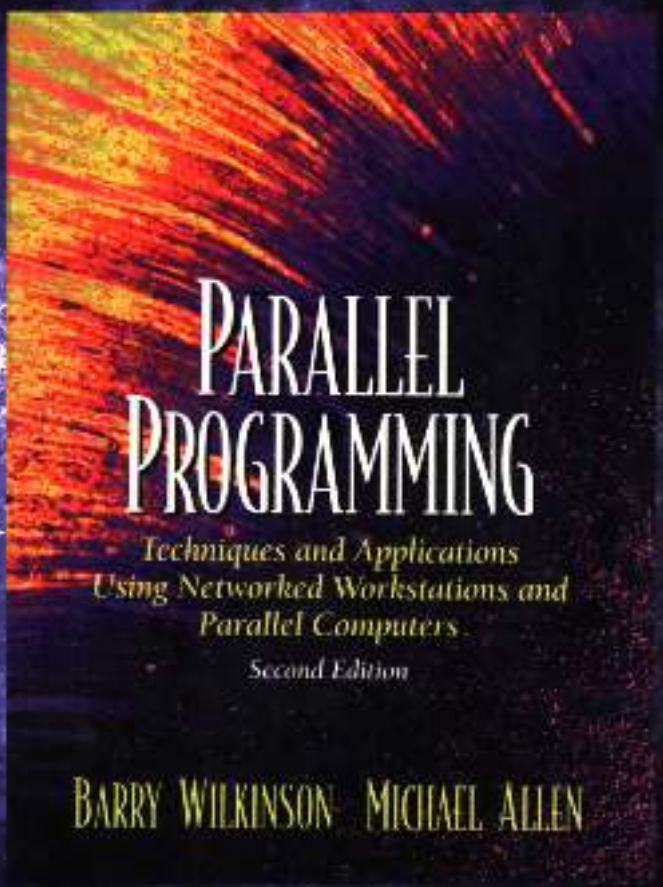
计 算 机 科 学 丛 书

第2版

并行程序设计

(美) Barry Wilkinson 北卡罗来纳大学 Michael Allen 著
陆鑫达 上海交通大学 等译



Parallel Programming
Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition

机械工业出版社
China Machine Press

本教材的编写和出版得到了美国国家科学基金会的大力支持，并经过美国许多大学的授课检验。本书覆盖并行程序设计的众多技术，将重点放在使用免费获得的并行软件工具在联网计算机上执行的并行程序上，所涉及的应用是独立于系统的。从顺序程序设计开始进行自然扩充，介绍并行程序设计技术，拓展了消息传递并行程序设计的基本技术，并探究了在数值和非数值领域特定问题的算法。学习本书并不需要并行程序设计的任何预备知识，读者只需具有C程序设计知识。

本书特点

- 使用MPI伪代码描述算法，有助于在不同的程序设计工具上实现。
- 详细介绍共享存储器程序设计，包括Pthread和OpenMP，辅助学生完成共享存储器程序设计的课外作业。
- 每章末尾包含大量习题，其中“现实生活习题”非常有趣，既可增强学习兴趣又能提高并行程序设计技巧，并且这些习题的求解无需专门的数学知识。
- 配合一个综合的教师指导网站，其中包括：实例、课外作业以及使用MPI软件的辅导材料等，网址为：www.cs.uncc.edu/par_prog。

作者简介



Barry Wilkinson 是美国北卡罗来纳大学夏洛特分校计算机科学系教授，并在西卡罗来纳大学任教。他于1974年在英格兰的曼彻斯特大学（计算机科学系）获得博士学位。自1983年起，他一直是IEEE的资深会员，并在2001年由于从事IEEE机群计算特别工作组（TFCC）的教育计划工作而荣获IEEE计算机学会的感谢证书。他撰写了多部专著，还在主要的计算机刊物上发表过许多论文。



Michael Allen 是北卡罗来纳大学夏洛特分校计算机科学系教授。于1968年在纽约州立大学布法罗分校获得博士学位。从1985到1987年，他在DataSpan公司任董事长和主席。他还曾经在Eastman Kodak, Sylvania Electronics, 宾夕法尼亚州的Bell, Wachovia Bank以及许多其他公司从事电子设计和软件系统开发工作。

www.PearsonEd.com

ISBN 7-111-16260-9

9 787111 162605

华章图书

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037

读者服务热线：(010)68995259, 68995264

读者服务信箱：hzedu@hzbook.com

ISBN 7-111-16260-9/TP · 4232

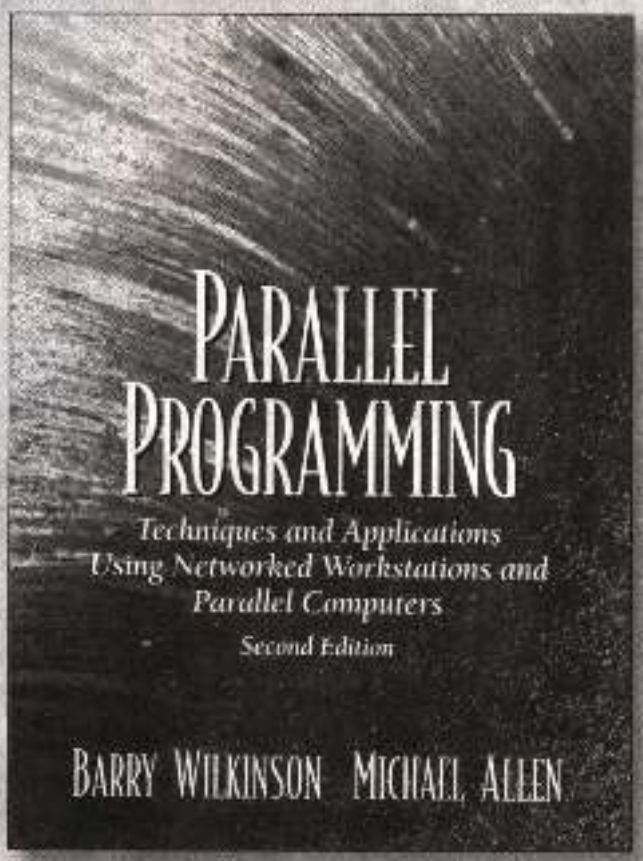
定价：48.00 元

计 算 机 科 学 丛 书

第2版

并行程序设计

(美) Barry Wilkinson Michael Allen 著
北卡罗来纳大学
陆鑫达 等译
上海交通大学



Parallel Programming
Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition

机械工业出版社
China Machine Press

本书系统介绍并行程序设计原理及应用。除介绍常用的一些算法范例，包括分治、流水、同步计算、主从及工作池，还介绍了一些常用的经典数值和非数值算法，如排序、矩阵相乘、线性方程组求解、图像处理中的预处理和相应的变换、搜索和优化（包括遗传算法）等。第2版新增了机群计算和使用机群的内容，对如何打造专用和通用的机群以及设置相应的程序设计环境做了较为详尽的介绍。章后包含大量习题，其中“现实生活习题”非常实用，既可增强学习兴趣，又可提高并行程序设计技巧。

本书可作为高等院校计算机专业高年级本科生或研究生的教材，对从事高性能计算的科技工作者也是一本很有价值的参考书。.

Simplified Chinese edition copyright © 2005 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Parallel Programming : Techniques and Applications Using Networked Workstations and Parallel Computers , Second Edition* by Barry Wilkinson and Michael Allen , Copyright 2005.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall .

本书封面贴有Pearson Education (培生教育出版集团) 激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2004-4098

图书在版编目（CIP）数据

并行程序设计（第2版）/（美）威尔金森（Wilkinson, B.），（美）阿兰（Allen, M.）著；陆鑫达等译. -北京：机械工业出版社，2005.5

（计算机科学丛书）

书名原文：Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition

ISBN 7-111-16260-9

Ⅰ.并… Ⅱ.①威… ②阿… ③陆… Ⅲ.并行程序-程序设计 Ⅳ.TP311.11

中国版本图书馆CIP数据核字（2005）第020161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：隋 曦

北京中兴印刷有限公司印刷·新华书店北京发行所发行

2005年5月第2版第1次印刷

787mm × 1092mm 1/16 · 23.25印张

印数：0 001-4000册

定价：48.00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、从业人员较少的现状下，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图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray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，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，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，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步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步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界

名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

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，但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。教材的出版只是我们的后续服务的起点。华章公司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：

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

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周立柱	周克定	周傲英	孟小峰	岳丽华
范　明	郑国梁	施伯乐	钟玉琢	唐世渭
袁崇义	高传善	梅　宏	程　旭	程时端
谢希仁	裘宗燕	戴　葵		

第2版译者序

自本书第1版出版以来，联网计算机，特别是以消息传递为基础的联网机群的程序设计技术有了长足的进步，机群计算技术已成为经济有效地获取高性能计算能力的主流技术，相应的MPI程序设计环境也已逐步成为主流的消息传递程序设计环境，而PVM程序设计环境则逐渐淡出。

共享存储器程序设计在经历了较长时间的发展后也终于有了像OpenMP那样为各界共同接受的共享存储器程序设计标准，它在以SMP作为基本结点的高性能并行机系统的并行程序设计中起着关键的作用。将消息传递程序设计和共享存储器程序设计两者有机结合的分布式共享存储器程序设计将是未来机群程序设计的发展方向。本书第2版的内容变动正是对这些发展趋势的及时反映。

长期以来，计算机体系结构课程缺少相应的实验素材，第2版新增了机群计算和使用计算机机群的内容，对如何打造专用和通用的计算机机群以及设置相应的程序设计环境做了较为详尽的介绍，译者相信这将为开设网络环境下的计算机体系结构实验课程打下良好的基础。

本书第1版的翻译工作由陆鑫达教授负责和组织，并翻译了目录、序、第1～3章，以及第11和12章，曾志勇翻译了第5章，张建翻译了第7章，支小莉翻译了第6章，徐蔚文翻译了第10章，钟嵘翻译了第8章，吴欣翻译了第9章，汤勇平翻译了第4章。译稿全文由陆鑫达教授作了校对。

第2版的翻译和审校由陆鑫达教授负责，翻译了新增的内容（1.2节提高计算速度的潜力，1.4节机群计算，2.2节使用计算机机群，6.4节部分同步方法，8.4节并行程序设计语言和构造，8.5节OpenMP，8.6节性能问题，第9章分布式共享存储器系统及其程序设计，10.3节在专用网络上排序，10.4节其他排序算法，以及附录C——OpenMP命令、库函数以及环境变量及各章新增的习题等），重译了各章中改写的内容，并仔细校对了删除的内容。

本书是有关使用联网计算机进行并行计算的一本很实用的教科书，是两位教授多年教学和科研工作的结晶。作者用此教材引导美国大学一年级学生进行并行程序设计实践，具有超前意识，为此获得美国国家科学基金会的资助，使此书得以出版。我们翻译此书的宗旨是想促使并行/分布计算，特别是使用联网计算机的并行/分布计算，在中国的普及和发展，尤其是在高等院校中的普及和发展。

本书除介绍了常用的一些算法范例，包括分治、流水、同步计算、主从及工作池等，还介绍了一些常用的经典数值和非数值算法，如排序、矩阵相乘、线性方程组求解、图像处理中的预处理和相应的变换、搜索和优化（包括遗传算法）等，这是本书的一大特色。读者掌握了这些算法范例和经典算法后就可为并行程序设计打下良好基础。书中每一章后面均附有习题，除与每章内容有关的习题之外，还有不少习题源自现实生活，既富有启发性，又具有趣味性，应该说这是本书的另一大特色。学习本书的唯一要求是具有C语言的程序设计知识，对并行程序设计知识没有要求。对算法的描述采用MPI伪代码，并允许使用不同的程序设计工具加以实现。

本书可作为计算机专业本科生高年级选修课程的教材，也可作为研究生的学位或非学位课程的教材，对在其他各个专业领域中从事高性能计算的科技工作者也是一本很有价值的参考书。

值本书第2版出版之际，译者谨向机械工业出版社华章公司的策划和编辑人员表示深切的谢意。

由于翻译时间较仓促，再加有的英语术语国内没有统一的译法，故翻译中的错误或不妥之处在所难免，敬请读者不吝指正。

译者简介

陆鑫达 现为上海交通大学计算机科学与工程系教授、博士生导师，网格计算中心副主任及高性能计算研究室主任，中国计算机学会体系结构专委会副主任，中国计算机学会开放式分布与并行计算专委会高级委员，中国计算机学会高级会员，贵州大学兼职教授。1961年和1964年分别获哈尔滨工业大学计算机专业学士和硕士学位。1979～1981年为英国纽卡斯尔大学计算机系访问学者，主要从事高度并行计算技术、VLSI芯片设计技术和处理机互联等技术研究。1987～1990年为德国GMD-FIRST 柏林计算机研究所和柏林工业大学（TUB）的客座首席科学家，主要从事新型数据结构高性能计算系统研究，并负责国家自然科学基金会重大项目中的中德国际合作项目。曾任《中国大百科全书》“自动控制和系统工程”卷“信息处理”分卷副主编。主编教材《计算机系统结构》（高等教育出版社1996年3月）。教材译著：《可扩展并行计算：技术、结构与编程》（机械工业出版社，2000年5月），《并行程序设计》（机械工业出版社，2002年1月第1版）。

目前的主要研究方向为高性能计算及应用、网格计算和服务计算、异构计算、网络计算及其编程环境，机群计算（包括体系结构及中间件）、遗传和进化算法在映射和调度中的应用、分布计算及移动计算等。

前　言

本教科书的目的是介绍并行程序设计技术。并行程序设计使用多计算机或多个内部处理器的计算机来求解问题，它比使用单台计算机的计算速度要快得多。它也为求解更大规模的问题提供了机会，这些问题需要更多的计算步或更大存储容量需求，之所以能满足后一要求，是因为多计算机和多处理机系统通常比单计算机有更大的总存储容量。在本书中，我们讨论的重点是使用多计算机进行并行程序设计，它们之间的通信是通过发送消息来完成的，从而出现了消息传递并行程序设计的术语。我们所使用的计算机可以是不同的类型（PC、SUN、SGI等），但它们必须由网络进行互联，此外还必须有一个软件环境以在计算机间进行消息传递。适当联网的计算机（或者在网络中或者具有互联的能力的计算机）可广泛地作为学生的基本计算平台，以便避免使用特殊设计的多处理机系统。为实现消息传递并行程序设计，可使用几种软件工具，特别是几个MPI的实现方案，它们均可免费得到。这些软件也可在特殊设计的多处理机系统上使用，如果这些系统可以使用的话。我们所讨论的技术和应用是独立于系统的，因本书非常实用。

第2版　自从本书第1版出版后，使用互联的计算机作为高性能计算平台已很普遍，并用术语“机群计算”来描述这类计算。通常在机群中使用的计算机是“商品”计算机，即在家庭和办公室中使用的低价个人计算机。虽然本教科书的重点在于使用多计算机和多处理机作为高性能计算这一宗旨没有改变，但我们对第1章做了修订以反映这种商品机群的趋势，并远离专门设计的、自含的多处理机。在第1版中，我们叙述了PVM 和MPI，并为它们各提供了一个附录，但是一般在课堂上只使用其中一个。在第2版中，我们从教科书中删去了有关PVM的详尽细节，因为现在的MPI是一个被广泛接受的标准，并提供了更为强大的机制。如果读者愿意，仍可使用PVM，我们在网页中仍提供对它的支持。

消息传递程序设计有某些缺点，特别是要求程序员显式地说明消息传递应在程序的何处、何时出现以及要发送什么。数据不得不通过相当慢的消息发送给需要该数据的计算机。有些学者将这种类型的程序设计比作汇编语言的程序设计，即使用计算机内部语言编程，这是一种非常低级和麻烦的程序设计方法，除非在非常特殊的情况下，一般不采用这种方法来编程。另一种程序设计模型是共享存储器模型。在第1版中，共享存储器程序设计适用于具有多个内部处理器和一个公共共享存储器的计算机。这种共享存储器多处理机现在已变得非常经济有效和普遍，特别是双处理器和四处理器系统。线程的程序设计是用Pthread描述的。在第2版中，保留了共享存储器的程序设计，并增加了许多新的内容，包括共享存储器的编程性能和OpenMP，OpenMP是比Pthread有更高层次的、基于线程的共享存储器程序设计的标准。任何实践并行程序设计的宽范围的课程将包括共享存储器的程序设计，因而具有OpenMP程序设计经验是非常合意的。在新的附录中，增加了OpenMP。对教育机构来讲，只需用低廉的价格就可获得OpenMP编译器。

使用机群是重点，因此增加了有关在机群上进行共享存储器程序设计的新的一章。使用合适的分布式共享存储器（DSM）软件就可在机群上实现共享存储器模型。分布式共享存储器程序设计试图获得机群可扩展性的优点和共享存储器的长处。提供DSM环境的软件可免费得到，而我们将展示学生能编写他们自己的DSM系统（已经有若干学生做到了这一点）。应该

指出的是DSM系统存在一些性能方面的问题。不能期待软件DSM的性能会如同在一个共享存储器多处理机上真正的共享存储器程序设计那样好。但一个大型的、可扩展的共享存储器多处理机比起商品机群来要昂贵得多。

第2版中的其他变化是有关机群程序设计的。在第6章中增加了部分同步计算的内容，这在机群中是特别重要的，因为在机群中同步很花时间，因此应尽量避免。我们对第10章的排序算法内容作了修订和补充，包括适用于机群的其他排序算法。我们还在本书的第一部分增加了对算法的分析，包括计算/通信比，因为这对消息传递计算非常重要。此外，还增加了一些习题。为保持合理的篇幅，在附录中删去了并行计算模型。

曾将本教科书的第1版定位为主要用作大学本科生的并行程序设计课程的教科书。但我们发现某些单位也将它作为研究生课程的教科书。我们也已将它既作为高年级本科的也作为研究生课程的教科书，本书适合作为研究生的初级课程。作为研究生的课程，应包括更先进的内容，例如DSM的实现和快速傅里叶变换，并选择有更高要求的程序设计研究作业。

内容编排　如第1版一样，本教科书分为两部分。第一部分现在包括第1章到第9章，而第二部分现在包括第10章到第13章。在第一部分中，将讨论并行程序设计的基本技术。在第1章中，对并行计算机的概念现在更注重机群的介绍。第2章概述消息传递例程，特别是MPI软件。从理论和实践两方面讨论如何评估消息传递程序的性能。第3章叙述理想的并行化问题，即易并行计算，在这种计算中，问题可被分割成独立部分。事实上，许多重要应用都可以这种方式加以并行化。第4、5、6、7各章叙述了各种程序设计策略（分割和分治、流水线、同步计算、异步计算和负载平衡）。第一部分的这几章涉及了并行程序设计的所有基本内容，并通过对消息传递和对简单问题的求解来示范技术，而这些技术的本身可在许多场合用来求解问题。通常我们首先给出顺序的示范代码，然后再给出并行的伪代码。一般来讲，基本算法在本质上是并行的，而顺序版本“不自然”地使用循环将其串行化。当然，某些算法为了高效地进行并行求解必须进行重构，但这种重构并不是立即就能看清的。第8章叙述了共享存储器程序设计，其中包括了广泛采用的IEEE标准系统Pthread以及OpenMP。在该章中，还增加了有关定时和性能问题的全新的一节内容。新增的一章是有关分布式共享存储器程序设计的，它被放在共享存储器这一章之后，以此结束第一部分，在此之后的各章已重新编号。

许多并行计算问题具有一些专门开发的算法，在第二部分中所研究的专用算法涉及非数值和数值范畴。在学习第二部分时，将需要一些数学概念，如矩阵。在第二部分中涉及的主题包括排序（第10章），数值算法、矩阵乘法、线性方程组、偏微分方程（第11章），图像处理（第12章）以及搜索和优化（第13章）。图像处理特别适合于并行化，因此在第二部分中将其作为饶有兴趣的一个应用，专门用一章加以介绍，这种应用对许多研究项目有很大的潜在参考价值。在图像处理这一章中，还讨论了所涉及的快速傅里叶变换，这一重要变换还被用于许多其他领域中，包括信号处理和语音识别。

在每章的末尾列出了许多“现实生活”习题，其中绝大部分源自实际生活。这些习题的求解无需专门的数学知识，它是本书的特色之一；这些习题有助于开发使用并行程序设计技术的技巧，而不是简单地学习如何去求解像数的排序或矩阵相乘那样的专门问题。

预备知识　学习第一部分所需的准备是有关顺序程序设计的知识，这可通过使用C语言学到。教科书中的并行伪代码是用类似C的赋值语句和控制流语句编写的。但是，如果学生只有Java知识应不难理解伪代码，因为这些语句的语法与Java是类似的。在掌握了基本的顺序程序设计后可立即进行第一部分的学习。在第一部分中的许多作业无需专门的数学知识就可尝试求解。如果作业需使用MPI，则可用具有MPI消息传递库调用的C或C++语言来编写程序。在

附录A中，对如何进行具体的库调用做了说明。也可以使用Java，虽然学生只具有Java知识，他们应没有任何困难用C/C++来完成程序设计作业。

在第二部分中，学习排序这一章时，假定学生已在学习数据结构或顺序程序设计课程时掌握了顺序排序的知识。学数值算法这一章时，学生应具有相应的数学背景，这些背景是高年级计算机科学或工程系的学生应该掌握的。

课程结构 教师可灵活地介绍本书的内容，没有必要包括所有内容。事实上，用一个学期是不可能讲完整本书的内容的。宜从第一部分中选择一部分主题作为一般顺序程序设计课程的补充。我们以这种方式指导一年级学生进行并行程序设计。在这种环境下，本书便是对顺序程序设计教科书的补充。整个第一部分及第二部分的一部分合在一起，可作为高年级本科生或研究生初期的并行程序设计/计算课程，我们以这种方式使用本书。

主页 为本书已开发了Web网站以帮助学生和教师。网址为www.cs.uncc.edu/par_prog。在该网站中还有许多Web页面帮助学生学习如何对并行程序进行编译和运行，网站上还提供了一些示范程序，作为简单的初期作业以检查软件环境。在准备本书第2版时，对整个网站做了重新设计，包括使用导航按钮对学生进行逐步指导，还提供了DSM程序设计的细节。对指导教师提供了新的教师手册，并给出了MPI的解答。最初的解答手册包含PVM的解答，仍可使用。可从作者处得到电子版的解答手册。从主页上还可得到大量的幻灯片。

致谢 本教科书的第1版是美国国家科学基金会资助作者在北卡罗来纳大学夏洛特分校的向一年级学生介绍并行程序设计的直接结果。㊀ 没有当时的国家科学基金会计划主任、已故的M.Mulder博士的支持，我们不可能去追求在教科书中所提及的那些想法。许多研究生参加了这一独创的研究项目。Uday Kamath先生创作了最初的习题解答手册。

我们要向部门的系统管理员James Robinson致谢，他建成了我们本地的工作站机群，如果没有该机群我们不可能完成这一著作。我们还要感谢UNC（北卡罗来纳大学）夏洛特分校的许多学生，多年来他们选了我们的课程并帮助我们改进了教材。其中包括了“远程课程”，在这些课程中，本书第1版的材料成为以独特方式实验的课堂。这些远程课程除了在UNC夏洛特分校外，还向几所北卡罗来纳大学得到了普及，其中包括北卡罗来纳大学阿什维尔分校、北卡罗来纳大学格林斯伯勒分校、北卡罗来纳大学威尔明顿分校和北卡罗来纳州立大学。北卡罗来纳州立大学的Mladen Vouk教授除了以客座专家身份授课外，还设计了一个给人印象深刻的Web页面，其中包括了我们讲课的“实况录音”和“自动翻页”的幻灯片。（这些授课情况可通过链接我们的主页看到。）杜克大学的John Board教授以及北卡罗来纳大学查珀尔希尔分校的Jan Prins教授也欣然做了客座专家的讲演，承蒙Raul Gallard教授的友好邀请，我们还在阿根廷的圣路易斯国立大学讲授了基于本书素材的并行程序设计课程。

是国家科学基金会继续支持我们对机群计算的研究工作使我们得以撰写本书的第2版。一项国家科学基金会的资助项目鼓励我们开发分布式共享存储器的工具和教学素材。㊁ 本书第9章分布式共享存储器程序设计叙述了这一研究工作。此后，国家科学基金会又资助我们一个项目，于2001年7月在UNC夏洛特分校组织一个三天的有关机群计算教学的专题学术讨论会，㊂ 这使我们进一步地精练了本书的素材。我们要向国家科学基金会计划主任Andrew Bernat博士对我们的继续支持表示感谢。是他提议在夏洛特分校举行机群计算的专题学术讨论会。参加这次专题学术讨论会的共有来自美国各地的18位教学人员。它导致了另一个三天的、

㊀ 美国国家科学基金会项目“将并行程序设计技术引入大学一年级课程”，项目编号DUE 9554975。

㊁ 美国国家科学基金会项目“在工作站机群上的并行程序设计”，项目编号DUE 995030。

㊂ 美国国家科学基金会项目对机群计算专题学术讨论会的补充资助，项目编号DUE 0119508。

于2001年12月在印度阿哈玛德巴德的Gujarat大学举行的有关机群计算教学的专题学术讨论会，这一次是受IEEE机群计算特别工作组（Task Force on Cluster Computing，TFCC）及印度IEEE计算机学会的邀请。这次专题学术讨论会约有40位教学人员参加。我们要衷心感谢潜心于该专题学术讨论会的人们，特别是Rajkumar Buyya先生，IEEE机群计算特别工作组的主席，是他提议召开这次专题学术讨论会。我们也非常感谢Prentice Hall出版社为每位出席代表免费提供了本教科书。

我们仍继续与在UNC夏洛特分校的和在别处的（包括波士顿的马萨诸塞大学，在休假离校时）学生一起检测本书素材。在编写第2版时，好几位UNC夏洛特分校的本科生与我们一起从事该项目。第2版的新Web页面是由Omar Lahbabi开发的，并由 Sari Ansari进一步改进，两位都是本科生。MPI的解答手册是由Thad Drum和Gabriel Medin完成的，他们两位也是UNC夏洛特分校的本科生。

我们特别感谢Prentice Hall出版社高级组稿编辑Petra Rector，他在出版本书第2版的整个过程中给予了我们支持。评阅人对我们提供了非常有用的指点，特别是一位匿名评阅人，是他直言不讳的评论使我们重审本书的许多方面，无疑地改进了本书的素材。

最后，我们要感谢许多与本书第1版有关的人们，他们向我们提供了修正和建议。我们保留了一份在线勘误表，这在本书再版时将非常有用。已在第2版中对所有在第1版中发现的错误做了修正。也将为第2版保留一份在线勘误表，它与我们的主页相连。我们将永远欢迎对本书进行评论和提供修改意见。请将你们的评论和修改意见通过下列电子邮件地址发给我们：wilkinson @ email.wcu.edu (Barry Wilkinson)或cma@uncc.edu (Michael Allen)。

Barry Wilkinson
西卡罗来纳大学
Michael Allen
北卡罗来纳大学，夏洛特分校

作者简介

Barry Wilkinson是北卡罗来纳大学夏洛特分校计算机科学系教授，并在西卡罗来纳大学任教。在此之前他曾在英格兰布赖顿大学（1984～1987）、纽约州立大学纽珀尔兹学院（1983～1984）、威尔士加的夫大学学院（1976～1983）以及英格兰阿斯顿大学（1973～1976）任职。从1969到1970年，他曾在Ferranti有限公司从事过程控制计算机系统的工作。他是《Computer Peripherals》（与D. Horrocks、Hodder和Stoughton合作,1980, 第2版1987年）、《Digital System Design》（Prentice Hall，1987，第2版1992年）、《Computer Architecture Design and Performance》（Prentice Hall，1991，第2版1996年）和《The Essence of Digital Design》（Prentice Hall，1997）的作者。除了上述的著作之外，他在主要的计算机刊物上发表了许多论文。1969年他在英国的索尔福德大学获得电气工程的学士学位（优等成绩），1971和1974年在英格兰的曼彻斯特大学（计算机科学系）分别获得硕士和博士学位。自1983年起，他一直是IEEE的资深会员，并在2001年由于从事IEEE机群计算特别工作组（TFCC）的教育计划工作而荣获IEEE计算机科学学会的感谢证书。

Michael Allen是北卡罗来纳大学夏洛特分校计算机科学系教授。在此之前他曾是北卡罗来纳大学夏洛特分校电气工程系的副教授和教授（1974～1985），并曾是纽约州立大学布法罗分校的讲师和助理教授（1968～1974）。在1985到1987年他离开了北卡罗来纳大学夏洛特分校，在DataSpan公司任董事长和主席。他还曾经在Eastman Kodak、Sylvania Electronics、宾夕法尼亚州的Bell、Wachovia Bank以及许多其他公司中从事电子设计和软件系统开发工作。他于1964年和1965年分别在卡内基-梅隆大学获得电气工程的学士和硕士学位，并于1968年在纽约州立大学布法罗分校获得博士学位。

目　录

出版者的话
专家指导委员会
第2版译者序
译者简介
前言
作者简介

第一部分　基本技术

第1章　并行计算机 ……2
1.1　对计算速度的需求 ……2
1.2　提高计算速度的潜力 ……4
1.2.1　加速系数 ……4
1.2.2　什么是最大的加速比 ……5
1.2.3　消息传递计算 ……9
1.3　并行计算机的类型 ……9
1.3.1　共享存储器多处理机系统 ……10
1.3.2　消息传递多计算机 ……11
1.3.3　分布式共享存储器 ……17
1.3.4　MIMD和SIMD的分类 ……17
1.4　机群计算 ……18
1.4.1　以互联计算机作为计算平台 ……18
1.4.2　机群的配置 ……23
1.4.3　打造“Beowulf风格”的专用机群 ……26
1.5　小结 ……27
推荐读物 ……27
参考文献 ……28
习题 ……30
第2章　消息传递计算 ……31
2.1　消息传递程序设计基础 ……31
2.1.1　编程的选择 ……31
2.1.2　进程的创建 ……31
2.1.3　消息传递例程 ……33
2.2　使用计算机机群 ……37
2.2.1　软件工具 ……37
2.2.2　MPI ……37
2.2.3　伪代码构造 ……44
2.3　并行程序的评估 ……45
2.3.1　并行执行时间方程式 ……45
2.3.2　时间复杂性 ……48
2.3.3　对渐近分析的评注 ……50
2.3.4　广播/集中的通信时间 ……50
2.4　用经验方法进行并行程序的调试和评估 ……51
2.4.1　低层调试 ……52
2.4.2　可视化工具 ……52
2.4.3　调试策略 ……53
2.4.4　评估程序 ……53
2.4.5　对优化并行代码的评注 ……55
2.5　小结 ……55
推荐读物 ……55
参考文献 ……56
习题 ……57
第3章　易并行计算 ……59
3.1　理想的并行计算 ……59
3.2　易并行计算举例 ……60
3.2.1　图像的几何转换 ……60
3.2.2　曼德勃罗特集 ……64
3.2.3　蒙特卡罗法 ……69
3.3　小结 ……73
推荐读物 ……73
参考文献 ……73
习题 ……74
第4章　划分和分治策略 ……79
4.1　划分 ……79
4.1.1　划分策略 ……79
4.1.2　分治 ……82
4.1.3　*M*路分治 ……86
4.2　分治技术举例 ……87
4.2.1　使用桶排序法排序 ……87

4.2.2 数值积分 ……91
4.2.3 N体问题 ……93
4.3 小结 ……96
推荐读物 ……97
参考文献 ……97
习题 ……98
第5章 流水线计算 ……104
5.1 流水线技术 ……104
5.2 流水线应用的计算平台 ……107
5.3 流水线程序举例 ……107
5.3.1 数字相加 ……108
5.3.2 数的排序 ……110
5.3.3 生成质数 ……112
5.3.4 线性方程组求解——特殊个例 ……114
5.4 小结 ……117
推荐读物 ……117
参考文献 ……117
习题 ……117
第6章 同步计算 ……122
6.1 同步 ……122
6.1.1 障栅 ……122
6.1.2 计数器实现 ……123
6.1.3 树实现 ……124
6.1.4 蝶形障栅 ……125
6.1.5 局部同步 ……126
6.1.6 死锁 ……126
6.2 同步计算 ……127
6.2.1 数据并行计算 ……127
6.2.2 同步迭代 ……129
6.3 同步迭代程序举例 ……130
6.3.1 用迭代法解线性方程组 ……130
6.3.2 热分布问题 ……135
6.3.3 细胞自动机 ……142
6.4 部分同步方法 ……143
6.5 小结 ……144
推荐读物 ……144
参考文献 ……144
习题 ……145
第7章 负载平衡与终止检测 ……151
7.1 负载平衡 ……151
7.2 动态负载平衡 ……152
7.2.1 集中式动态负载平衡 ……152
7.2.2 分散式动态负载平衡 ……153
7.2.3 使用线形结构的负载平衡 ……155
7.3 分布式终止检测算法 ……157
7.3.1 终止条件 ……157
7.3.2 使用确认消息实现终止 ……158
7.3.3 环形终止算法 ……158
7.3.4 固定能量分布式终止算法 ……160
7.4 程序举例 ……160
7.4.1 最短路径问题 ……160
7.4.2 图的表示 ……161
7.4.3 图的搜索 ……162
7.5 小结 ……166
推荐读物 ……166
参考文献 ……167
习题 ……168
第8章 共享存储器程序设计 ……172
8.1 共享存储器多处理机 ……172
8.2 说明并行性的构造 ……173
8.2.1 创建并发进程 ……173
8.2.2 线程 ……175
8.3 共享数据 ……178
8.3.1 创建共享数据 ……179
8.3.2 访问共享数据 ……179
8.4 并行程序设计语言和构造 ……185
8.4.1 并行语言 ……185
8.4.2 并行语言构造 ……186
8.4.3 相关性分析 ……187
8.5 OpenMP ……189
8.6 性能问题 ……193
8.6.1 共享数据的访问 ……193
8.6.2 共享存储器的同步 ……195
8.6.3 顺序一致性 ……196
8.7 程序举例 ……199
8.7.1 使用UNIX进程的举例 ……199
8.7.2 使用Pthread的举例 ……201
8.7.3 使用Java的举例 ……203
8.8 小结 ……204
推荐读物 ……205

参考文献 ……205
习题 ……206
第9章 分布式共享存储器系统及其程序设计 ……211
9.1 分布式共享存储器 ……211
9.2 分布式共享存储器的实现 ……212
9.2.1 软件DSM系统 ……212
9.2.2 DSM系统的硬件实现 ……213
9.2.3 对共享数据的管理 ……214
9.2.4 基于页面系统的多阅读器/单写入器策略 ……214
9.3 在DSM系统中实现一致性存储器 ……214
9.4 分布式共享存储器的程序设计原语 …216
9.4.1 进程的创建 ……216
9.4.2 共享数据的创建 ……216
9.4.3 共享数据的访问 ……217
9.4.4 同步访问 ……217
9.4.5 改进性能的要点 ……217
9.5 分布式共享存储器的程序设计 ……219
9.6 实现一个简易的DSM系统 ……219
9.6.1 使用类和方法作为用户接口 ……220
9.6.2 基本的共享变量实现 ……220
9.6.3 数据组的重叠 ……222
9.7 小结 ……224
推荐读物 ……224
参考文献 ……224
习题 ……225

第二部分 算法和应用

第10章 排序算法 ……230
10.1 概述 ……230
10.1.1 排序 ……230
10.1.2 可能的加速比 ……230
10.2 比较和交换排序算法 ……231
10.2.1 比较和交换 ……231
10.2.2 冒泡排序与奇偶互换排序 ……233
10.2.3 归并排序 ……236
10.2.4 快速排序 ……237
10.2.5 奇偶归并排序 ……239
10.2.6 双调谐归并排序 ……240
10.3 在专用网络上排序 ……243
10.3.1 二维排序 ……243
10.3.2 在超立方体上进行快速排序 ……244
10.4 其他排序算法 ……247
10.4.1 秩排序 ……248
10.4.2 计数排序 ……249
10.4.3 基数排序 ……250
10.4.4 采样排序 ……252
10.4.5 在机群上实现排序算法 ……253
10.5 小结 ……253
推荐读物 ……254
参考文献 ……254
习题 ……255
第11章 数值算法 ……258
11.1 矩阵回顾 ……258
11.1.1 矩阵相加 ……258
11.1.2 矩阵相乘 ……258
11.1.3 矩阵-向量相乘 ……259
11.1.4 矩阵与线性方程组的关系 ……259
11.2 矩阵乘法的实现 ……259
11.2.1 算法 ……259
11.2.2 直接实现 ……260
11.2.3 递归实现 ……262
11.2.4 网格实现 ……263
11.2.5 其他矩阵相乘方法 ……266
11.3 求解线性方程组 ……266
11.3.1 线性方程组 ……266
11.3.2 高斯消去法 ……266
11.3.3 并行实现 ……267
11.4 迭代方法 ……269
11.4.1 雅可比迭代 ……269
11.4.2 快速收敛方法 ……272
11.5 小结 ……274
推荐读物 ……275
参考文献 ……275
习题 ……276
第12章 图像处理 ……279
12.1 低层图像处理 ……279

12.2 点处理 ……280
12.3 直方图 ……281
12.4 平滑、锐化和噪声消减 ……281
12.4.1 平均值 ……281
12.4.2 中值 ……283
12.4.3 加权掩码 ……284
12.5 边缘检测 ……285
12.5.1 梯度和幅度 ……285
12.5.2 边缘检测掩码 ……286
12.6 霍夫变换 ……288
12.7 向频域的变换 ……290
12.7.1 傅里叶级数 ……291
12.7.2 傅里叶变换 ……291
12.7.3 图像处理中的傅里叶变换 ……292
12.7.4 离散傅里叶变换算法的并行化 ……294
12.7.5 快速傅里叶变换 ……296
12.8 小结 ……300
推荐读物 ……300
参考文献 ……300
习题 ……302
第13章 搜索和优化 ……305
13.1 应用和技术 ……305
13.2 分支限界搜索 ……306
13.2.1 顺序分支限界 ……306
13.2.2 并行分支限界 ……307
13.3 遗传算法 ……308
13.3.1 进化算法和遗传算法 ……308
13.3.2 顺序遗传算法 ……310
13.3.3 初始种群 ……310
13.3.4 选择过程 ……312
13.3.5 后代的生成 ……312
13.3.6 变异 ……314
13.3.7 终止条件 ……314
13.3.8 并行遗传算法 ……314
13.4 连续求精 ……317
13.5 爬山法（hill climbing） ……318
13.5.1 银行业务应用问题 ……319
13.5.2 爬山法在金融业务中的应用 ……320
13.5.3 并行化 ……321
13.6 小结 ……321
推荐读物 ……321
参考文献 ……322
习题 ……323
附录A 基本的MPI例程 ……329
附录B 基本的Pthread例程 ……335
附录C OpenMP命令、库函数以及环境变量 ……339
索引 ……347

第一部分 基本技术

第1章 并行计算机
第2章 消息传递计算
第3章 易并行计算
第4章 划分和分治策略
第5章 流水线计算
第6章 同步计算
第7章 负载平衡与终止检测
第8章 共享存储器程序设计
第9章 分布式共享存储器系统及其程序设计

第1章 并行计算机

本章首先叙述对计算机有更强计算能力的需求，以及使用具有多个内部处理器的计算机和多个互联计算机的概念。然后讨论使用多计算机或多处理机以提高执行速度的前景和极限。最后将探讨构成这些系统的各种方法，特别是在机群中使用多计算机的方法，机群已成为进行高性能计算非常经济有效的计算机平台。

1.1 对计算速度的需求

人们要求计算机系统提供更强的计算能力的需求总是不断增长的。需要很高计算速度的领域包括科学和工程问题的数值建模和模拟。这些问题常需要对大量数据进行很多次重复计算以得到有效结果。此外计算必须在“合理”的时间内完成。在制造业领域，工程计算和模拟如果可能的话必须在几秒或几分钟内完成。在设计环境中，如果进行一次模拟需要两星期才能得到结果通常是不可接受的。因为只有模拟完成时间足够短时，设计者方可高效地工作。当系统变得更复杂时，就需要增加更多时间对系统进行模拟。有一些应用问题的计算对时间有特定的期限（最著名的要数气象预报），花两天时间来获取当地第二天精确的天气预报将使得这种预报毫无意义。某些研究领域，如对大型DNA结构建模以及进行全球天气预报均属于*巨大挑战性问题*（grand challenge problem）。所谓巨大挑战性问题是指无法用当今计算机在合理的时间内完成求解的那些问题。

由计算机进行的天气预报（*数值气象预报*）是一个被广泛引用的、需要功能非常强大计算机的一个典型例子。大气建模是通过将大气层划分成三维区域或单元完成的。该模型使用相当复杂的数学方程来估测各种影响。实质上，在某一时间间隔内的每个单元中的条件（温度、压力、湿度、风速和风向等）是通过使用在前一时间间隔中的条件进行计算得到的。每个单元的计算要重复许多次以模拟时间的推移。使此模拟有效的关键点在于必须有足够多的单元数。为预报数天的天气，由于大气层会受远距离事件的影响，因此必须有较大的覆盖范围。假定我们将整个地球大气层分成大小为1英里[⊖] × 1英里 × 1英里的许多单元，布满从地面直至10英里的高度（即10个单元高），粗略地估算约有5×10^8个单元。假设每个单元的计算需要200次浮点运算（当数具有小数部分或是提升为幂时，必须使用这种类型的运算）。则在一个时间步中就必须完成10^{11}次浮点运算。如果我们想预报7天以上的天气，使用的时间间隔为1分钟，此时就需要10^4个时间步和总计10^{15}次浮点运算。那么对一台运算速度为1Gflops（10^9浮点运算/秒）的计算机就将需要10^6秒（即超过10天）才能完成上述计算。要想在5分钟内完成这一计算，我们就需要运算速度为3.4Tflops（3.4×10^{12}浮点运算/秒）的计算机。

另一个需要巨大计算量的应用问题是预测太空中天体的运动。每个天体由于万有引力而互相吸引。这些远距离的力可用简单公式加以计算（参见第4章），而每个天体的运动可以通过计算天体所受合力的计算加以预测。如果共有N个天体，则每个天体将总共需计算$N-1$个力，约N^2次计算。在确定这些天体的新位置后，必须重复这种计算。图1-1中示出了一个快照，是一名大学生求解此问题所得到的结果，由于该问题是作为指定的程序设计习题的，故而只给定了

⊖ 1英里 = 1 609.344m

几个天体。但是实际上可能要考虑具有大量天体的情况。例如，银河系可能有10^{11}个星体，此时就需重复计算10^{22}次。即使使用第4章中所述的仅需$N\log_2 N$次计算的高效近似算法（但每次计算中含有更多的计算量），计算的总量仍异常巨大（$10^{11}\log_2 10^{11}$）。若在单处理器系统上运算将需要很长时间。即使每次计算仅需1微秒（10^{-6}秒，这是非常乐观的估计，因为一次计算中含有多次乘法和除法），则使用N^2算法时，每次迭代就需时10^9年。而用$N\log_2 N$算法时，一次迭代也需几乎一年的时间。N体问题在分子级建模化学和生物学系统时也会出现，并且需要巨大的计算能力。



图1-1 由Scott Linssen（北卡罗来纳大学夏洛特分校本科生）完成的天体物理学N体模拟

全球的天气预报和大量物体的模拟（天体的或是分子的）是需要巨大计算能力的传统应用例子。但是人的本能会不断想像那些超过当今计算机系统能力的新的应用，从而需要比目前可提供的更高速度。近期的一些应用，如虚拟现实，需要很高的计算速度使图像和运动真实、连贯从而获得所需结果。看来不论当前的处理器达到什么样的计算速度，总是会有应用需要更高的计算能力。

传统的计算机只有一个处理器来完成程序中所说明的动作。提高计算速度的一种方法是在一台计算机中用多个处理器（多处理机）协同求解一个问题，这种方法实际上好多年来一直在进行研究；或是使用另一种方法，即用多台计算机协同求解一个问题，不论是哪一种方法，整个的求解问题被分成若干部分，然后每个部分各由一个处理器并行地计算。编写这种形式的程序被称为是*并行程序设计*（parallel programming）。计算平台，即一台*并行计算机*（parallel computer），可以是专门设计的、含有多个处理器的计算机系统或是以某种方式互联的若干台独立的计算机。这种方法将显著地提高性能。基本的想法是，p台处理器/计算机应能提供p倍的单处理器/单计算机速度。不论当前处理器/计算机的速度为多少，可以期待求解问题将以$1/p$时间完成。当然这是一个理想情况，实际上很难达到，这是因为问题经常不能完全分解为各个独立的部分，同时各部分之间必须进行交互，包括计算中的数据传送和同步。尽管如此，仍可达到实质性的性能改进。这取决于欲求解问题和问题中的并行性程度。对并行计算的要求永远不会终止，这一方面是因为单处理器执行速度的不断提升使并行计算机速度越来越快，另一方面总是有一些具有巨大挑战性的问题，这些问题在当前计算机上求解时不可能在合理的时间内完成。

除了可使对现有的问题得到加速求解以外，多计算机/多处理机的使用常常可使一个更大的问题或更精确的问题求解能在合理的时间内完成。例如，许多物理现象的计算涉及到将问题分成为离散的求解点。天气预报计算包含着将大气层分为三维的求解格点。二维和三维的求解格点出现在许多其他应用中。用多计算机或多处理机求解时，常允许在给定的时间内计算更多的求解点，从而可获得更精确的解。一个相关的因素是多计算机常比单计算机有更大的总主存储器容量，从而使需要较大主存储器容量的问题得到解决。

即使一个问题可在合理的时间内完成求解，但当相同问题必须用不同的输入值进行多次求解时，会出现如下特别适用于并行计算机的情况，因为不需要对程序作任何改动，相同程

序的多个实例就可在不同的处理器/计算机上同时执行。模拟实验常属于这种类别。模拟代码只需简单地在不同的计算机上对不同的输入值同时执行。

最后，因特网和万维网的出现为并行计算机产生了一个新的领域。例如，Web服务器常常必须每小时处理来自用户的成千上万个请求。一台多处理机，或当今更可能的是将多台计算机连接起来而构成的“机群”，将用来为客户提供这种服务。此时，各个请求可由不同的处理器或计算机同时提供服务。在线银行业和在线零售商都使用计算机机群为他们的顾客提供服务。

并行计算机不是新设想，事实上这是一个很古老的想法。例如，Gill 在1958年就写出了有关并行程序设计的论文[Gill,1958]。Holland在1959年发表了一篇关于“能同时执行任意多子程序的计算机”的论文[Holland, 1959]。Conway在1963年叙述了并行计算机的设计及其编程[Conway, 1963]。尽管有如此长的历史，但Flynn和Rudd于1996年写道：“对更高性能系统的不断增长的需求……将向我们指明一个简单结论：并行是计算机的未来。”我们完全赞同这个结论。

1.2 提高计算速度的潜力

在以下和以后几章中，进程数或处理器数将用p来表示，而将用术语“多处理机”（multiprocessor）来表示所有多于一个处理器的并行计算机系统。

1.2.1 加速系数

在一个多处理机上研发求解时，也许最感兴趣的一个问题是：采用多处理机求解该问题能快多少？为进行这一比较，我们应使用在单处理器能得到的最好解，即在单处理器系统上所使用的最好顺序算法，与在多处理机上所研究的并行算法进行比较。*加速系数*（speedup factor）$S(p)$[⊖]是对相对性能的衡量，它被定义为：

$$S(p)=\frac{\text{使用单处理器系统执行时间（用最好的顺序算法）}}{\text{使用具有}p\text{个处理器的多处理机的执行所需的时间}}$$

我们用t_s表示在单处理器上运行最好的顺序算法所需执行时间，而用t_p表示在一个多处理机上求解相同问题所需执行时间。那么就有：

6

$$S(p)=\frac{t_s}{t_p}$$

$S(p)$给出了使用多处理机后所能得到的速度增加。应注意的是并行实现的基本算法可能与运行在单处理器系统上的算法不尽相同（并且通常是不同的）。

在理论分析中，加速系数也可根据计算步进行预测：

$$S(p)=\frac{\text{使用单处理器所需的计算步数}}{\text{使有}p\text{个处理器所需的并行计算步数}}$$

对于顺序计算，通常用时间复杂性来比较不同的算法，在第2章中我们将回顾这一点。如我们将见到的那样，可将时间复杂性扩展到并行算法，并应用到加速系数。但是单独考虑计算步可能不一定有用，因为并行实现可能在各并行部分间需要昂贵的通信，它通常比计算步需要更多的时间。在第2章中我们还将对这一点进行进一步的讨论。

对于p个处理器而言，可能获得的最大加速比通常为p（*线性加速*（linear speedup））。当

⊖ 加速系数通常是p和欲处理数据项数n两者的函数，即$S(p, n)$。我们将在后面介绍数据项数。这里的唯一变量是p。

计算可被分成相等持续时间的进程，且一个进程被映射到一个处理器上（且并行求解无附加的开销）时，就可获得加速系数p。

$$S(p) \leqslant \frac{t_s}{t_s/p} = p$$

偶尔会出现超线性加速比（superlinear speedup），即$S(p) > p$，通常这是由于使用的是次优化顺序算法或是某一有利于并行构造的独特体系结构特性，或是算法的不确定性而造成的。一般而言，如果一个纯确定性并行算法能比目前的顺序算法获得大于p倍的加速比，可让该并行算法在单处理器上逐个地对并行部分加以模拟，从而可发现原先的顺序算法并非是优化的。

超线性加速的更为常见的一个原因是由于在多处理机系统中有额外的存储器。例如，假设多处理机系统中每个处理器所含有的主存储器容量与在单处理机系统中所拥有的一样，那么由于多处理机系统中总的主存储器容量要大于单处理机系统中的主存储器容量，因而在任何时刻它总能保存更多的求解问题的数据，这就会导致较少的磁盘和主存储器间的数据交换。

效率　有时需要知道处理器在多长时间内用于有用的计算，这可从（系统）效率（efficiency）推得。效率E被定义为:

$$E = \frac{\text{使用单处理器的执行时间}}{\text{使用多处理机执行时间} \times \text{处理器个数}} = \frac{t_s}{t_p \times p}$$

$$\text{它可写成: } E = \frac{S(p)}{p} \times 100\%$$

其中E以百分比形式表示。例如，如果E = 50%，则平均来讲，处理器仅有一半的时间是用于计算的。100%的最大效率仅当所有处理器在所有时间都用于计算时才会出现，此时的加速系数$S(p)$将为p。

1.2.2 什么是最大的加速比

将有几个因素在并行的版本中以开销形式出现，从而限制了加速:

1）有时并非所有处理器都能完成有用的工作，从而导致一些处理器处于闲置状态。

2）在并行版本中需要在顺序计算中不会出现的额外计算，例如对常数重新进行局部计算。

3）进程间的通信时间。

显然期望计算的某些部分不能分成并发进程而必须对它们串行地求解是合情合理的。在某些时间段内，也许在初始化阶段或是在并发进程被创建以前的一段时间内，只有一个处理器在做有用工作，而在其余的计算中，其他的处理机将运行这些进程。

假定程序中有某些部分只能在单处理器上执行，则理想的情况将是所有可用处理器在余下的时间内同时运行。如果不能分解成并发任务的计算部分所占的比例为 f，并认为将计算分为并发部分时无需开销，则用p个处理器完成此计算所需的时间为: $ft_s + (1-f)t_s/p$，如图1-2所示。图1-2所说明的是这样的一种情况，即计算开始时是一个串行的计算部分，实际上该串行计算部分可能分布在整个计算中。因此加速系数可由下式给定:

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{p}{1+(p-1)f}$$

该方程式即是阿姆达尔定律（Amdahl's law）[Amdahl, 1967]。图1-3中示出了$S(p)$相对于处理器数以及相对于 f 的变化图。从图1-3上的确可看到速度的改进，但是若要使速度有显著的增长就需使能由并发进程执行的计算部分成为整个计算的主要部分。即使使用无限多个

处理器，最大的加速比仍被限制在$1/f$，即，

$$\lim_{p\to\infty} S(p) = \frac{1}{f}$$

例如，当仅有5%的计算为串行时，可获得的最大加速比为20，而与处理器个数无关。在20世
8 纪60年代，阿姆达尔用这一论点提倡使用单处理器系统。当然人们可能以此来进行反驳，即20倍的加速比已经是足够好了。

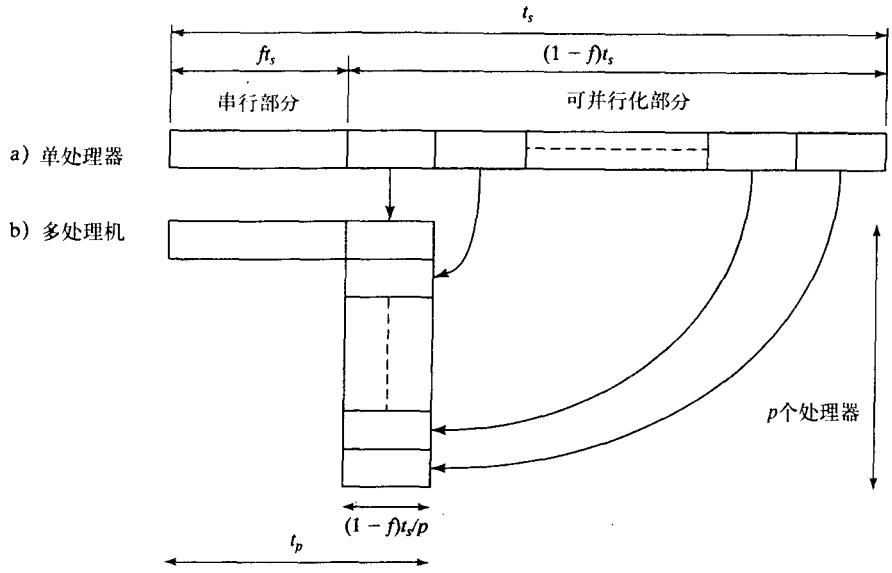


图1-2　顺序问题的并行化—阿姆达尔定律

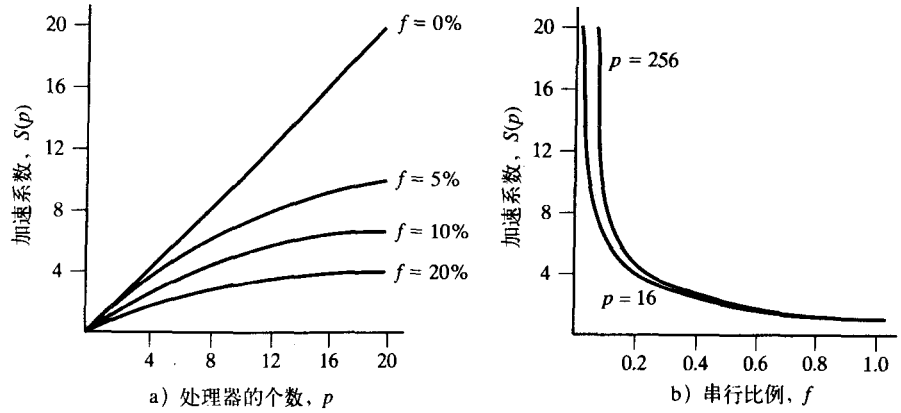
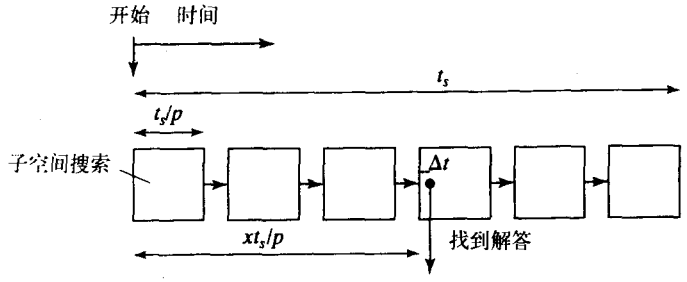


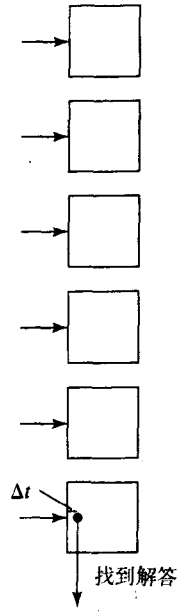
图1-3　a）加速比与处理器数的关系　b）加速比与串行比例f的关系

在某些情况下，速度的改进可达到几个数量级。例如，在搜索算法中能出现超线性加速比。在搜索问题中，求解是通过穷尽查找进行的，假定求解空间在处理器间分割，每一个处
9 理器完成一个独立的搜索。在顺序实现中，对不同的搜索空间是逐个进行搜索的。在并行实现中，逐个搜索可同时进行，且一个处理器几乎可能立即找到求解。在顺序版本中，假定在搜索x个子空间后求解答案在搜索下一个子空间的Δt时间找到。则在前面所搜索过的子空间个数（即x）是不确定的，并将依赖于求解的问题。在并行版本中，解答在时间Δt立即可以找到。

如图1-4所示。



a）顺序搜索每个子空间



b）并行搜索每个子空间

图1-4 超线性加速比

此时的加速比由下式给定:

$$S(p)=\frac{\left(x\times\frac{t_s}{p}\right)+\Delta t}{\Delta t}$$

10

顺序搜索的最坏情况出现在求解是在最后一个子空间的搜索中找到时，此时并行版本可提供最大的好处:

$$S(p)=\frac{\left(\frac{p-1}{p}\right)\times t_s+\Delta t}{\Delta t}\rightarrow\infty \text{ 当}\Delta t\text{趋于}0$$

并行版本可提供最小的好处出现在当顺序搜索在第1个子空间中找到答案时:

$$S(p)=\frac{\Delta t}{\Delta t}=1$$

确切的加速比将依赖于在哪一个子空间中包含有该搜索的解答，但该加速比可能会非常大。

可扩展性（*Scalability*）系统的性能将依赖于系统的规模，如处理器数，一般来讲，系统规模越大性能就越好，但成本也就越高。可扩展性是一个相当不精确的的术语。它用来表示一种允许系统规模增大的硬件设计，这样做的结果是可获得更高的性能，这种可扩展性被描述为是体系结构可扩展性（architecture scalability）或硬件可扩展性（hardware scalability）。可扩展性也可用来指明一个并行算法能容纳更多的数据项而只需增加少量和有限的计算步，这种可扩展性被描述为算法的可扩展性（algorithmic scalability）。

当然，我们希望所有多处理机系统是体系结构可扩展的（制造厂商以这种方式将它们的系统推向市场），但这将严重依赖于系统的设计。通常当我们将更多的处理器加到系统中时，我们必须同时扩展互联网络。这种扩展将导致更大的通信延时和更多的争用，从而使系统的效率E减小。大多数多处理机设计的基本目标是具有可扩展性，这一点已反映在设计了大量的互联网络的这一举动上。

组合的体系结构/算法可扩展性暗示着对于特定的体系结构和算法而言，随着系统规模的增大，它可容纳更大规模的问题。增大系统规模显然意味着要增加处理器的数量，而增大问题的规模则需要作有关说明。直觉上我们会想到以算法中要处理的数据元素总数作为衡量规模大小的标准。但是使问题规模扩大一倍并不一定会使计算步数也增大一倍。这将依赖于要求解的问题本身。例如在第11章中所讨论的两个矩阵的相加会有这样的结果，但将两个矩阵相乘就不会遵从这一规律。因为扩展的矩阵相乘将使计算步数增为四倍。因此，扩大不同的问题意味着会有不同的计算要求。问题规模（problem size）的另一种定义是使问题规模与最佳顺序算法中所需的基本步数等同起来。当然，使用这种定义时，如果我们增加了数据点的数目，我们就将增大问题的规模。

在以后的几章中，除了处理器数p外，我们还将用n来表示问题中的输入数据元素的个数[⊖]。要改进性能，通常可改变p和n两者。改变p即是改变了计算机系统的规模，而改变n意味着改变了问题的规模。通常，增加问题的规模将改善相关性能，因为此时可有更多的并行性。

11

Gustafson根据可扩展性概念提出了一个论点，以证明阿姆达尔定律并非如当初假设的那样会对限止可能的加速比有严重的影响[Gustafson,1988]。Gustafson认为将该思想公式化成方程形式是E.Barsis的功劳。Gustafson提出，实际上一个有更大规模的多处理机通常允许以合理的执行时间求解更大规模的问题。因此，实际上问题规模的大小经常是与可用的处理器数相关的。与其假设问题规模固定，倒不如假设并行执行时间固定。当系统的规模增加时（增加p），则增加问题规模就可保持固定的并行执行时间。当增加问题规模时，Gustafson还认为代码的串行部分一般是固定的，并不会随问题规模的增大而增加。

将固定的并行执行时间作为约束后，所推得的加速系数与阿姆达尔加速系数在数值上是不同的，它被称为比例加速系数（scaled speedup factor）（即加速系数随问题规模增大而增加）。对于Gustafson的比例加速系数，并行执行时间t_p是常数，而不像在阿姆达尔定律中顺序执行时间t_s是常数。我们将使用推导阿姆达尔定律中相同的表示法来推导Gustafson定律，但必须将顺序执行时间t_s分离成顺序和可并行化部分$ft_s + (1-f)\,t_s$，而其中顺序部分ft_s是常数。为推导方便起见，让并行执行时间$t_p = ft_s + (1-f)\,t_s/p = 1$，再对该式作一点小的代数变换，顺序执行时间$t_s$就变为$ft_s + (1-f)\,t_s = p + (1-p)\,ft_s$。这样比例加速系数就将变成：

$$S_s(p) = \frac{ft_s + (1-f)t_s}{ft_s + (1-f)t_s/p} = \frac{p + (1-p)ft_s}{1} = p + (1-p)ft_s$$

⊖　对矩阵，将认为是$n \times n$的矩阵。

该公式被称为*Gustafson*定律（Gustafson's law）。在该方程式中有两点假设：并行执行时间是常数，以及必须顺序执行的部分ft_s也是常数，而不是p的函数。Gustafson的观察结果是：比例加速系数是负斜率为（$1-p$）的直线而并非是如图1-3b所示的急剧下降的曲线。例如，假定串行部分的比例为5%，而处理器数为20，则按此公式计算得到的加速比为0.05 + 0.95 (20) =19.05，而不是按阿姆达尔定律计算得到的10.26（当应注意不同的假设）。Gustafson引用了当用1024个处理器系统求解数值和模拟问题时，实际上可达到加速系数为1021、1020和1016的例子。

除了固定问题规模尺度（阿姆达尔假设）和时间受限尺度（Gustafson假设）外，尺度也可以是存储器受限尺度。在存储器受限的尺度中，问题的规模随可用存储器容量的增大而增大。当处理器数增加时，一般存储器的容量也将按比例增加。这种形式可能导致执行时间的显著增加[Singh, Hennessy, and Gupta, 1993]。 12

1.2.3 消息传递计算

到目前为止的分析，没有考虑消息传递，它在消息传递编程的计算中可能是一个很大的开销。在这种形式的并行程序设计中，在进程间传送消息是为了传递数据和进行同步。因此，

$$t_p = t_{\text{comm}} + t_{\text{comp}}$$

其中t_{comm}是通信时间，而t_{comp}是计算时间。由于我们将问题分成各个并行部分，使得这些并行部分变小，因此一般这些并行部分的计算时间会减少，而这些并行部分间的通信通常会增加（因为有更多的部分通信）。到达某一点后，通信时间将会成为整个执行时间的主要部分，从而使并行执行时间实际上增加。由于处理器间的通信将花费大量时间，因此减少通信开销就变得非常关键。并行求解中的通信部分通常不会出现在顺序求解中，因而被认为是一种开销。

计算/通信比

$$\text{计算/通信比}=\frac{\text{计算时间}}{\text{通信时间}}=\frac{t_{\text{comp}}}{t_{\text{comm}}}$$

可用做衡量指标。在以后的几章中，我们将为算法和问题推导与处理器数（p）和数据元素数目（n）有关的计算时间和通信时间的方程式，以了解可能的潜在加速以及增加p和n的影响。

在实际情况中，对p值，即所使用的系统规模我们没有太多的控制机会（除了我们能将求解问题的多个进程映射到一个处理器上，尽管通常这没有太多好处）。例如，假定对某个p值，一个求解问题需要c_1n计算和c_2n^2通信。显然，当n增加时通信时间的增加将快于计算时间的增加。从计算/通信比（c_1/c_2n）可清楚看到这一点，该比值可通过在时间复杂性的符号中去掉常数后得到（参见第2章）。通常我们希望计算/通信比应尽可能高，即应是n的高增长函数，以使得问题规模的增长仅导致较小的通信时间的增长。当然这是一个与许多因素有关的复杂问题。最后应指出的是，只有通过在一个实际的多处理机系统上执行求解程序才能验证其执行速度，并假定在那时该程序将会被完成。在下一章中，我们将叙述各种衡量实际执行时间的方法。

1.3 并行计算机的类型

在我们相信随着多处理机或多计算机的使用会有加速比的潜在增长后，让我们转而探讨如何构成多处理机或多计算机系统。一台并行计算机可以是一台具有多个内部处理器的单计算机，也可以是多个互联的计算机构成一个一体的高性能计算平台。在本节我们将论述专门 13
设计的并行计算机，而在本章的后面将讨论使用非定制（现货供应）的“商品化”计算机所

构成的机群。术语*并行计算机*（parallel computer）通常是指专门设计的部件。现在主要有两种基本的并行计算机类型:

1）共享存储器多处理机。

2）分布式存储器多计算机。

1.3.1 共享存储器多处理机系统

如图1-5所示，一台通常的计算机是由执行存放于主存储器中程序的处理器组成的。计算机主存储器中的每个单元由称为其*地址*（address）的数字所定位，当地址有b位（二进制位）时，地址从0开始直至2^b-1。

扩展单处理器模型的一个自然方法是使多个处理器连到多个存储器模块，以使得每个处理器能以*共享存储器*（shared memory）配置的形式访问任意一个存储器模块，如图1-6所示。处理器和存储器之间的连接是通过某种*互联网络*（interconnection network）实现的。共享存储器的多处理机系统使用*单一编址空间*（single address space），这意味着在整个主存储器系统中的每一个单元有一个唯一地址，用此地址每个处理器就可以访问该单元。虽然在这些“模型”中没有显示出来，但在实际的系统中都有高速缓冲存储器，稍后我们将对其作进一步讨论。

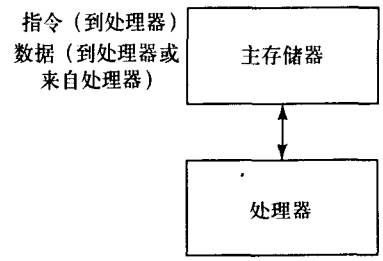


图1-5　由单处理器和主存储器所组成的普通计算机

对共享存储器多处理机进行编程涉及到在共享存储器中存有可由每个处理器执行的代码。每个程序所需的数据也将存于共享存储器中，因此如有需要的话，每个程序可访问所有的数据。可用不同方法由程序员为处理器建立可执行代码和共享数据，但其最终结果必须是使每个处理器在共享存储器执行它自己的程序和代码。（通常是所有处理器执行相同的程序。）

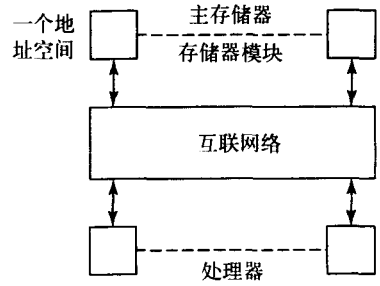


图1-6　传统的共享存储器多处理机模型

程序员为每个处理器生成可执行代码的一种方法是使用一种新的、高级并行程序设计语言，它具有特殊的并行程序设计构造和语句，以声明共享变量和并行代码段。然后由编译器根据程序员对程序的说明来产生最后的可执行代码。但是全新的并行程序设计语言对程序员来讲不是流行的语言。较受欢迎的方法是使用编译器从程序员的“原代码”生成并行代码，此时使用规则的顺序编程语言，再用预处理器命令对程序中的并行性加以说明。该方法的一个例子是OpenMP[Chandra et al., 2001]，它是具有编译器命令和构造的一个工业标准，可融入到C/C++和Fortran语言中。另外也可使用*线程*（threads），线程中含有为各个处理器执行的规整的高级语言代码序列。这些代码序列可用来访问共享单元。已开发了好多年的、但仍令人感兴趣的另一种方法是使用一个规则的顺序编程语言，并修改语法以说明并行性。该方法的一个最近例子是UPC（统一并行C，Unified Parallel C）（参见http://upc.gwu.edu）。在第8章中将详细叙述如何具体使用线程和其他方法对共享存储器系统进行编程。

从程序员观点来看，共享存储器多处理机是很有吸引力的，因为它方便了对数据的共享。图1-7中所示的基于总线互联结构的小型（2-处理器和4-处理器）共享存储器多处理机系统是

很流行的，例如双-奔腾（dual-Pentium）和四-奔腾（quad-Pentium）系统。特别是2-处理器共享存储器系统非常经济有效。但是用硬件来达到所有处理器对所有存储器的快速访问是很困难的，特别是在有大量处理器时。因此大多数大型的共享存储器系统都具有某种形式的层次或分布式存储器结构，以使处理器能以更快的速度访问物理上相近的处理单元（与访问远距离的存储单元相比较）。在这种情况下使用*非均匀存储器存取*（NUMA，nonuniform memory access）术语，以区别于*均匀存储器存取*（UMA，uniform memory access）。

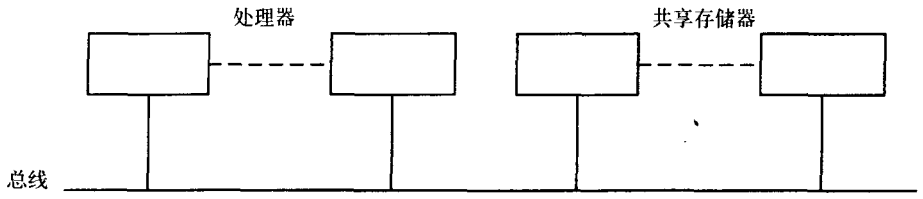


图1-7 小型共享存储器多处理机的简图

普通的单处理器都有快速的高速缓存以保存近期访问主存储器单元的副本，这样就可减少对主存储器的访问。通常在处理器和主存储器间有两级高速缓存。在共享存储器多处理机中继续保留高速缓存是为了给每个处理器提供一个自己的本地高速缓存。每个处理器具有快 15
速的本地高速缓存可在某种程度上在更大系统中缓和对不同的主存储器的不同访问时间的问题，但要使在不同高速缓存中的相同数据拷贝保证一致是必须考虑的一个复杂问题。当一个处理器对已高速缓存的数据项进行写入时，通常需要使系统中所有其他已高速缓存的拷贝变为无效。在第8章中将简要地论述这些问题。

1.3.2 消息传递多计算机

另一种多处理机系统的形式可以通过互联网络连接多台完整的计算机来构成。如图1-8所示。图1-8中每台计算机由一个处理器和本地主存储器组成，其他处理器不可访问该本地的主存储器。系统中的互联网络是供处理器间传递消息用的。这些消息可能含有由程序所指明的其他处理器进行计算时所需的数据。这种多处理机系统通常称为*消息传递多处理机*（message-passing multiprocessor），或简称*多计算机*（multicomputer），特别是如果它们是由能独立运行的自含式计算机组成时。

对消息传递多计算机进行编程仍涉及到将问题分解为各个部分，以使每个部分能同时执行以完成求解。也可使用并行或扩展的顺序语言进行编程，但一个更通用的方法是使用消息传递库例程，这些库例程与通常的顺序程序相连接以进行消息传递。我们常提到*进程*（process）这一术语，一个问题可被分成为多个并发进程，它们可在各台计算机上分别执行。如果有6个进程和6台计算机，则我们可在每台计算机上执行一个进程；如果进程数大于计算机数，则在一台计算机中就可能以分时方式执行多个进程。进程间将通过发送消息进行通信，这是在进程间分布数据和结果的唯一方法。

消息传递多计算机比共享存储器多处理机更容易在物理上加以*扩展*（scale）。也就是说，它更易构成较大规模。有一些专门设计的消息传递的处理器的例子。然而，消息传递系统也可使用通用计算机系统。

1. 多计算机网络

图1-8中所示的互联网络的作用是为从一台计算机向另一台计算机传递消息提供物理通

路。网络设计中的一些关键问题是带宽（bandwidth）、时延（latency）和成本（cost）。易于构造这一点也很重要。带宽是指在单位时间内可传输的位数（二进制位），以bits/sec（位/秒）表示。网络时延（network latency）是指消息经过网络传输时所需的时间。通信时延
16 （communication latency）是指发送消息所需的总时间，包括软件开销和接口延迟。消息时延（message latency）或启动时间（startup time），这是指发送0长度消息所需的时间，这实质上是发送消息时所需的软、硬件开销（查找路由、打包、解包等），还必须加上沿互联通路发送数据所需的确切传送时间。

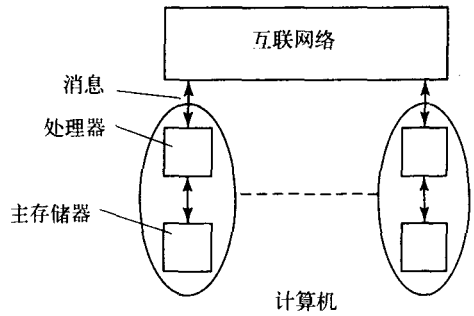


图1-8　消息传递多处理机模型（多计算机）

两个结点之间路经的物理链路数是一个重要的考虑因素，因为它是确定消息延时的主要因素。网络直径（diameter）是网络内两个相距最远的结点（计算机）间的最小链路数。应注意的是只考虑最短路由。如何用具有特定网络的多计算机有效地求解并行应用问题是非常重要的。网络直径给定了单个消息必须经历的最大距离，它可为某些并行算法找出通信的下限。

网络的对分宽度（bisection width）是指当将网络分成两个相等的部分时，所必须切割的链路（有时为线）数。对分带宽（bisection bandwidth）是在这些链路上的集合带宽，即从被分割网的一部分向另一部分在单位时间内可传送的最大位数。这两个因数在评估并行算法时也非常重要。并行算法通常需要数据在网络中移动。要将数据从网络的一部分移向另一部分必须使用处于两部分之间的链路，而对分宽度指明的是可用的链路数。

有好几种方法可用来互联计算机以形成多计算机系统。对于非常小的系统可以考虑用链路将每一台计算机与其他所有计算机互联起来。对于c台计算机，则总共需要$c(c-1)/2$条链路。这种穷尽的互联仅适用于非常小的系统。例如，4台计算机的集合采用穷尽互联就较为恰当。然而当规模增加时，互联数就会变得很大，因而从经济和工程角度来看再采用穷尽互联就不现实了。那时我们就需要考虑具有有限互联和交换式互联的网络。

有两类广泛使用的具有有限直接互联的网络-网格（mesh）网络和超立方体（hypercube）网络。作为互联网络它们不但是重要的，而且它们的概念也出现在并行算法的工程中。

（1）网格　二维网格是指在二维阵列中的每个结点能与其4个最邻近的结点相连的网络，如图1-9所示。一个$\sqrt{p}\times\sqrt{p}$网格的网络直径为$2(\sqrt{p}-1)$，因为从一个角结点到一个对顶角结点需要横跨$(\sqrt{p}-1)$个结点，再向下经过$(\sqrt{p}-1)$结点。如果将网格中的所有自由端结点与其对立端的结点循环相连，则就构成了环绕网（torus）。

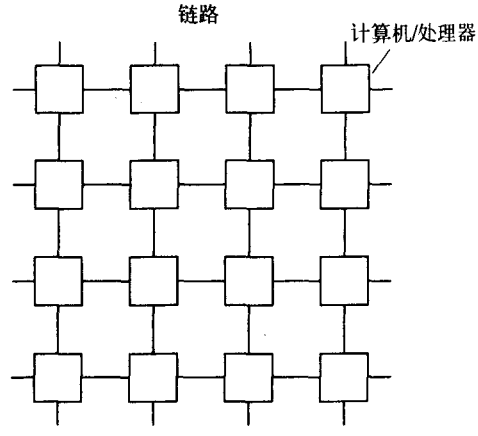


图1-9　二维阵列（网格）

网格和环绕网（torus network）非常流行，因为它们很容易排列和扩展。必要时还可将网络折叠（fold），即对结点进行行交叉和列交叉的排列，从而可使围绕连接变成简单的折回连接，而不再是从一边伸展到另一边。三维网格可用这样一种方法形成，即每个结点在x，y和z的三个平面中，

各与两个结点相连。在许多科学和工程问题中，使用网格结构非常方便，因为在这些问题中，求解点常排列成二维或三维阵列。

已有好多个使用二维或三维网格构成消息传递多计算机系统的实例，其中包括Intel的Touchstone Delta计算机（二维网格，1991年交付使用），以及1991年在MIT研制成的三维网格的样机，J-machine。更近期使用网格的例子是美国能源部加速战略计算创新计划于1995-97所 17
开发的ASCI Red超级计算机。ASCI Red落成在Sandia美国国家实验室，由9472个Pentium-II Xeon（自强）处理器组成，并使用了一个进行消息传递的38 × 32 × 2网格互联。网格也可用在共享存储器系统中。

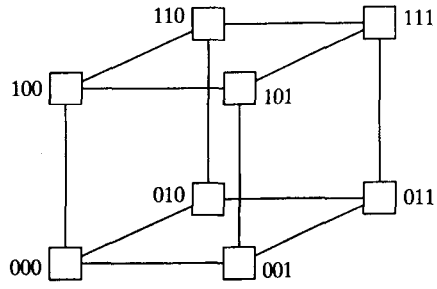


图1-10 三维超立方体

（2）*超立方体网络* 在一个d维（二元）*超立方体网络*中，每个结点与网络中每一维上的一个结点相连接。例如在一个三维超立方体中，x方向、y方向和z方向的连接形成一个立方体，如图1-10所示，一个d维超立方中的每一结点，将被分配一个d位（二进位）地址。每一位相应于一维，它可以是0或1，表示在该维上的2个结点。在一个三维超立方体中的结点将有一个3位地址。结点000将与地址为001，010和100的结点相连。结点111将与结点110，101和011相连。应注意的是与每个结点相连的那些结点仅与该结点的地址相差1位，这一特征，可扩展到具有更高维的超级立方体。例如，在一个五维的超立方体中，结点11101将与结点11100、11111、11001、10101以及01101相连。

超立方体的一个显著优点是，如果超立方体有p个结点，则它的网络直径将由$\log_2 p$给定，它随p的增加而合理地（低）增加。从每个结点发出的链路数也只按对数增长。超立方体的另 18
一个非常方便的地方是存在有最小距离的无死锁路由算法。为更好地叙述该算法，让我们从具有结点地址$X = x_{n-1}x_{n-2}\cdots x_1x_0$的结点$X$开始路由一个消息到结点地址为$Y = y_{n-1}y_{n-2}\cdots y_1y_0$的目的结点$Y$。$Y$中的每一位若与$X$中的对应位不同，则表明在该维上应该路由消息，而两者（地址位）的不同是可通过执行对应地址位对的异或操作$Z = X \oplus Y$发现的。Z位为1的那些位，均应该进行路由。处于路径上的每个结点，要执行当前结点地址和目的结点地址的异或操作。通常选择Z中的最高有效位为1的那一位作为路由的开始。例如，在一个六维超立方体中，要从结点13 (001101)路由到结点42（101010），则将是从结点13路由到结点45(101101)，再到结点41（101001），再到结点43（101011），最后到达结点42（101010）。超立方体路由算法有时称为*e-立方体路由算法*（e-cube routing algorithm），或*从左到右的路由*（left-to-right routing）。

一个d维超立方体实际是由两个$d-1$维超立方体用第d维链路将两者连接起来而组成的。图1-11中示出了由两个三维超立方体用8条连接画出的四维超立方体。因此它的对分宽度为8（一个p结点超立方体的对分宽度为$p/2$）。一个五维的超立方体是由两个四维超立方体以及它们之间的连接所组成的，用这种方法可构成更大的超立方体。在一个实际系统中，网络必须设计成是二维或是三维的。

超立方体是更大的k元d立方体系列中的一部分，但仅有二元超立方体（$k = 2$）在多计算机构造和并行算法中起到重要作用。20世纪80年代初期，由于在美国加州理工学院建成了称为Cosmic Cube的先驱研究系统[Seitz, 1985]，此后超立方体网络便成为构造消息传递多计算机的流行方式。但自80年代后期起对超立方体网络的兴趣已经减弱。

在各个计算机间进行直接连接的另一种方法是在各种配置中使用交换器来路由消息。

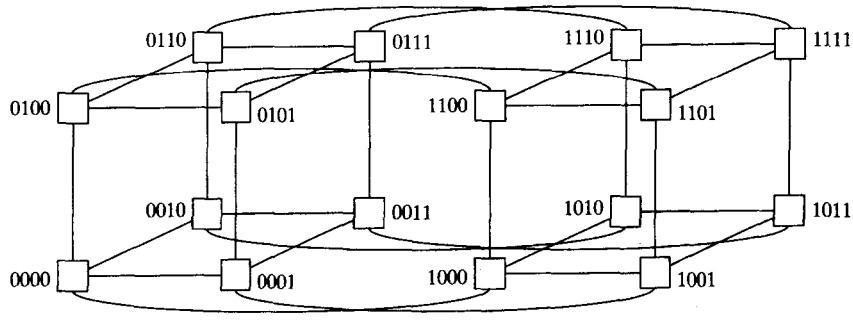


图1-11　四维超立方体

(3) 纵横交叉交换器　纵横交叉交换器通过为每一个连接使用一个交换器来提供完全的连接。纵横交叉交换器在共享存储器系统中的使用比在消息传递系统中多，用以连接处理器与存储器。纵横交叉交换器的线路图如图1-12所示。已有许多系统的实例在系统的某一层次上使用纵横交叉交换器，特别是在具有很高性能的系统中。我们的一个学生在20世纪70年代曾建成一个纵横交叉交换器多处理机系统[Wilkinson and Abachi，1983]。

19

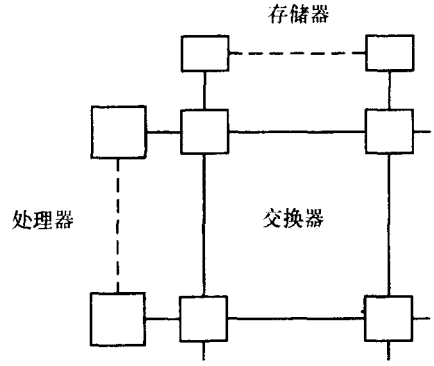


图1-12　纵横交叉交换器

(4) 树状网　另一种交换器的配置是使用二叉树(binary tree)，如图1-13所示。树中的每个交换器(结点)有2条链路与位于其下层的2个结点相连，就如同网络从根结点扇出一样。这一特定的树称为完全二叉树。因为每一层均是占满的。树的高度是从根到最低层叶的链路数。树结构的一个主要特点是其高度与叶结点数成对数关系；对于一个有p个处理器(在叶结点上)的二叉树，它将有$\log_2 p$层交换器。树状网可以是非完全的，也可以是非二叉的。在一个m叉树中，每个结点将与其下一层的m个结点相连。

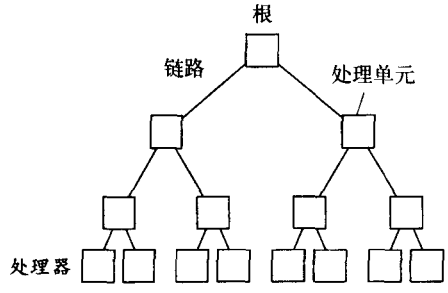


图1-13　树结构

在均匀请求的模式下，接近根结点处的通信量会变得很大，从而可能成为性能瓶颈。在粗树网络(fat tree network)中[Leiserson，1985]，链路数越接近根结点处时就越多。在二叉粗树中，随二叉树层间的需求，以并行方式增加链路数，而越接近根结点处时，就越增加更多的链路数。Leiserson按此思想提出一种通用粗树(universal fat tree)，在这种树中，趋向根结点的结点间的链路数将指数式地增加，从而允许趋近根结点处的通信量的增加，以达到减小通信瓶颈的目的。以这种粗树互联网络所设计的非常著名的计算机是Thinking Machine 公司的Connectim Machine CM5计算机，它使用的是4叉粗树[Hwang,1993]。粗树在此后也一直被使用。例如，Quadrics QsNet网(参见http://www.quadrics.com)就使用粗树。

20

(5) 多级互联网　多级互联网(multistage interconnection network，MIN)是一类含有多种配置的具有相同特征的多级交换器。在某一级上的交换器与其邻接的交换器以各种对称的方式相连，以形成从网络的一边到另一边的一个通路(有时是反向的)。多级互联网的一个例

子是示于图1-14的Omega网络（有8个输入和输出）。借助目的地址该网络有一个非常简单路由算法。如图1-14中所示，输入和输出用地址给定。每个交换单元需要一个控制信号以选择是上输出或是下输出（0表示是上输出，1表示是下输出）。目的地址的最高位用来控制第1级中的交换器；若最高位为0，就选择上输出，若为1，就选择下输出。目的地址的次高位用来选择下一级的交换器输出，如此等等，直到选择出最后的输出。尽管在一个自由网中总是可以连通任意的一个输入到任意的一个输出，但Omega网络是高度阻塞的。

多级互联网已有很长的历史，它最初是为电话交换机而开发的，它有时仍被用来互联计算机或计算机组以构成真正大型的系统。例如，ASCI White超级计算机就使用Omega多级互联网。有关多级互联网的更多信息请参见[Duato，Yalamanchili，and Ni，1997]。

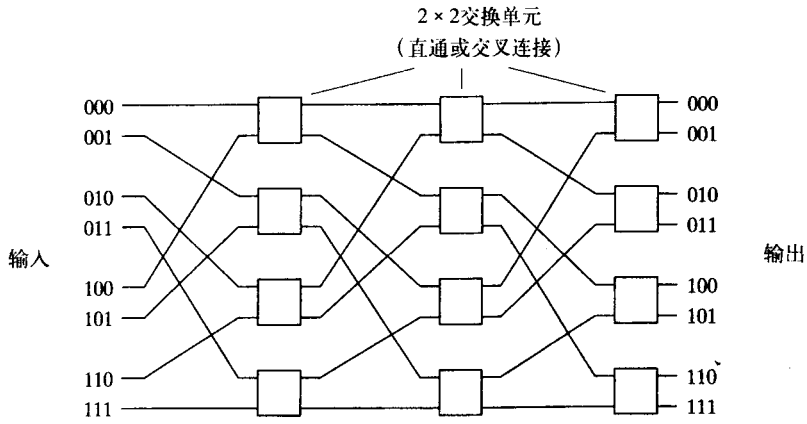


图1-14 Omega网络

2. 通信方法

将一个消息从源结点传送到目的结点的理想情况是在它们之间有一条直接链路。但在大多数系统和计算中，将消息从源结点路由到目的结点时，常需要经过中间结点，有两种方法可将消息从源送到目的地：电路交换（circuit switching）和包交换（packet switching）。 21

电路交换需在源和目的地之间建立通路，并维持消息途径的所有链路的畅通。传递所需的链路均被保留，直至消息传递结束。简单的电话系统（不使用先进的数字技术）是电路交换系统的一个例子。一旦电话接通，线路就保持连接，直至通话结束。电路交换被使用在某些早期的多计算机中（如Intel IPSC-2超立方体系统），其缺陷是在整个传递消息过程中，要强行保留通路中的所有链路，在消息传递结束之前，不允许其他消息使用通路中的任一链路。

在包交换中，消息被分成为多个信息“包”，每个包都含有源地址和目的地址，以在互联网络中路由该包。包有一个最大尺寸，例如1000个数据字节，如果消息长度超过此值，就必须将此消息分成多个包进行传递。在包被传递到下一结点之前结点内部要提供的缓冲区保留这些包。如果通向下一结点的通路被封锁，包就被保留在缓冲区中。邮递系统是包交换系统的一个例子。信件从邮箱送到邮局，并在一些中间站中处理，最后被送往目的地。这种包交换形式称为存储转发包交换（store-and-forward packet switching），这种交换允许一旦当前的包被转发以后，其链路就可由其他包使用。不幸的是，刚才所叙述的存储转发包交换，由于不论输出链路是否可用，必须将包首先存到每个结点的缓冲区中，从而将导致显著的时间延迟。在直通（cut-through）交换方式中，可消除这种需求，这种交换技术最初是为计算机网络开发的[Kermani and Kleinrock，1979]。在直通交换中，如果输出链路可用，则消息便可直

接向前传递而无需存入结点的缓冲区，即“直通”。因此，如果整个通路可用，则消息将立即传送到目的地。但应注意的是，如果通路受阻，则就需要足够的存储区以保留正被接收的、完整的消息/包。

相对于通常的存储转发路由，Seitz提出另一种称为*虫孔*（wormhole）的路由方案[Dally and Seitz，1987]，以减少缓冲区大小及减少时延。在存储转发包路由中，消息完整地存储在结点中，当输出链路空闲时，就将整个消息送出。而在虫孔路由中，消息被分成比包更小的单位，称为*片*（流控制数字）。一个片通常为1个或2个字节[Leighton，1992]。结点间的链路可能为片中的每一位（二进制）提供一条链路以使片能并行传送。当连接链路可用时，从源结点向下一结点传送的初始消息仅是消息的头部。当所有链路都可用时，消息的各后继片才会被传送，从而使这些片在网中分布。当头片向前移动时，下一个片才可跟着移动，依次类推。在结点之间必须有一个请求/确认系统，以“拖动”片向前移动。当一个片已准备从它的缓冲区向前移动时，它就向下一结点发请求。当下一结点的片缓冲区为空时，它就要求发送结点发送该片。当消息的一部分（片）已经链接时，必须为消息传递保留整个通路。因此它不允许其他消息包与这些消息片交叉地占用这些链路。

在虫孔路由方法中，每个结点所需的存储容量较少，且由它导致的时延与通路长度无关。[Ni and Mckinley，1993]给出了有关分析以指明在虫孔路由方法中，时延与通路长度无关。如果消息片的长度远小于整个消息长度，
22 则虫孔路由的时延将为常数而与路由的长度无关。（电路交换具有类似特征。）与之相反，存储转发包交换所产生的时延近似地正比于路由长度。图1-15对此作了说明。

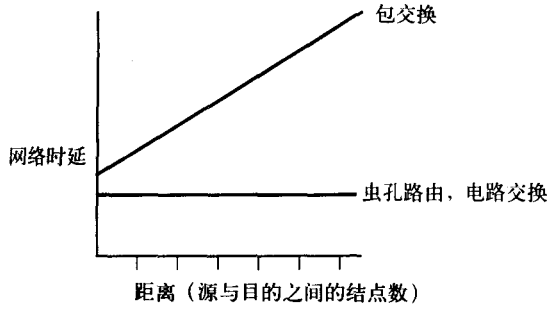


图1-15　网络时延特征

正如我们已见到的，互联网络需要路由算法，以在结点间找到一条通路。某些路由算法是自适应的，即它们依照某些准则，特别是局部的通信量情况，在网络中选择不同的通路。一般的路由算法，除非是精心设计的，很容易导致*活锁*（livelock）和*死锁*（deadlock）。*活锁*特别会在自适应路由算法中出现，它描述这样一种情景，即一个消息包不停的在网中传送，但总不能找到其目的地。当包由于被其他等待转发的包阻塞而这些包又以同样的方式被阻塞以致于没有包能向前移动时就发生了*死锁*。

无论是存储转发还是虫孔网络都有可能出现死锁。出现在使用存储转发路由方法的通信网中的死锁问题，已在那样的环境中做了广泛研究。有关在任何网络不会出现死锁的路由算法的数学条件和解决方案，可参见[Dally and Seitz，1987]。避免死锁的通常方法是为网络提供一些*虚拟通道*（virtual channel），每个虚拟通道由自己独立的缓冲区供不同类型的消息使用。*物理链路*或通道是指结点间的实际硬件链路。多个虚拟通道与一个物理通道相关，它们分时共享该物理通道，如图1-16所示。Dally和Seitz提出了使用独立的虚拟通道来避
23 免在虫孔网络中可能出现的死锁。

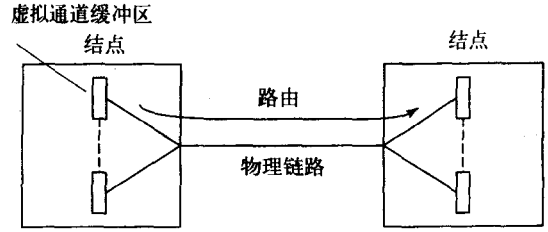


图1-16　多个虚拟通道映射到一个物理通道

1.3.3 分布式共享存储器

对程序员而言，消息传递范例不如共享存储器范例那样有吸引力。因为它通常需要程序员在它们的代码中使用显式的消息传递调用，而这又非常容易出错，且很难调试。因而很类似于低级的汇编语言的编程（使用处理器的内部语言编程）。数据不能共享而必须拷贝，在一些具体应用中这很可能成为问题，因为这些应用问题需要对大量数据进行多次操作。但是消息传递范例具有如下优点，即它不需要专门的同步机制以控制对数据的同时访问。如果使用这些机制，则将显著地增加并行程序的执行时间。

由于意识到从编程观点而言期望使用共享存储器范例，一些研究人员开始追求*分布式共享存储器系统*的概念。如其名字所提及的那样，在这种系统中，存储器物理地分布在每个处理器中，但每个处理器使用单一的存储器地址空间对整个存储器进行访问。当一个处理器要访问的单元不在本地存储器中时，必须使用消息传递方法在处理器和存储器单元之间以某种自动方式进行数据的传递，以隐藏存储器是分布的这一事实。当然，远程访问将导致更大的延迟，而且比起本地访问来，此延迟常常是相当大的。

多处理机系统可设计成在物理上存储器是分布的，但运行却如共享存储器，且对程序员来讲犹如一个共享存储器。已有不少研究项目试图用专门设计的硬件达到这一目的，而且已出现基于这一思想的商品化系统。也许最受青睐的方法是使用联网的计算机。在一组联网的计算机上实现分布式共享存储器的一种方法是使用现有的单计算机上的虚拟存储器管理系统，几乎在所有的系统中都提供了这一管理系统以管理它的本地存储器层次结构。可将虚拟存储器管理系统加以扩展，以使得分布在不同计算机中的存储器在感觉上是全局共享存储器。这一思想被称为*是共享虚拟存储器*（shared virtual memory）。共享虚拟存储器的最先开发者之一是Li[Li，1986]。还有一些实现分布式共享存储器的其他方法，它们不需要使用虚拟存储器管理系统或专用硬件。不管如何，该系统在物理上如同图1-8所示的消息传递多计算机那样，除了现在本地的存储器变成了共享存储器的一部分之外，而且还如图1-17所说明的那样它可由所有处理器加以访问。

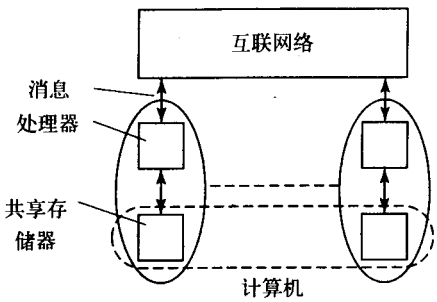


图1-17 共享存储器多处理机

在第8章介绍共享存储器编程的基本概念后，第9章将详细考察分布式共享存储器系统的实现和编程。共享存储器和消息传递均应被视为编程范例，它们中的任何一个均可是任何类型多处理机的编程模型，尽管特定系统可以被设计成是这样或是那样。 24

应指出的是，在一个消息传递系统上实现的DSM通常不具有一个真正共享存储器系统的性能，也不会有在消息传递系统上直接使用消息传递的性能。

1.3.4 MIMD和SIMD的分类

在单处理机计算机中，由程序生成的是一个单指令流。这些指令对数据项进行运算，Flynn（1996）创造了一种计算机分类方法，将这种形式的单处理器计算机称为*单指令流单数据流*（single instruction stream-single data stream, SISD）计算机。在一个通用的多处理机系统中，每个处理器拥有一个独立的程序，由每个程序为每一个处理器生成一个指令流，每条指令对不同的数据进行操作。Flynn将这种计算机分类为*多指令流多数据流*（multiple instruction stream- multiple data stream, MIMD）计算机。到目前为止所叙述的共享存储器或

消息传递多处理机都属于MIMD类型。术语MIMD已经受了时间的考验，至今仍被广泛地用于在这种操作模式下的计算机系统中。

除了上述的SISD和MIMD的两个极端情形外，如果对某些应用而言将计算机设计成由单一程序生成指令流，但却有多个数据流存在时，将会在性能上有很大优势。源于程序的指令被广播到多个处理器。每个处理器实质上是一个没有（程序）控制器的算术运算处理器。由一个独立的控制器负责从存储器中取出指令并将这些指令发送到各个处理器中。每个处理器同步执行相同的指令，但使用不同的数据。为灵活起见，可以禁止某些处理器参加指令的运算。由数据项形成一个阵列，在一个指令周期内指令对整个阵列进行运算。Flynn将这类计算机称为*单指令流多数据流*（single instruction stream- multiple data stream, SIMD）计算机。开发SIMD计算机的原因是因为有许多重要应用绝大部分都对数据阵列进行运算。例如大多数物理系统的计算机模拟（从分子系统到天气预报），均是以必须处理的巨大数据点阵列开始。另一个重要应用领域是低层图像处理，此时需对图像的图元素（像素，Pixel）加以存储和加工，如像将在第12章中所叙述的那样，如果系统能同时对数据点完成相似操作，则不但硬件能有效地工作，也使编程变得更为简单。该程序简单地由对数据点阵列进行运算的单指令序列和由一个独立的控制器执行通常的控制指令所组成。在本教科书中，我们将不讨论SIMD计算机，因为它们是为某
25 种特殊应用而专门设计的计算机。当今的计算机为多媒体和图形应用设置有SIMD指令。例如从奔腾II开始的奔腾系列增加了称为MMX（MultiMedia eXtension）的这种SIMD指令以加速对多媒体和其他应用的处理，这些应用需要在不同的数据上完成相同的操作。

Flynn分类的第四种组合，即*多指令流单数据流*（multiple instruction stream- single data stream, MISD）计算机，实际上并不存在，除非特别地将流水体系结构归为这一类，或是把某些容错系统归为这一类。

在我们所关心的MIMD类别中，每个处理器执行自己的程序。这可被描述为*多程序多数据*（multiple program multiple data, MPMD）结构，如图1-18所示。当然所有要执行的程序可能是不同的，但通常只需编写两个源程序，其中一个供指定的主处理器使用，而另一个则供其余称为从处理器的处理器使用。我们可以使用或不得不使用的一种编程结构是*单程序多数据*（single program multiple data, SPMD）结构。在此种结构中，只需编写一个源程序，每个处理器将执行该程序自身的拷贝。虽然它们是独立地执行但却不是同时的。可以这样来构成源程序，即根据计算机的特征使得该程序的一部分只由某些计算机执行而不被其他计算机执行。对于主从结构来说，该程序中的一部分由主处理器执行，而另一部分则由其余的从处理器执行。

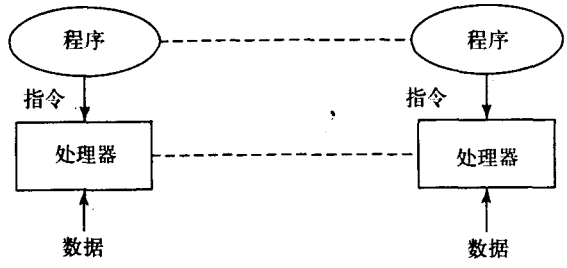


图1-18　MPMD结构

1.4　机群计算

1.4.1　以互联计算机作为计算平台

到目前为止，我们已叙述了专门设计的、用作并行计算平台的并行计算机，其中包括多处理机或多计算机。多年来，许多大学的研究项目致力于设计这种多处理机系统，通常采用几乎完全不同的体系结构方式和不同的软件方案，而每一个项目都力求能达到最佳性能。在

大型系统中，直接连接已被交换器和多级交换器（多级互联网）所替代。计算机系统的制造商们已提出许多此类设计。但大多数制造商面临的主要问题是处理器的速度不断地提高，每个新一代的处理器速度更快且在内部能同时完成更多的操作以提高性能。计算机购买者最能 26
注意到的改进是个人计算机的时钟速率的增加。基本的时钟速率仍以毫不减弱的势头继续增长。以购买一台奔腾（或其他）计算机来说，来年买的相同系统将比当年买的主频要高出一倍。而除了时钟速率外，其他的因素也使系统运行得更快。例如，较新的设计可能在处理器中实现了更多的内部并行以及采用了其他的一些方法使操作执行得更快。他们还使用具有更高带宽的存储器结构。对于“超级计算机”制造商而言，针对不断出现更快处理器的出路就是去使用巨量的可得到的处理器。例如，假设一台多处理机用2004年当今最先进的3GHz的处理器来设计，则使用500个处理器所组成的系统其性能仍将超过若干年后任何一台单处理器系统，当然其成本是非常昂贵的。

在20世纪80年代后期和90年代早期，一些大学尝试了另一种更为经济有效的方法，即使用互联的工作站和PC机来构成一个强大的计算平台。从不同的观点出发，许多研究项目探索构成计算机群。某些早期的研究项目探索使用实验室中的工作站来构成*工作站机群*（cluster of workstations，COWs）或*工作站网络*（network of workstations，NOWs），如Berkeley 的NOW研究项目[Anderson，Culler，and Patterson，1995]。另一些探索项目利用尚未作它用的现有工作站的空余时间，因为在通常的时间中，工作站特别是那些在办公室中的工作站不会被连续使用，或者即使使用也不需要100%的使用处理器时间。

开始时，用工作站网络进行并行计算之所以吸引人是因为已有的工作站网是为通用计算服务的。工作站，顾名思义已用于各种编程和从事与计算机相关的活动。此后，人们很快意识到工作站网络可为用户提供一个非常吸引人的从事高性能计算的另一种途径，通常的途径是使用昂贵的超级机和并行计算机系统。使用工作站网络有许多显著优点胜过那些专门设计的多处理机系统。关键的优点如下；

1）可用较低的成本使用有很高性能的工作站和PC机。

2）系统可很容易地引入那些可买到的最新的处理器，且系统可通过加入附加的计算机、磁盘以及其他资源增量式地加以扩展。

3）可使用已有的应用软件或对之加以修改。

要使这些工作站能集合地工作必须要有相应的软件，很巧合的是在大约同时开发了消息传递工具使上述概念变得可行。为这些工作站提供并行程序设计软件工具的最重要的消息传递研究项目是始于20世纪80年代后期的*并行虚拟机*PVM（Parallel Virtual Machine）。PVM是工作站网的一个关键支撑技术，它使得在工作站网上的并行程序设计得以成功地实现。在此之后又定义了标准的消息传递库，即*消息传递接口*MPI（Message-Passing Interface）。

到了20世纪90年代，由于PC机已是相当便宜和具有强大功能，使得采用众多互联的PC机作为并行计算平台的概念变得成熟。定位作为实验室计算机的工作站正在部分地被常规的PC 27
机所替代，而在通用实验室中工作站和PC机两者的区别已经消失。“工作站网”已让位于简单的计算机“机群”（cluster），而在机群中集合地使用众多计算机求解一个问题的计算，称为*机群计算*（cluster computing）。⊖

⊖ 虽然“机群计算”现在是一个被接受的术语，但它自20世纪90年代早期开始就用在工作站/PC机网络中用来集合求解问题。例如，在1992年和1993年于佛罗里达州立大学的超级计算研究所举行过称为机群计算的专题学术讨论会。

1. 以太网连接

联网计算机的通信方法通常使用的是以太网类型，最初的以太网由单线组成，所有的计算机都附接到该线上，如图1-19所示。图1-19中列出的是一个文件服务器，它保存所有的用户文件和系统的实用例程。单线的使用被认为是以太网设计的低成本和布局上的优点。现在单线结构已被各种交换器和集线器所替代，但以太网的协议仍保持不变。交换器顾名思义，在计算机间提供直接的交换式连接，从而允许同时有多对连接接通，如图1-20所示，而一个集线器简单地说就是与所有计算机相连的一个点。交换器自动地将包路由到它们的目的地，并允许在独立的计算机对之间同时有多对连通。交换器可用各种构造互联，并在网络中的计算机间路由消息。

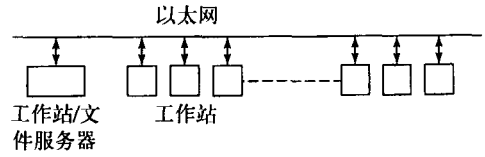


图1-19　最初以太网型单线网

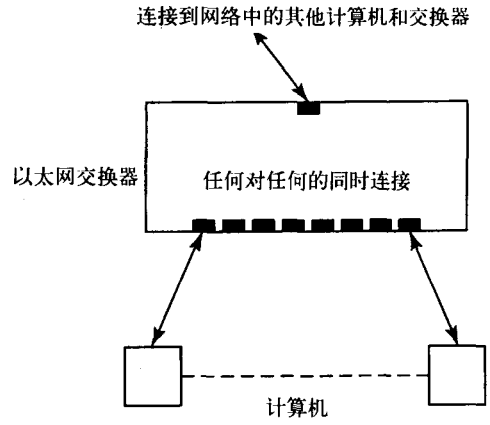


图1-20　以太网交换器

在以太网类型的连接中，源和目的结点之间的所有传递是以包形式顺序进行的（在线上一位一位地进行）。包中含有源地址、目的地址以及数据。
28 基本的以太网格式如图1-21所示。图中的前同步码（preamble）部分是作同步用的。可载的最大数据大小为1.5K字节，如果欲传送的数据大小超过该上限值，则就需要将其分成多个包，而每一个包都带有自己的源和目的地址。⊖从源到目的地，各个包可取不同路径，在大型网络或因特网上通常都是如此，但在到达目的地后必须将这些包以正确的顺序重组。

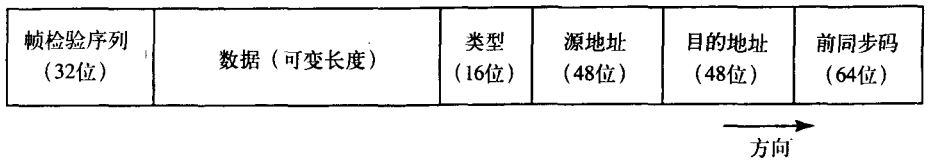


图1-21　以太网的帧格式

如前所述，最初以太网协议的设计是使用单线来连接众多计算机的。由于每个工作站是完全独立运行且可在任何时间发送消息，因此当一台计算机需要使用以太网线路时，此时该以太网线路可能正被用来传送由另一台计算机发出的包。如果能检测到已有信息在网上传播，就不会再向网络发送消息包。一台接到以太网线路欲发送消息包的计算机只有在没有其他信息沿以太网线路传送的时刻，才可能发送它的消息包。但是在几乎同一时刻由不同的工作站发出多个包，如果有多个消息包同时提交给网络，则来自它们的信息将被破坏。在源结点处通过比较正欲发送信息和以太网线路上的实际消息就可检测出这种情况。如果两者不一样，则就按照以太网协议（IEEE 标准802.3）将自己的包在间隔一段时间后再次提交到网线上。

最初的以太网速度为10Mb/s，后被改进成100Mb/s和1000Mb/s（后者称为千兆位以太网）。互联可以是双绞线（铜）、同轴线、或是光纤，后者可用于更高速度和更长的距离。应该特别提醒的是，以太网的消息传递时延是非常大的，特别是由于使用某种消息传递软件引入的额外开销。

⊖ 增加包大小是可能的。Alteron Networks有一项专利技术称为巨帧，可使包大小从1 500字节增加到9 000字节。

2. 网络寻址

TCP/IP（传输控制协议/网际协议）是一个标准，由它构成了联网计算机进行通信和传递数据的规则。在因特网上，为识别目的，每个“主机”都给有一个地址。TCP/IP定义了一个32位数的地址，它被分成四个8位数（用于网际协议版本4，IPv4，）。在某些约束下，每个数的取值范围为0～255。一个完整地址的标记是由点分开的四个数所组成的。例如，可以给定一台计算机的IP地址号为:

```
129.49.82.1
```

以二进制表示的该地址应是:

```
10000001.00110001.01010010.00000001
```

29

地址被划分成字段以选择一个网络、一个可能的子网、以及子网或网络中的计算机（“主机”）。由地址的第1或前2、3、4位区别多种格式。图1-22中示出了IPv4格式的布局，该信息是有关设置一个机群的。

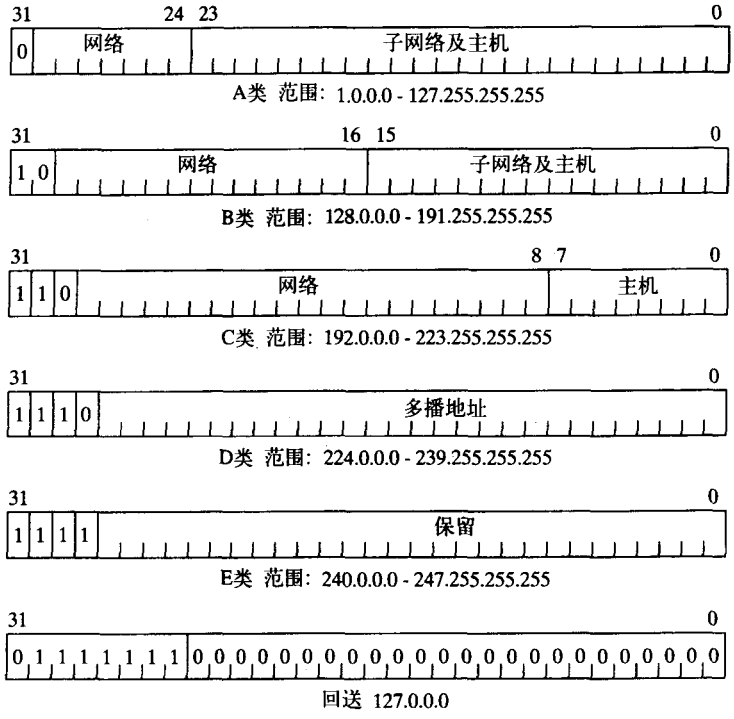


图1-22 IPv4的格式

A类格式的首位地址为0，并使用后7位来识别网络，其余的24位用来识别子网和“主机”（计算机）。在一个网络中，它可提供多达16 777 216（2^{24}）台主机，但它只能识别128个网络。主机可在各种子网的配置中加以安排。A类格式主要用在非常大的网中。

B类主要用在中型网中，由地址的前2位（10）加以识别，用后面的14位识别网络。余下的16位用来识别子网和主机。它在一个网中可提供65 536（2^{16}）个主机，而可用的网络数为16 384（2^{14}）个。同样，主机可在各种子网的配置中加以安排，但一个简单的配置可为256个子网和每个子网中有256台主机。即子网/主机部分的前8位用来识别子网，而后8位则用来识别子网中的主机。

C类主要用在小型网中，由地址的前3位（110）加以识别。网络用后面的21位加以识别。余 30

下的8位用来识别主机。它在一个网络中可提供256（2^8）个主机，而可用的网络数为2 097 152（2^{21}）个。同样，主机可在各种子网的配置中加以安排，但一个简单的配置是不含有子网。

D类主要用来同时向多个目的地广播一个消息，即传输将被多台计算机获得（称为多播）。回送（loopback）格式用来向自身发回一个消息以作测试用。某些地址是被保留的，如在图1-22中所示的那样，而在A、B、C类中的某些网络地址是预留供私人网用的（从10.0.0.0到10.255.255.255，172.16.0.0到172.32.255.255，以及192.168.0.0到192.168.255.255）。私人网地址可用做专用机群，有关这一点将在稍后讨论。

IPv4中的32位地址可识别约40亿台主机（2^{32} = 4 294 967 296，减去那些为特定主机保留而未使用的地址）。因特网已有了巨大的发展，多数人估计在2001年将超过1亿台主机[Knuckles，2001]，而且很快就需要更多的IP地址。IP地址不但用于如在计算机实验室中的那样固定连接到因特网的计算机，而且也用于因特网服务提供者（Internet Service Provider），供用户拨号和其他连接。已经开发了IPv6（网际协议版本6）以扩展IPv4的寻址能力，IPv6使用128位的地址码，它被分成8个16位段。因而它可提供多达2^{128}个主机地址（很巨大的一个数字！）。IPv6还对消息传送能力作了许多增强。所设计的IPv6网络软件也适用于IPv4。以下的论述将基于IPv4的地址。

在设置一个机群时，IP的寻址信息至关重要，因为IP寻址通常用于机群中计算机间的通信，以及机群与机群外用户之间的通信。网络地址由因特网赋号授权局（Internet Assigned Number Authority）分配给机构。而子网及主机的地址指派则由机构来完成（即由子网/主机的系统管理员来完成）。在通信软件中，通过设置掩码来选择网络、子网以及主机的字段。掩码是一个32位数，用1来定义地址的网络/子网部分。例如，在B类地址中用8-15位来定义子网的掩码，如图1-22所示：

```
255.255.255.0
```

其二进制码的表示为：

```
11111111.11111111.11111111.00000000
```

由它区分开主机地址与网络/子网的地址。应注意的是，子网和主机字段的分割不必一定要以8位作为分界，它需由本地的系统管理员确定，但网络地址（A、B、或C）是分配给机构的。

计算机通过以太网网络接口卡（NIC）连接到以太网电缆。在图1-21中所示的以太网格式的源和目的地址不是IP地址；它们是网络接口卡地址。这些地址长为48位，称为MAC（媒体访问控制器，Media Access Controller）地址。每个网络接口卡有一个预先定义的和独特的48位MAC地址，它在制造芯片或网卡时就已设置好（由IEEE注册局控制地址的分配）。虽然一台计算机的IP地址是由软件选择的，但每个NIC的MAC地址却是固定的。为在这两个地址之
31 间建立通信路径必需进行翻译。高层的软件使用IP地址，而低层的网络接口软件则使用MAC地址。实际上，MAC地址和IP地址两者都被包含在以太网的消息包中，在图1-21中IP地址被包含在消息包的数据部分中。

在IP寻址上面还有一层，该层将IP地址转换成名称以方便用户的交互。例如，`sol.cs.wcu.edu`是西卡罗来纳大学中的一个服务器，它位于数学和计算机科学系；它的IP地址是`152.30.5.10`。名称和IP地址间的关系是由域名服务（Domain Naming Service）建立的，它是一个分布式名称数据库。[⊖]

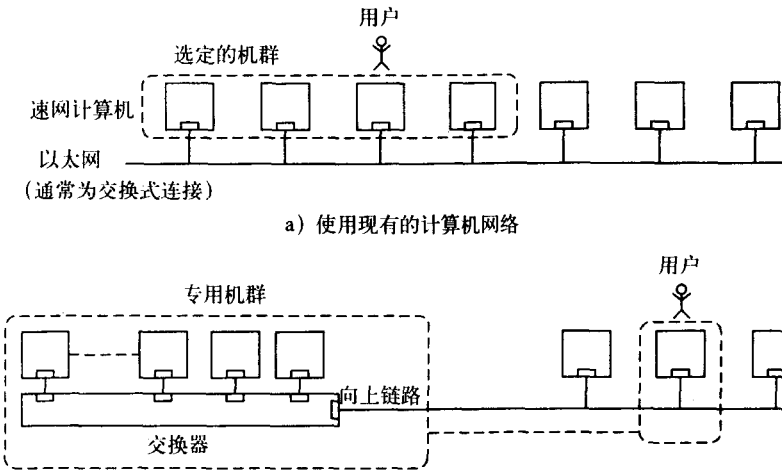
⊖ 在UNIX系统中主机名和IP地址间的关系被保存在一个称为主机的文件中，借助如cat /etc/hosts命令可对它进行检查。还有一张存有name/IP地址和主机以太网MAC地址间关系的查找表。该表可用地址确定协议命令arp -a加以检查。

1.4.2 机群的配置

可用多种方法来构成一个机群。

1. 现有联网计算机

第一类方法中的一个是使用实验室中现有的工作站来构成机群，如图1-23a所示。这些工作站为进行网络通信已分配有IP地址。通信软件提供了通信的手段。确实，在20世纪90年代早期，作者为讲授机群计算课程就试用了第一种方法，即利用现有的联网计算机。对教育单位来讲使用现有计算机的网络具有很强的吸引力，因为这种方法不需要其他的资源，但这种方法在计算机的使用方面存在较大问题。机群计算意味着要同时使用多台计算机。显然，借助现代的操作系统可以安排计算机在后台从事机群计算程序，而其他用户仍可在计算机前直接工作。此外，那时使用的消息传递软件（PVM）的结构也比较容易做到这一点。但在实践中发生的情况使此方法无法工作，因为当机群计算正在进行时，在计算机前工作的用户可使计算机停止（他们可简单地将计算机关掉！）。反过来，学生所从事的机群计算活动也可能使计算机陷入麻烦。此外，这种方法也需要对计算机有远程访问的能力，但如果处理不当，就会可能出现安全问题。在那时，实现远程访问（UNIX）的通常方法是使用“r”命令（rlogin，rsh），由消息传递软件调用它们以远程地启动进程。由于这些命令是不安全的，因此学生能够远程访问其他计算机，将可能导致浩劫。（口令不加密的传送）。当然，近来通过使用ssh使远程访问已变得安全。 32



a）使用现有的计算机网络

b）用实验室计算机构成专用机群

图1-23 构成机群的早期方法

2. 转向专用的计算机机群

很快我们发现如果简单地将实验室中因升级而淘汰的原有计算机来构成专用机群会显得非常经济有效（免费的！），而且少了许多麻烦。每次实验室计算机升级时，机群也跟着升级，但机群用的计算机型号正是实验室欲将其替换掉的去年的计算机型号。机群的计算机不需要显示器或键盘，它们之间使用在实验室中所使用的相同通信介质进行连接。简单地将计算机移到一个专用的机群中时，不需要改变IP地址。除了不再有本地用户坐在其控制台前之外，计算机仍如以前那样归属于子网。所有的访问都是远程完成的。用户在机群组外登录到一台计算机，并与自己的那台计算机一起进入到机群计算机形成一个新的机群，图1-23b对此作了

说明。应注意的是，在此机群中的计算机类型将是最初为计算机实验室所选择的类型。例如，我们机群的构成包括20世纪90年代早期的8台SUN IPC计算机，它们后来被升级成通用实验室淘汰下来的8台SUN Ultra计算机。

3. Beowulf机群

一个小型但非常有影响力的机群计算研究项目是于1993年在NASA Goddard 空间飞行中心启动的，该项目致力于通过使用很易得到的低廉部件来构成经济有效的计算机机群。他们选用标准、现成的微处理器以及易得到的操作系统（Linux），并使用低价以太网进行互联。与其他有关构成机群的研究项目在设计中经常使用某些专用的部件及软件意图获取高性能的策略相反，NASA 的研究项目立足于只使用广泛可用的、低价的部件，并基于价格/性能作为部件的选用标准。该项目被命名为Beowulf研究项目[Sterling，2002a and 2002b]。该名称已被认为是一种标记，以代表任何使用商品互联网和易获得软件、旨在构成一个经济有效的计算平台的低廉的计算机机群。开始时使用的是Intel处理器（486）和免费的Linux操作系统，Linux到目前为止仍是采用Intel处理器的Beowulf机群的通用的操作系统。当然，在Beowulf机群中也可使用其他的类型的处理器。

对一个冠名为Beowulf的机群的主要特征是它采用已广泛流行的部件以获得最好的价格性
33 能比。术语*商品计算机*（commodity computer）意在突出这样一个事实，即个人计算机的价格现在是如此便宜，因而可以较短的间隔周期来更新它。个人计算机的巨大市场使得它们的制造成本非常低廉。而且这一情况也适用于处理器相关的所有部件，如存储器和网络接口。目前的商品以太网接口卡（NIC）其价格最为低廉。用这样的互联网连接商品计算机就可构成一个商品化计算机的机群。

4. 超越Beowulf

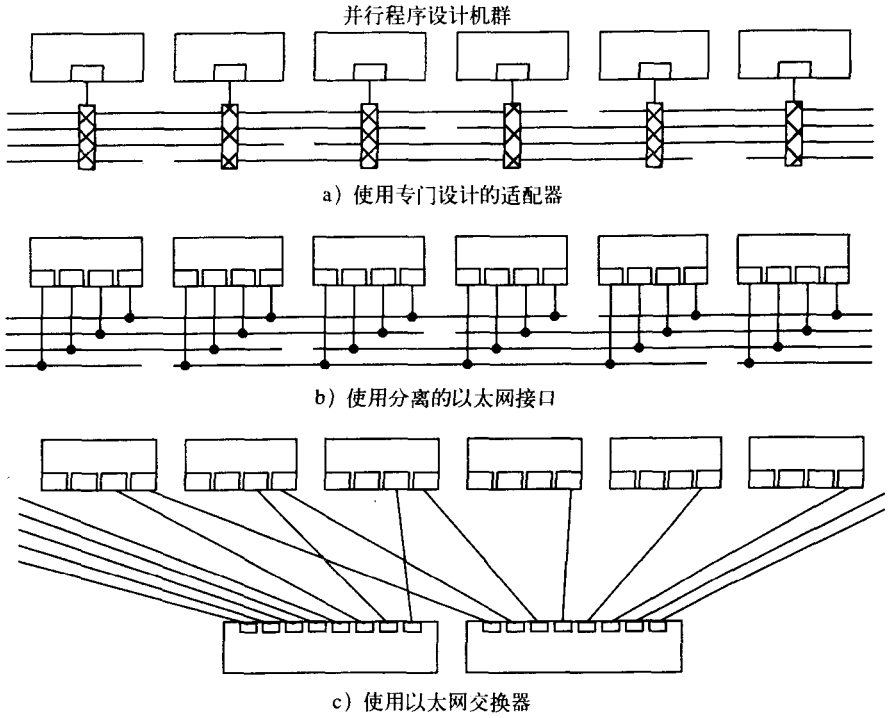
显然，如果具有经济意义的话，人们将使用具有更高性能的部件，而真正的高性能机群将使用具有最高性能的部件。

（1）*互联网*　在低成本的机群中，Beowulf通常使用快速以太网，一种简单的升级方法是采用千兆位以太网，在夏洛特北卡罗来纳大学就采用这种方法。但是，还有许多其他的更专用和更高性能的互联网，如带宽为2.4Gbits/sec的Myrinet互联网。此外，还可使用其他的互联网，其中包括cLan，SCI（Scalable Coherent Interface），QsNet以及Infiniband；有关更多细节请参见[Sterling，2002a]。

（2）*具有多个互联的机群*　Beowulf及其他的研究项目所开发的机群都使多个并行的互联以减少通信开销。机群可以配置多个以太网卡或是不同类型的网卡。在最初的Beowulf项目中，每台计算机使用两个正规的以太网互联以及采用了一种“通道绑定”（“channel bonding”）技术。通道绑定技术将若干个物理接口与一个虚拟通道连接在一起。已有可用的软件来实现这一点（例如，参见http://cesdis.gsfc.nasa.gov/beowulf/software）。就Beowulf而言，所生成的结构必须是经济有效的。这种技术确实使性能有了显著的改进（更详尽的细节请参见[Sterling，2002]）。一些近期的机群使用较慢的以太网作结构上的连接，而在程序执行时使用如Merinet那样的快速互联进行通信。

我们已致力于使用多条以太网线路的概念来配置机群，如图1-24a、b和c所示。对交换器的使用有很多种方法。图1-24中所示的方法就一般的分类而言是属于重叠互通网络[Hoganson，Wilkinson，and Carlisle，1997；Wilkinson，1990，1991，1992a，1992b]。重叠互通网具有提供互通区域以及区域间重叠的特征。在重叠互通的以太网中，它是通过如图1-24a、b和c所示的以太网段来实现的，但也有若干其他方法可实现重叠互通，例如可参见[Wilkinson and Farmer，

1994]。应提及的是图1-24的结构能显著地减少冲突，但时延和数据传输时间则仍与原来一样。



a）使用专门设计的适配器

b）使用分离的以太网接口

c）使用以太网交换器

图1-24 重叠互通以太网

（3）*对称多处理机（SMP）机群* 小型的共享存储器多处理机的互联是基于总线的，关于这一点已在1.3.1节中叙述过，由于处理器和存储器模块间是对称的，故被称为对称式（共享存储器）多处理机。基于奔腾处理器的小型共享存储器多处理机是非常经济有效的，特别是双处理器系统。因此，用“对称多处理机”（SMP）系统来构成图1-25那样的机群是合理的。 34
在这种结构的机群上可以进行某些很有趣的编程，在SMP间可使用消息传递方法，而在SMP内部则可使用线程或其他共享存储器方法。然而通常为方便起见，统一地使用消息传递。当消息要在一个SMP计算机内的处理器之间传送时，在实现时可使用共享存储器的单元来保留消息，这就大大加快了通信。

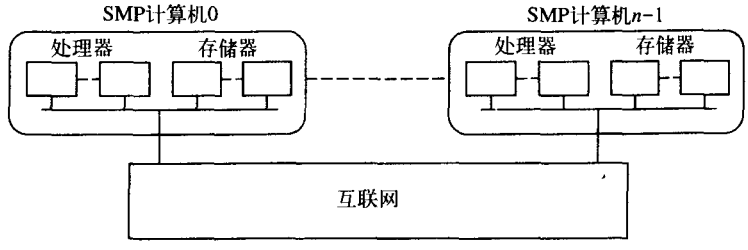


图1-25 共享存储器计算机机群

（4）Web*机群* 由于因特网和World Wide Web的问世，使在不同地点甚至不同国家的计算机互联起来。Web的出现已经导致这样的可能性，即使用连接到Web上的各地的计算机进行并行程序设计。已经有许多研究项目正在从事使用计算机“网”（web）来构成一个并行计算的 35
平台。这种想法原来称为*元计算*（metacomputing），而现在则称为*网格计算*（grid computing）。

从事这种大型机群计算类型的研究项目有Globus，Legion和WebFlow。有关这三个系统的更多细节可在[Baker and Fox，1999]中找到。

1.4.3 打造“Beowulf风格”的专用机群

“Beowulf风格”意味着使用商品部件。这便是可从著名的供应商那里购买到的普通PC机。尽管供应商们的目标是朝向使用多个双/四处理器的服务器以获得很高的性能，但这些供应商现在已信奉机群计算并提供预先包装好的机群计算系统。无论如何，由于引入了如Oscar那样的软件包使设置过程大为简化，Oscar软件包使装载操作系统和其他过程的工作可自动地完成。我们将在本小节的后面简述Oscar软件包。

1. 硬件配置

一个通常的硬件配置是使一台计算机作为主结点，而机群中的其他计算机在私有网络中作为计算结点。主结点传统地被认为是*前端机*（frontend），其作用如同一台文件服务器，但有很大的主存储器和外存储器容量。计算结点虽然可以不用磁盘，但一般为方便起见都带有磁盘驱动。计算结点和主结点间的连接可用快速以太网或千兆位以太网（或使用更专用的互联网，如Myrinet），如图1-26所示。主结点需要第二个以太网接口以与外界相连，并需要一个全局可访的IP地址。计算结点只有私有的IP地址，即这些计算机只能在机群网络中通信，在机群外不能对它们进行直接的访问。

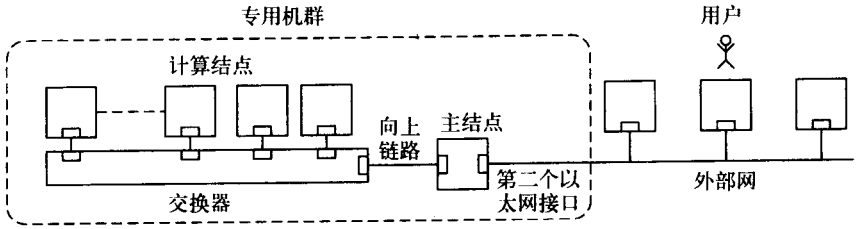


图1-26　具有主结点的专用机群

该模型可用若干方法进行增强。可使用另一个计算机作为管理结点。系统管理员用该结点来监控机群和进行测试。计算结点的唯一用途是完成计算，因此它们不需要键盘或显示器。但若能通过每台计算机通常的控制台输入对它们进行访问，则显然比较方便。因此，这些结点的串行连接可以通过一个集中器交换器接回到主结点或是管理结点（如果系统中有的话）。可用各种方法加以连接。图1-27中示出了一种方案，它有一个键盘和一个显示器，可在主结点和管理结点间切换。

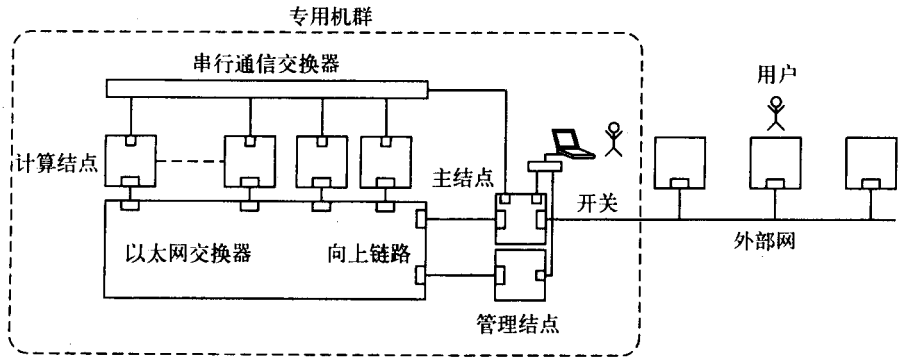


图1-27　具有主结点、管理结点和串行连接的专用机群

2. 软件配置

通常机群中的每一台计算机都有一个操作系统的拷贝（传统的是Linux，但也可构成Windows的机群）。通常主结点还为机群保存所有的应用程序文件，并能使用网络文件系统将其配置成一个文件服务器，它允许计算结点远程地阅看和直接访问已存储的文件。最常用的网络文件系统是NSF。应注意的是，机群的计算结点是与外部网分开的，因而所有的用户访问是借助另一个以太网接口和IP地址通过主结点进行的。在主结点上安装的还有消息传递软件（MPI和PVM，将在第2章中讨论）、机群管理工具以及并行应用程序。消息传递软件介于操作系统和用户之间，因此是中间件（middleware）。

一旦所有软件都安装好以后，用户可在主结点上登录，并使用消息传递软件登记到各计算结点，这也将在第2章中叙述。挑战性的任务是在于首先要设置机群，这需要有详尽的操作系统和网络的知识（即Linux命令及如何使用它们）。有许多书籍和Web网站（甚至专题讨论会）专门论及如何完成这一任务。很幸运的是（在前面已提及），由于机群设置软件包的引入大大地简化了为机群设置软件的工作，其中包括Oscar（开源机群应用资源，Open Source Cluster Application Resources）软件包，它是菜单驱动式的且是免费使用的。在启动Oscar之前，先要将操作系统（RedHat Linux）安装到主结点上。此后，在若干菜单式驱动步后，Oscar将安装好所需的软件，并完成对机群的配置。简单地讲，在主结点上设置NSF及网络协议，而在数据库中则定义机群。由用户选择IP地址对私有机群网络加以定义，并收集计算结点的以太网接口的MAC地址，这只需每个计算结点向主结点发一个Boot Protocol（自举协议，BOOTP/DHCP）请求就可得到，将为计算结点返回的是IP地址，此外还返回指明要自举哪一部分核心（操作系统的中心部分）的“自举”文件名。然后将核心下载并进行自举，并创建计算结点的文件系统。计算结点上的操作系统是使用Linux安装实用程序LUI通过网络完成的。最后，配置机群并安装和配置中间件。在运行测试程序后，机群就绪。此外为批处理队列、调度及作业监控还提供了工作负载的管理工具。对机群来讲很希望有这些工具。有关Oscar的更多细节可在网站http://www.csm.ornl.gov/oscar上找到。 37

1.5 小结

本章介绍了以下概念:

- 并行计算机及其编程
- 加速比及其他系数
- 由单处理器系统扩展成一个共享存储器多处理机系统
- 消息传递多处理机（多计算机）
- 适用于消息传递多计算机的互联网
- 联网工作站作为并行程序设计平台
- 机群计算

推荐读物

有关多处理机系统内部设计的更多信息，可参见计算机系统结构的教科书，如[Culler and Singh，1999]、[Hennessy and Patterson，2003] 以及 [Wilkinson，1996]。有关互联网技术已发表了大量的著作。关于互联网的更详细的信息可在由[Duato, Yalmanchili and Ni，1997]所编写的很有价值的教科书中找到，该书是一本互联网的专著。早期涉及以太网的论文可参见[Metcalfe and Boggs，1976]。

[Anderson, Culler, and Patterson, 1995]创造了一个实例，即将整个工作站网络用作为一个多计算机系统。基于Web的有关工作站机群研究项目的材料，可在包括http://cesdis.gsfc.nasa.gov/beowulf的网站中找到。在联网工作站上使用共享存储器的例子可在[Amza et al., 1996]中看到。

意识到在工作站机机群中使用商品化接口将使性能受限这一事实，已经导致一些研究人员去设计具有更高性能的网络接口卡（NIC）。关于这一领域的著作包括[Blumrich et al., 1995]、[Boden et al., 1995]、[Gillett and Kaufmann, 1997]以及[Minnich, Burns and Hady, 1995]。[Martin et al., 1997]对机群体系结构中的通信时延、开销和带宽的影响进行了详细的研究。他们得出的一点结论是改善通信系统的通信性能比单纯的加倍机器性能有更好的效果。

由[Buyya, 1999a and 1999b]主编的两卷集为机群提供了信息财富。[Williams, 2001]写了一本极好的侧重网络化的计算机系统体系结构的教科书。有关构建机群的细节可在[Sterling, 2002a and 2002b]中找到。

38

39
ι
40

参考文献

AMDAHL, G. (1967), "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities," *Proc. 1967 AFIPS Conf.*, Vol. 30, p. 483.

AMZA, C., A. L. COX, S. DWARKADAS, P. KELEHER, H. LU, R. RAJAMONY, W. YU, AND W. ZWAENEPOEL (1996), "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer*, Vol. 29, No. 2, pp. 18–28.

ANDERSON, T. E., D. E. CULLER, AND D. PATTERSON (1995), "A Case for NOW (Networks of Workstations," *IEEE Micro*, Vol. 15, No. 1, pp. 54–64.

BLUMRICH, M. A., C. DUBNICKI, E. W. FELTON, K. LI, AND M. R. MESRINA (1995), "Virtual-Memory-Mapped Network Interface," *IEEE Micro*, Vol. 15, No. 1, pp. 21–28.

BAKER, M., AND G. FOX (1999), "Chapter 7 Metacomputing: Harnessing Informal Supercomputers," *High Performance Cluster Computing*, Vol. 1 *Architecture and Systems*, (Editor BUYYA, R.), Prentice Hall PTR, Upper Saddle River, NJ.

BODEN, N. J., D. COHEN, R. E. FELDERMAN, A. E. KULAWIK, C. L. SEITZ, J. N. SEIZOVIC, AND W.-K. SU (1995), "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, Vol. 15, No. 1, pp. 29–36.

BUYYA, R., Editor (1999a), *High Performance Cluster Computing*, Vol. 1 *Architecture and Systems*, Prentice Hall PTR, Upper Saddle River, NJ.

BUYYA, R. Editor (1999b), *High Performance Cluster Computing*, Vol. 2 *Programming and Applications*, Prentice Hall PTR, Upper Saddle River, NJ.

CHANDRA, R., L. DAGUM, D. KOHR. D. MAYDAN, J. MCDONALD, AND R. MENON (2001), *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers, San Francisco, CA.

CONWAY, M. E. (1963), "A Multiprocessor System Design," *Proc. AFIPS Fall Joint Computer Conf.*, Vol. 4, pp. 139–146.

CULLER, D. E., AND J. P. SINGH (1999), *Parallel Computer Architecture A Hardware/Software Approach*, Morgan Kaufmann Publishers, San Francisco, CA.

DALLY, W., AND C. L. SEITZ (1987), "Deadlock-free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. Comput.*, Vol. C-36, No. 5, pp. 547–553.

DUATO, J., S. YALAMANCHILI, AND L. NI (1997), *Interconnection Networks: An Engineering Approach*, IEEE CS Press, Los Alamitos, CA.

FLYNN, M. J. (1966), "Very High Speed Computing Systems," *Proc. IEEE*, Vol. 12, pp. 1901–1909.

FLYNN, M. J., AND K. W. RUDD (1996), "Parallel Architectures," *ACM Computing Surveys*, Vol. 28, No. 1, pp. 67–70.

GILL, S. (1958), "Parallel Programming," *Computer Journal*, Vol. 1, April, pp. 2–10.

GILLETT, R., AND R. KAUFMANN (1997), "Using the Memory Channel Network," *IEEE Micro*, Vol. 17, No. 1, pp 19–25.

GUSTAFSON, J. L. (1988), "Reevaluating Amdahl's Law," *Comm. ACM*, Vol. 31, No. 1, pp. 532–533.

HENNESSY, J. L., AND PATTERSON, D. A. (2003), *Computer Architecture: A Quantitative Approach* 3rd edition, Morgan Kaufmann Publishers, San Francisco, CA.

HOGANSON, K., B. WILKINSON, AND W. H. CARLISLE (1997), "Applications of Rhombic Multiprocessors," *Int.Conf. on Parallel and Distributed Processing Techniques and Applications 1997*, Las Vegas, NV, June 30–July 2.

HOLLAND, J. (1959), "A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously," *Proc. East Joint Computer Conference*, Vol. 16, pp. 108–113.

KERMANI, P., AND L. KLEINROCK (1979), "Virtual Cut-Through: A New Communication Switching Technique," *Computer Networks*, Vol. 3, pp. 267–286.

KNUCKLES, C. D. (2001), *Introduction to Interactive Programming on the Internet using HTML and JavaScript*, John Wiley, New York.

LEIGHTON, F. T. (1992), *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA.

LEISERSON, C. L. (1985), "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Trans. Comput.*, Vol. C-34, No. 10, pp. 892–901.

LI, K. (1986), "Shared Virtual Memory on Loosely Coupled Multiprocessor," Ph.D. thesis, Dept. of Computer Science, Yale University.

MARTIN, R. P., A. M. VAHDAT, D. E. CULLER, AND T. E. ANDERSON (1997), "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," *Proc 24th Ann. Int. Symp. Comput. Arch.*, ACM, pp. 85–97.

METCALFE, R., AND D. BOGGS (1976), "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM*, Vol. 19, No. 7, pp. 395–404.

MINNICH, R., D. BURNS, AND F. HADY (1995), "The Memory-Integrated Network Interface," *IEEE Micro*, Vol. 15, No. 1, pp. 11–20.

NI, L. M., AND P. K. MCKINLEY (1993), "A Survey of Wormhole Routing Techniques in Direct Networks," *Computer*, Vol. 26, No. 2, pp. 62–76.

PACHECO, P. (1997), *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, CA.

SEITZ, C. L. (1985), "The Cosmic Cube," *Comm. ACM*, Vol. 28, No. 1, pp. 22–33.

SINGH, J. P., J. L. HENNESSY, AND A. GUPTA (1993), "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," *Computer*, Vol. 26, No. 7, pp. 43–50.

STERLING, T., editor (2002a), *Beowulf Cluster Computing with Windows*, MIT Press, Cambridge, MA.

STERLING, T., editor (2002b), *Beowulf Cluster Computing with Linux*, MIT Press, Cambridge, MA.

WILLIAMS, R. (2001), *Computer Systems Architecture A Networking Approach*, Addison-Wesley, Harlow, England.

WILKINSON, B. (1990), "Cascaded Rhombic Crossbar Interconnection Networks," *Journal of Parallel and Distributed Computing*, Vol. 10, No. 1, pp. 96–101.

WILKINSON, B. (1991), "Comparative Performance of Overlapping Connectivity Multiprocessor Interconnection Networks," *Computer Journal*, Vol. 34, No. 3, pp. 207–214.

WILKINSON, B. (1991), "Multiple Bus Network with Overlapping Connectivity," *IEE Proceedings Pt. E: Computers and Digital Techniques*, Vol. 138, No. 4, pp. 281–284.

WILKINSON, B. (1992a), "Overlapping Connectivity Interconnection Networks for Shared Memory Multiprocessor Systems," *Journal of Parallel and Distributed Computing*, Vol. 15, No. 1, pp. 49–61.

