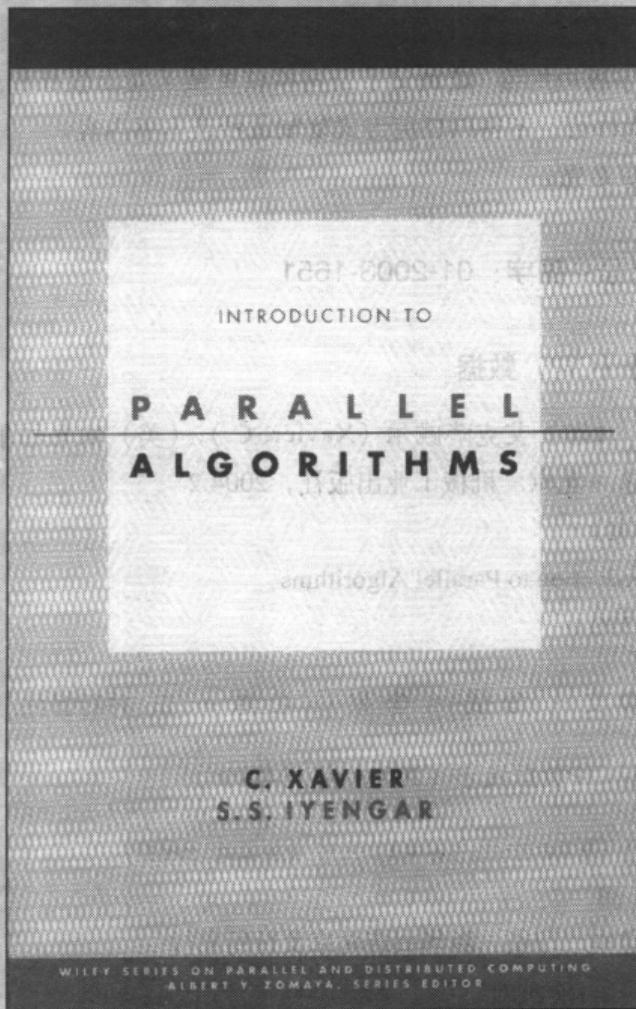


计 算 机 科 学 丛 书

并行算法导论

(印) C. Xavier (美) S. S. Iyengar 著 张云泉 陈英 译



Introduction to Parallel Algorithms



机械工业出版社
China Machine Press



中信出版社
CITIC PUBLISHING HOUSE

本书对并行算法作了入门级的介绍，用四部分讲解并行算法的设计过程和最新的设计方法，并对书中所描述的每一个算法提供分析和详细的实现细节。全书包括并行计算的基础，树和图的并行算法，排序、搜索和合并的并行算法及数值算法等内容。其中重点强调了图模型算法。在章节后面附有大量的习题和关于并行计算的参考文献。

本书可以作为大学计算机科学与工程专业高年级学生的并行算法课教材。对于计算机科学、数学和工程领域的研究生、科研工作者和工程师，也是一本不可多得的参考书。

C.Xavier, S.S.Iyengar: Introduction to Parallel Algorithms (ISBN: 0-471-25182-8)

Authorized translation from the English language edition published by John Wiley & Sons, Inc.

Copyright © 1998 by John Wiley & Sons, Inc.

All rights reserved.

本书中文简体字版由约翰·威利父子公司授权机械工业出版社和中信出版社合作出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书版权登记号：图字：01-2003-1651

图书在版编目（CIP）数据

并行算法导论 / (印) 艾克萨威尔 (Xavier, C.), (美) 依恩加尔 (Iyengar, S. S.) 著；张云泉，陈英译。—北京：机械工业出版社，2004.2
(计算机科学丛书)

书名原文：Introduction to Parallel Algorithms

ISBN 7-111-13390-0

I. 并… II. ①艾… ②依… ③张… ④陈… III. 并行算法 IV. TP301.6

中国版本图书馆CIP数据核字（2003）第104979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：刘渊

北京牛山世兴印刷厂印刷·新华书店北京发行所发行

2004年2月第1版第1次印刷

787mm×1092mm 1/16 · 17.5印张

印数：0 001-4 000册

定价：35.00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

译者序

近年来，随着社会众多行业信息化带来的对计算机信息处理能力和计算能力需求的不断提高，以及高性能计算机特别是高性能机群系统的普及和发展，并行计算机越来越多地开始被广大普通计算机专业人员以及其他专业的人员使用。然而，仅仅有并行计算机的硬件平台是不够的，还需要有配套的能充分发挥机器性能的并行计算应用软件。广大并行计算机用户和相关专业的学生迫切需要了解并行算法设计和分析的基础知识。我们翻译此书的目的是推动高性能计算在中国的进一步普及和发展，尤其是在高等院校中的普及和发展。

本书的两位作者C. Xavier博士和S.S. Iyengar博士都是多年从事并行算法设计研究的资深教授，有多年教学经验，并发表了大量的学术论文，出版了多部学术专著。其中Iyengar博士是IEEE的高级成员和IEEE计算机协会杰出访问者（1995~1998年），由于其对图像处理数据结构和算法以及传感器融合（sensor fusing）问题的杰出贡献而于1997年获得著名的IEEE技术成就奖。

本书分四个部分介绍共享存储器计算模型PRAM下的并行算法设计，第一部分介绍并行计算、并行计算机和并行算法设计的基本概念，并行计算机分类、共享存储器并行计算模型、并行算法设计中的数据结构和并行算法常用设计环境，以及几个简单的并行算法；第二部分首先介绍图的基本概念，随后介绍树算法、图算法和弦图的NC算法，以及相关的并行算法；第三部分介绍搜索与合并、排序等数组处理并行算法；第四部分介绍有关数值计算的并行算法，包括线性代数方程组求解并行算法，以及微分、积分、插值和偏微分方程求解的并行算法。

本书的翻译工作由张云泉组织，并翻译第0~1章、第2章第1、2节、第3~4章、第8~9章、第10章第1、2、3节及相应各章习题和习题答案。陈英翻译第2章第3节、第5~7章、第10章第4~8节、第11~12章及相应各章习题。

由于时间仓促，再加上有些专业术语目前国内没有统一的译法，故翻译中的错误或不妥之处在所难免，恳请广大读者不吝指正。

张云泉
中科院软件所并行计算实验室
2003年8月12日

译者简介

张云泉，男，1995年获北京理工大学计算机科学技术系计算机应用专业工学学士学位；2000年获中科院软件所计算机软件与理论专业工学博士学位（硕、博连读）。现为中科院软件所并行计算实验室副研究员，中科院计算机科学开放重点实验室兼职副研究员，中科院软件所并行计算实验室副主任，中国软件行业协会数学软件分会秘书长。主要研究兴趣是：高性能并行计算特别是大规模并行数值软件包的设计与性能优化，性能建模与评价，并行计算模型，平行数据挖掘等。曾获得2000年度中科院院长奖学金优秀奖；2000年度国家科技进步奖二等奖。

陈英，女，分别于1992年和1995年获南京师范大学学士学位和计算数学专业硕士学位。2001年于中科院数学与系统科学研究院获计算数学专业博士学位。主要研究兴趣是：数值分析、数值计算方法、并行计算及其应用等。目前，她是中科院计算所智能中心助理研究员。

前　　言

最近几年，人们学习、设计和分析并行算法的兴趣日渐浓厚。这部分是由于用新技术制造的计算机使得运算成本比较合理，主要还是由于对信息处理能力的需求日益复杂。本书用四部分讲解并行算法的设计：第一部分给出并行算法设计的基础概念。其中对流水线、多处理、分时和共享存储器模型进行重点介绍。由于数据结构是并行算法设计很重要的组成部分，本书专门用一章进行介绍。基于信息处理的重要性，在本书中重点强调图模型算法。对并行算法设计中重要的环境则给出大量的例子进行解释。

本书第二部分介绍图理论问题常用的并行算法。对各种图问题进行研究，并给出相应的并行算法。专门用一章对弦图进行讲解，其中主要介绍弦图判别算法和一些优化问题算法。第三部分介绍对数组进行处理的算法，并用两章对排序、搜索与合并算法进行介绍。

第四部分对数值计算的并行算法进行介绍。其中用一章介绍代数方程并行算法。详细给出了微分、积分和微分方程（包括偏微分方程）的相关算法。另外还对内插值和外插值并行算法进行了介绍。

本书的显著特色是：

- 用详细的例子一步一步讲解并行算法的设计；
- 详尽的并行算法分析与实现；
- 用大量的实例对每一个概念进行解释；
- 每一章都在最后给出了参考文献。

本书是我们两位共同合作的结果，对本书内容负有同等责任。我们欢迎并感谢读者提出评价和意见。

C. Xavier
S.S. Iyengar

致 谢

本书的素材来自于我们关于数据结构和并行算法的研究论文，这些论文刊登在以下杂志上：《IEEE Transactions on Computers》，《Journal of Computer Science and Informatics》，《Journal of Theoretical Computer Science》，《Information Processing Letters》以及《Advances in Computers》。

由于合作者的支持，本书才得以完成。他们是：Abha Moitra, R. L. Kashyap, N. S. V. Rao, Dekl和N. Chandra Sekharan。我们还要感谢Brooks和N.S.V.Rao，是他们仔细地审校了最初的原稿。Sartaj Sahni, Bella Bose, S. Q. Zheng和X. E. Sun所给的评价和建议使本书质量有了实质性的提高，我们也要感谢他们。在C.Xavier准备编写本书时，Rev. Fr. Francis Peter S. J., Rev. Fr. Albert Muthumalai, S. J., Rev. Fr. Antony A. Pappuraj S. J., Rev. Fr. Sebastian S. J., Rev. Fr. Francis Jeyapathy S. J., Dr. G. Arumugam和Dr. S. Ambrose给予了他很大支持。在此对他们表示深深的谢意。

Agnes Xavier太太和Manorama Iyengar太太也给予我们很多鼓励，我们要感谢她们。还要感谢Almighty God对本书的策划，使它能够顺利出版。

C.Xavier
S.S.Iyengar

作者简介

C. Xavier

C. Xavier博士曾获得数学专业理科硕士和哲学硕士学位，以及计算机科学专业的博士学位。他的博士论文是关于并行算法设计的。目前任教于印度Tirunelveli的圣Xavier学院（私立）计算机科学系。他在国际期刊和会议录上发表了多篇并行算法方面的研究论文，并出版了十多部计算机科学教材。印度的新时代国际出版公司（前Wiley东方公司）出版了其中的两本书：《*FORTRAN 77 and Numerical Methods*》和《*Introduction to Computers and BASIC Programming*》。同时，该出版公司即将出版另外一本书《*C Language and Numerical Methods*》。Xavier博士是印度计算机协会的高级终身会员。他是印度大学拨款委员会和印度科学技术部资助的多个项目的首席研究员，曾任1996年8月在印度Tirunelveli举行的关于数学建模和计算机虚拟现实的全国研讨会论文集的主编。他是那次研讨会的联合召集人之一。

S. Sitharama Iyengar

S.S. Iyengar 博士是美国路易斯安那州立大学计算机科学系的教授和系主任。自从1970年和1974年在印度理工大学分别获得硕士和博士学位后，他主要从事高性能算法和数据结构的研究工作，并在路易斯安那州立大学指导了超过27篇博士论文。曾担任过海军研究局（the Office of Naval Research）、国家航空航天管理局(NASA)、国家科学基金会（NSF）、加州理工学院喷气推进实验室、海军部—NORDA、美国能源部、LEQFS董事会和美国陆军部的研究员。他在高性能并行与分布式算法、图像处理和模式识别数据结构、自动导航及分布式传感器网络等领域出版了多本专著（由Prentice-Hall、CRC、IEEE计算机协会等出版社出版），并发表了250余篇研究论文。他还是喷气推进实验室、橡树岭国家实验室和印度科学院的客座教授。

Iyengar博士是《*Neuro Computing of Complex Systems*》丛书的编辑和《*Journal of Computer Science and Information*》的编辑。他还曾担任《*IEEE Transactions on Knowledge and Data Engineering*》、《*IEEE Transcations and SMC*》、《*IEEE Transcations on Software Engineering*》、《*Journal of Theoretical Computer Science*》、《*Journal of Computer and Electrical Engineering*》和《*Journal of the Franklin Institute*》等学术刊物的特邀编辑。

他还是IEEE的高级成员和IEEE计算机协会杰出访问者（1995～1998年）。另外，他从1985年就担任ACM全国讲师，并且是纽约科学院院士。曾担任多个全国和国际会议的程序委员会主席。在医学信息学领域，他是久负盛名的NIH-NLM评审委员会的成员。

1997年，Iyengar博士由于在图像处理数据结构和算法以及传感器融合问题上的杰出贡献而获得著名的IEEE技术成就奖。

1996年，Iyengar博士由于其出色的研究工作获得路易斯安那州立大学杰出教员奖和路易斯安那州立大学老虎运动（Tiger Athletic）基金教学奖。他还是一些工业机构和政府组织（如JPL、NASA等）的顾问。

目 录

出版者的话
专家指导委员会
译者序
前言
致谢
作者简介

第一部分 并行计算基础

第0章 引言	1
0.1 计算机简介	1
0.2 并行计算机	5
0.3 并行处理的概念	6
0.4 高性能计算机	8
0.5 本书的结构和内容	9
参考文献	10
第1章 并行计算要素	11
1.1 并行的层次	11
1.2 并行计算机分类	12
1.2.1 Flynn分类	12
1.2.2 Erlangen分类 (Handler分类)	14
1.2.3 Giloi分类	15
1.2.4 Hwang-Brigg分类	15
1.2.5 Duncan分类	15
1.3 并行计算模型	18
1.3.1 二叉树模型	18
1.3.2 网络模型	20
1.3.3 超立方体 (k -立方体)	21
1.3.4 网格网络	26
1.3.5 金字塔网络	26
1.3.6 星形图	27
1.4 PRAM模型	28
1.5 一些简单算法	32
1.6 并行算法的性能	34
1.7 小结	37

参考文献	37
习题	38
第2章 并行计算数据结构	40
2.1 数组和列表	40
2.2 链接列表	41
2.3 图与树	44
2.3.1 预备知识	44
2.3.2 欧拉图与哈密顿图	48
2.3.3 树	49
2.3.4 图的遍历	57
2.3.5 连通性	58
2.3.6 可平面图	62
2.3.7 染色与独立集	64
2.3.8 团覆盖	65
2.3.9 交图	65
2.3.10 弦图	66
2.3.11 更多的交图	70
2.3.12 图的匹配问题	70
2.3.13 图的中心	71
2.3.14 控制理论	72
2.3.15 图论中的一些问题	73
参考文献	74
第3章 并行算法设计环境	76
3.1 二叉树设计环境	76
3.2 二倍增长	79
3.3 指针跳转	79
3.4 分而治之	82
3.5 划分	83
3.6 小结	86
参考文献	86
习题	86
第4章 简单并行算法	88
4.1 向量内积	88
4.2 矩阵乘法	88

4.3 部分和	90	7.4 路图判别	164
4.4 二项式系数	94	7.4.1 一些概念和事实	164
4.5 范围内最小值问题	98	7.4.2 算法概述	168
参考文献	101	7.4.3 两个UV图的并	169
习题	101	7.4.4 正确性和复杂度	175
第二部分 图模型算法			
第5章 树算法	103	参考文献	177
5.1 欧拉圈	103	7.4 路图判别	164
5.2 给树加根	104	7.4.1 一些概念和事实	164
5.3 后序编号	105	7.4.2 算法概述	168
5.4 后代个数	107	7.4.3 两个UV图的并	169
5.5 顶点层数	107	7.4.4 正确性和复杂度	175
5.6 最低公共祖先	108	参考文献	177
5.7 树收缩	110		
5.8 算术表达式的计算	114	第三部分 数组处理算法	
5.9 森林求根问题	117	第8章 搜索与合并	179
5.10 到根的路	119	8.1 串行搜索	179
5.11 树变为二叉树	123	8.2 CREW PRAM模型下的并行搜索	180
5.12 顶点直径	125	8.3 更多数据的并行搜索	181
5.13 最远邻居	128	8.4 无序数组搜索	182
参考文献	130	8.5 秩合并	182
习题	131	8.6 双调合并	184
第6章 图算法	132	参考文献	187
6.1 简单图算法	132	第9章 排序算法	188
6.2 并行连通度算法	135	9.1 串行排序算法	188
6.2.1 广度优先搜索 (BFS)	135	9.1.1 冒泡排序	188
6.2.2 利用BFS搜索连通支	139	9.1.2 插入排序	189
6.2.3 传递闭包矩阵	141	9.1.3 Shell递减步长排序	190
6.2.4 顶点收缩	141	9.1.4 堆排序	191
6.3 2-连通支	145	9.2 合并排序	193
6.4 支撑树	146	9.3 排序网络	194
6.5 最短路问题	148	参考文献	195
参考文献	151	习题	196
习题	152		
第7章 弦图的NC算法	154	第四部分 数值算法	
7.1 弦图判别	154	第10章 代数方程和矩阵	197
7.2 弦图的极大团	161	10.1 代数方程	197
7.3 CV图的特征	163	10.1.1 几何解释	197
		10.1.2 对分法	198
		10.2 矩阵的行列式	199
		10.3 线性方程组	202
		10.3.1 高斯消元法	205
		10.3.2 Givens旋转	206
		10.4 傅里叶变换	208
		10.5 多项式乘法	215
		10.6 矩阵求逆	217

10.7 Toeplitz矩阵	219	参考文献	237
10.8 三对角方程组	222	习题	238
10.8.1 高斯消元法	222	第12章 微分方程	239
10.8.2 奇偶约化法	223	12.1 欧拉公式	239
参考文献	226	12.2 偏微分方程	239
习题	227	12.3 抛物方程	240
第11章 微分与积分	228	12.3.1 施密特法（求解抛物方程）	242
11.1 微分	228	12.3.2 Laasonen法（求解抛物方程）	246
11.2 偏微分	229	12.3.3 Crank-Nickolson法	248
11.3 定积分	233	12.3.4 三层差分法	249
11.4 插值	235	参考文献	251
11.4.1 线性插值	235	部分习题解答	252
11.4.2 二次插值	236	索引	258
11.4.3 拉格朗日插值	236		

第一部分 并行计算基础

第0章 引言

0.1 计算机简介

并行计算在诸如图像处理、机器人学等许多计算密集的应用领域是重要的核心问题。给定一个问题，并行计算就是这样的过程：把问题分解成子问题，同时计算子问题，最后把子问题的解合并得到原问题的解。本书试图给出各种不同问题的并行算法设计的课程材料。

