Parallel and Distributed Computation: Numerical Methods

OPTIMIZATION AND NEURAL COMPUTATION SERIES

1. Dynamic Programming and Optimal Control, Vols. I and II, by Dimitri P. Bertsekas, 1995 (ISBN 1-886529-11-6, 704 pages, hardcover)
2. Nonlinear Programming, by Dimitri P. Bertsekas, 1995 (ISBN 1-886529-14-0, 656 pages, hardcover)
3. Neuro-Dynamic Programming, by Dimitri P. Bertsekas and John N. Tsitsiklis, 1996 (ISBN 1-886529-10-8, 512 pages, hardcover)
4. Constrained Optimization and Lagrange Multiplier Methods, by Dimitri P. Bertsekas, 1996 (ISBN 1-886529-04-3, 410 pages, softcover)
5. Stochastic Optimal Control: The Discrete-Time Case by Dimitri P. Bertsekas and Steven E. Shreve, 1996 (ISBN 1-886529-03-5, 330 pages, softcover)
6. Introduction to Linear Optimization by Dimitris Bertsimas and John N. Tsitsiklis, 1997 (ISBN 1-886529-19-1, 608 pp., hardcover)
7. Parallel and Distributed Computation: Numerical Methods by Dimitri P. Bertsekas and John N. Tsitsiklis, 1997 (ISBN 1-886529-01-9, 731 pages, softcover)

Parallel and Distributed Computation: Numerical Methods

Dimitri P. Bertsekas and John N. Tsitsiklis

Massachusetts Institute of Technology

WWW site for book information and orders

http://world.std.com/~athenasc/

Athena Scientific, Belmont, Massachusetts

Athena Scientific
Post Office Box 391
Belmont, Mass. 02178-9998
U.S.A.

Email: athenasc@world.std.com
WWW information and orders: http://world.std.com/~athenasc/

Cover Design: *Ann Gallager*

Originally published by Prentice-Hall, Inc., in 1989. Corrections listed at the end.

Publisher's Cataloging-in-Publication Data

Bertsekas, Dimitri P.
Parallel and Distributed Computation: Numerical Methods
Includes bibliographical references and index
1. Parallel processing (Electronic computers)
I. John Tsitsiklis N., joint author. II. Title.
QA76.5.B457 1997 004'.35 97-70648

ISBN 1-886529-01-9

To Joanna and Daphne

Preface

Parallel and distributed computing systems offer the promise of a quantum leap in the computing power that can be brought to bear on many important problems. Whether and to what extent this promise can be fulfilled is still a matter of speculation, but several years of practical experience with both parallel computers and distributed data communication networks have brought about an understanding of the potential and limitations of parallel and distributed computation. The purpose of this book is to promote this understanding by focusing on algorithms that are naturally suited for large scale parallelization and that represent the best hope for solving problems which are much larger than those that can be solved at present.

Work on parallel and distributed computation spans several broad areas, such as the design of parallel machines, parallel programming languages, parallel algorithm development and analysis, and applications related issues. The focus of this book is on algorithms, and, even within this area, we restrict our attention primarily to numerical algorithms for solving systems of equations or optimization problems. Our choice of material is motivated by large problems for which it is essential to harness the power of massive parallelism, while keeping the communication overhead and delays within tolerable limits. Accordingly, we emphasize algorithms that admit a high degree of parallelization such as relaxation methods of the Jacobi and Gauss-Seidel type, and we

address extensively issues of communication and synchronization. In particular, we discuss algorithms for interprocessor communication and we provide a comprehensive convergence analysis of asynchronous iterative methods.

The design of parallel algorithms can be pursued at several levels, and this explains to some extent the diversity of the literature on the subject. For example:

(a) One approach is to parallelize an existing serial algorithm, perhaps after modifications, or to develop a new and easier to parallelize algorithm, without being too specific about the implementation in particular types of machines. Here one might be concerned with the algorithm's convergence and rate of convergence (in either a synchronous or an asynchronous computing environment), and with the algorithm's potential for substantial speedup over its serial counterpart.

(b) A second approach is to focus on the details of implementation on a particular type of machine. The issues here are algorithmic correctness, as well as time and communication complexity of the implementation.

(c) In still another approach, the choice of the algorithm and the parallel machine are interdependent to the point where the design of one has a strong influence on the design of the other. A typical example is when a VLSI chip is designed to execute efficiently a special type of parallel algorithm.

We have mostly followed the first approach, concentrating on algorithmic analysis at a rather high level of abstraction. Our choice of algorithms, however, is such that in most cases, the methods of parallel implementation are either evident and straightforward, or else are covered by our broad discussion of parallel computation given in Chapter 1. We have not dealt with implementations in specific machines because types of machines are rapidly changing. Nonetheless, at several points, we have made reference to computations in regular architectures, such as mesh and hypercube, which are widely used. We carry out the analysis of various algorithms in considerable depth, guided by the belief that a thorough understanding of an algorithm is typically essential for its application to a challenging problem.

The book was developed through a course that we taught to graduate students at MIT. It is intended for use in graduate courses in engineering, computer science, operations research, and applied mathematics. We have assumed the equivalent of a first course in linear algebra and a grasp of advanced calculus and real analysis that most students are exposed to by the end of their undergraduate studies. Probabilistic notions beyond the theory of finite-state Markov chains are not needed with the exception of Section 7.8, which deals with stochastic gradient methods. We have not required any background in numerical analysis, graph algorithms, optimization, convexity, and duality, and we have consequently developed this material as needed in the main body of the text or the appendices. We note, however, that the mathematically mature reader who has some background in some of these fields is likely to benefit more from the book, and to gain deeper appreciation of the material.

The book can be used for several types of courses. One possibility is a course targeted on parallel algorithms, and intended for students who already have some knowl-

edge of a subset of the fields of numerical analysis, graph theory, and optimization algorithms. Furthermore, such a course could have either a computer science flavor, by focusing on Chapters 1 and 8, and parts of Chapters 2, and 4 through 6, or alternatively a numerical computation flavor by focusing on Chapters 2, 3, and parts of Chapters 1 and 4 through 7. Another possibility is a general course on numerical methods with a strong bias towards parallelizable algorithms. The book lends itself for such a course because it develops economically a broad variety of self-contained basic material in considerable depth.

Chapter 1 contains an exposition of some generic issues that arise in a broad variety of parallel algorithms and are best addressed without specific reference to any particular algorithm. In particular, we discuss the scheduling of a set of processors for the parallel execution of a prescribed algorithm, some basic issues relating to interprocessor communication, and the effects of the communication penalty on the amount by which an algorithm can be speeded up. Special attention is paid to a few interesting architectures such as mesh and hypercube. We then consider issues of synchronization, and we contrast synchronous and asynchronous algorithms. In this chapter, we also introduce relaxation methods of the Gauss-Seidel and Jacobi type and some associated issues of parallelization, communication, and synchronization that are recurring themes throughout the book.

Chapter 2 deals with parallel algorithms for systems of linear equations and matrix inversion. It covers direct methods for general systems as well as systems with special structure, iterative methods, including their convergence analysis, and the conjugate gradient method.

Chapter 3 is devoted to iterative methods for nonlinear problems, such as finding fixed points of contraction mappings, unconstrained and constrained optimization, and variational inequalities. The convergence theory for such methods is developed in an economical way and emphasizes the case of Cartesian product constraint sets (in a primal and a dual setting), which lends itself naturally to parallelization and decomposition.

Chapter 4 deals with the shortest path problem and other, more general, dynamic programming problems. The dynamic programming algorithm can be viewed as a relaxation method and lends itself well for parallelization. We establish (and strengthen somewhat) the classical results for discounted and undiscounted Markovian decision problems, and we also discuss the associated parallel computation issues.

Chapter 5 is devoted to network flow problems. In the first four sections, we deal with the important class of linear problems, and we present some easily parallelizable algorithms, that are conceptually related to the Gauss-Seidel and Jacobi relaxation methods. We then discuss related algorithms for network problems with nonlinear convex cost. The methods of the first five sections can be viewed as relaxation methods applied in a space of dual (price) variables. In the last section we consider relaxation-like methods applied to nonlinear multicommodity flow problems in the primal space of flow variables.

The last three chapters deal with asynchronous algorithms in which each processor computes at its own pace, using intermediate results of other processors that are possibly outdated due to unpredictable communication delays. Among other topics, we develop

asynchronous versions of all the major types of synchronous parallel algorithms that were discussed in previous chapters.

In Chapter 6, we introduce a general class of asynchronous iterative methods (called "totally asynchronous"), and we develop a general theorem for proving their convergence. This theorem is used repeatedly to establish the validity of a broad variety of asynchronous algorithms involving iteration mappings that are either monotone or contracting with respect to a weighted maximum norm. In particular, we show convergence of linear and nonlinear iterations involving weighted maximum norm contractions arising in the solution of systems of algebraic or differential equations, discounted dynamic programming, unconstrained and constrained optimization, and variational inequalities. We also discuss iterations involving monotone mappings arising in shortest path problems, undiscounted dynamic programming, and linear and nonlinear network flow problems.

In Chapter 7, we consider "partially asynchronous" algorithms in which some mild restrictions are placed on the amount of asynchronism present. We prove convergence of a variety of algorithms for fixed points of nonexpansive mappings, deterministic and stochastic optimization, Markov chains, load balancing in a computer network, and optimal routing in data networks.

Chapter 8 is similar in philosophy to Chapter 1 in that it deals with generic issues of parallel and distributed computation. It discusses the organization of an inherently asynchronous network of processors for the purpose of executing a general type of parallel algorithm. It addresses issues like termination detection, processor scheduling, methods for taking a "snapshot" of the global state of a computing system, synchronization via "rollback," and methods for maintaining communication with a center in the face of topological changes.

Many of our subjects can be covered independently of each other, thereby allowing the reader or an instructor to use material selectively to suit his/her needs. For example, the following groups of sections can be omitted without loss of continuity:

(a) Sections 2.1 to 2.3, that deal with direct methods for linear systems of equations.

(b) Sections 2.8, 4.2, 4.3, 7.3, 7.4, 7.7, and 7.8 that develop or use the theory of Markov chains.

(c) The material on decomposition methods based on duality in Section 3.4 and Subsection 3.5.7.

(d) The dynamic programming material of Sections 4.2 and 4.3.

(e) The material on linear network flow problems in Sections 5.1 to 5.4, and 6.5.

(f) Sections 5.6 and 7.6, dealing with nonlinear multicommodity network flow problems.

(g) The material on nonlinear network flow problems in Sections 5.5, 6.6, and Subsection 7.2.3.

Each major section contains several exercises that, for the most part, illustrate and supplement the theoretical developments of the text. They include algorithmic variations, convergence and complexity analysis, examples, and counterexamples. Some of the

exercises are quite challenging, occasionally representing recent research. The serious reader will benefit a great deal from these exercises, which constitute one of the principal components of the text. Solutions of all the exercises are provided in a manual that will be available to instructors.

A substantial portion of our material has not been covered in other textbooks. This includes most of the last two sections of Chapter 1, much of the last two sections of Chapter 3, Sections 5.2 through 5.5, the entire Chapters 6 and 7, and most of Chapter 8. Some of the material presented was developed as the textbook was being written and has not yet been published elsewhere.

The literature on our subject is enormous, and our references are not comprehensive. We thus apologize in advance to the many authors whose work has not been cited. We have restricted ourselves to listing the sources that we have used, together with a selection of sources that contain material supplementing the text.

We are thankful to a number of individuals and institutions for their help. The inquisitive spirit of our students motivated us to think harder about many issues. We learned a great deal about distributed computation through our long association and collaboration with Bob Gallager and Pierre Humblet. We have appreciated our research collaboration with Michael Athans, David Castanon, Jon Eckstein, Eli Gafni, and Christos Papadimitriou, that produced some of the material included in the book. Tom Luo and Cuneyt Ozveren contributed research material that was incorporated in exercises. We are thankful for the helpful comments of a number of people, including Chee-Seng Chow, George Cybenko, Stratis Gallopoulos, George Hart, and Tom Richardson. Our greatest debt of gratitude to a single individual goes to Paul Tseng who worked closely with us on several of the topics presented, particularly the communication algorithms of Section 1.3, the network flow algorithms of Chapter 5, and the partially asynchronous algorithms of Section 7.2. In addition, Paul reviewed the entire manuscript, sharpened several proofs and results, and contributed much research in the form of exercises. We were fortunate to work at the Laboratory for Information and Decision Systems of M.I.T., which provided us with a stimulating research environment. Funding for our research was provided by the Army Research Office through the Center for Intelligent Control Systems, Bellcore Inc., the National Science Foundation, and the Office of Naval Research.

1

Introduction

As we embark on the study of parallel and distributed numerical methods it is useful to reflect on their differences from their serial counterparts. There are several issues related to parallelization that do not arise in a serial context. A first issue is *task allocation*, that is, the breakdown of the total workload in smaller tasks assigned to different processors, and the proper sequencing of the tasks when some of them are interdependent and cannot be executed simultaneously. A second issue is the *communication* of interim computation results between the processors; our objective here is to carry out the communication efficiently, and to estimate its impact on performance. A third issue is the *synchronization* of the computations of different processors. In some methods, called *synchronous*, processors must wait at predetermined points for the completion of certain computations or for the arrival of certain data, and the mechanism used to enforce such synchronization may have an important effect on performance. In other methods, called *asynchronous*, there is no requirement for waiting at predetermined points, and the corresponding implications for the methods' validity must be assessed. Other issues relate to the development of appropriate performance measures for parallel methods, and the effects of the system's architecture on these performance measures.

Issues such as the above are important in a variety of contexts and are, therefore, most economically studied without reference to a specific numerical method. We address some of them in this introductory chapter, and we develop some results and methodological approaches that will be used throughout the book. Our analysis is not always fully

rigorous because we do not always adhere to formal models of distributed computation. This helps us develop the main ideas in a more accessible and intuitive manner than it would be possible otherwise. At the same time our analysis is sufficiently detailed to provide the basis for more rigorous proofs where needed, and to convince most readers of the essential correctness of our results.

Section 1.1 contains a brief overview of some application domains and of the presently existing parallel computing systems. In Section 1.2, we consider a simple model of synchronous parallel computation, in which communication issues are ignored, and discuss the concepts of time complexity, speedup, and efficiency. We also discuss issues arising in the parallelization of iterative methods. In Section 1.3, we consider communication issues in parallel and distributed systems. Following a brief discussion of data link control and routing, we formulate some basic communication problems that arise frequently in the algorithms of subsequent chapters, and we provide optimal or nearly optimal algorithms for these problems. At the same time, we discuss the properties of some of the more popular processor interconnection networks. In Section 1.4, we consider methods for algorithm synchronization. We also introduce asynchronous algorithms, compare them informally with their synchronous counterparts, and provide a glimpse of some of their interesting convergence properties that will be the focal point of Chapters 6 and 7.

1.1 PARALLEL AND DISTRIBUTED ARCHITECTURES

Parallel and distributed computation is currently an area of intense research activity, motivated by a variety of factors. There has always been a need for the solution of very large computational problems, but it is only recently that technological advances have raised the possibility of massively parallel computation and have made the solution of such problems possible. Furthermore, the availability of powerful parallel computers is generating interest in new types of problems that were not addressed in the past. Accordingly, the development of parallel and distributed algorithms is guided by this interplay between old and new computational needs on the one hand, and technological progress on the other. To appreciate this effect, we briefly discuss some application areas and the types of computing systems that new technologies have made possible.

1.1.1 The Need for Parallel and Distributed Computation

We restrict attention to numerical computation, since this is the major application considered in this book. Symbolic computation and artificial intelligence applications have also played an important role in the development of the subject, but are outside our scope.

The original needs for fast computation have been in a number of contexts involving partial differential equations (PDEs), such as computational fluid dynamics and weather prediction, as well as in image processing, etc. In these applications, there is a large number of numerical computations to be performed. The desire to solve more and more

complex problems has always been running ahead of the capabilities of the time, and has provided a driving force for the development of faster, and possibly parallel, computing machines. The above mentioned types of problems can be easily decomposed along a spatial dimension, and have therefore been prime candidates for parallelization, with a different computational unit (processor) assigned the task of manipulating the variables associated with a small region in space. Furthermore, in such problems, interactions between variables are local in nature, thus leading to the design of parallel computers consisting of a number of processors with nearest neighbor connections.

More recently, there has been increased interest in other types of large scale computation. Some examples are the analysis, simulation, and optimization of large scale interconnected systems, queueing systems being a noteworthy representative. Other examples relate to the solution of general systems of equations, mathematical programming, and optimization problems. A common property of such problems, as they arise in practice, is that they can be decomposed, but the subtasks obtained from such a decomposition tend to be fewer and more complex than those obtained in the context of partial differential equations. In particular, the regular and repetitive structure of PDEs is lost. Accordingly, one is led to use fewer and more powerful processors, coordinated through a more complex control mechanism.

In both of the above classes of applications, the main concerns are cost and speed: the hardware should not be prohibitively expensive, and the computation should terminate within an amount of time that is acceptable for the particular application.

A third area of application of parallel, or rather distributed, computation is in information acquisition, information extraction, and control, within geographically distributed systems. An example is a sensor network in which a set of geographically distributed sensors obtain information on the state of the environment and process it cooperatively. Another example is provided by data communication networks in which certain functions of the network (such as correct and timely routing of the messages traveling in the network) have to be controlled in a distributed manner, through the cooperation of the computers residing at the nodes of the network. In this context of distributed computation, the predominant issues are somewhat different from those discussed earlier. Besides cost and speed, there is a more fundamental concern: the distributed system should be able to operate correctly in the presence of limited, sometimes unreliable, communication capabilities, and often in the absence of a central control mechanism.

1.1.2 Parallel Computing Systems and their Classification

We discuss here how technology has responded to the computational needs just mentioned, and we provide a classification of existing systems. An important distinction is between *parallel* and *distributed* computing systems. Roughly speaking, parallel computing systems consist of several processors that are located within a small distance of each other. Their main purpose is to execute jointly a computational task and they have been designed with such a purpose in mind; communication between processors is reliable and predictable. Distributed computing systems are different in a number of

ways. Processors may be far apart, and interprocessor communication is more problematic. Communication delays may be unpredictable, and the communication links themselves may be unreliable. Furthermore, the topology of a distributed system may undergo changes while the system is operating, due to failures or repairs of communication links, as well as due to addition or removal of processors. Distributed computing systems are usually loosely coupled; there is very little, if any, central coordination and control. Each processor may be engaged in its own private activities while at the same time cooperating with other processors in the context of some computational task. Often, the cooperative computation in a distributed computing system is not the raison d'être of the system; for example, a data network exists in order to service some data communication needs, and the distributed computation taking place in the network is only a side activity supporting the main activity. For this reason, while the architecture of a parallel system is typically under the control of a system's designer, the structure of some distributed systems is dictated by exogenous considerations. Our subsequent discussion in this section is geared toward parallel computing systems. A number of issues more relevant to distributed systems will be touched upon in Sections 1.3 and 1.4. Still, there is no clear dividing line between parallel and distributed systems: several algorithmic issues are similar and we will often use the two terms interchangeably.

Traditional serial computers are characterized by the presence of a single locus of control that determines the next instruction to be executed. The data to be operated upon, during the execution of each instruction, are fetched from a global memory, one at time. Thus, only one instruction is executed at a time, while the speed of memory access and the speed of input–output devices can slow down the computation. Several methods have been developed for alleviating these bottlenecks, cache memories and pipelining, for example. The first supercomputers were developed on the basis of such advanced computer architecture designs. By means of intelligent memory organization and use of pipelining, supercomputers have been able to execute vector operations (e.g., addition of two vectors) in time comparable to the time required for scalar operations (e.g., addition of two numbers). Thus, as far as the user is concerned, supercomputers behave as if the components of a vector are operated upon simultaneously. Nevertheless, there seem to be some fundamental limitations to the speed of fast serial computers, notwithstanding the fact that they are very costly.

Parallel computers have deviated from the above described model in a variety of ways. The first such computers consisted of a one– or two–dimensional array of processors, with nearest neighbor interconnections. Such an interconnection pattern is very natural for spatially decomposable problems like PDEs and image processing. Furthermore, there was a host computer overseeing and controlling the progress of the computation by passing to the processors the instruction to be executed next.

Processor arrays are well suited for the applications for which they are designed, but not necessarily for general purpose computation. Thus, more coarse–grained parallel computers have been introduced, in which each processor has considerably more control of its own computations, together with more computational power. Accordingly, the processors in such parallel computers are less tightly coupled. Such systems are sometimes called *multiprocessors*, and they are designed so that they can flexibly support general purpose computation.

Another line of development, resting on recent advances in very large scale integration (VLSI) technology, has led to closely coupled parallel computing systems (all computational elements are often placed on a single chip), designed for a special purpose, such as solving systems of linear equations with special structure, or performing fast Fourier transforms. Here the movement of data is very regular and the traditional notion of a stored program is not quite applicable; in effect, much of the program is encoded in the system hardware. *Systolic arrays* provide a prime example of such computing systems.

Still, the above discussion is too simple to accurately describe the wealth of parallel computers available today. For example, there are systems consisting of a large number of processors connected in some regular fashion, reminiscent of processor arrays, which are also capable of general purpose computation.

There are several parameters that can be used to describe or classify a parallel computer and we refer to these briefly.

(a) *Type and number of processors.* There are parallel computing systems with thousands of processors. Such systems are called *massively parallel*, and hold the greatest promise for significantly extending the range of practically solvable computational problems. A diametrically opposite option is *coarse–grained parallelism*, in which there is a small number of processors, say of the order of 10. In this case, each processor is usually fairly powerful, and the processors are loosely coupled, so that each processor may be performing a different type of task at any given time.

(b) *Presence or absence of a global control mechanism.* Parallel computers almost always have some central locus of control, but the question is one of degree: At what level of detail is the operation of the processors controlled? At one extreme, the global control mechanism is only used to load a program and the data to the processors, and each processor is allowed to work on its own thereafter. At the other extreme, the control mechanism is used to instruct each processor what to do at each step, as in the processor arrays mentioned earlier. Intermediate situations are also conceivable. A related popular classification along these lines distinguishes between SIMD (*Single Instruction Multiple Data*) and MIMD (*Multiple Instruction Multiple Data*) parallel computers, referring to the ability of different processors to execute different instructions at any given point in time.

(c) *Synchronous vs. asynchronous operation.* The distinction here refers to the presence or absence of a common global clock used to synchronize the operation of the different processors. Such synchronization is present in SIMD machines, by definition. Synchronous operation has some desirable properties: the behavior of the processors is much easier to control and algorithm design is considerably simplified. On the other hand, it may require some undesirable overhead and, in some contexts, synchronization may be just impossible. For example, it is quite hard to synchronize a data communication network and, even if this were feasible, it is questionable whether the associated overhead can be justified. Some related issues are discussed in Section 1.4. Finally, it should be noted that a parallel computing system operating asynchronously can simulate a synchronous system (see Section 1.4).

(d) *Processor interconnections.* A significant aspect of parallel computers is the mechanism by which processors exchange information. Generally speaking, there are two extreme alternatives known as *shared memory* and *message–passing* architectures, and a variety of hybrid designs lying in between. The first alternative uses a global shared memory that can be accessed by all processors. A processor can communicate to another by writing into the global memory, and then having the second processor read that same location in the memory. This solves the interprocessor communication problem, but introduces the problem of simultaneous accessing of different locations of the memory by several processors. A common approach for handling memory accesses is based on *switching systems*, such as the one depicted in Fig. 1.1.1. Naturally, the complexity of such switching systems has to increase with the number of processors; this is reflected in longer memory access times. On the other hand, under such an architecture, algorithm design is simplified, because, on a high level, the system behaves as if all processors were directly connected to each other.

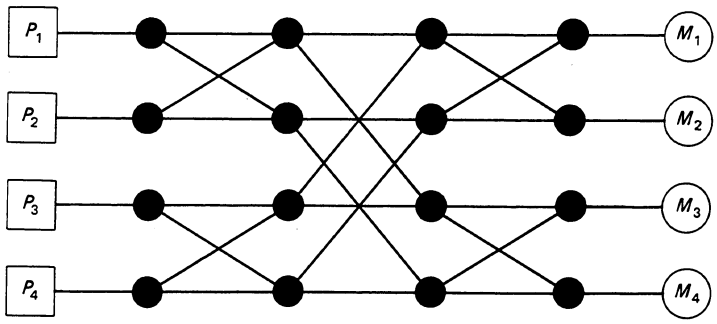


Figure 1.1.1 A switching system connecting processors P_i to memory elements M_i. Here the intermediate nodes correspond to switches. When a message reaches a switch, it can continue on either of the two outgoing links, depending on the destination and the routing algorithm being used. Notice that in this example, there are two alternative paths from each processor to each memory element, used to reduce the probability that two processors simultaneously attempt to utilize the same link. Such redundancy improves reliability, and provides some flexibility which reduces congestion.

In the second major approach, there is no shared memory, but rather each processor has its own *local memory*. (Of course, each processor may have its own local memory even if there is a shared memory.) Processors communicate through an *interconnection network* consisting of direct communication links joining certain pairs of processors, as shown in Fig. 1.1.2. Which processors are connected together is an important design choice. It would be best if all processors were directly connected to each other, but this is often not feasible: either there is an excessive number of links, which leads to increased cost, or the processors communicate through a shared bus, which leads to excessive delays when the number of processors is very large, due to the necessary bus contention.

There are also several possibilities for *hybrid* designs that combine certain features from the different approaches just described. Some examples are shown in Fig. 1.1.3, although several more combinations are possible.

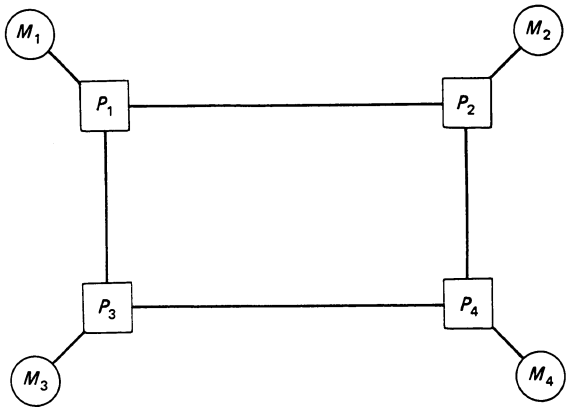


Figure 1.1.2 An interconnection network joining a set of processors P_i, each one endowed with its own local memory M_i.

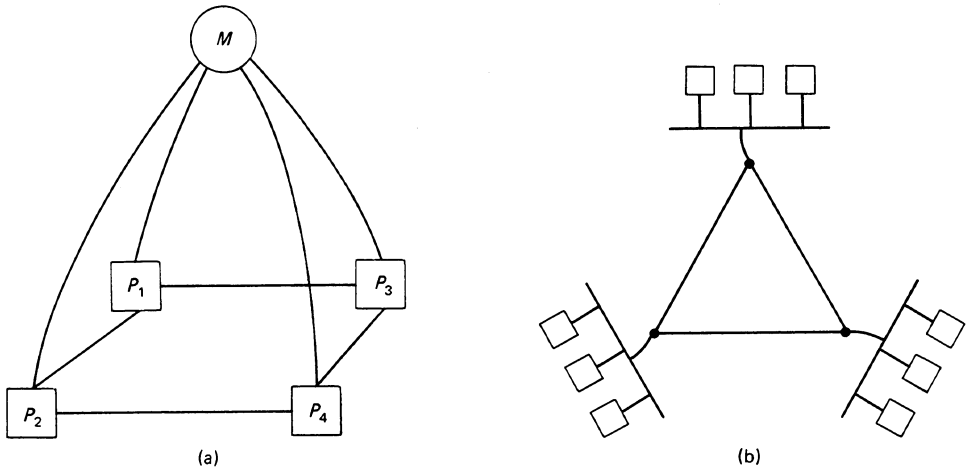


Figure 1.1.3 Examples of hybrid designs: (a) coexistence of a shared memory and a point–to–point network; and (b) clusters of processors: a high speed bus serves for intracluster communication, and an interconnection network is used for intercluster communication.

When distant processors communicate through an interconnection network, there is usually a choice of several paths that can be used. Paths should be chosen so as to avoid underutilization of some at the expense of congestion of others. Furthermore, path selection has to be done locally, by processors that have only partial information on the congestion levels at distant parts of the network. How to do this best is the subject of the distributed routing problem. Routing in interconnection networks is briefly discussed in Subsection 1.3.3, and a particular formulation of the routing problem, relevant to data communication networks, is studied in Chapters 5 and 7.

The structure (topology) of interconnection networks is very important in both parallel and distributed computing systems, but there is an important difference. In parallel computers, the interconnection network is under the control of the designer

and for this reason it is usually designed to be very regular, whereas in some distributed systems, like data communication networks, the topology of the network is predetermined and is usually irregular.

1.2 MODELS, COMPLEXITY MEASURES, AND SOME SIMPLE ALGORITHMS

1.2.1 Models

There is a variety of models of parallel and distributed computation, incorporating different assumptions on the computing power of each individual processor and on the interprocessor information transfer mechanism. For the applications considered in this book, formal models of parallel computation are not essential and we refer the reader to the literature for more detailed expositions (see the notes and sources at the end of the chapter).

Loosely stated, we shall assume that each processor is capable of executing certain basic instructions (such as the basic arithmetic operations, comparisons, branching instructions of the "if ... then" type, etc.), and that there is a mechanism through which processors may exchange information. Concerning the processors' computational power, it will be often assumed that each basic instruction requires one time unit. Concerning information exchange, we shall sometimes make the simplifying assumption that information transfers are instantaneous and cost–free. On other occasions, we shall assume that the processors communicate through a shared memory or by exchanging messages through an interconnection network. In the latter case, more specific assumptions on the delay incurred by messages as they travel through the network will be introduced as needed.

We postpone the discussion of communication issues for Section 1.3. We now describe in some detail a simple model that will be used to illustrate certain key aspects of parallel computation. This model is actually adequate for most of the synchronous algorithms considered in this book, as long as communication issues are ignored.

Representation of Parallel Algorithms by Directed Acyclic Graphs

A *directed acyclic graph* (*DAG*) is a directed graph that has no positive cycles, that is, no cycles consisting exclusively of forward arcs (see Appendix B). A DAG can be used to represent a parallel algorithm, as we proceed to show.

Let $G = (N, A)$ be a DAG, where $N = \{1, \ldots, |N|\}$ is the set of nodes, and A is the set of directed arcs. Each node represents an operation performed by an algorithm, and the arcs are used to represent data dependencies. In particular, an arc $(i, j) \in A$ indicates that the operation corresponding to node j uses the results of the operation corresponding to node i. An operation could be elementary (e.g., an arithmetic or a binary Boolean operation, as shown in Fig. 1.2.1), or it could be a high–level operation like the execution of a subroutine.

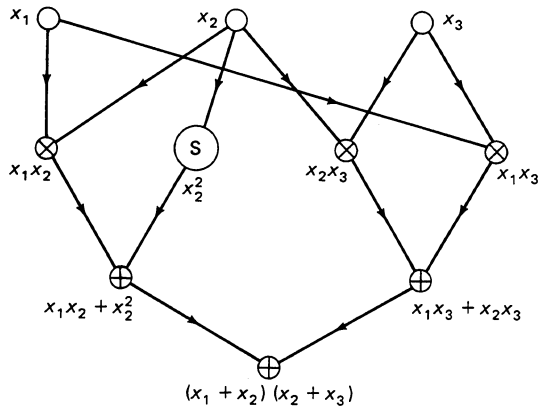


Figure 1.2.1 Representation of an algorithm for evaluating the arithmetic expression $(x_1 + x_2)(x_2 + x_3)$ by means of a DAG. The label at each node indicates the operation corresponding to that node. In particular, the label S stands for squaring.

We introduce some graph–theoretic terminology. We say that node $i \in N$ is a *predecessor* of node $j \in N$ if $(i, j) \in A$. The *in–degree* of node $i \in N$ is the number of predecessors of that node. The *out–degree* of node $i \in N$ is the number of nodes for which i is a predecessor. Nodes with in–degree zero are called *input* nodes and nodes with out–degree zero are called *output* nodes. We use N_0 to denote the set of nodes that are not input nodes. A *positive path* is a sequence $i_0, \ldots, i_K$ of nodes such that $(i_k, i_{k+1}) \in A$ for $k = 0, \ldots, K-1$. The number K is called the *length* of the path. The *depth* of a DAG is defined as the largest length of the positive paths, and is denoted by D. It is seen that D is finite, as a consequence of acyclicity, and that a longest positive path must start at an input node and end at an output node. We assume throughout that G has at least one arc and therefore $D \geq 1$.

Let us denote by x_i the result of the operation corresponding to the ith node in the DAG. Then, the DAG can be viewed as a representation of functional dependencies of the form

$$x_i = f_i\Big(\{x_j \mid j \text{ is a predecessor of } i\}\Big).$$

Here f_i is a function describing the operation corresponding to the ith node. If i is an input node, then x_i does not depend on other variables and is viewed as an external input variable. Thus, the operation corresponding to an input node i essentially amounts to reading the value of the input variable x_i, and we will assume that this takes negligible time. For any node i that is not an input node (i.e., $i \in N_0$), we shall assume that the corresponding operation (that is, the evaluation of the function f_i) takes one time unit. This assumption is reasonable if each node represents an arithmetic operation. However, in more complicated numerical algorithms, the execution times corresponding to different nodes could be widely different. In that case, the assumption of unit time per operation may be considerably violated, with an attendant complication of the scheduling issues discussed below.

A DAG is only a partial representation of an algorithm. It specifies what operations are to be performed, on what operands, and imposes certain precedence constraints on

the order that these operations are to be performed. To determine completely a parallel algorithm we have to specify which processor performs what operation and at what time. Let us assume that we have available a pool of p processors and that each processor is capable of performing any one of the desired operations. For any node i that is not an input node (i.e., $i \in N_0$), let P_i be the processor assigned the responsibility of performing the corresponding operation. Also, for $i \in N_0$, we let t_i be a positive integer variable specifying the time that the operation corresponding to node i is completed. No processors are assigned to input nodes, and we use the convention $t_i = 0$ for every input node i. There are two constraints that have to be imposed:

(a) A processor can perform at most one operation at a time. Thus, if $i \in N_0$, $j \in N_0$, $i \neq j$, and $t_i = t_j$, then $P_i \neq P_j$.

(b) If $(i, j) \in A$, then $t_j \geq t_i + 1$. This requirement reflects the fact that the operation corresponding to node j can only start after the operation corresponding to node i has been completed.

Once P_i and t_i have been fixed, subject to the above constraints, we say that the DAG has been *scheduled* for parallel execution, and we call the set $\{(i, P_i, t_i) \mid i \in N_0\}$ a *schedule*.

The above described setup could correspond to a variety of actual implementations. For example, processor P_i could store the result x_i of its operation in a shared memory from where it can be retrieved by other processors. Alternatively, in a message–passing implementation, processor P_i sends a message with the value of x_i to any processor P_j that needs this value [that is, $(i, j) \in A$]. In practice, a memory access or the transmission of a message may require some time and this has been neglected in our earlier discussion. For example, if a transmission of a message requires exactly τ time units, and if $(i, j) \in A$, then the constraint $t_j \geq t_i + 1$ should be modified to

$$t_j \geq t_i + 1, \qquad \text{if } P_i = P_j,$$

and

$$t_j \geq t_i + \tau + 1, \qquad \text{if } P_i \neq P_j.$$

In fact, even this requirement is rather crude, because the message delay τ may depend on the location of processors P_i and P_j in an interconnection network. In any case, memory access times and message delays are assumed to be negligible in this section and will be addressed in detail in Section 1.3.

1.2.2 Complexity Measures

We first define some notation that is used throughout the text. Let A be some subset of $\Re$ and let $f : A \mapsto \Re$ and $g : A \mapsto \Re$ be some functions. The notation $f(x) = O\big(g(x)\big)$ [respectively, $f(x) = \Omega\big(g(x)\big)$] means that there exists some positive constant

c and some x_0 such that for every $x \in A$ satisfying $x \geq x_0$, we have $|f(x)| \leq cg(x)$ [respectively, $f(x) \geq cg(x)$]. The notation $f(x) = \Theta\big(g(x)\big)$ means that both $f(x) = O\big(g(x)\big)$ and $f(x) = \Omega\big(g(x)\big)$ are true. We also use $\log x$ to denote the logarithm of x with base 2. Thus, $x = 2^{\log x}$ for every nonnegative real number x.

Complexity measures are intended to quantify the amount of computational resources utilized by a parallel algorithm. Some interesting complexity measures are the following:

(a) The number of processors

(b) The time until the algorithm terminates (time complexity)

(c) The number of messages transmitted in the course of the algorithm (communication complexity)

Complexity measures are often expressed as functions of the *size* of the problem being solved, informally defined as the number of inputs to the computation. (For example, in the problem of adding n integers, n is a natural measure of problem size.) If the problem size is held constant, it is still possible that the resources used depend on the actual values of the input variables. The usual approach in this case is to count the amount of resources required in the worst case over all possible choices of data corresponding to a given problem size.

There is a further subtlety in the definition of time complexity. It is conceivable that an algorithm has terminated, meaning that the desired outputs of the computation are available at some processors, but no individual processor is aware of this fact. In such a case, it is natural to count the additional time required for the processors to become aware of termination.

Time Complexity of Algorithms Specified by a DAG

In the case of parallel algorithms specified by a DAG, time complexity is easy to define precisely, as we proceed to show. Let $G = (N, A)$ be a DAG representing some parallel algorithm. Let $\{(i, P_i, t_i) \mid i \in N_0\}$ be a schedule for this DAG that uses p processors. The time spent by such a schedule is equal to $\max_{i \in N} t_i$. We define T_p as the minimum of $\max_{i \in N} t_i$, where the minimum is taken over all possible schedules that use p processors. We view T_p as the time complexity of the algorithm described by G. Note that T_p is a function of the number p of available processors.

We define

$$T_\infty = \min_{p \geq 1} T_p.$$

It is seen that T_p is a nonincreasing function of p, and is bounded below by 0. Since T_p is integer valued, there exists a minimal integer p^* such that $T_p = T_\infty$ for all $p \geq p^*$. We view T_∞ as the time complexity of the algorithm specified by G when a sufficiently large number of processors (at least p^*) is available.

We continue with a few observations. The quantity T_1 is the time needed for a serial execution of the algorithm under consideration. Evidently, T_1 is equal to the

number of nodes in the DAG that are not input nodes. Another important fact is that T_∞ is equal to the depth of the DAG, which we proceed to prove.

Let $i_0, \ldots, i_K$ be a longest positive path in G. Then, node i_0 is an input node and K is equal to the depth D, by the definition of D. For any schedule, we have $t_{i_0} = 0$ and $t_{i_{k+1}} \geq t_{i_k} + 1$ (for $k = 0, \ldots, K-1$), and it follows that $t_{i_K} \geq K = D$. We conclude that $T_\infty \geq D$. For the reverse inequality, we assign a different processor P_i to each node i and we let t_i be the number of arcs in a longest positive path from an input node to node i. (We set $t_i = 0$ if i is itself an input node.) If $(i, j) \in A$ then $t_j \geq t_i + 1$. This is because we can take a longest positive path from an input node to node i and append arc (i, j) to obtain a path to node j. It follows that we have a valid schedule and the corresponding time is $\max_i t_i = D$. Therefore, $T_\infty \leq D$, which proves that $T_\infty = D$.

For an arbitrary value of p, we have $T_1 \geq T_p \geq T_\infty$. The exact value of T_p is not easy to determine, in general. In fact the problem of computing T_p, given a particular DAG and a value of p, is a difficult combinatorial problem. This is not necessarily a concern because, as will be seen, there are some simple useful bounds for T_p.

Properties of T_p

Let us fix a DAG G. Our first result provides a fundamental limitation on the speed of a parallel algorithm.

Proposition 2.1. Suppose that for some output node i, there exists a positive path from every input node to i. Furthermore, suppose that the in–degree of each node is at most 2. Then,

$$T_\infty \geq \log n,$$

where n is the number of input nodes.

Proof. We say that a node j in the DAG depends on k inputs if there exist k input nodes and a positive path from each one of them to node j. (For completeness, we also say that an input node j depends on one input.) We prove, by induction on k, that $t_j \geq \log k$ for every node j depending on k inputs and for every schedule. The claim is clearly true if $k = 1$. Assume that the claim is true for every $k \leq k_0$ and consider a node j that depends on $k_0 + 1$ inputs. Since j can have at most two predecessors, it has a predecessor ℓ that depends on at least $\lceil (k_0 + 1)/2 \rceil$ inputs. Then, using the induction hypothesis,

$$t_j \geq t_\ell + 1 \geq \log \left\lceil \frac{k_0 + 1}{2} \right\rceil + 1 \geq \log(k_0 + 1),$$

and the induction is complete. **Q.E.D.**

The next result expresses the fact that if the number of processors is reduced by a certain factor, then the execution time is increased by at most that factor.