WILKINSON, B. (1992b), "On Crossbar Switch and Multiple Bus Interconnection Networks with Overlapping Connectivity," *IEEE Transactions on Computers*, Vol. 41, No. 6, pp. 738–746.

WILKINSON, B. (1996), *Computer Architecture Design and Performance*, 2nd ed., Prentice Hall, London.

WILKINSON, B., AND H. R. ABACHI (1983), "Cross-bar Switch Multiple Microprocessor System," *Microprocessors and Microsystems*, Vol. 7, No. 2, pp. 75–79.

WILKINSON, B., AND J. M. FARMER (1994), "Reflective Interconnection Networks," *Computers and Elect. Eng.*, Vol. 20, No. 4, pp. 289–308.

习题

1-1 一个多处理机系统由100台处理器组成，每台处理器的峰值执行速度为2 Gflops。当10%代码为顺序执行，而90%代码可并行化时，该系统以Gflops表示的性能为多少？

1-2 试讨论一个系统的效率（E）是否能大于100%？

1-3 对于给定的搜索的某些部分必须顺序地进行的求解问题，试将阿姆达尔定律方程与1.2.1小节中的超线性加速比分析结合起来，以推导出一个加速比的方程。

1-4 在你的系统上识别主机名、IP地址和MAC地址。确定用于网络的IPv4和IPv6的格式。

1-5 对以下的每个IPv4地址，识别它们的类：

(a) 152.66.2.3

(b) 1.2.3.4

(c) 192.192.192.192

(d) 247.250.0.255

其中只给定类A自0开始，类B从模式10开始，类C从模式110开始，以及类D从模式1110开始（即不要参考图1-22）。

1-6 假定被分配的（IPv4）网络地址是153.78.0.0，它需要6个子网，每个有250个主机。识别该网络的地址类，并为子网和主机划分地址。两者地址必须留出服务器结点。

1-7 欲打造一个由32台计算机所构成的机群。服务器结点有两个以太网连接，一个接到因特网，另一个接到机群。因特网的IP地址是216.123.0.0。用C类格式为此机群设计IP地址的分配。

1-8 一个公司正提议使用512位的IPv8格式。你认为这一提议合理吗？请给出理由。

1-9 物理上构造由消息传递多计算机和共享存储器多处理机两者混合的系统是可能的。请撰写一份论证报告，说明如何加以实现以及该混合系统相对于纯消息传递系统和纯共享存储器系统的优越性体现在什么地方？

1-10 **（研究课题项目）**提出一份有关真正增量式可扩展机群计算机系统的前景报告，该系统可接纳越来越快的处理器而不必丢弃老的处理器。其概念是，系统开始是仅有很少量的现代处理器，此后每年增加少量更新的处理器。这样每一年后，可用的处理器自然就会更好。在适当的时候，将最老的处理器丢弃，但仍保留增加的处理器，因此，系统就永远不会陈旧，且留下的较老的处理器仍可提供有用的服务。关键是如何设计该系统的体系结构以接受更快的处理器和更快的互联部件。另一个问题是，何时丢弃那些较老的处理器。对丢弃处理器和互联部件的最佳时间进行分析。

41

第2章 消息传递计算

在本章中我们将概述消息传递计算的基本概念。我们将介绍消息传递程序的基本结构以及如何对进程之间的消息传递加以说明。我们先是一般地加以讨论，然后我们概述一个特定系统，MPI（message-passing interface，消息传递接口）[⊖]。在本章最后，我们将从理论和实践两方面讨论如何评估消息传递并行程序。

2.1 消息传递程序设计基础

2.1.1 编程的选择

对消息传递多计算机编程可用以下方法进行:

1）设计一种专用的并行程序设计语言。

2）对现有的一种顺序高级语言的语法/保留字加以扩展来处理消息传递。

3）使用现有的一种顺序高级语言，并为它配备一个能进行消息传递的外部过程库。

上述三种方法均有不少例子。也许有关消息传递的专用的并行程序设计语言的唯一一个常用例子是occam语言，它是为transputer的消息传递处理机而专门设计的[Inmos，1984]。并行程序设计的语言扩展的例子有好多个，尽管大都数例子如高性能Fortran（HPF）更适用于共享存储器系统（参见第8章），但Fortran M [Foster，1995]是个例外，它使用显式的消息传递工具。

也可用专门的并行化编译器将以顺序语言（如Fortran）编写的程序转换成可执行的并行代码。多年以前就已提出这种方法，但它通常对消息传递是不适用的，因为传统的顺序程序设计语言没有消息传递这个概念，但在第8章论及共享存储器程序设计时我们仍会对并行化编译器作简单的介绍。

下面我们主要集中讨论第三种选择，即使用通常的高级语言程序设计，例如C，但对它加以扩展,使它能进行消息传递库的调用，以完成进程对进程的直接消息传递。在这种方法中，必须显式地说明要执行哪些进程，何时在并发进程间传送消息，以及传递什么消息。在这种类型的消息传递系统中编程必须使用两个基本方法:

1）创建独立进程使它们能在不同的计算机上执行的方法。

2）发送和接收消息的方法。

2.1.2 进程的创建

在往下讨论之前，让我们复习一下*进程*的概念。在第1章中，为了构造并行程序，引入了术语*进程*。在某些实例中，特别是在不同处理机器上测试程序时，可能会有多个进程映射到单个处理器上。由于该处理器将为这些进程分时共享，因此通常不可能得到最快的执行速度，但它却允许在多处理机系统执行程序之前对程序进行验证。有一种情况是希望构造一个程序使在单处理机上有多个进程运行，这主要是为了隐藏网络的时延（将在2.3.1节中讨论）。无论

⊖ 为本书准备的基于Web的材料中包括了对PVM和MPI两系统的支持。

如何，我们将假设只有一个进程映射到一个处理器，且使用术语*进程*而不是*处理器*，除非要强调处理器的操作时才会使用后一术语。首先必须创建进程并使它们开始执行。

创建进程有两种方法：

1）静态进程创建。

2）动态进程创建。

在*静态进程创建*（static process creation）时，所有进程在执行前必须加以说明，系统将执行固定数目的进程。程序员通常需在进程或程序执行之前用命令行动作显式标识它们。在*动态进程创建*（dynamic process creation）方法中，可在其他进程的执行期间创建进程并启动执行它们。通常用进程创建构造或库/系统的调用来创建进程。当然也可撤销进程。进程创建和撤销可以有条件地进行。此外在执行过程中进程数也可以发生变化。很显然，动态的进程
43 创建比起静态的进程创建来是功能更强大的技术，但在创建进程时它会导致显著的开销。*进程创建*常遭致某种误解，因为实际上不论在什么情况下，进程代码必须在任何进程执行之前完成编写和编译。

在大多数应用程序中，进程既不会全相同也不会全不同；通常总是有一个称为“主进程”（master process）的控制进程，而其余的进程则为“从”进程或“工作者”，这些进程在形式上相同，差别仅在于它们的进程标识符（ID）不同。进程的ID可用来修改进程的动作或为消息计算不同的目的地。进程由程序员所编写的程序所定义。

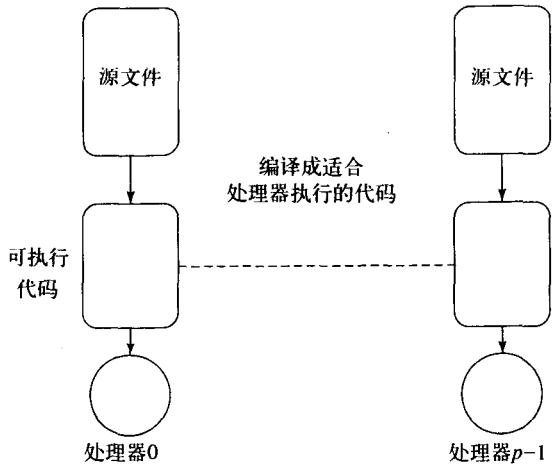


图2-1　多程序多数据（MPMD）模型

最通用的程序设计模型是*多程序多数据*（multiple-program, multiple-data, MPMD）模型，在这种模型中，需为不同的处理器编写完全独立和不同的程序，如图2-1所示。然而如前所述，通常只需编写两个不同的程序，即一个主程序和一个从程序就足够了。由一个处理器执行主程序，而由多处理机执行相同的从程序。通常即使从程序是相同的，进程的ID可用来定制执行——例如指明所生成消息的目的地。

特别地对于静态进程的创建，使用所谓的*单程序多数据*（SPMD, single-program, multiple-data）模型就更为方便。在SPMD模型中，不同的进程将被融合到一个程序中。在该程序中将由控制语句为每个进程选择程序的不同部分。在用控制语句分离每个进程所要完成动作的源程序构成之后，就将此源程序编译成每个处理器可执行的代码，如图2-2所示。每个处理器将装载该代码的拷贝到它的本地存储器中以便执行并且所有处理器可以同时开始执行这些代码。如果处理器的

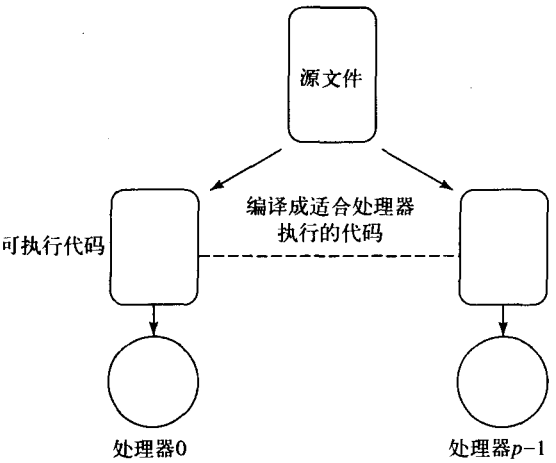


图2-2　单程序多数据模型（SPMD）

类型不同，那么就需为每种不同类型的处理器将源代码编译成可执行代码，而每个处理器必须装载附合自己类型的代码以便执行。稍后（在2.2.2节中）我们将更详细地叙述SPMD的程序设计，因为它是最常用的消息传递系统之一——MPI中采用的主要方法。 44

对于动态进程的创建，可以编写两个不同的程序，即一个主程序和一个从程序，对它们分别编译并准备执行。用库调用实现动态进程创建的例子可为如下形式：

```
spawn(name_of_process);
```

它将立刻启动另一个进程[⊖]，此后调用进程和被调用进程两者就一起向前执行，如图2-3所示。正被“派生”的进程实际上是一个事先编译好的、且可执行的程序。

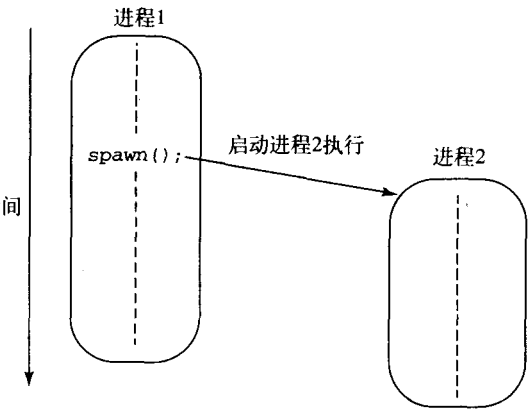


图2-3 派生一个进程 45

2.1.3 消息传递例程

1. 基本的发送和接收例程

发送和接收消息传递库的调用常用以下形式：

```
send(parameter_list)
recv(patameter_list)
```

其中send()出现在源进程中，由它发送消息，而recv()则出现在目的进程中以收集已被发出的消息。括号中的实参（实际参数）依赖于软件，在某些情况下可能较为复杂。在send()中最简单的参数将是目的ID和消息，而在recv()中则为源ID和收到消息的地点名。用C语言时，在源进程中我们可用以下的调用：

```
send(&x, destination_id);
```

在目的进程中则有以下的调用：

```
recv(&y, source_id);
```

这样就可将源进程中的数据x发送到目的进程的y处，如图2-4所示。具体的参数安排次序依赖于系统。本书采用的顺序是，在数据之后指明进程识别，并使用&后跟一个单个数据元素说明对指针的调用。在本例中，在x中必须预先装载要发送的数据，且x和y必须有相同类型和大小。通常我们要发送的是比单个数据元素更为复杂的消息，因而需要功能更强大的消息格式。有关实际的消息传递调用参数的确切细节和变化，将在2.2节中介绍，但在此之前我们要详尽阐述有关它的基本机制。为提高代码效率和增强灵活性，为发送/接收例程提供了各种机制。

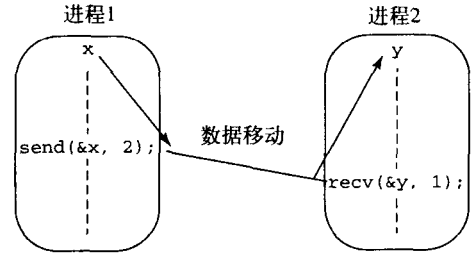


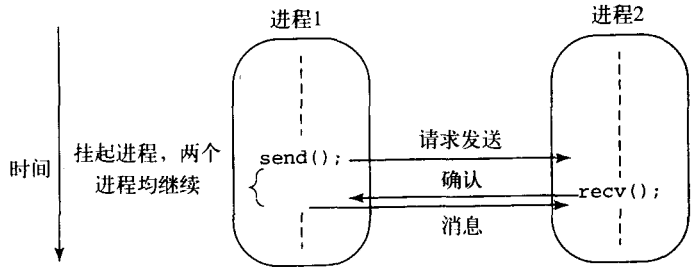
图2-4 使用库调用send()和recv()在进程间传送一个消息

（1）同步消息传递 术语同步（synchronous）用于在消息传递已经结束时确实已返回的例程。一个同步发送例程在返回之前将处于等待状态，直至它所发送的完整消息已被接收进程接收。一个同步接收例程在返回之前将处于等待状态，直至它所期待的消息已经到达和被存储。由一个同步发送操作和一个与之相匹配的同步接收操作所组成的一对进程同步时，在未完成将消息从源进程传送给目的进程之前，它们之中的任一个都不能继续执行。因此，同

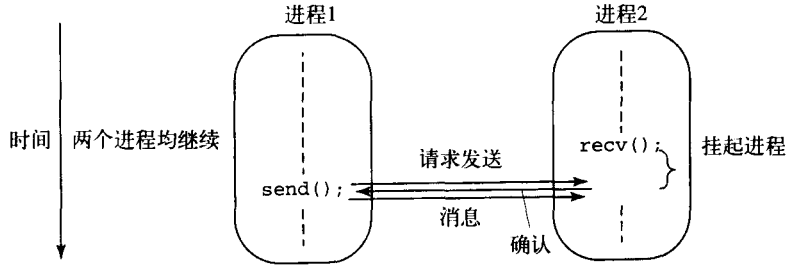
⊖ Courier字样用来突出代码，该代码可以是伪代码或是使用特殊语言或系统的代码。

步例程本质上要完成两个动作：传送数据和同步进程。常用术语会合（rendezvous）来描述通
46 过同步发送/接收操作实现两个进程会合和同步。

同步发送和接收操作不需要消息缓冲存储器。它们暗示需要有某些形式的信号量，如在三路协议中，由源进程首先向目的进程发送一个“请求发送”消息，当目的进程准备接受该消息时，就向源进程返回一个确认消息。一旦收到该确认，源进程就发送实际消息。使用三路协议的同步消息传递如图2-5所示。在图2-5a中，进程1在进程2到达相应的recv()之前到达它的send()。在某种状态下，进程1必须挂起直至进程2到达它的recv()。此时进程2必须以某种“信号量”形式唤醒进程1，此后进程1和进程2双方就可参与消息传送。应注意在图2-5a中，消息必须在源进程中加以保留，直至可被发送为止。在图2-5b）中，进程2在进程1到达其send()之前到达recv()。现在轮到进程2必须挂起，直至双方都能参与消息传递为止。至于挂起和唤醒进程的具体机制则与系统相关。



a）send()出现在recv()之前



b）recv()出现在send()之前

图2-5　使用三路协议的同步send()和recv()库调用

（2）阻塞和非阻塞消息传递　术语阻塞（blocking）原先也用来描述在传送完成之前不允许进程继续执行的例程。例程“被阻塞”而不能继续向下执行。就此而言，术语同步（synchronous）和阻塞是同义的。术语非阻塞（nonblocking）用来描述不管消息是否已收到，
47 例程马上返回的工作方式。但是术语阻塞和非阻塞在如MPI那样的系统中已被重新定义。稍后我们将考察有关精确的MPI定义，但现在让我们简单叙述在消息传递结束之前，消息传递例程如何能实现返回。一般而言，在源和目的之间需要有消息缓冲区（message buffer）来保存消息，如图2-6所示。这里的消息缓冲区是用来保存在recv()接收之前欲发送的消息，对接收例程来讲，如果需要该消息，该消息就必须要被接收。如果recv()先于send()到达，则消息缓冲区将是空的，而recv()就需要等待消息。但对发送例程来讲，一旦本地操作已经结束且消息已安全上路，则该进程就可继续执行后继工作。采用这种工作方式的发送例程可减少整个的执行时间。实际上，由于缓冲区的大小是有限的，因而完全可能在某一时间点，因为已用完了所有可用的缓冲区空间发送例程将受阻。在某一时间点上可能必须了解所发送

的消息是否确实已被接收，当然这就需要额外的消息传递。

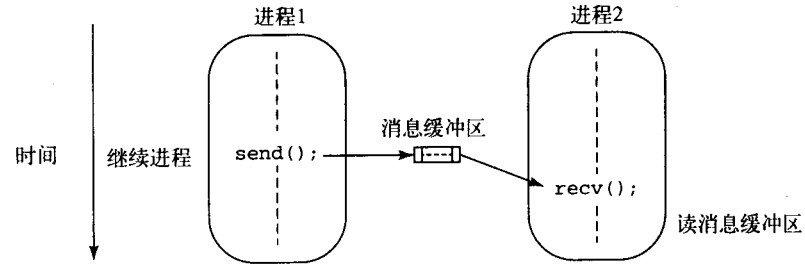


图2-6 消息缓冲区的使用

我们将采用与MPI相一致的术语的定义——称使用消息缓冲区并在它们的本地操作完成后就返回的例程是阻塞的，或更精确地说是本地阻塞的，虽然此时消息传递可能还未完成。而称那些立即返回的例程为非阻塞的。在MPI的非阻塞例程中，假设用于传送的数据存储器在数据存储器用于传送之前不能为后继的语句所修改，而这一点要由程序员设法保证。术语同步用来描述直到发送例程和接收例程都已到达且消息已从源向目的传送时，发送例程和接收例程才可返回的这样一种情况。在本书的大部分代码中，只需使用（本地）阻塞和同步。

2. 消息选择

到目前为止，我们已叙述了如何从一个特定的源进程向一个特定的目的进程发送消息，这里的目的进程的ID在发送例程中是作为一个参数给定的，而源进程的ID则是在接收例程中作为一个参数给定的。目的进程中的`recv()`将只接收`recv()`中以参数指明的源进程所发来的消息，对其他消息将不予接收。可以在源ID处使用一个特殊符号或数字作为源ID的通配符，以允许目的进程接收来自任何源进程的消息。例如可用-1作为源ID的通配符。

为了提供更大的灵活性，可用附于消息的消息标记（message tag）对发送来的消息加以选择。消息标记`msgtag`是一个典型的由用户选定的正整数（包括零），用它对发送的不同类型的消息加以区分。然后可制作特定的接收例程来接受具有特定消息标记的那些消息，而忽略其他消息。因此消息标记将成为`send()`和`recv()`中的一个附加参数，通常该参数紧跟在源/目的标识之后。例如，要从源进程1向目的进程2发送消息标记为5的消息`x`，并将其赋值给`y`，则在源进程中我们可用

48

```
send(&x, 2, 5);
```

而在目的进程中可用

```
recv(&y, 1, 5);
```

消息标记是在消息内自带的。如果不需要特殊类型的匹配，就可用通配符代替消息标记，这样`recv()`就可与任何`send()`相匹配。

消息标记的使用非常普及。但它需要程序员跟踪在程序中和任何由别人编写的内藏程序部分或是所调用的库例程中所使用的消息标记号。为区分是在内藏程序或库例程中或是在用户进程中所发送的消息，需要功能更强的消息选择机制。我们将在稍后阐述这种机制。

3. 广播，集中和分散（Broadcast ,Gather and Scatter）

通常还有许多其他的消息传递和有关的例程，它们将提供我们所期望的一些特性。对拥有消息源的进程通常要求它能将相同消息发送给多个目的进程。术语广播（broadcast）是指向所有与求解问题有关的进程发送相同的消息。术语多播（multicast）用于描述将相同消息发送给已定义的进程组中的每个进程。但是本书将不对两者加以区别，而是简单地统称它们为广播。

图2-7对广播的含义做了说明。必须对参加广播的所有进程加以标识，典型的做法是首先生成一个已命名的进程组，然后在广播例程中将其作为一个参数。在图2-7中，在广播例程的
49 参数中将进程0标识为*根进程*（root process）。实际上组中任何进程都可以被指定为根进程。在本例中，根进程在buf中保存要广播的数据。图2-7指明每个进程均将执行相同的bcast()例程，对SPMD模型来讲这种安排非常方便，因为该模型中的每个进程具有相同的程序。图2-7还指明了根进程本身也接收数据（这是一种用在MPI的安排），但是否要这样做，则依赖于具体的消息传递系统。对于MPMD模型，另一种安排是由源进程执行广播例程，而目的进程只执行常规的消息传递接收例程。在这种情况下，根进程就不会接收该数据，这种接收实际上是不必要的，因为根进程已经有了该数据。

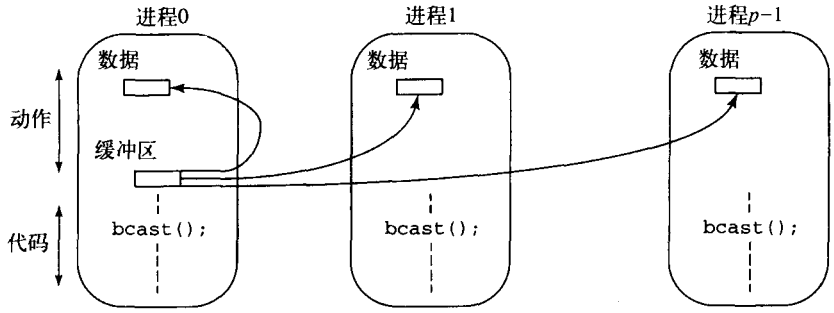


图2-7　广播操作

如前所述，仅在所有进程均已执行了它们的广播例程后，广播的动作才会出现，因而广播操作具有同步进程的作用。广播的具体实现将依赖于软件和底层的体系结构。在本章的后面将阐述这种实现。应该认识到，对在程序中广泛使用的广播进行有效的实现是非常重要的。

术语分散（scatter）被用于描述根进程中数据数组中的每个元素被分别发送给各个进程。数组中第i个单元的内容将被发送到第i个进程。与广播将相同数据发送给一组进程不同，分散将分配不同的数据元素给进程。广播和分散是两个常用的程序启动需求以发送数据给从进程。图2-8对分散作了说明。如同广播一样，需要标识一个进程组和根进程。在本例中，根进程也将接收一个数据元素。图2-8中指明每个进程将执行相同的scatter()例程，同样对SPMD模型来讲是很方便的。

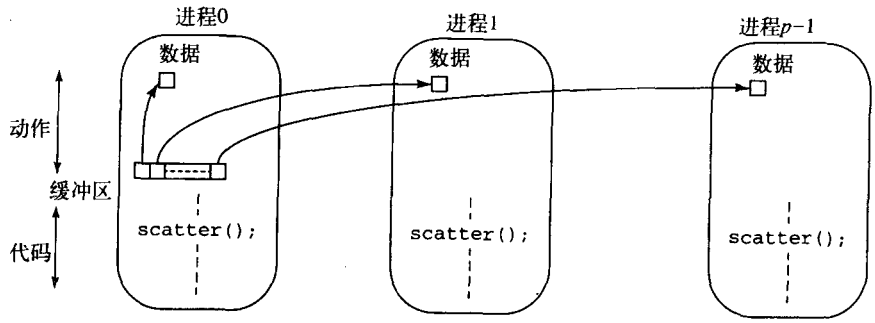


图2-8　分散操作

术语集中（gather）是指一个进程从一组进程中的每一个进程处收集一个值。集中通常使用在组中进程均已完成某种计算后的场合。集中实质上是分散的逆操作。根进程接收来自第i个进程的数据，并将它放到用来接收数据的数组中的第i个单元处。图2-9对集中作了说明。该例中的进程0为集中的根进程。同样，该例中的根进程也参加集中操作。

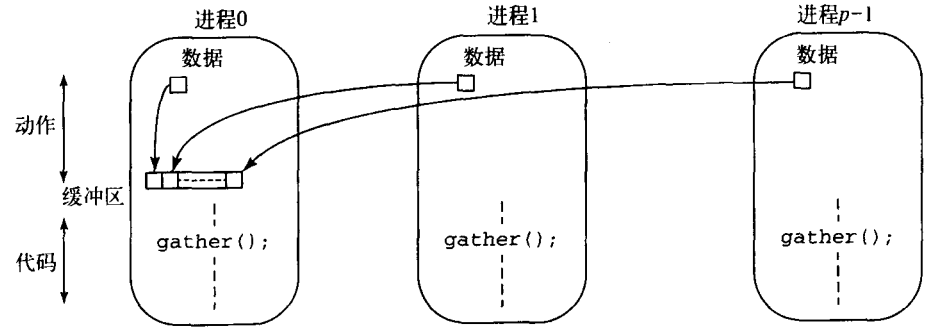


图2-9 集中操作

有时集中操作可与一个指定的算术或逻辑操作组合在一起。例如，可将各个值进行集中，然后再由根进程将它们相加，如图2-10所示。根进程也可完成其他的算术/逻辑操作。所有这些操作有时称为归约（reduce）操作。大多数消息传递系统支持这些操作和其他相关操作。而归约操所实现的具体方法完全依赖具体实现方案。在根中进行集中式操作不是唯一的解决方法。可以将部分操作分布在其他进程中完成。不论是什么样的实现，其基本思想是要使公共集合操作的实现尽可能高效。 50

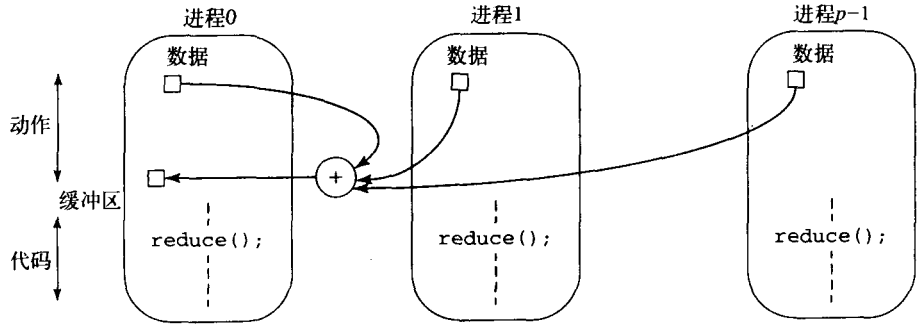


图2-10 归约操作（加）

2.2 使用计算机机群

2.2.1 软件工具

现在让我们针对计算机机群（机群计算）叙述基本的消息传递概念。已经有几种机群计算（早期被称为工作站网络）的并行程序设计软件包。也许第一个被广泛接受的、将工作站网络作为多计算机平台的并行程序设计软件是PVM（并行虚拟机），它是由美国Oak Ridge国家实验室在20世纪80年代后期开发并在20世纪90年代被广泛使用。PVM为同构或异构计算机间的消息传递提供了一个软件环境，它有一个库例程集，用户可在C或FORTRAN程序中调用它们。PVM已被广泛使用，部分原因是它是免费的且很容易从网站上下载，下载网址为http://www.netlib.org/pvm3。现在也可使用Windows实现。除了PVM外，也有由IBM等厂商为特殊系统所开发的专利消息传递库。但正是PVM在20世纪90年代初期使得人们能实际地使用工作站网络进行并行程序设计。 51

2.2.2 MPI

为促进更广泛地使用和可移植性，学术界和产业界的合作伙伴组决定联合研发一种他们

希望成为“标准”的消息传递系统。他们称其为MPI（Message-Passing Interface，消息传递接口）。MPI为消息传递和相关操作提供库例程。MPI的基本特征是，它定义的是一个“标准”而不是具体实现。这犹如程序设计语言的定义，而不管该语言的编译器是如何实现的。MPI有大量的例程（超过120个，这个数量还在增长），当然我们讨论的仅是它们的一个子集。开发MPI的重要因素是期望消息传递可移植且易于使用。MPI中也做了某些改变以改正早期如PVM那样消息传递系统中的一些技术上的缺陷。MPI第一个版本“版本1.0”是在经过两年的会议和讨论于1994年5月最后确定的。版本1有意识地删去了某些高级的或有争议的特性，而在以后的版本中加入了这些特性。版本1.2是版本1.0的增强版。在1997年又推出了版本2.0（MPI-2），增加了动态进程创建、单边操作和并行I/O。

实际上版本1中所含有的大量函数是应用程序员为了能编写高效的代码所期望的特征。但是，只需使用可用函数中的一个很小子集就可编写程序。[Gropp，Lusk and Skjellum，1999a]提及只需使用120多种函数中的6个函数就可成功地编写程序，我们要讨论的内容将多于6个“基本”函数。C和Fortran语言均可调用函数，我们只讨论C的版本。所有的MPI例程以前缀`MPI_`开始，且下划线后的第一个字母必须是大写的。通常，例程返回的信息包括成功调用和失败调用。这些信息可以在[Snir et al.,1998]中找到，在此就不赘述了。

已经有了好几种MPI标准的免费实现，其中包括美国Argonne国家实验室和密西西比（Mississippi）州立大学的MPICH，俄亥俄超级计算中心（Ohio Supercomputing Center）的LAM（现由Notre Dame大学维护）。此外还有HP、IBM、SGI、SUN以及许多其他供应商的实现。对Windows机群的实现也已出现。在网址http://www.osc.edu/mpi和http://www.erc.msstate.edu/misc/mpi/implementations.html上可找到有关MPI实现和其出处的列表。选择一个实现的关键因素是持续维护，因为少数早期的实现现在已完全不再维护。判断是否维护的一个好的标志是看该实现是否包含MPI的最近版本的特征（目前为MPI-2）。可用的实现特征可在网址
52 http://www.erc.msstate.edu/misc/mpi/implementations.html上找到(在写本书时共列出24个实现)。大多数实现，如果不是全部，并没有包括MPI-2的每个特性。例如，在写本书时MPICH完全不支持MPI的单边通信。MPI-2的扩展集合操作仅在两个实现中得到了支持（日立和NEC公司的商业实现）。这种不完全的支持在编写当前的程序时可能会出现问题，特别是非常有用的单边通信这一特征。

1. 进程创建和执行

如同一般的并行程序设计，要将并行计算分解成若干并发进程。在MPI标准中，有意不对创建和启动MPI进程加以定义，它们将依赖于实现。在MPI版本1中只支持静态的进程创建。这就是说，所有进程在执行前必须加以定义，且它们必须一起启动。MPI版本2中作为先进特征引入了动态进程创建，并设有派生例程`MPI_Comm_spawn()`。即使如此用户仍可能不用它，因为动态进程的创建会带来开销。

使用SPMD计算模型可只编写一个程序但由多处理器来执行。启动不同程序的方法由具体实现确定。通常启动可执行的MPI程序是通过命令行来实现的。例如，相同的可执行程序可同时在四个处理器上被启动通过：

```
mpirun prog1 -np 4
```

或是

```
prog1 -np 4
```

这两个命令没有说明`prog1`的这些拷贝将在何处执行。再强调一次，在MPI标准中并不定义如何将进程映射到处理器上。指定的映射也许可用命令行来完成或是使用一个文件来完成，

该文件中含有可执行程序的名称以及指定的用来运行每个可执行程序的处理器。MPI支持拓扑的定义（网格等），因此它具有自动映射的潜能。

在调用任何MPI函数前，必须用MPI_Init()例程将代码初始化，而在所有MPI函数调用后，代码必须以MPI_Finalize()例程结束。命令行的参数将传送给MPI_Init()以允许MPI进行设置将要发生的动作，即

```
main (int argc, char *argv[])
{
   MPI_Init(&argc, &argv);                          /* initialize MPI */
      .
      .
      .
   MPI_Finalize();                                  /* terminate MPI */
}
```

（如同在顺序C程序中，&argc为参数计数，提供参数个数，而&argv为参数向量，它是指向字符串数组的指针）。

开始时，所有进程进入一个称为MPI_COMM_WORLD的“宇宙”，且为每个进程给定一个独特的序号，如果有p个进程，则序号从0到$p-1$。在MPI术语中，MPI_COMM_WORLD是一个通信子（communicator），由它定义通信操作的作用域（scope），进程使序号与通信子相对应。为了建立进程组可使用其他的通信子。对于单个程序，使用默认的MPI_COMM_WORLD通信子就足够了。然而，通信子的概念允许程序，特别是库，构造成对于每个消息都有独立的作用域。 53

2. 使用SPMD计算模型

当每个进程实际执行相同代码时，SPMD将是一个理想的模型。但一般而言，所有应用中的一个或多个进程常需要执行不同代码。为在单个程序中方便地实现这一点，就需要在代码中插入一些语句来选择每个处理器执行哪一部分代码。因而，SPMD模型不排斥主－从方法，而只是必须将主代码和从代码放在同一程序中。下面的MPI代码段说明如何可做到这一点：

```
main (int argc, char *argv[])
{
   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);          /* find process rank */
   if (myrank == 0)
      master();
   else
      slave();

   MPI_Finalize();
}
```

其中的master()和slave()是由主进程和从进程分别执行的过程。该方法也可用于多于两个代码序列的情况。如果每个进程要执行完全不同的代码，那么SPMD模型在存储器需求方面将是低效的，但幸运的是不大会有这种需求。SPMD模型的一个优点是可将它的命令行参数传递给每个进程。

给定一个SPMD模型，任何全局变量声明将在每个进程内被复制。不被复制的那些变量需要局部地加以声明，也就是说，在仅由该进程执行的代码内部声明。例如：

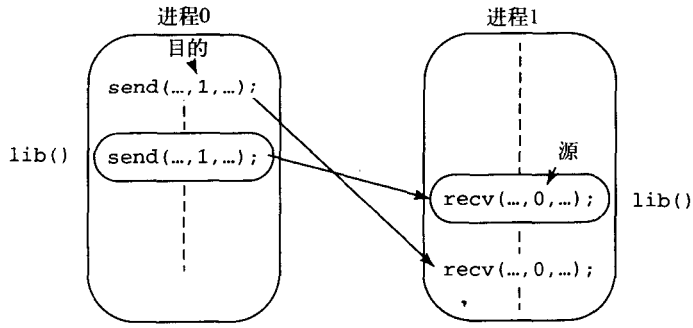
```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* find process rank */
if (myrank == 0) {              /* process 0 actions/local variables */
   int x, y;
      .
      .
      .
} else if (myrank == 1) {       /* process 1 actions/local variables */
   int x, y;
      .
      .
      .
}
```

其中，进程0中的x和y与进程1中的x和y是不同的局部变量。但是这种声明在C语言中是不欢迎的，因为C语言中的一个变量的作用域是从它的声明开始到程序或函数的结束，而不是声明到当前块的结束，要实现后者应在块内声明这些变量。在大多数情况下，应在程序的顶部声明所有的变量，然后为每个进程复制一份，实质上它们就成为每个进程的局部变量。

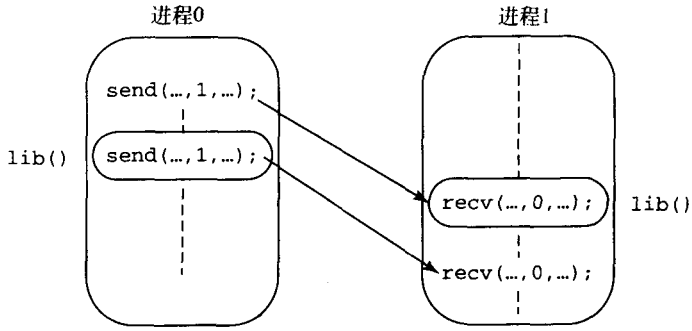
54

3. 消息传递例程

消息传递通信是错误操作的源头。MPI的一个目的是为用户提供一个安全的通信环境。图2-11表示了一个不安全通信的例子。在图2-11中，进程0欲向进程1发送一个消息，但如图2-11中所示，在库例程间也有一个消息传递。尽管每对send/recv将源和目的进行了匹配，但不正确的消息传递仍会发生。通配符的使用更增加了出现错误操作或死锁的可能。假定在一个进程中，一个非阻塞接收在消息标记和源字段中都使用了通配符。此时若有另一对进程调用库例程要求进行消息传递。那么，在此库例程中的第一次发送可能与使用通配符的非阻塞接收相匹配，引发一个错误动作。



a）期望的行为



b）可能的行为

图2-11　带有库的不安全消息传递

MPI为所有点对点和集合的MPI消息传递通信使用了通信子。一个通信子是一个定义了一组只允许组内进程相互通信的进程的通信域（communication domain）。用这种方法，就可使库的通信域与用户程序分隔开。在通信子中，每个进程有一个序号，它是从0到$p-1$的一个整数，其中p是进程数。实际中有两类通信子可以使用，一类是组内通信的内通信子（intracommunicator），另一类是组间通信的外通信子（intercommunicator）。组用来定义参与通信的进程集合。组内每个进程有一个唯一的序号（从0到$m-1$的一个整数，其中m是组中进程数），一个进程可以同时是多个组的成员。对于简单的程序只使用内通信子，因此就不需要组的附加概念。 55

MPI中有一个默认的内通信子MPI_COMM_WORLD，它作为应用程序中所有进程的第一个通信子存在。在简单的应用中，不需要引入新的通信子。MPI_COMM_WORLD可在所有点对点和集合操作中使用。新的通信子要在已有的通信子上创建。MPI中有一组例程，专门用来在已有的通信子上生成新的通信子（以及从已有的组中创建新的组），请参见附录A。

（1）点对点通信 消息传递是由常见的发送和接收调用来完成的。需使用消息标记，可使用通配符MPI_ANY_TAG代替消息标记，也可用通配符MPI_ANY_SOURCE在接收例程代替源ID。

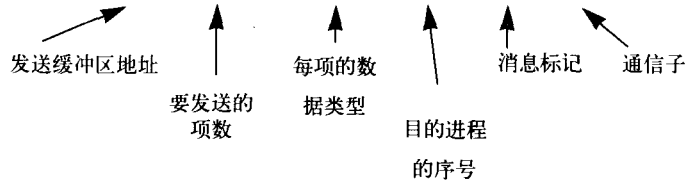
消息的数据类型在发送/接收参数中加以定义。数据类型可从标准MPI数据类型表中选择（MPI_INT，MPI_FLOAT，MPI_CHAR等）或由用户创建。用户定义的数据类型是从现有的数据类型中派生出来的。用这种方法，用户可创建一个数据结构以表示具有任何复杂性的消息。例如，可以创建一个由两个整数和一个浮点数组成的数据结构，这样就可在一个消息内就将它们发送出去。除了取消较早的PVM消息传递系统中对打包/解包例程的需要之外，已经声明的数据类型还具有数据类型可重用的优点。另外，它不需要显式的发送和接收缓冲区，这对减少大型消息所需的存储容量需求特别有用；消息不需要从源单元拷贝到一个显式的发送缓冲区中，拷贝到显式的发送缓冲区意味着需要两倍的存储空间以及时间开销。（如果需要的话，MPI设有提供显式缓冲区的例程。）

（2）完成 有若干种关于接收和发送的版本。可使用局部完成（locally complete）或全局完成（globally complete）来描述这种版本的变化。我们说一个例程已完成了操作中所有它的部分则该例程是局部完成的。如果该操作涉及到的所有例程都已完成了该操作中它们自己的部分且操作已全部进行则称例程是全局完成的。

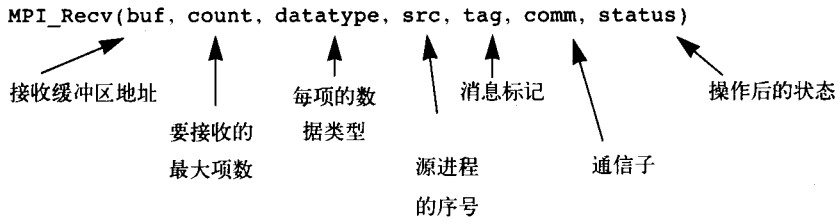
（3）阻塞例程 在MPI中，阻塞例程（发送或接收）在它们局部完成后就可返回。阻塞发送例程的局部完成条件是，保存消息的单元可再次为其他语句或例程所使用或是其内容改变但不会影响消息的发送。阻塞发送将发送消息并返回。但这并不表明消息已被接收，而只是指进程可以自由地继续执行而不会对消息产生负面影响。实质上是将源进程阻塞了一段最小时间，以在这段时间内对该数据进行访问。当一个阻塞接收例程局部完成时它也将返回，在这种情况下，这意味着该消息已被接收到目的单元，并可以读出该目的单元。 56

阻塞发送例程参数的一般格式如下：

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```



阻塞接收例程参数的一般格式为：



注意，在MPI_Recv()中对最大的消息尺寸作了说明。如果被接收的消息大于最大尺寸，将发生溢出错误。如果接收的消息小于最大尺寸，消息将被存储在缓冲区的前部，而其余的单元将不受影响。尽管我们通常期望发送已知尺寸的消息。

例　从进程0向进程1发送一个整数x。

```
int x;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);          /* find process rank */
if (myrank == 0) {
   MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
   MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

(4) *非阻塞例程*　非阻塞例程将立即返回；也就是说，无论例程是否已局部完成都允许继续执行下一条语句。非阻塞发送MPI_Isend()（其中I表示立即（immediate））将在源单元可安全地改变之前就可返回。非阻塞接收MPI_Irecv()即使没有接收消息也将立即返回。它们的格式如下：

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request)
MPI_Irecv(buf, count, datatype, source, tag, comm, request)
```

可以分别用MPI_Wait()和MPI_Test()例程来检查发送或接收操作是否已经完成。MPI_Wait()将一直等待直至操作已确实完成，然后再返回；而MPI_Test()则立即返回，
57 并以标记置位来指明在那一时刻操作是否已经完成。这些例程需与一个特定操作相关，通过使用相同的request参数就可作到这一点。非阻塞接收例程提供了在等待消息到达期间进程继续进行其他活动的能力。

例　从进程0向进程1发送一个整数x，并允许进程0继续执行。

```
int x;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);          /* find process rank */
if (myrank == 0) {
   MPI_Isend(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
   compute();
   MPI_Wait(req1, status);
} else if (myrank == 1) {
   MPI_Recv(&x, 0, MPI_INT, 1, msgtag, MPI_COMM_WORLD, status);
}
```

4. 发送通信方式

MPI发送例程可为四种通信方式中的一种，它们定义了发送/接收协议。这四种方式为*标准*（standard）、*缓冲*（buffered）、*同步*（synchronous）和*就绪*（ready）。

在*标准方式*发送中，并不假设相应的接收例程已经启动。如果有缓冲的话，缓冲区的大小依赖具体实现，MPI并未对其定义。如果提供缓冲的话，发送在接收到达前就可完成（如

果是非阻塞的，则当匹配了MPI_Wait()或MPI_Test()时返回便完成)。

在*缓冲方式*中，发送可在匹配接收之前就启动和返回。对于这种方式，必须在应用程序中提供指定的缓冲区空间。缓冲区空间可借助MPI例程MPI_Buffer_attach()提供给系统，并可用例程MPI_Buffer_datach()收回。

在*同步方式*中，发送和接收两个例程中任一个例程均可在另一个例程之前启动，但两者只能一起完成。

在*就绪方式*中，仅当与匹配接收例程已经到达时，发送才可开始，否则将出错。使用这种方式时必须非常谨慎，以免错误操作。

对于阻塞和非阻塞的发送例程，这四种方式均可使用。对其中的三种非标准方式在助记符中用一字母加以标识（缓冲式用b、同步式用s、以及就绪式用r)。例如，MPI_Issend()是一个非阻塞同步发送例程。这是一种罕见但却很有价值的组合。发送将立即返回，因此它不会与含有与之匹配接收的进程进行直接地同步，消息传送将假设在某一点完成，如同所有立即方式的例程一样，通过使用MPI_Wait()或MPI_Test()就可以确定该点。这就允许例如记录同步进程要花多长时间，或是确定是否存在缺少缓冲存储的问题。也存在一些不允许的组合。对于阻塞和非阻塞接收例程只可以使用标准方式，且并不假设相应的发送已经开始。任何类型的发送例程可以与任何类型的接收例程一起使用。

58

5. 集合通信

与点对点通信只涉及一个源进程和一个目的进程不同，*集合通信*（Collective Communication)（如广播通信）将涉及一组进程，这些进程是指那些由内通信子所定义的那些进程，它们不需要消息标记。

广播集中和分散例程 MPI提供了一个广播例程和一组集中和分散例程。通信子定义了将参与该集合操作的进程集合。操作数据的主要集合操作如下:

MPI_Bcast()	从根结点向所有其他进程广播
MPI_Gather()	为进程组集中值
MPI_Scatter()	将缓冲区中的部分值分散给进程组
MPI_Alltoall()	从所有进程向所有进程发送数据
MPI_Reduce()	将所有进程的值组合成一个值
MPI_Reduce_scatter()	组合值并分散结果
MPI_Scan()	在各个进程上计算前缀数据归约

上述各例程所涉及的进程是指在同一通信子中的进程。此外上述例程还有一些变形形式。有关各例程中的参数的细节可参见附录A。

例 为了从进程组中集中数据项到进程0，可在根进程中对存储器进行动态分配，我们可使用如下代码:

```
int data[10];                    /*data to be gathered from processes*/
      .
      .
      .

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);              /* find rank */
if (myrank == 0) {
   MPI_Comm_size(MPI_COMM_WORLD, &grp_size);         /*find group size*/
   buf = (int *)malloc(grp_size*10*sizeof(int));     /*allocate memory*/
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0, MPI_COMM_WORLD);
```

应注意的是，MPI_Gather()从所有进程处集中数据项，包括根进程。

障栅　如同与所有消息传递系统一样，MPI提供了一种同步进程的方法，它将停止每个进程直至所有进程都已到达指定的“障栅”调用。在第6章研究同步计算时，我们将更详细地叙述障栅同步。

6. MPI程序实例

图2-12中示出了一个简单的MPI程序。该程序的功能是将一组数相加。这些数是随机生成的，且保存在一个文件中。该程序可在网址http: //www.cs.uncc.edu/par_prog中找到，它可用来熟悉软件环境。

59

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000

void main(int argc, char **argv)
{
  int myid, numprocs;
  int data[MAXSIZE], i, x, low, high, myresult, result;
  char fn[255];
  char *fp;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  if (myid == 0) {            /* Open input file and initialize data */
     strcpy(fn,getenv("HOME"));
     strcat(fn,"/MPI/rand_data.txt");
     if ((fp = fopen(fn,"r")) == NULL) {
        printf("Can't open the input file: %s\n\n", fn);
        exit(1);
     }
     for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d", &data[i]);
  }

/* broadcast data */
   MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);

/* Add my portion Of data */
   x = MAXSIZE/numprocs;          /* must be an integer */
   low = myid * x;
   high = low + x;
   myresult = 0;
   for(i = low; i < high; i++)
      myresult += data[i];
   printf("I got %d from %d\n", myresult, myid);

/* Compute global sum */
   MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   if (myid == 0) printf("The sum is %d.\n", result);

   MPI_Finalize();
}
```

图2-12　MPI程序实例

2.2.3　伪代码构造

在前几小节中，我们已看到了用来实现基本消息传递的特定的MPI的例程。有时有许多参数需要使用附加的代码，而且常常会涉及许多其他方面的细节。例如，可能要在其中设置错

误检验代码。在C语言中，几乎所有MPI例程在错误的事件中将返回一个整数错误代码[⊖]；在C++中，抛出异常并提供错误处理程序。在任何事件中，从错误类表中可识别此代码并采取适当的动作。这种附加的代码虽然在构成结构化良好的程序时是需要的，但实际上将有损于程序的可读性。我们将使用伪代码而不使用真实代码来描述算法。我们的伪代码将略去那些凌乱的参数，这些参数对理解代码只具有辅助作用。 60

进程标识将放在参数表中的最后位置（如MPI中那样）。要从称为master（主）的进程向称为slave（从）的进程发送由一个整数x和一个浮点数y组成的消息，并将它们赋值给a和b，则我们可很简洁地将主进程写为

```
send(&x, &y, P_slave);
```

以及将从进程写为

```
recv(&a, &b, P_master);
```

其中x和a声明为整数，而y和b声明为浮点数。整数x将被拷贝到a，而浮点数y被拷贝到b（注意，这里允许灵活地说明具有不同数据类型的多个数据项；在实际代码中，可能需要使用独立的例程进行说明或创建数据类型）。我们保留了&符号以指明数据参数是指针（至少对recv()必须如此，对于发送数组也需如此）。相应地，第i个进程将以记号P_i表示，而标记将在源或目的名后面出现；即

```
send(&x, P_2, data_tag);
```

将把x发送给进程2，且它的消息标记为data_tag。相应的接收例程必须有相同的标记（或通配符标注）。有时需要定义更复杂的数据结构，在集合通信例程中还需要有附加的说明。

在我们的伪代码中，所需的基本消息传递例程的最常用的形式是本地阻塞send()和recv()，它们可写成如下形式：

```
send(&data1, P_destination);                /* Locally blocking send */
recv(&data1, P_source);                     /* Locally blocking receive */
```

在许多实例中，只使用本地阻塞的方式就足够了。其他的各种方式将以前缀加以区别：

```
ssend(&data1, P_destination);               /* Synchronous send */
```

实际上，除了消息传递例程外，所列出的代码段均是以标准C语言表示的，尽管它不必是最优化或最精确的方式。例如，除了循环计数器外，为清晰起见，我们不使用压缩形式的赋值语句（如x += y;）。在伪代码表示中，允许采用某种艺术性的表述。指数以通常的数学方式表示，一般不给出变量的初始化等等。但是，将伪代码翻译成以MPI或任何其他的消息传递“语言”所表示的实际消息传递代码是直截了当的。 61

2.3 并行程序的评估

在以后的几章中，我们将叙述实现并行化的各种方法，为此我们需要对这些方法进行评估。作为准备知识，下面我们将对一些关键问题做简单阐述。

2.3.1 并行执行时间方程式

首先我们关心的是并行实现到底有多快。我们可以通过统计最优顺序算法的计算步数开始来评估在单计算机上的执行时间t_s。对于一个并行算法，除了要确定计算步数外，我们还需

⊖ 著名的不含错误代码的MPI例程是MPI_Wtime()返回一个双精度的运行时间。该例程被假设不会引起错误。

要估计通信开销。在消息传递系统中，在求解问题的整个执行时间中必须考虑发送消息的时间。并行执行时间t_p是由两部分组成的：一个计算部分t_{comp}和一个通信部分t_{comm}；即有

$$t_p = t_{comp} + t_{comm}$$

1. 计算时间

计算时间可使用类似于顺序算法的方法通过计数计算步数加以推算。当有多个进程同时执行时，我们只需计数最复杂进程的计算步数。通常所有进程都完成相同的操作，所以我们只需简单地计数一个进程的计算步数。在其他的情况中，我们将需要找出并发进程中的最大计算步数。一般而言，计算步数是n和p的函数。即有

$$t_{comp} = f(n, p)$$

t_p的时间单位与计算步的一样。为方便起见，我们常将计算时间用消息传递分成各个部分，然后确定每个部分的计算时间。于是有

$$t_{comp} = t_{comp1} + t_{comp2} + t_{comp3} + \cdots$$

其中t_{comp1}、t_{comp2}、$t_{comp3}\cdots$是各个部分的计算时间。

计算时间的分析通常假设所有处理器均相同且以相同速度运行。对特殊设计的多计算机/多处理机来讲情况确为如此，但对机群来讲就不一定如此。机群的一个重要特征是计算机不必全相同。在进行数学分析时，若考虑异构系统情况将有很大难度，因此我们的分析将假设所有计算机均是相同的。通过选择可用计算机间平衡计算负载的实现方法（负载平衡，load balancing）就可将不同类型的计算机考虑在内，如第7章中所叙述的那样。

2. 通信时间

通信时间与消息的数量、消息的大小、底层的互联结构以及传送方式有关。每个消息的通信时间与许多因素有关，包括网络结构和网络竞争。作为最初的近似，我们将使用下列公式表示消息1的通信时间：

62

$$t_{comm1} = t_{startup} + wt_{data};$$

其中$t_{startup}$为*启动*时间，有时也称为*消息时延*。实际上它是发送不包含数据的消息所需的时间（这一时间可以用简单方法测得）。它包括在源进程处将消息打包以及在目的进程处将消息解包所需的时间。如同在第1章中一样，术语*时延*（latency）用来描述完整的通信延时，所以这里用*启动*时间这一术语，且假设启动时间为一常数。t_{data}这一项表示发送一个数据字所需的传送时间，也假设它为一常数，w则表示数据字的数目。传送速率通常以位/秒（bits/second）为单位。当数据字中有b位时，就用b/t_{data}位/秒来表示。图2-13对此方程做了说明。当然在实际系统中，我们不可能得到如此完美的线性关系。许多因素会影响通信时间，包括对通信介质的争用。方程式忽略了在实际系统中存在这样的事实，即源和目的可能不直接链接以致于消息传递必须经过中间结点。此外，还假设由于在包中含有非数据的信息而导致的开销也是一个常数，且是$t_{startup}$的一部分。

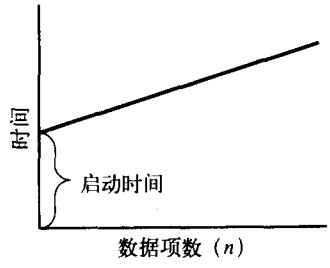


图2-13　理想的通信时间

最后的通信时间t_{comm}将是一个进程中所有顺序消息的通信时间的累加和，于是有：

$$t_{comm} = t_{comm1} + t_{comm2} + t_{comm3} + \cdots$$

其中，t_{comm1}，t_{comm2}，$t_{comm3}\cdots$是各消息的通信时间。（通常所有进程的通信模式是相同的，并假设是一起发生的，因而只需考虑一个进程。）

由于启动时间$t_{startup}$和数据传输时间t_{data}均以计算步单位衡量，这样我们就可将t_{comp}和t_{comm}加在一起来得到并行执行时间t_p。

3. 基准测试程序系数

一旦我们获得了顺序执行时间t_s、计算时间t_{comp}和通信时间t_{comm}，我们就可如在第1章中所叙述的那样为任何给定的算法/实现确定加速系数和计算/通信比，即:

$$\text{加速系数} = \frac{t_s}{t_p} = \frac{t_s}{t_{comp}+t_{comm}}$$

$$\text{计算/通信比} = \frac{t_{comp}}{t_{comm}}$$

63

两个系数均是处理器数p和数据元素数n的函数，并将为增加处理器数和增大问题规模的并行求解给出可扩展性的指示。特别是计算/通信比将突出通信在增加问题规模和系统规模时的影响。

4. 对方程式解释的重要评注

在后面几章的分析中将会有许多假设，而分析的唯一目的是给出一个起始点以指明在实际中一个算法是如何完成的。并行执行时间t_p将被标准化成以算术操作单位来衡量，当然这将依赖于计算机系统。为了便于分析，系统将被假设为是同构的。每个处理器均相同且以相同速度运行。同样，所有的算术操作将被认为需要相同时间，例如除法所需的时间与加法所需的时间相同。虽然在实际中很可能不是这样，但在分析时通常都做这样的假设。构成程序时所需的任何附加操作，如迭代的计数均不考虑。

我们将不对发送一个整数和发送一个实数或是其他格式加以区别。所有这些格式都被假设需要相同时间（这实际也是不真实的——在大多数实现中，传送一个8位的字符所需时间将小于发送一个64位的浮点数所需的时间。）然而在许多求解问题中，被发送数据的数据类型通常始终是相同的。实际的启动和传输时间也依赖于计算机系统，对不同系统它们可能相差很大。通常启动时间至少比传输时间高出1到2个数量级，而后者又远大于算术操作时间。实际上，启动时间在许多情况下将支配通信时间。我们不能忽略该项，除非n非常大（然而，在将方程式转换成阶记号时，可能将其忽略，参见2.3.2节）。

例 假设一台计算机的最大运行速率为1GFLOP（每秒10亿次浮点运算）并且启动时间为1μs。则在消息启动时间内，计算机可执行1 000次浮点操作。

5. 时延隐藏

在前面的例子中，每个消息需要花费相当于1 000次浮点运算的计算来进行消息的启动。这种影响常被共享存储器的支持者引证为消息传递多计算机的“阿喀琉斯的脚踵”（Achilles-heel）[⊖]。改善这种情况的一种方法是使通信与后续的计算重叠操作；也就是说，在等待通信结束时，同时使处理器忙于有用的工作，这就是所谓的*时延隐藏*。特别是非阻塞的发送例程提供了时延隐藏的可能性，但即使是（本地）阻塞发送例程也允许在等待目的进程接收消息时从事后续计算，且也许返回一个消息。习题2-8中将用该方法实验性地对时延隐藏进行研究。

时延隐藏也可以通过将多个进程映射到一个处理器上，并使用分时机制来加以实现，当第一个进程因未完成消息传递或由于其他原因而停顿时它就从一个进程转向另一个进程。有时称这些进程为*虚拟处理机*。在一个有p个处理器的计算机上实现m个进程（或虚拟机）算法被称为该计算机具有*m/p并行不完善性*（parallel slackness），这里的$p<m$。使用并行不完善性隐藏时延依赖于从一个进程切换到另一进程的有效的方法。*线程*能提供这种有效的机制。更详尽的阐述请参见第8章。

64

⊖ 在希腊神话传说中，英雄阿喀琉斯在出生后被其母倒提着在冥河中浸过，除未浸到的脚踵外，浑身刀枪不入。阿喀琉斯的脚踵喻指致命弱点。——译者注

2.3.2 时间复杂性

如同顺序计算一样，并行算法可用时间复杂性来加以评估（请注意记号O—“值的阶”、“大O”）[Knuth，1976]。在顺序程序设计中我们已熟悉了该记号，当某个变量（通常是数据大小）趋向无穷大时，可用它来把握一个算法的特征。在比较算法的执行时间时（时间复杂性）该记号特别有用，但它也可用于计算的其他方面，如对存储器的需求（空间复杂性）以及并行算法中的加速比和效率。让我们首先复习一下时间复杂性在顺序算法中的应用。

当选用表示执行时间的记号时，我们从估计计算步的数目开始，我们认为所有的算术和逻辑操作均相等，并忽略计算的其他方面，如计算测试。对计算步数表达式的推导通常与算法处理的数据项的数目有关。例如，假定一个算法A1需要对x个数据项完成个$4x^2+2x+12$个计算步。当增加数据项的数目时，总的操作数将越来越依赖于$4x^2$这一项。这个第一项将“支配”其他项，且最终使其他项变得无关紧要。在该例中，函数的增长是多项式的。对于另一个算法A2，求解同一问题将需要$5\log x+200$个计算步（在本书中，假设对数的底数均为2，除非另有说明）。对于较小的x，它将比第一个函数A1有更多的步数，但当x增大达到某一点时，第二个函数A2将比A1需要更少的计算步数，因而显得更为优越。在函数$5\log x+200$中，第一项（$5\log x$）最终将占支配地位，而另一项（200）可被忽略，因而在比较时，我们只需考虑占支配地位的那一项。函数$\log x$的增长是对数式的。对于足够大的x，对数增长将小于多项式增长。使用O记号（大O）可使我们把握增长模式。算法A1的大O为O（x^2），而算法A2的大O则为O（$\log x$）。

1. 形式化定义

O记号　可形式化定义如下：

$f(x)=O(g(x))$，当且仅当存在正常数c和x_0，如果对于所有的$x\geqslant x_0$，有$0\leqslant f(x)\leqslant cg(x)$。其中$f(x)$和$g(x)$是$x$的函数。例如，如果$f(x)=4x^2+2x+12$，则常数$c=6$将符合$f(x)=O(x^2)$的形
65 式化定义，因为当$x\geqslant 3$时，$0<4x^2+2x+12\leqslant 6x^2$。

不幸的是，该形式化定义也会导致$g(x)$的其他函数也可满足这一定义。例如，$g(x)=x^3$也满足定义，即当$x\geqslant 3$时，有$4x^2+2x+12\leqslant 2x^3$。一般地，我们将选用使$g(x)$具有最小增长的那个函数。事实上，在许多情况下，会有一个“严格的约束”，使得在一个常系数范围内，函数$f(x)$会与$g(x)$相等。这一关系可用记号Θ表述，

Θ记号　可形式化的定义如下：

$f(x)=\Theta(g(x))$，当且仅当存在常数c_1、c_2以及x_0，使得对于所有的$x\geqslant x_0$，有$0\leqslant c_1g(x)\leqslant f(x)\leqslant c_2g(x)$。

如果$f(x)=\Theta(g(x))$，则显然有$f(x)=O(g(x))$。对于函数$f(x)=4x^2+2x+12$而言，图2-14说明了如何满足这些条件的一种方法。实际上对于$g(x)=x^2$，我们可用许多方法来满足这些条件，但可以看到当$c_1=2$、$c_2=6$以及$x_0=3$时将会成功，即$2x^2\leqslant f(x)\leqslant 6x^2$。因此，我们可以说$f(x)=4x^2+2x+12=\Theta(x^2)$，它比使用O$(x^2)$更为精确。当且仅当能满足函数增长的上限时，我们才应使用大O记号。但是在实际使用过程中，在任何情况下总是使用大O。

Ω记号　函数增长的下限可用Ω记号描述，它的形式化定义如下：

66 $f(x)=\Omega(g(x))$，当且仅当存在正常数c和x_0使得对所有$x\geqslant x_0$有$0\leqslant cg(x)\leqslant f(x)$。

根据这一定义有$f(x)=4x^2+2x+12=\Omega(x^2)$（见图2-14）。我们将O ()读作“增长至多像……那样快”，而将Ω ( )读作“增长至少像……那样快”，函数$f(x)=\Theta(g(x))$为真，当且仅当$f(x)=\Omega(g(x))$和$f(x)=O(g(x))$。

Ω记号可用来指明最好个例情况。例如，一个排序算法的执行时间常依赖于欲被排序数的

原来的排序情况。为排序n个数，它可能至少需要$n\log n$步，但也可能需要n^2步，这全取决于原来这些数的排序情况。用时间复杂性来表示就可写成$\Omega(n\log n)$和$O(n^2)$。

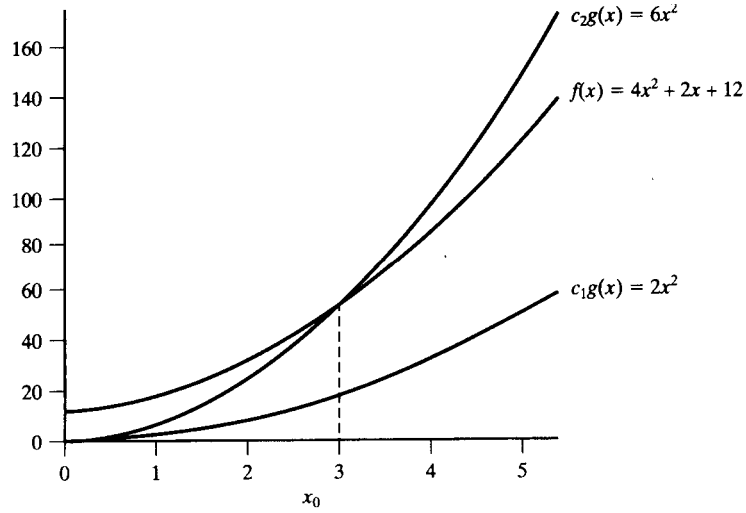


图2-14 函数$f(x) = 4x^2 + 2x + 12$的增长

2. 并行算法的时间复杂性

如果我们用时间复杂性分析，它将隐去那些较低项，这样t_{comm}的时间复杂性便为$O(n)$。t_p的时间复杂性将是计算复杂性和通信复杂性之和。

例 假定我们在两台计算机上加n个数，其中每台计算机各加$n/2$个数，开始时所有的数均存在第一台计算机中。第二台计算机将它的结果交给第一台计算机，再由第一台计算机将两个部分和加在一起。该问题的求解有以下几个阶段:

1）计算机1将$n/2$个数发送给计算机2。

2）两台计算机同时将$n/2$个数相加。

3）计算机2将它的部分结果送回给计算机1。

4）计算机1将两个部分和相加产生最后结果。

正如在大多数并行算法中那样，这里有计算和通信，通常将它们分别考虑:

计算（步骤2和步骤4):

$$t_{comp} = n/2 + 1$$

通信（步骤1和步骤3):

$$t_{comm} = (t_{startup} + n/2t_{data}) + (t_{startup} + t_{data}) = 2t_{startup} + (n/2+1)t_{data}$$

由计算公式可知: 其计算复杂性为$O(n)$，通信复杂性为$O(n)$。因而整个的时间复杂性为$O(n)$。

（1）*计算/通信比* 通常通信的代价是很高的。如果计算和通信有相同的时间复杂性，则增加n就不可能改善性能。理想的情况是，计算的时间复杂性应大于通信的时间复杂性，此时随着n的增加就可改进性能。例如，假定通信的时间复杂为$O(n)$，而计算的时间复杂性为$O(n^2)$，则当增加n时，最终可发现当n增大到一定值后，计算时间将支配整个的执行时间。有许多著名的例子都是这种情况。例如，在第1章中提及的，以及在第4章详尽讨论的N体问题，它的通信时间复杂性为$O(N)$，而计算时间复杂性为$O(N^2)$（使用直接的并行算法)。这是少数求解问题中的一个，这类问题的规模可能非常大。

（2）*代价和代价优化算法* 一个计算的处理器-时间乘积或代价（工作）可定义为;

$$\text{代价} = (\text{执行时间}) \times (\text{所使用的处理器总数})$$

一个顺序计算的代价仅是它的执行时间t_s。一个并行计算的代价是$t_p \times p$。一个代价优化并行算法是指在这种算法中求解问题的代价正比于在单处理机系统上的执行时间（用已知最快的顺序算法）；即

$$\text{代价} = t_p \times p = k \times t_s$$

其中k是一个常数。使用时间复杂性分析，我们说一个并行算法是代价优化算法，如果有

$$(\text{并行时间复杂性}) \times (\text{处理器数}) = \text{顺序时间复杂性}$$

例　假定求解一个有n个数的问题的已知最好顺序算法的时间复杂性为$O(n\log n)$。若求解同一问题的并行算法使用p个处理器并且它的时间复杂性为$O(\frac{n}{p}\log n)$，它便是代价优化的，而若一个并行算法使用p^2个处理器，它的时间复杂性为$O\left(\frac{n^2}{p}\right)$，它便不是代价优化的。

2.3.3　对渐近分析的评注

与时间复杂性广泛用作顺序程序分析和并行程序的理论分析不同，时间复杂性记号在评估并行算法的潜在性能方面却难有所作为。大O和其他复杂性记号使用渐近方法（即允许被考察的变量趋向无穷大），它可能不是完全相关的。由分析得出的结论是基于所考察的变量的，通常它或是数据大小或是处理器数，且它们的增长将趋向于无穷大。但是，通常处理器数是有限的，因此我们不可能将处理器数扩展成无穷大。类似地，我们感兴趣的是有限的和可管理的数据大小。此外，分析所忽略的次要项也许是非常重要的。例如，通信时间方程：

$$t_{\text{comm}} = t_{\text{startup}} + wt_{\text{data}}$$

的时间复杂性为$O(w)$，但对于合理大小的w值，启动时间将完全支配整个通信时间。最后，该分析也忽略了在实际计算机中出现的其他因素，例如通信竞争。

共享存储器程序

到目前为止，大多数的讨论都集中在消息传递程序上。对于共享存储器程序，由于通信部分不再存在，因此时间复杂性就简化成仅是计算的，尤如顺序程序那样。在那种情况下，时间复杂性就可能更为贴切。但是，共享存储器程序要考虑另一个附加因素，即对共享数据
68 的访问必须以受控方式进行，从而导致了附加的延迟。在第8章中将阐述这方面的内容。

2.3.4　广播/集中的通信时间

尽管我们对理论分析作了评注，让我们考察某些广播/集中操作以及它们的复杂性。几乎所有的问题都需要将数据广播给各个进程并从各个进程处集中数据。大多数的软件环境都支持广播和集中，而所采用的具体算法将依赖于多计算机基本的体系结构。在过去，程序员可能具有某些互联体系结构的知识（如网格等）并能对它有效利用，但今天这种互联体系结构通常对程序员是隐蔽的。

在本书中，我们讨论的重点是使用机群。同样，确切的互联结构对用户将是隐蔽的，尽管很可能使用交换器在计算机间提供完全的同时连接。在久远的过去，以太网用单线连接所有计算机。在单以太网链路上进行广播时，可以采用让单个消息由网络上的所有目的结点同时读取来完成（以太网协议提供这种通信方式）。因此，广播方式是非常高效的且只需一个

消息：

$$t_{comm} = t_{startup} + wt_{data}$$

对于一个数据项来讲其时间复杂性为$O(1)$；对w项数据来讲，其时间复杂则为$O(w)$。

当然大多数机群化计算机会使用各种网络结构，但这些网络结构不会提供如单以太网中那种便利的广播介质。通常消息从最初的计算机发送到多个目的，而这些目的每一个又将同一消息依次发送给多个目的。一旦消息到达目的进程，它就被转换成1到N个扇出的广播调用，从那里将同一消息依次发送给每一个目的，如图2-15所示。对集中来讲将需要同样的构造，只是消息以相反方向传送。不管是哪一种情况，限制的因素是消息的发送或接收是否是顺序的，这就导致单源到N个目的的通信时间复杂性为$O(N)$：

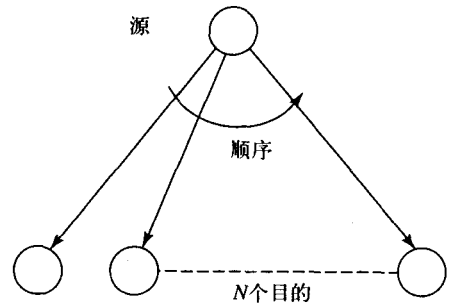


图2-15 1对N的扇出广播

$$t_{comm} = N(t_{startup} + wt_{data})$$

我们假设每一层的消息将同时出现（当然实际上不可能这样）。

图2-16中示出了应用在树结构中的1对N的扇出广播。这里的复杂性将依赖于每一层的结点数和层数。对一个二叉树，有$N=2$，且如果有p个最终目的结点它将有$\log p$级，则就有： 69

$$t_{comm} = 2(\log p)(t_{startup} + wt_{data})$$

这里也假定消息在每一层出现是同时的。（作为一个习题请读者推导不作此假定时的通信时间。）

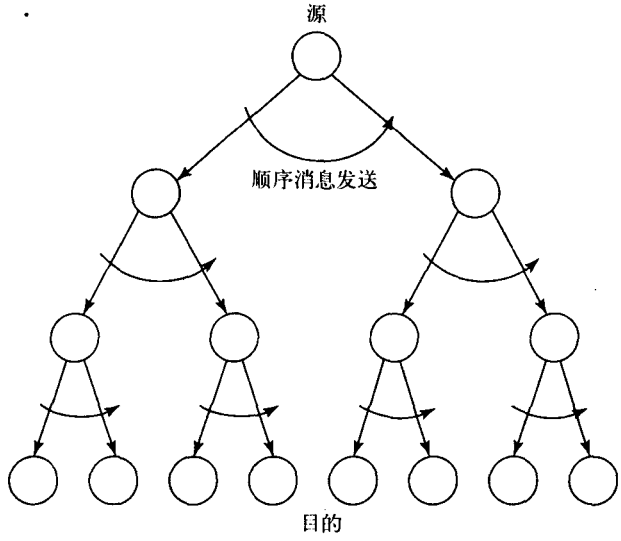


图2-16 树结构中的1对N的扇出广播

广播的二叉树实现的一个缺点是，如果在树中有一个结点失效，那么在它下面的所有结点都将收不到消息。此外，在库调用中实现二叉树是比较困难的。当然程序员也可在进程中以显式编码来实现二叉树。

2.4 用经验方法进行并行程序的调试和评估

在编写一个并行程序时，我们首先要使它能正确地执行。然后我们才会对程序能执行多

快感兴趣。最后，我们将看是否能做到执行得更快。

2.4.1 低层调试

使一个并行程序能正确地工作是巨大的智力挑战。先编写一个基于最终并行算法的顺序版本是非常有用的。已经说过，必须使并行代码能正确的工作。顺序（串行）程序中的错误可通过调试发现。通常采用的方法是对该代码进行测试，即在程序中插入一些在程序执行时能够输出中间计算值的代码，使用打印语句输出这些中间值。在并行程序中也可使用类似技术，但是使用这种方法会造成某些非常严重的后果。第一个也是最重要的一个后果是，测试
70 一个顺序代码只是简单地使其执行变慢，但它仍能确定性地有效工作并且会产生相同答案。但在不同进程中通过插入代码来调试并行程序时，则不但会使计算的速度明显变慢，而且还会使指令以不同的交叉顺序执行。这是因为，一般来说每个进程将会受到不同影响的缘故。此外还会出现这样的情况，即在插入测试代码后，有可能使原本不正常工作的程序转而开始工作——这就确定无疑地表明问题出在进程间的定时。

应提醒的是，由于进程可能在远程计算机上执行，因此打印语句的输出可能需要重定向到一个文件中，以使其能在本地计算机中看到。消息传递软件通常具有重定向输出的工具。

最低层的调试（在极端情况下）需使用调试器。初等的顺序程序调试工具，如dbx，可用来检验寄存器（虽然很少被使用），以及设置“断点”以停止执行。但在并行程序的调试中，使用这些技术几乎是毫无价值的，这是因为无法确切知道在不同进程中的交叉顺序等因素的缘故。一种解决方案是在各处理器上运行该调试器，并在各自的显示窗口监视输出。由于一个并行计算可能会有许多同时执行的进程，从而会使此方法无实用价值。在动态创建进程的情况下，可能需要系统设施通过调试器来启动派生进程。

并行计算具有无法由常规的顺序调试器捕捉到的一些特征，如事件的定时。事件仅在某种条件出现时方可被识别。除了在顺序程序中可能出现的如对一个存储单元的访问之外，在这种场合下，一个“事件”可能就是消息的发送或接收。已经开发了一些并行调试器[McDowell and Helmbold，1989]。

2.4.2 可视化工具

并行计算有助于其活动的可视化指示，而消息传递软件常把提供可视化工具作为整个并行程序设计环境的一部分。程序可以在时空图（或进程-时间图）中执行时被观察。图2-17中示出了一个假设的例子。每个等待间隔表明进程处于闲置状态，通常是等待接收一个消息。这种可视化的描述有助于准确定位有错的动作。可将构成时空图的事件加以捕获和保存，以便可再现描述而无需将程序重新执行。此外感兴趣的将是时间利用图（utilization-time diagram），它将显示出每个进程耗费在通信、等待以及消息传递库例程上的时间。除了有助于调试外，时间利用图还可表明计算的效率。最后，当进程在二维空间中表示以及状态的变化以电影形式表示时，动画可能很有用。

已为MPI提供了可视化工具的实现。Uphost程序可视化系统[Herrarte and Luske，1991]就是一个例子。可视化的所有形式蕴含着将软件“探针”插到执行中，它可能会改变计算的特征。可以肯定的是，它会使计算的进展大为放慢（也可使用硬件性能监控器，它通常不会影
71 响性能。例如对系统总线的简单监控，但是这些方法并未被广泛使用）。

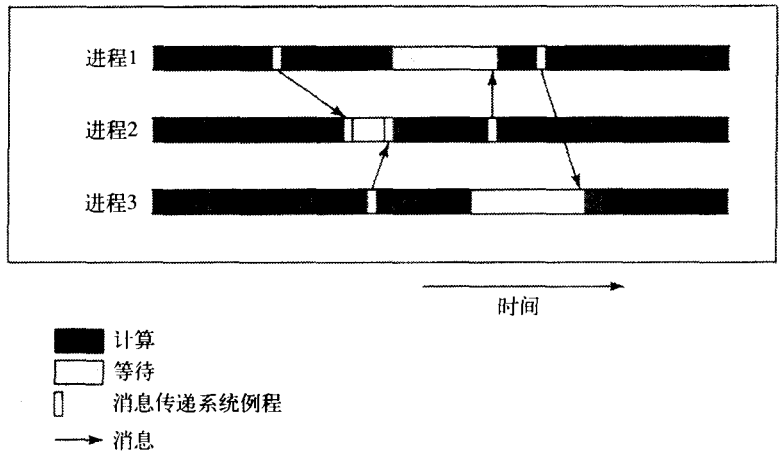


图2-17 并行程序的时空图

2.4.3 调试策略

[Geist et al., 1994a]建议用三步法来调试消息传递程序:

1）如果可能的话，将程序作为单进程运行，并作为通常的顺序程序进行调试。

2）在单计算机上用2到4个多任务进程来执行程序，同时对一些动作进行考察，如检查消息是否被送到了正确的位置。因为经常会用错消息标记，从而使消息传送到错误位置。

3）在若干台计算机上用同样的2到4个进程来执行程序。这一步将有助于发现因网络延时而导致的与同步和定时有关的问题。

作为良好的程序设计习惯，在程序中放置错误检测代码总是非常重要的，而在并行程序中特别重要的是要保证能处理导致出错的条件以及防止死锁的发生。当检测出一个错误后，许多消息传递例程将返回一个出错代码。虽然这些例程不一定会被执行，但一旦它们出现时，就应能识别这些出错代码。它们在调试时也很有用。也可使MPI返回出错代码，虽然其默认的情况是，当遇到错误时就使程序异常终止。

2.4.4 评估程序

1. 执行时间的测量

时间复杂性分析可洞察并行算法的潜力，且在比较不同算法时也非常有用。但是，如我们在2.3.3节中对时间复杂性分析所作的评注那样，仅当算法被编码且在多处理机系统上执行后，才能确实知道实际执行的算法倒底有多好。如同低层调试一样（参见2.4节），程序可用附加的代码进行调试。为测定一个程序的执行时间，即代码中两点之间以秒为单位的运行时间（elapsed time），我们可用常规的系统调用如`clock()`、`time()`、或`gettimeofday()`。因此，为了测量代码中点L1和点L2之间的执行时间，我们可用如下的代码结构:

72

```
        .
        .
        .
L1:  time(&t1);                          /* start timer */
        .
        .
        .
L2:  time(&t2);                          /* stop timer */
        .
        .
        .
```

```
elapsed_time = difftime(t2, t1);      /* elapsed_time = t2 - t1 */
printf("Elapsed time = %5.2f seconds", elapsed_time);
```

运行时间将包括等待消息的时间，并且假设在此时处理机不执行任何其他程序。

通常消息传递软件本身含有定时工具；例如，提供返回时间的库调用或是在时空图中显示时间（如在2.4节中所叙述的那样）。MPI提供例程MPI_Wtime()来返回时间（以秒为单位）。通常，每个处理器将使用自己所拥有的时钟，所以返回的时间将不一定会与其他处理器的时钟同步，除非可用时钟同步。在MPI中，时钟同步被定义为环境属性，但在一个具体的系统中它可能不被实现，因为通常它将导致很严重的系统开销。

2. 用乒乓法测量通信时间

一个指定系统的点对点通信的时间可用如下的乒乓方法加以测定。由进程P_0向另一进程P_1发送消息，而P_1收到消息后立即将该消息发回给P_0。而消息通信所需时间则在P_0加以记录。将该时间用2除后就可测定单程通信所需时间：

进程P_0

```
        .
        .
        .
L1:  time(&t1);
     send(&x, P1);
     recv(&x, P1);
L2:  time(&t2);
     elapsed_time = 0.5 * difftime(t2, t1);
     printf("Elapsed time = %5.2f seconds", elapsed_time);
        .
        .
        .
```

73

进程P_1

```
        .
        .
        .
  recv(&x, P0);
  send(&x, P0);
        .
        .
        .
```

习题2-5探讨了如何测量通信时间。

3. 特性形象

程序特性形象是一个用来指示程序中的不同部分在执行时所花时间的直方图。特性形象能图示特定源程序语句的执行次数，如图2-18所示。实际生成结果的特性形象器由于必须在程序执行过程中捕捉信息，因此会影响执行时间。对每条指令的出现次数进行统计显得过度，故常采用对执行代码进行间隔性的检测或采样以给出统计结果。任何形式的探测都将影响执行特征。在并发进程之间存在相互联系的并行程序中这一点显得尤为重要。

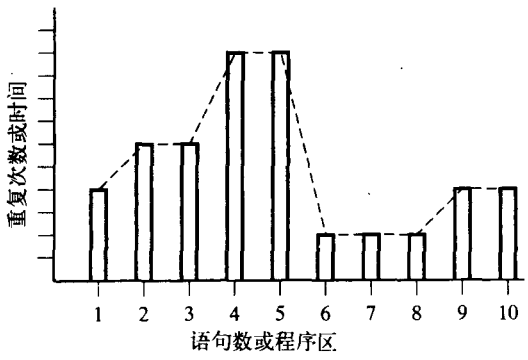


图2-18　程序的特征形象

特征形象可用来标识程序中的“热点”，

在程序执行过程中这些“热点”将被多次访问。应该首先对这些地方加以优化，无论是顺序还是并行程序都可以使用这一方法。

2.4.5 对优化并行代码的评注

一旦对性能进行测试后，为了改善性能很可能需要对程序进行结构上的改变。除了那些适用于常规单处理器程序的优化方法（如将常数计算移到循环之外）以外，还可使用一些并行优化的方法。这些方法通常与多处理机系统的体系结构有关。可以改变进程数从而使进程的颗粒度得以改变。还可设法增加消息中的数据量以减小启动时间的影响。此外可在本地对一些所需的值重新计算，通常总是比将计算所得的值作为附加消息从一个进程发送到需要这 74
些值的其他进程要好得多。对通信和计算还可进行重叠操作（时延隐藏，参见2.3.1节）。还可以对程序进行*关键路径分析*（critical path analysis）；也就是说要确定程序中的各种并发部分，并找出对整个执行时间起支配作用的最长路径。

与以上所提及的因素相比，存储器层次结构的影响是一个不很明显的因素。处理器一般使用高速缓冲存储器，处理器总是先访问它的高速缓存，仅当所需信息不在高速缓存中时它访问主存储器。此后就将从主存储器得到的信息装载到高速缓存中，但在高速缓存中能保存的数据量是有一定限制的。显然在需要的时候数据尽可能多地存放在高速缓存中就会导致最好的性能。为达到这一目的，有时可用特定的并行化策略，而有时仅需在程序中简单地重排对存储器请求的顺序。在第11章中将叙述某些数值算法，简单地以不同的顺序来完成算术操作序列，就可使后续引用的数据大多数在高速缓存中得到。

2.5 小结

本章介绍了以下概念:

- 基本的消息传递技术
- 发送、接收和集合操作
- 控制工作站网络的软件工具
- 通信的建模
- 通信时延和时延隐藏
- 并行算法的时间复杂性
- 并行程序的调试和评估

推荐读物

总的来讲，我们主要关注消息传递例程，特别是MPI系统。另一个广泛使用的系统是PVM（并行虚拟机，Parallel Virtual Machine）。有关PVM的最早期的一篇论文是[Sunderam, 1990]。有关PVM的更多详尽资料可参见[Geist et al., 1994a和1994b]。有关MPI的进一步资料可在[Gropp, Lusk and Skjellum, 1999]和[Snir et al., 1998]中找到。有关MPI-2的内容可在[Gropp et al., 1998]和[Gropp, Lusk and Thakur, 1999]的文献中见到。MPI论坛的主页 http://www.mpi.-forum.org 提供了有关MPI的许多官方文档和信息。其他的参考文献来源包括[Dongarra et al., 1996]，该论文除了PVM之外，给出了较早期的几个消息传递系统的有关参考文献。在[Sterling, 2002a and b]的许多章中描述了PVM和MPI的程序设计。有关消息分散和集中的论文包括[Bhatt et al., 1993]。

在经典教科书[Knuth, 1973]中可找到有关顺序算法的基本设计和分析。其他的教科书包 75

括[Aho, Hopcroft, and Ullman, 1974]，此后又有许多其他的教科书问世。[Cormen, Leiserson and Rivest, 1990]是一本论及该主题的现代综合性的教科书之佳作，该书篇幅超过了1000页。有关并行算法设计和分析可从几本教科书中找到，其中包括[Akl, 1989]和[JáJá, 1992]。[Berman and Paul, 1997]是一本较有特色的书籍，它将顺序和并行算法集成为一体。

在[McDowell and Helmbold, 1989]以及[Simmons et al., 1996]中可看到有关并行调试的细节。其他的论文包括[Kraemer and Stasko, 1993]以及[Sistare et al., 1994]。《*IEEE Computer*》为并行和分布式处理工具出了一期特刊（1995年11月），其中包括了许多篇有关并行系统性能评估工具的很有价值的文章。

参考文献

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, MA.

AKL, S. G. (1989), *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, NJ.

BERMAN, K. A., AND J. L. PAUL (1997), *Fundamentals of Sequential and Parallel Algorithms*, PWS Publishing Company, Boston, MA.

BHATT, S. N., ET AL. (1993), "Scattering and Gathering Messages in Networks of Processors, *IEEE Trans. Comput.*, Vol. 42, No. 8, pp. 938–949.

CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST (1990), *Introduction to Algorithms*, MIT Press, Cambridge, MA.

DONGARRA, J., S. W. OTTO, M. SNIR, AND D. WALKER (1996), "A Message-Passing Standard for MMP and Workstations," *Comm. ACM*, Vol. 39, No. 7, pp. 84–90.

FAHRINGER, T. (1995), "Estimating and Optimizing Performance for Parallel Programs," *Computer*, Vol. 28, No. 11, pp. 47–56.

FOSTER, I. (1995), *Designing and Building Parallel Programs*, Addison-Wesley, Reading, MA.

GEIST, A., A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK, AND V. SUNDERAM (1994a), *PVM3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, TN.

GEIST, A., A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK, AND V. SUNDERAM (1994b), *PVM: Parallel Virtual Machine*, MIT Press, Cambridge, MA.

GRAMA, A., A. GUPTA, G. KARYPIS, AND V. KUMAR (2003), *Introduction to Parallel Computing*, 2nd edition, Benjamin/Cummings, Redwood City, CA.

GROPP, W., S. HUSS-LEDERMAN, A. LUMSDAINE, E. LUSK, B. NITZBERG, W. SAPHIR, AND M. SNIR (1998), *MPI, The Complete Reference*, Volume 2, *The MPI-2 Extensions*, MIT Press, Cambridge, MA.

GROPP, W., E. LUSK, AND A. SKJELLUM (1999a), *Using MPI Portable Parallel Programming with the Message-Passing Interface*, 2nd edition, MIT Press, Cambridge, MA.

GROPP, W., E. LUSK, AND RAJEEV THAKUR (1999b), *Using MPI-2 Advanced Features of the Message-Passing Interface*, MIT Press, Cambridge, MA.

HERRARTE, V., AND LUSKE, E. (1991), "Studying Parallel Program Behavior with Upshot," Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory.

INMOS LTD. (1984), *Occam Programming Manual*, Prentice Hall, Englewood Cliffs, NJ.

JÁJÁ, J. (1992), *An Introduction to Parallel Algorithms*, Addison Wesley, Reading, MA.

KARONIS, N. T. (1992), "Timing Parallel Programs That Use Message-Passing," *J. Parallel & Distributed Computing*, Vol. 14, pp. 29–36.

KNUTH, D. E. (1973), *The Art of Computer Programming*, Addison-Wesley, Reading, MA.

KNUTH, D. E. (1976), "Big Omicron, Big Omega and Big Theta," *SIGACT News* (April–June),

pp. 18–24.

KRAEMER, E., AND J. T. STASKO (1993), "The Visualization of Parallel Systems: An Overview," *J. Parallel & Distribut. Computing*, Vol. 18, No. 2 (June), pp. 105–117.

KRONSJO, L. (1985), *Computational Complexity of Sequential and Parallel Algorithms*, Wiley, NY.

MCDOWELL, C. E., AND D. P. HELMBOLD (1989), "Debugging Concurrent Programs," *Computing Surveys*, Vol. 21, No. 4, pp. 593–622.

SIMMONS, M., A. HAYES, J. BROWN, AND D. REED (EDS.) (1996), *Debugging and Performance Tools for Parallel Computing Systems*, IEEE CS Press, Los Alamitos, CA.

SISTARE, S., D. ALLEN, R. BOWKER, K. JOURDENAIS, J. SIMONS, AND R. TITLE (1994), "A Scalable Debugger for Massively Parallel Message-Passing Programs," *IEEE Parallel & Distributed Technology*, (Summer), pp. 50–56.

SNIR, M., S. W. OTTO, S. HUSS-LEDERMAN, D. W. WALKER, AND J. DONGARRA (1998), *MPI - The Complete Reference Volume 1, The MPI Core, 2nd edition*, MIT Press, Cambridge, MA.

STERLING, T., editor (2002a), *Beowulf Cluster Computing with Windows*, MIT Press, Cambridge, MA.

STERLING, T., editor (2002b), *Beowulf Cluster Computing with Linux*, MIT Press, Cambridge, MA.

SUNDERAM, V. (1990), "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice & Experience*, Vol. 2, No. 4, pp. 315–339.

习题

2-1 试为消息通信时间t_{comm}书写一个方程式，它应能反映出如在静态互联网络中那样的经过多级链路所产生的延迟。为网格网络书写方程式，假定所有的消息目的地是随机选择的。

2-2 利用值作为传递参数可避免在书中及MPI中的发送和接受中所使用的指针。例如，在接收例程中使例程返回消息数据，然后将其赋值到一个变量，即`x = recv (sourceID)`，就可消除指针。"环绕"正规的MPI发送和接收例程编写不使用指针的新例程并演示它们的使用。

2-3 从一个指定的源进程向一个指定的目的进程发送一个消息时，源进程必须知道目的的TID（任务识别符）或序号，而目的需知道源的TID或序号。举一个程序例子解释在MPI中每个进程如何能获得其他进程的TID或序号。

2-4 （适合作为第1个家庭作业）编译和运行实现将数相加的MPI程序，如图2-14和图2-16所示的那样（或作为编译指令中的"实例程序"，在网址http: //www.cs.uncc.edu/par_prog上找到），并在你的系统上加以执行。修改此程序以实现查找最大数并与累加和一起输出。

76 ~ 77

2-5 在并行程序设计系统中用如下的代码段来测量发送消息的时间。

主进程

```
        .
        .
        .
L1:   time(&t1);
      send(&x, P_slave);
L2:   time(&t2);
      tmaster = difftime(t2, t1);
      recv(&tslave, P_slave);
      printf("Master Time = %d", tmaster);
      printf("Slave Time = %d", tslave);
        .
        .
        .
```

从进程

```
        .
        .
        .
L1:   time(&t1);
      recv(&x, P_master);
L2:   time(&t2);
      tslave = difftime(t2, t1);
      send(&tslave, P_master);
        .
        .
        .
```

这是对2.4.4节所述的乒乓方法的重复。以发送多个消息的组和发送大小不同的消息进行实验，来获得消息传送所需时间的精确评估，在发送一个消息所需时间相对于消息大小的坐标空间中，将实验所得结果画成一条直线。估算它的启动时间$t_{startup}$（时延），以及发送一个数据项所需时间t_{data}。

2-6　对广播以及在你的系统中所具有的其他集合例程，重做习题2-5。

2-7　使用单个的发送和接收例程经验性地比较广播和集中例程的用法。

2-8　在你的系统上对时延隐藏进行实验，以确定在发送消息间隔中能进行多少计算。要求使用非阻塞和本地阻塞发送例程两者进行推算。

2-9　试为在2.3.4节中所叙述的二叉树广播的通信时间和时间复杂性书写一个方程式，这里假定在每一级上的消息并不同时出现（如同实际中发生的那样）。再将其扩展为一个m叉树的广播（每个结点有m个目的）。

2-10　如果PVM和MPI两者都可供你使用（或任意两个系统），试用传递消息方法，对两个进程在该系统上进行通信所需时间作比较性的研究，在这些进程中已提供测量通信时间的手段。

第3章 易并行计算

在本章中，我们将从并行计算观点来考虑“理想”的计算——一种能分解成许多完全独立部分且每一部分可用一个独立的处理器执行的计算。这就是所谓的*易并行计算*(embarrassingly parallel computation)。在讨论不能将计算简单地分解的其他各章之前，我们将先着手讨论易并行计算的实例。本章的内容可作为编写第一个并行程序的基础。

3.1 理想的并行计算

并行程序设计包括将一个问题分解成若干部分，然后由各个处理器对各个部分分别进行计算。一个理想的并行计算是能被立即分解成许多完全独立部分且它们能同时执行的计算。形象化地称其为*易并行*（该术语由Geoffrey Fox提出，参见[Wilson，1995]），也许更贴切地应称其为*自然并行*（naturally parallel）。对这种问题进行并行化应是一目了然的，且无需特殊技术或算法就可得到一个成功解。一个理想的易并行计算在各个进程间没有通信，即它是一个完全独立的计算图，如图3-1所示。每个进程需要不同（或相同）数据，并由其输入数据产生最终的结果，而不需要使用其他进程生成的结果。如果在整个计算期间，所有可用处理器均被分配有进程，此时就可获得最大的加速比。这里所需的唯一结构，是对数据进行简单地分配并启动各进程。有趣的是有大量著名的实际应用是属易并行的，或至少是接近于易并行的。如图3-1所示，通常由于各个独立部分为相同计算，因此称其为SPMD(单程序多数据) 模型更为合适。由于数据不是共享的，因此使用分布式存储器多处理机或消息传递多计算机就更为合适。如果它们需要同一数据，则该数据必须拷贝到每个进程中去。重要的特点是进程间没有交互。

在一个实际的易并行计算中需将数据分布到各个进程，并用某种方式收集和组合结果。这就意味着在开始和最后，只有一个进程处于运行状态。如果要创建动态进程，通常的方法是采用主从结构。首先创建一个主进程，由它派生（启动）相同的从进程。图3-2示出了这种最终的结构（主进程在派生从进程后可继续自己的计算，虽然在结果一到达就需要主进程时通常不这样安排）。如2.2.3节中所提及的那样，主从方法可以用作静态的进程创建。其中，我们只需简单地将主进程和从进程都放在同一程序中，而使用IF语句根据进程标识符ID（主ID或从ID）来选择主代码或从代码。在后面给出的伪代码序列中，有关主、从进程启动的实际细节已省略。

在本章中我们将考虑从进程之间有交互最少的应用。即使从进程全都一样，静态地将进

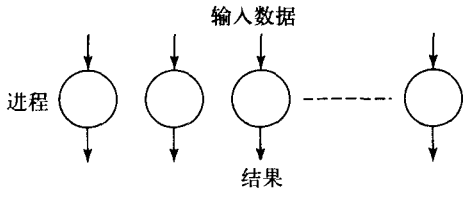


图3-1 独立的计算图（易并行问题）

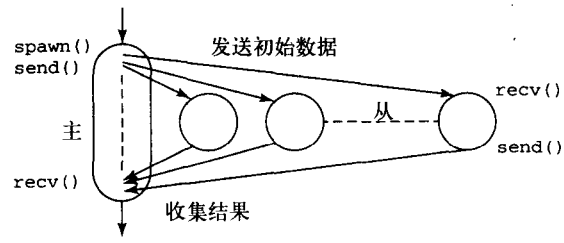


图3-2 使用动态进程创建和主从方法的实际易并行计算图

程分配给处理器仍可能得不到优化解，处理器不相同时就更为如此，而在联网工作站中这种情况是很平常的，使用负载平衡技术就可提供改进的执行速度。在这一章内我们将介绍负载平衡，但仅限于在从进程间无交互的情况。当进程之间有交互时，负载平衡就需要使用明显
80 不同的方法，这将在第7章中论及。

3.2 易并行计算举例

3.2.1 图像的几何转换

常将图像存于计算机中以便可用某种方法对图像加以改变。显示的图像常源于两种方法，一种是从如摄像机那样的外部源得到，这种图像可能需要以某种方法（*图像处理*）对它加以改变；另一种方法是采用人工创建，该方法通常与术语*计算机图形学*有关。不论是哪一种方法，都可在已存储的图像上进行许多图形操作。例如，我们可能想要将图像移动到显示空间的另一个位置、对图像大小进行缩小或放大或是在二维或三维空间中对它进行旋转。必须高速地完成这些图形转换以使观察者可以接受。通常还需要进行其他的图像处理操作，如平滑和边缘检测，特别对于源自外部的图像，因为它很可能是“有噪声的”。这些操作通常是易并行的。在第11章中将讨论这些图像处理的操作。这里我们将只考虑简单的图形转换。

存储一个二维图像的最基本的方法是使用*像素图*（pixmap），其中的每个*像素*（pixel或picture element）是以二进制数的形式存放在一个二维数组中。对于纯黑白图像，每个像素用一位二进制位就够了，像素为白色就用1表示，为黑色就用0表示，这就是所谓的*位图*（bitmap）。*灰度*图像就需使用多位来表示，通常用8位来表示256种不同的单色亮度。至于彩色就需要用更多位来加以说明。通常在显示器中使用的三种基本颜色是红、绿和蓝（RGB），每种颜色各以8位数分别加以存储。这样每个像素需用三个字节表示，每个字节分别表示红、绿和蓝色，总共为24位。使用这种表示的标准图像文件格式称为“tiff”格式。

可以使用查找表来减少彩色图像对存储器的需求，在该查找表中保存的仅是图像中要用到的指定颜色的RGB表示。例如，假定只出现256种不同颜色，那么具有256项且每项长为24位的一张表就可保存所要用到的颜色的表示。那么图像中的每个像素只需用8位地址码就可从查找表中选出所指定的颜色。该方法可用于外部图像文件中，此时该查找表与图像一起存放在文件中，例如“gif”文件格式。该方法也可用于内部生成的图像以减小视频存储器的容量（可降到仅为原来的1/3）。尽管随着视频存储器越来越便宜，该法已不如以前那样吸引人。在这一节中，我们假设所讨论的图像是一种简单的灰度图像（彩色图像可归约成灰度图像，图像处理常这样做，参见第11章）。经常不严格地使用术语*位图*（bitmap）和*位映射*（bit-mapped）表示用二进制形式存储的图像，尤如一个像素阵列。

当要将像素位置移动而又不影响其值时，就需对每个像素的坐标进行数学运算以实现几何变换。由于对每个像素的变换是完全独立于其他像素变换的，因而这确是一个易并行计算。变换的结果很简单地是一张已更新的位图。下面给出一些常见的几何变换的示例[Wilkinson and Horrocks，1987]:

①*平移*

81 在x方向移动Δx、在y方向移动Δy后的一个二维对象的坐标由下式给出:

$$x' = x + \Delta x$$
$$y' = y + \Delta y$$

式中x和y为原坐标，x'和y'为新坐标。

②缩放

在x方向以系数S_x缩放，而在y方向以S_y缩放的对象的坐标由下式给出:

$$x' = xS_x$$
$$y' = yS_y$$

当S_x和S_y大于1时，对象被放大；而当S_x和S_y在0和1之间时，对象就被缩小。注意，在x和y方向上的放大和缩小比例不必相同。

③旋转

绕坐标系原点旋转θ角后的对象的坐标由下式给出:

$$x' = x\cos\theta + y\sin\theta$$
$$y' = -x\sin\theta + y\cos\theta$$

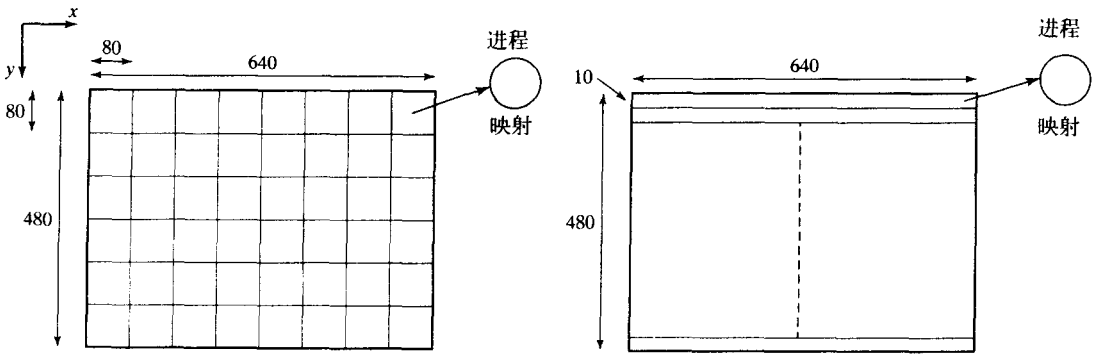
④裁剪

该变换将已定义的矩形边界应用到一个图形上，并从显示图中将那些落在定义区域之外的所有点删除掉。在经旋转、移位和缩放后，用裁剪来消除落在显示范围之外的坐标是很有用的。如果被显示区域的最低的x和y的值为x_l、y_l，而最高的x和y的值为、x_h，y_h，那么

$$x_l \leqslant x' \leqslant x_h$$
$$y_l \leqslant y' \leqslant y_h$$

对于要显示的点（x'，y'）应该为真；否则点（x'，y'）将不被显示。

输入数据是位图，通常存于文件中并被复制到数组中。该数组的内容可很容易地进行处理而无需任何特殊的编程技术。并行程序设计主要关心的是将位图分成一些像素组以供每个处理器加工，这是因为一般来讲像素数会远大于进程/处理器数。有两种一般的编组方法：以正方形/矩形区域编组和以列/行编组。我们可简单地将一个进程（或处理器）分配到一个显示区域。例如，对一个640 × 480图像和48个进程，我们可将显示区划成48个80 × 80矩形区，并为每个80 × 80矩形区分配一个进程。或者我们可将显示区划分成48行640 × 10像素分配给每个进程。将一个区域分成按行或按列的矩形/正方形区域的概念（如图3-3所示），在含有处理二维信息的许多应用中都有所体现。在第6章中我们将讨论如何在将一个区域划分成正方块或行（列）之间进行折衷。对于像这里所讨论的那样相邻区域间不存在通信的情况，除了可能易于编程外，采用哪种分割方法是无关紧要的。 82



a）每个进程的正方形区域　　b）每个进程的行区域

图3-3 为各个进程划分区域

假设我们使用一个主进程和48个从进程，并以10行一组进行分割。则每个从进程将处理

一个640 × 10区域，并向主进程返回新的坐标值以供显示。如果所进行的变换是平移；如前面（a）中已描述的那样，则主从方法可能以主进程向每个从进程发送需由每个进程处理的前10行中的第1行的行号开始。一旦接收到它的行号，每个进程逐步通过其行组中的每一个像素坐标转换其坐标，并将旧坐标和新坐标送回给主进程。为简单起见，可用各个消息而不是一个消息进行这种传送。然后由主进程更新位图。

假定原先的位图保存在数组map[][]，并声明一个临时位图temp_map[][]。通常显示坐标系统的原点在左上角，如图3-3所示。将原点在左下角的图像转换到显示坐标系统是很简
83 单的。为此我们将省略这些细节。应注意的是，在C语言中，元素是以第一个下标为行、第二个下标为列的形式逐行存放的。完成图像平移的伪代码如下：

主进程：

```
for (i = 0, row = 0; i < 48; i++, row = row + 10)   /* for each process*/
   send(row, Pi);                                    /* send row no.*/

for (i = 0; i < 480; i++)                            /* initialize temp */
   for (j = 0; j < 640; j++)
      temp_map[i][j] = 0;

for (i = 0; i < (640 * 480); i++) {                  /* for each pixel */
   recv(oldrow,oldcol,newrow,newcol, PANY);          /* accept new coords */
   if !((newrow < 0)||(newrow >= 480)||(newcol < 0)||(newcol >= 640))
      temp_map[newrow][newcol]=map[oldrow][oldcol];
}
for (i = 0; i < 480; i++)                            /* update bitmap */
   for (j = 0; j < 640; j++)
      map[i][j] = temp_map[i][j];
```

从进程：

```
recv(row, Pmaster);                                  /* receive row no. */
for (oldrow = row; oldrow < (row + 10); oldrow++)
   for (oldcol = 0; oldcol < 640; oldcol++) {        /* transform coords */
      newrow = oldrow + delta_x;                     /* shift in x direction */
      newcol = oldcol + delta_y;                     /* shift in y direction */
      send(oldrow,oldcol,newrow,newcol, Pmaster);    /* coords to master */
   }
```

在主进程接收部分，使用了一个通配符P_{any}说明可以以任何次序接收来自任何从进程的数据。被平移的图像可能超过显示区域的边界，为此需将不出现在显示区中的新图像的任一部分的位图置为0（黑色）。

在该例中，由主进程将各起始行号发送给各个从进程。但由于起始号与进程ID有关，因而每个从进程可自己确定起始行。此外，由于每次只返回一个结果而不是一组结果，从而就无法像成组返回结果那样可减少传送消息的开销时间。程序中没有代码指明从进程的终止，因此只有当所有结果都生成后才会终止，否则主进程将永远等待下法。程序中的从进程在完成自己的固定任务数之后可自行终止；在其他情况下，也可由主进程向所有从进程发送一个消息来做到这一点。上述的代码并非是完善的，为了便于讨论，显示区域的大小、每个进程处理的行数以及进程数均被固定地编码。要使上述代码具有通用性，这些系数应该易
84 于改变。

分析

假设每个像素需要两个计算步且总共有$n \times n$个像素，如果变换是顺序进行的，则变换共需$n \times n$步，因此有:

$$t_s = 2n^2$$

故顺序时间复杂性为$O(n^2)$。

并行的时间复杂性由通信和计算两部分组成，所有的消息传递并行计算实际上都是由这两部分组成的。在本书中，我们将分开处理通信和计算，然后再将每一部分合在一起以形成总的并行时间复杂性。

（1）通信 回顾一下在2.3.2节中所给出的有关通信分析的基础，我们假设处理器间是直接连接的，且从以下公式开始分析:

$$t_{comm} = t_{startup} + mt_{data}$$

其中$t_{startup}$是形成消息和启动传递所需的（固定）时间，t_{data}是发送一个数据项所需的（固定）时间，而m是数据项数。由t_{comm}给定的通信时间的并行时间复杂性为O(m)。但在实际中，我们通常不能忽略通信的启动时间$t_{startup}$，因为启动时间是一个较大的值，除非m特别大。

设进程数为p。在计算前，必须向每个进程发送开始行号。在前面的伪代码中，我们顺序地发送行号，即p个send()，每次有一个数据项。各个进程必须将自己的像素组经变换后的坐标送回给主进程，这里用各个send()来表示。要送回主进程的数据项为$4n^2$个，主进程将顺序地接收它们。因此通信时间为:

$$t_{comm} = p(t_{startup} + t_{data}) + n^2(t_{startrp} + 4t_{data}) = O(p + n^2)$$

（2）计算 并行实现（用行或列组或是正方/矩形区域）将图像分成若干组，每组有n^2/p个像素。每个像素需要两次加法（参见上述伪代码的从进程）。因此并行计算时间由下式给定:

$$t_{comp} = 2\left(\frac{n^2}{p}\right) = O(n^2/p)$$

（3）总的执行时间 总的执行时间由下式给定:

$$t_p = t_{comp} + t_{comm}$$

对于固定的处理器数，时间复杂性为O(n^2)。

（4）加速比系数 加速比系数为:

$$\text{加速比系数} = \frac{t_s}{t_p} = \frac{2n^2}{2\left(\frac{n^2}{p}\right) + p(t_{startup} + t_{data}) + n^2(t_{startup} + 4t_{data})}$$

85

（5）计算/通信比 如在第1章中所述，计算时间和通信时间的比为:

$$\text{计算/通信比} = \frac{\text{计算时间}}{\text{通信时间}} = \frac{t_{comp}}{t_{comm}}$$

从该式可看到通信开销的影响，特别是在增大问题规模时。就刚才所考虑的问题，该比例为:

$$\text{计算/通信比} = \frac{2(n^2/p)}{p(t_{startup} + 2t_{data}) + 4n^2(t_{startup} + t_{data})} = O\left(\frac{n^2/p}{p + n^2}\right)$$

在处理器数固定时当问题规模增大时该比值为常数。这不利于计算/通信比！理想的顺序算法时间复杂性应是最小幂之一（最小增长）；相反，理想的计算/通信比应是最大幂之一，因为增加问题规模会减少通信（通常是很大的）的影响。（可以使用通信/计算比，且作为顺

序时间复杂性的最期望的比将有最小的幂。）

事实上在大多数实际情况中，隐藏于通信部分的常数远远超出了那些隐藏于计算中的常数。这里t_{comm}中的启动时间为$4n^2 + p$。该代码的运行性能很差。因此减少要传送的消息数就显得非常重要。我们可通过向所有进程广播行号集以减小启动时间的影响，此外我们也可成组送回结果。即使如此，由于计算时间是最小的，故通信控制整个执行时间。事实上该应用可能最适用于共享存储器多处理机，因为此时位图将保存在共享存储器中，对所有处理器而言它将是立即可用的。

3.2.2 曼德勃罗特集

显示曼德勃罗特（Mandelbrot）集是处理位映射图像的另一个例子。但现在必须对图像进行计算，且计算量是很大的。曼德勃罗特集是复数平面中的点集，当对一个函数迭代计算时，这些点将处于拟稳定的状态（将增加或减小，但不会超过某一限度），通常该函数是：

$$z_{k+1} = z^2_k + c$$

式中z_{k+1}是复数$z = a + bi$（其中$i = \sqrt{-1}$）的第$k + 1$次迭代，z_k是z的第k次迭代，c是确定该点在复平面中位置的复数。z的初值为0。迭代将一直进行下去，直至z的幅值大于2（它表明z最终将变为无穷大），或是迭代次数已达到某种任意规定的限度。z的幅值是向量的长度，由下式给定：

86

$$z_{length} = \sqrt{a^2 + b^2}$$

对于复数函数$z_{k+1} = z^2_k + c$的计算可以化简，这是因为

$$z^2 = a^2 + 2abi + bi^2 = a^2 - b^2 + 2abi$$

即实部是$a^2 - b^2$，而虚部是$2ab$。因此，若用z_{real}表示z的实部，而用z_{imag}表示z的虚部，则下一迭代值可由以下计算得到：

$$z_{real} = z^2_{real} - z^2_{imag} + c_{real}$$

$$z_{imag} = 2z_{real}z_{imag} + c_{imag}$$

其中c_{real}是c的实部，而c_{imag}则是c的虚部。

1. 顺序代码

在编程时可用一个结构保存z的实部和虚部：

```
structure complex {
   float real;
   float imag;
};
```

对一点的值进行计算并返回迭代次数的例程形式可能如下：

```
int cal_pixel(complex c)
{
int count, max_iter;
complex z;
float temp, lengthsq;
max_iter = 256;
z.real = 0;
z.imag = 0;
count = 0;                                  /* number of iterations */
do {
   temp = z.real * z.real - z.imag * z.imag + c.real;
```

```
        z.imag = 2 * z.real * z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real * z.real + z.imag * z.imag;
        count++;
    } while ((lengthsq < 4.0) && (count < max_iter));
    return count;
}
```

长度的平方lengthsq与4相比较，而不是与2比较，这是为了避免进行开方运算。根据给定的终止条件，所有的曼德勃罗特点必将是处在以原点为中心、半径为2的圆中。

计算和显示这些点的代码需要对坐标系统进行一定缩放来与显示区域的坐标系统相匹配。实际的观看区域通常是一个任意大小的矩形窗口，并可放置到复平面中任何感兴趣的位置。 87
分辨率可随意提高以获得精彩的图像。假设显示高度为disp_height，显示宽度为disp_width，而点在显示区域中的位置是(x，y)。如果显示复平面的窗口具有最小值(real_min，imag_min)和最大值(real_max，imag_max)，则每个(x，y)点需用以下的系数加以缩放

```
c.real = real_min + x * (real_max - real_min)/disp_width;
c.imag = imag_min + y * (imag_max - imag_min)/disp_height;
```

来获得真正的复平面坐标。为提高计算效率，设

```
scale_real = (real_max - real_min)/disp_width;
scale_imag = (imag_max - imag_min)/disp_height;
```

包括缩放的代码如下：

```
for (x = 0; x < disp_width; x++)          /* screen coordinates x and y */
    for (y = 0; y < disp_height; y++) {
        c.real = real_min + ((float) x * scale_real);
        c.imag = imag_min + ((float) y * scale_imag);
        color = cal_pixel(c);
        display(x, y, color);
    }
```

其中display()是一个精心编写的例程用来显示经计算得到的颜色的像素(x，y)(如果必要的话，需考虑显示窗口中原点位置)。图3-4中示出了典型的结果。用Xlib调用图形学算法生成曼德勃罗特集的顺序程序版本可从http://www.cs.uncc.edu/par_prog处获得，并可用它作为简单并行程序编写的基础(习题3-7)。 88

由于曼德勃罗特集的计算是密集计算，故在并行计算机系统(以及顺序计算机)中它被广泛用来测试。依赖于计算机的速度和所需的图像分辨率，整个图像的计算将需几分钟的时间。另外它有很有趣的图形结果，如果想对选定区域加以放大，这只要对该区域进行重复计算并且对显示例程的结果进行缩放就可实现。

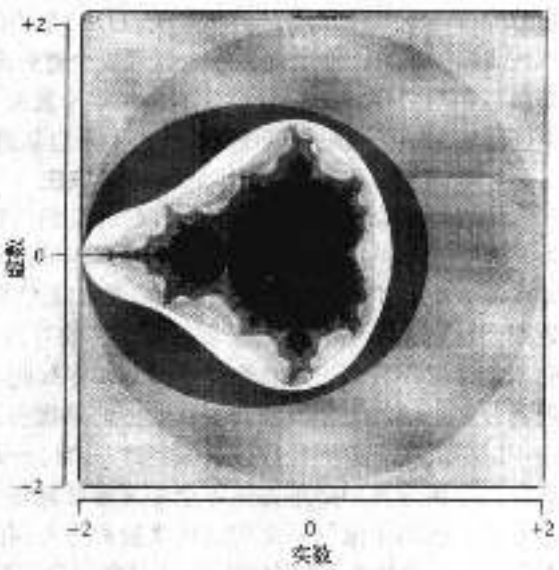


图3-4 曼德勃罗特集

2. 并行化曼德勃罗特集的计算

对消息传递系统而言，曼德勃罗特集特别便于并行化，因为每个像素的计算不需要其周围像素的任何信息。每个像素本身的计算倒是不易并行化。在前面变换图像的例子中，我们为每个进程分配了一个固定的显示区域，这种分配是*静态分配*。下面我们将讨论当进程所计算的区域发生变化时的静态分配和*动态分配*。

3. 静态任务分配

如图3-3所示，用正方/矩形区域或用列/行进行编组是很合适的。在给定了要计算的像素坐标后，每个进程需执行过程`cal_pixel()`。假定显示区域为640 × 480，而在一个进程中要计算10行（即将10 × 640像素编为一组，共48个进程）。则其伪代码形式可能如下所示：

```
Master

for (i = 0, row = 0; i < 48; i++, row = row + 10)/* for each process*/
   send(&row, Pi);                                  /* send row no.*/
for (i = 0; i < (480 * 640); i++) {   /* from processes, any order */
   recv(&c, &color, PANY);            /* receive coordinates/colors */
   display(c, color);                 /* display pixel on screen */
}

Slave (process i)

recv(&row, Pmaster);                  /* receive row no. */
for (x = 0; x < disp_width; x++)      /* screen coordinates x and y */
   for (y = row; y < (row + 10); y++) {
      c.real = min_real + ((float) x * scale_real);
      c.imag = min_imag + ((float) y * scale_imag);
      color = cal_pixel(c);
      send(&c, &color, Pmaster);      /* send coords, color to master */
   }
```

我们期望发送所有640 × 480像素，但根据计算像素值的迭代次数和计算机的速度这些像素可能以任何顺序出现。这种实现与3.2.1节中的变换有同样的严重缺点：每次只送回一个而
89 不是成组的结果。成组地发送数据可减少通信启动次数（每发一个消息需启动一次）。先将结果保留在数组中，然后以一个消息将整个数组发送给主进程是很容易实现的。应注意的是，主进程将用一个通配符以任何顺序接收来自从进程的消息（用记号P_{ANY}指明源通配符）。

4. 动态任务分配——工作池/处理器农庄

曼德勃罗特集对每个像素要进行大量的迭代计算。一般每个像素的迭代次数均不同。此外，各台计算机可能是不同型号或是以不同速度运行。因此某些处理器可能先于其他处理器完成所分配的任务。理想的情况是，我们希望所有处理器同时完成，从而可达到100%的系统效率，这就是所谓的*负载平衡*。在所有的并行计算中，这是一个复杂但却非常重要的概念，而不是仅仅在我们正在讨论的这个问题中如此。理想的情况是，在整个计算期间，为每个处理器分配足够的工作以使其始终处于忙碌状态。可为不同的处理器分配不同大小的区域，但由于以下的两个原因往往很难做到这一点：一是我们事先并不一定知道每个处理器的计算速度；二是因为我们必须知道每个处理器计算每个像素所需的准确时间，但后者是依赖于迭代次数的，而每个像素所需的迭代次数都是不同的。求解有些问题的计算时间较为一致，但不管怎样，一个较为系统有效的方法必将包含某种形式的动态负载平衡。

动态负载平衡可使用工作池方法加以实现，在这种方法中当有处理器处于空闲时就向其分配工作。有时用处理器农庄（processor farm）这一术语来描述这一方法，特别是当所有处理器都是同一种类型时。工作池中拥有所有要完成的任务集（或池）。在某些工作池问题中，进程能产生新的任务，我们将在第7章看到它的含义。

在我们的问题中，像素集（或更确切地是它们的坐标集）构成了任务。任务数是固定的，因为要计算的像素数在计算开始前是已知的。各个处理器从工作池中请求获得像素的一对坐标。当一个处理器完成像素颜色的计算后，它就返回此颜色，并从工作池中请求下一个像素的一对坐标。当所有像素坐标均已从工作池中取走后，然后就需等待所有处理器完成它们的任务，并要求输入更多的像素坐标。

逐个发送各个像素的一对坐标将造成过度的通信。为此一般不将单个坐标作为任务，而是将代表若干像素的一组坐标作为任务，从工作池中取出，这样就可减小通信开销。在开始时，从进程被告知像素组的大小（假定是为固定大小）。然后只需将组内第一对坐标作为一个任务发送给从进程即可。此方法将把通信开销减小到可接受的程度。图3-5中示出了整个安排。

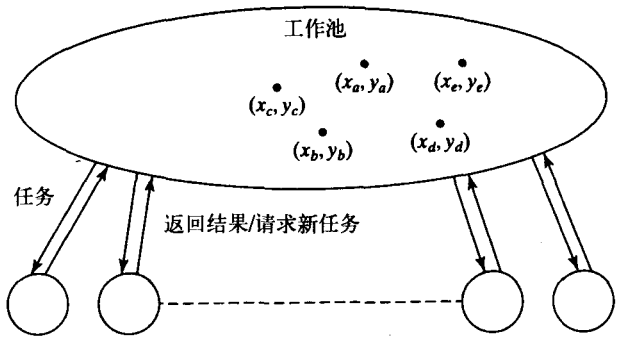


图3-5 工作池方法

当曼德勃罗特集计算以工作池解法编码时，我们发现像素不是一起产生的。某些像素先于其他像素出现。（实际上，这种情况在静态分配编码时也会出现，因为某些像素将比其他像素需要更多的计算时间，而且消息被接收的次序也不受限制。）

90

假定进程数用num-proc表示，且进程每次计算一行。在这种情况下，工作池将保持行号而不是各个像素的坐标。工作池的编码形式可能如下：

主进程

```
count = 0;                                  /* counter for termination*/
row = 0;                                    /* row being sent */
for (k = 0; k < num_proc; k++) {          /* assuming num_proc<disp_height */
   send(&row, P_k, data_tag);               /* send initial row to process */
   count++;                                 /* count rows sent */
   row++;                                   /* next row */
}

do {
   recv (&slave, &r, color, P_ANY, result_tag);
   count--;                              /* reduce count as rows received */
   if (row < disp_height) {
      send (&row, P_slave, data_tag);                    /* send next row */
      row++;                                             /* next row */
      count++;
   } else
      send (&row, P_slave, terminator_tag);              /* terminate */
   display (r, color);                                   /* display row */
} while (count > 0);
```

从进程

```
recv(y, P_master, ANYTAG, source_tag);      /* receive 1st row to compute */
while (source_tag == data_tag) {
   c.imag = imag_min + ((float) y * scale_imag);
   for (x = 0; x < disp_width; x++) {              /* compute row colors */
      c.real = real_min + ((float) x * scale_real);
      color[x] = cal_pixel(c);
   }
   send(&x, &y, color, P_master, result_tag);      /* row colors to master */
   recv(&y, P_master, source_tag);                 /* receive next row */
};
```

91

在该段代码中，首先让每个从进程计算一行，此后当它返回一个结果后就可得到另一行，直到没有要计算的行为止。当所有行均被发送后，主进程就发送一个终止符消息。为区别不同的消息就需使用标记，对发送给从进程的行消息使用`data_tag`，对终止符消息使用`terminator_tag`，对来自从进程的计算结果使用`result_tag`。为此就必须有一种机制来识别所接收到的不同标记。这里我们简单地以`source_tag`参数来表明。应注意的是，在显示结果之前主进程进行接收和发送消息，这样就可使从进程尽快重新启动，为此使用本地阻塞发送。还应注意的是，为了实现终止，需要对从进程中未完成的行数进行计数（`count`），如图3-6所示的那样。当然也可以简单地计算返回行数。当然，还可用其他方法对该问题编码。在第7章中我们将对终止问题做进一步的阐述。

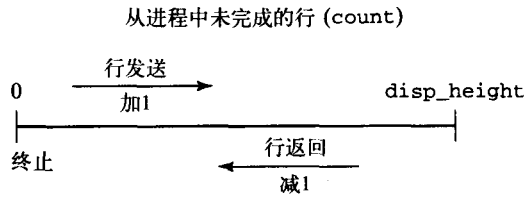


图3-6　计数器终止

分析

对曼德勃罗特计算的确切分析是非常复杂的，这是因为每个像素需进行多少次迭代事先并不知道。每个像素的迭代次数是c的某一函数，但不会超过`max_iter`。因此顺序时间为：

$$t_s \leqslant \texttt{max_iter} \times n$$

或者顺序时间复杂性为O (n)。

就并行版本而言，我们只考虑静态分配。设处理器的总数为p，则有$p-1$个从进程。并行程序基本上有三个阶段：通信、计算和通信。

阶段1：通信　首先，向每一个从进程发送行号，即向$p-1$个从进程的每一个发送一个数据项；即

$$t_{\text{comm1}} = (p-1)\,(t_{\text{startup}} + t_{\text{data}})$$

向每个从进程发送独立的消息导致了重复的启动时间。可以使用一个分散例程，它将减小这一影响（习题3-6）。

阶段2：计算　然后各个从进程并行地完成它们的曼德勃罗特计算；即

$$t_{\text{comp}} \leqslant \frac{\texttt{max_iter} \times n}{p-1}$$

92

假设像素在所有处理器中平均分配。至少会有某些像素将需要最大的迭代次数，而它们将在整个时间中占主要部分。

阶段3：通信　在最后的阶段中，结果传送回主进程，一次传送像素颜色的一行。假设每个从进程处理u行，并且每行有v个像素，于是：

$$t_{\text{comm2}} = u\,(t_{\text{startup}} + vt_{\text{data}})$$

启动时间开销的降低可以通过将结果收集到更少的消息中来实现。就静态分配而言，v的值（每

行的像素数）和u的值（行数）都是固定的（除非图像的精度改变）。不妨设$t_{comm2} = k$，k是一个常数。

总执行时间 综合起来，并行时间由下式给定：

$$t_p \leqslant \frac{\texttt{max_iter} \times n}{p-1} + (p-1)(t_{startup} + t_{data}) + k$$

加速系数 加速系数如下：

$$\text{加速系数} = \frac{t_s}{t_p} = \frac{\texttt{max_iter} \times n}{\frac{\texttt{max_iter} \times n}{p-1} + (p-1)(t_{startup} + t_{data}) + k}$$

加速系数可能接近p，如果max_iter足够大。

计算/通信比 计算/通信比如下：

$$\text{计算/通信比} = \frac{(\texttt{max_iter} \times n)}{(p-1)((p-1)(t_{startup} + t_{data}) + k)}$$

$$= O(n)，\text{当处理器数固定时}$$

前面的分析仅是为了说明是否值得进行并行化，看来结论是肯定的。

3.2.3 蒙特卡罗法

蒙特卡罗法(Monte Carlo method)的基本思想是在计算中使用随机选择以求解数值和物理问题。由于每个计算与其他计算是独立的，因此属于易并行方法。蒙特卡罗（Monte Carlo）这一名字是“在二次大战实施曼哈顿计划期间由Metropolis创造的，因为统计模拟与游戏中的获胜机会有类似性，而摩纳哥首都蒙特卡罗在那时是赌博中心”。

在文献中多次重复出现的一个例子是计算π值[Fox et al., 1998; Gropp, Lusk and Skjellum, 1994; Kalos and Whitlock, 1986]的方法如下。在正方形中有一内切圆，如图3-7所示。该圆的半径为1，故正方形的面积为2 × 2。圆面积与正方形面积的比由下式给定：

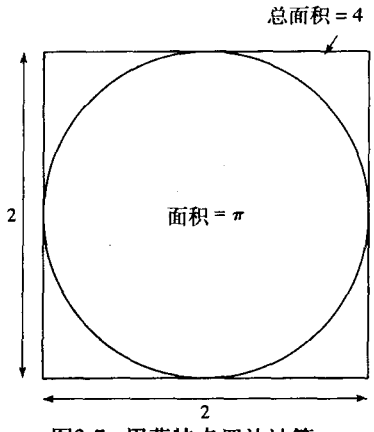


图3-7 用蒙特卡罗法计算π

$$\frac{\text{圆面积}}{\text{正方形面积}} = \frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}$$

（只要圆内切于正方形对任何尺寸的圆都会得到相同结果）。随机地从正方形中选点，并记录有多少点恰好也落在圆内。只要随机地选择足够多的采样点，那么处于圆中的点与全部点的比例将为π/4。

在已知界定区域内的任何形状的面积都可用上述方法计算或是在曲线下的面积用积分求解。图3-7中的四分之一圆，作为函数表示已示于图3-8，它可用积分描述为：

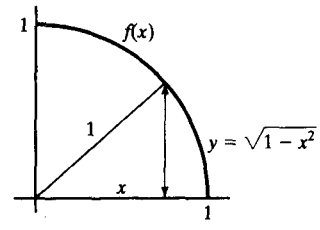


图3-8 用蒙特卡罗法计算π时的被积函数

$$\int_0^1 \sqrt{1-x^2}\,dx = \frac{\pi}{4}$$

(正的平方根)。将生成随机数对(x_r, y_r)，每一个数都处于0和1之间，如果$y_r \leq \sqrt{1-x_r^2}$就将它计为处在圆内，即有$y_r^2 + x_r^2 \leq 1$。

该方法可用来计算任何的定积分。不幸的是，该方法的效率很低，且需要知道所感兴趣区域内函数的最大值和最小值。另一种求积分方法是概率法，它用x的随机值计算$f(x)$，并累加$f(x)$的值:

$$\text{面积} = \int_{x_1}^{x_2} f(x)dx = \lim_{N\to\infty} \frac{1}{N} \sum_{r=1}^{N} f(x_r)(x_2 - x_1)$$

94

式中x_r是x在x_1和x_2之间随机生成的值，该方法也可算为蒙特卡罗法，虽然有些人只将在求解时不选择随机值的方法归属于蒙特卡罗法。已有论著对蒙特卡罗法做了重要的数学支持，在[Kalos and Whitlock，1986]这样的教科书中可看到有关材料。在实际情况中，蒙特卡罗法并不用于求解单重积分，因为用求面积定积分方法的效果更好（参见4.2.2节）。但是蒙特卡罗法对求解具有大量变量的积分显得非常有用，因而在这些情况下，它是非常实用的。

下面让我们来看一下用$f(x) = x^2 - 3x$作为具体例子时，如何实现蒙特卡罗法，也就是要计算积分

$$I = \int_{x_1}^{x_2} (x^2 - 3x)dx$$

1. 顺序代码

顺序代码可为如下形式:

```
sum = 0;
for (i = 0; i < N; i++) {              /* N random samples */
   xr = rand_v(x1, x2);                /* generate next random value */
   sum = sum + xr * xr - 3 * xr;       /* compute f(xr) */
}
area = (sum / N) * (x2 - x1);
```

例程`rand_v(x1,x2)`将返回处于`x1`和`x2`之间的一个伪随机数。代码中选取了固定的采样数；事实上积分计算应达到某一规定精度，这就要进行统计分析以确定所需的采样数。有关蒙特卡罗法的教科书，如[Kalos and Whitlock，1986]，对统计系数做了详尽的研究。

2. 并行代码

由于迭代相互独立，所以该问题是易并行的。问题的焦点是如何使每次计算都能使用不同的随机数且这些数之间又不存在相关性的方式生成随机数。可使用`rand()`这样的标准库伪随机数生成器（稍后论及）。在[Gropp、Lusk，and Skjellum，1994]中使用的方法是让一个独立的进程负责生成下一个随机数。图3-9对此结构做了说明。首先，由主进程启动为它们的每次计算向随机数进程请求一个随机数的从进程；然后由各个从进程形成各自的部分和，并将其返回给主进程，再由主进程进行最后的累加。如果要求每个从进程完成相同的迭代次数且该系统是同构的（相同处理器），那么各从进程将基本同时完成。

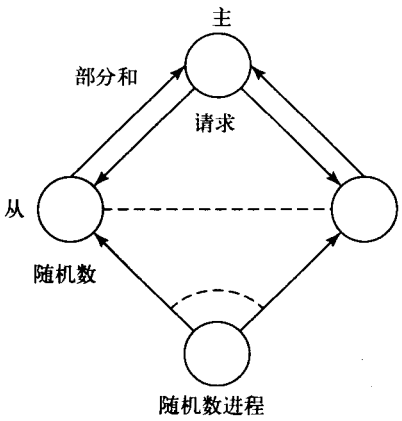


图3-9　并行蒙特卡罗积分

随机数进程每次只能为一个从进程服务，并且该方法只能向各个从进程逐个发送随机数，

从而具有很大的通信开销。可将随机数成组发送以减少启动时间的影响。也可将随机数生成器的代码放在主进程中，因为在整个计算期间这个进程在主进程中不会起作用。综上所述就可得到如下的并行伪代码形式: 95

主进程

```
for(i = 0; i < N/n; i++) {
   for (j = 0; j < n; j++)        /*n=number of random numbers for slave */
      xr[j] = rand();               /* load numbers to be sent */
   recv(P_ANY, req_tag, P_source);     /* wait for a slave to make request */
   send(xr, &n, P_source, compute_tag);
}
for(i = 0; i < num_slaves; i++) {  /* terminate computation */
   recv(P_i, req_tag);
   send(P_i, stop_tag);
}
sum = 0;
reduce_add(&sum, P_group);
```

从进程

