Parallel Program Design

A Foundation

K. Mani Chandy

Jayadev Misra

University of Texas at Austin

ADDISON-WESLEY PUBLISHING COMPANY

Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam
Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

Library of Congress Cataloging-in-Publication Data

Chandy, K. Mani.
Parallel program design : a foundation / by K. Mani Chandy and Jayadev Misra.
p. cm.
ISBN 0-201-05866-9 :
1. Parallel programming (Computer science) I. Misra, Jayadev. II. Title.
QA76.6.C42818 1988 87-26124
004'.35--dc19 CIP

BCDEFGHIJ-HA-898

To our families

Jean	Mamata
Christa	Amitav
Mani	Anuj
Chandy	Sashibhusan
Rebecca	Shanty

"The most general definition of beauty...
Multeity in Unity."

Samuel Taylor Coleridge
On the Principles of Genial Criticism (1814)

Foreword

A complete theory of programming includes

1. A method for specification of programs which permits individual requirements to be clearly stated and combined.
2. A method of reasoning about specifications, which aids in elucidation and evaluation of alternative designs.
3. A method of developing programs together with a proof that they meet their specification.
4. A method of transforming programs to achieve high efficiency on the machines available for their execution.

It is not often that we can welcome the advent of a new theory of programming. Twelve years ago, E.W. Dijkstra published his *Discipline of Programming*, which is still a definitive source-book on the development of sequential algorithms. And now Chandy and Misra have shown how Dijkstra's methods, and other more recent developments, can be generalized to distributed and concurrent algorithms. Their work deserves the warmest welcome.

This book treats all essential aspects of the theory of programming. The underlying logic is developed with elegance and rigour. It is illustrated by clear exposition of many simple examples. It is then applied, with matching elegance and simplicity, to a range of examples which have hitherto justified a reputation of baffling complexity. The authors' technique and style are worthy of imitation by all who publish descriptions of distributed algorithms.

The book will be studied with enjoyment and profit by many classes of computing scientists. Teachers will welcome it as a text for an advanced class on programming. Software engineers will use it as a handbook of methods and algorithms for the design of distributed systems. And theoreticians will find a rich source of research ideas that promise to be of relevance in the professional activity of programming.

C. A. R. Hoare

PREFACE

Our Prejudices about Programs, Architectures, and Applications

Goal

The thesis of this book is that the unity of the programming task transcends differences between the architectures on which programs can be executed and the application domains from which problems are drawn. Our goal is to show how programs can be developed systematically for a variety of architectures and applications. The foundation, on which program development is based, is a simple theory: a model of computation and an associated proof system.

Architectures

The architectures considered cover a wide spectrum, including sequential machines, synchronous and asynchronous shared-memory multiprocessor systems, and message-based distributed systems. We propose that there is a continuum from the design of programs to the design of electronic circuits. We also derive programs for fault-tolerant systems in the same way we derive programs for fault-free systems.

Application Areas

We develop programs in a uniform manner for a variety of application areas. Many of the problems considered here appeared initially as operating systems problems, including termination detection of asynchronous programs, garbage collection, and conflict resolution in parallel systems. However, programmers who write so-called "application" programs for execution on parallel architectures are also concerned with these problems—for instance, detecting the termination of a distributed application program is not always a trivial task. We also develop programs for combinatorial problems (shortest path, sorting, combinatorial search, etc.), matrix problems, and communication protocols.

Program Development

This book is based on a small theoretical foundation that is applied to all problems. The theory, which uses elementary mathematics, is described in the first few chapters. A continuum of solutions is developed for each problem, starting with simple programs, i.e., programs with relatively simple proofs. The specifications and proofs are refined to obtain efficient programs for various classes of target architectures.

A small number of heuristics is suggested to refine specifications and programs. Methods to compose larger programs from smaller ones are also suggested. The theory and the heuristics are the unifying framework for the book.

There are many ways to develop programs and even more ways to write books about them. When we began working on this book we were faced with several alternatives. The choices we made, and our reasons for doing so, are described next. Each alternative had some merit. Ultimately our choices were made on the basis of personal preference. By describing the alternatives and our choices, we hope to give the reader an idea of what sort of book this is—and equally important, what sort of book this is not.

