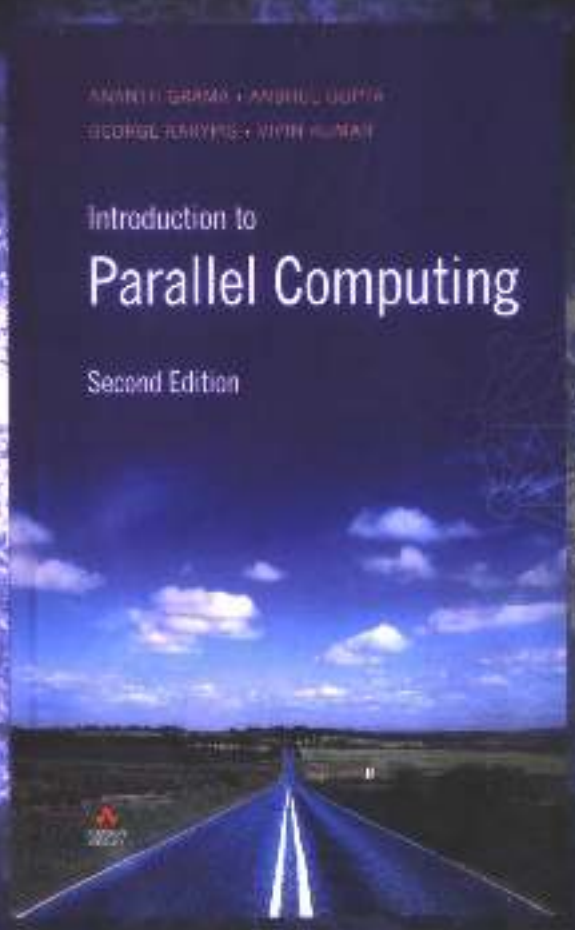
计 算 机 科 学 丛 书

原书第2版

并行计算导论

(美) Ananth Grama Anshul Gupta George Karypis Vipin Kumar 著
张武 毛国勇 程海英 等译 李伯民 审校



Introduction to Parallel Computing

Second Edition

机械工业出版社
China Machine Press

本书系统介绍涉及并行计算的体系结构、编程范例、算法与应用和标准等，覆盖了并行计算领域的传统问题，并且尽可能地采用与底层平台无关的体系结构和针对抽象模型来设计算法。书中选择MPI (Message Passing Interface)、POSIX线程和OpenMP这三个应用最广泛的编写可移植并行程序的标准作为编程模型，并在不同例子中反映了并行计算的不断变化的应用组合。本书结构合理，可读性强，加之每章精心设计的习题集，更加适合教学。

本书原版自1993年出版第1版到2003年出版第2版以来，已在世界范围内被广泛地采用为高等院校本科生和研究生的教材或参考书。

作者简介

Ananth Grama 普度大学计算机科学系的副教授，研究领域是并行和分布式系统和应用的不同方面。

Anshul Gupta IBM T.J. Watson Research Center的研究人员，研究领域是并行算法和科学计算。

George Karypis 明尼苏达大学计算机科学和工程系的副教授，研究领域是并行算法设计、数据挖掘和生物信息学等。

Vipin Kumar 明尼苏达大学计算机科学与工程系的教授，美国军用高性能计算研究中心的主任，研究领域是高性能计算、用于科学计算问题和数据挖掘的并行算法。

ISBN 7-111-14985-8

9 787111 149859

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 7-111-14985-8/TP・3551

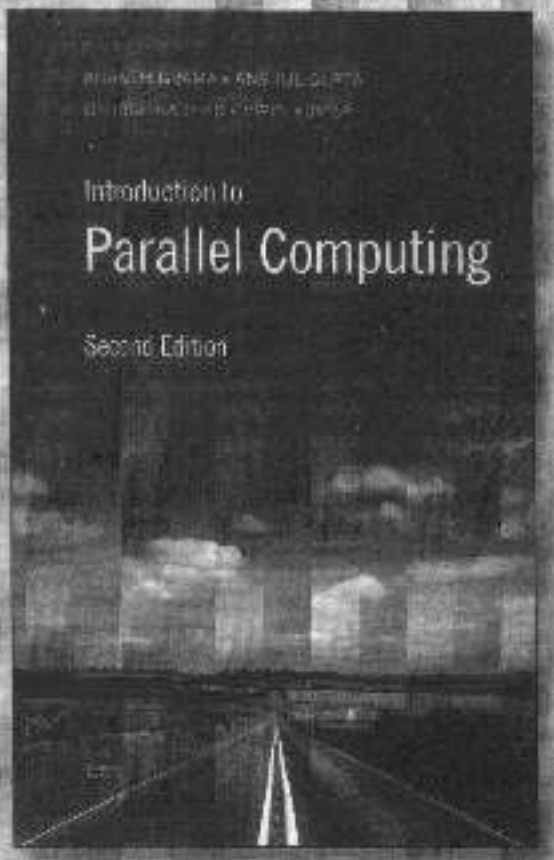
定价：49.00元

计 算 机 科 学 丛 书

原书第2版

并行计算导论

(美) Ananth Grama Anshul Gupta George Karypis Vipin Kumar 著
张武 毛国勇 程海英 等译
李伯民 审校



Introduction to Parallel Computing

Second Edition

机械工业出版社
China Machine Press

本书全面介绍并行计算的各个方面，包括体系结构、编程范型、算法和标准等，涉及并行计算中的新技术，也覆盖了较传统的算法，如排序、搜索、图和动态编程等。本书尽可能采用与底层平台无关的体系结构并且针对抽象模型来设计算法。书中选择MPI、POSIX线程和OpenMP作为编程模型，并在不同例子中反映了并行计算的不断变化的应用组合。

本书论述清晰，示例生动，并附有大量习题。适合作为高等院校计算机及相关专业本科生和研究生的教材或参考书。

Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar: *Introduction to Parallel Computing (Second Edition)* (ISBN:0-201-64865-2).

Copyright © 2003 by Pearson Education Limited.

This translation of *Introduction to Parallel Computing (Second Edition)* (ISBN:0-201-64865-2) is published by arrangement with Pearson Education Limited.

本书中文简体字版由英国Pearson Education培生教育出版集团授权出版。

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：01-2003-5007

图书在版编目（CIP）数据

并行计算导论（原书第2版）/（美）格兰马（Grama, M.）等著；张武等译. -北京：机械工业出版社，2005.1

（计算机科学丛书）

书名原文：Introduction to Parallel Computing (Second Edition)

ISBN 7-111-14985-8

Ⅰ.并… Ⅱ.①格… ②张… Ⅲ.并行算法 Ⅳ.TP301.6

中国版本图书馆CIP数据核字（2004）第076848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：朱起飞

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1月第1版第1次印刷

787mm × 1092mm 1/16 · 28印张

印数：0 001-4000册

定价：49.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

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

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、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		

秘　书　组

武卫东　　温莉芳　　刘　江　　杨海玲

中文版序

我很高兴《并行计算导论》一书终于有了它的中文译本。在此，恭贺并感谢上海大学计算机学院张武教授承担了这项巨大的工程，以将此书呈献给了广大的中国读者。

并行计算是一个充满活力的领域，经过几十年的发展，这一领域的研究成果已在科学与技术的众多领域随处可见。现在，超级并行计算机在大规模科学与工程计算、电子商务、网络搜索、数据挖掘等领域都得到了广泛的应用。同时，利用现有的软硬件和网络技术制造机群式超级计算机也已变得越来越方便。此外，所有的超级计算机供应商都可为客户提供专用的超级并行计算机。重要的问题是如何充分利用这些超级并行计算机，这需要发展有效的并行算法和以适当的程序范例实施这些算法，这就是本书所讨论的重点。

我衷心希望本书能为读者在学习和应用并行计算这一重要的和不断发展着的最新技术的过程中提供有益的帮助。

Vipin Kumar
美国明尼苏达大学计算机科学与工程系教授
美国陆军高性能计算研究中心主任
Email: kumar@cs.umn.edu
URL: http://www.cs.umn.edu/~kumar

译 者 序

当代科学、技术和社会经济的发展对大规模科学与工程计算的需求是无止境的，诸如核反应模拟、数值天气预报、基因工程、城市交通、电子商务和网络搜索等问题都对计算提出了巨大的挑战。随着超级并行计算机的飞速发展，尤其是机群式超级计算机提供越来越方便的并行计算资源，如何充分利用这些并行计算机资源，已对并行计算及其应用构成严峻的考验。

《并行计算导论》的作者Vipin Kumar 教授等是国际知名的并行计算专家，他们在并行算法设计与分析、并行计算应用等方面有很深的造诣。自本书第1版1994年出版到2003年出版第2版以来，已在世界范围内被广泛地采用为高等院校本科生和研究生的并行计算教材或参考书。本书系统介绍涉及并行计算的体系结构、编程范例、算法与应用和标准等。本书结构合理，可读性强，加之每章精心设计的习题集，形成了本书的主要特色。相信本书对培养国内急需的并行计算人才和推动并行计算课程建设与改革必将发挥积极的作用。

本书的前言、第1、8、9、13章由张武翻译，第2、10、11、12章由毛国勇翻译，第6、7章由程海英翻译，第3章由宋安平翻译，第4章由付朝江翻译，第5章由张衡翻译。全书由张武统稿。

本书主要作者Vipin Kumar 教授应邀为本书中文版写了序，张武教授在美国伊利诺理工学院（IIT）计算机系工作期间，曾与Kumar 教授有过交流与合作。美国IIT计算机系孙贤和教授、上海大学童维勤教授、支小莉博士和夏骄雄讲师等在本书的翻译过程中给予了帮助。

本书翻译期间正值全国抗击SARS，上海大学于2003年10月接受国家教育部“本科教学水平”评估。因此，翻译时间非常仓促。虽然在翻译过程中力求尊重原意和翻译准确，但由于水平有限及新术语的不断出现，不当和疏漏之处在所难免，敬请读者提出宝贵意见。

译 者

上海大学计算机学院

2004年1月

前　言

自1994年本书第1版问世以来，并行计算领域发生了巨大的变化。十多年前，紧耦合可扩展的信息传递平台是并行计算的主流模式，而今，主导地位则由大量成本较低的工作站、多处理机工作站以及服务器构成的机群式计算平台所占据。在此期间，这些平台的程序设计模型也得以发展。尽管十多年前绝大部分的机器还依赖于特定的应用程序编程接口（API）来实现信息交换与基于循环的并行，而现在的模型已在不同平台上将API标准化。一些程序设计模型已被接受为标准，比如消息传递库PVM和MPI、POSIX线程库、基于指令的OpenMP等，并被移植到多种平台上。

十年前，流体力学、结构力学和信号处理等是并行计算应用的主要领域，并且这些应用仍对当前并行计算平台构成挑战。但是，今天许多新的应用也正变得越来越重要，其中包括事务处理、信息检索、数据挖掘与分析和多媒体服务等数据密集型应用。新兴应用领域，如计算生物学与纳米技术，对并行算法和系统开发也提出了巨大的挑战。在并行体系结构、程序设计模型与应用上的变化，也伴随着如何使用户以基于网格服务的形式利用并行计算平台方面的改变。

以上发展对并行算法的设计、分析与实现过程产生了深刻的影响。十年前，并行算法设计主要强调的是如何将任务精确地映射到如格网和超立方体这样的特定拓扑结构上，而今天却是从算法设计与实现的角度来强调可编程性与可移植性。为此，本书尽可能采用与底层平台无关的体系结构，并针对抽象模型来设计算法。关于程序设计模型，本书选择消息传递接口（Message-Passing Interface, MPI）、POISX线程和OpenMP。对于涉及并行计算的应用组合，也反映在本书的各种例子中。

本书构成一门专注于并行计算的课程的基础，也可分为两部分讲授。以下是对按两部分讲授的建议：

1）并行计算导论：第1～6章。这个课程提供并行算法设计和并行编程的基础。

2）并行算法设计与分析：第2、3章以及第8～12章。这个课程对各种并行算法的设计与分析进行深入的探讨。

本书材料曾在明尼苏达大学和普度大学的并行算法与并行计算课程中试用过。这些课程是为计算机科学专业低年级研究生与高年级本科生开设的。另外，对学习计算密集型问题的科学与工程专业的研究生，在他们选修的与科学计算相关的课程中也试用过这些材料。

本书多数章包括：1）例子与图解；2）教材的补充习题，用于测试学生对所学内容的理解程度；3）书目评注，用于帮助有兴趣的学生和研究人员了解相关的和更高级的课题。本书的索引帮助读者快速查找到感兴趣的术语。术语所在的页号使用黑体排印。内容相对比较复杂的节上加了“*”号。此外，从出版商（http://www.booksites.net/kumar）那里可以获得更多支持信息。

与本书前一版一样，我们认为本书也在不断发展之中。要感谢那些对我们第1版书提出批

注：原英文版的“参考文献”部分的电子文件现放在http://www.hzbook.com上。有需要的读者请到网站上下载。

评、建议、问题和提供代码或者其他信息的读者，并真诚地希望围绕着这本新版书，我们之间还能继续这种交流。我们鼓励读者通过电子邮件 book-vk@cs.umn.edu 与我们交流。所有相关的读者反馈将在征得同意后加到http://www.cs.umn.edu/~parbook下的归档信息中。该网站还维护着本书的在线勘误表。我们相信，在像并行计算这个高度动态变化的领域，以这种健全的方式进行思想和资料的交流将使我们受益良多。

致 谢

首先要感谢我们各自的妻子Joanna、Rinku、Krista和Renu，如果没有她们的包容与理解，这项工作是不可能如期完成的。我们也要感谢我们各自的父母及其他家庭成员，Akash、Avi、Chethan、Eleni、Larry、Mary-Jo、Naina、Petros、Samir、Subhasish、Varun、Vibhav和Vipasha，感谢他们在本书写作期间给予作者的热情支持与鼓励。

我们各自所属的机构，普度大学计算机科学与计算研究所（CRI）、明尼苏达大学计算机科学与工程系、美国陆军高性能计算研究中心（AHPCRC）和数字技术中心（DTC），以及位于Yorktown Heights的IBM T. J. Watson研究中心，所具备的计算资源和活跃的学术氛围为我们提供了极大帮助。

本项目源于我们的第1版书，因此要感谢所有在两个版本书的写作期间帮助过我们的人们。许多人以不同方式为这个项目做出了贡献。我们要感谢Ahmed Sameh一如既往的支持与鼓励，Dan Challou、Michael Heath、Dinesh Mehta、Tom Nurkkala、Paul Saylor和Shang-Hua Teng为本书的各个版本提供了有价值的输入工作。感谢明尼苏达大学与普度大学中选修了并行计算导论课程的学生们，他们找出了本书早期手稿中的各种错误。特别要提到的是Jim Diehl和Rasit Eskicioglu，他们极其耐心地为本书的手稿勘正了大量的错误。Ramesh Agarwal、David Bailey、Rupak Biswas、Jim Bottum、Thomas Downar、Rudolf Eigenmann、Sonia Fahmy、Greg Frederickson、John Gunnels、Fred Gustavson、Susanne Hambrusch、Bruce Hendrickson、Christoph Hoffmann、Kai Hwang、Ioannis Ioannidis、Chandrika Kamath、David Keyes、Mehmet Koyuturk、Piyush Mehrotra、Zhiyuan Li、Jens Palsberg、Voicu Popescu、Alex Pothen、Viktor Prasanna、Sanjay Ranka、Naren Ramakrishnan、Elisha Sacks、Vineet Singh、Sartaj Sahni、Vivek Sarin、Wojciech Szpankowski、Srikanth Thirumalai、JanVitek和David Yau为本书提供了宝贵的技术资料。与有合作精神和乐于助人的Pearson Education出版社的员工们一起工作是件非常愉快的事。特别地，我们要感谢Keith Mansfield与Mary Lince对本书的出版所进行的专业运作。

美国陆军研究实验室（The Army Research Laboratory，ARL）、陆军研究局（Army Research Office，ARO）、能源部（Department of Energy，DoE）、国家宇航局（National Aeronautics and Space Administration，NASA）、国家科学基金会（The National Science Foundation，NSF）为作者Ananth Grama、George Karypis与Vipin Kumar提供了并行计算研究的支持。我们还要特别感谢Kamal Abdali、Michael Coyle、Jagdish Chandra、Frederica Darema、Stephen Davis、Wm Randolph Franklin、Richard Hirsch、Charles Koelbel、Raju Namburu、N. Radhakrishnan、John Van Rosendale、Subhash Saini与Xiaodong Zhang在并行计算领域为我们的研究项目所提供的支持。感谢IBM公司的Andrew Conn、Brenda Dietrich、John Forrest、David Jensen与Bill Pulleyblank多年来对作者Anshul Gupta研究工作的一贯支持。

目　录

出版者的话
专家指导委员会
中文版序
译者序
前言
第1章　并行计算介绍 ……1
1.1　推动并行化 ……1
1.1.1　计算能力因素——从晶体管到浮点运算速度 ……1
1.1.2　内存及磁盘速度的因素 ……2
1.1.3　数据通信因素 ……2
1.2　并行计算适用范围 ……3
1.2.1　在工程及设计中的应用 ……3
1.2.2　科学计算中的应用 ……3
1.2.3　商业应用 ……4
1.2.4　计算机系统中的应用 ……4
1.3　本书的组织及内容 ……4
1.4　书目评注 ……6
习题 ……6
第2章　并行编程平台 ……9
2.1　隐式并行:微处理器体系结构的发展趋势* ……9
2.1.1　流水线与超标量执行 ……9
2.1.2　超长指令字处理器 ……12
2.2　内存系统性能的局限* ……12
2.2.1　使用高速缓存改善有效内存延迟 ……13
2.2.2　内存带宽的影响 ……14
2.2.3　躲避内存延迟的其他方法 ……16
2.2.4　多线程与预取间的权衡 ……17
2.3　并行计算平台剖析 ……18
2.3.1　并行平台的控制结构 ……18
2.3.2　并行平台的通信模型 ……20
2.4　并行平台的物理组织 ……22
2.4.1　理想并行计算机的体系结构 ……22
2.4.2　并行计算机互连网络 ……23
2.4.3　网络拓扑结构 ……24
2.4.4　静态互连网络评价 ……31
2.4.5　动态互连网络评价 ……33
2.4.6　多处理器系统中的高速缓存一致性 ……34
2.5　并行计算机的通信成本 ……39
2.5.1　并行计算机的消息传递成本 ……39
2.5.2　共享地址空间计算机的通信成本 ……44
2.6　互连网络的路由选择机制 ……46
2.7　进程-处理器映射的影响和映射技术 ……47
2.7.1　图的映射技术 ……48
2.7.2　成本-性能平衡 ……53
2.8　书目评注 ……54
习题 ……55
第3章　并行算法设计原则 ……63
3.1　预备知识 ……63
3.1.1　分解、任务与依赖图 ……63
3.1.2　粒度、并发性与任务交互 ……65
3.1.3　进程和映射 ……69
3.1.4　进程与处理器 ……69
3.2　分解技术 ……70
3.2.1　递归分解 ……70
3.2.2　数据分解 ……72
3.2.3　探测性分解 ……77
3.2.4　推测性分解 ……79
3.2.5　混合分解 ……80
3.3　任务和交互的特点 ……81
3.3.1　任务特性 ……81
3.3.2　任务间交互的特征 ……82
3.4　负载平衡的映射技术 ……84
3.4.1　静态映射方案 ……85
3.4.2　动态映射方案 ……95
3.5　包含交互开销的方法 ……96
3.5.1　最大化数据本地性 ……97
3.5.2　最小化争用与热点 ……98

3.5.3 使计算与交互重叠……98
3.5.4 复制数据或计算……99
3.5.5 使用最优聚合交互操作 ……100
3.5.6 一些交互与另一些交互的重叠 ……100
3.6 并行算法模型……101
3.6.1 数据并行模型 ……101
3.6.2 任务图模型 ……101
3.6.3 工作池模型 ……102
3.6.4 主-从模型 ……102
3.6.5 流水线模型或生产者-消费者模型 ……103
3.6.6 混合模型 ……103
3.7 书目评注……103
习题……103
第4章 基本通信操作 ……107
4.1 一对多广播以及多对一归约……108
4.1.1 环或线性阵列 ……108
4.1.2 格网 ……110
4.1.3 超立方体 ……111
4.1.4 平衡二叉树 ……111
4.1.5 算法细节 ……112
4.1.6 成本分析 ……114
4.2 多对多广播和归约……114
4.2.1 线性阵列和环 ……115
4.2.2 格网 ……117
4.2.3 超立方体 ……117
4.2.4 成本分析 ……120
4.3 全归约与前缀和操作……121
4.4 散发和收集……123
4.5 多对多私自通信……125
4.5.1 环 ……126
4.5.2 格网 ……128
4.5.3 超立方体 ……128
4.6 循环移位……131
4.6.1 格网 ……131
4.6.2 超立方体 ……133
4.7 提高某些通信操作的速度……135
4.7.1 消息分裂和路由选择 ……135
4.7.2 全端口通信 ……136
4.8 小结……137
4.9 书目评注……138
习题 ……139
第5章 并行程序的解析建模 ……143
5.1 并行程序中的开销来源……143
5.2 并行系统的性能度量……144
5.2.1 执行时间 ……144
5.2.2 总并行开销 ……144
5.2.3 加速比 ……144
5.2.4 效率 ……147
5.2.5 成本 ……149
5.3 粒度对性能的影响……149
5.4 并行系统的可扩展性……152
5.4.1 并行程序的扩展特性 ……153
5.4.2 可扩展性的等效率度量 ……155
5.4.3 成本最优性和等效率函数 ……158
5.4.4 等效率函数的下界 ……159
5.4.5 并发度和等效率函数 ……159
5.5 最小执行时间和最小成本最优执行时间……159
5.6 并行程序渐近分析……162
5.7 其他可扩展性的度量……162
5.8 书目评注……165
习题 ……166
第6章 使用消息传递模式编程 ……171
6.1 消息传递编程的原理……171
6.2 操作构件：发送和接收操作……172
6.2.1 阻塞式消息传递操作 ……172
6.2.2 无阻塞式消息传递操作 ……175
6.3 MPI：消息传递接口 ……176
6.3.1 启动和终止MPI库 ……177
6.3.2 通信器 ……177
6.3.3 获取信息 ……178
6.3.4 发送和接收消息 ……178
6.3.5 实例：奇偶排序 ……182
6.4 拓扑结构与嵌入……184
6.4.1 创建和使用笛卡儿拓扑结构 ……184
6.4.2 实例：Cannon的矩阵与矩阵相乘 …185
6.5 计算与通信重叠……187

6.6 聚合的通信和计算操作……191
6.6.1 障碍……191
6.6.2 广播……191
6.6.3 归约……191
6.6.4 前缀……193
6.6.5 收集……193
6.6.6 散发……194
6.6.7 多对多……195
6.6.8 实例：一维矩阵与向量相乘……195
6.6.9 实例：单源最短路径……197
6.6.10 实例：样本排序……199
6.7 进程组和通信器……200
6.8 书目评注……203
习题……204
第7章 共享地址空间平台的编程……205
7.1 线程基础……205
7.2 为什么要用线程……206
7.3 POSIX 线程API……207
7.4 线程基础：创建和终止……207
7.5 Pthreads中的同步原语……210
7.5.1 共享变量的互斥……210
7.5.2 用于同步的条件变量……216
7.6 控制线程及同步的属性……219
7.6.1 线程的属性对象……219
7.6.2 互斥锁的属性对象……220
7.7 线程注销……221
7.8 复合同步结构……221
7.8.1 读-写锁……222
7.8.2 障碍……225
7.9 设计异步程序的技巧……228
7.10 OpenMP：基于命令的并行编程标准……229
7.10.1 OpenMP编程模型……229
7.10.2 在OpenMP中指定并发任务……232
7.10.3 OpenMP中的同步结构……237
7.10.4 OpenMP中的数据处理……240
7.10.5 OpenMP库函数……241
7.10.6 OpenMP中的环境变量……242
7.10.7 显式线程与基于OpenMP编程的比较……243
7.11 书目评注……243
习题……244
第8章 稠密矩阵算法……247
8.1 矩阵向量乘法……247
8.1.1 一维行划分……247
8.1.2 二维划分……250
8.2 矩阵与矩阵的乘法……253
8.2.1 简单的并行算法……254
8.2.2 Cannon算法……254
8.2.3 DNS算法……256
8.3 线性方程组求解……258
8.3.1 简单高斯消元算法……259
8.3.2 带部分主元选择的高斯消元算法……269
8.3.3 求解三角系统：回代法……271
8.3.4 求解线性方程组时的数值因素……272
8.4 书目评注……272
习题……273
第9章 排序……279
9.1 并行计算机中的排序问题……279
9.1.1 输入输出序列的存放位置……279
9.1.2 如何进行比较……280
9.2 排序网络……281
9.2.1 双调排序……282
9.2.2 将双调排序映射到超立方体和格网……285
9.3 冒泡排序及其变体……290
9.3.1 奇偶转换……290
9.3.2 希尔排序……293
9.4 快速排序……294
9.4.1 并行快速排序……295
9.4.2 用于CRCW PRAM的并行形式……296
9.4.3 用于实际体系结构的并行形式……298
9.4.4 主元选择……302
9.5 桶和样本排序……303
9.6 其他排序算法……305
9.6.1 枚举排序……305
9.6.2 基数排序……306

9.7 书目评注……307
习题……308
第10章 图算法……315
10.1 定义和表示……315
10.2 最小生成树：Prim 算法……317
10.3 单源最短路径：Dijkstra算法……321
10.4 全部顶点对间的最短路径……321
10.4.1 Dijkstra算法……322
10.4.2 Floyd算法……323
10.4.3 性能比较……327
10.5 传递闭包……327
10.6 连通分量……328
10.7 稀疏图算法……331
10.7.1 查找最大独立集……332
10.7.2 单源最短路径……334
10.8 书目评注……340
习题……341
第11章 离散优化问题的搜索算法……345
11.1 定义与实例……345
11.2 顺序搜索算法……349
11.2.1 深度优先搜索算法……349
11.2.2 最佳优先搜索算法……351
11.3 搜索开销因子……353
11.4 并行深度优先搜索……353
11.4.1 并行DFS的重要参数……355
11.4.2 并行DFS分析的一般框架……357
11.4.3 负载平衡方案分析……359
11.4.4 终止检测……360
11.4.5 试验结果……362
11.4.6 深度优先分支定界搜索的并行形式……364
11.4.7 IDA*的并行形式……365
11.5 并行最佳优先搜索……365
11.6 并行搜索算法的加速比异常……368
11.7 书目评注……371
习题……374
第12章 动态规划……379
12.1 动态规划概述……379
12.2 串行一元DP形式……381
12.2.1 最短路径问题……381
12.2.2 0/1背包问题……382
12.3 非串行一元DP形式……384
12.4 串行多元DP形式……387
12.5 非串行多元DP形式……387
12.6 综述与讨论……390
12.7 书目评注……391
习题……391
第13章 快速傅里叶变换……395
13.1 串行算法……395
13.2 二进制交换算法……398
13.2.1 全带宽网络……399
13.2.2 有限带宽网络……404
13.2.3 并行快速傅里叶变换中的额外计算……405
13.3 转置算法……407
13.3.1 二维转置算法……407
13.3.2 转置算法的推广……410
13.4 书目评注……413
习题……414
附录A 函数的复杂度与阶次分析……417
索引……419

第1章 并行计算介绍

在过去的十年里，微处理器技术有了巨大的发展。处理器的时钟频率从大约40 MHz（例如，1988年前后的MIPS R3000）增加到超过2 GHz（例如，2002年前后的奔腾IV）。同时，处理器现在能够在同一周期里执行多条指令。高端处理器的平均指令周期数（CPI）在过去的十年里提高了大约一个数量级。所有这一切导致峰值浮点运算执行速度（浮点运算次数每秒或FLOPS）增加了几个数量级。除此之外，同一时期计算机在其他方面也取得了重要的进展，其中最重要的进展可能要算内存系统向处理器提供数据速度的提高。计算机硬件和软件方面的重大革新有效地缓解了由数据通道和内存导致的瓶颈。

人们认识并发性对于加速计算单元的作用已经几十年了。然而，它们在提供多样性的数据通道、日益增加的存储单元（包括内存和外存）访问、可扩展性能以及较低成本方面的作用，只反映在并行计算的各种应用中。以2个、4个甚至8个处理器连接在一起的桌面计算机、工程用工作站以及计算服务器正成为通用的设计应用平台。大规模科学与工程计算要依靠更多处理器的并行计算机，其处理器数量通常达到几百台。像数据库或网络服务器这样的数据密集型平台，或者像事务处理及数据挖掘这样的应用，通常使用提供高聚合磁盘带宽的工作站机群。图形图像方面的应用往往用多个渲染管道和多个处理单元来计算及实时地渲染由几百万个多边形构成的现实场景。一些要求高可用性的应用程序要依靠并行或分布式平台提供冗余。因此，无论从性能、价格还是应用程序的需求来看，理解目前可用的各种各样并行计算平台的机理、工具以及编程技术，都是非常重要的。

1.1 推动并行化

并行软件的设计一直被认为要耗费大量的时间和精力，这可能主要由于说明和协调并行任务固有的复杂性、缺少可移植的算法、标准化的环境和软件开发工具包。当看到微处理器的快速发展时，人们不禁对花大力气研究并行化来加速应用程序的必要性产生怀疑。毕竟，如果花两年时间去开发一个并行程序，而在这段时间里硬件或软件平台已变得过时，显然这两年的开发就白费了。然而，硬件设计方面的某些明显趋势表明，单一处理器（或隐式并行）体系结构不能保证未来提升性能的速度，因为缺少隐式并行化以及存在数据通道及内存方面的其他瓶颈。同时，标准化的硬件接口减少了从设计微处理器到设计基于微处理器并行计算机的过渡时间。此外，在编程环境标准化方面取得的可观进展增加了并行应用程序的寿命。所有这一切都为发展并行计算平台提供了有力的支持。

1.1.1 计算能力因素——从晶体管到浮点运算速度

1965年，戈登·摩尔提出如下的观察结果：

“半导体上的晶体管数量大约每年增长了一倍。在短时间内这个增长速度可望保持下去。虽然没有理由相信在至少10年的时间内增长速度不能继续稳定，但对较长时间而言，多少难

以确定其增长速度。也就是说，到1975年，每块集成电路上的晶体管数将达到65 000个。”

他的推理是根据观察基于元件复杂度和时间之间的对数-线性经验关系中的三个数据点得
2 到的。他由此证明，到1975年，在只有四分之一平方英寸的单一芯片上含有65 000个晶体管是可能的。当1975年拥有65 000个晶体管的16K CCD存储器制造出来后，他的预测被证明是准确的。他在1975年接下来的文章里，把对数-线性关系改成了芯片尺寸、晶体管最小尺寸以及“电路与元件合理排列”之间的指数关系。接下来他写道：

“空间太少挤不进任何东西，除非布局更加合理化，我们只能依靠两个尺寸方面的因素——更大的芯片及更小的晶体管。”

他把元件数量每增长一倍的时间修改为18个月并预测从1975年起按这个降低后的速度增长。这就是著名的“摩尔定律”，也就是芯片的晶体管数量每18个月增长一倍。许多年以来，这条经验定律令人惊奇地在微处理器及动态随机存取存储器上反复地得到验证。通过将元件密度及芯片大小与计算能力相联系，摩尔定律推断出给定价格下计算能力每隔大约18个月翻一番。

在过去几年里，摩尔定律的局限性受到了广泛的争议。避开争议不谈，把晶体管数量转换成每秒运算次数（Operations Per Second, OPS）是至关重要的。虽然制造拥有巨大晶体管数量的元件已经成为可能，但如何利用这些晶体管以提高运算能力是对体系结构的挑战。一个合理的解决方案就是依靠并行化——隐式及显式并行。2.1节将简单地讨论隐式并行，本书的其他部分都将探讨显式并行。

1.1.2 内存及磁盘速度的因素

总体计算速度并非仅仅由处理器一个因素所决定，它也取决于内存系统向处理器提供数据的能力。在过去的十年里，当高端处理器的时钟速度大约以每年40%速度增长的同时，此期间动态随机存取存储器（DRAM）访问速度的年增长率仅为10%左右。伴随每个时钟周期处理指令数量的增加，处理器速度与访问内存之间的差异带来了巨大的性能瓶颈。这种处理器速度和动态随机存取存储器延迟之间越来越大的差距通常通过一种称为高速缓存的快速存储器来弥补，它依靠数据访问的局部性来提升内存系统的性能。除了延迟时间外，处理器与动态随机存取存储器间的净有效带宽也对持续计算速度产生影响。

内存系统的总体性能是由高速缓存能够满足总内存需求的部分所决定的。在2.2节对内存系统的性能做更详细的介绍。并行平台通常能提供更好的内存系统性能，因为它能提供（1）更大的集合高速缓存，（2）更高的内存系统的集合带宽（两者通常与处理器个数都是线性关系）。此外，并行算法的核心原则，即数据访问的局部性原则，也适用于便利高速缓存实现的
3 串行算法。这个论点可推广到用并行平台能对二级存储达到高集合带宽的磁盘。由此，并行算法产生发展非核心计算的见解。事实上，在一些发展最快的并行计算应用领域，如数据服务器（数据库服务器，网络服务器），已经不再过多地依靠它们的聚合计算能力，而更多的是依靠它们更快取出数据的能力。

1.1.3 数据通信因素

随着网络基础设施的不断发展，将因特网作为一个巨大的异构并行/分布式计算环境的构

想开始形成。许多应用很自然地采用了这种计算形式，一些大规模并行计算最令人关注的应用就属于这种广域网分布式平台范畴。外星智能探索（SETI）项目就使用许多家用计算机来分析来自外层空间的电磁信号，人们也尝试用这种方法对极大的整数进行因式分解和求解大规模的离散优化问题。

在许多应用程序中，因特网上的数据或资源的位置是有约束的，对分布在相对低带宽网络上的大型商业数据集进行数据挖掘就是这样的例子。在这样的应用中，即使无需重新安排并行计算，计算能力也可用来完成所需的任务，但想要在一个中心位置收集数据也许是不可行的。在这些情况下，并行化的动机不仅来自对计算资源的要求，而且也来自改变（集中化）方法的不可行性或不必要性。

1.2 并行计算适用范围

从科学及工程应用的计算模拟，到商业应用的数据挖掘及事务处理等许多领域，并行计算已经产生了巨大的影响。并行化的成本优势与应用对性能上的需求相结合，为促进并行计算提供了令人信服的论据。下面给出并行计算在诸多应用方面的例子。

1.2.1 在工程及设计中的应用

并行计算传统上已成功地应用于机翼设计（优化升力、阻力、稳定性），内燃机设计（优化负载分布、燃烧），高速电路设计（设计时延、电容以及电感效应）以及结构设计（优化结构完整性、设计参数、成本等）等方面。最近，微电机系统及毫微电机系统（MEMS及NEMS）的设计引起了人们极大的重视。虽然绝大多数工程和设计中的应用提出了多空间与多时间尺度以及耦合物理现象的问题，但对MEMS/NEMS设计，这些问题更加突出。即使在一个单一的系统里，也经常要处理与量子现象、分子动力学以及像传导、对流、辐射和结构力学等物理过程的随机模型及连续模型有关的混合问题。这对几何建模、数学建模、建立算法以及所有与并行计算机有关的方面都提出了巨大的挑战。

在工程与设计方面的其他应用则着重对各种过程进行优化。并行计算机已被用来解决许多离散及连续优化问题。如像线性最优化和分支定界中的单纯型法和内点法以及离散最优化中的遗传规划这样一些算法，都已被有效地并行化和频繁使用。

1.2.2 科学计算中的应用

过去几年，在高性能科学计算应用方面取得了革命性的进展。国际人类基因测序联盟和Celera公司对人类基因的测序，在生物信息学方面开辟了令人激动的新领域。理解基因及蛋白质的功能及结构上的特点能使人类理解并根本性地影响生物学过程。以开发新的药品、治疗疾病及改善医疗条件为目的的生物序列分析既需要大规模计算能力，也需要创新的算法。实际上，一些最新的并行计算技术就是专门针对生物信息学方面的应用而发展的。

在计算物理学及计算化学方面的进展强调对尺度从量子现象到大分子结构之间变化过程的理解。这已导致设计新材料、理解化学反应途径和更有效的反应过程。在天体物理学方面的应用包括对银河系的演化过程及热核反应过程的研究，以及对来自天文望远镜的巨大数据集的分析等。并行计算还广泛应用于气象模型、矿物勘探以及洪水预测等领域并对人类日常生活产生极大的影响。

由于要分析巨大的数据集，生物信息学和天体物理学提出了一些极富挑战性的问题。蛋白质和基因数据库（如PDB、SwissProt 、ENTREZ 以及NDB）和天体观测数据集（如Sloan数字天体观测）代表了一些最大的科学数据集。有效地分析这些数据集需要巨大的计算能力，有效地分析这些数据集，也是取得重大科学发现的关键。

1.2.3 商业应用

随着网络和静态与动态信息的广泛使用，人们对能提供可扩展性能的具有成本-效益的服
5 务器的要求日益增加。从多处理器到Linux 机群的各种并行平台通常被用作网络和数据库服务器。例如，在事务繁忙的日子，华尔街的大经纪行要处理成千上万个同时发生的用户交易及数以百万计的订单。处理这些商业事务的服务器都是一些超级计算机，如IBM SP，以及Sun的Ultra HPC服务器等。一些最大的超级计算机网络就建立在华尔街。

利用大规模事务数据在优化商业和开拓市场的数据挖掘和分析方面产生了巨大的商业利益。巨大的数据量以及数据的地理上分布特性要求采用有效的并行算法来解决诸如规则挖掘、聚集、分类以及时间-序列分析等问题。

1.2.4 计算机系统中的应用

随着计算机系统的广泛普及以及计算在网络上的流行，并行计算也深入到了各个应用领域。在计算机安全方面，入侵检测是一个突出的难题。在网络入侵检测中，计算机从分布的站点收集数据，并对信号入侵迅速分析。由于在一个中心位置收集数据进行分析是不可行的，所以要求发展有效的并行及分布式算法。在密码学领域，某些最引人注目的基于因特网的并行计算应用就是对极大的整数分解。

嵌入式系统越来越多地依靠分布式控制算法来完成不同的任务。一辆现代化的汽车中拥有数十个互连的微处理器以完成复杂的任务，从而优化汽车操作及性能。在这样的系统中，经常用到一些传统的并行及分布式算法来处理如优先选择、最大独立集等问题。

虽然传统的并行计算限于在具有完善的计算及网络单元的平台上进行，其中，故障和错误不起重要作用，但是对特殊的、移动的或有故障的环境的并行计算能提供有价值的经验。

1.3 本书的组织及内容

本书提供用并行计算机求解问题的全面且完备的方案。算法及度量强调并行计算机的实用和可移植模型。算法设计原则强调并行算法与技巧的必需属性，并在大量的应用和体系结构中获得这些属性。编程技巧则涵盖如MPI和POSIX线程之类的标准范例，这些范例可用在许多并行平台。

本书各章可以分成如图1-1所示的四个主要部分。这四部分的内容如下:

6 **基础篇** 包括第2章到第4章。第2章并行编程平台，讨论并行平台的物理结构。建立用于算法设计的成本度量。这一章的目的并非为并行体系结构提供一个完美的解决方案，而是为有效地使用这些机器提供所需的细节。第3章并行算法设计原则，讨论实现有效并行算法的关键因素，并提供一套可以应用于各种应用程序中的设计方法。第4章基本通信操作，讲述用于全书的有助于在并行计算中进行有效数据传输的一组核心操作。最后，第5章并行程序的解析建模，论述量化并行算法性能的各种度量。

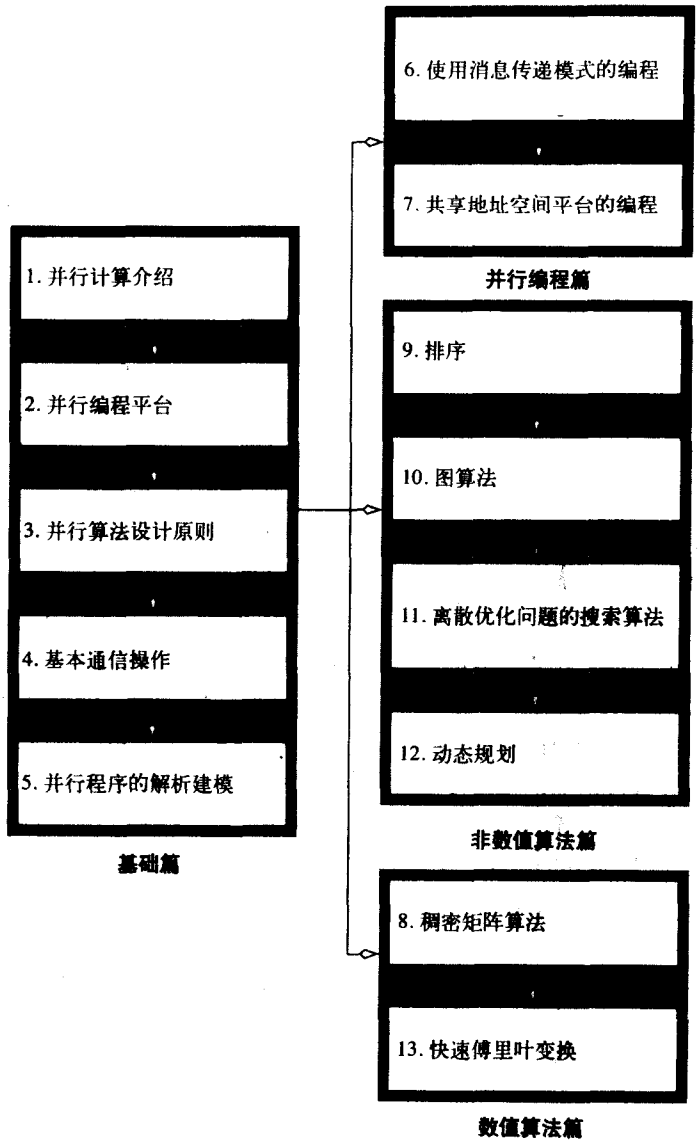


图1-1 本书各章的推荐阅读顺序

7

并行编程篇 包括第6章及第7章。第6章使用消息传递模式的编程，着重于消息传递平台（包含机群）编程的消息传递接口（MPI）。第7章共享地址空间平台的编程，讲述如线程及基于命令方法的编程模式。利用如POSIX线程及OpenMP等模式，描述在共享地址空间并行计算机编程所必需的各种要素。这两章使用大量并行程序的例子阐述不同的编程概念。

非数值算法篇 包括第9～12章，讲述非数值并行算法。第9章排序，讲述双调排序、冒泡排序及其变体、快速排序、样本排序以及希尔排序等。第10章图算法，讲述各种图论问题的算法，如最小生成树、最短路径以及连通分量等。这一章还讨论稀疏图。第11章离散优化问题的搜索算法，讲述组合问题中基于搜索的方法，如分支定界及启发式搜索等。第12章动态规划，对各种动态规划算法进行分类并提出并行公式。

数值算法篇 包括第8章和第13章，讲述并行数值算法。第8章稠密矩阵算法，讲述稠密矩阵的一些基本运算，如矩阵乘法、矩阵向量乘法以及高斯消元法等。这一章放在非数值算法篇之前，是因为对于许多非数值算法来说，分割并分配矩阵给处理机对许多非数值算法是常见的技术。而且，矩阵向量乘法算法及矩阵乘法算法是许多图论算法的核心。第13章快速傅里叶变换，讲述计算快速傅里叶变换的算法。

1.4 书目评注

许多书籍在不同层次上讨论并行处理。有几本教科书及专著广泛讨论了并行计算机的硬
8 件方面[CSG98, LW95, HX98, AG94, Fly95, AG94, Sto93, DeC89, HB84, RF89, Sie85, Tab90, Tab91, WF84, Woo86] ⊖。许多教材讨论并行计算机编程语言及范例[LB98, Pac98, GLS99, GSNL98, CDK+00, WA98, And91, BA82, Bab88, Ble90, Con89, CT92, Les93, Per87, Wal91]。Akl [Akl97]、Cole [Col89]、Gibbsons 和Rytter [GR90]、Foster [Fos95]、Leighton [Lei92]、Miller和Stout [MS96]以及Quinn [Qui94]讨论并行算法设计与分析的诸多方面。Buyya（编辑）[Buy99]和Pfister讨论使用机群的并行计算。Jaja [Jaj92]讨论用于PRAM计算模型的并行算法。Hillis [Hil85, HS86]、Hatcher和Quinn [HQ91]讨论数据并行编程。Agha [Agh86]讨论一个基于角色（actor）的并发计算模型。Sharp [Sha85]讨论数据流计算。有些书籍提供关于并行计算课题的总览[CL93, Fou94, Zom96, JGD87, LER92, Mol93, Qui94]。许多书籍讲述数值分析与科学计算方面并行处理的应用[DDSV99, FJDS96, GO93, Car89]。Fox et al. [FJL+88]以及Angus et al. [AFKW90]对于科学计算中的问题提出一种面向应用的算法设计观点。Bertsekas和Tsitsiklis [BT97]着重讨论数值应用方面的并行算法。

Akl和Lyons[AL93]讨论计算几何中的并行算法。Ranka和Sahni [RS90b]以及Dew，Earnshaw和Heywood [DEH89]讲述计算机视觉方面的并行算法。Green[Gre91]讲述图形应用中的并行算法。许多书籍讲述并行处理在人工智能方面的应用[Gup87, HD89b, KGK90, KKKS94, Kow88, RZ89]。

计算机协会（Association for Computing Machinery，ACM）把许多有用的综述、参考书目以及索引收集到一起 [ACM91]。Messina和Murli [MM91]收集许多关于并行计算在各方面应用以及潜在应用领域的文章。在NSF报告[NSF91, GOV99]中还讨论了并行计算的应用范围以及美国政府对并行计算多方面的支持。

许多会议讨论并行计算的各个方面。重要的会议有超级计算会议（Supercomputing Conference, SC），并行算法及体系结构的ACM讨论会，并行处理国际会议，并行及分布式处理国际讨论会，并行计算以及关于并行处理的SIAM会议。并行处理的重要期刊包括《IEEE并行与分布式系统学报》，《并行编程国际杂志》，《并行及分布式计算杂志》，《并行计算》，《IEEE并发及并行处理快报》。这些会议的会议录和杂志为并行处理的发展水平提供了丰富的信息资源。

习题

1.1 访问世界前500名超级计算机站点（http://www.top500.org），列出功能最强大的前5

⊖ 注：原英文版的“参考文献”部分的电子文件现放在http://www.hzbook.com上，有需要的读者请到网站上下载。

台超级计算机及它们的FLOPS。 9

1.2 在以下领域，列出需要使用超级计算机的三个主要问题:

(i) 结构力学

(ii) 计算生物学

(iii) 商业应用

1.3 收集若干年来集成电路发展过程中元件数目的统计信息，画出元件数目作为时间函数的曲线，并把增长率与摩尔定律进行比较。 10

1.4 对处理器峰值浮点运算速度重复上一题的试验，并把运算速度与摩尔定律进行比较。

第2章 并行编程平台

传统意义上的串行计算机是由经数据通道连接到处理器的内存所组成的。所有这三个部件——处理器、内存以及数据通道——形成计算机系统总体处理速度的瓶颈。多年来，许多体系结构上的革新都是为了解决这些瓶颈。其中最重要的革新之一就是使用多处理器、多数据通道以及多内存单元。这种多重性对于程序员来说，要么像隐式并行性一样完全隐藏，要么以不同的形式提供给程序员。本章将对与并行处理相关的重要的体系结构概念作一综述，其目的是为了给程序员提供足够的细节，使他们能在各种各样的平台上写出高效的代码。本章提出用来量化不同并行算法性能的成本模型和抽象，并辨识由各种编程结构导致的瓶颈。

本章对并行平台的讨论将从简单地回顾串行及隐式并行体系结构开始。这是因为，在很多情况下用简单的程序转换工具得到的代码会有显著的速度提升（比未优化时的速度快2倍至5倍）。对次优化的串行代码并行化，往往会导致一些不良后果，如不可靠的速度提升，易使人误解的运行时间等。因此，在对代码并行化之前，最好先优化串行代码的性能。正如本章所示，对代码的串行优化及并行优化有着非常相似的特点。在讨论串行及隐式并行体系结构后，本章的其余部分将讨论并行平台的组织、算法的基本成本模型以及用于可移植算法设计的平台抽象等。想直接研究并行体系结构的读者可以跳过2.1节和2.2节。

2.1 隐式并行：微处理器体系结构的发展趋势*

在过去的10年间，虽然微处理器技术在时钟频率上取得了重大进展，但也遇到了多种性能瓶颈。为减小瓶颈，微处理器设计者尝试了多种方法，以获得有成本-效益的性能。本节将对这些方法作出概括，了解这些方法的局限性以及它们对算法及代码开发的影响。本节不是为了对处理器的体系结构作全面的描述。关于处理器的体系结构，参考书目中提到几本很好的教科书。

在过去的20年里，微处理器的时钟频率提高了2至3个数量级。然而，内存技术的限制严重地抵消了时钟速度的提高。同时，设备的高度集成也导致晶体管数目变得非常巨大，如何最好地发挥它们的作用便成了明显的问题。结果，允许在一个时钟周期内执行多个指令已成为普遍的技术。事实上，在如今一代的微处理器，如Itanium、Sparc Ultra、MIPS以及Power4中，这种趋势都是显而易见的。本节将简单地研究各种处理器用于支持多指令执行的机制。

2.1.1 流水线与超标量执行

处理器长时间以来依靠流水线来提升执行速度。通过在指令执行过程中重叠不同的阶段（取指令、调度、译码、取操作数、执行以及存储等），流水线操作能使执行速度加快。装配线与流水线的机理类似。如果装配一辆汽车需要100个时间单元，将它分成10个流水线阶段，每个阶段10个时间单元，那么一条装配线每10个时间单元就能生产一辆汽车！这表明比完全按串行方式一步接一步地生产汽车的速度提高了10倍。从这个例子还可以看到，为了提高单个流水线的速度，可以将任务分成更小的单元，这样就能使流水线变长，增加执行时的重叠

时间。对处理器而言，因为现在的任务变小了，这样的做法能使时钟的速度更快。例如，2.0 GHz的奔腾IV处理器就有一个20阶段的流水线。注意，单流水线的速度最终受到流水线中最大原子任务的限制。而且，对典型的指令跟踪后发现，每隔5到6条指令就是一个分支指令，因此，长指令流水线需要有效的技术来预示分支的目的地，使得流水线在理论上是满的。当流水线深度变大，大量指令需要清除时，预示错误的后果就会增加，这些因素限制处理器流
12 水线的深度，进而影响处理器的性能。

除此之外使用多个流水线是提高指令指令执行速度的一个有效方法。在每个时钟周期，多条指令被并行地提交给处理器。这些指令在多个功能部件执行。下面用例子来说明这个过程。

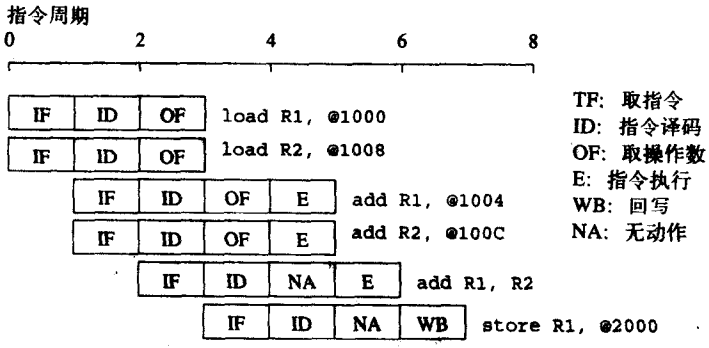
例2.1 超标量执行

设某处理器具有两条流水线，并能同时发送两条指令。这种处理器有时称为超流水线处理器。处理器在同一周期发送多条指令的能力称为超标量执行。由于图2-1的体系结构允许在每个时钟周期发送两条指令，它也称为两路超标量或双指令发送执行。

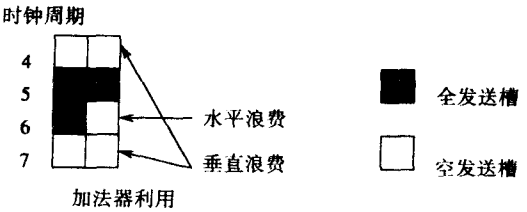
```
1. load R1, @1000      1. load R1, @1000      1. load R1, @1000
2. load R2, @1008      2. add R1, @1004       2. add R1, @1004
3. add R1, @1004       3. add R1, @1008       3. load R2, @1008
4. add R2, @100C       4. add R1, @100C       4. add R2, @100C
5. add R1, R2          5. store R1, @2000     5. add R1, R2
6. store R1, @2000                            6. store R1, @2000
```

(i) (ii) (iii)

a) 4个数相加的三种不同的代码段



b) 代码段(i)的执行调度



c) 流程b)中调度的硬件利用跟踪

13 图2-1 两路超标量指令执行示例

考察图2-1中4个数加法的第一个代码段。第一条指令与第二条指令相互独立，因此这两条指令可以同时发送。如图所示在t = 0同时发出了load R1，@1000与load R2，@1008两条指令，随后取出指令、对指令译码以及取操作数。接下来的两条指令add R1，@1004与 add R2，@100C虽然必须在最开始的两条指令后执行，但它们也是彼此独立的。由于处理器的流水线操作，在t = 1时这两条指令也同时发送。这些指令在t = 5时终止。接下来的两条指令add R1，R2以及store R1，@2000不能同时执行，因为第二条指令要用到第一条指令的结果（R1寄存器的内容）。因此，只有add指令在t = 2时发送，store指令在t = 3时发送。注意add R1，R2指令只有在它的前两条指令执行完毕后才能执行。指令调度如图2-1b所示。该调度假设每一次内存访问占用一个时钟周期。事实上并非一定如此。这个假设隐含的内容在2.2节的内存系统性能部分讨论。 ■

超标量执行的原理是很自然的，甚至是简单的。然而，关于它还有很多问题需要解决。首先，如例2.1所示，程序中的指令很可能彼此相关。某一指令的执行结果可能要被它的后续指令用到。这种情况称为*真实数据相关性*（true data dependency）。例如，考虑图2-1中4个数字相加的第二个代码段。在load R1，@1000与add R1，@1004之间就存在真实数据相关性，后面的指令间也存在类似的相关性。在同时发送指令前，这种类型的相关性必需解除。这句话有两层含义：首先，由于要在运行时解除相关性，必需要有硬件支持。这样硬件就会变得很复杂。其次，程序中指令级并行的数量通常是有限的，可以通过编程技巧解决。在第二个代码段，不能同时发送指令，这样资源的利用率就会很低。图2-1a中的三个代码段也表明，在许多情况下，如果重排指令的顺序并且改变代码，便有可能实现更高的并行性。注意在这个例子中，代码的重新组织对应于可由指令发送机制利用的一种并行形式。

另一种指令间的相关性是由多个流水线分享有限的资源造成的。例如，在只有一个浮点部件的双指令发送机器上同时执行两个浮点操作，虽然指令间可能不存在数据相关性，但两个操作也不能同时进行，因为它们都需要用到浮点部件。这种两条指令竞争单一处理器资源造成的相关性称为*资源相关性*（resource dependency）。

程序间的控制流程造成指令间的第三种相关性。当执行一个条件分支指令时，只有在执行点才知道分支将转向何处，而事先指令通过在分支处的转向就可能导致错误。这种相关性称为*分支相关性*（branch dependency）或*过程相关性*（procedural dependency）。通常在通过 14
分支处用推测调度表处理，并且一旦出错就回退。典型的跟踪研究表明，平均每隔5到6条指令就会遇到一条分支指令。因此，如在集中的指令流水线中一样，对于有效的超标量执行来说，准确的分支预测是非常重要的。

对于超标量执行来说，处理器检测及调度并发指令的能力是很重要的。例如，图2-1中第三个代码段也计算4个数的和。读者会发现其实它是第一个代码段语义上等价的重新排列。然而，在这种情况下，开始的两条指令load R1，@1000与add R1，@1004之间存在数据相关性。因此，这两条指令不能被同时发送。但是，如果处理器具有预测能力，它就可能在调度第一条指令的同时调度第三条指令load R2，@1008。在下一个发送周期，指令2与指令4也能同时调度。这样第一个代码段与第三个代码段具有同样的执行调度。然而，处理器必须具有*乱序*（out-of-order）发送指令的能力，才能完成需要的指令顺序重排。就上面的例子而言，只能*依序*（in-order）发送指令的并行化就难以达到这一步。绝大多数流行的处理器能够乱序发送指令并完成指令。这种模型也称为*动态指令发送*，利用最大指令级并行性。处理器

使用一个指令窗口，从中选择同时发送的指令。该窗口对应于调度程序的预测。

超标量体系结构的性能受到可用指令级并行性的限制。考虑图2-1中的例子。为使讨论简单化，我们忽略例子中的流水线操作方面，只注重程序的执行方面。假设有两个执行部件（乘法-加法部件），图中说明有几个零发送周期（浮点部件空闲的周期）。从执行部件的角度来说，这些周期实际上都被浪费掉了。如果某一周期中执行部件没有指令发送，就称为*垂直浪费*（vertical waste）；如果一个周期中只用到部分执行部件，就称为*水平浪费*（horizontal waste）。本例中有两个周期的垂直浪费及一个周期的水平浪费，总共8个可用周期中只有3个用于计算。这就意味着代码段的运算速度不会超过处理器峰值浮点运算速度的3/8。通常，并行性的限制、资源相关性以及处理器不能实现并行性，都会造成超标量处理器资源严重利用不足。当前的微处理器通常支持最高到4指令发送的超标量执行。

2.1.2 超长指令字处理器

由超标量处理器抽出的并行性通常受到指令预测的限制。在常规处理器中，用于动态相
15 关性分析的硬件逻辑一般占总逻辑的5%～10%（大约是4路超标量处理器Sun UltraSPARC的5%），复杂度大约以发送指令数的二次方增长，并可能导致瓶颈。在超长指令字（VLIW）处理器中使用的利用指令级并行性的另一种概念，依赖于编译时解决相关性与资源可用性的编译器。它把能够并发执行的指令并入到一个组里，作为一个超长指令字（这也是名称的由来）分发给处理器，并同时在多个功能部件上执行。

VLIW的概念最初出现在 Multiflow Trace（circa 1984）里，然后它的一种变体出现在Intel的IA64体系结构中，与超标量处理器相比，它既有优点也有缺点。因为调度用软件来完成，在VLIW处理器中的解码与指令发送机制都要简单一些。与硬件发送部件相比，编译器有更大的指令语句选择余地，可以用许许多多的变换来优化并行处理。其他的并行指令通常被编译器用来控制并行执行。然而，编译器中没有动态程序状态（如分支历史缓冲器）来对调度进行决策，这样就降低了分支与内存预测的准确性，但是能使用更复杂的静态预测方法。其他运行期的情况，如因高速缓存没有命中而造成的取数据故障，则更加难以准确预测。这就限制了基于编译器静态调度的应用范围及性能。

最后，VLIW处理器的性能对编译器检测数据及资源相关性的能力、读写故障和为实现最大并行性而调度指令等都非常敏感。循环的跳出、分支预测以及推测性执行等对VLIW处理器的性能来说都至关重要。虽然超标量与VLIW处理器对利用隐式并行性非常成功，但它们通常都局限在4路到8路并行的小规模并发处理上。

2.2 内存系统性能的局限*

计算机中程序的有效性能不仅依赖于处理器的速度，还依赖内存系统向处理器传送数据的能力。从逻辑层次上讲，内存系统可能由多级缓存构成，当它收到一条内存字请求，在l纳秒的延迟后，会返回大小为b的包含请求字的数据块。这里，l称为内存的*延迟*（latency）。数据从内存送到处理器的速度决定了内存系统的*带宽*（bandwidth）。

理解延迟与带宽的区别是非常重要的，因为不同的通常相互竞争的技术需要处理这种区别。打个比方说，如果打开消防龙头后2秒钟水才从消防水管的尽头流出，那么这个系统的延迟就是2秒。水流开始后，如果水管1秒钟能流出1加仑的水，那么这个水管的“带宽”就是1

加仑/秒。如果想立刻扑灭火灾，延迟时间应该更小。这样就需要消防栓处有更大的水压。另 16
一方面，如果希望扑灭更大的火，就需要更大的水流速度，这样就要求更大的水管及消防栓。这一切和内存系统的运作方式很相似。延迟与带宽都对内存系统的性能起着关键作用。下面将用几个例子更详细地分别研究这两个方面。

为研究内存系统延迟的作用，在下面的例子中假设内存块由一个字组成。在后面研究内存带宽时放宽这个假设。由于我们只对可获得最佳性能感兴趣，我们同时假设使用最佳高速缓存替代策略。对内存系统设计的详细讨论，请读者阅读有关的参考文献。

例2.2 内存延迟对性能的影响

考虑某一处理器以1 GHz（1纳秒时钟）运行，与之相连的DRAM有100纳秒的延迟（无高速缓存）。假设处理器有两个乘法-加法部件，在每个1纳秒的周期内能执行4条指令。这样处理器的峰值速度就是4 GFLOPS。由于内存延迟时间等于100个周期，并且块大小为一个字，每次发送内存请求时，处理器在处理数据前必需等待100个周期。考虑在这样的平台上计算两个向量的点积。计算点积对每对向量元素进行一次乘法-加法运算，即每一次浮点运算需要取一次数据。很显然这种浮点运算的最大速度仅仅为每100纳秒1次，或10 MFLOPS，只是处理器峰值速度的很小部分。这个例子非常有力地说明有效内存系统性能对获得高运算速度的作用。 ■

2.2.1 使用高速缓存改善有效内存延迟

处理器与DRAM速度的不匹配促进了内存系统设计中许多体系结构上的创新。其中一种创新便是在处理器与DRAM之间放置更小更快的内存。这种内存称为高速缓存，是一种低延迟高带宽的存储器。处理器需要的数据首先被取到高速缓存中，以后所有对高速缓存中数据的访问都由高速缓存来完成。这样，从理论上说，如果某一数据被重复使用，高速缓存就能减少内存系统的有效延迟。由高速缓存提供的数据份额称为高速缓存*命中率*（hit ratio）。许多应用程序有效的运算速度不仅仅受CPU处理速度的限制，还受能送到CPU的数据传送速度的限制。这种运算称为*内存受限*（memory bound）的运算。高速缓存命中率严重影响受内存受限程序的性能。 17

例2.3 高速缓存对内存系统性能的影响

如上一个例子那样，处理器时钟频率仍为1 GHz，DRAM延迟仍为100纳秒。这个例子中，加入一个32 KB和延迟时间为1纳秒或一个时钟周期的高速缓存（通常集成在处理器上）。用这样的配置进行两个32 × 32的矩阵A和矩阵B相乘。精心选择这些数字，使高速缓存能存储矩阵A和矩阵B及结果矩阵C。另外，假设高速缓存替代策略是理想的，这样就能避免数据项被其他项覆盖。将两个矩阵取到高速缓存中等同于取2K字，这大约需要200微秒。两个$n \times n$的矩阵相乘需要进行$2n^3$次运算，这样本例就需要64K次操作，如果每个周期执行4条指令，就需要16K周期（或16微秒）。因此总计算时间大约是加载时间、存储时间以及计算时间之和，即200+16微秒。这与最大计算速度64K/216或303 MFLOPS对应。注意，虽然还不到处理器峰值性能的10%，但这是前一个例子的30倍。从这个例子可以看到，虽然只是放置了一小块高速缓存，却显著地提高了处理器的利用率。 ■

利用高速缓存达到性能提升必需假定能够重复引用同一数据项。在一段短的时间间隔内重复引用数据项的概念称为引用的时间本地性（temporal locality），在这个例子中，有$O(n^2)$次数据访问及$O(n^3)$次计算（附录中有关于O的解释）。对高速缓存的性能来说，数据重用非常重要，因为如果数据项只用到了一次，那么使用一次后还要到DRAM中取其他数据，这样每次操作都要遇到DRAM延迟。

2.2.2 内存带宽的影响

内存带宽指的是数据在内存与处理器间传送的速度，它是由内存总线的带宽及内存部件决定的。一个常用的提高内存带宽的技术是增加内存块的大小。为了解释清楚所讨论的问题，先放松对内存块大小的简化限制，并假设某一内存请求返回4个字的连续块。这种4个字的单一单元也称为高速缓存行（cache line）。普通计算机经常取2至8个字到高速缓存中。我们会看到，这种技术有助于提高那些数据重用受限制的应用程序的性能。

例2.4 块大小的影响：两个向量的点积

18 再次假设内存系统的高速缓存为1个周期，处理器时钟频率为1 GHz，DRAM延迟为100纳秒。如果块大小为一个字，则处理器需要100个周期来取每一个字。对每对字来说，点积需要一次乘法-加法运算，即2FLOP。因此，每隔100个周期进行一次浮点运算，正如例2.2所示，运算的最大速度为10 MFLOPS。

如果块大小增加到4个字，即处理器每100个周期能取4个字的高速缓存行。假设向量在内存中线性排列，那么在200个周期内能进行8次浮点运算（4次乘法-加法）。这是因为一次内存访问取出向量中4个连续的字。因此，两次访问能取出每个向量中的4个元素。这等同于25纳秒进行一次浮点运算，或40 MFLOPS的最大速度。把块大小从1个字增加到4个字并没有改变内存系统的延迟时间。但是，这样把带宽提高4倍。在这种情况下，虽然点积算法没有数据重用，但内存系统带宽的提高加快了运算速度。

快速估算性能界限的另一种方法是估算高速缓存的命中率，用它来计算每个字的平均访问时间，并将此时间与基本算法的浮点运算速度相联系。例如，在这个例子中，算法要求每访问8个数据需要访问2次DRAM（高速缓存失误）。这对应于高速缓存命中率75%。假设最主要的时间开销由高速缓存失误造成，那么由失误造成的平均内存访问时间就是100纳秒的25%（或25纳秒/字）。由于点积对每个字有一次运算，同前面一样，这与计算速度40 MFLOPS对应。这个速度的更精确估计将计算出平均内存访问时间为0.75 × 1+0.25 × 100=25.75纳秒/字，对应的计算速度为38.8 MFLOPS。 ■

从物理上讲，例2.4的情况对应与多个存储区相连接的宽数据总线（4个字或128位）。实际上，构建这么宽的总线代价昂贵。在更切实可行的系统中，得到第一个字后，连续的字在紧接着的总线周期里被送到内存总线。例如，若数据总线为32位，第一个字在100纳秒的延迟后被送到内存总线，其后每个总线周期送入总线一个字。这样计算方法与上面所讲的略有不同，因为整个高速缓存行在100+3 × (内存总线周期)纳秒后才可用。如果数据总线以200 MHz运行，高速缓存行的访问时间就增加15纳秒。这并没有显著地改变执行速度的限制。

上面的例子清楚地说明增加带宽对提高峰值计算速度的影响。对程序员来说这些例子所作假设也非常重要。对数据的布局作这样的假设，内存中连续的数据字被连续的指令使用。

换句话说，如果以计算为中心，那么就有内存访问的空间本地性（spatial locality）。如果以数据布局为中心，那么计算按这样的顺序，连续的计算需要连续的数据。如果计算（或访问模式）不具有空间本地性，那么有效带宽就会比最大带宽小得多。 19

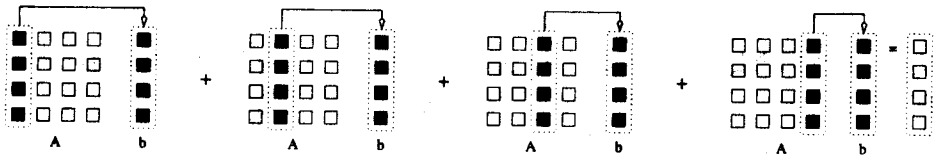
下面的例子讲述了这样的存储模式，稠密矩阵以行为主的方式存储在内存中，但是按列的方式读出。编译器可以对计算进行重新组织，充分利用空间本地性以达到好的效果。

例2.5 跨距访问的影响

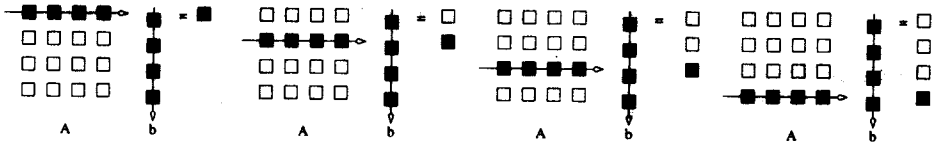
考察下面的代码段：

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

该段代码对矩阵b的列求和，把所得的结果送到向量column_sum中。从中可以看出两点：(i) 向量column_sum很小，很容易放到高速缓存中；(ii) 如图2-2a所示，按列顺序访问矩阵b。对于一个以列为主存储的1000 × 1000的矩阵，这意味着要访问每一起第1000个数据。因此，从内存中取出的每个高速缓存行中的数据只有一个字被用到。因此，上面代码段的性能很差。 ■



a) 以列为主的数据访问



b) 以行为主的数据访问

图2-2 向量与矩阵相乘：a) 列与列相乘，保留运行中的和；
b) 计算矩阵每一行与向量的点积作为结果的元素

上面的例子说明跨距访问的缺点（跨距大于1）。计算中空间本地性的缺少会导致内存系统性能变差。通常对计算重新组织就能消除跨距访问。在以下例子中，可重写循环。 20

例2.6 消除跨距访问

考察下面重组后的列求和代码段：

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

在本例中，矩阵按行序遍历，如图2-2 b所示。然而，读者会发现，该代码段依靠通过循环在高速缓存中保留向量column_sum。对这样一个特例来说，这样的假设是合理的。如果

向量更大，就必须将迭代空间分成多个块，然后一次计算一个块的乘积，这个概念称之为平铺（tiling）迭代空间。如何改善这个循环的性能留给读者作为习题。 ■

下一个问题是，我们是否已解决了由内存延迟及带宽造成的问题？在过去几十年里，处理器的最高速度显著提高，但内存延迟及带宽未能跟上处理器的步伐。对普通计算机来说，峰值FLOPS速度与峰值内存带宽之比介于1 MFLOPS/MB（该比率指峰值FLOPS 每兆字节/秒带宽）到100 MFLOPS/MB之间，大规模向量超级计算机的这个比值较低，而基于快速微处理器的计算机的比值较高。这个数字说明，一个字被取到完全带宽存储器（通常为L1高速缓存）后，平均必须重用100次才能使处理器达到最大利用率。这里，完全带宽定义为一个计算达到它的处理器的限制时所需的数据传输速度。

本节中一系列例子说明以下几个概念：

- 利用应用程序的空间本地性与时间本地性对于减少内存延迟及提高有效内存带宽非常重要。
- 某些应用程序具有较大的时间本地性，因此更能承受较低的内存带宽。计算次数与内存访问次数的比是一个很好的预测内存带宽的承受程度的指标。
- 内存的布局以及合理组织计算次序能对空间本地性和时间本地性产生重大影响。

2.2.3 躲避内存延迟的其他方法

21

如果在网络传输高峰期浏览网页，对浏览器的响应不足，可以使用下面的三种简单方法之一来缓解：（i）预测要浏览的网页，并提前发送请求；（ii）打开多个浏览器，在每个浏览器中访问不同的网页，这样当等待某个页面载入时，可以先看其他的页面；（iii）一次访问多个页面——在多个访问中分摊延迟。第一种方法称为预取（prefetching），第二种方法称为多线程（multithreading），第三种方法与访问内存时的空间本地性有关。在这三种方法中，前面已经讨论过内存访问的空间本地性。这里只介绍预取和多线程两种躲避延迟的方法。

1. 多线程躲避延迟

线程是程序流程中的单一控制流。下面是一个解释线程的简单例子：

例2.7 矩阵乘法的线程执行

下面的代码段将一个$n \times n$的矩阵与向量b相乘得到向量c。

```
1   for (i = 0; i < n; i++)
2       c[i] = dot_product(get_row(a, i), b);
```

该代码将c中的每一个元素作为a中相应行与向量c的点积。由于每一个点积是相互独立的，可以表示为并发的执行单元。可以安全地改写代码如下：

```
1   for (i = 0; i < n; i++)
2       c[i] = create_thread(dot_product, get_row(a, i), b);
```

两段代码的唯一区别是，后者明确指定每一次点积计算为一个线程。（在第7章会讲到许多指定线程的API，这里只是从字面意义上选择了一个创建线程的函数名称。）下面，考察函数dot_product的每一次执行，它的第一次执行要访问两个向量元素并等待它们。同时，函数的第二次执行能在下一个周期内访问另两个向量元素，依此类推。经过l时间单元后（l指内存系统的延迟），第一次执行从内存中得到了所需的数据，并能执行需要的计算。在下一个周期，第二次执行所需的数据到达，等等。据此可知，在每个时钟周期都能进行计算。 ■

例2.7的执行调度是根据两个假设判定的：内存系统能够为多个已提出的请求服务，并且处理器能够在每个周期内切换线程。此外，它还要求程序能够以线程的形式显式指定并发。 22
多线程处理器能够维持多个计算线程的上下文以及已提出的多个请求（内存访问、输入/输出或通信请求等），并在请求满足时执行它们。像HEP以及Tera这样依赖多线程的计算机，它们能够在每一个周期切换执行的上下文。因此它们能有效地躲避延迟，前提是并发（线程）足够多，使处理器不会空闲。并发与延迟之间的平衡在本书的其他许多章还会讲到。

2. 用预取躲避延迟

在典型程序中，数据项在很小的时间段被载入并由处理器使用。如果载入导致高速缓存不命中，那么处理器就不能使用数据。解决它的一个简单方法是：将载入操作提前，即使高速缓存没有命中，数据多半也能在用到的时候到达。然而，如果数据项在载入和使用之间的时间间隔内被覆盖，就必须重新发出载入指令。注意这种情况不比没有提前载入的情形更差。对此技巧的仔细研究表明，预取与多线程的工作目的相似。在将载入提前时，必须将没有资源相关性（即使用相同的寄存器）的独立执行的线程与其他的线程区别开来。许多编译器都竭力将载入提前，以屏蔽内存延迟。

例2.8 通过预取躲避延迟

考虑用一个`for`循环将向量`a`与`b`相加。在循环第一次迭代时，处理器需要`a[0]`与`b[0]`。由于这两个数不在高速缓存中，处理器必须承受内存延迟。请求满足后，处理器也需要`a[1]`与`b[1]`。假定每个请求都只占用一个周期（1纳秒），内存请求在100纳秒后得到满足，那么在100个这样的请求之后内存系统就能返回第一批数据项。其后，每个周期都会返回一对向量元素。这样，在接下来的每个周期里都能进行一次加法运算，处理器周期也不会浪费了。 ■

2.2.4 多线程与预取间的权衡

虽然看上去所有与内存系统性能相关的问题都可以通过多线程与预取来解决，这两种方法还是会受到内存带宽的很大影响。

例2.9 带宽对多线程程序的影响

假设某一计算机，具有1 GHz时钟频率的处理器，4个字的高速缓存行，访问高速缓存需要一个周期，DRAM的延迟为100纳秒。该计算机进行计算时访问1 KB高速缓存的命中率为 23
25%，访问32 KB的命中率为90%。考虑两种情况：第一，单一线程执行，整个高速缓存都可用来存放串行的上下文；第二，32线程执行，每个线程有1 KB的高速缓存驻留。如果计算在每1纳秒的周期都有一个数据请求，则第一种情况下，对DRAM的带宽需求为每10纳秒1个字，因为其他的字都从高速缓存中取得（90% 的高速缓存命中率）。这对应于400 MB/s带宽。第二种情况下，对DRAM的带宽需求提高到每个线程的每4个周期3个字（25%的高速缓存命中率）。如果所有的线程都呈现相同的高速缓存行为，则这种情况对应于0.75 字/纳秒，或3 GB/s。 ■

例2.9说明一个重要的问题，即由于每个线程中具有较小的高速缓存驻留，多线程系统的带宽需求显著增加。在这个例子中，维持在400 MB/s 的DRAM带宽是合理的，而3 GB/s 则是当前大多数系统能够提供的。从这个意义上说，多线程系统成为带宽的限制而不是延迟的限

制。应该认识到，多线程和预取只是用来解决延迟问题，而它们通常会使带宽问题加剧。

另一个问题与有效使用多线程和预取所需的额外硬件资源有关。考虑一种对寄存器进行10次提前载入的情况。这样的载入需要10个自由的寄存器。如果某一干扰指令覆盖寄存器，就要重新载入数据。同不用预取的情况相比，这不会增加任何更多的取数延迟。但是，我们对同一数据项取两次，对内存系统的带宽需求就会增加两倍。这种情况与例2.9所示的由于高速缓存限制所造成的情形相似。用更大的寄存器文件与高速缓存来实现多线程和预取能减少这种情况的出现。

2.3 并行计算平台剖析

上面几节讨论了影响串行或隐式并行程序性能的许多因素。当前微处理器的峰值性能与持续性能之间越来越大的差距、内存系统性能的影响以及许多问题的分布式特性，成为突出的并行化的推动因素。下面开始在更高层次上介绍一些并行计算平台的元素，它们对面向性能的以及可移植的并行编程来说都非常重要。为促进对并行平台性能的讨论，我们首先对并
24 行平台的物理及逻辑组织作一个剖析。逻辑组织是指程序员眼中的平台，而物理组织指的则是平台的实际硬件组织。在程序员眼里，表达并行任务的方法以及指定任务间相互作用的机制，是并行计算的两个重要组成部分。前者有时称为控制结构，而后者称为通信模型。

2.3.1 并行平台的控制结构

并行任务可以用不同层次的粒度来指定。在一个极端情况下，一系列程序中的每个程序都可看作一个并行任务。在另一个极端，程序中一些单独的指令也可以看作一个并行任务。在这两个极端之间，存在着一系列用来指定程序控制结构的模型，以及支持这些模型相应的体系结构。

例2.10 多处理器上单指令的并行性

考察下面两个向量相加的代码段：

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

在这个例子中，循环中的每次迭代都相互独立，即c[0]=a[0]+b[0]; c[1]=a[1]+b[1]; 等等，都可能相互独立地执行。因此，如果存在执行同样指令的机制，在本例中是在所有的处理器上将相应的数据相加，就能更快地执行循环。 ■

并行计算机的处理器可以在单一控制部件的集中控制下运行，也可以独立运行。在称为*单指令流多数据流*（single instruction stream, multiple data stream, SIMD）的结构中，单一控制部件向每个处理部件分派指令。图2-3 a说明一个典型的SIMD体系结构。在SIMD并行计算机中，同样的指令被所有的处理部件同时执行。在例2.10中，add指令被分派到所有处理器中，并由这些处理器并发执行。一些最早的并行计算机，如 Illiac IV、MPP、DAP、CM-2以及MasPar MP-1都属于这种类型。最近，这种类型的一些变体用到了协处理器部件中，如Intel处理器的MMX部件以及 Sharc 中的DSP芯片等。带有SSE（流SIMD 扩展）的Intel 奔腾处理器提供了多条指令，能够对多个数据项执行同样的指令。这些体系结构增强依赖于一些基本计算（如图像、图形处理）的高度结构化性质，以获得改进的性能。

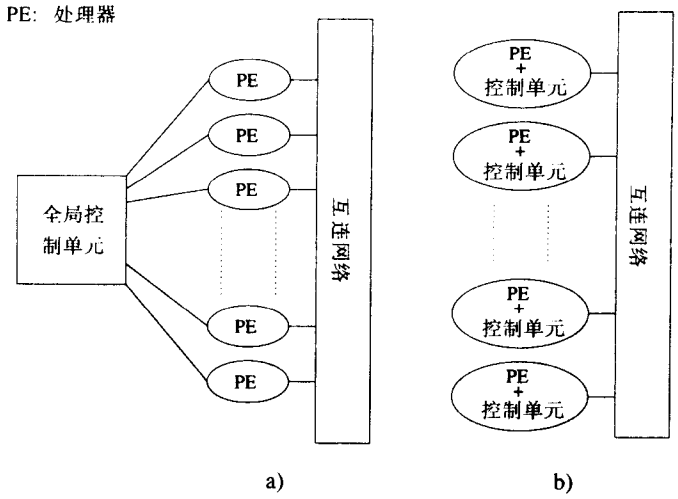


图2-3 a) 典型的SIMD体系结构 b) 典型的MIMD体系结构

虽然SIMD很适合用来对数组这样的并行数据结构进行结构化计算，但经常需要有选择地关闭对某些数据项的操作。因此，绝大多数SIMD编程模式允许“操作屏蔽码”，这是一种与每个数据项相关的二进制屏蔽码，用于指定数据项是否参与运算。一些基本的条件语句，如 25
`where (condition) then <stmnt> <elsewhere stmnt>`，用于支持选择执行。条件执行会降低SIMD处理器的性能，使用时一定要小心。

与SIMD体系结构形成对照，如果计算机中的每个处理器都能够独立于其他处理器执行不同的程序，就称为*多指令流多数据流*（multiple instruction stream, multiple data stream, MIMD）计算机。图2-3 b 描绘一个典型的MIMD计算机。该模型的一种简单的变形，称为*单程序多数据*（single program multiple data,SPMD）模型，用同一程序的多个实例在不同数据上执行。容易看出SPMD模型与MIMD模型具有同样的表现形式，因为多个程序的每一个都能用任务标识符指定的条件插入到一个大的if-else块中。许多并行平台都使用SPMD模型，该模型需要的体系结构支持也最小。这样的平台包括Sun Ultra Server、多处理器PC、工作站机群以及IBM SP等。

SIMD计算机比MIMD计算机需要的硬件更少，因为SIMD计算机只有一个全局控制部件。而且，因为SIMD计算机只需存储程序的一个副本，它需要更少的内存。对比之下，MIMD计算机要在每个处理器上存储程序及安装操作系统。然而，用SIMD处理器作为通用的计算引擎却没有流行，这是由SIMD专门的硬件体系结构、经济因素、设计限制、产品的生命期以及应用程序的特点所决定的。相反，支持SPMD模式的平台却可以用现成的元件以及相对少的投入很快制造出来。SIMD计算机的设计需要投入大量的人力物力，导致其开发周期很长。由于串 26
行处理器发展很快，SIMD计算机面临很快过时的危险。SIMD体系结构也不太适合许多具有不规则特性的应用程序。下面的例2.11说明，在条件执行时，SIMD体系结构将造成很低的资源利用率。

例2.11 在SIMD体系结构上执行条件语句

考察图2-4 所示的条件语句执行。图2-4 a 中的条件语句分两步执行。在第一步，所有$B = 0$的处理器执行指令$C = A$，所有其他的处理器空闲。在第二步，指令的“else”部分

($C = A/B$) 被执行，第一步时激活的处理器此时变成空闲的。这说明SIMD体系结构的一个缺点。

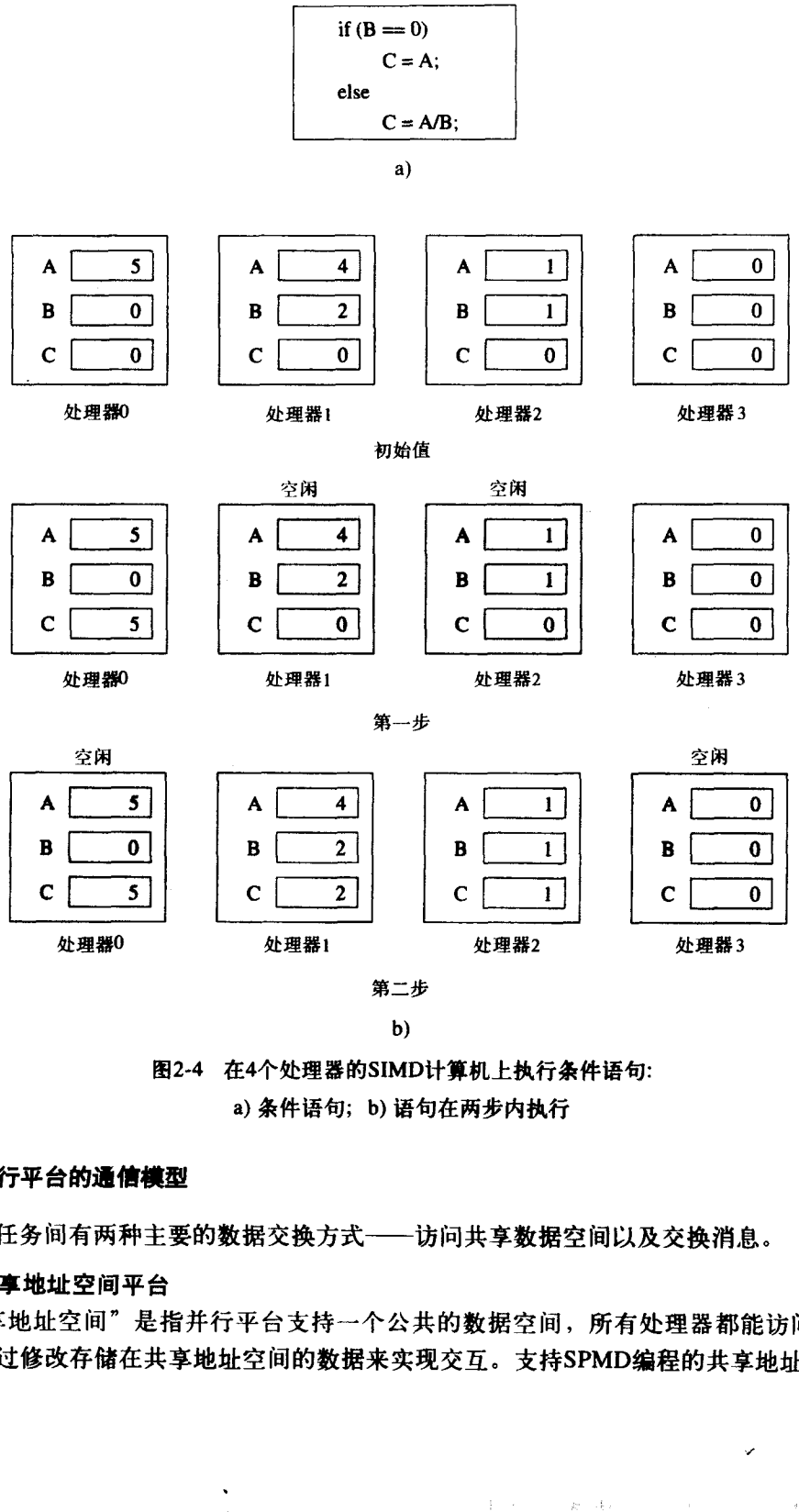


图2-4 在4个处理器的SIMD计算机上执行条件语句：
a) 条件语句；b) 语句在两步内执行

2.3.2 并行平台的通信模型

并行任务间有两种主要的数据交换方式——访问共享数据空间以及交换消息。

1. 共享地址空间平台

“共享地址空间”是指并行平台支持一个公共的数据空间，所有处理器都能访问该空间。处理器通过修改存储在共享地址空间的数据来实现交互。支持SPMD编程的共享地址空间平台

也称为多处理器（multiprocessor）。共享地址空间的内存可以是本地的（处理器独占），也可以是全局的（对所有处理器公用）。如果处理器访问系统中任何内存字（全局或本地）的时间都相同，平台就归类为一致内存访问（uniform memory access, UMA）多计算机。另一方面，如果访问某些内存字的时间长于其他内存字的访问时间，平台就称为非一致内存访问（non-uniform memory access, NUMA）多计算机。图2-5 a 和2-5 b 展示UMA平台，而图2-5 c 展示一种NUMA平台。图2-5 b 中的一个例子非常有趣。在这个例子中，访问高速缓存中的内存字要比访问内存地址快，但它仍然分类为UMA体系结构，因为目前所有的微处理器都有高速缓存的层次结构。因此，如果考虑高速缓存访问时间，即使单处理器也不会归类到UMA。基于这个原因，NUMA与UMA体系结构的定义根据的只是内存访问时间而不是高速缓存访问时间。像SGI Origin 2000服务器Sun Ultra HPC服务器这样的计算机属于NUMA多处理器这一类。区别UMA和NUMA平台很重要。如果访问本地内存比访问全局内存便宜，算法就必须构造本地性、结构化数据以及相应的计算等。

全局内存空间的存在使得在这样的平台上编程要简单得多。所有只读的操作对程序员来说是不可见的，因为编写这些代码与编写串行程序没有什么区别。这就大大减轻了写并行程序的负担。但是，读/写操作比只读操作更难编程，因为这种操作的并发访问是互斥的。所以，像线程（POSIX，NT）或指令（OpenMP）这样的共享地址空间编程模式采用锁及其他相关的机制来支持同步。

27
~
28

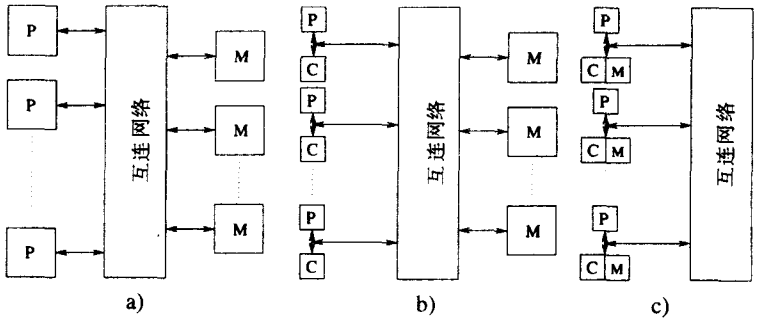


图2-5 典型的共享地址空间结构：a) 一致内存访问共享地址空间计算机；
b) 带有高速缓存及内存的一致内存访问共享地址空间计算机；
c) 只带本地内存的非一致内存访问共享地址空间计算机

处理器上高速缓存的出现也提出了同时由两个或更多处理器操作的单个内存字的多副本问题。以这种情况支持共享地址空间涉及两个重要任务：提供在系统中查找内存字地址的转换机制，以及保证对同一内存字的多个副本的并发操作有很好定义的语义。后者也称为高速缓存一致性（cache coherence）机制。这种机制以及它的应用在2.4.6节有更详细的介绍。支持高速缓存一致性需要相应的硬件支持。因此，一些共享地址空间计算机只支持单个地址转换机制，把保证一致性的任务留给程序员。这种平台的编程包含一些如get和put这样的基本语句。这些基本语句允许处理器获取（或放置）存储在远程处理器上的变量。然而，如果变量的副本之一改变了，其他的副本不能自动地更新或失效。

共享地址空间与共享内存计算机是两个常见的，但又常被误解的术语，理解它们的区别很重要。共享内存计算机这个术语从历史上说，指的是内存在物理上被多个处理器共享的一

种体系结构，即每个处理器对任意的内存段有同等的访问权。它和上面讲到的UMA模型相同。分布式内存计算机正好与之相反，不同的内存段物理上与不同的处理器对应。共享内存或分布式内存计算机的剖析属于计算机的物理组织范畴，在2.4节作更详细讨论。无论是共享内存
29 还是分布式内存的物理模型，都提供了不相交或共享地址空间平台的逻辑视图。分布式内存共享地址空间计算机与NUMA计算机相同。

2. 消息传递平台

从计算机逻辑观点上说，消息传递平台由p个处理节点构成，每个节点都有自己的独立地址空间。每个处理节点可以是单一处理器，也可以是共享地址空间的多处理器，而后者在现代消息传递并行计算机中正呈现快速增长的趋势。很自然，这样观点的实例来自工作站机群和非共享地址空间多计算机。在这样的平台上，运行在不同节点上的进程之间的交互必须用消息来完成，这也是*消息传递*（message passing）这个名称的由来。这种消息交换用来传送数据、操作以及使多个进程间的行为同步。按这种最常见的形式，消息传递模式支持在p个节点的每一个上执行一个不同的程序。

由于交互是通过发送及接收消息完成的，这种编程模式的基本操作为`send`和`receive`（相应的调用在不同的API之间可能会不同，但语义大体相同）。另外，由于发送和接收操作必须指定目标地址，必须有一个机制，来分配唯一的ID给每个执行并行程序的进程。这种ID通常通过像`whoami`之类的函数提供给程序，该函数对调用进程返回ID。通常还有一个函数用来完成消息传递的基本操作——`numprocs`，它给出参与操作的进程总数。有了这4种基本的操作，就能写出任意的消息传递程序。不同的消息传递API，例如消息传递接口（MPI）及并行虚拟机（PVM），支持这些基本操作和许多用不同函数名的更高层次的功能。支持消息传递模式的并行平台包括IBM SP、SGI Origin 2000以及工作站机群。

在有p个节点的共享地址空间计算机上，很容易模拟含有同样节点个数的消息传递体系结构。假设是单一处理器节点，通过将共享地址空间分割成p个不相交的分区，并独占地向每个处理器分配一个这样的分区，就能实现这种模拟。然后，处理器就能通过向其他处理器的分区写入信息或者读出信息来send或receive消息，当完成读出或写入操作后，再用适当的同步原语告知它的通信对方。然而，在消息传递计算机上模拟共享地址空间体系结构代价很高，
30 因为访问其他节点的内存需要发送和接收信息。

2.4 并行平台的物理组织

本节将研究并行计算机的物理体系结构。首先从理想的体系结构出发，概述与实现这种模型相关的实际困难，并讨论一些常见的体系结构。

2.4.1 理想并行计算机的体系结构

串行计算模型（随机访问计算机或RAM）的一种自然扩展包含p个处理器以及大小不受限制的全局内存，所有处理器同样可以访问该内存。所有处理器访问同样的地址空间，共享一个公用的时钟，但在每个周期内可以执行不同的指令。这种理想的模型也称为*并行随机访问计算机*（parallel random access machine, PRAM）。由于PRAM允许对不同内存单元进行并发访问，依据如何处理对内存的同时访问，PRAM可分为四小类:

1）*互斥读互斥写*（EREW）PRAM。这一类的PRAM独占访问内存单元，不允许并发的读写操作。这是一种最弱的PRAM模型，对内存访问提供最小的并发性。

2）*并发读互斥写*（CREW）PRAM。这种类型允许对内存单元多读，但对内存位置多写是串行的。

3）*互斥读并发写*（ERCW）PRAM。对内存单元允许多写访问，但多读访问是串行的。

4）*并发读并发写*（CRCW）PRAM。对内存单元允许多读多写。这是最强大的PRAM模型。

允许并发读访问不会造成程序中语义上的不一致。然而，对内存单元并发写却需要进行仲裁。有几种协议用来解决并发写的问题。最常用的几种协议如下:

- *共有*（common），如果处理器试图写的所有值都相同，则允许并发写。
- *任意*（arbitrary），任一处理器可以进行写操作，而其余的处理器则不行。
- *优先级*（priority），处理器按预定义的优先级列表排列，最高优先级的处理器可以进行写操作而其他的处理器不能。
- *求和*（sum），所有量的总和被写入（基于求和的写冲突解决模型能够扩展到任意由待写入量定义的相关操作符上）。

理想模型的体系结构复杂性

考虑将EREW PRAM作为具有p个处理器及m个字的全局内存的共享内存计算机来实现。这些处理器通过一系列开关连接到内存上。这些开关决定了由每个处理器访问的内存字。在EREW PRAM中，如果某一内存字没有同时被多个处理器访问，则p个处理器中的任何一个都可以访问任何内存字。为保证这样的连通性，开关总数必须是$\Theta(mp)$个。（附录中有关于Θ的解释。）如果内存大小可观，构造这样复杂度的开关网络就会花很高的代价。因此，PRAM计算模型在现实中不可能实现的。

2.4.2 并行计算机互连网络

互连网络提供多个处理节点之间或处理器与内存模块之间的数据传输的机制。互连网络的黑盒视图由n个输入及m个输出构成。输出个数不一定和输入个数相同。通常互连网络由链路和开关构成。链路指的是像导线或光纤这样能够携带信息的物理介质。许多因素会影响链路的特性。由于链路基于导电介质，导线之间的电容耦合限制了信号传播的速度。电容耦合和信号强度衰减是链路长度的函数。

互连网络分为*静态*（static）和*动态*（dynamic）两种。静态网络由处理节点间的点对点通信链路构成，也称为*直接*网络。动态网络由开关及通信链路构成，由开关在处理节点和存储区之间建立通道来动态实现通信链路间的互连。动态网络也称为*非直接*（indirect）网络。图2-6 a 列出一个简单的含4个处理器或节点的静态网络，每个处理节点通过一个网络接口在格网结构中与另两个节点相连。图2-6 b 列出一个含4个节点的动态网络，其节点通过开关网络与其他节点相连。

互接网络中的开关由一组输入及输出端口构成。开关能提供一系列的功能。开关的最小功能是提供从输入到输出端口间的映射。一个开关中端口的总数也称为该开关的*度*（degree）。开关还能提供对内部缓冲（当请求的输出端口忙时）、路由选择（减少网络拥塞）以及多点传

31

播（对多个端口的同样输出）等的支持。从输入端口到输出端口的映射可以通过多种机制获
32 得，这些机制基于物理交叉开关、多端口内存、多路转接器-多路分解器以及多路复用总线等。开关的成本受映射硬件成本、外围设备硬件以及包装成本的影响。通常，映射硬件按开关的度的平方增长，外围硬件与度线性相关，而包装成本与管脚的个数线性相关。

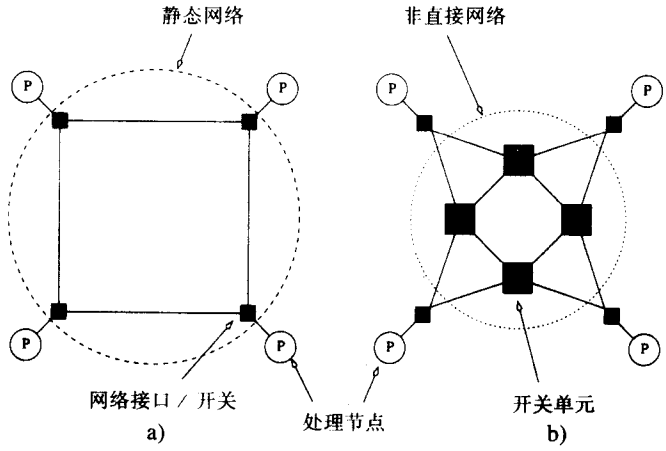


图2-6 互连网络的分类：a）静态网络；b）动态网络

节点与网络间的连通性由网络接口提供。网络接口的输入与输出端口使数据向网络流入和流出。通常，网络接口还有如下功能：数据封装，计算路由选择信息，对流入及流出的数据进行缓冲以便网络速度和处理器相匹配，以及错误检测等。处理器与网络间的接口的位置也很重要。普通网络接口挂起输入/输出总线，而高度耦合的并行计算机挂起内存总线。由于输入/输出总线通常比内存总线慢，后者能提供更高的带宽。

2.4.3 网络拓扑结构

互连网络中使用了各种各样的拓扑结构。这些结构用来实现成本及可扩展性与性能间的平衡。单纯的拓扑结构具有很有趣的数学特性，而在实际的互连网络中用到的拓扑是纯粹拓扑结构的组合或修改。

1. 基于总线的网络

基于总线的网络可能是最简单的网络，它包含一个所有节点公用的共享介质。总线具有所需的特性，即总线的成本与节点数目p是线性关系。通常这种成本与总线接口有关。此外，
33 网络中任意两个节点间的距离是常数（$O(1)$）。总线对于在节点间广播信息也是理想的。由于传输介质是共享的，与点对点信息传输相比，广播占用很小的系统开销。然而，当节点数目增多时，总线上有限的带宽限制了网络的整体性能。通常基于总线的计算机的节点数目限制在数十个以内。Sun Enterprise 服务器以及基于Intel 奔腾处理器的共享总线多处理器就是采用这种体系结构的例子。

对于一般的程序来说，大多数访问的数据对节点是本地的，利用这个性质可以降低对总线带宽的需求。对于这样的程序，可以为每个节点提供高速缓存。私有数据存放在节点的高速缓存中，只有远程数据才通过总线存取。

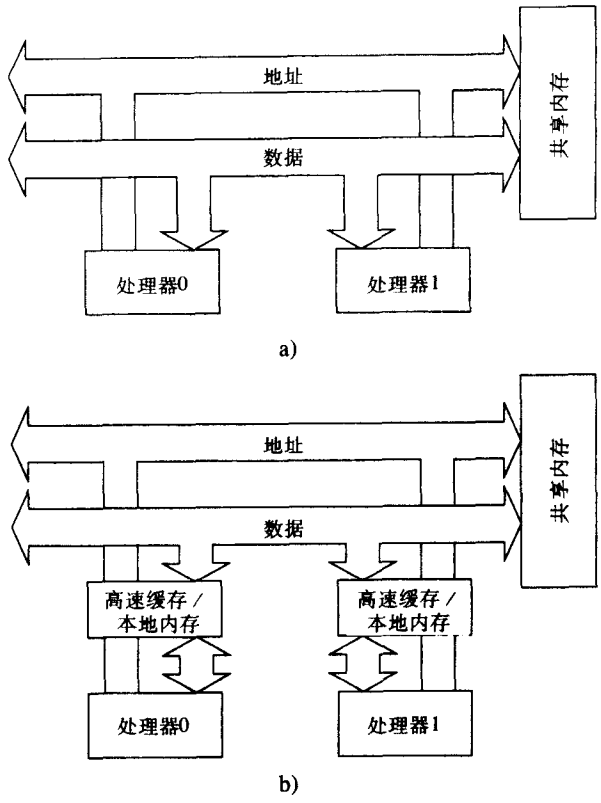


图2-7 基于总线的互连：a）没有本地高速缓存；
b）具有本地内存/高速缓存

例2.12 用高速缓存减少共享总线的带宽

在图2-7a中，p个处理器共享到内存的总线。如果每个处理器访问k个数据项，每次数据访问需要t_{cycle}时间，那么执行时间的下界为$t_{cycle} \times kp$秒。下面考察图2-7b的硬件结构。假设50%的内存访问（$0.5k$）为本地数据。这些本地数据位于处理器的私有内存中。再假设访问本地内存的时间与访问全局内存的时间相同，即是t_{cycle}。在这种情况下，总执行时间的下界为 $0.5 \times t_{cycle} \times k + 0.5 \times t_{cycle} \times kp$。这里，第一项是访问本地数据的时间，第二项是访问共享数据的时间。很容易看出，当p变大时，按图2-7b的结构，访问时间的下界逼近$0.5 \times t_{cycle} \times kp$。与图2-7a相比，执行时间下界提高50%。 ■ 34

在实际应用中，共享数据与私有数据是以更复杂的方式处理的。在2.4.6节将与高速缓存一致性一起简要讨论这个问题。

2. 交叉开关网络

如图2-8所示，用排成网格形的一组开关或开关节点，可以很简单地将p个处理器与b个存储区连接。交叉开关网络是一种非拥塞网络，连接一个处理节点与存储区不会拥塞其他任何处理节点与存储区之间的连接。

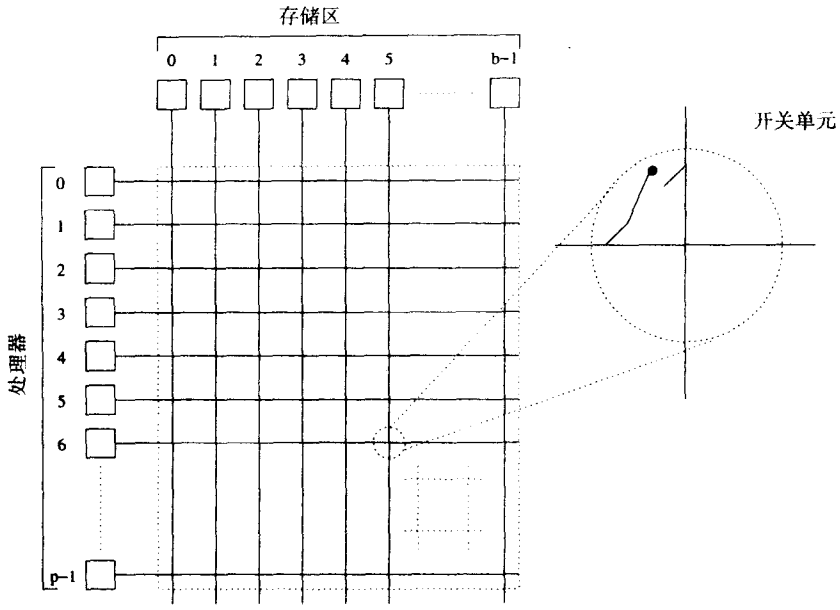


图2-8 连接p个处理器和b个存储区的完全非拥塞交叉开关网络

要实现这样的网络，需要的开关节点总数为$\Theta(pb)$。可以做出这样合理的假设：存储区b
35 的数目至少为p；否则，在任意时刻，就会有一些处理节点不能访问任何存储区。因此，当p增大时，开关网络的复杂性（组件数目）以$\Omega(p^2)$增大（附录中有关于Ω的解释）。当处理节点很多时，开关中的元件数目很多，很难实现高的数据传输速度。因此，从成本的角度上讲，交叉开关网络的可扩展性不好。

3. 多级网络

交叉开关网络就性能而言是可扩展的，但就成本而言却又不是可扩展的。相反，共享总线网络在成本上可扩展但在性能上不可扩展。在此两种极端之间，有一种折中的网络，称为*多级互连网络*（multistage interconnection network）。它比总线网络在性能方面可扩展性强，又比交叉开关网络在成本上可扩展性强。

图2-9为包含p个处理节点和b个存储区的多级网络示意图。omega网络是一种常用的多级互连网络，该网络的级数为$\log p$，p是输入（处理节点）的数目，也是输出（存储区）的数目。omega网络的每一级都含有一种互连模式，连接p个输入和p个输出。如果下式成立，则在输入i和输出j之间存在一条链路：

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

公式(2-1)代表从二进制表示的i获得j的左旋操作。这种互连模式也称为完全混洗
36 (perfect shuffle)。图2-10是含有8个输入和输出的完全混洗互连模式示意图。在omega网络的每一级，完全混洗互连模式送入$p/2$个开关或开关节点中。每个开关的连接方式有两种。一种是输入直接传送到输出处，如图2-11a所示。这种方式称为*直通式*（pass-through）连接。另一种方式是跨接开关节点输入，然后再送出，如图2-11b所示。这种方式称为*跨接*（cross-

over）连接。

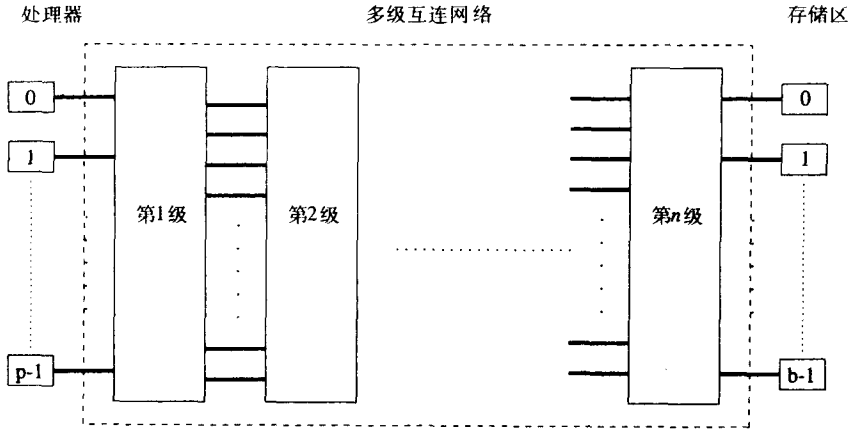


图2-9 一种典型的多级互连网络示意图

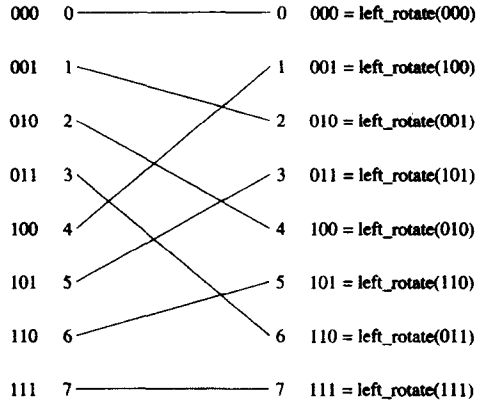


图2-10 含8个输入和输出的完全混洗互连

omega网络含$p/2 \times \log p$个开关节点，网络成本随$\Theta(p \log p)$增加。注意这个成本低于完全交叉开关网络的$\Theta(p^2)$。图2-12所示为含8个处理器（以左边的二进制数表示）和8个存储区（以右边的二进制数表示）的omega网络。omega网络中的数据选路用一种简单的方案来完成。设s为需要写数据到存储区t的处理器的二进制表示。数据穿过链路到达第一个开关节点。如果s和t的最高有效位相同，那么开关就会按直通式方式进行数据选路；如果最高有效位不同，则以跨接方式进行数据选路。在下一个开关级，再使用下一个最高有效位来重复同样的方案。穿越$\log p$级开关要使用s和t的二进制表示中的所有$\log p$个位。



a）直通式　　b）跨接式

图2-11 2×2开关的两种开关结构

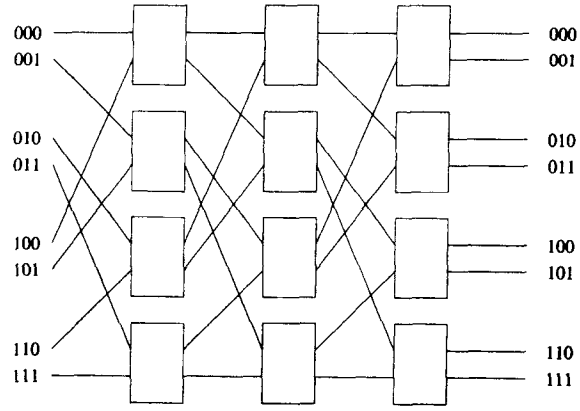
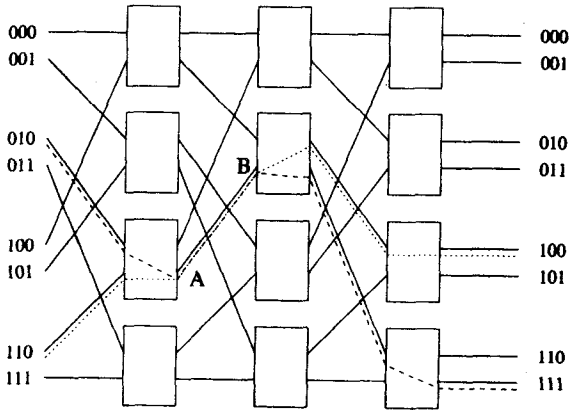


图2-12 连接8个输入和输出的完全omega网络

图2-13中显示omega 网络中，从2号处理器（010）到7号存储区（111），以及从6号处理器（110）到4号存储区（100）的数据路由选择。当2号处理器（010）和7号存储区（111）通信时，阻塞6号处理器（110）对4号存储区（100）的通道。通信链路AB由两个通信通道使用。因此，在omega 网络中，某一处理器对某一存储区的访问可能不允许其他处理器对其他存储区的访问。具有这种性质的网络称为阻塞网络（blocking network）。



38 图2-13 omega 网络中的阻塞实例：一个信息（010到111或110到100）在链路AB处阻塞

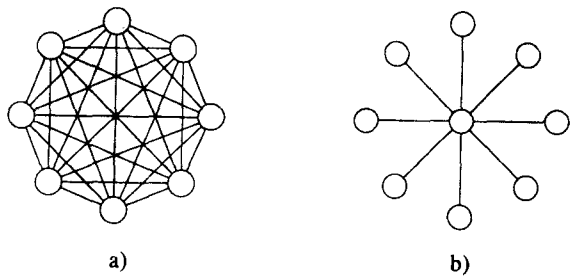


图2-14 a）8个节点的全连接网络；b）9个节点的星形连接网络

4. 全连接网络

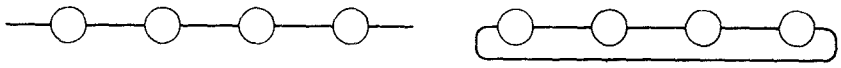
在全连接网络（completely-connected network）中，每个节点到其他所有节点都存在直接通信链路。图2-14a便是有8个节点的全连接网络。在这种网络中，任两个节点间通信链路的存在使得每个节点只需一步就能向另一个节点传送信息，在这个意义下这是理想的网络。全连接网络是交叉开关网络的静态对应网络，因为在这两种网络中，任意输入/输出对之间的通信都不会阻塞其他任何对之间的通信。

5. 星形连接网络

在星形连接网络（star-connected network）中，某一处理器作为中央处理器。其他的每个处理器有一条通信链路与中央处理器相连。图2-14b便是有9个节点的星形连接网络。星形连接网络与基于总线的网络类似。任一对处理器间的通信通过中央处理器选择路由，类似在基于总线的网络中共享总线构成所有通信间的媒介一样。中央处理器是星形结构的瓶颈。

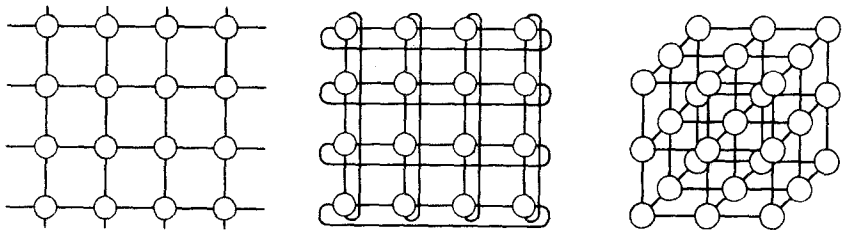
6. 线性阵列、格网和k-d格网

由于全连接网络中的链路太多，通常采用稀疏网络来构造并行计算机。线性阵列和超立方体体就属于这种稀疏网络。线性阵列是一种静态网络，它的每个节点（除了两个端节点）有两个邻居节点，一个在左边，一个在右边。环或1维环绕（图2-15b）是线性阵列（图2-15a）的简单扩展。环在线性阵列的两端间具有回绕连接。这种情况下每个节点有两个邻居。



a）无回绕链路　　b）有回绕链路

图2-15　线性阵列



a)　　b)　　c)

图2-16　2维与3维格网：a）无回绕2维格网；b）有回绕链路的2维格网(2维环绕)；c）无回绕3维格网

图2-16a所示的2维格网是线性阵列的2维的扩展。每个维有$\sqrt{p}$个节点，每个节点用二元组(i, j)标识。除了边界节点外，每个节点同上、下、左、右方向的4个节点相连。2维格网可以放在2维平面上，从布线的角度来看，这种网络颇具吸引力。此外，许多常规结构的计算很自然地与2维格网对应。因此，并行计算机通常采用2维格网式互连。如果用回绕链路将2维格网扩大，就得到如图2-16b所示的2维环绕。将2维格网推广到3维，就得到如图2-16c所示的3维立方体。除了边界上的节点外，3维立方体中的每个节点都和6个其他的节点相连，每维2个。许多在并行计算机上执行的物理仿真（如3维气象建模、结构建模等）都能自然地映射为

39

3维网络拓扑结构。因此，3维立方体通常用在并行计算机的互连网络中（如用于 Cray T3E 中）。

k-d格网指的是一种拓扑结构，它有d维，每一维上有k个节点。线性阵列构成k-d格网的一个极端，另一种称为超立方体的有趣拓扑结构构成另一极端。超立方体结构有log p维，在每一维上有两个节点。超立方体结构的构造如图2-17所示。0维超立方体有2^0个节点，也就是1个节点。将两个0维超立方体相连就得到了1维超立方体；4个节点的2维超立方体由两个1维超立方体连接相应节点构成。一般而言，d维超立方体由连接两个(d-1)维超立方体的相应节点
40 构成。图2-17展示16个节点的4维超立方体的构成。

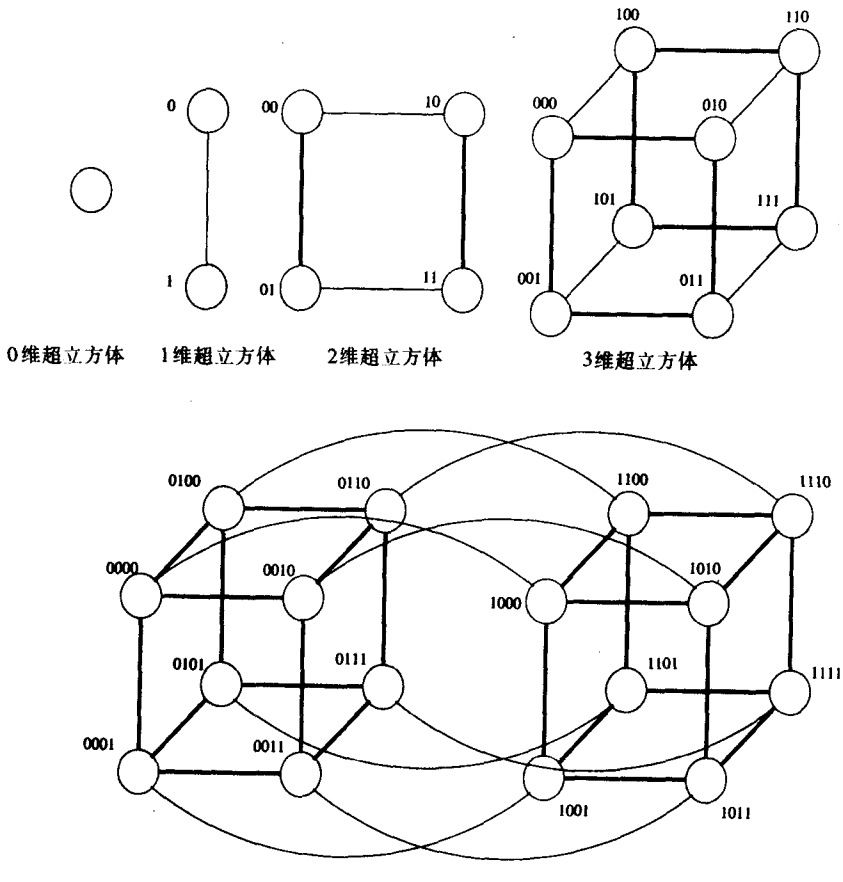


图2-17 从低维超立方体构建高维超立方体

从超立方体中导出节点的编号方式很有用。简单的编号方式可以从超立方体的构造得出。如图2-17所示，如果有两个包含p/2个节点的子立方体的编号，那么就可以通过给某个子立方体加一个“0”在前面，给另一个子立方体加一个“1”在前面，得到含p个节点的立方体的编号方式。这种编号方式具有一种有用的特性，即两个节点间的最小距离可以通过两个标号不同的位数得出。例如，标号0110的节点和标号0101的节点相隔两个链路，因为它们有两个位
41 不同。这种特性可用来从超立方体体系结构导出许多并行算法。

7. 基于树的网络

树网络（tree network）是指网络中任意一对节点间只存在一条通路的网络。线性阵列和星形连接网络都是树网络的特例。图2-18中所示为基于完全二叉树的网络。静态树网络在树的每个节点都有处理器（图2-18 a）。树网络也有动态的形态。在动态树网络中，中间层的节点为交换节点，叶子节点是处理器（图2-18 b）。

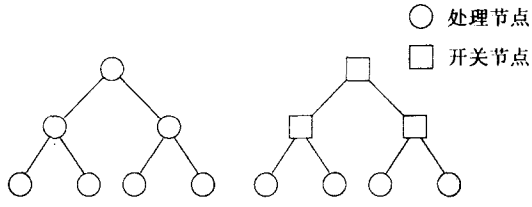


图2-18 完全二叉树网络

要实现在树中的信息选路，源节点必须沿树向上送信息，直到信息到达包含源节点和目的节点的最小子树的根节点，然后信息沿树向下选路，直到到达目的节点。

树网络在树的较高层中存在通信瓶颈。例如，当一个节点左边子树的许多节点与右边子树的节点通信时，根节点必须处理所有的信息。在动态网络中，可以通过增加通信链路，以及增加更接近根节点的交换节点的个数，来减少这种瓶颈。这种动态网络称为胖树（fat tree），如图2-19所示。

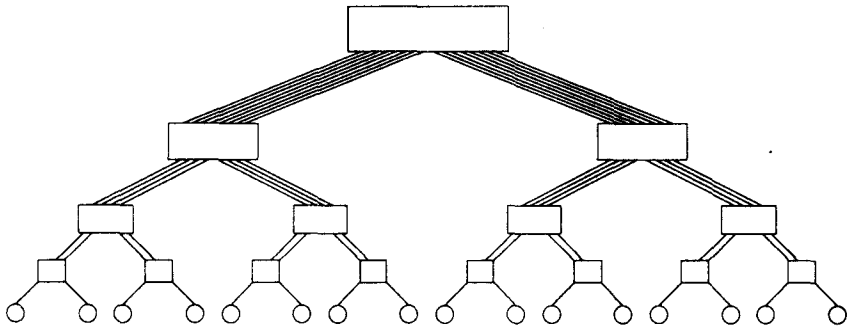


图2-19 含16个处理节点的胖树网络

42

2.4.4 静态互连网络评价

下面将讨论描述静态互连网络成本和性能的各种标准。我们用这些标准来评价上一小节讲到的静态网络。

直径 网络中任意两个处理节点之间的最长距离称为网络直径（diameter）。两个处理节点间的距离定义为它们间的最短路径（用链路数目表示）。全连接网络的直径是1；星形连接网络的直径是2；环形网络的直径是$\lfloor p/2 \rfloor$；无回绕连接的2维网络中，对角线相对角上的两个节点的直径为$2(\sqrt{p}-1)$；环绕格网的直径为$2\lfloor \sqrt{p}/2 \rfloor$；超立方体连接的网络由于两个节点的

标号至多在log p个位置可能会不同，其直径为log p；完全二叉树的直径为2 log((p+1)/2)，因为两个通信节点可能处于根节点的不同子树上，一条信息可能必须先上行到根节点，然后再下行到另一子树。

连通性 连通性（connectivity）是网络中任意两个节点间路径多重性的度量。网络的高连通性是希望达到的，因为这样才能减少通信资源的争用。连通性的一个度量是将一个网络分为两个不连通网络需要删去的最少弧数目。这也称为弧连通性（arc connectivity）。对于线性阵列、树以及星形网络，弧连通性是1。无回绕2维格网的弧连通性是2；2维环绕格网的连通性是4；d维超立方体的弧连通性是d。

对分宽度及对分带宽 网络的对分宽度（bisection width）是把网络分为两个相等网络时必须删去的最小通信链路的数目。环形网络的对分宽度是2，因为只要删去两个通信链路，就能把网络分为两个。同样，无回绕连接的含p个节点的2维格网的对分宽度为$\sqrt{p}$，2维回绕连接为$2\sqrt{p}$。树形连接和星形连接网络的对分宽度为1，而全连接网络为$p^2/4$。超立方体的对分宽度可以从它的构造导出。将两个(d−1)维的超立方体的相应链路连接，得到一个d维超立方体。由于两个子立方体含有$2^{(d-1)}$或$p/2$个节点，至少$p/2$个通信链路必须越过超立方体的任一划分才能把超立方体分割成两个子立方体（习题2.15）。

能够越过连接两个节点链路同时进行通信的位数称为通道宽度（channel width）。通道宽
43 度等于每个通信链路中物理线路的数目。单一物理线路能够传送位的峰值速度称为通道速度（channel rate）。两个通信链路端点之间能够传送数据的峰值速度称为通道带宽。通道带宽是通道速度和通道宽度的乘积。

表2-1 连接p个节点的静态网络特性一览表

网 络	直 径	对分宽度	弧连通性	成本（链路数目）
全连接	1	$p^2/4$	$p-1$	$p(p-1)/2$
星形	2	1	1	$p-1$
完全二叉树	$2\log((p+1)/2)$	1	1	$p-1$
线性阵列	$p-1$	1	1	$p-1$
无回绕2维格网	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2维环绕格网	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
超立方体	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
回绕k元d立方体	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

对分网络任何两半之间允许的最小通信量称为对分带宽，它是对分宽度和通道宽度的乘积。网络对分带宽有时也称为截面带宽（cross-section bandwidth）。

成本 评价网络的成本有多种标准，其中的标准之一是网络中所需的通信链路的数量或线路数量。线性阵列和树仅用$p-1$个链路连接p个节点。d维环绕格网有dp个链路，而超立方体连接网络有($p\log p$)/2个链路。

网络的对分带宽也可用作成本的评价标准，因为它提供2维封装中的面积下界或3维封装中的体积下界。如果网络的对分带宽为w，则2维封装中的面积下界为$\Theta(w^2)$，而3维封装中的体积下界为$\Theta(w^{3/2})$。根据这个标准，超立方体和全连接网络比其他网络的成本更高。

表2-1列出各种静态网络的不同特性，这些特性突出显示不同网络间的性能价格比权衡。

2.4.5 动态互连网络评价

动态网络的许多评价标准由相应的静态评价标准得出。由于信息经过开关时，开关必须付出开销，人们很自然地想到除本来的处理节点外，把每个开关看作网络中的节点。此时网络直径可以定义为网络中任意两个节点间的最大距离。这也表示信息在选择的一对节点间传送时所遇到的最大延迟。事实上，我们希望网络直径是任意两个处理节点间的最大距离；然而，对所有重要的网络而言，网络直径等同于任意一对（处理或开关）节点间的最大距离。 44

动态网络的连通性可以用节点或边的连通性来定义。节点连通性是指把网络分成两个部分必须删去的最小节点数目。像以前一样，这里只考虑开关节点（与考虑所有节点不同）。然而，考虑所有节点能给出动态网络路径多重性的很好的近似值，网络的弧连通性可以相似地定义为将网络分成两个不可及的部分必须删去的最小边数。

动态网络对分宽度的定义必须比直径和连通性的定义更准确。就对分宽度而言，我们要考虑所有可能的把p个处理节点分割成两个相等部分的方法。注意这里并没有限制分割开关节点。对每个这样的分割，选择分割开关节点的方法，使得穿过这个分割的边数最少。对任何这样的分割，边的最小数目是动态网络的对分宽度。对分宽度的另一直观定义是把网络分成具有相同处理节点数目的两半需要删去的最小边数。我们用下面的例子进一步阐述这个概念。

例2.13 动态网络的对分宽度

考察图2-20所示的网络。图中有3个对分A，B，C，每个对分都把网络分成两个含两个处理节点的部分。注意，每个划分不需要相等地划分网络中的节点。在本例中，每个划分都导致4条分割边之一。因此，该图的对分宽度为4。 ■

动态网络的成本与静态网络一样，由链路成本决定，同时也是开关成本。在常见的动态网络，开关的度为常数。因此，链路及开关的数目也差不多相同。而且，在常见的网络中，开关成本高于链路成本。因此，动态网络的成本通常由网络中的开关节点数目决定。

动态网络的各种性质总结于表2-2中。

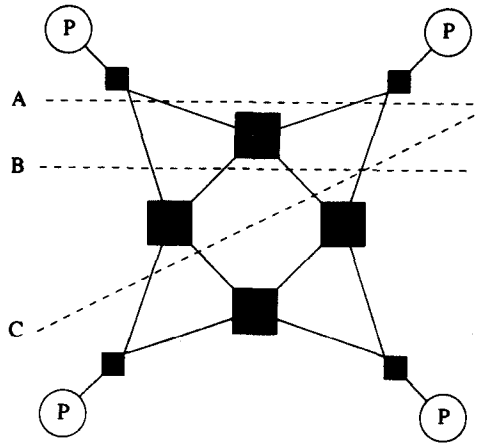


图2-20 动态网络的对分宽度计算方法：从处理节点的各个相等划分中选择越过划分的最少边数。在本例中，每个划分产生4条分割边之一。因此这个图的对分宽度为4

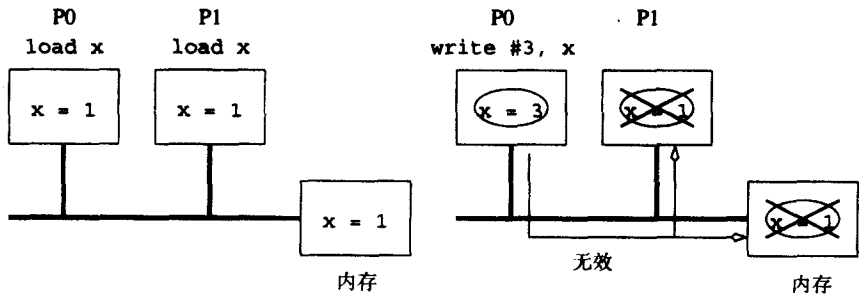
表2-2 连接p个处理节点的动态网络拓扑结构特性一览表

网 络	直 径	对分宽度	弧连通性	成本（链路数目）
交叉开关	1	p	1	p^2
omega网络	$\log p$	$p/2$	2	$p/2$
动态树	$2\log p$	1	2	$p-1$

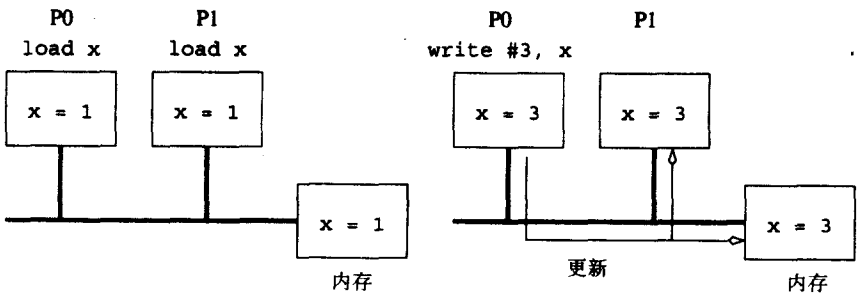
2.4.6 多处理器系统中的高速缓存一致性

45 互连网络提供基本的消息（数据）通信机制，而共享地址空间计算机需要其他的硬件才能使数据的多个副本保持一致。如果存在某一数据的两份副本（在不同的高速缓存/内存单元中），如何才能保证不同的处理器按预定义的语义对这两份数据进行处理？

保持高速缓存在多个处理器系统中的一致要比在单一处理器系统中复杂得多。这是因为除了和单处理器一样有多个副本外，修改这些副本的处理器可能会有多个。考虑如图2-21所示的简单情况。两个处理器P_0和P_1通过共享总线连接一个全局可存取的内存。两个处理器加载同样的变量。因此，就有变量的三个副本。现在，一致性机制必须保证所有对这些副本的操作是串行的（即是存在与并行调度对应的指令执行的串行顺序）。当处理器修改变量的一个
46 副本时必定发生两件事情之一：要么必须使其他的副本无效，要么必须使其他的副本更新。如果做不到这一点，其他的处理器就可能使用变量的错误值。这两个协议称为*无效*（invalidate）协议和*更新*（update）协议，如图2-21a和b所示。



a）无效协议



b）共享变量的更新协议

图2-21 多处理器系统的高速缓存一致性

在更新协议中，无论何时写入一个数据项，它在系统中的所有副本都被更新。因此，如果某处理器只读了一次数据项，就再也不使用它，那么，其他处理器对该数据项的更新将会造成多余的开销，如源延迟及网络带宽。另一方面，在这种情况下，无效协议在远程处理器上使第一次更新过的数据项无效，以后对该数据项的副本就不需要更新了。

另一个影响这些协议性能的重要因素是*假共享*（false sharing）。假共享是指不同的处理器更新相同高速缓存行中不同部分的情况。因此，虽然没有对共享变量进行更新，但系统并不进行检测。在无效协议中，当一个处理器更新它的高速缓存行它自己的一部分时，行中其他副本就成为无效的。当其他的处理器试图更新高速缓存行中它们自己的一部分时，就必须从那个远程处理器处获得行。容易看出，假共享会造成某一高速缓存行中的数据在不同的处理器之间像打乒乓球一样传来传去。在更新协议中，这种情况要稍微好一些，因为所有的读操作都可以本地进行，而写操作必须更新。这样就节省了一步使数据无效的操作。 47

无效和更新方案间的折中是典型的通信开销（更新）及空闲（无效时停止）间的折中。当前的高速缓存一致性计算机通常依靠无效协议。接下来的有关多处理器高速缓存系统的讨论都假设用无效协议。

用无效协议维持一致性 一个数据项的多个副本要保持一致，就必须留意这些副本的数量及各自的状态。这里要讨论的是与数据项相关联的一种可能的状态集合，以及在这些状态中能触发转换的事件。注意状态集合和转换都不是唯一的，可以定义其他的状态以及相关的转换。

下面再来看图2-21中的例子。开始时，变量x位于全局内存中。两个处理器第一步执行对变量加载的操作。在此时，变量的状态是*共享*的，因为它被多个处理器共享。当处理器P_0对变量执行存储操作时，就把该变量的所有副本标记为*无效*。它还必须把自己拥有的变量标记为已修改或脏（dirty）。这样做是为了保证接下来其他处理器对该变量的访问将转向P_0而不是内存。在此时，可以说，处理器P_1又一次执行加载x的操作。处理器P_1试图取得这个变量，因为变量已被处理器P_0标记为脏，处理器P_0对请求提供服务。在处理器P_1及全局内存中变量的副本都被更新，变量重新变为共享状态。因此，在这个简单模型中，高速缓存行经历三个状态：*共享*、*无效*以及脏。

图2-22是简单三状态协议的完全状态图。实线描述处理器操作，虚线描述一致性操作。例如，当处理器对无效块执行读操作时，块被取出，状态从无效转换为共享。同样，如果一个处理器对共享块执行写操作，一致性协议对块发出一个C_write（一致性写入）。这样将触发其他所有块的状态由共享转换为无效。

例2.14 用简单三状态协议维持一致性

考察如图2-23所示的由处理器P_0和P_1执行的两个程序段。系统除在处理器P_0及P_1包含本地内存（或高速缓存）外，还包含全局内存。假设本例中用到的三阶段协议与图2-22所示的状态图相同。系统中的高速缓存行的状态可以是共享的、无效的或脏的。每个数据项（变量）假设位于不同的高速缓存行中。开始时，变量x和y被标记为脏，这两个变量的唯一副本位于全局内存中。图2-23展示随着每条指令的执行，变量状态的转换以及变量的多个副本值的改变。 ■ 48

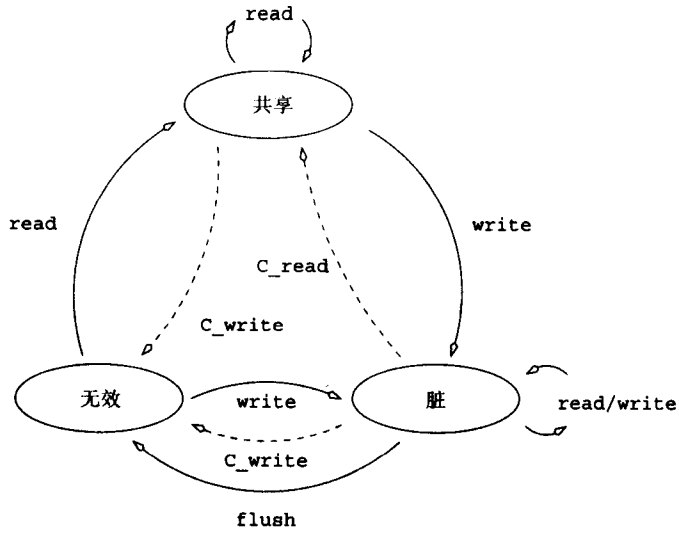


图2-22 简单三状态一致性协议状态图

时间	处理器0处的指令	处理器1处的指令	处理器0处的变量及其状态	处理器1处的变量及其状态	全局内存中的变量及其状态
					x = 5, D y = 12, D
	read x		x = 5, S		x = 5, S
		read y		y = 12, S	y = 12, S
	x = x + 1		x = 6, D		x = 5, I
		y = y + 1		y = 13, D	y = 12, I
	read y		y = 13, S	y = 13, S	y = 13, S
		read x	x = 6, S	x = 6, S	x = 6, S
	x = x + y		x = 19, D	x = 6, I	x = 6, I
		y = x + y	y = 13, I	y = 19, D	y = 13, I
	x = x + 1		x = 20, D		x = 6, I
		y = y + 1		y = 20, D	y = 13, I

图2-23 2.4.6节中讨论的用简单三状态一致性协议的并行程序执行实例

可以用各种硬件机制——侦听系统、基于目录的系统以及两者的结合来实现一致性协议。

1. 高速缓存侦听系统

侦听高速缓存通常与基于广播互连网络的多处理器系统相关，总线和环就属于这样的广播互连网络。在这样的系统中，所有处理器都侦听（监视）总线事务。这样处理器就能对它的高速缓存块做状态转换。图2-24所示为一典型的基于总线的侦听系统。每个处理器的高速缓存都有一组标记位，用来决定高速缓存块的状态。这些标记更新的依据是与一致性协议相

联系的状态图。比如，当侦听硬件检测到读指令向含有脏标记副本的高速缓存块发出时，它将控制总线，将脏数据消除。同样，当侦听硬件检测到写指令向含有副本的高速缓存块发出时，它会使该块无效。其他的状态转换也以这种方式在本地进行。

侦听高速缓存的性能 侦听协议已得到广泛的研究，并已在商用系统中得到应用。这主要是因为侦听系统比较简单，而且现有的基于总线的系统能够升级以适应侦听协议。侦听系统的性能增益表现在：当不同的处理器对不同的数据项操作时，这些数据项可以放在高速缓存中，当这些项被标记为脏时，随后的操作便可以对高速缓存本地执行，不会产生外部的数据传输。同样，如果某一数据项要被许多处理器读入，它就会在高速缓存中转换到共享状态，以后所有对它的读操作都在本地进行。在这两种情况下，一致性协议都不会增加任何开销。另一方面，如果多个处理器读取及更新同样的数据项，在多个处理器之间就会产生一致性动作。由于共享总线的带宽有限，在单位时间内只能执行这样一组数量不变的一致性操作。这就形成基于总线的侦听系统的一个主要瓶颈。

侦听协议与基于像总线广播网络这样的多计算机密切相关。这是因为所有的处理器都要侦听所有的消息。很显然，向所有的处理器广播一个处理器的所有内存操作并非可扩展的解决方案。解决此问题的一个明显方法是只将一致性操作传播给那些必须参与操作的处理器（即含有数据的相关副本的处理器）。这个方法要求查出哪些处理器中含有不同数据项的副本，以及这些数据项的相关状态信息。这种信息存储在一个目录中，而基于这种信息的一致性机制称为基于目录的系统。

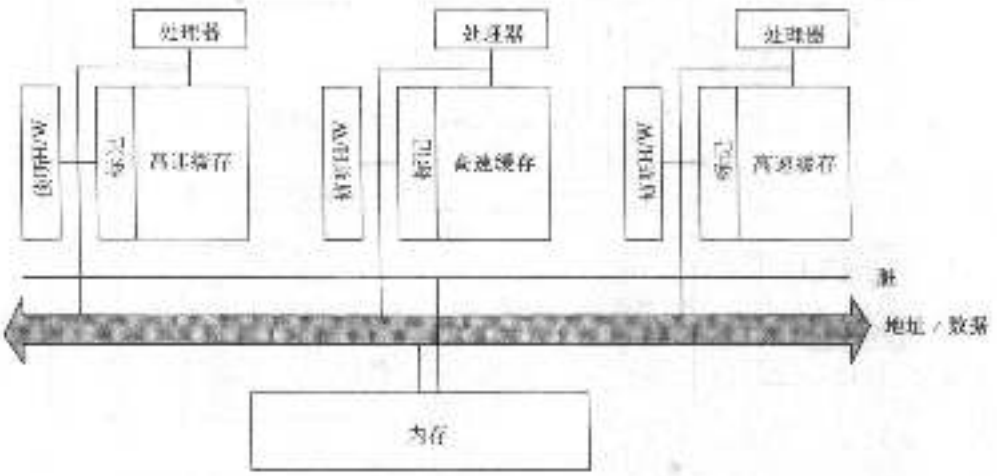


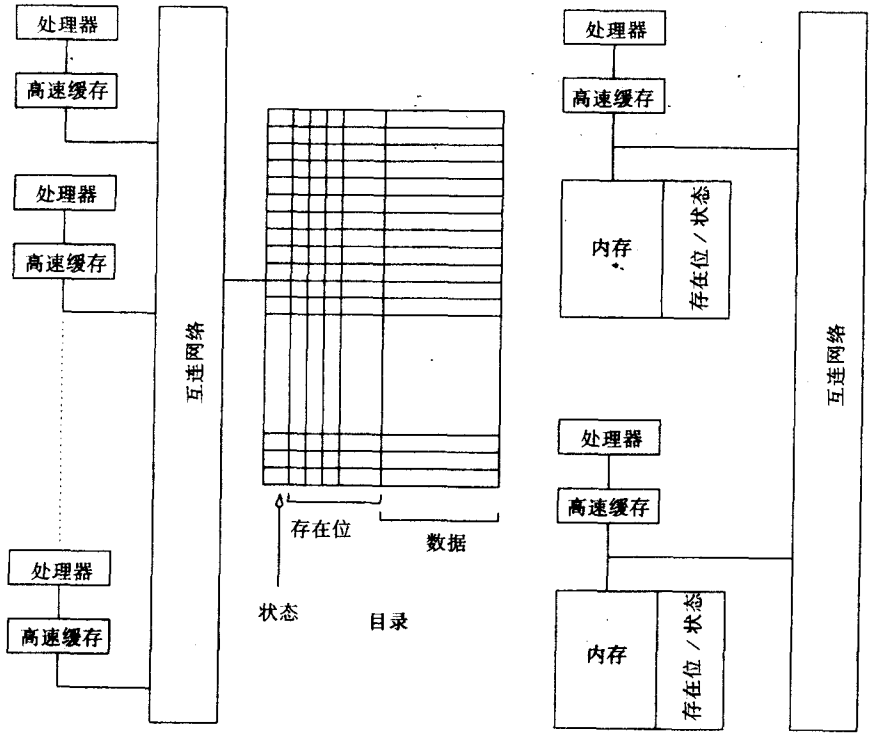
图2-24 一个简单的基于总线的侦听高速缓存一致性系统

2. 基于目录的系统

考察一个简单的系统，其全局内存增加一个目录，维护代表高速缓存块以及已缓存的处理器的位图（图2-25）。这些位图项有时称为存在位（presence bits）。与前面一样，假设三状态协议含有无效、脏以及共享等三种状态。基于目录的方案性能的关键，在于只有拥有某一特定块（或正在读取该块）的处理器，才会由于一致性操作参与状态转换。注意，其他的状态转换可能由处理器读出、写入或者清除（从高速缓存中清除一行）触发，但这些转换都可在本地处理，因为处理器操作反映在存在位中，状态反映在目录中。

再看一下图2-21所示的代码段。当处理器P_0和P_1访问与变量x相对应的块时，块的状态变为共享的，而存在位被更新，显示P_0和P_1共享该块。当P_0对变量执行了一次存储操作后，目录中的状态变为脏，P_1的存在位被复位。所有随后的P_0对该变量的操作都可以本地进行。如果另一个处理器读出该值，目录会注意到该脏标记，并用存在位将请求指向适当的处理器。处理器P_0更新内存中的块，并把块送到请求的处理器中。存在位被修改以反映这个变化和状态位转换为共享。

基于目录方案的性能 如像用侦听协议那样，如果不同的处理器对不同的数据块操作，这些块在各自的高速缓存中变为脏，后面所有的操作都可以本地进行。此外，如果多个处理器读出（但不更新）某一数据块，数据块在高速缓存中按共享状态被复制，以后的读出操作
51 就不会引起任何一致性开销。



a）集中式目录　　b）分布式目录

图2-25 基于目录系统的典型体系结构

当多个处理器试图更新同一数据项时，就会引发一致性操作。在这种情况下，除了必需的数据移动外，一致性操作增加两种系统开销：一种是传播状态更新（无效或更新），另一种是从目录产生状态信息。前者以通信开销的形式出现，后者增加资源争用。通信开销与要求状态更新的处理器数量及传播状态信息的算法有关。资源争用开销事实上是更主要的。由于目录在内存中，而内存系统在单位时间内只能进行有限数量的读/写操作，所以状态更新的数量最终由目录决定。如果某一并行程序要求大量的一致性操作（对共享数据块的大量读/写），

目录将最终限制并行程序的性能。

最后，从成本的角度来看，当处理器量增加时，用来存储目录的内存容量变大，其自身也会成为一种瓶颈。回忆目录的大小按$O(mp)$增加，其中m是内存块的数目，p是处理器的数目。减小目录的方案之一是增大内存块（这样在内存容量一定的条件下，就能减小m）。然而，这样会带来如假共享之类的其他开销——在程序中两个处理器更新不同的数据项，但这些数据项正好位于同一内存块中。这种现象在第7章作更详细的讨论。 52

由于目录构成争用的中心，人们自然想到把在多个处理器之间保持一致性的任务拆分。基本原理是让每个处理器保持自己内存块的一致性，假设在多个处理器之间有一个物理的（或逻辑的）内存块的分区存在。这就是分布式目录系统的原理。

分布式目录方案 在可扩展的体系结构中，内存物理分布于多个处理器之中。相应块的存在位也是分布式的，每个处理器负责维护自己内存块的一致性。图2-25b所示就是这样一种体系结构。由于每个块都有一个拥有者（通常可由内存块的地址算出），其目录位置对于所有处理器来说是隐式可知的。当某个处理器试图第一次读出块时，它向拥有者发送块请求，拥有者根据本地可获得的存在位及状态信息适当地引导该请求。同样，当某一处理器向内存块写入时，处理器向拥有者传播一个无效信息，然后无效信息转发到所有在高速缓存中含有该块副本的处理器。这样的方法使目录不再是中心，与中央目录相关的争用就减轻了。但是注意与状态更新消息相关的通信开销并没有减少。

分布式目录方案的性能 很明显，分布式目录允许$O(p)$个同时的一致性操作，前提是底层网络能够承受相应的状态更新消息。从这点来看，分布式目录本质上比侦听系统及中央目录系统具有可扩展性。对于这样的系统来说，网络的延迟及带宽成为主要的性能瓶颈。

2.5 并行计算机的通信成本

执行并行程序时，一个重要开销来自于处理器间的信息通信。通信成本与很多因素有关，如编程模型语义、网络拓扑结构、数据处理和路由选择以及相关的软件协议等。这些问题是本节讨论的中心。

2.5.1 并行计算机的消息传递成本

在网络的两个节点间传送一条消息所花的时间是准备传送消息所花的时间及消息从网络中传送到目的地所花时间之和。决定通信延迟的主要参数有以下几个： 53

1）*启动时间*（t_s）：启动时间是在发送节点和接收节点处理消息所花费的时间。它包括消息的准备时间（添加头、尾以及错误校正信息），执行路由算法的时间，以及在本地节点和路由器之间建立接口的时间。对于一条信息的传递来说，这种延迟只发生一次。

2）*每站时间*（t_h）：当消息离开一个节点后，需要花一定的时间到达路径上的下一个节点。消息头在两个直接连接的节点间传送所花费的时间称为每站时间，也称为节点延迟（node latency）。每站时间与决定消息将转发到哪个输出缓冲或通道的路由选择开关直接相关。

3）*每字传送时间*（t_w）：如果通道带宽是r个字每秒，那么每个字要花$t_w = 1/r$秒穿过链路。这个时间称为每字传送时间，它包括网络开销以及缓冲开销。

下面讨论两种已用于并行计算机的路由选择技术——存储转发路由选择及直通路由选择。

1. 存储转发路由选择

在存储转发路由选择过程中，消息穿过有多个链路的路径时，路径上的每个中间节点接收和存储完整的消息后，就把消息转发给下一个节点。图2-26a显示在存储转发路由选择网络传递消息的过程。

假设大小为m的消息要在这样的网络中传输，且要穿过l个链路。在每个链路，消息的头会导致t_h的时间开销，而消息的其他部分穿过链路会导致$t_w m$的时间开销。由于有l个这样的链路，总时间为$(t_h + t_w m)l$。因此，在存储转发路由选择过程中，m大小的消息穿过l个通信链路所需的总时间为

$$t_{comm} = t_s + (mt_w + t_h)l \tag{2-2}$$

在如今的并行计算机中，每站时间t_h非常小。对于绝大多数并行算法，即使m很小，t_h也小于$t_w m$，因此，t_h可以忽略不计。对于采用存储转发路由选择的并行平台，(2-2) 式可以简化为

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

2. 包路由选择

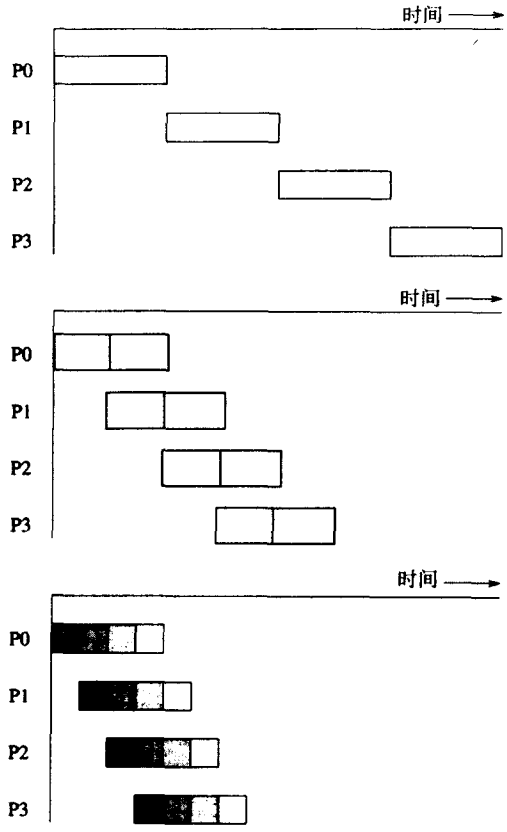
存储转发路由选择对通信资源的利用率很低。在某一个节点，一条消息只有在完全接收
54 后，才会传送到下一个节点（图2-26a）。考察如图2-26b所示的情况，原始消息在发送前被分成两个相等的部分。在这种情况下，中间节点只须等到原始消息的一半到达就可将消息传递下去。从图2-26 b看出，通信资源的利用率提高和通信时间减少都很明显。图2-26 c则更进一步，把消息分成4部分。除了提高通信资源利用率外，这个方法还有其他的好处——降低来自包损失（错误）的开销，降低从其他路径传送的可能性，以及提高错误校正的能力。因此，这个技术是像因特网这样的长距离通信网络的基础，在这样的网络中，错误率、站数目以及网络状态的变化都高得多。当然，每个包都必须携带路由选择信息、错误校正信息以及顺序信息，它们构成了网络开销。

考虑将m字的消息通过网络传送。对网络接口进行编程以及计算路由选择信息等所花时间
55 与消息长度无关。该时间累计在消息传送的开始时间t_s中。我们假设路由选择表在消息传送的过程中是静态的（即所有的包在同一条路径上传送）。虽然这不是在所有情况下都有效的假设，但它有利于建立一个消息传送的成本模型。消息被分成包，包再与它们的错误域、路由域以及顺序域组合。包的大小为$r + s$，其中r是原始消息，而s是包中带的附加信息。将消息分成包的时间与消息的长度成正比。这个时间用mt_{w1}表示。如果网络能在每t_{w2}秒传送一个字，在每站有t_h的延迟，同时如果第一个包穿过l个站，那么该包要花$t_h l + t_{w2}(r + s)$时间才能到达目的地。在此时间后，目的节点每隔$t_{w2}(r + s)$秒收到附加的包。由于附加包有$m/r-1$个，总通信时间由

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

给出，其中

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



a）在存储转发网络上发送简单消息

b）同样的消息分成两部分并在网络中传送

c）同样的消息分成4部分并在网络中传送

图2-26 从节点P_0传递消息到P_3：a) 通过存储转发通信网络；b) 和 c) 扩展为直通路由选择。带阴影区域代表消息在传送中的时间。这里假设与消息传送相关的开始时间为0

包路由选择适用于高动态状态及高错误率的网络，如局域网及广域网。这是因为每个包都可以选择不同的路由，可以只对丢失的包进行重发。

3. 直通路由选择

在并行计算机互连网络中，可以对消息传送添加另外的限制以减少与包开关相关的开销。通过强制所有包走同样的路径，就可以去除每个包的传送路由选择信息开销；通过强制顺序传送，顺序信息也可以去除；通过将错误信息与消息层而不是包层相联系，则与错误检测及修正相关的开销也可以减少；最后，因为并行计算机互连网络中的错误率非常低，可采用错误检测机制取代开销很大的错误校正方案。

这些优化的路由选择方案称为直通路由选择。在直通路由选择过程中，消息被分成固定大小的单元，称为流量控制数字（flow control digit）或数据片（flit）。由于数据片没有包的开销，它们可以比包小得多。首先从源节点向目的节点发送跟踪程序以建立两点间的连接。 56

连接完成后，数据片就一个接一个地传送。所有的数据片以吻合的形式通过同一路径。中间节点无须等待所有消息到达就可以发送数据片。当某一数据片被一中间节点收到后，该数据片被传递到下一节点。与存储转发路由选择不同，每个中间节点无须用缓冲区空间存储完整的信息。因此，直通路由选择在中间节点处只使用更小的内存和带宽，速度也快得多。

考虑在这样的网络中传送的一个消息。如果消息要穿过l个链路，t_h是每站时间，那么消息的头要花lt_h时间才能到达目的地。如果消息有m字长，那么在消息头到达后，再过$t_w m$时间，整个消息到达。因此，直通路由选择的整个通信时间为

$$t_{comm} = t_s + lt_h + t_w m \tag{2-3}$$

这个时间比存储转发路由选择有改进，因为直通路由选择中与站数及字数相关的量可看作执行加法运算，而前者是执行乘法运算。如果只在最近的两个节点间通信（也就是l =1），或者消息很小，那么存储转发路由选择和直通路由选择的通信时间相似。

绝大多数并行计算机及许多局域网支持直通路由选择。数据片的大小由各种网络参数决定。控制线路必须以数据片速度运行。因此，如果数据片很小，对与给定的链路带宽，所需的数据片速度就很大。这就给路由器的设计提出了难题，因为它要求控制线路以很高的速度运行。另一方面，当数据片增大时，内部缓冲区增大，消息传送的延迟随之增加。这两者都不符合需要。在当前的直通互连网络，数据片大小在4位到32字节之间。在许多主要依靠短消息（如高速缓存行）的并行编程模式中，消息延迟是关键问题。对于这类情况，把长消息分成短消息通过链路是不合理的。这类问题可用多道直通路由选择的路由器来解决。在多道直通路由选择时，单一物理通道被分成许多虚拟通道。

消息传送常量t_s、t_w以及t_h由硬件的特性、软件层以及消息传送语义决定。与像消息传递模式相关的消息传送语义最好使用长度可变的消息，而其他的用定长的短消息。对于长度可变的消息，有效带宽很重要，而对于定长短消息来说，减少延迟则更重要。消息传送层应能反映这些要求。

信息在网络中传送时，如果需要使用一个正在使用的链路，则该消息被阻塞。这样就会造成死锁。图2-27所示为一个直通路由选择网络的死锁实例。消息0，1，2，3的目的地分别
57 为A，B，C，D。来自消息0的数据片占据链路CB（以及相应的缓冲区）。然而，由于链路BA被来自消息3的数据片占据，造成来自消息0的数据片阻塞。同样，由于链路AD被占用，造成来自消息3的数据片阻塞。从图中可以看出，网络中没有消息能够继续通过，造成网络死锁。在直通网络，死锁可以通过使用适当的路由选择技术以及消息缓冲区来避免。这些内容在2.6节讨论。

4. 消息通信的简化成本模型

在2.5.1节中讲到，使用直通路由选择的两个相距l站的节点的消息通信成本由下式给出：

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

这个公式表明，为了优化消息传送的成本，需要做如下操作：

1）**大块通信** 这就是说，把多个小消息集中成一个大消息，这样就不用发送每条小消息
58 和为每条消息花费开始成本t_s，使开始延迟在大消息中减少。这是因为，在像机群和

消息传递计算机这样的常见平台，t_s的值要远大于t_h和t_w。

2）**减少数据的大小** 为了减少每字传送时间t_w的开销，有必要尽可能地减少待传送数据的大小。

3）**减小数据传送的距离** 减少消息必须通过的站的数目。

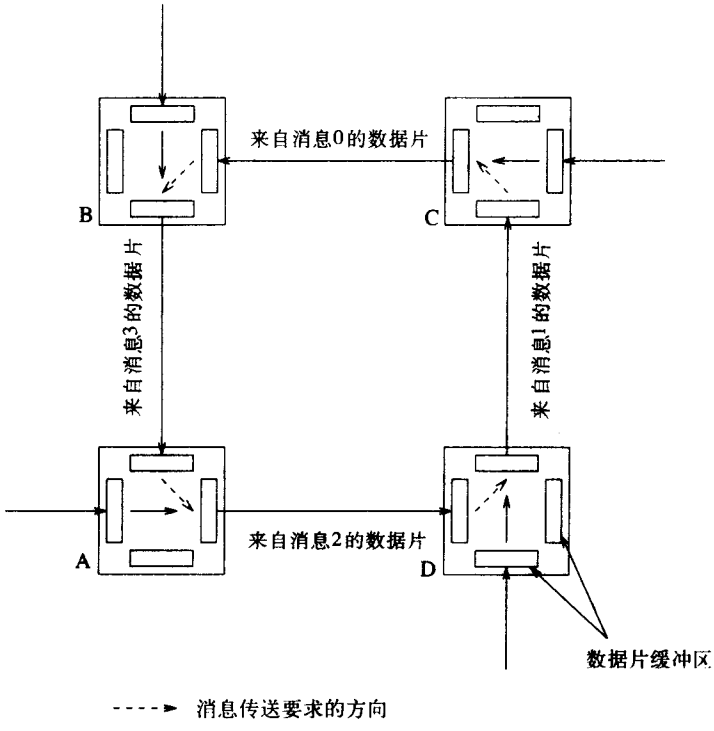


图2-27 直通路由选择网络的死锁实例

前两个目标很容易实现，但减少通信节点间的距离则很困难，很多情况下会造成算法设计者的不必要的负担。这是并行平台及模式如下性质的直接结果：

- 在许多像MPI这样的消息传递库中，程序员几乎不能控制把进程映射到物理处理器上。在这样的模式中，虽然任务可能有定义很好的拓扑结构，也可能只在任务拓扑结构的邻居间通信，但将进程映射到节点可能会破坏这种结构。
- 许多体系结构依靠随机路由选择（两步法），消息首先从源节点传送到一个随机节点，然后再从这个中间节点传到目标节点。这样就能避开网络中的热点及对网络的争用。在随机路由选择网络中减少站的个数没有好处。
- 每站时间（t_h）对小消息而言，主要由开始延迟决定，而对大消息则主要由每字的组成（$t_w m$）决定。由于在绝大多数网络中，站的最大值相对较小，每站时间可以忽略，而准确率几乎不会降低。

根据以上的特点，得出消息在网络中两个节点间传递的简化成本模型为

$$t_{comm} = t_s + t_w m \tag{2-4}$$

这个表达式对于体系结构独立的算法设计以及准确的运行时预测都有重要意义。由于该

成本模型包含在任意对节点间通信所花时间都相同，它与全连接网络对应。这样就不需要为各种特定的体系结构（比如格网、超立方体或树）设计算法，只须在设计算法时记住该成本模型，并将算法转到目标并行计算机中。

当算法由简化模型（假设为全连接网络）转到实际的计算机体系结构时，会带来一个重要的问题——准确性（保真度）预测降低。如果开始时假设t_h通常主要由t_s决定，或者t_w是有
59 效的，那么准确性只会降低很少。然而，这里要注意很重要的一点：我们的基本成本模型只对非拥塞网络有效。不同的体系结构对拥塞有不同的临界值；也就是说，线性阵列的拥塞临界值要比超立方体的低得多。此外，不同的通信模式对网络的拥塞程度也不一样。因此，只有当通信模式不拥塞网络时，上述的简化成本模型才有效。

例2.15 拥塞对通信成本的影响

考虑一个$\sqrt{p}\times\sqrt{p}$格网，它的每个节点都只和最邻近的节点通信。由于消息传送使用网络的链路不超过一个，通信操作的时间为$t_s + t_w m$，其中m是传送的字的数目。这个时间与简化模型一致。

下面考虑将上述情况修改为每个节点和另一个随机选择的节点通信。这种随机性意味着有$p/2$次通信（或$p/4$次双向通信）发生于格网的任何相等分区中（因为参与通信的节点位于任何一半的概率相等）。从对分宽度的讨论可知，2维格网的对分宽度为$\sqrt{p}$。由以上两点可以推断一些链路至少必须携带$\frac{p/4}{\sqrt{p}}=\sqrt{p}/4$条消息，这里假设通信通道是双向的。这些消息必须在网络中串行通过。如果每一条消息的大小为m，则此操作所需时间至少为$t_s+t_w m\times\sqrt{p}/4$。这个时间与简化模型不一致。 ■

上面的例子说明，对某一体系结构，有些通信模式不会造成拥塞，但另一些模式则有可能。这使建立通信成本模型的任务不仅仅要考虑体系结构，还要考虑通信模式。为了说明这一点，我们引进*有效带宽*（effective bandwidth）的概念。如果通信模式不会拥塞网络，那么有效带宽就与链路的带宽相同。但是，如果通信操作拥塞网络，有效带宽就变为链路带宽的一部分，缩小的比例为最拥塞链路上的拥塞度。此时有效带宽很难估计，因为它与进程到节点的映射、路由选择算法以及通信调度相关。因此，我们使用消息通信时间的下界。相应的链路带宽缩小的比例因子为p/b，其中b是网络的对分带宽。

在本文的后面，还会对使用有效每字时间t_w的消息传递应用简化的通信模型，因为这样可以按体系结构独立的方式设计算法。对算法中的通信操作造成网络拥塞以及如何影响并行运行时间，我们也要做出特别说明。本书中的通信时间适用于普通类型的k-d格网。虽然这些时
60 间在其他的体系结构中也可能实现，但这是通过基础体系结构得出的。

2.5.2 共享地址空间计算机的通信成本

将通信成本与并行编程相联系的主要目的，是为了将程序与质量相联系，以指导程序的开发。对于高速缓存一致的共享地址空间计算机，实现该目的要比在使用消息传递或非高速缓存一致结构的计算机上困难得多。其原因如下：

- 内存布局通常由系统决定。程序员除了改变数据结构之外，很少能控制特定数据项的位置以优化存取。这一点对于分布式内存共享地址空间体系结构尤其重要，因为该体系结

构很难区别本地与远程存取。如果对本地或远程数据项的存取时间有显著差别，那么通信成本在很大程度上依赖于数据布局。

- 有限大小的高速缓存会导致高速缓存颠簸。若某一节点需要整个数据的一部分来计算结果，如果这个部分比本地可用的高速缓存小，那么数据会在第一次访问时取出并进行计算。然而，如果该部分超出可用的高速缓存，那么数据的某些部分会被覆盖，随后再被访问几次。当程序变大，这种开销会造成严重的性能下降。为弥补这一点，程序员必须修改执行调度来减少工作区的大小（例如，习题2.5中串行矩阵相乘时分块循环）。此问题在串行及多处理器平台都很常见，但在多处理器系统中的性能损失则要大得多，因为每次失败都会带来一致性操作及处理器间的通信。
- 与无效及更新相关的开销很难量化。数据项被处理器取到高速缓存后，可能会被另一个处理器用于进行许多操作。例如，在无效协议中，远程处理器的写操作可能会使高速缓存行无效。在这种情况下，对该数据项的下一个读操作就会再次付出远程访问延迟的代价。同样，与更新协议相关的开销则会根据数据项的副本数量显著变化。一个数据项的并发副本数量以及指令执行调度通常不受程序员控制。
- 空间本地性很难模仿。因为高速缓存行通常长于一个字（从4到128字），即使是第一次访问，不同的字数也会有不同的与这些字相关的访问延迟。如果高速缓存行尚未被覆盖，则访问以前取过的邻近的字可能会非常快。同样，程序员对这一点很难控制，不同于改变数据结构以增大数据访问的空间本地性。 61
- 预取可以起到减少与数据访问相关开销的作用。如果有足够资源，编译器可以将加载提前，这样与这些加载相关的开销就可以被完全隐藏。由于预取与编译器、用到的程序以及资源的可用性（寄存器/高速缓存）相关，很难对它进行准确的模拟。
- 在许多程序中，假共享通常是重要的开销。由不同处理器（上的线程）使用的两个字可能位于同一高速缓存行中。这样就会造成一致性操作及通信开销，即使数据没有被共享。程序员必须充分研究不同处理器使用的数据结构，以减少假共享。
- 对共享访问的争用通常是共享地址空间计算机中的一项主要开销。不幸的是，争用与执行调度有关，故很难准确地对其模拟（独立于调度算法）。虽然通过对共享访问进行计数，可以得到它的渐近估计，但这样的估计往往没有太大的意义。

共享地址空间计算机的任何成本模型必须考虑上面讲到的所有开销。把这些开销全放到一个成本模型里，会使模型过于繁杂而难于编程，也会导致过于针对具体的计算机，使模型不能通用。

作为第一级模型，容易看到，存取一个远程字会导致某一高速缓存行被取到本地高速缓存中。与此存取相关的时间开销包括一致性开销、网络开销以及内存开销。一致性与网络开销与采用的互连方式有关（因为一致性操作必须传送给远程处理器，数据项必须被取出）。由于不知道某一次具体的存取与什么样的一致性操作相关，也不知道内存字从何处而来，我们就设存取含共享数据的高速缓存行的开销为常量。为了和消息传递模型保持一致，该常量称为t_s。因为有多种延迟隐藏协议（如预取）应用于现代计算机体系结构中，我们还假定常量t_s与初始存取m个字共享数据中的连续块有关，即使m大于高速缓存行的大小。我们再进一步假定访问共享数据的成本要高于存取本地数据（比如，在NUMA计算机中，本地数据很可能位于本地内存模块中，而由p个处理器共享的数据对于至少$p-1$个处理器都要从非本地模块中取

得）。因此，我们指定共享数据的每字存取成本为t_w。

从上面的讨论可以看出，不管是共享内存还是消息传递模式，计算在一对处理器之间共享m个字的单一块的成本仍然可以使用同样的表达式$t_s + t_w m$（公式2-4），所不同的是，在共享内存计算机中，常量t_s相对于t_w的值，要比在分布式内存计算机中小得多（对UMA计算机而言，
62 t_w接近于0）。注意，成本$t_s + t_w m$假定只读访问而无争用。如果多个进程访问同一数据，则成本要乘以进程的数目，就像在消息传递模型中一样，拥有数据的进程要把消息发给每个接收的进程。如果对数据进行读写访问，则处理器其后的访问也会带来成本，同一次写操作不同。这一点也和消息传递模型的相同。如果某一进程修改它接收到的消息的内容，则它必需把该消息送回给接下来要访问刷新后数据的进程。在共享地址空间计算机中，这个模型看上去过于简单，但是，该模型却能很好地估计在一对进程间共享m个字的数组的成本。

上面提到的简单模型主要考虑远程数据访问造成的开销，其他多种开销并没有考虑。对共享数据访问的争用，必须通过计算同时调度任务间的共享数据的访问次数显式地解决。该模型没有显式包含其他多种开销。由于不同计算机的高速缓存大小不一，很难用一种结构独立的方式，确定在何种情况下工作集的大小超过高速缓存的大小就能引起高速缓存颠簸。因此，在这种成本模型中，由于有限高速缓存带来的影响被忽略。最大化空间本地性（高速缓存行的作用）没有显式地包含在成本中。假共享同指令调度及数据布局有关。成本模型假设共享数据结构能够被恰当地填补，因此，不包含假共享成本。最后，成本模型也不考虑重叠的计算与通信。然而，即使对这些模型设计简单算法也是非常麻烦的。所以假设每个处理器只执行单一的并发计算单元，与在单处理器上进行多个并发计算（线程）相关的问题没有包含在模型中。

2.6 互连网络的路由选择机制

对于并行计算机的性能而言，有效的消息路由选择算法是非常重要的。*路由选择机制*（routing mechanism）确定了消息从源点传送到目标节点的路径，它将消息的源点和目标节点作为输入，还可能使用网络的状态信息，返回网络中从源点到目标节点的一条或多条路径。

路由选择机制可分成*最小化路由选择*及*非最小化路由选择*两种。在最小化路由选择机制中，总是选出从源点到目标节点的最短路径之一，在最小化路由选择方案中，每个链路都使消息更靠近目标，但这种方案会导致网络的一部分发生拥塞。非最小化路由选择方案与之相反，可能对消息沿更长的路径路由，以避免网络拥塞。

63 根据如何使用网络的状态信息，路由选择机制也可分成*确定性路由选择*（deterministic routing）和*自适应路由选择*（adaptive routing）两种。确定性路由选择方案根据消息的源点和目标节点，为消息确定唯一的路径，它没有使用任何网络状态信息。确定性方案可能导致对网络通信资源的不均衡使用。自适应路由选择方案与之相反，它使用网络的当前状态信息确定消息传送的路径。自适应路由选择能检测网络的拥塞信息，并能把消息绕过拥塞部分。

维序路由选择（dimension-ordered routing）是一种常用的确定性路由选择技术。基于由通道维数决定的编号方案，维序路由选择为消息传递提供相继的通道。维序路由选择用于二维格网时称为*XY路由选择*（XY-routing），用于超立方体网络时称为*E立方体路由选择*（E-cube routing）。

考虑一个无回绕连接的二维格网。按XY路由选择方案，消息首先沿X维发出，直到到达

目标节点的列，再沿Y维到达目的节点。以$P_{Sy,Sx}$表示源节点的位置，以$P_{Dy,Dx}$表示目标节点的位置，任何最小路由选择方案都会返回路径长度$|Sx - Dx| + |Sy - Dy|$。假设$Dx \geqslant Sx$及$Dy \geqslant Sy$。在XY路由选择方案中，消息沿X维通过中间节点$P_{Sy,Sx+1}$，$P_{Sy,Sx+2}$，…，$P_{Sy,Dx}$，再沿Y维通过节点$P_{Sy+1,Dx}$，$P_{Sy+2,Dx}$，…，$P_{Dy,Dx}$到达目标节点。注意这个路径长度的确为$|Sx - Dx| + |Sy - Dy|$。

超立方体连接网络的E立方体路由选择工作机理相似。考虑一个节点数为p的d维超立方体。令P_s和P_d分别为源节点和目标节点的标号。从2.4.3小节可知，用二进制对这些标号编码，其长度为d位。此外，节点间的最短距离由$P_s \oplus P_d$中1的个数给出（⊕表示位的异或操作）。在E立方体算法中，节点P_s计算$P_s \oplus P_d$的值并沿k维发送消息，其中k是$P_s \oplus P_d$中的最低非零有效位的位置。每个中间节点P_s接收到消息，计算出$P_s \oplus P_d$，再将消息沿着与最低非零有效位对应的维转发。这个过程一直进行到消息到达目标节点才结束。图2-16展示在三维超立方体网络中的E立方体路由选择过程。

例2.16 超立方体网络的E立方体路由选择

如图2-28所示的三维超立方体，令$P_s = 010$和$P_d = 111$分别表示消息传递的源节点和目标节点。节点P_s计算出$010 \oplus 111 = 101$。在第一步，节点P_s将消息沿着与最低有效位对应的维传送到节点011。节点011再沿着与最高有效位对应的维（$011 \oplus 111 = 100$）将消息送出。消息到达111，这是消息的目标节点。 ■