Proposition 2.2. If c is a positive integer and $q = cp$ then $T_p \leq cT_q$.

Proof. Consider a schedule which takes time T_q using q processors. At each stage, at most q operations are performed, and can be carried out in at most $q/p = c$ time units using p processors. We have thus obtained a schedule with p processors which takes at most cT_q time units. **Q.E.D.**

Another useful result is the following:

Proposition 2.3. For every p, we have

$$T_p < T_\infty + \frac{T_1}{p}.$$

Proof. Consider a schedule S for which the execution time is equal to T_∞ and, for every positive integer τ, let n_τ be the number of nodes i for which $t_i = \tau$. We define a new schedule S' that uses only p processors. The schedule S' proceeds in phases. At the τth phase, we perform the operations that were scheduled for time τ under the original schedule S. Given that there are p processors available, the τth phase can be completed in $\lceil n_\tau/p \rceil$ time units. Since T_p cannot be larger than the time required by schedule S', we obtain

$$T_p \leq \sum_{\tau=1}^{T_\infty} \left\lceil \frac{n_\tau}{p} \right\rceil < \sum_{\tau=1}^{T_\infty} \left(\frac{n_\tau}{p} + 1 \right) = \frac{T_1}{p} + T_\infty,$$

where we have used the fact that $\sum_{\tau=1}^{T_\infty} n_\tau$ is equal to T_1, the total number of nodes in the DAG that are not input nodes. **Q.E.D.**

The following result is a corollary of Prop. 2.3.

Proposition 2.4. (a) If $p \geq T_1/T_\infty$, then $T_p < 2T_\infty$. More generally, if $p = \Omega(T_1/T_\infty)$, then $T_p = O(T_\infty)$.
(b) If $p \leq T_1/T_\infty$, then

$$\frac{T_1}{p} \leq T_p < 2\frac{T_1}{p}.$$

More generally, if $p = O(T_1/T_\infty)$, then $T_p = \Theta(T_1/p)$.

Proof. (a) If $p \geq T_1/T_\infty$ [respectively, $p = \Omega(T_1/T_\infty)$], then $T_1/p \leq T_\infty$ [respectively, $T_1/p = O(T_\infty)$], and the result follows from Prop. 2.3.
(b) If $p \leq T_1/T_\infty$ [respectively, $p = O(T_1/T_\infty)$], then $T_\infty \leq T_1/p$ [respectively, $T_\infty = O(T_1/p)$], and Prop. 2.3 yields $T_p < 2T_1/p$ [respectively, $T_p = O(T_1/p)$]. Furthermore, Prop. 2.2 yields $T_1 \leq pT_p$, from which we obtain $T_p \geq T_1/p = \Omega(T_1/p)$. **Q.E.D.**

The last two results are of fundamental importance. They establish that although T_∞ is defined under the assumption of an unlimited number of processors, $\Omega(T_1/T_\infty)$ processors are actually sufficient to come within a constant factor of T_∞ [Prop. 2.4(a)]. Furthermore, a corresponding schedule is obtained by simply modifying an optimal schedule for the case of an unlimited number of processors (see the proof of Prop. 2.3), as opposed to solving a generally difficult scheduling problem. This suggests a methodology whereby we first develop a parallel algorithm as if an unlimited number of processors were available, and then adapt the algorithm to the available number of processors. The significance of Prop. 2.4(b) is that as long as $p = O(T_1/T_\infty)$, the availability of p processors allows us to speed up the computation by a factor proportional to p, which is the best possible. We thus see that for a number of processors nearly equal to T_1/T_∞, we obtain both optimal execution time and optimal speeding up of the computation (within constant factors).

Finding an Optimal DAG

It is seen that there can be several DAGs corresponding to different algorithms for the same computational problem (see Fig. 1.2.2). It may then be of interest to find a DAG for which T_p is minimized, where p is the number of available processors. Let us denote by T_p^* the value of T_p corresponding to such an optimal DAG and view it as the optimal parallel time, using p processors, for the computational problem under consideration. The value of T_p^* is a measure of the complexity of the problem, as opposed to T_p which is the complexity of a particular algorithm.

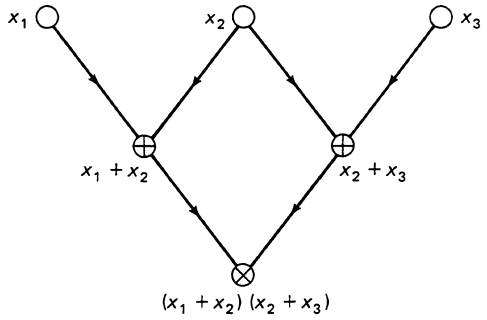


Figure 1.2.2 Another DAG representing an algorithm for evaluating the arithmetic expression $(x_1 + x_2)(x_2 + x_3)$. We have $T_1 = 3$ and $T_\infty = D = 2$. This should be contrasted with the DAG of Fig. 1.2.1 which solves the same computational problem and for which $T_1 = 7$ and $D = 3$. We conclude that the DAG given here represents a better parallel algorithm.

An explicit evaluation of T_p^* is usually very difficult. However, for several interesting classes of problems, there exist methods for constructing DAGs that come within a constant factor of the optimal. We do not pursue this issue any further and refer the reader to the notes and sources at the end of this chapter.

Speedup and Efficiency

We now assume that a particular model of parallel computation has been chosen. This could be the DAG model considered earlier, or any other model. Let us consider a computational problem parametrized by a variable n representing problem size. (In the DAG model, different problem sizes correspond to different numbers of input variables.

Thus, properly speaking, an algorithm is not specified by a single DAG, but rather by a family of DAGs, one for each problem size.) Time complexity is generally dependent on n, and we incorporate this dependence in our notation.

We describe a few concepts that are sometimes useful in comparing serial and parallel algorithms. Suppose that we have a parallel algorithm that uses p processors (p may depend on n), and that terminates in time $T_p(n)$. Let $T^*(n)$ be the optimal serial time to solve the same problem, that is, the time required by the best possible serial (uniprocessor) algorithm for this problem. The ratio

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

is called the *speedup* of the algorithm, and describes the speed advantage of the parallel algorithm, compared to the best possible serial algorithm. The ratio

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^*(n)}{pT_p(n)}$$

is called the *efficiency* of the algorithm, and essentially measures the fraction of time that a typical processor is usefully employed. Ideally, $S_p(n) = p$ and $E_p(n) = 1$, in which case, the availability of p processors allows us to speed up the computation by a factor of p. For this to occur, the parallel algorithm should be such that no processor ever remains idle or does any unnecessary work. This ideal situation is practically unattainable. A more realistic objective is to aim at an efficiency that stays bounded away from zero, as n and p increase.

There is a difficulty with the above definitions because the optimal serial time $T^*(n)$ is unknown, even for seemingly simple computational problems like matrix multiplication. For this reason, $T^*(n)$ is sometimes defined differently. Some alternatives are the following:

(a) Let $T^*(n)$ be the time required by the best existing serial algorithm.

(b) Let $T^*(n)$ be the time required by a benchmark serial algorithm. For example, for multiplication of two dense $n \times n$ matrices, $\Theta(n^3)$ is a reasonable benchmark, even though there exist algorithms with substantially smaller time requirements [AHU74].

(c) Finally, we may let $T^*(n)$ be the time required by a single processor to execute the particular parallel algorithm being analyzed. (That is, we let a single processor simulate the operation of the p parallel processors.) With this choice of $T^*(n)$, efficiency relates to how well a particular algorithm has been parallelized, but provides no information on the absolute merits of the algorithm [in contrast with our earlier definitions of $T^*(n)$].

Notice that if $T^*(n)$ is defined as in (c), and if algorithms are specified by means of the DAG model, then $T^*(n)$ coincides with $T_1(n)$. In particular, if $p \leq O\big(T_1(n)/T_\infty(n)\big)$, then $T_p(n) = \Theta\big(T_1(n)/p\big)$ [Prop. 2.4(b)] and

$$E_p(n) = \frac{T_1(n)}{pT_p(n)} = \Theta(1).$$

This shows that if the number of processors is suitably small, then efficient parallel implementations are possible. Furthermore, if $p = \Theta\big(T_1(n)/T_\infty(n)\big)$, we also have $T_p(n) = \Theta\big(T_\infty(n)\big)$ [Prop. 2.4(a)] and we have a parallel implementation that is both efficient and has a time complexity within a constant factor from the optimum.

The above discussion suggests that efficiency of parallel implementation is not a concern, at least when an algorithm is specified by a DAG, and as long as communication issues are ignored. A more fundamental issue is whether the maximum attainable speedup $T_1(n)/T_\infty(n)$ can be made arbitrarily large, as n is increased. In certain applications, the required computations are quite unstructured, and there has been considerable debate on the range of achievable speedups in real world situations. The main difficulty is that some programs have some sections that are easily parallelizable, but also have some sections that are inherently sequential. When a large number of processors is available, the parallelizable sections are quickly executed, but the sequential sections lead to bottlenecks. This observation is known as *Amdahl's law* and can be quantified as follows: if a program consists of two sections, one that is inherently sequential and one that is fully parallelizable, and if the inherently sequential section consumes a fraction f of the total computation, then the speedup is limited by

$$S_p(n) \leq \frac{1}{f + (1-f)/p} \leq \frac{1}{f}, \qquad \forall p.$$

On the other hand, there are numerous computational problems for which f decreases to zero as the size of the problem increases, and Amdahl's law is not a concern.

1.2.3 Examples: Vector and Matrix Computations

In this subsection, we consider some elementary but very common numerical computational tasks, present some simple parallel algorithms, and discuss their complexity and efficiency. All of the algorithms to be considered can be represented by DAGs and such representations will be occasionally employed. It is assumed that each addition or multiplication takes unit time and that processors are able to instantly exchange intermediate results. In practice, processors may be communicating through an interconnection network or through a shared memory and our analysis ignores the associated communication and memory access delays. Nevertheless, the algorithms considered here are simple enough so that they can be implemented in some architectures with negligible communication overhead. The communication aspects of such implementations will be discussed in Subsections 1.3.4 to 1.3.6.

Scalar Addition

The simplest computational task is the addition of n scalars. It is clear that the best serial algorithm requires $n-1$ operations. Thus, $T^*(n) = n-1$. We now present a parallel

algorithm under the simplifying assumption that n is a power of 2. We partition the n scalars into $n/2$ disjoint pairs and we use $n/2$ different processors to add the two scalars in each pair. Thus, after one time unit, we are left with the task of adding only $n/2$ scalars. Continuing similarly, after $\log n$ stages, we are left with a single number and the computation terminates (see Fig. 1.2.3). This algorithm generalizes easily to the case where n is not a power of 2: the execution time becomes $\lceil \log n \rceil$ using $\lfloor n/2 \rfloor$ processors (see Fig. 1.2.3).

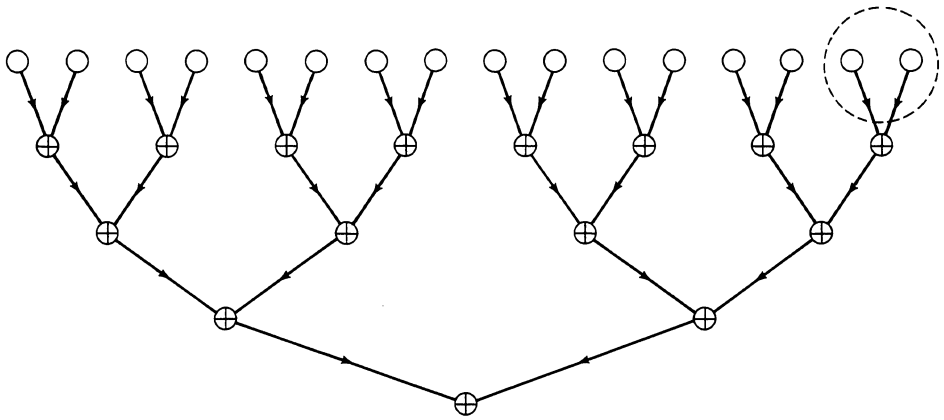


Figure 1.2.3 Parallel computation of the sum of 16 scalars. Eight processors are needed for the parallel additions at the first stage and a total of $4 = \log 16$ stages are needed. If the portion of the diagram enclosed in the dashed circle is removed, we obtain an algorithm for the parallel addition of 15 scalars. Notice that now only $7 = \lfloor 15/2 \rfloor$ processors are needed.

The efficiency of the above algorithm is

$$\frac{n-1}{\lfloor n/2 \rfloor \lceil \log n \rceil},$$

which goes to zero as n increases. An alternative parallel algorithm is obtained as follows (see Fig. 1.2.4). We assume for simplicity that $\log n$ is an integer, and that $n/\log n$ is an integer and a power of 2. We split the n numbers into $n/\log n$ groups of $\log n$ numbers each. We use $n/\log n$ processors and the ith processor adds the numbers in the ith group; this task takes time $\log n - 1$. We are then left with the task of adding $n/\log n$ numbers. This can be accomplished by our previous parallel algorithm in time $\log(n/\log n) \leq \log n$, using $n/(2\log n)$ processors. This two–phase algorithm requires time approximately equal to $2\log n$ (the speed is reduced by a factor of 2), but uses only $n/\log n$ processors and therefore its efficiency is approximately equal to 1/2. Notice that we have chosen the number of processors p to be approximately equal to $T_1(n)/T_\infty(n)$. As discussed earlier, such a choice always leads to efficient algorithms. This example illustrates that with a small sacrifice in speed, efficiency can be substantially improved.

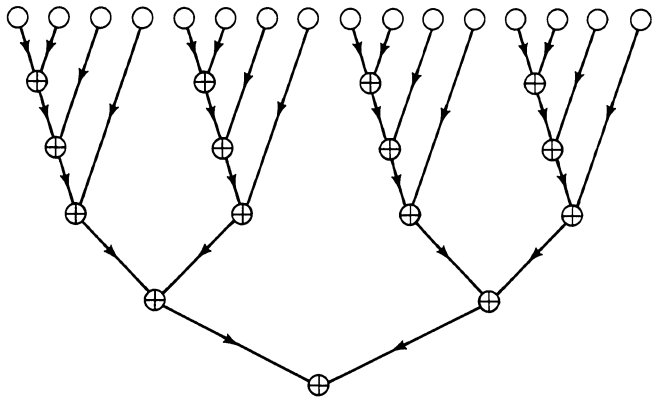


Figure 1.2.4 An alternative algorithm for the parallel addition of 16 scalars. Only four processors are used and the time requirements increase to 5 stages. Overall, however, there is an efficiency improvement over the algorithm of Fig. 1.2.3.

In fact, it will be seen later (Subsection 1.3.5) that decreasing the number of processors can also substantially decrease the communication requirements of an algorithm.

Inner Products

The inner product $\sum_{i=1}^{n} x_i y_i$ of two n–dimensional vectors can be computed in time $\lceil \log n \rceil + 1$ using n processors as follows: at the first step, each processor i computes the product $x_i y_i$ and then the $\lceil \log n \rceil$ time algorithm for scalar addition is used.

Matrix Addition and Multiplication

The sum of n matrices of dimensions $m \times m$ can be computed in time $\lceil \log n \rceil$ using $m^2 \lfloor n/2 \rfloor$ processors by letting a different group of $\lfloor n/2 \rfloor$ processors compute a different entry of the sum. Similarly, multiplication of two matrices of dimensions $m \times n$ and $n \times \ell$ consists of the evaluation of $m\ell$ inner products of n–dimensional vectors and can be therefore accomplished in time $\lceil \log n \rceil + 1$ using $nm\ell$ processors. In the case where $n = m = \ell$, the processor requirements become n^3. The corresponding number $T_1(n)$ of operations is $\Theta(n^3)$. In fact, there exist more economical algorithms for matrix multiplication in terms of processor requirements, or in terms of $T_1(n)$, but they are somewhat impractical and will not be considered here.

Powers of a Matrix

Suppose now that A is an $n \times n$ matrix and that we wish to compute A^k for some integer k. If k is a power of 2, this can be accomplished by repeated squaring: we first compute A^2; we then compute $A^2 A^2 = A^4$, etc. After $\log k$ stages, A^k is obtained. This procedure involves $\log k$ consecutive matrix multiplications and can be therefore carried out in time $\log k(\lceil \log n \rceil + 1)$ using n^3 processors. A simple modification of this procedure can be used to compute A^k in time $\Theta(\log k \cdot \log n)$ even if k is not a power of 2 (Exercise 2.4).

A consequence of the above discussion is that all the powers $A^2, \ldots, A^n$ of an $n \times n$ matrix can be computed in time $\Theta(\log^2 n)$ using n^4 processors by using a different group of n^3 processors for the computation of each power A^k. An alternative method for computing the powers $A^2, \ldots, A^n$, which avoids unnecessary duplication of computational effort, is shown in Fig. 1.2.5.

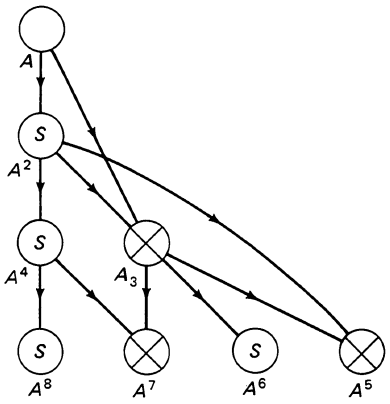


Figure 1.2.5 Parallel computation of the powers $A^2, \ldots, A^n$ of an $n \times n$ matrix A. A node with a label S represents a matrix squaring operation. At the first stage, A^2 is computed. At the second stage, A^3 and A^4 are computed by multiplying earlier computed matrices. More generally, at the kth stage, the matrices $A^{2^{k-1}+1}, \ldots, A^{2^k}$ are computed. Thus, $\Theta(\log n)$ stages suffice for the computation of $A^2, \ldots, A^n$. Each stage involves at most $\Theta(n)$ simultaneous matrix multiplications and can be carried out using n^4 processors in time $\Theta(\log n)$, leading to an overall time $\Theta(\log^2 n)$.

In the previously discussed algorithms, we have strived for the fastest possible execution times; such an approach often leads to excessive processor requirements and low efficiency. On the other hand, as discussed earlier, the same algorithms can be made efficient if the number of processors is chosen so that $p = O\big(T_1(n)/T_\infty(n)\big)$. For example, the product of two $n \times n$ matrices can be computed in time $\Theta(\log n)$ using $\Theta(n^3/\log n)$ processors, and the corresponding efficiency is $\Theta(1)$. If the number of processors is reduced even further, the execution time will be $\Theta\big(T_1(n)/p\big)$ [Prop. 2.4(b)]. Thus, two $n \times n$ matrices can be multiplied in time $\Theta(n)$ when n^2 processors are used, and in time $\Theta(n^2)$ when n processors are used.

1.2.4 Parallelization of Iterative Methods

Many interesting algorithms for the solution of systems of equations, optimization, and other problems have the structure

$$x(t+1) = f\big(x(t)\big), \qquad t = 0, 1, \ldots, \tag{2.1}$$

where each $x(t)$ is an n–dimensional vector, and f is some function from $\Re^n$ into itself. (Several examples will be seen in Chapters 2 and 3.) They are called *iterative* algorithms or, in certain contexts, *relaxation* methods. An alternative notation that is sometimes used in place of Eq. (2.1) is $x := f(x)$. Notice that if the sequence $\{x(t)\}$ generated by the above iteration converges to a limit x^*, and if the function f is continuous, then x^* is a *fixed point* of f, that is, it satisfies $x^* = f(x^*)$. A common special case arises when the function f is of the form $f(x) = Ax + b$, where A is a square matrix and b is a vector,

in which case we are dealing with a *linear* iterative algorithm. In this subsection, we make some general observations on the possibilities for parallel execution of iterative algorithms. It should be mentioned here that the concept of time complexity is not quite relevant to algorithms of the form $x := f(x)$ unless a termination criterion is also specified.

Let $x_i(t)$ denote the ith component of $x(t)$ and let f_i denote the ith component of the function f. Then, we can write $x(t+1) = f\big(x(t)\big)$ as

$$x_i(t+1) = f_i\big(x_1(t), \ldots, x_n(t)\big), \qquad i = 1, \ldots, n. \tag{2.2}$$

The iterative algorithm $x := f(x)$ can be parallelized by letting each one of n processors update a different component of x according to Eq. (2.2). At each stage, the ith processor knows the value of all components of $x(t)$ on which f_i depends, computes the new value $x_i(t+1)$, and communicates it to other processors in order to start the next iteration.

The communication required for the execution of iteration (2.2) can be compactly described by means of a directed graph $G = (N, A)$, called the *dependency graph.* The set of nodes N is $\{1, \ldots, n\}$, corresponding to the components of x. For any two distinct nodes i and j, we let (i, j) be an arc of the dependency graph if and only if the function f_j depends on x_i, that is, if and only if processor i needs to communicate the values of $x_i(t)$ to processor j (see Fig. 1.2.6).

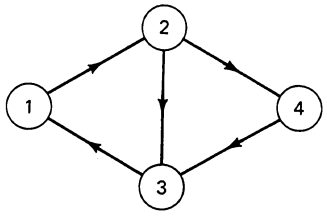


Figure 1.2.6 The dependency graph associated with an iteration of the form

$$\begin{aligned} x_1(t+1) &= f_1\big(x_1(t), x_3(t)\big) \\ x_2(t+1) &= f_2\big(x_1(t), x_2(t)\big) \\ x_3(t+1) &= f_3\big(x_2(t), x_3(t), x_4(t)\big) \\ x_4(t+1) &= f_4\big(x_2(t), x_4(t)\big). \end{aligned}$$

Assuming that the iteration (2.1) is to be carried out only for $t = 0, 1, \ldots, T$, where T is some positive integer, the structure of the algorithm can be represented by means of a DAG. This DAG is essentially an "unfolding" in time of the above defined dependency graph (see Fig. 1.2.7).

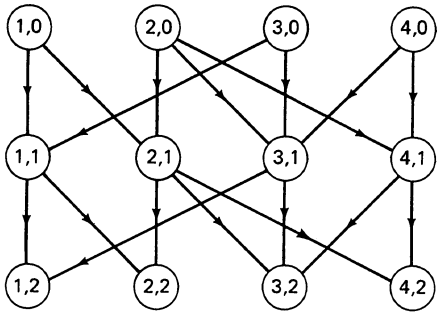


Figure 1.2.7 The DAG corresponding to two iterations when the function f has the dependency graph shown in Fig. 1.2.6. The nodes of the DAG are of the form (i, t), where $i \in \{1, \ldots, n\}$, and t is the iteration count. The arcs are of the form $\big((i, t), (j, t+1)\big)$, where (i, j) is an arc of the dependency graph or $i = j$.

Sometimes we may wish to employ a coarse–grained parallelization of the iteration $x := f(x)$. In particular, we decompose the vector space $\Re^n$ as a Cartesian product of lower dimensional subspaces $\Re^{n_j}$, $j = 1, \ldots, p$, where $\sum_{j=1}^{p} n_j = n$. Accordingly, any vector $x \in \Re^n$ is decomposed as $x = (x_1, \ldots, x_j, \ldots, x_p)$, where each x_j is itself an n_j–dimensional vector, called a *block–component* of x, or simply a component when no confusion can arise. Similarly, the iteration $x(t+1) = f\big(x(t)\big)$ can be written as

$$x_j(t+1) = f_j\big(x(t)\big), \qquad j = 1, \ldots, p, \tag{2.3}$$

where each f_j is a vector function mapping $\Re^n$ into $\Re^{n_j}$. We assign each one of p processors to update a different block–component according to Eq. (2.3), and the resulting parallel algorithm is said to be *block–parallelized.* A dependency graph $G = (N, A)$ can be again defined with $N = \{1, \ldots, p\}$ and $A = \{(i, j) \mid f_j \text{ depends on } x_i\}$. There are several reasons for being interested in block–parallelization. First, there may be too few processors available, so that we have to assign more than one component to each one of them. Second, certain scalar functions f_i may involve common computations, in which case it is natural to group them together. Finally, as will be discussed in Subsection 1.3.5, block–parallelization reduces the communication requirements of an algorithm.

In general, a parallelization that assigns the update of different components to different processors is meaningful when the computations involved in the update of each x_i are different, but could be wasteful otherwise. For example, suppose that each f_i is of the form

$$f_i(x_1, \ldots, x_n) = x_i + \left(\sum_{j=1}^{n} x_j^2 \right)^{1/2}.$$

In this example, it is clearly wasteful to let all processors simultaneously compute the value of $\left(\sum_{i=1}^{n} x_i^2\right)^{1/2}$. This could be done by a single processor that subsequently communicates the result to the others. Even better, the processors could cooperate in the computation of this quantity, using, for example, the methods of Subsection 1.2.3. Nevertheless, in many cases, the evaluation of each f_i involves very little or no duplication of effort, and this is the situation in which we are mostly interested.

Gauss–Seidel Iterations

Iteration (2.2), in which all of the components of x are simultaneously updated, is sometimes called a *Jacobi*–type iteration. In an alternative form, the components of x are updated one at a time, and the most recently computed values of the other components are used. That is, Eq. (2.2) is changed to

$$x_i(t+1) = f_i\Big(x_1(t+1), \ldots, x_{i-1}(t+1), x_i(t), \ldots, x_n(t)\Big), \qquad i = 1, \ldots, n. \tag{2.4}$$

The iteration (2.4) is called the *Gauss–Seidel algorithm based on the function* f. Gauss–Seidel algorithms are often preferable: they incorporate the newest available information,

and for this reason, they sometimes converge faster than the corresponding Jacobi–type algorithms. (A result of this type will be proved in Section 2.6.)

From now on, we concentrate on a single Gauss–Seidel iteration (sometimes called a *sweep*), and investigate its parallelization potential. A Gauss–Seidel iteration may be completely non–parallelizable. For example, if every function f_i depends on all components x_j, then only one component can be updated at a time. On the other hand, when the dependency graph is sparse, it is possible that certain component updates can be performed in parallel. An example is shown in Fig. 1.2.8.

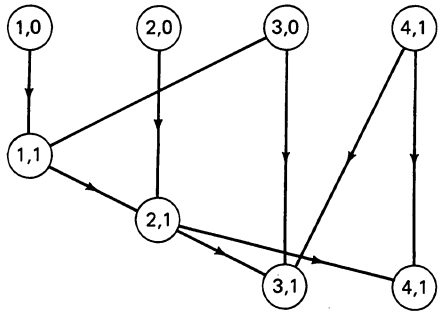


Figure 1.2.8 Illustration of the parallelization of Gauss–Seidel iterations. Let f be a function whose dependency graph is as in Fig. 1.2.6. The Gauss–Seidel algorithm based on f takes the form

$$
\begin{aligned}
x_1(t+1) &= f_1\big(x_1(t), x_3(t)\big) \\
x_2(t+1) &= f_2\big(x_1(t+1), x_2(t)\big) \\
x_3(t+1) &= f_3\big(x_2(t+1), x_3(t), x_4(t)\big) \\
x_4(t+1) &= f_4\big(x_2(t+1), x_4(t)\big).
\end{aligned}
$$

The DAG shown illustrates the data dependencies in one iteration of the Gauss–Seidel algorithm. There are four updates to be performed, but the depth of the DAG is only 3. In particular, it is seen that x_3 and x_4 can be updated in parallel.

We notice that there are several alternative Gauss–Seidel algorithms corresponding to the same function f, because there is freedom in choosing the order in which the components are to be updated. For example, we might wish to update the components of x starting with x_n and proceeding backwards, with x_1 being updated last. Different updating orders, strictly speaking, correspond to different algorithms and the results produced are generally different. Nevertheless, in several applications, a Gauss–Seidel algorithm converges in the limit of a large number of iterations to the same value, irrespective of the updating order. As long as the speed of convergence corresponding to different updating orders is not drastically different, it is natural to choose an ordering for which the parallelism in each iteration is maximized (see Fig. 1.2.9).

We now develop a graph–theoretic formulation of the problem of finding an updating order that minimizes the parallel time needed for a sweep. Given the dependency graph $G = (N, A)$, a *coloring* of G, using K colors, is defined as a mapping $h : N \mapsto \{1, \ldots, K\}$ that assigns a "color" $k = h(i)$ to each node $i \in N$. The idea is that similarly colored variables will be updated in parallel. The following result shows that maximizing parallelism is equivalent to an "optimal coloring" problem.

Proposition 2.5. The following are equivalent:

(i) There exists an ordering of the variables such that a sweep of the corresponding Gauss–Seidel algorithm can be performed in K parallel steps.

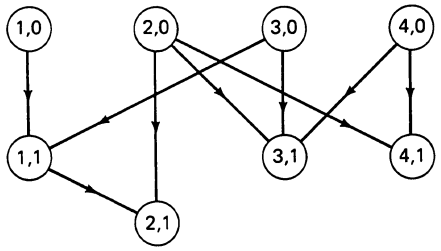


Figure 1.2.9 Illustration of the increase in parallelism when the updating order is changed. Here the function f is the same as in Figs. 1.2.6 to 1.2.8. We consider the following updating order:

$$x_1(t+1) = f_1\big(x_1(t), x_3(t)\big)$$
$$x_3(t+1) = f_3\big(x_2(t), x_3(t), x_4(t)\big)$$
$$x_4(t+1) = f_4\big(x_2(t), x_4(t)\big)$$
$$x_2(t+1) = f_2\big(x_1(t+1), x_2(t)\big).$$

The DAG shown illustrates the data dependencies in a typical iteration. Its depth is only 2. It is seen that a sweep can be executed in parallel in two time steps using two processors.

(ii) There exists a coloring of the dependency graph that uses K colors and with the property that there exists no positive cycle with all nodes on the cycle having the same color.

Proof. We first show that (i) implies (ii). Consider an ordering of the variables with which a Gauss–Seidel iteration takes K parallel steps. We define $h(i)$, the color of node i, to be equal to k if the variable x_i is updated at the kth parallel stage. Consider a positive cycle $i_1, i_2, \ldots, i_m$, with $i_m = i_1$. Let us choose the node i_ℓ in the cycle $(1 \leq \ell < m)$ that comes first in the assumed ordering. Since $i_{\ell+1}$ is ordered after i_ℓ, and since $(i_\ell, i_{\ell+1}) \in A$, the variable $x_{i_{\ell+1}}(t+1)$ depends on $x_{i_\ell}(t+1)$. It follows that x_{i_ℓ} and $x_{i_{\ell+1}}$ cannot be updated simultaneously and we have $h(i_\ell) \neq h(i_{\ell+1})$. This shows that in every positive cycle, there exist two nodes with different colors and (ii) has been proved.

We now prove an auxiliary result. We show that if G is a directed acyclic graph, then its nodes can be ordered so that if $(i,j) \in A$, then j comes before i. The proof is as follows. For each node i, we let d_i be the largest possible number of arcs in a positive path that starts at i. (We let $d_i = 0$ if node i has no outgoing arcs.) It is seen that d_i is finite as a consequence of acyclicity. We then order the nodes in order of increasing values of d_i. (Ties among nodes with the same value of d_i are broken arbitrarily.) We see that if $(i,j) \in A$, then $d_i > d_j$. [This is because we can take a longest path starting from j and append the arc (i,j) to obtain an even longer path starting from i.] We conclude that j comes before i whenever $(i,j) \in A$, as desired.

We now assume that (ii) holds. Let h be a coloring with K colors and with no positive cycle in which all nodes have the same color. For every color k, let G_k be the subgraph of G obtained by keeping only the nodes with color k and the arcs joining them. Each G_k is acyclic and, according to the result of the preceding paragraph, the nodes in G_k can be ordered so that j comes before i whenever $(i,j) \in A$. We order the nodes in G in order of increasing color; ties between nodes with the same color k are broken by using the ordering of the graph G_k. Consider the Gauss–Seidel iteration corresponding to this ordering. Let i and j be two distinct nodes with the same color k. If $(i,j) \notin A$ and $(j,i) \notin A$, then x_i and x_j can be clearly updated in parallel. The case

where $(i,j) \in A$ and $(j,i) \in A$ is impossible because G_k is acyclic. If $(i,j) \in A$ and $(j,i) \notin A$, then j appears before i in the order we have constructed and therefore the computation of $x_j(t+1)$ only requires the value of $x_i(t)$ and not the value of $x_i(t+1)$. Finally, the case where $(j,i) \in A$ and $(i,j) \notin A$ is similar. We conclude that every x_i with the same color can be updated in parallel, thus proving (i). **Q.E.D.**

For the dependency graph of Fig. 1.2.6 two colors suffice. In particular, we may let $h(1) = h(3) = h(4) = 1$ and $h(2) = 2$. Since every positive cycle goes through node 2, there exists no positive cycle with all nodes having the same color, as required. In particular, the subgraph G_1 in which only the nodes with color 1 are kept is acyclic. With d_i defined as in the proof of Prop. 2.5, we have $d_1 = 0$, $d_3 = 1$, and $d_4 = 2$. The ordering of the variables constructed in that proof is $(1,3,4,2)$, and the corresponding Gauss–Seidel iteration is precisely the one shown in Fig. 1.2.9. According to Prop. 2.5, this ordering requires the least possible number of parallel stages per sweep, a fact that is easy to verify directly for this particular example.

Proposition 2.5 can be somewhat simplified in the case where the dependency graph has a certain symmetry property.

Proposition 2.6. Suppose that $(i,j) \in A$ if and only if $(j,i) \in A$. Then, the following are equivalent:

(i) There exists an ordering of the variables such that a sweep of the corresponding Gauss–Seidel algorithm can be performed in K parallel steps.

(ii) There exists a coloring of the dependency graph that uses at most K colors and such that adjacent nodes have different colors [that is, if $(i,j) \in A$, then $h(i) \neq h(j)$].

Proof. It is sufficient to show that condition (ii) of this proposition is equivalent to condition (ii) of Prop. 2.5. Suppose that there exists a positive cycle with all the nodes on that cycle having the same color. Then there exist two adjacent nodes with the same color. Conversely, if $(i,j) \in A$, then $(j,i) \in A$ and the two arcs (i,j) and (j,i) form a positive cycle. Thus, if two neighboring nodes have the same color, there exists a positive cycle with all nodes on that cycle having the same color. This proves the equivalence of the two conditions and concludes the proof. **Q.E.D.**

Unfortunately, the optimal coloring problems of Props. 2.5 and 2.6 are intractable (NP–complete): there is no known efficient algorithm for solving them, neither is it likely that an efficient algorithm will be found [GaJ79]. Nevertheless, problems arising in practice often have a special structure and a coloring with relatively few colors can sometimes be found by inspection. Some interesting cases are the following:

(a) Consider the undirected graph $\tilde{G}$ obtained by ignoring the orientation of the arcs of G. If $\tilde{G}$ is a tree, then two colors suffice: we choose an arbitrary node in the tree and we assign color 1 (respectively, 2) to all nodes in the graph that can be reached by traversing an even (respectively, odd) number of arcs.

(b) If every node in G has at most D neighbors, then $D + 1$ colors suffice. To see this, we color the nodes one by one. Assuming that the first i nodes have been colored, we consider node $i + 1$. Since we are using $D + 1$ colors, there is some color that can be used for node $i + 1$ while ensuring that it is colored differently from the already colored neighbors of $i + 1$.

When a coloring scheme is used for the parallel implementation of a Gauss–Seidel algorithm, it is wasteful to assign a different processor to each component of x, because each processor will be idle while variables of different colors are being updated. The obvious remedy is to use fewer processors, with each processor being assigned several variables with different associated colors.

Example 2.1. *Red–Black Coloring of a Two–Dimensional Array*

In a variety of iterative algorithms of the form $x := f(x)$, employed for the numerical solution of partial differential equations or in image processing, each component of the vector x is associated with a particular point in a certain region of two–dimensional space. For example, let N be the set of all points $(i, j) \in \Re^2$, such that i and j are integers satisfying $0 \leq i \leq M$ and $0 \leq j \leq M$. Let x_{ij} be the component of the vector x corresponding to point (i, j). By connecting nearest neighbors, we form a graph $G = (N, A)$, as illustrated in Fig. 1.2.10. We view G as a directed graph, by making the arcs bidirectional, and we assume that it is the dependency graph associated to the iteration $x := f(x)$. Parallel execution of this iteration, in Jacobi fashion, is straightforward. We assign a different processor to each point (i, j). This processor is responsible for updating x_{ij} and, in order to do so, only needs to know the values of the components of x associated with neighboring points. Thus, it is most natural to assume that processors associated with neighboring points are joined by a direct communication link.

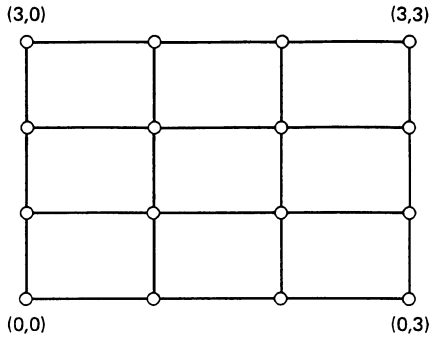


Figure 1.2.10 A common dependency graph associated with iterative algorithms arising in the solution of partial differential equations and in image processing.

Concerning the implementation of the associated Gauss–Seidel method, we notice that the graph of Fig. 1.2.10 can be colored using only two colors, as indicated in Fig. 1.2.11. If we were to assign one processor to each component x_{ij}, each processor would be idle half of the time. It is thus reasonable to assign two components with different corresponding colors to each processor. As shown in Fig. 1.2.11 this can be done while preserving the property that only nearest neighbors have to communicate to each other. In practice, the number of points involved is often large enough so that each processor is assigned more

than two components of x. The coloring indicated in Fig. 1.2.11 is commonly known as *red–black* coloring and the associated Gauss–Seidel algorithm is known as *Gauss–Seidel with red–black ordering*.

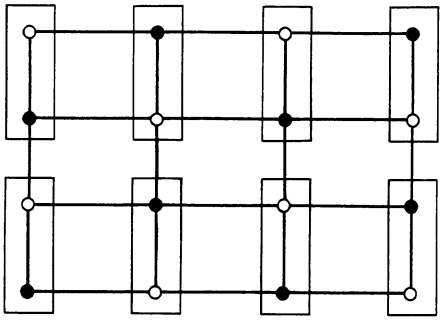


Figure 1.2.11 The nodes of the graph in Fig. 1.2.10 can be colored using two colors only. We can assign a pair of nodes with different colors to each processor and notice that only neighboring processors have to communicate.

EXERCISES

2.1. Formulate and prove a generalization of Prop. 2.1 under the assumption that the in–degree of each node is at most B, where B is some positive integer.

2.2. Prove Amdahl's law. How should $T^*(n)$ be defined for this law to hold?

2.3. **(Prefix Problem [LaF80].)**

(a) Let $a_1, a_2, \ldots, a_n$ be given scalars. Provide an $O(\log n)$ time algorithm which uses $O(n)$ processors and evaluates all products of the form $\prod_{i=1}^{k} a_i$, where $k = 1, 2, \ldots, n$.

(b) Generalize part (a) to provide an $O(\log n \cdot \log m)$ time algorithm for the case where each a_i is an $m \times m$ matrix.

(c) Consider the vector difference equation

$$x(t+1) = A(t)x(t) + u(t).$$

Here, for $t = 0, 1, \ldots, n$, $x(t)$ and $u(t)$ are vectors in $\Re^m$, and $A(t)$ is an $m \times m$ matrix. Assume that $A(t)$, $u(t)$ are known for each t, and that $x(0)$ is also given. Use the result of part (b) to obtain an $O(\log n \cdot \log m)$ time parallel algorithm for computing $x(n)$.

2.4. Show how to compute the kth power of an $n \times n$ matrix in time $\Theta\big(\log k \cdot \log n\big)$ using n^3 processors, when k is not a power of 2.

2.5. **(a)** Consider the scalar difference equation

$$x(t+1) = a(t)x(t) + b(t)x(t-1)$$

and consider the problem of computing $x(n)$. The inputs of the computation are $x(-1)$, $x(0)$, $a(0), \ldots, a(n-1)$, $b(0), \ldots, b(n-1)$. Assuming that enough processors are available, find a parallel algorithm that takes time $\Theta(\log n)$. *Hint:* Write the

difference equation in vector form and reduce the problem to the multiplication of n matrices of dimensions 2×2.

(b) Repeat part (a) with the only difference that $x(-1)$ is unknown but $x(n-1)$ is provided instead as an input to the computation, under the additional assumption that there exists a unique sequence $x(0), x(1), \ldots, x(n-1), x(n)$ satisfying the given initial and final conditions. *Hint:* Obtain a system of linear equations of the form

$$\begin{bmatrix} x(n) \\ x(n-1) \end{bmatrix} = D \begin{bmatrix} x(1) \\ x(0) \end{bmatrix},$$

where D is a 2×2 matrix that can be efficiently computed. Solve for $x(n)$ and $x(1)$.