当冯·诺依曼第一次提出计算机体系结构的时候，他把计算机看成一个快速计算设备。他提出了计算机系统的五大组成部分：

1. 输入单元；
2. 输出单元；
3. 存储单元；
4. 算术逻辑单元；
5. 控制单元。

图0-1中给出该体系结构的框图。控制单元控制整个系统。为了与数据线相区别，图中的控制线用虚线表示。输入单元负责使计算机获取输入。穿孔卡片、穿孔纸带、磁带、磁盘和键盘等都是计算机历史上用过的一些输入设备。输出单元负责计算机的输出。打印机和绘图仪等是流行的输出设备。在过去四十年左右的时间里，主要由于快速电子部件的采用，计算机的计算速度获得了极大的提高。存储单元和算术逻辑单元最早是用半导体器件制成的。

电子工业的发展帮助设计者制造更快的计算机。当电子工业开发出晶体管时，计算机工业就用晶体管制造计算机。

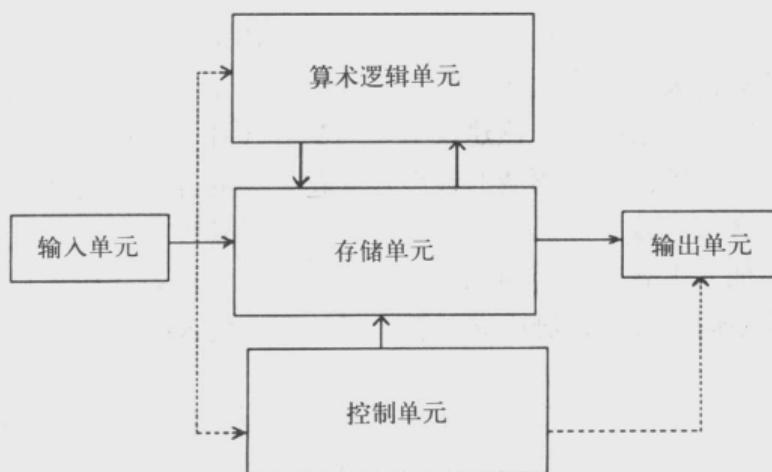


图0-1 冯·诺依曼计算机体系结构框图

在20世纪60年代后期，集成电路的概念在电子工业中成为现实。这使得制造包含几百个晶体管的一厘米小芯片成为可能。这使得我们能够制造出体积更小、存储容量更大和速度更快的计算机。在上述三个阶段制造的计算机分别称为第一代、第二代和第三代计算机。

在20世纪70年代早期，随着超大规模集成电路（VLSI）技术的诞生，电路集成技术取得了巨大进展。1971年由Intel公司实验室的Ted Hoff发明的微处理器是VLSI技术的主要突破。利用微处理器设计的计算机称作微型计算机。微型计算机和其他用VLSI技术制造的计算机称作第四代计算机。表0-1给出一些早期微处理器的详细情况。

表0-1 一些早期微处理器

微处理器	发布年份	芯片中部件数	速 度	意 义
Intel 4004	1971	2 250	11毫秒完成两个4位数相加	第一个微处理器
Intel 8080	1974	4 500	2.5微秒完成两个8位数相加	第一个用于制造通用计算机的微处理器
Mostech (金属氧化半导体6502)	1975	4 300	1微秒完成两个8位数相加	用于家用电脑
Motorola 68000	1979	70 000	3.2微秒完成两个16位数的相乘	内建乘积电路
HP Super Chip	1981	450 000	1.8微秒完成两个32位数的相乘	第一个32位微处理器

1974年，MIT的Ed Roberts制造了他称为Altair的微型计算机。这就是第一台个人计算机。随后Apple Macintosh和IBM PC进入市场。它们在处理数据方面非常快速和有效。Intel的486 DX2具有惊人的66MHz的速度。32位处理器奔腾（Pentium）由于其重新设计的浮点运算单元，声称在数学运算密集的应用中可以发挥出两倍于Intel 486 DX2的速度。最近，133MHz的奔腾处理器也已经发布。Pentium Pro是Intel公司发布的更先进的微处理器。另外一个公司Alpha也发布了主频为400MHz的微处理器。一个日本的科学家小组认为对于满足未来社会人工智能领域的计算需求而言，目前的计算机体系结构是不适宜的。1979年，日本政府任命由Tohru Moto Oka领导一个委员会对20世纪90年代的计算机需求进行预测。该委员会下设三个小组委员会。由Hajime Karatsu领导的第一个小组委员会有10个成员，研究未来需要的计算机类型；由Hideo Aiso领导的第二个小组委员会有12个成员，研究未来需要的计算机体系结构；由Hazuhiko Fuchi领导的第三个小组委员会有13个成员，研究未来需要的计算机的基本概念。这三个小组委员会提交了他们的建议书，并由主席Oka汇总成对未来计算机的最后建议书，这就是后来所谓的第五代计算机。日本政府同意实施第五代计算机项目，并在1981年10月主办了第一届世界第五代计算机大会。在这次大会上，详细讨论了第五代计算机建议书的细节，来自世界各地参加大会的科学家接受了该建议书。当第五代计算机进入市场时，现有的计算机不会变得毫无价值。对于解决计算领域的大多数问题，老的计算机仍然有用。第五代计算机对普通的应用程序不会很有效。然而，它们可以用于信息管理，自然语言处理，语音、文字和图像识别，以及其他人工智能应用领域。第五代计算机的体系结构将和当前的冯·诺依曼计算机体系结构完全不同。日本政府成立了新一代计算机技术研究所(ICOT)来建造第五代计算机。

第五代计算机的数据和指令都存放在计算机的内存中。每当进行处理时，从内存中取出数据。想想我们每天的活动，大部分人都是采用类似的过程。

以医生为病人看病为例（见图0-2）。医生从病人那里收集症状（数据）。而他已经知道所

有的病和其相应的症状，这些都存放在他的记忆里面。医生通过比较记忆中记录的数据和病人的症状诊断病症。

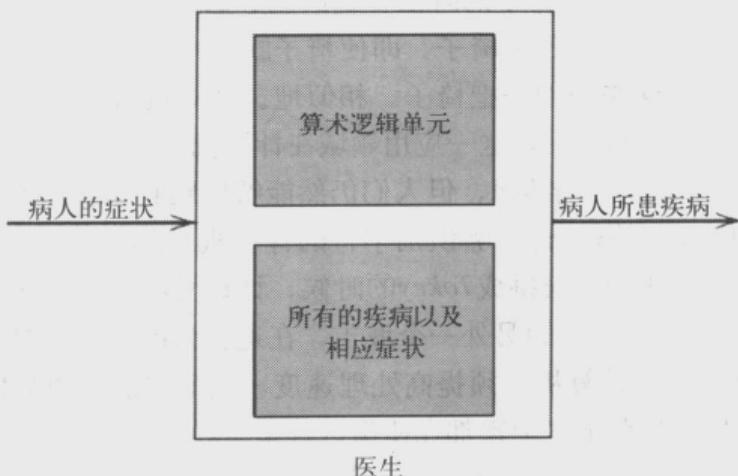


图0-2 医学诊断

在没有医生的情况下，我们能否利用计算机呢？也就是说计算机能否代替医生？科学家们已经接受了这一挑战，试图研制出能够解决这一问题的计算机。这类应用需要在几分之一秒的时间内完成大量的处理操作。这些问题都属于人工智能领域。用于解决人工智能问题的计算机系统就是专家系统。下列是人工智能的一些其他应用领域：

- 自然语言处理（NLP）；
- 图像识别和处理；
- 模式识别；
- 字符识别；
- 语音识别；
- 天气预报；
- 医学诊断；
- 智能机器。

自然语言处理 以英语、法语、德语等语言的语句为例，人能够理解这些语言并且采取相应的行动。根据上下文的不同，同样的语句可以有不同的含义。例如，考察句子

Joseph called his friend a taxi.

这是说Joseph给他朋友叫了一辆出租车呢，还是他用“出租车”作为朋友的外号？该语句的含义取决于上下文。同一语句可以有不同的含义，不同的语句可以有相同的含义。人可以根据上下文理解这些语句。在计算机中存储与语言上下文无关的文法，并把它用于语言理解和翻译，就是自然语言处理（NLP）。

图像识别和处理 各国发射的人造卫星不断发回从外层空间获得的地球图片。通过分析这些图片中的图像，可以发现天气变化、识别森林和农业区域等。图像处理的另一个有趣应用是通过照片识别人。在扫描一个人的照片之后，计算机可以生成这个人的图像，并可以修改诸如发型、胡须或髭等特征。这种应用对警察部门追捕罪犯有巨大作用。一幅图像通常用像素的矩阵表示。每个像素用颜色、强度、亮度等数据表示。在特定的应用中，一幅图像每英

寸可以有14 400个像素，并要转换成数字数据。这使得这类应用有很强的计算强度。在图像的实时处理中，整体的响应时间必须保证尽可能短。因此，加快像素处理的速度是很必要的。

模式/字符识别 在看到一把椅子的时候，我们能够认出它是一把椅子。为什么？它有四条腿。一头牛也有四条腿。但牛不是椅子。即使椅子断了一条腿，我们仍然可以认出它是椅子。我们用一种直观的方法来识别一把椅子。相似地，计算机可以被教会通过视觉来识别一把椅子，这涉及到学习理论的应用。这一应用领域在计算机科学中称为模式识别。

不同的人会写出不同风格的字母A，但人们仍然能够识别它。计算机识别手写字体的能力就是字符识别。在日本，当第一台能够识别字符的计算机在邮局投入到邮件分检工作中时，其识别精度为90%。当Tokyo被误拼成Tokey的时候，计算机不能容忍这一拼写错误，但它把Tokey当做没有编入其城市一览表的另外一个城市。在这种情况下，需要计算机有一定的容错能力。为了提高容错能力，计算机必须提高处理速度。从第一次应用邮件分检机器之后，又开展了大量的改进工作，目前很多国家都在使用这种系统。

在字符识别过程中，需要大量的计算。一个手写的字符首先通过摄像机扫描成高清晰度的图像，例如每英寸14 400个像素（见图0-3）。图像的颜色、光的强度和其他参数转换成数字表示。把该数字表示与字符的标准数字表示（比如A）进行比较。从这个比较中，通过复杂的过程，对所观察到的字符是A的概率进行计算。对于所观察字符为B、C、…的概率也重复同样的计算过程。最后比较所有得到的概率，并把手写字符识别成概率最高的那一个。

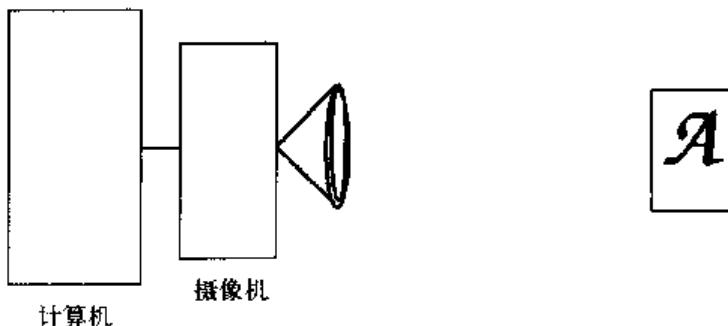


图0-3 字符识别

从上面的讨论中可以明显看出概率的计算量很大：

$$\{P(a) / a = A, B, C, \dots\}$$

其中 $P(a)$ 代表观察到的字符为 a 的概率。对于这类应用，普通的计算机是不能胜任的。

语音识别 语音识别是辨别一个人声音的能力。人们能够轻松识别不同的语音模式。由于任何两个人的语音都不同，设计出能够识别语音的计算机程序是很困难的任务。因而科学家在该领域还没有取得太多进展。已经设计出的计算机只限于能够识别特定人而不是任何人的语音。

天气预报 天气预报是通过获得大量的数据，如温度、气压、湿度等，然后对这些数据进行复杂的数值计算而得出的。这些复杂的计算需要非常快速的计算机，而且要求必须迅速给出预报结果。大气区域被分成很多小区域，并以固定的时间间隔测量湿度、气压和其他气象条件。这些也要与卫星拍摄的照片进行比较。在这些数据的基础上，计算机对未来24小时或48小时的天气进行预报。如果观测数据是即时的，只有完成计算、得到结果，并在24小时流逝之前把结果广播给公众，对未来24小时的天气预报才算有用。天气预报结果的可靠性随着

数据量的增加而提高，即分成的区域越小，收集和处理的数据越多，预报的可靠性就越高。但是，处理更多的数据可能需要普通的计算机超过24小时的处理时间。因此，为了快速得到更可靠的预报结果，目前的计算机是不够的。

医学诊断 之前已经讨论过这个应用。能够进行诊断的计算机根据病人的症状，进行疾病诊断。

智能机器 机器人是能够胜任特定工作（比如组装汽车、充当接待员等）有特殊目的的自动计算机。它们需要上面提到的多种不同能力。为它们编写的软件因此非常复杂，并需要可想象到的最快的计算速度。自治移动机器人（Autonomous Mobile Robot, AMR）是能够进行自治决策和行动的人工智能操作系统。AMR能够管理自己完成特定目标，并同时管理自己的资源和维护系统的一致性。这些能力来源于它们在感应到的和计算得到的信息基础上，与不断变化的环境进行交互的能力。AMR的设计包括如下方面的研究任务：

1. 快速多模型感知和集成：通过多种不同的传感器快速感应外部事件并把这些信息进行有意义集成的能力；
2. 实时响应：在没有不适当延迟的情况下，作出决策并采取适当的行动达到目标的能力；
3. 实时中断：中断当前的正常操作，对发生在周围的外部事件进行及时反应并在处理后返回被中断任务的能力；
4. 容错：在内部出现错误的情况下，依赖于其他功能部件继续操作的能力。

上述工作涉及高度复杂和快速的处理，目前的计算机不能胜任。

随着技术的进步使此类系统更加接近实用，AMR研究的实际重要性在稳步提高。工业机器人已经在涉及单调和琐碎任务的工业应用和有危害环境（如核反应堆等）中得到应用。为深海挖掘和打捞操作、太空服务/组装任务、化工厂等有毒环境维护活动而设计的下一代机器人已经列入日程。

高性能的第五代计算机的应用还包括飞机试航、新药开发、石油勘探、建模融合、反应堆、实时经济规划、密码分析、天文学、生物医学分析、地震学、空气动力学、原子物理学、核物理学和等离子物理学等。9

0.2 并行计算机

上一节，我们已经看到当前的计算机在一些重要应用中无能为力。本节将介绍针对重要应用所使用的并行计算机的概念。图0-4给出一个简单的计算机模型。为了简化，这里没有考虑输入输出。处理器与内存相连接。它读取放在内存中的数据，在处理之后再存入内存。由于VLSI技术的高度进步，包含有 10^6 个逻辑门的高性能处理器芯片已经出现。其线宽非常细，小于0.5微米（1微米=10⁻⁶米），存储密度达到每平方厘米1000KB。一种最新的Alpha微处理器的主频据称达到了400MHz。通过这些芯片，我们获得了所谓的可能的最高速度，科学家开始怀疑能否建造更快的设备。遗憾的是，这些最快的处理器仍不能满足在气象预报、海洋学、空间物理学、空气动力学、图像处理和遥感等领域的需要。这些在上节已经讨论过了。

在普通的计算机里，一般只有一个处理器。为了进一步提高性能，就需要更多的处理器。此类计算机就称作并行计

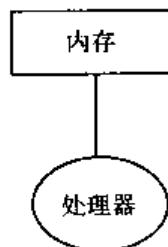


图0-4 串行计算机模型

10

算机。只有一个处理器的计算机称作串行计算机。我们可以用前一节介绍的字符识别应用来解释什么是并行（参见图0-3）。当用串行计算机的时候，概率 $P(A), P(B), \dots$ 在一个处理器上顺序计算。另一个方法是可以用多个处理器同时计算这些概率。如图0-5所示。

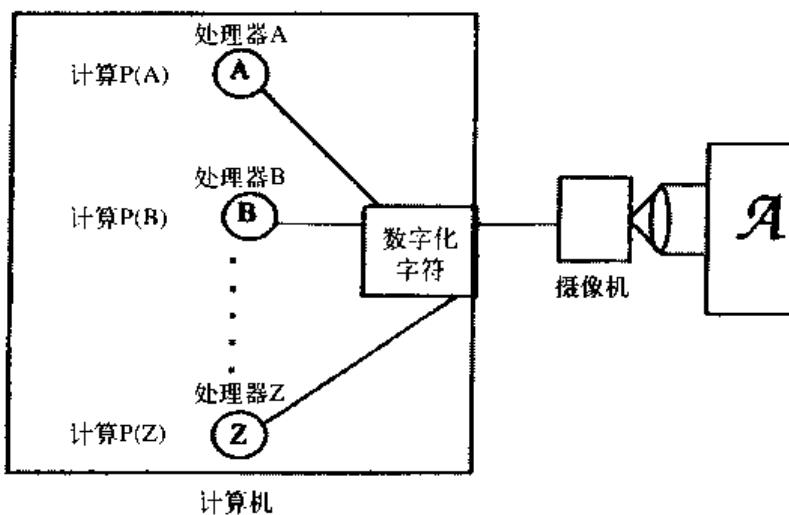


图0-5 字符识别的并行性

0.3 并行处理的概念

要用并行计算机求解问题，必须把问题分解成能够并行求解的子问题。而后这些子问题的结果需要进行高效的重新组合得到主问题的最后结果。把每一个非常大的问题分成子问题是不容易的，这是由于子问题之间可能存在数据相关性。由于数据相关性，处理器之间必须相互通信。需要解决的重要一点是两个处理器之间的通信时间。通常，两个处理器之间的通信时间与处理时间相比是很高的。由于上述因素，为了得到好的并行算法，通信方案应该进行很好地规划。

我们通过一个生活中的例子来说明可分解性和数据相关性。考察一个有人力资源和其他资源的建筑承包商能在五个月内造出一幢房子。假定该承包商不能同时造一幢以上房屋。如果房屋建筑委员会给该承包商提供造100幢房子的合同，他一个接一个地去造，需要500个月完成任务。图0-6仅仅演示四幢房子的情况。如果房屋建筑委员会想加快房屋的建造速度，他可以雇佣100个不同的建筑承包商同时建造100幢房子。在这种情况下，可以在五个月内完成建造100幢房子的任务。如果没有100个建筑承包商，而只有10个，那么可以在50个月内完成。

11

图0-7演示两个建筑承包商在15个月内完成6幢房子的建造过程。这是一个工作容易分解的例子。这里的每幢房子都是独立的单元，因此我们能够容易地对主问题进行分解。

现在让我们来考虑一幢房子的建造工作。把它分成五个步骤：

第1步，建造地基；

第2步，建造地上结构；

第3步，完成木工部分，如安装窗户、门等；

第4步，安装电气系统；

第5步，油漆和装饰。

值得注意的是，上述五个步骤不能同时进行，每个步骤都必须等待前面的步骤完成。可以说它们是内在串行的。在这种情况下，把任务分成子任务是困难的事情。然而，如果一个

承包商有五组不同的人完成上述五个步骤，他或她就可以更高效地完成该任务。让我们假定每组人可以在一个月内完成自己的任务。当第一幢房子的第1步完成之后，第二组人将开始第一幢房子的第2步。与此同时，第一组人将开始第二幢房子的第1步。这些在图0-8中给出。