在本书的剩下部分，将使用确定性及最小消息路由选择来分析并行算法。 64

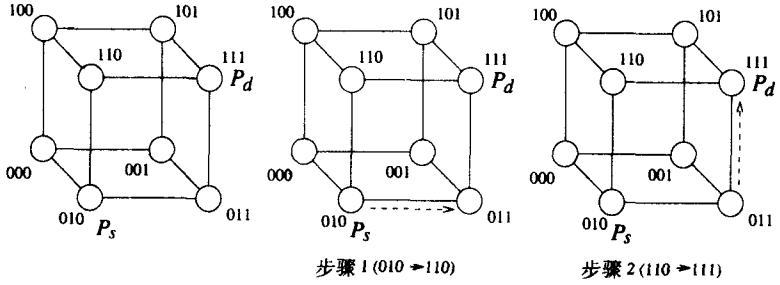


图2-28 在使用E立方体路由选择的三维超立方体中从源节点P_s (010) 到目标节点P_d (111)路由消息

2.7 进程-处理器映射的影响和映射技术

在2.5.1节中讲到，程序员通常不能控制将逻辑进程映射到网络中的物理节点上。因此，即使是本来不具拥塞性的通信模式也会使网络拥塞。我们用下面的例子说明这一点：

例2.17 进程映射的影响

图2-29中的基本体系结构为16个节点的格网，标号从1到16（图2-29a），算法按16个进程实现，标号为从“a”到“p”（图2-29 b）。格网中算法的执行经过调整，不会发生拥塞通信的操作。下面考虑如图2-29 c和d中所示的两个进程到节点的映射：图2-29 c所示只是一个直观的映射，基本体系结构中单一的链路仅携带与进程间单一的通信通道对应的数据。另一方面，在图2-29 d中，进程已被随机地映射到处理节点上。在这种情况下，易见计算机中每条链路携带进程间最多6个通道的数据。这样情况下，如果进程间通信通道上需求的数据率高，就有可

能造成更长的通信时间。 ■

从上例可以看出，虽然算法可以设计成不导致通信拥塞，但将进程映射到节点时还是会
65 诱发网络中的拥塞，并造成性能下降。

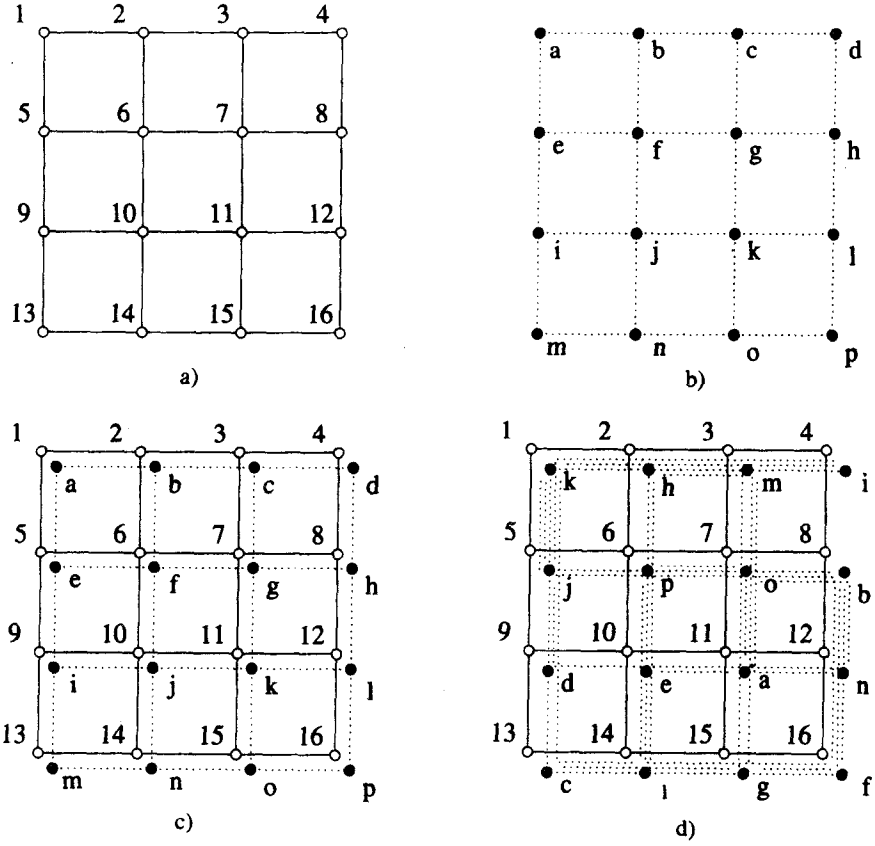


图2-29 进程映射对性能的影响：a）基本体系结构；
b）进程及其交互；c）直观的进程到节点
的映射；d）进程到节点的随机映射

2.7.1 图的映射技术

程序员通常无法控制进程-处理器映射，了解这种映射的算法是很重要的。因为这种映射可以用来度量某一算法的性能下降程度。假设有两个图G（V，E）和G'（V'，E'），图G到图G'的映射把集合V中的每个顶点映射到集合V'中的一个顶点（或顶点集合），并把集合E中的每一条边映射到E'中的一条边（或边的集合）。映射过程中，有三个参数很重要。第一，E中有可能不止一条边映射到E'的一条边上。映射到E'的任意边上的边数最大值称为映射拥塞度
66 （congestion）。在例2.17中，图2-29 c的拥塞度为1而图2-29 d的拥塞度为6。第二，E中的一条边可能被映射到E'中多个相邻的边上。这一点也很重要，因为在相应链路的通信必须穿过不止一条链路，可能导致网络拥塞。E中的任意一条边能映射到E'中的最大链路数目，称为映射膨胀度（dilation）。第三，集合V和V'可能包含不同数量的顶点。在这种情况下，V中的节点和V'中不止一个节点相对应。集合V'中节点的数目和集合V中节点数目之比称为映射扩充度

(expansion)。在进程-处理器映射这一部分，我们希望映射的扩充度与虚拟处理器及物理处理器的比率相同。

本节将讨论一些常见图的嵌入问题，如2维格网（第8章讲述矩阵操作）、超立方体（第8章讲述排序算法、第13章讲FFT算法）以及树（第4章讲广播及障碍）。讨论的范围将限制在集合V和V'有相同数量的节点（即扩充度为1）。

1. 将线性阵列嵌入超立方体

含2^d个节点（标号从0到2^d-1）的线性阵列（或环）可以嵌入到d维超立方体中，只需把线性阵列中的节点映射到超立方体的节点$G(i,d)$上。函数$G(i,x)$定义如下：

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

函数G称为二进制反射葛莱码（binary reflected Gray code, RGC）。$G(i,d)$表示d位葛莱码序列中的第i项。通过对d位葛莱码表的反射，对反映项加一个前缀1，对原项加前缀0，就得到$d+1$位葛莱码。该过程如图2-30 a所示。

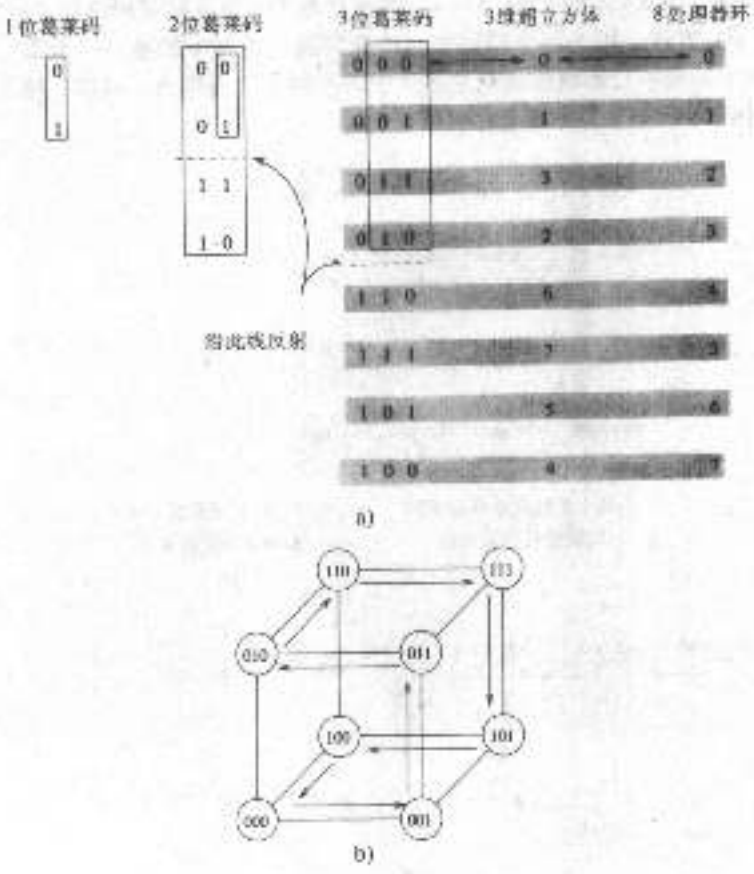


图2-30 二进制反射葛莱码，a）三位反射葛莱码环；
b）将环嵌入到三维超立方体中

仔细研究葛莱码表可见，两个相邻的项（$G(i, d)$和$G(i+1, d)$）只存在一位位置差异。由于在线性阵列中节点i映射到节点$G(i, d)$，节点$i+1$映射到$G(i+1, d)$，超立方体中存在直接链路与线性阵列中的每一个直接链路对应。（回忆超立方体中标号中只有1位位置不同的两个节点间存在直接链路。）因此，函数G指定的映射的膨胀度和拥塞度都是1。图2-30 b说明将8个节点环嵌入到3维超立方体中。

2. 将格网嵌入超立方体

67 将格网嵌入超立方体是将环嵌入超立方体的自然扩展。要将$2^r \times 2^s$的环绕格网嵌入到2^{r+s}个节点的超立方体中，只须将格网上的节点(i, j)映射到超立方体的节点$G(i, r-1) \| G(j, s-1)$上（这里$\|$表示两个葛莱码的串连）。格网中的直接邻居映射到超立方体中标号只差一位位置的节点上。因此，此映射的膨胀度和拥塞度都是1。

以将2×4格网嵌入8节点超立方体为例。r的值为1，s的值为2。格网的节点(i, j)映射到超立方体的节点$G(i, 1) \| G(j, 2)$上。因为$G(0, 0)$为0，$G(0, 2)$为00，将两者串连得到超立方体节点的标号。因此，格网中的节点$(0, 0)$就映射到超立方体的节点000上，同样，格网上的节点$(0, 1)$映射到超立方体的节点001上，以此类推。图2-31举例说明了将格网嵌入超立方体的过程。

68 将格网映射到超立方体中很有用。格网中同一行的所有节点映射到超立方体中有r个同样最高有效位的节点上。从2.4.3节可知，对$r+s$维超立方体的节点标号固定任意r位，就可得到含2^s个节点的s维子立方体。因为每个格网节点映射到超立方体中的唯一一个节点，格网中有2^s个节点，故格网中的每一行都映射到超立方体中的不同子立方体上。同样，格网中的每一列映射到超立方体中的不同子立方体上。

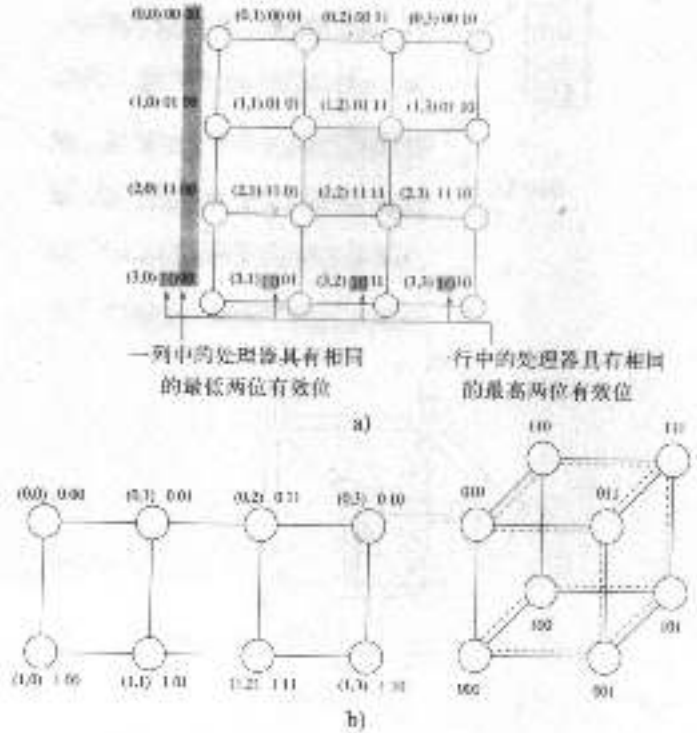
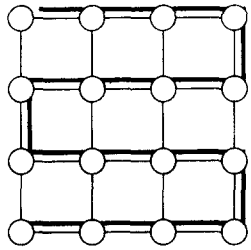


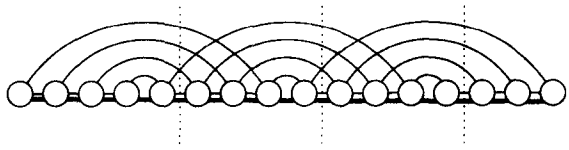
图2-31 将格网嵌入超立方体的过程：a）4×4格网节点映射到4维超立方体的节点中；b）2×4格网嵌入到3维超立方体中

3. 将格网嵌入线性阵列

上面所讲的都是将稀疏网络嵌入稠密网络中。2维格网有$2 \times p$条链路，而含p个节点的线性阵列有p条链路。因此，从2维格网到线性阵列的映射一定会造成拥塞。 69



a）将线性阵列映射到2维格网
（拥塞度为1）



b）反向映射——将2维格网映射到
线性阵列(拥塞度为5)

图2-32 从线性阵列到格网的映射：a）将16节点的线性阵列嵌入2维格网中；
b）反向映射。粗线代表线性阵列中的链路，正常线代表格网中的链路

首先来看将线性阵列映射到格网中。假设格网和线性阵列中都不含回绕连接。图2-32所示为一直观的从线性阵列到格网的映射。图中，粗线代表线性阵列中的链路，普通线代表格网中的链路。从图2-32 a中很容易看出，构建一个膨胀度为1、拥塞度为1的线性阵列到格网的映射是可能的。

下面再考虑相反的映射，即给定一个格网，用与上面相反的函数将格网中的顶点映射到线性阵列上。该映射如图2-32 b所示。与前面一样，粗线表示线性阵列中的边，正常线表示格网中的边。易见图中所示映射的拥塞度为5，即粗线与不超过5条普通线相连。对一般的p节点映射而言，这个（反）映射的拥塞度为$\sqrt{p}+1$（对于$\sqrt{p}$条到下一行的边为每条一个，以及一条附加的边）。

事实上，对上面用到的简单映射可以加以改善。我们从两个网络的对分宽度着手。前面讲过，无回绕连接的2维格网的对分宽度为$\sqrt{p}$，线性阵列的是1。假设最好的将2维格网映射到线性阵列的拥塞度为r，它表示如果将线性阵列从中间断开，那么只会切断一条线性阵列的链路，或者不超过r条格网链路。我们要求r不小于格网的对分宽度，因为对线性阵列的对分 70
同样会使格网对分，故至少$\sqrt{p}$条格网链路要穿过分区，即连接两个相等部分的线性阵列链路至少与$\sqrt{p}$个格网链路相连。因此，任何映射的拥塞度下界为$\sqrt{p}$，这与图2-32 b中所讲的简单（反向）映射拥塞度大致相同。

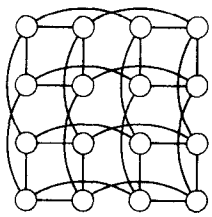
当映射稠密网络到稀疏网络时，上面建立起来的下界有更一般的适用性。对于从含x条链路的网络S到含y条链路的网络Q的映射，人们会想到拥塞度下界为x/y。从格网到线性阵列的映射，拥塞度下界则是$2p/p$或2。然而，该下界过于保守。比较两个网络的对分宽度得出的下界则更贴近。这一点在下一节中将进一步讨论。

4. 将超立方体嵌入2维格网

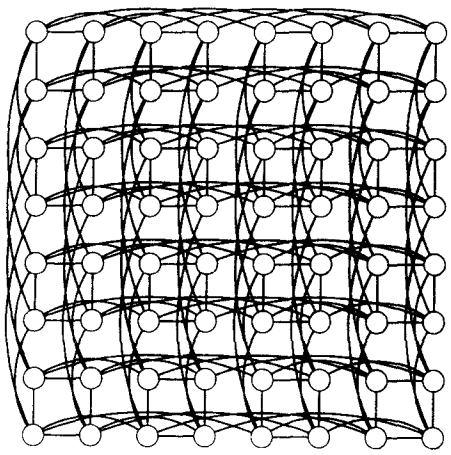
考察将p个节点超立方体嵌入到p个节点的2维格网。为方便起见，假设p是2的偶数次幂。这样就能把超立方体想像成$\sqrt{p}$个子立方体，每个子立方体有$\sqrt{p}$个节点。再令$d = \log p$为超立方体的维。由假设d是偶数，可使用$d/2$个最低有效位来定义含$\sqrt{p}$个节点的子立方体。例如，对4维超立方体而言，可使用最低的两位来定义子立方体为（0000, 0001, 0011, 0010）、（0100, 0101, 0111, 0110）、（1100, 1101, 1111, 1110）以及（1000, 1001, 1011, 1010）。如果将所有子立方体的$d/2$个最低有效位固定，就能得到另一个由$d/2$个最高有效位定义的子立方体。例如，如果把所有超立方体的最低2位固定为10，就会得到节点（0010, 0110, 1110, 1010）。读者可以验证它与一2维子立方体对应。

从超立方体到格网的映射可以按以下方法定义：每$\sqrt{p}$个节点的子立方体映射到格网的$\sqrt{p}$个节点的行中。只要把线性阵列逆映射到超立方体映射就能做到这一点。含$\sqrt{p}$个节点的超立方体的对分宽度为$\sqrt{p}/2$。对应的含$\sqrt{p}$节点的行的对分宽度为1。因此从子立方体到行的映射的拥塞度为$\sqrt{p}/2$（在连接两个对半行的边上）。图2-33 a说明p为16的情形，图2-33 b说明p为32时的情形。按这种方式，可以把每个子立方体映射到格网中的不同行上。这里要注意的是，上面计算出的拥塞度来自于子立方体到行的映射，我们没有讨论来自列映射的拥塞。如果将超立方体中$d/2$个最低有效位相同的节点映射到格网的同一列上，就变成了子立方体到列的映射。在子立方体到列的映射中，每个子立方体或列都有$\sqrt{p}$个节点。按与子立方体到行映射的相同的论证，可得子立方体到列映射的拥塞度也是$\sqrt{p}/2$。由于来自行映射的拥塞和
71 来自列映射的拥塞所影响的边的集合不相连，总的拥塞度还是$\sqrt{p}/2$。

按照2.7.1节相同的论证可以得到拥塞度的下界。由于超立方体的对分宽度为$p/2$，格网的对分宽度为$\sqrt{p}$，拥塞度的下界为两者的比值，即$\sqrt{p}/2$。注意我们的映射得到这个拥塞度的下界。



a) $p = 16$



b) $p = 32$

图2-33 将超立方体嵌入2维格网

5. 进程-处理器映射与互连网络设计

前面的分析表明，可以将稠密网络映射到与拥塞开销相关的稀疏网络中。这就是说，一个稀疏网络，如果其链路带宽增大到可以抵消拥塞，就可以按稠密网络一样运行（以膨胀效应为模）。例如，若格网的链路速度比超立方体的链路速度快$\sqrt{p}$/2倍，则格网的性能与超立方体相似。这样的格网称为胖格网。胖格网与超立方体有同样的对分带宽，但直径更大。在2.5.1小节中讲过，如果使用适当的消息路由选择技术，节点距离的影响可以缩小。指出这样一点是重要的，高维网络往往会导致网络布局复杂、线路纵横以及线长不一等缺点。因此，人们更喜欢用胖的低维网络来设计互连。下面将对并行体系结构的成本-性能平衡作更具体的研究。 72

2.7.2 成本-性能平衡

下面开始研究为何可用不同的成本度量来探讨互连网络中的成本-性能平衡。我们通过分析具有同样成本的格网和超立方体的性能来说明。

如果网络的成本与线路的数量成正比，那么对于每个通道有（log p）/4条线路的正方形p节点环绕格网，其成本与每个通道一条线路的p节点超立方体相同。比较这两种网络的平均通信时间可得，二维环绕格网中任意两节点间的平均距离l_{av}是$\sqrt{p}$/2，而超立方体中为（log p）/2；在使用直通路由选择的网络，相距l_{av}站的两个节点间传送m大小的消息所花时间为$t_s + t_h l_{av} + t_w m$。由于格网的通道宽度增大（log p）/4倍，每字传送时间减小同样的倍数。因此，如果在超立方体中的每字传送时间为t_w，在通道变胖的格网中的时间就是$4t_w/(\log p)$。由此可得，在超立方体中，平均通信延迟为 $t_s + t_h (\log p)/2 + t_w m$，而在同样成本的环绕格网中，平均通信延迟为$t_s + t_h \sqrt{p}/2 + 4t_w m/(\log p)$。

仔细研究这两个表达式可见，对于固定数目的节点，当消息增大时，t_w起决定作用。比较两个网络的t_w可见，如果p大于16并且消息m足够大，那么环绕格网中花的时间$4t_w m/(\log p)$要小于超立方体中的时间$t_w m$。在这种条件下，用直通路由选择，环绕格网中随机节点对之间的大消息的点对点通信时间小于同样成本的超立方体中的通信时间。而且，如果通信算法适用于格网，则每个通道的额外带宽将导致更好的性能。注意，在用存储转发路由选择时，格网不再比超立方体更具成本有效性。此时可用一般的k元d立方体来进行相应的成本-性能平衡分析（习题2.25 ~ 2.29）。

上面的通信时间是在轻负载的条件下计算出的。当消息增加，网络中就会出现争用。争用对格网网络的负面影响要大于超立方体网络。因此，如果网络负载很重，使用超立方体要比使用格网更好。

如果网络成本与对分宽度成正比，那么每通道$\sqrt{p}$/4条线路的p节点环绕格网与每通道一条线路的p节点超立方体的成本相同。下面再以这种成本度量用与上面相似的方法来分析成本-性能平衡。由于格网通道宽$\sqrt{p}$/4倍，每字传送时间减少同样倍数。因此，成本相同的超立方体和格网的通信时间分别为$t_s + t_h (\log p)/2 + t_w m$和$t_s + t_h \sqrt{p}/2 + 4t_w m/\sqrt{p}$。再次表明，对于数量一定的节点，当消息变大时，$t_w$将起决定作用。将两种网络比较得出，当$p$ >16且消息足够大时，同样成本的格网性能超过超立方体。因此，对于足够大的消息，在网络负载较轻的 73
条件下，同样成本的格网性能总是好于超立方体。即使网络负载较重，同样成本的格网的性能也与超立方体相当。

2.8 书目评注

有许多教科书讨论高性能体系结构的若干方面[PH90, PH96, Sto93]。在[CSG98, LW95, HX98, Fly95, AG94, DeC89, HB84, Lil92, Sie85, Sto93]里对并行体系结构与互连网络有很好的描述。从历史的角度上看，Flynn [Fly72]提出了将并行计算机分为SISD，SIMD以及MIMD。他还推出了MISD（多指令流单数据流）模型。MISD模型比其他类型的模型少见，虽然它可以看作流水线执行的模型。Darema [DRGNP]提出了单程序多数据（SPMD）模式。Ni[Ni91]提出基于硬件体系结构、地址空间、通信模型、语言、编程环境以及应用等的并行计算机的层次分类。

数十年来，互连网络成为人们关注的领域。Feng[Fen81]提供静态及动态互连网络教程。Stone[Sto71]介绍了全混洗互连模式。Lawrie[Law75]中提出了Omega网络。其他的多级网络也被提出，包括Flip 网络[Bat76]以及Baseline网络[WF80]。Leighton[Lei92]讨论树状格网及金字塔格网，对其他相关的网络也有详细的介绍。

C.mmp是早期基于交叉开关的MIMD共享地址空间并行计算机研究原型[WB72]。Sun Ultra HPC Server及富士通VPP500是基于交叉开关的并行计算机或其变形。基于多级互连网络的并行计算机包括BBN Butterfly [BBN89]、NYU Ultracomputer [GGK+83]以及IBM RP-3[PBG+85]等。SGI Origin 2000、Stanford Dash [LLG+92]以及KSR-1 [Ken90]都是NUMA共享地址空间计算机。

Cosmic Cube [Sei 85] 是最早的基于超立方体连接网络的消息传递并行计算机。接下来有nCUBE 2[nCU90]以及Intel的iPSC-1，iPSC-2，iPSC/860。最近的SGI Origin 2000采用类似于超立方体的网络。Saad 及Shultz [SS88，SS89a]从超立方体连接网络以及其他许多静态网络[SS89b]中得出了不少有趣的性质。许多并行计算机基于格网网络，如Cray T3E。Intel Paragon XP/S [Sup 91]和Mosaic C [Sei 92]是更早的基于二维格网的计算机的例子。MIT J-Machine [D+92]基于三维格网网络。通过用广播总线扩大格网网络可以提升格网连接计算机的
74 性能[KR87a]。Miller et al. [MKRS88]提出了格网结构重组（习题2.16中的图2-35）。其他的重组格网的例子还包括TRAC和PCHIP。

DADO并行计算机基于树网络[SM86]，它使用了深度为10的完全二叉树。Leiserson [Lei85b]提出了胖树互连网络并证明了该网络的几个有趣性质。他证明了在硬件条件一定的前提下，胖树的性能最好。Thinking Machines CM-5 [Thi91]是基于胖树连接网络的计算机。

Illiac IV [Bar68]是最早的SIMD并行计算机之一。其他的SIMD计算机包括Goodyear MPP [Bat80]，DAP 610，CM-2[Thi90]，MasPar MP-1以及MasPar MP-2 [Nic90]。CM-5和DADO中结合SIMD和MIMD的特点，两者都是MIMD计算机，但有实现快速同步的硬件，使它们可以按SIMD模式操作。CM-5中有用来增大数据网络的控制网络，该控制网络提供广播、归约、结合以及其他的全局操作。

Leighton [Lei92]以及Ranka和Sahni [RS90b]讨论将某一互连网络嵌入到另一个网络中。在Reingold [RND77]中讨论了嵌入线性阵列和格网结构时用到的葛莱码。Ranka和Sahni [RS90b]讨论拥塞度、膨胀度以及扩充度的概念。

Ni和McKinley [NM93]对直通路由选择技术作了全面的评述。Dally和Seitz [DS86]提出虫孔路由选择技术。Kermani和Kleinrock [KK79]讲述了称为虚拟直通（virtual cut-through）路

由选择的相关技术，该技术在每个中间节点提供通信缓冲。Dally和Seitz [DS87]讨论通道依赖图的无死锁虫孔路由选择。通常用基于维数的确定性路由选择方法来避免死锁。在几种并行计算机中已经使用了直通路由选择技术。超立方体的E立方体路由选择方案由[SB77]提出。

Dally [Dal90b]讨论消息传递计算机所使用网络的成本性能平衡。使用网络的对分宽度作为网络成本的度量，他指出低维网络（如二维格网）的成本有效性要比高维网络（如超立方体）高得多[Dal87, Dal90b, Dal90a]。Kreeger和Vempaty [KV92]得出了格网与超立方体连接计算机——通信时带宽的平衡因子（4.5节）。Gupta 和Kumar [GK93b]分析格网和超立方体网络中FFT计算的成本性能平衡问题。

在[FW78, KR88, LY86, Sni82, Sni85]中广泛研究了PRAM的性质。由Akl [Akl89]、Gibbsons [GR90]以及Jaja [Jaj92]所著的书里讲述PRAM算法。本书关于PRAM的讨论即基于Jaja [Jaj92]所著的书。许多处理器网络用来仿真PRAM模型[AHM87, HP89, LPP88, LPP89, MV84, Upf84, UW84]。Mehlhorn 和Vishkin [MV84]提出用*模块并行计算机*（module parallel computer, MPC）来仿真PRAM模型。MPC是一种含p个处理器的消息传递并行计算机，每个处理器都有固定容量的内存，处理器之间以全连接网络互连。如果总内存的增长倍数为log p， 75
则MPC在$T \log p$步内能概率仿真PRAM的T步。MPC模型的主要缺点是，如果处理器数目很多，则很难构建全连接网络。Alt et al. [AHMP87]推出另一个称为*限定度网络*（bounded-degree network, BDN）的模型。在这种网络中，每个处理器都与固定数目的其他几个处理器相连。Karlin 和Upfal [KU86] 在BDN上对PRAM描述量级为$O(T \log p)$的概率仿真。Hornick 和Preparata [HP89]提出连接若干组处理器和内存池的双向网络。他们对基于树格网的消息传递MPC和BDN进行了研究。

人们对PRAM模型提出了许多修改，使它更接近现实并行计算机。Aggarwal，Chandra和Snir[ACS89b]提出LPRAM（本地内存PRAM）及BPRAM（块PRAM）模型[ACS89b]。他们还介绍了计算的分层内存模型[ACS89a]。在这种模型中，不同层次的内存单元在不同的时间存取。这种模型的并行算法在使用数据前把数据放到快速内存单元里，用完后再送回到慢内存单元，以此来导出数据的本地性。此外，还提出了其他的PRAM模型，如phase PRAM[Gib89]，XPRAM[Val90b]以及延迟模型[PY88]等。许多学者研究了并行计算机的抽象通用模型 [CKP+93a, Sny86，Val90a]。同时提出了一些具有类似目的的模型，如BSP[Val90a]，Postal 模型[BNK92]，LogP[CKP+93b]，A^3[GKRS96]，C^3[HK96]，CGM[DFRC96]以及QSM[Ram97]等。

习题

2.1 设计一个实验（即设计编写程序并作测量）确定你的计算机的内存带宽，并估算不同级高速缓存的大小。用该实验估算出计算机的带宽及L1高速缓存，并证明结果的正确性。（*提示*：测试带宽时无须重用数据。测试高速缓存大小时则需要重用数据，以便观察高速缓存的效果，并增加高速缓存大小，直至数据重用明显下降。）

2.2 某内存系统，1级高速缓存为32 KB，DRAM为512 MB，处理器频率为1 GHz。L1高速缓存的延迟为1个周期，DRAM的延迟为100周期。在每个内存周期，处理器取出4个字（高速缓存行大小为4个字）。两个向量点积可获得的峰值性能是多少？注意：有必要的话考虑优

化的高速缓存替代策略。

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

2.3 考察用两重循环的点积公式将稠密矩阵与向量相乘的问题。矩阵为$4K \times 4K$。(存储
76 矩阵的每一行需16 KB。)使用这种方法可获得的峰值性能是多少?

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

2.4 续上题,将两个$4K \times 4K$稠密矩阵相乘。如果采用三重循环的点积公式,那么可获得的峰值性能是多少?(假设矩阵按以行为主的方式排列。)

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

2.5 重组矩阵相乘算法,以获得更好的高速缓存性能。缺少空间本地性是矩阵相乘算法性能较差的最直接原因。有时候,从内存中取出的4个字中有三个被浪费了。为了解决这个问题,我们每次让结果矩阵的元素计算4次。用这种方法,只须简单地重组程序就可提高FLOP次数。然而,性能还可以有更大的提升。我们可以把矩阵相乘问题看作是一个立方体,每一个内部网格点与一次乘法-加法运算对应。矩阵相乘算法以不同的方式横穿立方体,就得出了对立方体的不同划分。用于计算分区的数据随着分区的输入面的表面积增大而增加,而计算随着分区的体积增大而增加。上面讨论的算法从立方体中切出薄片,使得产生的面积与体积相当(这导致很差的高速缓存性能)。为弥补这一缺陷,可将立方体分成三个$k \times k \times k$的子立方体对计算进行重组。与每个立方相关的数据为$3 \times k^2$(每个矩阵k^2数据),计算量为k^3。只有当$3 \times k^2$等于$8K$时才能达到最佳性能,因为$8K$为可用高速缓存的大小(这里假设与习题2.2中的各个计算机参数相同)。这对应于$k = 51$。与该立方体相对应的运算为132 651次乘法-加法运算或265 302次浮点操作。为进行这个计算,需用两个51×51的子矩阵,对应于5202个字或1301个高速缓存行。对这些高速缓存行存取需用130 100纳秒,因为265 302次浮点运算在130 100纳秒内完成,故峰值计算速度为2.04 GFLOPS。对该例编写代码,并画出性能作为k的函数曲线。(可在任意普通计算机上编程。务必注意微处理器类型、时钟频率以及各级可用高速缓存的大小。)

2.6 考虑一个具有分布式共享地址空间的SMP。它的一个简单成本模型为:访问本地高
77 速缓存需10纳秒,访问本地内存为100纳秒,访问远程内存为400纳秒。某一并行程序运行于该计算机上,程序负载均衡,存取的80%是对本地高速缓存,10%对本地内存,10%对远程内存。这个计算的有效内存访问时间是多少?如果计算是内存受限的,那么峰值计算速度是多少?

再考虑在一个处理器上的同样计算。这里,处理器70%的时间命中高速缓存,30%的时间命中本地内存。单一处理器的有效峰值计算速度是多少?在并行结构中单一处理器的计算速度与串行结构相比的比值是多少?

提示：多处理器的高速缓存命中率高于单处理器。这是因为多个处理器上的累计高速缓存大于单一处理器系统的高速缓存。

2.7 消息传递计算机和共享地址空间计算机的主要区别是什么？请概括出两者的优点和缺点。

2.8 为什么很难构建真正的共享内存计算机？将p个处理器连接到b个字的共享内存上（每个字都可以独立存取），所需的最小开关数是多少？

2.9 在4种PRAM模型（EREW，CREW，ERCW及CRCW）中，哪种模型功能最强大？为什么？

2.10 [Lei92]蝶形网络（Butterfly network）是含log p个层的互连网络（与omega网络相似）。在蝶形网络中，在层l上的每一个开关节点i既与$l+1$层上编号相同的节点相连，又与编号与自己只在第l最高有效位不同的开关节点相连。因此，在l层，如果$j = i$或者$j = i \oplus (2^{\log p-l})$，则开关节点$S_i$与$S_j$相连。

图2-34为8个处理节点的蝶形网络。证明蝶形网络与omega网络的等价性。

提示：对omega网络的开关节点重排，使之看上去像蝶形网络。

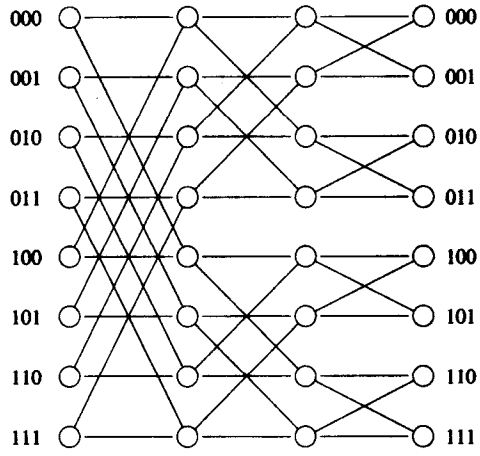


图2-34 含8个处理节点的蝶形网络

2.11 在2.4.3节讲到的omega网络是拥塞网络（就是说，当某处理器使用网络访问某一内存单元时会妨碍另一处理器访问另一内存单元）。设omega网络中有p个处理器，定义一个函数f，将$P = [0, 1, \ldots, p-1]$映射到P的一个置换P'上（即对所有的$0 \leq i < p$有$P'[i] = f(P[i])$和$P'[i] \in P$）。将此函数看作是处理器的通信请求映射，即处理器$P[i]$请求与$P'[i]$通信。

1）有多少个不同的置换函数存在？

2）这些函数中有多少个函数导致非拥塞通信？

3）任意一个函数导致非拥塞通信的概率是多少？

78

2.12 在图中，如果某一路径的起点和终点相同，则该路径称为圈。圈的长度是圈中边的数目之和。证明在d维超立方体中不存在长度为奇数的圈。

2.13 d维超立方体中的标号使用d位。如果对这些位中的k位固定，证明标号在剩余$d-k$个位置上不同的节点构成一个含$2^{(d-k)}$个节点的$(d-k)$维子立方体。

2.14 设A和B为d维超立方体中的两个节点。定义$H(A, B)$为A和B之间的Hamming距离，$P(A, B)$为连接A和B的不同路径的数目。这些路径称为*并行路径*，并且除了A和B之外，没有共同的节点。试证明下面4个结论:

1）A和B之间以通信链路表示的最小距离由$H(A, B)$给出。

2）任意两个节点间的并行路径总数目为$P(A, B) = d$。

3）A和B之间长度为$H(A, B)$的并行路径数目为$P_{length = H(A,B)}(A, B) = H(A, B)$。

4）剩余的$d-H(A, B)$条并行路径的长度为$H(A, B)+2$。

2.15 在超立方体对分带宽的非形式推导中，我们用超立方体的结构证明，d维超立方体可由两个$(d-1)$维的超立方体构成。我们证明，由于这些子立方体中相应的节点有直接的链路，共有$2d-1$条链路越过划分。但是，可以将超立方体分成两部分，使得两个划分都不是超立方
79 体。证明任意这样一种划分，在它们之间都有超过$2d-1$条链路存在。

2.16 [MKRS88] $\sqrt{p}\times\sqrt{p}$ 可重排格网（reconfigurable mesh）由 $\sqrt{p}\times\sqrt{p}$ 个处理节点阵列组成，它们与网格形的可重排广播总线相连接。图2-35所示为4×4可重排格网，每个节点都有本地可控制总线开关。在北（N）、东（E）、西（W）与南（S）4个端口间的内部连接可在算法的执行过程中设定。注意有15种连接方式。例如，{SW，EN}表示端口S与端口W相连，并且端口N与端口E相连。在任何时刻，总线上的每一位都带有一位1-信号或0-信号。开关可以让广播总线分成子总线，提供更小的可重排格网。对于给定的一组开关设置，子总线（subbus）是最大相连的节点的子集。不同于总线和开关，可重排格网与标准的二维格网类似。假定在任意时刻，只允许有一个节点在多个节点共享的子总线上广播。

对于一个由 $\sqrt{p}\times\sqrt{p}$ 个处理节点组成的可重排格网，确定其对分带宽、直径、开关节点数目以及通信链路数目。与环绕格网相比，可重排格网的优点和缺点是什么?

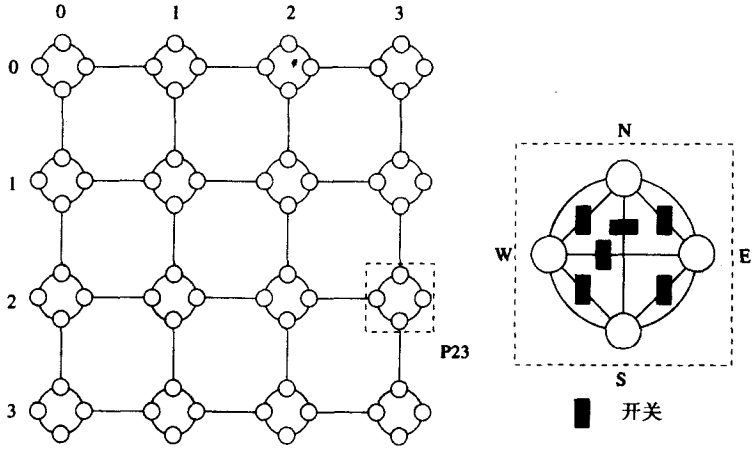


图2-35 可重排格网的开关连接模式

2.17 [Lei92]*树格网*是用树状连接将一个网格上的处理节点互连的一种网络。$\sqrt{p}\times\sqrt{p}$ 树格网按如下方法构成：从 $\sqrt{p}\times\sqrt{p}$ 网格开始，对网格的每一行构建一个完全二叉树。然后再对每一列构建一个完全二叉树。图2-36展示4×4树格网的构成。假设中间层为开关节点。
80 确定 $\sqrt{p}\times\sqrt{p}$ 树格网的对分宽度、直径以及开关节点总数。

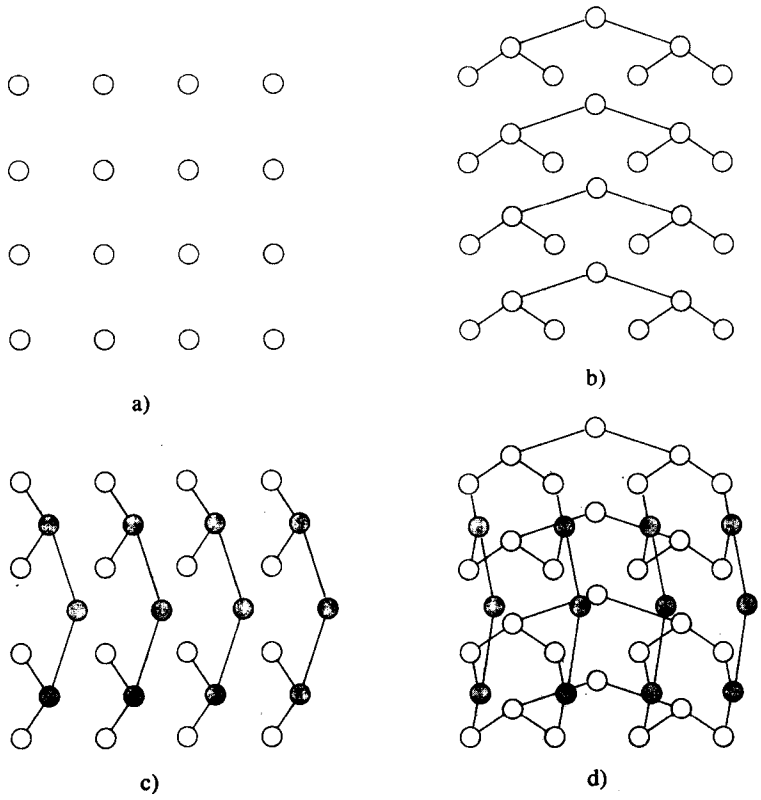


图2-36 4×4树格网的构成：a）4×4网格；b）对每一行构建的完全二叉树；c）对每一列构建的完全二叉树；d）完全4×4树格网

2.18 [Lei92]推广二维树格网（习题2.17）至d维，以构建$p^{1/d} \times p^{1/d} \times \cdots \times p^{1/d}$的树格网。构成方法如下：在所在维上把网格位置固定成不同的值，在变化的一维中构建一棵完全二叉树。

试求出$p^{1/d} \times p^{1/d} \times \cdots \times p^{1/d}$树格网中开关节点的总数。计算直径、对分宽度以及由总线路数表示的线路成本。与环绕格网相比，树格网的优点与缺点是什么？

2.19 [Lei92]与树格网相关的网络是d维金字塔格网（pyramidal mesh）。d维金字塔格网通过在处理节点网格上构建金字塔而成（与树格网的完全树相反）。产生方法如下：在树格网中，固定除一维以外的所有维，树就在剩下的这维上构建。在金字塔中，固定除两维以外的所有维，金字塔就构建在由这两维构成的格网上。在树中，k层的每个节点i与$k-1$层的节点$i/2$ 81
相连。同样，在金字塔中，k层的节点(i, j)与$k-1$层的节点$(i/2, j/2)$相连。而且，每一层的节点都连到一个格网上。图2-37为二维金字塔格网。

对于$\sqrt{p} \times \sqrt{p}$金字塔格网，假设中间节点为开关节点，试导出直径、对分宽度、弧连通率以及以由通信链路和开关节点数目表示的成本。与树格网相比，金字塔格网的优点和缺点是什么？

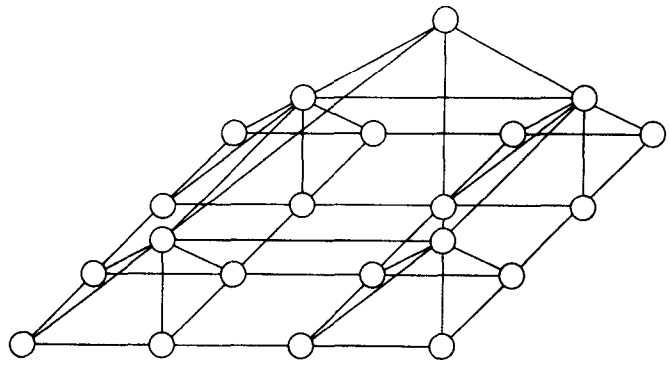


图2-37 4×4金字塔格网

2.20 [Lei92]超立方体连接网络的缺点之一是网络中的线路长度不同，即数据在不同的链路上传输所花的时间不同。很明显，二维环绕格网也存在这种缺点。然而，用定长的线路也能够构建二维环绕。请画出这样的4×4环绕说明这种布局。

2.21 说明如何将p个节点的三维格网嵌入到p节点的超立方体中。这种嵌入允许p取什么值？

2.22 说明如何将p个节点的树格网嵌入p个节点的超立方体中。

2.23 考察一个含2^d-1个节点的完全二叉树，它的每个节点都是处理节点。该树向d维超立方体的最小膨胀度映射是什么？

2.24 **最小拥塞度映射**（minimum congestion mapping）是很有用的概念。考虑两台采用不同互连网络的并行计算机，存在从第一台计算机向第二台计算机的拥塞度为r的映射。不考虑映射的膨胀度，如果第二台计算机的每条通信链路比第一台快r倍以上，则第二台计算机严格地优于第一台。

82 现在将d维超立方体映射到2^d节点的格网上。不考虑映射的膨胀度，从超立方体到格网的最小拥塞度映射是什么？利用求出的结果，判断链路速度为25M字节每秒含1024节点的格网的性能是否严格地优于链路速度为2M字节每秒含1024节点的超立方体（其节点与格网中使用的相同）？

2.25 导出含p个节点的k元d立方体的直径、链路数目以及对分带宽。若l_{av}为网络中任意两个节点间的平均距离，试求出k元d立方体的l_{av}。

2.26 若并行计算机使用存储转发路由选择。大小为m的消息经长度为d的路径从P_{source}传送到$P_{destination}$的成本为$t_s + t_w \times d \times m$。一种可替代的传送大小为$m$的消息的方法如下：将消息分成$k$部分，每部分的大小为$m/k$，然后将这$k$个不同的消息一个接一个地从$P_{source}$传送到$P_{destination}$。对于这种新方法，在下面两种情况下，导出将大小为$m$的消息传送到$d$站以外所需时间的表达式。

1）假设在路径中，当前一条消息到达下一个节点后，另一条消息就能从P_{source}发出。

2）假设只有当前一条消息到达$P_{destination}$后另一条消息才能从P_{source}发出。

对于每种情况，当k的值在1和m之间变化时，试讨论表达式的值。如果t_s很大，或者$t_s = 0$时，k的最优值是多少？

2.27 对p个节点的超立方体网络，假设每个通信链路的通道宽度为1。使k元d立方体($d <$

log p)网络与超立方体的成本相等，可增加k元d立方体中链路的通道宽度。现使用下面两种不同的度量来估计网络的成本：

1）以网络中总的线路数量表示成本（线路总数是通信链路数目和通道宽度的乘积）。

2）用对分带宽作为成本的度量。

利用上面的两种度量，再令k元d立方体与超立方体的成本相同，求出具有同样节点数、通道速度以及成本的k元d立方体的通道宽度。

2.28 习题2.25和2.27的结果可用来对静态互连网络作成本性能分析。考虑直通路由选择p节点k元d立方体网络。设含p个节点的超立方体连接网络的通道宽度为1。将其他网络的通道宽度按比例提高，直到其成本与超立方体的相同。设s和s'为习题2.27中两种不同度量下使成本相等导出的通道宽度的比例因子。对于这两个因子，将任意两个节点间的通信时间表达成k元d立方体中的维数（d）及节点数目的函数。在消息m大小为512字节，t_s =50.0μs，$t_h = t_w$ = 0.5μs（对超立方体）时，分别对$p = 256$，$p = 512$以及$p = 1024$求出通信时间与维数的函数表达式。对于上面给出的p和m，在给定成本的条件下，维数多大时性能最好？ 83

2.29 若选用存储转发路由选择，对k元d立方体重做习题2.28。 84

第3章　并行算法设计原则

算法设计是借助计算机解决问题的一个关键部分。串行算法实质上是用串行计算机解决问题的方法或基本步骤序列。与此类似，并行算法讲述的是怎样用多处理器解决问题。然而，设计并行算法远不仅仅是详细说明这些步骤。至少，一个并行算法增加了并发性的维，设计者必须指定能同时执行的步骤集。为了从使用并行计算机获得性能上的好处，这是必不可少的。在实践中，设计一个非平凡的并行算法包括下面某些步骤或者所有的步骤：

- 识别能并发执行的任务部分。
- 映射各并发任务块到并行运行的多处理器上。
- 分布与程序有关的输入、输出和中间数据。
- 管理对由多处理器共享的数据的访问。
- 在并行程序执行的各个阶段对处理器进行同步。

上面的每个步骤都有几种选择，但通常只有很少的选择组合所得到的并行算法，能够产生与要解决问题所用的计算和存储资源相匹配的性能。一般地，在不同并行体系结构或不同并行编程模式中要取得最佳性能，要使用不同的选择组合。

在本章中，将系统地讨论设计和实现并行算法的过程。我们假定提供并行算法或程序的完整描述是程序员或算法设计者的责任。在目前技术水平下，自动并行化的工具和编译器只能在高度结构化的程序或部分程序中良好地运行。因此，本章及本书的其他部分都不讨论有关这方面的内容。

3.1　预备知识

将计算划分成许多小的计算，再把它们分配到不同处理器中以便并行执行，这是并行算法设计中的两个关键步骤。在给出一些基本术语定义之后，将通过使用矩阵向量相乘和数据库查询处理这两个实例，来介绍并行算法设计的这两个关键步骤。

3.1.1　分解、任务与依赖图

把一个计算分为很多小的部分，其中的一些或所有部分都可能被并行执行，该划分过程称为分解（decomposition）。任务（task）是程序员定义的计算单元，其中为主要计算通过分解得到的划分。减少解决整个问题所需时间的关键是多任务同时执行。任务可以是任意大小的，但是一旦定义好，就被认为是不可再划分的计算单元。由问题分解出的任务大小可能并不相同。

例3.1　稠密矩阵-向量相乘

考虑一个$n \times n$稠密矩阵A与向量b的相乘，得到向量y。y的第i个元素$y[i]$是矩阵A的第i行与向量b的点积，即$y[i] = \sum_{j=1}^{n} A[i, j] \cdot b[j]$。如图3-1所示，对每一个$y[i]$的计算可作为一个任务。另外，如图3-4所示，计算也可分解成较少的任务，如4个任务，这样每一个任务计算的仅是

向量y的n/4部分。 ■

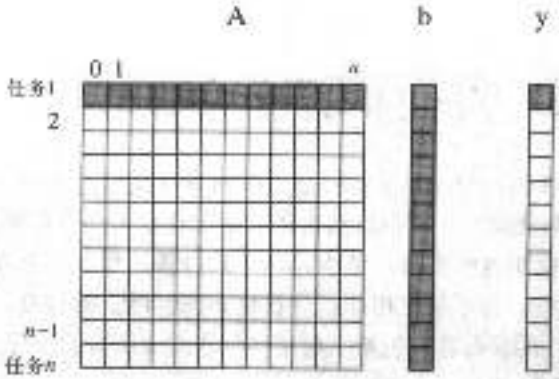


图3-1 分解稠密矩阵-向量相乘为n个任务，其中n为矩阵的行数。由任务1访问的矩阵和输入输出向量的部分在图中用高灰度显示

注意，图3-1中的所有任务都是独立的，可同时执行，也可以按任意顺序执行。然而，通常情况下，有一些任务可能需要使用别的任务所产生的数据，这样就要等到这些数据产生后再执行。我们用任务依赖图（task-dependency graph）抽象表示任务间的依赖关系和任务的执行次序关系。任务依赖图是一种有向无环图，其中的节点表示任务，有向边就代表节点间的依赖性。对应于一个节点的任务，只有当与此节点相连的输入边相连的所有任务都执行完毕
86 才能执行。注意，任务依赖图可以是不连接的，它的边集也可能是空的。矩阵-向量相乘就属于这种情况，每一个任务都计算乘积项的一个子集。下面的数据库查询处理是一个更加有趣的任务依赖图的例子。

例3.2 数据库查询处理

表3-1显示汽车的一个关系型数据库，表中的每一行对应于一种品牌汽车的数据记录，如ID、model（型号）、year（年份）、color（颜色）等。考虑处理下面查询时的计算：

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

这个查询寻找所有的2001的Civics型汽车，颜色是Green（绿色）或White（白色）。在关
87 系型数据库中，这种查询通过建立许多中间表来完成。一种可行的方法是，首先建立下面4张表：Civics汽车表、2001年汽车表、绿色汽车表以及白色汽车表。下一步，通过成对地求交集或并集求出每两张表的组合。如通过计算Civic表和2001年表的交集得到一张2001-Civics表。同样，通过计算绿色表和白色汽车表的并集得到一张汽车颜色为绿色或者白色的所有汽车表。最后计算这两张表的交集就能得到需要的表。 ■

在例3.2中，处理查询时用到的多种计算可以形象地显示在图3-2的任务依赖图中，图中的每一个节点是一个任务，这些任务对应的是需要通过计算求出的中间表，节点之间的箭头表示任务间的依赖性。例如，在计算对应于2001-Civics的表之前，必须首先计算Civics表和2001年表。

表3-1 存储使用汽车信息的一个数据库

ID#	Model（型号）	Year（年份）	Color（颜色）	Dealer（销售商）	Price（价格）
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

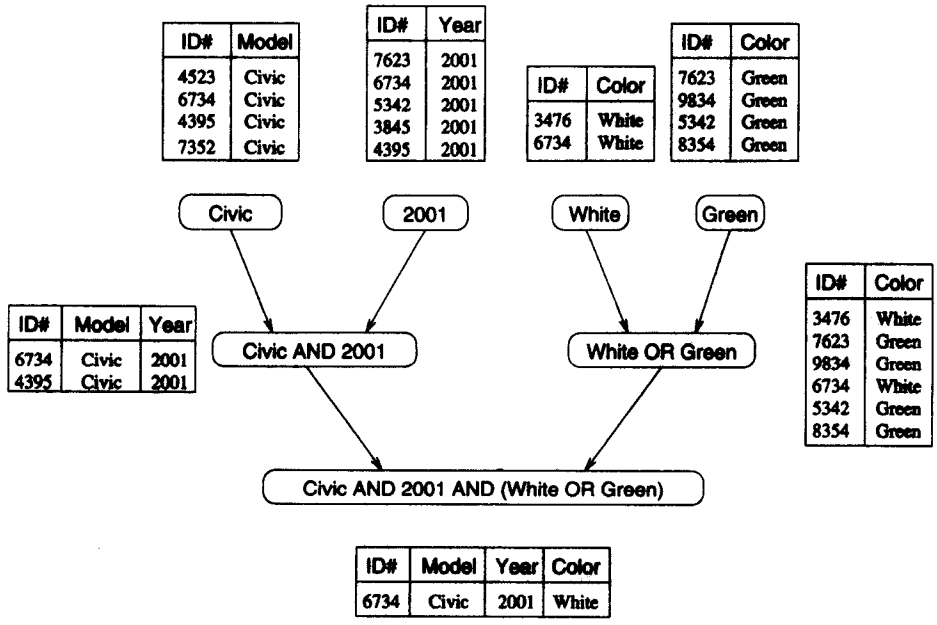


图3-2 数据查询操作中不同的表及它们之间的依赖性

注意，表示特定的计算可采用多种方法，尤其是那些包含结合操作符的计算，如加法、乘法、逻辑与以及逻辑或。不同方式的计算得到具有不同特征的任务依赖图。比如，在例3.2 88
的数据查询中，可以首先计算一张绿色汽车或白色汽车表，然后计算一张绿色汽车或白色汽车表和2001（年份）表的交集表，最后再计算此交集表与Civic表的交集。这个计算序列得到的任务依赖图如图3-3所示。

3.1.2 粒度、并发性与任务交互

分解问题得到的任务数量和大小决定了分解的*粒度*（granularity）。将任务分解成大量的细小任务称为*细粒度*（fine-grained），而将任务分解成少量的大任务称为*粗粒度*（coarse-grained）。例如，图3-1所示的矩阵-向量相乘就是细粒度分解，因为很大数量的任务执行一个

简单的点积操作。图3-4中的分解则属于粗粒度分解，它将同样问题分解为4个任务，每一个任务计算n维输出向量的$n/4$。

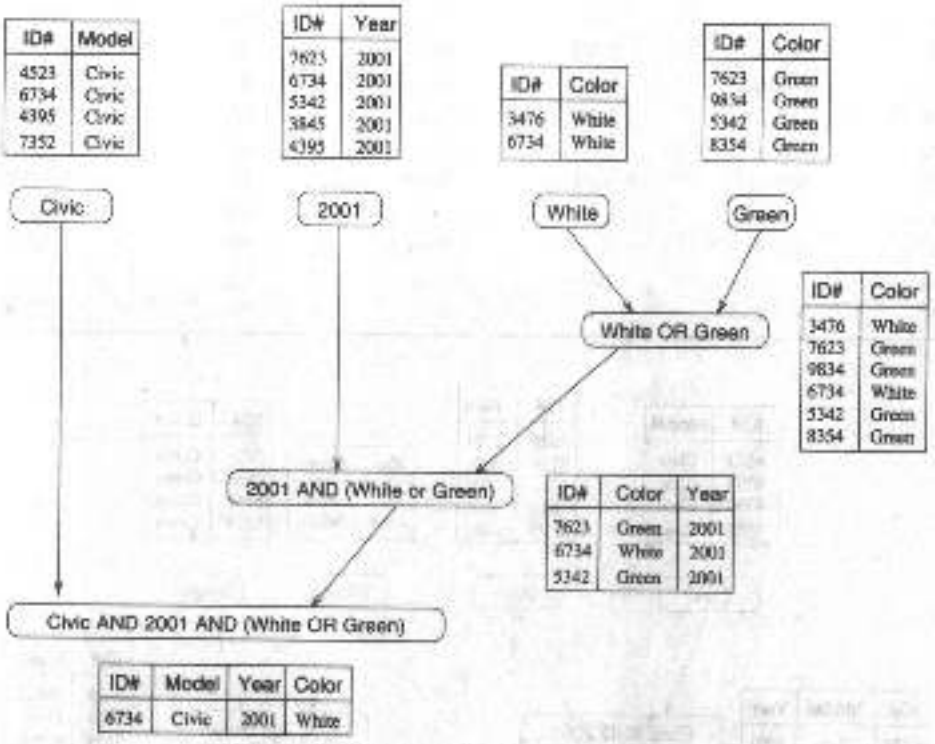


图3-3 数据查询处理操作的另一数据依赖图

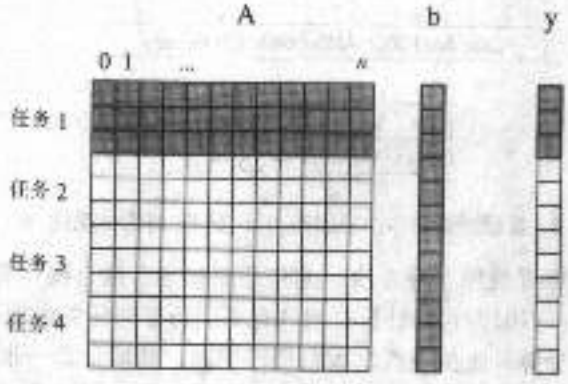


图3-4 稠密矩阵-向量的相乘分解为4个任务。由任务1访问的矩阵和输入输出向量部分在图中用高灰度显示

89 并发度（degree of concurrency）是与粒度相关的概念。并行程序中，任意时刻可同时执行的最大任务数称为最大并发度。由于任务间的相互依赖性，在大部分情况下，最大并发度总是小于总的任务数量。例如，图3-2和图3-3中的最大并发度是4。在这些任务图中，同时计算出Civics、2001、Green以及White 4张表时，就可获得最大并发度。一般情况下，对于树形任务的依赖图，最大并发度总是等于此树的叶子数。

平均并发度是与并行程序性能有关的一个更有用的指标，它是程序执行的整个过程中能并发运行的任务平均数。

最大和平均并发度通常都会随着划分的任务粒度的减小（变细）而增加。例如，图3-1中的矩阵-向量相乘分解具有很小的粒度和很大的并发度。图3-4中的同样问题则具有更大的粒度和更小的并发度。

并发度也依赖于任务依赖图的形状，通常，同样的粒度并不能确保同样的并发度。例如，考虑图3-5中的两个任务图，它们是任务图3-2、图3-3的抽象（习题3.1）。每个节点上的数字代表完成相应任务所需要的工作量。在图3-5a的任务图中，平均并发度是2.33，但在图3-5b中，任务图的平均并发度是1.88（习题3.1），虽然这两个任务依赖图都是基于同一分解。

关键路径（critical path）是任务依赖图的另一个特性，它决定一个给定粒度的任务依赖图的平均并发度。在一个任务依赖图中，我们把没有输入边的节点称为起始节点，把没有输出边的节点称为终止节点，任何一对起始节点和终止节点之间的最长有向路径就是关键路径。关键路径上所有节点的权之和称为关键路径长度，其中，节点的权是相应任务有关的工作量或任务的大小。总工作量与关键路径长度的比就是平均并发度。因此较短的关键路径有利于达到较高的并发度。例如，在图3-5a所示的任务依赖图中，关键路径长度为27，而在图3-5b所示的任务依赖图中，关键路径长度为34。由于使用两种分解求解问题的总工作量分别是63和64，两个任务依赖图的平均并发度分别是2.33和1.88。 90

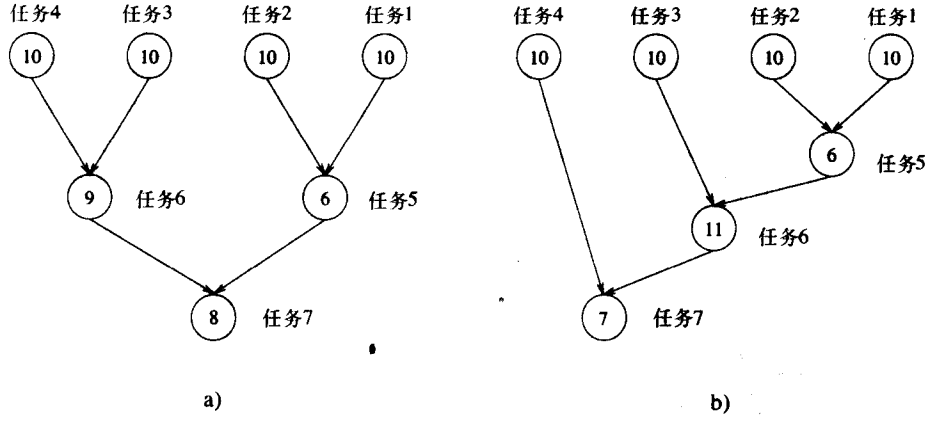


图3-5 对应任务图3-2和3-3的抽象图

虽然通过增加分解的粒度和利用所产生的并发度似乎可以并行执行越来越多的任务，以减少求解一个问题所需的时间，但实际上并非总是如此。通常对于某一问题，都有它自己固有的细粒度分解限度。例如，在图3-1的矩阵-向量相乘有n^2个乘法和加法，即使使用最小的细粒度分解，问题也不能被分解成多于$O(n^2)$个任务。

除了有限的粒度和并发度之外，还有一个重要因素，使我们不能从并行化获得不受限的加速比（串行执行时间与并行执行时间的比率）。此因素是运行在不同物理处理器上的任务之间的交互（interaction）。问题分解得到的任务通常要共享输入、输出或者中间数据。任务依赖图中的依赖性通常源自一个事实：某个任务的输出是另外一个任务的输入。例如，在数据查询的例子中，任务共享中间数据；一个任务产生的表被另外一个任务作为输入。根据对任务的定义以及并行编程模式，在任务依赖图中，表面上独立的任务间可能存在着相互交互。 91

例如，在矩阵-向量相乘的分解中，虽然所有的任务都相互独立，但它们都要访问整个输入向量b。由于初始时只有向量b的一个副本，在分布式存储模式中，任务仍必须通过发送和接受消息来访问向量b。

任务之间的交互方式通过所谓的任务交互图（task-interaction graph）来描述。任务交互图中的节点代表任务，边连接彼此交互的任务。如果任务交互图中，节点任务的计算量和节点间的交互量可知，那么节点和边就可以分配与这两个量成比例的权值。任务交互图中的边通常是无向的，但如果有向边是单向的，就可以用有向边指示数据的流向。任务交互图中的边集合通常是对应任务依赖图边集合的超集，在前面讨论的数据查询例子中，任务交互图与任务依赖图相同。下面给出一个更加有趣的任务交互图的例子，它从稀疏矩阵-向量相乘问题中得到。

例3.3 稀疏矩阵-向量相乘

考虑计算矩阵-向量相乘问题$y = Ab$，其中A是$n \times n$稀疏矩阵，b是$n \times 1$稠密向量。当一个矩阵中大部分的元素都是0，且0元素的分布没有规律，则此矩阵就是稀疏矩阵。通常，通过避免0元素参与运算，可以对稀疏矩阵的算术运算进行优化。例如，要计算向量y的第i个分量$y[i] = \sum'_{j=1}(A[i, j] \times b[j])$时，我们只需要计算当$A[i, j] \neq 0$时$A[i, j] \times b[j]$的积。例如，$y[0] = A[0, 0] \cdot b[0] + A[0, 1] \cdot b[1] + A[0, 4] \cdot b[4] + A[0, 8] \cdot b[8]$。

分解此计算的一种方法是划分结果向量y，并让每一个任务计算一个分量。图3-6a显示这种分解。除了把对输出向量中的元素$y[i]$的计算分配给任务i以外，也把任务i作为矩阵A的第i行$A[i, *]$及输入向量b的元素$b[i]$的“拥有者”。注意，对$y[i]$的计算要访问由其他任务拥有的向量b中的多个元素。这样，任务i必须从适当的位置获得这些元素。在消息传递模式中，作为$b[i]$的拥有者，任务i也继承发送$b[i]$到需要它的地方的责任。例如，任务4必须发送$b[4]$到任务0、5、8和任务9，并且必须获得$b[0]$，$b[5]$，$b[8]$和$b[9]$来执行自己的计算。得到的任务交互图显示在图3-6b中。 ■

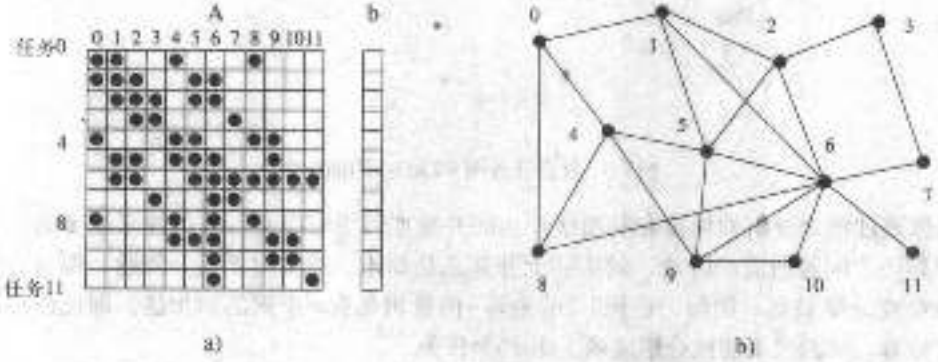


图3-6 稀疏矩阵-向量相乘的分解图和相应的任务交互图。
在分解中任务i计算$\sum_{0 \leq j < 11, A[i,j] \neq 0} A[i, j] \cdot b[j]$

92 任务的交互和有限的并行性，以及它们对并行算法结构组合的性能和扩展性的影响等都会造成开销。第5章中将对这些开销进行详细的定量分析。设计并行算法时必须仔细考虑这些因素，本节只对这些因素作了基本介绍。

3.1.3 进程和映射

问题分解后，得到的任务运行在物理处理器上。然而，由于下面将要讲到的一些原因，本章将使用进程(process) 来表示执行任务的处理代理或计算代理。本文中，术语“进程”并不严格等同于操作系统的进程定义。相反，进程只是一个抽象的实体，在并行程序激活任务后，它利用这个任务的代码和数据在有限的时间内产生输出结果。在此期间，进程不仅执行计算，在需要的时候还与其他进程进行同步或通信。为了获得对串行执行的加速，并行程序必须同时激活一些进程来运行不同的任务。给进程分派任务的机制称为映射（mapping）。例如，在例3.5的矩阵相乘中，4个进程的每一个都可以分配一个任务，用来计算矩阵C的一个子矩阵。

由对分解得到的任务依赖图和任务交互图，对于在并行算法中选择一个好的映射方法起着至关重要的作用。好的映射方法应该设法把相互独立的任务映射到不同的进程以获取最大并发度；好的映射方法应该确保可用进程来执行关键路径上的任务，以使得任务变成可执行的时候总计算时间最短；好的映射方法应该映射相互交互的平凡的任务到同一个进程以便最小化进程间的交互。但是，对于大多数非平凡并行算法来说，这些目标是相互冲突的。例如，最有效的分解-映射组合方法是把单个任务映射到单个进程上。这样就不会在空闲或交互中浪费时间，但同时就不能获得加速。对于一个成功的并行算法，找到整体并行性能间的最佳平衡是关键。因此，映射任务到进程的方案对提高并行算法的效率起着重要的作用。即使并发度由分解决定，但是映射方案决定了实际可用的并发度和并行效率。 93

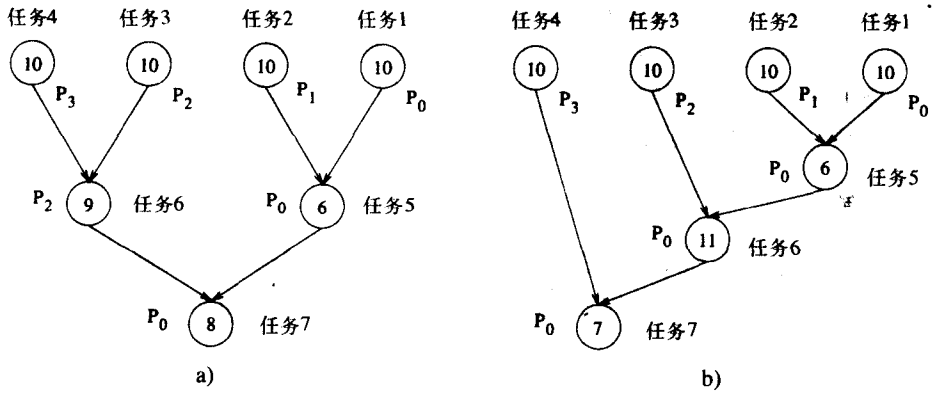


图3-7 将图3-5中的任务图映射到4个进程

例如，对于图3-5中的分解和任务交互图，图3-7显示映射它们到4个进程的有效映射。注意，这种情况下，虽然总的任务数是7个，但最多可以利用这4个进程。最后3个任务可以映射到任何进程上，都能满足任务依赖图的限制。然而，更有意义的映射是把由边相连的任务映射到同一个进程上，因为这样才能防止任务间的交互转变为进程间的交互。例如，在图3-7b中，如果任务5映射到进程P_2，那进程P_0和P_1都需要与P_2交互。在目前的映射方案中，只有进程P_0和P_1之间的一次交互。

3.1.4 进程与处理器

在讲述并行算法设计的文字中，进程是执行任务的逻辑计算代理。处理器是实际执行计

算的硬件部件。这本书中，我们选用术语进程来表示并行算法和程序。在多数情况下，关于并行算法设计的文字中提到进程时，进程和处理器间都有一一对应的关系，并且可以假定进程的数量和并行计算机上的CPU的数量一样多。然而，有时候可能要从更高的抽象层次来表
94 达某一个并行算法，尤其当复杂算法具有多阶段或多种并行性形式的时候。

为硬件设计支持多种编程模式的并行程序时，分别处理进程和处理器也是很有用的。例如，若一台并行计算机由许多计算节点组成，这些节点之间通过消息传递进行通信，那么每个节点都是具有多CPU的共享地址空间模块。考虑在这样的并行计算机上实现矩阵乘法。设计一个并行算法的最好方法就是分两个阶段完成。首先，开发适合消息传递模式的分解和映射策略，并使用这种策略来开发节点间的并行机制。原始矩阵相乘问题分解成的每个任务本身是矩阵相乘计算。然后，开发适合共享存储模式的分解和映射策略，并使用此策略在单个节点的多CPU上实现每个任务。

3.2 分解技术

如前面所述，并行求解问题的基本步骤就是将被执行的计算划分成一组按照任务依赖图并行执行的任务。本节将讨论一些为获得并发性而经常使用的分解技术，但并不列举所有可能的分解技术。而且，对某个给定的问题，某种分解技术并不总能得到它的最好并行算法。虽然有这些缺陷，但本节讨论的分解技术通常是解决许多问题的很好的切入点，将这些技术加以组合，能得到对许多问题的有效分解。

这些技术可概括分类为*递归分解*（recursive decomposition）、*数据分解*（data decomposition）、*探测性分解*（exploratory decomposition）以及*推测性分解*（speculative decomposition）。递归分解和数据分解技术是比较通用的，因为它们能适用于大量的各种问题。探测性和推测性分解技术是专用性质的，因为它们只适用特定类型的问题。

3.2.1 递归分解

递归分解采用分而治之的策略使问题可并行执行。这种策略首先划分问题为一组独立的子问题，子问题又可以采用相似的划分得到更小的子问题。分治策略能得到自然的并发性，因为不同子问题可并发求解。

例3.4 快速排序

95 通常使用快速排序算法对包含n个元素的序列A进行排序。快速排序使用分治算法，首先指定其中一个元素x为主元，然后划分序列A为两个子序列A_0和A_1，使A_0中的元素都小于x，而A_1中的元素都大于等于x。这个划分步骤就是分治算法中的分开步骤。每一个子序列A_0和A_1再递归调用快速排序，每一个递归调用再进行划分。这个过程在图3-8中用12个数字的一个序列说明。当每一个子序列中只有一个元素时递归结束。 ■

在图3-8中，我们定义一个任务为划分一个给定序列的操作。因此图3-8也代表问题的任务图。起初，仅有一个序列（即树的根），因此只能使用一个处理器来划分它。根任务产生成两个子序列（A_0和A_1对应于树的第一层两个节点），每个子序列可并行划分。类似地，沿着树向下移动，并发性不断增加。

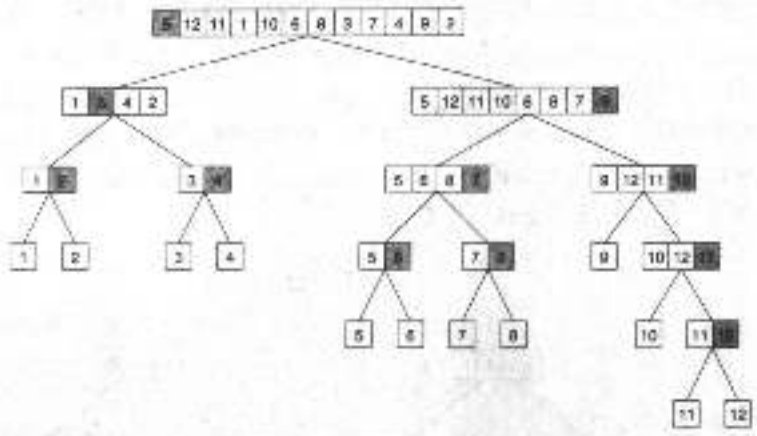


图3-8 对含有12个数字的序列排序时进行递归分解的快速排序任务依赖图

有时，即使常用于解决问题的算法没有采用分治策略，也可以重新构造计算，使之适用于递归分解。例如，在具有n个数字的数组A中查找最小元素。查找的串行算法就是遍历整个序列A，并在每一步中记下找到的最小元素，如算法3-1中的描述。容易看到此串行算法没有显示并发性。

算法3-1 在含有n 个数字的数组A 中查找最小数的串行程序

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

算法3-2 在含有n 个数字的数组A 中查找最小数的递归程序

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

96 将此计算重新构造为分治算法，就能使用递归分解而得到并发性。算法3-2是在一个数组中查找最小元素的分治算法。该算法首先划分数组A为两个子序列，每个子序列中的元素个数为$n/2$，在每一子序列中分别递归调用最小查找。这样数组A中的最小元素就是两个子序列中最小元素值更小的。当每一个子序列中只有一个元素时，递归结束。用这种方式重构串行算法，很容易构造问题的任务依赖图。图3-9显示这样的任务依赖图，用来查找8个数字中的最小数，其中每一任务只负责查找两个数中较小的一个。

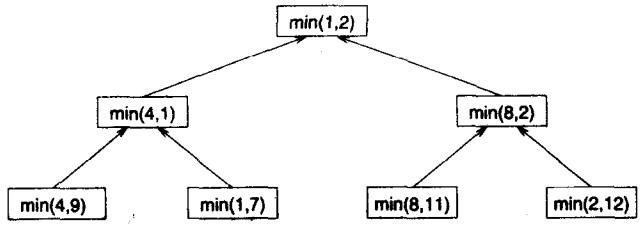


图3-9 在数组{4，9，1，7，8，11，2，12}中查找最小元素的任务依赖图。树中的每一节点表示查找一对数字中的较小数字的任务

3.2.2 数据分解

对于运行在大数据结构中的算法，要在算法中得到并发性，数据分解是常用的有效方法。
97 在这种方法中，计算的分解分两步完成。第一步对计算中的数据进行划分，第二步，在数据划分的基础上推导从计算到任务的划分。通常，对不同数据划分任务执行的操作是相似的（如例3.5中的矩阵相乘），或者选自一个小的操作集（如例3.10中的LU分解）。

下面将讨论各种可能的不同数据划分方法。一般而言，我们要探究并评估各种可行的数据划分方法，并从中找出最简单有效的对计算的分解。

划分输出数据 在许多计算中，每个输出元素都可作为输入元素的函数独立计算得到。在这样的计算中，输出数据的一个划分自动地将问题分解为多个任务，每一个任务负责计算一部分输出数据。例3.5有关矩阵相乘的问题说明以划分输出数据为基础的分解。

例3.5 矩阵相乘

考虑将两个$n \times n$矩阵A和B相乘得到矩阵C。图3-10表示，问题被分解为4个任务。每一个矩阵被认为由4个子矩阵或块组成，这些子矩阵或块通过把矩阵的每一维分成两半来定义。然后，C的4个子矩阵（每个大小约为$n/2 \times n/2$），就由4个任务独立地计算出，作为A和B的子矩阵的相应乘积的和。 ■

大部分矩阵算法，包括矩阵-向量相乘和矩阵-矩阵相乘，都能表示为矩阵块的操作。在这种表示方法中，矩阵被看成由块或子矩阵组成，对矩阵元素的标量算术运算由等价的对块的矩阵运算替换。这两种方法的运算结果相同（习题3.10）。块形式的矩阵算法通常被用来辅
98 助分解。

图3-10所示的分解以输出矩阵C划分为4个子矩阵为基础，而4个任务中的每一个任务分别计算这些子矩阵。读者必须注意，数据分解明显不同于将计算分解为任务。虽然这两者通常是相关的，而且前者常常用来协助后者，但一个数据分解并非只有一种任务分解。例如，图

3-11显示矩阵相乘的其他两种任务分解，每一种包含8个任务，对应于图3-10a中的同一个数据分解。

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

a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

b)

图3-10 矩阵相乘：a）划分输入和输出矩阵为2 × 2的子矩阵；b）以图a）中的矩阵划分为基础将矩阵相乘分解为4个任务

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

图3-11 分解矩阵相乘为8个任务的两个例子

下面引入另外两个例子说明以数据划分为基础的分解。例3.6描述事务数据库中，计算一个项目集出现的频率，这个问题也能以输出数据划分为基础进行分解。 99

例3.6 计算事务数据库中一个项目集出现的频率

考虑计算事务数据库中一个项目集的出现频率。在这个问题中，给定一个含n个事务的集合T和含m个项目集的集合I。每个事务和项目集都包含来自一个可能项目集中的少量项目。例如，T可能是一个食品店顾客销售数据库，每一事务是一个购物者的个人商品列表，每一项目集可能是商店中的一组商品集。如果商店想知道有多少顾客购买了指定商品集中的每一种商品，那么就要计算I中的每一项目集在所有事务中出现的次数之和；即每个项目集是事务的子集的事务数目。图3-12a显示这种计算。图3-12中的数据库由10个事务组成，而我们对计算表中第2列8个项目集出现的频率感兴趣。数据库中这些项目集的实际频率，即频率计算程序的输出，显示在第3列中。例如，项目集{D，K}出现两次，一次在第2个事务中，一次在第9个事务中。 ■ 100

图3-12b说明如何通过把输出划分为两部分，并让每个任务计算频率的一半，将项目集频率的计算分解成两个任务。在此过程中，注意项目集输入也被划分，但图3-12b中分解的首要目的是让每一个任务独立计算指定给它频率的子集。



a）事务（输入）、项目集（输入）和频率（输出）



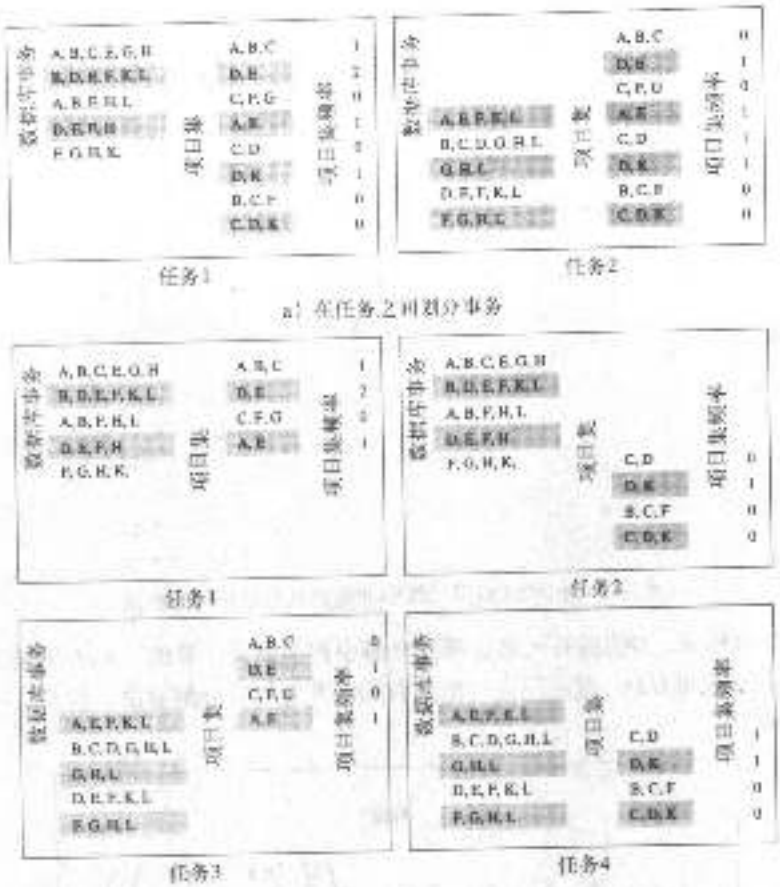
b）划分任务之间的频率（和项目集）

图3-12 计算事务数据库中项目集出现的频率

划分输入数据 只有当每一个输出结果都能作为输入的函数自然地计算时，才能划分输出数据。在许多算法中，不可能或不要求划分输出数据。例如，查找数据集的最小值、最大值或求和，输出仅仅是一个单独的未知值，在排序算法中，每一输出元素不能孤立地确定。在这些情况下，很可能要划分输入数据，然后再用这种划分导出并发性。对输入数据的每一个划分创建一个任务，而这个任务尽量利用本地数据执行尽可能多的计算。注意，由输入数据划分导出的任务解决方案可能并不能直接解决原始问题。这种情况下还需要对结果进行归并的计算。例如，用p个进程计算$N(N>p)$个数序列的和，就可以把输入划分为p个数量大致相等的子集。每一任务计算一个子集的和，最后，这p个部分和相加就得到最终结果。

例3.6描述的，计算事务数据库中一个项目集出现的频率计算问题也能通过输入数据的划分来分解。图3-13a显示以输入事务的划分为基础的分解。两个任务分别计算所有项目集在其事务子集中出现的频率。这两个频率集是两个任务独立的中间结果。通过成对相加组合中间结果就得到最终值。

划分输入和输出数据 在可能对输出数据进行划分的情况下，对输入数据的划分可能导致附加的并发性。例如，在图3-13b显示的项目集频率计算的4路分解中，事务集和频率集被分解为两部分，4种不同组合的每一个被分配给4个任务中的一个，然后，每个任务计算频率的本地集。最后，把任务1和任务3的输出加在一起，任务2和任务4的输出加在一起。



a）在任务之间划分事务

b）在任务之间划分事务和频率

图3-13 计算事务数据库中项目集频率的一些分解方法

划分中间结果数据 有些算法通常可以组成多级计算结构，其中某一级的输出是下一级输入。对这种算法的分解，可以从划分算法中间级的输入或输出数据导出。划分中间数据有时比划分输入或输出数据获得更高的并发性。通常，在求解问题的串行算法中并不显式产生中间数据，而某些对原始算法的重构可能需要用中间数据划分导出分解。 101

我们重新用矩阵相乘的例子说明以划分中间数据为基础的分解。如图3-10和图3-11所示，回忆以输出矩阵C的2×2划分为基础的分解的最大并发度为4。我们可以通过引入一个中间过程来增加并发度，如图3-14所示，在中间一级的8个任务分别计算乘积子矩阵，并把结果存储到一个临时三维矩阵D中，子矩阵$D_{k,i,j}$是$A_{i,k}$和$B_{k,j}$的乘积。

对中间矩阵D的划分导出8个任务的分解，如图3-15所示。乘法计算结束后，通过耗时相对较少的矩阵加法步骤计算结果矩阵C。对第2维和第3维i和j相同的所有子矩阵$D_{*,i,j}$相加得到$C_{i,j}$。图3-15中任务1到8的任务执行的操作为$O(n^3/8)$，每个任务对矩阵A和B的$n/2 \times n/2$子矩阵作乘法。然后，任务9到12的任务每个用时$O(n^2/4)$，对中间矩阵D的相应$n/2 \times n/2$子矩阵相加得到最后的结果矩阵C。图3-16显示了图3-15中的分解对应的任务依赖图。

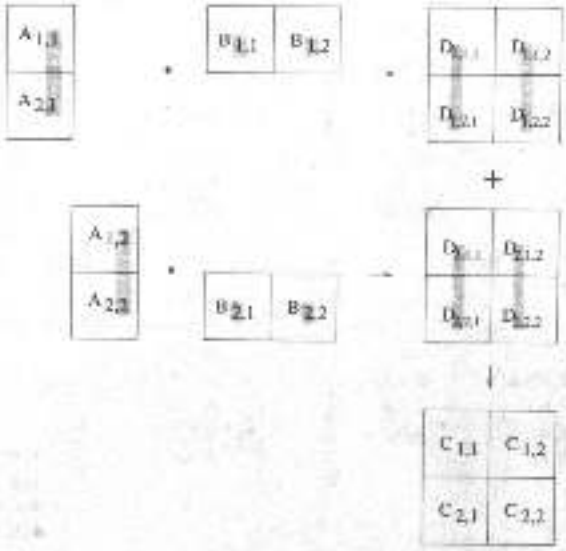


图3-14 矩阵A和B用三维中间矩阵D的划分时的相乘

注意如图3-11所示，D的所有元素在原始分解中都是隐式计算的，也不用显式存储。通过重构原始算法并显式保存D，就可以设计出具有更高并发性的分解算法。但是，这个算法要用到额外的聚合内存。

步骤I

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

步骤II

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

由D的划分导出的分解

Task 01: $D_{1,1,1} = A_{1,1}B_{1,1}$
Task 02: $D_{2,1,1} = A_{1,2}B_{2,1}$
Task 03: $D_{1,1,2} = A_{1,1}B_{1,2}$
Task 04: $D_{1,1,2} = A_{1,2}B_{2,2}$
Task 05: $D_{1,2,1} = A_{2,1}B_{1,1}$
Task 06: $D_{2,2,1} = A_{2,2}B_{2,1}$
Task 07: $D_{1,2,2} = A_{2,1}B_{1,2}$
Task 08: $D_{2,2,2} = A_{2,2}B_{2,2}$
Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$
Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$
Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$
Task 12: $C_{2,2} = D_{1,2,2} + D_{1,2,2}$

图3-15 矩阵相乘基于划分中间三维矩阵的分解

拥有者-计算规则 以划分输入或输出数据为基础的分解也常称为拥有者-计算（owner-compute）规则。这个规则的思想是，每一个划分都执行涉及它拥有的数据的所有计算。根据数据的性质或数据划分的类型，拥有者-计算规则可能具有不同的含义。例如，当把输入数据的划分分配给任务时，拥有者-计算规则意味着一个任务要执行能用这些数据完成的所有计算。另一方面，如果划分输出数据，那么拥有者-计算规则表明一个任务要计算分给它的所有分区中的数据。

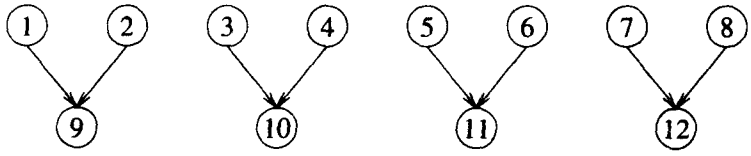


图3-16 图3-15所示分解的任务依赖图

102 ~ 104

3.2.3 探测性分解

有些问题的基本计算对应于解空间的一次搜索，探测性分解就用来分解这样的问题。在探测性分解中，划分搜索空间为更小的部分，然后并发搜索这些小的部分，直至找出希望的解。考察15迷宫问题作为探测性分解的例子。

例3.7 15迷宫问题

15迷宫问题由4 × 4网格组成，其中15个格的标号从1到15，另一个为空格。与空格邻接的一格可以移进这个空格，而原来位置成为空格。根据网格布局，最多有4个可移动的格：上、下、左、右。网格的初始和最终布局事先给定。目标是找到从初始布局到最终布局的任意移动序列或最短移动序列。图3-17展示一个从初始布局到最终布局的序列移动过程。 ■

通常，人们用搜索树技术解决15迷宫问题。从初始布局开始，产生所有可能的后继布局。一种布局可能有2、3、4种后继布局，每一种对应于一个相邻格移进空格。寻找一条从初始布局到最终布局的路径的任务变成寻找一条从这些新产生的布局之一到最终布局的路径。由于其中一个新产生的布局必定向最终布局靠近了一步（如果解存在），我们已经向求解靠近了一步。由搜索树产生的布局空间通常称为状态空间图。图中每个节点表示一种布局，图中每条边连接着两个布局，通过移动一格可以从一个布局到达另一个。

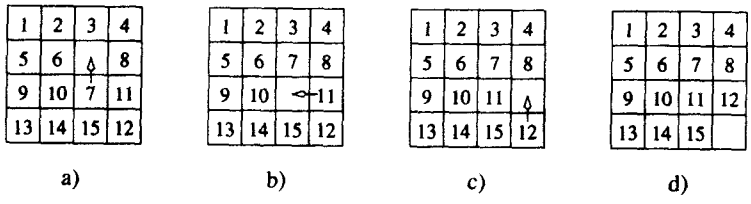


图3-17 15迷宫问题实例，显示起始布局a)、最终布局d）以及从起始布局到最终布局的移动序列

105

下面讲述求解这个问题的并行方法。首先，从初始布局依次产生少数层次的布局，直到搜索树有足够的叶节点（也就是15迷宫问题的布局）。然后每一节点分派一个任务进行进一步查找，直到其中至少一个找到一个解。只要一个并发任务找到一个解，就可以通知其他任务停止查找。图3-18显示分解为4个任务时的查找，其中任务4找到解。

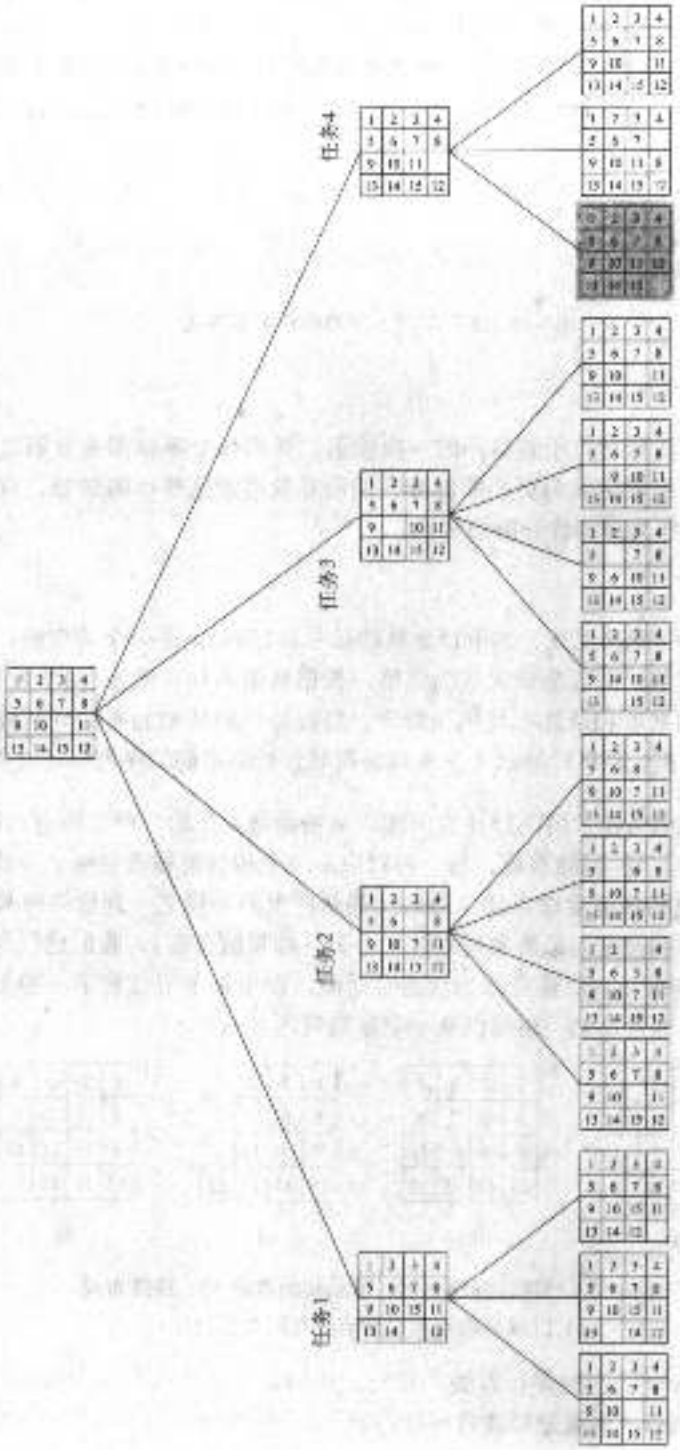


图3-18 一个15迷宫问题实例产生的状态

注意，即使探测性分解表面上与数据分解相似（搜索空间可认为是被划分的数据），但它们有下述本质上的不同。数据分解导出的任务都完全地执行，每一任务执行的计算都是最终解的一部分。在另一方面，探测性分解中只要一个任务找到答案，其他未完成任务就可以终止。因此，并行形式执行的一部分搜索（以及执行的累计操作量）与串行算法的搜索是完全不同的。并行形式执行的操作既可以少于也可以多于串行算法执行的操作。例如，图3-19中所示的搜索空间被划分为4个并发任务。假如解位于与任务3对应的搜索空间的开始处（图3-19a），那么并行形式几乎立刻就找到答案。串行算法要在执行完对应于任务1和2搜索整个空间后才能找到解。另一方面，假如解位于与任务1对应的搜索空间的末端（图3-19b），那么并行形式就要执行几乎4倍于串行算法的操作，也就不会产生加速比。

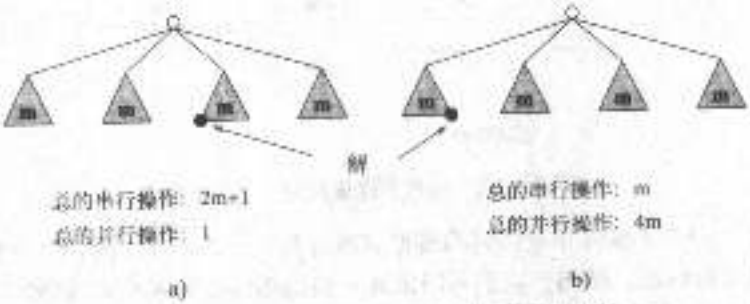


图3-19 由探测性分解得到的异常加速比的实例

3.2.4 推测性分解

有时程序会遇到多个可能的对计算很重要的分支，选择某个分支又取决于之前的其他计算的结果。此时就要用到推测性分解。这种情况下，当一个任务正在执行计算，而这个计算结果将决定下一步计算，其他任务就可以并发地执行下一步计算。这种情况类似于C语言中在得到*switch*语句的输入前，并行计算*switch*语句的一个或多个分支。当一个任务进行最终决定转向分支的计算时，其他任务可以并行地选取*switch*的多个分支。当完成对*switch*输入的计算时，与正确分支对应的计算将被利用，而对应的其他分支则被丢弃。通过考虑估计下一个任务依赖的条件所需的时间，由于利用了决定转向分支的计算时间去进行下一步的并行计算，并行计算时间要少于串行计算时间。然而，*switch*的这种并行形式肯定有一些计算浪费。为减少这种浪费，可使用一种稍微不同的推测性分解，在*switch*的一种结果更可能出现的情况下，这种方法最适合。在这种情况下，只有最可能发生的分支占用一个任务与前面的计算并行执行。如果*switch*的结果与推测的不同，那么就回滚并采用*switch*的正确分支。

106 ↓ 107

如果有多个推测性阶段，那么由推测性分解带来的加速比就可以累加起来。离散事件模拟（discrete event simulation）就使用推测性分解。对离散事件模拟的详细描述超出本章的范围；但是，我们在下面给出问题的一个简化描述。

例3.8 并行离散事件模拟

考虑对表示成网络或有向图系统的模拟。网络的节点代表组件。每一个组件有作业的一个输入缓冲区。每一组件或节点的起始状态是空闲的。如果队列中有作业，空闲组件从输入队列中选择一个作业，在有限的时间中处理该作业，并把它放到由输出边与其相连的组件输

入缓冲区中。如果输出的相邻组件之一的输入缓冲区已满，则组件就必须等待，直到相邻组件选取一个作业执行，从而在缓冲区中创建空间。输入作业的类型是有限的。一个组件的输出（以及与之相连组件的输入）和处理作业的时间是输入作业的函数。现在的问题是要对一个或一组输入作业队列模拟网络的功能，并计算总的完成时间和其他可能的系统行为。图3-20显示一个解决离散事件问题的简单网络。 ■

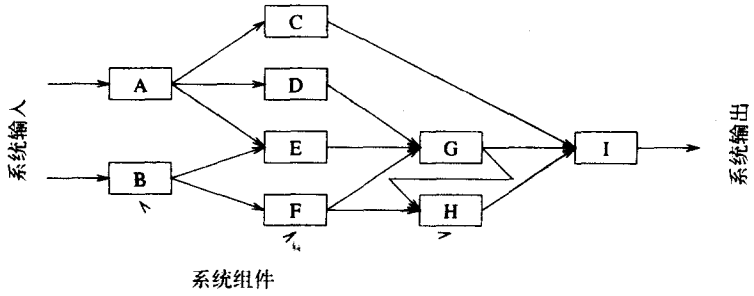


图3-20 用于离散事件模拟的一个简单网络

在例3.8中，网络上输入作业队列的模拟问题有着固有的顺序，因为一个典型组件的输入是另外一个组件的输出。然而，我们可以定义一些推测性任务来模拟网络的一部分，其中每
108 一部分都假设有那个阶段的几种可能输入之一。当某一阶段的实际输入得到后（作为前一阶段另一选择器任务组件的结果），如果推测是正确的，那么模拟这个输入需要的全部或部分工作已经完成；否则如果推测是错误的，那么就要用最近的正确输入重新对这个阶段进行模拟。

推测性分解与探测性分解有如下不同。推测性分解中导向各并行任务的一个分支输入是未知的，而在探测性分解中，源于一个分支的多任务输出是未知的。在推测性分解情况下，串行算法严格地执行一个推测阶段的单个任务，因为当到达此阶段的开始处时，已经确切知道应该执行哪一个分支。因此，通过预先计算只有一个实际被执行的多个可能的任务，采用推测性分解的并行程序要比相应的串行程序完成更多的累积工作。即使只是推测性地探测多个可能性中的一个，与串行算法相比，并行算法也要完成更多或一样多的工作量。另一方面，在探测性分解情况下，串行算法也可能探索不同的选择，因为在开始时可能通向结果的分支是未知的。所以，根据结果在搜索空间中的位置，与串行算法相比，并行算法可能执行更多、更少或同样的总工作量。

3.2.5 混合分解

前面已经讨论了许多分解方法，它们能用于导出许多算法的并发形式。这些分解方法并不是相互排斥的，往往可以组合应用。通常，计算由多个阶段构成，有时在不同阶段要使用不同的分解方法。例如，查询具有n个元素的大集合中的最小元素时，采用纯递归分解可能得到的任务数要远远多于可用的进程数P。有效的分解方法是把输入数据划分为P个大致相等的
109 部分，再让每一任务计算分配给它的序列的最小值，最后可使用图3-21所示的递归分解找到P个中间结果的最小值。

作为混合分解的另外一个应用的例子，考虑并行快速排序。例3.4中使用递归分解得到了快速排序的并发形式。对n元素的序列进行排序时，使用这种形式将导致$O(n)$个任务。但由于

这些任务间相互依赖性以及任务大小不一，有效的并发性相当有限。例如，第一个任务将输入划分为两个部分花费$O(n)$时间，这是使用并行性可获得性能的上界。但在并行快速排序中，任务执行的划分列表的步骤也可采用9.4.1节讨论的输入分解技术。递归分解与输入数据分解相组合能产生快速排序的高并发形式。

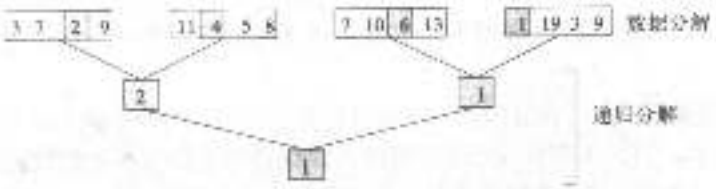


图3-21 使用4个任务查找含有16个元素数组的最小数的混合分解

3.3 任务和交互的特点

前一节讲述的各种分解技术，使我们能确定某个问题可获得的并发性，并把问题分解为可并行执行的多个任务。设计并行算法过程中的下一步就是取出这些任务，并把它们分配（即映射）给可用的处理器。当设计映射方案来构造好的并行算法时，常常能从分解中得到提示。任务的性质和任务之间的交互对映射有影响。本节将讨论对选择好的映射方案有影响的各种任务特性及任务间的交互特性。

3.3.1 任务特性

下面4个任务特性对映射方案的适用性有着重要影响。

任务产生 组成并行算法的任务既可以静态产生，也可动态产生。如果所有任务在算法开始执行前是已知的，这种情况就属于静态任务产生（static task generation）。数据分解通常导致静态任务产生，矩阵相乘和LU因子分解（习题3.5）就是数据分解导致静态任务产生的例子。递归分解也能得到静态任务依赖图，查找列表的最小元素（图3-9）是静态递归任务依赖图的例子。 110

执行算法时，某些分解导致动态任务产生（dynamic task generation）。在这样的分解中，实际任务和任务依赖图不能预先获得，虽然作为算法的一部分支配任务产生的高层规则或准则已知。递归分解也能导致动态任务产生。例如，考虑快速排序的递归分解（图3-8），任务是动态产生的，任务树的大小和形状由待排序的输入数组中的值决定。同样大小的数组可能导致不同形状的任务依赖图及不等的任务数量。

探测性分解既能静态地产生任务，也能动态地产生任务。例如，考虑3.2.3节讨论的15迷宫问题中。使用探测性分解产生静态任务依赖图的方法如下：首先，一个预处理任务从最初的布局开始，并按宽度优先方式扩展搜索树，直到预定数目的布局产生为止。此时，这些布局代表独立的任务，它们能映射到不同进程并独立地运行。在另一种动态产生任务的分解中，任务以一种状态作为输入，并以宽度优先方式扩展预定的步数，然后产生新的任务，对每个出现的状态进行同样的计算（直到找到解后算法终止）。

任务大小 任务大小指用来完成任务所需的相对时间。映射方案的复杂性常常取决于各个任务是否均匀，也就是说，是否需要大致相同的时间来执行这些任务。如果各个任务的执行

时间相差很大，就说它们是非均匀的。例如，图3-10和3-11中所示的矩阵相乘分解的任务就是均匀的。相反，图3-8中快速排序的任务则是非均匀的。

任务大小的知识 影响映射方案选择的第三个特点是任务大小是否为已知。假如所有任务大小是已知的，那么将任务映射到进程时就可用到这个信息。例如，到目前为止讨论的各种矩阵相乘的分解中，每一个任务的计算时间在并行程序开始运行前是已知的。相反，在15迷宫问题中，一个典型任务的大小是未知的。我们事先不知道从某一给定布局到求出解要移动多少步。

111 **与任务相关的数据大小** 任务的另一个重要特点是与它相关的数据大小。它是映射时要考虑的一个重要因素，因为进程执行这个任务时必须获得与此任务相关的数据，这些数据的大小和位置可以决定能够执行任务的进程是否会导致过多的数据移动开销。

与任务相关的不同数据类型，其大小也可能不同。例如，输入数据可能很小，而输出数据却很大，或者反过来。例如15迷宫问题中任务的输入只是问题的一个状态，相对于从初始状态到最终状态找出移动序列的总计算量而言，这仅仅是一个很小的输入。在计算序列最小元素的问题中，输入的大小与计算的大小成比例，而输出仅仅是一个数。在快速排序的并行计算中，输入和输出数据的大小与求解任务所需的串行运行时间是同阶的。

3.3.2 任务间交互的特征

在任何非平凡并行算法中，各个任务需要相互交互，以便共享数据、中间值或同步信息。不同的并行算法需要并发任务间的不同类型交互。这些交互的性质使得这些任务更适合某种特定的编程模式和映射方案，而对于其他一些编程模式和映射方案却不那么适合。任务之间交互的类型可以从不同方面进行描述，每一种对应于所采用计算的一个独特特征。

静态与动态 考虑交互是否具有静态或动态模式是一种对并发任务间发生的交互进行分类的方法。静态交互模式是：对于每一个任务，它们之间的交互在预定的时间发生，并且在这些时间交互的任务集在算法执行之前是已知的。换句话说，静态交互模式中，不仅任务-交互图是预先知道的，而且交互发生的计算阶段也是预先知道的。动态交互模式则是指在算法执行前，交互的时间或交互的任务集不预先确定。

在消息传递模式中容易编写静态交互程序，而编写动态交互程序则很难。这是因为，消息传递交互需要用到交互的任务信息——发送者和接收者的信息。动态交互的不可预测性质使得发送者和接收者都很难同时参与交互。因此，在消息传递模式中用动态交互实现并行算法时，必需分派额外的同步或轮询职责给任务。使用共享地址空间编程可以方便地编写静态交互和动态交互程序。

本章前面提到的对并行矩阵相乘的分解体现了静态任务交互。15迷宫问题属于动态任务交互，求解的第一步对初始状态使用宽度优先搜索产生要求的状态数目，其后对各个任务被
112 分配不同的探测状态。很可能某一个状态会导致死胡同，而一个任务穷尽其搜索空间但没有达到目标状态，同时其他任务还在忙于搜寻解。搜索完路径的任务可以从另外的繁忙任务队列中选取一个未探索的状态进行探测。从一个任务到另一任务的工作转换中包括的交互是动态的。

有规则与无规则 另一个对交互进行分类的方法基于空间结构。如果一个交互模式有一些结构有利于有效实现，那么这个模式被认为是有规则的（regular）。另一方面，如果交互模式

中不包含规则模式，则被称为无规则的（irregular）。无规则和动态通信是很难处理的，尤其在消息传递编程模式中。图像浓淡处理问题属于有规则交互模式的分解。

例3.9 图像浓淡处理

在图像浓淡处理中，图像中每一像素的颜色由它的初始值和相邻像素颜色值的加权平均值决定。对这个计算进行分解非常简单，把图像划分成多个方形区域，用不同任务对这些区域进行浓淡处理。注意，每个任务都需要访问分配给它的区域的像素值和围绕此区域的像素值。这样，如果把任务作为图的节点，用一条边连接一对交互的任务，那么这种模式就是一个二维格网，如图3-22所示。 ■

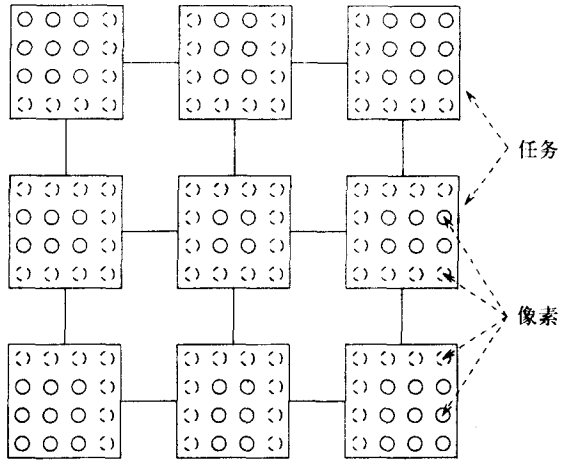


图3-22 图像浓淡处理的规则二维任务交互图。用虚线的像素需要相邻任务的边界像素颜色值

3.1.2节讨论的稀疏矩阵-向量相乘是很好的无规则交互的例子，如图3-6所示。在此分解中，即使每一任务通过分解不扫描分配给它的矩阵A的行，也预先知道需要访问矩阵A的哪些行，但任务并不知道它需要向量b的哪一项。原因是，向量b的访问模式取决于稀疏矩阵A的结构。

只读与读写 我们已经知道，任务间的数据共享会引起任务间的交互。然而，共享的类型可能会影响映射的选择。数据共享交互可分类为只读（read-only）或读写（read-write）交互。正如其名，在只读交互中，任务只需对许多并发任务之间共享的数据进行读访问。例如，在图3-10显示的对并行矩阵相乘的分解中，任务只需读共享的输入矩阵A和B。在读写交互中，多个任务需要对共享数据进行读写。例如，对于15迷宫问题，3.2.3节讨论的并行形式采用穷举搜索法来寻找解。在这个形式中，每个状态都被认为是同样适合用于进一步展开的候选者。 113
如果看上去更接近解的状态被给予优先探测权，则这种搜寻将更有效。另一种被称为启发式搜索的技术就采用这个策略。在启发式搜索中，采用启发式方法提供从每一状态到最终状态的大致距离（即到达最终状态需要移动的步骤）。在15迷宫问题中，某一给定状态下，离开原来位置的格数可作为这种启发。根据启发式的值，需要进一步扩充的状态存储在优先队列中。选择状态进行扩充时，先选择更有希望的状态，即离开原来位置的格数更少的状态，从这种状态很可能更快地得到解。这种情况下，优先队列组成共享数据，任务既要对此数据进行读

访问，也要对此数据进行写访问；任务需要在此优先队列中存放扩充后的状态，并选取下一个更有希望的状态进行下一步扩充。

单向与双向 在一些交互中，某一任务或任务子集所需要的数据或工作明显由另一任务或任务子集提供。这样的交互称为双向（two-way）交互，通常包括预定的生产者和消费者任务。在其他交互中，仅有一对通信的任务发起交互，完成此交互并不会妨碍其他交互。这样的交互模式称为单向（one-way）交互。所有的只读交互都可以被表述为单向交互。读写交互既可是单向的又可以是双向的。

在共享地址空间编程模式中，处理单向或双向交互同样容易。然而，单向交互不能直接
114 在消息传递模式中编程，因为传送的数据源必须显式送数据到接收者。在消息传递模式中，所有单向交互必须通过程序重构转化为双向交互。静态单向交互可以很容易地转化为双向通信。由于在静态单向交互编程中，交互的时间和位置预先知道，对于转化单向静态交互为双向静态交互，在合作任务中引入一个匹配交互操作就足够了。相反，动态单向交互转化为双向交互时必须经过一些非平凡的程序重构。这种重构通常包括轮询。每一任务都周期性地检测从其他任务发送来的未处理请求，如果有这样的请求，就为它提供服务。

3.4 负载平衡的映射技术

当一个计算被分解为多个任务，这些任务就被映射到不同进程，以便在最短运行时间内完成所有任务。为了减少运行时间，必需最小化并行任务的运行开销。对于一个给定的分解，有两个关键开销源，一是进程间交互花费的时间，另一个重要的开销源是某些进程的空闲时间。在总的计算完成前，由于多种原因，一些进程也会空闲。不均衡的负载分布会使一些进程比另外的进程更早结束。有时，由于任务依赖图的限制，映射到一个进程上的所有未完成任务可能要等到映射到另外进程上的任务先完成。交互与空闲通常都是映射的函数。因此，任务到进程的有效映射必须达到下面两个目标：1）减少进程间彼此交互的总时间，2）减少一些进程繁忙而其他进程空闲的总时间。

通常，这两个目标会相互冲突。例如，最小化交互的目标可以很容易地通过分配需要相互交互的任务到同一个进程得到。在大多数情况下，这种映射很容易造成进程间的负载不平衡。实际上，执行这种策略常常会映射所有任务到一个进程上。结果，在重负荷的进程还在执行任务时，轻负荷的进程经常会空闲。同样，要平衡各个进程的负载，就必须把交互频繁的任务分配到不同的进程中。由于这两个目标之间的矛盾，寻找好的映射方法是一个重要问题。

本节将讨论各种将任务映射到进程的方法，首要考虑的问题是让各个进程负载平衡，并使进程的空闲时间最小。减少进程间的交互是3.5节的主要内容。读者应该知道，分配均衡的聚合负载到每一进程是减少进程空闲的必要条件，但不是充分条件。回忆由分解得到的任务
115 并非总能同时执行。在并行算法的执行中，任务依赖图决定了哪些任务能并行执行，而哪些任务必须等到并行算法执行中某一阶段的其他任务先完成。因此，在某些并行算法中，虽然所有进程在不同时间执行同样数量的聚合任务，但只有一部分进程在运行，而剩下所包含任务的进程必须等待其他任务先执行。同样，假如其中一个任务必须等待其他任务接收或发送数据，则交互任务间较差的同步也会造成进程空闲。好的映射方法必须确保在并行算法的各个执行阶段，都能在进程间保持很好的计算和交互平衡。图3-23显示12任务分解的两种映射

方法。由于任务的依赖性，后面4个任务必需等到前面8个任务执行完才能开始，如图所示，工作负载平衡的两种映射具有不同的完成时间。

并行算法中用到的映射技术可大致分为两类：静态和动态。并行编程模式和任务的特性以及它们之间的交互决定更适合使用静态还是动态映射。

- **静态映射**：静态映射技术在算法执行前将任务分配给进程。对于静态产生的任务，既可使用静态映射，又可使用动态映射。这种情况下，选择好的映射方法取决于几个因素，包括任务大小是否已知，与任务相关的数据大小，任务间交互的特点，甚至还包括并行编程模式。即使任务大小已知，对于非均匀的任务，获得最佳映射方法通常也是NP难题。然而，在许多实际情况下，相对简单的启发式方法为最优静态映射问题提供了可接受的近似解。

  使用静态映射的算法一般更容易设计和编程。

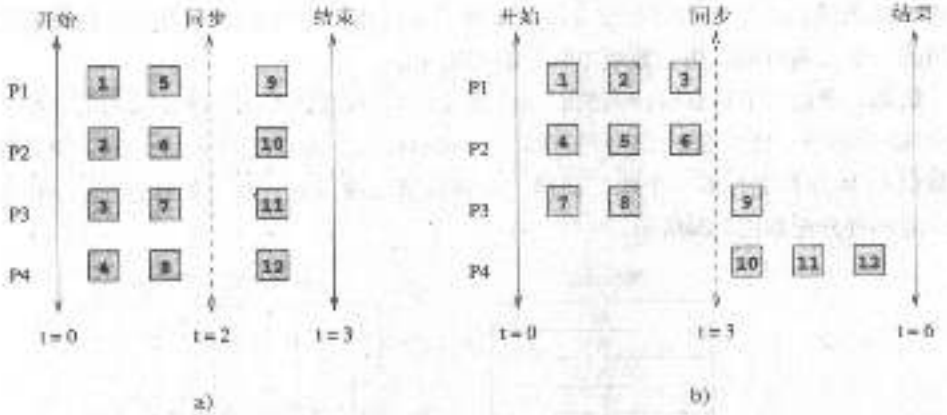


图3-23 带同步的假想分解中的两个映射

116

- **动态映射**：动态映射技术在算法执行期间在进程间分配任务。假如任务是动态产生的，那么任务也必须动态地映射。如果任务大小未知，那么静态映射会引起严重的负载不平衡，而动态映射往往会更有效。假如与任务相关的数据相对于计算很大，则动态映射也要负责在进程间移动这种数据。数据移动的开销可能会抵消动态映射的优势，反而使得静态映射更适合。然而在地址共享空间模式中，只要任务交互是只读的，那么，即使与任务相关的数据很大，动态映射也会工作得很好，读者应该知道，共享地址空间编程模式并不会自动免除数据移动的成本。假如使用的硬件是NUMA（2.3.2节），那么数据事实上，可能从远程存储器中移出。即使在cc-UMA体系结构中，数据也必须从一个高速缓存移到另一个高速缓存。

  需要使用动态映射的算法通常更复杂，尤其在消息传递编程模式中。

讨论了静态与动态映射方法的选择准则后，下面再从细节上描述这两种映射方法的各种设计方案。

3.4.1 静态映射方案

静态映射通常（虽然不是唯一的）与以数据划分为基础的分解联合使用。静态映射也常常用来映射能用静态任务依赖图自然表示的问题。下面将讨论以数据划分和任务划分为基础

的映射方案。

1. 以数据划分为基础的映射

本节将讨论基于划分的映射，划分涉及算法中最常用的表示数据的两种形式——数组和图。数据划分实际上会产生一个分解，但这种划分或分解的选择要考虑到最终的映射。

数组分配方案 以数据划分为基础的分解中，任务与由拥有者-计算规则限定的那部分数据密切相关。因此，映射相关数据到进程与映射任务到进程等价。下面将研究一些常用的在进程中分配数组或矩阵的技术。

(i) 块分配

117 块分配（block distribution）是分配数组并分派数组的均匀相邻部分到不同进程的最简单方式之一。在这些分配中，d维数组在进程之间分配，使得沿着数组维的给定子集，每一进程接收数组项的连续块。当存在交互的本地性时，数组的块分配是非常适合的，就是说，计算数组的一个元素时需要用到数组中临近的其他元素。

例如，考虑一个具有n行n列的$n \times n$二维数组A。现在选择一个维，例如第一维，并将数组划分成p个部分，使得第k个部分包含$kn/p \ldots (k+1)n/p-1$行，其中$0 \leqslant k < p$。就是每一划分包含数组A的n/p相邻行的一个块。同样，如果按第二维划分，每一划分包含n/p相邻列的一个块。这两种分配如图3-24所示。

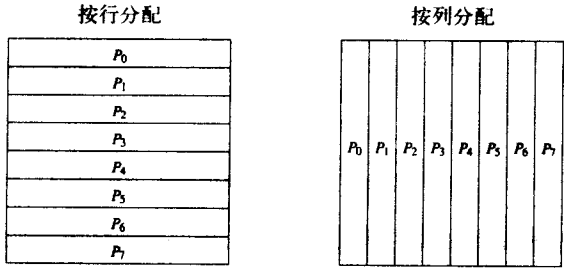


图3-24 一个数组在8个进程之间的一维划分示例

同样，如果不按一维来划分，则可以按多维进行划分。例如，对于数组A选取两个维并把它划分为多个块，每一块包含其中的$n/p_1 \times n/p_2$部分，其中$p = p_1 \times p_2$为进程数。图3-25显示分别在4×4和2×8进程网格上的两种不同的二维分配。一般说来，对于d维数组，可以使用最多d维块的分配。

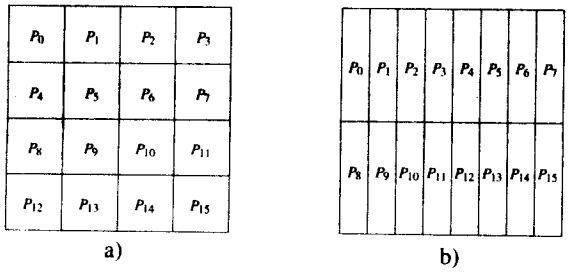
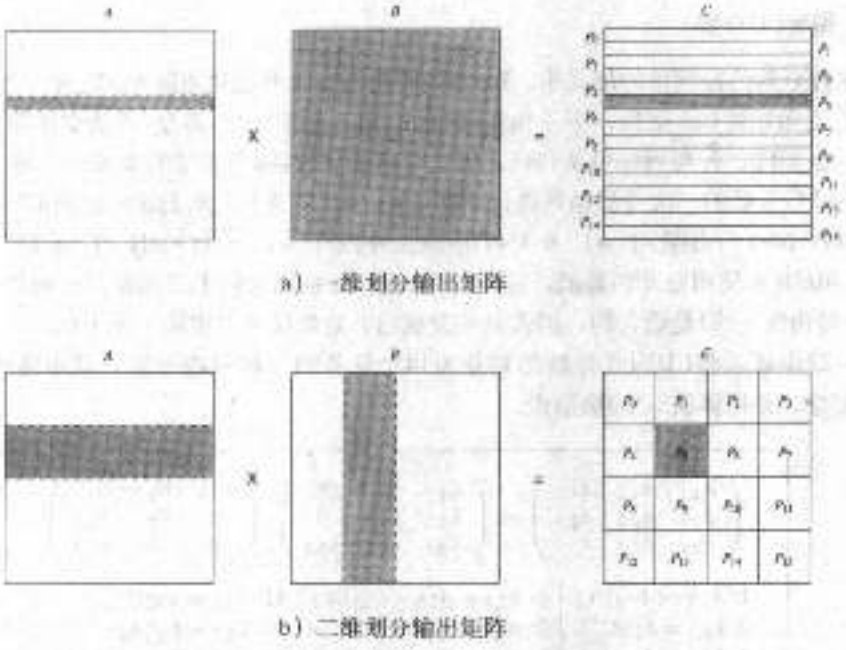


图3-25 数组二维分配的例子：a）在4×4进程网格上；
b）在2×8进程网格上

使用这些块分配，可以使多维数组上的各种并行计算操作达到负载平衡。例如，在3.2.2 [118]
节讨论过的$n \times n$矩阵乘法$C = A \times B$中，计算分解的一个方式就是划分输出矩阵C。由于C的每一元素需要同样的计算，可以使用一维或二维块分配在p个可用进程之间均衡划分C，使计算平衡。在第一种情况下，每一进程将获得C的n/p行（或列）的一个块。而在第二种情况下，每一进程将得到大小为$n/\sqrt{p} \times n/\sqrt{p}$的块。在两种情况，每一进程将负责计算$C$中分配给它的那部分元素。

如矩阵相乘的例子所示，我们经常可以选择一维或二维划分来进行映射计算（对于更高维数的数组有更多选择）。通常，高维的分配将可以使用更多的进程。例如，对于矩阵-矩阵相乘，一维划分通过简单地分配C的一行到一个进程，最多可以使用n个进程。另一方面，二维划分通过分派C的一个元素到每个进程，最多可以使用n^2个进程。

除了可以得到更高的并发度以外，对于许多问题，更高维分配有时也有助于减少不同进程间的交互。图3-26显示稠密矩阵相乘的情况。对于沿着行的一维划分，每一进程需要访问 [119]
矩阵A的相应n/p行以及整个矩阵B，如图3-26a中的进程P_3。这样需要访问的数据总量为$n^2/p + n^2$。然而，对于二维分配，每一进程需要访问矩阵A的$n/\sqrt{p}$行和矩阵B的$n/\sqrt{p}$列，如图3-26b中的进程P_5。在两维的情形，每一进程需要访问的总共享数据量为$O(n^2/\sqrt{p})$。它明显小于一维的$O(n^2)$共享数据量。



a）一维划分输出矩阵

b）二维划分输出矩阵

图3-26 矩阵相乘所需的数据共享。进程计算输出矩阵C的阴影部分要用到输入矩阵A和B的阴影部分

(ii) 循环分配和块循环分配

如果一个矩阵不同元素的计算量不同，那么块分配将会引起负载不平衡。矩阵的LU分解就是这种情况的一个典型例子，其中从矩阵的左上角到右下角，计算量不断增加。

算法3-3 分解非退化矩阵A为一个下三角矩阵L和一个上三角矩阵U的以列为基础的串行算法

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

注：矩阵L和U与矩阵A共享空间。如果$i>j$，第9行中赋值左边的$A[i,j]$与$L[i,j]$等价；否则$A[i,j]$与$U[i,j]$等价。

120 **例3.10 稠密LU分解**

在这种分解的最简单的形式中，LU分解算法把一个非退化方阵A分解为一个下三角矩阵L和一个上三角矩阵U的乘积，下三角矩阵L的对角线元素为1。算法3-3为它的串行算法。令A为一个$n\times n$矩阵，行与列标号从1到n。因式分解过程包括n个主要的步骤——每一步包括算法3-3中从第3行开始的一次外层循环迭代。在第k步中，首先用$A[k,k]$除部分列$A[k+1:n,k]$。然后从$(n-k)\times(n-k)$子矩阵$A[k+1:n,k+1]$中减去外积$A[k+1:n,k]\times A[k+1:n,k]$。在实际LU分解中，$L$和$U$并不使用分开的数组，并且$A$被修改，分别用它的上三角和下三角部分存储$L$和$U$。$L$中的主对角线上的1是隐含的，因式分解完成后，对角线元素实际上属于U。

图3-27中显示将LU因式分解的划分为14个任务的一种可能分解，其中使用矩阵的一个3×3块划分，并用算法3-3的块形式。 ■

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

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

图3-27 将LU因式分解为14个任务的一个分解

在算法3-3中，对于外层循环$k:=1$到n的每次迭代，下一嵌套循环都从$k+1$到n。换句话说，当计算进行的时候，矩阵参与计算的部分朝右下角方向收缩，如图3-28所示。因此，在

块分配中，与分配到最后行与列的进程相比，分配到开始行与列的进程（即左边行和顶部列）的计算量要少得多。例如，图3-27中对于LU因式分解的分解采用矩阵的3×3二维块划分。假如在总共有9个进程中，映射与某个块相关的所有任务到一个进程中，那么会造成很大的空闲时间。首先，因为计算矩阵不同块需要不同工作量，如图3-29所示。例如，计算$A_{1,1}$的最终值（实际上是$L_{1,1}$ $U_{1,1}$）仅需要一个任务——任务1，另一方面，计算$A_{2,2}$的最终值需要3个任务——任务9，任务13和任务14。第二，即使与块相关的任务还没有完成，在块上工作的进程也可能空闲。如果在一个或更多映射到其他进程中的任务执行完之前，任务依赖图不允许这个进程剩下的任务继续，那这个进程就会空闲。 121

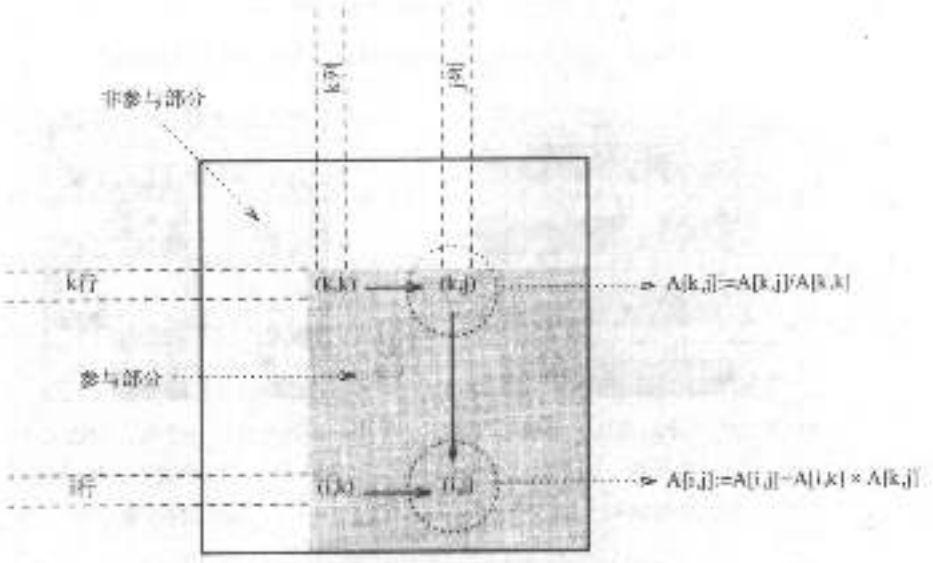


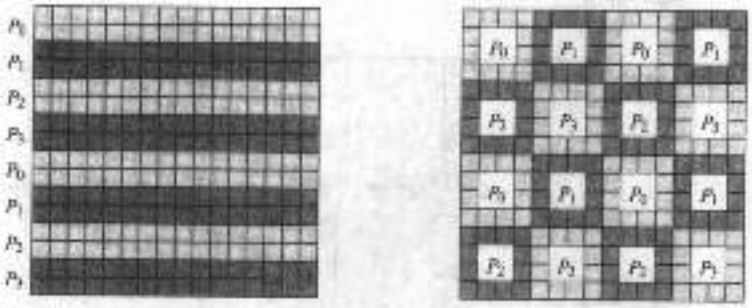
图3-28 高斯消元法的一种典型计算以及外循环第k次迭代中系数矩阵的参与部分

块循环分配（block-cyclic distribution）是块分配方案的一种变体，可用来缓解负载不平衡和进程空闲问题。第8章将详细地讨论使用块循环映射的LU分解，讲述块循环分配怎样使得任务分配比图3-29中的更加平衡。块循环分配的中心思想是把数组划分成比可用进程数更多的块，这样就可以采用循环的方式把划分（和相关任务）分派给进程，以便每一进程都能得到若干不相邻的块。更确切地说，在p个进程之间的矩阵的一维块循环分配中，$n \times n$矩阵的行（列）被划分为$n/(\alpha p)$相邻的行（列）的αp组，其中$1 \leqslant \alpha \leqslant n/p$。现在这些块以环绕方式分配到$p$个进程，块$b_i$被分派到进程$P_{i\%p}$（%是模运算符）。这种分配分派矩阵的$\alpha$块到每一进程，但以后分配到同一进程的每个块都相距$p$块。通过划分$n \times n$矩阵为大小为$\alpha\sqrt{p} \times \alpha\sqrt{p}$的方块，并以环绕方式分派它们到一个假想的$\sqrt{p} \times \sqrt{p}$进程数组中，就能获得矩阵的二维块循环分配。同样，块循环分配也能扩展到更高维的数组中。图3-30显示二维数组的一维和二维块循环分配。 122

因为所有进程都从矩阵的各个部分得到一个任务样本，所以块循环分配能极大地减少进程的空闲时间。这样，即使矩阵的不同部分需要不同计算时间，每一进程的工作量是均衡的。同时，由于分派到某一个进程的任务属于矩阵的不同部分，这样至少进程的某些任务很可能在任意指定时刻做好执行的准备。

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

图3-29 一个一维块分配的LU分解任务到进程的自然映射



a）数组的行分组为块，每块包含两行，得到8个行块，这些块以回绕方式分配到4个进程中

b）矩阵划分为16个块，每块大小为4×4，并以环绕方式映射到2×2的进程网格中

图3-30 4个进程间的一维和二维块循环分配的例子

假如增加α到它的上限n/p，那么在一维块循环分配中每一块只有一行（列），而在二维块循环分配中每一块只有一个元素。这种分配称为循环分配（cyclic distribution）。循环分配是块循环分配的极端情况，由于分解的极细粒度，可以得到很好的负载平衡。然而，由于进程中没有任何相邻数据，没有本地性可能造成严重的性能缺陷。另外，与每一任务的总计算量相比，这样的分解通常会导致高度的交互。α的下限1会产生最大的本地性和最优交互性，但这种分配退化成为块分配。因此，一个适当的α值必须用来保持交互守恒和负载平衡之间的均衡。

123

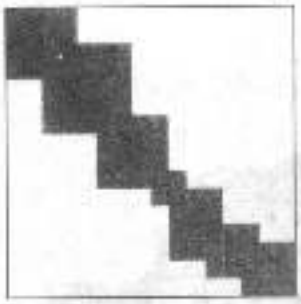
如像块分配的情形，高维块循环分配通常更可取，因为这样会使得任务间更少的交互。

(iii) 随机块分配

当任务分配具有一些特别的模式时，块循环分配并不总能平衡负载计算。例如，考虑图3-31a中的稀疏矩阵，其中阴影区域对应于包含非零元素的区域。假如这个矩阵的分配使用二维块循环分配，如图3-31b，那么对角线上的进程P_0、P_5、P_{10}和P_{15}将会被分配到比其他进程更多的非零元素。实际上，一些如P_{12}这样的进程，根本得不到任何任务。

随机块分配（randomized block distribution）是更适用的块分配形式，能够用在如图3-31所示的情况中。与块循环分配一样，通过使划分的数组块数多于可用进程数来寻求负载平衡。

然而，块被均匀地和随机地分配到进程中。可用如下方式实现一维随机块分配：使用长度为αp（等于块数）的向量V，对于$0 \leqslant j < \alpha p$，$V[j]$设置为j。现在随机排列V，并把在存储在$V[i\alpha \cdots (i+1)\alpha-1]$中的块分配给进程$P_i$。图3-32显示$p = 4$和$\alpha = 3$时的情况。对$n \times n$数组的一维随机块分配同样可以通过下面的方式计算得到：随机排列长度为$\alpha\sqrt{p}$两个向量的每一个，并用它们来选择分派到每个进程的块的行下标和列下标。如图3-33所示，随机块分配比图3-31中执行的负载平衡计算更为有效。



a)

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

b)

图3-31 用b）所示的块循环分配分配阵列a）中进行的计算将导致负载不平衡

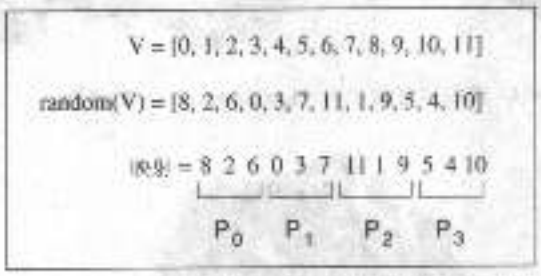
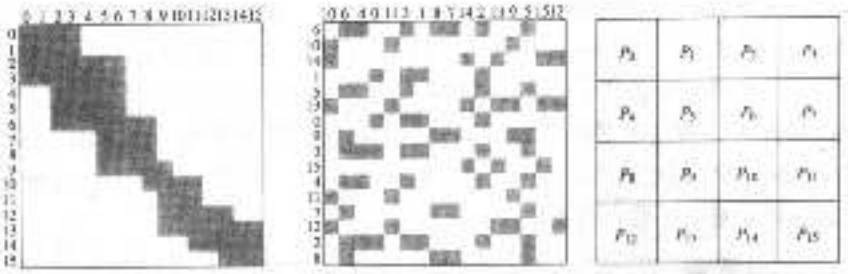


图3-32 一维随机块映射12个块到4个进程（即$\alpha = 3$）

图划分 对于使用稠密矩阵并有结构和规则交互模式的许多算法，在平衡计算和最小化交互方面，前面讲述的基于数组的分配方案都非常有效。然而，有许多运行在稀疏数据结构上的算法，这些算法的数据元素间的交互模式具有数据依赖，并且非常不规则。物理现象的数值模拟提供了这种计算类型的大量来源。在这些计算中，物理定义域被离散化并用格网单元来表示。物理现象的模拟被建模，然后计算各个格网点物理量的值。一个格网点上的计算通常需要与该点对应的数据以及与格网中相邻点对应的数据。例如，图3-34显示苏必利尔湖（Lake Superior）上的格网，对湖中水污染扩散物理现象的模拟包含计算在各种不同的时间间隔格网上各个顶点的污染等级。 124

通常各个格网点的计算量都相同，所以，只要分派相同数量的格网点到每一进程，就可以很容易地达到负载平衡。然而，如果不尽量分派相邻格网点到同一进程，就可能由于过多的数据共享而导致高的交互开销。例如，假如每一进程收到的格网点集是随机的，如图3-35所示，那么每一进程就要访问一个很大的属于其他进程的点集，才能完成分派给它的格网点计算。 125



a) b) c)

图3-33 用b）中所示的二维随机块分配来分配阵列a）中执行的计算，如c）中所示

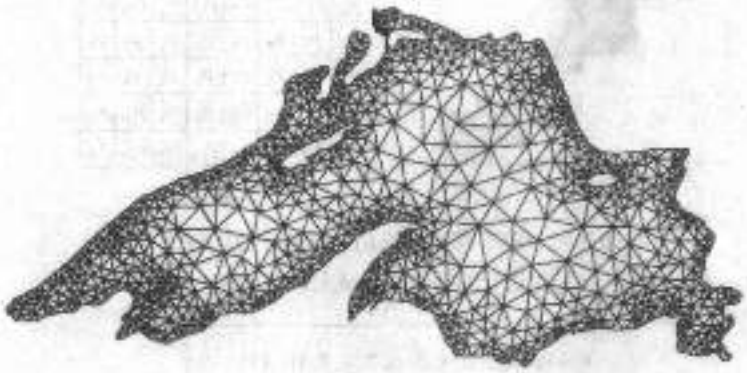


图3-34 模拟苏必利尔湖（Lake Superior）的格网

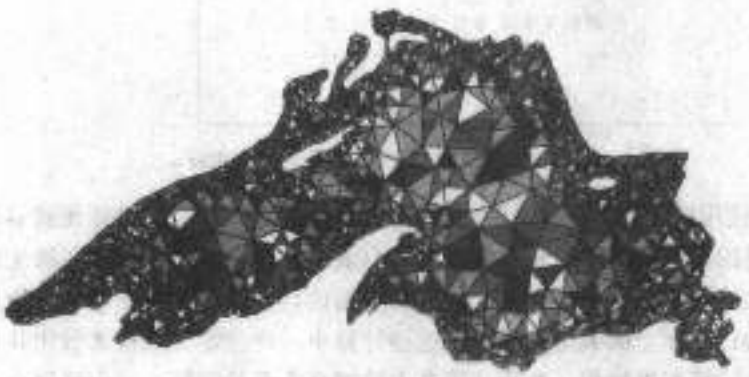


图3-35 格网单元分配到8个进程的一种随机分布

理想情况下，我们希望分配格网点使得负载平衡。同时，使得每一进程完成属于它的格网点计算需要访问的数据量最小化。因此就需要划分格网为p个部分，每一部分包含数量大致相等的格网点或顶点，而且穿过边界的边（也就是连接属于两个不同划分点的边）最小化。寻找确切的最优划分是一个NP完全问题。然而，采用强大的启发式算法能计算一些合理的划分。在用这种方式得到的格网点划分中，p个划分中的每一个被分派给p个进程中的一个。结果，每一进程被分配一个相邻区域的格网，使得需要跨划分边界访问的格网点数最小。图3-36显示苏必利尔湖格网的一种很好的划分——典型的图划分软件能产生这种划分。

126



图3-36 用图划分算法将格网单元分配到8个进程

2. 以任务划分为基础的映射

有一种映射基于划分任务依赖图，并将节点映射到进程中。当计算可用任务大小已知的静态任务依赖图形式自然地表示出来时，可以采用这种映射。与前面的方法一样，该映射也要达到并行算法中最小化空闲时间和最小化交互时间这两个冲突的目标。决定一个通用任务依赖图的最优映射是NP完全问题。但在特定情况下，人们通常可以找到简单的最优解或可接受的近似解。

完全二叉树的任务依赖图是以任务划分为基础的简单例子。这种任务依赖图会出现在一些使用递归分解的实际问题中。例如在一个列表中查找最小值的问题（图3-9）。图3-37显示将这个任务依赖图映射到8个进程时的情况。很容易看出，通过映射相互依赖的任务到同一个进程（即沿着树的直接分支上的任务），而其他任务映射到彼此相距只有一个通信链路的进程，就可使交互开销最小化。虽然会有一些不可避免的空闲（例如，进程0在处理根任务时所有其他进程空闲）。但这种空闲是任务依赖图所固有的。图3-37所显示的映射并没有造成任何附加的空闲。由任务依赖图允许并发执行的所有任务被映射到并行执行的不同进程。

对于某些问题，通过划分任务交互图，可以得到求解这些问题的映射方法的近似解法。在前面讨论的数据划分中，关于苏必利尔湖污染扩散的模拟问题，可以定义让每一任务负责一个特定格网点的有关计算。现在，这个用来使苏必利尔湖离散化的格网也可作为任务交互图。因此，对于这个问题，通过图划分寻找好的映射方法也可被看作为任务划分。 127

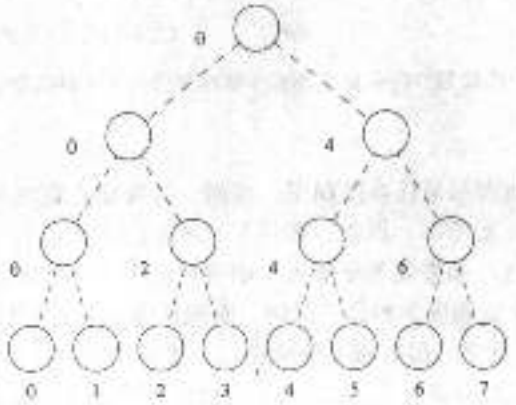


图3-37 二叉树任务依赖图映射到进程的超立方体中

3.1.2节讨论的稀疏矩阵相乘也可采用任务划分。图3-38中显示图3-6的一个简单任务交互图映射。这种映射把与b的4个相邻元素对应的任务分派到每个进程。对于图3-6中显示的3个进程的稀疏矩阵和向量相乘问题，图3-39显示另一种任务交互图的划分。列表Ci包含b的下标，它表示进程i上的任务需要从映射到其他进程的任务访问数据。在这两种情况下，比较列表C0，C1和C2可以发现，建立在划分任务交互图上的映射比简单的映射造成更小的进程间b的元素交换。

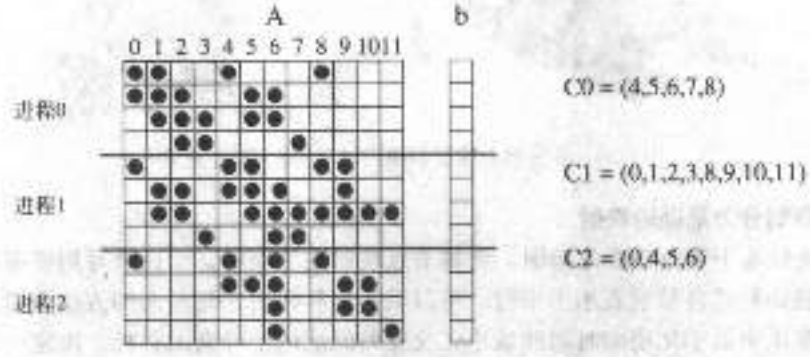


图3-38 稀疏矩阵向量相乘映射到3个进程的一种方法。

128 列表Ci包含进程i需要从其他进程访问的b元素下标

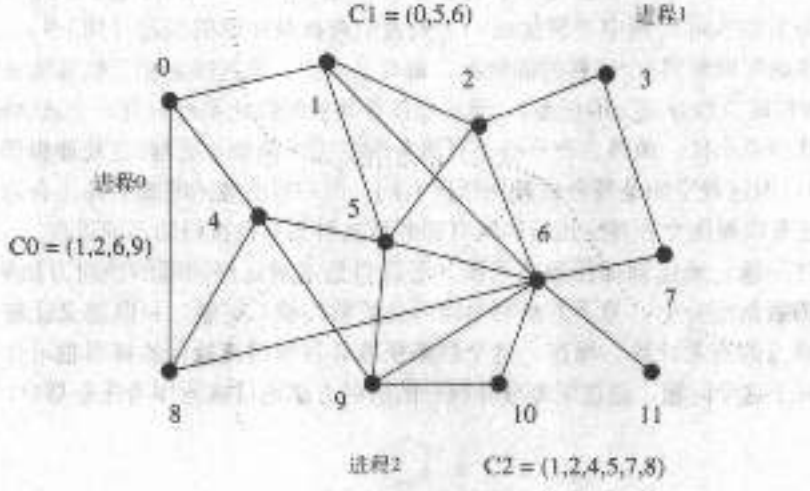


图3-39 通过划分任务交互图减少稀疏矩阵-向量相乘的交互开销

3. 分级映射

有些算法可以自然地表示为任务依赖图；然而，纯粹建立在任务依赖图上的映射可能会遇到负载不平衡、或并发度不够。例如，在图3-37所示的二叉树任务依赖图中，树的顶部只有小部分任务可并行执行。如果任务足够大，则将任务进一步分解成更小的子任务，就能得到更好的映射。如果任务依赖图为四层二叉树，则根任务可在8个进程中划分，下一层的每个任务在4个进程中划分，再下一层的每个任务在二个进程中划分。8个叶子任务可以一一映射到进程中。图3-40显示这种分级映射。例3.4介绍的并行快速排序具有和图3-37相似的任务依赖图，因此图3-40所示的分级映射的理想候选对象。

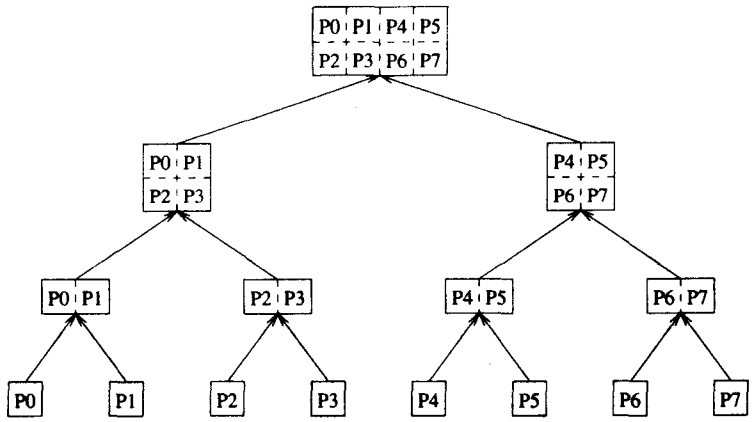


图3-40 任务依赖图分级映射的一个例子。数组代表的每一节点是超任务。数组的划分表示子任务，这些子任务映射到8个进程中

上面讨论的分级映射直接应用的一个重要实际问题是在稀疏矩阵分解。稀疏矩阵分解中的高层次计算由称为*消去图*（如果矩阵对称则为*消去树*）的任务依赖图引导。然而，*消去图*中的任务（尤其是靠近根的任务）通常包括大量计算，要使用数据分解进一步划分为子任务。分级映射在顶层使用任务划分，在底层使用数组划分，所以应用了混合分解。一般地，分级映射具有许多层，不同的分解和映射技术可以适合不同的层。 129

3.4.2 动态映射方案

如果静态映射导致进程间负载高度不平衡，或任务依赖图本身就是动态的，就必须采用动态映射而排除静态映射。由于用动态映射的首要原因是保持进程间的负载平衡，动态映射也称为动态负载平衡。动态映射技术通常分为集中式或分布式。

1. 集中式方案

在集中动态负载平衡方案中，所有可执行的任务都维持在一个公用的中心数据结构里，或者维持在一个特别进程中或进程子集中。如果用特别进程来管理可用任务池，则通常称该进程为*主进程*（master），而称其他靠主进程来获取任务的进程为*从进程*（slave）。当一个进程没有任务时，它就从中心数据结构或主进程处获取一部分可用任务。当新任务产生时，它就被添加到中心数据结构中或通报告主进程。通常，集中式负载平衡方案比分布方案更容易实施，但可能限制可扩展性。如果使用的进程越来越多，大量访问公用数据结构或主进程就会导致瓶颈。 130

下面举一个集中式映射适用的一个例子，对$n \times n$矩阵A的每一行中的元素进行排序。用串行方式，可用下面的简单程序段完成：

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

如果使用一些常用的排序算法，如快速排序，排序花费的时间会由于要排序元素的初始状态会大不相同。因此，上面程序段的每一次循环迭代可能花费不同的时间。如果用简单的

映射方法，将对矩阵的每一行进行排序的任务分配给每一进程，就会造成负载不平衡。这种情况下，要解决潜在的负载不平衡，可以通过维持一个待排序行的中心下标池来解决。当有进程空闲时，下标池只要不为空，空闲进程就会拾取一个可用下标，对此下标的行进行排序，并从下标池中删除此下标。这种在并行进程的调度循环中独立迭代方法称为*自调度*（self scheduling）。

对保持计算的平衡来说，一次只对一个进程分派一个任务是十分有效的；然而在访问共享工作队列时，可能会成为瓶颈，尤其当每一个任务（即这种情况下的每一次循环迭代）不需足够大的计算量时更是如此。假如每一任务大小为M，要花费Δ时间分派一个任务到进程，那么最多只有M/Δ个进程能有效地保持忙状态。如果进程一次得到多个任务，这种瓶颈就可以缓解。在*块调度*（chunk scheduling）中，每次一个没有工作的进程会得到一组任务。这种方案的潜在问题是，如果一次分派到的任务数量（也就是块）很大，就可能导致负载不平衡。当程序运行中减小块的大小，可以减少由大块造成的负载不平衡。也就是说，开始块很大，但随着剩下的迭代次数的减少，块也不断减小。已经设计出许多逐渐调节块大小的方法，减小块大小既可以是线性的也可以是非线性的。

2. 分布式方案

在分布式动态负载平衡方案中，可执行任务集被分布在多个进程中，在运行时通过交换任务来保持负载平衡。每一进程都可以发送任务或从其他进程接收任务。这些方法不会引起集中式方法中的瓶颈。分布式负载平衡方案中的一些关键参数如下：

- 怎样成对地发送和接收进程？
- 由发送者还是接收者启动任务的传递？
- 每次交换中传递多少任务？假如传递任务太少，那么接收者可能得不到足够任务，而频繁的传递会导致过多的交互。而如果传递任务太多，那发送者很快就会空闲，又会导致频繁的传递。
- 何时传递任务？例如，在接收者发起的负载平衡中，当进程已经运行完任务或接收者只剩下太少任务并且预先的任务不久就要执行完时就需要新的任务。

131

对上面每一个参数的详细研究超出本章的范围。在后面各章还会讨论这些负载平衡方法适用的并行算法，特别是第11章的并行搜索算法。

3. 并行结构的适应性

原则上，集中式和分布式映射方法都能应用在消息传递和共享地址空间模式中。然而，动态任务负载平衡方案的性质决定了这些方法需要在进程间传递任务。因此，为了让这些方案能在消息传递计算机上有效实现，与计算对应的任务大小应远大于与任务对应的数据大小。在共享地址空间模式中，虽然也有一些隐式的向本地高速缓存或进程存储区的数据移动，但任务不需显式移动。通常，在共享地址体系结构中，待移动任务的计算粒度比消息传递体系结构上的要小得多。

3.5 包含交互开销的方法

如早先所述，对于有效的并行程序而言，减少并行任务间的交互开销非常重要。并行程序由于进程间的交互而引起的开销取决于多个因素，如交互中交换数据的大小、交互的频率以及交互的空间和时间模式，等等。

本节将讨论减少并行程序中交互开销的一些常用技术。这些技术用来处理上面提到的三个因素中的一个或多个来减少交互开销。有些技术适用于设计算法的分解和映射方案，而有些技术则适用于在给定的模式中编写算法程序。没有一种技术对于所有并行编程模式而言都是适用的，其中一些技术还需要底层硬件的支持。

3.5.1 最大化数据本地性

在大多数非平凡并行程序中，不同进程执行的任务需要访问某些公用数据。例如，在稀 132
疏矩阵-向量乘法$y = Ab$中，与计算向量y的每个元素（图3-6）对应的任务需要访问输入向量b的所有元素。除了共享原始输入数据外，如果进程需要用到由其他进程产生的数据，那么也会造成交互。如果更多地使用本地数据或最近已被取来的数据，则交互开销能够减少。数据本地性增强技术包括许多方案，用来最小化对非本地数据的访问，最大化重用最近访问过的数据，以及最小化访问频率。许多情况下，这些方案与数据重用优化在本质上是相似的，它们常用在基于现代高速缓存的微处理器中。

最小化数据交换量 最小化并发进程要访问的共享数据是减少交互开销的一个最基本方法。这与最大化时间数据本地性相似，就是尽可能多地连续引用相同数据。很明显，对于一个进程执行的任务，要尽可能多地使用可用的本地数据进行计算，避免从其他地方引入数据到本地内存或高速缓存。如前所述，使用合适的分解和映射方案能达到这一目的。例如，在矩阵相乘中，如果使用从计算到进程的二维映射，就可以减少每一任务需要访问的共享数据（即矩阵A和B）量，与一维映射（图3-26）相比，数据量从$n^2/p + n^2$减到$2n^2/\sqrt{p}$。通常，使用更高维分布有助于减少对非本地数据的访问。

使用本地数据存储中间结果是减少多个进程需要访问的共享数据量另一个方法，并只在存放最终计算结果的时候才访问共享数据。例如，并行计算两个长度为n的向量点积时，让p个任务中的每一个对n/p对元素相乘。不将一对元素的的单独乘积添加到最终结果中去，而是每一任务首先把分派给它的长为n/p的向量生成部分的点积存储在本地，最后只需访问最终共享位置一次，把这个部分结果添加到最终结果中去。访问共享数据的次数将从n次减少到p次。

最小化交互频率 最小化交互频率是减少并行程序中交互开销的一个重要方法，因为在许多体系结构中，每一次交互都有一个相对高的启动开销与之对应。重构算法，使得共享数据以大量的数据块访问和使用，就能减少交互频率。这样，即使这种重构未必能减少需要访问的总共享数据量，通过大量的访问减少启动开销，就可以减少总交互开销。这与增加数据访问的空间本地性很相似，就是确保连续访问数据位置很接近。在共享地址空间体系结构中， 133
每访问一个字时，整个包括许多字的高速缓存行都被取来。如果程序具有空间本地性，那么就只要访问很少的高速缓存行。在消息传递系统中，空间本地性使得网络中的消息传递很少，因为每一个消息可以传递更大量有用的数据量。在消息传递系统中，如果交互模式允许，且多个消息的数据可同时获得，尽管在分离的数据结构中，也可以将同样一对源/目标消息组合为一个更大的消息来进一步减少消息数量。

稀疏矩阵-向量相乘的并行形式可以使用这种技术来减少交互开销。在典型的应用中，要用矩阵的非零模式但不同数值的非零值，对稀疏矩阵-向量相乘重复地进行计算。并行解决这个问题时，如果进程执行本地计算时要访问输入向量元素，则它可能要和其他进程交互。进程通过对分派给它的稀疏矩阵行的非零模式进行一次扫描，就可以确切地确定需要哪些输入

向量的元素，以及从哪些进程中可以得到它们。这样，在开始每个乘法前，进程可以首先收集它所需要的输入向量的非本地元素，然后再执行无交互相乘。与在计算需要的时候再去访问输入向量的非本地元素相比，这种策略要好得多。

3.5.2 最小化争用与热点

到目前为止，我们讨论的减少交互开销的方法都集中在直接或间接减少数据的传输频率和传输量。然而，数据访问和任务间的交互模式导致的争用也会增加总交互开销。通常，当多个任务试图并发访问同一资源时就会产生争用。同一个互连链路上多个数据的同时传输、对同一存储块的多个同时访问或者多个进程同时对同一个进程发送消息，都会产生争用。这是因为在某一时刻多个操作中只有一个操作可以执行，而其他的必须排队并且顺序执行。

考虑两个矩阵相乘$C = AB$，采用如图3-26b所示二维划分。假设p是任务的数量，每一任务映射到一个进程。让每一任务负责计算一个单独的值$C_{i,j}$，其中$0 \leqslant i, j < \sqrt{p}$。可按下面公式直接计算$C_{i,j}$的值（以矩阵块的记号写出）：

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} * B_{k,j} \tag{3-1}$$

从上面公式的存储访问模式可以看出，在$\sqrt{p}$个步骤的任何一步中，$\sqrt{p}$个任务都将访问
134 矩阵A和B的同一块。特别是，计算C的同一行时所有任务都将访问A的同一块。例如，所有计算$C_{0,0}$，$C_{0,1}$，…，$C_{0,\sqrt{p}-1}$的$\sqrt{p}$个进程都将同时试图访问$A_{0,0}$。类似地，所有计算C的同一列的任务都要访问B的同一块。在NUMA共享地址空间和消息传递并行体系结构中，并发访问矩阵A和B的这些块的需求将会造成争用。

减少争用的一个方法是重新设计并行算法，以无争用的模式访问数据。对于矩阵相乘算法，通过改变公式（3-1）执行的块相乘的顺序，可以消除这种争用。执行这些块乘法计算$C_{i,j}$的一种无争用方法是用下面的公式：

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} * B_{(i+j+k)\%\sqrt{p},j} \tag{3-2}$$

其中‘%’表示模运算。使用这个公式，所有计算C中同一行的任务$P_{*,j}$都将访问块$A_{*,(*+j+k)\%\sqrt{p}}$，对于每一任务这都是不同的。类似地，所有计算C中同一列的任务$P_{i,*}$都将访问块$B_{(i+*+k)\%\sqrt{p},*}$，对于每一任务这也是不同的。这样，只通过简单地重新安排计算块相乘的顺序就可以完全消除争用。例如，在所有计算C的第一行块的进程间，计算$C_{0,j}$的进程将访问A的第一行块中的$A_{0,j}$而不是$A_{0,0}$。

对于通向主进程的共享数据结构和通信通道，动态映射（3.4.2节）的集中式方案通常是造成争用的一个主要来源。选择分布式映射方案法代替集中式映射方案，可以减少争用，虽然前者可能更难实施。

3.5.3 使计算与交互重叠

在交互启动后，如果在等待的同时做一些有用的计算，进程花在等待共享数据到达或接收额外工作上的总时间可以减少，而且会减少很多。许多技术可用来使计算与交互重叠。

简单的重叠方法是，足够早地启动交互，以便在计算需要的时候它已经完成。为了达到这个目的，必须能够鉴别出不依赖交互并能在交互之前执行的计算。然后必须组织并行程序，以便在比原来算法需要的执行时间点之前启动交互。通常，可能做到这一点，只要交互模式是空间静态的或时间静态的（因此是可预测的），或者准备执行的多个任务在同一进程中是可用的，使得一个任务在等待交互完成时，进程就可以执行另一个任务。读者应该注意到，如果通过增加并行任务的数量来提高交互重叠，就会减少任务的粒度，通常这样也会导致开销增加。因此，必须审慎地使用这种技术。 135

在某些动态映射方案中，当某个进程执行完任务后，它就发出请求并从其他进程得到额外的任务。然后它就等待请求被响应。假如进程能预测它将要完成任务，并预先启动任务传递交互，那么该进程就可以在等待请求被响应的同时继续执行任务。根据问题本身的性质，估算剩余工作可能很困难也可能很容易。

在大多数情况下，使计算与交互重叠需要从编程模式、操作系统以及硬件得到支持。编程模式必须提供允许交互与计算同时执行的机制，这种机制应该得到底层硬件的支持。不相连的地址空间模式与体系结构通常通过非阻塞消息传递的原语提供这种支持。这种编程模式提供发送和接收消息的函数，在发送和接收完成前，将控制返回给用户程序。这样的话，程序就可以使用这些原语来启动交互，然后继续执行计算。假如硬件允许计算与消息传递并发执行，那么交互开销就可以大大减少。

在共享地址空间体系结构中，预取硬件常有助于计算与交互的重叠。这种情况下，访问共享数据只不过是常规的读取或存储指令。预取硬件能预测即将访问的存储地址，并能在需要它们之前启动访问。没有预取硬件时，如果用编译器检测访问模式，并在使用存储地址之前，对某些关键存储地址进行伪引用，也能达到同样的效果。这种方法的成功程度取决于预取硬件可推测出程序的可用结构，以及在计算进行时预取硬件能发挥作用的独立程度。

3.5.4 复制数据或计算

复制数据或计算是可能减小交互开销的另一种有用的技术。

在一些并行算法中，多个进程可能需要以不规则的模式频繁地以只读方式访问共享数据结构，如散列表。在每一进程中复制共享数据结构的一个副本，使得在复制阶段最初交互开始后，所有随后对该数据结构的访问都没有任何交互开销，除非额外存储需求被禁止，否则这可能是最好的一种情况。

在共享地址空间模式中，频繁被访问只读数据的复制常常受到没有程序员显式干预的高速缓存的影响。在某些结构和编程模式中，只读访问共享数据比访问本地数据的开销大得多 136
或更难表示，显式数据复制特别适合于这种情况。因此，数据复制更利于消息传递编程模式，它能减少交互开销，并极大地简化并行程序的编写。

然而，数据复制的好处也并非没有成本。数据复制增加并行程序的存储需求。用来存储复制数据的总存储量随着并发进程的数量线性增加。这就可能限制并行计算机可以解决问题的规模。因此，必须有选择地复制相对少量的数据。

除了输入数据以外，并行程序中的多个进程也常常共享中间结果。在某些情况下，一个进程计算出中间结果要比从其他进程得到这个中间结果更具成本有效性。此时，用复制计算可用来抵消交互开销。例如，在N点级数中执行快速傅里叶变换时，要计算N个ω的不同次幂（或旋转因子）并用在各个点中。在并行实现FFT时，不同进程要重叠这N个旋转因子各次幂

的子集。在消息传递模式中，每一进程最好在本地计算它需要的所有旋转因子。虽然并行算法可能比串行算法执行更多的旋转因子计算，但还是比共享旋转因子快。

3.5.5 使用最优聚合交互操作

如3.3.2节中的讨论，并发任务间的交互模式常常是静态的和规则的。这种静态的和规则的交互模式就是指那些由一组任务来执行，并被用于达到规则数据访问，或对分布数据执行特定的计算类型。已经确定经常出现在许多并行算法中的像聚合（collective）交互操作这样的关键操作。对所有进程广播数据或对属于不同进程的每一进程添加数据是这种聚合操作的例子。聚合数据共享操作可划分为三类。第一类包含被任务用来访问数据的操作，第二类操作被用于执行一些通信密集的计算，而最后的第三类被用于同步。

已经建立高度优化这些聚合操作的实现，用来最小化由于数据传送以及争用引起的开销。第4章将描述一些实现常用的聚合交互操作的算法。这些操作的优化实现可从大多数并行计算机供应商处以库的形式获得，如MPI（消息传递接口）。所以，算法设计者不需要考虑这些操作的实现，而只需集中考虑从这些操作中可得到的功能。然而，正如3.5.6节所述，有时交互
137 模式使得并行程序员实现自己的聚合通信过程更有价值。

3.5.6 一些交互与另一些交互的重叠

如果底层硬件的数据传送能力允许，那么多对进程之间的交互重叠将可以减少有效通信量。让我们看一个重叠交互的例子，在具有4个进程P_0、P_1、P_2和P_3的消息传递模式中，使用常用的一对多广播聚合通信操作。从P_0广播数据到其他进程的常用算法按如下方式进行。第一步，P_0发送数据到P_2。第二步，P_0发送数据到P_1，同时，P_2发送从P_0接收到的同一数据到P_3。这样整个操作在两步内完成，因为第二步中两个任务的交互只需一步完成。操作过程如图3-41a所示。另一方面，简单的广播算法将从P_0发送数据到P_1、P_2和P_3，因此总共需要三步完成，如图3-41b所示。

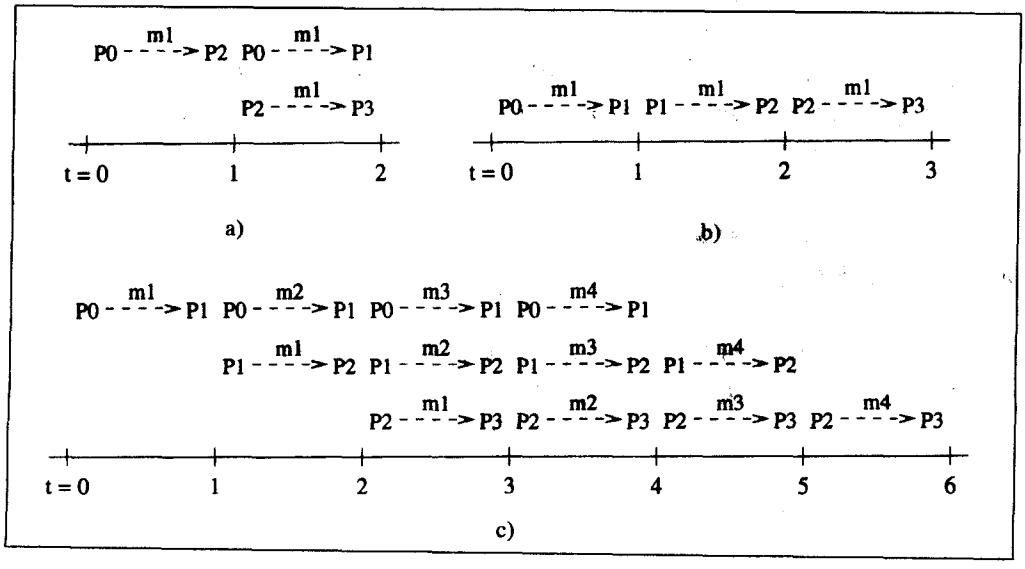


图3-41 从一个进程广播数据到4个进程的重叠交互过程

然而有趣的是，在某些情况下，图3-41b所示的简单广播算法可能更适合于增加重叠的数量。假设一个并行算法需要先后广播4个数据结构。如果使用第一个的二步广播算法，整个交互将需要8个步骤。但使用简单算法来完成这个交互只需6个步骤，如图3-41c所示。在第一步，P_0发送第一个消息到P_1。第二步，P_0发送第二个消息到P_1，同时P_1发送第一个消息到P_2。第三 138
步，P_0发送第三个消息到P_1，P_1发送第二个消息到P_2，而P_2发送第一个消息到P_3。以这种类似流水线的方式进行，4个消息中的最后一个在第4步后从P_0发出，并在第6步到达P_3。对于单个广播操作而言，这个方法的开销很大，所以它不大可能被放到聚合通信库中。然而，在这种情况下，程序员必定从算法的交互模式中推断出，自己写聚合通信函数要好于使用3.5.5节讨论的方法。

3.6 并行算法模型

到目前为止，我们已经讨论了分解、映射以及最小化交互开销的技术，下面将提出常用的一些并行算法模型。一个算法模型就是通过选择一种分解和映射技术并用合适的策略最小化交互来构造并行算法的一种典型方法。

3.6.1 数据并行模型

数据并行模型（data-parallel model）是最简单的算法模型之一。在这种模型中，任务被静态或半静态地映射到进程，并且每个任务都对不同数据进行相似的操作。这种并行性是把同样的操作并发地运用到不同数据项上产生的结果，因此称为*数据并行*（data parallelism）。任务可能分阶段执行，并且运行在不同阶段上的数据可能不同。通常，数据并行计算阶段穿插着各个交互，以使各个任务同步，或找到新的数据给各个任务。由于所有任务执行类似的计算，分解问题到各任务通常以数据划分为基础，因为均匀数据划分加上静态映射就足以保证负载平衡。

数据并行算法既可以在共享地址空间模式中实现，也可以在消息传递模式中实现。但是，消息传递模式中划分的地址空间可能允许更好的位置控制，这样就能为本地性提供一个更好的处理。另一方面，共享地址空间可以简化编程工作，在不同算法阶段有不同数据分布的情况下尤其如此。

选择在本地保存分解，并且在适合时将计算与交互重叠和利用优化的聚合交互例程，能使数据并行模式中的交互开销最小化。数据并行问题的一个重要特点，就是数据并发度随着问题规模的增加而增加，这样就可以使用更多的进程来有效地解决更大的问题。

3.1.1节讲述的稠密矩阵相乘就是数据并行算法例子。在图3-10中显示的分解中，所有任务都相同，而且它们应用于不同的数据。 139

3.6.2 任务图模型

如3.1节所述，任何并行算法中的计算都可看作为一个任务依赖图。任务依赖图既可以是平凡的，如在矩阵相乘中，也可以是非平凡的（习题3.5）。但在某些并行算法中，任务依赖图显式地用在映射中。在*任务图模型*（task graph model）中，使用任务之间的相互关系来提高本地性或减少交互开销。如果某一问题中，与任务对应的数据量远大于与任务相对应的计算，则通常用任务图模型来解决这类问题。任务的静态映射常被用来减少任务间的数据传送

开销，有时也可能用分布式动态映射，但即便如此，动态映射也使用与任务依赖图结构以及任务交互模式的信息来减少交互开销。在具有全局可访问空间的模式中，任务更容易被共享，但在不相连的地址空间中，也可以使用其他一些机制来共享任务。

典型的适用于这种模型的交互减少技术包括：在映射以任务的交互模式为基础的任务时，通过提高本地性来减少交互量和交互频率，以及使用异步交互方法让交互与计算重叠。

以任务图模型为基础的算法的例子包括快速并行排序（9.4.1节）、稀疏矩阵分解以及从分治分解中导出的许多并行算法。这种在任务依赖图中通过独立任务自然表示的并行形式称为*任务并行*（task parallelism）。

3.6.3 工作池模型

工作池（work pool）或*任务池*（task pool）模型的特征是，动态映射任务到进程以保持负载平衡，在这种映射中，任何任务可能由任何进程执行。没有必要预映射任务到进程。映射既可以是集中式的也可以是分布式的。指向任务的指针可以保存在物理共享列表、优先队列、散列表或树中，或存储在物理分布的数据结构中。工作既可以在开始时静态地获得，也可以动态地产生；也就是说，进程可以产生工作并把它添加到全局（也可能是分布式）工作池中。如果工作是动态产生的，并且使用分散映射方法，那么所有进程就要使用终止检测算法（11.4.4节），使所有进程能实际检测整个程序是否已经执行完（即穷尽所有可能的任务），并停止寻找更多工作。

在消息传递模式中，当与任务相关的数据量远小于与任务相关的计算量时，通常就使用工作池模型。这种情况下，任务容易地移动而不会引起很大的数据交互开销。任务的粒度可
140 以调整，以求在负载不平衡与访问工作池的开销之间取得所要求的折中而添加和减少任务。

块调度（3.4.2节）的循环并行化或相关方法就是使用工作池模型的例子，在任务可静态获得的时候，这种工作池采用集中式映射。任务由集中式或分布式数据结构表示的并行树搜索，则是任务动态产生的情况下使用工作池模型的例子。

3.6.4 主-从模型

在主-从（master-slave）模型或*管理者-工作者*（manager-worker）模型中，一个或多个主进程产生任务并分派给工作者进程。如果管理者能估计任务的大小，或者一种随机映射能完成负载平衡的工作，任务就可以预先分配。在另一种情况下，工作者在不同时间被分派更小的任务。如果主进程产生任务很费时间，不希望所有工作者一直等待主进程生产出所有任务块，则后一种方案更适合。有些情况下，任务必须分阶段执行，而且每一阶段的任务必须在后面的任务产生前完成。这种情况下，管理者会在每一阶段后使所有工作者同步。通常，并没有人们想要的从任务到进程的预映射，使得任一工作者能执行任意分派给它的任务。管理者-工作者模型能够推广到层次或多级管理者-工作者模型，在这种模型中，顶级管理者分派大任务块给二级管理者，二级管理者再细分任务给它自己的工作者，而它们自己也可以执行部分任务。这种模型同样适用于共享地址空间模式或消息传递模式，因为交互自然是双向的；即管理者知道它要分发任务，而工作者知道它们要从管理者那接收任务。