2.6. Show that a polynomial $p(x) = a_n x^n + \cdots + a_1 x + a_0$ can be evaluated in parallel in time $\Theta(\log n)$. Here the inputs of the computation are the coefficients of the polynomial and the value of x.

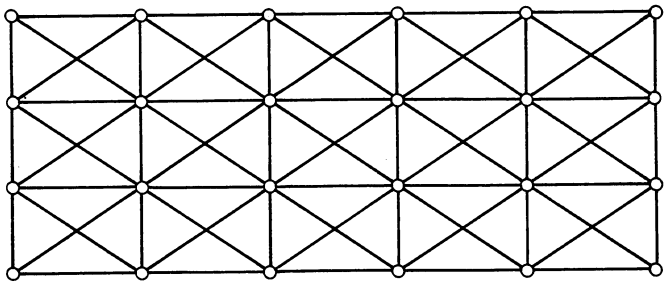


Figure 1.2.12 Dependency graph for nine-point discretizations (Exercise 2.7).

2.7. Consider the dependency graph G of Fig. 1.2.12, where all arcs are interpreted as being bidirectional. Such a dependency graph is obtained from so called "nine–point" discretizations of partial differential equations with two spatial variables [Ame77].

(a) Show that G cannot be colored with less than four colors.

(b) Find a coloring of G with four colors.

(c) Assuming that a mesh of processors is available (that is, a two–dimensional array of processors with nearest neighbor connections), show that we can assign four differently colored nodes to each processor in a way that the execution of the Gauss–Seidel algorithm requires only nearest neighbor communication.

1.3 COMMUNICATION ASPECTS OF PARALLEL AND DISTRIBUTED SYSTEMS

In many parallel and distributed algorithms and systems the time spent for interprocessor communication is a sizable fraction of the total time needed to solve a problem. In this case we say that the algorithm experiences substantial communication penalty or communication delays. We can think of the communication penalty as the ratio

$$CP = \frac{T_{TOTAL}}{T_{COMP}}, \tag{3.1}$$

where T_{TOTAL} is the time required by the algorithm to solve the given problem, and T_{COMP} is the corresponding time that can be attributed just to computation, that is, the time that would be required if all communication were instantaneous. This section is devoted to a discussion of a number of factors affecting the communication penalty.

To analyze communication issues, it is helpful to view the distributed computing system as a network of processors connected by communication links. Each processor uses its own local memory for storing some problem data and intermediate algorithmic results, and exchanges information with other processors in groups of bits called *packets* using the communication links of the network. The length of packets can be widely varying, ranging from a few tens of bits, to several thousands of bits. We assume that when a packet travels on a communication link, the bits of the packet are consecutively transmitted without interruption. A shared memory can also be viewed as a communication network, since each processor can send information to every other processor by storing it in the shared memory. This analogy can be extended to the case where the shared memory is organized in a hierarchy of memory sections, each, possibly, having a different access time for different processors. We will emphasize, however, a communication model based on direct processor–to–processor links, since such a model is somewhat easier to understand and analyze. We will also adopt a *store–and–forward* packet switching data communication model, whereby a packet that must travel over a route involving several processors, may have to wait at any one of these processors for some communication resource to become available before it gets transmitted to the next node. In some systems, it is possible that packets are divided and recombined at intermediate nodes on their routes, but we will not consider this possibility in our discussion. In another approach, called *circuit switching* (see [BeG87], [Sch87]), the communication resources needed for a packet's transfer are reserved via some mechanism before the packet's transfer begins. As a result the packet does not have to wait at any point along its route. Circuit switching is almost universally employed in telephone networks, but is seldom used in data networks or in parallel and distributed computing systems. It will not be considered further in this book.

Communication delays can be divided into four parts:

(a) *Communication processing time.* This is the time required to prepare information for transmission; for example, assembling information in packets, appending addressing and control information to the packets, selecting a link on which to transmit each packet, moving the packets to the appropriate buffers, etc.

(b) *Queueing time.* Once information is assembled into packets for transmission on some communication link, it must wait in a queue prior to the start of its transmission for a number of reasons. For example, the link may be temporarily unavailable because other information packets or system control packets are using it or

are scheduled to use it ahead of the given packet; or because a contention resolution process is underway whereby the allocation of the link to several contending packets is being decided. Another reason is that it may be necessary to postpone the transmission of a packet to ensure the availability of needed resources (such as buffer space at its destination). In the case where the possibility of transmission errors is nonnegligible, the queueing time includes the time for the packet retransmissions needed to correct the errors. Queueing time is generally difficult to quantify precisely, but simplified models are often adequate to obtain valuable qualitative and quantitative conclusions.

(c) *Transmission time.* This is the time required for transmission of all the bits of the packet.

(d) *Propagation time.* This is the time between the end of transmission of the last bit of the packet at the transmitting processor, and the reception of the last bit of the packet at the receiving processor.

Depending on the given system and algorithm, one or more of the above times may be negligible. For example, in some cases the information is generated with sufficient regularity and the transmission resources are sufficiently plentiful so that there never is a need for queueing packets, whereas in other cases the physical distance between transmitter and receiver is so small that propagation delay is negligible.

For most systems, we can reasonably assume that the processing and propagation time on a given link is constant for all packets, and the transmission time is proportional to the number of bits (or length) of the packet. We thus arrive at the following formula for the delay of a packet in crossing a link:

$$D = P + RL + Q, \tag{3.2}$$

where P is the processing and propagation time, R is the transmission time required for a single bit, L is the length of the packet in bits, and Q is the queueing time. (We are using bits as the packet length unit, but any other unit that requires a fixed transmission time can be used.)

It is difficult to make general statements regarding the size of the various terms in the delay formula (3.2). In some systems, the transmission time RL is much larger than the processing and propagation time P, particularly when L includes a substantial number of overhead bits. In other cases, the reverse is true. In the great majority of presently existing systems, even when the packet does not contain much more than overhead, the sum $P + RL$ is much larger than the time required to execute an elementary numerical operation such as a floating point multiplication. This means that if a parallel algorithm requires transmission of a packet for every few numerical operations it performs, the communication time is likely to dominate its execution time.

Throughout this section we focus on packets as units of communication. It should be noted, however, that a packet is sometimes part of some "message" that makes sense only when received as a whole. A message may be segmented into several packets for transmission for a number of reasons, and it is then appropriate to focus on the delay of

the entire message (from start of transmission of its first packet to the end of reception of its last packet). This complicates the situation because the message delay depends on how the message is segmented into packets, and on whether the transmission of the packets can be parallelized. For example, if a message is divided into n equal length packets that are transmitted over n equal delay, independent parallel communication paths, the message delay will be smaller by a factor n over the case where the entire message is transmitted over one of the paths. (This accounting assumes that the extra communication overhead when the message is segmented into several packets is negligible, and that the processing, propagation, and queueing delays are also negligible.)

Another possibility for parallelizing communication arises when a message is to be transmitted over a path of $k > 1$ links. If the message is segmented into n packets that are transmitted sequentially over the k–link path with a transmission time on each link equal to T per packet, the delay of the message will be $(n+k-1)T$ as compared with the delay of knT that will be required if the entire message is transmitted as a single packet. (This accounting neglects the effect of overhead, processing and propagation delays, and queueing, and assumes that a node must receive a packet in its entirety before relaying any portion of the packet to some other node.) The delay reduction is achieved by pipelining the transmission of the packets over the k links as shown in Fig. 1.3.1. It is seen that by making the packet size very small, the delay can be reduced by a factor nearly equal to the number of links of the path, to almost the time required to transmit the message over a single link. This motivates a special type of transmission method, sometimes called *cut–through transmission*, where a node can relay to another node any portion of a packet without waiting to receive the packet in its entirety. This amounts to segmenting the packet into many smaller packets to take advantage of the type of pipelining illustrated in Fig. 1.3.1. Naturally, this type of transmission method should be organized so that packets can be correctly put back together at their ultimate destination. We will not go into this subject further. We note also that the idea of pipelining of communication is applicable in other situations such as for example transmitting over a spanning tree (see Exercise 3.19).

Some of the most important factors that influence communication delays are the following:

(a) The algorithms used to control the communication network, mainly error control, routing, and flow control.

(b) The communication network topology, that is, the number, nature, and location of the communication links.

(c) The structure of the problem solved and the design of the algorithm to match this structure, including the degree of synchronization required by the algorithm.

The above factors are discussed in the subsequent subsections.

1.3.1 Communication Links

The precise method by which information is physically transmitted over a communication channel will not be important for us. It suffices for our purposes to view a communication

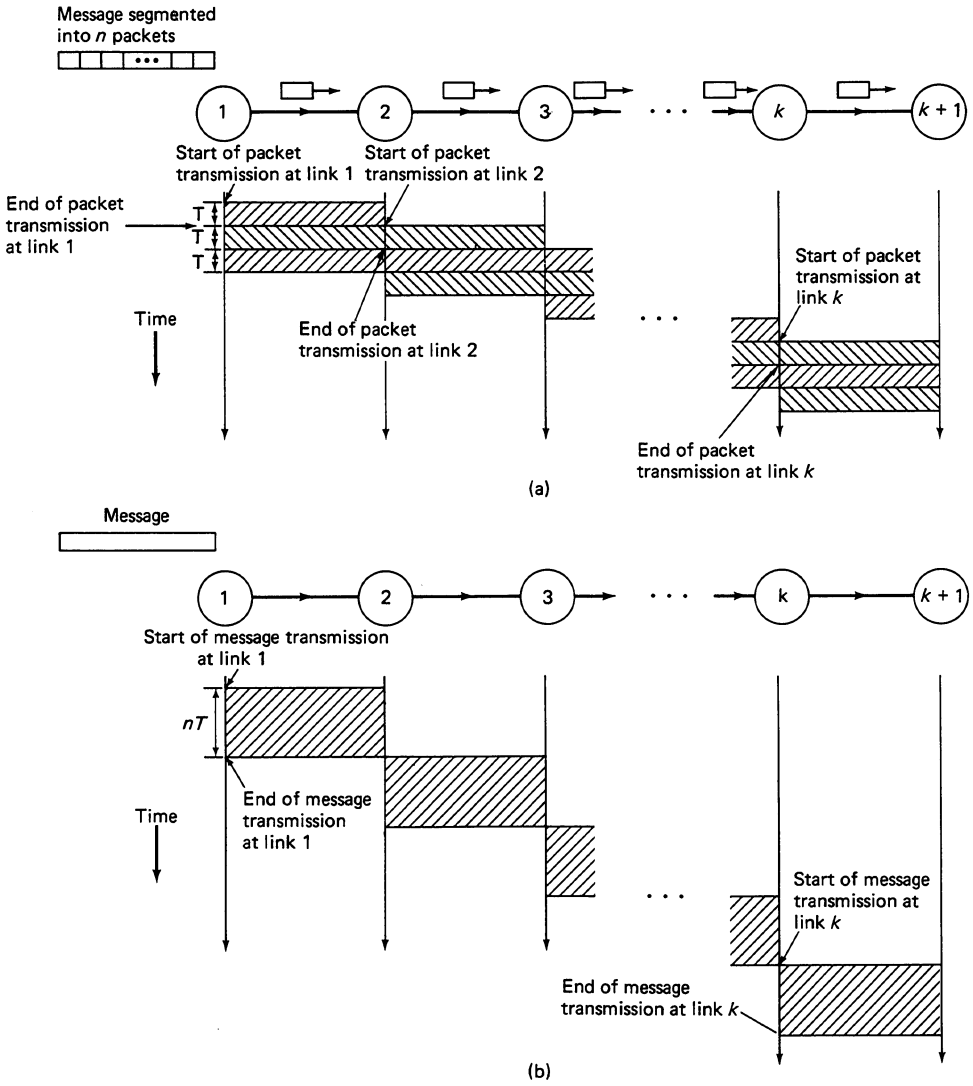


Figure 1.3.1 Segmentation of a message into packets to take advantage of pipelining over a path consisting of k communication links. In (a) the message is divided into n packets, each requiring T time units for transmission over a single link. The total time required is $(n + k - 1)T$. In (b) the message is transmitted as a whole on each link requiring nT time units on each link for a total of knT time units. (This accounting assumes that the overhead per packet and the processing, propagation, and queueing delays on all links are negligible.)

link as a *virtual bit pipe* along which bits travel, starting from one point referred to as the *transmitter* or the *origin*, and arriving at another point referred to as the *receiver* or the *destination*. There are several types of such pipes. One is the *synchronous* bit pipe, where the transmitter continuously sends bits at a fixed rate; packet bits if a packet is

available for sending, and dummy bits otherwise. An example is a high speed point–to–point wire connection; this is the type of bit pipe that is the most common in parallel and distributed computing systems. A second type is the *intermittent synchronous* bit pipe, where the transmitter sends bits at a fixed rate when it has a packet to send, and sends nothing otherwise. An example is Ethernet, which is a local area network bus (see e.g. [BeG87]). Finally, a third type is the *asynchronous character* pipe, where packets are transmitted in groups of bits called *characters* (usually eight or nine bits some of which are used for synchronization); the bits within each character are sent at a fixed rate, but successive characters can be separated by variable delays, subject to a given minimum. A typical example involves low speed communication using personal computers, and/or dial up telephone lines. In all of these pipes, the physical representation of bits can take many different forms, using a variety of modulation and coding techniques that may include forward error correction; see [BeG87] and [Sta85].

The *capacity* of a communication link is the maximum rate at which bits can be transmitted over the corresponding bit pipe, and is equal to $1/R$, where R is the bit transmision time used in the delay equation (3.2). In the context of a given parallel or distributed algorithm, the transmission rate of bits that carry algorithm–related information is actually smaller due to a number of reasons:

(a) Each packet may carry overhead bits used for detecting the packet's start and end, and, possibly, for detecting errors in transmission (see the discussion of data link control in the next subsection); the effect of the overhead bits can usually be accounted for in the terms P or L of the delay formula (3.2).

(b) The link may be used in part to transmit packets that are unrelated to the distributed algorithm under consideration. For example, it may be necessary to transmit periodically some control packets needed to sustain the organizational structures of the underlying computing system.

(c) In some links the communication hardware may be time–shared between several virtual bit pipes, so each of these pipes can be used only part of the time. A typical example is when a processor can, at any one time, send a packet along at most one of several incident physical communication channels. Another example arises in multiaccess communication links, such as Ethernet and other local area networks, where a physical channel is shared among several virtual bit pipes on a contention basis. (Multiaccess channels and the algorithms that are used to control them will not be discussed in this book; see, e.g., [BeG87] and [Sta87].) When the physical communication hardware is shared between several communication links, the queueing time Q in the delay equation (3.2) is nonnegligible and must be taken into account. This complicates seriously the analysis of the communication penalty.

(d) Some scarce resource needed by the packets (such as, for example, buffer space at a subsequent destination) may be unavailable. Packets are then forced to wait for the resource to become available even though the communication link is available for transmission. Thus the rate of transmission of the packets is reduced to the rate at which the scarce resource becomes available. Algorithms that effect this type

of rate reduction are called *flow control* algorithms. Flow control is discussed in many sources, e.g. [BeG87]. It will not be considered further in this book.

(e) In some communication links, there is a nonnegligible possibility that some bits are received differently than they were transmitted (e.g., a transmitted 0 is received as a 1), or are lost altogether (that is, their presence is not even detected). In such cases it is necessary to use a scheme that detects these malfunctions and retransmits the packets involved as many times as is necessary for ultimate correct reception. This lowers the rate at which information can be transmitted over the link, and simultaneously affects the queueing time Q in the delay formula (3.2), since a packet that is retransmitted may conceptually be viewed as waiting in queue up to the start of its first correct transmission.

Regardless of the nature of the corresponding virtual bit pipe, we would like to view a communication link as an *asynchronous packet pipe* that packets enter and exit after a delay D given by Eq. (3.2). The times of entry and exit, and the delay D need not exhibit any kind of deterministic regularity, but we would like to be able to assume that packets exit the pipe in the same order that they enter it, and that they are not altered in any way inside the pipe. This can be accomplished with the use of data link control algorithms that are discussed next.

1.3.2 Data Link Control

The two principal components of data link control (DLC) algorithms for virtual bit pipes are the mechanism of recognizing the start and end of packets (this is called *framing*), and the mechanism for *error detection and retransmission* of packets received in error. We assume throughout that packets arrive in the same order that they are sent, but that any one of them can get damaged in transmission or can get lost in the sense that its transmission may not be detected at the receiver.

We discuss framing briefly, and we refer to [BeG87] and [Sta85] for further description and analysis. A prerequisite for any framing technique, especially important for intermittent synchronous pipes and asynchronous character pipes, is a mechanism for recognizing the start of any bit stream following a period of idleness. This is a subject that relates to the design of the communication hardware, and will not be discussed further; see [Sta85]. One major framing technique is based on special bit sequences, called *flags*, that appear immediately before and after each packet. The receiver scans the incoming bit stream, and looks for a nonflag bit sequence following (or preceding) a flag to indicate the start (or end, respectively) of the packet. A technique known as *bit* or *character stuffing* is used to guard against the possibility that the flag sequence appears inside a packet; see [BeG87] and [Sta85]. A second major technique for framing is to encode, at the start of each packet, the number of bits of the packet (or to use a fixed packet length known to the receiver), allowing the receiver to distinguish the end of each packet. In this scheme a special resynchronization mechanism must be provided for the system to recover from a situation where a packet length is incorrectly received.

Error detection techniques are based on appending to the packets a sequence of extra bits, that are used to check for errors in transmission. An example of a primitive

form of error detection is to form at the transmitter the modulo 2 sum of the bits of each packet, and add it to the end of the packet in the form of an extra bit (called the *parity check*). The receiver also forms the modulo 2 sum of the bits of the received packet, and compares it with the parity check. Assuming the parity check is received correctly, this method will detect all damaged packets with an odd number of bits in error, but will miss all damaged packets with an even number of bits in error. More sophisticated and reliable error detection techniques are based on adding, at the end of each packet, a bit sequence [called *cyclic redundancy check* (CRC) sequence] that is the remainder of the modulo 2 division of the polynomial having as coefficients the bits of the packet divided by a fixed polynomial having binary coefficients and called the *generator polynomial*. An example of a polynomial that is standard in data networks is $x^{16} + x^{15} + x^2 + 1$, in which case the CRC is 16 bits long. More generally, the length of the CRC is equal to the degree of the generator polynomial. The error detecting capability of the scheme typically increases with the length of the CRC sequence (see [Gal68] and [BeG87]). Note, however, that if there is a positive probability that any one bit can be received with error, there is no scheme that can guarantee foolproof protection against undetected errors. One must deal in practice with bit pipes where undetected errors are possible but very rare. Our subsequent discussion assumes that the error detection scheme used is infallible.

The typical method for correcting transmission errors is based on detecting which packets have been transmitted in error and retransmiting them as many times as is necessary for correct reception. In the simplest type of retransmission protocol, called *stop–and–wait*, the transmitter A sends a packet and the receiver B replies with a packet that contains either a positive acknowledgment (ACK) for a correct reception, or a negative acknowledgment (NAK) for an incorrect reception. If A receives a NAK, it retransmits the packet, and if it receives an ACK it transmits the next packet. It is interesting to note that the algorithm is distributed, since it involves two processors that do not share simultaneously the same information.

Despite its simplicity, the preceding algorithm involves considerable subtleties which illustrate some of the issues one has to deal with in implementing and justifying a distributed algorithm. The difficulty is that packets from A to B and from B to A can be delayed unpredictably in the communication channel, and they can also be lost. To guard against the possibility of a loss, it is necessary for A to take a timeout following the transmission of a packet, and retransmit the packet if it does not receive an ACK within a given period of time. On the other hand, it may be impossible to choose a timeout interval that is sufficiently small to ensure timely retransmission of lost packets, and is also sufficiently long to preclude retransmission of some packets that are merely delayed. Fig. 1.3.2 shows how this can lead to confusion at either A or B if the packets do not carry enough information to allow the unambiguous association of A to B packets with their corresponding acknowledgments. The remedy is to number sequentially the A to B packets, and to include on each acknowledgment packet the number of the A to B packet that is being acknowledged.

A final difficulty has to do with the fact that packet numbers can become arbitrarily large if transmission continues indefinitely, and can overflow any field with a finite

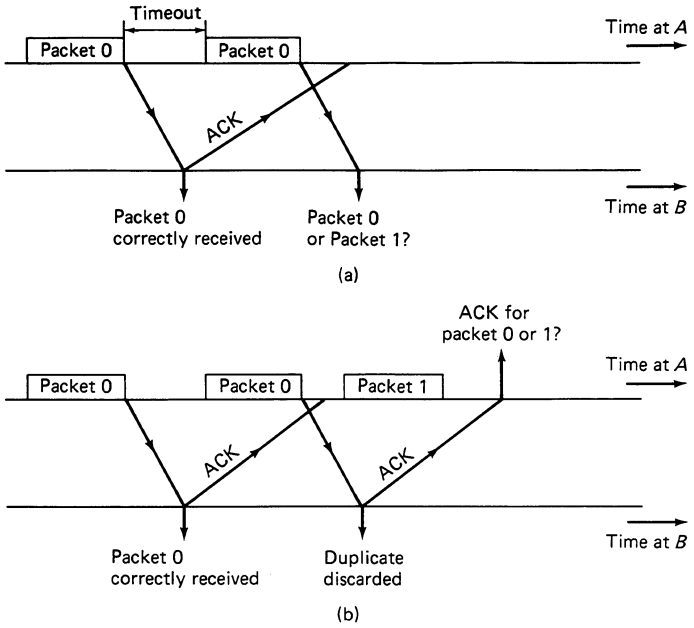


Figure 1.3.2 Illustration of the need to number both the A to B packets and the B to A packets in the stop–and–wait protocol. (a) Example of the confusion that can arise when packets from A to B are not numbered. The ACK for packet 0 is late to arrive, so A retransmits packet 0 after a timeout, but B cannot tell whether the second packet received is a duplicate of packet 0 or whether it is packet 1. (b) Example of the confusion that can arise when the B to A packets do not identify the packet that is being acknowledged. Here A cannot tell whether the second ACK is for packet 0 or for packet 1.

number of bits that will be provided for them. It turns out that it is sufficient to number packets modulo 2. An intuitive reason is that, at any given time, if there is a copy of packet i in the system (meaning that its transmission has started but its acknowledgment has not been received), there cannot simultaneously be a copy of packet $i + 2$ in the system, so that the former packet cannot be confused for the latter even though they carry the same modulo 2 sequence number. (All the copies of packet i are transmitted before all the copies of packet $i + 1$, so all the acknowledgments for copies of packet i are received before all the acknowledgments for copies of packet $i + 1$. Therefore, if a copy of packet i is in the system, then no acknowledgment for packet $i + 1$ must have been received, which implies that transmission of any copy of packet $i + 2$ cannot have started.)

The correctness of the preceding algorithm is intuitively rather obvious, but a rigorous proof requires a model of the combined state of A and B, as well as a prescription of how this state changes in response to all the possible packet receptions. The reader who tries to work out the details of this model (outlined in Exercise 3.1) may be surprised by the complexity and laboriousness of the argument needed to show rigorously the validity of a simple distributed algorithm such as the stop–and–wait protocol. This difficulty

is symptomatic of the complicated nature of distributed algorithms involving several processors that exchange information along communication links. A detailed model of such a distributed algorithm requires that each processor be viewed as a system with an internal state that accepts as inputs packets received from other processors, and produces as outputs packets sent to other processors. The state changes in response to the input, and the output depends on the state and the input (see Fig. 1.3.3). While a serial algorithm consists of a single such system, a distributed algorithm consists of an interconnection of many such systems, yielding a composite system that can be quite complex to model, analyze, and implement.

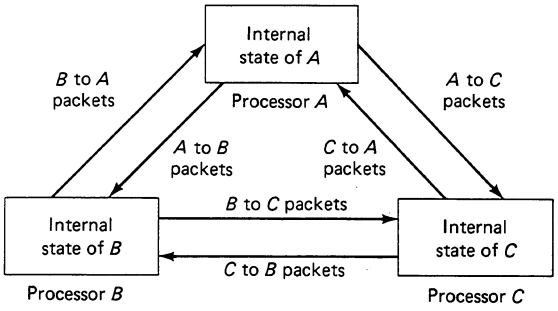


Figure 1.3.3 Representation of a distributed algorithm as an interconnection of subsystems, one per processor. Each subsystem/processor has an internal state that changes in response to packet receptions from other processors. A description of the distributed algorithm must include the rule by which the state of each processor changes, and the rule by which packets destined for other processors are generated.

The stop–and–wait protocol leads to long delays because a packet must wait in a queue until the reception of the acknowledgment of the previous packet. A more efficient scheme, which is widely used, is the *go–back–n algorithm.* The idea here is to allow sending as many as n packets following the last packet that has been acknowledged, while using timeouts to decide when to restransmit packets whose acknowledgment is late in coming. Thus, assuming that packet i is the highest numbered packet for which an acknowledgment packet has been received, the transmitter can send any of the packets $i+1, i+2, \ldots, i+n$. The receiver, however, acknowledges packets in order, that is, it does not acknowledge correct reception of packet $i+1$ before it acknowledges correct reception of packet i. In fact, an acknowledgment packet indicates the next packet expected by the receiver, and thus simultaneously acknowledges all packets already received correctly. This algorithm yields, for $n = 1$, the stop–and–wait protocol, but avoids the long delays of that protocol by using $n > 1$. It is sufficient to number packets modulo m for any $m > n$. A rigorous proof, together with a description of other implementation details is given in [BeG87]. (For a heuristic argument, note that, at any given time, the number of consecutively numbered unacknowledged packets at the transmitter is at most n. Therefore, the range of possible packet numbers that are next expected at the receiver contains no more than $n+1$ consecutive numbers. Hence, if $m > n$, packet number k and packet number $k+m$ cannot simultaneously be in this range, and there is no possibility of the receiver getting confused when packets are numbered modulo m.) The standard modulo number used for terrestrial communication links in data networks is $m = 8$.

We finally note that in some systems, where communication links are very reliable, an error detection and retransmission scheme is not used in order to save the associated overhead. On the other hand there are systems where, even with an infallible data link control scheme, some packets can get lost due to a failure of a node of the communication network. (Think of a packet that clears the DLC unit of an incoming link to a node, and waits to enter the DLC unit of an outgoing link of the node. If the node crashes while the packet is waiting, the packet will be lost.) In such cases, it may be necessary to provide an additional error detection and retransmission scheme at a network–wide level rather than at a link level; see [BeG87] and [Tan81]. Furthermore it is necessary to provide a mechanism, called a *topology broadcast algorithm*, for informing all nodes about node or link failures. Such algorithms are typically distributed, and are discussed in Section 8.5.

1.3.3 Routing

The routing algorithm in an interconnection network is the mechanism by which packets are guided to their destinations through the network. The main objective of the routing algorithm is to select paths of small total delay for each packet. If there were no queueing delays along any link, the path selection would be quite simple. From Eq. (3.2), the delay associated with every link (i, j) would be equal to $P_{ij} + R_{ij}L$, where P_{ij} and R_{ij} are the processing plus propagation time, and the transmission time per bit on link (i, j), respectively, and L is the number of bits of the given packet. The problem of minimum delay routing from an origin node to a destination node would then be reduced to the problem of finding a path connecting the two nodes with minimum sum of link delays; this problem, known as the *shortest path problem*, will be discussed in detail in Sections 4.1 and 6.4. Unfortunately this method for selecting routes is totally insensitive to a potentially large packet arrival rate at any one link. As a result, some links may receive an excessive amount of traffic, thereby invalidating the assumption of negligible queueing delay.

To alleviate systematic tendencies for data traffic to be concentrated on a few links, several forms of multiple–path and randomized routing have been suggested. The idea here is to use more than one path for every origin–destination node pair, and to select, more or less at random, one of these paths for each packet. In one such method a shortest path is determined for every origin–destination pair; however, a packet originating at node A and destined to node B is not routed on the A to B shortest path. Instead, an intermediate node D is randomly chosen, and the packet is first routed from A to D along the A to D shortest path, and then from D to the packet's destination B along the D to B shortest path. Choosing the intermediate node at random for each individual packet tends to avoid congestion on links that lie on the shortest paths of one or more origin–destination pairs with excessive traffic. The randomized routing method just described is easy to implement in many systems, and is supported by some interesting analysis for highly regular interconnection networks (see [VaB81], [Val82], [HaC87], and [MiC87]). It has the drawback that it delivers the packets from A to B out of order, since a different intermediate destination D may be chosen for two successive packets. Furthermore, by

randomizing its routes, this method does not use any available knowledge of the traffic pattern associated with a given algorithm.

The possibility of multiple–path and randomized routing arises also in connection with a different context relating to storage limitations. In practice, one must deal with the fact that when the storage space of a node is full, the node cannot receive any packet along its incident links. Several approaches to cope with this problem include flow control methods (see [BeG87]), and/or acknowledgment and retransmission schemes, whereby a packet that cannot be received due to unavailability of buffer space is retransmitted following a suitable timeout. A different approach modifies the routing algorithm so that there is always available buffer space to store a received packet at each node [Hil85]. To understand this approach, let us assume that all links of the communication network can be used simultaneously and in both directions, and that each packet carries a destination address and requires unit transmission time on every link. We also assume that packets are transmitted in slots of unit time duration, and that slots are synchronized so that their start and end is simultaneous at all links. In a typical routing scheme, based on the shortest path routing method, each node, upon reception of a packet that is destined for a different node, uses table lookup to determine the next link on which the packet should be transmitted; we refer to this link as the *assigned link of the packet.* It is possible that more than one packet with the same assigned link is received by a node during a slot. We refer to this situation as a *packet collision.* If there is a packet collision in a given slot at a given node, and we insist that all the packets involved in the collision are routed through their assigned link, then at most one packet involved in the collision can be transmitted by the node in the subsequent slot, and the remaining packets must be stored by the node in a queue. This storage requirement can be eliminated by modifying the routing scheme so that all the packets involved in a collision at a given node during a given slot are transmitted in the next slot; one of them is transmitted on its assigned link, and the others are transmitted on some other links chosen at random from the set of links that are not assigned to any packet received in the previous slot. It can be seen that with this modification, at any node with d incident links, there can be at most d packets received in any one slot, and each of these packets will be transmitted along some link (not necessarily their assigned one) in the next slot. Therefore, there will be no queueing, and the storage space needed at the node is minimized. The price paid for this reduced storage requirement is that successive packets of the same origin–destination pair may be received out of order, and some packets may travel on long routes to their destinations; indeed, in this scheme, one may need to take precautions to ensure that a packet cannot travel on a cycle indefinitely.

An appropriate formulation of the minimum delay routing problem must take into account the queueing delays at the links, but, unfortunately, these delays cannot be easily quantified in general. There are, however, simplified models that represent queueing delay at a link as a function of the packet arrival rate at the link. Distributed optimal routing algorithms based on such models are discussed in Sections 5.6 and 7.6. They are useful in data network situations where there are many users sharing the network, with each user having a data rate that is small relative to the combined data rate of all users. In such networks, the packet arrival rate at a link is a meaningful quantity that can be

measured as a time average over a suitable period of time; see [BeG87]. Unfortunately, in most computing systems, and for most algorithms, there is no meaningful notion of packet arrival rate at a link, in which case these simplified queueing models are not appropriate.

On the other hand, the networks of many parallel computing systems have some regular form, and the routing problem may be posed in connection with a given algorithm that generates packets in a regular and predictable pattern. It may then be possible to design a routing algorithm that is tailored to the system and the algorithm at hand. In the next subsection, we look at some possibilities along these lines.

1.3.4 Network Topologies

In many systems such as data networks, sensor networks, distributed databases, etc., geographical and other considerations usually lead to interconnection networks with irregular form. On the other hand, in systems whose principal function is numerical computation, the network typically exhibits some regularity, and is sometimes chosen with a particular application in mind. In this subsection we discuss some example networks, and we focus on their communication properties.

We will represent a communication network of processors as a graph $G = (N, A)$, also referred to, somewhat loosely, as a *topology*. The nodes of the graph correspond to the processors, and the presence of an (undirected) arc (i, j) indicates that there is a direct communication link that serves as an error free, asynchronous packet pipe between processor i and processor j in both directions. We assume that communication can take place simultaneously on all of the incident links of a node and in both directions. We also assume in each of the communication analyses of this subsection that, in the absence of queueing, the delays of all packets of equal length (i.e., with the same number of bits) are equal on all links. Unless the opposite is clearly implied by the context, we assume that the delay of each packet is one time unit on every link. A somewhat restrictive assumption that is implicit in our analysis of some communication algorithms is that these algorithms are simultaneously initiated at all processors. However, the qualitative conclusions of our discussion remain largely unchanged even if the preceding assumptions do not hold. We will often use the notation $O(\cdot)$, $\Omega(\cdot)$, and $\Theta(\cdot)$, introduced in Subsection 1.2.2 and in Appendix A.

Topologies are usually evaluated in terms of their suitability for some standard communication tasks. The following are some typical criteria:

(a) The *diameter* of the network, which is the maximum distance between any pair of nodes. Here the distance of a pair of nodes is the minimum number of links that have to be traversed to go from one node to the other. For a network of diameter r, the time for a packet to travel between two nodes is $O(r)$, assuming no queueing delays at the links.

(b) The *connectivity* of the network, which provides a measure of the number of "independent" paths connecting a pair of nodes. We can talk here about the node or the arc connectivity of the network, which is the minimum number of nodes (or

arcs, respectively) that must be deleted before the network becomes disconnected. In some networks, a high connectivity is desirable for reliability purposes, so that communication can be maintained in the face of several link and node failures. Another important point is that if the network has arc connectivity k, then communication between any two nodes can be parallelized by making use of at least k paths with no pair of these paths having an arc in common (this follows from the max flow – min cut theorem [PaS82], and will not be proved here). Thus, a long message can be sent from node A to node B by splitting it into several packets, and by sending these packets in parallel on the arc–disjoint paths connecting A and B. In the absence of queueing delays and with negligible overhead and propagation time per packet, this reduces the communication time between any pair of nodes by a factor at least equal to the arc connectivity of the network. (However, the packets may arrive at the destination node in unpredictable order, so they may have to be put back in order to reconstruct the message. In some systems this may be undesirable because of the overhead involved, and possibly other reasons.) Note that the arc and node connectivity is bounded from above by the minimum value of the *degree* of a node defined as the number of incident arcs to the node.

(c) The *flexibility* provided in running efficiently a broad variety of algorithms. As an example, assuming that we have developed an algorithm that runs on a given network of processors represented by a graph $G' = (N', A')$, we may want to run the same algorithm on another network of processors represented by a graph $G = (N, A)$. This will be possible if G' can be imbedded into G in the sense that each node of G' can be mapped to a node of G in a way that the arcs of G' are associated with arcs of G, that is, if there exists a function $\sigma : N' \mapsto N$ such that $\sigma(i) \neq \sigma(j)$ if $i \neq j$ and $\big(\sigma(i), \sigma(j)\big) \in A$ for all $(i, j) \in A'$. In this case we say that G' can be *mapped* into G. The mapping problem also arises in another interesting context. Given a processor network with a fixed interconnection topology, an important issue is to divide a given computational task among the processors so that the communication penalty is kept at a minimum. Assume that the main task is divided into computational subtasks, and that each subtask will be assigned to a separate processor. Assume also that certain pairs (i, j) of subtasks interact, meaning that the execution of subtask i or j occasionally requires knowledge of certain values computed during execution of subtask j or i, respectively. It is then desirable to allocate subtasks to processors so that interacting subtasks are assigned to processors with a direct communication link. This problem can be formulated as a problem of mapping the graph $G' = (N', A')$ representing the subtask interactions into the graph $G = (N, A)$ representing the processor network. As an example, in executing the relaxation iteration

$$x_i(t+1) = f_i\big(x_1(t), \ldots, x_n(t)\big), \qquad \forall\, i = 1, \ldots, n, \tag{3.3}$$

discussed in Subsection 1.2.4, we want to be able to map the dependency graph (cf. Subsection 1.2.4) into the processor network. We thus see the advantages of a topology that is flexible, in the sense that many other topologies can be mapped into it.

(d) The *communication delay* required for some standard tasks that are important in many algorithms such as inner product computation, matrix–vector multiplication, etc. We describe a few such tasks.

Single Node and Multinode Broadcast: In the first communication task, we want to send the same packet from a given processor to every other processor (we call this a *single node broadcast*). In a generalized version of this problem, we want to do a single node broadcast simultaneously from all nodes (we call this a *multinode broadcast*). A typical example where a multinode broadcast is needed arises in relaxation iterations of the form (3.3). If we assume that there is a separate processor assigned to each variable, and that each function f_i in the right–hand side of Eq. (3.3) depends on all variables, then, at the end of an iteration, there is a need for every processor to send the value of its variable to every other processor, which is a multinode broadcast. A special case of this example arises in matrix–vector multiplication, and will be discussed in Example 3.1, and in Subsection 1.3.6.

Clearly, to solve the single node broadcast problem, it is sufficient to transmit the given node's packet along a *spanning tree rooted at the given node*, that is, a spanning tree of the network together with a direction on each link of the tree such that there is a unique positive path from the given node (called the *root*) to every other node. With an optimal choice of such a spanning tree, a single node broadcast takes $O(r)$ time, where r is the diameter of the network, as shown in Fig. 1.3.4(a). Note that if a long packet is involved in a single node broadcast, it can be segmented into smaller packets that can be transmitted sequentially along the spanning tree, thereby resulting in a potentially significant reduction of the broadcast time; this is similar to the pipelining effect that we discussed in connection with transmitting a long packet over a sequence of links (cf. Fig. 1.3.1). Suppose, in particular, that the packet requires one time unit for transmission on any link, and that it is segmented into m packets each requiring $1/m$ time units for transmission on any link (i.e., there is no extra communication overhead due to the segmentation). Then it can be seen that the time for the single node broadcast with a worst choice of the root node and an optimal choice of the spanning tree is reduced from r to $(r + m - 1)/m$ time units. For a more precise estimate that takes overhead into account, see Exercise 3.19.

To solve the multinode broadcast problem, we need to specify one spanning tree per root node. The difficulty here is that some links may belong to several spanning trees; this complicates the timing analysis, because several packets can arrive simultaneously at a node, and require transmission on the same link with a queueing delay resulting. This issue will be discussed later in the context of specific interconnection networks.

Single Node and Multinode Accumulation: There are two important communication problems that are dual to the single and multinode broadcasts, in the sense that the spanning tree(s) used to solve one problem can also be used to solve the dual in the same amount of communication time. In the first problem, called *single node accumulation*, we want to send to a given node a packet from every other node; we assume, however, that packets can be "combined" for transmission on any communication link, with a "combined" transmission time equal to the

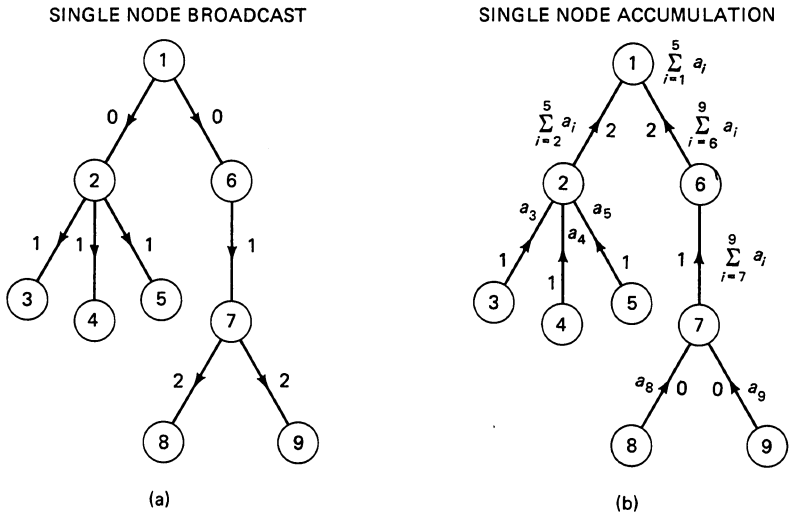


Figure 1.3.4 (a) A single node broadcast uses a tree that is rooted at a given node (which is node 1 in the figure). The time next to each link is the time at which transmission of the packet on the link begins. (b) A single node accumulation problem involving summation of n scalars $a_1, \ldots, a_n$ (one per processor) at the given node (which is node 1 in the figure). The time next to each link is the time at which transmission of the "combined" packet on the link begins, assuming that the time for scalar addition is negligible relative to the time required for packet transmission. The time for single node accumulation (or broadcast) is the maximum length of a path from a node to the root (or from the root to a node, respectively), counting each link as one unit. Thus, the single node accumulation and the single node broadcast take the same amount of time if a single packet in the latter problem corresponds to a scalar in the former problem.

transmission time of a single packet. This problem arises, for example, when we want to form at a given node a sum consisting of one term from each node as in an inner product calculation [see Fig. 1.3.4(b)]; we can view addition of scalars at a node as "combining" the corresponding packets into a single packet. The second problem, which is dual to a multinode broadcast, is called *multinode accumulation*, and involves a separate single node accumulation at each node. For example, it will be seen in Subsection 1.3.6 that a certain method for carrying out parallel matrix–vector multiplication involves a multinode accumulation.

