HZ BOOKS
华章教育

经　典　原　版　书　库

PEARSON
Addison Wesley

并行程序设计原理

（英文版）

PRINCIPLES OF
PARALLEL
PROGRAMMING

CALVIN LIN
LAWRENCE SNYDER

（美） Calvin Lin
得克萨斯大学奥斯汀分校
Lawrence Snyder
华盛顿大学西雅图分校 著

机械工业出版社
China Machine Press

并行程序设计原理

（英文版）

Principles of Parallel Programming

（美） Calvin Lin
得克萨斯大学奥斯汀分校
Lawrence Snyder 著
华盛顿大学西雅图分校

机械工业出版社
China Machine Press

本书版权登记号：图字：01-2008-3270

图书在版编目（CIP）数据

并行程序设计原理（英文版）/（美）林（Lin, C.），（美）斯奈德（Snyder, L.）著. —北京：机械工业出版社，2008.8

（经典原版书库）

书名原文：Principles of Parallel Programming

ISBN 978-7-111-24734-0

Ⅰ.并…　Ⅱ.① 林…　② 斯…　Ⅲ.并行程序－程序设计－英文　Ⅳ.TP311.11

中国版本图书馆CIP数据核字（2008）第109994号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：迟振春

北京瑞德印刷有限公司印刷·新华书店北京发行所发行

2008年8月第1版第1次印刷

170mm×242mm · 22.25印张

标准书号：ISBN 978-7-111-24734-0

定价：49.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevien，MIT，John Wiley & Sons Wiley，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzedu@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

HZ BOOKS
华章教育

华章科技图书出版中心

To Mom and Dad
(Josette and Min Shuey)

To Julie, Dave, and Dan

Preface

Welcome!

For readers who are motivated by the advent of multi-core chips to learn parallel programming, you've come to the right place. This book is written for a world in which parallel computers are everywhere, ranging from laptops with two-core chips to supercomputers to huge data-center clusters that index the Internet.

This book focuses on scalable parallelism, that is, the ability of a parallel program to run well on any number of processors. This notion is critical for two reasons: (1) Most of the techniques needed to create scalable parallel computations are the same techniques that produce efficient solutions on a multi-core chip, and (2) while multi-core chips currently have a modest number of processors, typically 2–8, the number of cores per chip promises to increase dramatically in the coming years, making the notion of scalable parallelism directly relevant. Thus, while today's multi-core chips offer opportunities for low latency communication among cores, this characteristic is likely a short-term advantage, as on-chip delays to different parts of the chip will become increasingly apparent as the number of cores grows. So, we focus not on exploiting such short-term advantages, but on emphasizing approaches that work well now and in the future. Of course, multi-core chips present their own challenges, particularly with their limited bandwidth to off-chip memory and their limited aggregate on-chip cache. This book discusses these issues as well.

First, we discuss the principles that underlie effective and efficient parallel programs. Learning the principles is essential to acquiring any capability as sophisticated as programming, of course, but principles are perhaps even more important for parallel programming because the state of the art changes rapidly. Training that is tied too closely to a specific computer or language will not have the staying power needed to keep pace with advancing technology. But the principles—concepts that apply to any parallel computing system and ideas that exploit these features—lead to an understanding that is timeless and knowledge that will always be applicable.

But we do more than discuss abstract concepts. We also apply those principles to everyday computations, which makes the book very practical. We introduce several parallel programming systems, and we describe how to apply the principles in those programming systems. On completion, we expect readers to be able to write parallel programs. Indeed, the final chapter is devoted to parallel programming techniques and the development of a term-long parallel programming capstone project.

Audience

Our intended audience is anyone—students or professionals—who has written successful programs in C or similar languages and who describes himself as a programmer. It is helpful to have a basic idea of how a computer executes sequential programs, including knowledge of the fetch/execute cycle and basics of caching. This book was originally targeted to upper level undergraduate computer science majors or first year graduate students with a CS undergraduate degree, and it continues to be appropriate for that level. However, as the book evolved, we reduced the assumed knowledge and emphasized pedagogy in the belief that if some explanations cover knowledge the reader already has, it's easy to skip forward.

Organization

Because parallel programming is not a direct extension of sequential programming with which the reader is doubtless familiar, we have organized this book into four parts:

Foundations: Chapters 1–3

Abstractions: Chapters 4–5

Languages: Chapters 6–9

Looking Forward: Chapters 10–11

To enable you to select intelligently from these parts, we now explain their goals and content.

Foundations. In Chapter 1 we discover the many issues that parallel programmers must address by showing how difficult it is to implement a computation that is trivial when written for sequential computers. The example focuses our attention on issues that concern us throughout the entire book, but it also emphasizes the importance of understanding how a parallel computer operates. Chapter 2 introduces five different types of parallel computers, giving a few details about their architecture and their ability to scale to a larger size. There are two key conclusions from the chapter: First, unlike sequential computing, there is no standard architecture. Second, to be successful at spanning this architectural diversity we need an abstract machine model to guide our programming. And we give one. With the architectures in mind, Chapter 3 covers basic ideas of concurrency, including threads and processes, latency, bandwidth, speedup, and so forth, with an emphasis on issues related to performance. These foundations of Part 1 prepare us for an exploration of algorithms and abstractions.