在使用主-从模型时，一定要确保主进程不成为瓶颈，但当任务太小（或工作者相对太快）时会发生这种情况。选择任务粒度时，要确保执行任务的成本对传送任务的成本和同步的成

本占有优势。异步交互有助于与由主进程产生任务相关的交互与计算的重叠。如果工作者的请求性质是非确定性的，那么异步交互也可以减少等待时间。

3.6.5 流水线模型或生产者-消费者模型

在流水线模型（pipeline model）中，数据流通过一串进程传递，每一进程执行一个任务。在一个数据流中，同时执行不同程序称为流并行（stream parallelism）。除了发起流水线的进程外，新数据的到达触发了流水线中的一个进程执行一个新任务。这些进程可能形成各种形状的流水线，如线性或多维数组、树或一般的有圈或无圈图。流水线是生产者和消费者链。流水线中的每一个进程都可看成它前面进程数据序列的消费者和它后面进程数据的生产者。 141
流水线并不一定是线性链；它也可以是一个有向图。流水线模型通常包含从任务到进程的静态映射。

负载平衡是任务粒度的函数。粒度越大，填满流水线花费时间就越长，就是说，如果链中第一个进程产生的触发者要传播到最后一个进程，则有些进程必须等待。但是，粒度太小也会增加交互开销，因为进程在小块计算后，必须交互才能接收新的数据。适合于这种模型的最常用技术是重叠交互与计算。

并行LU分解算法是使用二维流水线的例子，在8.3.1节详细讨论。

3.6.6 混合模型

有时候，可以用多个模型解决一个问题，这就得到混合算法模型。混合模型既可由多级应用的多个模型组成，也可由在一个并行算法的不同阶段串行应用的多个模型组成。在某些情况下，算法公式可能具有多个算法模型的特点。例如，数据可能按任务依赖图描述的模式以流水线形式流动。在另一些情况下，主计算可能由一个任务依赖图来描述，但图中每一个节点可能表示一个包含多个子任务的超任务，适用于数据并行或流水线并行。并行快速排序（3.2.5节和9.4.1节）是混合模型非常好的应用。

3.7 书目评注

有很多教科书，如Wilson[Wil95]，Akl[Akl97]，Hwang和Xu[HX98]，Wilkinson和Allen[WA99]，Culler和Singh[CSG98]等，提供相似的或略有不同的并行程序编程模型，以及开发并行算法的步骤。Goedecker和Hoisie编写的书[GH01]是少数几本关于高性能并行程序编程的实用教科书之一。Kwok和Ahmad[KA99a，KA99b]对映射任务到进程的技术作了完整的综述。

本章作为例子的大部分算法在本书其他专门讨论相关问题的章中有详尽描述。读者可以参考那些章的书目评注进一步了解这些算法。 142

习题

3.1 在例3.2中，每个并集与交集运算的执行时间与两个输入表中的总记录数成比例。据此构造与图3-2和3-3对应的负载任务依赖图，图中每一节点的权值等于相应任务所需的工作量。每一图中的平均并发度是多少？

3.2 对于图3-42显示的任务图，确定下面的值：

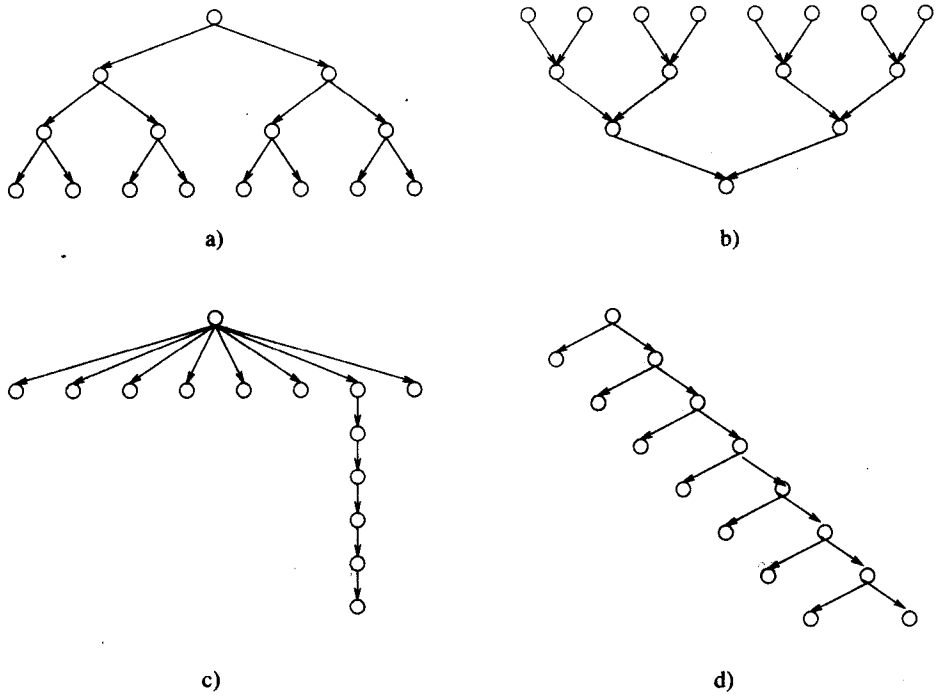
1）最大并发度。

2）关键路径长度。

3）假设进程数量不受限制，那么相对一个进程可获得的最大加速比是多少？

4）要得到最大可能的加速比，需要的最小进程数是多少？

5）假如进程数的限制分别为a）2，b）4，c）8，那么最大可能得到的加速比是多少？



143

图3-42 习题3.2中的任务依赖图

3.3 在图3-10和3-11中，与矩阵相乘分解对应的任务依赖图的平均并发度和关键路径长度是多少？

3.4 设d为任务依赖图的最大并发度，图中有t个任务，关键路径长度为l。证明$\lceil t/l \rceil \leq d \leq t-l+1$。

3.5 考虑算法3-3中所示的稠密矩阵的LU分解。图3-27显示LU分解为14个任务的情况，它基于矩阵A划分为9个块$A_{i,j}$，（$1 \leq i,j \leq 3$）的二维划分。作为因式分解的结果，A的块变为对应的L块和U块。L的对角线块是对角线元素为1的下三角子矩阵，而U的对角线元素是上三角子矩阵。任务1使用算法3-3分解子矩阵$A_{1,1}$。任务2和3实现算法3-3中第4到6行循环的块版本。任务4和5对应任务2和3的上三角部分。算法3-3中LU分解的元素版本不显示这些步骤，因为L的对角线元素是1；但是，块版本必须计算U的一个行块，作为A的块行与对应对角线块L的逆的乘积。任务6～9实现算法3-3中第7～11行循环的块版本。这样，任务1～9对应算法3-3中的最外层循环的第一个迭代的块版本。剩余的任务完成A的因式分解。试画出与图3-27所示的分解对应的任务依赖图。

3.6 列举图3-27中显示的LU分解的关键路径。

3.7 指出图3-27所示分解的任务依赖图对3个进程的一个有效映射。非形式地证明你的映射是最可能的映射。

3.8 对图3-27所示分解的任务依赖图，描述并绘制一个有效的到4个进程的映射。并证明它是对4个进程最可能的映射。

3.9 假设每一任务花费一个时间单元[⊖]，那么两种映射——对3个进程的映射或对4个进程的映射——哪个解决问题更快？

3.10 证明在图3-27中，块大小为b的块步骤1到块步骤14（即每一个$A_{i,j}$，$L_{i,j}$和$U_{i,j}$是子矩阵$b \times b$）在数学上等价于在$n \times n$矩阵A中用算法3-3，其中$n = 3b$。

提示：可对b使用归纳法。

144

3.11 图3-27显示矩阵分解为14个任务的LU因式分解，用3 × 3二维划分将矩阵分割成块。假如使用$m \times m$划分，在沿相似行的分解中，导出任务数$t\ (m)$对m的函数表达式。

提示：证明$t\ (m) = t\ (m-1) + m^2$。

3.12 对于习题3.11，导出最大并发度$d\ (m)$对m的函数表达式。

3.13 对于习题3.11，导出关键路径长度$l\ (m)$对m的函数表达式。

3.14 对于图3-2和3-3显示的数据库查询问题，请给出一个有效的分解映射。在每一种情况下，要使用的最大进程数是多少？

3.15 在算法3-4给出的算法中，假设分解使第7行的每次执行都是一个任务。画出任务依赖图和任务交互图。

算法3-4 串行问题的并行化实例

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

3.16 算法3-4中，假如$n = 16$，对于16个进程设计一个好的映射。

3.17 算法3-4中，假如$n = 16$，对于8个进程设计一个好的映射。

3.18 假如算法3-4中第3行语句变成$m = (\log_2 n) - 1$，重做习题3.15，3.16和3.17。

3.19 考虑一种简化的桶-排序。给定一个数组A作为输入，它包含n个随机的整数，范围在[1...r]之间。输出数据由r个桶组成，使得在算法的最后，桶i包含A中所有等于i的元素的下标。

- 描述以划分输入数据（即数组A）为基础的一种分解，以及到p个进程的一个相应的映射。简要说明所得的并行算法如何工作。
- 描述以划分输出数据（即r个桶的集合）为基础的一种分解，以及到p个进程的一个相应的映射。简要说明所得的并行算法如何工作。

145

⊖ 事实上，对于块大小>>1，任务1，10和14需要大约$2/3b^3$个算术运算；任务2，3，4，5，11和12需要大约b^3个运算；任务6，7，8，9和13需要大约$2b^3$个运算。

3.20 在习题3.19中，哪一种分解能得到更好的并行算法？n和r的相对值对选用两种分解方案中的哪一种有影响？

3.21 如果7个任务的运行时间分别为1，2，3，4，5，5和10个时间单元。假如分派工作到进程不花时间，对于两个进程动态映射的集中式方案，分别计算最好和最差的加速比。

3.22 假设采用集中式动态负载平衡方案对M个任务进行映射，关于这些任务有如下的信息：

- 平均任务大小为1。
- 最小任务大小为0。
- 最大任务大小为m。
- 进程拾取一个任务要花Δ时间。

假设可以获得l ($l < m$)批成批任务，计算自调度和块调度方法的最好和最差的加速比。当
146 $p = 10$，$\Delta = 0.2$，$m = 20$，$M = 100$和$l = 2$时，这两种调度方法实际的最好和最差的加速比是多少？

第4章 基本通信操作

在多数并行算法中，进程间需要交换数据。这些数据的交换常常对并行程序的效率产生重大影响，因为这些交换会在程序的执行过程中引起交互延迟。例如，回忆2.5节，在采用直通路由选择的互连网络中，运行在两个不同节点的两个进程间进行一次m字消息的简单交换所需的时间大约为$t_s + mt_w$。其中t_s是数据传送的延迟或启动时间，t_w是每字传送时间，它与节点间的可用网络带宽成反比。在许多定义得完善的模式里，实际并行程序中的交互包含的进程数远远不止两个。通常，要么所有进程都参与一个全局的交互操作，要么进程的一些子集参与各自子集的本地交互操作。这些常见的进程间的基本交互或通信模式常常作为构件用于各种并行算法中。在各种并行体系结构中，正确实现这些基本通信操作是有效利用并行算法的关键。

我们在这一章中介绍一些算法，这些算法实现简单互连网络中常用的通信模式，例如线性阵列、二维格网和超立方体。选择这些互连网络的目的是为了教学。譬如说，尽管大规模的并行计算机不大可能基于线性阵列或环形拓扑结构，但是理解线性阵列下的各种通信操作是非常重要的，因为格网的行和列就是线性阵列。对网状拓扑按行展开或者按列展开的并行算法就是应用线性阵列的算法。在格网中使用通信操作的算法只不过是相应的线性阵列算法在二维上的简单推广。而且，使用如像数组这样的规则数据结构的并行算法常常可以自然地映射到一维或者二维进程的数组。这也使得在线性阵列或格网互连的网络中，研究进程间的交互更加重要。另一方面，超立方体体系结构是很有意义的，因为许多使用递归交互模式的算法可以自然地映射到超立方体拓扑结构。这些算法的大部分可以很好的运行在非超立方体结构的网络中，但在超立方体上实现这种通信模式会更简单。

实际上，虽然多数的现代并行计算机不可能和本章叙述的任一互连网络完全匹配，但本章讲述的简单网络结构中的算法对现代并行计算机是实际的和高度适用的。这是因为在现在的并行计算机中，在两个节点之间传送一定大小的数据所需时间通常与节点在网络中的相对位置无关。这种同质性是由各种固件特性和硬件特性决定的，如随机路由选择算法和直通路由选择算法等。而且，终端用户通常不能显式控制进程到处理器之间的映射。因此，我们假定在互连网络中，任意一对节点间传送m字的数据引起$t_s + mt_w$的时间开销。在大多数体系结构中，只要数据传送的源节点和目标节点间有一条可用的通信链路，这一假定是相当准确的。然而，当有多对节点同时通信时，这些消息的传送将会占用更长的时间。如果通过网络截面的消息传递数量超过网络截面的带宽（参看2.4.4节），就会产生这种消息延迟。在这种情况下，我们需要调整t_w的值来反映由网络拥塞带来的延迟。如2.5.1节中所述，我们把调整以后的t_w作为一个有效的t_w。当遇到在某些网络中引起拥塞的通信操作时我们将在书中作出说明。

如2.5.2节中所述，在共享地址空间模式的各个处理器间，数据共享的开销可以用一个相同的表达式$t_s + mt_w$表示，通常对于并行计算机的不同处理器和不同的计算速度，t_s和t_w有不同的值。因此，在这一章的讨论中，可以假定需要一个或者多个交互模式的并行算法的成本

和消息传递模式中导出的成本表达式接近。

在下面的几节中，我们将介绍各种通信操作，并推导相应时间复杂度的表达式。我们假定互连网络支持直通路由选择（2.5.1节），并且，任何一对节点间的通信时间与它们之间的通信路径上的中间节点数无关。同时，我们若假定通信链路是双向的；也就是说，两个直接相连的节点可以在$t_s + mt_w$时间内同时相互发送m字的消息。我们假定采用单端口通信模式，其中一个节点一次只能在它的一条链路上发送消息。同样，一个节点一次只能在一条链路上接
148 收消息。但是，一个节点同时在同一条链路或另一条链路发送消息时可以接收消息。

这里描述的许多操作都有对偶性和其他的相关操作，这些操作可以通过使用与原来操作非常类似的过程来实现。通信操作的对偶是原始操作的逆操作，可以通过逆转原来操作中的通信方向和消息序列来完成。我们将在下面可能应用的地方讨论这些操作。

4.1 一对多广播以及多对一归约

并行算法常常需要一个进程发送相同的数据给其他所有的进程或其他所有进程的子集。这种操作称为*一对多广播*（one-to-all broadcast）。开始时，只有源进程具有需要广播的m字的数据。广播结束时，就会有p个原始数据的副本——每个进程一个。一对多广播的对偶是*多对一归约*（all-to-one reduction）。在多对一归约操作中，p个参与进程的每一个都有一个缓冲区M，它包含m个字。来自所有进程的数据通过一个相关的操作符组合起来，并被累加到一个目标进程中一个m字的缓冲区中。归约操作可以用来求一些数字集的和、乘积、最大值和最小值——累加结果M中的第i个字是每个原缓冲区中第i个字的和、乘积、最大值和最小值。图4-1显示p个进程之间的一对多广播和多对一归约操作。

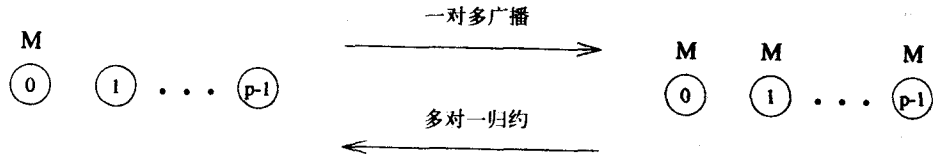


图4-1 一对多广播和多对一归约

一对多广播和多对一归约操作被用于几种重要的并行算法中，包括矩阵-向量乘法、高斯消去法、最短路径以及向量内积。在下面几小节中，我们将讨论在多种互连网络拓扑中实现一对多广播。

4.1.1 环或线性阵列

连续地从源进程向其他$p-1$个进程发送$p-1$个消息是一种比较简单的实现一对多广播的方法。但是，由于源进程成为瓶颈，这种方法的效率不高。而且，因为每次只有源节点和一个目标节点连接，通信网络的利用率非常低。利用一种称为*递归加倍*（recursive doubling）的技
149 术，可以设计一个比较好的广播算法。递归加倍的源进程首先发送消息给另外一个进程。然后，这两个进程可以同时发送消息给还在等待消息的其他两个进程。继续这一过程，直到所有进程都收到了数据，这样消息可以在$\log p$步广播完毕。

图4-2显示在一个8节点的线性阵列或环中，一对多广播的步骤。节点的标号从0至7。每个消息传送步骤也用一个编号的虚线箭头表示，箭头从消息的源节点指向目标节点。表示同

一时间发送消息的箭头有同样的标号。

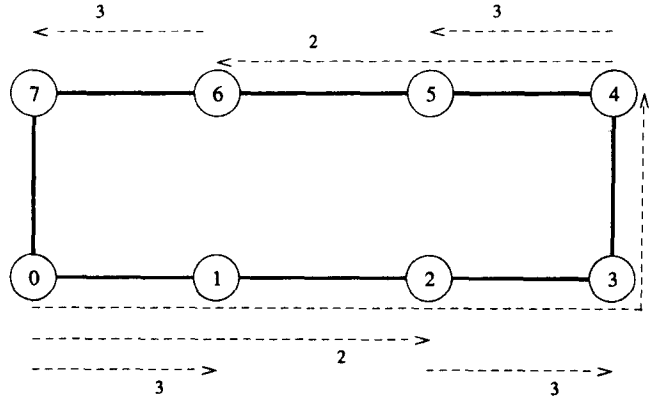


图4-2 8节点环上的一对多广播。节点0是广播中的源节点。每个消息传送步骤用一个编号的从源节点到目标节点的虚线箭头表示。箭头上的数字表示消息发送的时间步骤

要注意的是，在一个线性阵列中，每一步都要仔细地选择消息将要发送到的目标节点。在图4-2中，消息首先从源节点0发送到最远的节点4。第二步中，发送和接收节点间的距离减半，依此类推。在每一步中都采用这种方式来选择消息的接收者，以避免网络的拥塞。例如，如果在第一步从节点0把消息发送给节点1，然后在第二步节点0和节点1试图分别发送消息给节点2和节点3，那么节点1和节点2间的通信链路将会出现拥塞，因为这条通信链路是第二步中两条消息要通过的最短路径的一部分。

如图4-3所示，线性阵列中的归约操作可以通过简单地反转广播操作的发送方向和消息序列的顺序实现。第一步每个奇数号节点把它们缓冲区中的数据发送到前面一个偶数号的节点，并在偶数号节点中把缓冲区中数据合并成一个。当第一步完成后，只有节点0，2，4，6的4个缓冲区的数据需要归约。第二步把节点0和2的数据累加到节点0，而节点6和4的数据累加到节点4。最后，节点4把它的缓冲区中的数据发送到节点0，在节点0中计算最终的归约结果。

150

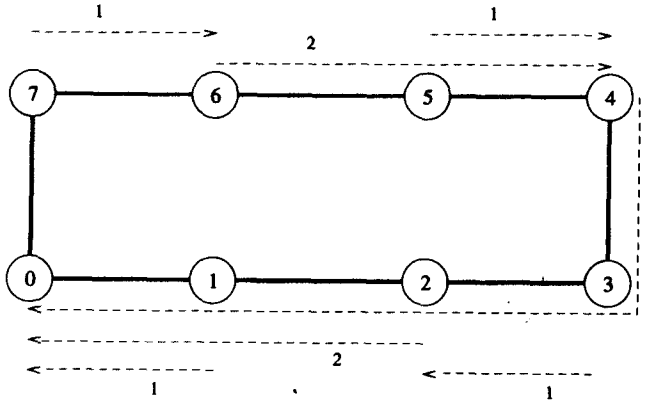


图4-3 在8节点环上的归约操作，节点0是归约的目标节点

例4.1 矩阵-向量的乘法

考虑一个$n \times n$矩阵A和一个$n \times 1$向量x的相乘问题，计算在一个$n \times n$的节点格网中进行，产生一个$n \times 1$结果向量y。算法8-1显示求解这一问题的一系列算法。图4-4显示矩阵和向量的一个可能的映射，其中矩阵的每个元素属于不同的进程，而且向量分布在格网最顶端的行中，结果向量由进程最左边的列产生。

由于矩阵的所有行必须和向量相乘，每个进程都需要驻留在其列中最顶层进程的向量的元素。因此，在计算矩阵-向量的乘积之前，节点中的每一列执行向量元素的一对多广播，以列中最上面的进程作为广播源。在进行这一操作时，我们可以把该$n \times n$格网的每一列看成一个n个节点的线性阵列，并对前面讨论的所有列同时应用线性阵列广播过程。

在广播结束以后，每个进程把它的矩阵元素和广播的结果相乘。此时，每一行进程需要把它的结果相加产生乘积向量的相应元素。这一过程需要对进程格网的每一行执行多对一归约并把每一行的第一个进程作为归约操作的目标来完成。

例如，P_9从P_1接收$x[1]$作为广播的结果，再把$x[1]$和$A[2, 1]$相乘，并参与和P_8、P_{10}及P_{11}的
151 多对一归约，并在P_8累积$y[2]$。 ■

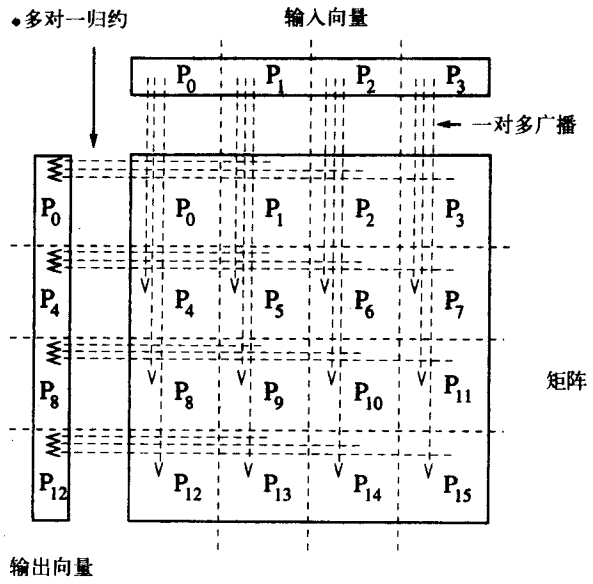


图4-4 在一个4 × 4矩阵和4 × 1向量的乘法中的一对多广播和多对一归约

4.1.2 格网

我们可以把一个具有p个节点的方形格网的行或者列看作一个有$\sqrt{p}$个节点的线性阵列。这样，格网中的许多通信算法仅仅是一些线性阵列通信算法的简单推广。一个线性阵列的通信操作可以在一个格网中分两个阶段来执行。第一阶段，可以将格网中的行看作是线性阵列，将操作沿着一行或所有行进行。第二阶段，再对列进行同样的操作。

考虑在有$\sqrt{p}$行和$\sqrt{p}$列的二维方形格网中进行的一对多广播。首先，在每一行中执行一对多广播，从源点广播到同一行的余下的$\sqrt{p}-1$个节点。一旦格网中的一行的所有的节点都

已经收到了数据，它们就在各自的列上启动一对多广播。当第二阶段结束时，格网中的每一个节点都会具有初始消息的一个副本。在格网中进行一对多广播的通信步骤显示在图4-5中，其中$p=16$，源节点为左下角的节点0。步骤1和步骤2对应于第一阶段，步骤3和步骤4对应于第二阶段。

三维格网中的一对多广播也是一个相似的过程。在三维格网中，每一个具有$p^{1/3}$个节点的行都可以被看成一个线性阵列。就像在线性阵列一样，在二维和三维网络中可以执行归约，只需反转消息的方向和顺序。 152

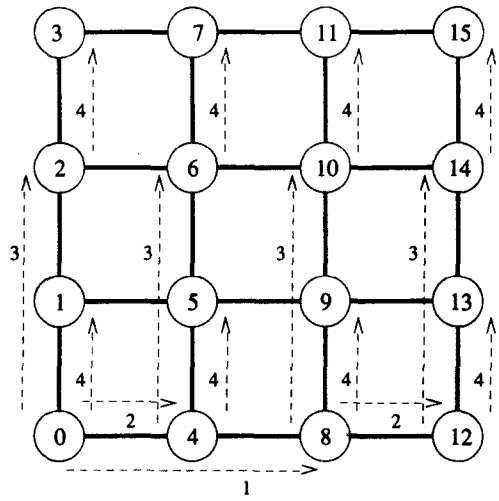


图4-5 在16节点格网中的一对多广播

4.1.3 超立方体

在上一小节说明二维格网上的一对多广播分成两个阶段，每一个阶段的通信沿不同的维进行。同样，在一个三维格网上进行一对多广播要分三个阶段进行。如果有一个超立方体，它具有2^d个节点，那么可以将它看作是一个d维的格网，每个维上有两个节点。因此，格网算法能够扩展到超立方体，唯一不同的是，在超立方体中的通信需要分d步来进行，每一步对应于每一维。

图4-6中显示在8节点超立方体中进行的一对多广播，其中节点0是源节点。在图中，通信沿最高维开始（即由节点标号二进制表示的最高有效位确定的维），并在随着的步骤中，沿逐步降低的维数进行。值得注意的是，图4-6中，算法从源节点到目标节点的三个通信步骤，与图4-2所示的线性阵列中的广播算法一样。但是，在一个超立方体中，通信中选择维的顺序并不影响最后的输出结果。图4-6中只显示这些顺序中的一种。和线性阵列不同，如果节点0在第一步将消息发送给节点1，紧接着节点0和1将消息分别发送给节点2和3，最后，节点0，1，2，3再将消息分别发送给节点4，5，6，7，那么超立方体广播不会出现拥塞。

4.1.4 平衡二叉树

一对多广播的超立方体算法自然地映射到平衡二叉树，在平衡二叉树中，每一个叶子是处理节点，每一个中间节点是开关单元。图4-7显示8个节点的情形。图4-7中的通信节点和图 153

4-6中所示的超立方体算法中的节点有相同的标号。图4-7表明，所有通信链路在任何时刻都不会出现拥塞。图4-7还说明，在超立方体中和树中的通信之间的区别在于，树中沿不同路径有不同数目的开关节点。

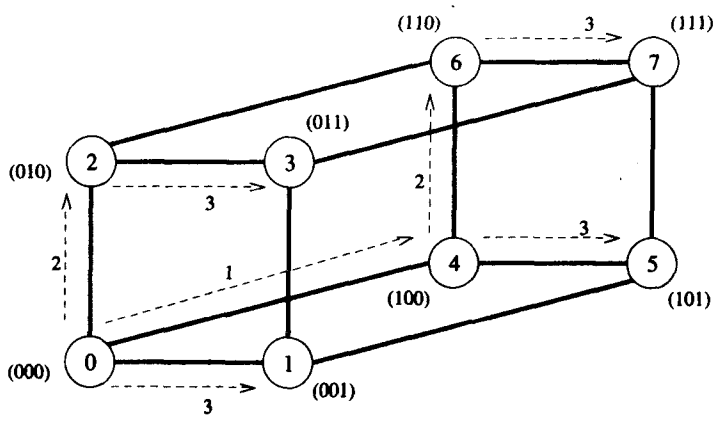


图4-6 在一个三维超立方体上的一对多广播。节点标号的二进制表示显示在括号中

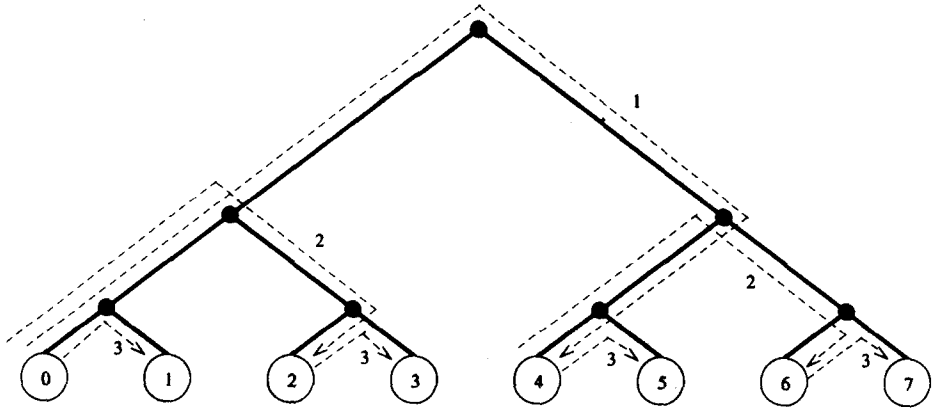


图4-7 在8节点树上的一对多广播

4.1.5 算法细节

仔细观察图4-2、图4-5、图4-6和图4-7会发现，这一节的4种互连网络中，一对多广播所采用的基本通信模式是相同的。现在我们说明广播和归约操作的实现过程。为简单起见，这里是在超立方体结构中描述算法并假定通信进程的数目是2的幂。但是，这些算法适用于任何网络拓扑，同时可以很容易推广到任意进程数目（习题4.1）。

算法4-1显示在一个2^d节点网络上一对多广播的过程，其中节点0是广播的源节点。这个过程将在所有节点上执行。在任何节点上，*my_id*的值就是节点的标号。令X为要广播的消息，它最初驻留在节点0上。这个过程执行d步通信，每步沿假定的超立方体上的一个维。在算法4-1中，通信从最高维向最低维进行（实际上这一过程中选择的维数顺序对结果没有影响）。循环计数器i显示超立方体中当前正在进行通信的维。只有节点标号的i位最低有效位全为0的

节点才参与沿第i维的通信。例如，在图4-6所示的三维超立方体中，在第一步中i等于2。因此，只有节点0和4通信，因为它们的最低两位有效位为0。在下一步中，当$i=1$时，最低有效位为0的所有节点（即节点0，2，4和6）参与通信。当沿所有的维完成通信后，这个过程结束。 154

变量*mask*用来帮助判断在循环的一次特定迭代中有哪些节点通信。变量*mask*有d (= log n)位，所有位最初被置为1(第3行)。在每次循环开始时，*mask*的最高有效非0位被置0(第5行)。第6行决定外层循环的当前迭代中参与通信的节点。例如，在图4-6中的超立方体中，*mask*最初被置为111，在i = 2(*mask*的i位最低有效位为1)的迭代中*mask*被置为011。第6行中的AND操作只选择i位最低有效位为0的那些节点。

在沿第i维选出的通信的节点中，第i位为0的节点发送数据，而第i位为1的节点接收数据。第7行代码用来判定接收和发送节点。例如，在图4-6中，在对应于i = 2的迭代内，节点0(000)是发送节点，而节点4(100) 是接收节点。同样，当i = 1时，节点0(000)和4(100)是发送节点，而节点2(010)和6(110) 是接收节点。

只有当节点0是广播源时，算法4-1才能执行。对于任意源的广播，在调用这个过程前，必须把假定的超立方体中的每个节点的标号和源节点的标号进行XOR操作，对节点重新编号。算法4-2为修改后的一对多广播算法，其中源节点是0和$p-1$中的任何节点。通过第3行的XOR操作，算法4-2把源节点重新标号为0号，并对相对于源节点的其他节点重新编号。重新编号以后，可以用算法4-1来进行广播。 155

算法4-1 在一个d 维p 节点超立方体的节点0上进行消息X的一对多广播，其中 d = log p 。AND和XOR分别为按位逻辑与和按位逻辑异或操作

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

算法4-3给出在假设的超立方体上执行多对一归约的过程，其最终结果累积在节点0中。单节点的累加是一对多广播的对偶。因此，只要反转原来一对多广播的消息的顺序和方向，就能得到实现归约所需的通信模式。算法4-3中显示的过程ALL_TO_ONE_REDUCE(*d*, *my_id*, *m*, *X*, *sum*) 和算法4-1显示的过程ONE_TO_ALL_BC(*d*, *my_id*, *X*) 非常相似。一个差别在于，多对一归约中的通信从最低维向最高维进行。这个改变反映在算法4-3对变量*mask*和i的操作中。另外，在一对通信节点中如何判定源节点和目标节点的准则也刚好相反(第7行)。除了这

些不同点外，过程ALL_TO_ONE_REDUCE还有一些附加的代码（第13和14行）用以将每次迭代中接收的消息加起来(可以用其他任何结合运算来代替加法运算)。

4.1.6 成本分析

对一对多广播操作和多对一归约进行成本分析是非常简单的。假设有p个进程参加广播或者归约通信，消息长度为m个字。广播或归约过程将涉及$\log p$个点到点的简单消息传送，每一次的时间开销为$t_s + t_w m$。因此，过程的总时间为：

156

$$T = (t_s + t_w m) \log p \tag{4-1}$$

算法4-2 在假想的d维超立方体上由源节点发起的对消息X的一对多广播。AND和XOR是按位逻辑操作

```
1.  procedure GENERAL_ONE_TO_ALL_BC(d, my_id, source, X)
2.  begin
3.      my_virtual_id := my_id XOR source;
4.      mask := 2^d − 1;
5.      for i := d − 1 downto 0 do     /* Outer loop */
6.          mask := mask XOR 2^i;   /* Set bit i of mask to 0 */
7.          if (my_virtual_id AND mask) = 0 then
8.              if (my_virtual_id AND 2^i) = 0 then
9.                  virtual_dest := my_virtual_id XOR 2^i;
10.                 send X to (virtual_dest XOR source);
        /* Convert virtual_dest to the label of the physical destination */
11.             else
12.                 virtual_source := my_virtual_id XOR 2^i;
13.                 receive X from (virtual_source XOR source);
        /* Convert virtual_source to the label of the physical source */
14.             endelse;
15.     endfor;
16. end GENERAL_ONE_TO_ALL_BC
```

4.2 多对多广播和归约

多对多广播（all-to-all broadcast）是一对多广播的推广，其中所有p个节点同时发起一个广播。虽然一个进程发送相同的m字消息给其他每个进程，但是不同的进程可以广播不同的消息。多对多广播用于矩阵运算中，包括矩阵相乘和矩阵-向量相乘。多对多广播的对偶是多对多归约（all-to-all reduction），其中每个节点是多对一归约的目标节点(习题4.8)。图4-8说明多对多广播和多对多归约。

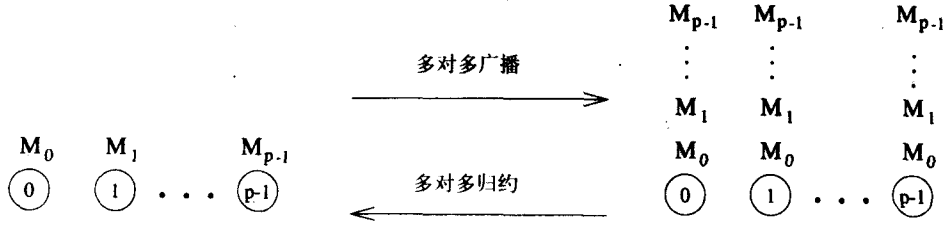


图4-8 多对多广播与多对多归约

算法4-3 在一个d 维超立方体上的单节点累积。每个节点都提供一个长度为m字的消息X，节点0是和的目标节点。AND和XOR是按位逻辑操作

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

执行多对多广播的一种方法是执行p个一对多广播，每个节点启动一个一对多广播。在简 157
单执行的情况下，在一些常见的结构中，采用这种方法进行多对多广播的开销是一对多广播的p倍。如果同时进行p个一对多广播，并把所有需要在同一条路径上传送的消息链接成一个长度为各个消息长度总和的消息进行发送，就能更加充分地利用互连网络的通信链路。

下面几小节将描述在线性阵列、格网以及超立方体拓扑结构中进行多对多广播。

4.2.1 线性阵列和环

在线性阵列或者环中执行多对多广播时，在整个通信完成以前，网络中所有的通信链路可以一直保持繁忙状态，因为每个节点与它的相邻节点之间总有某些信息需要传递。每个节点都首先把需要广播的数据发送给它的相邻节点之一。下一步，从它相邻节点之一把接收的数据转发给其他相邻节点。

图4-9中显示8节点环中的多对多广播。在具有双向链路的线性阵列中，可以执行同样的过程。与以前的图一样，图中箭头上的整数标号表示消息发送过程的第几步。在多对多广播中，p个不同的消息在一个p节点的网络中循环。在图4-9中，每个消息由它的初始源节点识别， 158
初始源节点的编号和步骤放在括号里。例如，节点0和节点1间的弧上的标号2(7)，表示第2步中通信的数据，是节点0上一步从节点7收到的。如图4-9所示，如果消息通信是在单方向上循环执行，那么每个节点会在$p-1$步之内从其他节点接收到全部$p-1$个信息。

算法4-4给出了p节点环上的多对多广播过程。在每个节点中，将要广播的初始消息在本地命名为*my_msg*。当过程结束时，每个节点把p个消息集合存储在*result*中。如程序所示，格

网上的多对多广播两次应用线性阵列过程，其中一次沿格网的行，另一次沿格网列。

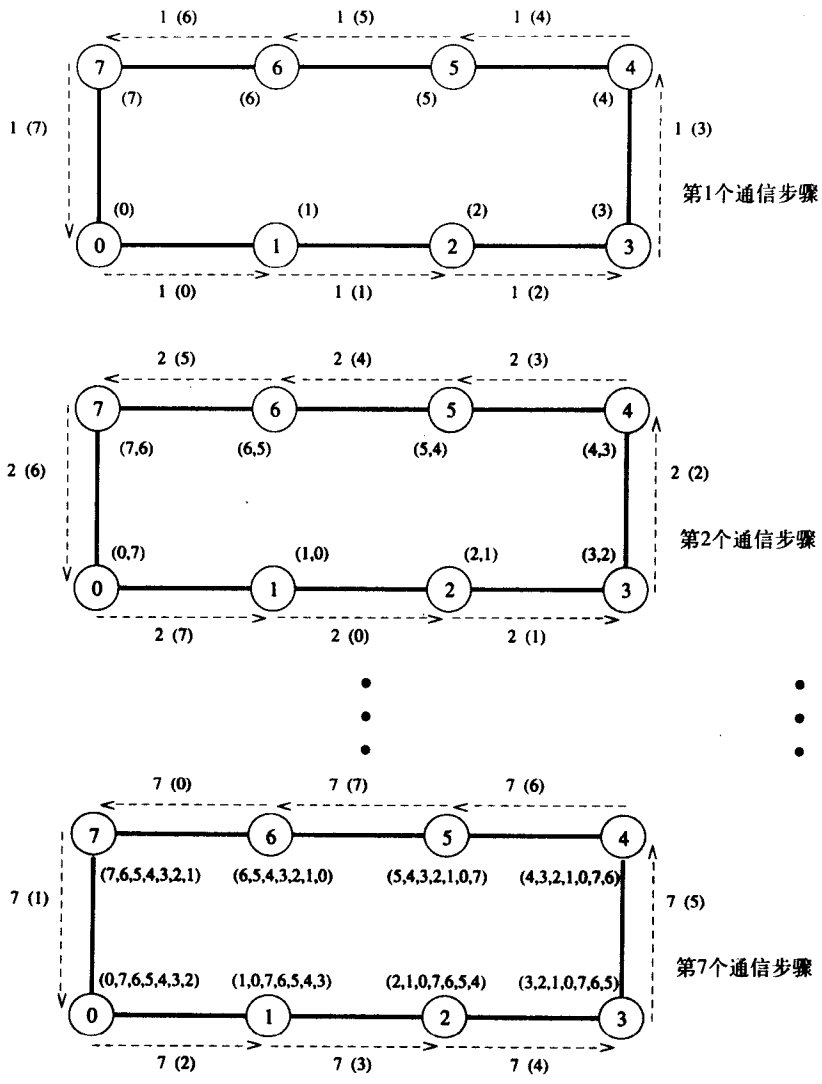


图4-9 8节点环上的多对多广播

注：每个箭头上的标号显示通信步骤，它后面括号里的标号表示在广播开始之前拥有当前被传送消息的节点。在每个节点后面括号里的整数序列是节点的标号，这些节点中的数据在当前通信步骤进行前已经接收了数据。图中只显示第一、第二和最后一个通信步骤。

159

多对多归约是多对多广播的对偶，归约操作开始时，每个节点有p个消息，每个消息被累加到某个节点。多对多归约可以通过反转消息的方向和顺序来实现。例如，8节点的环中的多对多归约的第一通信步骤对应图4-9中的最后一步，节点0向节点7发送消息$msg[1]$，而不是从节点7接收消息。在归约操作中，接收消息时，唯一的附加步骤是，在向其他相邻节点转发组合消息前，把接收到的消息和与接收到的消息有相同目标的本地消息副本进行合并，并作为接收的消息。算法4-5给出p节点环上多对多归约的过程。

算法4-4 p 节点环上的多对多广播

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

算法4-5 p 节点环上的多对多归约

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

4.2.2 格网

和一对多广播一样，2-D格网上的多对多广播算法也是基于线性阵列算法，同样把格网的行和列作为线性阵列。通信过程也分成两阶段进行。第一阶段，格网中的每一行执行一次线性阵列形式的多对多广播。在这一阶段里，每个节点从它们各自所属的具有$\sqrt{p}$个节点的行上 160
收集$\sqrt{p}$个消息。每个节点把这些收集到的消息聚合成一个大小为$m\sqrt{p}$的消息，然后进行算法的第二通信阶段。第二通信阶段按列对合并后的消息执行多对多广播。当这一阶段完成时，每个节点获得p个m字的数据，这些数据原来驻留在不同的节点上。在一个3×3的格网中，算法的第一和第二阶段开始时的数据分布情况如图4-10所示。

算法4-6给出$\sqrt{p} \times \sqrt{p}$格网上多对多广播的过程。多对多归约的格网过程留作读者的练习(习题4.4)。

4.2.3 超立方体

超立方体上的多对多广播算法是把格网算法扩展到$\log p$维，所以超立方体上的多对多广播需要分$\log p$步进行。通信的每一步沿p节点超立方体的不同维进行。每一步有多对节点交换

它们的数据，而且加倍在下一步要传送消息的长度，把接收到的消息和节点本身的数据连接起来。图4-11显示在一个具有双向通信通道的8节点超立方体中进行上述通信的步骤。

算法4-6 p 节点方形格网上的多对多广播

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

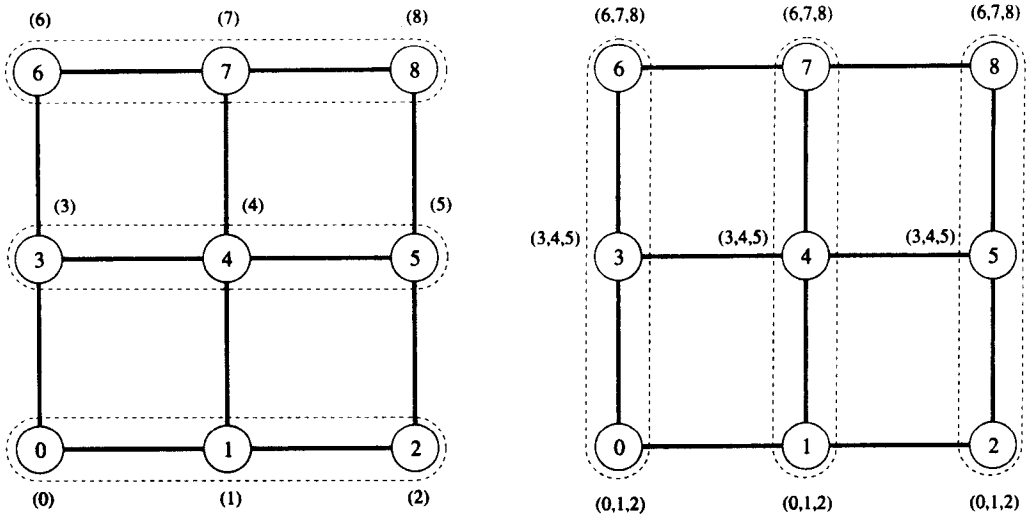
/* Communication along rows */
3.      left := my_id − (my_id mod √p) + (my_id − 1)mod√p;
4.      right := my_id − (my_id mod √p) + (my_id + 1) mod √p;
5.      result := my_msg;
6.      msg := result;
7.      for i := 1 to √p − 1 do
8.          send msg to right;
9.          receive msg from left;
10.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

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

算法4-7是在d维超立方体上多对多广播的实现过程。通信从超立方体的最低维开始，连续向高维方向进行(第4行)。在每次迭代中节点成对地进行通信，所以，在第i步迭代中，以二进制表示的相互通信的两个节点的标号，其第i位最低有效位不同(第5行)。在一次迭代的通信步骤后，每个节点把它接收到的数据与它自身的数据连接起来(第8行)。这个连接后的消息在下一次迭代中传送。

算法4-7 d 维超立方体上的多对多广播

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



a) 初始数据分布　　b) 按行广播后的数据分布

图4-10　3 × 3格网上的多对多广播

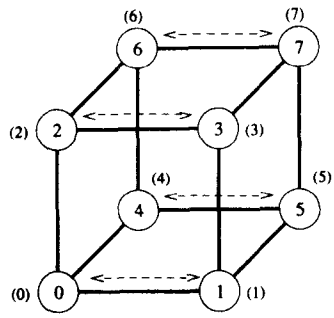
注：每一阶段中相互通信的节点包含在虚线内。在第二阶段结束时，所有节点都接收到(0, 1, 2, 3, 4, 5, 6, 7)(即来自每个节点的消息)。

算法4-8　d 维超立方体上的多对多广播。AND和XOR分别是按位逻辑与和按位异或操作

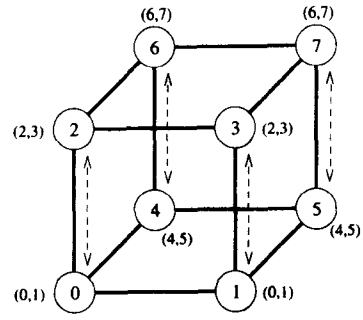
```
1.  procedure ALL_TO_ALL_RED_HCUBE(my_id, msg, d, result)
2.  begin
3.      recloc := 0;
4.      for i := d − 1 to 0 do
5.          partner := my_id XOR 2^i;
6.          j := my_id AND 2^i;
7.          k := (my_id XOR 2^i) AND 2^i;
8.          senloc := recloc + k;
9.          recloc := recloc +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

如通常那样，通过反转对多对多广播消息的方向和顺序，可以导出多对多归约算法。而且，归约操作要在缓冲区中选择合适的子集进行发送，并在每次迭代中累积接收到的消息，而不是像广播操作那样只对消息进行连接。算法4-8给出*d*维超立方体上多对多归约的过程。每次迭代中，算法用*senloc*指向即将发送的消息的开始位置，用*recloc*指向累积接收到的消息的位置。

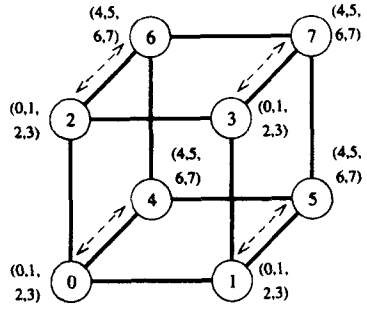
161 ~ 163



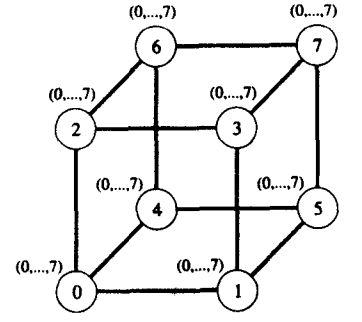
a) 消息的初始分布



b) 第2步之前的分布



c) 第3步之前的分布



d) 消息的最后分布

图4-11 8节点超立方体上的多对多广播

4.2.4 成本分析

在环或者线性阵列中，多对多广播包含最近的相邻节点间的$p-1$步通信。每一步通信中的消息长度为m，用时$t_s + t_w m$。因此，整个操作用时为

$$T = (t_s + t_w m)(p-1) \tag{4-2}$$

同样，在格网中，第一阶段在时间$(t_s + t_w m)(\sqrt{p}-1)$内进行$\sqrt{p}$个多对多广播(每个广播在
164 $\sqrt{p}$个节点之间进行)。第二阶段参加多对多广播的节点数依然是$\sqrt{p}$个，但是每个消息的长度变为$m\sqrt{p}$。因此，本阶段的通信时间是$(t_s + t_w m\sqrt{p})(\sqrt{p}-1)$。在$p$节点的二维方形格网中进行多对多广播操作的总时间开销为每个阶段所用时间的和，即

$$T = 2t_s(\sqrt{p}-1) + t_w m(p-1) \tag{4-3}$$

在p节点的超立方体中，一共将进行log p步通信操作，第i步中交换的消息长度为$2^{i-1}m$。在第i步，一对节点相互发送和接收消息将耗时$t_s + 2^{i-1}t_w$。因此，完成整个过程的时间为

$$\begin{aligned} T &= \sum_{i=1}^{\log p}(t_s + 2^{i-1}t_w m) \\ &= t_s \log p + t_w m(p-1) \end{aligned} \tag{4-4}$$

公式4-2、4-3和4-4说明，对所有体系结构，在多对多广播通信时间表达式中和t_w有关的项是$t_w m(p-1)$。在每个节点一次只能有一个端口通信的并行计算机上，这一项同时也是多对多广播通信时间的下界。这是因为无论在何种体系结构中，每个节点接收至少长度为$m(p-1)$个字的数据。所以，对于大消息来说，像超立方体这样高度连接的网络，进行多对多广播或多对多归约时，其性能并不优于简单的环形结构。事实上，在环这样简单结构中的直接多对多广播算法更有实际意义。仔细考查这一算法看出，它是p次一对多广播的序列，只是每次的源节点不同。这些广播都以流水线方式进行的，所以可在一共p步最相邻节点间的通信步骤内完成。许多并行算法都涉及一系列不同源节点的一对多广播，其中还会伴随着一些计算。如果每个一对多广播都采用4.1.3节中的超立方体算法进行，那么n次广播将耗时$n(t_s + t_w m)\log p$。另一方面，如果用如图4-9所示的方式进行流水线广播，那么通信的时间不会超过$(t_s + t_w m)(p-1)$，只要所有广播的源节点不同，并且$n \leqslant p$。在后面几章，我们将介绍流水线广播如何提高一些并行算法的性能，例如高斯消元法(8.3.1节)、回代法(8.3.3节)以及寻找图中最短路径的Floyd算法(10.4.2节)。

多对多广播还有一个值得注意的性质，与一对多广播不同，超立方体中的算法不能不作修改地应用于环和格网结构中。这是因为如果将超立方体中的多对多广播过程应用于具有相同节点数但维度较小的网络中，就可能在通信链路中引起拥塞。例如，图4-12显示在环中进行的超立方体多对多广播操作执行到第三步时(图4-11c)的结果。所有4个消息都将在环的一条链路上通过，所耗费时间为完成通信步骤时间的4倍。 165

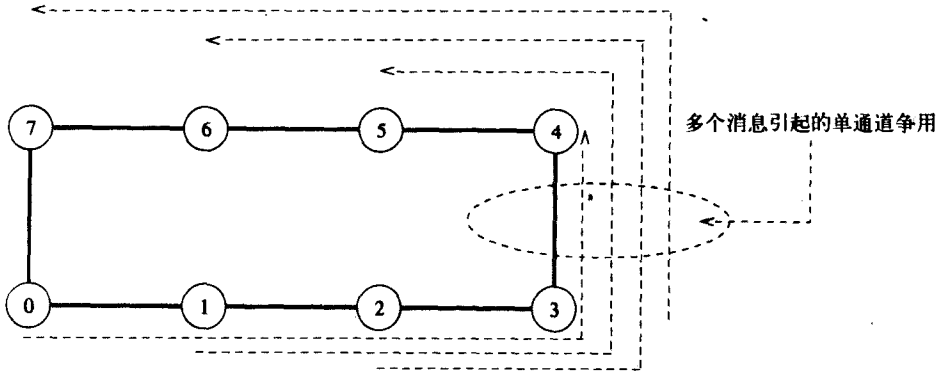


图4-12 把图4-11 c 中的超立方体通信映射到环上时出现的通道争用

4.3 全归约与前缀和操作

多对多广播通信模式也可用来执行一些其他操作。其中一种操作是归约操作的第三种变体，操作开始时每个节点有一个m字的缓冲区，操作结束时每个节点的缓冲区的长度同为m，这些缓冲区是用一个结合操作符对原来p个缓冲区进行组合得到的。从语义上讲，这个操作通常称为全归约（all-reduce）操作，它等同于先进行一个多对一归约，再进行一个一对多广播。这个操作与多对多归约不同，多对多归约是p个多对一归约同时进行，并且每个操作都有不同的目标节点。

每个节点上的单字消息全归约操作常被用在消息传递计算机上实现障碍同步。归约操作的语义是，当执行某个并行程序时，节点在提供一个数据前，归约操作不可能结束。

进行全归约的一个简单方法是首先进行一个多对一归约，接着进行一次一对多广播。但

是，使用多对多广播通信模式，可以更快地进行全归约操作。图4-11对一个8节点超立方体说明这一算法。假设图中括号内的整数不代表消息，而是代表将要和原来驻留在标有整数标号的节点中的数进行加法运算的数。为了进行归约，我们按照多对多广播过程的通信步骤进行，只是在每一步结束时，对两个数相加而不是连接两个消息。在归约过程终结时，每个节点保存和 (0 +1 +2+ … +7) (而不是像多对多广播中那样的8个从0到7的消息)。与多对多广播不同，
166 归约操作中每个被传送的消息只有一个字。消息的长度并没有在每一步中加倍，因为对数字进行相加而不是进行连接。因此，所有p步通信的总时间为

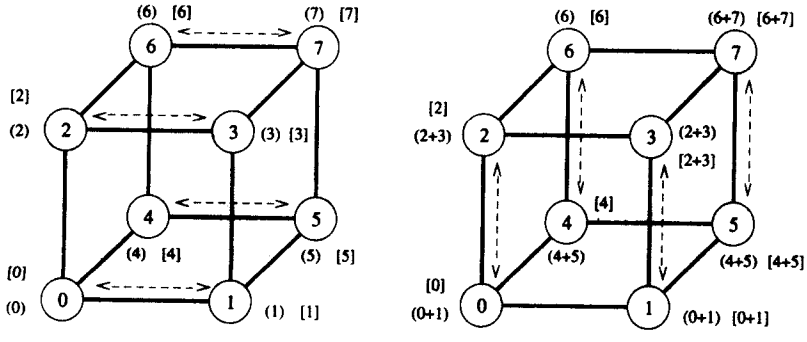
$$T = (t_s + t_w m)\log p \tag{4-5}$$

如果*my_msg*、*msg*和*result*都是数字(而不是消息)，且把第8行的并操作(‘∪’)用加法代替，则算法4-7可用来对p个数字求和。

求前缀和（prefix sum）（也称为*扫描*(scan)操作）是另一个重要问题，它也可以用类似多对多广播和多归约操作的通信模式解决。给出p个数字$n_0, n_1, \ldots, n_{p-1}$（每个节点一个），问题是对所有$k$计算 $s_k = \sum_{i=0}^{k} n_i$，其中$k = 0, 1, \ldots, p-1$。例如，如果原始数字序列为 (3, 1, 4, 0, 2)，那么前缀和序列为 (3, 4, 8, 8, 10)。开始时，n_k驻留在节点k中，过程结束后，节点k中保存s_k。每个节点可从一个大小为m的缓冲区或者向量开始，而不仅仅是一个数，最后m字的结果将是原来缓冲区中的相应单元的和。

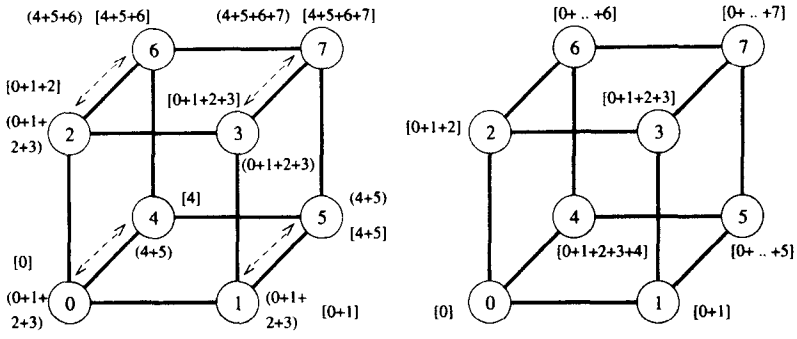
图4-13显示一个8节点超立方体的求前缀和过程。这个图是在图4-11的基础上修改得到的。修改的原因是为了适应这样一个事实，在前缀和操作中，节点k只用到一些节点的k节点子集中的信息，这些节点的标号小于或等于k。为了累加正确的前缀和，每个节点维护一个额外的结果缓冲区。这一缓冲区在图4-13中用方括号表示。在一个通信步骤的终点，只有当消息源节点的标号小于接收节点的标号时，接收到的消息内容才加到结果缓冲区中。就像在全归约操作中一样，将要发送的消息内容（在图中用括号表示）用每个接收到的消息进行更新。例如，第一步通信后，节点0、2、4并不把从节点1、3、5中接收到的数据加到它们的结果缓冲区中。但是，将下一步要发送的消息内容更新。

由于不是把一个节点接收的所有消息都提交给最终结果，节点接收到的某些消息可能是冗余的。在图4-13中，我们已经从标准的多对多广播通信模式中略去了这些步骤，尽管这些步骤的存在与否并不影响算法的最终结果。算法4-9给出在d维超立方体中求解前缀和问题的过程。



a）数据的初始分布　　b）第二步之前前缀和的分布

图4-13　计算8节点超立方体上的前缀和。在每个节点上，方括号表示累积在结果缓冲区中的本地前缀和，圆括号内是下一步将发送的消息缓冲区中的内容



c）第 i 步之前前缀和的分布　　d）前缀和的最后分布

图4-13 （续）

算法4-9 d 维超立方体上的前缀和

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

4.4 散发和收集

在散发(scatter)操作中，单个节点发送一个大小为m的唯一消息给每一个其他的节点。这个操作也称为一对多私自通信（one-to-all personalized communication）。一对多私自通信和一对多广播不同，私自通信的源节点从p个独自消息开始，每个消息将发给不同的节点。与一对多广播不同，一对多私自通信并不涉及任何数据复制。一对多私自通信或散发操作的对偶是收集（gather）操作或连接（concatenation）操作，在本操作中，一个节点从其他各个节点那里处收集消息。收集操作也和多对一归约操作不同，收集操作并不涉及数据的组合与归约。图4-14中说明散发和收集操作。

167 ~ 168

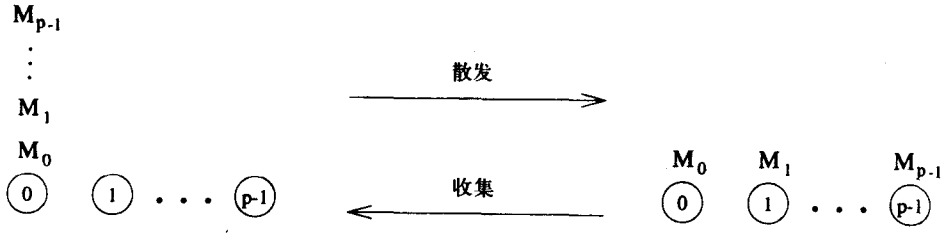
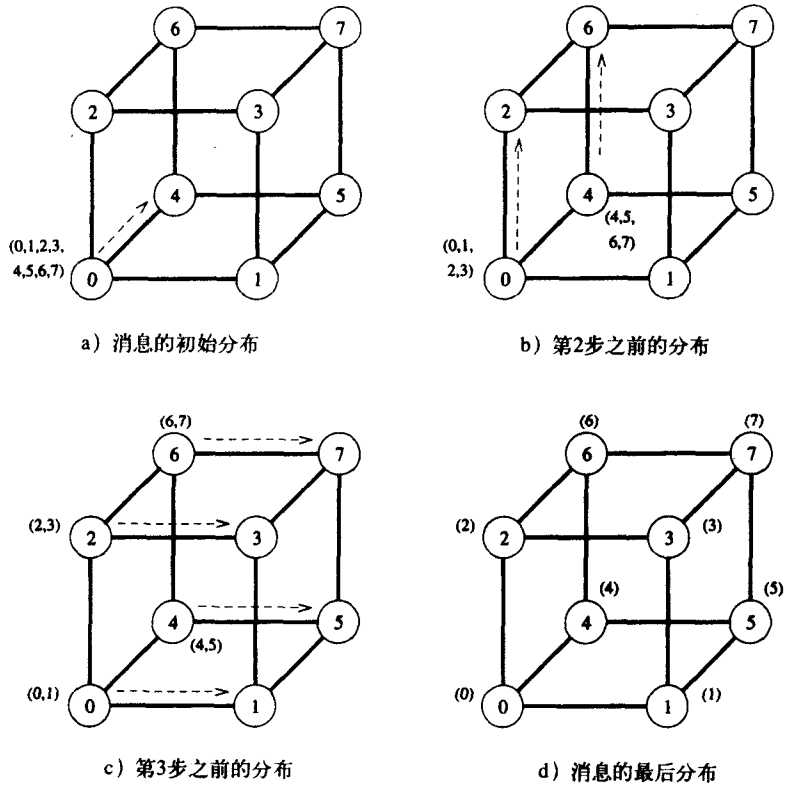


图4-14 散发和收集操作

尽管散发操作在语义上和一对多广播不同，但是散发操作的算法和广播算法十分相似。图4-15给出8节点超立方体上散发操作的通信步骤。一对多广播（图4-6）的通信模式和散发（图4-15）的通信模式相同，只是消息的内容和大小不同。在图4-15中，源节点(节点0) 中包含所有的消息，这些消息由它们的目标节点的标号标识。在通信的第一步，源节点把消息的一半传送给它的一个相邻节点。在接下来的各个步骤中，每个有数据的节点把它的一半传递给还未接收到任何数据的一个相邻节点。这样，在log p维的超立方体中共有log p个通信步骤。



a）消息的初始分布　　b）第2步之前的分布

c）第3步之前的分布　　d）消息的最后分布

图4-15　8节点超立方体中的散发操作

收集操作是散发操作的逆操作。通信开始时，每个节点有m字的消息。在第一步，每个奇数号的节点把它的缓冲区发送给它后面的一个偶数号相邻节点，偶数节点把接收到的消息和自己缓冲区连接。只有偶数号节点参加下一步通信，在其中这4个节点收集更多的数据，并使其数据的大小加倍。该过程将一直继续到节点0收集到整个消息时结束。

像一对多广播和多对一归约一样，超立方体中的散发和收集算法可以不作修改地应用于线性阵列和格网互连拓扑结构中，并且不会增加通信时间。

成本分析　在p节点超立方体中，沿某一维的所有链路连接着两个$p/2$节点的子立方体（2.4.3节）。如图4-15所示，在散发操作的每一个通信步骤，数据从一个子立方体流向另一个子立方体。在开始沿某一维的通信前，需要把个个节点拥有数据的一半发送给另一个立方体中的一个节点。每一步里，正在通信的节点保存一半数据，这一半用于它的子立方体中的节点，而把另一半数据发送给另一个子立方中它的相邻节点。把所有的数据都分配到相应的目

标节点的时间为

$$T = t_s \log p + t_w m(p-1) \tag{4-6}$$

169

散发和收集操作同样可以在线性阵列和2-D方形格网中进行，完成一个操作耗时$t_s \log p + t_w m(p-1)$（习题4-7）。需要注意的是，如果不考虑消息启动时间的项，那么在k-d格网互连网络（2.4.3节）中，大消息的散发和收集操作的成本是类似的。散发操作中，至少有$m(p-1)$字的数据需要从源节点传出，而在收集操作中，至少有$m(p-1)$字的数据需要由目标节点接收。因此，同多对多广播一样，$t_w m(p-1)$是散发和收集操作通信时间的下界。这一下界与互连网络结构无关。

4.5 多对多私自通信

在多对多私自通信（all-to-all personalized communication）中，每个节点发送一个大小为m的不同消息给其他每个节点。每个节点都发送不同的消息给不同的节点，而在多对多广播中，每个节点发送相同的消息给所有其他的节点。图4-16显示多对多私自通信操作。仔细观察该图将会发现，这一操作等价于使用一维数组划分(图3-24)，把在p个进程之间分布的数据的二维数组转置。多对多私自通信也称为总体交换（total exchange）。这一操作常用于一些并行算法中，例如快速傅里叶变换、矩阵转置、样本排序以及某些并行数据库连接操作。

170

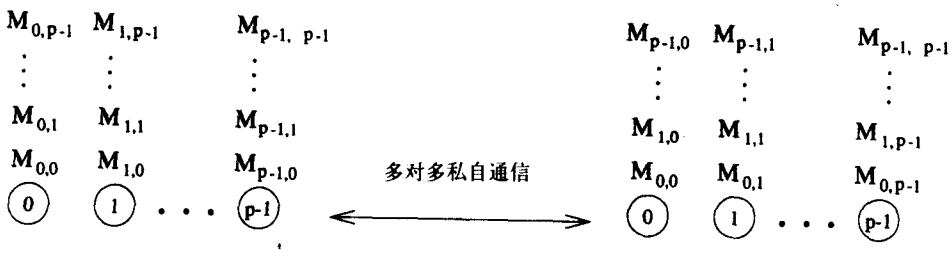


图4-16 多对多私自通信

例4.2 矩阵转置

$n \times n$矩阵A的转置是一个同样大小的矩阵A^T，其中$A^T[i, j] = A[j, i]$，$0 \leq i, j < n$。考虑映射到n个处理器中的一个$n \times n$的矩阵，每个处理器包含矩阵的一行。用这样的映射，处理器P_i最初包含的矩阵元素为$[i, 0], [i, 1], \ldots, [i, n-1]$。转置后，元素$[i, 0]$属于处理器$P_0$，元素$[i, 1]$属于处理器$P_1$，依此类推。一般说来，元素$[i, j]$最初驻留在处理器$P_i$中，但在转置以后被移到处理器$P_j$。图4-17显示这个过程的数据通信模式，使用一维按行划分把一个4×4矩阵映射到4个进程。需要注意的是，图中每个处理器发送矩阵的不同元素给每个其他的处理器。这是一个多对多私自通信的例子。

一般说来，如果使用p个进程，$p \leq n$，那么每个进程最初拥有矩阵的n/p行（也就是n^2/p个元素）。进行转置涉及一个矩阵块大小为$n/p \times n/p$而不是单个元素的多对多私自通信。 ■

现在，我们分别在采用线性阵列、格网以及超立方体互连网络的并行计算机中，讨论多对多私自通信的实现。在这三种体系结构中，多对多私自通信的通信模式与多对多广播的通信模式相同，只是消息的大小和内容不同而已。

171

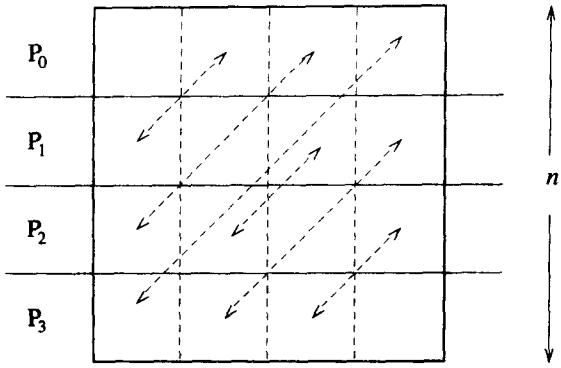


图4-17 用4个进程的4 × 4矩阵转置中的多对多私自通信

4.5.1 环

图4-18显示一个6节点线性阵列中的多对多私自通信的步骤。为了完成这一操作，每个节点发送$p-1$个大小为m字的数据。图中这些消息用整数对$\{i, j\}$标识，其中i为消息源，而j为消息的最终目标。首先，每个节点把它们所有的数据作为一个合并的大小为$m(p-1)$的消息，发送给它们的一个相邻节点(所有节点在同一方向上通信)。在这一步一个节点接收的$m(p-1)$个字的数据中，有一个m字的包属于该节点。因此，每个节点把属于它的信息从接受的数据中取出来，并将余下的$(p-2)$个大小为m的信息转发给下一个节点，这一过程将继续进行$p-1$步。在相继的每一步中，节点之间传送的数据大小将减小m字。每个节点在每一步把来自一个不同节点的一个m字的包加到它的集合中。因此，在$p-1$步中，每个节点从其他所有节点处接收到信息。

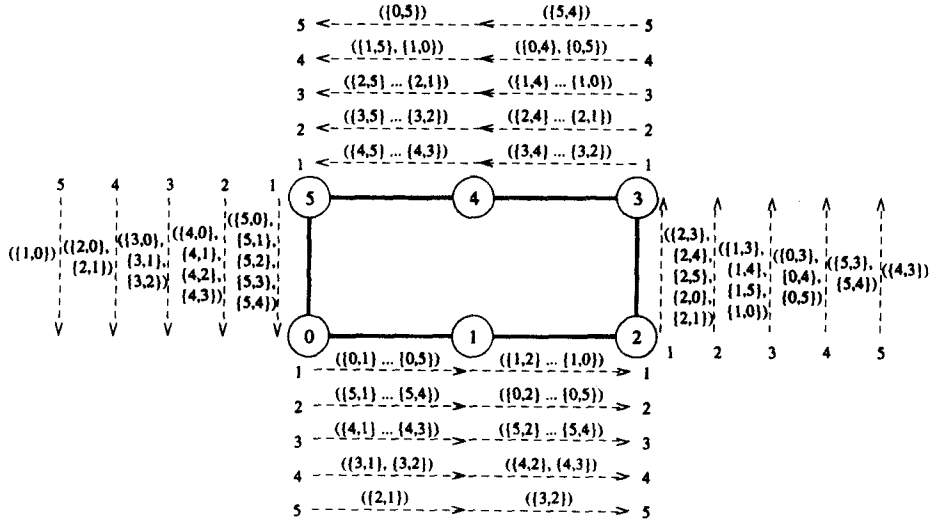


图4-18 6节点环中的多对多私自通信

注：每个消息的标号是$\{x, y\}$的形式，其中x是最初拥有消息的节点的标号，而y是消息最终目标节点的标号。标号$\{\{x_1, y_1\}, \{x_2, y_2\}, \ldots, \{x_n, y_n\}\}$表示$n$个消息连接而成的消息。

在上面的过程里，所有的消息沿相同的方向发送。如果让一半消息沿一个方向发送，而

让另外一半消息沿另外一个方向发送，那么，由t_w导致的通信成本会以2为因子下降。为简单起见，我们忽略该常数因子的改进。

成本分析 在环或者双向线性阵列中，多对多私自通信包含$p-1$个通信步骤。由于在第i步中，传送消息的大小为$m(p-i)$，这个操作耗费的总时间为

$$\begin{aligned} T &= \sum_{i=1}^{p-1}(t_s + t_w m(p-i)) \\ &= t_s(p-1) + \sum_{i=1}^{p-1} i t_w m \\ &= (t_s + t_w mp/2)(p-1) \end{aligned} \tag{4-7}$$

在上面描述的多对多私自通信过程中，每个节点发送$m(p-1)$字的数据，因为它要给其他每个节点发送一个m字的包。假设所有的消息沿顺时针方向或者逆时针方向进行发送。这样一个m字的包的平均传送距离为$(\sum_{i=1}^{p-1} i)/(p-1)$，即$p/2$。因为一共有$p$个节点，每个节点执行同样的通信操作，网络中的总通信量（两个直接相连的节点间传送的总数据字数）为$m(p-1) \times p/2 \times p$。网络中分担这一负载的内部节点链路总数为$p$。因此，这一操作的通信时间至少为$(t_w \times m(p-1)p^2/2)/p$，也就是$t_w m(p-1)p/2$。如果不考虑消息的启动时间$t_s$，这一时间就是线性阵列过程耗费的准确时间。因此，这一节介绍的多对多私自通信算法是最优的。 172 ~ 173

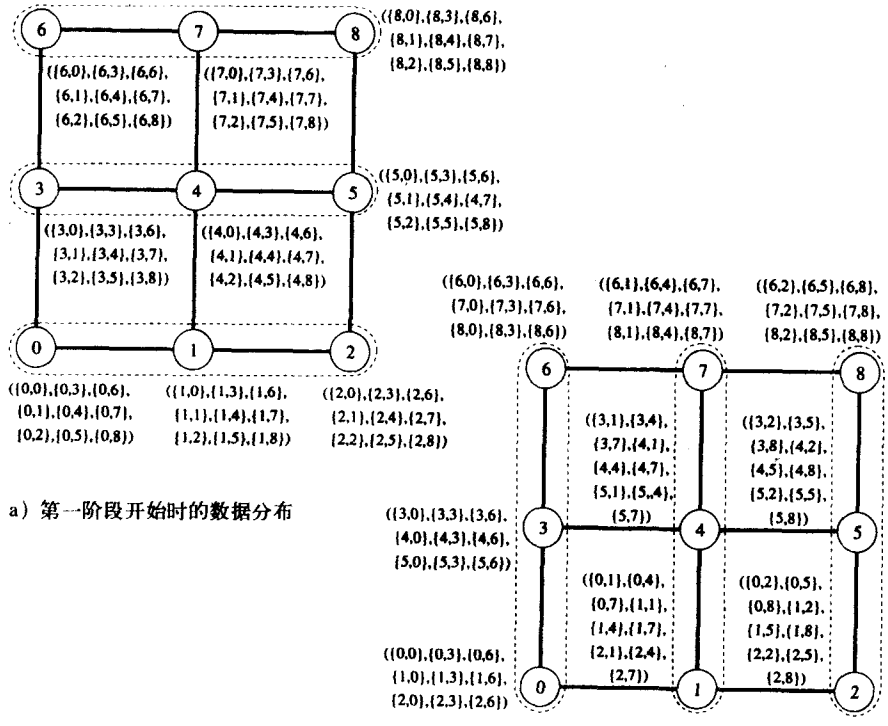


图4-19 在3 × 3格网上的多对多私自通信每个阶段开始时的消息分布。当第二阶段结束时节点i有消息$\{\{0,i\},\ldots,\{8,i\}\}$，其中$0 \leq i \leq 8$。每一阶段中一起通信的节点组包含在虚线框中

4.5.2 格网

在$\sqrt{p}\times\sqrt{p}$的格网上的多对多私自通信中，每个节点首先根据它的目标节点的列把p个消息分组。图4-19显示一个3×3格网，其中每个节点最初含有9个m字的消息，每个对应一个节点。每个节点把它的数据分成3组，每组3个消息（一般说来分为$\sqrt{p}$组，每组$\sqrt{p}$个消息）。第一组包含目标节点为0、3、6的消息；第二组包含的目标节点为1、4、7的消息；第三组包含目标节点为2、5、8的消息。

消息分组后，多对多私自通信开始在每行中独立执行，其中传递大小为$m\sqrt{p}$字的群集。
174 每个群集包含要发给特定列的所有$\sqrt{p}$个节点的信息。图4-19b显示在这一通信阶段结束时节点之间的数据分布。

在第二通信阶段以前，每个节点里的消息被再次排序，此次排序根据它们的目标节点所在列进行；于是通信类似于第一阶段在格网的所有列上进行的通信。在这个通信阶段结束时，每个节点都从其他每个节点接收一个消息。

成本分析 在等式(4-7)中，用$\sqrt{p}$代替节点数，$m\sqrt{p}$代替消息的大小，可以算出第一阶段耗费的时间是$(t_s+t_w mp/2)(\sqrt{p}-1)$。通信的第二阶段的时间开销与第一阶段相同。所以，在p节点二维方形格网上消息大小为m的多对多私自通信的总时间为

$$T=(2t_s+t_w mp)(\sqrt{p}-1) \tag{4-8}$$

等式(4-8)中多对多私自通信时间开销的表达式不包括数据重新排列时间(即按行或按列对消息进行排序的时间)。假设最初数据是为通信第一阶段准备的，通信第二阶段需要对mp字的数据进行重新排列。如果设t_r是在节点的本地存储器上进行一次单字读写操作的时间，那么整个通信过程中用于进行数据重新排列的总时间为$t_r mp$(习题4.21)，这一时间比每个节点花在通信中的时间要小得多。

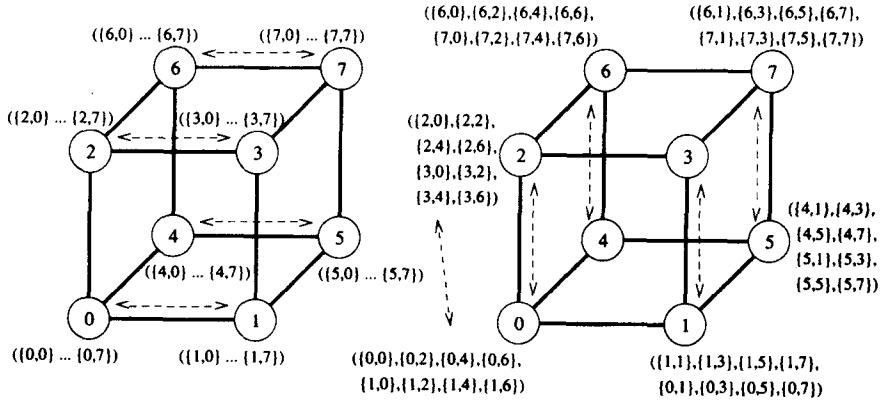
从对线性阵列的分析可知，在一个比较小的常数因数范围里，表示方形格网中多对多私自通信的时间公式（4-8）是最优的。

4.5.3 超立方体

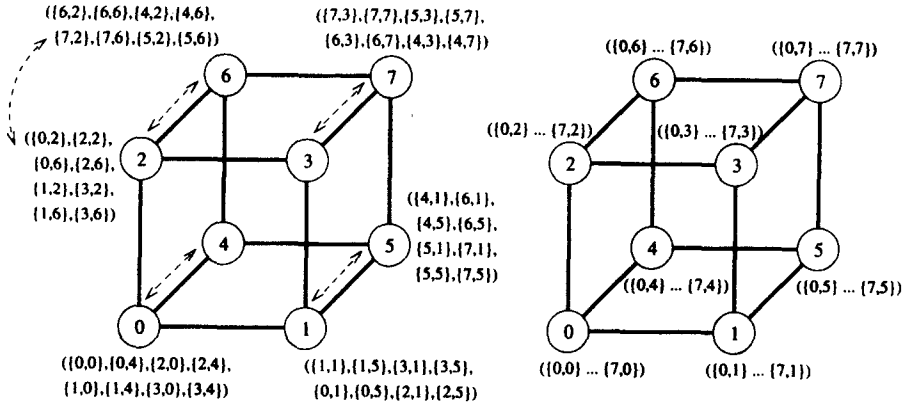
在p节点超立方体上，进行多对多私自通信的方法之一是把它在二维格网中的算法扩展到log p维。图4-20显示在三维超立方体上进行多对多私自通信所需的通信步骤。如图所示，整个通信过程分log p步进行。在每一步，处于不同维的节点对将交换数据。前面讲过，在p节点超立方体中，在同一维中有一个$p/2$条链路的链路集，把两个具有$p/2$个节点的子立方体连接起来(2.4.3节)。在多对多私自通信的任意阶段中，每个节点上保存p个大小为m的数据包。在每一个特定的维上进行通信时，每个节点发送它这些包中的$p/2$个包（合并为一个消息里）。这
175 些包的目标是当前维中由链路连接起来的其他子立方体的节点。

在上述过程中，一个节点在每个log p步通信步骤前，必须在本地重新安排它的消息。为了确保在一个通信步骤里，发往相同节点的所有$p/2$个消息占有相邻的内存地址，使得它们可以作为一个合并的消息传送，这样做是必须的。

成本分析 在上述多对多私自通信的超立方体算法中，log p次迭代的每一次都有$mp/2$字的数据在双向通道中交换。导致的总通信时间为



a）消息的初始分布　　b）第2步前的消息分布



c）第3步前的消息分布　　d）消息的最后分布

图4-20　三维超立方体上的多对多私自通信算法

$$T = (t_s + t_w mp/2)\log p \qquad (4\text{-}9)$$

176

在log p个通信步骤的每一步之前，一个节点要重新排列mp个字的数据。因此，在整个过程中，每个节点在本地重新排序数据耗费的时间为$t_r\, mp\, \log p$。这里，t_r为在节点的本地存储器上执行一次读写一个字操作所需的时间。对于大部分计算机来说，t_r比t_w小得多；因此，通信时间开销是多对多私自通信的主要时间开销。

有趣的是，与在本节描述的线性阵列和格网算法不一样，超立方体算法并不是最优的。p个节点每个都发送和接收$m(p-1)$字的数据，并且超立方体中的任意两个节点间的平均距离为(log p)/2。因此，网络中传送的总数据量为$p \times m(p-1) \times (\log p)/2$。由于超立方体网络中共有($p$ log p)/2条通信链路，多对多私自通信时间的下界为

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

优化算法实例

多对多私自通信将有效地完成每对节点对某些数据的交换。在超立方体上，进行这种数据交换的最好办法是让每一对节点直接通信。这样，每个节点只执行$p-1$个通信步骤，在每一步每个不同的节点交换m字的数据。每个节点在每个通信步骤中选择通信对象，所以超立方体链路不会出现拥塞。图4-21显示在三维超立方体中逐对交换数据的无拥塞调度。如图所示，在第j步通信步骤中，节点i和节点(i XOR j)交换数据。例如，在图4-21a（步骤1）中，参加通信的节点对的二进制标号最低有效位不同。在图4-21g（步骤7）中，参加通信的节点对的二进制标号的所有位都不同，因为7的二进制表示为111。在这个图中，每一步通信的所有路径都是无拥塞的，并且在所有双向链路的同一方向上不会传送一个以上的消息。这一点对任何维的超立方体来说都成立。如果对消息适当地选择路由，则在p节点超立方体上存在一个具有$p-1$步通信的无拥塞多对多私自通信调度。在2.4.3节中讲过，在超立方体上从节点i到节点j，传送一个消息至少要经过l条链路，其中l为节点i到节点j之间的Hamming距离（也就是(i XOR j)的二进制表示中非0位的个数）。一个消息从节点i传送到节点j要通过l个维（对应于(i XOR j)的二进制表示中非0位的个数）中的链路。尽管消息可以经i和j中多条长度为l（假设$l > 1$）的路径中的任意一条进行传送，但是消息传送的路径通过对维数按升序排序的方式选择。按照这一策略，第一条链路选择在(i XOR j)的最低非0有效位所对应的维中，以此类推。这种路由选择方案称为E立方体路由选择（E-cube routing）。算法4-10的d维超立方体上的多对多私自通信就基于这一策略。

177

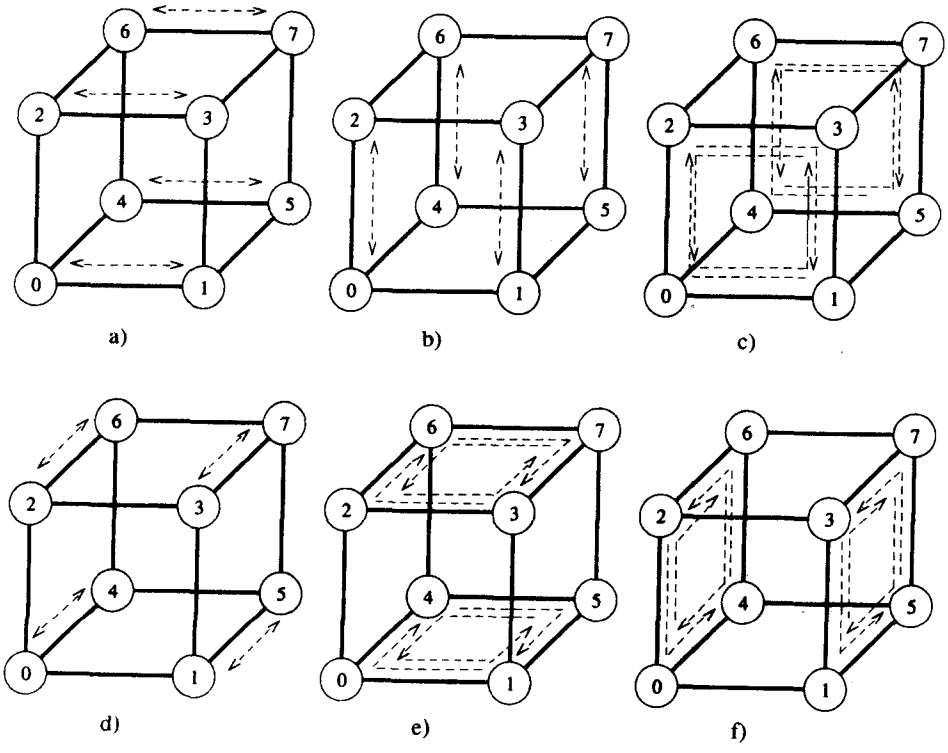
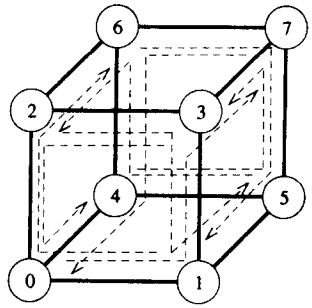


图4-21 8节点超立方体上的多对多私自通信的7个步骤



0 ---> 1 ---> 3 ---> 7
1 ---> 0 ---> 2 ---> 6
2 ---> 3 ---> 1 ---> 5
3 ---> 2 ---> 0 ---> 4
4 ---> 5 ---> 7 ---> 3
5 ---> 4 ---> 6 ---> 2
6 ---> 7 ---> 5 ---> 1
7 ---> 6 ---> 4 ---> 0

g)

图4-21 （续）

178

算法4-10 d维超立方体上进行多对多私自通信的过程。
消息$M_{i,j}$最初驻留在节点i中，通信目标为节点j

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

成本分析 E立方体路由选择确保，按照算法4-10选择参加通信的节点对，在通信时间$t_s + t_w m$内可以将消息从节点i传送到节点j，因为在节点i和节点j之间的通信链路的同一方向上不会出现其他消息的争用。总的通信时间为

$$T=(t_s + t_w m)(p - 1) \tag{4-10}$$

把等式(4-9)和(4-10)进行比较会发现，第二个超立方体算法中和t_s有关的项比第一个算法中的大，而与t_w有关的项比第一个算法中的小。因此，对于小消息来说，启动时间可能占支配地位，所以第一种算法可能还是有用的。

4.6 循环移位

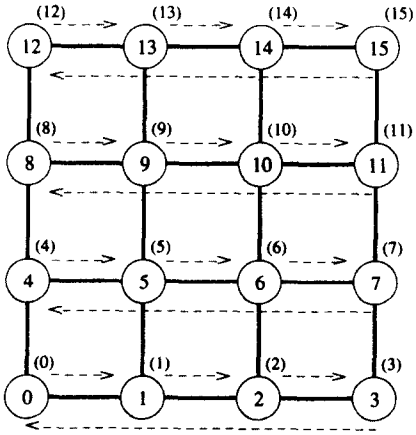
循环移位是称为**置换**（permutation）的更大一类全局通信操作的一种。置换是同时进行的一对一数据重新分布的操作，其中每个节点发送一个m字的包给一个唯一的节点。*循环q移位*（circular q-shift）定义为这样一种操作：在一个p节点的总体中，节点i发送一个数据包给节点$(i + q) \bmod p$，其中$(0 < q < p)$。移位操作在某些矩阵计算及字符串和图像模式匹配上都有应用。

4.6.1 格网

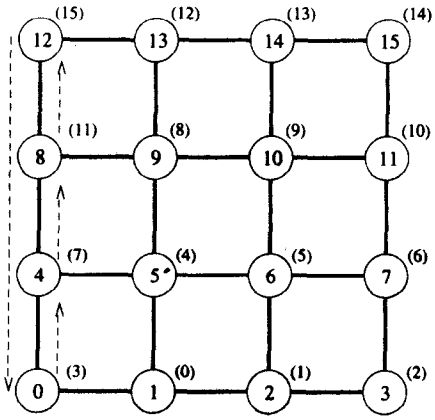
循环q移位操作在环以及双向线性阵列中的实现是很直观的。它可以通过$\min\{q, p-q\}$次相

179 邻节点间沿一个方向的通信来实现。循环移位的格网算法可以由环算法导出。

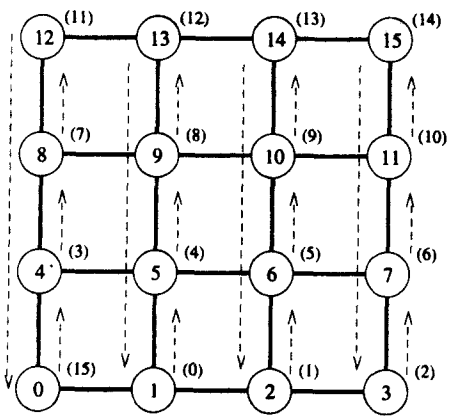
如果格网中的节点按行进行编号，那么在p节点的方形环绕格网中，循环q移位操作可以分两个阶段完成。图4-22中显示一个4 × 4格网中进行循环5移位的过程。首先，整个数据集沿行方向同时移位 $(q \bmod \sqrt{p})$ 步。然后沿列方向移位 $\lfloor q/\sqrt{p} \rfloor$ 步。在循环行移位中，某些数据通过从最高标号节点到最低标号节点的环绕连接。所有这样的数据包必须向列方向进行一次附加的移位步骤，以补偿它们在各自行的后向边上移动时少走的$\sqrt{p}$距离。例如，图4-22中的5移位操作需要一次行移位，还需一次补偿列移位，以及最后一次列移位。



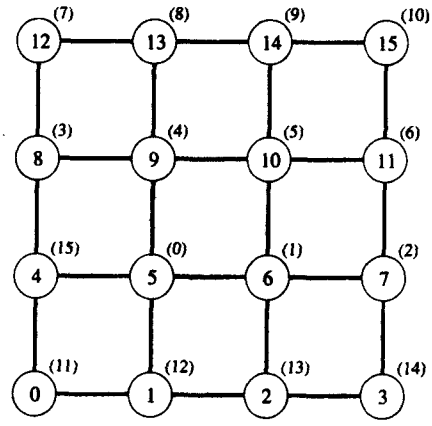
a）初始数据分布和第一个通信步骤



b）补偿反向行移位的步骤



c）第三个通信步骤中的列移位



d）最后的数据分布

180 图4-22 4 × 4格网上的一个循环5移位的通信操作步骤

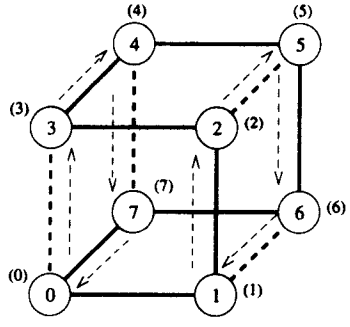
实际上，我们可以在行上和列上选择移位的方向，以使循环移位操作的步骤最少。例如，一个4 × 4格网上的3移位操作可以由一次反向的行移位完成。用这一策略，沿一个方向的单元移位数不会超过 $\lfloor \sqrt{p}/2 \rfloor$。

成本分析 如果考虑一些包的补偿列移位操作，那么p节点格网上数据包大小为m的任何循环q移位的总时间的上界为

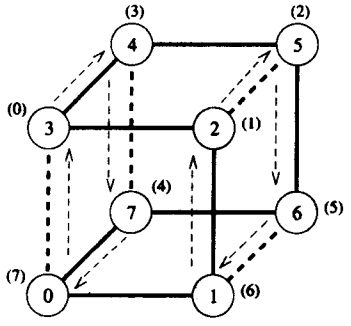
$$T = (t_s + t_w m)(\sqrt{p} + 1)$$

4.6.2 超立方体

在设计移位操作的超立方体算法时，我们把具有2^d个节点的线性阵列映射到d维超立方体中。这样做时，把线性阵列中的节点i映射到超立方体中的节点j，使得j是i的d位二进制反射葛莱码（Reflected Gray Code, RGC）。图4-23中显示8个节点的这种映射过程。这种映射的一个性质是线性阵列中距离为2^i的任何两个节点都恰好由超立方体中的两条链路分开。一个例外是$i = 0$（即在线性阵列中直接相联的节点），这时两个节点只由超立方体中的一条链路分开。

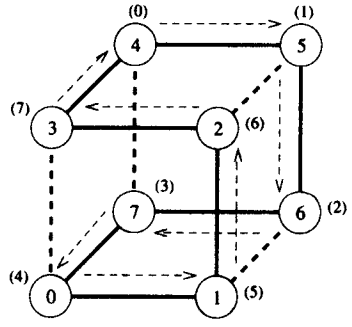


4移位的第一个通信步骤

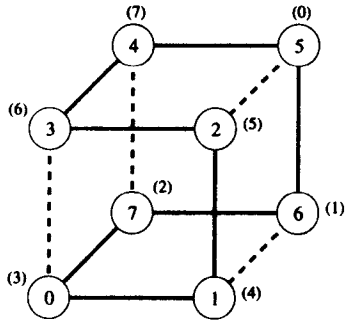


4移位的第二个通信步骤

a）第一阶段（4移位）



b）第二阶段（1移位）



c）5移位后的最后数据分布

图4-23 一个8节点线性阵列到三维超立方体的映射，执行一次由4移位和1移位组成的循环5移位操作

为了进行一次q移位，我们把q扩充为不同的2的幂之和，和中的项数与q的二进制表示中的1的个数相同。例如，数5可以表示为$2^2 + 2^0$。这两项对应于5的二进制表示（即101）中的第0位和第2位。如果q是s个不同的2的幂之和，那么超立方体上的循环q移位在s步内完成。

在每个通信阶段，通过以2的幂作为间隔简化线性阵列（即映射到超立方体），所有的数据包更移近各自的目标。例如，如图4-23所示，一个5移位操作要首先进行一次4移位操作，再

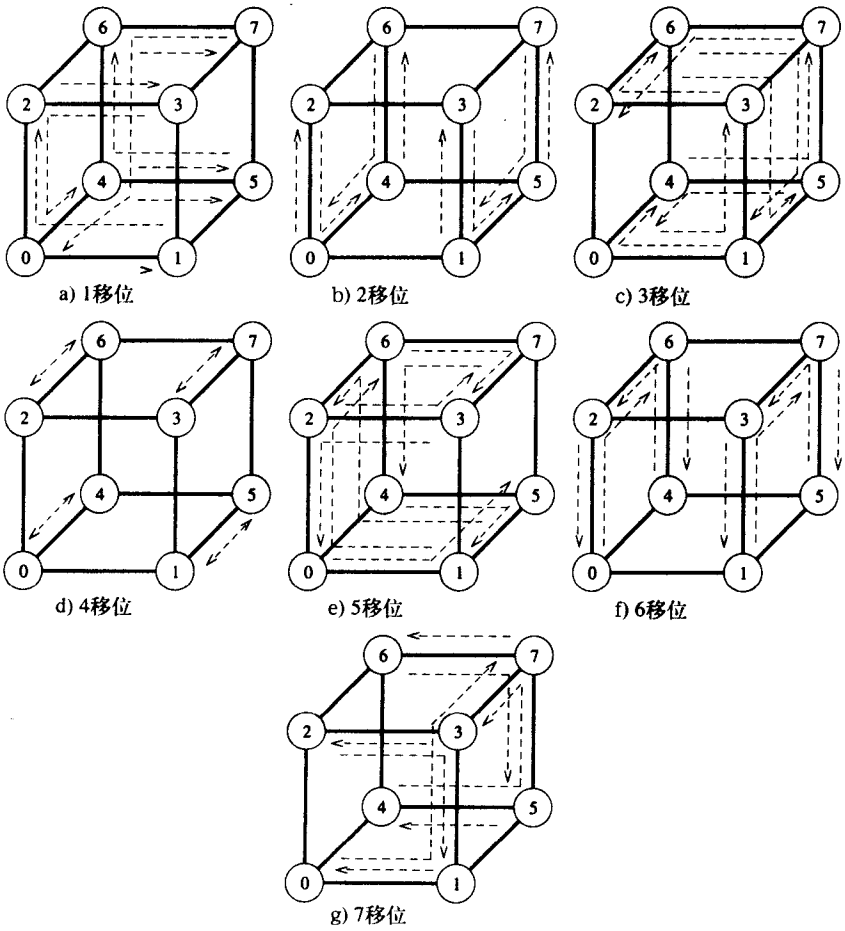
进行一次1移位操作。q移位中通信阶段的数目等于q的二进制表示中1的个数。除1移位操作外，每个阶段都包含两个通信步骤，如果需要1移位操作（即是如果q的最低有效位为1），只有一
181 个操作步骤。例如，在一个5移位操作中，第一阶段的4移位操作（图4-23a）包含两个步骤，而第二阶段的1移位操作（图4-23b）包含一个步骤。因此，在p节点的超立方体中，任何q移位总的步骤至多为$2\log p-1$。

在一个给定的时间步，所有的通信都是无拥塞的。这一点是由线性阵列到超立方体的映射性质保证的，因为线性阵列中所有距离为2的幂的节点被映射到超立方体后重新排列在超立方体不相联的子阵列上。这样，所有节点可以以循环的形式分别在它们的子阵列中自由通信。如图4-23a所示，图中标号为0、3、4、7的节点形成一个子阵列，而标号为1、2、5、6的节点
182 形成另一个子阵列。

对于p节点超立方体上任何m字的消息包的移位操作，总的通信时间的上界为

$$T=(t_s+t_wm)(2\log p-1) \tag{4-11}$$

通过同时进行向前和向后的移位，可以把这个上界减小到$(t_s+t_wm)\log p$。例如，在8节点的超立方体中，一个6移位操作可以通过一次向后的2移位操作代替一次向前的4移位操作再加上一次向前的2移位操作来实现。



183 图4-24 8节点超立方体上的循环q移位操作，其中$1<q<8$

现在要说明的是，如果使用4.5节介绍的E立方体路由选择，则超立方体上长消息的循环移位操作的时间几乎可以提高一个log p的因子。因为使用E立方体路由选择后，在具有双向通路的超立方体中，每一对距离为常数$l(i \leqslant l < p)$的节点间有一条无拥塞的通路(习题4.22)。图4-24显示8节点超立方体上所有消息的循环q移位操作的无冲突通路，其中$1 \leqslant q < 8$。在p节点超立方体上的循环q移位中，最长的路径包含$\log p - \gamma(q)$条链路，其中$\gamma(q)$是使得q可以被2^j整除时最大整数j（习题4.23）。因此，长度为m的消息总的通信时间为

$$T = t_s + t_w m \tag{4-12}$$

4.7 提高某些通信操作的速度

本章至此为止，我们假设通信的源消息不能被划分成更小的部分，同时每个节点只有一个端口可用来发送和接收数据，由此导出了各种通信操作的过程以及它们的通信时间。本节将简单地讨论当放宽假设中的条件时给通信操作带来的影响。

4.7.1 消息分裂和路由选择

在4.1～4.6节描述的过程中，我们假定整个m字的数据包在源节点与目标节点间的同一条路径上传送。如果我们把这些大的消息划分成比较小的消息，然后通过不同的路径传送这些小的消息，就能更充分地利用通信网络。除了一些特例以外，如一对多广播、多对一归约以及全归约等，这一章讨论的通信操作对大消息来说是渐近最优的；也就是说，这些操作的成本中与t_w有关的那些项不能再渐近减少了。本节将提出三个全局通信操作的渐近最优算法。

注意本节中的算法依赖于m大到足以被分成p个大致相等的部分。因此，以前的算法对于短消息来说仍然有用。把本节讨论的算法与本章前面讨论的相同操作的算法进行比较就会发现，当消息被划分以后，与t_s有关的项增加，而与t_w有关的项减少。因此，依赖于t_s、t_w和p的实际值，消息的大小m存在一个截止值，只有当消息长度大于截止值时才可以从本节的算法中获益。 184

1. 一对多广播

下面考虑在p节点的网络中，从源节点向所有节点广播一个大小为m的消息M。如果m大到可以把消息M划分成p部分M_0，M_1，…，M_{p-1}，每部分的大小为m/p，则每个散发操作(4.4节)就可以在时间$t_s \log p + t_w(m/p)(p-1)$内把消息$M_i$放到节点$i$中。注意一对多广播要求的结果就是把消息$M = M_0 \cup M_1 \cup \cdots \cup M_{p-1}$发送到所有的节点上。这可以通过在散发操作后对驻留在各个节点上的大小为m/p的消息进行一次多对多广播来实现。在超立方体上，这个多对多广播可以在时间$t_s \log p + t_w(m/p)(p-1)$内完成。因此，在超立方体上，一对多广播可以在以下时间内完成：

$$\begin{aligned} T &= 2 \times (t_s \log p + t_w(p-1)\frac{m}{p}) \\ &\approx 2 \times (t_s \log p + t_w m) \end{aligned} \tag{4-13}$$

与公式(4-1)比较，这个算法的启动成本是公式(4-1)中的两倍，但与t_w项有关的成本下降$(\log p)/2$倍。同样，在线性阵列和格网互连网络中，一对多广播也能改进性能。

2. 多对一归约

多对一归约是一对多广播的对偶。因此，通过反转一对多广播的通信方向和顺序，就可

以得到多对一归约的算法。上面我们说明了如何通过先进行一次散发再进行一次多对多广播获得一个优化的一对多广播算法的过程。因此，利用对偶性概念，我们应该也可以先进行一次多对多归约（多对多广播的对偶），再进行一次收集操作（散发的对偶），来实现多对一归约。我们把这样一个算法的细节留给读者作为练习(习题4.17)。

3. 全归约

由于全归约操作在语义上等价于一个多对一归约操作加上一个一对多广播操作，所以，上述两个操作的渐近最优算法可以用于构建全归约的一个类似的算法。把多对一归约操作和一对多广播操作分割成它们的组件操作，可以看出，全归约操作可以通过进行一个多对多归约操作加上一个收集操作，再加上一个散发操作和一个多对多广播操作来完成。由于中间的收集和散发操作将抵消相互的作用，所以全归约操作正好要求一个多对多归约操作和一个多对多广播操作。首先，p节点的每个节点上的m字的消息在逻辑上被划分成p个大小大致为m/p字的部分。接着，再用多对多归约操作把p_i上的所有第i个部分合并。在这个步骤后，每个节点留下一个最后结果的大小为m/p字的部分。用一个多对多广播操作可以构建每个节点上这些
185 部分的连接。

p节点超立方体互连网络允许对大小为m/p字的消息进行多对一归约操作和一对多广播操作，每个操作花费的时间为$t_s \log p + t_w(m/p)(p-1)$。因此，全归约操作可在以下时间内完成:

$$\begin{aligned} T &= 2 \times (t_s \log p + t_w(p-1)\frac{m}{p}) \\ &\approx 2 \times (t_s \log p + t_w m) \end{aligned} \tag{4-14}$$

4.7.2 全端口通信

在并行体系结构中，一个节点可以有多个通信端口和链路与其他节点通信。例如，二维环绕格网中的每个节点有4个端口，d维超立方体的每个节点有d个端口。本书中，我们一般假设使用*单端口通信*（single-port communication）模型。在单端口通信中，一个节点一次只能在它的端口之一发送数据。同样，一个节点一次只能在一个端口上接收数据。但是，一个节点可以在同一端口或者不同端口同时接收和发送数据。与单端口模式不同，*全端口通信*（all-port communication）模型允许一个节点在和它相连的多条通道上同时通信。

在全端口通信的p节点的超立方体中，一对多广播和多对一广播以及私自通信时间表达式中，t_w的系数比它们在单端口通信中的对应部分小$\log p$倍。由于在线性阵列或格网中，每个节点的通道数是一个常数，这两种结构的全端口通信不能给通信时间提供任何渐近提高。

尽管有明显的加速，但是全端口通信模型存在某些局限。例如，全端口通信模型不仅难于编程，而且还要求消息大到可以分裂成许多部分，使其在不同通道之间有效地传送。在一些并行算法中，消息增大意味着节点中计算粒度也变大。当节点处理大数据集时，如果算法的计算复杂度高于通信复杂度，那么节点间的通信时间将主要是计算时间。例如，在矩阵相乘中，对于节点间传送的n^2字的数据有n^3次计算。如果通信时间在整个并行运行时间内只是一小部分，那么，使用非常复杂的技术来改进通信对整个并行算法的运行时间就没有什么意义。

全端口通信的另外一个局限，是只有把它需要的所有数据取出来并存储在内存中，存取速度要快到足以支持并行通信，这时通信才是有效的。例如，为了在一个p节点超立方体上有效地利用全端口通信，内存带宽必须比单通道通信带宽至少大$\log p$倍；也就是说，内存带宽

必须随着节点数的增加而增加，以支持全端口的同时通信。一些现代的并行计算机，如IBM SP，对于这个问题有比较自然的解决方法。分布式内存并行计算机的每一个节点是一个NUMA共享内存的多处理机。如果发送或者接收数据的缓冲区能被正确地置于不同的存储区，那么就通过单独存储区和全存储器提供多端口服务，并且能够利用通信带宽。 186

4.8 小结

表4-1中汇总本章中讨论的各种聚合通信操作的通信时间。其中，一对多广播、多对一归约以及全归约操作的时间是两种表达式中的最小值。这是因为，这些通信操作的时间取决于通信消息的大小m，或许4.1和4.3节中讲述的算法更快，也可能是4.7节中讲述的算法更快。表4-1假定选择的是最适合于给定消息大小的算法。表4-1中的通信时间表达式是从本章前面几节采用直通路由选择的超立方体网络环境中推导的。然而，凡是截面带宽为$\Theta(p)$的体系结构（2.4.4节），这些表达式及相应的算法是合法的。事实上，除了多对多私自通信和循环移位，表4-1中列举的所有通信操作表达式中，与t_w有关的项保持不变，即使在环和格网网络（或任何k-d格网网络）也是如此，只要逻辑进程能正确地映射到网络的物理节点。假设采用从进程到节点的最优映射，在表4-1第二列给出的时间内，表4-1最后一列给出完成一个操作所需的渐近截面带宽。对于大消息来说，只有多对多私自通信和循环移位需要全部$\Theta(p)$截面带宽。因此，如2.5.1节的讨论，当截面带宽较小的网络应用这些操作的时间表达式时，t_w项必须反映有效带宽。例如，p节点方形格网的对分带宽是$\Theta(\sqrt{p})$，而p节点环的对分带宽是$\Theta(1)$。因此，在方形格网中执行多对多私自通信时，每字传送的有效时间将是每条链路的t_w的$\Theta(\sqrt{p})$倍，而在环中，它应该是每条链路的t_w的$\Theta(p)$倍。 187

表4-1 4.1～4.7节中讨论的超立方体互连网络中的各种操作的通信时间汇总。每种操作的消息大小为m，节点数为p

操 作	通信时间	带宽需求
一对多广播，多对一归约	$\min((t_s + t_w m)\log p, 2(t_s \log p + t_w m))$	$\Theta(1)$
多对多广播，多对多归约	$t_s \log p + t_w m(p-1)$	$\Theta(1)$
全归约	$\min((t_s + t_w m)\log p, 2(t_s \log p + t_w m))$	$\Theta(1)$
散发、收集	$t_s \log p + t_w m(p-1)$	$\Theta(1)$
多对多私自	$(t_s + t_w m)(p-1)$	$\Theta(p)$
循环移位	$t_s + t_w m$	$\Theta(p)$

表4-2 本章中讨论的通信操作的MPI名称

操 作	MPI名称
一对多广播	MPI_Bcast
多对一归约	MPI_Reduce
多对多广播	MPI_Allgather
多对多归约	MPI_Reduce_scatter
全归约	MPI_Allreduce
收集	MPI_Gather
散发	MPI_Scatter
多对多私自通信	MPI_Alltoall

在许多并行算法中，能经常看到本章中讨论的聚合通信操作。为了使有效的并行程序更快速和更具可移植性，绝大多数并行计算机厂商都为执行这些聚合通信操作提供预先打包的软件。这些通信操作的最常用的标准API称为消息传递接口(Message Passing Interface)，或MPI。表4-2给出本章中描述的通信操作对应的MPI函数名称。

4.9 书目评注

这一章我们讨论了线性阵列、格网以及超立方体互连拓扑结构中的各种数据通信操作。Saad和Schultz[SS89b]讨论在这些结构以及其他一些结构（如共享内存和开关或者总线互连的结构）中，这些操作的实现问题。大部分并行计算机厂商为进程间用消息传递通信提供标准的API。两个最常见的API是消息传递接口(MPI)[SOHL+96]和并行虚拟机(PVM)[GBD+94]。

在一些其他连接较少的网络结构中，如果这些结构支持直通路由选择，则通信操作的超立方体算法通常也是这些网络中的最好算法。由于超立方体结构的通用性以及超立方体算法的广泛应用，人们已做了大量的工作，用于在超立方体中实现多种通信操作[BOS+91, BR90,
188 BT97, FF86, JH89, Joh90, MdV87, RS90b, SS89a, SW87]。Sadd和Schultz[SS88]描述用来推导各种通信操作算法的超立方体网络的特性。

Boppana和Raghavendra[BR90]、Johnsson和Ho[JH91]，Seidel[Sei89]以及Take[Tak87]特别分析了超立方体结构中的多对多私自通信问题。Nugent[Nug88]描述算法4-10中多对多私自通信时保证无拥塞通信的E立方体路由选择。

Ranka和Sahni[RS90b]描述4.3节中的全归约和前缀和操作算法。本书中讨论的循环移位操作改自Bertsekas和Tsitsiklis[BT97]。一般形式的前缀和，通常称为*扫描*，常被研究人员用在数据并行程序设计中作为基本原语。Blelloch[Ble90]定义*扫描向量模型*（scan vector model），并且描述如何用原始的扫描原语以及它的变形来表示各种并行程序。

一对多广播的超立方体算法用到Bertsekas和Tsitsiklis[BT97]以及Johnsson和Ho[JH89]中描述的生成二项式树。在4.7.1节中描述的生成树算法中，为了简化算法的表现形式，我们把m字的消息划分成大小为$m/\log p$的$\log p$个部分进行广播。Johnsson和Ho[JH89]说明每部分的最优大小为$\left\lceil \sqrt{t_s m / t_w \log p} \right\rceil$。在这种情况下，消息的个数可能会大于$\log p$。这些更小的消息从生成二项式树的根节点以循环的方式发往它的$\log p$个子树。使用这种策略，p节点超立方体上的一对多广播可以在$t_s \log p + t_w m + 2t_w \left\lceil \sqrt{t_s m / t_w \log p} \right\rceil \log p$时间内完成。

Bertsekas和Tsitsiklis[BT97]，Johnsson和Ho[JH89]，Ho和Johnsson[HJ87]，Saad和Schultz[SS89a]以及Stout和Wagar[SW87]中描述在超立方体结构中采用全端口通信模型的各种通信操作算法。Johnsson和Ho[JH89]中说明在采用全端口通信的p节点超立方体上，在一对多广播、多对多广播以及私自通信的通信时间表达式中，t_w的系数比单端口通信中的对应部分小$\log p$倍。Gupta和Kumar[GK91]说明在并行结构中，全端口通信可能不会比单端口通信更能提高算法的可扩展性。

在并行应用程序中，还用到其他一些本章中未讲到的基本通信操作。文献中提到了许多其他用于并行计算机的有用操作，其中包括选择[Ak189]，指针跳转[HS86, Jaj92]，BPC置换[Joh90, RS90b]，取数并操作[GGK+83]，打包[Lev87, Sch80]，位取反[Loa92]，以及键控扫描或多前缀[Ble90, Ran89]。

有时数据通信并不遵循任何预先定义的模式，而是任意的，根据其应用的情况来选择。

此时，过分简单地沿源节点和目标节点间的最短数据路径选择消息的路由，将导致争用和不均衡的通信。Leighton，Maggs和Rao[LMR88]，Valiant[Val82]以及Valiant和Brebner[VB81]讨论任意消息置换中有效的路由选择方法。 189

习题

4.1 修改算法4-1、4-2和4-3，使它们适用于任意数目的进程而不只是2的乘幂的进程。

4.2 4.1节中提出一对多广播的递归加倍算法，它适用于三种网络(环、格网、超立方体)。注意在图4-6中的超立方体算法中，消息首先沿最高维发送，然后再发送给较低的维（在算法4-1的第4行中，i从$d-1$减少到0）。同样的算法可以用于格网和环中，并保证在不同时间步中发送的消息间不会相互干扰。

现在修改算法，使得消息首先沿最低维发送（即在算法3-1的第4行中，i从0增加到$d-1$）。所以在第一步，处理器0将和处理器1通信；在第二步，处理器0和1将分别与处理器2和3通信；依次类推。

1）这个修改后的算法在超立方体上的运行时间是多少？

2）这个修改后的算法在环上的运行时间是多少？

对于这些推导，如果k个消息必须在同一时间内通过同一条链路，那么可以假设这些消息的有效每字传送时间为kt_w。

4.3 在环中，多对多广播有两种不同的实现方法：a）图4-9中所示的标准环算法，b）图4-11中所示的超立方体算法。

1）方法a）的运行时间是多少？

2）方法b）的运行时间是多少？

如果k个消息必须在同一时间通过同一链路，那么假定这些消息的有效每字传送时间为kt_w。同时假定$t_s = 100 \times t_w$。

1）如果消息m非常大，那么上面a) 和b) 两种算法中哪个更好？

2）如果消息m非常小（可能是一个字），那么哪个算法更好？

4.4 根据算法4-6写一个在格网中进行多对多归约的过程。

4.5 （**树中的多对多广播操作**）给定如图4-7所示的平衡二叉树，描述一个实现多对多广播的过程，完成p节点上m字消息的通信耗费时间$(t_s + t_w mp/2)\log p$。假设只有树的叶子包含节点，并且如果通信通道（或部分通道）由k个同时传播的消息共享，那么在通过双向通道连接的任意两个节点间，交换两个m字消息的时间为$t_s + t_w mk$。 190

4.6 考虑全归约操作中，其中每个处理器开始时有一个m字的数组，并且需要得到每个处理器的数组中各自的字的全局和。在环上实现这一操作有下面三种选择：

i）先进行所有数组的多对多广播，再在本地对数组中每个元素求和。

ii）先在单节点上累积数组的元素，然后对结果数组进行一次一对多广播。

iii）使用多对多广播模式的一个算法，但只是简单地进行加法操作而不对消息进行链接。

1）对上述的每一种情况，计算用m、t_s和t_w表示的运行时间。

2）假设$t_s \gg 100$，$t_w = 1$，并且m非常大。上述三种选择（(i)、(ii)或(iii)）中哪一种更好？

3）假设$t_s \gg 100$，$t_w = 1$，并且m非常小（例如m为1）。上述三种选择（i)、ii)或iii)）中哪一种更好？

4.7 （**线性阵列和格网上的一对多私自通信**）在线性阵列和格网结构中，给出p个节点上m字消息的一对多私自通信的过程以及它们的通信时间。

提示：对于格网来说，算法像通常那样分两阶段进行，开始时，源节点在它行上的$\sqrt{p}$个节点间分布大小为$m\sqrt{p}$字的消息，使得每个节点接收的数据对应源节点列上的所有$\sqrt{p}$个节点。

4.8 （**多对多归约**）多对多广播的对偶是多对多归约。在多对多归约中，每个节点是一个多对一归约的目标节点。例如，考虑p个节点的情况，其中每个节点有一个p个元素向量，第i个节点（对所有的i，$0 \leqslant i < p$）将得到所有向量中第i个元素的和。描述一个在超立方体上进行多对多归约操作的算法，其中以加法作为结合操作符。如果每个消息包含m个字，并且执行一次加法操作的时间为t_{add}，那么你的算法所需的时间是多少（用m，p，t_{add}，t_s和t_w表示）？

提示：在多对多广播中，每个节点都从一个消息开始，操作结束时将收集到p个这样的消息。在多对多归约中，每个节点从p个不同的消息开始（每个对应不同的节点），但操作结束时每个节点上只有一个消息。

4.9 图4-21的c）、e）和f）部分表明，对于三维超立方体中的任何一个节点，恰好有3个节点离此节点的最短距离为两条链路。请推导在p节点超立方体中，离任意一个给定节点的最短距离为l的节点数目的表达式（用p和l表示）。

191 4.10 请给出计算n个数的前缀和的超立方体算法，其中p是节点数，而n/p是大于1的整数。假设两个数相加的时间为t_{add}，而在两个直接连接的节点间发送一个单位长度消息的时间为t_s，请给出算法花费的总时间表达式。

4.11 如果消息的启动时间t_s为零，请证明$\sqrt{p} \times \sqrt{p}$格网上的多对多私自通信花费时间的表达式$t_w mp(\sqrt{p}-1)$，在一个较小($\leqslant 4$)的常数因子内是最优的。

4.12 修改4.1～4.5节中讨论的线性阵列和格网算法，使它们能用于无端对端环绕连接的网络中。对比修改后的算法和原算法的通信时间。在线性阵列或者格网中，对于任何操作，时间增加的最大因子是多少？

4.13 （**3-D格网**）在p个节点的用存储转发路由选择的$p^{1/3} \times p^{1/3} \times p^{1/3}$三维格网中，给出一对多广播、多对多广播以及私自通信的最优算法(在一个小的常数内)，并推导这些过程总的通信时间表达式。

4.14 假设建立一个p个节点的并行计算机的成本和它内部的通信链路总数成正比。令一种体系结构的成本效率反比于这种结构的p节点成本和其中某一操作的通信时间的乘积。假设t_s为零，那么，对本章中讨论的每种操作，标准3-D格网或稀疏3-D格网中哪种结构的成本效率更高？

4.15 当t_s为非零常数而$t_w = 0$时，重做习题4.14。在这种通信模型下，两个直接相连节点间的消息传送时间为固定值，与消息大小无关。此外，如果把两个包组合起来作为一个消息发送，则通信延迟依然是t_s。

4.16 （**k对多广播**）令k对多广播是k个节点同时进行消息大小为m字的一对多广播的操作。请给出在p节点超立方体上花费时间为$t_s \log p + t_w m(k \log(p/k) + k - 1)$的$k$对多广播的一个算法。假设$m$字的消息不能被再划分，$k$是2的幂，并且$1 \leqslant k \leqslant p$。

4.17 在p节点超立方体上进行多对一归约操作时，可通过把大小为m的原始消息划分成p个大小为m/p的几乎相等的部分，使得操作的时间为$2(t_s \log p + t_w m(p-1)/p)$，请给出这个操作

的一个算法的详细描述。

4.18 如果消息可以被划分，且划分后的部分可以独立地进行路由选择，试推导一个k对多广播算法，使得其通信时间小于习题4.16中p节点超立方体算法的通信时间。

4.19 试证明，如果$m \geqslant p$，那么在p节点的超立方体中，消息大小为m的多对一归约操作可以在时间$2(t_s \log p + t_w m)$内完成。

提示：将多对一归约表述成多对多归约和收集的结合。 192

4.20 **（k对多私自通信）**在k对多私自通信中，k个节点同时执行p节点结构中每个包大小为m的一对多私自通信($1 \leqslant k \leqslant p$)。证明：如果$k$是2的幂，那么这个操作可以在超立方体上执行，执行时间为$t_s(\log(p/k) + k - 1) + t_w m(p-1)$。

4.21 假定在节点的本地内存中，对单字数据执行一次读写操作的时间是t_r，证明在p节点格网（4.5.2节）上的多对多私自通信中，花费在节点上内部数据转移的总时间是$t_r mp$，其中m是单个消息的大小。

提示：内部数据转移等价于转置大小为m的消息的$\sqrt{p} \times \sqrt{p}$数组。

4.22 证明在p节点的超立方体上，如果使用E立方体路由选择（4.5节），循环q移位的所有p条数据路径是无拥塞的。

提示：1）如果$q > p/2$，那么p节点超立方体上的q移位和$(p-q)$移位同构。2）在超立方体上，用归纳法证明：如果p节点超立方体上的q移位($1 \leqslant q < p$)的所有路径是无拥塞的，那么对于$2p$个节点的超立方体，所有这些路径也是无拥塞的。

4.23 证明：在p个节点的超立方体上，循环q移位的任何消息，其最长路径长度为$\log p - \gamma(q)$，其中$\gamma(q)$是q可以被2^j整除的最大整数j。

提示：1）在p个节点的超立方体上，如果$q = p/2$，那么$\gamma(q) = \log p - 1$。2）在超立方体上用归纳法证明，对于给定的q，$\gamma(q)$随着每次节点数目加倍而增加1。

4.24 推导以下超立方体算法的并行运行时间表达式：一对多广播，多对多广播，一对多私自通信，以及多对多私自通信。这些算法是不改变同样通信链路（同样的通道宽度和通道速度）的格网时改写而来的。将这些改写后的算法性能与最好的格网算法性能相比较。

4.25 如2.4.4节所述，网络成本有两种通用的度量：1）并行计算机中的线路数目（它是通信链路和通道带宽的乘积）；2）对分带宽。考虑每个链路的通道带宽为1的超立方体，即$t_w = 1$。如果节点数目和成本相同，那么格网连接计算机的通道带宽较高，它由所使用的成本度量决定。令s和s'代表两个与成本度量相对应格网的通道带宽增加因子。试求出s和s'的值，并用这些值导出在格网上下列操作的通信时间：

1）一对多广播。

2）多对多广播。

3）一对多私自通信。 193

4）多对多私自通信。

将这些时间与在超立方体上有同样成本的相同操作所用的时间进行比较。

4.26 考虑p个节点的全连接网络。请根据习题4.25中的4个通信操作，推导在完全连接网络上超立方体算法的并行运行时间表达式。讨论增加网络连接对于提高这些操作的性能有什么影响。

194

第5章　并行程序的解析建模

一个串行算法的优劣通常按其执行时间来评估，并表示为其输入数据大小的函数。一个并行算法的执行时间不仅依赖于输入数据大小，还依赖于所用的处理器的数目，及它们相关的计算速度和进程间的通信速度。因此，并行算法不可能在精度方面毫无损失地根据并行体系结构孤立地进行评估。*并行系统*（parallel system）是算法和并行体系结构的组合且在这个并行体系结构上实现算法。本章我们研究评估并行系统性能的各种度量。

很多性能评测方法是直观的，其中最简单的是在给定并行平台上求解给定问题所花费的挂钟时间。但是，正如我们将要看到的，这种单一性能特征不能外推到其他问题实例中或更大的机器结构中。其他直观的方法是量化并行性的效益，即并行程序比串行程序运算快多少。然而，除以上提到的那些外，这种特性还有其他的缺陷。例如使用一个更适合于并行处理的较差的串行算法有什么影响？由于这些原因，人们需要将更复杂的评测方法推广到结构更大的计算机结构或问题中。基于这些目标，本章重点讨论量化并行程序的性能。

5.1　并行程序中的开销来源

如果使用多达两倍的硬件资源，自然指望使一个程序运行快两倍。然而，在典型的并行程序中，由于涉及并行化的各种开销，这是非常少有的情况。对这些开销精确量化是理解并行程序性能的关键。

图5-1说明一个并行程序的典型运行过程。除进行基本的计算（即对求解相同问题实例进行的串行程序计算）外，并行程序还在进程间通信、空闲以及额外计算（在串行形式中不进行的计算）中花费时间。

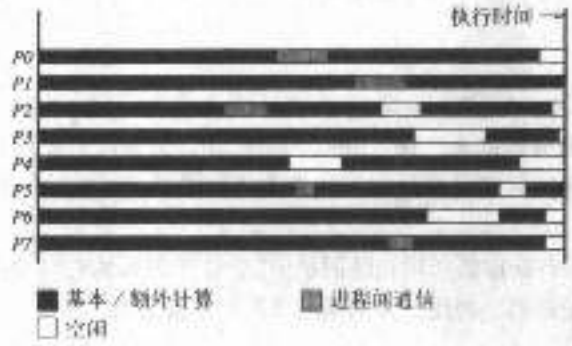


图5-1　在8个处理器上执行的一个假想并行程序的执行图。本图表示执行计算（基本的和额外的）、通信和空闲时花费的时间

进程间的交互　任何非平凡的并行系统都要求其处理器交互和传送数据（如中间结果）。这种花费在处理器间数据通信的时间通常是并行处理开销最重要的来源。

空闲　由于很多原因，如负载不平衡、同步及程序中串行部分的出现，处理器在并行系统

中可能成为空闲。在许多并行应用中（如任务的动态生成）预测分配给各个处理器的子任务大小是不可能的（或至少是困难的）。因此当维持均衡工作负载时，在处理器中，问题不可能静态地再细分。如果不同的处理器有不同的工作负载，在其他单元正在处理问题的部分时间里，一些处理器可能空闲。在一些并行程序中，并行程序运算时处理器必须在某些点同步。如果在同一时间所有处理器没有做好同步准备，那么较早就绪的处理器将会空闲，直到所有其余处理器均就绪。一个算法的有些部分可能是不能并行化的，只容许单个处理器执行。当
196 处理器在串行部分工作时，所有其他处理器必须等待。

额外计算 对某个问题的串行算法而言，已知的最快算法也许不能或很难并行化，这样，迫使我们使用一个较慢但容易并行化（即有较高的并发度）的串行算法来进行并行化。并行程序和最好的串行程序在计算中的差异是由于并行程序引起的额外计算。

基于最快串行算法的并行算法也许进行比串行算法更多的聚合计算。这种计算的一个实例是快速傅里叶变换算法。使用串行算法时，某些计算结果能重复使用。然而，使用并行算法时，这些结果不能重复使用，因为它们是由不同处理器产生的。因此，在不同处理器上，一些计算会多次进行。第13章将详细讨论这些算法。

由于求解相同问题时，用不同并行算法会导致不同的开销，对每个算法建立一个性能特征来量化这些开销是很重要的。

5.2 并行系统的性能度量

从确定最好算法、评估硬件平台及检查并行化的益处的观点研究并行程序的性能是重要的。根据性能分析的预期结果，人们采用了多种度量。

5.2.1 执行时间

程序串行运行时间是指在串行计算机上，程序从开始到运行结束所用的时间。*并行运行时间*（parallel runtime）是从并行计算开始时刻到最后的处理器完成运算所经过的时间。我们用T_s表示串行运行时间，T_p表示并行运行时间。

5.2.2 总并行开销

在称为*开销函数*（overhead function）的单一表达式中，包含由并行程序导致的开销。我们定义并行系统的开销函数或*总开销*（total overhead）为由所有处理器花费的总时间，除去在单个处理器上求解相同问题时最快的串行算法所需的时间。我们用符号T_o表示并行系统的开销函数。

求解一个问题，所有处理器所用总时间是pT_p。这个时间的T_s部分是完成有用工作所花费的时间，剩余部分即是开销。因此，开销函数（T_o）可表示为

197
$$T_o = pT_P - T_S \tag{5-1}$$

5.2.3 加速比

评估并行系统时，我们常常感兴趣的是，通过对一个给定应用并行化，比串行计算能获得多少性能增益。加速比是并行求解问题获得相应益处的一种度量。它被定义为在单个处理

器上求解问题所花的时间与用p个相同处理器并行计算机求解同一问题所花时间之比。我们用符号S表示加速比。

例5.1 用n个处理器对n个数相加

考虑用n个处理器加n个数的问题。最初，对每个处理器指定一个待相加的数，计算结束时，其中一个处理器保存相加的总和。假定n为2的幂，我们可以通过衍生部分总和为处理器的一棵逻辑二叉树，用$\log n$步来实现这个运算。图5-2说明$n = 16$时的计算过程。处理器标号为从0到15，类似地，相加的16个数也加上标号0到15。这些带有连续标号i到j的数之和用 $\sum_i^j$ 表示。

图5-2中显示的每一步由一个加法和单个字的通信组成。加法可在某个固定时间（例如t_c）内实现，而单个字的通信可在时间$t_s + t_w$内实现。因此，加法和通信运算花费一个常数时间。这样，

$$T_P = \Theta(\log n) \tag{5-2}$$

由于这种问题可用单个处理器在时间$\Theta(n)$内求解，其加速比为

$$S = \Theta\left(\frac{n}{\log n}\right) \tag{5-3}$$

■

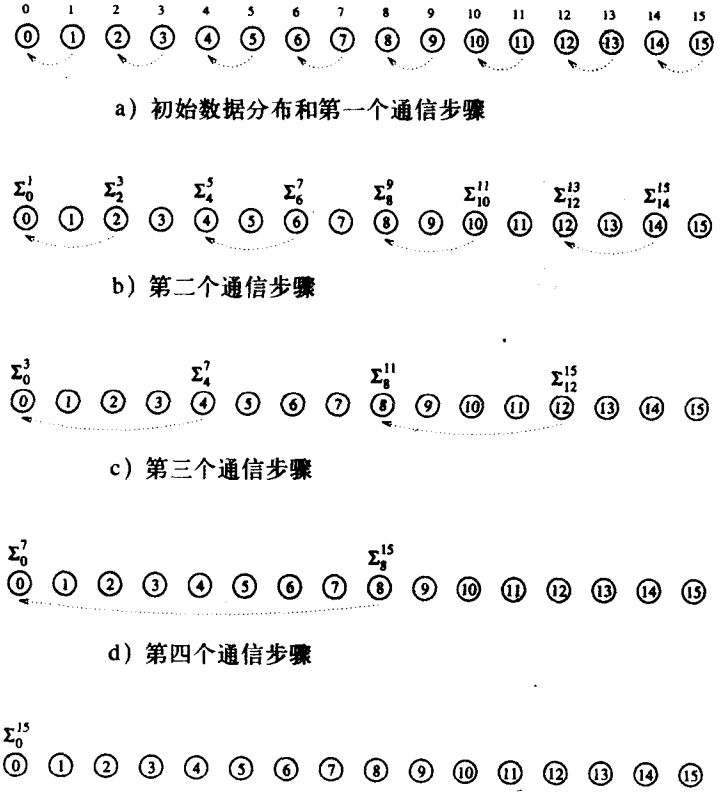


图5-2 用16个处理器计算16个数的总和，$\sum_i^j$ 表示标号从i到j的数的和

对某一个给定的问题，有多个串行算法可用，但这些算法并不是都能适合并行计算的。当使用串行计算机时，自然使用这样的串行算法，它以最少的时间求解问题。给定一个并行算法，用相对于在单个处理器上求解相同问题的最快的串行算法来判断并行性能是公正的。有时，我们并不知道求解问题时近似最快的串行算法，或由于其运算时间过长使之实际无法实现。这种情况下，我们采取最快的已知算法，对串行计算机而言它是最好的，这将是一种实用的选择。我们把并行算法求解问题同最好的串行算法求解同样问题的性能进行比较。定
198 义*加速比*S为最好串行算法求解问题的运算时间与用p个处理器求解相同问题的并行算法所花时间的比值。假定使用并行算法的p个处理器与串行算法使用的处理器相同。

例5.2 计算并行程序加速比

考虑并行冒泡排序（9.3.1节）的例子。假定10^5个记录的冒泡排序的串行形式耗时150秒，并且串行快速排序能在30秒内排序相同记录。如果冒泡排序的并行形式（也称为奇-偶排序），在4个处理器上花费40秒，那么看上去并行奇-偶排序算法的加速比为150/40或3.75。然而，这个结果会使人误解，因为在实际的并行算法中，得到的相对于最快串行算法的加速比为
199 30/40或0.75。 ■

理论上，加速比不可能超过处理器数p。如果最好串行算法在单个处理器上求解给定问题花费T_s个时间单位，那么，若没有一个处理器花费时间多于T_s/p，则加速比p能在p个处理器上获得。加速比大于p是可能的，只要每个处理器在求解问题时花费的时间少于T_s/p。在这种情况下，单个处理器模仿p个处理器，并且用少于T_s个时间单位求解问题。但这是矛盾的，因为按照定义，加速比是关于最好的串行算法来计算。如果T_s为串行算法的运算时间，那么问题不可能在单个处理器上用少于T_s时间求解。

实际上，有时可观察到加速比大于p的情况（此现象称为*超线性加速比*）。当串行算法执行的任务大于并行形式，或由于硬件特性使得任务不适合用串行实现时，这种情况常常发生。例如，问题的数据可能太大而不能放到单个处理器的高速缓存中，这样，由于使用较慢的储存单元使性能下降。但是，当在几个处理器之间划分数据时，单个的数据划分会小到能放入各个处理器的高速缓存中。本书后部分，由于考虑分级储存技术，我们就不再讨论超线性加速比。

例5.3 高速缓存的超线性效果

考虑在2处理器的并行系统上执行的一个并行程序。该程序试图求解规模为W的问题实例，其中，每个处理器上有64 KB高速缓存，程序的高速缓存命中率达到80%。假定高速缓存的延迟为2纳秒，DRAM延迟为100纳秒，有效的存储器存取时间为$2 \times 0.8+100 \times 0.2$或21.6纳秒。如果计算为内存受限的，并计算进行一个FLOP/内存访问，则处理速度相当于46.3 MFLOPS。现在考虑一种情况，即2个处理器中的每一个都有效地执行问题的一半（即$W/2$大小）。对这个问题规模，由于有效问题规模较小，高速缓存命中率有望更高。我们假定高速缓存的命中率为90%，剩余数据的8%来自本地DRAM，其余2%自远程DRAM（通信开销）。假定远程数据存取耗时400纳秒，这相当于$2 \times 0.9+100 \times 0.08+400 \times 0.02$或17.8纳秒的总存取时间。因此，每个处理器相应的执行速度为56.18 MFLOPS，总执行速度为112.36 MFLOPS。此时加速比可由比串行方式更快的速度来计算，即112.36/46.3或2.43。这是由于每个处理器上较小的问题规

模提高了高速缓存的命中率，我们注意到超线性加速比。 ■ 200

例5.4 搜索分解的超线性效果

考虑搜索非结构树的叶节点算法。每一叶有一个相应的标号并且目标是要寻找一个有专门标号的节点，该专门标号记为‘S’。这样的计算常常用来求解组合问题，其中标号‘S’可能包含这类问题的解（11.6节）。图5-3就是这种树。解节点是在树的最右边的叶。基于深度优先树遍历的串行方法求解此问题时将搜索整个树，即所有14个节点。如果花费时间t_c来访问一个节点，这个遍历时间为$14t_c$。现在考虑并行形式，此时，左子树由处理器0搜索，右子树由处理器1搜索。如果2个处理器以相同速度搜索树，在结果找到前，并行方式只搜索有阴影的节点。注意到并行算法做的全部工作只涉及到9个节点，即$9t_c$。假设根节点扩展是串行的，相应的并行时间为$5t_c$（一个根节点扩展，随后在每个处理器上伴随4个节点扩展）。因此，2个处理器执行的加速比为$14t_c/5t_c$或2.8。

这种超线性加速比是由于并行和串行算法所作的工作不同造成的。实际上，如果2个处理器的算法作为相同处理器的2个进程来实现，则对于这种问题的情况，算法的超线性将会消失。注意，当采用搜索分解时，由并行和串行算法完成的相关工作量依赖于解的位置，要找到一个对于全部事例都是最优的串行算法是不可能的。这种影响会在第11章更详细地分析。 ■

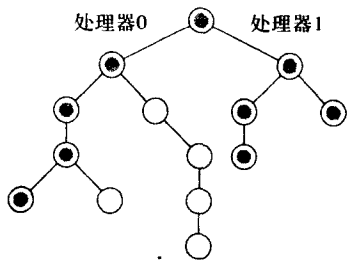


图5-3 对一个给定标号‘S’的节点，在2个处理器上使用深度优先遍历方式搜索一棵非结构树

注：在使用两个处理器的方法中，处理器0搜索左子树，处理器1搜索右子树，在找到解之前，只有阴影节点被扩展。相应串行方式扩展到整个树。显然，串行算法比并行算法做更多的工作。 201

5.2.4 效率

只有包含p个处理器的理想并行系统可能提供加速比p。但实际上，实现不了理想的性能，因为当执行一个算法时，处理器不可能投入100%的时间来进行算法的计算。如例5.1中所见，处理器计算n个数的总和所需的一部分时间是空闲时间（及用于在实际系统中通信）。*效率*（efficiency）是处理器被有效利用的部分时间的度量。它定义为加速比与处理器数目的比率。在理想的并行系统中，加速比等于p，效率等于1。实际上，加速比小于p而效率在0和1之间，它依赖于处理器被利用的效率。我们用符号E表示效率。效率E在数学上可表示为

$$E = \frac{S}{p} \tag{5-4}$$

例5.5 在n个处理器上相加n个数的效率

从公式(5-3)和前面的定义，在n个处理器上相加n个数的算法的效率是

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

在保留并行平台各种常数的同时，我们来推导并行运算时间、加速比以及效率。 ■

例5.6 图像边缘检测

已知一个$n \times n$像素的图像，检测边缘问题相当于应用一个3×3模板到每个像素。应用模板过程相当于把像素值同相应的模板值相乘，并沿整个模板求和（一个卷积运算）。这个过程用图5-4a及典型模板（图5-4b）说明。由于我们对每个像素应用9次乘法-加法运算，如果每个乘法-加法运算开销时间t_c，则在串行计算机上整个运算花费时间$9t_c n^2$。

对于这个问题，一个简单的并行算法是在处理器上均分图像，并且在每个处理器把模板应用于各自的子图像。注意为把模板应用到边界像素，处理器必须得到被分配给相邻处理器
202 的数据。特别地，如果一个处理器被分配一个垂直分块的$n \times (n/p)$维的子图像，它必须从左边的处理器存取一层n个像素，及从右边的处理器存取一层n个像素（注意，对在两端分配子图像的2个处理器，这些存取中的一个是多余的）。这在图5-4c说明。

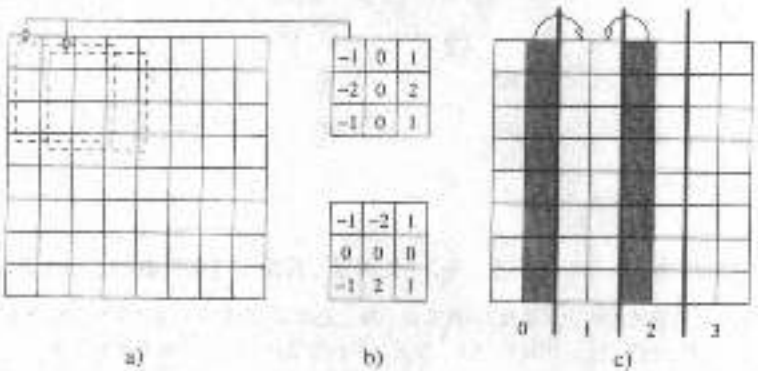


图5-4 边缘检测实例：a）8×8图像；b）边缘检测典型模板；c）图像划分给4个处理器，带阴影区域表示必须从相邻的处理器到处理器1进行通信的图像数据

在消息传递计算机上，算法分两步执行：i）用2个相邻处理器的每一个交换n个像素的层；ii）对局部子图像应用模板。第一步包含两个n字的消息（假设每个像素用一个字交换RGB数据）。它耗时$2(t_s + t_w n)$。第二步耗时$9t_c n^2/p$。因此，算法的总时间为

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

相应的加速比和效率为

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

和

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

■

5.2.5 成本

在并行系统中，我们定义求解问题的成本为并行运行时间和所用处理器数目的乘积。成 203
本反映每个处理器求解问题所花费时间的总和。效率也可表示为已知最快串行算法求解问题的执行时间与用p个处理器求解相同问题的成本之比。

在单处理器上求解问题的成本是已知最快串行算法的执行时间。如果在并行计算机上，求解问题的成本作为输入数据大小的函数，与在单处理器上的已知最快串行算法有着同样的的渐近增长（用Θ表示），那么就称并行系统是成本最优的（cost-optimal）。由于效率是串行成本与并行成本之比，成本最优的并行系统的效率为$\Theta(1)$。

有时成本是工作或处理器-时间乘积，而成本最优的系统也称作pT_p最优系统。

例5.7 用n个处理器求n个数之和的成本

例5.1给出的用n个处理器求n个数之和的算法有处理器-时间之积$\Theta(n \log n)$。由于这个运算的串行运算时间为$\Theta(n)$，这个算法不是成本最优的。 ■

成本最优是一个非常重要的实用概念，尽管它按渐近性定义。我们用下面的例子说明其应用。

例5.8 非成本最优算法的性能

考虑排序算法，用n个处理器对列表进行排序耗时$(\log n)^2$。由于（基于比较）排序的串行运算时间为$n \log n$，这个算法的加速比和效率分别为$n/\log n$和$1/\log n$。该算法的pT_p积为$n(\log n)^2$。因此，该算法不是成本最优的，但仅差一个$\log n$的因子。让我们考虑一个真实的情况：处理器的数目p远小于n。分配n个任务给$p < n$个处理器得出的并行时间小于$n(\log n)^2/p$。这个结果从下述事实推出：如果n个处理器花时间$(\log n)^2$，则一个处理器花时间$n(\log n)^2$；p个处理器花时间$n(\log n)^2/p$。这种形式对应的加速比为$p/\log n$。考虑用32个处理器对1024个数进行排序（$n = 1024$，$\log n = 10$）。期望得到的加速比仅为$p/\log n$或3.2。随n的增加，这个加速比减小。对$n = 10^6$，$\log n = 20$，加速比仅为1.6。很显然，存在一个不是成本最优甚至相差一个很小因子的重要成本（注意因子$\log p$甚至小于$\sqrt{p}$）。这说明成本最优在实用上的重要性。 ■ 204

5.3 粒度对性能的影响

例5.7说明一个非成本最优算法的实例。该例中的算法使用了与输入个数一样多的处理器，使用的处理器数过多。实际上，我们分配更多的输入数据块给处理器。这相当于在处理器上增加了计算粒度。根据处理器的数目，如果使用少于最大可用的处理器数来执行一个并行算法，则该系统被称为按处理器数目缩小（scaling down）一个并行系统。缩小并行

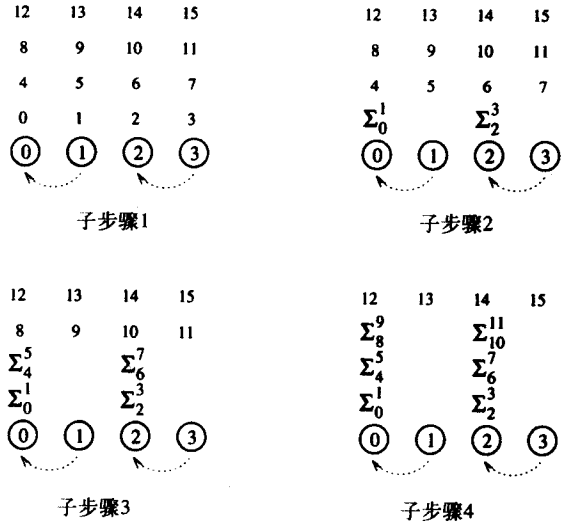
系统的一个简单方法是为每个处理器的输入单元设计一个并行算法，然后，用较少的处理器来模拟更多的处理器。如果有n个输入但只有p个处理器($p < n$)，通过假设n个虚拟处理器，我们能使用为n个处理器设计的并行算法，并使用p个物理处理器的每一个来模拟n/p个虚拟处理器。

当处理器数目减少n/p倍，每个处理器的计算增加n/p倍，因为每个处理器现在完成n/p个处理器的工作。如果虚拟处理器被适当地映射到物理处理器，总通信时间的增加不会多于n/p倍。总的并行运行时间至多增加n/p倍，而处理器-时间的乘积不会增加。因此，如果含n个处理器的并行系统是成本最优的，用p个处理器($p < n$)来模拟n个处理器也会保持成本最优。

这种简单增加计算粒度方法的缺陷是：如果并行系统开始不是成本最优的，在计算粒度增加后，它仍然不是成本最优的。这一点可用以下n个数相加的例子来说明。

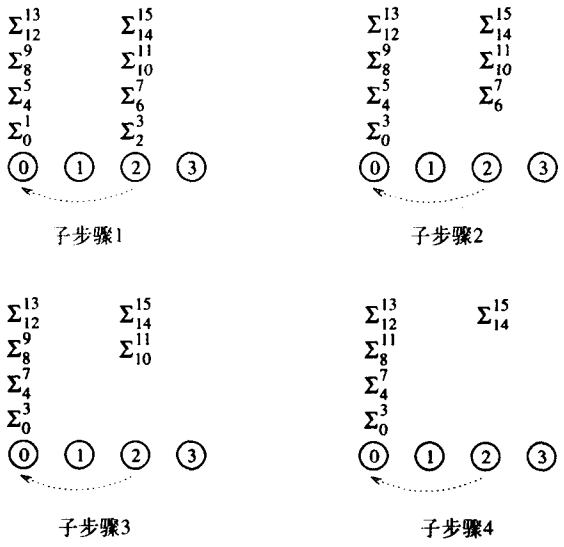
例5.9 用p个处理器求n个数之和

考虑用p个处理器求n个数之和的问题，其中$p < n$，并且n和p为2的幂。我们用与例5.1相同的算法，并在p个处理器上模拟n个处理器。对于$n = 16$和$p = 4$的求解步骤如图5-5所示。虚拟处理器i由标号为$i \bmod p$的物理处理器来模拟；待加的数也相应地进行分配。在最初的算法中，$\log n$步中的前$\log p$步用p个处理器在$(n/p)\log p$步中模拟。在剩余步中没有通信要求，因为在最初的算法中，进行通信的处理器由相同的处理器模拟；因此，剩下的数字在本地相加。在需要通信的步骤中，算法花费的时间为$\Theta((n/p)\log p)$，此后，单个处理器中剩下n/p个数相加，花费时间$\Theta(n/p)$。这样，这个并行系统所花的总并行执行时间为$\Theta((n/p)\log p)$。结果，其
205 成本为$\Theta(n \log p)$，它渐近地大于串行n个数相加的成本$\Theta(n)$。因此，该并行系统不是成本最优的。 ■

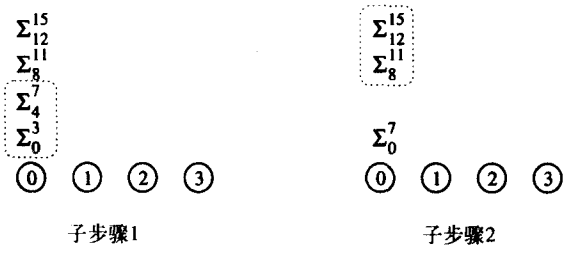


a）4个处理器模拟16个处理器的第一个通信步骤

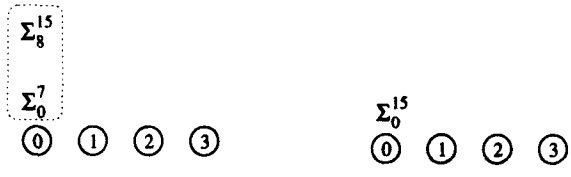
图5-5 用4个处理器模拟16个处理器计算16个数之和。$\sum_i^j$表示从i到j的连续标号的数之和



b）4个处理器模拟16个处理器的第二个通信步骤



c）第三步用两个子步骤模拟



d）第四步模拟　　e）最后结果

图5-5 （续）

例5.1说明，在n个处理器的机器上加n个数花费时间$\Theta(\log n)$。当用p个处理器模拟n个虚拟处理器($p < n$)时，预期的并行时间为$\Theta((n/p)\log n)$。然而，在例5.9中，这个任务可在时间$\Theta((n/p)\log p)$内完成，原因是不必模拟最初算法的每个通信步；有时，通信在由相同物理处理器模拟的虚拟处理器之间发生。对这种运算，没有相关的开销。例如，第3和第4步（图5-5c和d）的模拟不要求任何通信。然而，通信减少不足以使算法达到成本最优。例5.10说明，用一个更巧妙方法将数据分配给处理器，相同的问题（用p个处理器求n个数之和）能达到成本最优。

206

例5.10 求n个数之和的成本最优方法

用p个处理器求n个数之和的另一方法见图5-6 ($n = 16$, $p = 4$)。在算法的第一步，每个处
207 理器在时间$\Theta(n/p)$内本地相加各自的n/p个数。现在问题简化为在p个处理器上加p个部分和，用例5.1描述的方法可在时间$\Theta(\log p)$内完成。此算法的并行运算时间为

$$T_P = \Theta(n/p + \log p) \tag{5-5}$$

其成本为$\Theta(n + p \log p)$。当$n = \Omega(p \log p)$时，成本为$\Theta(n)$，这与串行运行时间相同。因此，这个并行系统是成本最优的。 ■

这些简单的例子说明，计算映射到处理器的方式可以决定并行系统是否为成本最优的。然而，注意我们并不能通过缩减处理器数，使所有非成本最优的系统变为成本最优的。

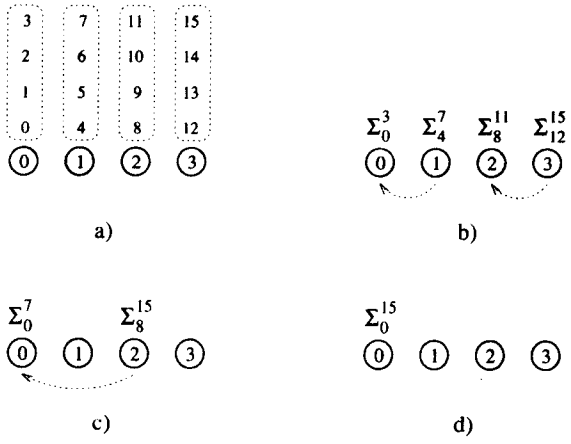


图5-6 用4个处理器计算16个数之和的成本最优方法

5.4 并行系统的可扩展性

通常，程序都是在较少的处理器中针对较小的问题进行设计和测试。然而这些程序所要求解的实际问题却非常大，并且机器包含很多处理器。虽然通过使用机器和问题的缩小版本，可以简化代码开发，但在缩减的系统中，程序的性能和正确性是很难确定的。本节中我们应用分析工具来分析各种评估并行程序可扩展性的方法。

例5.11 为什么性能推测如此困难?

208 考虑用64个处理器计算一个n点的快速傅里叶变换（FFT）的3个并行算法。图5-7表示当n的值增加到18 K时的加速比。保持处理器数不变，当n值较小时，从获得的加速比可推断出2进制交换算法和3-D转置算法的加速比最好。然而，当问题扩展到18 K个点或更多的点时，从图5-7可看出2-D转置算法能得到最好的加速比。（这些算法将在第13章详细讨论。） ■

当问题规模保持不变，而处理器数发生变化时，也能得到类似的结果。不幸的是，相对于特例而言，这种并行性能迹线是标准的，这样就使得基于有限观测数据的性能预测变得非常困难。

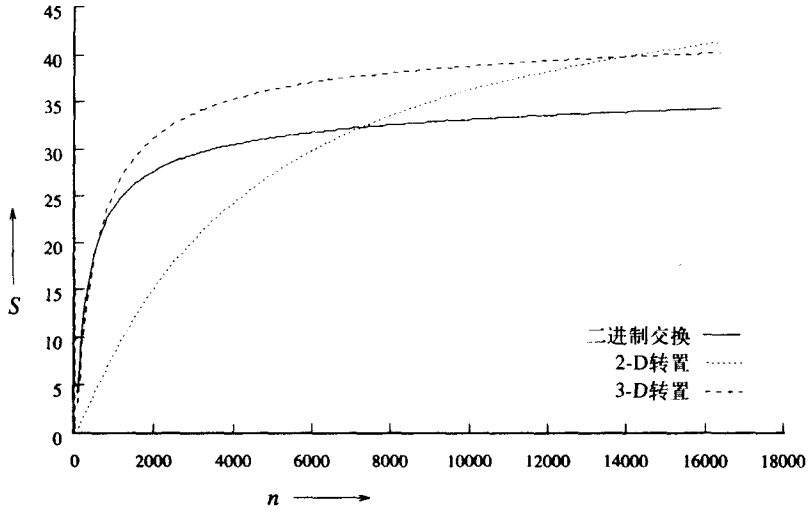


图5-7 用64个处理器的二进制交换、2-D转置和3-D转置算法的加速比比较，其中$t_c = 2$，$t_w = 4$，$t_s = 25$，$t_h = 2$（详见第13章）

5.4.1 并行程序的扩展特性

并行程序的效率可表示为

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

应用并行开销（公式(5-1)）的表达式，我们可以重写这个表达式为

$$E = \frac{1}{1 + \frac{T_o}{T_S}} \tag{5-6}$$

209

总开销函数T_o是一个p的递增函数。这是由于每个程序需包含一些串行部分。如果程序的串行部分花费时间t_{serial}，那么这期间所有其他处理器必定为空闲。相应的总开销函数为$(p-1) \times t_{serial}$。因此，总开销函数T_o至少随p线性增加。另外，由于通信、空闲以及超额计算，这个函数也可能随处理器的数目超线性地增加。公式(5-6)为我们提供了对并行程序的扩展的若干有趣的见解。首先，对给定问题规模（即T_s保持不变），当我们增加处理器数目时，T_o增加，在这种情况下，从公式(5-6)明显看出并行程序的总效率下降。这种对于给定问题规模随处理器数增加而效率却降低的特性，在所有并行程序中都是常见的。

例5.12 加速比和效率作为处理器数目的函数

考虑用p个处理器求n个数之和的问题。我们使用与例5.10相同的算法。为了说明实际的加速比，用常数代替渐进值。假设加2个数花费单位时间，算法的第一阶段（本地通信）花费时间大约为n/p。第二阶段包含$\log p$步，每一步有一次通信和一个加法。如果单个通信也花费单位时间，此阶段时间为$2\log p$。因此，我们得到并行时间、加速比和效率为

$$T_P = \frac{n}{p} + 2\log p \tag{5-7}$$

$$S = \frac{n}{\frac{n}{p} + 2\log p} \tag{5-8}$$

$$E = \frac{1}{1 + \frac{2p\log p}{n}} \tag{5-9}$$

对任何一对n和p，这些表达式可用来计算加速比和效率。图5-8为一些不同的n值和p值的$S(p)$函数曲线。表5-1所示为相应的效率。

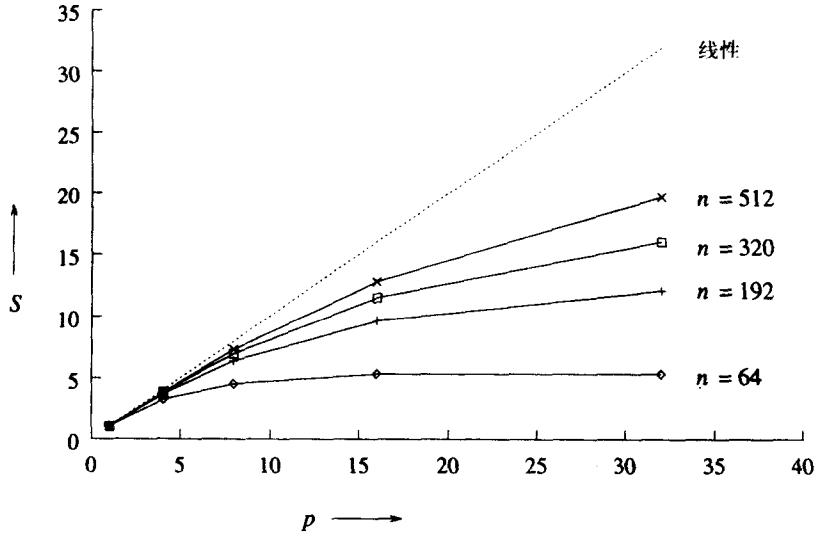


图5-8 数列相加时加速比作为处理器数目的函数

表5-1 用p个处理器相加n个数时，效率作为n和p的函数

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

图5-8和表5-1说明，加速比趋于饱和而效率下降，与Amdahl定律的结果（习题5.1）相同。再者，尽管加速比和效率随p增加而继续下降，但对相同处理器数目，同样问题的较大实例程能得到更大的加速比和效率。■

让我们分析保持处理器数不变而增加问题规模的效果。我们知道，总系统开销函数T_o是
210 问题规模T_s和处理器数p的函数。在很多情况下，T_o随T_s亚线性地变化。此时，如果问题规模增加而保持处理器数目不变，我们能看出效率增加。对这样的算法，通过同时增加问题的规模和处理器数目，应能保持效率不变。例如，表5-1用4个处理器加64个数的效率为0.80。如果处理器数目增加到8，问题的规模扩展为加192个数，效率保持为0.80。p增加到16和n增加到512会得到相同的效率。在很多并行系统中，都具备同时增加处理器数目和问题规模来保持效率为固定值的能力。我们称这种系统为可扩展（scalable）并行系统。并行系统的可扩展性

(scalability) 是与处理器数目成比例地增加加速比能力的度量。它反映一个并行系统有效地利用增加的处理资源的能力。 211

回顾5.2.5节，成本最优并行系统的效率为T_o。因此，并行系统的可扩展性与成本最优性是相关的。如果处理器数目和计算规模适当地选定，一个可扩展并行系统总是可成为成本最优的。例如，例5.10显示，当$n=\Omega(p \log p)$时，用p个处理器相加n个数的并行系统是成本最优的。例5.13显示，如果当p增加时，n与$\Theta(p \log p)$成比例增加，则同样的并行系统也是可扩展的。

例5.13 *n*个数相加的扩展性

对于在p个处理器上成本最优的n个数相加，$n=\Omega(p \log p)$。如表5-1所示，当$n=64$、$p=4$时，效率为0.80。此时，n和p之间关系为$n=8p \log p$。如果处理器数目增加到8，则$8p \log p=192$。表5-1显示，对于8个处理器，$n=192$时的效率确实为0.80。类似地，对于$p=16$，$n=8p \log p=512$的效率为0.80。因此，如果n随$8p \log p$增加，则这个并行系统在效率为0.80时维持成本最优。 ■

5.4.2 可扩展性的等效率度量

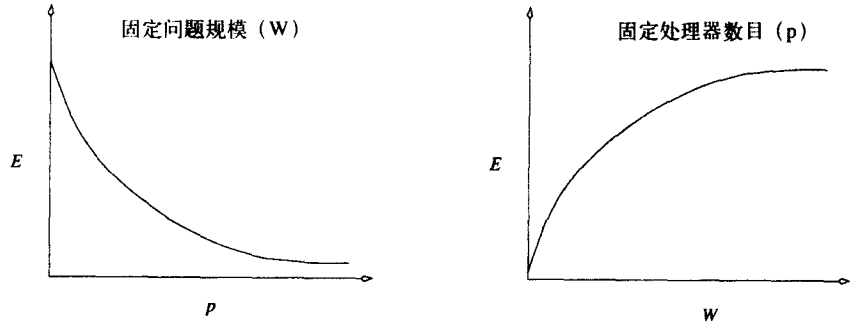


图5-9 效率的变化：a)对给定问题规模，效率随处理器数增加而变化；b)对给定处理器数目，效率随问题规模增加而变化。b)说明的现象在所有并行系统中并不常见

我们用以下两个结论来概括本节前面的讨论：

1）对于给定的问题规模，当增加处理器数目时并行系统的总效率下降。这种现象在所有并行系统是常见的。

2）在很多情况下，如果问题规模增加而处理器数目不变，并行系统的效率增加。 212

这两种现象分别用图5-9a和b说明。从这两个结论中，我们定义可扩展并行系统为：随着处理器数目的增加，问题规模也增加，而效率能保持不变。为了保持效率不变，我们还必需知道问题规模随着处理器数目以什么样的速度增加。对于不同的并行系统，为了保持效率不变，当处理器数增加时，问题规模必须以不同的速度增加。这个速度决定并行系统可扩展的程度。下面我们会指出，问题规模以较低的速度增加比以较高的速度更合乎需要。我们现在来定量地分析决定并行系统可扩展程度的度量。但在分析之前，我们必须准确地定义问题规

模（problem size）的概念。

问题规模 当我们分析并行系统时，常常会遇到被求解问题的规模这个概念。至今为止，我们非正式地使用了术语问题规模而没有给出确切的定义。把问题规模表示成输入大小的参数是一种简单的方法。例如，在$n \times n$阶的矩阵运算中，问题规模为n。这个定义的缺陷是问题规模的解释随问题的不同而变化。例如，输入大小加倍会导致矩阵乘法运算时间增加8倍，矩阵加法运算时间增加4倍（假设常规的$\Theta(n^3)$算法为最好的矩阵乘法算法，不考虑有更好渐近复杂度的更复杂算法）。

问题规模或大小的一致性定义应该是，不管是什么问题，问题规模加倍总意味着需要的计算量加倍。因此，我们按照求解问题所需的基本运算总数来表示问题规模。按这个定义，对于$n \times n$阶的矩阵乘法（假设用常规算法），问题规模为$\Theta(n^3)$，而对于$n \times n$阶矩阵加法，问题规模为$\Theta(n^2)$。为了使问题规模对于一个给定问题是唯一的，我们定义问题规模为：在单处理器上用最好的串行算法求解问题所需的基本计算步数，其中最好的串行算法在5.2.3节中定义。因为这是按串行时间复杂度定义的，问题规模为输入大小的函数。表示问题规模的符号为W。

在本章的剩余部分，我们假定执行算法的一个基本计算步骤要花费单位时间。这个假设不会影响对任何并行系统的分析，因为其他有关硬件的常数，如消息开始时间、每字转送时间以及每站时间，都能用相对于基本计算步骤所花的时间标准化。按这个假设，问题规模W
213 等于在串行计算机上用最快的已知算法求解问题花费的串行运算时间T_s。

等效率函数

并行执行时间可表示为问题规模、开销函数和处理器数目的函数。我们把并行运行时间写为

$$T_P = \frac{W + T_o(W, p)}{p} \tag{5-10}$$

加速比表达式为

$$\begin{aligned} S &= \frac{W}{T_P} \\ &= \frac{Wp}{W + T_o(W, p)} \end{aligned} \tag{5-11}$$

最后，效率表达式为

$$\begin{aligned} E &= \frac{S}{p} \\ &= \frac{W}{W + T_o(W, p)} \\ &= \frac{1}{1 + T_o(W, p)/W} \end{aligned} \tag{5-12}$$

在公式(5-12)中，如果问题规模不变而p增加，效率则下降，因为系统总开销T_o随p增加。如果W增加而处理器数不变，则对可扩展并行系统，效率增加。这是由于，在p值固定的情况下T_o比$\Theta(W)$变化慢。对这些并行系统，如果当p增加时W也增加，效率能保持在一个预期值（0与1之间）。

对不同的并行系统，为了保持固定的效率，W必须随着p以不同速度增加。例如在某些情况，为了防止效率随p增加而下降，W可能需要p按指数函数增加。这种并行系统的扩展性较差。因为在这些并行系统中，除非问题规模巨大，否则对于大量的处理器，很难获得良好的加速比。另一方面，如果W只需随p线性地增加，则并行系统的可扩展性很高。这是因为，对适度的问题规模，它能容易得到与处理器数目成比例的加速比。

对可扩展并行系统，如果公式(5-12)中的比值T_o/W保持固定值，效率就能保持一个定值(0与1之间)。如果要求出效率值E

$$\begin{aligned} E &= \frac{1}{1+T_o(W,p)/W} \\ \frac{T_o(W,p)}{W} &= \frac{1-E}{E} \\ W &= \frac{E}{1-E}T_o(W,p) \end{aligned} \tag{5-13}$$

214

令$K = E/(1-E)$为依赖于效率的一个常数。由于T_o为W和p的一个函数，公式(5-13)可重写为

$$W = KT_o(W,p) \tag{5-14}$$

由公式(5-14)可知，问题规模W可作为p的函数通过代数运算求出。这个函数表明，当p增加时要保持效率固定，W所需的增长速度。我们称这个函数为并行系统的**等效率函数**(isoefficiency function)。等效率函数决定并行系统能维持不变效率的难易程度，从而能获得与处理器数目成比例的加速比增长。小的等效率函数表明，小量地增加问题的规模，就足以有效地利用增加的处理器数，表明并行系统的可扩展性高。但是大的等效率函数表明并行系统的可扩展性差。对不可扩展的并行系统不存在等效率函数，因为在这种系统中不管问题规模增加多快，效率不能随着p的增加而保持任何固定值。

例5.14 n个数相加的等效率函数

在p个处理器中n个数相加的开销函数近似为$2p \log p$，如公式(5-9)和(5-1)所示。用$2p \log p$代入公式(5-14)中的T_o，得到

$$W = K2p\log p \tag{5-15}$$

这样，这个并行系统的渐近等效率函数为$\Theta(p \log p)$。这意味着，如果处理器数目从p增加到p'，问题规模（本例中为n）必须增加$(p' \log p')/(p \log p)$倍，才能得到在处理器数目为p时同样的效率。换言之，处理器数目增加p'/p倍要求n加$(p' \log p')/(p \log p)$倍，才能使加速比增加p'/p倍。 ■

在这个n个数相加的简单例子中，由通信导致的开销（今后都称为**通信开销**(communication overhead)）只是p的函数。通常，通信开销依赖于问题规模和处理器数目。一个典型的开销函数有相应于p和W的多个不同阶的特定项。此时，要获得等效率函数作为p的闭函数是烦琐的（甚至是不可能的）。例如，考虑一个假想的并行系统，其中$T_o = p^{3/2} + p^{3/4}W^{3/4}$。对于这个开销函数，公式(5-14)可重写为$W = Kp^{3/2} + Kp^{3/4}W^{3/4}$，显然，很难按$p$求解这个方程中的$W$。

215 回顾效率为常数的条件就是比值T_o/W保持不变。随着p和W的增加，只要T_o中没有一项比W增加快，效率就不会下降。如果T_o中有乘法项，我们用T_o中的每一项平衡W，并对单个项计算各自的等效率函数。T_o中要求问题规模随p以最高速度增加的部分决定并行系统的总渐近等效率函数。例5.15进一步说明这种等效率函数的分析方法。

例5.15 具有复杂开销函数的并行系统的等效率函数

考虑$T_o = p^{3/2} + P^{3/4}W^{3/4}$的并行系统。在公式(5-14)中只用$T_o$的第一项，得到

$$W = Kp^{3/2} \tag{5-16}$$

只用第二项，公式(5-14)产生下面W与p关系:

$$\begin{aligned} W &= Kp^{3/4}W^{3/4} \\ W^{1/4} &= Kp^{3/4} \\ W &= K^4p^3 \end{aligned} \tag{5-17}$$

为确保效率不会随处理器数目的增加而降低，开销函数的第一项和第二项要求问题规模分别按$\Theta(p^{3/2})$和$\Theta(p^3)$增加。两个速度渐近较高的项$\Theta(p^3)$，由于它包含由其他项决定的速度，所以它只能给出这个并行系统开销总的渐近等效率函数。读者可以实际证明，如果问题规模以这个速度增加，效率将为$\Theta(1)$，并且随着p的增加，低于这个速度的任何速度都会引起效率下降。 ■

在用单个表达式中，等效率函数捕捉并行算法以及实现并行算法的并行体系结构的特征。进行等效率分析后，我们能用较少的处理器测试并行程序的性能，然后预测在处理器很多时的性能。然而，等效率分析的用处并不局限于预测增加处理器对性能的影响。5.4.5节说明等效率函数如何描述并行算法中固有的并行化程度。我们将会在13章中见到，当硬件参数如处理器速度和通信频道发生变化时，等效率分析也能用来研究并行系统的性能。第11章讲述甚
216 至对于不能导出并行运行时间值的并行算法如何使用等效率分析。

5.4.3 成本最优性和等效率函数

我们在5.2.5节曾指出，如果在某一并行系统中，处理器数目和并行执行时间之积与在单处理器上已知的最快串行算法的执行时间成比例，那么并行系统是成本最优的。换言之，一个并行系统为成本最优，当且仅当

$$pT_P = \Theta(W) \tag{5-18}$$

把公式(5-10)的右端代入T_p，得到下述公式:

$$W + T_o(W, p) = \Theta(W)$$

$$T_o(W, p) = O(W) \tag{5-19}$$

$$W = \Omega(T_o(W, p)) \tag{5-20}$$

公式(5-19)和(5-20)表明，并行系统为成本最优，当且仅当其开销函数不会渐近地超过问题规模。这同公式(5-14)给出的条件非常类似，即当在并行系统中增加处理器数目时保持效率

为一个固定值所需的条件。如果公式(5-14)产生等效率函数$f(p)$，则从公式(5-20)可知，当并行系统扩展时，为保证其成本最优，必须满足关系$W = \Omega(f(p))$。下面的例子进一步说明了成本最优性与等效率函数之间的关系。

例5.16 成本最优性与等效率之间的关系

考虑p个处理器上n个数相加问题的成本最优解，这在例5.10已经提出。对这个并行系统，$W \approx n$，而$T_o = \Theta(p \log p)$。从公式(5-14)，其等效率函数为$\Theta(p \log p)$；即是说，为保持固定效率，问题规模必须按$\Theta(p \log p)$增加。在例5.10中，当$W = \Omega(p \log p)$时，我们已推导出成本最优的条件。 ■

5.4.4 等效率函数的下界

以前我们讨论过，较小的等效率函数表示较高的可扩展性。因此，一个理想的可扩展并行系统必须有最低可能的等效率函数。对一个由W个任务单元组成的问题，处理器不超过W个时，它们能成本最优地使用；其余的处理器将空闲。当处理器数目增加时，如果问题规模以小于$\Theta(p)$的速度增加，则处理器数目最终会超过W。即使对一个没有通信及其他开销的理想并行系统，效率也会下降，因为处理器增加超过$p = W$时，超出的处理器将空闲。因此，为保持固定的效率，问题规模必须至少像$\Theta(p)$一样快地渐近地增加。因此，$\Omega(p)$是等效率函数的渐近下界。由此可知，理想可扩展的并行系统的等效率函数为$\Theta(p)$。 217

5.4.5 并发度和等效率函数

通过能并发执行的运算步数，$\Omega(p)$的下界会影响并行系统的等效率函数。在并行算法中，任何时刻能同时执行的任务的最大数称为*并发度*（degree of concurrency）。并发度是问题规模为W时算法能并行执行的运算步数的一个度量；它不依赖于并行体系结构。如果$C(W)$为并行算法的并发度，则对规模为W的问题，不超过$C(W)$个处理器能被有效地利用。

例5.17 并发性对等效率函数的影响

考虑用高斯消元法（8.3.1节）求解n个变量的n个方程组。计算总量为$\Theta(n^3)$。但这n个变量必须一个接一个地消去，并且消去每个变量需要$\Theta(n^2)$次计算。因此，在任意时刻，至多有$\Theta(n^2)$个处理器在工作。由于这个问题的$W = \Theta(n^3)$，并发度$C(W)$为$\Theta(W^{2/3})$，且最多$\Theta(W^{2/3})$个处理器被有效利用。另一方面，给定p个处理器，为了全部使用它们问题规模至少应该为$\Omega(p^{3/2})$。因此，由并发性导致的这个计算的等效率函数为$\Theta(p^{3/2})$。 ■

只有并行算法的并发度为$\Theta(W)$时，由并发性所导致的等效率函数才是最优的（即$\Theta(p)$）。如果一个算法的并发度小于$\Theta(W)$，则由并发性所导致的等效率函数比$\Theta(p)$更差（即更大）。此时，并行系统的总等效率函数由并发度、通信及其他开销所导致的等效率函数的最大值给出。

5.5 最小执行时间和最小成本最优执行时间

假如处理器数量不受限制，我们的兴趣通常在于问题求解速度的快慢，或并行算法的最小可能执行时间的多少。对于给定的问题规模，如果我们增加处理器数，要么并行执行时间继续减小并渐近地接近一个最小值，要么并行时间在达到最小值后开始增加（习题5.12）。通 218

过表达式T_p对p进行微分并使导数为零（假设函数$T_p(W, p)$对p是可微的），对给定的W我们能确定最小并行运行时间T_p^{min}。T_p最小时，处理器数目可由下面的公式确定：

$$\frac{\mathrm{d}}{\mathrm{d}p}T_P = 0 \tag{5-21}$$

令P_0为满足公式(5-21)的处理器数目的值。T_p^{min}的值可以通过在表达式T_p中用P_0代替p得到。在下面的例子中，我们对n个数相加的问题推导T_p^{min}的表达式。

例5.18 n个数相加的最小执行时间

按照例5.12的假设，p个处理器上n个数相加问题的并行运行时间近似为

$$T_P = \frac{n}{p} + 2\log p \tag{5-22}$$

在公式(5-22)右端对p求导数并令其等于零，我们得到p的解如下：

$$\begin{aligned} -\frac{n}{p^2} + \frac{2}{p} &= 0 \\ -n + 2p &= 0 \\ p &= \frac{n}{2} \end{aligned} \tag{5-23}$$

把$p = \frac{n}{2}$代入公式(5-22)中，得到

$$T_P^{min} = 2\log n \tag{5-24}$$

■

在例5.18中，$p = p_0$时处理器-时间乘积为$\Theta(n \log n)$，它大于问题的串行复杂度$\Theta(n)$。因此，对于导致最小并行运行时间的p值，并行系统不是成本最优的。假定问题求解是成本最优的，我们来推出一个重要结果，即给出并行运行时间的下界。

令$T_p^{cost_opt}$为问题能用成本最优并行系统求解的最小时间。从5.4.3节中关于成本最优和等效率函数等价的讨论可知，如果并行系统等效率函数为$\Theta(f(p))$，则只有$W = \Omega(f(p))$时，规模为W的问题能成本最优地求解。换言之，给定问题规模W，成本最优求解要求$p = O(f^{-1}(W))$。由于成本最优并行系统（公式(5-18)）的并行运行时间为$\Theta(W/p)$，成本最优地求解规模为W的问题的并行运行时间下界为

219

$$T_P^{cost_opt} = \Omega\left(\frac{W}{f^{-1}(W)}\right) \tag{5-25}$$

例5.19 n个数相加的最小成本最优执行时间

如例5.14的推导，并行系统的等效率函数$f(p)$为$\Theta(p \log p)$。如果$W = n = f(p) = p \log p$，则$\log n = \log p + \log\log p$。忽略二重对数项，$\log n \approx \log p$。如果$n = f(p) = p \log p$，则$p = f^{-1}(n) = n/\log p \approx n/\log n$。因此，$f^{-1}(W) = \Theta(n/\log n)$。根据成本最优和等效率函数之间的关系，能被用来成本最优地求解问题的最大处理器数目为$\Theta(n \log n)$。在公式(5-2)中用$p = n/\log n$，得到

$$\begin{aligned} T_P^{cost_opt} &= \log n + \log\left(\frac{n}{\log n}\right) \\ &= 2\log n - \log\log n \end{aligned} \tag{5-26}$$

■

令人感兴趣的是注意n个数相加的T_p^{min}和$T_p^{cost_opt}$均为$\Theta(\log n)$（公式(5-24)和(5-26)）。因此，对这个问题，成本最优解也是渐近最快解。对于给定的问题规模，如果使用大于等效率函数所要求的p值，并行执行时间并不能减小（由于成本最优与等效率函数之间的等价性）。一般说来，这个结论对于并行系统是不成立的，但是很可能有$T_p^{cost_opt} > \Theta(T_p^{min})$。下面的例题将说明这种并行系统。

例5.20 $T_p^{cost_opt} > \Theta(T_p^{min})$的并行系统

考虑例5.15假设的并行系统，其中

$$T_o = p^{3/2} + p^{3/4}W^{3/4} \tag{5-27}$$

从公式(5-10)，这个系统的并行运行时间为

$$T_P = \frac{W}{p} + p^{1/2} + \frac{W^{3/4}}{p^{1/4}} \tag{5-28}$$

用例5.18的方法，

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}p}T_P = -\frac{W}{p^2} + \frac{1}{2p^{1/2}} - \frac{W^{3/4}}{4p^{5/4}} &= 0 \\
-W + \frac{1}{2}p^{3/2} - \frac{1}{4}W^{3/4}p^{3/4} &= 0 \\
p^{3/4} &= \frac{1}{4}W^{3/4} \pm (\frac{1}{16}W^{3/2} + 2W)^{1/2} \\
&= \Theta(W^{3/4}) \\
p &= \Theta(W).
\end{aligned}$$

220

从前面的分析可知$p = \Theta(W)$。在公式(5-28)中，用p_0代替p，得

$$T_P^{min} = \Theta(W^{1/2}) \tag{5-29}$$

根据例5.15的结果，这个并行系统的总等效率函数为$\Theta(p^3)$，这意味着能成本最优地使用的最大处理器数目为$\Theta(W^{1/3})$。在公式(5-28)中，将$p = \Theta(W^{1/3})$代入，我们得到

$$T_P^{cost_opt} = \Theta(W^{2/3}) \tag{5-30}$$

比较公式(5-29)和(5-30)可知$T_p^{cost_opt}$渐近大于T_p^{min}。 ■

本节中，我们看到了两种类型并行系统的例子：一类为$T_p^{cost_opt}$渐近等于T_p^{min}，另一类为$T_p^{cost_opt}$渐近大于T_p^{min}。本书提出的大多数并行系统为第一类。使用渐近大于等效率函数所需要的处理器数，就能使执行时间减少一个数量级的并行系统是非常少见的。

在对任何并行系统推导最小执行时间时，重要的是要了解，能使用的最大处理器数目受到并行算法的并发度$C(W)$的限制。对并行系统而言（习题5.13和5.14），p_0大于$C(W)$的可能性很大。此时，p_0的值毫无意义，并且T_p^{min}可表示为

$$T_P^{min} = \frac{W + T_o(W, C(W))}{C(W)} \tag{5-31}$$

5.6 并行程序渐近分析

到目前为止，我们已积累了一个非常有效的工具库用于定量分析算法的性能和可扩展性。下面将说明如何利用这些工具来评估求解给定问题的并行程序。通常，我们忽略常数因子而
221 关心各个量的渐近特性，因为这样往往更能了解各种并行程序的相对优点和缺点。

考虑n个数的列表的排序问题。这个问题的最快串行程序的运行时间为$O(n \log n)$。对于一个给定列表的排序，我们考查4种不同的并行算法A1、A2、A3和A4。4种算法的并行运行时间以及它们可用的处理器的数目在表5-2中给出。这个练习的目的是确定哪个算法最好。如果速度是最简单的度量，则T_p最小的算法最好。按此度量，算法A1最好，随后依次为A3、A4和A2。这也反映在这样的事实中，这组算法的加速比也有同样的顺序。

然而，在实际情况下，我们很少能有像算法A1所要求的n^2个处理器。再者，资源利用率是实际程序设计的一个重要方面。所以我们必需考查每个算法的效率如何。这种算法的评估度量提出了一个完全不同的景象。根据此度量，算法A2和A4最好，其次为A3和A1。表5-2最后一行表示算法的代价。从这行中可以看出，算法A1和A3的代价高于串行运算时间$n \log n$，因此，这两个算法没有一个是成本最优的。但是，算法A2和A4是成本最优的。

这组算法说明，重要的是先了解并行算法分析的目的和使用适当的度量。这是因为使用不同的度量通常可能导致相互矛盾的结果。

表5-2 对给定的数字列表排序的4种不同算法的比较。表中显示处理器数目、并行运行时间、加速比、效率和pT_p乘积

算法	A1	A2	A3	A4
p	n^2	$\log n$	n	$\sqrt{n}$
T_p	1	n	$\sqrt{n}$	$\sqrt{n} \log n$
S	$n \log n$	$\log n$	$\sqrt{n} \log n$	$\sqrt{n}$
E	$\log n/n$	1	$\log n/\sqrt{n}$	1
pT_p	n^2	$n \log n$	$n^{1.5}$	$n \log n$

5.7 其他可扩展性的度量

人们已提出了多种并行系统可扩展性的其他度量。这些度量专门适用于不同的系统要求。
222 例如，在实时应用中，其目的是将某一系统扩充为在特定时间内完成某个任务。多媒体解压缩就属于这种应用，其中，MPEG流必须以每秒25帧的速度解压缩。因此，一个并行系统必须在40毫秒内解码单帧（或通过缓冲，以平均40毫秒1帧通过缓冲帧）。实时控制也要用到这种应用，其中控制向量必须实时生成。另外，还有些可扩展度量考虑了物理结构的限制。很多应用中，问题最大规模不受时间、效率或基本模型所限制，但是受计算机可用内存的限制。在这些情况下，度量对可用内存（与处理器数目）的增长函数作出假设，并估计并行系统的性能如何随这样的扩展变化。本节将研究一些相关的度量以及它们在各种并行应用中如何使用。

可扩展加速比 这种度量被定义为当问题规模随处理器数目线性增加时获得的加速比。如果可扩展加速比与处理器数目的关系曲线近似为线性，则认为并行系统是可扩展的。如果考虑的并行算法具有线性或近似线性等效率函数，则这种度量与等效率有关。此时，可扩展加

速比度量提供的结果很接近等效率分析的结果，并且可扩展加速比与处理器数目呈线性或近似线性关系。对于等效率较差的并行系统，由这两种度量提供的结果可能差异较大。此时，可扩展加速比与处理器数目的关系曲线是亚线性的。

人们已研究了可扩展加速比两个广义的概念。它们在方法上不同于问题规模随着处理器数目扩展的方法。在其中一个方法中，问题的规模增加到存满并行计算机的可用内存。这里假定系统的聚集内存随处理器数目而增加。在另一种方法中，问题规模随p的增加取决于一个执行间的上界。

例5.21 矩阵-向量相乘算法中内存和时间受限的可扩展加速比

$n \times n$阶矩阵与一维向量相乘的串行运行时间为$t_c n^2$，其中t_c为单个乘法-加法运算的时间。使用简单的并行算法，相应的并行运行时间为

$$T_P = t_c \frac{n^2}{p} + t_s \log p + t_w n$$

加速比S为

$$S = \frac{t_c n^2}{t_c \frac{n^2}{p} + t_s \log p + t_w n} \tag{5-32}$$

223

算法总的内存需求为$\Theta(n^2)$。下面我们考虑扩展问题规模的两种情况。在内存受限的扩展中，假设并行系统的内存随处理器数目线性增加，即$m = \Theta(p)$，对目前大多数并行平台而言，这种假设是合理的。由于$m = \Theta(n^2)$，对某个常数c可得$n^2 = c \times p$。因此，可扩展加速比S'为

$$S' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + t_w \sqrt{c \times p}}$$

或

$$S' = \frac{c_1 p}{c_2 + c_3 \log p + c_4 \sqrt{p}}$$

在限制情况下，$S' = O(\sqrt{p})$。

在时间受限的扩展情况下，$T_p = O(n^2/p)$。由于这是恒定的限制，所以$n^2 = O(p)$。我们注意这种情况与内存受限的情况相同。这种情况的发生是因为内存和算法的运行时间渐近相等。

■

例5.22 矩阵-矩阵相乘算法中内存和时间受限的可扩展加速比

两个$n \times n$阶矩阵相乘的串行运算时间为$t_c n^3$，其中t_c如前所述，为单个乘法-加法运算的时间。使用简单的并行算法，相应的并行运行时间为

$$T_P = t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}$$

加速比S为

$$S = \frac{t_c n^3}{t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}} \tag{5-33}$$

算法的总内存需求为$\Theta(n^2)$。让我们考虑问题扩展的两种情况。在内存受限的情况下，如前所述，假设并行系统的内存随处理器数目线性增加，即$m = \Theta(p)$。由于$m = \Theta(n^2)$，对某个常数c可得$n^2 = c \times p$。因此，可扩展加速比S'为

$$S' = \frac{t_c(c \times p)^{1.5}}{t_c \frac{(c\times p)^{1.5}}{p} + t_s \log p + 2t_w \frac{c\times p}{\sqrt{p}}} = O(p)$$

224 在时间受限的扩展情况下，$T_p = O(n^3/p)$。由于这是恒定的限制，所以$n^3 = O(p)$，或$n^3 = c \times p$（对某个常数c）。因此，时间受限的加速比S''为

$$S'' = \frac{t_c c \times p}{t_c \frac{c\times p}{p} + t_s \log p + 2t_w \frac{(c\times p)^{2/3}}{\sqrt{p}}} = O(p^{5/6})$$

这个例题说明：矩阵相乘时，内存受限的扩展产生线性加速比，而时间受限的扩展产生亚线性加速比。 ■

串行部分f 根据实验确定的串行部分f可用来定量分析固定问题规模并行系统的性能。下面我们来考虑一种情况，即一个计算的串行运行时间能被分解为一个完全并行的和一个完全串行的部分，即

$$W = T_{ser} + T_{par}$$

式中T_{ser}和T_{par}分别对应于完全串行和完全并行部分。据此可得

$$T_P = T_{ser} + \frac{T_{par}}{p}$$

这里假定了所有其他并行总开销，如超额计算和通信，都包含在串行部分T_{ser}中。从这些公式可得

$$T_P = T_{ser} + \frac{W - T_{ser}}{p} \tag{5-34}$$

并行程序的串行部分f定义为

$$f = \frac{T_{ser}}{W}$$

因此，由公式（5-34）可得

$$\begin{aligned} T_P &= f \times W + \frac{W - f \times W}{p} \\ \frac{T_P}{W} &= f + \frac{1-f}{p} \end{aligned}$$

由于$S = W/T_p$，我们有

$$\frac{1}{S} = f + \frac{1-f}{p}$$

求解f，得到

$$f = \frac{1/S - 1/p}{1 - 1/p} \tag{5-35}$$

容易看出，f的值越小越好，因为越小的f值能带来越高的效率。如果f随处理器数目增加，则它被看作通信开销增加以及可扩展性下降的指示器。 225

例5.23 矩阵-向量相乘的串行部分

由公式(5-35)和(5-32)，我们有

$$f = \frac{\frac{t_c n^2}{p} + t_s \log p + t_w n}{t_c n^2} \Big/ (1 - 1/p) \tag{5-36}$$

上述表达式可简化为

$$f = \frac{t_s p \log p + t_w n p}{t_c n^2} \times \frac{1}{p-1}$$

$$f \approx \frac{t_s \log p + t_w n}{t_c n^2}$$

该公式的分母为算法的串行运行时间，而分子对应于并行执行的开销。 ■

除这些度量外，文献中也提到很多其他的性能度量。我们把与这些有关的参考书目介绍给感兴趣的读者。

5.8 书目评注

为有效地使用当今大规模的并行计算机，必需通过增加更多的处理器来求解更大的问题。然而，当问题规模固定时，必需在效率和并行运算时间之间获得最好的折中。对规模固定的问题，多位研究者对性能进行了讨论[FK89, GK93a, KF90, NW88, TL90, Wor90]。在大多数情况下，由增加处理器数目所带来的额外计算能力能用来求解更大的问题。然而，在某些情况下，可以采用不同的方法增加问题规模，并且，使用各种限制可以引导提高相对于处理器数目的工作负载[SHG93]。Gustafson 等[GMB88, Gus88, Gus92]、Sun和Ni[sn90,sn93]以及Worley[Wor90, Wor88, Wor91] 研究了时间受限的扩展和内存受限的扩展（习题5.9）

一种重要的情况是最有效地利用并行系统；换言之，我们希望并行系统的总性能随p线性增加。这种可能性仅局限于可扩展的并行系统，这是这样一种系统，对任意大的p，只要增加问题规模，就能使效率保持不变。对这种系统，使用等效率函数或相关的度量是很自然的[GGK93, CD87, KR87b, KRS88]。研究人员发现，等效率分析对于描述各种并行算法的可扩展性很有用[GK91, GK93b, GKS92, HX98, KN91, KR87b, KR89, KS91b, RS90b, SKAT91b, TL90, WS89, WS91]。Gupta和Kumar[GK93a, KG94]阐明了在固定时间的情况下等效率函数的 226
相关性质。他们指出，如果等效率函数大于$\Theta(p)$，当执行时间保持固定时，不管用多少处理器，问题规模都不能无限地增加。许多其他研究者分析了与整体效率有关的并行系统的性能[EZL89, FK89, MS88, NW88, TL90, Zho89, ZRV89]。

Kruskal，Rudolph和Snir[KRS88]定义*并行效率*（parallel efficient, PE）问题的概念。他们的定义与等效率函数的概念有关。PE类中的问题在某些效率有具备多项式等效率函数的算法。类PE构成多项式等效率函数算法和较差的等效率函数算法之间的一个重要区别。对多种并行计算模型和互连方案，Kruskal等证明了PE的不变性。这个结果的一个重要推论是，在一种结

构中，具有多项式等效率的算法，也会在很多其他结构中具有多项式等效率。但是也存在例外，例如，Gupta 和Kumar[GK93b]指出，快速傅里叶变换算法在超立方体上有多项式等效率，但在网格上为指数等效率。

Vitter 和Simons[VS86]定义称之为PC*的一类问题。在PRAM上具有有效并行算法的问题就属于PC*。类P（多项式时间类）中的问题，如果有一个能使用多项式（通过输入大小）处理器数目的PRAM上的并行算法，并达到一个最小的效率$\in$，则问题属于PC*。任何PC*中的问题至少有一个并行算法，使得对于效率$\in$，其等效率函数存在并且是一个多项式。

在Kumar 和Gupta [KG94]的综述中，有关于多种可扩展性和性能度量的讨论。另外，除了到目前为止已引用的度量外，也提出了很多其他关于并行系统可扩展性和性能的度量[BW89, CR89, CR91, Fla90, Hil90, Kun86, Mol87, MR, NA91, SG91, SR91, SZ96, VC89]。

Flatt 和Kennedy[FK89, Fla90]指出，如果开销函数满足某个数学特性，则存在一个唯一的处理器数目的值p_0，对给定的W，T_p为最小值。他们的分析很大程度上依赖于T_o的一个特性，即$T_o > \Theta(p)$。Gupta 和Kumar[GK93a]指出，有的并行系统不满足这个条件，在这情况下，峰值性能点由所使用算法的并发度决定。

Marinescu和Rice[MR]提出了一个模型，用来描述和分析在MIMD计算机上的并行计算，模型中通过控制线程数p划分计算，事件数$g(p)$作为p的函数。他们假设每个事件都有一个固定的持续时间θ，因此$T_o = \theta g(p)$。根据对T_o的假设，他们得出结论：如果$T_o = \Theta(p)$，随着处理器数目增加，加速比将在某个定值饱和。并且，如果$T_o = \Theta(p^m)$，则加速比将渐近接近于零，其中$m \geq 2$。对更广泛的一类开销函数，Gupta和Kumar[GK93a]推广了这些结果。他们指出，如果$T_o < \Theta(p)$，则加速比将在某个最大值饱和。如果$T_o > \Theta(p)$，则加速比达到一个最大值，然
227 后随p单调下降。

Eager等[EZL89]，Tang和Li[TL90]提出了一个并行系统的最优化准则，以便在效率和加速比之间找到一个平衡。他们提出，在运算时间和效率的曲线上，好的操作点位于增加每个处理器带来的利益大约为1/2处，换言之，效率为0.5。他们得出，对于$T_o = \Theta(p)$，这等价于在ES积为最大的操作点，或在$P(T_p)^2$为最小的操作点。这个结论是Gupta和Kumar[GK93a]提出的更一般情况的特例。

在同时执行的不同可扩展性的多个应用中，Belkhal和Banerjee[BB90]，Leuze等[LDP89]，Ma和Shea[MS88]以及Park和Dowdy[PD89]解决了并行计算机处理器最优划分的重要问题。

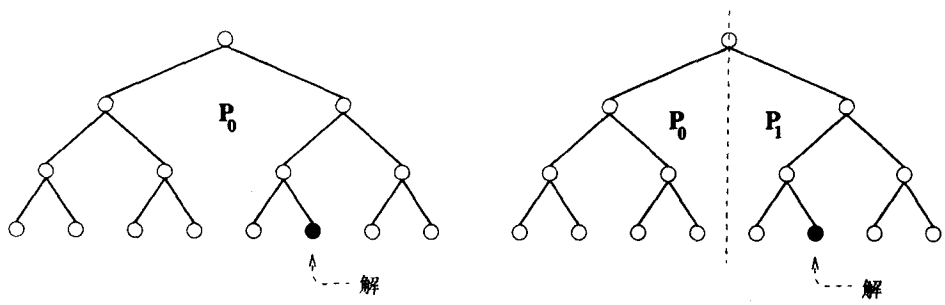
习题

5.1 （Amdahl定律[Amd67]） 如果规模为W的问题有串行部分W_S，证明，不管用多少处理器，W/W_S都是其加速比的上界。

5.2 （超线性加速比） 考虑图5-10a所示的搜索树，其中黑色节点代表解。

a）如果一个串行搜索树用标准深度优先搜索（DFS）算法实现（11.2.1节），如果遍历树的每条弧花费一个单位时间，那么需要多少时间才能找到解？

b）假设树在执行搜索任务的两个处理器之间划分，如图5-10b所示。如果两个处理器在它们各自树的一半执行一个DFS，那么需要多少时间才能找到解？加速比为多少？是否存在异常加速比？如果存在，试解释这样的异常。



a）用一个处理器的DFS　　b）用两个处理器的DFS

图5-10　并行深度优先搜索中的超线性(?)加速比

228

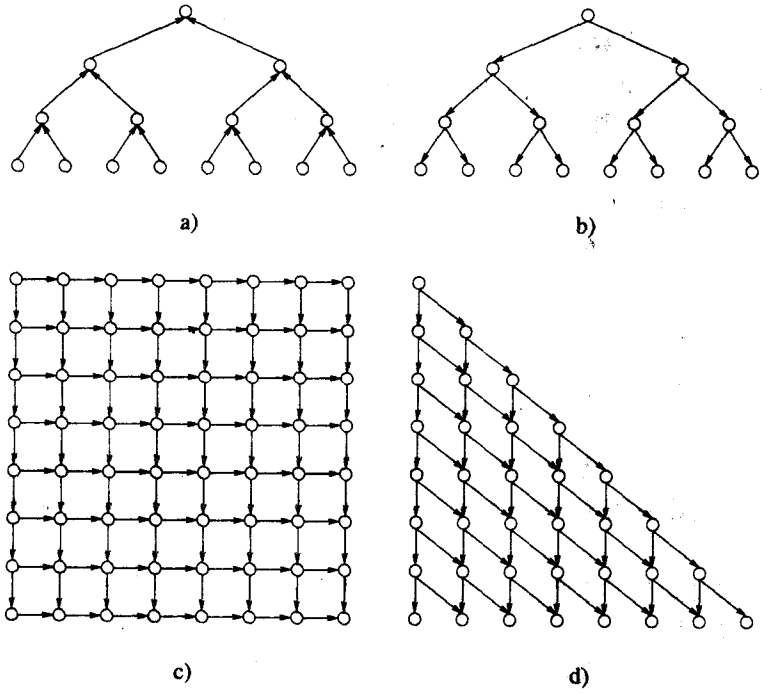


图5-11　习题5.3的依赖图

5.3 （**并行计算的DAG模型**）并行计算通常能用依赖图表示。图5-11显示4种这样的依赖图。如果一个程序能被分解为几个任务，则图的每一个节点代表一个任务。图的有向边表示任务间的依赖关系，或为保证任务执行得到正确结果需遵守的顺序。依赖图中的一个节点可以被调度执行，只要有入射边到达该节点的所有节点上的任务完成执行。如图5-11b，只有在根节点的任务完成后，从根部开始的第2层节点才能开始执行。任何无死锁的依赖图必须是一个有向无圈图（DAG）；即图中没有圈。如果有足够多的处理器，则所有调度执行的节点都能并行地工作。如果N为图中节点数，n为整数，则对图a）和b）$N = 2^n - 1$，对图c）$N = 2^n$，对图d）$N = n(n + 1)/2$（图a）和b）用$n = 4$绘制，图c）和d）用$n = 8$绘制）。假设对于每一种图表示的算法，每个任务花费一个单位时间以及处理器间的通信时间为零：

229 1）计算并发度。

2）如果可用的处理器数目不受限制，试计算最大可能的加速比。

3）计算加速比、效率及开销函数的值，如果处理器数目为 i）与并发度相同，ii）等于并发度的一半。

5.4 考虑包含p个处理器的并行系统，求解由W个工作单元组成的问题。证明，如果系统的等效率函数差于（大于）$\Theta(p)$，则问题不能用$p = \Theta(W)$成本最优地求解。反之，如果只对$p < \Theta(W)$能够成本最优地求解问题，则并行系统的等效率函数比线性差。

5.5 （**可扩展加速比**）*可扩展加速比*（scaled speedup）定义为当问题规模随处理器数目线性增加时获得的加速比；即是说，如果以W作为对单个处理器的基本问题规模，则

$$\text{可扩展加速比} = \frac{pW}{T_p(pW,p)} \tag{5-37}$$

对于p个处理器上的n个数相加问题（例5.1），假设对$p = 1$的基本问题是256个数的相加，画出加速比曲线。分别使用$p = 1$、4、16、64和256。假设在两个处理器之间1个数的通信花费10个时间单位，两个数的相加花费1个时间单位。对基本问题规模画出标准加速比曲线，并同可扩展加速比曲线比较。

提示：并行运行时间为$(n/p-1) + 11 \log p$。

5.6 对习题5.5画出第3条加速比曲线，其中问题规模按等效率函数$\Theta(p \log p)$扩大。对T_p使用相同的表达式。

提示：用这种方法扩大问题规模的可扩展加速比由下面的公式给出：

$$\text{等效率可扩展加速比} = \frac{pW \log p}{T_p(pW \log p,p)}$$

5.7 对p个处理器上n个数相加的问题画出等效率曲线，分别对应于标准加速比曲线（习题5.5），可扩展加速比曲线（习题5.5），以及当问题规模按等效率函数增加时的加速比曲线（习题5.6）。

5.8 增加处理器数目而不增加总工作负荷的缺陷是，加速比不会随处理器数目线性增加，且效率会单调下降。根据从习题5.5和5.7中得到的经验，讨论在通常情况下可扩展加速比是否
230 随处理器数目线性增加。如果在某并行系统中，可扩展加速比曲线与根据等效率函数增加问题规模所确定的加速比曲线相匹配，试对并行系统的等效率函数加以讨论。

5.9 （**时间受限的扩展**）对$p = 1$、4、16、64、256、1024和4096，使用习题5.5中关于T_p的表达式，如果总执行时间不超过512个时间单位，那么能求解的最大问题规模是多少？一般情况下，在固定的时间里，如果处理器数目没有限制，可能求解一个任意大的问题吗？为什么？

5.10 （**前缀和**）考虑在n个处理器上求n个数前缀和的问题（例5.1）。算法的并行运行时间、加速比以及效率是多少？假设两个数相加花费1个单位时间，两个处理器之间1个数的通信花费10个单位时间。算法是成本最优的吗？

5.11 在p个处理器上计算n个数的所有前缀和，其中$p < n$，设计一个前缀和算法（习题5.10）的成本最优版本。假设两个数相加花费1个单位时间，两个处理器之间1个数的通信花费10个单位时间，推导T_p、S、E、成本以及等效率函数的表达式。

5.12 [GK93a]试证明，对给定的问题规模，如果$T_o \leqslant \Theta(p)$，则并行执行时间将随p增加而继续减小，并且将渐近地接近一个常数值。再证明，如果$T_o > \Theta(p)$，则T_p随p先减小后增加。因此，它有一个明显的最小值。

5.13 对一个长度为n的输入序列（公式(13-4)中$t_s = 0$），具有p个处理器的FFT算法并行实现的并行运行时间为$T_p = (n/p)\log n + t_w(n/p)\log p$。算法能用于一个$n$点FFT的最大处理器数为$n$。问$p_0$值（满足公式（5-21）的$p$值）是多少？$t_w = 10$时$T_p^{min}$的值是多少？

5.14 [GK93a] 考虑有相同系统开销函数和不同并发度的两个并行系统。令两个并行系统的系统开销函数为$W^{1/3}p^{3/2} + 0.1\ W^{2/3}p$。画出$W = 10^6$，$1 \leqslant p \leqslant 2048$时$T_p$作为$p$的函数的曲线。如果第一个算法的并发度为$W^{1/3}$，第二个算法的并发度为$W^{2/3}$，计算两个并行系统的$T_p^{min}$值。在$T_p(p)$曲线上，在它们各自达到最小运行时间的点处，计算两个并行系统的成本和效率。 231

第6章 使用消息传递模式编程

人们已经开发了许多编程语言和程序库用于显式并行编程，这些编程语言及程序库的区别在于程序员可使用的地址空间，对并发活动的同步程度，以及程序的多重性。在并行计算机上编程，*消息传递编程模式*（message-passing programming paradigm）是最老也是最广泛使用的方法之一。它的根源可以追溯到并行处理的早期阶段，而其广泛应用可能是由于对底层硬件的需求最低。

在本章中，首先描述消息传递编程模式的基本概念，然后讨论使用标准的和广泛使用的消息传递接口的各种消息传递编程技巧。

6.1 消息传递编程的原理

消息传递编程模式有两个关键特性，其一是假设存在一个分块地址空间，其二是只支持显式并行化。

逻辑上把支持消息传递模式的计算机视为含有p个进程，每个进程都有独占的地址空间。这种观点的例子可以从工作站机群及非共享地址空间多计算机看到。分块地址空间的直接含义有两个。第一，每一数据单元必须属于空间的分块之一；因此，数据必须被显式地划分和存放。这会使编程更复杂，但能促进存取本地化，这一点对于在非UMA结构上获得高性能是至关重要的，因为在这种结构中，处理器存取本地数据的速度要比存取远程数据快得多。第二，所有的相互操作（只读或读/写）需要两个进程间的协作——拥有数据的进程及想要存取数据的进程。这种对协作的要求由于多种原因使编程变得更复杂。拥有数据的进程必须参与相互操作，哪怕它与请求进程的事件没有逻辑联系。在某些情况下，这种要求导致编出不自然的程序。特别，如果相互操作是动态的或非结构化的，这种模式下代码编写的复杂度就会非常高。然而，显式双向交互的一个主要优点在于程序员能完全知道非本地交互的全部成本，并更有可能设计算法（以及映射）使交互最大限度减少。这种编程模式的另一个主要优点在于它能够有效地在多种体系结构中实现。

消息传递编程模式要求由程序员显式地编写出并行程序。也就是说，程序员要分析串行算法/应用程序，对计算进行分解，并提取出并发操作。因此，使用消息传递模式的编程很困难，且要求很高的智能；但是，从另一方面讲，编写较好的消息传递程序往往能得到很好的性能和扩展到非常多的进程。

消息传递程序的结构 人们通常使用*异步*（asynchronous）或*松散同步*（loosely synchronous）模式来编写消息传递程序。在异步模式中，所有并发的任务都异步执行。这使得实现任何并行算法成为可能。然而，这样的程序很难理解，并且由于存在竞争条件可能导致不确定的行为。*松散同步*程序是这两个极端间的很好的折衷方案。在这种程序中，任务或任务的子集同步执行交互。然而，在这些交互之间，任务的执行是完全异步的。由于交互同步发生，很容易理解程序，许多已知的并行算法能够很自然地用松散同步程序来实现。

从最一般的形式来看，消息传递模式支持在p个进程的每个进程中执行一个不同的程序。

这样就给并行编程带来非常大的灵活性，但也造成编写并行程序的工作难以扩展。因此，绝大多数消息传递程序使用*单程序多数据*（single program multiple data，SPMD）方法来编写。在SPMD程序中，除了在很少进程（如“根”进程）以外，不同进程中执行的代码相同。这并不表示进程以锁步的方式工作。在极端的情况下，即使在SPMD程序中，每个进程也会执行不
234 同的代码（程序中含有一个与每个进程代码执行有关的大型选择语句）。但除了这种退化的选择外，绝大多数进程都执行同样的代码。SPMD程序可以是松散同步的，也可以是完全异步的。

6.2 操作构件：发送和接收操作

由于交互要通过发送及接收消息来完成，消息传递编程模式中最基本的操作就是send（发送）和receive（接收）。采用最简化的形式时，这些操作的原型定义如下：

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

sendbuf指向存储待发送数据的缓冲区，recvbuf指向存储待接收数据的缓冲区，nelems是待发送和接收的数据单元的数目，dest是接收数据进程的标识符，source是发送数据进程的标识符。

然而，仅仅停留在这些解释上会使如何实现这些功能的编程和性能细节过于简单化。为了进一步研究这些函数，让我们从下面简单的代码段例子开始，将一个进程的数据送到另一个进程中：