It can be shown that a single node (or multinode) accumulation problem can be solved in the same time as a single node (respectively, multinode) broadcast problem. In particular, any single node (or multinode) accumulation algorithm can be viewed as a single node (or multinode, respectively) broadcast algorithm running in reverse time; the converse is also true. The process is illustrated in Fig. 1.3.4; the detailed mathematical proof is left for the reader.

Single Node Scatter, Single Node Gather, and Total Exchange: Another interesting communication problem is to send a packet from every node to every other node (here a node sends different packets to different nodes in contrast with

the multinode broadcast problem where each node sends the same packet to every other node). We call this the *total exchange problem*, and we will see later that it arises frequently in connection with matrix computations. A related problem, called the *single node scatter* problem, involves sending a separate packet from a single node to every other node. A dual problem, called *single node gather* problem, involves collecting a packet at a given node from every other node. An algorithm that solves the single node scatter (or gather) problem consists of a schedule of packet transmissions on each link that properly takes queueing into account. By reversing this schedule as discussed in connection with the single node accumulation problem, it can be seen that for every algorithm that solves the single node scatter (or gather) problem, there is a corresponding algorithm that solves the single node gather (or scatter, respectively) problem, and takes the same amount of communication time.

Note that in a multinode broadcast, each node receives a different packet from every node, thereby solving the single node gather problem. Note also that the total exchange problem may be viewed as a multinode version of both a single node scatter and a single node gather problem, and also as a generalization of a multinode broadcast, whereby the packets sent by each node to different nodes are different. We conclude that the communication problems of the preceding discussion form a hierarchy in terms of difficulty, as illustrated in Fig. 1.3.5. An algorithm solving one problem in the hierarchy can also solve the next problem in the hierarchy in no additional time. In particular, a total exchange algorithm can also solve the multinode broadcast (accumulation) problem; a multinode broadcast (accumulation) algorithm can also solve the single node gather (scatter) problem; and a single node scatter (gather) algorithm can also solve the single node broadcast (accumulation) problem. Therefore, the communication requirements for these problems decrease in the order just given, regardless of the network being used.

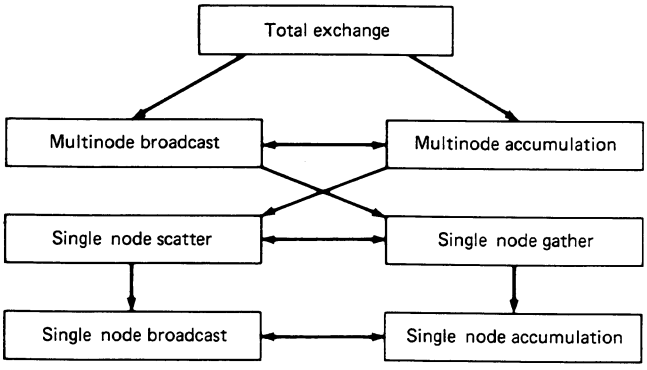


Figure 1.3.5 Hierarchy of basic communication problems in interconnection networks. A directed arc from problem A to problem B indicates that an algorithm that solves A can also solve B, and that the optimal time for solving A is not more than the optimal time for solving B. A horizontal bidirectional arc indicates a duality relation.

We now consider a number of specific topologies.

Complete Graph

Here there is a direct link between every pair of processors. Such a network can be implemented by means of a bus which is shared by all processors, or by means of some type of crossbar switch. Clearly this is an ideal network in terms of flexibility. Unfortunately, when the number of processors is very large, a crossbar switch becomes very costly, and a bus involves large queueing delays. Complete graphs, however, are frequently used to connect small numbers of processors in clusters in a hierarchical network, where the clusters are themselves connected via some other type of communication network.

Linear Processor Array

Here there are p processors/nodes numbered $1, 2, \ldots, p$, and there is a link $(i, i+1)$ for every pair of successive processors [see Fig. 1.3.6(a)]. The diameter and connectivity properties of this network are the worst possible. Furthermore, one can map a linear array into most other networks of interest (all networks discussed in this section with the exception of trees). This means that the communication penalty for a given algorithm using a linear array can be no better than the corresponding penalty using most other networks. The time taken by an optimal single node broadcast algorithm depends on the origin node; at worst, it is $p-1$ time units (assuming each packet transmission requires unit time), since the diameter of the linear array is $p-1$. An optimal multinode broadcast algorithm takes the same amount of time thanks to the possibility of using all communication links in parallel [see Fig. 1.3.6(b)]. The time taken by an optimal single node scatter algorithm lies between the times taken by an optimal single node and an optimal multinode broadcast algorithm, and is therefore at worst $p-1$ time units. In fact it can be shown that the time taken by an optimal single node broadcast algorithm as well as by an optimal single node scatter algorithm starting at node k is $\max\{k-1, p-k\}$ time units (see Exercise 3.9). Finally, an optimal total exchange algorithm takes $\Theta(p^2)$ time. To see this, consider the link $(k, k+1)$ that separates the array in two node subsets with k and $p-k$ nodes respectively. Since in a total exchange each node of one subset must send a packet to each node of the other subset, the link must carry $k(p-k)$ packet transmissions in each direction. Allowing for the worst possible link selection, we see that any total exchange algorithm takes at least $\max_k [k(p-k)]$ or (after some calculation) $\lceil (p^2-1)/4 \rceil$ time units. On the other hand, one way to solve the total exchange problem (not necessarily the fastest) is to solve sequentially p single node scatter problems, one for each of the p processors. Every one of these problems can be solved in no more than $p-1$ time units, thereby showing that an optimal total exchange algorithm takes $\Theta(p^2)$ time.

Example 3.1. *Matrix–Vector Multiplication*

Consider the problem of parallel multiplication of an $n \times n$ fully dense matrix with an n–dimensional vector, with subsequent communication of the result to all of the p processors of a linear array. We assume here that $n = pk$, where $k \geq 1$ is some integer, and that processor i knows the vector and the rows $(i-1)k+1$ to ik of the matrix, and calculates coordinates $(i-1)k+1$ to ik of the matrix–vector product. At the end of the calculation each processor

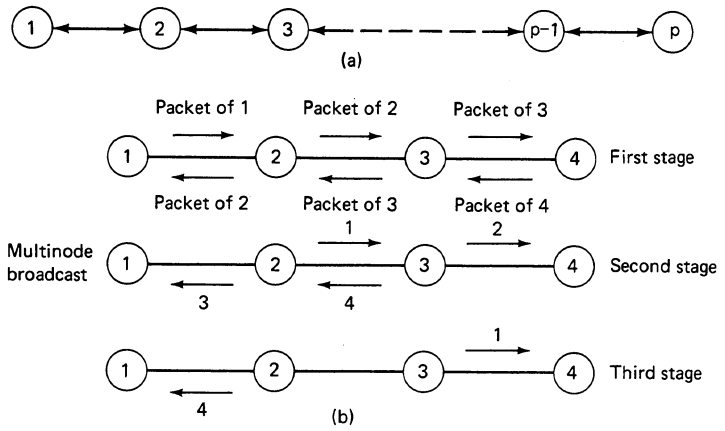


Figure 1.3.6 (a) Linear array with p processors. There is a bidirectional communication link $(i, i+1)$ for each $i = 1, 2, \ldots, p-1$. (b) A multinode broadcast in a linear array can be conducted in stages. At the first stage, each node sends to its neighbor(s) its own packet. Each node $i \in \{2, 3, \ldots, p-1\}$ that receives a packet from $i+1$ (or $i-1$) at some stage, relays this packet to $i-1$ (or $i+1$, respectively) at the next stage. Nodes 1 and p send a packet only at the first stage. Thus, at stage k, node i receives the packet of node $i-k$ (if $i > k$), and the packet of node $i+k$ (if $i \leq p-k$). The multinode broadcast is completed after $p-1$ stages. The figure illustrates this process for $p = 4$.

i sends these k coordinates to all other processors in a single packet. This is a multinode broadcast. Normalizing the length of a packet, we assume that each packet contains k data units and w units of overhead. We assume that the delay of each packet on each link is $\alpha(k+w)$, where α is some constant. (We neglect the processing and propagation delay; for the purposes of the subsequent calculation, it can be lumped into the overhead w.) The communication time using an optimal multinode broadcast algorithm is then $\Theta\big(p(k+w)\big)$, whereas the corresponding computation time is $\Theta(pk^2)$. We see, therefore, that for any given number of processors p, if $n = pk$ is sufficiently large, the time spent for communication is negligible relative to the time for computation. This phenomenon holds for many problems of interest (see Subsection 1.3.5), and is significant since it indicates that the communication penalty does not prevent the effective use of an increased number of processors in matrix–type problems as their dimension becomes larger. The appropriate number of processors used, however, should be chosen judiciously; it is not always advantageous to use as many processors as available. To see this, note that the computation time is $\Theta(nk)$ and the communication time is $\Theta\big(n(1+w/k)\big)$. For many practical systems, the size of w is such that when k is small, the communication time becomes dominant.

Ring

This is a simple and common network that has the property that there is a path between any pair of processors even after any one communication link has failed. However, the number of links separating a pair of processors can be as large as $\lceil (p-1)/2 \rceil$, where p is the number of processors. It can be seen that all of the basic communication problems discussed earlier (single node and multinode broadcast, single node scatter, and total

exchange) can be solved on a ring in a time that lies between the corresponding time on a linear array with the same number of nodes, and one–half that time (see Fig. 1.3.7 for the case of a multinode broadcast).

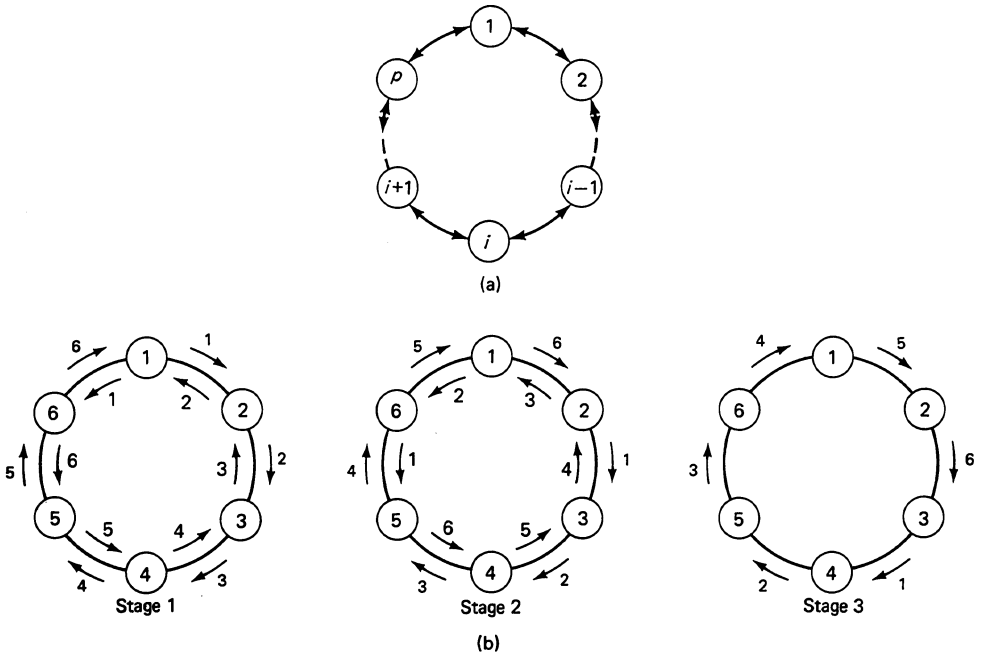


Figure 1.3.7 (a) A ring of p nodes having as links the pairs $(i, i+1)$ for $i = 1, 2, \ldots, p-1$, and $(p, 1)$. (b) A multinode broadcast on a ring with p nodes can be performed in $\lceil (p-1)/2 \rceil$ stages as follows: at stage 1, each node sends its own packet to its clockwise and counterclockwise neighbors. At stages $2, \ldots, \lceil (p-1)/2 \rceil$, each node sends to its clockwise neighbor the packet received from its counterclockwise neighbor at the previous stage; also, at stages $2, \ldots, \lceil (p-2)/2 \rceil$, each node sends to its counterclockwise neighbor the packet received from its clockwise neighbor at the previous stage. The figure illustrates this process for $p = 6$.

Tree

A tree network with p processors provides communication between every pair of processors with a minimal number of links $(p-1)$. One disadvantage of a tree is its low connectivity; the failure of any one of its links creates two subsets of processors that cannot communicate with each other. Furthermore, depending on the particular type of tree used, its diameter can be as large as $p-1$ (note that the linear array is a special case of a tree). The star network has minimal diameter among tree topologies; however the central node of the star handles all the network traffic, and can become a bottleneck [see Fig. 1.3.8(a)]. It can be shown that the optimal time for a single node broadcast (or accumulation) and a single node gather (or scatter) on a tree with p processors is no more than $p-1$ time units, whereas the optimal time for a multinode broadcast is

$p-1$ time units (Exercises 3.2 and 3.9). The optimal time for a total exchange depends on the type of tree considered, and it is $O(p^2)$ based on the result for the single node gather problem stated earlier. An interesting type of tree is the *binary and balanced tree* described in Fig. 1.3.8(b). It can be seen that an optimal total exchange algorithm on a binary balanced tree takes $\Theta(p^2)$ time (Exercise 3.2).

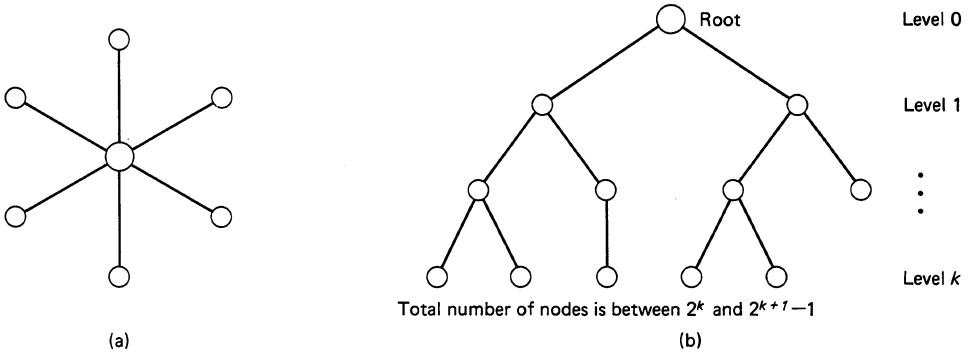


Figure 1.3.8 (a) A star network. (b) A binary balanced tree. Here there is a special node called the *root*. Each node i is connected to the root via a unique simple walk. The first node on this walk is called the *parent* of i. If j is the parent of i, then i is called a *child* of i. In a binary tree there can be at most two children for each node. A node with no children is said to be a *leaf* of the tree. A binary tree with p nodes is said to be balanced if the walk from each leaf node to the root contains either $\lceil \log(p+1) \rceil - 2$ or $\lceil \log(p+1) \rceil - 1$ links.

Mesh

Many large problems of interest are closely tied to the geometry of physical space. Mesh–connected processor arrays are often well suited for such problems. In a d–dimensional mesh the processors are arranged along the points of d–dimensional space that have integer coordinates, and there is a direct communication link between nearest neighbors. Using the graph formalism, the nodes of a d–dimensional mesh with n_i points along the ith dimension are the d–tuples $(x_1, \ldots, x_d)$ where each of the coordinates x_i, $i = 1, \ldots, d$, takes an integer value from 1 to n_i. The links are the pairs $\big((x_1, \ldots, x_d), (x'_1, \ldots, x'_d)\big)$ for which there exists some i such that $|x_i - x'_i| = 1$ and $x_j = x'_j$ for all $j \neq i$. The diameter of a mesh–connected network is $\sum_{i=1}^{d}(n_i - 1)$, which can be much smaller than the diameter of a ring and much larger than the diameter of a binary balanced tree with the same number of processors. A variation with smaller diameter is the *mesh network with wraparound* shown in Fig. 1.3.9.

Consider now the time needed to solve various communication problems on a d–dimensional mesh with p processors, which is *symmetric* in the sense that it has an equal number ($p^{1/d}$) of processors along each dimension. We assume that d is fixed and we estimate the communication time as a function of p. An optimal single node broadcast (or accumulation) algorithm takes $\Theta(p^{1/d})$ time, since the diameter is $d(p^{1/d} - 1)$. It is easily seen that a linear array with p nodes can be mapped into the symmetric mesh,

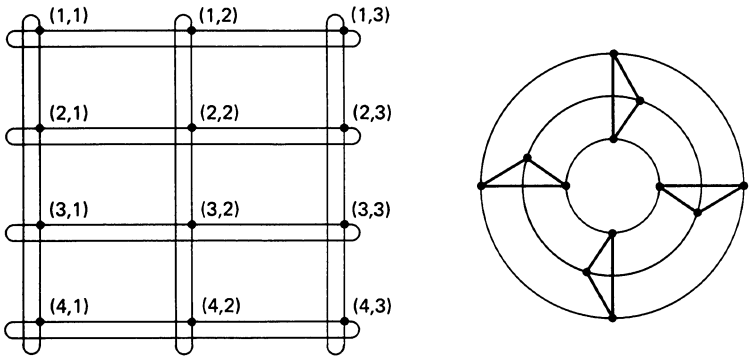


Figure 1.3.9 Meshes with wraparound. Here, in addition to the links of the ordinary mesh, we have the links $\big((x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_d), (x_1, \ldots, x_{i-1}, n_i, x_{i+1}, \ldots, x_d)\big)$. The diameter is roughly half the diameter of the corresponding ordinary mesh.

and, therefore, the time taken by an optimal single node scatter (or gather) algorithm and by an optimal multinode broadcast (or accumulation) algorithm do not exceed the corresponding times for the linear array, which are $O(p)$. On the other hand, any single node scatter (and, *a fortiori*, multinode broadcast) algorithm takes $\Omega(p)$ time since $p-1$ packets are transmitted by the given node, and these packets must go over the incident links to the node, which are no more than $2d$. Therefore an optimal single node scatter (or gather) algorithm, and an optimal multinode broadcast (or accumulation) algorithm take $\Theta(p)$ time. Consider, finally, the total exchange problem. Exercise 3.6 gives a total exchange algorithm that takes $O(p^{(d+1)/d})$ time, and it turns out that any total exchange algorithm takes $\Omega(p^{(d+1)/d})$ time. To see this latter fact, assume for convenience that p is even and consider a $(d-1)$–dimensional plane that separates the mesh in two identical "halves" (a similar argument applies when p is odd). Each half contains $p/2$ processors, which must receive a total of $(p/2)^2$ packets from the $p/2$ processors of the other half. These packets must be transmitted over the $p^{(d-1)/d}$ communication links connecting the two halves requiring $\Omega(p^{(d+1)/d})$ time. Therefore, an optimal total exchange algorithm for a symmetric d–dimensional mesh takes $\Theta(p^{(d+1)/d})$ time.

Hypercube

Consider the set of all points in d–dimensional space with each coordinate equal to zero or one. These points may be thought of as the corners of a d–dimensional cube. We let these points correspond to processors, and we consider a communication link for every two points differing in a single coordinate. The resulting network is called a *hypercube* or *d–cube*. Fig. 1.3.10 shows a 3–cube and a 4–cube.

Formally, a d–cube is the d–dimensional mesh that has two processors in each dimension, that is, $n_i = 2$ for all i. To visualize better a d–cube, we assume that each processor has an identity number which is a binary string of length d (corresponding to the coordinates of a node of the d–cube). We can construct a hypercube of any dimension

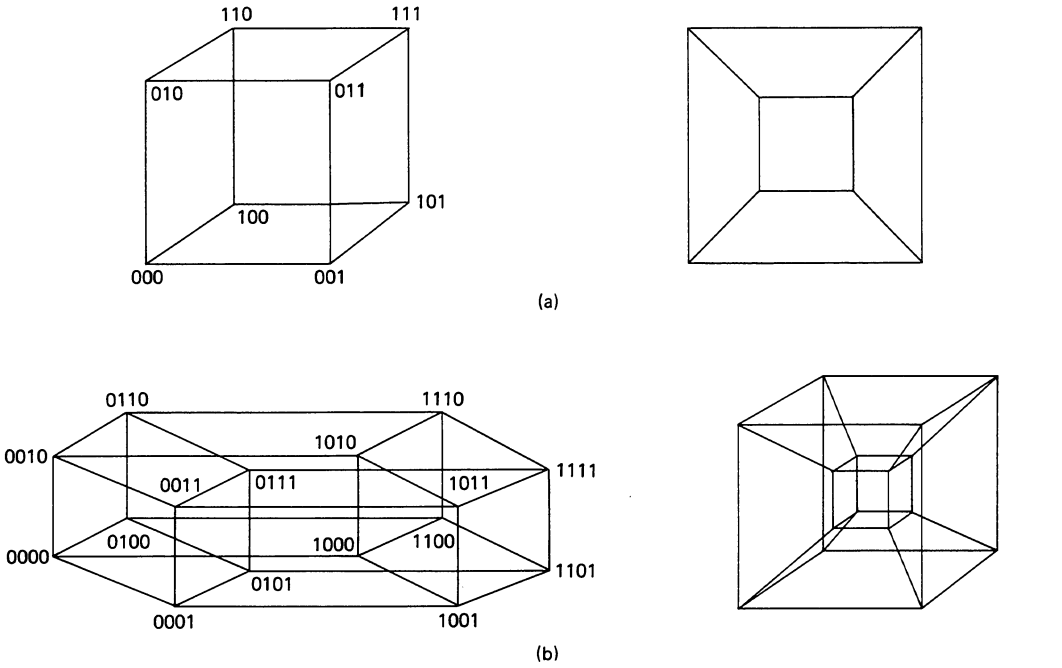


Figure 1.3.10 Two views of a 3–cube and a 4–cube. The cubes have been constructed by connecting the corresponding nodes of two identical lower–dimensional cubes. In the cubes on the left, a node belongs to the first lower–dimensional cube or the second depending on whether its identity has a leading 0 or a leading 1.

by connecting lower–dimensional cubes, starting with a 1–cube. In particular, we can start with two $(d-1)$–dimensional cubes and introduce a link connecting each pair of nodes with the same identity number. This constructs a d–cube with the identity number of each node obtained by adding a leading 0 or a leading 1 to its previous identity, depending on whether the node belongs to the first $(d-1)$–dimensional cube or the second (see Fig. 1.3.10).

The *Hamming distance* between two processors is the number of bits in which their identity numbers differ. Two processors are directly connected with a communication link if and only if their Hamming distance is unity, that is, if and only if their identity numbers differ in exactly one bit. The number of links on any path connecting two nodes cannot be less than the Hamming distance of the nodes. Furthermore, there is a path with a number of links that is equal to the Hamming distance. Such a path can be obtained by switching in sequence the bits in which the identity numbers of the two nodes differ (equivalently, by traversing the corresponding links of the hypercube). For example, in a 4–cube, to go from node (1101) to node (0110), we can first go to (0101), then to (0111), and finally to (0110). It follows that the diameter of a d–cube is d or $\log p$, where $p = 2^d$ is the number of processors.

Hypercube Mappings

The hypercube is a versatile architecture with many attractive features, some of which will be discussed in the sequel. We first illustrate the flexibility of the hypercube by showing how to map a ring and a mesh into it. Mapping a linear array of 2^d nodes into a hypercube amounts to constructing a sequence of 2^d distinct binary numbers with d bits each, with the property that successive numbers in the sequence differ in only one bit. Such sequences are called *Gray codes*, and have been studied extensively in coding theory [Ham86]. We can generate a particular type of Gray code, called a *reflected Gray code* (RGC), by a construction that is similar to the one used for constructing a d–cube from two $(d-1)$–cubes. This code has the property that the first and the last numbers in the sequence also differ in only one bit, so it provides a mapping of a ring with 2^d nodes into the hypercube. We start with the 1–bit Gray code sequence $\{0, 1\}$, and then insert a zero and a one in front of the two elements obtaining the two sequences $\{00, 01\}$ and $\{10, 11\}$. We then reverse the second sequence to obtain $\{11, 10\}$, and then concatenate the two sequences to obtain the 2–bit RGC

$$\{00, 01, 11, 10\}.$$

Generally, given a $(d-1)$–bit RGC

$$\{b_1, b_2, \ldots, b_p\},$$

where $p = 2^{d-1}$ and $b_1, \ldots, b_p$ are binary strings, the corresponding d–bit RGC is

$$\{0b_1, \ldots, 0b_p, 1b_p, \ldots, 1b_1\}.$$

As an example, the 3–bit and the 4–bit RGC, and the corresponding rings on the 3–cube and the 4–cube are shown in Fig. 1.3.11. The above construction maps a ring with 2^d nodes into the d–cube. It is also possible to map a ring of any even number of nodes p into a hypercube with $2^{\lceil \log p \rceil}$ nodes (see Exercise 3.4). It can be shown that every cycle in a hypercube has an even number of nodes (Exercise 3.4), so a ring with an odd number of nodes cannot be mapped into a hypercube.

The preceding recursive construction of the RGC sequence can be generalized in a way that will prove useful later. Let d_a and d_b be positive integers, and let $d = d_a + d_b$. Suppose that $\{a_1, a_2, \ldots, a_{p_a}\}$ and $\{b_1, b_2, \ldots, b_{p_b}\}$ are the d_a–bit and d_b–bit RGC sequences, where $p_a = 2^{d_a}$ and $p_b = 2^{d_b}$. Consider the $p_a \times p_b$ matrix of d–bit strings $\{a_i b_j \mid i = 1, 2, \ldots, p_a,\ j = 1, 2, \ldots, p_b\}$

$$\begin{bmatrix} a_1b_1 & a_1b_2 & \ldots & a_1b_{p_b} \\ a_2b_1 & a_2b_2 & \ldots & a_2b_{p_b} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p_a}b_1 & a_{p_a}b_2 & \ldots & a_{p_a}b_{p_b} \end{bmatrix}.$$

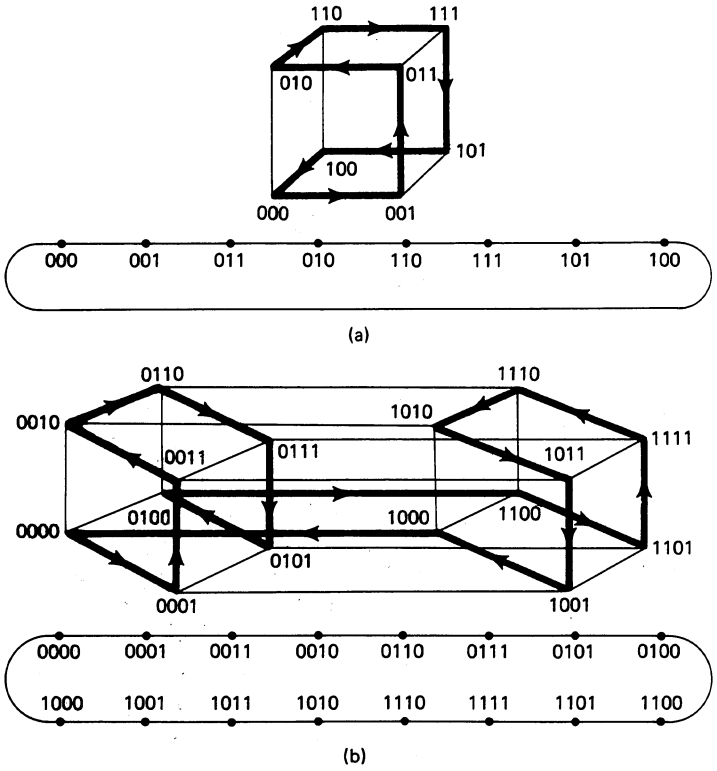


Figure 1.3.11 Reflected Gray code sequences, and the corresponding mappings of rings on (a) a 3–cube and (b) a 4–cube.

It can be proved that we can obtain the d–bit RGC sequence by sequentially traversing the rows of this matrix alternately from left to right, and from right to left, as shown:

$$\begin{bmatrix} a_1b_1 & \Longrightarrow & a_1b_2 & \Longrightarrow & \cdots & \Longrightarrow & a_1b_{p_b} \\ & & & & & & \Downarrow \\ a_2b_1 & \Longleftarrow & a_2b_2 & \Longleftarrow & \cdots & \Longleftarrow & a_2b_{p_b} \\ \Downarrow & & & & & & \\ a_3b_1 & \Longrightarrow & a_3b_2 & \Longrightarrow & \cdots & \Longrightarrow & a_3b_{p_b} \\ & & & & & & \Downarrow \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ & & & & & & \Downarrow \\ a_{p_a}b_1 & \Longleftarrow & a_{p_a}b_2 & \Longleftarrow & \cdots & \Longleftarrow & a_{p_a}b_{p_b} \end{bmatrix}.$$

An example is given in Fig. 1.3.12(a). A formal proof is obtained using the definition of the RGC sequence and induction on d, as illustrated in Fig. 1.3.12(b). The preceding construction also shows that the nodes of a d–cube can be arranged along a two–dimensional mesh with p_a and p_b nodes in the first and second dimensions, respectively. The (i, j)th

element of the mesh, where $i = 1, 2, \ldots, p_a$ and $j = 1, 2, \ldots, p_b$, is the d–cube node with identity number $a_i b_j$. Each "row" (or "column") of the mesh corresponds to a hypercube with p_b (or p_a, respectively) nodes.

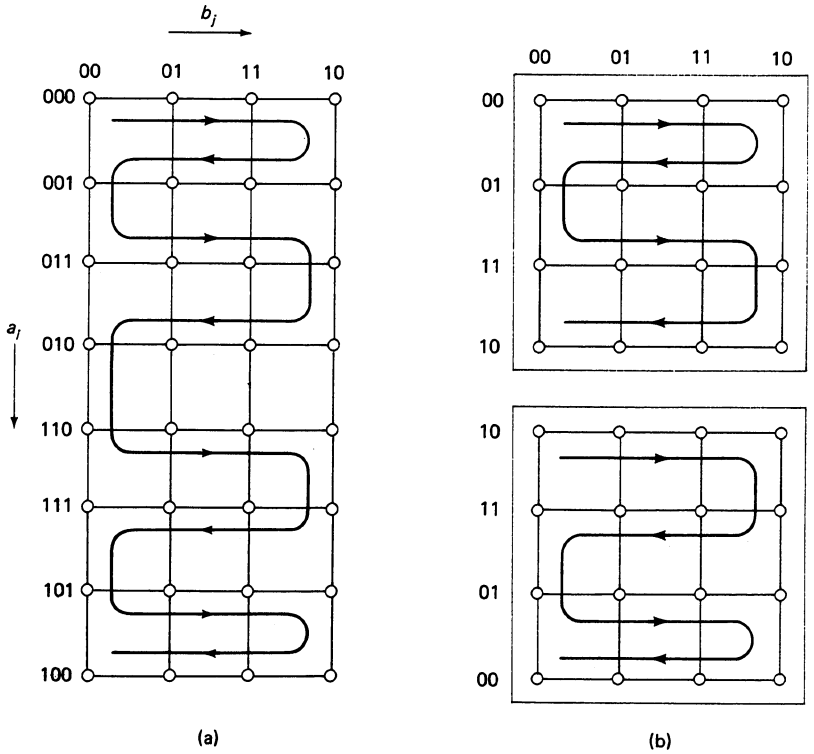
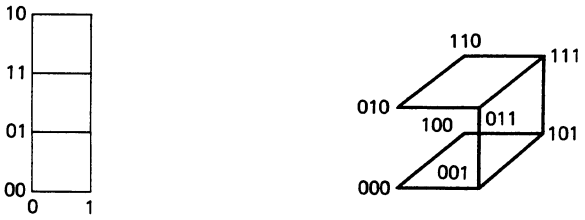
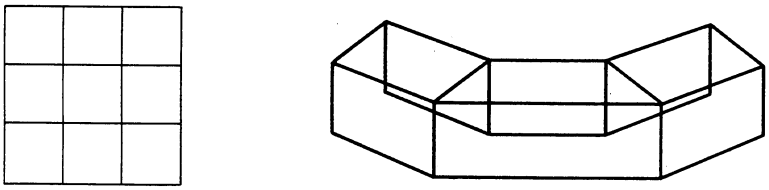


Figure 1.3.12 (a) Arrangement of the nodes of a 5–cube in a two-dimensional mesh with 8 rows and 4 columns. The 5–bit RGC sequence is obtained by starting at the mesh point (000, 00), going right along the first row, then left along the second row, then right along the third row, etc., as indicated by the arrows. Each row corresponds to a 2–cube, and each column corresponds to a 3–cube. (b) Construction of the mapping of an 8×4 mesh (and the corresponding RGC) into a 5–cube, using two 4×4 meshes mapped into the 4–cube. The row numbering of the second mesh is first reversed as shown, and then the two meshes are joined, and a leading 0 or 1 is appended to the row number of the first or second mesh, respectively. This procedure can be generalized to prove by induction that the $(d_a + d_b)$–bit RGC sequence can be constructed from the d_a–bit and the d_b–bit RGC sequences, as stated in the text.

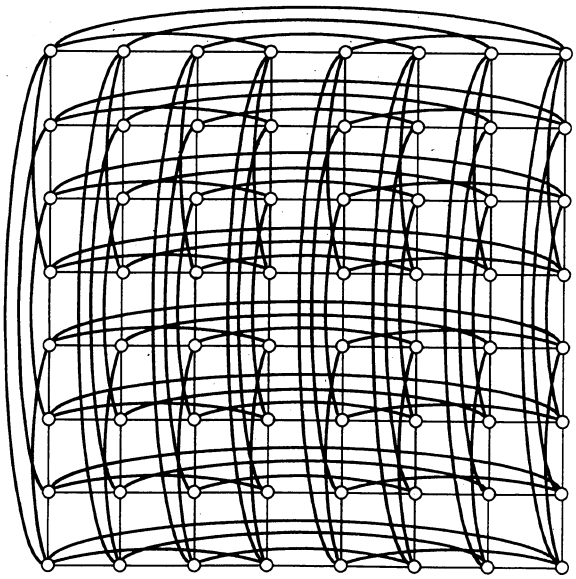
We can similarly obtain a general method for mapping a multidimensional mesh into a hypercube. Suppose we have a k–dimensional mesh with n_i points in the ith dimension, where $i = 1, \ldots, k$. We assume that $n_i = 2^{d_i}$, where d_i is some integer. Thus the number of mesh points is 2^d, where $d = d_1 + d_2 + \cdots + d_k$, and each mesh point can be denoted by $(x_1, x_2, \ldots, x_k)$, where x_i is an integer taking values from 1 to n_i. We map the mesh point $(x_1, x_2, \ldots, x_k)$ into the hypercube node with identity $s_1 s_2 \cdots s_k$, where s_i is the d_i–bit binary string which is the x_ith element of the d_i–bit RGC. Adjacent



(a)

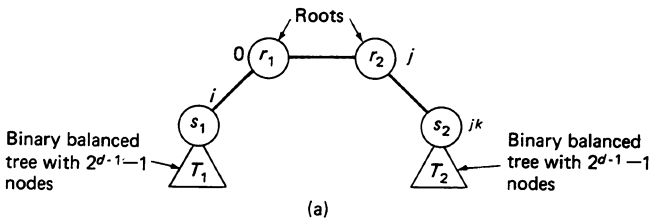


(b)

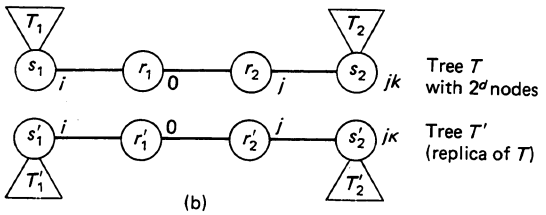


(c)

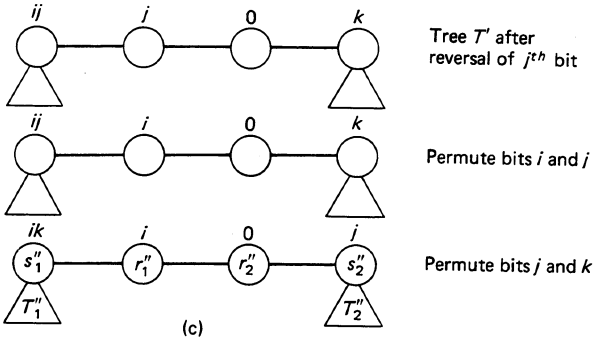
Figure 1.3.13 (a) Mapping a 2 × 4 mesh into a 3–cube. (b) Mapping a 4 × 4 mesh into a 4–cube. (c) A 6–cube arranged as an 8 × 8 mesh. Note that each row and each column of the mesh is a 3–cube.


Roots
0
r_1
r_2
j
i
s_1
s_2
jk
Binary balanced tree with $2^{d-1}-1$ nodes
T_1
T_2
Binary balanced tree with $2^{d-1}-1$ nodes

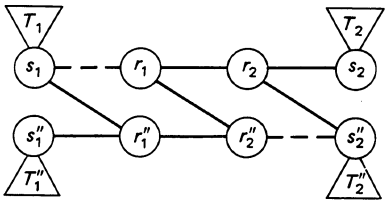
(a)


T_1
T_2
s_1
r_1
r_2
s_2
i
0
j
jk
Tree T with 2^d nodes
s'_1
r'_1
r'_2
s'_2
jK
Tree T' (replica of T)
T'_1
T'_2

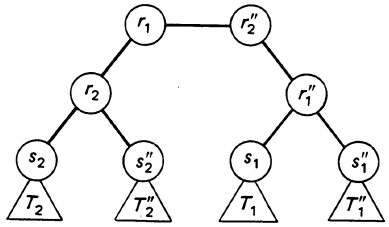
(b)


ij
j
0
k
Tree T' after reversal of j^{th} bit
i
Permute bits i and j
ik
s''_1
r''_1
r''_2
s''_2
T''_1
T''_2
Permute bits j and k

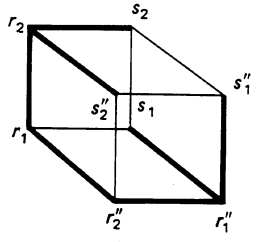
(c)


T_1
T_2
s_1
r_1
r_2
s_2
s''_1
r''_1
r''_2
s''_2
T''_1
T''_2

(d)


r_1
r''_2
r_2
r''_1
s_2
s''_2
s_1
s''_1
T_2
T''_2
T_1
T''_1

(e)


s_2
r_2
s''_1
s''_2
s_1
r_1
r''_2
r''_1

(f)

Figure 1.3.14 Recursive construction of a mapping of a two–rooted binary balanced tree into a hypercube. (a) A two–rooted binary tree with 2^d nodes mapped into a d–cube. Node r_1 is mapped into $(00\cdots0)$; nodes s_1 and r_2 are mapped into the d–cube nodes with the ith and jth bits set, respectively; node s_2 is mapped into the node with the jth and kth bits set. (b) Method for constructing a mapping of a two–rooted tree with 2^{d+1} nodes into the $(d+1)$–cube starting with two trees T and T', mapped into the d–cube. Three tranformations are applied sequentially to the node identity numbers of T', all of which yield mappings of T' into the d–cube. These are a reversal of the jth bit, a permutation of the ith and jth bits, and a permutation of the jth and kth bits [see part (c)]. The identity number for node r_2'' is now $(00\cdots0)$. The identities of nodes r_1'', and s_2'' differ from that of r_2'' in the ith and jth bit, respectively. The identity of node s_1'' differs from that of r_2'' in the ith and kth bits. A leading zero (one) is next appended to the identity number of each node of T (T', respectively, as transformed above). We now introduce the links (s_1, r_1''), (r_1, r_2''), (r_2, s_2'') as shown in part (d), thereby obtaining a mapping of the two–rooted tree shown in (e) into the $(d+1)$–cube. (f) Illustration of the mapping for $d = 2$.

nodes in the mesh differ by a unit in a single coordinate, say x_i, so the corresponding strings s_i are adjacent elements of the RGC sequence. Therefore, the corresponding nodes of the d–cube differ by a single bit in their identities, and must be adjacent. The desired mapping has thus been obtained. Fig. 1.3.13 illustrates the mapping for meshes in two dimensions. Note that a mesh with n_i points in the ith dimension can be mapped into a mesh with $2^{\lceil \log n_i \rceil}$ points in the ith dimension, and therefore can be mapped into a d–cube, where $d = \sum_{i=1}^{k} \lceil \log n_i \rceil$. Note also that because the first and the last elements of a RGC sequence differ in a single bit, the mapping just given can be used to map the corresponding mesh with wraparound (cf. Fig. 1.3.9) into the hypercube.