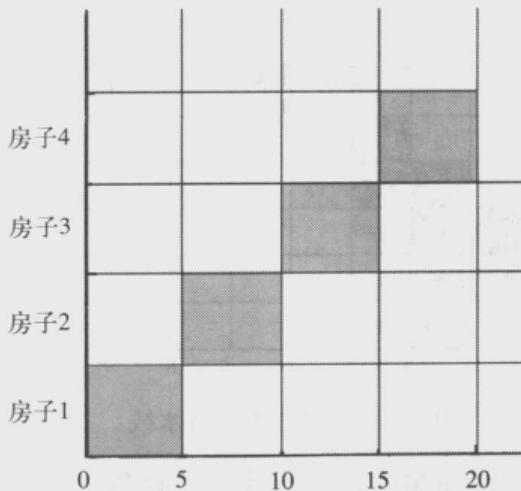


图0-6 串行房屋建造



图0-7 房屋建造中的并行性

这一并行过程就称作流水线。在上面给出的例子中，在第五个月里，所有的组都工作在不同的房子上。

第一组在进行第五幢房子的第1步；

第二组在进行第四幢房子的第2步；

第三组在进行第三幢房子的第3步；

第四组在进行第二幢房子的第4步；

第五组在进行第一幢房子的第5步。

在第五个月结束的时候，第一幢房子已经完成。从第五个月开始，每个月都完成一幢房子。在这种情况下，承包商能够以每月一幢的速度进行房屋的建造。

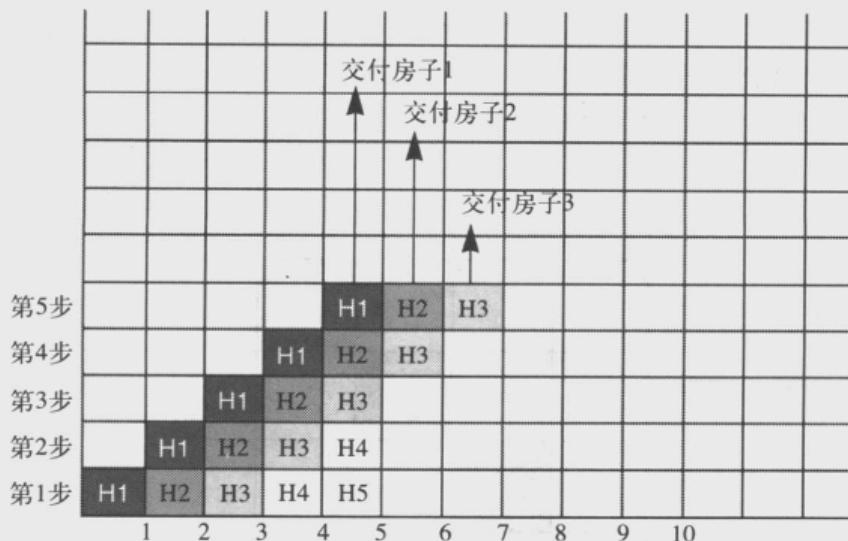


图0-8 房屋建造中的流水线

让我们用另外一个例子来说明问题的可分解性和子问题之间的相互依赖性。假定有一家公司，需要安装100台相同的机器。如果有一个工程师小组，他们只能每次组装一台机器，那么他们能在100个时间单位内完成这项工作。如果有100个不同的工程师小组能同时工作，那么就可以在一个单位时间内完成任务。如果每个小组包含三个分组，分别进行设备安装（机械工程师）、电气安装（电气工程师）和测试（过程控制人员），那么可以流水线操作。

以上我们讨论了流水线的概念。并行处理中的另一个重要概念是多处理。它是指不同处理器同时处理多个任务。一般分成两大类：

1. 多处理器；
 2. 多计算机。

在多处理器系统中，多个处理器同时工作，并共享存储器。在多计算机中，有一组处理器，每个处理器都有足够的本地存储器。处理器之间的通信是通过消息进行的。这也称作分布式处理。这种情况下既没有公用存储器也没有公用时钟。

0.4 高性能计算机

由于技术进步的刺激，过去的四十五年间出现了大量的硬件设计方案，这些推动计算机工业经历了前所未有的快速变化和发展。从物理上看，显著的标志就是计算机基本部件从继电器和真空管（20世纪40年代至50年代）发展到目前的超大规模集成电路（IC）。

由于上述技术的进步和设计的改进，计算机经历了从20世纪50年代速度很慢的单处理器到今天的高性能计算机（包括峰值速度是早期计算机成千上万倍的超级计算机）的显著变化。工程师和科学家对更强大数字计算机的需求是推动数字计算机发展的主要力量。获得高速计算能力一直是最具挑战性的需求。计算机工业界对这些挑战的回应是丰富多彩的，其结果就是在短短的四十年左右的时间里计算机所取得的显著进展。这些从这几十年里所开发的商用和试验性高性能计算系统中可以很好地体现。上面提到的各种各样的科学技术应用对更强大系统的需求在21世纪也不会减弱。

革命性的设计使得技术转变进一步加快，这经常伴随技术的改进而出现。现代计算机最常见的设计技术是加入并行的特性。自从20世纪60年代以来，文献上提出了成百上千的高度

并行体系结构，许多在70年代被建造并投入使用。

1972年，Illiac IV在NASA Ames研究中心投入运行，这是德州仪器公司的第一台并行计算机；同年，德州仪器公司先进的科学计算机（TI-ASC）在欧洲投入使用；1974年，控制数据公司（CDC）的第一台并行机Star-100在Lawrence Livermore国家实验室投入使用；1976年，Cray公司第一台并行机Cray-1在Los Alamos国家实验室投入使用。

上面提到的机器不但是创新设计的开拓者，赋予了机器前所未有的计算能力，而且也是后来发布的一系列更加强大机器的先驱。因此，在它们依然名声显赫的时候，便迅速地让位给了其他几代更加强大的计算机，并最后发展成现代的超级计算机。到20世纪80年代，大量的20世纪60和70年代生产的高速处理器或者停止运行或者扮演越来越次要的角色。与此同时，处理速度的增加和性能价格比的提高从未减弱。最后的结果就是引入了新的根本的体系结构设计理念，如精简指令集计算机RISC（一种广泛商业化的多处理）和超大规模并行MPP的出现。这样，Illiac IV在1981年停止运行；TI-ASC从1980年开始不再生产；从1976年开始，Star-100演化成CDC Cyber 203（已经停产）和Cyber 205，这也标志着CDC公司进入到超级计算领域；Cray-1（流水线单处理器）已经演化成Cray-1S，比原来的Cray-1有大得多的存储容量。下面是一些性能更高的计算机：

- Cray XMP4（4处理器，128M字超级计算机，峰值速度840MFLOPS）；
- Cray-2（256M字，4处理器，可重构超级计算机、峰值速度2GFLOPS）；
- Cray-3（16处理器，2G字超级计算机，峰值速度16GFLOPS）。

其他在20世纪80年代生产的超级计算机包括Eta-10、Fujitsu VP-200、Hitachi S-810和IBM 3090/400/VF。到20世纪80年代，60和70年代的高速处理器已经发展成了80年代到90年代的超级计算机。

别的一些计算机也有历史意义，虽然其开发的最初目的不是数值计算。它们包括Goodyear公司的STARAN和卡内基-梅隆大学的C.mmp系统。还有些机器有一定的历史意义，虽然没有得到商用，比如Burroughs公司的科学处理器（Scientific Processor，CSP）。 15

阵列处理器的问世 Illiac-IV只有64个处理器。另外一些计算机有大量的处理器，如Denelcor的HEP、国际计算机公司的数据阵列处理器（ICL DAP）、NASA Langley研究中心的有限元机器、Lawrence Berkeley实验室的MIDAS、加州理工学院的Cosmic Cube、德州大学的TRAC、卡内基-梅隆大学的CM*、马里兰大学的ZMOB、华盛顿大学和普度大学的Pringle、NASA Goddard空间飞行中心的超大规模并行处理器（MPP）等等。其中只有少数几个（如MPP、ICL DAP）主要是为数值计算设计的，其他的都是用于研究。

0.5 本书的结构和内容

本书分四个阶段介绍并行算法的概念。第一部分介绍并行算法设计的基础概念，包括流水线、多处理、分时和共享存储器模型。由于数据结构是并行算法很重要的组成部分，专门用一章介绍各种数据结构。图模型在信息处理中是很重要的数据结构，因此加以特别强调。对并行算法设计中一些很重要的范例作了详细的说明。

本书的第二部分介绍关于图的并行算法。研究了各种图问题的类型并设计其并行算法。特意用一章讨论弦图类型，主要是针对它的识别算法和一些优化问题的算法。

第三部分介绍数组的操作。在两章里给出了很多关于排序、搜索和合并问题的重要算法。

16 第四部分介绍数值计算的并行算法。其中一章给出代数方程的并行算法。另外两章介绍微分、积分和微分方程（包括偏微分方程）的算法。

参考文献

- Aho, A., Hopcroft, J., and Ullman, J. (1974). *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.
- Akl, S. G. (1989) *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, NJ.
- Cook, S. A. (1981) Towards Complexity Theory of Synchronous Parallel Computation, *L'enseignement Mathématique XXVII*, pp. 99–124.
- Cook, S. A. (1983) Overview of Computational Complexity, *Commun. ACM*, **26**(6), 400–409.
- Duncan, R. (1990) A Survey of Parallel Computer Architectures, Private Communication.
- Golub, G. and Ortega, J. M. (1993) *Scientific Computing: An Introduction with Parallel Computing*, Academic Press, Boston and New York.
- Haynes, L. et al., (1982) A Survey of Highly Parallel Computers, *Computer*, **15**(1), 9–24.
- Hockney, R. and Jesshope, C. (1983) *Parallel Computers*, Adam Hilger, Bristol and Philadelphia.
- Hockney, R. W. (1987) Classification and Evaluation of Parallel Computer Systems, Springer-Verlag Lecture Notes in Computer Science, No. 295, pp. 13–25.
- Hwang, K. and Briggs, F. A. (1984) *Computer Architecture and Parallel Processing*, McGraw-Hill, New York.
- Hwang, K. and Degroot, F. (1989) *Parallel Processing for Super Computers and Artificial Intelligence*, McGraw-Hill, New York.
- Kuck, D. J. (1971) A Survey of Parallel Machine Organisation and Programming. *ACM Comput. Survey*, **9**, 29–59.
- Moto-oka, T., ed. (1982) *Fifth Generation Computer Systems*, North-Holland, New York.
- Wold, E. H. and Despain, A. M. (1984) Pipeline and Parallel-Pipeline FFT Processors for VLSI Implementations, *IEEE Transactions on Computers*, **C-33**, 414–426.

第1章 并行计算要素

前一章已经介绍了对并行计算机的需求。本章重点介绍并行计算机的基本概念。并行计算机的分类和各种不同的并行计算模型也在本章进行详细讨论。

1.1 并行的层次

并行可以在不同的层次实现。例如，如果10个作业都不相同且两两独立，那么这10个作业可以分配给10个不同的机器。由于这些作业并行执行，其并行性是高层次的。通常把这种并行称为作业级并行或者程序级并行。为了进一步提高程序的效率，我们可以发掘下一个层次的并行。每个作业可以分成更小的子任务。这些子任务可以并行执行，并通过合并结果得到最终结果。对每一个子任务，都有一个子程序且这些子程序可以并行执行，这就是通常所谓的子程序级并行。任何程序或子程序都有若干语句，这些语句可以并行执行，这种并行称为语句级并行。在一个语句里可以执行多个操作。我们可以考虑这些操作的并行执行，这称为操作级并行。通常，一个操作又由若干微操作组成。比如，两个变量 A 和 B 相加，并把结果存到 C 变量里。这个操作包含如下微操作：

1. 把 A 的内容加载到累加器里；
2. 把 B 的内容和累加器里的内容相加；
3. 把累加器里的内容存入变量 C 中。

如果并行执行微操作，则称为微操作级并行（见图1-1）。程序级并行和子程序级并行很容易理解，不需要进一步的解释。为了说明语句级并行，我们以下面的算法为例：

For $i = 1$ to n do

$x_i \leftarrow x_i + 1$

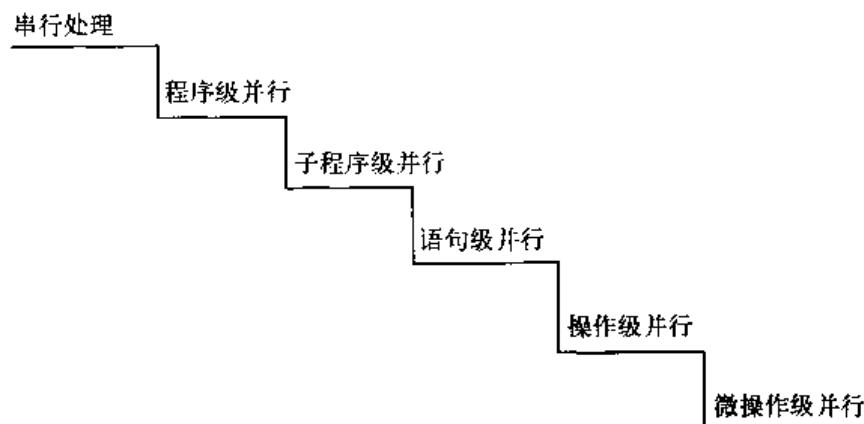


图1-1 并行的层次

在上面的算法中，对每个 x_i 加1。这个语句串行执行 n 次，因此需要 $O(n)$ 的时间。这是可以并行的，可以用 n 个处理器同时在 $O(1)$ 的时间完成。在这种情况下，我们可以把算法写成下面的形式：

```

For  $i = 1$  to  $n$  do in parallel
     $x_i \leftarrow x_i + 1$ 
End parallel

```

上面的算法需要 n 个处理器，它们可能是 P_1, P_2, \dots, P_n 。处理器 P_i 执行 $x_i \leftarrow x_i + 1$ 操作。所有处理器同时工作，并在 $O(1)$ 时间内完成。这种类型的并行被称为语句级并行。

现在考虑下面给出的算法：

```

 $S \leftarrow 0$ 
For  $i = 1$  to  $n$  do
     $S \leftarrow S + x_i$ 

```

这个算法求和

19

$$S = x_1 + x_2 + \dots + x_n$$

请注意该算法不能像前一个算法那样并行，它可以采用不同的并行方法，在 $O(\log n)$ 时间内完成。我们将在第3章给出完整的并行方法。操作级并行与此类似。考虑语句

$$Y \leftarrow a^t + b^t + c^t$$

在这个语句中，必须计算 a^t, b^t, c^t ，并把和存在 Y 里。对 a^t, b^t, c^t 的计算可以用三个不同处理器执行，这就是操作级并行。综合上述并行的层次，我们在下一节来看一下 Flynn 和 Handler 是如何对并行计算机进行分类的。

1.2 并行计算机分类

20世纪的最后30年出现了各种并行处理的新计算机体系结构，这些是对并行计算主要方法的补充和扩充。最近并行处理技术扩展包含了几种新的硬件体系结构。现在该领域面临的一个重大障碍是总结现有的并行计算体系结构，并定义它们之间的有序关系。我们先介绍最早但最流行的 Flynn 分类，然后介绍最近提出的对 Flynn 分类不足方面进行修正的 Handler (Erlangen) 分类以及其他一些分类。

1.2.1 Flynn分类

任何系统都包含两个重要的组成元素：

1. 指令；
2. 数据。

对数据元素的操作依据指令进行。根据执行的指令和同时操作的数据的数目，Flynn做出了如下分类。其中最简单的是通常的串行计算机，即每条指令一次只对一个数据集执行操作。Flynn称之为单指令单数据 (Single Instruction Single Data, SISD) 系统。图1-2所示为SISD模型。

单指令多数据 (Single Instruction Multiple Data, SIMD) 系统是指同一条指令同时对不同的数据集进行并行操作。这里的数据集个

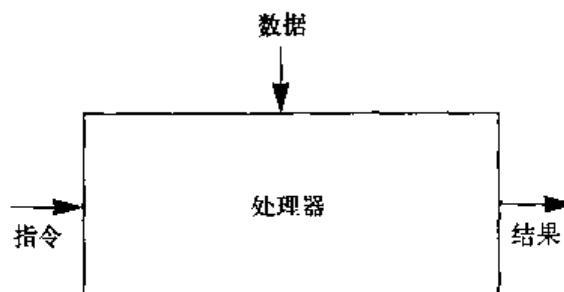
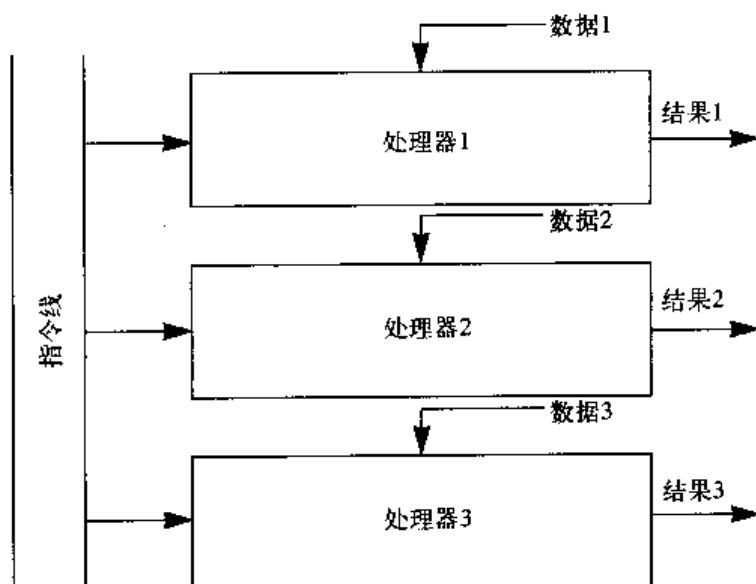


图1-2 SISD模型

数是同时进行操作的处理器的数量。图1-3所示为SIMD模型。比如，对一个数组的每个元素都加1：

For $i = 1$ to n do

$$x_i \leftarrow x_i + 1$$



21

图1-3 SIMD模型

上述计算可以直接并行进行。我们可以指定 n 个处理器 P_1, P_2, \dots, P_n 。如果这些处理器是SIMD体系结构，那么对每个处理器都给予共同的“对数据加1”的指令。每个处理器 P_i 会得到 x_i ，并对它加1。可以看出不同处理器的数据不同而指令相同。在SIMD模型中，有两种类型的体系结构：

1. 共享存储器模型；
2. 直接连接网络。

在共享存储器模型中有一个所有处理器共享的内存。两个处理器之间的通信是通过共享内存实现的。这在图1-4中给出。在直连网络模型中，独立的处理器通过网线连接，它们可以根据不同的拓扑结构（如环型、超立方体等）进行连接。我们在第1.3节讨论这个问题。