```
1          P0                                P1
2
3          a = 100;                          receive(&a, 1, 0)
4          send(&a, 1, 1);                   printf("%d\n", a);
5          a = 0;
```

在这个简单例子中，进程P0发送消息到进程P1，P1接收消息并输出消息。值得注意的是，进程P0在发送数据后立刻将a的值改变为0。发送操作的语义要求进程P1接收到的数据必须是100而不是0。也就是说，发送操作时a的值必须是进程P1接收到的值。

要保证发送和接收的语义看上去可能很直观。然而，由于发送和接收操作的实现方式不同，情况未必如此。绝大多数消息传递平台都有另外的硬件来支持消息的发送和接收。它们能支持DMA（直接内存访问）和使用网络接口硬件的异步消息传输。网络接口能允许消息从内存缓冲区传到需要的位置，而不需要CPU的干预。同样，DMA允许将某一内存位置处的数据复制到另一位置（如通信缓冲）而不需要CPU的支持（一旦DMA编程完毕）。因此，如果
235 发送操作对通信硬件编程，并在通信操作完成前返回，进程P1就有可能接收到0而不是100。

虽然这种情况是人们不希望看到的，但出于性能方面的原因，人们又支持这样的发送操作，在本节的剩余部分将讨论在这样的硬件环境下的发送和接收操作，并提出各种实现这些操作的细节及消息传递协议来保证发送和接收操作的语义。

6.2.1 阻塞式消息传递操作

要解决上面代码段提出的问题，一个简单的方法是让发送操作在代码语义上安全时才返回。这句话的意思不是说发送操作要等到接收方收到数据后才返回，只是表示发送操作阻塞，直到能够保证代码的语义不被破坏才返回，不管程序后面会出现什么情况。有两种机制可以

实现此方法。

1. 阻塞式无缓冲发送/接收

第一种方法是，发送操作在接收进程处遇到相应的接收操作前不返回。在这种情况下，消息发出，发送操作在通信操作完毕后返回。通常这个过程包含发送进程和接收进程之间的握手。发送进程发送一个与接收进程通信的请求。当接收进程遇到了接收目标，就会响应请求。发送进程在收到响应后启动传输操作。该操作如图6-1所示。因为在发送端和接收端都没有缓冲区，这个操作也称为无缓冲阻塞式操作（non-buffered blocking operation）。

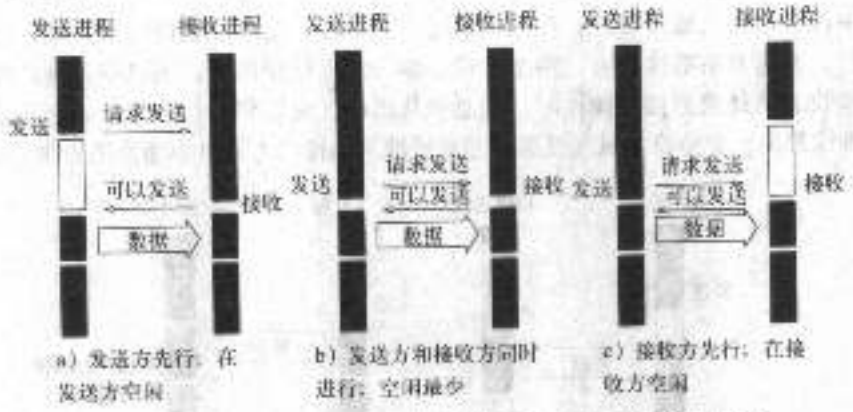


图6-1 阻塞式无缓冲发送/接收操作的握手。容易看出当发送方和接收方不能同时到达通信点时，会出现严重的空闲开销

236

阻塞式无缓冲操作中的空闲开销 图6-1中展示三种情形，即发送在接收发出前到达，发送和接收几乎同时发出，以及接收在发送到达前发出。在a）和c）中，注意在发送和接收进程都有显著的空闲。从图中也明显看出，只有当发送和接收几乎同时发出时，阻塞式无缓冲协议才适合。然而，在异步环境中，这一点是很难预测的。空闲开销是这种协议的一个主要缺点。

阻塞式无缓冲操作中的死锁 下面简单的消息交换可能导致死锁：

```
1          P0                                    P1
2
3          send(&a, 1, 1);                       send(&a, 1, 0);
4          receive(&b, 1, 1);                    receive(&b, 1, 0);
```

上面的代码段使进程P0和P1中都有a的值。然而，如果发送和接收操作用阻塞式无缓冲协议实现，P0处的发送就会等待P1中相应的接收，而进程P1处的发送也会等待P0处的相应接收，这样会导致无限期的等待。

由此可以推断，在阻塞式协议中死锁是很容易出现的，必须小心地破除上述的循环等待。在上面的例子中，可以通过将某一进程中的操作顺序更换一下来解除死锁，也就是将两个进程中的发送和接收顺序反过来。但这往往会使代码变得更复杂，且易出现错误。

2. 阻塞式有缓冲的发送/接收

可以利用发送端和接收端的缓冲区来解决上面提到的空闲和死锁问题。来看一个简单的例子，发送方有一个预先分配的缓冲用来进行消息通信。当遇到发送操作时，发送方只将数

据复制到指定的缓冲区，并在复制操作完成后返回。这时，发送方进程就能继续执行程序，因为它知道数据的任何变化都不会影响程序的语义。根据可用硬件资源的不同，实际的通信操作可以用多种方法完成。如果硬件支持异步通信（与CPU独立），那么当消息复制到缓冲区后网络传输就被启动。要注意的是，在接收端数据不能直接存储到目标位置，因为这样会破坏程序语义。用相反的方法也把数据复制到接收方的缓冲。当接收进程遇到一个接收操作，它会检查消息是否在接收缓冲中。如果在，数据就被复制到目标位置。这个操作如图6-2a所示。

237 在上面说明的协议中，发送方和接收方都使用缓冲区，通信操作由专用的硬件处理。有时计算机中没有这样的通信硬件，在这种情况下，只在一端缓冲也能减少开销。例如，遇到发送操作时，发送方中断接收方，两个进程都参与到通信操作中，消息存储到接收方端的缓冲区。当接收方最终遇到接收操作时，消息就从缓冲区复制到目标位置。这个协议如图6-2b所示。不难设想出一个协议，只在发送方考虑缓冲，而接收方通过中断发送方来启动传输。

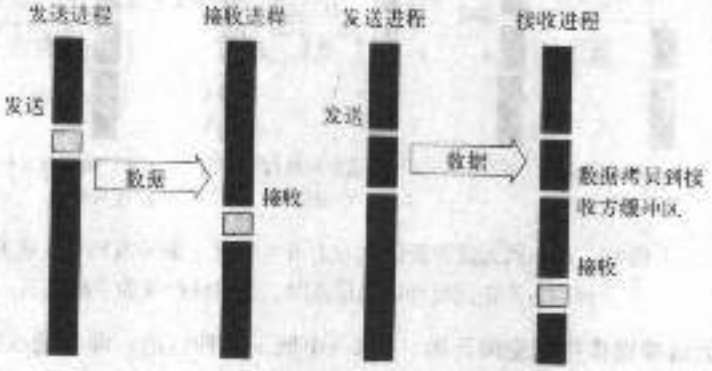


图6-2 阻塞式缓冲的传输协议：a）有通信硬件，在发送端和接收端都有缓冲区；b）无通信硬件，发送方中断接收方并在接收方端的缓冲区存储数据

容易看出，缓冲协议以增加缓冲管理开销的成本减少空闲开销。通常，如果并行程序高度同步（即发送和接收几乎在同时发出），无缓冲的发送的性能可能比有缓冲的发送好。但是，在一般的应用程序中，情况并非如此，除非缓冲区容量成为一个大问题，都采用有缓冲的发送。

例6.1 消息传递中有限缓冲的影响

考察下面的代码段：

```
P0                                    P1
for (i = 0; i < 1000; i++) {          for (i = 0; i < 1000; i++) {
  produce_data(&a);                     receive(&a, 1, 0);
  send(&a, 1, 1);                       consume_data(&a);
}                                     }
```

在这段代码中，进程P0产生1000个数据项并由进程P1消费它们。然而，如果进程P1进入循环的速度太慢，P0可能已经把所有的数据发送。如果有足够的缓冲区空间，则两个进程都
238 能继续；然而，如果缓冲不够（即缓冲溢出），发送方就只能阻塞，直到相应的接收操作发出，

这样才能腾空缓冲区空间。这通常会导致不可预见的开销以及性能降低。一般情况下，编写只有有限缓冲区需求的程序是一种好的主意。 ■

有缓冲发送和接收操作中的死锁 虽然缓冲减少死锁的出现，但仍可能编写造成死锁的代码。这是因为，和无缓冲时的情况一样，接收调用总是阻塞着（为了保证代码语义上的一致性）。因此，如下的简单代码段也能造成死锁，因为两个进程都等待着接收数据，但没有进程发出数据。

```
1       P0                                  P1
2
3       receive(&a, 1, 1);                  receive(&a, 1, 0);
4       send(&b, 1, 1);                     send(&b, 1, 0);
```

再次看出，这样的循环等待必须破除。然而，在本例中，死锁仅由对接收操作的等待造成。

6.2.2 无阻塞式消息传递操作

在阻塞协议中，保证代码语义正确性的开销以空闲（无缓冲）形式或缓冲区管理（有缓冲）形式出现。通常，可能要求程序员保证语义的正确性，并提供快速的发送/接收操作以减少开销。这种类型的无阻塞协议在语义上这样做是安全的以前，从发送或接收操作返回。因此，用户一定要小心，不要修改可能参与通信操作的数据。无阻塞式操作通常伴随着一个check-status（状态检查）操作，它显示前一个发起的传输的语义是否可能受到损害。当从无阻塞式发送或接收操作返回时，进程可以不受约束地执行不依赖于操作完成的任何计算。如果有必要的话，在程序的后面可以检查无阻塞式操作是否完成，并等待它的完成。

如图6-3所示，无阻塞式操作自身可以是有缓冲的，也可以是无缓冲的。在无缓冲情况下，如果一个进程想要发送数据到另一个进程，它只需发出待发的消息，并返回到用户程序中。程序然后可以做其他有用的工作。当后来相应的接收发出后，通信操作启动。当这个操作完成后，状态检查操作显示此时程序员对这个数据进行处理是安全的。这个传输如图6-4a所示。

比较图6-4a和图6-1a容易发现，在阻塞式操作中，进程等待相应接收的空闲时间可以用于计算，只要它不更新正发送的数据。这样只要对程序进行某些结构重组，就能减少与前者相 239
关的主要瓶颈。如果有专门的通信硬件，则无阻塞式操作的好处就会更多。这一点在图6-4b中说明。此时，通信开销几乎可以完全被无缓冲操作屏蔽。但是，在这种情况下，接收操作过程中被接收的数据是不安全的。

无阻塞式操作也可以和有缓冲协议一起使用。在这种情况下，发送方启动一个DMA操作并立即返回。DMA操作一结束，数据就成为安全的。在接收端，接收操作启动一个从发送方的缓冲区到接收方的目标位置的传输。在无阻塞式操作中使用缓冲区可以减少数据不安全的时间。

典型的消息传递库，如消息传递接口（MPI）和并行虚拟机（PVM），既使用阻塞式操作又使用无阻塞式操作。阻塞式操作使编程更安全，更简单，而无阻塞式操作可以通过屏蔽通信开销来进行性能优化。但是，使用无阻塞协议时一定要小心，因为不安全地存取处于通信过程中的数据会引发错误。



图6-3 用于发送和接收操作的可能协议空间

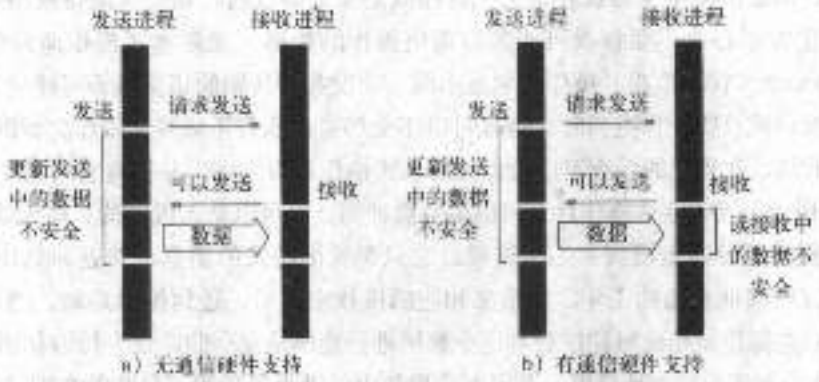


图6-4 无阻塞式无缓冲的发送与接收操作

6.3 MPI：消息传递接口

240 许多早期的商用并行计算机是基于消息传递体系结构的，因为相对于共享地址空间体系结构，它的成本更低。对这些计算机而言，消息传递是自然的编程模式，这样就产生了许多不同的消息传递程序库。事实上，消息传递演变成现代形式的汇编语言，每个硬件制造商都提供他自己的程序库，这些库在它们自己的硬件上工作得很好，但是与其他制造商提供的并行计算机不兼容。由不同制造商提供的特定的消息传递库之间的许多差别其实只是句法上的，但是，将消息传递程序从一个库移植到另一个库中时，对一些重大的语义差别需要作很大修改。

消息传递接口（通称为MPI）就是用来解决这个问题。MPI定义消息传递的标准库，这些库可以使用C或Fortran来开发可移植的消息传递程序。MPI标准定义一组核心的库例行程序的语法及语义，它们对编写消息传递程序非常有用。MPI由许多来自学术机构和企业的研究人

员开发，并得到几乎所有的硬件制造商的支持。在几乎所有的商用并行计算机上都有MPI的硬件实现。

MPI库中包含超过125个例行程序，但关键的例行程序要少得多。事实上，只需要使用表6-1中列出的6个例行程序，就能写出全功能的消息传递程序。这些例行程序分别用来初始化及终止MPI库，从并行计算环境中获取信息，以及发送和接收消息。

表6-1 MPI例行程序的最小集

MPI_Init	初始化MPI
MPI_Finalize	终止MPI
MPI_Comm_size	确定进程的数目
MPI_Comm_rank	确定调用的进程的标号
MPI_Send	发送一条消息
MPI_Recv	接收一条消息

本节将讲述这些例行程序，以及用MPI编写正确和有效的消息传递程序的某些重要的基本概念。 241

6.3.1 启动和终止MPI库

对MPI_Init的调用要优先于对其他MPI例行程序的调用。它的作用是初始化MPI环境。在程序的执行过程中对MPI_Init调用超过一次就会出错。计算结束时调用MPI_Finalize，它将执行多个收尾任务以终止MPI环境。在调用MPI_Finalize后，不能再有其他的MPI调用，甚至包括MPI_Init。MPI_Init和MPI_Finalize必须被所有的进程调用，否则MPI的状态会成为不确定的。在C语言中，对这两个例行程序的正确调用序列如下：

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

MPI_Init中的参数argc和argv是C程序中的命令行参数。MPI的实现在返回到程序前，将从argv数组中移走应在实现中处理的所有命令行参数，并相应地减小argc。因此，对命令行的处理只有在调用MPI_Init后才能进行。在MPI_Init和MPI_Finalize成功执行后返回MPI_SUCCESS；否则返回一个由实现定义的错误代码。

对这两个函数的绑定和调用顺序表明MPI遵循通常的命名规则和参数约定。所有的MPI例行程序、数据类型以及常量都加一个前缀“MPI_”。如果调用成功，则返回码是MPI_SUCCESS，在C语言中它和其他的MPI常量以及数据结构定义在“mpi.h”文件中。每个MPI程序必需包含这个头文件。

6.3.2 通信器

通信域（communication domain）是一个贯穿于MPI的关键概念。一个通信域是可以相互通信的一组进程的集合。有关通信域的信息存储在类型为MPI_Comm的变量中，这些变量称为通信器（communicator）。这些通信器作为参数用于所有的消息传送MPI例行程序中，并唯一标识参与消息传送操作的进程。注意，每一个进程都可以属于许多不同的（可能重叠的）通信域。 242

通信器用来定义能够相互通信的进程集合。这个进程集合构成一个通信域。通常，所有

的进程都需要相互通信。因此，MPI定义一个默认的通信器MPI_COMM_WORLD，它包含所有参与并行执行的进程。然而，很多时候人们只想在几组（可能重叠的）进程间通信。通过对每组进程使用不同的通信器，就能杜绝发送到某一组的消息干扰发送到其他组的消息。在本章的后面，将讨论如何创建并使用这样的通信器。下面先来讲述把MPI_COMM_WORLD作为通信器参数用在所有需要通信器的所有MPI函数中。

6.3.3 获取信息

MPI_Comm_size和MPI_Comm_rank函数分别用来确定进程的数目以及调用进程的标号。这两个例行程序的调用序列如下:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

函数MPI_Comm_size在变量size中返回属于通信器comm的进程数目。因此，如果每个处理器有一个进程，调用MPI_Comm_size (MPI_COMM_WORLD, &size) 就会在size中返回程序用到的处理器的数目。属于通信器的每个进程都由它的等级（rank）唯一确定。进程的等级是一个整数，其值从0到通信器的规模减1。进程可以用MPI_Comm_rank函数确定其在通信器中的等级，该函数有两个参数，即通信器和一个整型变量rank。返回时，变量rank存储进程的等级。注意调用这两个函数的每个进程都必须属于所提供的通信器，否则将会出错。

例6.2 Hello World

下面我们可以用上面讲到的4个MPI函数来编写一个程序，从每一个处理器输出“Hello World”信息。

```
1  #include <mpi.h>
2
3  main(int argc, char *argv[])
4  {
5    int npes, myrank;
6
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &npes);
9    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10   printf("From process %d out of %d, Hello World!\n",
11           myrank, npes);
12   MPI_Finalize();
13 }
```

243

■

6.3.4 发送和接收消息

在MPI中，发送和接收消息的基本函数分别是MPI_Send和MPI_Recv。这两个例行程序的调用序列如下:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Status *status)
```

MPI_Send发送存储在由buf指向的缓冲区中的数据，缓冲区中包含由参数datatype指定类型的连续项目。缓冲区中项目的个数由参数count给出。表6-2中列出MPI数据类型与C语言中给出的数据类型之间的对应。注意对于C语言中所有的数据类型，都有一个等价的MPI

数据类型。但是，MPI中有两个C语言中没有的数据类型，它们是MPI_BYTE和MPI_PACKED。

MPI_BYTE对应于一个字节（8位），而MPI_PACKED与一个数据项集合对应，这些数据项由不相邻的数据打包构成。注意MPI_Send以及其他MPI例行程序中的消息的长度不用字节数而是以待发送的数据项的数目给出。以数据项的数目来指定长度可以使MPI代码具有可移植性，因为在不同的体系结构中，用来存储不同的数据类型的字节数也可能不同。

由MPI_Send发出的消息的目的地由dest和comm两个参数指定。dest参数是由通信器comm指定的通信域中的目标进程的等级。每一条消息都有一个整数值的tag与之关联，它用来区分不同类型的消息。消息-tag的取值范围从0到MPI定义的常量MPI_TAG_UB。虽然MPI_TAG_UB的值是在实现时指定，但最小为32 767。

表6-2 MPI和C语言中的数据类型间的对应

MPI数据类型	C 数据类型
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI_Recv接收到由一个进程发来的消息，该进程的等级由comm参数指定的通信域中的source给出。已发送消息的标志必需由tag参数指定。如果来自同一进程的许多消息的标志相同，那么这些消息中的任意一个被接收。MPI允许对source和tag使用通配符。如果source被设置为MPI_ANY_SOURCE，那么通信域中的任意进程都能成为消息源。同样，如 244
果tag被设置为MPI_ANY_TAG，那么具有任意标志的消息都被接收。接收到的消息存储在由buf指向的缓冲区中的连续单元。MPI_Recv中的count和datatype参数用来指定提供的缓冲区的长度。收到的消息应该等于或者小于此长度。这样就允许接收进程无需知道要发送的消息的确切大小。如果接收到的消息的长度大于提供的缓冲的长度，就会出现溢出错误，例行程序将返回一个MPI_ERR_TRUNCATE错误。

接收消息后，可以用status变量获取有关MPI_Recv操作的信息。在C语言中，status以MPI_Status数据结构存储。这个数据结构用含3个字段的结构实现如下：

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
};
```

MPI_SOURCE和MPI_TAG保存接收的消息源和标志。它们在source和tag参数设置为

MPI_ANY_SOURCE及MPI_ANY_TAG时尤其有用。MPI_ERROR保存接收的消息的出错码。

状态参数也能返回有关接收的消息的长度信息。此信息不能直接用status变量获得，但可以通过调用MPI_GET_Count函数获得。此函数的调用序列如下:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
        int *count)
```

245

MPI_Get_count中有3个参数，分别是由MPI_Recv返回的status、datatype中接收到的消息的类型，以及count变量中实际接收到的数据项的数目。

只有在请求的消息已收到并复制到缓冲区后，MPI_Recv才返回。也就是说，MPI_Recv是一个阻塞式接收操作。然而，MPI允许两种不同的MPI_Send实现。在第一种实现中，MPI_Send只有当相应的MPI_Recv已发出并且消息已发送到接收方时才返回。在第二种实现中，MPI_Send首先将消息复制到缓冲区中，然后就返回，无需等待相应的MPI_Recv执行。在这两种实现中，由MPI_Send中的buf参数指向的缓冲可以安全地重用和覆盖。无论采用哪一种MPI_Send实现，MPI程序都必须能正确运行。这样的程序称为安全（safe）程序。在编写安全的MPI程序时，忘掉这两种MPI_Send实现，只把它看作是有阻塞的发送操作，有时会对编程有帮助。

避免死锁 MPI_Send和MPI_Recv的语义对我们如何对发送和接收操作进行混合和搭配提出某些限制。例如，在下面的代码段中，进程0发送两条带有不同标志的消息到进程1，进程1以相反的顺序接收这两条消息。

```
1  int a[10], b[10], myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
5  if (myrank == 0) {
6    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
7    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
8  }
9  else if (myrank == 1) {
10   MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
11   MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
12 }
13 ...
```

如果MPI_Send用缓冲区方式实现，那么在有足够的缓冲区空间时，代码就能正确地运行。然而，如果MPI_Send在相应的接收发出前一直阻塞，那么发送和接收的两个进程都不能进行。这是因为进程0（即myrank= =0）要一直等到进程1发出相应的MPI_Recv（即标志为1的进程）；同时，进程1要一直等到进程0执行相应的MPI_Send（即标志为2的进程）。这段代码是不安全的，因为它的状态与实现有关。在任何MPI实现中，只有靠程序员来保证程序的正确性。此程序中的问题可以通过匹配发送和接收操作发出的顺序来修正。当进程发送消息给自己时，也会出现同样类型的死锁。虽然发送消息给自己是合法的，但它的状态与实现有关，因而必须避免。

246

当每个处理器需要循环地发送和接收消息时，不恰当地使用MPI_Send和MPI_Recv也会造成死锁。考虑下面的代码段，其中进程i发送一条消息给进程$i+1$（进程数目的模），并从进程$i-1$（进程数目的模）接收一条消息。

```
1  int a[10], b[10], npes, myrank;
2  MPI_Status status;
3  ...
```

```
4  MPI_Comm_size(MPI_COMM_WORLD, &npes);
5  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
7  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
8  ...
```

当MPI_Send用缓冲区方式实现时，上面的程序能正确运行，因为每次调用MPI_Send时，数据都会存入缓冲区，这样对MPI_Recv的调用就能被执行，由调用传送所需要的数据。然而，如果MPI_Send在相应的接收发出前一直阻塞，所有的进程都将进入一种无限的等待状态，等待邻近的进程发出MPI_Recv操作。注意即使只有两个进程时，死锁也会出现。因此，当成对的进程需要交换数据时，上面的方法将会导致不安全的程序。按下面的方式改写代码就能使上面的例子成为安全的:

```
1  int a[10], b[10], npes, myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_size(MPI_COMM_WORLD, &npes);
5  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6  if (myrank%2 == 1) {
7    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
8    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
9  }
10 else {
11   MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
12   MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
13 }
14 ...
```

新的方法将进程分成两组。一组中是奇数编号的进程，另一组中是偶数编号的进程。奇数编号的进程在执行发送操作后执行接收操作，而偶数编号的进程在执行接收操作后执行发送操作。因此，当奇数编号进程调用MPI_Send时，目标进程（具有偶数编号）在试图发送自己的消息前将调用MPI_Recv来接收发送来的消息。

同时发送和接收消息 在许多消息传递程序中，经常能看到以上的通信模式，因此，MPI提供MPI_Sendrecv函数，它既发送又接收消息。 247

在MPI_Send和MPI_Recv中出现的循环死锁不会在MPI_Sendrecv中发生。你可以把MPI_Sendrecv想象成允许同时为发送和接收传送数据。MPI_Sendrecv的调用序列如下:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

MPI_Sendrecv中的参数实际上是MPI_Send和MPI_Recv中参数的结合。发送和接收缓冲区必需分开，消息源和目标可以相同，也可以不同。上面提到的使用MPI_Sendrecv的例子的安全版本如下:

```
1  int a[10], b[10], npes, myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_size(MPI_COMM_WORLD, &npes);
5  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6  MPI_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
7               b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
8               MPI_COMM_WORLD, &status);
9  ...
```

在许多程序中，因为MPI_Sendrecv要求发送和接收缓冲区分离，这可能使我们不得不用临时缓冲区。这样就使程序的内存需求量增加，并且总体运行时间也因额外的复制而增加。可以用MPI_Sendrecv_replace函数来解决此问题。此函数执行阻塞式发送和接收操作，但对发送和接收操作只使用单一的缓冲区。也就是说，接收到的数据替换发送出缓冲的数据。这个函数的调用序列如下：

```
int MPI_Sendrecv_replace(void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

注意，发送和接收操作必须传送同一种数据类型的数据。

6.3.5 实例：奇偶排序

我们将用前面几小节讲到的MPI函数来写一个完整的消息传递程序，该程序用奇偶排序算法对一个数字列表排序。参考9.3.1节可知，奇偶排序算法在总共p个阶段中对n个元素用p个进程排序。在每个阶段中，奇数或偶数的进程都要和它们的右边邻居进行一次比较-分裂操作。
248 程序6-1列出进行并行奇偶排序的MPI程序。为了简化程序的表示，这个程序假设n能被p整除。

程序6-1 奇偶排序

```
 1  #include <stdlib.h>
 2  #include <mpi.h> /* Include MPI's header file */
 3
 4  main(int argc, char *argv[])
 5  {
 6    int n;          /* The total number of elements to be sorted */
 7    int npes;       /* The total number of processes */
 8    int myrank;     /* The rank of the calling process */
 9    int nlocal;     /* The local number of elements, and the array that stores them */
10    int *elmnts;    /* The array that stores the local elements */
11    int *relmnts;   /* The array that stores the received elements */
12    int oddrank;    /* The rank of the process during odd-phase communication */
13    int evenrank;   /* The rank of the process during even-phase communication */
14    int *wspace;    /* Working space during the compare-split operation */
15    int i;
16    MPI_Status status;
17
18    /* Initialise MPI and get system information */
19    MPI_Init(&argc, &argv);
20    MPI_Comm_size(MPI_COMM_WORLD, &npes);
21    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
22
23    n = atoi(argv[1]);
24    nlocal = n/npes; /* Compute the number of elements to be stored locally. */
25
26    /* Allocate memory for the various arrays */
27    elmnts  = (int *)malloc(nlocal*sizeof(int));
28    relmnts = (int *)malloc(nlocal*sizeof(int));
29    wspace  = (int *)malloc(nlocal*sizeof(int));
30
31    /* Fill-in the elmnts array with random elements */
32    srandom(myrank);
33    for (i=0; i<nlocal; i++)
34      elmnts[i] = random();
35
36    /* Sort the local elements using the built-in quicksort routine */
37    qsort(elmnts, nlocal, sizeof(int), IncOrder);
```

```
38
39    /* Determine the rank of the processors that myrank needs to communicate during the */
40    /* odd and even phases of the algorithm */
41    if (myrank%2 == 0) {
42      oddrank  = myrank-1;
43      evenrank = myrank+1;
44    }
45    else {
46      oddrank  = myrank+1;
47      evenrank = myrank-1;
48    }
49
50    /* Set the ranks of the processors at the end of the linear */
51    if (oddrank == -1 || oddrank == npes)
52      oddrank = MPI_PROC_NULL;
53    if (evenrank == -1 || evenrank == npes)
54      evenrank = MPI_PROC_NULL;
55
56    /* Get into the main loop of the odd-even sorting algorithm */
57    for (i=0; i<npes-1; i++) {
58      if (i%2 == 1) /* Odd phase */
59        MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
60            nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
61      else /* Even phase */
62        MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
63            nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);
64
65      CompareSplit(nlocal, elmnts, relmnts, wspace,
66                   myrank < status.MPI_SOURCE);
67    }
68
69    free(elmnts); free(relmnts); free(wspace);
70    MPI_Finalize();
71  }
72
73  /* This is the CompareSplit function */
74  CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace,
75               int keepsmall)
76  {
77    int i, j, k;
78
79    for (i=0; i<nlocal; i++)
80      wspace[i] = elmnts[i];  /* Copy the elmnts array into the wspace array */
81
82    if (keepsmall) { /* Keep the nlocal smaller elements */
83      for (i=j=k=0; k<nlocal; k++) {
84        if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
85          elmnts[k] = wspace[i++];
86        else
87          elmnts[k] = relmnts[j++];
88      }
89    }
90    else { /* Keep the nlocal larger elements */
91      for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
92        if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
93          elmnts[k] = wspace[i--];
94        else
95          elmnts[k] = relmnts[j--];
96      }
97    }
98  }
99
100 /* The IncOrder function that is called by qsort is defined as follows */
101 int IncOrder(const void *e1, const void *e2)
102 {
103   return (*((int *)e1) - *((int *)e2));
104 }
```

249

6.4 拓扑结构与嵌入

MPI将进程看成按一维拓扑结构组织并用线性顺序对进程编号。然而，在许多并行程序中，进程实际上是按高维（如二维或三维）拓扑结构组织的。在这样的程序中，交互进程的计算和设置都自然地依据它们在这种拓扑结构中的坐标来标识。例如，在进程按二维拓扑结构组
250 织的并行程序中，进程(i, j)可能需要发送消息到进程(k, l)或从(k, l)接收消息。为了用MPI实现这些程序，就有必要将每个MPI进程映射到高维拓扑结构中的一个进程。

许多这样的映射是可能的，图6-5展示从8个MPI进程到4×4二维结构的一些可能的映射。例如，对于图6-5a所示的映射，等级为*rank*的MPI进程与网格中的进程(*row*, *col*)相对应，使得*row* = *rank*/4，*col* = *rank*%4（其中“%”是C语言中的模操作符）。作为例子，等级为7的进程会映射到网格中的进程（1, 3）。

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

a）以行为主的映射

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

b）以列为主的映射

0	3	4	5
1	2	7	6
14	13	8	9
15	12	11	10

c）空间填充曲线映射

0	1	3	2
4	5	7	6
12	13	15	14
8	9	11	10

d）超立方体映射

图6-5 将多个进程映射到二维网格中的不同方法。a）和b）显示按行和列的进程映射，c）显示按空间填充的曲线（虚线）的映射，d）显示邻接的进程在超立方体中直接连接的映射

通常，映射的优良性由更高维结构中进程间的交互模式、物理处理器间的连通性以及从MPI进程到物理处理器间的映射决定。例如，考虑某一程序使用二维拓扑结构，每个进程需要沿着拓扑结构的x和y方向与邻近的进程通信。如果当前并行系统的处理器通过超立方体互连网络连接，那么按6-5d所示进行映射更好，因为网格中邻近的进程也是超立方体拓扑结构中邻近的处理器。

然而，这种由MPI用来对通信域中的进程分配等级的机制没有利用任何互连网络的信息，使得它不可能以智能的方式进行拓扑结构嵌入。而且，即使有了互连网络的信息，对于不同的互连网络，也需要指定不同的映射，这样就减弱了MPI与体系结构无关的优点。更好的解决方法是，让程序库自身计算最适合的从给定的拓扑结构到并行计算机的处理器的嵌入。这正好是MPI使之容易实现的方法。MPI中提供一组例行程序，它们能让程序员在不同的拓扑结构中安排进程，无需显式指定进程如何映射到处理器上。直接用MPI库找出最适合的映射可以减少发送和接收消息的成本。

6.4.1 创建和使用笛卡儿拓扑结构

251 在MPI例行程序中，允许按图的方式指定任意连通的虚拟进程拓扑结构。图中的每个节点都与一个进程对应，如果两个节点相互通信，则它们就是相连的。进程图可以用来指定任意需要的拓扑结构。但是，在消息传递程序中最常用的结构是一维、二维或更高维的网格，它们也称为笛卡儿拓扑结构（Cartesian topology）。因此，MPI提供一组特别的例行程序用来指定和处理这种类型的多维网格拓扑结构。

用于描述笛卡儿结构的MPI函数是MPI_Cart_create，它的调用序列如下：

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
        int *periods, int reorder, MPI_Comm *comm_cart)
```

这个函数获取属于通信器comm_old的一组进程，并创建一个虚拟进程结构。结构信息附属在由MPI_Cart_create创建的新通信器comm_cart上。任何后面的MPI例行程序，如果想利用这种新的笛卡儿结构，就必须使用comm_cart作为通信器参数。注意所有属于通信器comm_old的进程都必须调用这个函数。拓扑结构的形状与属性由参数ndims、dims以及periods指定。参数ndims指定拓扑结构的维数。数组dims指定拓扑结构中每一维的大小。这个数组中的第i个元素存储拓扑结构中第i维的大小。数组periods用于指定拓扑结构中是否有环绕连接。特别，如果periods[i]为真（在C语言中为非0），那么拓扑结构沿i维有环绕连接，否则没有。最后，参数reorder用于确定新组（即通信器）中的进程是否需要进行重新排序。如果reorder为假，那么新组中每个进程的等级与旧组中的等级相同；如果reorder为真，且由此导致更好地将虚拟拓扑结构嵌入到并行计算机中，那么MPI_Cart_create可以对进程重新排序。如果由dims数组指定的进程的总数小于由comm_old指定的通信器中的进程总数，那么某些进程将不是笛卡儿结构的组成部分。对于这些进程来说，comm_cart的值将被设置为MPI_COMM_NULL（一个由MPI定义的常量）。若由dims指定的进程的总数大于通信器comm_old中的进程总数，则导致出错。

进程命名 使用笛卡儿拓扑结构后，每个进程就能更好地用结构中的坐标标识。但是，我们描述的所有用于发送和接收消息的MPI函数，都要求用进程的等级指定每条消息的源和目标。所以，MPI提供两个函数MPI_Cart_rank和MPI_Cart_coord，用来分别进行从坐标到等级以及从等级到坐标的转换。这两个例行程序的调用序列如下：

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims,
        int *coords)
```

252

MPI_Cart_rank以数组coords中的进程的坐标作为参数，并用rank返回进程的等级。MPI_Cart_coord以进程的等级作为参数，并返回数组coords中的笛卡儿坐标及长度maxdims。要注意，maxdims至少要等于由通信器comm_cart指定的笛卡儿结构的维数。

通常，在笛卡儿结构中的进程之间的通信都是沿着某一维交换数据。MPI提供函数MPI_Cart_shift，用来计算在数据交换操作中源进程和目标进程的等级。这个函数的调用序列如下：

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir, int s_step,
        int *rank_source, int *rank_dest)
```

数据交换的方向由参数dir指定，并且是拓扑结构的一个维。交换数据的大小由参数s_step指定。计算出的等级在rank_source和rank_dest中返回。如果笛卡儿结构是用环绕连接创建的（即periods[dir]项设置为真），那么数据环绕交换；否则，对于那些在拓扑结构外的进程，对其rank_source和rank_dest返回值MPI_PROC_NULL。

6.4.2 实例：Cannon的矩阵与矩阵相乘

为了展示如何使用不同的拓扑结构函数，我们用这些函数实现将矩阵A和B相乘的Cannon算法，算法在8.2.2节中说明。Cannon算法把进程看作按虚拟的二维正方形数组排列，它用这

个数组以块的形式来分配矩阵A、B以及结果矩阵C。也就是说，如果每个矩阵的大小为$n \times n$，进程的总数为p，那么每个矩阵都被分成大小为$n/\sqrt{p} \times n/\sqrt{p}$（假设$p$是完全平方数）的方块。对网格中的进程$P_{i,j}$分配每个矩阵的$A_{i,j}$、$B_{i,j}$和$C_{i,j}$块。经过初始的数据对齐后，算法将执行$\sqrt{p}$步。在每一步中，每个进程都对本地可获得的矩阵$A$和$B$中的块相乘，然后把$A$中的块送给左侧的进程，$B$中的块送给上面的进程。

程序6-2显示实现Cannon算法的MPI函数。矩阵的维数用参数n提供。参数a、b、c分别指向矩阵A、B、C中在本地存储的部分。数组的大小为$n/\sqrt{p} \times n/\sqrt{p}$，其中$p$是进程的数目。这个例行程序假设$p$是完全平方数，且$n$是$\sqrt{p}$的整数倍数。参数comm存储描述进程的通信器，这些进程调用MatrixMatrixMultiply函数。本章中剩余的程序以函数的形式提供，而不
253 用完全独立的程序形式。

程序6-2 用MPI拓扑实现Cannon的矩阵与矩阵相乘

```
1  MatrixMatrixMultiply(int n, double *a, double *b, double *c,
2                       MPI_Comm comm)
3  {
4    int i;
5    int nlocal;
6    int npes, dims[2], periods[2];
7    int myrank, my2drank, mycoords[2];
8    int uprank, downrank, leftrank, rightrank, coords[2];
9    int shiftsource, shiftdest;
10   MPI_Status status;
11   MPI_Comm comm_2d;
12
13   /* Get the communicator related information */
14   MPI_Comm_size(comm, &npes);
15   MPI_Comm_rank(comm, &myrank);
16
17   /* Set up the Cartesian topology */
18   dims[0] = dims[1] = sqrt(npes);
19
20   /* Set the periods for wraparound connections */
21   periods[0] = periods[1] = 1;
22
23   /* Create the Cartesian topology, with rank reordering */
24   MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
25
26   /* Get the rank and coordinates with respect to the new topology */
27   MPI_Comm_rank(comm_2d, &my2drank);
28   MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
29
30   /* Compute ranks of the up and left shifts */
31   MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
32   MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
33
34   /* Determine the dimension of the local matrix block */
35   nlocal = n/dims[0];
36
37   /* Perform the initial matrix alignment. First for A and then for B */
38   MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
39   MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE, shiftdest,
40       1, shiftsource, 1, comm_2d, &status);
41
42   MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
43   MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
44       shiftdest, 1, shiftsource, 1, comm_2d, &status);
45
46   /* Get into the main computation loop */
47   for (i=0; i<dims[0]; i++) {
48     MatrixMultiply(nlocal, a, b, c); /* c = c + a*b */
```

```
49
50      /* Shift matrix a left by one */
51      MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
52          leftrank, 1, rightrank, 1, comm_2d, &status);
53
54      /* Shift matrix b up by one */
55      MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
56          uprank, 1, downrank, 1, comm_2d, &status);
57    }
58
59    /* Restore the original distribution of a and b */
60    MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
61    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
62        shiftdest, 1, shiftsource, 1, comm_2d, &status);
63
64    MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
65    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
66        shiftdest, 1, shiftsource, 1, comm_2d, &status);
67
68    MPI_Comm_free(&comm_2d);  /* Free up communicator */
69  }
70
71  /* This function performs a serial matrix-matrix multiplication c = a*b */
72  MatrixMultiply(int n, double *a, double *b, double *c)
73  {
74    int i, j, k;
75
76    for (i=0; i<n; i++)
77      for (j=0; j<n; j++)
78        for (k=0; k<n; k++)
79          c[i*n+j] += a[i*n+k]*b[k*n+j];
80  }
```

254

6.5 计算与通信重叠

到目前为止，每当进行点对点通信时，我们开发的MPI程序都使用阻塞式发送和接收操作。前面已经讲过，阻塞式发送操作在消息从发送缓冲复制出之前一直阻塞着（无论是在源进程的系统缓冲还是发送到目标进程）。同样，阻塞式接收操作只有在消息接收完毕并复制到接收缓冲后才返回。例如，考虑程序6-2中Cannon的矩阵相乘算法。在主计算循环（47行至57行）的每一次迭代中，首先要计算存储在a和b中的子矩阵的矩阵相乘，然后用MPI_Sendrecv_replace交换a中的块和b中的块，这个操作在指定的矩阵块被相应的进程发送和接收前一直阻塞。在每一次迭代中，每个进程花费时间$O(n^3/p^{1.5})$进行矩阵相乘，花费时间$O(n^2/p)$进行矩阵A和B的块交换。现在，由于矩阵A和B中的块在处理器之间交换时并不改变，可以让矩阵和矩阵相乘的运算与这些块的传送重叠。许多新近的分布式内存并行计算机都有专门的通信控制器，能够执行消息传送操作而不需要CPU的干预。

无阻塞式通信操作

为了让计算与通信重叠，MPI提供一对执行无阻塞式发送和接收操作的函数。它们分别是MPI_Isend和MPI_Irecv。MPI_Isend开始一个发送操作，但不等它完成，也就是说，在数据复制出缓冲区前就返回。同样，MPI_Irecv开始一个接收操作，但在数据接收完毕并且复制到缓冲区前就返回。依靠相应的硬件支持，在两个函数返回前，消息的传送和接收就能和程序中的计算操作同时进行。

255

然而，在程序的后面部分，开始一个无阻塞式发送或接收操作的进程必须确保操作在计

算前已经结束。这是因为，开始无阻塞式发送操作的进程可能需要覆盖存储待发送数据的缓冲区，或者开始无阻塞式接收操作的进程可能需要使用它请求的数据。为了检查无阻塞式发送和接收操作是否完成，MPI提供一对函数MPI_Test和MPI_Wait。前者检查无阻塞式操作是否已完成，后者等待（即阻塞）直到无阻塞式操作真正结束。

MPI_Isend和MPI_Irecv的调用序列如下：

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Request *request)
```

注意这两个函数与相应的阻塞式发送和接收函数有类似的参数。主要区别是这两个函数中多了参数request。MPI_Isend和MPI_Irecv函数分配一个请求(request)对象并返回一个指向request变量的指针。这个请求对象作为MPI_Test和MPI_Wait函数中的参数，用来确定那些我们需要查询其状态或等待其完成的操作。

注意，与阻塞式接收操作类似，MPI_Irecv函数不用status参数，但是与接收操作有关的状态信息通过MPI_Test和MPI_Wait函数返回。

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

MPI_Test测试由request确定的无阻塞式发送或接收操作是否已完成。如果已完成，则返回flag={true}（在C语言中的非0值），否则返回{false}（C语言中的0值）。在无阻塞式操作完成后，由变量request指向的请求对象被解除分配，request被设置为MPI_REQUEST_NULL。同时status对象被设置为包含与操作有关的信息。如果操作未完成，request就不被修改，status对象中的值是未定义的。MPI_Wait函数在由request确定的无阻塞式操作完成前一直阻塞。在这种情况下，它解除分配request对象，将其设置为
256 MPI_REQUEST_NULL，并在status对象中返回有关已完成操作的信息。

对于程序员想要显式解除分配一个请求对象的情况，MPI提供下述函数：

```
int MPI_Request_free(MPI_Request *request)
```

注意，请求对象的解除分配对相关的无阻塞式发送和接收操作没有任何影响。也就是说，如果这些操作未完成，它们会继续进行直至完成。因此，一定要小心地对请求对象进行显式解除分配，因为没有请求对象就不能测试无阻塞式操作是否已完成。

无阻塞式通信操作可以和相应的阻塞式操作相匹配。例如，进程可以用无阻塞式发送操作发送消息，此消息可以由其他进程用阻塞式接收操作接收。

避免死锁 使用无阻塞式通信操作可以除去绝大多数与阻塞式通信操作相关的死锁。例如，我们在6.3节讨论过，下面的代码是不安全的。

```
1  int a[10], b[10], myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
5  if (myrank == 0) {
6    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
7    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
8  }
9  else if (myrank == 1) {
10   MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
```

```
11    MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
12  }
13  ...
```

然而，如果将发送和接收操作中的任一个用它们对应的无阻塞式操作代替，代码就将是安全的，并可以在任何MPI实现上正确运行。

```
1   int a[10], b[10], myrank;
2   MPI_Status status;
3   MPI_Request requests[2];
4   ...
5   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6   if (myrank == 0) {
7     MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
8     MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
9   }
10  else if (myrank == 1) {
11    MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
12    MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
13  }
14  ...
```

257

这个例子也说明，由任意进程启动的无阻塞式操作，可以按相应消息的传送或接收所确定的任意顺序完成。例如，第二个接收操作将在第一个之前完成。

实例：Cannon的矩阵与矩阵相乘（使用无阻塞式操作）

程序6-3显示用无阻塞式发送和接收操作实现的Cannon算法的MPI程序，其中的参数与程序6-2中的相同。

程序6-3 无阻塞式Cannon的矩阵与矩阵相乘

```
1  MatrixMatrixMultiply_NonBlocking(int n, double *a, double *b,
2                                   double *c, MPI_Comm comm)
3  {
4    int i, j, nlocal;
5    double *a_buffers[2], *b_buffers[2];
6    int npes, dims[2], periods[2];
7    int myrank, my2drank, mycoords[2];
8    int uprank, downrank, leftrank, rightrank, coords[2];
9    int shiftsource, shiftdest;
10   MPI_Status status;
11   MPI_Comm comm_2d;
12   MPI_Request reqs[4];
13
14   /* Get the communicator related information */
15   MPI_Comm_size(comm, &npes);
16   MPI_Comm_rank(comm, &myrank);
17
18   /* Set up the Cartesian topology */
19   dims[0] = dims[1] = sqrt(npes);
20
21   /* Set the periods for wraparound connections */
22   periods[0] = periods[1] = 1;
23
24   /* Create the Cartesian topology, with rank reordering */
25   MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
26
27   /* Get the rank and coordinates with respect to the new topology */
28   MPI_Comm_rank(comm_2d, &my2drank);
29   MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
30
31   /* Compute ranks of the up and left shifts */
32   MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
33   MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
```

```
34
35    /* Determine the dimension of the local matrix block */
36    nlocal = n/dims[0];
37
38    /* Setup the a_buffers and b_buffers arrays */
39    a_buffers[0] = a;
40    a_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
41    b_buffers[0] = b;
42    b_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
43
258 44    /* Perform the initial matrix alignment. First for A and then for B */
45    MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
46    MPI_Sendrecv_replace(a_buffers[0], nlocal*nlocal, MPI_DOUBLE,
47        shiftdest, 1, shiftsource, 1, comm_2d, &status);
48
49    MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
50    MPI_Sendrecv_replace(b_buffers[0], nlocal*nlocal, MPI_DOUBLE,
51        shiftdest, 1, shiftsource, 1, comm_2d, &status);
52
53    /* Get into the main computation loop */
54    for (i=0; i<dims[0]; i++) {
55      MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
56          leftrank, 1, comm_2d, &reqs[0]);
57      MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
58          uprank, 1, comm_2d, &reqs[1]);
59      MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
60          rightrank, 1, comm_2d, &reqs[2]);
61      MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
62          downrank, 1, comm_2d, &reqs[3]);
63
64      /* c = c + a*b */
65      MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);
66
67      for (j=0; j<4; j++)
68        MPI_Wait(&reqs[j], &status);
69    }
70
71    /* Restore the original distribution of a and b */
72    MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
73    MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
74        shiftdest, 1, shiftsource, 1, comm_2d, &status);
75
76    MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
77    MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
78        shiftdest, 1, shiftsource, 1, comm_2d, &status);
79
80    MPI_Comm_free(&comm_2d);  /* Free up communicator */
81
82    free(a_buffers[1]);
83    free(b_buffers[1]);
84  }
```

阻塞式程序（程序6-2）与这个无阻塞式程序之间有两个主要区别。第一个区别是，无阻塞式程序需要使用另两个数组a_buffers和b_buffers，它们作为块A和块B的缓冲区，在执行涉及前面块的计算进行时被接收。第二个区别是，在主计算循环，无阻塞式程序首先启动无阻塞式发送操作，将本地存储的块A和块B发送到网格左边和上边的进程，然后启动无阻塞式接收操作，接收下一次迭代来的网格右边和下边进程的块。在启动这4个无阻塞式操作后，继续进行当前存储块的矩阵与矩阵相乘。最后，在进行下一次迭代前，使用MPI_Wait等待发送和接收操作完成。

注意，为了让计算与通信重叠，我们必需使用两个辅助数组——一个用于A，一个用于B。
259 这是为了保证到来的消息不覆盖在计算中使用的块A和块B，计算与数据传送是并发进行的。

这样，性能的提高（通过计算与通信重叠）以增加内存需求为代价。这是消息传递程序中经常作出的一种折中，因为对于低耦合分布式内存的并行计算机而言，通信开销非常大。

6.6 聚合的通信和计算操作

MPI提供了大量的函数，用来进行许多常用的聚合通信操作。特别，MPI支持第4章讲到的绝大多数基本通信操作。所有MPI提供的聚合通信函数以通信器作为参数，通信器定义参与聚合操作的一组进程。属于通信器的所有进程都参与到操作中，并且都必需调用聚合通信函数。虽然聚合通信操作不像障碍那样工作（即处理器可能跳过对聚合通信操作的调用，即使在其他的进程到达它之前），但它像下述意义下的虚拟同步：即使在聚合调用前或者聚合调用后进行了一次全局同步操作，并行程序也应该正确运行。由于操作是虚拟同步的，操作不需要标志。在一些聚合函数中，要求数据从单个进程（源进程）发送，或者由单个进程（目标进程）接收。在这些函数中，源进程或目标进程是对例行程序提供的参数之一。所有组内（即通信器）的进程必须指定相同的源进程或目标进程。对于绝大多数聚合通信操作，MPI提供两个不同的变体，一个用来在每个进程间传送同样大小的数据，另一个传送的数据大小可以不同。

6.6.1 障碍

在MPI中，障碍同步操作用MPI_Barrier函数实现。

```
int MPI_Barrier(MPI_Comm comm)
```

MPI_Barrier函数中只有通信器一个参数，它定义被同步的进程组。对MPI_Barrier的调用只有在组中的所有进程调用了这个函数后才返回。

6.6.2 广播

4.1节中讲过的一对多广播操作在MPI中用MPI_Bcast函数实现。 260

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
        int source, MPI_Comm comm)
```

MPI_Bcast将存储在进程source的缓冲区buf中的数据发送到组中所有其他的进程。每个进程接收到的数据都存储在缓冲区buf中。被广播的数据中，类型为datatype的数据有count项。由source进程发送的数据量必需等于每个进程要接收的数据量，也就是说，在所有的进程中，count和datatype字段必需匹配。

6.6.3 归约

4.1节中讲过的多对一归约操作在MPI函数中用MPI_Reduce函数实现。

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int target,
        MPI_Comm comm)
```

MPI_Reduce函数使用由op指定的操作符将组内每个进程中存储于缓冲区sendbuf的元素组合起来，并用等级为target的进程中的缓冲区recvbuf返回组合后的值。sendbuf和recvbuf中类型为datatype的数据项数目都必需为count。注意，所有的进程都必需提供

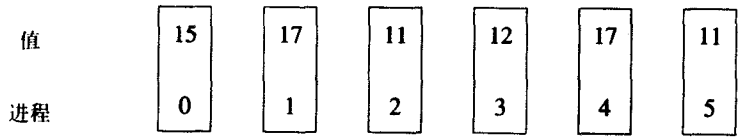
一个recvbuf数组，即使这些进程不是归约操作的目标（target）。当count大于1时，组合操作按序列中的每个数据项的逐个进行。所有的进程都必需以同样的count、datatype、op、target和comm调用MPI_Reduce。

261 MPI提供一系列预定义的操作，可以用于对存储于sendbuf中的元素进行组合。MPI也允许程序员自己定义操作，但本书不讨论这一点。表6-3列出了预定义的操作。例如，为了计算存储在sendbuf中的元素的最大值，op参数的值就必需为MPI_MAX。并非表中所有的操作都能应用于MPI支持的所有可能的数据类型。例如，按位或操作（即op = MPI_BOR）对实型的数据就没有定义，如MPI_FLOAT和MPI_REAL。表6-3中的最后一列显示可以用于每个操作的不同数据类型。

表6-3 预定义的归约操作

操 作	含 义	数据类型
MPI_MAX	最大值	C整型及浮点型
MPI_MIN	最小值	C整型及浮点型
MPI_SUM	求和	C整型及浮点型
MPI_PROD	求积	C整型及浮点型
MPI_LAND	逻辑与	C整型
MPI_BAND	按位与	C整型及字节型
MPI_LOR	逻辑或	C整型
MPI_BOR	按位或	C整型及字节型
MPI_LXOR	逻辑异或	C整型
MPI_BXOR	按位异或	C整型及字节型
MPI_MAXLOC	最大-最小值单元	数据对
MPI_MINLOC	最小-最小值单元	数据对

MPI_MAXLOC操作对数值对(v_i, l_i)进行组合，并返回数值对(v, l)，其中v是所有v_i中的最大值，l是当$v = v_i$时所有l_i中的最小值。同样，MPI_MINLOC操作对数值对进行组合，并返回数值(v, l)对，其中v是所有v_i中的最小值，l是当$v = v_i$时所有l_i中的最小值。MPI_MAXLOC和MPI_MINLOC可以用来计算位于不同进程中的一系列数的最大值或最小值，并求出存储最大值或最小值的第一个进程，如图6-6所示。由于MPI_MAXLOC和MPI_MINLOC需要成对的数据，MPI又定义了一组新的数据类型，如表6-4所示。在C语言中，这些数据类型是用包含相应类型的结构实现的。




```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

图6-6 一个使用MPI_MINLOC和MPI_MAXLOC操作符的例子

如果所有的进程都需要归约操作的结果，则可使用MPI提供的MPI_Allreduce操作。这
262 个函数提供4.3节讲述的全归约操作的功能。

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

注意，由于所有的进程都接收归约操作的结果，此函数中不含target参数。

表6-4 用于MPI_MAXLOC和MPI_MINLOC归约操作的MPI数据对数据类型

MPI数据类型	C数据类型
MPI_2INT	int对
MPI_SHORT_INT	short和int
MPI_LONG_INT	long和int
MPI_LONG_DOUBLE_INT	long double和int
MPI_FLOAT_INT	float和int
MPI_DOUBLE_INT	double和int

6.6.4 前缀

4.3节讲述的前缀和操作在MPI中用MPI_Scan函数实现。

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

MPI_Scan对存储在每个进程的缓冲区sendbuf中的数据进行前缀归约，并在缓冲区revcbuf中返回结果。在归约操作的最后，等级为*i*的进程的接收缓冲区，将存储等级由0到*i*（包括*i*）的进程的发送缓冲区中的归约结果。MPI_Scan支持的操作类型（即op）以及对函数中不同变量的限制，都与MPI_Reduce归约操作相同。

6.6.5 收集

4.4节讲述的收集操作在MPI中用MPI_Gather函数实现。

```
int MPI_Gather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int target, MPI_Comm comm)
```

每个进程（包括target进程）将存储在数组sendbuf中的数据发送给target进程。结果，如果*p*是通信comm中处理器的数目，则目标进程将接收的缓冲区总数为*p*。数据按等级顺序存储在目标进程的recvbuf数组中。就是说，来自等级为*i*的进程的数据存储在recvbuf中，起始单元为*i* * sendcount（假设数组recvbuf与recvdatatype的类型相同）。

每个进程发送的数据必需具有相同的类型和大小，即调用MPI_Gather时，参数sendcount和senddatatype必需在每个进程中具有同样的值。只有目标进程需要知道有关接收缓存的信息，如接收缓存的长度及类型，而其他所有进程都忽略这些信息。参数recvcount指定每个进程接收的元素数目而不是接收的元素总数目。所以，recvcount必需和sendcount相同，这两者的类型也必需匹配。

MPI还提供MPI_Allgather函数，它为所有的进程收集数据而不是仅给目标进程。 263

```
int MPI_Allgather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

其中各个参数的含义与函数MPI_Gather中的相似；但是，每个进程都必需提供一个recvbuf数组来存储收集的数据。

上面提到的收集操作中，每个进程发送的数组大小都相等，此外，MPI还提供了数组大小可以不等的收集操作，MPI称这种操作为向量（vector）变体。MPI_Gather和MPI_Allgather操作的向量变体分别由函数MPI_Gatherv和MPI_Allgatherv提供。

```
int MPI_Gatherv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int *recvcounts, int *displs,
        MPI_Datatype recvdatatype, int target, MPI_Comm comm)

int MPI_Allgatherv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int *recvcounts, int *displs, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```

将recvcount参数用数组recvcounts代替，这两个函数就能使每个进程发送不同数量的数据元素。进程*i*发送的数据量等于recvcounts[i]。注意Recvcounts与通信器comm的大小相等。数组参数displs的大小也和它们相等。displs用来确定由每个进程发送出的数据存储于revcbuf中的位置。特别，由进程*i*发送出的数据存储于recvbuf中起始单元为displs[i]的地方。注意与非向量变体相反，对于不同的进程，参数sendcount也可以不同。

6.6.6 散发

4.4节讲到的散发操作在MPI中用MPI_Scatter函数实现。

```
int MPI_Scatter(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int source, MPI_Comm comm)
```

源进程将发送缓冲sendbuf中的不同部分发送给每个进程，也包括它自身。接收到的数据存储于recvbuf中。进程*i*接收sendcount个类型为senddatatype连续的元素，这些元素从源进程中缓冲区sendbuf的*i* *sendcount单元开始（假设sendbuf与senddatatype的类型相同）。所有进程调用MPI_Scatter时，都必须使用同样值的sendcount、
264 senddatatype、recvcount、recvdatatype、source以及comm参数。再次注意，sendcount是发送到每一个进程的元素数目。

与收集操作类似，MPI提供散发操作的向量变体，称为MPI_Scatterv，它允许将不同数量的数据发送给不同的进程。

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int source, MPI_Comm comm)
```

从上面的函数可以看出，数组sendcounts代替了参数sendcount，该数组确定发送给每个进程的元素数目。特别地，target进程发送sendcounts[i]个元素给进程*i*。同样，数组displs用来确定这些元素从sendbuf的何处发送出。特别，如果sendbuf和senddatatype的类型相同，那么发送到进程*i*的数据起始于数组sendbuf的displs[i]处。

数组sendcounts和displs的大小与通信器中的进程数目相等。注意如果适当地设置displs数组，就能用MPI_Scatterv发送sendbuf中的重叠区域。

6.6.7 多对多

4.5节中讲过的多对多私自通信操作在MPI中用MPI_Alltoall函数实现。

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

每个进程发送数组sendbuf中的不同部分给其他每个进程，包括它自身。每个进程发送给进程i sendcount个类型为senddatatype连续的元素，这些元素从它的sendbuf数组中i *sendcount单元开始。接收到的数据存储于数组recvbuf中。每个进程从进程i接收recvcount个类型为recvdatatype的元素，并将这些元素存储在recvbuf数组中从i *sendcount开始的单元。所有进程调用MPI_Alltoall时，必须使用同样值的sendcount、senddatatype、recvcount、recvdatatype以及comm参数。注意，sendcount和recvcount是发送到每个进程以及从每个进程接收到的元素数目。

对于多对多私自通信操作，MPI也提供一个向量变体MPI_Alltoallv，它允许每个进程发送或接收不同数量的数据。

```
int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls
        MPI_Datatype senddatatype, void *recvbuf, int *recvcounts,
        int *rdispls, MPI_Datatype recvdatatype, MPI_Comm comm)
```

265

参数sendcounts用来指定发送到每个进程的元素数目，而参数sdispls用来指定这些元素在sendbuf中存储的位置。特别，每个进程发送给进程i sendcounts[i]个连续的元素，从数组sendbuf的sdispls[i]单元开始。参数recvcounts用来指定每个进程接收的元素数目，而参数rdispls用来指定这些元素存储在recvbuf中的位置。特别，每个进程从进程i接收recvcounts[i]个元素，这些元素存储在recvbuf中连续的单元，从rdispls[i]单元开始。所有进程调用MPI_Alltoallv时，必须使用同样值的senddatatype、recvdatatype以及comm参数。

6.6.8 实例：一维矩阵与向量相乘

我们第一个使用聚合通信的消息传递程序是将稠密$n \times n$矩阵A与向量b相乘，即$x = Ab$。如8.1节中的讨论，一种并行执行这种乘法的方法是让每个进程计算乘积向量x的不同部分。特别，p个进程中的每一个计算x中n/p个连续的元素。这个算法可通过将矩阵A按行分配用MPI实现，每个进程接收n/p个行，与进程计算的乘积向量x中的部分相对应。向量b也以与x类似的方式分配。

程序6-4显示按行分配矩阵A的MPI程序。矩阵的维数用参数n给出，参数a和参数b分别指向矩阵A和向量b中本地存储的部分，而参数x指向输出的矩阵与向量乘积的本地部分。这个程序假设n是处理器数目的整数倍。

266

计算x的另一种方法是将对x中每个元素进行点乘的任务并行化。也就是说，对于向量x中的每个元素x_i，所有进程都计算一部分，将部分的点乘结果相加得到最终结果。这个算法可通

过将矩阵A按列方式分配用MPI实现。每个进程得到矩阵A中n/p个连续的列，以及向量b中与这些列相对应的元素。此外，计算结束后我们将乘积向量x按与向量b类似的方式分配。程序6-5显示这种按列分配矩阵的MPI程序。

程序6-4 按行的矩阵与向量相乘

```
1  RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
2                          MPI_Comm comm)
3  {
4    int i, j;
5    int nlocal;          /* Number of locally stored rows of A */
6    double *fb;          /* Will point to a buffer that stores the entire vector b */
7    int npes, myrank;
8    MPI_Status status;
9
10   /* Get information about the communicator */
11   MPI_Comm_size(comm, &npes);
12   MPI_Comm_rank(comm, &myrank);
13
14   /* Allocate the memory that will store the entire vector b */
15   fb = (double *)malloc(n*sizeof(double));
16
17   nlocal = n/npes;
18
19   /* Gather the entire vector b on each processor using MPI's ALLGATHER operation */
20   MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
21       comm);
22
23   /* Perform the matrix-vector multiplication involving the locally stored submatrix */
24   for (i=0; i<nlocal; i++) {
25     x[i] = 0.0;
26     for (j=0; j<n; j++)
27       x[i] += a[i*n+j]*fb[j];
28   }
29
30   free(fb);
31 }
```

267

程序6-5 按列的矩阵与向量相乘

```
1  ColMatrixVectorMultiply(int n, double *a, double *b, double *x,
2                          MPI_Comm comm)
3  {
4    int i, j;
5    int nlocal;
6    double *px;
7    double *fx;
8    int npes, myrank;
9    MPI_Status status;
10
11   /* Get identity and size information from the communicator */
12   MPI_Comm_size(comm, &npes);
13   MPI_Comm_rank(comm, &myrank);
14
15   nlocal = n/npes;
16
17   /* Allocate memory for arrays storing intermediate results. */
18   px = (double *)malloc(n*sizeof(double));
19   fx = (double *)malloc(n*sizeof(double));
20
21   /* Compute the partial-dot products that correspond to the local columns of A. */
22   for (i=0; i<n; i++) {
23     px[i] = 0.0;
24     for (j=0; j<nlocal; j++)
25       px[i] += a[i*nlocal+j]*b[j];
```

```
26    }
27
28    /* Sum-up the results by performing an element-wise reduction operation */
29    MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);
30
31    /* Redistribute fx in a fashion similar to that of vector b */
32    MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0,
33        comm);
34
35    free(px); free(fx);
36  }
```

比较上面两个矩阵向量相乘的程序可以发现，按行的程序只需要进行一次MPI_Allgather操作，而按列的程序需要进行一次MPI_Reduce和MPI_Scatter操作。通常，按行序分配更好，因为它造成的通信开销较小（见习题6.6）。但是，在许多时候，程序不仅要计算Ax，还要计算A^Tx。在这种情况下，按行分配可用来计算Ax，但计算A^Tx则需要按列分配（对A的按行分配是A的转置A^T的按列分配）。使用按列分配的程序，要比先转置矩阵再用按行分配的程序简便得多。使用全聚集（all-gather）操作的对偶操作，就能使使用按列分配的程序与使用按行分配的程序运行得一样快（见习题6.7）。但是，MPI中没有提供这种对偶操作。

6.6.9 实例：单源最短路径

我们的第二个使用聚合通信操作的消息传递程序是计算图中从源顶点s到其他所有顶点的最短路径，程序使用10.3节中描述的Dijkstra的单源最短路径算法。程序6-6为这个程序的代码。

参数n存储图中的顶点总数，参数source存储计算单源最短路径的源顶点，参数wgt指向图的加权邻接矩阵的本地存储部分。参数lengths指向一个向量，该向量将用来存储从源顶点到本地存储顶点的最短路径。最后，参数comm是将被MPI例行程序使用的通信器。注意这个例行程序假设顶点数目是处理器数目的整数倍。 268

程序6-6 Dijkstra的单源最短路径

```
1  SingleSource(int n, int source, int *wgt, int *lengths, MPI_Comm comm)
2  {
3    int i, j;
4    int nlocal;  /* The number of vertices stored locally */
5    int *marker;  /* Used to mark the vertices belonging to Vo */
6    int firstvtx;  /* The index number of the first vertex that is stored locally */
7    int lastvtx;  /* The index number of the last vertex that is stored locally */
8    int u, udist;
9    int lminpair[2], gminpair[2];
10   int npes, myrank;
11   MPI_Status status;
12
13   MPI_Comm_size(comm, &npes);
14   MPI_Comm_rank(comm, &myrank);
15
16   nlocal   = n/npes;
17   firstvtx = myrank*nlocal;
18   lastvtx  = firstvtx+nlocal-1;
19
20   /* Set the initial distances from source to all the other vertices */
21   for (j=0; j<nlocal; j++)
22     lengths[j] = wgt[source*nlocal + j];
23
```

```
24    /* This array is used to indicate if the shortest part to a vertex has been found or not. */
25    /* if marker[v] is one, then the shortest path to v has been found, otherwise it has not. */
26    marker = (int *)malloc(nlocal*sizeof(int));
27    for (j=0; j<nlocal; j++)
28      marker[j] = 1;
29
30    /* The process that stores the source vertex, marks it as being seen */
31    if (source >= firstvtx && source <= lastvtx)
32      marker[source-firstvtx] = 0;
33
34    /* The main loop of Dijkstra's algorithm */
35    for (i=1; i<n; i++) {
36      /* Step 1: Find the local vertex that is at the smallest distance from source */
37      lminpair[0] = MAXINT; /* set it to an architecture dependent large number */
38      lminpair[1] = -1;
39      for (j=0; j<nlocal; j++) {
40        if (marker[j] && lengths[j] < lminpair[0]) {
41          lminpair[0] = lengths[j];
42          lminpair[1] = firstvtx+j;
43        }
44      }
45
46      /* Step 2: Compute the global minimum vertex, and insert it into Vc */
47      MPI_Allreduce(lminpair, gminpair, 1, MPI_2INT, MPI_MINLOC,
48          comm);
49      udist = gminpair[0];
50      u = gminpair[1];
51
52      /* The process that stores the minimum vertex, marks it as being seen */
53      if (u == lminpair[1])
54        marker[u-firstvtx] = 0;
55
56      /* Step 3: Update the distances given that u got inserted */
57      for (j=0; j<nlocal; j++) {
58        if (marker[j] && udist + wgt[u*nlocal+j] < lengths[j])
59          lengths[j] = udist + wgt[u*nlocal+j];
60      }
61    }
62
63    free(marker);
64  }
```

Dijkstra的并行单源最短路径算法的主计算循环要分三步执行。第一步，每个进程寻找V_o中本地存储的与源点距离最短的顶点；第二步，确定在所有进程中距离最短的顶点，该顶点
269 包含在V_c中；第三步，所有的进程更新它们的距离数组以反映出V_c中加入了新的顶点。

第一步的实现需要扫描V_o中本地存储的顶点，并确定具有更小*lengths*[*v*]值的顶点*v*。计算结果存储于数组*lminpair*中。特别地，*lminpair*[0]存储顶点的距离，*lminpair*[1]存储顶点自身。使用这种存储方案的原因在我们考虑下一步时就会清楚，那时必需计算所有顶点中离源点最短的顶点。通过对存储在*lminpair*[0]中的距离值进行一次最小归约，就能计算出最短距离。然而，除了求出最短距离外，还要求出处于最短距离的顶点。因此，最适合的归约操作是MPI_MINLOC，因为它不但返回最小值，还返回与最小值相对应的下标值。因为MPI_MINLOC，我们使用二元素数组*lminpair*来存储距离以及达到这个距离的顶点。此外，由于所有进程都需要归约操作的结果来进行第三步，我们使用MPI_Allreduce来执行归约操作。归约操作的结果在数组*gminpair*中返回。在每次迭代中，第三步和最后一步通过扫描属于V_o的本地顶点并更新这些顶点离源顶点的最短距离来实现。

避免负载不平衡 程序6-6分配*W*的*n*/*p*个连续列给每个处理器，在每次迭代中，程序使用MPI_MINLOC归约操作选出顶点*v*包含到V_c中。对于操作数(a, i)和(a, j)，回顾MPI_MINLOC

将返回下标较小的一个（因为两者的值相同）。因此，对于离源顶点同样近的顶点，这有利于编号较小的顶点。这可能导致负载不平衡，因为存储在低等级进程中的顶点会比存储在高等级进程中的顶点更快地包含到V_c中（特别当V_o中的多个顶点到源顶点的最小距离相同时）。因此集合V_o的大小在较高等级的进程中会较大，在整体运行时间中占支配地位。

改正这个问题的一个方法是，将W列按循环的方式分配。在这种分配方式中，进程i得到从顶点i开始的每个第p顶点。这个方案也分配n/p个顶点给每个进程，但是这些进程的下标跨越几乎整个图。因此，MPI_MINLOC有利于较小编号的顶点不会导致负载不平衡。

6.6.10 实例：样本排序

下面再看最后一个使用聚合通信的程序，将n个元素的序列A用将在9.5节讲述的样本排序算法进行排序。程序6-7是该程序的代码。

SampleSort函数将存储在每个进程中的元素序列作为输入，并返回指向一个数组的指针，该数组存储排序的序列以及序列中元素的数目。SampleSort函数的元素是整型的，且按增序排序。参数n指定待排序的元素的总数目；参数elmnts指定一个指针，指向存储这些 270
元素的本地部分的数组。返回时，参数nsorted将存储返回的排序的数组中的元素个数。这个例行程序假设n是进程数目的整数倍。

程序6-7 样本排序

```
 1  int *SampleSort(int n, int *elmnts, int *nsorted, MPI_Comm comm)
 2  {
 3    int i, j, nlocal, npes, myrank;
 4    int *sorted_elmnts, *splitters, *allpicks;
 5    int *scounts, *sdispls, *rcounts, *rdispls;
 6
 7    /* Get communicator-related information */
 8    MPI_Comm_size(comm, &npes);
 9    MPI_Comm_rank(comm, &myrank);
10
11    nlocal = n/npes;
12
13    /* Allocate memory for the arrays that will store the splitters */
14    splitters = (int *)malloc(npes*sizeof(int));
15    allpicks = (int *)malloc(npes*(npes-1)*sizeof(int));
16
17    /* Sort local array */
18    qsort(elmnts, nlocal, sizeof(int), IncOrder);
19
20    /* Select local npes-1 equally spaced elements */
21    for (i=1; i<npes; i++)
22      splitters[i-1] = elmnts[i*nlocal/npes];
23
24    /* Gather the samples in the processors */
25    MPI_Allgather(splitters, npes-1, MPI_INT, allpicks, npes-1,
26        MPI_INT, comm);
27
28    /* sort these samples */
29    qsort(allpicks, npes*(npes-1), sizeof(int), IncOrder);
30
31    /* Select splitters */
32    for (i=1; i<npes; i++)
33      splitters[i-1] = allpicks[i*npes];
34    splitters[npes-1] = MAXINT;
35
36    /* Compute the number of elements that belong to each bucket */
37    scounts = (int *)malloc(npes*sizeof(int));
38    for (i=0; i<npes; i++)
```

```
39      scounts[i] = 0;
40
41    for (j=i=0; i<nlocal; i++) {
42      if (elmnts[i] < splitters[j])
43        scounts[j]++;
44      else
45        scounts[++j]++;
46    }
47
48    /* Determine the starting location of each bucket's elements in the elmnts array */
49    sdispls = (int *)malloc(npes*sizeof(int));
50    sdispls[0] = 0;
51    for (i=1; i<npes; i++)
52      sdispls[i] = sdispls[i-1]+scounts[i-1];
53
271   54    /* Perform an all-to-all to inform the corresponding processes of the number of elements */
55    /* they are going to receive. This information is stored in rcounts array */
56    rcounts = (int *)malloc(npes*sizeof(int));
57    MPI_Alltoall(scounts, 1, MPI_INT, rcounts, 1, MPI_INT, comm);
58
59    /* Based on rcounts determine where in the local array the data from each processor */
60    /* will be stored. This array will store the received elements as well as the final */
61    /* sorted sequence */
62    rdispls = (int *)malloc(npes*sizeof(int));
63    rdispls[0] = 0;
64    for (i=1; i<npes; i++)
65      rdispls[i] = rdispls[i-1]+rcounts[i-1];
66
67    *nsorted = rdispls[npes-1]+rcounts[i-1];
68    sorted_elmnts = (int *)malloc((*nsorted)*sizeof(int));
69
70    /* Each process sends and receives the corresponding elements, using the MPI_Alltoallv */
71    /* operation. The arrays scounts and sdispls are used to specify the number of elements */
72    /* to be sent and where these elements are stored, respectively. The arrays rcounts */
73    /* and rdispls are used to specify the number of elements to be received, and where these */
74    /* elements will be stored, respectively. */
75    MPI_Alltoallv(elmnts, scounts, sdispls, MPI_INT, sorted_elmnts,
76        rcounts, rdispls, MPI_INT, comm);
77
78    /* Perform the final local sort */
79    qsort(sorted_elmnts, *nsorted, sizeof(int), IncOrder);
80
81    free(splitters); free(allpicks); free(scounts); free(sdispls);
82    free(rcounts);  free(rdispls);
83
84    return sorted_elmnts;
85  }
```

6.7 进程组和通信器

在许多并行程序中，通信操作需要限制在部分进程中。MPI提供几种方法将属于某一通信器的进程组划分为子组，每个子组都与不同的通信器相对应。通常使用MPI_Comm_split函数划分进程图，函数的定义如下：

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
        MPI_Comm *newcomm)
```

此函数是聚合操作函数，因此必需由通信器comm中的所有进程调用。除了通信器comm外，函数还有两个输入参数color和key，并将属于通信器comm的进程组划分成不相连的子组。每个子组包含对color参数提供同样值的所有进程。在每个子组内部，进程按由参数key的值定义的顺序确定等级，与它们在老的通信器（即comm）中进程等级的划分相联系。

272 newcomm参数为每个子组返回一个新的通信器。图6-7的例子展示用MPI_Comm_split函数

将通信器分裂的过程。如果每个进程都用图6-7所示的color和key的参数值调用MPI_Comm_split，那么将创建3个通信器，分别包含进程{0, 1, 2}、{3, 4, 5, 6}以及{7}。

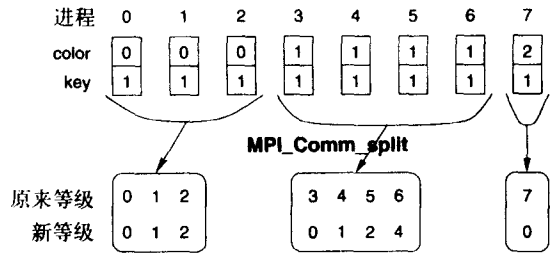


图6-7 使用MPI_Comm_split将通信器中的进程组划分为子组

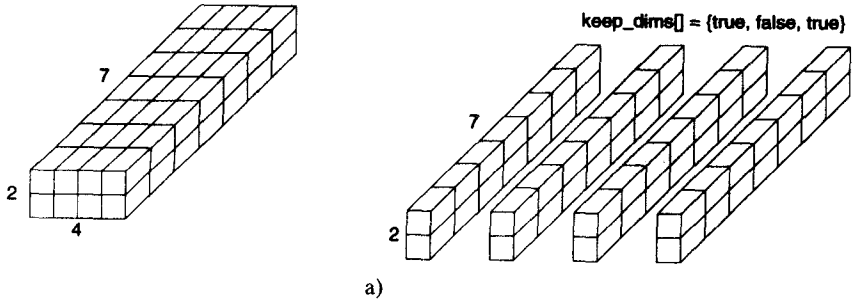
分裂笛卡儿拓扑结构 在许多并行算法中，进程按虚拟网格排列，在算法的不同步骤中，通信必需限制在网格的一个不同子集内。MPI提供了一种方法，能方便地将笛卡儿拓扑结构划分成低维网格的形式。

MPI提供函数MPI_Cart_sub，用于将笛卡儿拓扑结构划分成构成低维网格的子结构。例如，可以将二维结构划分成组，每个组包含沿着该结构的行或列的进程。函数MPI_Cart_sub的调用序列如下:

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
        MPI_Comm *comm_subcart)
```

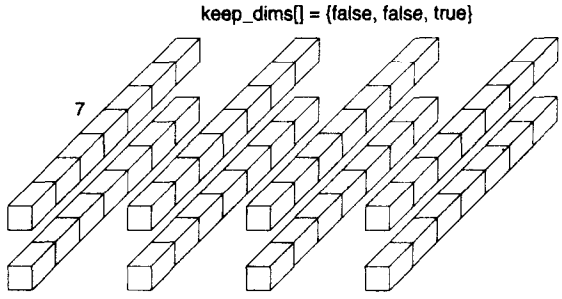
数组keep_dims用于指定如何划分笛卡儿拓扑结构。特别，如果keep_dims[i]为真(在C语言中为0)，那么在新的子结构中，第i维保持不变。例如，考虑一个$2 \times 4 \times 7$的三维结构。如果keep_dims的值是{true, false, true}，那么原始结构将分裂为4个大小为2×7的二维子结构，如图6-8a所示。如果keep_dims的值是{false, false, true}，那么原始结构将分裂为8个大小为7的一维子结构，如图6-8b所示。注意，新创建的子结构的数目等于沿未保留的各维上的进程数目的乘积。原始结构由通信器comm_cart指定，返回的通信器comm_subcart存储创建的子结构的信息。每个进程只返回一个通信器，对于不属于同一子结构的进程，由返回的通信器指定的组也不同。

属于某一给定子结构的进程可用如下方法确定：假设某三维结构的大小为$d_1 \times d_2 \times d_3$，且keep_dims值设定为{true, false, true}。若进程用坐标(x, y, z)给出，则属于同一子结构的进程组的坐标由$(*, y, *)$给出，这里坐标位置的“*”表示该坐标可以取任意可能的值。 273
同时注意，由于第二个坐标的值可以是d_2，一共创建d_2个子结构。



a)

图6-8 分裂大小为$2 \times 4 \times 7$的笛卡儿拓扑结构：a）分成大小为$2 \times 1 \times 7$的4个子结构；b）分成大小为$1 \times 1 \times 7$的8个子结构



b)

图6-8 （续）

由函数MPI_Cart_sub创建的子结构中的进程坐标也可以通过原始结构中的坐标获得，无需考虑与未保留维对应的坐标。例如，在基于列的子结构中，进程的坐标等于它在二维结构中的行坐标。如坐标为 (2, 3) 的进程，在子结构中有一个坐标为 (2)，与网格中的第3列对应。

实例：二维矩阵与向量相乘

在6.6.8节，我们提供了两个程序，计算对矩阵按行及按列分配时的矩阵与向量相乘 $x = Ab$。如8.1.2节的讲述，对矩阵分配还可以使用另一种二维的方法，据此可以得到矩阵与向量相乘算法二维并行形式。

274 程序6-8说明如何用这些结构及其划分来实现二维矩阵与向量相乘。参数n指定矩阵的维数，参数a和参数b分别指向矩阵A和向量b中本地存储的部分，参数x指向输出矩阵与向量乘积的本地部分。注意只有沿着进程网格第一列的进程才会在开始时存储b，并在返回时，同样的进程将存储结果x。为了简化程序，假设进程数目p是完全平方数，且n是 $\sqrt{p}$ 的整数倍。

程序6-8 二维矩阵与向量相乘

```
 1  MatrixVectorMultiply_2D(int n, double *a, double *b, double *x,
 2                          MPI_Comm comm)
 3  {
 4    int ROW=0, COL=1;  /* Improve readability */
 5    int i, j, nlocal;
 6    double *px;  /* Will store partial dot products */
 7    int npes, dims[2], periods[2], keep_dims[2];
 8    int myrank, my2drank, mycoords[2];
 9    int other_rank, coords[2];
10    MPI_Status status;
11    MPI_Comm comm_2d, comm_row, comm_col;
12
13    /* Get information about the communicator */
14    MPI_Comm_size(comm, &npes);
15    MPI_Comm_rank(comm, &myrank);
16
17    /* Compute the size of the square grid */
18    dims[ROW] = dims[COL] = sqrt(npes);
19
20    nlocal = n/dims[ROW];
21
22    /* Allocate memory for the array that will hold the partial dot-products */
23    px = malloc(nlocal*sizeof(double));
24
25    /* Set up the Cartesian topology and get the rank & coordinates of the process in this topology */
26    periods[ROW] = periods[COL] = 1; /* Set the periods for wrap-around connections */
```

```
27
28    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_2d);
29
30    MPI_Comm_rank(comm_2d, &my2drank); /* Get my rank in the new topology */
31    MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);  /* Get my coordinates */
32
33    /* Create the row-based sub-topology */
34    keep_dims[ROW] = 0;
35    keep_dims[COL] = 1;
36    MPI_Cart_sub(comm_2d, keep_dims, &comm_row);
37
38    /* Create the column-based sub-topology */
39    keep_dims[ROW] = 1;
40    keep_dims[COL] = 0;
41    MPI_Cart_sub(comm_2d, keep_dims, &comm_col);
42
43    /* Redistribute the b vector. */
44    /* Step 1. The processors along the 0th column send their data to the diagonal processors */
45    if (mycoords[COL] == 0 && mycoords[ROW] != 0) { /* I'm in the first column */
46      coords[ROW] = mycoords[ROW];
47      coords[COL] = mycoords[ROW];
48      MPI_Cart_rank(comm_2d, coords, &other_rank);                          275
49      MPI_Send(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d);
50    }
51    if (mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0) {
52      coords[ROW] = mycoords[ROW];
53      coords[COL] = 0;
54      MPI_Cart_rank(comm_2d, coords, &other_rank);
55      MPI_Recv(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d,
56          &status);
57    }
58
59    /* Step 2. The diagonal processors perform a column-wise broadcast */
60    coords[0] = mycoords[COL];
61    MPI_Cart_rank(comm_col, coords, &other_rank);
62    MPI_Bcast(b, nlocal, MPI_DOUBLE, other_rank, comm_col);
63
64    /* Get into the main computational loop */
65    for (i=0; i<nlocal; i++) {
66      px[i] = 0.0;
67      for (j=0; j<nlocal; j++)
68        px[i] += a[i*nlocal+j]*b[j];
69    }
70
71    /* Perform the sum-reduction along the rows to add up the partial dot-products */
72    coords[0] = 0;
73    MPI_Cart_rank(comm_row, coords, &other_rank);
74    MPI_Reduce(px, x, nlocal, MPI_DOUBLE, MPI_SUM, other_rank,
75        comm_row);
76
77    MPI_Comm_free(&comm_2d);  /* Free up communicator */
78    MPI_Comm_free(&comm_row);  /* Free up communicator */
79    MPI_Comm_free(&comm_col);  /* Free up communicator */
80
81    free(px);
82  }
```

6.8 书目评注

有关MPI的最好资料来源是MPI库本身的实际参考手册[Mes94]。在写这本书时，MPI标准已有两个主要版本。第一个版本1.0版于1994年发布，其最近的修订1.2版，已由绝大多数硬件供应商实现。MPI标准的第二个版本2.0版[Mes97]，对1.x版的许多方面进行了增强，如单侧通信、动态进程创建以及扩展的聚合操作等。然而，虽然标准在1997年就制定了，却没有

广泛可用的MPI-2实现来支持由标准指定的所有特性。除了上面提到的参考手册外，许多书籍也致力于使用MPI的并行编程[Pac98, GSNL98, GLS99]。

除了由多个硬件提供商提供的MPI实现外，许多政府研究机构和大学也开发了多个公用的MPI实现，其中包括由Argonne国家实验室发布的MPICH[GLDS96, GL96b]（从http://www-
276 unix.mcs.anl.gov/mpi/mpich可以获得相关信息），以及由Indianna大学发布的LAM-MPI（在http://www.lam-mpi.org可以获得相关信息），这些MPI实现已得到广泛的应用，并能移植到多种不同的体系结构中。事实上，这些MPI实现已被用作一些专门化MPI实现的出发点，而这些专门化MPI实现很适合用在如千兆以太网和Myrinet网等流行的互连网络中。

习题

6.1 试描述只由发送进程执行缓冲操作的有缓冲发送和接收的消息传递协议。为了使这些协议切实可行，需要什么样的硬件支持？

6.2 无阻塞式通信操作的优点之一是它们能使数据的传送与计算并发进行。试讨论为了实现计算与通信的最大重叠对程序进行重新组织的类型。发送进程是不是比接收进程更能从这种重叠中受益？

6.3 如6.3.4节所述，MPI标准允许用两种不同形式的`MPI_Send`操作实现——一种使用有缓冲的发送；另一种使用阻塞式发送。试讨论MPI允许这两种不同实现的潜在原因。特别，请考虑不同的消息大小及不同的体系结构特点。

6.4 考虑如图6-5所示的从16个处理器到4 × 4二维网格的不同映射。说明在$n = \sqrt{p} \times \sqrt{p}$个处理器时，四种映射方案分别如何映射。

6.5 考虑矩阵与矩阵相乘的Cannon算法。我们讨论的Cannon算法中，矩阵A和矩阵B都被限制为正方形矩阵，映射到正方形网格的进程中。然而，Cannon算法可以扩展到A、B以及进程网格都不是正方形的情形。若矩阵A的大小为$n \times k$，矩阵B的大小为$k \times m$，则矩阵A和B相乘得到的矩阵C的大小为$n \times m$。同样，设排列在q行r列的网格中的进程数目为$q \times r$。请用Cannon算法设计一个MPI程序，实现在$q \times r$的进程网格上，将上述的两个矩阵相乘。

6.6 当矩阵的维数不是进程数目的整数倍时，说明按行的矩阵与向量相乘的程序（程序6-4）需要修改才能正确运行。

6.7 考虑按列实现的矩阵与向量相乘（程序6-5）。另一种实现方法是用`MPI_Allreduce`进行需要的归约操作，然后让每个进程从向量`fx`处复制向量x的本地存储部分。这种实现的成
277 本是什么？还有一种实现方法是以不同的进程作为根，执行p个单节点归约操作。这种实现的成本是什么？

6.8 考虑6.6.9节中所述的Dijkstra单源最短路径算法。说明为什么加权邻接矩阵的按列序分配要优于按行序分配。

6.9 对于$q \times r$的进程网格上大小为$n \times m$的矩阵，说明二维矩阵与向量相乘的程序（程序
278 6-8）为什么需要修改才能正确执行。

第7章 共享地址空间平台的编程

显式并行编程要求指定并行任务以及这些任务间的交互。这些交互可能表现为并发任务的同步形式，或中间结果的通信形式。在共享地址空间结构中，由于所有的处理器可以访问部分（或者全部）内存，通信是隐式指定的。因此，在共享地址空间计算机上的编程模式侧重于并发与同步的构建，以及相关开销最小化的技术等。本章将讨论共享地址空间的编程模式及其性能问题，以及基于命令模式编程的扩充等。

在不同的数据共享、并发模型和同步支持机制中，共享地址空间编程模式可能有所不同。在默认情况下，基于模型的进程假设与一个进程相关的所有数据都是私有的，除非用shmget和shmat这样的UNIX系统调用进行另外说明。虽然这一点在多用户系统中对提供保护很重要，但当用多个并发聚集协作解决同一个问题时是不需要的。与加强保护域相关的开销使得这种进程不大适用于并行编程。相反，轻量级进程及线程假设所有的内存都是全局的。通过释放保护域，轻量级进程及线程支持更快的运行。因此，这是并行编程的首选模型，构成本章的核心。通过推动线程的创建和同步，基于命令的编程模型扩充线程模型。在本章我们对使用线程和并行命令的编程进行多方面讨论。

7.1 线程基础

线程（thread）是程序流程中的单一控制流。下面用一个简单的例子来讲述线程：

例7.1 什么是线程

下面的代码段计算两个大小为$n \times n$的稠密矩阵的乘积。

```
1   for (row = 0; row < n; row++)
2       for (column = 0; column < n; column++)
3           c[row][column] =
4               dot_product(get_row(a, row),
5                           get_col(b, col));
```

在这个代码段中有n^2次迭代，每一次迭代都能独立执行。这样一种独立的指令序列称为一个线程。在上面的例子中，共有n^2个线程，每一个对应for循环中的一次迭代。由于每个线程能独立于其他线程执行，它们可在多处理器中并发调度。上面的代码段可以修改如下：

```
1   for (row = 0; row < n; row++)
2       for (column = 0; column < n; column++)
3           c[row][column] =
4               create_thread(dot_product(get_row(a, row),
5                                         get_col(b, col)));
```

这里，我们使用create_thread函数提供创建线程的机制，指定一个C语言函数为线程。这样，系统就能在多处理器中对线程进行调度。 ■

线程的逻辑内存模型 要在多处理器中执行例7.1的代码，每个处理器必须对矩阵a、b、c进行访问。这是通过一共享地址空间实现的（在第2章中讲述）。如图7-1a所示，在线程的逻辑计算机模型中，所有内存对于每个线程是全局可访问的。然而，由于线程作为函数调用，

与函数调用相对应的堆栈通常被处理成线程本地的堆栈，这也是出于对堆栈活性的考虑。由于线程是在运行时调度（而且任何对线程执行的事先调度都是不安全的），无法确定哪些堆栈是活的。因此，只有那些较差的程序才将堆栈（线程本地变量）当成全局数据。这就隐含如图7-1b所示的一个逻辑计算机模型，模型中的内存模块*M*中保存线程本地（分配的堆栈）数据。

280 虽然这种逻辑计算机模型给出同等可访问地址空间的观点，但是，模型的物理实现偏离这个假设。在像Origin 2000这样的分布式共享地址空间计算机中，物理上访问本地内存的成本要比访问远程内存的成本少一个数量级。即使在所有处理器对内存真正可以同等访问的结构中（如有全局共享内存的共享总线结构），处理器上高速缓存的存在还是会使内存存取时间出现偏差。所以，内存的本地访问问题对于在这样的结构中榨取性能是十分重要的。

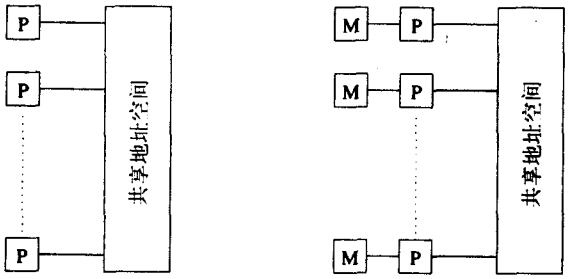


图7-1 基于线程的编程模式中的逻辑计算机模型

7.2 为什么要用线程

与消息传递编程模型相比，线程编程模型既有显著的优点，又有一些缺点。在讨论线程API之前，先简单地看一下这些优缺点。

软件可移植性 线程应用程序可以在串行计算机上开发，并且不作任何改变地在并行计算机上运行。这种在不同的结构平台上移植程序的能力是线程API的一个非常重要的优点。它的意义不仅仅在于软件的利用率上，还在于应用程序的开发上，因为超级计算机时间通常既少又贵。

躲避延迟 无论是串行还是并行程序，其最大的开销之一都是内存存取、I/O以及通信方面的存取延迟。但通过在同一处理器上执行多个线程，线程API就能躲避延迟时间（见第2章）。实际上，当一个线程等待通信操作时，其他的线程可以利用CPU，这样就能屏蔽相应的开销。

调度及负载平衡 在编写共享地址空间并行程序时，程序员要表现并发就必须最小化远程
281 交互以及空闲。虽然在许多结构化的应用程序中，分配同等的任务给处理器很容易实现，但在一些非结构化以及动态的应用程序（如游戏及离散优化）中，这一点却很难做到。线程API允许程序员指定大量的并发任务，并支持系统级的从任务到处理器的动态映射，以最小化空闲的开销。通过在系统级提供这种支持，线程API使程序员摆脱显式调度和平衡负载的负担。

容易编程，广泛使用 因为上述优点，线程化的程序要比相应的使用消息传递API的程序容易编写得多。但是，要使这两种程序达到同样的性能还需要其他方面的努力。随着POSIX线程API被广泛接受，开发POSIX线程的工具也越来越多，越来越稳定。从程序开发及软件工

程的角度来说，这些特点都很重要。

7.3 POSIX线程API

许多供应商提供专用的线程API。IEEE指定一个标准1003.1c-1995，POSIX API。POSIX也称为Pthreads，已成为标准的被绝大多数供应商支持的线程API。我们将使用Pthreads API来引出多线程的概念。这些概念本身在很大程度上独立于API，可以与其他的线程API（NT线程、Solaris线程以及Java线程等）一起用于编程。本章提出的所有说明性程序既能在工作站上运行，也能在支持Pthreads的并行计算机上运行。

7.4 线程基础：创建和终止

首先用一个简单的计算π值的程序来开始我们的讨论。

例7.2 计算π的线程程序

这里所用的方法，是基于在单位长度的正方形中产生随机数，并计算落在正方形的内接圆中的点的数目。由于圆的面积(πr^2) 等于$\pi/4$，正方形的区域为1×1，落在圆中的随机点的数目应该接近$\pi/4$。

使用线程计算π值的一个简单策略是：分配固定数目的点给每个线程，每个线程产生这些随机点并记录下落在圆内部的点的数目。在所有的线程执行完毕后，将所有的点相加计算π的值（将记录下的点数总和除以分配给所有线程的点，再乘以4）。

要实现这个线程程序，需要用一个函数创建线程并等待所有的线程完成执行（这样才能计算点数）。线程可以在Pthreads API中用函数pthread_create创建。这个函数的原型为 282

```
1  #include <pthread.h>
2  int
3  pthread_create (
4      pthread_t   *thread_handle,
5      const pthread_attr_t   *attribute,
6      void *    (*thread_function)(void *),
7      void   *arg);
```

pthread_create函数创建一个单一的线程，对应thread_function函数(以及任何其他的由thread_function调用的函数) 的调用。成功地创建一个线程后，有一个唯一的标识符与之对应，标识符被分配到由thread_handle指向的位置。线程的属性由参数attribute给出，当该参数为NULL时，创建一个具有默认属性的线程。在7.6节中将详细讨论attribute参数。字段arg指定一个指针，指向函数thread_function的参数。这个参数通常用来将工作区或其他线程指定的数据传送给一个线程。在compute_pi例子中，它用来传送一个当作随机种子的整数id。变量thread_handle要写在函数pthread_create返回之前；新的线程一旦创建就会立刻执行。如果线程在同一个处理器上调度，新的线程可能先占其创建者。这一点非常重要，因为在线程创建之前，所有的线程初始化程序都必须完成。否则，就可能出现基于线程调度的错误。这是一类很常见的错误，由数据存取时的竞争状态所致，这种竞争状态只在某些执行实例中会出现。成功地创建一个线程后，pthread_create返回0；否则返

回一个错误代码。有关错误代码详细内容，读者可以参考Pthread说明书。

在上面计算π值的程序中，首先读入所需线程的数目num_threads，以及所需的样本点数目sample_points。这些点在线程间平均分配。程序中使用数组hits来分配整数id给每个线程（这个id被随机数发生器作为产生随机数的种子）。同一数组也被用于记录每个线程返回前命中的次数（圆内的点数）。程序用pthread_create函数创建num_threads个线程，每个线程都调用函数compute_pi。

在每个compute_pi线程产生了分配数目的随机点并计算出命中率后，必须将结果
283 结合起来以确定π。主程序必须等待所有的线程运行完毕。这一点是通过函数pthread_join实现的，它挂起正在调用的线程，直到指定的线程终止。函数pthread_join的原型如下：

```
1  int
2  pthread_join (
3      pthread_t   thread,
4      void   **ptr);
```

对这个函数的调用将等待由thread给出其id的线程终止。在对pthread_join的一次成功的调用中，传递给pthread_exit的值返回到由ptr指向的位置。在pthread_join调用成功完成后返回0，否则返回一个错误代码。

所有的线程结合后，将合并后的命中率乘以4.0就得到π的值。完整的程序代码如下：

```
1  #include <pthread.h>
2  #include <stdlib.h>
3
4  #define MAX_THREADS     512
5  void *compute_pi (void *);
6
7  int total_hits, total_misses, hits[MAX_THREADS],
8      sample_points, sample_points_per_thread, num_threads;
9
10 main() {
11     int i;
12     pthread_t p_threads[MAX_THREADS];
13     pthread_attr_t  attr;
14     double computed_pi;
15     double time_start, time_end;
16     struct timeval tv;
17     struct timezone tz;
18
19     pthread_attr_init (&attr);
20     pthread_attr_setscope (&attr,PTHREAD_SCOPE_SYSTEM);
21     printf("Enter number of sample points: ");
22     scanf("%d", &sample_points);
23     printf("Enter number of threads: ");
24     scanf("%d", &num_threads);
25
26     gettimeofday(&tv, &tz);
27     time_start = (double)tv.tv_sec +
28                  (double)tv.tv_usec / 1000000.0;
29
30     total_hits = 0;
31     sample_points_per_thread = sample_points / num_threads;
32     for (i=0; i< num_threads; i++) {
33         hits[i] = i;
34         pthread_create(&p_threads[i], &attr, compute_pi,
35             (void *) &hits[i]);
36     }
```

```
37      for (i=0; i< num_threads; i++) {
38          pthread_join(p_threads[i], NULL);
39          total_hits += hits[i];
40      }
41      computed_pi = 4.0*(double) total_hits /
42          ((double)(sample_points));
43      gettimeofday(&tv, &tz);
44      time_end = (double)tv.tv_sec +
45                 (double)tv.tv_usec / 1000000.0;
46
47      printf("Computed PI = %lf\n", computed_pi);
48      printf(" %lf\n", time_end - time_start);
49  }
50
51  void *compute_pi (void *s) {
52      int seed, i, *hit_pointer;
53      double rand_no_x, rand_no_y;
54      int local_hits;
55
56      hit_pointer = (int *) s;
57      seed = *hit_pointer;
58      local_hits = 0;
59      for (i = 0; i < sample_points_per_thread; i++) {
60          rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
61          rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
62          if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
63              (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
64              local_hits ++;
65          seed *= i;
66      }
67      *hit_pointer = local_hits;
68      pthread_exit(0);
69  }
```

284

编程须知 读者必须注意，在上面的例子中用到函数rand_r(而不是如drand48这样更好的随机数产生器)。这是因为许多函数（包括rand和drand48）都不是可重入的(reentrant)。可重入函数是在调用过程中当另一个实例被挂起后还能安全地调用的函数。容易看出为什么所有线程函数必须是可重入的，因为线程在执行过程中可能被先占。如果另一个线程在先占的这一点开始执行同一个函数，不可重入的函数就可能出现意想不到的情况。

性能须知 我们在4处理器的SGI Origin 2000上执行这个程序。线程数的对数与执行时间的关系曲线如图7-2所示（曲线标记为“local”）。从图中可以看出，使用32个线程时，程序的运行时间大约比使用单线程时的时间少3.92倍。在4个处理器的计算机中，它对应的并行效率为0.98。

285

图7-2中的其他曲线解释一种称为假共享（false sharing）的重要性能开销。考虑将程序作如下修改：代替对本地变量local_hits加1和分配到在循环外部的数组中，现在直接将hits数组中对相应的项加1。具体做法是把第64行改为*(hit_pointer)++;，并删除第67行。容易验证修改后的程序与修改前的程序在语义上是完全相同的，然而，执行修改后的程序在图7-2中得到的却是标记为“spaced_1”的性能曲线。这表示性能非但没有提升，反而显著地降低了。

程序看上去没有什么大的改变，但影响却非常大，这可以用所谓假共享的现象来解释。在本例中，两个相邻接的数据项（很可能位于同一高速缓存行中）持续地被多个线程写入，这些线程可能在不同的处理器上调度。从第2章的讨论可知，向共享高速缓存行

写入将导致无效操作，随后的读操作必须从最近的写操作位置取高速缓存行。由于重复地进行增1操作，与hits数组相对应的高速缓存行将产生大量的无效操作和读操作。两个线程“虚假地”共享数据因为数据正好位于同一高速缓存行，这种情况称为假共享。

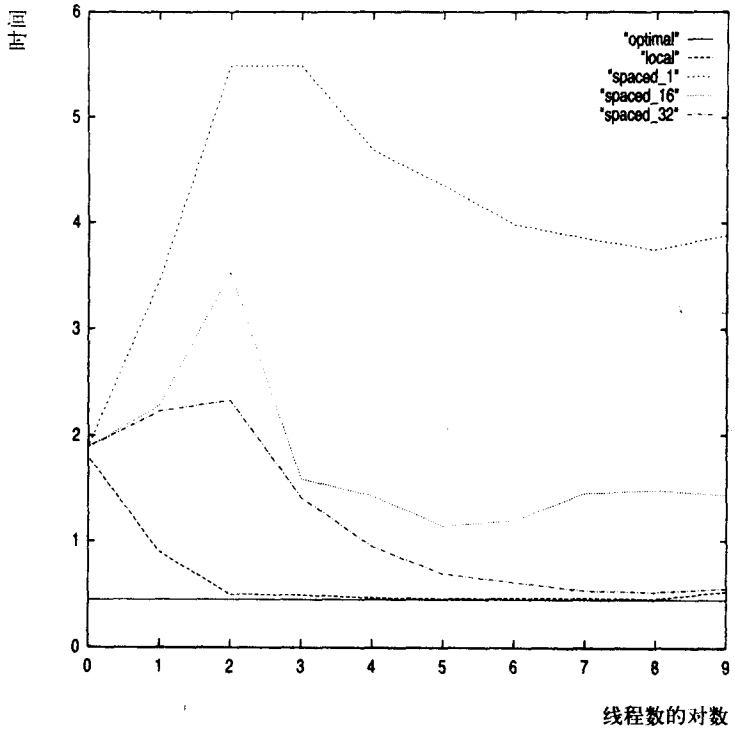


图7-2 compute_pi程序的执行时间与线程数的关系曲线

286 实际上用这个简单的例子可以估计系统的高速缓存行的大小。将hits改成二维数组，并且只用数组的第一列来存储点数。通过改变第二维的大小，就能强制hits数组中第一列的数据项放到不同的高速缓存行中（因为在C语言中，数组按行存储）。这个实验的结果在图7-2中由标记为“space_16”和“space_32”的曲线说明，这两条曲线对应的hits数组的第二维大小分别为16及32。从图中很容易看出，当数据项在空间上分开时，就能提高性能。这一点和我们的理解是一致的，因为将数据项分开，就能使数据项进入不同的高速缓存行，从而降低假共享的开销。 ■

在理解如何创建及连接线程后，再来研究Pthreads中线程同步的机制。

7.5 Pthreads中的同步原语

虽然通信在共享地址空间编程中是隐式的，但与编写正确的线程程序相关的主要工作是使得与数据存取或调度有关的并发线程同步。

7.5.1 共享变量的互斥

使用pthread_create和pthread_join调用能创建并发的任务。这些任务共同操作

数据并完成给定的任务。当多个线程试图操作同一数据项时，如果没有采取适当的方法使这些线程同步，结果往往会不一致。考察下面由多线程执行的代码段，其中my_cost是线程本地的变量，而best_cost是由所有线程共享的全局变量。

```
1  /* each thread tries to update variable best_cost as follows */
2  if (my_cost < best_cost)
3       best_cost = my_cost;
```

为了理解共享数据存取问题，先来试验上面代码段的一个执行实例。假设有两个线程，变量best_cost的初始值为100，在线程t1和t2中my_cost的值分别为50和75。如果两个线程并发地执行if语句中的条件，那么两个线程都会进入语句的then部分。依据哪个线程首先执行，最终best_cost的值可以是50，也可以是75。这里有两个问题：首先是结果的不确定性；其次，也是更重要的，在两个线程串行化也不能使变量best_cost的值为75的情况下，这个值是不一致的。这种情况不是人们希望看到的，有时也称为竞赛条件（这样称呼是因为计算的结果取决于竞争线程间的竞赛）。 287

上面提到的情况发生的原因是，前面讲到的测试并更新操作是原子操作，即不能再分成子操作的操作。而且，代码与一个临界段对应；即在任何时候，该代码段只能由一个线程执行。在像C语言这样的高级语言中，许多语句看上去是原子语句，但事实上不是；例如，语句global_count + = 5可能包含几条汇编指令，因此，在处理该语句时一定要小心。

线程API用mutex-lock（互斥锁）为实现临界段和原子操作提供支持。mutex-lock有两种状态：锁定状态与非锁定状态。在任意时刻，只有一个线程能锁定一个互斥锁。锁是原子操作，通常与操作共享数据的一段代码有关。要存取共享数据，线程必须首先获得mutex-lock，如果mutex-lock已经锁定，则进程就不能获得锁。这是因为，锁定的mutex-lock表示另一个线程当前处于临界段，不允许其他任何线程进入。当线程离开临界段后，它必须对mutex-lock解锁，这样其他的线程才能进入临界段。在程序的开始，所有的mutex-lock都必须初始化为非锁定状态。

Pthreads API提供多个处理mutex-lock的函数。函数pthread_mutex_lock可用来锁定mutex-lock。函数的原型为：

```
1  int
2  pthread_mutex_lock (
3       pthread_mutex_t   *mutex_lock);
```

对此函数的调用试图锁定互斥锁mutex-lock（mutex-lock的数据类型预定义为pthread_mutex_t）。如果mutex-lock已经锁定，则调用的线程阻塞；否则mutex-lock被锁定，并且调用的线程返回。函数成功返回时返回数值0。其他的值则表示死锁等错误情况。

离开临界段时，线程必须为与mutex-lock关联的段解锁。否则，其他的线程将不能进入这个段，通常会导致死锁。Pthreads函数pthread_mutex_unlock用来对mutex-lock解锁，这个函数的原型如下：

```
1  int
2  pthread_mutex_unlock (
3       pthread_mutex_t *mutex_lock);
```

调用此函数时，对于正常的mutex-lock，锁被放弃，阻塞的线程之一被调度进入临界段。指定的线程由调度策略确定。在7.6节讨论一些其他类型的锁（不同于正常的锁），以及相关

的pthread_mutex_unlock函数的语义。如果程序员试图对已经解锁的互斥锁或对被另一
288 个线程锁定的互斥锁，进行pthread_mutex_unlock操作，其结果是不确定的。

使用互斥锁之前，还需要另一个函数，即将互斥锁初始化为未锁定状态的函数。执行这一任务的pthreads函数为pthread_mutex_init，它的原型如下：

```
1  int
2  pthread_mutex_init (
3      pthread_mutex_t   *mutex_lock,
4      const pthread_mutexattr_t   *lock_attr);
```

此函数将互斥锁mutex_lock初始化为未锁定状态。互斥锁的属性由参数lock_attr指定。如果该参数设置为NULL，则函数将使用互斥锁的默认属性（正常互斥锁）。线程的属性对象在7.6节更详细讨论。

例7.3 计算一整数列表的最小项

有了基本的mutex-lock函数后，下面来编写一个简单的线程程序，计算一整数列表中的最小值。整数列表在线程间平等划分，每个线程划分的大小存储在变量partial_list_size中，指针list_ptr指向每个线程部分列表的起始位置，并传递给变量partial_list_size。完整的线程程序如下：

```
1  #include <pthread.h>
2  void *find_min(void *list_ptr);
3  pthread_mutex_t minimum_value_lock;
4  int minimum_value, partial_list_size;
5
6  main() {
7      /* declare and initialize data structures and list */
8      minimum_value = MIN_INT;
9      pthread_init();
10     pthread_mutex_init(&minimum_value_lock, NULL);
11
12     /* initialize lists, list_ptr, and partial_list_size */
13     /* create and join threads here */
14 }
15
16 void *find_min(void *list_ptr) {
17     int *partial_list_pointer, my_min, i;
18     my_min = MIN_INT;
19     partial_list_pointer = (int *) list_ptr;
20     for (i = 0; i < partial_list_size; i++)
21         if (partial_list_pointer[i] < my_min)
22             my_min = partial_list_pointer[i];
23     /* lock the mutex associated with minimum_value and
24     update the variable as required */
25     pthread_mutex_lock(&minimum_value_lock);
26     if (my_min < minimum_value)
289 27         minimum_value = my_min;
28     /* and unlock the mutex */
29     pthread_mutex_unlock(&minimum_value_lock);
30     pthread_exit(0);
31 }
```

编程须知 在本例中，对minimum_value的测试并更新操作由互斥锁minumum_value_lock保护。线程执行pthread_mutex_lock以获得对变量minumum_value的独占存取。只要这次存取获得后，数值便按要求更新，随后锁被释放。由于在任意时刻只有一个线程能

持有一个锁，也就只有一个线程能对变量进行测试并更新。 ■

例7.4 生产者-消费者工作队列

互斥锁的一个常见的用法是在线程间建立生产者-消费者关系。生产者创建任务，并把它们插入到工作队列中。消费者线程从任务队列中取出任务并执行。让我们来考虑这种模式的一个简单情况，其中每个任务队列中只能保存一个任务（通常情况下，工作队列会更长，但长度是有限的）。生产者-消费者关系是普遍存在的，习题7.4是在多媒体处理中的一个例子。一个简单的（和不正确的）线程程序里，生产者线程创建任务，并将任务放到共享的数据结构中，而消费者线程从共享数据结构中取出任务，并执行任务。但是，这种简单的程序并没有考虑如下可能性:

- 当前面一个任务没有被消费者线程取出时，生产者线程一定不能覆盖共享缓冲区。
- 在共享数据结构非空之前，消费者线程一定不能取出任务。
- 每一个消费者线程一次只能取出一个任务。

为了实施，我们可以用变量task_available。如果这个变量为0，则消费者线程必须等待，但生产者线程可以把任务插入到共享数据结构task_queue中。如果task_available为1，则生产者线程必须等待，不能马上将任务插入共享数据结构中，但消费者线程之一可以取出已有的任务。所有这些对变量task_available的操作必须由mutex_lock保护，以保证只有一个线程对它执行测试并更新。这个程序的线程版本如下:

```
1  pthread_mutex_t task_queue_lock;
2  int task_available;
3
4  /* other shared data structures here */                    290
5
6  main() {
7      /* declarations and initializations */
8      task_available = 0;
9      pthread_init();
10     pthread_mutex_init(&task_queue_lock, NULL);
11     /* create and join producer and consumer threads */
12 }
13
14 void *producer(void *producer_thread_data) {
15     int inserted;
16     struct task my_task;
17     while (!done()) {
18         inserted = 0;
19         create_task(&my_task);
20         while (inserted == 0) {
21             pthread_mutex_lock(&task_queue_lock);
22             if (task_available == 0) {
23                 insert_into_queue(my_task);
24                 task_available = 1;
25                 inserted = 1;
26             }
27             pthread_mutex_unlock(&task_queue_lock);
28         }
29     }
30 }
31
32 void *consumer(void *consumer_thread_data) {
33     int extracted;
```

```
34      struct task my_task;
35      /* local data structure declarations */
36      while (!done()) {
37          extracted = 0;
38          while (extracted == 0) {
39              pthread_mutex_lock(&task_queue_lock);
40              if (task_available == 1) {
41                  extract_from_queue(&my_task);
42                  task_available = 0;
43                  extracted = 1;
44              }
45              pthread_mutex_unlock(&task_queue_lock);
46          }
47          process_task(my_task);
48      }
49  }
```

编程须知 在本例中，生产者线程创建一个任务，并等待队列中的空间。这一点由变量task_available为0指示。对这个变量的测试并更新，以及向共享队列插入任务和从共享队列中取出任务，都由互斥锁task_queue_lock保护。一旦任务队列中的空间可用，则最近创建的任务就插入到任务队列中，于是变量task_available被设置为1，说明有任务可
291 用。在生产者线程中，变量inserted设置为1，表示最近创建的任务已被插入队列，这样生产者就可以产生下一个任务。无论最近创建的任务是否成功地插入到队列中，锁都会被解除。这样消费者线程在队列中有任务的情况下就能从队列中获取任务。如果锁没有被解除，线程就会死锁，这是因为消费者线程无法得到锁来获取任务，生产者也不能将任务插入到任务队列中。消费者线程等待任务变成可用的并在可用时执行它。如像生产者线程那样，在while循环的每一次迭代中，消费者都要解除锁，使生产者在队列为空的情况下将任务插入到队列中。 ■

1. 锁开销

锁表示串行化的点，因为临界段必须由线程一个接一个地执行。将大段程序封装在锁中会导致性能下降。最小化临界段的大小很重要。例如在上面的例子中，函数create_task和process_task处于临界段之外，但insert_into_queue和extract_from_queue函数处于临界段之内。前者在临界段之外是为了使临界段尽可能地小，而insert_into_queue和extract_from_queue函数处于临界段之中，是因为如果锁在更新task_available后被解除，但没有插入或获取任务，当其他的线程正在进行插入或获取时就可能获得对共享数据结构的访问，这样就会导致错误。因此，一定要非常小心地处理临界段和共享数据结构，以避免这种错误发生。

2. 减小锁开销

用另一个函数pthread_mutex_trylock可以减少与锁相关的空闲开销。这个函数试图锁定mutex_lock。如果锁定成功，函数返回0；如果锁已被其他的线程锁定，函数返回一个值EBUSY而不是阻塞线程操作。这样就能使线程执行其他的任务，并轮询互斥锁。此外，在典型的系统中，通常pthread_mutex_trylock要比pthread_mutex_lock快得多，因为它无需处理当多个线程等待锁时与锁有关的队列。函数pthread_mutex_trylock的原型如下：

```
1  int
2  pthread_mutex_trylock (
3      pthread_mutex_t *mutex_lock);
```

我们用下面的例子来说明这个函数的用法:

例7.5 找出一个列表中k个匹配的数

本例从给定的列表中找出k个与查询项匹配的数。列表在所有线程间同等划分。假设列表 292
有n项，有p个线程，每个线程负责查找列表的n/p项。使用pthread_mutex_trylock函数求解的程序段如下:

```
1  void *find_entries(void *start_pointer) {
2
3      /* This is the thread function */
4
5      struct database_record *next_record;
6      int count;
7      current_pointer = start_pointer;
8      do {
9          next_record = find_next_entry(current_pointer);
10         count = output_record(next_record);
11     } while (count < requested_number_of_records);
12 }
13
14 int output_record(struct database_record *record_ptr) {
15     int count;
16     pthread_mutex_lock(&output_count_lock);
17     output_count ++;
18     count = output_count;
19     pthread_mutex_unlock(&output_count_lock);
20
21     if (count <= requested_number_of_records)
22         print_record(record_ptr);
23     return (count);
24 }
```

这个程序段从数据库的一部分中找到一项，更新全局计数器，然后再找下一项。如果一个锁定-更新计数-解锁循环所需的时间为t_1，找到一个匹配数的时间为t_2，那么总的查找时间就是$(t_1 + t_2) \times n_{max}$，其中$n_{max}$是任一线程找到的最大匹配数目。如果$t_1$与$t_2$相当，那么锁将导致可观的开销。

用pthread_mutex_trylock函数能减少锁开销。每个线程现在寻找下一项，并试图得到锁并更新计数。如果另一个线程已经得到了锁，则把记录插入本地列表中，而线程能继续查找其他匹配的数。当线程最终得到锁时，就把到目前为止在本地找到的所有项插入到列表中（只要这个数目不超过所需要的数目）。相应的output_record函数如下:

```
1  int output_record(struct database_record *record_ptr) {
2      int count;
3      int lock_status;
4      lock_status = pthread_mutex_trylock(&output_count_lock);
5      if (lock_status == EBUSY) {
6          insert_into_local_list(record_ptr);
7          return(0);                                          293
8      }
9      else {
10         count = output_count;
11         output_count += number_on_local_list + 1;
12         pthread_mutex_unlock(&output_count_lock);
```

```
13          print_records(record_ptr, local_list,
14                requested_number_of_records - count);
15          return(count + number_on_local_list + 1);
16      }
17  }
```

编程须知 认真研究这个函数发现，如果更新全局计数的锁不可用，则函数将当前记录插入到一个本地列表中并返回；如果锁可用，则先将本地列表中的记录数目加到全局计数中，并再加1（代表当前记录）。然后再解除相关的锁，并用函数print_records打印要求的记录。

性能须知 这个版本代码比前面的代码的执行时间短，原因有两点：第一，前面也已提到，执行pthread_mutex_trylock的时间通常比执行pthread_mutex_lock的时间少得多。第二，由于每个锁可能有多个记录插入，锁操作的数目也减少了。实际查找的记录数目（经过所有线程）可能要比真正需要的记录数目略微大一点（因为在本地列表中可能有的项永远不会被打印）。然而，由于这个时间原本该是花在锁空闲上的，这种开销不会造成性能降低。

上面的例子说明用函数pthread_mutex_trylock而不用pthread_mutex_lock的原因。通常pthread_mutex_trylock函数用来减少与互斥锁相关的空闲开销。如果计算中临界段可以被延迟，其他的计算可以在这个间歇期进行，那么pthread_mutex_trylock将是选择的函数。另一个前面也曾提到过的关键因素是，对于绝大多数的实现而言，函数pthread_mutex_trylock比函数pthread_mutex_lock开销更小。事实上，对于高度优化的代码，即使需要使用函数pthread_mutex_lock，通常也需要在循环内部使用函数pthread_mutex_trylock，因为如果在最初的几次调用中获得了锁，使用pthread_mutex_trylock也会比pthread_mutex_lock开销小。 ■

7.5.2 用于同步的条件变量

如上一节提到的，不加区别地使用锁可能导致由阻塞线程引起的空闲开销。虽然函数
294 pthread_mutex_trylock减少这种开销，却带来对可用锁轮询的开销。例如，如果重写生产者-消费者的例子用函数pthread_mutex_trylock而不用pthread_mutex_lock，那么生产者和消费者就必须周期性地轮询锁的可用性（以及随后的缓冲区空间或队列中任务的可用性）。解决此问题的一个自然方法是，中止生产者的执行，直到空间变为可用（使用中断驱动机制而不是轮询机制）。空间的可用性由消费任务的消费者线程给出信号。实现该功能要使用条件变量（condition variable）。

条件变量是用来同步线程的数据对象。这个变量允许某一线程阻塞自己，直到指定的数据到达预定的状态。在生产者-消费者例子中，对消费者线程给出信号前，共享变量task_available必须变成1。布尔条件task_available == 1称为一个谓词，某一条件变量与这个谓词对应。如果谓词为真，条件变量就用来给出信号表示一个或多个等待条件的线程。一个条件变量可与多个谓词对应。但是，由于这样会造成程序难以排错，强烈建议不要这样做。

一个条件变量总是有一个互斥锁与之对应。一个线程锁定这个互斥锁并测试对共享变量（在这种情况下对task_available）定义的谓词；如果谓词非真，则线程等待与使用函数pthread_cond_wait的谓词相关的条件变量。这个函数的原型如下：

```
1  int pthread_cond_wait(pthread_cond_t *cond,
2      pthread_mutex_t *mutex);
```

对此函数的调用阻塞线程的执行，直至线程收到来自另一个线程的信号或者被一个操作系统的信号中断。除了阻塞线程外，函数pthread_cond_wait解除对互斥锁的锁定。这一点非常重要，否则没有其他线程能够对共享变量task_available进行操作，且谓词永远不会满足。当线程在得到信号被释放后，在恢复执行前等待重新获得互斥锁。把每个条件变量想象成与一个队列联系是方便的。执行对变量条件等待的线程放弃它们的锁并进入队列。当条件发出信号（使用函数pthread_cond_sign）时，队列中的线程之一便解除阻塞，并在互斥锁变为可用时，它就被送给这个线程（线程变成可运行的）。

在上面的生产者-消费者例子中，生产者线程产生任务，由于对互斥锁的锁定已被放弃（通过等待消费者），生产者线程能够将任务插入到队列中，并在锁定互斥锁后置task_available为1。由于谓词得到满足，生产者必须通过发出信号来唤醒消费者线程之一。这是用函数pthread_cond_signal实现的，函数的原型如下：

```
1       int pthread_cond_signal(pthread_cond_t *cond);
```

295

函数至少对当前等待条件变量cond的一个线程解除阻塞。然后生产者通过显式调用函数pthread_mutex_unlock放弃对互斥锁的锁定，使被阻塞的消费者线程之一消费任务。

在用条件变量重写生产者-消费者例子前，需要介绍两个初始化及消除条件变量的函数调用，分别为pthread_cond_init及pthread_cond_destroy。这两个函数调用的原型如下：

```
1   int pthread_cond_init(pthread_cond_t *cond,
2       const pthread_condattr_t *attr);
3   int pthread_cond_destroy(pthread_cond_t *cond);
```

函数pthread_cond_init初始化一个条件变量（由cond指向），其属性由属性对象attr定义。将attr置为NULL对条件变量赋予默认的属性。如果在程序的某处，条件变量不再是需要的，可以用函数pthread_cond_destroy将它废弃。有了这两个处理条件变量的函数，就可以重写生产者-消费者程序段如下：

例7.6 使用条件变量的生产者-消费者程序

条件变量在工作队列满时，可用来阻塞生产者线程的执行，而在工作队列为空时，可用来阻塞消费者线程的执行。两个条件变量cond_queue_empty和cond_queue_full分别用来指定空和满的队列。与cond_queue_empty相关的谓词为task_available == 0，与cond_queue_full相关的谓词为task_available == 1。

生产者队列锁定与共享变量task_available对应的互斥锁task_queue_cond_lock。它检查task_available是否为0（即队列为空）。如果为空，生产者就将任务插入到工作队列中，并通过给出条件变量cond_queue_full的信号唤醒所有等待的消费者线程。然后继续创建其他的任务。如果task_available为1（即队列是满的），生产者对条件变量cond_queue_empty执行条件等待（即等待队列变空）。此时，不难看出隐式解除对task_queue_cond_lock的锁定的原因。如果没有解除锁定，所有的消费者都不能消费任务，队列永远也不会变空。此时，生产者线程被阻塞。由于消费者可获得锁，消费者线程可以消费任务，并在任务从工作队列除去时给出条件变量cond_queue_empty的信号。

消费者线程锁定互斥锁task_queue_cond_lock，检查共享变量task_available

296 是否为1。如果不是，它就执行对cond_queue_full的条件等待。(注意，信号是在任务插入到工作队列时从生产者处产生的。) 如果有任务可用，消费者将任务从工作队列中取出并发送信号给生产者。以这种方式，生产者和消费者线程通过彼此发信号进行操作。不难看出，这种操作方式与基于中断的操作类似，但不同于pthread_mutex_trylock基于轮询的操作。完成这种生产者-消费者行为的程序段如下:

```
1  pthread_cond_t cond_queue_empty, cond_queue_full;
2  pthread_mutex_t task_queue_cond_lock;
3  int task_available;
4
5  /* other data structures here */
6
7  main() {
8      /* declarations and initializations */
9      task_available = 0;
10     pthread_init();
11     pthread_cond_init(&cond_queue_empty, NULL);
12     pthread_cond_init(&cond_queue_full, NULL);
13     pthread_mutex_init(&task_queue_cond_lock, NULL);
14     /* create and join producer and consumer threads */
15 }
16
17 void *producer(void *producer_thread_data) {
18     int inserted;
19     while (!done()) {
20         create_task();
21         pthread_mutex_lock(&task_queue_cond_lock);
22         while (task_available == 1)
23             pthread_cond_wait(&cond_queue_empty,
24                 &task_queue_cond_lock);
25         insert_into_queue();
26         task_available = 1;
27         pthread_cond_signal(&cond_queue_full);
28         pthread_mutex_unlock(&task_queue_cond_lock);
29     }
30 }
31
32 void *consumer(void *consumer_thread_data) {
33     while (!done()) {
34         pthread_mutex_lock(&task_queue_cond_lock);
35         while (task_available == 0)
36             pthread_cond_wait(&cond_queue_full,
37                 &task_queue_cond_lock);
38         my_task = extract_from_queue();
39         task_available = 0;
40         pthread_cond_signal(&cond_queue_empty);
41         pthread_mutex_unlock(&task_queue_cond_lock);
42         process_task(my_task);
43     }
44 }
```

297 **编程须知** 在这个代码段中，有一点要特别注意: 对与条件变量相关的谓词的检查是在循环内进行的。可能有人会认为，当cond_queue_full被断定时，task_available的值必定为1。但是，对条件的检查最好在循环内进行，因为线程可能会由于其他的原因 (如OS信号) 被唤醒。在其他情况下，如果条件变量通过条件广播发出信号 (对所有等待的线程发出信号而不只是对一个线程)，较早得到锁的线程就会使条件无效。当有多个生产者及消费者时，一个队列中可用的任务可能要被其他的消费者之一消费。

性能须知 当线程执行条件等待时，它将自己从可运行列表中移走——因此，在它唤醒前不会使用CPU周期。这一点与互斥锁相反，互斥锁在轮询锁时要消耗CPU周期。 ■

在上面的例子中，每个任务只能被一个消费者线程消费。因此，我们每次选择向一个阻塞的线程发信号。在某些其他的计算中，唤醒所有等待条件变量的线程而不是单个线程可能会有利。这一点可以通过函数pthread_cond_broadcast实现。

```
1  int pthread_cond_broadcast(pthread_cond_t *cond);
```

这样做的一个例子是在生产者-消费者程序中，工作队列很大，在每个插入周期都有多个任务要插入到工作队列。这一问题留给读者作为习题（习题7.2）。使用函数pthread_cond_broadcast的另一个例子是7.8.2节所述的实现障碍。

在条件等待时设置超时通常是很有用的。使用函数pthread_cond_timedwait，线程就能执行等待条件变量，直至指定的时间届满。此时，如果线程没有收到信号或广播，就会被自己唤醒。这个函数的原型为:

```
1  int pthread_cond_timedwait(pthread_cond_t *cond,
2       pthread_mutex_t *mutex,
3       const struct timespec *abstime);
```

如果在收到信号或广播前，指定的绝对时间abstime届满，函数返回一个出错信息。当函数成为可用时，它也会重新获得互斥锁。

7.6 控制线程及同步的属性

到目前为止的讨论中，我们发现，像线程及同步变量这样的实体通常都与几种属性有关。例如，不同线程的调度方式可能也不同（如循环调度、优先调度，等等），堆栈的大小可能也不一样，等等。同样，像互斥锁这样的同步变量也可能有多种不同的类型。Pthreads API允许 298
程序员用*属性对象*（attribute object）改变实体的默认属性。

属性对象是一个数据结构，用来描述实体（线程、互斥锁、条件变量）的性质。在创建线程或同步变量时，可以指定决定实体性质的属性对象。一旦创建，线程或同步变量的性质就大体上固定了（Pthreads允许用户改变线程的优先权），以后对属性对象的修改不能改变先于用属性对象创建的实体的性质。使用属性对象的好处有：第一，它将程序语义与实现分开。线程的性质由用户指定。这些性质在系统层次如何实现对于用户来说是透明的。这样就能使程序在不同操作系统间有更大的可移植性。第二，使用属性对象增强程序的模块性及可读性。第三，它使程序员修改程序更容易。比如，如果用户想对所有线程的调度从循环调度形式改为时间片调度形式，他们只需改变属性对象中的指定属性。

要创建具备要求性质的属性对象，首先必须创建带有默认性质的对象，然后按要求修改对象。下面我们考查为线程和同步变量完成属性对象创建的Pthreads函数。

7.6.1 线程的属性对象

函数pthread_attr_init用来创建线程的属性对象。此函数的原型如下:

```
1  int
2  pthread_attr_init (
3       pthread_attr_t   *attr);
```

这个函数用默认值初始化属性对象attr。初始化成功后，函数返回0，否则返回一个错误代码。属性对象可用函数pthread_attr_destroy取消，这个函数的原型如下：

```
1  int
2  pthread_attr_destroy (
3      pthread_attr_t   *attr);
```

成功地取消属性对象attr后，函数的调用返回0。与属性对象有关的各个性质可用如下的函数修改：pthread_attr_setdetachstate，pthread_attr_setguardsize_np，pthread_attr_setstacksize，pthread_attr_setinheritsched，
299 pthread_attr_setschedpolicy和pthread_attr_setschedparam。这些函数分别用来设置线程属性对象的分离状态、堆栈保护大小、堆栈大小、调度策略是否从创建线程继承、调度策略（在未继承的情况下）以及调度参数。读者可以参考Pthreads手册了解这些函数的详细描述。对于绝大多数并行程序而言，通常默认的线程性质就够用了。

7.6.2 互斥锁的属性对象

Pthreads API支持三种不同的锁。对这三种锁的锁定和解锁使用同样的函数；然而，锁的类型由锁属性决定。到目前为止的例子中用到的互斥锁称为*正常互斥锁*（normal mutex）。这是一种默认类型的锁。在任意时刻，只允许一个线程锁定互斥锁。如果获得锁的线程试图再一次锁定它，则第二次锁定调用将导致死锁。

考虑下面在二叉树中搜索元素的线程例子。为了保证在搜索过程中其他的线程不改变树，线程使用一个互斥锁tree_lock锁定树。搜索函数如下：

```
1  search_tree(void *tree_ptr)
2  {
3      struct node *node_pointer;
4      node_pointer = (struct node *) tree_ptr;
5      pthread_mutex_lock(&tree_lock);
6      if (is_search_node(node_pointer) == 1) {
7          /* solution is found here */
8          print_node(node_pointer);
9          pthread_mutex_unlock(&tree_lock);
10         return(1);
11     }
12     else {
13         if (tree_ptr -> left != NULL)
14             search_tree((void *) tree_ptr -> left);
15         if (tree_ptr -> right != NULL)
16             search_tree((void *) tree_ptr -> right);
17     }
18     printf("Search unsuccessful\n");
19     pthread_mutex_unlock(&tree_lock);
20 }
```

如果tree_lock是正常互斥锁，则第一个对函数search_tree的递归调用将导致死锁，因为已获得锁的线程试图锁定一个互斥锁。为了解决这个问题，Pthreads API支持一种*递归互斥锁*（recursive mutex）。递归互斥锁允许单一的线程多次锁定互斥锁。线程每次锁定一个互斥锁时，锁计数器加1，每次解锁计数器减1。如果其他任何线程想成功锁定一个递归互斥锁，锁计数器必须为0（即每一个被其他线程的锁定必须有一个对应的解锁）。当线程函数需要递归地调用自己时，就要用到递归互斥锁。

除了正常及递归互斥锁以外，Pthreads API还支持第三种互斥锁，称为*错误检查互斥锁*

(errorcheck mutex)。错误检查互斥锁的操作与正常互斥锁相似——一个线程只能锁定一个互斥锁一次。然而，与正常互斥锁不同，当线程试图锁定一个已经锁定的互斥锁时，错误检查互斥锁将返回一个错误而不是死锁。因此，错误检查互斥锁在排错时更有用。 300

互斥锁的类型可用互斥锁属性对象来指定。为了对默认值创建和初始化一个互斥锁属性对象，Pthreads提供函数pthread_mutexattr_init。这个函数的原型如下:

```
1  int
2  pthread_mutexattr_init (
3       pthread_mutexattr_t   *attr);
```

这个函数创建并初始化一个互斥锁属性对象attr，互斥锁的默认类型为正常互斥锁。Pthreads提供函数pthread_mutexattr_settype_np用来设置由互斥锁属性对象指定的类型。这个函数的原型如下:

```
1  int
2  pthread_mutexattr_settype_np (
3       pthread_mutexattr_t   *attr,
4       int   type);
```

这里，type指定互斥锁的类型，与正常、递归和错误检查这三种互斥锁的类型相对应，它可以取下述值:

- PTHREAD_MUTEX_NORMAL_NP
- PTHREAD_MUTEX_RECURSIVE_NP
- PTHREAD_MUTEX_ERRORCHECK_NP

用函数pthread_attr_destroy可以取消互斥锁属性对象，函数以互斥锁属性对象attr作为唯一参数。

7.7 线程注销

考虑在象棋比赛中用来对一系列位置进行估值的一个简单程序。假设有k次移动，每次移动由一个独立的线程估值。如果在任意时刻，某一位置确立具有某个品质，那么对其他已知品质差的位置必须停止估值。换句话说，对相应棋盘位置进行估值的线程必须被注销。POSIX线程用函数pthread_cancel提供线程注销功能。这个函数的原型如下:

```
1  int
2  pthread_cancel (
3       pthread_t   thread);
```

301

这里，thread是要被注销线程的句柄。一个线程可以注销自己，也可以注销其他线程。调用此函数后，注销就发送给指定的线程，但不能保证指定的线程一定会收到注销，也不能保证指定的线程执行注销。线程可以保护自己不被注销。当真正执行注销时，调用清理函数恢复线程数据结构。这一过程与调用函数pthread_exit终止线程类似。对线程的注销独立于发送原来注销请求的线程。函数pthread_cancel在注销发送后返回，而注销操作可能推迟执行。注销成功时返回0，但并不代表请求的线程已被注销；这仅说明指定的线程是有效的注销线程。

7.8 复合同步结构

虽然Pthreads API提供了一系列基本的同步结构，但是经常需要更高层次的结构。这些较高

层次的结构可以用基本同步结构来构造。在本节中，将考查这些结构以及它们的性能和应用。

7.8.1 读-写锁

在许多应用程序中，某一数据结构会被频繁地读出，但很少写入。对于这种情况，多读可以在不带来任何一致性问题的情况下进行。然而，写必须被串行化。这就引出了另一种结构——读写锁。正在读共享数据项的线程获得对变量的读出锁，如果其他线程已经有了读出锁，则读出锁就被授与。如果存在对数据的写入锁（或者有排队的写入锁），线程将执行条件等待。同样，如果有多个线程请求写入锁，这些线程也必须执行条件等待。我们使用这些原则设计读出锁函数mylib_rwlock_rlock、写入锁函数mylib_rwlock_wlock以及解锁函数mylib_rwlock_unlock。

所述的读-写锁基于一种称为mylib_rwlock_t的数据结构。这种结构含有读出者的数量、写入者（0/1整数表示写入者是否存在）、读出者能进行时发出的条件变量readers_proceed信号、写入者能进行时发出的条件变量writer_proceed信号、挂起的写入者计数器pending_writes以及与共享数据结构相对应的互斥锁read_write_lock。函数mylib_rwlock_init用来对这个数据结构的不同分量进行初始化。

函数mylib_rwlock_rlock试图对数据结构加上读出锁。它检查是否有写入锁或挂起的写入者。如果有，就执行对条件变量readers_proceed的条件等待，否则读出者计数器
302 加1，并授与一个读出锁。函数mylib_rwlock_wlock试图对数据结构加上写入锁。它检查是否有读出者或写入者；如果有，挂起的写入者计数器加1，并执行对条件变量writer_proceed的等待。如果没有读出者或写入者，就授予一个写入锁并继续。

函数mylib_rwlock_unlock解除读出锁或写入锁。它检查是否存在写入锁，如果有，该函数就会通过设置writer的字段为0为数据结构解锁。如果有读出者存在，读出者readers的数量减1。如果没有剩余的读出者，但有挂起的写入者，函数发出写入者之一继续的信号（通过发出writer_proceed信号）。如果没有挂起的写入者，但有挂起的读出者，则函数发出所有读出者线程继续的信号。初始化和锁定/解锁的代码如下：

```
1  typedef struct {
2      int readers;
3      int writer;
4      pthread_cond_t readers_proceed;
5      pthread_cond_t writer_proceed;
6      int pending_writers;
7      pthread_mutex_t read_write_lock;
8  } mylib_rwlock_t;
9
10
11 void mylib_rwlock_init (mylib_rwlock_t *l) {
12    l -> readers = l -> writer = l -> pending_writers = 0;
13    pthread_mutex_init(&(l -> read_write_lock), NULL);
14    pthread_cond_init(&(l -> readers_proceed), NULL);
15    pthread_cond_init(&(l -> writer_proceed), NULL);
16 }
17
18 void mylib_rwlock_rlock(mylib_rwlock_t *l) {
19    /* if there is a write lock or pending writers, perform condition
20    wait.. else increment count of readers and grant read lock */
21
```

```
22      pthread_mutex_lock(&(l -> read_write_lock));
23      while ((l -> pending_writers > 0) || (l -> writer > 0))
24          pthread_cond_wait(&(l -> readers_proceed),
25              &(l -> read_write_lock));
26      l -> readers ++;
27      pthread_mutex_unlock(&(l -> read_write_lock));
28  }
29
30
31  void mylib_rwlock_wlock(mylib_rwlock_t *l) {
32      /* if there are readers or writers, increment pending writers
33      count and wait. On being woken, decrement pending writers
34      count and increment writer count */
35
36      pthread_mutex_lock(&(l -> read_write_lock));
37      while ((l -> writer > 0) || (l -> readers > 0))  {
38          l -> pending_writers ++;
39          pthread_cond_wait(&(l -> writer_proceed),
40              &(l -> read_write_lock));
41      }                                                          303
42      l -> pending_writers --;
43      l -> writer ++
44      pthread_mutex_unlock(&(l -> read_write_lock));
45  }
46
47
48  void mylib_rwlock_unlock(mylib_rwlock_t *l) {
49      /* if there is a write lock then unlock, else if there are
50      read locks, decrement count of read locks. If the count
51      is 0 and there is a pending writer, let it through, else
52      if there are pending readers, let them all go through */
53
54      pthread_mutex_lock(&(l -> read_write_lock));
55      if (l -> writer > 0)
56          l -> writer = 0;
57      else if (l -> readers > 0)
58          l -> readers --;
59      pthread_mutex_unlock(&(l -> read_write_lock));
60      if ((l -> readers == 0) && (l -> pending_writers > 0))
61          pthread_cond_signal(&(l -> writer_proceed));
62      else if (l -> readers > 0)
63          pthread_cond_broadcast(&(l -> readers_proceed));
64  }
```

下面用一些例子说明读-写锁的用法。

例7.7 用读-写锁计算一个数字列表中的最小数

计算一个数字列表中的最小值是读-写锁的一个简单应用。在前面的实现中，一个锁与最小值相关。所有线程在需要的时候锁定并更新最小值。通常，检查最小值的次数要大于最小值更新的次数。因此，好的方法是用读出锁允许多次读，只在需要时在写入锁后写入。相应的程序段如下：

```
1   void *find_min_rw(void *list_ptr) {
2       int *partial_list_pointer, my_min, i;
3       my_min = MIN_INT;
4       partial_list_pointer = (int *) list_ptr;
5       for (i = 0; i < partial_list_size; i++)
6           if (partial_list_pointer[i] < my_min)
7               my_min = partial_list_pointer[i];
8       /* lock the mutex associated with minimum_value and
```