One can show that a complete binary tree, that is, a binary balanced tree with $2^d - 1$ nodes, cannot be mapped into a d–cube if $d \geq 3$ [BhI85]. On the other hand, a related tree of 2^d nodes, called a *two–rooted binary balanced tree*, can be mapped on the d–cube [BhI85]. This tree is obtained from a binary balanced tree by replacing the root node with two root nodes, each connected with the other and also connected with one binary balanced subtree as shown in Fig. 1.3.14. The construction of this mapping proceeds recursively using two trees each mapped into a d–cube to construct a tree mapped into the $(d+1)$–cube, starting with $d = 2$ (see Fig. 1.3.14). From the point of view of communication, we may consider the two root nodes, together with the link connecting them, as a single node that emulates the function of the (single) root of a binary balanced tree. Thus, using this mapping, it is possible to execute on a hypercube algorithms that are naturally suited for binary tree topologies.

Hypercube Communications

We now consider issues of communication. We first note that in contrast with tree topologies, the d–cube provides several "independent" paths between any pair of nodes (that is paths that do not share any links). The number of such paths is at most d, since there are d links incident to each node. It turns out that there are exactly d such paths (see Fig. 1.3.15). These paths, in addition, do not share any node other than the two end nodes, which shows that the node connectivity of the d–cube is d. If the identity

numbers of the two end nodes differ by k bits, then k of the independent paths have k links, and the remaining $d-k$ paths have $k+2$ links (see Fig. 1.3.15). This implies that simultaneous communication between two nodes of a hypercube along several paths can be done very efficiently.

Another property of the d–cube is that for each node i, there is a spanning tree rooted at i, and providing a path of d links or less from i to every node. Such a tree is constructed as shown in Fig. 1.3.16, and can be used for a single node broadcast from the root to all nodes that takes d time units. This is an improvement by a factor $(2^d - 1)/d$ over the corresponding time for the linear array with 2^d nodes. The same tree can be used to solve the dual problem of a single node accumulation in d time units. The two–rooted tree of Fig. 1.3.14 can also be used for the same purpose in place of the spanning tree of Fig. 1.3.16.

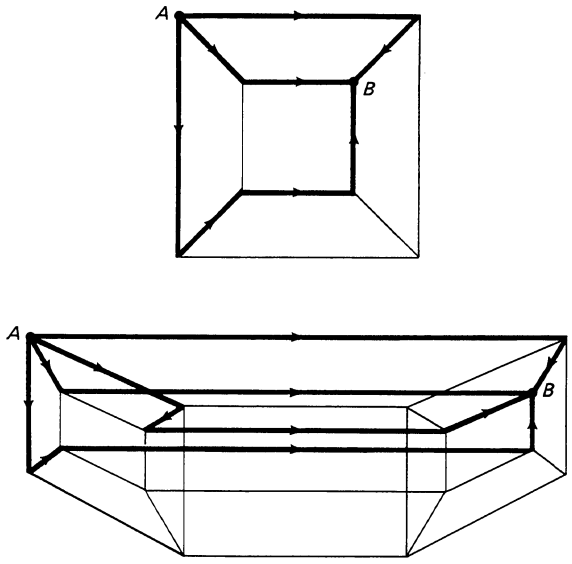


Figure 1.3.15 Construction of d independent paths connecting two nodes A and B of the d–cube with identity numbers differing in k bits [SaS88]. Without loss of generality, we assume that the first k bits of the identity numbers of A and B are different, and the remaining $d-k$ bits are the same.

We first construct k paths from A to B having k links each. The ith path is constructed as follows: start with the identity of A; reverse sequentially bit i through k; reverse sequentially bits 1 through $i-1$.

We next construct $d-k$ paths from A to B having $k+2$ links each. The ith path is constructed as follows: start with the identity of A; reverse the $(k+i)$th bit; reverse sequentially bits 1 through k; reverse again the $(k+i)$th bit.

It can be seen that all these paths do not share any node other than A and B, proving that the node connectivity of a d–cube is d. The figure illustrates the paths for a pair of nodes in the 3–cube and in the 4–cube.

Consider next the time needed for a multinode broadcast, whereby each processor sends a packet to every other processor. As in linear arrays, it is possible to exploit parallel communication on the links. In a d–cube, each node can receive at most d new packets simultaneously along its d incident links, and, since a separate packet is to be received from each of the $(2^d - 1)$ other nodes, we see that (assuming unit time for each packet transmission) any multinode broadcast algorithm takes at least $\lceil (2^d - 1)/d \rceil$ time. There are algorithms that attain this lower bound, and are therefore optimal. We construct such algorithms by means of a general procedure that generates multinode broadcast algorithms, starting from a single node broadcast algorithm and exploiting the symmetry of the network (see also Exercises 3.7 and 3.15).

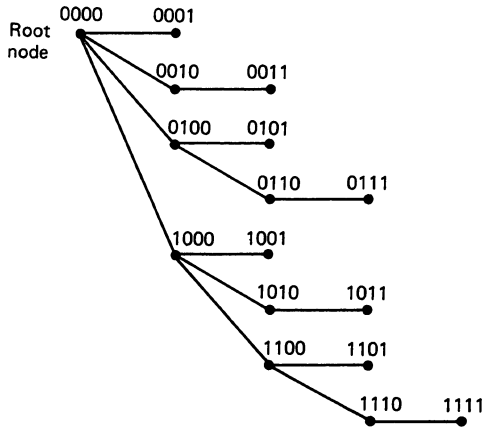


Figure 1.3.16 Spanning tree of a d–cube that is rooted at node $(00\cdots0)$, and provides a path of d links or less from the root node to every other node. The figure shows one possible construction for $d = 4$. The tree is constructed sequentially starting from the root by using the rule that the identities of the children of each node are obtained by reversing one of the zero bits of the identity of the parent that follow the rightmost unity bit. The leaf nodes are the ones that have one as the final bit in their identity. Using this tree, a single node broadcast from the root to all nodes, and a single node accumulation take $\Theta(d)$ communication time. The tree corresponding to an arbitrary root node with identity i can be obtained from the tree corresponding to the node with identity $(00\cdots0)$ by addition (mod 2) of the identity of each node on the tree with the identity of node i. [Here we use the fact that any two node identities x and y differ in exactly the same bits as $i \oplus x$ and $i \oplus y$, where $z \oplus w$ denotes the d-bit string obtained by performing modulo 2 addition of the kth bit of z and w for $k = 1, 2, \ldots, d$. Furthermore, we have $i \oplus (00\cdots0) = i$. As a result, if all identities j in the spanning tree shown are replaced by $i \oplus j$, all links shown will continue to be hypercube links and the resulting spanning tree will be rooted at the node with identity i.]

We represent an algorithm that broadcasts a packet from node $(00\cdots0)$ to all other nodes in m time units by a sequence of disjoint sets of directed links $A_1, A_2, \ldots, A_m$. Each A_i is the set of links on which transmission of the packet begins at time $i-1$ and ends at time i. We impose on the sets A_i certain consistency requirements for accomplishing the single node broadcast. In particular, if S_i (E_i) is the set of identity numbers of the start (end, respectively) nodes of the links in A_i, we must have $S_1 = \{(00\cdots0)\}$, and $S_i \subset \{(00\cdots0)\} \cup \left(\cup_{k=1}^{i-1} E_k\right)$. Furthermore, every nonzero node identity must belong to some E_i. The set of all nodes together with the set of links $\left(\cup_{i=1}^{m} A_i\right)$ must form a subgraph which is a spanning tree [see Fig. 1.3.17(a)].

Consider now a d–bit string t representing the identity number of some node on the d–cube. For any node identity z, we denote by $t \oplus z$ the d–bit string obtained by performing modulo 2 addition of the jth bit of t and z for $j = 1, 2, \ldots, d$. It can be seen that an algorithm for broadcasting a packet from the node with identity t is specified by the sets

$$A_i(t) = \{(t \oplus x, t \oplus y) \mid (x, y) \in A_i\}, \qquad i = 1, 2, \ldots, m,$$

where $A_i(t)$ denotes the set of links on which transmission of the packet begins at time $i-1$ and ends at time i. The proof of this is based on the fact that $t \oplus x$ and $t \oplus y$ differ in a particular bit if and only if x and y differ in the same bit, so $(t \oplus x, t \oplus y)$ is a link if and only if (x, y) is a link (see also Fig. 1.3.16).

We now describe a procedure for generating a multinode broadcast algorithm specified by the sets $A_i(t)$ for all possible values of i and t, starting from a single node broadcast algorithm specified by the sets $A_1, A_2, \ldots, A_m$. For any link (x, y), let $r_i(x, y)$ be the number of node identities t for which $(x, y) \in A_i(t)$, or, equivalently, $x = t \oplus w$

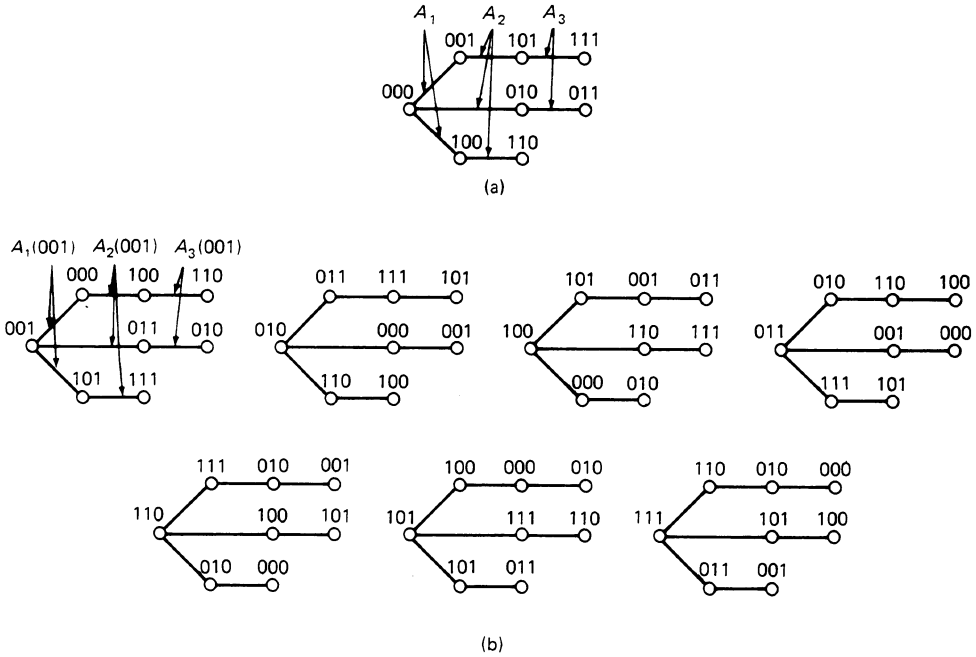


Figure 1.3.17 Generation of a multinode broadcast algorithm for the d–cube, starting from a single node broadcast algorithm. (a) The algorithm that broadcasts a packet from the node with identity $(00\cdots0)$ to all other nodes is specified by a sequence of sets of directed links $A_1, A_2, \ldots, A_m$. Each A_i is the set of links on which transmission begins at time $i-1$ and ends at time i. (b) A corresponding broadcast algorithm for each root node identity t is specified by the sets of links

$$A_i(t) = \{(t \oplus x, t \oplus y) \mid (x, y) \in A_i\},$$

where we denote by $t \oplus z$ the d–bit string obtained by performing modulo 2 addition of the jth bit of t and z for $j = 1, 2, \ldots, d$. The multinode broadcast algorithm is divided in m stages. Within stage i, the packet of each t is transmitted over the links in $A_i(t)$. Stage i takes time T_i. The figure shows the construction for an example where $d = 3$, $T_1 = 1$, $T_2 = 2$, $T_3 = 1$, and the multinode broadcast algorithm takes 4 time units. If the link $(000, 010)$ belonged to A_1 instead of A_2, the required time would be the optimal 3 time units.

and $y = t \oplus z$ for some link $(w, z) \in A_i$. For each of these node identities t, $A_i(t)$ specifies that there is a packet of t to be transmitted over link (x, y). It follows that for a fixed i, if transmission starts simultaneously in all the links of all the sets $A_i(t)$, then, allowing for queueing delays, the transmissions in all these links will be completed within time T_i given by

$$T_i = \max_{(x,y)} r_i(x, y).$$

Therefore, the total time taken by the multinode broadcast is at most $T_1 + T_2 + \cdots + T_m$. Thus efficient multinode broadcast algorithms can be obtained by choosing the sets $A_1, A_2, \ldots, A_m$ of the single node broadcast so that $T_1 + T_2 + \cdots + T_m$ is small. Figure

1.3.17(b) illustrates the sets $A_i(t)$ corresponding to all possible t and the times T_i for the case where $d = 3$. Note that $T_i = 1$ means that given any two links (x, y) and (x', y') of A_i, the bit in which x and y differ is not the same as the bit in which x' and y' differ. Thus, we have $T_2 > 1$ in Fig. 1.3.17 because the links $\big((000), (010)\big)$ and $\big((100), (110)\big)$ belong to A_2 but do not satisfy the preceding requirement. In Fig. 1.3.18 we give a method for selecting A_i so that $T_i = 1$ for all i, and the number of elements in each one of the sets $A_1, A_2, \ldots, A_{m-1}$ is d, while the number of elements in A_m is less than or equal to d. Since the total number of links in the spanning tree specified by $\cup_{i=1}^{m} A_i$ is $2^d - 1$, we conclude that $T_1 + T_2 + \cdots + T_m = m = \lceil (2^d - 1)/d \rceil$, and the corresponding multinode broadcast algorithm takes the optimal time $\lceil (2^d - 1)/d \rceil$.

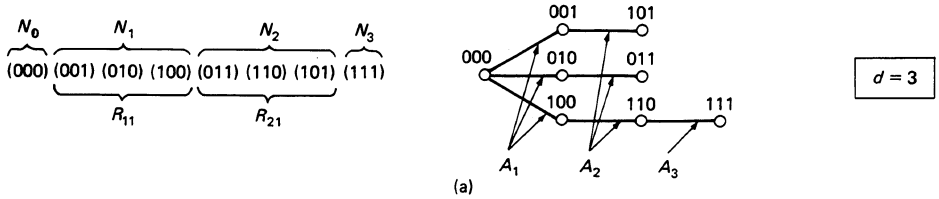
An optimal single node scatter algorithm takes no more than the $\lceil (2^d - 1)/d \rceil$ time taken by an optimal multinode broadcast algorithm. Also, since $(2^d - 1)$ packets must be transmitted along the d incident links of the origin node, any single node scatter algorithm takes at least $\lceil (2^d - 1)/d \rceil$ time (assuming each packet requires unit time for transmission). It follows that $\lceil (2^d - 1)/d \rceil$ is the optimal time to solve the single node scatter problem and its dual, the single node gather problem. The sets of links $A_1, A_2, \ldots, A_m$ constructed in Fig. 1.3.18 can be used to define optimal scatter and gather algorithms for every processor. An alternative, based on a general method for constructing scatter and gather algorithms, is outlined in Exercise 3.9.

Consider next the total exchange problem, whereby each node transmits a separate packet to every other node. We can decompose the d–cube into two $(d-1)$–cubes connected by 2^{d-1} links. We then see that $\left(2^{d-1}\right)^2$ packets from each of the two cubes must be transmitted to the other cube over these 2^{d-1} links. Therefore, any total exchange algorithm cannot take less time than 2^{d-1} units. An algorithm that attains this lower bound within a factor of 2 is given in Fig. 1.3.19. We see, therefore, that an optimal total exchange algorithm takes $\Theta(2^d)$ communication time on the d–cube.

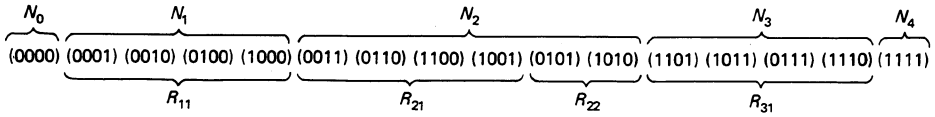
Table 3.1 compares the performance of a ring (or a linear array), a binary balanced tree, a symmetric mesh, and a hypercube for the basic communication problems discussed in this section.

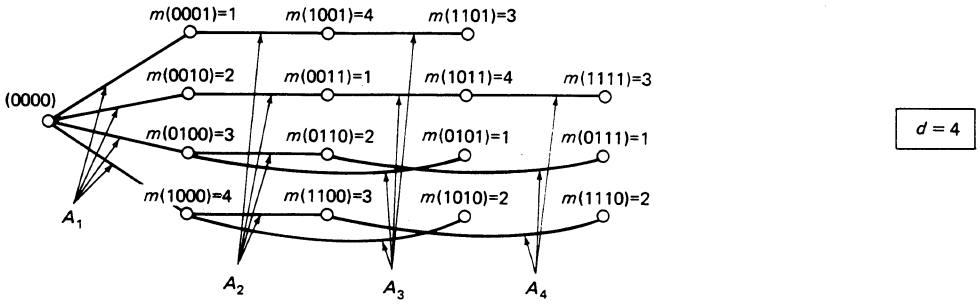
TABLE 3.1 Solution times of optimal algorithms for the basic communication problems using a ring, a binary balanced tree, a d–dimensional symmetric mesh, and a hypercube with p processors. The times given for the ring hold also for a linear array.

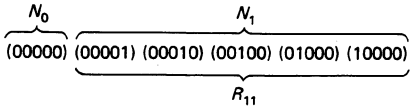
Problem	Ring	Tree	Mesh	Hypercube
Single node broadcast (or single node accumulation)	$\Theta(p)$	$\Theta(\log p)$	$\Theta(p^{1/d})$	$\Theta(\log p)$
Single node scatter (or single node gather)	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$
Multinode broadcast (or multinode accumulation)	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$
Total exchange	$\Theta(p^2)$	$\Theta(p^2)$	$\Theta(p^{(d+1)/d})$	$\Theta(p)$

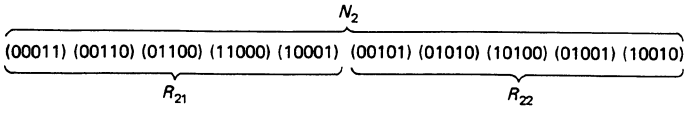

N_0 N_1 N_2 N_3
(000) (001) (010) (100) (011) (110) (101) (111)
R_{11} R_{21}
001 101
000 010 011
100 110 111
A_1 A_2 A_3
$d = 3$

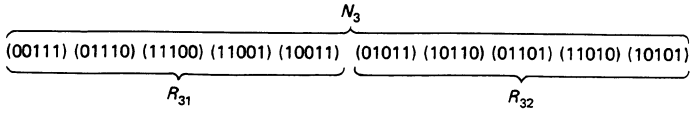
(a)

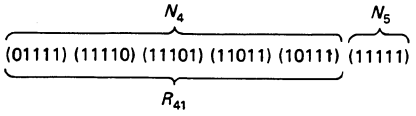

N_0 N_1 N_2 N_3 N_4
(0000) (0001) (0010) (0100) (1000) (0011) (0110) (1100) (1001) (0101) (1010) (1101) (1011) (0111) (1110) (1111)
R_{11} R_{21} R_{22} R_{31}

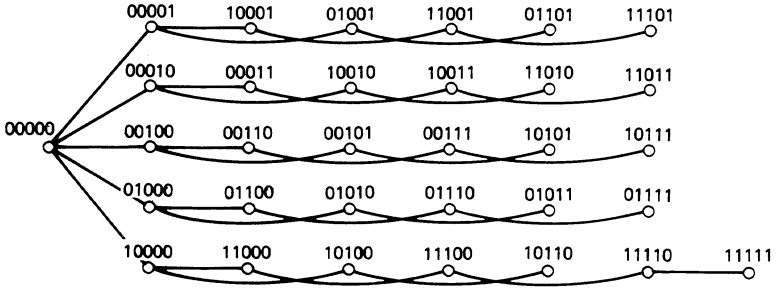

m(0001)=1 m(1001)=4 m(1101)=3
m(0010)=2 m(0011)=1 m(1011)=4 m(1111)=3
(0000)
m(0100)=3 m(0110)=2 m(0101)=1 m(0111)=1
m(1000)=4 m(1100)=3 m(1010)=2 m(1110)=2
A_1 A_2 A_3 A_4
$d = 4$


N_0 N_1
(00000) (00001) (00010) (00100) (01000) (10000)
R_{11}


N_2
(00011) (00110) (01100) (11000) (10001) (00101) (01010) (10100) (01001) (10010)
R_{21} R_{22}


N_3
(00111) (01110) (11100) (11001) (10011) (01011) (10110) (01101) (11010) (10101)
R_{31} R_{32}


N_4 N_5
(01111) (11110) (11101) (11011) (10111) (11111)
R_{41}


00001 10001 01001 11001 01101 11101
00010 00011 10010 10011 11010 11011
00000
00100 00110 00101 00111 10101 10111
01000 01100 01010 01110 01011 01111
10000 11000 10100 11100 10110 11110 11111
$d = 5$

(c)

Figure 1.3.18 Construction of a multinode broadcast algorithm for a d–cube that takes $\lceil (2^d - 1)/d \rceil$ time. Let N_k, $k = 0, 1, \ldots, d$, be the set of node identities having k unity bits and $d - k$ zero bits. The number of elements in N_k is $\binom{d}{k} = d!/\big(k!\,(d-k)!\big)$. In particular, N_0 and N_d contain one element, the strings $(00\cdots0)$ and $(11\cdots1)$, respectively; the sets N_1 and N_{d-1} contain d elements; and for $2 \le k \le d-2$ and $d \ge 5$, N_k contains at least $2d$ elements (when $d = 4$, the number of elements of N_2 is 6, as shown in the figure). We partition each set N_k, $k = 1, \ldots, d-1$, into disjoint subsets $R_{k1}, \ldots, R_{kn_k}$ which are equivalence classes under a single bit rotation to the left, and we select R_{k1} to be the equivalence class of the element whose k rightmost bits are unity. Then we associate each node identity t with a distinct number $n(t) \in \{0, 1, 2, \ldots, 2^d - 1\}$ in the order

$$(00\cdots0)R_{11}R_{21}\cdots R_{2n_2}\cdots R_{k1}\cdots R_{kn_k}\cdots R_{(d-2)1}\cdots R_{(d-2)n_{d-2}}R_{(d-1)1}(11\cdots1)$$

[i.e., $n(00\cdots0) = 0$, $n(11\cdots1) = 2^d - 1$, and the other node identities are numbered consecutively in the above order between 1 and $2^d - 2$]. Let

$$m(t) = 1 + \big[(n(t) - 1)(\text{mod d})\big].$$

Thus the sequence of numbers $m(t)$ corresponding to the sequence of node identities $R_{11}R_{21}\ldots R_{(d-1)1}$ is $1, 2, \ldots, d, 1, 2, \ldots, d, 1, 2, \ldots$ (cf. the figure for the case $d = 4$). We specify the order of node identities within each set R_{kn} as follows: the first element t in each set R_{kn} is chosen so that the relation

$$\text{the bit in position } m(t) \text{ from the right is a one} \qquad (*)$$

is satisfied, and the subsequent elements in R_{kn} are chosen so that each element is obtained by a single bit rotation to the left of the preceding element. Also, for the elements t of R_{k1}, we require that the bit in position $m(t) - 1$ [if $m(t) > 1$] or d [if $m(t) = 1$] from the right must be a zero. Then property $(*)$ is satisfied for all elements of all sets R_{kn} (see the figure for the case $d = 4$). For $i = 1, 2, \ldots, \lceil (2^d - 1)/d \rceil - 1$, define

$$E_i = \{t \mid (i-1)d + 1 \le n(t) \le id\},$$

and for $i = 0$, and $i = m = \lceil (2^d - 1)/d \rceil$, define

$$E_0 = \{(00\cdots0)\}, \qquad E_m = \{t \mid (m-1)d + 1 \le n(t) \le 2^d - 1\}.$$

We define the set of links A_i as follows:

For $i = 1, 2, \ldots, m$, each set A_i consists of the links that connect the node identities $t \in E_i$ with the corresponding node identities of $\cup_{k=0}^{i-1} E_k$ obtained from t by reversing the bit in position $m(t)$ [which is always a one by property $(*)$]. In particular, the node identities in each set R_{k1} are connected with corresponding node identities in $R_{(k-1)1}$, because, by construction, the bit in position $m(t)$ lies next to a zero for each node identity t in the set R_{k1}. There is an exception to the preceding construction in the case where $m(11\cdots1) = d$. The exception is that bit $d - 1$ of $(11\cdots1)$ (instead of bit d) is reversed to connect to $(101\cdots1)$ [which must be the last element of E_{m-1} because of the rule that the bit in position $m(t) - 1$ is a zero for all $t \in R_{(d-1)1}$ with $m(t) > 1$]; furthermore bit d (instead of bit $d - 1$) of the next to last element of E_m [which must be $(1101\cdots1)$] is reversed to connect to $(0101\cdots1)$. [Without this exception, $(11\cdots1)$ would be connected to $(011\cdots1)$, which is the first element of E_m and therefore does not belong to $\cup_{k=1}^{m-1} E_k$.]

To show that this definition of the sets A_i is legitimate, we need to verify that by reversing the specified bit of a node identity $t \in E_i$, we indeed obtain a node identity t' that belongs to $\cup_{k=1}^{i-1} E_k$, as opposed to E_i. [It cannot belong to E_k for $k > i$, because $n(t') < n(t)$.] In the case where $t = (11\cdots1)$, there is no difficulty if $m(11\cdots1) = d$ because of the way that this exceptional case was handled, while if $m(11\cdots1) < d$, it is seen that $(11\cdots1)$ is connected to the node $t' \in E_{m-1}$ for which $m(t') = m(11\cdots1) + 1$. In the case $t \ne (11\cdots1)$, it is sufficient to show that $n(t) - n(t') \ge d$. We consider two cases: a) If $t \in R_{kn}$ for some $n > 1$, then all of the d elements of R_{k1} are between t' and t, and the inequality $n(t) - n(t') \ge d$ follows. b) If $t \in R_{k1}$ then $t' \in R_{(k-1)1}$ and all of the elements of the sets $R_{(k-1)2}, \ldots, R_{(k-1)n_{k-1}}$ are between t' and t. There are $\binom{d}{k-1} - d$ such elements. If $2 < k < d$ and $d \ge 5$, it can be verified that $\binom{d}{k-1} - d \ge d$ and we are done. The cases $d = 3$ and $d = 4$ can be handled individually (see the figure). The cases $k = 1, 2$ create no difficulties because $R_{11} = E_1$, $R_{21} = E_2$. We finally notice that any two links in A_i correspond to reversals in different bit positions, so that $T_i = 1$ for all i.

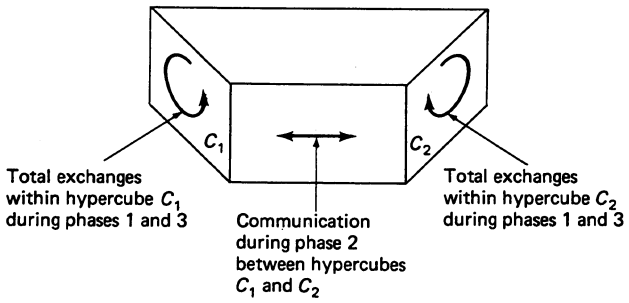


Figure 1.3.19 Recursive construction of a total exchange algorithm for the d–cube requiring time

$$T_d \leq 2^d - 1,$$

which is within a factor of 2 of the lower bound of 2^{d-1}. When $d = 1$, the obvious total exchange algorithm takes time $T_1 = 1$, so the above inequality holds for $d = 1$. Assuming we have a total exchange algorithm for the d–cube satisfying the inequality, we will construct a corresponding total exchange algorithm for the $(d+1)$–cube. Indeed let the $(d+1)$–cube be decomposed into two d–cubes denoted C_1 and C_2. The algorithm consists of three phases, the first two of which are carried out simultaneously. In the first phase, there is a total exchange within each of the cubes C_1 and C_2 (each node in C_1 exchanges its packets with the other nodes in C_1 and similarly for C_2). In the second phase, each node transmits to its counterpart node in the opposite d–cube all of the 2^d packets that are destined for the nodes of the opposite d–cube. In the third phase, there is a total exchange in each of the two d–cubes of the packets received in phase two. Phases 1 and 2 are carried out simultaneously. Since phase 1 is completed in time T_d, which is less than 2^d by the induction hypothesis, both phases 1 and 2 are completed by time 2^d. Phase 3 takes time T_d, and the entire algorithm takes time $T_{d+1} \leq T_d + 2^d \leq 2^{d+1} - 1$, where the last inequality follows from the induction hypothesis. The induction is complete.

Vector Shift on a Hypercube

We next consider a problem of redistributing data among the hypercube processors, which arises sometimes in connection with matrix computations. We mentioned earlier that a ring with 2^d nodes can be mapped into a d–cube, so that node i of the ring $(i = 0, \ldots, 2^d - 1)$ is mapped to the node whose identity is the $(i + 1)$st element of the d–bit RGC sequence. Let us number the nodes of a d–cube as $0, \ldots, 2^d - 1$ according to this mapping. Given some $m \in \{1, 2, 3, \ldots, 2^d - 1\}$, the problem is to send a packet from node i to node $(i + m)(\text{mod } 2^d)$, and to do this simultaneously for all nodes $i = 0, \ldots, 2^d - 1$. We call this the *generalized vector shift problem* because it arises when we have a 2^d–dimensional vector, which is distributed among the processors of a ring so that processor i holds coordinate i, and we want to shift the vector cyclically by m positions while preserving the property that processor i holds coordinate i. We will provide a generalized vector shift algorithm that takes $O(d)$ communication time. For this, we need an interesting class of ring mappings into the hypercube, which we now describe.

Consider the RGC sequence and the ring that it defines on the hypercube as just discussed. Let us define the *logical distance* of two nodes i and j to be the distance between i and j on the ring, and let us define the *physical distance* of i and j to be the minimal number of links that must be traversed on the d–cube to go from i to j (i.e., the Hamming distance of i and j). An important fact is that *two nodes at logical distance* 2^k, $k = 1, \ldots, d-1$, *are at physical distance* 2. The proof is given in Fig. 1.3.20, where it is also shown that for each $k = 0, 1, \ldots, d-1$, the nodes that are at logical distance 2^k from each other form a system of subrings called *subrings of level* k, and each subring has 2^{d-k} nodes. The subrings of level k can also be visualized from the mapping of the $2^{k-1} \times 2^{d-k+1}$ mesh on the d–cube given in Fig. 1.3.20. For $k > 0$, each subring consists of the elements of a column corresponding to either the even–numbered rows or the odd–numbered rows. Each node is on exactly one subring of level k for each k, and successive nodes on each subring are at a physical distance of 2 from each other if $k > 0$, and at a physical distance of 1 if $k = 0$; see Fig. 1.3.21. This shows that each

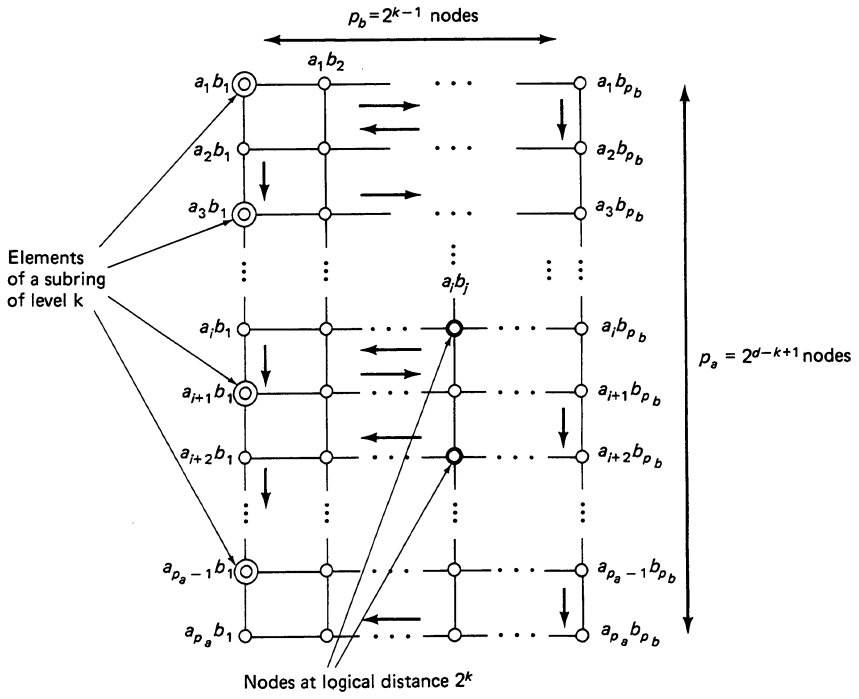


Figure 1.3.20 Proof that two nodes at logical distance 2^k are at physical distance 2 on the hypercube. Consider the mapping of the $p_a \times p_b$ mesh into a d–cube as shown, where $p_a = 2^{d-k+1}$ and $p_b = 2^{k-1}$. Then $\{a_1, a_2, \ldots, a_{p_a}\}$ is the $(d-k+1)$–bit RGC, and $\{b_1, b_2, \ldots, b_{p_b}\}$ is the $(k-1)$–bit RGC. The d–bit strings a_ib_j ordered as in the figure are the elements of the d–bit RGC. Two nodes at logical distance 2^k have identity numbers of the form a_ib_j and $a_{i+2}b_j$ as shown in the figure, and are, therefore, at physical distance 2.

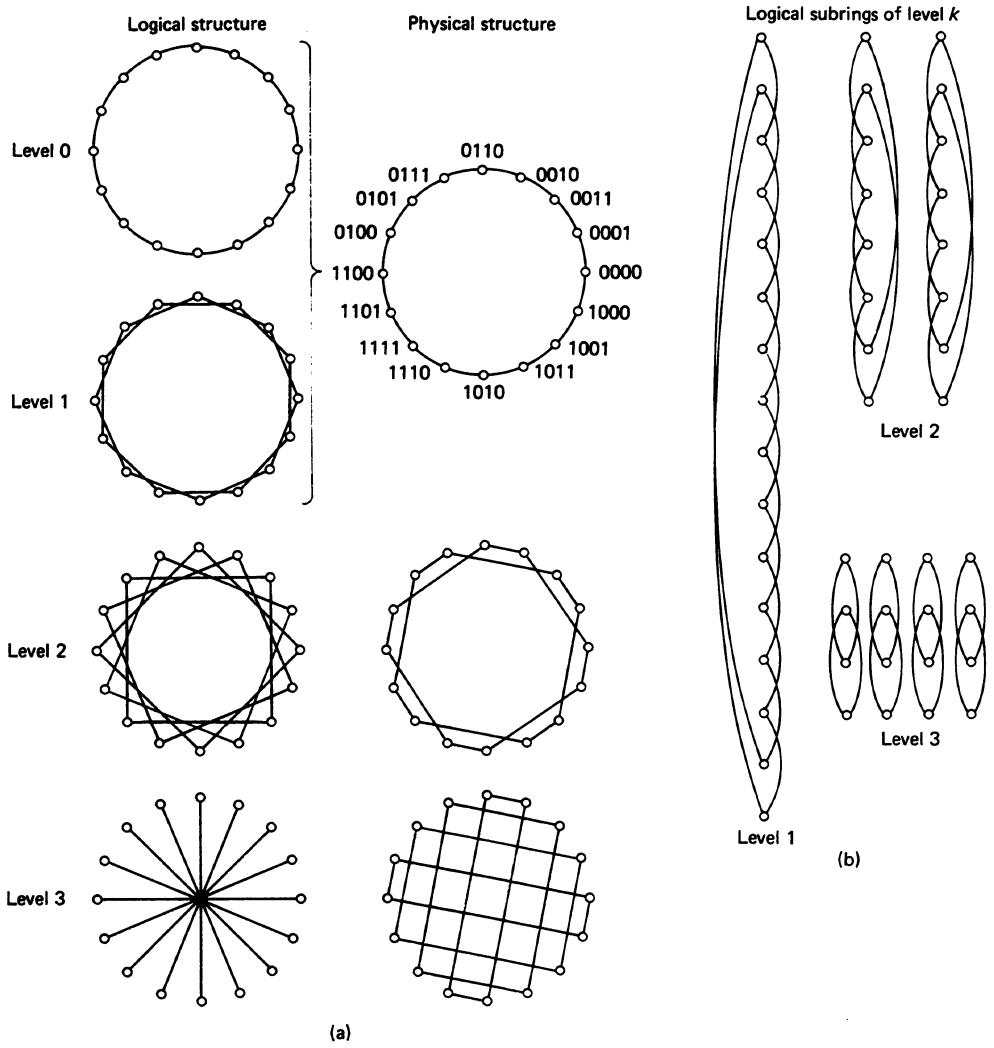


Figure 1.3.21 (a) Subrings of level $k = 0, 1, 2, 3$ on the 4-cube (from [McV87]). (b) The same subrings imbedded in the corresponding $2^{k-1} \times 2^{5-k}$ meshes for $k = 1, 2, 3$.

node i can send a packet to nodes $(i + 2^k) \pmod{2^d}$ and $(i - 2^k) \pmod{2^d}$ simultaneously with every other node over a path with two links if $k > 0$, or one link if $k = 0$.

An algorithm for solving the generalized vector shift problem is now clear. If $b_{d-1}b_{d-2} \cdots b_0$ is the binary representation of m, we move the packets successively on the subrings of the levels i that correspond to nonzero bits b_i. In particular, the packet of node i is sent to node $(i + m) \pmod{2^d}$ by first sending it to node $(i + 1) \pmod{2^d}$ on the zero level ring (if $b_0 = 1$), then to node $(i + b_0 + 2b_1) \pmod{2^d}$ on a level 1 subring (if $b_1 = 1$), then to node $(i + b_0 + 2b_1 + 4b_2) \pmod{2^d}$ on a level 2 subring (if $b_2 = 1$), etc.

The packet of node i traverses $b_0 + 2\sum_{k=1}^{d-1} b_k$ links, and travels simultaneously with the packet of every other node j. Thus, the algorithm takes $b_0 + 2\sum_{k=1}^{d-1} b_k$ time units, which is $O(d)$ as claimed earlier. This time, compared with the corresponding time when a ring of 2^d nodes is used for communication, is found much faster for $1 << m << 2^d - 1$. It turns out that the worst case communication time for the generalized vector shift problem can be reduced from $2d - 1$ to d via an algorithm that uses backward as well as forward shifts on the subrings (Exercise 3.12). For example, a shift of 7 can be effected by a shift on the level 0 subring (logical distance 1), followed by a shift on the level 1 subring (logical distance 2), followed by a shift on the level 2 subring (logical distance 4), requiring a total of 5 time units. Alternately, a shift of 7 can be effected by a shift on the level 3 subring (logical distance 8), followed by a backward shift on the level 0 subring (logical distance 1 backwards), for a total of 3 time units.

Communication Algorithms Using at Most One Link per Node

We next consider briefly the basic communication problems of this section under a potential restriction imposed by the transmission hardware of the interconnection network. We assume in particular that each processor can at any time transmit along at most one of its incident links. Among the algorithms considered so far, only the generalized vector shift algorithm on the d–cube satisfies this restriction.

It turns out that we can characterize the times taken by optimal algorithms for each of the basic communication problems of this section using a broad variety of interconnection networks, and subject to the constraint that each processor can transmit at most one packet at a time. The main results are collected in Table 3.2. To justify these results, suppose that $d(p)$ is the maximum degree of a node of a given type of interconnection network as a function of the number of processors p. Then, any communication algorithm where simultaneous transmission along all the incident links of a node is allowed can be emulated by an algorithm where transmission along only one incident link is allowed at the expense of a slowdown by a factor $d(p)$; in particular, the transmissions within each time unit of the former algorithm can be emulated by corresponding transmissions within $d(p)$ time units in the latter algorithm. It follows that for networks where $d(p)$ is independent of p, such as a ring, a binary balanced tree, and a symmetric mesh, the order of optimal time to solve a communication problem is the same when transmission along multiple incident links is allowed and when it is not. This justifies all entries of Table 3.2, with the exception of the entries for a hypercube where $d(p) = \log p$. For a hypercube, the preceding argument and the results of Table 3.1 show that any single node scatter, multinode broadcast, or total exchange algorithm takes time $O(p)$, $O(p)$, or $O(p \log p)$, respectively. The corresponding lower bounds are obtained by counting the total number of packets that must be transmitted in each problem, and by dividing by the number of nodes (see also Exercises 3.9 through 3.11). The estimate $\Theta(\log p)$ for a single node broadcast for a hypercube in Table 3.2 is shown by using the two–rooted tree mapping of Fig. 1.3.14. We finally note that the estimates of Table 3.2 can be established under the stronger requirement that each node can transmit at most one packet, and, simultaneously, *receive at most one packet* along its incident links. This can be done by modifying some of the algorithms given so that this stronger requirement is

TABLE 3.2 Solution times of optimal algorithms for the basic communication problems using a ring, a binary balanced tree, a d-dimensional symmetric mesh, and a hypercube with p processors, and assuming that a node can transmit along at most one incident link. The times given for the ring hold also for a linear array.