```
sum = 0;
send(P_master, req_tag);
recv(xr, &n, P_master, source_tag);
while (source_tag == compute_tag) {
   for (i = 0; i < n; i++)
      sum = sum + xr[i] * xr[i] - 3 * xr[i];
   send(P_master, req_tag);
   recv(xr, &n, P_master, source_tag);
};
reduce_add(&sum, P_group);
```

在上述代码中，主进程用一个源通配符（P_{ANY}）等待任何从进程的响应。实际从进程响应的序号可以从状态调用或参数中获得。我们简单地在消息信封中指明源。用握手方式通信非常可靠，但与不带请求的简单发送数据相比，它有额外的通信开销；正如我们所见到的那样，为获得高的执行速度，减小通信开销也许是最为重要的。如何取消握手过程将作为一个练习 96 留给读者自己解答。例程`reudce_add()`是我们用来指明集合完成加法的归约例程的记号，并用记号P_{group}说明一个进程组。

3. 并行随机数的生成

要成功进行蒙特洛罗模拟，随机数之间必须相互独立。生成伪随机数序列x_1，x_2，…，x_{i-1}，x_i，x_{i+1}，…，x_{n-1}，x_n的最流行的方法，是从精心选择的x_i的函数从而求得x_{i+1}。问题的关键是要找到一个能以正确的统计特性生成很长的序列的函数。通常所用的这种函数的形式为:

$$x_{i+1} = (ax_i + c) \bmod m$$

其中a、c和m为常数，选用它们是为了生成一个具有类似于真正随机序列特征的序列。使用这种方程形式的生成器称为**线性同余生成器**（linear congruential generator）。a、c和m可选取许多不同值，其中大多数的这种生成器的统计特性已公开报导（参见[Knuth,1981]、标准参考书目，以及[Anderson，1990]）。一个“好”的生成器对这三个常数的取值为$a = 16807$，$m = 2^{31}-1$（质数）以及$c = 0$ [Park and Miller,1998]。该生成器能生成一个有$2^{31}-2$个不同数的重复序列（即这类生成器能产生的最大数为$m-1$）。用这类生成器进行蒙特卡罗模拟的缺点是

其速度较慢。

尽管伪随机数的计算本质上是顺序的，因为每个数需从前一数计算得到，但并行公式实现生成序列的速度的增长仍是可能的。它具有以下形式:

$$x_{i+1} = (ax_i + c) \bmod m$$

$$x_{i+k} = (Ax_i + C) \bmod m$$

式中$A = a^k \bmod m$，$C = c\,(a^{k-1} + a^{k-2} + \cdots + a^1 + a^0) \bmod m$，且$k$是一个所选的“跳跃”常数。对$A$和$C$进行计算和使用时必须小心，这是因为它们含有大量数的缘故，但它们仅需计算一次（对先前叙述的好的生成器来讲，不需要C。）若给定m个处理器，则序列中的前m个数将顺序生成。此后这些数中的每一个可并行地用来创建下m个数，如图3-10所示，接着再生成下m个数，以此类推。

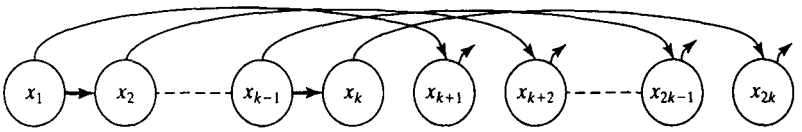


图3-10　序列的并行计算

如果m为2的乘方，那么计算就可简化，因为取模运算将简单地返回低m位。不幸的是，这种形式的生成器常要对m使用一个质数以获得良好的伪随机数特征。[Fox，Williams and Messina，1994]叙述了使用$x_i = (x_{i-63} + x_{i-127}) \bmod 2^{31}$公式的不同类型的随机数生成器，它能
97 自然地从与以前的数有较远距离的数中生成数。

有好几种类型的伪随机数生成器，每种类型使用不同的数学公式。一般而言，公式用前面生成的数来计算序列中的下一个数，这样所有的数就是确定的和可重复的。可以重复这一点有利于测试程序，但公式必须统计地生成好的随机数序列。测试随机数生成器的研究领域已有很长的历史，但这已超出本书的范围。然而在选择和使用随机数生成器时需要特别小心。某些早期的随机数生成器在相对于特征统计数字的测试时已显出相当差的特性。即使一个随机数生成器从统计测试中显出能创建一系列随机数，我们仍不能肯定取自序列的不同子序列或数的采样是不相关的。这就使得在并行程序中使用随机数生成器充满了困难，因为在并行程序中使用简单的原始方法可能不会产生随机序列，而且程序员也许没有意识到这一点。

一般在并行程序中，如我们已描述的那样可能试图依赖一个集中式线性同余伪随机数生成器向请求的从进程发送数。除了引起通信开销和在集中式通信点处造成瓶颈以外，还存在这样一个很大的问题即是否每一个从进程得到的真是一个随机序列，且这些序列本身不存在某种形式的相关（可通过统计确认这一点）。另外，所有的随机数生成器会在某点重复它们的序列，而使用单个的随机数生成器将更可能使它形成序列的重复。另一种方案为每个从进程使用一个独立的伪随机数生成器。但是，如果使用相同的公式即使是起始于不同的数字，在多处理器中仍可能出现部分的相同序列，退一步讲即使这种情况不会发生，我们仍然不能立即假设在序列间不存在相关。可以看到为并行程序想出好的伪随机序列是一个的挑战性问题。

由于伪随机数生成器在并行蒙特卡罗计算中至关重要，已为发现可靠伪随机数生成器的并行版本作了不少努力。SPRNG（Scalable Pseudorandom Number Generator，可扩展伪随机数生成器）是一个专门用于并行蒙特卡罗计算的库并且为并行进程生成随机数流。该库有几个不同的生成器，其主要特征是能使进程间的相关性最小化以及为MPI提供了一个接口。

对于在并发进程间肯定不存在交互的易并行蒙特卡罗模拟，使用子序列也许是满意的（如果全序列是随机的则这些子序列可能是相关的），且计算完全与顺序完成的一样。对此问

题的进一步探讨作为一个习题留给读者（习题3-14）。

3.3 小结

本章介绍了以下概念：

- （理想的）易并行计算
- 易并行问题和分析
- 分割二维数据集
- 获得负载平衡的工作池方法
- 计数器终止算法
- 蒙特卡罗法
- 并行随机数生成

98

推荐读物

[Fox，Williams，and Messina，1994]提供了关于易并行应用问题（独立并行性）的重要研究细节。有关图形学的大量细节可在[Foley et al.，1990]中找到。有关图像处理的内容则可在[Haralick and Shapiro，1992]中找到，我们将在第11章中对此主题做进一步的探讨。[Dewdney，1985]撰写了一系列有关编写计算曼德勃罗特集的论文。有关蒙特卡罗模拟的细节可在[Halton，1970]、[Kalos and Whitlock，1986]、[McCracken，1955]和[Smith，1993]中找到。[Fox et al.，1988]以及[Gropp，Lusk，and Skjellum，1999]对并行实现做了讨论。有关并行随机数生成器的讨论可参见[Bowan and Robinson，1987]、[Foster，1995]、[Fox，Williams，and Messina，1994]、[Hortensius，McLeod，and Card，1989]，以及[Wilson，1995]。即使在顺序程序中也应非常谨慎地使用随机数生成器。有关在这种应用中的不同随机数生成器的研究可在[Wilkinson，1989]中找到。[Masuda and Zimmermann，1996]阐述了专为并行计算使用的随机数生成器的库。有关例子是用MPI编写的。有关SPRNG的细节可在http://sprng.cs.fsu.edu/main.html 中找到。

参考文献

ANDERSON, S. (1990), "Random Number Generators," *SIAM Review*, Vol. 32, No. 2, pp. 221–251.

BOWAN, K. O., AND M. T. ROBINSON (1987), "Studies of Random Generators for Parallel Processing," *Proc. 2nd Int. Conf. on Hypercube Multiprocessors*, Philadelphia, pp. 445–453.

BRÄUNL, T. (1993), *Parallel Programming:An Introduction*, Prentice Hall, London.

DEWDNEY, A. K. (1985), "Computer Recreations," *Scientific American*, Vol. 253, No. 2 (August), pp. 16–24.

FOLEY, J. D., A. VAN DAM, S. K. FEINER, AND J. F. HUGHES (1990), *Computer Graphics Principles and Practice*, 2nd ed., Addison-Wesley, Reading, MA.

FOSTER, I. (1995), *Designing and Building Parallel Programs*, Addison-Wesley, Reading, MA.

FOX, G., M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER (1988), *Solving Problems on Concurrent Processors, Vol. 1*, Prentice Hall, Englewood Cliffs, NJ.

FOX, G. C., R. D. WILLIAMS, AND P. C. MESSINA (1994), *Parallel Computing Works*, Morgan Kaufmann, San Francisco, CA.

GROPP, W., E. LUSK, AND A. SKJELLUM (1999), *Using MPI Portable Parallel Programming with the Message-Passing Interfaces*, 2nd edition, MIT Press, Cambridge, MA.

HALTON, J. H. (1970), "A Retrospective and Perspective Survey of the Monte Carlo Method," *SIAM Review*, Vol. 12, No. 1, pp. 1–63.

HARALICK, R. M., AND L. G. SHAPIRO (1992), *Computer and Robot Vision* Volume 1, Addison-Wesley, Reading, MA.

HORTENSIUS, P. D., R. D. MCLEOD, AND H. C. CARD (1989), "Parallel Random Number Generation for VLSI Systems Using Cellular Automata," *IEEE Trans. Comput.*, Vol. 38, No. 10, pp. 1466–1473.

KALOS, M. H., AND P. A. WHITLOCK (1986), *Monte Carlo Methods*, Volume 1, *Basics*, Wiley, NY.

KNUTH, D. (1981), *The Art of Computer Programming*, Volume 2, *Seminumerical Algorithms*, Addison-Wesley, Reading, MA.

MASUDA, N., AND F. ZIMMERMANN (1996), *PRNGlib: A Parallel Random Number Generator Library*, Technical report TR-96-08, Swiss Center for Scientific Computing (available at http://www.cscs.ch/Official/).

MCCRACKEN, D. D. (1955), "The Monte Carlo Method," *Scientific American*, May, pp. 90–96.

PARK, S. K., AND K. W. MILLER (1988), "Random Number Generators: Good Ones Are Hard to Find," *Comm. ACM*, Vol. 31, No. 10, pp. 1192–1201.

SMITH, J. R. (1993), *The Design and Analysis of Parallel Algorithms*, Oxford University Press, Oxford, England.

WILKINSON, B. (1989), "Simulation of Rhombic Cross-Bar Switch Networks for Multiprocessor Systems," *Proc. 20th Annual Pittsburgh Conf. on Modeling and Simulation*, Pittsburgh, May 4–5, pp. 1213–1218.

WILKINSON B., AND D. HORROCKS (1987), *Computer Peripherals*, 2nd edition, Hodder and Stoughton, London.

WILSON, G. V. (1995), *Practical Parallel Programming*, MIT Press, Cambridge, MA.

习题

科学/数值习题

3-1 试编写一个能以合适的非压缩格式（例如PPM格式）读出一个图像文件并生成一个向右移动N个像素的图像文件的并行程序，这里的N是一个输入参数。

3-2 实现本章中所描述的图像变换。

3-3 重写3.2.1节中的伪代码，以使它能运行在80 × 80的正方形区域而不是行组区域。

3-4 视窗变换涉及到在一个未被显示的图中选择感兴趣的一个矩形区域，并将已得到的视图移植到一个指定的位置显示。需注意的是，所选的矩形区域以ΔX和ΔY度量，且在未显示图像的坐标系统中左下角的坐标为（X，Y）。处于该矩形中的点（x，y）将通过以下的转换公式转换到以$\Delta X'$和$\Delta Y'$度量的矩形中：

99
~
100

$$x' = (\Delta X'/\Delta X)(x-X) + X'$$
$$y' = (\Delta Y'/\Delta Y)(y-Y) + Y'$$

如果$\Delta X'$与ΔX以及$\Delta Y'$与ΔY不相等，则会有缩放。只要有可能就应在进行其他变换前，先进行视窗变换，这样可以减小进行后续变换的计算量。试编写能完成视窗变换的程序。

3-5 以（x，y，z）坐标形式表示的三维图画可用透视转换方法将其投影到一个二维平面上。在做这种变换时，必须要消除隐藏线。在此之前，可对图画进行三维的平移、缩放和旋转变换。将一个三维对象绕x轴旋转θ角度需进行如下的变换：

$$x' = x$$
$$y' = y\cos\theta + z\sin\theta$$
$$z' = z\cos\theta - y\sin\theta$$

对y和z轴可进行类似的旋转变换。编写能完成三维变换的并行程序。

3-6 用分散例程代替单个地向每个从进程发送起始行，重写3.2.2节中进行曼德勃罗特计算的并行伪代码。而且在每个从进程中使用单个消息返回它的集合结果。对你的编码进行必要的分析。

3-7 从http://www.cs.uncc.edu/par_prog/下载顺序的曼德勃罗特程序，并按说明编译和运行该程序。（此程序用Xlib调用图形，它必须与相应的库连接）。修改该程序以使用静态负载平衡运行并行程序（也就是，将图像简单地分成固定区域）。在你的系统执行并测试代码以获得并行执行时间。

3-8 重做习题3-7，但用动态负载平衡。

3-9 继续习题3-7和3-8，在曼德勃罗特计算中对不同的z的起始值进行实验。

3-10 在以下两个函数的基础上，分别编写计算分形（fractal）（分数维）图像的一个顺序和一个并行程序，其中第一个函数：

$$z_{i+1} = z_i^3 + c$$

而第二个函数[Bräunl，1993]为：

$$z_{i+1} = z_i^3 + (c-1)z_i - c$$

式中$z_0 = 0$，而c则以复数形式提供图像中一个点的坐标。

3-11 实验性地比较在3.2.3节中所描述的计算积分的两种蒙特卡罗方法。使用计算π/4的积分：

$$\int_0^1 \sqrt{1-x^2}\,dx$$

3-12 重写3.2.3节中的蒙特卡罗积分代码，要求取消主进程中显式地请求数据部分，并对你的解答进行必要的分析。

3-13 阅读[Hortensius，McLeod and Card，1989]的论文，并根据该论文所描述的方法，编写并行随机数生成器的代码。

3-14 研究在易并行蒙特卡罗模拟中使用取自随机数生成器但存在潜在相关子序列的影响。对这一主题进行文选搜索并写出报告。 101

3-15 整数集合的缩灭（collapse）定义为该集合中的整数之和。类似地，单个整数的缩灭定义为它的各个十进位之和。例如，整数134957的缩灭为29。显然这一过程可递归地实现，直到生成单个十进制数的结果：29的缩灭为11，而11的缩灭为单个十进制数2。一个整数集的最终缩灭就是它们的缩灭跟着递归地缩灭直到只剩下单个十进制数{0，1，…，9}所得到的结果。你的任务是编写一个程序找到由N个整数组成的一维数组的最终缩灭。你可采下面所列出的不同方法：

1）并行使用K台计算机，每台计算机对大约N/K个整数相加，并将它们的局部和送给主进程，由后者将这些部分和加起来，形成该整数的最终缩灭。

2）并行使用K台计算机，每台计算机对N/K个整数的局部集合进行缩灭，并将部分结果送给主进程，由后者生成部分缩灭的最终缩灭。

3）并行使用K台计算机，每台计算机对它们局部的N/K个整数集中的每一个整数逐个地进行最终缩灭，然后将这些局部的已缩灭的整数加在一起，再递归地缩灭这些结果以得到一个单个的十进制数。然后K台计算机中的每一台将它的十进数发送给主进程，做最后的求和和最终的缩灭。

4）按照前述三种方法中的任一种，用单台计算机处理所有N个整数。

5）（特别鼓励做这一小题）证明前面的三种方法在对N个整数集合进行最终缩灭

时，就产生相同的十进制数结果而言是等价的。

现实生活习题

3-16　Kim从她的英语课程中知晓，回文（palindrome）是指忽略大写、从右向左与从左向右阅读完全一样的短语。她回想起该问题的称呼要归功于拿破仑，他被流放到厄尔巴岛并死在那里。由于是属数学头脑类型，Kim将此作为她的业余爱好，在她的汽车里程表中寻找回文245542(她的旧车)或002200（她的新车）。

现在，在她完成计算机科学的大学课程后所找的第一份工作中，Kim再次与回文打交道。作为其工作的一部分，她正在开发另一种安全编码算法，其前提是编码字符不是回文。她的第一项任务是考察一个巨大的字符串集，识别出其中哪些是回文，这样就可从候选编码字符串列表中将它们删除。

字符串由字母（$a, \cdots, z, A, \cdots, Z$）和数字（$0, \cdots, 9$）组成。Kim使用一个一维的指针数组`mylist[]`，其中的每个元素指向一个字符串的起始位置。如同所有字符串一样，字符序列以空字符“\0”终止。Kim的程序须检验每一个字符串并打印出对应于回文的所有字符串的号码（即标识字符串`mylist[]`的下标）。

3-17　Andy、Tom、Bill和Fred花费了一年级的大部分时间一起玩一种简单的纸牌游戏。他们分发一叠52张的纸牌，每人13张。游戏规则类似于桥牌：两个人组成一队，他们与另一队的两个人相隔而坐。所有52张牌以顺时针方向每次分发一张。由发牌者首先出牌，游戏者按顺时针方向依次出牌，且只要可能必须以同一花色跟着出牌；同花色中的最大的牌将赢得一墩四张牌，除非有人出了一张“将牌”；且在这种情况下，最高的将牌将赢得该墩牌。在出完每个人手中所有13张牌后，就将发牌机会转给左边的游戏者。游戏者的目标是为自己的队赢得最多的墩数。由竞叫最多墩数的游戏者确定何种花色为将牌，也就是由他叫出他（她）认为他（她）的队可能赢得的最高墩数。竞叫从发牌者开始，并顺时针方向轮流竞叫，直到四个游戏者均宣布不再竞叫为止。每次后继的竞叫必须至少比前一次高出一墩。如果无人竞叫，就认为发牌者以最小的7墩竞叫成功。

102

但是近来这组中的一位重新对学习有了浓厚的兴趣，结果是在某些晚上只能是少于四个人在一起玩。Tom决定编写一个小型的游戏程序以填补所缺少的游戏者 。Fred则要使其成为一个并行计算的实现，以允许有多于一人的缺赛者。你的工作是帮助Fred实现这一并行计算的程序。

3-18　一家小公司正面临满足服务要求的困难：从一个巨大的数据库中检索数据。该公司的传统做法是将要检索的条款清单交给一位雇员，由他（她）人工地从文件中找到它们。但该公司已有了神速进步，现在它将条款清单交给程序，由该程序从数据库中找出它们。但近来条款清单迅速膨胀到如此大的程度，使得检索过程非常耗时，以致于引来了顾客的投诉。为此该公司提供给你一份工作，要求你用多台机器并行地进行检索，并将要检索的条款清单加以分割，以便在多台机器上完成检索。

第一部分：在将检索进程转向用并行处理实现检索的过程中，要认清你所面临的所有陷阱或障栅，这些困难在现有的串行/单处理器中是不会出现的。

第二部分：对在第一部分中所认清的每一项，理出一个或多个解决办法。

第三部分：模拟一个复合解，并行使用多处理机从大型数据库中检索出清单上所列的

所有条款。

3-19 在过去的35年中，一系列无人参与的雷达映射任务已绘制出月球表面十分详尽的地形图。这些信息已被数字化，且以称为Mercator投影的类似栅格格式供人们使用。下一个无人着陆区域的地形数据包含在100 × 100的栅格点数组中，用来指明在10公里 × 10公里范围内，高出月球表面平均水平面之上（或之下）的高度。选择这一特定着陆的区域是因为它的渐变地形；你可以假设在任何两个邻近栅格点间进行线性插值以精确地描述那些栅格点间的地形。

在火箭着落到所指定的10公里 × 10公里区域内的某地时，它将派遣许多自动机器人。这些机器人将对该区域进行详尽考察，并将测得的结果借助可见光光波通信链路（由机器人发射闪烁光，而由火箭加以检测）传回给火箭。一旦考察完成后，机器人能很快找到可进行可见光通信的最近地点是至关重要的，因为机器人只有很短的电池寿命。

火箭设计者们保证他们的接收天线将在着落点之上20米，而机器人上的发送天线在机器人所在地点的1米之上，无论它在区域中的什么地方。这样，在给定100 × 100数组以及火箭和机器人的栅格点位置后，你的任务是确定离机器人最近的栅格点，在该点上允许机器人以可见光与火箭进行通信。你可以假设地形图数据数组包含的仅是处于+100米到−100米之间的以整数值表示的高度，而火箭和机器人在接受你的程序时将 103
处在栅格点上。

3-20 给定一个100 × 10 000的浮点数数组，这些浮点数用来表示在Nella学院进行烹饪艺术最终考试的一系列“烘焙比赛”中所收集到的数据。如同所有评分系统一样，在给出确切的评分等级前，必须对这些数据进行规范化。对100个学生中的每一个（他的数据在具有10 000个值的行中），必须完成以下操作：

INS（初始规范化评分）

在指定行中所有大于0且小于100的数据值的平方的平均值。

FNS（最终规范化评分）

将该学生的INS的分数与所有其他学生的INS的分数进行比较。如果该生的INS分数处在全部学生的前10%内，则该生将得到4.0的FNS；处在接下来的20%分数段内的学生，得到的FNS为3.0；处在再接下来的30%分数段内的将得到2.0的FNS；处在再往下20%分数段内的学生得分将为1.0的FNS；其余的学生（20%）将得到0.0的FNS，这些学生的结局是只得进入下一年的“烘焙比赛”。

你编写的程序要用以下两种方式打印出FNS的分数：

1）FNS的分数表，对所有学生用学生（行号，FNS）表示。

2）学生表，对所有的FNS值，用FNS（FNS，具有该FNS值的所有行[学生]表）表示。

3-21 近来，出现了某种对公共健康的恐慌，就是有关于在几个大城市的供水系统中出现了细菌神秘孢子（cryptosporidium）。这已经引起了大家的注意，即一个文学恐怖主义者团伙正在扩散细菌，他们将细菌巧妙地秘密地嵌入小说中。你受一家大出版公司的雇用对一本新小说进行搜索，以查找单词cryptosporidium的出现。已知恐怖主义者将采用插入标点符号、大写化以及空格来伪装cryptosporidium的出现；查找该单词的实例比在正文中搜查一个单词要做更多的工作。例如，在一个高度公开案例中，其中的一页用以下的句子结尾：

“Leaving his faithful companion, Ospor, to guard the hallway, Tom crept slowly

down the stairs and entered the darkened crypt"

而在下一页则以以下的句子开头

"Ospor, I dium, HELP!" cried Tom, as the giant bats he had disturbed flew around his head.

当一个职员幸运地偶然发现这可能是一个打印排版错误，并将"I dium"改成"I'm dying"后，就使该书在即将出版时侥幸地避免了一场灾难。由于该出版公司要处理大量书籍，因此尽快地对每本书进行扫描至关重要。为了做到这一点，你提出将任务分成较小的块，然后并行使用多台计算机进行搜查；这样每台计算机将只负责搜查正文的一部分。如果成功的话，你将从把联网计算机销售给出版公司的业务中获得相当可观的佣金。

此外，还有如下的两个方法：

1）将正文分成大小相同的段，并将每一段分配给一个处理器。每个处理器检查所分到的那一段，并将有关信息（是否在该段中发现cryptosporidium，是否发现作为该段的最前面或最后面的字符是潜在的该细菌的一部分，或干脆没有发现任何细菌的证
104 据）返回给主进程，主进程将细查传回给它的信息，并对全书加以整体的报导。

2）将正文分成更多更小的段，其数目将多于处理器数，并使用工作池方法，在这种方法中速度快的处理器将完成更多的工作，但本质上是与前一方法相类似的。

3-22 纳米技术是最近的热点研究领域。它的一个目标是利用大量以并行方式运算的细小部件来求解涉及从环境净化（如清除油溢）、战场清理（排除未爆炸的军火或地雷）到对
105 火星表面的探索和分析的问题。

作为一个并行化的专家，选择纳米技术这些应用领域中的一个，并讨论部件间通信的需求，它要求保证只有少于X%的正在寻找的对象会被漏掉。

第4章　划分和分治策略

本章中，我们将讨论并行程序设计中两种最基本的技术，即所谓划分（partitioning）和分治（divide and conquer），它们二者是紧密相关的。所谓划分就是将问题简单分为几个独立的部分，而且每部分都独立计算。分治技术则是以递归的方式应用划分，在将问题分为几个部分后，并不急于解决这些部分和将结果合并，而是连续地将这些部分分为更小的部分。我们先讨论划分技术，然后再讨论递归的分治方法，接着我们会概述一些可以用这些方法解决的典型问题。和前几章一样，我们会在本章的末尾给出习题，它们包括科学/数值习题和现实生活习题。

4.1　划分

4.1.1　划分策略

划分仅是简单将问题分为几个部分。它是所有并行程序设计的基础，在实际应用中可能有多种不同的形式。第3章中讨论的一些易并行问题就使用了划分技术划分后的部分之间无交互。然而，大多数划分模式需要将各个部分结果合并后才能获得所需的结果。划分技术可以应用于程序的数据（如将数据分解，然后对分解的数据并行操作），这种方法称为*数据划分或域分解*。划分技术也可以应用于程序的功能，（如将程序分为独立的功能模块，然后并发地执行这些功能模块），这种方法称为*功能分解*。将一个任务分为几个小任务，不论这些小任务是操作数据的各个部分还是独立的并发功能，当小任务执行结束时则大任务也就执行结束，这种思想是众所周知的且可以应用于很多场合。在问题中存在并发功能的情况不太常见，所以数据划分是并行程序设计的主要策略。

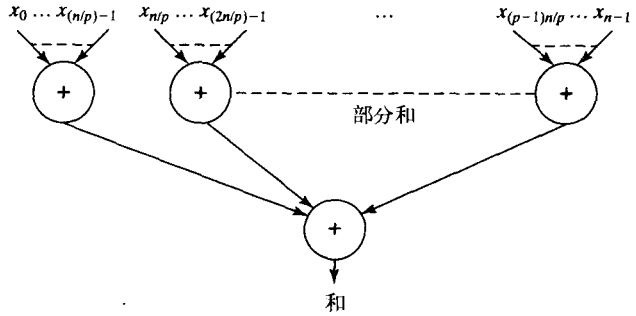
现在让我们考虑一个实际的简单数据划分的例子，假设一个数列，$x_0, \cdots, x_{n-1}$，将它们求和。这是一个在教材中反复出现的用来阐释概念的问题，显然如果数列不是很大，并行的解决方法是不值得的。然而，这种方法可用于解决一些更实际的问题，譬如大型数据库的复杂计算问题。

我们可考虑将数组分为p个部分，每个部分有n/p个数据，$(x_0 \cdots x_{(n/p)-1})$，$(x_{n/p} \cdots x_{(2n/p)-1})$，$\cdots$，$(x_{(p-1)n/p} \cdots x_{n-1})$，这样就可用$p$个处理器（或进程）分别独立地对一个分组求和以获得部分和。p个部分和需要累加起来获得最终和。图4-1展示了在一个处理器上累加p个部分和的情况。（最终相加可以使用一个树结构来并行实现，但在图中没有表示出来。）应注意每个处理器要访问它所累加的每个数据。在消息传递系统中，需要分别将数据传送到每个处理器。（在共享存储器系统中，每个处理器可以从共享存储器中访问它所需的数据，在这一点上，共享存储器系统对于这个问题或类似的问题要比消息传递系统方便得多。）

这个例子的并行代码是很直观的。对于一个简单的主从方式，数据首先由主处理器发送到各个从处理器，每个从处理器独立并发地将自己的数据求和，然后将部分和发送到主处理器，最后主处理器将这些部分和累加求得结果。通常对于代码序列我们更多的是用进程而不是用处理器，在最佳情况下，一个处理器正好对应一个进程。

采用广播的方法将整个数列发送到每个从进程或只给每个从进程发送特定的数据哪种情

况更好些仍是一个未决的问题，因为两种情况下都需要主进程发送所有的数据。为了确定广播机制的相对优点，我们有必要了解广播机制的特点。广播机制只需要一个启动时间，这可能比采用多个发送例程从而需要多个独立的启动时间要优越一些。



107

图4-1　划分一个数列并对每个部分求和

首先，我们使用单独的发送函数send()将指定的数发送到各个从进程。假设给定n个数和p个从进程，其中n/p是个整数，每一组分配给一个从进程，使用独立的send()和recv()的程序代码如下：

主进程

```
s = n/p;                                /* number of numbers for slaves*/
for (i = 0, x = 0; i < p; i++, x = x + s)
   send(&numbers[x], s, P_i);           /* send s numbers to slave */

sum = 0;
for (i = 0; i < p; i++) {               /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;                /* accumulate partial sums */
}
```

从进程

```
recv(numbers, s, P_master);             /* receive s numbers from master */
part_sum = 0;
for (i = 0; i < s; i++)                 /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, P_master);              /* send sum to master */
```

108

如果采用广播或多播例程将完整的数列发送给每个从进程，则从进程的代码需要从中找出需要的那部分数据，这会增加从进程的计算步，代码如下：

主进程

```
s = n/p;                                /* number of numbers for slaves */
bcast(numbers, s, P_slave_group);       /* send all numbers to slaves */
sum = 0;
for (i = 0; i < p; i++) {               /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;                /* accumulate partial sums */
}
```

从进程

```
bcast(numbers, s, P_master);            /* receive all numbers from master*/
```

```
start = slave_number * s;              /* slave number obtained earlier */
end = start + s;
part_sum = 0;
for (i = start; i < end; i++)          /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, P_master);             /* send sum to master */
```

从进程由进程ID来标识，这通常可通过调用一个库例程实现。在更多情况下首先会创立一个进程组，从进程号是该组内的一个实例或序号。当一个组中有m个进程时，实例或序号是一个从0到$m-1$的整数。MPI则需要先建立通信子，同一通信子内的进程有各自的序号，这在第2章中已有叙述。组也可以并入一个通信子，此时进程拥有该组内的一个序号。

如果可以使用分散和规约例程，代码可以为：

主进程

```
s = n/p;                                    /* number of numbers */
scatter(numbers,&s,P_group,root=master);    /* send numbers to slaves */
reduce_add(&sum,&s,P_group,root=master);    /* results from slaves */
```

从进程

```
scatter(numbers,&s,P_group,root=master);    /* receive s numbers */
.                                           /* add numbers */
.
reduce_add(&part_sum,&s,P_group,root=master);  /* send sum to master */
```

请记住，在本书中将到处使用简单的伪代码。分散和规约（以及使用集中例程时）在实际应用中有很多附加的参数，包括源和目的ID。一般而言，规约例程的操作是以一个参数来指定的，而不是这里出现的例程名的一部分。使用参数允许方便地选择不同的操作。代码同样要为参加广播、分散和规约的进程建立一个组。

虽然我们这里是对数求和，其他很多操作也可以同样进行。例如，从进程可以在分组的数中找出局部最大的数发送回主进程，主进程再从这些局部最大值中找出全局最大值。类似地，从进程可以在该数据组中找出某个数（或是字符，或是字符串）的出现次数，然后再发送回主进程。

分析

顺序计算需要$n-1$次加法操作，时间复杂性是O(n)。在并行实现中，有p个从进程。对并行实现的分析，在SPMD模型中我们将假设主进程的操作被认为是从进程中的一个，因为在实际的实现中很可能是这样的。（请记住在MPI中，在集合操作中使用根进程的数据。）因此，处理器数是p。在并行实现中我们的分析将始终独立地考虑通信和计算。如果我们将操作分为不同的阶段，则更容易形象地勾勒出整个程序的过程。像大多数问题一样，一般总是一个通信阶段后面跟着一个计算阶段，这些阶段不断重复。

阶段1：通信 首先，我们要考虑p个从进程读取它们各自的n/p个数据的通信问题。使用单个发送和接收例程需要的通信时间为：

$$t_{comm1} = p(t_{startup} + (n/p)\,t_{data})$$

109

其中$t_{startup}$为传送时间中固定的部分，t_{data}是传送一个数据字的时间。使用分散可以减少启动的次数，即：

$$t_{comm1} = t_{startup} + nt_{data}$$

它依赖于分散的实现。在任何情况下时间复杂性仍然是O (n)。

阶段2：计算 接下来，我们需要估计计算步数。从进程对n/p个数求和，需要$n/p-1$次加

法操作。因为所有p个从进程是同时运行的，因此我们可以认为所有部分和在$n/p-1$步获得。因此，这一阶段的并行计算时间为：

$$t_{comp1} = n/p-1$$

阶段3：通信　使用单个的发送和接收例程返回部分结果需要的通信时间为：

$$t_{comm2} = p\ (t_{startup} + t_{data})$$

使用集中和规约需要的时间为：

$$t_{comm2} = t_{startup} + pt_{data}$$

阶段4：计算　对于最后的累加，主进程必须对p个部分和求和，这需要$p-1$步：

$$t_{comp2} = p-1$$

总的执行时间　该问题的总的执行时间（使用发送和接收）是：

$$\begin{aligned} t_p &= (t_{comm1} + t_{comm2}) + (t_{comp1} + t_{comp2}) \\ &= p\ (t_{startup} + (n/p)\ t_{data}) + p(t_{startup} + t_{data}) + (n/p-1 + p-1) \\ &= 2pt_{startup} + (n + p)\ t_{data} + p + n/p-2 \end{aligned}$$

即，对于固定的处理器数有：

$$t_p = O(n)$$

可以看到并行时间复杂性与顺序时间复杂性$O(n)$是一样的。当然，如果我们只考虑计算方面，并行模式要比顺序模式好。

加速系数　加速系数为

$$\text{加速系数} = \frac{t_s}{t_p} = \frac{n-1}{2pt_{startup} + (n+p)t_{data} + p + n/p - 2}$$

该式提示对于固定的处理器数只能获得小的加速比。

计算/通信比　计算/通信比由下式给定：

$$\text{计算/通信比} = \frac{t_{comp}}{t_{comm}} = \frac{p + n/p - 2}{2pt_{startup} + (n+p)t_{data}}$$

110

同样，对于固定的处理器数，该式再一次提示没有多大的改进机会。

忽略通信部分后，加速系数将由下式给定：

$$\text{加速系数} = \frac{t_s}{t_p} = \frac{n-1}{n/p + p - 2}$$

对于很大的n，加速比趋向p；然而对于较小的n，加速比会很低且会随着从进程的数目增加而降低，因为在第四个阶段形成最终结果时有$p-1$个从进程是空闲的。

理想情况下，我们希望所有的进程在所有时间内都是活动的，但在这个问题模式中是不可能得到的；然而，另一种模式将非常有帮助的且可应用于广泛的问题中，这就是我们接下来要讨论的分治方法。

4.1.2 分治

分治方法的特点是将一个问题分为与原来的较大问题有相同形式的子问题。进一步分为较小的问题通常是通过递归，递归是我们在顺序程序设计中常使用的一种方法。递归的方法会将问题不断分为更小的问题直到不能再分为止。接着就完成这些非常简单的任务并把结果合并，然后按更大的任务继续合并，直到获得最终的结果。[JáJá,1992]区别了在分割问题时的主要工作和合并结果时的主要工作。他将合并结果时的主要工作归类为分治而把分割问题

时的主要工作归类为划分。我们对此不加区别，但我们用分治这个术语表示持续将问题划分为更小的问题。

对一个数列求和的顺序递归定义为[⊖]

```
int add(int *s)                                    /* add list of numbers, s */
{
   if (number(s) <= 2) return (n1 + n2);           /* see explanation */
   else {
      Divide (s, s1, s2);          /* divide s into two parts, s1 and s2 */
      part_sum1 = add(s1);         /*recursive calls to add sub lists */
      part_sum2 = add(s2);
      return (part_sum1 + part_sum2);
   }
}
```

像在所有递归定义中一样，当分割不能再进行时必须有一个方法终止递归。在上面的代码中，number(s)返回由指针s指向的数列中数据的个数。如果只有两个数据，称它们为n1和n2； 111
如果只有一个数，则称它为n1并设n2为0；如果一个数也没有，则n1和n2都为0。独立的if语句可以处理所有情况：数列有0、1或2个数。每种情况都可以结束递归调用。

这种方法也可以用于数列的其他全局操作，例如寻找最大值。还可以通过把数列分割为越来越小的数列，用对小数列排序来实现对大数列的排序。归并排序和快速排序算法通常用于描述这样的递归定义，参见[Cormen, Leiserson,and Rivest,1990]。因为有简单的迭代解决方案存在，实际上没有人会用递归的方法来对数列求和，但接下来的内容对任何用递归分治方法模式化的问题都是适用的。

当每次分割产生两部分时，递归分治模式就会形成一棵二叉数。这棵树在发生调用时下行遍历，在返回结果时上行遍历（对树的前序遍历可以给出递归的定义）。图4-2中的（完全）二叉树显示了分治产生的“分割”部分，最终任务在底部，而根在顶部。根进程首先将问题分为两个部分。这两个部分每个再分为两个部分，重复此过程直到叶进程的位置，在那里完成问题的基本操作。这种结构也可以用于上一个问题，数列先被分为两部分，然后是四部分，直到每个进程都有整个问题的相同部分为止。在将树底部的叶进程成对相加后，累加过程依逆向树结构进行。

图4-2显示了一棵完全二叉树，即带有同一层上所有叶结点的理想平衡树。这种情况发生在一个任务可以分割为2的乘方个子任务时，如果不是2的乘方，则会有一个或多个叶结点要比其他叶结点高一层。为方便起见，如果没有特别说明，我们都假设任务可以分割为2的乘方个子任务。

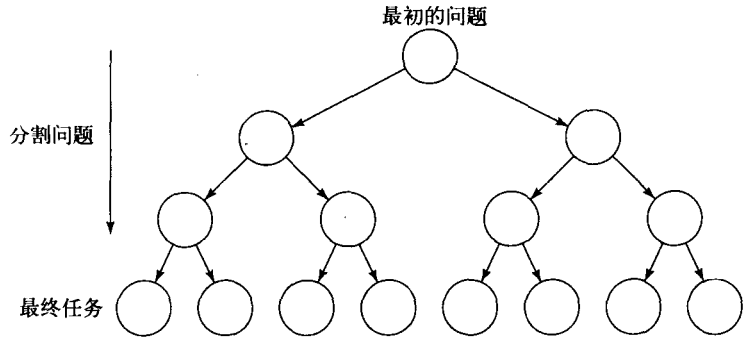


图4-2 树结构

⊖ 像在我们所有的伪代码中一样省略了实现细节。例如，数列的长度也许需要作为参数传递。

1.并行实现

112 在顺序实现中，一次只有一个树的结点可以被访问，并行方案提供同时遍历树的几个部分的前景。一旦一次分割产生了两个部分，这两个部分就可以同时处理。虽然可以模式化一个递归的并行方案，但不用递归更容易对求解直观化，关键是要意识到整个结构是一个树结构。可以为每个树中的结点分配一个处理器，最后需要$2^{m+1}-1$个处理器把任务分为2^m个部分。因为每个处理器仅在树的某一层中处于活动状态，这样会导致一个效率很低的方案（习题4-5讨论了这种方法）。

一个更有效的方案是重用树中每层的处理器，如图4-3所示，其中使用了8个处理器。当所有处理器都使用时分割停止。在此之前，每一步每个处理器自己保留数列的一半，将另一半传送。首先P_0将数组的一半发送给P_4，然后P_0和P_4分别将自己拥有的数列传送一半给P_2和P_6。最后，P_0、P_2、P_4和P_6再分别将自己拥有的数列传送一半给P_1、P_3、P_5和P_7。最后一步时，每个子数列有$n/8$个数。一般情况下p个处理机每个有n/p个数，该树一共分为$\log p$层。

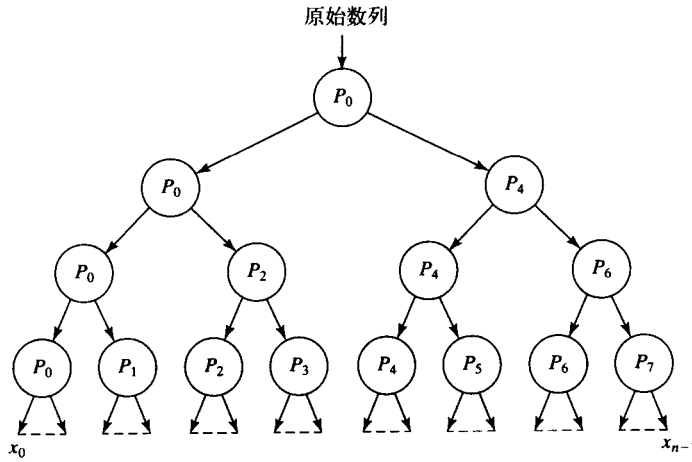


图4-3　分割数列

累加部分和的“合并”过程可以如图4-4所示进行。一旦求出了部分和，每个奇数号的处理器将它的部分和传送给邻近的偶数号的处理器；即P_1发送给P_0，P_3发送给P_2，P_5发送给P_4，以此类推。接着偶数号的处理器将接收的部分和与自己的部分和相加，并如图4-4所示向上传送，直到P_0得出最终结果。

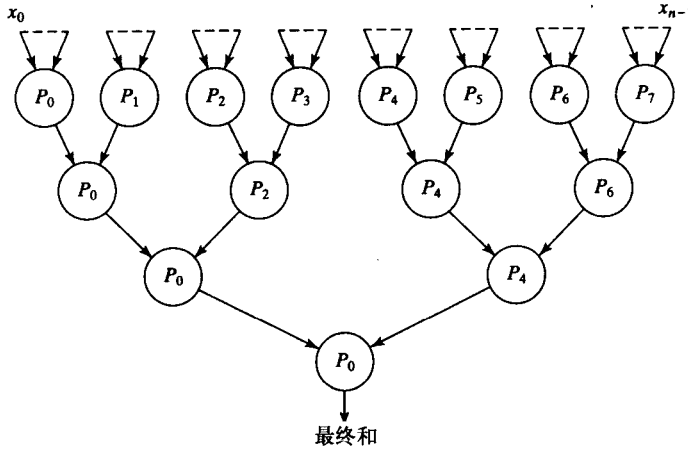


图4-4　部分求和

可以看到这些结构和描述的二进制超立方体广播和集中算法是一样的。这些结构可以很好地映射到一个超立方体，当然也可以应用到其他系统。根据超立方体广播/集中算法，与另一台处理器通信的处理器可通过它们的二进制地址找到，处理器与那些地址与其地址有一位不同的处理器通信，分割阶段从最高位开始，合并阶段从最低位开始。

假设我们静态创建8个处理器（或进程）来对数列求和。进程P_0的并行代码可以为如下形式: 113

进程P_0

```
                                    /* division phase */
divide(s1, s1, s2);                 /* divide s1 into two, s1 and s2 */
send(s2, P4);                       /* send one part to another process */
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P1);
part_sum = *s1;                     /* combining phase */
recv(&part_sum1, P1);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
part_sum = part_sum + part_sum1;
```

进程P_4的代码可能采用如下形式:

进程P_4

```
recv(s1, P0);                       /* division phase */
divide(s1, s1, s2);
send(s2, P6);
divide(s1, s1, s2);
send(s2, P5);                                                    114
part_sum = *s1;                     /* combining phase */
recv(&part_sum1, P5);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P6);
part_sum = part_sum + part_sum1;
send(&part_sum, P0);
```

其他进程的代码类似。显然，其他相关的运算，例如减法、逻辑或、逻辑与、最小、最大或字符串并置，都可以替代上例中的加法运算。这种思想也可应用于求操作数与算术操作符相连接的算术表达式，树结构也可用于搜索等操作。在这种情况下，上传的信息是一个布尔符号，用以表示指定的项或条件是否找到。此时在每个结点执行的是或操作，如图4-5所示。

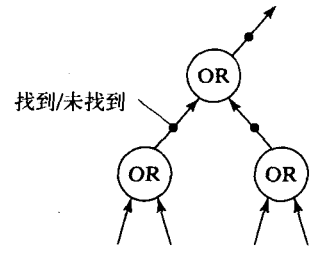


图4-5 搜索树的一部分

2. 分析

我们假设n是2的乘方，为简单起见不考虑通信的启动时间$t_{startup}$，后面以习题形式给出考虑$t_{startup}$的情况。

如果我们认为把数列分为两部分只需要很少的计算，则分割阶段实际上只包括通信。合并阶段包括计算和通信来完成对所接收到的部分和求和以及传送结果。

（1）通信 分割阶段需要对数步数，即p个进程需要log$p$步。这一阶段的通信时间由下式给出:

$$t_{comm1} = \frac{n}{2}t_{data} + \frac{n}{4}t_{data} + \frac{n}{8}t_{data} + \cdots + \frac{n}{p}t_{data} = \frac{n(p-1)}{p}t_{data}$$

其中t_{data}为传送一个数据字所需的时间。时间t_{comm1}要比一个简单的广播稍好些。合并阶段类似，只是每个消息（部分和）中仅有一个数据项要传送，即:

$$t_{comm2} = (\log p)\, t_{data}$$

总的通信时间为:

$$t_{comm} = t_{comm1} + t_{comm2} = \frac{n(p-1)}{p}t_{data} + (\log p)t_{data}$$

当p为常数时，时间复杂性为O (n)。

（2）*计算*　分割的最后需要把n/p个数相加。合并阶段每步执行一次加法，得到:

$$t_{comp} = \frac{n}{p} + \log p$$

115

同样当p为常数时，时间复杂性为O (n)。对于大的n和可变的p，时间复杂性为O (n/p)。

（3）*总的执行时间*　总的并行执行时间为:

$$t_p = \left(\frac{n(p-1)}{p} + \log p\right)t_{data} + \frac{n}{p} + \log p$$

（4）*加速系数*　加速系数为:

$$\text{加速系数} = \frac{t_s}{t_p} = \frac{n-1}{((n/p)(p-1) + \log p)t_{data} + n/p + \log p}$$

当全部p个处理器都在计算它们的部分和时，用这种方法我们所期望的最好的加速比当然是p，但由于有分割和合并阶段，实际的加速比比p小。

（5）*计算/通信比*　计算/通信比由下式给出:

$$\text{计算/通信比} = \frac{t_{comp}}{t_{comm}} = \frac{n/p + \log p}{((n/p)(p-1) + \log p)t_{data}}$$

4.1.3 *M*路分治

分治策略也可以应用于将一个任务在每一步分为多个（大于两个）部分的情况。例如，如果一个任务每次分为四个部分，顺序递归定义应为:

```
int add(int *s)                          /* add list of numbers, s */
{
  if (number(s) =< 4) return(n1 + n2 + n3 + n4);
  else {
    Divide (s,s1,s2,s3,s4);              /* divide s into s1,s2,s3,s4*/
    part_sum1 = add(s1);                 /*recursive calls to add sublists */
    part_sum2 = add(s2);
    part_sum3 = add(s3);
    part_sum4 = add(s4);
    return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
  }
}
```

图4-6显示了一棵每个结点有四个孩子的树，称为四叉树。四叉树在分解二维区域为四个子区域中有特殊的应用。例如，一幅数字化的图像可以分为四个象限，每个象限再进一步分为四个子象限，以此类推，如图4-7所示。八叉树是每个结点有八个孩子的树，用于解决递归

地分割三维空间的问题。如果每次分割产生m部分就可以形成m叉树（即每个结点有m个孩子），随着m的增大，并行度也增大，因为有更多的部分可以同时考虑。这样的问题将留作习题，让读者自己推导计算时间和通信时间的公式（习题4-7）。 116

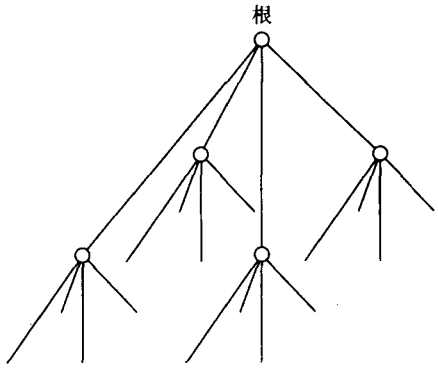


图4-6 四叉树

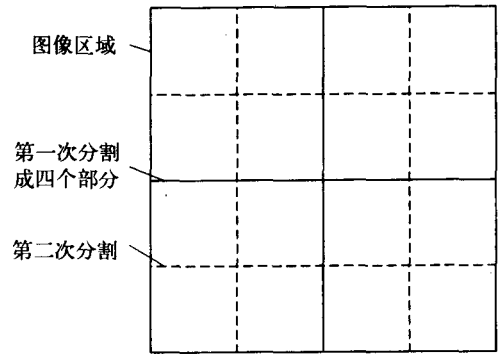


图4-7 分割一幅图像

4.2 分治技术举例

4.2.1 使用桶排序法排序

现在假设问题不是简单地对数列求和，而是将它们排序。有很多情况需要对数据排序，而在顺序程序设计的课程中花费很多时间在开发数排序的方法上。顺序排序算法大多数都是基于数对的比较和交换，我们将在第10章中讨论如何将这些经典顺序算法并行化。现在我们先讨论一种称为桶排序的算法。桶排序不是基于比较和交换的，它实质上是一种划分方法。如果原始的数是在一个已知的间隔内均匀分布时，桶排序才有良好的效果，这个间隔我们设它为0到$a-1$。把这个间隔平均分为m个区域，即0到$a/m-1$、a/m到$2a/m-1$、$2a/m$到$3a/m-1$，… 且分配一个“桶”来保存落在为每个区域内的数，这样共有m个桶，这些数仅是简单地放在对应的桶中。当然可以为每个数分配一个桶（即$m = n$），另外也可以使用分治的方法不断地将桶分割为更小的桶。如果此过程以这种方式持续到每个桶中只有一个数，那么该方法就很类似于快速排序，不同之处是快速排序将区域分割成由“枢轴（或支点）”（pivots）定义的区域（参见第10章）。这里，我们将使用有限数目的桶。每个桶中的数将使用一种顺序排序算法，如图4-8所示。 117

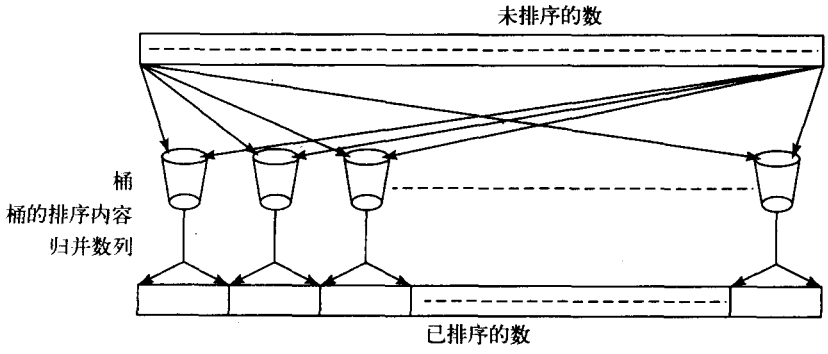


图4-8 桶排序

1. 顺序算法

要将一个数放入指定的桶中需要识别这个数所属的区域。实现这一点的一个方法是将这个数与区域头做比较，即a/m、$2a/m$、$3a/m$、…在顺序计算机中这需要对每个数进行$m-1$次比较。更有效的方法是使用m/a来除这个数，并用得到的结果识别0到$m-1$个桶，这样每个数只需要一个计算步（尽管做除法需要较多的时间）。如果m/a是2的乘方，则只需要简单察看这个数以二进制表示时的高位几位。例如若$m/a = 2^3 (= 8)$，而一个数的二进制表示为1100101，考虑最高3位，它应该落入区域110(6)。在任何情况下，让我们假设把一个数放入某个桶中只需要一步，因此放置n个数需要n步。如果数是均匀分布的，则每个桶中大致应该有n/m个数。

接下来必须对桶中的数排序，顺序排序算法如快速排序或归并排序对n个数排序的时间复杂性为O ($n\log n$)（快速排序的平均时间复杂性）。任何比较和交换排序算法的下界大致就是$n\log n$次比较[Aho，Hopcroft，and Ullman，1974]。让我们假设顺序排序算法的确需要$n\log n$次比较，一次比较被视作一个计算步。因此，使用这些顺序排序算法对每个桶中n/m个数进行排序需要 (n/m) log (n/m) 步。排好顺序的数需要拼接为一个最终的已排序数列，我们假设这个拼接过程不需要附加的步。综合以上过程，顺序时间为：

118

$$t_s = n + m((n/m)\log(n/m)) = n + n\log(n/m) = O(n\log(n/m))$$

如果$n = km$，k是一个常数，时间复杂性为O (n)。注意，这比顺序的比较和交换排序算法的下界要好很多，但是要求数是均匀分布的。

2. 并行算法

显然，为每个桶分配一个处理器就可以并行化桶排序算法，这将使上面公式中第二项在有p个处理器时缩减为 (n/p) log (n/p) (其中$p = m$)。这种实现方法如图4-9所示。在这个实现中，每个处理器要检查每一个数，因此会耗费很多操作。如果在向桶中放数据时同时从数列将其删除（即数据真的被取走），则这些数就不会被别的处理器考虑，从而可改进算法的实现。

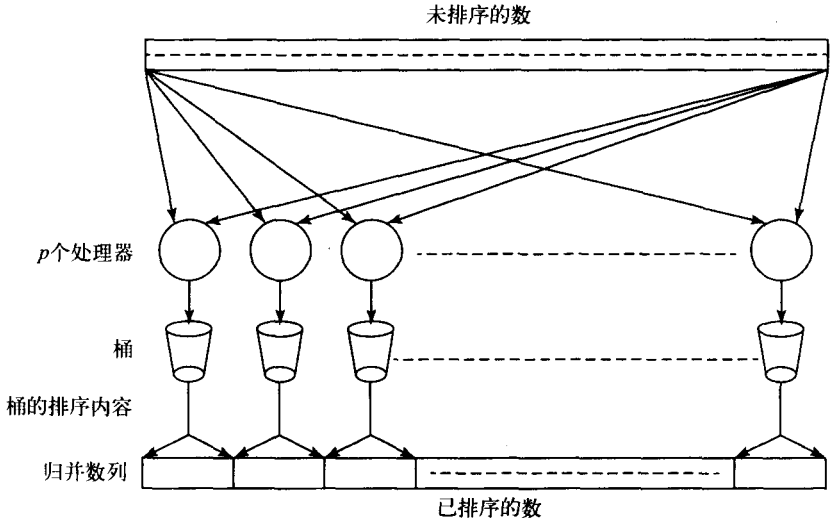


图4-9　一种桶排序的并行算法

如果将数列划分为m个区域，并为每个区域分配一个处理器，就可以进一步并行化这个算法。每个处理器都有p个“小”桶，并将自己区域中的数分离到这些小桶中。最后这些小桶的数据“倒”入p个最终的桶完成排序，这需要每个处理器向其他所有处理器各发送一个小桶（桶i对应处理器i）。总体的算法如图4-10所示，注意这种方法是一个简单的划分方法，在这种

方法中完成划分只需要最少的工作量。

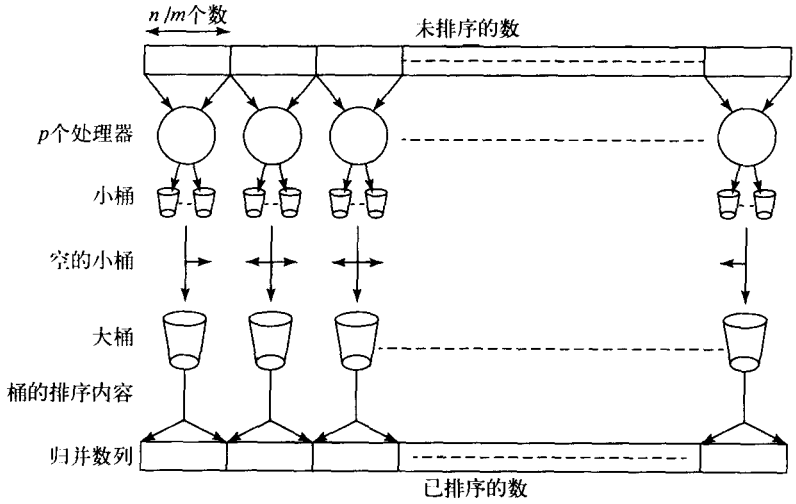


图4-10 并行桶排序

该算法需要下列阶段:

(1) 划分数据

(2) 排序到小桶中

(3) 发送到大桶中

(4) 对大桶中的数据排序

119

阶段1: 计算和通信 第1步向每个处理器发送数组。标记一个数组到划分中可在固定时间内完成。在总的计算时间中该时间将被忽略。比起先进行划分然后向每个处理器发送一个划分的方法，简单地向每个处理器广播所有数并由各个处理器进行它自己的划分是一个更为有效方案。(采用这种方法时必须保证所创建的每个划分是分离的，但划分的合成应包含所有的数。)当使用一个广播或分散例程时，则包含启动时间在内的通信时间为:

$$t_{comm1} = t_{startup} + nt_{data}$$

阶段2: 计算 将含有n/p个数的每个划分分离到p个小桶需要的时间:

$$t_{comp2} = n/p$$

阶段3: 通信 接下来对小桶进行分配。(阶段3没有计算。)每个小桶中有n/p^2个数(假设均匀分布)。每个进程必须把$p-1$个小桶中的内容发送到其他所有进程(一个为自己的大桶而保留的桶)。由于p个进程中的每个进程都要向别的进程发送，如果通信在时间上不能重叠，且必须单个地调用`send()`，则通信时间为:

$$t_{comm3} = p(p-1)(t_{startup} + (n/p^2)\ t_{data})$$

上面的公式是这个通信阶段所用时间的上界。如果所有通信都可以重叠，则得到通信时间的下界为:

$$t_{comm3} = (p-1)(t_{startup} + (n/p^2)\, t_{data})$$

120

实际上，每个处理器都会和其他所有处理器通信，因此考虑一个“全部到全部”的机制比较适合。“全部到全部”例程会将每个进程的数据发送到其他所有进程，如图4-11所示。这种例程在MPI中是可用的(`MPI_Alltoall()`)，这种实现比单个使用`send()`和`recv()`更为有效。“全部到全部”例程实际上会将数组的行传送到列中，如图4-12所示(因此实际上完成了矩阵的转置，参见10.3.1节)。

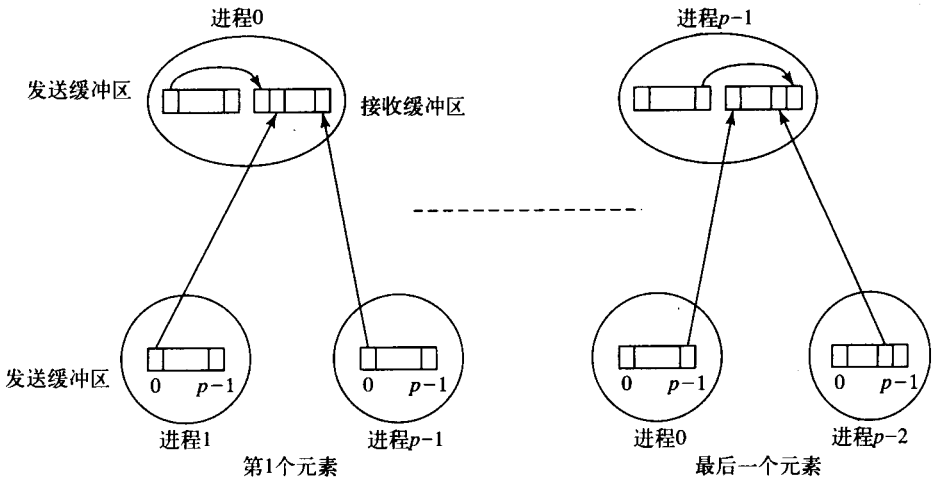


图4-11　“全部到全部”广播

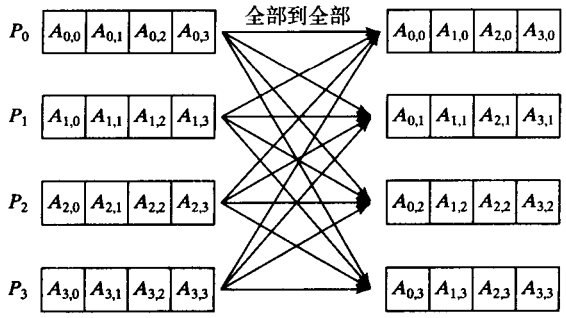


图4-12　对数组执行“全部到全部”操作的效果

阶段4：计算　在最后一个阶段中，同时对大桶排序。因为每个大桶中有n/p个数据，所以

$$t_{comp4} = (n/p)\log(n/p)$$

总的执行时间　包含通信在内的总的运行时间为：

$$t_p = t_{comm1} + t_{comp2} + t_{comm3} + t_{comp4}$$

$$t_p = t_{startup} + nt_{data} + n/p + (p-1)\,(t_{startup} + (n/p^2)\ t_{data}) + (n/p)\log(n/p)$$

$$= (n/p)\,(1 + \log(n/p)) + pt_{startup} + (n + (p-1)\,(n/p^2))\,t_{data}$$

加速系数　当与顺序桶排序比较时，加速系数为：

$$\text{加速系数} = \frac{t_s}{t_p} = \frac{n + n\log(n/m)}{(n/p)(1+\log(n/p)) + pt_{startup} + (n+(p-1)(n/p^2))t_{data}}$$

实际的加速系数公式由上式定义，其中t_s是最好的顺序算法的求解时间。使用比较和交换操作以及不要求序列分配或专门特征的顺序排序算法的下限是$n\log n$步。可是桶排序有更好的下限并用作t_s，但需假设数是均匀分布的。

121

计算/通信比　计算/通信比由下式给出：

$$\text{计算/通信比} = \frac{t_{comp}}{t_{comm}} = \frac{(n/p)(1+\log(n/p))}{pt_{startup} + (n+(p-1))(n/p^2)t_{data}}$$

获得以上公式的前提是假设数是均匀分布的。如果要排序的数不是均匀分布的，则有一些桶中的数要比另一些桶中的多，对数多的桶中的数排序将决定总的计算时间，最坏的情况

是所有数都在一个桶中。

4.2.2 数值积分

前面我们讨论了分割一个问题并求解每个子问题，问题被假设为分割成相等大小的部分，并使用了简单的划分技术。有时候简单的划分不能得到最优解，特别是当每个部分的工作量难以估计的时候。例如桶排序仅在每个区域中数的个数大致相同时才有较好的效果。（桶排序可以被修改用于等化工作。）

通用的分治技术将区域连续地分割为更小的部分，并使用最优化函数来决定特定区域何时被充分地分割。让我们来看一个不同的例子，数值积分:

$$I = \int_a^b f(x)dx$$

要对这个函数积分（如计算曲线下区域的面积），我们可将该区域分割为独立的部分，每个部分由一个独立的进程来计算。每个区域的面积可以用一个矩形来估算，如图4-13所示，其中$f(p)$和$f(q)$是矩形两边的高度，δ是宽度（间隔）。整个积分可以由从a到b的矩形区域的和来估算。更好的估计是将矩形对齐，使得矩形上边的中点落在函数的曲线上，如图4-14所示。这样构造的好处在于中点两边的误差趋向抵消。另外一种更明显的构造是利用垂直线与函数的交点来创建梯形区域，如图4-15所示。现在每个区域的计算为$1/2\,(f(p)+f(q))\delta$。这种使用值的线性组合求定积分的数值估算方法称为*求积法*（quadrature method）。

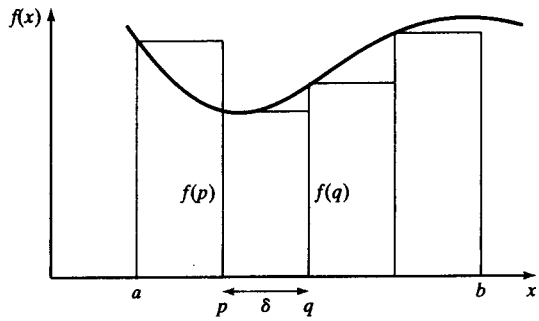


图4-13 用矩形进行数值积分

122

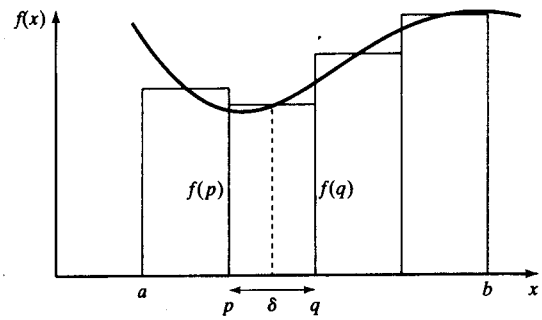


图4-14 用更精确的矩形进行数值积分

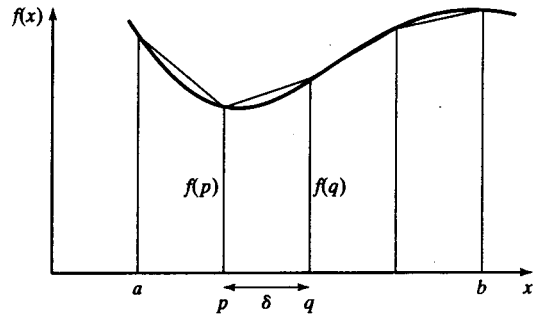


图4-15 用梯形进行数值积分

1. 静态分配

让我们考虑*梯形法*（trapezoidal method）。在开始计算之前，为计算每个区域静态分配一个进程。通过使间隔越来越小，我们会逐步接近实际的求解值。

因为每个部分的计算形式都是相同的，故使用SPMD（单程序多数据）模型比较合适。假设我们使用编号为0到$p-1$的p个进程对从$x = a$到$x = b$的区域面积求和，每个进程计算的区域范围为$(b-a)/p$。要按所述方式计算面积，SPMD的伪代码段为:

进程P_i

```
if (i == master) {                    /* read number of intervals required */
123   printf("Enter number of intervals ");
   scanf(%d",&n);
}

bcast(&n, P_group);                 /* broadcast interval to all processes */
region = (b - a)/p;                   /* length of region for each process */
start = a + region * i;               /* starting x coordinate for process */
end = start + region;                 /* ending x coordinate for process */
d = (b - a)/n;                                        /* size of interval */
area = 0.0;
for (x = start; x < end; x = x + d)
   area = area + 0.5 * (f(x) + f(x+d)) * d;
reduce_add(&integral, &area, P_group);                /* form sum of areas */
```

规约操作将单个进程计算所得的面积求和。为提高计算效率，计算每个区域面积的代码最好写成：

```
area = 0.0;
for (x = start; x < end; x = x + d)
   area = area + f(x) + f(x+d);
area = 0.5 * area * d;
```

在上面的代码中，我们假设变量area不超过允许的最大值（这种变化的一个不利之处）。要进一步提高效率，我们采用如下的代数处理来简化计算：

$$\text{面积} = \frac{\delta(f(a)+f(a+\delta))}{2} + \frac{\delta(f(a+\delta)+f(a+2\delta))}{2} \cdots + \frac{\delta(f(a+(n-1)\delta)+f(b))}{2}$$

$$= \delta\left(\frac{f(a)}{2} + f(a+\delta) + f(a+2\delta) + \cdots + f(a+(n-1)\delta) + \frac{f(b)}{2}\right)$$

给定n个间隔，每个间隔的宽度为δ，一种实现将对每个进程处理的区域使用这一公式：

```
area = 0.5 * (f(start) + f(end));
for (x = start + d; x < end; x = x + d)
   area = area + f(x);
124
area = area * d;
```

2. 自适应积分

如果预先知道能够得出足够精确解的间隔的大小δ，则到目前为止上面所用各种的方法效果都不错。我们还假设了在整个区域中间隔大小是固定的。如果合适的间隔是未知的，就需要用某种迭代求收敛解。例如我们可以从某个间隔开始，并逐步减小它，直到获得足够精确的近似值。这也就是说整个面积将使用不同的间隔计算，所以我们不能像上面的例子中那样简单地将整个区域分割为固定数目的子区域。

一种方法是将每个进程的间隔数不断翻倍，直到连续两次获得的近似值足够接近。树结构可以用来分割区域，树的深度受可用的进程/处理器的数目限制。在我们的例子中，树可能随区域精确度的增加以不对称的形式生长，*足够接近*（sufficiently close）将依赖应用程序和算术计算的精确度。

另外一种终止计算的判定方法是使用三个面积A、B和C，如图4-16所示。当算得的A或B中最大的那个区域的面积足够接近其他两个区域的面积之和时计算终止。例如，如果B的面积是最大的，当B的面积足够接近A和C的面积之和时计算终止。另外我们也可以简单地在C的面

积足够小时终止计算。这样的方法称为自适应积分（adaptive quadrature），因为解随曲线的形状改变。（可以推出自适应积分的简化公式，参见[Freeman and Phillips，1992]）。

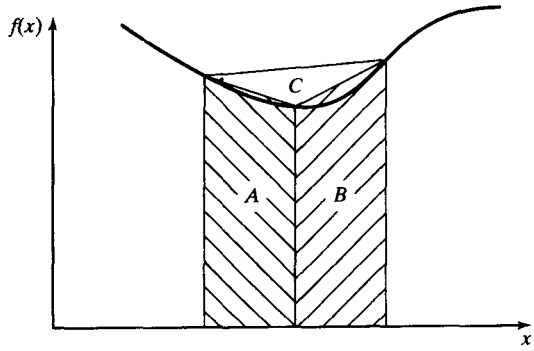


图4-16 自适应积分结构

计算变化慢的曲线下区域的面积比计算变化快的曲线下区域的面积所用时间短，间隔宽度δ随间隔变化。其结果是固定的进程任务分配将不能最有效的利用处理器。使用3.2.2节中所描述的负载平衡技术较为适合，我们将在第7章中对负载平衡做进一步讨论。要指出的是在选择何时终止时要谨慎，例如图4-17中的函数曲线可能会让我们过早终止计算，因为两个大区域的面积是相等的（如，$C = 0$）。

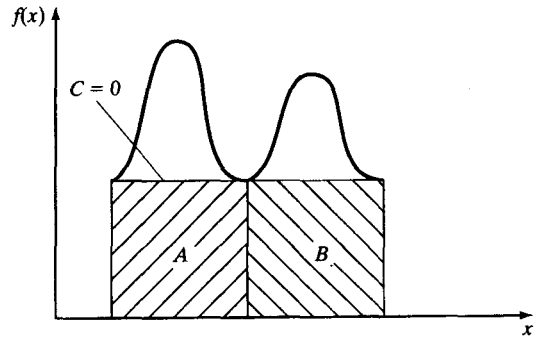


图4-17 错误终止的自适应积分结构

4.2.3 *N*体问题

另一个利用分治优势的求解问题是*N*体问题。*N*体问题是研究物体之间相互作用力产生的效果的问题（例如天体间通过引力相互吸引）。其他领域的*N*体问题包括分子动力学和流体动力学。我们将以天体系统为例来讨论这个问题，尽管这些技术也同样应用于其他应用问题。我们提供基本的等式使该应用可以作为一个编程练习来编码，对图形输出有兴趣的话可以使用与第3章中求解曼德勃罗特问题相同的图形例程。

125

1. 引力*N*体问题

这个问题的目标是确定太空中通过引力相互作用的天体（如行星）的位置和运动，可以使用的物理知识是牛顿定律。质量为m_a和m_b的两个天体相互间的引力由下式给出：

$$F = \frac{Gm_am_b}{r^2}$$

其中G是引力常数，r是天体间的距离。我们可以看到引力被描述为一个与距离r的平方成反比的定律，也就是说，两个天体间的引力与$1/r^2$成正比，r是天体间的距离。每个天体都会依据这个定律受其他天体的影响，所有力都会被累加（要考虑每个力的方向）。在受力的情况下，一个天体将根据牛顿第二定律加速：

$$F = ma$$

其中m是天体质量，F是天体所受的力，a是得到的加速度。因此，所有天体都会由于这些力的作用移动到新的位置并获得新的速度。要得到精确的数值描述，需要使用微分方程（即$F = mdv/dt$，而$v = dx/dt$）。然而，对于超过三个天体的系统的*N*体问题还没有确切的近似解。

使用计算机模拟时，我们使用特定时间的值，t_0、t_1、t_2等等，时间间隔取得尽可能短以获得最精确的解。不妨设时间间隔为Δt，则对于一个质量为m的特定天体，其所受力为：

126

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

新的速度为：

$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

其中v^{t+1}为天体在时刻$t+1$的速度，v^t为天体在时刻t的速度。如果天体以速度v移动了Δt时间，则位置的变化为:

$$x^{t+1} - x^t = v\Delta t$$

其中x^t是天体在时刻t的位置。一旦天体移动到新的位置，所受的力就会发生变化，计算过程也要重复。

在一个时间间隔Δt内的速度实际上并不是一个准确的常数，因此得到的仅是一个近似解。使用“蛙跳”式计算是很有帮助的，这种方法交替计算速度和位置，即:

$$F^t = \frac{m(v^{t+1/2} - v^{t-1/2})}{\Delta t}$$

和

$$x^{t+1} - x^t = v^{t+1/2}\Delta t$$

其中，位置在t、$t+1$、$t+2$等时刻计算，速度在$t+1/2$、$t+3/2$、$t+5/2$等时刻计算。

（1）*三维空间*　由于天体是位于三维空间，所有的值都是矢量，必须被分解到三个方向x、y和z方向。在一个坐标系统为（x，y，z）的三维空间中，位于（x_a，y_a，z_a）的天体和位于（x_b，y_b，z_b）的天体间的距离为:

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}$$

力在三个方向分解，使用以下公式:

$$F_x = \frac{Gm_a m_b}{r^2}\left(\frac{x_b - x_a}{r}\right)$$

$$F_y = \frac{Gm_a m_b}{r^2}\left(\frac{y_b - y_a}{r}\right)$$

$$F_z = \frac{Gm_a m_b}{r^2}\left(\frac{z_b - z_a}{r}\right)$$

127 其中，天体的质量分别为m_a和m_b，坐标分别为（x_a，y_a，z_a）和（x_b，y_b，z_b）。最后就可以计算新的位置和速度，速度也可以在三个方向上分解。为简单起见，我们假设三维空间有固定的边界。当然，实际上宇宙是无限伸展的，并无固定边界!

（2）*其他应用*　虽然以上我们讨论的是天体，同样的概念也可以应用到其他情形。例如，带电粒子同样相互作用，这时将遵循库仑静电定律（也是一个与距离平方成反比的定律），带异种电荷的粒子相互吸引，带同种电荷的粒子相互排斥。这个问题与天体问题微妙的不同之处在于带电粒子会互相远离，而天体只会相互靠近形成簇。

2. 顺序代码

总的引力N体计算可以由以下算法描述:

```
for (t = 0; t < tmax; t++) {              /* for each time period */
  for (i = 0; i < N; i++) {               /* for each body */
    F = Force_routine(i);                 /* compute force on ith body */
    v[i]new = v[i] + F * dt / m;          /* compute new velocity and
    x[i]new = x[i] + v[i]new * dt;        /* new position (leap-frog) */
  }
  for (i = 0; i < N; i++) {               /* for each body */
    x[i] = x[i]new;                       /* update velocity and position*/
```

```
        v[i] = v[i]new;
    }
}
```

3. 并行代码

对顺序算法代码并行化可以使用简单的划分，每个处理器对应物体组，每个力由处理器间的独立消息“承载”，但这样会产生大量消息。这是一个$O(N^2)$的算法（对于一次迭代），N个物体每个都受其他$N-1$个物体的影响。对于N比较大的大多数有趣的N体问题使用这个直接的算法是不可行的。

通过简单观察，一簇远距离物体可以大略地作为一个簇的总质量位于该簇物体重心的单个远距离物体，如图4-18所示，而且这种簇化思想可以递归使用。

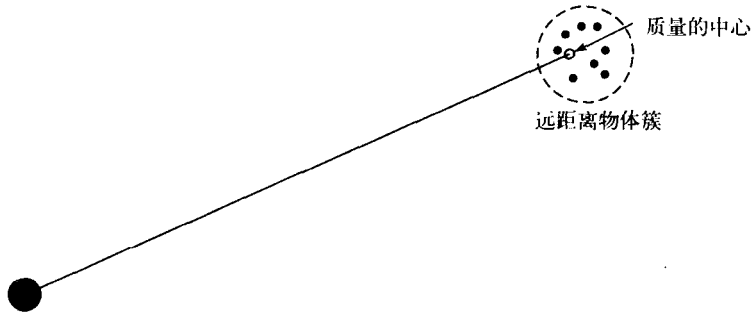


图4-18 簇化远距离的物体

Barnes-Hut算法 一种应用簇思想的更加聪明的分治模式是将整个空间看作一个包含物体（或粒子）的立方体。首先，我们将这个立方体分割为八个子立方体，如果某个子立方体不包含物体，则在下一步考虑前将它删除，如果某个子立方体包含超过一个的物体，则将它递归分割为多个只含一个物体的子立方体。这个进程将创建一棵八叉树，即每个结点都有八条边的树，树的叶子表示只含一个物体的单元。（我们假设原始空间是一个立方体，因此可以实现每个层次上的递归，当然也可以使用别的假设。）

对于二维问题，每次递归子分割将创建四个子区域和一棵四叉树（每个结点有四条边，参见4.1.3节）。一般而言，产生的树将会非常不对称。图4-19显示了二维空间的分解（比较容易画）和产生的四叉树。三维空间的情况遵循同样的结构，只是每个结点将会有八条边。 128

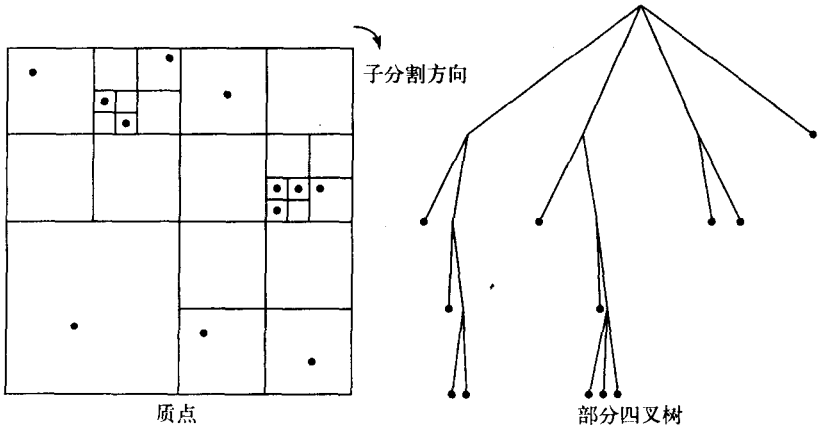


图4-19 二维空间的递归分割

在*Barnes-Hut*算法[Barnes and Hut，1986]中，当树构建好后，子立方体总的质量和重心将存储在每个结点中。每个物体所受的力可通过从根结点遍历这棵树来获得，具体算法是当到达某个结点时若簇的估计值已近似实际物体的值，则遍历结束，否则继续向下遍历树。在天体N体问题模拟中，判断何时获得近似值的简单标准如下：假设簇包含在一个体积为$d \times d \times d$的立方体中，到重心的距离是r，当下式满足时就可使用估计值：

$$r \geqslant \frac{d}{\theta}$$

129

其中θ是一个常数，通常为1.0或小于1.0（θ称为开放角）。这种方法可大大减少计算量。

一旦所有的物体都被给定新的位置和速度后，在每个时间段重复这个进程。这意味着每个时间段都要重新构建整个八叉树（因为所有的物体都已经移动了）。构建树需要的时间为O ($n$log$n$)，所有力的计算也是如此，因此这个方法的时间复杂性为O($n$log$n$)[Barnes and Hut，1986]。

算法可被描述为如下：

```
for (t = 0; t < tmax; t++) {        /* for each time period */
   Build_Octtree();                 /* construct Octtree (or Quadtree) */
   Tot_Mass_Center();               /* compute total mass & center /*
   Comp_Force();                    /* traverse tree/computing forces */
   Update();                        /* update position/velocity */
}
```

`Build_Octtree()`例程可以从物体的位置开始构建，依次考虑每个物体。每个结点在计算总质量和重心时，`Tot_Mass_Center()`例程必须遍历树。这可以递归实现。总质量M可通过对孩子的总质量简单相加来获得：

$$M = \sum_{i=0}^{7} m_i$$

其中m_i是第i个孩子的总质量。重心C可由下式获得：

$$C = \frac{1}{M} \sum_{i=0}^{7} (m_i \times c_i)$$

其中重心的位置有三个分量，即x、y和z三个方向。`Comp_Force()`例程必须访问结点以确定是否簇估计值可以用来计算那一单元中所有物体受的力。如果簇估计值不能使用，则必须访问该结点的孩子结点。

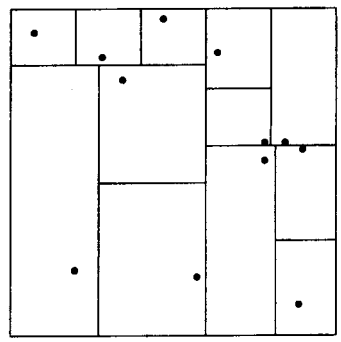


图4-20　正交递归二分法

一般情况下八叉树是很不对称的，而且在模拟的过程中其形状会不断变化。因此，简单的静态划分策略在负载平衡方面效果不好。一种将物体分为组的更好的方法称为*正交递归二分法*[Salmon，1990]。我们以二维正方形区域来描述这个方法。首先找到一条将区域分为两个区域的竖直线，每个区域包含一样多的物体，然后在每个子区域中用一条水平线将该区域分为包含同样多物体的两个区域。重复以上过程，直到区域的数目和处理器的数目相等，每个区域分配给一个处理器。图4-20是一个这种分割的例子。

130

4.3　小结

这一章介绍了以下概念：

- 作为并行计算技术基础的划分和分治概念
- 树结构
- 划分和分治问题的例子，即桶排序、数值积分和*N*体问题

推荐读物

分治技术在许多数据结构和算法教科书中均有叙述（例如，[Cormen Leiserson and Rivest, 1990]），正如我们所见，这种方法将产生一棵树结构。如果按树形网络结构构建多处理机，则很适合分治问题。在大多数应用都可以使用分治技术的思想下，已经建立了一两台树状网结构的机器。然而，正如我们在第1章中所讨论的那样，树可以嵌入网格或超立方体，因此没有必要使用树状网。将分治算法映射到不同体系结构是研究论文的主题，如[Lo and Rajopadhye, 1990]。

一旦问题划分后，某些情况下使用调度算法为处理器分配划分块或进程是很有必要的。教科书[Sarkar, 1989]专门研究划分和调度，该教科书未讨论映射（静态调度）。但我们将在第7章中讨论动态负载平衡的问题，即在程序执行过程中决定如何为处理器分配任务。

桶排序在许多教科书的排序算法中作了描述（参见第9章），在[Lindstrom, 1985]和[Wagner and Han, 1986]中有更加详细的叙述。并行程序设计中的数值积分评述可以在[Freeman and Phillips, 1992]、[Gropp, Lusk, and Skjellum, 1994]和[Smith, 1993]中找到并且经常作为并行程序设计中的一个简单应用。Barnes-Hut算法的源出处是[Barnes and Hut, 1986]。其他的论文包括[Bhatt et al., 1992]和[Warren and Salmon, 1992]。[Liu and Wu, 1997]考虑用C++对该算法编程。除了Barnes-Hut分治算法，另一种方法是快速多极算法[Greengard and Rokhlin, 1987]。此外还有一些混合方法。

参考文献

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.

BARNES, J. E., AND P. HUT (1986), "A Hierarchical O(*N*log*N*) Force Calculation Algorithm," *Nature*, Vol. 324, No. 4 (December), pp. 446–449.

BHATT, S., M. CHEN, C. Y. LIN, AND P. LIU (1992), "Abstractions for Parallel *N*-Body Simulations," *Proc. Scalable High Performance Computing Conference*, pp. 26–29.

BLELLOCH, G. E. (1996), "Programming Parallel Algorithms," *Comm. ACM*, Vol. 39, No. 3, pp. 85–97.

BOKHARI, S. H. (1981), "On the Mapping Problem," *IEEE Trans. Comput.*, Vol. C-30, No. 3, pp. 207–214.

CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST (1990), *Introduction to Algorithms*, MIT Press, Cambridge, MA.

FREEMAN, T. L., AND C. PHILLIPS (1992), *Parallel Numerical Algorithms*, Prentice Hall, London.

GREENGARD, L., AND V. ROKHLIN (1987), "A Fast Algorithm for Particle Simulations," *J. Comp. Phys.*, Vol. 73, pp. 325–348.

GROPP, W., E. LUSK, AND A. SKJELLUM (1999), *Using MPI Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA.

JÁJÁ, J. (1992), *An Introduction to Parallel Algorithms*, Addison Wesley, Reading, MA.

LINDSTROM, E. E. (1985), "The Design and Analysis of BucketSort for Bubble Memory Secondary Storage," *IEEE Trans. Comput.*, Vol. C-34, No. 3, pp. 218–233.

LIU, P., AND J.-J. WU (1997), "A Framework for Parallel Tree-Based Scientific Simulations," *Proc.*

1997 Int. Conf. Par. Proc., pp. 137–144.

LO, V. M., AND S. RAJOPADHYE (1990), "Mapping Divide-and-Conquer Algorithms to Parallel Architectures," *Proc. 1990 Int. Conf. Par. Proc.*, Part III, pp. 128–135.

MILLER, R., AND Q. F. STOUT (1996), *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*, MIT Press, Cambridge, MA.

PREPARATA, F. P., AND M. I. SHAMOS (1985), *Computational Geometry: An Introduction*, Springer-Verlag, NY.

SALMON, J. K. (1990), *Parallel Hierarchical N-Body Methods*, Ph.D. thesis, California Institute of Technology.

SARKAR, V. (1989), *Partitioning and Scheduling Parallel Programs for Multiprocessing*, MIT Press, Cambridge, MA.

SMITH, J. R. (1993), *The Design and Analysis of Parallel Algorithms*, Oxford University Press, Oxford, England.

WAGNER, R. A., AND Y. HAN (1986), "Parallel Algorithms for Bucket Sorting and Data Dependent Prefix Problem," *Proc. 1986 Int. Conf. Par. Proc.*, pp. 924–929.

WARREN, M., AND J. SALMON (1992), "Astrophysical *N*-Body Simulations Using Hierarchical Tree Data Structures," *Proc. Supercomputing 92*, IEEE CS Press, Los Alamitos, pp. 570–576.

习题

科学/数值习题

4-1 编写一个程序证明使用如4.1.1节所描述的简单划分对一组数求和可获得的最大加速比是$p/2$，其中p是进程的个数。

4-2 使用4.1.1节中推导的用于描述将一组数划分成m个划分后分别求和的方程式，证明当 $m=\sqrt{p/(1+t_{\text{startup}})}$ 时并行执行时间最小，其中m是划分的个数，p是处理器的个数。（提示：对并行执行时间的方程求导。）

4-3 4.1.1节中对一组数的求和给出了三种实现方法：使用独立的`send()`和`recv()`，使用广播例程和独立的`recv()`返回部分结果，以及使用分散和规约例程。对三种方法分别编写并行程序，并使用附加指令提取时间信息（参见2.3.4节），比较所得结果。

4-4 假设一个计算的结构包含一棵有n个叶子（最终子任务）的二叉树，树的深度为$\log n$，树中的每个结点包含一个计算步。如果处理器的数目小于n，则执行时间的下限是什么？

4-5 依据通信、计算、总的并行执行时间、加速比和效率，分析为一棵用于对一组数求和的树的每个结点分配一个处理器的分治方法。

4-6 完成4.1.2节中使用分治（二分）方法的所有八个进程的并行伪代码。

4-7 推导m路分治的通信和计算时间的等式，沿用4.1.2节中所使用的方法。

4-8 设计一种分治算法，要求使用$n/2$个处理器在O ($\log n$) 步内，在一组n个数中找出最小的数。如果处理器的数目少于$n/2$，则时间复杂性为多少？

4-9 编写一个并行程序以O($\log n$) 的时间复杂性计算任意n阶多项式

$$f = a_0x^0 + a_1x^1 + a_2x^2 + \cdots + a_{n-1}x^{n-1}$$

其中系数$a_i (i = 0 \cdots n)$、x和n是输入。

4-10 编写一个使用分治方法的并行程序从一个存放在数组中整数数列中找出第一个0。使用16个进程和256个数。

4-11 按照下列方法编写计算n个整数的和的并行程序，并评价它们的性能。假设n是2的乘方。

(a) 将n个整数划分为$n/2$对。使用$n/2$个进程一起对每对数求和以获得$n/2$个整数。重复以上过程对$n/2$个数求和获得$n/4$个整数，直到求出最后的解。(这是一个二叉树算法。)

(b) 将n个整数划分为$n/\log n$组，每组$\log n$个数。使用$n/\log n$个进程，每个进程对每组数顺序相加。最后用(a)中的方法对$n/\log n$个结果求和。算法如图4-21所示。

131 ~ 133

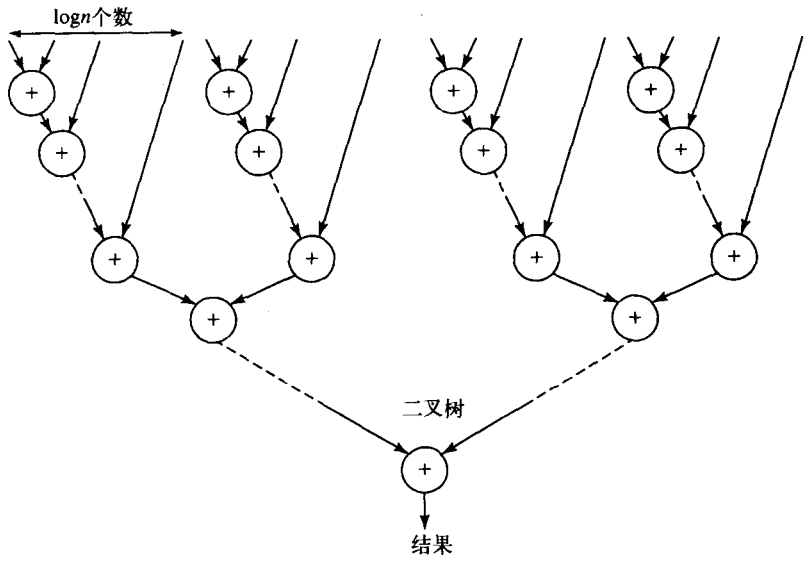


图4-21 习题4-11(b)的进程图

4-12 按照下列方法编写并行程序计算$n!$，并评价它们的性能。n可以是奇数或是偶数，且是一个正常数。

(a) 使用两个并发进程计算$n!$，每个进程约计算完整序列的一半。然后由主进程合并两个部分解。

(b) 使用连接在一起的一个生产者进程和一个消费者进程，生产者进程按顺序生成数1、2、3、…、n，消费者进程从生产者进程接受数据并累积结果，即$1 \times 2 \times 3 \cdots$。

4-13 编写一个分治并行程序用于判断一个二进制文件中1的个数是偶数还是奇数(即创建一个奇偶检查器)。修改程序使得可以向文件内容添加一个位，通过设置它为0或1使得1的个数为偶数(奇偶生成器)。

4-14 如果数不是均匀分布时，桶排序和它的并行实现性能就较差，因为在后续排序时会有较多的数落入同一个桶中。修改并实现该算法以使每个桶所收集的区域是可变的。这可在算法执行时完成或是在算法执行前的预处理步中完成，实现你的算法。

4-15 一种计算π的方法是对曲线$f(x) = 4/(1 + x^2)$下在0到1的区间内的面积进行计算，它在数值上等于π。使用10个进程按这种方法编写一个并行程序来计算π。另一种计算π的方法是计算半径为$r = 1$的圆的面积(即$\pi r^2 = \pi$)，确定计算圆面积的近似方程，并用这种方法编写并行程序计算π。评价这两种计算π的方法。

134

4-16 推导一个公式以评估使用4.2.2节中所描述的自适应积分法的积分。使用梯形积分的推导方法。

4-17　使用任意方法编写并行程序以计算积分

$$I=\int_{0.01}^{1}\left(x+\sin\left(\frac{1}{x}\right)\right)dx$$

4-18　编写一个静态分配的并行程序来计算π，使用如下公式下:

$$\int_0^1\sqrt{1-x^2}\,dx=\frac{\pi}{4}$$

使用下列方法:

1）矩形分解，如图4-13所示。

2）矩形分解，如图4-14所示。

3）梯形分解，如图4-15所示。

用速度和精度评估每一种方法。

4-19　使用二分法寻找函数的零点。在这种方法中，计算函数的两个点，分别为$f(a)$和$f(b)$，其中$f(a)>0$而$f(b)<0$，则函数在a和b之间必然存在一个零点，如图4-22所示。连续地分割区间就可以找到零点的确切位置。编写一个分治的程序寻找函数$f(x)=x^2-3x+2$的零点（该函数有两个零点，即$x=1$和$x=2$）。

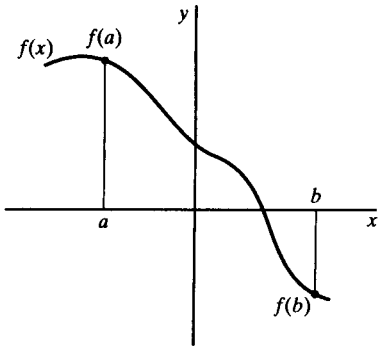


图4-22　用二分法寻找零点

4-20　编写使用辛普生(Simpson)法则计算一个函数积分的并行程序，积分的形式为:

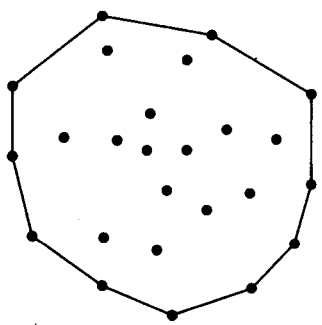
$$I=\int_a^b f(x)dx=\frac{\delta}{3}[f(a)+4f(a+\delta)+2f(a+2\delta)+4f(a+3\delta)+2f(a+4\delta)+\cdots 4f(a+(n-1)\delta)+f(b)]$$

其中δ是固定的[$\delta=(b-a)/n$，且n必须是偶数]。选择一个合适的函数（或对它进行适当安排以使函数可以输入）。

4-21　编写一个顺序程序和一个并行程序来模拟天体的N体系统，但仅限于二维平面。所有天
135 体初始均为静止状态，它们的初始位置和质量被随机选择（使用随机数生成器）。使用求解曼德勃罗特问题时所使用的图形例程来显示天体的运动。这些例程可以在http://www.cs.uncc.edu/par_prog或其他地方找到，为每个天体指定一种颜色并用大小来指明质量。

4-22　根据库仑定律为带电粒子（自由电子和正电荷）建立一个N体问题的方程。编写一个顺序程序和一个并行程序模拟这一系统，假设粒子处于二维空间。用图形输出粒子的运动。自行假设粒子的初始分布和运动状态以及问题求解空间。

4-23　（讨论题）假定平面上有n个点，设计一个算法和一个并行程序找出包含所有点的最小区域，并连接区域边界上的点，如图4-23所示。这个问题也叫平面凸包问题，可以通过与快速排序类似的递归的分治方法解决，用枢轴（pivot）点将区域递归地分为两部分。有很多关于对平面图凸包问题的信息，包括[Blelloch，1996]、[Preparata and Shamos，1985]以及[Miller and Stout，1996]。


图4-23　凸多边形包含问题（习题4-23）

现实生活习题

4-24 编写一个顺序程序和一个并行程序分别用于模拟太阳周围的行星（或其他天体系统）。产生一个图形输出表示这些行星的运动。可以自行提供行星的初始分布和运动状态。（尽量使用真实数据。）

4-25 假设你所在州的一家大银行平均每天处理200万客户账户的3000万张支票。其中最费时的一个问题是将这些支票按客户账目排序，以便将其录入客户的每月账户清单。（此外，银行还为其他客户银行处理支票排序事务。）现在银行使用大型机和快速排序算法完成此项工作。然而，你告诉银行说：可以用N台较小的计算机进行并行处理，让每台机器处理全部工作（3000万张支票）的$1/N$，然后将并行排序的结果合成一个整体。在银行对这一新技术投资之前，你被聘为顾问并需用消息传递程序设计演示这一过程。根据下面假设，为银行演示这一新方法。

假设：

1）每张支票有三个识别号：9位数的银行识别号，9位数的账户识别号和3位数的支票号（不打印或不显示前导的0）。 136

2）按账户号对同一银行识别号的所有支票排序并传往该客户行。

估计$N = 10$和$N = 1000$时的加速比。估计用于通信的时间和用于计算时间的比率。

4-26 Sue，21岁，来自一个经济上精打细算的家庭，她多年看着父母省吃俭用和投资。她每天都在大学图书馆里（免费！）阅读《Wall Street Journal》并且知道在她49岁退休后不可能依靠社会保险。在她大学毕业后，父母给她一张CD，上面有从1900年1月1日到上月底证券交易的收盘价格的日记录。

为简单起见，你可假设在CD上，从1900年开始的358000个股票的数据以日期/记号/收盘价格的记录有组织地存放（每天只列出了一部分，歇业的公司和新加入的公司）。而且，你可假设记录的格式如下：

日期	年份的最后三个数字，后跟Julian(恺撒)日（1月15日是Julian 15 日，2月1日是Julian 32日等）
象征	最多10个字符，如PCAI、KAUFX或IBM.AZ，分别代表NASDAQ股票（PCA International）、共同基金（Kaufman Aggressive Growth）和在指定时间范围内以指定价格选择购买IBM股票。
收盘价格	三个整数，X（表示每股收盘价的美元整数位），Y（表示每股收盘价的美元真分数的分子数字），Z（表示收盘价的美元真分数的分母）

例如“996033/PCAI/10/3/4”表示1996年2月2日，PCAI股票的收盘价为每股$10.75。Sue想知道CD上的历史纪录中有多少种股票（列在前一个月末的清单上）清盘前至少连续50个交易日价格变动不大或有所上扬。

4-27 Samantha越想她祖父在1963年赢得多米诺骨牌的世界杯冠军,她就越想提高她在这项比赛中的技术。但她却有个大难题:她已经没有可以匹敌的对手。她已达到在每场比赛中都能战胜所剩无几的几个对手的程度!

Samantha知道有计算机上的围棋、国际象棋、桥牌、扑克和跳棋游戏，她认为没有理由那些计算机科学的高手不能将多米诺骨牌做成计算机游戏。加拿大大学新校U-Can-2的一位计算机科学的教授告诉她，在理论上她可以用计算机实现任意她想做的事（当然不超过理论的极限）。Samantha确实很想赢得下一届世界杯冠军!

拿出她那又慢又老、几乎过时的2“立方”的Itanium机（2GHz、2GB RAM、2TB硬盘），她很快编出了一个简单的运行于单处理器上的模拟程序。她的方法的基本要点是使程序对每张牌和已放好的牌进行比较，来确定计算机的最佳的移动。这个程序包含了大量计算，包括牌的旋转和牌的试验位置。Samantha发现往往是她在等程序算出下一个位子，她开始像厌烦她的对手一样厌烦她的游戏程序的性能。因此，Samantha希望得到你的帮助，编写一个并行程序。

1）略述Samantha的单处理器算法。

137

2）略述你的并行处理机算法。

3）假如你将50台和她一样的老计算机联网，估计可能得到的加速比；并向她推荐或是就用这50台机器联网进行计算，或是花$800买一台号称在进行这类模拟时能比她的老Itanium机器至少快50倍的最新的机器：14GHz双处理器Octium。该机器有一个标准的1024位前侧的数据总线和可对它的0.5ns的16TB主存储器进行2路的同时访问。

4-28 Area公司为本地区的许多小的工程公司提供数值积分服务。当这些公司在一个域中定义有一个连续函数并无法积分时，就求助于Area公司。你刚被雇用去帮助Area公司提高其提供积分计算结果的速率。Area公司每年都为此蒙受经济损失，以至连发放下星期的工资都成了问题。假设你还想保住你的饭碗，你必须帮助Area公司。

同时假设你对并行计算很有研究，并立即发现问题所在：Area一直用单处理器来实现标准的数值积分算法。

步1：将独立坐标轴分为N个等分区间。

步2：用区间宽度乘以函数值（当在区间左边求该值时）所得到的乘积来近似估计任何区间中函数下面部分的面积。

步3：将N个近似值相加，得到总面积。

步4：将区间长度减半。

步5：重复1-4步直到第i次重复的结果与第$i-1$次的结果的误差小于第i次结果的0.001%。

由于你的经理对新奇的并行算法表示怀疑，她要求你在两种不同的机器配置下分别进行演示：第一种是双处理器，第二种是8处理器。她告诉你，如果演示成功，将购买更多的处理器并预示你将得到下周的工资。

4-29 外星智能搜索研究项目（the Search for Extra-Terrestrial Intelligence project，SETI）利用数百万台计算机分析来自波多黎各Arecibo天文台射电望远镜信号。给每一台计算机分配由世界上最大的射电望远镜所记录的一段信号的有限的时频段并要求它们完成密集计算的分析，以确定该部分含有来自其他生命形式的智能通信的可能性。显然，这是一个分治方法的例子。

讨论如何在世界范围的反恐怖主义斗争中将这一方法用于更为本土的对无线电频率通信分析的问题。

4-30 Rafic喜爱求解纵横填字游戏。近来，当无法找到合适的具有挑战性的难题时他开始自己来设计它们。该过程分为两部分：规划欲填入字母的各个空白方格的布局，以及规划分离词或短语的涂黑方格的布局。他有一本单词和短语的字典，该字典含有超过10万个单词和短语涉及从古希腊和罗马的典故到相当时髦的用语如Itanium-III和微微（pico）技术。Rafic需要你的帮助：

开发一个顺序算法，它会生成大小为$N \times N$的一个纵横字谜难题，再把该顺序算法转换成对应的并行算法。

4-31 不时地，计算机科学新来的一些大学生受诱惑而去拷贝其他同学完成的作业作为自己提交的作业，这是一种公认的剽窃的行为。一旦被发现常导致学校的严厉惩罚。有时 138
更有创造力的学生在提交作业前做了一些改变：改变变量名，改变缩排，有时甚至改变循环结构（用“for”替代“while”）。通常第一年的课程有400个学生，这大约需要为每一个编程作业进行80000次程序比较。其结果是很少进行全面的检查。

（容易）开发一个使用N台计算机的并行方法，它能彻底检查出一个典型编程的400份作业的确实复制。

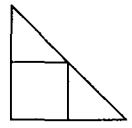
（较难）对每个程序中的变量名做施加标记的预处理，使每个程序转换成一个标准的变量名集。再使用前面的“容易”部分的方法。

（最难）对每个程序进行预处理使所有循环放入同一“for”结构。然后实现前面两个部分。

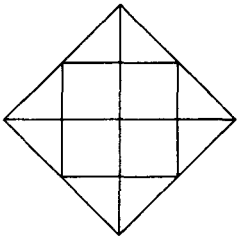
4-32 Tom的朋友Sarah将设计拼板玩具视为自己的业余爱好。在与她自己的工作间中的可控计算机磨床连在一起的她的家中的CAD系统上，她设计了三种基本的拼板形状并为每一种形状制造了几千个拼板。第1个形状是直角三角形，它的两条短边长度为1（单位）。第2个形状是边长为1的正方形。第3个形状则是一个直角三角形和与其相贴的一个矩形组合，矩形的两个边长分别为4和13，而直角三角形的短边长为4。对于给定的任意一个的拼板组合（F代表类型1，S代表类型2，以及T代表类型3），Sarah需要你开发和实现一个算法，该算法将确定最大的直角三角形的面积，它可通过将某些或所有的$F + S + T$类型的拼板放置在一起而构成。先是用顺序算法，然后用N台计算机的并行算法来求解。图4-24中示出的这些拼板的实例组合可作为你设计的出发点。



a）单三角形



b）一对三角形及一个正方形



c）四个(b)的实例，(b)实例中的小三角形面积为0.5平方单位，(b)实例的面积为2平方单位。经90°旋转后的四个(b)实例形成图示的正方形，每条边长为$2\sqrt{2}$，正方形面积为8个平方单位 139

图4-24 三角形（习题4-32）

第5章 流水线计算

本章中，我们提出一种称为流水线（pipelining）的并行处理技术，它广泛适用于本质上是部分串行的问题，也就是说这些问题必须执行一系列步骤。因此我们可以采用流水线对其中的顺序代码进行并行化。在本章中还将概述为了达到提高性能的目的而必须满足的一些要求。

5.1 流水线技术

在流水线技术中，问题被分成一系列必须是一个接一个完成的任务。实际上，这也是顺序程序设计的基础。在流水线操作中，每个任务由独立的进程或处理器所执行，如图5-1所示。我们有时把一个流水线进程称作一个流水线级（stage）。每一级只解决问题的一部分并且把相关的信息传送给需要它的下一级。这种并行性可以看作是*功能分解*（functional decomposition）的一种形式。问题被划分为必须执行的独立的功能，而且在这种情况下这些功能必须连续执行。我们将看到，输入数据通常被分解且分别处理。

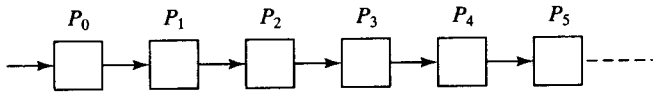


图5-1 流水化进程

作为一个能被流水线处理的顺序程序的例子，考虑下面一个简单的循环：

```
for (i = 0; i < n; i++)
  sum = sum + a[i];
```

这个循环求出数组a的所有元素之累加和。它可以按如下方式展开：

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
    ⋮
```

流水线的一种解决方法是对其中的每条语句分别设有对应的独立的级，如图5-2所示。每个级从它的输入S_{in}处接收累加和，从输入a处接受a[ ]的一个元素，而在它的输出S_{out}处产生新的累加和。因此，第i级将执行下列操作：

```
s_out = s_in + a[i];
```

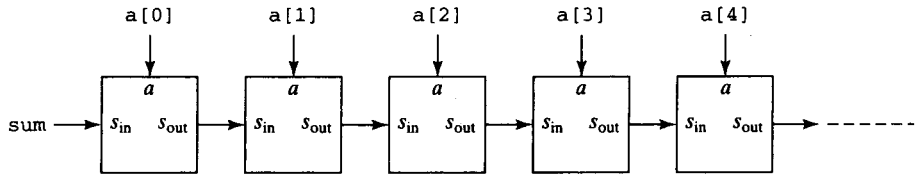


图5-2 展开循环流水线

这样，一系列的功能就取代了简单的语句而以流水线的方式进行。一个更实际的例子是

频率过滤器，在该例中问题被分成一系列的功能（功能分解）。它的目标是从（数字化）信号$f(t)$中去掉指定的频率（比如频率f_0、f_1、f_2、f_3等等）。信号从流水线的左方进入，如图5-3所示。每一（流水线）级负责去掉一个频率。

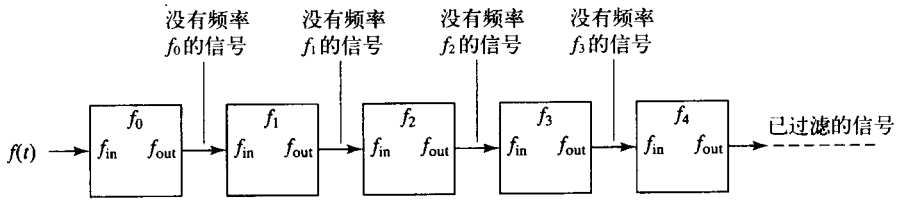


图5-3 频率过滤器的流水线

另一个类似的应用是识别信号中的特定频率。例如，家庭和专业音响系统通常在音频输出处显示特定的声音频率。每个流水线级能识别一种频率并且把它的振幅显示出来而组成频率-振幅直方图的一部分。习题5-13将讨论这个应用。 141

假定一个问题能够被分解成一系列的顺序任务，那么采用下列三种计算类型，可以使用流水线方法来取得加速：

1）如果将执行整个问题的多个实例；

2）如果必须处理一系列的数据项，而每个数据项需要多次操作；

3）如果进程在完成自己的所有内部操作之前能够把下一个进程启动所需的信息向前传送。

我们将把这三种解法视为类型1、类型2和类型3。

类型1的方式在计算机的内部硬件设计中广泛应用。它还应用于一些需要用不同参数进行多次模拟以对比实验结果的模拟实验中。类型1的流水线可以用图5-4所示的时空图来说明。在这种图中，假设每一个进程完成任务的时间是相同的。每一个时间段就是一个流水线周期。因此问题的每一个实例需要经历六个顺序进程，P_0、P_1、P_2、P_3、P_4和P_5。请注意起始时的阶梯效果。过了这个阶梯效果之后，每一个流水线周期就可以完成问题的一个实例。图5-4中的信息也可以用另外一种时空图来表示，如图5-5所示。在这种图中，实例是在纵轴上列出的，这种形式的图在必须显示一个任务实例到另一个任务实例所传递的信息时比较有用（如在处理机流水线中出现的那样）。

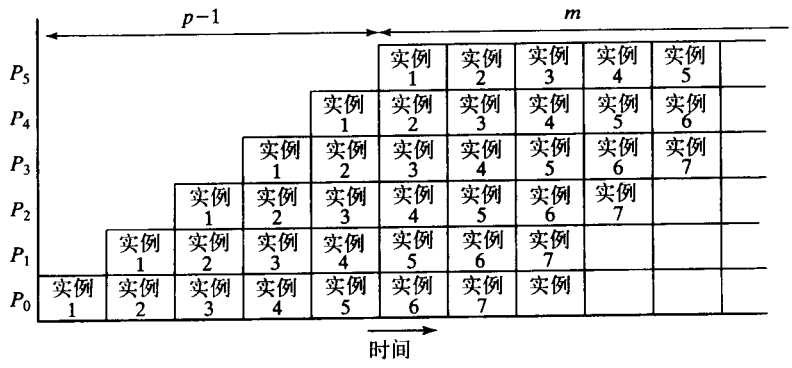


图5-4 流水线的时空图

由p个进程构成的流水线完成m个问题实例的执行需要$m+p-1$个流水线周期，平均周期数为$(m+p-1)/m$。当m很大时，每个问题实例趋近于一个流水线周期。在任何情况下，流水线过了开始的$p-1$个周期（流水线时延）后，以后的每一个周期将完成问题的一个实例。在以后

142 的分析中，对于一个p级流水线和m个问题实例的情况，我们还将使用$m+p-1$这个公式。

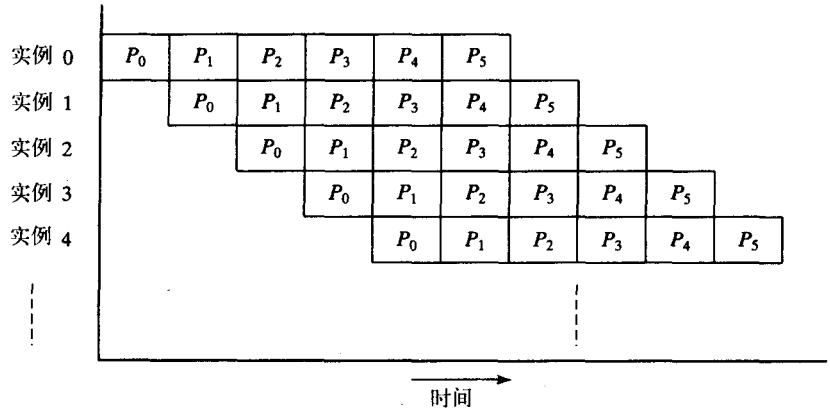
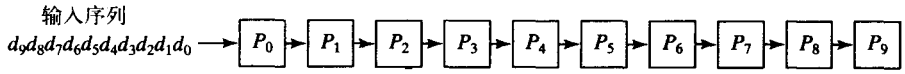
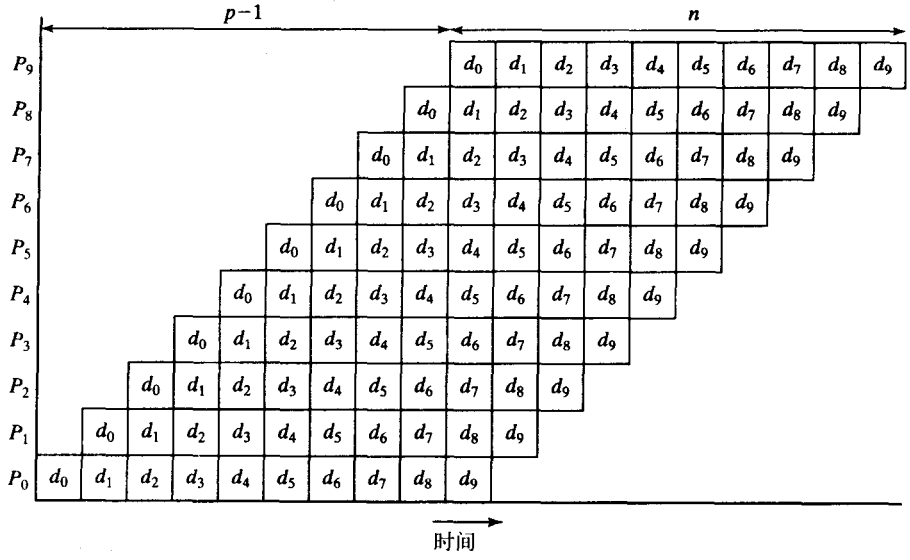


图5-5　另一种时空图

类型2的方式，即一串数据项必须被顺序处理，通常发生在算术计算中，如数组元素相乘，其中每一个元素按顺序进入流水线，如图5-6所示。其中10个进程组成流水线并且有10个元素d_0、d_1、d_2、d_3、d_4、d_5、d_6、d_7、d_8、d_9待处理。如果每级流水线周期都相等的话，则p个进程、n个数据项所需的总的执行时间是$(p-1)+n$个流水线周期。



a）流水线结构

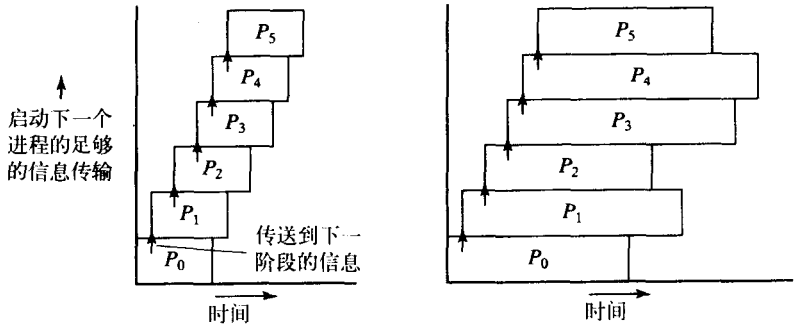


b）时序图

143

图5-6　处理10个数据元素的流水线

类型3的情况经常出现，它用于只有问题的一个实例要执行的并行程序中，但每个进程都能在它执行完毕之前把信息传递给下一个进程。图5-7显示了这种情况的时空图，其中进程在执行完毕之前，信息从该进程传递到另一个进程。



a）进程具有相同的执行时间　　b）进程具有不同的执行时间

图5-7 在进程执行结束之前传递信息给下一级的流水线

在任一流水线中，如果级比处理器的数目多，每个处理器就会分配到一组级，如图5-8所示。当然，这种情况下每个处理器中的流水级都是顺序执行的。

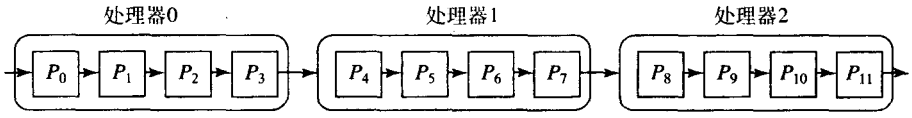


图5-8 在处理器上划分进程

5.2 流水线应用的计算平台

对流水线操作的一个关键要求是流水线的相邻进程之间要有发送消息的能力。这就意味着相邻进程所映射到的相应的处理器之间应有直接的通信链路。理想的互联结构是线形或环形结构，例如一串处理器连接到一个主机系统，如图5-9所示。当然，如第1章所述，线形和环形结构可以完美地嵌入到网格和超立方体结构之上。这就使得网格和超立方体成为合适的平台。线形结构看起来虽然并不灵活，但是事实上对于许多应用来说却很方便，而且廉价。要在联网计算机和计算机机群上有效地使用流水线，需要能够在相邻的进程或处理器之间同时传递消息的互联结构。大多数计算机机群使用交换式的互联结构允许这种传送。但一个简 144
单的共享以太网不会提供这种同时的传送。在这种情况下，使用（本地）阻塞式send()（也就是通常用的send()）会稍微方便一些，此时一个进程可不必等待接收方是否就绪就继续下一步的操作。

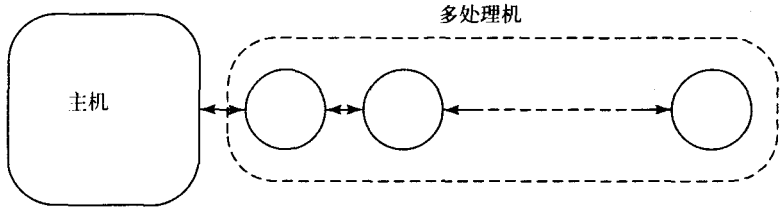


图5-9 线形结构的多处理机系统

5.3 流水线程序举例

在这一节中，我们将用实例讨论流水线处理方法，其中包括类型1、类型2或类型3。

5.3.1 数字相加

我们的第一个例子是对一个数列进行相加（这也可以用于任何数的组合运算）。当每个进程中含有一个数时（$p = n$），流水线的解决方案就是让流水线中的每个进程加一个数到累加和中，如图5-10所示。上一个进程把部分和传递给下一个进程，每个进程负责将该进程中的数加到累加和中。

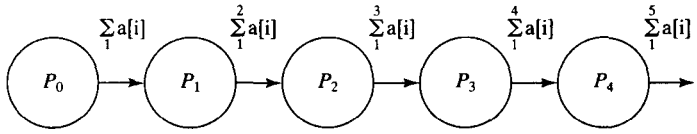


图5-10　流水线加法

除了第一个进程P_0的代码为：

```
send(&number, P1);
```

和最后一个进程P_{p-1}代码为：

```
recv(&number, Pp-2);
accumulation = accumulation + number;
```

此外，进程P_i的基本代码很简单：

```
recv(&accumulation, Pi-1);
accumulation = accumulation + number;
send(&accumulation, Pi+1);
```

145

因此，相应的SPMD程序可以有如下形式：

```
if (process > 0) {
   recv(&accumulation, Pi-1);
   accumulation = accumulation + number;
}
if (process < p-1) send(&accumulation, Pi+1);
```

在最后的进程中得到最终结果。除了加法，还可以进行其他的算术运算。例如，单个数x与输入的x相乘并将结果向前传递就可产生x的乘方。因此，一个五级的流水线就可以得到x^6。习题5-1探讨论了进行同一计算时采用这种方法与采用分治的树结构方法相比的优越性。

在我们对流水线的一般描述中，我们看到数据被输入到第一个进程中而不是在相应的进程中。如果输入数据放入第一个进程，从最后一个进程中返回结果是很自然的，如图5-11所示。如果处理器以环形结构连接时尤为合适。

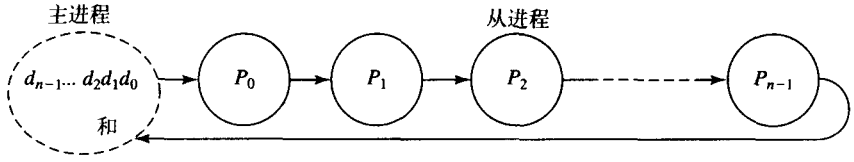


图5-11　采用主进程和环形结构的数的流水线加法

对于主-从结构，采用图5-12所示的结构较为合适。求解问题的数仅在进程需要时才输入到每一个进程中。第一个进程最先得到数。第二个进程在一个周期后得到它所需的数，以此类推。正如将在第11章看到的那样，这种形式的消息传递出现在一些数值问题中。在第11章里，还将介绍能从两端同时输入信息的流水线结构和二维流水线结构。

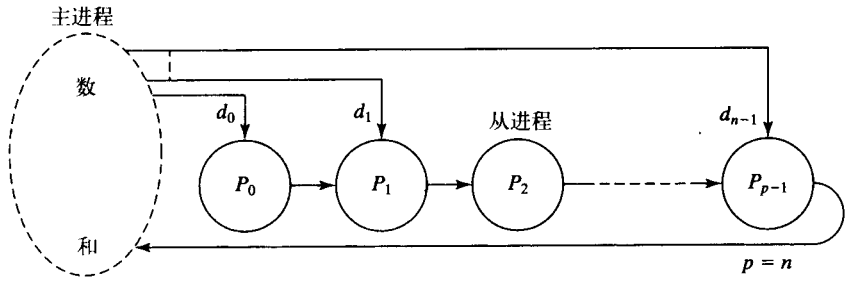


图5-12 直接访问从进程的数的流水线加法

回到数相加这个问题上，让每一个进程对应一个数是没有意义的，因为那样的话，1000个数就需要1000个从进程。通常，一个进程会把一组数相加后再将结果向前传送。很多数值应用中都使用了这样的数据划分（data partitioning）以减少通信开销，我们的所有例子也不例外。

分析

我们举的第一个流水线的例子属于类型1；它仅在我们要解决某个问题的多个实例时有效（也就是有多于一组的数要进行相加）。 146

在分析流水线时，如前几章所述，把所有的进程看作是同时具有通信和计算的阶段是不恰当的，因为问题的每个实例的起止时间都不同。我们假设每个进程在每个流水线周期执行的操作相似。这样我们就可计算出一个流水线周期所需的通信和计算的时间。从而总的执行时间t_{total}就等于每个周期所花的时间乘上周期数；即

$$t_{total} = (\text{一个流水线周期的时间}) \times (\text{周期数})$$

$$t_{total} = (t_{comp} + t_{comm})(m + p - 1)$$

式中m为问题的实例数，p为流水线级（进程数）。t_{comp}和t_{comm}分别表示计算时间和通信时间。计算的平均时间为:

$$t_a = t_{total}/m$$

假设使用图5-11所示的结构，并且有n个数。

（1）单个实例的问题 让我们首先考虑每一级只负责加一个数，也就是n等于p。每个流水线周期的长度就由一次加法和两次通信时间（一次通信从左到右，一次通信从右到左）决定，

$$t_{comp} = 1$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

因此，每个流水线周期t_{cycle}至少需要$t_{comp} + t_{comm}$，即

$$t_{cycle} = 2(t_{startup} + t_{data}) + 1$$

t_{data}表示传递一个数据字通常所需的时间，$t_{startup}$为通信启动时间。最后一个进程仅有一次通信，但因为所有的进程都被分配了同样的时间周期，所以这并不会改变结果。 147

如果我们仅仅计算一组数据（$m = 1$），则总的执行时间t_{total}就可表示为

$$t_{total} = (2(t_{startup} + t_{data}) + 1)n$$

（也就是等于n个流水线周期，因为每个进程都必须等待前一个进程完成其计算并接收前一个进程的计算结果）。时间复杂性为O (n)。

（2）多个实例问题 然而，如果我们要分别得到n个数的m组相加的结果，每组导致一个独立的结果，每个流水线周期的时间还是原来的大小，但所需的周期数却变成了$m + n - 1$个，这样

$$t_{total} = (2(t_{startup} + t_{data}) + 1)(m + n - 1)$$

当m很大时，平均执行时间t_a大约为

$$t_a = t_{total}/m \approx 2(t_{startup} + t_{data}) + 1$$

也就是一个流水线周期。

（3）*多个实例问题的数据划分*　现在让我们来考虑每一级处理d个数的一组的数据划分。进程的数目为$p = n/d$。每次通信仍然传递一次结果，但是计算需要先把d个数累加（$d-1$步），再加上输入数。所以有

$$t_{comp} = d$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

$$t_{total} = (2(t_{startup} + t_{data}) + d)(m + n/d - 1)$$

显然，当我们增大数据划分d时，通信对于整个计算时间的影响减小。但是这样同时又减小了并行性，增加了执行时间。我们把怎样寻找这个平衡点留给习题（习题5-5）。

5.3.2　数的排序

排序的目标是把一组数字以升序（或降序）的顺序重新排列（严格地说，如果有重复数字的话，应该是非递减/非递增的顺序）。流水线排序的方法是让第一个进程P_0每次接收要排序的这一组数中的一个，保存当前接收到的最大的数字且把比这个数小的其他数传递给下一个进程。如果输入的数比当前保存的数大，就把当前存储的数传递给下一个进程，把输入的数保存为当前的最大数存储。每个后继进程都执行同样的算法，即总是保存当前接收到的最大的数。当所有数都处理完毕后，P_0中就保存有最大的数，P_1中保存有次大的数，P_2中保存有第三大的数，以此类推。这个算法其实是*插入排序*（insertion sort）的并行版本。它的顺序版本类似于用插入的方法整理扑克牌（参见[Cormen, Leiserson, and Rivest,1990]（顺序插入
148 排序算法仅在排序少量数的时候有效。）

图5-13显示了排序5个数时的动作。进程P_i的基本算法如下：

```
recv(&number, P_{i-1});
if (number > x) {
   send(&x, P_{i+1});
   x = number;
} else send(&number, P_{i+1});
```

对于n个数，第i个进程要接收多少数是知道的，即$n-i$。它要上传多少个数也是知道的，即$n-i-1$，这是因为接收到的数有一个并不上传。所以，可以使用一个简单的循环：

```
right_procNum = n - i - 1;          /* number of processes to the right */
recv(&x, P_{i-1});
for (j = 0; j < right_procNum; j++) {
   recv(&number, P_{i-1});
   if (&number > x) {
      send(&x, P_{i+1});
      x = number;
   } else send(&number, P_{i+1});
}
```

图5-14解释了这个流水线。因为每一个流水线进程执行的代码相同，所以使用SPMD或主从方法的消息传递程序是特别直观的。我们从代码中看出，在一个流水线周期中一个进程在传递一个数后就没有机会继续有效地工作了（类型3）。然而，一系列操作要在一系列数据项上完成（类型2），所以即使只有一个问题实例，也可以取得显著的加速比。

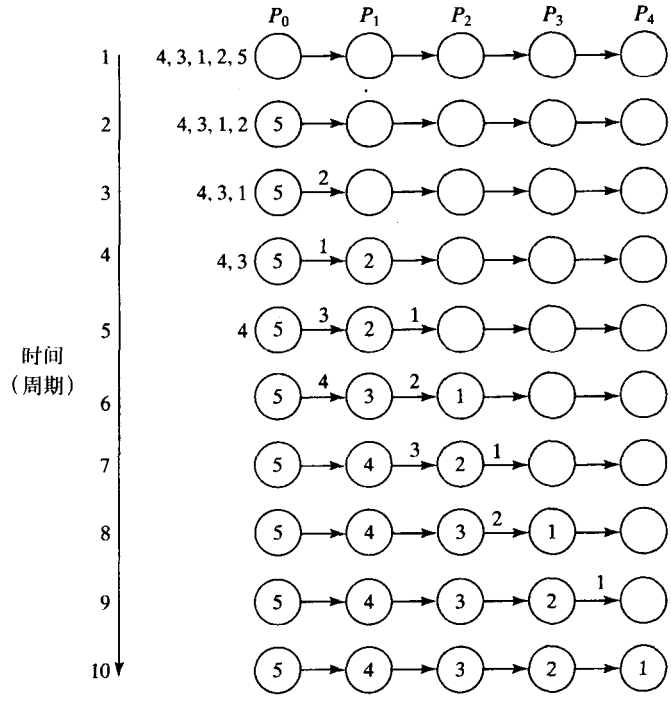


图5-13 五个数的插入排序的步骤

149

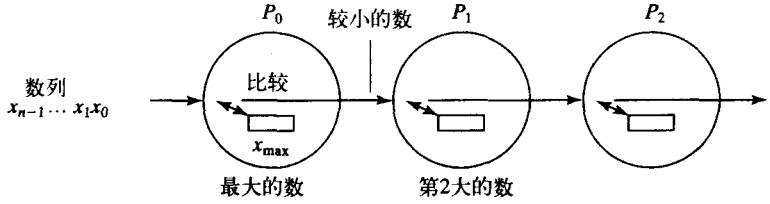


图5-14 插入排序的排序流水线

排序算法的结果可以从使用图5-11的环形结构或图5-15的双向线形结构的流水线中提取出来。后者更有优越性，因为某进程只要最后一个数通过它被传递就可以返回结果，而不必等到所有数都被排序。[Leighton，1992]描述了这种和其他三种收集结果的方式，得出的结论是这种方法是最好的。

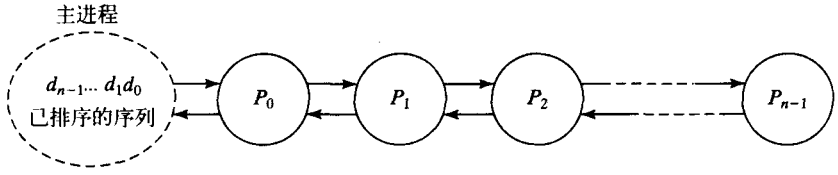


图5-15 使用双向线形结构把结果返回给主进程的插入排序

150

加上返回结果，进程i的形式就变为如下：

```
right_procNum = n - i - 1;          /* number of processes to the right */
recv(&x, P_{i-1});
for (j = 0; j < right_procNum; j++) {
   recv(&number, P_{i-1});
```

```
    if (number > x) {
       send(&x, P_{i+1});
       x = number;
    } else send(&number, P_{i+1});
}
send(&x, P_{i-1});                          /* send number held */
for (j = 0; j < right_procNum; j++) {       /* pass on other numbers */
    recv(&number, P_{i+1});
    send(&number, P_{i-1});
}
```

一个进程越靠近主进程，通过它传递的数就越多。在编程时要非常小心保证每个进程有正确数目的recv()和send()。进程$n-1$没有recv()，但有一个send()（到进程$n-2$）。进程$n-2$有一个recv()（从进程$n-1$处）和两个send()（到进程$n-3$），以此类推。send()和recv()很容易不匹配，这种情况下就将导致死锁。

分析

假设把比较交换操作看作为一个计算步，这个算法的顺序实现需要

$$t_s = (n-1) + (n-2) + \cdots + 2 + 1 = n\,(n-1)/2$$

因为它找到最大数需要$n-1$步，在余下的数中找到次大数需要$n-2$步，以此类推。大约的步数是$n^2/2$，显然，顺序排序算法非常低效且只适用于n非常小的情况。

如果有n个数要排序的话，n个流水线进程，那么并行实现需要$n + n - 1 = 2n-1$个流水线周期。每一个周期有一次比较交换操作。通信包括一个recv()和一个send()，最后一个进程除外，它只有一个recv()；但这个细小的差别可以忽略掉。所以，每个流水线周期至少需要

$$t_{comp} = 1$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

总的执行时间t_{total}等于

$$t_{total} = (t_{comp} + t_{comm})\,(2n-1) = (1 + 2(t_{startup} + t_{data}))\,(2n-1)$$

如果结果是向左传回到主程序的，就可以得到图5-16所示的时序图，图中需要$3n-1$个流水线周期[Leighton,1992]。当然，由于通信介质的时延和其他原因，实际的并行程序中的操作不可能像图5-16中所示的那样完全同步。

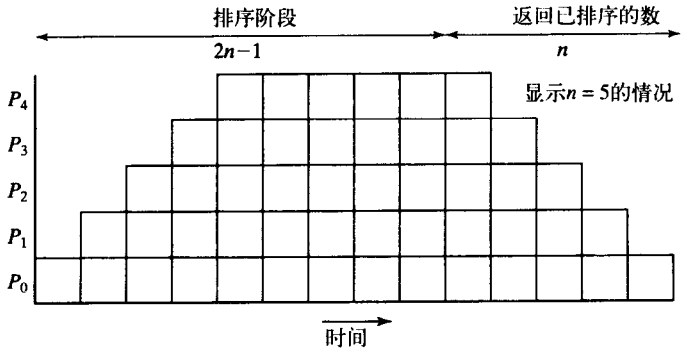


图5-16　插入排序和结果返回

5.3.3　生成质数

生成质数的经典方法是两千多年前由塞利尼的埃拉托色尼（Eratosthenes）提出的埃拉托色尼筛选[Bokhari，1987]。在这种方法中，所有的一系列整数从2开始产生。第一个数2是质

数并且给予保留，所有2的倍数都被删除，因为它们不可能是质数。对余下的每个数字重复这个过程。这个算法是将非质数删除，留下的就仅有质数了。

例如，假设我们想要得到2到20之间的质数。我们从所有的数开始：

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

在考虑了2之后，我们得到

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

这里划/的数字表示它们不是质数且不再被考察。当考察了3之后，我们得到

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

后继的数字以同样的方式被考察。然而，为了找出到n为止的质数，只需考察从头到$\sqrt{n}$的数。因为所有比$\sqrt{n}$大的倍数由于也是小于或等于$\sqrt{n}$的某数的倍数而已经是被删除了。例如，如果$n = 256$，($\sqrt{n}$=16)，就没有必要考察17的倍数，因为17 × 2已作为2 × 17被删除了，17 × 3已作为3 × 17被删除了，对其他超过16 的可以此类推。所以在我们的例子里，我们只要考察2和3就可以得到20以内的质数。

应该指出的是这种基本的算法不适合顺序地寻找非常大的质数（这恰恰又是人们最感兴趣的），原因是顺序的时间复杂性非常之大并且由早先扫描表的遍数决定而，一个提高性能的简单办法是只考虑奇数（这被留作练习）。

1. 顺序代码

这个问题的顺序程序通常使用一个元素初始化为1（TRUE）的数组，当某元素下标对应的数不为质数时，再把它置为0(FALSE)。假设最后一个数为n且n的平方根为sqrt_n，于是我们有

```
for (i = 2; i <= n; i++)
   prime[i] = 1;                          /* Initialize array */
for (i = 2; i <= sqrt_n; i++)             /* for each number */
   if (prime[i] == 1)                     /* identified as prime */
      for (j = i + i; j <= n; j = j + i)  /* strike out all multiples */
         prime[j] = 0;                    /* includes already done */
```

数组的元素仍然用置为1标识质数（数组下标对应的数字）。通过检测数组中的1元素就能找出所有质数。

剔出质数的倍数所需的迭代次数依赖于这个质数本身。2的倍数有$\lfloor n/2-1 \rfloor$次，3的倍数有$\lfloor n/3-1 \rfloor$次，等等。因此，总的顺序时间为 152

$$t_s = \left\lfloor \frac{n}{2}-1 \right\rfloor + \left\lfloor \frac{n}{3}-1 \right\rfloor + \left\lfloor \frac{n}{5}-1 \right\rfloor + \cdots + \left\lfloor \frac{n}{\sqrt{n}}-1 \right\rfloor$$

假设每一次迭代等于一个计算步。顺序的时间复杂性为O(n^2)。

上面这种实现是非常低效的，因为内层循环剔出的数可能已经被前面考察过的数所剔出。实际上，每次扫描仅需从质数i的平方i^2开始，而不是从2i开始。例如，考察数5时，扫描可以从25（也就是5 × 5）处开始，因为5 × 2，5 × 3，和5 × 4已经被前面的质数考虑过了。可以在[Quinn，1994]找到埃拉托色尼筛选这一版本的分析。

2. 并行代码

注意到前面的表达式中前几项决定了总的时间（2的倍数比3多，3的倍数比4多，以此类推）。采用每个进程负责剔出一个数的倍数的基于划分的并行实现不可能是高效的。实际上，[Quinn，1994]指出了这种方法的加速比的上限大约是2.83而与使用的处理器的数目无关（在一定假设

前提下)。[Bohkari，1987]也发现这种方法在实际条件下只能使用有限数目的处理器。这个问题可以用其他方法解决，例如，每个进程负责剔出一定范围内的数的倍数（参考习题5-11)。

流水线的实现是相当有效的。首先，第一个流水线级输入一系列连续的数，然后剔出所有2的倍数并把余下的数传递给第二级。第二级剔出所有3的倍数并把余下的数传递给第三级，以此类推。如图5-17所示。这里，流水线级的个数必须和质数的个数相等（除非用了“块”划分，即每个流水线级处理一组数)。注意，流水线的实现并没有顺序版本会重复考察已标为质数的数的缺点。

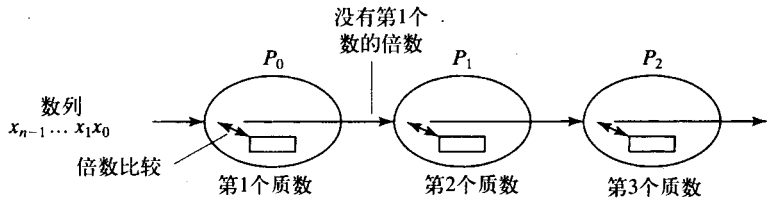


图5-17　进行埃拉托色尼筛选的流水线

进程P_i的代码可以是:

```
recv(&x, P_{i-1});
/* repeat following for each number */
recv(&number, P_{i-1});
if ((number % x) != 0) send(&number, P_{i+1});
```

153

因为每个进程将接收不同数目的数且事先也不知道数的多少，所以用一个简单的for循环来重复这些操作是不够的。在流水线中处理这种情形的一个通常的技术是在序列的末尾发送“终止符”消息，那么每个进程的形式可为如下形式:

```
recv(&x, P_{i-1});
for (i = 0; i < n; i++) {
   recv(&number, P_{i-1});
   if (number == terminator) break;
   if (number % x) != 0) send(&number, P_{i+1});
}
```

注意　使用取模算符%去检测一个数是否是另一个数的倍数是不合算的（在执行时间上)。我们在顺序代码中有意地避免使用它。怎样在并行代码中避免使用它将作为一个练习（习题5-7)。

分析

和排序一样，这个流水线实现也属于类型2。该算法的分析和排序算法类似，只是由于后面的流水线级并不接收前一个流水线进程接收的所有数，所以每个流水线进程完成任务所需的步骤比前一个流水线进程要少。

5.3.4　线性方程组求解——特殊个例

最后一个例子是类型3，即进程传递信息后还可继续有用的工作。这里要解决的是上三角的线性方程组:

$$
\begin{aligned}
&a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad \cdots \quad + a_{n-1,n-1}x_{n-1} = b_{n-1}\\
&\qquad\vdots\\
&a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2\\
&a_{1,0}x_0 + a_{1,1}x_1 = b_1\\
&a_{0,0}x_0 = b_0
\end{aligned}
$$

式中a和b都是常量，x是要求的未知数。这种求解未知数x_0、x_1、x_2、…、x_{n-1}方法是一种简单地重复回代。首先，从最后一个等式可以得到未知数x_0，即

$$x_0 = \frac{b_0}{a_{0,0}}$$

把x_0的值代入下一个等式可以得到x_1，即

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

154

把x_1和x_0的值代入再下一个等式就可以得到x_2，即

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

以此类推，直到得到所有的未知数。

显然，这个算法可以用流水线来实现。第一个流水线级计算出x_0并且把x_0传递给第二个流水线级，第二个流水线级从x_0计算出x_1并把x_0和x_1传递给第三个流水线级，第三个流水线级又从x_0和x_1计算出x_3，以此类推，如图5-18所示。每一级用一个进程实现。n个等式有n个进程（如，$p = n$）。第i个进程($0< i <p$)接收x_0、x_1、…、x_{i-1}的值并从下面的等式中计算出x_i:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j}x_j}{a_{i,i}}$$

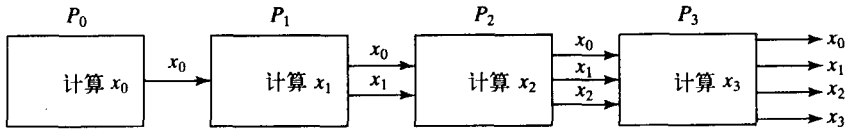


图5-18 用流水线解上三角线性方程组

1. 顺序代码

假设常量$a_{i,j}$和b_k分别保存在数组a[ ][ ]和b[ ]中，未知数保存在数组x[ ]中，串行代码为

```
x[0] = b[0]/a[0][0];                    /* x[0] computed separately */
for (i = 1; i < n; i++) {               /* for remaining unknowns */
   sum = 0;
   for (j = 0; j < i; j++)
      sum = sum + a[i][j]*x[j];
   x[i] = (b[i] - sum)/a[i][i];
}
```

2. 并行代码

一种流水线版本的进程P_i(1<i<p) 的伪代码为[⊖]

```
for (j = 0; j < i; j++) {
   recv(&x[j], P_{i-1});
   send(&x[j], P_{i+1});
}
sum = 0;
for (j = 0; j < i; j++)
   sum = sum + a[i][j]*x[j];
```

⊖ 有实现回代的另一种流水线解决方案，参见第10章。

```
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], P_{i+1});
```

P_0只是计算x_0和传递x_0。P_i在接收和传播值以后还有附加计算要完成。这就导致了图5-19所示的时序特性。

P_i的代码如下：

```
sum = 0;
for (j = 0; j < i; j++) {
   recv(&x[j], P_{i-1});
   send(&x[j], P_{i+1});
   sum = sum + a[i][j]*x[j];
}
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], P_{i+1});
```

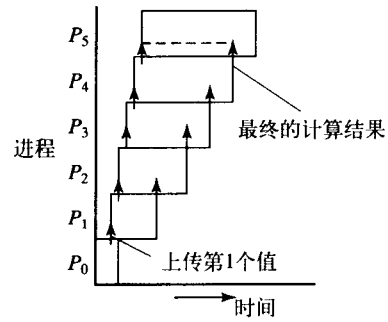


图5-19　使用回代的流水线处理

3. 分析

对于这个流水线，我们不能假定每个流水线级的计算量是相同的（参见图5-19）。第一个进程P_0执行一次除法和一次send()。第i个进程（$0< i <p-1$）执行i次recv()，i次send()s，i次乘/加，一次除/减，最后还有一个send()。假如乘、加、除、减分别为一个计算步，则总的通信次数是$2i + 1$，而总的计算步数也是$2i + 2$。最后一个进程P_{p-1}执行$p-1$次recv()，$p-1$次乘/加和一次除/减，总共是$p-1$次通信和$2p-1$个计算步。图5-20显示了通信时间与乘/加或除/减的时间都一样的操作。在这种情况下send()和recv()同步得很好，且并行的执行时间就等于最后一个进程的时间加上$p-1$个send()再加上一次除法（计算x_0）的时间。

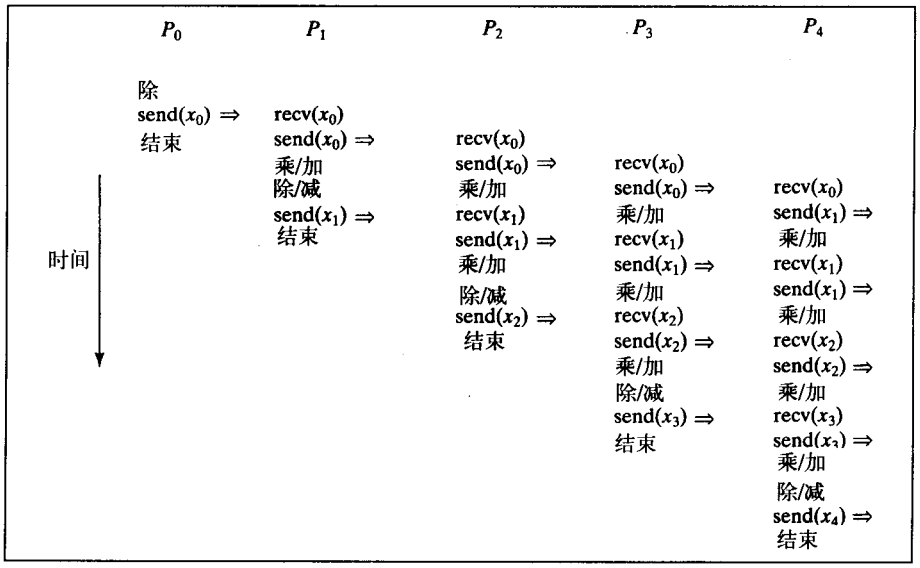


图5-20　回代流水线中的操作

基本上，当$p = n$时这种并行实现的时间复杂性是O (n)。顺序版本的时间复杂性是O (n^2)。但是，实际上的加速比并不是n，这个值严重依赖于实际的系统参数。我们期望一个计算步比一次通信步快，即使是除法也如此。为了减少通信的开销，我们使用非阻塞的发送例程以使发送的源进程尽可能快地进入下一步计算。处理器通常受阻塞接收限制。

4. 线性方程组求解的最后总结

我们已经研究了如何并行化上三角的线性方程组的解法（显然它也适用于下三角形的线性方程组）。实践中出现过这样的等式，例如，[Quinn，1994]就描述了一个上三角形线性方程组来求一个包括有电阻和电源的电路中的电流。然而更重要的是回代法是用高斯消去法解一般形式的线性方程组的关键部分。我们将在第11章中描述解线性方程组的高斯消去法。高斯消去法首先把线性方程组转化为三角形式之后用回代法解方程。

5.4 小结

本章介绍了以下内容:

- 流水线的思想和它的应用范围
- 流水线的分析
- 显示流水线潜力的例子，包括插入排序、生成质数、解上三角线性方程组

157

推荐读物

流水线处理经常出现在用于算术算法的特殊的超大规模集成电路（very large scale integration VLSI）部件中。除了只能从一端输入数据的简单的一维流水线外，还有能同时从左右两端输入数据并能同时在两个方向传播信息的更复杂的流水线或线性阵列。而且可以特别地设计以VLSI实现的二维阵列。我们在第11章将要考虑对向量和矩阵进行操作的二维阵列。这些阵列在第11章的脉动阵列分类中可以看到。流水线也能设计成对数字的位进行操作以实现各种各样的算术运算。[Leighton，1992]讨论了这种流水线的应用。

埃拉托色尼筛选是生成质数的一个基本方法，并且已经在顺序程序设计教程里作为例子出现了很多次。Bokhari(因早期在多处理器映射问题方面的工作而闻名)提出了在共享存储器多处理机上用埃拉托色尼筛选作为基准程序得到的结果[Bokhari,1987]。[Lansdowne, Cousins, and Wilkinson，1987]继续研究这个问题并且演示了一种能提高筛选编程性能的方法。

参考文献

BOKHARI, S. H. (1987), "Multiprocessing the Sieve of Eratosthenes," *Computer*, Vol. 20, No. 4, pp. 50–58.

CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST (1990), *Introduction to Algorithms*, MIT Press, Cambridge, MA.

HENNESSY, J. L., AND D. A. PATTERSON (2003), *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann, San Mateo, CA.

LANSDOWNE, S. T., R. E. COUSINS, AND D. C. WILKINSON (1987), "Reprogramming the Sieve of Eratosthenes," *Computer*, Vol. 24, No. 8, pp. 90–91.

LEIGHTON, F. T. (1992), *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA.

QUINN, M. J. (1994), *Parallel Computing Theory and Practice*, McGraw-Hill, NY.

WILKINSON, B. (1996), *Computer Architecture Design and Performance*, 2nd edition, Prentice Hall, London.