```
9       update the variable as required */
10      mylib_rwlock_rlock(&read_write_lock);
11      if (my_min < minimum_value) {
12          mylib_rwlock_unlock(&read_write_lock);
13          mylib_rwlock_wlock(&read_write_lock);
14          minimum_value = my_min;
15      }
16      /* and unlock the mutex */
304     17      mylib_rwlock_unlock(&read_write_lock);
18      pthread_exit(0);
19  }
```

编程须知 在本例中，每个线程计算自己部分列表中的最小元素。然后试图对与全局最小值对应的锁加上读出锁。如果全局最小值大于本地最小值（这样就需要一次更新），则放弃读出锁，寻找写入锁。一旦获得写入锁，就能更新全局最小值。从读-写锁得到的性能增益受线程数量及所需更新（写入锁）数量的影响。在极端的情况下，全局最小值中的第一个值也是真正的最小值，就不用再寻找写入锁。此时，这种使用读-写锁的版本的程序性能较好。相反，如果每个线程必须更新全局最小值，就会有过多的读出锁，增加程序的开销。 ■

例7.8 用读-写锁实现散列表

在从数据库查询到状态空间搜索这样的应用程序中，一个常见的操作是从数据库中搜索一个关键码，数据库按散列表组织。在本例中，假设通过将冲突的项链接成链表来处理冲突。每个链表有一个锁与之对应。锁保证不会在同时对链表更新和查找。这里我们考虑两种程序版本：一种使用互斥锁，另一种使用本节中介绍的读-写锁。

使用互斥锁的程序版本将关键码散列到表中，锁定与表索引对应的互斥锁，并在链表中进行搜索/更新。实现这个操作的线程函数如下：

```
1   manipulate_hash_table(int entry) {
2       int table_index, found;
3       struct list_entry *node, *new_node;
4
5       table_index = hash(entry);
6       pthread_mutex_lock(&hash_table[table_index].list_lock);
7       found = 0;
8       node = hash_table[table_index].next;
9       while ((node != NULL) && (!found)) {
10          if (node -> value == entry)
11              found = 1;
12          else
13              node = node -> next;
14      }
15      pthread_mutex_unlock(&hash_table[table_index].list_lock);
16      if (found)
17          return(1);
18      else
19          insert_into_hash_table(entry);
305 20  }
```

这里，函数insert_into_hash_table在进行实际插入前必须锁定hash_table[table_index].list_lock。当在散列表中发现大部分查询时（即它们不需要被插入），这些搜索就会被串行化。容易看出，可以安全地允许多个线程搜索散列表，只有对表的更新必须被串行化。这样就能用读-写锁来实现。函数manipulate_hash_table可以重写如下：

```
1  manipulate_hash_table(int entry)
2  {
3      int table_index, found;
4      struct list_entry *node, *new_node;
5
6      table_index = hash(entry);
7      mylib_rwlock_rlock(&hash_table[table_index].list_lock);
8      found = 0;
9      node = hash_table[table_index].next;
10     while ((node != NULL) && (!found)) {
11         if (node -> value == entry)
12             found = 1;
13         else
14             node = node -> next;
15     }
16     mylib_rwlock_rlock(&hash_table[table_index].list_lock);
17     if (found)
18         return(1);
19     else
20         insert_into_hash_table(entry);
21 }
```

这里，函数insert_into_hash_table在进行实际插入前必须得到hash_table[table_index].list_lock的写入锁。

编程须知 在这个例子中，假设list_lock字段已被定义为类型mylib_rwlock_t，并且所有与散列表相关的读-写锁已用函数mylib_rwlock_init初始化。与使用互斥锁相比，使用mylib_rwlock_rlock允许多个线程并发地搜索各自的数据项。这样的话，如果成功的搜索次数超过插入次数，就能得到很好的性能。注意，函数insert_into_hash_table必须做适当的修改才能使用写入锁（而不是如像前面的互斥锁）。 ■

弄清使用读-写锁优于使用正常锁的情况是很重要的。由于在写入时，读-写锁与正常互斥锁相比没有优势，只在有大量的读出操作时使用读-写锁才有利。此外，当临界段变大时，使用读-写锁的优势更大。这是因为，由正常互斥锁导致的串行化开销比较高。最后，由于读-写锁依赖条件变量，支撑的线程系统必须提供快速条件等待、发信号以及广播函数。通过 306
对读-写锁做一个简单的分析，可以了解使用读-写锁的优点（习题7.7）。

7.8.2 障碍

在线程化程序（以及其他的并行程序）中，barrier（障碍）是重要且经常用到的结构。用障碍调用挂起一个线程，直至其他所有加入障碍的线程到达障碍。障碍可以用计数器、互斥锁以及条件变量实现。（障碍也可以只使用互斥锁实现，然而，这样的实现会引起繁忙等待的开销。）一个整数（计数器）用于记录到达障碍的线程数目，如果计数小于线程总数目，线程执行条件等待。最后进入的线程（将计数器设置为线程数目）使用条件广播唤醒所有的线程。完成这个过程的代码如下：

```
1  typedef struct {
2      pthread_mutex_t count_lock;
3      pthread_cond_t ok_to_proceed;
4      int count;
5  } mylib_barrier_t;
6
7  void mylib_init_barrier(mylib_barrier_t *b) {
8      b -> count = 0;
```

```
9       pthread_mutex_init(&(b -> count_lock), NULL);
10      pthread_cond_init(&(b -> ok_to_proceed), NULL);
11  }
12
13  void mylib_barrier (mylib_barrier_t *b, int num_threads) {
14      pthread_mutex_lock(&(b -> count_lock));
15      b -> count ++;
16      if (b -> count == num_threads) {
17          b -> count = 0;
18          pthread_cond_broadcast(&(b -> ok_to_proceed));
19      }
20      else
21          while (pthread_cond_wait(&(b -> ok_to_proceed),
22              &(b -> count_lock)) != 0);
23      pthread_mutex_unlock(&(b -> count_lock));
24  }
```

在上面的障碍实现中，线程进入障碍，直至广播信号释放线程。线程逐个地被释放，因为互斥锁count_lock在线程间逐个地传递。因此，对于n个线程，这个函数的执行时间下界为非常小的$O(n)$。用多个障碍变量可加速这种障碍实现。

下面考虑另一种障碍实现方法：用$n/2$对条件变量-互斥锁来实现n个线程的障碍。障碍工作如下：首先，线程配成对，每对线程共享一个条件变量-互斥锁对，指定的对成员等待两个
307 线程以成对障碍的方式到达。一旦这种情况发生，所有指定的成员就被组织成对，这一过程一直进行到只有一个线程时才结束。此时，所有的线程都已到达障碍点。我们必须在这时释放所有的线程。不过，释放它们需要发出所有$n/2$个条件变量信号。完成这项工作我们同样采用层次策略。线程对中指定的线程发送各自的条件变量信号。

```
1   typedef struct barrier_node {
2       pthread_mutex_t count_lock;
3       pthread_cond_t ok_to_proceed_up;
4       pthread_cond_t ok_to_proceed_down;
5       int count;
6   } mylib_barrier_t_internal;
7
8   typedef struct barrier_node mylog_logbarrier_t[MAX_THREADS];
9   pthread_t p_threads[MAX_THREADS];
10  pthread_attr_t attr;
11
12  void mylib_init_barrier(mylog_logbarrier_t b) {
13      int i;
14      for (i = 0; i < MAX_THREADS; i++) {
15          b[i].count = 0;
16          pthread_mutex_init(&(b[i].count_lock), NULL);
17          pthread_cond_init(&(b[i].ok_to_proceed_up), NULL);
18          pthread_cond_init(&(b[i].ok_to_proceed_down), NULL);
19      }
20  }
21
22  void mylib_logbarrier (mylog_logbarrier_t b, int num_threads,
23              int thread_id) {
24      int i, base, index;
25      i = 2;
26      base = 0;
27
28      do {
29          index = base + thread_id / i;
30          if (thread_id % i == 0) {
31              pthread_mutex_lock(&(b[index].count_lock));
```

```
32              b[index].count ++;
33              while (b[index].count < 2)
34                  pthread_cond_wait(&(b[index].ok_to_proceed_up),
35                      &(b[index].count_lock));
36                  pthread_mutex_unlock(&(b[index].count_lock));
37              }
38              else {
39                  pthread_mutex_lock(&(b[index].count_lock));
40                  b[index].count ++;
41                  if (b[index].count == 2)
42                      pthread_cond_signal(&(b[index].ok_to_proceed_up));
43                  while (pthread_cond_wait(&(b[index].ok_to_proceed_down),
44                      &(b[index].count_lock)) != 0);
45                  pthread_mutex_unlock(&(b[index].count_lock));
46                  break;
47              }
48              base = base + num_threads/i;
49              i = i * 2;
50      } while (i <= num_threads);
51      i = i / 2;
52      for (; i > 1; i = i / 2) {
53          base = base - num_threads/i;
54          index = base + thread_id / i;
55          pthread_mutex_lock(&(b[index].count_lock));
56          b[index].count = 0;
57          pthread_cond_signal(&(b[index].ok_to_proceed_down));
58          pthread_mutex_unlock(&(b[index].count_lock));
59      }
60  }
```

308

在这种障碍的实现中，可以把障碍看成为二叉树，线程到达树的叶节点。考虑有8个线程障碍的实例，线程0和线程1在一个叶节点配对。线程之一被指定为树的下一层中线程对的代表。在上面的例子中，线程0被指定为代表，它等待条件变量ok_to_proceed_up，以便线程1赶上。所有偶数编号的线程进入树的下一层。现在线程0与线程2配对，线程4与线程6配对。最终线程0与线程4配对。此时，线程0了解所有的线程已到达需要的障碍点，它通过发送条件变量ok_to_proceed_down信号释放线程。所有的线程被释放后，障碍完成。

容易看出，对于n个线程的障碍，树中有$n-1$个节点。每个节点与两个条件变量对应，一个变量用来释放线程，让线程赶上，另一个用来释放线程，让线程中止。另外，每个节点还与一个锁以及到达该节点的线程数目对应。树节点线性排列在数组mylog_logbarrier_t中，$n/2$个叶节点中含有前$n/2$个元素，更高一层中$n/4$个树节点含有$n/4$个元素，以此类推。

研究这个程序的性能很有意义。由于线性障碍中的线程一个接一个地被释放，可以作出合理的预测：即使在多处理器中，运行时间也与线程的数目成线性关系。图7-3绘出1000个线性障碍在32个处理器的SGI Origin 2000上的运行时间。串行障碍的线性运行时间很清楚地在运行时间中反映出来。在一个处理器上执行的对数障碍的工作量渐近地等于一个串行障碍（虽然有一个更大的常数）。然而，在并行计算机上，在二叉障碍树的子树被分配给不同处理器这样一种理想的情况下，运行时间以$O(n/p + \log p)$增长。如果不能像子树分配给每个处理器一样相应地检查或者分配成块的线程，虽然这一点很难做到，但对数障碍还是显示出比串行性障碍好得多的性能。对于给定数目的处理器，当n变大时，对数障碍的性能趋于与n成线性关系，因为在式$O(n/p + \log p)$中，n/p项在执行时间中开始对$\log p$项占支配位置。无论从观察中还是从直观分析中都能得出这样的结论。

309

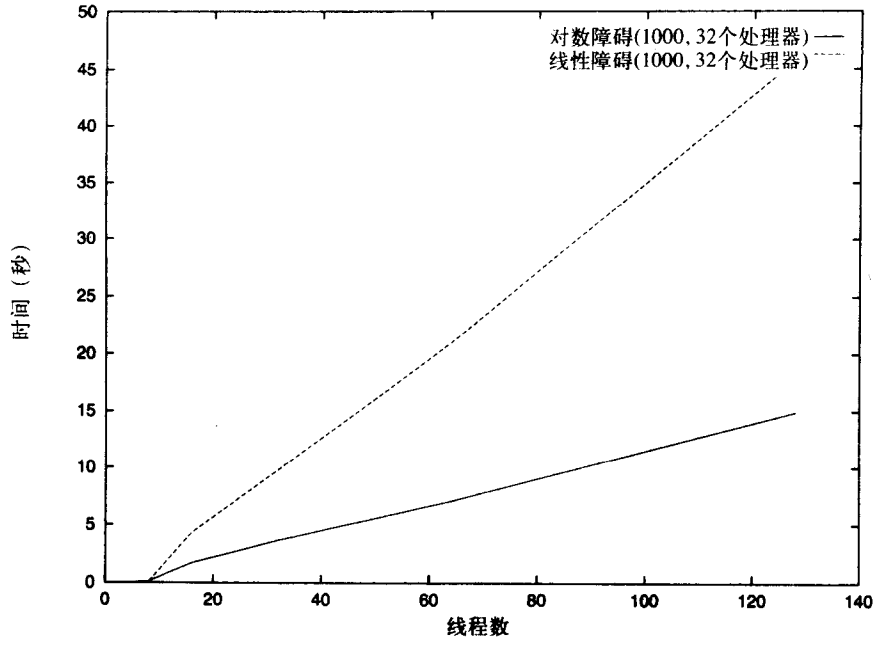


图7-3 在32个处理器的SGI Origin 2000上，1000个线性障碍和对数障碍的运行时间与线程数目的函数关系

7.9 设计异步程序的技巧

设计多线程应用程序时，一定要记住，不能假设有关其他线程的任何执行顺序。任何这种顺序都必须用前面讨论过的同步机制显式地建立：互斥锁、条件变量以及连接。另外，系统也可以提供其他同步手段。但是，由于可移植性的原因，不提倡使用其他形式的同步。

在许多线程库中，线程在半确定性的时间间隔切换，这样的库在程序中更能原谅同步错误。这种库称为*轻微异步*（slightly asynchronous）库。另一方面，内核线程（内核支持的线程）以及在多个处理器上调度的线程却较少宽容错误。因此，关于线程库中异步的等级，程序员一定不能做任何假设。

下面看看由于不正确地假设线程的相对执行时间而导致的常见错误：

- 比方说，线程T1创建另一个线程T2，T2需要从T1得到一些数据。这种数据通过全局内存单元传送。然而，线程T1在创建T2后才把数据放到指定单元。此处隐含的假设为：T1在阻塞前不会被切换；或者T2只在T1存放完数据后才访问数据单元。由于T1在创建T2后可能立刻被切换，这种假设就可能导致错误。在这种情形下，T1将接收未被初始化的数据。
- 假设和前一种情况一样，线程T1创建线程T2，然后需要传送数据给驻留堆栈中的线程T2。它通过传递一个指向线程T2在堆栈位置的指针传送数据。假设T1在T2被调度前运行结束。此时，栈帧已被释放，而一些其他的线程可能覆盖原先由栈帧指向的空间。在这种情况下，线程T2从单元读出的数据可能是无效的。同样的问题也会发生在全局变量上。
- 强烈建议不要使用调度技术作为同步的手段。在并行计算机中，要记录调度决策是极其

310

困难的。而且，当处理器数目改变时，根据线程的调度策略，这些问题可能会改变。事实上，较低优先级的线程运行时，较高优先级的线程可能在等待。

下面推荐有助于减少线程化程序错误的一些经验方法。

- 在实际创建线程前，将线程的所有需求设置好。这包括初始化数据、设置线程属性、线程优先级以及互斥锁属性等。一旦创建一个线程，在创建线程重新调度前，新创建的线程可能实际运行结束。
- 当两个线程间的某些数据项存在生产者-消费者关系时，一定要保证生产者线程在数据被消费前存放数据，且保证中间缓冲区不溢出。
- 在消费者端，要保证数据能保留到所有潜在的消费者都消费了数据。这一点特别与堆栈变量有关。
- 在可能的情况下，定义和使用组同步以及数据复制。这样能显著地提升程序性能。

虽然上面这些简单技巧为编写无错误的线程程序提供了准则，在实际编程时还是要特别小心，以避免与同步有关的竞赛条件以及并行开销。

7.10 OpenMP：基于命令的并行编程标准

本章的第一部分讲述了使用线程API在共享地址空间计算机上编程。虽然对这些API的标 311
准化及支持已有了很长的时间，但使用者主要还是系统程序员而不是应用程序员。其原因之一是，像Pthreads这样的API被认为是低层原语。依照常识，许多应用程序可由较高层的结构（或命令）有效地支持，程序员不用对线程进行操作。这种基于命令的语言已存在很长时间，但直到最近才以OpenMP的形式完成了标准化。OpenMP是一种可以用FORTRAN、C以及C++在共享地址空间计算机上进行编程的API。OpenMP命令提供对并发、同步以及数据处理的支持，但不需要显式设置互斥锁、条件变量、数据范围以及初始化。在本章其余部分将使用OpenMP C API。

7.10.1 OpenMP编程模型

下面先用一个简单的程序来开始讲述OpenMP编程模型。C语言及C++中的OpenMP命令基于#pragma编译器命令，命令本身由命令名及后面的子句构成。

```
#pragma omp directive [clause list]
```

除非遇到parallel命令，否则OpenMP程序串行执行。parallel命令用来创建一组线程。线程的确切数目可由命令指定，用一个环境变量指定，或者在运行时用OpenMP函数指定。遇到parallel命令的主线程变为这一组线程的主（master）线程。在这一组中，它被分配的线程id为0。parallel命令的原型如下：

```
1  #pragma omp parallel [clause list]
2  /* structured block */
3
```

每个由这条命令创建的线程执行由parallel命令指定的structured block。子句列表用来指定条件并行、线程数目以及数据处理。

- **条件并行**：子句if（标量表达式）决定并行结构是否将导致创建线程。一条parallel命令只能使用一个if子句。

• **并发度**：子句num_threads（整数表达式）指定由parallel命令创建的线程数目。

• **数据处理**：子句private（变量表）表示指定的变量集对每个线程都是本地的——即
312 每个线程对表中的每个变量都有一个副本。子句firstprivate（变量表）与子句private类似，除了进入线程变量的值要在parallel命令前初始化成相应的值以外。子句shared（变量表）表示表中的所有变量对于所有线程都是共享的，即只有一个副本，当用线程处理这些变量以保证串行性时，一定要特别小心。

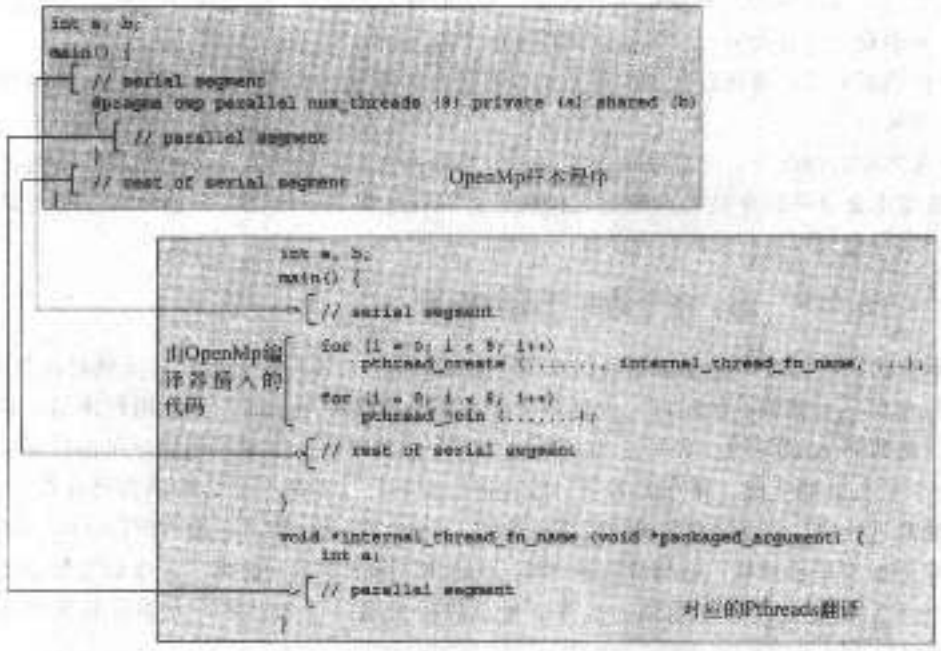


图7-4 OpenMP程序样本以及可能由OpenMP编译器执行的该程序的Pthreads 翻译

观察对应的Pthreads 翻译内容，就很容易理解OpenMP的并发模型。图7-4中列出一种可能的从OpenMP程序到Pthreads程序的翻译，感兴趣的读者可能注意到，这种翻译可以轻易地通过Yacc或CUP脚本自动实现。

例7.9 使用parallel命令

```
1  #pragma omp parallel if (is_parallel == 1) num_threads(8) \
2                       private (a) shared (b) firstprivate(c)
3  {
4      /* structured block */
313 5  }
```

程序中，如果变量is_parallel的值等于1，就创建8个线程，每个线程得到变量a和变量c的私有副本，并共享变量b的值。此外，c的每个副本的值用parallel命令之前的c的值进行初始化。 ■

变量的默认状态由子句default (shared) 或default (none) 指定。子句default (shared) 表明，在默认情况下，一个变量被所有的线程共享。子句default

(none) 表明，线程中用到的每个变量的状态必须显式指定。通常要防止由无意的对共享数据的并发存取引起的错误。

像子句firstprivate指定变量的多个本地副本如何在线程中初始化一样，当多个线程退出时，子句reduction指定在不同线程中的变量的多个副本如何在主线程中组合成一个副本。子句reduction的用法为reduction (操作符: 变量表)。这个子句用操作符对变量表中指定的标量变量进行归约。表中的变量被隐式指定为对线程私有。操作符可以是+, *, -, &, |, ^, &&或||。

例7.10 使用reduction子句

```
1    #pragma omp parallel reduction(+: sum) num_threads(8)
2    {
3        /* compute local sums here */
4    }
5    /* sum here contains sum of all local instances of sums */
```

本例中，8个线程中的每一个都获得变量sum的一个副本。当多个线程退出时，所有本地副本的和存储在变量的一个单一副本中（在主线程）。 ■

除了上面讲的数据处理子句外，还有另一个子句copyin。在我们讨论数据范围后，将在7.10.4节详细地讲述copyin子句。

至此，我们能用parallel命令及多种子句编写第一个OpenMP程序。为便于介绍我们引入两个函数：函数omp_get_num_threads()返回并行范围内的线程数目，函数omp_get_thread_num()返回每个线程的整数id（回想主线程的id为0）。

例7.11 用OpenMP命令计算PI

我们第一个OpenMP程序仿照例7.2，用一个Pthreads程序计算π。并行命令指定除 314
npoints以外的所有变量都是本地的，npoints为穿过所有线程的二维空间中的随机点的总数。此外，命令指定有8个线程，所有线程执行完毕后sum的值是每个线程中的本地值之和。函数omp_get_num_threads用来确定线程的总数。和例7.2一样，一个for循环用来产生所需数目的随机点（在二维空间），并确定有多少个点位于指定的单位直径的圆内。

```
1  /* ****************************************************
2     An OpenMP version of a threaded program to compute PI.
3     **************************************************** */
4
5      #pragma omp parallel default(private) shared (npoints) \
6                           reduction(+: sum) num_threads(8)
7      {
8        num_threads = omp_get_num_threads();
9        sample_points_per_thread = npoints / num_threads;
10       sum = 0;
11       for (i = 0; i < sample_points_per_thread; i++) {
12         rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
13         rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
14         if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
15              (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
16              sum ++;
17       }
18     }
```

与相应的POSIX线程程序相比，该程序在指定创建和终止线程方面更容易编写。

7.10.2 在OpenMP中指定并发任务

命令parallel可以和其他命令一起用于指定通过迭代和任务的并发。OpenMP提供两条命令——for和sections——用来指定并发的迭代和任务。

1. for命令

for命令用来在多个线程间分割并行迭代空间。for命令的一般形式如下:

```
1       #pragma omp for [clause list]
2           /* for loop */
3
```

这种情况下可用的子句有private、firstprivate、lastprivate、reduction、
315 schedule、nowait和ordered。前4个子句用来处理数据，这4个子句的语义和parallel命令中一样。子句lastprivate用来处理变量的多个本地副本在并行for循环结束时如何写回到单一副本中。当使用for循环（或者将会见到的sections命令）将工作分配给线程时，有时候需要在for循环的最后一次迭代（与串行执行定义的一样）中更新变量的值。这一点是通过lastprivate命令实现的。

例7.12 用for命令计算π

回顾例7.11，for循环的每次迭代都是独立的，由此可以并发地执行。在这种情况下，可以用for命令简化程序。修改后的代码段如下:

```
1     #pragma omp parallel default(private) shared (npoints) \
2                          reduction(+: sum) num_threads(8)
3     {
4       sum = 0;
5       #pragma omp for
6       for (i = 0; i < npoints; i++) {
7         rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
8         rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
9         if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
10            (rand_no_y - 0.5) * (rand_no_y - 0.5)) <= 0.25)
11            sum ++;
12      }
13    }
```

这个例子中的for命令用来指定后面紧跟着的for循环必须并行执行，即在多个线程间分割。请注意，与例7.11中的sample_points_per_thread不同，本例中的循环索引从0到npoints。默认情况下，for命令的循环索引假设为私有。注意这段OpenMP代码与对应的串行代码间只有两条命令的区别。本例说明，将许多串行程序转换为基于OpenMP的线程程序非常简单。 ■

2. 将迭代分配给线程

for命令的schedule子句用来将迭代分配给线程。schedule命令的一般形式为schedule(scheduling_class[,parameter])。OpenMP支持4种调度类: static、dynamic、guided和runtime。

例7.13 OpenMP中的调度类——矩阵相乘

316 本例用稠密矩阵相乘来研究不同的调度类。将矩阵a与矩阵b相乘得到矩阵c的代码如下:

```
1       for (i = 0; i < dim; i++) {
2           for (j = 0; j < dim; j++) {
3               c(i,j) = 0;
4               for (k = 0; k < dim; k++) {
5                   c(i,j) += a(i, k) * b(k, j);
6               }
7           }
8       }
```

上面的代码段指定一个三维迭代空间，为学习OpenMP中不同的调度类提供了理想的例子。 ■

static调度 static调度类的一般形式为schedule(static[,chunk-size])。它将迭代空间分割成大小为chunk-size的相等的块，并将这些块以循环的方式分配给线程。如果没有指定chunk-size，迭代空间就分成与线程数目一样多的块，每个块分配给每个线程。

例7.14 矩阵相乘的循环static调度

下面修改后的矩阵相乘程序使最外面的迭代在多个线程间静态分割，如图7-5a所示。

```
1   #pragma omp parallel default(private) shared (a, b, c, dim) \
2                        num_threads(4)
3       #pragma omp for schedule(static)
4       for (i = 0; i < dim; i++) {
5           for (j = 0; j < dim; j++) {
6               c(i,j) = 0;
7               for (k = 0; k < dim; k++) {
8                   c(i,j) += a(i, k) * b(k, j);
9               }
10          }
11      }
```

由于一共有4个线程，且没有指定块的大小，如果dim = 128，则每个分区的大小为32列。使用schedule(static, 16)将导致如图7-5b所示的迭代空间划分。若例7.13中程序的每个for循环在多线程间用schedule(static)并行化，且允许嵌套并行，则划分的结果如图7-5c所示，这是空间分割的另一个例子（参见7.10.6节）。 ■

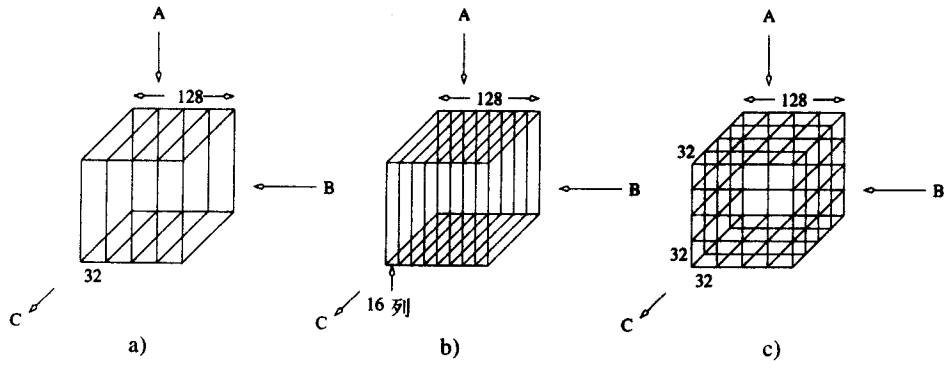


图7-5 使用OpenMP的static调度类的3种不同调度

dynamic调度 通常，由于许多原因，从不同类的计算机资源到非均匀处理器负载，同等划分的工作负载的执行时间有很大的差别。因此，OpenMP中有一个dynamic调度类，这个类的一般形式为schedule(dynamic[,chunk-size])。迭代空间被划分成chunk-size

317

大小的块。但是，这些块只在线程变为空闲时才分配给它们。这样就能考虑到由static调度造成的瞬时不平衡。如果没有指定chunk-size，则默认为每个块一个迭代。

guided调度 如果迭代空间中有100次迭代，块大小为5，则迭代空间将被分成20块。如果有16个线程，在最佳情况下，12个线程的每个线程得到1块，其余4个线程的每个线程得到2块。因此，如果处理器与线程的数量一样多，这样的配送就会导致可观的空闲。这个问题（也称为*边缘效应*（edge effect））的解决办法是在计算的进行过程中减少块的大小。这是guided调度类的原则。这个类的一般形式为schedule(guided[,chunk-size])。在这个类中，当每个块分配给每个线程时，块的大小按指数缩小。chunk-size为应分配的最小块的大小。因此，当剩余的迭代数目小于chunk-size时，所有的迭代将一次分配完。如果chunk-size的大小没有指定，则默认为1。

运行时间 通常需要将调度决策延迟到运行时间。例如，如果想看到不同的调度策略的影响，然后再选出最佳的一种，就可以将调度设置为runtime。在这种情况下，由环境变量OMP_SCHEDULE决定调度类以及块的大小。

当不用omp for命令指定调度类时，实际的调度方法就没有指定，而要依靠具体的实现。for命令对跟在它后面的for循环设置另外的限制。例如，循环中不能有中断语句，循环控制
318 变量必须是整数，for循环的设定初值的表达式必须是一个整数，逻辑表达式必须是<、≤、>或者≥，增量表达式中必须只有整数增加量或减少量。读者可以参考OpenMP手册了解关于这些限制的详细情况。

3. 在多个for命令间同步

通常，并行结构中需要有一系列的for命令，而且这个并行结构在每个for命令结束时也不执行隐式障碍。OpenMP提供一个子句nowait，它可以和for命令一起使用，用来表明线程可以进入下一个语句，无需等待其他所有的线程结束for循环执行。下面的例子说明这个子句的用法。

例7.15 使用nowait子句

这个例子中，变量name需要在两个列表——current_list和past_list中查找。如果该变量名存在于一个列表中，则必须被相应地处理。变量名也可能在两个列表中。此时，没有必要等待所有的线程在进入第二个循环前完成第一个循环的执行。因此，可以使用nowait子句节省空闲及同步开销，程序代码如下：

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i = 0; i < nmax; i++)
            if (isEqual(name, current_list[i])
                processCurrentName(name);
    #pragma omp for
        for (i = 0; i < mmax; i++)
            if (isEqual(name, past_list[i])
                processPastName(name);
}
```

■

4. sections命令

for命令适用于划分跨越多个线程的迭代空间。现在考虑有3个任务（taskA，taskB，taskC）需要执行。假设这3个任务彼此独立，可以分配给不同的线程。在OpenMP中，可用

sections命令进行这种非循环的并行任务分配。sections命令的一般形式如下:

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```

319

sections命令将与每个段对应的结构化块分配给一个线程（事实上可将一个以上的段分配给一个线程)。clause list中可能会包括如下子句——private、firstprivate、lastprivate、reduction以及nowait，这些子句的语法和语义与for命令中的一样。本例中，子句lastprivate指定更新变量值的sections命令的最后段（词法上的）；在sections命令的最后，子句nowait指定所有线程没有隐式同步。

为了执行3个并发任务taskA，taskB，taskC，相应的sections命令如下:

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

如果有3个线程，每个段（此时即为相应的任务）分配给一个线程。线程在分配段执行结束时同步（除非使用了nowait子句)。注意，分支进入或分支跳出section块都是不合法的。

5. 合并命令

到目前为止，讨论了用命令parallel创建并发的线程，用for命令和sections命令将工作分配给线程。如果没有指定parallel命令，则for命令和sections命令将串行执行(所有的工作都分配给一个线程，即主线程)。因此，parallel命令通常在for命令和sections命令之前。OpenMP允许程序员把parallel命令分别合并成parallel for和parallel sections命令。合并后命令的子句列表可以是parallel命令的子句列表，也可以是for/sections命令的子句列表。例如:

320

```
#pragma omp parallel default (private) shared (n)
{
    #pragma omp for
    for (i = 0 < i < n; i++) {
    /* body of parallel for loop */
    }
}
```

与

```
#pragma omp parallel for default (private) shared (n)
{
    for (i = 0 < i < n; i++) {
    /* body of parallel for loop */
    }
}
```

相同，同时

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
           taskA();
        }
        #pragma omp section
        {
           taskB();
        }
        /* other sections here */
    }
}
```

等同于:

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        taskA();
    }
    #pragma omp section
    {
        taskB();
    }
    /* other sections here */
}
```

6. 嵌套parallel命令

321 回到程序7-13。要在多个线程间分割每个for循环，必须对程序作如下修改:

```
#pragma omp parallel for default(private) shared (a, b, c, dim) \
                num_threads(2)
    for (i = 0; i < dim; i++) {
    #pragma omp parallel for default(private) shared (a, b, c, dim) \
                num_threads(2)
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            #pragma omp parallel for default(private) \
                    shared (a, b, c, dim) num_threads(2)
            for (k = 0; k < dim; k++) {
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

先来看看这段代码是如何编写的。它没有将3个for命令嵌套到一个parallel命令中，而是使用了3个parallel for命令。这是因为OpenMP不允许for、sections以及

single命令绑定到同一个被嵌套的parallel命令。进而，遇到嵌套的parallel命令时，这段代码只会产生一队逻辑线程。新产生的这队逻辑线程仍然由与外部parallel命令对应的同一线程执行。为了产生一组新的线程，必须用OMP_NESTED环境变量来启用嵌套并行。如果OMP_NESTED环境变量被设置为FALSE，那么内部的parallel区域将串行化，并由一个线程执行。如果OMP_NESTED环境变量被设置为TRUE，则嵌套并行被启用。环境变量的默认状态为FALSE，即嵌套并行被禁用。OpenMP环境变量在7.10.6节详细地讲述。

在嵌套并行中，还有许多与同步结构相关的限制。关于这些限制的详细情况，读者可以查阅OpenMP手册。

7.10.3 OpenMP中的同步结构

在7.5节，我们讨论了需要对多个线程的执行进行协调。这可能是由所需要的执行顺序、指令集的原子性或者代码段的串行执行需要造成的。Pthreads API支持互斥锁和条件变量。使用它们可以通过读-写锁、障碍以及监控器等形式实现更高层次的功能。OpenMP标准以易于使用的API提供这种高层的功能。本节将考查这些命令以及它们的使用。

1. 同步点：barrier命令

barrier是最常用的同步原始变量。OpenMP提供了barrier命令，它的语法如下： 322

```
1       #pragma omp barrier
```

遇到此命令时，队中所有的线程都要等待其他的线程跟上，然后释放。当与嵌套的parallel命令一起使用时，barrier命令绑定到最近的parallel命令。对于按条件执行障碍，一定要注意，barrier命令必须包含在条件执行的复合语句中。这是因为pragma是编译器命令，不是语言的一部分。障碍也能通过结束及重新开始parallel区域实现。但是，与此相关的开销通常都会更高。因此，这不是实现障碍的可选方法。

2. 单线程执行：single和master命令

一个并行段内的计算通常只需由一个线程执行。计算一个数字列表的平均值便是这样一个简单例子。每个线程计算部分列表中的本地和，并把这些本地和加到一个共享的全局和中，然后，让一个线程以全局和除以列表中的项目数得到平均值。最后一步可以用single命令实现。

single命令指定一个由单一（任意的）线程执行的结构化块。命令的语法如下：

```
1   #pragma omp single [clause list]
2           structured block
```

子句表(clause list)可以是子句private、firstprivate以及nowait。这些子句的语义和前面的一样。遇到single块时，第一个线程进入块。所有其他的线程进行到块的末尾。如果在块的尾部指定了nowait子句，则其他的线程继续；否则这些线程在single块的尾部等待，直至线程完成块的执行。这条命令在计算全局数据以及进行I/O操作时很有用。

master命令是single命令的特殊情况，它指定只能由主线程执行结构化块。master命令的语法如下：

```
1   #pragma omp master
2           structured block
```

与single命令相反，不存在伴随master命令的隐式障碍。

3. 临界段：critical和atomic命令

在讨论Pthreads时，曾经考查用锁来保护临界区域，即必须串行地一次执行一个线程的区域。除了显式锁管理（7.10.5节）外，OpenMP提供critical命令实现临界区域。
323 critical命令的语法如下：

```
1  #pragma omp critical [(name)]
2           structured block
```

这里，可选标识符name用来标识临界区域。使用name可以使不同的线程执行不同的代码，且相互保护。

例7.16 在生产者-消费者线程使用critical命令

在生产者-消费者模式中，生产者线程产生一个任务，并将它插入到任务队列。消费者线程从队列中提取任务，并一次执行一个任务。由于存在对任务队列的并发访问，这些访问必须用临界块串行化。特别地，将任务插入到队列以及从队列中提取任务必须被串行化。这可用下述方法实现：

```
1       #pragma omp parallel sections
2       {
3           #pragma parallel section
4           {
5               /* producer thread */
6               task = produce_task();
7               #pragma omp critical ( task_queue)
8               {
9                   insert_into_queue(task);
10              }
11          }
12          #pragma parallel section
13          {
14              /* consumer thread */
15              #pragma omp critical ( task_queue)
16              {
17                  task = extract_from_queue(task);
18              }
19              consume_task(task);
20          }
21      }
```

值得注意的是，队列满和队列空的条件必须在函数insert_into_queue和extract_from_queue中显式处理。 ■

critical命令保证在程序执行的任意一点处，只有一个线程在由一个名称指定的临界段内部。如果某一线程已经在临界段内部（已命名），所有其他的线程必须等待该线程完毕才能进入命名的临界段。名称字段为可选项。如果没有指定名称，则临界段映射到一个默认名
324 称，这个名称与所有未命名的临界段的名称相同。在整个程序中，临界段的名称都是全局的。

容易看出，命令critical是Pthreads中相应的mutex函数的直接应用。名称字段映射到进行锁操作的互斥锁名称。和Pthreads中一样，一定要记住，critical段表示程序中串行化的点，因此，一定要尽可能减小临界段的大小（在执行时间方面），以求获得好的性能。

使用critical命令时一定要注意几个明显的安全措施。指令的block必须表示结构化

块，即不允许跳入块或从块中跳出。容易看出，跳入块将导致非临界访问，而跳出块将导致未释放的锁，引起线程无限等待。

通常，临界段只由对某单个内存单元的一个更新组成，例如，整数相加或者增加一个增量。OpenMP提供另一个命令atomic，用来实现对内存单元的原子更新。atomic命令指定，对随后指令中内存单元的更新一定应作为原子操作进行。更新指令可以是下面几种形式之一：

```
1  x binary_operation = expr
2  x++
3  ++x
4  x--
5  --x
```

这里，expr是一个标量表达式，它不包括对x的引用，x本身是标量类型的左值，binary_operation是{+，*，-，/，&，||，《,》}之一。应该记住，命令atomic只对载入及存储标量变量原子化。表达式的计值不是原子的。要非常小心地避免竞赛条件隐含其中，这也可以解释为什么atomic命令中的expr项不能包含更新的变量自身。所有的atomic命令都可以被critical命令代替，只要它们的名称相同。但是，与转换成critical命令相比，原子化硬件指令的实用性可以优化程序的性能。

4. 按顺序执行：ordered命令

在许多情况下，需要像执行串行版本程序的顺序一样执行并行循环的代码段。例如，考虑一个for循环，于循环中的某点，计算一个存储在数组list中的列表与数组cumul_sum元素的累积和。数组cumul_sum的计算可在for循环中用下标i串行地执行cumul_sum[i]= cumul_sum[i-1]+list[i]得到。当在多个线程间执行这个for循环时，应该注意必须等到cumul_sum[i-1]的值计算后，才能计算cumul_sum[i]的值。所以，语句一定要在一个ordered块中执行。

ordered命令的语法如下：

325

```
1  #pragma omp ordered
2      structured block
```

由于ordered命令代表for循环的按序执行，它必须位于for或parallel for命令的作用域内。进而，for或parallel for命令中必须含有ordered子句，用来表示在循环中包含ordered块。

例7.17 用ordered命令计算列表的累积和

前面已经讲过，要计算列表中i个数的累积和，可以把当前数加到列表中i-1个数的累积和上。然而，这个循环必须按顺序执行。而且，第一个元素的累积和就是这个元素本身。因此，可用ordered命令编写代码如下：

```
1       cumul_sum[0] = list[0];
2       #pragma omp parallel for private (i) \
3                   shared (cumul_sum, list, n) ordered
4       for (i = 1; i < n; i++)
5       {
6           /* other processing on list[i] if needed */
7
8           #pragma omp ordered
9           {
10              cumul_sum[i] = cumul_sum[i-1] + list[i];
11          }
12      }
```

■

需要特别注意的是，ordered命令代表程序中一个有序的串行化点。当所有先前的线程（由循环索引确定）退出后，只有一个线程能进入一个有序块。因此，如果循环的大部分包含在ordered命令中，则相应的加速比就不能达到。在上例中，除非在ordered命令之外有一个重要的处理与list[i]有关，并行形式不会比串行形式快。限制一个for命令中只能有一个ordered块。

5. 内存一致性：flush命令

flush命令提供在线程间实现内存一致的机制。在共享地址空间计算机中，此命令看起来有点多余，但要注意，变量经常被分配给寄存器，而寄存器分配的变量可能会不一致。这
326 时，通过强制变量写入内存系统或将变量从内存系统读出，flush命令提供一个内存栅栏。所有对共享变量的写操作必须在一个flush中提交到内存，所有在栅栏后对共享变量的引用必须从内存实现。由于私有变量只和一个线程相关，flush命令只应用于共享变量。

flush命令的语法如下：

```
1  #pragma omp flush[(list)]
```

可选项list指定需要被刷新的变量，默认情况下，所有的共享变量都被刷新。

几个OpenMP命令中有隐式flush，特别是flush隐含在barrier中，在critical、ordered、parallel for以及parallel sections块的入口和出口处，在for、sections以及single块的出口处。如果有nowait子句，则不会隐含flush。在for、sections以及single块的入口处，以及master块的入口或出口处，也没有隐含flush。

7.10.4 OpenMP中的数据处理

线程对数据的处理是影响程序性能的重要因素之一。前面已经简单地讲述了OpenMP支持的多种数据类，如private、shared、firstprivate以及lastprivate。现在再对它们进行更详细地讨论，了解如何使用这些数据类。下面我们给出一些实用的建议：

- 如果一个线程初始化并使用了一个变量（如循环索引），且没有其他的线程存取数据，那么应为线程制作一个变量的本地副本，数据也应该指定为private。
- 如果一个线程重复地读取一个变量，且此变量在程序的早期已被初始化，那么，对变量作一个副本并继承线程创建时变量的值是有利的。这样，当线程在处理器中调度时，数据可以驻留在同一处理器中（可能在它的高速缓存中），对数据的存取不会造成处理器间的通信。这样的数据应该指定为firstprivate。
- 如果多个线程处理一块数据，则必须寻求方法将这些处理分成多个本地操作，再加上一个全局操作。例如，如果多个线程记录着某个事件的计数，则最好保留本地计数，并在并行块的最后用求和操作让计数增加。子句reduction支持这样的操作。
- 如果多个线程处理一个大数据结构的不同部分，则程序员应寻求方法将大数据结构分成
327 小数据结构，并使这些小数据结构对于处理它们的线程是私有的。
- 所有上面的方法都试过后，用子句shared可使剩余的数据项被多个线程共享。

除了private、shared、firstprivate以及lastprivate外，OpenMP还支持另一个数据类，称为threadprivate。

threadprivate和copyin命令 在保持线程数目不变条件下，一组对象通过并行或串行块时以持久的方式使其对一个线程本地可用，通常这样做是有用的。与private变量相反，这

些变量对于通过并行区域保持不变的对象是有用的，否则这些对象就必须复制到主线程的数据空间中，并在下一个并行块重新初始化。该类变量在OpenMP中用threadprivate命令支持。命令的语法如下:

```
1  #pragma omp threadprivate(variable_list)
```

这条命令隐含在variable_list中的所有变量对每个线程而言都是本地的，且在一个并行区域内被访问前初始化一次。此外，如果线程数目的动态调整被禁止和线程数目不变，则这些变量在不同的并行区域中能够保持不变。

与firstprivate相似，OpenMP提供一种机制，用于将同样的值分配给一个并行区域中所有线程的threadprivate变量。这就是copyin子句，它能和parallel命令一起使用，语法为copyin(variable_list)。

7.10.5 OpenMP库函数

除了命令以外，OpenMP还提供许多函数，使程序员能够控制线程程序的执行。可以看出，这些函数和相应的Pthreads函数类似；但是，通常这些函数处在更高的抽象层次，更易于使用。

1. 控制线程和处理器数目

下面的OpenMP函数与线程程序中用到的并发性和处理器数目有关:

```
1  #include <omp.h>
2
3  void omp_set_num_threads (int num_threads);
4  int omp_get_num_threads ();
5  int omp_get_max_threads ();
6  int omp_get_thread_num ();
7  int omp_get_num_procs ();
8  int omp_in_parallel();
```

328

函数omp_set_num_threads设置默认的线程数目，如果parallel命令中没有使用num_threads子句，那么这些线程在遇到下一个parallel命令时创建。这个函数必须在并行区域的范围外调用，且必须启用对线程的动态调整（使用7.10.6节中讲述的OMP_DYNAMIC环境变量或者omp_set_dynamic库函数）。

函数omp_get_num_threads返回参与到队中的线程数目。它绑定到最靠近它的并行命令上，在没有并行命令时，返回1（指主线程）。函数omp_get_max_threads返回可能由遇到的parallel命令创建的最大线程数目，它没有num_threads子句。函数omp_get_thread_num对一个队中的每个线程返回一个唯一的线程id。此id取从0（指主线程）到omp_get_num_threads()-1的整数值。函数omp_get_num_procs返回在那一点可用来执行线程程序的处理器数目。最后，函数omp_in_parallel对于从并行区域内的调用返回一个非0值，从并行区域外的调用返回0。

2. 控制和监控线程的创建

下面的OpenMP函数使程序员能够设置和监控线程的创建:

```
1  #include <omp.h>
2
3  void omp_set_dynamic (int dynamic_threads);
```

```
4  int omp_get_dynamic ();
5  void omp_set_nested (int nested);
6  int omp_get_nested ();
```

函数omp_set_dynamic允许程序员动态地改变在遇到并行区域时创建的线程数目。如果dynamic_threads的计值为0，则停止动态调整，否则启用动态调整。函数必须在并行区域外调用。相应的状态，即动态调整被启用或停止，可用函数omp_get_dynamic查询，它在动态调整启用时返回一个非0值，没有启用时返回0。

函数omp_set_nested在其参数nested为非0值时允许嵌套并行，在参数nested为0时停止嵌套并行。当嵌套并行停止时，所有随后遇到的嵌套并行区域都被串行化。嵌套并行的状态可用函数omp_get_nested查询，它在嵌套并行被启用时返回一个非0值，停止使用时返回0。

3. 互斥

329 OpenMP提供对临界段和原子更新的支持，但有时使用一个显式锁更方便。对于这种程序，OpenMP提供初始化锁、锁定、解锁以及放弃锁的函数。在OpenMP中，锁数据结构的类型为omp_lock_t。OpenMP定义以下函数:

```
1  #include <omp.h>
2
3  void omp_init_lock (omp_lock_t *lock);
4  void omp_destroy_lock (omp_lock_t *lock);
5  void omp_set_lock (omp_lock_t *lock);
6  void omp_unset_lock (omp_lock_t *lock);
7  int omp_test_lock (omp_lock_t *lock);
```

一定要先初始化锁，然后才能使用锁。锁的初始化使用函数omp_init_lock。当锁不再需要时，必须用函数omp_destroy_lock放弃。对已被初始化的锁进行初始化，或放弃未初始化的锁，都是不合法的。一旦锁被初始化，它就可以用函数omp_set_lock锁定和函数omp_unset_lock解锁。锁定一个先前被解锁的锁时，线程获得对锁的独占访问。所有其他的线程试图用omp_set_lock锁定时都必须等待这个锁。只有拥有锁的线程可以对其解锁。如果线程试图对由其他线程拥有的锁进行解锁，则将得到不确定的结果。锁定和解锁操作先于初始化锁或在放弃锁后都不合法。函数omp_test_lock用来测试锁是否已被设置，如果函数返回一个非0值，则锁已被成功设置，否则锁当前被另一个线程拥有。

与Pthreads中的递归互斥锁一样，OpenMP也支持可嵌套锁，同一线程对可嵌套锁可以锁定多次。这种情况下，锁对象是omp_nest_lock_t，处理嵌套锁的相应函数为:

```
1  #include <omp.h>
2
3  void omp_init_nest_lock (omp_nest_lock_t *lock);
4  void omp_destroy_nest_lock (omp_nest_lock_t *lock);
5  void omp_set_nest_lock (omp_nest_lock_t *lock);
6  void omp_unset_nest_lock (omp_nest_lock_t *lock);
7  int omp_test_nest_lock (omp_nest_lock_t *lock);
```

这些函数的语义与对应的简单锁函数类似。注意，所有这些函数都在Pthreads中有直接对应的互斥锁调用。

7.10.6 OpenMP中的环境变量

OpenMP提供额外的环境变量帮助控制并行程序的执行。OpenMP有如下的环境变量。

OMP_NUM_THREADS 这个环境变量指定进入parallel区域时创建线程的默认数目。线程的数目既可用函数omp_set_num_threads改变，也可以用parallel命令的 330
num_threads子句改变。注意只有当变量OMP_SET_DYNAMIC设置为TRUE或用非0参数调用函数omp_set_dynamic时，才能动态地改变线程数目。例如，如果程序执行前在csh中输入以下命令，那么默认的线程数目设置为8，

```
1  setenv OMP_NUM_THREADS 8
```

OMP_DYNAMIC 当此变量设置为TRUE时，允许线程的数目在运行时用函数omp_set_num_threads或num_threads子句进行控制。如果用参数0调用函数omp_set_dynamic，则对线程数目的动态控制被停止。

OMP_NESTED 当这个变量设置为TRUE时，启用嵌套并行，除非用参数0调用函数omp_set_nested停用嵌套并行。

OMP_SCHEDULE 这个变量控制与使用runtime调度类的for命令有关的迭代空间的指定。变量的取值可以是static、dynamic以及guided，再加上可选的块大小。例如，

```
1  setenv OMP_SCHEDULE "static,4"
```

指定在默认情况下，所有的for命令使用static调度，块大小为4。指定的其他例子包括：

```
1  setenv OMP_SCHEDULE "dynamic"
2  setenv OMP_SCHEDULE "guided"
```

在这两个例子中，使用的默认块大小都是1。

7.10.7 显式线程与基于OpenMP编程的比较

OpenMP在所有本地线程的顶部提供一个层，使得更容易执行许多与线程有关的任务。使用由OpenMP提供的命令，程序员无需再承担初始化属性对象、设置线程参数、划分迭代空间等工作。当要解决的问题具有静态的和/或正则的任务图时，这种便利尤其有用。在许多应用程序中，与从命令自动产生线程代码有关的开销是最小的。

然而，使用命令也有缺点。用显式线程编写的程序将使数据交换更透明，这有助于减少一些与数据移动、假共享以及争用相关的开销。显式线程还提供更丰富的API，这些API的形式为条件等待、不同类型的锁以及对构建复合同步操作时的更大灵活性（如7.8节所述）。最后，由于显式线程比OpenMP使用更广泛，更容易找到Pthreads程序的工具和对它的支持。

程序员在决定使用何种API编程前，一定要权衡所有这些利弊。 331

7.11 书目评注

无论是显式的基于线程的编程，还是基于OpenMP的编程，都有许多优秀的参考文献。Lewis和Berg[LB97, LB95a]提供详细的Pthreads编程指南。Kleiman, Shah和Smaalders[KSS95]对线程系统及用线程编程提供出色的描述。其他几本书籍也讲述了与多线程编程有关的编程及系统软件问题[NBF96, But97, Gal95, Lew91, RRRR96, ND96]。

人们也开发了许多其他的线程API及系统，它们经常被用于各种应用程序中。这些系统包括：Java线程[Dra96, MK99, Hyd99, Lea99]、微软的线程API[PG98, CWP98, Wil00, BW97]以及Solaris线程API[KSS95, Sun95]。无论从硬件还是从软件来看，对线程系统的研究都有着很

长的和富有成果的历史，可以追溯到HEP Denelcor[HLM84]时期。最近，又开发了一些软件系统，如Cilk[BJK+95, LRZ95]，OxfordBSP[HDM97]，Active Threads[Wei97]以及Earth Manna[HMT+96]。多个系统提供对多线程的硬件支持，其中包括MIT Alewife[ADJ+91]，Horizon[KS88]，同时多线程[TEL95, Tul96]，多标量体系结构[Fra93]以及超线程体系结构[TY96]。

人们也研究了关于线程性能方面的问题。早期关于多线程处理器性能权衡方面的工作在[Aga89, SBCV90, Aga91, CGL92, LB95b]中已有介绍。人们也广泛地研究了共享内存的一致性模型。其他活跃的研究领域包括运行系统、编译器支持、基于对象的扩展、性能评估以及软件开发工具。人们还致力于在工作站网络上共享内存支持软件的研究。所有这些仅仅涉及有关用线程编程的问题。

由于出现时间不长，有关OpenMP编程的书籍相对较少[CDK+00]。在`http://www.openmp.org`可以找到OpenMP标准及扩展的资料集。许多其他的文章（以及特刊）讨论了有关OpenMP性能、编译以及互用性方面的问题[Bra97, CM98, DM98, LHZ98, Thr99]。

习题

7.1 估计下面执行Pthreads中的每一种操作花费的时间：

- 线程创建
- 线程连接
- 成功锁定
- 成功解锁
- 成功trylock（试探锁）
- 不成功trylock（试探锁）
- 条件等待
- 条件标记
- 条件广播

332

针对每个操作，仔细记录计算每种函数调用时间所使用的方法，同时记录所用的计算机。

7.2 使用多个线程插入到队列及从队列中提取多个线程，实现一个多存取线程队列。使用互斥锁同步对队列的存取。记录下1000次插入和1000次提取的时间，每次有64个插入线程（生产者）和64个提取线程（消费者）。

7.3 使用条件变量（除了互斥锁以外）重做习题7.2。记录下进行与上题同样试验花费的时间，并对时间的差别加以评论。

7.4 一个简单的流媒体播放器由以下部分组成：一个监视网络端口到达数据的线程，一个对数据包解压缩并产生图像序列中的帧的解压缩器线程，以及一个在规划的间隔显示帧的绘制线程。这3个线程必须通过共享缓冲实现通信——界于网络和解压缩器之间的输入缓冲，以及界于解压缩器与绘制器之间的输出缓冲。实现这个简单的线程框架。网络线程调用虚拟函数`listen_to_port`从网络收集数据。就此程序而言，这个函数产生所需长度的随机字节串。解压缩线程调用函数`decompress`，该函数从输入缓冲取得数据，并返回预先确定大小的帧。对于这个习题，产生一个具有随机字节数的帧。最后，绘制器线程从输出缓冲取出帧，

并调用显示函数。显示函数将一帧作为参数，对于这个习题，它没有操作。用条件变量实现这个线程框架。注意，很容易改变这3个虚拟函数，得到一个有意义的流媒体解压缩器。

7.5 请使用二叉树搜索算法展示递归锁的用法。程序要求一个很大的数字列表。列表在多个线程间划分。每个线程试图用与树对应的单个锁将它的元素插入到树中。说明即使在线程数目不多的情况下，单个锁也会成为瓶颈。

7.6 通过将锁与树中的每个节点对应（与单个锁和整个树对应相反），改进二叉树搜索程序。线程在读或写节点时锁定该节点。考查这种实现的性能特点。

7.7 用读-写锁进一步改善二叉树搜索程序。线程在读一个节点前对节点加上读出锁。线程只在需要写入树节点时才对节点加上写入锁。请实现该程序，并记录下使用读-写锁比常规锁带来性能改进时程序参数的范围。 333

7.8 请实现用链解决冲突的线程散列表。散列表中，单个锁与含k个散列表项的块对应。线程如果试图读/写块中的元素，必须先锁定对应的块。考查你的实现中作为k的函数的性能。

7.9 将散列表中的锁改为读-写锁，只在插入数据项到链表时才使用写入锁。考查此程序作为k的函数的性能。比较此实现与使用常规锁实现时的性能。

7.10 编写一个线程程序计算Eratosthenes筛子。在实现前仔细考虑线程策略。弄清一些问题是重要的，例如，不能从筛子中去除6的倍数，除非已去除了3的倍数（此时可以看出，无需先去除6的倍数）。用流水作业（装配线）策略，即用当前最小元素构成装配线的下一站是解决问题的一种方法。

7.11 请编写一个线程程序解决15迷宫问题。程序取某个初始位置，并有一张开放表记录未走过的位置。表按迷宫板度量的“优良度”排序。曼哈坦距离（即每个板格所需移动的x方向的位移和y方向的位移之和）是较好的度量方法之一。开放表是用堆来实现的工作队列。每个线程从工作队列中提取工作（迷宫板），将它扩张到所有可能的后继者，如果后继者还没有用过，则将它插入到工作队列中。使用散列表（习题7.9）记录以前未用过的项。请绘出程序线程数目对增加运行速度的关系曲线。你可以计算出对某个参考迷宫板程序的加速对于不同的线程数目是相同的。

7.12 修改上面的程序，使之有多个（如k个）开放表。此时，每个线程取出一个随机的开放表，并试图从随机表中取出一个“迷宫板”，把表扩充后再插回到另一个随机选择的表中。请绘出现在程序对于线程数目的加速比曲线。将性能与上一题作比较。请小心使用锁及trylock（试探锁），使串行化开销达到最小限度。

7.13 实现并测试例7.14中矩阵相乘的OpenMP程序。使用OMP_NUM_THREADS环境变量控制线程的数目，并绘出不同线程数目的性能曲线。考虑如下三种情况：i)只有最外层的循环并行化；ii)最外层的两个循环并行化；iii)所有三个循环都并行化。这三种情况的结果是什么？ 334

7.14 考虑一个调用函数dummy的简单循环，函数包含一个可控制的延迟。所有对该函数的调用彼此独立。在4个使用static，dynamic和guided调度的线程间划分这个循环。对static调度和guided调度使用不同的参数。记录下当dummy函数的延迟变大时实验的结果。

7.15 考虑以行压缩格式存储的一个稀疏矩阵（在网页或任何有关稀疏线性代数的书籍中可以找到这种格式的描述）。编写一个OpenMP程序计算这种矩阵与向量相乘的乘积。从

Matrix Market (http://math.nist.gov/MatrixMarket/) 下载样本矩阵，并测试程序的性能，以矩阵大小和线程数目作为性能的函数。

7.16 用sections创建单一的producer任务及单一的consumer任务，在OpenMP中建立一个生产者-消费者程序框架。用锁来保证适当的同步。针对不同数目的生产者和消费者测试程序。

335

第8章　稠密矩阵算法

在数值与非数值计算中经常使用矩阵与向量算法。本章讨论关于稠密矩阵（dense matrix）或者满矩阵（full matrix）的一些关键算法，这种矩阵没有或很少见可用的零元素。为了方便教学，我们专门论述方阵，但是在应用中，本章的算法同样适用于矩形矩阵。

由于矩阵和向量的规则结构，涉及矩阵与向量的并行计算容易用于数据分解（见3.3.2节）。取决于要进行的计算，可通过划分输入、输出或者中间数据导出分解。3.4.1节详细描述并行计算中各种划分矩阵的方案。本章讨论的算法使用一维和二维块、循环以及块循环等划分。

本章所述多数算法的另外一个特性是它们的每个进程只有一个任务。由于任务到进程的映射是一对一关系，我们通常并不明确地讨论任务，而是把问题直接分解或者划分成进程。

8.1　矩阵向量乘法

这一节，我们讨论的问题是，一个$n \times n$稠密矩阵A乘一个$n \times 1$向量x产生一个$n \times 1$结果向量y。算法8-1显示这个问题的一个串行算法，该算法需要n^2次乘法和加法运算。 337

算法8-1　n × n矩阵A与n × 1向量x相乘产生一个n × 1向量y的串行算法

```
1.  procedure MAT_VECT (A, x, y)
2.  begin
3.      for i := 0 to n − 1 do
4.      begin
5.          y[i] := 0;
6.          for j := 0 to n − 1 do
7.              y[i] := y[i] + A[i, j] × x[j];
8.      endfor;
9.  end MAT_VECT
```

假设一次乘法和加法运算需要一个单位时间，则串行运行时间为

$$W = n^2 \tag{8-1}$$

根据使用的是一维行划分、一维列划分或者是二维划分，至少可能有三个不同的矩阵向量乘法的并行公式。

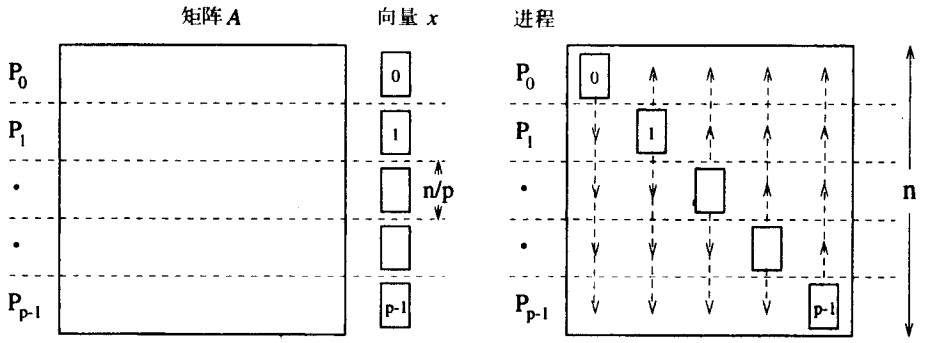
8.1.1　一维行划分

本节讨论使用一维行块划分的矩阵-向量乘法的并行算法。使用一维列块划分的矩阵-向量乘法的并行算法类似（见习题8.2），而且关于并行运行时间有相似的表达式。图8-1表示一维块划分方式下矩阵向量乘法运算中数据的分布与移动。

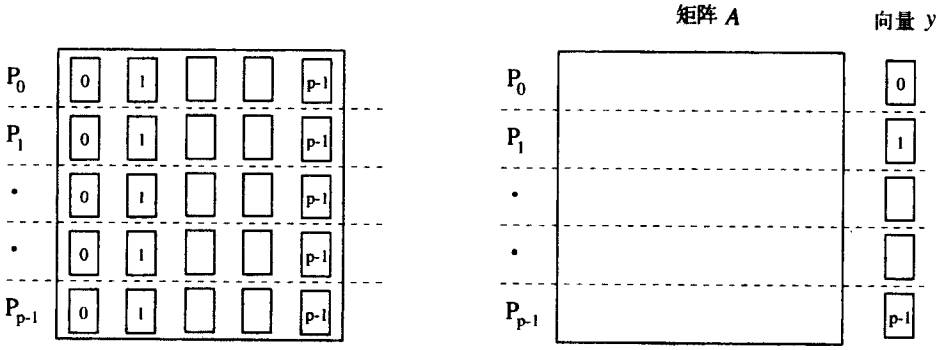
1. 每个进程一行

首先考虑$n \times n$矩阵A在n个进程之间划分的情形，这时每个进程存储矩阵A的一整行以及

$n \times 1$向量x的一个元素。关于一维行块划分矩阵与向量的初始分布如图8-1a所示。进程P_i最初拥有$x[i]$和$A[i,0]$，$A[i,1]$，…，$A[i,n-1]$并且负责计算$y[i]$。矩阵A的每一行与向量x相乘（算法8-1），因此每个进程需要整个向量x。但是因为每个进程开始时，只拥有向量x的一个元素，所以需要用多对多广播的方法将所有的元素分布到所有的进程中。图8-1b说明了这个通信步骤。完成向量x在进程中的分布后（图8-1c），进程P_i计算 $y[i] = \sum_{j=0}^{n-1}(A[i,j] \times x[j])$（算法8-1的
338 第6和第7行）。如图8-1d所示，结果向量y的存储方法与初始向量x完全相同。



a）对矩阵和初始向量x的初始划分　　b）通过多对多广播在所有进程中整个向量的分布



c）整个向量在广播后分配到各个进程　　d）矩阵和结果向量y的最终分配

图8-1　使用一维行块划分的$n \times n$矩阵与$n \times 1$向量的乘法
对于每个进程一行的情况，$p = n$

并行运行时间　每个进程从一个向量元素开始，在任意体系结构（表4-1）中，在n个进程中向量元素的多对多广播所需时间为$\Theta(n)$。每个进程中，向量x与矩阵A一行的乘法所需要的时间也为$\Theta(n)$。因此，完成全部n个进程需要的时间为$\Theta(n)$，从而得到进程时间的乘积$\Theta(n^2)$。由于串行算法的复杂度是$\Theta(n^2)$，并行算法是成本最优的。

2. 进程数少于n

考虑p个进程($p < n$)的情况，使用块一维划分方法在进程之间划分矩阵。每个进程最初存

储矩阵A的n/p个整行和向量x的n/p个元素。由于矩阵A的每一行要与向量x相乘，每个进程需 339
要完整的向量x（即各个进程包含向量x的所有部分）。这又要求使用如图8-1b和8-1c所示的多对多广播。多对多广播在p ($p < n$)个进程之间进行，传输的信息量为n/p。完成通信之后，每个进程中矩阵A的n/p个整行要与$n \times 1$向量x相乘，得到结果向量y的n/p个元素。如图8-1d所示，结果向量y的分布方式与初始向量x相同。

并行运行时间 根据表4-1，在$p(p < n)$个进程之间进行的多对多广播，传输的信息量为n/p，通信时间为$t_s \log p + t_w(n/p)(p-1)$。当$p$很大时，通信时间可用$t_s \log p + t_w n$近似。完成通信之后，每个进程执行矩阵$A$的$n/p$个行与向量$x$的乘法所需要的时间为$n^2/p$。因此，该过程的并行运行时间为

$$T_P = \frac{n^2}{p} + t_s \log p + t_w n \tag{8-2}$$

这个并行公式的进程时间乘积为$pT_p = n^2 + pt_s \log p + pt_w n$。$p = O(n)$时，该算法是成本最优的。

可扩展性分析 按照5.4.2节的分析步骤，通过逐个研究开销函数的各项，现在我们来推导矩阵向量乘法的等效率函数。考虑由公式(8-2)给出的关于超立方结构的并行运行时间。关系$T_o = pT_p - w$给出超立方体上使用一维块划分时的矩阵-向量乘法开销函数的下述表达式:

$$T_o = t_s p \log p + t_w np \tag{8-3}$$

回忆第5章，决定并行算法等效率函数的主要关系是$W = KT_o$（公式(5-14)），其中$K = E/(1-E)$，E是期望效率。重写矩阵-向量乘法算法中的这个关系，首先只用T_o的t_s项，得到

$$W = Kt_s p \log p \tag{8-4}$$

公式(8-4)给出关于消息启动时间的等效率函数项。同理，对于开销函数中的t_w项，可得

$$W = Kt_w np$$

由于$W = n^2$（公式(8-1)），我们使用K、p和t_w导出W的表达式（即由t_w导出的等效率函数）如下:

$$\begin{aligned} n^2 &= Kt_w np \\ n &= Kt_w p \\ n^2 &= K^2 t_w^2 p^2 \\ W &= K^2 t_w^2 p^2 \end{aligned} \tag{8-5}$$ 340

现在考虑这个并行算法的并发度。使用一维划分时，最多可以使用n个进程实现$n \times n$矩阵A和$n \times 1$向量x的乘法。换句话说，$p = O(n)$，它能得到下面的条件

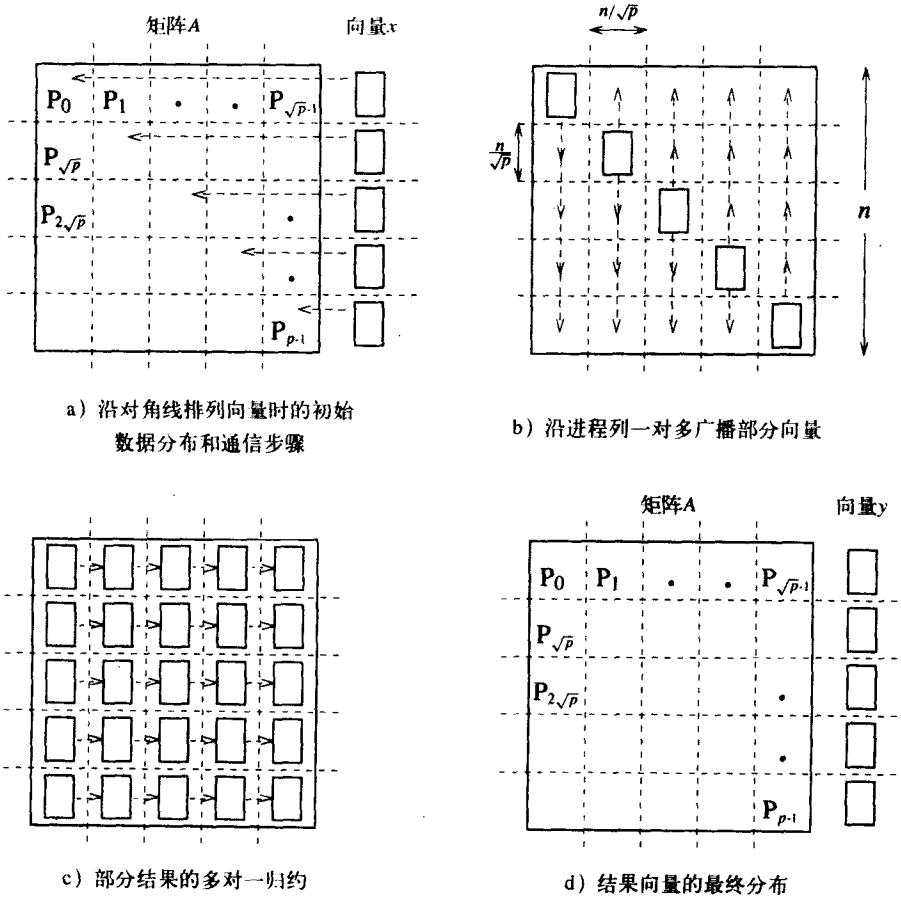
$$\begin{aligned} n &= \Omega(p) \\ n^2 &= \Omega(p^2) \\ W &= \Omega(p^2) \end{aligned} \tag{8-6}$$

通过比较公式(8-4)、(8-5)和(8-6)，可以确定总的渐近等效率函数。在三个公式中，(8-5)

和(8-6)给出最高的渐近速度，按照这个速度，问题的规模必需随着进程数的增加而增加，才能保持固定的效率。这个速度$\Theta(p^2)$就是使用一维划分时并行矩阵-向量乘法算法的渐近等效率函数。

8.1.2 二维划分

本节讨论使用二维块划分在各个进程间分布矩阵时矩阵-向量乘法的并行算法。图8-2给出矩阵在各个进程中的分布，以及向量在各个进程中的分布与移动。



a）沿对角线排列向量时的初始数据分布和通信步骤

b）沿进程列一对多广播部分向量

c）部分结果的多对一归约

d）结果向量的最终分布

图8-2 使用二维块划分的矩阵-向量乘法。对于每个进程一个元素的情形，对于$n \times n$矩阵有$p = n^2$

1. 每个进程一个元素

我们先考虑简单的情况，把$n \times n$矩阵A在n^2个进程中划分，这样每个进程只拥有矩阵A的一个元素。把$n \times 1$向量x分布在n个进程的最后一列，每个进程拥有向量x的一个元素。由于在计算中，向量x的每个元素要与矩阵A的每行的相应元素相乘，因此向量x的第i个元素必须分布在能与矩阵A每一行的第i个元素进行运算的进程中。图8-2a和b给出这样分布的通信步骤。请注意图8-2与图8-1的相似之处。在进行乘法运算前，矩阵A的元素与向量x的元素必须位于同样的相对位置，如图8-1c所示。但是，向量的通信步骤因不同的划分策略而异。使用一维

划分时，向量的元素只需要通过水平的划分边界（图8-1）。而使用二维划分时，向量的元素需要通过水平与垂直的两个划分边界（图8-2）。

如图8-2a所示，使用二维划分时，第一个通信步骤是将向量x沿着矩阵A的主对角线排列。通常向量x沿着矩阵A的主对角线而不是最后一列存储，此时上述步骤是不需要的。第二步是把向量元素从每个对角进程复制到相应列的所有进程上。如图8-2b所示，这个步骤包括n个同时进行的一对多广播操作，每一列的进程包括一个操作。这两个通信步骤后，每个进程再将其矩阵元素与x的相应元素相乘。为了得到结果向量y，必须将每一行的乘积结果相加，并把结果放到最后一列的进程中。图8-2c显示这一步骤，该步骤以行的最后一个进程为目标，每一行都需要进行一次多对一归约操作（见4.1节）。在完成归约步骤后，并行矩阵-向量的乘法即告结束。

并行运行时间 在这个算法中使用三个基本通信操作：将向量x的元素沿矩阵A的主对角线分布的一对一通信；在每一列的n个进程中对每个向量元素的一对多广播；在每一行中进行的多对一归约。每个操作需要的时间是$\Theta(\log n)$。由于在固定时间内每个进程执行一次乘法，这个算法的总体并行运行时间为$\Theta(n)$。计算成本（进程时间乘积）为$\Theta(n^2 \log n)$；因此算法不是成本最优的。

341 ~ 342

2. 进程数少于n^2

使用二维块划分矩阵时，通过使用少于n^2个进程增加每个进程的计算粒度，可能得到矩阵向量乘法的成本最优的并行算法。

考虑有p个进程的逻辑二维格网，每个进程拥有矩阵的一个$(n/\sqrt{p})\times(n/\sqrt{p})$块。向量只分布在最后一个进程列的$n/\sqrt{p}$个进程中。图8-2显示这种情况下的初始数据映射和通信步骤。在进行乘法运算前，必须将所有的向量分布在每一行进程中。首先向量应该沿着主对角线排列。为此，最右列的每个进程必须把它的$n/\sqrt{p}$个向量元素传输到它所在行的对角线进程中。然后，进行这$n/\sqrt{p}$个向量元素按列的一对多广播操作。接着，每个进程执行n^2/p次乘法运算和$n/\sqrt{p}$个乘积的局部求和运算。在这一步的最后，如图8-2c所示，每个进程有$n/\sqrt{p}$个部分和，对它们必须沿着每一行累加，得到结果向量。因此，以每行的最右边进程为目标，本算法的最后一步是每行的$n/\sqrt{p}$个值的多对一归约。

并行运行时间 在第一步中，从最右边的进程发送大小为$n/\sqrt{p}$的消息到对角线上的进程（图8-2a），需要的时间为$t_s + t_w n/\sqrt{p}$。使用4.1.3节讲述的程序，我们可以执行按列的一对多广播操作，需要的时间最多为$(t_s + t_w n/\sqrt{p})\log(\sqrt{p})$。忽略加法运算的时间，最后按行的多对一归约操作也需要同样的时间。假设一对乘法和加法需要单位时间，则计算中每个进程需要的时间大约为n^2/p。因此，这个过程的并行运行时间为

$$
\begin{aligned}
T_P &= \overbrace{n^2/p}^{\text{计算}} + \overbrace{t_s + t_w n/\sqrt{p}}^{\text{排列向量}} + \\
&\quad \overbrace{(t_s + t_w n/\sqrt{p})\log(\sqrt{p})}^{\text{按列一对多广播}} + \overbrace{(t_s + t_w n/\sqrt{p})\log(\sqrt{p})}^{\text{多对一归约}} \\
&\approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p
\end{aligned}
\tag{8-7}
$$

可扩展性分析 使用公式(8-1)和(8-7)，并利用关系式$T_o = pT_p - W$（公式(5-1)），我们可以得到这个并行算法开销函数的下述表达式:

$$T_o = t_s p \log p + t_w n\sqrt{p} \log p \tag{8-8}$$

343 现在我们按照5.4.2节的方法，逐一考虑开销函数的各项，进行近似等效率分析（更精确的等效率分析见习题8.4）。对开销函数的t_s项，公式(5-14)产生

$$W = Kt_s p \log p \tag{8-9}$$

公式(8-9)给出关于消息启动时间的等效率项。通过用问题的规模n^2平衡$t_w n\sqrt{p}\log p$项，可得到由t_w导出的等效率函数。利用公式(5-14)的等效率关系，可得:

$$\begin{aligned} W = n^2 &= Kt_w n\sqrt{p}\log p \\ n &= Kt_w\sqrt{p}\log p \\ n^2 &= K^2 t_w^2 p \log^2 p \\ W &= K^2 t_w^2 p \log^2 p \end{aligned} \tag{8-10}$$

最后，由于二维划分的并发度为n^2（即可以使用的最大进程数目为n^2），可得下列公式:

$$\begin{aligned} p &= O(n^2) \\ n^2 &= \Omega(p) \\ W &= \Omega(p) \end{aligned} \tag{8-11}$$

在公式(8-9)、(8-10)、(8-11)中，等号右边最大的表达式决定本并行算法的整体等效率函数。为了简化分析，我们忽略常数的影响，只考虑保持固定效率所需的问题规模增加的渐近速度。由t_w（公式(8-10)）导致的渐近等效率项显然对由t_s（公式(8-9)）和并发性（公式(8-11)）导致的渐近等效项占有优势。因此，整体渐近等效率函数为$\Theta(p \log^2 p)$。

等效率函数也决定成本最优的标准（见5.4.3节）。如果等效率函数为$\Theta(p \log^2 p)$，则对于给定规模为W的问题，使成本最优的最大进程数目可由以下关系式确定:

$$\begin{aligned} p\log^2 p &= O(n^2) \\ \log p + 2\log\log p &= O(\log n) \end{aligned} \tag{8-12}$$

忽略低次项，

$$\log p = O(\log n)$$

在公式(8-12)中用$\log n$代替$\log p$，

$$\begin{aligned} p\log^2 n &= O(n^2) \\ p &= O\left(\frac{n^2}{\log^2 n}\right) \end{aligned} \tag{8-13}$$

344

对于$n \times n$矩阵-向量乘法的二维矩阵划分方法，公式(8-13)的右端给出成本最优时进程数目的渐近上界。

3. 一维与二维划分的比较

比较公式(8-2)与(8-7)可见，在进程数目相等的情况下，矩阵-向量乘法中使用二维划分要比使用一维划分更快。如果进程数目超过n，就不能使用一维划分。根据本节的分析，即使进程数目小于或者等于n，使用二维划分仍然是最好的选择。

在另两个划分方案中，二维划分有更好（更小）的渐近等效率函数。因此，使用二维划分的矩阵-向量乘法有更好的可扩展性；也就是说，与一维划分相比，二维划分可以在更多的进程中得到同样的效率。

8.2 矩阵与矩阵的乘法

本节讨论两个$n \times n$稠密矩阵A与B相乘得到乘积矩阵$C = A \times B$的并行算法。本章矩阵乘法的所有并行算法都基于算法8-2给出的传统串行算法。如果我们假定一次加法与乘法运算对（第8行）需要一个单位时间，则这个算法的串行运行时间为n^3。虽然也可以找到具有更好渐近串行复杂度的矩阵乘法算法，例如Strassen算法，但是，为简单起见，在本书中我们仍然假定传统算法是可用的最好串行算法。习题8.5探讨用Strassen方法作为基本算法时的并行矩阵乘法算法的性能。

算法8-2 两个n × n矩阵相乘的传统串行算法

```
1.  procedure MAT_MULT (A, B, C)
2.  begin
3.      for i := 0 to n − 1 do
4.          for j := 0 to n − 1 do
5.              begin
6.                  C[i, j] := 0;
7.                  for k := 0 to n − 1 do
8.                      C[i, j] := C[i, j] + A[i, k] × B[k, j];
9.              endfor;
10. end MAT_MULT
```

345

算法8-3 n × n矩阵块大小为(n/q) × (n/q)的块矩阵乘法算法

```
1.  procedure BLOCK_MAT_MULT (A, B, C)
2.  begin
3.      for i := 0 to q − 1 do
4.          for j := 0 to q − 1 do
5.              begin
6.                  Initialize all elements of C_{i,j} to zero;
7.                  for k := 0 to q − 1 do
8.                      C_{i,j} := C_{i,j} + A_{i,k} × B_{k,j};
9.              endfor;
10. end BLOCK_MAT_MULT
```

在矩阵乘法以及矩阵的其他各种算法中，块矩阵运算是一个很有用的概念。在矩阵计算中，我们经常可以把对矩阵所有元素的标量代数运算转换成对原矩阵的子矩阵或者块矩阵的矩阵代数运算。这种对子矩阵的代数运算称为块矩阵运算（block matrix operation）。例如，

$n \times n$矩阵可以看成块$A_{i,j}(0 \leqslant i, j < q)$的$q \times q$矩阵，其中每个块$A_{i,j}$是原矩阵的一个$(n/q) \times (n/q)$子矩阵。因此，算法8-2中表示的矩阵乘法算法可写成算法8-3，其中第8行的乘法与加法运算分别为矩阵的乘法与加法运算。算法8-2与算法8-3不仅是最后结果相同，而且每个算法的标量加法与标量乘法的总次数也完全相同。算法8-2需要进行n^3次标量的加法与乘法运算，而算法8-3需要执行q^3次矩阵乘法，每次涉及$(n/q) \times (n/q)$个矩阵并需要$(n/q)^3$次标量加法与乘法运算。我们可以用p（选择 $q = \sqrt{p}$）个进程实现并行块矩阵乘法，每个进程计算不同的$C_{i,j}$块。

下面几小节介绍算法8-3的几种并行形式。以下每个并行矩阵乘法算法都使用矩阵的块二维划分。

8.2.1 简单的并行算法

考虑两个$n \times n$矩阵A和B，把它们分别划分成p个大小为$(n/\sqrt{p}) \times (n/\sqrt{p})$的块$A_{i,j}$和$B_{i,j}(0 \leqslant i, j < \sqrt{p})$。这些块映射到$\sqrt{p} \times \sqrt{p}$个进程的逻辑格网。这些进程的标号为$P_{i,j}(0 \leqslant i, j \leqslant \sqrt{p-1})$。进程$P_{i,j}$最初存储$A_{i,j}$和$B_{i,j}$，并计算结果矩阵的块$C_{i,j}$。计算子矩阵$C_{i,j}$需要所有子矩阵$A_{i,k}$和$B_{k,j}$$(0 \leqslant k \leqslant \sqrt{p-1})$。为了得到全部$A_{i,k}$和$B_{k,j}$，要在这些进程的每行进行矩阵$A$的块的多对多广播操
346 作，同时在这些进程的每列进行矩阵B的块的多对多广播操作。在$P_{i,j}$得到$A_{i,k}$和$B_{k,j}$$(0 \leqslant k \leqslant \sqrt{p-1})$后，再执行算法8-3中第7和第8行的子矩阵乘法和加法运算。

性能与可扩展性分析 本算法要在各进程组（每组$\sqrt{p}$个进程）中进行两次多对多广播步骤（每次包括在进程格网的所有行和列同时进行$\sqrt{p}$次广播）。传输的消息是包含n^2/p个元素的子矩阵。从表4-1可知，总通信时间是$2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p}-1))$。通信步骤结束后，每个进程计算子矩阵$C_{i,j}$，需要进行$(n/\sqrt{p}) \times (n/\sqrt{p})$子矩阵的$\sqrt{p}$次乘法操作（算法8-3中第7、8行，$q = \sqrt{p}$）。这一过程需要的总时间为$\sqrt{p} \times (n/\sqrt{p})^3 = n^3/p$。所以并行运行时间近似为：

$$T_P = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}} \tag{8-14}$$

进程-时间积为$n^3 + t_s p \log p + 2t_w n^2 \sqrt{p}$，当$p = O(n^2)$时，并行算法是成本最优的。

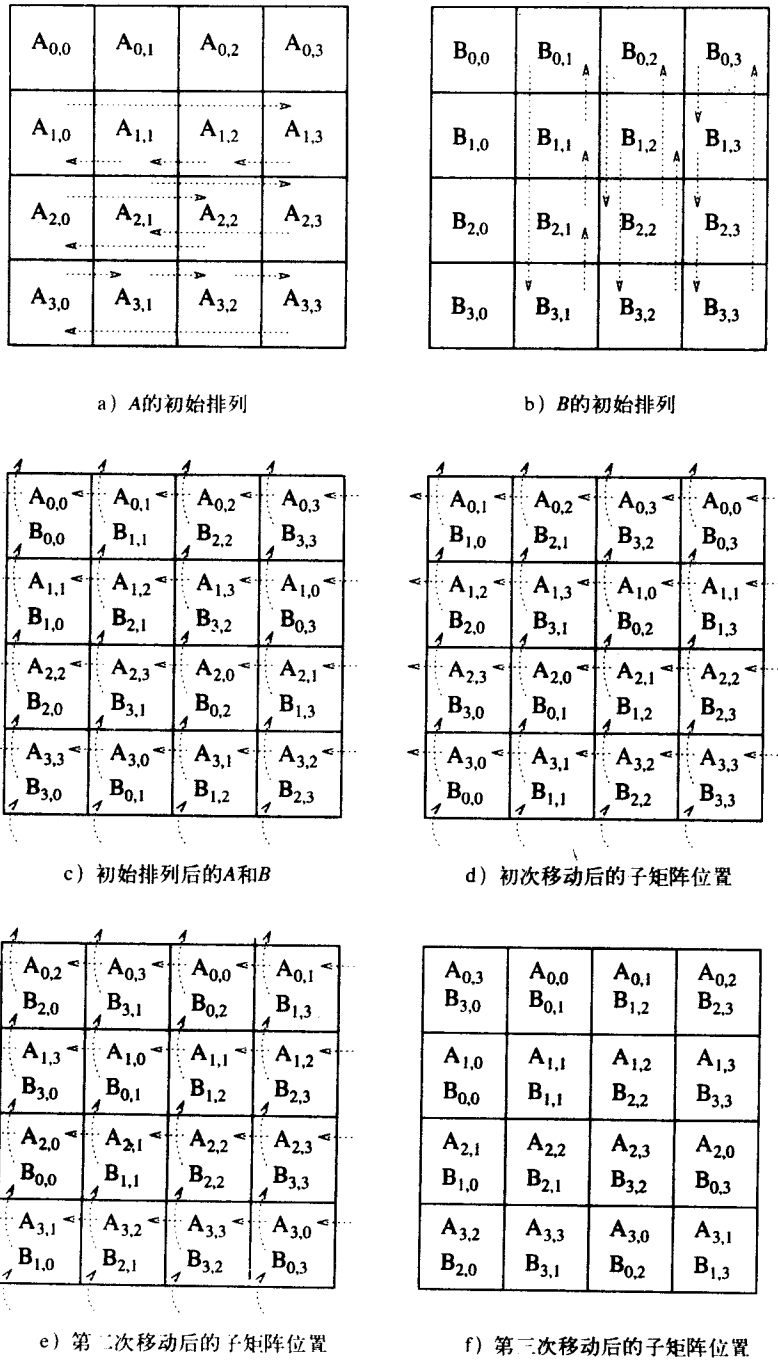
由t_w和t_s导致的等效率函数分别是$t_s p \log p$和$8(t_w)^3 p^{3/2}$。因此，由通信开销导致的整体等效率函数为$\Theta(p^{3/2})$。这个算法使用的最大进程数目为n^2；所以$p \leqslant n^2$或$n^3 \geqslant p^{3/2}$。从而由并发导致的等效率函数也是$\Theta(p^{3/2})$。

本算法的一个显著缺陷是它对内存需求过大。在通信阶段的最后，每个进程有矩阵A和B的$\sqrt{p}$个块，每个块需要的内存为$\Theta(n^2/p)$，每个进程需要的内存为$\Theta(n^2/\sqrt{p})$。全部进程需要的总内存为$\Theta(n^2\sqrt{p})$，这是串行算法所需内存的$\sqrt{p}$倍。

8.2.2 Cannon算法

Cannon算法是8.2.1节提出的简单并行算法的内存高效版本。为了研究这个算法，我们再把矩阵A和B划分成p个方块，进程的标号从$P_{0,0}$到$P_{\sqrt{p-1},\sqrt{p-1}}$，并在最初把子矩阵$A_{i,j}$和$B_{i,j}$分配给进程$P_{i,j}$。虽然第$i$行的每个进程需要全部$\sqrt{p}$个子矩阵$A_{i,k}(0 \leqslant k < \sqrt{p})$，但我们还是能调度第$i$行$\sqrt{p}$个进程的计算，使得每个进程在任何时刻都使用不同的$A_{i,k}$。每完成一次子矩阵乘法，

这些块在各进程之间被轮流使用，使得每次轮流之后每个进程都可以得到新的$A_{i,k}$。对列使用同样的调度，则在任何时刻，任何进程至多拥有每个矩阵的一个块，在所有的进程中，该算法需要的总内存量为$\Theta(n^2)$。Cannon算法正是基于这个思想。图8-3说明Cannon算法中不同进程上子矩阵乘法的调度过程，其中有16个进程。



a）A的初始排列　　b）B的初始排列

c）初始排列后的A和B　　d）初次移动后的子矩阵位置

e）第二次移动后的子矩阵位置　　f）第三次移动后的子矩阵位置

图8-3　Cannon算法在16个进程上的通信步骤

347
i
348

在算法的第一个通信步骤中，用这样一种方式排列矩阵A和B的块，使得每个进程对其本地子矩阵相乘。如图8-3a所示，排列矩阵A和B的块时，用i步把矩阵A的所有子矩阵$A_{i,j}$都移到左边（带回绕）。与之类似，用j步把矩阵B的所有子矩阵$B_{i,j}$都移到上边（带回绕）。进程的每行和每列都要进行循环移位操作（见4.6节），并且保持子矩阵 $A_{i,(j+i) \bmod \sqrt{p}}$ 和 $B_{(j+i) \bmod \sqrt{p},j}$ 在进程$P_{i,j}$中。图8-3c显示初始排列后的矩阵A和B的块，此时每个进程可以进行第一次子矩阵相乘运算。完成一个子矩阵相乘步骤后，矩阵A的每个块向左移动一步，矩阵B的每个块向右移动一步（也带回绕），如图8-3d所示。在进程$P_{i,j}$中一共进行 $\sqrt{p}$ 次这样的$A_{i,k}$与$B_{k,j}$ $(0 \leqslant k < \sqrt{p})$的相乘和一步向上移位。这就完成矩阵$A$和$B$的乘法运算。

性能分析 如图8-3a和b所示，两个矩阵的初始排列包括一次按行和一次按列的循环移位。在其中任何一次移位中，每个块移动的最大距离是 $\sqrt{p}-1$，两次移位操作需要的总时间为 $2(t_s + t_w n^2/p)$（见表4-1）。本算法的每个计算与移位阶段中都有 $\sqrt{p}$ 次单步移位，每个单步移位需要的时间为$t_s + t_w n^2/p$，因此，算法这一阶段两个矩阵的总通信时间为$2(t_s + t_w n^2/p)\sqrt{p}$。在带宽足够的网络中，当$p$充分大时，与计算和移位阶段相比，初始排列的通信时间可以忽略不计。

每个进程要进行 $\sqrt{p}$ 次 $(n/\sqrt{p}) \times (n/\sqrt{p})$ 子矩阵乘法运算。假设每次加法和乘法需要一个单位时间，每个进程花在计算中的总时间为n^3/p。因此，本算法需要的总并行运行时间大约为

$$T_P = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w \frac{n^2}{\sqrt{p}} \tag{8-15}$$

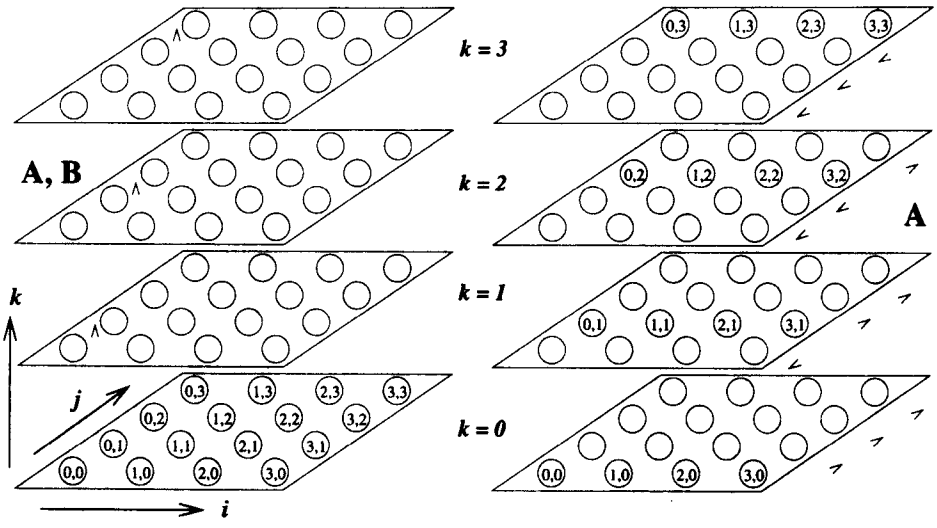
Cannon算法的成本最优条件与8.2.1节提出的简单算法中的条件相同。与简单算法一样，Cannon算法的等效率函数是$\Theta(p^{3/2})$。

8.2.3 DNS算法

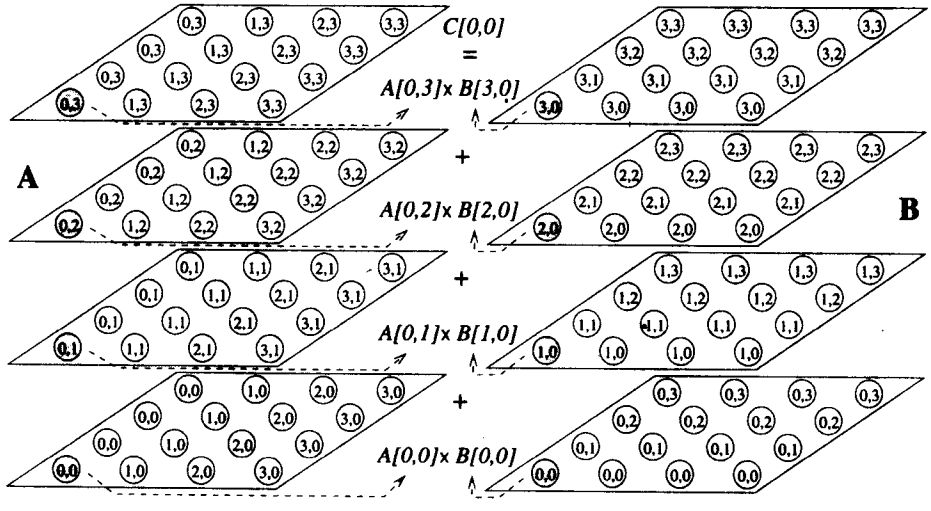
到目前为止提出的矩阵相乘算法中，都对输入和输出矩阵进行块二维划分，而且对$n \times n$矩阵使用的进程数不超过n^2。由于在串行算法中有$\Theta(n^3)$次操作，所以这些算法的并行运算时间为$\Omega(n)$。我们现在给出一种并行算法，它基于划分中间数据，最多能够使用n^3个进程，通过使用$\Omega(n^3/\log n)$个进程，执行矩阵乘积所需的时间为$\Theta(\log n)$。这个算法称为DNS算法，因为它是由Dekel、Nassimi和Sahni提出的。

我们首先介绍基本思想，不考虑进程间的通信。假设可以用n^3个进程计算两个$n \times n$矩阵
349
的乘积。这些进程排列在一个三维$n \times n \times n$逻辑阵列中。由于矩阵相乘算法要执行n^3次标量乘法，每个进程都安排一次标量乘法。根据进程在阵列中的位置对它们标号，分配进程$P_{i,j,k}$ $(0 \leqslant i,j,k < n)$执行$A[i,k] \times B[k,j]$的乘法运算。每个进程完成一个乘法运算步骤后，再将各个进程$P_{i,j,0}$，$P_{i,j,1}$，…，$P_{i,j,n-1}$的内容相加，得到$C[i,j]$。所有$C[i,j]$的求和运算可以同时在$\log n$个步骤中进行。因此，乘法运算需要一个步骤；而求和运算需要$\log n$个步骤；也就是说，用这个算法计算$n \times n$矩阵的乘积所需时间为$\Theta(\log n)$。

基于这个思想，我们现在来描述矩阵乘法的一个并行实现。如图8-4所示，可以把进程排列设想成n个平面，每个平面$n \times n$个进程。每个平面对应k的一个不同值。初始时，如图8-4a所示，矩阵分布在三维进程阵列的底部与$k = 0$相对应的n^2个进程中。最初进程$P_{i,j,0}$拥有$A[i,j]$和$B[i,j]$。



a）A和B的初始分布　　b）将$A[i,j]$从$P_{i,j,0}$移动到$P_{i,j,j}$后的分布



c）沿j轴广播$A[i,j]$后的分布　　d）矩阵B中对应的分布

图8-4　在64个进程上的4 × 4矩阵A和B的乘法中DNS算法的通信步骤。c）中带阴影的进程存储A的第一行元素，d）中带阴影的进程存储B的第一列元素

进程$P_{i,j,*}$的纵向列计算$A[i,*]$行与$B[*,j]$列的点积。因此，A的行与B的列需要进行相应的移动，保证进程$P_{i,j,*}$的每个纵向列有$A[i,*]$行与$B[*,j]$列。准确地说，进程$P_{i,j,k}$应有$A[i,k]$与$B[k,j]$。

图8-4a ~ c显示各个进程中分布的矩阵A的元素的通信模式。首先，把A的每一列移到不同的平面，保证第j列占据对应于$k = j$的平面中的同样位置，就像最初在对应于$k = 0$的平面中所做的那样。图8-4b显示把$A[i,j]$从$P_{i,j,0}$移动到$P_{i,j,j}$后矩阵A的分布。通过沿着j轴的一对多广播，矩阵A的各列在相应的平面上被复制n次 。这一步骤的结果如图8-4c所示，其中n个进程$P_{i,j,0}$，

$P_{i,j,1}$，…，$P_{i,j,n-1}$都从$P_{i,j,j}$处得到$A[i, j]$的一个副本。此时，$P_{i,j,*}$的每个纵向列有了$A[i, *]$行。准确地说，进程$P_{i,j,k}$有了$A[i, k]$。

矩阵B的通信步骤是类似的，但是进程下标i和j的作用相互对调。在开始的一对一通信步骤，把$B[i, j]$从$P_{i,j,0}$移动到$P_{i,j,i}$，然后从$P_{i,j,i}$广播到$P_{0,i,j}$，$P_{1,i,j}$，…，$P_{n-1,i,j}$。图8-4d表示沿i轴进行这个一对多广播后矩阵B的分布。至此，进程$P_{i,j,*}$的每个纵向列有了$B[*, j]$列。现在进程$P_{i,j,k}$除$A[i, k]$之外还拥有$B[k, j]$。

在这些通信步骤之后，$A[i, k]$与$B[k, j]$在进程$P_{i,j,k}$中相乘。现在通过沿k轴的多对一归约，可以得到乘积矩阵的所有元素$C[i, j]$。在这个步骤中，进程$P_{i,j,0}$累加来自进程$P_{i,j,1}$，…，$P_{i,j,n-1}$的乘积结果。图8-4表示的是求$C[0, 0]$的这一步骤。

DNS算法中有三个主要的通信步骤，1. 把A的列和B的行移动到它们各自的平面，2. 沿A的j轴和B的i轴执行一对多传播，3. 沿k轴进行多对一归约。所有这些操作都在各进程组（每组n个进程）中进行，所需时间为$\Theta(\log n)$。因此，用DNS算法对n^3个进程进行两个$n \times n$矩阵相乘所需的并行运行时间为$\Theta(\log n)$。

350
|
351

进程数少于n^3的DNS算法

n^3个进程的DNS算法不是成本最优的，因为它的进程-时间乘积$\Theta(n^3 \log n)$超过矩阵乘法的串行复杂度$\Theta(n^3)$。现在我们给出这个算法的成本最优的版本，其中使用的进程数少于n^3个。习题8.6中提出使用进程少于n^3个的DNS算法的另一个变体。

对于某个$q < n$，假设p是进程数，$p = q^3$。为了实现DNS算法，我们把两个矩阵划分成$(n/q) \times (n/q)$的块。从而每个矩阵可以看成是由这些块排成的$q \times q$二维方阵。这个算法在q^3个进程上的实现与在n^3个进程上的实现非常相似。唯一的不同是现在对块操作而不是对矩阵的单个元素操作。由于$1 \leq q \leq n$，进程数可在1与n^3之间变化。

性能分析 第一个通信步骤是矩阵A和B的一对一通信，每个矩阵需要的时间为$t_s + t_w (n/q)^2$。第二个通信步骤是矩阵A和B的一对多通信，每个矩阵需要的时间为$t_s \log q + t_w (n/q)^2 \log q$。最后的通信步骤是矩阵$C$的一次多对一归约，需要的时间为$t_s \log q + t_w (n/q)^2 \log q$。每个进程中$(n/q) \times (n/q)$子矩阵相乘需要的时间为$(n/q)^3$。第一步矩阵$A$和$B$一对一通信的时间可以忽略，因为它远远小于一对多通信的时间和多对一归约的时间。在最后的多对一归约步骤中，加法的计算时间也可以忽略，因为它的数量级小于子矩阵相乘时间的数量级。用这些假设，可得如下关于DNS算法并行运行时间的近似表达式：

$$T_P \approx \left(\frac{n}{q}\right)^3 + 3t_s \log q + 3t_w \left(\frac{n}{q}\right)^2 \log q$$

由于$p = q^3$，我们得到

$$T_P = \frac{n^3}{p} + t_s \log p + t_w \frac{n^2}{p^{2/3}} \log p \tag{8-16}$$

并行算法的全部成本为$n^3 + t_s p \log p + t_w n^2 p^{1/3} \log p$。等效率函数为$\Theta(p(\log p)^3)$。当$n^3 = \Omega(p(\log p)^3)$或者$p = O(n^3/(\log n)^3)$时，本算法是成本最优的。

8.3 线性方程组求解

这一节我们讨论如何求解如下形式的线性方程组：

$$\begin{aligned}
a_{0,0}x_0 &+ a_{0,1}x_1 &+ \cdots &+ a_{0,n-1}x_{n-1} &= b_0\\
a_{1,0}x_0 &+ a_{1,1}x_1 &+ \cdots &+ a_{1,n-1}x_{n-1} &= b_1\\
\vdots & \vdots & & \vdots & \vdots\\
a_{n-1,0}x_0 &+ a_{n-1,1}x_1 &+ \cdots &+ a_{n-1,n-1}x_{n-1} &= b_{n-1}
\end{aligned}$$

352

用矩阵表示法这个方程组可以写成$Ax = b$。此处系数矩阵A是稠密的，$A[i,j] = a_{i,j}$，b是$n \times 1$向量$[b_0, b_1, \cdots, b_{n-1}]^T$，$x$是$n \times 1$未知解向量$[x_0, x_1, \cdots, x_{n-1}]^T$。下面用$A[i,j]$表示$a_{i,j}$，用$x[i]$表示$x_i$。

解方程组$Ax = b$通常分两个阶段。首先，用一些代数运算把$Ax = b$化简成上三角形式的方程组

$$\begin{aligned}
x_0 &+ u_{0,1}x_1 &+ u_{0,2}x_2 &+ \cdots &+ u_{0,n-1}x_{n-1} &= y_0\\
& x_1 &+ u_{1,2}x_2 &+ \cdots &+ u_{1,n-1}x_{n-1} &= y_1\\
& & & & \vdots & \vdots\\
& & & & x_{n-1} &= y_{n-1}
\end{aligned}$$

把这个方程组写成$Ux = y$，其中U是单位上三角矩阵，即主对角线元素为1，而所有对角线下的元素为0。在形式上，$i < j$时$U[i,j] = 0$，否则$U[i,j] = u_{i,j}$。此外$U[i,i] = 1$ $(0 \leqslant i < n)$。第二阶段，使用回代法（见8.3.3节），按照相反的次序从$x[n-1]$到$x[0]$，求方程$Ux = y$的解。

我们将在8.3.1和8.3.2节中讨论用上三角化的经典高斯消元法的并行公式。在8.3.1节介绍直接高斯消元算法时，假设系数矩阵是非奇异的，并且用数值稳定的算法进行行列的置换。在8.3.2节讨论方程组稳定数值解的情形，在高斯消元算法的执行过程中，需要对矩阵的列进行置换。

虽然我们只讨论上三角化的高斯消元法，但也可用类似的方法把矩阵A分解成下三角矩阵L和单位上三角矩阵U的乘积，使得$A = L \times U$。这种分解通常称为LU分解。在求解具有同样右端项的多个线性方程组时，进行LU分解是非常有用的。算法3.3给出了一个面向列的LU分解过程。

8.3.1 简单高斯消元算法

串行的高斯消元算法有三个嵌套循环。根据循环的排列顺序，算法还有几种变体。算法8-4是高斯消元的一个变体。本节余下部分中将用来进行并行实现。这个算法程序把线性方程组$Ax = b$变成主对角线为1的上三角方程组$Ux = y$。我们假设矩阵U与A共享存储，并且覆盖A的上三角部分。算法8-4第6行计算的$A[i,j]$实际是$U[i,j]$。类似地，第8行$A[k,k] = 1$实际是$U[k,k]$。第6、7行中，用$A[k,k]$作除数时，算法8-4假设$A[k,k] \neq 0$。

353

在这一节，我们仅讨论在算法8-4中对矩阵A的操作。程序第7行和第13行关于向量b的运算很简单，所以下面省略这些步骤。如果不执行第7、8、13、14行的步骤，则算法8-4进行关于A的LU分解，将A分成乘积$L \times U$。在过程结束后，L存储在A的下三角部分，而U占据A的主对角线以上部分。

在k从0到$n-1$变化时，运用高斯消元过程系统地将变元$x[k]$从方程$k+1$到方程$n-1$中消去，从而把系数矩阵A变成上三角矩阵。如算法8-4所示，在外层循环的第k次迭代（从第3行开始）中，从第$k+1$到第$n-1$的每个方程都减去第k个方程的适当倍数（循环从第9行开始）。第k个方

程（或者矩阵A的第k行）的倍数选择要保证从第$k+1$到第$n-1$的每个方程中第k个系数为零，从而从这些方程中消去$x[k]$。在外层循环的第k次迭代中，高斯消元过程的典型计算如图8-5所示。外层循环的第k次迭代不包括第1行到第k行和第1列到第k列的任何计算。因此，在这一阶段，仅有A的右下方的$(n-k)\times(n-k)$子矩阵（图8-5的阴影部分）参与运算过程。

算法8-4 把线性方程组$Ax=b$变成单位上三角方程组$Ux=y$的串行高斯消元算法。矩阵U占用A的上三角部分。当A[k,k]在第6、7行中用作除数时，算法假设A[k,k]≠0

```
1.  procedure GAUSSIAN_ELIMINATION (A, b, y)
2.  begin
3.      for k := 0 to n − 1 do              /* Outer loop */
4.      begin
5.          for j := k + 1 to n − 1 do
6.              A[k, j] := A[k, j]/A[k, k];  /* Division step */
7.          y[k] := b[k]/A[k, k];
8.          A[k, k] := 1;
9.          for i := k + 1 to n − 1 do
10.         begin
11.             for j := k + 1 to n − 1 do
12.                 A[i, j] := A[i, j] − A[i, k] × A[k, j]; /* Elimination step */
13.             b[i] := b[i] − A[i, k] × y[k];
14.             A[i, k] := 0;
15.         endfor;             /* Line 9 */
16.     endfor;                 /* Line 3 */
17. end GAUSSIAN_ELIMINATION
```

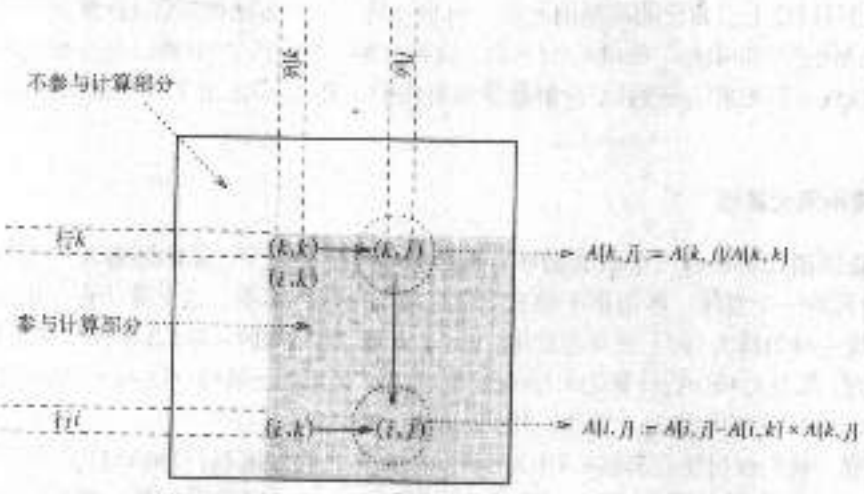


图8-5 高斯消元法中的一个典型计算

高斯消元法大约包括$n^2/2$次除法运算（第6行）和$(n^3/3)-(n^2/2)$次减法和乘法运算（第12行）。在本节，我们假设每次标量运算需要一个单位时间。根据这个假设，高斯消元过程的串行运行时间大约为$2n^3/3$（对于大的n），即

$$W=\frac{2}{3}n^3 \tag{8-17}$$

1. 用一维划分的并行实现

现在考虑算法8-4的并行实现，其中系数矩阵在进程中间进行按行一维划分。算法采用一维按列划分的并行实现与之类似，其实现细节可以根据按行一维划分的实现过程确定（习题8.8和8.9）。

先考虑每个进程分配一行的情形，$n \times n$系数矩阵A按照从P_0到P_{n-1}的n个进程中的行划分。在这个映射中，进程P_i最初存储的元素为$A[i,j]$，$0 \leqslant j < n$。图8-6给出$n = 8$时这种从矩阵到进程的映射。图中也说明$k = 3$时外层循环迭代中的通信与计算情况。

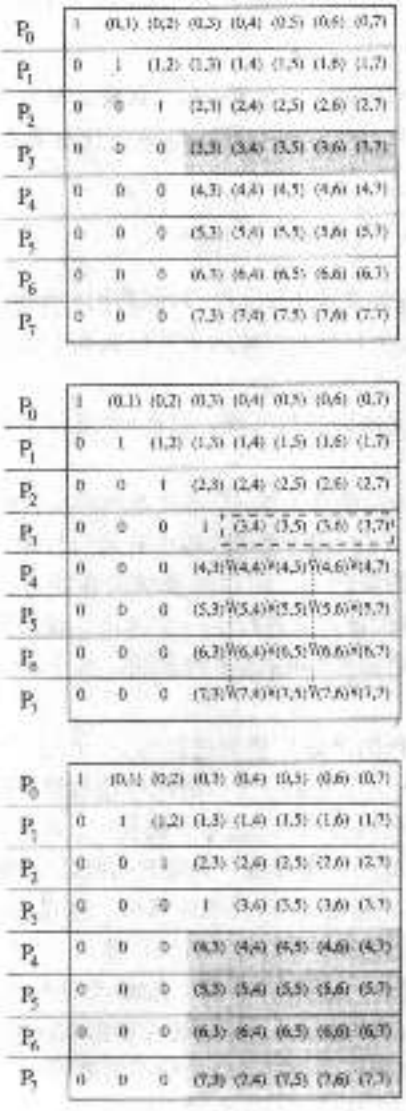


图8-6 在8个进程中按行划分的8 × 8矩阵中，对应于$k = 3$迭代的高斯消元步骤

算法8-4和图8-5表示在第k次迭代开始时，$A[k,k]$除$A[k,k+1]$，$A[k,k+2]$，…，$A[k,n-1]$的情形。参与运算的所有元素（由图8-6a阴影部分所示）属于同一个进程。所以这个步骤不需要任何通信。在算法的第二个计算步骤（第12行的消元步骤）中，矩阵参与计算部分的所有

354
2
356

其他行都要使用第k行修改后（经过除法运算）的元素。如图8-6b所示，这个过程需要一次从第k行参与计算部分到存储$k+1$至$n-1$行的各个进程的一对多广播。最后，如图8-6c中的阴影部分所示，在矩阵剩余的参与计算部分，计算$A[i,j] := A[i,j]-A[i,k] \times A[k,j]$。

与图8-6a对应的第k次迭代中的计算步骤需要在进程P_k中进行$n-k-1$次除法运算。类似地，图8-6c中的计算步骤包括在所有的进程P_i $(k < i < n)$的第k次迭代中进行$n-k-1$次减法和乘法运算。假设一次单独的算术运算需要一个单位时间，则第k次迭代中需要的计算时间为$3(n-k-1)$。注意，当P_k进行除法运算时，其他$p-1$个进程是空闲的，当进程P_{k+1}，P_{k+2}，…，P_{n-1}执行消元操作步骤时，进程P_0，P_1，…，P_k是空闲的。因此在高斯消元法的并行实现中，图8-6a和c显示的计算步骤需要的总时间为 $3\sum_{k=0}^{n-1}(n-k-1) = 3n(n-1)/2$。

图8-6b中的通信步骤所需的时间为$(t_s + t_w(n-k-1))\log n$（见表4-1）。因此，全部迭代过程所需的总时间为$\sum_{k=0}^{n-1}(t_s + t_w(n-k-1))\log n$，等于$[t_s n + t_w(n(n-1)/2)]\log n$。这个算法的整体并行运行时间为

$$T_P = \frac{3}{2}n(n-1) + t_s n\log n + \frac{1}{2}t_w n(n-1)\log n \tag{8-18}$$

由于进程数为n，由(8-18)式中与t_w相关的项导致的成本（即进程-时间乘积）为$\Theta(n^3\log n)$。这个成本渐近地高于该算法的串行运行时间（公式(8-17)），因此这个并行实现不是成本最优的。

流水线通信与计算 我们现在给出一个高斯消元法的并行实现，它在n个进程上是成本最优的。

在刚讨论的并行高斯消元算法中，算法8-4的外层循环中的n个迭代是顺序执行的。在任何给定时间，所有的进程都在相同的迭代中运行。只有当第k次迭代的所有通信与计算全部完成后，才能开始第$k+1$次迭代。在进程异步工作的条件下，可以对算法的性能进行重大改进；也就是说，任何进程不必等待其他进程完成迭代就可以进行下一步迭代。我们把它称为高斯异步消元法或者高斯消元流水线法。图8-7说明算法8-4的流水线算法，其中5×5矩阵沿着行划分成5个进程的逻辑线性阵列。

在算法8-4的第k次迭代中，进程P_k把矩阵第k行的部分广播到进程P_{k+1}，P_{k+2}，…，P_{n-1}（见图8-6b）。假设这些进程形成一个逻辑线性阵列，P_{k+1}是第一个从进程P_k接收第k行的进程，
357 则P_{k+1}必须把收到的数据转发到P_{k+2}。但是，把第k行转发到P_{k+2}之后，进程P_{k+1}不必等待直到P_{n-1}的所有进程收到第k行后再执行消元步骤（第12行）。类似地，只要P_{k+2}把第k行数据传送到P_{k+3}后就能开始自己的计算，以此类推。同时，完成第k次迭代的计算后，进程P_{k+1}可以执行除法步骤（第6行），并且通过把第$k+1$行发送到进程P_{k+2}开始这一行的广播。

在流水线高斯消元法中，每个进程独立地重复执行下述操作步骤，直到n个迭代全部完成。为简单起见，我们假设步骤 1）与步骤 2）需要相同的时间（这样的假设不影响分析）：

1）如果一个进程有发送给其他进程的数据，就把那些数据发送给相关的进程。

2）如果一个进程能够使用已经有的数据进行一些计算，就进行这种计算。

3）否则，进程等待接收数据，用于进行上述操作。

图8-7显示在5个进程中，对5×5矩阵按行划分的高斯消元法流水线并行执行的16个步骤。如图8-7a所示，第一步是在进程P_0执行第0行的除法运算。然后把修正后的第0行送到进程P_1（图8-7b），进程P_1再把它传输到进程P_2（图8-7c）。这样进程P_1可以独立地使用第0行执行消元

步骤（图8-7d）。下一步进程P_2使用第0行执行消元步骤（图8-7e），同时，已经完成第0次迭代的进程P_1开始它的第1次迭代的除法步骤。在任何给定的时间，可以在不同的进程中执行相同迭代的不同阶段。例如，在图8-7h中，当进程P_3和进程P_4进行第1次迭代的通信时，进程P_2执行同一次迭代的消元步骤。此外，在不同的进程中可以同时执行不止一次迭代。例如，在图8-7i中，当进程P_3进行第1次迭代的消元步骤时，进程P_1执行第2次迭代的除法步骤。

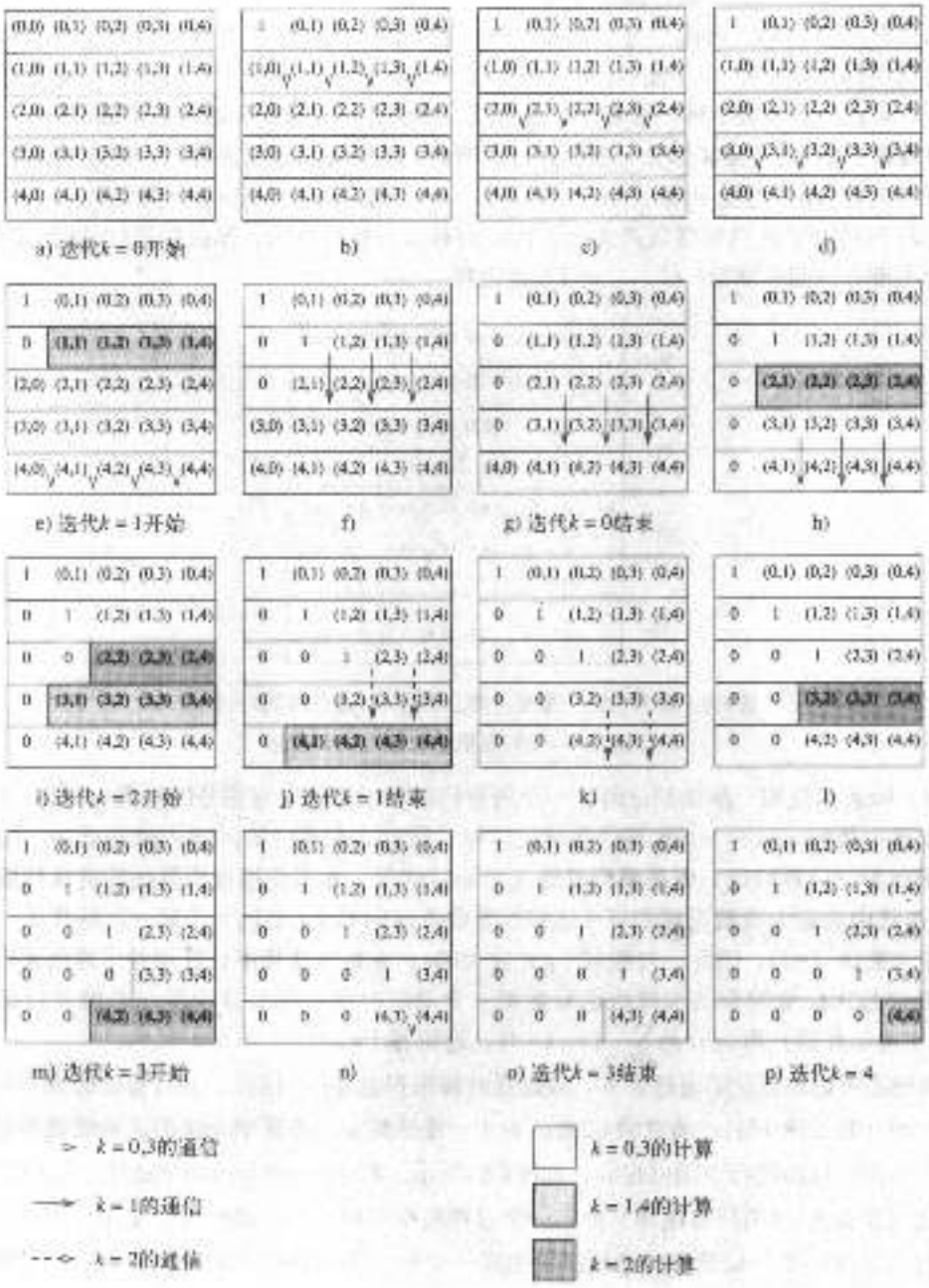


图8-7 用每个进程一行划分的5×5矩阵的流水线高斯消元法

在同步算法中，所有的进程在同一时间只能执行同一迭代，而高斯异步消元法或者流水线高斯消元法与此不同，它是成本最优的。如图8-7所示，算法8-4中外层循环的连续迭代的启动由数目固定的操作步骤分开。一共有n个这样的迭代被启动。最后的迭代只改变系数矩阵的右下角元素；因此，在迭代启动以后，算法在固定的时间内完成。所以整个流水线过程的步骤总数为$\Theta(n)$（习题8.7）。在任何步骤中，要么有$O(n)$个元素在直接连接的进程之间通信，要么有一行的$O(n)$个元素在进行除法运算，或有一行的$O(n)$个元素在执行消元。每个这样的过程需要的时间为$O(n)$。因此，每个这样完整的过程由$\Theta(n)$个复杂度为$O(n)$的步骤组成，它的
358 ~ 359 并行运行时间为$O(n^2)$。由于使用n个进程，成本是$O(n^3)$，与高斯消元法的串行复杂度的数量级相同。因此，用系数矩阵一维划分的流水线版本的并行高斯消元法是成本最优的。

进程数少于n的一维块划分 前面并行高斯消元法的流水线实现可以方便地改为用于$n > p$的情况。考虑在p个进程中划分的$n \times n$矩阵($p < n$)，每个进程分配n/p个相邻的行。图8-8说明在用这样一种映射的典型高斯消元法迭代中的通信。如图所示，算法的第k次迭代需要把第k行的参与部分送到存储第$k+1, \cdots, n-1$行的进程。

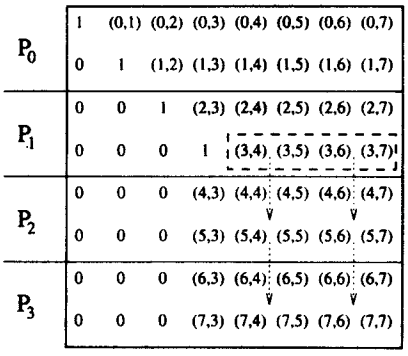


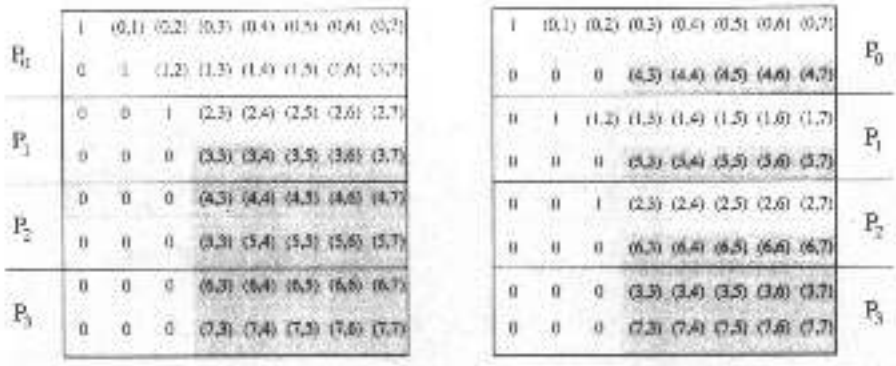
图8-8 对于用块一维划分的分布在4个进程间的8 × 8矩阵，
对应于$k = 3$的高斯消元迭代中的通信

图8-9a表示使用一维块划分时，一个所有行都属于矩阵参与部分的进程，在第k次迭代的消元步骤中执行$(n-k-1)n/p$次乘法和减法运算。注意，在最后第$(n/p)-1$次迭代中，没有进程
360 包含所有参与运算的行，但是我们忽略这个异常情况。如果使用这个算法的流水线版本，则第k次迭代中的最大负载进程的算术运算次数$(2(n-k-1)/p)$，要远大于同一次迭代中一个进程的通信字数$(n-k-1)$。因此，对相对于p充分大的n，在每次迭代中计算相对于通信来说是主要的。假设每次标量的乘法与减法运算需要一个单位时间，那么这个算法总的并行运行时间（不考虑通信开销）为$2(n/p)\sum_{k=0}^{n-1}(n-k-1)$，近似等于$n^3/p$。

即使不考虑该算法的通信成本，其进程时间乘积也是n^3。因此，并行算法的成本高于串行运行时间（公式(8-17)），为它的3/2倍。对于一维块划分，高斯消元法的这种低效率是由负载分配不平衡造成的进程空闲引起的。如图8-9a所示，对于8 × 8矩阵和4个进程，在对应于$k = 3$的迭代（在算法8-4的外部循环）中，一个进程完全空闲，一个进程部分空闲，只有两个进程在满负荷工作。在外层循环的迭代次数完成一半时，仅仅有一半的进程在工作。进程空闲使并行算法比串行算法的成本更大。

如果在使用图8-9b所示的循环一维映射的进程中划分矩阵，就可以缓解这个问题。使用

循环一维划分。在任何迭代中，最大与最小负载进程的计算负载量最多相差一行（即算术运算量为$O(n)$）。由于有n次迭代，使用一维循环映射时，由进程空闲引起的累积开销仅为$O(n^2p)$，而使用块映射时的累积开销为$\Theta(n^3)$（习题8.12）。



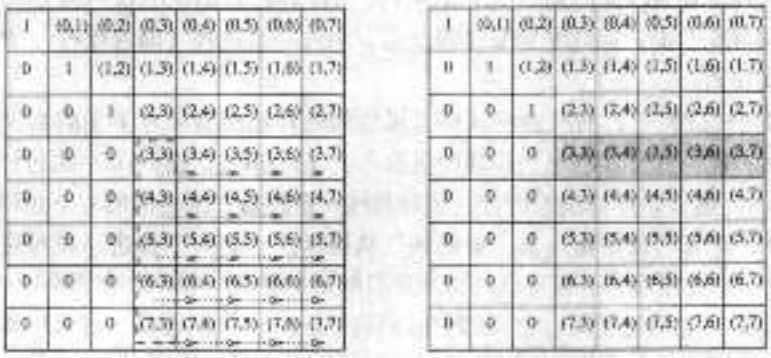
a) 块一维映射　　b) 循环一维映射

图8-9　在$k=3$的高斯消元迭代中，对4个进程的8×8矩阵使用块和循环一维划分时不同进程上的计算负载

2. 二维划分的并行实现

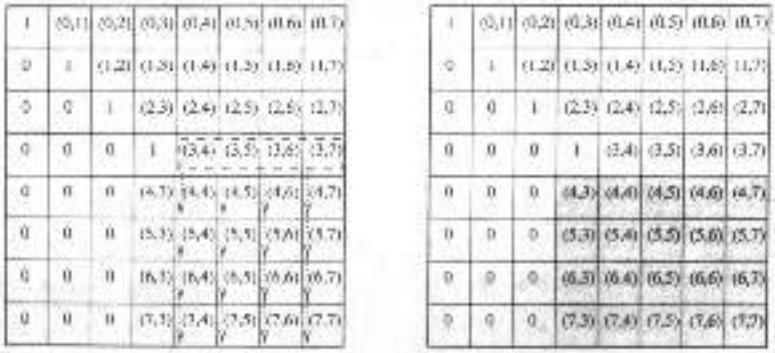
现在我们来介绍算法8-4的并行实现，其中$n\times n$矩阵A映射到这样一个P_{ij}进程的$n\times n$进程格网，进程最初P_{ij}存储$A[i,j]$。图8-10举例说明$n=8$时与$k=3$对应的外层循环迭代中的计算与通信步骤。算法8-4、图8-5和8-10表示在外层循环的第k次迭代中，进程$P_{k,k+1}$，$P_{k,k+2}$，…，$P_{k,n-1}$需要用$A[k,k]$分别除$A[k,k+1]$，$A[k,k+2]$，…，$A[k,n-1]$。在完成第6行的除法后，用第k行修改后的元素对矩阵参与部分的所有其他各行执行消元步骤。类似地，用第k列的元素对矩阵参与部分的所有其他各列执行消元步骤。如图8-10所示，第k次迭代中的通信需要$A[i,k]$沿第i行进行一对多广播（图8-10a），$k\leqslant i<n$，$A[k,j]$沿第j列进行一对多广播（图8-10c），$k<j<n$。如果这些广播在所有的进程上同步进行，则与一维划分的情形一样，它也不是成本最优的并行公式（习题8.11）。

361



a) 按行广播$A[i,k]$，$(k-1)<i<n$　　b) $A[k,j]:=A[k,j]/A[k,k]$，$k<j<n$

图8-10　对于分配在逻辑二维格网上64个进程的8×8矩阵，与$k=3$对应的高斯消元法迭代的计算步骤



c) 按列广播$A[k,j]$, $k<j<n$　　d) $A[i,j]=A[i,j]-A[i,k]\times A[k,j]$, $k<i<n$且$k<j<n$

图8-10 （续）

流水线通信与计算 根据前面对系数矩阵使用一维划分进行高斯消元的经验，我们给出一个使用二维划分算法的流水线版本。

如图8-10所示，在外层循环的第k次迭代（算法8-4的3－16行）中，$A[k,k]$从$P_{k,k}$向右传送到$P_{k,k+1}$，到$P_{k,k+2}$，…，直到到达$P_{k,n-1}$。$P_{k,k+1}$从$P_{k,k}$收到$A[k,k]$后立即进行除法运算$A[k,k+1]/A[k,k]$。在此之前，不需要一直等待$A[k,k]$到达$P_{k,n-1}$，类似地，随后第k行的各进程$P_{k,j}$也在收到$A[k,k]$后立即进行除法运算。除法运算之后，$A[k,j]$就可以在第j列中向下通信。在$A[k,j]$向下移动的过程中，它经过的每个进程都可以使用它进行计算。第j列的各进程不需要一直等待$A[k,j]$到达该列的最后一个进程。因此，只要$A[i,k]$和$A[k,j]$到达，$P_{i,j}$就执行消元步骤$A[i,j]:=A[i,j]-A[i,k]\times A[k,j]$，对于给定的迭代，一些进程比其他进程更早开始进行运算，所以它们也很快开始后继的迭代运算。

362

可以用几种方法以流水线方式进行通信与计算。我们提出一个如图8-11所示的方案。在图8-11a中，当$P_{0,0}$把$A[0,0]$送到$P_{0,1}$时，在进程$P_{0,1}$开始$k=0$的外层循环迭代。$P_{0,1}$收到$A[0,0]$后执行除法计算$A[0,1]:=A[0,1]/A[0,0]$（图8-11b）。$P_{0,1}$把$A[0,0]$转发到$P_{0,2}$，并且把更新后的$A[0,1]$向下传送到$P_{1,1}$（图8-11c）。同时$P_{1,0}$把$A[1,0]$传送到$P_{1,1}$。$P_{1,1}$收到$A[0,1]$和$A[1,0]$后执行消元步骤$A[1,1]:=A[1,1]-A[1,0]\times A[0,1]$，而$P_{0,2}$收到$A[0,0]$执行除法步骤$A[0,2]:=A[0,2]/A[0,0]$（图8-11d）。在这个计算步骤之后，另外一组进程（即进程$P_{0,2}$、$P_{1,1}$和$P_{2,0}$）就开始通信（图8-11e）。

在特定的迭代过程中，所有进行计算与通信的进程都沿着从左下角到右上角的对角线方向排列（例如，图8-11e中执行通信步骤的进程$P_{0,2}$、$P_{1,1}$和$P_{2,0}$，图8-11f中执行计算步骤的进程$P_{0,3}$、$P_{1,2}$和$P_{2,1}$）。随着并行算法的进行，该对角线向着逻辑二维格网的右下角移动。因此在每次迭代过程中，计算与通信像一条“前沿线”从格网左上到右下推进。在对应某个迭代的这样一条线通过一个进程后，进程可以自由地进行随后的迭代。例如，在图8-11g中，与$k=0$对应的线通过$P_{1,1}$后，它通过把$A[1,1]$传送到$P_{1,2}$启动对应于$k=1$的迭代，这又启动对应于$k=1$的线，并且紧挨着与$k=0$对应的线。类似地，与$k=2$对应的第三条线从$P_{2,2}$开始（图8-11m）。因此，对应于不同迭代的多条这样的线是同时移动的。

迭代的每个步骤，比如除法、消元、或传递一个数据到相邻的进程，都是时间不变的操作。因此，在恒定的时间内，这样的线向着矩阵的右下角移动一步（等价于图8-11的两步）。

与$k=0$对应的线在$P_{0,0}$启动后到达$P_{n-1,n-1}$所花的时间为$\Theta(n)$。对于外层循环的n次迭代，该算法形成n条线，每条线比前一条线延后一步。因此在第一条线之后，最后一条线通过矩阵的右下角要经过$\Theta(n)$步。在$P_{0,0}$开始的第一条线到最后一条线完成所花的时间是$\Theta(n)$。在最后一条线通过矩阵的右下角后这个过程才算完成，因此总的并行运行时间为$\Theta(n)$。由于使用n^2个进程，流水线高斯消元法版本的成本为$\Theta(n^3)$，与算法的串行运行时间相同。由此可见用二维划分的流水线高斯消元法版本是成本最优的。

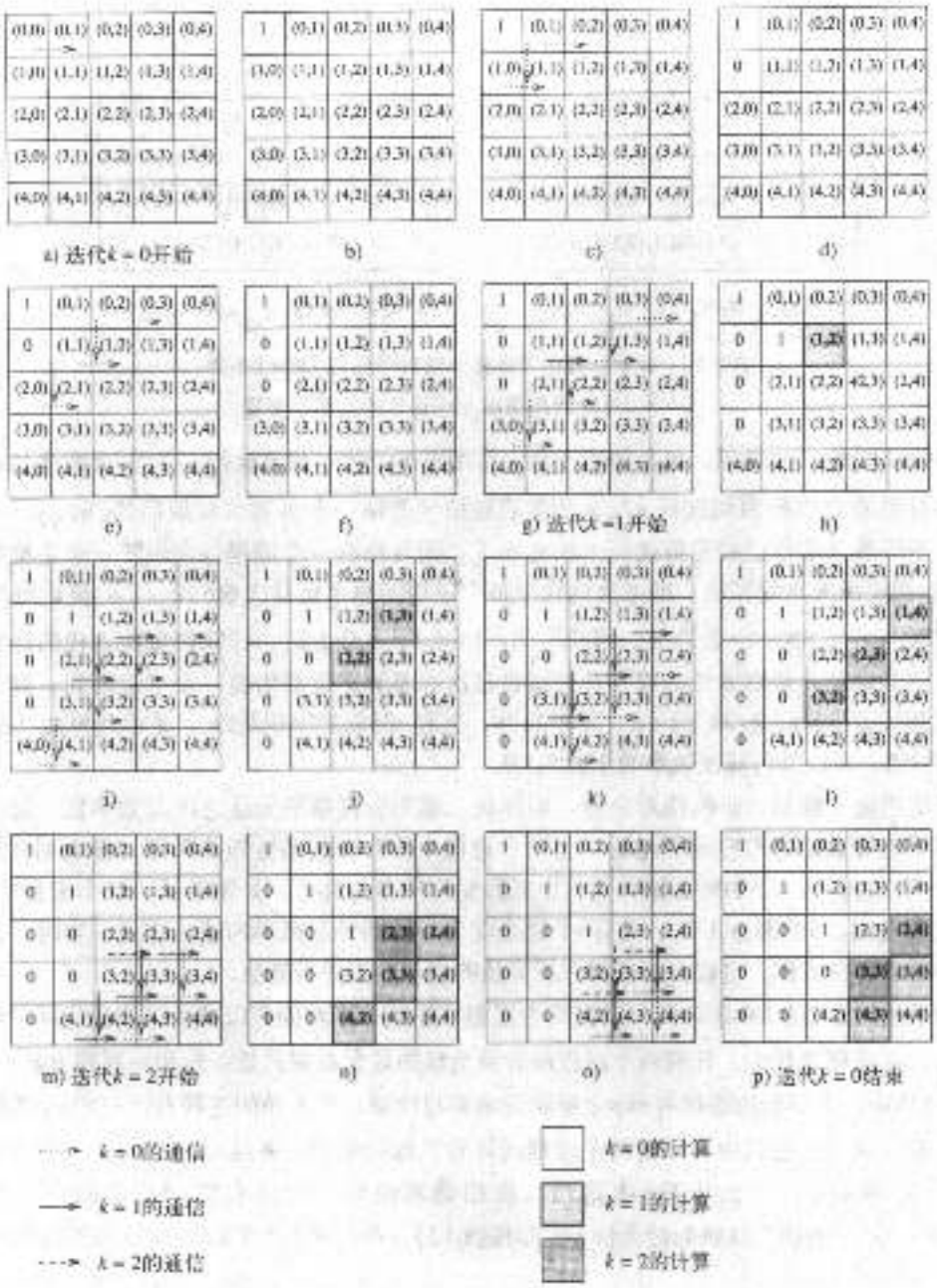
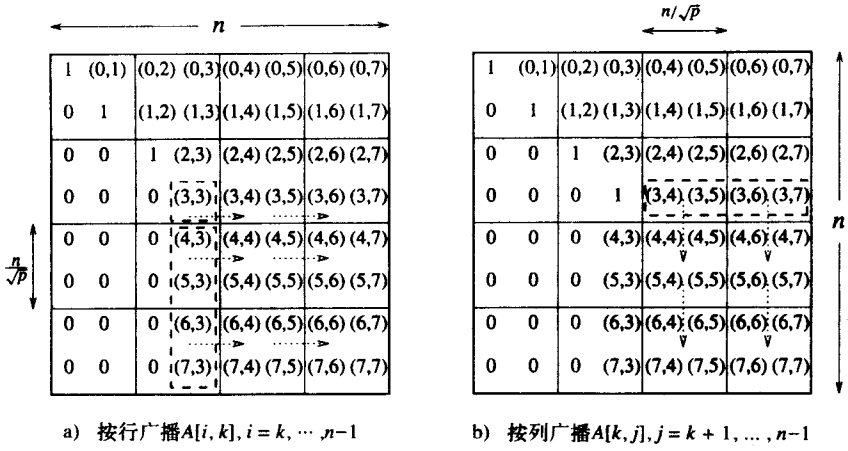


图8-11 用25个进程的5 × 5矩阵的流水线高斯消元

进程个数小于n^2的二维划分 考虑进程的个数$p < n^2$的情形，并用块二维划分把矩阵映射到$\sqrt{p} \times \sqrt{p}$格网。图8-12说明，典型的并行高斯迭代包括$n/\sqrt{p}$个值的按行和按列通信。图8-13a说明$n = 8$、$p = 16$时的块二维划分方法的负载分布。



a) 按行广播$A[i,k], i = k, \cdots, n-1$　　b) 按列广播$A[k,j], j = k+1, \ldots, n-1$

图8-12 对于分布在二维格网的16个进程的8×8矩阵，与$k = 3$对应的高斯消元法迭代中的通信步骤

图8-12和图8-13a说明，包含矩阵一个完整参与部分的进程要执行$n^2/\sqrt{p}$次乘法与减法运算，并且沿着它的行和列进行$n/\sqrt{p}$个字的通信（忽略一个事实：在最后的$(n/\sqrt{p})-1$次迭代中，矩阵参与部分已经变得比一个块更小了，而且没有一个进程包含矩阵一个完整参与部分）。如果使用流水线算法，则每次迭代中每个进程的算术运算次数$(2n^2/\sqrt{p})$要比每个进程通信字数$(n/\sqrt{p})$高一个数量级。因此，当n^2相对于p充分大时，计算在每次迭代的计算与通信中占主导地位。忽略通信的成本并且假设每次标量算术运算需要一个单位时间，则这个算法的总并行运行时间是$(2n^2/p) \times n$，等于$2n^3/p$。进程-时间的乘积是$2n^3$，是串行算法（公式(8-17)）的3倍。所以并行算法效率的上界为1/3。

与使用块一维划分时的情形一样，矩阵块二维划分高斯消元法之所以效率低，是由负载分配不平衡导致的进程空闲引起的。当16个进程中的8×8的系数矩阵采用二维块划分时，图8-13a显示对应于$k = 3$的外层循环迭代中系数矩阵的参与部分。如图所示，16个进程中有7个进程完全空闲，5个部分工作，只有4个完全在工作。到外层循环的迭代完成一半时，只有四分之一的进程在工作。进程空闲使得并行算法的成本高于串行算法。

如果使用如图8-13b所示的二维循环方式划分矩阵，则空闲可以减少。使用循环二维方式划分时，在任何迭代中，任何两个进程间计算负载的最大差别只是一行和一列的更新。例如，在图8-13b中，右下角的进程有n^2/p个矩阵元素参与计算，左上角的进程有$(n-1)^2/p$个矩阵元素参与计算。在任何迭代中，任何两个进程间计算负载的差别最多是$\Theta(n/\sqrt{p})$，它对开销函数的贡献是$\Theta(n/\sqrt{p})$。由于有n次迭代，使用循环映射时由进程空闲造成的累积开销只有$\Theta(n^2/\sqrt{p})$，而使用块映射时为$\Theta(n^3)$（习题8.12）。在高斯消元和LU因式分解的实际并行实现中，用块循环映射减少由于和单纯循环映射相关的消息启动时间引起的开销，并通过执行块矩阵操作获得更好的串行CPU利用率（习题8.15）。

363
|
365

1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6) (0,7)
0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6) (1,7)
0	0	1	(2,3)	(2,4)	(2,5)	(2,6) (2,7)
0	0	0	(3,3)	(3,4)	(3,5)	(3,6) (3,7)
0	0	0	(4,3)	(4,4)	(4,5)	(4,6) (4,7)
0	0	0	(5,3)	(5,4)	(5,5)	(5,6) (5,7)
0	0	0	(6,3)	(6,4)	(6,5)	(6,6) (6,7)
0	0	0	(7,3)	(7,4)	(7,5)	(7,6) (7,7)

a) 块方格映射

1	(0,4)	(0,1)	(0,5)	(0,2)	(0,6)	(0,3) (0,7)
0	(4,4)	0	(4,5)	0	(4,6)	(4,3) (4,7)
0	(1,4)	1	(1,5)	(1,2)	(1,6)	(1,3) (1,7)
0	(5,4)	0	(5,5)	0	(5,6)	(5,3) (5,7)
0	(2,4)	0	(2,5)	1	(2,6)	(2,3) (2,7)
0	(6,4)	0	(6,5)	0	(6,6)	(6,3) (6,7)
0	(3,4)	0	(3,5)	0	(3,6)	(3,3) (3,7)
0	(7,4)	0	(7,5)	0	(7,6)	(7,3) (7,7)

b) 循环方塔映射

图8-13 对于与$k=3$对应的高斯消元迭代，在8×8矩阵到16个进程的块映射和循环二维映射中不同进程的计算负载

综上所述，我们得到以下结论：对于$n\times n$矩阵，在p个进程上使用一维和二维划分方案，流水线并行高斯消元法花费的时间为$\Theta(n^3/p)$。对于$n\times n$系数矩阵，二维划分比一维划分可以使用更多的进程($O(n^2)$)，一维划分使用进程的数量级为$O(n)$。因此，二维划分方法更具可扩展性。

8.3.2 带部分主元选择的高斯消元算法

在算法8-4中，如果系数矩阵的任何主对角线元素$A[k,k]$接近或者等于零，高斯消元法算法就是无效的。为了避免这个问题，保证算法的数值稳定性，我们使用一个称为部分主元选择（partial pivoting）的技术。在第k次迭代的外层循环开始时，这个方法选择一个列i（称为主元列）使$A[k,i]$在数量上是所有$A[k,j]$ ($k\leqslant j<n$)中最大的。然后在开始迭代前交换第k列与第i列。可以用两种方式进行交换：一种是显式交换，即将这些列从物理上交换位置；一种是隐式交换，即维护一个$n\times1$置换向量，记录矩阵A中列的新下标。如果用隐式列下标交换实现部分主元选择，那么因式L和U并不恰好是三角矩阵，而是三角矩阵的按列置换。 366

假设对列进行显式交换，则在第k次迭代中，交换第k列与第i列之后，算法8-4中第6行用来做除数的$A[k,k]$的值大于或者等于在k次迭代中被它除的任何$A[k,j]$的值。算法8-4中的部分主元选择产生一个单位上三角矩阵，其中主对角线以上的所有元素的绝对值都小于1。

1. 一维划分

如8.3.1节讨论的那样，进行部分主元选择在按行向划分时是直截了当的。在第k次迭代中进行除法运算之前，储存第k行的进程在该行的参与部分中进行比较，选择绝对值最大的元素作除数。这个元素确定主元列，而所有的进程都必需知道这一列的下标，这个信息可以沿第k行（经过除法后）修改的元素传送到其他进程。与不用主元选择的高斯消元法一样，在第k次迭代中，主元搜索与除法步骤结合需要的时间为$\Theta(n-k-1)$。因此，如果矩阵是按行划分的，则部分主元选择对算法8-4的性能就没有什么影响。

现在考虑系数矩阵的按列一维划分。不用主元选择时，高斯消元法的并行实现用按行一维划分与按列一维划分来说几乎是一样的。然而，使用部分主元选择时两者则有重大差别。

第一个差别在于，与按行划分不同，主元搜索是按列划分分布的。对$n\times n$矩阵，如果进

程的个数是p，则按列划分的主元搜索分为两步。对于第k次迭代的主元搜索中，每个进程首先确定它所存储的第k行的n/p（或者更少）个元素的最大值。下一步是找出所产生的p（或者更少）个值中的最大值，并且把这个最大值分布到所有的进程上。每次主元搜索花费的时间是$\Theta(n/p) + \Theta(\log p)$。当$n$相对于$p$充分大时，$\Theta(n/p) + \Theta(\log p)$小于$\Theta(n)$，其中$\Theta(n)$是采用按行划分时执行主元搜索花费的时间。这似乎表明在部分主元选择中，按列划分要优于按行划分。但是，下面的因素有利于按行划分。

图8-7说明在使用按行一维划分的流水线高斯消元法中，通信与计算“前沿线”是如何从上到下移动的。类似地，在按列一维划分的情形，通信与计算“前沿线”从左到右移动。这意味着，在与第k次迭代对应的“前沿线”到达最右边的进程之前，第$k+1$行不能进行与第$k+1$次迭代对应的主元搜索（因为它的更新没有完成）。因此在第k次迭代全部完成之前，不
367 能开始第$k+1$次迭代。这在实际上是消除流水线，因此不得不使用效率低下的同步版本。

当进行部分主元选择时，系数矩阵的列可以显式交换，也可以不显式交换。在任何一种情况下，按列一维划分都不利于算法8-4的性能。回忆在高斯消元法中，循环映射或者块循环映射的负载平衡都要比块映射好。在高斯消元法的每一阶段，循环映射保证矩阵的参与部分几乎均匀地在各进程中分布。如果主元列不是显式交换的，则这个条件可能不成立。使用主元列以后，该列不再保留在矩阵的参与计算部分内。由于主元选择没有显式交换，这些列就可以从矩阵不同进程的参与计算部分中任意地删除。这样的任意性可能干扰参与计算部分的均匀分布。另一方面，如果属于不同进程的列进行显式交换，则这个交换需要进程之间的通信。如果不进行列的显式交换，则按行一维划分既不需要因为交换列而进行通信，也不会失去负载平衡。

2. 二维划分

对系数矩阵进行二维划分时，部分主元选择即使没有完全抵消流水线的作用，却也严重地限制了它的作用。回忆在采用二维划分的流水线高斯消元法中，对应于各个迭代的“前沿线”从左上向右下移动。当对应于第k次迭代的“前沿线”的移动越过参与矩阵中连接左下与右上的对角线时，第$k+1$次迭代的主元搜索可以立即开始。

因此，在采用二维划分的并行高斯消元法中，部分主元选择可能导致相当大的性能下降。如果数值条件允许，由部分主元选择引起的性能损失有可能减少。在第k次迭代中，我们可以把对选主元的搜索限制在q个列的范围（而不是所有$n-k$列）。在这种情况下，如果$A[k,i]$是第i行参与部分的q个元素中的最大元素，则选择第i列作为第k次迭代的主元。这个受到限制的部分主元选择不仅减少通信开销，也允许有限的流水线。把主元搜索的列数限制为q，只要前一次迭代更新了前$q+1$列就可以开始下一次迭代。

使用4.7.1节所描述的一对多广播的快速算法，可以避免用二维划分的高斯消元法中由于部分主元选择造成的流水线损失。对p个进程的$n \times n$系数矩阵采用二维划分时，在流水线版本的高斯消元法的每次迭代中，一个进程在通信中花费的时间为$\Theta(n/\sqrt{p})$。忽略消息的启动时间t_s，一个非流水线版本的高斯消元法当其使用4.1节中的算法执行显式一对多广播时，每次迭代中花费的通信时间为$\Theta((n/\sqrt{p})\log p)$。这个通信时间高于流水线版本。4.7.1节描述的一对多广播算法每次迭代花费的时间为$\Theta(n/\sqrt{p})$（忽略启动时间）。这个时间渐近地等于流水线
368 算法每次迭代的通信时间 。因此，使用巧妙的算法进行一对多广播，即使采用非流水线并行高斯消元法，也能达到能与流水线算法可比的性能。但是，4.7.1节中讲述的一对多广播算法

把消息分成更小的部分并分开发送。为了使这些算法有效，消息容量应该足够大，即n对p而言应该足够大。

虽然在采用二维划分的高斯消元法中，流水线与主元选择不能同时使用，但是本节讨论的二维划分仍然是有用的。稍做修改，它就可以用于Cholesky因式分解算法中（习题8.16的算法8-6），且不需要主元选择。Cholesky因式分解只用于对称的正定矩阵。一个$n \times n$的实矩阵A称为正定的（positive definite），如果对于任意的$n \times 1$非零实向量x，矩阵A满足$x^T Ax > 0$。Cholesky因式分解中的通信模式与高斯消元法的通信模式非常相似（习题8.16），所不同的是，由于矩阵上下三角的对称性，Cholesky因式分解只使用矩阵一半的三角。

8.3.3 求解三角系统：回代法

我们现在简单地讨论求解线性方程组的第二个阶段。当矩阵A被简化成主对角线元素为1的上三角矩阵U之后，我们用回代法确定向量x。算法8-5为求解上三角方程组$Ux = y$的串行回代法算法。

从最后一个方程开始，算法8-5的主循环（3～8行）的每次迭代计算变量的值，并把变量的值回代到剩余方程中。程序大约需要执行$n^2/2$次乘法和减法运算。注意，在回代法中，算术运算次数要比高斯消元法中的次数小一个$\Theta(n)$个倍数。因此，如果将回代法与高斯消元法结合使用，就可以采用使并行高斯消元法最有效的矩阵划分方案。

算法8-5 回代法的串行算法，U是主对角线元素为1和次对角线元素为零的上三角矩阵

```
1.  procedure BACK_SUBSTITUTION (U, x, y)
2.  begin
3.      for k := n − 1 downto 0 do  /* Main loop */
4.          begin
5.              x[k] := y[k];
6.              for i := k − 1 downto 0 do
7.                  y[i] := y[i] − x[k] × U[i, k];
8.          endfor;
9.  end BACK_SUBSTITUTION
```

369

考虑对p个进程的$n \times n$矩阵U的按行的块一维映射。令向量y在所有的进程中均匀分布。在主循环（第3行）一次典型迭代中解出的变量值必须送到使用包含变量的方程的所有进程中。这个通信可以以流水线方式实现（习题8.22）。如果这样，则在迭代中，与通信时间相比，计算时间是主要的。在流水线实现的每次迭代中，每个进程收到（或者产生）一个变量的值，并且把它送到另一个进程。进程使用当前迭代求出的变量值执行多达n/p次乘法和减法（第6、7行）。因此流水线实现的每一步需要的通信时间为常数，而计算时间为$\Theta(n/p)$。算法执行$\Theta(n)$步（习题8.22）终结，整个算法的并行运行时间为$\Theta(n^2/p)$。

如果使用对进程的$\sqrt{p} \times \sqrt{p}$逻辑格网上的二维划分对矩阵进行划分，并且向量的元素沿进程格网的列分布，则只有包含向量的$\sqrt{p}$个进程进行所有的计算。使用流水线方式，将U的适当元素传送到包含y的相应元素的进程，以进行置换步骤（第7行），算法能在时间$\Theta(n^2\sqrt{p})$内完成（习题8.22）。因此，采用二维映射的并行回代法的成本为$\Theta(n^2\sqrt{p})$。由于算法的串行成本仅为$\Theta(n^2)$，算法不是成本最优的。然而，对于$\sqrt{p} = O(n)$，求解线性方程组的整个过程

（包括使用高斯消元法进行矩阵的上三角化）仍然是最优的，因为整个过程的串行复杂度是$Q(n^3)$。

8.3.4 求解线性方程组时的数值因素

对于$Ax = b$型的线性方程组，可以把A表示成一个下三角矩阵L和一个单位上三角矩阵U的乘积，用因式分解算法求解。于是方程组可以写成$LUx = b$，然后分两步求解。首先解下三角方程组$Ly = b$得到y，然后解上三角方程组$Ux = y$得到x。

算法8-4中给出的高斯消元算法能够有效地把A分解成L和U。但是，它也可以用第7、13行的步骤解下三角方程组$Ly = b$。算法8-4中给出称为*面向行*（row-oriented）的高斯消元算法。在这个算法中，行与数的相乘与其他行相减。如果如8.3.2节所叙述的那样，把部分主元选择与算法结合，则得到所有元素的值都小于或者等于1的上三角矩阵U。下三角矩阵L不论显式还是隐式，可以有数值更大的元素。求解方程组$Ax = b$时，首先解下三角方程组$Ly = b$。如果L包含大的元素，则求解y时，由于计算机浮点数的精度有限，会产生舍入误差。y中这些误差
370 通过方程组$Ux = y$的解传播。

通过把行和列的角色互换，可以从算法8-4得出*面向列*（column-oriented）的高斯消元算法。在面向列的算法中，从其他列中减去列与数的相乘，主元搜索也沿着列进行，如果需要，可以用行交换保证数值的稳定性。面向列算法产生的下三角矩阵L的所有元素的值都小于或者等于1，这使得求解方程组$Ly = b$产生的数值误差达到最小，而且使全部解产生的误差小于面向行的算法。算法3.3给出面向列的LU分解的过程。

从实际的观点来看，面向列的高斯消元算法比面向行的高斯消元算法更有用。本章我们之所以选择面向行的高斯消元算法作详细叙述，是因为它更为直观。显然，从其他方程中减去一个方程的倍数所得到的线性方程组与原方程组等价。通过转换行与列的角色，本节讨论的算法8-4中面向行的算法都可以应用到面向列的算法中。例如，对于用部分主元选择的面向列的算法，按列一维划分比按行一维划分更合适。

8.4 书目评注

采用一维划分的矩阵转置问题的实质是多对多私自通信问题[Ede89]。因此，第4章中关于多对多个性化通信的所有参考文献都可以应用到矩阵转置中。递归转置算法（通称为RTA）是Eklundh [Ekl72]最早提出来的。Bertsekas和Tsitsklis[BT97]，Fox和Furmanski[FF86]，Johnsson[Joh87]，以及McBryan和Van de Velde[MdV87]将它用在超立方体中，用于每个进程的单一端口通信。Johnsson[Joh87]还通过允许在所有通道上进行通信的超立方体讨论并行RTA。Ho和Raghunath[HR91]，Johnsson和Ho[JH88]，Johnsson[Joh90]以及Stout和Wagar[SW87]对超立方体的RTA提出了进一步改进。

现在可以找到许多并行稠密线性代数算法的来源，包括矩阵-向量乘法，矩阵乘法[CAHH9，GPS90，GL96a，Joh87，Mod88，OS85]。由于稠密矩阵乘法是计算高度密集型的，研究这些算法的并行形式，并在不同的并行体系结构上测试它们的性能具有很大的意义[Akl89，Ber89，CAHH91，Can69，Cha79，CS88，DNS81，dV89，FJL+88，FOH87，GK91，GL96a，Hip89，HJE91，Joh87，PV80，Tic88]。Cannon[Can69]，Dekel，Nassimi和Sahni[DNS81]，Fox et al. [FOH87]等开发了一些早期的矩阵乘法并行公式。Berntsen [Ber89]，

以及Ho，Johnsson 和Edelman[HJE91]对这些算法提出变体和作了改进。尤其是Berntsen [371]
[Ber89] 提出一个算法，比Cannon算法的通信开销要小，但是并发度也小。Ho，Johnsson 和Edelman[HJE91]针对超立方体提出Cannon算法的另外一个变体，允许在所有通道上同时进行通信。这个算法在减少通信的同时也降低并发度。Gupta和Kumar[GK91]对几种矩阵乘法算法进行详细的可扩展性分析。他们对p个进程的超立方体和n, p及硬件相关常数的不同取值范围，就如何确定两个$n \times n$矩阵相乘的最好算法给出一种分析。他们还证明由Berntsen和Ho等人提出的改进算法并没有改进超立方体上矩阵乘法的整体可扩展性。

一些研究人员[Ber84，BT97，CG87，Cha87，Dav86，DHvdV93，FJL+88，Gei85，GH86，GPS90，GR88，Joh87，LD90，Lei92，Mod88，Mol86，OR88，Ort88，OS86，PR85，Rob90，Saa86，Vav89]研究了关于LU分解和稠密线性方程组求解的并行算法。Geist和Heath[GH85，GH86]以及Heath[Hea85]专门研究了稠密Cholesky因式分解的并行算法。三角方程组求解的并行算法在[EHHR88，HR88，LC88，LC89，RO88，Rom87]中作了详细的研究。Demmel，Heath和van der Vorst[DHvdV93]对考虑数值含义的并行矩阵计算作出详细而全面的综述。

本章中讨论的所有矩阵和向量运算，以及更多的操作，可找到可移植的软件实现，称为PBLAS（并行的基本线性代数子程序）[C+95]。ScaLAPACK程序库[B+97]使用PBLAS实现多种有实际意义的线性代数子程序，其中包括矩阵因式分解和线性方程组求解的各种方法的过程。

习题

8.1 考虑4.5.3节用于多对多私自通信的两个算法。如果$t_s = 100\mu s$, $t_w = 1\mu s$，在有64个节点和对分宽度为$\Theta(p)$的并行计算机上，使用一维划分对一个1024 × 1024矩阵进行转置，你将使用哪种方法？为什么？

8.2 给出一个矩阵-向量乘法的并行形式，其中对矩阵按列进行一维块划分，向量被均匀地分布到所有的进程中。证明并行运行时间与按行一维块划分的情形相同。

提示：在按列一维划分中使用的基本通信操作为多对多归约，与按行一维划分时的多对多广播相反。习题4.8描述多对多归约。

8.3 8.1.2节描述和分析二维划分情况下矩阵-向量乘法。如果 $n >> \sqrt{p}$，则给出的方法将并行运行时间改进为 $n^2/p + 2t_s \log p + 2t_w(n/\sqrt{p})$。改进的方法比8.1.2节使用的方法有更好的 [372]
可扩展性吗？

8.4 在p个进程中进行采用二维划分的$n \times n$矩阵与$n \times 1$向量相乘，其开销函数是$t_s p \log p + t_w n\sqrt{p} \log p$（公式(8-8)）。把这个表达式代入公式(5-14)中，得到关于n的二次方程。用这个方程确定并行算法的精确等效率函数，并与公式(8-9)和(8-10)进行比较。比较结果是否改变与t_w相关的项确定这个并行算法整体等效率函数的结论？

8.5 Strassen的关于矩阵相乘的方法[AHU74，CLR90]是一个基于分治技术的算法。使用Strassen的算法，两个$n \times n$矩阵乘法计算的串行复杂度是$\Theta(n^{2.81})$。对于用p个进程的两个$n \times n$矩阵乘法计算，考虑简单矩阵相乘算法（8.2.1节）。假设在每个进程上用Strassen算法执行 $n/\sqrt{p} \times n/\sqrt{p}$ 子矩阵的乘法。试推导这个算法的并行运行时间表达式。该并行算法是成本最优的吗？

8.6 （不超过n^3个进程的DNS算法[DNS81]）8.2.3节描述使用不超过n^3个进程的DNS算法的一个并行形式。该算法的一个变体使用$p = n^2q$个进程，其中$1 \leqslant q \leqslant n$。这里进程排列成称为“超级进程”的$q \times q \times q$逻辑三维阵列，其中每个“超级进程”都是一个$(n/q) \times (n/q)$进程格网。现在除了假设每个进程是一个$(n/q) \times (n/q)$的逻辑进程格网外，该变体可以看成与8.2.3节中描述的块变体相同。这就意味着$(n/q) \times (n/q)$子矩阵的每次块相乘都是在$(n/q)^2$个进程中并行执行的，而不是只在一个进程中执行。8.2.1节或8.2.2节中介绍的任何算法都可以用来执行这样的乘法。

对于DNS算法的这个变体推导出用n, p, t_s, t_w表示的并行运行时间表达式。用它与公式(8-16)比较。讨论使用不超过n^3个进程时，DNS算法的两个变体的优点和缺点。

8.7 图8-7说明，对于在5个进程上一个按行划分的5×5矩阵，流水线版本的高斯消元法需要16个步骤。证明，在一般情况下，把一个$n \times n$矩阵按行划分，一个进程分配一行，则完成如图所示的算法需要$4(n-1)$个步骤。

8.8 如果在p个进程中，把$n \times n$系数矩阵按列划分，试详细描述不用主元选择的算法8-4中高斯消元算法的并行实现。请考虑流水线和非流水线两种实现方式。也考虑$p = n$和$p < n$时的情形。

提示：8.3.1节描述的高斯消元法的并行实现说明在进程的逻辑二维格网上的横向与纵向
373 通信（图8-12）。一个按行划分只需要这个通信的纵向部分。类似地，一个按列划分只需要这个通信的横向部分。

8.9 推导出习题8.8中所有实现的并行运行时间公式。这些并行实现的运行时间与采用按行一维划分时的并行运行时间有显著差异吗？

8.10 使用部分主元选择，重做习题8.9。在采用按行划分和按列划分的两种实现中，并行运行时间有显著差异吗？

8.11 证明：如果$2n$次一对多广播同步执行，对在进程的$n \times n$逻辑格网上采用二维划分的$n \times n$矩阵的高斯消元法不是成本最优的。

8.12 证明：对于块映射的高斯消元算法，所有进程的累积空闲时间为$\Theta(n^3)$。不论对$n \times n$矩阵的划分是沿一维还是二维，证明使用循环一维划分的空闲时间减少到$\Theta(n^2p)$，循环二维划分的空闲时间减少到$\Theta(n^2\sqrt{p})$。

8.13 证明：如果不用主元选择，采用二维映射的高斯消元法异步版本的等效率函数是$\Theta(p^{3/2})$。

8.14 如果$n \times n$系数矩阵在逻辑方形二维格网的p个进程中按下面的格式划分，分别推导出带部分主元选择和不带部分主元选择的高斯消元法并行运行时间的精确表达式：

A. 按行块一维划分；B. 按行循环一维划分；C. 按列块一维划分；D. 按列循环一维划分。

8.15 按照8.2节开始讨论的块矩阵操作方法重写算法8-4。考虑把$n \times n$矩阵划分成$q \times q$子矩阵阵列的高斯消元法，其中每个子矩阵的大小为$n/q \times n/q$。这些块阵列以循环的方式映射到进程的一个逻辑$\sqrt{p} \times \sqrt{p}$格网上，导致一个从原矩阵到格网的二维块循环映射。假设$n > q > \sqrt{p}$，并且q可整除n，$\sqrt{p}$可整除q。试推导高斯消元法同步版本和流水线版本并行运行时间的表达式。

提示：除法步骤$A[k,j] := A[k,j]/A[k,k]$用子矩阵运算$A_{k,j} := A_{k,k}^{-1}A_{k,j}$代替，其中$A_{k,k}$是第$k$个

对角子矩阵的下三角部分。

8.16 （Cholesky因式分解）算法8-6描述把一个对称正定矩阵分解为$A = U^T U$形式的Cholesky因式分解算法的面向行的版本。Cholesky因式分解不需要主元选择。推导在进程的方形格网上使用二维划分的这个算法的流水线并行公式。画一个与图8-11类似的图。 374

算法8-6 面向行的Cholesky因式分解算法

```
1.  procedure CHOLESKY (A)
2.  begin
3.     for k := 0 to n − 1 do
4.        begin
5.           A[k, k] := √A[k, k];
6.           for j := k + 1 to n − 1 do
7.              A[k, j] := A[k, j]/A[k, k];
8.           for i := k + 1 to n − 1 do
9.              for j := i to n − 1 do
10.                A[i, j] := A[i, j] − A[k, i] × A[k, j];
11.       endfor;        /* Line 3 */
12. end CHOLESKY
```

8.17 （按比例加速比）按比例加速比（5.7节）定义为当问题的规模随着进程数的增加而线性增加时获得的加速比；也就是说，如果W表示一个进程上问题的基本规模，则

$$\text{按比例加速比} = \frac{Wp}{T_P(Wp, p)} \tag{8-19}$$

根据8.2.1节描述的简单矩阵相乘算法，对于16 × 16矩阵相乘的基本问题画出标准曲线和加速比曲线，其中$p = 1, 4, 16, 64, 256$。假设在公式(8-14)中$t_s = 10, t_w = 1$。

8.18 对于习题8.17，画出第三条加速比曲线，其中问题的规模按等效率函数$\Theta(p^{3/2})$的比例增长。同样取$t_s = 10, t_w = 1$。

提示：这种方法的按比例加速比为

$$\text{按等效率比例加速比} = \frac{Wp^{3/2}}{T_P(Wp^{3/2}, p)}$$

8.19 对于标准加速比曲线（习题8.17）、按比例加速比曲线（习题8.17）以及当问题规模随等效率函数增加时（习题8.18）的加速比曲线分别画出对应的简单矩阵相乘算法的效率曲线。

8.20 只增加进程的数量而不增加总工作负载的缺点是，加速比不能随着进程数量的增加而线性增加，而效率却单调下降。根据你从习题8.17和8.19获得的经验，讨论一般情况下求解问题是否应使用按比例加速比而不用标准加速比。如果某一并行算法的按比例加速比曲线与按等效率函数增加问题规模确定的加速比曲线一致，那么对并行算法的等效率函数能得出什么结论？ 375

8.21 （时间受限制的扩展）假设在8.2.1节讨论的矩阵相乘算法的并行执行时间表达式（公式（8-14））中，$t_s = 10, t_w = 1$。对于p = 1、4、16、64、256、1024、4096，如果总运行时间不超过512个时间单位，那么能解决的最大问题规模是多少？一般情况下，如果进程数目不受限制，是否能在固定的时间内解决任意大的问题？试给出简要说明。

8.22 给出一个使用回代法求解$Ux = y$型三角方程组的流水线算法，其中使用二维划分把

$n \times n$单位上三角矩阵U划分到进程的$n \times n$格网上。给出算法的并行运行时间的表达式。修改算法使其可以在少于n^2个进程上工作，推导出修改后算法的并行运行时间的表达式。

8.23 考虑算法8-7给出的两个$n \times n$矩阵A和B相乘得到积矩阵C的并行算法。假设对一个矩阵元素进行一次内存读写需要的时间为t_{local}，两个数字相加与相乘需要的时间为t_c。确定这个算法在n^2个处理器CREW PRAW上的并行运行时间。这个并行算法是成本最优的吗？

8.24 假设在EREW PRAM上把对内存的并发读访问串行化，推导出算法8-7给出的算法在n^2个处理器EREW PRAM上的并行运行时间。这个算法在EREW PRAM上是成本最优的吗？

376

8.25 考虑一台有n^2个处理器的共享地址空间并行计算机。假设每个处理器都有一些本地内存，$A[i,j]$, $B[i,j]$被存储在$P_{i,j}$的本地内存中。此外，$P_{i,j}$在它的本地内存中计算$C[i,j]$。假设它在非本地内存中进行一次读或写需要的时间是$t_{local} + t_w$，在本地内存中进行一次读或写需要的时间是t_{local}。推导出算法8-7给出的算法在这台并行计算机上的并行运行时间的表达式。

算法8-7 CREW PRAM上两个n × n矩阵A和B相乘的算法，得出矩阵C = A × B

```
1.  procedure MAT_MULT_CREW_PRAM (A, B, C, n)
2.  begin
3.     Organize the n^2 processes into a logical mesh of n × n;
4.     for each process P_{i,j} do
5.     begin
6.        C[i, j] := 0;
7.        for k := 0 to n − 1 do
8.           C[i, j] := C[i, j] + A[i, k] × B[k, j];
9.     endfor;
10. end MAT_MULT_CREW_PRAM
```

算法8-8 EREW PRAM上两个n × n矩阵A和B相乘的算法，得出矩阵C = A × B