Problem	Ring	Tree	Mesh	Hypercube
Single node broadcast (or single node accumulation)	$\Theta(p)$	$\Theta(\log p)$	$\Theta(p^{1/d})$	$\Theta(\log p)$
Single node scatter (or single node gather)	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$
Multinode broadcast (or multinode accumulation)	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$
Total exchange	$\Theta(p^2)$	$\Theta(p^2)$	$\Theta(p^{(d+1)/d})$	$\Theta(p \log p)$

met, without affecting the corresponding order of solution time. The details are left for the reader (see also Exercises 3.9 through 3.11).

Optimal Algorithms

We have focused so far on the order of time taken by an optimal algorithm for a given type of communication problem and processor interconnection network. In every case, we obtained an algorithm that is optimal within a constant factor; that is, if T_a is the time required by the algorithm, there is a lower bound B on the number of time units required to solve the communication problem, and a scalar c (independent of the number of processors) such that

$$B \leq T_a \leq cB,$$

assuming that each packet transmission requires unit time. On several occasions, we gave exact values for B, T_a, and c. If, for a given algorithm, we have $c = 1$, then the algorithm attains the lower bound for the time to solve the communication problem, and is therefore optimal. Several algorithms given either in the main body of this section or in the exercises are optimal in this sense. Table 3.3 gives the results obtained for the d–cube in this subsection and in Exercises 3.9 through 3.11 for the case where simultaneous transmission along all incident links of a node is allowed and for the case where it is not. It is seen that we have obtained an optimal algorithm in all cases with the exception of a total exchange when simultaneous transmission along all incident links of a node is allowed.

Communication Bottlenecks of Interconnection Networks

We have examined so far in this section several communication problems for a variety of interconnection networks. In each case, we have been able to establish both a lower

TABLE 3.3 Bounds on the optimal times for solving the basic communication problems on a hypercube with p processors for the case where simultaneous transmission along all incident links of a processor is allowed, and for the case where it is not. We assume that each packet requires unit time for transmission on any link.

Problem	Simultaneous Transmission on All Incident Links Allowed		Simultaneous Transmission on Multiple Incident Links Not Allowed	
	Lower Bound	**Upper Bound**	**Lower Bound**	**Upper Bound**
Single node broadcast (or single node accumulation)	$\log p$	$\log p$	$\log p$	$\log p$
Single node scatter (or single node gather)	$\lceil \frac{p-1}{\log p} \rceil$	$\lceil \frac{p-1}{\log p} \rceil$	$p-1$	$p-1$
Multinode broadcast (or multinode accumulation)	$\lceil \frac{p-1}{\log p} \rceil$	$\lceil \frac{p-1}{\log p} \rceil$	$p-1$	$p-1$
Total exchange	$\frac{p}{2}$	$p-1$	$\frac{p}{2}\log p$	$\frac{p}{2}\log p$

bound on the order of time taken by an optimal algorithm and an algorithm that attains this lower bound. It is interesting to note that for a given communication problem, there is a particular characteristic, common to all interconnection networks examined, that determines the lower bound. We may view this characteristic as a *communication bottleneck* associated with the corresponding type of problem. Based on this viewpoint, we can obtain insight on the features of interconnection networks that make them desirable or undesirable for specific types of communication tasks, as we now explain.

In the following discussion we assume that all packet transmissions require one time unit, and that simultaneous transmission along all incident links of a node is allowed. Since the diameter of the network is equal to the time taken by an optimal single node broadcast algorithm with a worst choice of root node, we can view the diameter as the communication bottleneck for the single node broadcast problem (and, therefore, also for the single node accumulation problem).

Consider next the single node gather problem for a node with d incident links (i.e., a degree equal to d). If p is the number of processors, the node must receive $p-1$ packets over its d incident links, so $\lceil (p-1)/d \rceil$ is a lower bound on the solution time of any algorithm. This lower bound is tight for a hypercube, as shown earlier (Table 3.3), and is either tight or nearly tight for the other topologies examined in this section. We conclude that the minimum node degree in an interconnection network is a communication bottleneck for the single node gather (and, therefore, also for the single node scatter problem).

The reasoning used above for the single node gather problem applies also for the multinode accumulation problem, and it can be seen that the minimum node degree is a bottleneck for multinode accumulation (as well as for the multinode broadcast problem).

Consider, finally, the total exchange problem. For any partition of the node set N into two disjoint nonempty subsets N_1 and N_2, let L_{12} be the number of links connecting a node of N_1 with a node of N_2. The number of packets that will travel over these L_{12} links in any total exchange algorithm is at least $|N_1||N_2|$, where $|N_1|$ and $|N_2|$ are the

number of nodes of N_1 and N_2, respectively. Therefore the corresponding solution time is bounded from below by

$$\max_{\text{All partitions } (N_1,N_2)} \left\{ \frac{|N_1||N_2|}{L_{12}} \right\}.$$

This number, called the *cross–section bound*, provides a tight underestimate of the order of time taken by an optimal total exchange algorithm for all the interconnection networks considered in this section, and may be viewed as a communication bottleneck for the total exchange problem. We note, however, that the cross–section bound is not valid when transmission along multiple incident links of a node is not allowed. An appropriate bound can be developed under these circumstances by lower bounding the number of link transmissions in any total exchange algorithm and dividing by the number of nodes. Such a lower bound is tight for the hypercube (see Exercise 3.10).

1.3.5 Concurrency and Communication Tradeoffs

We use the term *concurrency* as a broad measure of the number of processors that are, in some aggregate sense, simultaneously active in carrying out the computations of a given parallel algorithm. The degree of concurrency generally depends on the method by which the overall computation is broken down into smaller subtasks and is divided among the various processors for parallel execution. It is important for efficiency purposes that the computation time of parallel subtasks be relatively uniform across processors; otherwise, some processors will be idle waiting for others to finish their subtasks. This is known as *load balancing*. It is reasonable to conjecture that the number of packet exchanges used to coordinate the parallel subtasks increases with the number of subtasks, and that this is particularly true when the size of the subtasks is relatively uniform. It then follows that as the concurrency of an algorithm increases, the communication penalty for the algorithm also increases. Therefore, as we attempt to decrease the solution time of a given problem by using more and more processors, we must contend with increased communication penalty. This may place an upper bound on the size of problems of a given type that we can realistically solve even with an unlimited number of processors.

For a given problem, there are both general and problem–specific reasons why the communication penalty tends to increase with the number of processors. A first reason is the possibility of *pipelining of computation and communication.* If some of the computation results at a processor can be communicated to other processors while other results are still being computed, the communication penalty will be reduced. This type of pipelining is possible, for example, in relaxation iterations of the form

$$x_i(t+1) = f_i\big(x_1(t), \ldots, x_p(t)\big), \qquad i = 1, \ldots, p \tag{3.4}$$

(cf. the model considered in Subsection 1.2.4), where each x_i is a vector of dimension k that is assigned to a separate processor i and $n = pk$ is the dimension of the problem.

Pipelining of computation and communication is more pronounced when there is a large number of variables assigned to each processor; then the variables that have been already updated within an iteration can be made available to other processors while the updating of other variables is still pending. A second reason is that in many systems, a portion of each packet is used to carry overhead information. The length of this portion is usually fixed and independent of the total length of the packet. This means that there is a gain in efficiency when packets are long, since then the overhead per bit of data is diminished. It is clear that the length of the packets can be made longer if the number of variables updated by each processor using the relaxation iteration (3.4) is larger, since then the values of many variables can be transmitted to other processors as a single packet.

Even in the absence of overhead, and of pipelining of computation and communication, the communication penalty tends to be reduced as the dimension k of the component vectors x_i in the relaxation iteration (3.4) is increased. Suppose that processor i uses Eq. (3.4) to update the k–dimensional vector x_i, with knowledge of the other vectors x_j, $j \neq i$. Suppose also that the computation time for each update is $\Theta(nk)$ [as it will be for example when the function f_i in Eq. (3.4) is linear without any special sparsity structure]. After updating x_i, processor i must communicate the corresponding k variables to all other processors so that the next iteration can proceed. This can be done via a multinode broadcast, and if a linear array is used for this purpose, the optimal communication time is $\Theta(n)$, assuming that communication of k variables over a single link takes $\Theta(k)$ time. Thus, the ratio

$$\frac{T_{COMM}}{T_{COMP}} = \frac{\text{Communication time per iteration}}{\text{Computation time per iteration}}$$

is $\Theta(1/k)$, and the communication penalty becomes relatively insignificant as the number k of variables updated by each processor increases. The ratio T_{COMM}/T_{COMP} is independent of the problem dimension n; it only depends on k, that is, the size of the computation task per iteration for each processor.

A further observation from this analysis is that the speedup obtained through parallelization of the relaxation iteration (3.4) can be increased as the dimension n of the problem increases. In particular, the computation time per iteration on a serial machine is $\Theta(n^2)$ [it is $\Theta(nk)$ based on our earlier hypothesis and, for a serial machine, we have $p = 1$ and $k = n$], so the speedup using a linear array of p processors, each updating $k = n/p$ variables, becomes

$$\frac{\Theta(n^2)}{\Theta(n) + \Theta(nk)} = \Theta(p),$$

where the $\Theta(n)$ and $\Theta(nk)$ terms correspond to the communication time and the computation time, respectively.

We have thus reached the important conclusion that for relaxation iterations of the form (3.4), the communication penalty will not prevent the fruitful utilization of a large number of processors in parallel when the problem is large, even when a linear array (the

"least powerful" network) is used for communication. What is needed, as the dimension of the problem increases, is a proportional increase of the number of processors p of the linear array that will keep the number k of variables per processor roughly constant at a level where the communication penalty is relatively small. Note also that when a hypercube is used in place of a linear array, the optimal multinode broadcast time is $\Theta\big((pk)/\log p\big)$, so the ratio T_{COMM}/T_{COMP} decreases from $\Theta(1/k)$ to $\Theta\big(1/(k\log p)\big)$. Therefore, as the dimension of the problem increases by a certain factor, the number of processors of the hypercube can be increased by a larger factor while keeping the communication penalty at a relatively insignificant level, and increasing the attainable speedup at a faster rate than with a linear array.

The preceding analysis does not assume any special structure for the iteration (3.4) other than the hypothesis that a single variable update takes $\Theta(n)$ time. In many other cases where there is special structure, the ratio T_{COMM}/T_{COMP} is also small for large k. An important example is associated with problems arising from discretization of two–dimensional physical space and with the so called *area–perimeter effect* (see Fig. 1.3.22). As shown in the figure, the number of variables that have to be communicated by a processor is $\Theta(\sqrt{k})$, and the time taken for communication on a mesh network or a hypercube is $\Theta(\sqrt{k})$. The time taken for each variable update is a constant, and the parallel computation time for each iteration is $\Theta(k)$. The ratio T_{COMM}/T_{COMP} is $\Theta(1/\sqrt{k})$. Figure 1.3.23 provides another example of a sparsity structure (block-tridiagonal) where the ratio T_{COMM}/T_{COMP} decreases quickly with k. Fig. 1.3.24 provides an unfavorable type of sparsity structure for matrix–vector multiplication, where T_{COMM}/T_{COMP} is relatively large.

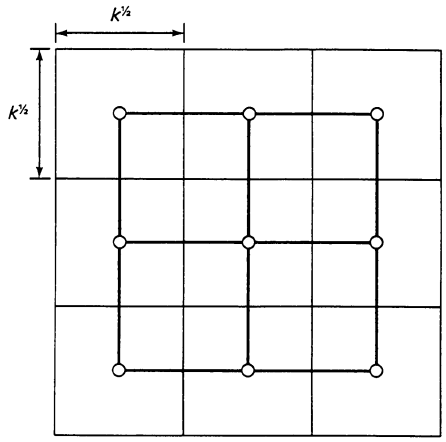


Figure 1.3.22 Structure arising from discretization of 2–dimensional space. Here the variables are partitioned in rectangles of physical space, and we assume that only neighboring variables interact (cf. the example in Subsection 1.2.4). Each rectangle contains k variables, and at the end of a relaxation iteration of the form (3.4), each rectangle must exchange $\Theta(\sqrt{k})$ variables with each of its neighboring rectangles. The communication time per relaxation iteration, using a mesh network for communication, is $\Theta(\sqrt{k})$, but the computation time is $\Theta(k)$.

The preceding discussion has focused on relaxation iterations of the form (3.4) which are important for the purposes of this book as they appear in the context of many algorithms. The conclusion is that with proper selection of the size of the computation task for each processor, the effects of communication can be minimized. Furthermore, as the size of the given problem increases without bound, the speedup can typically also

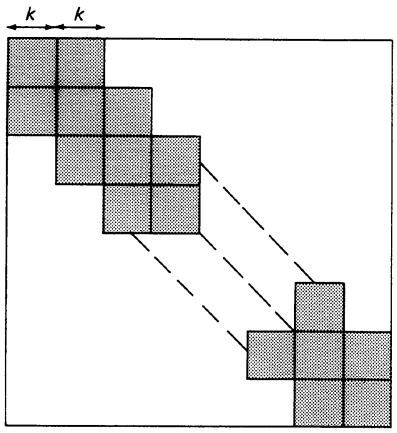


Figure 1.3.23 A block–tridiagonal sparsity structure for matrix–vector multiplication where the communication to computation time ratio is low when using a linear array for communication. Each block is assumed to be dense and to have k variables. Each relaxation iteration of the form (3.4) takes $\Theta(k)$ communication time (assuming a linear array is used), and $\Theta(k^2)$ computation time.

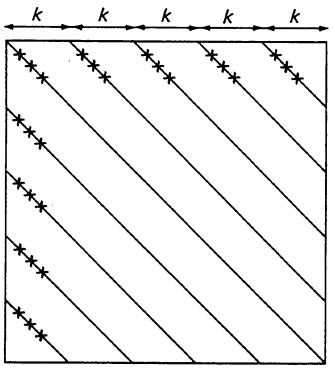


Figure 1.3.24 A matrix sparsity structure for matrix–vector multiplication where the communication to computation time ratio using a linear array of processors is relatively large. Here there are p processors, each computing $k = n/p$ successive coordinates of the product. The communication and the computation times are $\Theta(kp)$.

increase without bound by using an appropriate parallel machine. In other words, there is no *a priori* bound on the attainable speedup that is imposed by the communication requirements.

1.3.6 Examples of Matrix–Vector Calculations

In this subsection, we discuss some generic communication aspects of matrix calculations, and at the same time we illustrate some of the ideas of the previous subsections. We make the same assumptions as in Subsection 1.3.4 regarding the interconnection network of processors used. In particular, we assume that communication can take place simultaneously along all the incident links of a processor, and that in the absence of queueing, the delays of all packets of equal length are equal on all links.

Our principal examples are inner product formation and matrix–vector calculations such as

$$x(t+1) = Ax(t) + b, \qquad \forall\, t = 0, 1, \ldots, \tag{3.5}$$

which arise in iterative methods of the Jacobi and Gauss–Seidel types (Subsection 1.2.4 and Section 2.4). Similar situations arise also in the conjugate gradient method (Section 2.7), and more generally in cases where a sequence of matrix–vector and matrix–matrix multiplications, inner product formations, and vector additions are required with each calculation using the results of the preceding ones. Related examples also arise in other situations involving calculation of the minimum of a set of numbers instead of an inner product (see the material on shortest paths and dynamic programming in Chapter 4). An important characteristic of iterative calculations such as (3.5) is that following an iteration, it is necessary to store the results of the iteration [e.g., the vector $x(t+1)$ of Eq. (3.5)] at the appropriate processors as dictated by the needs of the next iteration. In our analysis, we will not account for the time taken for initial storage of the problem data [e.g., the matrix A, and the vectors b and $x(0)$ of Eq. (3.5)] at the appropriate processors. This additional time is negligible, assuming a large number of iterations are executed; otherwise, the following analysis can be easily modified to take it into account. This time can be reduced by pipelining the initial data input with the algorithmic computation and communication. Schemes that take advantage of this possibility are known as *systolic* algorithms, and are used for fast execution of highly specialized calculations on VLSI chips. Such algorithms are beyond the scope of this book (see [MeC80], [Kun82], [ADM82], and [Kun88]).

Inner Product

Assume that we have a network of p processors, and we want to form the inner product of two vectors a and b in $\Re^n$, where $n \geq p$. For simplicity, we assume that n is divisible by p, and we let $k = n/p$. It is then natural to store at each processor i the k coordinates of a and b numbered $(i-1)k+1$ through ik, to form the partial inner product $c_i = \sum_{j=(i-1)k+1}^{ik} a_j b_j$, and then to accumulate the result using a spanning tree rooted at some node. We recognize this as a single node accumulation problem [cf. Fig. 1.3.4(b)]. The root node can send the final value of the inner product to all other nodes if needed; this is a single node broadcast problem. The following analysis can be generalized to account for this possibility without affecting the associated time estimates, but to simplify matters, we will assume that it is sufficient to obtain the inner product at the root node. An alternative to collecting the sum at a single node and then broadcasting it back to all nodes is given in Exercise 3.22 for the hypercube network.

Suppose that an addition and a multiplication take time α, and that transmission of a partial inner product along a link takes time β. Assume first that the processors are connected in a linear array. Then the optimal choice for the root node is a "middle" node that is at distance no more than $\lfloor p/2 \rfloor$ from every other node. The time to compute the inner product at the root node is then

$$k\alpha + (\alpha + \beta)\left\lfloor \frac{p}{2} \right\rfloor.$$

For a given dimension $n = pk$, this is written as

$$\frac{n\alpha}{p} + (\alpha + \beta)\left\lfloor \frac{p}{2} \right\rfloor. \tag{3.6}$$

We optimize approximately this expression over p by neglecting the fact that p is integer, thereby obtaining the approximate optimum value

$$p \approx \left(\frac{2\alpha n}{\alpha + \beta}\right)^{1/2}. \tag{3.7}$$

Thus, the optimal number of processors is substantially smaller than the maximum possible number n. In particular, when $\beta > (2n-1)\alpha$, it is optimal to avoid the high communication cost associated with parallelism, and to use a single processor. This illustrates the tradeoff between concurrency and communication (cf. Subsection 1.3.5). From Eqs. (3.6) and (3.7) it is seen that the optimal parallel time to calculate the inner product grows with n as $n^{1/2}$ when the processors are connected in a linear array.

Assume now that a hypercube with p nodes is used to compute the inner product along the spanning tree of Fig. 1.3.16. Then it can be seen that the total time is

$$\frac{n\alpha}{p} + (\alpha + \beta)\log p,$$

and the optimal number of processors is given approximately by

$$p \approx \frac{\alpha n}{\alpha + \beta}. \tag{3.8}$$

The optimal parallel time to calculate the inner product grows with n as $\log n$, which is also the rate of growth when n processors are used and communication is assumed instantaneous (cf. Subsection 1.2.3). The optimal number of processors is much larger, and the time to solve the problem is much smaller, than with a linear array, reflecting the fact that communication on the hypercube is much more efficient. The performance gap between the two interconnection networks is narrowed when it is required to calculate several inner products simultaneously. Then a multinode accumulation is required in place of a single node accumulation. This takes little additional communication time on a linear array, but a lot more communication time on a hypercube (cf. the discussion of Subsection 1.3.4, and the following discussion on matrix–vector multiplication).

Matrix–Vector Multiplication

We next consider the parallel calculation of the matrix–vector product Ax on a network of p processors, where A is an $n \times n$ matrix, and x is an n–dimensional vector. We assume that n is divisible by p, and we denote $k = n/p$. There are two basic methods here, depending on whether A is distributed by rows or by columns among the processors.

(a) The *row storage method*, where processor i stores the k rows of A numbered $(i-1)k+1$ through ik as well as the vector x. Processor i then calculates the k coordinates numbered $(i-1)k+1$ through ik of the product Ax. We assume that the product Ax is to be used subsequently by all processors, so each node is required to transmit the updated values of these coordinates to all other nodes. This is a multinode broadcast problem.

(b) The *column storage method*, where processor i stores the k columns of A numbered $(i-1)k+1$ through ik, and the k coordinates of x numbered $(i-1)k+1$ through ik. Processor i then calculates the corresponding k terms in the sum that defines each coordinate of the product Ax. It is then necessary to accumulate the terms corresponding to the coordinates $(i-1)k+1$ through ik at node i. This is a multinode accumulation problem.

Note that converting a matrix from the row storage format to the column storage format requires solving a total exchange problem as defined in Subsection 1.3.4 [see also Exercise 3.10(d)].

There is an interesting duality relation between the times taken by the row and column storage methods, which is illustrated in Fig. 1.3.25. If the matrix A is fully dense, it can be seen that the two methods take comparable times regardless of the type of interconnection network used; they both involve an equal number of multiplications, and either a multinode broadcast or a multinode accumulation, which take equal time (except for the typically negligible time for a few additions) on any interconnection network. This conclusion extends to the case where the sparsity structure of A is the same as the sparsity structure of the transpose of A, for example, when A is symmetric. To see this, note that for each multiplication and addition performed in the case of the row storage method when A is used, there is a corresponding multiplication and addition for the column storage method when the transpose of A is used. Furthermore, if a node i must send a packet to node j in the case of the row storage method [i.e., the submatrix of A corresponding to rows $(j-1)k+1$ through jk and columns $(i-1)k+1$ through ik is nonzero], then node j must participate in the summation relating to the coordinates accumulated at node i in the column storage method when the transpose of A is used.

When A is not symmetric it may be difficult to determine which method is preferable without detailed knowledge of the sparsity structure of A and the interconnection network used. For an example where the row storage method takes more time than the column storage method, consider a matrix A with zero elements everywhere except along the first row and along the diagonal. Then, the row storage method requires a single node gather operation at node 1, whereas the column storage method requires a less time consuming single node accumulation at node 1. By using the transpose of the matrix just described we obtain an example where the row storage method takes less time than the column storage method. We finally note that when using the column storage method, each processor stores only a k–dimensional subvector of x, as opposed to storing the full vector x that is required in the row storage method. Thus the column storage method has an advantage in terms of memory requirements.

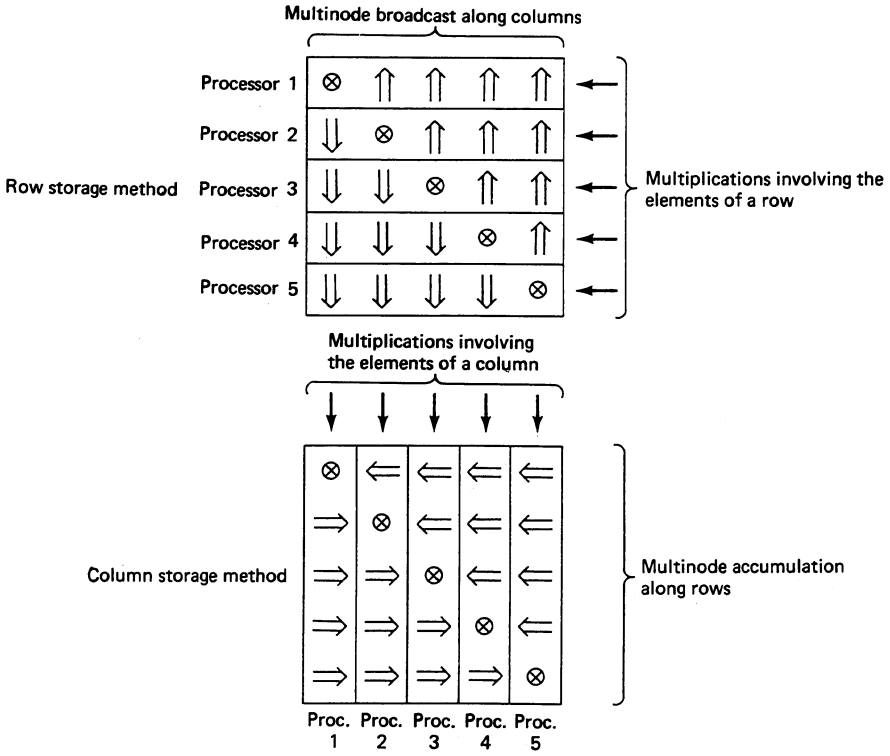


Figure 1.3.25 Duality of matrix–vector multiplication using the row storage and the column storage methods. In the row storage method, each processor performs multiplications involving the stored matrix rows, forms the corresponding coordinates of the product, and broadcasts these coordinates to the other processors. In the column storage method, each processor performs multiplications involving the stored matrix columns, and accumulates the corresponding coordinates of the product. All broadcasts (accumulations) are done in parallel by the processors, so a multinode broadcast (or accumulation, respectively) is required.

For a discussion of additional methods of matrix storage that can take advantage of sparsity structure, and corresponding timing analyses of matrix–vector calculations, we refer the reader to [McV87] and [FJL88].

Consider now the time to compute the product Ax using $p \leq n$ processors arranged in a linear array. We assume that the row storage method is used with each processor storing $k = n/p$ rows. Suppose that the time for an addition and a multiplication is α, and that the time to transmit a packet of k numbers over a link is $\beta + k\gamma$, where α, β, and γ are some positive constants. Suppose also that communication starts only after all computation is completed. Then the time for matrix–vector multiplication using a linear array is

$$\frac{\alpha n^2}{p} + (p-1)\left(\beta + \frac{n\gamma}{p}\right), \tag{3.9}$$

where the first term corresponds to the computation time, and the second term corresponds to the subsequent multinode broadcast needed to store the product Ax at all processors. The time (3.9) is optimized for

$$p \approx n \left(\frac{\alpha - \gamma/n}{\beta} \right)^{1/2},$$

which yields a roughly constant optimal number of rows per processor, $k = n/p \approx (\beta/\alpha)^{1/2}$, when n is large (cf. the discussion of Subsection 1.3.5). The total optimal time grows linearly with n. This is the same order of growth as when communication is instantaneous. Since the optimal multiplication time using any network into which a linear array can be mapped (such as a hypercube) cannot be more than the one using a linear array, and cannot be less than when communication is instantaneous, we conclude that for just about any interconnection network of interest, the time to form the product Ax using an optimally chosen number $p \leq n$ of processors grows linearly with n.

We next consider the case where the number of available processors is larger than n. It was seen in Subsection 1.2.3 that when communication is instantaneous, the product Ax can be computed in $O(\log n)$ time using n^2 processors. We cannot achieve this bound when a linear array is used for communication, since it was seen earlier that the optimal time for an inner product of two n–dimensional vectors (an easier calculation than the matrix–vector product) grows as $n^{1/2}$. We can achieve the bound $O(\log n)$, however, by using a hypercube and the algorithm described in Fig. 1.3.26.

Matrix–Matrix Multiplication

We next discuss the problem of multiplying two $n \times n$ matrices. We assume first that a square mesh of n^2 processors is used. We also assume that the ijth element of each matrix is stored in processor (i, j), and the ijth element of the product must be eventually stored in the same processor (see Exercise 3.16 for the case of a different initial storage specification). Figure 1.3.27 gives an algorithm that takes $O(n)$ time. This is also the bound obtained when the product of two $n \times n$ matrices is formed in the usual way using n^2 processors with instantaneous communication. Note also that the kth power of an $n \times n$ matrix can be computed in time $O(n \log k)$ using a square mesh of n^2 processors with the ijth processor holding initially the ijth element of the matrix. This is done by successive squaring of the matrix if k is a power of 2. If k is not a power of 2, a similar procedure works in the same order of time; for example, to calculate A^{11}, we calculate successively A^2, A^4, A^8, $A^{10} = A^2 A^8$, and $A^{11} = AA^{10}$ (see the discussion of Subsection 1.2.3).

We finally consider the matrix–matrix multiplication problem when n^3 processors are available. It was seen earlier that matrix–vector multiplication can be performed in time $O(\log n)$ using a hypercube of n^2 processors. The product of two $n \times n$ matrices A and B is the matrix having as columns the matrix–vector products $Ab_1, \ldots, Ab_n$, where $b_1, \ldots, b_n$ are the columns of B. These products can be computed in parallel using n^3 processors in time $O(\log n)$ (the same time as each one of them). One possible algorithm is formalized below.

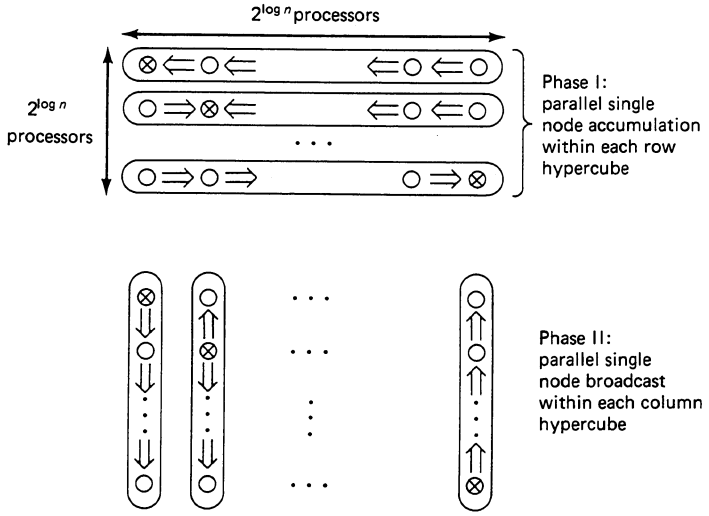


Figure 1.3.26 An $O(\log n)$ algorithm for multiplying an $n \times n$ matrix A with a vector $x \in \Re^n$ on a hypercube with n^2 processors, where n is a power of 2. We use the mapping of the $n \times n$ mesh into the hypercube under which each row and each column of the mesh is a hypercube with n nodes (cf. the construction described in Subsection 1.3.4 and illustrated in Fig. 1.3.12). Initially, processor (i, j) stores the ijth element of A and the jth coordinate of x, and at the end of the algorithm, it stores the jth coordinate of the product Ax. The algorithm consists of two phases, with each phase requiring $O(\log n)$ time. The first phase consists of a single node accumulation within each row, whereby the ith diagonal processor (i.e., the ith processor in the ith row) calculates the ith coordinate of Ax. The second phase consists of a single node broadcast within each column hypercube. Each processor in the jth column receives the jth coordinate of Ax accumulated in the first phase at the jth diagonal processor.

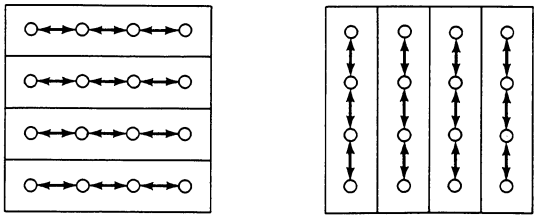


Figure 1.3.27 An $O(n)$ algorithm for multiplying two $n \times n$ matrices A and B with elements a_{ij} and b_{ij}, respectively, on an $n \times n$ mesh. Initially, processor (i, j) of the mesh holds elements a_{ij} and b_{ij}, and at the end of the algorithm, processor (i, j) will hold the ijth element $\sum_{m=1}^{n} a_{im} b_{mj}$ of the product AB.

The algorithm consists of three phases, each requiring $O(n)$ time. In the first phase, each processor (i, j) broadcasts a_{ij} to the processors in the ith row; these are n multinode broadcasts, one within each row, requiring $O(n)$ time. In the second phase, each processor (i, j) broadcasts b_{ij} to the processors in the jth column; these are n multinode broadcasts, one within each column, requiring $O(n)$ time. At the end of two phases, each processor (i, j) holds the values a_{im} and b_{mj} for $m = 1, 2, \ldots, n$, and can form the ijth element $\sum_{m=1}^{n} a_{im} b_{mj}$ of the product AB in time $O(n)$ (which is the third phase). Note that phases 1 and 2 can be done in parallel, assuming that simultaneous communication along all incident links is possible. Also, by appropriately interleaving additions, multiplications, and communications, the algorithm can be made more economical in terms of time and storage.

We assume that n is a power of 2, and that we have a hypercube of n^3 processors arranged in an $n \times n \times n$ array. Let a_{ij} and b_{ij} be the ijth elements of A and B, respectively, and let $c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$ be the ikth element of the product AB. We assume that each processor (i, j, k) initially holds the elements a_{ij} and b_{jk}, and we require that at the end of the algorithm, processor (i, j, k) holds the elements c_{ij} and c_{jk} of the product. Figure 1.3.28 gives a three–phase algorithm that performs the matrix multiplication in $O(\log n)$ time. A similar algorithm works in the same order of time for different initial and final storage specifications, taking advantage of the possibility

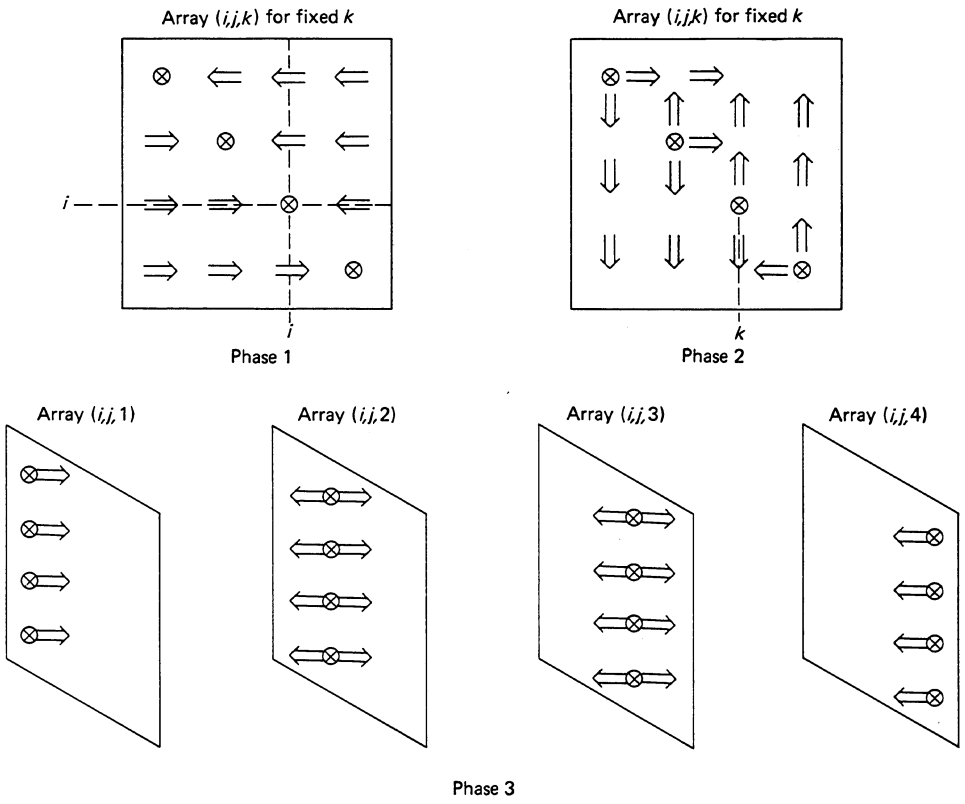


Figure 1.3.28 An $O(\log n)$ algorithm for multiplying two $n \times n$ matrices A and B with elements a_{ij} and b_{ij}, respectively, on an $n \times n \times n$ hypercube array. We assume that each processor (i, j, k) initially holds the elements a_{ij} and b_{jk}, and we require that at the end of the algorithm, processor (i, j, k) holds the ijth and jkth elements of the product. The algorithm consists of three phases, each requiring time $O(\log n)$. In the first phase, the ikth element of the product $c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$ is accumulated at processor (i, i, k) along the array (i, j, k), $j = 1, \ldots, n$. In the second phase, each processor (i, i, k) performs a single node broadcast of c_{ik} along the array (j, i, k), $j = 1, \ldots, n$, and also sends c_{ik} to processor (i, k, k). In the third phase, each processor (i, j, j) performs a single node broadcast of c_{ij} along the array (i, j, k), $k = 1, \ldots, n$.

of doing matrix transposition on a hypercube with n^3 processors in time $O(\log n)$ (see Exercise 3.18).

Consider now the calculation of A^k, where A is an $n \times n$ matrix and k is an integer. We can compute A^k by $O(\log k)$ multiplications of powers of A as discussed earlier. Thus, by using the preceding $O(\log n)$ matrix multiplication algorithm on a hypercube of n^3 processors, we can compute A^k in time $O\big((\log n)(\log k)\big)$. This is the same order of time as the one obtained in Subsection 1.2.3, where all communication was assumed instantaneous.

Table 3.4 summarizes some of the results of this subsection. The conclusion is that by using a suitable interconnection network such as a hypercube, we can perform some of the basic matrix calculations in the same order of time when there are communication delays as when communication is instantaneous. This is encouraging, but it does not imply that the communication penalty is negligible for these calculations; it only implies that the communication time grows with the problem dimension at a rate that is no larger than the rate of growth of the time needed exclusively for computations. For example, the communication time can be larger than the computation time by an arbitrary factor that is constant (independent of n), and still grow at the same rate as the computation time. Alternatively, in some algorithms, we can make the communication time negligible relative to the computation time (see the discussion in Subsection 1.3.5), but this requires a reduction of the number of processors used, and affects the attainable speedup in a different way.

TABLE 3.4 Upper bounds on the optimal times for matrix–vector calculations in $\Re^n$, and the corresponding interconnection networks with which these bounds can be achieved. These bounds are the same as those obtained for the same number of processors in Subsection 1.2.3, where communication was assumed instantaneous.

Problem	Time	Corresponding Topology
Inner Product	$O(\log n)$	Hypercube w/ $p = n$ processors
Matrix–Vector Multiplication	$O(n)$	Linear Array w/ $p = n$ processors
Matrix–Vector Multiplication	$O(\log n)$	Hypercube w/ $p = n^2$ processors
Matrix–Matrix Multiplication	$O(n)$	Mesh w/ $p = n^2$ processors
kth Power of a Matrix	$O(n \log k)$	Mesh w/ $p = n^2$ processors
Matrix–Matrix Multiplication	$O(\log n)$	Hypercube w/ $p = n^3$ processors
kth Power of a Matrix	$O\big((\log n)(\log k)\big)$	Hypercube w/ $p = n^3$ processors

EXERCISES

3.1. Use the following model to establish the validity of the stop–and–wait protocol between a transmitter A and a receiver B as outlined in Subsection 1.3.3 (with packets and acknowledgments numbered modulo 2). The rule for A is that if packet n [numbered n(mod 2)] is the last packet transmitted by A at a given time, and if A started transmission of that packet at time t, then A will start retransmission of packet n at time $t + \Delta$ if no packet

from B numbered n(mod 2) is received correctly in the interval $(t, t + \Delta)$, and it will start transmission of packet $n + 1$ upon receiving a packet from B numbered n(mod 2) (Δ here is some positive number denoting the timeout interval for A). The rule for B is that upon correct reception of a packet numbered k ($k = 0, 1$) it sends a packet numbered k to A. Furthermore, B stores in memory only the first correctly received packet numbered k ($k = 0, 1$) from every sequence of consecutive packets that are all numbered k. Assume that error detection is infallible, and that each packet is transmitted correctly after a finite time of tries. Assume also that at time $t = 0$, A starts sending packet 0, and there are no packets in transit in the communication channel between A and B in either direction. Show that the algorithm works correctly in the sense that all packets $0, 1, 2, \ldots$ sent by A are stored by B in the order sent, without errors, and only once. *Hint:* Let k_A (k_B) be the number (mod 2) of the last correctly received packet by A (B, respectively). Initially, $k_A = 1$, $k_B = 1$. Model how (k_A, k_B) changes in response to packet receptions.

3.2. **(a)** [Top85] Consider a tree with p nodes. Show that an optimal multinode broadcast algorithm takes $p - 1$ time units. *Hint:* Use the following algorithm: at every time unit each processor i considers each of its incident links (i, j). If i has received a packet that it has neither sent already to j nor it has yet received from j, then i sends such a packet on link (i, j). If i does not have such a packet, it sends nothing on (i, j).

(b) Show that the time taken by an optimal total exchange algorithm on a binary balanced tree is $\Theta(p^2)$. *Hint:* Count the number of packets that must go through the incident links of the root node and use part (a). For an upper bound, use part (a) or use the mapping of Fig. 1.3.29.

3.3. **[SaS88]** Let A and B be two adjacent nodes of a hypercube, and let S_A and S_B be the sets of nodes adjacent to A and B, respectively. Show that for every node $i \in S_A$, there exists a unique node $j \in S_B$ such that i is adjacent to j.

3.4. **[SaS88]**

(a) Show that every cycle of a hypercube has an even number of nodes. *Hint:* Count the number of bit reversals as the cycle is traversed.