习题

科学/数值习题

5-1 使用流水线方法编写一个计算x^{16}的并行程序。然后用分治方法重新编写这一并行程序。

从理论分析和实验两个方面对这两种方法进行比较。

5-2　根据下面的公式设计一个计算$\sin\theta$的流水线解决方案:

$$\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \frac{\theta^9}{9!} - \cdots$$

输入一系列的值θ_0，θ_1，θ_2，θ_3，…。

158　5-3　修改习题5-2的程序以计算$\cos\theta$和$\tan\theta$。

5-4　编写一个使用流水线方法的并行程序计算下列多项式:

$$f = a_0x^0 + a_1x^1 + a_2x^2 + \cdots + a_{n-1}x^{n-1}$$

其中a_i $(0 \leqslant i < n-1 \neq 0)$，$x$和$n$为输入。把这种方法和分治方法（第4章的习题4-8）进行比较。

5-5　探讨5.3.1节中所述的流水线加法中增加数据划分的平衡点，为你的系统编写一个找出最优化数据划分的并行程序。

5-6　比较插入排序（5.3.2节）的顺序实现和流水线实现在加速比和时间复杂性方面的不同。

5-7　重新编写5.3.3节中的生成质数的并行程序，要求避免使用取模运算符以使算法更加有效。

5-8　基数排序类似于4.2.1节中描述的桶排序，只是使用了数的位来确定每个数应放在哪个桶里。首先根据最高位把每个数分别置于两个桶中的一个，接着根据次高位把每个桶中的数再分别置于下两个桶中的一个，以此类推，一直到最低位比较完毕。改写这个算法为流水线方法，所有的数重排序后从一级传递到下一级直至排序完毕。为这种方法编写一个并行程序并对这种方法进行分析。

5-9　一个流水线包括四个级，如图5-21所示。每一级执行的操作为

$$y_{out} = y_{in} + a \times x$$

请指出整个流水线所需的计算量。

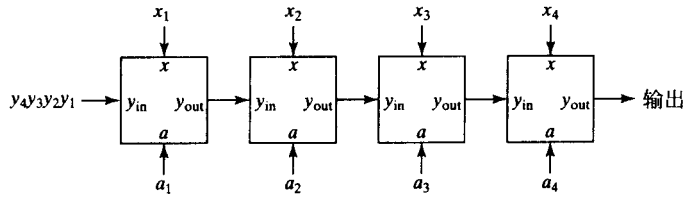


图5-21　习题5-9的流水线

5-10　两个向量（一维数组）A和B的外积产生一个矩阵（二维数组）C，如下

$$AB^T = \begin{bmatrix} a_0 \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix} \begin{bmatrix} b_0 \cdots b_{n-1} \end{bmatrix} = \begin{bmatrix} a_0b_0 & \cdots & a_0b_{n-1} \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ a_{n-1}b_0 & \cdots & a_{n-1}b_{n-1} \end{bmatrix}$$

（A　B^T　C）

159　模式化这一计算的流水线实现，假设A的元素（$a_0, a_1, \cdots, a_{n-1}$）一起从流水线的左边输入，而$B$的每个元素分别保存在每个流水线级中（$P_0$保存$b_0$，$P_1$保存$b_1$等等）。为这个问题写一个并行程序。

5-11　比较下列各种不同的埃拉托色尼筛选的实现方案:

(a) 5.3.3节中描述的流水线方法。

(b) 每个进程负责剔除一个数的倍数。

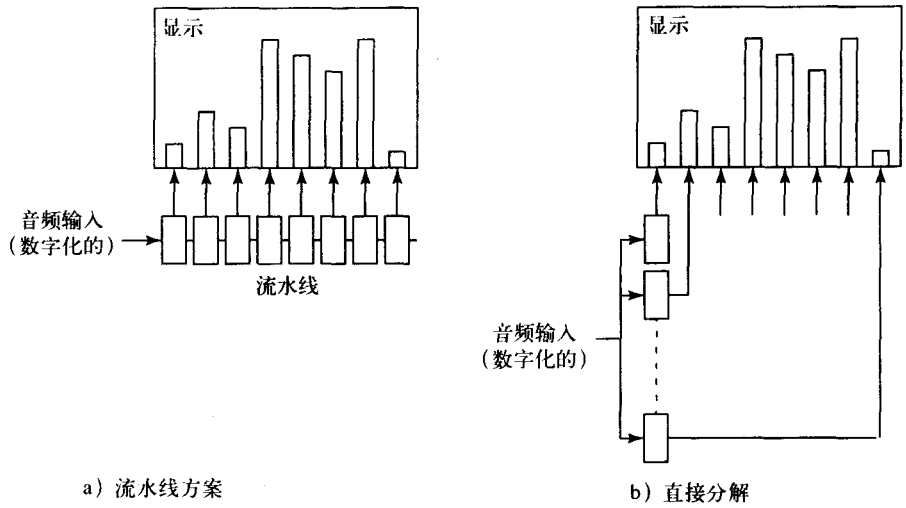
(c) 把数划分为m个区域，为每个进程分配一个区域来剔除质数的倍数。再用一个主进程把已找到的质数广播给各个进程。

请分析每一种方法。

5-12 （只对有计算机体系结构知识的学生）写一个并行程序模拟[Hennessy and Patterson, 2003]中描述的五级流水线的RISC处理器（精简指令集计算机）。这个程序接受一串机器指令并且把它们变为流水线形式的指令流，包括由于相关性/资源冲突所带来的流水线时延。用一个与寄存器相对应的有效位来控制存取寄存器，如[Wilkinson, 1996]所述。

现实生活习题

5-13 5.1节中已提到，可以用流水线的方法实现音响系统中的音频振幅直方图显示，如图5-22a所示。这种应用也可以用另外一种易并行的、功能分解的方式来实现，即每个进程直接接收音频输入，如图5-22b所示。请使用一个音频文件作为输入，分别用两种方法编写并行程序来产生频率振幅直方图的显示，然后分析这两种方法。（可能要对怎样识别数字信号中的频率做一些研究。）



a) 流水线方案　　b) 直接分解

图5-22 音频直方图显示

5-14 由于始料未及的汽车数量的增加和州政府对汽车检验要求的增加，New Caroltucky州的居民们在抱怨为完成一次汽车检验花的时间太长。典型地，一个刹车有35个检验点（6项检查是独立的：卸下每个轮子，然后测量刹车内套/衬垫的厚度，检查主汽缸的密封性和性能，寻找总刹车索上的泄漏和裂缝），其他29项检查也是很费时的。一旦开始对车辆检查，一般需用时一小时；一些人声称他们为了进入检验间就不得不排队等待。立法委员们接到报告，在极端的情况下，队列的时延甚至要72小时。 160

于是New Caroltucky州的立法机关准备决定是否应该改进州立检验站的工作。因为立法机关听说你正在学习一门关于顺序和并行思想的课程，所以他们决定雇佣你对目前的检验系统和设想的检验系统做个模拟。要求假设政府准备把目前的“纯顺序”的检查过程改为“流水线”方式的话，你要确定总时间（队列等待时间加上检查时间）

的减少。

目前的检查过程开始于司机把车开进检验站的等待队列。当一个检验员空闲下来时，队列中的第一辆车就开进检验站。然后按顺序执行35项检验。如果检验通过，再把它开出检验间并盖上一个戳，整个耗时约一小时。

两种设想的检验过程都是从汽车开进队列开始的。在设想新的“改进顺序”型系统中，有三名检验员在三个独立的检验间里同时对三辆汽车进行检验，每名检验员准备就绪时，就从队列的队头开进一辆汽车进行检验直至检验完毕。由于空间的限制(只有三个检验间)，不可能增加更多的检验员同时检验更多的汽车。

而在设想的新的“流水线”系统中，政府安装了一些自动装置可以使汽车自动地在检验间之间传送：先进入第1检验间，从那里出来后进入第2检验间以让下一部汽车进入第1检验间，再进入第3检验间以让第2部汽车进入第2检验间，并且让第3部汽车进入第1检验间。这种方法能够增加检验员人数以加速检验的步骤。例如，可以先为每一个轮子设一个检验员（负责卸轮子、测量内套/衬垫、检验轮汽缸泄漏、更换轮子等)，再设第五个检验员负责寻找刹车索上的泄漏。自然，政府关心的是代价和效率；他们只雇用在获得最大吞吐量的情况下最小数目的检验员。检验员没有事干是不能容忍的，而开除检验员就会导致总的检验时间的增加。

每个检验间有一张任务表，表中有完成每一项任务需要的时间，还有每个检验员的费用（基本工资、福利、办公费用)。

你的任务就是模拟这两种新的检验系统以确定几个结果：

(a) 在新的流水线系统中，获得最大检验吞吐量需要的检验员的最小数目是多少?

(b) 在每一种设想的新的系统中，每次检验的劳动力费用是多少?

(c) 在每一种设想的新的系统中，预期等待时间会减少多少?

(d) 在不做进一步模拟的情况下（仅分析前面得到的结果)，对于政府投资新设备、增加检验间的数量以进一步减少平均检验时间的两种新系统作一个评估。(自然，流水线系统中分配给每一个检验间的任务会有改变，但是我们假设可以重新培训检验员。)

161

任务表

a. 卸左前轮	1分钟
b. 卸左后轮	1分钟
c. 检验内套/衬垫（每个轮子）	1分钟
...	
i. 换左前轮	1分钟
j. 校直轮子	5分钟
k. 检验排气装置的泄漏	1分钟
l. 检验空载时的引擎排放	4分钟
m. 检验负荷时的引擎排放	3分钟
...	
z. 把旧图章换成新图章	2分钟
总共	60分钟

人工费用表

1. 检验员（“工蜂”）	$40 000/年
2. 经理（“雄蜂”）	$60 000/年
3. 高级经理（“蜂后”）	$80 000/年

注意1：每5个检验员（或是其中一部分）配一个经理，超过两个经理后每4个经理（或是其中一部分）配一个高级经理。例如，如果有13个检验员，就需要3个经理和一个高级经理。

注意2：这是一个开放式的问题，需要学生做一些假设，如到达率、随机到达时间等等。而且相对于平常的作业而言，这个问题可能更适合于做为整门课程最终的课程设计。

5-15 回顾影片或新闻报导有关人类链从一个库存区传递物品到需要这些物品的地方。（有关例子包括接力地传递装满的沙袋到河堤以建立堤防防止河水溢过河堤，以及传递水桶救火队接力地将水桶从供水处传递到火场。）根据以下给定的数据模拟一个N人链，并与N个人独立地从一个库存区搬运物品到需要这些物品的地方的方法进行比较。目的是要确定加速比，即通过流水线解决方案比通过独立操作解决方案在传递所需物品时所能获得的传递速率的增加。假设要搬运一百万件物品，针对可用的人数分别为150、300和3 000的情况确定加速比。

数据：从库存区到需要物品的地方的距离是300米；一个人独立工作时每次可搬运一件物品，搬运物品时和空手时（返程时）的行走速度分别为每秒1米和1.5米。集体工作时人的间隔距离为1米，手对手地从后面一个人手中接过物品再将该物品传给前面一个人所需的时间为1.25秒。显然，如果人很少的话，链或流水线的方法是不实用的。类似地，如果有多个300人的链。那么多个链可并行操作。 162

第6章 同步计算

本章中我们讨论由一组必须不时互相等待即同步才能继续独立计算的求解问题。这类应用中很重要的一种被称为全同步（fully synchronous）应用。在全同步应用中，所有的进程在一些规则的点上同步。通常，一个同样的计算或操作应用于一组数据点，所有操作以类似于SIMD计算的锁步方式同时开始执行。Fox和他的同事们在研究中发现，在加州理工学院（Caltech）具有开创性意义的研究项目中，有70%的第一类应用可以归为同步应用这一类[Fox, Williams, and Messina, 1994]。下面我们先讨论同步进程，然后讨论全同步应用。最后，我们叙述为了增加计算速度如何减少所需的同步数量，我们称其为部分同步（partially synchronous）方法。为获得高的计算速度部分同步方法是非常重要的。

6.1 同步

6.1.1 障栅

想像有许多计算值的进程，每个进程最终都必须相互等待，直到所有的进程都到达了计算中某一个指定的参考点。这种情况通常发生在进程需要相互交换数据然后从一个已知状态一起继续运行的场合。如果可以动态创建进程，那么通过进程退出和再派生就可以实现这种效果，但这种方法开销大耗时多。采用某种机制阻止任何进程通过某个特定点，直到所有进程就绪后才让进程继续。实现这种同步的基本机制称为障栅（barrier）。在每个进程必须等待的点处插入一个障栅，当所有进程到达它时（在某些实现中，到达的进程超过一定数目时），所有的进程才能从这一点处继续运行。图6-1中对此概念做了说明，在该例中，进程P_2最后一个到达障栅。所以，所有其他进程都得等待或转入不活动状态直到进程P_2到达它的障栅。此后不活动的进程将被唤醒（重启）且所有进程各自向自己的目标点前进。

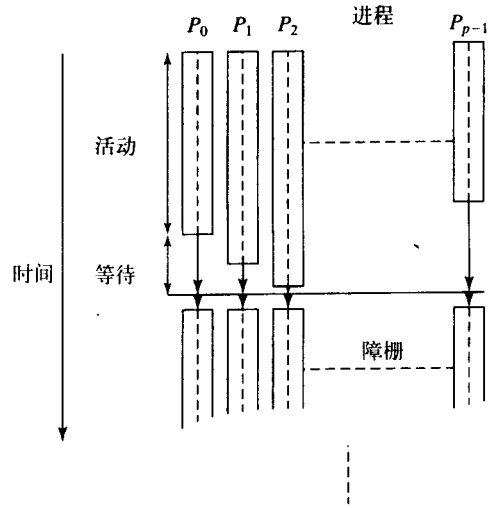


图6-1 进程在不同时间到达障栅

障栅可以应用于共享存储器系统或消息传递系统。我们将在第8章和第9章中讨论应用于共享存储器系统的障栅。在消息传递系统中，障栅常常以库例程的形式提供。例如，MPI的障栅例程是`MPI_Barrier()`，唯一的参数是一个已命名的通信子。组中的每一个进程都要调用`MPI_Barrier()`，它阻塞进程直到所有组的成员到达障栅调用并且只在那时才返回。虽然不是在MPI中允许障栅定义为说明必须到达障栅的进程数目以释放进程，且该进程数可以少于总的组进程数，但利用这种特征的情况很少见。障栅自然是同步的，也就用不着消息标记了。

图6-2示出了障栅的库调用的方法。因为单个障栅调用在需要障栅的情况下被重用，所以障栅匹配其他进程中正确的障栅是很关键的。通过实现可以确保这个特性。障栅调用的实现

方式取决于实现者，实现者反过来也受基本体系结构的影响。不同的基本体系结构有其特定的更有效的实现方式。照常，MPI没有说明它的内部实现。然而，为了评估障栅的复杂性我们需要知道一些该实现方面的细节。下面我们介绍几个常见的障栅实现方式。 164

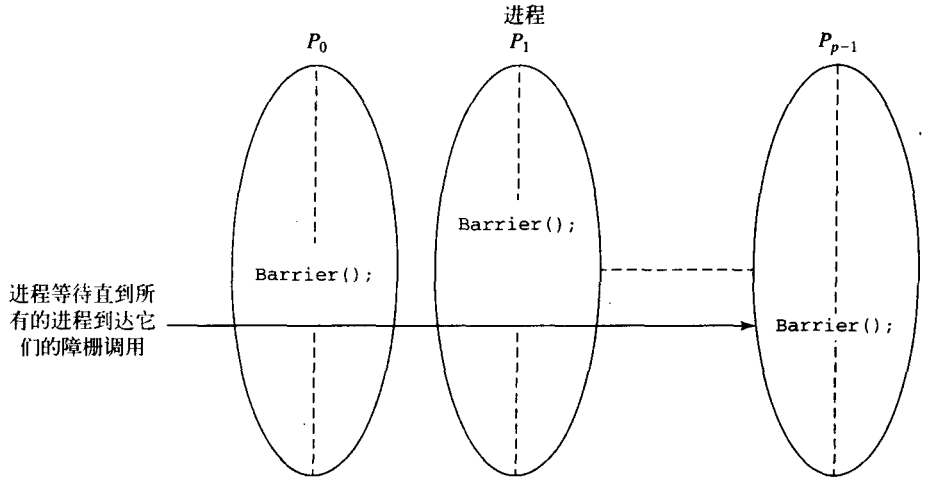


图6-2 库调用障栅

6.1.2 计数器实现

图6-3表示一种障栅实现方式，集中式计数器实现（有时称作线性障栅）。用一个计数器对到达障栅的进程数目计数。在任何进程到达它的障栅前，计数器先初始化为0。然后每个调用障栅的进程将使计数器增加1个增量，并检查是否已达到正确数目p。若计数器未到p，进程 165
就停顿或转入不活动/“空闲”状态。如已达到p，就释放这个进程及其他等待着的进程。需要一种机制来释放“空闲”进程。

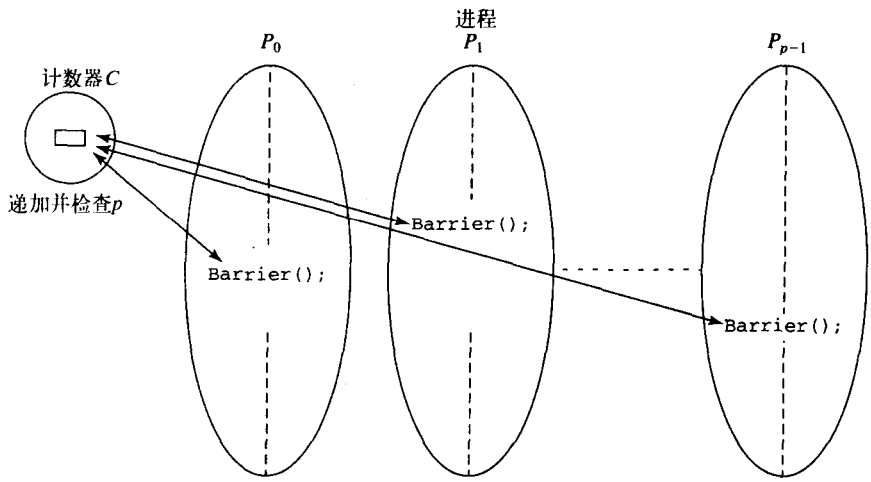


图6-3 用集中式计数器实现障栅

基于计数器的障栅常分两个阶段，一个是到达（或陷入）阶段，另一个是离开（或释放）阶段。进程进入到达阶段后直到所有进程都进入这个阶段才能离开此阶段。然后进程进入离开阶段并被释放。一个良好的障栅实现须考虑一个障栅可能被进程不只使用一次。前面的进

程还没有第一次离开障栅之前一个进程可能第二次进入障栅。两阶段设计要处理这种情况。

假设是主进程维护障栅计数器。在到达阶段，当从进程到达它们的障栅时主进程对来自从进程的消息计数，而在离开阶段主进程释放从进程。使用（本地）阻塞send()和recv()发送/接收消息且用for循环计数的主进程的障栅代码形式如下：

```
for (i = 0; i < p; i++)     /* count slaves as they reach their barrier */
   recv(Pany);
for (i = 0; i < p; i++)     /* release slaves */
   send(Pi);
```

变量*i*是障栅计数器。从进程的障栅代码很简单：

```
send(Pmaster);
recv(Pmaster);
```

完整的方案如图6-4所示。在这段代码中，可以以任何次序接收从进程发来的消息，但送往从进程的消息则是按数字次序的。这种实现允许在一个进程中障栅被重复调用，因为明确定义了所有进程都必须先进入到达阶段才可能转入离开阶段。但应注意，本地阻塞send()并不停
166 止进程。在消息已被构成好准备发送之后但在接收之前从进程直接执行recv()。recv()将阻塞因为进程不会过早移出离开阶段直到它们接收了它们的消息。到达阶段也可用一个集中（gather）例程实现，而离开阶段可用广播例程实现。图6-4中的send()和recv()在消息中没有指定的数据，因此发送的是一个空（NULL）消息。

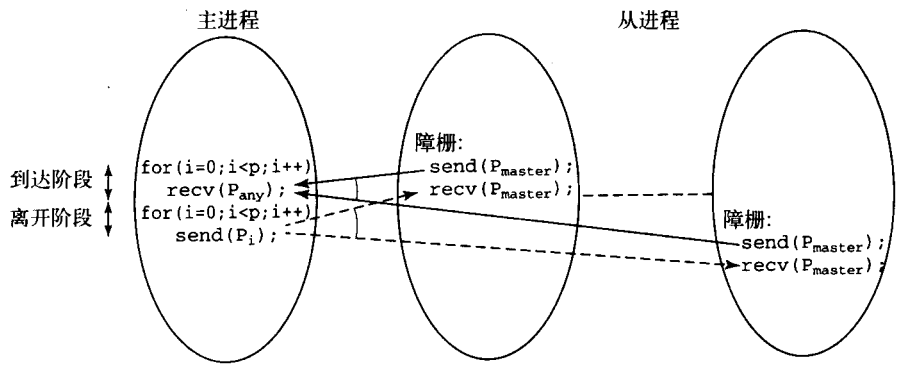


图6-4　消息传递系统中的障栅实现

6.1.3 树实现

障栅用计数器实现的时间复杂性是O（*p*）（包括主进程的计算复杂性和通信复杂性如消息数两者），*p*是进程数。一个更有效的障栅实现可以用2.3.4节中介绍的分散式树结构实现。假设有八个进程P_0、P_1、P_2、P_3、P_4、P_5、P_6、P_7。算法执行如下：

第一阶段：P_1发送消息给P_0；（当P_1到达障栅时）
P_3发送消息给P_2；（当P_3到达障栅时）
P_5发送消息给P_4；（当P_5到达障栅时）
P_7发送消息给P_6；（当P_7到达障栅时）

第二阶段：P_2发送消息给P_0；（P_2和P_3已到达障栅）
P_6发送消息给P_4；（P_6和P_7已到达障栅）

第三阶段：P_4发送消息给P_0；(P_4、P_5、P_6和P_7已到达障栅)

P_0结束到达阶段；(当P_0到达障栅且已收到P_4发来的消息)

现在进程必须从障栅中释放出来。这可以用一个反向的树构造过程来完成。图6-5是完整的障栅构造过程。在这种情况下，该算法只发送和接收消息而没有显式的计算。到达阶段和离开阶段均用树实现的八个进程障栅算法需要2log8步，一般对p个进程的算法需要2log$p$步，通信时间复杂性为O (log p).

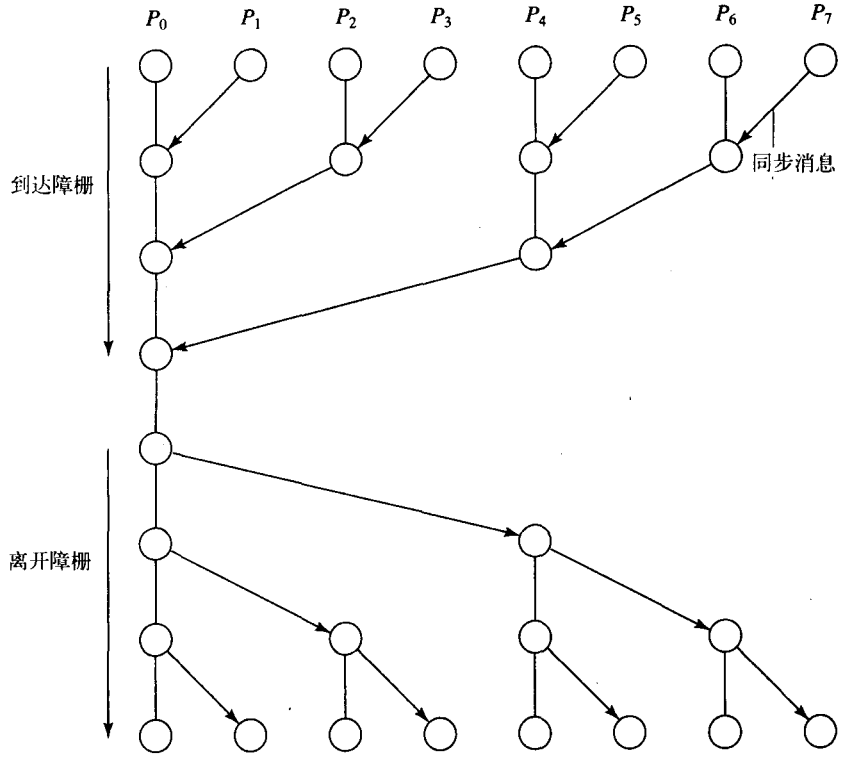


图6-5 树形障栅

6.1.4 蝶形障栅

树构造可以发展成蝶形障栅（butterfly barrier)。在蝶形障栅中进程对按下述方式在每一级进行同步（以八个进程为例):

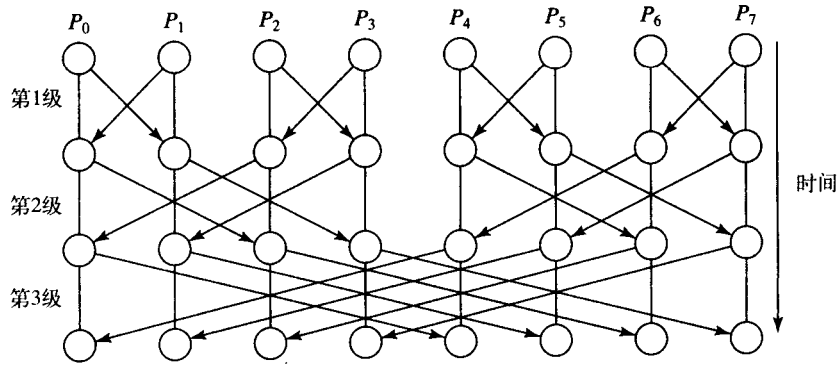
第一级：$P_0 \leftrightarrow P_1$，$P_2 \leftrightarrow P_3$，$P_4 \leftrightarrow P_5$，$P_6 \leftrightarrow P_7$

第二级：$P_0 \leftrightarrow P_2$，$P_1 \leftrightarrow P_3$，$P_4 \leftrightarrow P_6$，$P_5 \leftrightarrow P_7$

第三级：$P_0 \leftrightarrow P_4$，$P_1 \leftrightarrow P_5$，$P_2 \leftrightarrow P_6$，$P_3 \leftrightarrow P_7$

图6-6中同步进程间的两条“链路”表示两对`send()/recv()`。进程间交换数据时这一点将被用到（如同其他蝶形应用)。对障栅来说，每次同步仅需一对`send()/recv()`。所有 167
的同步级结束后，每个进程都已与其他进程同步，因此所有进程可以继续。

在第s级，如果p是2的乘方，则进程i与进程$i + 2^{s-1}$进行同步；否则，进程i与进程$(i + 2^{s-1})$ mod p进行同步。p个进程的蝶形障栅构造需要log$p$步（p是2的乘方)，是树形障栅实现所需步数的一半，但通信时间的复杂性同样是O (log$p$)。

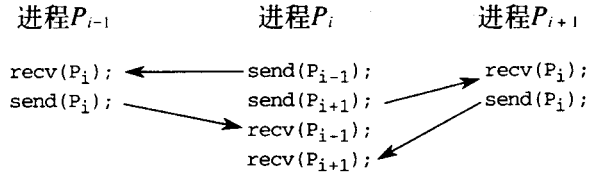


168

图6-6　蝶形障栅

6.1.5　局部同步

有些问题只需进程与一些进程同步而不是与该问题的全体进程同步。进程组织成网格或流水线结构的算法中经常出现这种情况，此时只有相邻进程才需同步。6.1.2节中使用的消息传递技术可以简化成仅在需同步的进程间发送消息。例如，设进程P_i在继续执行前需要和P_{i-1}、P_{i+1}同步并交换数据，那么代码为：



注意，这不是个很好的三进程障栅，因为进程P_{i-1}只需与P_i同步，一旦P_i允许它就可继续，P_{i+1}只需与P_i同步与此类似。不过，在许多应用中，这种同步方法已经满足了。

6.1.6　死锁

在此叙述的树构造、蝶形障栅和局部同步算法使用同步例程来实现进程间的同步。当一对进程各向对方发送和接收消息时，可能会发生死锁。如果两个进程在调用同步例程（或调用阻塞例程而又没有足够缓存）后都先进行发送，就会导致死锁。因为谁也不能返回，双方都等着匹配的接收，而接收永远无法到达。显然，这一问题的解决方法是安排一个进程先接收后发送，另一个先发送后接收。对偶数标号的进程仅与奇数标号的进程通信的场合，反之亦然，如线性流水线系统中，让偶数标号的进程先执行发送而奇数标号的进程先执行接收，就可以避免死锁。

由于双向数据传输很普遍，可以提供一个复合阻塞例程`sendrecv()`，由内部实现细节负责避免死锁。`sendrecv()`例程发送一个消息给目的进程并从源进程接收一个消息。为了灵活性，源和目的可以是不同或同一个进程。MPI中提供了此类例程`MPI_Sendrecv()`。它还提供了`MPI_Sendrecv_replace()`，它发送和接收消息使用同一个缓冲区，接收来的消

169 息将替换发送的消息。实现这些例程可避免死锁的发生。把`sendrecv()`用于前面的例子，就可使代码简化为：

进程 P_{i-1} 进程 P_i 进程 P_{i+1}

```
sendrecv(Pi);  <->  sendrecv(Pi-1);
                    sendrecv(Pi+1);  <->  sendrecv(Pi);
```

MPI_Sendrecv()例程的参数达12个之多。有人对某些MPI例程的参数列表之庞大颇有微词，它倒可以充当一个例证了。

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
             recvcount, recvtype, source, recvtag, comm, status)
```

这些参数实质上是MPI_Send()和MPI_Recv()参数列表的并集。

6.2 同步计算

6.2.1 数据并行计算

一种隐含同步要求的计算是*数据并行计算*。在数据并行计算中，相同操作同步在不同的数据元素上执行，即并行。数据并行编程特别方便有两个原因，第一是容易编程（本质上只要一个程序），第二是可以方便地扩展问题规模。许多数值问题和某些非数值问题都可以数据并行化。在1.3.4节中曾简单提及的SIMD（单指令流多数据流）计算机就是数据并行化计算机，不同处理器用同样的指令处理不同的数据，所有操作都是同步的。在SIMD计算机中，同步由硬件实现，处理器以锁步的方式操作。

数据并行计算的一个简单例子是给数组的每个元素加上同一个常量:

```
for (i = 0; i < n; i++)
  a[i] = a[i] + k;
```

语句a[i]=a[i]+k可以被多个处理器同时执行，每个处理器使用一个不同的下标i值(0<i<n)，如图6-7所示。在SIMD计算机上，等价于a[]=a[]+k的相同指令将被同时送往每个处理器。

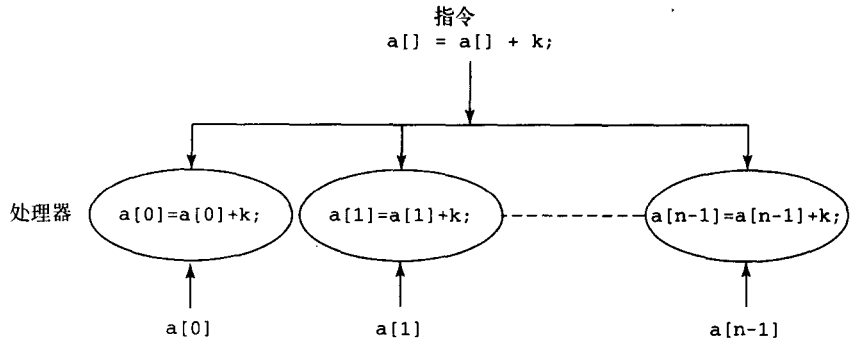


图6-7 数据并行计算

并行程序设计语言中，有一个特定的“并行”结构用以指明数据并行操作，即forall语句。下面这条forall复合语句:

```
forall (i = 0; i < n; i++) {
  body
}
```

表示循环体的n个实例（instance）可同时被执行。循环变量i在每个循环体的实例中有一个不同的值，第一个实例的i值为0，下一个为1，以此类推。循环变量可用来“个性化”循环体拷贝（例如，访问数组的不同元素）。为说明这一点，给数组a的每个元素加上同一个常量k，我们可以编写如下代码：

```
forall (i = 0; i < n; i++)
  a[i] = a[i] + k;
```

不管何种情况，每个循环体实例必须独立于其他实例。（第8章中我们将讨论如何从数学上建立这种独立性。）遗憾的是术语forall其实不含有循环迭代，不过是借以表示有n个循环体的拷贝，每一个被赋予了不同的i值。

虽然我们不讨论适用于SIMD计算机的程序，但数据并行技术可以用于多处理机和多计算机中。在这些并行计算机中，循环体实例在不同处理器上运行，直到所有实例都执行完后整个结构才算执行完。因而，在forall结构中障栅是隐式的。在使用库例程的消息传递计算机中，forall结构一般是不可用的，因此需要有一种显式的障栅。例如，在SPMD（单程序多数据）程序中，数组元素加k的程序编写如下：

```
i = myrank;
a[i] = a[i] + k;          /* body */
barrier(mygroup);
```

其中myrank是进程的序号，取0至$n-1$之间的值。可以认为每个进程都可访问所需的数组元
171 素。一般来说，由于障栅的开销，这么小的循环体是比较低效的。

我们可以构造比数组元素加一个常量复杂得多的SIMD计算。[Hillis and Steele，1986]描述了几个数据并行算法，包括求和、排序和链表操作。某些SIMD计算机和数据并行算法在位模式而不是完全在数据上操作。第12章将介绍的许多图像处理算法就是操作在位模式上的数据并行算法。

前缀求和问题

数据并行算法的一个例子是*前缀求和*（prefix sum）问题。在前缀求和问题中，给定一个数到x_0、…、x_{n-1}，要求算出所有的部分和（即$x_0 + x_1$；$x_0 + x_1 + x_2$；$x_0 + x_1 + x_2 + x_3$；…）。前缀运算也可定义在除加法之外的组合运算上，如乘、求最大值、求最小值、字符串的连接和逻辑（布尔）运算（与、或、异或等）。多种计算模型的研究都会牵涉到前缀问题。前缀运算在处理器分配，数据压缩，排序和多项式方程计算等领域中有实际应用[Wagner and Han，1986]。

前缀求和问题的顺序代码如下：

```
sum[0] = x[0];
for (i = 1; i < n; i++)
  sum[i] = sum[i-1] + x[i];
```

这是一个O (n)算法。

图6-8显示一种由[Hillis and Steele，1986]描述的对16个数求部分和的数据并行方法。这种方法构造一个多重树状结构，直接在$x[i]$（$0 \leqslant i < 16$）上计算部分和，原来的值被覆盖（可用另外一个数组保存原值。）每一步的计算量不同，首先是15(16−1)个$x[i-1]$加$x[i]$（$1 \leqslant i < 16$）的加法；然后是14(16−2)个$x[i-2]$加$x[i]$（$2 \leqslant i < 16$）的加法；接着是12(16−4)个$x[i-4]$加$x[i]$（$4 \leqslant i < 16$）的加法。

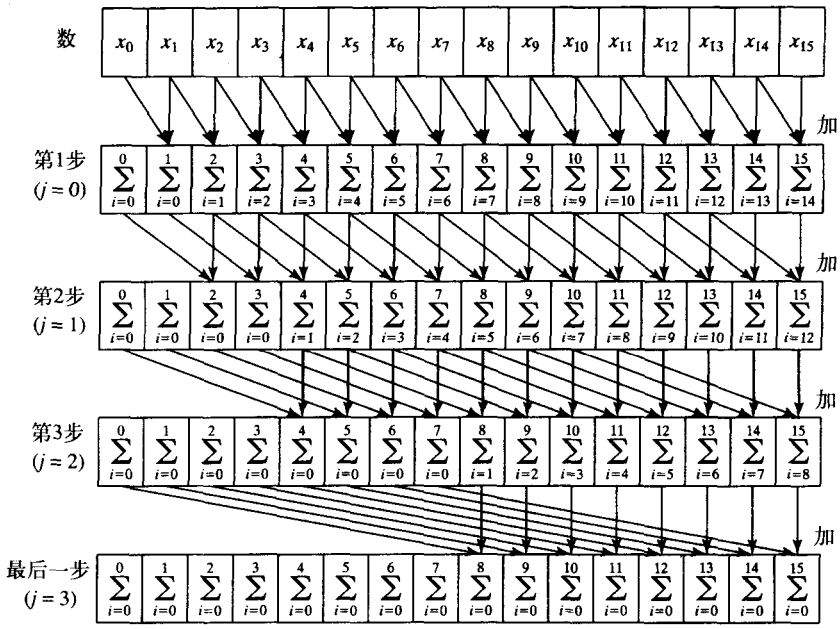


图6-8 数据并行前缀和操作

通常，这种方法对n个数要log$n$步（n是2的乘方）。在第j步（$0 \leqslant j < \log n$），有$n-2^j$个$x[i-2^j]$被加到$x[i]$ ($2^j \leqslant i < n$)。此方法的顺序代码如下：

```
for (j = 0; j < log(n); j++)              /* at each step */
  for (i = 2^j; i < n; i++)               /* add to accumulating sum */
    x[i] = x[i] + x[i - 2^j];
```

因为SIMD计算机发送相同指令给所有处理器，需要一种机制来阻止某些处理器执行该指令。为说明这点，可编写如下并行代码段：

```
for (j = 0; j < log(n); j++)              /* at each step */
  forall (i = 0; i < n; i++)              /* add to accumulating sum */
    if (i >= 2^j) x[i] = x[i] + x[i - 2^j];
```

172

其中，最多使用$n-1$个处理器以及需要log n步。这个并行算法的时间复杂性就计算和通信两方面而言均为O (log n)。由于每步只使用了少数处理器，因而效率<1。

一种使用平衡树的前缀求和算法也是O(log n)算法，但总共需要O (n)次运算而不是总共O (n log n) 次运算。有关对平衡树前缀求和算法的描述可参见[JáJá，1992]。

6.2.2 同步迭代

迭代是重复一种操作，它在顺序程序设计中是一种关键技术。每种编程语言都提供了用于迭代的结构（如for、while或do-while等结构）。迭代方法是求解数值问题的强大工具，特别适用于那些不易处理的数值逼近问题的求解。通常一次迭代产生的中间结果用于下一次迭代，以便更接近实际解。直到得到一个足够接近的实际解，进程才停止重复。迭代方法的基本思想本质上是顺序的，因此不适合并行实现。然而，迭代方法中如果存在多个独立的迭代实例，就可以有效地并行化实现。这有时是问题说明的一部分，有时我们必须重新调整问题说明来获得多个独立的实例。

术语同步迭代或同步并行性用来描述用迭代方法解决问题，每次迭代由几个进程组成，
173 这些进程在每次迭代开始时同时启动，直到所有进程结束前一次迭代时才开始下一次迭代。同步迭代的并行计算体可以用forall结构来说明：

```
for (j = 0; j < n; j++)                    /* for each synchronous iteration */
   forall (i = 0; i < p; i++) {            /* p processes each executing */
      body(i);                             /* body using specific value of i */
   }
```

如写成SPMD程序，则需要一个特定的障栅：

```
for (j = 0; j < n; j++) {                  /* for each synchronous iteration */
   i = myrank;                             /* find value of i to be used */
   body(i);                                /* body using specific value of i */
   barrier(mygroup);
}
```

下面介绍几个具体的同步迭代例子。

6.3 同步迭代程序举例

6.3.1 用迭代法解线性方程组

5.3.4节中介绍过如果一个线性方程组是特殊的三角形方程组形式应如何求解。假设方程组并不是那种特殊形式，而是有n个未知数和n个方程的一般形式：

$$\begin{aligned}
&a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \cdots + a_{n-1,n-1}x_{n-1} = b_{n-1}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\vdots\\
&a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \cdots + a_{2,n-1}x_{n-1} = b_2\\
&a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \cdots + a_{1,n-1}x_{n-1} = b_1\\
&a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \cdots + a_{0,n-1}x_{n-1} = b_0
\end{aligned}$$

其中x_0，x_1，x_2，$\cdots x_{n-1}$ $(0 \leqslant i < n)$是未知数。解此方程组的一种方法是迭代法。把第i个方程$(0 \leqslant i < n)$

$$a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \quad \cdots \quad + a_{i,n-1}x_{n-1} = b_i$$

改写成：

$$x_i = (1/a_{i,i})[b_i - (a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \cdots a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} \cdots + a_{i,n-1}x_{n-1})]$$

即：

$$x_i = \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j}x_j\right]$$

$(0 < i < n,\ 0 < j < n)$。上面这个方程中x_i是其他未知数的显式函数，可作为迭代公式计算每个
174 未知数逼近值。

这种迭代方法就是雅可比（Jacobi）迭代法。在这种方法中，所有的x值同时更新（另一种迭代法——高斯-塞德尔（Gauss-Seidel）法，将在第11章介绍）。可以证明如果方程组对角线上a的绝对值大于同一行中其他a的绝对值之和（a的阵列是对角占优的），那么雅可比法收敛。因此，收敛条件是：

$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|$$

这是个充分条件但不是必要条件，此条件不满足时，方程组也可能收敛。但是，如果对角线上的任一系数等于0该迭代公式将不起作用，因为这意味着要除以0。

迭代法开始时先对所有未知数假设一个初值。例如，设$x_i = b_i$。然后用迭代公式计算未知数的新值。这些新值再代入迭代公式，然后重复下一次迭代。迭代过程持续到所有未知数的值都已满足精度（假设迭代法是收敛的）。解线性方程组也有“直接”的方法，将在第11章介绍。当直接法的计算量太大时，采用迭代法往往比较有效。迭代法的另一个优点是对主存储器空间要求低，缺点是不能保证总是收敛。

1. 终止

并行化中终止是个特别关键问题。我们将在第7章中结合负载平衡对它做更详细地讨论。一个简单的常用方法是比较未知数在迭代前后的计算值，如果所有值都在一个给定的误差范围内，就在第t次迭代时终止计算过程。就是对于所有i，存在:

$$|x_i^t - x_i^{t-1}| < \text{误差范围}$$

式中x_i^t是x_i在第t次迭代后的值，x_i^{t-1}是x_i在第（$t-1$）次迭代后的值。然而，这并不能保证解的精度。假设误差范围为1%并且已计算出一个值是它的上一次计算值的1%。这不是求解的实际值的1%。如果算法是收敛的，我们可以期望下一次x_i的计算值与本次值的差异小于1%，如0.9999%。再下一次值可能相差0.9998%。误差累积起来，就可能使算出的值与最终的准确值偏差很大，如图6-9所示。另外，一次计算值的误差会影响使用它计算出的其他值的精度。

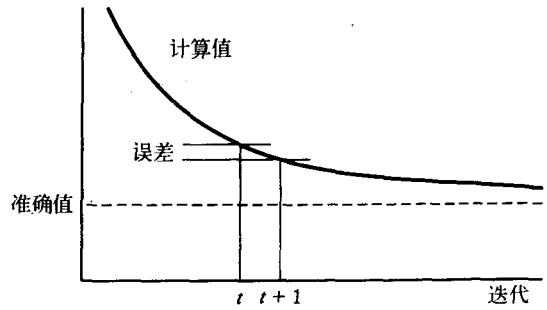


图6-9 收敛速率

[Pacheco，1997]提出一个更复杂的向量终止条件:

$$\sqrt{\sum_{i=0}^{n-1}(x_i^t - x_i^{t-1})^2} < \text{误差范围}$$

175

[Bertsekas and Tsitsiklis，1989]提出另一种终止条件。就是对于所有的i，存在:

$$\left|\sum_{j=0}^{n-1} a_{i,j}x_j^t - b_i\right| < \text{误差范围}$$

这种方法仅使用本次计算出的值而没有使用前一次迭代的值。计算出等号左边的值然后将这个结果与右边的常值比较。由于除$a_{i,i}x_i$外的累加和已在迭代中算出，因而计算量不大。不同终止公式的有效性和收敛性的完整讨论可在数值方法方面的书籍中找到。

不管何种迭代，由于迭代过程可能无限，当达到最大迭代步数时迭代应当停止。是使用复杂的终止条件达到较少的迭代步数，还是使用可能需要较多的迭代步数的简单的终止条件需要综合权衡。在多次迭代后才检查终止条件也许是个不错的策略。并行化要求每次迭代使用前一次迭代的所有结果，所以全局同步是必要的。雅可比迭代法的收敛速度可能较慢。习题6-12从经验角度分析雅可比迭代法的收敛性质。第11章将介绍更快的迭代法。

2. 顺序代码

记线性方程组的系数矩阵为a[][]，方程组右边常数项记为矩阵b[]，用x[]存放各未知数的值，迭代步数固定，则代码可为如下形式:

```
for (i = 0; i < n; i++)
   x[i] = b[i];                                    /*initialize unknowns*/
for (iteration = 0; iteration < limit; iteration++) {
   for (i = 0; i < n; i++) {                       /* for each unknown */
      sum = 0;
      for (j = 0; j < n; j++)              /* compute summation of a[][]x[] */
         if (i != j) sum = sum + a[i][j] * x[j];
      new_x[i] = (b[i] - sum) / a[i][i];           /* compute unknown */
   }
   for (i = 0; i < n; i++)
      x[i] = new_x[i];                             /* update values */
}
```

176

效率对顺序代码同样重要。有许多改善顺序算法代码效率的方法。我们用if语句防止将a[i][i]*x[j]用于a[i][j]*x[j]累加和。避免重复执行if语句的另一种方法是使循环中包括a[i][j]*x[j]，但在循环外减去它，代码如下：

```
for (i = 0; i < n; i++)
   x[i] = b[i];                                    /*initialize unknowns*/
for (iteration = 0; iteration < limit; iteration++) {
   for (i = 0; i < n; i++) {                       /* for each unknown */
      sum = -a[i][i] * x[i];
      for (j = 0; j < n; j++)                      /* compute summation */
         sum = sum + a[i][j] * x[j];
      new_x[i] = (b[i] - sum) / a[i][i];           /* compute unknown */
   }
   for (i = 0; i < n; i++) x[i] = new_x[i];        /* update values */
}
```

还有一种方法是用双重循环计算累加和，第一层从0到$i-1$而第二层从$i+1$到$n-1$。不过我们还是喜欢可读性较好的第二种形式。

3. 并行代码

每个未知数（$p = n$）分配给一个进程，每个进程迭代步数一样。在每次迭代中，未知数的新计算出的值需广播给所有其他进程。在顺序代码中，for循环是每次迭代的一个天然障栅。在并行代码中，我们需要显式插入一个指定的障栅。进程P_i的代码如下：

```
x[i] = b[i];                                   /*initialize unknown*/
for (iteration = 0; iteration < limit; iteration++) {
   sum = -a[i][i] * x[i];
   for (j = 0; j < n; j++)                     /* compute summation */
      sum = sum + a[i][j] * x[j];
   new_x[i] = (b[i] - sum) / a[i][i];          /* compute unknown */
   broadcast_receive(&new_x[i]);               /* broadcast value */
   global_barrier();                           /* wait for all processes */
}
```

广播例程broadcast_receive()把进程i新算出的x[i]值送给所有其他进程并收集其他进程广播的数。因为每个进程要和其他所有进程的“广播接收”的新计算出的值相配，所以一个广播是不起作用的。因此，broadcast_receive()共需要由n个广播组成，每个广播带特定的参数。

另一种简单的方法是使用基本的send()和recv()代替broadcast_receive()，进程i的代码可为：

```
for (j = 0; j < n; j++) if (i != j) send(&x[i], Pj);
for (j = 0; j < n; j++) if (i != j) recv(&x[j], Pj);
```

177

MPI中消息可以送给自己，因此if结构可删去。由于进程在收到所有新计算出的值之前不会继续，故一个独立的障栅也可去掉。

前面讨论过的蝶形障栅能用一种自然的方式在一个复合结构中广播和集中值。因此我们可以用蝶形障栅来改写上面的并行代码。然而，蝶形障栅的效率与基本的体系结构有关，一个预定义的例程也许会有帮助。在MPI中就有这样一个例程，即MPI_Allgather。全集中（Allgather）操作如图6-10所示。每个进程集中相同数量的数据项，在参数表中定义该数量。MPI_Allgather的一个变形称作MPI_Allgatherv，它允许每个进程集中不同数量的数据项。

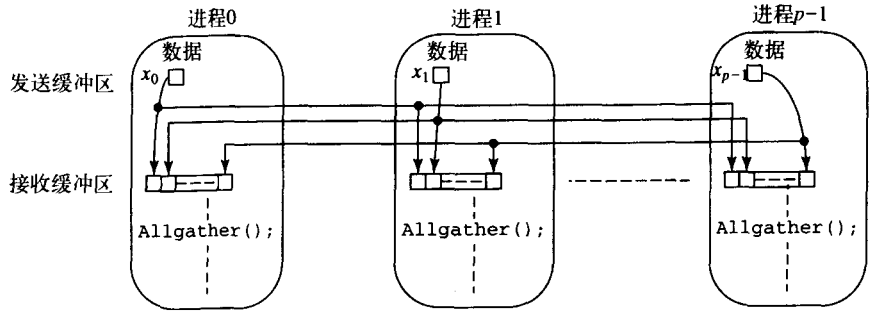


图6-10 全集中操作

178

一般来说，我们希望迭代过程在逼近程度足够高时才停止，而不是受固定的迭代步数的控制（它可能不能提供足够精确的解），每个进程需要检查自己的计算值，如下：

```
x[i] = b[i];                                /*initialize unknown*/
iteration = 0;
do {
   iteration++;
   sum = -a[i][i] * x[i];
   for (j = 0; j < n; j++)                  /* compute summation */
      sum = sum + a[i][j] * x[j];
   new_x[i] = (b[i] - sum) / a[i][i];       /* compute unknown */
   broadcast_receive(&new_x[i]);            /* broadcast value and wait */
} while (tolerance() && (iteration < limit));
```

其中如果准备终止tolerance()返回FALSE，否则返回TRUE。

最简单的机制是当全部进程收敛时才允许进程继续。那么，在同一次迭代中每个进程的tolerance()都返回FALSE，因此所有进程也同时停止。如果我们允许一个进程在自己得到满足精度的解时就停止，那么各个进程的迭代步数就可能不一样。这种情况下，由于广播例程广播到组中所有进程并且希望所有进程都有匹配的例程，因此必须小心以避免死锁。

4. 划分

像所有并行模式一样，通常，处理器的数目比待处理的数据项（这里指正在计算的未知数）的数目少得多。常见的做法是划分问题，使一个处理器处理多个数据项。我们可以简单地按升序给处理器分配未知数，设有p个处理器和n个未知数。处理器P_0负责计算x_0至$x_{(n/p)-1}$，$x_{n/p}$至$x_{(2n/p)-1}$分配给P_1，以此类推（假设n能被p整除）——这就是第4章中的求数字和的块分配。把x_0，x_p，x_{2p}，…，$x_{((n/p)-1)p}$分配给P_0；x_1，x_{p+1}，x_{2p+1}，…，$x_{((n/p)-1)p+1}$分配给P_1的将未知数按

顺序分配给处理器的循环分配方法对这里讨论的问题没有什么优势，反而有不利之处，因为未知数的下标值的计算更复杂，把这些未知数的值打包到一个消息中也更费劲。

5. 分析

本问题的顺序执行时间等于一次迭代时间乘以迭代次数。假定迭代次数为τ。这里有两层循环，其中一个嵌套在另一个中。外层循环有n次迭代，内层循环总共有n^2次迭代。每个内层循环包括一个乘和一个加，即2个计算步。外层循环在内层循环前有一个乘和减，而在内层循环后的有一个减和除，即共有4个计算步。因此，顺序计算时间由下式给定:

$$t_s = n(2n + 4)\tau$$

如果迭代次数为常数，则时间复杂性为O (n^2)。

如果我们假定所有处理器执行相同的迭代次数则并行执行时间即为一个处理器的执行时间。假设有n个方程和p个处理器，一个处理器计算n/p个未知数。一次迭代有一个计算阶段和一个广播通信阶段。

（1）*计算*　在计算阶段，内层循环有n次迭代，外层循环有n/p次迭代，两者均需执行与嵌套顺序循环相同的计算量。因而计算时间为:

$$t_{comp} = (n/p)\ (2n + 4)\tau$$

如果迭代次数为常数，时间复杂度为O (n^2/p)。随着p增加，计算时间减少。

（2）*通信*　通信在每次迭代计算完后进行，包括多个广播。实质上，由每个处理器计算出的n个值都应传给其他处理器。设有p个独立广播，每个包含n/p个数据项，每个数据项的发送要t_{data}个单位时间，那么总的通信时间为:

$$t_{comm} = p\ (t_{startup} + (n/p)\ t_{data})\tau = (pt_{startup} + nt_{data})\tau$$

当n给定时，通信时间是p的线性增长函数。

（3）*总的执行时间*　总的并行执行时间由下式给定:

179

$$t_p = ((n/p)\ (2n + 4) + pt_{startup} + nt_{data})\tau$$

当$t_{startup}$的时间不可忽略时，总的并行执行时间由p (t_{comp})的一个递减函数和p (t_{comm})的一个递增函数组成，故总的执行时间存在一个最小值。这个性质对大多数划分问题都成立。最小值可通过求导得到。举个具体例子，设$t_{startup} = 10000$且$t_{data} = 50$（这两个值在实际系统中具有代表性，参见第2章）。图6-11示出了n/p是整数情况下计算出的总执行时间、计算时间和通信时间。当$p = 16$时执行时间最小。

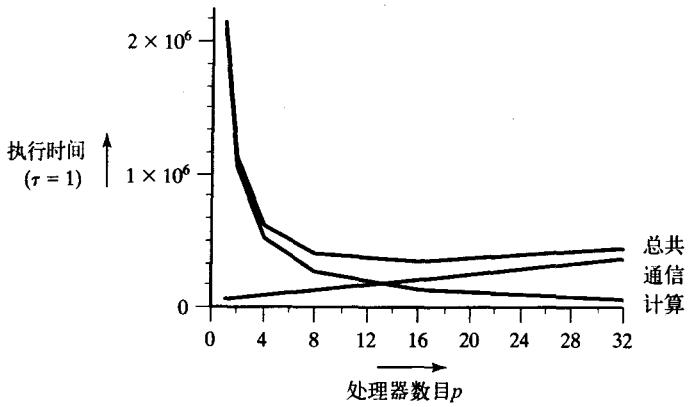


图6-11　雅可比迭代的计算时间和通信时间

（4）*加速比系数*　加速比系数由下式给定:

$$\text{加速比系数} = \frac{t_s}{t_p} = \frac{n(2n+4)}{(n/p)(2n+4) + pt_{\text{satrtup}} + nt_{\text{data}}}$$

如果通信时间可忽略时则加速系数为p。另一方面依赖于t_{startup}和t_{data}的值我们已为此问题推导出了一个优化的处理器数。

（5）计算/通信比 计算/通信比由下式给定:

$$\text{计算/通信比} = \frac{t_{\text{comp}}}{t_{\text{comm}}} = \frac{(n/p)(2n+4)}{pt_{\text{startup}} + nt_{\text{data}}}$$

该式表示较大的n（可扩展）值可改善计算/通信比。

6.3.2 热分布问题

前面的问题需要全局同步。现在我们来考虑一个局部同步的问题。有一块正方形金属薄板，每个边上的温度已知，内部表面上任一点的温度依赖于它周围的温度。为求得表面温度 180
分布，我们可以把表面区域划分成细小的网格点$h_{i,j}$。内部网格点的温度可认为是它的四个相邻网格点的温度的均值，如图6-12所示。为了这一计算，通过一系列与内点相邻的点来描述边是很方便的。那么当$h_{i,j}$的$0<i<n$，$0<j<n$时，内点共有$(n-1)\times(n-1)$个。而当$i=0$、$i=n$、$j=0$或$j=n$时，为边界点。边界点有与边的固定温度相对应的固定值。因此$h_{i,j}$的完整范围是$0 \leqslant i \leqslant n$、$0 \leqslant j \leqslant n$且共有$(n+1)\times(n+1)$个点，通过迭代下面这个方程:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4} \quad (0<i<n, 0<j<n)$$

对于固定迭代次数或直到一个点迭代误差达到约定精度，我们可以计算出每个点的温度。

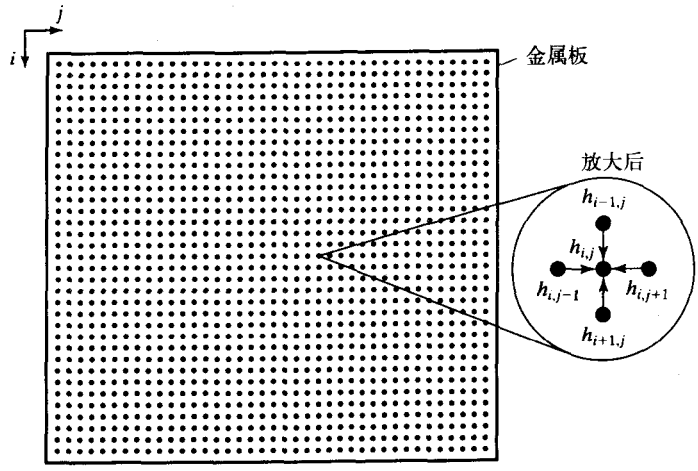


图6-12 热分布问题

这个迭代方程也用于其他类似问题中，如压力和电压问题。在科学和工程计算解决重要的问题时有更复杂的迭代方程。事实上我们是在求解线性方程组系统。每个点是一个未知数，只依赖于其他几个未知数，而不是像一般化情况下依赖于全部其他的未知数。为表明这种关系，考虑以自然顺序编号点的阵列，从左上角开始，以m个点为一行，序号按行递增，如图6-13所示。为书写方便从1开始编号，并包括那些代表边的点。（注意，这里$m=n$。m用来区分编号方法。）每个点使用如下方程:

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-m} + x_{i+m}}{4}$$

181

这可以写成包含未知数x_{i-m}、x_{i-1}、x_{i+1}、x_{i+m}的线性方程形式：

$$x_{i-m} + x_{i-1} - 4x_i + x_{i+1} + x_{i+m} = 0$$

其中$0 \leqslant i \leqslant m^2$，即有$m^2$个方程。这种方法也叫有限差分方法，可以推广至三维情况，此时要取六个相邻点的平均值，每一维取二个。有限差分法也可用于求解拉普拉斯方程，它与线性方程的关系将在第11章做更深入地讨论。（在第11章中用n表示方程数。）

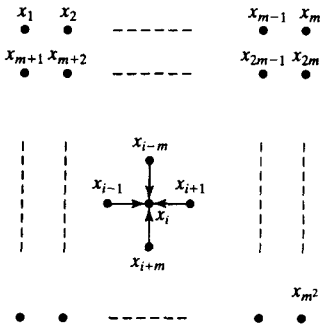


图6-13　热分布问题网络点的自然序编号

1. 顺序代码

下面我们回到点的初始数值系统，假定每点的温度用数组h[i][j]表示，边界点h[0][x]、h[x][0]、h[n][x]和h[x][n]($0 \leqslant x \leqslant n$)已经用边温度初始化。该计算的顺序代码如下：

```
for (iteration = 0; iteration < limit; iteration++) {
  for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
      g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
  for (i = 1; i < n; i++)                    /* update points */
    for (j = 1; j < n; j++)
      h[i][j] = g[i][j];
}
```

使用固定次数的迭代。点的新值的计算中之所以乘0.25而不是除以4，是因为乘法通常比除法效率更高。顺序编码中提高效率的常用方法同样适用于并行编码，只要有可能就应使用这些优化方法。（当然，一个好的优化编译器可以帮助完成这些转换。）

假如我们想在达到某个精度时停止迭代，有几种方法，但在所有情况中，所有的点必须达到它们的精度要求。经过对数组适当的初始化，顺序编码可能如下：

```
do {
  for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
      g[i][j] = 0.25*(h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
    for (i = 1; i < n; i++)                  /* update points */
    for (j = 1; j < n; j++)
182   h[i][j] = g[i][j];

  continue = FALSE;                  /* indicates whether to continue */
  for (i = 1; i < n; i++)            /* check each point for convergence */
    for (j = 1; j < n; j++)
      if (!converged(i,j) {      /* point found not converged */
        continue = TRUE;
        break;
      }

} while (continue == TRUE);
```

该代码表明了在所有点都计算过之后检验收敛性，可以使用不同的可能的收敛算法。如果元素g[i][j]已收敛到要求的精度，例程converged(i,j)就返回值TRUE；反之，它返回值FALSE。如果一次迭代中至少有一点没有收敛，布尔型标记continue就会被设置为TRUE。通常，我们希望确保循环即使在不发生收敛时也能终止。这可以通过加入一个循环计数器来实现。

2. 改进

在上面的代码中，需要另一个数组g[ ][ ]来存放从旧值计算得到的点的新值。在g[ ][ ]中的所有值被计算后就用保存在g[ ][ ]中的新值更新数组h[ ][ ]。使用一次迭代的值以计算下一次的迭代值的这种方式称为雅可比迭代。一个显著的改进是取消第二个数组:

```
for (iteration = 0; iteration < limit; iteration++) {
  for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
      h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
```

对于给定计算的顺序次序，该迭代中用来计算h[i][j]的两个值(h[i-1][j]和h[i][j-1])已经计算出并被使用，而在该迭代中还未计算出的另两个值(h[i+1][j]和h[i][j+1])将从前面的迭代中加以计算。如此，我们所使用的是最近的可用值。这就是所谓的高斯-塞得尔迭代，它通常可生成一个更快的收敛。但是它依赖于计算的顺序次序。

在高斯-塞得尔方法中对未知数（点）的计算，如x_i $(0 < i < n-1)$，是按序进行的，所以在当前的未知数x_j $(j < i)$ 之前的那些未知数已经计算完毕并加以使用，而那些在当前点之后的未知数x_k $(k > i)$在当前的迭代中还未计算出，所以就使用在前一迭代中计算得到的值。基本的高斯-塞得尔方法能很好地与未知数按某种顺序次序进行计算的顺序程序相匹配的，但如前所述，对于未知数的计算是同时进行的并行程序而言它并不是一个好的基础。然而存在某些特定的次序允许进行同时计算。我们将在本章的末尾探讨适合并行化的不同的次序和更快的迭代方法，而更多的细节将在第11章中叙述。现在我们将集中讨论雅可比迭代因为它允许我们研究局部同步，但我们应明白通常雅可比迭代既不是最好的顺序算法也不是并行算法的最好基础。 183

3. 并行代码

顺序代码显得“不太自然”，因为我们使用了for循环来访问各点，实际上这些点可以同时访问而不会引起算法的任何变动:

```
for (iteration = 0; iteration < limit; iteration++) {
  forall (i = 1; i < n; i++)
    forall (j = 1; j < n; j++)
      h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
}
```

在这个结构中，计算（等式）右边的所有值都是从前一次迭代计算出来的，而不需要任何显式的数组g[ ][ ]。我们通常会对问题进行划分，使得每个进程处理多个点。尽管如此，我们在第一个实例中仍假定一个进程对应一点。

每个进程需要4个邻近点。按自然对应的方式将进程安排进网格是最方便的组织方法，也就是说，如果我们用行主序的网格下标来表示进程，其中，第一个下标是行，第二个下标是列。对于固定的迭代次数，对w，x，y和z进行了适当的初始化后，进程P_{ij}（边界点除外）的形式如下:

```
for (iteration = 0; iteration < limit; iteration++) {
  g = 0.25 * (w + x + y + z);
  send(&g, P_{i-1,j});          /* non-blocking sends */
  send(&g, P_{i+1,j});
  send(&g, P_{i,j-1});
  send(&g, P_{i,j+1});
  recv(&w, P_{i-1,j});          /* synchronous receives */
  recv(&x, P_{i+1,j});
  recv(&y, P_{i,j-1});
  recv(&z, P_{i,j+1});
}
```

局部障栅

每个进程有自己的迭代循环。迭代次数必须发送到每个进程。在等待recv()时使用不阻塞的send()很重要；如果不这样，进程可能死锁，在继续进行之前每个进程都在等待recv()。recv()必须同步，并且等待send()。每个进程将通过recv()同其4个邻居同步。这里，我们使用的是一种局部同步技术。没有必要为更新数组使用单独的迭代。4个过程间的消息传递如图6-14所示。

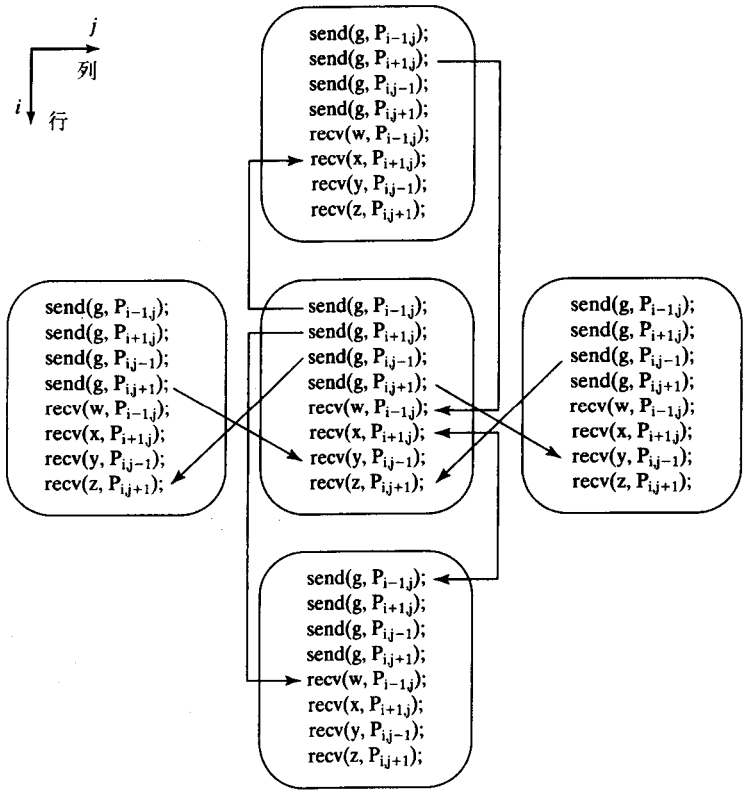


图6-14　热分布问题的消息传递

184　如果进程在达到要求的精度时就可以停止，那么需要一个主进程来接收所有（从）进程结束的通知。一个进程可以在（本地）达到精度时向主进程发送数据，例如：

```
iteration = 0;
do {
   iteration++;
   g = 0.25 * (w + x + y + z);
   send(&g, P_{i-1,j});                    /* locally blocking sends */
   send(&g, P_{i+1,j});
   send(&g, P_{i,j-1});
   send(&g, P_{i,j+1});
   recv(&w, P_{i-1,j});                    /* locally blocking receives */
   recv(&x, P_{i+1,j});
   recv(&y, P_{i,j-1});
185 recv(&z, P_{i,j+1});
} while((!converged(i, j)) && (iteration < limit));
send(&g, &i, &j, &iteration, P_master);
```

对于在边界操作的进程，我们可以用进程ID来决定进程在数组中的位置，从而有如下代码：

```
if (last_row) w = bottom_value;
if (first_row) x = top_value;
if (first_column) y = left_value;
if (last_column) z = right_value;
iteration = 0;
do {
   iteration++;
   g = 0.25 * (w + x + y + z);
   if !(first_row) send(&g, P_{i-1,j});
   if !(last_row) send(&g, P_{i+1,j});
   if !(first_column) send(&g, P_{i,j-1});
   if !(last_column) send(&g, P_{i,j+1});
   if !(last_row) recv(&w, P_{i-1,j});
   if !(first_row) recv(&x, P_{i+1,j});
   if !(first_column) recv(&y, P_{i,j-1});
   if !(last_column) recv(&z, P_{i,j+1});
} while((!converged) && (iteration < limit));
send(&g, &i, &j, iteration, P_master);
```

将进程标号转换为实际数字对我们来说是一件很简单的事情。假定进程从网格的左上角开始按行（自然排序）编号，进程P_i就需同P_{i-1}（左）、P_{i+1}（右）、P_{i-k}（上）和P_{i+k}（下）$(0 \leqslant i < k^2)$进行通信。

4. 划分

显然，我们一般会给每个处理器分配多个点，因为处理器数往往比点数少得多。可将点的网络分割成方块或条（列），如图6-15所示（p分割）。划分方法与第3章的图像位图划分方法一致，不过现在划分间存在通信关系。所以我们要使通信最小化。对于n^2个点、p个处理器且块大小相等，每个划分有n^2/p个点。图6-16示出两种划分方法的划分间通信关系。

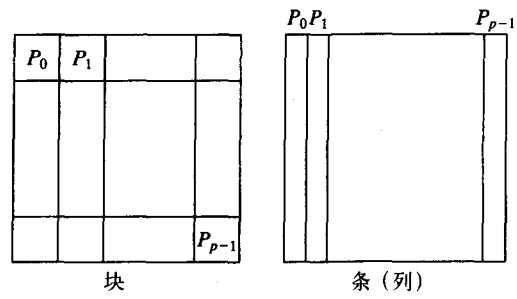


图6-15 热分布问题的带状划分

186

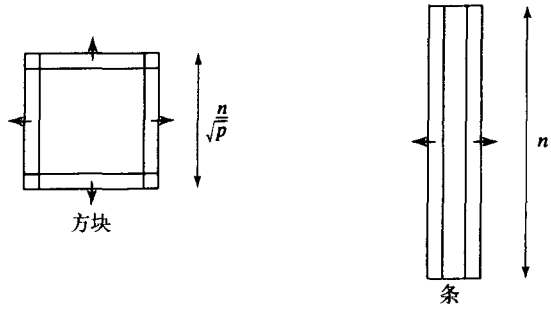


图6-16 划分后的通信关系

在块划分中，一个块要与四个相邻块交换四条边上的点的值，每个进程在每次迭代中发送四条消息并接收四条消息（假设一条边上的点的数据被打包成一条消息）。因而，通信时间为：

$$t_{\text{commsq}} = 8\left(t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}}\right)$$

这个方程成立的前提是至少有一块拥有四个相邻块，即$p \geq 9$。

在条划分中，一个块最多只与两个相邻块交换两条边上的点的数据。进程在每次迭代中发送两条消息和接收两条消息（同样设一条边上的所有点的数据被打包成一条消息）。因而，通信时间为:

$$t_{commcol} = 4(t_{startup} + nt_{data})$$

值得注意的是通信时间与处理器数目p无关，即与划分数无关。

这两种通信时间均将受通信启动时间影响。例如，设$t_{startup} = 10000$、$t_{data} = 50$(与图6-11中一样)且$n^2 = 1024$。在这种情况下条划分的通信时间是46400（个时间单位），不管p取何值。块划分的通信时间是$80000 + 12800/\sqrt{p}$（个时间单位）。不论处理器数目是多少，这个时间都要比条划分的通信时间长。但是，如果启动时间等于100，那么条划分的通信时间为6800，而块划分的通信时间为$800 + 12800/\sqrt{p}$。当$p > 4$时，条划分的通信时间比块划分的通信时间长。

一般来说，启动时间长时条划分较好，启动时间短时则块划分较好。对前面两个等式，如果:

187

$$8\left(t_{startup} + \frac{n}{\sqrt{p}}t_{data}\right) > 4(t_{startup} + nt_{data})$$

即

$$t_{startup} > n\left(1 - \frac{2}{\sqrt{p}}\right)t_{data}$$

则块划分的通信时间长于条划分的通信时间（$p \geq 9$）。当p缓慢增长时，用我们的假设数据求得1600，不等式右边趋于$\sqrt{nt_{data}}$。例如，当$t_{startup} = 1200$时，$p = 64$是条划分优还是块划分优的分界点。图6-17示出了$p = 9$、16、64、256和1024时的情况。

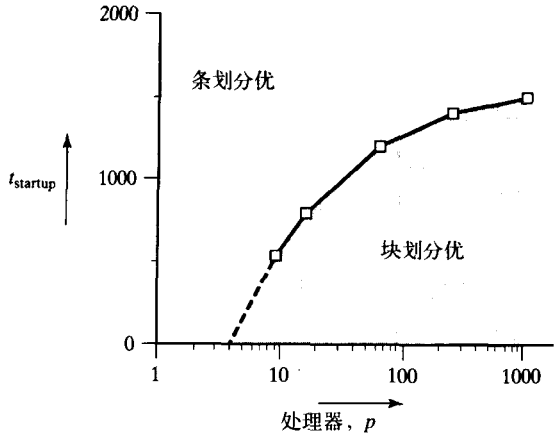


图6-17　启动时间与块划分和条划分的关系

对于大多数系统特别是工作站机群，启动时间都比较长。所以，看上去条划分方法是更合适的选择，因为消息只是左右传递，而不是考虑上下左右传递，所以对编程来讲它是最容易的划分。

5. 实现细节

我们必须在一个消息中传送整列数据点给相邻进程。当数组以行主序存储时（如在C语言中那样），为方便起见，我们可以将点的二维数组划分成行而非列。只要简单地指明行的起始地址和存储在其中（元素的邻近组）数据元素个数就能把一行数据在一个消息中传送出去。

如果我们不想考虑实际存储结构，可用一个单独的一维数组保存要传送到邻接进程的点。下面的讨论就假设划分按行而不是列，不过这一实现细节对算法本身没有影响。

除了将点划分成行外，图6-18还示出了在每条边上具有附加点的行的每个进程，附加称为*幻象点*（ghost point），附加点行用来保存来自邻接边的值。在相邻的每个进程中增加一个点数组是为了保存幻象行，幻象点的引入是为了方便编程。

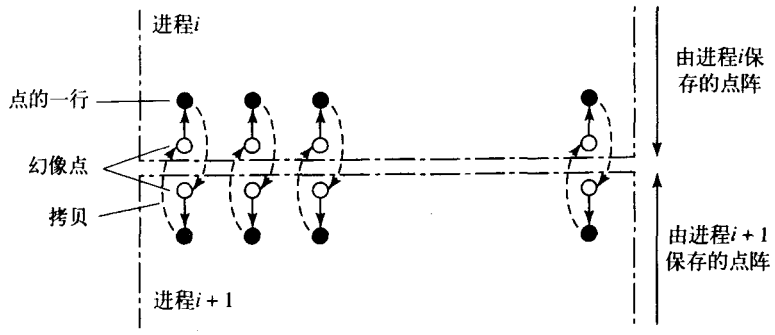


图6-18 每个进程中数组构成连续的行，包括幻象点

进程P_i（不包括边界上的进程）的程序代码形式如下:

```
for (k = 1; k <= n/p; k++)        /* compute points in partition */
  for (j = 1; j <= n; j++)
   g[k][j] = 0.25 * (h[k-1][j] + h[k+1][j] + h[k][j-1] + h[k][j+1]);
for (k = 1; k <= n/p; k++)        /* update points */
  for (j = 1; j <= n; j++)
   h[k][j] = g[k][j];
send(&g[1][1], n, P_{i-1});       /* send row to adjacent process */
send(&g[n/p][1], n, P_{i+1});
recv(&h[0][1], n, P_{i-1});       /* receive row from adjacent process */
recv(&h[n/p + 1][1], n, P_{i+1});
```

188

安全与死锁 上面所有的代码中，所有进程都先发送它们的消息然后才开始接收它们的消息，从MPI的领域来看，这样是不安全的。这是因为程序的正确性依赖于`send()`的缓冲。MPI中不指明缓冲容量。如果一个发送例程在调用时没有足够的可用存储区，那么这个例程就应延时，直到有足够的存储区或消息绕过缓冲发送出去。因而，本地阻塞的`send()`在行为上就相当于一个同步的`send()`，仅当匹配的`recv()`执行时才返回。如果所有的`send()`都是同步的，那么匹配的`recv()`可能永远无法执行，这种情况就会导致死锁。对于我们的问题，如果n/p相对较小，那么有足够的存储空间可用——你可能会产生疑问：为什么系统要提供一个不安全的本地阻塞型发送消息的例程呢?

改造成安全代码的一种方法是在相邻进程中调换`send()`和`recv()`的使用次序，使得仅有一个进程先执行`send()`。那么即使是同步的`send()`例程也不会导致死锁。事实上，一种测试程序安全性的好办法是把程序中的消息传递例程替换成同步的版本。

调换了`send()`和`recv()`次序的安全的代码如下:

```
if ((i % 2) == 0) {                    /* even-numbered processes */
   send(&g[1][1], n, P_{i-1});
   recv(&h[0][1], n, P_{i-1});
   send(&g[n/p][1], n, P_{i+1});
   recv(&h[n/p + 1][1], n, P_{i+1});
```

```
} else {                                        /* odd-numbered processes */
   recv(&h[n/p + 1][1], n, P_{i+1});
189  send(&g[n/p][1], n, P_{i+1});
   recv(&h[0][1], n, P_{i-1});
   send(&g[1][1], n, P_{i-1});
}
```

本例中调换send()和recv()的使用次序容易做到，其他情况下可能要困难一些。

MPI提供了安全通信的几种不同方法：

- 复合发送接收例程：MPI_Sendrecv()（此例程保证不会死锁）。
- 缓冲发送例程：MPI_Bsend()——由用户显式提供存储区空间。
- 非阻塞例程：MPI_Isend()和MPI_Irecv()——例程立即返回，另外用一个独立的例程来测试消息是否已被接收（MPI_Wait()、MPI_Waitall()、MPI_Waitany()、MPI_Test()、MPI_Testall()或者MPI_Testany()）。

使用第三种方法的伪代码段如下：

```
isend(&g[1][1], n, P_{i-1});
isend(&g[n/p][1], n, P_{i+1});
irecv(&h[0][1], n, P_{i-1});
irecv(&h[n/p + 1][1], n, P_{i+1});
waitall(4);
```

本质上来说，等待例程成为一个障栅，等待所有的消息传递例程完成。

6.3.3 细胞自动机

我们在此要提及一个特别适合于同步迭代的概念，*细胞自动机*（cellular automaton）。在这个方法中，问题空间首先被分为很多胞元。每个胞元总是处于有限个状态中的一个。按某种规则，胞元受它邻居的影响，并且“一代”中的所有胞元同时受影响。这些规则被重新应用到子代中，一代一代地，胞元进行演化即改变状态。

最著名的细胞自动机是剑桥数学家John Horton Conway设计的“生命游戏”，由Gardner出版[Gardner，1967]。Gardner指出细胞自动机的概念可以追溯到20世纪50年代早期冯·诺依曼的研究工作。生命游戏是一种木板游戏；木板由一个（理论上是无限的）二维的胞元数组组成。每一个胞元中可能有一个有机体，包括对角线邻接的胞元它有8个相邻胞元。最初一些胞元处于同一种模式，然后应用下列规则：

1）有2个或3个邻居的有机体可以幸存到下一代。

190　2）有4个或更多邻居的有机体由于过多而死亡。

3）有1个或没有邻居的有机体由于隔离而死亡。

4）与3个非空邻居邻接的空胞元可以产生一个有机体。

这些规则是由Conway从“长期的实验中”推导出来的。

另外一个简单有趣的细胞自动机的例子是海里的“鲨鱼和小鱼”，它们中的每一个都有自己的行为规则。在[Fox et al.，1988]中研究过这个问题的一个二维版本。海洋可以看作是一个三维的胞元数组。每一个胞元可能有一条鱼或一条鲨鱼（但不能拥有二者）。鱼遵循下列规则移动：

1）若有1个相邻的空胞元，鱼移动到这个胞元。

2）若有2个或多个相邻的空胞元，鱼随机移动到其中的一个中。

3）若没有相邻的空胞元，鱼呆在原处不动。

4）若鱼移动并且到了繁殖期，它就产下一条小鱼，小鱼呆在清空的胞元中。

5）鱼经历x代后死去。

鲨鱼遵循下列规则移动:

1）若有1个相邻的胞元被鱼占据，那么鲨鱼移动到这个胞元并吃掉这条鱼。

2）若有2个或多个相邻的胞元被鱼占据，那么鲨鱼随机移动到其中一个胞元并吃掉这条鱼。

3）若邻近胞元中没有鱼，鲨鱼按鱼移动相同的方式移动到一个未被占领的相邻胞元。

4）若鲨鱼移动并且到了繁殖期，它就产下一条小鲨鱼，小鲨鱼呆在清空的胞元中。

5）若鲨鱼经历y代后还未吃到鱼，那么它就死去。

习题6-21描述了一个类似的狐狸和兔子的问题。兔子的行为是欢快地在周围跳动，而狐狸的行为则是若遇到兔子就吃掉它。

有许多细胞自动机的重要应用，可以避免求解微分方程。例如，对于给定流体/气体动力学的规则，流体和气体绕物体的流动或气体扩散就能够用这种方法建模。生物的成长也能用这种方法建模。在习题中给出的例子还包括穿过机翼的空气流动（习题6-24）和海滩或河岸的沙子的流动/侵蚀（习题6-23）。除了这些，当然还有许多其他细胞自动机的应用（习题6-22）。

6.4 部分同步方法

很显然，同步导致性能的明显降低。在本章的最后这小节中，我们将探讨如何减少在前面讨论过的同步迭代问题中的同步开销。我们以热分布问题作为例子。其并行代码如下: 191

```
for (iteration = 0; iteration < limit; iteration++) {
  forall (i = 1; i < n; i++)
    forall (j = 1; j < n; j++)
      h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
}
```

其中假设计算表达式的右边各值是由前一迭代计算得到的。因此代码是基于前一迭代值来计算下一迭代值的。这是传统的雅可比迭代方法，但它需要一个进程的全局同步点（障栅）以等待直至所有进程都完成了它们的计算。是完成障栅同步所需时间而不是计算时间导致了性能的降低。我们在前面已提及在计算中使用某些当前迭代值的可能性，如在高斯-塞德尔迭代方法中，但因为所有进程在同一迭代中一起操作，所以仍需要一个障栅来获得同步。如果将该障栅取消，则将允许某些已完成本次迭代的进程在其他进程未完成本次迭代之前继续执行后面的迭代，这将导致这些向前执行的进程使用不仅仅是前一次迭代所得到的值，而可能是前一次甚至前几次迭代所得到的值。该方法被称为*异步迭代法*（asynchronous iterative）。在异步迭代中，收敛的数学条件可能更为严格，即除非某些数学条件存在，否则计算可能不收敛。要使该方法收敛，就不允许每个进程使用以前的迭代值。[Chazan and Miranker，1969]年引入了一种称为*无序松弛*（chaotic relaxation）的异步迭代方法，其收敛条件为如下:

“必存在一个固定正整数s使得在求第i次迭代时，进程不能使用第j次迭代中形成的任何值，如果$j<i-s$”[Baudet，1978]。

该条件指出对并行代码的一个简单变化就允许进程继续执行直到进程试图使用s次迭代前的值。每个已存储的值将需要一个“时间戳”（time stamp），即与该存储值相关的迭代号。

检查每次迭代是否收敛的最后部分代码也可省去。如果在进行收敛检查前允许继续执行若干次迭代则更好。将无序松弛和延迟收敛检查两者结合起来就可允许每个进程在同步前完

成s次迭代，而且当进程向前执行时仍可更新其本地所存储的值。在完成每s次迭代后记录最大的偏离值，接着就进行收敛检查。在任何时候，相应于实际迭代正在使用的数组元素，可能来自较早的迭代，但不会超过前s次迭代。在消息传递的方案中，所有从其他进程得到的数据值将来自进程间最近一次通信的前s次迭代。在进程中的计算所用到的数据值来自当前迭代或是以前的迭代，这取决于顺序计算得到的数据值的次序。

应注意的是，不能将以上叙述方法应用到所有的同步问题中，例如胞元自动机操作就不
192 适合这种方法。

6.5 小结

本章介绍了以下内容:

- 障栅的概念及其实现（包括全局和局部障栅）
- 数据并行计算
- 同步迭代概念
- 使用全局障栅和局部障栅的实例
- 安全通信的概念
- 细胞自动机
- 部分同步方法

推荐读物

在大多数并行编程的教科书中都讨论过障栅的概念。除了这里讨论的障栅的软件实现，一些多处理机系统（例如CRAY T3D）对障栅提供了硬件上的支持。[Pacheco，1997]开发了雅可比迭代的MPI代码，在[Snir et al.，1996]也用MPI代码实现了雅可比迭代，并着重讨论了安全程序及利用MPI的不同特性如何编写可替代代码，如`MPI_Sendrecv()`例程、张贴例程和空进程。雅可比迭代的MPI代码的细节也可在其他MPI“参考文献”中见到，[Gropp，Lusk，and Skjellum，1999]中也包括MPI特性的使用，如拓扑。关于计算与通信的平衡的讨论可在[Snir et al.，1996]和[Wilson，1995]中找到。除了本章讨论的全局同步技术，还有不严格结构化的同步方式或松散的同步方式，此时进程间或地进行同步。一些松散同步技术的应用在[Fox，Williams，and Messina，1994]中有详细的讨论。

由于[Chazan and Miranker，1969]的介绍，其他几位作者包括[Baudet，1978] 和[Evans and Yousif，1992]已经对无序松弛进行了研究，尽管在过去的并行程序设计和算法的教科书中忽略了这一点。当无序松弛可以应用时，比起全同步方法来在执行速度方面它有很大的改进潜能。

参考文献

BAUDET, G. M. (1978), Asynchronous Iterative Methods for Multiprocessors, *J. ACM*, Vol. 25, pp 226–244.

BERTSEKAS, D. P., AND J. N. TSITSIKLIS (1989), *Parallel and Distributed Computation Numerical Methods*. Prentice Hall, Englewood Cliffs, NJ.

CHAZAN, D, AND W. MIRANKER (1969), Chaotic Relaxation, *Linear Algebra and Its Applications*, Vol 2, pp. 199–222.

EVANS, D. J, AND N. Y. YOUSIF (1992), "Asynchronous Parallel Algorithms for Linear Equations," in *Parallel Processing in Computational Mechanics* (editor H. Adeli), Marcel Dekker, NY, pp. 69–130.

FOX, G. C., M. A. JOHNSON, G. A. LYZENGA, S. W. OTTO, J. K. SALMON, AND D. W. WALKER (1988), *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, Englewood Cliffs, NJ.

FOX, G. C., R. D. WILLIAMS, AND P. C. MESSINA (1994), *Parallel Computing Works*, Morgan Kaufmann, San Francisco, CA.

GARDNER, M. (1967), "Mathematical Games," *Scientific American*, October, pp. 120–123.

GROPP, W., E. LUSK, AND A. SKJELLUM (1999), *Using MPI Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA.

HILLIS, W. D., AND G. L. STEELE, JR. (1986), "Data Parallel Algorithms," *Comm. ACM*, Vol. 29, No. 12, pp. 1170–1183.

JÁJÁ, J. (1992), *An Introduction to Parallel Algorithms*, Addison Wesley, Reading, MA.

PACHECO, P. (1997), *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, CA.

SNIR, M., S. W. OTTO, S. HUSS-LEDERMAN, D. W. WALKER, AND J. DONGARRA (1996), *MPI: The Complete Reference*, MIT Press, Cambridge, MA.

WAGNER, R. A., AND Y. HAN (1986), "Parallel Algorithms for Bucket Sorting and the Data Dependent Prefix Problem," *Proc. 1986 Int. Conf. Par. Proc.*, IEEE CS Press, pp. 924–929.

WILSON, G. V. (1995), *Practical Parallel Programming*, MIT Press, Cambridge, MA.

习题

科学/数值习题

6-1 实现并测试图6-4中的计数器障栅。是否有必要为发送和接收使用阻塞或同步的例程？请解释原因。

6-2 编写一个障栅，`barrier(procNum)`，这个障栅将阻塞直到`procNum`个进程到达障栅，然后释放这些进程。允许障栅以不同数目的进程和以不同的`procNum`值被调用。

6-3 分析下述代码中障栅的时间开销：

```
t1 = time();
Barrier(group);
t2 = time();
printf("Elapsed time = %", difftime(t2, t1));
```

（MPI中的障栅例程是`MPI_Barrier(Communicator)`，时间例程是`MPI_Wtime()`。）要求考虑进程数不同的情况。

6-4 用6.1.3节介绍的树结构，编写代码以实现8个进程的障栅，并与其他障栅调用（如MPI的`MPI_Barrier()`）进行比较。 193 ~ 194

6-5 实现6.1.4节中介绍的蝶形障栅，并与其他障栅调用进行比较。

6-6 实验证明你的系统使用非阻塞发送例程时在何点达到缓冲的限额。如果要求超过有效容量的缓冲，有什么后果？（可用缓冲的容量可能与用于其他用途的存储器容量有关。）

6-7 不具有可交换性的操作如除法是否可用于图6-8的前缀运算？

6-8 计算图6-8的前缀运算的效率。

6-9 给定一个边长分别为x和y的长方形区域，通信开销与周长$2(x + y)$成比例，证明当$x = y$（即成正方形）时通信开销最小。

6-10 编写一个并行程序求解基于下面的有限差分方程的一维问题

$$x_i = \frac{x_{i-1} + x_{i+1}}{2}$$

其中$0 < i < 1000$，并设$x_0 = 10$，$x_{1000} = 250$。

6-11　在6.3.2节的热分布问题中我们假设了一个正方形阵列。如果这是个长有n个点宽有m个点的列阵，那么选择块划分或条划分的数学条件是什么？

6-12　分别用6.3.2节中介绍过的不同终止方法的收敛精度分析热分布问题。根据点的现值与下一次值的差值来判断终止条件是否合适？或者是否有必要使用一个更复杂的终止条件？这里要分析的基本问题是“每个点的计算值已在其前一次计算值的1%（譬如说）之内时，解的精度是多少？”

6-13　编写一个并行程序模拟6.3.3节描述的生命游戏，用实验说明使用不同初始生物总数时的结果。

6-14　实验比较在6.3.2节中介绍过的热分布问题的全局同步和部分同步的实现。对6.4节中说明的收敛条件试用不同的s值。书写一个有关报告叙述你的发现包括你所获得的特定的速度改进。

现实生活习题

6-15　图6-19表示一个房间，有四堵墙和一个壁炉。墙的温度是20°C，壁炉的温度是100°C。编写一个并行程序用雅可比迭代法计算房间里的温度并以10°C为温度间隔，用Xlib库或其他图形库的绘图例程绘出温度分布曲线（彩色更好），运行时间也要显示出来。（这道程序设计题可以在曼德勃罗特问题之后做，因为它可以使用同样的图形例程调用。）

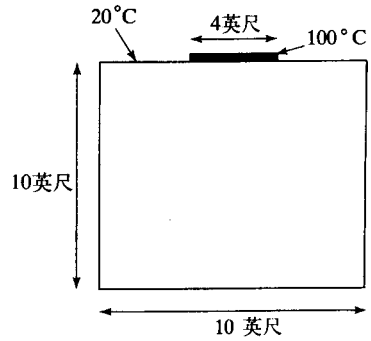


图6-19　（习题6-15）房间

6-16　重做习题6-15，但现在是一个直径为20英尺[⊖]的圆形房间，中心有一个100°C的点热源，墙的温度是20°C。

6-17　模拟一个红绿灯控制的十字路口，如图6-20所示。车辆从四个方向驶过来，要么穿过十字路口直走，要么左转弯要么右转弯。平均有70%的车辆直走，10%的右转弯，20%的左转弯。每辆车以同样速度到达路口。用细胞自动机方法设计一套行驶规则解决这个问题，编写并行程序实现它们并用你自己的测试数据（如车辆数和位置）来测试它。

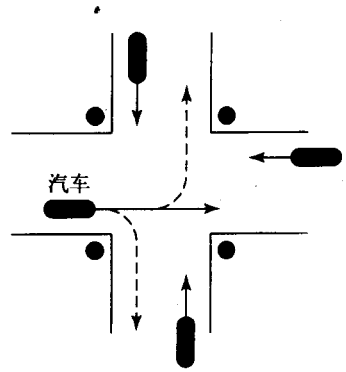


图6-20　（习题6-17）十字路口

6-18　编写一个并行程序模拟6.3.3节中描述的鲨鱼和小鱼的行动。输入参数包括海洋面积、鲨鱼和小鱼的数量、鲨鱼和小鱼在海洋中的初始位置、繁殖年龄以及鲨鱼耐饥饿时间。对角相邻的胞元不作为邻接胞元。所以一个胞元（边上胞元除外）共有六个邻接胞元。每一代鲨鱼和小鱼的年龄以1递增。修改模拟程序考虑水流的因素。

6-19　Michaels博士以心不在焉闻名校园。因此，他去Uwharrie国家森林公园野营旅行时，毫不奇怪他会忘

⊖　1英尺 = 0.305m

记带地图和指南针。幸运的是，他带了一个具有新型蜂窝调制解调器的笔记本计算机。更幸运的是，他让你留在计算机科学大楼做研究项目！

你对他的这次旅行有种预感，所以你已下载了这片森林的详细地图，这些最新的NASA卫星图像表示了每棵树、每处悬崖、每条路的位置。数据以“胞元”阵列的格式存储，胞元是边长为0.3米的正方形。每个胞元代表的森林区域的特征用一个字母“T”、“C”、“O”或“R”表示：

“T”这个区域有一棵不能通过的树；

“C”这个区域有一处陡峭的绝壁（悬崖）；

“O”这个区域是块空地，教授能通过；

“R”这个区域是条道路或是标有记号的小路，教授能通过。 196

所以，你对Michaels博士发来电子邮件请求你的帮助并不感到奇怪。由于医疗条件的原因他越早离开森林越好。他要求你编一个程序以完成以下两件事情：

1）识别他现在在森林中的位置。

2）指导他离开森林到森林南边界的路上去。

你的程序可以询问教授他现在所处“小胞元”的前后左右都有什么东西。作为每次询问的响应，他发回四个字母。例如，Query( )的结果可能是“T”、“O”、“O”、“C”，说明他的前面有棵大树挡路，可以向后或向左转，他的右边有一处通不过的绝壁。这些信息暗示他处于一个包含“O”或“R”的胞元中。

你的程序可以通过发送“F”（向前移一个胞元）、“B”（向后移一个胞元）、“L”（向左移一个胞元）或“R”（向右移一个胞元）告诉教授向哪个方向移一个胞元。语法上，Move（“L”）表示向左移一个胞元，其他以此类推。记住，如果你告诉他移到一个有树的胞元中，而最后不得不返回时，你的成绩会被扣减；如果你竟然把教授引到一个包含悬崖的胞元去，你除了受良心谴责外，成绩报告单上也会多一个“F”（教授的亲戚会把这些都记入档案）。

你的程序要用尽可能少的询问/移动来识别出教授的位置并指引教授通过最短的路线到沿着森林南边界的路上去。

Query( )和Move( )函数的原型定义如下：

```
char * Query(void)
/* Query returns a pointer to a string of four characters */

int Move(char direction);
/* if the move is successful, Move returns the value 0. If it is unsuccessful
because you directed him into a tree, Move returns a -1. If Move is
unsuccessful because you directed the professor off a cliff, Move returns a
-100 indicating you just failed your research project work and need to call
the coroner. */
```

提示：教授可能面朝东南西北任一个方向，且没有指南针。因此，你须将他对Query( )的回答的模式与你的地图数据在四个可能方向上进行匹配，缩小他所在位置的可能范围。然后你指导他移动（Move( )），然后再次询问（Query( )）。当你最终识别出他的位置时，你须找出走出森林到南边界路上的最短路线。

6-20 Eric对昨晚看的延迟像带神秘侦探片“谁下的手？”的最后一个片断着了迷，侦探Sam Shovel对几种笔迹样本进行模式匹配的天才是解开谜团的关键。Eric决定写一个简单的程序来模仿Sam的模式匹配行为。Eric做的第一件事是在15×21格子上创建26个“完美”

的印刷体字母，作为与手写样本匹配的模板，这些模板然后逐一与实际的手写样本加以比较，从中推导出实际的手写字母。他的第一次试验竟是一败涂地！很快他发现失败原因是没有一个手写字母能与他的“完美”字母完全匹配，结果当然是半个嫌疑犯也识别不出。

他决定试试三种截然不同的方法。第一种方法是使用一个流水线系统：将待辨识
197 字母缩放到标准大小，放于格子中心，定出它的对称轴后把它旋转到正常方向，然后与“完美”字母表中字母比较。第二种是应用数学方法，对嫌疑犯歪歪扭扭的字母进行过滤，并将其平滑以去除噪声，再将它进行数学变换，将变换结果与“完美”字母的同样变换的结果进行匹配。第三种方法中，Eric决定进一步简化问题，根据15 × 21格子上待辨识字母与“完美”字母匹配的格子数目来决定匹配结果。他在格子上移动待辨识字母，试图得到与“完美”字母的最佳匹配，记下匹配的格子数目作为本次匹配结果。对余下的25个字母，重复以上匹配过程。最后，选取有最多匹配格子数的字母作为识别结果。

简单分析Eric的匹配方法，你认为哪一种最适于并行处理？

6-21 从前有一个岛，岛上仅有野兔、狐狸和植被。此岛形如标准的棋盘。当地几个地理学者在它上面均匀地画了一些线，把岛划分成64个方块，以便于研究岛上居住者的数量分布。

每个方块中野兔和狐狸的数量取决于以下几个因素：

- 每个方块上这天开始时野兔和狐狸的数量。
- 这天野兔和狐狸的繁殖速率（整个岛一样）。
- 植被生长速率。
- 这天老野兔和老狐狸的死亡速率。
- 狐狸吃野兔的特性（狐狸完全靠吃野兔生存，植被丰茂处野兔很难被狐狸发现）。
- 野兔吃植被的特性（野兔靠吃植被生存，一个方块中野兔太多会导致挨饿和/或繁殖率降低和/或易于成为狐狸的腹中之物）。
- 野兔从一个方块向邻接方块迁移（日复一日）的特征。
- 狐狸从一个方块向其他任何一个距离小于“两跳”的方块迁移的特征。

由于这是一个岛，所以有某些边界条件：靠水的28个方块毫无迁移的可能，野兔和狐狸都不能跳下水或从水中跳上来。类似地还有一些初始条件：野兔和狐狸的初始数量，在程序开始运行时野兔和狐狸在各个块中的数量分布。

你的任务是模拟岛上10年的生活，以1天为时间单位，算出10年后岛上的每个小方块中野兔和狐狸的数量。小方块中一对野兔在繁殖日开始时生下一窝小兔，繁殖日以9个星期为周期。小兔的数量从2只到9只不等，取决于食物（植被）丰足程度和那天开始时方块中野兔的数量（“野兔密度”）。它们之间的关系参见表6-1。小方块中一对狐狸在繁殖日开始时生下一窝小狐狸，繁殖日以6个月为周期。小狐狸的数量从0到5只不等，取决于食物（野兔数目）丰足程度和那天开始时方块中狐狸的数量（“狐狸密度”），如表6-2所示。

表6-1 小野兔出生数量（习题6-20）

在一天开始时的植被	在一天开始时野兔的数量				
	<2	2至200	201至700	701至5000	>5000
< 0.2	0	3	3	2	2
⩾0.2 且 < 0.5	0	4	4	3	3
⩾0.5 且 < 0.8	0	6	5	4	4
⩾0.8	0	9	8	7	5

表6-2 野兔和狐狸数量（习题6-20）

在一天开始时野兔的数量（每只狐狸）	在一天开始时狐狸的数量				
	<2	2至10	11至50	51至100	> 100
<3.0	0	2	2	1	0
⩾3.0 且 <10	0	3	3	2	1
⩾10 且 <40	0	4	3	3	2
⩾40	0	5	4	3	3

一只狐狸一周中吃到两只野兔就能存活，只要能找到野兔，它最多能吃四只。如植被茂密度小于0.6，野兔就容易被狐狸发现，这种情况下，如果有足够的野兔供应的话，任何一天狐狸吃到野兔的概率是4/7。如果野兔不够或植被茂密度大于（等于）0.6，狐狸吃到野兔的概率是2/7——前提是在那种消费水平上有足够的野兔。（如野兔数量不够维持狐狸生存，10%的挨饿的狐狸会死去，不包括自然死亡。）狐狸的生命期是四年 198
左右，每天狐狸自然死亡的数量用一个随机数决定。

如果没有食物限制（即植被足够多，方块中所有野兔尽可放开胃口），每只野兔每天消耗掉方块中植被的千分之一。野兔的正常寿命是18个月左右。如植被茂密度小于0.35，野兔饿死的数量急剧上升，参见表6-3。

表6-3 野兔生命期

植被茂密度	野兔生命期
0.1 至 0.15	3个月
0.15 至 0.25	6个月
0.25 至 0.3	12个月
0.35以上	18个月

每天野兔自然死亡和饿死的总数用一个随机数决定。亚热带岛有理想的生长条件，没被野兔啮食的植被很快增长。植被茂密度遵从如下生长/消耗公式：

在一天结束时的植被 = (110% × 这一天开始时的植被) − (0.001 × 这一天开始时的野兔数量)

植被茂密度限制在0.1和1.0之间。每天结束时，20%的野兔随机迁入相邻方块。各 199
个相邻方块中迁入的野兔数量用随机数确定。类似地，每天结束时，狐狸随机迁往与当天开始时所在方块相距0块、1块或2块的方块。注意，往哪个方块迁移的可能性相同。

请分析以下三个案例：

案例1：初始时，每个方块均匀地分布有2只狐狸和100只野兔；植被到处都是，茂密度为1.0。

案例2：总共有20只狐狸，都在一个角上的方块中，其他地方没有。与狐狸所在方块同一对角线的另一个角上的方块中有800只野兔，除此之外，其他每个方块都只有10只野兔。植被茂密度处处都为0.3。

案例3：岛上没有狐狸，但每个方块中有2只野兔。植被茂密度处处都为0.5。

6-22　设计一个细胞自动机解决一个实际问题，并实现它。

6-23　（研究作业）设计一套规则，用于建模海滩上一个沙丘在海浪冲刷下的运动（侵蚀）过程（一个类似的问题是建模河岸在水的作用下的受侵蚀过程）。

6-24　（研究作业）设计对穿过机翼的气流建模的必要规则，如图6-21所示（二维）。解空间的维数和对象的维数自定。自己选取网格点数，并编写程序解决该问题。

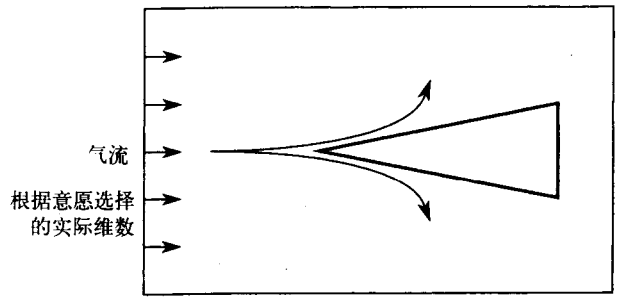


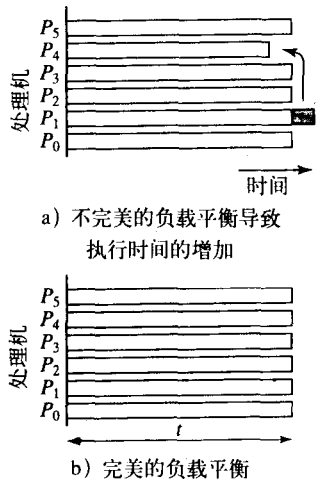
图6-21　习题6-24的图

第7章　负载平衡与终止检测

本章我们要介绍*负载平衡*（load balancing）的概念，它用于在处理器间合理地分配计算，以获得尽可能快的执行速度。一个相关的问题是检测计算何时已经结束，即所谓的*终止检测*（termination detection）。当计算是分布式时，终止检测将成为一个重要问题，并且必须考虑负载平衡。在讲述多种负载平衡和终止检测技术后，将给出一个详细的应用实例以加深对这些技术的理解。

7.1　负载平衡

迄今为止，我们把一个问题划分成固定数量、可并行执行的进程，每个进程执行已知数量的工作。另外，假定对进程在可用的处理器间只做简单地分配，而不讨论处理器类型和速度的影响。可是，由于工作不是均衡地划分或者一些处理器运行得比其他处理器快（或两者皆有），会使一些处理器比其他处理器先完成任务而变得空闲。理想的情况是，让所有的处理器连续执行这些任务，这会使执行时间最小。通过在处理器间均衡地分配任务来实现这个目标称为*负载平衡*。在第3章的曼德勃罗特计算中曾提到过负载平衡，在这种计算中没有进程间的通信。现在我们进一步阐述处理器间有通信的负载平衡问题。在执行前不知道工作量的情况下使用负载平衡技术特别有用，即使预先知道了工作量，它也有助于减轻处理器间速度差异的影响。



a）不完美的负载平衡导致执行时间的增加

b）完美的负载平衡

图7-1　负载平衡

图7-1说明了负载平衡如何可得到最小执行时间（真正的目标）。在图7-1a中，处理器P_1运行时间较长，而处理器P_4则提前完成了任务，但整个执行时间将取决于较长的P_1运行时间。理想的情况是将P_1的部分工作分给P_4以均衡工作负载。在图7-1b中，在t秒的运行时间里，所有的处理器都在运行，负载平衡很完美。看待这个问题的另一种方式，就是这个计算用一个处理器需要k个时钟周期；若用p个处理器，则在没有额外开销的并行实现时，其执行时间可减少为k/p个时钟周期。

可以在任何进程执行之前尝试*静态负载平衡*或在进程执行过程中尝试*动态负载平衡*。静态负载平衡通常指*映射问题*（mapping problem）[Bokhari，1981]或*调度问题*（scheduling problem）。有关这个问题有大量文章，多数利用优化技术，通常从程序分段的估计执行时间和它们的相关性入手。下面列出的是一些可用的静态负载平衡技术：

- *循环算法*——按进程顺序分配任务，当所有进程都给了一个任务后，再回到第一个。
- *随机算法*——随机选择进程执行任务。
- *递归对分*——将问题递归划分成同等计算量的子问题，同时使消息传递最少。
- *模拟退火*——一种优化技术。
- *遗传算法*——另一种优化技术，将在第13章中讲述。

202 图7-1也可看作装箱（bin packing）问题的一种形式（就是将对象放进箱中来减少箱子的数量），可以用装箱算法处理调度[Coffman，Garey，and Johnson，1978]。在我们的例子中，即是有固定数量大小相同的箱子（进程），目标是最小化箱子的大小。

对于通过静态链路互联网互联的处理器/计算机应把通信进程放在具有直接通信路径的处理器上运行，以减少通信延迟。这是该类系统映射问题的基本部分。对不同的网络，也许需要不同的映射解决方法。一般来讲，这是个难处理的计算问题，即所谓的NP完备问题。NP代表“不确定的多项式”，意思是解决该问题可能没有多项式时间算法。因此，常常要用启发式方法为进程选择处理器。

即使存在数学解，静态负载平衡有几个与系统和应用有关的根本缺陷。首先也是最重要的是，如果没有实际执行程序的各个分段，那是很难准确估计程序的各个部分的执行时间的。因此，不使用实际执行时间调度这些分段就注定是不准确的。另外，一些系统也许有通信延迟，这种延迟在不同情况下是不一样的，所以很难在静态负载平衡中体现这种变化的通信延迟。有些问题的求解步数是不确定的，例如，搜索算法常常需遍历一个图来求解，不论是并行地还是顺序地进行都不知道要搜索多少路径。由于静态负载平衡在这些情况中的应用不是很好，为此我们需要求助于动态负载平衡。

在动态负载平衡中，通过负载的划分依赖于将要执行分段的运行，把上面所有这些因素都考虑进去。这在运行期间确实会造成额外的开销，但比起静态负载平衡来则要有效得多。本章将侧重讨论动态负载平衡，描述获得动态负载平衡的不同方式。我们还将详细讨论一个计算最终如何结束，这在动态负载平衡中可能是一个重要问题，称为*终止检测*。

而计算则要被划分成待执行的*工作*（work）或*任务*（task），然后由进程执行这些任务。通常要把进程映射到处理器。由于我们的目标是使处理器保持繁忙，因此我们对处理器的活动感兴趣。不过，由于我们常常将1个进程映射到1个处理器上，因此我们将互换使用*进程*和*处理器*这两个术语。（映射多个进程主要是为了时延隐藏。）

7.2 动态负载平衡

动态负载平衡中，任务是在程序运行期间分配到处理器的。动态负载平衡可以划分为下述中的一类：

- 集中式
- 分散式

在集中式动态负载平衡中，任务是从一个中心位置分发的。存在清晰的主从结构，其中主进程直接控制一组从进程中的每个进程。相反，在分散式动态负载平衡中，任务是在任意进程
203 间传送的。一组工作者进程对问题进行操作，它们之间互相交互，最后向一个单独进程报告。一个工作者进程可从其他工作者进程处接收任务，也可向其他工作者进程发送任务（由自己决定完成或传递）。

7.2.1 集中式动态负载平衡

在集中式动态负载平衡中，主进程持有要执行的任务集。任务由主进程发送给从进程。从进程完成一个任务后，会向主进程请求另一个任务。这种机制是*工作池*方法的本质，在第3章中讨论生成曼德勃罗特图像时介绍了工作池。在那种情况下，工作池里放有按像素的坐标所指明的任务。因为所有从进程都是相同的，有时也用术语*复制工作者*（replicated worker）来描述这种方法。（这种思想可发展成为让特定的从进程去执行某些任务）。另一种描述同一

方法的术语是处理器农庄（processor farm）。

工作池技术可容易地用于简单的分治问题，也可用于那些任务很不相同、大小不同的问题。一般最好先分配较大或最复杂的任务。如果在计算中分配较大任务较晚，完成较小任务的从进程会闲呆着，以等待较大任务的完成。

当在执行期间任务数会发生变化时，也较适合应用工作池技术。在一些应用中，特别是搜索算法中，一个任务的执行会产生一些新的任务，尽管最终任务数必会减为零以指示计算的完成。可用一个队列存放当前等待的任务，如图7-2所示。如果所有任务大小相同且同等重要，则可使用简单的先进先出队列；如果某些任务比其他任务更重要（如期望更快地得到解），就要首先把这些任务送到从进程。其他的一些信息，如当前的最佳解等，可由主进程加以保存。

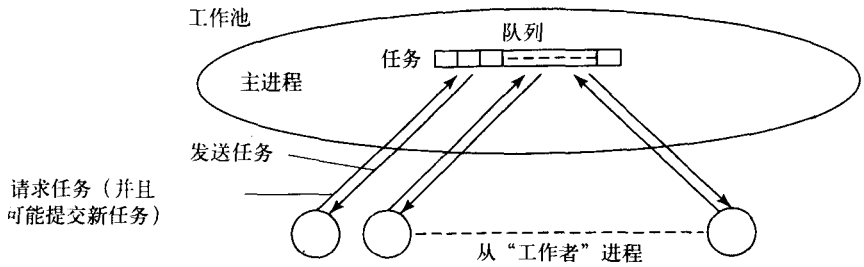


图7-2 集中式工作池

终止

在得到解后停止计算称为终止。集中式动态负载平衡的一个突出优点是，主进程很易识别计算会何时终止。对一个计算，如果其中的任务是从任务队列获取的，那么当下面两项都 204
满足时计算就终止：

- 任务队列为空
- 每个从进程为空闲态并且已经请求了另一任务，而又没有任何新的任务产生

注意，这里必须确定没有新任务的产生。（那些不产生新任务的问题在执行过程中，如曼德勃罗特计算，在任务队列为空并且所有从进程完成时就可终止。）

在一些应用中，一个从进程通过一些本地终止条件就可以检测出整个程序的终止条件，如搜索算法中对项的查找。在这种情况下，从进程会向主进程发送一个终止消息，然后主进程关闭所有其他从进程。在另一些应用中，每个从进程必须到达一个指定的本地终止条件，如本地解的收敛，像第6章的同步迭代问题一样。

7.2.2 分散式动态负载平衡

虽然集中式工作池已被广泛使用，但它的一个严重缺点是主进程一次只能发送一个任务，在初始任务发送后，它只能一次一个地响应新的任务请求。因此，当很多从进程同时请求时，就存在着潜在的瓶颈。如果从进程很少且任务又是计算密集型的，则集中式工作池是会令人满意的。但对于较细颗粒度任务和有很多从进程的情况，则把工作池分布在多个地点将会更合适。

一种方法是按图7-3那样分布工作池。这里，主进程已将最初的工作池分成几个部分，并且将每一部分发送给一组“迷你主进程”（M_0到M_{p-1}）中的每一个。每个迷你主进程控制一组从进程。对于优化问题，迷你主进程会找到本地最优，然后将其返回给主进程，主进程再选 205
出最优解。很显然可通过几个层次的分解来发展这种方法；在叶结点放置从进程，用内部结

点分割工作，就可形成一棵树，这是将一个任务等分成子任务的基本方法。对于一棵二叉树，可在树的每一层由进程把任务的一半送给一棵子树，而把另一半送给另一棵子树。另一种分布式方法则是让从进程实际持有工作池的一部分并对这一部分求解。

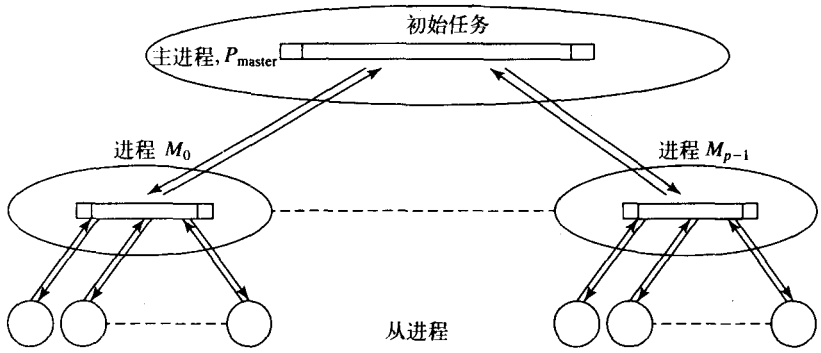


图7-3　分布式工作池

1. 全分布式工作池

一旦进程分配了工作负载，它又产生自己的任务，就存在进程间相互执行任务的可能性，如图7-4所示。任务可按如下方法传递：

1）由*接收器启动*方法。

2）由*发送器启动*方法。

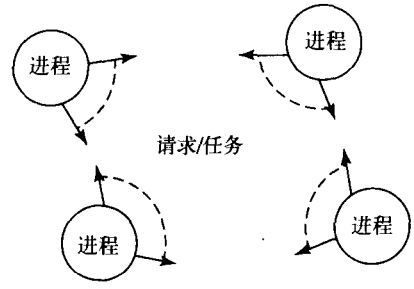


图7-4　分散式工作池

在*接收器启动*方法中，一个进程向它选择的别的进程请求任务。典型地，当一个进程有很少或没有任务执行时，会向其他进程请求任务。已经表明该方法在高系统负载时会工作得很好。在*发送器启动*方法中，一个进程向它选择的其他进程发送任务。在这种方法中，典型的是一个负载很重的进程会向愿意接收的其他进程传递一些它的任务。已经表明这种方法在整个系统负载较轻时工作得较好。另一种选择是将两种方法结合起来。不幸的是，确定进程负载状况的代价昂贵。在系统负载非常重时，由于缺少可用进程，负载平衡也可能会很难实现。

现在让我们讨论接收器启动方法中的负载平衡，不过，它也适用于发送器启动方法。有几种可行的策略。可将进程组织成一个环，进程向其最近的邻居请求任务。环形结构适合一个用环形互联网构成的多处理机系统。类似地，在一个超立方体中，进程可向每一维上与其直接相连的一个进程请求任务。当然，对于任何策略，都要小心不要让已接收的相同任务不停地传递。

2. 进程选择

如果没有特定互联网络的限制（和优势），则所有进程都是同等的候选者，进程可以选择
206 其他任何进程。对于分布式操作，每个进程可有其自己的选择算法，如图7-5所示。当本地实现时，如果问题或网络合适，则该算法可作用于问题的所有进程或不同子集。选择进程的算法包括*循环算法*（round robin algorithm）。在*循环算法*中，进程P_i向进程P_x请求任务，其中x由每次请求后都要增值的计数器给定，使用模p运算（p个进程）。如$p = 8$，x则为0、1、2、3、4、5、6、7、0、1、2、3、4、5、6、7、…。进程不选择自己（$x = i$），当$x = i$时，计数器会再加一。在*随机轮询*（random polling）算法中，进程P_i从进程P_x请求任务，其中x是在0到$n-1$（不包括i）间随机选取的一个数。

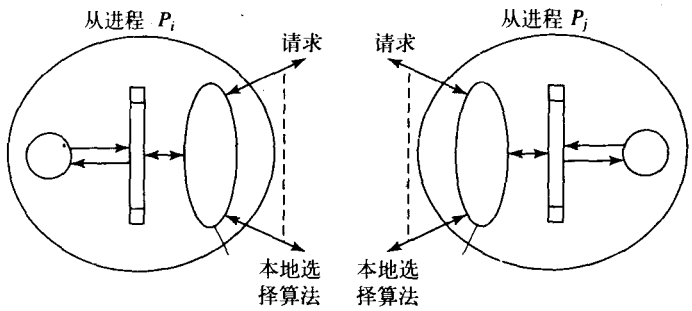


图7-5 从进程间请求任务的分散式选择算法

当进程收到一个任务请求时，它会将自己还未处理的部分任务发送给请求进程。例如，假定问题是用深度优先搜索法遍历一棵搜索树，从根开始向下访问结点，此时要保存一个未访问的结点列表，这些结点与一个进程要访问的结点通过边相连，进程将从该列表选择一个适当的未访问结点集返回给请求进程。可以使用多种策略来决定返回多少结点以及返回哪些结点。

7.2.3 使用线形结构的负载平衡

[Wilson，1995]描述了一种负载平衡技术，该技术特别适用于线形结构（或流水线）拓扑连接的处理器。不过该技术可扩展到其他互联结构。他描述的技术与transputer相关，transputer常互联成一条线。我们在这里讨论该技术以说明特定互联网络的可能性。基本思想是创建一个任务队列，并由各个处理器访问队列中的各个单元，如图7-6所示。主进程（图7-6中的P_0）从一端向队列输入任务，且任务沿队列向下移动。当一个“工作者”进程P_i（$1 \leqslant i < p$）从队列上其输入端处检测到一个任务且该进程空闲时，它就会从队列上取得这个任务。在队列左边的那些任务继续下传以填满队列的空间。新的任务从队列的左端插入，最终所有的进程都会有一个任务，此后队列会被新来的任务填充。显然这种机制可使工作者进程保持繁忙。高优先权或较大的任务可首先放入队列。

207

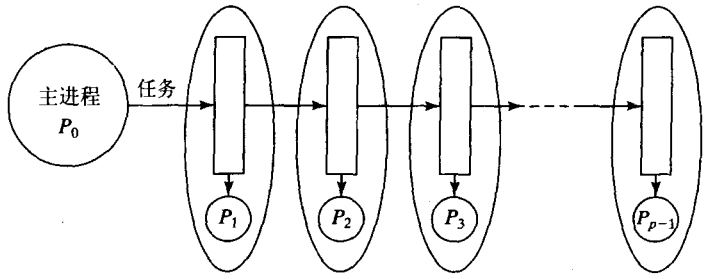


图7-6 使用流水线结构的负载平衡

在邻接的进程之间可以使用消息来编排这种移动动作。也许最好的方法是在每个处理器上运行两个进程：

- 用于左右两边的通信
- 用于当前任务

如图7-7所示。甚至可以构造3个进程：

- 用于左边通信
- 用于右边通信

• 用于当前任务

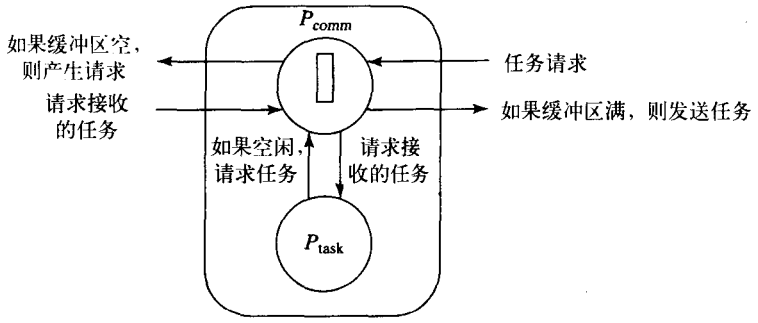


图7-7 在线形负载平衡中使用通信进程

这些构造在transputer程序中较典型，因为在transputer的硬件中支持并发进程。将这一观点应用于那些允许每个处理器上运行多个进程的MPI的实现是没有任何困难的。然而，对于那些不允许在一个处理器上运行多个进程的MPI实现，会引入巨大的且不可接受的开销。此时我们就可以依靠手工编码来实现通信和任务计算间的分时。(更吸引人的方法是使用线程，
208 如第8章中所描述的那样。)

我们用手工编码来实现通信和任务计算间的分时:

主进程 (P_0)

```
for (i = 0; i < num_tasks; i++) {
   recv(P1, request_tag);                    /* request for task */
   send(&task, P1, task_tag);                /* send tasks into queue */
}
recv(P1, request_tag);                       /* request for task */
send(&empty, P1, task_tag);                  /* end of tasks */
```

从进程$P_i(1< i <p)$

```
if (buffer == empty) {
   send(Pi-1, request_tag);                  /* request new task */
   recv(&buffer, Pi-1, task_tag);            /* task from left proc */
}
if ((buffer == full) && (!busy)) {           /* get next task */
   task = buffer;                            /* get task*/
   buffer = empty;                           /* set buffer empty */
   busy = TRUE;                              /* set process busy */
}
nrecv(Pi+1, request_tag, request);           /* check message from right */
if (request && (buffer == full)) {
   send(&buffer, Pi+1);                      /* shift task forward */
   buffer = empty;
}
if (busy) {                                  /* continue on current task */
   Do some work on task.
   If task finished, set busy to false.
}
```

在该代码中，如果有的话，可用组合的sendrecv()，而不必用send()/recv()对。

非阻塞接收例程

在前面的代码中，有必要用非阻塞nrecv()检查从右边接收来的请求。我们在伪码中只是简单地加入参数request，如收到了消息就将其置为TRUE。在实际的编程系统中，会有一些专门的机制。在MPI中，非阻塞接收MPI_Irecv()返回(在一个参数里)一个请求“句柄”，它用于随后的完成例程中以等待消息或证实在那一点消息是否已被接收（分别用于MPI_Wait()和MPI_Test()处）。实际上，非阻塞接收MPI_Irecv()在发出对消息的请求后便立即返回。

其他结构

虽然在[Wilson，1995]中没有提及，显然将该方法扩展到树是可能的，如图7-8所示。当结点缓冲区变空时，任务会从一个结点传送到它下面两个结点中的一个。 209

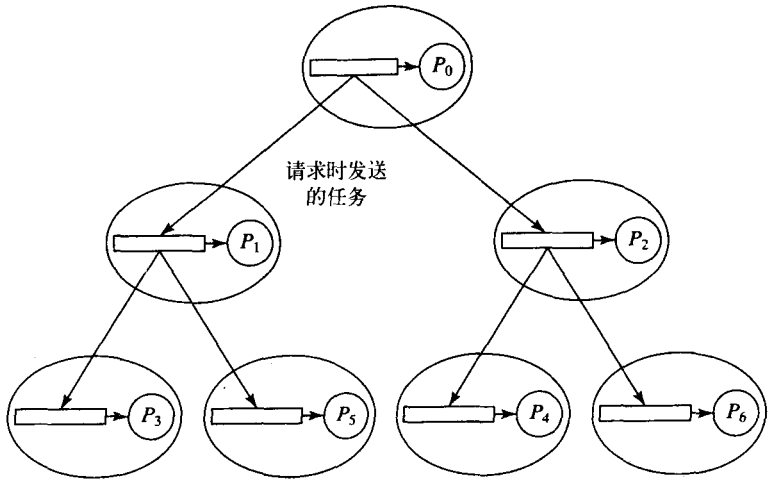


图7-8 用树进行负载平衡

7.3 分布式终止检测算法

迄今为止，我们已考虑了任务的分配。现在让我们看看如何终止这些分布式任务。虽然已提出了各种分布式终止算法，但首先让我们考察一下终止条件。

7.3.1 终止条件

当计算是分布式时，识别计算已经结束可能是困难的，除非是那种一个进程可得到一个解的问题。通常在时间t的分布式终止需要满足如下条件[Bertsekas and Tsitsiklis，1989]:

- 在时间t，对于所有进程集，存在有特定应用的本地终止条件。
- 在时间t，进程间没有消息在传送。

这些终止条件和集中式负载平衡系统中的那些相比，它们之间的细微差别是需要考虑传送中的消息。对于分布式终止系统，第二个条件是必要的，因为传送中的消息也许会重新启动一个已终止的进程。可以想像一下一个进程已到达它的本地终止条件并准备终止，而此时有另一进程正向它发送一条消息。通常第一个条件相对容易识别，只要每个进程在满足其本地终止条件时向主进程发送一条消息便可。不过第二个条件就较难识别。消息在进程间传送的时间预先是不知道的。可以等待足够长的时间以便传送中的消息到达，但这种方法是不受欢迎的，而且不允许代码在不同的体系结构上可移植。 210

7.3.2 使用确认消息实现终止

[Bertsekas and Tsitsiklis，1989]描述了一种使用请求和确认消息的分布式终止方法。该方法通用性好，数学证明正确，而且当进程将要本地终止时仍能处理正在传送的消息。Bertsekas 和Tsitsiklis给出了详细的形式化数学证明。

该方法参见图7-9。每个进程处于两种状态之一：

1）不活动

2）活动

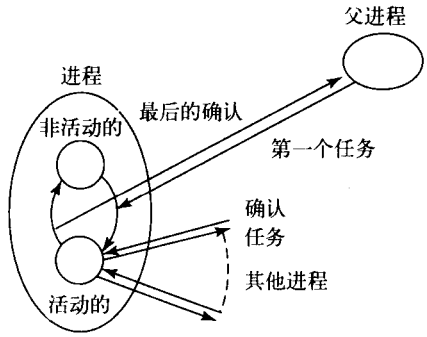


图7-9　用消息确认实现终止

开始时，没有任何任务要执行，进程处于不活动态。一旦从一个进程处收到任务，它就变成活动态。发送任务使进入活动态的进程成为其“父进程”。如果进程向一个不活动进程传递任务，它就类似地成为该进程的父进程。这样就会构建一棵进程树，每个进程有唯一的父进程。一个处于活动态的进程有可能从其他活动进程处接收更多任务，而它们不是该进程的父进程。因此，计算本身不必是树结构。在进程每次向另一进程发送任务时，它都期望对方的确认消息。当它每次从一个进程接收任务时，它立即发送一个确认消息，除非它所接收的任务来自其父进程。它只在准备变成非活动态时向其父进程发送一个确认消息。当以下条件满足时它将变成不活动的：

- 其本地终止条件已满足（所有任务已完成）。
- 它对收到的所有任务发送了确认。
- 它收到了它所发出所有的任务的确认。

最后一个条件意味着一个进程必须在其父进程之前变成不活动的。当第一个进程空闲后，
211 计算终止。

由于该算法的通用性和已被证明的正确性，它也许是最好用的。然而，一个特定应用也许会适宜于另一个解，此外某些互联结构可能会暗示有别的终止机制。

7.3.3 环形终止算法

为达到终止目的，把进程组织成一环形结构，如图7-10所示。单通环型终止算法（single-pass ring termination）如下：

1）当P_0已终止时，它产生一令牌传给P_1。

2）当P_i（$1 \leqslant i<p$）接收令牌且已经终止，它就将令牌向前传给P_{i+1}；否则，它会等待本地终止条件，然后将令牌前传。P_{p-1}将把令牌传给P_0。

3）当P_0收到令牌后，它就知道环中所有进程已经终止。如有必要，可向所有进程发送消息，通知它们全局终止。

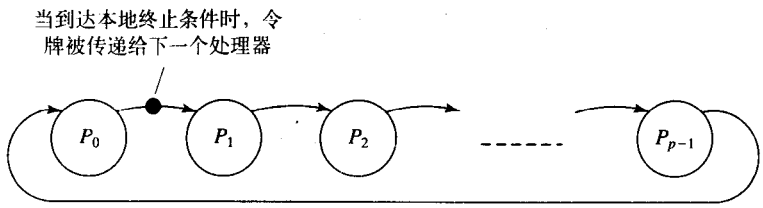


图7-10　环形终止检测算法

除了第一个进程，每个进程实现一个功能，如图7-11所示。算法假定一个进程到达本地终止条件后不能重新激活。这种假设不适用于工作池问题，在工作池问题中一个进程可向一空闲进程传送一个新任务。

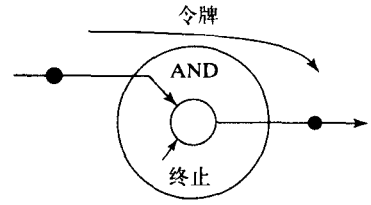


图7-11 本地进程终止算法

双通环形终止算法[Dijkstra, Feijen and Gasteren, 1983]能够处理进程重新激活问题，但需要在环上有两个通路。重新激活的原因是进程P_i传给P_j一个任务，这里$j < i$，且是在令牌已通过P_j，参见图7-12。如发生这种情况，令牌必须第二次沿环巡回。为区别这些情况，把令牌分成白色和黑色，进程也被标为白色和黑色两种。接收一黑色令牌意味着全局终止也许还没发生，令 212
牌还必须沿环重新流动。该算法为如下，也从P_0开始:

1）P_0终止时变成白色，产生一白色令牌并送给P_1。

2）当每个进程终止后，令牌沿环从一进程传递到下一个进程，但令牌的颜色可能会发生变化。如果P_i向P_j发送一个任务，且$j < i$（P_j在环中位于P_i之前），它就变成一个*黑色进程*，否则它是*白色进程*。黑色进程会把令牌染成黑色，并让它继续前传。白色进程让令牌以原有颜色（黑色或白色）前传。在P_i传出令牌后，它就变成白色进程。P_{p-1}把令牌传给P_0。

3）当P_0收到黑色令牌，它就发出一白色令牌；如收到白色令牌，则所有进程已经终止。

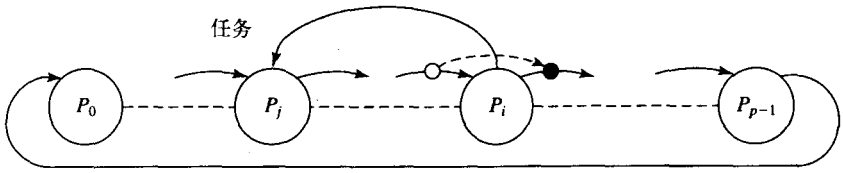


图7-12 向前面进程传递任务

注意，在两种环形算法中，P_0成为全局终止的中心点。还假定对每个请求将产生一确认信号。

树算法

图7-11描述的本地活动可用于各种互联结构，特别是树结构，它表示到该点的那些进程已经结束。使用这种机制的树的两个分支示于图7-13。当树的每个分支收到令牌且本地终止条件存在时，就将令牌向前传递。当根收到足额令牌且终止时，就发生全局终止。然后，其 213
他所有进程必须再次得到通知，也许通过树广播算法。

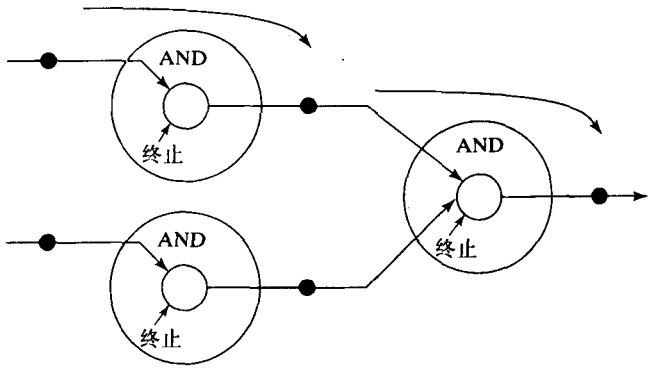


图7-13 树终止

7.3.4 固定能量分布式终止算法

另一种终止算法在系统中利用称为“能量”的定量表示。该能量类似于令牌，但有一数值。系统开始时，所有能量由一个进程即主进程持有，它把部分能量与任务传送给请求任务的进程；类似地，如果这些进程收到任务请求，它会把能量进一步划分传送给这些进程。如果进程空闲，它要在请求新的任务之前把持有的能量返回。这一能量可直接返给主进程或传送给给予其原始任务的那个进程。对于后一种情况，算法将构造一种类似于树的结构。一个进程只有在它发出的所有能量均已返回，并已组合到所持有的总能量中后，才会交回其能量。当所有能量返回到根，而且根变成空闲时，所有进程必定空闲，计算就能终止。

固定能量法的一个严重缺点是能量划分精度有限。如使用浮点运算，部分能量的累加之和可能会与原始能量不相等。另外，只能在能量实际上为零之前划分能量。如果原始整数能量足够大到应付划分的个数，则一般带验证的整数运算能克服第一个问题。

7.4 程序举例

在本节我们要讨论如何能把各种负载平衡策略应用于一个有代表性的问题。有几个应用领域，包括显式搜索和优化领域，其他领域有图像处理，光线追踪及体渲染（volume rendering）。事实上，任何能分治的问题都是工作池方法的侯选者。大多数能充分利用动态负载平衡的问题，其任务数是可变的和未知的。当然动态负载平衡对异构计算机网络也是特别有用的。

7.4.1 最短路径问题

我们要研究图上两点间最短路径问题。这是一个有名的问题，以某些形式出现在多数顺序程序设计课程中。它可叙述如下:

给定一组互联结点，结点间的链路用“权值”标记，求出从一指定结点到另一指定结点的路径，要求该路径的累计权值最小。

互联结点可用图来表示。按照图的术语，结点称为顶点，链路称为边。如果边隐含着方向
214 (即边只能沿一个方向遍历）则该图是有向图。要求解的问题是搜索图中最佳路径之一。图本身可用于求解很多不同问题，例如:

1）地图中两城镇或其他点之间的最短距离，其中权值代表距离。

2）旅行的最快路由，其中权值代表时间（如果有不同的旅行方式，最快的路由不一定是最短的路由，比如，飞行到某些城镇）。

3）坐飞机最便宜的路线，权值表示城市（顶点）间的航班费用。

4）爬山的最好路线，给定等高线的地形图。

5）计算机网络的最好路由，使消息延迟最小（顶点表示计算机，权值代表两台计算机间的延迟）。

6）最有效的制造系统，权值表示工作时间。

下面将用“爬山的最好路线”作为实例，如图7-14所示，对应的图参见7-15，其中权值表示通过连接两个营地间路径所花费的代价。注意，本例中的图是有向图，权值与沿一特定方向通过路径有关。理论上，我们可以沿两个方向作出所有营地间的路径，即全连通图，不过由于每个方向的权值会不一样它仍是一个有向图。一个方向的代价会与相反方向的代价不同（下山而不是上山！）。在有些问题中，两个方向的权值是一样的，比如，寻求开车的最短路

由时，两个方向的距离是一样的，权值相等，因此是一个无向图。

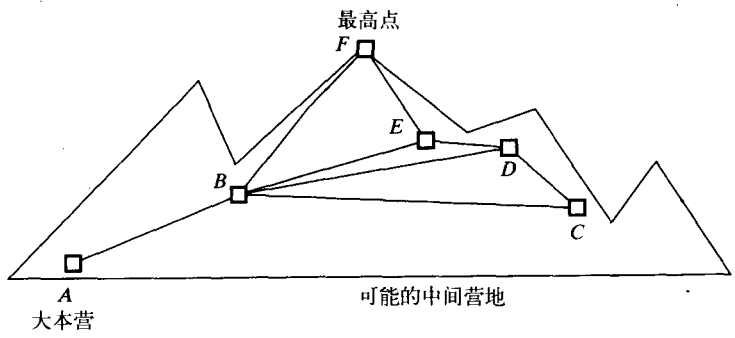


图7-14 爬山

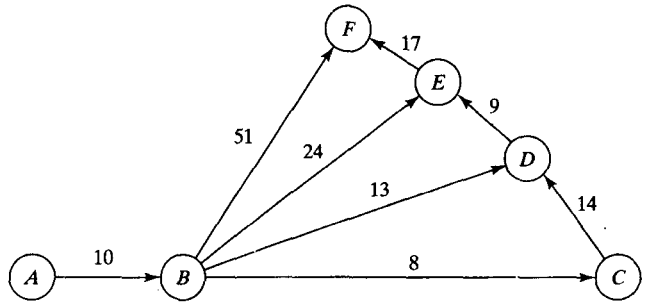


图7-15 爬山图

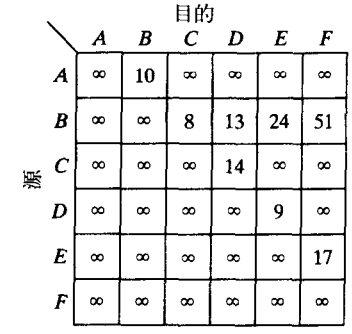
7.4.2 图的表示

我们首先建立一种在程序中表示图的方法。从顺序编程中我们已熟悉，在程序中图可以有两种基本表示方法：

1）邻接矩阵——一个二维数组a，其中a[i][j]存放与顶点*i*和*j*之间的边（如存在）有关的权值。

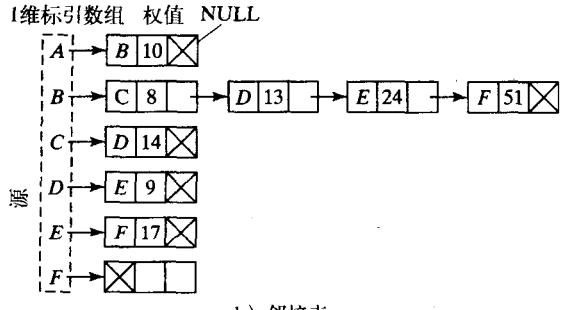
2）邻接表——对每一个顶点，有一个通过边和与边相关的对应权值直接连接到该顶点的顶点列表。

对于我们的爬山问题，两种方法如图7-16所示。邻接表是用链表实现的。邻接表中边的顺序是任意的，选定某一方法应依赖于图的特征和程序的结构。对于顺序程序，通常邻接矩阵用于稠密图，即图中每个顶点都有很多边；而邻接表主要用于稀疏图，即图中每个顶点的边很少。两者的差别在于对空间（存储）需求的不同，邻接矩阵的空间需求为O (v^2)，而邻接表的空间需求则为O (ve)，其中每个顶点有e条边，而总共有v个顶点。



源 \ 目的	A	B	C	D	E	F
A	∞	10	∞	∞	∞	∞
B	∞	∞	8	13	24	51
C	∞	∞	∞	14	∞	∞
D	∞	∞	∞	∞	9	∞
E	∞	∞	∞	∞	∞	17
F	∞	∞	∞	∞	∞	∞

a）邻接矩阵



b）邻接表

图7-16 图的表示

215

216

一般每个顶点的e是不同的，因此，邻接表空间需求的上界为O (ve_{max})。访问邻接表比访问邻接矩阵慢，因为需要顺序遍历链表，这可能要v步。对并行程序而言，可以并行访问以加速进程。除了空间和时间特征外，对于并行程序还需要考虑任务的划分及对访问信息的影响。下面我们假定采用邻接矩阵表示（尽管该图是稀疏图）。

7.4.3 图的搜索

在我们的例子中，因为到达最高点只有几种方式，因此搜索最高点也相当简单；但在更复杂的问题中，搜索不像这样易于管理，因此必须使用算法的方法。单源最短路径图算法求出从一个源顶点到一个目的顶点的最小累计权值。要确认到达最高点的最好方式，有两个著名的单源最短路径算法可做为候选:

- Moore的单源最短路径算法[Moore,1957]
- Dijkstra的单源最短路径算法[Dijkstra,1959]

这两种算法是类似的。选择Moore算法是因为它更易于并行实现，尽管它可能要做更多的工作[Adamson and Tick，1992]，该算法要求权值须是正值。(有其他算法对正负权值都能工作。)

1. Moore算法

由源顶点开始，当在考虑顶点i时，其基本算法的实现如下：找出经过顶点i到顶点j的距离，并与当前到顶点j的最短距离比较；如果经过顶点i的距离更短，则改变最短距离。用数学记号表示即是，如果d_i是从源顶点到顶点i的当前最短距离，$w_{i,j}$是从顶点i到顶点j边的权值，则有

$$d_j = \min(d_j,\ d_i + w_{i,j})$$

图7-17对算法作了说明。有趣的是，通过简单地重复使用上述公式就可以解决问题（一个迭代解法），细节请参见[Bertsekas and Tsitsiklis，1989]。

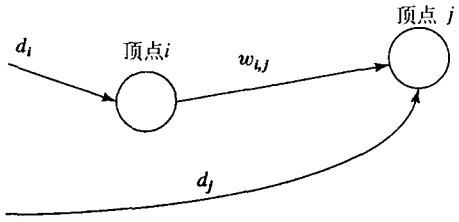


图7-17　Moore最短路径算法

该公式用有向搜索实现。需构造一个先进先出的顶点队列，存放要检查的顶点列表，只有在顶点队列中的顶点才会被考察。开始时只有源顶点在队列中；另外还需要一个结构以存放从源点到其他每个顶点的
217 当前最短距离。假定有n个顶点，且顶点0是源顶点。从源顶点到顶点i的当前最短距离存放在数组`dist[i]` ($1 \leqslant i < n$) 中。开始时因为并不知道这些距离，故数组元素的初值为无穷大。假定`w[i][j]`存放从顶点i到顶点j的边的权值（无边则权值为无穷大），代码可为如下形式:

```
newdist_j = dist[i] + w[i][j];
if (newdist_j < dist[j]) dist[j] = newdist_j;
```

当找到到顶点j的较短距离后，就把顶点j加入到队列（如还不在队列），这会使顶点j被再一次检查，这是该算法的一个重要特征，在Dijkstra算法中则没有这一特征。

2. 图的搜索阶段

用爬山图作为例子，按照其步骤，看看该算法是如何从源顶点进行的。

两个关键数据结构的初值为:

要考察的顶点

A					

vertex_queue

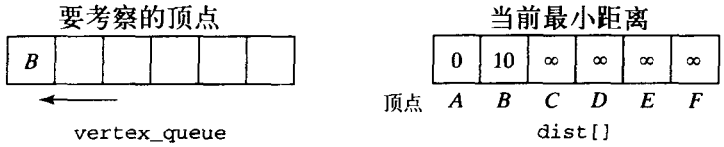
当前最小距离

0	∞	∞	∞	∞	∞
顶点 A	B	C	D	E	F

dist[]

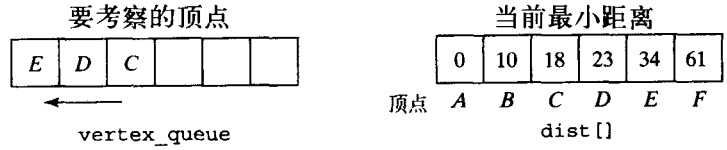
当A是源顶点时，元素dist[A]总是为零。如果不把A选作源顶点，则该结构就提供了完整的通用性。

首先，检查从顶点A出发的每条边。在我们的图中，将是顶点B。到B的权值为10，它提供了到B的第一个距离（实际上唯一的距离）。两个数据结构vertex_queue和dist[]更新如下：



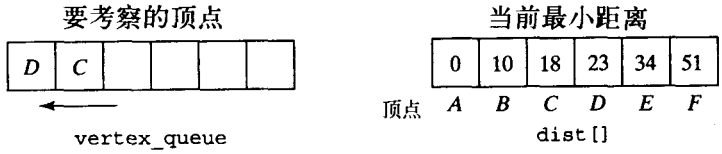
一旦一个新顶点B放入顶点队列，环绕B的搜索任务就开始了。现在还有四条边要检查： 218
即到C、D、E和F。该算法中，不必以特定的次序检查这些边。但Dijkstra算法要求首先检查最近的顶点，从而必须顺序处理；而Moore算法也许会要求重新检查顶点。我们以F、E、D和C的次序来检查各边。

经过顶点B到各顶点的距离分别为dist[F]=10 + 51 = 61、dist[E]= 10 + 24 = 34、dist[D]= 10 + 13 = 23和dist[C]= 10 + 8 = 18。由于它们都是新距离，因此所有顶点都要加到队列中（F除外），如下：

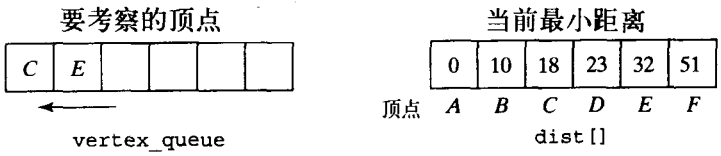


不需要加入顶点F是因为它是终点、没有出边且不需处理（如果加入F，会发现没有出边）。

从顶点E开始，它有一权值为17的边到顶点F。通过顶点E到顶点F的距离是dist[E]+ 17 = 34 + 17 = 51，它比当前到顶点F的距离小，故要替换这一距离，导致

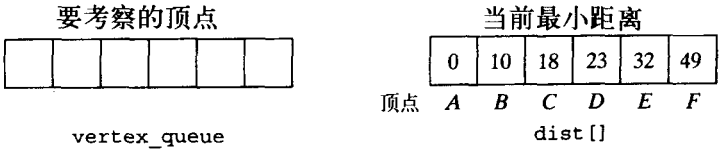


接下来是顶点D，它有一条边到顶点E，权值为9。经过顶点D到顶点E的距离为dist[D]+ 9 = 23 + 9 = 32，它比当前到顶点E的距离小，故替换该距离。把顶点E加到队列，如下：



接下来是顶点C，它有一条边到顶点D，权值为14，因此，经过顶点C到顶点D的距离为dist[C]+ 14 = 18 + 14 = 32，它比当前到顶点D的距离23大，因此该距离保持不变。

接下来是顶点E(又一次)，它有一条边到F，权值为17，使经过顶点E到顶点F的距离为dist[E]+ 17 = 32 + 17 = 49，它比到顶点F的当前距离小，故替换该距离，如下：



219

现在再没有顶点要考察。我们得出从顶点A到其他各点的最短距离，包括终点F。通常除了距离外，还要求出实际路径，那么在记录距离时就需把路径存储下来。该例中的路径为$A \rightarrow B \rightarrow D \rightarrow E \rightarrow F$。

3. 顺序代码

在该代码中省略了维护顶点队列的细节。由next_vertex()返回顶点队列中的下一个顶点，如没有则返回no_vertex。假定我们使用邻接矩阵，名为w[][]，采用顺序访问以得到下一条边。顺序代码可为如下形式:

```
while ((i = next_vertex()) != num_vertex)              /* while a vertex */
   for (j = 0; j < n; j++)                             /* get next edge */
      if (w[i][j] != infinity) {                       /* if an edge */
         newdist_j = dist[i] + w[i][j];
         if (newdist_j < dist[j]) {
            dist[j] = newdist_j;
            append_queue(j);           /* vertex to queue if not there */
         }
      }                                /* no more vertices to consider */
```

4.并行实现

下面我们来观察一下集中式工作池和分散式工作池这两种解决方案。

(1) 集中式工作池

所考虑的第一个并行实现将使用一个集中式工作池来存放顶点队列vertex_queue[]作为任务。每个从进程从顶点队列取得顶点，并按照前面图7-2所示方式返回新的顶点。对于要确认边和计算距离的从进程，它们需要访问存放图权值的结构（邻接矩阵或邻接表）和存放当前最小距离的数组dist[]。如果这些信息由主进程所有，就要向主进程发送消息以访问这些信息。这会导致非常严重的通信开销。由于存放图权值的结构是固定的，可将该结构拷贝到每个从进程中。假定使用的是拷贝的邻接矩阵。现在，再假设距离数组dist[]为中央存放的，且与顶点一起整体拷贝，也可单独对距离进行请求。代码可为如下形式:

主进程

```
while (vertex_queue() != empty) {
   recv(P_ANY, source = P_i);                  /* request task from slave */
   v = get_vertex_queue();
   send(&v, P_i);                              /* send next vertex and */
   send(&dist, &n, P_i);                       /* current dist array */
   recv(&j, &dist[j], P_ANY, source = P_i);    /* new distance received */
   append_queue(j, dist[j]);                   /* append vertex to queue */
                                               /* and update distance array */
};
recv(P_ANY, source = P_i);                     /* request task from slave */
send(P_i, termination_tag);                    /* termination message*/
```

220

从进程（进程i）

```
send(P_master);                                /* send request for task */
recv(&v, P_master, tag);                       /* get vertex number */
if (tag != termination_tag) {
   recv(&dist, &n, P_master);                  /* and dist array */
   for (j = 0; j < n; j++)                     /* get next edge */
      if (w[v][j] != infinity) {               /* if an edge */
         newdist_j = dist[v] + w[v][j];
         if (newdist_j < dist[j]) {
```

```
            dist[j] = newdist_j;
            send(&j, &dist[j], P_master);      /* add vertex to queue */
        }                                      /* send updated distance */
    }
}
```

很明显，顶点数和距离数组可放在一个消息中发送。还要注意各个从进程或许有不完全一样的距离，因为它们是由不同的从进程连续更新的。

主进程等待任何来自任何从进程的请求，但必须对进行请求的指定从进程加以响应。在我们的伪码中，source = P_i用来表示消息源。在实际编程系统中，可通过让每个从进程发送其标识（可能作为唯一标记）来确认源。在MPI中，能通过读取MPI_Recv()例程返回的状态字找到实际的消息源。

（2）分散式工作池

有一种分布式工作池方法能用于我们的求解问题。任务队列，在我们的实例中的vertex_queue[]，也可以是分布式的。一种方便的方法是让从进程i只围绕顶点i搜索，如果顶点i在队列中存在，就让进程i拥有顶点i的顶点队列项。换句话说，队列中有一个元素专门用来存放顶点i，该项在进程i中。数组dist[]也分布在进程中间，以便进程i保存当前到顶点i的最短距离。为了确认顶点i的边，进程i还需存储顶点i的邻接矩阵/表。

根据我们的安排，算法可按如下方式进行：由一协调进程激活搜索，将源顶点装载到适当的进程。在我们的例子中，A是第一个要搜索的顶点。首先激活指派给顶点A的进程，该进程立即在顶点周围开始搜索，找出到相连顶点的距离；然后，把该距离送到相应的进程。到顶点j的距离将被送到进程j，以与它当前存储的值相比较，如果当前存储的值较大就被替换。在我们的例子中，到顶点B的距离将与负责顶点B的进程联系。按这种方式，在搜索中将更新所有的最短距离。如果d[i]的内容改变，进程i就要被重新激活，再次搜索。图7-18示出了消息传递的情况。注意消息传递分布在许多从进程间，而不是集中在主进程上。 221

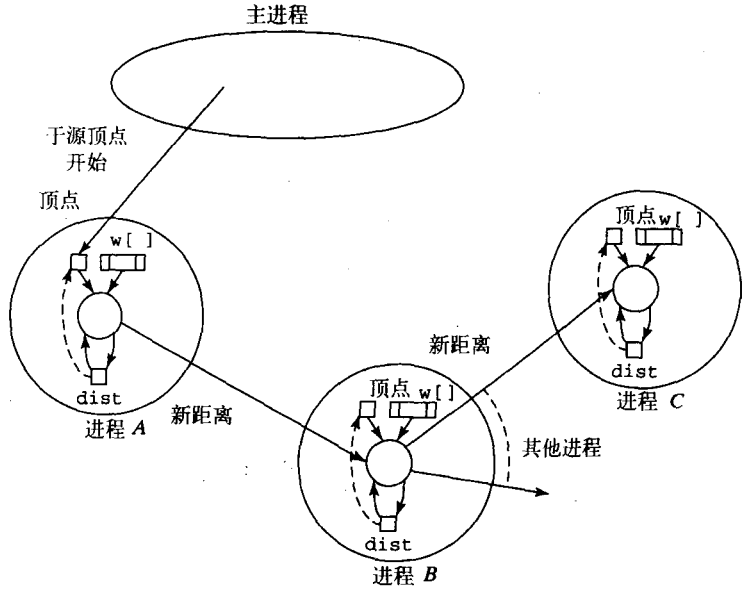


图7-18 分布式图搜索

从进程的代码段可为如下形式：

从进程（进程i）

```
recv(newdist, P_ANY);
if (newdist < dist) {
   dist = newdist;
   vertex_queue = TRUE;                    /* add to queue */
} else vertex_queue == FALSE;
if (vertex_queue == TRUE)                  /* start searching around vertex */
   for (j = 0; j < n; j++)                 /* get next edge */
      if (w[j] != infinity) {
         d = dist + w[j];
         send(&d, P_j);                    /* send distance to proc j */
      }
```

上述代码显然可简化为:

从进程（进程i）

```
recv(newdist, P_ANY);
if (newdist < dist) {
   dist = newdist;                         /* start searching around vertex */
   for (j = 1; j < n; j++)                 /* get next edge */
      if (w[j] != infinity) {
         d = dist + w[j];
         send(&d, P_j);                    /* send distance to proc j */
      }
}
```

222

需要用一种机制来重复这些动作，并在所有进程空闲时终止。该机制必须处理传输中的消息。最简单的解决方法是利用同步消息传递，其中一个进程只有在目的方收到消息后才能继续运行。对该方法及确定收到最后一个确认（如7.3节中描述的那样）的唯一父进程的更有效的方法的研究留作习题。

注意，一个进程只有在其顶点放入队列之后才是活动的。有可能很多进程不是活动的，从而导致一种低效的解决方案。如果将一个顶点分到每个处理器，则该方法对大图也是不实用的。在那种情况下，可把一组顶点分配到一个处理器上。

7.5 小结

本章介绍了以下内容:

- 集中式和分布式工作池及负载平衡技术
- 几个分布式终止算法
- 最短路径图搜索的应用

推荐读物

这些年来，有关静态和动态的任务调度已有大量的研究论文。在[Graham，1972]中可找到静态负载平衡，另一篇早期任务分配论文是[Chu et al.，1980]参考了前人的工作。在[Efe，1982]、[Lo，1988]、[Shirazi and Wang，1990]中描述了启发式方法。在[Iqbal，Salz，and Bokhari，1986]中对静态和动态方法进行了比较。静态负载平衡的其他细节和方法可在[Lewis and El-Rewini，1992]和[El-Rewini，1996]中找到。有关调度的教科书可参见[Bharadwaj et al.，1996]，它提供了详细的数学处理。

分布式系统中的负载平衡在很多论文中也有描述，比如[Tantawi and Towsley，1985]、[Shivaratri，Krueger and Singhal，1992]和[El-Rewini，Ali and Lewis，1995]。在[Shirazi，Hurson and Kavi，1995]中发表了一个论文集。[Jacob，1996]特别在工作站网络中考虑了负载平衡。在[Bertsekas and Tsitsiklis，1989]中，依据数学支持，利用接收最后确认的一个父进程的概念，全面描述了强有力的动态负载平衡技术。该方法也用在[Lester，1993]的并行程序中。把负载平衡看作物理系统最小能量优化的概念在[Fox et al.，1988]中作了描述。不少教科书如[Barbosa，1996]对终止检测进行了论述。

在[Mateti and Deo，1982]、[Paige，1985]和[Adamson and Tick，1992]中考虑了最短路径问题的并行算法；而[Lester，1993]则考虑了最短路径问题的并行程序设计问题。 223

参考文献

ADAMSON, P., AND E. TICK (1992), "Parallel Algorithms for the Single-Source Shortest-Path Problem," *Proc. 1992 Int. Conf. Par. Proc.*, Vol. 3, pp. 346–350.

BARBOSA, V. C. (1996), *An Introduction to Distributed Algorithms*, MIT Press, Cambridge, MA.

BERMAN, K. A., AND J. L. PAUL (1997), *Fundamentals of Sequential and Parallel Algorithms*, PWS, Boston, MA.

BERTSEKAS, D. P., AND J. N. TSITSIKLIS (1989), *Parallel and Distributed Computation Numerical Methods*, Prentice Hall, Englewood Cliffs, NJ.

BHARADWAJ, V., D. GHOSE, V. MANI, AND T. G. ROBERTAZZI (1996), *Scheduling Divisible Loads in Parallel and Distributed Systems*, IEEE CS Press, Los Alamitos, CA.

BOKHARI, S. H. (1981), "On the Mapping Problem," *IEEE Trans. Comput.*, Vol. C-30, No. 3, pp. 207–214.

CHU, W. W., L. J. HOLLOWAY, M.-T. LAN, AND K. EFE (1980), "Task Allocation in Distributed Data Processing," *Computer*, Vol. 13, No. 11, pp. 57–69.

COFFMAN, E. G., JR., M. R. GAREY, AND D. S. JOHNSON (1978), "Application of Bin-Packing to Multiprocessor Scheduling," *SIAM J. on Computing*, Vol. 7, No. 1, pp. 1–17.

DIJKSTRA, E. W. (1959), "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, Vol. 1, pp. 269–271.

DIJKSTRA, E. W., W. H. FEIJEN, AND A. J. M. V. GASTEREN (1983), "Derivation of a Termination Detection Algorithm for a Distributed Computation," *Information Processing Letters*, Vol. 16, No. 5, pp. 217–219.

EFE, K. (1982), "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, Vol. 15, No. 6, pp. 50–56.

EL-REWINI, H. (1996), "Partitioning and Scheduling," Chap. 9 in *Parallel and Distributed Computing Handbook*, Zomaya, A. Y., ed., McGraw-Hill, NY.

EL-REWINI, H., H. H. ALI, AND T. LEWIS (1995), "Task Scheduling in Multiprocessor Systems," *Computer*, Vol. 28, No. 12, pp. 27–37.

FOX, G., M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER (1988), *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, Englewood Cliffs, NJ.

GRAHAM, R. L. (1972), "Bounds on Multiprocessing Anomalies and Packing Algorithms," *Proc. AFIPS 1972 Spring Joint Computer Conference*, pp. 205–217.

IQBAL, M. A., J. H. SALZ, AND S. H. BOKHARI (1986), "A Comparative Analysis of Static and Dynamic Load Balancing Strategies," *Proc. 1986 Int. Conf. Par. Proc.*, pp. 1040–1047.

JACOB, J. C. (1996), "Task Spreading and Shrinking on a Network of Workstations with Various Edge Classes," *Proc. 1996 Int. Conf. Par. Proc.*, Part III, pp. 174–181.

LESTER, B. (1993), *The Art of Parallel Programming*, Prentice Hall, Englewood Cliffs, NJ.

LEWIS, T. G., AND H. EL-REWINI (1992), *Introduction to Parallel Computing*, Prentice Hall, Englewood Cliffs, NJ.

LO, V. M. (1988), "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Trans. Comput.*, Vol. 37, No. 11, pp. 1384–1397.

LO, V. M., AND S. RAJOPADHYE (1990), "Mapping Divide-and-Conquer Algorithms to Parallel Architectures," *Proc. 1990 Int. Conf. Par. Proc.*, Part III, pp. 128–135.

MATETI, P., AND N. DEO (1982), "Parallel Algorithms for the Single Source Shortest Path Problem," *Computing*, Vol. 29, pp. 31–49.

MATTSON, T. G. (1996), "Scientific Computation," Chap. 34 in *Parallel and Distributed Computing Handbook*, Zomaya, A. Y., ed., McGraw-Hill, NY.

MOORE. E. F. (1957), "The Shortest Path Through a Maze," *Proc. Int. Symp. on Theory of Switching Circuits*, pp. 285–292.

PAIGE, R. C. (1985), "Parallel Algorithms for Shortest Path Problems," *Proc. 1985 Int. Conf. Par. Proc.*, pp. 14–20.

SHIRAZI, B., AND M. WANG (1990), "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *J. Par. Dist. Comput.*, Vol. 10, pp. 222–232.

SHIRAZI, B. A., A. R. HURSON, AND K. M. KAVI (1995), *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE CS Press, Los Alamitos, CA.

SHIVARATRI, N. G., P. KRUEGER, AND M. SINGHAL (1992), "Load Distribution for Locally Distributed Systems," *Computer*, Vol. 25, No. 12, pp. 33–44.

TANTAWI, A. N., AND D. TOWSLEY (1985), "Optimal Load Balancing in Distributed Computer Systems," *J. ACM*, Vol. 32, No. 2, pp. 445–465.

WILSON, G. V. (1995), *Practical Parallel Programming*, MIT Press, Cambridge, MA.

ZOMAYA, A. Y., ed. (1996), *Parallel and Distributed Computing Handbook*, McGraw-Hill, NY.

习题

科学/数值习题

7-1 把进程分配到处理器的一个方法是利用随机数生成器进行随机分配。试通过该技术用于将一串数进行累加的并行程序中来探讨该技术。

7-2 对任何独立算术任务集，利用7.2.3节中所述的流水线结构编写一个并行程序以实现负载平衡技术。

7-3 旅行商问题是一个经典的计算机科学问题（尽管它也可看作是一个实际生活问题）。从一个城市出发，目标是沿某一路由访问n个城市，且每个城市只访问一次，要使旅行的距离为最小。n个城市可以认为有不同的连接，用一加权图描述连接。从含有25个主要城市的地图上获得实际数据，然后编写一个并行程序解决该旅行商问题。

7-4 利用7.2.3节中所述的线形结构负载平衡实现Moore算法。

7-5 如本书中指出的那样，7.4节所述的分散式工作池方法在搜索图时效率不高，因为只有在其顶点放入队列后进程才是活动的。开发一个更加有效的工作池方法，使进程更加活跃。

224 ~ 225

7-6 分别用Moore算法和Dijkstra算法编写搜索图的负载平衡程序，比较算法的性能并给出结论。

现实生活习题

7-7 单源最短路径算法能用于求出多计算机互联网络（如网格网络或超立方体网络或人们想设计的任何互联网络）中消息的最短路由。编写一个并行程序，它能求出通过任意互联网络的最短路由和通过没有完全交换的你的计算机机群中特定一个的最短路由。

7-8 QoS（quality-of-service，服务质量）用来描述在一定约束条件下一个通信网络（最著名的要数因特网）提供数据传输的好坏程度。QoS有若干个可由用户说明的参数（初始响应时间，最大数据传输延迟等等），并可在每条弧上用不同的权值建模。编写一个搜索图的并行程序，该图中每条弧有两个权值，要求找到一条路径使两个权值的累加权值最小化。可能得不到两个权值的累加权值的绝对最小值，为此需要对每一个累加权值提供一个可接受的最大值。

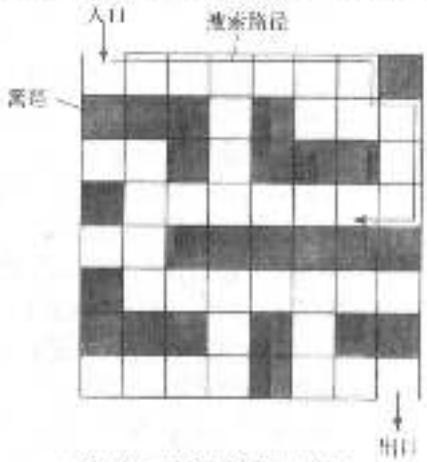


图7-19 习题7-9的迷宫样图

7-9 你被委托开发一个挑战性迷宫，它要建在一个富丽堂皇的家庭中。该迷宫在棚格上的布局参见图7-19，开发一个并行程序，求出奶酪的位置，以便当使用图7-19所示的迷宫算法时在迷宫中停留的时间最长。迷宫算法是保持路径的左边有奶酪或墙，这样就可保证最终找到出口[Berman and Paul，1997]。

7-10 一栋建筑有许多互联的房间，其中一间有一盆金子，如图7-20。试画一张图描述房间的格局，其中每个顶点是一个房间，连接房间的门表示为房间之间的边，如图7-21所示。编写一个程序求出从门外到放置金子房间的路径。注意，边是双向的，图中也许会有环路。

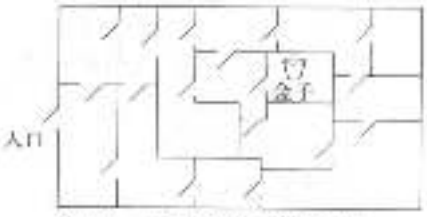


图7-20 习题7-10的房间平面图

7-11 历史上，银行使用过两个竞争算法中的一个或另一个来处理出纳处的顾客流量：多队列和单队列。在多队列方法中，每个出纳有其自己的队列，就像超市中的那样。在这种模型的标准形式中，顾客进入银行，选择一个队列排队，一直呆在队列直到出纳员为他服务。一种流行的变通方法允许“跳队”，即队列中的每位顾客在不断地评估如果他站到另一队是否会使得他得到更快服务的机会。而在单队列方法中，只有一个队列。

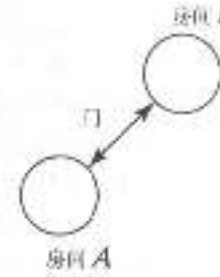


图7-21 问题7-10的表示图

226

队列头部的顾客首先被出纳选择完成业务。你的任务是使用一个并行程序模拟标准的多队列和单队列方法，准备一页总结（管理报告），简述在如下的假设条件下所发现的每种方法的优点和缺点。除了顾客平均等待时间和最大等待时间这些条目外，收集你认为与报告中结论相关的其他统计数字。

假设：

1）有5位出纳员。

2）所有的队列长度没有限制，如有必要，顾客可沿占位点鞠躬而行。不过，队列在每天上班的时候是空的。下班后不允许顾客再到队中排队，但是已经在队中的顾客

允许完成他们的业务。

3）顾客随机到达银行。由于银行位于一所一流大学附近，顾客往往集中在下课的时候：这时的前后10分钟内每分钟有10个新顾客到达（平均）；而在所有其他时间每
227 分钟有2个新顾客到达（平均）。实际到达是随机的，在该段时间上，均匀分布在每分钟到达1至19的范围内，其他时间则分布在每分钟到达0至4的范围内。

4）每项业务的完成将花费一个随机时间。平均起来，业务处理需要5分钟，但是在1到9分钟的范围内均匀分布。此外认为每个顾客只进行一项业务。

5）在上午9点到下午6点间（银行的上下班时间）运行该模拟程序100天，得出数据，根据该数据，为你的总结报告得出结论。

7-12 你是一个大公司的董事长，该公司雇佣将近1百万人。公司的人事部门巧妙地用树形结构把所有雇员组织起来，每个雇员向一个管理人员汇报，每个管理人员最多有8个或最少有2个雇员向其汇报。假定向一个管理人员汇报的雇员数为5人（这也许是不恰当的）。因此树的平均深度大致是9。（略少于1000000个最低层的雇员向大约200000名第一层管理人员汇报，他们又向大约40000名第二层管理人员汇报，后者又向大约8000名第三层管理人员汇报，以此类推。）

你刚从美国检察官那里听说你的一个雇员受到指控，这可能会也可能不会影响你的公司。你不知道该雇员的名字。你的任务是根据官方的人事组织图，找出这个雇员。从你直接管理的雇员开始，采用广度优先搜索，直到你找到受指控的雇员。注意：你可假设任何不受指控的雇员将回答“不是我！”，而受到指控的雇员将回答“是的，联邦政府抓了我！”

7-13 有张表定义了一个主要城市某区域的一组街道，其中许多街道是单行道，另外，有几条隧道和桥允许驾驶员越过街道。所有街道都编了号，偶数用于东西向街道，而奇数用于南北向街道。表的每行有如下形式：

- 描述的街道号码。
- 越过街道。
- 越过街道。
- 方式（单向或双向）。

作为例子，其中一行也许像13、4、6、1，表示它描述街区的13号街道，它横跨4号和6号街道，是从4号到6号的单行街道。（如该行是13、6、4、1，则该街道是从6号到4号的单行街道）。另一行或许是13、6、22、2，它表示13号街道是双向街道，街区中的一个隧道或桥连接6号和22号街道（在6号和22号之间的越过街道上没有出入口）。完成下面的一个或多个问题：

1）求出城中出租车能够从一个十字路口行驶到另一十字路口的路径数，任何十字路口只能经过一次。

2）求出出租车能够从一个十字路口行驶到另一十字路口的最短路径（经过最少的街区），任何十字路口只能经过一次。注意：与桥或隧道相连的唯一十字路口是那些在桥或隧道两端的十字路口。

3）求出出租车能够从一个十字路口行驶到另一十字路口的最长路径（经过最多的街区），任何十字路口只能经过一次。注意：与桥或隧道相连的唯一十字路口是那些在桥或隧道两端的十字路口。

7-14 生物系一位有才气但却色盲的研究员，她在Petri碟中人工培育了一种非常可怕的细菌

样品。培养液是不透明的白色液体，而细菌在可见光下呈粉红色。由于她不能辨别黄色、橙色和红色，这给她每天评价细菌生长的工作带来了很大的障碍。 228

她装配了将数据直接输入计算机的数字相机，雇佣你编写一个扫描程序来计算被细菌覆盖的Petri碟表面的百分比。另外，你的程序要用兰色/绿色显示Petri碟的表面。

在进行初步的实验后，你已确定以（x，y）坐标为中心的一块Petri碟区域，其从白到粉红范围的平均色调依赖于（x，y）坐标和实验已进行的时间长短t。由于不完全清楚的原因，准确的关系似乎为:

$$\frac{t}{100}+\frac{x+y}{x_{\max}+y_{\max}}=Z$$

其中，一个区域的色调在$Z<0.95$时为白色，在其他情况下则为粉红色。你的程序要计算和显示在特定的实验时刻t时Petri碟中的细菌分布。实现该程序以致于能对任何特定点进行放大。注意：尽管图形不会与锯齿形类似，但它在计算上应与随时间变化的不规则碎片形状类似。

7-15 最近，电视、报纸和电影都是有关外星人的故事，对Tom好像也是这样。因此，当一个向他所在的公寓的居民提出一个多维递归问题的、模样怪异的陌生人走近他时，汤姆欣然地接受了该问题。虽然他模糊地知道，如果他不能解决这个问题，他的家人也许再也见不到他，但他对自己的数学才能十分自信而并没有丝毫的操心。

汤姆唯一的担心是外星人在处理维数大于3时似乎比他更轻松，但汤姆自信自己的能力，立刻对它研究起来。

给定一个半径为r的N维球体，以N维坐标系的原点为中心。计算球体中整数坐标点的个数。下面是外星人为检查他的工作所提供的例子:

（a）半径为1.5的3维球体中有19个整数坐标点:

当第一个坐标为-1时球中有5个点:

(-1, 0, 0), (-1, 0, 1), (-1, 0, -1), (-1, -1, 0), (-1, 1, 0),

当第一个坐标为1时球中还有5个点:

(1, 0, 0), (1, 0, 1), (1, 0, -1), (1, -1, 0), (1, 1, 0),

当第一个坐标为0时球中有9个点:

(0, 0, 0), (0, 1, 0), (0, 1, 1), (0, 1, -1), (0, -1, 0), (0, -1, 1),

(0, -1, -1), (0, 0, -1)和(0, 0, 1)。

（b）半径为2.05的2维球体中有13个整数坐标点:

(0, 0), (-1, 0), (-2, 0), (1, 0), (2, 0), (-1, -1), (-1, 1), (1, -1),

(1, 1), (0, -2), (0, -1), (0, 1), (0, 2) 229

（c）半径为25.5的一维球体中有51个整数坐标点:

(±25, ±24, ±23, ..., ±1, 0)。

第8章　共享存储器程序设计

在本章中，我们将概述在共享存储器中进行编程的一些方法，包括进程、线程、并行程序设计语言以及具有编译器命令和库例程的顺序语言的使用。我们将从标准的UNIX进程开始。UNIX进程方法引入了“fork-join”（“分叉-接合”）模型，在稍后讨论的OpenMP中使用了这种模型。然后我们将较详细地描述已广泛应用于众多多处理机和单处理器平台上的IEEE线程标准Pthread。对于并行程序设计语言方法，我们将局限于讨论一些典型特征和通用技术，而不对某个特定的并行程序设计语言进行描述。作为使用编译器命令（以及有关的库例程）的例子，我们将叙述在共享存储器多处理机中已广泛接受的并行程序设计的工业标准OpenMP。此外，我们将描述不论使用何种编程工具的并行程序设计中的性能问题，以及涉及包括顺序一致性在内的共享数据和同步问题。最后我们提供某些并行程序设计的例子。在机群上进行共享存储器编程使用了许多相同的概念，这将在第9章中加以讨论。

8.1　共享存储器多处理机

在1.3节中，我们讲述了多处理机系统的两种基本类型——共享存储器多处理机和消息传递多计算机。到目前为止，我们对消息传递的多计算机即计算机机群进行了较多的讨论。现在我们将把注意力转移到共享存储器系统的编程上来。共享存储器系统通常是为特定目的而设计和生产的系统，但可能非常经济有效，特别是如双或四奔腾系统那样的小型共享存储器多处理机系统。

在一个共享存储器系统中，任一个处理器都可以访问全部的存储器单元。所谓单-编址空间（single address space），就是每一个存储单元都由一个单地址范围内的某个特定地址所指定。对于少量处理器的系统，一个通用的体系结构就是单总线的体系结构。其中，所有的处理器和存储模块都连接在同一总线上，如图8-1所示。这种体系结构只适用于最多大约8个处理器的系统，因为总线在某一特定时刻只能被一个处理器使用，在总线上增加处理器意味着总线竞争的增加，从而很快使总线变得饱和。虽然cache（高速缓冲存储器）的使用可大大减少对主存储器的访问需求，但如同在单处理器系统中一样，每个处理器通常使用多级cache，可是单总线仍只有有限的带宽。

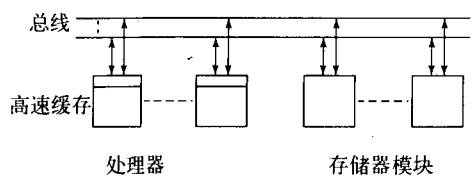


图8-1　使用单总线相连的共享存储器多处理机

对于具有较多处理器的系统，为获得足够带宽可以使用包括如图8-2所示的全交叉开关那样的多重互联。价格昂贵的交叉开关可提供处理器和单个存储器模块间的全互联。也可使用包括多级互联网（参见第1章）以及交叉开关和总线组合在内的其他互联结构。理想地，共享存储器系统应具有均匀存

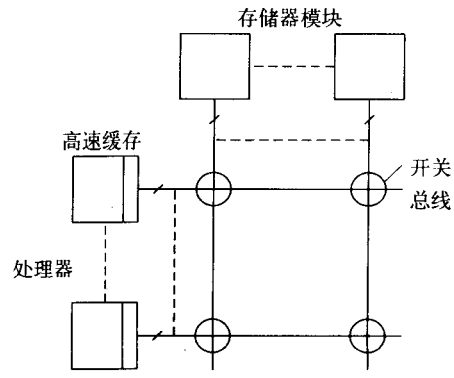


图8-2　使用纵横交叉开关的共享存储器多处理机

储器存取（uniform memory access，UMA），即从任何处理器对任何存储单元的访问都具有相同的高速访问时间。可以构建一个也许有100个以上处理器的UMA系统（例如，SUN公司的Fire 15K服务器可具有多至106个处理器）。为降低成本和增加可扩展性的另一种方式是使互联网。互联网的使用会使某些存储器模块在物理上更靠近某些处理器，从而使对主存储器单元的访问时间随离开它的距离远近而不同，即非均匀存取（NUMA）系统。不论是哪一种情况，在所有系统中高速缓存通常都会存在，以存放最近访问过的主存储器单元的内容。在本章中，我们将对共享存储器多处理机系统中编程方法的典型特征加以描述。在8.6.1节中将讨论在共 231
享存储器多处理机系统中编程时应知晓的一些非常重要的高速缓存的有关特征。

存在几种对共享存储器多处理机系统编程的方法，如下：

- 使用一种全新的用于并行编程的程序设计语言
- 修改现有的顺序程序设计语言的语法以构成新的并行程序设计语言
- 使用附有说明并行性编译器命令的现有顺序程序设计语言
- 在现有顺序程序设计语言中使用库例程
- 使用重量级进程
- 使用线程

人们也可使用规范的顺序程序设计语言，然后使用*并行编译器*将顺序程序转换成并行执行代码。在这里由编译器决定哪些语句可以同时执行，它还可重新调整语句的次序以实现并发操作，但必须保持程序员的最初意图。在20世纪70年代曾对该方法进行了广泛研究。但使用全新程序设计语言只获得了很有限的赞同，因为它要求人们从零开始来学习一种新的语言。在某种程度上被使用的一个例外语言是由美国国防部提倡的Ada语言。

对现有的顺序语言加以修改的方法有更大的吸引力，因为此时人们只需学习这些修改。最有号召力的修改方法是使用编译器命令和库例程而不是修改语法。一个已被接受的修改方法标准是OpenMP，但它仍需一个专门的编译器。

有趣的是，C++的发明者Stroustrup曾在[Wilson and Lu，1996]的序言中写到，他在设计最初的C++的规范的时候并没有将并发的特性放入其中，虽然他可以这么做。他的结论是“每种并发模型都只能服务于其中一小部分的用户”。他还有另一个结论，即“库方法存在弱点是因为库方法能提供比基于语言扩展方法更高的可移植性”。

在本章中我们将从传统的进程开始，然后介绍使用线程Pthreads标准的*线程*。Pthread目前已在多种平台上实现（单处理器的工作站系统和多处理机系统）。Java也是基于线程的，而且能够提供一些比这里所描述的更高级的特征。如果目的多处理机系统上已经有了Java实现的话，那么使用Java来进行基于线程的并行程序设计是最好的选择。

8.2 说明并行性的构造

8.2.1 创建并发进程

也许描述并发进程的结构的最早的例子是由[Conway，1963]所给出的FORK-JOIN语句组 232
（其中Conway引用了更早的工作，可能这个想法在1960年之前就出现了）。FORK-JOIN结构曾被应用于FORTRAN语言和UNIX操作系统的扩展。在最初的FORK-JOIN结构中，一个FORK语句产生一个新的并发进程的路径，并且并发进程在其结尾使用JOIN语句。当原进程和新产生的进程都到达JOIN语句后，代码继续以顺序的方式执行。如果需要更多的并发进程，则需要执行更多的FORK语句。图8-3中显示了多次嵌套的FORK-JOIN结构。每个产生的进程

都需要在结尾执行一个JOIN语句，从而将所有的并发进程归约到同一个终止点上，仅当所有产生的并发进程都完成，主进程才可以执行后续的语句。通常使用一个计数器来保持未完成进程的记录。FORK/JOIN结构本质上与消息传递系统中的派生/退出（spawn/exit）操作是相同的，并且可以是一个库/系统例程或者是一种语言结构。

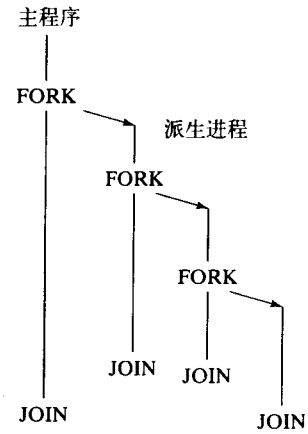


图8-3　FORK-JOIN结构

UNIX重量级进程（UNIX Heavyweight Process）

像UNIX这样的操作系统都是基于进程的概念设计的。在单处理器系统中，处理器必然要被多个进程分时共享；处理器的使用从一个进程切换到另一个进程。进程的切换可能是有固定的时间间隔，也可以是当活动进程的执行发生了停滞时发生。当一个进程因为某种原因而阻塞执行时，比如等待I/O操作的完成，则分时共享系统就可以将此进程调出处理器。在多处理机系统中，进程的执行可以真正地并发。UNIX系统提供创建进程的系统调用，从而可以使用这些工具来编写并行程序。当然，我们在单处理器上得不到随并发进程数的增加而相应提高的执行速度。（实际上，执行速度会因为创建进程和进程切换导致的上下文变换的开销而降低。）

UNIX系统调用fork()来创建新进程。除了唯一的进程ID外，被创建的新进程（子进程）是调用进程的完全拷贝（exact copy）子进程拥有其父进程所有变量的拷贝。当fork()被成
233 功调用时，向子进程返回0并向父进程返回子进程的进程ID。（如果失败，fork()返回给父进程－1，同时没有子进程被创建。）系统调用wait()（或waitpid()）和exit()被用于进行进程的“join”操作，wait()和exit()定义为：

```
wait(statusp);   /*delays caller until signal received or one of its */
                 /*child processes terminates or stops */
exit(status);    /*terminates a process. */
```

因此，一个子进程可以通过以下的方式被创建：

```
      :
pid = fork();                                        /* fork */
   Code to be executed by both child and parent
if (pid == 0) exit(0); else wait(0);                 /* join */
      :
```

（这里没有给出检测分叉（fork）是否出错的代码。）如果父进程先于子进程到达“join”点，则父进程将等待子进程结束；如果子进程先到达“join”点，则子进程终止。这个程序结构基本上是一种SPMD（单程序多数据）的模型。像这种模型的其他例子一样，我们通常使用控制语句来分隔不同进程的执行代码。如果子进程需要执行不同的代码，我们可以使用如下代码：

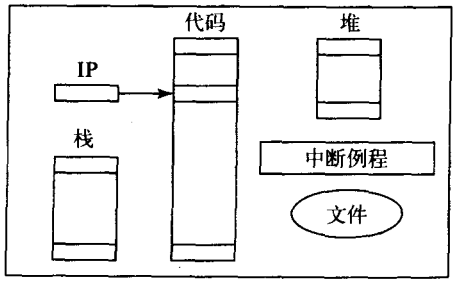
```
pid = fork();
if (pid == 0) {
   code to be executed by slave
} else {
   Code to be executed by parent
}
if (pid == 0) exit(0); else wait(0);
      :
```

每个进程都将对原程序中的所有变量进行复制，作为这个进程的本地变量；新进程的变

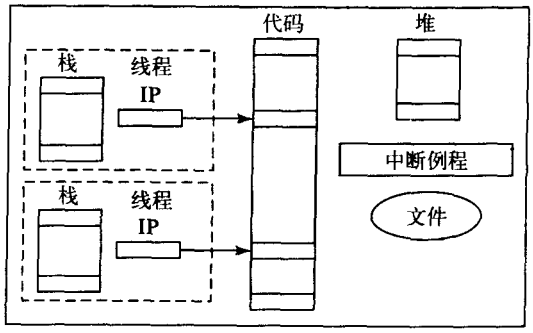
量都被赋予与原变量相同的值。新进程（派生进程）将在分叉点开始执行。

8.2.2 线程

使用UNIX fork创建的进程是一个“重量级”进程；它是一个完全独立的程序，拥有自己的变量、堆栈和存储器的分配。使用系统调用可使进程共享存储器空间（参见8.7节中的例子），但重量级进程在时间和存储器空间上（开销）都很昂贵。即使新创建的进程只从分叉“fork”点开始，一个拥有自己的存储器分配、变量、和堆栈的新进程还是需要对原进程进行完全的拷贝。通常来说，这种对原进程的完全拷贝是不必要的，一种更加高效的机制是在进程中定义称为线程[⊖]的独立并发序列。所有线程共享进程的同一存储器空间和进程的全局变量，因 234
而线程在时间和存储器空间上的开销比进程本身要小得多。图8-4中解释了进程和线程之间的区别。进程的一些最基本部分显示在在图8-4a中。指令指针（IP）存储的是下一条要执行的指令的地址，栈用于过程调用，还包括堆（heap）、系统例程和文件；而如图8-4b所示，进程中的每个线程都有自己的指令指针指向本线程需要执行的下一条指令，每个线程还有自己的栈并且各自存储一些关于寄存器的信息。而代码和其他部分则由各个线程共享。



a）进程



b）线程

图8-4 进程和线程之间的区别

创建一个线程的时间比创建一个进程的时间少三个数量级。此外，线程可以立即访问到共享的全局变量。同样重要的是，线程的同步可以比进程的同步更高效，因为进程的同步需要耗时的系统操作，而线程的同步操作则只需通过访问变量即可实现。我们将在后续讨论中讲述同步的问题。

生产商很早就在他们的操作系统中使用了线程机制了，因为线程提供了一个处理操作系

⊖ 有些学者认为轻量级进程和线程这两个术语是有区别的，把轻量经进程描述为一个内核线程或者是操作系统中的一种线程。

统内部并发活动的强大而又优美的解决方案。不论何时一个线程的活动时延或者阻塞，比如等待I/O，则另一个线程就可以接手使用处理器。多线程操作系统的实例包括SUN Solaris、IBM AIX、SGI IRIX和Windows XP。在这些操作系统里都有某些供用户在他们的程序中使用线程的设施，而不同的系统其设施有所不同。幸运的是，现在存在着一个线程机制的标准，*Pthread*（来源于IEEE可移植操作系统接口（IEEE Portable Operating System Interface，POSIX）1003.1节），它已被广泛采用。我们将主要针对于Pthread展开讨论。在附录C中提供
235 了一个Pthread例程的简略的列表。

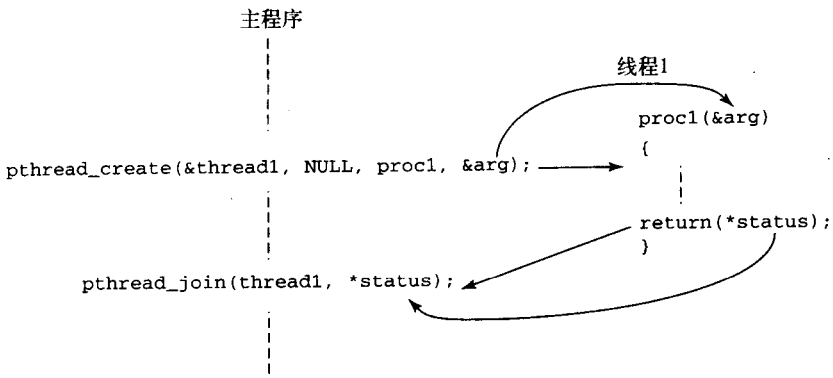
多线程也能够减少消息传递中产生的高时延，因为当一个线程在等待消息的时候，系统可以快速的从一个线程切换到另外一个线程，从而提供了一个强有力的机制来实现时延隐藏。Solaris线程机制中则有一些消息传递的例程，但Pthread并未提供这些例程[Sunsoft，1994]。

1. 执行Pthread线程

在Pthread规范中，主程序本身也是一个线程。如图8-5所示，一个单独的线程可以用以下例程来创建和终止：

```
pthread_t thread1;              /* handle of special Pthread datatype */
      ⋮
pthread_create(&thread1, NULL, (void *) proc1, (void *) &arg);
      ⋮
pthread_join(thread1, void *status);
      ⋮
```

新线程开始执行例程proc1，并接受一个参数，在本例中为&arg。（pthread_create()中的NULL参数将导致系统使用默认的线程“属性”。）系统将分配一个线程ID或者“句柄”，并通过在对该线程的后续引用中使用的&thread1获得。如果调用pthread_join()的时候，新创建的线程还没有完成，则它被用于pthread_join()中促使调用线程等待新产生的线程终止。线程一旦终止，则被系统撤销，释放其占用的资源。而线程的完成状态可以在pthread_join()中返回（*status）。如果该线程返回值，则这个终止状态就可以被使用（该值必须被类型转换为void）。如果线程不需要返回任何的值，则指定为NULL。一个线程也可以通过在其例程的终点自然终止（return()），并随之返回其终止状态；它也可以通过调用pthread_exit(void*status)例程来终止和撤销；它可以被其他的进程撤销（“取消”），在这种情况下，其返回的状态值将是PTHREAD_CANCELED。如果一个线程需要返回一定的值，则需要注意在某些情况下可能与PTHREAD_CANCELED搞混。



236 图8-5　pthread_create()和pthread_join()

pthread_join()例程用于等待一个特定的线程的终止。要创建一个等待所有线程的障栅，可重复使用pthread_join()例程:

```
        :
        :
for (i = 0; i < p; i++)
  pthread_create(&thread[i], NULL, (void *) slave, (void *) &arg);
        :
        :
for (i = 0; i < p; i++)
  pthread_join(thread[i], NULL);
        :
        :
```

我们创建一个线程ID的数组，并使p个从线程一起被创建和终止。一个线程可以通过调用例程pthread_self()来获得它的线程ID，通过使用pthread_equal(thread1,thread2)例程比较线程的ID就可以识别特定的线程。其中，thread1和thread2是线程的ID。

2. 分离线程

当一个被创建的线程终止的时候，可以不用通知它的创建线程，从而创建线程就无需连接（join）。这种无需连结的线程就称为*分离线程*（detached thread），如图8-6所示。这种分离线程终止的时候，会被撤销并释放它们所占用的资源。在创建线程时通过指定线程属性可得到一个分离线程。在创建者无需等待被创建线程终止的情况下，就应使用分离线程，因为这将有更高的效率。

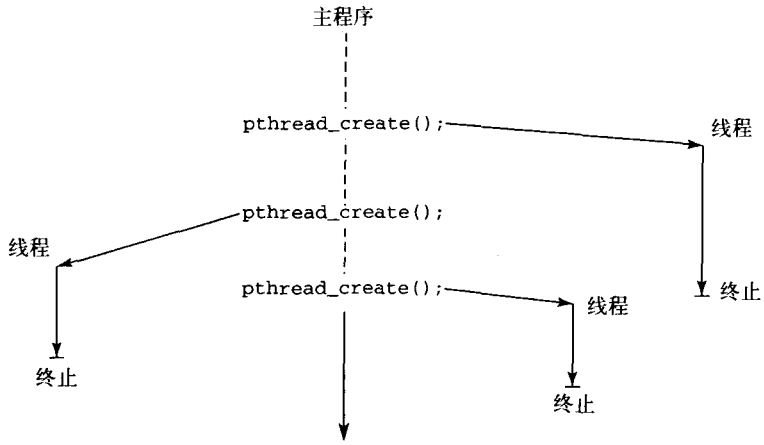


图8-6 分离线程

3. 线程池

所有以前应用于进程的机制对于线程都是适用的。一个主线程需要控制一系列的从线程。可以构成一个线程池，线程可以通过共享单元或者用信号来互相通信。 237

4. 语句执行顺序

一旦一个进程或者线程被创建后，它们的语句执行顺序就由系统来决定了。在单处理器的系统上，处理器由多个进程或线程分时共享，如果事先没有指定，则语句执行顺序由系统来决定。不过通常情况下，如果一个线程不阻塞的话，它将一直执行直到结束。在多处理机系统上，则会有不同的进程/线程同时在不同的处理器上执行的情况出现。不管在哪种情况下，我们必须注意到各个进程/线程的指令可能在时间上是交叉的。比如，如果有带有如机器指令的两个进程:

进程1　　进程2
指令1.1　　指令2.1
指令1.2　　指令2.2
指令1.3　　指令2.3

则有很多种可能的顺序，包括:

指令1.1
指令1.2
指令2.1
指令1.3
指令2.2
指令2.3

假设这些指令不可再分为更小的可中断步。比如，如果两个进程都要打印一条消息，则这两条消息可以以任一种顺序出现，这依赖于调用打印例程的这两个进程的调度情况。更坏的情况是，如果打印例程的实例的机器指令也可以交叉执行的话，则这两条消息的各个字符也可能交叉打印出来。

除了在进程/线程中交叉执行机器指令导致的进程/线程语句执行顺序的变化外，编译器（或者处理器）也可能因为优化执行的原因在保持程序逻辑上正确的前提下重新安排指令的执行顺序。比如，如下语句:

```
a = b + 5;
x = y + 4;
```

可能在编译后以颠倒的顺序执行，而仍然保持其逻辑正确性:

```
x = y + 4;
a = b + 5;
```

因为正在处理器执行的前一条指令需要更多的时间才能得到b的值，所以将`a = b + 5;` 拖后
238 执行是有好处的。在现代的超标量处理器中，经常以乱序方式执行机器指令以提高执行速度。

5. 线程安全的例程

如果一个系统调用或者库例程/库函数同时被多个线程调用的情况下仍然都能得到正确的结果，则这个系统调用或者库例程称为是*线程安全*的。例如，能够保持完整的消息被打印出来而不会出现字符交叉的打印例程就是线程安全的。幸运的是，标准I/O操作都被设计为线程安全的。然而，那些访问共享数据或者静态数据的例程就需要加以特殊的处理以达到线程安全。例如，返回时间的系统例程就不一定是线程安全的。所有线程安全的例程的列表可以在Ptheads参考书中找到，如[Kleiman，Shah，and Smaalders，1996]。事实上，除了那些在技术上难以成为线程安全的例程之外，几乎所有POSIX例程被定义成是线程安全的。虽然一般而言函数的线程安全依赖于操作系统。我们可以通过强制在任何时间点只有一个线程执行某个例程来避免例程的线程安全问题，这可以通过将例程简单的包裹在一个临界区（参见8.3.2节）中来实现，但这种方法是非常低效的。

8.3 共享数据

共享存储器编程的主要特征是共享存储器提供了创建出能够直接被处理器访问而不用像消息传递环境中那样用消息来传递数据的变量和数据结构。

8.3.1 创建共享数据

如果要让UNIX中的重量级进程共享数据，额外的共享存储器系统调用是必要的。通常，每个进程在虚拟存储器管理系统中都有自己的虚拟地址空间。共享存储器系统使得进程能将一段物理存储器链入自己的虚拟地址空间内。调用共享存储器段通过系统调用`shmget()`（“获得共享存储器段标识符”）来创建，这个系统调用返回一个共享存储器标识符。一旦共享存储器段创建，就可以使用系统调用`shmat()`来将这个共享存储器段链入成为调用进程的数据段，`shmat()`返回的是这个数据段的起始地址。在8.7.1节中可以看到使用这些调用的代码序列。

如果使用线程，则没有必要显式地创建共享数据项。在主程序（主线程）的顶部声明的变量是全局的，可以被所有线程使用。而在例程的内部声明的变量自然是局部的。

8.3.2 访问共享数据

如果共享数据被进程所改动，则在访问这些共享数据的时候就需加倍小心地控制（我们使用*进程*这个术语，其实所有情况也适用于线程）。各个进程读同一个共享变量不会导致冲突，而写新值则会导致冲突。考虑有两个进程，每一个都要对一个共享的数据项x加1，为了对x加1我们可以将代码写成是`x++;`或者`x = x + 1;`。不管使用哪种写法，都会进行：将x单元的内 239
容读出，计算`x + 1`，将结果写回到该单元这三个步骤。如果两个进程几乎同时进行这个操作，则如图8-7所示，会有：

指令	进程1	进程2
`x = x + 1;`	`read x`	`read x`
时间 ↓	`compute x + 1`	`compute x + 1`
	`write to x`	`write to x`

假设初始时x的值为10。则当进程1和进程2都写回加1的结果后，预期的x值应该等于12。而实际情况是，两个进程都从x中读出10，并最终都写回11。在共享数据库中，当多个进程对共享数据同时执行算术操作时就会出现上述的情况。比如，在[Nichols, Buttlar and Farrell, 1996]中就提出了一个例子——两个自动取款机（ATM）的问题（如果一个被丈夫使用，而另一个则同时被妻子使用）。来自不同源的自动借方也会出现类似情况。

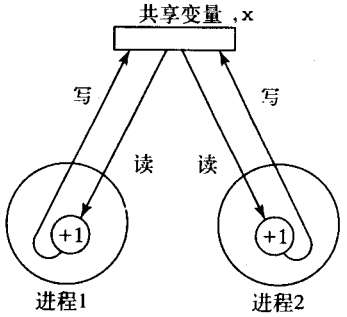


图8-7 访问共享变量时产生的冲突

访问共享数据的问题可以归纳为共享资源的访问问题。除了共享数据之外，资源还可以是物理设备，比如输入输出设备。确保一次只能有一个进程访问特定的共享资源的机制是，建立涉及到这个共享资源的代码段即所谓的*临界区*(critical section)，并使得一次只能有一个临界区被执行。第一个为特定资源到达临界区的进程可以进入并执行临界区。一旦有一个进程进入临界区，则其他进程就不能再进入了，直到这个进程完成临界区，才允许另一进程进入同一资源的临界区。这种机制就是所谓的*互斥*（mutual exclusion）。

1. 锁机制

确保临界区互斥的最简单的机制就是使用锁机制。一个锁是一个1位的变量，这个变量的

值是1表示有一个进程进入了临界区；如果是0，则表示没有进程在临界区中。这种锁的操作
240 类似于普通的门锁。一个进程来到临界区的“门”，发现“门”开着，它就进入这个临界区，并将“门”关上以防止其他的进程进入。一旦进程完成了临界区，它就打开“门”并离开。

假设一个进程执行到了一个锁，且这个锁已经置位，则这个进程将被临界区排斥。它于是必须等待，直到被允许进入临界区为止。进程在一个紧凑的循环中不停地检查锁位，例如：

```
while (lock == 1) do_nothing;        /* no operation in while loop */
lock = 1;                            /* enter critical section */

  critical section

lock = 0;                            /* leave critical section */
```

这样的锁叫做*自旋锁*（spin lock），而这种机制就叫做*忙等待*（busy waiting）。图8-8显示了通过锁机制而实现的临界区串行化。忙等待机制在处理器的利用上是低效的，因为在等待锁被释放的过程中进程没有做任何有用的工作。在某些情况下，我们其实可以将处于忙等待的进程调出处理器，并调入其他的进程。当然，这种做法会在保存和读取进程信息时产生一定的开销。另外，如果有多于一个进程在等待锁的释放，则当在锁释放的时候，应有一种机制来选择最好的或者最高优先级的进程先进入临界区，而不是通过忙等待机制不确定地选择进入临界区的进程。

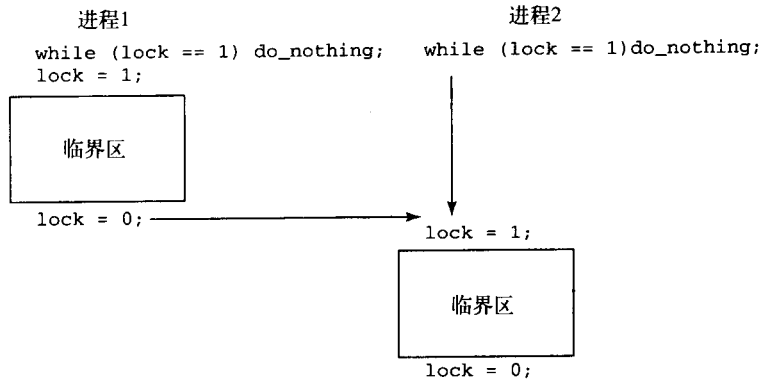


图8-8　用忙等待控制临界区

确保不能有多于一个的进程同时进入临界区是很重要的。类似地，如果一个进程发现锁开着，而在它关闭锁之前，另一个进程也发现锁开着，此时必须保证两个进程不会同时进入它们的临界区。因此，检查锁是否开着并关闭锁的操作应该实现为是不可中断的。在此过程中，任何其他的进程都不能对锁进行操作。这种互斥机制通常采用不可分的机器指令以硬件方式（如测试置位指令）来实现。尽管锁机制也可以不使用这种机器指令的方式来实现（参
241 见[Ben-Ari，1990]）。在后续的讨论中，我们说“加锁”或者“解锁”，意味着这些操作的执行是*原子性*（即不可中断的操作）的；也就是说，在这些操作的执行过程中，没有其他的进程能够访问该锁，或者影响这些操作的执行结果。

Pthread锁例程　在Pthread规范中，锁通过所谓的*互斥锁*（mutually exclusive lock或“mutex”）变量来实现。为了使用互斥锁，首先它必须在“主”线程中声明为pthread_mutex_t类型并初始化：

```
pthread_mutex_t mutex1;
```

```
pthread_mutex_init(&mutex1, NULL);
```

NULL表明使用互斥锁的默认属性。互斥锁也可以使用malloc方法动态地创建，并通过pthread_mutex_destroy()例程被撤销。通过使用pthread_mutex_lock()和pthread_mutex_unlock()可以实现对临界区的保护:

```
        :
pthread_mutex_lock(&mutex1);

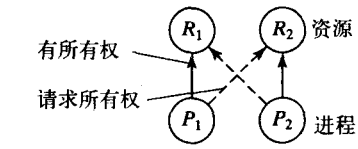
  critical section

pthread_mutex_unlock(&mutex1);
        :
```

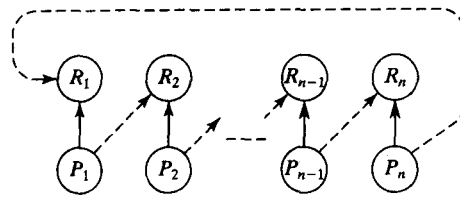
如果一个线程到达互斥锁并发现互斥锁关闭，则它将等待直到互斥锁打开。如果多个线程等待互斥锁打开，则当其打开的时候，系统会选择一个线程来进入临界区。只有加锁互斥锁的线程能够进行解锁。(如果其他线程尝试解锁的话，则它会得到一个出错状态。)

2. 死锁

使用锁机制时一个重要的考虑是要避免死锁，因为死锁会导致进程无法执行下去。对两个进程而言，当一个进程需要由另一个进程保持的资源而这个进程反过来也需要由这个进程保持的资源的时候就会发生死锁，如图8-9a所示。在图8-9a中每个进程已经获得了两个资源中的一个，除非某个进程释放另一个进程所需的资源，否则两个进程都将永远停滞而没有进程可以继续。死锁也可能以循环的方式出现。这里牵涉到多个持有另外一个进程所需资源的进程，如图8-9b所示。进程P_1申请访问进程P_2持有的资源R_2；进程P_2则申请访问进程P_3持有的资源R_3；以次类推进程P_n申请访问进程P_1所持有的资源R_1；这样，就形成了一个死锁环境。这种特殊形式的死锁就叫做*死亡拥抱*(deadly embrace)。假设给定一组持有不同资源请求的进程，如果在其中的某些进程之间的资源请求构成了环路，则这就意味着有潜在的死锁可能。对于访问多个资源两个进程而言，如果它们都以相同次序先请求某个资源，然后再请求另一个资源，就可以消除死锁的可能。



a) 两个进程死锁



b) n个进程死锁

图8-9 死锁(死亡拥抱)

Pthread规范提供一个例程pthread_mutex_trylock()，它可以不阻塞线程来测试一个锁是否真的关闭。这个例程可以将一个未加锁的互斥锁加锁并随后返回0；如果这个互斥锁已经加锁，则它会返回EBUSY。这个例程对解决死锁的问题是有一定的帮助的。

3. 信号量

信号量(Semaphore)的概念是由[Dijkstra，1968]发明的，它是一个非负的整数。信号量有两种操作，分别命名为**p**操作和**v**操作。**p**操作作用在信号量s上(写做**p**(s))，使进程等到s变为大于0后，将s减1，此后才允许进程的继续执行。**v**操作则将s加1，如果存在等待s的进程，则将其中一个释放。**p**操作和**v**操作都是不可分的操作。(字母**p**来自于荷兰语的单词*passeren*，意思是“通过”；而字母**v**则来自于荷兰语的单词*vrijgeven*，意思是“释放”。)

242

p操作和**v**操作中的唤醒等待进程的机制也是隐式的。虽然这个机制所使用的实际算法没有给出，但是应该至少能够达到公平。被**p**(s)操作停滞的进程应被搁置直到由同一信号量上的**v**(s)操作释放。进程可以使用自旋锁（忙等待）停滞，而更好的方法则是将它们调出处理器，让给其他已就绪的进程。

访问同一资源的多个进程临界区的互斥可以通过使用只有0或者1值的信号量（称为二元信号量（binary semaphore））来实现。这种信号量起锁变量的作用，而**p**和**v**操作包含一个进程调度的机制。这种信号量初始化为1，表示目前没有与此信号量相关的进程处在临界区里。每个互斥的临界区都以**p**(s)开头，并以同一个信号量的**v**(s)结尾，即:

进程1	进程2	进程3
非临界区	非临界区	非临界区
⋮	⋮	⋮
p(s)	**p**(s)	**p**(s)
临界区	临界区	临界区
v(s)	**v**(s)	**v**(s)
⋮	⋮	⋮
非临界区	非临界区	非临界区

其中任何一个进程都可能先执行到它的**p**(s)操作（或者多个进程可能同时到达**p**(s)操作）。第
243 一个执行到**p**(s)操作的进程，或者信号量检查认可的进程，会把信号量设为0，禁止其他的进程继续它们的**p**(s)操作。这些到达它的**p**(s)操作的任何进程都会被记录下来，以便将来临界区释放的时候进行选择。被认可的进程则可以执行它的临界区。当这个进程执行到**v**(s)操作的时候，就会将信号量s设为1，并从等待的进程中挑选出一个，使之进入临界区。

更为一般的信号量（或者称为计数信号量）可以采用正整数而不是0和1。这种信号量提供了一种记录可用的或者已用的“资源单元”数目的方法，并且可以用于解决生产者/消费者问题。

UNIX进程中有信号量例程，而在Pthread规范中则不存在，尽管我们可以实现。但在Pthread的实时系统扩展中有信号量例程。在8.7.1节中我们将给出UNIX信号量的代码的示例。概要地说，我们可以使用`semget(key,nsems,semflg)`来创建信号量。这个例程返回与key相关的信号量标识符。`Semctl()`调用则用来设置信号量的值，而**P**和**V**操作是用`Semop()`信号量操作来实现的，它们在信号量数组上完成原子的操作。

4. 监控程序

众所周知，虽然信号量能够应用于大多数临界区问题的处理，但是却很容易招致人为的使用错误。对于一个特定信号量上每个**p**操作，都必须要有一个同一信号量上的**v**操作的对应，而这个**v**操作可能是由另外的进程来执行的。遗漏一个**p**操作或者**v**操作的使用，或者使用到错误的信号量上面，都会产生严重的结果。一种更高级的技术是使用*监控程序*[Hoare，1974]。监控程序是一组过程，由它提供对某个共享资源的唯一访问方法。基本上所有的数据和对该数据的操作都被封装在一个结构里。对数据的读和写都只能由监控程序的过程来完成，并且在任何时间只能由一个进程使用监控程序的过程。如果在某个进程执行监控程序过程的时候，另一个进程请求使用，则这个请求的进程将被挂起，并放入等待队列中。当活动进程对监控程序的使用结束后，等待队列中的第一个进程（如果有的话）将被允许使用一个监控程序的过程。

一个监控程序过程使用保护它入口的一个信号量来加以实现，即:

```
monitor_proc1()
{
   P(monitor_semaphore);

      monitor body

   V(monitor_semaphore);
   return;
}
```

在Java中就有监控程序的概念。Java中的关键词synchronized可以使方法中的一段代码成为线程安全的，它可防止在该方法中有多于一个的线程。在8.7.3节中给出了使用Java监控程序方法的一个简单的程序。读者可以参考很多有关Java的书籍以了解更多的细节。

5. 条件变量

有些临界区通常需要特定的全局条件满足了以后才可以执行，例如，某个变量达到了一个特定的值。如果使用锁机制来实现，则我们就需要在临界区内频繁地检查这个全局变量的值（“查询”，polled），显然这是很耗时而又无效的。所以，我们可以使用一种出现在监控程序的上下文内部的称为条件变量（condition variables）的机制来克服这个问题。为条件变量定义了如下三种操作： 244

```
Wait(cond_var)   ——等待条件出现
Signal(cond_var) ——条件已经出现的信号
Status(cond_var) ——返回等待条件出现的进程的个数
```

等待操作在执行过程中将释放锁或者信号量，并且可以用于让其他的进程来改变这个条件。当调用wait()的进程被允许执行下去时，锁或信号量被重新设置。下面将会看到为什么解锁再加锁是必须的。在调用signal()前，要由程序来识别条件是否满足。

作为使用条件变量的例子，考虑一个或多个设计为当计数器x的值为0时采取行动的进程（或者线程）。另一个进程（或者线程）负责计数器的递减。该例程形式如下：

```
action()                          counter()
{                                 {
     ⋮                                 ⋮
   lock();                           lock();
   while (x != 0)                    x--;
     wait(s);  <-------------------  if (x == 0) signal(s);
   unlock();                         unlock();
   take_action();
     ⋮                                 ⋮
}                                 }
```

在counter例程和action例程中，使用同一个锁来对共享的计数器变量x的访问进行控制。我们可以假设action例程先执行到其临界区，因为即使counter例程先执行到它的临界区，发出的信号由于不被记忆而可能丢失。在action例程里，wait()将解锁并等待被信号s释放。当signal()产生信号s后，wait()被释放。action例程中的while语句用来再检查这个条件是否满足，即便这个条件假设已经成立。这种“双重检查”通常是必要的，因为它可以用来进行出错检查，并且如果有多个线程/进程可以同时被唤醒或者其他的信号可能已唤醒线程/进程的情况下，这种双重检查特别重要。

Pthread的条件变量 Pthread中提供了与指定互斥锁相关的条件变量。为了在程序中使用

条件变量，我们通常也是在“主”线程中说明一个pthread_cond_t类型的变量，并将其初始化：

```
pthread_cond_t cond1;
pthread_cond_init(&cond1, NULL);
```

NULL用来说明将使用互斥锁的默认属性。条件变量可以通过使用pthread_cond_destroy()例程
245 来撤销。

Pthread中提供了两个例程来使一个线程等待某个条件变量的信号：

```
pthread_cond_wait(cond1, mutex1);
pthread_cond_timedwait(cond1, mutex1, abstime);
```

例程pthread_cond_wait()负责将调用线程挂起，直到另外的线程发出此条件变量的信号并将指定的互斥锁解锁。（这两个动作都是“原子性”的。）当该信号被接收后，互斥锁被加锁并且调用返回。例程pthread_cond_timedwait()除了在系统时间达到或者超过了abstime时也会返回外与pthread_cond_wait()例程类似。

Pthread规范中提供了两个例程用来从调用线程向另一个线程发送信号释放它：

```
pthread_cond_signal(cond1);
pthread_cond_broadcast(cond1);
```

pthread_cond_signal()例程释放一个因为等待条件变量cond1而被阻塞的线程。例程pthread_cond_broadcast()则将所有等待条件变量cond1而被阻塞的线程唤醒。然而，只有一个处于等待状态的线程能够获得互斥锁；而其他的则被置于等待互斥锁的状态。

给定变量的声明和初始化：

```
pthread_cond_t cond1;
pthread_mutex_t mutex1;
      ⋮
pthread_cond_init(&cond1, NULL);
pthread_mutex_init(&mutex1, NULL);
```

Pthread的信号和等待如下：

```
action()                                counter()
{                                       {
          ⋮                                    ⋮
pthread_mutex_lock(&mutex1);            pthread_mutex_lock(&mutex1);
while (c !=0)                           c--;
  pthread_cond_wait(cond1,mutex1);      if (c == 0) pthread_cond_signal(cond1);
pthread_mutex_unlock(&mutex1);          pthread_mutex_unlock(&mutex1);
take_action();
          ⋮                                    ⋮
}                                       }
```

对信号将不加以记忆，这意味着线程必须已经是在等待信号来接收它。

6. 障栅

如消息传递系统一样，在共享存储器程序中常需要进程/线程的同步。Pthread不具有内在
246 的障栅（POSIX 1003.1j扩展版除外），因此就必须使用条件变量和互斥锁手工编写障栅，有

关的全部细节已超出本书范围。在[Butenhof, 1997]、[Kleiman, Shah, and Smaalders, 1996]和[Prasad, 1997]中可找到有关的编程举例。障栅实现常使用在6.1.2节中所描述的集中式计数器方法。每当一个线程到达障栅时就使全局计数器加1，而当计数器到达所定义的线程数时就释放所有线程。由最后一个到达的线程发出广播信号pthread_cond_broadcast()并待其他正等待的线程接收该信号后，就实现了对所有进程的释放（在一个循环中使用pthread_cond_wait()）。然后将计数器置位成0以供障栅的下一次使用。

如同消息传递系统中的障栅一样，必须考虑障栅可被多次调用，而且实现时必须处理其他线程还没有第一次离开障栅之前一个线程已第二次进入障栅的情况。在第6章中我们已看到一种具有到达障栅和离开障栅两阶段的设计方法，它可防止这种情况的出现。

[Butenhof, 1997]提出了一种不同的实现，他使用一个与障栅有关的称为count的变量，用来计数到达障栅的线程，并使用另一个取值为0或1 的、名为cycle的变量。当线程到达障栅时，保存在每个线程中的变量cycle是一个局部变量。当障栅的一个周期结束时由最后一个到达障栅的线程将变量cycle倒置（即从0变为1或从1变为0）。仅当存放在线程中的cycle值不同于cycle的实际值时线程才被释放（pthread_cond_wait()在while的循环中，仅当cycle局部值与cycle值不同时循环才会终止。）这里没有涉及共享存储器障栅编程的其他方面，例如，如何避免在线程未被创建之前和被初始化之前去访问障栅这样的错误条件。详情可在[Butenhof, 1997]中找到。

8.4 并行程序设计语言和构造

8.4.1 并行语言

使用专门设计的并行程序设计语言似乎一直有所呼吁，特别是对共享存储器系统。共享存储器变量可以如同程序中任何变量那样加以说明和访问。并行程序设计语言提供了一个高层次的抽象，且能隐蔽实际计算平台体系结构的某些细节。并行程序设计语言的发展已有很长历史。在这些年中，已提出了许多并行程序设计语言，但没有一个能被普遍接受。表8.1列出了几种早期开发的并行程序设计语言。[Bal, Steiner, and Tanenbaum, 1989]列出了直至1989年的、大量的有关系统/语言的参考文献。某些语言是为开发通用控制并行性而设计的（即不同进程中的指令是分别加以控制的）。而另一些语言则专门是为了开发数据并行性（即由一条指令指明针对一组数据项的相同操作）。[Karp and Babb, 1988]描述了12种并行Fortran语言。[Foster, 1995]则详细叙述了3种并行程序设计语言：Compositional C++、Fortran M以及高性能Fortran (HPF)。由[Wilson and Lu, 1996]编辑的巨著中，详尽叙述了编写并行程序的 247
15种不同语言，它们都将C++作为基本语言。（其中只有一种语言是面向共享存储器多机系统的。）在20世纪80年代和90年代所提出的所有语言中，到目前仅有HPF语言还能在某些场合下看到。

人们为并行程序设计提供语言扩展仍有不断增长的兴趣。最近的一个例子是Unified Parallel C (UPC)，它是对C的扩展，由学术界、工业界及政府联盟开发（参见http://www.gwu.edu/~upc）。这种团队的努力有望被接受。UPC对原来的基本语言只作了相对较少的扩展，并主要是面向机群那样的分布式共享存储器系统（参见第9章）。与其选择如UPC那样的语言扩展，不如让我们简短地回顾出现在这种扩展中的，或出现在针对共享存储器系统的并行程序设计语言中的这些构造，以及使用这些构造的语言/扩展。

表8-1　一些早期的并行程序设计语言

语　言	创始者/时间	注　解
Concurrent Pascal	Brinch Hansen, 1975①	Pascal的扩展
Ada	U.S. Dept. of Defense, 1979②	全新语言
Modula-P	Bräunl, 1986③	Modula 2的扩展
C*	Thinking Machines,1987④	SIMD系统的C扩展
Concurrent C	Gehani and Roome, 1989⑤	C扩展
Fortran D	Fox et al., 1990⑥	用于数据并行程序设计的Fortran扩展

① Brinch Hansen, P. (1975), "The Programming Language Concurrent Pascal," *IEEE Trans. Software Eng.*, Vol.1, No.2(June), pp. 199-207.

② U.S. Department of Defense (1981), "The Programming Language Ada Reference Manual," *Lecture Notes in Computer Science*, No.106, Springer- Verlag, Berlin.

③ Bräunl, T., R. Norz(1992), *Modula-P User Manual*, Computer Science Report, No.5/92 (August). Univ. Stuttgart, Germany.

④ Thinking Machines Corp. (1990), *C* Programming Guide, Version 6*, Thinking Machines System Documentation.

⑤ Gehani, N., and W. D. Roome (1989), *The Concurrent C Programming Language*, Silicon Press, New Jersey.

⑥ Fox, G., S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu (1990), *Fortran D Language Specification*, Technical Report TR90-141, Dept. of Computer Science, Rice University.

8.4.2 并行语言构造

1. 共享数据

在支持共享存储器的并行程序设计语言中，变量可以被定义成是共享的，如

```
shared int x;
```

或者，如果是一个指针

```
shared int* p;
```

其中，p是一个指向共享整数的指针。这类说明并不必定意味这个变量可被多个进程同时访
248 问；它们只是简单地表示任何进程可以访问这一变量。一个合适的机制需在适当的位置去保证实际上每次仅能有一个进程访问该变量。UPC有如上的共享说明（shared），且还有对数组的共享说明，而数组中的元素被分布在不同的线程中。共享的概念也能在一个面向对象语言中被扩展成为共享对象。

2. par构造

并行语言提供了指明并发语句的可能性，如在par构造中:

```
par {
   S1;
   S2;
    :
   Sn;
}
```

关键字par指明体中的语句需并发执行。这是指令级的并行性。在指令级的并行性中，并发进程可仅为一个语句。在许多系统中，单语句可导致无法接受的开销，虽然构造允许这种可能性。

多个并发进程或线程可以通过列出需被并发执行的例程来加以说明:

```
par {
   proc1();
   proc2();
      ⋮
   procn();
}
```

其中，proc1()，proc2()，…，procn()，如果可能的话可并发执行。

在许多并行语言中都有par{…}构造；例如CC++[Foster，1995]。对于类Pascal并行语言，我们可以看到PARBEGIN…PAREND构造或COBEGIN…COEND构造。更早的例子可在ALGOL-68中找到。由逗号而不是分号分割开的语句（或复合语句）的执行顺序未加定义；即在一个单处理器系统中，这些语句可按任何次序执行，而在一个多处理机系统中可同时执行。

3. forall构造

有时多个相似进程需要一起开始执行。这可以用forall构造（或parfor构造）实现：

```
forall (i = 0; i < p; i++) {
   S1;
   S2;
    ⋮
   Sm;
}
```

249

它将生成p个进程，每个由构成for循环体的语句S_1，S_2，…，S_m所组成。每个进程使用一个不同的i值。例如，

```
forall (i = 0; i < 5; i++)
  a[i] = 0;
```

将并发地把a[0]、a[1]、a[2]、a[3]和a[4]清为0。在[Terrano, Dunn, and Peters，1989]中可找到基于C语言的并行语言的forall构造的一个例子。在CC++[Foster, 1995]中也可发现类似的parfor构造。高性能Fortran（HPF）也有forall构造。UPC的forall构造还有一个指明体如何在线程中分布的独特特征。

8.4.3 相关性分析

并行程序设计中的一个关键问题是要确定哪些进程可在一起执行。当使用一个并行程序设计语言时人们希望编译器能指出阻止并发执行的问题所在。如果进程之间有相关性，则这些进程就不能一起执行，而必须顺序执行。寻找程序中相关性的过程就叫做*相关性分析*（dependency analysis）。比如，我们马上就可以从下面的代码中发现这个forall执行体的每个实例都是独立于其他实例的，它们可以同时执行：

```
forall (i = 0; i < 5; i++)
   a[i] = 0;
```

然而，像下面的代码这种关系就不是很明显：

```
forall (i = 2; i < 6; i++) {
   x = i - 2*i + i*i;
   a[i] = a[x];
}
```

在这种情况下，体中的不同实例是否能同时执行就很不明显。为此我们需要一种更可取的算法性的方法来识别实例之间的相关性，这种方法可由*并行化编译器*（parallelizing compiler）（一种能将顺序代码转换成并行代码的编译器）加以使用。

Bernstein条件

[Bernstein，1966]设立了一组条件，这组条件足以决定两个进程是否可以同时执行。我们可以将这些条件演绎成简单的形式，这些条件与在进程执行中由用来存放这些被读出或被改变的变量的进程所使用的存储单元有关。首先我们定义两组存储单元，I（输入）和O（输出），使得：

I_i 是进程P_i读取的所有存储单元的集合。

O_i 是进程P_j改变的所有存储单元的集合。

对于要同时执行的两个进程P_1和P_2，P_1的输入一定不能是P_2输出的一部分，并且P_2的输入也
250 一定不能是P_1的输出的一部分，即

$$I_1 \cap O_2 = \phi$$
$$I_2 \cap O_1 = \phi$$

其中∅表示空集。同时两个进程的输出集合也必须不同：

$$O_1 \cap O_2 = \phi$$

上述三个条件就是我们所说的Bernstein条件。

如果这三个条件全都满足，那么两个进程就可并发执行。这些条件可以应用于任意复杂性的进程情况。进程可以是一条语句，此时允许我们决定两个语句是否可同时执行。在这种情况下，I_i就对应于语句右边的变量，而O_j则对应于语句左边的变量。

例：　假设有两条语句（以C语言表示）：

```
a = x + y;
b = x + z;
```

则我们有

$$I_1 = (x, y)$$
$$I_2 = (x, z)$$
$$O_1 = (a)$$
$$O_2 = (b)$$

并且Bernstein条件

$$I_1 \cap O_2 = \phi$$
$$I_2 \cap O_1 = \phi$$
$$O_1 \cap O_2 = \phi$$

满足。因此，这两条语句a = x + y和b = x + z就可以同时执行。如果我们假设有以下两条语句：

```
a = x + y;
b = a + b;
```

由于$I_2 \cap O_1 \neq \varnothing$，故这两条语句就不能同时执行。

这种方法可以扩展为确定多个语句是否能够并行执行。Bernstein条件可以集成到编译器中以自动地实现对程序的检查，同时也可由程序员人工地执行。这些条件都是非常普通的，在寻找并行性时，无需使用任何特殊的计算。它们可用来识别指令级并行或粗粒度的并行性。粗粒度并行性考虑的是一组例程之间的并发操作问题。在那种情况下，例程的参数是输入，
251 而返回的变量/值则是输出。

一些普通的程序设计结构具有自然的编译器或程序员能加以利用的并行性，特别是在程序的循环中。例如，C循环

```
for (i = 1; i <= 20; i++)
   a[i] = b[i];
```

可以扩展为

```
a[1] = b[1];
a[2] = b[2];
   ⋮
a[19] = b[19];
a[20] = b[20];
```

当给定20个处理器时，这些语句便可同时执行（Bernstein条件满足）。

有时将循环分解成多个相互独立的小循环以处理循环中的相关性。例如，C循环

```
for (i = 3; i <= 20; i++)
  a[i] = a[i-2] + 4;
```

计算

```
a[3]  = a[1] + 4;
a[4]  = a[2] + 4;
   ⋮
a[19] = a[17] + 4;
a[20] = a[18] + 4;
```

这里a[5]只有在a[3]得到后才能进行计算，a[6]则只有在a[4]得到后才能计算，以此类推。这个计算过程可被分为以下两个独立的序列:

```
a[3]  = a[1] + 4;          a[4]  = a[2] + 4;
a[5]  = a[3] + 4;          a[6]  = a[4] + 4;
   ⋮                          ⋮
a[17] = a[15] + 4;         a[18] = a[16] + 4;
a[19] = a[17] + 4;         a[20] = a[18] + 4;
```

或者写为两个for循环:

```
for (i = 3; i <= 20; i+=2) {
  a[i] = a[i-2] + 4;
}
```

和

```
for (i = 4; j <= 20; i+=2) {
  a[i] = a[i-2] + 4;
}
```

252

Bernstein条件可以用来识别这两个循环。除此之外，并行编译器还可以使用许多其他技术来识别或创建并行性。我们可以在[Wolfe，1996]中找到用于并行编译器的各种技术的详细内容。

8.5 OpenMP

在前面两节中，我们已讨论了在并行程序设计语言中用来说明并行性的语言构造。有代表性的是，这些构造是对现有顺序语言的扩展。尽管这些并行程序设计的构造显示出是一种很吸引人的方法，而且有几种顺序语言的扩展已开发了多年，但它们只获得了很有限的成功。另一种方法是从一般的顺序程序设计语言出发，但通过明智地使用嵌入的编译器命令来建立并行说明。这些编译器命令可说明在8.4.2节中描述过的par和forall那样的操作。OpenMP采用了这种方法，OpenMP是在20世纪90年代后期由一群工业界的专家们所开发的一个已被接受的标准。OpenMP由一个小型的编译器命令集组成的，一个扩展的小型库函数和使用Fortran和C/C++基本语言的环境变量所组成。有好几种可用的OpenMP编译器，其中有些对学术界是

可以免费使用的。

对于C/C++，OpenMP命令被包含在#pragma语句中。OpenMP的#pragma语句的格式为：

```
#pragma omp directive_name ...
```

其中，omp是OpenMP的一个关键字，在命令名后可以有附加的参数（子句），以供不同的选用。某些命令需要在紧随命令后的结构块（一条语句或若干条语句）中用代码进行说明，此时的命令与结构块一起组成一个“构造”。正规的C/C++编译器将不理会#pragma语句。如果谨慎地使用#pragma语句来编写并行程序，则正规的C/C++编译器将生成一个可执行的顺序程序，而一个OpenMP的编译器将生成同一程序的并行版本。编译器命令方法的另一个优点是OpenMP编译器可进行相关性分析和语句次序的重新安排，这已在8.4.3节中提及（在那一节还叙述了更多更先进的分析和语句重排序方法）。程序员可获得这些分析的结果，并帮助编译器对语句次序重新安排。由于有时单独使用命令来说明并行性比较麻烦，所以OpenMP也有少量的库例程，主要是用来创建锁和设置并发程度。

在本小节，我们将简短地描述在OpenMP中所提供的主要特征。在附录C中汇总了OpenMP中的所有特征，而更多的细节可在[OpenMP Architecture Review Board，2002]和[Chandra et at.，2001]中找到。

OpenMP使用在8.2.1节中所叙述的分叉-汇合（fork-join）模型，但基于线程。开始时，由主线程执行一个单线程。并行区域是可由多个线程（线程组）执行的代码段。parallel
253 命令（见下面）用来创建一组线程，并指明一个可由多个线程并行执行的代码块。线程组中的确切线程数可从多种方法中选用一种方法加以确定。parallel构造中的其他命令用来指明循环的并行以及线程的不同代码块。可用命令中的private()子句、thread_private命令或其他方法将数据声明为私有变量。当用thread_private命令创建时，该私有变量将从一个并行区域持续到下一个，即这些变量值将保持不变。其他数据则为共享。

1. Parallel 命令

OpenMP中的基本命令是parallel命令

```
#pragma omp parallel
  structured_block
```

它将创建多个线程，每个线程将执行所指定的structured_block。structured_block可以是一条语句或是用{…}创建的一个复合语句，但必须只有一个入口和一个出口。在该构造的结束处有一个隐含的障栅。如果已定义OMP_NUM_THREADS且本地变量i没有越界，则该命令就相应于前面的forall构造

```
forall(i = 0; i < OMP_NUM_THREADS; i++)
  structured_block
```

例

```
#pragma omp parallel private(x, num_threads)
{
  x = omp_get_thread_num();
  num_threads = omp_get_num_threads();
  a[x] = 10*num_threads;
}
```

其中使用了两个库例程，omp_get_num_threads()将返回在并行命令中并发使用的线程数，而omp_get_thread_num()将返回线程号（从0到omp_get_num_threads()-1的

一个整数值，其中线程0为主线程）。数组a[]为全局数组，而x和num_threads被说明为是线程的私有变量。

线程组中的线程数可用下列任一方法按给定的顺序设定：

1）在parallel命令后用num_threads子句。

2）在库例程被调用之前用omp_set_num_threads()。

3）用环境变量OMP_NUM_THREADS加以定义。

如果不使用上述任一方法，则线程数将取决于系统。可用的线程数也可通过“动态调整”机制自动地加以改变以达到最佳地使用系统资源。当线程数大于可用处理器数时，将得不到最佳性能，因为那时多个线程将不得不分时共享处理器。如果代码中有足够的并行性，则通常 254
在线程数与可用处理器数相等时，就可最佳地使用资源。（注意，因为在系统上可能运行有其他任务，故可用处理器可能少于系统中的处理器总数。）只要不是按默认方式进行，动态调整就能在程序执行前进行，这只要设置环境变量OMP_DYNAMIC就可实现。当程序执行时，可在并行区外用库函数omp_set_num_dynamic(int num_threads)对动态调整功能进行激活和抑制。如果是激活，则每个并行区将使用能最佳地利用系统资源的线程数。使用这一选项意味着程序必须编写成在并行区中能以不同的线程数工作。

2. 工作共享

在这一分类中有三个构造：sections、for和single。在所有情况下，于构造的末尾有一个隐含的障栅，除非该构造中含有子句nowait。应注意的是，这些构造并不启动一个新线程组。新线程组的启动在引入parallel构造时就已完成。

（1）Sections（段）构造

```
#pragma omp sections
{
  #pragma omp section
    structured_block
  #pragma omp section
    structured_block
        ⋮
}
```

将使结构块在组内的线程间共享。注意，这里使用了#pragma omp sections和#pragma omp section两个命令。#pragma omp sections出现在结构块集前，而#pragma omp section则需出现在每个结构块前。第一个段命令是可选的。

该构造对应于前面叙述的par构造：

```
par {
  structured_block
  structured_block
      ⋮
  structured_block
}
```

它指明所有结构块可以而且应该并发执行。但是否能做到这一点将依赖于可用的处理器数。

（2）For Loop（For 循环）命令

```
#pragma omp for
  for_loop
```

255 将使for循环划分成几个部分，而这些部分将由组内的线程共享。for循环必须是规范的形式，即具有一个简单的初始化表达式，一个简单的布尔条件，和一个简单的增量表达式（在OpenMP的规范文档中对此作了全面的描述）。分割for循环的方法可由一个附加的“调度”子句加以说明。例如，子句schedule(static,chunk_size)将使for循环按chunk_size所指明的大小进行分割，且以轮转方式分配给线程。

（3）Single（单一）命令

```
#pragma omp single
  structured block
```

将使结构块仅由一个线程执行。

3. 并行工作共享的组合构造

如果在一个parallel命令后跟随单个for命令，则可将它们组合成

```
#pragma omp parallel for
  for_loop
```

它将有类似效果，即每个进程将执行相同的for循环。

如果一个parallel命令后跟随单个sections命令，则可将它们组合成

```
#pragma omp parallel sections {
  #pragma omp section
    structured_block
  #pragma omp section
    structured_block
        ⋮
}
```

它将有类似效果。（在上述两种情况中，都不允许使用nowait子句。）

4. Master命令（主命令）

master命令

```
#pragma omp master
  structured_block
```

将使主线程执行结构块。该命令与工作共享组中的不同，它在构造的末尾处没有隐含的障栅（或是开始处）。遇到该命令的其他线程将不会理会它和它的相应的结构块，而是继续向前执行。

5. 同步构造

256 在这一分类中有五个构造：critical、barrier、atomic、flush和ordered。

（1）Critical（临界） critical命令将只允许一个线程执行相应的结构块。当一个或多个进程到达critical命令时

```
#pragma omp critical name
  structured_block
```

它们将等待直至无其他进程在执行相同的临界区（即具有相同名字），然后一个线程将前行去执行该结构块。name是可选的。无名的所有临界区将被映射到一个未定义的名字。

（2）Barrier（障栅） 当一个线程到达障栅时

```
#pragma omp barrier
```

将等待直至所有其他线程都到达障栅，然后所有线程一起向前。在程序中布置障栅命令有一

些限制。特别是所有线程必须能到达障栅。

（3）Atomic（原子） 原子命令

```
#pragma omp atomic
  expression_statement
```

可用来高效地实现一个临界区，如果临界区只是简单地更新一个变量（加1、减1或完成某些其他由expression_statement所定义的简单算术操作）。在处理器中常有高效的原子指令来完成这些操作。当然，一个好的编译器，不论如何应能在临界区中发现原子命令。算术操作在expression_statement中给定，它必须是由OpenMP定义的简单形式（见附录C）。

（4）Flush（刷新） 共享对象最初存储在共享存储器中，当它们被处理器访问时就会被引入局部存储器（处理器寄存器或高速缓存）。刷新命令是一个同步点，该点将使存储器中的某些或所有变量具有“一致性”观察，并允许所有当前对变量的读和写操作完成且将值写回主存储器，但代码中任何在刷新命令后的存储器操作均不能开始，从而就构成了一个“存储器篱笆（memory fence）”。刷新命令的格式为

```
#pragma omp flush (variable_list)
```

刷新命令将自动地出现在parallel和critical命令（以及组合的parallel for和parallel sections的命令中）的入口处和出口处，且出现在for、sections及single（如果没有nowait子句）的出口处。注意，它不会出现在master中，或是for、sections及single的入口处。此外，它只适用于执行刷新命令的线程，而不是组中的所有线程，它们不会自动地获得一致性的存储器观察。要做到这一点，则每个线程都必须执行一条刷新命令。

（5）Ordered（按序） ordered命令与for及parallel for命令联合在一起使用，它将使迭代按顺序循环方式编写所呈现的次序加以执行。更多的详细内容请参见附录C。 257

8.6 性能问题

8.6.1 共享数据的访问

即使处理器有能力访问共享存储器中的任何单元，由于所有的现代计算机系统中都有高速缓存，它们是一些高速的存储器，非常接近地连接到每个处理器上，因此，为获取最好的性能，尝试组织数据仍是非常重要的。使用高速缓存的原因是因为处理器访问存储单元的速度远远超过主存储器响应的速度。一个速度更高、容量较小的高速缓存能更好地与处理器的速度相匹配。现在的系统甚至拥有多于一级的高速缓存：一个小容量的与处理器处于同一芯片内的一级高速缓存（L1），以及在一级缓存之后的一个更大些的片外二级高速缓存（L2）。此外甚至可以有一个共享的三级高速缓存（L3），特别在对称共享存储器多处理机（SMP）中。下面我们将考虑每个处理器只有一级高速缓存的情况。

程序由可执行的指令（代码）和相应的数据所组成。现实情况中，可执行指令在程序执行过程中是不会改变的。相反，数据可以改变，从而导致系统设计的显著复杂，并将对总体性能产生很大影响。当某个处理器首次访问一个主存储器单元时，该内容的一个拷贝将被传送到处理器的高速缓存中。假设被放入高速缓存的信息是数据，则当这个处理器此后再访问该数据时，它可以直接访问高速缓存。如果另一个处理器此后也访问该相同的主存储器单元时，则同样数据的另一拷贝也将传送到与该处理器相连的高速缓存中，因此，同一数据就有了多份拷贝。这并不成问题，但当某个处理器改变了它的高速缓存中的拷贝时，也就是它要写入新的数据值时，问题就出现了。此时就需要使用高速缓存一致协议（cache coherence

protocol）来确保后继处理器引用这个数据时，得到的是最新变动过的数据。

高速缓存一致性协议可以使用更新策略，或更通常的是使用使无效策略。在更新策略中，当数据在高速缓存中的某个拷贝被更新时，这个数据在所有其他高速缓存中的所有其他拷贝都将做同样的变动。在使无效策略中，当数据的某个拷贝变动时，通过重置高速缓存中的有效位，使所有其他高速缓存中的相同数据拷贝都变为无效。当一个处理器试图访问一个无效拷贝时，该无效拷贝就将被更新。现在有各种不同版本的高速缓存一致性协议（更多细节请参见[Tomasevic and Milutinovic, 1993]）。程序员可以假设系统中已经存在一个有效的高速缓存一致协议，而这将对系统的性能有所影响。

高速缓存的存在意味着为了获取更高的性能我们应该对所使用的并行算法加以一定的改变。关键的特征在于高速缓存是以连续单元的块（也称为行）的形式组成的。当处理器首次引用块中的一个或几个字节时，整个块就会从主存储器传送到高速缓存中。因此，如果要访问该数据块的另一部分时，它已在高速缓存中而不必从主存储器将它传送到高速缓存。如果高速缓存所使用的块的大小和编译器存储数据的方法已知的话，并行算法就可以利用高速缓存的这一特征。当然，这种方法将使性能高度依赖于执行这个程序所使用的实际计算机系统的情况。

258 在高速缓存中使用块的原因是顺序程序的一个基本特征是存储器的访问趋向于在以前存储器访问的附近，即所谓*时间局部性*（temporal locality）。然而，不同处理器所需的访问可能是一个高速缓存块的不同部分，而不是块中的那些相同字节。尽管实际数据不共享，但如果一个处理器对该块的一部分进行写，则这个块在其他高速缓存中的拷贝就必须全部进行更新或者使无效。这就是所谓的*假共享*（false sharing），它对系统的性能有负面的影响。图8-10中对假共享作了说明。在图8-10中，一个高速缓存块由8个字组成，从0到7。两个处理器访问同一块中的不同字节（处理器1访问字3，而处理器2访问字5）。假定处理器1改写了字3，则高速缓存一致协议将使处理器2中的该高速缓存块更新或使其无效，即便处理器2永远也不访问字3。假定现在处理器2又改写了字5，则现在高速缓存一致协议反过来又要将使处理器1中的该高速缓存块进行更新或使其无效，即便处理器1可能也许从不访问字5。这样下去，就会不幸地导致造成高速缓存块的乒乓效应（ping-ponging）。

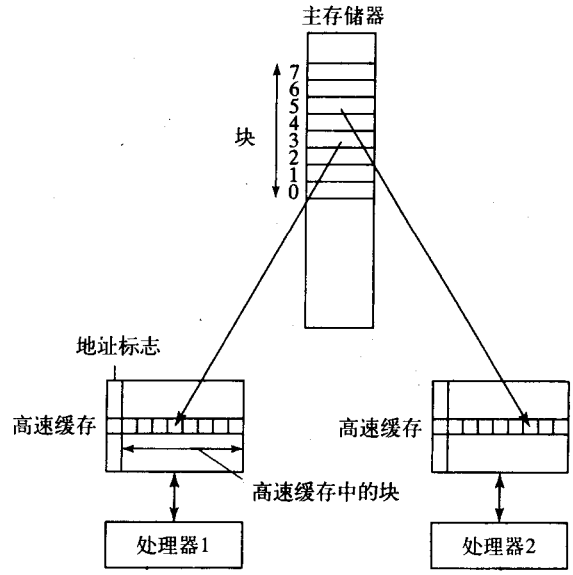


图8-10 高速缓存中的假共享

解决这一问题的一个方法是让编译器改变共享数据在主存中存放的布局，将所有仅由一个处理器所写的数据分布到不同的块中。当然，这一点可能很难在所有情况中满足。例如，

259 代码

```
forall (i = 0; i < 5; i++)
  a[i] = 0;
```

很可能产生假共享，因为a的元素a[0]、a[1]、a[2]、a[3]和a[4]通常在主存中会被存

放在连续的单元中。避免假共享的唯一方法就是将每个元素放在不同的块中，对于一个大型数组来讲这将导致很大的存储空间的浪费。可是，即使是如下的代码

```
par {
   x = 0;
   y = 0;
}
```

其中，x和y是共享变量，仍可能产生假共享，因为变量x和y很可能被存放在同一个数据段中。

一般而言，程序员需要将它们的并行算法作如下的处理，以充分利用高速缓存的特征并尽量减少假共享（即不共享的数据项不放在同一块中）。

8.6.2 共享存储器的同步

同步原语的使用是导致共享存储器程序性能降低的主要原因。在共享存储器程序中同步使用的主要目的为下面三个之一：

- 互斥同步
- 进程/线程同步
- 事件同步

互斥同步用来控制对临界区的访问，且可用加锁/开锁（lock/unlock）例程来实现。高性能的程序应尽少使用临界区，因为它们的使用将使代码顺序化，如图8-11所说明的那样。假设如通常那样，每个处理器正执行一个进程，且所有处理器正好一起到达它们的临界区。这些临界区将依次执行。在这种情况下，执行时间会变成如同一个单处理器的执行时间。正如[Pfister，1998]所指出的那样，此时增加处理器数目会适得其反。例如，假定有p个处理器，每个有一个临界区，需要t_{crit}时间单位，而在临界区外的计算需要t_{comp}时间单位，且这两部分重复。当$t_{comp} < pt_{crit}$时，处于活动的处理器数有时将小于p（见图8-11）。

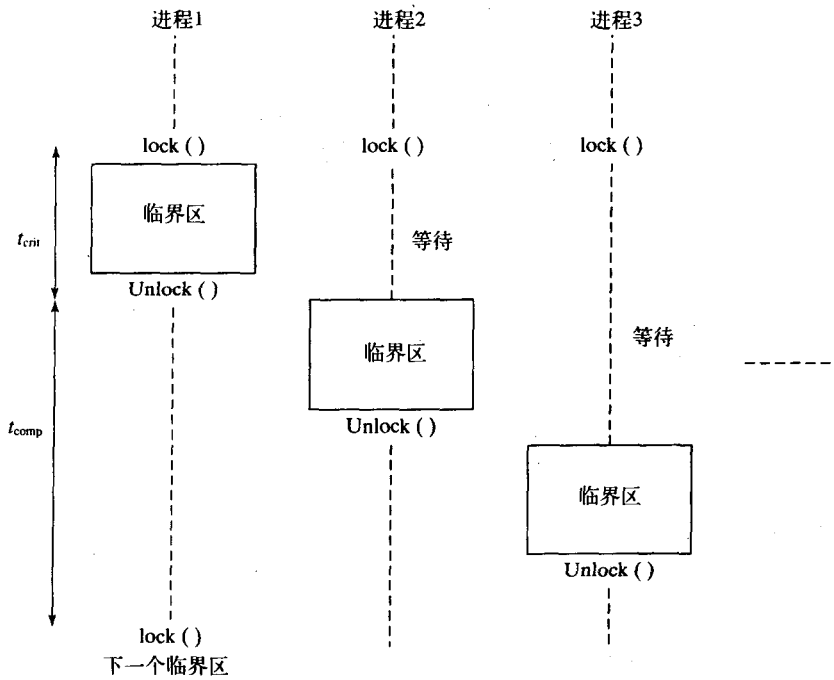


图8-11 临界区序列化代码

如我们所知道的那样，在消息传递程序和共享存储器程序中都使用障栅，但障栅有时会导致处理器不必要的等待。例如，可将某些同步算法修改成异步的或是部分同步的（关于部分同步在第6章的末尾已做了描述），此时只需使用较少的障栅，这将显著减小执行时间。

事件同步是指当在一个进程/线程中某一条件或某一值出现时，用信号去通知另一个进程/线程。事件同步可用如下代码加以实现

```
Process 1                          Process 2
    ⋮
data = new;
flag = TRUE;                           ⋮
    ⋮                              while (flag != TRUE) { };
                                   data_copy = data;
                                       ⋮
```

260

其中，进程2被告知数据已被进程1更新。而flag是一个共享变量，但不一定要将它限制在临界区中加以访问。如[Culler and Singh，1999]所提及的，与其使用一个独立的flag变量，不如使用数据变量本身：

```
Process 1                          Process 2
    ⋮
data = new;
    ⋮                                  ⋮
                                   while (data != new) { };
                                   data_copy = data;
                                       ⋮
```

尽管该代码的可读性变差，而且还不得不特别小心地研究可能性和隐含性。这里的一种假设是，data在被进程1设置成new之前，不可能等于new，即使发生了也没有关系。另一种假设是，每个进程中的那些语句是按程序中给定的次序执行的。最后，高速缓存的存在是一个很关键的因素。保留在高速缓存的那些值可能未被更新，除非高速缓存将其显式地加以更新。

261

在下一小节，我们将进一步探讨程序顺序和存储器操作。

8.6.3 顺序一致性

在任何的（MIMD）多处理机系统中，每个处理器将执行保存在本地存储器中自己的程序，因此在同时将有多于一个的程序执行。术语*顺序一致性*（sequential consistency）指的是这些程序执行的最终结果将是相同的，而与各个程序的时间关系无关。即在任何时间点，每个程序的各条指令执行的实际进展可以不同，且可以任何次序相互交叉。唯一的约束是每个程序中的指令需按程序次序执行。由[Lamport，1979]形式化定义的顺序一致性为：

> *一台多处理机是顺序一致的，如果任何执行的结果都尤如所有处理器的操作按某种程序次序执行的一样，且每个处理器的操作顺序是按它的程序所说明的次序进行的。*

也就是说，不管指令的执行在时间上以何种方式交叉，一个并行程序的整体执行效果并没有发生变化。

保证程序最终结果正确的关键所在是处理器对存储器单元访问请求的次序。对于一个具有顺序一致性的系统而言，它必须仍产生程序设计时所希望得到的结果，即使来自不同处理器

262

的各个访问请求可以任何次序交叉。这一情景可如图8-12所示来加以描述。显然，在图8-12中如果一个处理器去读一个共享变量x，它通常需要的是该变量的最新值。顺序一致性将在定时

变化范围内提供此值，而这种定时变化在程序执行时就可能出现。

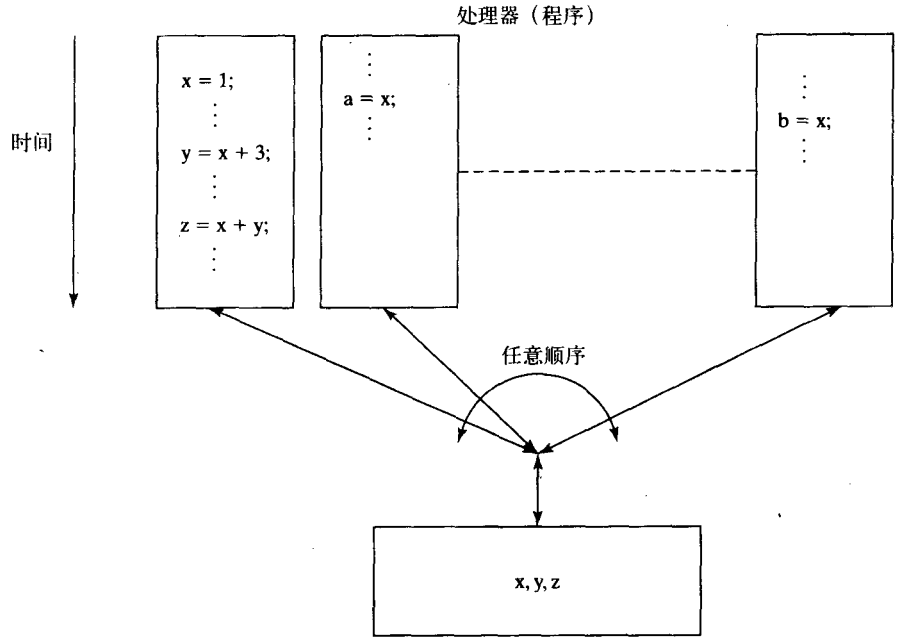


图8-12 顺序一致系统

当为一个已知是顺序一致系统编写一个并行程序时，我们就能推断程序的执行结果。在8.6.2节中所给定的代码序列中

```
Process 1                    Process 2

    ⋮
data = new;
flag = TRUE;                     ⋮
    ⋮                        while (flag != TRUE) { };
                             data_copy = data;
                                 ⋮
```

我们将期待data_copy会被设置为new，因为我们期待语句data = new会在语句flag = TRUE之前执行，而语句while(flag! = TRUE){}会在data_copy = data之前执行[Hill, 1998]。在顺序一致系统中使用该代码序列可保证一个进程（例中为进程2）从另一个进程（例中为进程1）读得的值确是新值。进程2只需简单地等待新值的产生。

1. 程序顺序

Lamport关于顺序一致性的定义是“每个处理器的操作顺序是按它的程序所指定的次序”或*程序次序*。在图8-12中，该次序即是要被执行的、已存储的机器指令的次序。然而，如果允许编译器为改善性能对语句次序重新排序，则已存机器指令的次序就可能与相应的源程序中高级语言语句的次序不一样。在这种情况下，前面的例子可能失败。

即使不允许编译器对代码次序进行重排，在现代的高性能处理器中机器指令的执行次序也不是必须与存储器中机器指令的次序相同，因为现代的处理器为了增强性能在执行时通常会在内部重排机器指令的次序。然而，一个处理器可能不按程序次序执行机器指令的这一事实，并不会改变多处理机的顺序一致性，如果处理器所产生的最后结果与按程序次序执行的

结果一样的话；也就是说按程序次序将值收回到寄存器和存储器单元，通常处理器用这种方法来保持顺序一致性。

作为处理器重新排序的一个例子，假定前面例子中的新数据的计算按下式进行：

```
Process 1                          Process 2

    ⋮
new = a * b;
data = new;
flag = TRUE;                           ⋮
    ⋮                              while (flag != TRUE) { };
                                   data_copy = data;
263                                    ⋮
```

在new = a * b中的乘法操作相当于可执行程序中的乘法机器指令。而下一条与其相关的指令data = new必须在乘法指令完成后并生成其结果才可发动再下一条指令flag = TRUE则完全是独立的，一台聪明的处理器不会追随顺序一致性，它会在乘法结束之前就启动这一操作（事实上确是这样），这将导致如下序列：

```
Process 1                          Process 2

    ⋮
new = a * b;
flag = TRUE;
data = new;
    ⋮                                  ⋮
                                   while (flag != TRUE) { };
                                   data_copy = data;
                                       ⋮
```

现在，while语句若在new赋值给data之前出现，将导致出错。

所有多处理机具有在顺序一致性模型下运行的选项，即强迫指令按程序次序存放它们的结果。可以使用一种显式的存储器篱笆，例如，在OpenMP中使用flush命令。但是，这将显著限制编译器的优化和处理器的性能。（这一结论已被[Hill，1998]质疑。）

2. 松弛的读/写次序

可为处理器提供相应的设施，以允许它们松弛相对于另一处理器的读、写次序的一致性。例如，术语*处理器一致性*（processor consistency）描述了这样一种情景，即各个处理器的写按程序次序，但不同处理器的交叉写可以按不同的次序。这种松弛将为改善性能提供一些机会（通过缓存和重排指令加以实现）。

为支持通用的松弛读和写的次序，已提供了各种称为*存储器篱笆*（memory fence）或*存储器障栅*（memory barrier）的专用机器指令，当程序中需要时就用来同步存储器操作。例如，Alpha处理器有一条存储器障栅指令，它会在发动任何新的存储器操作前，等待所有以前发动的存储器访问指令的执行结束。它还有一条写存储器障栅指令，其功能类似于存储器障栅指令，但只考虑存储器的写操作。SUN Sparc V9处理器有一条存储器障栅指令，并设有4位供变化之用。在该存储器障栅指令之前的所有写操作未完成之前，write-to-read位将防止在其后的任何读操作的发动。其他位为write-to-write、read-to-read以及read-to-write。IBM PowerPC 处
264 理器有一条sync指令，它类似于Alpha的存储器障栅指令。

8.7 程序举例

在本节中，我们将通过编写使用多个进程/线程对数组a[1000]的元素求和的简单程序来演示UNIX系统调用、Pthread和Java的用法:

```
int sum, a[1000];
   sum = 0;
   for (i = 0; i < 1000; i++)
      sum = sum + a[i];
```

当然，我们通常不会使用UNIX重量级进程来解决这一问题，但这样做可以展示一下使用临界区的技术。对于使用UNIX进程的例子来说，将静态分配任务，而对于使用Pthread和Java的例子来说，我们将使用动态负载平衡方法来负责任务的分配。对于UNIX进程和Pthread线程，必须使用临界区来保护对共享变量的访问。Java则使用监控程序方法。

8.7.1 使用UNIX进程的举例

这个例子中，所有的计算将被分割成两部分，奇数i部分和偶数i部分，即

进程1

```
sum1 = 0;
for (i = 0; i < 1000; i = i + 2)
   sum1 = sum1 + a[i];
```

进程2

```
sum2 = 0;
for (i = 1; i < 1000; i = i + 2)
   sum2 = sum2 + a[i];
```

每个进程都将把自己的结果（sum1或者sum2）加到累加结果sum中（在sum被初始化后），得到最后结果:

```
sum = sum + sum1;          sum = sum + sum2;
```

存放最后结果sum的存储器单元应该是由两个进程共享的，并且由锁机制来保护访问。在这个程序中，我们创建了一个共享数据结构，如图8-13所示。仅有一个进程访问数组a[]的特定元素，并且在任何情况下访问仅仅是读访问，所以对数组a[]的访问就无需保护。但是每个进程都可改变sum，因此对sum的操作必须在一个临界区内进行。我们使用了一个二元信号量来实现这个临界区。

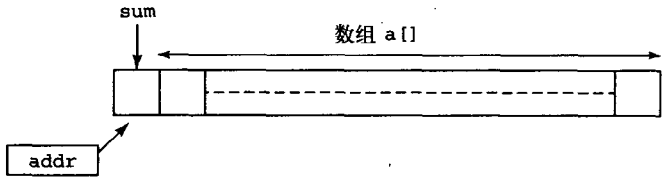


图8-13 8.7.1节程序举例中使用的共享存储器单元

265

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <errno.h>
#define array_size 1000          /* no of elements in shared memory */
extern char *shmat();
void P(int *s);
void V(int *s);
int main() {
```

```
int shmid, s, pid;                  /* shared memory, semaphore, proc id */
char *shm;                          /*shared mem. addr returned by shmat()*/
int *a, *addr, *sum;                /* shared data variables*/
int partial_sum;                    /* partial sum of each process */
int i;
                                    /* initialize semaphore set */
int init_sem_value = 1;
s = semget(IPC_PRIVATE, 1, (0600 | IPC_CREAT))
if (s == -1) {                      /* if unsuccessful*/
   perror("semget");
   exit(1);
}
if (semctl(s, 0, SETVAL, init_sem_value) < 0) {
   perror("semctl");
   exit(1);
}
                                    /* create segment*/
shmid = shmget(IPC_PRIVATE,(array_size*sizeof(int)+1),
   (IPC_CREAT|0600));
if (shmid == -1) {
   perror("shmget");
   exit(1);
}
                                    /* map segment to process data space */
shm = shmat(shmid, NULL, 0);
                                    /* returns address as a character*/
if (shm == (char*)-1) {
   perror("shmat");
   exit(1);
}

addr = (int*)shm;                   /* starting address */
sum = addr;                         /* accumulating sum */
addr++;
a = addr;                           /* array of numbers, a[] */

*sum = 0;
for (i = 0; i < array_size; i++)    /* load array with numbers */
   *(a + i) = i+1;
pid = fork();                       /* create child process */
if (pid == 0)   {                   /* child does this */
   partial_sum = 0;
   for (i = 0; i < array_size; i = i + 2)
      partial_sum += *(a + i);
else {                              /* parent does this */
   partial_sum = 0;
   for (i = 1; i < array_size; i = i + 2)
      partial_sum += *(a + i);
}
P(&s);                              /* for each process, add partial sum */
   *sum += partial_sum;
V(&s);

printf("\nprocess pid = %d, partial sum = %d\n", pid, partial_sum);
if (pid == 0) exit(0); else wait(0);          /* terminate child proc */
```

```
printf("\nThe sum of 1 to %i is %d\n", array_size, *sum);

                                                    /* remove semaphore */
if (semctl(s, 0, IPC_RMID, 1) == -1) {
   perror("semctl");
   exit(1);
}
                                                    /* remove shared memory */
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
   perror("shmctl");
   exit(1);
}

exit(0);
}                                                   /* end of main */

void P(int *s) {                                    /* P(s) routine*/
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num = 0;
   sops->sem_op = -1;
   sops->sem_flg = 0;
   if (semop(*s, sops, 1) < 0) {
      perror("semop");
      exit(1);
   }
   return;
}

void V(int *s) {                                    /* V(s) routine */
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num = 0;
   sops->sem_op = 1;
        sops->sem_flg = 0;
        if (semop(*s, sops, 1) <0) {
           perror("semop");
           exit(1);
        }
        return;
     }

SAMPLE OUTPUT

process pid = 0, partial sum = 250000
process pid = 26127, partial sum = 250500
The sum of 1 to 1000 is 500500
```

8.7.2 使用Pthread的举例

本例中，我们将创建num_thread个线程，每个线程都将从列表中获取数并加到它们的和中。当所有的数都被取走后，这些线程就将它的部分和加到共享单元sum中。在本程序中，可按照图8-14所示的结构创建共享的数据结构。各个线程通过共享单元global_index获取

a[]数组的下一个未加元素。在这个index所指的数组元素被读后，index就会加1，从而指向下一个未加元素，为下一次元素的获取做好准备。如上例一样，结果所在的主存单元将是sum，并将同样被共享且使用锁机制来保护对其的访问。

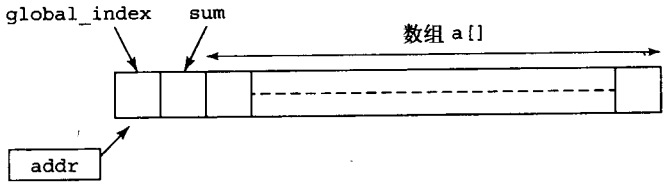


图8-14　8.7.2节程序举例中使用的共享存储器单元

有一点非常重要，那就是不能在临界区外访问全局索引下标global_index。这包括我们察看索引下标是否达到最大值的操作。如下语句:

```
while (global_index < array_size) ...
```

需要访问global_index，在while的语句体被执行之前，这个索引下标可能被其他的线程改变。在本代码中，使用一个局部变量，local_index来存放当前读到的global_index的值，以便更新部分和以及检查下标是否达到了最大值。

266
~
268

在本代码中，我们使用了互斥锁机制，而没有使用条件变量。使用该方法的程序如下:

```
#include <stdio.h>
#include <pthread.h>
#define array_size 1000
#define num_threads 10
                                    /* shared data */
int a[array_size];                  /* array of numbers to sum */
int global_index = 0;               /* global index */
int sum = 0;                        /* final result, also used by slaves */
pthread_mutex_t mutex1;             /* mutually exclusive lock variable */
void *slave(void *ignored)  {       /* Slave threads */
int local_index, partial_sum = 0;
do {
   pthread_mutex_lock(&mutex1);  /* get next index into the array */
      local_index = global_index;/* read current index & save locally*/
      global_index++;           /* increment global index */
   pthread_mutex_unlock(&mutex1);

   if (local_index < array_size)
        partial_sum += *(a + local_index);

} while (local_index < array_size);

pthread_mutex_lock(&mutex1);     /* add partial sum to global sum */
   sum += partial_sum;
pthread_mutex_unlock(&mutex1);

return ();                       /* Thread exits */
}

main () {
int i;
pthread_t thread[num_threads];          /* threads */
```

```
    pthread_mutex_init(&mutex1,NULL);          /* initialize mutex */

    for (i = 0; i < array_size; i++)           /* initialize a[] */
      a[i] = i+1;

    for (i = 0; i < num_threads; i++)          /* create threads */
      if (pthread_create(&thread[i], NULL, slave, NULL) != 0)
        perror("Pthread_create fails");

    for (i = 0; i < num_threads; i++)          /* join threads */
      if (pthread_join(thread[i], NULL) != 0)
        perror("Pthread_join fails");

    printf("The sum of 1 to %i is %d\n", array_size, sum);
    }                                          /* end of main */
```

```
SAMPLE OUTPUT
The sum of 1 to 1000 is 500500
```

269

习题8-14中探讨了一种从线程一次取得多至10个的连续放置的数以组为单位进行求和的方法，由于减少了读取索引下标的次数，因此这是一个更高效的方法。因为这些线程都无返回值，所以可使它们成为分离线程。

8.7.3 使用Java的举例

下面的代码就是这个求和问题的一个简单的Java实现。这个程序是由北卡罗来纳大学夏洛特分校（UNCC）的学生P.Shah所编写的，用于演示Java的监控程序方法。（我们应该指出，依赖于Java的实现，一个线程可以承担所有的工作。）

```
public class Adder {
  public int[] array;
  private int sum = 0;
  private int index = 0;
  private int number_of_threads = 10;
  private int threads_quit;

  public Adder() {
    threads_quit = 0;
    array = new int[1000];
    initializeArray();
    startThreads();
  }

  public synchronized int getNextIndex() {
    if(index < 1000) return(index++); else return(-1);
  }

  public synchronized void addPartialSum(int partial_sum) {
    sum = sum + partial_sum;
    if(++threads_quit == number_of_threads)
      System.out.println("The sum of the numbers is " + sum);
  }

  private void initializeArray() {
```

```
        int i;
        for(i = 0;i < 1000;i++) array[i] = i;
    }

    public void startThreads() {
        int i = 0;
        for(i = 0;i < 10;i++) {
            AdderThread at = new AdderThread(this,i);
            at.start();
        }
    }
    public static void main(String args[]) {
        Adder a = new Adder();
    }

}

class AdderThread extends Thread {
    int partial_sum = 0;
    Adder parent;
    int number;
    public AdderThread(Adder parent, int number) {
        this.parent = parent;
        this.number = number;
    }

    public void run() {
        int index = 0;
        while(index != -1) {
            partial_sum = partial_sum + parent.array[index];
            index = parent.getNextIndex();
        }
        System.out.println("Partial sum from thread " + number + " is "
            + partial_sum);
        parent.addPartialSum(partial_sum);
    }
}
```

8.8 小结

本章介绍了以下内容:

- 进程的创建
- 线程的概念和线程的创建
- Pthread规范的例程
- 数据如何创建为共享数据
- 共享数据访问的控制方法
- 条件变量的概念
- 并行程序设计语言结构

- OpenMP
- 使用Bernstein条件来进行相关性分析
- 影响系统性能的因素（同步，高速缓存）
- 代码举例

270
~
271

推荐读物

现在有许多关于并行程序设计语言的文献，著名的参考文献包括基于FORTRAN的一些早期语言的 [Karp and Babb，1988]和基于C++语言的[Wilson and Lu，1996]。[Skillicorn and Tabia，1995]提供了那些重要文献的再印版。[Brawer，1989]的教科书中阐述了基于UNIX系统调用的并行程序设计方法。不过，我们需要指出的是，在很多实际的并行程序设计环境中没有使用这种UNIX进程的方法是因为创建UNIX进程的代价太大了。Pthread规范和多线程程序设计则在许多书中有所描述，著名的有[Kleiman，Shah and Smaalders，1996]、[Butenhof，1997]、[Nichols，Buttlar and Farrell，1996]和[Prasad，1997]。

[Chandra et al.，2001]描述了OpenMP。有关OpenMP的权威原始资料出自http://www.OpenMP.org的体系结构评审委员会（2002）。

使用Java进行共享存储器程序设计也是进一步研究的方向。对一个Java初学者来说比较不错的网站是http://java.sun.com/。大多数入门的Java书籍不涉及线程或同步，但有关信息可在[Campione，Walrath, and Huml，2001]中找到。更专业的书籍包括[Lewis and Berg，2000]。

参考文献

BAL, H. E., J. G. STEINER, AND A. S. TANENBAUM (1989), "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, Vol. 21, No. 3, pp. 261–322.

BEN-ARI, M. (1990), *Principles of Concurrent and Distributed Programming*, Prentice Hall, Englewood Cliffs, NJ.

BERNSTEIN, A. J. (1966), "Analysis of Programs for Parallel Processing," *IEEE Trans. Elec. Comput.*, Vol. E-15, pp. 746–757.

BRÄUNL, T. (1993), *Parallel Programming: An Introduction*, Prentice Hall, London.

BRAWER, S. (1989), *Introduction to Parallel Programming*, Academic Press, San Diego, CA.

BUTENHOF, D. R. (1997), *Programming with POSIX® Threads*, Addison-Wesley, Reading, MA.

CAMPIONE, M., K. WALRATH, AND A. HUML (2001), *The Java™ Tutorial 3rd edition: A Short Course on the Basics*, Addison-Wesley, Boston, MA.

CHANDRA, R, L. DAGUM, D. KOHR, D. MAYDAN, J. MCDONALD, AND R. MENON (2001), *Parallel Programming in OpenMP*, Academic Press, San Diego, CA.

CONWAY, M. E. (1963), "A Multiprocessor System Design," *Proc. AFIPS Fall Joint Computer Conf.*, Vol. 4, pp. 139–146.

CULLER, D. E., AND J. P. SINGH (WITH A. GUPTA) (1999), *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, San Francisco, CA.

DIJKSTRA, E. W. (1968), "Cooperating Sequential Processes," in *Programming Languages*, F. Genuys (ed.), Academic Press, New York, pp. 43–112.

FOSTER, I. (1995), *Designing and Building Parallel Programs*, Addison-Wesley, Reading, MA.

HILL, M. D. (1998), "Multiprocessors Should Support Simple Memory Consistency Models," *Computer*, Vol. 31, No. 8, pp. 29–34.

HOARE, C. A. R. (1974), "Monitors: An Operating System Structuring Concept," *Comm. ACM*, Vol. 17, No. 10, pp. 549–557.

KARP, A., AND R. BABB (1988), "A Comparison of Twelve Parallel Fortran Dialects," *IEEE Software*, Vol. 5, No. 5, pp. 52–67.

KLEIMAN, S., D. SHAH, AND B. SMAALDERS (1996), *Programming with Threads*, Prentice Hall, Upper Saddle River, NJ.

LAMPORT, L. (1979), "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Comp.*, Vol. C-28, No. 9, pp. 690–691.

LEWIS, B. AND D. L. BERG (2000), *Multithreaded Programming with Java Technology*, Sun Microsystems Press, Palo Alto, CA.

NICHOLS, B., D. BUTTLAR, AND J. P. FARRELL (1996), *Pthreads Programming*, O'Reilly & Associates, Sebastopol, CA.

OPENMP ARCHITECTURE REVIEW BOARD (2002), *OpenMP C and C++ Application Program Interface Version 2, March 2002*, from http://www.OpenMP.org.

PFISTER, G. F. (1998), *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing*, Prentice Hall, Upper Saddle River, NJ.

POLYCHRONOPOULOS, C. D. (1988), *Parallel Programming and Compilers*, Kluwer Academic, Norwell, MA.

PRASAD, S. (1997), *Multithreading Programming Techniques*, McGraw-Hill, New York.

SKILLICORN, D. B., AND D. TABIA (1995), *Programming Languages for Parallel Processing*, IEEE CS Press, Los Alamitos, CA.

SUNSOFT (1994), *Pthread and Solaris Threads: A Comparison of Two User Level Threads APIs*, Sun Microsystems, Mountain View, CA.

TERRANO, A. E., S. M. DUNN, AND J. E. PETERS (1989), "Using an Architectural Knowledge Base to Generate Code for Parallel Computers," *Comm. ACM*, Vol. 32, No. 9, pp. 1065–1072.

TOMASEVIC M., AND V. MILUTINOVIC (1993), *The Cache Coherence Problem in Shared Memory Multiprocessors: Hardware Solutions*, IEEE CS Press, Los Alamitos, CA.

WILKINSON, B. (1996), *Computer Architecture Design and Performance*, 2nd edition, Prentice Hall, London.

WILSON, G. V., AND P. LU, eds. (1996), *Parallel Programming Using C++*, MIT Press, Cambridge, MA.

WOLFE, M. (1996), *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, CA.

习题

许多在其他章习题部分的"科学/数值习题"和"现实生活习题"中出现的程序设计题，可作为程序设计作业用Pthread和OpenMP实现。本章的部分习题特为线程和OpenMP设计。

科学/数值习题

8-1 如果有两个进程，每个进程都有三条指令，请列出这些指令的各种可能顺序。

272
~
273

8-2 8.3.2节中给出了两个"action"例程等待"counter"例程将计数器减到0的例子，请使用Pthread规范的条件变量编写实现此例的代码。

8-3 指出下列代码要完成什么动作?

```
forall (i = 0; i < n; i++) {
   a[i] = a[i + n];
}
```

8-4 分析8.4.3节中给出的如下代码，是否体中任何实例都可以同时执行:

```
forall (i = 2; i < 6; i++) {
    x = i - 2*i + i*i;
    a[i] = a[x];
}
```

8-5 能否将下列代码:

```
for (i = 0; i < 4; i++) {
    a[i] = a[i + 2];
}
```

改写成:

```
forall (i = 0; i < 4; i++) {
    a[i] = a[i + 2];
}
```

而仍能获得正确的结果? 请解释原因。

8-6 试解释下列的每个程序段为何不能工作，并重写每个实例的代码使在给定n = 100 和11个处理器时能够工作。

(a) 对数组a中的每个元素加常数5:

```
for (i = 1;i <= n; i++)
    FORK a[i] = a[i] + 5;
```

(b) 计算数组a中所有元素的和:

```
forall (i = 1;i <= n; i++)
    sum = sum + a[i];
```

8-7 列出下列代码执行时的所有可能输出:

```
j = 0;
k = 0;
forall (i = 1; i <= 2; i++) {
    j = j + 10;
    k = k + 100;
}
printf("i=%i,j=%i,k=%i\n",i,j,k);
```

假定每个赋值语句是原子性的。(提示: 将赋值语句编号，然后找出每一种可能序列。) 274

8-8 假设下面的类C并行代码用于将一个矩阵转置:

```
forall (i = 0; i < n; i++)
    forall (j = 0; j < n; j++)
      a[i][j] = a[j][i];
```

请指出为什么这些代码并不能实现上述功能，请重写代码使其能够达到设计目的。

8-9 如在8.3.2节末尾所提及的，基本的Pthread（POSIX.1标准）不具有内在的障栅。书写一个障栅例程并加以测试，其中应包括创建和初始化任何必需的数据结构（共享变量、互斥锁、条件变量）例程以及撤销数据结构的例程。

8-10 基本的Pthread（POSIX.1标准）不具有内部的读/写锁。读/写锁是区分读访问和写访问的一种锁的形式，它允许多于一个的线程读数据，但只允许一个进程改变它。当一个线程设置一个读/写锁时要指明该锁是（共享）读访问锁还是（互斥）写访问锁。如果另一个线程进行写访问，则就不允许该线程继续进行，反之就允许进行。当多个线程正等待一个读/写锁时，就需要建立一个先后次序: 不是读访问优先于写访问就是写访问优先于读访问。如果是前者，则能尽快进行多个同时读访问; 如果是后者则能尽快

进行更新数据。用Pthread实现这样一个读/写锁。

8-11　假设下面的类C并行例程用于将前n个数求和:

```
int summation(int n);
{
int sum = 0;
forall (i = 1; i <= n; i++)
   sum = sum + i;
return(sum);
}
```

为什么上述例程其实不能完成这个功能？并请重写代码使其能够在51个处理器上完成$n = 200$的计算。

8-12　请确认并解释下面的障栅代码是如何工作的（基于6.1.3节给出的两阶段障栅):

```
void barrier()
{
   lock(arrival);
   count++;
   if (count < n) unlock(arrival)
   else unlock(departure);
   lock(departure);
   count--;
   if (count > 0) unlock(departure)
   else unlock(arrival);
   return;
}
```

为什么必须使用两个锁变量，arrival和departure?

8-13　编写一个使用Pthread或OpenMP的程序以实现4.2.2节中所述的数值积分，并使用不同的分解方法（矩形方法和梯形方法）进行比较。

275

8-14　重写8.7.2节中的Pthread例子，使得各个从进程可以一次最多提取10个连续的数，以组为单位进行求和，从而减少对索引下标的访问。

8-15　条件变量可以用来检查分布式终止。请用第7章中描述的有分布式终止的负载平衡的程序中使用条件变量。

8-16　编写一个拥有两个线程的多线程程序，其中一个文件被一个线程读入缓冲区，而被另一个线程写到另一个文件中。

8-17　编写一个Pthread或OpenMP程序来求解二次方程$ax^2 + bx + c = 0$的根。使用下面的公式:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

其中中间值用不同的线程计算，并使用条件变量来识别何时所有的线程都完成了它的计算。

8-18　如果三个进程同时到达它们的临界区且每个临界区需要t_c秒，则进程等待进入临界区的所花费的总时间为多少?

