve readability */
 5     int i, j, nlocal;
 6     double *px; /* Will store partial dot products */
 7     int npes, dims[2], periods[2], keep_dims[2];
 8     int myrank, my2drank, mycoords[2];
 9     int other_rank, coords[2];
10     MPI_Status status;
11     MPI_Comm comm_2d, comm_row, comm_col;
12
13     /* Get information about the communicator */
14     MPI_Comm_size(comm, &npes);
15     MPI_Comm_rank(comm, &myrank);
16
17     /* Compute the size of the square grid */
18     dims[ROW] = dims[COL] = sqrt(npes);
19
20     nlocal = n/dims[ROW];
21
22     /* Allocate memory for the array that will hold the partial dot-products */
23     px = malloc(nlocal*sizeof(double));
24
25     /* Set up the Cartesian topology and get the rank & coordinates of the proce
   this topology */
26     periods[ROW] = periods[COL] = 1; /* Set the periods for wrap-around connecti
27
28     MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_2d);
29
30     MPI_Comm_rank(comm_2d, &my2drank); /* Get my rank in the new topology */
31     MPI_Cart_coords(comm_2d, my2drank, 2, mycoords); /* Get my coordinates */
32
33     /* Create the row-based sub-topology */
```

```
34      keep_dims[ROW] = 0;
35      keep_dims[COL] = 1;
36      MPI_Cart_sub(comm_2d, keep_dims, &comm_row);
37
38     /* Create the column-based sub-topology */
39      keep_dims[ROW] = 1;
40      keep_dims[COL] = 0;
41      MPI_Cart_sub(comm_2d, keep_dims, &comm_col);
42
43     /* Redistribute the b vector. */
44     /* Step 1. The processors along the 0th column send their data to the diagor
processors */
45      if (mycoords[COL] == 0 && mycoords[ROW] != 0) { /* I'm in the first column *
46        coords[ROW] = mycoords[ROW];
47        coords[COL] = mycoords[ROW];
48        MPI_Cart_rank(comm_2d, coords, &other_rank);
49        MPI_Send(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d);
50      }
51      if (mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0) {
52        coords[ROW] = mycoords[ROW];
53        coords[COL] = 0;
54        MPI_Cart_rank(comm_2d, coords, &other_rank);
55        MPI_Recv(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d,
56            &status);
57      }
58
59     /* Step 2. The diagonal processors perform a column-wise broadcast */
60      coords[0] = mycoords[COL];
61      MPI_Cart_rank(comm_col, coords, &other_rank);
62      MPI_Bcast(b, nlocal, MPI_DOUBLE, other_rank, comm_col);
63
64     /* Get into the main computational loop */
65      for (i=0; i<nlocal; i++) {
66        px[i] = 0.0;
67        for (j=0; j<nlocal; j++)
68          px[i] += a[i*nlocal+j]*b[j];
69      }
70
71     /* Perform the sum-reduction along the rows to add up the partial dot-produc
72     coords[0] = 0;
73      MPI_Cart_rank(comm_row, coords, &other_rank);
74      MPI_Reduce(px, x, nlocal, MPI_DOUBLE, MPI_SUM, other_rank,
75          comm_row);
76
77      MPI_Comm_free(&comm_2d); /* Free up communicator */
78      MPI_Comm_free(&comm_row); /* Free up communicator */
79      MPI_Comm_free(&comm_col); /* Free up communicator */
80
81      free(px);
82    }
```

6.8 Bibliographic Remarks

The best source for information about MPI is the actual reference of the library itself [Mes94]. At the time of writing of this book, there have been two major releases of the MPI standard. The first release, version 1.0, was released in 1994 and its most recent revision, version 1.2, has been implemented by the majority of hardware vendors. The second release of the MPI standard, version 2.0 [Mes97], contains numerous significant enhancements over version 1.x, such as one-sided communication, dynamic process creation, and extended collective operations. However, despite the fact that the standard was voted in 1997, there are no widely available MPI-2 implementations that support the entire set of features specified in that standard. In addition to the above reference manuals, a number of books have been written that focus on parallel programming using MPI [Pac98, GSNL98, GLS99].

In addition to MPI implementations provided by various hardware vendors, there are a number of publicly available MPI implementations that were developed by various government research laboratories and universities. Among them, the MPICH [GLDS96, GL96b] (available at http://www-unix.mcs.anl.gov/mpi/mpich) distributed by Argonne National Laboratories and the LAM-MPI (available at http://www.lam-mpi.org) distributed by Indiana University are widely used and are portable to a number of different architectures. In fact, these implementations of MPI have been used as the starting point for a number of specialized MPI implementations that are suitable for off-the-shelf high-speed interconnection networks such as those based on gigabit Ethernet and Myrinet networks.

Problems

6.1 Describe a message-transfer protocol for buffered sends and receives in which the buffering is performed only by the sending process. What kind of additional hardware support is needed to make these types of protocols practical?

6.2 One of the advantages of non-blocking communication operations is that they allow the transmission of the data to be done concurrently with computations. Discuss the type of restructuring that needs to be performed on a program to allow for the maximal overlap of computation with communication. Is the sending process in a better position to benefit from this overlap than the receiving process?

6.3 As discussed in Section 6.3.4 the MPI standard allows for two different implementations of the `MPI_Send` operation – one using buffered-sends and the other using blocked-sends. Discuss some of the potential reasons why MPI allows these two different implementations. In particular, consider the cases of different message-sizes and/or different architectural characteristics.

6.4 Consider the various mappings of 16 processors on a 4 x 4 two-dimensional grid shown in Figure 6.5. Show how $n = \sqrt{p} \times \sqrt{p}$ processors will be mapped using each one of these four schemes.

6.5 Consider Cannon's matrix-matrix multiplication algorithm. Our discussion of Cannon's algorithm has been limited to cases in which A and B are square matrices, mapped onto a square grid of processes. However, Cannon's algorithm can be extended for cases in which A, B, and the process grid are not square. In particular, let matrix A be of size n x k and matrix B be of size k x m. The matrix C obtained by multiplying A and B is of size n x m. Also, let q x r be the number of processes in the grid arranged in q rows and r columns. Develop an MPI program for multiplying two such matrices on a q x r process grid using Cannon's algorithm.

6.6 Show how the row-wise matrix-vector multiplication program (Program 6.4) needs to be changed so that it will work correctly in cases in which the dimension of the matrix does not have to be a multiple of the number of processes.

6.7 Consider the column-wise implementation of matrix-vector product (Program 6.5). An alternate implementation will be to use `MPI_Allreduce` to perform the required reduction operation and then have each process copy the locally stored elements of vector x from the vector `fx`. What will be the cost of this implementation? Another implementation can be to perform p single-node reduction operations using a different process as the root. What will be the cost of this implementation?

6.8 Consider Dijkstra's single-source shortest-path algorithm described in Section 6.6.9. Describe why a column-wise distribution is preferable to a row-wise distribution of the weighted adjacency matrix.

6.9 Show how the two-dimensional matrix-vector multiplication program (Program 6.8) needs to be changed so that it will work correctly for a matrix of size n x m on a q x r process grid.