(b) Show that a ring with an even number of nodes p ($4 \leq p \leq 2^d$) can be mapped into a d–cube, and that a ring with an odd number of nodes cannot be mapped into a hypercube. *Hint:* If p is even, imbed a ring with p nodes into a $2 \times 2^{d-1}$ mesh.

3.5. Number appropriately the nodes of the 4–cube in Fig. 1.3.13(b) in order to demonstrate that a 4×4 mesh with wraparound can be mapped into the 4–cube.

3.6. **(Total Exchange on a Mesh.)** Show inductively that an optimal total exchange algorithm takes $O(p^{(d+1)/d})$ time on a d–dimensional symmetric mesh with p processors by

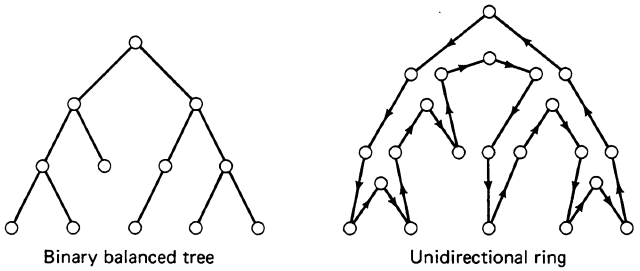


Figure 1.3.29 Mapping a unidirectional ring into an undirected binary balanced tree.

showing that given a total exchange algorithm that takes $O\left(p^{d/(d-1)}\right)$ time on a $(d-1)$–dimensional symmetric mesh with p processors, there is a total exchange algorithm that takes $O\left(p^{(d+1)/d}\right)$ time on a d–dimensional symmetric mesh with p processors. *Hint:* Let the processors of the d–dimensional mesh be numbered $(x_1, x_2, \ldots, x_d)$, where $x_i = 1, \ldots, p^{1/d}$. In the first phase of the algorithm, perform in parallel $p^{1/d}$ total exchanges within each of the $p^{1/d}$ symmetric $(d-1)$–dimensional meshes obtained by fixing the value of x_1. In the second phase, perform in parallel $p^{(d-1)/d}$ total exchanges within each of the $p^{(d-1)/d}$ linear arrays obtained by fixing $x_2, x_3, \ldots, x_d$. Show that each phase takes $O(p^{(d+1)/d})$ time.

3.7. (Multinode Broadcast for a Mesh with Wraparound, [Ozv87] and [Tse87].) Consider an $n \times n$ mesh with wraparound, and assume that each packet transmission takes one time unit. Show that an optimal multinode broadcast algorithm takes $(n^2-1)/4$ time units if n is odd, and $n^2/4$ time units if n is even. *Hint:* For n odd, consider the spanning tree shown in Fig. 1.3.30 for broadcasting the packet of the middle node. For n even, consider first the spanning tree of Fig. 1.3.30 for the upper–left $(n-1) \times (n-1)$ portion of the mesh.

3.8. (Two–Node Broadcast.) Show that a two–node broadcast (a simultaneous single node broadcast from two distinct root nodes) can be performed on the d–cube in d time units. *Hint:* Split the d–cube into two halves each containing one of the root nodes.

3.9. (Single Node Scatter Algorithms [Tse87].) Consider an interconnection network G with p processors, a spanning tree T of G, and the problem of single node scatter from a node s of G.

(a) Assume that transmission along at most one incident link of a processor is allowed. Show that a single node scatter from s takes $p-1$ time units using an optimal algorithm. *Hint:* Consider sending continuously packets from s along the spanning tree T, giving priority to the packets destined for nodes that are furthest away from s (break ties arbitrarily).

(b) Assume that transmission along all the incident links of a processor is allowed. Let r be the number of neighbor nodes of s in the spanning tree T. Let T_i be the subtree of nodes that are connected with s via a simple walk that lies in T, and passes through the ith neighbor of s. Let N_i be the number of nodes in T_i. Construct an algorithm for single node scatter from s that uses links of T and takes $\max\{N_1, N_2, \ldots, N_r\}$ time units. *Hint:* Consider the following rule for s to send packets in each subtree: continuously send packets to distinct nodes in the subtree, giving priority to nodes furthest away from s (break ties arbitrarily).

(c) Assume that transmission along all the incident links of a processor is allowed. Construct a spanning tree T for the d–cube such that the time $\max\{N_1, N_2, \ldots, N_r\}$ for the single node scatter is equal to the optimal time $\lceil (2^d-1)/d \rceil$. *Hint:* Modify the construction of Fig. 1.3.18. Define R_{kn}, $n(t)$, and $m(t)$ as in Fig. 1.3.18, but choose the first element in each equivalence class R_{kn}, so that all nodes t with the same value of $m(t)$ belong to the same subtree. Do this by choosing the first element t in each R_{kn}, so that $m(t') = m(t)$, where t' is t with some unity bit of t changed to a zero, and the equivalence class of t' has d elements.

3.10. (Total Exchange on the Hypercube [Tse87].) Consider the total exchange problem in an interconnection network $G = (N, A)$, where transmission along at most one of the incident links of a node is allowed.

(a) Define the *distance* from a node i to a node j to be the minimum number of arcs contained in walks that start at i and end at j. Suppose that G has a symmetry property, whereby the quantity

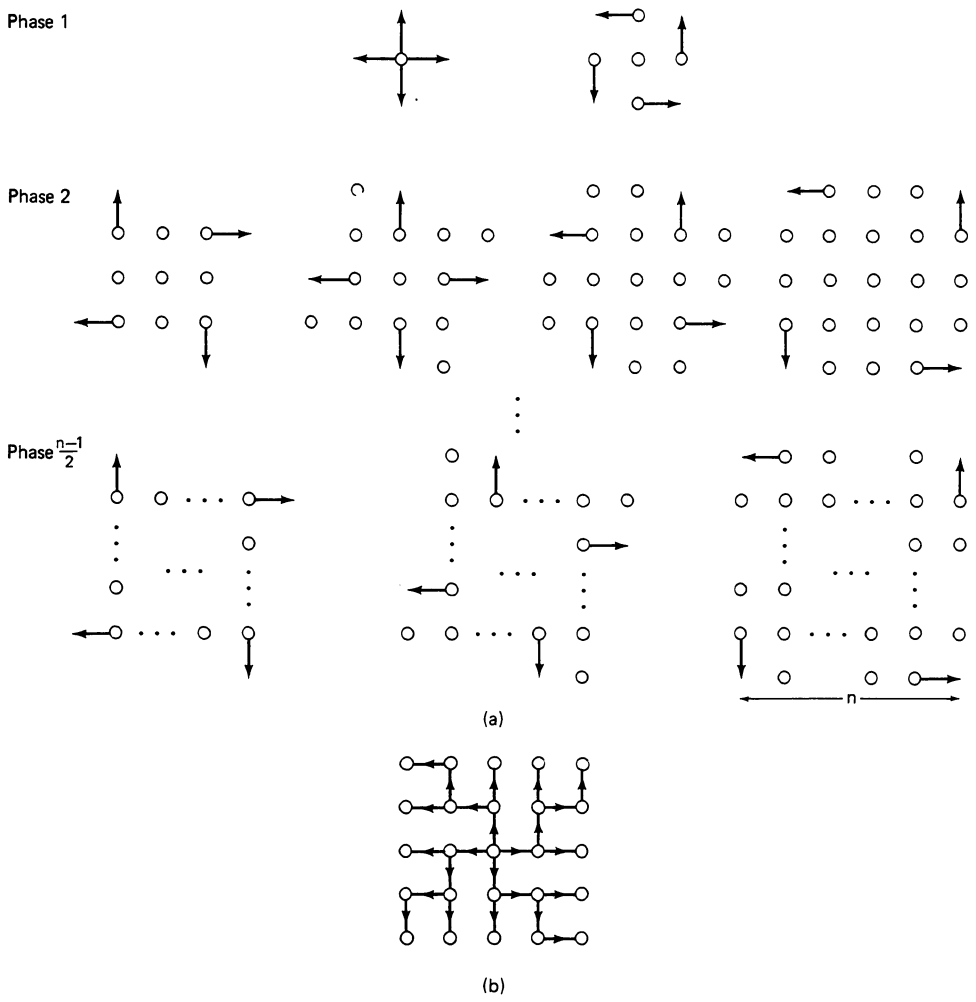


Figure 1.3.30 (a) Procedure to generate a spanning tree rooted at node $\big((n+1)/2, (n+1)/2\big)$ for a multinode broadcast algorithm in an $n \times n$ mesh with wraparound, where n is odd (cf. Exercise 3.7). The tree is generated by labeling nodes starting from the root node $\big((n+1)/2, (n+1)/2\big)$. The labeling procedure consists of $(n-1)/2$ phases. Phase k starts with a $(2k-1) \times (2k-1)$ mesh of labeled nodes and ends with a $(2k+1) \times (2k+1)$ mesh of labeled nodes. (b) Final spanning tree for a 5×5 mesh.

$$D(G) = \sum_{j \in N} (\text{Distance from } i \text{ to } j)$$

is the same for all nodes i. (Examples of networks that have this property are the hypercube and the mesh with wraparound.) Show that $D(G)$ is a lower bound for the time taken by any total exchange algorithm. *Hint:* In a total exchange, every node i must send a packet to every other node j. The number of packet transmissions before

j receives the packet is no less than the distance from i to j. Therefore, the total number of packet transmissions is at least

$$\sum_{i \in N} \sum_{j \in N} (\text{Distance from } i \text{ to } j) = pD(G),$$

where p is the number of processors. Since at most one packet transmission per processor is allowed at a time, the number of simultaneous packet transmissions can be at most p, thereby establishing the lower bound of $D(G)$ time units.

(b) For a d–cube show that $D(G) = d2^{d-1}$. *Hint:* There are exactly $\binom{d}{k} = d!/\big(k!\,(d-k)!\big)$ nodes that are at distance k from a given node, so

$$D(G) = \sum_{k=1}^{d} k \binom{d}{k} = d2^{d-1}.$$

(c) Modify the total exchange algorithm for the d–cube of Fig. 1.3.19 so that phases 1 and 2 are carried out sequentially rather than in parallel, and show inductively that it attains the lower bound of $d2^{d-1}$ time units.

(d) Verify that Fig. 1.3.31 correctly interprets the hypercube total exchange algorithms of Fig. 1.3.19 and (c) above as matrix transposition algorithms. Consider both the case where it is possible to use simultaneously all incident links of a node and the case where this is not possible. For $d = 4$, specify the time that the packet of each of the 16 processors reaches each of the other processors.

3.11. (Broadcast Algorithms on the Hypercube [Tse87].) Show that an optimal single node broadcast algorithm and an optimal multinode broadcast algorithm on the d–cube, under the restriction that each node can transmit at most one packet and simultaneously receive at most one packet at a time, take d and $(2^d - 1)$ time units, respectively. *Hint:* For a single node broadcast, d is a lower bound for the optimal time, since for every node, there is another node at distance d from it. Also, since in a multinode broadcast, each node receives $(2^d - 1)$ packets from the other nodes, $(2^d - 1)$ is a lower bound on the optimal time. To show that there are algorithms attaining these bounds, argue inductively. For $d = 1$, the obvious algorithms work. Assume that there are single node and multinode broadcast algorithms for the d–cube that take d and $(2^d - 1)$ time units, respectively. Consider a decomposition of the $(d+1)$–cube into two d–cubes. For a single node broadcast, send first the packet of the root node s to its counterpart s' in the opposite d–cube; then, perform a single node broadcast in parallel from s and s' within the corresponding d–cubes requiring d time units (by the induction hypothesis) for a total of $(d+1)$ time units. For a multinode broadcast, use the imbedding of a ring in a hypercube.

3.12. (Optimal Generalized Vector Shift Algorithm [Ozv87].) Show that there exists a generalized vector shift algorithm on the d–cube that takes no more than d time units. *Hint:* For $k = 1, 2, \ldots, d$, consider the set S_k of all integers $x(k)$ that can be generated by the recursion

$$x(t+1) = 2x(t) + u(t), \qquad \forall\, t = 0, 1, \ldots,$$

where $x(0) = 0$, and $u(t)$ can take the values -1, 0, or 1. Thus, S_k consists of all integers $x(k)$ of the form

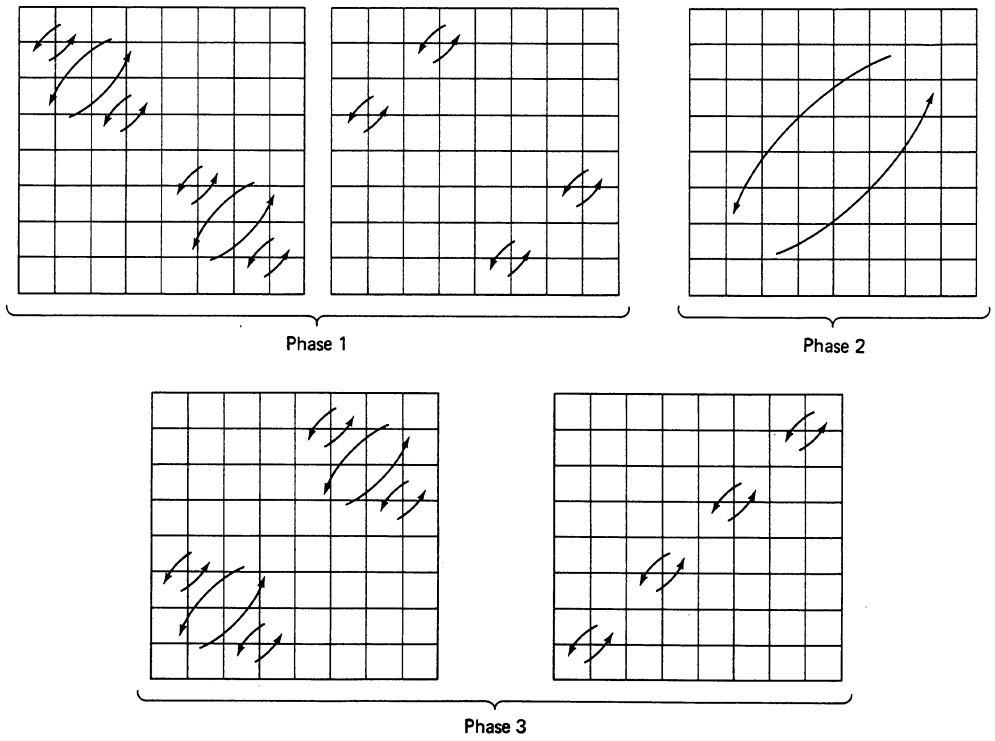


Figure 1.3.31 Interpretation of a total exchange algorithm for the 3–cube (cf. Fig. 1.3.19) as an 8×8 matrix transposition. Long arrows correspond to submatrix moves. The ith processor initially holds the ith row of the matrix, and at the end, it holds the ith column of the matrix. Assuming transmission along all the incident links of a node is allowed, phases 1 and 3 take 3 time units, phase 2 takes 4 time units, whereas phases 1 and 2 can be done in parallel.

$$x(k) = u(k-1) + 2u(k-2) + 2^2u(k-3) + \cdots + 2^{k-1}u(0),$$

where $u(t)$ can take the values -1, 0, or 1. Show that S_k consists of all the integers in the interval $[-(2^k - 1), (2^k - 1)]$.

3.13. (The Butterfly.) The butterfly network consists of $(d+1)2^d$ processors, where d is some integer. The processors are arranged in $d+1$ rows and 2^d columns. The links are specified as shown in Fig. 1.3.32.

(a) Show that if the processors of the top row and the corresponding incident links are removed, we obtain two identical butterflies of $d2^{d-1}$ processors.

(b) Show that an optimal multinode broadcast algorithm on the butterfly takes $\Theta(d2^d)$ time.

3.14. (The Cube–Connected Cycles.) The cube–connected cycles network has $d2^d$ processors, where d is some integer. It is obtained from the d–cube by replacing each processor with a cycle of d processors, as illustrated in Fig. 1.3.33. In particular, each processor has an identity (i, j), where j is a d–bit binary string which is the corresponding d–cube processor identity, and i is an integer from 1 to d. There is a link between two processors with

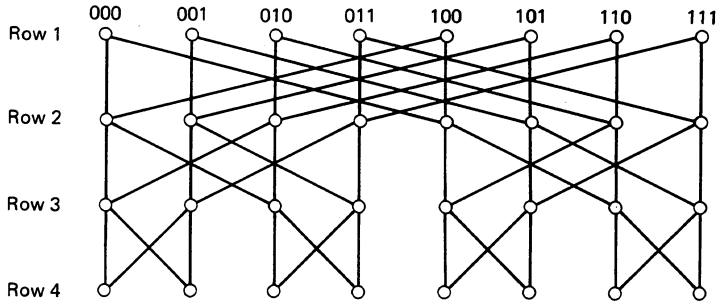


Figure 1.3.32 Illustration of the butterfly network consisting of $(d+1)2^d$ processors. The processors are arranged in $d+1$ rows and 2^d columns, where d is some integer. We label the rows consecutively as $1, 2, \ldots, d+1$, and we label the columns consecutively from 0 to $2^d - 1$ using the d–bit binary representation. Each processor n of row $i = 1, 2, \ldots, d$ is connected with two processors of row $i+1$: the processor below it (same column), and the processor of the column whose label differs from that of the column of n in the ith bit from the left. The figure illustrates the case where $d = 3$.

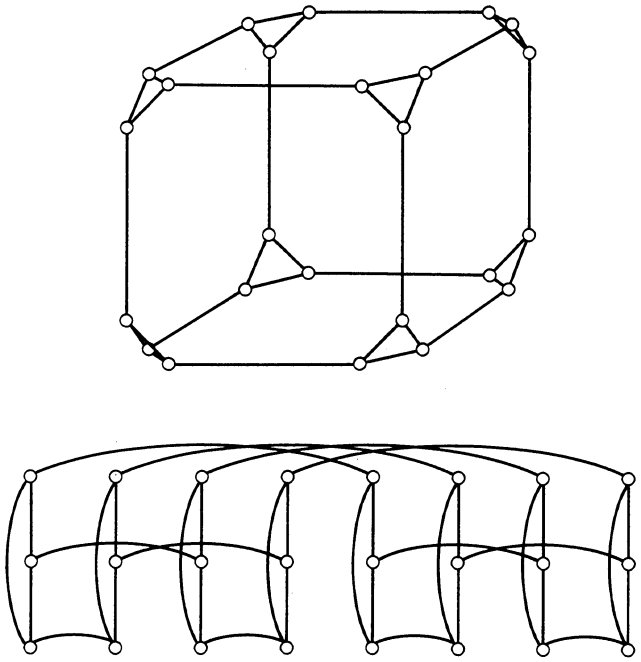


Figure 1.3.33 Two views of the cube–connected cycles network for $d = 3$.

identities (i, j) and (k, m) if and only if a) $i = k$ and j differs from m in the ith bit from the left or b) $j = m$ and $|i-k| = 1$ or $|i-k| = d-1$. Derive the order of time taken by an optimal single node and multinode broadcast algorithm, and by an optimal total exchange algorithm.

3.15. **[Ozv87]** Modify the procedure given for hypercubes for generating multinode broadcast algorithms from single node broadcast algorithms so that it works for a ring and for a mesh

with wraparound. *Hint:* Define the sets A_i as for a hypercube, and generate the sets $A_i(t)$ by using an appropriate operation in place of $\oplus$.

3.16. (Matrix Multiplication.) Given a square mesh of n^2 processors, show that multiplication of an $n \times n$ matrix A with the transpose of an $n \times n$ matrix B can be done in $O(n)$ time. We assume that the (i, j)th elements of A and B are stored in processor (i, j), and the (i, j)th element of the product AB' must be eventually stored in the same processor. *Hint:* Show that transposition of the matrix B can bo done in $O(n)$ time.

3.17. (Matrix Transposition on the Hypercube [Tse87].) Show that transposition of an $n \times n$ matrix can be done in $2 \log n$ time units on a hypercube of n^2 processors arranged as an $n \times n$ array (n is a power of 2 and each packet requires unit transmission time). We assume that the (i, j)th processor holds initially the (i, j)th element of the matrix, and must hold at the end of the algorithm the (j, i)th element of the matrix. *Hint:* Use the algorithm of Fig. 1.3.34.

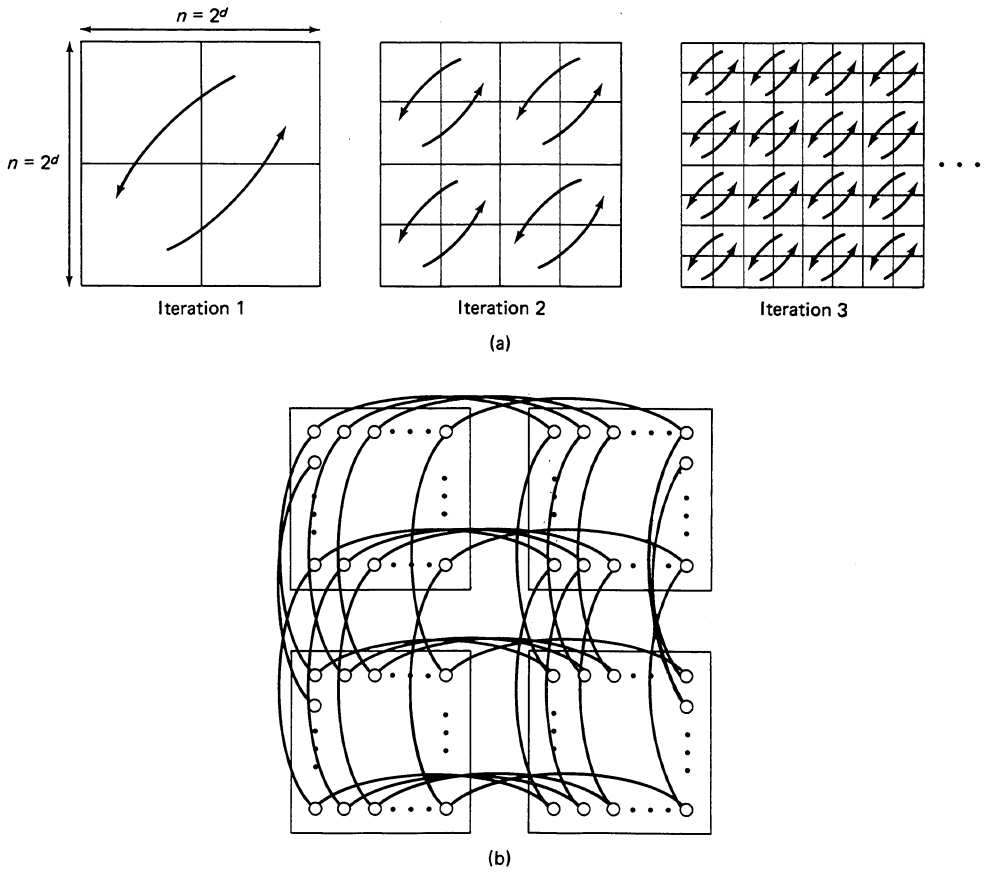


Figure 1.3.34 An algorithm for transposing an $n \times n$ matrix on an $n \times n$ hypercube array. Each of the iterations of (a) takes 2 time units using the links illustrated in (b).

3.18. (Matrix Transposition on the Hypercube.) This exercise considers algorithms for transposing 2–dimensional array data on a hypercube arranged as a 3–dimensional mesh array. Assume that n is a power of 2, and consider the hypercube with n^3 processors arranged in an $n \times n \times n$ array. Suppose that each processor (i, j, k) $(k = 1, \ldots, n)$ initially stores the (i, j)th element a_{ij} of a matrix A.

(a) Show how the algorithm of Exercise 3.17 can be used to move a_{ij} to the processors (j, i, k) for $k = 1, \ldots, n$.

(b) Construct an $O(\log n)$ time algorithm that for every (i, j), moves a_{ij} to the processors (i, k, j) for $k = 1, \ldots, n$.

3.19. (Pipelining of Computation and Communication in Single Node Broadcast.) Consider the problem of a single node broadcast of a packet in an interconnection network of n processors such that the maximum distance from every node to the root node is r. The packet can be divided in m packets, each requiring $(w + 1/m)$ time units for transmission on every link, where w represents the extra transmission time required for overhead. Show that it is possible to perform the single node broadcast in $(w + 1/m)(r + m - 1)$ time units, and that if $w > r - 1$, the optimal value of m is 1. Characterize the optimal value of m when $w \leq r - 1$.

3.20. Consider an iteration of the form

$$x := C'Cx + b,$$

where $x \in \Re^n$, $b \in \Re^n$, and C is an $m \times n$ matrix that has no special sparsity structure. This problem deals with parallelization of the iteration using n processors. Assume that each arithmetic operation takes one time unit and that each packet transmission takes d time units.

(a) Assume that each processor i knows the entries of the ith column of C, and the ith coordinates x_i and b_i. Estimate the time needed per iteration for the two cases where the n processsors communicate via a hypercube and via a linear array.

(b) Repeat part (a) for the case where the matrix product $C'C$ is given at the beginning of the algorithm, and processor i knows the entries of the ith row of $C'C$, the vector x, and the ith coordinate b_i.

(c) Which implementation is better? (The answer may depend on the value of m.)

3.21. (Sparse Matrix–Vector Multiplication.) Consider forming the matrix–vector product Ax in $\Re^n$ using a hypercube of n processors. Initially, each processor stores the vector x and the ith row of A, and at the end, it must hold the product Ax. Assume that each arithmetic operation and each packet transmission take unit time. Show that if each row of A has no more than r nonzero elements, there is an algorithm that takes time $O\big(\max(n/\log n, r)\big)$. Provide a similar result for the case where each processor stores a column of A and the corresponding entry of x, and each column has no more than r nonzero elements.

3.22. (Addition of p Scalars on a Hypercube [DNS81].) Consider a hypercube with p processors, and suppose that each processor i holds a scalar a_i. We wish to calculate and store the sum $\sum_{i=1}^{p} a_i$ in each processor. Assume that transmission of a scalar over any link requires one time unit, and an addition requires negligible time. Consider the following algorithm: each processor i updates a scalar s_i at the end of each of $\log p$ stages. Initially, $s_i = a_i$. During the kth stage, each processor i transmits its current value s_i to the processor j whose identity number agrees with the one of i except for the kth bit from the left; processor j then adds the received value s_i to its current value s_j, and stores the result in place of s_j. Show that

after $\log p$ time units, each processor holds the required sum. (This is faster by a factor of 2 over collecting the sum at a single processor along a spanning tree and broadcasting it back to all processors.)

3.23. (Repeated Matrix–Vector Multiplication on a Hypercube [Tse88].) Suppose that we have a hypercube with n^2 processors. Processor (i, j) stores the ijth element c_{ij} of an $n \times n$ matrix C and the jth coordinate x_j of a vector x.

(a) Use the addition method of Exercise 3.22 in an algorithm that computes the jth coordinate of $C'Cx$ in $2 \log n$ time units. (Each transmission of a scalar over any link requires one time unit, and each multiplication and addition requires negligible time.)

(b) Show that if C is symmetric, then the products $C^k x, k = 1, 2, \cdots, r$, where r is a positive integer, can be found in $r \log n$ time units.

3.24. (Multinode Broadcast with Packets Combined [KVC88].) Consider the multinode broadcast problem with the following difference: we assume that a node can combine any number k of packets and transmit them on any one link as a single packet in one time unit, rather than transmit them as k separate packets in k time units, as we assume in our standard model. Show that this problem can be solved by using a single node accumulation algorithm followed by a single node broadcast algorithm. In particular, an optimal algorithm for the problem takes at most $2d$ time units on a d–cube.

1.4 SYNCHRONIZATION ISSUES IN PARALLEL AND DISTRIBUTED ALGORITHMS

In any parallel or distributed algorithm, it is necessary to coordinate to some extent the activities of the different processors. This coordination is often implemented by dividing the algorithm in "phases". During each phase, every processor must execute a number of computations that depend on the results of the computations of other processors in previous phases; however, the timing of the computations at any one processor during a phase can be independent of the timing of computations at other processors within the same phase. In effect, within a phase, each processor does not interact with other processors as far as the given algorithm is concerned. All interaction takes place at the end of phases. We call such algorithms *synchronous*, and in this section, we compare them with other algorithms, called *asynchronous*, for which there is no notion of phases and the coordination of the computations of different processors is less strict. Throughout this section we emphasize the case of a message–passing system. Some of the ideas, however, are applicable, with proper interpretation, to shared memory systems.

1.4.1 Synchronous Algorithms

Suppose that a sequence of computations is divided into consecutive segments, called *phases* and numbered $1, 2, 3, \ldots$. The computations within each phase are divided among the n processors of a computing system. During a phase t, each processor i does some computations using the problem data, together with the information that it received from the other processors during the previous phases $1, 2, \ldots, t-1$. Processor i then

sends some information in the form of a message to each processor in a given subset $P(i,t) \subset \{1,2,\ldots,n\}$, and the process is repeated at the next phase $t+1$. (The method by which a message is transmitted is not material in our discussion. In particular, a message can consist of several packets, in which case the time of reception of the message is the time of reception of the last packet.) An implicit assumption here is that the computations of different processors can be carried out "independently" within a phase, and that their relative timing is immaterial. This is necessary for the distributed algorithm to replicate the results of the original serial algorithm at the end of each phase. Note that in some cases, a processor may not know from which processors to expect a message during a phase, that is, a processor j may not know the set $\{i \mid j \in P(i,t)\}$.

As an example, consider the relaxation iteration

$$x_i(t+1) = f_i\big(x_1(t),\ldots,x_n(t)\big), \qquad i = 1,\ldots,n, \tag{4.1}$$

discussed in Subsection 1.2.4, where variable x_i is updated by processor i. A corresponding distributed algorithm can be implemented by associating phases with the time instants t. The requirement here is that each processor i updates x_i using the relaxation iteration (4.1) and then sends a message with the updated value to all processors j for which x_i appears explicitly in the function f_j. Thus, the subset of processors $P(i,t)$ receiving a message from i during phase t is the set of all j for which (i,j) is an arc in the dependency graph discussed in Subsection 1.2.4.

Consider also the case where the relaxation iteration (4.1) is implemented in a shared memory machine. Here a processor sends a message to all processors simultaneously by writing it in the shared memory. Suppose that a processor will start a computation of a new phase only after all the messages of the previous phase have been written in the shared memory. Then, we have a special case of the preceding model, where

$$P(i,t) = \{1,2,\ldots,n\},$$

and the reception of a message is simultaneous at all processors.

A distributed algorithm such as the one described above is said to be *synchronous*. It is mathematically equivalent to an algorithm governed by a global clock, that is, one for which the start of each phase is simultaneous for all processors, and the end of the message receptions is simultaneous for all messages. In order to implement a synchronous algorithm in an inherently asynchronous distributed system, we need a *synchronization mechanism*, i.e., an algorithm that is superimposed on the original and by which every processor can detect the end of each phase. Such an algorithm is called a *synchronizer*. In what follows in this section, we describe two approaches on which synchronizers are based, called *global synchronization* and *local synchronization*. A third approach, based on the idea of *rollback*, is discussed in Section 8.4, together with its application in simulation problems.

Global Synchronization

The idea here is to let each processor detect when all messages sent during a phase have been received, and only then to start the computation of the next phase. The conceptually simplest way for effecting global synchronization is through the use of *timeouts*. We assume that there is no global clock accessible by all processors, but instead, each processor can measure accurately the length of any time interval using a local clock. Suppose that there is a known upper bound T_p for the time required for each processor i to execute the computations of a phase, and for the associated messages sent by i to be received at their destinations. Suppose also that there is an (unknown) time interval of known length T_f during which all processors started the first phase. Then synchronization will be effected if each processor i starts the kth phase $k(T_p + T_f)$ time units after it started the first phase. Fig. 1.4.1 illustrates this process. The difficulty with this method is that the bounds T_p and T_f may be too conservative or unavailable.

Another approach is for every processor to send a *phase termination message* to every other processor once it knows that all of its own messages for a given phase have been received. We assume here that each message sent is acknowledged through a return message sent by the receiver to the transmitter, so each processor knows eventually that all the messages it sent during a phase have been received, at which time it can issue a phase termination message. Once a processor has sent its own phase termination message and has received the corresponding phase termination messages from all other processors, it can proceed with the computations of the next phase. In a shared memory system, this method is conceptually straightforward through the use of special variables that are accessible to all processors. There are a number of possible implementations (spin locks, semaphores, monitors; see [AHV85] and [Qui87]), which we will not discuss in detail. The basic conceptual idea is that the special variables should take at the right times particular values that indicate to the processors that a phase has ended, and it is therefore safe to proceed with the computations of the next phase.

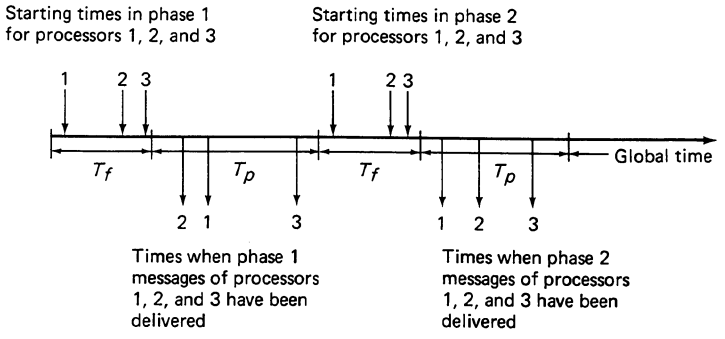


Figure 1.4.1 Implementation of global synchronization using timeouts. The starting times of the three processors for each phase are known to be within an interval of length T_f. For every processor, the time required for computation and message delivery within a phase is known to be no more than T_p. It is sufficient for each processor to start a new phase every $(T_f + T_p)$ time units.

In a message–passing system, global synchronization can be implemented by using a spanning tree and a special node designated as the root of the tree. The phase termination messages of the processors are collected at the root of the tree, starting from the leaves. Once the root has received the phase termination messages of all processors, it can send a phase initiation message to all other processors along the spanning tree, and each processor can begin a new phase upon reception of this message (see Fig. 1.4.2). It can be seen that this method essentially requires a single node accumulation followed by a single node broadcast (cf. Subsection 1.3.4). Therefore, the communication time for phase termination (subsequent to the time when acknowledgments for all processor messages in the phase have been received) is $\Theta(r)$, where r is the diameter of the network.

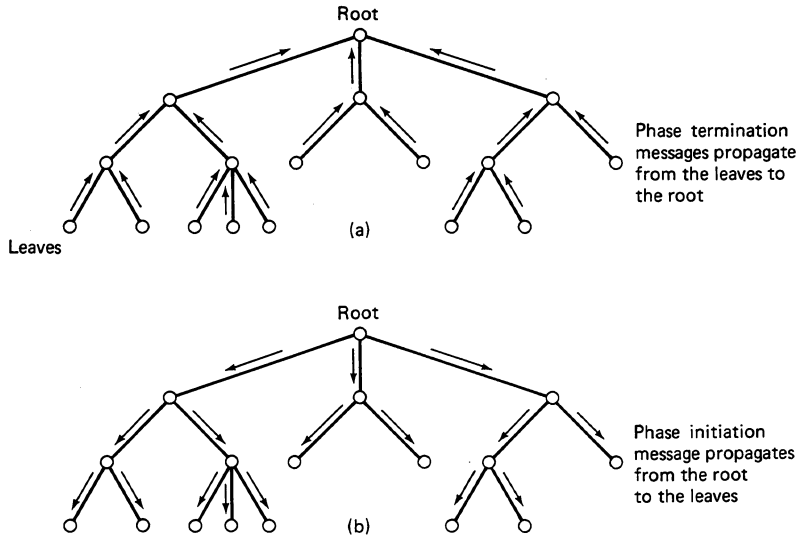


Figure 1.4.2 Global synchronization using a spanning tree, with a special node designated as the root of the tree. (a) Each node sends a phase termination message to its parent, when it has received acknowledgments for all the messages it sent during a phase, as well as phase termination messages from all its children. (b) The root, upon receiving a phase termination message from all its children, sends a phase initiation message to its children, who relay it to their children, and so on. Each node, upon receiving the phase initiation message, can start a new phase.

Local Synchronization

The main idea of this method is that if a processor knows which messages to expect in each phase, then it can start a new phase once it has received all these messages. In particular, processor j can start the computation of a new phase $t+1$ once it has received the messages of the previous phase t from all processors i in the set

$$\{i \mid j \in P(i,t)\}. \tag{4.2}$$

It is not necessary for a processor to know whether any other messages sent during phase t (including its own) have been received, so there is no need to waste time waiting for these receptions to be completed and to be confirmed. When processor j does not know the set (4.2) but instead knows a set S_j, where

$$S_j \supset \{i \mid j \in P(i,t)\}, \qquad \forall\, t,$$

the scheme can be modified so that all nodes $i \in S_j$ with $j \notin P(i,t)$ are required to send a "dummy" message to j during phase t. In this way, the situation is reduced to the case where the set (4.2) is known by j. Note, however, that with this modification, the local synchronization scheme may become undesirable if S_j contains many more elements than the set $\{i \mid j \in P(i,t)\}$.

When each processor j knows the set (4.2), it can be seen that the local synchronization method leads to no more communication penalty than any global synchronization method. Often, the communication penalty is considerably less. This is particularly so when the transmission time of messages on communication links is comparable to the time needed for computation in each phase, since, for global synchronization, additional messages may be required for acknowledgments, etc., as discussed earlier. Even without counting the time for acknowledgments, the difference between the execution times associated with the global and the local methods typically increases with the number of phases, due to the variability of the times for computation and message delivery by different processors within each phase. We demonstrate this phenomenon in the case where the times required by a processor to complete the delivery of messages associated with a phase are deterministic. For a related analysis when these times are random, independent, and exponentially distributed, see Exercise 4.1.

Let $T_{ij}(t)$ be the time required for processor i to do the computations of phase t and to deliver the corresponding message to processor $j \in P(i,t)$. To simplify notation, we assume that $i \in P(i,t)$, and we denote by $T_{ii}(t)$ the time required for i to do just the computations for phase t. Suppose that $P(i,t)$ and $T_{ij}(t)$ are known and independent of t [i.e., $P(i,t) = P_i$ and $T_{ij}(t) = T_{ij}$, for all t, i, and $j \in P_i$]. We will compare the local synchronization method with a global synchronization method, whereby a phase is considered completed as soon as every message of the phase has been delivered. We thus neglect any time needed for acknowledging the messages of the phase or for broadcasting the phase termination messages mentioned earlier. Assume that all processors start phase 1 simultaneously, and consider the times $L(k)$ and $G(k)$ required to complete k phases at all processors using the local and the global synchronization methods, respectively.

The time for a single phase using the global synchronization method is

$$G(1) = T_{max},$$

where

$$T_{max} = \max_{i=1,\ldots,n,\ j \in P_i} T_{ij}.$$

Since each phase must be completed at all processors before a new phase can begin, we have

$$G(k) = kT_{max}. \tag{4.3}$$

To obtain the corresponding time for the local synchronization method, we define

$$C_{max} = \max_{Y} \frac{\sum_{(i,j)\in Y} T_{ij}}{|Y|},$$

where the maximum in the preceding definition is over all sequences Y of the form

$$\{(i_1, i_2), (i_2, i_3), \ldots, (i_m, i_1)\},$$

where $m = |Y| \geq 1$ and $i_{s+1} \in P_{i_s}$ for $s = 1, \ldots, m-1$, $i_1 \in P_{i_m}$. We form a directed graph having as node set

$$N = \big\{(t, i) \mid t = 1, \ldots, k+1,\ i = 1, \ldots, n\big\},$$

(see Fig. 1.4.3). There is an arc corresponding to each pair $\big((t, i), (t+1, j)\big)$ where $t = 1, 2, \ldots, k$, and $j \in P_i$, and we view T_{ij} as the "length" of such an arc. Consider the set of all paths p starting at a node of the form $(1, i)$ and ending at a node of the form $(k+1, j)$. Let M_p be the length of path p, that is, the sum of the lengths of its arcs. Then, as seen from Fig. 1.4.3, $L(k)$ is the maximum of M_p over all paths p of the type just described, and for $k \geq n$, we have

$$L(k) \leq \big[k - (n-1)\big] C_{max} + (n-1) T_{max}. \tag{4.4}$$

Comparing Eqs. (4.3) and (4.4) we see that

$$G(k) - L(k) \geq \big[k - (n-1)\big] (T_{max} - C_{max}).$$

Thus, when

$$T_{max} - C_{max} > 0,$$

and k is large, the difference $G(k) - L(k)$ grows, in effect, proportionally with k. Note, however, that if the time required for message delivery is small relative to the time needed for the processors' computation during a phase, that is, $T_{ii} \simeq T_{ij}$ for all j, then the difference $T_{max} - C_{max}$ will be small or zero. (Take, for example, $n = 2$, $T_{11} = 2$, $T_{22} = 1$, and $T_{12} + T_{21} < 4$, in which case there is no difference in the communication penalty associated with the global and local synchronization methods.)