Abstractions. As an aid to designing and discussing parallel algorithms, Chapter 4 introduces an informal pseuodcode notation for writing parallel programs in a language-independent way. The notation has a variety of features that span various programming models and approaches, allowing us to discuss algorithms without bias toward any particular language or machine. To bootstrap your thinking about parallel algorithms, Chapter 5 covers a series of basic algorithmic techniques. By the end of Part 2, you should be able to conceptualize ways to solve a problem in parallel, bringing us to the final issue of encoding your algorithms in a concrete parallel programming language.

Languages. There is no single parallel programming language that fulfills the role that, say, C or Java plays in sequential programming, that is, a language widely known and accepted as a baseline medium to encode algorithms. As a result, Part 3 introduces three kinds of parallel programming languages: thread-based (Chapter 6), message-passing (Chapter 7), and high-level (Chapter 8). We cover each language well enough for you to write small exercises; serious computations require a more complete language introduction that is available through online resources. In addition to introducing a language, each chapter includes a brief overview of related languages that have a following in the parallel programming community. Chapter 9 briefly compares and contrasts all of the languages presented, noting their strengths and weaknesses. There is benefit to reading all three chapters, but we realize that many readers will focus on one approach, so these chapters are independent of one another.

Onward. Part 4 looks to the future. Chapter 10 covers a series of new, promising parallel technologies that will doubtless impact future research and practice. In our view, they are not quite "ready for prime time," but they are important and worth becoming familiar with even before they are fully deployed. Finally, Chapter 11 focuses on hands-on techniques for programming parallel machines. The first two sections of the chapter can be read early in your study of parallel programming, perhaps together with your study of abstractions in Chapters 4 and 5. But the main goal of the chapter is to assist you in writing a substantial program as a capstone design project. In this capacity we assume that you will return to Chapter 11 repeatedly.

Using This Book

Although the content is presented in a logical order, it is not necessary to read this book front to back. Indeed, in a one term course, it may be sensible to begin programming exercises before all of the topics have been introduced. We see the following as a sensible general plan:

- Chapters 1, 2
- Chapter 11 first section, Chapter 3 through Performance Tradeoffs; begin programming exercises
- Chapters 4, 5

- One of Chapters 6–8, programming language chapters
- Complete Chapter 3 and 11, begin term project
- Complete remaining chapters in order: language chapters, Chapters 9, 10

There is, of course, no harm in reading the book straight through, but the advantage of this approach is that the reading and programming can proceed in parallel.

Acknowledgments

Sincere thanks are due to E Christopher Lewis and Robert van de Geijn, who critiqued an early draft of this book. Thanks also to the following reviewers for their valuable feedback and suggestions:

David Bader, Georgia Institute of Technology
Purushotham Bangalore, University of Alabama, Birmingham
John Cavazos, University of Delaware
Sandhya Dwarkadas, University of Rochester
John Gilbert, UC Santa Barbara
Robert Henry, Cray Inc.
E Christopher Lewis, VMWare
Kai Li, Princeton
Glenn Reinman, UCLA
Darko Stefanovic, University of New Mexico

We thank Karthik Murthy and Brandon Plost for their assistance in writing and running parallel programs and for finding bugs in the text, and we are grateful to Bobby Blumofe, whose early collaborations on a multi-threaded programming course are evident in many places in the book. We recognize and thank the students of the Parallel Programming Environments Seminar (CSE590o) at the University of Washington in autumn quarter, 2006 for their contributions to the text: Ivan Beschastnikh, Alex Colburn, Roxana Geambasu, Sangyun Hahn, Ethan Katz-Bassett, Nathan Kuchta, Harsha Madhyastha, Marianne Shaw, Brian Van Essen, and Benjamin Ylvisaker. Other contributors are Sonja Keserovic, Kate Moore, Brad Chamberlain, Steven Deitz, Dan Grossman, Jeff Diamond, Don Fussell, Bill Mark, and David Mohr.

We would like to thank our editor, Matt Goldstein, and the Addison Wesley team: Sarah Milmore, Marilyn Lloyd, Barbara Atkinson, Joyce Wells, and Chris Kelly. Thanks to Gillian Hall who has been especially tolerant of our antics.

Finally, we thank our families for their patience through the writing of this book.

Calvin Lin
Lawrence Snyder
February 2008

Contents

part 1

Foundations

We begin our study of parallel programming by building a solid foundation. The most important goal is to clarify the difference between the sequential and parallel programming worlds. In sequential computing, operations are performed one at a time, making it straightforward to reason about the correctness and performance characteristics of a program. In parallel computing many operations take place at once, complicating our reasoning about correctness and performance, and as a result, modifying our programming approach. This part explains the main consequences of this distinction.