```
1.  procedure MAT_MULT_EREW_PRAM (A, B, C, n)
2.  begin
3.     Organize the n^2 processes into a logical mesh of n × n;
4.     for each process P_{i,j} do
5.     begin
6.        C[i, j] := 0;
7.        for k := 0 to n − 1 do
8.           C[i, j] := C[i, j]+
                        A[i, (i + j + k) mod n] × B[(i + j + k) mod n, j];
9.     endfor;
10. end MAT_MULT_EREW_PRAM
```

8.26 可对算法8-7进行修改，使其在EREW PRAM上的并行运行时间比习题8.24中的时间少。算法8-8给出修改后的程序。如果内存访问时间与习题8.24和8.25中给出的相同，那么算法8-8在EREW PRAM上的并行运行时间是多少？在共享地址空间并行计算机上的并行运行时间是多少？算法在这些体系结构上是成本最优的吗？

8.27 在一台少于n^2个（比如p个）处理器的共享地址空间并行计算机上考虑算法8-8的实现，假定内存访问时间与习题8.25中给出的时间相同。算法的并行运行时间是多少？

8.28 在一台共享地址空间的并行计算机上，如果内存访问时间与习题8.25中给出的时间相同，考虑8.2.1节给出的矩阵相乘算法的并行实现。在这个算法中，每个处理器首先把它接收到的所有数据存入它的本地内存中，然后执行计算。推导算法的并行运行时间的表达式。对这个算法的性能同习题8.27中的算法作比较。

8.29 使用习题8.23～8.28的结果，试评论将PRAM模型作为并行算法设计平台的可行性。同时对消息传递模型与共享地址空间计算机上关联作出评论。 377

第9章 排 序

排序是计算机上执行的一项最常见的操作之一。由于排序的数据比顺序随意的数据更容易操作，许多算法都需要排序的数据。因为排序与进程间确定数据传送路线的任务紧密相关，排序对于并行算法具有额外的重要性。人们已研究了许多能在不同并行计算机体系结构中使用的并行排序算法。本章讲述一些体系结构中的并行排序算法，如PRAM、格网、超立方体以及通用的共享地址空间和消息传递结构。

排序被定义为，把元素的无序集合按单调递增（或单调递减）的顺序排列的任务。具体说就是：令$S = \{a_1, a_2, ..., a_n\}$为$n$个元素以任意顺序排列的序列；排序把$S$转换成单调递增的序列$S' = \{a_1', a_2', ..., a_n'\}$，使得对于$1 \leqslant i < j \leqslant n$，有$a_i' \leqslant a_j'$，而$S'$是$S$的置换。

排序算法可以分为内排序（internal sort）算法和外排序（external sort）算法。在内排序算法中，排序元素的数目要足够少，使得它们能放入进程的主存储器。相反，外排序算法利用辅助存储器件（如磁带和硬盘）来排序，因为待排序的元素数目太多，不适合放进内存。本章只讲内排序算法。

排序算法又可以分为基于比较的（comparison-based）排序算法和基于非比较的（noncomparison-based）排序算法。基于比较的排序算法在排序时，对于无序元素序列里的元素对反复进行比较，如果次序颠倒，就交换其位置。这种基于比较的排序的基本操作称为比较交换（compare-exchange）。如果被排序的元素数目是n，那么任何基于比较排序算法的序列复杂度的下界为$\Theta(n \log n)$。基于非比较的排序利用元素的某些已知性质（如它们的二进制表示或它们的分布）。这种算法的复杂度下界为$\Theta(n)$。本章将主要介绍基于比较的排序算法，在9.6节中也要对某些基于非比较的排序作简要讨论。 379

9.1 并行计算机中的排序问题

将串行排序算法并行化时，需要将待排序元素分配到可用的进程中。这个过程中将会出现很多必须着重考虑的问题，这样才能更清晰地描述并行排序算法。

9.1.1 输入输出序列的存放位置

串行排序算法中，输入序列和排序后的序列存放在进程的内存里。但是在并行排序中，这些序列可以存放在两个地方：只存放在其中一个进程，或者分散存放在各个进程中。如果排序只是另一个算法的中间步骤，则后一种方法更为实用。在本章中，我们假设输入序列和排序后的序列分散存放在各个进程中。

下面考虑排序后的输出序列在各个进程中的确切分配。常用的分配方法是，枚举出所有进程，并利用此枚举对排序后的序列指定一个全局排序。换句话说，待排序序列将与进程枚举相对应，或序列将相对于进程枚举排序。例如，如果枚举中P_i排在P_j之前，则存放在P_i中的所有元素都要比存放在P_j中的任何元素小。进程的枚举有很多种方法，对于特定的并行算法和互连网络，有些枚举比其他枚举导致更有效的并行形式。

9.1.2 如何进行比较

串行排序算法很容易进行两个元素的比较交换，因为它们被存放在本地进程的内存中。在并行排序算法中，这个步骤不是那么容易。如果元素被存放在同一个进程，很容易进行比较。但如果元素被存放在不同进程，情形就复杂得多。

1. 每个进程存放一个元素

下面考虑每个进程只存放待排序序列的一个元素的情况。在算法执行的某一点，一对进程P_i和P_j可能需比较它们的元素a_i和a_j。在比较之后，P_i将存放$\{a_i, a_j\}$中的较小者，而P_j存放其中的较大者。这种比较可以通过两个进程互相发送它们的元素到对方进程来完成。每个进
380 程将它收到的元素与它自己的元素比较，然后保留合适的元素。在我们的例子中，P_i将保存$\{a_i, a_j\}$中的较小者，而P_j将保留其中的较大者。如同串行排序算法中的情形，这个操作称做比较交换。如图9-1所示，每一次比较交换操作需要一个比较步骤和一个通信步骤。

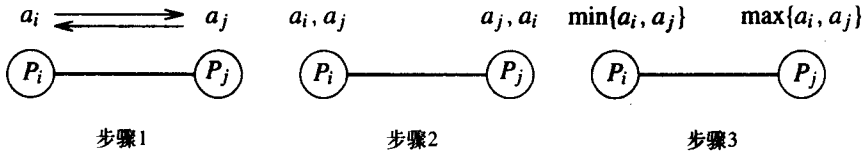


图9-1 并行的比较交换操作。进程P_i和P_j发送各自的元素到对方进程，进程P_i保留$\min\{a_i, a_j\}$，而P_j保留$\max\{a_i, a_j\}$

假设进程P_i和进程P_j相邻，而且通信通道是双向的，那么比较交换步骤的通信成本是$(t_s + t_w)$，其中t_s和t_w分别是消息启动时间和每字传输时间。在商用消息传递计算机中，t_s比t_w更大，所以通信时间以t_s为主。请注意，在当今的并行计算机中，进程间传递元素比单纯地对元素进行比较将花更长的时间。因此，如果并行排序形式使用与待排序元素同样多的进程，则它的性能很差，因为进程间的通信在整个并行运行时间中占主导地位。

2. 每个进程中存放多个元素

通用并行排序算法必须可以使用数目相对小的进程对大的序列进行排序。令p为进程P_0，P_1，…，P_{P-1}的数目，令n为待排序元素的数目。每个进程分配一个存放n/p个元素的块，所有的进程协作完成排序。令A_0，A_1，…，A_{P-1}是分别分配给进程P_0，P_1，…，P_{P-1}的块。如果A_i中的每个元素小于或等于A_j中的每个元素，那么就说$A_i \leqslant A_j$。当排序算法完成时，每个进程P_i将存放这样一组A_i'，如果$i < j$，则$A_i' \leqslant A_j'$，且$\bigcup_{i=0}^{p-1} A_i = \bigcup_{i=0}^{p-1} A_i'$。

与每个进程一个元素中的情况相同，两个进程P_i和P_j可能必须重新分配它们包含n/p个元素的块，使得其中一个得到较小的n/p个元素块，而另外一个得到较大的n/p个元素块。令A_i和A_j是分别存放在进程P_i和P_j中的元素块。如果每个进程中n/p个元素块已经被排序，重分配就可以按如下方式有效进行：每个进程将它的块发送给另一个进程。现在，每个进程将两个已排好序的块合并，只保留合并块的适当的一半。这种对两个已排序的块进行的比较和划分的操作称为比较分裂（compare-split）。比较分裂操作如图9-2所示。

381 如果假设进程P_i和P_j相邻，而且通信通道是双向的，那么比较分裂操作的通信成本是$(t_s + t_w n/p)$。随着块的增大，t_s的重要性下降，如果块足够大，则t_s可以忽略。这样，合并两个排好序的有n/p元素的块所需要的时间是$\Theta(n/p)$。

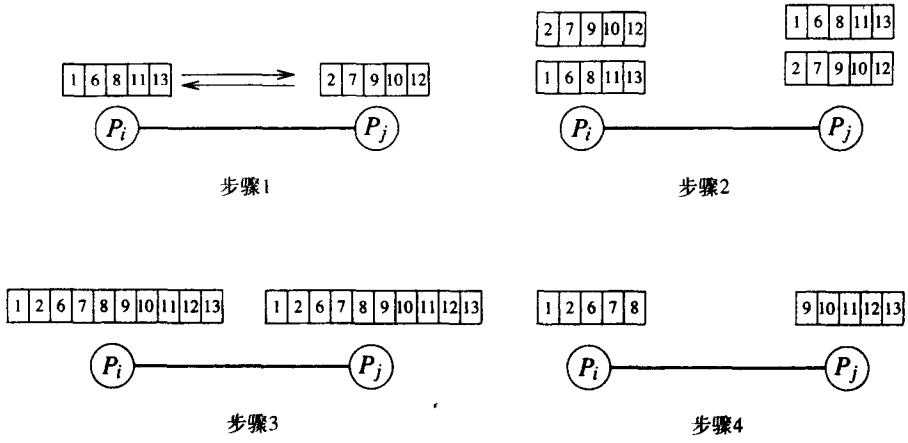
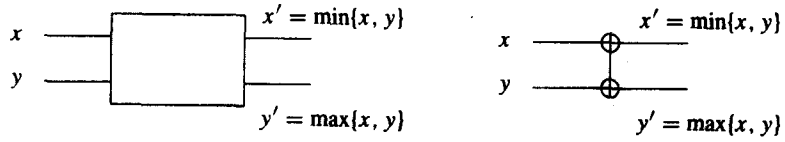


图9-2 比较分裂操作。每个进程发送n/p个元素的块到其他进程。每个进程将接收到的块与它们自己的块合并，只保留适合的那一半块。在这个例子中，进程P_i保留较小的那些元素，而进程P_j保留较大的那些元素

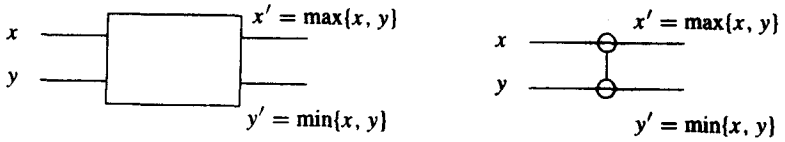
9.2 排序网络

为了探索快速排序方法，许多网络都设计成对n个元素排序的时间要远小于$\Theta(n \log n)$。这些排序网络基于比较网络模型，在模型中许多比较操作都是同时执行的。

这些网络的主要组件是*比较器*（comparator）。比较器是具有两个输入x和y及两个输出x'和y'的设备。对于*递增比较器*（increasing comparator），$x' = \min\{x, y\}$, $y' = \max\{x, y\}$；对于*递减比较器*（decreasing comparator），$x' = \max\{x, y\}$, $y' = \min\{x, y\}$。图9-3给出两种类型的比较器的图示。当两个元素进入比较器的输入线时，就对它们进行比较，并且如果需要的话，在它们到输出线前进行交换。我们用⊕表示递增比较器，用⊖表示递减比较器。排序网络通常由一连串的列组成，并且每个列都包含许多并行连接的比较器。比较器的每个列都执行置换，并且从最后列得到的输出按递增或递减顺序排序。图9-4显示一个典型的排序网络。网络的深度（depth）是指其包含的列的数目。由于比较器的速度是一个与技术有关的常量，网络的速度与深度成正比。 382



a）递增比较器



b）递减比较器

图9-3 比较器的图示

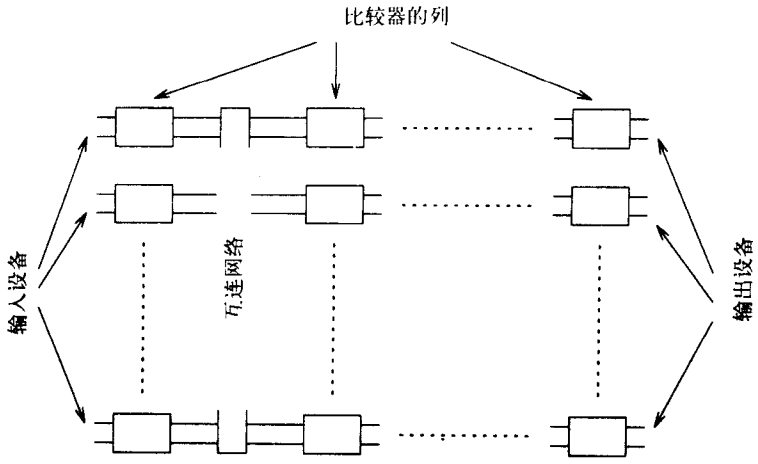


图9-4 一个典型的排序网络。每个排序网络由一连串的列组成，每个列包含许多并行连接的比较器

用软件仿真比较器，并且对每一列实行串行比较，可以将任何排序网络转换成串行排序算法。比较器通过比较交换操作进行仿真，其中x与y进行比较，如果需要的话，将两者进行交换。

下一节介绍在时间$\Theta(\log^2 n)$内对n个元素进行排序的排序网络。为了简单起见，假定n是2的幂。 383

9.2.1 双调排序

双调排序网络排序n个元素的时间为$\Theta(\log^2 n)$。其关键操作是重新调整双调序列为有序序列。*双调序列*（bitonic sequence）是指序列的元素$\{a_0, a_1, \ldots, a_{n-1}\}$有以下属性之一：1）存在下标$i$，$0 \leqslant i \leqslant n-1$，使得$\{a_0, a_1, \ldots, a_i\}$是单调递增的而$\{a_{i+1}, \ldots, a_{n-1}\}$是单调递减的；2）存在循环移位下标使得(1)能够满足。例如，{1, 2, 4, 7, 6, 0}是双调序列，因为它先递增后递减。同样，{8, 9, 2, 1, 0, 4}也是个双调序列，因为它是序列{0, 4, 8, 9, 2, 1}的循环移位。

下面介绍重新排序双调序列获得单调递增序列的方法。令$s = \{a_0, a_1, \ldots, a_{n-1}\}$是双调序列，其中$a_0 \leqslant a_1 \leqslant \ldots \leqslant a_{n/2-1}$并且$a_{n/2} \geqslant a_{n/2+1} \geqslant \ldots \geqslant a_{n-1}$。考虑下面$s$的子序列：

$$\begin{aligned} s_1 &= \langle \min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\}, \ldots, \min\{a_{n/2-1}, a_{n-1}\} \rangle \\ s_2 &= \langle \max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\}, \ldots, \max\{a_{n/2-1}, a_{n-1}\} \rangle \end{aligned} \tag{9-1}$$

在序列s_1中，存在一个元素$b_i = \min\{a_i, a_{n/2+i}\}$，使得$b_i$以前的所有元素都来自原始序列的递增部分，而$b_i$以后的所有元素都来自递减部分。同样，在序列$s_2$中，存在元素$b'_i = \max\{a_i, a_{n/2+i}\}$，$b'_i$之前的元素都是来自原始序列的递减部分，而$b'_i$之后的所有元素都来自递增部分。这样，序列$s_1$和$s_2$就都是双调序列。而且，第一个序列中的每个元素都小于第二个序列中的元素。原因是，b_i大于等于s_1中的所有元素，b'_i小于等于s_2中所有的元素，并且b'_i大于等于b_i。这样，再排列大小为n的双调序列，原始问题就化简为再排列两个较小的双调序列，并将结果连接起来。我们把大小为n的双调序列划分成公式(9-1)中定义的两个双调序列的操作称为*双调分裂*（bitonic split）。虽然在得到s_1和s_2时我们假定初始序列的递增和递减序列有相同的长度，

但双调划分操作也可以应用于任何的双调序列（习题9.3）。

使用公式(9-1)可以递归地得到每个双调子序列的更短一些的双调序列，直到得到的子序列的长度为1为止。到那时，输出序列按单调递增顺序排列。因为每次双调划分操作后，问题的长度都会减半，用来重新调整双调序列成有序序列所需划分的次数目是log n。应用双调划分对双调序列进行排序的过程称为双调合并（bitonic merge）。递归双调合并过程如图9-5所示。

原始序列	3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
第1次分裂	3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
第2次分裂	3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
第3次分裂	3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
第4次分裂	0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95

图9-5 通过一系列log 16双调分裂合并一个有16个元素的双调序列

我们现在有了合并双调序列成为有序序列的方法。这个方法容易在比较器网络上实现。这种比较器网络称为双调合并网络（bitonic merging network），如图9-6所示。网络包含log n个列，每个列包含$n/2$个比较器，并执行一步双调合并。该网络将双调序列作为输入，输出有序序列。我们用⊕BM[n]表示有n个输入的双调合并网络。如果在图9-6中将⊕比较器替换为⊖比较器，则输入将按单调递减的顺序排序；这样的网络用⊖BM[n]表示。 384

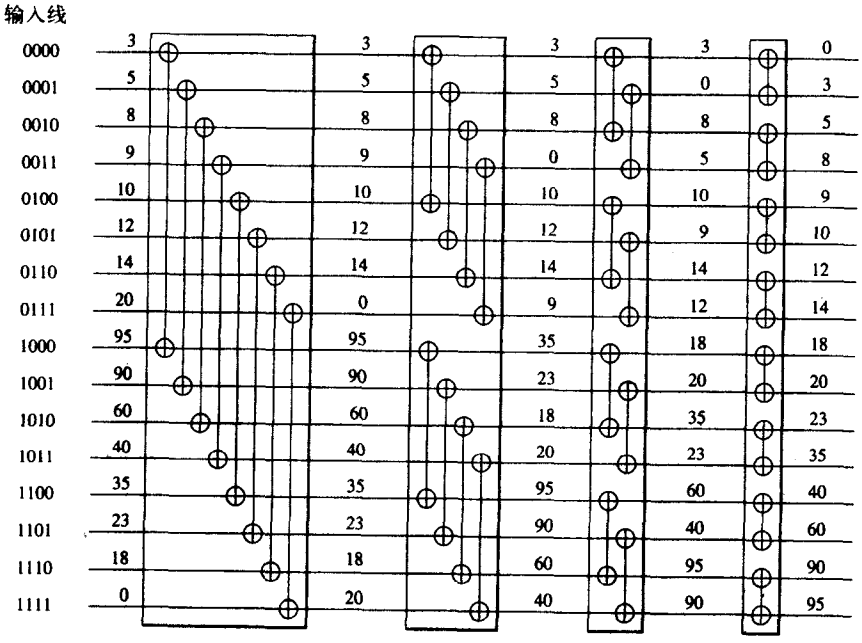


图9-6 n = 16的双调排序网络

注：输入线被编号为0, 1, ..., n-1，并且显示这些数字的二进制表示。单独画出比较器的每一列；整个图描绘一个⊕BM[16]双调合并网络。网络接收双调序列，输出排序后的序列。

有了双调合并网络后，再来考虑对n个无序元素的排序任务。此项任务可通过重复合并长度渐增的双调序列完成，如图9-7所示。

现在来看这种方法的工作原理。两个元素x和y的序列构成一个双调序列，因为如果$x \leqslant y$，x和y都位于双调网络的递增部分，而递减部分没有元素，或者$x \geqslant y$，双调序列的递减部分有两个元素x和y，而递增部分没有元素。因此，任何无序序列元素都是两个双调序列的连接。 385

图9-7所示的网络的每一级都按照递增和递减顺序合并相邻的双调序列。按照双调序列的定义，通过连接递增和递减序列而得到的序列是双调的。因此，图9-7所示网络的每一级的输出是双调序列的连接，其长度是输入的两倍。通过连接越来越多的双调序列，最后得到大小为n的双调序列。合并这个序列对输入进行排序。我们将具体体现在这个方法中的算法称为*双调排序*(bitonic sort)，而将网络称为*双调排序网络*(bitonic sorting network)。图9-8明白显示图9-7所示网络的前三个阶段。图9-6明白显示图9-7的最后阶段。

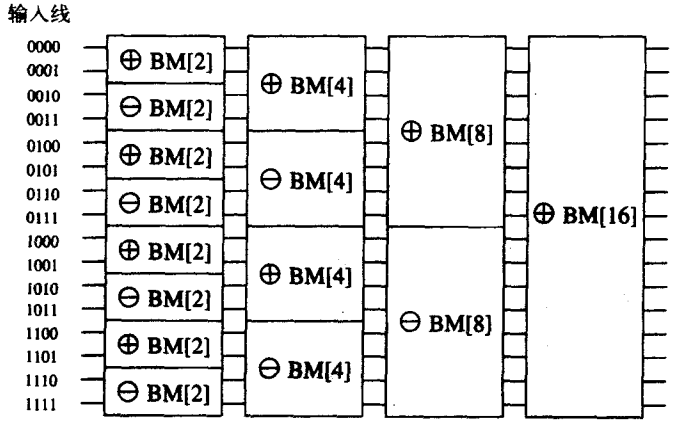


图9-7 将输入序列转化成双调序列的网络图示

注：在这个例子中，⊕BM[k]和⊖BM[k]表示输入大小为k的双调合并网络，分别用≈比较器和⊖比较器。最后的合并网络(⊕BM[16])对输入排序，在本例中n = 16。

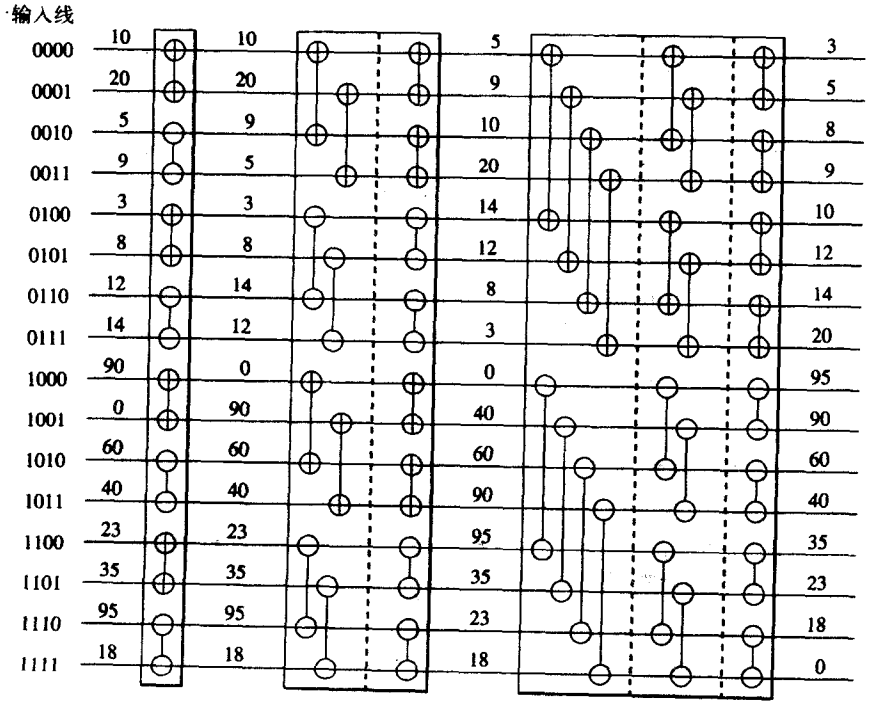


图9-8 将包含16个无序元素的输入序列转换成双调序列的比较网络

注：与图9-6相比，在每个双调合并网络上的比较器的列都画在一个单独的方框中，由虚线分隔开。

n个元素双调排序网络的最后一级包含具有n个输入的双调合并网络，其深度是log n。其他级进行$n/2$个元素的完全排序。因此，图9-7所示网络的深度$d(n)$由下列递推关系给出：

$$d(n) = d(n/2) + \log n \tag{9-2}$$

解方程(9-2)，得到 $d(n) = \sum_{i=1}^{\log n} i = (\log^2 n + \log n)/2 = \Theta(\log^2 n)$。这个网络可在串行计算机上实现，产生花费时间为$\Theta(n \log^2 n)$的排序算法。双调排序网络也可以修改成用于并行计算机上的排序算法。下一节介绍如何在超立方体和格网连接的并行计算机上实现这一点。

9.2.2 将双调排序映射到超立方体和格网

本节将讨论如何将双调排序算法映射到通用并行计算机上。双调算法的关键特点是它的密集通信，好的算法映射必须考虑到底层互连网络的拓扑结构。因此，我们要讨论如何将双调排序算法映射到超立方体连接的和格网连接的并行计算机的互连网络中。

对n个元素进行排序的双调排序网络包括log n级，其中第i级由$n/2$个比较器的i列组成。如图9-6和图9-8所示，比较器的每一列在n条输入线上执行比较-交换操作。在并行计算机上，比较-交换功能是由一对进程执行的。

1. 每进程一个元素

在这种映射中，n个进程中的每一个包含输入序列中的一个元素。从图解的角度来说，双调排序网络中的每条线分别代表不同的进程。在算法的每一个步中，列比较器进行的比较-交换操作由$n/2$对进程执行。在比较-交换操作中为了尽量减少元素移动的距离，如何将进程映射到线上是一个重要问题。如果映射很差的话，元素在可以进行比较前要移动很长的距离，这样将会降低性能。在理想情况下，执行比较-交换的线应该映射到邻近的进程，那么双调排序形式的并行性能就会比其他需要n个进程的并行形式的性能更好。

386 ~ 387

为了得到良好的映射，在双调排序的每个阶段都要深入研究输入线组对的方法。考虑n = 16时图9-6和图9-8所示的全双调排序网络。在每个(1 + log16)(log16)/2 = 10的比较器列中，有特定的线比较-交换它们的元素。下面考虑线标号的二进制表示。在每一步，仅当线标号有一位不同时，才在两条线之间执行比较-交换操作。在四个阶段的每一阶段中，最低有效位标号不同的线在每个阶段的最后一步执行比较-交换。在最后三个阶段，次最低有效位标号不同的线在每个阶段的倒数第二步执行比较-交换。通常，第i个最低有效位标号不同的线执行(log n $-i+1$)次比较-交换。这个观察结果，有助于我们通过把更频繁执行比较-交换操作的线映射到相互接近的进程，有效地将线映射到进程上。

超立方体 将线映射到超立方体连接的并行计算机的进程上比较简单。比较-交换操作发生在标号只相差一位的线上。在超立方体上，标号只相差一位的进程是相邻的（见2.4.3节）。这样，将标号为l的输入线映射到标号为l的进程上，其中l = 0, 1, …, $n-1$，这就是将输入线映射到超立方体进程上的最优映射。

考虑在d维超立方体（即$p = 2^d$）中为了比较-交换如何对进程组对。在双调排序的最后一级，输入被转变成双调序列。在这一级的第一步，对进程标号二进制表示的第d位（即最高有效位）有差别的进程比较-交换它们的元素。这样，比较-交换操作发生在沿第d维的进程之间。同样地，在算法的第二步，比较-交换操作发生在沿第$(d-1)$维的进程中。通常，在最后一级的第i步，进程沿第$(d-(i-1))$维通信。图9-9显示双调排序算法的最后一级的通信。

大小为2^k的序列的双调合并可在k维子立方体上执行，每个这样的序列都被分配到不同的子立方体上（见习题9.5）。而且，在该双调合并的第i步，沿第$(k-(i-1))$维进行比较的进程是

388 相邻的。图9-10是对图9-7的修改，说明超立方体上的双调排序算法的通信特征。

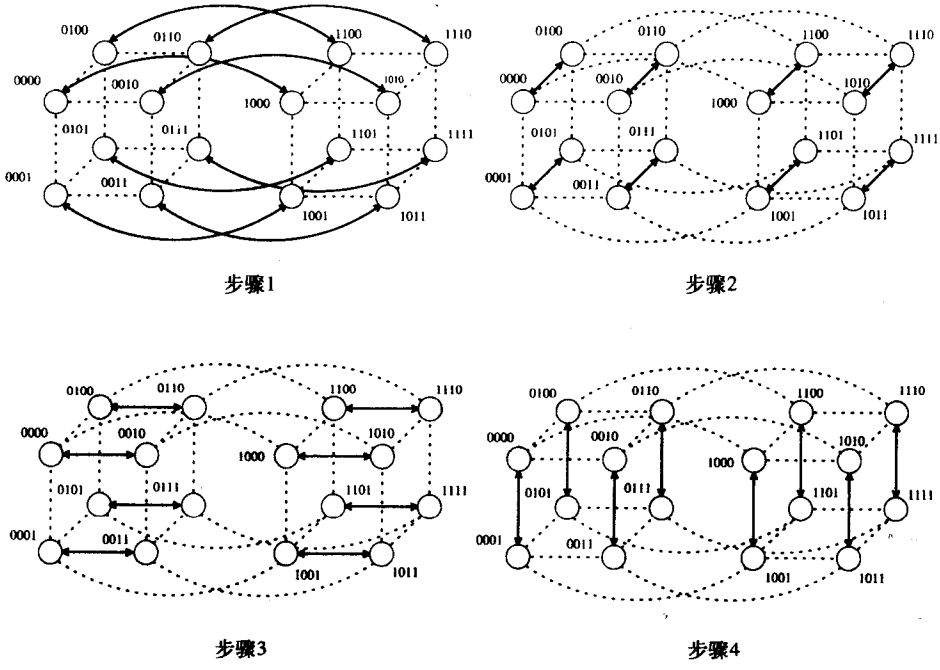


图9-9 双调排序最后阶段的通信。每条线都被映射到一个超立方体进程；每个连接都代表进程间的一次比较-交换

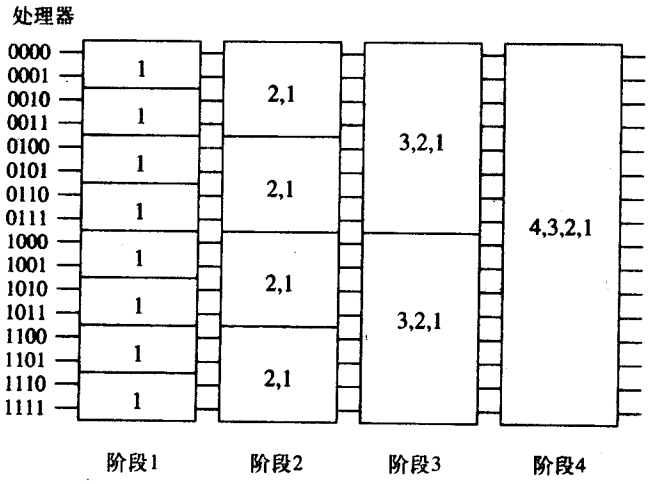


图9-10 在超立方体上的双调排序的通信特征。在算法的每个阶段，进程通信沿着显示的维进行

389

超立方体上的双调排序算法如算法9-1所示。算法依赖于两个函数*comp_exchange_max*(*i*)和*comp_exchange_min*(*i*)。这两个函数沿第*i*维将本地元素与最近进程上的元素相比较，并且保留两个元素中最小的或最大的元素。习题9.6分析算法9-1的正确性。

算法9-1 在n = 2^d个进程的超立方体上的双调排序并行形式。在这个算法中，label是进程的标号，d是超立方体的维

```
1.  procedure BITONIC_SORT(label, d)
2.  begin
3.      for i := 0 to d − 1 do
4.          for j := i downto 0 do
5.              if (i + 1)st bit of label ≠ jth bit of label then
6.                  comp_exchange_max(j);
7.              else
8.                  comp_exchange_min(j);
9.  end BITONIC_SORT
```

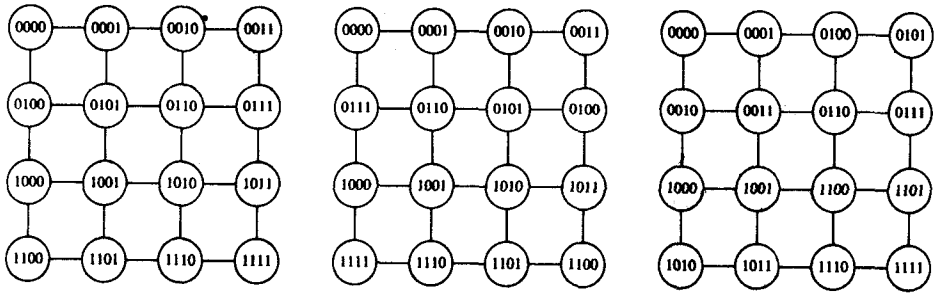
在算法的每一步、每个进程执行一次比较-交换操作。算法总共执行$(1 + \log n)(\log n)/2$次这样的步骤；因此，并行运行时间为

$$T_P = \Theta(\log^2 n) \tag{9-3}$$

相对于双调排序的串行实现（即，进程时间积为$\Theta(\log^2 n)$），这个双调排序的并行形式是成本最优的，但与串行时间复杂度为$\Theta(n \log n)$的基于比较的排序算法相比，它不是成本最优的。

格网 考虑如何有效地将双调排序网络的输入线映射到n个进程的格网上。不幸的是，由于格网的连通性低于超立方体的连通性，所以不可能把线映射到进程上，使得每个比较-交换操作只出现在相邻的进程上，相反，映射线时最频繁的比较-交换操作出现在相邻进程之间。

有几种方法将输入线映射到格网进程上。图9-11中列出其中一些方法。图中的每个进程用映射到它的线作为标号。在三种映射中我们将重点研究图9-11c所示的以行为主的混洗映射，另外两个留作练习（习题9.7）。



a）以行为主的映射　　b）以行为主的蛇行映射　　c）以行为主的混洗映射

图9-11 映射双调排序网络的输入线到进程格网的不同方法

以行为主的混洗映射的优点在于，执行比较-交换操作的进程驻留在格网的方形子段上，而方形子段的大小与比较-交换的频率具有相反的关系。例如，在双调排序的每个阶段，执行比较交换的进程（即对应于最低有效位不同的线的进程）是相邻的。通常，第i个最低有效位不同的线被映射到相距$2^{\lfloor (i-1)/2 \rfloor}$个通信链路的格网进程。在以行为主混洗映射的双调排序中，其最后阶段的比较-交换步骤如图9-12所示。注意在每个较早阶段可能只会有这些步骤 390

的一部分。

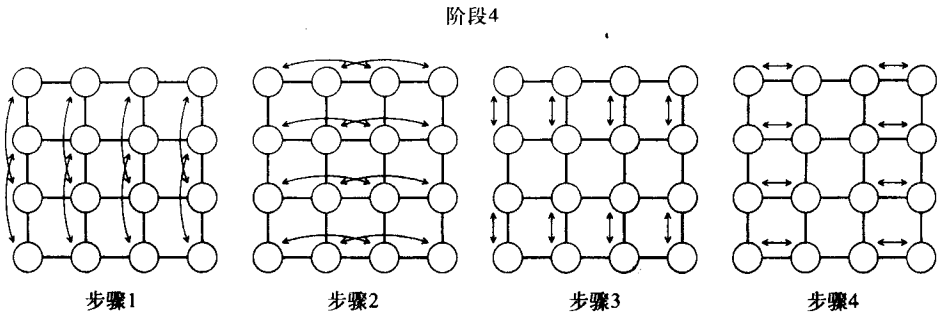


图9-12 格网上$n=16$的双调排序算法的最后一个阶段，使用以行为主的混洗映射。在每一步，进程对比较-交换它们的元素。箭头表明执行比较-交换操作的进程对

在算法的$(1+\log n)(\log n)/2$个步骤中，相距一定距离的进程比较-交换它们的元素。进程间的距离决定并行形式的通信开销。每个进程执行的通信总量是 $\sum_{i=1}^{\log n}\sum_{j=1}^{i}2^{\lfloor (j-1)/2\rfloor}=7\sqrt{n}$，也就是 $\Theta(\sqrt{n})$（见习题9.7）。在算法的每步中，每个进程最多执行一次比较；这样，每个进程执行的总计算量为$\Theta(\log^2 n)$。它产生的并行运行时间为

391

$$T_P=\overbrace{\Theta(\log^2 n)}^{\text{比较}}+\overbrace{\Theta(\sqrt{n})}^{\text{通信}}$$

这不是成本最优的方法，因为进程-时间乘积为$\Theta(n^{1.5})$，而排序的串行复杂度是$\Theta(n\log n)$。虽然与双调排序的串行复杂度相比，超立方体的并行形式是最优的，但格网的方法不是最优的。我们可以做得更好吗？回答是否定的。当对n个元素进行排序时，每个格网进程一个元素，存放在左上角进程的特定输入元素将在右下角进程结束。为了做到这一点，元素在到达它目的地前必须沿着 $2\sqrt{n}-1$ 条通信链路移动。这样，在格网上排序的运行时间以 $\Omega(\sqrt{n})$ 为界。我们的并行形式可以达到这个下界；这样，对格网结构它是渐近最优的。

2. 每个进程一个元素块

在迄今介绍的双调排序算法的并行方式中，都假定进程和待排序元素一样多。现在考虑待排序元素数目多于进程数目的情况。

设p是进程数目，n是待排序元素数目，其中$p<n$。每个进程分配到包含n/p个元素的块，并与其他进程合作对这些元素进行排序。将每个进程看作由n/p个小进程构成，对这种新设置得出一个并行形式。换句话说，设想使用单进程模拟n/p个进程。因为每个进程都在做n/p个进程的工作，这种并行形式的运行时间会大n/p倍。这种虚拟进程方法（见5.3节）导致双调排序并行的较差实现。为了说明这一点，考虑超立方体上有n个进程时的情况。其运行时间将是$\Theta((n\log^2 n)/p)$，这不是成本最优的，因为进程-时间乘积为$\Theta(n\log^2 n)$。

处理元素块的另一个方法是使用已在9.1节提出的比较-分裂操作。把(n/p)元素块想象成用比较-分裂操作进行排序的元素。对p个块的排序问题，与使用比较-分裂操作而不是比较-交换操作（见习题9.8）对p个块执行双调排序完全相同。由于块的总数目是p，双调排序算法总共有$(1+\log p)(\log p)/2$个步骤。因为比较-分裂操作保存每个块中元素的初始排序，所

以在这些步骤的最后，这n个元素形成有序序列。该方法与使用虚拟进程的并行方式的主要差别是，分配到每个进程的n/p个元素最初使用快速串行排序算法在本地排序。这个初始本地排序使得新方式更有效，并且是成本最优的。

超立方体 n个进程的超立方体的基于块的算法与每个进程一个元素的算法类似，只不过前者有p个大小为n/p的块而不是p个元素。此外，比较-交换操作被比较-分裂操作代替，每次操作需要$\Theta(n/p)$计算时间和$\Theta(n/p)$通信时间。最初，进程需要时间$\Theta((n/p)\log(n/p))$对它们的n/p个元素（用合并排序法）进行排序，然后执行$\Theta(\log^2 p)$个比较-分裂步骤。这种形式的并行运行时间为

392

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{本地排序}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{比较}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{通信}}$$

由于最好的排序算法的串行时间复杂度是$\Theta(n\log n)$，加速比和效率如下：

$$\begin{aligned} S &= \frac{\Theta(n\log n)}{\Theta((n/p)\log(n/p)) + \Theta((n/p)\log^2 p)} \\ E &= \frac{1}{1 - \Theta((\log p)/(\log n)) + \Theta((\log^2 p)/(\log n))} \end{aligned} \tag{9-4}$$

从公式(9-4)中可以看出，对于成本最优化形式，$(\log^2 p)/(\log n) = O(1)$。这样，这个算法可以有效地使用最多 $p = \Theta(2^{\sqrt{\log n}})$个进程。从公式(9-4)还可以看出，由于通信和额外的工作导致的等效率函数是$\Theta(p^{\log p}\log^2 p)$，对于足够大的$p$，它比任何多项式等效率函数还要差。因此，双调排序的并行方式可扩展性很差。

格网 基于块的格网方式也与每个进程一个元素的方式类似。这种形式的并行运行时间如下：

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{本地排序}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{比较}} + \overbrace{\Theta\left(\frac{n}{\sqrt{p}}\right)}^{\text{通信}}$$

注意，这种基于格网的并行双调排序的通信开销是 $(O(n/\sqrt{p}))$，将它与基于超立方体方式的通信开销$(O((n\log^2 p)/p))$相比，可以看出高出$O(\sqrt{p}/\log^2 p)$倍。这个倍数要小于这些结构上的对分带宽差 $O(\sqrt{p})$。这说明，正确地将双调排序映射到基础格网中，比简单地把超立方体算法映射到格网上能获得更好的性能。

加速比和效率如下：

$$\begin{aligned} S &= \frac{\Theta(n\log n)}{\Theta((n/p)\log(n/p)) + \Theta((n/p)\log^2 p) + \Theta(n/\sqrt{p})} \\ E &= \frac{1}{1 - \Theta((\log p)/(\log n)) + \Theta((\log^2 p)/(\log n)) + \Theta(\sqrt{p}/\log n)} \end{aligned} \tag{9-5}$$

393

从公式(9-5)看出，对于成本最优的形式，$\sqrt{p}/\log n = O(1)$。这样，这个形式可以有效地使用最多$p = \Theta(\log^2 n)$个进程。从公式(9-5)还可以看出，等效率函数为 $\Theta(2^{\sqrt{p}}\sqrt{p})$。这种形式的等效率函数是指数的，因此比超立方体的甚至更差。

从对超立方体和格网的分析可以看出，双调排序的并行形式效率既不高，扩展性也不好。这主要是由于串行算法是次最优的缘故。仅当待排序的元素数目非常大时，对于很大的进程数目才可能得到好的加速。在这种情况下，内部排序的效率会超过双调排序的低效率。表9-1概括在超立方体、格网以及环形连接的并行计算机上的双调排序的性能。

表9-1 在p个进程上对n个元素进行并行形式双调排序的性能

结 构	$E=\Theta(1)$时的最大进程数	对应的并行运行时间	等效率函数
超立方体	$\Theta(2^{\sqrt{\log n}})$	$\Theta(n/(2^{\sqrt{\log n}})\log n)$	$\Theta(p^{\log p}\log^2 p)$
格网	$\Theta(\log^2 n)$	$\Theta(n/\log n)$	$\Theta(2^{\sqrt{p}}\sqrt{p})$
环	$\Theta(\log n)$	$\Theta(n)$	$\Theta(2^p p)$

9.3 冒泡排序及其变体

前一节介绍了可以在时间$\Theta(\log^2 n)$对n个元素进行排序的排序网络。现在把注意力转到更传统的排序算法。既然时间复杂度为$\Theta(n \log n)$的串行算法是存在的，那么应能利用$\Theta(n)$个进程在时间$\Theta(\log n)$对n个元素进行排序。我们将看到，这实现起来很困难。但是，将许多复杂度为$\Theta(n^2)$的串行算法并行化是容易实现的。我们现在介绍基于*冒泡排序*（bubble sort）的算法。

串行冒泡排序算法在待排序序列中比较并交换相邻的元素。给定序列$\{a_1, a_2, \ldots, a_n\}$，算法首先将以下列顺序执行$n-1$次比较-交换操作：$(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$。这一步将最大的元素移到序列的末端。变换后序列的最后一个元素被忽略，并且把比较-交换的步骤应用在得到的序列$\{a_1', a_2', \ldots, a_{n-1}'\}$上。经过$n-1$次迭代后，序列就排好序了。在迭代中，如果没有交换发生就终止，以此来提高冒泡排序的性能。冒泡排序算法如算法9-2所示。

算法9-2 串行冒泡排序算法

```
1.  procedure BUBBLE_SORT(n)
2.  begin
3.      for i := n - 1 downto 1 do
4.          for j := 1 to i do
5.              compare-exchange(a_j, a_{j+1});
6.  end BUBBLE_SORT
```

冒泡排序内层循环中的一次迭代用时$\Theta(n)$，算法总共需要$\Theta(n)$次迭代，因此算法的时间复杂度为$\Theta(n^2)$。冒泡排序很难并行化。为看出这一点，考虑在算法的每个阶段如何执行比
394 较-交换操作（算法第4、5行）。冒泡排序按顺序比较所有相邻的对；因此，它本质上是串行的。下面两小节将介绍非常适合并行化的冒泡排序的两种变体。

9.3.1 奇偶转换

奇偶转换（odd-even transposition）算法在n个阶段内对n个元素（n是偶数）排序，每个阶段需要$n/2$次比较-变换操作。算法在两个阶段之间交替执行，称为奇数阶段和偶数阶段。令$(a_1, a_2, \ldots, a_n)$为待排序序列。在奇数阶段，下标为奇数的元素与它们右边相邻的元素比较，如果它们未按顺序排列，就交换它们；这样，每个对$(a_1, a_2), (a_3, a_4), \ldots, (a_{n-1}, a_n)$都进行比较-交换（假设$n$是偶数）。同样，在偶数阶段，下标为偶数的元素与它们右边相邻的元素比较，

如果它们未按顺序排列，就交换它们；这样，每个对$(a_2, a_3), (a_4, a_5), \ldots, (a_{n-2}, a_{n-1})$进行比较-交换。经过$n$个阶段的奇偶转换，序列就完成排序。算法的每个阶段（不管是奇数阶段还是偶数阶段）需要$\Theta(n)$次比较，并且共有n个阶段；这样，串行时间复杂度就是$\Theta(n^2)$。奇偶转换排序如算法9-3所示，并在图9-13中说明。

算法9-3 串行奇偶转换排序算法

```
1.  procedure ODD-EVEN(n)
2.  begin
3.      for i := 1 to n do
4.      begin
5.          if i is odd then
6.              for j := 0 to n/2 − 1 do
7.                  compare-exchange(a_{2j+1}, a_{2j+2});
8.          if i is even then
9.              for j := 1 to n/2 − 1 do
10.                 compare-exchange(a_{2j}, a_{2j+1});
11.     end for
12. end ODD-EVEN
```

395

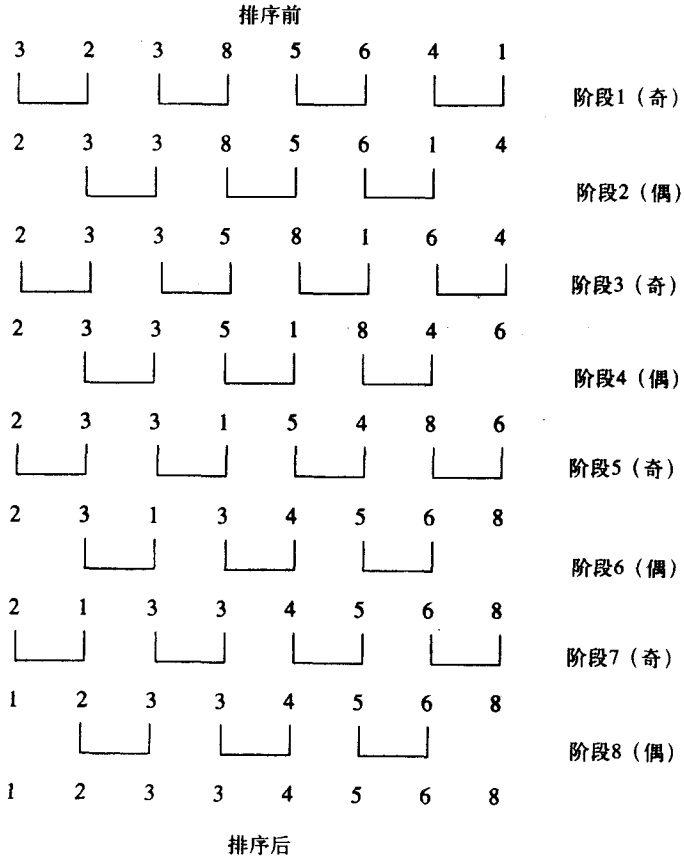


图9-13 排序n = 8个元素，使用奇偶转换排序算法。在每个阶段，比较n = 8个元素

并行形式

将奇偶转换排序并行化是很简单的。在算法的每个阶段，每对元素上的比较-交换操作同时执行。考虑每个进程一个元素的情况。令n是进程的数目（也是待排序元素的数目）。假定进程在一维数组中排序。对于$i = 1, 2, \cdots, n$，元素a_i初始驻留在进程P_i。在奇数排序阶段，每个奇数下标的进程将其元素与其右的相邻元素进行比较-交换。同样，在偶数阶段，每个偶数
396 下标的进程将其元素与其右边的相邻元素进行比较-交换。算法9-4给出这种并行形式。

算法9-4 n个进程环上奇偶转换排序的并行形式

```
1.  procedure ODD-EVEN_PAR(n)
2.  begin
3.      id := process's label
4.      for i := 1 to n do
5.      begin
6.          if i is odd then
7.              if id is odd then
8.                  compare-exchange_min(id + 1);
9.              else
10.                 compare-exchange_max(id − 1);
11.         if i is even then
12.             if id is even then
13.                 compare-exchange_min(id + 1);
14.             else
15.                 compare-exchange_max(id − 1);
16.     end for
17. end ODD-EVEN_PAR
```

在算法的每个阶段，奇数进程或偶数进程执行一次与其右邻的比较-交换。从9.1节可知，这个阶段需要$\Theta(1)$的时间。由于总共要执行n个这样的阶段，这种形式的并行运行时间为$\Theta(n)$。由于n个元素的最优串行排序算法的复杂度是$\Theta(n \log n)$，这种奇偶变换排序形式不是成本最优的，因为算法的进程-时间乘积是$\Theta(n^2)$。

为了获得成本最优的并行形式，我们使用较少的进程。令p是进程的数目，其中$p < n$。最初，对每个进程分配n/p个元素的块，用时间$\Theta((n/p)\log(n/p))$进行内部排序（使用合并排序或者快速排序）。然后，进程执行p个阶段（$p/2$个奇数阶段和$p/2$个偶数阶段），执行比较-分裂操作。在这些阶段结束时，序列排序完毕(见习题9.10)。在每个阶段，合并两个块执行$\Theta(n/p)$次比较，通信所需时间为$\Theta(n/p)$。这样，该形式的并行运行时间是

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{本地排序}} + \overbrace{\Theta(n)}^{\text{比较}} + \overbrace{\Theta(n)}^{\text{通信}}$$

由于串行排序的时间复杂度是$\Theta(n \log n)$，这种形式的加速比和效率如下：

$$S = \frac{\Theta(n \log n)}{\Theta((n/p)\log(n/p)) + \Theta(n)}$$

397

$$E = \frac{1}{1 - \Theta((\log p)/(\log n)) + \Theta(p/\log n)} \qquad (9\text{-}6)$$

从公式(9-6)看出，当$p = O(\log n)$时，奇偶转换排序是成本最优的。这种并行形式的等效率函数是$\Theta(p2^p)$，是指数函数。因此这种形式很难扩展，且只适合于进程数目较少的情况。

9.3.2 希尔排序

奇偶转换排序的主要局限在于，移动元素每次只能移动一个位置。如果序列只有少数元素是无序的，并且，如果这些元素与它们合适位置的距离是$\Theta(n)$，那么串行算法需要时间$\Theta(n^2)$对序列排序。为了使排序比奇偶转换排序有重大改进，就需要一种将元素作长距离移动的算法。希尔排序就是这样一种串行排序算法。

令n为待排序的元素数，p为进程数。为简单起见，假定进程的数目是2的幂，即$p = 2^d$，不过此算法也很容易扩展到有任意个进程的情况。对每个进程分配n/p个元素的块。考虑按逻辑一维数组排列进程，进程在数组中的顺序定义待排序序列的全局顺序。算法包含两个阶段。在第一阶段，数组中相互远离的进程间比较-分裂它们的元素。在少数几步内，元素将长距离移动，达到靠近它们最后目的地的位置。在第二阶段，算法转换成一种奇偶转换排序，与前一节介绍的类似。唯一的差别是，奇偶阶段只在进程中的块改变的时候才执行。由于算法第一阶段把元素移动到最后目的地附近，第二阶段执行的奇偶阶段的数目就可能远小于p了。

最初，每个进程在时间$\Theta(n/p \log(n/p))$内对其n/p个元素的块进行内部排序。然后，每个进程与数组中按相反顺序对应的进程配对。就是说，进程P_i（其中$i < p/2$）与进程P_{p-i-1}组对。每对进程执行一次比较-分裂操作。然后，进程被分成两组，一组包含前$p/2$个进程，另一个组包含后$p/2$个进程。此时，每个组都被看作分开的$p/2$个进程，并且应用上面的进程配对方案决定对哪个进程执行比较-分裂操作。这个过程继续d步，直到每个组只包含一个进程为止。$d = 3$时第一阶段的比较分裂操作如图9-14所示。要注意的是，这并不是串行希尔排序的直接并行形式，但它基于类似的原理。

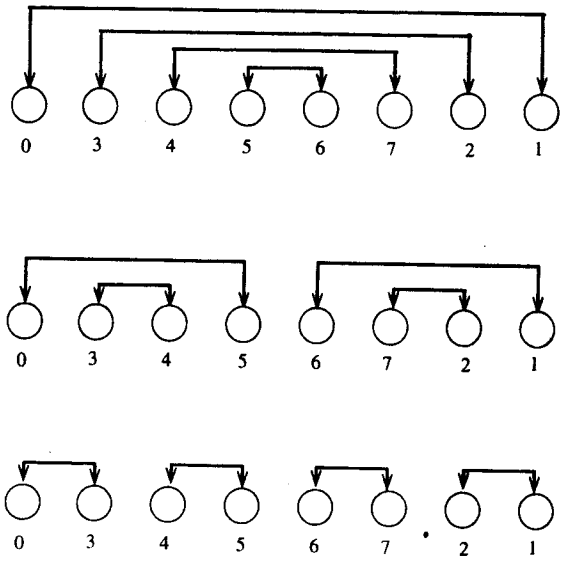


图9-14　在一个8进程数组上的并行希尔排序的第一阶段的例子

在算法的第一阶段，每个进程都执行$d = \log p$次比较-分裂操作。在每一次比较-分裂操

作中，总共$p/2$对进程需要交换它们内部存放的n/p个元素。这些比较-交换操作所需的通信时
398 间取决于网络的对分带宽。在对分带宽为$\Theta(p)$的情况下，每个操作所需的总时间是$\Theta(n/p)$。因此，这个阶段的时间复杂度是$\Theta((n \log p)/p)$。第二阶段要执行l次奇偶转换排序，每次需要时间$\Theta(n/p)$。因此，算法的并行运行时间是

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{本地排序}} + \overbrace{\Theta\left(\frac{n}{p}\log p\right)}^{\text{第一阶段}} + \overbrace{\Theta\left(l\frac{n}{p}\right)}^{\text{第二阶段}} \tag{9-7}$$

希尔排序的性能取决于l的值。如果l比较小，那么算法显著好于奇偶转换排序；如果l是$\Theta(p)$，那么两个算法的性能类似。习题9.13探讨最坏情况下的l值。

9.4 快速排序

基于比较的排序的复杂度下界是$\Theta(n \log n)$，在到目前为止介绍过的所有算法中，串行时间复杂度比这个下界更差。本节研究*快速排序*（quicksort）算法，其平均复杂度是$\Theta(n \log n)$。快速排序是串行计算机最常用的排序算法之一，因为它具有简单、低开销以及最优平均复杂度等优点。

快速排序是一种分而治之的算法，通过递归地将序列分成一些更小的子序列进行排序。假定待排序n个元素序列存放在数组$A[1 \ldots n]$中。快速排序包括两个步骤：划分和征服。在划
399 分步骤中，序列$A[q \ldots r]$被划分（重新排列）成两个非空的子序列$A[q \ldots s]$和$A[s+1 \ldots r]$，使得第一个子序列中的每个元素都小于或等于第二个子序列中的每个元素。在征服步骤中，递归地应用快速排序对子序列排序。因为子序列$A[q \ldots s]$和$A[s+1 \ldots r]$已排好序，并且第一个子序列的元素都小于第二个子序列的元素，于是整个序列就排好序了。

算法9-5 串行快速排序算法

```
1.  procedure QUICKSORT (A, q, r)
2.  begin
3.     if q < r then
4.     begin
5.        x := A[q];
6.        s := q;
7.        for i := q + 1 to r do
8.           if A[i] < x then
9.           begin
10.             s := s + 1;
11.             swap(A[s], A[i]);
12.          end if
13.       swap(A[q], A[s]);
14.       QUICKSORT (A, q, s);
15.       QUICKSORT (A, s + 1, r);
16.    end if
17. end QUICKSORT
```

如何把序列$A[q \ldots r]$划分成两个部分，并使得一部分的所有元素都小于另一部分的元素？通常的方法是从$A[q \ldots r]$中选择一个元素x，并用这个元素把序列$A[q \ldots r]$划分成两部分，一部

分的元素小于等于x，而另一部分的元素大于x。元素x被称为主元（pivot）。快速排序算法在算法9-5中介绍。这个算法任意选择序列$A[q \ldots r]$中的第一个元素作为主元。快速排序的操作如图9-15所示。

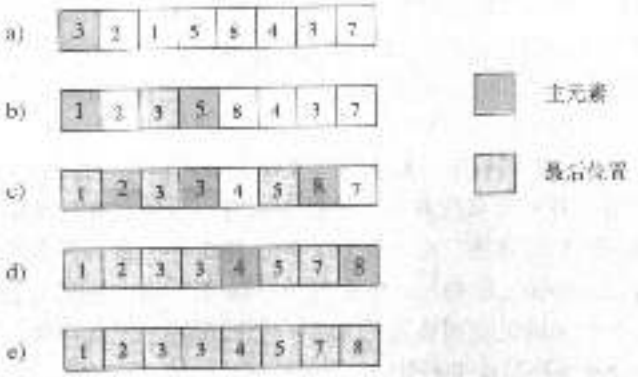


图9-15 排序序列大小$n = 8$的快速排序算法的例子

划分长度为k的序列的复杂度是$\Theta(k)$。快速排序的性能受到划分序列方法的重大影响。考虑这样一个例子，把大小为k的序列划分成两个子序列，一个子序列的大小为1，另一个为$k-1$。这是一种很差的划分。此例的运行时间由递归关系$T(n) = T(n-1) + \Theta(n)$给出，其解为$T(n) = \Theta(n^2)$。考虑另一种划分，把序列划分成两个大小几乎相等的分别具有$\lfloor k/2 \rfloor$和$\lceil k/2 \rceil$个元素的子序列。这种情况下，运行时间由递归关系$T(n) = 2T(n/2) + \Theta(n)$给出，其解为$T(n) = \Theta(n \log n)$。第二种划分产生最佳算法。虽然快速排序在最坏情况下的复杂度为$O(n^2)$，但其平均复杂度要好得多；对随机排列的输入序列进行快速排序时，需要进行的比较-交换操作的平均次数是$1.4\, n \log n$，它是渐近最优的。选择主元有多种方法，比如，序列元素较少时，主元可以是所有数的中间值，主元也可以是随机选择的元素。对于特定的输入序列，某些主元选择策略会比其他的更有优势。 400

9.4.1 并行快速排序

有多种方法可以将快速排序并行化。首先，考虑已在3.2.1节的递归分解中简单讨论过的一种自然的并行形式。从算法9-5的第14和15行可见，对QUICKSORT的每次调用，数组被分成两部分，每一部分以递归的方式求解。排序较小数组表示为两个可以并行求解的完全独立的子问题。因此，并行化快速排序算法的一个方法是：最初在单进程上执行，接着，当算法执行递归调用（第14和15行）时，把其中一个子问题分配给另一个进程。至此，每个进程都用快速排序算法对数组进行排序，并且分配它的子问题之一到其他的进程。当数组不能再细分时，算法结束。在结束后，每个进程持有数组的一个元素，并且通过遍历进程可以恢复排序的顺序，这一点将在后面介绍。这个并行快速排序算法用n个进程对n个元素进行排序。它的主要缺点是用单进程把数组$A[q \ldots r]$分成两个更小的数组$A[q \ldots s]$和$A[s+1 \ldots r]$。由于一个进程必需划分最初的数组$A[1 \ldots n]$，其运行时间以$\Omega(n)$为下界。这个形式不是成本最优的，因为它的进程处理时间乘积是$\Omega(n^2)$。

前面的并行形式的主要局限是，它串行执行划分步骤。我们将会看到，在其后的形式中，执行并行划分对于获得有效的并行快速排序是必需的。为看出这一点，考虑递归公式$T(n) =$

$2T(n/2) + \Theta(n)$，它给出最优主元选择的快速排序的复杂度。项$\Theta(n)$是由数组分块产生的。将
401 此复杂度与算法的整体复杂度$\Theta(n \log n)$比较。从这两种复杂度中，我们可以认为快速排序包含$\Theta(\log n)$步，每一步需要用$\Theta(n)$的时间划分数组。因此，如果划分步骤在时间$\Theta(1)$内完成，需要用$\Theta(n)$个进程，那么可能获得总的并行运行时间$\Theta(\log n)$，这样就得到成本最优的形式。但是，如果不对划分步骤并行化，为了维持成本最优，最好的办法就是在时间$\Theta(n)$内，只用$\Theta(\log n)$个进程对n个元素排序（参看习题9.14）。因此，并行化划分步骤有可能得到更快的并行形式。

在前一段中，我们得到启示，我们可以用$\Theta(n)$个进程在时间$\Theta(1)$内将n维数组划分成两个更小的数组。但是，对于大多数并行计算模型而言，要做到这一点很困难。唯一已知的算法是用于抽象PRAM模型的算法。由于通信开销，在实际共享地址空间和消息传递并行计算机中，划分步骤所需的时间比$\Theta(1)$长。下面几小节将给出3种不同的并行形式：一个用于CRCW PRAM，一个用于共享地址空间体系结构，一个用于消息传递平台。这三种形式都通过并行执行划分步骤对快速排序算法并行化。

9.4.2 用于CRCW PRAM的并行形式

下面介绍快速排序的一种并行形式，用于在n个进程的随意CRCW PRAM上对n个元素排序。回忆2.4.1节，CRCW PRAM是一种并发读、并发写的并行随机访问计算机，这种计算机以随意的方式解决写冲突。换句话说，当多个进程试图同时写同一个内存地址时，只允许一个随意选出的进程进行写操作，而其他的进程都被忽略。

执行快速排序的过程可以形象化为构建一棵二叉树。在这棵树中，主元就是根，小于或等于主元的元素放到左子树上，大于主元的元素放到右子树上。图9-16显示根据图9-15所示的快速排序算法构建的二叉树。通过在这棵树上执行有序遍历，可以得到有序序列。PRAM形式就是建立在快速排序的这种解释的基础上。

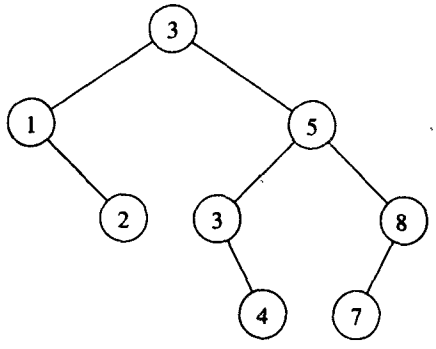


图9-16 由执行快速排序算法产生的二叉树

注：树的每一层代表一个不同的数组划分迭代。如果主元选择是最优的，那么树的高度是$\Theta(\log n)$，这也是迭代的次数。

算法通过选择主元元素开始，将数组划分成两部分：一部分是小于主元的那些元素，另一部分是大于主元的元素。然后，后面的每个主元元素被并行选择，每个元素分配给一个新的子数组。这种形式并不重排元素；相反，由于所有进程可以在固定时间内读取主元，它们知道元素应该分配到哪个子数组（较小的或是较大的子数组）。这样它们可以进行下一次迭代。

算法9-6 用于CRCW PRAM并行快速排序形式的二叉树构建程序

```
1.  procedure BUILD_TREE (A[1 ... n])
2.  begin
3.     for each process i do
4.     begin
5.        root := i;
6.        parent_i := root;
7.        leftchild[i] := rightchild[i] := n + 1;
8.     end for
9.     repeat for each process i ≠ root do
10.    begin
11.       if (A[i] < A[parent_i]) or
             (A[i] = A[parent_i] and i <parent_i) then
12.       begin
13.          leftchild[parent_i] := i;
14.          if i = leftchild[parent_i] then exit
15.          else parent_i := leftchild[parent_i];
16.       end for
17.       else
18.       begin
19.          rightchild[parent_i] := i;
20.          if i = rightchild[parent_i] then exit
21.          else parent_i := rightchild[parent_i];
22.       end else
23.    end repeat
24. end BUILD_TREE
```

构建二叉树的算法如算法9-6所示。待排序的数组存放在数组$A[1 \ldots n]$，元素$A[i]$被分配给进程i。数组$leftchild[1 \ldots n]$和$rightchild[1 \ldots n]$记录主元的子元素，对于每个进程，局部变量$parent_i$存放其元素为主元的进程标号。最初，所有的进程将各自的进程标号写到第5行代码的变量$root$中。由于并发写操作是随意的，实际上只有其中一个标号真正被写到$root$中。对于每个进程i，值$A[root]$被用来作为第一个主元，变量$root$的值被复制到变量$parent_i$中。接下来，那些数组元素的值小于$A[parent_i]$的进程将它们的进程标号写到$leftchild[parent_i]$，而那些值大于$A[parent_i]$的进程将进程标号写到$rightchild[parent_i]$。这样，所有那些元素属于较小划分的进程将它们的标号写到$leftchild[parent_i]$，元素属于较大划分的进程将标号写到$rightchild[parent_i]$。由于并发写操作是随意的，只有两个值被写到这些位置——一个是$leftchild[parent_i]$，另一个是$rightchild[parent_i]$。在下一次迭代中，数组被分成两个更小的数组，而这两个值将变成持有主元元素的进程的标号。算法继续进行，直到n个主元元素被全部选出。当进程的元素变成主元时，该进程就退出。二叉树的构建如图9-17所示。在算法的每一次迭代过程中，构建树的一层所需时间为$\Theta(1)$。这样，构建二叉树算法的平均时间复杂度为$\Theta(\log n)$，因为树的平均高度是$\Theta(\log n)$（参看习题9.16）。

402 ~ 403

建立二叉树后，算法确定排序后的数组中每个元素的位置。算法遍历二叉树，并且在任一元素的左子树和右子树中保存元素个数的计数。最后，在时间$\Theta(1)$内把每个元素放到合适的位置，数组就排好序了。遍历二叉树和确定每个元素位置的算法留作练习（习题9.15）。在

有n个进程的PRAM中，这个算法的平均运行时间是$\Theta(\log n)$。这样，其总的进程-时间乘积为$\Theta(n \log n)$，它是成本最优的。

a)

1	2	3	4	5	6	7	8
33	21	13	54	82	33	40	72

b) root = 4

c)

	1	2	3	4	5	6	7	8
leftchild				1				
rightchild				5				

d)

	1	2	3	4	5	6	7	8
leftchild	2			1	8			
rightchild	6			5				

e)

	1	2	3	4	5	6	7	8
leftchild	2	3		1	8			
rightchild	6			5		7		

f)

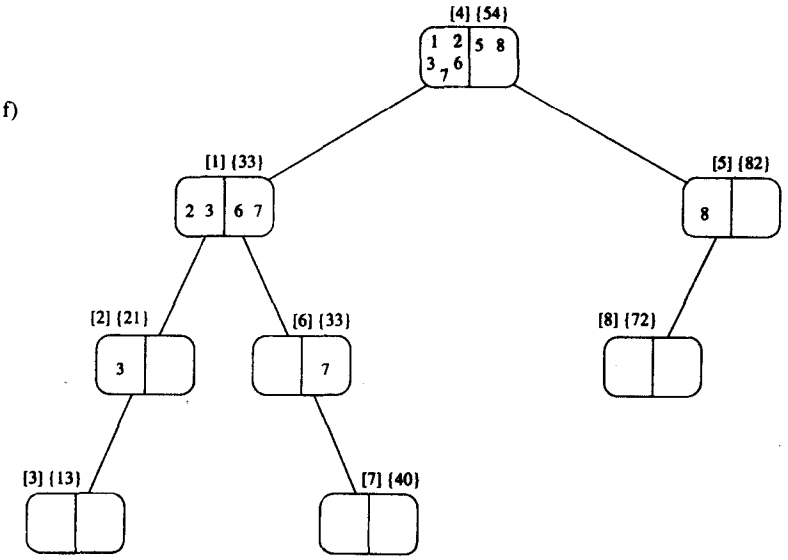


图9-17 对a)中所示数组执行PRAM算法的情况

注：算法执行中数组*leftchild*和*rightchild*显示在c)、d)、e）中。f）显示由算法建立的二叉树。每个节点由进程（在方括号中）标记，并且元素存放在那个进程里（在大括号中）。元素是主元。在每个节点上，具有比主元小的元素的进程归入节点左边的组，元素大于主元的进程归入右边的组。这两个组构成原始数组的两个划分。对于每一划分来说，主元元素是从两个组中随机选出的，这两个组构成节点的子节点。

9.4.3 用于实际体系结构的并行形式

现在转向更实际的并行体系结构——由互连网络连接的有p个进程的系统。首先，我们主要讨论共享地址空间系统的算法开发，然后，说明如何改变这个算法使其适用于消息传递系统。

1. 共享地址空间并行形式

共享地址空间系统快速排序形式的工作原理如下：令A是有n个元素的需要排序的数组，p是进程数目。对每个进程分配有n/p个元素的连续块，并且进程的标号决定排序后序列的全局

顺序。令A_i是分配给进程P_i的元素的块。

算法从选择主元开始，并且将这个主元向所有进程广播。每个进程P_i接收到主元后，对分配的元素块进行重排，将它分成两个子块，一个子块S_i包含所有的小于主元的元素，另一个子块L_i包含所有大于主元的元素。这个局部重排用快速排序的分裂循环（collapsing the loops）方法来实现。算法的下一步重新排列原始数组A中的元素，使得所有小于主元的元素（即$S=\bigcup_i S_i$）都存放在数组的开始位置，而所有大于主元的元素（即$L=\bigcup_i L_i$）都存放在数组的末端。 404 ~ 405

当全局重排完成后，算法着手将所有进程划分为两个组，把那些较小的元素S的排序任务分配给第一组，较大的元素L的排序任务分配给第二组。所有这些步骤都通过递归调用并行快速排序算法来完成。注意，通过同时划分进程和原始数组，每组进程都可以独立地进行。当某一子块的元素只分配给一个进程时，递归结束，此时进程使用串行快速排序算法对元素排序。

根据块S和L的相对大小将进程划分为两个组。具体就是：前面的$\lceil |S| p/n+0.5 \rceil$个进程用来对较小元素组$S$排序，剩下的进程用来对较大元素组$L$排序。注意，上面公式中的数0.5用来保证以最均衡的方式分配进程。

例9.1 有效的并行快速排序

图9-18用20个整数和5个进程的例子说明这个算法。第一步，围绕着主元元素（本例中的主元元素是7），每个进程局部重排各自最初负责的四个元素，使得小于或等于主元的元素被移到数组的局部分配部分的开始处（如图中阴影部分所示）。当局部重排完成后，进程再执行全局重排，得到如图中所示的第三个数组（后面将简单地讨论如何执行）。在第二步，进程被分成两个组。第一组包括进程$\{P_0, P_1\}$，负责对那些值小于等于7的元素排序。第二组包括进程$\{P_2, P_3, P_4\}$，负责对值大于7的元素排序。注意这两个进程组的大小是根据大于或小于主元数组的相对大小决定的。现在，对每个进程组以及子数组，递归地重复上面的主元选择、局部重排以及全局重排的步骤，直到子数组分配给一个进程为止，这时进程就在本地进行排序。同时要注意，这些最后本地子数组的大小通常是不同的，因为它们取决于被选出来作为主元的元素的多少。 ■

为了全局重排A中的元素为或小或大的子数组，在重排结束时，我们需要知道每个A中的每个元素最终被排在了哪里。图9-19的底部显示这种重排。用这个方法，按照进程标号的递增顺序，通过连接所有进程中的不同块S_i，就能得到S。同样，以同样的顺序连接不同的块L_i就能得到L。因此，对于进程P_i而言，它的S_i子块的第j个元素将存放在$\sum_{k=0}^{i} |S_k| + j$位置，它的L_i子块的第j个元素将存放在$n - \sum_{k=i}^{p-1} |L_k| - j$位置。 406

用4.3节介绍的前缀和操作很容易计算出这些位置。要计算出两个前缀和，一个包括S_i子块的大小，另一个包括L_i子块的大小。令Q和R是大小为p的数组，分别存放这两个前缀和。它们的元素为

$$Q_i = \sum_{k=0}^{i-1} S_i \text{ 和 } R_i = \sum_{k=0}^{i-1} L_i$$

注意对于每个进程P_i，Q_i是最终数组中存储比主元小的元素的开始位置，而R_i是最终数组中存储比主元大的元素的结束位置。一旦确定这些位置，使用大小为n的辅助数组A'，就很

容易对数组A进行整体重排。这些步骤如图9-19所示。需要注意的是，上面的前缀和定义与4.3节介绍的略有不同：这是指计算出的位置Q_i（或R_i）并不包括S_i（或L_i）本身。这种类型的前缀和有时候称为*无包含前缀和*。

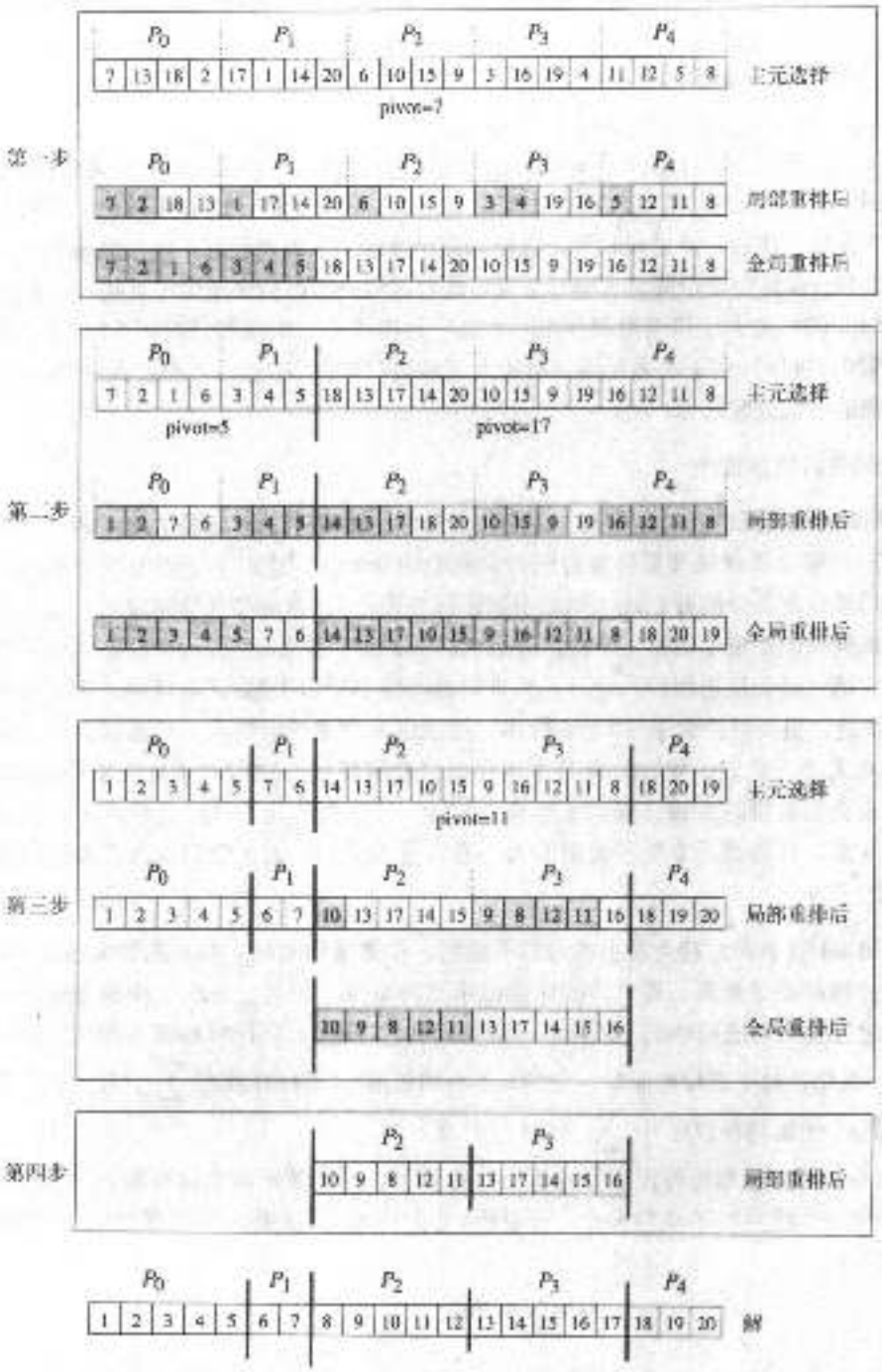


图9-18 执行有效的共享地址空间的快速排序算法的例子

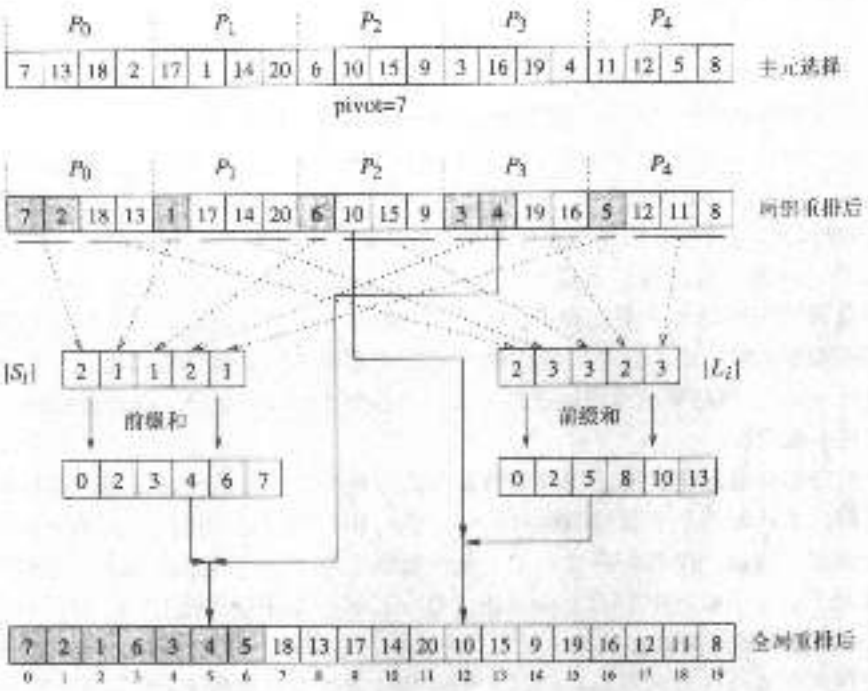


图9-19 数组的有效全局重排

407 ~ 408

分析 快速排序算法共享地址空间形式的复杂性取决于两方面：一方面是将某一数组划分成两个子数组的时间，这两个子数组中，一个的所有元素小于主元，另一个的所有元素大于主元；另一方面是选择各种主元而导致均衡划分的等级。在本节，为了简化分析，假定主元选择总是产生均衡划分。正确的主元选择及其对总并行性能的影响在9.4.4节中说明。

对于给定有n个元素和p个进程的数组，快速排序算法的共享地址空间形式需要执行四个步骤：i) 确定并广播主元；ii) 局部重排分配到每个进程的数组；iii) 在全局重排后的数组中，确定本地的元素将要存放的位置；iv) 执行全局重排。第一步对共享地址空间广播用有效的递归倍增方法，可以在时间$\Theta(\log p)$内执行。第二步用传统的快速排序算法围绕主元元素分裂，可以在时间$\Theta(n/p)$内完成。第三步用前缀和操作可以在时间$\Theta(\log p)$内完成。最后，第四步需要将本地元素复制到最后的目的地，至少需要$\Theta(n/p)$时间。这样，划分n个元素的数组所需的总体时间复杂度为$\Theta(n/p) + \Theta(\log p)$。对于两个子数组的每一个，这个过程要在一半的进程上递归地重复，直到数组被分成p部分为止，那时，每个进程用串行快速排序算法对分配给它的数组进行排序。因此，并行算法的总体时间复杂度是：

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{局部排序}} + \overbrace{\Theta\left(\frac{n}{p}\log p\right) + \Theta(\log^2 p)}^{\text{数组分裂}} \tag{9-8}$$

上面的公式的通信开销反映在项$\Theta(\log^2 p)$中，它导致总等效率函数为$\Theta(p\log^2 p)$。有趣的是，注意算法的整体可扩展性由执行主元广播与前缀和操作的时间决定。

2. 消息传递并行形式

消息传递系统的快速排序形式和常用的共享地址空间形式的结构相同。但是，共享地址空间中数组A和全局重排数组A'都存放在共享内存中，并且可以被所有进程访问，现在这两个数组都显式地分布在各个进程中。这就使得划分A的任务多少更困难一些。

特别是，在消息传递版本的并行快速排序中，每个进程都存放数组A的n/p个元素。使用两阶段方法，可以把数组围绕某一主元元素划分开。在第一阶段（与共享地址空间形式相似），
409 存放在进程P_i的本地数组A_i被局部划分成值小于主元和值大于主元的子数组S_i和L_i。在第二阶段，算法首先决定哪些进程负责递归地对小于主元的子数组（也就是$S=\bigcup_i S_i$）进行排序，哪些进程负责递归地对大于主元的子数组（也就是$L=\bigcup_i L_i$）进行排序。当这些做完后，进程将它们的数组S_i和L_i发送给对应的进程。此后，进程已经被划分成两组，一个是对于S的，另一个是对于L的，然后算法递归地进行下去。当每个子数组分配给一个单进程时，递归结束，至此排序在本地进行。

用于决定哪些进程负责排序S和L的方法与共享地址空间形式的方法是一样的，也就是尝试划分进程以求匹配两个子数组的相对大小。令p_S和p_L分别是为排序S和L所分配的进程的数目。p_S个进程中的每一个都将存放小于主元子数组中的$|S|/p_S$个元素，而p_L个进程中的每一个都将存放大于主元子数组中的$|L|/p_L$个元素。每个进程P_i用于确定把它的S_i和L_i元素发送到哪里的方法，所遵循的策略和在共享地址空间形式的总体策略一样。也就是说，不同的S_i（或L_i）子数组将存放在依进程号而定的S（或L）的连续位置。负责这些元素的实际的进程，是由划分S（或L）成p_S（或p_L）个相等大小的部分决定的，并且可以使用前缀和操作来计算。注意每个进程P_i可能需要划分它的S_i（或L_i）子数组成多个部分，并且将每部分发送给不同的进程。这种情况的出现是因为它的元素可能被分配到S（或L）中跨越超过一个进程的位置。总的说来，每个进程可能必须发送其元素到两个不同的进程，但是出现需要两个以上划分的情况也是很有可能的。

分析 对快速排序消息传递形式的分析，将反映共享地址空间形式的相应的分析。

考虑有p个进程和$O(p)$对分带宽的消息传递并行计算机。划分n个元素的数组所需要的时间包括，广播主元素的时间$\Theta(\log p)$，划分数组本地分配部分的时间$\Theta(n/p)$，执行前缀和与决定进程的划分大小和不同S_i、L_i子数组的目标的时间$\Theta(\log p)$，以及发送和接收各种数组所需的时间。最后一步取决于进程如何映射到系统结构上，以及与每个进程需要与其通信的最大进程数。通常，这种通信步骤包含多对多私自通信（因为某一进程可以终止从所有其他进程接收元素），其时间复杂度的下界是$\Theta(n/p)$。因此，划分的总时间复杂度是$\Theta(n/p)+\Theta(\log p)$，这与共享地址空间形式的复杂度非常相似。结果，总运行时间也与公式(9-8)中的结果相同，
410 并且算法有相似的等效率函数$\Theta(p\log^2 p)$。

9.4.4 主元选择

在并行快速排序算法中我们没有涉及主元选择。主元选择是很困难的，而且它对算法性能有显著的影响。考虑第一个主元是序列中最大元素的情况。这时，第一次分裂后有一个进程只分配到一个元素，剩下的$p-1$个进程将分配剩下的$n-1$个元素。因此，将面临一个问题：序列只减少了一个元素，但有$p-1$个进程参与排序操作。虽然这是人为的例子，但它说明并行化快速排序算法遇到的一个重要问题。理想状态下，分裂应使得每个划分具有原数组的并非

无足轻重的部分。

选择主元的一个方法是随机选择：在第i个划分，每个进程组的一个进程随机选择其中的一个元素作为这个划分的主元。这与串行快速排序算法中的随机主元选择类似。虽然这种方法适用于串行快速排序，但它不适合并行形式。通过一个主元选择不当的例子，就能明白其原因。在串行快速排序中，这将导致一个子序列明显大于另一个子序列的划分。如果后来的所有主元选择都是好的，一个坏的主元也会增加总工作量，但增加量至多等于子序列长度。这样，不会严重降低串行快速排序的性能。然而，在并行形式中，一个坏的主元可能导致有的进程空闲的划分，而且这种空闲在算法的执行过程中一直存在。

如果每个进程的元素最初都均匀分布，那么就能导出一个好的主元选择方法。此时，最初存放在每个进程的n/p个元素构成所有n个元素的典型样本。换句话说，每个有n/p个元素的子序列的中值，非常接近于整个n元素序列的中值。假设最初分布相同时，为什么这是一个很好的主元选择方案？由于每个进程上的元素分布与n个元素的总体分布相同，将第一步选择的中值作为主元就是整体中值很好的近似。由于选择的主元很接近于整体中值，每个进程中大约一半元素小于主元，而另一半元素大于主元。因此，第一次分裂产生两个划分，每个划分大约有$n/2$个元素。同样，与初始列表中$n/2$个较小（较大）元素一样，分配给组中的每个进程的元素具有同样的分布，这里的组是负责对小于主元的元素进行排序的进程组（和负责对大于主元的元素进行排序的进程组）。这样，分裂不仅能保持负载平衡，而且可以保证进程组中的元素是均匀分布的。因此，对进程的子组应用同样的主元选择方案可以不断选择到好的主元。

但是，我们真正可以假定每个进程里的n/p个元素与总序列中的元素有相同的分布吗？答 411
案要根据实际应用而定。在某些应用中，随机选择主元或者选择中值作为主元能很好运行，但在另一些应用中两种方案都不能提供好的性能。习题9.20和9.21提出另两种主元选择方案。

9.5 桶和样本排序

桶排序（bucket sort）算法是对值均匀分布在区间$[a, b]$上的n元素数组进行排序的常见串行排序算法。这个算法中，区间$[a, b]$被分成m个相等大小的子区间，称为桶（bucket），每个元素被放在适当的桶中。由于n个元素均匀地分布在区间$[a, b]$上，每个桶中的元素数目大约是n/m。算法然后在每个桶中对这些元素进行排序，生成排好序的序列，算法运行时间为$\Theta(n \log (n/m))$。如果$m = \Theta(n)$，则为线性运行时间$\Theta(n)$。注意，桶排序的时间复杂度之所以可以达到这么低，是因为假定待排序的n个元素均匀地分布在区间$[a, b]$上。

并行化桶排序很简单。令n是待排序的元素数目，p是进程数目。一开始，分配给每个进程含有n/p个元素的块，而将桶的数目选择为$m = p$。桶排序的并行形式有三个步骤。在第一步，每个进程将其含n/p个元素的块划分成p个子块，分别放入p个桶中。这是可能的，因为每个进程都知道区间$[a, b)$以及由此而来的每个桶的区间。在第二步，每个进程发送其子块到适当的进程。执行完这一步后，每个进程只有属于分配给它的桶的元素。第三步，每个进程使用最优串行排序算法对它的桶进行内部排序。

但是，假设输入元素平均分布在区间$[a, b]$上是不现实的。在多数情况下，实际的输入可能不具有这样的分布，或者其分布是未知的。因此，使用桶排序可能会导致各个桶中的元素明显不同，从而降低其性能。这种情况下，一种称为*样本排序*（sample sort）的算法能明显提

升性能。样本排序的思想很简单。从n元素序列中选出大小为s的样本，桶的范围由排序样本以及从结果中选取$m-1$个元素决定。这些元素（称为分割器）将样本分成m个大小相等的桶。在定义桶之后，算法的执行与桶排序相同，样本排序的性能取决于样本的大小s，以及它从n元素序列中选择样本的方法。

下面考虑一种分割器选择方案，对于所有的桶它能保证在结束时每个桶中的元素数目大致相等。令n是待排序的元素数目，m是桶的数目。方案的工作原理如下：将n个元素分成m块，每块大小为n/m。用快速排序对每个块进行排序。从每个排序后的块中选择$m-1$个均匀分隔的
412 元素。从所有块中选择出的第$m(m-1)$个元素就是用来决定桶的样本。这个方案保证每个桶结束时的元素数目少于$2n/m$（参考习题9.28）。

如何并行化分割器选择方案？令p是进程的数目。和桶排序中的一样，令$m = p$；这样，在算法的结尾，每个进程只包含仅属于一个桶的元素。对每个进程分配n/p个元素的块，进程对这个块进行串行排序。然后，从排序后的块中选择出$p-1$个均匀分隔的元素。每个进程将其$p-1$个样本元素发送到一个进程，此进程称为P_0。进程P_0再对$p(p-1)$个样本元素进行串行排序，并选择$p-1$个分裂器。最后，进程P_0将$p-1$个分裂器广播给所有其他进程。至此，算法将按与桶排序相同的方法继续下去。此算法如图9-20所示。

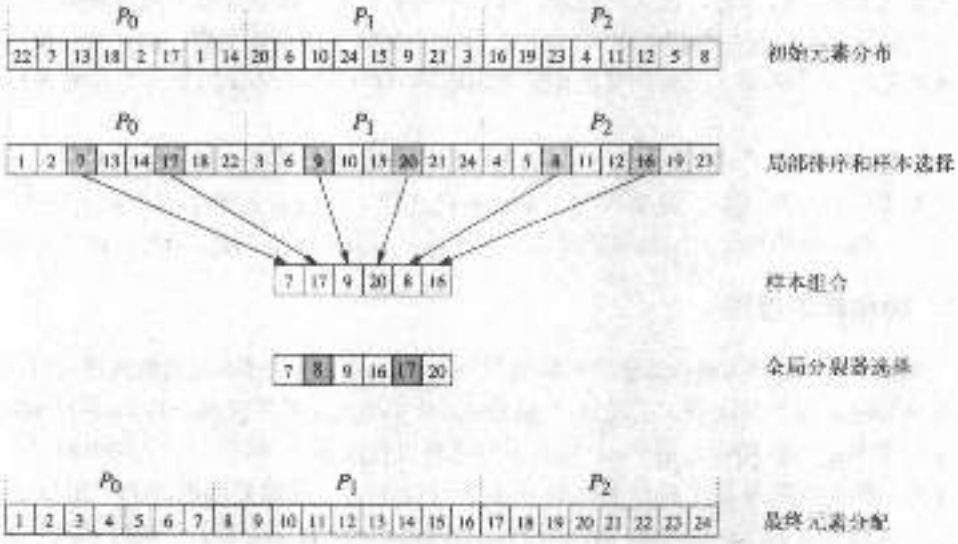


图9-20 在三个进程上对一个有24个元素的数组进行样本排序的例子

分析 下面，我们在p个进程和对分带宽为$O(p)$的消息传递计算机上，分析样本排序的时间复杂度。

对n/p个元素进行内部排序所需的时间是$\Theta((n/p)\log(n/p))$。选择$p-1$个样本元素所需的时间是$\Theta(p)$。发送$p-1$个元素到进程P_0和收集操作类似（参考4.4节），所需时间是$\Theta(p^2)$。在进程P_0对$p(p-1)$个样本元素进行内部排序所需时间是$\Theta(p^2\log p)$，而选择$p-1$个分割器所需时间是$\Theta(p)$。使用一对多广播将$p-1$个分割器发送到其他的进程（见4.1节），所需时间是$\Theta(p \log p)$。每个进程可以通过执行$p-1$次二分搜索，将$p-1$个分割器插入到它的局部排序后的含有n/p个元
413 素的块中。这样，每个进程将其块划分成p个子块，每个桶中存放一个子块。划分所需时间为$\Theta(p \log(n/p))$。每个进程然后将子块发送到相应的进程(即桶中)。这一步的通信时间很难精确

计算，因为它取决于通信的子块大小，而这些子块的大小可以在0和n/p之间任意改变。因此，通信时间的上界是$O(n) + O(p \log p)$。

如果假定存放在每个进程中的元素是均匀分布的，那么每个子块大约有$\Theta(n/p^2)$个元素。这种情况下，并行运行时间为

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{局部排序}} + \overbrace{\Theta\left(p^2\log p\right)}^{\text{排序样本}} + \overbrace{\Theta\left(p\log\frac{n}{p}\right)}^{\text{块划分}} + \overbrace{\Theta(n/p) + O(p\log p)}^{\text{通信}} \tag{9-9}$$

这时等效率函数是$\Theta(p^3 \log p)$。如果使用双调排序对$p(p-1)$个样本元素进行排序，那么对样本进行排序的时间将是$\Theta(p \log p)$，而等效率函数降低到$\Theta(p^2 \log p)$（见习题9.30）。

9.6 其他排序算法

在本章简介中曾提到过，排序算法有很多种，但不能在此一一讨论。不过，本节将简要介绍另两种排序算法，这两种算法不论在理论上还是实际上都是很重要的。我们对这两种方案作简要讨论。从书目评注（见9.7节）中可以找到这两种算法以及其他算法的参考文献。

9.6.1 枚举排序

到目前为止，讨论的所有算法都基于比较-交换操作。本节考虑基于*枚举排序*（enumeration sort）的算法，不使用比较-交换。枚举排序的基本思想是确定每个元素的秩。一个元素a_i的*秩*(rank)是指待排序序列中小于a_i的元素的数目。a_i的秩可以用来将其放置在排好序序列中正确的位置。基于枚举排序的并行算法有好几种。这里我们介绍适用于CRCW PRAM机型的算法。这形式可在时间$\Theta(1)$内使用n^2个进程对n个元素进行排序。

假定对CRCW PRAM同一内存单元的并发写将导致在这个单元存储所有被写值的和(见2.4.1节)。考虑在二维网格中排列n^2个进程的情况。算法包含两步：第一步，进程的每个列j计算值小于a_j的元素的数目；第二步，每个第一行的进程$P_{1,j}$将a_j放到由进程的秩决定的正确位置。算法如算法9-7所示。它用辅助的数组$C[1 \ldots n]$存放每个元素的秩。算法中的关键步骤是 414
第7行和第9行。在那里，如果元素$A[i]$小于$A[j]$，则每个进程$P_{i,j}$在$C[j]$中写1，否则写0。由于添加-写（additive-write）冲突分辨格式，这些指令的作用是对值小于$A[j]$的元素进行计数，从而计算出它的秩。算法的运行时间是$\Theta(1)$。习题9.26中讨论对此算法的修改，使之适用于不同的并行结构。

算法9-7 在CRCW PRAM上用添加-写冲突分辨的枚举排序

```
1.  procedure ENUM_SORT (n)
2.  begin
3.     for each process P1,j do
4.        C[j] := 0;
5.     for each process Pi,j do
6.        if (A[i] < A[j]) or (A[i] = A[j] and i < j) then
7.           C[j] := 1;
8.        else
9.           C[j] := 0;
```

```
10.      for each process P_{1..j} do
11.          A[C[j]] := A[j];
12.  end ENUM_SORT
```

9.6.2 基数排序

基数排序（radix sort）算法依赖于待排序元素的二进制表示。令b是元素二进制表示中的位数。基数排序算法每次检测待排序元素的r位，其中$r < b$。基数排序需要b/r次迭代。在第i次迭代中，根据其第i个有r位的最低有效块来排序。为了使基数排序能正确进行，b/r次排序中的每一次都必须是稳定的。一个排序算法是稳定的，是指输出元素保持输入元素的顺序和值不变。基数排序是稳定的，是指当任何两个r位的块相等时保持这两个块的输入顺序不变。使用枚举排序（参考9.6.1节）是实现中间b/r次2^r基数排序最常用的方法，因为可能的取值范围$[0 \ldots 2^r - 1]$很小。对于这样的情况，枚举排序更优于任何基于比较的排序算法。

在n个进程的消息传递计算机上考虑n个元素基数排序的并行形式。并行基数排序算法如算法9-8所示。算法的主循环（第3～17行）对r个位的块执行b/r次枚举排序。枚举排序由函数*prefix_sum*()和*parallel_sum*()执行，这两个函数与4.1节和4.3节中描述的函数相似。在内层循环（第6～15行）的每次迭代中，基数排序决定r个位的值为j的元素的位置。基数排序将所有具有
415 这同一值的元素求和，然后再分配到各个进程。变量*rank*保存每个元素的位置。在循环的结尾（第16行），每个进程将其元素发送到相应的进程。进程的标号决定排序后的元素的全局顺序。

算法9-8 并行基数排序算法

```
1.   procedure RADIX_SORT(A, r)
2.   begin
3.       for i := 0 to b/r − 1 do
4.       begin
5.           offset := 0;
6.           for j := 0 to 2^r − 1 do
7.           begin
8.               flag := 0;
9.               if the i^th least significant r-bit block of A[P_k] = j then
10.                  flag := 1;
11.              index := prefix_sum(flag)
12.              if flag = 1 then
13.                  rank := offset + index;
14.              offset := parallel_sum(flag);
15.          endfor
16.          each process P_k send its element A[P_k] to process P_rank;
17.      endfor
18.  end RADIX_SORT
```

注：在这个算法中，待排序数组$A[1 \ldots n]$的每个元素被分配到一个进程。函数*prefix_sum* ()计算特征变量的前缀和，而函数*parallel_sum* ()返回特征变量的总和。

如4.1节和4.3节所示，在n个进程的消息传递计算机上，*parallel_sum*()和*prefix_sum*()运算的复杂度为$\Theta(\log n)$。第16行的通信步骤的复杂度是$\Theta(n)$。这样，这个算法的并行运行时间为

$$T_P = (b/r)2^r(\Theta(\log n) + \Theta(n))$$

9.7 书目评注

Knuth[Knu73]讨论排序网络及其历史。排序网络能否在时间$O(\log n)$内对n个元素进行排序的问题在很长一段时间内未得到解决。1983年，Ajtai，Komlos和Szemeredi[AKS83]发现了可以在时间$O(\log n)$内使用$O(n \log n)$个比较器对n个元素进行排序的排序网络。不过，他们排序网络的常数太大（高达数千），因而是不可行的。Batcher[Bat68]发现了双调排序网络，他还发现了奇偶排序网络。这些是第一批能够在时间$O(\log^2 n)$内对n个元素进行排序的网络。Stone[Sto71]将双调排序映射到完全混洗的互连网络，在时间$O(\log^2 n)$内使用n个进程对n个元 416
素进行排序。Siegel[Sie77]指出双调排序也可以在超立方体上执行，所需时间是$O(\log^2 n)$。Johnsson[Joh84]和Fox等[FJL+88]讨论双调排序的基于块的超立方体形式。算法9-1就采用[FJL+88]的方法。Thompson和Kung[TK77]介绍格网连接计算机上的双调排序的混洗行优先索引形式。他们还说明如何把奇偶合并排序与蛇形行优先索引一起使用。Nassimi和Sahni[NS79]介绍一种格网的行优先索引的双调排序形式，其性能与混洗行优先索引形式一样。格网奇偶合并的改进版本由Kumar和Hirschberg[KH83]提出。比较-分裂操作可用多种方法来实现，Baudet和Stevenson[BS78]介绍一种执行这种操作的方法。Hsiao和Menon[HM80]发现了基于双调排序（习题9.1）的执行比较-分裂操作的另一种方法，这种方法不需要附加的内存。

Knuth[Knu73]讲述奇偶转换排序。Knuth[Knu73]和Kung[Kun80]给出一些较早的使用奇偶转换的并行排序的参考文献。奇偶转换排序的基于块的扩展算法由Baudet和Stevenson[BS78给出。另一个基于块的奇偶转换排序的变体使用双调合并-分裂，由DeWitt，Friedland，Hsiao和Menon[DFHM82]给出。他们的算法使用p个进程，运行时间是$O(n+n \log(n/p))$。与Baudet和Stevenson[BS78]的算法相比，此算法更快，但是每个进程需要$4n/p$个存储单元，DeWitt等的算法只需要$(n/p)+1$个存储单元执行比较-分裂操作。

9.3.2节讲述的希尔排序算法由Fox等[FJL+88]提出。他们指出，当n增加时最终奇偶转换出现最差性能（换句话说，即需要p个阶段）的概率将会减少。Quinn[Qui88]给出另一种基于原始串行算法[She59]的希尔排序算法。

串行快速排序算法由Hoare[Hoa62]提出。Sedgewick[Sed78]提供一批有关实现算法细节及其如何影响性能的很好的参考资料。Robin[Rob75]提出并分析随机主元选择方案。Sedgewick[Sed78]提出单进程上的序列划分算法，而Raskin[Ras78]、Deminet[Dem82]以及Quinn[Qui88]则使用该算法的并行形式。CRCW PRAM算法（9.4.2节）由Chlebus和Vrto[CV91]提出。已经有多人提出PRAM和共享地址空间并行计算机上的许多其他基于快速排序的算法，能够使用$\Theta(n)$个进程在时间$\Theta(\log n)$内对n个元素进行排序。Martel和Gusfield[MG89]对CRCW PRAM提出一个快速排序算法，平均需要$\Theta(n^3)$空间。Heidelberger，Norton和Robinson[HNR90]发现了有读取-相加能力的适用于共享地址空间并行计算机的算法。他们的算法的平均运行时间是$\Theta(\log n)$，并可改用在商用共享地址空间计算机上。习题9.17讲述的快速排序的超立方体形式由Wagar[Wag87]提出。他的超快速排序算法使用基于中值的主元选择方案，并且假定每个进 417
程中的元素都有相同的分布。他的试验结果显示，超快速排序比超立方体上的双调排序快。Fox等[FJL+88]实现了另一种主元选择方案（习题9.20）。当每个进程上的元素分布不均匀时，此方案显著地改进超快速排序的性能。Plaxton[Pla89]描述p个进程超立方体上的快速排序算法，能在时间$O((n \log n)/p + (n \log^{3/2} p)/p + \log^3 p \log(n/p))$内对$n$个元素进行排序。此算法使用需要时间$O((n/p)\log\log p + \log^2 p \log(n/p))$的并行选择算法确定最好的主元。快速排序的格网形式

(习题9.24) 由Singh, Kumar, Agha和Tomlinson[SKAT91a]提出。他们也对算法作了修改，使得每一步的复杂度降低$\Theta(\log p)$倍。

1956年Isaac和Singleton首次提出了串行桶排序算法。Hirschberg[Hir78]提出EREW PRAM机型上的桶排序算法。算法用n个进程，在$\Theta(\log n)$时间对n个范围在[0 ... $n-1$]的元素进行排序。此算法的副作用是消去重复元素。算法需要$\Theta(n^2)$的空间。Hirschberg[Hir78]对算法作了推广，使得重复的元素仍保留在排序后的数组中。推广的算法使用$n^{1+1/k}$个进程，需要时间$\Theta(k \log n)$对n个元素进行排序，其中k是任意整数。

串行样本排序算法由Frazer和McKellar[FM70]提出。并行样本排序算法（9.5节）由Shi和Schaeffer[SS90]提出。人们还提出了几种不同并行结构上的样本排序并行形式。Abali, Ozguner和Bataineh[AOB93]介绍一种分割器选择方案，保证在每个桶结束时的元素数目为n/p。在p个进程的超立方体上，他们的算法平均需要时间$\Theta((n \log n)/p + p \log^2 n)$对$n$个元素进行排序。Reif和Valiant[RV87]介绍一种样本排序算法，在n个进程的超立方体连接计算机上，以很高概率在时间$O(\log n)$内对n个元素排序。Won和Sahni[WS88]以及Seidel和George[SG88]提出样本排序的变体的一种并行形式，称为*箱排序*（bin sort）[FKO86]。

人们也提出了其他多种并行排序算法。各种并行排序算法可以有效地在PRAM机型或共享地址空间计算机上执行。Akl[Akl85]、Borodin和Hopcroft[BH82]，Shiloach和Vishkin[SV81]，Bitton、DeWitt、Hsiao和Menon[BDHM84]都对此作了很好的评述。Valiant [Val75]提出了用于共享地址空间SIMD型计算机上的排序算法，使用合并进行排序。它用$n/2$个进程对n个元素进行排序的时间为$O(\log n \log \log n)$。Reischuk[Rei81]最早提出一种算法，在n进程的PRAM上对n个元素进行排序的时间为$\Theta(\log n)$。Cole[Col88]提出了一个并行合并排序算法，在EREW PRAM上对n个元素进行排序的时间为$\Theta(\log n)$。Natvig[Nat90]证明了渐进公式中隐含一个非常大的常量。实际上，只要n小于7.6×10^{22}，用时为$\Theta(\log^2 n)$的双调排序就要比用时$\Theta(\log n)$的合并排序好。Plaxton[Pla89]提出了一种超立方体排序算法，称为*光滑排序*（smoothsort），在那种结构上算法运行比过去已知的所有算法都快。Leighton[Lei85a]提出了一种称为*列排序*
418 (columnsort) 的排序算法，包含一系列排序以及随后的初等矩阵运算。列排序是Batcher的奇偶排序的一种推广。Nigam和Sahni[NS93]介绍了基于Leighton的列排序的算法，适用于含有总线的可重配置格网，在n^2个进程的格网上对n个元素进行排序的时间为$O(1)$。

习题

9.1 考虑执行比较-分裂操作的下述方法：令$x_1, x_2, \ldots, x_n$是在进程P_i以递增顺序存放的元素，令$y_1, y_2, \ldots, y_n$是在进程P_j以递减顺序存放的元素。进程P_i发送x_1到P_j，进程P_j将其与y_1进行比较，然后将较大的元素发送回进程P_i并保存较小的元素。这同一过程在各个对(x_2, y_2), (x_3, y_3), ..., (x_k, y_k)之间重复进行。如果对于任何一对(x_l, y_l)，$1 \leqslant l \leqslant k$，出现$x_l \geqslant y_l$，就不需要再交换。最后，每个进程对自己的元素排序。说明这个方法正确地执行比较-分裂操作。分析其运行时间，并比较这个方法与课文中提出的方法的优缺点。这种方法更适合于MIMD并行计算机还是SIMD并行计算机？

9.2 说明9.1节中对元素块定义的$\leqslant$关系是一个偏序关系。

提示：一个关系是*偏序*（partial ordering）的，如果它是自反的、反对称的和传递的。

9.3 考虑序列$s = \{a_0, a_1, \ldots, a_{n-1}\}$，$n$是2的幂。在下列情况，证明序列$s_1$和$s_2$可以通过执

行双调分裂操作获得，这个操作是9.2.1节中介绍的，对于序列s满足属性 1) s_1和s_2都是双调序列，2) s_1的元素小于s_2的元素。

a）s是双调序列，其中$a_0 \leqslant a_1 \leqslant \ldots \leqslant a_{n/2-1}$，$a_{n/2} \geqslant a_{n/2+1} \geqslant \ldots \geqslant a_{n-1}$。

b）s是双调序列，其中对于某个i $(0 \leqslant i \leqslant n-1)$，$a_0 \leqslant a_1 \leqslant \ldots \leqslant a_i$，$a_{i+1} \geqslant a_{i+2} \geqslant \ldots \geqslant a_{n-1}$。

c）s在移动其元素后，成为递增递减双调序列。

9.4 在双调排序的并行形式中，对n个元素进行排序时假设有n个进程可用。如果只有$n/2$个进程可用，说明如何修改算法。

9.5 证明在双调排序的超立方体形式中，大小为2^k的序列的每个双调合并都是在k维超立方体上执行的，并且各个序列被分配到分离的超立方体上。 419

9.6 证明算法9-1所示的双调排序的并行形式是正确的。特别是，证明算法能正确地比较-交换元素，且元素最终在适当的进程中。

9.7 考虑在格网连接的并行计算机上双调排序算法的并行公式。计算下列形式的确切并行运行时间：

a）如图9-11a所示，在使用存储转发路由的格网上用行优先映射；

b）如图9-11b所示，在使用存储转发路由的格网上用行优先的蛇行映射；

c）如图9-11c所示，在使用存储转发路由的格网上用行优先的混洗映射；

另外，确定使用开通路由时上述运行时间如何改变。

9.8 证明使用比较-分裂操作的基于块的双调排序算法是正确的。

9.9 考虑有n个进程的环形连接并行计算机。证明如何将双调排序网络的输入线映射到环中，使得通信成本最低。分析你的映射的性能。再考虑只能使用p个进程的情况。分析在这种情况下你用的并行形式的性能。要保持一种成本最优的并行形式，最大可用进程数目是多少？你用的方案的等效率函数是什么？

9.10 说明基于块的奇偶转换排序产生的算法是正确的。

提示：这个问题类似于习题9.8。

9.11 说明如何在p个进程的格网连接计算机上应用希尔排序算法的思想（9.3.2节）。不要求算法与超立方体形式完全相同。

9.12 说明如何在p个进程的超立方体上并行化串行希尔排序算法。注意9.3.2节介绍的希尔排序算法并不是串行算法的精确并行化。

9.13 考虑9.3.2节介绍的希尔排序算法。它的性能取决于l的值，l是在执行算法第二阶段时奇数和偶数阶段的数目。描述需要一种$l = \Theta(p)$个阶段的最坏情况的初始主要分布。这种最坏情况出现的概率是多少？

9.14 在9.4.1节中讨论了用于CREW PRAM的快速排序并行形式，它基于分配每个子问题到单独的进程。这种形式使用n个进程对n个元素进行排序。请根据这种方法推导出使用p个进程的并行形式，其中$p < n$。导出并行运行时间、效率以及等效率函数的表达式。你的并行形式可用的且能保持成本最优的最大进程数目是多少？ 420

9.15 推导遍历由算法9-6的算法构造的二叉搜索树的一种算法，并确定每个元素在排序后的数组中的位置。在任意的CRCW PRAM上，你的算法应能用n个进程在时间$\Theta(\log n)$内求解问题。

9.16 考虑快速排序算法（9.4.2节）的PRAM形式。计算由算法产生的二叉树的平均高度。

9.17 考虑下面的并行快速排序算法，它利用p个进程超立方体连接的并行计算机结构。令n是待排序元素的数目，$p = 2^d$是在d维超立方体的进程数目。给每个进程分配n/p个元素的块，并且进程的标号决定待排序序列的全局顺序。算法开始时选择主元元素，再将这个主元对所有进程广播。每个进程在接收到主元以后，把本地元素分成两个块，其中一块的元素小于主元，而另一个块的元素大于主元。然后沿着第d个通信链路连接的进程交换相应的块，使得一个保持其元素小于主元，而另一个保持元素大于主元。特别是，如果进程标号的二进制表示的第d位（最高有效位）是0的话，则保留小于主元的元素；如果第d位是1的话，则保留大于主元的元素。这个步骤以后，在$(d-1)$维超立方体上标号的第d位为0的进程，其元素将小于主元，而在其他$(d-1)$维超立方体上的进程，其元素将大于主元。该过程在每个子立方体上递归地执行，进一步分裂子序列。经过d次这样的分裂——每一次沿着一个维——则序列按照加于进程的全局顺序排序。但这并不意味着每个进程的元素都排好了序。每个进程还需要用串行快速排序对其本地元素进行排序。算法9-9给出这个快速排序的超立方体形式。算法的执行过程如图9-21所示。

分析这个基于超立方体的并行快速排序算法的复杂度。推导并行运行时间、加速比以及效率的表达式。在分析时，假定初始分配到每个进程的元素是均匀分布的。

9.18 考虑习题9.17描述的在d维超立方体上的快速排序并行形式。证明，经过d次分裂后——每一次分裂沿着一个通信链路——元素已经按照进程标号定义的全局顺序排好了序。

421 9.19 考虑习题9.17描述的在d维超立方体上的快速排序并行形式。将此算法与9.4.3节描述的消息传递快速排序算法进行比较。哪个算法的扩展性更好？哪个算法对差的主元元素选择更敏感？

算法9-9 在d维超立方体上快速排序的并行形式。B是分配到每个进程的有n/p个元素的子序列

```
1.  procedure HYPERCUBE_QUICKSORT (B, n)
2.  begin
3.      id := process's label;
4.      for i := 1 to d do
5.      begin
6.          x := pivot;
7.          partition B into B1 and B2 such that B1 < x < B2;
8.          if ith bit is 0 then
9.          begin
10.             send B2 to the process along the ith communication link;
11.             C := subsequence received along the ith communication link;
12.             B := B1 ∪ C;
13.         endif
14.         else
15.             send B1 to the process along the ith communication link;
16.             C := subsequence received along the ith communication link;
17.             B := B2 ∪ C;
18.         endelse
19.     endfor
20.     sort B using sequential quicksort;
21. end HYPERCUBE_QUICKSORT
```

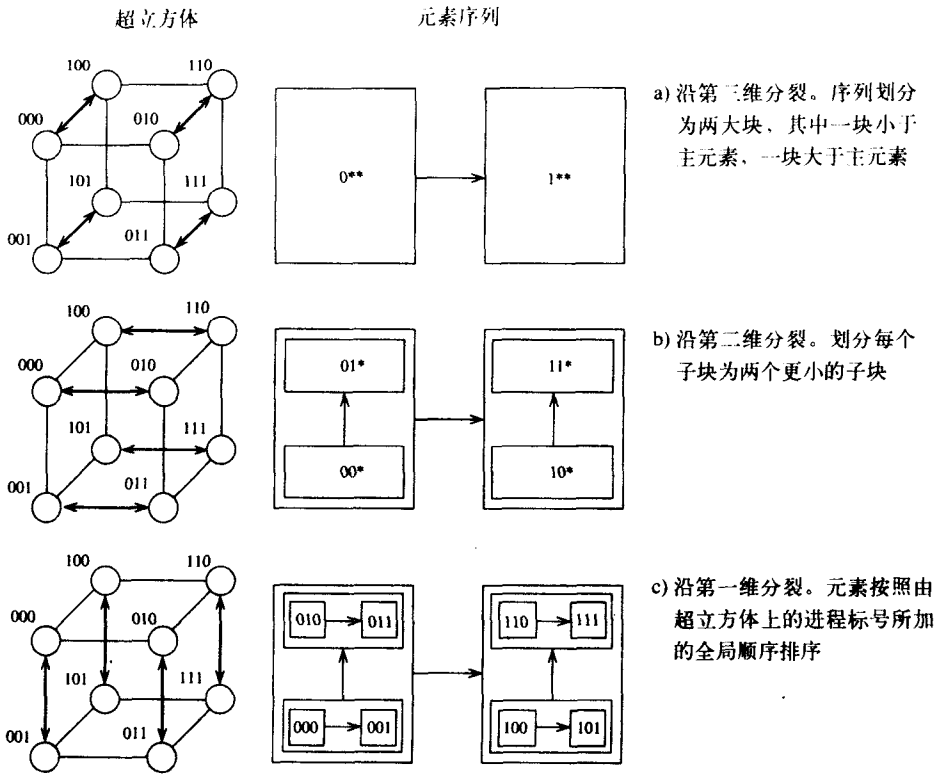


图9-21 $d = 3$的快速排序超立方体形式的执行

注：三次分裂如图a)、b）和c)，每次沿一个通信链路。第二列表示将n个元素的序列划分成子立方体。子立方体之间的箭头指示较大元素的移动。每个方框用子立方体中的进程标号的二进制表示标记。*号表示包含所有二进制组合。

9.20 在d维超立方体上的快速排序并行形式（习题9.17）中，另一种主元选择的方法是一次选择所有2^d-1个主元，如下所示：

a）每个进程随机挑选有l个元素的样本；

b）使用希尔排序算法（9.3.2节），所有进程共同对有$l \times 2^d$个项的样本进行排序；

c）从列表中选择2^d-1个等距主元；

d）广播各个主元，使得所有进程都知道主元。

这个主元选择方案的质量是如何依赖于l的？你认为l应是n的函数吗？在什么假设下，这个方案可以选择出好的主元？当每个进程的元素分布不同时，这个方案可行吗？分析这个方案的复杂度。

9.21 超立方体上并行快速排序主元选择（第9.17节）的另一个方案如下。在沿第i维分裂过程中，有2^{i-1}对进程交换元素。主元选择分两个步骤：第一步，2^{i-1}对进程中的每对进程都计算其合并序列的中值；第二步，计算出2^{i-1}个中值的中值。这个中值的中值就成为沿第i个通信链路分裂的主元。用同样的方法在参与的子立方体中选择后来的主元。在什么假设下，这个方案将产生好的主元选择？它比正文中介绍的中值方案好吗？试分析这种主元选择的复杂度。 422

提示：如果A和B是两个有序序列，每个有n个元素，那么可以在时间$\Theta(\log n)$内找到

$A \cup B$的中值。

423 9.22 在共享地址空间和消息传递体系结构上的快速排序算法的并行形式（9.4.3节）中，每次迭代后都跟着一个障碍同步。此障碍同步对于保证算法的正确性是必需的吗？如果不是，那么没有障碍同步时其性能会如何改变？

9.23 考虑9.4.3节中介绍的快速排序算法的消息传递形式。假设选择了正确的主元，请计算算法准确的（也就是使用t_c、t_w和t_s）并行运行时间和效率。对于

a）$t_c = 1$，$t_w = 1$，$t_s = 1$

b）$t_c = 1$，$t_w = 1$，$t_s = 10$

c）$t_c = 1$，$t_w = 10$，$t_s = 100$

期望得到效率为0.50，0.75和0.95这几种情况，计算你所用形式的等效率函数的不同项。

这种形式的可扩展性是否取决于期望得到的效率值和计算机的体系结构特征？

9.24 考虑下述在采用格网连接的消息传递并行计算机上的快速排序并行形式。假定每个进程分配一个元素。递归划分步骤包括选择主元以及在格网中重排元素，使得那些小于主元的元素放在格网的一部分，而大于主元的元素放在另一部分。假定格网中的进程按以行优先的顺序编号。在快速排序算法结束时，元素按照这种顺序排好序。

考虑图9-22a所示的对任意子序列进行划分的步骤。令k是此序列的长度，P_x、P_{x+1}、…、P_{x+k-1}是存放序列的格网进程。划分包括下列四步：

424 1）随机选择主元并发送给进程P_x。如图9-22b所示，通过使用内嵌树，进程P_x将此主元广播到所有的k个进程。根(P_x)向叶子传送主元。在下面的步骤中也使用到内嵌树。

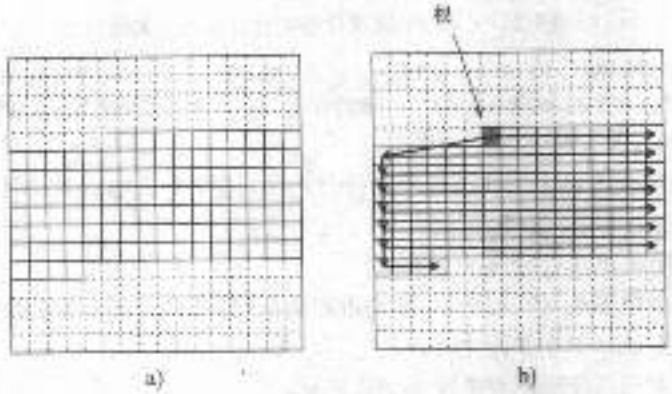


图9-22 a）在执行快速排序时，存放在某一点部分待排序序列的格网任一部分；b）嵌入格网相同部分的二叉树

2）信息收集到每个进程并且在树中往上传递。特别，每个进程计算出其左子树和右子树中小于和大于主元的元素的数目。每个进程知道主元的值，因此可以决定其元素是小于还是大于主元。每个进程将两个值传播到其父进程：一个是进程子树中小于主元的数目，另一个是其中大于主元的数目。由于嵌入格网中的树是不完全的，有些节点没有左子树或右子树。在此步骤结束时，进程P_x知道k个元素中有多少个小于主元，有多少个大于主元。如果s是小于主元的元素的数目，那么在排序后的序列中，主元的位置是P_{x+s}。

3）信息在树中往下传递，使得每个元素移动到小于主元的划分或大于主元的划分中的合

适位置。树中每个进程都从父进程接收小于主元的划分和大于主元的划分中的下一个空位置。根据存放在每个进程中的元素小于主元或大于主元，每个进程向下传播合适的信息到其子树。最初，小于主元的元素的位置是P_0，大于主元的元素的位置是$P_{0,3}$。

4）进程执行置换操作，每个元素移动到小于主元的划分或大于主元的划分中的合适位置。

此算法如图9-23所示。

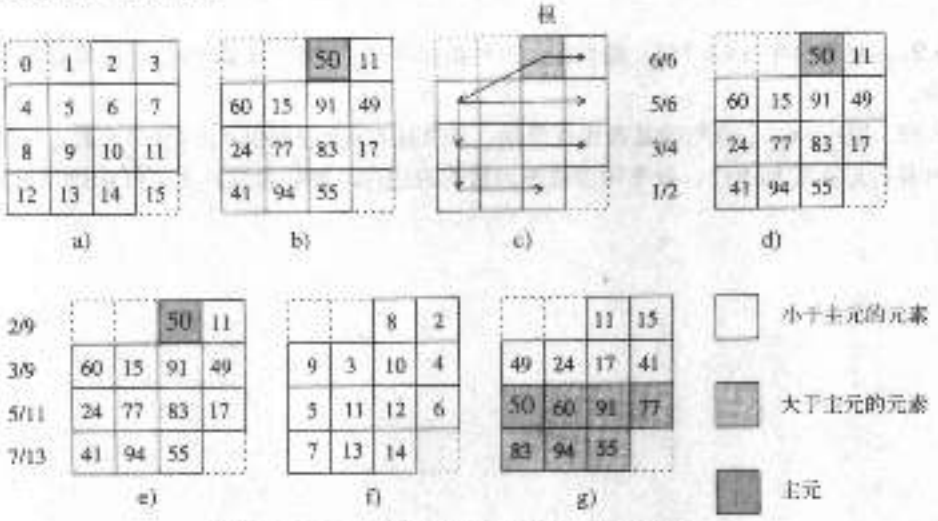


图9-23 在4 × 4格网上划分有13个元素的序列

注：a）格网进程以行优先方式编号；b）存放在每个进程中的元素（带阴影的元素是主元）；c）嵌入一部分格网中的树；d）执行完第二步后，进程第一列中小于或大于主元的元素的数目；e）在第三步，向下传播小于或大于主元的元素到第一列的进程；f）在第三步末尾，元素的目的地；g）在一对一私有通信后元素的位置。

分析这个基于格网的并行快速排序算法的复杂度。推导出并行运行时间、加速比和效率的表达式。进行分析时假设最初分配到各个进程的元素是均匀分布的。

9.25 考虑习题9.24介绍的格网上的快速排序形式。请给出一种使用$p < n$个进程的缩小的形式。分析它的并行运行时间、加速比和等效率函数。

9.26 考虑9.6.1节介绍的枚举排序算法。说明算法如何在下列计算机上实现：

a）CREW PRAM

b）EREW PRAM

c）超立方体连接的并行计算机

d）格网连接的并行计算机

分析你的算法形式的性能。另外，说明如何将这种枚举排序扩展到超立方体上对使用p个进程的n个元素进行排序。

425

9.27 推导9.5节介绍的桶排序并行形式的加速比、效率以及等效率函数。将这些表达式与本章中介绍的其他排序算法的表达式进行比较。哪些并行形式比桶排序的性能好？哪些性能差？

9.28 证明：9.5节讲述的分裂器选择方案能够保证m个桶中每个桶的元素数目少于$2n/m$。

9.29 推导9.5节介绍的样本排序并行形式的加速比、效率和等效率函数。分别在下列条件下推导这些度量：1）每个进程上的p个子块大小相等；2）每个进程上的p个子块的大小相差$\log p$倍。

9.30 在9.5节介绍的样本排序算法中，所有进程发送$p-1$个元素到进程P_0，P_0对$p(p-1)$个元素进行排序，并发送分裂器给所有的进程。修改此算法，使得进程使用双调排序对$p(p-1)$个元素进行并行排序。你如何选择分裂器？计算你的算法形式的并行运行时间、加速比和效率。

426

9.31 基数排序（9.6.2节）的性能是怎样依赖于r的值的？计算r的值，使得算法的运行时间最少。

9.32 推广9.6.2节介绍的基数排序算法，使其用在p个进程($p < n$)对n个元素进行排序。推导这种并行形式的加速比、效率和等效率函数的表达式。你能否设计出更好的排序机制？

427

第10章 图 算 法

在计算机科学中，图论的地位非常重要，因为它为许多问题提供简单而系统化的建模方法。许多问题都可以用图表示，并用标准的图算法解决。本章讨论一些重要和基本的图算法的并行形式。

10.1 定义和表示

无向图（undirected graph）G是一对（V, E），其中V是有限个称为*顶点*的点集，E是有限条*边*的边集。边$e \in E$是无顺序的对(u, v)，这里$u, v \in V$。边(u, v)表示顶点u和顶点v相连。同样，*有向图*（directed graph）G也是一对（V, E），V和上面定义的顶点集合相同，但边$(u, v) \in E$是有序对；即表示有一个从u到v的连接。图10-1所示为一无向图和一有向图。在本章中，无向图和有向图都称为*图*（graph）。

虽然无向图和有向图的某些术语的含义可能稍有区别，但两者的许多定义是相同的。如果图是无向图，那么边(u, v)与顶点u和v关联。然而，如果图是有向图，那么边(u, v)从顶点u*射出*，并*射入*到顶点v。例如，在图10-1a中，边e关联顶点5及顶点4，但是在图10-1b中，边f从顶点5射出并射入到顶点2。如果(u, v)是无向图$G = (V, E)$中的边，就说顶点u和v彼此*邻接*；如果是有向图，就说顶点v*邻接到*顶点u。

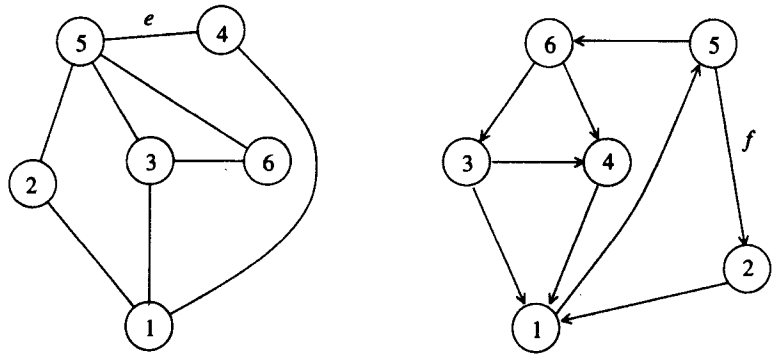


图10-1 a）无向图； b）有向图

从顶点u到顶点v的*路径*是顶点序列$< v_0, v_1, v_2, \ldots, v_k, >$，其中$v_0 = v$，$v_k = u$，且对$i = 0, 1, \ldots, k-1$，$(v_i, v_{i+1}) \in E$。路径的长度被定义为路径中边的数目。如果存在一条从u到v的路径，则u到v是*可达的*(reachable)。如果路径中所有的顶点都是唯一的，则称路径为*简单的*(simple)。 429
如果路径的起点和终点相同，即$v_0 = v_k$，则路径构成一个*圈*(cycle)。一个图没有圈称为*无圈的*(acyclic)，如果所有的中间顶点都不相同，则称圈为*简单的*。例如，在图10-1a中，序列$< 3, 6, 5, 4 >$是从顶点3到顶点4的路径，在图10-1b中，存在一个有向简单圈$< 1, 5, 6, 4, 1 >$。

此外，在图10-1a中，序列< 1, 2, 5, 3, 6, 5, 4, 1 >是无向圈，但不是简单圈，因为它包含环< 5, 3, 6, 5 >。

如果无向图中的每一对顶点通过一条路径连接，则称无向图是*连通的*。对于图$G = (V, E)$，如果存在一个图$G' = (V', E')$，且$V' \subseteq V, E' \subseteq E$，则称$G'$为$G$的*子图*。如果集合$V' \subseteq V$，由$V'$*导出的*$G$的子图为图$G' = (V', E')$，其中，$E' = \{(u, v) \in E | u, v \in V'\}$。如果图中的每一对顶点都邻接，则称图为*完全图*（complete graph）。*树林*（forest）是无圈图，而*树*（tree）是连通无圈图。注意，如果图$G = (V, E)$是树，则$|E| = |V| - 1$。

有时E中的每一条边都有权与之对应。权通常是实数，表示穿越相应的边所需的成本或收益。例如，在电子回路中，电阻器可以用边来表示，边的权表示电阻。如果图中的每条边都有权与之对应，则该图称为*加权图*（weighted graph），用$G = (V, E, w)$表示，其中V和E的定义同前，而$w : E \to \mathcal{R}$为定义在E上的实值函数。图的权定义为边的权之和。路径的权是路径中边的权之和。

在计算机程序中，有两种表示图的标准方法。第一种方法使用矩阵，第二种方法使用链接表。

考虑含n个顶点的图$G = (V, E)$，顶点编号为1, 2, 3, ..., n。图的*邻接矩阵*是一个$n \times n$的数组$A = (a_{i,j})$，定义如下：

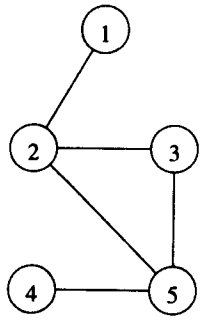
$$a_{i,j} = \begin{cases} 1 & \text{如果 } (v_i, v_j) \in E \\ 0 & \text{否则} \end{cases}$$

430

图10-2为一无向图及其邻接矩阵表示。注意，无向图的邻接矩阵是对称的。为方便表示加权图，可修改邻接矩阵表示法。此时$A = (a_{i,j})$定义如下：

$$a_{i,j} = \begin{cases} w(v_i, v_j) & \text{如果 } (v_i, v_j) \in E \\ 0 & \text{如果 } i = j \\ \infty & \text{否则} \end{cases}$$

我们称这种修改过的邻接矩阵为*加权邻接矩阵*（weighted adjacency matrix）。存储含n个顶点的图的邻接矩阵所需的空间为$\Theta(n^2)$。



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

图10-2 无向图及其邻接矩阵表示

图$G = (V, E)$的*邻接表*（adjacency list）由表的数组$Adj[1 .. |V|]$构成。对于每个$v \in V$，如果图G包含边$(v, u) \in E$，则$Adj[v]$是所有顶点u的链表。换句话说，$Adj[v]$是所有与v邻接的顶点的列表。图10-3为图的邻接表表示的例子。如果对邻接表作修改，在顶点v的邻接表中存储每条边$(u, v) \in E$的权，则可用邻接表来表示加权图。存储这种邻接表所需的空间为$\Theta(|E|)$。

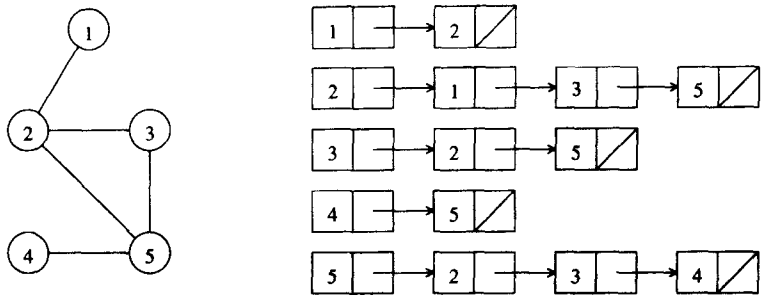


图10-3 无向图及其邻接表表示

图的性质决定图使用哪种表示方法。如果$|E|$远小于$O(|V|^2)$，则图$G = (V, E)$为*稀疏图*；否 431
则为*稠密图*。邻接矩阵用来表示稠密图很用效，而用邻接表表示稀疏图则较有效。注意，如果要遍历图中所有的边，使用邻接矩阵表示时，由于要访问整个数组，串行运行时间的下界为$\Omega(|V|^2)$。然而，如果使用邻接表表示，则基于同样的原因，运行时间的下界为$\Omega(|V|+|E|)$。因此，如果图是稀疏的（$|E|$远小于$|V^2|$），则邻接表表示法要优于邻接矩阵表示法。

本章余下的部分提出了几个图的算法。前4节提出的是稠密图的算法，最后一节讨论稀疏图的算法。我们假设稠密图用邻接矩阵表示，稀疏图用邻接表表示。在全章中，n都表示图中的顶点数目。

10.2 最小生成树：Prim算法

无向图G的*生成树*（spanning tree）是图G的子图，它是一棵包含图G的所有顶点的树。在加权图中，子图的权是子图中边的权值之和。加权无向图的*最小生成树*（minimum spanning tree, MST）是权值最小的生成树。许多问题需要找出无向图的最小生成树。例如，要求出连接网络中一批计算机所需电缆的最小长度，就需要找出包含所有可能连接的无向图的最小生成树。图10-4所示为一无向图及其最小生成树。

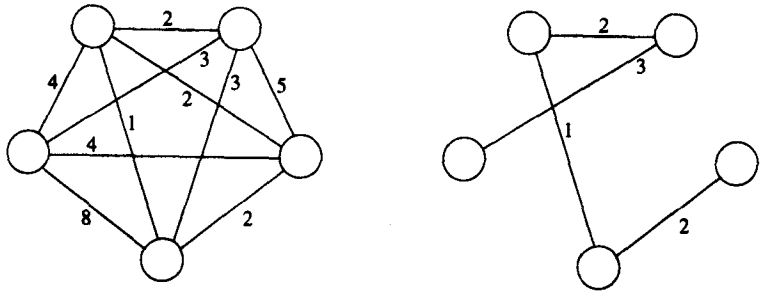


图10-4 无向图及其最小生成树

如果图G是不连通的，那么它不存在生成树。但它有*生成树林*（spanning forest）。为了更简单地描述最小生成树算法，我们假设G是连通的。如果G不连通，则可以找出它的连通分量（10.6节），并对每个连通分量应用MST算法。同样，我们也可以修改MST算法以输出最小生成树林。

寻找最小生成树的Prim算法为贪婪算法。算法首先选择任意的起点，然后不断加入能保 432

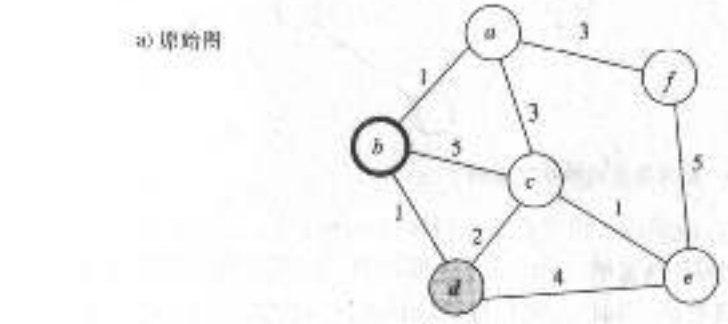
证生成树成本最小的顶点和边，扩大最小生成树。算法一直进行到所有的顶点都被选择才结束。

令$G = (V, E)$为待求最小生成树的加权无向图，$A = (a_{ij})$为加权邻接矩阵。Prim算法如算法10-1所示。算法中用集合V_T来保存最小生成树构造过程中的顶点，并使用数组$d[1..n]$，对于每个顶点$v \in (V-V_T)$，$d[v]$中保存从V_T中的任何顶点到顶点v的边中最小的权值。最初，V_T包含一任意的顶点r，它成为最小生成树的根。而且，$d[r] = 0$，对于所有的$v \in (V-V_T)$，如果边(r, v)存在，则$d[v] = w(r, v)$；否则$d[v] = \infty$。在算法的每次迭代中，如果$d[u] = \min\{d[v] | v \in (V-V_T)\}$，一个新的顶点$u$就加入到$V_T$中。这个顶点加入后，由于在顶点$v$和新加入的顶点$u$之间可能存在一条权更小的边，对于$v \in (V-V_T)$，$d[v]$中的所有值都需要更新。当$V_T = V$时，算法终止。图10-5表示该算法。Prim算法终止时，最小生成树的成本为$\sum_{v \in V} d[v]$。很容易对算法10-1进行修改，使其能存储属于最小生成树的边。

算法10-1 Prim的串行最小生成树算法

```
1.  procedure PRIM_MST(V, E, w, r)
2.  begin
3.      V_T := {r};
4.      d[r] := 0;
5.      for all v ∈ (V − V_T) do
6.          if edge (r, v) exists set d[v] := w(r, v);
7.          else set d[v] := ∞;
8.      while V_T ≠ V do
9.      begin
10.         find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.         V_T := V_T ∪ {u};
12.         for all v ∈ (V − V_T) do
13.             d[v] := min{d[v], w(u, v)};
14.     endwhile
15. end PRIM_MST
```

在算法10-1中，while循环的循环体（第10行到第13行）执行$n-1$次，计算$\min\{d[v] | v \in (V-V_T)\}$(第10行)和for循环（第12行和第13行）都要执行$O(n)$步。因此，Prim算法的算法复杂度为$\Theta(n^2)$。

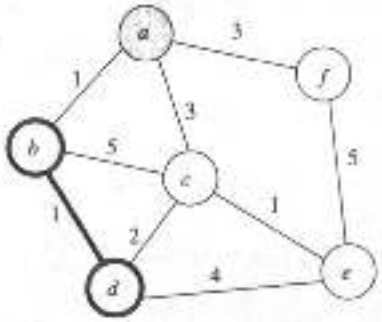


	a	b	c	d	e	f
d[]	1	0	5	1	∞	∞

	a	b	c	d	e	f
a	0	1	3	∞	∞	3
b	1	0	5	1	∞	∞
c	3	5	0	2	1	∞
d	∞	1	2	0	4	∞
e	∞	∞	1	4	0	5
f	2	∞	∞	∞	5	0

图10-5 Prim的最小生成树算法。MST的根为顶点b。在每次迭代中，选择的V_T中的顶点和边用粗线表示。数组$d[v]$表示更新后$V-V_T$中顶点的值

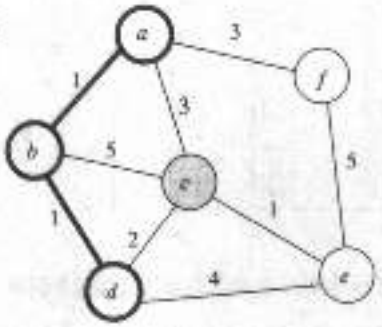
b) 选择出第一条边后



	a	b	c	d	e	f
d[]	1	0	2	1	4	∞

	a	b	c	d	e	f
a	0	1	3	∞	∞	3
b	1	0	5	1	∞	∞
c	3	5	0	2	1	∞
d	∞	1	2	0	4	∞
e	∞	∞	1	4	0	5
f	2	∞	∞	∞	5	0

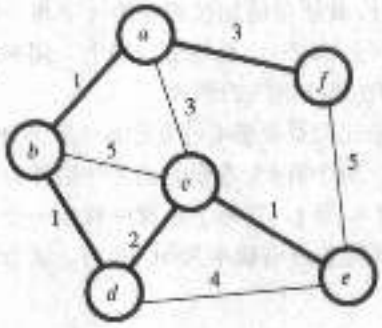
c) 选择出第二条边后



	a	b	c	d	e	f
d[]	1	0	2	1	4	3

	a	b	c	d	e	f
a	0	1	3	∞	∞	3
b	1	0	5	1	∞	∞
c	3	5	0	2	1	∞
d	∞	1	2	0	4	∞
e	∞	∞	1	4	0	5
f	2	∞	∞	∞	5	0

d) 最后的最小生成树



	a	b	c	d	e	f
d[]	1	0	2	1	1	3

	a	b	c	d	e	f
a	0	1	3	∞	∞	3
b	1	0	5	1	∞	∞
c	3	5	0	2	1	∞
d	∞	1	2	0	4	∞
e	∞	∞	1	4	0	5
f	2	∞	∞	∞	5	0

图10-5 （续）

算法的并行形式

Prim算法通过迭代实现。每一次迭代向最小生成树中添加一个新的顶点。由于每次加入新的顶点u到V_T时，顶点v的$d[v]$值可能改变，很难选择多于一个顶点加入到最小生成树中。例如，在图10-5中，选择顶点b后，如果再选择顶点d和顶点c，就不能找到最小生成树。这是因为，在选择顶点d后，$d[c]$的值便从5更新为2。因此，很难以并行方式执行while循环中不同的迭代。然而，每次迭代都可用下面的方法并行化。

433 | 434

令p为进程的数目，n为图中顶点的数目。用1维块映射将集合V划分成p个子集合（3.4.1节）。每个子集合有n/p个连续的顶点。与每个子集合相关的任务被分配给不同的进程。令V_i是分配给进程P_i的顶点的子集，其中，$i = 0, 1, \ldots, p-1$。每个进程P_i存储数组d中与V_i对应的部分（即在$v \in V_i$时，进程P_i存储$d[v]$）。图10-6a表示划分。在while循环的每次迭代中，每个进程P_i计算$d_i[u] = \min\{d_i[v] | v \in (V - V_T) \cap V_i\}$。对所有的$d_i[u]$进行多对一归约操作（4.1节）就得到

全局最小值，并存储在进程P_0中。此时进程P_0中保存新的顶点u，它将被插入到V_T中。进程P_0使用一对多广播操作将u广播给所有的进程。负责顶点u的进程P_i将u标记为属于集合V_T。最后，每个进程对自己的本地顶点更新$d[v]$的值。

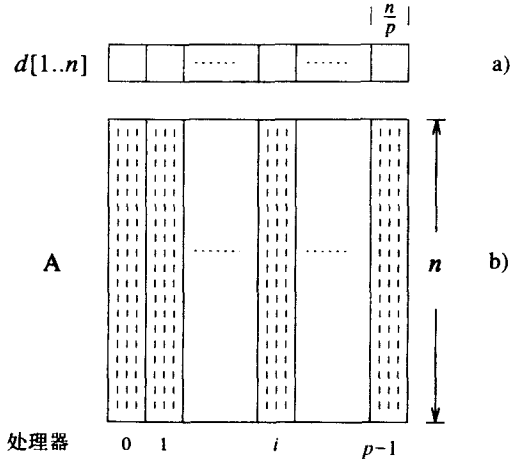


图10-6 在p个进程中划分距离数组d以及邻接矩阵A

当一个新的顶点u插入到V_T后，对于$v \in (V-V_T)$，$d[v]$的值必须更新。负责顶点v的进程必须知道边(u, v)的权。因此，每个进程P_i必须存储加权邻接矩阵中与分配给它的顶点集合V_i对应的列。这对应于矩阵的一维块映射（3.4.1节）。在每个进程中，用来存储邻接矩阵中所需部
435 分的空间为$\Theta(n^2/p)$。图10-6b表示对加权邻接矩阵的划分。

在每次迭代过程中，每个进程最小化以及更新$d[v]$所需的计算为$\Theta(n/p)$。每次迭代的通信操作取决于多对一归约及一对多广播。对于有p个进程的消息传递并行计算机，一个字的一对多广播操作所需时间为$(t_s + t_w)\log p$（4.1节）。在每个进程中找出一个字的全局最小值所需时间和它相同（4.1节）。因此，每次迭代的总通信成本为$\Theta(\log p)$。这个形式的并行运行时间可用下式给出：

$$T_P = \overbrace{\Theta\left(\frac{n^2}{p}\right)}^{\text{计算}} + \overbrace{\Theta(n \log p)}^{\text{通信}}$$

由于串行运行时间为$W = \Theta(n^2)$，并行化得到的加速比和效率如下：

$$\begin{aligned} S &= \frac{\Theta(n^2)}{\Theta(n^2/p) + \Theta(n \log p)} \\ E &= \frac{1}{1 + \Theta((p \log p)/n)} \end{aligned} \tag{10-1}$$

从公式(10-1)可以看出，对于成本最优的并行形式，$(p \log p)/n = O(1)$。因此，Prim算法的这个并行形式可以只使用$p = O(n/\log n)$个进程。而且，从公式(10-1)还可以看出，由通信造成的等效率函数为$\Theta(p^2 \log^2 p)$，由于在这种并行形式中，n增长必须至少和p一样快，由并发带来的等效率函数为$\Theta(p^2)$。所以，这种形式的整体等效率为$\Theta(p^2 \log^2 p)$。

10.3 单源最短路径：Dijkstra算法

对于加权图$G=(V,E,w)$，*单源最短路径*（single-source shortest path）问题是找出从一个顶点$v \in V$到V中所有其他顶点的最短路径。从u到v的最短路径是权最小的路径。根据应用，边权可以表示时间、成本、损失、损耗或者其他任何沿一条路径的相加累积量，而且是最小值。下面将首先讲述Dijkstra算法，它在非负权的无向图和有向图上解决单源最短路径问题。

Dijkstra算法从某一顶点s找出最短路径，这与Prim的最小生成树算法类似。与Prim算法一样，Dijkstra算法渐增地找出从顶点s到G中其他顶点的最短路。它也是贪婪算法；即它总是选择一条看上去与顶点最近的边。算法10-2为Dijkstra算法。将此算法与Prim的最小生成树算法相比，可以发现两者几乎相同。主要区别是，对于每个顶点$u \in (V-V_T)$，Dijkstra算法存储$l[u]$，它是从顶点s经V_T中的顶点到达顶点u的最小成本；而Prim算法存储$d[u]$，它是连接V_T中的一个顶点与u的最小成本边的成本。Dijkstra算法的运行时间为$\Theta(n^2)$。 436

算法10-2 Dijkstra的串行单源最短路径算法

```
1.  procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
2.  begin
3.      V_T := {s};
4.      for all v ∈ (V − V_T) do
5.          if (s, v) exists set l[v] := w(s, v);
6.          else set l[v] := ∞;
7.      while V_T ≠ V do
8.      begin
9.          find a vertex u such that l[u] := min{l[v]|v ∈ (V − V_T)};
10.         V_T := V_T ∪ {u};
11.         for all v ∈ (V − V_T) do
12.             l[v] := min{l[v], l[u] + w(u, v)};
13.     endwhile
14. end DIJKSTRA_SINGLE_SOURCE_SP
```

算法的并行形式

Dijkstra单源最短路径算法的并行形式与Prim最小生成树算法的并行形式非常相似（10.2节）。用1维块映射划分加权邻接矩阵（3.4.1节）。加权邻接矩阵中n/p个连续的列分配给p个进程中的每个进程，进程计算数组l中n/p个值。在每次迭代中，所有进程进行的计算及通信操作与Prim算法的并行形式类似。因此，Dijkstra单源最短路径算法的并行性能和可扩展性与Prim的最小生成树算法相同。

10.4 全部顶点对间的最短路径

除了找到从某一个顶点到其他每个顶点的最短路径外，有时也需要找出全部顶点对间的最短路径。在形式上，对于加权图$G(V,E,w)$，*全部顶点对间的最短路径*（all-pairs shortest path）问题就是找出全部顶点对$v_i, v_j \in V\ (i \neq j)$间的最短路径。对于有$n$个顶点的图，全部顶点对间的最短路径算法的输出为一$n \times n$矩阵$D=(d_{i,j})$，其中$d_{i,j}$是从顶点$v_i$到顶点$v_j$的最短路径的成本。 437

下面讲述两种求解全部顶点对间的最短路径的算法。第一种算法使用Dijkstra的单源最短路径算法，第二种算法使用Floyd算法。Dijkstra算法要求图的边权非负（习题10.4），而Floyd算法使用有负权边的图，只要它们不包含负权的圈。

10.4.1 Dijkstra算法

在10.3节我们讲述了在图中查找从顶点v到其他所有顶点最短路径的Dijkstra算法。这个算法也可用来解决全部顶点对间的最短路径问题，只需让每个进程对每个顶点v执行单源算法。我们将这种算法称为Dijkstra全部顶点对间的最短路径算法。由于单源最短路径的Dijkstra算法的复杂度为$\Theta(n^2)$，全部顶点对间的最短路径算法的复杂度为$\Theta(n^3)$。

算法的并行形式

Dijkstra的全部顶点对间的最短路径算法可用两种不同的方法并行化。一种方法是将顶点在不同的进程间划分，让每个进程计算分配给它的所有顶点间的单源最短路径。我们称这种方法为*源划分形式*（source-partitioned formulation）。另一种方法分配每一个顶点给一组进程，并用单源算法的并行形式（10.3节）求解每组进程上的问题。我们称这种方法为*源并行形式*（source-parallel formulation）。下面将讨论和分析这两种方法。

源划分形式 Dijkstra算法的源划分并行形式使用n个进程。通过执行Dijkstra的串行单源最短路径算法，每个进程P_i找出从顶点v_i到所有其他顶点间的最短路径。这种形式不需要进程间的通信（假设邻接矩阵在所有进程处被复制）。因此，此形式的并行运行时间由下式给出：

$$T_P = \Theta(n^2)$$

由于串行运行时间为$W = \Theta(n^3)$，加速比和效率如下：

$$\begin{aligned} S &= \frac{\Theta(n^3)}{\Theta(n^2)} \\ E &= \Theta(1) \end{aligned} \tag{10-2}$$

由于没有通信开销，看上去这是极佳的并行形式。然而，事实并非如此。算法至多可以用n个进程。因此，由并发带来的等效率函数为$\Theta(n^3)$，这是算法的整体等效率函数。如果可用于解决问题的进程数目较少（即$n = \Theta(p)$），那么算法的性能很好。但是，如果进程的数目
438 大于n，由于该算法的可扩展性较差，其他算法的性能最终将胜过此算法。

源并行形式 源划分并行形式的主要问题是，它只能让n个进程保持忙碌进行有益的工作。如果Dijkstra的单源最短路径算法的并行形式（10.3节）用于对每个顶点v求解问题，则性能可能提高。除了单源算法运行于不相连的进程子集上以外，源并行形式与源划分形式是相似的。

特别地，源并行形式把p个进程划分成n组，每组有p/n个进程（此形式只有当$p > n$时才有意义）。n个单源最短路径问题的每一个都由n个组中的一个求解。也就是说，首先通过分配每个顶点给一组独立的进程对全部顶点对间的最短路径问题并行化，然后再用p/n个进程组对单源算法并行化对问题求解。使用这种形式，得到有效利用的进程总数为$O(n^2)$。

可用10.3节中的分析推导出Dijkstra全部顶点对间的最短路径算法并行形式的性能。假设有一个含p个进程的消息传递计算机，p是n的整数倍。p个进程被划分成大小为n/p的组。如果单源算法在每p/n个进程组中执行，则并行运行时间为

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{计算}} + \overbrace{\Theta(n \log p)}^{\text{通信}} \tag{10-3}$$

注意公式(10-3)与公式(10-2)之间的相似性。这些相似性并不奇怪，因为每组p/n个进程构成不同的组，并独立地进行计算。所以，解决单源问题的每组p/n个进程所需的时间决定整体运行时间。由于串行运行时间为$W = \Theta(n^3)$，加速比和效率如下:

$$\begin{aligned} S &= \frac{\Theta(n^3)}{\Theta(n^3/p) + \Theta(n \log p)} \\ E &= \frac{1}{1 + \Theta((p \log p)/n^2)} \end{aligned} \tag{10-4}$$

从公式(10-4)可以看出，对于成本最优的并行形式，$(p \log p)/n^2 = O(1)$。因此，这种形式最多可有效使用$O(n^2/\log n)$个进程。公式(10-4)还表明，由通信带来的等效率函数为$\Theta((p \log p)^{1.5}))$，由并发带来的等效率函数为$\Theta(p^{1.5})$。因此，整体等效率函数为$\Theta((p \log p)^{1.5}))$。

比较Dijkstra全部顶点对间的最短路径算法的两种并行形式可以发现，源划分形式不进行通信，只能使用不超过n个进程，求解问题所需的时间为$\Theta(n^2)$。与之相反，源并行形式可使用最多$n^2/\log n$个进程，有某些通信开销，当使用$n^2/\log n$个进程时，求解问题所需的时间为$\Theta(n \log n)$。因此，源并行形式比源划分形式利用更多的并行性。 439

10.4.2 Floyd算法

解决全部顶点对间的最短路径的Floyd算法基于下面的考虑。令$G = (V, E, w)$为加权图，$V = \{v_1, v_2, \ldots, v_n\}$是图$G$的顶点。考虑$k$个顶点的子集$\{v_1, v_2, \ldots, v_k\}$，其中$k \leqslant n$。对于任一对顶点$v_i, v_j \in V$，考虑从$v_i$到$v_j$的所有路径，其中间顶点属于集合$\{v_1, v_2, \ldots, v_k\}$。令$p_{i,j}^{(k)}$为这些路径中权最小的路径，再令$d_{i,j}^{(k)}$为$p_{i,j}^{(k)}$的权。如果顶点$v_k$不是从$v_i$到$v_j$的最短路径，则$p_{i,j}^{(k)}$与$p_{i,j}^{(k-1)}$相同。但是，如果$v_k$在$p_{i,j}^{(k)}$中，则可将$p_{i,j}^{(k)}$分成两条路径——一条从$v_i$到$v_k$，另一条从$v_k$到$v_j$。这两条路径都使用集合$\{v_1, v_2, \ldots, v_{k-1}\}$中的顶点。因此，$d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$。以上的结论可用下面的递归公式表示:

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & , k = 0 \\ \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\} & , k \geqslant 1 \end{cases} \tag{10-5}$$

从v_i到v_j的最短路径的长度由$d_{i,j}^{(n)}$给出。通常解为矩阵$D(n) = (d_{i,j}^{(n)})$。

Floyd算法按k值递增的顺序自底向上求公式(10-5)的解。算法10-3为Floyd的全部顶点对算法。Floyd算法的运行时间由第4至7行的三重嵌套for循环决定。每次执行第7行所需时间为$\Theta(1)$；因此，算法的复杂度为$\Theta(n^3)$。算法10-3似乎隐含必须存储n个大小为$n \times n$的矩阵。但是，计算矩阵$D^{(k)}$时，只需要矩阵$D^{(k-1)}$。因此，最多只需存储两个$n \times n$的矩阵。所以，整体空间复杂度为$\Theta(n^2)$。而且，即使只使用D的一个副本，该算法也能正确执行（习题10.6）。

算法10-3 Floyd的全部顶点对间的最短路径算法

```
1.  procedure FLOYD_ALL_PAIRS_SP(A)
2.  begin
3.      D^(0) = A;
4.      for k := 1 to n do
5.          for i := 1 to n do
6.              for j := 1 to n do
7.                  d_{i,j}^(k) := min(d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1));
8.  end FLOYD_ALL_PAIRS_SP
```

440

注：程序用邻接矩阵A计算图$G=(V,E)$的全部顶点对间的最短路径。

算法的并行形式

Floyd算法的一般并行形式是，对于每一个k值，把计算矩阵$D^{(k)}$的任务分配给一组进程。令p为可用的进程数目。矩阵$D^{(k)}$被划分成p部分，每部分分配给一个进程。每个进程计算它自己部分的$D^{(k)}$值。为了做到这一点，进程必须访问矩阵$D^{(k-1)}$的第k行和k列的相应部分。下面讨论一种划分矩阵$D^{(k)}$的技巧。另一种技巧在习题10.8中考虑。

2维块映射 可用2维块映射划分矩阵$D^{(k)}$（3.4.1节）。特别，矩阵$D^{(k)}$分为大小为$(n/\sqrt{p})\times(n/\sqrt{p})$的$p$个块，每块分配给$p$个进程之一。把$p$个进程想象成排列在大小为$\sqrt{p}\times\sqrt{p}$的逻辑网格中是有帮助的。注意这仅仅是概念上的排列，并不一定反映真正的互连网络的连接方式。我们将第i行第j列上的进程称为$P_{i,j}$。$D^{(k)}$的一个子块分配给进程$P_{i,j}$，子块的左上角为$((i-1)n/\sqrt{p}+1,(j-1)n/\sqrt{p}+1)$，右下角为$(in/\sqrt{p},jn/\sqrt{p})$。在每次迭代中，每个进程更新矩阵中自己的部分。图10-7a表示2维块映射技巧。

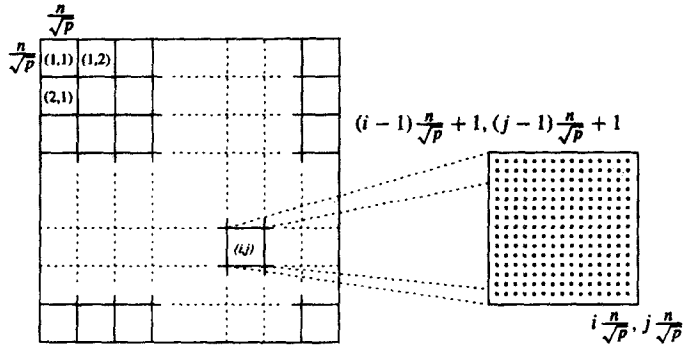


图10-7 a）用2维块映射将矩阵$D^{(k)}$分成$\sqrt{p}\times\sqrt{p}$个子块； b）$D^{(k)}$的子块分配给进程$P_{i,j}$

在算法的第k次迭代过程中，每个进程$P_{i,j}$都需要矩阵$D^{(k-1)}$中第k行和第k列中的某些部分。例如，计算$d_{i,j}^{(k)}$需要得到$d_{i,k}^{(k-1)}$和$d_{k,j}^{(k-1)}$。如图10-8所示，$d_{i,k}^{(k-1)}$驻留在与$P_{i,j}$同一行的进程中，而$d_{k,j}^{(k-1)}$保留在与$P_{i,j}$同一列的进程中。这些值的传输方式如下：在算法的第k次迭代过程，$\sqrt{p}$个进程中包含第k行一部分的每个进程都将这一部分发送给同一列的$\sqrt{p}-1$个进程。同样，$\sqrt{p}$个

441 进程中包含第k列一部分的每个进程都将这一部分发送给同一行的$\sqrt{p}-1$个进程。

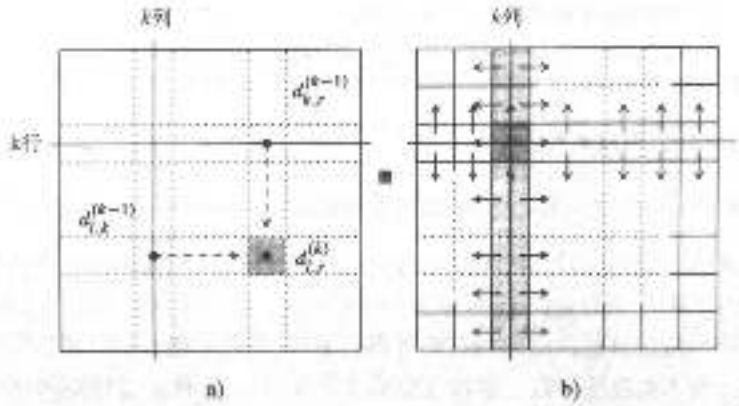


图10-8 a）使用2维块映射的通信模式。计算$d_{i,j}^{(k)}$时，信息必须从同一行和同一列的其他两个进程发送到高亮的进程；b）包含第k行和k列的$\sqrt{p}$个进程的行和列沿进程的列和行发送信息

算法10-4显示使用2维块映射的Floyd算法的并行形式。我们在含p个进程的消息传递计算机上分析这个算法的性能，计算机的交叉对分带宽为$\Theta(p)$。在算法的每次迭代中，进程的第k行和第k列沿$\sqrt{p}$个进程的一行或一列进行一对多广播操作。每一个这样的进程都有第k行或第k列的$n/\sqrt{p}$个元素，所以发送的元素数目为$n/\sqrt{p}$，广播需要的时间为$\Theta((n \log p)/\sqrt{p})$。第7行的同步操作需要的时间为$\Theta(\log p)$。由于矩阵$D^{(k)}$的$n^2/p$个元素分配给每个进程，计算相应的$D^{(k)}$值的时间为$\Theta(n^2/p)$。因此，使用2维块映射的Floyd算法的并行运行时间为

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{计算}} + \overbrace{\Theta\left(\frac{n^2}{\sqrt{p}}\log p\right)}^{\text{通信}}$$

由于串行运行时间为$W = \Theta(n^3)$，加速比和效率如下：

$$\begin{aligned} S &= \frac{\Theta(n^3)}{\Theta(n^3/p) + \Theta((n^2 \log p)/\sqrt{p})} \\ E &= \frac{1}{1 + \Theta((\sqrt{p}\log p)/n)} \end{aligned} \tag{10-6}$$

从公式(10-6)中可以看出，对于成本最优的并行形式，$(\sqrt{p}\log p)/n = O(1)$；因此，2维块映射可以有效地使用最多$O(n^2/\log^2 n)$个进程。用公式(10-6)推导出由通信带来的等效率函数为$\Theta(p^{1.5}\log^3 p)$。由并发带来的等效率函数为$\Theta(p^{1.5})$。因此，整体等效率函数为$\Theta(p^{1.5}\log^3 p)$。 442

算法10-4 使用2维块映射的Floyd算法并行形式

```
1.  procedure FLOYD_2DBLOCK(D^(0))
2.  begin
3.      for k := 1 to n do
4.      begin
5.          each process P_{i,j} that has a segment of the k^th row of D^(k-1);
                broadcasts it to the P_{*,j} processes;
6.          each process P_{i,j} that has a segment of the k^th column of D^(k-1);
```