Choices in a Study of Programming

Design versus Coding

A conventional view of programming is that given a specification, a programmer produces code and then proves that the code meets the specification. A book based on this view emphasizes programming languages and proofs of program texts. This book, by contrast, is about the *design* of programs. A design consists of a series of small decisions, each dealing with a manageable set of concerns. The larger part of program design is concerned with the stepwise refinement of specifications; the production of code is postponed until the final stages. Therefore the emphasis here is on a model of computation, a notation for specifications, and a theory for proving the correctness of specifications. Much of this book deals with the notation and proofs of *designs*; programming languages and proofs of program texts are accorded only secondary importance. The derivation of a program (in any programming language) from the final specification should be the most mechanical and least creative part of programming.

Taxonomy versus Foundation

There are several paradigms for the development of science. On the one hand, theoretical physics attempts to find the fundamental laws that explain all physical phenomena; the smaller the number of laws, the better. On the other hand, much of experimental botany is concerned with observing, classifying, and describing what exists. In writing this book we were faced with the choice either of attempting to propose a foundation for parallel programming and then applying it to a variety of problems, or of observing, classifying, and describing programs and architectures. Certainly, there is a great deal of interest in program and machine taxonomy. There are categories of machines (sequential machines, single-instruction-multiple-data machines, multiple-instruction-multiple-data machines, etc.), categories of systems (message-based, shared-variable-based, synchronous, asynchronous, etc.), categories of programming styles (functional, logic, imperative, object-oriented, etc.) and categories of applications (operating systems, communication protocols, expert systems, etc.). The advantage of the observation-classification-description paradigm is that it gives the reader a comprehensive description of what exists; very little is left out. Furthermore, it suggests a solution strategy: Given a problem, one looks for the categories that the problem falls into and uses solutions that have proved successful. Finally, a careful taxonomy is guaranteed to be useful. Nevertheless, we have chosen the riskier option of proposing a small foundation and basing our book on it. Our reason is subjective. Today programming appears to be fragmented into increasingly esoteric subdisciplines, each with its priests, acolytes, and incantations. We believe that there is a unity underlying all programming; we hope to contribute to its appreciation.

Choice of Foundation

There seems to be some consensus in physics as to which laws are fundamental: The fundamental laws are those from which all others are derived. In programming, the choice of a foundation—the computational model and proof system—is less clear-cut. The Turing Machine is not our model of choice, though it is a basic model, because it is inappropriate for our purposes—the development of programs. We have a strong personal preference, however, for making do with less (when it comes to computational models). Therefore we have attempted to design a computational model and associated theory with the smallest number of concepts adequate for our purposes.

There are alternatives to our approach of applying a small theory to a wide range of problems. A notation (and an associated theory) rich in expressive power can be used, or many theories can be proposed—an appropriate one for each problem area. Employing rich notations and several theories has advantages, the most obvious of which is that a skillful choice of theory in solving a problem may result in an elegant solution. Our choice of a small theory is based on our desire to explore the unity of all program development. We are willing to pay the price of less elegant solutions for the advantage of a unified framework.

Formal versus Informal Descriptions of Programs

Programs, especially parallel programs, are often described informally. Problems, such as that of the dining philosophers, are posed and solved without using a formal notation. There are advantages to informal descriptions. A problem can be studied without the baggage of a formal notation and proof system. Also, word pictures such as philosophers sitting around a table sending forks to one another are vivid and helpful. We have attempted to take advantage of the benefits that informal reasoning has to offer, but we have chosen to employ a formal notation.

In our choice of programming notation, we limited ourselves to one that *we* could handle mathematically. We limited our expressive power in this way because we are fallible as programmers, and we hope that the use of mathematics will make us less so. We have been amazed at the errors we have made in informal arguments. Programs that seemed so obviously correct at one time are, in retrospect, so obviously wrong. It is humility as programmers that drives us to seek formalism and mathematics. We are fallible as mathematicians as well, of course, but the discipline helps.