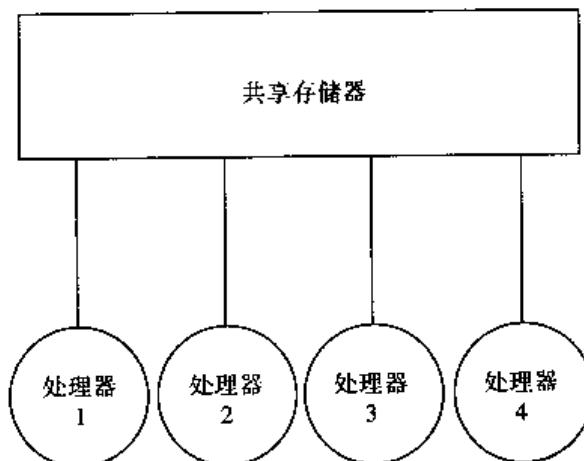


图1-4 共享存储器模型

Flynn也解释了多指令单数据（Multiple Instruction Single Data, MISD）系统，它能是对单个数据集执行多个不同操作的机器理论模型。目前为止，还没有设计出符合这种模型的计算机。图1-5展示该模型。

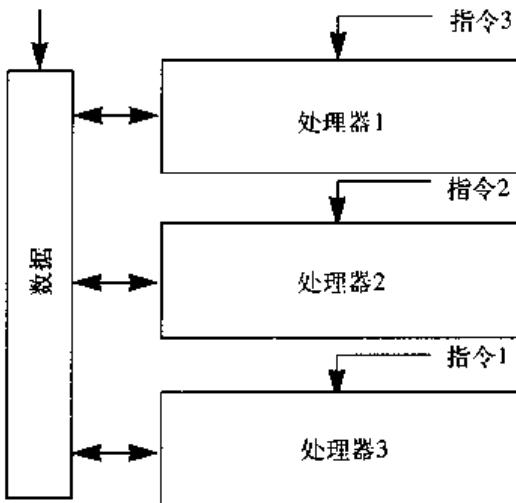


图1-5 MISD模型

多指令多数据（Multiple Instruction Multiple Data, MIMD）模型是指多处理器系统，它有多个处理器，可以独立运行并生成全局系统的结果。每个处理器都可以执行单独的指令对单独的数据集进行操作。

虽然Flynn的分类提供计算机体系结构的基本特征，但却不足以对当前的计算机进行有效分类。比如，流水线向量处理器组成的并行计算机，体系结构应包含在分类之列，因为它们能够执行大量的并发算术运算，且能并行操作成百上千的向量元素。但由于这些计算机的处理器不能像SIMD计算机一样通过单步同步执行同一指令，且没有MIMD模型的自主能力，我们不能把它归入任何一类Flynn分类中。由于Flynn分类的缺陷，已经有一些尝试想通过对Flynn分类进行扩展来囊括新的并行计算机体系结构。由于其所依据的原则与Flynn分类不同，有另外两种与Flynn分类截然不同的分类。但是这些分类通常缺乏Flynn分类所具有的内在简单性，因此不能像Flynn分类一样被广泛采用。在下面几节里，我们将给出这两种分类特点的简述。这两种分类分别是Erlangen分类（由Erlangen-Nürnberg的Friederich Alexander大学的Handler提出）和由Giloi提出的Giloi分类。

1.2.2 Erlangen分类 (Handler分类)

与Flynn的分类不同，Erlangen分类系统是基于对硬件的三层分类：

1. 程序控制单元 (Program Control Unit, PCU)；
2. 算术逻辑单元 (Arithmetic Logic Unit, ALU)；
3. 基本逻辑电路 (Elementary Logic Circuit, ELC)。

通常，一个计算机包含一个或多个PCU；每个PCU控制一个或多个ALU；一个ALU由和其数据通道位数一样多的多个ELC组成。 $w = (ELC/ALU)$ 是机器的字长度。在这个分类系统中，一个计算机 c 的最简描述是一个三元组： $t(c) = (k, d, w)$ ，其中 k 是PCU的个数， d 是 (ELC/PCU) 的值， w 是 (ELC/ALU) 的值。任何计算机都可以根据这三个参数分类：

例如：

$$\begin{aligned}t(\text{IBM 701}) &= (1, 1, 36); \\t(\text{Illiac IV}) &= (1, 64, 64); \\t(\text{c.mmp}) &= (16, 1, 16).\end{aligned}$$

1.2.3 Giloi分类

Gilioi分类模式依据形式文法：

$$G = \langle V_n, V_r, P, S \rangle$$

对体系结构进行分类，其中 V_n 是表示某些复杂的或者更高层的体系结构特征的非终结符号的集合； V_r 是表示基本的、未定义或“公理性的”体系结构特征的终结符号的集合； P 是表示由其他（基本的或复杂的）特征组合而成的复杂体系结构特征的结果的集合； S 表示计算机体系结构的起始符号。基于上述概念，Gilioi分类模式能够用一个语法分析树完成任何体系结构的分类。

除了Erlangen和Gilioi这两个与Flynn分类依据截然不同的体系之外，其他一些文献中提出的分类体系主要是对Flynn分类的扩展，以便把新的计算机体系结构包括进去。其中最有名的是Hwang-Brigg分类和Duncan分类。

1.2.4 Hwang-Brigg分类

Hwang-Brigg分类是通过对Flynn分类进行如下四项修改得到的：

1. 取消MISD模型；
2. 把原来的SISD模型精细化为两个子模型：只有一个功能部件的处理器(SISD-S)和有多个功能部件的处理器(SISD-M)；
3. 把原来的SIMD模型精细化成为两个子模型：进行字片(word-slice)处理的机器(SIMD-W)和进行位片(bit-slice)处理的机器(SIMD-B)；
4. 把原来的MIMD模型精细化成为两个子模型：处理器松耦合的机器(MIMD-L)和处理器紧耦合的机器(MIMD-T)。

Hwang-Brigg的改进增强原有模式的预测能力。在这种情况下，SISD-M具有比SISD-S更高的性能。然而，对SIMD-B和SIMD-W不能作出这样的一般性结论。我们用Hwang-Brigg分类方法对以下几种机器进行分类。

例子：依据其最大潜在并行度，下列机器根据性能的上升顺序排列：

STARAN	:	(SIMD-B)
Illiad IV	:	(SIMD-W)
PEPE	:	(SIMD-W)
MPP (Aerospace)	:	(SIMD-B)

因此，位片机器的性能可以比字片机器的性能好，也可以比它差。

1.2.5 Duncan分类

Duncan对Flynn分类提出了一些改进，以便包含流水线向量处理器和其他直观上被认为是并行计算机但又不能被原来的Flynn分类模式容纳的机器。它主要是通过对分类依据的简单扩

展，增加Flynn模式中没有的子类别，以反映体系结构特点的变化和覆盖底层的并行特点。图1-6给出基于Duncan改进的分类。主要有三类：

1. 同步体系结构；
2. MIMD；
3. MIMD变形。

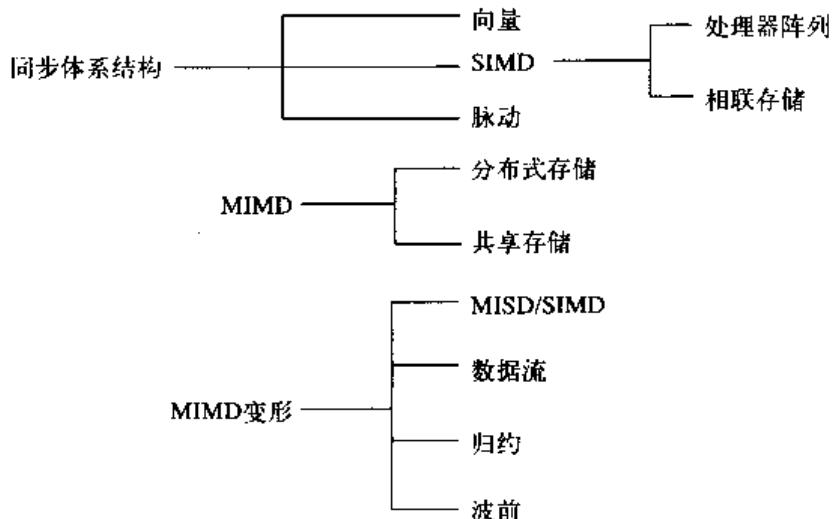


图1-6 Duncan对Flynn分类的改进

同步体系结构 同步体系结构通过全局时钟、中央控制部件或向量部件控制器在同步中协调并发操作。该类体系结构包括三个子类。它们是：

1. 向量处理器；
2. 脉动阵列；
3. SIMD体系结构。

SIMD体系结构更进一步分成处理器阵列和相联存储处理器。

向量处理器 向量处理器的特点是用多个流水线功能部件对向量操作数提供并行向量处理功能。有两类向量处理器：寄存器到寄存器向量处理器和内存到内存向量处理器。在寄存器到寄存器向量处理器中，操作数和向量操作的结果保存在专用的高速寄存器中；而在内存到内存向量处理器中，这些数据保存在专用的内存缓存区中。在向量长度比较大时，向量处理器会获得比较高的性能。最近的向量处理超级计算机（如Cray-XMP/4和ETA-10）由4到10个处理器共享一个大的内存。此类体系结构能够支持任务级的并行，虽然其设计基础是向量处理能力，但可以归为MIMD体系结构。向量处理器的例子如下：

- 内存到内存向量处理器：CDC Star 100, Cyber 203, Cyber 205, TI-ASC
- 寄存器到寄存器向量处理器：Cray-1, Cray-2, Cray-XMP/4, ETA-10, Fujitsu-200

脉动阵列 脉动阵列由流水线多处理器组成。数据以规律性的脉动方式流出内存，通过处理器网络，最后返回内存。全局时钟和显式的时间延迟对流水线数据流进行同步，此数据流由从内存取得的操作数和每个处理器使用的部分结果组成。脉动阵列计算机的例子是Warp和Saxpy的Matrix，它是由卡内基-梅隆大学设计的可重构脉动计算机。

SIMD处理器阵列 处理器阵列是为大规模科学计算（如图像处理和核能源建模）设计的。它们是在20世纪60年代后期开发的（如Illiad-IV），稍近的机器是Burroughs科学处理器

(Burroughs Scientific Processor, BSP)。这类机器采用能够进行字级长度操作的处理器。这些操作数通常是浮点(或复型)数据，其长度范围从32位到64位。处理器阵列的一个变形是由大量位处理器组成的处理器网格(如 64×64)。这类处理器阵列被称作大规模并行处理器(Massively Parallel Processor, MPP)，例如Loral的MPP、ICL DAP、FPS 164/MAX、Goodyear的MasPar Models (MPP)、Thinking Machine公司的Connection Machine和SX-2的SCS-40等。

相联存储处理器 此类处理器围绕相联存储而制造，且包含类型特别的SIMD体系结构，这一结构使用特殊的能根据存储内容并行访问存储数据的比较逻辑。搭建相联存储处理器的研究开始于20世纪50年代后期，其最明显的目的是能够在内存中并行搜索那些符合某些特定条件的数据。大多数当前的相联存储处理器采用位串行的组织方式，可以支持对相联存储中所有字的一个位片(位列)进行并发操作。这类机器的例子是贝尔实验室的Parallel Element Processing Ensemble (PEPE)和Loral的Associative Processor (Aspro)。

MIMD体系结构 MIMD体系结构采用可以使用本地数据执行独立指令流的多个处理器，因此MIMD计算机支持处理器大范围自治处理方式下的并行执行。虽然MIMD体系结构上的软件进程执行通过互联网消息传递或访问共享存储器中的数据进行同步，但它们仍是以分散型硬件控制为特征的异步计算机。因此，MIMD体系结构也多被称作多处理器。它可以分成两个子类型：

1. 共享存储(紧耦合)；
2. 分布式存储(松耦合)。

共享存储多处理器(紧耦合) 紧耦合MIMD体系结构在处理器之间共享存储器。互联网络结构分为两类：总线双向连接和直接连接。在总线双向连接结构中，处理器、并行存储器、网络接口和设备控制器都连接在同一总线上；而在直接连接结构中，处理器直接连接到高端大型机上。紧耦合多处理器的例子有：Univac 1100/94、Cary-XMP、Alliant/8和IBM 3090/400等。

分布式存储多处理器(松耦合) 松耦合MIMD体系结构由连接到多处理器节点的分布式本地存储器组成。比较流行的互联拓扑有超立方体、环、蝶形交换器、超树(HyperTree)和超网(Hypernet)(见第1.3节)。消息传递是处理器之间的主要通信方式。大多数多处理器在设计时就考虑到性能的可伸缩性。松耦合多处理器的例子有：DADO2、哥伦比亚大学的Non-Von、Cosmic Cube、Ametec 2010系列、Intel个人超级计算机(Intel Personal Supercomputer)、Ncube/10、Lawrence Snyder的可配置高度并行计算机(Configurable Highly Parallel Computer, CHIP)和Howard Siegel的可划分 SIMD/MIMD系统(Partitionable SIMD/MIMD System, Pasm)等。

基于MIMD体系结构的变形 MIMD/SIMD混合、数据流体系结构、归约机和波前阵列处理器等都给Flynn有条理的分类造成困难。例如，所有这些结构类型都有MIMD体系结构的异步执行和对多指令流及数据流并发操作的特征。然而，它们也有别于MIMD体系结构的设计原则。因此，这些体系结构都被归为“基于MIMD体系结构的变形”中，既突出它们与MIMD的不同基础特点，也说明它们与MIMD的共同特点。

MIMD/SIMD体系结构选择了部分MIMD体系结构并以SIMD的方式进行控制。可重构体系结构和控制SIMD执行的实现机制是很不相同的。比较流行的实现方式是采用树形结构的消息传递计算机作为基本结构。MIMD/SIMD机器的例子有DADO、Non-Van、Pasm和Texas可

重构阵列计算机 (Texas Reconfigurable Array Computer, TRAC)。

数据流体系结构的基本特点是一旦指令的所有操作数可用就可执行指令。因而指令的执行顺序基于数据相关性，从而使得数据流体系结构能够发掘任务、子程序和指令级别上的并发性。数据流体系结构的主要目的是开发新的计算模型和语言来实现大规模的并行性。一些著名的数据流计算机有曼彻斯特DataFlow Computer、MIT Tagged Token DataFlow architecture和LAU System。

28 归约（或需求驱动）机 这类机器实现的执行环境是当某个指令的计算结果被另外一个准备执行的指令用作操作数时，就执行该指令。归约机运行的程序由嵌套的表达式组成。表达式被递归定义为以文字表达式为参数的文字或函数。实现归约机要解决的问题包括指令执行结果的同步需求和维持表达式演化的结果。归约机的例子有Newcastle归约机、北卡罗林娜Cellular Tree Machine、犹他Applicative Multiprocessing System等。

波前处理器阵列 此类处理器阵列把脉动数据流水线和异步数据流执行语义结合在一起。S.Y.Kung于20世纪80年代早期提出了这一概念。这激发了脉动阵列的研究设计，可以为利用高I/O带宽平衡计算密集特殊应用系统提供高效和价廉物美的体系结构。波前和脉动阵列体系结构都具有模块化处理器和本地的常规互联网络的特点。但是，波前阵列用协调处理器间数据传递的异步握手机制取代了用在同步脉动数据流水线中的全局时钟和显式时间延迟。握手机制允许计算波前以阵列的处理器为波传播介质平滑地通过阵列而不相交。Kung认为波前阵列与脉动阵列相比具有更大的可扩展性、更简单的程序设计和更强的容错能力等特点。约翰霍普金斯大学、斯坦福Telecommunication、Royal Signal and Radar Establishment（英国）都建造了波前处理器阵列。

1.3 并行计算模型

任何并行算法的设计都是基于并行计算机体系结构假设基础上的，这就是并行计算模型。很多研究者提出了许多不同的设计并行算法的计算模型。本节介绍其中几个重要的。

1.3.1 二叉树模型

在这一模型中，整个问题的处理以二叉树的形式表示，其中每个非叶节点都有两个子节点。（二叉树的正式定义见第2章）每个非叶节点代表一个操作。在同一层的所有操作可以并行执行。二叉树模型有时也叫做有向无环图（Directed Acyclic Graph, DAG）模型。

29 假定我们想求8个数的和。我们必须画一个以这8个数为叶子的最小高度的二叉树。二叉树的内部节点代表两个子节点值的相加操作。图1-7说明了这一过程。假定有4个可用的处理器。现在可以调度这些处理器来完成内部节点所代表的操作。调度一般是函数SCH，它给每个内部节点指定一个有序对 (p, t) ，其中 p 代表处理器数目， t 代表操作发生的时间。图1-7中的内部节点已经被标上了有序的数字1, 2, 3, 4, 5, 6和7。

如下是一个调度函数：

调 度	含 义
SCH(1) = (1, 1)	在时间1处理器1执行
SCH(2) = (2, 1)	在时间1处理器2执行
SCH(3) = (3, 1)	在时间1处理器3执行

(续)

调度	含义
SCH(4) = (4, 1)	在时间1处理器4执行
SCH(5) = (1, 2)	在时间2处理器1执行
SCH(6) = (2, 2)	在时间2处理器2执行
SCH(7) = (1, 3)	在时间3处理器1执行

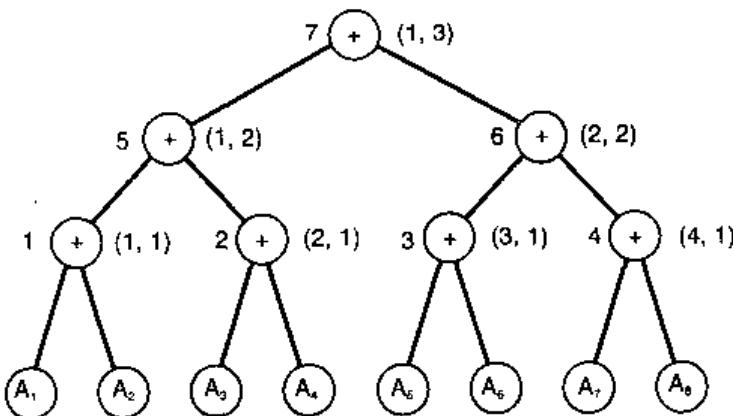


图1-7 相加算法的调度

这里8个数相加的任务已经被4个处理器在3个单位时间内完成了。有 n 个节点的完全二叉树的高度是 $\log n$ ，因此用 $n/2$ 个处理器相加 n 个数的任务可以在 $\log n$ 个单位时间内完成。