```
            broadcasts it to the P_{i,*} processes;
7.          each process waits to receive the needed segments;
8.          each process P_{i,j} computes its part of the D^(k) matrix;
9.       end
10.   end FLOYD_2DBLOCK
```

注：$P_{*,j}$表示第j列的所有进程，$P_{i,*}$表示第i行的所有进程。矩阵$D^{(0)}$是邻接矩阵。

加速方法 在Floyd算法的2维块映射形式中，计算矩阵$D^{(k)}$的元素前，一个同步步骤保证所有的进程得到矩阵$D^{(k-1)}$中的对应部分(算法10-4第7行)。换句话说，只有当第$(k-1)$次迭代完成后且矩阵$D^{(k-1)}$的相应部分已传送给所有的进程，第k次迭代才能开始。消除同步步骤不会影响算法的正确性。为了实现这一点，进程在完成了第$(k-1)$次迭代并得到矩阵$D^{(k-1)}$的相应部分后开始第k次迭代。这种形式称为**流水线2维块映射**（pipelined 2-D block mapping）。在8.3节用同样的技巧来改善高斯消元法的性能。

考虑一个按2维拓扑结构排列的p个进程的系统。假设进程$P_{i,j}$在完成了第$(k-1)$次迭代并得到了矩阵$D^{(k-1)}$的相应部分后开始第k次迭代。当进程$P_{i,j}$在得到第k行的元素并完成第$(k-1)$次迭代后，把本地存储的矩阵$D^{(k-1)}$的相应部分发送给进程$P_{i,j-1}$和$P_{i,j+1}$。这样做的原因是，矩阵$D^{(k-1)}$的那一部分被用于计算矩阵$D^{(k)}$。同样，当进程$P_{i,j}$在得到第k列的元素并完成第$(k-1)$次迭代后，把本地存储的矩阵$D^{(k-1)}$的相应部分发送给进程$P_{i-1,j}$和$P_{i+1,j}$。当进程$P_{i,j}$从沿逻辑格网同一行的进程中接收矩阵$D^{(k)}$的元素时，将元素存储在本地，并把它们转发给与接收方向相反的进程。在列上也使用这种通信协议。当到达格网结构的边界时，不再转发矩阵$D^{(k)}$的元素。

443 图10-9说明一行（或列）的进程的通信及终止协议。

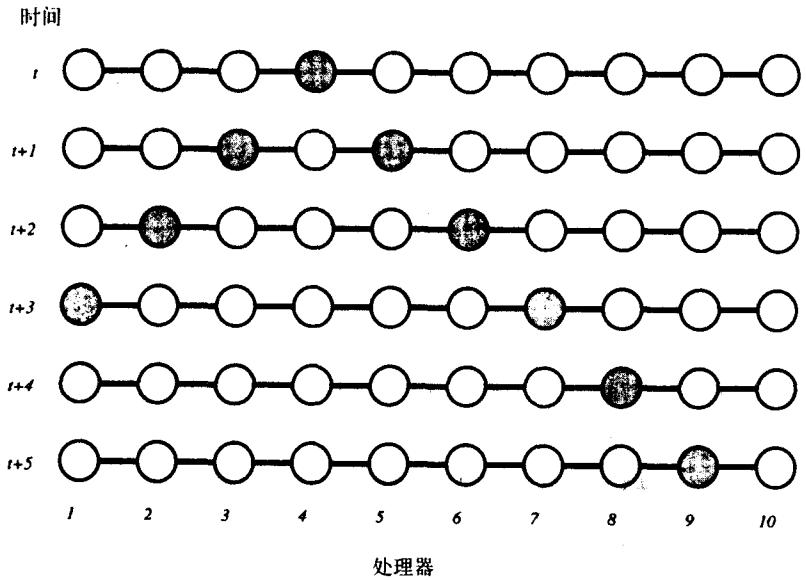


图10-9 使用流水线2维映射的Floyd算法并行形式的通信协议

注：假设进程4在时刻t刚计算出矩阵$D^{(k-1)}$的第k列的一部分。它将此部分发送给进程3和进程5。在$t+1$时刻，这两个进程收到这个部分（时间单位为矩阵部分在邻接的进程间的通信链路上传送所需时间）。同样，远离进程4的进程收到该部分的时间更迟。进程1（在边界上）收到该部分后并不转发。

考虑第一次迭代中元素的移动。在每一步，第一行的 $n/\sqrt{p}$ 个元素从进程$P_{i,j}$发送到进程$P_{i+1,j}$。同样，第一列的元素从进程$P_{i,j}$发送到进程$P_{i,j+1}$。每一步耗时$\Theta(n/\sqrt{p})$。在$\Theta(\sqrt{p})$步后，进程 $P_{\sqrt{p},\sqrt{p}}$ 得到第一行和第一列的相应元素，时间为$\Theta(n)$。在流水线模式下，后继行和列的元素耗时$\Theta(n^2/p)$。因此，进程 $P_{\sqrt{p},\sqrt{p}}$ 完成最短路计算中自己部分的时间为$\Theta(n^3/p)+\Theta(n)$。当进程 $P_{\sqrt{p},\sqrt{p}}$ 完成第$(n-1)$次迭代后，就将第n行和第n列的相应值发送给其他的进程。这些值在$\Theta(n)$ 时间后到达进程$P_{1,1}$。这种形式的并行运行时间为

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{计算}} + \overbrace{\Theta(n)}^{\text{通信}}$$

由于串行运行时间为$W=\Theta(n^3)$，加速比和效率如下：

$$S = \frac{\Theta(n^3)}{\Theta(n^3/p)+\Theta(n)}$$

$$E = \frac{1}{1+\Theta(p/n^2)} \qquad (10\text{-}7)$$

444

从公式(10-7)可以看出，对于成本最优的并行形式，$p/n^2 = O(1)$。因此，Floyd算法的流水线形式最多能有效使用$O(n^2)$ 个进程。同样，用公式(10-7)也能推导由通信带来的等效率函数为$\Theta(p^{1.5})$。这也是整体等效率函数。比较流水线形式与同步的2维块映射形式可知，前者要快得多。

10.4.3 性能比较

对分带宽$O(p)$的并行体系结构上，前面讲过的全部顶点对间的最短路径算法的性能汇总在表10-1中。Floyd流水线形式的可扩展性最强，并能最多使用$\Theta(n^2)$ 个进程在$\Theta(n)$时间内求解问题。而且，即使在像格网连接计算机这样的对分带宽为 $O(\sqrt{p})$ 的体系结构上，这种并行形式也有同样好的执行性能。此外，它的性能独立于路由选择类型（存储转发路由选择或直通路由选择）。

表10-1 对分带宽为O(p)的不同体系结构上，全部顶点对间的最短路径算法的性能及可扩展性，如果进程能正确映射到底层处理器，则k-d立方体系结构的运行时间也相同

	$E=\Theta(1)$时的最大进程数	对应的并行运行时间	等效率函数
Dijkstra源划分	$\Theta(n)$	$\Theta(n^2)$	$\Theta(p^3)$
Dijkstra源并行	$\Theta(n^2/\log n)$	$\Theta(n\log n)$	$\Theta((p\log p)^{1.5})$
Floyd 一维块映射	$\Theta(n/\log n)$	$\Theta(n^2\log n)$	$\Theta((p\log p)^3)$
Floyd 二维块映射	$\Theta(n^2/\log^2 n)$	$\Theta(n\log^2 n)$	$\Theta(p^{1.5}\log^3 p)$
Floyd流水线二维块映射	$\Theta(n^2)$	$\Theta(n)$	$\Theta(p^{1.5})$

10.5 传递闭包

在许多应用中，我们需要确定图中的任意两个顶点是否连通。通常，这是通过找出图的

传递闭包实现的。从形式上说，如果$G = (V, E)$是一个图，G的传递闭包（transitive closure）定义为图$G^* = (V, E^*)$，其中$E^* = \{(v_i, v_j) | G$中存在从v_i到v_j的路径$\}$。我们通过计算连通性矩
445 阵A^*来计算图的传递闭包。G的连通性矩阵（connectivity matrix）为这样一个矩阵$A^* = (a^*_{i,j})$，如果存在从v_i到v_j的路径或$i = j$，则$a^*_{i,j} = 1$，否则$a^*_{i,j} = \infty$。

为了计算A^*，我们对E的每条边赋权值1，并对这个加权图使用任一全部顶点对间的最短路径算法。矩阵A^*可从矩阵D得出，其中D是全部顶点对间的最短路径的解，如下式：

$$a^*_{i,j} = \begin{cases} \infty & , \ d_{i,j} = \infty \\ 1 & , \ d_{i,j} > 0 \text{ or } i = j \end{cases}$$

计算A^*的另一种方法是在图G的邻接矩阵上用Floyd算法，在算法10-3中分别用逻辑or和逻辑and操作替换第7行的min和+操作。此时，我们首先在$i = j$或$((v_i, v_j) \in E$时设置$a_{i,j} = 1$，在其他情况下设置$a_{i,j} = 0$。在$d_{i,j} = 0$时设置$a^*_{i,j} = \infty$，在$d_{i,j}$不为0时设置$a^*_{i,j} = 1$，就可得出A^*。计算传递闭包的复杂度为$\Theta(n^3)$。

10.6 连通分量

无向图$G = (V, E)$的连通分量（connected component）是这样的最大不相连集合$C_1, C_2, \ldots, C_k$，$V = C_1 \cup C_2 \cup \ldots \cup C_k$，且$u, v \in C_i$当且仅当$u$从$v$可达和$v$从$u$可达。无向图的连通分量是在“从…可达”关系下的顶点的等价类。例如，图10-10是有3个连通分量的无向图。

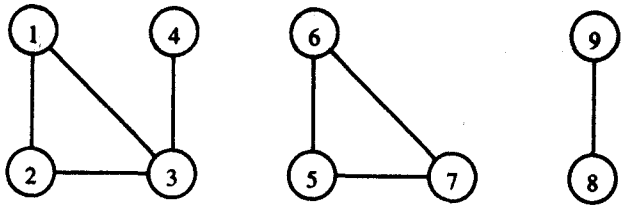
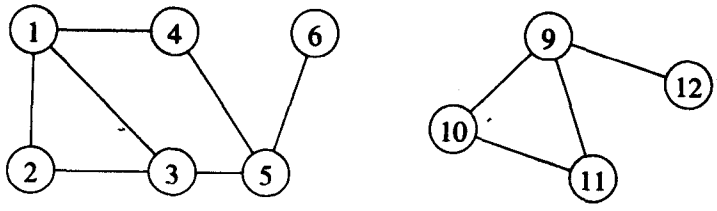


图10-10 有3个连通分量{1, 2, 3, 4}、{5, 6, 7}和{8, 9}的图

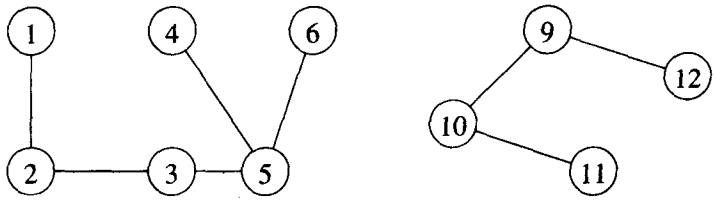
深度优先搜索算法

对图进行深度优先遍历可以找出图的连通分量。深度优先遍历的结果是深度优先树的树林，树林中每一棵树都包含属于不同连通分量的顶点。图10-11展示这种算法，算法的正确性直接从生成树的定义（即深度优先树也是由深度优先树中一组顶点导出的图的生成树）以及G是无向图这一事实推出。假设图用稀疏表示法存储，由于深度优先遍历算法要遍历图G中的所
446 有边，算法的运行时间为$\Theta(|E|)$。



a)

图10-11 图b）是对图a）进行深度优先遍历得到的深度优先树林。这些树的每一棵都是图a）的连通分量



b)

图10-11 （续）

算法的并行形式

连通分量算法的并行化步骤是：将图G的邻接矩阵划分成p个部分，并把每个部分分配给p个进程之一。每个进程P_i拥有图G的一个子图G_i，其中$G_i = (V, E_i)$，E_i是与分配给进程的邻接矩阵的部分对应的边。在这种并行形式的第一步，每个进程P_i计算图G_i的深度优先生成树林。在此步结束时就构造了p个生成树林。在第二步，生成树林按对合并，直到只剩下一个生成树林。这剩下的生成树林有一个性质：如果两个顶点位于同一棵树上，则它们是在图G的同一连通分量中。图10-12展示这个算法。

为了有效地合并成对的生成树林，算法只使用不相连的边集。假设图G的子图的生成树林中的每一棵树都用一个集合表示。不同树的集合成对地不相连。下面是定义在不相连集合上的操作：

find(x)返回一个指针，指向包含x的集合中的代表元素。每个集合都有其唯一的代表。

union(x, y)合并包含元素x和y的集合。假定这两个集合在操作前不相连。 447

生成树林的合并方法如下：令A和B为两个待合并的生成树林，一个树林中最多$n-1$条边（因为A和B是树林）与另一个树林中的边合并。假设要把树林A合并到树林B。对A的每一条边(u, v)，对每一个顶点执行一次find操作确定是否两个顶点已处于B中同一棵树上。如果不是，则B中包含u和v的两棵树（集合）通过union操作合并，否则不需要进行union操作。因此，合并A和B最多需要$2(n-1)$次find操作和$(n-1)$次union操作。我们可使用分等级及路径压缩的不相连集合树林来实现这种不相连集合数据结构。用这种实现方法，进行$2(n-1)$次find操作和$(n-1)$次union操作的成本为$O(n)$。关于不相连集合树林的详细讨论它超出本书的范围，读者可以查阅书目评注的参考文献（10.8节）。

讨论了如何有效地合并两个生成树林后，现在集中到如何划分图G的邻接矩阵，并在p个进程中分配它。下面的部分讨论使用1维块映射的划分形式。另一种划分方案在习题10.12中讨论。

1维块映射 $n \times n$的邻接矩阵被分成p个条带（3.4.1节）。每个条带由n/p个连续的行构成，并分配给p个进程之一。为了计算连通分量，每个进程首先为n个顶点的图计算生成树林，用分配给它的邻接矩阵的n/p个行表示。

考虑含p个进程的消息传递计算机。计算基于分配给每个进程的$(n/p) \times n$邻接矩阵的生成树林，需要时间$\Theta(n^2/p)$。算法的第二步——逐对合并生成树林——通过向进程嵌入一个虚拟树来实现。有$\log p$个合并阶段，每个阶段耗时$\Theta(n)$。因此，合并花费的成本为$\Theta(n \log p)$。最后，在每个合并阶段，生成树林在最近的邻居间传送。回忆生成树林要传送$\Theta(n)$条边。因此，通信成本为$\Theta(n \log p)$。连通分量算法的并行运行时间为

$$T_P = \overbrace{\Theta\left(\frac{n^2}{p}\right)}^{\text{局部计算}} + \overbrace{\Theta(n \log p)}^{\text{树林合并}}$$

由于串行复杂度为$W = \Theta(n^2)$，加速比和效率如下：

$$S = \frac{\Theta(n^2)}{\Theta(n^2/p) + \Theta(n \log p)}$$
$$E = \frac{1}{1 + \Theta((p \log p)/n)} \quad (10\text{-}8)$$

从公式(10-8)可以看出，对于成本最优的并行形式，$p = O(n/\log n)$。从公式(10-8)还可推导出等效率函数为$\Theta(p^2 \log^2 p)$。这是由通信以及在合并阶段额外的计算带来的等效率函数。由于并发产生的等效率函数为$\Theta(p^2)$；所以，整体等效率函数为$\Theta(p^2 \log^2 p)$。在消息传递计算机上，这个并行形式的性能与Prim的最小生成树算法以及Dijkstra的全部顶点对间的最短路径算法的性能相似。

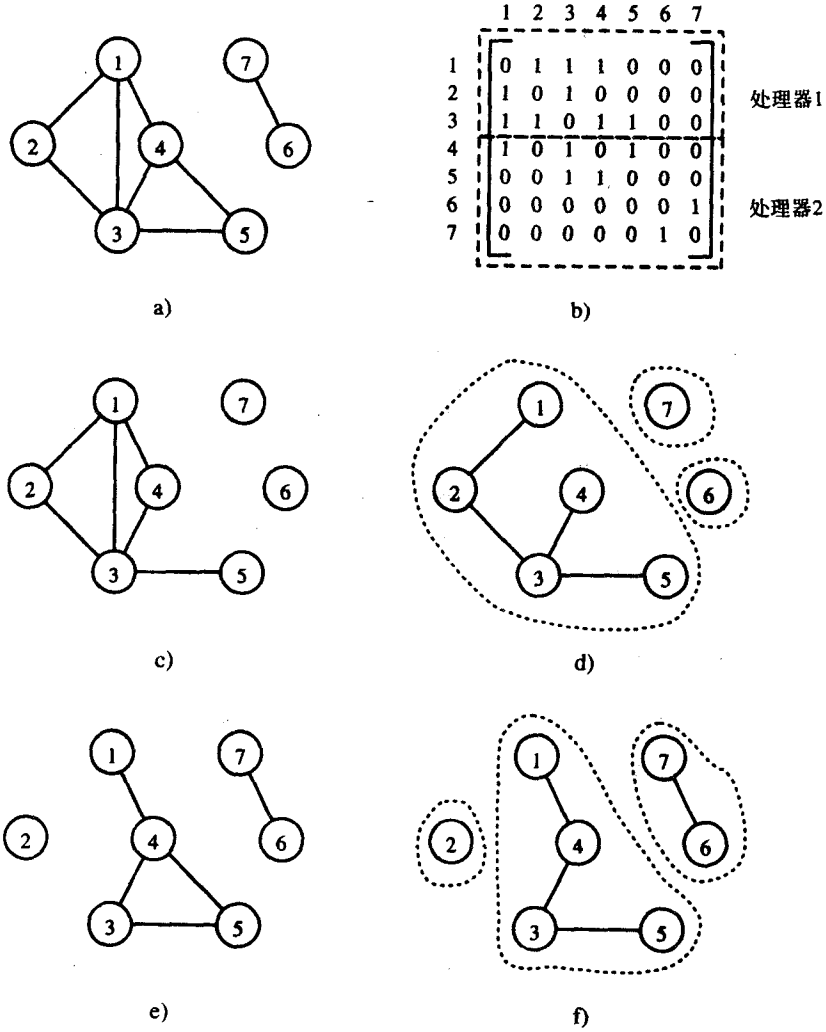


图10-12 并行计算连通分量

注：a）中图G的邻接矩阵被分成如图 b）所示的两个部分。接下来，每个进程得到如图 c）和图 e）所示的G的子图。然后每个进程计算子图的生成树林，如图 d）和图 f）所示。最后，两棵生成树合并得到最终结果。

10.7 稀疏图算法

以前各节讲到的并行算法都是稠密图问题的知名算法。但是，我们尚未涉及稀疏图的并行算法。前面曾提到，如果$|E|$远小于$|V|^2$，则图$G = (V, E)$为稀疏图。图10-13表示几种稀疏图。

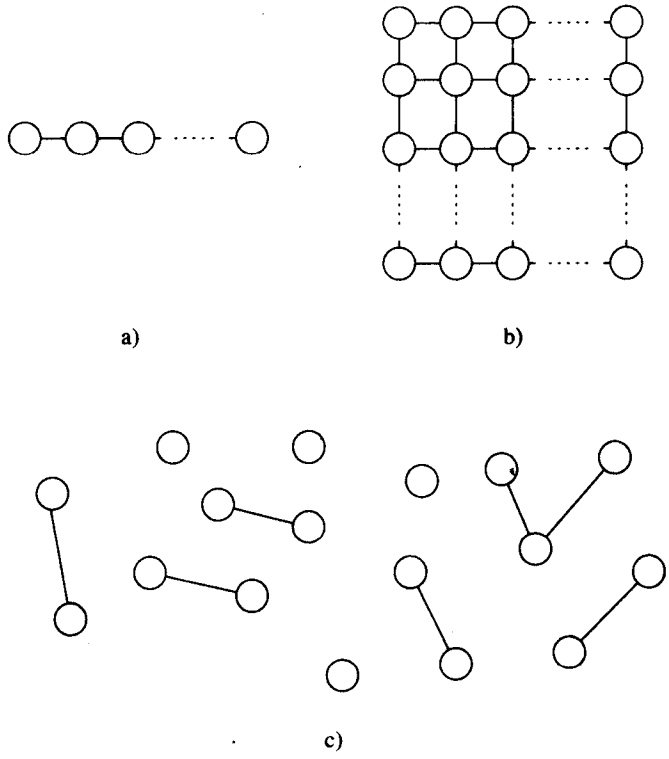


图10-13 稀疏图实例：a）每个顶点有两条关联边的线性图；
b）每个顶点有4条关联边的网格图；c）随机稀疏图

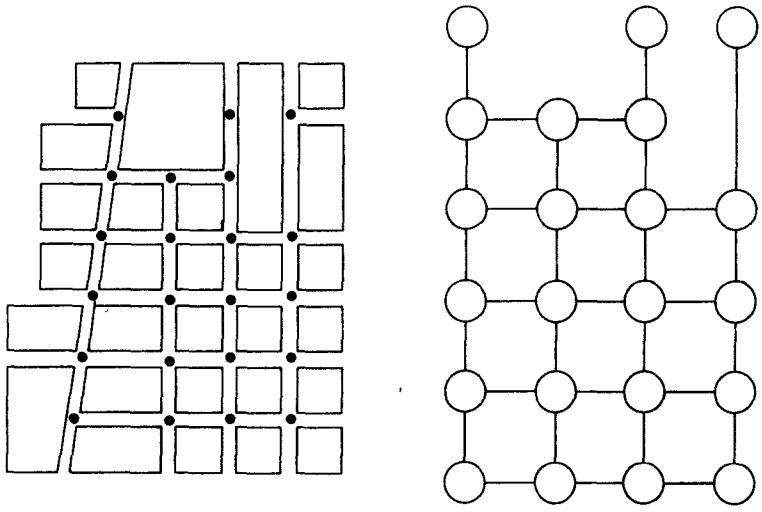
所有稠密图算法都能在稀疏图上正确运行。但是，如果进一步考虑图的稀疏度，则通常可以显著地提升性能。例如，不考虑图的边数时，Prim的最小生成树算法的运行时间为$\Theta(n^2)$。如果修改Prim算法，使用邻接表及一个二元堆，则算法的复杂度下降为$\Theta(|E|\log n)$。所以只要$|E| = O(n^2/\log n)$，修改后的算法就会好于原算法。提出稀疏图算法很重要的一个步骤是使用邻接表而不是邻接矩阵。这种表示上的改变是关键性的，因为基于邻接矩阵的算法的复杂度通常是$\Omega(n^2)$，并且与边数无关。相反，基于邻接表的算法的复杂度通常是$\Omega(n+|E|)$，它依赖于图的稀疏度。

448 ~ 450

在稠密图串行算法的并行形式中，通过划分图的邻接矩阵，使得每个进程能分配到大致相同的任务，并且通信是本地化的，从而获得良好的性能。我们能达到这种结果，主要由于图是稠密的。例如，考虑Floyd全部顶点对间的最短路径算法。通过从邻接矩阵中分配同样大小的块给每个进程，使任务被均匀分配。而且，由于每个块由连续的行和列组成，通信开销受到限制。

然而，对于稀疏图很难实现均匀分配任务及低通信开销。考虑划分图的邻接表的问题。一种可能的划分是分配同等数目的顶点和它们的邻接表给每个进程。但是，与给定顶点关联

的边数可能不同。因此，有些进程可能会被分配大量的边，而另一些进程分到很少，导致进程间明显的任务不平衡。同样，我们也可以分配同等数目的边给每个进程。这就可能需要在多个进程间分割一个顶点的邻接表。结果，存储邻接表不同部分的进程间的通信时间可能显著增加。因此，对于常见的稀疏图，很难导出有效的并行形式（习题10.4和10.5）。但是，如果稀疏图具有一定的结构，则通常能得出有效的并行形式。例如，考虑如图10-14所示的街道地图。与地图对应的图是稀疏的：与任意顶点关联的边数最多为4。这种图称为网格图（grid graph）。我们也可以从其他类型的稀疏图中推导有效的并行形式，如与形状规则的有限元格网对应的图以及顶点度数相近的图等。对于这些类型的图，下面两小节将给出有效的算法，用于查找顶点的最大独立集以及计算单源最短路径。



a) b)

图10-14 街道地图

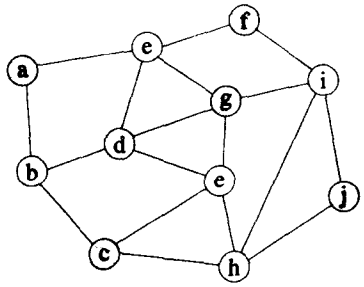
注：街道地图a）可用图b）来表示。在图b）中，每个街道的交点是一个顶点，每条边是一段街道。图b）中的顶点是图a）中用黑点标出的交点。

10.7.1 查找最大独立集

下面考虑查找图顶点的最大独立集（Maximal Independent Set，MIS）问题。给定稀疏的无向图$G = (V, E)$。如果I中任何一对顶点都不能由G中的一条边相连，则称顶点集合$I \subset V$为独立集（independent set）。如果在I中加入不属于I的任何其他顶点集合就不再独立，则独立集
451 称之为最大的。这两个定义在图10-15中说明。注意，如图中的例子所示，最大独立集不是唯一的。在某些类型的任务图中，顶点的最大独立集可用来确定哪些计算可以并行进行。例如，在不完全因式分解并行算法中，最大独立集可用来确定可并发因式分解的行的集合，另外，最大独立集也可用来并行计算图的着色问题。

计算顶点的最大独立集已提出许多算法。最简单的一类算法从初始设置集合I为空开始，
452 将所有的顶点放入集合C中，作为等待进入I的候选顶点集合。然后算法重复地将顶点从C中移到I中，并从C中除去所有和v邻接的顶点。这个过程到集合C为空时终结，此时I为最大独立集。

得到的集合I含有顶点的独立集，因为每次向I中添加顶点时，都要从C中除去所有以后添加进去将违反独立条件的顶点。此外，得到的集合也是最大的，因为任何一个还未在I中的顶点都至少与I中的一个顶点邻接。



{a, d, i, h}是独立集
{a, c, j, f, g}是最大独立集
{a, d, h, f}是最大独立集

图10-15 独立集及最大独立集示例

虽然上面的算法非常简单，但因为其串行性质，不适合并行处理。因此，并行MIS算法通常基于最初由Luby开发的用于计算图的着色问题的随机算法。使用Luby算法，一个图的顶点V的最大独立集I用下面的递增形式计算：集合I初始设置为空，并设置候选顶点集合C与集合V相等。一个唯一的随机数被分配给C中的每一个顶点，如果某一顶点的随机数小于它的相邻顶点的所有随机数，则它加入到I中。集合C被更新，除去所有被选择加入到I中的顶点以及它们的相邻顶点。注意被选择加入到I中的顶点确实是独立的（即没有通过一条边的直接连接）。这是因为，如果v被插入到I中，则分配给v的随机数是所有分配给它的相邻顶点的随机数中最小的；因此，不会选择相邻于v的其他顶点u加入。对于C中剩余的顶点，重复上面的随机数分配及顶点选择步骤，而I同样地扩大。当C变为空时，I的递增扩大结束。平均而言，这个算法在$O(\log|V|)$次递增后收敛。图10-16展示算法在一个小图上的执行情况。下面讨论在共享地址空间计算机上Luby算法的并行形式。这个算法在消息传递平台上的实现在讲述消息传递的那一章中讨论。

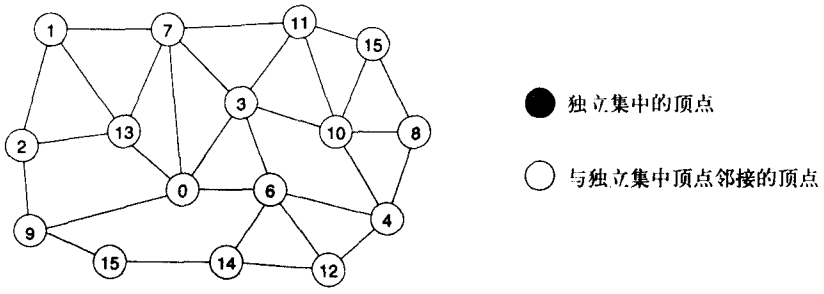
共享地址空间并行形式

在共享地址空间并行计算机上，Luby的MIS算法的并行形式如下。令I为大小为$|V|$的数组。在算法终止时，如果v_i是MIS的一部分，则$I[i]$存储1，否则0。初始时，所有I中的元素都被设置为0，在Luby算法的每次迭代中，数组中的一些项会变为1。令C为大小是$|V|$的数组，在算法的执行过程中，如果顶点v_i是候选集的一部分，则$C[i]$为1，否则为0。初始时，所有C中的元素被设置为1。最后，令R为大小是$|V|$的数组，存储分配给每个顶点的随机数。

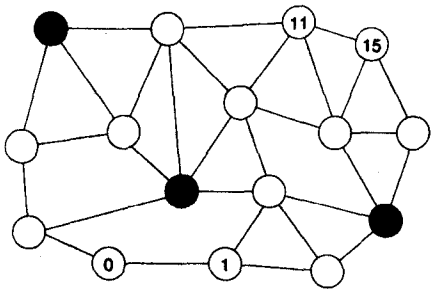
在每次迭代中，集合C在p个进程中划分。每个进程为从C中分配到的顶点产生一个随机数。当所有的进程都产生随机数后，就来确定哪些顶点能包含在I中。特别地，对每个分配给它们的顶点，这些进程都要检查分配给自己的随机数是否小于分配给它的所有相邻顶点的随机数。如果是，这些进程设置I中相应的项为1。由于R是共享的并能被所有的进程访问，确定某一顶点是否包含在I中非常简单。 453

数组C也可用以如下的直接方式更新。每个进程在确定了某一个顶点v将成为I中的一员后，将与它的邻接矩阵对应的C中的项置0。注意即使可能有不止一个进程会设C中的项为0（因为它可能和不止一个待插入到I中的顶点邻接），这种并发写入操作也不会改变结果的正确性，因为被并发写的值是相同的。

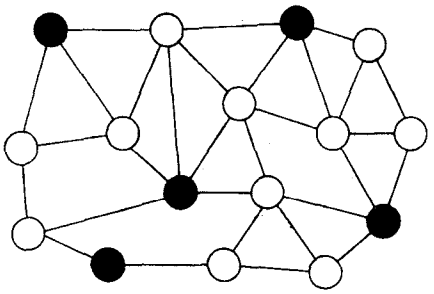
Luby算法每次迭代的复杂度与串行算法的相似，但要加上每个随机数分配后额外的全局同步开销。有关Luby算法的详细分析留作习题（习题10.16）。



a）第一次随机数赋值后



b）第二次随机数赋值后



c）最后的最大独立集

图10-16 Luby随机最大独立集算法的不同扩大步骤。每个顶点中的数字对应于分配给每个顶点的随机数

454

10.7.2 单源最短路径

容易修改Dijkstra的单源最短路径算法，使其能有效地应用于寻找稀疏图最短路径。修改后的算法称为Johnson算法。回忆Dijkstra算法在每次迭代中进行如下两步操作。第一步，找出一个顶点$u \in (V-V_T)$，其中$l[u] = \min\{ l[v] | v \in (V-V_T)\}$，并把它插入到集合$V_T$中。第二步，对于每个顶点$v \in (V-V_T)$，计算$l[v]= \min\{ l[v], l[u] + w(u, v)\}$。要注意的是，在第二步中，只需要考虑$u$的邻接表中的顶点。由于图是稀疏的，与$u$邻接的顶点数目要小于$\Theta(n)$；因此，使用邻接表表示能提升性能。

Johnson算法使用优先队列Q来存储每个顶点$v \in (V-V_T)$的$l[v]$值。优先队列的构造方法是，l中值最小的顶点总在队列的前面。通常用最小二分堆来实现优先队列。我们可用最小二分堆在时间$O(\log n)$内对每个顶点更新$l[v]$。算法10-5显示Johnson算法。初始时，对于不是源点的每个顶点v，算法将$l[v] = \infty$插入到优先队列中。对于源顶点s，算法插入$l[s] = 0$。在算法的每一步，l中具有最小值的顶点$u \in (V-V_T)$从优先队列中除去。遍历u的邻接表，对于每条边(u, v)，在堆中更新从$l[v]$到顶点v距离。更新堆中的顶点占据算法整体运行时间的主要部分。更新的总次数等同于边的个数；因此，Johnson算法的整体复杂度为$\Theta(|E| \log n)$。

算法10-5 Johnson的串行单源最短路径算法

```
1.  procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s)
2.  begin
3.      Q := V;
4.      for all v ∈ Q do
5.          l[v] := ∞;
6.      l[s] := 0;
7.      while Q ≠ ∅ do
8.      begin
9.          u := extract_min(Q);
10.         for each v ∈ Adj[u] do
11.             if v ∈ Q and l[u] + w(u, v) < l[v] then
12.                 l[v] := l[u] + w(u, v);
13.     endwhile
14. end JOHNSON_SINGLE_SOURCE_SP
```

455

1. 并行化策略

有效的Johnson算法的并行形式必须能有效地维持优先队列Q。简单的策略是让一个进程，例如P_0，来维持Q。其他的进程可以对$v \in (V-V_T)$计算$l[v]$的新值，并把值传给P_0以更新优先队列。这种策略有两个主要局限。首先，由于用一个进程来维护优先队列，整体并行运行时间为$O(|E|\log n)$（共有$O(|E|)$次队列更新，每次更新需时$O(|E|\log n)$），这样就导致此并行形式没有渐近加速比，因为$O(|E|\log n)$与串行运行时间相同。其次，在每次迭代过程中，算法大约要更新$|E|/|V|$个顶点。结果，在任一指定时刻不会有超过$|E|/|V|$个进程忙于运算，对于绝大多数有价值的稀疏图类别来说，$|E|/|V|$是很小的值，并且它与图的大小无关。

通过将维持优先队列的任务分配给多个进程可以减少以上第一种局限，但这不是一项简单的任务，只有在像共享地址空间计算机这样的低延迟体系结构上才能有效解决。然而，即使在最佳条件下，当每次优先队列更新只需$O(1)$时间时，能得到的最大加速比也只是非常小的$O(\log n)$。缓解第二种局限的方法是，根据处于优先队列顶部顶点的l值大小，一次可以取出一个以上的顶点。特别地，如果v是位于优先队列顶部的顶点，满足$l[u] = l[v]$的所有的顶点都可以提取出来，而它们的邻接表进行并发处理。这是因为，和源点有着同样最小距离的顶点可以按任意顺序处理。注意，为了实现这个方法，必须用锁步的方法处理和源点有着同样最小距离的所有顶点。如果我们知道图中所有边的最小权值为m，那么还可以进一步提高并发度。在这种情况下，所有满足条件$l[u] \leqslant l[v] + m$都可（以锁步形式）并发处理。这些顶点u称为安全（safe）顶点。然而，只有当多于一个优先队列的更新操作能并发进行时，这种并发才能带来比$O(\log n)$更好的渐近加速比，但却使维持单一优先队列的并行算法更加复杂。

到目前为止，我们主要讨论开发Johnson算法的并行形式，以串行算法同样的顺序查找到每个顶点的最短路径，以及并发地探寻安全顶点。但是，从上面可以看出，这种方法会导致算法复杂化，且使并发程度降低。另一种方法是，开发一种对安全的和不安全的顶点都并发处理的并行算法，只要这些不安全的顶点可以通过源点经由一条路径到达即可，这条路径包含最短路径已算出的顶点（即它们在优先队列中相应的l值不是无穷）。特别地，在这种算法中，p个进程的每一个都取出p个顶部顶点的一个，并继续更新与它相邻的顶点的l值。当然，这种方法的问题是不能保证从优先队列中取出的顶点的l值一定与最短路径的成本对应。例如，

456

考虑u和v是优先队列顶部的两个顶点，且$l[v] < l[u]$。根据Johnson算法，当顶点从优先队列中取出时，它的l值是从源点到该顶点的最短路径的成本。现在，如果有一条边连接v和u，使得$l[v] + w(u, v) < l[u]$，则此时到u的最短路径的正确值是$l[v] + w(u,v)$，而不是$l[u]$。但是，如果能检查出到某一顶点的最短路径的计算有误，并用更新后的l值插入回优先队列，就能保证结果的正确性。我们可用如下的方法进行检查。考虑一个顶点v刚从队列中取出并令u是与v相邻且已从队列中取出的顶点。如果$l[v] + w(u, v)$小于$l[u]$，则到u的最短路径的计算有误，需要将u插回到优先队列中，且$l[u] = l[v] + w(u, v)$。

图10-17所示的网格图展示这个方法的执行过程。图中的例子有三个进程，要找出来自顶点a的最短路径。初始化优先队列后，顶点b和顶点d从源点可达。在第一步，进程P_0和P_1取出顶点b和顶点d，并更新与b和d相邻的顶点的l值。在第二步，进程P_0、P_1和P_2取出顶点e、c和g，并更新与它们相邻的顶点的l值。注意，当处理顶点e时，进程P_0检查$l[e] + w(e, d)$是否小于或大于$l[d]$。在本例中$l[e] + w(e, d) > l[d]$，表示考虑e时，先前计算出的到d的最短路径不会改变，并且到此步为止所有的计算都是正确的。在第三步，进程P_0和P_1分别处理h和f。当进程P_0比较$l[h] + w(h, g)$（值为5）与在前一次迭代中取出的$l[g]$（值为10）时，进程P_0发现$l[h] + w(h, g)$较小。因此，它把顶点g插回到优先队列，且更新$l[g]$值。最后，在第四步和最后一步，从优先队列中取出剩余的两个顶点并计算所有单源最短路径。

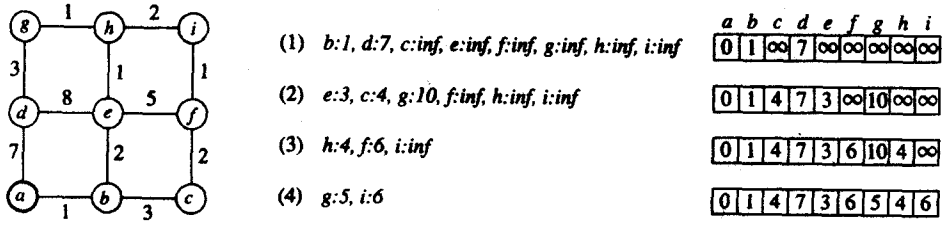


图10-17 用修正后的Johnson算法并发处理不安全顶点的实例

这种Johnson算法的并行形式属于3.2.4节讨论过的推测分解方法。事实上，算法假设优先
457 队列中顶部p个顶点的l[]值在处理这些顶点时不发生改变，这样就能继续进行Johnson算法所需的计算。但是，如果在后面的步骤发现假设错误，算法就返回，并重新计算有关顶点的最短路径。

为了有效实现这种推测分解算法，必须消除用一个优先队列的瓶颈。接下来我们将提出一个消息传递算法，它用推测分解来达到并发，且没有单一的优先队列。作为代替，每个进程对分配给它的顶点维持自己的优先队列。习题10.13还讨论另一种方法。

2. 分布式内存形式

令p为进程数，$G = (V, E)$为稀疏图。我们将顶点集合V划分为p个不相连的集合V_1，V_2，...，V_p，并将每个顶点集及其对应的邻接表分配给p个进程之一。每个进程对分配给它的顶点维持一个优先队列，并计算从源点到这些顶点的最短路径。这样优先队列Q就分成p个不相连的优先队列Q_1，Q_2，...，Q_p，每个分配给一个独立的进程。除了优先队列外，每个进程P_i还维持一个数组sp，使得对于每个$v \in V_i$，$sp[v]$存储从源顶点到v最短路径的成本。每次当顶点v从优先队列中取出时，成本$sp[v]$更新为$l[v]$。初始时，对于不是源点的顶点v，$sp[v] = \infty$，而对于

源顶点s，我们将$l[s]$插入到相应的优先队列中。每个进程对其本地的优先队列执行Johnson算法。在算法的最后，$sp[v]$存储从源点到顶点v的最短路径长度。

当进程P_i从Q_i中取出使$l[u]$最小的顶点$u \in V_i$时，分配给其他进程的顶点的l值可能需要更新。进程P_i发送消息给存储与u相邻顶点的进程，告知它们这些新值。收到这些新值后，进程更新l的值。例如，假设有一条边(u, v)，其中$u \in V_i$，$v \in V_j$，进程P_i刚从优先队列中取出顶点u。然后进程P_i发送消息给P_j，P_j中包含$l[v]$可能的新值$l[u] + w(u, v)$。进程P_j收到这个消息后，将存储在它的优先队列中的$l[v]$改为$\min\{ l[v], l[u] + w(u, v)\}$。

由于进程P_i和P_j执行Johnson算法，进程P_j可能已从它的优先队列中取出了顶点v。也就是说进程P_i可能已经算出从源点到顶点v的最短路径$sp[v]$。这样就有两种可能情况：或者$sp[v] \leq l[u] + w(u, v)$，或者$sp[v] > l[u] + w(u, v)$。第一种情况表示存在一条经过顶点$u$到顶点$v$的更长路径，第二种情况表示存在一条经过顶点$u$到顶点$v$的更短路径。对于第一种情况，进程$P_j$无需做什么，因为到$v$的最短路径没有改变。对于第二种情况，进程$P_j$必须更新到顶点$v$的最短路径成本。为了实现这一步，必须以$l[v] = l[u] + w(u, v)$将顶点$v$插回到优先队列，不考虑$sp[v]$的值。由于能把顶点$v$再插入到优先队列中，只有当所有的队列为空时，算法才终止。 458

初始时，只有含源点进程的优先队列为非空。此后，当创建包含l新值的消息并发送到相邻的进程时，其他进程的优先队列中就会含较多的项。当进程收到l的新值后，把新值插入到自己的优先队列并进行计算。考虑在网格图中计算源点位于左下角的单源最短路径问题，计算在网格图中以波动的形式向前传播。在波到达前进程空闲，波通过后进程也空闲，如图10-18所示。在算法执行的任意时刻，只有沿着波的进程繁忙。其他的进程要么还没有开始计算，要么已经计算完毕。下面几段将讨论三种将网格图映射到p个进程格网的方法。

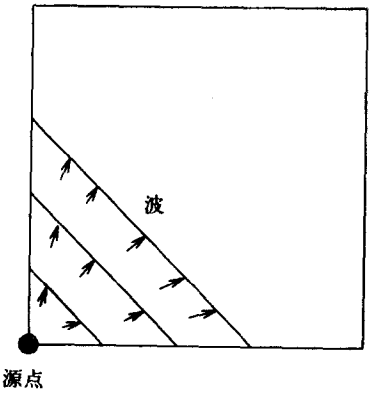


图10-18 优先队列中活动的波

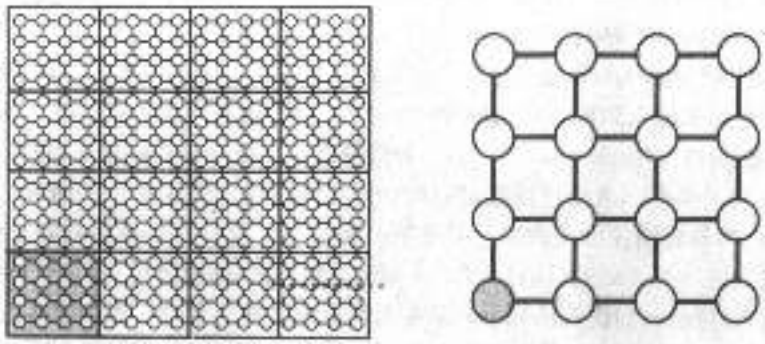
2维块映射 使用2维块映射（3.4.1节）可以将$n \times n$网格图映射到p个处理器。特别地，我们把p个进程看作逻辑格网，并把大小为 $n/\sqrt{p} \times n/\sqrt{p}$ 个顶点的不同块分配给每个进程，图10-19说明这种映射。

在任何时刻，繁忙的进程数目等于与波相交的进程数目。由于波按对角线方向运动，在任何时刻繁忙的进程数目不超过$O(\sqrt{p})$。令W为执行串行算法需要做的全部工作。如果我们假设在任何时刻都有$\sqrt{p}$个进程进行计算，并忽略进程间的通信开销以及其他的额外工作，则 459
最大加速比及效率如下：

$$S = \frac{W}{W/\sqrt{p}} = \sqrt{p}$$

$$E = \frac{1}{\sqrt{p}}$$

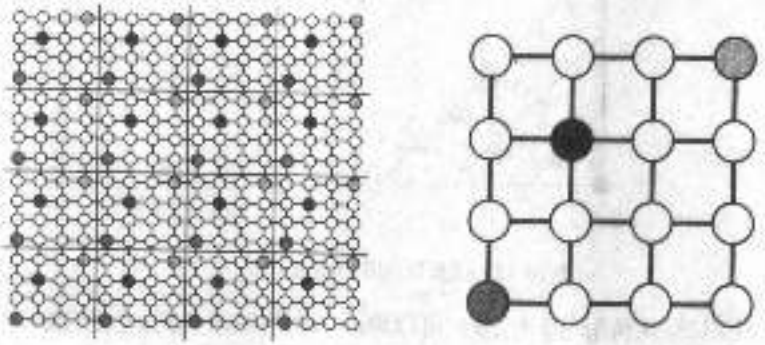
这种映射的效率很低，且当进程数目增加时效率会更糟。



a) b)

图10-19 用2维块映射将网格图a）映射到格网b）。本例中，$n=16$，$\sqrt{p}=4$，阴影顶点映射到阴影进程

2维循环映射 2维块映射的主要限制是每个进程只负责网格中一个很小的有限区域。如果使用2维循环映射，则可以使每个进程负责网格中一个扩散的区域（3.4.1节）。这样就延长每个进程繁忙的时间。在2维循环映射中，$n \times n$网格图被分成n^2/p个块，每个块的大小为$\sqrt{p} \times \sqrt{p}$。每个块映射到$\sqrt{p} \times \sqrt{p}$进程格网。图10-20展示这种映射。每个进程包含一个有n^2/p个顶点的块。这些顶点在图的对角线上，相隔$\sqrt{p}$个顶点。每个进程大约分配$2n/\sqrt{p}$个这种对角线。



a) b)

图10-20 用2维循环映射将网格图a）映射到格网b）。本例中，$n=16$，$\sqrt{p}=4$，阴影顶点映射到相应的阴影格网进程

现在，每个进程都负责属于网格图中不同部分的顶点。当波经过图传播时，波与每个进程的一些顶点相交。因此，在算法的绝大部分时间里处理器保持繁忙。但2维循环映射比2维块映射有更大的通信开销，因为相邻的顶点驻留在分开的进程中，每次进程从它的优先队列中取顶

点u时必须将$f[u]$的新值告知其他的进程。对这种映射的分析留给读者作为练习（习题10.17）。

1维块映射 上面讨论的两种映射都有局限性。在任何时刻2维块映射最多只能保持 $O(\sqrt{p})$ 个进程繁忙，而2维循环映射有很高的通信开销。另一种映射把p个进程看成线性阵列，并使用1维块映射分配网格图的n/p个条给每个处理器。图10-21展示这种映射。

460

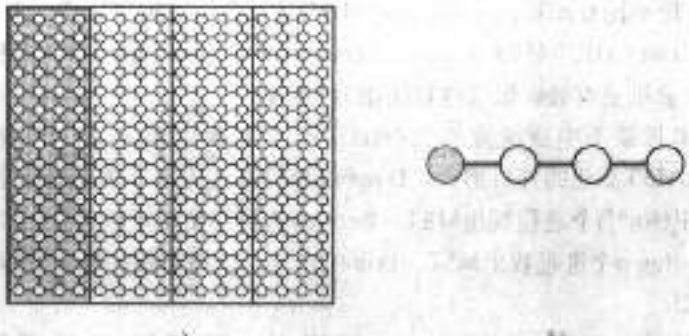


图10-21 将网格图a）映射到进程线性阵列b）。本例中，$n=16$，$p=4$。阴影顶点映射到阴影进程

初始时，波只和一个进程相交。随着计算的进展，波与第二个进程相交，使两个进程都繁忙。随着算法的继续，波与更多的进程相交，使这些进程变成繁忙的，直到所有进程繁忙（即它们都与波相交）。过了这一点后，繁忙的进程数目减少。图10-22展示波的传播。如果假设波以恒定的速度传播，则有$p/2$个进程（平均值）繁忙。如果忽略所有开销，这种映射的加速比和效率如下：

461

$$S = \frac{W}{W/(p/2)} = \frac{p}{2}$$

$$E = \frac{1}{2}$$

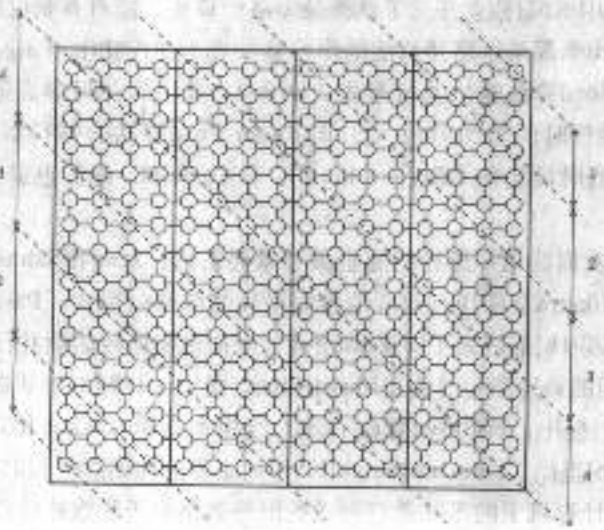


图10-22 计算波在网格图中穿过时繁忙的进程数目

因此，这种映射的最大效率为50%。1维块映射在本质上好于2维块映射，但不能使用多于$O(n)$个进程。

10.8 书目评注

许多书籍详细讨论图论及图算法。Gibbons[Gib85]是本章中提到算法的一本很好的参考书。Aho，Hopcroft和Ullman[AHU74]和Cormen，Leiserson和Rivest[CLR90]详细讲述多种图的算
462 法，以及与在串行计算机上有效实现这些算法相关的问题。

10.2节讲述的串行最小生成树算法由Prim[Pri57]提出。Bentley[Ben80]以及Deo和Yoo[DY81]给出PrimMST算法的并行形式。Deo和Yoo的算法适用于共享地址空间计算机。它能在$\Theta(n^{1.5})$时间内用$\Theta(n^{0.5})$个进程找出MST。Bentley的算法使用于树连接的收缩数组，能在$\Theta(n \log n)$时间内用$n/\log n$个进程找出MST。Prim的最小生成树算法的超立方体形式（10.2节）与Bentley的算法相似。

图的MST也可用Kruskal[Kru56]或Sollin[Sol77]的串行算法计算。Sollin算法（习题10.21）的复杂度为$\Theta(n^2 \log n)$。Savage和Jaja[SJ81]在CREW PRAM上为Sollin算法开发了一种并行形式，能在$\Theta(\log^2 n)$时间内用n^2个进程找出最小生成树。而Chin，Lam和Chen[CLC82]在CREW PRAM上为Sollin算法开发了另一种并行形式，能在$\Theta(\log^2 n)$时间内用$n[n/\log n]$个进程找出最小生成树。Awerbuch和Shiloach[AS87]在混洗-交换网络为Sollin算法提出一种并行形式，能在$\Theta(\log^2 n)$时间内使用$\Theta(n^2)$个进程完成。Doshi和Varman[DV87]在含p个进程的环形连接计算机上为Sollin算法提出用时为$\Theta(n^2/p)$的算法。Leighton[Lei83]，以及Nath，Maheshwari和Bhatt[NMB83]在格网树网络为Sollin算法提出并行形式。对于$n \times n$的格网树，前一个算法用时为$\Theta(\log^2 n)$，后一种算法用时为$\Theta(\log^4 n)$。对$\sqrt{p} \times \sqrt{p}$的树形格网。Huang[Hua85]描述Sollin算法的并行形式的运行时间为$\Theta(n^2/p)$。

10.3节讲述的单源最短路径算法由Dijkstra[Dij59]发现。因为Dijkstra算法与Prim的MST算法很相似，前面几段讲述的Prim算法的所有并行形式都能应用于单源最短路径问题。Bellman[Bel58]和Ford[FR62]独立开发了单源最短路径算法，能对有负权但没有负权圈的图进行操作。Bellman-Ford单源最短路径算法的串行复杂度为$O(|V||E|)$。Paige和Kruskal[PK89]为Dijkstra和Bellman-Ford算法提出并行形式。Dijkstra算法的并行形式运行在$\Theta(n)$个进程的EREW PRAM上，运行时间为$\Theta(n \log n)$；Bellman-Ford算法的并行形式运行在p个进程的EREW PRAM上，运行时间为$\Theta(n|E|/p + n \log p)$，其中$p \leq |E|$。他们也提出CRCW PRAM上的算法[PK89]。

人们对全部顶点对间的最短路径问题也做了重要工作。Jenq和Sahni[JS87]以及Kumar和Singh[KS91b]讨论Dijkstra全部顶点对间的最短路径的源划分形式。Paige和Kruskal[PK89]，以及Kumar和Singh[KS91b]都研究了Dijkstra全对顶点间最短路径的源并行形式。10.4.2节讲述的Floyd全部顶点对间的最短路径算法由Floyd[Flo62]提出。1维和2维块映射形式（习题10.8）由Jenq和Sahni[JS87]提出，而Floyd算法的流水线版本由Bertsekas和Tsitsiklis[BT89]以及Kumar和Singh[KS91b]提出。Kumar和Singh[KS91b]还为全部顶点对间的最短路径算法在超立
463 方体连接和格网连接计算机上的不同并行形式提出等效率分析和性能比较。10.4.3节的讨论借鉴Kumar和Singh[KS91b]以及Jenq和Sahni[JS87]的工作。特别地，算法10-4取自Jenq和

Sahni[JS87]的论文。Levitt和Kautz[LK72]提出Floyd算法的一种二维细胞阵列的并行形式，用n^2个进程而运行时间为$\Theta(n)$。Deo，Pank和Lord在CREW PRAM模型上为Floyd算法开发了一种并行形式，使用n^2个进程的复杂度为$\Theta(n)$。Candy和Misra[CM82]提出基于扩散式计算的分布式全部顶点对间的最短路径算法。

10.6节讲述的连通分量算法由Woo和Sahni[WS89]提出。Cormen，Leiserson和Rivest[CLR90]讨论使用等级和路径压缩有效地实现不相连集数据结构的方法。还有几种算法用于计算连通分量，其中有些算法基于顶点压缩技术，与Sollin的最小生成树算法相似。绝大多数Sollin算法的并行形式都能找出连通分量。Hirschberg[Hir76]以及Hirschberg，Chandra和Sarwate[HCS79]为连通分量算法开发了基于顶点压缩的并行形式，前者在CREW PRAM上使用进程数目为n^2时的复杂度为$\Theta(\log^2 n)$，而后者使用$m\lceil n/\log n\rceil$个进程的复杂度相同。Chin，Lam和Chen[CLC82]通过在CREW PRAM上把进程数目减少为$n\lceil n/\log^2 n\rceil$，更有效地实现了顶点压缩算法，且使运行时间保持在$\Theta(\log^2 n)$。Nassimi和Sahni[NS80]使用顶点压缩技术对格网连接的计算机开发了一种并行形式，使用n^2个进程，运行时间为$\Theta(n)$。

10.7.2节中讲述的稀疏图单源最短路径算法是由Johnson[Joh77]提出的。Paige和Kruskal[PK89]讨论并行维持队列Q的可行性。Rao和Kumar[RK88a]提出在优先队列中进行并发插入和删除操作的方法。Johnson算法的2维块映射、2维循环映射以及1维块映射（10.7.2节）由Wada和Ichiyoshi[WI89]提出。他们对这些方法在格网连接并行计算机上也进行了理论和实验评估。

Luby[Lub86]提出10.7.1节讲述的串行最大独立集算法，其在共享地址空间体系结构上的并行形式受到Karypis和Kumar[KK99]算法的影响。Jones和Plassman[JP93]为Luby算法开发了一个异步版本，它特别适用于分布式内存的并行计算机。在他们的算法中，每个顶点被分配一个随机数，经过一步通信操作后，每个顶点都要确定与它相邻顶点中大于它的随机数或小于它的随机数的顶点个数。此时，每个顶点进入一个循环，等待接收它的相邻顶点中随机数更小的顶点的色彩值。一旦收到了所有的色彩值，顶点选择一个连续的色彩，并把它发送给所有有更大随机数的顶点。算法在所有的顶点着色后终止。注意，除了用来确定更小的和更 464
大的相邻顶点数目的初始通信操作外，这个算法都异步执行。

人们也提出了其他的并行图算法。Shiloach和Vishkin[SV82]提出了在有向流网络查找最大流的算法，网络有n个顶点，在n个进程的EREW PRAM上的运行时间为$O(n^2\log n)$。Goldberg和Tarjan[GT88]提出了一种不同的最大流算法，算法在n个进程的EREW PRAM上的运行时间为$O(n^2\log n)$，但需要的空间较小。Atallah和Kosaraju[AK84]提出了多个用于格网连接并行计算机的算法。他们考虑的算法包括：用于查找无向图中桥和联接点的算法、查找最短圈长度的算法、查找MST的算法、查找循环索引的算法以及测试图是否为二分图的算法。Tarjan和Vishikin[TV85]提出了计算图的双连通分量的算法。他们的CRCW PRAM形式在使用$\Theta(|E|+|V|)$个进程时所需时间为$\Theta(\log n)$，而他们的CREW PRAM形式在使用$\Theta(n^2/\log^2 n)$个进程时所需时间为$\Theta(\log^2 n)$。

习题

10.1 在Prim最小生成树算法的并行形式中（10.2节），超立方体中能有效使用的最大进程数目为$\Theta(n/\log n)$。使用$\Theta(n/\log n)$个进程所需的运行时间为$\Theta(n\log n)$。如果使用$\Theta(n)$个进

程，则运行时间是多少？在消息传递并行计算机上能获得的最小并行运行时间是多少？这个运行时间与使用$\Theta(n/\log n)$个进程相比，结果如何？

10.2 说明为什么Dijkstra单源最短路径算法及其并行形式（10.3节）为了输出最短路径而不是输出成本时，需要做一些修改？分析你的串行形式及并行形式的运行时间。

10.3 给定一个图$G=(V, E)$，图G中顶点的宽度优先等级是从节点v对图作宽度优先遍历时，分配给V中的顶点的值。请说明如何在p个进程的格网中实现图G顶点的宽度优先等级。

10.4 Dijkstra的单源最短路径算法（10.3节）需要边权值为非负。请修改该算法，使之能适用于边权值为负但不含负权圈的图，且运行时间为$\Theta(|E||V|)$。在p个进程的消息传递计算机上分析修改后算法并行形式的性能。

10.5 10.4节中讨论了全部顶点对间的最短路径算法的多种并行形式，请计算这些形式所需的内存总量。

10.6 如果用下面的语句行替换算法10-3的第7行，说明10.4.2节讲述的Floyd算法仍是正确的：

465

$$d_{i,j}=\min\{d_{i,j},(d_{i,k}+d_{k,j})\}$$

10.7 请计算Floyd全部顶点对间的最短路径算法的并行运行时间、加速比以及效率。a. 在采用存储转发路由选择的p个进程格网上使用2维块映射，b. 在采用直通路由选择的p个进程超立方体中使用2维块映射 c. 在采用直通路由选择的p个进程格网中使用2维块映射。

10.8 在Floyd全部顶点对间的最短路径算法中，也可以使用1维块映射（3.4.1节）划分矩阵$D^{(k)}$。矩阵$D^{(k)}$中n/p个连续的列被分配给p个进程的每一个。

a）计算在超立方体连接的并行计算机上使用1维块映射时的并行运行时间、加速比以及效率。与10.4.2节中讲述的2维块映射相比，这种映射的优点和缺点分别是什么？

b）分别计算在采用存储转发路由选择的p个进程格网、直通路由选择的p个进程格网以及直通路由选择的p个进程环中使用1维块映射时的并行运行时间、加速比以及效率。

10.9 描述并分析使用10.4.2节讨论的1维块映射及流水线技术时，Floyd算法的并行形式的性能。

10.10 试计算Floyd流水线形式（10.4.2节）的准确并行运行时间、加速比以及效率。

10.11 分别计算在p个进程的格网中采用存储转发路由选择及直通路由选择时，10.6节讲述的连通分量算法并行形式的并行运行时间、加速比以及效率。比较两种不同体系结构上的性能差异。

10.12 10.6节讲述的连通分量问题的并行形式用1维块映射在多个进程间划分矩阵。请考虑另一种使用2维块映射的并行形式。并分别在超立方体、采用存储转发路由选择的格网以及采用直通路由选择的格网上描述并分析这种形式的性能和可扩展性。将这种方法与1维块映射相比较。

10.13 请考虑如何针对稀疏图（10.7.2节）将Johnson单源最短路径算法并行化。一种并行形式是使用p_1个进程维持优先队列，p_2个进程计算新的l值。有多少个进程能被有效地用于维持优先队列（换句话说，p_1的最大值是多少）？有多少个进程可用来更新l的值？使用p_1+p_2个进程得到的并行形式是不是成本最优的方案？描述用p_1个进程维持优先队列的算法。

10.14 考虑针对稀疏图的Dijkstra单源最短路径算法（10.7节）。通过在p个进程的超立方

体上将n个邻接表在水平方向的进程间划分使该算法并行化；也就是，每个进程得到n/p个表。 466
这种形式的并行运行时间是多少？邻接表也可在进程间垂直分配，即每个进程得到每个邻接表的一部分。如果邻接表中包含m个元素，则每个进程包含有m/p个元素的子表。每个子表的最后一个元素有一个指针，指向下一个进程的元素。这种形式的并行运行时间及加速比是多少？它能使用的最大进程数目是多少？

10.15 对Floyd的全部顶点对间的最短路径算法重复习题10-14。

10.16 在10.7.1节中，讨论了用于查找稀疏图中顶点最大独立集的Luby共享地址空间算法。试分析该算法的性能，并求出该算法的并行运行时间及加速比。

10.17 计算在格网连接计算机中稀疏图单源最短路径算法（10.7.2节）在采用2维循环映射时的并行运行时间、加速比以及效率。计算时可以忽略额外工作的开销，但必须考虑由通信带来的开销。

10.18 分析使用2维块-循环映射（3.4.1节）时稀疏图的单源最短路径算法（3.4.2节）的性能。将其与上一题采用2维循环映射时的性能比较。与上一题一样，忽略额外工作的开销，但必须考虑由通信带来的开销。

10.19 考虑3.4.1节中讲述的1维块-循环映射。描述如何将这种映射应用到稀疏图的单源最短路径算法中。计算这个映射的并行运行时间、加速比以及效率。计算时可以忽略额外工作的开销，但必须考虑由通信带来的开销。

10.20 将10.7.2节提出的方案与习题10.18和10.19中的方案作比较，哪种方案由额外计算带来的开销最小？

10.21 Sollin算法（10.8节）从n个孤立顶点的树林开始。在每次迭代中，对于树林中的每一棵树，算法同时确定树中的任意顶点与另一棵树中的顶点相连的最小边。所有这些最小边都被加入到树林中。而且，两棵树只可能用一条边相连。这一过程进行到树林中只有一棵树时结束，这棵树就是最小生成树。由于在每次迭代中树的数目至少减少一半，算法最多需要$\log n$次迭代寻找最小生成树。每次迭代最多需要$O(n^2)$次比较来找出与每个顶点相关联的最小边；因此，串行复杂度为$\Theta(n^2 \log n)$。试在n个进程的超立方体连接并行计算机上，开发Sollin算法的并行形式。你的并行形式的运行时间是多少？这种形式是不是成本最优的？ 467

第11章　离散优化问题的搜索算法

搜索算法可用来求解离散优化问题(DOP)，这是一类具有重大理论与实践意义的计算代价很高的问题。求解DOP的搜索算法通过从有限或者可数无限的可能解集合中找出满足特定问题准则的候选解。离散优化问题也称为组合问题。

11.1　定义与实例

离散优化问题（discrete optimization problem）可以用二元组(S, f)表示。集合S是所有满足特定约束条件的有限或可数无限的解集合。这个集合称为可行解集合（feasible solutions）。f是成本函数，它将S中的每一个元素映射到实数集合R中：

$$f : S \rightarrow R$$

DOP的目标就是找到一个可行解x_{opt}，使得$f(x_{opt}) \leqslant f(x)$，其中$x \in S$。

DOP可用来表示许多领域的问题，例如规划和调度，VLSI芯片布局优化，机器人动作规划，数字电路测试模式产生，以及后勤和控制等。

例11.1　0/1整数线性规划问题

在0/1整数线性规划问题中，有一个$m \times n$的矩阵A，一个$m \times 1$的向量b，以及一个$n \times 1$的向量c。目的是找出一个元素的取值只能是0或1的$n \times 1$维向量$\bar{x}$，$\bar{x}$必须满足约束条件 469

$$A\bar{x} \geqslant b$$

而且函数

$$f(\bar{x}) = c^T\bar{x}$$

必须取最小值。对于这个问题，集合S就是所有满足方程$A\bar{x} \geqslant b$的向量$\bar{x}$的值的集合。 ■

例11.2　九宫问题

九宫问题就是在3×3的格子中，放有编号从1到8的8个滑块，还剩一空格（blank）。与空格相邻的滑块可以移动到空格上面，当某个滑块移走后，它原来的位置出现一新空格。根据滑块的排列方式不同，滑块有4种可能的移动方式：上、下、左、右。九宫重排问题就是在初始排列以及最终排列已经给定情况下，找出一种移动方法：要求用最少的滑块移动次数，将滑块由初始排列转变为最终排列。图11-1是一九宫问题的例子，前面给出了初始排列以及最终排列，以及从初始排列到最终排列一种移动方法。 470

此问题的S集合就是将滑块从初始排列转变为最终排列的所有移动序列的集合。S中一个元素的成本函数f就是该移动序列中滑块的移动次数。 ■

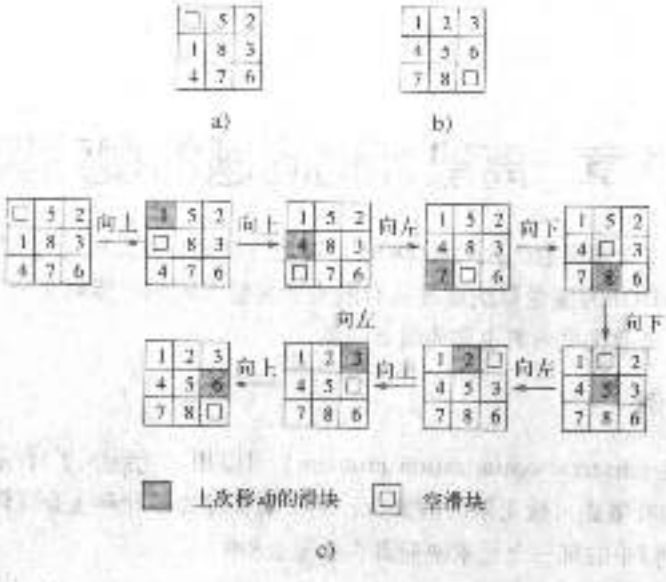


图11-1 九宫问题的一个实例：a）滑块初始排列状态；b）滑块最终排列状态；c）从初始排列到最终排列的滑块移动序列

对于绝大多数实际问题而言，解集合S是相当大的。因此，要找出S中全部元素，并从中确定最优解x_{opt}，实际上并不可行。然而，DOP可以看成是寻找一条从初始节点到几个可能的目标节点的最小成本路径问题。S中的每个元素x均可看成是一条从初始节点到某个目标节点的路径。图中的每条边均有相应的成本，成本函数f被定义为这些边的成本。就大多数问题而言，路径的成本就是边的成本之和。这样的图称为状态空间（state space），图中节点称为状态，没有后继的节点称为终止节点（terminal node），其他的节点称为非终止节点（nonterminal node）。九宫问题可以很自然地表述成图的搜索问题。特别，它的初始排列就是初始节点，最终排列为目标节点。例11.3展示把重新表述0/1整数线性规划问题的过程作为一个图搜索问题。

例11.3 重新表述0/1整数线性规划问题

考察例11.1中定义的0/1线性规划问题的一个实例。令A，b，c的值如下：

$$A=\begin{bmatrix}5 & 2 & 1 & 2\\ 1 & -1 & -1 & 2\\ 3 & 1 & 1 & 3\end{bmatrix},\ b=\begin{bmatrix}8\\ 2\\ 5\end{bmatrix},\ c=\begin{bmatrix}2\\ 1\\ -1\\ -2\end{bmatrix}$$

对A，b，c的限制条件如下：

$$\begin{aligned}5x_1+2x_2+x_3+2x_4 &\geqslant 8\\ x_1-x_2-x_3+2x_4 &\geqslant 2\\ 3x_1+x_2+x_3+3x_4 &\geqslant 5\end{aligned}$$

另外，取最小值的函数$f(x)$为

$$f(x)=2x_1+x_2-x_3-2x_4$$

向量 $\bar{x}$ 中的4个元素只能取0或1。$\bar{x}$ 共有$2^4 = 16$个可能的值。然而，许多值并不满足问题的限制条件。

现将问题重新表述为图的搜索问题。在初始节点中，向量中的所有元素均未赋值。本例中，我们按向量中元素下标的次序对向量元素赋值，即首先对x_1赋值，然后是x_2，并依次类推。 471
这样，初始节点产生两个节点，即$x_1 = 0$和$x_1 = 1$。在一个变量x_i被赋值后，它就称为固定变量(fixed variable)。所有非固定变量称为自由变量（free variable)。

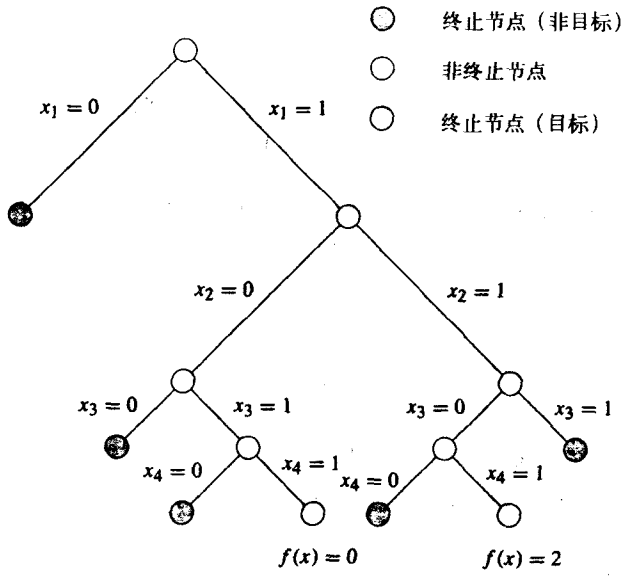


图11-2 对应于0/1整数线性规划问题的图

在一个变量实例化为0或1后，就可以检查在剩余自由变量的实例中是否可能导致可行解。我们可以用下面的条件进行判断:

$$\sum_{x_j \text{ 是自由的}} \max\{A[i,j],0\} + \sum_{x_j \text{ 是固定的}} A[i,j]x_j > b_i, i = 1,\ldots,m \qquad (11\text{-}1)$$

通过把自由变量实例化为0或1，方程11-1的左端是 $\sum_{k=1}^{n} A[i,k]x_k$ 能取到的最大值。若这个值大于或等于$b_i (i = 1, 2, \cdots, m)$，那么节点可导致一个可行解。

对于对应于$x_1 = 0$和$x_1 = 1$的各个节点，选定变量x_2并对其赋值。然后检查节点是否能得到可行解。不断重复这个过程，直到所有变量都被赋值，这样，就得到问题的可行解集。图11-2展示这个过程。

对每一个可行解均计算它的$f(x)$，其中函数值最小的就是我们要找的解。注意，为了确定最终解，没有必要产生全部可行解。有几种搜索算法仅搜索图的一部分就可确定最优解。 ■ 472

在求解某些问题时，我们可以估算出从某个中间状态到目标状态的成本。这个成本称为启发式估计（heuristic estimate)。令函数$h(x)$表示从状态x到目标状态的启发式估计，函数$g(x)$表示从初始状态s沿当前路径到达状态x的成本。函数h称为启发式函数（heuristic function)。若在所有状态x中，$h(x)$是从x到目标状态的成本的下界，则称h为容许的（admissible)。定义

函数$l(x) = h(x) + g(x)$。若h是容许的，则$l(x)$就是到达一个目标状态的路径成本的下界，这个目标状态可以通过扩展s与x之间的当前路径获得。在下面的几个例子中，我们将利用容许启发式确定从初始状态到达目标状态的最小成本移动序列。

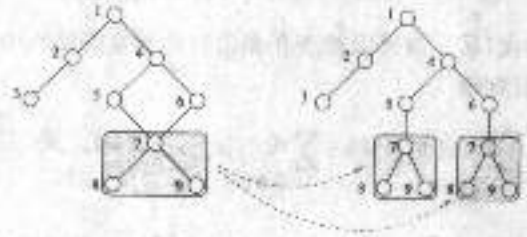
例11.4 九宫重排问题的容许启发式函数

设在九宫重排问题格子中的每个位置用一个数字对表示。数字对(1, 1)表示左上角的位置，数字对(3, 3)表示右下角的位置。从(i, j)到(k, l)的距离定义成$|i-k|+|j-l|$，这个距离称为曼哈坦距离（Manhattan distance）。所有的滑块从初始位置移动到最终位置的曼哈坦距离之和就是从当前排列状态转变到最终排列状态移动次数的一种估计，这种估计称为曼哈坦启发式（Manhattan heuristic）。注意，若$h(x)$是排列状态x到最终排列状态之间的曼哈坦距离，则$h(x)$也是从排列状态x到最终排列状态的移动次数的下界。因此，曼哈坦启发式是容许的。 ■

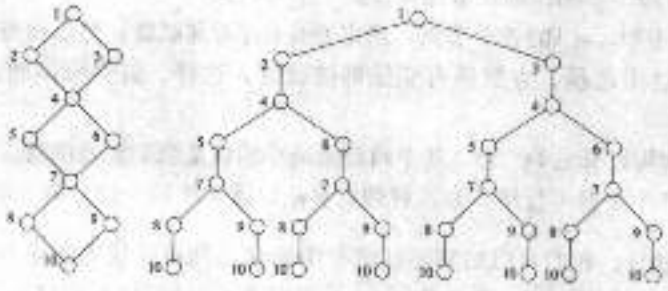
一旦将DOP表示成一个图的搜索问题，我们就可利用分支定界搜索和启发式搜索等算法求解。这些算法借助启发式和搜索空间结构求解DOP，无需完全搜索集合S。

DOP属于NP难题一类问题。可能有人认为，采用并行算法处理这类问题没有意义，因为在最坏状态下，必须使用指数级数量的处理器，才能将时间复杂度降低到多项式级。然而，对大多数问题来说，启发式搜索算法的时间复杂度是多项式级的。此外，针对某些特定的问题，有些启发式搜索算法找出次优解的时间复杂度也是多项式级。在这种情况下，我们可以使用并行计算机求解较大的问题。许多DOP需要实时处理，如机器人动作规划、语音理解和任务调度等。在这些应用中，只有采用并行处理，性能才能达到要求。在其他追求最优解的问题中，采用并行搜索技术，我们可以在合理的时间内求解中等规模的DOP实例（例如VLSI芯片布置图优化和计算机辅助设计）。

473



a)



b)

图11-3 将图展开成树的两个例子

11.2 顺序搜索算法

应用于一个状态空间的最合适的顺序搜索算法，依赖于空间能否构成图或树。在树中，每个新的后继都通向搜索空间中未曾到达的部分。0/1整数线性规划问题就是一典型例子。然而，对图而言，沿不同路径可能到达同一个状态。九宫问题就是这样的例子。处理这类问题，无论何时产生一新状态，都要检查该状态是否已经产生。若不作检查，高效的搜索图将展开成树，一个状态将在每一条通向它的路径中重复出现（见图11-3）。

在许多问题中（如九宫问题），展开只是稍微增加了搜索空间的大小。然而，在某些问题中，展开的图可能比原来的图大得多。图 11-3b显示一个图及其展开树，该树拥有指数级的状态数目。在这一节里，我们将概述求解不同的DOP的顺序搜索算法，这些DOP均可表达成图或树的搜索问题。

11.2.1 深度优先搜索算法

若DOP可表达成树的搜索问题，就可以采用*深度优先搜索*（depth-first search, DFS）算法 474
求解DOP。DFS首先扩展根节点并产生一后继。在每个后继步中，DFS都扩展新生成的节点中的一个。若节点没有后继（或从节点不能导致任何解），则DFS回溯，扩展另外一个节点。在某些DFS算法中，按后继的启发式值的大小顺序，依次扩展节点的后继。深度优先算法的存储需求与当前搜索的状态空间深度成线性关系，这是它的一主要优点。下面讨论基于深度优先搜索的三种算法。

1. 简单回溯

简单回溯（simple backtracking）是一种深度优先搜索方法，当找到第一个解时停止搜索。因此，它不能保证找到最小成本的解。简单回溯不用启发式信息对扩展节点的后继排序。*有序回溯*（ordered backtracking）是一种变形，采用启发式对扩展节点的后继排序。

2. 深度优先分支定界

深度优先分支定界（depth-first branch-and-bound, DFBB）搜索整个状态空间。当找到一条解路径后，再继续查找下一条。若找到一条新解路径，及时更新当前最好解路径。DFBB抛弃局部较差的路径（即是展开后生成的路径肯定比当前最佳解路径差的部分解路径）。若当前最佳解是全局最优解时，搜索终止。

3. 迭代加深A*

对应于DOP的树可能非常深。因此，当一个解存在于另一个更高的分支时，搜索算法可能卡在搜索空间某个深的部分。对于这种树，我们可以设定深度优先算法搜索的一个深度限制。如果某个被扩展的节点超过限制，节点将不再扩展，算法回溯。若最终没有找到解，则重设一个更大的限制值，重新搜索整个状态空间。这个技术称为*迭代加深深度优先搜索*（iterative deepening depth-first search, ID-DFS），注意这个方法只能保证找到边最少的解路径，而不能保证找到成本最小的路径。

*迭代加深A**（IDA*）是ID-DFS的变形。IDA*采用节点的l值限制深度（回顾11-1节，对于节点x，$l(x) = g(x) + h(x)$）。IDA*在搜索空间内重复进行成本受限的深度优先搜索。在每次迭代中，IDA*都对节点进行深度优先扩展。如果要扩展节点的l值高于成本限制值，算法立即

回溯。如果在当前的限制下，没有找到解，则设置更大的成本限制值，重复上述的深度优先搜索过程。在首次迭代中，限制值设为初始状态s的f值。注意由于$g(s)$为0，$f(s)$等于$h(s)$。在
475 后面的每次迭代中，成本限制值逐步增大，新的成本限制值设为所有节点中的最小f值，这些节点是在以前迭代中产生但尚未扩展的节点。当扩展到目标节点时，算法终止。如果启发式函数容许的，则IDA*能保证找出最优解。显然，在迭代过程中，IDA*执行很多多余的操作。不过，对许多问题来说，IDA*执行的多余操作是最少的，因为大部分工作是在较深的搜索空间里完成的。

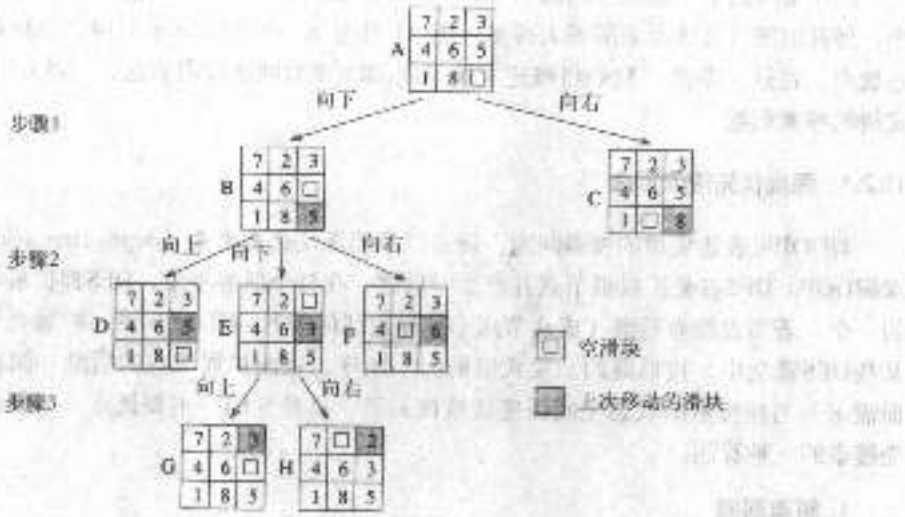


图11-4 九宫问题中采用深度优先搜索方法的前三步状态

例11.5 深度优先搜索：九宫重排问题

图11-4显示采用深度优先算法求解九宫问题的过程。搜索从初始布局开始。这个状态的后继由滑块的可能的移动方式产生。在搜索算法的每步中，选择一新状态，同时产生它的后继。DFS算法扩展树中最深的节点。在第一步，初始状态A产生状态B和C。根据预先定义的准则，选择其中一个状态。本例中，我们根据下述移动方式对后继排序：上、下、左、右。第二步，在DFS算法中选择状态B，并产生状态D、E、F。注意状态D可以丢弃，因为它是状态B的父节点的复制。在第三步中，我们选择状态E进行扩展，产生状态G和H。状态G又可丢
476 弃，因为它是B的复制。用这种方法进行搜索，直到算法回溯或者产生最终的排列状态。 ■

在DFS算法的每步中，必须存储未经处理的状态。例如，在九宫问题中，每步可能要存储最多三个未处理的状态。一般情况下，如果m是存储一个状态所需存储单元量，d是最大深度，那么DFS所需的总的存储量为$O(md)$。采用并行DFS算法搜索的状态空间树可用栈来表示。由于栈的深度随树的深度线性增长，栈对内存的要求很低。

借助栈，有两种方法可以存储未经处理的状态。一种方法是，在每步中，将未处理的状态压入到栈中。一个状态的祖先不会在栈中出现。图11-5b显示图11-5a中所示树的表示。另一种方法，如图11-5c所示，未处理状态同它们的父状态存储在一起。如果从初始状态到目标状态的转换序列需要作为解的一部分，则必须采用第二种存储方法。再者，如果状态空间是这

样的图，其中通过应用到当前状态的一个变换序列可以产生一个祖先状态。那么就要采用第二种存储方式，因为它允许我们检查祖先状态的重复，从而在状态空间中消除任何圈。第二种方法对求解九宫问题这样的问题是很有用的。在例11.5中，使用第二种存储方法，使算法能够检测到节点D和G应被舍弃。 477

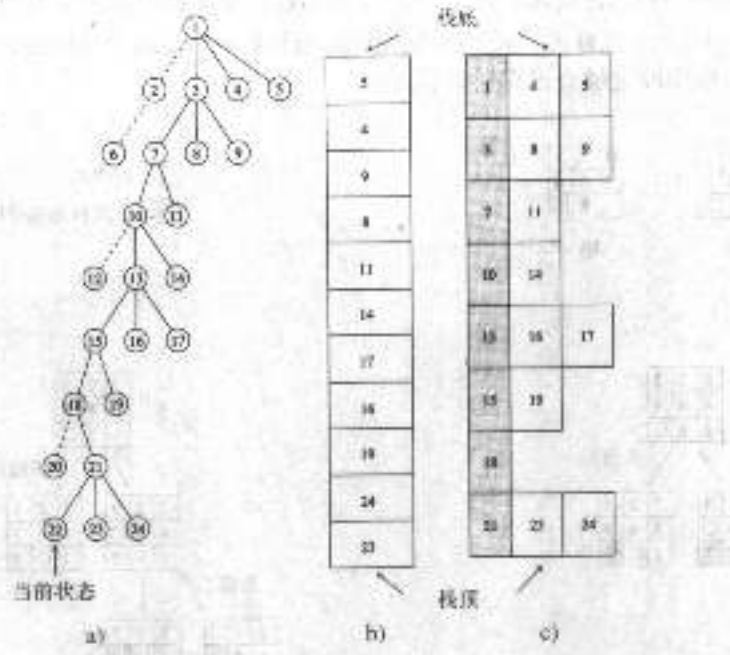


图11-5 DFS树的表示

注：a）DFS树，树中虚线表示的后继节点已搜索过。b）仅存放未搜索节点的堆栈。c）存放未搜索节点及其父节点的堆栈。阴影部分表示父节点状态，其右边的块表示未搜索过的后继节点状态。

11.2.2 最佳优先搜索算法

最佳优先搜索(BFS)算法也可以搜索图和树，这些算法借助启发式直接搜索空间中最有可能产生解的那些部分。对最有希望得到解的节点赋予较小的启发式值。BFS维护两个表：开放表与封闭表。开始时，初始节点放进开放表。启发式评价函数估算各个节点产生解的可能性，并对开放表中的节点按可能性大小排序。在搜索过程中的每步，移出开放表中最有可能产生解的节点。若这个节点是目标节点，算法终止。否则，扩展节点。扩展的节点进入封闭表。若新扩展节点的后继节点满足以下条件之一，则进入开放表：1）后继节点不在开放表与封闭表中，2）后继节点在开放表或者封闭表中，但具有较小的启发式值。对于第二种情况，具有较高启发式值的节点被删除。

A*算法是一种通用的BFS算法。A*算法利用下界函数l作为启发式评价函数（回忆11.1节，对于每个节点x，$l(x) = g(x) + h(x)$）。开放表中的节点按照l函数的值排序。每步移出l值最小的节点（即最佳节点），并对其扩展。该节点的后继插入开放表的适当位置。同时，节点本身插入封闭表。对于一个容许的启发式函数，A*算法能找出最优解。

BFS算法的主要缺点是它对内存的要求随搜索空间的大小线性增长。对许多问题而言，搜

索空间的大小随树深度呈指数级增长。因此，对于搜索空间大的问题，巨大的内存需求限制了算法的应用。

例11.6 最佳优先搜索：九宫问题

考察例11.2和11.4的九宫问题。图 11-6展示采用最佳优先搜索求解九宫问题的4个步骤。每步中，选择值 $f(x) = g(x) + h(x)$ 最小的状态x进行扩展。以前产生的所有节点均在开放表或封闭表中，因此BFS能够查出重复的节点。 ■

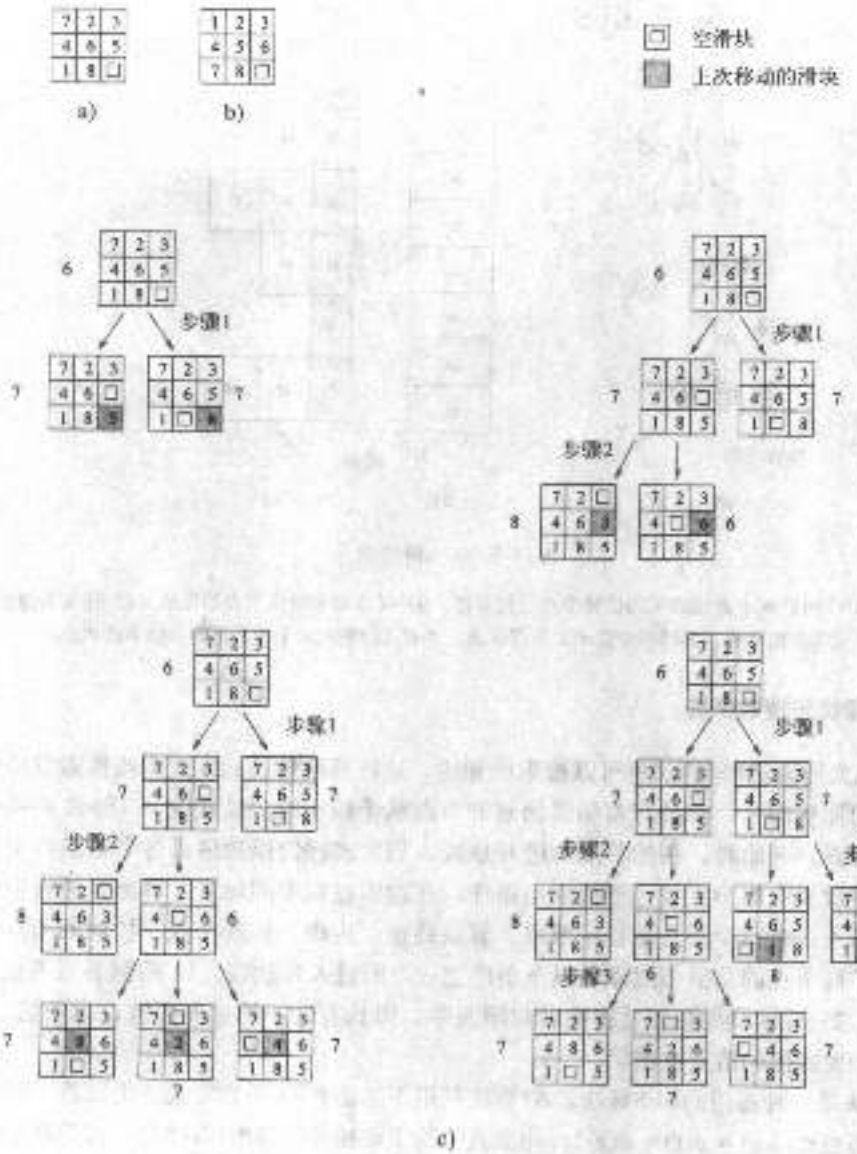


图11-6 用最佳优先搜索求解九宫问题：a）初始排列状态；b）目标排列状态；c）最佳优先搜索求解的前4步产生的状态。每个状态标以相应的h值（也就是从当前状态到目标状态的曼哈顿距离）

11.3 搜索开销因子

478 ~ 479

并行搜索算法的开销来自几个方面。这些开销包括通信开销、负载不平衡引起的空闲时间以及对共享数据结构的争用。因此，如果某个算法的串行及并行形式完成同样数量的工作，在p个处理器进行并行搜索的加速比要小于p。然而，由于串行与并行形式可能对搜索空间不同部分搜索，它们的工作量一般也不相同。

令W是由单处理器完成的工作量，W_p是用p个处理器所完成的总工作量。并行系统的*搜索开销因子*（search overhead factor）是并行形式所做工作与串行形式完成工作量的比率，即W_p/W。因此，并行系统的加速比上界为$p \times (W_p/W)$。不过，由于在并行处理中还存在其他开销，实际的加速比可能小于该值。在大多数并行搜索算法中，搜索开销因子都大于1。在某些情况下，也可能小于1，导致超线性加速比。若平均搜索开销因子小于1，说明串行搜索算法并不是求解问题的最快算法。

为使叙述和分析简化，假设扩展每个节点所花费的时间是相同的，W和W_p分别为串行和并行形式扩展的节点数目。若每次扩展花费时间为t_c，那么串行运行时间为$T_S = t_c W$。在本章余下的部分，我们设$t_c = 1$。这样，问题规模W与串行运行时间T_S是相同的。

11.4 并行深度优先搜索

我们现在开始讨论并行深度优先搜索，重点是搜索中的简单回溯。深度优先分支定界以及IDA*的并行形式类似于本节讨论的并行形式，它们将在11.4.6和11.4.7节中讨论。

并行深度优先搜索算法中的关键问题是处理器之间的搜索空间的分配。我们来看图11-7所示的树。注意，树的左子树（以节点A为根）能和树的右子树（以节点B为根）并行搜索。通过静态分配树中的一个节点到一个处理器，可能扩展根在那个节点的整个子树，且不需要和另一个处理器通信。因此，看起来这种静态分配处理器产生一个好的并行搜索算法。

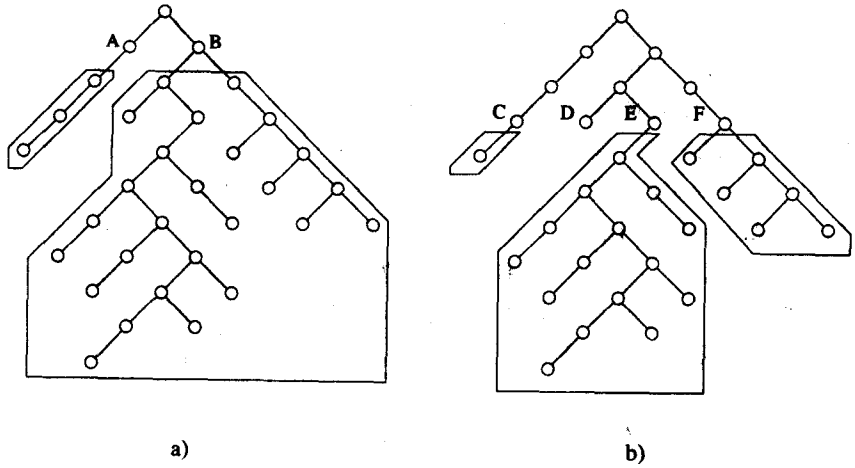


图11-7 搜索树的非结构性质以及由于静态划分引起的负载不平衡

现在，假如我们试图把这个方法应用到图11-7所示的树上，看看会发生什么情况。假定有两个处理器。根节点被扩展产生两个节点（A和B），每一个节点被分配到其中一个处理器。现在，每一个处理器各自搜索以这个节点为根节点的子树。到这一步，静态节点分配问题是

明显的。搜索以节点A为根节点的子树的处理器明显比另一个处理器扩展更少节点。由于这种负载不平衡，一个处理器在大量时间是空闲的，使效率降低。使用更多数量的处理器会加重负载不平衡。考虑把树划分并分配到4个处理器。扩展节点A和B产生节点C、D、E和F。假定
480 将每一个节点分配到一个处理器。现在，搜索以节点E为根节点的子树的处理器做的工作最多，搜索以节点C和D为根节点的子树却花费了很多空闲时间。由于以不同的节点为根节点时，搜索空间大小划分的变化很大，非结构树的静态划分会使得性能低下。而且，通常因为搜索空间动态产生，所以很难预先估计搜索空间的大小。因此，必须在所有处理器的搜索空间之间动态平衡负载。

在*动态负载平衡*（dynamic load balancing）中，当一个处理器完成工作时，它从另一个已有工作的处理器获得新的工作。考虑图11-7a中树的两个处理器划分。假定节点A和B被分配到像我们刚刚描述的两个处理器上。在这种情况下，当处理器搜索以节点A为根节点的子树而超负荷时，它需要另一个处理器帮助工作。尽管动态分配工作会导致因任务需求和任务转移而造成的通信开销，但是它减少处理器间的负载不平衡。本节介绍几种方案动态平衡处理器间的负载。

基于动态负载平衡的DFS并行形式如下。每个处理器对搜索空间的不相交部分执行DFS。当处理器结束自己搜索空间的搜索时，它请求另外处理器没有搜索的部分。这在消息传递体系结构以任务请求和消息响应的形式出现，在共享地址空间计算机中以锁定和提取任务的形式出现。无论何时，当处理器发现目标节点时，所有的处理器终结。假如搜索空间是有限的，并且任务没有解，那么所有的处理器最终都超负荷工作，算法终止。

由于每个处理器以深度优先搜索状态空间，没有搜索到的状态空间能被方便地作为堆栈
481 存储。每个处理器维持自己的本地堆栈，并在栈上执行DFS。当处理器的本地堆栈为空时，它请求（经过显式消息或通过锁定）另一个处理器堆栈的未经处理的状态。起初，全部搜索空间被分配到一个处理器，另外的处理器分配的是空的搜索空间（也就是空的堆栈）。当处理器请求任务时，搜索空间被分配到这些处理器上。我们把发送任务的处理器叫作*施主*（donor）处理器，而把请求和接收任务的处理器作为*接受者*（recipient）处理器。

如图11-8所示，每一个处理器可能处于两种状态之一：*活跃状态*（也就是有任务）或空闲状态（也就是试图获得任务）。在消息传递体系结构中，空闲处理器选择施主处理器并且发出任务请求给它。假如空闲处理器从施主处理器接收任务（搜索状态空间的一部分），那么它就变成活跃状态。假如它接收到*拒绝*消息（由于施主没有任务），它会选择另一个施主并且发出一个任务请求到那个施主。这个过程将重复直到处理器得到任务或者所有的处理器都变成空闲状态。当处理器是空闲的并且它接收到任务请求，那么这个处理器返回*拒绝*消息。通过锁定另外的处理器的堆栈，检查处理器是否有任务，提取任务并且释放堆栈，能在共享地址空间计算机上实现同样的过程。

在消息传递体系结构中，在活跃状态下，一个处理器执行固定量的任务（扩展固定量的节点），然后检查挂起的任务请求。当它接接收到任务请求时，处理器把它的任务分成两部分，并且发送一部分给提出请求的处理器。当处理器处理完自己的搜索空间时，它就变为空闲的。这个过程继续直到找到了解或所有的搜索空间都被搜索完才停止。如果找到了解，那么把消息广播到所有的处理器以停止搜索。终止检测算法用来检查是否所有的处理器都没有找到解而变成空闲的（11.4.4节）。

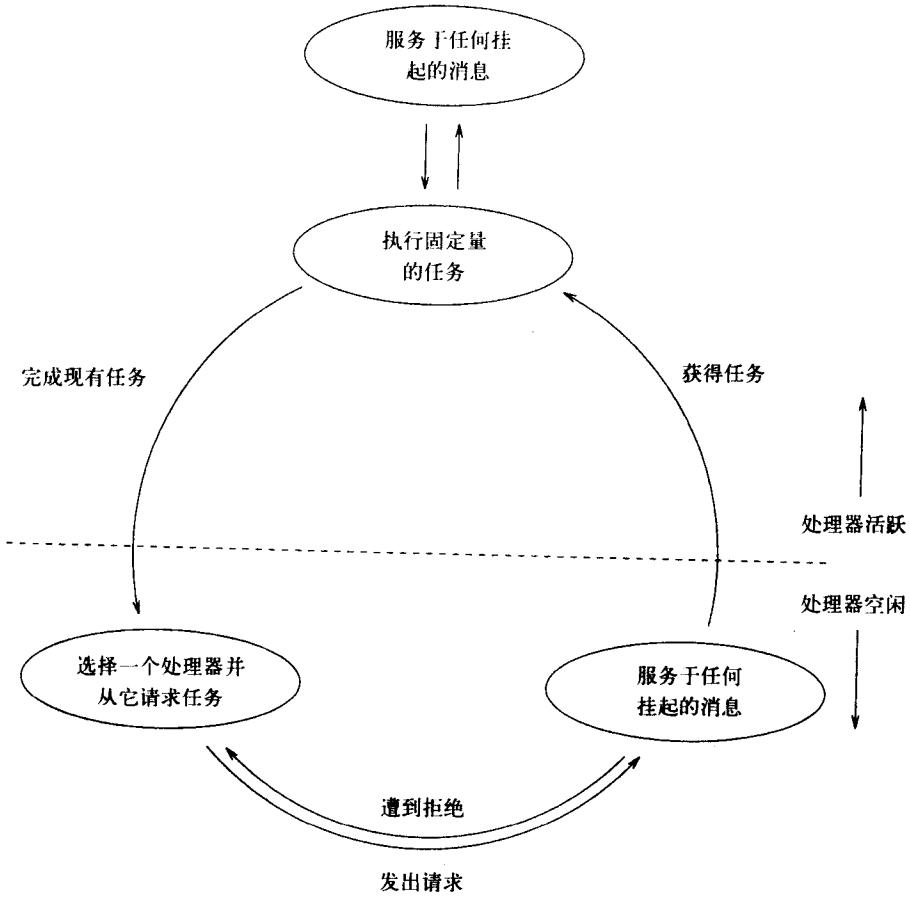


图11-8 动态负载平衡的一般方案

11.4.1 并行DFS的重要参数

并行DFS的两个特性在决定其性能上起着关键作用。第一个是在处理器上分配任务的方法，第二个是当处理器成为空闲时决定施主处理器的方案。

1. 任务划分策略

当任务转移时，施主的堆栈被分成两个堆栈，其中一个发送给接收者。换言之，一些节点（两部分中的一部分）从施主的堆栈中除去而加到接收者的堆栈中。假如发送给接收者的任务太小，那么接收者很快就会变空闲；假如太多，施主很快就会变空闲。理想情况下，堆栈被分成两个相等的部分。这样，每个堆栈代表的搜索空间大小是一样的。这种划法称为*半分*（half-split）。可是，对于根节点在未扩展部分的堆栈中的树来说，很难估计这种树的大小。然而，栈底部分（也就是接近于初始的节点的部分）趋向于有更大的以它们为根节点的树，栈顶部分则趋向于有更小的以它们为根节点的树。为了避免发送很小的任务量，当节点超过某一特定栈深度时，任务不发送。这个深度被称之为*截止深度*（cutoff depth）。 482

另外一些划分搜索空间的策略是 1) 发送栈底附近的节点，2) 发送截止深度附近的节点，3) 发送栈底至截止深度间的一半节点。选择哪种划分策略依赖于搜索空间的性质。假如搜索

空间是一致的，则策略1和策略3都很适用。假如搜索空间高度不规则，策略3常常是适用的。假如能使用强启发式（排序后继者使目标节点能移到状态空间树的左边），那么策略2将实现得更好，因为它力求分配搜索空间中可能包含一个解的那些部分。假如栈很深，那么划分成本也是很重要的。对于这种栈来说，策略1比策略2和3的成本更低。

483 图11-9显示，使用策略3将图11-5a的DFS树划分成两棵子树。注意，超过截止深度的状态没有划分。图11-9也显示相应于两棵子树的栈表示。图中使用的栈表示仅存储没有探询的部分。

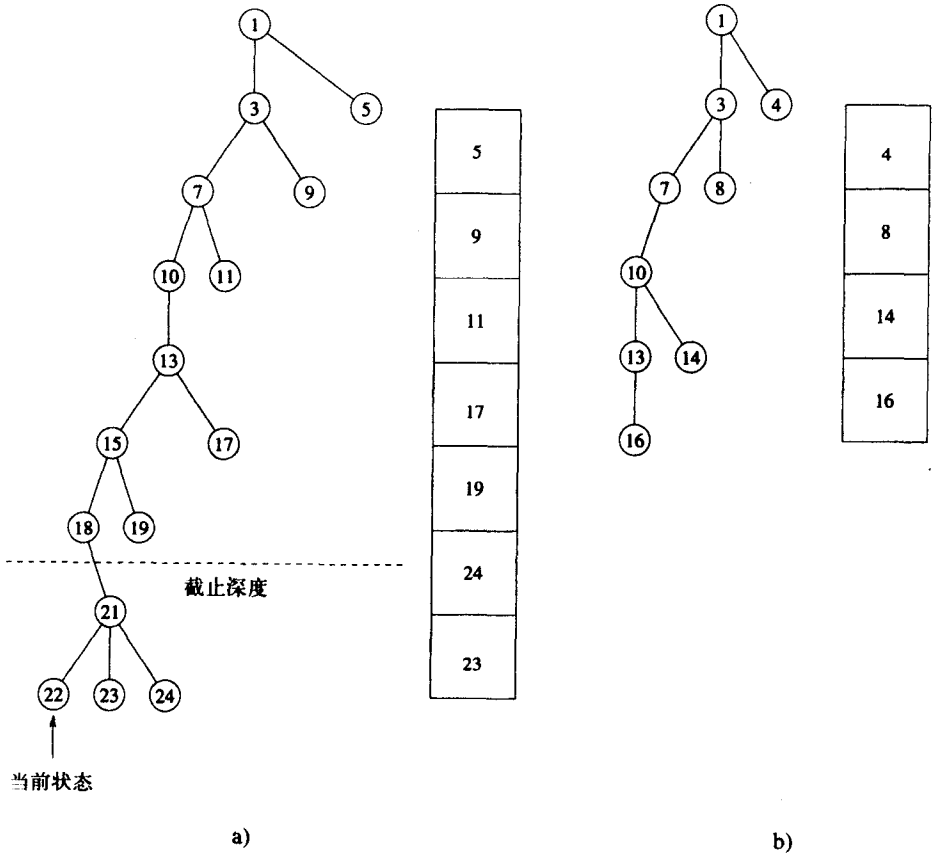


图11-9 图11-5中DFS树的划分，两棵子树及其栈表示如a）和b）所示

2. 负载平衡方案

这一部分讨论三种动态负载平衡方案：异步循环、全局循环和随机轮询。每种方案都能对消息传递和共享地址空间计算机编码。

异步循环 在异步循环(ARR)中，每个处理器有一个独立的变量*target*。无论何时处理器超负荷工作，它使用*target*变量作为施主处理器的标记并且试图从它那里得到任务。*target*变
484 量的值在每次请求的任务被送出时增加1（以*p*为模）。每个处理器最初的*target*变量被设置成(*label* + 1)（以*p*为模），*label*是本地处理器的标记。注意，任务请求被每个处理器独立产生。然而，有可能两个或更多的处理器从同一个施主在几乎同一时间请求任务。

全局循环 全局循环(GRR)使用一个称为*target*的全局变量。这个变量能存储在共享地址

空间计算机的全局可访问空间中，或者在消息传递计算机的指定处理器上。无论何时，当处理器需要任务时，它请求并且接收*target*值，要么通过锁定、读取或者在共享地址空间计算机解锁实现，要么通过发送消息请求指定的处理器（如P_0）实现。*target*值在响应下一个请求之前增加1（以p为模）。接收者处理器然后试图从那些标记值为*target*的施主处理器得到任务。GRR确保相继的任务请求被均匀地分配到所有的处理器上。这种方案的缺陷是访问*target*时的争用。

随机轮询 随机轮询(RP)是最简单的负载平衡方案。当一个处理器成为空闲时，它随机选择一个施主处理器。每个处理器被选作成为施主的概率是相等的，确保任务请求被均匀分配。

11.4.2 并行DFS分析的一般框架

为了分析并行DFS算法在任意负载平衡方案中的性能和可扩展性，我们必须计算算法的开销T_o。任何负载平衡方案中的开销是由于通信（请求和发送任务）、空闲时间（等待任务）、终止检测以及争用共享资源而产生的。假如搜索开销因子大于1（即假如并行搜索比串行搜索做更多工作），这将会对T_o增加另外一项。本节我们假设搜索开销因子为1，即算法的串行和并行版本完成同样数量的计算。我们在11.6.1节中分析搜索开销因子大于1的情况。

对于11.4.1节中讨论的负载平衡方案，空闲时间被包含在由任务请求和传送引起的通信开销中。当处理器成为空闲，它立即选择一个施主处理器并且对它发送一个任务请求。处理器剩余空闲的总时间等于请求到达施主处理器加上接收处理器响应到达的时间。在那个时刻，空闲处理器要么变成忙的，要么产生另一个任务请求。因此，花在通信开销上的时间包括处理器空闲的时间。由于通信开销是并行DFS上的主要开销，我们现在考虑一种计算每个负载平衡方案通信开销的方法。

要想得到DFS负载平衡方案中通信开销的精确表达式是很困难的，因为这个过程是动态的。本小节陈述一种方法，用来求得这个开销的上界。在分析中我们作以下假设： 485

1）一旦某一处理器的任务大小超过一个阈值ε，这个任务就被划分成一些独立的部分。

2）有一个合理的任务划分机制。假定一个处理器上的任务w被分成两部分：ψw和$(1-\psi)w$，且$0 \leqslant \psi \leqslant 1$。这时存在一个任意小的常量$\alpha(0 < \alpha \leqslant 0.5)$，使得$\psi w > \alpha w$和$(1-\psi)w > \alpha w$。我们把这样一种划分机制称为$\alpha$-划分。常量$\alpha$对任务划分引起的不平衡负载设置一个下界：$w$的两部分划分至少有$\alpha w$的任务量。

大多深度优先搜索算法满足以上第一个假设。11.4.1节中描述的第三种任务划分策略，即使对高度不规则搜索空间也能导致平均α-划分。

在被分析的负载平衡方案中，总的任务在处理器之间动态划分。处理器独立地工作在搜索空间的不相连部分。空闲处理器轮询请求任务。当它找到一个有任务的施主处理器时，这个任务就被划分，并且其中一部分任务被传送到空闲处理器。若施主有任务w_i，并且它被划分成大小为w_j和w_k的两部分，那么假设条件2说明存在常量α，使得$w_j > \alpha w_i$和$w_k > \alpha w_i$。注意α小于0.5。因此，在任务传送之后，所有处理器（施主处理器和接收者处理器）的任务量多于$(1-\alpha)w_i$。假设有任务的p部分，其大小分别为$w_0, w_1, \ldots, w_{p-1}$。假定最大部分的任务量为$w$。如果所有这些任务部分是分隔开的，划分策略就产生$2p$个任务，其大小分别是$\psi_0 w_0$，$\psi_1 w_1$，…，$\psi_{p-1} w_{p-1}, (1-\psi_0)w_0, (1-\psi_1)w_1, \ldots, (1-\psi_{p-1})w_{p-1}$。在它们之中，最大部分任务的大小是$(1-\alpha)w$。

假设有p个处理器，每个处理器分配一个任务。如果每个处理器至少接收到一次任务请求，那么这p块任务的每一部分都被至少划分一次。这样，任一处理器上的最大任务量至多减少$(1-\alpha)$倍。我们定义这样一个$V(p)$，在每$V(p)$次任务请求后，每一个处理器都至少接收到一次任务请求。注意$V(p) \geqslant p$。一般说来，$V(p)$取决于负载平衡算法。开始时，处理器P_0有W单位的任务，而所有其他的处理器都没有任务。在$V(p)$次请求后，任一个处理器上的留下最大任务量减少到不足$(1-\alpha)W$；在$2V(p)$次请求后，任一个处理器上留下的最大任务量少于$(1-\alpha)^2W$。类似地，在$(\log_{1/(1-\alpha)}(W/\varepsilon))V(p)$次请求后，任一个处理器上留下的最大任务量低于阈值$\varepsilon$。因此，总的任务请求量是$O(V(p)\log W)$。

通信开销由任务请求和任务传送引起。总的任务传送量不能超过总的任务请求量。所以，任务请求的总数，加上一个任务请求和相应任务传送的总通信成本的权值，给出总通信开销
486 的一个上界。为简单起见，我们假设与任务请求及任务传送有关的数据是常量。一般说来，栈的大小应与搜索空间的大小按对数关系增长。对于这种情况，我们可作类似的分析（习题11.3）。

如果t_{comm}是一个任务通信所需的时间，那么通信开销T_o为

$$T_o = t_{comm} V(p) \log W \tag{11-2}$$

相应的效率E如下：

$$\begin{aligned} E &= \frac{1}{1 + T_o/W} \\ &= \frac{1}{1 + (t_{comm} V(p) \log W)/W} \end{aligned}$$

我们曾在5.4.2节指出，通过平衡问题大小W和开销函数T_o能推导等效率函数。如公式(11-2)所示，T_o取决于两个值t_{comm}和$V(p)$。t_{comm}的值由底层体系结构决定，而函数$V(p)$由负载平衡方案决定。在下面几小节中，我们将对11.4.1节中介绍的每种方案导出$V(p)$的值。然后使用这些$V(p)$值，导出在消息传递和共享地址空间计算机上各种方案的可扩展性。

各种负载平衡方案上的V(p)值的计算

公式(11-2)表明，$V(p)$在总通信开销中是一个重要的部分。在此我们将计算不同负载平衡方案的$V(p)$值。

异步循环 当所有的处理器在同一时间向同一个处理器发出任务请求时，ARR的$V(p)$值出现最坏情况。对这种情况说明如下。假设所有的任务都在处理器$p-1$上，并且所有其他的处理器（0到$p-2$）的本地计数器都指向处理器0。在这种情况下，当处理器$p-1$接收到一个任务请求时，一个处理器必须发出$p-1$个请求，而余下的$p-2$个处理器中每个产生多至$p-2$个任务请求（对除了处理器$p-1$和它自身外的所有处理器）。这样，$V(p)$有一个$(p-1) + (p-2)(p-2)$上界。也就是说，$V(p) = O(P^2)$。注意，$V(p)$的实际值在p和p^2之间。

全局循环 在全局循环中，所有的处理器按顺序接收请求。在p个请求之后，每个处理器接收到一个请求。因此，$V(p)$等于p。

随机轮询 对于随机轮询，最坏情况是$V(p)$的值没有界限。因此，我们计算$V(p)$的平均
487 值。

考虑p个盒子的一个集合。在每次试验中，随机选一个盒子并做上标记。我们对标记所有

盒子所需试验次数的平均值感兴趣。在我们的算法中，每次试验对应于一个处理器向另外一个被随机选定的处理器发送任务请求。

我们用$F(i, p)$代表p个盒子中i个为已作标记而另外的$p-i$个盒子未作标记的状态。由于下一个被标记的盒子是随机的，选出盒子已有标记的概率是i/p，没有标记的概率是$(p-i)/p$。因此，系统停留在$F(i, p)$状态的概率为i/p，而转移到$F(i+1, p)$状态的概率为$p-i/p$。我们用$f(i, p)$表示从$F(i, p)$状态转移到$F(p, p)$状态所需的平均试验次数。这样，$V(p) = f(0, p)$。我们有

$$\begin{aligned} f(i,p) &= \frac{i}{p}(1+f(i,p)) + \frac{p-i}{p}(1+f(i+1,p)) \\ \frac{p-i}{p}f(i,p) &= 1+\frac{p-i}{p}f(i+1,p) \\ f(i,p) &= \frac{p}{p-i}+f(i+1,p) \end{aligned}$$

因此

$$\begin{aligned} f(0,p) &= p\times\sum_{i=0}^{p-1}\frac{1}{p-i} \\ &= p\times\sum_{i=1}^{p}\frac{1}{i} \\ &= p\times H_p \end{aligned}$$

其中H_p是一个调和数。这表明当p很大时，$H_p \approx 1.69\ln p$（$\ln p$代表p的自然对数）。因此，$V(p) = O(p \log p)$。

11.4.3 负载平衡方案分析

本节分析在11.4.1节中引入的负载平衡方案的性能。对于每种情况，我们假设任务以固定大小的消息传送（放宽这种假定的影响在习题11.3中考查）。

回想一下，成本模型中m字的消息通信成本的简化形式是$t_{comm} = t_s + t_w m$。由于假设消息大小m为一个常量，如果在互连网络上没有拥塞，则$t_{comm} = O(1)$。通信开销T_o（公式(11-2)）简化为

$$T_o = O(V(p)\log W) \tag{11-3}$$

我们对每一种负载平衡方案用大小为W的问题平衡开销，推导由于通信产生的等效率函数。

488

异步循环 如11.4.2节的讨论，异步循环的$V(p)$为$O(p^2)$。代入公式(11-3)，得到的通信开销T_o为$O(p^2 \log W)$。对大小为W的问题平衡通信开销，可得

$$W = O(p^2 \log W)$$

把W代入相同方程的右端且简化得

$$\begin{aligned} W &= O(p^2\log(p^2\log W)) \\ &= O(p^2\log p + p^2\log\log W) \end{aligned}$$

重对数项$(\log \log W)$渐近地小于第一项，只要p的增长速度不慢于$\log W$，因此可以被忽略。所以，这种方案的等效率函数为$O(p^2 \log p)$。

全局循环 从11.4.2节可知，全局循环的$V(p) = O(p)$。代到公式(11-3)中，产生$O(p \log W)$的T_o通信开销，像异步循环一样简化，可以得到这种方案由于通信开销引起的等效率函数为$O(p \log p)$。

然而，在这种方案中，全局变量*target*被重复访问，可能会引起争用。这个变量被访问的次数等于任务请求的总数$O(p \log W)$。如果处理器被有效地利用，总执行时间为$O(W/p)$。假设在p个处理器上求解大小为W问题时没有对*target*的争用，那么，W/p大于共享变量被访问的时间。随着处理器数目的增加，执行时间(W/p)下降，但是共享变量被访问的次数增加。这样，就存在一个交叉点，当超过这个点时，共享变量就成为一个瓶颈，阻碍进一步减少运行时间。通过以这样一种速度增加W，使W/p与$O(p \log W)$之比保持为常量，就可能消除这个瓶颈。这要求W关于p的增长按如下关系：

$$\frac{W}{p} = O(p \log W) \tag{11-4}$$

我们通过p表示W可简化公式(11-4)。这样得出一个等效项$O(p^2 \log p)$。

由于来源于争用的等效率函数渐近支配来源于通信的等效率函数，总的等效率函数由$O(p^2 \log p)$给出。注意，尽管很难估计由于对共享变量争用的实际开销，但我们可以确定最后的等效率函数。

随机轮询 在11.4.2节看到对随机轮询的$V(p) = O(p \log p)$。把它代入公式(11-3)，得到通信开销T_o为$O(p \log p \log W)$。使T_o与问题大小W相等并且像前面一样简化，可推导出由于通信
489 开销导致的等效率函数是$O(p \log^2 p)$。由于在随机轮询中没有争用，这个函数也是总等效率函数。

11.4.4 终止检测

并行DFS中，到目前为止还没有讲述终止检测。在这节中，我们将讲述两种终止检测方案，这两种方案能用在11.4.1节讨论的负载平衡算法中。

1. Dijkstra的令牌终止检测算法

考虑这样一种简单情况：某个处理器一旦变为空闲，就再也不会得到任务。假想p个处理器用一种逻辑环连接（注意，逻辑环结构很容易映射到物理拓扑结构上）。处理器P_0在变为空闲时初始化一个令牌，这个令牌被送给环中的下一个处理器P_1。在计算的任何阶段，假如一个处理器接收到令牌，那么令牌被保留在这个处理器上，直到分配到这个处理器上的计算完成为止。一旦计算完成，令牌又被传到环中的下一个处理器。假如处理器已经空闲，那么令牌也被传到下一个处理器。注意，在任意时刻，令牌被传到P_i时，处理器$P_0, \ldots, P_{i-1}$已经完成它们的计算任务。处理器P_{p-1}传递它的令牌给处理器P_0；当P_0接收到这个令牌时，它知道所有的处理器已完成自己的计算任务，因此算法可以终止。

这种简单方案不能应用于本章描述的搜索算法，因为当一个处理器变为空闲时，它也许从另外的处理器接收到更多的任务。这样，必须修改这个令牌终止检测方案。

在修改方案中，所有处理器也组织成一个环。任一处理器都可处于两种状态之一：*黑色*和*白色*。最初，所有处理器处于白色状态。正如前面一样，令牌按P_0，P_1，…，P_{p-1}，P_0的顺序传递。如果系统中任务传递仅仅允许从处理器P_i到P_j $(i < j)$，那么简单的终止方案依然适用。

然而，若处理器P_j发送任务给处理器P_i，那么令牌必须又沿环反向传递。在这种情况下，处理器P_j被标记为白色，因为它使得令牌再次沿环传递。处理器P_0必须通过查看自己收到的令牌，决定令牌是否又必须沿着环传递。因此，令牌本身必须有两种类型：一种是白色（或有效）令牌，当处理器P_0接收到这个令牌时，隐含算法终止；另一种类型是黑色（或无效）令牌，当处理器P_0接收到这个令牌时，它隐含令牌必须再沿环反向传递。终止算法修改后，其工作方式如下：

1）当处理器P_0变空闲时，它通过使自己变成白色并发送一个白色令牌给P_1，开始终止检测。

2）如果处理器P_j发送任务给处理器P_i，且$j > i$，那么处理器P_j变成黑色。

3）若处理器P_i已经有令牌并且P_i是空闲的，那么它传递令牌给P_{i+1}。假如P_i是黑色，那么令牌在送到P_{i+1}之前设置成黑色。假如P_i是白色，那么令牌传递不改变颜色。 490

4）P_i传递令牌给P_{i+1}后，P_i本身变为白色。

当处理器P_0收到一个白色令牌并且它自己为空闲时，算法终止。通过考虑处理器在自己已经被令牌标记后接收任务的可能性，算法能正确地检测终止。

这个算法的运行时间量级是带一个小常量的$O(p)$。当处理器数目较小时，这种方案对于总体性能而言影响不大。当处理器数目较大时，这个算法可以使负载平衡方案的总等效率函数至少为$O(p^2)$（习题11.4）。

2. 基于树的终止检测

基于树的终止检测和单个任务块的权有关。开始时，处理器P_0拥有所有任务，并且它的权为1。当它的任务被划分并且发送到另一个处理器之后，它保留一半的权，并且发送一半权给接收任务的处理器。假如P_i是接收者处理器并且w_i是它的权，那么在第一个任务被传送之后，w_0和w_i都是0.5。每次处理器上的任务被划分后，权都减半。当某个处理器完成它的计算时，它返回权给那个发送任务给它的处理器。当处理器P_0上的权w_0变成1并且处理器P_0完成它的任务时，发出终止信号。 491

例11.7 基于树的终止检测

图11-10所示是4个处理器的基于树的终止检测。开始时，处理器P_0拥有全部的权($w_0 = 1$)，并且其他处理器的权都是0($w_1 = w_2 = w_3 = 0$)。在第1步，处理器P_0将其任务划分，并且分一部分任务给处理器P_1之后，w_0和w_1都变成0.5，w_2和w_3依然是0。第2步，处理器P_1分自己的一半任务给处理器P_2，在这次任务传送之后，权w_1和w_2都变成0.25，而权w_0和w_3保持不变。在第3步，处理器P_3从处理器P_1得到任务并且所有处理器的权都变成0.25。在第4步，处理器P_2完成任务并且把它的权发送给处理器P_1。P_1的权变成0.5。当所有处理器都完成它们的任务时，权沿树向上传递直到处理器P_0上的权w_0变成1。这时，所有任务完成且发出终止信号。 ■

这个终止检测算法有一个大的缺陷。由于计算机的有限精度，权的递归减半可能使得权太小以至它变成0。在这种情况下，权将丢失而永远不会发出终止信号。使用权的倒数，可以减少这种情况的发生。如果处理器P_i有权w_i，不用权本身而用它的倒数$1/w_i$。算法的细节在习题11.5中考虑。

基于树的终止检测算法并没有改变我们考虑过的任何搜索方案的总等效率函数。这是由于这样一个事实，恰好有两次权传送与每次任务传送相关。因此，算法给通信开销增加了一

个常数因子。用渐近的说法，这种改变并没有改变等效率函数。

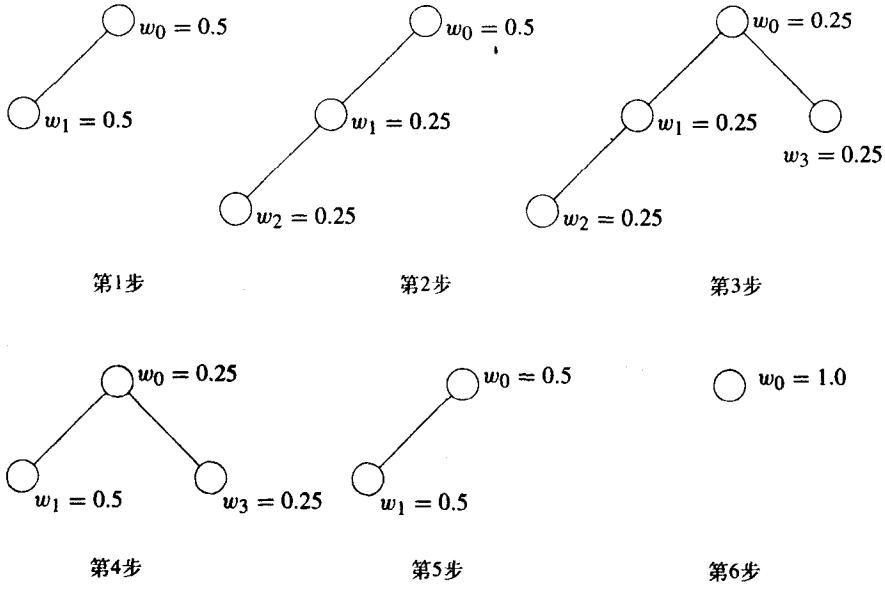


图11-10 基于树的终止检测。第1-6步说明在每次任务被传送之后，不同处理器上权的变化

11.4.5 试验结果

在这节中，我们对多种并行DFS算法说明可扩展性分析的有效性。这种可满足性问题测试布尔公式的有效性。这样的问题出现在如VLSI设计和定理证明等领域中。可满足性问题（satisfiability problem）可以表述为：给定一个布尔公式，包括多个合取范式形式的二进制变量，确定它是否是不可满足的。一个布尔公式如果不存在赋真值的变量使其为真，那么这个布尔公式是不可满足的。

Davis-Putnam算法是求解这个问题的快速而又有效的算法。它通过给一棵二叉树完成深度优先算法实现，而这棵二叉树由赋给布尔表达式的文字的真假值组成。令n是文字的个数。那么树的深度最大不能超过n。如果在文字赋值之后表达式变为假值，那么算法回溯。假如深度优先搜索没有找到一种对变量的赋值能使表达式为真，那么这个表示式是不可满足的。

492 即使表达式是不可满足的，实际上也只要搜索2^n种可能的小子集组合。例如，对于一个有65个变量的问题，总的可能组合是2^{65}（大约3.7×10^{19}），但是在具体问题实例中，大约只有10^7个节点被实际搜索。这种问题的搜索树以一种高度不一致的方式被修剪，并且任何想把树静态地划分的努力都会导致极度的负载不平衡。

可满足性问题用来测试多达1024个处理器的消息传递并行计算机的负载平衡方案。我们实现了Davis-Putnam算法，并且用到11.4.1节讨论的负载平衡算法。这个程序运行在几个不可满足的公式上。通过选择不可满足的实例，我们确保并行形式展开的节点数和串行形式展开的节点数一样；任何加速比减少都仅由负载平衡的开销引起。

在方案测试的实例问题中，树的总节点数大约在10^5到10^7之间。树的深度（它等于公式中的变量个数）在35到65之间。相对于同样问题最佳串行执行时间的加速比也被计算出来。通过给定数目的处理器并行求解所有问题的累计时间与对应的串行求解累计时间之比，计算出

了平均加速比。对于给定数目的处理器，加速比和效率在很大程度上由树的大小决定（树的大小和串行运行时间大致成正比）。这样，对于大小相似的问题，其加速比很相似。

我们在5个问题实例的样本集上测试了所有方案。表11-1表示通过不同负载平衡技术的并行算法得到的平均加速比。图11-11为得到的加速比曲线。表11-2表示RP（随机轮询）和GRR（全局循环）获得的对一个问题例程的任务请求总数。图11-12表示相应的图，并比较RP和GRR的期望值分别为$O(p \log^2 p)$和$O(p \log p)$时产生的消息数目。

表11-1 不同的负载平衡方案的平均加速比

方案	处理器数目							
	8	16	32	64	128	256	512	1024
ARR	7.506	14.936	29.664	57.721	103.738	178.92	259.372	284.425
GRR	7.384	14.734	29.291	57.729	110.754	184.828	155.051	
RP	7.524	15.000	29.814	58.857	114.645	218.255	397.585	660.582

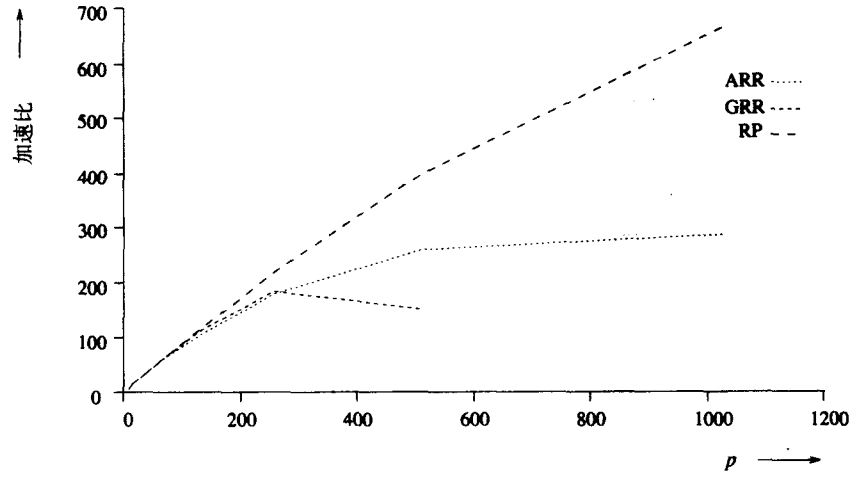


图11-11 使用ARR（异步循环）、GRR（全局循环）和RP（随机轮询）的并行负载平衡方案的加速比

表11-2 GRR（全局循环）和RP（随机轮询）产生的任务请求数

方案	处理器数目							
	8	16	32	64	128	256	512	1024
GRR	260	661	1572	3445	8557	17088	41382	72874
RP	562	2013	5106	15060	46056	136457	382695	885872

GRR的等效率函数是$O(p^2 \log p)$，它比RP的等效率函数差很多。这一点反映在实现的性能中。从图11-11可知，GRR的性能在处理器的数目超过256个时迅速变差。在$p > 256$时，仅仅在非常大的问题实例中才能获得好的加速比。试验结果同时表明ARR的可扩展性比GRR好，但是远没有RP的可扩展性好。尽管ARR和GRR的等效率函数都是$O(p^2 \log p)$，但是ARR的性能胜过GRR。其原因是，$p^2 \log p$是一个上界，是使用$V(p) = O(p^2)$推出的。这个$V(p)$值对ARR只是一个宽松的上界。相反，$V(p)$值对GRR是一个紧密的上界。

493 ~ 494

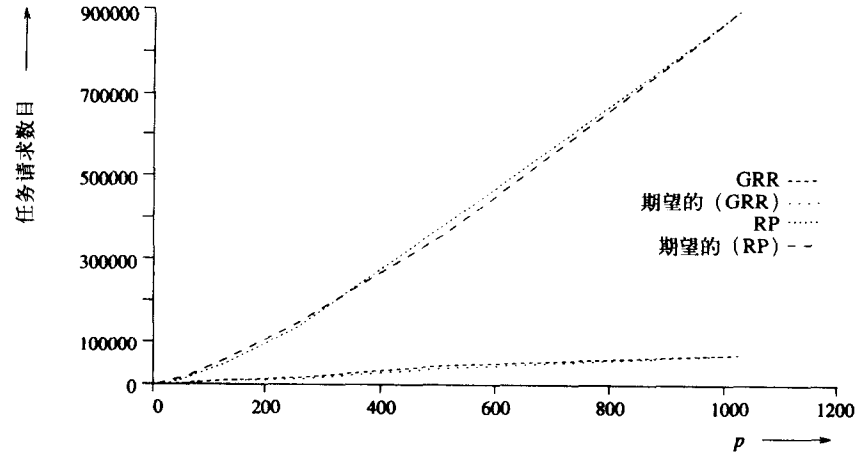


图11-12 RP（随机轮询）和GRR（全局循环）产生的任务请求数目以及它们的期望值（分别是$O(p \log^2 p)$和$O(p \log p)$）

为了确定各种方案等效率函数的精确性，我们用实验方法检验RP方法（选择这种方法是随意的）的等效曲线，我们选取了30个大小从10^5个节点到10^7个节点的不同问题实例，在不同数目的处理器上运行，并计算出每个问题实例的加速比和效率。对不同问题规模和不同数目的处理器而言，具有同样效率的数据点被分成组。当没有相同效率点时，通过计算效率为所需值的相邻点的平均值，计算出问题的规模。这些数据可以从图11-13中看到。图中给出效率分别为0.9、0.85、0.74和0.64时问题规模W与$p \log^2 p$的关系曲线。我们期望对应于同样效率的点共线。从图11-13看出这些点是共线的，表明RP的实验等效率函数接近理论上导出的等效率函数。

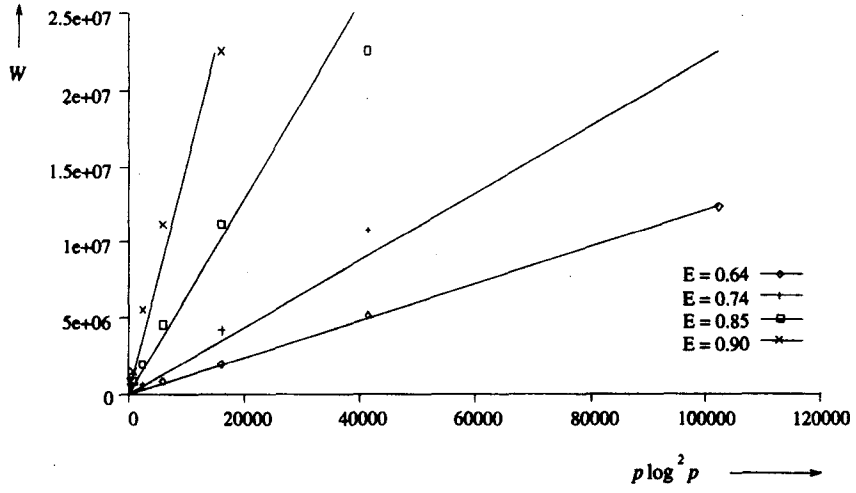


图11-13 RP（随机轮询）对于不同效率的实验等效率曲线

11.4.6 深度优先分支定界搜索的并行形式

深度优先分支定界搜索的并行形式(DFBB)和DFS的并行形式相似。对前面讲述的DFS方法作小的修改，就可以应用到DFBB上，这个修改就是在所有处理器上都保存当前最佳解路径。

对许多问题而言，当前最佳解路径可用一个小数据结构表示。对于共享地址计算机，这个数 495
据结构可以存储在一个全局可访问内存中。处理器每次找到一个解，就把它的成本与当前最佳解路径的成本进行比较。如果它的成本更低，那么它将取代当前最佳解路径而成为新的当前最佳解路径。在消息传递计算机上，每个处理器都保持它知道的当前最佳解路径。无论何时，当有处理器发现比当前最佳解路径成本更低的解路径，就把这个新的解路径成本广播给所有其他的处理器，然后这些处理器更新（如果需要）它们的当前最佳解路径成本。由于只通过单个数值来获得解的成本，并且经常找不到解路径，所以由这个成本值通信所导致的开销非常小。注意，假如某个处理器上的当前最佳解路径比全局最佳解路径差，那么受到影响的就是搜索效率而非正确性。因为DFBB的低通信开销，并行DFBB的性能和可扩展性与前面所讨论的并行DFS的相似。

11.4.7 IDA*的并行形式

由于IDA*以重复方式探查搜索树不断地增加成本界限，自然会想到一种并行形式，就是以单独的处理器独立地探查搜索空间单独的部分。处理器可能用不同的成本界限探查树。这种方法有下面两个缺陷:

1）对特定的处理器而言，如何选一个阈值是不明确的。若为某个处理器选择的阈值正好比全局最小阈值更高，那么这个处理器将探查树中没有被串行IDA*探查的部分。

2）这种方法可能不能找到最优解。某一个处理器在某次迭代时找到的解，不能证明就是最优解，直到所有其他的处理器已穷尽了比找到解的成本低的那些阈值相关的搜索空间。

通过用并行DFS（11.4节）执行IDA*的每次迭代是一种更有效的方法。所有的处理器都使用同样的成本界限；每个处理器在本地存储这个界限，并且在自己的搜索空间中执行DFS。每次并行IDA*迭代后，一个指定的处理器决定下次迭代的成本界限，并重新开始具有新界限的并行DFS。当有处理器找到一个目标节点并通知所有其他的处理器时，搜索结束。这种IDA*的并行形式的性能和可扩展性与并行DFS算法的性能和可扩展性相似。

11.5 并行最佳优先搜索

回忆11.2.2节，开放表是最佳优先搜索算法 (BFS) 的一个重要部分。它维持搜索图中未展 496
开的节点，这些节点按照它们的l值排序。在串行算法中，最有希望的节点从开放表中移去并展开，而新产生的节点加入到开放表中。

在BFS的绝大多数并行形式中，不同的处理器并发地展开开放表中不同的节点。这些方法的区别在于它们实现开放表时使用的数据结构。给定p个处理器，最简单的策略是将每个处理器分配给开放表中当前最佳节点之一。这称为*集中式策略*（centralized strategy），因为每个处理器从一个全局开放表获取任务。由于这种并行BFS形式一次展开不止一个节点，它可能展开在串行算法中不被展开的节点。考虑开放表中第一个节点就是解的情况。并行形式仍然要展开开放表中的前p个节点。但是，由于它总是选取最佳的p个节点，额外工作量是有限的。图11-14说明了这种策略。这种方法存在以下两种问题: 497

1）串行BFS的终止标准不适用于并行BFS。由于在任意时刻，开放表中有p个节点被展开，可能是答案的节点并不对应于最佳目标节点（或找到的路径不是最短路径）。这是因为，剩下的$p-1$个节点可能导致包含更好目标节点的搜索空间。所以，如果某一处理器找到的解的成本

为c，并不能保证这个解与最佳目标节点对应，直到其他处理器搜索的节点成本已知至少为c。必须修改终止标准，保证终止只发生在找到最佳解之后。

2）由于每次节点展开时都要访问开放表，所有处理器必须是容易访问开放表的，这样可能严重地影响性能。即使在共享地址空间体系结构计算机中，对开放表的争用会限制加速比。令t_{exp}为展开一个节点所需的平均时间，t_{access}为展开一个节点时访问开放表的平均时间。如果串行算法和并行算法都要展开n个节点（假设它们的工作量相同），则串行运行时间为$n(t_{access} + t_{exp})$。假设并行化单个节点的展开是可能的，则并行运行时间至少为$n\ t_{access}$，因为展开每个节点时必须至少访问开放表一次。因此，加速比的上界为$(t_{access} + t_{exp})/t_{access}$。

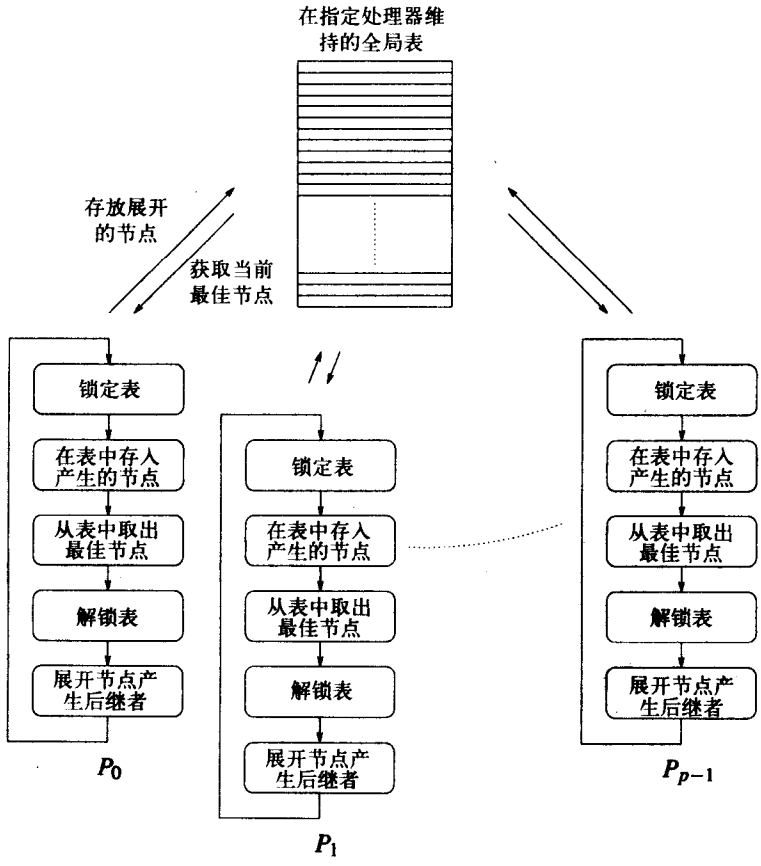


图11-14 使用集中式策略的通用并行最佳优先搜索原理图。
这里的锁定操作用来实现不同处理器的队列访问串行化

为了避免由集中式开放表带来的争用，让每个处理器拥有一个本地开放表。开始时，通过展开一些节点，并将这些节点分配给不同处理器的本地开放表，在处理器间静态地划分搜索空间。然后，所有的处理器同时选择并展开节点。考虑处理器间不互相通信的情况。此时，某些处理器可能探查一些不会被串行算法探查的部分搜索空间。这样就可能带来更高的搜索开销因子及较差的加速比。因此，处理器之间必须通信，以减少不必要的搜索。使用分布式开放表时要对通信和计算作出权衡：减少分布式开放表间的通信开销会增加搜索开销因子，而用增加通信减少搜索开销因子会增加通信开销。

搜索空间是树还是图决定并行BFS最佳通信策略的选择。搜索图会导致检查封闭表中重复节点的额外开销。下面将分别讨论图和树的搜索的一些通信策略。

1. 并行最佳优先树搜索的通信策略

一种通信策略允许状态空间的节点在不同处理器的开放表之间交换。通信策略的目的是为了保证具有较好l值的节点能在处理器间平均分配。本节中我们将讨论如下三种这样的策略。 498

1）在随机通信策略（random communication strategy）中，每个处理器定期将自己的一些最佳节点发送到随机选出的处理器的开放表中。这种策略保证，如果某个处理器存储了搜索空间的较好部分，其他的处理器也能得到搜索空间的一部分。如果节点传输的频率较高，则搜索开销因子会很小；否则搜索开销因子会很大。通信成本决定了最佳节点传输频率。如果通信成本很低，则最好在每个节点展开后再通信。

2）在环形通信策略（ring communication strategy）中，把处理器映射到一个虚拟环。每个处理器定期把自己的一些最佳节点与环中相邻节点的开放表交换。这种策略可以在消息传递或共享地址空间计算机中实现，把处理器组织成一个逻辑环。和上面的策略一样，通信成本决定了节点传输频率。图11-15展示这种环形通信策略。

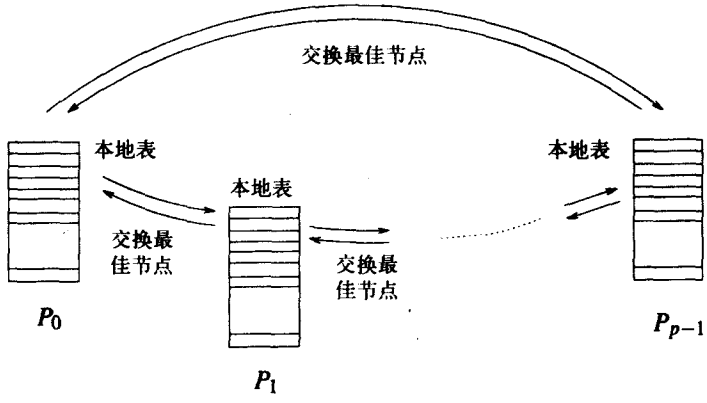


图11-15 用环形通信策略在消息传递计算机上实现并行最佳优先搜索

除非搜索空间高度均匀，否则这种方案的搜索开销因子会非常高，因为要花很长时间将好节点从一个处理器分配到其他所有处理器。

3）在黑板通信策略（blackboard communication strategy）中，节点通过一个共享黑板在处理器间按如下方式交换。处理器从本地开放表中选出最佳节点后，只有当节点的l值处于黑板中最佳节点的容许范围内，处理器才会展开节点。如果选出的节点比黑板中的节点好得多，则处理器在展开当前节点前，将它的一些最佳节点送到黑板中。如果选出的节点比黑板中的节点差得多，则处理器从黑板中取回一些最佳节点，并重新选择节点展开。图11-16展示这种黑板通信策略。黑板策略只适用于共享地址空间计算机，因为在每个节点展开后，必须检查黑板中最佳节点的值。 499

2. 并行最佳优先图搜索的通信策略

当搜索图时，算法必须检查节点是否重复。这个任务在处理器间分配。一种检查重复的方法是将每个节点映射到某一指定的处理器，随后，无论何时产生了节点，都要映射到同一处理器中，并让处理器在本地检查节点重复。这种方法可以使用散列函数实现，将某一节点

作为函数的输入，并返回一个处理器标号。当节点产生后，传送给由散列函数返回标号的处理器。处理器收到节点后，检查本地的开放表或封闭表中是否已有该节点。如果没有，则节点插入到开放表中；如果已有，且新的节点有更好的成本与之对应，则原来的节点就被开放表中新的节点代替。

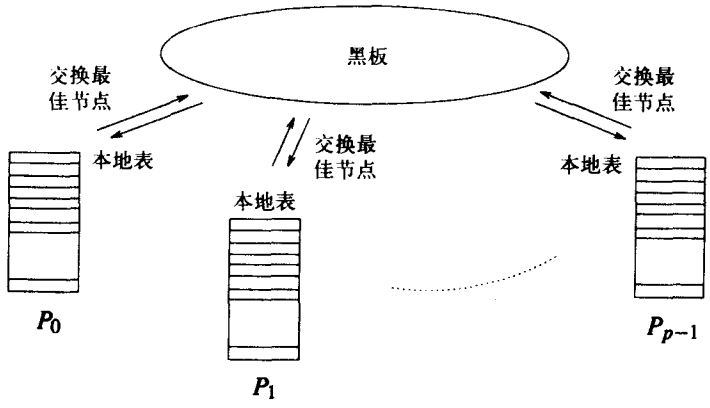


图11-16 使用黑板通信策略的一种并行最佳优先搜索实现

对于随机散列函数来说，这个分配策略的负载平衡性质与上一节讨论的随机分配方法类似。这是因为每个处理器都有同等可能被分配搜索空间一部分，这个部分搜索空间将由串行形式探查。这个方法保证启发值好的节点能在所有的处理器间均匀分配（习题11.10）。但是，
500 由于每个节点的产生都会导致通信，散列方法将使性能下降（习题11.11）。

11.6 并行搜索算法的加速比异常

在并行搜索算法中，加速比在不同的执行间差别很大，因为由不同处理器考查的搜索空间部分是动态决定的，且对于每个执行都可能不同。考虑图11-17所示的串行和并行DFS在树中执行的情况。图11-17a展示串行DFS搜索。节点展开的顺序用节点的标号表示。串行形式在到达目标节点G前产生13个节点。

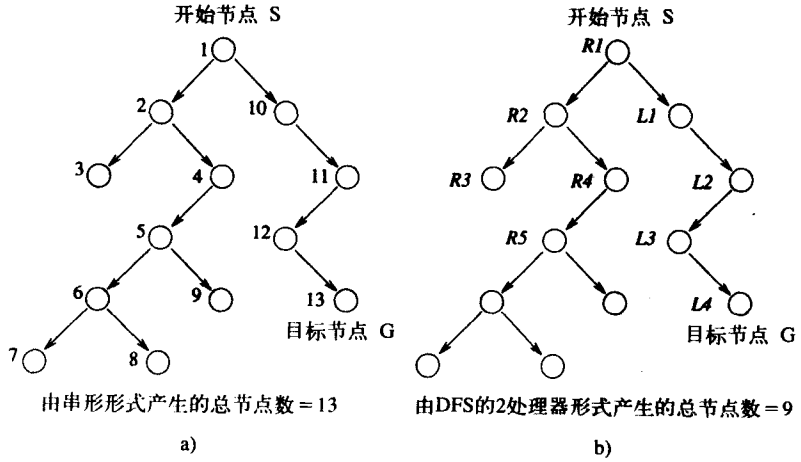


图11-17 DFS的并行形式和串行形式搜索不同数量的节点。本例中，并行DFS到达目标前搜索的节点比串行DFS的要少

下面考虑图11-17b所示两个处理器的同一树。由处理器展开的节点标号为*R*和*L*。并行形式仅在产生9个节点后到达目标节点。也就是说，与串行形式相比，并行形式到达目标节点所需搜索的节点较少。在这种情况下，搜索开销因子为9/13（小于1），如果通信开销不太大，则加速比将是超线性的。

最后，考虑图11-18所示的情况。串行形式（图11-18a）在到达目标节点前产生7个节点，但并行形式产生12个节点。在这种情况下，搜索开销因子大于1，导致亚线性加速比。

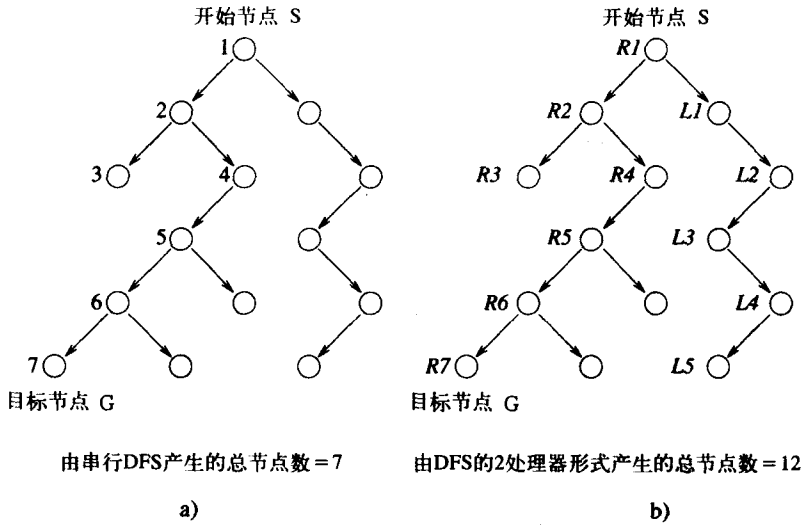


图11-18 并行DFS形式搜索节点数目多于其串行形式

总之，对于有些执行，并行形式求出解时产生的节点比串行形式少，使其可能获得超线性加速比。对于其他的执行，并行形式求出解时产生的节点比串行形式多，导致亚线性加速比。用*p*个处理器执行所产生的加速比大于*p*，称为*加速异常*（acceleration anomaly）。用*p*个处理器执行产生的加速比小于*p*，称为*减速异常*（deceleration anomaly）。 501

最佳优先搜索算法中也显示出加速异常。这里，由于开放表中节点的启发值完全一样，但求解所需的搜索数量却大不相同，因而造成加速异常。假设有两个这样的节点，节点A能很快通向目标节点，而节点B在大量操作后没有达到目标。在并行DFS中，不同的处理器选中这两个节点进行展开。下面考虑并行DFS和串行DFS的对应性能。如果串行算法选中节点A展开，会很快到达目标。但是，并行算法浪费时间展开B，导致减速异常。相反，如果串行算法展开节点B，在丢弃节点B并选择节点A前将浪费大量的时间。然而并行算法不会在B上浪费如此多的时间，因为节点A会很快产生解，导致加速异常。

并行DFS的平均加速比分析

在并行搜索算法孤立执行时，搜索开销因子可能等于1、小于1或大于1。从搜索开销因子的平均值可以看出一些有趣的现象。如果这个值小于1，则说明串行搜索算法不是最优的。此时，运行在串行处理器上的并行搜索算法（通过用时间分片法模拟并行处理器）展开的节点数量平均少于串行算法。在本节中，我们将说明，对于某一类型的搜索空间，并行DFS搜索开销因子的平均值小于1。因此，如果通信开销不太大，就平均而言，并行DFS将对这类搜索 502

空间提供超线性加速比。

1. 假设条件

为了分析加速比异常，我们作如下假设：

1）状态空间树有M个叶节点。解只出现在叶节点上。产生每个叶节点所需的计算量是相同的。树中产生的节点数与产生的叶节点数成正比。由于平均每个节点有一个以上后继节点，关于搜索树的这个假设是合理的。

2）串行和并行DFS在求出一个解后都停止。

3）在并行DFS中，状态空间树在p个处理器中平均划分；因此，每个处理器得到含M/p个叶节点的子树。

4）整棵树中至少有一个解（否则，并行搜索和串行搜索产生整棵树后都找不到解，导致线性加速比）。

5）没有说明状态空间树搜索顺序的信息；因此，通过未搜索节点的解密度与搜索的顺序无关。

6）解密度ρ定义为叶节点是解的概率。假设解服从贝努利分布；即叶节点为解的事件独立于任何其他叶节点是解的事件。另外，我们还假设$\rho << 1$。

7）在某个处理器找到解之前，p个处理器产生的叶节点总数用W_p表示。串行DFS在找到解之前产生的叶节点平均数目为W。W和W_p都小于或等于M。

2. 搜索开销因子分析

考虑M个叶节点被静态地分成p个区域，每个区域有$K = M/p$个叶节点的情形。令第i个区域中叶子之间的解密度为ρ_i。在并行算法中，每个处理器P_i独立地搜索区域i，直到某一处理器找到解。在串行算法中，区域的搜索顺序是随机的。

定理 令ρ为某一区域的解密度；假设区域中的叶子数目K很大。如果$\rho > 0$，那么由搜索
503 区域的单个处理器产生叶子的平均数为$1/\rho$。

证明：由于服从贝努利分布，试验的平均次数为

$$\begin{aligned}\rho + 2\rho(1-\rho) + \cdots + K\rho(1-\rho)^{K-1} &= \frac{1-(1-\rho)^{K+1}}{\rho} - (K+1)(1-\rho)^K \\ &= \frac{1}{\rho} - (1-\rho)^K\left(\frac{1}{\rho} + K\right) \end{aligned} \tag{11-5}$$

对于固定的ρ值和很大的K值，公式(11-5)中的第二项变得很小；因此，平均试验次数近似等于$1/\rho$。

串行DFS从p个区域中选出任一个的概率为$1/p$，并搜索这个区域求解。因此，串行DFS展开叶节点的平均数为

$$W \simeq \frac{1}{p}\left(\frac{1}{\rho_1} + \frac{1}{\rho_2} + \cdots + \frac{1}{\rho_p}\right)$$

这个表达式假设在选中的区域中总有解；因此，只需搜索一个区域。但是，区域i中不包含解的概率为$(1-\rho_i)^K$。在这种情况下，必须搜索另一个区域。考虑到这一点使W的表达式更精确，同时也使W的平均值有所增加。分析的总体结果不变。

在并行DFS的每一步中，p个区域中每个区域的一个节点被同时探查。因此，一步并行算法中成功的概率为$1-\prod_{i=1}^{p}(1-\rho_i)$。这近似于$\rho_1+\rho_2+\cdots+\rho_p$（忽略二次项，因为假设每个$\rho_i$都很小）。因此，

$$W_p \simeq \frac{p}{\rho_1+\rho_2+\cdots+\rho_p}$$

从上式可以看出，$W = 1/HM$，$W_p = 1/AM$，其中HM是$\rho_1, \rho_2, \cdots, \rho_p$的调和平均值，$AM$是它们的算术平均值。由于算术平均值($AM$)和调和平均值($HM$)满足关系$AM \geq HM$，我们有$W \geq W_p$。特别是：

- 当$\rho_1 = \rho_2 = \cdots = \rho_p$时，$AM = HM$，因此$W \simeq W_p$。当解均匀分布时，并行DFS的平均搜索开销因子为1。
- 当每个ρ_i不同时，$AM > HM$，因此$W > W_p$。当解密度在不同区域不均匀时，并行DFS的平均搜索开销因子小于1，使其可能获得超线性加速比。

上面假设每个节点可能独立于其他是解的节点，对于绝大多数实际问题来说，这种假设是不成立的。还有，前面的分析假定，如果解不在搜索空间内均匀分布，且没有与不同区域的解密度相关的信息，则并行DFS比串行DFS能获得更高的效率。这种特性能适用于许多用简 504
单回溯搜索的问题空间。并行DFS的搜索开销因子平均至少是1的结果很重要，因为DFS是目前已知的算法中最好的，也是用于求解许多重要问题最实用的串行算法。

11.7 书目评注

大量文献研究关于离散优化问题的搜索算法，如分支定界搜索及启发式搜索[KK88a, LW66, Pea84]。Kumar和Kanal[KK83, KK88b]研究人工智能中的分支定界法搜索、动态规划以及启发式搜索之间的关系。Smith[Smi84]和Wilf[Wil86]指出，在许多问题中，启发式搜索算法的平均时间复杂度为多项式量级。关于搜索算法的并行形式，也有许多人做了大量工作。下面我们简要概括一下这些人的贡献。

1. 并行深度优先搜索算法

已经提出许多关于DFS的并行算法[AJM88, FM87, KK94, KGR94, KR87b, MV87, Ran91, Rao90, SK90, SK89, Vor87a]。负载平衡是并行DFS的中心问题。在本章中，并行DFS的任务分配使用堆栈分割法实现[KGR94, KR87b]，另一种分配任务方案使用节点分割法，在其中只分配单个节点[FK88, FTI90, Ran91]。

本章中讨论了状态空间搜索的形式，其中当处理器空闲时请求任务。这种负载平衡方案称之为*接收者启动*（receiver-initiated）的方案。在其他的负载平衡方案中，有任务的处理器分出一部分任务给其他（收到或未收到请求的）处理器。这些方案称为*发送者启动*（sender-initiated）的方案。

有几位研究人员在并行DFS中使用接收者启动的负载平衡方案[FM87, KR87b, KGR94]。Kumar 等[KGR94]分析这些负载平衡方案，包括全局循环、随机轮询、异步循环以及最近邻居等。11.4节对这些方案的描述和分析就是基于Kumar等[KGR94, KR87b]的文章。

有些研究人员提出了使用发送者启动的负载平衡方案的并行DFS[FK88, FTI90, PFK90, Ran91, SK89]。Furuichi等提出了单层及多层基于发送者的方案[FTI90]。Kimura和Nobuyuki

[KN91]提出了这些方案的可扩展性分析。Ferguson和Korf[FK88, PFK90]提出称为分布式树搜索(distributed tree search, DTS)的负载平衡方案。

对状态空间树并行DFS提出了使用随机分配的其他方法[KP92, Ran91, SK89, SK90]。在[RK87, SK89]中讨论了并行DFS中的有关粒度控制的问题。

505 在Saletore和Kale[SK90]提出的并行DFS形式中，节点被赋予优先权并被相应地展开。他们指出，优先权DFS形式的搜索开销因子非常接近于1。对于足够大的问题，用不断增加处理器的数目就能获得持续增加的加速比。

在一些关于深度优先搜索的并行形式中，由不同的处理器按随机顺序独立地搜索状态空间[JAM87, JAM88]。Challou 等[CGK93]和Ertel[Ert92]分别指出，这样的方法能够用于求解机器人动作规划及定理证明问题。

绝大多数通用的DFS形式适用于深度优先分支定界以及IDA*，一些研究人员专门研究了深度优先分支定界并行形式[AKR89, AKR90, EDH80]。在[RK87, RKR87, KS91a, PKF92, MD92]中提出了许多IDA*的并行形式。

绝大多数并行DFS形式仅适用于MIMD计算机中。由于搜索问题的性质，SIMD计算机被认为本质上不适合并行搜索。但是，Fyre和Myczkowski[FM92]，Powley 等[PKF92]以及Mahanti和Daniels[MD92]的研究表明，并行深度优先方法也能应用在SIMD计算机中。Karypis和Kumar[KK94]提出了SIMD计算机的并行DFS方案，这些方案同MIMD计算机中的方案一样是可扩展的。

有几位研究人员从实验的角度评估了并行DFS。Finkel和Manber[FM87]在由Wisconsin大学研制的Crystal多计算机上，给出一些问题的性能结果，如旅行商问题和骑士巡游问题。Monien和Vornberger[MV87]指出在超级微处理机网络中许多组合问题的线性加速比。Kumar 等[AKR89, AKR90, AKRS91, KGR94]为15迷宫、重言式验证、自动测试模式产生等问题在多种不同的体系结构上实现线性加速比，包括128个处理器的BBN Butterfly、128个处理器的Intel iPSC、1024个处理器的nCUBE2以及128个处理器的Symult 2010。Kumar, Grama和Rao[GKR91, KGR94, KR87b, RK87]在超立方体、格网以及工作站网络研究了其中多个方案的可扩展性及性能。11.4.5节中的实验结果取自Kumar, Grama和Rao的文章[KGR94]。

也有研究人员研究了DFBB的并行形式。这些形式中，有许多都基于维持一个当前最佳解，用作全局界限。研究人员发现，维持当前最佳解的开销只是动态负载平衡开销的很小部分。对于许多问题和体系结构，DFBB的并行形式被证实能产生接近线性的加速比[ST95, LM97, Eck97, Eck94, AKR89]。

还有许多研究人员提出了用于并行搜索中的终止检测算法。Dijkstra[DSG83]提出了环形终止检测算法。在11.4.4节中讨论的基于权的终止检测算法，与Rokusawa 等[RICN88]提出的
506 算法类似。Dutt和Mahapatra[DM93]讨论基于最小生成树的终止检测算法。

2. Alpha-Beta搜索的并行形式

Alpha-beta搜索从本质上说是深度优先分支定界搜索方法，它从AND/OR图中找出一棵最优解树[KK83, KK88b]。许多研究人员开发了Alpha-Beta搜索的并行形式[ABJ82, Bau78, FK88, FF82, HB88, Lin83, MC82, MFMV90, MP85, PFK90]。这些方法中的某些方法在多处理器中实现了可观的加速比[FK88, MFMV90, PFK90]。

并行处理的实用性在多种游戏中得到了证明，如在国际象棋中。1990年开发的深思(Deep Thought)就基于大规模并行的Alpha-Beta搜索[Hsu90]。这个程序的棋力达到大师级水平。后来设计出的IBM深蓝(Deep Blue)[HCH95, HG97]改进了专用硬件、并行处理和算法，击败了世界冠军Gary Kasparov。Feldmann 等[FMM94]开发了一个分布式国际象棋程序，被认为是完全使用通用硬件的最好的计算机下棋程序之一。

3. 并行最佳优先搜索

许多研究人员研究了A*及分支定界算法的并行形式[KK84, KRR88, LK85, MV87, Qui89, HD89a, Vor86, WM84, Rao90, GKP93, AM88, CJP83, KB57, LP92, Rou87, PC89, PR89, PR90, PRV88, Ten90, MRSR92, Vor87b, Moh83, MV85, HD87]。所有这些形式都用不同的数据结构存储开放表，有些形式使用集中式策略[Moh84, HD87]；有些使用分布式策略，如随机通信策略[Vor87b, Dal87, KRR88]、环形通信策略[Vor86, WM84]以及黑板通信策略[KRR88]。Kumar等[KRR88]通过旅行商问题、顶点覆盖问题以及15迷宫问题，从实验角度评估了集中式策略以及一些分布式策略的性能。Dutt和Mahapatra [DM93, MD93]提出并评估了许多其他的通信策略。

Manzini分析了在并行图搜索中分配节点的散列技术[MS90]。Evett 等[EHMN90]提出了并行收缩A*(PRA*)，它能在内存受限的条件下运行。在这种形式中，每个节点散列到唯一的处理器。如果处理器接收到的节点数目多于它能在本地存储的数目，则用较差的启发值收缩节点。当更多有希望的节点不能成为解时，这些收缩的节点被再次展开。

Karp和Zhang[KZ88]用节点的一种指定搜索树模型的随机分布分析并行最佳优先分支定界(即A*）的性能。Renolet 等[RDK89]使用Monte Carlo模拟建立并行最佳优先搜索的性能模型。Wah 和Yu[WY85]提出分析深度优先分支定界搜索以及最佳优先分支定界搜索并行形式性能的随机模型。

Bixby[Bix91]提出用于求解对称旅行商问题的切枝算法。他还提出用于航班乘员调度模型 507
的LP松驰解。Miller等[Mil91]提出一种最佳优先分支定界并行形式，用于在异构网络计算机体系结构中求解不对称旅行商问题。Roucairol[Rou91]提出共享地址空间计算机中的并行最佳优先分支定界并行形式，用于求解多背包问题及二次分配问题。

4. 搜索算法并行形式的加速比异常

许多研究人员分析了并行搜索算法的加速比异常[IYF79, LS84, Kor81, LW86, MVS86, RKR87]。Lai和Sahni[LS84]提出早期用于最佳优先搜索量化加速比异常的方法。Lai和Sprague[LS86]改进和推广了这项工作。Lai和Sprague[LS85]还提出一种解析模型，并导出下界函数的性质，能够保证当处理器的数目增加时异常不会发生。Li和Wah[LW84, LW86]以及Wah等[WLY84]研究支配关系及启发式函数，以及它们对不利异常（使用p个处理器的加速比小于1）及加速异常的影响。Quinn和Deo[QD86]导出用最佳定界搜索策略的分支定界算法的任意并行形式可达到的加速比上界。Rao和Kumar[RK88b, RK93]针对使用及不使用启发式排序信息的两种不同模型，分析并行DFS的平均加速比。他们指出，这两种情况下的搜索开销因子至多为1。11.6.1节的内容即基于Rao和Kumar[RK93]的结果。

最后，人们也开发了用于实现并行搜索的多种编程环境，例如DIB[FM87]，Chare-Kernel[SK89]，MANIP[WM84]以及PICOS[RDK89]。

5. 启发式的作用

在搜索方法中，启发式方法是最重要的部分，并且必须把搜索算法的并行形式算作这种启发式方法。在BFS方法中，启发式注重通过降低有效分支因子进行搜索。在DFBB方法中，启发式提供更好的界限，并从而能减小搜索空间。

通常，在启发式能力和搜索空间的有效大小之间存在一种平衡。更好的启发式导致更小的搜索空间，但是计算成本也更高。例如，在混合整数规划（MIP）中计算界限是强启发式的一个重要应用。多年来，混合整数规划的进展非常迅速[JNS97]。但在15年前，含100个整型变量的MIP问题被认为是非常复杂的，如今，在工作站级别的计算机上，使用枝切法就能够求解最多含1000个整型变量的许多问题。广为人知的旅行商问题(TSP)和二次分配问题(QAP)的实例就是用强启发式分支定界法求解的[BMCP98, MP93]。强启发式的存在可以很好
508 地减少搜索空间的大小。例如，当Padberg和Rinaldi于1987年提出分支定界算法时，他们用算法求解532个城市的TSP，这是当时解决的最大的也是解最优的TSP。算法改进后能求解2392个城市的TSP[PR91]。最近，使用切割平面，已能在串行计算机中求解超过7000个城市的TSP[JNS97]。但是，对于许多问题来说，减小的搜索空间仍然需要使用并行方法[BMCP98, MP93, Rou87, MMR95]。将强启发式与有效的并行处理相结合，可以求解极大的问题[MP93]。例如，在NEC的Cenju-3并行计算机中，只用不到9天时间，就求解了Nugent和Eschermann测试程序中最多到4.8×10^{10}个节点的QAP问题（Nugent22与Gilmore-Lawler界限）。IBM的深蓝也同样令人吃惊。深蓝结合用于产生和计算棋盘位置的专门硬件与使用IBM SP2的并行博弈树搜索算法，击败了国际象棋世界冠军Gary Kasparov[HCH95, HG97]。

启发式方法对研究并行化有若干重要的结论。强启发式使状态空间变小，从而降低搜索空间中可用的并发性。同样，使用强启发式也会给并行处理带来其他计算方面的困难。例如，在切枝方法中，在某一状态的切割可能在其他状态也需要。因此，除了要平衡负载外，并行切枝方法还必须在处理器间划分切割，使工作于某个LP域的处理器能访问它们需要的切割[BCCL95, LM97, Eck97]。

除了已经讨论的节点间的并行外，如果启发式的计算成本很高，则内部节点并行也是一种可行的方案。例如，Petkny等的赋值启发式已被有效地并行化，用于求解城市数目很大时的TSP问题[MP93]。如果节点间的并行度很小，则可选用此种并行形式。分支定界法求解MIP问题时也可使用此并行形式。切枝法利用LP松弛法产生已部分求出的解的下界，然后再产生有效不等式[JNS97]。这些LP松弛构成整体计算时间的主要部分。许多工业代码都依靠单纯形法求解LP问题，因为即使行或列增加，解也适用。虽然内点法更适合并行，但当行或列增加后（用切枝法时），重新优化LP解时效率较差。如果处理器数目较少，使用单纯形法的LP松弛并行化较好，但如果扩展到大量的处理器，还未取得成功。基于LP的分支定界法也用来求解二次分配问题，使用迭代求解程序，例如，用带前置条件的共轭梯度近似计算内点方向[PLRR94]。人们使用这些方法，用超过150 000个约束条件和300 000个变量的线性规划，求解QAP的下界。当有大量的处理器时，这些求解程序和其他一些迭代求解程序也能非常有效
509 地并行化。计算启发式解的通用并行框架是由Pramanick和Kuhl[PK95]提出的。

习题

11.1 [KGR94]在11.4.1节，我们指出对全局指针*target*的访问是GRR负载平衡方案中的瓶

颈。考虑对方案进行修改，在其中增加消息结合。这个方案的工作方式如下。所有对处理器0全局指针*target*的值的读取请求都被结合到中间处理器。因此，处理器0要处理的请求总数就大大减少了。本质上，这个方法是取数和相加操作的软件实现。这个方法也称为GRR-M（带消息结合的GRR）。

图11-19展示如何实现这个方法。每个处理器处在完全（逻辑）二叉树的叶节点上。注意这样的逻辑树可以方便地映射到物理结构中。当处理器想对*target*进行原子读并增1操作时，它将请求沿着树向上递交给处理器0。树的一个内部节点保留它的一个子节点的请求，保留时间最多为δ，然后把消息转发给父节点。如果在δ时间内有请求来自节点的另一个子节点，则这两个请求被结合，并作为一个请求向上传送。如果i是被结合的增加请求的总数目，则*target*的请求增量为i。

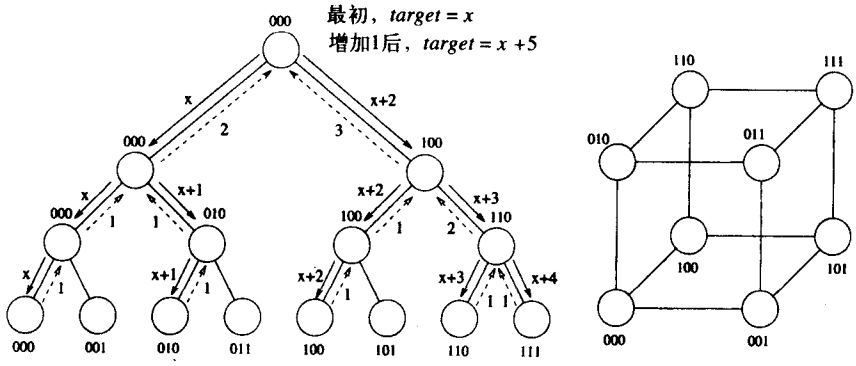


图11-19 消息结合及其在8处理器超立方体中的一个实现实例

每个处理器处返回的值等于所有对*target*的请求被串行化时处理器返回的值。实现的过程如下：每个结合后的消息存储在每个处理器的表中，直到请求被满足。当*target*的值送回到内部节点时，如果左右子节点都从*target*处请求值，则两个值下传给左子节点和右子节点。这两个值由表中的项决定，它与两个子节点的增量请求对应。方案如图11-19所示，图中起始时*target*的值为x，处理器P_0、P_2、P_4、P_6和P_7发出请求。总请求增量为5。在所有的消息结合并处理后，*target*从这些处理器处接收的值分别为x、$x+1$、$x+2$、$x+3$和$x+4$。 510

对消息传递体系结构，分析该方案的性能及可扩展性。

11.2 [Lin92] 考虑另一种负载平衡策略。假设每个处理器维持一个称为*counter*的变量。初始时，每个处理器将*counter*的本地副本置0。只要有处理器空闲时，就在嵌入到任意结构的逻辑环中查找处理器P_i和P_{i+1}，使得在P_i处的*counter*值大于在P_{i+1}处的*counter*值。然后，空闲的处理器向处理器P_{i+1}发送任务请求。如果不存在这样的处理器对P_i和P_{i+1}，则请求发送给处理器0。收到一个任务请求后，处理器将本地*counter*的值加1。

请设计检测处理器P_i和P_{i+1}对的算法。根据检测P_i和P_{i+1}对的算法，在消息传递结构上分析负载平衡方案的可扩展性。

提示：这个方案的任务传送数目的上界与GRR的类似。

11.3 在分析11.4.2节中的多种负载平衡方案时，我们假设传送任务的成本独立于已传送任务的数量。但是，在有些问题中，任务传送的成本是已传送任务数量的函数。例如，有的应用程序在可使用强启发式的域中进行树搜索。对于这种应用程序，根据搜索树的节点数目

不同，表示搜索树的堆栈大小会有显著变化。

考虑表示含w个节点的搜索空间的堆栈大小为$\sqrt{w}$的情况。假定所用的负载平衡方案是GRR。分析在消息传递体系结构中这种方案的性能。

11.4 考虑11.4.4节中的Dijkstra令牌终止检测方案。证明用这种终止检测方案对整体等效率函数的贡献为$O(p^2)$。对与这个等效率项相关的常量加以评论。

11.5 考虑11.4.4节中的基于树的终止检测方案。在这种算法中，由于计算机的精度有限，权可能变得很小，最终可能会变为0。在这种情况下，终止信号永远也不会发出。可对算法修改，只对权的倒数而不是权自身进行操作。请编写修改后的终止算法，并证明算法能正确进行终止检测。

11.6 [DM93]考虑一种终止检测算法，在这种算法中，直径最小的生成树映射到给定的并行计算机的体系结构中。这种树的中心是一个顶点，它到离它最远顶点的距离达到最小。
511 生成树的中心被作为树根。

进行并行搜索时，处理器可能空闲，也可能繁忙。终止检测算法要求系统中所有的任务传送都被*ack*消息确认。如果处理器有任务，或者它已把任务送给另一个处理器，但对应的*ack*消息还没收到，那么处理器繁忙；否则处理器空闲。生成树叶子上的处理器在空闲时，发送*stop*消息给它们的父节点。生成树中间层上的处理器在收到所有子节点传来的*stop*消息后，把*stop*消息上传给它们的父节点，然后它们自己变为空闲。当根处理器从所有的子节点收到*stop*消息并变为空闲时，就发出终止信号。

由于处理器发送*stop*消息给父节点后还可能接收到任务，处理器通过发送*resume*消息给它的父节点表示它已接收到任务。*resume*消息在树中上传，直到遇到先前发出的*stop*消息。遇到*stop*消息时，*resume*消息使*stop*消息无效。然后，发送一条*ack*消息给转送出部分任务的处理器。

用实例证明这种终止检测方法能正确发送终止信号。对于深度为log p的生成树，确定基于该终止检测算法的等效率项。

512 11.7 [FTI90，KN91]考虑一种单层负载平衡方案，工作方式如下：一个指定的称为*manager*的处理器产生许多子任务，并把子任务一个接一个地分配给请求的处理器。*manager*按深度优先遍历搜索树到一预定的截止深度，并将此深度的节点作为子任务分配。增加截止深度会增加子任务的数目，但使子任务更小。处理器只有在完成以前的子任务后，才能从*manager*请求另一个子任务。因此，如果某一处理器得到了与大子树对应的子任务，就会向*manager*发送很少的请求。如果截止深度足够大，这个方案会在处理器间导致好的负载平衡。但是，如果截止深度过大，则分配给处理器的子任务会变得很小，处理器会更频繁地发送请求给*manager*。在这种情况下，*manager*成为瓶颈。因此，这个方案的可扩展性很差。图11-20展示这种单层任务分配方案。

假设在任意两个处理器间传送一部分任务的成本可以忽略不记，请推导单层负载平衡方案可扩展性的解析表达式。

11.8 [FTI90，KN91]考虑多层任务分配方案，通过多层子任务的产生防止单层方案的子任务产生瓶颈。在这种方案中，处理器排列成深度为d的m元树。顶层子任务产生的任务交给根处理器。该方案将任务分成超级子任务，并将这些超级子任务分配给请求中的后面的处理器。这些处理器再细分超级子任务为子任务，并把子任务分配给请求中的后面的处理器。一

且叶子处理器完成了前面的任务，就重复地从它们的父节点请求任务。当某一叶子处理器指定的子任务产生器处理完任务时，该叶子处理器就分配给另一个子任务产生器。当$d = 1$时，多层和单层方案完全相同。试评价这个方案的性能和可扩展性。

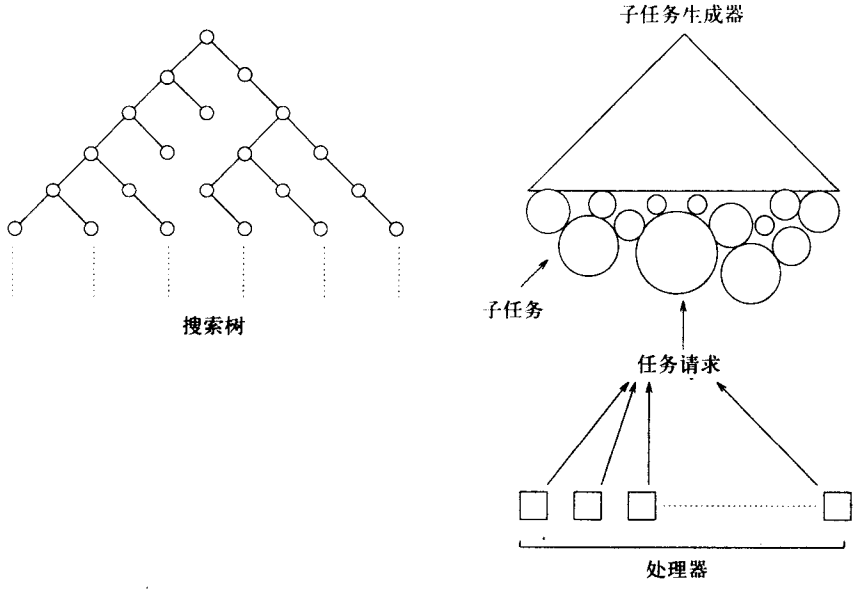


图11-20 树搜索的单层任务分配方案

11.9 [FK88]考虑分布树搜索方案，其中处理器被动态地分配给搜索树的不同部分。初始时，所有的处理器都被分配给根。当根节点展开时（由分配给它的处理器之一展开），按照选择的处理器分配策略，把在根节点的不相连的处理器子集分配给每个后继。一个可能的处理器分配策略是，将处理器平等地在先辈节点中划分。这个过程直到只有一个处理器分配给一个节点时结束。此时，处理器串行地搜索根在节点的树。如果处理器完成了对根在节点的搜索树的搜索，它被重新分配给父节点。如果父节点仍然有其他的后继节点要探查，则这个处理器被分配给这些后继节点之一。否则，处理器分配给父节点。直到整个树搜索完毕，过程才终止。评价这个方案的性能和可扩展性。

11.10 考虑一种图的最佳优先搜索并行形式，使用散列函数将节点分配给处理器（11.5节）。这种方案的性能受两个因素影响：通信成本以及展开的“好”节点数目（“好”节点是串行算法也会展开的节点）。对这两个因素的分析可以相互独立地进行。 513

假设一个完全随机的散列函数（函数中每个节点被散列到处理器的概率等于$1/p$）。证明由这个并行形式展开的预期节点数目不等于通过常量因子（即独立于p）展开的最优节点数目。假设将一个节点从一个处理器传送到另一个处理器所需的通信成本为$O(1)$，导出这个方案的等效率函数。

11.11 在习题11-10的并行形式中，假设串行形式和并行形式展开的节点数目相同。对消息传递结构分析这种形式的通信开销。这种形式可扩展吗？如果可以，它的等效率函数是什么？如果不可以，在什么样的互连网络中，它才是可扩展的？

提示：注意完全随机的散列函数与多对多私自通信操作对应，它是带宽敏感的。 514

第12章 动 态 规 划

动态规划（dynamic programming, DP）是一种常用的方法，可用来求解多种多样的离散优化问题，如调度、字符串编辑、包装问题以及仓储管理等。最近，人们又将动态规划应用于生物信息学中，如氨基酸及核苷酸测序等（Smith-Waterman算法）。DP把问题看作一组相互依赖的子问题。先求解子问题，再用子问题的结果去求解更大的子问题，直到求解整个问题。在分而治之方法中，问题的求解只与子问题的求解有关，与之相反，DP中子问题间可能存在相互联系。在DP中，子问题的解是以前面层的一个或多个子问题的解的函数来表示的。

12.1 动态规划概述

下面，我们首先用一个简单的DP算法来计算图的最短路径。

例12.1 最短路径问题

考虑查找无圈图中一对顶点间最短路径问题的DP形式。（有关图的术语的介绍见10.1节。）连接节点i和节点j的边的成本为$c(i, j)$。如果两个顶点i和j不相连，则$c(i, j) = \infty$。图中含n个节点，编号为0, 1, 2, …, $n-1$，只有在$i < j$时才存在从i到j的边。最短路径问题就是在节点0和节点$n-1$间找到最小成本路径。用$f(x)$表示从节点0到节点x的最小成本路径的成本。于是$f(0)$为0，求出$f(n-1)$的值就得到了问题的解。这个问题的DP形式得出下面$f(x)$的递归公式： 515

$$f(x) = \begin{cases} 0 & x = 0 \\ \min\limits_{0 \le j < x} \{f(j) + c(j, x)\} & 1 \le x \le n-1 \end{cases} \qquad (12\text{-}1)$$

作为这个算法的例子，考虑图12-1所示的5节点无圈图。问题是求出$f(4)$。它可由$f(3)$和$f(2)$计算。具体的计算公式如下：

$$f(4) = \min\{f(3) + c(3, 4), f(2) + c(2, 4)\}$$

因此，$f(4)$的求解要依靠子问题$f(3)$和$f(2)$。同样，$f(3)$要依靠$f(1)$和$f(2)$，而$f(1)$和$f(2)$依靠$f(0)$。由于$f(0)$已知，它可用来求解$f(1)$和$f(2)$，而$f(1)$和$f(2)$可用来求解$f(3)$。 ■

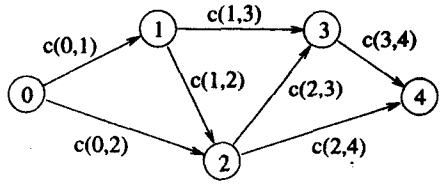


图12-1 计算节点0和4之间最短路径的图

通常，DP问题的解表示为可能选择解的最小值（或最大值）。每个这样的可选择解由组合一个或多个子问题来构造。如果r表示由子问题$x_1, x_2, \cdots, x_l$所组成解的成本，则r可写成

$$r = g(f(x_1), f(x_2), \ldots, f(x_l))$$

函数g称为*组合函数*（composition function），其性质由问题决定。如果每个问题的最优解都由组合子问题的最优解并选择最小值（或最大值）决定，则形式称为DP形式。图12-2展示这种组合及最小值的实例。问题x_8的解是三个成本为r_1、r_2和r_3的可能解当中的最小值。第一个解的成本由子问题x_1和x_3的组合解决定，第二个解的成本由子问题x_4和x_5的组合解决定，第三个解的成本由子问题x_2、x_6和x_7组合的解决定。

DP将优化问题的解表示成一个递归公式，其左边是未知量，右边是一个最小值（或最大值）的表达式。这种公式称为*泛函方程*（functional equation）或*优化方程*（optimization
516 equation）。在公式(12-1)中，复合函数g由$f(j) + c(j, x)$给出，这个函数是可加的，因为它是两个项的和。在一般的DP形式中，成本函数不必要是可加的。泛函方程中如果只包含一个递归项（如$f(j)$），就变成*一元*（monadic）DP形式。对于任意的DP形式，成本函数可能包含多个递归项。如果成本函数包含多个递归项，则DP形式称为*多元*（polyadic）形式。

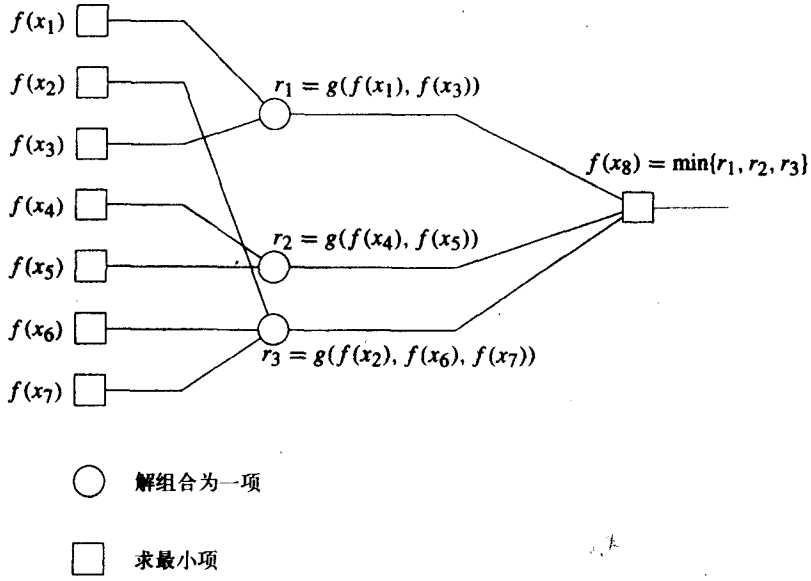


图12-2 计算和组合子问题的解以求解问题$f(x_8)$

DP形式中，子问题间的相关性可用有向图来表示。图中的每个节点表示一个子问题。从节点i到节点j的有向边的含义是：节点i表示的子问题的解用来计算由节点j表示的子问题的解。如果图是无圈图，则图中的节点可按层来组织，只要某一层上的子问题只取决于上一层的子问题。在这种情况下，DP形式可用如下方法分类。如果所有层的子问题只取决于它的直接上层的结果，则形式称为*串行*DP形式；否则称为*非串行*DP形式。

根据上面的分类原则，我们将DP形式分成四类：*串行一元*、*串行多元*、*非串行一元*，以及*非串行多元*。但这种分类并不完备：有些DP形式不能归到这四类形式的任何一类中。

由于用DP形式能求解很多类问题，很难开发一般性的并行算法。但是，四类DP中每个问题的并行形式都有相似之处。在本章中，我们将对每一类有代表性的问题讨论并行DP形式。
517 这些问题的并行算法也能用在同一类的其他问题中。但是要注意，不是所有的DP问题都能用这些例子所讲的方法并行化。

12.2 串行一元DP形式

使用串行一元DP形式可求解许多问题。本节将讨论多级图的最短路径问题以及0/1背包问题。我们对这两个问题都提供了并行算法，并指出影响这些并行算法的具体性质。

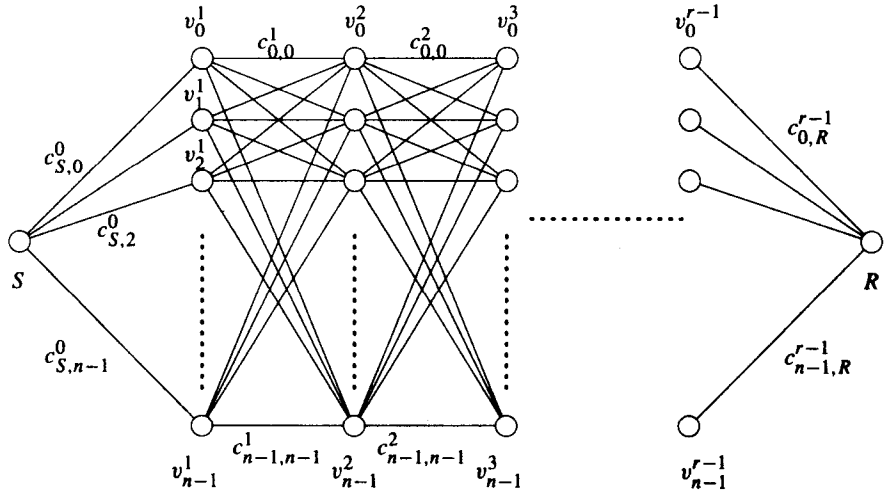


图12-3 在节点可组织成层的图中用串行一元DP形式查找最短路径的例子

12.2.1 最短路径问题

考虑如图12-3所示的$r+1$层加权多级图。层i上的每一个节点都与$i+1$层上的每个节点相连。0层和r层只有一个节点，每个其他的层都有n个节点。我们将0层上的节点称为起始节点S，将r层上的节点称为终止节点R。这个问题的目标是找出从S到R的最短路径。图中l层的第i个节点标记为 v_i^l 。连接节点 v_i^l 与 v_j^{l+1} 的边的成本标记为 $c_{i,j}^l$ 。从任意节点 v_i^l 到目标节点R的成本用 C_i^l 表示。如果层l有n个节点，则向量$[C_0^l, C_1^l, \cdots, C_{n-1}^l]^T$记为$C^l$。最短路径问题简化为计算$C^0$。由于图只含一个起始节点，所以$C^0 = [C_0^0]$。图的结构为，从 v_i^l 到R的任何路都要包含节点 $v_j^{l+1}(0 \leqslant j \leqslant n-1)$。任何这种路径的成本都是 v_i^l 和 v_j^{l+1} 间边的成本与 v_j^{l+1} 与R（由 C_j^{l+1} 给出）间 518
最短路径的成本之和。因此， v_i^l 和R间的最短路径成本 C_i^l ，等于所有通过层$l+1$上每个节点的路径的最小成本。所以，

$$C_i^l = \min\left\{(c_{i,j}^l + C_j^{l+1}) \mid j \text{ is a node at level } l+1\right\} \tag{12-2}$$

由于所有的节点 v_j^{r-1} 在r层只有一条边将它们与目标节点R相连，成本 C_j^{r-1} 等于 $c_{j,R}^{r-1}$ 。因此，

$$C^{r-1} = [c_{0,R}^{r-1}, c_{1,R}^{r-1}, \ldots, c_{n-1,R}^{r-1}] \tag{12-3}$$

因为公式(12-2)的右边只含一个递归项，它是一元形式。注意子问题的解只需要它的直接上层的子问题的解。因此，它是串行一元形式。

对最短路径问题使用这种递归形式，可得从l层上$(0 < l < r-1)$任意节点到达目标节点R的

成本为

$$\begin{aligned} C_0^l &= \min\{(c_{0,0}^l + C_0^{l+1}), (c_{0,1}^l + C_1^{l+1}), \ldots, (c_{0,n-1}^l + C_{n-1}^{l+1})\} \\ C_1^l &= \min\{(c_{1,0}^l + C_0^{l+1}), (c_{1,1}^l + C_1^{l+1}), \ldots, (c_{1,n-1}^l + C_{n-1}^{l+1})\} \\ &\vdots \\ C_{n-1}^l &= \min\{(c_{n-1,0}^l + C_0^{l+1}), (c_{n-1,1}^l + C_1^{l+1}), \ldots, (c_{n-1,n-1}^l + C_{n-1}^{l+1})\} \end{aligned}$$

现在考虑矩阵与向量相乘操作。在矩阵-向量乘积中，如果加法操作用求最小值代替，乘法操作用加法操作代替，则上面的方程组等于

$$C^l = M_{l,l+1} \times C^{l+1} \tag{12-4}$$

其中C^l和C^{l+1}是$n \times 1$向量，表示从l层和$l+1$上的每个顶点到达目标顶点的成本，$M_{l,l+1}$是$n \times n$矩阵，矩阵中的项(i, j)存储连接l层节点i和$l+1$层节点j的边的成本。这个矩阵可写为

$$M_{l,l+1} = \begin{bmatrix} c_{0,0}^l & c_{0,1}^l & \cdots & c_{0,n-1}^l \\ c_{1,0}^l & c_{1,1}^l & \cdots & c_{1,n-1}^l \\ \vdots & \vdots & & \vdots \\ c_{n-1,0}^l & c_{n-1,1}^l & \cdots & c_{n-1,n-1}^l \end{bmatrix}$$

最短路径问题重新表示成一系列矩阵-向量相乘。在串行计算机上，DP形式开始时用
519 公式(12-3)计算C^{r-1}，然后用公式(12-4)对$k = 1, 2, \cdots, r-2$计算C^{r-k-1}。最后，用公式(12-2)计算C^0。

由于每一层上有n个节点，计算每个向量C^l的成本为$\Theta(n^2)$。可以从8.1节讨论的矩阵-向量相乘并行算法中得到这个问题的并行算法。例如，$\Theta(n)$个处理器可以在$\Theta(n)$时间内计算每个向量C^l，而求解整个问题的时间为$\Theta(r\,n)$，其中r是图的层数。

如果依赖图与上面讲的图相同，则许多关于依赖图的串行一元DP形式可用类似的并行算法并行化。但是，对于某些依赖图而言，这种形式并不适合。如果某个图中，某一层的每个节点只能从前一层的一小部分节点处到达，则矩阵$M_{l,l+1}$中会包含许多值为∞的元素。此时，对于求最小值和加法运算而言，矩阵M被考虑成稀疏矩阵。这是因为，对于所有的x，$x + \infty = \infty$，而$\min\{x, \infty\} = x$。因此，对于值为∞的元素不需要进行求最小值和加法操作。如果使用常规的稠密矩阵-向量相乘算法，则每个矩阵-向量相乘的计算复杂度比对应的稀疏图矩阵-向量相乘算法高出很多。所以，我们必须使用稀疏图矩阵-向量相乘算法计算每个向量。

12.2.2 0/1背包问题

一维0/1背包问题定义如下：给定背包的容量为c，n个物品的编号为$1, 2, \cdots, n$，每个物品i的重量为W_i，利润为p_i。物品的重量和利润为整数。令$v = [v_1, v_2, \cdots, v_n]$为解向量，其中，如果物品$i$不在背包中，则$v_i = 0$，如果物品在背包中，则$v_i = 1$。目标是找出能放到背包中的物品的一个子集，使得

$$\sum_{i=1}^{n} w_i v_i \leqslant c$$

（即物品能放到背包中）且

$$\sum_{i=1}^{n} p_i v_i$$

为最大（即利润最大）。

求解这个问题的一个直接方法是，考虑所有2^n个可能的物品子集，并选出一个能放入背包且利润最大的子集。这里将使用DP形式，在$c = O(2^n/n)$时，速度比直接方法快。令$F[i, x]$是容量为x的背包中存放物品$\{1, 2, \cdots, i\}$时的最大利润，则$F[n, c]$就是问题的解。这个问题的DP形式如下：

520

$$F[i,x] = \begin{cases} 0 & x \geqslant 0, i = 0 \\ -\infty & x < 0, i = 0 \\ \max\{F[i-1,x], (F[i-1,x-w_i] + p_i)\} & 1 < i < n \end{cases}$$

这个递归公式将求出最大利润的背包。在当前的背包容量为x时，向背包中加入物品i的决策将导致两种情况：i) 不加入物品，背包容量仍然是x，利润不变；ii) 加入物品，背包容量变为$x-w_i$，利润增加p_i。DP算法根据是否能得到最大利润来决定是否加入物品。

这个DP形式的串行方法需要维持一个大小为$n \times c$的表F。表按以行为主的方式构建。算法开始时，对于不同容量的背包，只使用第一件物品作为最大利润，这对应于填充表的第一行。填充后面的行需要前一行的两项：一项来自同一列，另一项来自抵消物品的重量列。因此，计算任一项$F[i, j]$需要$F[i-1, j]$和$F[i-1, j-w_i]$，如图12-4所示。计算每一个项所需的时间为常数；这个算法的串行运行时间为$\Theta(n\,c)$。

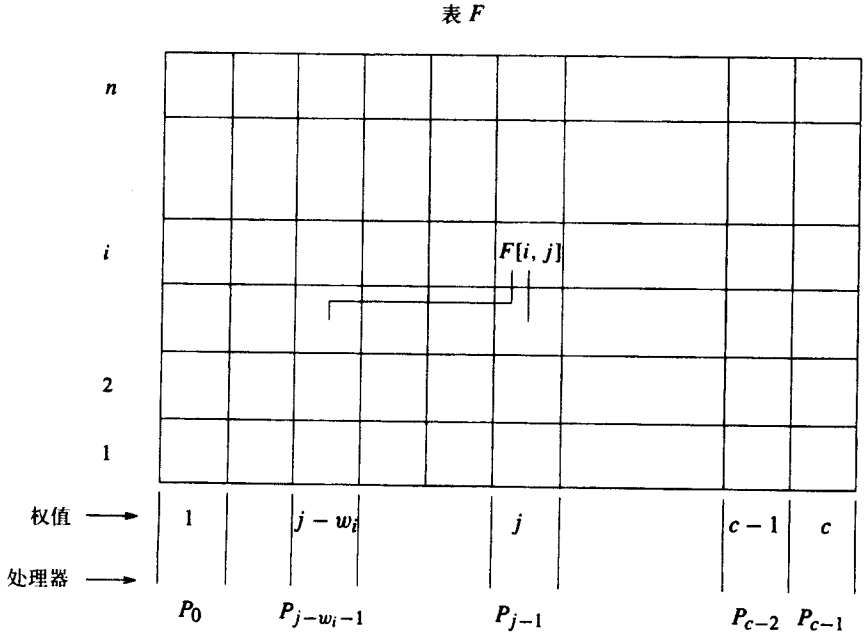


图12-4 在0/1背包问题中计算表F中的项。计算$F[i, j]$中的项需要和包含$F[i-1, j]$及$F[i-1, j-w_i]$的处理器通信

这个形式是串行一元形式。对于$i = 1, 2, \cdots, n$，子问题$F[i, j]$被组织到n层中。i层问题的计

算只依赖于$i-1$层的子问题。因此形式是串行的。由于$F[i, x]$的两个可替换解的每一个只依赖于一个子问题，该形式是一元的。进而，由于某一层的问题只依赖上一层的两个子问题，层间的依赖性是很小的。

考虑将这个算法在CREW PRAM上并行化，设CREW PRAM有c个处理器，标号从P_0到P_{c-1}。处理器P_{r-1}计算矩阵F的第r列。在迭代j中计算$F[j, r]$时，处理器P_{r-1}需要$F[j-1, r]$和$F[j-1, r-w_j]$的值。处理器p_{r-1}在常数时间内可以读取矩阵F中的任何元素，因此，计算$F[j, r]$也需要常数时间。所以每次迭代需要常数时间。由于一共有n次迭代，并行运行时间为$\Theta(n)$。该形式使用c个处理器，故处理器-时间乘积为$\Theta(nc)$。所以，算法是成本最优的算法。

现在在有c个处理器的分布式内存的计算机上考虑算法。表F在所有的处理器间分配，每个处理器负责计算一列，如图12-4所示。每个处理器本地存储所有物品的重量和利润。在第j次迭代中，要在处理器P_{r-1}计算$F[j, r]$，$F[j-1, r]$本地可得到，但$F[j-1, r-w_j]$需要从另一个处理器获取。这对应于4.6节所讲的w_j循环移位操作。如果有p个处理器，消息大小为m，网络有足够的带宽，则这种循环移位所需时间的上界为$(t_s + t_w m)\log p$。由于消息的大小只有一个字，且$p = c$，这个时间由$(t_s + t_w)\log c$表示。如果求和及求最大值操作需时t_c，则每次迭代所需时间为$t_c + (t_s + t_w)\log c$。由于这样的迭代有n次，总时间为$O(n \log c)$。这种形式的处理器-时间乘积为$O(n c \log c)$；因此，算法不是成本最优的算法。

如果增加每个处理器的元素个数，该算法又会是什么样的结果呢？使用p个处理器，在每次迭代中，每个处理器计算表中c/p个元素。在第j次迭代中，处理器P_0计算元素$F[j, 1], \cdots, F[j, c/p]$，处理器$P_1$计算元素$F[j, c/p+1], \cdots, F[j, 2c/p]$，以此类推。对于任意的$k$，计算$F[j, k]$都需要$F[j-1, k]$和$F[j-1, k-w_j]$的值，所需的表$F$中的数据可通过循环移位从远程处理器取得。根据$w_j$和$p$的值，需要的非本地数据可从一个或两个处理器处取得。注意，不管数据来自一个还是两个处理器，通过这些消息传送的字总数为c/p。假设c/p很大，且网络有足够的带宽（4.6节），这个操作所需的时间最多为$(2t_s + t_w c/p)$。由于每个处理器计算c/p个元素，每次迭代的总时间为$t_c c/p + 2t_s + t_w c/p$。因此，算法在n次迭代中的并行运行时间为$n(t_c c/p + 2t_s + t_w c/p)$。用渐近的说法，并行运行时间为$O(nc/p)$。它的处理器-时间乘积为$O(nc)$，是成本最优的。

521
522

这个并行形式的效率有一个上界，因为需要通信的数据量和计算量处于同一个数量级。这个上限由t_w和t_c的值决定(习题12.1)。

12.3 非串行一元DP形式

从两个序列中找出最长公共子序列的DP算法可用非串行一元DP形式表示。

最长公共子序列问题

给定一个序列$A = \langle a_1, a_2, \cdots, a_n \rangle$，$A$的子序列可从$A$中删除一些项得到。例如，$\langle a, b, z \rangle$是$\langle c, a, d, b, r, z \rangle$的子序列，但$\langle a, c, z \rangle$和$\langle a, d, l \rangle$则不是。最长公共子序列（longest-common-subsequence, LCS）问题可以表述如下：给定两个序列$A = \langle a_1, a_2, \cdots, a_n \rangle$及$B = \langle b_1, b_2, \cdots, b_m \rangle$，找出既是$A$又是$B$的子序列中最长的一个。例如，如果$A = \langle c, a, d, b, r, z \rangle$，$B = \langle a, s, b, z \rangle$，则$A$和$B$的最长公共子序列为$\langle a, b, z \rangle$。

令$F[i, j]$表示序列A中前i个元素和序列B中前j个元素的最长公共子序列的长度。LCS问题的目标是找出$F[n, m]$。这个问题的DP形式用$F[i-1, j-1]$、$F[i, j-1]$以及$F[i-1, j]$表示

$F[i,j]$如下:

$$F[i,j]=\begin{cases}0 & \text{如果 } i=0 \text{ 或 } j=0\\ F[i-1,j-1]+1 & \text{如果 } i,j>0 \text{ 且 } x_i=y_j\\ \max\{F[i,j-1],F[i-1,j]\} & \text{如果 } i,j>0 \text{ 且 } x_i\neq y_j\end{cases}$$

给定两个序列A和B，考虑两个指向序列起点的指针。如果两个指针指向的项相同，则它们是最长公共子序列的成员。因此，两个指针都可以移动到各自序列的下一项，最长公共子序列的长度也相应增加1。如果指针指向的项不同，则可能有两种情况：最长公共子序列可能由A的最长子序列及将指针移动到B的下一个项得到的序列构成，或者由B的最长子序列及将指针移动到A的下一个项得到的序列构成。由于我们要确定最长子序列，必须选择这两种情况中的最大值。

这种DP形式的串行实现按照以行为主的顺序计算表F中的值。由于表中每一项的计算量不变，该算法的整体复杂度为$\Theta(nm)$。如图12-5 a所示，这个DP算法是非串行一元的。把沿对角线的节点当成属于同一层，每个节点要依赖上一层的两个子问题及上两层的一个子问题。由于某一层任意子问题的解只是上一层某一个解的函数，这个形式是一元的。（注意，在公式(12-5)的第三种情况下，$F[i,j-1]$及$F[i-1,j]$都是$F[i,j]$的可能解，而这两者中的最大值是$F[i,j]$的最优解。）从图12-5看出，这个问题有非常规则的结构。 523

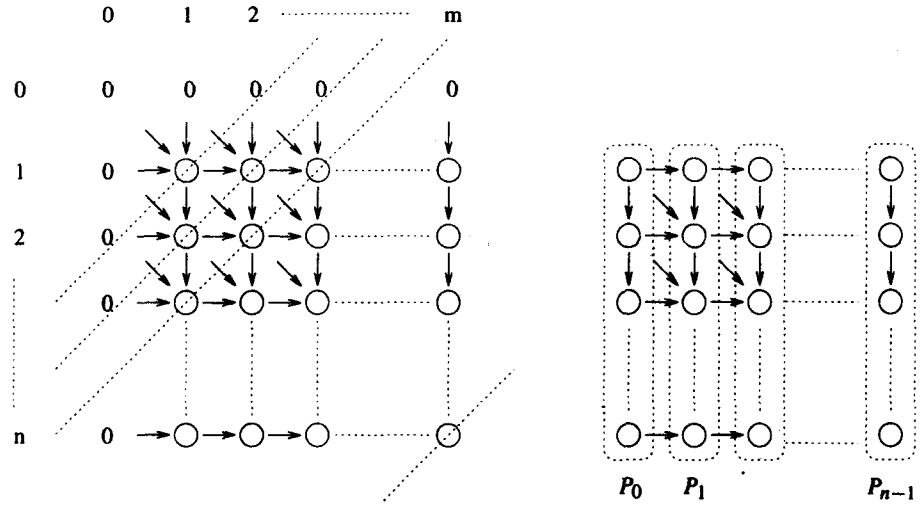


图12-5 最长公共子序列问题：a）计算表F中的项。计算沿虚斜线进行；b）映射表中的元素到处理器

例12.2 计算两个氨基酸序列的LCS

让我们来考虑两个氨基酸序列H E A G A W G H E E和P A W H E A E的LCS。对感兴趣的读者，对应的氨基酸名称为，A-丙胺酸，E-谷氨酸，G-甘氨酸，H-组氨酸，P-脯氨酸，W-色氨酸。图12-6中列出这两个序列中表F的项。通过找出最大值并枚举出所有的匹配，可得出这两个序列的LCS为A W H E E。 ■

为了简化讨论，我们只讨论$n = m$时的并行形式。考虑这种算法在n个处理器的CREW
524 PRAM上的并行形式。每个处理器P_i计算表F的第i列。从左上角到右下角按对角线扫描的方法计算表中的项。因为有n个处理器，每个处理器能存取表F中的任何项，计算每条对角线上的元素需要的时间为定值（对角线最多能包含n个元素）。由于这样的对角线有$2n-1$条，算法需要$\Theta(n)$次迭代。因此，并行运行时间为$\Theta(n)$。算法是成本最优的，因为它的处理器-时间乘积为$\Theta(n^2)$，等于串行复杂度。

		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1	1	1	1	1
W	0	0	0	1	1	1	2	2	2	2	2
H	0	1	1	1	1	1	2	2	3	3	3
E	0	1	2	2	2	2	2	2	3	4	4
A	0	1	2	3	3	3	3	3	3	4	4
E	0	1	2	3	3	3	3	3	3	4	5

图12-6 用于计算序列H E A G A W G H E E和P A W H E A E的LCS的表F

这个算法也可用在含n个处理器的逻辑线性阵列中，在不同的处理器间分配表F。注意使用2.7.1节讲述的嵌入技术，这种逻辑拓扑结构可以映射到多种物理结构中。处理器P_i存储表中的第$(i+1)$列。表F中的项按图12-5b所示分配到处理器中。计算$F[i, j]$的值时，处理器P_{j-1}需要从它左边的处理器得到$F[i-1, j-1]$或$F[i, j-1]$的值。它从相邻的处理器传送一个字需时$t_s + t_w$。要计算表中的每个项，处理器都需要从它的直接邻居处得到一个值，然后才能进行计算，所需时间为t_c。由于每个处理器计算对角线上的一个项，每次迭代所需时间为$(t_s + t_w + t_c)$。算法对整个表作$(2n-1)$次对角线扫描（迭代），因此，总并行运行时间为

525
$$T_P = (2n-1)(t_s + t_w + t_c)$$

由于串行运行时间为n^2t_c，这个算法的效率为

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

仔细研究这个表达式可以看出，不可能获得超过某一阈值的效率。为了计算这个阈值，假设处理器间的值同时通信是可能的；即$t_s = t_w = 0$。在这种情况下，并行算法的效率为

$$E_{max} = \frac{1}{2 - 1/n} \tag{12-5}$$

因此，效率的上界为0.5。即使有多个列映射到一个处理器，这个上界也不改变。使用其他的映射方法则可达到更高的效率（习题12-3）。

注意这个算法允许有效的并行形式的基本特性，在于能够这样划分表F，使得计算每个元素时只需从相邻的处理器取得数据。换句话说，算法显示数据存取的本地性。

12.4 串行多元DP形式

Floyd全部顶点对间的最短路径算法可以改用串行多元DP形式表示。

Floyd全部顶点对间的最短路径算法

考虑加权图G，它包含节点集合V和边集合E。E中从节点i到节点j的边的权为$c_{i,j}$。Floyd算法确定集合V中全部顶点对(i, j)间的最短路径的成本$d_{i,j}$（10.4.2节）。路径的成本是路径中所有边的权值之和。

令从节点i到节点j的路径的最小成本为$d_{i,j}^k$，只使用节点$v_0, v_1, \cdots, v_{k-1}$。这个问题DP形式的泛函方程为

$$d_{i,j}^k = \begin{cases} c_{i,j} & k = 0 \\ \min\{d_{i,j}^{k-1}, (d_{i,k}^{k-1} + d_{k,j}^{k-1})\} & 0 < k < n-1 \end{cases} \tag{12-6}$$

由于$d_{i,j}^n$是使用全部n个节点时从节点i到节点j的最短路径，它也是节点i到节点j间总的最短路径成本。这个算法的串行形式需要进行n次迭代，每次迭代需时$\Theta(n^2)$。因此，串行算法的整体运行时间为$\Theta(n^3)$。

公式(12-6)表示的是串行多元形式。节点$d_{i,j}^k$可以分成n层，每个k值一层。$k+1$层上的元素只依赖k层上的元素。因此，该形式是串行的。由于$d_{i,j}^k$的一个解需要用到上一层两个子问题$d_{i,k}^{k-1}$和$d_{k,j}^{k-1}$解的组合，该形式是多元的。而且，层与层之间的依赖性很小，因为计算$d_{i,j}^{k+1}$中的每个元素只需要上一层的三个结果（一共有n^2个结果）。 526

这个算法的一个简单CREW PRAM形式使用n^2个处理器，处理器被组织成一个逻辑二维数组，其中处理器$P_{i,j}$计算$d_{i,j}^k$的值，$k = 1, 2, \cdots, n$。在每次迭代k中，处理器$P_{i,j}$需要用到$d_{i,j}^{k-1}$，$d_{i,k}^{k-1}$和$d_{k,j}^{k-1}$的值。有了这些值后，算法在常数时间计算$d_{i,j}^k$的值。因此，PRAM并行形式的并行运行时间为$\Theta(n)$。由于处理器-时间乘积等于串行运行时间$\Theta(n^3)$，这个形式是成本最优的。这个算法能应用在许多实际体系结构中，得出有效的并行形式（10.4.2节）。

与串行一元形式一样，数据本地性也是串行多元形式的本质，因为在许多这样的形式中，层与层之间的联系非常少。

12.5 非串行多元DP形式

在非串行多元DP形式中，除了并行处理同一层上的子问题外，也可使用流水线方式计算提高效率。下面用最优矩阵带括号问题说明这种形式。

最优矩阵带括号问题

考虑将n个矩阵相乘的问题，n个矩阵分别为$A_1, A_2, \cdots, A_n$，其中A_i是具有r_{i-1}行和r_i列的矩阵。矩阵间相乘的顺序对用来求乘积的计算量总和有重要的影响。

例12.3 最优矩阵带括号问题

考虑三个矩阵A_1、A_2和A_3，其大小分别为10 × 20、20 × 30和30 × 40。这三个矩阵的乘积可通过$(A_1 \times A_2) \times A_3$或$A_1 \times (A_2 \times A_3)$计算。如果使用$(A_1 \times A_2) \times A_3$，则计算$(A_1 \times A_2)$需要10 × 20 × 30次操作，得到大小为10 × 30的矩阵。将它乘以A_3还需要10 × 30 × 40操作。因此总操作次数为10 × 20 × 30 + 10 × 30 × 40 = 18 000。同样，计算$A_1 \times (A_2 \times A_3)$需要20 × 30 × 40 + 10 × 20 × 40 = 32 000次操作。很明显，第一种带括号方法是可取的。 ■

带括号问题的目标是为了确定括号的位置，使得操作次数最少。枚举出所有可能的带括
527 号方法是不可行的，因为这种可能性的数量是指数级的。

令$C[i, j]$是矩阵A_i, ..., A_j相乘的最优成本。这条矩阵链可以表示为两条更小的矩阵链A_i, A_{i+1}, ..., A_k及A_{k+1}, ..., A_j的乘积。链A_i, A_{i+1}, ..., A_k导致维数是$r_{i-1} \times r_k$的矩阵，而链A_{k+1},..., A_j导致维数是$r_k \times r_j$的矩阵。将这两个矩阵相乘的成本为$r_{i-1} r_k r_j$。因此，带括号$(A_i, A_{i+1}, ..., A_k)(A_{k+1}, ..., A_j)$的成本由$C[i, k] + C[k+1, j] + r_{i-1} r_k r_j$给出。这样就推导出带括号问题的如下递推关系：

$$C[i, j] = \begin{cases} \min\limits_{i \le k < j} \{C[i, k] + C[k+1, j] + r_{i-1} r_k r_j\} & 1 \le i < j \le n \\ 0 & j = i, 0 < i \le n \end{cases} \tag{12-7}$$

有了公式(12-7)，问题就简化为求解$C[1, n]$的值。矩阵链的成本组合如图12-7所示。用自底向上的方法构建存储$C[i, j]$的值的表C，可以求解公式(12-7)。算法填充表C的顺序与求解在不断加长的矩阵链上带括号的问题相对应。将它想象成按对角线填充则更形象化（图12-8）。对角线l中的项与将长度为$l+1$的矩阵链相乘的成本对应。从公式(12-7)中可以看出，$C[i, j]$的值用$\min\{C[i, k] + C[k+1, j] + r_{i-1} r_k r_j\}$计算，其中$k$的取值范围从$i$到$j-1$。因此，计算$C[i, j]$需要求$(j-i)$个值，并选出其中的最小值。计算每一个值需时$t_c$，计算$C[i, j]$需时$(j-i)t_c$。因此，对角线$l$上的每一个项都能在$lt_c$时间内计算出。

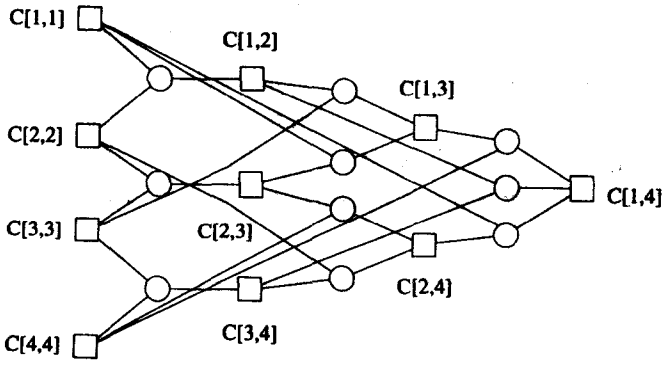


图12-7 使用一种非串行多元形式寻找含4条矩阵链的最优矩阵带括号方法。方形节点表示一个矩阵链相乘的最优成本。圆形节点表示一种可能的带括号方法

计算最优带括号序列的成本时，算法要计算长度为2的$(n-1)$条链，需时$(n-1)t_c$。同样，计
528 算$(n-2)$条长度为3的链需时$(n-2)2t_c$。在最后一步，算法计算长度为n的一条链，需时$(n-1)\ t_c$。

因此，这个算法的串行运行时间为

$$\begin{aligned} T_S &= (n-1)t_c + (n-2)2t_c + \cdots + 1(n-1)t_c \\ &= \sum_{i=1}^{n-1}(n-i)it_c \\ &\simeq (n^3/6)t_c \end{aligned} \tag{12-8}$$

算法的串行复杂度为$\Theta(n^3)$。

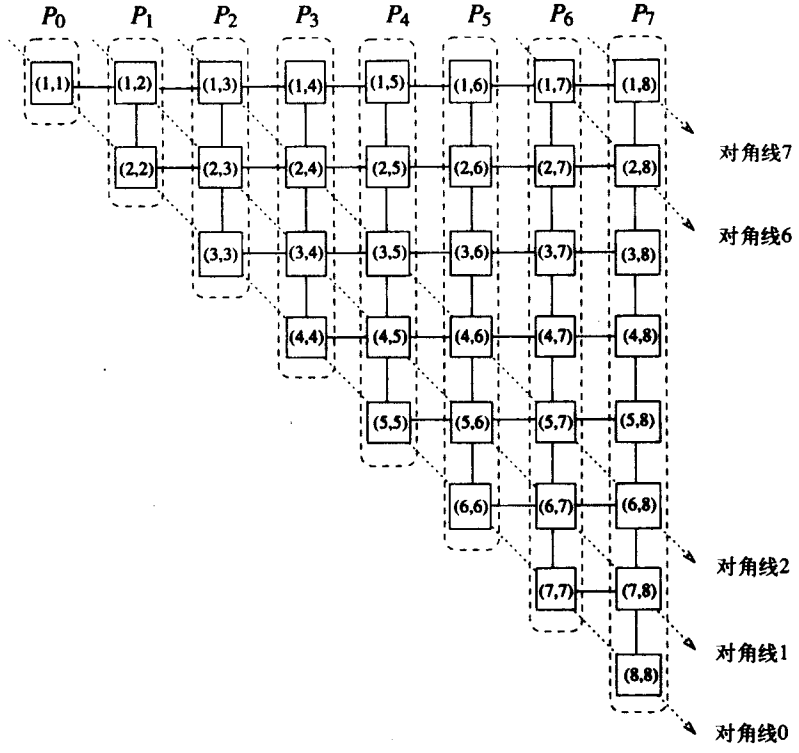


图12-8 计算最优矩阵带括号问题的对角线顺序

考虑这个算法在含n个处理器的逻辑环中的并行形式。在第l步，每个处理器计算属于第l条对角线的一个元素。处理器P_i计算表C中的第$(i+1)$列。图12-8说明这种表在处理器间的划分。在计算表C中元素的赋值后，每个处理器用多对多广播（4.2节）对所有其他处理器发送它的值。因此，下一次迭代中的赋值可以在本地计算。在第l次迭代中，计算表C中的一个项需时lt_c，因为它对应于长度为$l+1$的链相乘的成本。在n个处理器中，对一个字进行多对多广播需时$t_s \log n + t_w(n-1)$（4.2节）。计算对角线l的项所需的总时间为$lt_c + t_s \log n + t_w(n-1)$。并行运行时间是对$n-1$条对角线进行计算所需的时间之和: 529

$$\begin{aligned} T_P &= \sum_{l=1}^{n-1}(lt_c + t_s \log n + t_w(n-1)) \\ &= \frac{(n-1)(n)}{2}t_c + t_s(n-1)\log n + t_w(n-1)^2 \end{aligned}$$

这个算法的并行运行时间为$\Theta(n^2)$。由于处理器-时间乘积为$\Theta(n^3)$，与串行复杂度相同，这个算法是成本最优的算法。

当使用按逻辑环方式组织的p个处理器($1 \leqslant p \leqslant n$)时，如果对角线上有$n$个节点，则每个处理器存储$n/p$个节点。每个处理器计算分配给它的项的成本$C[i, j]$。计算后，使用多对多广播将最近一次计算的对角线上的子问题的成本发送给所有其他的处理器。由于每个处理器中都有前一条对角线上子问题成本的完全信息，不需要其他的通信。对n/p个字进行多对多广播所需时间为$t_s \log p + t_w n(p-1)/p \approx t_s \log p + t_w n$。计算第$l$条对角线的表中的$n/p$个项需时$lt_c n/p$。并行运行时间为

$$\begin{aligned} T_P &= \sum_{l=1}^{n-1} (lt_c n/p + t_s \log p + t_w n) \\ &= \frac{n^2(n-1)}{2p} t_c + t_s(n-1)\log p + t_w n(n-1) \end{aligned}$$

用数量级的说法，$T_P = \Theta(n^3/p) + \Theta(n^2)$。这里，$\Theta(n^3/p)$是计算时间；$\Theta(n^2)$是通信时间。如果$n$与$p$相比足够大，则通信时间与计算时间相比只占很小的部分，这样就能导致线性加速比。

这种形式最多能使用$\Theta(n)$个处理器在$\Theta(n^2)$时间内完成计算任务。如果以流水线方法用$n(n+1)/2$个处理器计算$C[i, j]$，则时间还会缩短。每个处理器计算矩阵C的一个项$c(i, j)$。由于问题的非串行性质，可使用流水线方法。计算对角线t上的项不仅仅需要对角线$t-1$上的项，还需要所有更前面对角线中的项。因此，对角线t的计算甚至可以在对角线$t-1$上的计算完成前开始。

12.6 综述与讨论

本章提出推导使用动态规划的并行算法的框架，既指出并行化可能的来源，又说明在何
530 种条件下才能有效地使用并行化。

通过把计算表示成图的形式，我们指出并行化的三个来源。第一，计算单个子问题（某层上的一个节点）的成本可以并行化。例如，计算如图12-3所示的多级图中的最短路径，由于节点计算的复杂度本身是$\Theta(n)$，对节点的计算可以并行化。但是，对于很多问题来说，节点计算的复杂度较低，限制了并行化的使用。

第二，每一层上的子问题可以并行求解。这就给从很大一类问题（包含本章的所有问题）获取并行提供了可行的方法。

串行与非串行形式都可以使用前两种来源的并行化。非串行形式还可使用第三种来源的并行化：将计算在不同层间用流水线方法进行。流水线方法能够在某一问题所依赖的子问题求解时，马上开始求解该问题。这种形式的并行用于带括号问题中。

注意，流水线方法也能应用在Floyd全部顶点对间的最短路径算法（10.4.2节）的并行形式中。正如10.4.2节所讨论的那样，这个算法与一串行DP形式对应。这个算法的流水线性质与非串行DP形式的性质不同。在Floyd算法的流水线版本中，某一级的计算以及前几级间的通信都用流水线方法实现。如果通信成本为0（如在PRAM中），则Floyd算法并不能从流水线方法中获得好处。

数据本地性的重要性贯穿全章。如果某一问题的解依赖于其他子问题的解，则传送结果的成本必须低于求解问题的成本。在某些问题中（如0/1背包问题），其本地性程度要远远小

于其他问题（如最长公共子序列问题以及Floyd全部顶点对间的最短路径问题）的本地性。

12.7 书目评注

动态规划最早是由Bellman[Bel57]提出的，用于求解多级决策问题。此后人们为DP开发了多种形式模型[KH67, MM73, KK88b]。不少教材和文章也对最长公共子序列问题、矩阵链乘法问题、0/1背包问题以及最短路问题提出了串行DP形式[CLR90, HS78, PS82, Bro79]。

Li和Wah[LW85, WL88]提出，一元串行DP形式可在脉动阵列中用矩阵-向量相乘的并行形式求解。通过将该问题表示成矩阵-矩阵相乘，他们还提出了并发性更好但不是成本最优的形式。Ranka和Sahni[Rs90b]为字符串编辑问题提出多元串行形式，并导出基于棋盘划分的并行形式。

一大类优化问题的DP形式与最优矩阵带括号问题类似。这些问题包括最优多边形三角化 531
问题、最优二叉查找树[CLR90]问题以及CYK句法分析问题[AU72]。所有这些问题的标准DP形式的串行复杂度为$\Theta(n^3)$。在Ibarra等[IPS91]提出的几个并行形式中，在超立方体上使用$\Theta(n)$个处理器，能在$\Theta(n^2)$时间内求解问题。Guibas，Kung和Thompson[GKT79]提出一种脉动算法，能使用$\Theta(n^2)$个处理器在$\Theta(n)$时间内求解问题。Karypis和Kumar[KK93]分析由Guibas等[GKT79]提出的脉动算法的三种不同映射方法，并通过求解矩阵相乘带括号问题，用实验方法评价三种映射方法。他们指出，如果将这个算法直接映射到格网结构中，则效率的上界为1/12。对于矩阵带括号问题，他们还提出了另一种更好的映射方法，即将格网映射到256个处理器的超立方体中，这种方法既没有上一种方法的缺点，还能带来近乎线性的加速比。

对于求解带括号问题，人们提出了许多快速并行算法，但这些算法往往不是成本最优的，且只能应用于像PRAM这样的理论模型中。例如，Valiant等[VSBR83]为并行化求解这类问题提出了一种通用方法，使用$O(n^9)$个处理器，运行时间为$O(\log^2 n)$。Rytter[Ryt88]通过在树中使用并行卵石博弈，能够在CREW PRAM中将处理器的数量减少到$O(n^6/\log n)$，在超立方体中减少到$O(n^6)$，而运行时间只需$O(\log^2 n)$。Huang等[HLV90]在CREW PRAM模型中提出一个类似的算法，使用$O(n^{3.5}\log n)$个处理器，运行时间为$O(\sqrt{n}\log n)$。DeMello 等[DCG90]在Cray计算机中使用向量化DP形式求解最优控制问题。

从本章可以看出，0/1背包问题的串行多元形式很难并行化，因为缺少通信本地性。Lee等[LSS88]利用背包问题的特殊性质，在MIMD消息传递计算中，导出并行化背包问题DP算法的分治策略（习题12.2）。Lee等用实验说明，对于超立方体中问题的大量实例可以获得线性加速比。

习题

12.1 考虑12.2.2节中求解0/1背包问题的并行算法。导出这个算法的加速比及效率。证明，对于固定数目的处理器，通过增大背包问题规模，这个算法的效率不可能超过某个值。对于这种形式，效率作为t_w和t_c的函数，其上界是多少？

12.2 [LSS88]在12.2.2节讨论的0/1背包问题的并行形式中，并发度与背包的容量c成比例。由于待传送的数据量和每个处理器的计算量处于同一数量级，这个算法的数据本地性较差。 532
Lee等提出另一种形式，其中并发度与物品的重量n成比例。这种形式的数据本地性也好得多。在这种形式中物品的重量在处理器间划分，对于最大为c的不同容量背包，每个处理器计算分

配给自己的本地重量所能得到的最大利润。这个信息用表来表示，表合并后将得到全局解。

试计算这种形式的并行运行时间、加速比以及效率。比较这个算法的性能与12.2.2节讲述的算法的性能。

12.3 我们提到最长公共子序列问题并行形式的效率上界为0.5。对于这个问题，采用另一种映射方法以可得到更高的效率。试推导出一种效率不受此上界约束的形式，并给出这种形式的运行时间。

提示：考虑3.4.1节中讲述的块-循环映射。

12.4 [HS78] 旅行商问题（TSP）定义如下：给定城市的集合及每两个城市间的距离，确定经过所有城市的最短旅行路线。经过所有城市的路线是指经过每个城市一次并回到出发点。旅行的距离是所有经过的距离之和。

这个问题可用DP形式求解。将城市看作图$G(V, E)$中的顶点。将城市集V用$\{v_1, v_2, \ldots, v_n\}$表示，并令$S \subseteq \{v_2, v_3, \ldots, v_n\}$。此外令$c_{i,j}$为城市$i$和城市$j$间的距离。如果$f(S, k)$表示从城市$v_1$出发，经过$S$中的所有城市，并停止在城市$k$，则可用下面的递归公式计算$f(S, k)$：

$$f(S,k)=\begin{cases} c_{1,k} & S=\{k\} \\ \min\limits_{m\in S-\{k\}}\{f(S-\{k\},m)+c_{m,k}\} & S\neq\{k\} \end{cases} \tag{12-9}$$

试根据公式(12-9)导出并行形式。计算该形式的并行运行时间和加速比。这个并行形式是成本最优的吗？

12.5 [HS78]考虑将含$O(n)$和$O(m)$个记录的两个有序文件的合并问题。这些文件能在时间$O(m+n)$内合并到一个有序文件中。假设有r个这样的文件，将它们合并到一个有序文件中可以表示成文件对间的一系列合并操作。合并操作的整体成本是合并序列的函数。最优合并顺序可用贪婪问题来表示，其并行形式可用本章讲述的原则推导出。

试对这个问题写出递归公式。导出使用p个处理器合并文件的并行形式。计算并行运行时间及加速比，并判断并行形式是否为成本最优的。

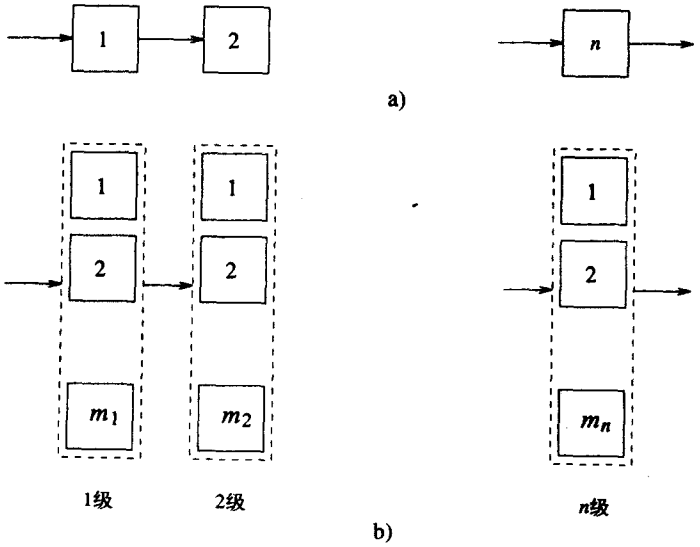


图12-9 a）n个设备串联的电路；b）电路中的每一级有m_i个功能部件，共有n个串联的级

12.6 [HS78] 考虑容错电路的设计问题，电路中包含n个串联设备，如图12-9a所示。如果每一个设备的出错概率为f_i，则电路的总出错概率为$1-\prod(1-f_i)$。这里，$\prod$表示对指定项的乘积。如果将多个功能设备在每级并行连接，如图12-9b所示，就可以提高电路的可靠性。如果电路中第i级有r_i个重复的功能部件，每个部件的出错概率为f_i，则这一级总出错概率减小为$r_i^{m_i}$，电路的总出错概率为$1-\prod(1-r_i^{m_i})$。一般情况下，由于物理原因，某一层的出错概率可能不是$r_i^{m_i}$，而是某个函数$\phi_i(r_i, m_i)$。问题的目标是使电路总出错概率$1-\prod(1-\phi_i(r_i,m_i))$达到最小。 533

使用成本对问题增加新的维。如果第i级使用的每个功能部件的成本为c_i，则由于成本约束，总成本$\sum c_i m_i$应该小于一固定值c。

问题可从形式上定义为

$$\text{最小化 } 1-\prod(1-\phi_i(r_i,m_i))\text{，使得 } \sum c_i m_i < c$$

其中，$m_i > 0$且$0 < i \leqslant n$。

令$f_i(x)$代表成本x的i个阶段的系统可靠性。最优解为$f_n(c)$。$f_i(x)$的递归公式如下：

$$f_i(x)=\begin{cases}1 & i=0\\ \max\limits_{i<m_i<u_i}\{\phi_i(m_i)f_{i-1}(c-c_i m_i)\} & i>1\end{cases} \qquad (12\text{-}10)$$ 534

将这种形式归到四类DP形式中的一类，并导出这种算法的并行形式。确定它的并行运行时间、加速比以及等效率函数。

12.7 考虑简化的最优多边形三角剖分问题。这个问题可定义如下：给定一简单多边形，通过用弦连接多边形的顶点，将多边形分成多个三角形。这个过程如图12-10所示。用顶点v_i，v_j，v_k构建三角形的成本由一个函数$f(v_i, v_j, v_k)$定义。对于这个问题，令成本为三角形的边的总长度（使用欧几里得距离）。最优多边形三角剖分问题将多边形分成多个三角形，使得所有三角形的总长度（三角形每个长度之和）最小。请对这个问题给出DP形式，把它归到四类DP形式中的一类，并对p个处理器推导出并行形式。确定它的并行运行时间、加速比以及等效率函数。

提示：该问题与最优矩阵带括号问题类似。

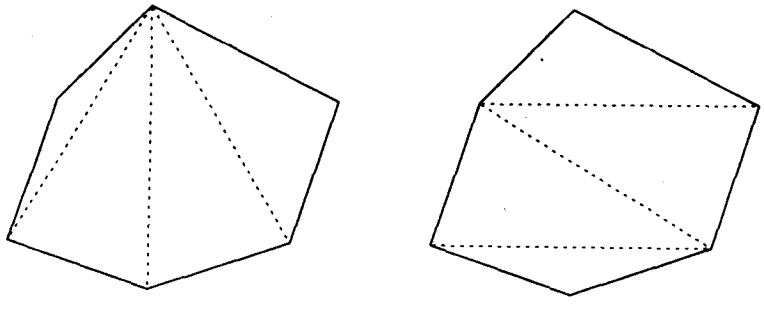


图12-10 规则多边形两种可能的三角剖分 535

第13章 快速傅里叶变换

离散傅里叶变换（DFT）在许多科学和技术应用中发挥着重要的作用，如时间序列和波形分析、线性偏微分方程求解、卷积、数字信号处理以及图像滤波等。离散傅里叶变换是一种线性变换，它把某个周期性信号（如正弦波）单个周期中的n个规则抽样点映射到相同数目的点上，这些点表示这个信号的频谱。1965年，Cooley和Tukey提出一个运算量为$\Theta(n \log n)$的算法，计算任一n点序列的离散傅里叶变换。与过去已知的运算量为$\Theta(n^2)$的其他计算离散傅里叶变换的算法比较，他们的新算法有了显著的改进。Cooley和Tukey的这一革命性的算法及其变体称为*快速傅里叶变换*（Fast Fourier Transform, FFT）。由于它在科学与工程领域的广泛运用，在并行计算机上实现快速傅里叶变换一直倍受关注。

快速傅里叶变换算法有几种不同的形式。本章讨论它的最简形式：一维的、无序的和基数为2的快速傅里叶变换。更高基数的、多维的快速傅里叶变换的并行形式与本章讨论的简单算法类似，因为所有串行快速傅里叶变换的基本思想是一样的。有序的快速傅里叶变换通过对无序快速傅里叶变换的输出序列使用位反转（13.4节）得到。位反转并不影响快速傅里叶变换并行实现的整体复杂度。

本章讨论基本算法的两种并行形式：*二进制交换算法*（binary-exchange algorithm）和*转置算法*（transpose algorithm）。在实际运用中，根据输入n的大小、进程的数目p以及内存和网络的带宽大小，这两种算法中的某一个可能比另一个运行更快。

13.1 串行算法

考虑一个长度为n的序列$X = \{X[0], X[1], \cdots, X[n-1]\}$。序列$X$的离散傅里叶变换是序列$Y = \{Y[0], Y[1], \cdots, Y[n-1]\}$，其中

$$Y[i] = \sum_{k=0}^{n-1} X[k]\omega^{ki}, \quad 0 \leq i < n \tag{13-1}$$

在公式(13-1)中，ω是复平面上的第n个单位原根；即$\omega = e^{2\pi\sqrt{-1}/n}$，$e$是自然对数的底数。更一般地说，公式中$\omega$的幂可看作是以模为$n$的整数有限交换环的元素。在快速傅里叶变换计算中用到的ω的幂也称为*旋转因子*（twiddle factor）。

按照公式(13-1)计算每个$Y[i]$时需要进行n步复数乘法运算。因此，计算整个长度为n的序列Y的串行复杂度为$\Theta(n^2)$。下面要讲的快速傅里叶变换算法将这一复杂度降至$\Theta(n \log n)$。

假设n是2的幂。快速傅里叶变换算法通过下面步骤把一个n点的离散傅里叶变换计算分裂成两个$(n/2)$点的离散傅里叶变换计算：

$$\begin{aligned} Y[i] &= \sum_{k=0}^{(n/2)-1} X[2k]\omega^{2ki} + \sum_{k=0}^{(n/2)-1} X[2k+1]\omega^{(2k+1)i} \\ &= \sum_{k=0}^{(n/2)-1} X[2k]e^{2(2\pi\sqrt{-1}/n)ki} + \sum_{k=0}^{(n/2)-1} X[2k+1]\omega^{i}e^{2(2\pi\sqrt{-1}/n)ki} \end{aligned}$$

$$= \sum_{k=0}^{(n/2)-1} X[2k]e^{2\pi\sqrt{-1}ki/(n/2)} + \omega^i \sum_{k=0}^{(n/2)-1} X[2k+1]e^{2\pi\sqrt{-1}ki/(n/2)} \tag{13-2}$$

538 令 $\tilde{\omega} = e^{2\pi\sqrt{-1}/(n/2)} = \omega^2$；就是说，$\tilde{\omega}$ 是第$(n/2)$个单位原根。因此我们可以把公式(13-2)重写成如下形式:

$$Y[i] = \sum_{k=0}^{(n/2)-1} X[2k]\tilde{\omega}^{ki} + \omega^i \sum_{k=0}^{(n/2)-1} X[2k+1]\tilde{\omega}^{ki} \tag{13-3}$$

在公式(13-3)中，右端的两个求和分别是对$(n/2)$点的离散傅里叶变换计算。若n是2的幂，可类似地把每个这样的离散傅里叶变换计算用递归的方式分成更小的计算。这就得到算法13-1所给出的递归FFT算法。这个快速傅里叶变换算法称为基数为2的算法，因为每次递归时都把输入序列分为两等分。

算法13-1 递归的一维、无序和基数为2的FFT算法，其中$\omega = e^{2\pi\sqrt{-1}/n}$