8-19　用OpenMP重写习题8-3中的代码。

8-20　选择在其他章中的一个科学/数值习题，并使用Pthread和MPI进行比较研究。测量每一种情况下的执行时间。再用顺序方法实现，并测量此时所需的执行时间。

8-21　重做习题8-20但比较OpenMP和MPI的实现。

8-22　重做习题8-20但比较Pthread和OpenMP的实现。

现实生活习题

8-23 编写一个多线程程序来模拟两个人使用两个自动取款机来访问同一个共享账号的情况，并扩展程序以允许自动借方出现。

8-24 编写一个多线程程序来实现航空公司的机票预订系统，使其能够应付多个旅行社访问同一个空余机票源（存放在共享存储器中）的情况。

8-25 编写一个多线程程序来实现医疗信息系统，使其能够被多个试图检索和更新（有时是增加）共享存储器中存储的病人病历的医生访问。

8-26 编写一个多线程的程序来实现一个售票系统，来售卖摇滚乐队“紫色母亲”的下一场将在北卡罗来纳州夏洛特城的Ericsson体育场举行的音乐会。

8-27 编写一个多线程程序来模拟一个计算机网络，在这个网络中各个工作站和一个主服务器通过单个以太网连接，并以随机的间隔互相传递消息。使用一个线程来模拟每个随机请求向其他工作站传递消息的工作站，并在实现时考虑消息的大小和消息冲突的情况。

8-28 请扩展习题8-27的程序，通过提供多个以太网线路的方法（如1.4节所描述的）。

8-29 编写一个多线程程序来模拟一个超立方体网络和一个网格（mesh）网络，其中两者都在结点之间具有多条并行的通信链路。观察一下当结点之间的并行链路数增加的时候，性能是如何变化的；并使用模拟的结果对超立方体和网格这两种网络的性能进行比较研究。其中，性能的衡量标准包括每个时间段内接受的请求数。请参见[Wilkinson, 1996]来了解这个模拟练习的更多细节和采样结果。 276

8-30 设计并实现一个使用锁来保护临界区和条件变量并且只需要少于3页的代码就可以实现的问题。

8-31 编写一个程序来模拟一个由AND、OR和NOT门以各种用户定义的方式连接而组成的数字系统。每个AND和OR门有两个输入和一个输出，而每个NOT门则有一个输入和一个输出。每个门由一个线程来实现，并从其他的门接收布尔值。这个程序的输入数据则是一个定义各个门的门函数和它们之间的互联关联的数组，例如表8-2定义了图8-15中所示的逻辑电路。请首先建立你的程序，使其能够模拟图8-15中所示的逻辑电路，然后修改这个程序，使其能够应用于最多8个门的任意逻辑线路。

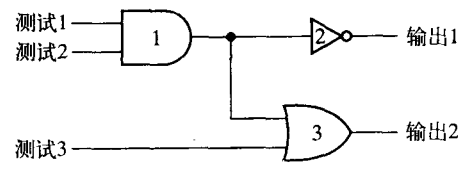


图8-15 示例的逻辑电路

表8-2 图8-15的逻辑电路的描述

门	功能	输入1	输入2	输出
1	AND	测试1	测试2	门1
2	NOT	门1		输出1
3	OR	测试3	门1	输出2

8-32 编写一个多线程程序来实现下面的拱廊街道（arcade）游戏：在一个河流中，一些圆木向下漂浮移动（或者上下移动）；在河流旁，一只青蛙想要过河；这个青蛙必须在圆木经过的时候，跳到这些圆木上青蛙才可以过河，如图8-16所示。青蛙只能垂直于河岸跳动，由用户来控制青蛙何时跳。如果青蛙达到对岸，那么你就赢了；如果青蛙掉

到河里你就输了。游戏的过程需要进行图形化的显示而且最好能够加上声音效果。每行圆木的并发移动由独立的线程来控制。（这个习题是由北卡罗来纳大学夏洛特分校的四年级学生Christopher Wilson在1997年作为短期的开放式作业（习题8-30）提出并实现的。其他的拱廊街道游戏也可用线程的方法实现。）

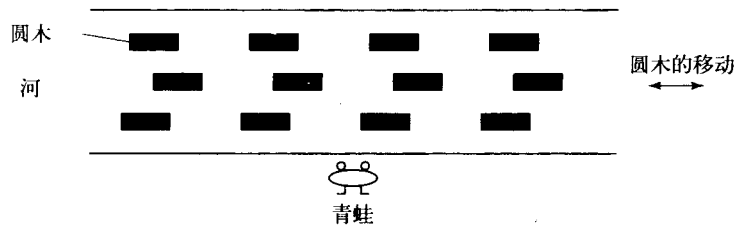


图8-16　习题8-32的河流和青蛙

8-33　在一个典型的加油站上有许多从一个油箱中拉出的抽泵，更复杂一些的情况是存储在油箱中的燃料通常只有最低和最高两档不同的燃料，但每个抽泵能提供从最高档油箱、
277　最低档油箱中的燃料或两者混合的中档燃料。用基于线程的并行实现来模拟一个大型的加油站，该加油站有多至20个抽泵从这两个储存油箱中提供燃料并间歇地有一辆运货车向油箱加入燃料。

8-34　使用主从结构组织的线程集编写一个简单的Web服务程序。主线程接收服务请求。当它得到一个新的请求后，它将从线程池中找到一个空闲的从线程来完成服务请求，如图8-17所示。（这个习题是由北卡罗来纳州立大学（North Carolina State University）的三年级学生Kevin Vaughan作为短期的开放式作业（习题8-30）在1997年提出并实现的。）

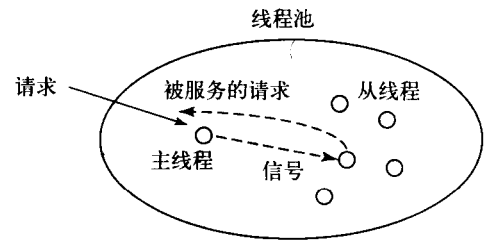


图8-17　习题8-34的线程池

8-35　选择在其他章中的一个现实生活习题，并用Pthread和MPI进行比较研究。测量每一种情况下的执行时间。再用顺序方法实现，并测量此时所需的执行时间。

8-36　重做习题8-35，但比较OpenMP和MPI的实现。

278　8-37　重做习题8-35，但比较Pthread和OpenMP的实现。

第9章 分布式共享存储器系统及其程序设计

本章主要论述如何在具有物理上分布和独立存储器的计算机机群上使用共享存储器程序设计模型。从程序设计角度讲，存储器是组合在一起的并为各个处理器所共享。这种方法被称为*分布式共享存储器*（DSM，distributed shared memory），并可用软件和/或硬件方法加以实现。我们将注重软件方法，除了要安装软件的代价外，软件方法只需很少甚至不需任何代价就可方便地在现有的机群上加以使用，尽管软件DSM的性能一般不如在同一机群上使用显式消息传递的方法。在分布式共享存储器系统上进行程序设计所使用的基本技术与在真正共享存储器上进行程序设计是一样的，后者已在第8章中作了叙述，但现在由于存储器在物理上是分布的，因而需对共享存储器模型作一些附加的考虑。第9章可在第8章后立即学习，也可在稍后学习。

9.1 分布式共享存储器

在第8章中，已叙述如何对一个共享存储器多处理机系统进行程序设计。在这种多处理器系统类型中，存在一个中央的“共享”存储器，每个处理器可对它直接访问。处理器与存储器以某种方式物理上连在一起，并形成一个单一的高性能计算机系统。共享存储器允许每个处理器在执行代码时对它进行直接的数据访问，而不是通过消息将数据从一台计算机发送给另一台计算机。对共享存储器进行程序设计一般比对消息传递进行程序设计更为方便，因为它允许各个处理器对任何大小的数据进行访问，而无需显式地向处理器发送数据。人们可无需复制地处理复杂和大型的数据库，但对共享数据的访问不得不由程序员用锁或其他方法加以控制。不论是共享存储器模型还是消息传递模型，通常需对进程进行同步，例如，在适当的场合使用障栅同步。

分布式共享存储器（DSM）是指使一组互联的计算机，尽管每台计算机拥有自己的存储器且在物理上是分布的，但看起来像具有单一编址空间的单一存储器，如图9-1所示。一旦实现了分布式共享存储器，那么任何处理器就可访问任何存储单元，而不管该存储器是否在本地，这样就可使用普通共享存储器的程序设计技术。当然，人们可以简单地使用共享存储器多处理机，但传统的总线互联的共享存储器多处理机只允许有限的处理器连到总线上，因而要将它们扩展成更大的系统时就很困难。为了扩大系统，就需要使用各种更为复杂的互联网络。与此相反，机群可以很容易地扩展为任何大小。用机群构成DSM的引人之处在于它的经济性，使用商品互联网可构成低廉的机群。

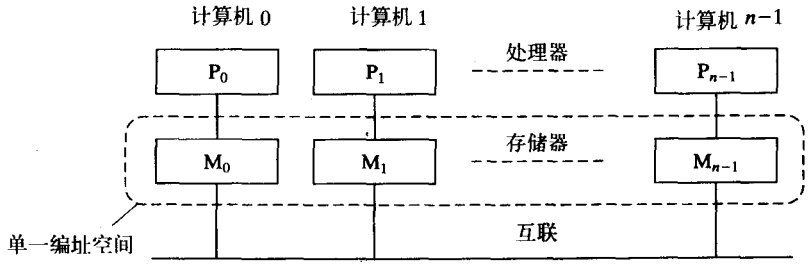


图9-1 分布式共享存储器

在机群上实现的DSM系统仍需传送消息以在计算机间移动数据，但是这种消息传递对用户来讲是透明的，因为用户不必在程序中显式地说明这些消息。简单地使用合适的共享存储器构造或例程访问共享数据就可引发所需的消息传递。而该发送什么消息、是否需复制数据、或真正地将数据从一台计算机移动到另一台计算机将由低层的DSM系统加以确定。如将要叙述的那样，可使用各种协议来进行这种消息传递。

DSM系统存在一些缺点。与真正的共享存储器多处理机系统相比，它所能提供的性能是比较低下的，因为独立计算机间的互联网络其运行速度远低于共享存储器多处理机系统中的互联网络速度。当然，对于给定的处理器数，机群有低得多的成本和更好的可扩展性。当与采用常规消息传递例程的机群相比时，DSM通常会在性能上略显逊色，因为可以预期由程序员插入的消息传递例程比自动的DSM方法会有更高的效率。但某些例子证明，这并非是永远正确的，因为DSM系统能利用优化或更聪明的协议，而对程序员来讲通常无法在自己的程序
280 中将它们识别出来。

一个机群可由一组单处理器系统、一组如图9-2所示的4奔腾系统那样的SMP多处理机系统或单处理器及多处理器的组合系统来构成。对SMP机群的程序设计会出现一些很有趣的可能性。借助消息传递程序设计或是在SMP计算机间用分布式共享存储器进行程序设计可在每个SMP计算机中完成真正的共享存储器程序设计。充分地使用SMP计算机就可创建一个单一的DSM环境。

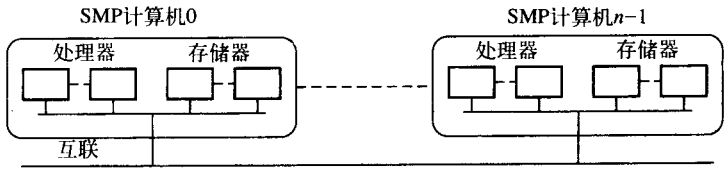


图9-2　SMP机群

9.2　分布式共享存储器的实现

自20世纪80年代中期以来，在研究团体中就已开始对DSM进行研究，DSM可用软件、硬件或是硬软件结合的方法加以实现。

9.2.1　软件DSM系统

在软件方法中，不需对机群的硬件作任何改变，一切只需借助软件例程就可完成。通常只需在操作系统和应用层之间增加一个软件层，而是否需对操作系统的核心进行修改则取决于具体的实现。该软件层可以是:

- 基于页面的
- 基于共享变量的
- 基于对象的

在基于页面的方法中，使用系统现有的虚拟存储器来引发计算机间的数据移动，如图9-3所示，这仅在要访问的页面不在本地时才会发生。该方法有时称为*虚拟共享存储器系统*(virtual shared memory system)。也许是[Li，1986]第一个开发了基于页面的DSM系统，之后又有一些基于页面的DSM系统，其中最著名的要数Rice大学[Amza et al.，1996]开发的TreadMarks。另一个使用虚拟存储器机制的一个分布式共享存储器系统的例子是Locust[Verma and Chiueh，1998]。

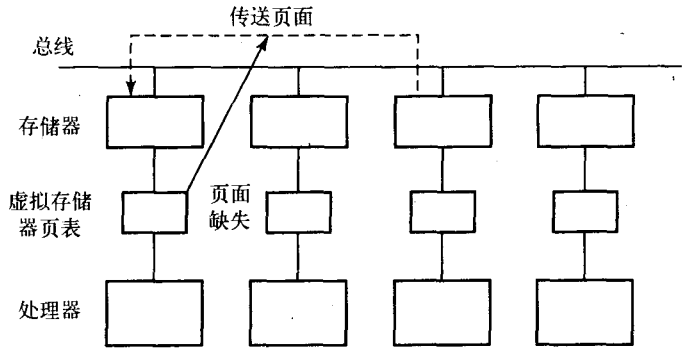


图9-3 基于页面的DSM系统

基于页面方法的主要缺点是由于要移动的数据是一个完整的页面，依赖于低层的虚拟存储器系统，它可以是1024或更多个字节，一般这总是多于所指定要访问的数据。从而导致要传送比所需的更长消息。此外，大的页面使假共享的影响在页面一级比在高速缓存一级更加 281
严重。在基于页面系统的环境中，假共享是指不同的处理器需要访问的是不同的页面部分，它们并没有真正共享实际的信息，但整个页面不得不为每个访问不同页面部分的处理器所共享。最后，基于页面的系统可能移植性不好，因为它们常常与某个具体的虚拟存储器硬件和软件捆绑在一起。

在共享变量方法中，传送的只是那些被声明为共享的变量，而这是按需加以说明的。它不使用分页机制来完成传递。相反，它由程序员通过直接或间接的调用软件例程来完成这一动作。该方法的一个例子是Munin [Benett, Carter, and Zwaenepoel, 1990]。其他更多的属于这一类的近期系统包括JIAJIA[Hu, Shi, and Tang, 1999]及Adsmith [Liang, King, and Lai, 1996]。后者虽是用C++编写的，但从用户看来是一个共享变量系统。稍后我们将更详细地叙述如何实现共享变量方法。如果性能不是一个关键因素，这种方法的实现将是非常容易的。

在面向对象方法中，共享数据被包含在对象中，对象包括数据项和其唯一的过程（方法），该方法可用来访问共享数据。在其他方面，面向对象的方法类似于共享变量方法，因而可被看成是共享变量方法的一种扩展。使用像C++或Java这样基于对象的语言可相当容易地实现这种方法，其胜过共享变量方法之处是在于它提供的是面向对象的规则。

9.2.2 DSM系统的硬件实现

在硬件方法中，系统需引入专用的网络接口和高速缓存一致电路，使对远程存储单元的访问如同对本地的存储单元访问一样。有不少专用的接口支持共享存储器的操作。实例包括虚拟存储器映射网络接口（Virtual Memory-Mapped Network Interface）[Blumrich et al., 1995]、Myrinet[Boden et al., 1995]、SCI [Hellwagner and Reinefeld, 1999]，以及存储器集成网络接口（Memory-Integrated Network Interface）[Minnich, Burns, and Hady, 1995]。 282

比起软件方法来，硬件方法会提供更高的性能。软件方法通常需要在操作系统和用户应用程序间增加一层额外的软件。有些甚至需要一个独立的消息传递层。此外，软件方法通常需要使用现有的商品接口（如以太网）从而导致显著的性能开销。就教学目的而言，纯软件方法比硬件方法更有吸引力，因为它可不加修改地就使用现有的计算机系统，为此下面将主要讨论软件方法。

9.2.3 对共享数据的管理

处理器可使用好几种方法来实现对共享数据的访问。最简单的方法是设置一个*中央服务器*(central server)由它负责所有的对共享数据的读和写操作，并使处理器向该服务器进行请求。对共享数据的所有读和写都发生在一个地方并且是顺序的，即它实现了一个*单阅读器/单写入器*策略。这一策略很少使用（在简单的学生作业中除外），因为所有的请求必须集中到一个地方，从而导致显著的瓶颈。如果使用多个服务器这一问题可得到某种程度的缓和，因为每个服务器只负责共享变量的一个子集，但此时需要一个映射功能以定位各个服务器。

一般希望有多个数据拷贝以允许不同处理器同时对该数据进行访问，此时必须使用*一致性策略*以维持这些拷贝的一致性。*多阅读器/单写入器策略*允许多个处理器同时读共享数据，但在任何瞬间只允许一个处理器改变此数据，将数据在需要它的地方加以复制，就可有效地实现这一点。这种策略只允许在一个场所（*拥有者*）改变该数据。

在多阅读器/单写入器策略中，当拥有者改变共享数据时，其他拷贝便不再正确。处理这一情况可使用以下两种方法:

- 更新策略
- 使无效策略

在更新策略中，通过一个广播消息使数据的所有其他拷贝立即改变以反映这一变化。在使无效策略中，所有其他的数据拷贝被标记为无效。如果它们在以后被访问，将得到一个指明它们是无效数据的响应，并使处理器向该数据的拥有者请求以获得该数据最近的值。一般都青睐使无效策略，因为仅当处理器试图访问已更新的拷贝时，才会产生消息，而任何在以后不再被访问的数据拷贝将保持无效不变。通常两种策略都需是可靠的。广播消息可能需要分别的确认动作的确认（reply）。

在多阅读器/多写入器策略中，会有多个数据拷贝，并允许由不同的处理器改变不同的拷贝。这是最复杂的情景，需要对每个写操作进行编号以对写操作进行排序。更多的细节可在
283 [Protic，Tomasevic，and Milutinovic，1996]中找到。

9.2.4 基于页面系统的多阅读器/单写入器策略

在基于页面的系统中，当访问一个共享变量而它又不在本地时，将传送含有该变量的整个页面。页面是共享的最小单位。存于该页面上不被共享的一个变量，当该页面上的另一变量被其他地方需要时，整个页面将被移动，则该变量将随该页面移动或是被作废（即可能出现假共享）。TreadMarks在处理这一问题时，允许页面的不同部分被不同的进程改变，但在同步点（即是页面级而不是共享变量级的多写入器协议）更新每个拷贝。假定两个进程正对一个页面的不同部分进行写入。每个进程在写入之前，首先为该页面生成另一个拷贝（*一个孪生页面*（twin））。该页面的不同拷贝仅在同步点才会变成一致。这是每个进程通过对该页面与它的未被修改的孪生页面按字比较为它的页面修改创建一个记录来完成的。所创建的“diff”是页面修改运行长度的编码。此后，每个diff被送到另一个进程，并允许它更改它的拷贝。有关TreadMarks的更多细节可在[Amza et al.，1996]中找到。

9.3 在DSM系统中实现一致性存储器

*术语存储器一致性模型*主要论及一个共享变量的当前值何时可被其他处理器看到。已经有各种各样的模型，它们以递降的约束条件来提供潜在的更高性能。最严格的模型是*严格一*

致性（strict consistency）模型。

严格一致性

严格一致性是指，当一个处理器读一个共享变量时，它得到的将是最近一次写入该共享变量的值。图9-4对严格一致性作了说明。例中，x和y是共享变量，一旦它们被改变，所有其他处理器将被告知这种改变（可采用使无效消息而不是更新消息。如前所述，使无效方法一般好于更新方法。）

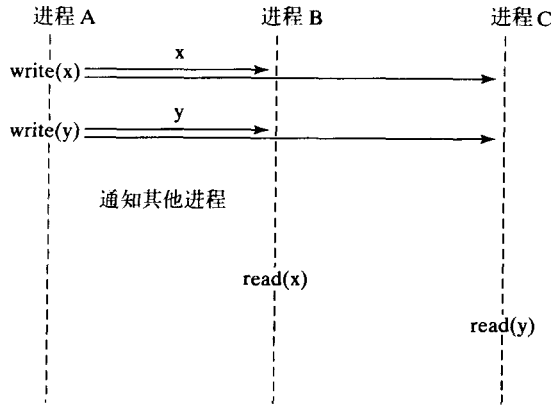


图9-4 严格一致性

284

严格一致性的主要缺点是它会生成大量的消息。另外，即使在该模型中，处理器在实际看到使无效/更新之前有一个延迟，这是因为改变不可能在瞬间完成。最近一次写的时间一般是不确定的，因为它依赖于各个处理器对指令的执行时间，而这些处理器的运行又是相互独立的。

在*松弛存储器一致性*（relax memory consistency）模型中，为减少消息数，其他处理器所看到的写已被延迟。存在有多种松弛存储器一致性的模型:

（1）*弱一致性*

在弱一致性（weak consistency）模型中，当需要强制实现顺序一致性时，由程序员使用同步操作来完成（参见8.4.3节）。这就允许编译器和处理器在其他地方重新排序指令，而不用考虑顺序一致性。这是一个相当合理的模型，因为对共享数据的任何访问将用同步操作（例如锁等）加以控制。

（2）*释放一致性*

释放一致性（release consistency）模型是对弱一致性模型的扩展，在该模型中，必须指明同步操作。程序员必须使用acquire和release这两个同步操作符:

acquire（获得）操作－在读共享变量之前使用。

release（释放）操作－在共享变量已被改变之后使用。

典型地，用一个加锁操作完成获得，而用一个开锁操作完成释放（尽管不一定非如此）。图9-5对释放一致性模型作了说明。在该例中由进程1更新共享变量x和y，而后由进程2读共享变量x和y。

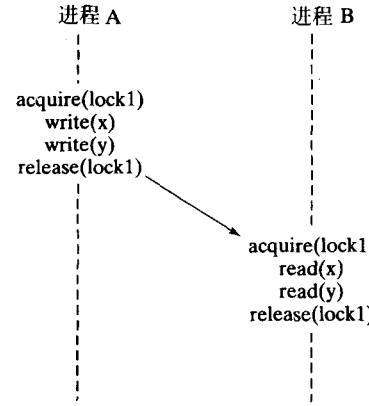


图9-5 释放一致性

（3）*滞后释放一致性*

对DSM系统来讲，关于释放一致性的流行观点是更新仅在acquire时完成，而不是在

release时完成，如图9-6所示。与释放一致性相比，滞后释放一致性（lazy release consistency）
285 将产生更少的消息。

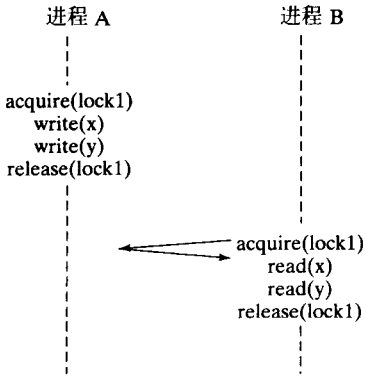


图9-6　滞后释放一致性

9.4　分布式共享存储器的程序设计原语

在共享存储器程序设计中，必须提供以下4种基本和必需的操作：

1）进程/线程的创建（和终止）
2）共享数据的创建
3）互斥同步（对共享数据的可控访问）
4）进程/线程和事件的同步

在一个DSM系统中也必须提供这些操作，典型地是通过用户级的库调用加以提供。

不同DSM系统的例程集，如Adsmith 和TreadMarks，实际上互相之间是非常类似的。这里我们将回顾在大多数DSM系统中可找到的公共例程，并使用带前缀dsm的通用例程名。特有的Adsmith例程带有前缀adsm。（注意字母串“dsm”是在A-dsm-ith中。）我们让学生使用Adsmith来完成课外作业的程序设计。

9.4.1　进程的创建

如果支持动态进程创建的话，一个如下的DSM例程

```
dsm_spawn(filename, num_processes);
```

将启动一个新进程。此后进程如要“会合（join）”，则可用

```
dsm_wait();
```

286 它将导致该进程等待所有它的子进程（即它所创建的进程）终止。

9.4.2　共享数据的创建

声明共享数据需要使用如下的例程或构造，如:

```
dsm_shared(&x);
```

它将为共享数据提供一个指针。在Adsmith 和TreadMarks的DSM系统中，将以C语言的malloc为共享数据动态地创建一个存储器空间:

```
dsm_malloc();
```

在使用后，该存储器空间可用例程

```
dsm_free();
```

加以释放。未在Adsmith 和TreadMarks中采用的一种更优美的方法是使用如下简单共享数据的声明:

```
shared int x;
```

更进一步，则可完全采用面向对象的设计，且在接口处将共享变量封装在对象和作用于其上的方法中。然而，无论怎样的声明，共享数据或对象通常有一个由创建例程选择的“宿主”单元，它靠近对它进行写入的进程。达到更高的性能将取决于访问模式，可移动（迁移）宿主单元以临近其他的处理器。

9.4.3 共享数据的访问

在一个使用松弛一致性模型的DSM系统（即大多数DSM系统）中，程序员需要显式地使用锁或其他机制防止来自不同进程对读/写的竞争访问。对共享变量sum的增量操作的代码序列如下:

```
dsm_lock(lock1);
dsm_refresh(sum);
   *sum++;
dsm_flush(sum);
dsm_unlock(lock1);
```

其中，lock1是一个与sum有关的锁变量。在加锁后，dsm_refresh()获得sum的当前值。然后，进程增量sum，接着dsm_flush()用一个新值更新sum的宿主单元。

对共享变量的某些访问可能不会发生竞争。例如，数据是简单地只读（永不改变），或各个进程以这样的方式同步，即不允许在同一时间有多于一个的进程访问共享数据。在这些情 287
况下，就不需要将访问放在临界区中。此时前一代码就可变为:

```
dsm_refresh(sum);
   *sum ++;
dsm_flush(sum);
```

如果变量只是读的，那么只需一个更新（refresh）就足够了；例如

```
dsm_refresh(sum);
a = *sum + b;
```

某些系统为不同类型的访问提供了高效的例程，以区分对共享变量的不同使用。例如Adsmith 提供了三种类型的访问:

- 一般访问——访问共享变量的规整赋值语句
- 同步访问——用于同步目的的竞争访问
- 非同步访问——不是用于同步的竞争访问

一个早期的系统，Munin [Carter，Bennett，and Zwaenepoel，1995]，将这一概念更延伸了一步，它共区分九种类型的访问（后来缩小为五种），但需由程序员选择合适的类型。

9.4.4 同步访问

与消息传递程序设计一样，进程同步以两种主要形式进行：全局同步和进程对的同步，必须对两者都加以提供。全局同步通常用障栅来完成，而进程对的同步，作为一个选项在同一例程中完成，或更好的是用独立的例程来完成。在消息传递的系统中，如在第6章中所叙述的那样，可简单地用同步发送/接收例程来完成进程对的同步。使用障栅时需要说明一个识别符以识别所指定的障栅:

```
dsm_barrier(identifier);
```

在使用现有的消息传递系统的系统中，消息传递系统中已有同步例程可供使用。当然，这些DSM系统也可提供它们自己的同步例程。

9.4.5 改进性能的要点

面向研究的DSM系统的基本目的之一是设计改进系统性能的方法，它通常包括重叠计算

288 和通信操作以及减少消息数。

1. 重叠计算与通信

重叠计算与通信的一种方法是使用一个“预取（prefetch）”例程在需要它的结果前启动一个非阻塞通信。预取例程应尽量在代码中向后插入，这受制于向后多远被传送的数据仍是最新的；例如，

```
    .
    .
    .
barrier();
dsm_prefetch(sum);          /* sum known to be up-to-date at this point */
    .
    .
    .
a = *sum + b;
    .
    .
    .
```

在预取且数据正被取出时，程序将继续执行。而在稍后的某个点将需要此数据。若该数据已到达，就可立即加以使用，否则执行将停止直到该数据到达为止。

预取甚至可猜测地进行，即纵然在某些情况下可能不需要此数据但仍预取它，如在以下的代码中：

```
    .
    .
    .
barrier();
dsm_prefetch(sum);          /* sum known to be up-to-date at this point */
    .
    .
    .
if (b == 0) a = *sum + b;
    .
    .
    .
```

这里，如果b不等于0就不需要sum，只需简单地将其丢弃。

预取机制非常类似于使用在某些先进处理器中的猜测装载（speculative-load）机制，它可重叠存储器操作和程序执行。存储器装载操作较费时间，而猜测装载在程序中需要该数据前的较早的位置处进行启动。在等待装载完成前，允许程序继续执行。如果在程序中将猜测装载的位置放得过前就会导致程序执行顺序发生改变（如在我们的例子中，在预取之上），则必须在合适的位置用专门的机制来处置存储器的异常（出错条件）。类似地，在一个DSM系统中当执行一个预取时也可能出现错误，因为本来这个预取可能不会出现。例如在我们上面的序列中，假定当b不等于0时在预取点sum是无效的，而当b等于0时为有效。如果不论b是什么值就进行预取，那么就可能出现一个异常，而该异常在没有预取时本不会出现的。程序员必须确定预取能安全的使用。

2. 减少消息的数量

289 可以通过提供一些组合基本例程的公共序列而构成的例程以减少消息数（“聚集”消息）。例如，前面使用4个例程的临界区：

```
dsm_lock(lock1);
dsm_refresh(sum);
   *sum ++;
dsm_flush(sum);
dsm_unlock(lock1);
```

可减少成两个:

```
dsm_acquire(sum);
   *sum ++;
dsm_release(sum);
```

例程dsm_acquire(sum)实际是将dsm_lock(lock1)和dsm_refresh(sum)组合在一起完成，类似地dsm_release(sum)完成的动作是将dsm_flush(sum)和dsm_unlock(lock1)组合在一起。在上述两种情况下，在实现中均可减少消息数。

可对每个产生它们自己消息的相同例程序列进行安排，使这些消息可被组合在一起。也可为一些公共操作提供高效的例程，例如，共享变量的例程可作为累加器使用。

9.5 分布式共享存储器的程序设计

在第8章中已提及，在机群上进行分布式共享存储器程序设计所使用的概念，与在一个共享存储器多处理机系统上进行共享存储器程序设计所使用的概念是一样的，但前者使用的是用户级库例程或方法。以第6章中所叙述的热分布问题为例，求解空间被分为二维的点数组，而每个点的取值是通过重复计算该点的4个邻点值的平均值得到的，直至这些值的求解收敛到足够的精度。在SPMD结构中，用DSM例程所写的代码的直接解释导致如下的代码（将一行分配到$n-1$个进程的每一个）:

```
dsm_sharedarray(*h, n*n);        /* Shared array int h[n][n], size n*n */
dsm_sharedarray(*g, n*n);        /* Shared array int g[n][n], size n*n */
dsm_shared(max_dif);        /*shared variable to test for convergence */
   .
   .
   .
i = processID;                          /* process ID from 1 to n-1 */
do {
   for (j = 1; j < n; j++)
      g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
   dsm_barrier(group);
   for (j = 1; j < n; j++) {          /* find max divergence/update pts */
      dif = h[i][j] - g[i][j];        /* dif in each process */
   if (dif < 0) dif = -dif;
   dsm_acquire(max_dif)
      if (dif < max_dif) max_dif = dif;     /* max_dif a shared variable */
   dsm_release(max_dif);
   h[i][j] = g[i][j];
   }
   dsm_barrier(group);                /* wait for all processes here*/
} while (max_dif < tolerance);        /* check convergence */
```

290

所出现的消息传递对用户来讲已被隐藏，但为此需要对效率进行附加的考虑。减少低层的消息传递是一个关键方面。该代码调用两个进程同步（障栅）和一个互斥同步（对一个临界区的获得/释放）。如在稍前所指出的那样，同步点将使执行顺序化，且将显著增加执行时间，所以应寻找减少同步点数目的方法，特别是在机群上执行时。在第6章中，我们叙述了使用无序松弛可将代码改成异步或是部分同步的，这也适用于DSM的程序设计。最后，应考虑采用在9.4.5节中所叙述的那些可提高性能的例程。

9.6 实现一个简易的DSM系统

编写你自己的简易DSM系统是相当直接的。在本小节，我们将论述如何来做到这一点。

在本章后的课程设计中就包括了创建你自己的DSM系统，它适宜作为扩充的课外作业。

9.6.1 使用类和方法作为用户接口

首先要确定的事情是用户程序设计方法学和用户接口。可以遵循用户级例程的方法，如在Adsmith 和TreadMarks中以及在9.4节中所叙述的方法一样。在C++和Java的面向对象方法学的基础上，我们已用更好的方法做了实验。对于共享数据，采用包装类可能更为合适。例如在Java中，我们可以写:

```
SharedInteger sum = new SharedInteger();
```

它扩展了整数类使其能提供使用在临界区中的方法lock、unlock、refresh和flush:

```
sum.lock();
sum.refresh();
   sum++;
sum.flush();
sum.unlock();
```

其中的lock和unlock方法隐含地使用一个与sum相关的锁。(当然，可采用独立的锁方法，如lock1.lock ( ) 和lock1.unlock ( )。)

291 可将该方法进一步扩展成具有组合方法lockandRefresh和flushandUnlock:

```
sum.lockandRefresh();
   sum++;
sum.flushandUnlock();
```

我们已实现的另一个非常新颖的方法是过载（overload）算术运算符，当书写以下的赋值语句时

```
x = y + z;
```

它会自动产生适当的动作以刷新y和z，其中y和z为共享变量。这些仅是供我们自己使用的一些不同的设计。

9.6.2 基本的共享变量实现

最简单的DSM实现方法是使用一个具有用户级DSM库例程的共享变量方法，如前面所定义的那样，这些库例程是建立在现有的消息传递系统，如MPI，之上的。如已所述的那样，这些例程可被包含在类和方法中。最基本的用户级DSM例程是共享变量的读例程和共享变量的写例程。写例程可包含加锁和开锁，或是用分别的例程对它们加以实现。例程能向一个中央单元发送消息，如图9-7所示，该中央单元负责管理共享变量。这相当于一个单阅读器/写入器协议。严格地讲，共享变量的加锁和开锁不是必须的，因为对中央服务器而言，它每次只

292 能做一件事，而没有其他进程能干扰它的动作。因此，针对整数共享变量的服务器代码将非常简单（采用我们惯用的伪代码):

```
do {
   recv(&command, &shared_x_name, &data, &source, any_source, any_tag);
   find(&shared_x_name, &x);  /* find shared variable, return ptr to it */
   switch(command)
   case rd:                        /* read routine */
      send(&x, source);            /* no lock needed */
   case wr:                        /* write routine */
```

```
        x = data;
        send (&ack, source);        /* send an acknowledgement update done*/
    .
    .
    .
} while (command != terminator);
```

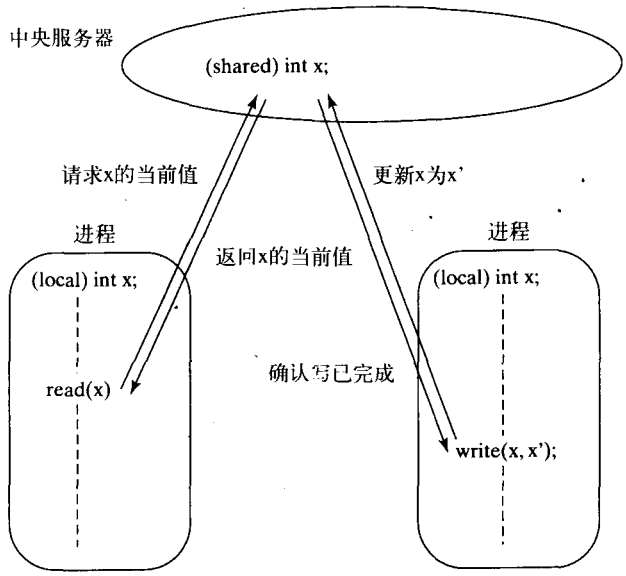


图9-7 使用中央服务器的简易DSM系统

消息由&command（由它指明所请求的操作（读、写或某些其他操作））、&shared_x_name（由它指明共享变量）、data（在写的情况下，它拥有用来更新共享变量的值）以及&source（由它识别发送消息的进程）所组成。

如所提及的那样，集中式的服务器将导致瓶颈，因为它每次只能响应一个消息和生成一个消息。这样的方法在任何理性的实际系统中都不会采用，但它却是一个好的起点。用这种方法去开发运行在不同处理器上的多个服务器的代码是相当简单的事情，其中，每个服务器负责指定的那些共享变量，从而可缓解瓶颈，如图9-8中所示。此时的读和写例程应能定位负责正被访问的共享变量的服务器。（通过一个查找表或是使用其他方法，如散列函数）。该方法仍使用单阅读器/单写入器协议。 293

对该模型的进一步开发就可使其具有多阅读器的能力，它通常需要对共享数据进行复制。假设我们使用使无效策略。图9-9中对这种多服务器的工作情况作了说明。这里仍使用我们的两例程模型，即读共享变量例程和写共享变量例程。在第一次调用读例程时，将从服务器获得最近的该共享变量值，并将此共享变量值保留在本地。有了本地拷贝后，就允许多个进程同时读它们的拷贝。如以前一样，写例程在一个服务器处更新共享变量，但现在是由一个指定的服务器负责该共享变量。此时，其他的本地拷贝，如存在的话，需被使无效。使无效的消息对接收进程来讲，是一个非期待的消息，因此可能需要使用单边（one-side）发送或放置（put）例程（不需要相应接收的发送例程，MPI-2中提供这一功能，但使用时需特别小心）。如果再次调用读例程，它将返回本地的拷贝值，如果该拷贝值是有效的话，此时就不会向服务器发送任何消息。如果本地拷贝为无效，表明有一个更新的值存在，此时便向服务器发送消息，以获得此最新值。异步算法可能不需要最近的值，因而可简单地使用赋值语句以获得x的本地拷贝而不是使用读例程。

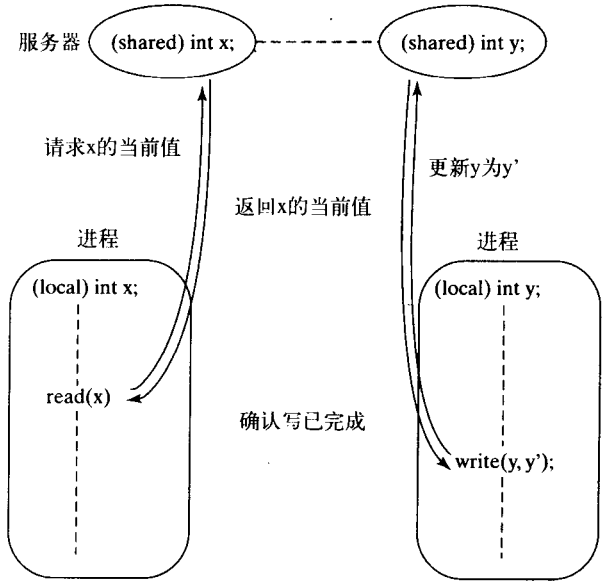


图9-8　使用多服务器的简易DSM系统

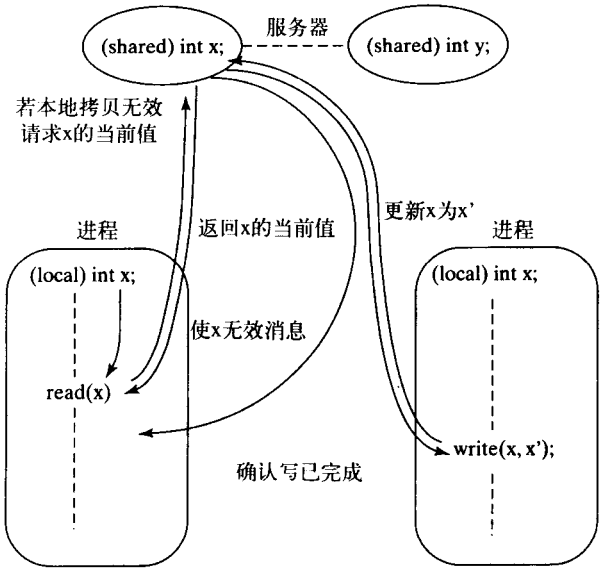


图9-9　使用多服务器和多阅读器策略的简易DSM系统

下一步是取消服务器并为每个共享变量提供一个宿主单元和一个进程负责该变量。读例程向该进程请求获得最近的值。事实上，代码不需作任何改变，只需简单地映射该服务器和指定作为共享变量的宿主进程到同一个处理器。使用宿主单元的优点在于在宿主单元进行写操作时不会产生任何消息。

294

9.6.3　数据组的重叠

在第1章中我们介绍了重叠互通网络的概念，在这种网络中提供了互通区域及区域的重叠。这种互通方式导致形成一个具有很好可扩展性的网络，从而可与许多科学与工程中的物理应

用相匹配。许多现实生活中的应用不需要全局共享存储器，而是只需局部的共享存储器，在那里对存储器的访问是在重叠区域内进行。实例包括许多物理系统的模拟（例如，物理学、机械工程、天气预报中的问题）。在这些应用中，很典型的情况是完成计算的处理器需要与逻辑上临近的那些处理器进行通信以获得最终结果。当然还有其他好处，包括能提供一个机制以检测程序中的逻辑错误。

为相同目的提供数据访问区域的概念可用到DSM系统中。让我们在对称多处理机机群（SMP）上来考虑这一概念。一个对称多处理机是一个共享存储器多处理机，其中每个处理器对共享存储器有相同的访问时间，而与处理器在共享存储器中的位置无关（即均匀存储器存取系统）。具有少量处理器的这种类型的系统是非常经济有效的，并已被广泛应用，特别是作为Web服务器。重叠局部共享存储器的概念考虑了SMP机群系统的物理结构。可以使用软件的DSM 技术，但对共享数据结构的访问只能限于由程序员定义的逻辑重叠组。也可以使用一组用户级的例程，这些例程将与机群上的消息传递软件相互作用，给用户一个共享存储器的感觉，并为共享存储器的程序设计提供基本设施，即创建共享数据、提供对共享数据（锁）的保护访问以及同步机制（障栅）。然而，与全局共享存储器系统的相反，这些例程需预先对共享数据重叠组加以定义。将MPI作为低层的消息传递软件是非常方便的，因为它已经有了概念性的通信区域（MPI通信子），可用来定义数据访问区域。图9-10对该系统作了说明。

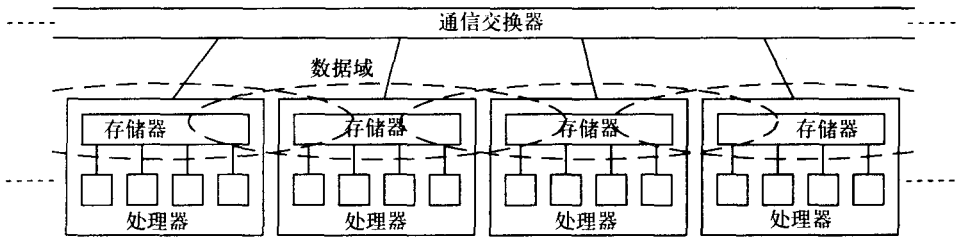


图9-10 具有重叠数据域的对称多处理机系统

重叠组将现有的互联结构和应用的访问模式两方面密切地结合起来。人们可以定义静态的重叠组，由程序员在程序执行前定义，或是在程序执行时动态地进行改变或创建。静态的重叠组可在基本的数据访问例程中用参数加以声明。例如， 295

```
create (data, data_region)
destroy(data, data_region)
read(data, data_region)
write(data, data_region)
```

最后，根据使用情况，共享变量可以迁移，如图9-11所示。

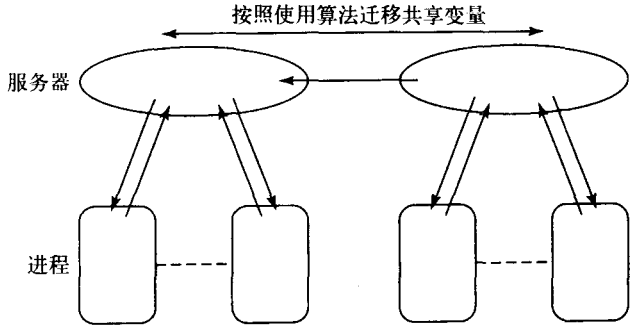
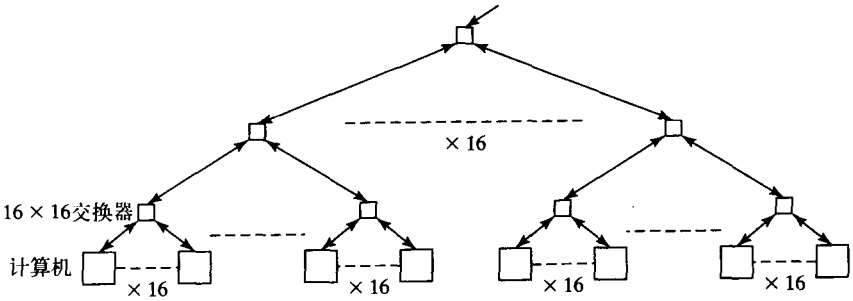


图9-11 使用多服务器和多阅读器策略的简易DSM系统

所有可扩展分布式共享存储器系统使用某种形式的层次互联结构将处理器组互联起来。这将内在地造成处理器间的非一致访问。一个可扩展的互联结构例子是n叉树，它适合于使用$\times n$的以太网交换器（100 Mbps或Gigabit以太网）。一个典型的商品以太网交换器能互联16台计算机（一个×16交换器），并允许与以太网交换器另一层次进行连接。图9-12示出了如何用商品部件来构成一个更大的系统。在这种情况下，在树的独立部分边界处的重叠数据区，其通信会有显著开销，而在部分树内部的数据区通信则将较为高效。



296

图9-12　使用商品交换器的可扩展系统

9.7　小结

本章介绍了以下内容：

- 分布式共享存储器（DSM）的概念
- 如何在机群上实现DSM
- 各种管理共享数据的方法，包括阅读器/写入器协议
- 针对DSM系统的松弛一致性模型，特别是释放一致性和滞后释放一致性
- 分布式共享存储器程序设计原语
- 改进DSM系统性能的方法
- 简易DSM实现的细节

推荐读物

有关分布式共享存储器的论文包括[Judge et al., 1999]、[Nitzberg and Lo, 1991]、[Protic, Tomasevic, and Milutinovic, 1996]和[Stumm and Zho, 1990]。[Protic, Tomasevic, and Milutinovic, 1998]发表了有关分布式共享存储器的一组重要论文。直接使用共享存储器程序设计工具是可能的，如果适当的加以实现，则单独地用在机群上也是可能的。例如，一个“大小相当的OpenMP子集”已由[Lu, Hu, and Zwaenepoel, 1998]在一个工作站网络上实现。

参考文献

AMZA, C., A. L. COX, S. DWARKADAS, P. KELEHER, H. LU, R. RAJAMONY, W. YU, AND W. ZWAENEPOEL (1996), "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer*, Vol. 29, No. 2, pp. 18–28.

BENNETT, J. K., J. B. CARTER, AND W. ZWAENEPOEL (1990), "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *2nd ACM SIGPLAN Symp. Principles & Practice of Parallel Programming*, March 14–16, pp. 168–176.

BLUMRICH, M. A., C. DUBNICKI, E. W. FELTON, K. LI, AND M. R. MESRINA (1995), "Virtual-Memory-Mapped Network Interface," *IEEE Micro*, Vol. 15, No. 1, pp. 21–28.

BODEN, N. J., D. COHEN, R. E. FELDERMAN, A. E. KULAWIK, C. L. SEITZ, J. N. SEIZOVIC, AND W. K. SU (1995), "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, Vol. 15, No. 1, pp 29–36.

CARTER, J. B., J. K. BENNETT, AND W. ZWAENEPOEL (1995), "Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems, *ACM Trans. Computer Systems*, Vol. 13, no. 3, pp. 205–243.

HELLWAGNER, H., AND A. REINEFELD (editors) (1999), *SCI: Scalable Coherent Interface*, Springer, Berlin, Germany.

HU, W., W. SHI, AND Z. TANG (1999), "JIAJIA: A Software DSM System Based on a New Cache Coherence Protocol," *Proc. 7th Int. Conf. on High Performance Computing and Networking Europe*, Amsterdam, pp. 463–472. Also available at http://www.ict.ac.cn/chpc/dsm.

LI, K. (1986), "Shared Virtual Memory on Loosely Coupled Multiprocessor," Ph.D. thesis, Dept. of Computer Science, Yale University.

LIANG, W.-Y., C.-T. KING, AND F. LAI (1996), "Adsmith: An Efficient Object-Based DSM Environment on PVM," *Proc. 1996 Int. Symp. on Parallel Architecture, Algorithms and Networks*, Beijing, China, pp. 173–179. Also see http://archiwww.ee.ntu.edu.tw/~wyliang/adsmith.

JUDGE, A., P. NIXON, B. TANGNEY, S. WEBER, AND V. CAHILL (1999), "Distributed Shared Memory," in *High Performance Cluster Computing*, Volume 1, R. Buyya (ed.), Prentice Hall, Upper Saddle River, NJ, Chapter 17, pp. 409–438.

MINNICH, R., D. BURNS, AND F. HADY (1995), "The Memory-Integrated Network Interface," *IEEE Micro*, Vol. 15, No. 1, pp. 11–20.

NITZBERG, B., AND V. LO (1991), "Distributed Shared Memory: A Survey of Issues and Algorithms," *Computer*, Vol. 24, No. 8, pp. 52–60.

PROTIC, J, M. TOMASEVIC, AND V. MILUTINOVIC (1996), Distributed Shared Memory: Concepts and Systems, *IEEE Parallel & Distributed Technology*, Vol. 4, No. 2, pp. 63–79.

PROTIC, J, M. TOMASEVIC, AND V. MILUTINOVIC (1998), *Distributed Shared Memory: Concepts and Systems*, IEEE Computer Society Press, Los Alamitos, CA.

STUMM, M., AND S. ZHO (1990), "Algorithms Implementing Distributed Shared Memory," *Computer*, Vol. 23, No. 5, pp. 54–64.

SUN, X.-H., AND J. ZHU (1995), "Performance Considerations of Shared Virtual Memory Machines," *IEEE Trans. Parallel and Distributed Systems*, Vol. 6, No. 11, pp. 1185–1194.

WILKINSON, B., T. PAI, AND M. MIRAJ (2001), "A Distributed Shared Memory Programming Course," *Int.Workshop on Cluster Computing Education (CLUSTER-EDU_2001)*, IEEE Int. Symp. Cluster Computing and the Grid (CCGrid), Brisbane, Australia, May 16–18.

WOO, S. C., M. OHARA, E. TORRIE, J. P. SINGH, AND A. GUPTA (1995), "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int. Symp. Computer Architecture*, pp. 24–36.

习题

在下面，当编写一个DSM程序时，可使用任何你能获得的DSM系统。在第8章中所提供的共享存储器习题也适合DSM的实现。

科学/数值习题

9-1 编写一个DSM程序，用奇偶互换排序对n个整数数列进行排序，假定n是2 的乘方。要求对你的程序进行清晰的说明。

9-2 编写一个矩阵相乘的DSM程序，需将矩阵分成4个子矩阵，并使用4个进程。要求对你的程序进行清晰的说明。

9-3 就你自己所选的问题，对DSM程序设计和消息传递程序设计进行比较研究。

297 ~ 298

9-4　对并行系统的研究，常常采用基准测试程序，其中一套程序SPLASH-2 [Woo, 1995]，含有应用代码，如Barnes-Hut的*N*体计算L-U矩阵分解、以及模拟。设法获得这套基准测试程序，并在你自己的DSM并行/机群计算机系统上对它进行评估。

9-5　对第6章的习题6-14所叙述的用同步迭代和异步迭代方法求解热分布问题进行比较研究。为每一种方法编写DSM程序，并如在9.5节中所叙述的那样确定使用异步方法在执行速度上的改进。对在同步点间采用不同的迭代步数进行实验。

现实生活习题

9-6　Big-I是世界上最大的某保险公司，它的办公室分散在世界各地的35个城市中，并有超过735 000 000张的有效保险单，总面值为530.8亿美元。一个高速的办公室间的网络链接了Big-I的35个办公室和它的中央总部（位于洛基山脉某处的一个安全的地下设施中。）所有的保险单信息被保存在总部的RAID-5的存储系统中。

由代理输入有关新保险单的信息，改变现有的保险单，以及日常的客户关系的变化（结婚/离婚、出生、死亡、医疗记录、法院记录等）。源自世界任何地方的这些信息可能与某一具体客户有关。例如，在美国Big-I保险的一个客户Fred Jones旅游到巴黎去庆祝他的50岁生日，在那里生了病并作了手术；同一天在北卡罗莱纳州对他的离婚作了终判。Big-I的代理在巴黎和北卡罗来纳州要同时访问/更改Fred的记录以反映有关他的医疗情况和婚姻状况的新信息。

描述有关Big-I的中央存储系统的设计问题。描述如果中央存储系统需镜像到几百英里以外的第2个地下设施中时应对设计进行何种改变。

Big-I刚好到任一位新执行总裁，他认为中央存储的思想已经过时，并想用分布式方法取而代之。他提议36个机构的每一个维持自己的本地信息存储，但仍能输入和访问所有有关一个客户的信息，而不论该信息是在何处输入/存储的：一个全球的分布/共享存储系统。描述将中央存储的信息分布到世界各地35个办公室的一个转变计划。叙述Big-I的IT部门如何能模拟网络对分布式共享存储系统的影响以及网络对一个中央存储但允许（可能）同时访问的系统的影响，并对两者加以比较。

9-7　Adamms家庭享受着非常亲密关系的乐趣。Tom和Sue Adamms已结婚23年，并有8个子女，所有成员至少在3项运动上很活跃，再加上教堂和学校。Tom和Sue都是一家大型保险公司Big-I的不同部门的高层行政管理人员，他们的日程表通常有10到30个约会/会议事项。随着子女们的长大，以及他们更多地潜心于个人的活动日程而不是Tom和Sue的活动日程，几乎可以肯定会出现小的灾难！就在上个星期，Polly（9岁）排定要参加她所在球队的一场足球比赛，但这一信息在Tom和Sue的日程表中都没有反映。因为过少的队员出现在球场上导致Polly的队取消了该比赛；一个星期后Polly仍为此事而闷闷不乐。

有关可供整个家庭访问（及修改）的中央日程表系统的设计应涉及哪些方面？就日程安排而言，如果某个人相对于其他人确有优先权，则应作何种修改？（孩子们可输入/改变只影响他们自己的事项，但不能改变安排在他们父母日程表上的事项。另一个经常会发生的情况是，Tom应能输入/改变Sue日程表上一项非公务的事项，但不能输入/改变一个不同类的事项：公务事项。）

DSM的实现设计项目

9-8 用C++编写一个DSM系统，采用MPI作为低层的消息传递和进程通信。

9-9 用Java编写一个DSM系统，采用MPI作为低层的消息传递和进程通信。

9-10 （更难的习题）软件DSM系统的一个根本缺点是缺乏对低层消息传递的控制。在DSM例程中提供参数就能控制消息传递。编写允许通信和计算可重叠执行的例程。

300

第二部分 算法和应用

第10章 排序算法
第11章 数值算法
第12章 图像处理
第13章 搜索和优化

第10章 排序算法

本章讨论一些关于数字排序方法。人们通常使用顺序程序设计方法来研究排序并得到许多著名排序方法。这里我们选择几个常用的顺序排序算法来将它们转换成并行实现。本章还将描述几个特地为并行实现而设计的排序算法。最后，将讨论一些近期受关注的在机群上实现的排序算法。

10.1 概述

10.1.1 排序

数字排序，即将一系列数字按升序（或降序）进行排列，是很多应用中的一个基本操作。如果在这一系列数字中有重复的数字存在，那么更合理的排序定义应为：将一系列数字按非降序（或非升序）进行排列。排序也被应用于非数字领域，如将一些字符串按字母表顺序排列。人们使用排序是为了方便有些操作如查找。为了方便读者查找指定的书籍，图书馆将藏书按主题/书名依次排列。

有两种排序算法用于示范特殊的并行技术。在4.2.1节提到用桶排序来演示分治策略，在5.2.2节中使用插入排序来示范流水线技术。本章将观察另外一些排序算法并对这些算法使用最合适的并行化技术。许多并行排序算法和顺序排序算法的并行实现都是同步算法，在这些
303 算法中对数字一组操作必须等待上一组操作完成后才能进行，因此它们可使用第6章叙述的同步迭代技术。

10.1.2 可能的加速比

快速排序和归并排序是常用的顺序排序算法。它们属于“基于比较”的排序算法，即在比较数对的基础上进行排序。如对n个数字排序，在最差情况下归并排序的时间复杂性和快速排序的平均时间复杂性都是$O(n\log n)$。如果不使用数字的特殊属性，$O(n\log n)$实际上是所有基于比较的顺序排序算法中最佳的。因此如果我们使用p个处理器基于顺序排序算法的最好的并行时间复杂性为:

$$\text{最佳的并行时间复杂性} = \frac{o(n\log n)}{p} = O(\log n)\text{，如果}p = n$$

[Leighton，1984]提出一个使用n个处理器、时间复杂性为$O(\log n)$的排序算法，该算法基于[Ajtai，Komlós，and Szemerédi，1983]所提出的算法，但隐藏在阶符号中的常量非常大。[Leighton，1994]则为由n个处理器组成的、使用随机操作的超立方体结构提出了一个时间复杂性为$O(\log n)$的排序算法。[Akl，1985]描述了20个不同的并行排序算法，其中一些算法对于特殊的互联网络达到了较低的下限。但总的来说，用n个处理器的基于比较的顺序排序算法实际要达到$O(\log n)$并不是一件容易的事。为达到这个目标使用的处理器数可能超过n。下面让我们首先讨论传统的基于比较的算法。

10.2 比较和交换排序算法

10.2.1 比较和交换

比较和交换操作是一些（如果不是大多数）顺序排序算法的基础。在比较和交换操作中，两个数，记为A和B，进行比较。如果$A>B$，那么A和B 单元中的内容交换，也就是说，保存A的单元内容被移到保存B的单元中而保存B的单元的内容被移到保存A的单元中，否则单元的内容保持不变。以下的顺序代码描述了该比较和交换：

```
if (A > B) {
   temp = A;
   A = B;
   B = temp;
}
```

比较和交换很适合于消息传递系统。假设比较和交换在两个数A和B之间进行，A存放在进程P_1中而B存放在P_2中。执行比较和交换操作的一个简单办法是P_1将A发向P_2，由P_2比较A和B，如果A大于B则将B发回P_1，否则将A发回P_1。P_2不是必须将A发回P_1，因为P_1已经有A了，但这样做可以使无论在何种情况下（A大于B或相反）发送和接受操作的次数一样。图10-1展示了这种方法，相关代码如下：

304

进程P_1

```
send(&A, P2);
recv(&A, P2);
```

进程P_2

```
recv(&A, P1);
if (A > B) {
   send(&B, P1);
   B = A;
} else
   send(&A, P1);
```

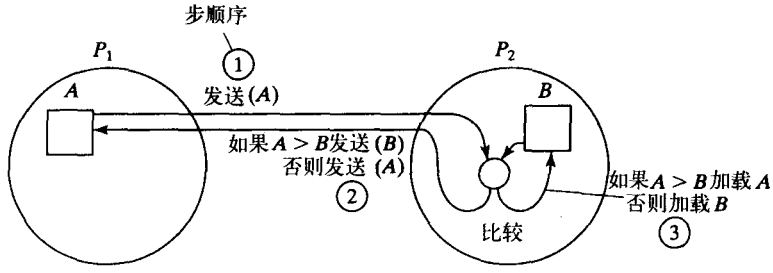


图10-1 消息传递系统中的比较和交换——版本1

另外一种实现方法是P_1将A发向P_2 的同时P_2将B发向P_1，然后两个进程同时进行比较操作，P_1保留较小的数，而P_2保留较大的数，如图10-2所示，该方法的代码如下：

305

进程P_1

```
send(&A, P2);
recv(&B, P2);
if (A > B) A = B;
```

进程P_2

```
recv(&A, P1);
send(&B, P1);
if (A > B) B = A;
```

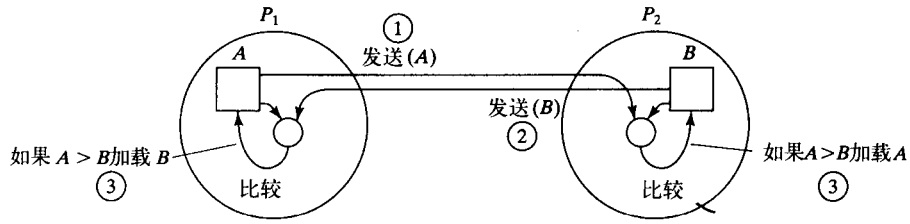


图10-2　消息传递系统中的比较和交换——版本2

进程P_1先执行send()，同时进程P_2先执行recv()以防止死锁（在版本1中send()和recv()的顺序不会导致死锁）。如果使用本地阻塞（异步）发送并且有足够大的缓冲区，那么P_1和P_2就可以都先执行send()，这样两个进程就可以通过同时初始化它们的消息传输来覆盖消息传输的开销。我们必须指出在MPI意义上讲这种编程方式并不安全，因为如果系统没有提供足够的缓冲区，死锁就可能发生。

1. 注意重复计算的精确性

上面的代码假设两个处理器的if条件A>B会返回相同的布尔答案。但当进行实数比较时，以不同精确度运行的处理器可能得到不同的答案。任何为了降低消息传递量而在不同处理器上的重复的计算都可能发生这种情况。在我们的代码中消息传递量并没有减少，但对所有进程来说使每个进程的代码看上去类似可以使这个单一程序更容易构造。

2. 数据划分

虽然到目前为止我们假设一个数分配到一个处理器，但通常待排序数的数量远远大于处理器（或进程）的数量。在这些情况下，每个处理器可能分配到一组数。这个方法可以用于所有排序算法。假设有p个处理器和n个数。每个处理器会分配到一列数量为n/p的数。比较和交换操作将基于图10-1（版本1）或图10-2（版本2）。图10-3显示了版本1的运作。只有一个处理器将分配给自己的数传送到另一台处理器，后者在进行完归并操作后将这列数较小的一半传回第一个进程。图10-4显示了版本2的运作，两个处理器交换它们的组，在总共$2n/p$个数中一个处理器将留下较小的n/p个数，另一个则留下较大的n/p个数。一般方法是在每个处理器上保存一组已排完序的数列，将存储的数列与得到的数列进行归并，丢弃归并后数列的上半段或下半段。同样在这种方法中，假设每个处理器将产生相同的结果，并按相同的方法来划分数据。在任何情况下，归并两组大小为n/p的数列需要$2(n/p)-1$步和两次消息传递。归并两个已排序的数序列在排序算法中是通用操作。

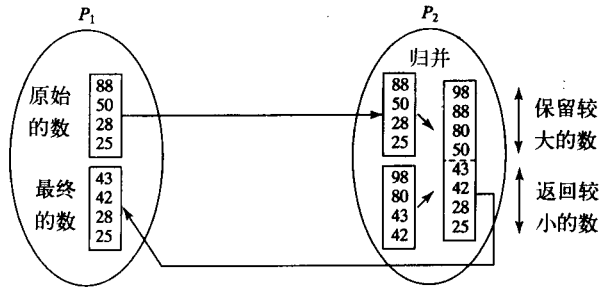


图10-3　归并两个子序列——版本1

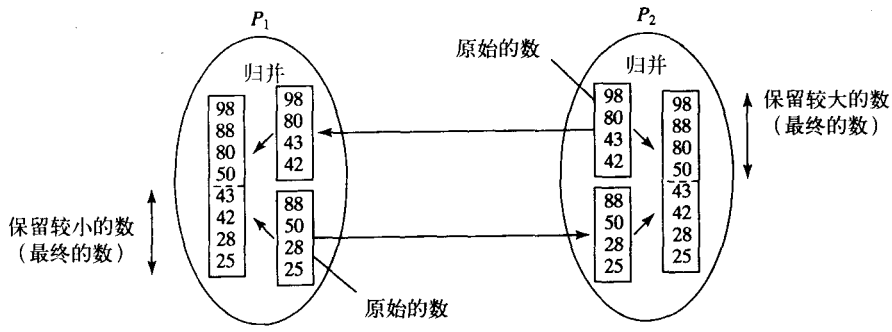


图10-4 归并两个子序列——版本2

比较和交换可以用于重排数字对的顺序，同时比较和交换的反复使用可以完成一个数列的排序，它出现在快速排序和归并排序中。但首先让我们考虑冒泡排序，它是使用比较和交换最典型的排序算法之一，虽然对一台顺序计算机来说它不如快速排序和归并排序有吸引力。 306

10.2.2 冒泡排序与奇偶互换排序

在冒泡排序（bubble sort）中，首先通过一系列的比较和交换将最大的数首先移动到序列的一端，而比较和交换则在另一端开始。对于给定的一系列数x_0，x_1，x_2，…，x_{n-1}。首先将x_0和x_1进行比较，较大的数移动到x_1（较小的数则移动到x_0），然后x_1和x_2进行比较，较大的数移动到x_2，依此类推直到最大的移动到x_{n-1}。反复执行这些操作直到上一次最大数的前一个位置，以得到次最大数，对每个数都执行这些操作。通过这种方法，较大的数就向数列的一端移动（“冒泡”），图10-5中对8个数的序列进行排序的情况做了说明。

对n个数来说，在第一阶段得到最大数并将其移动到序列的一端的过程中有$n-1$次比较和交换操作。在第二阶段得到次最大数需要$n-2$次比较和交换操作，依此类推。因此总的操作次数为：

$$\text{比较和交换操作次数} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

307 ~ 308

这表示它的时间复杂性为$O(n^2)$，一次比较和交换操作有固定的复杂性：O(1)。

1. 顺序代码

假设待排序数存放在数组a[]中，则顺序代码可为：

```
for (i = n - 1; i > 0; i--)
   for (j = 0; j < i; j++) {
      k = j + 1;
      if (a[j] > a[k]) {
         temp = a[j];
         a[j] = a[k];
         a[k] = temp;
      }
   }
```

2. 并行代码——奇偶互换排序

冒泡排序是一个纯顺序算法。内循环中的每一步发生在下一步前并且整个内循环在外循环的下一次迭代前完成。但在顺序代码中使用依赖前一条语句的指令并不意味着它不能被改写为并行算法。内循环的下一次迭代的“冒泡”动作可以在前一次迭代完成前开始，只要下一次“冒泡”动作不影响前一次迭代。这意味着流水线的结构将比较适合。

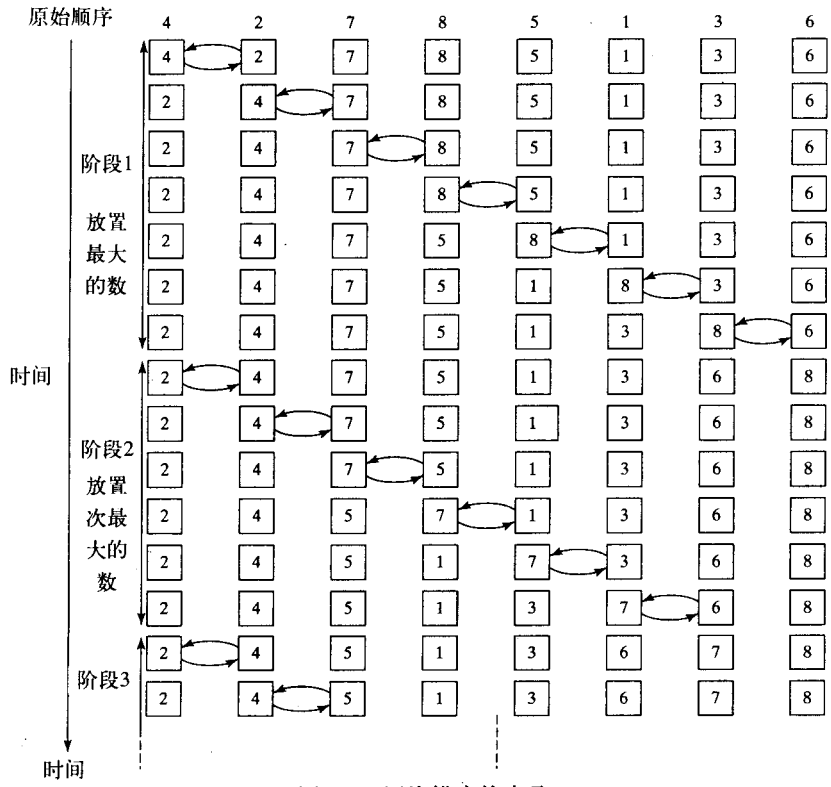


图10-5　冒泡排序的步骤

309　图10-6说明在流水线中冒泡排序的随后交换操作是如何在其他操作之后完成的。我们可以发现如果迭代1和迭代2可由一个独立的进程加以分开，则迭代2可以在迭代1之后同时进行，同样迭代3可以在迭代2之后同时进行。

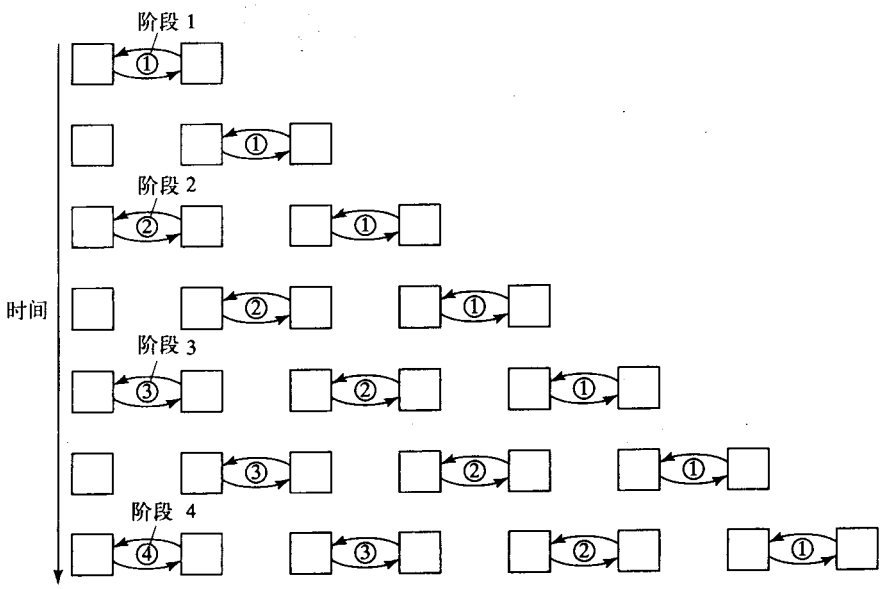


图10-6　流水线中冒泡排序动作的重叠

这个想法导致冒泡算法的变形即奇偶（互换）排序odd-even(transpositiow) sort的产生，该算法运行于两个交替阶段，奇阶段和偶阶段。在偶阶段偶数编号进程与它们的右邻居交换数据，同样在奇阶段奇数编号进程与它们的右邻居交换数据。奇偶互换排序通常不会在顺序程序设计中加以讨论，因为和普通冒泡算法相比它没有特殊的优点。但并行实现可以将它的时间复杂性降到O(n)。奇偶互换排序可以在线形网络中实现并且在该网络中是最佳的（因为在最差情况下重新放置一个数仅需要n步）。图10-7显示了奇偶互换排序如何应用在8个数的序列中，每个进程存放一个数。

310

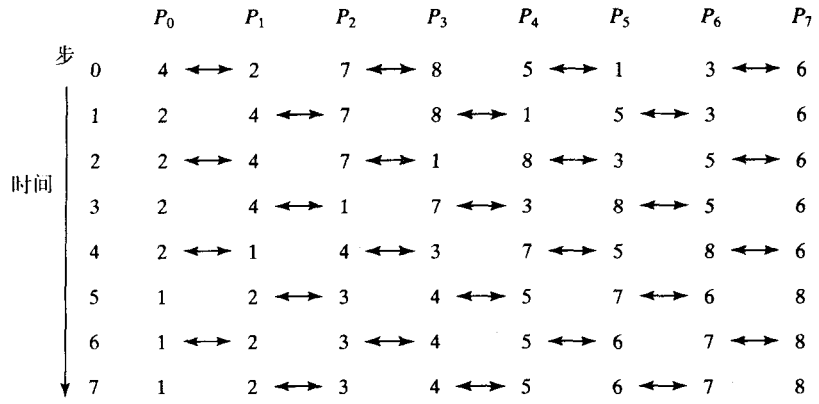


图10-7 对8个数进行奇偶互换排序

首先让我们分别观察两个不同的运算阶段。在偶阶段，我们有以下比较和交换：$P_0 \leftrightarrow P_1$、$P_2 \leftrightarrow P_3$等等，使用与10.2.1节中提到的版本2相同形式进行比较和交换，代码如下：

P_i，$i = 0, 2, 4, \cdots, n-2$（偶数）

```
recv(&A, P_{i+1});
send(&B, P_{i+1});
if (A < B) B = A;
```

P_i，$i = 1, 3, 5, \cdots, n-1$（奇数）

```
send(&A, P_{i-1});        /* even phase */
recv(&B, P_{i-1});
if (A < B) A = B;         /* exchange */
```

P_i(偶)中存放的是B，而P_i(奇)中存放的是A。在奇阶段我们则有以下比较和交换：$P_1 \leftrightarrow P_2$、$P_3 \leftrightarrow P_4$等等，相应代码为：

P_i，$i = 1, 3, 5, \cdots, n-3$（奇数）

```
send(&A, P_{i+1});
recv(&B, P_{i+1});
if (A > B) A = B;
```

P_i，$i = 2,4,6, \cdots, n-2$（偶数）

```
recv(&A, P_{i-1});          /* odd phase */
send(&B, P_{i-1});
if (A > B) B = A;           /* exchange */
```

无论在哪个阶段都是奇数编号进程首先执行`send()`例程，而偶数编号进程首先执行`recv()`例程。综合起来的代码为：

P_i，$i = 1,3,5, \cdots, n-3$（奇数）

```
send(&A, P_{i-1});
recv(&B, P_{i-1});
if (A < B) A = B;
if (i <= n-3) {
   send(&A, P_{i+1});
   recv(&B, P_{i+1})
   if (A > B) A = B;
}
```

P_i，$i = 0,2,4, \cdots, n-2$（偶数）

```
recv(&A, P_{i+1});          /* even phase */
send(&B, P_{i+1});
if (A < B) B = A;
if (i >= 2) {               /* odd phase */
   recv(&A, P_{i-1});
   send(&B, P_{i-1});
   if (A > B) B = A;
}
```

这些代码段可以合成为一个单程序多数据（SPMD）格式的程序，其中的进程的标识符可以用来选择特定处理器将执行程序的那部分代码（习题10-5）。

10.2.3 归并排序

归并排序是使用分治方法的经典顺序排序算法。首先将待排序的序列一分两半，每一半又被分成两半，一直进行到得到单个的数，然后数字对被合并到一个每2个数的有序序列中，4个数的有序对被合并到8个数的有序序列中，这样一直继续到得到一个完整的已排序完毕的序列。根据以上描述，该算法可以很好地映射到第4章所提到的树结构上去。图10-8展示了该算法如何对8个数进行排序。可以看到这个结构与用来分割问题（图4-3）、合并问题（图4-4）的树结构非常相似，因此图4-3和图4-4中的处理器分配策略也可以使用。

使用树结构的一个明显缺点是处理器之间的负载不能很好地平衡。一开始只有一个处理器在运行，然后是两个，接着是四个，以此类推，参加计算的处理器数到达最大值后又逐步减少。

分析

顺序时间复杂性是$O(n\log n)$，在图10-8所示的并行版本中共有$2\log n$步，但根据待排序数的多少，每步可能需要完成不止一个基本操作。让我们假设数据的部分序列被分配到各自的处理器中，然后开始归并排序。

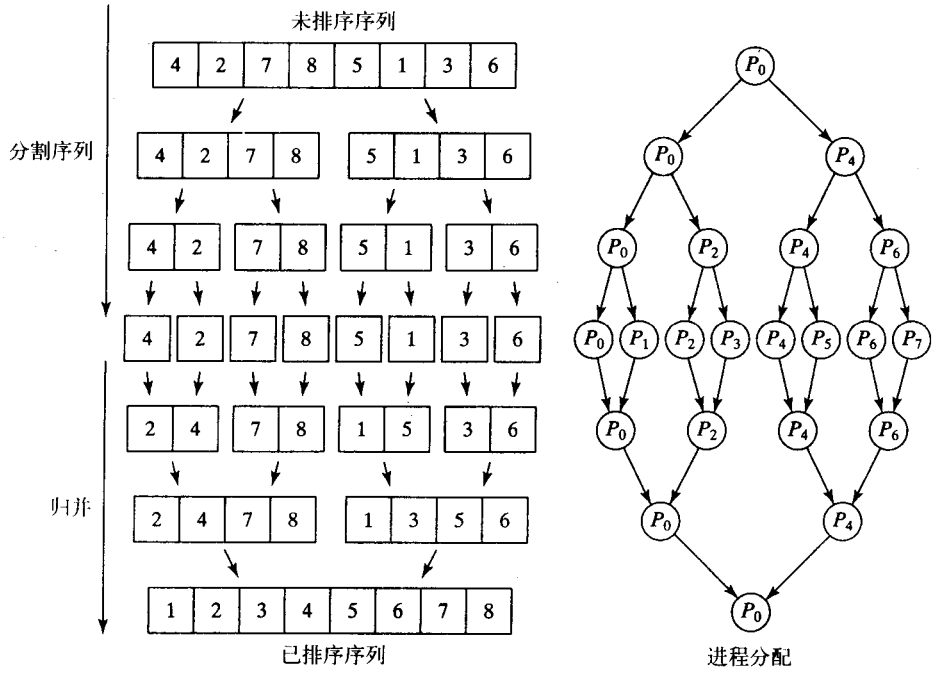


图10-8　使用树状进程分配的归并排序

（1）通信　在分割阶段，只发生以下通信：

每步通信耗时	处理器通信
$t_{startup} + (n/2)t_{data}$	$P_0 \rightarrow P_4$
$t_{startup} + (n/4)t_{data}$	$P_0 \rightarrow P_2$; $P_4 \rightarrow P_6$
$t_{startup} + (n/8)t_{data}$	$P_0 \rightarrow P_1$; $P_2 \rightarrow P_3$; $P_4 \rightarrow P_5$; $P_6 \rightarrow P_7$
⋮	

311

若有p个处理器，则需进行$\log p$步。在归并阶段发生相反的通信：

$$\vdots$$

$t_{startup} + (n/8)t_{data}$	$P_0 \leftarrow P_1; P_2 \leftarrow P_3; P_4 \leftarrow P_5; P_6 \leftarrow P_7$
$t_{startup} + (n/4)t_{data}$	$P_0 \leftarrow P_2; P_4 \leftarrow P_6$
$t_{startup} + (n/2)t_{data}$	$P_0 \leftarrow P_4$

同样进行log$p$步。因此总的通信时间为:

$$t_{comm} = 2\ (t_{startup} + (n/2)\ t_{data} + t_{startup} + (n/4)\ t_{data} + t_{startup} + (n/8)\ t_{data} + \cdots)$$

即

$$t_{comm} \approx 2\ (\log p)\ t_{startup} + 2nt_{data}$$

(2) 计算 计算仅仅发生在归并子序列阶段。通过遍历每个子序列来进行归并，并把找到的最小数先移进存储计算结果的序列中。在最坏情况下归并两个分别有n个数的已排序序列到一个结果序列需要$2n$-1步。计算步骤如下:

$t_{comp} = 1$	$P_0; P_2; P_4; P_6$
$t_{comp} = 3$	$P_0; P_4$
$t_{comp} = 7$	P_0

312

计算时间:

$$t_{comp} = \sum_{i=1}^{\log p} (2^i - 1)$$

如果使用p个处理器并且每个处理器存放一个数，那么并行计算的时间复杂性为O(p)。通常在所有排序算法中将数据划分为若干组，每组分给一个处理器进行排序。

10.2.4 快速排序

快速排序[Hoare, 1962]是一种非常流行的顺序排序算法，它的平均顺序时间复杂性很好，为O($n$log$n$)。需要回答的问题是该算法的直接并行版本（n个处理器）能否达到O(log$n$)的时间复杂性。根据上节的分析归并排序算法的性能并不理想，现在我们来检验一下作为并行排序算法基础的快速排序。

让我们从顺序程序设计角度来回忆一下，和归并排序一样，快速排序算法首先将数列分为两个子序列。其中一个子序列中的所有数字比另一个子序列中的所有数字都小。这可以通过以下方法实现，首先选择一个数，称为枢轴（pivot），用它与其他数进行比较。如果一个数比枢轴小就被放入一个子序列，否则放入另一个子序列。枢轴可以是序列中的任何数，但人们通常选序列的第一个数作为枢轴。枢轴本身可以放入一个子序列，或者枢轴被分离出来并放在它的最终位置，我们选择分离枢轴的方法。

在子序列上重复上述过程，创建四个子序列，实质上像4.2.1节提到桶排序一样将数字分别放入四个子序列区域，所不同的是区域的大小由每步所选择的枢轴来决定。经过多次重复后，每个子序列中都只有一个数字，这样我们就得到一个有序的序列。

1. 顺序代码

人们经常使用递归算法来描述快速排序。假设数组`list[]`存放待排序的数字，`pivot`是枢轴最终位置的数组下标。我们可得到以下形式的代码:

```
quicksort(list, start, end)
{
  if (start < end) {
```

```
        partition(list, start, end, pivot)
        quicksort(list, start, pivot-1);   /* recursively call on sublists*/
        quicksort(list, pivot+1, end);
    }
}
```

partition()移动在list数组中start和end之间的数字，使得小于枢轴的数字被移动到枢
313 轴前面，而大于等于枢轴的数字被移动到枢轴后面。此时枢轴已处在排序好序列的最终位置。

2. 快速排序的并行化

并行化快速排序的典型方法是在一个处理器上开始然后将一个递归调用传给另一个处理器计算，而保留另一个递归调用自己执行。这样就得到一个和归并排序相似的树结构，如图10-9所示。在这个例子里，枢轴被分配到左边的子序列。注意，位于两个子序列之间的枢轴位置就是枢轴在序列中的最后位置，并且在以后的排序中这个数就不用再考虑了。将树重画一下来显示枢轴是如何被隐藏的，然后通过按序遍历树就可以得到排完序的序列了，如图10-10所示。

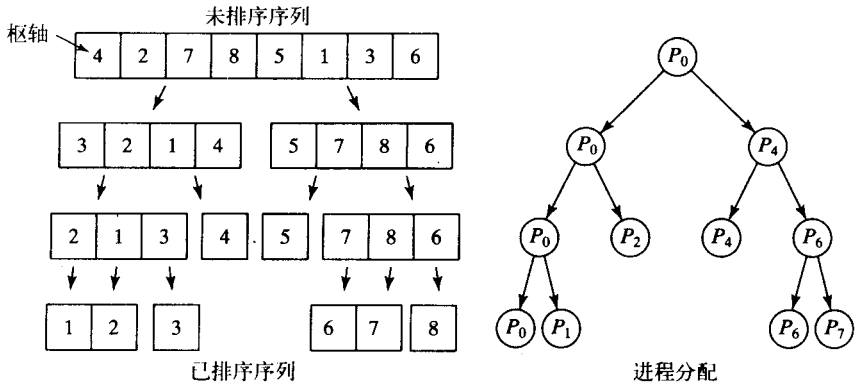


图10-9　使用树状进程分配的快速排序

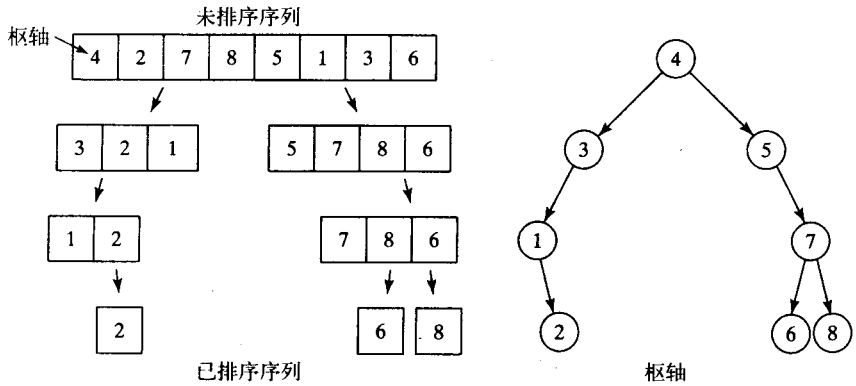


图10-10　快速排序中枢轴的隐藏

所有树结构的基本问题是最初的数据划分是由一个处理器完成的，这将严重限制速度。
314 假设枢轴选择是理想的并且序列被分为两个大小一样的子序列。

（1）计算　首先一个处理器对n个数字进行操作，然后两个处理器分别对$n/2$个数据进行操作，接着四个处理器各自对$n/4$个数据进行操作，依此类推：

$$t_{comp} = n + n/2 + n/4 + n/8 + \cdots \approx 2n$$

（2）通信 通信情况和归并排序类似:

$$t_{comm} = (t_{startup}+(n/2)\ t_{data}) + (t_{startup}+(n/4)\ t_{data}) + (t_{startup}+(n/8)\ t_{data}) + \cdots$$
$$\approx (\log p)t_{startup}+nt_{data}$$

以上分析都是针对理想情况的，但快速排序的树结构通常不是完全平衡的，这是和归并排序的树结构的主要区别。如果枢轴不能将序列分成相等的两个子序列，则快速排序的树结构的深度就不再固定为$\log n$，最差情况是每次选择枢轴时都碰巧选到子序列最大的数，这时时间复杂性将下降到$O(n^2)$。如果总是选择子序列的第一个数做枢轴，那么待排序序列的初始顺序将成为决定快速排序算法速度的关键因素。其他数字也可以被选做枢轴，在那种情况下枢轴的选择就非常重要了。

3. 工作池的实现

第7章中所描述的负载平衡技术，著名的工作池，可以被应用于诸如快速排序之类的分治排序算法。工作池可将欲进行划分的子序列作为任务加以保存。首先工作池保存一个未排序的序列，并将其交给第一个处理器。这个处理器将序列分为两部分，一部分交回工作池以给其他处理器，对另一部分进行类似操作，这种方法如图10-11所示。这种方法并不会减少空闲处理器的数量，但它能处理一些子序列比其他的子序列更长因此也需要更多工作量的情况。

在10.3.2节中我们将讨论在超立方体网络上的快速排序，这是一种能获得更好性能的排序方法。

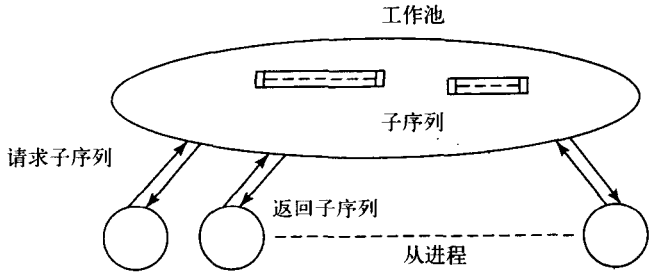


图10-11 快速排序的工作池实现

315

10.2.5 奇偶归并排序

奇偶归并排序算法是Batcher在1968年基于他的奇偶归并算法而提出的并行排序网络算法[Batcher,1968]。奇偶归并算法将两个有序序列归并为一个有序序列，通过递归调用该算法来构建更大的有序序列。对于给定的两个有序序列a_1，a_2，a_3，…，a_n和b_1，b_2，b_3，…，b_n（n为2的乘方），该算法完成如下动作:

1）每个序列的奇数下标索引的元素，即a_1，a_3，a_5，…，a_{n-1}和b_1，b_3，b_5，…，b_{n-1}被归并为一个有序序列c_1，c_2，c_3，…，c_n。

2）每个序列的偶数下标索引的元素，即a_2，a_4，a_6，…，a_n和b_2，b_4，b_6，…，b_n被归并为一个有序序列d_1，d_2，d_3，…，d_n。

3）最后的有序序列e_1，e_2，e_3，…，e_{2n}通过以下方法得到:

$$e_{2i} = \min\{c_{i+1}, d_i\}$$
$$e_{2i+1} = \max\{c_{i+1}, d_i\}$$

其中$1 \leqslant i \leqslant n-1$。从根本上说奇数和偶数的下标序列依次插入最后序列，必要时奇/偶元素对互换将较大的移向一端。第一个数$e_1 = c_1$（e_1是每个序列中第一个元素中最小的数，即a_1或b_1），

而最后的一个数$e_{2n}=d_n$（d_n为每个序列中最后一个元素中最大的数，即a_n和b_n）。Batcher提供了该算法的证明[Batcher，1968]。将两个各包含4个数的有序序列归并为一个包含8个数的有序序列的过程见图10-12。归并算法也可以用于不同长度的序列，但我们这里假设序列的长度都是2的乘方。

可以使用同样的算法来建立最初的每个有序子序列，如图10-13所示。该算法可以递归调用，使得时间复杂性为O ($\log^2 n$)（使用n个处理器）（习题10-15）。在该例中成对的处理器执行比较和交换操作。Batcher认为整个算法也可以使用执行比较和交换操作的硬件部件来完成。
316 最终的器件排列将作为习题留给读者。

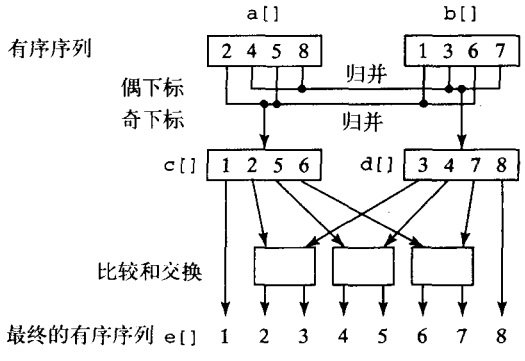


图10-12　两个有序序列的奇偶归并

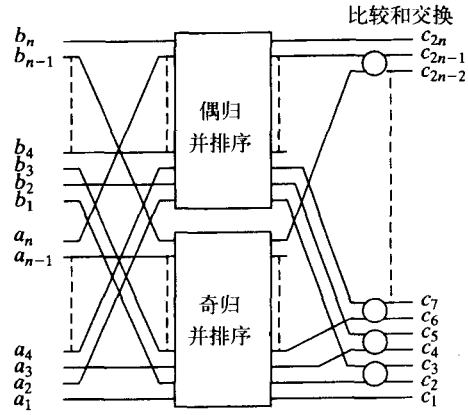


图10-13　奇偶归并排序

10.2.6　双调谐归并排序

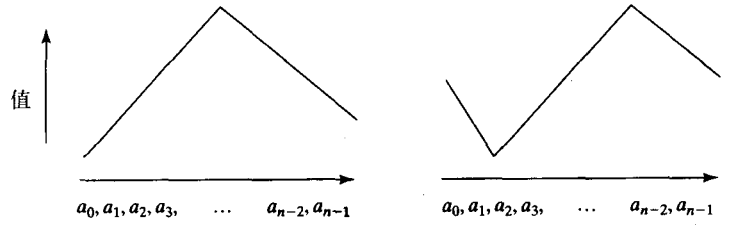
作为一种并行排序算法，*双调谐归并排序*（bitonic mergesort）也是Batcher于1968年提出的。

1. 双调谐序列

双调谐归并排序的基础是*双调谐序列*（bitonic sequence），一个在排序算法中应用的具有特殊属性的序列。一个单调增加序列指序列中的数从小到大递增排列，而一个双调谐序列中则包含两个序列：一个递增，一个递减。通常一个双调谐序列中的一系列数字a_0，a_1，a_2，a_3，…，a_{n-2}，a_{n-1}是这样排列的：先单调递增到一个最大值然后单调递减，即对某一个i的值（$0 \leqslant i < n$）有：

$$a_0 < a_1 < a_2 < a_3, \cdots, a_{i-1} < a_i > a_{i+1}, \cdots, a_{n-2} > a_{n-1}$$

如果通过循环移数（左移或右移）能得到前面的数，则该序列也为双调谐序列。图10-14对双调谐序列做了说明。注意一个双调谐序列可以通过合并两个有序序列（一个升序一个降序）来得到。



a）一个最大值　　b）一个最大值和一个最小值

图10-14　双调谐序列

双调谐序列的一个“特殊”性质是：如果我们对所有的i将a_i和$a_{i+n/2}$（序列中有n个数，$0 \leqslant i < n/2$）进行比较和交换操作，就能得到两个双调谐序列，其中一个双调谐序列中的所有数都小于另一个双调谐序列中的数。例如有以下双调谐序列： 317

3，5，8，9，7，4，2，1

对所有a_i和$a_{i+n/2}$进行比较和交换操作，所得结果则如图10-15所示。

比较和交换操作将每对数中较小的数移动到左边的序列，将较大的数移动到右边的序列。注意所有在左边序列中的数都比右边序列中的数小，并且这两个序列都是双调谐序列。显然对一个双调谐序列递归地执行比较和交换操作来得到子序列实际上就是对该序列进行排序，如图10-16所示。最后得到一组每个只含有一个数的双调谐序列，这也就是排完序的序列。

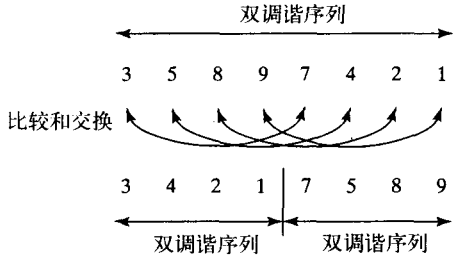


图10-15 由一个双调谐序列生成两个双调谐序列

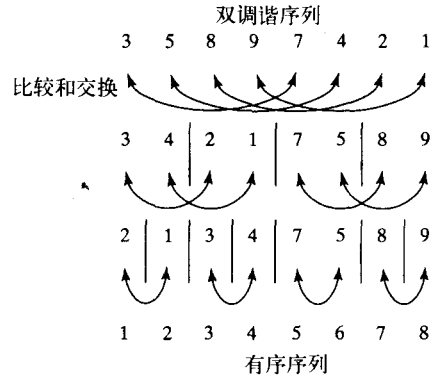


图10-16 对一个双调谐序列进行排序

2. 排序

为了对无序序列进行排序，从一对对邻近的数开始，序列被归并为较大的双调谐序列。通过比较和交换操作，一对对邻近的数形成升序或降序序列，它们两两形成双调谐序列，其大小是原来每个序列的两倍。重复上述过程，则所得到的双调谐序列将越来越长。在最后一步，一个双调谐序列将被排序成一个升序序列（排完序的序列），该算法如图10-17所示。比较和交换操作既可以形成升序序列也可以形成降序序列，且改变操作方向就可形成一个中间有最大值的双调谐序列。在图10-17中右边第一个双调谐序列在序列中心有一个最小数，而左边的双调谐序列在序列中心却有一个最大数。（如果两个双调谐序列在序列中心都有最大数，该算法也能正常工作。） 318

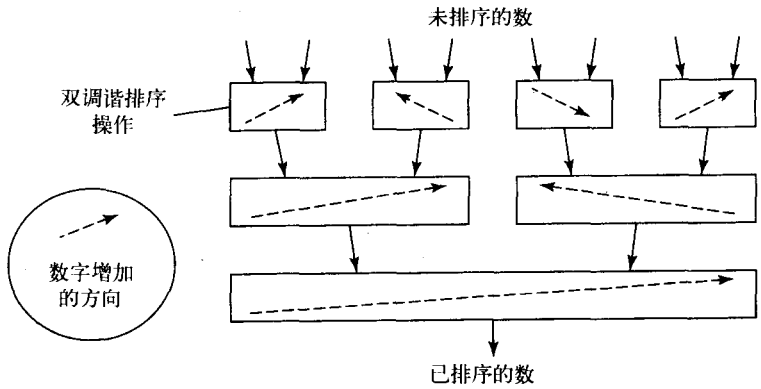


图10-17 双调谐归并排序

首先让我们用一个数字例子将算法扩展为基本的操作，图10-18显示了如何使用双调谐归并排序对8个数进行排序。基本的比较和交换操作由一个盒子代表，盒中的箭头指明哪个输出是该操作的数的较大值。6步（对8个数）分为3个阶段：

第1阶段（第1步）　将有2个数的数对转化为升序/降序序列并形成两个各有4个数的双调谐序列。

第2阶段（第2、3步）　将两个各有4个数的双调谐序列分为2个各有2个数的双调谐序列，并把较大的序列放在中间；对两个各有4个数的双调谐序列按升序/降序进行排序，并把它们归并为一个有8个数的双调谐序列。

第3阶段（第4、5、6步）　对有8个数的双调谐序列进行排序（如图10-17所示）。

如果$n = 2^k$，一般该算法就有k个阶段，每个阶段包含1、2、3、…、k步,则总的步数就等于:

$$\text{步数} = \sum_{i=1}^{k} i = \frac{k(k+1)}{2} = \frac{\log n\,(\log n+1)}{2} = O(\log^2 n)$$

使用n个处理器（每个处理器负责一个数）可达到$O(\log^2 n)$的时间复杂性是相当有吸引力的。像在其他排序算法中一样，可以通过数据分割的办法，用增加内部步数来减少处理器的数量。双调谐归并排序可以被映射在网格或超立方体上，[Quinn，1994]中对此有详细的描述，也可参见[Nassimi and Sahni，1979]。

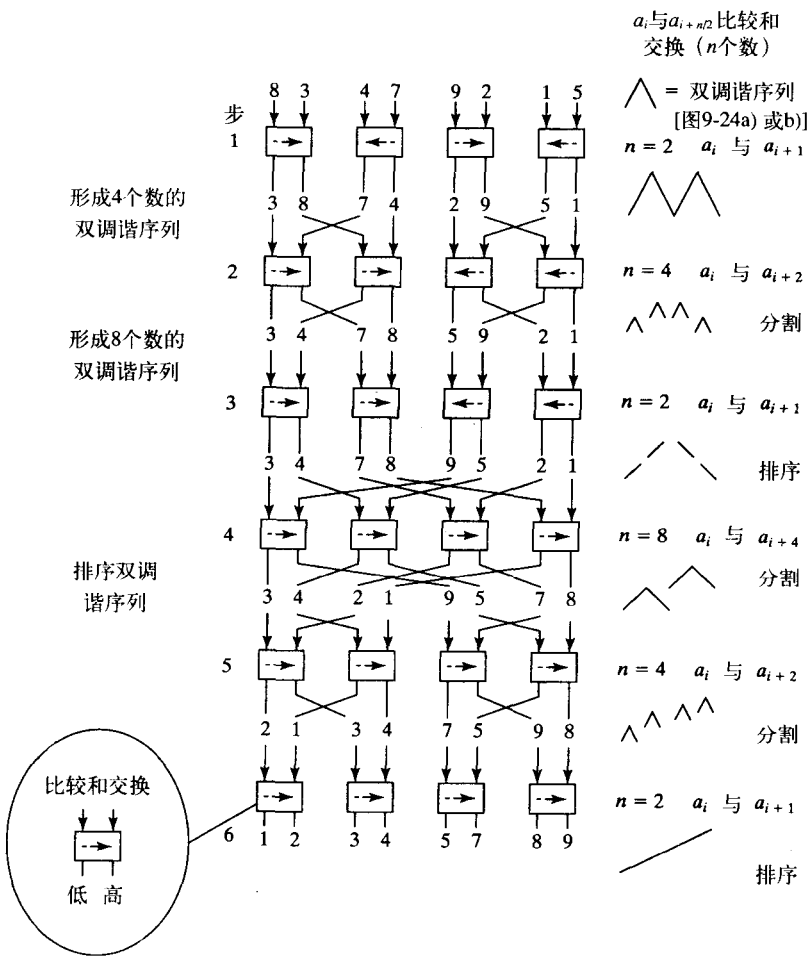


图10-18　8个数进行双调谐归并排序

使用硬件来实现这个算法（如由一个逻辑电路来接收数据并对它们进行排序，比较和交换操作由逻辑电路比较器完成）也是相当有吸引的，在本书中也经常是如此描述的。算法中 319
的每一步需要 $n/2$个2入/2出的比较器。图10-18中的“接线”本可以画得更直接，但图10-18中的“接线”能够清楚地显示数据被比较的位置以及随后的去向。

10.3 在专用网络上排序

算法可以利用并行计算机中底层的互联网络，近年来网格和超立方体这两种网络结构得到了众多的关注，因为许多并行计算机是用这两种互联网络建成的。我们将描述几个有代表 320
性的算法。一般而言，现在对这类算法已少有兴趣，因为当今系统的底层体系结构通常已对用户隐藏。（但是，MPI具有将算法映射到网格的特征，并且人们总是能使用网格或超立方体算法，即使底层的体系结构不尽相同）。

10.3.1 二维排序

如果数据被映射在网格上那就存在其他排序的方法。在网格上一组排序后的数字的排列是逐行即*扭曲的*（snakelike）。在扭曲排列中数字是按非降序排列的，如图10-19所示。数字可通过向储存最大数的结点移动来取出。一个映射在线形结构上的一维排序算法，如奇偶互换排序，可以应用于这些数字，从而得到一个在网格上复杂性为 O (n) 的排序算法，但这既没达到减少代价也没有充分利用网格结构的优势。任何在 $\sqrt{n}\times\sqrt{n}$ 网格上的排序算法的下限为 $2(\sqrt{n}-1)$ 步即 $O(\sqrt{n})$，因为在最坏情况下需要这些步来重新定位一个数字。注意，这里网格网络的直径是 $2(\sqrt{n}-1)$。

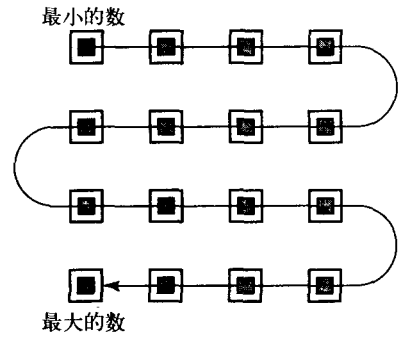


图10-19 扭曲排列的有序序列

[Scherson, Sen, and Shamir, 1986]描述了一种网格结构的精巧的排序算法，该算法叫*扭曲排序*（shearsort），在 $\sqrt{n}\times\sqrt{n}$ 网格上对n个数进行排序需要 $\sqrt{n}(\log n+1)$ 步，[Leighton, 1992]也详细描述了该算法。首先数被映射在网格上，然后执行一系列阶段（1，2，3，…）。在奇阶段（1，3，5，…）中执行下列动作:

每行数字以交替方向各自独立排序:

偶数行: 每列的最小的数放在最右端，最大的数放在最左端;

奇数行: 每列的最小的数放在最左端，最大的数放在最右端。

在偶阶段（2，4，6，…）中执行以下动作:

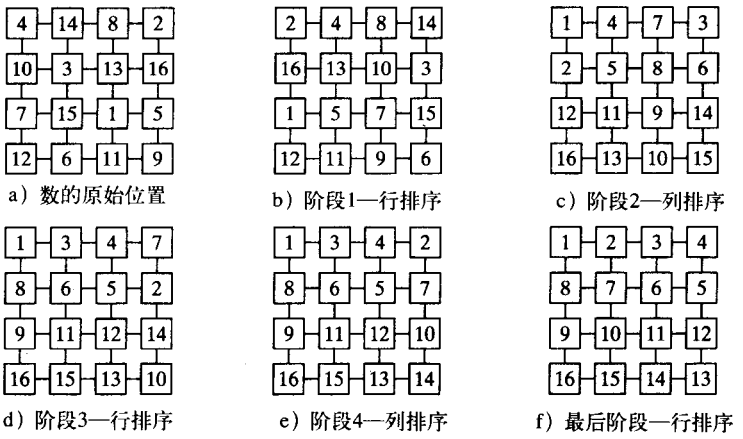
每列数据各自独立排序，最小的数放在最上端，最大的数放在最下端。 321

$\log n$ + 1个阶段后，排完序的数字就扭曲排列在网格上了。（注意行排序阶段方向的交替，这符合最终的扭曲布局。）图10-20显示了分布在4 × 4网格上的16个数是如何排序的。[Scherson, Sen, and Shamir, 1986]和[Leighton, 1992]给出了具体证明。行和列的排序可使用包括奇偶互换排序算法在内的任何排序算法。如果使用奇偶排序，有 $\sqrt{n}$ 个数的行或列排序都需要 $\sqrt{n}$ 步比较和交换，所以总共需要 $\sqrt{n}(\log n+1)$ 步。

除了扭曲排序还有其他特地为网格设置的排序算法，如[Gu and Gu, 1994]描述的算法。其他现存的排序算法也可以被修改以应用于网格结构。[Thompson and Kung, 1977]也提出一个应用于 $\sqrt{n}\times\sqrt{n}$ 网格的、下限为 $O(\sqrt{n})$的排序算法。

使用转置

人们可以通过转置每个阶段间的数据点的数组来将任何需交替进行行、列操作的网格算法（如扭曲算法）的操作限制在行操作中。转置操作将每列中的元素转置到一个行中，设数组中的元素为a_{ij}（$0 \leqslant i < n$，$0 \leqslant j < n$）。将元素从数组对主角线的下方移动到主对角线的上方以使$a_{ij}=a_{ji}$。转置操作被放置在行操作和列操作之间，如图10-21所示；在图10-21a中，操作在行中元素间进行；在图10-21b中发生一次转置操作，将列中的元素移到行中并将行中的元素移到列中。在图10-21c中原先在列中进行的操作现在在行中进行。



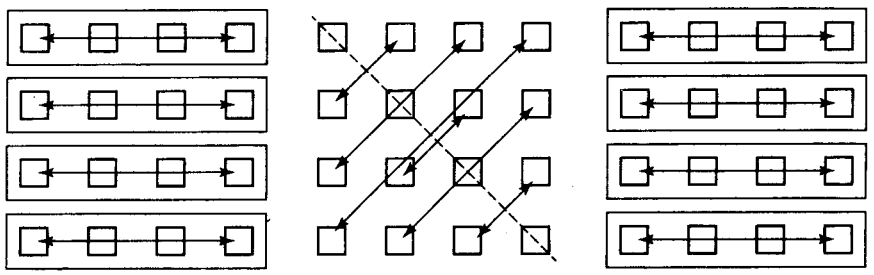
a）数的原始位置　b）阶段1—行排序　c）阶段2—列排序

d）阶段3—行排序　e）阶段4—列排序　f）最后阶段—行排序

图10-20　扭曲排序

图10-21表示使用一组处理器的并行实现，其中每个处理器负责对一行数据的排序。例如，考虑使用$\sqrt{n}$个处理器对一个$\sqrt{n} \times \sqrt{n}$数组进行排序，每个处理器负责一行，需要$\log n + 1$次迭代。在每次迭代中，每个处理器在$O(\sqrt{n}\log\sqrt{n})$步内来完成其行中数据的排序（根据该算法奇数行和偶数行选择不同的处理方式）。转置操作可以通过$\sqrt{n}(\sqrt{n}-1)$即$O(n)$（习题10-7）次通信来完成。注意数据在进程对之间被交换，每次交换需要两次通信。如果可用，一个单一的*全部到全部例程*（all-to-all routine）可以减少这种通信量（参见第4章）。然后每行再使用$O(\sqrt{n}\log\sqrt{n})$步进行排序。在网格上总的通信时间复杂性为$O(n)$，这是由转置操作决定的。这项技术可以应用在其他结构上，例如每行可以映射在一个线形结构的处理器上。

322



a）行中元素间的操作　b）转置操作　c）行（最初的列）中元素间的操作

图10-21　使用转置操作将排序操作限制在行上

10.3.2　在超立方体上进行快速排序

超立方体网络有一些结构特点，这些特点可以为实现有效的分治策略的排序算法，如快速排序，提供活动空间。

1. 整个序列在一个处理器上

假设最初有一个n个数的序列被放在d维超立方体的一个结点上。使用该处理器所选择的枢轴，按照快速排序算法，这个序列被分成两个部分，其中一个部分被送到最高维的邻近结点。然后这两个结点重复上述过程：将它们的序列使用局部选择的枢轴再一分为二，将一部分送到次最高维的邻近结点。这个过程持续d步以后，每个结点上都将有一部分数据。对一个三维超立方体来说，假设数最初都放在结点000上，则将有以下划分过程：

	结点		结点
第一步：	000	→	001（得到大于枢轴的数，如p_1）
第二步：	000	→	010（得到大于枢轴的数，如p_2）
	001	→	011（得到大于枢轴的数，如p_3）
第三步：	000	→	100（得到大于枢轴的数，如p_4）
	001	→	101（得到大于枢轴的数，如p_5）
	010	→	110（得到大于枢轴的数，如p_6）
	011	→	111（得到大于枢轴的数，如p_7）

323

这些操作详见图10-22。最后，所有结点并行地用顺序排序算法对部分数据进行排序。如果需要，排完序的部分将被送回一个处理器以允许该处理器拼接已排序的序列来建立最终的排完序的序列。

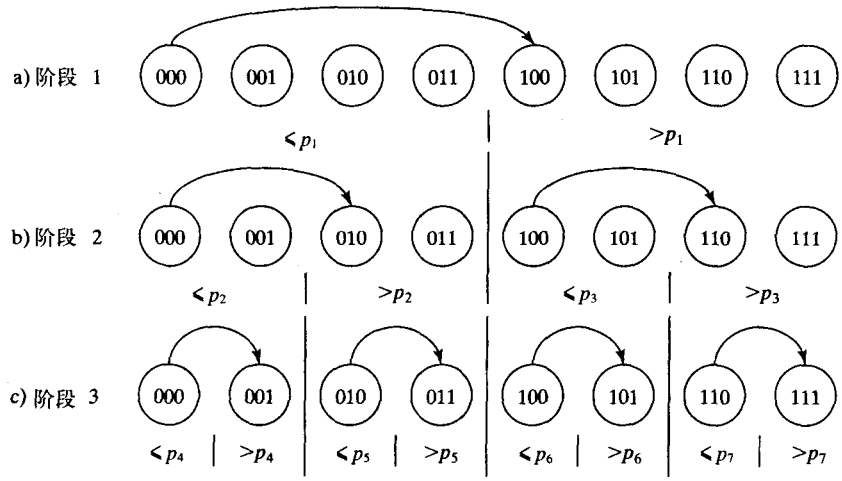


图10-22 数最初放在结点000时的超立方体快速排序算法

2. 最初数据散布在所有处理器上

假设待排序数最初均匀地、不按任何特殊顺序地分布在结点上。一个2^d个结点的超立方体（d维超立方体）由两个2^{d-1}个结点的超立方体组成，它们通过每个立方体中第d维结点之间的链路互联。按同样的办法这些小超立方体可以分为更小的超立方体，直到每个超立方体中只有一个结点。这个特性可以被用于超立方体的实现快速排序算法的直接扩展中，其步骤如下：首先将超立方体看成两个子立方体，位于高端子立方体中的处理器的地址首位为1，位于低端子立方体中的处理器的地址首位为0。成对的地址分别为0xxx和1xxx的处理器称为“伙伴”（partner），其中x是相同的。第一步为：

1）一个处理器（如P_0）选择（或计算）一个合适的枢轴，并通过广播把它送到立方体中所有其他处理器。

2）低端子立方体中的处理器将大于该枢轴的数传送给它们位于高端子立方体中的伙伴处理器。高端子立方体中的处理器将小于或等于该枢轴的数传送给它们位于低端子立方体中的伙伴处理器。

3）每个处理器将收到的序列和本身的序列拼接。

给定一个d维超立方体，经过这些步骤以后，低端（$d-1$）维子立方体中的数都小于或等于枢
324 轴，而高端（$d-1$）维立方体中的数都大于枢轴。

两个（$d-1$）维子立方体重复递归调用第2、3步，其中每个子立方体中的一个进程为该子立方体选择枢轴并通过广播传送到子立方体中的其他进程，递归调用log$d$次后终止。假设超立方体有三维，那么当递归调用终止后，处理器000中的数都小于处理器001中的数，而001中的数又都小于010中的数，依此类推，如图10-23所示。三维超立方体中的通信模式在图10-24中做了说明。

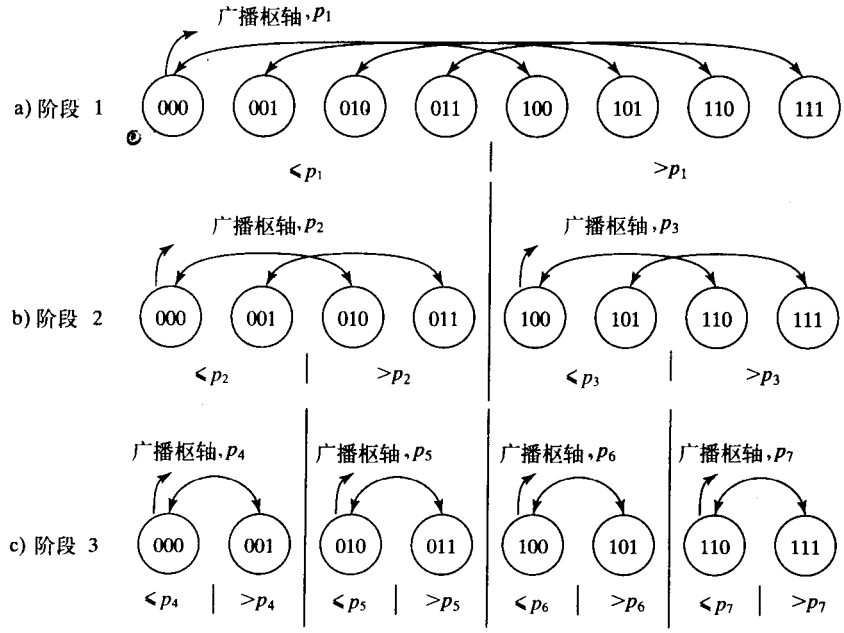


图10-23　数字散布在结点中时的超立方体快速排序算法

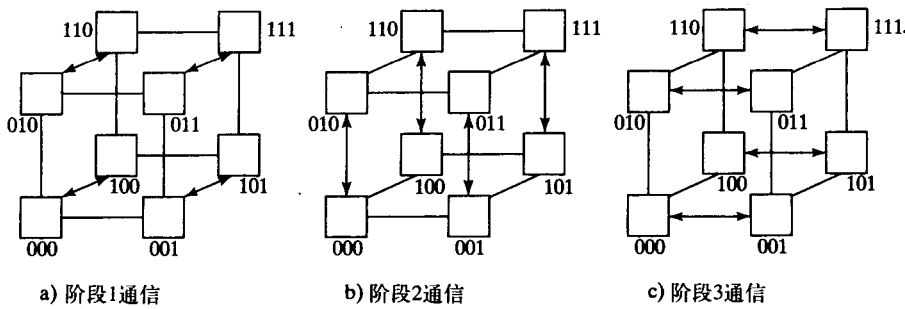


图10-24　超立方体快速排序中的通信

为了完成排序，每个处理器中的数需要顺序进行排序。最后按数字顺序访问处理器就可从处理器中检索数。由于按数字顺序相邻的处理器之间不一定有直接的链路，所以更方便的方法是使用图10-25所示的结构，它将排完序的数存放在一组按增序葛莱码（Gray Code）顺

序排列的处理器中。在更大的超立方体中进行排序的探讨将作为习题（习题10-13）。

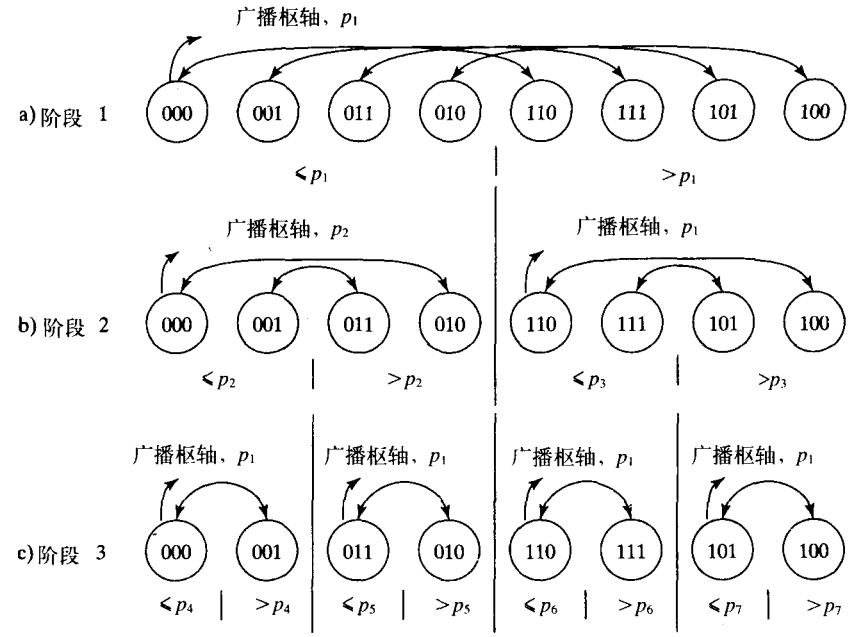


图10-25 用葛莱码排序的超立方体快速排序算法

3. 枢轴的选择

在快速排序的所有正式描述中，枢轴的选择都是很重要的，对于多处理机的实现就更是如此。一个拙劣的枢轴选择方法可能导致将大多数数分配到超立方体中的少数处理器中，而使其他处理器空闲。在第一次分裂中，这是最有害的。在顺序快速排序算法中，常常选择序列中的第一个数作枢轴，这样只需一步或O(1)的时间复杂性就可得到枢轴。改进枢轴选择的一种方法是：从序列中抽取数作样本，计算其平均值，并选择中值作为枢轴。如果选择中值作为枢轴，那么这些取样数必须至少进行一次半排序以找到中值。我们可选择一个遇到中值 325
就会终止的简易冒泡算法来进行此项工作。

4. 超快速排序

为了保持每个处理器中数据的有序，另一个版本的超立方体快速排序算法总是在每个阶段都对数进行排序。这样做不仅使每步的枢轴选择变得简单，而且消除了最后的排序操作。这种在超立方体上的快速排序称为*超快速排序*（hyperquicksort）[Wagar，1987]，并按照以下步骤在每个阶段保持数有序。 326

10.4 其他排序算法

我们在本章开始时给出了基于比较的顺序排序算法时间复杂性的下限$O(n\log n)$（确切地应是$\Omega(n\log n)$，但我们一直使用大O记号）。显然，基于比较的并行排序算法的时间复杂性，在使用p个处理器时应是$O((\log n)/p)$，而在使用n个处理器时应是$O(\log n)$。事实上，有些排序算法能获得好于$O(n\log n)$的顺序时间的复杂性，因此它们是并行化的很有吸引力的候选者，但它们常假设被排序的数具有特殊性。不过首先让我们考虑一种秩排序算法，它虽然达不到$O(n\log n)$的顺序时间，但很容易进行并行化，在使用n个处理器时它可达到$O(n)$的并行时间复杂性，而在使用n^2个处理器时可达到$O(\log n)$，并启发我们对线性顺序时间算法进行并行化以

达到 O(log$n$) 的并行时间，对机群来讲这是很具吸引力的算法。

10.4.1 秩排序

在秩排序（也称为*枚举排序*）中，人们统计小于每个被选数的数的个数。这种计数提供了所选择的数在数字序列中的位置，即在序列中的“秩”。假设有n个数存放在一个数组a[0]…a[n-1]中。首先将a[0]读出并与其他数a[1]，…，a[n-1]进行比较，记录小于a[0]的数的个数，设为x。则x就是a[0]在排序后的数组中的位置，a[0]被复制到排序后的数组b[0]…b[n-1]中的b[x]。然后，读出a[1]并与其他数a[0]，a[2]，…，a[n-1]比较，依此类推。如果顺序执行的话将一个数与其他n-1个数进行比较至少需要n-1步。对全部n个数执行此操作需要n（n-1）步，因此总的顺序排序的时间复杂性为O(n^2)（这不是一个好的顺序排序算法！）。实际的顺序代码如下：

```
for (i = 0; i < n; i++) {      /* for each number */
   x = 0;
   for (j = 0; j < n; j++)     /* count number of numbers less than it */
      if (a[i] > a[j]) x++;
   b[x] = a[i];                /* copy number into correct place */
}
```

在两个for循环中每个迭代都有n步，但如果该数字序列中存在重复的数那么上述代码将无法正常工作，因为重复的数将放在已排序表中的相同位置，不过该代码很容易修改成能处理存在重复的数的情况，如下所示：

```
for (i = 0; i < n; i++) {      /* for each number */
   x = 0;
   for (j = 0; j < n; j++)     /* count number of numbers less than it */
      if (a[i] > a[j] || (a[i] == a[j] && j < i)) x++;
   b[x] = a[i];                /* copy number into correct place */
}
```

327

该代码将重复的数按它们在原来序列中的相同次序存放。将重复的数以它们在原始序列中相同次序的排序算法称为*稳定的排序算法*（stable sorting algorithms）。以下为简单起见我们将略去处理重复数的修改代码。还应指出的是，如果数全为整数，则秩排序可以编码成能达到O（n）顺序时间复杂性的形式。但该代码需要一个附加的数组以存放数的每个可能值。我们将在名为*计算排序*的10.4.2节中考虑它的实现。

1. 使用n个处理器

假设我们有n个处理器。一个处理器与一个数相关联，那么找到一个数在排序完成后的数组中的位置需要O(n)步。若所有n个处理器并行进行操作则并行时间的复杂性也是O(n)。使用forall记号的相关代码如下：

```
forall (i = 0; i < n; i++) {      /* for each number in parallel*/
   x = 0;
   for (j = 0; j < n; j++)        /* count number of nos less than it */
      if (a[i] > a[j]) x++;
   b[x] = a[i];                   /* copy number into correct place */
}
```

能获得线性并行时间复杂性O(n)固然很好，但若有更多处理器时，就可获得更好的时间复杂性。

2. 使用n^2个处理器

如果使用多处理机来执行一个数与多个数的比较会取得更好的效果。例如可使用图10-26所示的结构。这里我们使用$n-1$个处理器来找一个数的秩，那么要找到n个数的秩就需要$n(n-1)$个即将近n^2个处理器。每个数各需要一个计数器，在图10-26中，对计数器实现增量是顺序执行的且最多需要n步（包括初始化计数器的一步）。因此总的执行步数为$1+n$（1为处理器并行操作时间）。为减少增量计数器的步数，可采用如图10-27所示的树形结构。这样就得到一个使用n^2个处理器的O(log$n$)排序算法。这种方法的实际处理器效率是很低的（具体计算也留作习题）。在并行计算的理论模型中，完成增量操作和将它们的结果组合起来只需一步。在这种模型中可将秩排序的时间复杂性减为O(1)。O(1)当然是任何求解问题下限，但我们不在这里探讨理论模型。 328

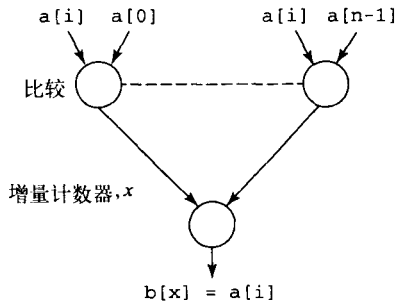


图10-26 用并行方法找秩

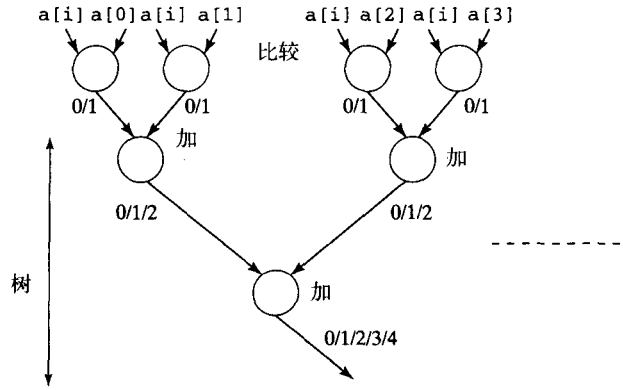


图10-27 秩计算的并行化

已经提及用n个处理器或用n^2个处理器可以以O(n)或O(log$n$)的时间复杂性分别完成秩排序。但在实际的应用中，由于价格过高不会使用n^2个处理器。该算法需要共享对数字序列的访问，这使该算法最适合于共享存储器系统。该算法也可用消息传递实现，此时，一个主进程响应来自从进程对数字的请求，如图10-28所示。

当然，我们可以通过将数按组（每组m个数）划分来减少处理器数。这时就仅需n/m个处理器来完成数组的排序（不对比较操作并行化），而每个处理器所执行的操作数乘上因子m。这种数据分割方法可应用在许多排序算法中，因为通常数的个数n远大于可用的处理器数。

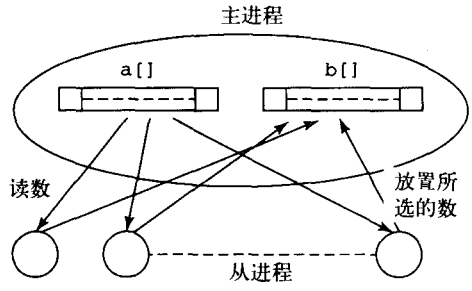


图10-28 使用主从进程实现秩排序 329

10.4.2 计数排序

如果要排序的数是整数，在10.4.1小节中已介绍过一种编写秩排序算法的方法，可将顺序时间复杂性从O(n^2)降为O(n)。[Coren, Leiserson, and Rivest, 1990]将此法称为计数排序(counting sort)。计数排序是自然的稳定排序算法（即它将相同的数按它们在原始序列中所出现的相同次序来存放它们）。如同10.4.1节中的秩排序代码，最初未排序的数存放在数组a[]，而最后已排序好的数将存放在数组b[]。该算法使用了一个附加数组c[]，该数组中的每个元素对应于数的一个可能值。假设整数的取值范围为1到m，则数组就含有元素c[1]到c[m]。现在让我们分阶段来叙述该算法。

首先，用c[ ]来保存序列的直方图，即每个数的个数。这可用以下代码在O(m)时间内完成计算:

```
for (i = 1; i <= m; i++)
  c[i] = 0;
for (i = 1; i <= m; i++)
  c[a[i]]++;
```

算法的下一步，通过在数组c[ ]上完成前缀求和操作，就可找到小于每个数的数的个数。在前缀求和的计算中，对给定的数序列x_0，…，x_{n-1}，将计算所有的部分和（即x_0；x_0+x_1；$x_0+x_1+x_2$；$x_0+x_1+x_2+x_3$；…），关于这一点早在6.2.1节中提及。但现在的前缀求和计算是用最初保存在c[ ]中的直方图在O(m)时间内完成的，如下面所示的那样:

```
for (i = 2; i <= m; i++)
  c[i] = c[i] + c[i-1];
```

算法的最后一步，用O(n)时间将这些数按已排序好的次序放到c[ ]中，如下面所示:

```
for (i = 1; i <= n; i++) {
  b[c[a[i]]] = a[i];
  c[a[i]]--;                    // done to ensure stable sorting
}
```

整个代码的顺序时间的复杂性为O($m+n$)。在某些应用中如果m与n线性相关（见下面的例子），则代码的顺序时间复杂性为O(n)。图10-29示出了在一个有8个数的序列上进行计数排序的情况。图中突出序列中第一个数的移动。其他数的移动按类似方式进行。

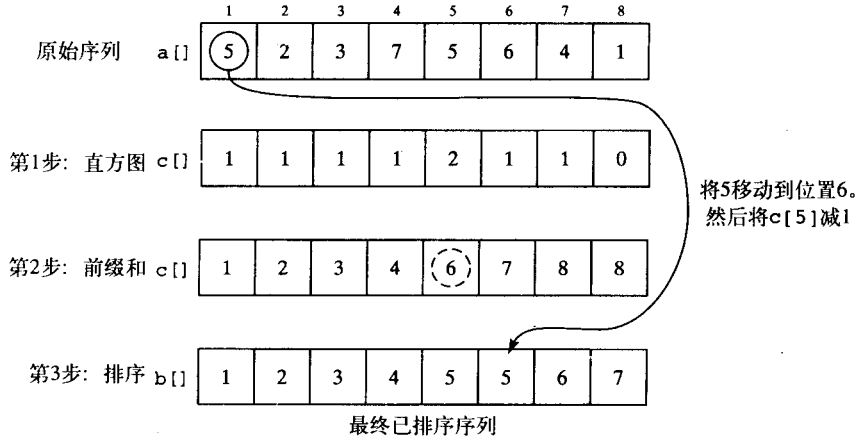


图10-29　计数排序

计数排序并行化可使用前缀求和计算的并行化版本，用$n-1$个处理器时需O(log$n$)时间（见6.2.2节）。最后的排序阶段，用p个处理器时需时O(n/p)，而用n个处理器简单地使循环体由不同的处理器来完成，则只需O(1)时间。

当只有很少的不同数且它们全是整数时，计数排序算法是一个非常高效的算法。在下一小节的基数排序算法中，将用到计数排序。

330

10.4.3　基数排序

基数排序（radix sort）假设欲排序的数用一位数字表示，如2进制数和10进制数。数字表

示值，而每个数字的位置指明相应的权重（即用什么值来乘，在10进制数中用基10，2进制数中用基2）。位置是从最低有效位向最高有效位排列。基数排序从最低有效位开始（不是直观上的从最高有效位开始），并按它们的最低有效位排序。此后，再按次最低有效位排序，依此类推，直至最高有效位，此时序列已排好序。要使此算法能正常工作，必须对具有相同数字的数保持数的次序，即必须使用一个稳定的排序算法。

基数排序可以对数的单个数字位进行，也可对数字位组进行。图10-30中示出了对单个10进制数字位进行基数排序的情况。在图10-31中示出的则是对单个2进制数字位进行基数排序 331
的情况。在这种情况下，如果每个数有b位就需要b个阶段。应注意，已保持具有相同位值数（在该例中是0或1）的次序。通常基数排序在每一阶段对2进位组而不是只对一位进行排序。一般而言，在基数排序中，对b位2进制数可通过每次对由r个数字位所组成的组进行排序来完成，此时将需$\lceil b/r \rceil$个阶段。基数排序的时间复杂性将依赖于每一阶段的排序算法；且必须是一个稳定的排序算法。当给定数的取值范围较小时，采用计数排序较为合适，但它不是唯一的选择，因为它不在适当的位置排序（即当数被排序后，它需要附加的存放已被排序数的存储空间）。

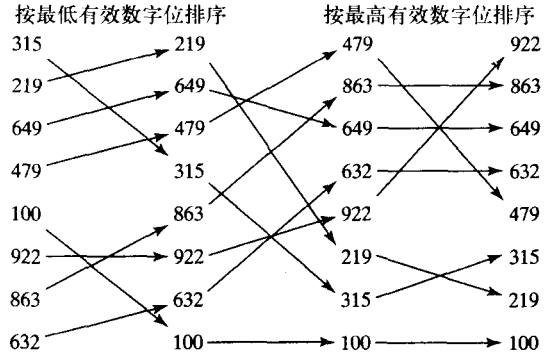


图10-30 用10进制数进行基数排序

假设在基数排序中使用计数排序。如在10.4.2节所提及的那样，在n个整数上（每个整数的取值范围为1到m）计数排序的顺序时间复杂性为O $(m + n)$。给定r位的数，取值范围从1到2^r-1。共有$\lceil b/r \rceil$阶段，其中所有数都是b位。因此，顺序时间的复杂性为O $(b/r(n + 2^r))$。如果b和r是常数[⊖]，则顺序时间的复杂性为O(n)，即仍是线性时间复杂性。

基数排序可以通过在每个阶段对位或位组使用并行排序算法来并行化。我们已提及用前缀求和计算来并行化计数排序，用$n-1$个处理器且b和r为常数时需时O(log$n$)。[Bader and JáJá, 1999]在他们的SMP机群程序设计的研究中，采用了带计数排序的基数排序。

以图10-31中的对2进制数字（即$r = 1$）的排序例子来讲，较明智的并行化基数排序的方法是，在每一阶段用前缀求和计算来定位每个数。当对一位列中的（2进）位进行前缀求和计算时，它使1的个数等于每个数字位的位置，因为数字位只能为0或1，且前缀计算将只是简单地加1的个数。第2个前缀计算将使0的个数等于每个数字位的位置，这是通过逆向的对数字位进行前缀计算来完成的（有时称为*减少前缀求和计算*，diminished prefix-sum calculation）。在被考察的数的数字位为0时，减少前缀计算将为该数提供新的位置。若数字位为1，则通常的

⊖ 在n和b之间存在某种关系。假设无重复的数，则有$n \leqslant 2^b$，即$b \geqslant \log_2 n$。然而，计算机运算时，通常用固定位数来表示数而不论有多少个数。

前缀求和计算加上最大的减少前缀求和计算的结果将为该数确定最后的位置。例如，如果所
332 考察的数有一个1，以及四个为0的数加上前面两个为1的数，减少前缀计算给定4，加上常规前缀求和计算给定3，就使该数定位在7的位置（从位置1开始计数）。显然，当用$n-1$个处理器时也将导致O(log$n$)时间，但当$r=1$时，它需要b个阶段。

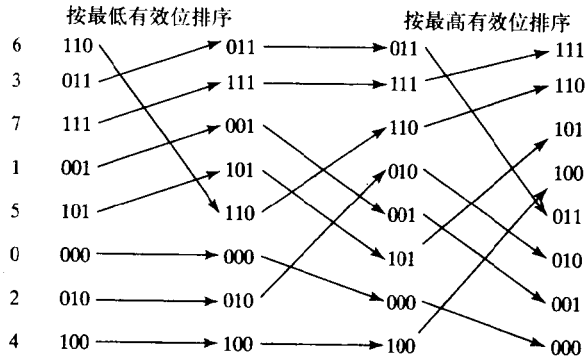


图10-31　用2进制数进行基数排序

10.4.4　采样排序

如同其他许多基本的排序想法一样，采样排序是一个老概念[Frazer and McKellar，1970]。在叙述快速排序和桶排序时，已对采样排序进行了讨论。在快速排序环境下，采样排序从n个数的序列中采样s个数的一个样本。将该样本的平均值作为第1个枢轴把序列分成两部分，这是快速排序算法所需的第一步，而不是通常序列中的第一个数。为了加速，后续步所需的所有枢轴可同时求得。例如，上下两个四分点可作为前两个后续子序列的枢轴使用。这一方法存在这样一个问题，即差的枢轴选择在快速排序中将造成不平均的序列划分。

采样排序也可用于桶排序，如在4.2节中所叙述的那样。桶排序的基本问题与快速排序中的一样：不平均的划分序列。在桶排序中，如果数不是相等分布的，则较多的数会落入某些桶中。在最坏的情况下，所有的数会落入一个桶中，从而使原本可能的线性顺序时间复杂性退化成用比较排序算法对n个数进行排序（即O($n$log$n$)）。其根本问题是在于，每个桶的数的范围是固定的且是通过将整个数的范围划分成相等区域加以选择的。

采样排序的目的在于在划分数的范围时，使得每个桶拥有大致相同的数的个数。为做到这一点采用了如下的采样方案，即从n个数的序列中，挑出一些数作为分裂数，再由这些分裂数来定义每个桶的数的范围。如有m个桶，则就需$m-1$个分裂数。这些分裂数可用下述方法找到。首先，将欲排序的数分成n/m个组。对每个组进行排序，并从每个组选择s个具有相等空间间隔的数作为一次采样。这将产生总数为ms个的采样，然后对它们排序，而将$m-1$个具有相等间隔的数选为分裂数。图10-32中示出了选择这些分裂数的方法。在分裂数对每个桶的范围设定后，算法将以桶排序的方式继续进行排序（对桶进行排序，并将结果拼接）。

该方法可用与桶排序几乎相同的办法进行并行化，即为每个处理器分配一个有n/m个数的组，以及一个桶（$p=m$）。将mp个数送到一个处理器进行排序，当然也存在其他方法。s的值越大，则在最后的分裂数中所使用的数的个数就越多。如果$s=m-1$，则在任何一个桶中的数的个数将小于$2n/m$。在所叙述的采样排序中，还有一些变形，在这些方法中利用规整的集合操作进行高过度采样，以在机群上获得上好的性能（参见[Helman,Bader and JáJá，1998]）。

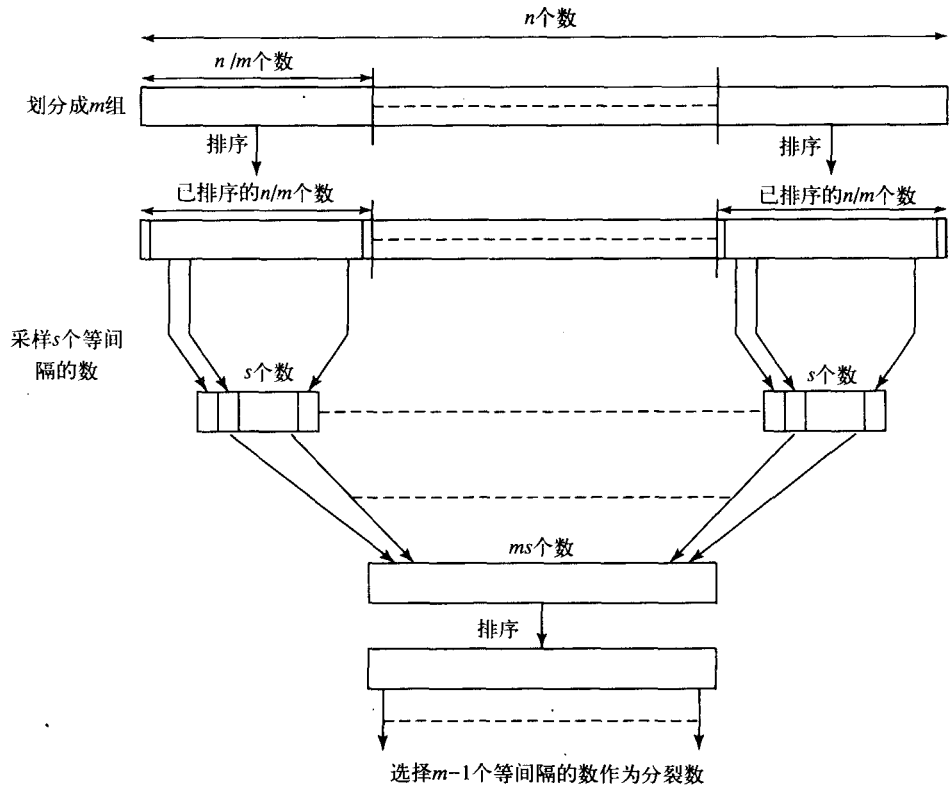


图10-32 在桶排序的采样排序版本中选择分裂数

333
~
334

10.4.5 在机群上实现排序算法

想在机群上实现高效的排序算法，要求使用在MPI那样的消息传递软件中所提供的广播以及其他的集合操作，如集中、分散和归约，而不是那种需要点对点通信的非一致通信模式，因为集合操作能高效地实现。机群的分布式存储器不适合需要访问非常分散存储的数据的那类算法。只需进行本地操作的算法是为更好，纵然在最坏情况下，所有排序算法在最后不得不将数从序列的一端移动到另一端。

处理器总是具有高速缓存，因此好的算法应对存放在高速缓存的数字块进行操作。其结果是，我们需要知晓高速缓存的大小和组成，而且这必须成为算法参数的一部分。最后，随着SMP处理器机群（SMP机群）的出现，算法需要考虑在每个SMP系统中的处理器组内可能以共享存储器方式运行，但其中的共享存储器只存在于每个SMP系统中；而机群中各个SMP系统间的通信则需以消息传递的方式运行。考虑到这一点就需把每个SMP系统中的处理器数以及每个SMP系统中的存储器大小也作为算法的参数。在[Bader and JáJá, 1999]中可找到有关在SMP系统上使用的算法的更多信息。

10.5 小结

本章讨论了一些使用多处理机的排序方法。我们从基于比较且顺序实现的排序算法开始，说明其下限是O($n$log$n$)以及用n个处理器并行实现时是O(log$n$)。然后我们讨论了比较和交换操作和排序算法：

- 冒泡排序
- 奇偶互换排序
- 归并排序
- 快速排序，包括超立方体上的快速排序
- 奇偶归并排序
- 双调谐归并排序

接着我们简要地观察了利用特定互联网络的一些有代表性的排序算法：

- 扭曲排序（在网格上）
- 在超立方体上的快速排序

最后，我们探讨了不使用比较和交换操作的排序算法：

- 秩排序
- 只适用于整数的计数排序

以及并行化后能获得较高并行性能的算法：

- 基数排序
- 采样排序

推荐读物

有非常多的排序算法及其变形在本章中没有涉及。我们选择的是最常用、最具技术代表性或具有并行化潜力的那些排序算法。[Richards，1986]提供了一份有373篇关于并行排序论文的清单（到20世纪80年代早期为止）。[Akl，1985]也撰写了一本关于并行排序算法的专著。此后涉及到排序算法的论文包括[Blackston and Ranade，1993]、[Bolorforoush et al.，1992]。在已出版的有关并行排序的概述文章中，著名的有[Bitton et al.，1984]和[Lakshmivarahan，Dhall，and Miller，1984]，后者完全关注于排序网络。

在此第2版中，我们对内容作了重新排序，并增加了两个著名的排序算法，基数排序和采样排序。由于这两个算法是在机群，特别是SMP机群，上进行排序的基础，故近来备受关注，尽管由于复杂性的原因本书略去了机群实现的全部细节。更多的细节可在[Helman, Bader, and JáJá，1998]中找到。[Shi and Schaeffer, 1992]描述了采样排序。

参考文献

AJTAI, M., J. KOMLÓS, AND E. SZEMERÉDI (1983), "An O($n$log$n$) Sorting Network," *Proc. 15th Annual ACM Symp. Theory of Computing*, Boston, MA, pp. 1–9.

AKL, S. (1985), *Parallel Sorting Algorithms*, Academic Press, New York.

335 BADER, D. A., AND J. JÁJÁ (1999), "SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs), *J. Par. Dist Computing*, Vol. 58, pp. 92–108.

BATCHER, K. E. (1968), "Sorting Networks and Their Applications," *Proc. AFIPS Spring Joint Computer Conference*, Vol. 32, AFIPS Press, Reston, VA, pp. 307–314.

BITTON, D., D. J. DEWITT, D. K. HSIAO, AND J. MENON (1984), "A Taxonomy of Parallel Sorting," *Computing Surveys*, Vol. 16, No. 3 (September), pp. 287–318.

BLACKSTON, D. T., AND A. RANADE (1993), "SnakeSort: A Family of Simple Optimal Randomized Sorting Algorithms," *Proc. 1993 Int. Conf. Par. Proc.*, Vol. 3, pp. 201–204.

BOLORFOROUSH, M., N. S. COLEMAN, D. J. QUAMMEN, AND P. Y. WANG (1992), "A Parallel Randomized Sorting Algorithm," *Proc. 1992 Int. Conf. Par. Proc.*, Vol. 3, pp. 293–296.

CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST (1990), *Introduction to Algorithms*, MIT Press, Cambridge, MA.

FRAZER, W. D., AND A. C. MCKELLAR (1970), "Samplesort: A sampling approach to minimal storage tree sorting," *J. ACM*, Vol. 17, No. 3 (July), pp. 496–567.

FOX, G., M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER (1988), *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, Englewood Cliffs, NJ.

GU, Q. P., AND J. GU (1994), "Algorithms and Average Time Bounds of Sorting on a Mesh-Connected Computer," *IEEE Trans. Par. Distrib. Syst.*, Vol. 5, No. 3, pp. 308–314.

HELMAN, D. R., D. A. BADER, AND J. JÁJÁ (1998), "A Randomized Parallel Sorting Algorithm with an Experimental Study, *Journal of Parallel and Distributed Computing*, Vol. 52, No. 1, pp. 1–23

HIRSCHBERG, D. S. (1978), "Fast Parallel Sorting Algorithms," *Comm. ACM*, Vol. 21, No. 8, pp. 538–544.

HOARE, C. A. R. (1962), "Quicksort," *Computer Journal*, Vol. 5, No. 1, pp. 10–15.

JÁJÁ, J. (1992), *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA.

LAKSHMIVARAHAN, S., S. K. DHALL, AND L. L. MILLER (1984), "Parallel Sorting Algorithms," *Advances in Computers*, Vol. 23, pp. 295–354.

LEIGHTON, F. T. (1984), "Tight Bounds on the Complexity of Parallel Sorting," *Proc. 16th Annual ACM Symposium on Theory of Computing*, ACM, New York, pp. 71–80.

LEIGHTON, F. T. (1992), *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA.

NASSIMI, D., AND S. SAHNI (1979), "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Trans. Comp.*, Vol. C-28, No. 2, pp. 2–7.

QUINN, M. J. (1994), *Parallel Computing Theory and Practice*, McGraw-Hill, NY.

RICHARDS, D. (1986), "Parallel Sorting — A Bibliography," *SIGACT News* (Summer), pp. 28–48.

SCHERSON, I. D., S. SEN, AND A. SHAMIR (1986), "Shear-Sort: A True Two-Dimensional Sorting Technique for VLSI networks," *Proc. 1986 Int. Conf. Par. Proc.*, pp. 903–908.

SHI, H, AND J. SCHAEFFER (1992), "Parallel Sorting by Regular Sampling," *Journal of Parallel and Distributed Computing*, Vol. 14, pp. 361–372.

THOMPSON, C. D., AND H. T. KUNG (1977), "Sorting on a Mesh-Connected Parallel Computer," *Comm. ACM*, Vol. 20 (April), pp. 263–271.

WAGAR, B. (1987), "Hyperquicksort: A Fast Sorting Algorithm for Hypercubes," *Proc. 2nd Conf. Hypercube Multiprocessors*, pp. 292–299.

WAINWRIGHT, R. L. (1985), "A Class of Sorting Algorithms Based on Quicksort," *Comm. ACM*, Vol. 28, No. 4, pp. 396–402. Also see "Technical Correspondence," *Comm. ACM*, Vol. 29, No. 4, pp. 331–335.

习题

科学/数值习题

10-1 重写10.2.1节中的比较和交换操作代码，以在不需要交换时取消消息传递。

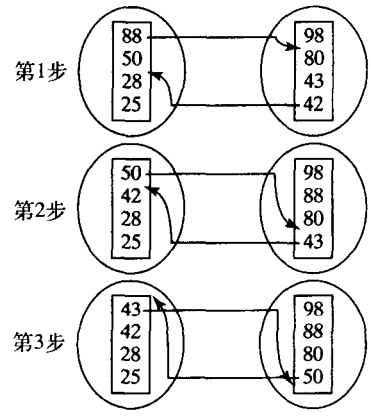
10-2 实现10.2.1节中提到的两个比较和交换操作的方法（即版本1和版本2），每个数据块有4个数，并测量两种方法的性能。设计一个例子以证明当异构系统中计算机之间存在不同计算精度时，可能得到错误的结果。

10-3 根据10.4.1节所描述的基于秩排序确定O (log$n$)算法的处理器的效率。

10-4 [Fox et al., 1988]提出了一个进行比较和交换的方法，在这个方法中各个数在两个处理器（记为P_0、P_1）之间交换，假设每个处理器中每一个组有4个数。处理器P_0将自己组

中最大的数发送到P_1，而处理器P_1将自己组中最小的数发送到处理器P_0。然后每个处理器各自将收到的数插入到自己的组中，因此两者依然还有n/p个数，接着以新的最大数和最小数重复上述过程。若使用带有指向序列顶或序列底的指针的队列，该算法的实现是最方便的，如图10-33所示。在图10-33中当插入操作都发生在序列顶或序列底时，算法终止。算法初始时若左边序列的所有数都大于其他组中的所有数时，通信步数最大（每组有x个数时最大是x步）。编写一个程序实现Fox的方法，并和本书中描述的其他算法进行比较来评估这一方法。

336
~
337



当插入操作在序列顶/底时终止

图10-33　习题10-4的比较和交换算法

10-5　以下代码为一SPMD程序，试图完成10.2.2节中提到的奇偶互换排序，请判断这段代码是否正确，若不正确请加以改正。

进程P_i

```
evenprocess = (i % 2 == 0);
evenphase = 1;
for (step = 0; step < n; step++, evenphase = !evenphase){
   if ((evenphase && evenprocess) || (!evenphase) && !evenprocess)) {
     send(&a, P[i+1]);
     recv(&x, P[i+1]);
     if (x < a) a = x;              /* keep smaller number */
   } else {
     send(&a, P[i-1]);
     recv(&x, P[i-1]);
     if (x > a) a = x;              /* keep larger number */
   }
}
```

10-6　10.2.2节中描述的奇偶互换排序有一个前提条件，即待排序数的个数和处理器数均为偶数。如果待排序数的个数和处理器数均为奇数时，应该如何更改代码？

10-7　证明10.3.1节中提到的在$\sqrt{n}\times\sqrt{n}$数组中进行转置操作需要$\sqrt{n}\left(\sqrt{n}-1\right)$次通信。

10-8　编写一个并行程序来实现扭曲排序。

10-9　编写一个并行程序找出一组数字中第k个最小的数。使用快速排序的并行版本但仅应用于含有第k个最小的数的一组数。本习题的进一步讨论参见[Cormen，Leiserson and Rivest，1990]。

10-10　为了提高顺序快速排序的性能，人们提出了很多建议（参见[Wainwright，1985]以及其中的参考文献）。以下是该文提到的两个建议：

1）不选第一个数做枢轴，而是从随机抽取3个数中取中值作为枢轴，即“三者取中”（“median-of-three”技术）。

2）使用第一个数作枢轴最初的一组数被分割成两部分，在其他数与枢轴进行比较的同时计算两部分的每一部分中的数字和，根据和值得到每部分数据的平均值。将每部分数据的平均值作为下一分割阶段的枢轴，计算两个平均值的过程在下一个枢轴的后续步骤中完成。这个算法称为平均排序算法（meansort algorithm）。

根据经验，展望一下并行化上述两个方法的前景。

10-11　使用10.2.6节中描述的并行超立方体画出四维超立方体的数交换过程，以嵌入式环的

次序保留结果，并据此得出适用于任意规模超立方体的通用算法。

10-12 画出构成10.2.5节中所描述的奇偶归并排序算法的比较和交换电路排列(针对16个数)。使用奇偶归并排序算法，手工对以下序列进行排序:

12 2 11 4 9 1 10 15 5 7 14 3 8 13 6 16

10-13 对双调谐归并排序重做习题10-12。

10-14 比较Batcher的奇偶归并排序算法（10.2.5节）和他的双调谐归并排序算法（10.2.6节），并估计在消息传递的多计算机系统中它们并行实现的相对优越性。

10-15 证明使用n个处理器的奇偶归并排序算法的时间复杂性为$O(\log^2 n)$。 338

10-16 试识别在第10章中哪些排序算法是就地（in place）操作的（即在它们排序数时不需要使用附加的存储区）。就地算法将减少存储需求，这一因素在排序大量数时可能非常重要。

10-17 讨论如果是从最高有效位到最低有效位进行排序操作，基数排序能否工作?

10-18 用MPI实现两个排序算法并比较它们的性能。

10-19 阅读[Helman，Bader，and JáJá，1998]所写的论文，并用MPI实现他们的排序算法。

现实生活习题

10-20 Fred有52张完全打乱的扑克牌，他请你确定关于重新排列它们的几个问题:

1）给定扑克有四种花色（黑桃、红桃、草花和方块），则用双调谐归并排序算法对扑克牌进行排序时应对该算法做怎样的改动?

2）Fred至少要邀请多少朋友来并行执行更改后的双调谐归并排序，每个朋友要执行多少步?