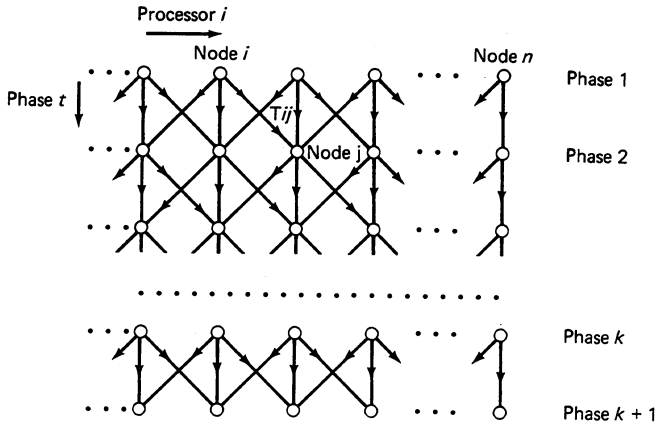


Figure 1.4.3 Estimating the time to complete k phases using the local synchronization method [cf. Eq. (4.4)]. We consider the acyclic graph with node set

$$N = \big\{(t,i) \mid t = 1,\dots,k+1,\ i = 1,\dots,n\big\},$$

and arcs $\big((t,i),(t+1,j)\big)$, where $t = 1,2,\dots,k$, and $j \in P_i$. In the example of the figure we have $P_1 = \{1,2\}$, $P_n = \{n-1,n\}$, and $P_i = \{i-1,i,i+1\}$, for $i \neq 1,n$. We view T_{ij} as the length of such an arc. Since all the messages of the k phases must be received before the k phases can be completed, and each processor cannot begin a new phase without receiving the messages of the previous phase, the length of any path in this graph is less than or equal to $L(k)$. By induction on the number of phases k, it can be verified that $L(k)$ is equal to the length of some path. Therefore, $L(k)$ is equal to the length of a longest path. Let $\{(i_1,1),(i_2,2),\dots,(i_{k+1},k+1)\}$ be a longest path. Assume first that $i_m \neq i_{m+1}$ for all m. Consider the path $\{i_1,i_2,\dots,i_{k+1}\}$ in the graph with set of nodes $\{1,\dots,n\}$ and set of arcs $\{(i,j) \mid j \in P_i,\ j \neq i,\ i = 1,\dots,n\}$. By using the Path Decomposition Theorem of Appendix B on this latter path, it is seen that the longest path can be broken down in two components:

(a) a (possibly empty) collection of subpaths of the form $\{(j_1,t),(j_2,t+1),\dots,(j_m,t+m-1),(j_1,t+m)\}$, where $m > 1$, and $j_{s+1} \in P_{j_s}$ for $s = 1,\dots,m-1$, $j_1 \in P_{j_m}$, corresponding to cycles obtained from the path $\{i_1,i_2,\dots,i_{k+1}\}$;

(b) a set of at most $(n-1)$ additional arcs, corresponding to a simple path obtained from the path $\{i_1,i_2,\dots,i_{k+1}\}$.

By bounding from above the lengths of the two components, we obtain the estimate (4.4). If $i_m = i_{m+1}$ for some m, the length of the arc $\big((i_m,m),(i_{m+1},m+1)\big)$ is bounded above by C_{max}, and it is seen that Eq. (4.4) remains valid even in the presence of such arcs.

The preceding discussion suggests that the local synchronization method is usually superior to the global method in terms of communication penalty. On the other hand, there may be other factors, such as software complexity, that argue in favor of the global method, since this method is largely independent of the structure of the algorithm executed.

1.4.2 Asynchronous Algorithms and the Reduction of the Synchronization Penalty

The communication penalty, and the overall execution time of many algorithms, can often be substantially reduced by means of an asynchronous implementation. The analysis of asynchronous distributed algorithms is one of the focal points of this book, and in this subsection, we provide a preliminary and informal contrast with synchronous algorithms. A precise model of an asynchronous algorithm will be given later, but for the purposes of this section the following rough description will suffice.

Given a distributed algorithm, for each processor, there is a set of times at which the processor executes some computation, some other times at which the processor sends some messages to other processors, and yet some other times at which the processor receives messages from other processors. The algorithm is termed synchronous, in the sense of the preceding subsection, if it is mathematically equivalent to one for which the times of computation, message transmission, and message reception are fixed and given *a priori*. We say that the algorithm is asynchronous if these times (and, therefore, also the order of computations and message receptions at the processors) can vary widely in two different executions of the algorithm with an attendant effect on the results of the computation.

For another contrasting view, which is appropriate primarily for a message–passing system, we can think of a distributed algorithm as a collection of local algorithms. Each local algorithm is executed at a different processor and occasionally uses data generated by other local algorithms. In the simplest case of a synchronous algorithm, the timing of operations at each processor is completely determined and is enforced by using a global clock. A more complex type of synchronous algorithm is one in which the exact timing of operations at each local algorithm is not predetermined, but the local algorithms still have to wait at predetermined points for predetermined data to become available (cf. the discussion of local synchronization in the previous subsection). An example of an asynchronous algorithm is when local algorithms do not wait for predetermined data to become available; they keep on computing, trying to solve the given problem with whatever data happen to be available at the time.

The most extreme type of asynchronous algorithm is one that can tolerate changes in the problem data or in the distributed computing system, without restarting itself to some predetermined initial conditions. This situation arises principally in data networks, where the nodes and the communication links can fail or be repaired as various distributed algorithms that control the network are executed. In some networks, such as mobile packet radio networks, changes in the network topology can be frequent. In other networks, such changes may be infrequent, but the execution time of the algorithm considered may be so long that there is a nonnegligible probability of occurence of a topological change while the algorithm executes. For example, in general purpose data networks, some algorithms, such as the routing algorithm, are essentially always operating, so they must inevitably operate in the face of node and link failures. There are a number of difficulties here. First, one may have to keep all nodes informed of the link and node failures and repairs; this information can have a bearing on the distributed

algorithm being executed. Doing so is not as easy as it may appear since failure information must be communicated over links that are themselves subject to failure; Fig. 1.4.4 provides an example. Second, a link or node failure that occurs while a distributed algorithm is executing on the data network will typically affect the algorithm. To cope with the situation, the algorithm should either be capable to adapt to the failure or it should be aborted and be restarted. Doing the latter may be nontrivial, since more failures can occur while the algorithm is being restarted. Such issues will be discussed in Chapter 8, where it will become evident that it is typically far simpler to restart asynchronous rather than synchronous algorithms, because asynchronous algorithms generally allow more flexibility in the choice of initial conditions.

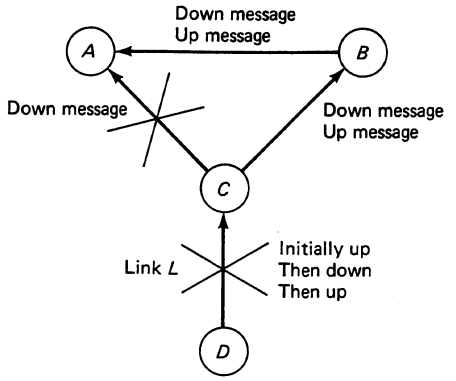


Figure 1.4.4 An example of the difficulties in communicating link failure information in a data network (from [BeG87]). Consider an algorithm that works as follows: whenever the status of a link changes, the end nodes of the link send this information to all their adjacent nodes, which in turn relay this information to their own adjacent nodes (if this information changes their view of the status of the link), etc. Here link L is initially up, then it goes down, and then up again. Suppose the down and up messages on links (C,B) and (B,A) travel faster than the down message travels on link (C,A). Also, suppose that link (C,A) goes down before the up message regarding link L travels on it. Then the last message received by A asserts that link L is down while the link is actually up. The difficulty here is that link failures can cause old information to be perceived as new.

An important question is whether or not asynchronism helps to reduce the communication penalty and the overall solution time of a given algorithm. We discuss this next in the context of the Jacobi and Gauss–Seidel relaxation methods introduced in Subsection 1.2.4.

Asynchronous Relaxation Methods

Consider an n–processor system and an n–dimensional fixed point problem, whereby we want to find a vector $x = (x_1, x_2, \ldots, x_n)$ satisfying

$$x_i = f_i(x_1, x_2, \ldots, x_n), \qquad i = 1, 2, \ldots, n,$$

where f_i are given functions of n variables. Suppose that each processor i updates the variable x_i according to

$$x_i := f_i(x_1, x_2, \ldots, x_n), \qquad i = 1, 2, \ldots, n, \tag{4.5}$$

starting from a set of initial values for all variables. We will discuss in Chapters 2 and 3 several examples of updates of this type, and we will also consider more general types of updates.

A synchronous implementation of the algorithm requires that a processor i does not carry out its kth update without first receiving the results of the $(k-1)$st update from the processors whose variables appear in the function f_i. There is a certain inefficiency built into this requirement, which we call, somewhat loosely, the *synchronization penalty*. It is due to two factors:

(a) A processor i upon updating x_i must remain idle while waiting for messages to arrive from other processors (see Fig. 1.4.5). In particular, a slow communication channel slows the progress of the entire computation, as shown in Fig. 1.4.6(a).

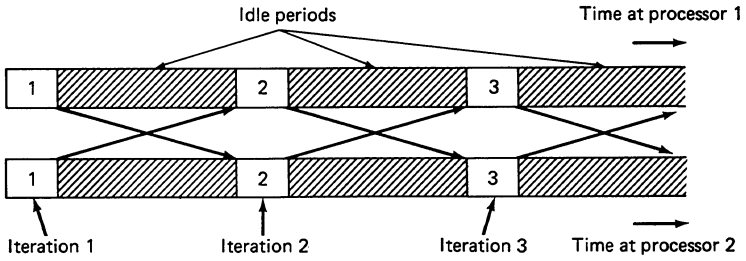


Figure 1.4.5 Illustration of the synchronization penalty due to long communication delays. Arrows indicate the times of message reception. Shaded areas indicate the idle periods when a processor is waiting for a message from the other processor.

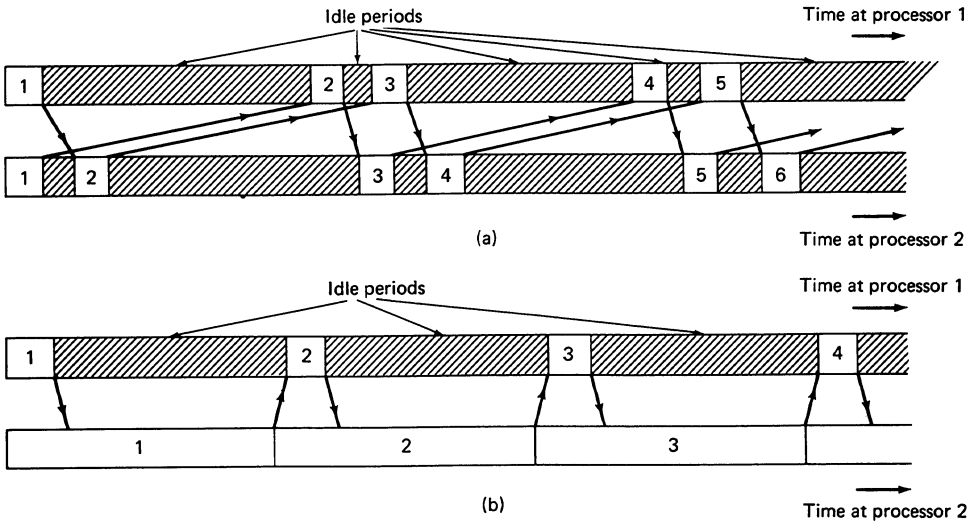


Figure 1.4.6 Illustration of the synchronization penalty due to a single slow communication channel [channel from 2 to 1 in (a)], and due to a single slow processor [second processor in (b)].

(b) Processors that are fast either because of high computing power or because of small workload per iteration, must wait for the slower processors to finish their iteration.

Thus the pace of the algorithm is dictated by the slowest processor, as shown in Fig. 1.4.6(b).

Note that the synchronization penalty contributes to the communication penalty in view of factor (a). It also contributes to a loss of efficiency when the computational load of each iteration is not well balanced among all processors in view of factor (b), and this can happen even if all communication is instantaneous.

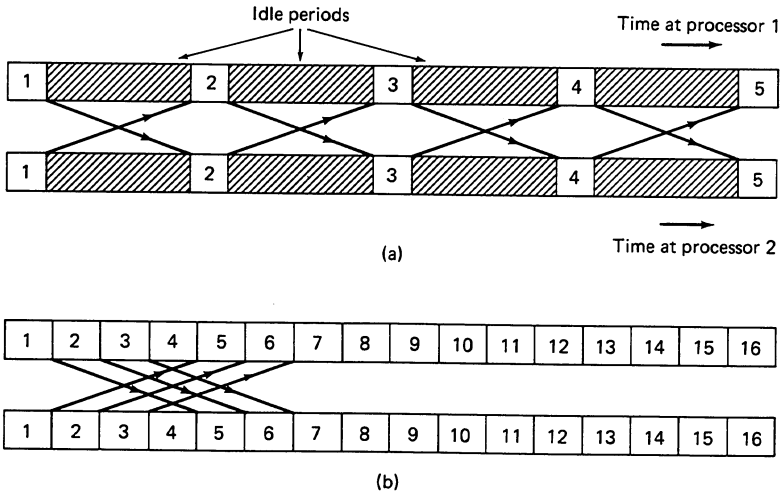


Figure 1.4.7 (a) Timing diagram for a synchronous algorithm. Arrows indicate the times of message reception. Shaded areas indicate the idle periods when the processor is waiting for a message from the other processor. Numbered areas indicate update periods at each processor. Idle periods are three times longer than computation periods in this example. (b) Timing diagram for an asynchronous algorithm. Arrows indicate the times of message reception. Numbered areas indicate update periods at each processor. The potential advantage of an asynchronous algorithm lies in the fact that it can execute more iterations per unit time because there is no waiting for message receptions, (three times more in this example). It is unclear, however, whether the additional iterations will accelerate convergence, since these iterations are based on out–of–date information.

In an asynchronous version of the preceding algorithm, there is much more flexibility regarding the use of the information received from other processors. What is required is that the kth update at processor i is carried out with knowledge of the results of *some* past update of every other processor, not necessarily the most recent update. Thus, for example, processor i may be executing its kth update using the results of the $(k+10)$th update of some other processor j, while at the same time processor j may be executing its $(k+100)$th update using the result of the $(k-10)$th update of processor i. The speed of computation and communication can be different at different processors, and there can be substantial communication delays. Furthermore, these speeds and delays can vary unpredictably as the algorithm progresses. It is not even required that the communication links preserve the order of messages, so it is possible for a processor to

use the kth update of some other processor at some time, and use the results of, say, the $(k-10)$th update of the same other processor at some later time. There is also flexibility regarding the initial conditions at each processor; the variables x_i available initially at different processors can differ, for the same i.

An asynchronous algorithm can potentially reduce the synchronization penalty caused by fast processors waiting for slow processors to complete their updates, and for slow communication channels to deliver messages (see Fig. 1.4.7). The reason is that processors can execute more iterations when they are not constrained to wait for the most recent results of the computation in other processors. There is a danger that iterations performed on the basis of outdated information will not be effective and may even be counterproductive. This issue will be considered in Chapters 6 and 7, where it will be seen that asynchronous iterative methods work only under certain conditions. When they do work, they typically reduce the synchronization penalty and result in faster problem solution. Their communication requirements, however, may exceed those of their synchronous counterparts. We provide a simple illustrative example. The conclusions from this example will be shown in more generality in Section 6.3.

Example 4.1. *Convergence Rate Comparison of Synchronous and Asynchronous Methods*

Consider the two–dimensional fixed point problem

$$x = Ax,$$

where

$$A = \begin{bmatrix} a & b \\ b & a \end{bmatrix}.$$

The corresponding iteration is

$$x_1 := ax_1 + bx_2, \tag{4.6a}$$

$$x_2 := bx_1 + ax_2, \tag{4.6b}$$

where variables x_1 and x_2 are updated at processors 1 and 2, respectively, and are subsequently communicated to the other processor. Suppose that each update requires one unit of time, and the subsequent communication requires D units of time, where D is a positive integer. (The following conclusions depend crucially on the fact $D \geq 1$; see Exercise 4.2.) We consider a synchronous and an asynchronous algorithm operating as shown in Fig. 1.4.7.

In the synchronous algorithm, values of variables are received at times $t = D + 1,\ 2(D+1), \ldots$; variable updates are also initiated at these times, as well as at time $t = 0$, and variable updates are completed one time unit later. If $x_i(t)$, $i = 1, 2$, is the value available at processor i at an integer time t, we have

$$x_1(t+1) = ax_1(t-D) + bx_2(t-D), \tag{4.7a}$$

$$x_2(t+1) = bx_1(t-D) + ax_2(t-D), \tag{4.7b}$$

where

$$x_i(t) = x_i(0), \qquad -D \leq t < 0, \quad i = 1, 2.$$

In the asynchronous algorithm, processor i updates variable x_i as many times as possible regardless of whether it has an up–to–date value of the variable of the other processor (cf. Fig. 1.4.7). Then $x_i(t)$ evolves for all t according to

$$x_1(t+1) = ax_1(t) + bx_2(t-D), \tag{4.8a}$$

$$x_2(t+1) = bx_1(t-D) + ax_2(t), \tag{4.8b}$$

where

$$x_i(t) = x_i(0), \qquad -D \leq t < 0, \quad i = 1, 2.$$

We now want to compare the synchronous iteration (4.7) with the asynchronous iteration (4.8) for the same initial conditions $x_1(0)$ and $x_2(0)$. We first consider the issue of convergence. This subject will be discussed in detail in Chapters 2 and 6. It can be shown that if

$$|a| + |b| < 1,$$

the synchronous iteration (4.7) converges to the unique fixed point $x^* = (0,0)$ starting from arbitrary initial conditions; let $D = 0$ in the following argument that applies to the asynchronous version of the iteration.

For the matrix A considered, convergence of the asynchronous iteration (4.8) is also guaranteed if $|a|+|b| < 1$. We prove this with an argument that we will use in more generality in Section 6.2 to show convergence of asynchronous fixed point iterations under a broad set of assumptions. The key idea is that if for some $t' \geq 0$ and all t with $t' - D \leq t \leq t'$, the vector $x(t) = \big(x_1(t), x_2(t)\big)$ belongs to the l_∞–sphere of radius r

$$\big\{x \big| |x_1| \leq r, |x_2| \leq r\big\}, \tag{4.9}$$

then from Eq. (4.8) it follows that $x(t)$ will belong to the (smaller) l_∞–sphere of radius $(|a| + |b|)r$ for all $t \geq t' + 1$. Based on this fact, it follows that $x(t)$ will belong to the (even smaller) l_∞–sphere of radius $(|a| + |b|)^2 r$ for all $t \geq t' + D + 2$, and so on, proving convergence of $x(t)$ to the zero vector.

We next consider the rate of convergence of the synchronous iteration (4.7). We first observe that if $\rho > 0$ is such that

$$|a|\rho^{-D} + |b|\rho^{-D} \leq \rho,$$

or, equivalently,

$$\big(|a| + |b|\big)^{1/(D+1)} \leq \rho, \tag{4.10}$$

then the sequence generated by the synchronous iteration (4.7) satisfies

$$|x_i(t)| \leq C\rho^t, \qquad \forall\, t = 0, 1, \ldots, \tag{4.11}$$

where

$$C = \max\{|x_1(0)|, |x_2(0)|\}. \tag{4.12}$$

Indeed Eq. (4.11) holds for $t \leq 0$. Assume that it holds for all t up to some $\bar{t}$. Then, using Eqs. (4.10) and (4.11), we have

$$|x_1(\bar{t}+1)| \leq |a||x_1(\bar{t}-D)| + |b||x_2(\bar{t}-D)| \leq \big(|a|+|b|\big)C\rho^{\bar{t}-D} \leq C\rho^{\bar{t}+1},$$

and similarly $|x_2(\bar{t}+1)| \leq C\rho^{\bar{t}+1}$. Therefore, Eq. (4.11) holds for all $t \leq \bar{t}+1$ and the induction is complete. The smallest value of ρ for which Eq. (4.10), and hence also the rate of convergence estimate $|x_i(t)| \leq C\rho^t$, holds is

$$\rho_S = \big(|a|+|b|\big)^{1/(D+1)}. \tag{4.13}$$

A little thought shows also that there exist initial conditions for which the rate of convergence estimate $|x_i(t)| \leq C\rho^t$ fails to hold when $\rho < \rho_S$.

A nearly verbatim repetition of the preceding argument can be used to show that if

$$|a| + |b|\rho^{-D} \leq \rho, \tag{4.14}$$

then the sequence generated by the asynchronous iteration (4.8) satisfies the rate of convergence estimate $|x_i(t)| \leq C\rho^t$, with C given by Eq. (4.12). The smallest ρ satisfying Eq. (4.14), denoted by ρ_A, is the unique positive solution of the equation

$$|a| + |b|\rho^{-D} = \rho. \tag{4.15}$$

The construction of ρ_A and its relation with ρ_S are illustrated in Fig. 1.4.8. It can be seen that

$$\rho_A \leq \rho_S,$$

and the convergence rate estimate of the asynchronous iteration is better than the one of the synchronous version (see also Fig. 1.4.9). This indicates that the asynchronous iteration typically converges faster. Computational results support this hypothesis.

The preceding conclusions can be generalized as will be shown in Section 6.3. For example, the matrix A could be any $n \times n$ matrix with elements a_{ij} satisfying $\sum_{j=1}^{n} |a_{ij}| < 1$ for all i. Also, the updating and the communication delays need not be equal for all processors, communication channels, and iterations. It appears that variability of these delays favors the asynchronous over the synchronous algorithm, but this seems difficult to establish analytically.

Consider now the number of messages required to solve the problem to within a given $\epsilon > 0$. If for a given t and ρ, we have $|x_i(t)| \leq C\rho^t$, we obtain $|x_i(t)| \leq \epsilon$ if $C\rho^t \leq \epsilon$

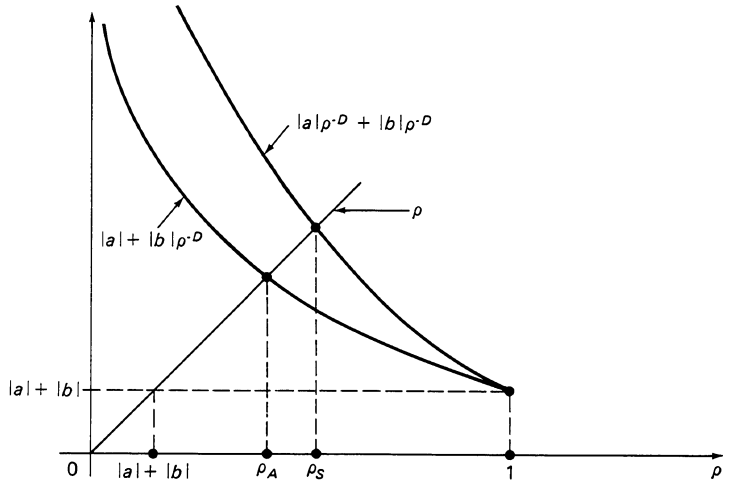


Figure 1.4.8 Construction of the convergence rate parameter ρ_A of the asynchronous iteration (4.8), and the corresponding parameter ρ_S for the synchronous iteration (4.7). We have $\rho_A < \rho_S$ except when $a = 0$ in which case $\rho_A = \rho_S$. For a given value of $|a| + |b|$, the ratio ρ_A/ρ_S is minimized when $|b| = 0$, in which case,

$$\frac{\rho_A}{\rho_S} = |a|^{D/(D+1)}.$$

More generally, as the strength of the coupling between the two variables x_1, and x_2 (i.e., the magnitude of $|b|$) decreases, the advantage of the asynchronous implementation in terms of rate of convergence increases.

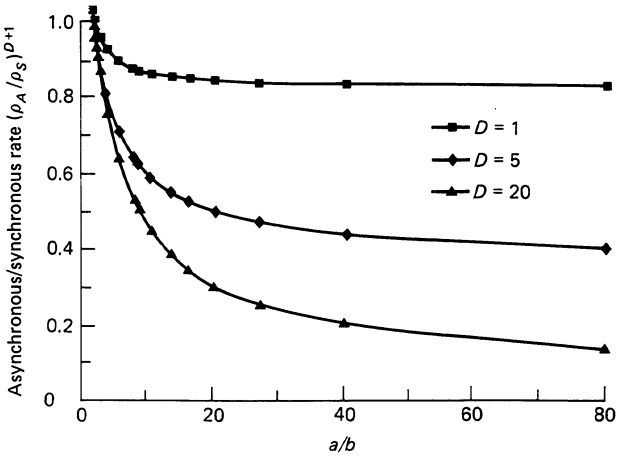


Figure 1.4.9 Plot of the value of $(\rho_A/\rho_S)^{(D+1)}$ for various values of the communication delay D, and the parameters a and b. In this example we have $|a| + |b| = 0.8$. The quantity $(\rho_A/\rho_S)^{(D+1)}$ is the ratio of error reduction factors per synchronous iteration ($D + 1$ time units), and tends to

$$\frac{|b|}{(1 - |a|)(|a| + |b|)}$$

as D increases to infinity.

or equivalently $t \geq |\log(\epsilon/C)|/|\log \rho|$. Therefore, if t_S and t_A are the times required to obtain $|x_i(t)| \leq \epsilon$, $i = 1, 2$, based on the convergence rate estimates for the synchronous and the asynchronous iterations, respectively, we obtain

$$\frac{t_S}{t_A} = \frac{|\log \rho_A|}{|\log \rho_S|} \geq 1.$$

The number m_S of message receptions per processor up to t_S for the synchronous iteration does not exceed $t_S/(D+1)$, whereas the corresponding number m_A for the asynchronous iteration is t_A (cf. Fig. 1.4.7). Therefore, using Eq. (4.13) for ρ_S, we have

$$\frac{m_S}{m_A} \leq \frac{|\log \rho_A|}{(D+1)|\log \rho_S|} = \frac{|\log \rho_A|}{|\log(|a| + |b|)|}.$$

From Fig. 1.4.8 we see that $|a| + |b| \leq \rho_A < 1$, so

$$\frac{m_S}{m_A} \leq 1.$$

The asynchronous algorithm requires more message transmissions because of the requirement of communication at the end of each variable update. An alternative asynchronous algorithm communicates the results every kth update, where $k \geq 2$ is some integer (but still updates every variable at each time unit). When $k = D + 1$ in the present example, the frequency of communication is the same as for the synchronous algorithm (4.7), and similar analysis as the one above (Exercise 4.3) shows that the asynchronous algorithm is guaranteed to solve the problem within any ϵ both faster and with fewer message transmissions than the synchronous version (but using a larger number of variable updates).

The conclusion in this example is that *asynchronism improves the convergence rate of iteration (4.6) but may increase the number of message transmissions.* In other words, *the communication penalty is reduced by asynchronous implementation at the expense of perhaps more frequent message exchange between processors.* This conclusion is consistent with other conclusions to be reached in related contexts. For example, in Section 6.4 we will see that the asynchronous Bellman–Ford algorithm takes no more time to solve a shortest path problem than the synchronous version, but may require many more message transmissions.

Note also that the tradeoff between execution time and number of message transmissions assumes that communication delays are not affected by the frequency of communication, that is, no queueing of messages occurs along communication links. With substantial queueing delays, an asynchronous algorithm can be much slower that its synchronous counterpart.

An important disadvantage of asynchronism is that it can destroy convergence properties that the algorithm may possess when executed synchronously. Indeed we will see (Section 6.3 and Chapter 7) that in some cases, it is necessary to place limitations on the size of communication delays to guarantee convergence. In all cases, the analysis of asynchronous algorithms is considerably more difficult than for their synchronous counterparts. We will also see that there are unifying themes in this analysis. In particular, it is possible to establish simultaneously the validity of important classes of

asynchronous algorithms with widely varying character through the use of general and powerful convergence theorems and techniques (see Chapters 6 and 7).

Asynchronous algorithms can offer an additional advantage in relaxation methods, which can be understood by considering the Jacobi and the Gauss–Seidel methods for solving fixed point problems (cf. Subsection 1.2.4). The sequential form of the Jacobi method generates a sequence $x(t) = \big(x_1(t), x_2(t), \ldots, x_n(t)\big)$ according to

$$x_i(t+1) = f_i\big(x_1(t), x_2(t), \ldots, x_n(t)\big), \qquad i = 1, 2, \ldots, n$$

from a given set of initial values. The Gauss–Seidel method is similar, but uses the most recently generated values of the variables x_i in the update formulas. It takes the form

$$\begin{aligned}
x_1(t+1) &= f_1\big(x_1(t), x_2(t), \ldots, x_{n-1}(t), x_n(t)\big) \\
x_2(t+1) &= f_2\big(x_1(t+1), x_2(t), \ldots, x_{n-1}(t), x_n(t)\big) \\
&\cdots \quad \cdots \quad \cdots \\
x_n(t+1) &= f_n\big(x_1(t+1), x_2(t+1), \ldots, x_{n-1}(t+1), x_n(t)\big).
\end{aligned}$$

The Jacobi method is better suited for distributed implementation; it is in effect equivalent to the synchronous distributed algorithm described earlier in connection with the update (4.5). The Gauss–Seidel method is not as well suited for parallel or distributed implementation. It is inherently sequential, although it can be parallelized to a substantial degree when the dependency graph is sparse; see Subsection 1.2.4. On the other hand the Gauss–Seidel method is frequently much faster than Jacobi; see Section 2.6. Generally speaking, convergence is typically accelerated if the updated values of the variables are incorporated into subsequent updates of other variables as quickly as possible. With some thought, it can be seen that in an asynchronous algorithm, updated values of variables can be incorporated into the computation faster than in a synchronous algorithm, as illustrated by Fig. 1.4.10. It is therefore plausible that an asynchronous algorithm can realize some of the speed advantage of Gauss–Seidel over Jacobi, without sacrificing any of the parallelism potential of Jacobi. This conjecture is supported only by limited computational experience at present.

EXERCISES

4.1. **[PaT87a]** Consider a synchronous algorithm operating in phases, and let $G(k)$ and $L(k)$ be the times required for all processors to complete k phases using the global and the local synchronization methods, respectively (cf. Subsection 1.4.1). Suppose that $P(i,t) = P_i$ for all i and t, and that the times $T_{ij}(t)$ required for processors i to send their messages to processors $j \in P_i$ are random, independent, and exponentially distributed with mean equal to 1. Assume that the number of elements in all the sets P_i is equal to some positive integer d. Show that

$$E\big[G(k)\big] = \Theta\big(k \log(nd)\big),$$

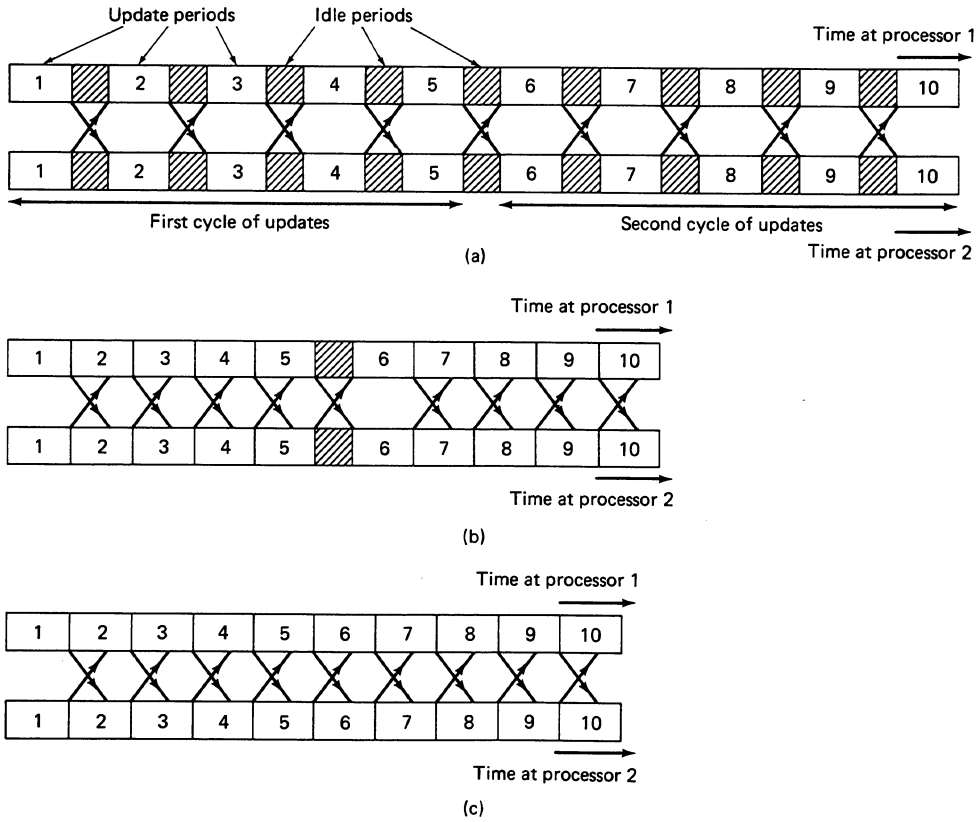


Figure 1.4.10 Illustration of the mechanism by which an asynchronous algorithm can realize the speed advantage of the Gauss–Seidel method over the Jacobi method. There are two processors solving a 10–variable problem and updating 5 variables each. The timing diagram in (a) is for a synchronous method that tries to incorporate new information as early as possible. This method has a large synchronization penalty. In the synchronous method of (b), the synchronization penalty is reduced through pipelining of computation and communication, but the updates of each processor are taken into account every five updates, thereby resulting in a Jacobi–like method. In the asynchronous method of (c), there is no synchronization penalty, while the result of each update is taken into account at the other processor after a delay of only one update. In this example, all updates take equal time. The synchronization penalty is exclusively due to communication delays. Unequal variable update times increase the synchronization penalty further.

whereas

$$E\big[L(k)\big] = \Theta(\log n + k \log d),$$

where $E[\cdot]$ denotes expectation. *Hint:* Use Props. D.1 and D.2 of Appendix D.

4.2. Consider the iteration (4.6) for the case where $|a| + |b| < 1$, $b \neq 0$, and the communication delay D is smaller than the time needed for an update. Show that for D sufficiently small, the synchronous algorithm has a better rate of convergence than the asynchronous algorithm.

4.3. Consider the iteration (4.6) for the case where $|a| + |b| < 1$, $a \neq 0$, let D be a positive integer and consider an asynchronous algorithm with limited communication that works as follows: values of variables are received at times $t = (D+1), 2(D+1), \ldots$ (just as in the synchronous version of Example 4.1), and variable updates are initiated at each time $t = 0, 1, \ldots$ and completed one time unit later (just as in the asynchronous version of Example 4.1). Thus, the update for variable x_1 and for $t = 0, (D+1), 2(D+1), \ldots$ has the form

$$\begin{aligned} x_1(t+1) &= ax_1(t) + bx_2(t-D), \\ x_1(t+2) &= ax_1(t+1) + bx_2(t-D), \\ &\cdots \qquad \cdots \\ x_1(t+D+1) &= ax_1(t+D) + bx_2(t-D). \end{aligned}$$

Show that for all $i = 1, 2$ and t of the form $t = (D+1), 2(D+1), \ldots$, we have

$$|x_i(t-k)| \leq \max\{|x_1(0)|, |x_2(0)|\}\rho_S^{t-D}, \qquad \forall\, k = 0, 1, \ldots, D,$$

where $\rho_S = (|a| + |b|)^{1/(D+1)}$ is the synchronous convergence rate parameter of Example 4.1. Show also that the asynchronous algorithm is better than the synchronous algorithm both in terms of execution time and in terms of number of message transmissions.

NOTES AND SOURCES

There are several texts and surveys that describe parallel and distributed computing systems; see, for example, [HoJ81], [KuP81], [HwB84], [Hwa84], [Hoc85], and [Hwa87]. See [Sch80] and [Hil85] for integrated discussions of several aspects of some interesting computing systems. There have been a number of special issues on parallel and distributed computation in several journals, as well as a number of journals dealing exclusively with the subject. There is a number of textbooks dealing with parallel algorithms, see e.g., [HoJ81], [Sch84], [Akl85], [Qui87], and [FJL88]. See [MeC80], [KSS81], [Kun82], and [Kun88] for parallel computation using VLSI systems and systolic algorithms. Finally, different aspects of parallel computation are surveyed in [QuD84], [OLR85], [OrV85], [KiL86], and [Rib87].

1.1. Interconnection networks and switching systems are surveyed in [BrH83], [WuF84], and [IEE87]. The classification of parallel computers into SIMD and MIMD systems is from [Fly66].

1.2.1. The DAG model for parallel computation has been extensively used for arithmetic problems [BoM75], as well for problems involving Boolean variables.

In another class of models of parallel computation (known as PRAM models, Parallel Random Access Machine), each processor is modeled as a random access machine and processors communicate to each other through a shared memory. There are a few

variations here depending on whether different processors are allowed to simultaneously read or write the same location in the shared memory. Some early references on this subject are [FoW78] and [Gol78]; see [OLR85] for a survey. For models of VLSI computation, see [Ull84].

1.2.2. Propositions 2.3 and 2.4 are from [Bre74]. The problem of finding a schedule that minimizes T_p has been thoroughly studied, see [Cof76] and [OLR85], for example. It can be efficiently solved when the number p of processors is 2 or when the DAG is a tree, but it is NP–complete in general.

The scheduling problem for the case where communication costs are taken into account has been considered in [PaY88]. In this context, it is sometimes preferable to let more than one processor perform the operation corresponding to some node, in order to avoid communication delays.

The construction of optimal or near–optimal parallel algorithms (that is, DAGs with close to minimal depth) for a given computational problem, has been well studied, particularly for the case of arithmetic problems (see e.g., [Bre74] and [MuP76]). See also [MiR85] for a method whereby a tree–like DAG is restructured on–line to obtain logarithmic depth.

Complexity measures for synchronous parallel computation have provided an interesting extension of the theory of computational complexity that had originated in the context of serial computation; see [OLR85]. Of particular interest here is the class NC, which is the class of problems that can be solved in time $O(\log^k n)$ using $O(n^k)$ processors, where k is some integer, and n is the size of the problem [Coo81]. This is sometimes described as the class of problems most amenable to massive parallelization.

1.2.3. Parallel algorithms have been developed for a variety of algebraic problems, such as polynomial evaluation ([MuP73] and [Mar73]), linear recurrences ([Kun76a] and [HyK77]), etc.

1.2.4. Proposition 2.6 is well known but Prop. 2.5 seems to be new. For further references on the application of coloring in the parallelization of Gauss–Seidel algorithms in the context of the numerical solution of partial differential equations see [OrV85]. The parallelization of iterations with a particular type of a regular structure is considered in [KMW67] and [RaK88].

1.3.1. Extensive discussions of communication network issues appear in a number of textbooks, such as [BeG87], [Hay84], [Sch87], [Sta85], and [Tan81].

1.3.2. For a view of the go–back–n DLC as a distributed algorithm and a corresponding analysis, see [BeG87].

1.3.3. Routing algorithms for data networks are discussed extensively in [BeG87]. For analysis of randomized routing algorithms in regular interconnection networks, see [VaB81], [Val82], [HaC87], and [MiC87].

1.3.4. The mapping problem for general architectures has been considered in [Bok81]. The problem of mapping various topologies on hypercubes is discussed in

a number of sources including [BhI85], [SaS88], and [McV87]. Communication complexity analyses of various basic algorithms for hypercubes and other topologies can be found in [SaS85], [SaS86], and [McV87]. Our discussion is motivated by these references, and improves upon them in that it provides several new algorithms and problem complexity estimates. In particular, the multinode broadcast algorithm for a hypercube is new, and is the first that takes the minimal $\left(\lceil (2^d - 1)/d \rceil\right)$ number of steps (a related algorithm was given earlier in [Ozv87]; an $O(2^d/d)$ time algorithm was first derived in [SaS85]). The total exchange algorithms of Fig. 1.3.19 and Exercise 3.6 are new. Private communications by P. Tseng relating to this section have been very helpful.

1.3.5. For a discussion of the concurrency and communication tradeoff in the context of matrix computations, see [GHN87].

1.3.6. Matrix calculations on hypercubes are discussed extensively in [Joh87a], and [McV87]. Several matrix multiplication algorithms on a mesh and a hypercube are given in [DNS81].

1.4.1. The complexity of a number of synchronization algorithms is analyzed in [Awe85]. Communication analyses of synchronized iterative methods are given in [DuB82]. The comparison between the local and the global synchronization methods is new.

1.4.2. An early comparative discussion of synchronous and asynchronous algorithms is [Kun76b]. The convergence rate comparison between synchronous and asynchronous relaxation methods given in Example 4.1 is new; it will be generalized in Subsection 6.3.5.