```
1.  procedure R_FFT(X, Y, n, ω)
2.  if (n = 1) then Y[0] := X[0] else
3.  begin
4.      R_FFT(⟨X[0], X[2], ..., X[n − 2]⟩, ⟨Q[0], Q[1], ..., Q[n/2]⟩, n/2, ω²);
5.      R_FFT(⟨X[1], X[3], ..., X[n − 1]⟩, ⟨T[0], T[1], ..., T[n/2]⟩, n/2, ω²);
6.      for i := 0 to n − 1 do
7.          Y[i] := Q[i mod (n/2)] + ω^i T[i mod (n/2)];
8.  end R_FFT
```

图13-1说明如何用这个递归算法对8点的序列进行计算。如图所示，对应于算法13-1中第7行的第一组计算发生在递归的最深层。在这一层中，序列中下标的差为$n/2$的元素被用于计算中。在每个子序列的层中，用于计算的元素的下标差减小一半。图中也显示出每步计算中ω的幂。

计算快速傅里叶变换算法时，输入序列的大小在递归的每一层减小一半（见算法13-1的
539 第4、5行）。因此，对一个长度为n的初始序列来说，递归的层数最多为$\log n$。在递归的第m层，共计算了2^m个每个大小为$n/2^m$的快速傅里叶变换。因此每层的算术运算（见第7行）的总次数是$\Theta(n)$，而整个串行算法的复杂度为$\Theta(n \log n)$。

这种串行快速傅里叶变换也可用迭代的方式计算。迭代形式的并行实现比较容易表述。因此，在描述并行快速傅里叶变换算法之前，我们先给出串行算法的迭代形式。迭代快速傅里叶变换通过计算每个递归层，从最深层开始进行迭代导出。算法13-2给出对n点、一维、无序和基数为2的快速傅里叶变换的经典的迭代Cooley-Tukey算法。程序将第5行开始的外层循环迭代$\log n$次。迭代算法中循环下标m的值对应于递归算法中第$(\log n - m)$层递归（图13-1）。正如在递归中的每一层一样，每个迭代执行n次复数的乘法和加法。

算法13-2有两个主循环。始于第5行的外层循环对n点的串行快速傅里叶变换执行$\log n$次，而始于第8行的内层循环在外层循环的每次迭代中执行n次。内层循环的所有运算都是固定时间长度的算术运算。因此，本算法的串行时间复杂度为$\Theta(n \log n)$。在外层循环每次迭代时，

序列R都用前一次迭代中储存在序列S中的元素更新。第一次迭代时的输入序列X作为初始序列R。最终迭代的更新序列R即为所要求的傅里叶变换，并复制到输出序列Y中。

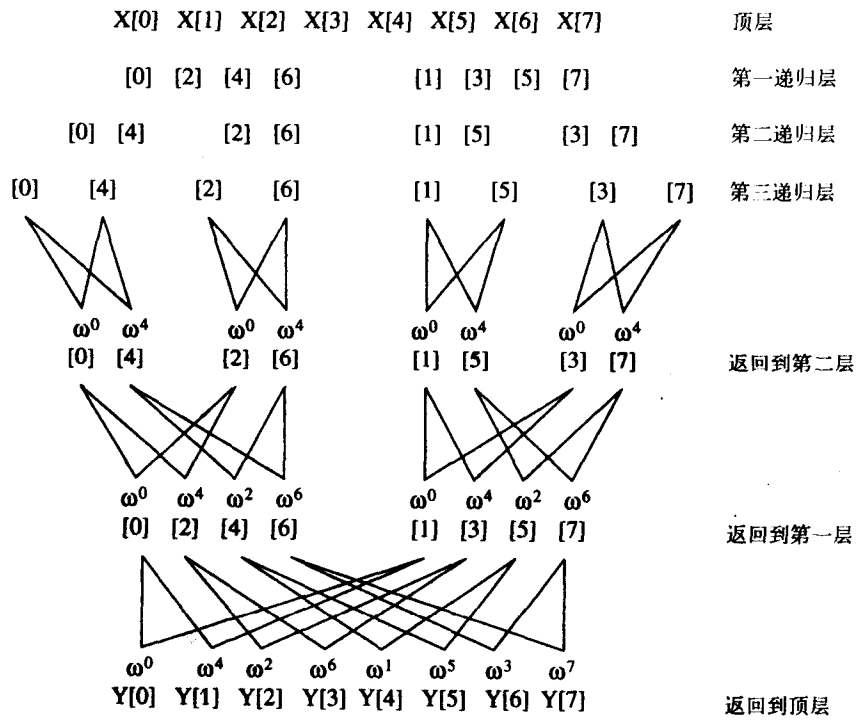


图13-1 递归的8点无序的FFT计算

算法13-2 Cooley-Tukey的一维、无序和基数为2的FFT算法，其中$\omega = e^{2\pi\sqrt{-1}/n}$

```
1.  procedure ITERATIVE_FFT(X, Y, n)
2.  begin
3.      r := log n;
4.      for i := 0 to n − 1 do R[i] := X[i];
5.      for m := 0 to r − 1 do          /* Outer loop */
6.          begin
7.              for i := 0 to n − 1 do S[i] := R[i];
8.              for i := 0 to n − 1 do  /* Inner loop */
9.                  begin

    /* Let (b0b1 ··· br−1) be the binary representation of i */

10.                     j := (b0 ... bm−1 0 bm+1 ··· br−1);
11.                     k := (b0 ... bm−1 1 bm+1 ··· br−1);
12.                     R[i] := S[j] + S[k] × ω^(bm bm−1 ··· b0 0 ··· 0);
13.             endfor;  /* Inner loop */
14.     endfor;          /* Outer loop */
15.     for i := 0 to n − 1 do Y[i] := R[i];
16. end ITERATIVE_FFT
```

算法13-2第12行执行本算法的一步关键操作：用$S[j]$和$S[k]$更新$R[i]$。下标j和k通过下述步骤从i中得到：假设$n = 2^r$。由于$0 \leqslant i < n$，i的二进制表示有r位。令$(b_0 b_1 \ldots b_{r-1})$为下标i的二进制表示。在外层循环$(0 \leqslant m < r)$的第m次迭代中，下标j在置i的第m个最高有效位（即b_m）为0时产生。下标k在置b_m为1时产生。因此，j和k的二进制表示仅在于它们的第m个最高有效位不同。在i的二进制表示中，b_m非0即1。因此j和k这两个下标至少有一个与下标i相同，这取决于b_m为0还是1。在外层循环的第m次迭代中，对每个在0到$n-1$间的i，将$S[i]$和另一个下标仅与i在最高有效位上不同的另一S的元素值代入算法13-2第12行，执行后得到$R[i]$。图13-2说明当$n = 16$时这些元素的配对模式。

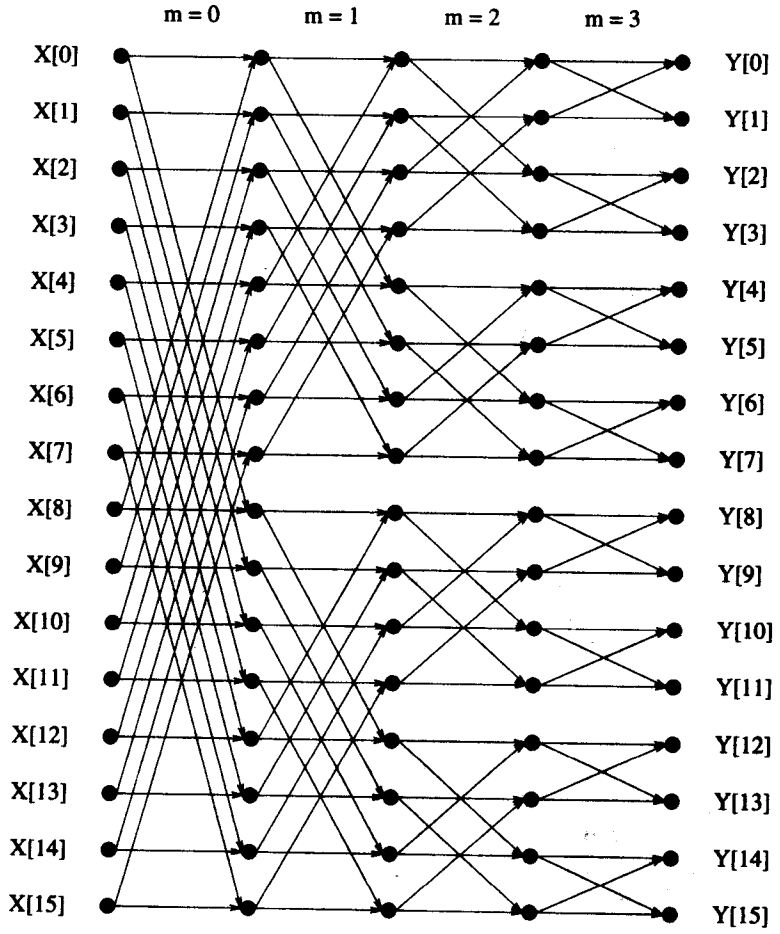


图13-2 16点无序FFT计算中输入序列及中间结果序列元素的组合模式

13.2 二进制交换算法

本节讨论在并行计算机上实现快速傅里叶变换的二进制交换算法。首先，由划分输入或输出向量导入一个分解。因此，每个任务从输入向量的一个元素开始，来计算输出的对应元素。若指定每个任务的标号与其输入或输出元素下标相同，则在算法$\log n$次迭代的每一次迭代中，数据交换出现在标号仅相差一位的一对任务之间。

13.2.1 全带宽网络

在本小节，我们介绍二进制交换算法在并行计算机上的实现。这种并行计算机对p个并行进程可用的对分带宽为$\Theta(p)$（见2.4.4节）。由于在并行快速傅里叶变换中，任务间的交互模式与超立方体网络中的模式是匹配的，我们描述算法时采用这样一种互连网络。然而，在其他任何整体同时数据传输容量为$O(p)$的并行计算机上，这种性能和可扩展性的分析是有效的。

1. 每个进程一个任务

我们先来考虑将一个任务简单地映射到每个进程上。图13-3说明当$n = 16$时，采用二进制交换算法的这种映射方式导致的交互模式。如图所示，进程i $(0 < i < n)$开始时储存$X[i]$，最后生成$Y[i]$。在外层循环$\log n$次迭代的每次迭代中，进程P_i通过执行算法13-2第12行来更新$R[i]$的值。全部n个更新都是并行执行的。 541

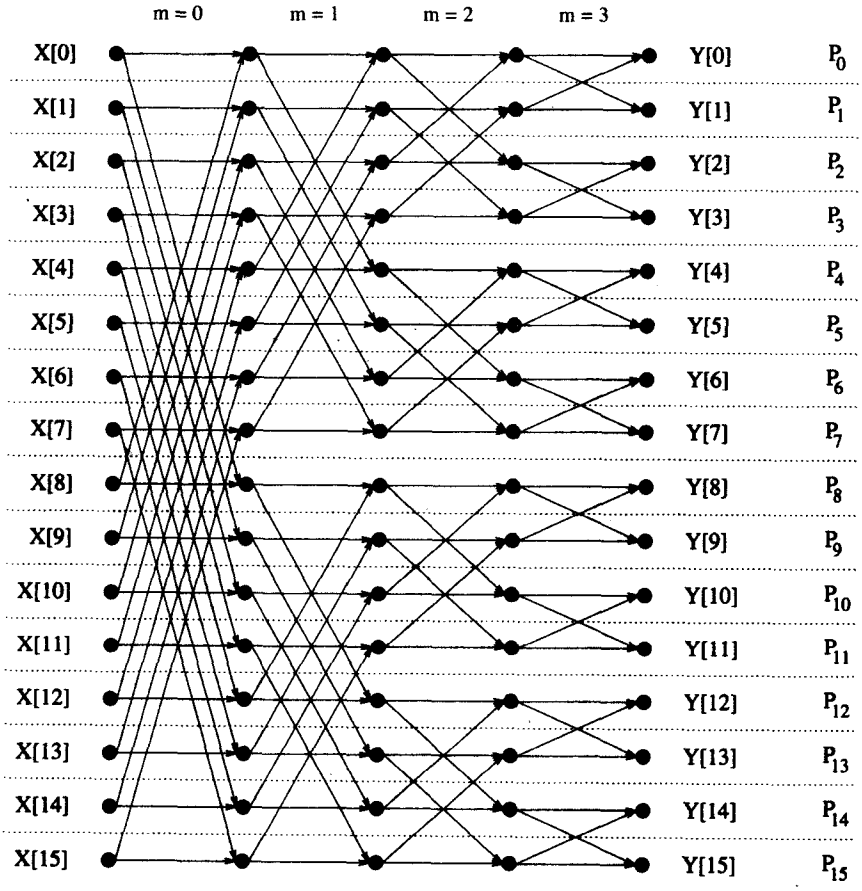


图13-3 对16个进程的16点无序快速傅里叶变换。
P_i表示标号为i的进程

为了实现更新，进程P_i需要从一个进程标号与i仅差一位的进程中，得到序列S的一个元素。回忆在超立方体中，一个节点与那些标号仅与自己差一位的节点相连。因此并行快速傅里叶变换计算自然地映射到有着一对一进程对节点映射的超立方体中。在外层循环的第一次迭代中，每对进行通信的进程的标号仅在最高有效位上不同。比如，P_0到P_7分别与P_8到P_{15}通信。

类似地，在第二次迭代中，每对进行通信的进程的标号仅在次高有效位上不同，依此类推。

在本算法的log n次迭代中，每个进程都要进行复数的乘法和加法，并且要与另一进程交
542 换复数。这样，每次迭代中都要完成固定量的工作。因此在有n节点的超立方体上并行执行算法的时间复杂度是$\Theta(\log n)$。这种超立方体上的快速傅里叶变换形式是成本最优的，因为它的进程-时间乘积为$\Theta(n \log n)$，这与串行的n点快速傅里叶变换的复杂度相同。

2. 每个进程多个任务

我们现在考虑将n点快速傅里叶变换的n个任务映射到p个进程，其中$n > p$。为简单起见，我们假设n和p都是2的幂，即$n = 2^r$，$p = 2^d$。如图13-4所示，将序列划分成n/p个邻接的元素块，并对每个进程分配一块。

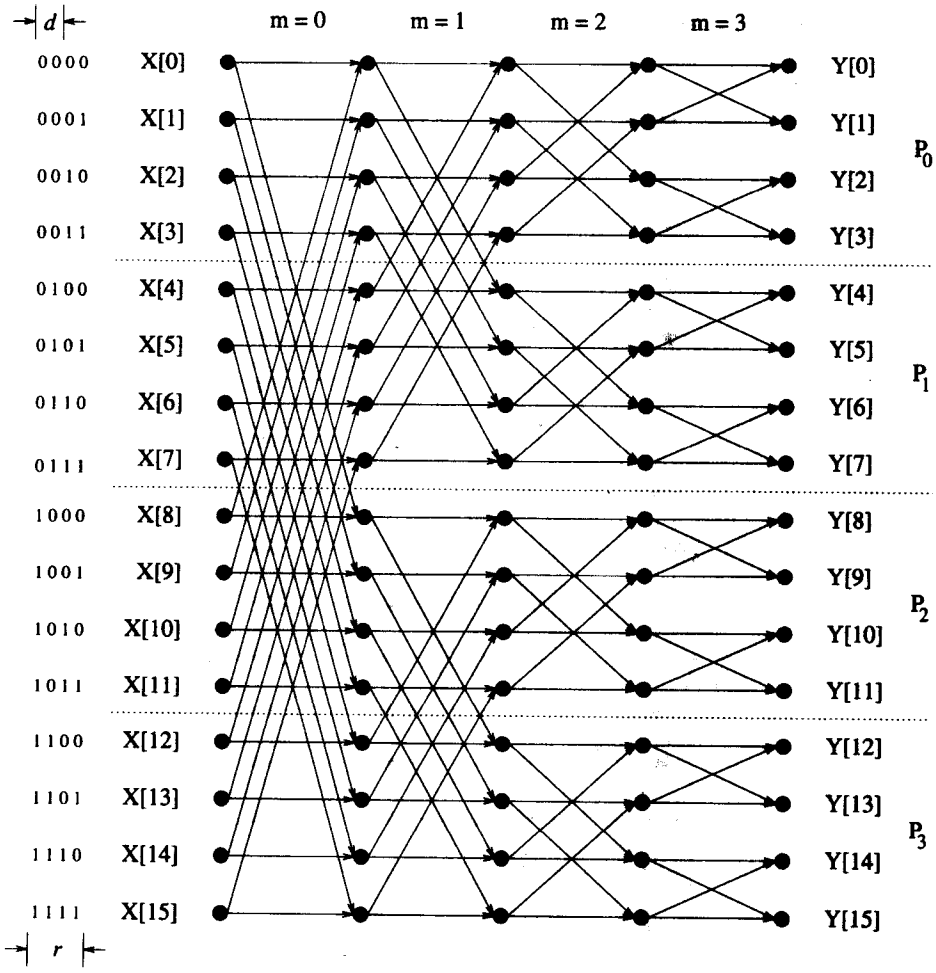


图13-4 对4个进程的16点快速傅里叶变换

注：P_i表示标号为i的进程。一般情况下，进程数$p = 2^d$，输入序列长度$n = 2^r$。

543 图13-4所示的映射有一个有趣的特性：若$(b_0 b_1 \cdots b_{r-1})$为任一i $(0 \leqslant i < n)$的二进制表示，则$R[i]$和$S[i]$映射到标号为$(b_0 \cdots b_{d-1})$的进程。也就是说，序列的任意元素下标的d个最高有效位是元素所在进程的标号的二进制表示。在决定并行计算机上执行快速傅里叶变换的通信量时，

这一映射的特性起着重要的作用。

图13-4显示，下标的d(=2)个最高有效位不同的元素映射到不同的进程。但是，所有下标具有相同d个最高有效位的元素映射到同一进程。回忆前一节，一个n点的快速傅里叶变换需要进行$r = \log n$次外层循环迭代。在循环的第m次迭代中，下标第m个最高有效位不同的元素被组合。这样，在前d次迭代中组合的元素分属不同的进程，而在后$(r-d)$次迭代中组合的元素对属于同一进程。因此，这种并行快速傅里叶变换算法仅需在$\log n$次迭代的前$d = \log p$次迭代中进行进程间交互。在后$(r-d)$次迭代中没有交互。更进一步说，在前d次迭代的第i次迭代中，一个进程所需的元素全部来自另外一个进程，这个进程标号的第i个最高有效位与其不同。

544

每个交互操作交换n/p字的数据。因此，整个算法的通信时间花费为$t_s \log p + t_w (n/p)\log p$。在$\log n$次迭代的每一次，进程都更新$R$中$n/p$个元素。若一个复数乘法和复数加法需时为$t_c$，则对$p$节点超立方体网络的$n$点快速傅里叶变换的二进制交换算法的运行时间为

$$T_P = t_c \frac{n}{p} \log n + t_s \log p + t_w \frac{n}{p} \log p \tag{13-4}$$

进程-时间乘积为$t_c n \log n + t_s p \log p + t_w n \log p$。要使该并行系统达到成本最优，这个乘积应是$O(n \log n)$，即为快速傅里叶变换算法的串行时间复杂度。这对于$p \leqslant n$是成立的。

加速比和效率的表达式由下面的式子给出:

$$\begin{aligned} S &= \frac{t_c n \log n}{T_P} \\ &= \frac{pn \log n}{n \log n + (t_s/t_c) p \log p + (t_w/t_c) n \log p} \\ E &= \frac{1}{1 + (t_s p \log p)/(t_c n \log n) + (t_w \log p)/(t_c \log n)} \end{aligned} \tag{13-5}$$

3. 可扩展性分析

由13.1节可知，一个n点快速傅里叶变换的问题规模W为

$$W = n \log n \tag{13-6}$$

因为一个n点快速傅里叶变换可以最多利用如图13-3所示映射的n个进程，只要$n \geqslant p$或$n \log n \geqslant p \log p$就可保证进程$p$繁忙。因此，由于并发性，这一并行快速傅里叶变换算法的等效率函数为$\Omega(p \log p)$。现在我们来推导基于不同通信相关项的二进制交换算法的等效率函数。将公式（13-5）重写为

$$\frac{t_s p \log p}{t_c n \log n} + \frac{t_w \log p}{t_c \log n} = \frac{1 - E}{E}$$

为了达到某个固定的效率E，表达式$(t_s p \log p) / (t_c n \log n) + (t_w \log p) / (t_c \log n)$必须等于常数$1/K$，其中$K = E/(1-E)$。这样定义常数$K$是为了与第5章在术语上保持一致。如在5.4.2节那样，我们用近似方法得到等效率函数的近似表达式。先决定关于p的问题规模的增长率，使得与t_s有关的项成为常数。为此，假设$t_w = 0$。这样保证固定效率E的条件变成下面的形式:

545

$$\begin{aligned}\frac{t_s p\log p}{t_c n\log n} &= \frac{1}{K}\\ n\log n &= K\frac{t_s}{t_c}p\log p\\ W &= K\frac{t_s}{t_c}p\log p\end{aligned}\tag{13-7}$$

公式(13-7)给出由于来自交互延迟时间或消息启动时间的开销的等效率函数。

类似地可以得到包含由于来自t_w的开销的等效率函数。假设$t_s = 0$，则固定效率E要求下式得到满足：

$$\begin{aligned}\frac{t_w\log p}{t_c\log n} &= \frac{1}{K}\\ \log n &= K\frac{t_w}{t_c}\log p\\ n &= p^{Kt_w/t_c}\\ n\log n &= K\frac{t_w}{t_c}p^{Kt_w/t_c}\log p\\ W &= K\frac{t_w}{t_c}p^{Kt_w/t_c}\log p\end{aligned}\tag{13-8}$$

若项Kt_w/t_c小于1，则公式(13-8)所需的问题规模增长率也小于$\Theta(p\log p)$。此时，公式(13-7)确定并行系统的整体等效率函数。然而，若Kt_w/t_c大于1，则公式(13-8)确定并行系统的整体等效率函数，并且大于由公式(13-7)给出的$\Theta(p\log p)$的等效率函数。

对本算法，渐进等效率函数依赖于K，t_w和t_c的相关值。其中，K是保证效率E的增函数，t_w依赖于并行计算机的通信带宽，t_c依赖于处理器的运算速度。等效率函数的量级依赖于预期的效率E和硬件相关参数，这是快速傅里叶变换算法的特别之处。事实上，$Kt_w/t_c = 1$（即$1/(1-E) = t_c/t_w$，或$E = t_c/(t_c + t_w)$）时对应的效率可看作是一种阈值。对于固定的t_c和t_w，很容易得到接近阈值的效率。对$E \leqslant t_c/(t_c + t_w)$，渐进等效率函数为$\Theta(p\log p)$。远高于阈值$t_c/(t_c + t_w)$的效率仅在问题规模极大时可能达到。这是由于在这些效率时，其渐进等效率函数是 546 $\Theta(p^{Kt_w/t_c}\log p)$。下面的例子显示$Kt_w/t_c$的值在等效率函数中的作用。

例13.1 二进制交换算法中阈值的影响

考虑一个超立方体，给定硬件参数的相关值为$t_c = 2$、$t_w = 4$、$t_s = 25$时。用这些值，可得阈值效率$t_c/(t_c + t_w)$为0.33。

现在来看对于在超立方体上保持某些低于或高于阈值的效率，二进制交换算法的等效率函数。算法由于并发的等效率函数为$p\log p$。从公式(13-7)和(13-8)知，与开销函数中的t_s和t_w相关的等效率函数分别是$K(t_s/t_c)p\log p$和$K(t_w/t_c)p^{Kt_w/t_c}\log p$。为保证某一给定的效率$E$（即给定的$K$），整体等效率函数由下式给出：

$$W = \max\{p\log p, K\frac{t_s}{t_c}p\log p, K\frac{t_w}{t_c}p^{Kt_w/t_c}\log p\}$$

图13-5显示当$E = 0.20, 0.25, 0.30, 0.35, 0.40$和$0.45$时这个函数的曲线。从图中看出，对于直到阈值的效率，各等效率曲线是等距的。为了维持阈值之上的效率，问题规模需要大大地

增加。$E = 0.20$，0.25，0.30时的渐进等效率函数为$\Theta(p \log p)$。$E = 0.40$时的渐进等效率函数为$\Theta(p^{1.33}\log p)$，而$E = 0.45$时为$\Theta(p^{1.64}\log p)$。

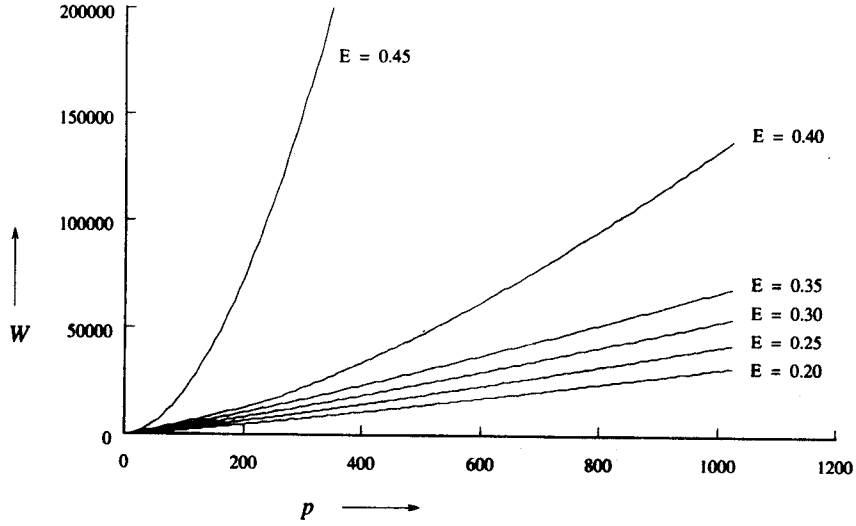


图13-5 E取不同值时超立方体上二进制交换算法的等效率函数，给定$t_c = 2$，$t_w = 4$，$t_s = 25$

图13-6显示同样硬件参数条件下，在256节点超立方体上的n点快速傅里叶变换的效率曲线。效率E对于不同的n值由公式(13-5)计算，p取256。由图可见，效率最初随问题规模增加而快速增加，但效率曲线在阈值后就变平直了。 ■ 547

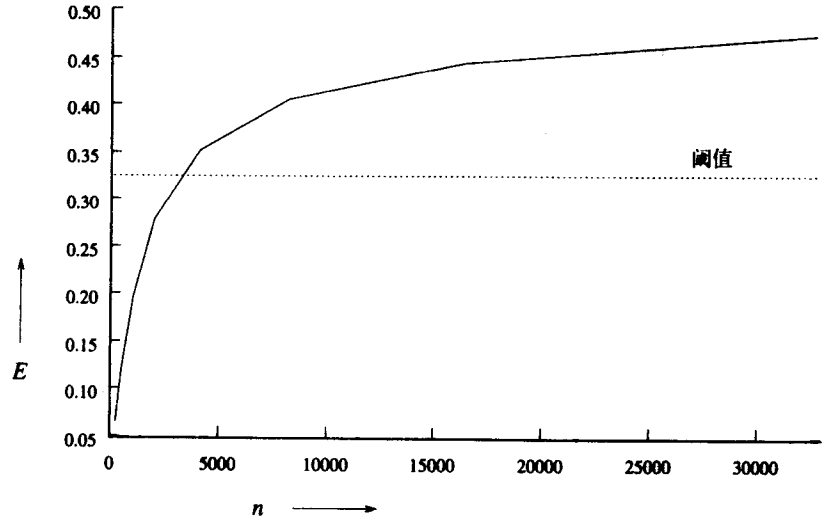


图13-6 n点快速傅里叶变换在256节点超立方体上二进制交换算法的效率曲线，给定$t_c = 2, t_w = 4, t_s = 25$

例13.1表明，对于合理的问题规模来说，效率是有限制的，这种限制取决于并行计算机的CPU速度和通信带宽的比率。可以通过增加通信通道带宽来提升效率限制。然而，仅仅提高

CPU的速度而不增加通信通道带宽只会降低效率限制。因此，在通信和计算速度不平衡的并行计算机上运行时，二进制交换算法性能不佳。若硬件在这两者上平衡的话，本算法将是可扩展的，并且当按照$\Theta(p \log p)$的速度增加问题规模时，可以保持合理的效率。

13.2.2 有限带宽网络

下面我们讨论如何在交叉段带宽小于$\Theta(p)$的并行计算机上实现二进制交换算法。我们选用格网互连网络来表示算法及其性能特征。假定将n个任务映射到运行于具有$\sqrt{p}$行和$\sqrt{p}$列的格网中的p个进程上，$\sqrt{p}$是2的整数次幂。令$n = 2^r$，$p = 2^d$。另外假设进程按行标号，而且数据的分布方式与超立方体上的一致；即下标为$(b_0 b_1 \ldots b_{r-1})$的元素映射到标号为$(b_0 \ldots b_{d-1})$的
548 进程上。

与在超立方体中一样，标号仅有一位不同的进程之间在最初的$\log p$次迭代中进行通信。与超立方体不同之处在于，通信的进程间并不直接链接在格网中。因此消息通过多级链路传递，并且在同一链路上会发生消息的重叠。图13-7显示在64节点格网中计算快速傅里叶变换时进程0和37收发的消息。如图所示，进程0与进程1、2、4、8、16、32通信。注意这些进程都与进程0处在相同的行或列。进程1、2、4与进程0处于同一行，距离分别为1、2、4。进程8、16、32与进程0处于同一列，距离也分别为1、2、4。更精确地说，在需要通信的$\log p$个步骤的$\log\sqrt{p}$步中，通信的进程处在同一行，在余下的$\log\sqrt{p}$步中，它们处在同一列。共享至少一条链路的消息数目等于每个消息经过的链路数（习题13.9），因为在一给定的快速傅里叶变换迭代中，所有进行通信的节点对都经过相同数量的链路交换消息。

在同一行或一列中的通信进程间的距离从1增加到$\sqrt{p}/2$个链路，$\log\sqrt{p}$次迭代的每一次都倍增。这对格网中的所有进程都是一样的，如图13-7中所示的进程37。这样，花费在按行通信的时间总量是$\sum_{m=0}^{d/2-1}(t_s + t_w(n/p)2^m)$。在列上花费的时间也一样。我们曾假设一次复数乘法和加法运算的耗时为t_c。由于在$\log n$次迭代的每个迭代中一个进程执行n/p次这样的计算，总体并行运行时间由下式给出：

$$\begin{aligned} T_P &= t_c\frac{n}{p}\log n + 2\sum_{m=0}^{d/2-1}\left(t_s + t_w\frac{n}{p}2^m\right) \\ &= t_c\frac{n}{p}\log n + 2\left(t_s\log\sqrt{p} + t_w\frac{n}{p}(\sqrt{p}-1)\right) \\ &\approx t_c\frac{n}{p}\log n + t_s\log p + 2t_w\frac{n}{\sqrt{p}} \end{aligned} \tag{13-9}$$

加速比和效率由下式给出：

$$\begin{aligned} S &= \frac{t_c n\log n}{T_P} \\ &= \frac{pn\log n}{n\log n + (t_s/t_c)p\log p + 2(t_w/t_c)n\sqrt{p}} \\ E &= \frac{1}{1 + (t_s p\log p)/(t_c n\log n) + 2(t_w\sqrt{p})/(t_c\log n)} \end{aligned} \tag{13-10}$$

这个并行系统的进程-时间乘积为$t_c n\log n + t_s p\log p + 2t_w n\sqrt{p}$。要达到成本最优，则该

乘积应该为$O(n \log n)$。在$\sqrt{p} = O(\log n)$或$\sqrt{p} = O(\log^2 n)$时获得这个结果。由于在公式(13-9)中与t_c相关的通信项与在超立方体中的一样，对应的等效率函数还是由公式(13-7)中给出的$\Theta(p \log p)$。通过进行与13.2.1节中一样的等效率分析，可以证明与t_s项相关的等效率函数为$2K(t_s/t_c)2^{(K(t_s/t_c)\sqrt{p})}\sqrt{p}$（习题13.4）。给定这一等效率函数，为维持某一效率值不变，问题规模必须以指数级增长。因此快速傅里叶变换的二进制交换算法在格网上的可扩展性不佳。

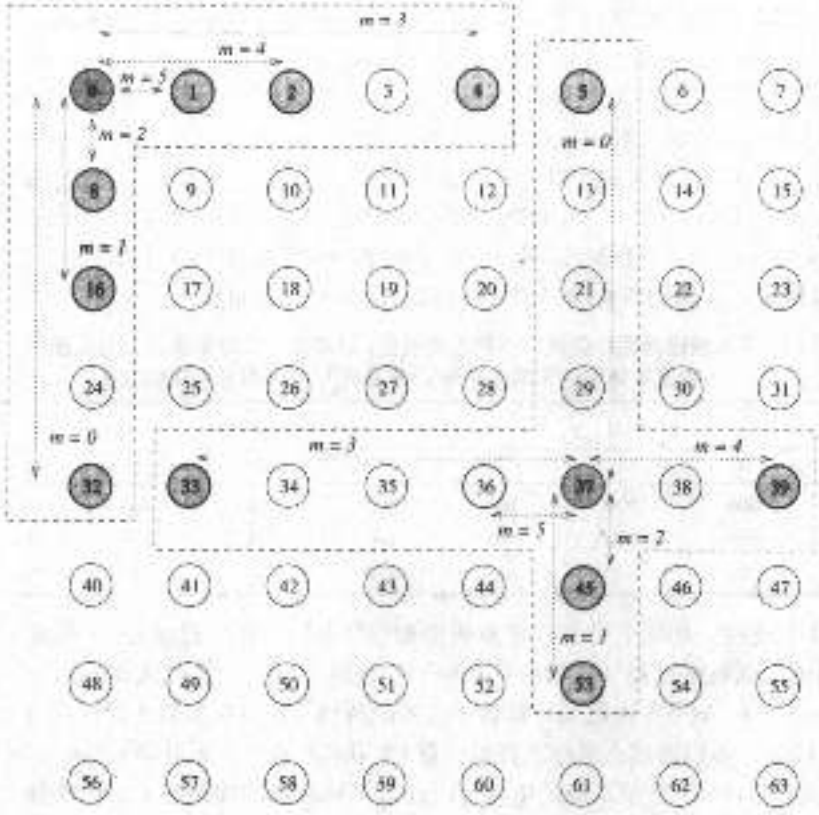


图13-7 64进程的逻辑方阵格网上进行快速傅里叶变换时的数据通信。
图中显示与标号为0和37的进程交换数据的所有进程

格网上的二进制交换算法的通信开销无法通过采用到进程的不同的映射序列来减少。在任何映射中，都至少有一次迭代，其中一对相互通信的进程相距至少$\sqrt{p}/2$条链路（习题13.2）。该算法本身要求在p节点整体的对分带宽为$\Theta(p)$，而在对分带宽为$\Theta(\sqrt{p})$的二维格网型这样的体系结构中，通信时间不可能渐近地好于上面讨论的$t_c \log p + 2(n/\sqrt{p})t_s$。

549
↓
550

13.2.3 并行快速傅里叶变换中的额外计算

到目前为止，我们讨论了超立方体和格网上的快速傅里叶变换的并行形式，也讨论了在这两种体系结构上考虑到通信开销后的性能和可扩展性。本节我们讨论可能在并行快速傅里叶变换实现中出现的另一种开销。

回忆算法13-2第12行中S的一个元素与ω（旋转因子）的一个幂相乘的计算步骤。对n点的

快速傅里叶变换来说，第12行在串行算法中要执行$n \log n$次。但是，在整个算法中用到了只有ω的n个不同的幂次（即ω^0，ω^1，…，ω^{n-1}）。所以，有些旋转因子被重复使用。在串行算法的实现中，可以在执行主算法前预先把n个旋转因子算好，并保存起来。这样计算旋转因子就只需$\Theta(n)$次复数运算，而不再是在每次迭代时都要计算所有旋转因子所需的$\Theta(n \log n)$次运算。

在并行算法实现中计算旋转因子的工作量无法缩减到$\Theta(n)$步。因为即便某旋转因子被重复使用，它可能是不同时间在不同的进程使用的。若同样规模的快速傅里叶变换是在相同数目的进程中计算，那么进程的每步计算都要相同的旋转因子集。在此情况下，旋转因子可以预先计算并储存起来，其计算开销可以在相同规模的快速傅里叶变换所有实例的执行中分摊。但是，如果只考虑快速傅里叶变换的一个实例，则旋转因子的计算增加了整个并行实现的额外开销，因为它比串行算法要进行更多的整体运算。

例如，考虑一个8点快速傅里叶变换三次迭代中所用ω的不同的幂。在算法第5行开始的循环的第m迭代中，对所有$i(0 \leq i < n)$都要计算ω^l，其中l是通过反转i的第$m-1$最高有效位的顺序并在右边填充$\log n-(m+1)$个0而得到的（关于l的推导请参见图13-1及算法13-2）。表13-1说明一个8点快速傅里叶变换的所有i和m的值所需要ω的幂的二进制表示。

表13-1 8点快速傅里叶变换中不同迭代所需ω的幂次的二进制表示（另见图13-1）。m的值是外层循环的迭代号，i是算法13-2中内层循环的索引

	i							
	0	1	2	3	4	5	6	7
$m=0$	000	000	000	000	100	100	100	100
$m=1$	000	000	100	100	010	010	110	110
$m=2$	000	100	010	110	001	101	011	111

551 若使用8个进程，则每个进程计算和使用表13-1中的一列。进程0在其所有迭代中只计算一个旋转因子，而有些进程（本例中是其他2～7进程）每三次迭代就要计算一个新的旋转因子。若$p = n/2 = 4$，则每个进程要计算表中连续的两列。这时最后的进程计算表中最后两列的旋转因子。因此，最后的那个进程总共要计算4个不同的幂：$m = 0(100)$和$m = 1(110)$各一个，$m = 2$两个(011和111)。虽然不同的进程可以计算不同数量的旋转因子，但额外工作的总开销同p与任何单个进程计算的旋转因子的最大数目的乘积成正例。令$h(n, p)$是n点快速傅里叶变换的p个进程计算的旋转因子的最大数目。表13-2列出$h(8, p)$在$p = 1$、2、4、8时的值。表中也列出任一进程在每次迭代中最多要计算的新旋转因子的数目。

表13-2 8点快速傅里叶变换计算每次迭代中任一进程所用的ω新的幂次的最大数目

	p = 1	p = 2	p = 4	p = 8
$m=0$	2	1	1	1
$m=1$	2	2	1	1
$m=2$	4	4	2	1
总计= $h(8,p)$	8	7	4	3

函数h由下面的递归关系（习题13.5）定义：

$$
\begin{aligned}
&h(n, 1) = n \\
&h(p, p) = \log p \qquad (p \neq 1) \\
&h(n, p) = h(n, 2p) + n/p - 1 \qquad (p \neq 1, n > p)
\end{aligned}
$$

对$p > 1$且$n \geqslant p$，上述递归关系的解是

$$h(n, p) = 2\left(\frac{n}{p} - 1\right) + \log p$$

这样，若计算一个旋转因子需时 t_c'，那么一个进程为了计算旋转因子至少要花费的时间是 $t_c' 2(n/p-1) + t_c' \log p$。计算旋转因子的开销是对所有进程求和，总计为$2\, t_c' (n-p) + t'_c p \log p$。因为即使是串行实现计算旋转因子的开销也达到 $t_c' n$，由于额外工作而导致的并行总开销 $T_o^{extra_work}$ 由下面的公式给出:

$$\begin{aligned} T_o^{extra_work} &= (2t_c'(n-p) + t_c' p \log p) - t_c' n \\ &= t_c'(n + p(\log p - 2)) \\ &= \Theta(n) + \Theta(p \log p) \end{aligned}$$

这一开销与用于计算快速傅里叶变换的并行计算机的体系结构无关。与 $T_o^{extra_work}$ 相关的 552
等效率函数为$\Theta(p \log p)$。由于这个项的顺序与消息启动时间和并发相关的等效率项的顺序相同，这些额外的计算不会影响并行快速傅里叶变换的整体可扩展性。

13.3 转置算法

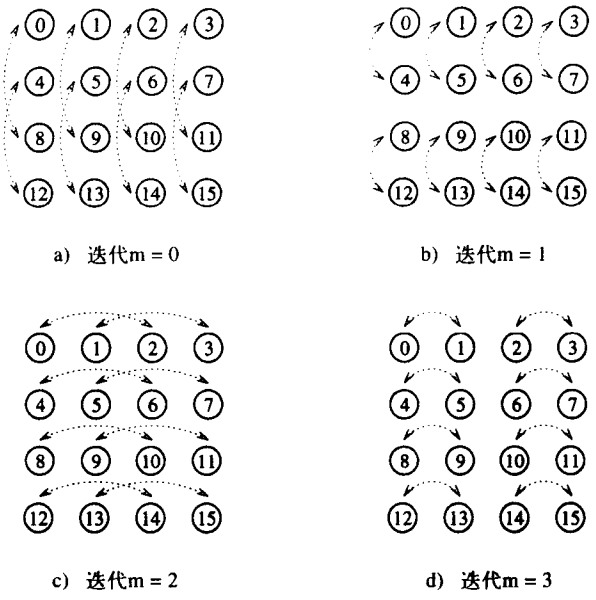
只有在通信带宽与CPU处理速度相比足够高的并行计算机上，二进制交换算法才有良好的性能。在某一阈值以下的效率可以在适度增加处理器数目的条件下通过增加问题的规模来保持。然而，若通信带宽比处理机速度低的话，这一阈值是很低的。在本节中，我们介绍一种不同的快速傅里叶变换的并行形式，它牺牲一部分效率以换取更为一致的并行性能。由于这种算法用到了矩阵的转置，因此称为*转置算法*。

转置算法的效率在低于阈值时要比二进制交换算法差。但是，在超过二进制交换算法的效率阈值后使用转置算法更容易提高效率。因此，本算法在通信带宽与CPU速度的比值较小和需要较高效率时尤其有用。在超立方体或对分带宽为$\Theta(p)$的p节点的网络上，转置算法有固定的渐进等效率函数$\Theta(p^2 \log p)$。就是说，等效率函数的顺序是与点对点通信和计算速度的比率无关的。

13.3.1 二维转置算法

最简单的转置算法只需对一个二维阵列进行转置运算，所以我们称之为*二维转置算法*(two-dimensional transpose algorithm)。

假设 $\sqrt{n}$ 是2的幂，并且在算法13-2中使用的大小为n的序列排列在一个 $\sqrt{n} \times \sqrt{n}$ 的二维方阵中，$n = 16$的情况如图13-8所示。请回忆在算法13-2中，计算一个n点序列的快速傅里叶变换要经过外层循环的$\log n$次迭代。若数据按照图13-8所示方式排列，那么进行快速傅里叶变换时每列能够独立经$\log \sqrt{n}$ 次迭代处理完成，而不需要从其他列取数据的任何列。类似地，在余下的$\log \sqrt{n}$ 次迭代中，每行独立进行计算，不需要从其他行取数据的任何行。图13-8说明16点快速傅里叶变换的元素组合模式。由图可知，如果规模为n的数据排成 $\sqrt{n} \times \sqrt{n}$ 的二维阵列，那么n点的快速傅里叶变换等价于阵列中列的独立 $\sqrt{n}$ 点快速傅里叶变换，随后接着阵列中行的独立 $\sqrt{n}$ 点快速傅里叶变换。



a) 迭代m = 0　　b) 迭代m = 1

c) 迭代m = 2　　d) 迭代m = 3

图13-8 数据排列成4 × 4二维方阵的16点快速傅里叶变换的元素组合模式

如果在计算 $\sqrt{n}$ 点列的快速傅里叶变换后将 $\sqrt{n} \times \sqrt{n}$ 数据阵列转置，则问题余下部分是计
553 算转置后矩阵的 $\sqrt{n}$ 点按列的快速傅里叶变换。转置算法利用这一特性并行计算快速傅里叶变换，计算中在进程之间按列进行带状划分分配数据的 $\sqrt{n} \times \sqrt{n}$ 阵列。例如，考虑图13-9所示的16点快速傅里叶变换的计算，4 × 4的数据阵列在4个进程间分配，每个进程存储阵列的一个列。一般地，二维转置算法分三阶段进行。第一阶段计算每列的 $\sqrt{n}$ 点快速傅里叶变换；第二阶段转置数据阵列；第三阶段也就是最后一阶段与第一阶段一样，包含转置阵列中各列的 $\sqrt{n}$ 点快速傅里叶变换。从图13-9看出，算法的第一和第三阶段无需进行进程间通信。在这两个阶段，对于每个按列的快速傅里叶变换的所有 $\sqrt{n}$ 点在同一进程是可用的。只有第二阶段需要在转置 $\sqrt{n} \times \sqrt{n}$ 矩阵时进行通信。

在图13-9所示的转置矩阵算法中，数据阵列的一列分配给一个进程。在更进一步分析转置算法前，先考虑下面更一般的情形：使用p个进程，$1 \leqslant p \leqslant \sqrt{n}$ 。$\sqrt{n} \times \sqrt{n}$ 数据阵列带状划分成块，每个进程分到 $\sqrt{n}/p$行的一块。在算法的第一和第三阶段计算大小为 $\sqrt{n}$ 的 $\sqrt{n}/p$个快速傅里叶变换。第二阶段转置 $\sqrt{n} \times \sqrt{n}$ 矩阵，这是用一维划分在p个进程间分配的矩阵。回忆4.5节，这样的转置需要多对多私自通信。

554 现在来推导二维转置算法并行运行时间的表达式。本算法唯一的进程间通信发生在将按行或按列划分并映射到p个进程的 $\sqrt{n} \times \sqrt{n}$ 矩阵进行转置运算时。在表4-1中的多对多私自通信的表达式中，用每个进程所拥有的数据量n/p代替消息的大小m，获得算法第二阶段的时间花费 $t_s(p-1) + t_w n/p$。第一和第三阶段分别耗时 $t_c \times \sqrt{n}/p \times \sqrt{n} \log \sqrt{n}$ 。因此，在超立方体或任何对分带宽为$\Theta(p)$的网络上，转置算法的并行运行时间如下：

$$\begin{aligned} T_P &= 2t_c \frac{\sqrt{n}}{p} \sqrt{n} \log \sqrt{n} + t_s(p-1) + t_w \frac{n}{p} \\ &= t_c \frac{n}{p} \log n + t_s(p-1) + t_w \frac{n}{p} \end{aligned} \qquad (13\text{-}11)$$

加速比和效率的表达式为:

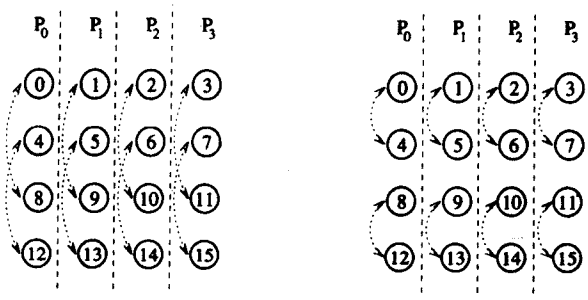
$$S \approx \frac{pn\log n}{n\log n + (t_s/t_c)p^2 + (t_w/t_c)n}$$

$$E \approx \frac{1}{1 + (t_s p^2)/(t_c n\log n) + t_w/(t_c \log n)} \qquad (13\text{-}12)$$

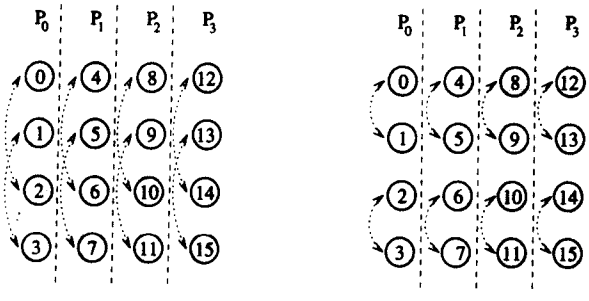
555

本并行系统的进程-时间乘积为$t_c n \log n + t_s p^2 + t_w n$。若$n \log n = \Omega(p^2 \log p)$，则它是成本最优的。

注意，在公式(13-12)的效率表达式中，与t_w相关的项同进程数无关。本算法的并发度要求$\sqrt{n} = \Omega(p)$，因为最多有$\sqrt{n}$个进程可以用于以带状方式划分$\sqrt{n} \times \sqrt{n}$数据阵列。因此，$n = \Omega(p^2)$，或$n \log n = \Omega(p^2 \log p)$。为了有效地利用全部进程，问题规模的增长必须至少与关于进程数的$\Theta(p^2 \log p)$同样快。在超立方体或任何对分带宽为$\Theta(p)$的互联网络上，二维转置算法的整体等效率函数为$\Theta(p^2 \log p)$。这一等效率函数与表示点对点间通信的t_w和t_c的比率无关。在截面带宽b小于p节点的$\Theta(p)$的网络中，为了推导出T_p、S、E及等效率函数，t_w必须乘以一个适当的$\Theta(p/b)$的表达式（习题13.6）。



a）转置算法第一阶段中的步骤（转置前）



b）转置算法第三阶段中的步骤（转置后）

图13-9 4进程的16点快速傅里叶变换的二维转置算法

与二进制交换算法的比较

比较公式(13-4)与(13-11)可知，转置算法由于消息启动时间t_s而比二进制交换算法具有更高的开销，但是由于每字传输时间t_w而比后者具有更低的开销。结果，哪个算法更快取决与t_s和t_w的相对值。若延迟t_s非常低，则转置算法更理想。另一方面，在高通信带宽和启动时间很

长的并行计算机上，二进制交换算法可能会工作得更好。

从13.2.1节可知，$\Theta(p \log p)$的整体等效率函数可用二进制交换算法实现，只要效率满足$Kt_w/t_c \leqslant 1$，其中$K = E/(1-E)$。若要求的效率满足$Kt_w/t_c = 2$，则二进制交换算法和转置算法的整体等效率函数都是$\Theta(p^2 \log p)$。当$Kt_w/t_c > 2$时，二维转置算法的可扩展性比二进制交换算法的更好；因此在$n \geqslant p^2$时，应该选择前者。但是值得注意的是，二维转置算法仅在目标机器体系结构的p节点的截面带宽为$\Theta(p)$时才比二进制交换算法有更好性能（习题13.6）。

13.3.2 转置算法的推广

在二维转置算法中，大小为n的输入排列成$\sqrt{n} \times \sqrt{n}$的二维阵列，按一维划分给p个进程。忽略并行计算机的底层体系结构，这些进程可以认为是按一个逻辑上的一维线性数组排列。
556 作为这一模式的推广，我们可以把排列在$n^{1/3} \times n^{1/3} \times n^{1/3}$三维阵列中的$n$个数据点映射到逻辑上的$\sqrt{p} \times \sqrt{p}$的二维进程格网。图13-10说明这种映射。为简化算法的描述，将三维阵列的三个轴记作x、y、z。在这个映射中，阵列的x-y平面划分成棋盘式的$\sqrt{p} \times \sqrt{p}$块。如图中所示，每个进程存储$(n^{1/3} \times \sqrt{p}) \times (n^{1/3} \times \sqrt{p})$列的数据，每列的长度（沿$z$轴）为$n^{1/3}$。所以每个进程有$(n^{1/3} \times \sqrt{p}) \times (n^{1/3} \times \sqrt{p}) \times n^{1/3} = n/p$个数据元素。

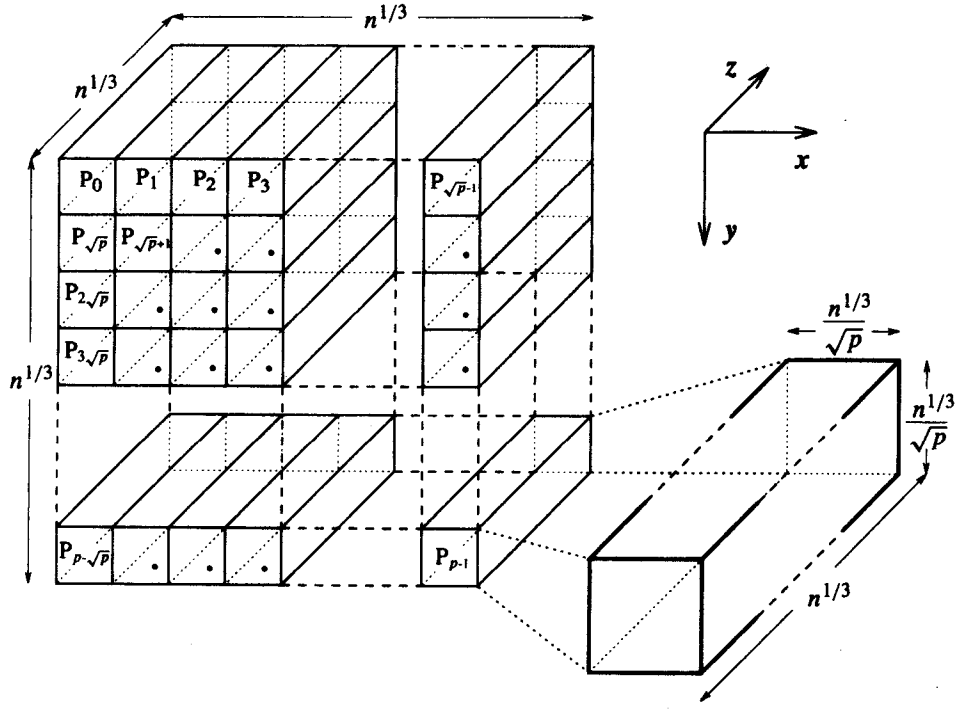


图13-10 p个进程的n点快速傅里叶变换的三维转置矩阵算法中的数据分布（$\sqrt{p} \leqslant n^{1/3}$）

在13.3.1节中已知，计算排列成$\sqrt{n} \times \sqrt{n}$输入的快速傅里叶变换，可以先计算数据所有列上的$\sqrt{n}$点一维快速傅里叶变换，再计算所有行上的$\sqrt{n}$点一维快速傅里叶变换。若数据排列成$n^{1/3} \times n^{1/3} \times n^{1/3}$三维阵列，则整个$n$点快速傅里叶变换的计算是类似的。在本例的所有三个维

中，$n^{1/3}$点快速傅里叶变换的计算是在阵列各列的元素上进行的，每次选择一维。这个算法我们称为*三维转置算法*。算法分成下面5个阶段：

1）第一阶段，沿z轴的所有行计算$n^{1/3}$点快速傅里叶变换。 557

2）第二阶段，把大小为$n^{1/3} \times n^{1/3}$的所有$n^{1/3}$个截面沿y-z平面转置。

3）第三阶段，对更改后的阵列的所有行沿z轴计算$n^{1/3}$点快速傅里叶变换。

4）第四阶段，转置每个沿x-z平面的$n^{1/3} \times n^{1/3}$个截面。

5）第五阶段，再次沿z轴的所有行计算$n^{1/3}$点快速傅里叶变换。

对如图13-10所示的数据分布，在算法的第一、三、五阶段，所有进程进行$(n^{1/3} \times \sqrt{p}) \times (n^{1/3} \times \sqrt{p})$快速傅里叶变换的计算，每个大小为$n^{1/3}$。由于所有用到的数据都在每个进程本地，无需在这三个奇数阶段中进行进程间通信。在这三个阶段的每一阶段中时间花费为 $t_c n^{1/3} \log (n^{1/3}) \times (n^{1/3}/\sqrt{p}) \times (n^{1/3}/\sqrt{p})$。因此一个进程在计算中的总时间花费为$t_c(n/p)\log n$。

图13-11说明三维矩阵算法第二、四阶段的情况。从图13-11a可看出，算法第二阶段要在 558
y-z平面上转置大小为$n^{1/3} \times n^{1/3}$的截面方阵。$\sqrt{p}$ 个进程的每列对这样的截面进行$n^{1/3}/\sqrt{p}$ 的转置。这种转置要在$\sqrt{p}$ 个进程的组之间用到多对多私自通信，其私有信息大小为$n/p^{3/2}$。若使用对分带宽为$\Theta(p)$的p节点网络，本阶段耗时$t_s(\sqrt{p}-1) + t_w n/p$。如图13-11b所示，第四阶段是类似的。此处$\sqrt{p}$ 个进程的每行沿x-z平面对$n^{1/3} \times \sqrt{p}$ 的截面进行转置。再次看到，每个截面包含$n^{1/3} \times n^{1/3}$数据单元。本阶段的通信时间与第二阶段一样。n点快速傅里叶变换的三维转置算法的总体并行运行时间为

$$T_P = t_c \frac{n}{p} \log n + 2t_s(\sqrt{p} - 1) + 2t_w \frac{n}{p} \tag{13-13}$$

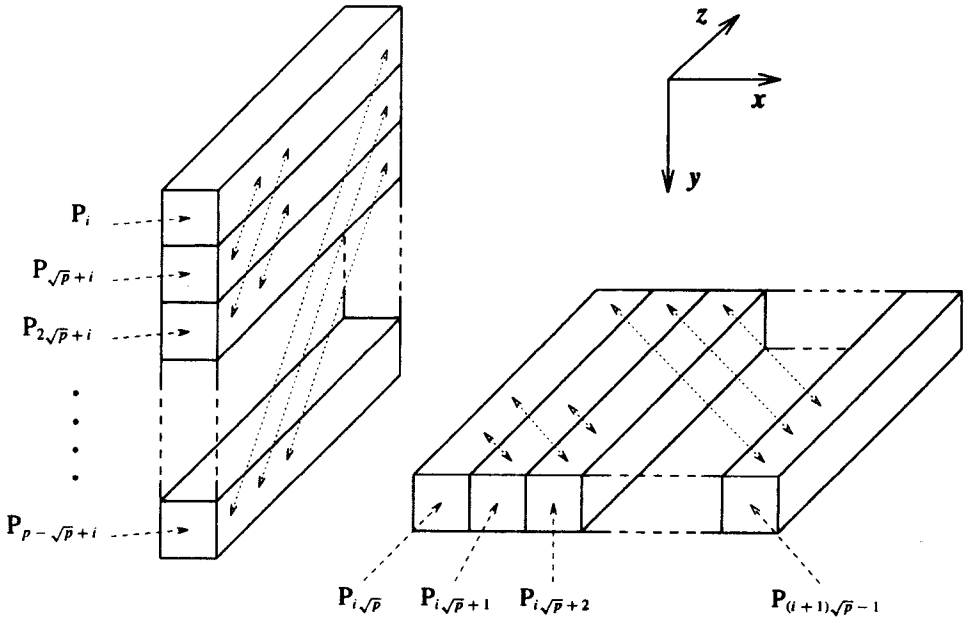
学过二维和三维转置算法后，我们可以推导更一般的类似的q维转置算法。令n点输入排列成一个逻辑的q维$n^{1/q} \times n^{1/q} \times \cdots \times n^{1/q}$（共$q$项）阵列。整个$n$点快速傅里叶变换可以视作$q$个子计算。在不同的维上的每个子计算包含$n^{1/q}$个数据点的$n^{(q-1)/q}$个快速傅里叶变换。我们将数据阵列映射到$p$个进程的逻辑上的$(q-1)$维阵列，其中$p \leqslant n^{(q-1)/q}$，$p = 2^{(q-1)s}$，$s$为某个整数。整个数据的快速傅里叶变换现在包含$2q-1$计算阶段（回忆二维转置算法有3阶段，三维转置算法有5阶段）。在q个奇数编号的阶段中，每个进程计算所需的$n^{(q-1)/q}/p$个$n^{1/q}$点的快速傅里叶变换。在所有q个阶段上每个进程的总计算时间是q（计算阶段数）、$n^{(q-1)/q}/p$（每个进程在每个阶段计算的$n^{1/q}$点快速傅里叶变换的个数）和$t_c n^{1/q} \log (n^{1/q})$（计算单个$n^{1/q}$点快速傅里叶变换的时间）的乘积。把这三项相乘就得到总计算时间$t_c(n/p)\log n$。

在$(q-1)$个偶数编号的阶段里，大小为$n^{1/q} \times n^{1/q}$ 的子阵列按进程的q维逻辑阵列的行转置。每个这样的行包含$p^{1/(q-1)}$个进程。在这$(q-1)$个通信阶段的每个阶段里，沿着$(q-1)$维进程阵列的每一维都要进行一次这样的转置。每次转置中的通信耗时为$t_s(p^{1/(q-1)}-1) + t_w n/p$。这样，在对分带宽为$\Theta(p)$的$p$节点网络上$n$点快速傅里叶变换的$q$维转置算法总的并行运行时间为

$$T_P = t_c \frac{n}{p} \log n + (q-1)t_s(p^{1/(q-1)} - 1) + (q-1)t_w \frac{n}{p} \tag{13-14}$$

公式(13-14)可以通过用2、3代替q并将结果分别与公式(13-11)和(13-13)进行比较进行验证。

比较公式(13-11)、(13-13)、(13-14)和(13-4)可以发现一个有趣的趋势：随着转置算法维数q的增加，由于t_w的通信开销也随之开销反而增加，但由于t_s的减少。二进制交换算法和二
559 维转置算法可以看作是两个极端。前者把由于t_s的通信开销最小化，但有着最大的t_w的通信开销。后者把由于t_w的通信开销最小化，但有着最大的t_s的通信开销。对于$2 < q < \log p$的各种转置算法处于这两个极端之间。对于给定的并行计算机，特定的t_c、t_s和t_w值可以决定这些算法中哪个算法具有最优的并行运行时间（习题13.8）。



a）第二阶段中处理器第i列中的转置　　b）第四阶段中处理器第i行中的转置

图13-11　p个进程n点FFT的三维转置算法中的通信（转置）阶段

从实用的观点来看，只有二进制交换算法，二维、三维转置矩阵算法是实用的，更高维的转置矩阵算法过于复杂，难以编程。此外，对n和p的限制影响了其适应性。对q维的转置矩阵算法来说，限制条件是：n必须是2的幂、q的倍数，p必须是2 的幂、$(q-1)$的倍数。换句话说，$n = 2^{qr}$，$p = 2^{(q-1)s}$，其中s、r、q都是整数。

例13.2　二进制交换算法、二维转置算法和三维转置算法的比较。

本例说明，不论是二进制交换算法还是某种转置算法，都可以作为给定的并行计算机上的算法选择，这取决于快速傅里叶变换的规模。考察例13.1中描述的64节点的超立方体，$t_c = 2$，$t_w = 4$，$t_s = 25$。图13-12给出用二进制交换算法、二维转置算法、三维转置算法在不同问题规模下得到的加速比。加速比分别来自于公式(13-4)、(13-11)以及(13-13)的并行运行时间。图中显示，不同算法在不同的n范围内对n点快速傅里叶变换提供最高的加速比。就已给定的硬件参数来说，二进制交换算法最适合低粒度情况下的快速傅里叶变换计算。二维转置算法适合极高粒度情况下的快速傅里叶变换计算。三维转置算法的加速比则在中粒度情况下是最好的。

■

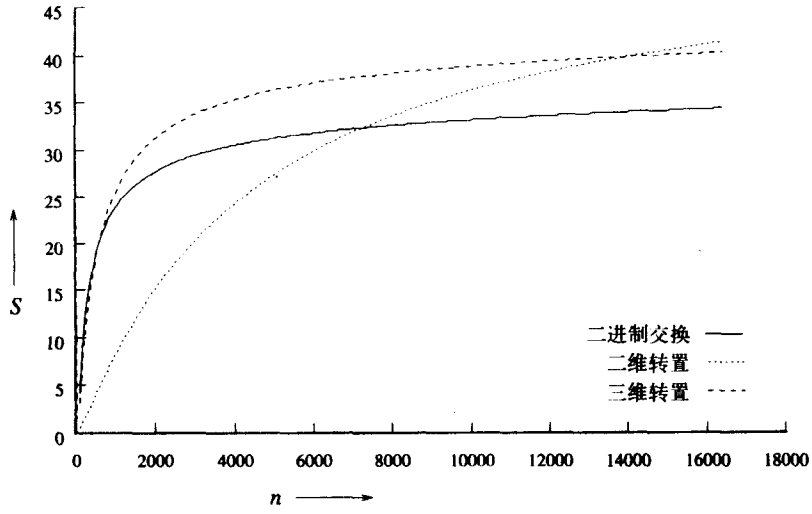


图13-12 二进制交换算法、二维转置算法和三维转置算法在64节点超立方体上（$t_c = 2$，$t_w = 4$，$t_s = 25$）加速比的比较

13.4 书目评注

由于傅里叶变换在科学与工程计算中的重要作用，在并行计算机上实现快速傅里叶变换一直倍受关注，有关的性能研究也很多。Swarztrauber[Swa87]介绍向量计算机和并行计算机上的许多快速傅里叶变换实现。Cvetanovic[Cve87]，Norton和Silberger[NS87]对基于伪共享内存体系结构的机器（如IBM RP-3）上的快速傅里叶变换给出详尽的性能分析。他们考虑了各种在内存块中数据划分的情况，对每种情况都按问题规模、进程数、内存延迟、CPU速度和通信速度得出通信开销和加速的表达式。Aggarwal，Chandra和Snir[ACS89c]分析在新的并 560
行计算模型LPRAM上快速傅里叶变换和其他算法的性能。LPRAM与标准PRAM的不同之处在于，LPRAM中的远程访问较之本地访问的成本更高。还有许多其他研究人员在从事不同体系结构上的并行快速傅里叶变换算法及其实现和实验评估方面的研究[AGGM90，Bai90，BCJ90，BKH89，DT89，GK93b，JKFM89，KA88，Loa92]。

基本的快速傅里叶变换算法（本章讨论它的并行形式）称为无序快速傅里叶变换，因为输出序列的元素是按位反转的下标顺序储存的。换句话说，输入信号的频谱要经过对输出序列Y元素的重新排列后才能得到，其中Y是由算法13-2产生的，产生的方式是对所有i，都用$Y[j]$来代替$Y[i]$，j是通过反转二进制表达的i而得到的。这是一步置换操作，称为*位反转*（bit reversal）。Norton和Silberger[NS87]证明最多只要经过$2d+1$步通信就可以得到一个有序的变换，其中$d = \log p$。由于无序快速傅里叶变换的计算只需d步通信，有序快速傅里叶变换的总通信开销差不多比无序快速傅里叶变换的两倍。显然在合适的场合都会采用无序快速傅里叶变换。当这个变换是更大计算的一部分，且对用户是不可见部分的时候，就无需对输出序列进行排序[Swa87]。在许多实际的快速傅里叶变换应用程序中，如离散泊松方程的求解和卷积，位反转是可以避免的[Loa92]。如果需要，位反转可以用由Van Loan[Loa92]给出的基于分布式内存并行计算机的算法执行。这个算法的渐进通信复杂度与超立方体上的二进制交换算法的一样。

561 文献中提出了一些简单快速傅里叶变换算法的变体。Gupta和Kumar[GK93b]证明一维和二维快速傅里叶变换在格网和超立方体上的总通信开销是一样的。使用某些计算格式可以使在串行计算机上计算离散傅里叶变换时比简单的Cooley-Tukey快速傅里叶变换算法少一些算术运算[Nus82, B76, Win77]。其中最引人注目的方法包括：以大于2的基数对一维快速傅里叶变换进行计算，以及用多项式变换法，把多维快速傅里叶变换转化为一组一维快速傅里叶变换进行计算。计算基数为q的快速傅里叶变换时，要先把大小为n的输入序列分成大小为n/q的q个子序列，再计算这q个较小的快速傅里叶变换，最后组合得到结果。例如，一个基数为4的快速傅里叶变换，每步从4个输入计算4个输出，总的迭代次数为$\log_4 n$而不是$\log_2 n$。输入序列的长度是4的幂。尽管减少了迭代次数，基数q的快速傅里叶变换的累计通信时间还是与基数为2时一样。例如，对于一个超立方体上的基数4的算法，每步通信涉及分布在二维的4个进程，而不是一维的两个进程。与此相反的是，在基数为4的快速傅里叶变换中，乘法的计算量比基数为2时少25%[Nus82]。使用更高的基数，这一数字可以稍稍改进，但通信量还是保持不变。

习题

13.1 若一n点快速傅里叶变换计算的串行运行时间为$t_c n \log n$。考虑它在一并行运行时间为$(t_c n \log n)/p + (t_w n \log p)/p$的体系结构上的实现。设$t_c = 1$，$t_w = 0.2$。

1）写出加速比和效率的表达式。

2）若期望效率为0.6，等效率函数是什么？

3）若期望效率为0.4，等效率函数如何改变（若有变化的话）？

4）若$t_w = 1$，其余不变，重做上面1)、2）小题。

13.2 [Tho83] 试证，当在由p个进程组成的方形格网中进行快速傅里叶变换时，至少有一次迭代，其中一对需要进行通信的进程相距至少为$\sqrt{p}/2$个链路。

13.3 描述在由p个进程组成的线性阵列中二进制交换算法的通信模式。线性阵列上二进制交换算法的并行运行时间、加速比、效率和等效率函数分别是什么？

13.4 试证，若$t_s = 0$，格网上二进制交换算法的等效率函数由 $W = 2K(t_w/t_c)2^{2K(t_w/t_c)\sqrt{p}}\sqrt{p}$
562 给出。

提示：利用公式(13-10)。

13.5 证明：在使用p个进程的n点快速傅里叶变换的并行实现中，任一进程计算的旋转因子的最大数量由13.2.3节中给出的递归关系给出。

13.6 对p节点二维格网和p节点线性阵列上的n点快速傅里叶变换，推导13.3.1节中描述的二维转置算法的并行运行时间、加速比和效率。

13.7 若忽略t_s，通过什么因素可增加p节点格网的通信带宽，使得它产生与p节点超立方体上的n点快速傅里叶变换的二维转置算法同样的性能？

13.8 给定超立方体网络的下列通信相关常数：i) $t_s = 250, t_w = 1$， ii) $t_s = 50, t_w = 1$， iii) $t_s = 10, t_w = 1$， iv) $t_s = 2, t_w = 1$， v) $t_s = 0, t_w = 1$。

1）$n = 2^{15}$，$p = 2^{12}$时，对于上述各组t_s，t_w的值，从二进制交换算法以及二维、三维、四维和五维转置算法中选择最合适的算法。

2）在(a) $n = 2^{12}, p = 2^6$，(b) $n = 2^{20}, p = 2^{12}$时重做1）题。

13.9 [GK93b] 考虑在$\sqrt{p}\times\sqrt{p}$的格网上计算n点快速傅里叶变换。若通道带宽随格网节点数p以$\Theta(p^x)(x>0)$的速度增长，试证由通信开销引起的等效率函数为$\Theta(p^{0.5-x}2^{2(t_w/t_c)}p^{0.5-x})$，由并发度引起的等效率函数为$\Theta(p^{1+x}\log p)$。另外，证明即使通道带宽随格网中节点数目任意 563
地增长，格网上快速傅里叶变换的最好可能的等效率函数为$\Theta(p^{1.5}\log p)$。

附录A 函数的复杂度与阶次分析

在本书中，许多地方都用阶次分析和函数的渐近复杂度来分析算法的性能。

A.1 函数的复杂度

当分析本书中的并行算法时，我们用到下面三种类型的函数：

1. **指数函数** 从实数到实数的函数f，如果可以用$f(x) = a^x$的形式表示，则称f为x的指数（exponential）函数，其中$x, a \in \Re$（实数集）且$a > 1$。例如，2^x、1.5^{x+2}和$3^{1.5x}$都是指数函数。

2. **多项式函数** 从实数到实数的函数f，如果可以用$f(x) = x^b$的形式表示，则称f为x的b次多项式函数（polynomial function），其中x，$b \in \Re$（实数集）且$b > 0$。线性函数（linear function）是1次多项式函数，二次函数（quadratic function）是2次多项式函数。例如，2，$5x$和$5.5x^{2.3}$都是多项式函数。

如果函数f是两个多项式函数g和h的和，则f仍是多项式函数，它的次数等于g和h中的最高次数。例如，$2x + x^2$是2次多项式函数。

3. **对数函数** 从实数到实数的函数f，如果可以表示以$f(x) = \log_b x$的形式表示，则称f为x的对数函数（logarithmic function），其中$x, b \in \Re$且$b > 1$。在这种表示方式中，b称为对数的底（base）。例如，$\log_{1.5}x$和$\log_2 x$都是对数函数。除非特别说明，本书中的对数都以2为底。我们用$\log x$表示$\log_2 x$，用$\log^2 x$表示$(\log_2 x)^2$。

本书中的绝大多数函数可以表示成两个或多个函数的和。如果函数$f(x)$比$g(x)$增长快，则称函数f支配（dominate）函数g。因此，函数f支配函数g，当且仅当$f(x)/g(x)$是关于x的单调递增函数。换句话说，f支配g当且仅当对任意常数$c > 0$，存在一个值x_0，使得$x > x_0$时，$f(x) > cg(x)$。指数函数支配多项式函数，多项式函数支配对数函数。关系支配是传递的，如果函数f支配函数g，函数g支配函数h，则函数f也支配函数h。因此，指数函数也支配对数函数。

A.2 函数的阶次分析

对算法进行分析时，通常很难或者不可能导出如运行时间、加速比和效率等参数的确切表达式。因此，许多情况下，只能够近似地表示精确表达式，这种近似表示可能更能说明函数的性质，因为它注重于影响参数的关键因素。

例A.1 三辆汽车行驶的距离

考察三辆汽车A，B，C。假设我们在$t = 0$时刻开始监视它们。在$t = 0$，汽车A以1000英尺每秒的速度恒定行驶；在$t = 0$，汽车B以100英尺每秒的速度行驶，加速度为20英尺/秒^2；汽车C在$t = 0$时刻处于静止状态，然后行驶的加速度为25英尺/秒^2。用$D_A(t)$，$D_B(t)$，$D_C(t)$分别表示汽车A，B，C在t秒内行驶的距离，从初等物理学可知

$$
\begin{aligned}
D_A(t) &= 1000t \\
D_B(t) &= 100t + 20t^2 \\
D_C(t) &= 25t^2
\end{aligned}
$$

现在，我们比较三辆车在指定时间内行驶的距离。当$t > 45$秒时，汽车B将超越汽车A。同样，当$t > 20$秒时，汽车C超越汽车B，当$t > 40$秒时，汽车C超越汽车A。而且，$t > 20$时，$D_C(t) < 1.25\ D_B(t)$且$D_B(t) < D_C(t)$，这表示经过一定时间后，汽车B和汽车C的性能差异由一个常数乘数因子成比例地限定。所有这些事实都可用表达式的阶次分析表示。 ■

566

Θ符号：从例A.1可知，对于$t > 20$，$D_C(t) < 1.25D_B(t)$且$D_B(t) < D_C(t)$；即在$t > 0$时，汽车B和汽车C的性能差异由一个常数乘数因子限定。在分析性能时，这样的等价关系非常重要。这两个函数间的关系可用Θ符号表示为$D_C(t) = \Theta(D_B(t))$或$D_B(t) = \Theta(D_C(t))$。而且，这两个函数都等于$\Theta(t^2)$。

Θ符号的形式定义如下：给定函数$g(x)$，$f(x) = \Theta(g(x))$当且仅当对于任意常数$c_1, c_2 > 0$，存在一个$x_0 \geqslant 0$，使得对所有的$x \geqslant x_0$，有$c_1 g(x) \leqslant f(x) \leqslant c_2 g(x)$。

O符号：通常，我们希望用一个简单的函数来限制某一特别参数的增长。从例A.1中可以看到，如果$t > 45$，则$D_B(t)$总是大于$D_A(t)$。$D_A(t)$和$D_B(t)$之间的这种关系可用O符号表示为$D_A(t) = O(D_B(t))$。

O符号的形式定义如下：给定函数$g(x)$，$f(x) = O(g(x))$当且仅当对于任意常数$c > 0$，存在一个$x_0 \geqslant 0$，使得对于所有的$x \geqslant x_0$，有$f(x) \leqslant cg(x)$。从该定义我们可得出$D_A(t) = O(t^2)$，$D_B(t) = O(t^2)$。而且，$D_A(t) = O(t)$也满足O符号的条件。

Ω符号：O符号对函数的增长率设置上界，与这正好相反，Ω符号对函数的增长率设置下界。从例A.1中可知，对于$t > 40$，$D_A(t) < D_C(t)$，这个关系可用Ω符号表示为$D_C(t) = \Omega(D_A(t))$。

Ω符号的形式定义为，给定函数$g(x)$，$f(x) = \Omega(g(x))$当且仅当对于任意常数$c > 0$，存在一个$x_0 \geqslant 0$，使得对于所有的$x \geqslant x_0$有$f(x) \geqslant cg(x)$。

以阶次符号表示的函数性质

表达式中的阶次符号有许多性质，对于分析算法的性能很有用。其中一些重要性质如下：

1）$x^a = O(x^b)$，当且仅当$a \leqslant b$。

2）对于所有的a和b，$\log_a(x) = \Theta(\log_b(x))$。

3）$a^x = O(b^x)$，当且仅当$a \leqslant b$。

4）对于任意常数c，$c = O(1)$。

567

5）如果$f = O(g)$，则$f + g = O(g)$。

6）如果$f = \Theta(g)$，则$f + g = \Theta(g) = \Theta(f)$。

568

7）$f = O(g)$，当且仅当$g = \Omega(f)$。

8）$f = \Theta(g)$，当且仅当$f = \Omega(g)$且$f = O(g)$。

索　引

索引中页码是英文版原书页码，与正文中边栏页码相对应。

α-splitting（α划分），486
k-ary d-cube network，（k元d立方体网络，参看interconnection networks）
0/1 integer-linear-programming（0/1整数线性规划），469，471
　fixed variable（固定变量），472
　free variable（自由变量），472
0/1 knapsack（0/1背包问题），520
1-D mapping （1维映射）
　block（块）
　　for connected components（连通分量），449
　　for Dijkstra' s algorithm（Dijkstra算法），437
　　for Johnson' s algorithm（Johnson算法），461
　　for Prim' s algorithm（Prim算法），435
2-D mapping（2维映射）
　block（块）
　　for Floyd' s algorithm（Floyd算法），441
　　for Johnson' s algorithm（Johnson算法），459
　cyclic（循环）
　　for Johnson' s algorithm（Johnson算法），460
8-Puzzle（九宫重排），470，474
　admissible heuristic for（允许启发式），473
　best-first search（最佳优先搜索），478
　depth-first search（深度优先搜索），476

A* algorithm（A*算法），478，507
acceleration anomalies（加速异常），502
acyclic graph（无圈图），430
adaptive routing（自适应路由选择，参看 routing mechanism）
adjacency list（邻接表），431，451，455
adjacency matrix（邻接矩阵），430，433，446，447
adjacent vertex（邻接顶点），429
admissible heuristic（允许启发式），473
all-pairs shortest paths（全部顶点对间的最短路径），437，446
　Dijkstra's algorithm（Dijkstra 算法），438
　　source-parallel formulation（源并行形式），439
　　source-partitioned formulation（源划分形式），438
　Floyd' s algorithm（Floyd算法），165，440，446，451，463，526
　　1-D block mapping（1维块映射），463
　　2-D block mapping（2维块映射），441，463
　　for cellular arrays（细胞阵列），464
　　pipelined 2-D block mapping（流水线2维块映射），443，463
　　performance comparisons（性能比较），445
all-port communication（全端口通信），186，189
all-reduce（全归约，参看reduction）
all-to-all broadcast（多对多广播），157，187，338-340，346
　dual of（…的对偶，参看dual）
　on hypercube（在超立方体上），161
　pseudocode（伪码），162，163
　with all-port communication（全端口通信），189
　on linear array（在线性阵列上），158
　on mesh（在格网上），160
　　pseudocode（伪码），162
　on ring（在环上）
　　pseudocode（伪码），160
　on tree（在树上），190
all-to-all personalized communication（多对多私自通信），170，187，189，554，559
　on hypercube（在超立方体上），175，177
　　pseudocode（伪码），179
　　with all-port communication（全端口通信），189
　on mesh（在格网上），174
　on ring（在环上），173
all-to-all reduction（多对多归约，参看reduction）
　dual of（…的对偶，参看dual）
all-to-one reduction（多对一归约，参看reduction），342，343，351，352，435
alpha-beta search（alpha-beta搜索），507
Amdahl' s law（Amdahl定律），210，228
applications（应用）
　bioinformatics（生物信息学），5
　commerce and business（商业和商务），5
　computational physics（计算物理），5

computer systems（计算机系统），6
computer vision（计算机视觉），9
embedded systems（嵌入式系统），6
engineering and design（工程与设计），4
scientific computing（科学计算），5
arc connectivity（弧连通性），43
arithmetic mean（算术平均），504
array partitioning（阵列划分，参看partitioning）
artificial intelligence（人工智能），505
atomic directive in OpenMP（OpenMP中的原子命令），325
atomic operations（原子操作），288
attribute objects（属性对象）
changing properties（改变属性），299
initialization（初始化），299
mutex（互斥），301
properties（属性）
mutex（互斥），301
synchronization variables（同步变量），300
threads（线程），299
automatic test pattern generation（自动测试模式发生器），506

back-substitution（回代），165，353，369
backtracking（回溯）
ordered（有序的），475
simple（简单的），475
barrier（障碍），307
logarithmic（对数的），309
barrier directive in OpenMP（OpenMP中的障碍命令），322
barrier synchronization（障碍同步）
using all-reduce（使用全归约），166
BDN（参看bounded-degree network）
Bernoulli distribution（贝努利分布），504
best-first branch-and-bound（最佳优先分支定界）
parallel（并行），508
best-first search（最佳优先搜索），478
A* algorithm（A*算法），478
closed list（封闭表），478
duplication in（重复），478
open list（开放表），478
parallel（并行），500，507
blackboard strategy（黑板策略），499，507
centralized strategy（集中式策略），497
graph-search strategies（图搜索策略），500
hash function（散列函数），500，507
random strategy（随机策略），499，507
termination（终止），498
storage requirement（存储需求），478
binary-exchange algorithm（二进制交换算法，参看fast Fourier transform）
binomial tree（二元树），189
bisection bandwidth（对分带宽），44
bisection width，73，79
static networks（静态网络），43
bit reversal（位反转），537，561
bitonic merge（双调合并），384
bitonic merging network（双调合并网络），384
bitonic sequence（双调序列），384
bitonic sort（双调排序），384，386
hypercube（超立方体），387，388
block of elements per process（每进程的元素块），392，417
one element per process（每进程一个元素），388
scalability（可扩展性），393
mesh（格网），387，390
block of elements per process（每进程的元素块），393
one element per process（每进程一个元素），390
row-major（以行为主），391，417
row-major shuffled（以行为主混洗），390，391，417
row-major snakelike（以行为主蛇形），391，417
scalability（可扩展性），394
perfect shuffle（完全混洗），417
performance summary（性能概括），394
bitonic sorting network（双调排序网络），386，416
bitonic split（双调分裂），384
block mapping（块映射，参看mapping）
block matrix operations（块矩阵运算），346
block-cyclic mapping（块循环映射，参看mapping）
bounded-degree network（限定度的网络），76
BPC permutations（BPC置换），189
BPRAM，76
branch prediction（分支预测），12，16
branch-and-bound search（分支定界搜索），473，505，507
broadcast（广播，参看one-to-all broadcast，all-to-all broadcast）
bubble sort（冒泡排序），394
odd-even transposition（奇偶变换），395
MPI implementation（MPI实现），249
bucket sort（桶排序）
EREW PRAM，418
bus-based network（基于总线的网络），33，39
butterfly network（蝶形网络，参看multistage network）

cache（高速缓存）
hit ratio（命中率），17
impact on memory system performance（对内存系统性能的影响），18
temporal locality（时间本地性），18
cache coherence（高速缓存一致性），29，45
directory-based systems（基于目录的系统），51
cost of（成本），52
performance of（性能），51
dirty（脏），48
distributed directory systems（分布式目录系统），53
performance of（性能），53
example protocol（例子协议），48
false sharing（假共享），47
invalidate protocol（无效协议），47，48
shared（共享的），48
snoopy systems（侦听系统），49
performance of（性能），49
update protocol（更新协议），47
channel bandwidth（通道带宽），43
channel rate（通道速度），43
channel width（通道宽度，参看channel bandwidth）
Cholesky factorization（Cholesky因式分解），369，372，374，375
circular shift（循环移位），179，187，193，349
on hypercube（在超立方体上），181，184
on linear array（在线性阵列中），179
on mesh（在格网上），181
closed list（封闭表），478
columnsort（列排序），418
combinatorial problem（组合问题），469
combining（结合），75
communication costs（通信成本），53
message passing（消息传递），53
simplified model（简化模型），58
shared-address-space（共享地址空间），61
communication latency（通信延迟）
for cut-through routing（直通路由选择），57
for packet routing（报文路由选择），56
for store-and-forward routing（存储转发路由选择），54
communication overhead（通信开销），215
comparator（比较器），382，384
decreasing（减少），382，385
increasing（增加），382，385
compare-exchange operation（比较交换操作），379，381，383，387，394
compare-split operation（比较分裂操作），381，392，417
comparison-based sorting（基于比较的排序），379
complete graph（完全图），430
completely-connected network（全连接网络），39，44
composite synchronization constructs（复合同步构造），302
composition function（组合函数），516
concatenation（连接，参看gather）
concurrency（并发，参看degree of concurrency）
condition（条件）
wait（等待），295
condition signal（条件信号），295
condition variables（条件变量），294
broadcast（广播），298
destroy（清除），296
initialization（初始化），296
timed wait（定时等待），298
congestion（拥塞，参看embedding），154，165，177
connected components（连通分量），446，464
1-D block mapping（1维块映射），449
parallel formulation（并行公式），447
Sollin's algorithm（Sollin算法），464
using depth first search（使用深度优先搜索），446
connected graph（连通图），432
connectivity matrix（连通性矩阵），445
connectivity，interconnection networks（连通性，互连网络），43
control structure（控制结构），25
copyin clause in OpenMP（OpenMP中的复制子句），328
cost（成本），203
cost of interconnection networks（互连网络的成本），44，73
cost optimality（成本最优性），204，205，208，212，219，220
relationship with isoefficiency function（与等效率函数的关系），217
critical directive in OpenMP（OpenMP中的关键命令），323
critical path（关键路径），90
length of（长度），91
critical section（临界段），288
cross-section bandwidth（截面带宽），44
crossbar network（交叉开关网络），35，39
cut-through routing（直通路由选择），56-57，188
deadlocks（死锁），58
message transfer time（消息传递时间），57
survey of techniques（技术调查），75
virtual cut-through（虚拟直通），75
cutoff depth（截止深度），483

cycle（循环），430
simple（简单的），430
with negative weight（含负权），438，463
without negative weight（不含负权），438
cyclic mapping（循环映射，参看mapping）

DAG（参看directed acyclic graph）
data parallelism（数据并行），139
data partitioning（数据划分，参看partitioning）
data-decomposition（数据分解，参看decomposition）
data-flow computing（数据流计算），9
data-parallel programming（数据并行编程），9，27，189
Davis-Putnam algorithm（Davis-Putnam算法），492
deceleration anomalies（减速异常），502
decomposition（分解），86
coarse-grained（粗颗粒的），89
data（数据），97-105
exploratory（探索），105-107
fine-grained（细颗粒的），89
hybrid（混合），109-110
recursive（递归），95-97
speculative（推测性的），107-109
default clause in OpenMP（OpenMP中的默认子句），314
degree of concurrency（并发度），89，218，221
average（平均值），90
maximum（最大值），90
dense graphs（稠密图），431，438
dependency（相关性）
branch（分支），15
data（数据），14
procedural（过程的），15
resource（资源），14
depth-first branch-and-bound（深度优先分支定界），475
depth-first search（深度优先搜索），228，474
load balancing（负载平衡），484
asynchronous round robin（异步循环法），484
global round robin（全局循环法），485
random polling（随机轮询），485
receiver-initiated（接收启动的），505
sender-initiated（发送方启动的），505
parallel（并行），480-496
acceleration anomalies（加速异常），502
deceleration anomalies（减速异常），502
storage requirement（存储需求），477
deterministic routing（确定性路由选择，参看routing mechanism）
DFT（参看discrete Fourier transform）
diameter of interconnection networks（互连网络的直径），43
dilation（膨胀度，参看embedding）
dimension-ordered routing（维序路由选择，参看routing mechanism）
direct interconnection networks（直接互连网络，参看interconnection networks）
directed acyclic graph（有向无圈图），229
model of computation（计算模型），229
directed graph（有向图），429
directory-based systems（基于目录的系统），51
cost of（成本），52
performance of（性能），51
presence bits（存在位），51
discrete Fourier transform（离散傅立叶变换），538
discrete optimization problem（DOP，离散优化问题），469
multiknapsack（多背包），508
quadratic-assignment（二次分配），508
robot motion planning（机器人动作规划），473
speech understanding（语音理解），473
VLSI floor-plan optimization（VLSI芯片布置图优化），473
disjoint set（不相交集），447
path compression（路径压缩），449
ranking（等级），449
distributed directory systems（分布式目录系统），53
performance of（性能），53
distributed memory computer（分布式内存计算机），29
distributed platforms（分布式平台），4
distributed tree search（分布式树搜索），513
divide and conquer（分治），96
donor processor（施主处理器），482
dual（对偶），149
of all-to-all broadcast（多对多广播），157
of all-to-all reduction（多对多归约），157
of all-to-one reduction（多对一归约），149
of gather（收集），169
of one-to-all broadcast（一对多广播），149，156
of scatter（散发），169
dynamic instruction issue（动态指令发送），15
dynamic interconnection networks（动态互连网络，参看interconnection networks）
dynamic load balancing（动态负载平衡），481
donor processor（施主处理器），482
recipient processor（接受者处理器），482
dynamic programming（动态规划），505，515
algorithms（算法）

0/1 knapsack（0/1背包问题），520
Floyd's all-pair shortest-paths（Floyd全部顶点对间的最短路径），526
longest-common-subsequence（最长公共子序列），523
optimal-matrix-parenthesization（最优矩阵带括号方法），527
shortest-path（最短路径），515
classification（分类）
monadic（一元的），517
nonserial（非串行的），517
nonserial-monadic（非串行一元的），517，523-526
nonserial-polyadic（非串行多元的），517，527-530
polyadic（多元的），517
serial（串行），517
serial-monadic（串行一元），517，518-523
serial-polyadic（串行多元），517，526-527
composition function（组合函数），516
functional equation（泛函方程），516
optimization equation（优化方程），516
recursive equation（递归方程），516
dynamic scheduling in OpenMP（OpenMP中的动态调度），317

E-cube routing（E立方体路由选择，参看routing mechanism），177，189
edge（边），429
incident from（关联来自），429
incident on（关联），429
incident to（关联到），429
efficiency（效率），202，210，211
embedding（嵌入），66
congestion of（拥塞），66，75，82
dilation of（膨胀），67，75
expansion of（扩充），67，75
hypercube into mesh（超立方体到格网），71
interconnection network design（互连网络设计），72
linear array into hypercube（线性阵列到超立方体），67，68
mesh into hypercube（格网到超立方体），67，69
mesh into linear array（格网到线性阵列），69
enumeration sort（枚举排序），414
equivalence class（等价类），446
execution time（执行时间，参看parallel runtime），197
expansion（扩充，参看embedding），67
exploratory decomposition（探测性分解，参看decomposition），105
exponential function（指数函数），565

false sharing（假共享），47，285
fast Fourier transform（快速傅立叶变换），537
binary-exchange algorithm（二元交换算法），541，556，560
extra computations（额外计算），551
higher radix（更高的基数），562
on linear array, ring（在线性阵列中），562，563
on mesh（在格网中），562
ordered（有序的），561
serial algorithm（串行算法）
iterative（迭代的），540，542
recursive（递归的），538，539
transpose algorithm（转置算法），553
generalized（广义的），559
three-dimensional（3维的），557，559，560
two-dimensional（2维的），553，554，556，560
fat tree（胖树），42，75
feasible solutions（可行解集合），469
fetch-and-add operation（取数和相加操作），510
fetch-and-op（取数并操作），189
FFT（参看fast Fourier transform）
firstprivate clause in OpenMP（OpenMP中的firstprivate子句），315
flit（数据片），56
flow control digit（流控制数字），56
flush directive in OpenMP（OpenMP中的flush命令），326
for directive（for命令），315
forest（树林），430，446
Fujitsu VPP 500（参看VPP500）
function（函数）
exponential（指数的，参看exponential function）
logarithmic（对数的，参看logarithmic function）
polynomial（多项式的，参看polynomial function）
functional equation（泛函方程），516

gather（收集），167，187，192
dual of（…的对偶），（参看dual）
Gaussian elimination（高斯消元法），353-369
column-oriented（面向列的），371
dense（稠密的）
with pipelining（用流水线），165
numerical considerations（数值考虑），370
row-oriented（面向行的），370
with block 1-D mapping（用块1维映射），355，367
with block 2-D mapping（用块2维映射），361，368
with cyclic 1-D mapping（用循环1维映射），361
with cyclic 2-D mapping（用循环2维映射），365
with partial pivoting（用部分选主元法），366

with pipelining（用流水线），357，362，367，368
global minimum（全局最小值），435
granularity（粒度），89
of computation（计算），205
effect on performance（对性能的影响），205
of processors（处理器），205
graph partitioning（图划分），124
graphs（图），429，462
acyclic（无圈的），430
adjacent vertex（相邻顶点），429
algorithms（算法），462
articulation points（关节点），465
bipartite（双向的），465
bridges（桥），465
complete（完全的），430
cycle（圈），430
cyclic index（循环索引），465
definition（定义），429
dense（稠密的），431，438
directed（有向的），429
forest（树林），430
independent set（独立集），451
maximal independent set（最大独立集），451
path（路径），429
representation（表示），430，431，451
adjacency list（邻接表），431
adjacency matrix（邻接矩阵），430
relative merits（优缺点），432
weighted adjacency list（加权邻接表），431
weighted adjacency matrix（加权邻接矩阵），431
sparse（稀疏的），431
state-space（状态空间），471
subgraph（子图），430
transitive closure（传递闭包），445
tree（树），430
undirected（无向的），429
connected（连通的），430
weighted（加权的），430
Gray code（葛莱码），67，68，69，75，181
greedy algorithm（贪婪算法），432，436
grid graphs（网格图），451，451，458-462
guided scheduling in OpenMP（OpenMP中的导向调度），318

Hamming distance（Hamming 距离），177
harmonic mean（调和平均值），504
hash function（散列函数），500
heuristic estimate（启发式估计），473
heuristic search（启发式搜索），473，505
admissible heuristic（允许启发式），473
hit ratio（命中率，参看cache）
hypercube network（超立方体网络），39，44，82
properties（属性），40，79
hyperquicksort（超快速排序），418

IBM SP，30
if clause in OpenMP（OpenMP中的if子句），312
in-order execution（依序执行），15
incident edge（关联边）
directed graph（有向图），429
undirected graph（无向图），429
indirect interconnection networks（无向互连网络，参看interconnection networks）
induced subgraph（导出子图），430
interconnection networks（互连网络），32
k-ary *d*-cube（*k*元 *d*立方体），39
blocking（阻塞），35，38
buses（总线），33
completely connected（全连接的），39
cost-performance tradeoffs（成本性能权衡），73
crossbar（交叉开关），35，36
direct（直接），32
dynamic（动态），32
properties of（属性），44-45
hypercube（超立方体），39
indirect（间接），32
linear array（线性阵列），39
mesh（格网），39
multistage（多级），36
network interface（网络接口），33
static（静态的），32
properties of（属性），43-44
switch（开关），32
degree（度），32
topologies（拓扑结构），33
trees（树），42
invalidate protocol（无效协议，参看cache coherence）
isoefficiency（等效率），212
analysis（分析），212，216
function（函数），212，215，220，227
lower bound（下界），217
relationship with concurrency（与并发的关系），218
relationship with cost-optimality（与最优成本的关系），217
Itanium，12
iterative deepening A*（迭代加深A*），475，506

k-ary *d*-cube network（*k*元 *d*立方体网络），44
k-to-all broadcast（*k*对多广播），192
k-to-all personalized communication（*k*对多私自通信），193
keyed-scan（键控扫描），189

lastprivate clause in OpenMP（OpenMP中的lastprivate子句），315
linear array（线性阵列），44
linear function（线性函数），565
link bandwidth（链路带宽）
  effective bandwidth（有效带宽），60
load balancing（负载平衡）
  asynchronous round robin（异步循环），484
  global round robin（全局循环），485
  global round robin with message combining（带消息结合的全局循环），510
  random polling（随机轮询），485
  receiver initiated（接收者启动的），505
  sender-initiated, 505
    distributed tree search（分布式树搜索），505
    multi-level（多层），505, 513
    single-level（单层），505, 513
load imbalance（负载不平衡），196
load-balancing（负载平衡），115-132
logarithmic function（对数函数），566
  base of（底），566
loop interchanging（循环交换），20
loop unrolling（循环跳出），16
LPRAM，76
LU factorization（LU因子分解）（参看Gaussian elimination），120，353, 354, 366, 370, 372
  column-oriented（面向列的），120
Luby algorithm（Luby算法），453
  Parallel formulation（并行形式），453

Manhattan distance（曼哈坦距离），473
mapping（映射），93
  block（块），117
  block-cyclic（块循环），122
  dynamic（动态的），117, 130-132
  for load-balancing（负载平衡），115
  one-dimensional（1维的），118
  randomized block（随机块），124
  static（静态的），116-130
  two-dimensional（2维的），118
master directive in OpenMP（OpenMP中的master命令），323
matrix multiplication（矩阵相乘），98, 102, 345-352, 371
  Cannon's algorithm（Cannon算法），347-349
    MPI implementation（MPI实现），253, 259
  DNS algorithm（DNS算法），349-352, 373
  simple algorithm（简单算法），346-347
  Strassen's algorithm（Strassen算法），345, 373
matrix transposition（矩阵转置），171, 371
  in FFT（在FFT中），553, 559
matrix-vector multiplication（矩阵-向量相乘），151, 519
  dense（稠密的），86, 93, 337-345
    isoefficiency analysis（等效率分析），340, 343
    MPI implementation（MPI实现），266, 274
    with 1-D mapping（用1维映射），338
    with 2-D mapping（用2维映射），341
  sparse（稀疏的），92
maximal independent set（最大独立集），453
  Luby algorithm（Luby算法），453
    parallel formulation（并行形式），453
memory bandwidth（内存带宽），18
example（例子），18
memory latency（内存延迟）
  role of caches（高速缓存的作用），17
memory system（内存系统）
  bandwidth（带宽），16
  impact of bandwidth（带宽的影响），18
  impact of strided access（跨距访问的影响），20
  latency（延迟），16
  limitations of（局限），16
  spatial locality（空间本地性），19
memory-constrained scaling（内存受限的扩展，参看scaling）
merge sort（合并排序）
  EREW PRAM，418
mesh network（格网网络），39, 44
  of trees（树的），80, 81
  pyramidal（金字塔的），81
  reconfigurable（可重构的），74, 80
message passing（消息传递），30
message-passing programming（消息传递编程），233
  asynchronous（异步），234
  deadlocks（死锁），237, 239
  loosely synchronous（松散同步），234
  MPI，240
  operations（操作）
    blocking receive（阻塞式接收），236

blocking send（阻塞式发送），236
buffered receive（有缓冲接收），237
buffered send（有缓冲发送），237
non-blocking receive（无阻塞式接收），239
non-blocking send（无阻塞式发送），239
non-buffered receive（无缓冲接收），236
non-buffered send（无缓冲发送），236
receive（接收），235
send（发送），235
principles（原理），233
SPMD model（SPMD模型），234
microprocessor architectures（微处理器体系结构），12
MIMD（参看multiple instruction stream multiple data stream），26，74，506
minimal routing（最小路由选择，参看routing mechanism）
minimum cost-optimal execution time（最小成本最优执行时间），218
minimum execution time（最小执行时间），218
minimum spanning tree（最小生成树），432
Kruskal's algorithm（Kruskal算法），463
Prim's algorithm（Prim算法），432，463
for sparse graphs（用于稀疏图），451
shared-address-space（共享地址空间），463
Sollin's algorithm（Sollin算法），463，467
CRCW PRAM，463
CREW PRAM，463
mesh of trees（树格网），463
ring（环），463
minimum-congestion mapping（最小拥塞映射），82
MIPS，12
MISD(参看multiple instruction stream, single data stream），74
module parallel computer（模块并行计算机），75
Moore's Law（摩尔定律），2
motion planning（动作规划），473
MPC（参看module parallel computer），75
MPI，240
concepts（概念）
avoiding deadlocks（避免死锁），246，257
Cartesian topologies（笛卡儿拓扑结构），252
collective communication operations（聚合通信操作），260
communicators（通信器），243，272
embeddings（嵌入），251
groups（组），272
overlapping communication with computation（计算与通信重叠），255
topologies（拓扑结构），251
examples（例子）
Cannon's algorithm（Cannon算法），254
Dijkstra's single-source shortest-path（Dijkstra单源最短路径算法），268
matrix-vector multiplication（矩阵-向量相乘），266，274
odd-even sort（奇偶排序），249
sample sort（样本排序），270
operations（操作）
all-to-all（多对多），265
barrier（障碍），260
broadcast（广播），261
collective（聚合），260
create Cartesian topologies（创建笛卡儿拓扑结构），252
gather（收集），263
non-blocking send and receive（无阻塞发送与接收），256
prefix sum（前缀和），263
querying Cartesian topologies（在笛卡儿拓扑结构中查询），252
querying non-blocking send and receive（无阻塞发送和接收查询），256
querying the communicator（查询通信器），243
reduction（归约），261
scatter（散发），264
send and receive（发送与接收），244，248
splitting a communicator（分裂通信器），272
sub-dividing Cartesian topologies（细分笛卡儿拓扑结构），273
MPI_Allgather，264
MPI_Allgatherv，264
MPI_Allreduce，263
MPI_Alltoall，265
MPI_Alltoallv，266
MPI_Barrier，260
MPI_Bcast，261
MPI_Cart_coord，253
MPI_Cart_create，252
MPI_Cart_rank，253
MPI_Cart_shift，253
MPI_Cart_sub，273
MPI_Comm_rank，243
MPI_Comm_size，243
MPI_Comm_split，272
MPI_COMM_WORLD，243
MPI_Finalize，242

MPI_Gather, 263
MPI_Gatherv, 264
MPI_Get_count, 246
MPI_Init, 242
MPI_Irecv, 256
MPI_Isend, 256
MPI_PROC_NULL, 253
MPI_Recv, 244
MPI_Reduce, 261
MPI_Request_free, 257
MPI_Scan, 263
MPI_Scatter, 264
MPI_Scatterv, 265
MPI_Send, 244
MPI_Sendrecv, 248
MPI_Sendrecv_replace, 248
MPI_Status, 245
MPI_SUCCESS, 242
MPI_Test, 256
MPI_Wait, 256
multi-prefix（多前缀）, 189
multiknapsack（多背包）, 508
multilevel work-distribution（多层任务分配）, 505, 513
multiple instruction stream, multiple data stream（多指令流，多数据流）, 26, 74, 506
multiple instruction stream, single data stream（多指令流，单数据流）, 74
multistage network（多级网络）, 36
  butterfly（蝶形）, 78, 79
  omega, 36, 37, 78
    blocking nature of（阻塞特性）, 38
  perfect shuffle（完全混洗）, 36
multithreading（多线程）
  impact on bandwidth（对带宽的影响）, 23
mutex（互斥锁）
  alleviating overheads（减少开销）, 292
  errorcheck（错误检查）, 300
  normal（正常的）, 300
  overheads of serialization（串行化开销）, 292
  recursive（递归的）, 300
mutex-unlock（互斥锁解锁）, 288
mutual exclusion（相互排斥）, 287

negative-weight cycles（负权圈）, 463
network topologies（网络拓扑结构）, 33
node latency（节点延迟，参看per-hop time）
node splitting（节点分裂）, 505
non-minimal routing（非最小路由选择，参看routing mechanism）
non-uniform memory access（非一致内存访问）, 27
noncomparison-based sorting（基于非比较的排序）, 379
nonterminal nodes（非终端节点）, 471
nowait clause in OpenMP（OpenMP中的nowait子句）, 319
NP-hard（NP难问题）, 473
num_threads clause in OpenMP（OpenMP中的num_threads子句）, 312
NUMA（参看non-uniform memory access）

odd-even sort（奇偶排序）, 416
odd-even transposition sort（奇偶转换排序）, 417
  in shellsort（在希尔排序中）, 398
  scalability（可扩展性）, 398
omega network（omega网络，参看multistage network）
Ω notation（Ω符号）, 567
omp for, 315
omp parallel, 312
omp_destroy_lock, 330
omp_destroy_nest_lock, 330
OMP_DYNAMIC, 331
omp_get_dynamic, 329
omp_get_max_threads, 329
omp_get_nested, 329
omp_get_num_procs, 329
omp_get_num_threads, 329
omp_get_thread_num, 329
omp_init_lock, 330
omp_init_nest_lock, 330
OMP_NESTED, 322, 331
OMP_NUM_THREADS, 330
OMP_SCHEDULE, 331
OMP_SET_DYNAMIC, 331
omp_set_dynamic, 329
omp_set_lock, 330
omp_set_nest_lock, 330
omp_set_nested, 329
omp_set_num_threads, 329
omp_test_lock, 330
omp_test_nest_lock, 330
omp_unset_lock, 330
omp_unset_nest_lock, 330
one-to-all broadcast（一对多广播）, 149, 166, 185, 187, 341-343, 351, 352, 357, 361, 368, 435, 442
  dual of（对偶）
  on balanced binary tree（在平衡二叉树上）, 153
  on hypercube（在超立方体中）, 153
    with all-port communication（用全端口通信）, 189

on linear array（在线性阵列中），150
on mesh（在格网中），152
pipelining multiple broadcasts（流水线多播），165，357
pseudocode（伪码），156，157
one-to-all personalized communication（一对多私自通信，参看scatter）
O notation（*O*符号），567
open list（开放表），478
OpenMP，311
atomic instructions（原子指令），325
barrier（障碍），322
conditional parallelism（条件并行），312
critical sections（临界段），323
data handing（数据处理），312，327
defining parallel regions（定义并行区域），312
dynamic control（动态控制）
number of threads（线程数），329
environment variables（环境变量），330
for loops（for循环）
scheduling（调度），316
library functions（库函数），328
local variable（局部变量）
initialization（初始化），314
master block（主块），323
memory consistency（内存一致性），326
merging directives（合并命令），320
mutual exclusion（相互排斥），329
nesting directives（嵌套命令），321
number of threads（线程数），312，328
ordered execution（有序执行），325
programming model（编程模型），312
reduction（归约），314
scheduling across for loops（跨越for循环调度），319
single block（单个块），323
splitting for loops（分裂for循环），315
synchronization（同步），322
task parallelism（任务并行化），319
optimal matrix parenthesization（最优矩阵带括号方法），527
optimization equation（优化方程），516
ordered directive in OpenMP(OpenMP中的ordered命令)，325
out-of-order execution（乱序执行），15
overhead（开销），115
methods to reduce（归约方法），132
overhead function（开销函数），197，215，217，227，228，231
sources of（源）
extra computations（额外计算），196
interprocessor communication（处理器间通信），196
load imbalance（负载不平衡），196
processor idling（处理器空闲），196
owner-computes rule（拥有者计算规则），103

packet routing（包路由选择），54
message transfer time（消息传送时间），56
packing（包），189
parallel best-first search（并行最佳优先搜索），496
parallel computing（并行计算）
books（书），8
conferences（参看文献），9
journals（期刊），9
parallel depth-first search（并行深度优先搜索）
analysis（分析），488，485-495
assumptions（假设），486
contention（争用）
in accessing global variables（访问全局变量），489
depth-first branch-and-bound（深度优先分支定界），495
iterative deepening A*（迭代加深A*），496
termination detection（终止检测），490
Dijkstra's token algorithm（Dijkstra令牌算法），490
tree-based algorithm（基于树的算法），491
parallel efficiency（并行效率，参看efficiency）
parallel efficient problems（并行效率问题），227
parallel platforms（并行平台）
communication costs（通信成本），53
message passing（消息传递），53
shared-address-space（共享地址空间），61
control structure（控制结构），25
dichotomy of（二分法），24
interconnection networks（互连网络），32
logical organization（逻辑组织），24
physical organization（物理组织），24，31
parallel processes（并行进程），93
versus processors（与处理器比较），94
parallel random access machine（并行随机访问计算机），9，31-32，75，76，227
BPRAM，76
concurrent write（并发写）
arbitrary（任意），31
common（普通），31
priority（优先级），31
sum（求和），32
CRCW，31

CREW, 31
ERCW, 31
EREW, 31, 32
LPRAM, 76
parallel runtime (并行运行时间), 197
cost-optimal (成本最优的), 219
minimum (最小的), 219
parallel speedup (并行加速比, 参看speedup)
parallel system (并行系统), 195
scalable (可扩展的), 211, 214, 215, 226
ideally scalable (理想可扩展的), 217
unscalable (不可扩展的), 215
parallel tasks (并行任务), 86
characteristics of (特点), 110-112
parallelism (并行化)
implicit (隐式), 12
partial pivoting (部分选主元法), 366
with 1-D mapping (1维映射), 367
with 2-D mapping (2维映射), 368
partitioning (划分), 98
input data (输入数据), 101
intermediate data (中间数据), 101
output data (输出数据), 98
path (路径), 429
cycle (圈), 430
shortest (最短的), 436
simple (简单的), 429
single-source shortest paths (单源最短路径), 436
weight of (权), 430
PC* class of problems (PC*类问题), 227
PE problems (PE问题, 参看parallel efficient problems)
Pentium (奔腾处理器), 12
Streaming SIMD Extensions (SIMD流扩充), 25
per-hop time (每站时间), 54
per-word transfer time (每字传送时间), 54
perfect shuffle (完全混洗), 36
permutation communication operation (置换通信操作), 179, 189
permutations (置换), 189
pipelining (流水线), 12, 141
pivoting (选主元, 参看partial pivoting)
pointer jumping (指针跳转), 189
Poisson equation (泊松方程), 561
polynomial function (多项式函数), 565
degree of (次), 565
linear (线性的), 565
positive definite matrix (正定矩阵), 369
POSIX threads (POSIX线程), 282
Power4, 12
prefetching (预取), 23
example (例子), 23
prefix sums (前缀和), 167, 189, 191, 231
pseudocode (伪码), 168
priority queue (优先队列)
in Johnson's algorithm (在Johnson算法中), 455
private clause in OpenMP (OpenMP中的private子句), 315
problem size (问题规模), 213, 226
process mapping (进程映射)
impact of (影响), 65
processor-time product (处理器-时间乘积, 参看cost)
producer-consumer (生产者-消费者), 290
using condition variables (使用条件变量), 296
pthread_attr_destroy, 299, 301
pthread_attr_init, 299
pthread_attr_setdetachstate, 299
pthread_attr_setguardsize_np, 299
pthread_attr_setinheritsched, 299
pthread_attr_setschedparam, 299
pthread_attr_setschedpolicy, 299
pthread_attr_setstacksize, 299
pthread_cancel, 301
pthread_cond_broadcast, 298
pthread_cond_destroy, 296
pthread_cond_init, 296
pthread_cond_signal, 295
pthread_cond_timedwait, 298
pthread_cond_wait, 295
pthread_create, 283
pthread_join, 284
pthread_mutex_init, 289
pthread_mutex_lock, 288
pthread_mutex_t, 288
pthread_mutex_trylock, 292, 294
pthread_mutex_unlock, 288
pthread_mutexattr_init, 301
pthread_mutexattr_settype_np, 301
Pthreads, 282

quadratic function (二次函数), 565
quadratic-assignment (二次分配), 508
quicksort (快速排序), 96, 399
constructed binary tree (结构化二叉树), 402
CRCW PRAM, 402
divide-and-conquer (分治), 399, 401
pivot (主元), 400

selection schemes（选择方案），401，417
radix sort（基数排序），415
randomized routing（随机路由选择），59
reachable vertices（可达顶点），429
read-write locks（读写锁），302
reconfigurable mesh（可重构格网），74，80
recursive decomposition（递归分解，参看decomposition）
recursive doubling（递归加倍），149
reduction（归约），75
all-reduce（全归约），166，185，187，189
all-to-all（多对多），157，187
all-to-all reduction（多对多归约），191，192
all-to-one（多对一），149，187，192
all-to-one reduction（多对一归约），149，166，192
dual of（…的对偶，参看dual）
reduction clause in OpenMP，314（OpenMP中的reduction子句）
reentrant functions（可重入函数），285
RGC（参看Gray code）
ring network（环形网络），44
routing mechanism（路由选择机制），63
adaptive（自适应的），64
deterministic（确定性的），64
dimension-ordered routing（维序路由选择），64
E-cube routing（E立方体路由选择），64
minimal（最小值），63
non-minimal（非最小值），63
XY-routing（XY路由选择），64
RP-3，560
RTA（参看recursive transposition algorithm）
runtime（运行时间，参看parallel runtime）
runtime scheduling in OpenMP（OpenMP中的运行时调度），318

sample sort（样本排序），412
hypercube（超立方体），418
MPI implementation（MPI实现），270
sample selection（样本选择），412
splitters（分割器），412
satisfiability problem（可满足性问题），492
scalability（可扩展性），208-218，226，227
scalability metrics（可扩展性度量），222
scaled speedup（可扩展加速比），223
memory-constrained（内存受限的），223
time-constrained（时间受限的），223
serial fraction（串行部分），225
scaled-speedup（可扩展加速比，参看speedup，scaled）
scaling（扩展），205
time-constrained（时间受限的），231，376
scan（扫描，参看prefix sums）
scan vector model（扫描向量模型），189
scatter（散发），167，187
dual of（…的对偶，参看dual）
on hypercube（在超立方体），169
with all-port communication（全端口通信），189
on linear array（在线性阵列），191
on mesh（在格网），191
schedule clause in OpenMP（OpenMP中的schedule子句），315，316
scientific computing（科学计算），537
books（书），9
search overhead factor（搜索开销因子），480
sections directive in OpenMP（OpenMP中的sections命令），319
selection（选择），189
serializability（可串行性），287
serializable schedule（串行调度），46
SGI Origin 2000，27，30
shared address space（共享地址空间），27
shared clause in OpenMP（OpenMP中的shared子句），313
shared-address-space architecture（共享地址空间结构），29
in parallel graph search（在并行图搜索中），496，498-500，508
shared-address-space programming（共享地址空间编程）
directives（命令），279
lightweight processes（轻量级进程），279
processes（进程），279
threads（线程），279
shared-memory computer（共享内存计算机），29
shellsort（希尔排序），398
hypercube（超立方体），417
performance（性能），399
shortest path（最短路径），436，437
SIMD（参看single instruction stream，multiple data stream）
single directive in OpenMP（OpenMP中的single子句），323
single instruction stream，multiple data stream（单指令流，多数据流），25，26，28，74，506
advantages of（优点），26
examples of（例子），75
single instruction stream，single data stream（单指令流，单数据流），74
single-level work-distribution（单层任务分配），512，

513
single-port communication（单端口通信），148，186，189
single-source shortest paths（单源最短路径），436，455
Bellman-Ford algorithm（Bellman-Ford算法），463
Dijkstra' s algorithm（Dijkstra算法），436，438，463
CRCW PRAM，463
EREW PRAM，463
MPI implementation（MPI实现），268
for multistage graph（多级图），518
Johnson's algorithm（Johnson算法），455，464
using apriorinty queue（使用优先队列），456
using distributed queues（使用分布式队列），458
smoothsort（平滑排序）
hypercube（超立方体），418
snoopy cache systems（侦听高速缓存系统），49
performance of（性能），49
sorting（排序），379
comparison-based（基于比较的），379
external（外部的），379
internal（内部的），379
lower-bound（下界），380
noncomparison-based（基于非比较的），379
process enumeration（进程枚举），380
stable（稳定的），415
sorting networks（排序网络），382，384
bitonic（双调），384
comparator（比较器），380
depth（深度），382，386
odd-even sort（奇偶排序），416
spanning binomial tree（生成二叉树），189
spanning forest（生成树林），432，446，447，449
merging different forests（合并不同的树林），447
spanning tree（生成树），189，432，446，447
sparse graphs（稀疏图），431，450-451
examples（例子），450
grid graphs（网格图），451
work imbalance（任务不平衡），451
speculative decomposition（推测性分解，参看decomposition）
speculative execution（推测性执行），16
speech recognition（语音识别），473
speedup（加速比），198，198，210
scaled（可扩展的），230，375
superlinear（超线性的），200，228
caches（高速缓存），200
suboptimal serial algorithm（次优串行算法），201
speedup anomalies（加速比异常），508
speedup anomalies in parallel search（并行搜索中的加速比异常），501-505
acceleration anomalies（加速异常），502
analysis for DFS（DFS分析），502
assumptions（假设），503
best-first search（最佳优先搜索），502
deceleration anomalies（减速异常），502
SPMD，234
stack splitting（堆栈分割），482，505
star-connected network（星形连接网络），39
startup time（启动时间），54
state-space graph（状态空间图），471
nonterminal node（非终端节点），471
states（状态），471
terminal node（终端节点），471
static interconnection networks（静态互连网络，参看interconnection networks）
store-and-forward routing（存储转发路由选择），54
message transfer time（消息传送时间），54
subgraph（子图），430
Sun Ultra HPC，27
superlinear speedup（超线性加速比），228，480
superscalar execution（超标量执行），12
example（例子），13
symmetric matrix（对称矩阵），431
synchronization（同步）
removing it（消除），443

t_h（参看per-hop time）
t_s（参看startup time）
t_w（参看per-word transfer time）
task parallelism（任务并行化），140
task-dependency graph（任务依赖图），86
task-interaction graph（任务交互图），92
tautology verification（重言式验证），506
terminal node（终端节点），471
Θ notation（Θ符号），567
thread（线程），280
threadprivate directive in OpenMP（OpenMP中的threadprivate命令），328
threads（线程）
accessing shared data（访问共享数据），287
advantages of（优点），281
attributes（属性），298
cancellation（取消），301
creation（创建），282
join（加入），284
latency hiding（延迟隐藏），281

logical memory model（逻辑内存模型），280
motivation for（动机），281
mutex-locks（互斥锁），288
mutual exclusion（相互排斥），287
portability（可移植性），281
reentrant functions（可重入函数），285
scheduling and load balancing（调度及负载平衡），281
termination（终止），282, 284
time（时间，参看parallel runtime）
time-constrained scaling（时间受限的扩展，参看scaling）
total exchange（总体交换，参看all-to-all personalized communication）
transitive closure of a graph（图的传递闭包），445
transpose algorithm（转置算法，参看fast Fourier transform）
tree（树），430, 446
spanning（生成），432
tree network（树形网络），42
dynamic（动态的），42
fat tree（胖树，参看fat tree）
static（静态的），42
tree search（树搜索）
static allocation（静态分配），480
triangular matrix（三角矩阵），353, 369, 370
triangular system（三角系统），353, 369, 370
twiddle factors（旋转因子），538, 551-552, 563

UMA（参看uniform memory access）
undirected graph（无向图），429, 431, 432, 433
connected（连通的），430
connected components（连通分量），446
uniform memory access（一致内存访问），27
update protocol（更新协议），47

vertex（顶点），429
adjacent（邻接的），429
connected（连通的），445
reachable（可达的），429
vertex collapsing（顶点压缩），464
Very Long Instruction Word Processors（超长指令字处理器），15
VLIW（参看Very Long Instruction Word Processors）
VLSI floor-plan optimization（VLSI布置图优化），473
VPP 500, 74

wall-clock time（挂钟时间），195
wave（波）
spreading of computation（计算的扩散），461
weighted adjacency matrix（加权邻接矩阵），431
weighted graphs（加权图），430
work-splitting（任务分割）
cutoff depth（截止深度），483
half-split（半分割），482
tree search（树形搜索），482

XY-routing（XY路由选择，参看routing mechanism）