这意味着在 $t=0$ 到 $t=1$ 的时间间隔内，下列任务可以同时执行：

处理器1执行 $A_1 + A_2$ ；

处理器2执行 $A_3 + A_4$ ；

处理器3执行 $A_5 + A_6$ ；

处理器4执行 $A_7 + A_8$ 。

现在我们获得了 $(A_1 + A_2)$, $(A_3 + A_4)$, $(A_5 + A_6)$ 和 $(A_7 + A_8)$ 的值。而在 $t=1$ 到 $t=2$ 的时间间隔内，下列任务可以同时执行：

处理器1执行 $(A_1 + A_2) + (A_3 + A_4)$ ；

处理器2执行 $(A_5 + A_6) + (A_7 + A_8)$ 。

这样我们就得到了 $(A_1 + A_2 + A_3 + A_4)$ 和 $(A_5 + A_6 + A_7 + A_8)$ 的值。在 $t=2$ 到 $t=3$ 的时间间隔内，处理器1完成 $(A_1 + A_2 + A_3 + A_4)$ 和 $(A_5 + A_6 + A_7 + A_8)$ 相加的任务。整个任务按照下式表示的过程执行：

$$\{(A_1 + A_2) + (A_3 + A_4)\} + \{(A_5 + A_6) + (A_7 + A_8)\}$$

假定只有1个处理器可用。在这种情况下，整个计算过程可以用图1-8中给出的斜树表示。相应的调度函数值也在每个内部节点旁边给出。在此情况下需要7个单位时间。而要把 n 个数相加需要 $n-1$ 个单位时间。

当只有3个处理器时，对8个数相加的二叉树模型如图1-9所示。这时需要4个单位时间。

由于定义调度函数很烦琐，在并行计算中二叉树模型并不常用。

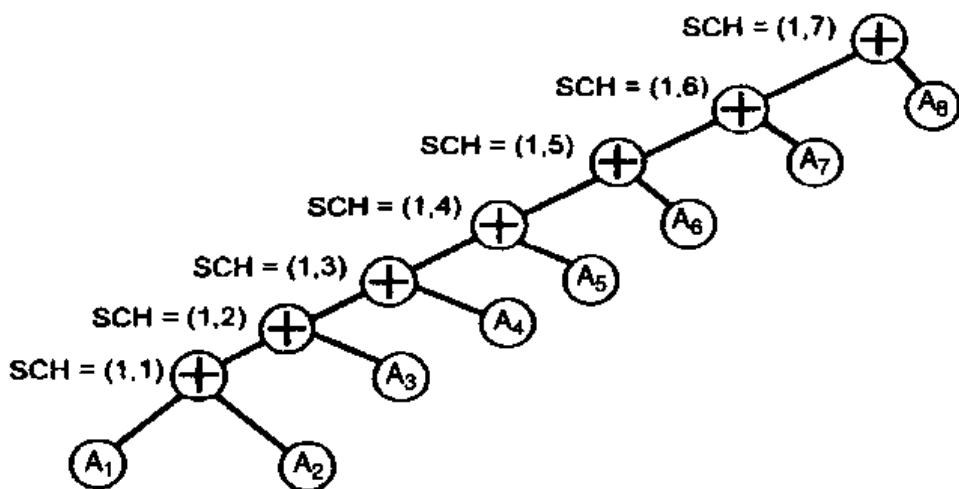


图1-8 串行计算的二叉树模型

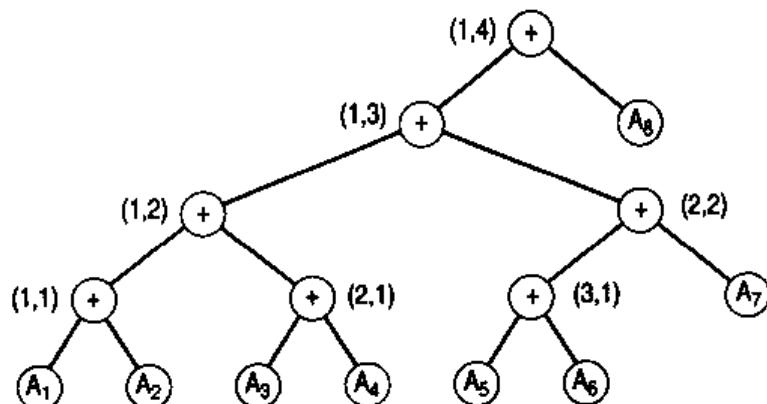


图1-9 用3个处理器求和

1.3.2 网络模型

当需要并行化时，很自然要用到多个处理器。我们假定这些处理器独立工作并能相互通信。为了通信，这些处理器必须用物理的链路相连接。这类模型叫做网络模型或直连机器。在网络模型中，几个处理器通过物理链路互相连接，并有如下六个假设：

1. 每个处理器都有一个相联大存储器，即处理器的本地存储器。处理器的程序以及输入输出值都存放在存储器中；
2. 不存在所有处理器都能访问的公用存储器；
3. 处理器之间直接用物理链路连接，互连拓扑称做网络拓扑。网络拓扑是一个以处理器为顶点，以互连为边的图。链路可以是单向的或者双向的；
4. 如果两个处理器相邻，那么数据可以直接从一个处理器传送到另外一个；
5. 在一个时钟周期内，任一个处理器执行一步单位操作；
6. 处理器可以在单位时间内把数据传递到它相邻的处理器中。

当为一个网络模型系统设计算法时，必须首先指定如下内容：

1. 网络拓扑；
2. 输入配置；
3. 输出配置。

网络拓扑 遗憾的是，不存在对所有问题都合适的理想通用拓扑。因此我们必须调查问题，然后提出互连拓扑。它可以是任意的图、环、树、立方体等。

输入配置 针对特定网络拓扑而设计的求解特定问题的算法中，输入必须在各处理器进行分布。我们称此为输入配置。

输出配置 输出结果在各处理器的分布，称为输出配置。

下面将研究用于求解一些有趣问题的网络拓扑。我们从最流行的拓扑——超立方体开始。

1.3.3 超立方体 (k -立方体)

首先用递归的方式来定义 k -立方体。0-立方体表示单个处理器。1-立方体表示两个相互连接的处理器。2-立方体表示4个处理器为顶点组成的正方形处理器网络。2-立方体可以通过两个1-立方体构建，即把 P_0-P_1 和 P_2-P_3 两个1-立方体的 P_0 和 P_2 ， P_1 和 P_3 连接。如图1-10所示。一个3-立方体由一对2-立方体组成，

$$P_0P_1P_2P_3 \text{ 和 } P_4P_5P_6P_7$$

而且需要在如下节点间建立连接：

P_0 到 P_4

P_1 到 P_5

P_2 到 P_6

P_3 到 P_7



图1-10 2-立方体

图1-11画出了一个3-立方体（目前可以忽略节点上的标识）。通常， k -立方体可以定义为一对 $(k-1)$ -立方体通过在相应的节点建立连接而形成。 k -立方体也可以称为 k 维超立方体。

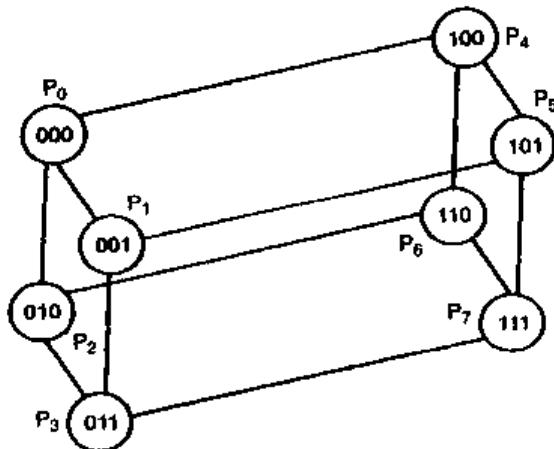


图1-11 3-立方体

以上对 k -立方体的定义就是递归方式的。一个 k -立方体包含 2^k 个节点，且用 k 位的二进制数标识。两个节点的二进制标识当且仅当有1位不同时，这两个节点用一条边相连接。例如，3-立方体包含8个节点，且分别标识为000、001、010、011、100、101、110、111。一般，任一节点 $(b_1b_2b_3\cdots b_k)$ 通过边与下述 k 个节点连接（其中的 \bar{b}_i 表示对该位取反， $i = 1, 2, \dots, k$ ）：

$$\begin{aligned} & \bar{b}_1b_2b_3\cdots b_k \\ & b_1\bar{b}_2b_3\cdots b_k \\ & \cdots \\ & b_1b_2\bar{b}_3\cdots \bar{b}_k \end{aligned}$$

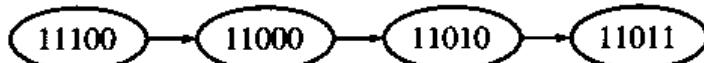
例如，在3-立方体里，000与100、010和001连接；001与101、011和000连接；010与110、000和011连接；100与000、110和101连接；101与001、111和100连接；110与010、100和111连接；111与011、101和110连接。这些都在图1-11中给出。从 k -立方体的定义来看，我们可以观察到如下的属性：

属性1 k -立方体就是有 2^k 个节点的 k 正则图。 k 正则图是每个节点都有 k 度的图。

[34]

属性2 两个节点 $a = (a_1a_2\cdots a_k)$ 和 $b = (b_1b_2\cdots b_k)$ 之间的距离是 a 和 b 之间不同的二进制位个数。

例如，在5-立方体里，(11100)和(11011)之间的距离是3。这是因为它们有3个二进制位不同。(11100)与(11000)相邻，而(11000)与(11010)相邻，(11010)又与(11011)相邻。



由于数据可以在单位时间内从一个节点传送到相邻节点，那么从节点(11100)到(11011)的传送时间为3个单位。

在一个6-立方体中，假定我们要从节点 P_{27} 传送数据到 P_{43} ，并有 $(27)_{10} = (011011)_2$ 和 $(43)_{10} = (101011)_2$ 。

这两个节点的标识有两位不同。所以，在两个单位时间内我们可以把数据从 P_{27} 传送到 P_{43} 。传送的路径通过匹配相应的二进制位确定：

$$\begin{array}{ccc} P_{27} & P_{59} & P_{43} \\ (011011) & (111011) & (101011) \end{array}$$

属性3 在一个具有 n 个节点的超立方体（其中 $n = 2^k$ ）中，数据最多在 $\log_2 n$ 个单位时间内从一节点传送到另一节点。这是由于属性2和 $n = 2^k$ 隐含着 $\log_2 n = k$ 。本书里，由于只处理底数为2的对数，我们简单地把 $\log_2 n$ 表示为 $\log n$ 。

属性4 设 $e(k)$ 为 k -立方体的边数。通过 k -立方体的递归搭建，我们有如下的递归方程：

$$e(k) = 2e(k-1) + 2^{k-1}$$

递归带入 $e(k-1)$ ，就有：

$$\begin{aligned} e(k) &= 2(2e(k-2) + 2^{k-2}) + 2^{k-1} \\ &= 2^2 e(k-2) + 2 \cdot 2^{k-1} \\ &= 2^3 e(k-3) + 3 \cdot 2^{k-1} \\ &= 2^{k-1} e(1) + (k-1)2^{k-1} \\ &= 2^{k-1} + (k-1)2^{k-1} \quad (e(1) = 1) \\ &= k2^{k-1} \end{aligned}$$

[35]

如果 $n = 2^t$ 是 k -立方体的节点个数，就有：

$$e(k) = k2^{k-1} = (\log n)n/2$$

这就是 k -立方体的边的个数。

当 n 个节点连接成完全图时，有 $n(n-1)/2$ 条边。在完全图里，任两个节点都是相邻的，从一节点到另一节点传送数据都在一个单位时间内。但这需要 $n(n-1)/2$ 条边才能实现，而这么多条物理链路的开销是很大的。但是如果连接成 k -立方体，只需要 $(n/2)\log n$ 条边，从任一节点到其他节点数据传送需要 $\log n$ 个单位时间。表 1-1 给出了对不同的 n , $n(n-1)/2$ 和 $(n/2)\log n$ 相应的值。

表 1-1 k -立方体和完全图的比较

端点个数 n	$\log n$	完全图的边数 $n(n-1)/2$	k -立方体的边数 $(n/2)\log n$
2	1	1	1
4	2	6	4
8	3	28	12
16	4	120	32
32	5	496	80
64	6	2016	192
128	7	8128	448
1024	10	523776	5120

表 1-1 也表明了 k -立方体拓扑与完全图相比较下的经济优势。下面我们解释如何为直连网络中的 k -立方体拓扑设计算法。现在设计一个算法从有 n 个元素的数组中找出最小的元素。

假定该数组表示为：

$$a_0 a_1 a_2 \cdots a_{n-1}$$

下面给出简单的串行算法：

```

BEGIN
    small = an
    For i = 1 to n-1 do
        If small > ai then
            small = ai
        endif
    edfor
END

```

36

此串行算法需要 $O(n)$ 的时间。我们采用 k -立方体连接网络的处理器来设计一个时间复杂度为 $O(\log n)$ 的算法，从输入配置开始：

输入配置 把 n 个数分配给 n 个处理器。为了方便，假定 $n = 2^t$ 。开始时 a_i 位于处理器 P_i ($0 \leq i \leq n-1$) 的本地内存中。这在图 1-12 中给出。

输出配置 要求最终的结果（数组的最小元素）存储在 P_0 中。

执行过程 充分利用 k -立方体是一对 $(k-1)$ -立方体的定义，把 P_i 与 P_{i+2} 相连接。现在传送 P_{i+2} 处理器中的内容到 P_i ($0 \leq i \leq 2^{t-1}$)。处理器 P_0 中存有 a_0 和 a_2 。相似的， P_1 中存有 a_1 和 a_3 。每个处理器 P_i 都比较 a_i 和 a_{i+2} ，并把较小值存在 a_i 中。在此过程之后，最小的数存在下面数组之中：

$$a_0 a_1 a_2 \cdots a_{(n/2)-1}$$

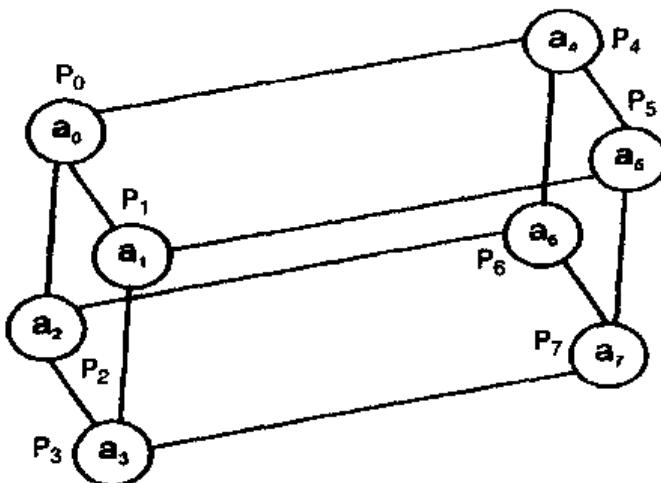


图1-12 输入配置

在由处理器 $P_0P_1\cdots P_{(n^2)-1}$ 组成的 $(k-1)$ -立方体上重复此过程，最小的数存储在数组中。

37

重复过程 k 次，最小的元素到达节点 P_0 。以包含15, 10, 20, 9, 8, 7, 21和5这8个元素的数组为例，此过程在图1-13a到图1-13g中给出。

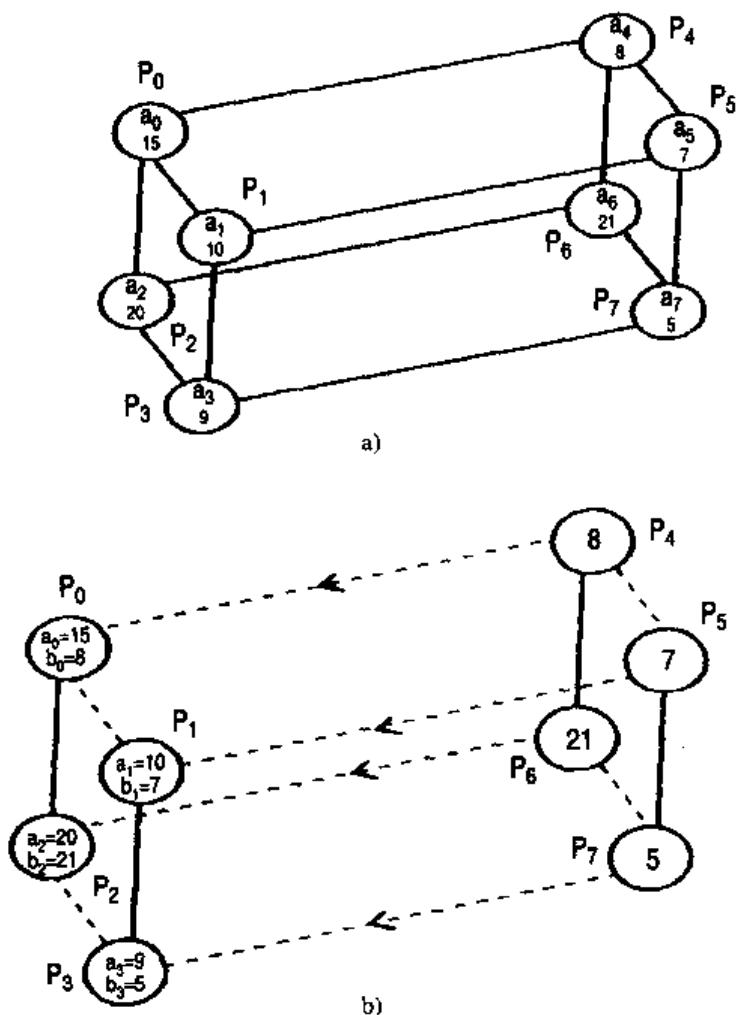


图 1-13

a) 数组元素的初始位置 b) P_i 从处理器 $P_{i+4}(0 \leq i \leq 3)$ 接收数组元素 a_{i+4} ，并存为 b_i

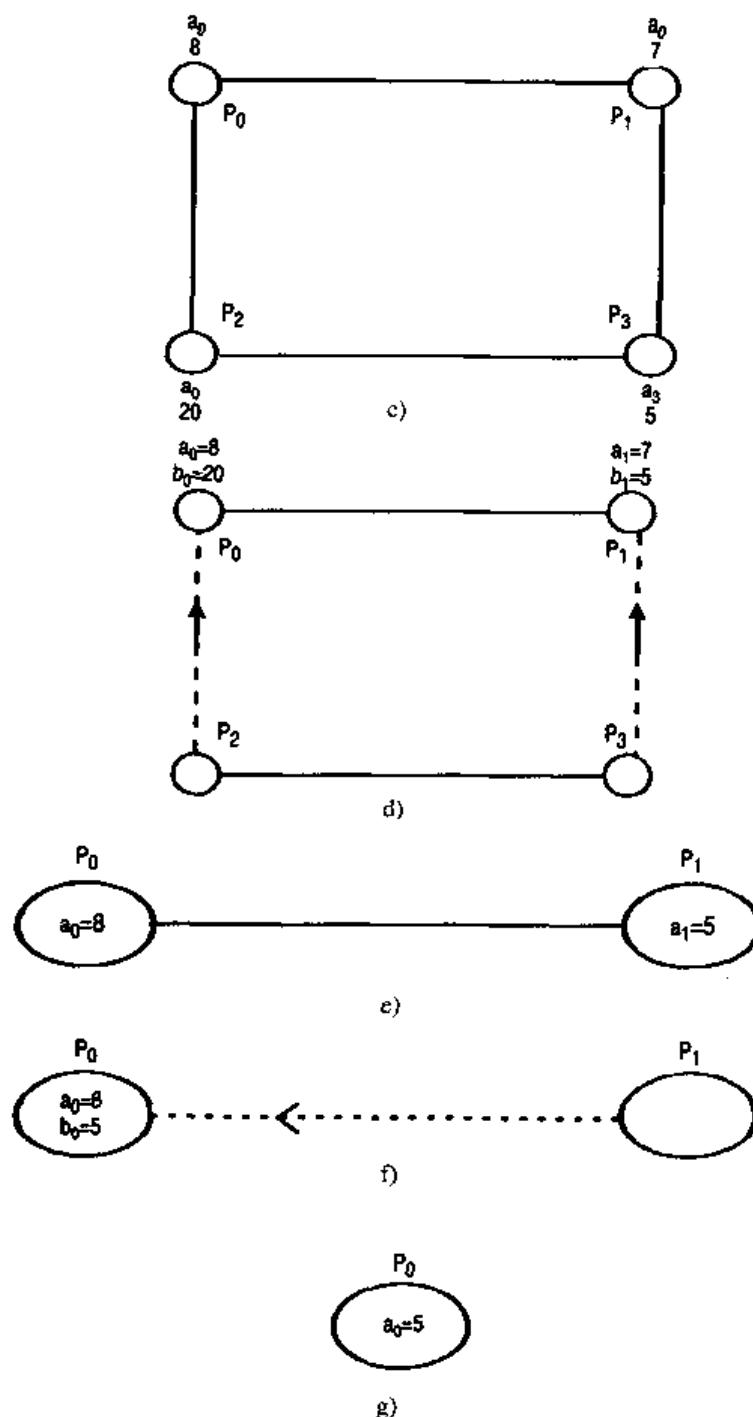


图1-13 (续)