Our desire for a simple unifying framework has led to a restricted notation. Restricting concepts to a small, mathematically manageable set has resulted in our not exploring some interesting avenues of research. For instance, we are afraid of self-modifying rule-based systems—in which new rules are added as computation proceeds—because we do not know how to construct proofs for such systems.

Operational versus Nonoperational Reasoning about Programs

One can reason about the unfolding computations of a program (this is called operational reasoning) or one can focus, as much as possible, on static aspects (such as invariants) of the program. We favor the static view here for three reasons. First, we made more mistakes when we used operational reasoning—for example, a common mistake is forgetting to consider certain sequences of events that could occur. Second, we have found it hard to convince skeptics about the correctness of our programs by using operational arguments. (We speak from our experience in teaching distributed systems to many students who are, quite properly, skeptics. Inevitably, a student asks, "But, you don't seem to have considered the case where B follows C which follows D which. . .?" Operational arguments have reduced us to saying, after fielding several such questions, "Check it out for yourself and you'll see that it is okay.") Third, our operational arguments tend to be longer.

Operational reasoning has value. Again, being very subjective, we have found that the flash of insight that sparks the creation of an algorithm is often based on operational, and even anthropomorphic, reasoning. Operational reasoning by itself, however, has gotten us into trouble often enough that we are afraid of relying on it exclusively. Therefore we reason formally about properties of a program, using predicates about *all* states that may occur during execution of the program.

The Medium versus the Message

The message of this book is that a small theory—a computation model and its associated proof system—is adequate for program design in a variety of application areas and for a variety of architectures. The medium in which this message is couched is a (small amount of) notation to express the programs and their properties. We found that we could not express our ideas without introducing some notation. However, *this book is not about a programming language*. We have paid no attention to several important aspects of programming languages (such as data structures) and little attention to others (such as abstraction mechanisms). We wish to emphasize the message; that is, the theory. We have attempted to write a book that is useful to programmers no matter what programming language they employ. It is our hope that programmers will benefit from our theory for developing programs written in languages of their choice.

Choice of Material

The grandeur of our vision is constrained by the limitation of our ability (and also by the practical necessity of having to finish this book). We have not done everything we wanted to do. In our choice of architectures and applications we have attempted to demonstrate the unity of the programming task. However, we have not given some architectures the space they deserve, and we have omitted some application areas altogether—a situation we hope to remedy in future editions. For instance, designs of electronic circuits and fault-tolerant systems do not receive the attention they deserve. A significant omission is real-time systems. Other significant omissions include the study of complexity models such as NC and different modal logics appropriate for reasoning about parallel programs.

The Bibliography at the end of the book is far from exhaustive. Many of the references are books and survey papers that point to large lists of references and the original sources.

Reading this Book

Who Should Read It?

This book is for those interested in program design and computer architecture. The ideas have been used, and are being used, to teach an introductory (i.e., first-semester) graduate course, and a more advanced, second-semester course in computer science. However, the reader should be warned that the view of program design taken here is somewhat idiosyncratic; the contents do not exactly match the subject matter of a standard course in a standard curriculum. We hope that this preface gives readers some idea of our prejudices, which should help them decide whether to continue reading.

Few prerequisites are needed to read the book. It is based on a small theory that is described in Chapters 3 and 7. The theory employs elementary mathematics including the predicate calculus.

The sources of the example problems are not described in detail. For instance, in describing garbage collection, the origins of the problem in memory management are described with more brevity than in most programming textbooks. Problems are specified formally, so there is little likelihood that the reader will misunderstand them; however, the reader who is not aware of the problems may not appreciate their practical significance. Thus readers should have some maturity in programming or should believe the authors that the problems studied here are genuine.

How to Read It

Read this preface! Here we confess our prejudices, and readers whose prejudices don't match ours are best forewarned.