10-21 找出两个文件中相匹配内容的一种方法是将文件中的内容先进行排序然后进行项比较;另一种方法是不对文件内容进行排序而是简单地直接进行逐项比较。请对比这两种方法，并根据第一种方法编写一个并行程序来找出两个文件中相匹配的项。

10-22 使用排序算法可以生成输入序列的任意变换，这只要将输入元素按所需次序编号，并按编号对其排序即可。编写一个并行程序随机化文档文件的内容来保护文件的安全性，首先使用随机数生成器将文件中的字编号，然后对这些随机数进行排序。(有关随机数的细节参见第3章。）请问这是不是加密文件的好方法?再编写一个并行程序将此文档文件恢复成它原始状态。 339

第11章 数值算法

在本章中，我们研究几个重要的数值问题：

- 矩阵相乘
- 用直接法求解一般线性方程组
- 用迭代方法求解稀疏线性方程组和偏微分方程组

在前面几章中描述具体的并行程序设计技术时，我们已经介绍了解决这些问题的一些方法。在本章中，我们将更详细的描述这些方法，并且还将介绍一些其他的程序设计方法。

11.1 矩阵回顾

许多科学问题的基础都是矩阵。让我们回顾一下矩阵的数学概念，矩阵是一些数（或代表数的变量）的二维数组。一个$n \times m$的矩阵有n行和m列元素，一个$n \times m$的矩阵A，如图11-1所示。我们在顺序程序设计中已经知道这种结构可以用二维数组表示，并且通常用数组来存储一个矩阵。

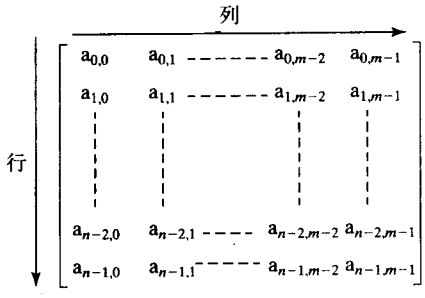


图11-1 $n \times m$ 矩阵

11.1.1 矩阵相加

矩阵相加就是简单地将每个矩阵的对应元素加起来形成一个结果矩阵。将矩阵A的元素记为$a_{i,j}$，矩阵B的元素记为$b_{i,j}$，则矩阵C的元素被计算为：

340

$$c_{i,j} = a_{i,j} + b_{i,j}$$
$$(0 \leqslant i < n, 0 \leqslant j < m)$$

11.1.2 矩阵相乘

两矩阵A和B相乘，乘积C的元素$c_{i,j}$（$0 \leqslant i < n, 0 \leqslant j < m$）可由下式计算得到：

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

其中，A是$n \times l$矩阵，B是$l \times m$矩阵。矩阵A第i行的每个元素与矩阵B第j列的各元素分别相乘，并将乘积相加得到矩阵C的第i行第j列的一个元素$c_{i,j}$，如图11-2所示。矩阵A的列数必须与矩阵B的行数相同，但矩阵的大小可以不同。矩阵也可以与常数相乘（即将矩阵的所有元素都与该常数相乘）。

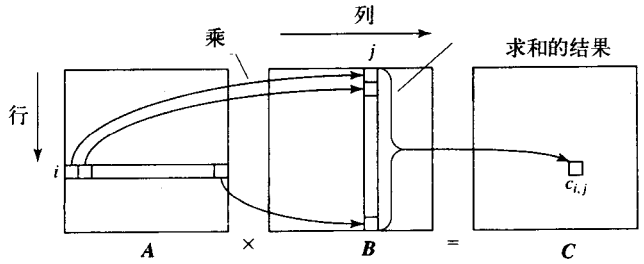


图11-2 矩阵相乘$C = A \times B$

11.1.3 矩阵-向量相乘

向量是一个列数为1的矩阵（即是一个$n \times 1$矩阵）。如果规定矩阵-矩阵乘法的定义中的矩阵B是一个$n \times 1$矩阵，便得到矩阵-向量乘法的定义。相乘的结果是一个$n \times 1$矩阵，如图11-3所示。相反，向量A也可以只有1行（即$1 \times n$矩阵），而矩阵B是一个$n \times m$矩阵，向量-矩阵乘法便得到一个$1 \times m$矩阵（向量）。 341

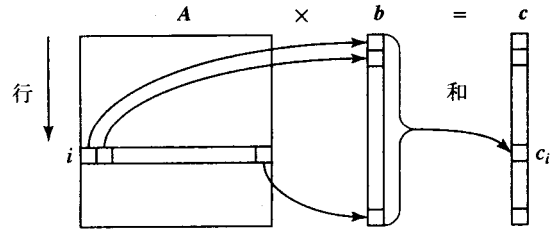


图11-3 矩阵-向量相乘$c = A \times b$

11.1.4 矩阵与线性方程组的关系

在描述工程、商业和科学领域中的日常问题的数学公式中，经常会遇到矩阵。在6.3.1节中所描述的线性方程组可写成如下的矩阵形式:

$$Ax = b$$

A是一个含有常数a的矩阵，x是一个未知向量，b是一个常数b向量。矩阵乘法可用来变换线性方程组[Kreyszig, 1962]。矩阵和矩阵运算可以出现在其他情况中。例如，它可以用来寻找图中两个顶点间的最短路径。

11.2 矩阵乘法的实现

11.2.1 算法

1. 顺序代码

为方便起见，我们总是假设矩阵是一个$n \times n$方阵。根据上述矩阵相乘的定义，计算$A \times B$的顺序代码可简单地表示为:

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++) {
      c[i][j] = 0;
      for (k = 0; k < n; k++)
         c[i][j] = c[i][j] + a[i][k] * b[k][j];
   }
```

这个算法需要进行n^3次乘法运算和n^3次加法运算，顺序时间的复杂性为O (n^3)。(为了提高计算效率，可以用临时变量sum代替c[i][j]，这样在内层for循环的每一次迭代中就不必进行地址计算了。)

2. 并行代码

并行矩阵乘法通常直接基于顺序矩阵相乘算法。只要粗略地观察一下顺序代码，便可以发现两个外层循环的每一次迭代不依赖于其他任何迭代，并且内层循环的各个步骤可以并行执行。因此，当用$p = n$个处理器进行$n \times n$矩阵乘法时，可容易地获得并行时间的复杂性为O (n^2)。 342

当用$p = n^2$个处理器时，可容易地获得并行时间的复杂性为O (n)。这时，一台处理器只分得$\boldsymbol{A}$和$\boldsymbol{B}$的一个元素。这些实现是代价最优的（因为O (n^3) = n × O (n^2) = n^2 × O(n)）。正如我们将要说明的那样，当用$p = n^3$个处理器时，通过并行化内层循环，可以获得O(log$n$)的时间复杂性，虽然这种实现不是代价最优的[因为O(n^3) ≠ n^3 × O(log$n$)]。根据[Moldovan，1993]O(log$n$)实际上是并行矩阵相乘时间复杂性的下界。这里所引用的时间复杂性是仅就计算而言的。但不幸的是，任何额外的通信开销都是非常显著的。

划分成子矩阵　通常，我们希望对$n \times n$矩阵所使用的处理器数目远小于n。因此，每个处理器要操作一组数据点（数据划分）。矩阵乘法可以很容易地处理数据划分，每一个矩阵都可以分成若干元素块，称为子矩阵，这些子矩阵作为单个的矩阵元素参加运算（参见[Fox et al.，1988]）。假设矩阵被分成s^2个子矩阵（行s等分，列s等分），每个子矩阵有$n/s \times n/s$个元素，若记$m = n/s$，则称为有$m \times m$个元素。用$\mathtt{A}_{p,q}$表示第p行第q列的子矩阵，我们简单地用矩阵乘法替换前面代码中的内部求和运算；即：

```
for (p = 0; p < s; p++)
  for (q = 0; q < s; q++) {
    C[p,q] = 0;                          /* clear elements of submatrix */
    for (r = 0; r < m; r++)              /* submatrix multiplication and */
      C[p,q] = C[p,q] + A[p,r] * B[r,q];  /* add to accumulating submatrix */
  }
```

其中

```
C[p,q] = C[p,q] + A[p,r] * B[r,q];
```

表示利用矩阵乘法操作将子矩阵$\mathtt{A}_{p,r}$和$\mathtt{B}_{r,q}$相乘，再利用矩阵加法操作将乘积累加到子矩阵$\mathtt{C}_{p,q}$上。事实上，内层循环包括加循环。这种方法便是众所周知的**块矩阵乘法**，当处理器数目少于n时，该方法是所有并行实现的核心。块矩阵乘法如图11-4所示。注意到子矩阵相加只是简单将两个子矩阵的对应元素相加，得到结果子矩阵的相应元素。图11-5表示两个4 × 4矩阵被划分成4个2 × 2子矩阵，这种结构实际上暗示了一个递归解。但还有一些实现矩阵相乘的不同算法。

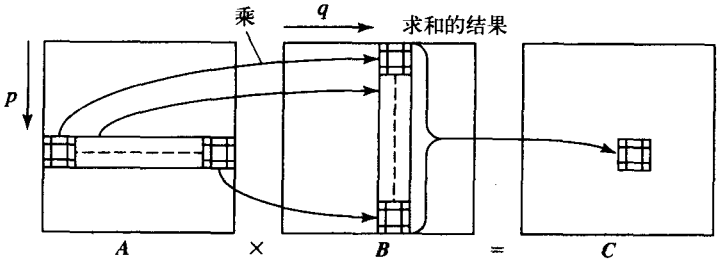


图11-4　块矩阵乘法

11.2.2　直接实现

一种实现矩阵乘法的方法就是，用一台处理器计算$\boldsymbol{C}$的一个元素，这就需要n^2个处理器，
343 且每台处理器分得矩阵$\boldsymbol{A}$的一行元素和矩阵$\boldsymbol{B}$的一列元素，如图11-6所示。使用主从方法，这些元素从主处理器发往选择的从处理器。注意，一些相同的元素必需被发往多个处理器。利用子矩阵，一台处理器将计算$\boldsymbol{C}$的一个$m \times m$子矩阵。

$$\left[\begin{array}{cc|cc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array}\right] \times \left[\begin{array}{cc|cc} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{array}\right]$$

a）矩阵

$$\begin{matrix} A_{0,0} & & B_{0,0} & & A_{0,1} & & B_{1,0} \end{matrix}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix} \times \begin{bmatrix} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0} + a_{0,1}b_{1,0} & a_{0,0}b_{0,1} + a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0} + a_{1,1}b_{1,0} & a_{1,0}b_{0,1} + a_{1,1}b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2}b_{2,0} + a_{0,3}b_{3,0} & a_{0,2}b_{2,1} + a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0} + a_{1,3}b_{3,0} & a_{1,2}b_{2,1} + a_{1,3}b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0} + a_{0,3}b_{3,0} & a_{0,0}b_{0,1} + a_{0,1}b_{1,1} + a_{0,2}b_{2,1} + a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0} + a_{1,1}b_{1,0} + a_{1,2}b_{2,0} + a_{1,3}b_{3,0} & a_{1,0}b_{0,1} + a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

b）$A_{0,0} \times B_{0,0}$得到$C_{0,0}$

图11-5 子矩阵乘法

1. 分析

（1）通信 我们先假设$n \times n$矩阵而不是子矩阵（子矩阵将在稍后考虑）。通过独立的消息将数据分发到n^2台从处理器上，则每台处理器收到一行和一列元素（即$2n$个元素）。此外，每台从处理器将向主处理器返回C的一个元素。其通信时间为：

344

$$t_{comm} = n^2(t_{startup} + 2nt_{data}) + n^2(t_{startup} + t_{data}) = n^2(2t_{startup} + (2n+1)t_{data})$$

用主处理器收集结果，可以使这个收集串行化。

可以看到矩阵A和B的元素的选择性传输会带来过重的通信开销并支配通信时间。事实上，依赖于具体的广播算法，将两个完整矩阵用一次广播传送给每个从处理器可减少通信开销。广播沿单总线发送所需的通信时间为：

$$t_{comm} = (t_{startup} + n^2 t_{data}) + n^2(t_{startup} + t_{data})$$

现在，通信的大部分时间消耗在返回结果上，因为$t_{startup}$远大于t_{data}。如果使用像树结构这样的有效算法，那么集中例程就会减少返回结果所需的时间（如第2.3.4节所描述的利用树结构的方法）。

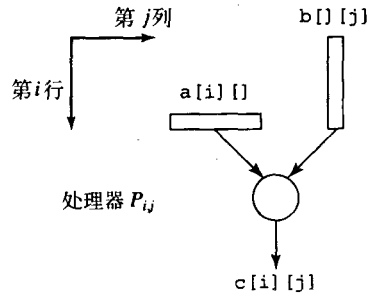


图11-6 矩阵相乘的直接实现

（2）计算 计算仅发生在从处理器上。每台从处理器并行执行n个乘法操作和n个加法操作；即：

$$t_{comp} = 2n$$

使用树结构可在n台处理器上用$\log n$时间步完成n个元素的加法运算，从而提高了性能。图11-7

示出了4 × 4矩阵$c_{0,0}$的计算。因此，我们在使用n^3台处理器时可获得O (log$n$)的时间复杂性而不是用n^2台处理器获得O (n)的时间复杂性。

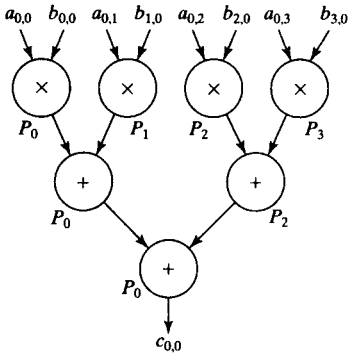


图11-7　使用树结构求和

2. 子矩阵

对于每一种方法，我们都可以用子矩阵代替矩阵元素，以减少处理器的数目。假设，我们选择$m \times m$的子矩阵并令
345 $s = n/m$，即有s行s列个子矩阵。这样，每个矩阵就有s^2个子矩阵，系统需要s^2个处理器。

（1）通信　s^2个从处理器中的每一台，必须分别接收子矩阵中的一行和一列元素，每一个子矩阵有m^2个元素。此外，每台从处理器必需向主处理器返回一个有m^2个元素的子矩阵$\boldsymbol{C}$。其通信时间为:

$$t_{\text{comm}} = s^2\{2(t_{\text{startup}} + nmt_{\text{data}}) + (t_{\text{startup}} + m^2 t_{\text{data}})\} = (n/m)^2\{3t_{\text{startup}} + (m^2 + 2nm)t_{\text{data}}\}$$

同样，全部矩阵都会被广播到每台处理器上。注意，当矩阵规模m增加时（处理器数减少引起的），每条消息的数据传输时间增加了，但消息的实际数目减少了。至于这个通信时间函数在给定自变量m的情况下，是否存在最小值，留给习题（习题11-1）。

（2）计算　算法操作的总数基本没有变。每个从处理器并行执行s个子矩阵乘法和s个子矩阵加法。顺序执行一次子矩阵乘法运算需要m^3次乘法操作和m^3次加法操作，此外，执行一次子矩阵加运算又需要m^2次加法操作。因此有:

$$t_{\text{comp}} = s(2m^3 + m^2) = \text{O}(sm^3) = \text{O}(nm^2)$$

在整个算法分析中，我们假设结果单元在初始化时被置为零，之后的值就在其上累加得到。这使得加法操作的次数与被累加到和的值相等。

11.2.3　递归实现

块矩阵相乘算法暗示了一个递归的分治求解，就像[Horowitz and Zorat，1983]和[Hake，1993]所描述的那样。该算法对于并行实现具有巨大的潜力，尤其是对于共享存储器的并行实现更为如此。

首先考虑，两个$n \times n$矩阵$\boldsymbol{A}$和$\boldsymbol{B}$，这里n是2的乘方（习题11-2研究了如何处理n不是2的乘方的情况）。每个矩阵被划分成4个子方阵，如前面图11-5所示。设$\boldsymbol{A}$的子矩阵为A_{pp}、A_{pq}、A_{qp}
346 和A_{qq}并且$\boldsymbol{B}$的子矩阵为B_{pp}、B_{pq}、B_{qp}和B_{qq}（p和q分别标识行位置和列位置）。最后结果需要8对子矩阵相乘。$A_{pp} \times B_{pp}$、$A_{pq} \times B_{qp}$、$A_{pp} \times B_{pq}$、$A_{pq} \times B_{qp}$、$A_{qp} \times B_{pp}$、$A_{qq} \times B_{qp}$、$A_{qp} \times B_{pq}$和$A_{qq} \times B_{qq}$，并且将结果对加到一起。如图11-8所示。用相同的算法可对每个子矩阵做乘法运算。将子矩阵分成4个次子矩阵，用相同的算法可对每个子矩阵做乘法运算，依此类推。因此，算法可写成下面的递归形式:

```
mat_mult(A, B, s)
{
if (s == 1)                    /* if submatrix has one element */
   C = A * B;                  /* multiply elements */
else {                         /* else continue to make recursive calls */
   s = s/2;                    /* the number of elements in each row/column*/
   P0 = mat_mult(App, Bpp, s);
   P1 = mat_mult(Apq, Bqp, s);
   P2 = mat_mult(App, Bpq, s);
   P3 = mat_mult(Apq, Bqq, s);
```

```
    P4 = mat_mult(A_qp, B_pp, s);
    P5 = mat_mult(A_qq, B_qp, s);
    P6 = mat_mult(A_qp, B_pq, s);
    P7 = mat_mult(A_qq, B_qq, s);
    C_pp = P0 + P1;              /* add submatrix products */
    C_pq = P2 + P3;              /* to form submatrices of final matrix */
    C_qp = P4 + P5;
    C_qq = P6 + P7;
  }
  return (C);                    /* return final matrix */
  }
```

8次递归调用的每一次调用在不同处理器上同时执行，其他处理器在更深层递归调用后才被分配。通常情况下，如果用一台处理器来执行由递归调用产生的任务中的一个任务，则处理器的数目应为8的乘方。递归调用的深度不是由s（每个子矩阵的每行和每列中的元素数）等于1时停止递归调用来限制的，而是由可用的处理器数目指定的某个较大的数来限止的。例如，在8台处理器的情况下，在第一层（即$s = n/2$）就停止递归调用可能比较好[Hake，1993]，因为更深层次的问题划分仍需要将任务映射到这些处理器上。 347

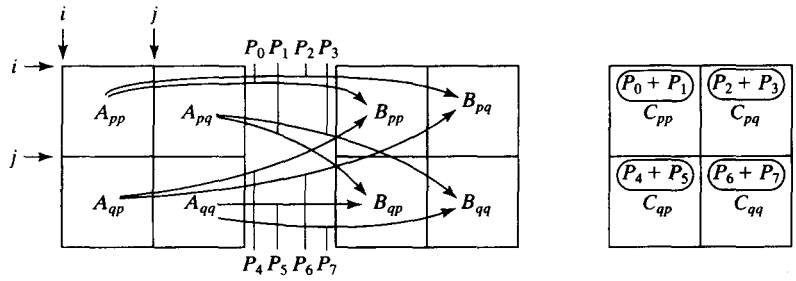


图11-8 子矩阵相乘并求和

这种方法的一个非常有利的方面就是，在每一次递归调用中，被传递的数据大小被减小并且被局部化。这在具有高速缓存的多处理机系统中将可获得最佳的性能。由于这种算法中没有特定的消息传递，所以这种方法尤其适用于共享存储器系统。与此相反，下一节中的矩阵相乘算法含有消息传递特征，因此更适用于消息传递系统。

11.2.4 网格实现

最适合矩阵运算的消息传递体系结构就是二维网格，网格中的每一个结点计算结果数组中的一个元素（或是一个子矩阵）。网格的连接将使相邻结点间的消息能够同时传递。即使在非网格的体系结构中，尽管存在消息竞争和巨大的延迟，也可用适当数目的处理器来建模网格。例如，使用单以太网线连接的工作站时，网格算法肯定会引起消息争用，而使用结点间不同的以太网链路上的实际的网格计算机时这种争用就不会发生。这种性能方面的差异表明，每个结点应处理大块元素（子矩阵）以减少结点间的通信。

已经在网格结构上开发了好几种矩阵相乘算法，我们将介绍其中的两个，即Cannon 算法和脉动方法。另一个矩阵相乘算法是由Fox提出的，有关它的详细描述请参见[Fox et al.，1988]。

1. Cannon算法

Cannon算法[Cannon，1969]使用带有环绕连接的处理器网格向左移动A元素（或子矩阵）并向上移动B元素（或子矩阵），如图11-9所示。所有移动都可环绕。尽管通常使用子矩阵，

但为清晰起见我们仍使用数组**A**和**B**的元素。该算法的工作步骤可描述为如下:

1）初始时处理器$P_{i,j}$有元素$a_{i,j}$和$b_{i,j}$（$0 \leqslant i < n, 0 \leqslant j < n$）。

2）将元素从初始位置移到“对准的”位置：即**A**的第i行整行向左移动i个位置，**B**的第j列整列向上移动j个位置。这使得元素$a_{i,j+i}$和$b_{i+j,j}$均被移到处理器$P_{i,j}$上。这些成对的元
348 素是在$c_{i,j}$累加时所需要的，如图11-10所示。

3）将每个处理器$P_{i,j}$上的元素相乘。

4）将**A**的第i行向左移动1个位置，**B**的第j列向上移动1个位置，从而使得**A**、**B**的相邻元素在一起以满足求和所需，如图11-11所示。

5）将移到每个处理器$P_{i,j}$上的元素对相乘，并将乘积加到累加和上。

6）重复第4步和第5步直到产生最后结果（将n行n列元素做了$n-1$次移动）。

该算法不需要存放部分结果的额外存储空间。

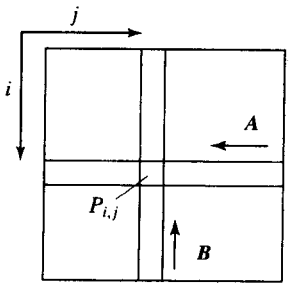


图11-9　**A**和**B**的元素的移动

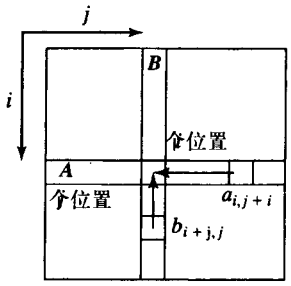


图11-10　第2步——对准**A**和**B**的元素

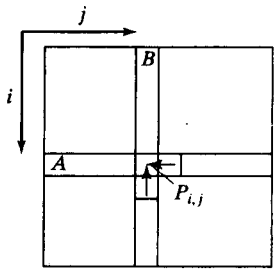


图11-11　第4步——**A**和**B**的元素移动1个位置

2. 分析

（1）通信　回到子矩阵，给定s^2个$m \times m$子矩阵，初始A和B各自对准最多需要$s-1$次移动（通信）操作。之后，A和B分别将有$s-1$次移动操作，每次移动操作涉及到$m \times m$个元素。因此，

$$t_{\text{comm}} = 4(s-1)(t_{\text{startup}} + m^2 t_{\text{data}})$$

也可以写成通信复杂性O (sm^2)或O (mn)。

（2）计算　每一个子矩阵乘法需要m^3次乘操作和m^3次加操作。因此，在$s-1$次移动中，计算时间为:

$$t_{\text{comp}} = 2sm^3 = 2m^2 n$$

349 也可以写成时间复杂性的形式O ($m^2 n$)。

3. 二维流水线——脉动阵列

脉动阵列（Systolic）一词源于医学领域——就像心脏泵取血液一样，信息通过脉动阵列，以某种规则的间隔从不同方向被泵取。这里考虑的是二维脉动阵列，信息从左被泵取到右、从上被泵取到下。信息在内部结点上相遇，在其中被处理，而同一信息又向前（从左到右或从上到下）传递。

图11-12描绘了一个用二维脉动阵列，对两个4 × 4矩阵**A**和**B**进行乘法运算的过程。**A**的元素从左输入，**B**的元素从上输入。最后的结果**C**存储在如图11-12所示的各个处理器中。用x和y坐标恰当地给处理器编号，从左上角（0, 0）开始。每台处理器$P_{i,j}$重复地执行相同的算法（在c被初始化为0之后）:

```
recv(&a, P_{i,j-1});                /* receive from left */
recv(&b, P_{i-1,j});                /* receive from right */
```

```
c = c + a * b;                    /* accumulate value for ci,j */
send(&a, P_{i,j+1});              /* send to right */
send(&b, P_{i+1,j});              /* send downwards */
```

该算法累加所需的和。

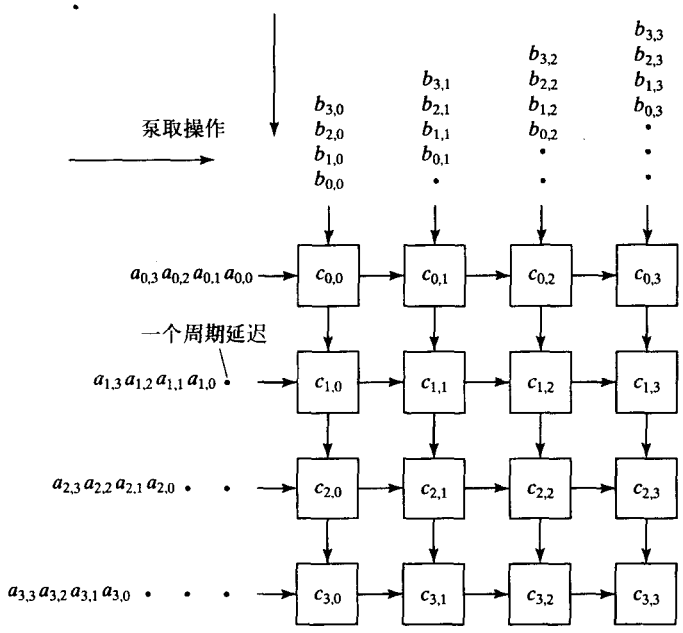


图11-12 使用脉动阵列进行矩阵相乘

我们考虑一下在处理器$P_{0,0}$上发生的情况。首先，$a_{0,0}$和$b_{0,0}$进入$P_{0,0}$，相乘得$c_{0,0} = a_{0,0}b_{0,0}$。$a_{0,0}$和$b_{0,0}$继续向前分别被传送到$P_{0,1}$和$P_{1,0}$上。接着，$P_{0,0}$从左边接收$a_{0,1}$，从上边接收$b_{1,0}$。在将这两个元素相乘后，乘积加到$c_{0,0}$上，并将$a_{0,1}$和$b_{1,0}$按规定方向继续向前传送。对每一对从左边和上边进入的元素进行同样的操作，最后在四次循环之后，我们得到所需的结果：

350

$$c_{0,0} = a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0} + a_{0,3}b_{3,0}$$

$P_{0,1}$以类似的方式操作，但它迟后一个周期从$P_{0,0}$处接收数据，因此来自上面的数据也需迟后一个周期。$p_{0,1}$从对$a_{0,1}$和$b_{0,1}$相乘开始，在四个周期后得到$c_{0,1}$；即，

$$c_{0,1} = a_{0,0}b_{0,1} + a_{0,1}b_{1,1} + a_{0,2}b_{2,1} + a_{0,3}b_{3,1}$$

在其他处理器上也进行类似的计算。尽管没有一个公共的时钟信号，计算操作通过定时的`send()`和`recv()`以同步方式进行。必须在输入处生成$\boldsymbol{A}$和$\boldsymbol{B}$的元素，如图11-12所示。这些元素的交错布局仅是为了指明定时。这种方法也可以应用到子矩阵分解上，当然此时需用子矩阵代替矩阵元素。

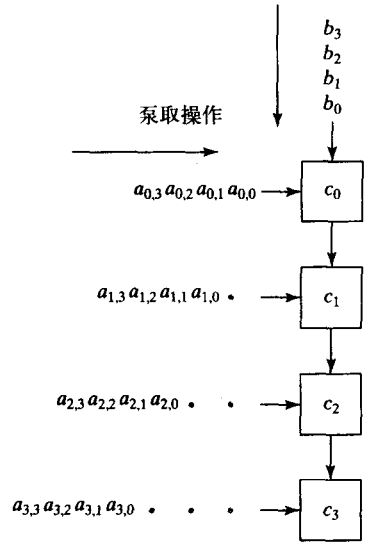


图11-13 使用脉动阵列进行矩阵-向量相乘

351

4. 分析

此处的分析与Cannon算法的分析非常类似，将其留作习题（习题11-3）。

5. 矩阵-向量乘法

显然，在矩阵相乘的算法中，若矩阵B只有一列元素，则算法就可以简单地应用到矩阵-向量乘法中。用来求解矩阵-向量乘法的脉动阵列如图11-13所示。

11.2.5 其他矩阵相乘方法

顺序矩阵乘法的时间复杂性O (n^3)可以用[Strassen，1969]的一个更巧妙的方法得到一点改进。这个算法在习题11-4中描述，在一些关于算法的教科书中也有描述，如[Aho, Hopcroft, and Ullman，1974]和[Cormen, Leiserson, and Rivest，1990]。该算法的时间复杂性为O ($n^{2.81}$)。后续的研究人员对这个结果只做了极小的改进即，O ($n^{2.376}$)。但是，在一般情况下，由于所有方法的附加复杂性没理由使用它们，除非n确实非常大。[Hake，1993]在巨型机Cray-Y-MP机器上，利用基本的分治矩阵相乘算法和Strassen算法所得到的结果表明，使用Strassen算法仅稍微提高了执行速度。矩阵乘法也可以使用到三维数组上。

11.3 求解线性方程组

11.3.1 线性方程组

假设我们有一个线性方程组：

$$
\begin{aligned}
&a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad \cdots \quad + a_{n-1,n-1}x_{n-1} = b_{n-1}\\
&\qquad\qquad\qquad

所代替。此过程从第一行开始，并向下进行。在第i行，第i行以下的每一行j都将被（第j行）+（第i行）$(-a_{j,i}/a_{i,i})$所替换。在第j行用的常数是$-a_{j,i}/a_{i,i}$，以使第i列第j行以下的元素变成零，这是因为：

$$a_{j,i} = a_{j,i} + a_{i,i}\left(\frac{-a_{j,i}}{a_{i,i}}\right) = 0$$

图11-14是考虑第i行的情况。行i被称为是*主元行*（pivot row）。在这时，所有在对角线以下、第i列以左的列都被以前的操作置为零；所有在对角线以下、第i列以右的列都被以后的操作置为零。一旦这个过程持续到最后一行，便得到了一个上三角方程组。

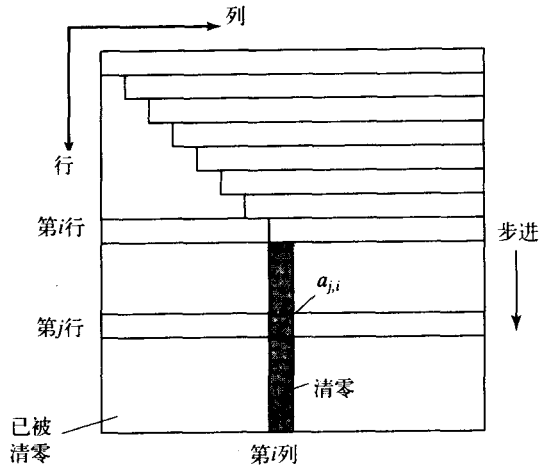


图11-14 高斯消去法

不幸的是，这个过程在数字计算机上没有表现出很好的稳定性。尤其当$a_{i,i}$为0或接近于0时，我们不能计算$-a_{j,i}/a_{i,i}$的值；所以这个过程必需被修改成所谓的*局部选主元*（partial pivoting），即交换第i行和它下面的行，使第i行以下的所有行的第i列中绝对值最大的元素被交换到第i行上。（交换方程组中方程的位置所得的新方程组与原方程组等价。）那么就可以在第i行进行高斯消去法了。在此过程的每一步都必须检查数值的稳定性，并且无法在高斯消去之前进行。下面，我们将不考虑会带来一些额外计算的局部选主元的方法。

353

顺序代码

下面是不考虑局部选主元方法实现高斯消去发的顺序代码：

```
for (i = 0; i < n-1; i++)          /* for each row, except last */
   for (j = i+1; j < n; j++) {     /* step through subsequent rows */
      m = a[j][i]/a[i][i];         /* Compute multiplier */
      for (k = i; k < n; k++)      /* modify last n-i-1 elements of row j */
         a[j][k] = a[j][k] - a[i][k] * m;
      b[j] = b[j] - b[i] * m;      /* modify right side */
   }
```

时间复杂性为O (n^3)。

11.3.3 并行实现

观察顺序代码，显然有并行化余地。内部循环中，修改第j行包括一些独立于第j行上的元素的操作。一种划分该问题的方法是，一台处理器持有一行元素并操作该行。如果有n个方程，就需要n台处理器。在处理器能够操作这行之前，必须从第i行接收元素，这可以通过广播操作实现。首先，处理器P_0（持有第0行元素）向其他所有$n-1$台处理器广播这行元素，之后这些处理器计算它们各自的乘数，并修改它们的行。这一过程在$P_1, P_2, \cdots, P_{n-2}$上被重复执行。被处理器$P_i$广播的第$i$行元素是a[i][i+1]，a[i][i+2]，…，a[i][n-1]和b[i]，共$n-i+1$个元素（$0 < i < n-1$），如图11-15所示。处理器P_{i+1}到P_{n-1}包括它们自己（共$n-i-1$台处理器）将

收到广播消息，并处理它们各自行上的$n-i+1$个元素。

1.分析

（1）通信　假设广播消息能在一步内完成。因为在每一次广播中要交换行，因此$n-1$个广播消息必须顺序执行。第i次广播消息包括$n-i+1$个元素。因此，整个消息通信时间为:

$$t_{\text{comm}}=\sum_{i=0}^{n-2}(t_{\text{startup}}+(n-i+1)t_{\text{data}})=\left((n-1)t_{\text{startup}}+\left(\frac{(n+2)(n+1)}{2}-3\right)t_{\text{data}}\right)$$

即时间复杂性为O (n^2)。对于n较大的情况，在前几步数据时间t_{data}将占去通信时间的大部分。

354 实际上，每一次广播消息所需的时间不只一步，它依赖于具体系统的体系结构。

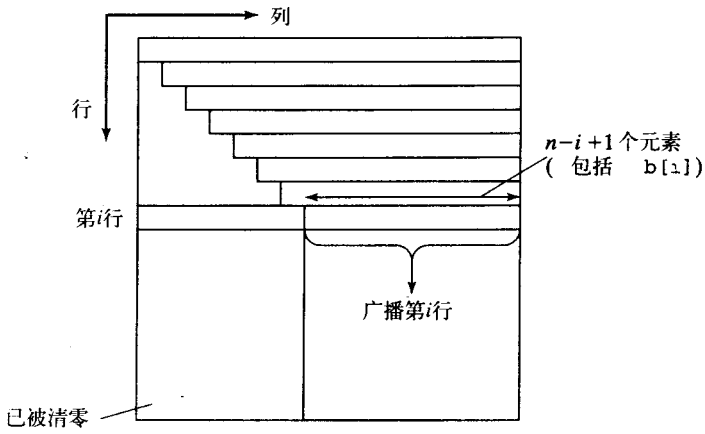


图11-15　高斯消去法中广播的并行实现

（2）计算　广播完一行之后，除了广播处理器P_i外每个处理器P_j就会收到消息，计算乘数，并对该行上的$n-j+2$个元素进行操作。忽略乘数的计算后，还有$n-j+2$次乘法和$n-j+2$次减法，因此计算时间为:

$$t_{\text{comp}}=2\sum_{j=1}^{n-1}(n-j+2)=\frac{(n+3)(n+2)}{2}-3=\text{O}(n^2)$$

注意，由于所有处理器在持有第i行元素之前不参加计算，所以效率会相当低。起初，$n-1$台处理器可以计算，随后，$n-2$台处理器可以参加计算，之后，$n-3$台处理器可以参加计算，以此类推。每一步所做的工作要比上一步少。效率的推导留作习题（习题11-15）.

2. 流水线结构

处理器可以被组织成流水线形式，如图11-16所示。

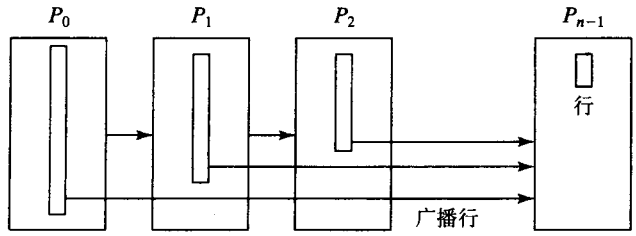


图11-16　高斯消去法的流水线实现

3. 划分

为了使处理器的数目减少到小于n，我们可以将矩阵分割成若干行组，每台处理器持有一

组，这就是如图11-17所示的条状划分。例如，如果有p台处理器，则每台处理器持有n/p行。 355
不幸的是，处理器在处理完它们的最后一行元素后，不再参加计算。另一种划分方法是，使各个处理器的工作负载相等，如图11-18所示，这是一种循环带状划分。现在每台处理器将在较长一段时间内工作。例如，P_0分到第0、n/p、$2n/p$、$3n/p$行，并且动作持续到超过第$3n/p$行。P_1分到第1、$n/p+1$、$2n/p+1$、$3n/p+1$行，并且动作持续到超过第$3n/p+1$行。划分矩阵的最后一种方式就是棋盘方式，如同为矩阵乘法运算对处理器进行划分那样。

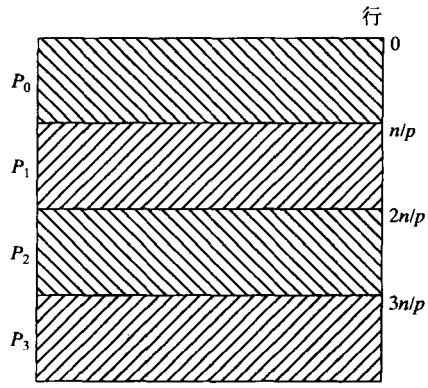


图11-17 条状划分

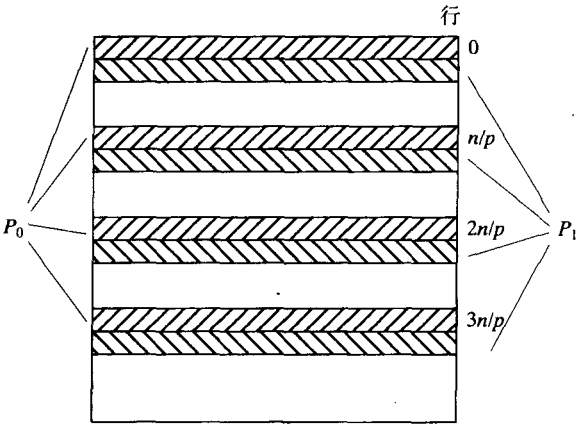


图11-18 用循环划分使工作负载相等

11.4 迭代方法

在这一节中，我们研究通过迭代求解线性方程组和偏微分方程的方法。我们将说明用离散化方法求解偏微分方程组与求解线性方程组的关系。

11.4.1 雅可比迭代

基本的雅可比迭代方法已经在6.3.1节介绍局部同步时介绍过了。给出n个线性方程的一般形式：

$$\boldsymbol{Ax} = \boldsymbol{b}$$

雅可比迭代用下面的迭代公式描述： 356

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1}\right]$$

其中，上角标表示迭代次数，例如，x_i^k表示x_i的第k次迭代，x_j^{k-1}表示x_j的第k−1次迭代$(0 \leqslant i < n, 0 \leqslant j < n)$。迭代公式就是将第$i$个未知数放在第$i$个方程的左边。

至此，我们已经给出了两个求解线性方程组的方法：一种是直接方法（高斯消去法），另一种是迭代法。直接法的时间复杂性是$O(n^2)$（在有n台处理器的情况下）。迭代法的时间复杂性依赖于迭代次数和精度要求。

迭代法的一种特殊应用就是求解稀疏线性方程组，也就是说，方程的项不是很多的线性方程组。这样的方程出现在求解偏微分方程组的方程中。例如最基本的偏微分方程就是拉普拉斯方程：

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

目的是在具有x、y坐标的二维空间中求解函数f。以下的方法也可以应用到泊松方程。泊松方程是类似于拉普拉斯方程的另一种基本方程，只是等式右边是一个函数。习题11-8研究了它的必要变化。

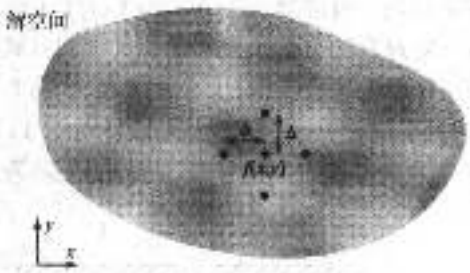


图11-19　有限差分方法

对于计算机求解，有限差分方法是最合适的。在这个方法中，把二维解空间离散化成大量解的点，如图11-19所示。如果两点间在x和y方向上的距离Δ足够小，就可以在第2次迭代中使用中心差分近似：

357

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{1}{\Delta^2}[f(x+\Delta, y) - 2f(x, y) + f(x-\Delta, y)]$$

$$\frac{\partial^2 f}{\partial y^2} \approx \frac{1}{\Delta^2}[f(x, y+\Delta) - 2f(x, y) + f(x, y-\Delta)]$$

（证明参见[Bertsekas and Tsitsiklis，1989]）用这个差分公式代入拉普拉斯方程可得：

$$\frac{1}{\Delta^2}[f(x+\Delta, y) + f(x-\Delta, y) + f(x, y+\Delta) + f(x, y-\Delta) - 4f(x, y)] = 0$$

整理后就可得到：

$$f(x, y) = \frac{[f(x-\Delta, y) + f(x, y-\Delta) + f(x+\Delta, y) + f(x, y+\Delta)]}{4}$$

即为6.2.3节中使用的结果。

这个公式可被重新写成迭代公式的形式：

$$f^k(x, y) = \frac{[f^{k-1}(x-\Delta, y) + f^{k-1}(x, y-\Delta) + f^{k-1}(x+\Delta, y) + f^{k-1}(x, y+\Delta)]}{4}$$

其中$f^k(x, y)$是在第k次迭代中得到的，$f^{k-1}(x, y)$是在第$k-1$次迭代中得到的。通过重复应用这个公式，最终可以收敛到我们所需的解，如第6章所描述的那样。

像在第6章中一样，我们假设求解空间是正方的，并且解的点是按行排列的。设每行有m个点，共有m行，故数组有$m \times m$个点。为了计算方便，假设边界是由点构成的，但它们的值是已知的。边界点在计算与其相邻点的值时被用到，并且被包含在$m \times m$数组中。假设这些点按自然数序编号，在网格中是按行排列的。第1行中含有点$x_1, x_2, x_3, \cdots, x_m$，第2行有点$x_{m+1}, x_{m+2}, x_{m+3}, \cdots, x_{2m}$，以此类推，图11-20给出了100个点，每行10个点的情况（包括已知值的边界点）。



图11-20　按自然数序编号的网格点

1. 与一般线性方程组的关系

第i个点的值由第i个方程计算出，这个方程包含点x_i，x_{i-1}、x_{i-m}、x_{i+1}和x_{i+m}，按自然数序即：

$$x_i = \frac{x_{i-m} + x_{i-1} + x_{i+1} + x_{i+m}}{4}$$

或者是:

$$x_{i-m} + x_{i-1} - 4x_i + x_{i+1} + x_{i+m} = 0$$

这是一个具有5个未知变量（边界点除外）的线性方程组。在一般形式中，则第i个方程为:

$$a_{i,i-m}x_{i-m} + a_{i,i-1}x_{i-1} + a_{i,i}x_i + a_{i,i+1}x_{i+1} + a_{i,i+m}x_{i+m} = 0$$

其中$a_{i,i} = -4$, $a_{i,i-m} = a_{i,i-1} = a_{i,i+1} = a_{i,i+m} = 1$。

矩阵-向量形式的方程如图11-21所示。向量x包括已知边界变量（和4个没被使用过的角上的点）。类似地，矩阵A包括边界值（和没被使用过的项）。例如，在图11-20中，向量x中的变量x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{20}, x_{21}, x_{30}, x_{31}, x_{40}, x_{41}, x_{50}, x_{51}, x_{60}, x_{61}, x_{70}, x_{71}, x_{80}, x_{81}, x_{90}, x_{91}, x_{92}, x_{93}, x_{94}, x_{95}, x_{96}, x_{97}, x_{98}, x_{99}和x_{100}，被置为1，而矩阵A中相应的$a_{i,i}$是边界常数。由此可看到A的通用特征。主对角元素是-4，其上下两条副对角线元素是1，并且在与主对角线相距为n的上下两条副对角线的元素也为1。只有那些在对角线上有未知变量的方程，才能在求解过程中被用到，并且这些方程中的边值作为常数可被移到方程右边。 358

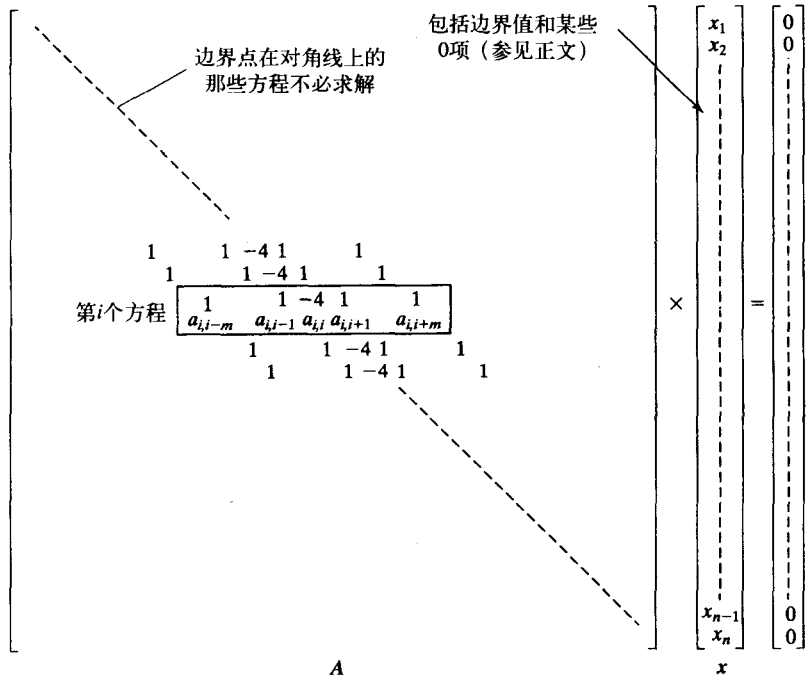


图11-21 拉普拉斯方程求解的稀疏矩阵

对于一个拥有很多点的数组，矩阵就会非常大且稀疏。在这种情况下，迭代方法是最受欢迎的，因为它占用的存储器空间较少，并且具有更快的收敛速度。

2. 收敛速度

不幸的是，由于依赖于矩阵本身，雅可比迭代法收敛得非常慢（有时甚至不收敛！）。按照[Leighton，1992]的观点，对于某一类特定的矩阵，如对角占优矩阵（对角线上的元素比这一行上其他元素的绝对值之和还大），雅可比松弛法可以在log$n$步收敛到较好的解。在这些情况下，n台处理器的并行时间复杂性为O (log$n$)，这比高斯消去法要好。当然，这完全依赖于收敛速率。下面让我们看以下几种可能的快速收敛法。 359

11.4.2 快速收敛方法

1. 高斯-赛德尔松弛法

一种试图加快收敛速度的方法是，在迭代中利用计算出的新值来计算其他值。假设，我们计算出的值排成自然数序x_1, x_2, x_3等，当计算x_i时，x_1, x_2, x_3,…, x_{i-1}已经被算出，可以与还没被重新计算的x_{i+1}, x_{i+2}, x_{i+3},…, x_n一起被用到求解x_i的迭代公式中。这种方法就是著名的*高斯-赛德尔松弛法*。它通常比基本的雅可比方法具有更高的（但不是一定）收敛速度，并且方便进行顺序程序设计，因为它能以指定的串行顺序计算每一点。

高斯-赛德尔迭代利用下面的迭代公式:

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^k - \sum_{j=i+1}^{n} a_{i,j}x_j^{k-1}\right]$$

其中，上角标代表迭代次数，即第k次迭代使用了第k次和第$k-1$次迭代的值。对于未知数以自然数序标号的拉普拉斯方程，公式可化简为:

360

$$x_i^k = (-1/a_{i,i})[a_{i,i-n}x_{i-n}^k + a_{i,i-1}x_{i-1}^k + a_{i,i+1}x_{i+1}^{k-1} + a_{i,i+n}x_{i+1}^{k-1}]$$

（注意: $a_{i,i}=-4$）在第k次迭代中，四个值中的两个（在第i个元素之前的点）来源于第k次迭代，而另外两个（在第i个元素之后的点）来源于第$k-1$次迭代。使用原始的有限差分概念，可有:

$$f^k(x,y) = \frac{[f^k(x-\Delta,y) + f^k(x,y-\Delta) + f^{k-1}(x+\Delta,y) + f^{k-1}(x,y+\Delta)]}{4}$$

当然，要使用这个公式，我们必须在计算其他值之前在第k次迭代中已经计算出某些特定的值，这表明每一步计算必需以一定顺序执行。在高斯-赛德尔法中，更新的顺序是以自然数序排序的。因此，计算将扫过网格点，如图11-22所示。这对并行来讲不是一个特别方便的特性（也许线性流水线解法除外）。但是，可以改变这个顺序，使该方法更适于并行化。

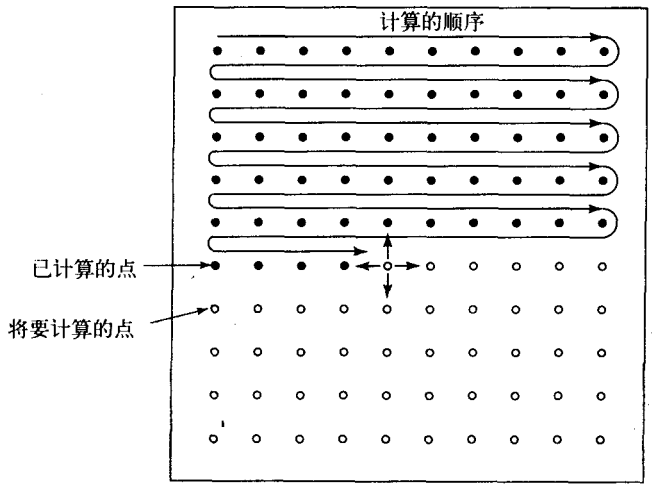


图11-22　以自然数序顺序计算的高斯-赛德尔松弛法

2. 红黑排序法

在红黑排序中，网格中的点被分成交替出现的红点和黑点，如图11-23所示。黑点由四个

相邻的红点计算而来，而红点则由四个相邻的黑点计算而来。这两个阶段交替进行，直到结果收敛。首先，黑点被计算，之后红点被计算。所有的黑点可以同时被计算，所有的红点也可以同时被计算。网格点用下标 (i, j) 表示，当$i+j$为偶数（假设）时该点为黑点，当$i+j$为奇数时该点为红点。显然，这个方法适合并行化。 361

（1）*并行代码* 使用forall来描述并行化，代码形式如下:

```
forall (i = 1; i < m; i++)
  forall (j = 1; j < m; j++)
    if ((i + j) % 2 != 0)                    /* compute red points */
      f[i][j] = 0.25*(f[i-1][j] + f[i][j-1] + f[i+1][j] + f[i][j+1]);
forall (i = 1; i < m; i++)
  forall (j = 1; j < m; j++)
    if ((i + j) % 2 == 0)                    /* compute black points */
      f[i][j] = 0.25*(f[i-1][j] + f[i][j-1] + f[i+1][j] + f[i][j+1]);
```

其中数组有$(m-1)\times(m-1)$个内部数据点。也许通过使用两个独立的数组，一个存储黑点，一个存储红点，可以避免求余运算以提高计算效率。这个修改工作留作习题（习题11-11）。

（2）*处理器分配* 如果我们将一台处理器分给一个点，则每台处理器有一半的时间闲置。更有效的分配方案是，一台处理器分给一对相邻点，一个红点、一个黑点。这样n^2个点就需要$n^2/2$台处理器。

（3）*最大划分* 通常情况下，可能希望使用比网格结点更少的处理器。这里所描述的红黑排序可以推广到红黑跳棋盘排序如[Fox et al., 1988]中所描述的那样，此时网格被分成红区域和黑区域，每个区域不是只有一个点而是有一组相邻点。此后红区域中的点可以同时计算，黑区域中的点也可以同时计算。而分别在红黑区域中的点，则要在单个处理器上顺序计算。

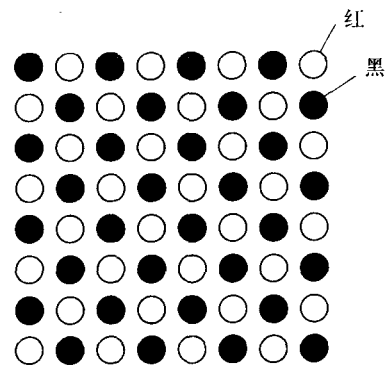


图11-23 红黑排列

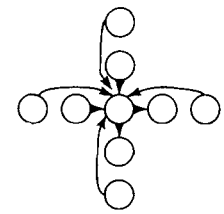
3. 高阶差分法

到目前为止，我们在计算中使用了4个相邻点。为了提高收敛速度和求解精度，可以在计算中使用更远的点。修改过的公式如下: 362

$$f^k(x,y)=\frac{1}{60}\Big[16f^{k-1}(x-\Delta,y)+16f^{k-1}(x,y-\Delta)+16f^{k-1}(x+\Delta,y)+16f^{k-1}(x,$$
$$-f^{k-1}(x-2\Delta,y)-f^{k-1}(x,y-2\Delta)-f^{k-1}(x+2\Delta,y)-f^{k-1}(x,y+2\Delta)\Big]$$

该公式使用8个点来更新一个点，如图11-24所示，这就是所谓的*九点（星形）*模板（而不是以前所说的五点“星形”模板）。这里，我们以模板的大小来计数中心点，该点可在迭代公式中使用。（参见习题11-10）。

有几个可选择的模板，例如，九点模板除了也使用最近的对角线外，与五点星形模板相同；十三点星形模板，它使用九点星形模板中的点和最近对角线的点。在每一步的计算量多到什么程度与收敛步骤数目少到什么程度之间存在一个平衡。将处理器映射到网格点的最明显的方法是:


图11-24 九点模板

或者将一台处理器分到一列网格点上，或者将网格点分成跳棋盘，一台处理器分到一个区域的点。事实上，[Terrano, Dunn, and Peters，1989]已指出最优的方法是，为每一台处理器安排一个多边形，该多边形的边界就是那些离中心点最远的在模板中定义的点。例如，一个简单的五点星型模板可以构造一个菱形多边形。将这几个点映射到一台处理器上，就仍会使每台处理器得到一个菱形。

4. 过度松弛法

在每个雅可比或高斯-赛德尔公式中加入权因子 $(1-\omega)x_i$，就可分别得到过度松弛雅可比法或逐次过度松弛高斯-赛德尔法。因子ω是过度松弛参数。一般线性方程组的过度松弛雅可
363 比迭代公式为:

$$x_i^k = \frac{\omega}{a_{ii}}\left[b_i - \sum_{j\neq i} a_{ij}x_i^{k-1}\right] + (1-\omega)x_i^{k-1}$$

其中，$0 < \omega < 1$，一般线性方程组的逐次过度松弛高斯-赛德尔迭代公式为:

$$x_i^k = \frac{\omega}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_i^k - \sum_{j=i+1}^{N} a_{ij}x_i^{k-1}\right] + (1-\omega)x_i^{k-1}$$

其中$0< \omega \leqslant 2$。如果$\omega = 1$，就得到高斯-赛德尔迭代法。

5. 多网格法

到目前为止，我们所使用点的数目是固定的。在多网格方法中，迭代过程中对网格点的操作与前面所讲的相同，只是在计算过程中，每一步中参加运算的点数可以改变。首先，使用粗网格点（即解空间被分成很少的几个点）。使用少的点可以使迭代过程快速收敛。在某一步，点的数目增加到包括粗网格点和两个粗网格点之间的额外点。额外点的初始值可以通过插值法求得。在这些较细网格点上继续进行计算。随着计算的推进，网格会越来越细，当然也可以交替进行粗网格和细网格的计算。由于粗网格更快地考虑了距离因素，因而为以后的更细网格的形成提供了好的起始点。

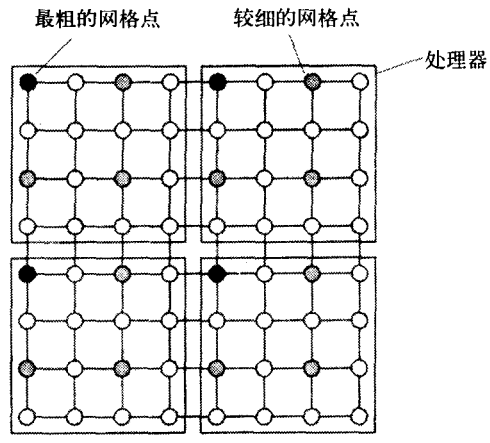


图11-25　多网格处理器分配

为了方便，网格的大小通常是2的乘方。假设，最细的网格有$2^r \times 2^r$个点。稍粗一点的网格有$2^{r-1} \times 2^{r-1}$个点，而再粗一点的网格则有$2^{r-2} \times 2^{r-2}$个点。图11-25示出了将粗网格点分配到每个处理器上去的情况。[Leighton，1992]指出不同层上的网格最合适的数目是log r，即对$1<k\leqslant \log r$网格的大小为$2^k \times 2^k$。

在有些问题中，在解空间的不同位置，可能
364 更适合采用不同大小的网格。网格大小应该能调节以适合局部计算（所谓的自适应网格方法）。这种方法更多细节可以在各种研究论文中找到，例如，[De Keyser and Roose，1997]。

11.5　小结

本章详细讨论了前面几章中所提到的数值问题:

- 矩阵乘法的不同并行实现（直接法，递归法，网格法）
- 用高斯消去法求解线性方程组及其并行实现
- 用雅可比迭代法求解偏微分方程组
- 与线性方程组的关系
- 快速收敛方法（高斯-赛德尔松弛法、红黑排序法、高阶差分法、过度松弛法、多网格法）

推荐读物

关于并行数值方法主要权威的参考书是[Bertsekas and Tsitsiklis，1989]。其他关于并行数值算法的四年级本科生/研究生水平的教科书包括[Freeman and Phillips，1992]、[Modi，1988]和[Smith，1993]。[Grama et al.，2003]提出了用稠密和稀疏矩阵处理并行方法的重要观点。[Chaudhuri，1992]以矩阵计算一章为特色，包括处理布尔矩阵的方法。进一步研究的其他主题还有共轭梯度法和有限元法。关于并行方法的一个精彩的额外信息源是[Van de velde，1994]。[Young，1971]是关于线性方程组迭代解法的教科书。描述多网格法的教科书包括[Hackbrush，1985]。

参考文献

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.

AVILA, J. (1994), "A Breadth-First Approach to Parallel Processing for Undergraduates," *Conf. Par. Computing for Undergraduates*, Colgate University, June 22–24.

BERTSEKAS, D. P., AND J. N. TSITSIKLIS (1989), *Parallel and Distributed Computation Numerical Methods*. Prentice Hall, Englewood Cliffs, NJ.

CANNON, L. E. (1969), *A Cellular Computer to Implement the Kalman Filter Algorithm*, Ph.D. thesis, Montana State University, Bozman, MT.

CHAUDHURI, P. (1992), *Fundamentals of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, NJ.

CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST (1990), *Introduction to Algorithms*, MIT Press, Cambridge, MA.

DE KEYSER, J., AND D. ROOSE (1997), "A Software Tool for Load Balanced Adaptive Multiple Grids on Distributed Memory Computers," *Proc. 6th Distrib. Memory Comput. Conf.*, IEEE CS Press, Los Alamitos, CA, pp. 122–128.

FOX, G., M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER (1988), *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, Englewood Cliffs, NJ.

FREEMAN, T. L., AND C. PHILLIPS (1992), *Parallel Numerical Algorithms*, Prentice Hall, London.

GRAMA, A., A. GUPTA, G. KARYPIS, AND V. KUMAR (2003), *Introduction to Parallel Computing*, 2nd edition, Benjamin/Cummings, Redwood City, CA.

HACKBRUSH, W. (1985), *Multigrid Methods with Applications*, Springer-Verlag, New York.

HAKE, J.-F. (1993), "Parallel Algorithms for Matrix Operations and Their Performance on Multiprocessor Systems," in *Advances in Parallel Algorithms*, L. Kronsjö and D. Shumsheruddin, eds., Halsted Press, New York.

HOROWITZ, E., AND A. ZORAT (1983), "Divide-and-Conquer for Parallel Processing," *IEEE Trans. Comput.*, Vol. C-32, No. 6, pp. 582–585.

KREYSZIG, E. (1962), *Advanced Mathematics*, Wiley, New York.

LEIGHTON, F. T. (1992), *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA.

MODI, J. J. (1988), *Parallel Algorithms and Matrix Computations*, Oxford University Press, Oxford, England.

MOLDOVAN, D. I. (1993), *Parallel Processing from Applications to Systems*, Morgan Kaufmann, San Mateo, CA.

MURTHY, C. S. R., K. N. B. MURTHY, AND S. ALURU (2000), *New Parallel Algorithms for Direct Solution of Linear Equations*, Wiley, New York.

SMITH, J. R. (1993), *The Design and Analysis of Parallel Algorithms*. Oxford University Press, Oxford, England.

STRASSEN, V. (1969), "Gaussian Elimination Is Not Optimal," *Numerische Mathematik*, Vol. 13, pp. 353–356.

TERRANO, A. E., S. M. DUNN, AND J. E. PETERS (1989), "Using an Architectural Knowledge Base to Generate Code for Parallel Computers," *Comm. ACM*, Vol. 32, No. 9, pp. 1065–1072.

VAN DE VELDE, E. F. (1994), *Concurrent Scientific Computing*, Springer-Verlag, New York.

WILSON, G. V. (1995), *Practical Parallel Programming*, MIT Press, Cambridge, MA.

YOUNG, D. M. (1971), *Iterative Solution of Large Linear Systems*, Academic Press, Boston, MA.

习题

科学/数值习题

11-1 11.2.2节中推导的通信时间在子矩阵大小可变的情况下，是否有最小值？

365 ~ 366

11-2 修改递归$n \times n$矩阵相乘算法，这里n不是2的乘方。

11-3 分析11.2.4节中描述的矩阵相乘算法所用到的脉动阵列，推导计算和通信的方程。

11-4 矩阵相乘$\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$，其中

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

$$\boldsymbol{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\boldsymbol{C} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

矩阵相乘的Strassen 方法给出如下:

$Q_1 = (a_{11} + a_{22})\,(b_{11} + b_{22})$

$Q_2 = (a_{21} + a_{22})\,b_{11}$

$Q_3 = a_{11}\,(b_{12} - b_{22})$

$Q_4 = a_{22}\,(-b_{11} + b_{21})$

$Q_5 = (a_{11} + a_{12})\,b_{22}$

$Q_6 = (-a_{11} + a_{21})\,(b_{11} + b_{12})$

$Q_7 = (a_{12} - a_{22})(b_{21} + b_{22})$

$c_{11} = Q_1 + Q_4 - Q_5 + Q_7$

$c_{21} = Q_2 + Q_4$

$c_{12} = Q_3 + Q_5$

$c_{22} = Q_1 + Q_3 - Q_2 + Q_6$

给出此算法的递归并行程序。

11-5 矩阵-向量乘法的一个应用是卷积（convolution），它常用在数字信号处理和图像处理上（参见第12章）。给定常数$\omega_1, \omega_2, \omega_3, \cdots, \omega_n$的序列和输入数据$x_1, x_2, \cdots, X_{N+n-1}$的序列，则卷积运算的输出序列$y_1, y_2, \cdots, y_N$，由下式给出：

$$y_i = \sum_{j=1}^{n} x_{i-j+n} \times w_j$$

其中，$1 \leqslant i \leqslant N$。这个计算可以描述成矩阵-向量相乘形式（对于$N=4$，$n=5$）：

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} x_5 & x_4 & x_3 & x_2 & x_1 \\ x_6 & x_5 & x_4 & x_3 & x_2 \\ x_7 & x_6 & x_5 & x_4 & x_3 \\ x_8 & x_7 & x_6 & x_5 & x_4 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix}$$

367

要求开发一个卷积的并行程序结构。

11-6 求出具有下面有限差分方程解的偏微分方程：

$$x_i = \frac{x_{i-1} + x_{i+1}}{2}$$

$$x_i = \frac{x_{i-1} - 2x_i + x_{i+1}}{2}$$

11-7 开发一个求解下面偏微分方程的有限差分方程：

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} = 0$$

替换11.4节中的中心差分公式，之后编写出求解该方程的并行程序。假设边界值固定，解的维数需被输入。

11-8 写出泊松方程五点和九点雅可比收敛公式。

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = g(x, y)$$

11-9 针对3 × 3点网格画出图11-21中的数组A和向量x的全部内容。

11-10 编写一个能实现下面两个雅可比迭代公式的并行程序，并确定哪一个收敛最快。

$$f^k(x,y) = \frac{1}{4}\left[f^{k-1}(x-\Delta,y) + f^{k-1}(x,y-\Delta) + f^{k-1}(x+\Delta,y) + f^{k-1}(x,y+\Delta)\right]$$

或

$$f^k(x,y) = \frac{1}{8}\left[4f^{k-1}(x,y) + f^{k-1}(x-\Delta,y) + f^{k-1}(x,y-\Delta) + f^{k-1}(x+\Delta,y) + f^{k-1}(x,y+\Delta)\right]$$

11-11 重写11.4.2节中红黑排序的伪代码来避免求余操作。

11-12 为11.4.2节中高斯-赛德尔法开发一个线性流水线解，并编写出实现它的并行程序。

11-13 比较雅可比迭代法的五点、九点、十三点模板。求出计算代价和迭代次数之间的平衡。

11-14 分别写出用下面三种方法求解拉普拉斯方程的并行程序：

1）标准雅可比迭代

2）红黑排序迭代

3）多网格雅可比迭代

使用256 × 256的网格点，并将4条边初始化为10.0、5.0、10.0和5.0。当两次迭代的差值小于0.01时，停止迭代。用16台处理器，对于标准迭代和红黑迭代方法，将问题分

成16列，每列有256 × 16个点，一台处理器分到一列。对于多网格迭代法，开始时网格大小为16 × 16，每10次迭代后，网格大小增加1倍，直到达到网格大小的最大值。继续迭代直到得到解。对其他由粗到细策略进行试验。

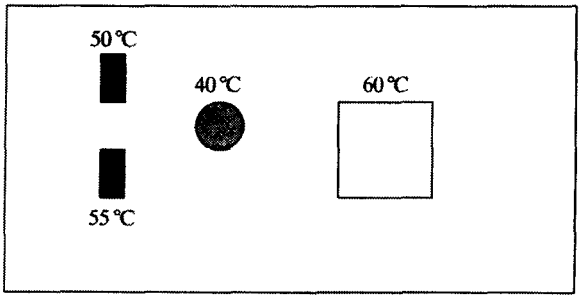
11-15　分别给出在11.3.2节中以条状划分和循环划分实现的高斯消元法的系统效率。

现实生活习题

11-16　编写一个求解室内温度分布问题（如第6章习题6-15所描述的）的并行程序，要求使
368 用高斯消去法和回代的直接方法而不是迭代法。只有高斯消去法可以并行计算，回代只能在一台处理器上运行。首先，确定线性方程组$Ax = 0$中数组A的元素。这个数组的主对角线上的元素总有非零元素，因此没必要使用部分选主元。然后，分解问题，使得连续的10行由一个进程处理。

11-17　写一个求解室内温度分布问题（第6章习题6-15所描述的）的并行程序，但允许使用接近壁炉的较细网格。假设用户可以控制网格的大小。对自适应的网格进行实验。

11-18　图11-26示出了一个装有不同电子元件的印刷电路板，这些电子元件会产生热量，从而达到图11-26上所示的温度。编写一个能计算温度分布的并行程序。自己选择电路板的元件和电路板的尺寸以及元件的布局。这个问题的思路来源于[Avila，1994]。



369 图11-26　习题11-18的印刷电路板

第12章 图像处理

本章是“专业”的一章，它所提供的信息可用作并行编程的课程设计。本章内容包含了图像处理中的主题，它们具有很大的潜在并行性。我们从*低层预处理操作*（low-level preprocessing operations）开始讨论，因为这些操作是在图像增强初期进行的。有时需要识别直线（和曲线），为此我们将叙述完成这种识别的称为*霍夫变换*（Hough transform）的算法。最后，我们将描述使用离散傅里叶变换实现将已存储图像转换成频率域的方法。完成这种变换的一种快速的计算方法称为*快速傅里叶变换*（fast Fourier transform FFT）。有关*傅里叶变换*的资料，除了图像处理外，还可在许多其他应用中加以使用。

12.1 低层图像处理

低层图像处理将直接在已存储图像上进行操作，以改善或增强图像，从而有助于人和计算机更好地识别图像。这种图像处理可用在许多应用场合，包括医疗诊断、警局中指纹识别、检查制造中有缺陷的部件以及胶卷工业等。图像首先由照相机或其他传感器捕捉并以数字化方式存储起来。被存储的图像是由一个二维的*像素*（pixel）（图元）阵列表示的。许多低层图像处理操作都假设图像是单色的，并将像素看成具有某一*灰度级*（gray level）或亮度。此灰度级有一个从最低到最高值的范围（即*灰度*——grayscale）。典型地，灰度被正则化成从零开始，零用来表示黑色而用255来表示白色，因此一个像素需用8位表示。彩色图像则通常需要使用三个值，每一个表示一种主色（红、绿和蓝），或是使用像素值指向一个查找表，关于这一点已在3.2.1节中提及。

我们假设已存储图像所使用的坐标系的原点在左上角，如图12-1所示。图像的像素被存储在二维数组中，一个*像素图*（pixmap）`P[i][j]`以及各个像素点的亮度可以通过访问此数组得到。这里需要指出的是，从本质上来讲，图像和已存储信息两者是一一对应的关系，但大型图像文件可使用压缩结构或使用与图像一起存储的彩色查找表以减小对存储量的需求。当然在进行任何图像处理操作之前，该文件必须复原，且信息仍应以二维数组排列。

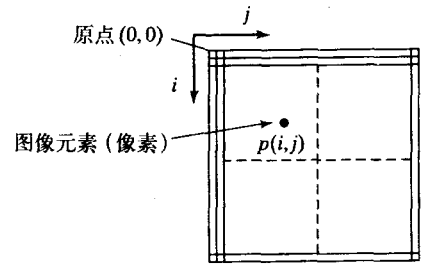


图12-1 像素图

对已存储图像信息可以进行许多不同操作，通常是从*低层处理*开始。低层处理利用各个像素值以某种方式对图像加以修改。通常这些操作是完全并行的。例如，一个图像通常具有“噪声”，这是由传感器或环境所产生的不希望的变化，这种噪声会改变实际的像素值。理想情况下，应去除这种噪声，而只保留所需的图像。完成这种操作通常称为*噪声清除*（noise cleaning）或*噪声减少*（noise reduction）。由于图像像素的真实值是未知的（否则就不需要用传感器来检测了），因此有必要采用实验方法。另一个例子是对图像边缘的检测，这有助于对图像的识别。边缘是指亮度的巨大变化。在*边缘检测*中，图像中的亮度变化将被增强或突出。其他的低层操作包括将像素标志为属于某一指定对象以及对象*匹配*，此时一个对象将以某种样式与一已知对象进行比较。一种简单的匹配形式是*模板匹配*（template matching），这种方法是将图像与一

个已存储的模板进行比较。

有时我们需要识别与直线和曲线有关的那些像素。完成这种操作的有效方法是使用霍夫变换。霍夫变换利用像素坐标去发现最符合要求的线方程。因此我们将霍夫变换归为低层图像处理（但不是预处理）。霍夫变换非常适合于并行实现。

在某些应用中，如果能将图像从原来的空间域转换成频率域将是非常有益的。例如，在对数字图像进行滤波时就要进行这种变换。这类变换也是对像素值进行操作，以生成一个与已数字化图像频率有关的一个新的值集。在本章末尾我们将看到这种变换。

计算需求

在开始讨论各种低层操作之前，让我们首先看看为什么有必要进行并行处理。假设一个
371 像素图有1024 × 1024个像素，对于这样的像素图以及8位的像素，需要2^{20}字节（1M字节）的存储空间，就今天的技术而言，这个需求并非是不合理的，但更关键的因素则是计算速度。假定对每个像素必须进行一次操作，那么对一帧的操作就需2^{20}次。当今的计算机速度已相当快，但即使每次操作只需10^{-8}秒/操作（10ns/操作），总计也需10ms。在实时应用中，计算速度必须达到帧速率（通常为每秒60帧~85帧）。图像中的所有像素必须在一帧的时间内，即12~16ms之内，处理完毕。通常许多高复杂性操作必须完成的不只是一次操作。这种需求对顺序计算机而言是如此迫切，导致经常需开发专用的图像处理硬件。这类专用硬件会继续用信号处理芯片加以开发。但是这类系统缺少真正的并行计算机所能提供的灵活性。此外，采用专用硬件将难于适应新的图像处理算法。

12.2 点处理

图像处理操作可分为产生的输出基于单个像素（点处理）的值的操作、产生的输出基于邻近像素组（局部操作）的值操作、产生的输出基于图像所有像素（全局操作）的值操作。点运算符不需要图像中其他像素的值，因此并行化点处理是非常简单且是完全并行的；局部操作也是可高度并行化的。点处理的例子包括阈值化（thresholding）、对比度扩展（contrast stretching）和灰度级减少（gray level reduction）。

1. 阈值化处理

在单级阈值化处理中，将保留所有具有超过预定阈值的像素，而小于该阈值的其他像素被归约为0；即对于一个给定像素x_i，对每个像素进行的操作为

$$if\ (x_i < \text{阈值})x_i = 0;\text{else}\ \ x_i = 1$$

2. 对比度扩展

在对比度扩展中，灰度级的范围被扩展到可更清楚地观察到图像细节部分。对于给定处于x_l和x_h范围中的像素值x_i，对比度通过以下公式将x_i扩展到x_L和x_H范围：

$$x_i = (x_i - x_l)\left(\frac{x_H - x_L}{x_h - x_l}\right) + x_L$$

例如在医疗图像中，通常如骨头那样的密集结构会比如肌肉或器官那样的软结构吸收多得多的入射能量（由X射线、超声或其他技术产生）。对比度扩展通常用来放大软组织部分的灰度（也许只有5%到20%）。这样做之后，具有最小密度软组织灰度值将成为可显示像素亮度的一极值，而具有有最大密度软组织灰度值将成为另一极值。对比度扩展也用于如骨头那样的密集结构以使细微裂缝那样的微小变化也能看得清楚。

3. 灰度级减小

372

在灰度级减小中，用来表示灰度级的位数将减少，也许这也将减少存储需求。实现这种减小的一个简单方法是舍弃低位有效位，但这种方法仅当图像中的灰度级的全部范围均能被很好表示时才会比较有效，且能保留足够信息。

12.3 直方图

通过建立图像的*直方图*（histogram）能够发现灰度级的变化。在阈值化处理之前，一个直方图能用来确定一个合适的阈值级。直方图的生成是一个全局操作，因为在操作中需要用到所有的像素值。事实上，直方图是非常有用的。一个图像的直方图是由指明图像中处于每个灰度级的像素数的函数所组成的。图12-2中示出了一个可能的直方图。

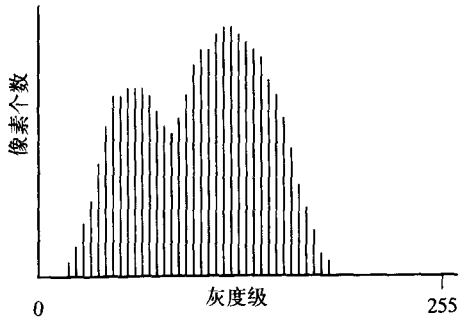


图12-2 图像直方图

生成一个直方图的顺序代码可简单地表示为：

```
for(i = 0; i < height_max; x++)
   for(j = 0; j < width_max; y++)
      hist[p[i][j]] = hist[p[i][j]] + 1;
```

其中像素被包含在数组p[][]中，而hist[k]将保存具有第*k*级灰度级的像素数。

上述的顺序代码类似于第8章中所述的将数加到一个累加和的代码，因此类似的并行求解方法可用来计算直方图。可将内层循环展开并将它们映射到各个处理器上。对于共享存储器求解方法，语句：hist[p[i][j]]=hist[p[i][j]]+1将需要放入一个临界区。一般而言，我们应使各个处理器完成各自的局部累加，此后再进行全局累加以减少对全局单元访问的延迟。对于消息传递方法，由各个独立的处理器完成部分累加，然后由一个主进程替代临界区来完成最后累加。

373

12.4 平滑、锐化和噪声消减

现在我们转向下一级的预处理操作——*平滑*（smoothing）、*锐化*（sharpening）和*噪声消减*（noise reduction）。由于这些操作均需要邻近像素值，因此它们均是局部操作。已存图像可能含有随机“噪声”或是其他不良影响的结果。*平滑*将在图像区域内抑止亮度的巨大波动，并可用减少高频成分来做到这一点。*锐化*将突出变化和增强细节，它可用两种方法加以实现：第一种方法是减少低频成分，第二种方法是通过微分突出变化。*噪声消减*将抑止图像中噪声信号。噪声信号本身可能具有各种形式，可能是与图像信号完全无关的一个随机信号。平滑将减小噪声；但也将使图像变得模糊。减小完全是随机噪声信号的一种方法是，多次捕获图像并取每一像素值的平均值。可以容易证明：随着所用图像数目增加，平均化的图像将接近于无噪声图像（假设噪声是不相关的）[Gonzalez and Woods, 1992]。这种方法一定要求每个图像处于相同位置。

12.4.1 平均值

一个简单的平滑技术是取一组像素的*平均值*（mean或average）作为中心像素的新值（一般推荐使用mean，表示两极值的中间值）。它需要对欲更新像素的周围像素组进行访问，并完

成局部操作。通常一个组的大小为3 × 3，如图12-3所示。对于一个给定的3 × 3组，其计算为:

$$x_4' = \frac{x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8}{9}$$

其中x'_4是x_4的新值。

1. 顺序代码

平均值操作需要对所有像素进行，使用像素的原始值。对每一像素计算平均值需要9步，因此n个像素需要$9n$步 (顺序时间复杂性为O (n))。

2. 并行代码

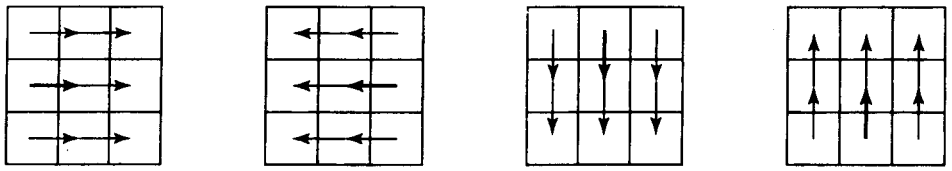
首先假定为每一个像素分配为一个处理器（不大可能是给定像素数，除了某些非常专用的图像处理硬件之外）。一种直接的并行实现将需要每个处理器完成9步操作，而所有处理器同时进行操作。由于对像素数据的访问仅是读访问（除了最后的更新以外，它是唯一的像素），对共享存储器实现而言，不会出现冲突。

x_0	x_1	x_2
x_3	x_4	x_5
x_6	x_7	x_8

图12-3　3 × 3组的像素值

374

并行方法能利用每个处理器为它相邻的处理器和自身进行部分和的计算。假定像素值是x_0、x_1、x_2、x_3、x_4、x_5、x_6、x_7和x_8，如图12-3所示，通过将计算分成四个数据传送步可减少计算步数，如图12-4所示。



第1步每个像素加上来自左方的像素　第2步每个像素加上来自右方的像素　第3步每个像素加上来自上方的像素　第4步每个像素加上来自下方的像素

图12-4　平均值计算所需的四步数据传递

每个处理器以锁步方式完成以下的四步:

第1步　每个处理器接收来自左边的像素值，并将此值加到自身的像素值上生成累加和。中心处理器（x_4）将生成$x_3 + x_4$的累加和。

第2步　每个处理器接收来自右边的像素值，并将此值加到自身的累加值上。中心处理器将生成$x_3 + x_4 + x_5$累加和。

第3步　每个处理器接收来自上边的，在第2步中生成的累加值，并将其加到自身的累加和中，为了进行下一步的计算，每个处理器必须保留原来的累加值。中心处理器生成累加和$x_0 + x_1 + x_2 + x_3 + x_4 + x_5$，与此同时，它必须将$x_3 + x_4 + x_5$分开保存。

第4步　每个处理器接收来自下方的，在第2步中生成的累加值，将其加到自身的累加值上。此时中心处理器将生成$x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$的累加和。

图12-5中示出了执行以上每一步后生成的结果。最后，每个进程用9除自身的累加和以获得它们的平均值。对于n个像素来说，总共需4步通信/加和1步除。显然，此算法也可使每个处理器处理一组像素，但此时每个处理器中的通信和运算步数将会增加（习题12-1）。

应注意的是，这类计算很自然地提示使用单指令多数据模型即SIMD，因为每个进程将同时完成相同操作。对于通用多处理机系统，可使用单程序多数据模型即SPMD。

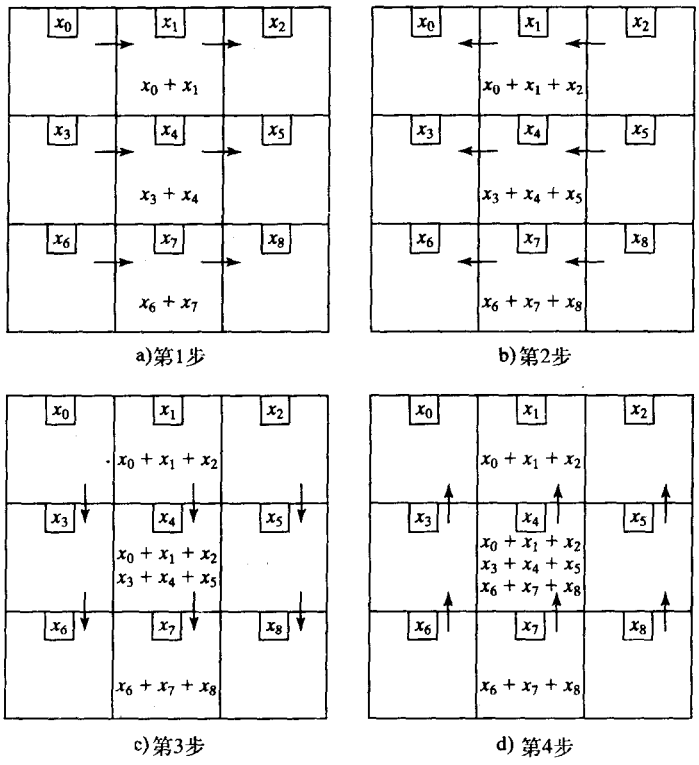


图12-5 并行的平均数据累加

12.4.2 中值

刚才描述的平均值方法会使边缘和其他锐化细节变得模糊。为减小噪声，可采用另一种方法，即用邻近像素的中值（median）来替代原来的像素值。当图像属于强“尖峰”（“spike-like”）图像时[Gonzalez and Woods，1992]，这种方法在保持边缘的锐化性方面就显得更为有效。将像素值从最小到最大按序排列，并选取位于中间的像素值（假定像素数为奇数）就可得到中值。对于3 × 3像素组，假定像素值按递升序排列为y_0、y_1、y_2、y_3、y_4、y_5、y_6、y_7和y_8，则中值便为y_4。这种操作要求组中所有值必须先排序，然后用第5个元素替代像素的原来值。采用如冒泡排序那样的顺序排序算法，将依次找到较小值，因而事实上排序在获得第5个最低值后便可终止。在这种情况下，找到每一个中值所需的步数为8 + 7 + 6 + 5 + 4 = 30，而对n个像素便需30n步。 375

并行代码

可采用10.3.1节中的扭曲排序（网格排序算法中之一）来实现并行化。扭曲排序将排序分成两个重复阶段，即排序行的内容和列的内容，并且在排序行时交替改变排序方向。为方便并行操作和提高速度可采用另一种近似的排序算法，它只需使所有行和列排序一次，且各行均以相同方向排序。在每一阶段使用有效的冒泡排序且只需三步。首先，对每一行进行比较和交换操作，它需要三步。对于第i行，我们有： 376

$$x_{i,j-1} \leftrightarrow x_{i,j}$$
$$x_{i,j} \leftrightarrow x_{i,j+1}$$
$$x_{i,j-1} \leftrightarrow x_{i,j}$$

其中$x_{i,j}$表示第i行第j列元素的值，并且↔代表“比较和交换，如果左边灰度级大于右边灰度级”（通常的冒泡排序动作是将最大值冒泡到右边）。然后，再用三次比较和交换操作对各列进行排序。对于第j列，我们有：

$$x_{i-1,j} \leftrightarrow x_{i,j}$$
$$x_{i,j} \leftrightarrow x_{i+1,j}$$
$$x_{i-1,j} \leftrightarrow x_{i,j}$$

图12-6中示出了全过程。$x_{i,j}$的值将取第5个最大像素值。该算法并不总是选取第5个最大值。例如，如果第5个最大值为x_2（在图12-3中），且它也是所在行中的最大值，则它将被保留在原来位置而不被此算法选中。尽管如此，此算法是一个合理的渐近算法，如果每个像素分配给一个处理器，则整个图像仅需6步就可完成取中值。同样，该算法可扩展为每个处理器处理一组像素。习题12-2将探讨该算法的精度。

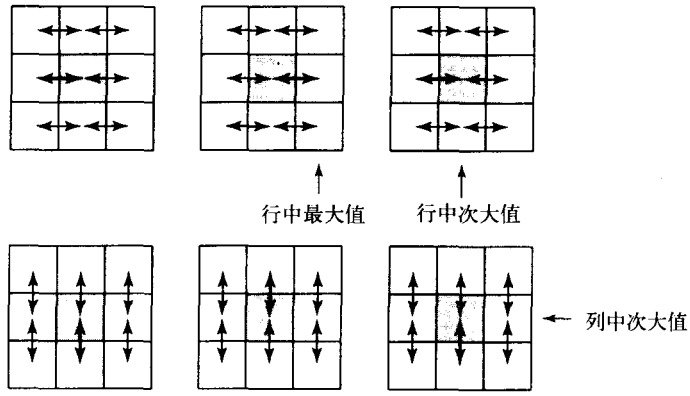


图12-6　需六步的求中值近似算法

12.4.3　加权掩码

12.4.1节的简单平均方法可由一个带权值的3 × 3掩码加以描述，该掩码描述了用于求平均
377 值的每一个像素值的总量。为叙述方便，假定权值为w_0、w_1、w_2、w_3、w_4、w_5、w_6、w_7和w_8，且像素值为x_0、x_1、x_2、x_3、x_4、x_5、x_6、x_7和x_8。新的中心像素值x'_4由下式决定：

$$x'_4 = \frac{w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7 + w_8x_8}{k}$$

其中比例因子$1/k$是在操作后用来维持正确灰度平衡用的。通常k值由$w_0 + w_1 + w_2 + w_3 + w_4 + w_5 + w_6 + w_7$确定。由$w$和$x$两函数生成的积（$w_i x_i$）的总和是$f$与$w$的（离散）互相关函数（记作$f \otimes w$）[Haralick and Shapiro，1992]。

图12-7中画出了对3 × 3掩码的操作，此3 × 3掩码操作应用到图像中的所有像素。处于图像边界的那些像素将不会有邻近像素的全集。解决这一问题的最简单的方法是将图像沿每一边扩展一个像素并对这些像素赋于一个固定值，例如赋予与它最邻近的像素同样的值，或者干脆赋予零。（注意，每个局部图像处理算法必须考虑有关边界条件。）

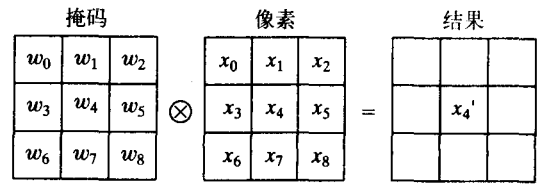


图12-7　使用3 × 3的加权掩码

虽然通常对所有图像处理操作的掩码大小为3 × 3，但也可使用其他大小，例如5 × 5、9 × 9以及11 × 11。一般总是选奇数，这样就只有一个中心像素。对于12.4.1节中的平均值操作，其所有权值为1，如图12-8所示的掩码中，它的比例因子为1/9。但是，权值不一定全要为1，也不必全相同。例如图12-9中的掩码是用来减少噪声的，此处还有许多其他可能的掩码。若使权值为负就可达到锐化效果。图12-10中示出了一种锐化掩码。用这种掩码进行的计算是：

$$x_4' = \frac{8x_4 - x_0 - x_1 - x_2 - x_3 - x_5 - x_6 - x_7 - x_8}{9}$$

$k = 9$

1	1	1
1	1	1
1	1	1

图12-8 计算平均值的掩码

$k = 16$

1	1	1
1	8	1
1	1	1

图12-9 噪声消减的掩码

$k = 9$

−1	−1	−1
−1	8	−1
−1	−1	−1

图12-10 高通锐化滤波器掩码

378

如果有关权值的某种条件满足的话，可将计算分解成先进行行操作，然后完成列操作。有关此技术的细节以及用于各种用途的许多其他掩码，可参见[Haralick and Shapiro，1992]。

12.5 边缘检测

为进行计算机识别，从图像的其他对象中区别出所需识别对象，常常需要增亮对象边缘（*边缘检测*），其中边缘是指在灰度级强度上有显著变化。

12.5.1 梯度和幅度

让我们首先考虑一维灰度级函数$f(x)$（例如沿行方向）。如果对此函数$f(x)$进行微分，则一阶求导$\partial f/\partial x$就表明了梯度变化，而在变化时将出现一个正向和负向尖端。而函数的变化方向（增加或减少方向）可由如图12-11所示的一阶导数的极性来加以标识。图12-11中还示出了二阶导数$\partial^2 f/\partial x^2$，它在转换处将穿越零点，这可能有助于识别转换的精确位置。

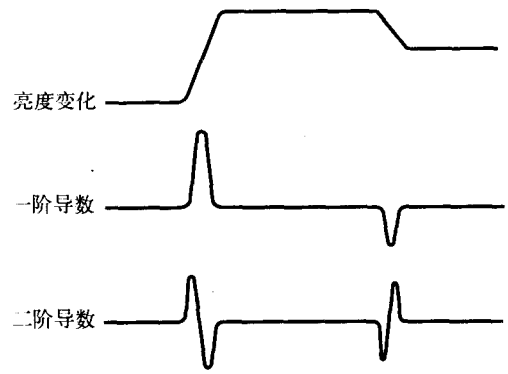


图12-11 用微分进行边缘检测

一个图像是一个二维的离散化的灰度级函数$f(x, y)$。它的灰度级改变有一个*梯度幅值*（或简称*梯度*）和一个用角度表示的*梯度方向*，这里是相对于y轴的角度，如图12-12所示。对给定的二维函数$f(x, y)$，$f(x, y)$的梯度（幅值）∇f由下式给定：

379

$$\nabla f = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

而梯度方向用角度给定：

$$\phi(x, y) = \tan^{-1}\left(\frac{\frac{\partial f}{\partial y}}{\frac{\partial f}{\partial x}}\right)$$

其中，ϕ是与y轴的夹角（见图12-12）。梯度可近似地表示成：

$$\nabla f \approx \left|\frac{\partial f}{\partial y}\right| + \left|\frac{\partial f}{\partial x}\right|$$

以减少所需计算量。

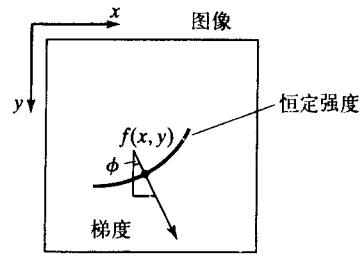


图12-12　灰度级梯度和方向

12.5.2　边缘检测掩码

对于离散函数，导数可近似地用差分来代替。使用带权掩码可用若干种方法数字地实现求导。$\partial f/\partial x$项是指x方向差分，而$\partial f/\partial y$为y方向差分。因此我们就可取行中相邻像素灰度级之差分和列中相邻像素之差分。假定有一个如图12-3中所示的3 × 3像素值的集。我们可考虑用像素x_5和x_3的值来计算渐近梯度以得到$\partial f/\partial x$，用x_7和x_1以得到$\partial f/\partial y$，即：

$$\frac{\partial f}{\partial x} \approx x_5 - x_3$$

$$\frac{\partial f}{\partial y} \approx x_7 - x_1$$

380

从而有

$$\nabla f \approx |x_7 - x_1| + |x_5 - x_3|$$

需要使用两个掩码，一个用来得到$x_7 - x_1$，而另一个用来获得$x_5 - x_3$。每一个掩码的结果的绝对值将加在一起。可使用负的权值来实现减法。（为进行求导，未示出任何比例因子，但若要获取合适对比度，就需要选择比例因子）。

1. Prewitt算子

使用更多的像素值就可获得更好的结果。例如，近似梯度可由下式得到：

$$\frac{\partial f}{\partial y} \approx (x_6 - x_0) + (x_7 - x_1) + (x_8 - x_2)$$

$$\frac{\partial f}{\partial x} \approx (x_2 - x_0) + (x_5 - x_3) + (x_8 - x_6)$$

然后可得

$$\nabla f \approx |x_6 - x_0 + x_7 - x_1 + x_8 - x_2| + |x_2 - x_0 + x_5 - x_3 + x_8 - x_6|$$

它需要使用两个称为Prewitt算子的3 × 3掩码，如图12-13所示。同样，每个掩码的结果绝对值需加在一起。

−1	−1	−1
0	0	0
1	1	1

−1	0	1
−1	0	1
−1	0	1

图12-13　Prewitt算子

2. Sobel算子

这是一个非常流行的边缘检测算子。它使用如图12-14所示的两种掩码。这里，导数可渐近地表示为：

$$\frac{\partial f}{\partial y} \approx (x_6 + 2x_7 + x_8) - (x_0 + 2x_1 + x_2)$$

$$\frac{\partial f}{\partial x} \approx (x_2 + 2x_5 + x_8) - (x_0 + 2x_3 + x_6)$$

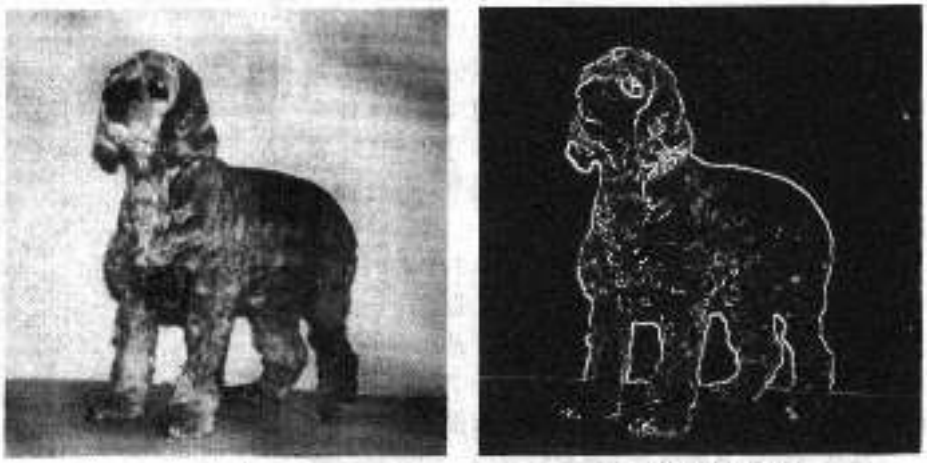
−1	−2	−1
0	0	0
1	2	1

−1	0	1
−2	0	2
−1	0	1

图12-14　Sobel 算子

381　实现一阶导数的算子将增强噪声。然而，Sobel算子同时具有平滑

作用。由Sobel的两个掩码算子可算得$\partial f/\partial x$和$\partial f/\partial y$，并可方便地计算梯度∇f和梯度方向ϕ。图12-15示出了一个图像和Sobel算子的效果。



a）原图像　　b）Sobel算子的效果

图12-15 用Sobel算子进行边缘检测

3. 两种掩码并行代码

因为两种掩码的结果相互独立，因此只要提供充分资源两者就可同时完成。一种并行实现方法是，对每一种掩码进行如图12-5中所示的四步操作，不同之处是在进行加法时要带上权值。

4. 拉普拉斯算子

代之以一阶导数，可使用二阶求导进行边缘检测，因为边缘可导致一个可识别的输出（一个正向和一个负向脉冲，见图12-11）。拉普拉斯二阶导数的定义为：

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

可使用多种差分方法逼近地求解上式。它可近似地表示为：

$$\nabla^2 f = 4x_4 - (x_1 + x_3 + x_5 + x_7)$$

而这可以借助图12-16所示的单掩码得到。中心像素x_4可使用四个相邻像素加以更新，图12-17中对此做了说明（$x'_4 = 4x_4 - x_1 - x_3 - x_5 - x_7$）。应注意的是，与掩码相交叉的固定灰度级的计算，其计算结果将会是零。通常在拉普拉斯算子（以及所有边缘检测算子）作用后，需再加上一个阈值化操作以获得黑或白的像素。图12-18中示出了拉普拉斯算子作用于图12-11中原图像后的效果（无噪声）。不幸的是，拉普拉斯算子通常在边缘检测时对噪声过于敏感。在拉普拉斯算子后再施以阈值算子，往往可以改进单用拉普拉斯算子进行边缘检测的效果（见习题12-7）。事实上，阈值化通常能使所有边缘检测方法得到改进。

382

0	−1	0
−1	4	−1
0	−1	0

图12-16 拉普拉斯算子



图12-17 在拉普拉斯算子中使用的像素

12.6 霍夫变换

霍夫变换[Hough，1962]的目的是要寻找能最好适合于图像中像素集的线方程的参数。霍夫变换是模板匹配的基础，它也是将一个对象转换成可供识别的向量集或作其他用途的基础。从理论上边缘检测将勾划出对象的轮廓，但到目前为止我们所叙述的边缘检测可能在边缘中留下某些间隙，而霍夫变换可用来填补这些间隙。霍夫变换将对图像中的所有像素作用，因而它是全局操作而非前几节所遇到的局部操作。



图12-18　拉普拉斯算子的效果

一条线可由方程$y = ax + b$加以描述，其中参数a和b唯一地描述了一条特定线，a为斜率，b为与y轴的截距。此方
383 程便是通常的斜率-截距形式，一个像素可有无限多条线经过它。如果a和b是离散化的，则便只存在有限条线经过该像素。可以寻找所有点的所有线，且可能找到图像中最可能的那些线，这些线会使最多的像素被映射到它们，但这种寻找在计算上是非常费时的[O (n^3) 算法]。

然而，我们可将线方程改写为：

$$b = -xa + y$$

图12-19中画出了原 (x, y) 平面中的线以及以参数 (a, b) 为坐标的称为参数空间中的线。(为清晰起见，我们仍回到通常的x-y坐标，而不是原点在左上角的坐标)。在参数空间中，线由单点表示，这意味着处于x-y空间中指定线上的每一点（即对于指定的a和b的值）都将映射到参数空间中的同一点，所以当将线映射到a-b空间中的一点上时，我们可以借助简单的计数来找到x-y空间中处于同一线上的点数。

下面让我们来看看如何实现这一方法。在原x-y空间中会有许多线通过单点，而每一条线将映射到a-b空间中的不同点，它只受a和b的精度约束。事实上，单点 (x_1, y_1) 可映射到a-b空间中线$b = -x_1a + y_1$的各个点上。在映射过程中，将使用离散值作为粗略的规定精度，而计算将被舍入到最接近可能的a-b坐标。例如，每个a和b可被分成100个值，对于每一个 (a, b) 值有相应的一个累加器或计数器。对于x-y空间中的每一点都要进行这种映射处理，将相应的累加器不断增量，就可记录下那些已得到的a-b点。因此，每一个累加器将有映射到参数空间中单点的像素数，从而使我们可确定在x-y空间中每一条可能线上的像素数。最后，我们选择在参数空间中具有局部最大像素数的那些点作为线，从而可找到x-y空间中最可能的那些线。

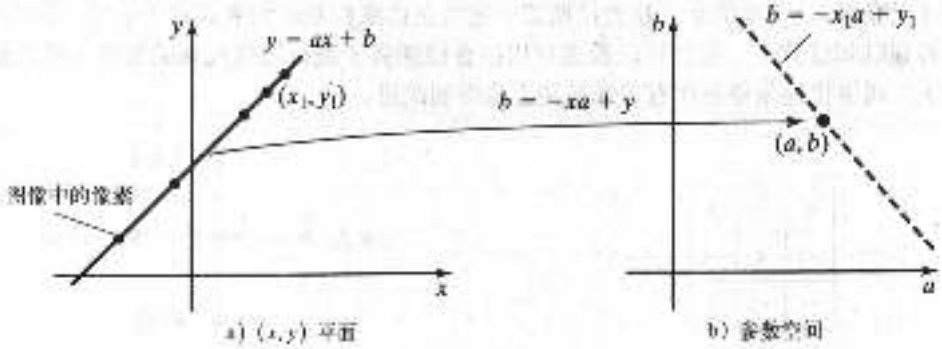


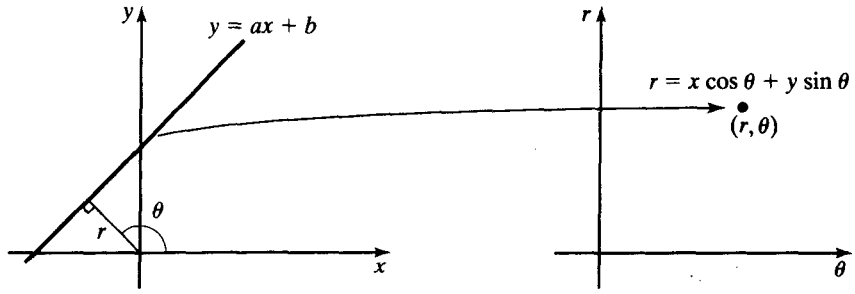
图12-19　将线映射到 (a, b) 空间

不幸的是，这一方法对于垂直线（即斜率a为无限大，以及与y轴截距b为无限大）以及那些接近垂直线的线是无效的。为避免这一情况，[Duda and Hart，1972]提议，应将线方程转换成极坐标（标准表达式）形式：

384

$$r = x\cos\theta + y\sin\theta$$

其中r是原（x，y）坐标系中原点到线的垂直距离，而θ是r和x轴的夹角，如图12-20所示。θ值将以度来表示，且它也是该线的梯度角（相对于x轴）。每一条线将映射到（r，θ）空间中的单点。而如果与x轴有正截距，一条垂线将简单地映射到一点，且$\theta = 0°$，而如果与x轴有负截距，则θ就等于180°。一条水平线的$\theta = 90°$。x-y空间中的每一点将映射到（r，θ）空间中的一条曲线上，即（x_1，y_1）映射到$r = x_1\cos\theta + y_1\sin\theta$。



a）(x, y) 平面　　b）(r, θ) 平面

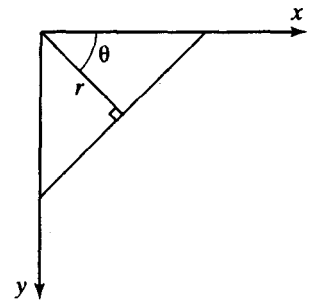
图12-20 将一条线映射到（r，θ）空间

应指出的是，不一定非要使用极坐标标准表达式，例如，[Krishnaswamy and Banerjee，1997]在他们的论著中使用通常的斜率-截距方程形式，但为了避免出现无限大值的可能性，将斜率分成四个区，0到π/4、π/4到π/2、π/2到3π/4、和3π/4到π。对于斜率在0到π/4范围内的那些线，处理按常规进行，而对处于其他区域的线，则坐标要作相应改变。更多细节请参见[Krishnaswamy and Banerjee，1997]。

1. 实现

图12-21中示出了原点在左上角的图像坐标系的标准表示形式。注意其中的x和y值只能为正，参数r也限制取正值。θ的取值范围为0°到360°，尽管在图像坐标系中θ不会处于180°到270°之间。r的取值范围将依赖于原来的x和y值的取值范围。

参数空间被分成许多小的矩形区域。每个区域有一个累加器。区域数的多少取决于所期望的离散化精度。如果r被分成5的增量和θ被分成10°增量，则每一区域为10° × 5，且有一个累加器，称为acc[r][θ]，如图12-22所示。被一个像素映射到的那些区域的累加器将加1。这一过程必须对图像中的所有像素进行。计算效果将取决于区域数。如果所有θ值均试验的话（即增量θ值取它的所有可取值），计算量将由θ取离散值数目即k间隔所确定。对于n个像素，其复杂性为O (kn)。对于固定的k，虽然本质上这是一个线性复杂性，但仍值得减小其复杂性。



385

图12-21 图像坐标系的通常表示

采用某种准则以限制各个像素线的取值范围，就可使计算量显著减小。可以根据线的梯度来选择θ的单一值。而梯度角可用诸如Sobel算子（12.5.2节）那样的梯度算子来找到。一旦得到此值，就只需增量所找到的那个累加器。由于此时对每个像素只需一个计算步，因而对

于n个像素，其复杂性就变成O (n)。

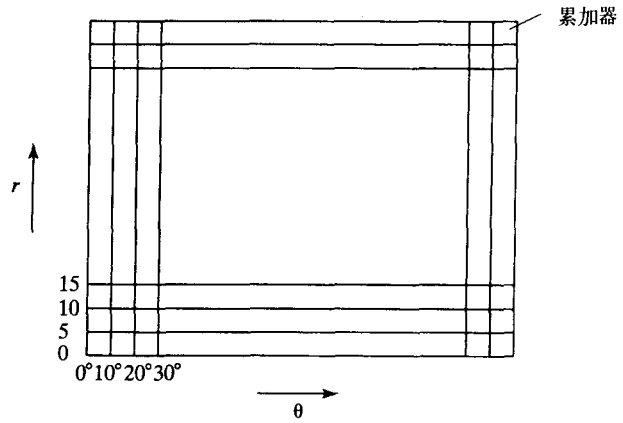


图12-22　霍夫变换使用的累加器acc[r][θ]

通常我们希望知道线的端点，通过记录实际映射到该线上像素就可找到端点。细察该表就可找到该线上最远的那些像素。

2. 顺序代码

顺序代码可为如下形式:

```
for (x = 0; x < xmax; x++)                    /* for each pixel */
   for (y = 0; y < ymax; y++) {
      sobel(x, y, dx, dy);                    /* find x and y gradients */
      magnitude = grad_mag(dx, dy);           /* find magnitude if needed */
      if (magnitude > threshold) {
         theta = grad_dir(dx, dy);            /* atan2() fn */
         theta = theta_quantize(theta);
         r = x * cos(theta) + y * sin(theta);
         r = r_quantize(r);
         acc[r][theta]++;                     /* increment accumulator */
         append(r, theta, x, y);              /* append point to line */
      }
   }
```

在上述代码的最后，当所有像素均被考虑后，在每个累加器中保留的值便是表示能映射到相应线上的像素的数目。具有最多像素的那些线是最可能的那些线，因此要选择的是具有最大局部值的累加器。为此必须设计选择局部最大值的算法。

3. 并行代码

显然，前面的顺序代码具有很大并行化的潜力。因为每个累加器的计算是与其他累加操作相互独立的，因此它们可以同时进行计算，尽管每个计算需要对整个图像进行读访问。采用共享存储器的实现将不需要临界区，因为访问全为读访问，但由于对单元进行同时的读请求，因此会有一定的延迟和竞争。

386

12.7　向频域的变换

一个周期（时间）函数x (t)可分解成一系列具有不同频率和幅度的正弦波。由于这种分解是由傅里叶在19世纪早期提出的，因而被称为傅里叶级数。傅里叶变换为原函数x (t)产生一个

连续的频率函数$X(f)$。傅里叶变换在科学和工程中有许多应用，包括数字信号处理和图像处理。这里感兴趣的主要是图像处理，然而下面所述的方法将适用于所有傅里叶变换的应用领域。在图像处理领域内，傅里叶变换（其中特别是实现傅里叶变换的快速算法，即快速傅里叶变换）主要是用作图像的增强、恢复和压缩。

图像是一个二维离散函数$f(x, y)$，但让我们先来看看一维的连续实例，如稍后将要讲到的，二维实例可通过将一维实例应用到两个维上便可解决。为完整起见，让我们从最基本原理出发，来回顾一下傅里叶级数和傅里叶变换的概念。有关方程的推导，包括数学的严格性，可在[Elliott and Rao，1982]以及许多其他有关傅里叶级数和变换的教科书中找到。

12.7.1 傅里叶级数

傅里叶级数是一串正弦和余弦项的累加和，它可写为:

$$x(t)=\frac{a_0}{2}+\sum_{j=1}^{\infty}\left(a_j\cos\left(\frac{s\pi jt}{T}\right)+b_j\sin\left(\frac{2\pi jt}{T}\right)\right)$$

式中T为周期（$1/T=f$, f为频率）。傅里叶系数a_j和b_j可由定积分求得。经过相应的数学处理后，可得到此级数更方便的表达式:

$$x(t)=\sum_{j=-\infty}^{\infty}X_j e^{2\pi ij\frac{t}{T}}$$

其中，X_j是复数形式的第j个傅里叶系数，而$i=\sqrt{-1}$。复数形式的傅里叶系数也可由定积分求得。 387

12.7.2 傅里叶变换

1. 连续函数

稍作数学上的处理，前面的求和表达式可演变成下面的积分:

$$x(t)=\int_{-\infty}^{\infty}X(f)e^{2\pi if}df$$

其中，$X(f)$是频率的连续函数。函数$X(f)$可由下式得到:

$$X(f)=\int_{-\infty}^{\infty}x(t)e^{-2\pi ift}dt$$

式中$X(f)$是$x(t)$的*频谱*，或更简单地是$x(t)$的*傅里叶变换*。代之以傅里叶级数中的离散频率，现在我们得到的是一个连续的频率域，甚至具有无限周期的频率（以及负频率！）。有关细节请参见[Elliott and Rao，1982]。

使用给定的一次积分可由$X(f)$推得原函数$x(t)$，这种积分被称为*反向傅里叶变换*，它在形式上类似于傅里叶变换。因此，能实现傅里叶变换的任何算法也适用于反向傅里叶变换。

2. 离散函数

要用数字计算机来实现傅里叶变换，必须对输入函数$x(t)$加以采样，并将它们以一组离散值存储起来，例如对于N次采样有离散值组x_0，x_1，x_2，…，x_{N-1}。可将前述的傅里叶变换用求和方式替代积分方式，就可演变成有N个离散值的集合，从而变成*离散傅里叶变换*（Discrete Fourier Transform，DFT），由下式给定:

$$X_k = \frac{1}{N}\sum_{j=0}^{N-1} x_j e^{-2\pi i\left(\frac{jk}{N}\right)}$$

相应的反向离散傅里叶变换为:

$$x_k = \sum_{j=0}^{N-1} X_j e^{2\pi i\left(\frac{jk}{N}\right)}$$

388 其中$0 \leqslant k \leqslant N-1$。$N$个（实数）输入值$x_0$，$x_1$，$x_2$，…，$x_{N-1}$，将产生$N$个（复数）变换值$X_0$，$X_1$，$X_2$，…，$X_{N-1}$。

（1）比例因子　N是一个比例因子。因为在大多数应用中，一个函数将被转换、处理并返回到它的原来形式。在进行变换或反向变换时，可求得比例因子（或在每个变换中使用$\sqrt{N}$）。有时，离散傅里叶变换以不带比例因子的显式方式表示，即:

$$X_k = \sum_{j=0}^{N-1} x_j e^{-2\pi i\left(\frac{jk}{N}\right)}$$

除了12.7.5节中的快速傅里叶变换外，我们将省略该比例因子。

（2）正、负系数　如像[Cochran et al., 1967]所指出的那样，除了改变比例因子的使用之外，某些论著作者在变换时使用正系数，而在反变换时使用负系数，而不像这里在变换时使用负系数，而在反向变换时使用正系数（注意:　$e^{i\theta} = \cos\theta + i\sin\theta$以及$e^{-i\theta} = \cos\theta - i\sin\theta$。）

（3）时间复杂性　求和计算比较容易，特别是如果指数项的值存放在查找表中时，但对N点而言（如方程式所列出的那样）仍需N^2次乘法和加法，即其顺序复杂性为$O(N^2)$。这种顺序复杂性一般被认为是难于接受的，特别是N取较大值时。幸运的是，已开发了称为快速傅里叶变换算法，它将复杂性降为$O(N\log N)$。12.7.5节中将叙述这一算法，但在此之前，让我们先来看一下离散傅里叶变换的图像处理应用。

12.7.3　图像处理中的傅里叶变换

在图像处理中，输入将是形成离散二维函数的一组像素。在本节中，我们将使用j和k坐标（而不是早先使用的x和y坐标）。在(j, k)坐标中的像素是$x(j, k)$。

二维的傅里叶变换为:

$$X_{lm} = \sum_{j=0}^{N-1}\sum_{k=0}^{M-1} x_{jk} e^{-2\pi i\left(\frac{jl}{N}+\frac{km}{M}\right)}$$

其中j和k是行和列的坐标，且$0 \leqslant j \leqslant N-1$和$0 \leqslant k \leqslant M-1$。为方便起见，让我们假定该图像是一
389 个方阵，即$N = M$。这样此方程可重写为:

$$x_{lm} = \sum_{j=0}^{N-1}\left[\sum_{k=0}^{N-1} x_{jk} e^{-2\pi i\left(\frac{km}{N}\right)}\right] e^{-2\pi i\left(\frac{jl}{N}\right)}$$

式中的内部累加和是作用于一行中N个点上的一维DFT操作，它将生成一个已变换的行；而外部累加和是作用于一列中N个点上的一维DFT操作。我们可写:

$$X_{lm} = \sum_{j=0}^{N-1} X_{jm} e^{-2\pi i\left(\frac{jl}{N}\right)}$$

因此，二维的DFT能被分为两个顺序阶段，一个阶段作用于行元素而另一阶段作用于（已变

换）列元素，如图12-23中所示。因此需要实现的仅是一维DFT算法。很显然，此过程也可从列开始，然后再是行。具体选择则依赖于为实现有效并行实现原始数据是如何存放的[Fox et al.，1988]。由于行变换相互独立，且列变换也相互独立，因而存在有很多的并行化机会。

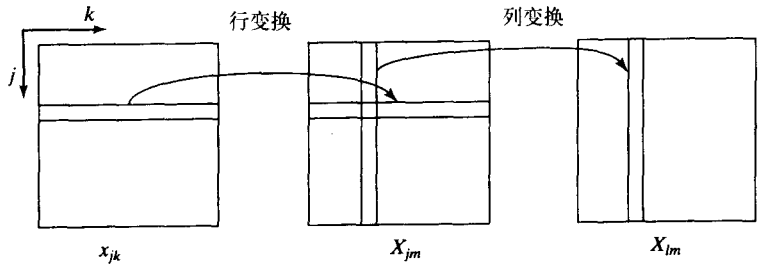


图12-23 二维DFT

图12-23中示出了二维DFT的一种实现，先进行一维DFT操作，然后进行转置操作，接着再作一次相同的DFT操作。（在4.2.1节中，首次介绍了转置操作，它可用全部到全部（all-to-all）例程加以实现）。

应用

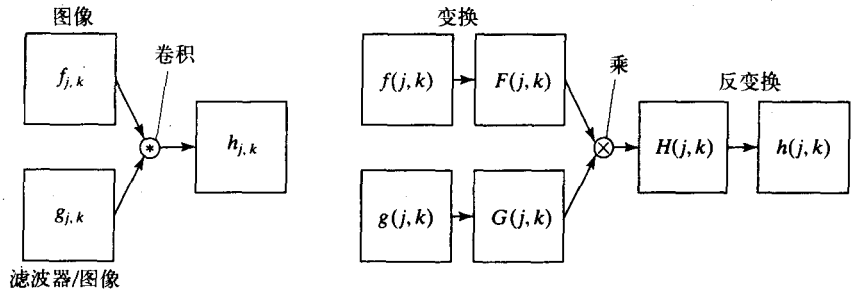
在频率域内，图像处理和分析有非常广泛的应用。DFT的一个应用领域是频率滤波，在平滑和边缘检测中都要用到它（低通和高通滤波器）。频率滤波早先采用加权掩码，它可用卷积操作描述：

$$h(j,k) = g(j,k) * f(j,k)$$

（与对称掩码的互相关操作相同；[Haralick and Shapiro，1992]，式中$g(j, k)$描述了加权掩码（滤波器），而$f(j, k)$描述了图像。可以证明函数积的傅里叶变换可由各函数变换的卷积确定（故称频率卷积理论；[Brigham，1988]）。因此，两个函数的卷积可将每个函数先进行傅里叶变换，然后将两个变换相乘得到

$$H(j,k) = G(j,k) \times F(j,k)$$

（元素与元素相乘），式中$F(j, k)$是$f(j, k)$的傅里叶变换，而$G(j, k)$是$g(j, k)$的傅里叶变换。 390
此后对其取反变换，就可使结果回到原来的空间域。这种滤波方法比起在空间域中使用简单的加权掩码方法需要更多的计算量，但它可完成其他更复杂的操作。也可对两个完整图像进行卷积以产生一个新的图像，如图12-24所示。应注意的是，由于两者的变换是相互独立的，因此它们可并行地完成。



a) 直接卷积　　b) 使用傅里叶变换

图12-24 使用傅里叶变换的卷积

12.7.4 离散傅里叶变换算法的并行化

在叙述更聪明、更快的傅里叶变换的算法之前，让我们首先研究基本的DFT算法以及它的并行化方法。我们从以下公式开始:

$$X_k = \sum_{j=0}^{N-1} x_j e^{-2\pi i\left(\frac{jk}{N}\right)}$$

在使用记号 $w = e^{-2\pi i/N}$ 后，可得:

$$X_k = \sum_{j=0}^{N-1} x_j w^{jk}$$

式中w项被称为*旋动系数*（twiddle factor)。每个输入值必须乘以旋动系数。用w^{-1}替代w就可得到反变换。

1. 顺序代码

产生全部N点的DFT的顺序代码形式可为:

```
for (k = 0; k < N; k++) {                    /* for every point */
   X[k] = 0;
   for (j = 0; j < N; j++)                   /* compute summation */
      X[k] = X[k] + w^(j*k) * x[j];
}
```

391

其中X[k]为第k个被变换点，x[k]为第k个输入，共有N个输入点，而$w = e^{-2\pi i/N}$。求累加和的计算步要进行复数运算。由于每步求累加和要使用前一步的w的升幂值并乘以w^k（即$w^{(j-1)*k} * w^k = w^{j*k}$)，故上述代码可重写成:

```
for (k = 0; k < N; k++) {
   X[k] = 0;
   a = 1;
   for (j = 0; j < N; j++) {
      X[k] = X[k] + a * x[j];
      a = a * w^k;
   }
}
```

其中a是临时变量。

2. 并行代码　可采用几种方法使上述代码并行化。这里我们将只简单涉及最明显的主从方法、较明显的流水方法以及在最后叙述的矩阵-向量乘积方法。

（1）*基本的主-从实现*　在*主-从方法*中，N个从进程中的每一个可被指定来产生一个变换值，即第k个从进程生成X[k]。所需的a值可由主进程预先求得，如图12-25所示，尽管我们也可使每个从进程同时计算它们各自的值。该方法要求每个从进程拥有一份所有输入点的拷贝，从而会使存储需求增加到N倍。（每次向主进程请求所需元素，但这将显著增加消息传递。）N（从进程）个进程并行求解的时间复杂性为O(N)，相对于顺序算法实现的O(N^2)该并行方法有了很大的优化。不幸的是，

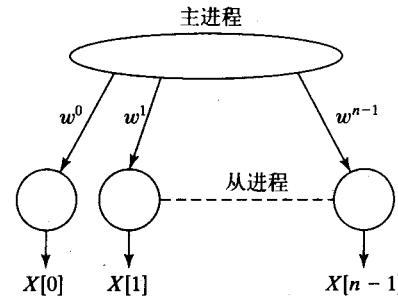


图12-25　直接实现DFT的主从方法

数据点数很可能远大于可用的进程数。此时，每个从进程将需要进行多次累加和。

（2）流水线实现 该算法可采用流水线结构加以实现，因为内层循环中的每次迭代需要使用前一次迭代生成的值。将X[h]的内层循环展开可得:

```
X[k] = 0;
a = 1;
X[k] = X[k] + a * x[0];
a = a * w^k;
X[k] = X[k] + a * x[1];
a = a * w^k;
X[k] = X[k] + a * x[2];
a = a * w^k;
X[k] = X[k] + a * x[3];
a = a * w^k;
  ⋮
```

392

其中的每一对语句