- c) 对每个处理器 P_i ($0 \leq i \leq 3$)，执行操作 $a_i = \text{Min}\{a_i, b_i\}$ 。此图表示了在进行操作之后的 a_0, a_1, a_2, a_3 的值 d) 处理器 P_i ($0 \leq i \leq 1$) 从处理器 P_{i+2} 接收数组元素 a_{i+2}
e) 对 $i = 0, 1$ ，在执行 $a_i = \text{Min}\{a_i, b_i\}$ 操作之后处理器上的内容
f) 处理器 P_1 发送 a_1 到 P_0 g) $a_0 = \text{Min}\{a_0, b_0\}$

上述算法可以形式化描述如下：

算法k-CUBE-Min

输入：每个处理器 P_i 都在本地内存中包含有数组元素 a_i ($0 \leq i \leq n-1$)。假定 $n = 2^k$
输出：数组 a_0, a_1, \dots, a_{n-1} 的最小元素存入变量 a_0

39

```

BEGIN
  1. FOR  $d = k-1$  to 1 step -1 do
  2.  $m = 2^d$ 
  3. For  $i = 0$  to  $m-1$  do in parallel
  4. Processor  $P_{i+m}$ : Send data  $a_{i+m}$  to processor  $P_i$ 
  5. Processor  $P_i$ : Receive data  $a_{i+m}$  sent by processor  $P_{i+m}$  and store in the local memory as  $b_i$ 
  6.  $a_i = \text{Min} \{a_i, b_i\}$ 
  7. end parallel
  8. end for
END

```

由于存在for循环（第1~8步），此算法的时间复杂度是 $O(k)$ ，即 $O(\log n)$ 。在上述过程中，第4步和第5步描述从一个处理器到另外一个的通信。通信的时间与处理操作的时间相比通常很长。因此，上述算法不会有很高的效率。

接下来将研究并行计算中用到的更多的网络拓扑。虽然网络模型不是本书关心的主要内容，但为了完整性，我们给出这些内容。

1.3.4 网格网络

处理器以网格的形式排列称做网格网络。图1-14给出了一个二维网格。三维网格由二维网格的集合组成，其中相应的处理器相连接。相似地，任意 k -维网格都是 $(k-1)$ -维网格的集合且相应的处理器能够相互通信。

40

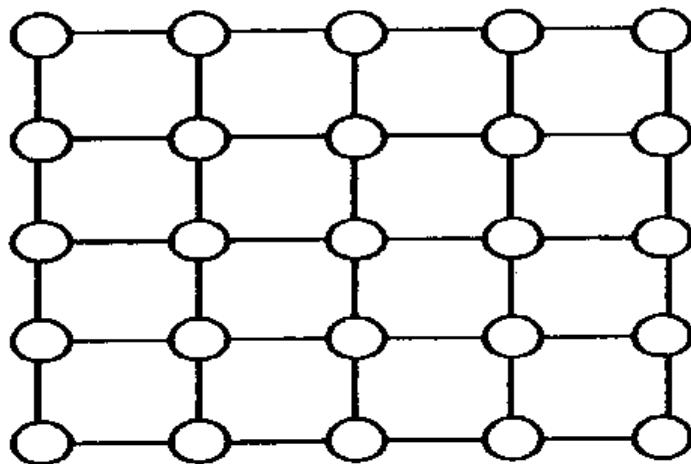


图1-14 二维网格

1.3.5 金字塔网络

金字塔网络与有根树的构建方式相似。网络的根包含1个处理器。在其第2层有4个处理器以二维网格的形式排列，且这4个都是根处理器的子处理器。第1层的4个节点每个都在第2层有4个子节点。所有在第2层的节点都以网格的形式排列。相似地，金字塔网络可以构建到任意高度。图1-15给出的是高度为2的金字塔网络。

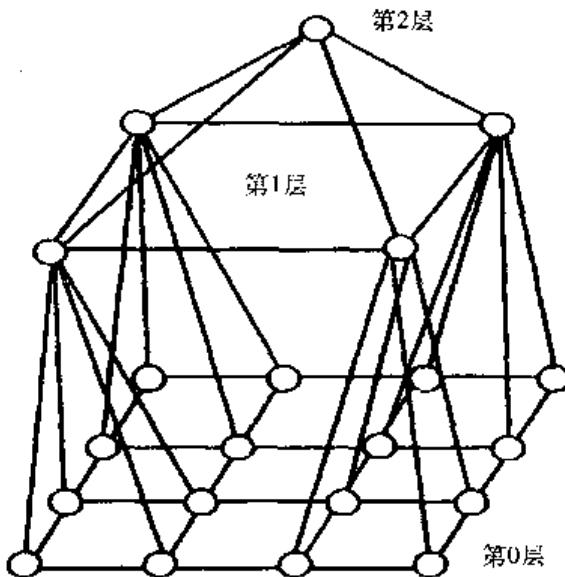


图1-15 大小为16的金字塔网络

1.3.6 星形图

k -立方体被定义为标识成0到 2^{k-1} （用二进制数表示）的 2^k 个节点的集合，当且仅当两个节点的二进制表示有1位不同时，二者用一条边相连。图1-11给出了3-立方体的二进制标识。上述定义启发研究者对星形图作如下定义：假定 k 是正整数，考察 k 个符号的置换。对 k 个符号，有 k 个置换。相应于 k 个置换可以定义 k 个节点。当且仅当两节点相应排列最左位和任意一个其他位置不同时，这两个节点相邻。由此得到的图被称做 k -星形图。以 $k=3$ 为例。在这种情况下，有六个置换：

$$\begin{array}{lll} P_0 = (1, 2, 3) & P_1 = (1, 3, 2) & P_2 = (2, 1, 3) \\ P_3 = (2, 3, 1) & P_4 = (3, 1, 2) & P_5 = (3, 2, 1) \end{array}$$

可简写为：

$$\begin{matrix} P_0 = 1, 2, 3 \\ 1, 2, 3 \end{matrix}$$

即 $P_0(1) = 1$, $P_0(2) = 2$ 和 $P_0(3) = 3$ 。

$$\begin{matrix} P_1 = 1, 2, 3 \\ 1, 3, 2 \end{matrix}$$

即 $P_1(1) = 1$, $P_1(2) = 3$ 和 $P_1(3) = 2$ 。

相应的3-星形图在图1-16给出。星形图也可以用递归的方式定义为：为了构建 k -星形图，考虑有 k 份 $(k-1)$ -星形图。在任一个 $(k-1)$ -星形图中，每个顶点都用 $k-1$ 个符号的置换标识。我们用 S_1, S_2, \dots, S_k 表示这 k 份 $(k-1)$ -星形图，用除 i 之外1到 k 的符号的置换标识 S_i ($1 < i < k$)的顶点。即 S_i 的每个顶点用符号 $2, 3, 4, \dots, k$ 的置换标识； S_i 的顶点用符号 $1, 3, 4, \dots, k$ 的置换标识。相似地，其他图也用这类方式标识。然后在图 S_i 每个顶点置换的最右边加入符号 i ，并生成如下的新边：当且仅当一个置换可以通过交换另外一个置换的第一个和最后一个符号得到时，这两个顶点 A 和 B 相连。Day和Tripathi(1991)已经研究了 k -立方体和 k -星形图的拓扑性质。下表给出了这两种拓扑的基本参数。

41

42

性质	k -立方体	k -星形图
顶点数	$2k$	k
度	k	$k-1$
直径	k	$3/2(k-1)$

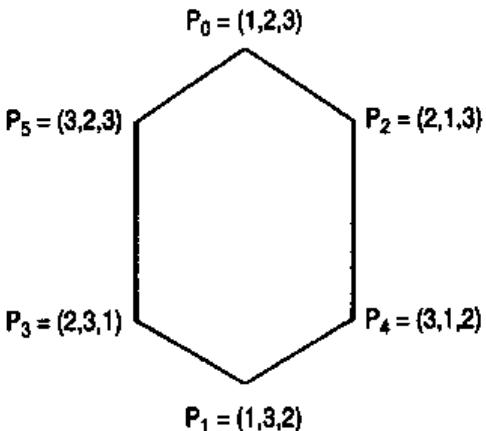


图1-16 3-星形图

Day和Tripathi(1992)提出了另外一种互连拓扑称做排列图 (Arrangement Graph)，是星形图的推广。他们证明了排列图在选择如度、直径和顶点数等主要参数时比星形图更具有灵活性。任何一个星形图都是排列图。Day和Tripathi(1993) 提出了对 k -星形图的边指定方向的方法，并推出了 k -星形图的几个有趣的性质。

1.4 PRAM模型

在直连网络里，处理器之间是通过物理链路直接连接的。而在PRAM (Parallel Random Access Machine) 模型里，所有的处理器都与全局大内存连接并且共享此内存。如图1-17所示，这一模型也称作共享存储器模型。假定所有处理器通过公用时钟同步运行。每个处理器都能访问（读/写）整个内存。处理器间的通信仅通过共享内存实现，即从处理器 P_i 发出的数据通过如下步骤到达 P_j ：

1. 处理器 P_i 把数据写入全局内存中；
2. 处理器 P_j 把数据从全局内存中读出。

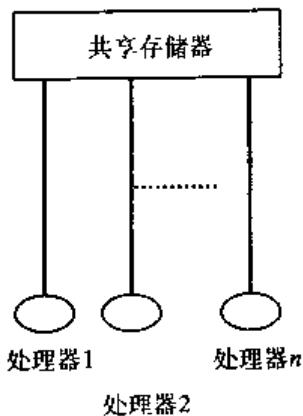


图1-17 PRAM模型

如图1-18所示。本书中的PRAM模型都以Flynn分类中的SIMD模型方式工作。但是有公用内存的SISD和MIMD机器也不能排除。在PRAM模型中，根据多个处理器对一个存储单元读/写数据的能力，可以有四种不同类型：

1. 绝对读绝对写PRAM (EREW);
2. 并发读绝对写PRAM (CREW);
3. 绝对读并发写PRAM (ERCW);
4. 并发读并发写PRAM (CRCW)。

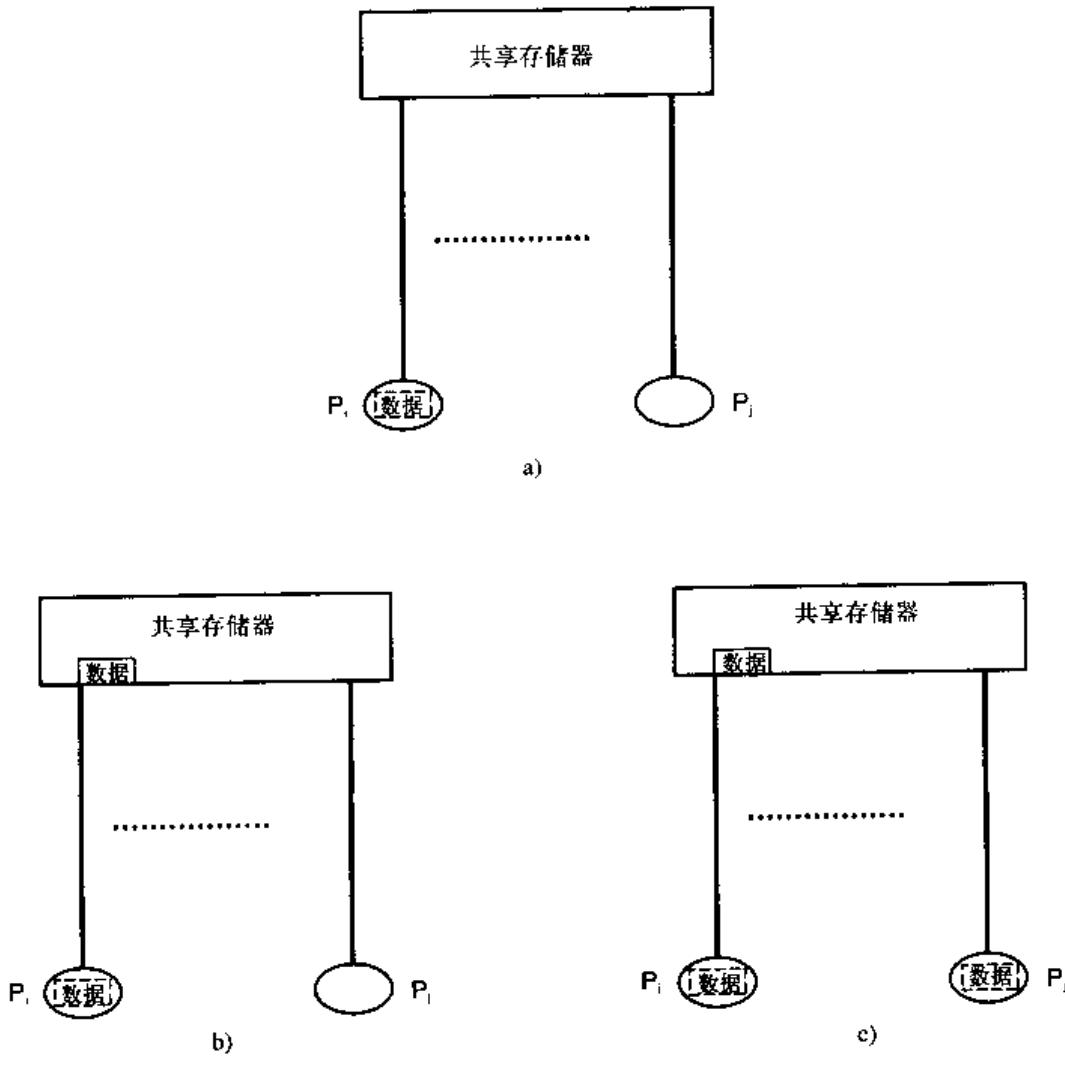


图 1-18

a) 处理器 P_1 b) 处理器 P_1 把数据写入全局内存

c) 处理器 P_1 把数据从全局内存中读出

绝对读绝对写PRAM模型同一时刻只允许一个处理器对同一存储单元读/写，而不允许多个处理器对同一个存储单元同时读/写。CREW PRAM模型允许多个处理器对同一存储单元并发读但不允许并发写。ERCW PRAM模型只允许对同一存储单元并发写。CRCW PRAM模型是功能最强的模型，允许对同一存储单元并发读和并发写。当一个或多个处理器想对同一存储单元并发读的时候，我们假定所有的处理器都会成功。然而，当多个处理器想对同一存储单元并发写的时候，必须对由此引起的冲突进行妥善处理。目前主要有三种解决冲突的方法：

1. 平等冲突解决 (Equality Conflict Resolution, ECR);
2. 优先权冲突解决 (Priority Conflict Resolution, PCR);
3. 任意冲突解决 (Arbitrary Conflict Resolution, ACR)。

在ECR的情况下，仅当所有的处理器对同一存储单元写的内容一样时，我们假定所有的处理器写操作成功。在PCR的情况下，假定每个处理器都有其优先权。当多个处理器同时对一个存储单元进行写操作时，具有最高优先权的处理器成功。在ACR的情况下，假定在并发写的所有处理器中，任意一个处理器都可能成功。

MIMD机器中的每个处理器都有自己的时钟且各处理器异步运行。每个处理器都是拥有自己本地内存的独立计算机，并能访问全局内存。每个处理器称作一个处理单元。如果处理单元通过中心转换机制连接到全局内存，处理器之间的通信就不会占用太多的时间且不复杂。此类MIMD系统称为紧耦合MIMD模型。反之就称为松耦合。

至此所提到的并行体系结构有其内在的缺陷，关注特定的体系结构并不能深入了解算法的逻辑结构。这样把算法移植到不同的体系结构会变得困难。另外，异步并行的计算模型实质上很抽象，这也使得对算法的分析很困难。因此，我们只限于讨论同步PRAM模型。在介绍了本书所有算法所采用的PRAM并行计算模型之后，让我们来介绍在本书中用来写并行算法的语言结构。