Most of the sections in this book contain both informal and formal parts. The informal part is an English description with little formal notation. The formal part contains the specifications, programs, and proofs employing the notation presented in Chapters 2, 3, and 7. A reader can learn the main ideas of a chapter by reading all the informal parts and skipping the formal ones. A reader who wishes to obtain all the ideas in a chapter should read it in sequence: The informal part describes what is coming in the next formal part. There are a couple of other ways of reading this book that have proved helpful, depending on the reader's facility with formalisms. One is to read all the informal parts before reading the entire chapter in sequence. The other is to read the informal part and then carry out the derivation in the formal parts oneself, treating each formal section as a worked-out exercise; this is the best way to read the book because it requires the active participation of the reader.

We recommend that the first five chapters be read sequentially; these chapters present the theory and apply it to one example in detail. Chapter 6—"Toy Examples"—illustrates the theory; the reader can pick and choose among these examples. Chapter 7 presents a theory for constructing large programs by composing small ones. Chapter 8 considers a special case of this theory, for the construction of process networks. Chapters 9 through 16 describe operating systems problems; the later chapters in this sequence are based on earlier ones. Chapters 17 and 18 deal with fault-tolerant systems. The next two chapters deal with combinatorial problems—sorting and combinatorial-search strategies. Chapter 21 deals with algorithms for systolic arrays. In Chapter 22, different programming styles—sequential, functional, logical—are contrasted with the style adopted in this book; the chapter also includes the rationale for some of the design decisions of our theory. In the epilog, we give some random thoughts on programming. The initial four chapters and Chapter 7 are necessary for understanding the later ones; the remaining chapters are largely independent and may be read in any order.

A first-level graduate course can be designed around the first five chapters plus one chapter each on combinatorics, operating systems, fault tolerance, and systolic arrays.

Acknowledgments

Our ideas about programming have been deeply influenced by arguments and discussions over a period of years with Edsger W. Dijkstra. Detailed and generous criticism by C. A. R. Hoare helped us understand some of the implications of our work; we are especially indebted to him for his insistence on the importance of designing as opposed to coding, and of proofs of specifications as opposed to proofs of programs. We are thankful to our graduate students, in many sections of CS380D and CS390D at The University of Texas at Austin and in CS131 at Caltech, and the members of the reading course 6.893 at MIT, who have suggested valuable technical and editorial improvements; special thanks go to Alan Fekete, Edgar Knapp, and Mark Staskauskas. Interactions with various individuals in the Concurrent Programming Institute (held under the auspices of The University of Texas Year of Programming, March 2–6, 1987, Austin, Texas) clarified several cloudy issues. We are grateful to the following individuals and groups, who have commented on our work and the earlier drafts of this book: K. R. Apt, the Austin Tuesday Afternoon Club, Manfred Broy, The Eindhoven Tuesday Afternoon Club, Allen Emerson, He Ji Feng, Ira Forman, A. J. M. van Gasteren, M. Gouda, David Gries, Rick Hehner, Ted Herman, Hank Korth, Ben Kuipers, Simon Lam, Chris Lengauer, M. D. McIlroy, Al Mok, Ben Moszkowski, Avi Silberschatz, and Chuck Seitz. We are especially indebted to Nissim Francez, Leslie Lamport, Amir Pnueli, and Jan L. A. van de Snepscheut, for their constructive criticisms. We are grateful to the following reviewers for their detailed comments: Bruce Arden, University of Rochester; Stephen Garland, MIT Laboratory for Computer Science; Maurice Herlihy, Carnegie-Mellon University; Nancy Lynch, Massachusetts Institute of Technology; Alain Martin, California Institute of Technology; Fred B. Schneider, Cornell University; Ken Sevcik, University of Toronto; and William Weihl, MIT Laboratory for Computer Science. Thanks also to Bob Comer for his implementation of our notation on an IBM PC.

Over the years we have received generous research grants for this work from the Air Force Office of Scientific Research and the Office of Naval Research; we are thankful for their continuing confidence. IBM supported the early work in this area and we are especially grateful to Fred May of the Austin Division for his help.

Special thanks go to Nancy Lawler for her outstanding editorial and production skills; this book would not have been possible without her assistance. We are also thankful to Debra Davis, Julie Barrow, and Mary Ghaleb, who typed various portions of the manuscript.

It is a particular pleasure to acknowledge the help of the editorial staff at Addison-Wesley.

Austin, Texas

K. M. C.
J. M.

Contents