```
X[k] = X[k] + a * x[0];
a = a * w^k;
```

可由一个独立的流水级完成，如图12-26所示。流水线及它的时序则如图12-27所示。

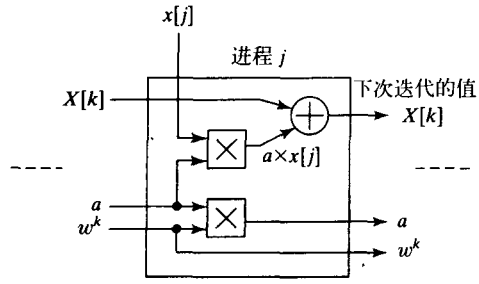
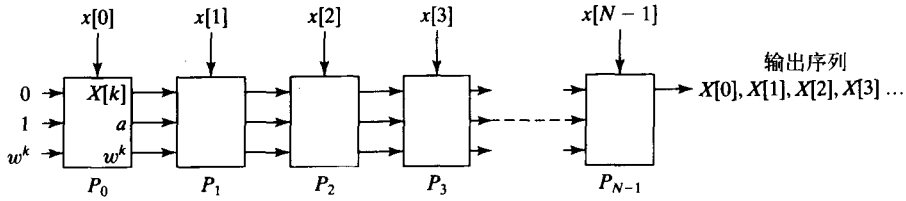
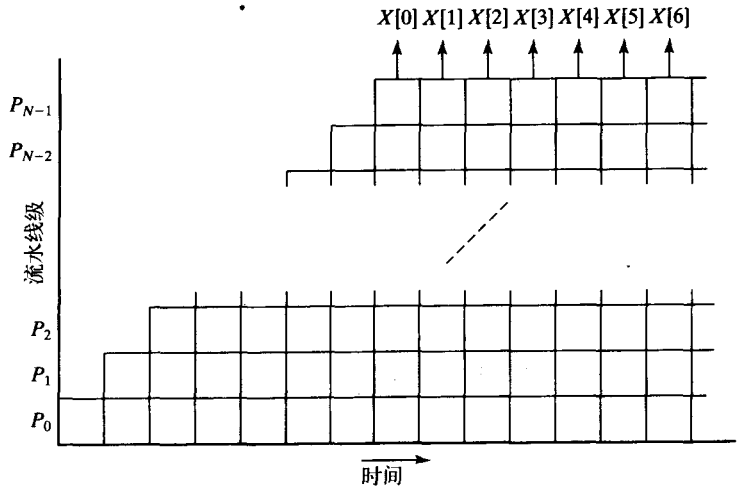


图12-26 DFT算法流水线实现的一个流水线级



a) 流水线结构



b) 时序图

图12-27 用流水线进行离散傅里叶变换

除了我们的方法外，[Thompson，1983]还叙述了其他的几种流水线计算的方法。

（3）由矩阵-向量积实现DFT　离散傅里叶变换的第k个元素可由下式给定：

393

$$X_k = x_0 w^0 + x_1 w^1 + x_2 w^2 + x_3 w^3 + \cdots + x_{N-1} w^{N-1}$$

而整个变换可由矩阵-向量乘积表示：

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_k \\ \vdots \\ X_{N-1} \end{bmatrix} = \frac{1}{N} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & w^3 & \cdots & w^{N-1} \\ 1 & w^2 & w^4 & w^6 & \cdots & w^{2(N-1)} \\ 1 & w^3 & w^6 & w^9 & \cdots & w^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & w^k & w^{2k} & w^{3k} & \cdots & w^{(N-1)k} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & w^{N-1} & w^{2(N-1)} & w^{3(N-1)} & \cdots & w^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \\ \vdots \\ x_{N-1} \end{bmatrix}$$

（注意$w^0 = 1$）。因此在第11章所述的产生矩阵-向量积的那些并行方法可用来进行离散傅里叶

394 变换，w项的某些可以归约掉，这将在习题12-15中论及。

12.7.5 快速傅里叶变换

快速傅里叶变换（Fast Fourier Transform，FFT）是获取离散傅里叶变换的快速算法，它可使时间复杂性从$O(N^2)$降为$O(N\log N)$。通常将发明FFT归功于[Cooley and Tukey，1965]，虽然后来的相关信息（在文献中已指出）表明事实上FFT的基本思想是相当古老的，它可一直追溯到20世纪初期（[Cooley，Lewis and Welch，1967]；[Gonzalez and Woods，1992]）。在下面的叙述中，将假设N是2的乘方。

让我们以具有比例因子的离散傅里叶变换方程开始进行讨论，以表明比例因子不会影响算法的推导：

$$X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j w^{jk}$$

式中$w = e^{-2\pi i/N}$。快速傅里叶变换有许多公式。例如，[Swarztrauber，1987]就叙述了8种公式。一般而言，使用分治方法将求和不断地分解。下面我们描述的一个公式是将求和分解成以下两部分来进行的：

$$X_k = \frac{1}{N} \left[\sum_{j=0}^{(N/2)-1} x_{2j} w^{2jk} + \sum_{j=0}^{(N/2)-1} x_{2j+1} w^{(2j+1)k} \right]$$

其中第一项求和处理具有偶下标的x值，而第二项求和处理具有奇下标的x值。重写后可得：

$$X_k = \frac{1}{2} \left[\frac{1}{(N/2)} \sum_{j=0}^{(N/2)-1} x_{2j} w^{2jk} + w^k \frac{1}{(N/2)} \sum_{j=0}^{(N/2)-1} x_{2j+1} w^{2jk} \right]$$

或是

$$X_k = \frac{1}{2} \left[\frac{1}{(N/2)} \sum_{j=0}^{(N/2)-1} x_{2j} e^{-2\pi i \left(\frac{jk}{N/2}\right)} + w^k \frac{1}{(N/2)} \sum_{j=0}^{(N/2)-1} x_{2j+1} e^{-2\pi i \left(\frac{jk}{N/2}\right)} \right]$$

现在可将每个和看成是一个$N/2$离散傅里叶变换分别作用于$N/2$偶数点和$N/2$奇数点。因此有：

$$X_k = \frac{1}{2} \left[X_{\text{偶}} + w^k X_{\text{奇}} \right]$$

其中$k = 0, 1, \cdots, N-1$，$X_{偶}$是具有偶下标数$x_0, x_2, x_4, \cdots$的$N/2$点的DFT；而$X_{奇}$则是具有奇下标数$x_1, x_3, x_5, \cdots$的$N/2$点的DFT。

现在，若假设k限制为0，1，…，$N/2-1$，即全部N个值的前$N/2$个值。这样整个序列可分成为两部分：

$$X_k = \frac{1}{2}\left[X_{偶} + w^k X_{奇}\right]$$

395

$$X_{k+\frac{N}{2}} = \frac{1}{2}\left[X_{偶} + w^{k+N/2} X_{奇}\right] = \frac{1}{2}\left[X_{偶} - w^k X_{奇}\right]$$

因为$w^{k+N/2} = -w^k$，其中$0 \leqslant k < N/2$。这样我们可用两个$N/2$点变换来计算X_k和$X_{k+N/2}$，如图12-28所示。

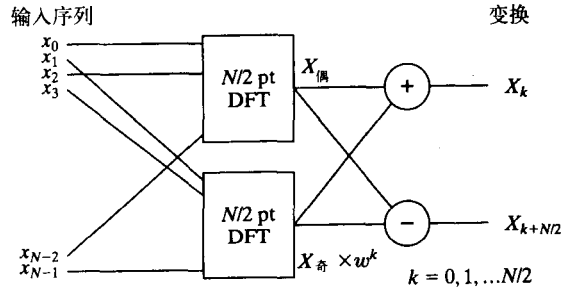
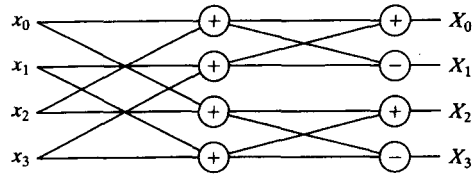


图12-28 将N个点的DFT分解成两个$N/2$个点的DFT

以上每一个$N/2$点的DFT又可分解成两个$N/4$点的DFT，此分解可一直进行下去直到对单点进行变换。一个单点的DFT即是该点的值。对每个要变换的点都要进行上述的计算，图12-29示出了以这种方法对四点进行的变换。图12-29中未画出旋动系数，但应注意到，根据$X_{偶}$和$X_{奇}$中所使用的数的个数以及指定的X变换输出下标，在不同级上出现的不同值。注意到由于数的个数以2倍系数减小，相应地w的幂以2倍系数增加，从而方便了旋动系数的获得。($w = e^{2\pi i/(N/2)}$，$e^{2\pi i/(N/4)} = w^2$，$e^{2\pi i/(N/8)} = w^4$等)。为清晰起见，1/2比例因子也已被省略，从而在原来的DFT方程中若没有比例因子，则它就不会存在。



12-29 四个点离散傅里叶变换

图12-30中示出了一个16点DFT的分解。应注意的是，初始化时，元素以*逆向位序选择*，即下标的二进制表达式是与通常的数序相反的；0和8组合在一起（0000和1000，而不是0000和0001，等等）。图12-31中给出了16点FFT的结果流。

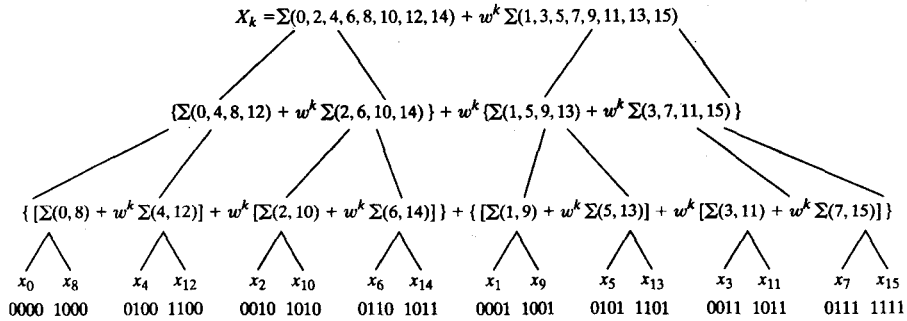


图12-30 16个点DFT的分解

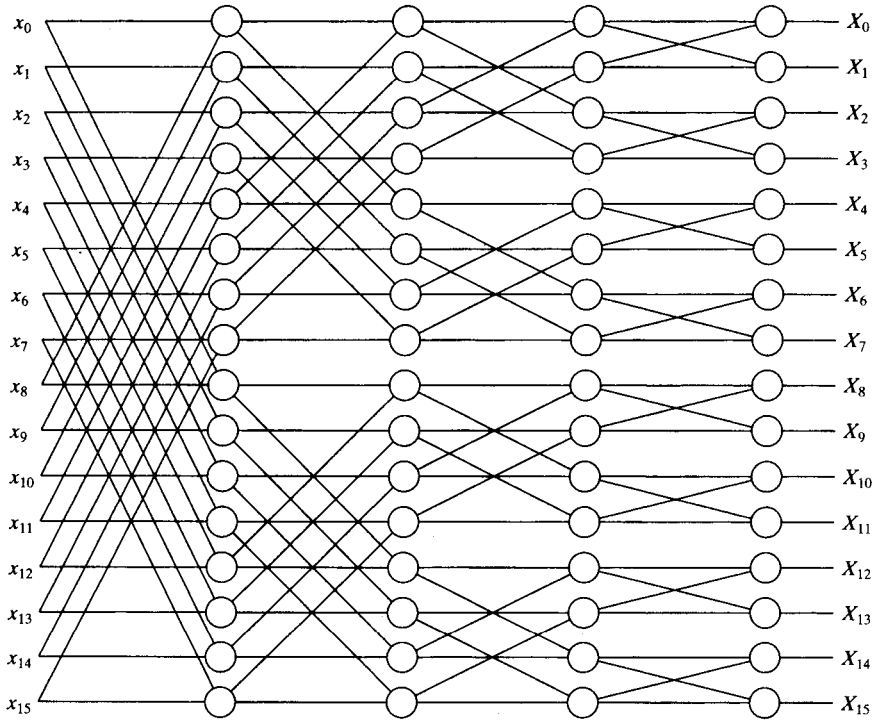


图12-31　16个点FFT的计算流

1. 顺序代码

顺序计算的时间复杂性基本上是O(Nlog$N$)，因为共有log$N$步，而每一步需要进行正比于N的计算，这里的N是指共有N个数。该算法可用递归或迭代方法加以实现。

2. FFT算法的并行化

396
~
397

由于FFT的顺序计算的时间复杂性为O(Nlog$N$)，因而当使用N个处理器时，理想的成本优化并行计算时间复杂性应为O(log$N$)。下面我们叙述两个并行化的方法。第一个称为*二元交换算法*，由[Gupta and Kumar，1993]提出，而第二个则在行列操作之间使用转置操作。

二元交换算法　在图12-31中，假定为每个数据点（对第j个进程为$x[j]$，相应于图12-31中的一行）分配一个处理器。每个进程最终将生成一个变换点。图12-31中的连接模式称为*蝶形连接*，如果每一行分配一个处理器，则它可很好地映射到超立方体上，这是因为本级产生的结果将传送给下一级的进程，而该进程所拥有的地址位仅与本级的地址有一位不相同。例如，在第一个通信步中，处理器0将与处理器8通信，而在下一步中与处理器4通信，再下一步与处理器2通信，最后一步与处理器0通信。

同样地，如果处理器数小于数据点数，从而每个处理器将分配有一组数据点，但此时的处理器间通信模式仍具有相同特征。假定有p个处理器和N个数据点，则每个处理器有N/p行。如果N和p均为2的乘方，则下面处理器的编号取数据点下标中log$p$最高位。余下的位则用来标识组内的数据点。图12-32中示出了N/p = 4的情况。

3. 分析

（1）*计算*　给定p个处理器和N个数据点，每个处理器在每一步将计算N/p个点，而每一个点的计算需进行一次乘法和一次加法。因共需log$N$步，故并行计算的时间复杂性由下式确定：

$$t_{comp} = O(N \log N)$$

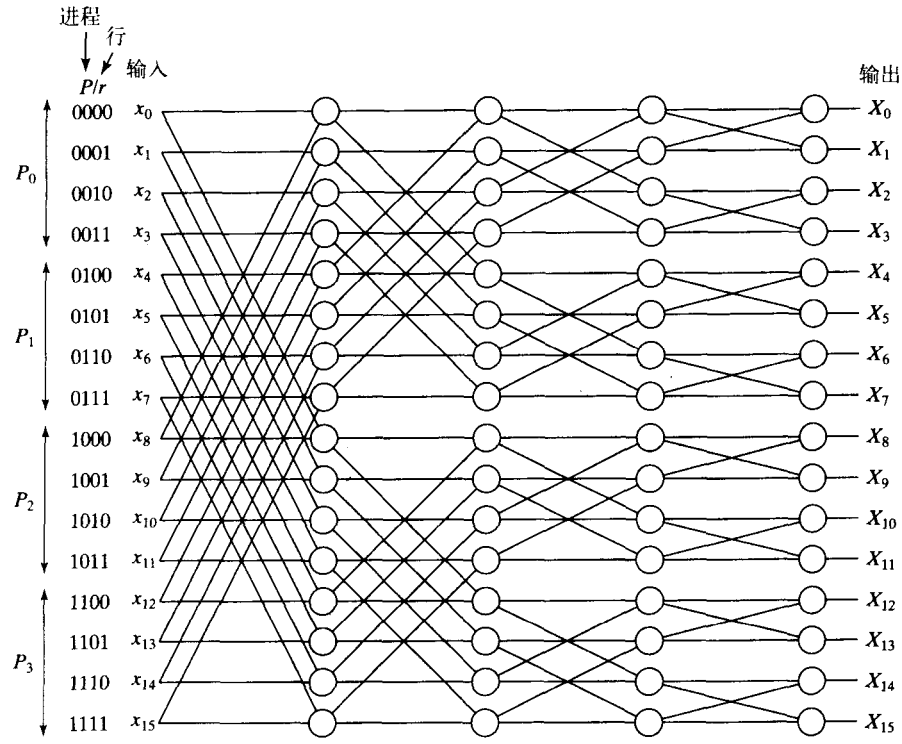


图12-32 将处理器映射到16个点FFT计算

398

（2）通信 若$p = N$，则每一步都需通信，因而在log$N$步中的每一步在处理器对之间要进行一次数据交换。假定所使用的是超立方体或其他允许进行同时交换的互联网络，则此时的通信时间复杂性由下式给定：

$$t_{\text{comm}} = \text{O}(\log N)$$

如果$p<N$，处理器间通信仅出现在前log$p$步中。在第一步中，所有p个处理器进行交换。在下一步时，仅有一半处理器进行数据交换，在再下一步时，仅有1/4处理器进行交换，以此类推。如果互联网络允许同时进行交换，则通信时间复杂性可简化成由下式确定：

$$t_{\text{comm}} = \text{O}(\log p)$$

当然，如果互联网络只允许进行顺序通信，则上述的这些复杂性将会变糟。

（3）转置算法 现假定$N = p$并且每个处理器从一个数据点开始启动。为说明方便，假定有16个数据点。如果处理器以二维阵列排列，例如以行主序，则通信将先出现在每列的处理器中，然后通信将出现在每行的处理器中。假定如图12-33所示的那样，为每一列分配一个处理器。则在前两步中，所有通信均在同一个处理器内进行；而在最后两步中，通信将在处理器间进行。在转置算法中，在前两步和最后两步之间，数组元素被转置，即将每一列中元素移到对应的行中，如图12-34所示。在转置后，进行最后两步操作时，现在将只涉及到处理器内的通信，如图12-35所示。处理器间要进

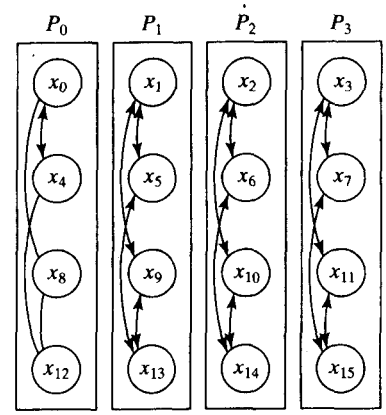


图12-33 用转置算法实现FFT（前两步）

399

行的唯一通信是将转置数组。

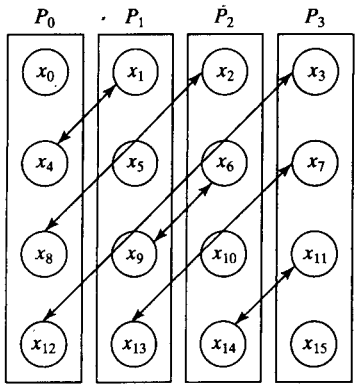


图12-34　转置算法中对数组进行转置

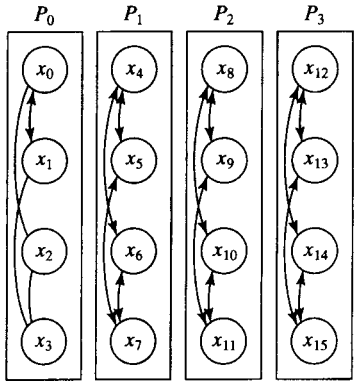


图12-35　用转置算法实现FFT（最后两步）

12.8　小结

本章涉及并行程序设计中有关图像处理的内容：

- 基本的低层预处理操作（阈值化、对比度扩展、直方图、平滑、锐化以及噪声消减）
- 用掩码进行边缘检测（著名的有Sobel算子和拉普拉斯算子）
- 霍夫变换
- 离散傅里叶变换和快速傅里叶变换

400

推荐读物

有几本有关图像处理的教科书，其中包括[Gonzalez and Woods，1992]、[Haralick and Shapiro，1992]、[Jain，1989]、[Sonka，Hlavac，and Boyle，1993]、[Castleman，1996]，以及[Bässmann and Besslich，1995]。而有关多处理机实现的研究性教科书是[Uhr et al.，1986]。

许多论文对低层预处理作了描述，如[Davis，1975]、[Sahoo,Soltani，and Wong，1988]，以及[Weszka，1978]。在[Ercan and Fung，1996]中可找到有关的硬件实现。在[Cole and Yap，1985]以及[Sen，1990]中可找到有关中值的算法。有许多其他的图像处理操作在本章中未被论及，例如收缩算法可将一个对象的范围减到仅是一个像素，从而使对象计算变得十分容易。[Rao，Prasada，and Sarma，1976]叙述了使用二进制加权掩码的并行收缩算法。[Arcelli and Levialdi，1972]考虑了三维收缩。

霍夫变换已用各种方法实现。[Choudhary and Ponnusamy，1991]中讨论了共享存储器的实现。[Carlson，Evans，and Wilson，1994]则涉及了许多有趣的应用。

[Cooley and Tukey，1965]是FFT算法的最早出处。以后又陆续出现了许多有关FFT各方面的其他论文，其中包括[Gupta and Kumar，1993]、[Illingworth and Kittler，1988]、[Norton and Silberger，1987]、[Swarztrauber，1987]以及[Thompson，1983]。论述FFT的教科书有[Elliott and Rao，1982]以及[Brigham，1988]。

参考文献

ARCELLI, C., AND S. LEVIALDI (1972), "Parallel Shrinking in Three Dimensions," *Comput. Graphics Image Process.*, Vol. 1, pp. 21–30.

BÄSSMANN, H., AND P. W. BESSLICH (1995), *Ad Oculos Digital Image Processing*, International Thomson Publishing, London, England.

BORN, G. (1995), *The File Formats Handbook*, International Thomson Computer Press, London, England.

BRIGHAM, E. O. (1988), *The Fast Fourier Transform and Its Application*, Prentice Hall, Englewood Cliffs, NJ.

CARLSON, B. D., E. D. EVANS, AND S. L.WILSON (1994), "Search Radar Detection and Track with Hough Transform Part I: System Concept, Part II Detection Statistics, Part III Detection Performance with Binary Integration," *IEEE Trans. Aerospace and Electronic Syst.*, Vol. 30, No. 1, pp. 102–125.

CASTLEMAN, K. R. (1996), *Digital Image Processing*, Prentice Hall, Upper Saddle River, NJ.

CHOUDHARY, A. N., AND R. PONNUSAMY (1991), "Implementation and Evaluation of Hough Transform Algorithms on a Shared Memory Multiprocessor," *J. Par. Distribut. Comput.*, Vol. 12, No. 2, pp. 178–188.

COCHRAN, W. T., J. W. COLLEY, D. L. FAVIN, H. D. HELMS, R. A. KAENEL, W. W. LANG, G. C. MALING, D. E. NELSON, C. M. RADER, AND P. D. WELCH (1967), "What Is the Fast Fourier Transform," *IEEE Trans. Audio Electroacoustics*, Vol. AU-15, No. 2, pp. 45–55.

COLE, R., AND C. M. YAP (1985), "A Parallel Median Algorithm," *Inform. Process. Letters*, Vol. 20, pp. 137–139.

COOLEY, J. W., P. A. W. LEWIS, AND P. D. WELCH (1967), "Historical Notes on the Fast Fourier Transform," *IEEE Trans. Audio Electroacoustics*, Vol. AU-15, No. 2, pp. 76–79.

COOLEY, J. W., AND J. W. TUKEY (1965), "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.*, Vol. 19, pp. 297–301.

DAVIS, L. (1975), "A Survey of Edge Detection Techniques," *Computer Graphics and Image Processing*, Vol. 4, pp. 248–270.

DUDA, R., AND P. HART (1972), "Use of Hough Transformations to Detect Lines and Curves in Pictures," *Comm. ACM*, Vol. 15, No. 1, pp. 11–15.

ELLIOTT, D. F., AND K. R. RAO (1982), *Fast Transforms, Algorithms, Analyses, Applications*, Academic Press, New York.

ERCAN, M. F., AND Y. F. FUNG (1996), "Low-Level Processing on a Linear Array Pyramid Architecture," *Computer Architecture Technical Committee Newsletter*, June, 9–15, pp. 9–11.

FOX, G., M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER (1988), *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, Englewood Cliffs, NJ.

GONZALEZ, R. C., AND R. E. WOODS (1992), *Digital Image Processing*, Addison-Wesley, Reading, MA.

GUPTA, A., AND V. KUMAR (1993), "The Scalability of FFT on Parallel Computers," *IEEE Trans. Par. Distrib. Syst.*, Vol. 4, No. 8, pp. 922–932.

HARALICK, R. M., AND L. G. SHAPIRO (1992), *Computer and Robot Vision*, Volume 1, Addison-Wesley, Reading, MA.

HOUGH, P. V. C. (1962), *A Method and Means for Recognizing Complex Patterns*, U.S. Patent 3,069,654.

HUERTAS, A., W. COLE, AND R. NEVATIA (1990), "Detecting Runways in Complex Airport Scenes," *Comput. Vision, Graphics, Image Proc.*, Vol. 51, No. 2, pp. 107–145.

ILLINGWORTH, J., AND J. KITTLER (1988), "Survey of the Hough Transform," *Computer Vision, Graphics, and Image Processing*, Vol. 44, No. 1, pp. 87–116.

JAIN, A. K. (1989), *Fundamentals of Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ.

KRISHNASWAMY, D., AND P. BANERJEE (1997), "Exploiting Task and Data Parallelism in Parallel Hough and Radon Transforms," *Proc. 1997 Int. Conf. Par. Proc.*, pp. 441–444.

MANDEVILLE, J. R. (1985), "A Novel Method for Analysis of Printed Circuit Images," *IBM J. Res.*

Dev., Vol. 29, Jan., pp 73–86.

NORTON, A., AND A. J. SILBERGER (1987), "Parallelization and Performance Analysis of the Cooley-Tukey FFT Algorithm for Shared Memory Architectures," *IEEE Trans. Comput.*, Vol. C-36, No. 5, pp. 581–591.

RAMIREZ, R. W. (1985), *The FFT: Fundamentals and Concepts*, Prentice Hall, Englewood Cliffs, NJ.

RAO, C. V. K., B. PRASADA, AND K. R. SARMA (1976), "A Parallel Shrinking Algorithm for Binary Patterns," *Comput. Graphics Image Process.*, Vol. 5, pp. 265–270.

SAHOO, P. K., S. SOLTANI, and A. K. C. WONG (1988), "A Survey of Thresholding Techniques," *Computer Vision, Graphics and Image Processing*, Vol. 41, pp. 233–260.

SEN, S. (1990), "Finding an Approximate Median with High Probability in Constant Parallel Time," *Inform. Process. Letters*, Vol. 34, pp. 77–80.

SONKA, M., V. HLAVAC, AND R. BOYLE (1993), *Image Processing, Analysis and Machine Vision*, Chapman and Hall, London, England.

SWARZTRAUBER, P. N. (1987), "Multiprocessor FFTs," *Parallel Computing*, Vol. 5, pp. 197–210.

THOMPSON, C. D. (1983), "Fourier Transforms in VLSI," *IEEE Trans. Comput.*, Vol. C-32, No. 11, pp. 1047–1057.

UHR, L., Editor (1987), *Parallel Computer Vision*, Academic Press, Boston, MA.

UHR, L., K. PRESTON JR., S. LEVIALDI, AND M. J. B. DUFF (1986), *Evaluation of Multicomputers for Image Processing*, Academic Press, Boston, MA.

WESZKA, J. S. (1978), "A Survey of Threshold Selection Techniques," *Computer Graphics and Image Processing*, Vol. 7, pp. 259–265.

习题

科学/数值习题

12-1　对于给定的p个处理器和n个图像像素，求按12.4.1节所描述的算法计算平均值所需的并行时间复杂性？

12-2　确定在什么情况下12.4.2节所描述的中值算法可找到中值以及在什么情况下找不到中值？在找中值时，什么是最坏情况的位置不精确性？

12-3　在图12-36中示出了一个有亮度变化的图像。一个3 × 3掩码已移到此图像上，用Sobel算子掩码扫过此图像，根据所得结果求得不同位置的梯度幅值和梯度角。假定灰区的值为0，白区的值为255且至少需使50%以上的像素区被突显出来以便识别。

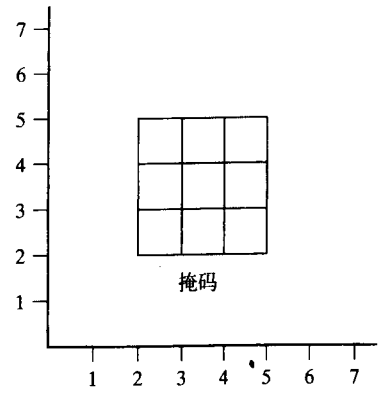


图12-36　习题12-3中的图像

12-4　虽然本章中使用的均是3 × 3的掩码，但实际上也可用其他大小的掩码，如5 × 5、7 × 7、9 × 9、11 × 11以及13 × 13的掩码。请编写实现加权掩码滤波的程序来处理任何$m \times m$掩码，其中的掩码大小（m为奇数）和它的内容将作为输入量输入。将此程序应用到一个图像上，并对不同的掩码进行相应的实验。

401
~
403

12-5　试编写一并行程序来对以标准未压缩结构，如PPM（可移植像素图）或PGM（可移植灰度图）文件格式，所存放的图像文件进行边缘检测。为便于观看，须将图像转换成

如gif那样的格式；这种转换可借助系统实用程序来完成。（可参见系统的“man”页面以及[Born，1995]）。

12-6 以P6格式（全色）读入PPM文件，编写一并行程序以将图像简化成灰度，这只需对每个像素取红色、绿色、蓝色值的平均值便可得到：

$$像素值 = \frac{红色值 + 绿色值 + 蓝色值}{3}$$

12-7 编写一个能实现拉普拉斯算子的程序。用以下的阈值化处理方法进行实验来进行边缘检测。即除了使用左、右、上、下邻接点外，还需对前述各点的邻接点进行实验。

12-8 假定已对一个图像进行了边缘检测，并生成了一个二元图像（边缘处为1，而其他地方为0）。设计一个算法并编写一个并行程序，它将1填入由边缘线围绕的对象中。进行这种填入的一个方法是，从设在对象中的一个种子点开始，递归地向其邻接像素点内填1，直至到达边缘为止。

12-9 对使用通常线表达式的霍夫变换编写一个并行程序，并在示例图像上评估该程序。

12-10 替代简化地增量累加器的另一种形式的霍夫变换（它具有的梯度算子也能提供梯度幅度∇f）允许将梯度加到累加器。用实验对此方法加以研究，并与简单地增量累加器方法做比较。

12-11 阅读由[Krishnaswamy and Banerjee，1997]合写的论文，该论文描述了如何使用以斜率-截距形式表示图像中的线进行霍夫变换，并请将此法与使用[Duda and Hart，1972]的通常表达式的方法做详尽比较。

12-12 确定在12.7.4节中所叙述的流水线离散傅里叶变换实现所需的计算时间。

12-13 阐明1点离散傅里叶变换的结果即是该点的值。

12-14 由于快速傅里叶变换是一个分治算法，它可被视为一棵树。请以树的形式设计连接。

12-15 试证明在12.7.4节中所描述的离散傅里叶变换的矩阵-向量公式（当$N = 4$时）为

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 \\ 1 & w^2 & 1 & w^2 \\ 1 & w^3 & w^2 & w \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

对于$N = 4$，试用向量-矩阵公式和FFT公式分别编写一个并行程序，并测量它们的执行速度。

现实生活习题

12-16 假定你受一个大型电影制片厂委托去开发一个非常快速的“造型”软件包，它能将一个图像改变成另一个图像。你的构思是有两个图像，一个是原图像，另一个是最后图像，通过将原图像中的每一个像素，以锁步的SIMD方式逐步改变成越来越接近最后图像。这种方法肯定是易于并行的，虽然它不可能最终获得一个非常平滑变化的形状。用此法进行相应实验，并用男演员或你朋友的像片向电影制片厂演示该软件包。 404

12-17 假定NASA已委派你一个任务，让你编写一个真正快速的图像识别程序，快到金星CAT（商业访问运输）足以能从以1000公里/小时的速度越过被绘制地图区域的VERMIN卫星（金星雷达图像和网络卫星）所拍摄到的地形图像中捕获着落地点，VERMIN图像地图涉及的范围为5公里×5公里并且水平和高度的精度均为0.5米。

适当的着落场所是一个范围为25米的圆形区域且区域内的最大的高度变化不超过1.5米。（为使CAT平稳的着落需要相当大和相当平的区域。）创建所采样的非完善地形的图像地图。一旦你的公司证明了它的CAT能力，NASA将向你提供详细的VERMIN地图。有关背景信息，请阅读由[Huertas, Cole and Nevatia，1990]的检测跑道的论文。

12-18　一家集成电路板的制造厂碰到的一个严重问题是，在它的电路板上有断开的印刷电路走线，制造厂委派你作为独立程序员找出一种方法，以能在集成电路板离开生产线之前，自动地检查板的图像并识别出断开的走线。已知的是所有走线在一个方向上是直线或者可以旋转90°到另一方向。设计一种策略以能识别走线中的断开处，并编写一个使用实际测试图像的并行程序。若感兴趣的话，请阅读[Mandeville，1985]。

12-19　Tom最喜爱的活动是将1 000块拼板（它们是从一幅完整的景色上冲切下来的单个且是独特形状的纸板块）。利用形状和着色作为提示，Tom捡起两块拼板试图将它们拼在一起，可能要对其中一块或另一块进行旋转。有时Tom判断错误使这两块拼板不能合适地拼在一起；此时他不得不将两块板放回板上并再次试拼。当它们能被拼在一起时Tom将它们放到板上，此后就将它们视为是一块板。然后Tom就重复这一过程。由于对计算机科学的开发的广泛兴趣和对并行程序设计的特别爱好，Tom在一张小桌上方装配了一台摄影机，而在小桌上则铺摆着他正在拼接的所有拼板。该摄影机连到他的N台计算机中的一台（是的，他有N台家用计算机，从一台老的2GHz的P4 到一台时钟为12GHz，数据通路为256位的64-CPU 的P9）生成一幅压缩的30 000 × 30 000像素图像。Tom的目标是使N台计算机并行工作以生成一张用最少步数装配拼板的指导表。

（容易）列出为了帮助Tom解决这一并行计算问题所需要用到的计算机科学的各个方面；解释每一方面所起的作用。

（较难）略述Tom可在N台计算机求解该拼板难题的方法。确定他需采取的步骤，
405 但不需要实际实现生成拼板装配指导的程序。

（更难）略述如果每块拼板形状不是独特的时候需要做什么。

（最难）在不论什么样的可用的并行系统上实现一个拼板求解程序。

第13章 搜索和优化

本章将讨论有关搜索和优化方法。首先我们讨论的是*分支限界方法*，它的并行实现会引出在前几章中已介绍过的一些技术。然后我们将详尽地论述另一种相当不同的方法：即使用*遗传算法*。我们将讨论遗传算法和它的各种并行化方法的基本技术。最后我们将转向叙述*爬山技术*，这种技术常被应用于金融的优化问题。如同第12章一样，本章所提供的信息可用作并行程序设计的课程设计。

13.1 应用和技术

组合搜索和优化技术的特征是从对问题的许多可能求解中找到一个解。对于许多搜索和优化问题，穷尽枚举搜索是不可行的，相反宜采用某种指导性搜索。此外，不仅是着眼于最优（优化）解，通常也寻找一个好的非优化解。

用组合搜索和优化技术要解决的经典计算机科学问题包括旅行商问题、0/1背包问题、n皇后问题，以及15和8拼块问题。在旅行商问题中，其目标是一个售货员从一个城市出发，访问列表中每一个城市且仅访问一次，最后再返回到出发城市，要求找出整个旅行的距离为最短的一条路线。在0/1背包问题中，各个对象被赋予不同的得益值，而目标是将所选对象装满背包以获得最大的得益值。记号“0/1”用来指明对象是被选（1）或是不被选（0）。n皇后问题的目标是将n个皇后放在一个$n \times n$棋盘上，使所有n个皇后不会互相攻击。在8拼块问题中，8个拼块的号码分别为从1到8，它们被放置在一个3×3的板上，其中有一个空块。其目标是，每次向空块中移入一块，最后要使所有块以行为主序（即在顶行中为1、2、3块，第2行中为4、5、6块，而在底行中为7、8块）在底板上排列。15拼块问题是类似的，不同的仅是用15块在一个4×4底板上进行排列。上述的所有问题所表现出的特征是，它们的求解需要进行大量的排列，而且进行完全的搜索将非常耗时。在背包问题和旅行商问题中，还未曾找到在多项式时间内可完成的算法（在最坏情况下），因而这类问题属于NP完全问题。

搜索和优化技术在商业、银行和工业中也有许多重要应用。例如，金融预报、航班和乘务员的编排以及VLSI芯片的布局。VLSI芯片布局问题在概念上类似于背包问题，其目标是，给定每个元件的形状，在VLSI芯片上布放这些元件使所占用的芯片面积最小。该问题的求解有一些约束条件，其中包括芯片形状需为矩形。各个元件被布放在芯片中的不同位置时，将会导致不同的未被使用空间。在实际设计中往往还有附加的约束条件，如芯片的方向比（长/宽比）以及元件间的走线等。组合搜索和优化技术在编译器的寄存器优化分配和多处理机调度中也有广泛的应用。

在前面提到的问题中，可使用的搜索和优化技术有好几种，其中包括：

- 分支限界搜索
- 动态规划
- 爬山
- 模拟退火
- 遗传算法

本章中，我们将首先对分支限界技术加以概述，因为它是基本的顺序搜索和优化技术之一。

然后我们将稍为详细地论述遗传算法。此后，我们将讨论爬山搜索的一个例子。我们将只论及上述各方法的并行化策略。有关其他方法的并行化策略，读者可参阅其他文献。在[Quinn，1994]中可找到并行的动态规划方法，在[Witte，Chamberlain and Franklin，1991]中可看到有关并行模拟退火方法，而在[Michalewicz，1996]中则对遗传算法、模拟退火以及爬山法三者做了比较。

13.2 分支限界搜索

13.2.1 顺序分支限界

分支限界的搜索行动可用状态空间树（state space tree）来描述，树的根表示搜索的起点。从根结点到下一层表示朝问题求解方向所做的各种选择；例如，在背包问题中选一个对象，在VLSI布局中选一个元件及对它的布放。一旦进行了这一选择，接着就要进行另一种选择，
407 并使搜索过程移向树的下一层。就实质而言，该问题被分成一些子问题，而这些子问题本身又以分治方式被进一步求解。这些树的结点就变成了要求解的子问题。

图13-1中说明了一个通用的状态空间树，它可在许多求解问题中加以应用。开始时可有n种选择，从C_0到C_{n-1}。对于背包问题，每个选择将是首选的对象。对于旅行商问题，每个选择便是对下一个要去的城市的选择。对于n皇后问题，这将是选择第一个皇后被摆放的位置。所述的树构造因为与求解问题相关，故称为*动态树*参见[Horowitz and Sahni，1978]。状态空间树也可通过在每一结点进行真伪选择而变成二叉树。例如，在背包问题中，在每个结点处的选择或是选一个特定对象或是不选。这种形式的树被称为*静态树*。

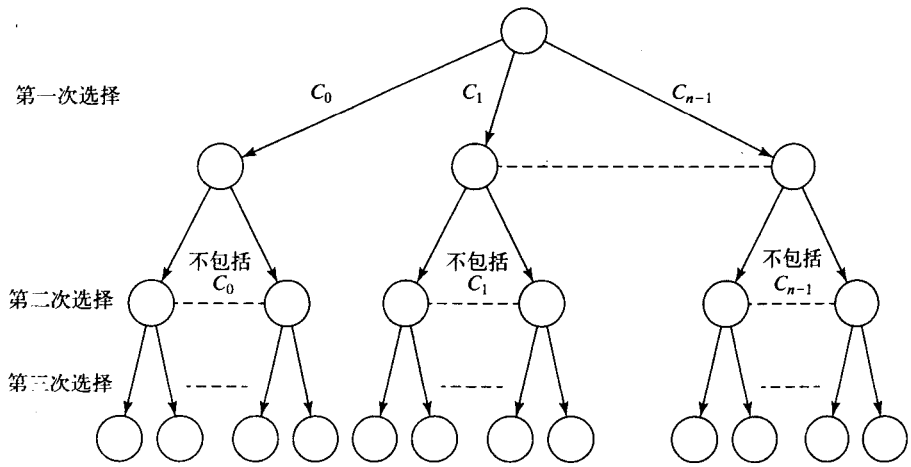


图13-1 状态空间树

状态空间树的搜索可用*深度优先*（depth-first）方法即从左边开始，从一个结点向下到下一层的一个结点，一直到最深的那层，然后从左转向右。搜索树的搜索也可用*广度优先*（breadth-first）方法，即在搜索移向下一层之前，在本层中从左到右先加以搜索。通常，对所有结点的穷尽搜索的代价过于昂贵，因此需要采用减少搜索的策略。一种常用的策略称为*最佳优先*（best-first），它将引导搜索沿着最可能获得最好解的路径进行。对于不能找到比现有解更好解的搜索路径，将不会向前搜索。为防止进入这种不被看好的搜索路径，需要对状态空间树进行*修剪*。

在减少搜索空间方面修剪是一个搜索方法的关键部分，但这要求正确判断在往下的状态空间树中是否不存在一个更好的解。为此，必须设计一个*限界函数*（bounding function）或*修剪函数*（cut-off function）。当搜索到达每一结点时，就需对所提供的下限（对求最小解问题）或上限（对求最大解问题）进行评估，以确定是否要进行任何更进一步的搜索。在背包问题情况下，限界函数将产生最大的可能得益值；而对VLSI布局问题，它将产生一个需最小芯片面积的一个解。所求得的限界函数的结果，将与到目前为止所获得的最好的解进行比较。 408

当搜索到达某一结点但尚未对它的所有孩子结点进行搜索时，称该结点为*活结点*（live node）。在一个活结点中，那些目前正被搜索的子结点被称为*E结点*（正被展开）。当所有子结点被搜索后，该父结点便成了*死结点*（dead node）。在分支限界中，一个*E*结点的所有子结点均为活结点。在搜索进行过程中，含有活结点的表被存放在一个队列中，在搜索算法中将使用此队列。有时称此队列为*开放表*（open list）（若必要的话，那些死结点将被放在一个*封闭表*中，以防止重复）。对于最佳优先策略，此队列将是一个*优先级队列*，其中最有可能找到最佳解的活结点将被放在队列的前面而将被首先选择。

在*回溯*中（通常不同于分支限界），当一个搜索不应再向下进行时，搜索便返回至上一层以继续在该层进行搜索；这就是“回溯”。实现这种机制的一个合适的数据结构是堆栈（后进先出），用它来保存已访问过的那些结点。

13.2.2 并行分支限界

状态空间树非常适合于并行化。例如用若干独立处理器可对状态空间树的不同部分进行独立的搜索。一种最自然的方法是从一个结点中将状态空间树的一部分分配到一个处理器。然后此处理器在它的搜索树区域中进行搜索。如果一个深度优先搜索被分解成多个独立的深度优先搜索时，通常就被称为是*并行深度优先搜索*，虽然严格地讲，当每个处理器以一个活结点向下移至下一个结点时，并行化在整个树的范围内所创建的实际上是一个广度优先的波前。此外也可以在评估限界函数和在对下一个*E*结点进行选择时实施并行化。

不幸的是，存在一些问题使得并行化变得更为复杂和效率低下，而并非像原先想像的那样简单和高效。分支限界搜索要求将限界函数求得的下限（或上限）通知所有的处理器，使它们在任何时刻能优化地对它们所搜索的部分树进行修剪。但是要进行修剪的那一层由于可能找到了更好的解，因而在搜索期间会发生变化。

此外，在任何分割部分中的状态空间树的大小事先并不知道，这就使负载平衡变成一个严重问题，为此可采用第7章中叙述的负载平衡技术。开放表队列是一个共享数据结构，类似于7.4节中的最短路径问题的顶点队列。事实上最短路径问题可视作为搜索和优化问题，因为其目的是通过图寻找最优路径。通常最短路径问题的求解常与其他的图算法组合在一起，如寻找已连接的元件。图搜索和本章所叙述的状态空间搜索技术之间的根本差别是在于，在图搜索中，图在开始搜索前是已知的，而状态空间树的结构在搜索开始前是未知的。

1. 并发的队列访问

对于共享存储器实现，必须对队列加以保护，以防止对它同时进行访问，为此我们可采用第8章中使用锁的共享数据技术。但是由此而造成的顺序化将严重限制可能达到的加速比。 409
如[Rao and Kumar，1998]所指出的那样，可达到的最大加速比为：

$$S(n) < \frac{t_{queue} + t_{comp}}{t_{queue}}$$

式中t_{queue}为访问队列的平均时间，t_{comp}为平均子问题（结点）的计算时间。该结果实质上是从阿姆达尔（Amdahl）定律推得的（1.2.2节）。

除了对整个队列加锁，可只对队列中的每一项加锁。当一个处理器访问队列中的第1项时，它就被上锁。下一个处理器由于第1项已被上锁，就只能访问队列中的下一项。如前面所提及的，最佳优先策略使用优先级队列。在每次插入或删除操作后，形成一张有序表就可实现这种优先级表。但这种方法的时间复杂性为$O(n\log n)$。优先级队列也可用堆（heap）数据结构加以实现，这种方法有更高效的插入和删除。堆是一个完全的二叉树，其特点是每个结点拥有的值至少会与其子女拥有的值相等。在优先级队列环境下，父母的优先级高于其子女的（或至少是一样的，如果有多个结点具有相同优先级值）。因此根结点拥有最高优先级的项。插入一个新项的算法在开始时是先将该项插入到堆的底部（从左到右，最底层中的第一个自由叶结点）。此后在进行一系列的比较和交换后，沿树向上移动，直至到达能保持堆特征的位置为止。对最高优先级结点（根结点）的删除意味着用堆的最后一项来替换根结点，然后将其向下移动直至堆的特征再次被保持为止。这种插入和删除算法只需$\log n$步，但需要访问多个结点。[Rao and Kumar，1988]叙述了一种算法，它只对称为*窗口*（三个结点用作插入，一个结点用作删除）的堆的一部分加锁。在每个结点中必须存有附加的状态信息以控制对堆的多重访问。有关更详尽的叙述请读者参见[Rao and Kumar，1988]。

为避免集中式队列中可能出现的问题，可将队列在处理器中分布式地存放，并间或互相广播较好的结点。这些策略将在消息传递系统中使用。[Janakiram，Agrawal，and Mehrotra，1988]已研究了（在回溯环境下）以随机方式而不是以定序方式来选择子结点的方法，以达到减少处理器间通信的目的。

2. 加速比异常

当用深度优先方法进行顺序搜索时，处理器从根结点开始搜索，然后首先遍历最左边路径，此后向下一层又是先遍历最左边路径，如此继续直到找到解答。当p个处理器独立地运行在分离的向下路径上时，很可能其中有一个处理器很快找到了一个可行解。在这种情况下，并行加速比会大于p（超线性加速比）。这就是所谓的*加速异常*。相反的，一个可行解可能处于树中的某一位置，该位置在搜索时间为$1/p$时仍未到达该点，而若用单处理器进行搜索，此
410 时已经到达。在这种情况下，加速比就会小于p。如果此时的加速比仍大于1，则这就是所谓的*减速异常*。加速比也有小于1的，此时的这种异常被称为*不利异常*（detrimental anomaly）。[Lai，and Sahni，1984]以及[Li，and Wah，1986]对为何会出现这些异常做了详尽研究。类似的研究结果也可参见[Wah，Li，and Yu，1985]。

13.3 遗传算法

13.3.1 进化算法和遗传算法

遗传算法的理论有其生物学根源。遗传算法企图模仿个体种群的自然界进化过程。尽管到目前为止对自然界进化的真正机制还未能很好了解，但在某些方面的研究已到了相当深度。长久以来，生物学中的一个重要研究领域是对染色体——含有生物特征的信息载体的研究。虽然对有关这些染色体支配生物组织的性质这一点仍存在某种不确定性，但事实上很少有人怀疑它们确是唯一地确定了生物组织的性质。

进化是在染色体层次上通过再生的过程，而不是生物个体本身的再生过程。当一个种群中的个体再生时，每个双亲的部分遗传信息（部分的双亲染色体）被组合在一起以生成它们

后代的染色体。通过这种方法，后代获得遗传构造信息，该信息是它们双亲信息的混合物并且显示出双亲特征的相应混合。组合产生后代的来自双亲的染色体是主要机制，借助这种机制可使染色体模式发生改变。从每对双亲处继承某些特征在术语中被称为交叉（crossover）。

另外，在一个特定个体的染色体模式中可能出现一个偶然的随机变化：这种影响在术语中被称为突变（mutation）。这些突变可能导致个体的染色体模式显著地不同于任一双亲。与个体的双亲相比，一个突变可能增强个体的生存力或是损伤它的生存力。突变最重要的作用是它可引起个体生活力的突然改变，而这与双亲无直接关系。由于衰减一个个体的生存力的方法比改善其生存力的方法要多的多，因此大规模出现的突变将趋向于衰减种群的生活力。很少会出现那种能偶然地产生有益改变的突变，而且这种改变还不会胜过对种群具有不利的改变。

如同对一个生物体的染色体的所有变化一样，还存在有一种生物体与环境的交互变化。相对于它的祖先，有时这些变化会改善已变化的生物体的生存力，从而导致增加其生存的可能性，并会将它的染色体信息传递给它的后代。很自然地，也存在同样的可能性，即这种改变会降低生物体的生存力，从而减少了将那些信息传递给其后代的机会。一般认为，虽然突变的发生频度较低，但与交叉相比，它在自然界的进化中起着更重要的作用。 411

当突变率相对较低时，不适应的个体在向它们的后代传递它们的特征前就趋向消失；相反地，较适应的个体一般会成功地将它们的特征传递给后代。然而，当突变率相对较高时，会引入大量的个体数，且它们的特性显现出与上一代的特性随机地不相同，从而在本质上导致种群的衰退和不稳定性，而在遗传算法的计算机实现中即是出现不收敛情况。

随着时间推移，染色体的这些变化将产生一些变种，它们与原先的生物体种群的成员有很大差异。一般而言，以某种方式增强物种生存力的那些变化随着时间推移会趋向处于支配地位。例如，在鸟种群中，对鸟巢中的鸟蛋担负主要照看作用的是雌鸟。雌鸟的羽毛颜色与雄鸟相比一般都较柔和。用这种方法，雌鸟与周围环境融合在一起，而不是像雄鸟那样以鲜艳的颜色羽毛引起对它们的注意。在鸟巢中长时间栖息不动并融入周围环境的那些鸟的个体的寿命，一般要比会引起食肉动物注意的那些个体的寿命要长。因而雌鸟趋向于将它们柔和色的染色体传递给更多的后代。最终，种群逐渐加权地朝那些能趋于增加寿命的特性方面改变。

上述理论是1859年由查尔斯·达尔文在他的经典进化论著作中提出的，即自然选择理论，或更普遍地被称为是“适者生存”理论。恰如前面所例举的鸟的例子那样，在任何的一个物种的种群中，“较强”或“更适应”的个体会有更多的生存机会，因而会向它们的后代传递会使它们易于生存的那些特征。当然也肯定存在这样的例子，即一个特别适合的个体不能存活长久，从而无法繁衍！这种例子将是那种正好是在“错误时间和错误地点”出现从而过早地成为另一种物种的食物源的个体。但正如权威人士所指出的那样，赛跑的获胜者不一定总是跑得最快的，格斗的获胜者也并不一定总是最强壮的，但平均而言，我们可断定会出现那样的结局。

重复地使用交叉和突变原理，然后每一次基于被繁殖成员的相对适应度来生成“下一代”，就被称为是遗传算法。如同自然界一样，这种有助于“解群”（population of solution）优化的方法，随着时间推移会趋向产生好的解。在后面几节中将更详细地叙述如何来实现这种遗传算法。

遗传算法是那种按生物进化过程模型对问题进行求解的计算方法。该算法首先要创建一个初始的“解群”（个体）。然后使用专门的适应标准对这些个体进行评估，将它们按从“最

适应”到“最不适应”的不同程度加以分类。此后用倾向于“较适应”个体的准则，从中选择一个种群的子集。接着用这一子集来生成新的后代。最后，再对下一代中的少量个体施以随机突变。

上述选择、交叉和突变的过程不断重复，从而可生成许多代。如同自然界中一样，基本假设是，只要遵循用给定代中更适应的个体去生成下一代中种群中的各个个体将趋向进化和
412 改善其适应性。尽管遗传算法不能保证找到最优解，但通常它能很快地找到相当好的解。

13.3.2 顺序遗传算法

以下是有关顺序遗传算法的伪代码概要:

```
generation_num = 0;
initialize Population(generation_num);
evaluate Population(generation_num);
set termination_condition to False;
while (not termination_condition) {
    generation_num++;
    select Parents(generation_num) from Population(generation_num - 1);
    apply crossover to Parents(generation_num) to get
        Offspring(generation_num);
    apply mutation to Offspring(generation_num) to get
        Population(generation_num);
    evaluate Population(generation_num) and update termination_condition;
}
```

在前面的讨论中，一些问题在某种程度中已被掩盖起来。其中包括:

- 如何用染色体来表示一个个体或一个可能解
- 如何生成可能解的初始种群
- 如何评估每一个可能解以确定其适应性
- 如何确定重复终止的条件
- 如何选择变为下一代“双亲”的个体，以及这些下一代个体如何确切生成

一旦搞清楚上述的这些问题后，我们将转向讨论有关并行化顺序方法的问题。

13.3.3 初始种群

为了演示遗传算法中开始的几步，下面我们讨论一个简单的求解问题，即对在−1 000 000到 + 1 000 000的整数范围内每一个变量值，确定下列函数表达式的最大值:

$$f(x, y, z) = -x^2 + 1\,000\,000x - y^2 - 40\,000y - z^2$$

(为简化讨论问题起见，将变量x，y和z限制为取整数值。)

对该问题可能偶然会有一个直接解，它能方便地用来验证用其他技术求解所得到的结果是否正确。然而在实际中，由于其他技术的求解不能保证得到一个最优解，因此对于那些不存在简单接近解的问题，只能采用搜索和优化技术。在对上式稍作简单的代数变换后，可将f分解成如下的形式，从中可容易地看出它的最大值:

$$f(x, y, z) = 2504 \times 10^8 - (x - 500\,000)^2 - (y + 20\,000)^2 - z^2$$

在这种形式中，可明显地看到$f(x, y, z)$的最大值为250 400 000 000，仅当x为500 000，y
413 为−20 000以及z为0时才会出现。因为只有当x、y、z取上述相应坐标值时，才会使式中的各平方项取值为0，而当它们取任何其他坐标值时，这些平方项就会是某一正值，而要从2504×10^8

这一项中减去从而减小 f。现在让我们假设这样的直接解不存在，以方便我们的讨论。（如所指出的那样，我们用此函数作为我们的测试案例，从而可使计算机的求解较容易地与上述确切解做比较；实际上可用任何函数）。

由于问题的计算规模大，采用对每一个可能坐标值进行函数计算的简捷的穷尽搜索方法是不现实的。因为共要评估 $(2\ 000\ 001)^3$ 个坐标值，并且即使假设相应于每次函数值的计算只需100ns（这是对任何合理的评估函数的高度乐观估计），在单处理器上运行将需要200 000 000多个小时。即使我们对此算法实施并行化使计算速度提高10 000倍，但仍需要两年多的时间才能计算完毕。

1. 数据表示

确定可能解的初始化种群的第一步是为一个个体选择一个合适的数据表示。在早期的遗传算法研究工作中，简单地以1和0的字符串来表示可能解；较近期的研究工作已将这种表示进行了扩展，包括浮点数、葛莱码以及整数字符串[Michalewicz，1996]。为简单起见，我们仍使用二进制字符串。

第i个可能解将由一个三元组（x_i，y_i，z_i）组成，三元组中的每一个在间隔范围$-1\ 000\ 000 \leqslant$元素值 (component_value) $\leqslant +1\ 000\ 000$内，将取2 000 001个可能整数值中的一个值。由于

$$2^{20} < 2\ 000\ 001 \leqslant 2^{21}$$

因此三元组中的每一个元素需用21位二进制位来表示。这样一个可能解（即一个染色体或一个个体）就由63位组成，它实际上是（x_i，y_i，z_i）表示的并置。

假定$x = +262\ 408_{10}$，$y = +16\ 544_{10}$，以及$z = -1\ 032_{10}$，则使用“符号加值”的表示方法时，上述每个数的二进制表示式为：

$$x = 001000000000100001000$$
$$y = 000000100000010100000$$
$$z = 100000000010000001000$$

每个数中的最左位是符号位（0表示正值，1表示负值），而其余位则表示值的大小。应注意的是，这种表示将保证值的大小处于$-1\ 048\ 575$和$+1\ 048\ 575$或$\pm 2^{20}$的范围之间。将它们并置成一个二进制字符串后，就可得到可能解的一个表示：

001000000000100001000000000100000010100000100000000001000000001000

现在我们已确定一个可能解的长度为63位，这样我们就可用伪随机数生成器方便地引入它们的一个初始种群。

2. 评估

下一步我们将使用函数中的x、y和z的值，对这一个体的适应性加以评估，即：

414

$$f(x, y, z) = -x^2 + 1\ 000\ 000x - y^2 - 40\ 000y - z^2$$

$$f(262\ 408, 16\ 544, -1\ 032) = -(262\ 408)^2 + 1\ 000\ 000 \times 262\ 408 - (16\ 544)^2$$
$$-40\ 000 \times 16\ 544 - (-1\ 032)^2$$
$$= -68\ 857\ 958\ 464 + 262\ 408\ 000\ 000 - 273\ 703\ 936 - 661\ 760\ 000 - 1\ 065\ 024$$
$$= 192\ 613\ 512\ 576$$

对种群中的每一个个体都要重复进行上述的评估过程。即对应于每一个个体要进行函数值或适应性的计算。此后按它们可使函数取最大值的好坏程度加以排序。对那些可使函数的值大于192 613 512 576的个体将排序为“较好适应”，而将那些小于此值的个体排序为“较差适应”。

3. 约束

很自然地，由于我们变量取值范围被限在$\pm 1\ 000\ 000$，那么，任何个体如果它的x、y或z

的值落在此范围之外时，就被认为会有差的适应性，因而就不必再求它的函数值以期望它可能成为双亲。在自然界，这就相当于流产的个体。同样，还有其他方法可对个体的取值范围加以限制。使个体落在±1 000 000间隔范围外的缺陷，可采用“外科手术修复”方法加以补救（将1位或多位加以变化以改正此缺陷）。我们也可采用另一种方法，即在对适应性进行评估之前，将已生成的个体映射到所限制的范围内，这只需对每一个坐标值乘上一个比例因子就可完成:

$$\text{乘比例因子后的坐标} = -1\,000\,000 + (\text{坐标}/2^{21}) \times (2\,000\,000)$$

有关在遗传算法中所使用的对个体求解限制方法的更进一步的讨论可参见[Michalewicz，1996]。

4. 个体数

最后的有关问题是“在初始种群中应包含多少个个体？”。一般而言，当个体数少时，将减少在较合理的繁殖代数内获得合适解的可能性，而当个体数很大时，将会增加每一代所需的计算量。通常一个初始种群由20到1000个伪随机可能解组成。就本例而言，这就意味着有20到1000个伪随机生成的二进制字符串，每一个的长度为63位。

13.3.4 选择过程

仿效自然界，任何个体应有可能被选为下一代后代的双亲。然而，如同在自然界中一样，一个有更好适应性个体比适应性差的有更多的机会被选中。这种向有更好适应性个体的倾斜被称为*选择压力*（selective pressure）。

初看起来，好像应采用高的选择压力（例如，只选最适应的种群部分来生成下一代中的
415 后代）。然而，对存在局部优化解的那些问题，这可能导致搜索过程很快收敛到一个局部优化解，从而完全丢失全局优化解。在相反情况下，如果采用过小的选择压力，则会造成很慢的收敛，甚至不收敛；因此上述两种极端情况均应避免。

基于在许多应用中业已证明能有相对较满意的性能，在挑选选择压力时，一个首选是*竞赛选择*。

竞赛选择

在此方法中，每个个体有均等机会随机地从总种群中被选中并参加竞赛。每次竞赛将选择k个个体参加，并对每个个体进行适应性评估。在k个个体中最适应的那个个体将成为整个竞赛的获胜者并将成为下一代个体的双亲。总计要进行n场这样的竞赛，以确定n个个体，并用它们生成下一代，它所保持的种群规模与前一代的规模是一样的。

很显然，当k为1时选择压力就完全消失；种群中的每个成员都有相等机会被用来生成下一代。当k值很大时，则选择压力的力度也相应地增大，因为从k个随机选择的个体中，仅是被确定具有最大适应性的个体会被选择作为双亲。通常较典型的是k取值2，令人回想起中世纪由两个骑士进行的格斗（或竞赛）。

13.3.5 后代的生成

一旦通过一系列竞赛从当前代中选择了有较好适应性的那些个体后，必须将它们的染色体组合在一起以确定这些个体的成分，从而形成它们下一代的种群。这种将每个双亲中的染色体的一部分组合起来以生成子女染色体的过程称为交叉。虽然存在有好几种常用的交叉形式，下面的讨论将被限制为单点交叉。

1. 单点交叉

在单点交叉中，将把交叉点随机地定位切割在A和B每个双亲m位染色体模式的第p位处。

该切割将把每个双亲的模式分为两部分，子女1的染色体模式由双亲A模式的前p位和双亲B模式的后$m-p$位组成。子女2的染色体模式则正好相反，由双亲B模式的前p位和双亲A模式的后$m-p$位所组成。如术语交叉所蕴含的，在形成子女染色体模式的过程中，是将双亲的染色体模式的一部分相互交叉而形成的。图13-2对此作了说明。例如，让我们来观察两个已被选为双亲的个体。每个个体有63位长的染色体模式

A =001100101000000100001 0100101000000101010100001111111100000011000

和

B = 000001000000101000001 1010000010101010101000111000101010101010000

如果切割点定在第15位，那么双亲染色体的两部分分别为:

双亲A（前15位）: 001100101000000

双亲A（后48位）: 100001010010100000010101010000111111100000011000 416

双亲B（前15位）: 000001000000101

双亲B（后48位）: 000001101000001010101010101000110001010101010000

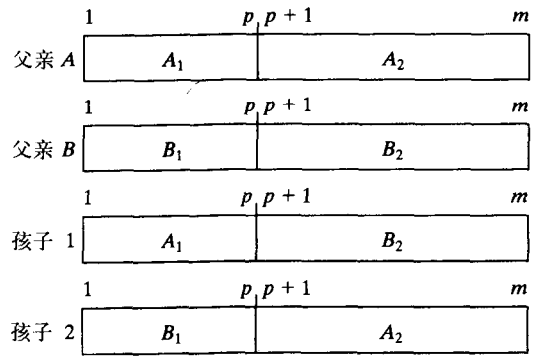


图13-2 单点交叉

在进行交叉后，将产生一对子女:

子女1: 001100101000000 000001101000001010101010101000110001010101010000

子女2: 000001000000101 100001010010100000010101010000111111100000011000

子女的基因，即它们染色体的个体位，是它们双亲基因的混合体。由于选择过程偏向从当前代中选择有较好适应性的个体，因此子女的特征是由较好适应性的双亲特征的混合表现。如同自然界中一样，交叉缓慢地迫使种群朝更强（即有较好适应性）的个体方向演化。由于这是一个渐进的、相对而言是可预测的变化，因此在遗传算法中通常鼓励使用交叉遗传操作，且一般在每一代中都要进行。

本例中我们所讨论的仅是单点交叉方法，应注意的是已研究了其他的一些方法。其中包括多点交叉和一致交叉，在前者中要进行多处切割并使多个较小的染色体部分混合在一起，在后者中，每一位（或基因）是从任意双亲中随机选取的。另一种已研究的方法是，从双亲池中共享使用基因，即一个后代中的每个基因是随机地从该池中一个双亲处选取的。

2. 突变

与交叉不同，突变出现在一个染色体的孤立的基因级上。这可视为类似于疾病、辐射或各种可控物质吸入对生物体的影响。突变将导致一个个体适应性的重大的、不可预测的变化。

在前面求函数最大值的例子中，对x、y或z的21位模式中的最左位加以改变，将导致该坐标值的符号改变，但幅值仍保持不变。对左数第2位的改变（该位权值为2^{19}）将导致x、y或z 417

在数值上发生524 288的变化。但若对最右位加以改变，则其数值变化仅为1。

当一个个体的x、y或z的分量中的某一位发生突变时，根据具体发生在哪一位上，将造成的坐标值的变化会从最小的(± 1)到最大的($\pm 1\ 000\ 000$)，因此很显然，突变可能是影响一个个体适应性的特别强的因素。当突变的概率较高时，就会使较适应的个体特性被改变的机会也相应增高(从而会以剧变方式衰减适应性)。在这种极端情况下，高突变率可能导致遗传算法的不收敛。

由于突变可能会急剧地改变个体的适应性，且与交叉的影响相比它的可预测性差的多，因此在遗传算法中通常使突变出现的机会维持在一个很小的概率。这就一方面允许在种群中引入已突变的个体，另一方面使它们出现的速率足够慢，从而防止种群连续地偏离对一个优化解的收敛。通过选择和交叉，从整体来看，突变的影响会逐渐地渗入到种群中。但突变有利时（即突变改进了个体的适应性），则已突变的个体被选作产生下一代个体的可能性也就增大。这就会自动地形成有利突变的传递，并不利突变消失。

13.3.6 变异

前几节所述的仅是寻求生成下一代成员的一些方法，而决非是全部。除了仅通过对当前代实施交叉和突变来生成下一代之外，还可采用如下一些方法:

- 从当前代中撷取少量最适应的个体作为下一代的一部分
- 在每一代中随机地创建少量新的个体，而不是仅在算法的初始化阶段随机生成个体
- 在逐代的演变中允许改变种群的大小

13.3.7 终止条件

已经采用了一些策略来终止遗传算法，这些方法包括从对预期的代的数目进行简单地计数到确定与真正的解有多接近以使其收敛。在最简单的情况下，连续代的生成执行s次。尽管这是一个易于实现的终止条件，但它却有两个明显的缺陷。在一种极端情况下，种群很可能在远小于s代之前就已收敛于一个解，而在另一种极端情况下，在获得满意的收敛之前，仍需要经过很多代。

由于优化解在事先无法知道，因此人们无法简单地通过测量当前“最好解”和优化解之间的差距来确定是否终止遗传算法。而在另一方面，我们能从数值分析领域中借用一种技术
418 对连续迭代的结果加以考察，然后根据连续代之间改善程度来确定是否终止。当然此时遗传算法也会像各种数值算法一样在一个解的附近显示出振荡，而不会进一步收敛。由于这种振荡不论是在个体选择概率差接近相同（最小选择压力）情况，还是在高突变率情况下，都已为实验证实会发生，因此单单依靠这种方式来终止遗传算法时，就必须较为谨慎。

还有另一种终止条件，它是基于种群中个体的相似程度来确定的。当个体解候选者的种群收敛于一个优化解时，由于它们的相似性会增加，因此测量种群的变异性就相当于测量收敛性。对一个遗传算法设置多个终止条件也很常见。

13.3.8 并行遗传算法

从概念上讲，将遗传算法改写成适用于多处理机的问题是非常直接了当的。几乎立刻可想到采用以下的两个方法:

1）让每个处理器独立地运行于个体的一个隔离子种群上，通过迁移与其他处理器周期地共享它的那些最好的个体。

2）让每个处理器在一个公共的种群上完成算法中每一步的一部分——选择、交叉和突变。

1. 隔离的子种群

在上述的第一种方法中，每个处理器首先独立地生成它自己个体的初始子种群。然后每个处理器对k代个体进行:

- 适应性的评估
- 从它的子种群中选择个体以用来生成下一代
- 在它的子种群上完成交叉和突变计算

在经历上述k代后（$k \geqslant 1$），这些处理器就与其他的处理器共享它们的最好个体。

（1）*迁移算子*（migration operator） 当将迁移结合进来后，种群的变化就不再单是由于继承其双亲的基因造成的，偶尔也会有随机突变所造成的，此外也是由于引入新的品种所造成的。很自然的是，子种群间的移动通常是生存反映，虽然人类通过提供“输送”机制加速（有时是无意的）了这一过程。这些被“输送”品种的几个例子中包括将兔子品种引入澳大利亚以及将斑马贻贝引入到北美的圣·劳伦斯河流系和大湖区。在遗传算法中，迁移算子要负责的任务是在子种群间实现个体的交换。这些任务包括:

419

- 选择迁民（Selecting the emigrants）
- 发送迁民（Sending the emigrants）
- 接收移民（Receiving the immigrants）
- 融合移民（Integrating the immigrants）

在消息传递并行环境中，发送和接受被选的个体相对容易实现，但对个体的选择和融合会提供有趣的结果。从每个子种群中选择最好的个体进行迁移并将已接收的个体融合到种群中以替代最差的个体将导致更快的收敛。所以通过从一个子种群中拷贝一些较适合的个体到另一子种群以代替最坏的个体将会达到更快的收敛。但是，以这种方式进行选择和融合最终将对种群施加太多的选择压力，从而可能阻碍结果的生成。此外，如果选择压力过大的话，可能使结果的改进趋向局部优化而非全局优化。

引入迁移也必将引入通信开销。如同使用消息传递的其他并行计算一样，当通信时间开始支配总的计算时间时，并行算法性能将受到影响。因此必须考虑从进程之间的通信信息的频率和总量。

（2）*迁移模型* 最流行的迁移模型方法是

- 孤岛模型（island model）
- 跳石模型（stepping-stone model）

在*孤岛模型*中，允许将个体发送给任何其他子种群，如图13-3所示。它不对个体可迁移到何处加以限制。在*跳石模型*中，对迁移作了限制，它只允许移民移入相邻子种群。图13-4对此概念作了说明。由于限制了移民可迁移的目的地数目，跳石模型可减少通信开销并且限制了消息数。孤岛模型由于允许更多迁移的自由，在某些方面代表了较好的自然模型。但是，这种模型在实现时会有显著的通信开销和延迟。

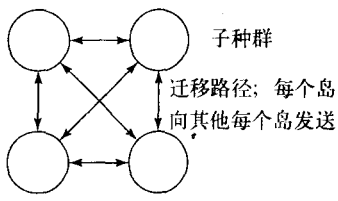


图13-3 孤岛模型

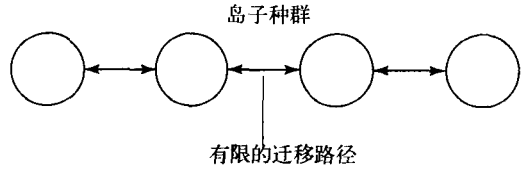


图13-4 跳石模型

420

2. 并行实现

下面的伪代码描述了从进程的实现:

```
set generation_num to 0;
initialize Population(generation_num);
evaluate fitness of Population(generation_num);
while (not termination_condition) {
   generation_num++;
   select Parents(generation_num) from
      Population(generation_num - 1);
   apply crossover to Parents(generation_num) to produce
      Offspring(generation_num);
   apply mutation to Offspring(generation_num) to get
      Population(generation_num);
   apply migration to Population(generation_num);
   evaluate Population(generation_num);
}
```

每个从程序将执行上述伪代码，并在生成结果后将被返回给主程序。在生成最后的结果之前，主程序将完成某些分析。

孤岛和跳石模型均有优缺点。最明显的优点是，除了每隔k代要进行“最好个体”的通信之外，两者都是自然易并行的。也就是说，使用p个处理器，每个独立地运行在(population_size/p)个个体子种群上，从而可潜在地使k代的计算加快p倍。

这种分离的并行性已被用来企图解释化石记录中明显断链的原因。[Cohoon et al., 1987]提出一个基于“不时被打断的平衡”(“punctuated equilibria”)理论的实现方法，它为在环境发生变化后在子种群中很快出现新品种的理论提供了基础。这些变化的出现，是由于迁移结果使新的遗传材料引入到种群后造成的。Cohoon等人观察到，当在分离的子种群间有大量的个体迁移后，很快就可找到新的解，正如化石纪录指明在相对较短时期内在自然中出现了重大发展一样。

由于每隔k代整个并行化将被中断，因此就可通信交流最适应个体的信息，用这种方法可
421 获得稍小于p倍的加速比。最适应个体的通信是与所有其他处理器都要进行还是只在“邻居”之间进行将会影响实际的加速比。处理器之间的通信体系结构也可能是影响因素之一。

在跳石模型中，由于将通信限于与最邻近的处理器间进行，而不许每个处理器与其他每个处理器通信，因而即使是k代后出现的处理器间通信也将减少。

如易并行方法的计算优点一样，它有一个潜在的缺点：种群子集相对分离的性质。这种子集的分离增加了使每个子集收敛于自己局部优化而非全局优化的可能性，而且也许会完全丢失全局解。此外，由于组成每个种群子集的个体数的减少，将会增加达到收敛所需的代数。

这里要补充一句，遗传算法中的这种分离/局部优化现象可以认为在大自然中也同样存在！研究板块地壳构造（地球地壳的大板块互相围绕移动、越过以及移入）的地质学家已提出了对过去的板块移动的计算机预测。其中一个板块与其他板块脱离接触，从而构成了澳大利亚的大陆块。生物学家长期以来已注意到该地区中存在的独特物种的数量，包括无翼鸟和大型跳跃哺乳动物。在分离地区中并行出现的进化并不一定导致每个地区中的类似解答！与此相反，可能的结果将是不同的局部优化。

当然，减小有效分离的机制，会增加处理器间的通信，从而引入了自身的缺点。算法中顺序（通信）部分的增加以及使用多个相互分离的多个处理器的实际加速比的减小就是这一缺陷的表现。

3. 公共种群的并行化

与分离子种群并行性不同，可采用的另一种方法是，在全局种群上并行实现遗传算子。例如，如果能用某种方法将个体上的信息提供给每个处理器；那么这些处理器就可并行地完成选择、交叉和突变。

很显然，如果我们使用竞赛选择，那么处理器可运行在独立的个体对上。类似地，每个处理器可独立地工作在被选个体的子集上以完成交叉和突变操作，并评估结果的适应性。不幸的是，虽然此方法能有效地并行化遗传操作，但它对加速整个计算几乎不起什么作用！与将种群信息分布到所有处理器所需时间加上选择过程的通信交换所需时间相比，整个计算时间主要依赖于计算各种操作（选择、交叉和突变）的相对时间。在许多情况下，遗传算子几乎不需要密集计算，因此整个过程的加速比将受实际通信的限制。

4. 共享存储器系统

如果个体被保存在共享存储器中，那么每个处理器在实行遗传算法的过程中都要访问种群并且可以读/修改个体。若将个体分配给指定的处理器，就可使存储器访问冲突减至最小，且不再需要同步，从而可避免大部分的顺序化的影响。除了会导致某种顺序化的代之间的同步外，每个处理器将与其他处理器并行地对它们的本地子种群进行计算。由于处理器间的通信本质上是访问存储器的速度，可期望在这种系统上求解遗传算法的时间可减小到与分布式处理器系统相当。 422

5. 分布式处理器系统

如果一个分布式处理器系统中的所有工作站处在同一子网中，那么它们处理器间的通信将争用网络资源。让每个处理器的最适合个体与其他处理器进行通信的简单任务将意味着争用网络带宽，并会导致串行化遗传算法中的一部分。通过使用分离的子种群使这些串行部分减至最小的方法，在历史上即是用来使并行化变得最大的方法。

另一方面，如果能做到将处理器分离成许多子网，那么就不必最小化子种群间的通信。为此，每个子网应独立于其他子网在它的处理器间进行通信。实际上，除了并行化（在各个处理器级）遗传算法算子所需的计算之外，上述方法至少可部分地并行化（在子网级）处理器间通信。

至少可实现部分通信并行化的另一种机制是将处理器互联成二维或三维环，即环形网。这就将通信限制在相邻互联的处理器之间进行，对应于分离子种群的跳石模型。通过将处理器间的通信限制在相邻子种群间的个体迁移，同时允许每个处理器与其他处理器并行地、独立地运行在自己的子种群上，那么就可以消除大部分的通信顺序化的影响。[Marin, Trelles-Salazar, and Sandoval, 1994]表明使用6个处理器便可获得近线性的加速比，并且还断言这一结果也将适合于具有更大规模的工作站网络。

13.4 连续求精

还有许多其他的搜索算法可用来获得我们所讨论的优化问题的好的解。最直观的方法之一是应用连续求精来搜索网格。开始时，所选择的网格空间允许使用蛮干（brute-force）考察方法来搜索容积，且只需要很少时间。在所讨论的问题的环境中，网格大小可以从每10/1000点开始考察。由于该容积的每一边的跨距是从−1 000 000到+1 000 000，则每次10 000的搜索增量将意味着只需评估大约200 × 200 × 200个点，或需完成略小于10^7个函数的评估。如果保留K个最好点，再以每个点为中心形成一个立方体，然后用更细的网格空间对每个点进行评估，例如使增量大小为100，那么此时就需完成大约$K \times 10^6$个附加的函数评估。

423 再次，为K个子容积中的每一个保留K个最好点并对网格大小为100^3的每一边进行穷举详尽搜索，则将导致大约$K^2 \times 10^6$个附加的函数评估。如果每次保留10个最好点，则该方法在完成大约10^8个函数评估后才会终止。虽然该方法比起对该容积中每个点进行完全穷举的考察来说，所需的评估数少了许多（10^8比10^{19}），但这仍是一个密集计算，且不能保证被定位的点可使函数得到优化。但是，对于遗传算法的求解来说，所需的总的函数的评估数一般要小好几个数量级（参见习题13-1）。

上述方法很易并行化。可将原始容积分成K个子容积，而K个处理器中的每一个以易并行方式在被分配的子容积中进行搜索。当算法的该部分搜索结束后，每个处理器将K个最好点的候选者送回给主进程。由主进程确定K^2个点中的哪K个是最好的并将它们再分配给各个从进程。每个从进程然后在自己的子容积中与其他子进程一起并行地搜索。

[Schraudolph and Belew, 1992]将连续求精概念应用到遗传算法。他们的方法使用相对较低的精度，如用5位来表示每一个坐标。当坐标的最高位显示出稳定时，就将它分离并保存起来，然后将余下的位左移一位，并加入一位新的最低位。这就相应于在每个坐标中将原来的间隔分为一半。此后它们就在已缩小的空间中重新开始遗传算法的优化操作。

13.5 爬山法（hill climbing）

另一个常用的搜索和优化方法与人们用来定位局部最大值的算法是相同的。例如在树林中被蒙住眼睛（或是在浓雾或暴风雪中徒步行走）徒步爬山寻找山顶。已有的经验可能提示我们有一种方法可达到山顶，即让移动的方向总是上升的，或至少永远不要朝下坡方向前进。无疑，这种方法会保证你绝不会走向比你目前所在位置的更低的地点，且根据地形最终你可到达树林中的最高处。

但是，很可能你最终到达的是一个小高地的顶部（局部最大值）。由于此时你周围的地形均低于你目前所在位置（而规则又不允许你向低处移动），因而你将无法跨过小山谷而爬上其他的山峰。仅在理想的情况下才能产生正确结果的算法显然没有太多的吸引力。那么，我们应如何来修改它？

如果我们跳出上述框框，即只有一个徒步者处在某一点，然后按照局部地形爬向局部的峰顶，而是将问题一般化为有n个徒步者从n个随机点出发爬山，就可使问题迎刃而解。如果每个徒步者按照爬升算法决不从他或她的起始位置向下移动，那么他们中的每一个都将到达一个局部的最高点。直观地，如果徒步爬山者越多且他们的出发点越分散，则至少他们中的一个到达树林中最高峰的可能性就越大。就算该方法不能保证徒步爬山者将到达最高峰，但只要增加爬山者的数目，并确保他们的出发点是随机的，则就可使不能达到实际顶峰的可能性变为任意小。

424 这实质上就是蒙特卡罗搜索技术的要点。如在3.2.3节中所提到的，该名称来自蒙特卡罗市的主要特征：赌博。概念上，我们“掷骰子”等价于一个爬高者产生一个随机的出发地点。当出发位置是随机选择时，一个“幸运”骰子点（爬山者的出发位置正好在最高峰的旁边）和一个“倒霉”骰子点（爬高者的出发位置正好在小山峰的旁边，从而排除了到达最高峰的可能）之间的差别是纯侥幸的，故此名称应是恰当的！

求解的实现可以用静态分配方法或是用动态工作池方法。在静态方法中，只需简单地在为不同的处理器划分要搜索的区域，并为每个处理器生成一个随机的出发点，此后跟随爬山者的移动到达局部最高点。每个处理器随后将它的查找结果返回给主进程，最后由主进程确定哪一个爬山者找到了最高点。动态方法则意识到各个爬山者在到达它们的局部峰顶之前，

有些爬山者将比另一些会有更长的遍历，或是说有些爬山者可能比其他的要爬得更快（相应于较快的处理器）。不论是哪一种方法，某些处理器可能在其他处理器之前完成。对该问题的一个使处理器处于忙碌状态的自然求解方法是，将欲搜索的区域分解成更小的区段，当一个处理器完成了目前正处理的区段并报告其结果后，再给它分配一个新的搜索区段。

13.5.1 银行业务应用问题

任何大公司在涉及公众的金融管理时经常碰到的问题是“我们客户应将款付到何处？”当大部分商务属本地时，回答很显然：“直接付给公司！”但当商务活动具有全国性时（且越来越多地具有国际性时），那么很显然上述的回答过于幼稚。

在客户向公司付款的日期和公司收到该存款的日期之间会有邮汇的延迟。此外，在公司收到付款和公司将该款存入银行之间还有一个存款处理的延迟。最后，银行通过它的全国和/或国际银行系统处理该款的收取和将此付款划给公司的银行账户之间又有一个延迟。在理想条件下（一个本地顾客的所开支票的银行恰好是该公司设有账户的银行），公司在顾客邮汇的当天收到付款，并在该银行结账前将该款存入，而在当晚的半夜该存款被划入到该公司的账户上。当条件不太理想时，国内的邮汇可能要有3~5天的延迟，正好错过银行当天的结账时间又会耽误一天，而在将付款资金划到账户时恰又赶上联邦结账的日期，这又将延误1~3天。从顾客最初的邮汇付款到公司在账户中接收到该付款的资金的延迟时间乘上付款的总金额，术语上称为是*浮动*（float）。当顾客和公司的所在地跨国时这种加权的平均收款的延迟，即*浮动天数*（float-days），还要增加。

这种4~9天甚至更多天数的浮动天数对单个客户来讲也许算不上什么（也许还会受欢迎！），但当每天也许涉及几千万美元的大量付款时，公司就会有不同的观点。如果利息的年利率为8%，那么对于每天一千万美元的付款，就相当于每年有80万美元的未清账款，这或者可成为公司的附加收入，或是可使公司不用去融资一千万美元。 425

意识到这一问题的严重性，一些国内和国际银行已提供了称为*锁箱*（lockbox）的分布式收集工具。为了收取付款，银行将在全美国以及与该公司有业务来往的其他国家中设置许多这种锁箱。银行将由专人管理每个锁箱设备，每天若干次从邮局检索公司的邮件，打开它并将付款存入到公司的账户。为此，公司将直接告知它的顾客，请他们邮寄付款到锁箱，从而就可达到最快回拢顾客资金的目的。如此，该优化问题就简化为该设置多少个锁箱以及在何处设置锁箱以使平均浮动为最小的问题。

通常的情况是，有n个城市可作为设置锁箱的场所；例如，单单在全美国设立超过100个的锁箱，而在世界其他地区则还要设置几百个。确定锁箱的优化数及设置场所的问题，概念上可简化为：

```
set Float_days to 1000;
for (i = 1; i <= MaxLockboxes; i++)
   for (all possible placements of i lockboxes) {
      compute CurrentFloat_days;
      if (Float_days > CurrentFloat_days) {
         Float_days = CurrentFloat_days;
         Save i and lockbox placement pattern;
      }
   }
```

这里的主要计算问题是，仅有n个位置可存放第一个锁箱，但却有$n(n-1)/2$种方法来定位它们中的两个，而要存放k个锁箱则共有$n!/((k!)(n-k)!)$种方法。考虑到仅有200个城市可作为存放

锁箱的候选地点，且仅有6个锁箱要存放在这些城市中，从而为计算平均浮动值大约要进行8×10^{10}锁箱位置的评估。此外，对每个锁箱布置的计算也并不简单。

由于两点之间邮汇的延迟依赖于邮寄出的日子是一星期中的哪一天，因此银行通常在计算中使用的是在不间断基础上所收集到的邮汇数据。其结果便是形成邮汇数据的一个三维数组，其规模为$n \times n \times 7$。由该数组的下标可识别邮汇账款的地区、邮汇至哪个地区以及在星期几汇出的。该数组中元素的值就代表了该款从邮汇到被收到之间所消费的时间，即平均邮汇延迟天数。

为替某公司评估所需锁箱的最佳个数及存放地点，必须知道应收账款（传递给公司的付款）的模式。银行记录公司应收账款的重要的统计子集，并将它们按来源地点、盖邮戳的星期日期以及付款金额加以分类。

在最简单的情况下，对特定的锁箱安放位置的平均浮动天数的评估首先是要为每个来源区域分配一个锁箱。邮汇目的地的选择则基于被采集到的应收账款的数据，且为了避免混淆，在一个地区中的每个顾客将被告知应将款付到同一锁箱。对每个来源区，则应选用对公司实
426 际应收账款模式来讲具有最小平均邮汇天数的目的地锁箱来接收付款。在这种收款模式中，将对每笔付款用它的总额和被汇出的日期加以加权。下面列出的是计算一个特定k个锁箱选址模式的平均浮动天数的算法：

```
TotalFloat = 0;
for all_receivables {
   accumulate amount_date-mailed weighted mail delay to each lockbox;
   accumulate each_region's_total_payments;
   accumulate number_of_items_mailed_from_region;
   accumulate TotalReceivables;
}
for each_region {
   determine destination_lockbox;
   determine AverageMailDelay;
}
for (i = 1; i <= n; i++)
   TotalFloat = TotalFloat + Total_amount_i * AverageMailDelay_i;
CurrentFloat_days = TotalFloat/TotalReceivables;
```

如果对特定锁箱选址模式的浮动天数的评估可在100 ms 内完成（相当乐观的），那么在200个可能场所中为6个锁箱选址的所有可能的评估将约需2000年。显然，即使一个并行化的改进可加快1000倍，仍不足以使所述算法有任何实际意义。

13.5.2 爬山法在金融业务中的应用

如果从不同角度来处理这一问题，则就有可能在一个较为合理的计算时间内找到一个好的解。如果不是用穷尽方法去评估n个可能场所中布置k个锁箱的每一种可能，而是把它视为一个k维地形并使用爬山方法，那么就会使情况大为改善。

就如徒步爬山者在一个x-y的平面中移步那样，不管是往哪个方向移动总是会在高度上得到最大的局部改善，那么该问题就可变为确定使浮动天数有最大减少的锁箱位置中增量变化的问题。借助增量地改变所在地，最终就可得到代表浮动天数局部最小值的解。

如同徒步爬山者的例子一样，随机地生成m个初始出发点。在这种情况下，它们就相当于为k个锁箱选定m种布局。对于每一个初始的k个锁箱的选择模式，我们将确定一个锁箱位置的

哪一个变化会使浮动天数有最大的减少，然后就采用哪种变化，重复地进行这一个过程。

```
for (i = 1; i <= m; i++) {
   Generate random siting pattern, p_m, of k lockboxes
      from n possibilities;
   Compute fd = FloatDays(p_m);   /* float for starting pattern */
   for (j = 1; j <= k; j++) {     /* vary each of the K lockboxes */
      x = 1;                      /* set a continuation flag */
      while (x != 0) {            /* loop while improvement is possible */
         x = 0;                   /* clear flag */
         for(s=1;s<=(n-k);s++){   /*try all possible lockbox j placements*/
            Evaluate fd_Temp as the FloatDays of placements of lockbox j;
            if (fd_Temp < fd) {   /* see whether float improved*/
               x = 1;             /* keep looping if improvement was made*/
               fd = fd_Temp;      /* save float_days for this placement */
               Save placement_s;  /* save placement that was responsible */
            }
         }
      }
   }
}
```

由于m是一个常数，类似于独立寻找山顶的徒步爬山者人数，因此在while循环中涉及$O(n)$项。在while中的循环数最坏将是指数式的而不是阶乘的，而在实际中则更优于这一情况，即大多数的局部优化解只需几十次循环就可得到。

13.5.3 并行化

如前面所指出的那样，该问题的并行化是相当直截了当的。我们可以对p个进程中的每一个，分布不同的“种子”，并使每一个生成k个锁箱的m/p个随机布局，或是用前面讨论过的工作池方法。在后一方法中，当进程完成它们的最初设置后，每个进程将请求给予额外的起始布局以继续工作。

13.6 小结

本章介绍了以下概念:

- 分支限界技术
- 并行遗传算法
- 连续求精法
- 爬山法

本章还介绍了以下重要的遗传算法的术语:

- 交叉
- 突变

推荐读物

在本章的分支限界这一节中，我们没有探讨特定问题，而只是简单地概述了这一基本技术。背包问题通常用来描述搜索和优化技术，在许多教科书中都可找到，甚至有一本专著全部讨论这一问题[Martello and Toth, 1990]。类似地，在[Lawler et al., 1985]中对旅行商问题

做了全面的论述。

在[Lawler and Wood，1996]对分支限界问题做了早期的评述。有关并行的分支限界方法的探讨可在[Lai and Sprauge，1985]、[Kumar and Kanal，1984]、[Wah，Li, and Yu，1985]中找到。[Wah and Ma，1982]和[Mohan，1983]对并行的最佳优先策略进行了研究。各种并行图和树搜索算法可在[Quinn and Deo，1984]中找到。

[Michalewicz，1996]对遗传算法作了很好的介绍。其他的有关教科书包括[Bäck，1995]、[Fogel，1995]、[Goldberg，1989]和[Mitchel，1996]。有关并行遗传算法的进一步的信息可在[Chipperfield and Fleming，1996]中找到。在[Biethahn and Nissan，1995]中论及了管理的应用，而有关工程的应用则可参见[Dasgupta and Michalewicz，1997]。此外，有好几本刊物讨论遗传算法、并行搜索和优化，其中包括*IEEE Transactions on Evolutionary Computation*。

参考文献

BÄCK, T. (1995), *Evolutionary Algorithms in Theory and Practice,* Oxford University Press, New York.

BÄCK, T., D. B. FOGEL, AND Z. MICHALEWICZ, editors (1997), *Handbook of Evolutionary Computation,* Oxford University Press, New York.

BIETHAHN, J., AND V. NISSAN, editors (1995), *Evolutionary Algorithms in Management Applications,* Springer-Verlag, Berlin, Germany.

CHIPPERFIELD, A., AND P. FLEMING (1996), "Parallel Genetic Algorithms," in *Parallel and Distributed Computing Handbook*, A. Y. H. Zomaya, editor, McGraw-Hill, NY.

COHOON, J., S. HEDGE, W. MARTIN, AND D. RICHARDS (1987), "Punctuated Equilibria: A Parallel Genetic Algorithm," in *Proc. Second Int. Conf. on Genetic Algorithms*, J. J. Grefenstette editor, Lawrence Erlbaum Associates, Hillsdale, NJ.

DASGUPTA, D., AND Z. MICHALEWICZ, editors (1997), *Evolutionary Algorithms for Engineering Applications,* Springer-Verlag, Berlin, Germany.

FOGEL, D. B. (1995), *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence,* IEEE Press, Piscataway, NJ.

GOLDBERG, D. E. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley, Reading, MA.

HOROWITZ, E., AND S. SAHNI (1978), *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD.

JANAKIRAM, V. K., D. P. AGRAWAL, AND R. MEHROTRA (1988), "A Randomized Parallel Backtracking Algorithm," *IEEE Trans. Comput.*, Vol. 37, No. 12, pp. 1665–1676.

KUMAR, V., AND L. KANAL (1984) "Parallel Branch-and-Bound Formulations for AND/OR tree Search," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-6, No. 6 (November), pp. 768–778.

LAI, T.-H., AND S. SAHNI (1984), "Anomalies in Parallel Branch-and-Bound Algorithms," *Comm. ACM*, Vol. 27, No. 6, pp. 594–602.

LAI, T.-H., AND A. SPRAGUE (1985), "Performance of Parallel Branch-and-Bound Algorithms," *IEEE Trans. Comput.*, Vol. C-34, No. 10, pp. 962–964. Also see extended paper in *Proc. 1985 Int. Conf. Par. Proc.*, pp. 194–201.

LAWLER, E. L., J. K. LENSTRA, A. H. G. R. KAN, AND D. B. SHMOYS, editors (1985), *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, New York.

LAWLER, E. L., AND D. E. WOOD (1966), "Branch-and-Bound Methods: A Survey," *Operations Research*, Vol. 14, pp. 699–719.

LI, G.-J., AND B. W. WAH (1986), "Coping with Anomalies in Parallel Branch-and-Bound Algorithms," *IEEE Trans. Comput.*, Vol. C-35, No. 6, pp. 568–573.

MARIN, F., O. TRELLES-SALAZAR, AND F. SANDOVAL (1994), "Genetic Algorithms on Lan-Message

Passing Architectures Using PVM: Application to the Routing Problem," in *Parallel Problem Solving from Nature–PPSN III, Lecture Notes in Computer Science*, Y. Davidor, H. P. Schwefel, and R. Manner, editors, Vol. 866, Springer-Verlag, Berlin, Germany.

MARTELLO, S., AND P. TOTH (1990), *Knapsack Problems: Algorithms and Computer Implementation*, Wiley, Chichester, England.

MICHALEWICZ, Z. (1996), *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd edition, Springer-Verlag, Berlin, Germany.

MITCHEL, M. (1996), *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA.

MOHAN, J. (1983), "Experiences with Two Parallel Programs Solving the Traveling Salesman Problem," *Proc. 1983 Int. Conf. Parallel Processing*, pp. 191–193.

QUINN, M. J. (1994), *Parallel Computing, Theory and Practice*, 2nd edition, McGraw-Hill, New York.

QUINN, M. J., AND N. DEO (1984), "Parallel Graph Algorithms," *Computing Surveys*, Vol. 16, No. 3 (Sept.), pp. 319–348.

RAO, V. N., AND V KUMAR (1988), "Concurrent Access of Priority Queues," *IEEE Trans. Comput.*, Vol. 37, No. 12, pp. 1657–1665.

SCHRAUDOLPH, N., AND R. BELEW (1992), "Dynamic Parameter Encoding for Genetic Algorithms," *Machine Learning*, Vol. 9, No. 1, pp. 9–21.

SCHWEFEL, H.-P. (1995), *Evolution and Optimum-Seeking*, Wiley, Chichester, England.

WAH, B. W., G.-J. LI, AND C. F. YU (1985), "Multiprocessing of Combinational Search Problems," *Computer*, Vol. 18, No. 6, pp. 93–108.

WAH, B. W., AND Y. W. MA (1982), "MANIP — A Parallel Computer System for Implementing Branch-and-Bound Algorithms," *Proc. 8th Ann. Int. Symp. Comput. Architecture*, pp. 239–262.

WITTE, E. E., R. D. CHAMBERLAIN, AND M. A. FRANKLIN (1991), "Parallel Simulated Annealing Using Speculative Computing," *IEEE Trans. Parallel and Distributed Systems*, Vol. 2, No. 4, pp. 483–494.

习题

科学/数值习题

13-1 编写求解8皇后问题的并行程序，即在8 × 8棋盘上，找出8个皇后的安放位置，使得没有两个皇后会互相攻击（不在同一列或对角线上）。由于每一个皇后必须放在棋盘的不同行上，因此每个解可用每个皇后的所在列的位置来表示，即用一个8元（x_1，x_2，x_3，…，x_i，…，x_8）来表示，其中x_i是皇后i所在列的位置。x_i的约束条件为$1 \leqslant x_i \leqslant 8$。还应注意的是，两个皇后的所在位置$(a, b)$和$(c, d)$不能处在同一对角线上，即不能有$a-b=c-d$或$a+b=c+d$。 429 ~ 430

13-2 用遗传算法方法编写一个并行程序以寻找函数$f(x, y, z) = -x^2 + 1\ 000\ 000x - y^2 - 40\ 000y - z^2$（13.3.3节中所使用的）的最大值。修改此程序以使用户可输入任何三维函数。对代码加以编排，以确定评估$f(x, y, z)$程序所需的次数。

13-3 用13.4中所讨论的连续求精算法，编写寻找上题函数中的最大值的并行程序。

现实生活习题

13-4 Shortpath房地产公司的总办公室中有一组互联的PC。Shortpath需要一个并行程序使它的房地产代理商能设计一个展览房屋的计划表，它能使巡视所需时间减至最小。试开发恰当的测试数据和程序。

13-5　迈克想要进行环绕欧洲的旅行直到他发现有许多看起来非常诱人的糕点和面包的Bäckerei点心店。为了乘火车旅行，他需要在小旅行袋中装吃的东西，但有以下一些限制:

1）他要很多品种，但每样不能超过两个。

2）他需要最大量的卡路里，但这不一定与食品容积有关。

3）所装入的食品不能超过旅行袋的容积极限。

开发一个并行程序，从描述Bäckerei销售食品的5元组（长度，宽度，厚度，食品类型，卡路里值）集合中很快地挑出他旅行所需的各种食品。除了编写程序外，你还需开发一个有20个数据项的集合，并用它来检验你的程序。

13-6　给定一组矩形，其中第i个矩形的面积为A_i，试编写一个并行程序在一个大矩形中定位这组矩形，但要遵从以下的约束条件:

1）各个矩形必须没有重叠。

2）所有n个较小的矩形必须为一个大的矩形所包含。

3）较大的矩形的面积应是最小的。

（注意: 该问题是集成电路布局问题的简化版。更复杂的布局问题有与较小的矩形间的关系相关的更多约束条件。）

13-7　Nat-Ex是一个全国性的包裹传递公司，它正在对它的收集和散发包裹的“集散中心”的布点进行重新评估。理想的布点应是将这些集散中心布放在全国各枢纽地，使成本和传递时间可减至最小。你被委任从事对这些集散中心可能分布在不同地点的研究，并基于遗传算法编写一个并行程序。你可以假设被收的包裹数量将正比于人口数量，且作为初步近似，只考虑布点在大城市。编写该程序，并确定恰当的输入数据和约束条件。其中一个约束条件便是集散中心的数量。

13-8　与习题13-7相当类似的问题是航线中心的选址。试编写一个并行程序为一条航线的中心点在全国的布点找到最好的位置，使该航线能获得最大的收益。同样你必须确定输入数据和约束条件。

431

13-9　最近发现的小行星Geometrica具有特别异常的表面。通过各种观察，它的表面可用以下公式加以模仿:

$$h = 35\,000\sin(3\theta)\sin(2\rho) + 9700\cos(10\theta)\cos(20\rho) - 800\sin(25\theta + 0.03\pi) + 550\cos(\rho + 0.2\pi)$$

其中h是高于或低于海平面的高度，θ是赤道平面角（定义了地球上的经度），而ρ是极平面角（定义了地球上的纬度）。

1）用爬山法编写一个顺序程序以找出在Geometrica表面海平面上最高点处的位(θ, ρ)位置。

2）为第1问开发一个易并行求解方法。

3）为第1问开发采用工作池的并行求解方法，假设条件是用来求解该问题所使用的各工作站的能力有很大差异。

4）对用前面三种方法求解所需的模拟时间进行比较。

13-10　一个常碰到的问题称为“箱子打包（bin packing）”，它要解决的问题是要将k个有不同特征的物体放到小于k个的类目即“箱子”中。尽管这是一个在节日里向亲朋好友赠送一套礼品时（你不得不寻找能装得下所有物品的最便宜的盒子）或在工业中（必须对一组有不同能力和速度的机器加以分配，以优化地生产不同的产品）经常碰到的

问题，但你被指定来解决一个不同的问题。这已引起有魄力（且富有）人的注意，即21点blackjack游戏也适用此模型：每个游戏者试图得到从2张到10张直至更多张纸牌，而它们的值加起来不能超过21点但又与庄家的牌值和相同或是比庄家的更高。所有前面已发牌的值均是已知的，此外，一副牌中每种值的数量也是已知的。你被雇佣来首先提出一个顺序算法以计算出你要的下一张牌会“崩”掉你手上牌的可能性（导致值总和超过21点），然后再设计一个并行算法。

13-11 有一个有魄力的企业家已断定他有一个绝好的机会去将他对狗的喜爱与巨大财产组合在一起，从而可垄断新的狗品种Softie的市场。Softie的特征如下：

A. 毛的长度：8英寸⊖或更长

B. 毛的特征：特别柔软（或较软）
白底上有棕色
白爪
黑尾

C. 尾巴特征：短（4～6英寸）
尾端挺直向上

D. 重量：特别重：重量等于或大于90公斤

E. 脚的特征：爪印面积超过9平方英寸⊜
趾间全有蹼

F.性情：特别温和

当对所有的狗进行观察时，这些特征中的每一个均有一个范围：

A. 可用8位二进制数表示，其中00000000相应于无毛狗。11111111相应于毛的长度为10.2英寸，而11001000对应于8英寸。 432

B. 可用6位来表示柔软性（其中000000相应于最柔软级别），000111表示非常软，而111111表示最硬；用3位表示底毛的亮度，000表示亮，111表示暗淡；用3位表示底毛颜色，000为白色，001为棕色，而所有其他颜色由其余六种组合表示；用3位表示前景亮度（000为亮）；用3位表示前景颜色（000为白，001为棕，010为红，011为黄，111为黑，其余的位模式组合表示其他颜色）；用1位表示爪的颜色，0对应于白色，而1为其他任意颜色；用1位表示尾巴颜色，0对应于任何其他颜色，而1表示黑色。

C. 可用10位表示，其中8位表示尾巴长度（00000000相应于无尾巴，而11111111相应于尾巴长为25.5英寸或更长），其余2位用来说明尾巴的外观，00表示尾端向上，01对应于尾端水平，10对应于尾端向下，而11对应于最不受欢迎的卷尾外貌。

D. 可用10位表示，其中前面7位对应于以公斤表示的重量，而后3位对应于1/8公斤的增量。因此，若重量特征位为0000101011，则重量为 $5\frac{3}{8}$ 公斤。

E. 可用10位表示，其中7位表示爪印面积，其余3位表示蹼的比例。在本例中，0000000对应于爪印面积为0.5平方英寸，1111111对应于爪印面积为13.2平方英寸，而0100011对应于爪印面积为4.0平方英寸。在后3位中，000表示1/8蹼，而111对应于全蹼。

⊖ 1英寸 = 0.025m

⊜ 1平方英寸 = $1.639 \times 10^{-5}m^3$

F. 可用6位表示，其中000000表示最温和的脾气禀性，000100为非常温和性格，而111111则表示“比垃圾狗的脾气还坏”，即极端不温和的脾气!

这样，总共需64位来表示该种狗中的每一条狗。一条狗可能用以下的64位来表示:

11001000 001011 000000 111001 01 1000000011 001101100 0100011 011 000100

其特征为:

11001000	毛长8英寸
001011	毛比所期望的稍硬
000000	底毛颜色为亮白
111000	前景颜色为暗棕
01	白爪；黑尾
10000000	尾长12.8英寸，卷尾
001101100	重13.5公斤
0100011	爪印面积4平方英寸
011	50%蹼
000100	非常温和的脾气

简言之，该狗有某些令人满意的特性（毛相当好，爪的大小也不错，此外脾气也好）以及某些不尽如人意的特性（毛的硬度稍超过标准，尾巴是卷曲的，与标准相比该狗相当小，此外该狗的水性不如水狗（water dog），因为它不是完全连蹼的）。

433

该企业家焦急地对该项目进行两面下注以防损失，为此他雇用你和另一个你的竞争者一起来开发Softies。你的竞争对手用一个大型的综合养狗场，计划喂养500条狗；而你则选择5个独立的养狗场，每个喂养100条狗。你和你的竞争者都计划去访问动物掩蔽所并带走所找到的500条未切除卵巢/已阉割的狗（即你们将以具有“随机”特性的种群开始）。

你和你的竞争对手在培育连续几代狗的同时，将始终保持你们养狗场中的种群大小是固定的。你们两个均计划使每个养狗场中保留当前一代中符合培育Softie品种标准的它的两个双亲。此外，由于一对双亲可能一胎生下多个狗崽（相应于进行多次“切割”和交叉），你们两个均计划把那些最不符合Softie标准的狗崽送掉（送给你们的朋友），使你们的局部种群大小保持固定。

你和你的竞争对手所使用的两种方法之间的主要不同是，他有一个单一的大种群，而你有5个独立的养狗场的“承包者”，且每一个均运作在孤立的子种群上。他将简单地生成n代的狗崽；而你的承包者们的每一个将孤立地生成5代狗崽，然后他们只将最合适的两条狗送给他们的邻居（A送给B和E各一条，而B送给A和C各一条，等等），如同孤岛模型那样。

1）按照你的竞争对手的方法，编写一个单处理机的遗传算法的求解。选几个随机生成的初始种群对其加以运行，并计算在种群中的10%（即50条狗）满足Softies标准之前所需的平均代数。

2）按照你的方法编写一个多处理机的遗传算法。如同问题1一样，选几个随机生成的初始种群对其加以运行，并计算在你的5个养狗场中，当总共有50条狗满足Softies标准时所需的平均代数。

提示：你将需要构造一个能体现所有这些特性的评估函数，使得当具有符合Softies“标准”特性的狗比起那些不符合的来，将显示出有更大的“适合性”。

13-12 新建立国家Nella的1535位参议员和众议员正在考虑从一家大公司接收“软钱”（间接给予竞选人的竞选费用）作为竞选活动的财政捐助。捐款是以称为Kerf的当地货币作为单位的，它来自于在继续销售其产品的问题上正面临某些管理上麻烦的一家大公司。可以肯定的是，每位参议员和众议员对此捐款有自己的评判标准。例如，有的议员认为应将来自该公司的所有捐款如数退回（肯定这是“少数人观点”）！另外一些议员觉得按照他正在进行的一流工作应该体面地接受所有捐款。再有一些议员可能会有不同的接受上限，超过此上限的捐款就将拒绝接受，因为他们担心如果无上限地接受捐款会在涉及该公司政策的投票中出现不公正的偏袒。

尽管“潜在获益”对公司来讲是一个值得争论的事实，但公司相信某些参议员或众议员比起其他议员来更易受影响，所以公司认为通过捐款尽可能地“支持”这些议员是非常重要的。不幸的是，在与管理代理人进行争斗后，公司已没有足够的余钱来全力“支持”所有的国会代表可接受的捐款的上限值。为此，公司必须对捐款的分配预算做更进一步的选择。

434

公司的政治顾问和律师们已设立了他们很自信的一个评估函数，该函数描述了每个国会成员的作为竞选活动Kerf的支持兴趣。SI（C_i，K_i）函数不但考虑了各人的捐款接受上限（阈值），而且还对国会议员由于接受捐款的影响在整个立法过程中所表现出来的对“公司的友情”的期望进行加权。到此为止，公司所缺少的是一个针对SI（C_i，K_i）函数在个人间进行Kerf特定分配的机制。作为一名遗传算法应用的著名专家，你被该公司雇佣来找出对整个国会成员进行Kerf分配的优化方案，当然该方案必须受公司预算额度的约束。

在略作考虑后，你设计了一个串行的遗传算法。就遗传算法角度而言，每个个体是一组1500Kerf的金额，它们的总和即是公司在这方面的预算。你随机地生成（但要遵从预算额度）Kerf的金额分配给1535位国会议员的每一位，从而产生一个遗传算法的个体，对此过程重复以产生1000个竞选活动捐款的分配方案：1000个遗传算法的个体。幸运的是，你很快就意识到为了及时地找到在国会成员中优化分配Kerf的方案以便在公司当天稍后时间的报告中提出该方案需要将方案的计算加速10倍。

设计一个并行遗传算法方法，该方法将使用联网工作站中那些由于不利的管理活动而导致裁减雇员的办公桌上闲置的机器。

13-13 在国际银行界最近几次的联合之后，其中最大银行之一——世界银行遇到了问题。它每天要处理大约350亿次信用卡交易，而且发现尽管它的中央计算机仍有能力记录交易和处理每月的结算单，但已没有多余的CPU时间来分析顾客消费的格局。世界银行的董事会注意到你有使用联网工作站来解决银行所面临问题的能力。

特别是，世界银行已有包括去年信用卡购货历史的超过8000T字节，即800万吉字节的在线存储纪录。这些数据是原始的购货点信息，没有进行过任何分类。每一项中包括有ID号、顾客账号、日期和总额。世界银行的国际市场部提议对这些历史数据进行挖掘并以各种方法将它们加以组织以使世界银行能获得更多的收益。

作为你的全部工作，世界银行要求你设计一种分析数据的并行计算方法。你要生成一张邮寄表，该表中要列出所有在去年内在任何家庭产品中心联盟处累计消费总额超过500美元的顾客名单。世界银行市场部相信，它可以将此邮寄清单（所有“好的”家庭产品中心顾客）的拷贝出售给这些商店中的每一家，而商店然后会使用该清单向那些顾客邮寄广告单。在不停止对所涉及的隐私进行分析的同时，你立即着手根据家

庭产品中心所标识的业务清单对顾客的购货进行累加统计的工作。

草拟出一个使用几千台闲置工作站（由于最近的合并而导致工作人员裁减而使这一情况成为可能）完成这一统计的算法。具体地讲，即要确定一个典型的联网工作站将要做什么，如何使工作站协调工作以及如何将结果融合在一起以生成所希望的邮寄表。你应记得的是，每个工作站只有非常有限的本地存储器容量：RAM为512MB，硬盘为48GB。任何企图需要超过任何单台工作站所具有存储容量的方法将指定会落得失败下场!

435

13-14　将在一个容积中搜索具有最大评估函数值点的遗传算法的例子转换成爬山问题，实现该解法并与一个等价的精确遗传算法求解方法所需时间进行比较。

1）作为顺序方法比较。

2）作为并行方法比较。

13-15　将求解银行应用的爬山方法转换成遗传算法问题，实现该解法并在所需时间上与一个等价的精确爬山求解方法相比较。

436

1）作为顺序方法比较。

2）作为并行方法比较。

附录A 基本的MPI例程

下面是MPI例程的一个集合，有了这些例程足以理解本书中的大多数的程序。MPI中提供了大量的例程。这里描述的例程被分为初等（用来建立环境和相关事宜）、基本点对点消息传递和集合消息传递三类。有关例程的全集和附加细节可在[Gropp, Lusk, and Skjellum，1999]、[Gropp, Lusk, and Thakur，1999]和[Gropp, et al.，1998]中找到。

A.1 初等例程

```
int MPI_Init(int *argc, char **argv[])
```

动作：初始化MPI的环境。

参数：`*argc` 来自`main()`的参量

`**argv[]` 来自`main()`的参量

```
int MPI_Finalize(void)
```

动作：终止MPI的执行环境。

参数：无。

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

动作：确定进程在通信子中的序号。

参数：`comm` 通信子

`*rank` 序号(返回的)

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

动作：确定与通信子关联的组的大小。

参数：`comm` 通信子

`*size` 组的大小(返回的)

```
double MPI_Wtime(void)
```

动作：以秒为单位返回从过去某点开始的执行时间。

参数：无。

A.2 点对点消息传递

MPI为`MPI_Datatype`定义了各种数据类型，大部分与C的数据类型相对应，包括

`MPI_CHAR`	`signed char`
`MPI_INT`	`signed int`
`MPI_FLOAT`	`float`

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

动作：发送消息（阻塞）。

参数：

*buf	发送缓冲区
count	缓冲区内的项数
datatype	项的数据类型
dest	目的进程序号
tag	消息标志
comm	通信子

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Status *status)
```

动作：接收消息（阻塞）。

参数：

*buf	接收缓冲区（已装载）
count	缓冲区内最大项数
datatype	项的数据类型
source	源进程序号
tag	消息标志
comm	通信子
*status	状态（返回的）

在接收例程中，tag内的MPI_ANY_TAG和source内的MPI_ANY_SOURCE为通配符。返回的状态是一个至少有三个成员的结构：

status->	MPI_SOURCE消息源的序号
status-> MPI_TAG	消息源的标志
status-> MPI_ERROR	潜在的错误

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request)
```

动作：启动非阻塞发送。

参数：

*buf	发送缓冲区
count	缓冲区内元素个数
datatype	元素的数据类型
dest	目的进程序号
tag	消息标志
comm	通信子
*request	请求句柄（返回的）

相关的例程：

MPI_Ibsend()	启动缓冲的非阻塞发送
MPI_Irsend()	启动就绪的非阻塞发送
MPI_Issend()	启动同步的非阻塞发送

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)
```

动作：启动非阻塞接收。

参数：`*buf` 接收缓冲区地址（已装载）
`count` 缓冲区内元素个数
`datatype` 元素的数据类型
`source` 源进程序号
`tag` 消息标志
`comm` 通信子
`*request` 请求句柄（返回）

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

动作：等待MPI发送或接收结束，然后返回。

参数：
`*request` 请求句柄
`*status` 状态（如果等待，则同`MPI_recv()`的返回状态）

相关的例程：
`MPI_Waitall()` 等待所有进程结束（附加参数）
`MPI_Waitany()` 等待任一进程结束（附加参数）
`MPI_Waitsome()` 等待某些进程结束（附加参数）

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

动作：测试非阻塞操作是否结束。

参数：`*request` 请求句柄
`*flag` 如果操作结束便为真（返回的）
`*status` 状态（返回的）

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

动作：对消息进行阻塞测试（不接收消息）。

参数：`source` 源进程序号
`tag` 消息标志
`comm` 通信子
`*status` 状态（返回的）

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Comm
*status)
```

动作：对消息进行非阻塞测试（不接收消息）。

参数：`source` 源进程序号
`tag` 消息标志
`comm` 通信子

*flag　　如果有消息便为真（返回的）
*status　状态（返回的）

A.3 组例程

```
int MPI_Barrier(MPI_Comm comm)
```

动作：阻塞进程直至所有进程已调用该例程。
参数：comm　　通信子

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm
comm)
```

动作：从根进程向comm中的所有进程和自身广播消息。
参数：*buf　　发送缓冲区(已装载)
count　　缓冲区内的项数
datatype　缓冲区的数据类型
root　　根进程的序号

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

动作：从所有进程向所有进程发送消息。
参数：*sendbuf　发送缓冲区
sendcount　发送缓冲区内元素数
sendtype　发送元素的数据类型
*recvbuf　接收缓冲区(已装载)
recvcount　每次接收的元素数
recvtype　接收元素的数据类型
comm　　通信子
相关的例程:
MPI_Alltoallv()　带偏移地向所有进程发送数据

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

动作：集中进程组的各个值。
参数：*sendbuf　发送缓冲区
sendcount　发送缓冲区内元素数
sendtype　发送元素的数据类型
*recvbuf　接收缓冲区(已装载)
recvcount　每次接收的元素数
recvtype　接收元素的数据类型
root　　接收进程的序号
comm　　通信子
相关的例程:

MPI_Allgather() 集中值并向所有进程分发
MPI_Gatherv() 集中值到指定单元中
MPI_Allgatherv() 集中值到指定单元中并向所有进程分发

MPI_Gatherv()和MPI_Allgatherv()在recvcount后需附加一个参数:*displs——数组偏移

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

动作: 从根进程缓冲区中分散部分值到进程组。
参数: *sendbuf 发送缓冲区
sendcount 每个进程发送的元素数
sendtype 元素的数据类型
*recvbuf 接收缓冲区(已装载)
recvcount 接收缓冲区的元素数
recvtype 接收元素的类型
root 根进程的序号
comm 通信子
相关的例程:
MPI_Scatterv() 分散根进程缓冲区中的指定部分值到进程组。
MPI_Reduce_scatter()归约值并分散结果。

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm comm)
```

动作: 将所有进程值归约为一个值。
参数: *sendbuf 发送缓冲区地址
*recvbuf 接收缓冲区地址
count 发送缓冲区内元素数
datatype 发送元素的数据类型
op 归约操作。其中包括:
MPI_MAX 最大
MPI_MIN 最小
MPI_SUM 求和
MPI_PROD 求积
root 结果的根进程的序号
comm 通信子
相关的例程:
MPI_Allreduce() 归约成单个值并返回给所有进程。

参考文献

GROPP, W., S. HUSS-LEDERMAN, A. LUMSDAINE, E. LUSK, B. NITZBERG, W. SAPHIR, AND M. SNIR (1998), *MPI—The Complete Reference*, Volume 2, *The MPI-2 Extensions*, MIT Press, Cambridge,

MA.

GROPP, W., E. LUSK, AND A. SKJELLUM (1999), *Using MPI Portable Parallel Programming with the Message-Passing Interface*, 2nd edition, MIT Press, Cambridge, MA.

GROPP, W., E. LUSK, AND R. THAKUR (1999), *Using MPI-2 Advanced Features of the Message-Passing Interface*, MIT Press, Cambridge, MA.

SNIR, M., S. W. OTTO, S. HUSS-LEDERMAN, D. W. WALKER, AND J. DONGARRA (1998), *MPI—The Complete Reference*, Volume 1, *The MPI Core*, MIT Press, Cambridge, MA.

附录B　基本的Pthread例程

下面是Pthread例程的一个集合，有了这些例程足以理解本书中的大多数的程序。有关例程的附加细节可在[Butenhof，1997]、[Kleiman, Shah, and Smaalders，1996]、[Nichols, Buttlar, and Farrell，1996]，以及[Prasad，1997]中找到。

B.1　线程管理

头文件<pthread.h>中含有类型定义(pthread_t等)和函数原型。

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
*(*routine)(void *), void *arg)
```

动作：创建线程。
参数：thread　线程标识符(返回的)
attr　线程属性(NULL为默认属性)
routine　新线程例程

```
void pthread_exit(void *value)
```

动作：终止调用线程。
参数：value　将值返回给已启动pthread_join()的线程

```
int pthread_join(pthread_t thread, void **value)
```

动作：使线程等待指定线程的终止。
参数：thread　线程标识符（返回的）
value　新线程例程

```
int pthread_detach(pthread_t thread)
```

动作：释放一个线程。
参数：thread　欲释放的线程

```
int pthread_attr_init(pthread_attr_t *attr)
```

动作：初始化一个线程的属性对象为默认值。
参数：attr　线程属性对象

```
int pthread_attrsetdetachedstate(pthread_attr_t *attr, int state)
```

动作：说明是否将释放一个以attr创建的线程。
参数：attr　线程属性
state　未释放-PTHREAD_CREATE_JOINABLE
已释放-PTHREAD_CREATE_DETACHED

```
int pthread_attr_destroy(pthread_attr_t *attr)
```

动作：取消一个线程的属性对象。

参数：attr　　线程属性对象

```
pthread_t pthread_self(void)
```

动作：返回调用线程的ID。

参数：无

```
int pthread_equal(pthread_t thread1, pthread_t thread2)
```

动作：比较两个线程thread1和thread2的ID，如果相等返回零，否则返回非零。

参数：thread1　线程

thread2　线程

```
int pthread_once(pthread_once_t *once_ctr, void (*once_rtn) void)
```

动作：如果指定的线程在以前未被调用过，则执行此例程。它可保证该例程只被调用一次。这对初始化很有用。例如，互斥锁应只被初始化一次。

参数：once_ctr 在一个全局变量被初始化成PTHREAD_ONCE_INIT（例如，static pthread_once_t once_ctr =PTHREAD_ONCE_INIT; ）以前，用该变量来判断例程once_routine是否已被调用过

once_rtn 例程只执行一次

B.2 线程同步

1. 互斥锁（mutex lock）

```
int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutexattr_t *attr)
```

动作：用指定的属性初始化互斥锁。

参数：mutex　　互斥锁

attr　　属性——NULL时为默认值

```
int pthread_ mutex_destroy(pthread_mutex_t *mutex)
```

动作：撤销一个互斥锁。

参数：mutex　　互斥锁

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

动作：对未加锁的互斥锁加锁（并成为互斥锁的拥有者）。如果已加锁，则阻塞，直至拥有互斥锁的线程释放它为止。

参数：mutex　　互斥锁

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

动作：对互斥锁解锁。（如果有线程正等待该互斥锁， 就将它唤醒。若有多个线程等待，则按线程的优先级加以选择和调度。

参数：mutex 互斥锁

```
int phread_mutex_trylock(pthread_mutex_t *mutex)
```

动作：对未加锁的互斥锁加锁（并成为互斥锁的拥有者）。如果已加锁，便立即以EBUSY返回。

参数：mutex 互斥锁

2. 条件变量

```
int pthread_cond_init(pthread_con_t *cond, const pthread_condattr_t *attr)
```

动作：以指定的属性创建一个条件变量。

参数：cond 条件变量
attr 属性——NULL时为默认

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

动作：撤销一个条件变量。

参数：cond 条件变量

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

动作：等待一个条件变量，由信号或广播唤醒。在等待前互斥锁已被解锁，等待后将再加锁。（在调用前互斥锁应由线程加锁。）

参数：cond 条件变量
mutex 互斥锁

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
const struct timespec *abstime)
```

动作：在规定时间内等待一个条件变量。与pthread_cond_wait（）类似，不同之处仅在于如果系统时间等于或大于指定时间时，例程将返回已加锁的互斥锁。

参数：cond 条件变量
mutex 互斥锁
abstime 条件未出现时，在返回前的时间。
将时间设置为5秒：

```
abstime.tv.sec = time(NULL) + 5;
abstime.tv.nsec = 0;
```

```
int pthread_cond_signal(pthread_cond_t *cond)
```

动作：解锁正等待一个条件变量的线程。若有多个线程等待， 则按线程的优先级加以选择和调度。如果没有线程等待，将不会为以后的线程保留此信号。

参数：cond 条件变量

```
pthread_cond_broadcast(pthread_cond_t *cond)
```

动作：与pthread_cond_signal()类似，不同之处仅在于将唤醒所有等待条件变量的线程。

参数：cond 条件变量

虽然“唤醒”例程的动作可能只唤醒一个线程，但在某些多处理机系统中可能唤醒多个线程，因此在编码时应考虑到这一点。

参考文献

BUTENHOF, D. R. (1997), *Programming with POSIX® Threads*, Addison-Wesley, Reading, MA.

KLEIMAN, S., D. SHAH, AND B. SMAALDERS (1996), *Programming with Threads*, Prentice Hall, Upper Saddle River, NJ.

NICHOLS, B., D. BUTTLAR, AND J. P. FARRELL (1996), *Pthreads Programming*, O'Reilly & Associates, Sebastopol, CA.

PRASAD, S. (1997), *Multithreading Programming Techniques*, McGraw-Hill, New York.

附录C OpenMP命令、库函数以及环境变量

以下是有关C/C++命令、库函数和环境变量的集合，它们共同构成了OpenMP。更多的细节可在[Chandra et al., 2001]以及《OpenMp C 及C++ 应用程序接口2002年3月的版本2 》的文档中找到（OpenMP体系结构评论委员会，2002)。这两个参考文献之间有一些差别，部分原因是因为OpenMP从版本1到版本2已作了一些修改。本附录中采用《 OpenMP C 及C++ 应用程序接口2002年3月的版本2 》作为权威的文档。

C.1 概要

OpenMP编译器命令以#pragma开始，在其后面是omp，名字和可选的子句，并用新行结束。某些子句可出现在不同的命令中，但对它们需要加以分别的定义。某些命令将作用于整个结构块（语句或语句组)。所谓构造是由编译器命令及跟在其后的结构块所组成。

记号

`[clause] ...`	从后面列出的子句中选择的一个或多个可选的子句。子句可以任何次序排列。在OpenMP版本2 中，它们之间用逗号分开。子句中列出的项以逗号分开。
`structured_block`	单语句或复合语句（只有一个入口和一个出口)。

C.2 PARALLEL REGION (并行区域)

```
#pragma omp parallel [clause] ...
      structured_block
```

动作: 创建多个线程，每个执行指定的结构块`structured_block`。

子句:

```
if (scalar_expression)
private(variable_list)
firstprivate(variable_list)
default(shared or none)
shared(variable_list)
copyin(variable_list)
reduction(operator : variable_list)
num_threads(integer_expression)
```

C.3 WORK-SHARING（工作共享）

工作共享构造不创建线程，它使用现有的线程。通常`parallel`构造出现在工作共享构造之前。在这类构造的结束处没有显式的障栅同步。

```
#pragma omp for [clause] ...
      for_loop
```

动作：将使for循环（for_loop）的各次迭代在组内的现有线程间分配。for循环必须是规范的形式：

```
for(initial_expression;boolean_expression;increment_expres
-sion)
```

其中每个表达式必须是一个简单的形式，如在《OpenMP C 及C++ 应用程序接口2002年3月的版本2》（OpenMP体系结构评论委员会，2002）所描述的那样形如。

```
for (i=0; i<10;i++)
```

那样的循环将被接受。但其中的i在循环中不允许修改。

子句：

```
private(variable_list)
firstprivate(variable_list)
lastprivate(variable_list)
reduction(operator : variable_list)
ordered
schedule(kind, chunk_size)
nowait
```

可将for命令与parallel命令在一行中组合使用，变成parallel for命令和parallel sections命令，除了nowait外，parallel和for中的所有子句均可在parallel for中使用。

```
#pragma omp sections [clause] ...
{
        #pragma omp section
            structured_block
        #pragma omp section
            structured_block
                :
}
```

动作：将使section构造在组内的现有线程间共享。每个section构造执行一次。

子句：仅出现在sections构造中，可以是：

```
private(variable_list)
firstprivate(variable_list)
lastprivate(variable_list)
reduction(operator : variable_list)
nowait
```

可将sections命令与parallel命令在一行中组合使用，变成parallel for命令和parallel sections命令，除了nowait外，parallel和sections中的所有子句均可在parallel sections中使用。

```
#pragma omp single [clause] ...
    structured_block
```

动作：每次仅由组内的一个线程执行结构块。

子句：

```
private(variable_list)
firstprivate(variable_list)
copyprivate(variable_list)
nowait
```

C.4 命令/构造

一般而言，在使用命令前要先建立线程组，因为使用parallel命令意味着需要多于一个的线程，除非另有说明。

```
#pragma omp atomic
    expression_statement
```

动作：语句expression_statement由线程用以下方式执行，使得由本语句所更新的存储单元以原子方式完成，即不受其他线程的干扰。语句必须是下列简单形式之一。

```
x binary_op= expression
x++
++x
x--
--x
```

子句：无子句

```
#pragma omp barrier
```

动作：线程将等待直到所有线程都到达障栅，此后所有线程将一起执行障栅后的语句。

子句：无子句

```
#pragma omp critical name
    structured_block
```

动作：每次由一个线程执行该结构块。线程将等待直至无其他线程执行该结构块，此后将选择等待线程中的一个来执行此结构块。OpenMP中没有定义此选择过程。

子句：可选子句名name用来标识临界区。如果省略，则将被映射到一个无名的临界区

```
#pragma omp flush (variable_list)
```

动作：创建一个同步点，在该点允许所有当前对变量表中的变量进行读和写的操作完成，并将求得的值写回存储器。在线程调用刷新命令时，表中所有变量将被更新。由于使用寄存器在本地保存共享变量的拷贝，故当共享变量被其他进程改变时通常需要执行此命令。应注意的是，该命令不会自动地作用到所有正访问这些变量的线程上。按照[Chandra et al., 2001]每个线程必须分别调用该命令，尽管OpenMP体系结构评论委员会（2002）关于这一点没有明确意见。如果省略变量表，则在线程中可访问的所有变量将被更新。如第8章所述，在各种其他的构造中该命令将自动出现。

子句：无子句

```
#pragma omp master
    structured_block
```

动作：仅由主线程执行结构块。在该构造的开始或结尾无障栅同步。

子句：无子句

```
#pragma omp ordered
    structured_block
```

动作：此命令只能使用在for或parallel　for的构造中。按顺序执行循环的次序执行结构块。一般，在ordered构造外的循环体的其他部分仍将并行执行。

子句：无子句

```
#pragma omp threadprivate(variable_list)
```

动作：表中的变量将变成线程的私有变量。这些变量预先在某个未被指明的地点曾被初始化。threadprivate命令应在任何并行构造前调用，而已被定义的threadprivate 变量，在从一个并行区到下一个并行区时将保持不变（假定线程数固定），即它们的值可在后继的并行区中使用。

子句：无子句

C.5　子句

copyin (variable_list)

用主线程中相应变量的值赋值表中的每个变量。表中的变量必须为threadprivate 变量。该子句可出现在parallel构造中（因此，也可出现在parallel for和parallel sections中）。

copyprivate (variable_list)

在OpenMP版本2中加入该子句，它只能用在single构造中。在组中的每个成员执行完结构块后将变量表中的每个变量值设置成执行该单结构块线程的值。它提供了用共享变量向组中成员广播值的另一种机制。当难以使用共享变量时，可使用该子句。

default (shared or none)

除了那些已说明为threadprivate（或const）的以外，子句default (shared) 指明所有变量为共享变量。如果有任何变量没有显式地声明为共享或私有（或是它的变化firstprivate或lastprivate）或没有出现在归约子句的变量表中，则子句default (none) 将产生一个错误消息。当不使用默认(default) 子句时，其效果如同default (shared) 一样。默认子句可出现在parallel构造中（因此，也可出现在parallel　for和parallel sections中）。

firstprivate (variable_list)

将变量表variable_list中的变量创建为每个线程的私有变量，除了这些变量被初始化为主线程中相应的变量值（在执行该构造之前）以外，其作用同私有子句中声明的一样。该子句可出现在所有命令中。

if (scalar_expression)

如果表达式scalar_expression求得的值为0，则多个结构块就顺序执行，否则就并行执行。这一特征允许在运行时间确定是并行执行还是顺序执行。该子句可出现在parallel构造中（因此，也可出现在parallel　for和parallel sections中）。

nowait

只可使用在for、parallel　for和sections构造中。它将使线程在构造末尾

不等待其他进程，而是允许它们完成计算后向前执行下一语句。

num_threads (integer_expression)

在OpenMP版本2中加入该子句，它只能用在parallel构造中。如果该子句出现，它将优先于其他方法（即omp_set_num_threads () 库函数及环境变量OMP_NUM_THREADS）去说明线程数，并要求由integer_expression给定线程数。

ordered

仅使用在for及parallel for构造中，而且当循环含有排序ordered命令（只允许有一个或没有）时必须使用。除了ordered命令需要使用ordered子句外，使用该子句的主要原因是为了提高编译器的效率。参见在同步命令下的ordered命令的动作部分。

private (variable_list)

将变量表variable_list中的变量创建为每个线程的私有变量。每个线程有这些变量的一个个人拷贝。该子句可出现在所有命令中。

reduction (operator : variable_list)

操作符可以是+、-、*、&、|、^、&&、或 || 中的任何一个。将为每个线程创建变量表中每个变量的一个私有拷贝，并根据操作符对这些变量进行初始化（+、-、|、^、和 || 初始化为0，*、&&初始化为 1，而&则初始化为全1)。在构造执行结束时，主线程保留每个变量的初始值以及使用指定操作符而得到的私有拷贝的最终值。除了single以外，该子句可出现在每个子句中。

schedule (kind, chunk_size)

该子句只可使用在for和parallel for构造中，且由它定义循环中的迭代如何在线程中分配。参数kind可以是:

static

迭代按chunk_size的块大小进行分割，并静态地按循环方式分配给线程。如果省略chunk_size，则迭代近似地被分割成相等块大小，然后给每个线程分配一块。

dynamic

迭代按chunk_size的块大小进行分割，当进程空闲时就分配给一块。如果省略chunk_size，就认为块的大小为1。

guided

当所指定的chunk_size大于1时，分配给线程的块数以指数方式减少，直至块的大小变为chunk_size。如果chunk_size等于1，则每一块的大小近似地由（未被分配的迭代数）/（线程数）给定。如果未指明chunk_size，则便假设为1。

run_time

由环境变量OMP_SCHEDULE确定调度（在run_time中，不使用chunk_size）例如，如果OMP_SCHEDULE被设置成“guided, 8”，以及setenv OMP_SCHEDULE “guided, 8” kind = guided 以及chunk_size = 8。

shared (variable_list)

variable_list中的变量在线程间共享。每个线程可访问这些变量。该子句可出现在parallel构造中（因此，也可出现在parallel for和parallel

sections中)。

C.6 库函数

使用下面的库函数，包括:

```
#include <omp.h>
```

它有两种锁的类型: omp_lock_t和omp_nest_lock。

1. 执行环境函数

```
int omp_get_dynamic(void)
```

动作: 如果动态调整使能，就返回非0值，否则返回0。

参数: 无

```
int omp_get_nested(void)
```

动作: 如果嵌套并行化使能，就返回非0值，否则返回0。

参数: 无

```
int omp_get_num_threads(void)
```

动作: 当在并行区被调用时返回线程组中当前的线程数。

参数: 无

```
int omp_get_max_threads(void)
```

动作: 如果遇到不带 num_threads 子句的并行区就返回组中可用的最大线程数。

参数: 无

```
int omp_get_num_procs(void)
```

动作: 返回程序可用的处理器数。

参数: 无

```
int omp_in_parallel(void)
```

动作: 如果在并行区中被调用时就返回非0值，否则返回0。

参数: 无

```
void omp_set_dynamic(int dynamic_threads)
```

动作: 如果dynamic_threads设置为非0值，则在程序执行过程中可改变线程数以达到最好的系统利用率（动态调整)。如果dynamic_threads设置为0，则该特征失效。在并行区期间实际的线程数是固定的。

参数: dynamic_threads　失效/使能　特征（0/非0)

```
void omp_set_nested(int nested)
```

动作：如果nested设置为非0值，就使能嵌套并行性。如果nested设置为0，则该特征失效并由当前线程顺序化和执行嵌套并行区。

参数：nested 失效/使能 特征（0/非0）

```
void omp_set_num_threads(int num_threads)
```

动作：设置默认线程数供后继并行区使用（除非存在num_threads子句）。

参数：num_threads 线程数

```
int omp_thread_num(void)
```

动作：返回线程的线程数。线程数编号从0到omp_get_num_threads()-1，主线程的线程号为0。

参数：无

2. 锁函数

这些函数用来处理锁变量。它们成对编组，一个用作简单锁，其形式为omp_lock_t，另一个用作嵌套锁，形式为omp_nest_lock。它们有相同的参数，锁变量。

```
void omp_init_lock(omp_lock_t *lock)
void omp_init_nest_lock(omp_nest_lock_t *lock)
```

动作：分配和初始化锁（对简单锁初始化为开锁，对嵌套锁初始化为0）。

```
void omp_destroy_lock(omp_lock_t *lock)
void omp_destroy_nest_lock(omp_nest_lock_t *lock)
```

动作：不初始化锁（退分配并释放锁）。

```
void omp_set_lock(omp_lock_t *lock)
void omp_set_nest_lock(omp_nest_lock_t *lock)
```

动作：阻塞线程直至所指明的锁可用。然后上锁，线程继续执行。对嵌套锁，阻塞线程直至所指明的锁可用，若该锁已为线程拥有则将嵌套计数值加1。

```
int omp_test_lock(omp_lock_t *lock)
int omp_test_nest_lock(omp_nest_lock_t *lock)
```

动作：如可能的话，设置锁。对简单锁，若可置位锁便返回一个非0，否则返回零。对嵌套锁，若可置位锁，该函数便返回新的嵌套计数值，否则返回0

```
void omp_unset_lock(omp_lock_t *lock)
void omp_unset_nest_lock(omp_nest_lock_t *lock)
```

动作：线程释放对锁的拥有。对嵌套锁，则将嵌套计数值减1。

3. 定时

以下的函数是在OpenMP版本2中增加的，其功能类似于MPI中的定时例程。

```
double omp_get_wtick(void)
```

动作：返回连续时钟记号间的秒数。

参数：无

```
double omp_get_wtime(void)
```

动作：返回自过去某个时间开始所花费的时钟时间。

参数：无

C.7 环境变量

在执行之前设置环境变量，典型地使用setenv语句。

OMP_DYNAMIC——使能/禁止动态调整（TRUE/FALSE）

OMP_NESTED——使能/禁止嵌套并行化（TRUE/FALSE）

OMP_NUM_THREADS——用所指明的一个整数设置线程数

OMP_SCHEDULE——用所指明的一个字符串设置运行时调度类型

参考文献

CHANDRA, R, L. DAGUM, D. KOHR, D. MAYDAN, J. MCDONALD, AND R. MENON (2001), *Parallel Programming in OpenMP*, Academic Press, San Diego, CA.

OPENMP ARCHITECTURE REVIEW BOARD (2002), *OpenMP C and C++ Application Program Interface Version 2 March 2002*, from http://www.OpenMP.org.

索　引

注：索引中的页码为英文原书页码，即本书页边所标页码。

符号

O notation（大O表示法）65
Θ notation（Θ表示法）66
Ωnotation（Ω表示法）66

A

Acceleration anomaly（加速异常），410
Acknowledgment message（确认消息），47
  termination detection（终止检测），211
Adaptive quadrature（自适应积分），125
Address（地址），14
Adjacency list（邻接表），216
Adjacency matrix（邻接矩阵），216
Allgather（全集中），178
All-to-all routine（全部到全部例程），121，323
Amdahl's law（阿姆达尔（Amdahl）定律），8
Asynchronous iterative method（异步迭代方法），192
Atomic instruction（原子命令），242

B

Backtracking（回溯），409
Bandwidth（带宽），16
Barnes-Hut algorithm（算法），128
Barrier（障栅），163
  butterfly（蝶形），167
  counter implementation（计数器实现），165
  linear（线性），165
  memory（存储器），264
  MPI（MPI），164
  OpenMP（OpenMP），257
  shared memory（共享存储器），246
  tree implementation（树实现），167
Beowulf cluster（机群），33
Bernstein's conditions（条件），250
Best-first search（最佳优先搜索），408
Binary exchange algorithm（二元交换算法），397，398
Binary tree network（二叉树网络），20
Bin-packing（装箱问题），202
Bisection bandwidth（对分带宽），17
Bisection width（对分宽度），17
Bitmap（位图），81
Bitonic mergesort（双调谐归并排序），317
Bitonic sequence（双调谐序列），317
Block allocation（块分配），179
Block partition（块划分），187
Blocking routine（阻塞例程），47，48，56
Bounding function（限界函数），408
Branch-and-bound search（分支限界搜索），407，409
Breadth-first search（广度优先搜索），408
Broadcast（广播），49
  communication time（通信时间），69
Broadcast routine（广播例程），59
Bubble sort（冒泡排序），307
Bucket sort（桶排序），117，333
Buffered mode, MPI（缓冲方式），MPI，58
Busy waiting（忙等待），241
Butterfly（蝶形），167，398

C

Cache coherence protocol（高速缓存一致性协议），258
Cache memory（高速缓冲存储器），15，258
Cannon's algorithm（Cannon算法），348
Cellular automata（细胞自动机），190
Chaotic relaxation（无序松弛），192
Circuit switching（电路交换），22
Closed list（封闭表），409
Cluster computing（机群计算），26～38
Cluster of workstations（工作站机群），27
Communication latency（通信时延），16
Communication time（通信时间），13，62
Communicator（通信子），53，55
Compare-and-exchange operation（比较和交换操作），304
Computation time（计算时间），13
Computation/communication ratio（计算/通信比），13，63，67
Computational time（计算时间），62
Condition variable（条件变量），244
  Pthread（*Pthread*），245
Contrast stretching（对比度扩展），372

Convolution（卷积），367，390
Cosmic cube（Cosmic立方体），19
Cost（成本/代价），68
  interconnection network（互联网络），16
Cost-optimal（代价－优化），68
Counting sort（计数排序），328
COW. See Cluster of workstations（COW，见工作站机群）
Critical section（临界区），239，240
  OpenMP（OpenMP），257
Cross-correlation（互相关），378
Crossover（交叉），411，416
Cut-off function（修剪函数），408
Cut-through（直通），22
Cyclic allocation（循环分配），179

D

Data parallel computation（数据并行计算），170
Data partitioning（数据划分），106，146，148，179，186，306
Dead node（死结点），409
Deadlock（死锁），23，169，189，242
Deadly embrace（死亡拥抱），242
Debugging（调试），70～72
Deceleration anomaly（减速异常），410
Dense matrix（稠密矩阵），352
Dependency analysis（相关性分析），250
Depth-first search（深度优先搜索），408
  parallel（并行），409
Detached thread（分离线程），237
Detrimental anomaly（不利异常），411
DFT,（参见离散傅里叶变换），388
Diagonally dominant array（对角占优矩阵），175
Diameter（直径），17
  hypercube（超立方体），18
  mesh（网格），17
Diminished prefix sum（减少前缀求和），332
Discrete Fourier transform（离散傅里叶变换），388
Distributed shared memory（分布式共享存储器），24，279～281
  hardware DSM system（硬件DSM系统），282～283
  software DSM system（软件DSM系统），281～282
Divide and conquer（分治），111
  M-ary（*M*路），116
Domain decomposition（域分解），106
DSM，参见分布式共享存储器
Duplicated computations, for reduced message passing（重复的计算，为减少消息传递），306
Dynamic process creation（动态进程创建），43，80
Dynamic task assignment（动态任务分配），90
Dynamic tree（动态树），408

E

E-cube routing algorithm（E立方体路由算法），19
Edge detection（边缘检测），371，379
Edge detection masks（边缘检测掩码），380
Edge, graph（边，图），214
Efficiency（效率），7
Eight-puzzle（8拼块），407
Elapsed time（运行时间），73
Embarrassingly parallel computation（易并行计算），79
E-node（*E*结点），409
Enumeration sort（枚举排序），327
Ethernet（以太网），28
Event synchronization（事件同步），260
  See also condition variable（也可参见 条件变量）
Execution time（执行时间），
  measuring（测量），72
  parallel（并行），62

F

False sharing, in caches（假共享，在高速缓存中），259
Fast Fourier transform（FFT）快速傅里叶变换（FFT），395
Fat tree network（粗树网络），20
Fifteen-puzzle（15拼块），407
Finite difference method（有限差分方法），182，357
Fitness, in genetic algorithms（适应度，在遗传算法中），412
Flit片（flow control digit，流控位），22
Folded network（折叠的网络），17
`forall` statement（`forall`语句），170，249
FORK（分叉），232
Fourier series（傅里叶级数），387
Fourier transform（傅里叶变换），387，388
  discrete（离散），388
  fast（快速），395
  for image processing（用于图像处理），389
  inverse（逆向），388
  scale factor（比例因子），389
  transpose（转置），390
Frequency domain（频域），371
Frequency filter（频率滤波器），141
Functional decomposition（功能分解），106，140

G

Game of Life（生命游戏），190

Gather（集中），50，59
  communication time（通信时间），69
Gaussian elimination（高斯消去），353
Gauss-Seidel（高斯-塞德尔）
  iteration（迭代），183
  relaxation（松弛），360
  successive overrelaxation（连续过度松弛），363
Genetic algorithm（遗传算法），411～423
  constraints（约束），415
  crossover（交叉），411，416
  distributed processor system（分布式处理器系统），423
  fitness（适应度），412
  genes（基因），417
  individuals（个体），412
  island model（孤岛模型），420
  migration（迁移），419
  mutation（突变），411，412，417
  selective pressure（选择压力），415
  shared memory（共享存储器），422
  stepping stone model（跳石模型），420
  termination（终止），418
  tournament selection（竞赛选择），416
Geometrical transformations（几何变换），81
Gflop（10亿次浮点运算（Gflop）），4
Ghost points（幻象点），188
Gradient direction（梯度方向），379，385
Gradient magnitude（梯度幅值），379
Grand challenge problem（巨大挑战性问题），3
Graph（图），214
  directed（有向），214
  representation（表示），215
  undirected（无向），215
Gray level（灰度级），370
  reduction（减少），373
Gray scale（灰度），370
Grayscale（灰度），81
Grid computing（网格计算），36
Gustafson's law（Gustafson定律），12

H

Heap（堆），410
Heavyweight process（重量级进程），233
Hill-climbing（爬山），424
  banking application（金融业务应用），427
Histogram（直方图），373
Hough transform（霍夫变换），371，383～387
Hypercube network（超立方体网络），18
Hyperquicksort（超快速排序），326

I

Image processing（图像处理），81
  contrast stretching（对比度扩展），372
  edge detection（边缘检测），371
  global operations（全局操作），372
  gray level reduction（灰度级减少），373
  histogram（直方图），373
  Laplace operator（拉普拉斯算子），382
  local operations（局部操作），372
  low-level（低层），370，371
  noise cleaning（噪声清除），371
  noise reduction（噪声减少），371，374
  point processing（点处理），372
  sharpening（锐化），374
  smoothing（平滑），374
  thresholding（阈值化），372
Insertion sort（插入排序），148
Instruction-level parallelism（指令级并行），249
Intercommunicator（外通信子），56
Interconnection network（互联网络），14
Internet protocol（网际协议）
  version 4 (IPv4)（版本)4（IPv4），29
  version 6 (IPv6)（版本)6（IPv6），31
Intracommunicator（内通信子），56
Invalidate policy（使无效策略），在DSM中，283
Inverse Fourier transform（逆傅里叶变换），388
IP address（IP地址），29
Island model（孤岛模型），420
Iteration（迭代），
  termination（终止），175

J

Jacobi（雅可比），
  overrelaxation（过度松弛），363
Jacobi iteration（雅可比迭代），175，357
  convergence speed（收敛速度），359
JOIN（JOIN（接合）），232

K

Knapsack problem（背包问题），406

L

LAM，（LAM），52

Laplace operator（拉普拉斯算子），382
Laplace's equation（拉普拉斯方程），182，357，358
Latency（时延），16，63
  pipeline（流水线），142
Latency hiding（时延隐藏），64
Lazy release consistency（滞后松弛一致性），285
Left-to-right routing（从左到右的路由），19
Linear congruential generator（线性同余生成器），97
Linear equations（线性方程），
  matrix relationship（矩阵关系），342
  solving（求解），352 ~ 356
  solving by iteration（用迭代法求解），174 ~ 180
  solving system with pipeline（流水线求解系统），154 ~ 157
  upper-triangular system（上三角系统），154
Live node（活结点），409
Livelock（活锁），23
Load balancing（负载平衡），90，201
  centralized（集中式），203
  centralized dynamic（集中式动态），204
  decentralized（分散式），203
  decentralized dynamic（分散式动态），205
  dynamic（动态），203
  line structure（线形结构），207
  pipeline（流水线），207
  static（静态），202
  termination（终止），204
Lock（锁），240
  mutex（互斥），268
  Pthread（Pthread），242
  read/write（读/写），275

M

MAC，参见媒体访问控制器
Mandelbrot set（曼德勃罗特（Mandelbrot）集），86
Mapping problem（映射问题），202
M-ary tree（*M*叉树），20
Master/slave（主/从），80，107，146
Master-slave（主-从），
  DFT implementation（DFT实现），392
Matrix（矩阵），340
  addition（加），340
  dense（稠密），352
  multiplication（乘），341
  sparse（稀疏），352
Matrix multiplication（矩阵乘法），
  block（块），343
  Cannon's algorithm（Cannon算法），348
  mesh implementations（网格实现），348
  recursive implementation（递归实现），346
  sequential（顺序），342
  Strassen's method（Strassen方法），367
  systolic array（脉动阵列），350
two-dimensional pipeline（二维流水线），350
Matrix transpose（矩阵转置），121
Matrix-vector multiplication（矩阵向量乘法），341，351
  for Discrete Fourier transform（用于离散傅里叶变换），393
Mean, for image smoothing（平均值，用于图像平滑），374
Media access controller（媒体访问控制器），31
Median, for image smoothing（中值，用于图像平滑），375
Memory barrier（存储器障栅），264
Memory consistency（存储器一致性），284
  relaxed（松弛的），285
Memory fence（存储器篱芭），264
Merge, sorted lists（归并，已排序序列），316
Mergesort（归并排序），311
  potential speedup（潜在加速比），304
Mesh（网格），17
Message buffer（消息缓冲），48
Message latency（消息时延），17，63
Message passing（消息传递），
  blocking（阻塞），47，48
  completion（完成），56
  globally complete（全局完成），56
  locally blocking（本地阻塞），48
  locally complete（局部完成），56
  nonblocking（非阻塞），47，48
  synchronous（同步），46
Message tag（消息标记），48，56
Message-Passing Interface（消息传递接口），27，52 ~ 60
Message-passing multicomputer（消息传递多计算机），16
Metacomputing（元计算），36
MIMD，（见多指令流多数据流），
MISD，（见多指令流单数据流），
Monitor（监控程序），244
  Java（Java），270
Monte Carlo method（Monte Carlo（蒙特卡罗）法），93，94
  hill climbing（爬山），424
Moore's algorithm（Moore摩尔算法），217
MPI，（见消息传递接口）
MPICH（MPICH），52

MPMD，（见多程序多数据）
Multicast（多播），49
Multicomputer（多计算机），16
Multigrid method（多网格法），364
Multiple instruction stream-multiple data stream（多指令流多数据流），25
Multiple instruction stream-single data stream（多指令流单数据流），26
Multiple program-multiple data（多程序多数据），26，44
Multiple reader/multiple writer, in DSM（多阅读器/多写入器，在DSM中），283
Multiple reader/single writer, in DSM（多阅读器/单写入器，在DSM中），283
Mutation（突变），411，412，417
Mutual exclusion（互斥），240
  synchronization（同步），260
Mutually exclusive lock variable(mutex)（互斥锁变量(mutex)，242，268

N

Natural order（自然顺序），181，358
N-body problem（*N*体问题），5，126～131
Network interface card（网络接口卡），31
Network latency（网络时延），16
Network of workstations（工作站网络），27
NIC，见网络接口卡
Non-blocking receive（非阻塞接收），209
Nonblocking routine（非阻塞例程），47，48，57
Non-uniform access（非均匀存取），231
Non-uniform memory access（非均匀存储器存取），15
Normal representation（标准表达式），385
NOW，见工作站网络
n-queens problem（*N*皇后问题），406
NUMA，见非均匀存储器存取
Numerical integration（数值积分），122～125
  adaptive quadrature（自适应积分），125
  Monte Carlo method（蒙特卡罗法），94
  static assignment（静态分配），123
  trapezoidal method（梯形法），123

O

Occam，(Occam语言），42
Octtree（八叉树），116，128
Odd-even mergesort（奇偶归并排序），316
Odd-even transposition sort（奇偶互换排序），310
Omega network（Omega网络），21
Open list（开放表），409
OpenMP（OpenMP），15，232，253～257
Orthogonal recursive bisection（正交递归二分法），130
Oscar(Open Source Cluster Application Resources)（开放源机群应用资源），37
Overlapping connectivity network（重叠互通网络），34
Overrelaxation（过度松弛），363
Overrelaxation parameter（过度松弛参数），363

P

P operation（P操作），242
Packet switching（包交换），22
`par` construct（`par`构造），249
Parallel computer（并行计算机），5，13
Parallel execution time（并行执行时间），62
Parallel programming（并行程序设计），5
Parallel slackness（并行不完善性），65
Parallel time complexity（并行时间复杂性），67
  sorting algorithms（排序算法），304
Parallel virtual machine（并行虚拟机），27，51
Parallelizing compiler（并行化编译器），42，232，250
Parameter space（参数空间），384
Partial pivoting（部分选主元），353
Partitioning（划分），106
  cyclic-striped（循环条状的），356
  strip（条状的），355
Ping-pong method（乒乓方法），73
Pipeline（流水线），
  adding numbers（数字相加），145
  for discrete Fourier transform（离散傅里叶变换），393
  frequency filter（频率过滤器），141
  latency（时延），142
  load balancing（负载平衡），207
  for matrix multiplication（矩阵乘法），350
  prime number generation（生成质数），152
  solving system of linear equations（线性方程组求解），155
  sorting numbers（数的排序），148
Pipelining（流水化），140
Pivot（枢轴），
  quicksort（快速排序），313，325
Pixel（像素），370
Pixmap（像素图），81，371
Point processing（点处理），372
Poisson's equation（泊松方程），357
Prefix sum problem（前缀求和问题），172
Prewitt operator（Prewitt算子），381
Prime number generation（生成质数），152～154

Priority queue（优先队列），409
using a heap（使用堆），410
Problem size（问题规模），11
Process（进程），16，43，203
heavyweight（重量级），233
root（根），50
Process creation（进程创建），
dynamic（动态），43，80
static（静态），43，80
Processor consistency（处理器一致性），264
Processor farm（处理器农庄），90，204
Processor-time product（处理器-时间乘积），68
Process-time diagram（进程-时间图），71
Profile, program（特性形象，程序），74
Program order（程序次序），263
Pseudocode constructs（伪代码构造），60
Pthread（线程），235
PVM（PVM），见并行虚拟机

Q

Quadtree（四叉树），116，128
Quicksort（快速排序），313，333
hypercube（超立方体），323
pivot（枢轴），313，325
potential speedup（潜在加速比），304
workpool（工作池），315

R

Radix sort（基数排序），331
Random number generator（随机数生成器），
linear congruential generator（线性同余生成器），97
parallel（并行），97
SPRNG（SPRNG），98，99
Random polling algorithm（随机轮询算法），207
Randomized algorithm（随机算法），202
Rank sort（秩排序），327
Read/write lock（读/写锁），275
Ready mode, MPI（就绪方式），MPI，58
Receive routine（接收例程），46
MPI（MPI），56
non-blocking（非阻塞），209，
synchronous（同步），46
Receiver-initiated task transfer（接收器启动的任务传输），206
Recursive bisection（递归对分），202
Red-black checkerboard（红黑棋盘），362
Red-black ordering（红黑排序），361
Reduce（归约），51，109
Release consistency（释放一致性），285
Rendezvous（会合），47
Replicated worker（复制工作者），204
Ring termination algorithm（环终止算法），212
Root process（根进程），50
Round robin algorithm（循环算法），202，207
Routing algorithm（路由算法），
hypercube（超立方体），19
Row major order（行主序），184

S

Safe communication（安全通信），55
Safe message passing（安全消息传递），189
Sample sort（采样排序），333
Scalability（可扩展性），11
Scalable Pseudorandom Number Generator(SPRNG)（可扩展伪随机数生成器）(SPRNG)，98，99
Scatter（分散），50，59，109
Scheduling（调度），202
Semaphore（信号量），242
binary（二元的），243，265
Send routine（发送例程），46
blocking（阻塞），56
MPI（MPI），56，57，58
nonblocking（非阻塞），57
synchronous（同步），46
Sender-initiated task transfer（发送器启动任务传输），206
Sequential consistency（顺序一致性），262
Shared memory（共享存储器），14
Shared virtual memory（共享虚拟存储器），24
Shared memory multiprocessor（共享存储器多处理机），14，230
programming（程序设计），14
Shearsort（扭曲排序），321
image processing（图像处理），376
Shortest path problem（最短路径问题），214 ~ 223
Sieve of Eratosthenes（埃拉托色尼（Eratosthenes）筛选），152
SIMD（SIMD，见单指令流多数据流）
Single address space（单一编址空间），231
Single instruction stream-multiple data stream（单指令流多数据流），25
Single instruction stream-single data stream（单指令流单数据流），25
Single instruction-multiple data（单指令多数据），170
Single program -multiple data（单程序多数据），26，44，54，80，123，171，234

Single reader/single writer, in DSM（单阅读器/单写入器），DSM，283，292
SISD（SISD，见单指令流单数据流）
SMP（SMP，见对称多处理机）
Sobel operator（Sobel算子），381
Sorting（排序），303
  pipeline（流水线），148
Sorting algorithms（排序算法）
  bitonic mergesort（双调谐归并排序），317
  bubble sort（冒泡排序），307
  bucket sort（桶排序），117，333
  cluster（机群），333
  counting sort（计数排序），328
  enumeration sort（枚举排序），327
  hyperquicksort（超快速排序），326
  insertion sort（插入排序），148
  mergesort（归并排序），304，311
  odd-even mergesort（奇偶归并排序），316
Sorting algorithms
  odd-even transposition（奇偶互换排序），310
  parallel time complexity（并行时间复杂性），304
  quicksort（快速排序），304，313，333
  radix sort（基数排序），331
  rank sort（秩排序），327
  sample sort（采样排序），333
  sequential time complexity（顺序时间复杂性），304
  shearsort（扭曲排序），321，376
  stable（稳定的），328
  two-dimensional（二维），321
Space complexity（空间复杂性），65
Space-time diagram（时空图），71，142
Sparse matrix（稀疏矩阵），352
Spatial domain（空间区域），371
Spawn（派生），45
Speedup factor（加速系数），6，63
  scaled（比例），12
Spin lock（自旋锁），241
SPMD（SPMD，见单程序多数据）
Stable sorting algorithm（稳定的排序算法），328
Standard mode，MPI（标准模式，MPI），58
Startup time（启动时间），17，63
State space tree（状态空间树），407
Static process creation（静态进程创建），43，80
Static task assignment（静态任务分配），89
Static tree（静态树），408
Stencil（模板），
  13-point（13点），363
  5-point（5点），363
  9-point（9点），363
Stepping stone model（跳石模型），420
Store-and-forward packet switching（存储转发包交换），22
Strassen's method（Strassen方法），367
Strict consistency（严格一致性），284
Strip partition（条状划分），187
Submatrix（子矩阵），343，345
Successive refinement（连续求精），423
Superlinear speedup（超线性加速比），7
  Symmetrical multiprocessor（对称多处理机），34
cluster（机群），281，295，334
Synchronization（同步）
  event（事件），260
  local（本地），169，180
  mutual exclusion（互斥），260
  OpenMP（OpenMP），256
Synchronous computation（同步计算），170～174
  fully（完全），163
Synchronous computations（同步计算）
  partial（部分），191～192
Synchronous iteration（同步迭代），173
Synchronous message passing（同步消息传递），46
Synchronous mode,MPI（同步方式），MPI，58
Synchronous parallelism（同步并行性），173
Synchronous receive routine（同步接收例程），46，48
Synchronous send routine（同步发送例程），46，48
Systolic array（脉动阵列），350

T

TCP/IP. *See* Transmission Control Protocol/Internet Protocol（TCP/IP，见传输控制协议/网际协议）
Template matching（模板匹配），371
Termination（终止）
  counter method（计数器方法），92
Termination conditions（终止条件）
  distributed detection（分布式检测），210
  genetic algorithm（遗传算法），418
Termination detection（终止检测），201,203
  fixed energy algorithm（固定能量算法），214
  ring termination algorithm（环终止算法），212
  tree algorithm（树算法），213
Tflop（万亿次浮点运算），4
Thread（线程），234
  detached（分离的），237
Thread-safe routine（线程安全例程），239
Thresholding（阈值化），372
Tiff image file（Tiff图像文件），81

Time complexity（时间复杂性），65～68
Torus（环绕网），17
Transmission Control Protocol/Internet Protocol（传输控制协议/网际协议），29
Transpose（转置），
for Fast Fourier transform（快速傅里叶变换），397，399
Transpose, use in Fourier transform（变换，用于傅里叶变换），390
Transposition array（转置，数组），322
Transputer，42，208
Traveling salesperson problem（旅行商问题），406
Tree network（树状网），20
Twiddle factor（旋动系数），391，396

U

UMA.（见一致存储器访问）
Unified Parallel C（统一并行C），15，248
Uniform memory access（均匀存储器访问），15，231
Universal fat tree（通用粗树），20
Unsafe message passing（不安全消息传递），189
UPC.（见统一并行C）
Update policy, in DSM（更新策略，在DSM中），283
Upper-triangular system of linear equations（上三角线性方程组），154
Upshot（Upshot），71
Utilization-time diagram（时间利用图），71

V

V operation（V 操作），242
Vector（向量），341
Vertex, graph（顶点，图），214
Virtual channel（虚拟通道），23
Virtual processor（虚拟处理器），65
Virtual shared memory（虚拟共享存储器），281
Visualization tools（可视化工具），71

W

Weak consistency（弱一致性），285
Weather forecasting（气象预报），4
Weighted masks（加权掩码），377
Wild card（通配符），48，56
Work pool（工作池），90，204
quicksort（快速排序），315
shortest path algorithm（最短路径算法），220
Wormhole routing（虫孔路由），22