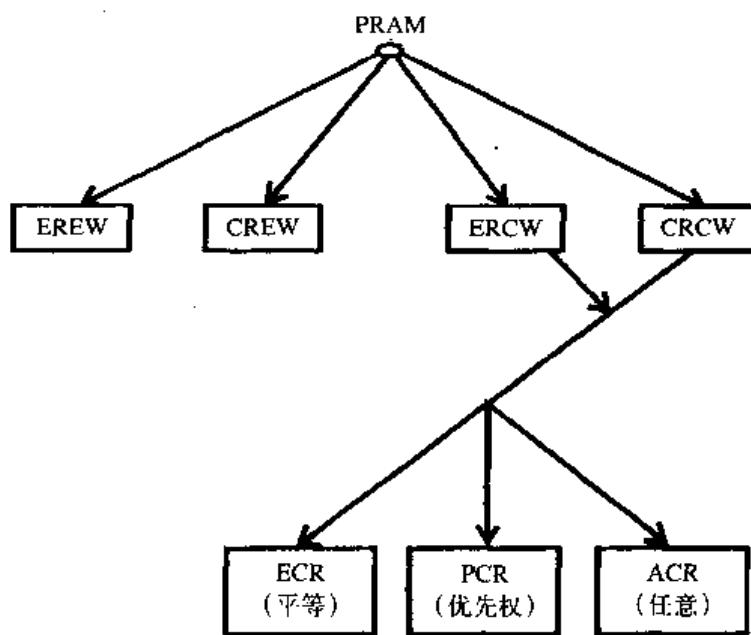


图1-19 PRAM的模型

并行算法的语言结构

我们现在定义表示算法的特定符号和语法。赋值语句的格式是：

Variable = expression

上式对表达式进行计算并把结果存入变量中。IF语句有如下格式：

If cond then

s₁

```

 $s_2$ 
⋮
Else
 $s'_1$ 
 $s'_2$ 
⋮
Endif

```

这里的语法 $cond$ 是有效逻辑条件。如果条件为真，则执行 s_1, s_2, \dots ，否则执行 s'_1, s'_2, \dots 。

For循环有如下的格式：

```

For variable = s to e step h
 $s_1$ 
 $s_2$ 
⋮
end for

```

这里通过设置变量集合的值为 $s, s+h, s+2h, \dots$ ，重复执行语句 s_1, s_2, \dots ，直到 $s+kh>e$ 。46

```

While cond do
 $s_1$ 
 $s_2$ 
⋮
end while

```

只要条件 $cond$ 为真，就不断重复执行语句 s_1, s_2, \dots 。用For-in-Parallel语句表示并行执行。该语句可以写成如下两种不同的结构：

结构1：

```

For variable = 1 to n do in parallel
 $s_1$ 
 $s_2$ 
⋮
End parallel

```

n 个处理器中的每一个同时执行相同的指令 s_1, s_2, \dots 。For语句的运行变量表示处理器的下标。

结构2：

```

For  $x \in S$  do in parallel
 $s_1$ 
 $s_2$ 
⋮
End parallel

```

同样地，每个处理器同时执行相同的指令 s_1, s_2, \dots 。这里，同时运行的处理器个数是 S 中元素

的个数。

假定 x, y, z 是存在全局内存中的三个变量，把

$$x = y + z$$

写成并行算法。每个处理器执行如下操作：

1. 读取变量 y 的内容，称其为 v_1 (v_1 是在处理器本地内存中的变量)；
2. 读取变量 z 的内容，称其为 v_2 (v_2 是在处理器本地内存中的变量)；
3. $v_3 = v_1 + v_2$ (v_3 是在处理器本地内存中的变量)；
4. 把 v_3 的值写入全局变量 x 。

以此为基础，我们给出一些简单的并行算法来说明并发读和并发写操作。

1.5 一些简单算法

考虑求取存储在数组 $A(1:n)$ 中的 n 个值的布尔与 (AND) 的值的问题。我们想要计算：

$$\text{RESULT} = A(1) \wedge A(2) \wedge A(3) \wedge \dots \wedge A(n)$$

一个 $O(n)$ 时间的串行算法如下：

```
BEGIN
    RESULT = .TRUE.
    For i = 1 to n do
        RESULT = RESULT  $\wedge$  A(i)
    End For
END
```

假定采用任意冲突解决 (ACR) 的ERCW PRAM模型。 $A(1:n)$ 是一个布尔值的数组，我们要计算：

$\text{RESULT} = \text{.FALSE.}$, 当任意一个值 $A(i)$ 为 .FALSE. 。否则,

$= \text{.TRUE.}$

假定 RESULT 作为全局变量初始时为 .TRUE. 值。我们想要占用 n 个处理器，且处理器 P_i 从全局内存中读取 $A(i)$ 的值，并检查其是否为 .FALSE. 。如果 $A(i) = \text{.FALSE.}$ ，处理器 P_i 立即在全局变量 RESULT 中写入 .FALSE. 。由于 n 个处理器并行运行，因此任务可以在单位时间内完成。如果有多个 $A(i)$ 值为 .FALSE. ，则有多个处理器并发地对全局变量 RESULT 写入值 .FALSE. 。这可以通过任意冲突解决 (ACR) 来处理。如果所有的 $A(i)$ 值都为 .TRUE. ，那就没有处理器想要对 RESULT 写入，其初始值 .TRUE. 仍然保持。

我们正式给出运行在采用ACR的ERCR PRAM模型里的并行算法：

算法 Boolean-AND

输入：布尔数组 $A(1:n)$

输出：布尔值 RESULT

```
BEGIN
    RESULT = TRUE
    For i = 1 to n do in parallel If A(i) = .FALSE.
    then      RESULT = .FALSE. endif
    END
```

在上述算法中， $O(n)$ 处理器在 $O(1)$ 时间内完成所有任务。可以看出，采用三种并发写冲突解决类型（ECR、PCR或ACR）的任何一个，上述算法都可以得到正确结果。

在ERCW-ECR模型中，对同一问题考虑如下算法：

算法 Boolean-AND-1

输入：布尔数组 $A(1:n)$

输出：布尔值RESULT

BEGIN

 RESULT = .FALSE.

 For $i = 1$ to n do in parallel

 RESULT = $A(i)$

 End Parallel

END

这里 n 个处理器中的每一个都试图把 $A(i)$ 的值写入RESULT中。如果冲突解决模型是ERCW-ACR类型，将不能得到正确结果。仅当PRAM模型是ERCW-ECR类型时上述算法将正确给出RESULT值。我们接下来设计能在EREW PRAM模型中处理同一问题的算法。

基本“与”操作是二元操作，因此，一个处理器一次只能对两个数执行“与”操作。当数据个数为 n 的时候，需要 $n/2$ 个处理器。这 $n/2$ 个处理器每个从 n 个数据分得2个数据并同时执行“与”操作。此时，得到的 $n/2$ 个结果作为数据重复上述过程，得到 $n/4$ 个结果。按上述方式重复，在 $O(\log n)$ 时间内将得到最后结果。例如，如果有8个数 $A(1:8)$ ，第一步执行如下操作：

处理器 P_1 ： $A(1) \leftarrow A(1)^A A(2)$ 处理器 P_2 ： $A(2) \leftarrow A(3)^A A(4)$

处理器 P_3 ： $A(3) \leftarrow A(5)^A A(6)$ 处理器 P_4 ： $A(4) \leftarrow A(7)^A A(8)$

一般地，处理器 P_i 执行 $A(i) \leftarrow A(2i-1)^A A(2i)$ 。这样就可以通过计算 $A(1)^A A(2)^A A(3)^A A(4)$ 得到最后结果。两个处理器也可以用同样的方法来计算。处理器 P_1 执行 $A(1)^A A(2)$ 并存入 $A(1)$ ；处理器 P_2 执行 $A(3)^A A(4)$ 并存入 $A(2)$ 。这样可以通过计算 $A(1)^A A(2)$ 得到最后结果。现在正式给出这一算法：

算法 Boolean-AND-2

输入：布尔值 $A(1:n)$

输出：RESULT

$p = \text{number of processors used.}$

BEGIN

$p = n/2$

 While $p > 0$ do

 For $i = 1$ to p do in parallel

$A(i) = A(2i-1)^A A(2i)$

 End Parallel

$p = [p/2]$

 End while

 RESULT

END

当 $n=8$ 时，此算法的执行过程如图1-20所示。其中的while循环需要重复执行 $\log n$ 次，因此此算法的时间复杂度为 $O(\log n)$ 。在EREW PRAM模型中需要 $n/2$ 个处理器。

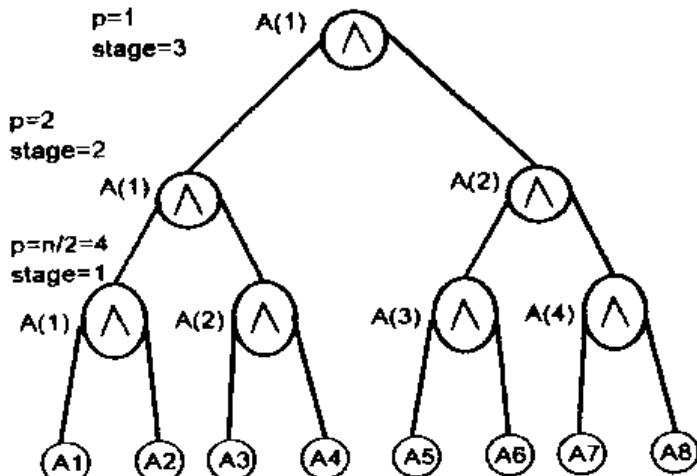


图1-20 $n=8$ 时的Boolean-AND-2算法

1.6 并行算法的性能

并行算法的性能主要通过下列三个因素评价：

1. 计算时间（时间复杂度）；
2. 所需要的处理器个数（处理器复杂度）；
3. 所需要的机器模型。

例如，我们在1.5节给出的算法的性能如下表：

算 法	时间复杂度	处理器复杂度	PRAM模型
Boolean-AND	$O(1)$	$O(n)$	EREW
Boolean-AND-1	$O(1)$	$O(n)$	EREW-ECR
Boolean-AND-2	$O(\log_2 n)$	$O(n)$	EREW

用常见符号表示的复杂度界限定义如下：

1. 如果存在正数 c 和 n_0 ，且对所有的 $n > n_0$ ，有 $T(n) < cf(n)$ ，那么 $T(n) = O(f(n))$ ；
2. 如果存在正的常数 c 和 n_0 ，且对所有的 $n > n_0$ ，有 $T(n) > cf(n)$ ，那么 $T(n) = \Omega(f(n))$ ；
3. 如果存在正的常数 c_1, c_2, n_0 和 n_1 ，且对所有的 $n > n_0$ ，有 $T(n) < c_1 f(n)$ ；对所有的 $n > n_1$ ，有 $T(n) > c_2 f(n)$ ，那么 $T(n) = \Theta(f(n))$ 。

即如果 $T(n) = O(f(n))$ ，且 $T(n) = \Omega(f(n))$ ，那么 $T(n) = \Theta(f(n))$ 。还有两个用于分析并行算法性能的量是加速比（speedup）和效率（efficiency）。

加速比和效率 考虑一个已知最好的串行算法的问题，其时间复杂度为 T_s 。我们有一个针对相同问题的并行算法，其时间复杂度为 T_p 和处理器复杂度 P 。我们定义

$$\text{加速比} = T_s/T_p$$

$$\text{效率} = T_s/PT_p$$

加速比最大不超过处理器个数，我们总是试图开发加速比几乎等于处理器数目的并行算法。

实际上，只有很有限问题的加速比能达到这一要求。效率最大为1，它表明所有处理器的有效利用率。

因为想要知道在并行计算机上运行应用程序的速度能有多快，并行领域的性能测量变得更加复杂。这一问题也可以说成我们利用并行得到的好处是什么。通常依据加速比来度量：

$$\text{加速比} = \frac{\text{串行执行时间}}{\text{并行执行时间}}$$

串行和并行执行时间的定义有多种方式。这种相异性导致有五种不同的加速比定义，即相对加速比 (relative speedup)、实际加速比 (real speedup)、绝对加速比 (absolute speedup)、渐近实际加速比 (asymptotic real speedup) 和渐近相对加速比 (asymptotic relative speedup)。Sahni和Thanvantri已经对上述性能度量进行了详细地研究。如想了解更多的信息，请参见参考文献Sahni [1997] and Thanvantri。

相对加速比 这里串行时间定义为运行在并行计算机的单个处理器上的并行程序的执行时间。因此，并行算法A在用 p 个处理器解决规模为 n 的问题实例 P 时的相对加速比定义为：

$$\text{Relative Speedup } (n, p) = \frac{\text{单个处理器用算法A解决问题P的时间}}{p\text{个处理器用算法A解决问题P的时间}}$$

并行系统的相对加速比并不是固定的，它依赖于问题的规模 n 和处理器的个数。相对加速比是 n 和 p 的函数。固定 n ，我们可以画出以 p 为一个坐标轴，相对加速比为另外一个坐标轴的相对加速比曲线。这可以用于在处理器数目增加时分析算法的性能。相似地，也可以固定 p ，画出以 n 为一个坐标轴，相对加速比为另外一个坐标轴的相对加速比曲线。这可以用于在问题规模增加时分析算法的性能。利用这两种曲线，我们可以找到相对加速比的最大点，称做最大相对加速比。平均相对加速比、最小相对加速比和期望相对加速比也可以用相似的方法定义。

实际加速比 在这一度量中，并行执行时间是和针对应用的最快串行算法或程序所需的时间相比较的。在并行计算机单个处理器上最快算法的执行时间将被采用。由于对很多应用我们尚不知道最快的算法；而对某些应用，不存在一种算法对所有情况都是最快的。因此，实际使用的串行算法执行时间被用来代替最快串行算法的执行时间。这样得到的加速比就是实际加速比。

$$\text{Real Speedup } (n, p) = \frac{\text{用最好的串行算法解决问题的时间}}{\text{用}p\text{个处理器解决问题的时间}}$$

我们可以像相对加速比那样定义最大实际加速比、最小实际加速比、平均实际加速比、期望实际加速比等。

绝对加速比 在此定义中，通过比较并行执行时间与最快串行算法在最快的串行计算机上的执行时间得到加速比。就像实际加速比那样，实际上我们是用“实际”使用的串行算法来完成比较。

$$\text{Absolute Speedup } (n, p) = \frac{\text{用最快的处理器和最好的串行算法解决问题的时间}}{\text{用算法A和}p\text{个处理器解决问题的时间}}$$

绝对加速比的定义可以扩展到最大绝对加速比、最小绝对加速比、平均绝对加速比等。

渐近实际加速比 设 $S(n)$ 为针对问题的最好串行算法的渐近复杂度， $P(n)$ 为在并行计算机

可以满足所需处理器的假设基础上并行算法A的渐近复杂度。渐近实际加速比定义为：

$$\text{Asymptotic Real Speedup}(n) = S(n)/P(n)$$

对于像排序之类的渐近复杂度不能用实例规模 n 惟一表征的问题，我们采用最坏情况下的复杂度。

渐近相对加速比 此加速比和渐近实际加速比的不同点在于串行复杂度，这里采用在单个处理器上运行的并行算法的渐近时间复杂度。

成本规范化加速比 除了想了解并行系统的运行速度之外，通常我们还想知道获得性能改进所需要的成本。因此定义成本规范化加速比（cost normalized speedup, CNS）如下：

$$\text{CNS}(n, p) = \frac{\text{speedup}(n, p)}{\text{并行系统的成本} / \text{串行系统的成本}}$$

53

为了得到成本规范化加速比，我们需要在加速比之外，得到并行系统的成本和串行系统的成本。并行系统的成本包括硬件和软件的成本。还要考虑是否包括维护成本、运算成本和保障人员成本等。由于折扣不同，每次购买硬件的价格都会不同。一年前购买的昂贵系统，今天可能花很少的钱就能买到。我们应该用去年的成本（即我们实际的花费）、现在的成本（即替代成本）还是明年的计划成本（即未来购买该系统的成本）呢？另外，并行系统的硬件所支持的用户数也与串行系统不同，我们需要对用户的硬件成本进行折旧。软件成本确定起来比较困难，除非购买的是标准商品软件（这在并行计算系统中不太可能，而在串行计算系统中有可能）。而且，我们仍然面临采用总体成本还是折旧成本、现在成本还是将来成本以及维护成本折旧和学习成本分解等问题。串行系统的成本还取决于所采用的加速比的版本。因此，一旦知道所采用的加速比版本，串行系统就很好定义。然而，实际确定串行系统的成本可能还是比较困难。其原因与并行系统一样。尽管成本规范化加速比可能是很吸引人的度量，但由于确定并行和串行系统的成本比较困难，这一度量不被广泛采用。

效率 效率是和加速比关系密切的性能尺度。它是加速比和处理器个数 p 的比值。由于所采用的加速比不同，可以得到不同的效率。就像加速比一样，效率不应该脱离运行时间来作为性能尺度。具体原因与加速比相同。

效率的一些替换公式可以在文献中找到。比如，有些作者定义效率为并行算法完成的工作（work accomplished, wa）和消耗的工作（work expended, we）之间的比值。我们定义wa为“最好的”串行算法所完成的工作；we为并行执行时间、单个并行处理器的速度 s 和处理器个数 p 的乘积。假定所有处理器具有相同的速度，我们有：

$$\begin{aligned} we &= \text{并行执行时间} \times s \times p \\ wa &= \text{最好的串行执行时间} \times s \\ \frac{wa}{we} &= \frac{\text{最好的串行执行时间}}{p \times \text{并行执行时间}} \end{aligned}$$

54

此效率定义等于实际加速比除以 p 。因此， wa/we 等于实际效率。相似地，如果wa定义为并行算法在单个处理器上执行时完成的工作，则 wa/we 等于相对效率（即相对加速比除以 p ）。

另外一个等价的效率公式根据浪费的工作（work wasted, ww且 $ww = we - wa$ ）来定义。可以写成：

$$\text{效率} = \frac{wa}{we} = \frac{wa}{ww + wa} = \frac{1}{1 + ww/wa}$$

可扩展性 对许多并行系统（即并行算法和并行计算机的结合）来说，当问题规模固定时，加速比会随着处理器数目的增加而降低；当处理器数目固定时，加速比会随着问题规模的增加而上升。并行系统的可扩展性就是指随着问题规模和处理器数目的增加，并行系统性能的改变。直观上说，如果系统的性能能够随着系统规模（问题规模和机器规模）的扩展（即增加）而不断提高，则此并行系统是可扩展的。想了解关于可扩展性的详细内容，请参见Sun的文章。

1.7 小结

本章介绍并行的概念并通过足够多的容易且合适的例子进行解释。还给出三种不同的模型。由于本书的主要内容不是研究二叉树或网络模型，我们没有对它们作详细介绍。对PRAM模型我们给出了必要的说明。本书的绝大多数算法是针对PRAM模型的。还对PRAM模型的绝对和并发读/写的能力作了介绍。对于并发写模型，为了解决当多个处理器同时对一个存储单元进行写时的冲突，本章介绍了三种不同的方法：ECR、PCR和ACR。还讨论了决定并行算法性能的因素，并对加速比和效率的概念作了解释。在介绍上述基础概念的同时，也给出了一些并行算法作为说明。