Our introduction to parallel computation in Chapter 1 begins by solving a simple problem of counting the number of occurrences of 3 in a 1-dimensional array. This trivial task requires four attempts before we create a program with reasonable performance. Even then, we find that our maximum hoped-for speedup can't be realized. While working through the example, we introduce a series of basic concepts of parallelism.

Chapter 2 describes the basic architectural features of parallel computers. It is an interesting topic in its own right, because challenging problems such as interprocessor communication have a multitude of potential solutions, and the techniques that architects use to address them exhibit considerable ingenuity. The main conclusion of our tour of parallel machines will be that they are extremely different. Because programmers need to know certain properties of the underlying machine to write quality programs, it will be necessary to find a machine model that unifies the disparate architectures. We introduce such a model as the basis for our subsequent study.

With a clear idea of how parallel computers work, Chapter 3 characterizes the many conceptual issues surrounding parallel performance. We introduce key ideas including latency, bandwidth, speedup and efficiency. Certain facets of programming, such as dependences, are highlighted as being a source of interference among parallel threads. Once these foundational ideas have been introduced, we will be prepared to move on to the algorithmic ideas presented in Part 2.

chapter

1 Introduction

Parallel computation is a fundamental technique by which computations can be accelerated, so the increasing availability of parallel hardware represents a tremendous opportunity. But implementing a parallel solution presents certain conceptual and programming challenges that this textbook is designed to address. To place the opportunities and challenges in perspective, this chapter sets the context and introduces basic ideas.

The Power and Potential of Parallelism

Parallelism arises frequently in everyday life. More importantly, parallelism has contributed in many ways to the steady performance improvement in computers over the past several decades. And now, new opportunities are available. Let's look closely.

Parallelism, a Familiar Concept

Parallelism is a familiar concept. Juggling is a parallel task that humans can perform. House construction is a parallel activity, because several workers can perform separate tasks simultaneously, such as wiring, plumbing, and furnace duct installation, and so on. Most manufacturing—cars, hairdryers, frozen dinners—is performed in parallel using an assembly line, or pipeline, in which many units of the product are under construction at once. A call center, where many employees service customers at the same time, is another organization that applies parallelism.

Although familiar, these forms of parallelism are different. The call center, for example, differs from house construction in a fundamental way: Calls are generally independent and can be serviced in any order with little interaction among the workers. In construction, some tasks can be performed simultaneously—wiring and plumbing—while others are ordered—framing must precede wiring. The ordering restricts the amount of parallelism that can be applied at once, limiting the speed at which a construction project can complete. The ordering also increases the degree of interaction among the workers. Manufacturing pipelines are different still, because they generally have strict ordering constraints with the separate stages often being performed sequentially; the parallelism comes from having many instances of the product in the pipeline at once. And juggling is an instance of event-driven parallelism, where an event—a falling ball—causes the execution of operations—catching, throwing—in response to the event. Such familiar forms of parallelism will also arise in our consideration of parallel computation.

Parallelism in Computer Programs

The main motivation for executing program instructions in parallel is to complete a computation faster. But most programs today are incapable of much improvement through parallelism, because they were written assuming that the instructions would be executed in order, one at a time, that is, *sequentially*. The semantics of most programming languages embed sequential execution, and the resulting programs typically rely so heavily on this property for their correctness that it is rare to find significant opportunities for parallel execution. To be sure, there are some opportunities, as when the expression `(a+b)*(c+d)` must be evaluated; assuming these are simple variables, the subexpressions `(a+b)` and `(c+d)` are independent of each other, so they can be computed simultaneously. Such opportunities are an example of Instruction Level Parallelism (ILP).

Indeed, one reason that we have continued to write sequential programs is because computer architects have been so successful at exploiting parallelism. They have used the steady improvements in silicon technology to add several kinds of parallelism, including ILP, into sequential processor design. First, architects provide separate wires and caches for instructions and data. The separation allows instruction and data memory references to execute in parallel without interfering. Second, instruction execution is pipelined, fetching and decoding future instructions while the current instruction is being executed and while the results of past instructions are still being written to memory. Furthermore, the processors issue (initiate) more than one instruction at a time, they prefetch instructions and data, they speculatively perform operations in parallel even if they cannot be sure that they will be needed, and they use highly parallel circuits to perform basic arithmetic operations. In short, modern processors are highly parallel systems.

The key point for programmers is that all of this parallelism has been transparently available to sequential programs. We call this *hidden parallelism*. Such parallelism, together with increasing clock speeds, has allowed each succeeding generation of processor chip to execute programs faster, while preserving the illusion of sequential execution. But the prospects for finding new opportunities to apply parallelism while preserving sequential semantics are becoming limited. More seriously, existing techniques for exploiting ILP have largely reached the point of diminishing returns, in terms of both power consumption and performance. So, given current