参考文献

- Aho, A., Hopcroft, J., and Ullman, J. (1974) *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.
- Akl, S. G. (1989) *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, NJ.
- Chaudhuri, P. (1991) *Parallel Algorithms, Design and Analysis*, Prentice-Hall, Englewood Cliffs, NJ.
- Cook, S. A. (1981) Towards Complexity Theory of Synchronous Parallel Computation, *L'enseignement Mathematique*, **XXVII**, 99–124.
- Cole, R. and Zajicek, O. (1989) The APRAM: Incorporating Asynchrony into the PRAM Model, *Proceedings ACM SPA*, 169–178.
- Cook, S. A. (1983) Overview of Computational Complexity *Comm. ACM*, **26**(6), 400–409.
- Day, K., and Tripathy, A. (1992) Arrangement Graphs: A Class of Generalized Star Graphs, *Information Processing Letters* **42**, 235–241.
- Gupta, A. and Kumar, V. (1993) Scalability of Parallel Algorithms for Matrix Multiplication, *Proc. International Conf. on Parallel Processing*, III, 115–119.
- Hall, D. and Driscoll, M. (1995) Hardware for Fast Global Operations on Multicomputers, *Proc. 9th Intl. Paral. Proc. Symposium*, 673–679.
- Helmbold, D. and McDowell, C. (1989) Modeling Speedup(n) Greater than n , *Proc. 1989 International Conf. on Parallel Processing*, III, 219–225.
- Hwang, K. and Briggs, F. A. (1984) *Computer Architecture and Parallel Processing*, McGraw-Hill, New York.
- Joseph, J. (1992) *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA.
- Kuck, D. J. (1976) Parallel Processing of Ordinary Programs, *Advances in Computers*,

- 15, 119–179.
- Kuck, D. J. (1977) A Survey of Parallel Machine Organisation and Programming, *ACM Comput. Survey*, 9, 29–59.
- Kung, H. T. (1980) The Structure of Parallel Algorithms, *Advances in Computers*, 19, 65–112.
- Kumar, V., Grama, A., Gupta, A. and Karypis, G. (1994) *Introduction to Parallel Computing*, Benjamin/Cummings, California.
- Leighton, T. (1992) *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufman, California.
- Mead, C. A. and Conway, L. A. (1980) *Introduction to VLSI Systems*, Addison-Wesley Reading, MA.
- Miller, R. and Stout, Q. F. (1992) *Parallel Algorithms for Regular Architectures*, MIT Press, Cambridge, MA.
- Sahni, S. and Thanvantri, V. (1996) “Performance Metrics: Keeping the Focus on Runtime,” IEEE-PDT, 1996 (Spring) 43–46.
- Stone, H. (1987) *High Performance Computer Architecture*, Addison-Wesley, Reading, MA.
- Valiant, L. G. (1982) Parallel Computations, *Proc. 7th IBM Symp. Math. Foundations of Comput. Sci.*
- Yoo, Y. B. (1983) Parallel processing for some network optimization problems, Ph.D. dissertation, Washington State University, Pullman, WA.
- X. H. Sun and D. Rover, “Scalability of Parallel Algorithm-Machine Combinations,” IEEE TPDS, Vol. 5, pp. 599–613, June, 1994.

56

习题

1.1 假定最多有五个处理器可以并行工作, 请画出下列计算的DAG图:

$$\text{a)} \quad u = \sum_{i=1}^{10} x_i^2$$

$$\text{b)} \quad \text{mean} = \frac{1}{10} \sum_{i=1}^{10} x_i$$

1.2 考虑求多项式值的问题:

$$y = a_{10}x^{10} + a_9x^9 + \cdots + a_3x^3 + a_2x^2 + a_1x + a_0$$

假定有十个处理器可以并行工作, 画出该计算的二叉树模型。

- 1.3 数值 x 起初存在于一个超立方体中的一个节点。请给出一个算法把 x 送到超立方体的所有其他节点。此问题称为消息广播问题。
- 1.4 在一个 k 维的超立方体中, 有 2^k 个数据, 每个节点一个。我们希望重新安排数据, 使得最小的数在处理器 P_0 中, 第二小的数在 P_1 中, 第三小的数在 P_2 中, 依次类推。最大的数最后一定在 P_{n-1} 中, 其中 $n = 2^k$ 。请为此问题设计一个并行算法。
- 1.5 考察数组 $A(0:n-1)$, 对于数 x , 秩($x:A$)是数组 A 中小于等于 x 的元素个数。假定数组 $A(0:n-1)$ 存在于超立方体中, 使得 A_i 在处理器 P_i ($0 < i < n$) 中。处理器 P_0 也包含一个键值 X 。请设计一个时间复杂度为 $O(\log n)$ 的算法求出秩($X:A$), 并存储在 P_0 中。假定 $n = 2^k$ 。
- 1.6 超立方体中的每一个处理器 P_i 都包含一个数值 a_i 。设计一个时间复杂度为 $O(\log n)$ 的

算法求数组 a_0, a_1, \dots, a_{n-1} 的平均值。

- 1.7 什么是环状网？它的直径是多少？
- 1.8 什么是弦环网络？请解释它为什么比普通网络更有优势。
- 1.9 什么是Barrel Shifter网络？请画出一个有15个节点的Barrel Shifter网络。
- 1.10 假定 A 是一个 $n \times n$ 的上三角矩阵。求解方程 $AX = B$ 的回代方法从求解最后一个方程 $a_{nn}x_n = b_n$ 的解 x_n 开始，然后把 x_n 代入前面的方程得到 x_{n-1} ，依次类推。请画出DAG图，并给出当处理器数目 $p < n$ 时，选择特定 p 和 n 的调度法。
- 1.11 在有 n^2 个处理器的超立方体上，设计一个时间复杂度为 $O(n)$ 的算法计算两个 $n \times n$ 矩阵的乘积。
- 1.12 请解释立方体连接环网络这一概念。
- 1.13 什么是星形网络？它的主要优点是什么？
- 1.14 在一个 k -立方体中，存有一个有 $n = 2^k$ 个元素的数组，其中的第 i 个元素位于处理器 P_i 中。请设计一个算法确定比特定数 X 小的数组元素的个数，其中 X 对所有处理器均已知。
- 1.15 针对线性处理器阵列求解问题1.13。
- 1.16 针对环状网求解问题1.14。
- 1.17 针对弦环网络求解问题1.14。

第2章 并行计算数据结构

算法其实就是在适当的数据结构上对数据进行操作的过程。数据结构是并行算法设计和分析中很重要的部分。对特定数据结构属性的深入了解可以为几大类重要的问题设计出非常高效的算法。基于数据结构的重要性，我们用一整章介绍并行算法设计和分析中的各种数据结构。考虑完整性，对所有的数据结构都进行了讨论。已经很熟悉数据结构的读者可以跳过本章。

2.1 数组和列表

数组通常代表同类数据项的集合。数组也是列表。例如，十个学生的分数用数组形式可以表示如下：

学生	1	2	3	4	5	6	7	8	9	10
分数	85	90	60	52	71	80	65	53	42	96

在算法的设计和分析中，我们经常用到下面的两个特殊数组类型：

- a) 堆栈；
b) 队列。

59
堆栈是一维数组，其元素的增加和删除都在一端完成。假定一个堆栈包含5个元素。指定的变量 $\text{top} = 5$ 表明在堆栈中有5个元素，第5个是最高项。如果新元素要加入，它可以作为第六项， top 就变成6。

如果想从有5个元素的堆栈（见图2-1）删除1个元素，只能把最高的元素去掉。向堆栈加入1个元素的简单过程如下所示：

过程STACK-ADD(S(1:n), top,item)

```
BEGIN
    IF top = n then
        call STACK-FULL
    ELSE
        top ← top+1
        S(top) ← item;
    Endif
END
```

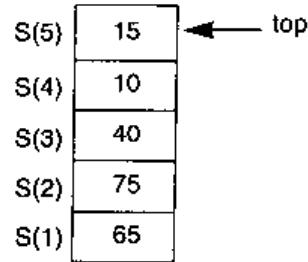


图2-1 有5个元素的堆栈

上述过程中假定堆栈的最大容量为 n 。如果堆栈已满，就不能再插入任何新元素。因此，如果 $\text{top} = n$ ，我们调用过程STACK-FULL，假定它会进行进一步的处理。与此相似，可以写一个进行元素删除的过程。如果 $\text{top} = 0$ ，我们假定堆栈已经空了。在这种情况下不能从堆栈中删除任何元素。

过程STACK-DEL(S(1:n), top,item)

```

BEGIN
    IF top = 0 then
        call STACK-EMPTY
    else
        item S(top)
        top ← top-1
    Endif
END

```

60

堆栈在串行和并行算法中均有大量的应用。算术表达式的前缀、后缀和中缀，递归过程，调度和多项式估值等都是堆栈的重要应用领域。

队列（见图2-2）是一个数组，元素的增加在其称作尾端（rear）的一端发生，元素的删除在其称作前端（front）的另一端发生。为了更方便地用队列进行算法设计，通常把它表示成循环数组的形式。当且仅当 $\text{rear} = \text{front}$ 时，我们假定队列是空的。为队列设计插入和删除元素的过程很容易，我们把它留给读者作为练习。

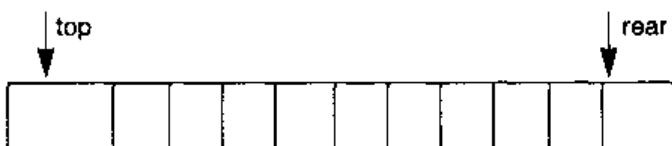


图2-2 队列

2.2 链接列表

称作列表的抽象数据类型是一个数据项序列的集合，这些数据项称作原子（atom）和针对这些数据的操作。如果 a_1, a_2, \dots, a_n 是列表中的原子，我们把列表写成 (a_1, a_2, \dots, a_n) 的形式。列表和集合的区别在于：

1. 在元素之间有顺序；
2. 如果应用程序需要，一个元素可以出现多次。

在列表上通常可以进行的操作有：

1. FIND(item) —— 检查元素是否在列表中，并给出其位置；
2. INSERT(item) —— 在列表中插入元素（通常在特定位置）；
3. DELETE(item) —— 删除元素的首次（或者所有）出现。

操作定义的灵活性可以适应多种应用。如果存在比较元素 a_i 和 a_j 的方法，即 $a_i < a_j$, $a_i > a_j$, 或者 $a_i = a_j$ ，我们可以定义有序列表。升序列表是对所有的 i , $a_i < a_{i+1}$ 的列表。

如果通过维护数组中的列表元素来实现一个列表，此列表为线性列表。典型的线性列表的表示形式为数组 $A(0:N)$ ，其中 $A(0)$ 存有列表元素的个数。这种表示方法局限了数组元素个数为 N ，很明显地会浪费 $A(A(0)+1), \dots, A(N)$ 。维护多个线性列表的集合是很困难的。比如，要维护列表 (A, B, C, D) , (M, N, O) 和 (W, X, Y, Z) ，每个列表允许保留十个元素。可以把这些列表看做二维数组 $A(0:3,0:10)$ 。

61

表2-1描述了上述列表的数据结构。现在考虑如何用如下的一个指向数组的指针数组来维护这三个列表。

名称	大小					
列表1	4	A	B	C	D	
列表2	3	M	N	O		...
列表3	4	W	X	Y	Z	...

名称	指 针
One	1
Two	5
Three	8

表2-1 二维数组的链接列表

位 置	数 据	链 接
1	A	5
2	X	11
3	M	8
4	W	2
5	B	9
6	O	0
7	D	0
8	N	6
9	C	7
10	Z	0
11	Y	10

名 称	指 针
One	1
Two	3
Three	4

指针数组称作索引。这样就有消除列表长度上限的优势。惟一的要求是列表不能在总长度上超过数组的大小。每个列表只占用它需要的空间，新列表可以很方便地在末端加入（见图2-3）。现在的问题是如何在列表中插入元素。把E插入列表1得到(A, B, C, D, E)的惟一方法就是把其他数据都向右移动，如图2-4所示。值得注意的是甚至索引都必须改变。

名 称	指 针
One	1
Two	6
Three	9

数据												
1	2	3	4	5	6	7	8	9	10	11		
A	B	C	D	M	N	O	W	X	Y	Z		
1	2	3	4	5	6	7	8	9	10	11		

图2-3 连续列表

A	B	C	D	E	M	N	O	W	X	Y	Z
1	2	3	4	5	6	7	8	9	10	11	12

图2-4 在第一个列表插入E后的列表

另外一个问题是在列表的删除。在很大的数组中追踪自由空间的轨迹是很困难的。因此利用线性列表找不到维护多个列表的有效实现方法。

链接列表的例子 可以替代线性列表且有更大灵活性的另一个数据结构是链接列表。其元素称做节点 (node)。一个节点包含数据和与列表中另外一个节点的链接。列表的起始由指针指示。如果有一个变量存有指向列表起始的指针，它称为列表的名称。前面例子的链接列表结构如表2-1所示。

指针值0代表任何不能作为链接的值。如果提取第一、第二和第三个列表，我们可以在图2-5中画出此结构。这一列表结构的名称为ONE。

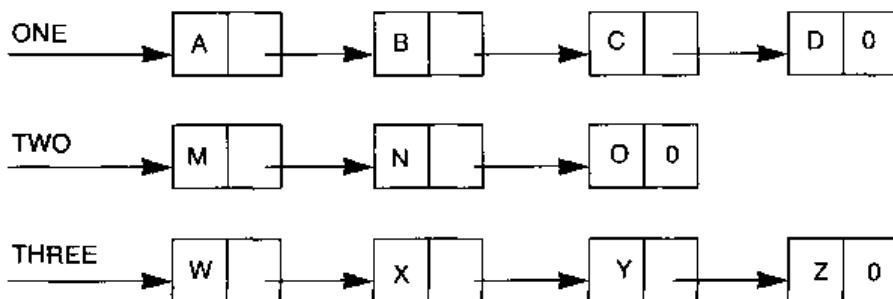


图2-5 三个链接列表

特别地，这些列表可以是任意长度。我们可以在一个存储区域中同时维护几个列表。惟一的空间限制是它们总的存储空间大小不能超过可用的存储空间。

链接列表在搜索上有一个很明显的弱点。如果线性列表是经过排序的，那么就可以用对分搜索。考虑图2-6a中的线性列表T，其链接列表的等价表示如图2-6b所示。

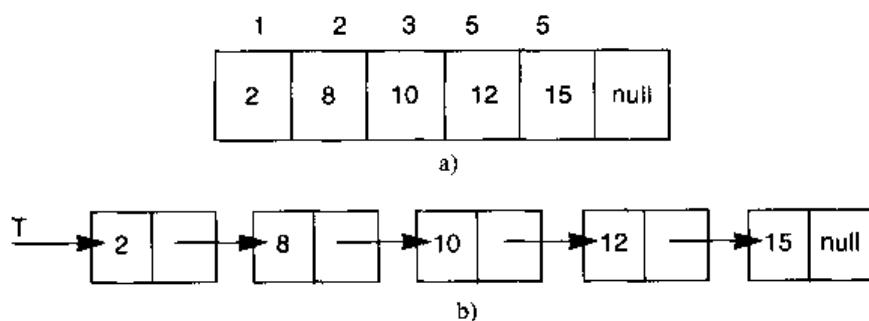


图 2-6

a) 列表 b) 链接列表

对分搜索函数可以通过求列表的中点而得到。下面的讨论描述了对分搜索的过程。上述线性列表中数组元素1和5之间的中点是3。但是在链接列表中却无法找到两个节点之间的中间节点，所以也不可能进行对分搜索。

循环列表 在循环列表中，最后节点的指针单元不是空的，而是指向第一个节点。这在图2-7a中给出。

双向链表 链接列表或循环列表最基本的困难是只能在一个方向移动。为了能双向移动，我们介绍双向链表的概念。双向链表的节点有两个链接，左链接LLINK和右链接RLINK。每个列表有指向左端和右端的指针。典型的双向链表的结构如图2-7b所示。

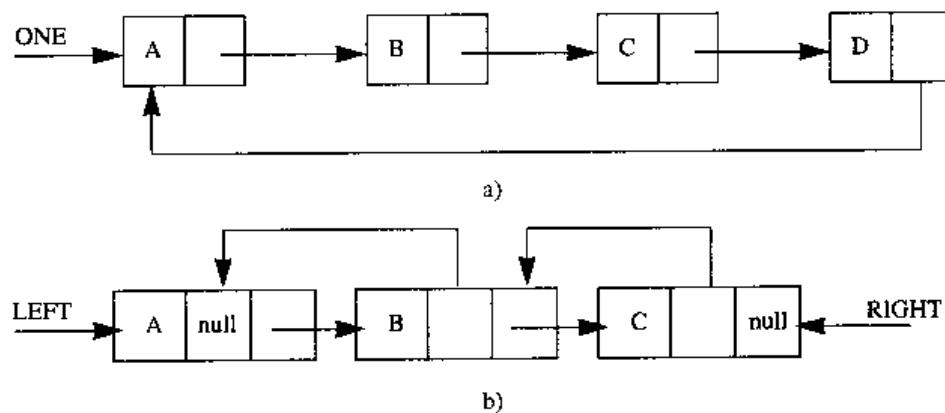


图 2-7

a) 循环链接列表 b) 双向链表

值得注意的是，随着LLINK走，会有一个链接列表；随着RLINK走，会有另一个列表。在链接列表中删除一个节点特别容易。链接可以帮助你找到不同的节点。假定节点X有左右两个相邻节点。它可以通过下面两条指令删除：

`RLINK(LLINK(X))=RLINK(X)`

`LLINK(RLINK(X))=LLINK(X)`

这使得最后生成的双向链表是成功跳过X的列表。在双向链表中的两个链表各自都可以循环。

为了插入一个包含所需数据的节点，我们首先必须得到一个未使用的节点。为此，我们维护一个特别的链接列表叫作自由表，用来保存当前不用的节点。当第一次为节点分配内存时，便初始化此列表。

2.3 图与树

本节介绍一类重要的数据结构——图模型，并给出以后几章将用到的各种类型的图以及一些性质。

2.3.1 预备知识

假设 V 是任意集合， E 是 $V \times V$ 的子集，我们称 (V, E) 是一个图，并记为 $G = (V, E)$ 。其中 V 称为图 G 的顶点集且 V 中的元素称为图 G 的顶点， E 称为图 G 的边集且 E 中的元素称为图 G 的边。

设 $V = \{a, b, c, d, e, f\}$ ， $E = \{(a, b), (b, c), (c, d), (c, e), (a, d), (b, d)\}$ ，图 $G = (V, E)$ 可以用图2-8表示。

若 $e = (v_1, v_2)$ 是一条边，我们称 e 与 v_1, v_2 关联，且 v_1, v_2 是相邻的顶点。边 (v, v) 称为环（self edge）。边 $e_1 = (v_1, v_2)$ 与 $e_2 = (v_1, v_2)$ 称为平行边（parallel edge）。不含环及平行边的图称为简单图（simple graph）。顶点集是无穷集的图称为无限图，否则称为有限图。若一个图的边集为空集，则称其为零图（null graph）。没有边与之相关联的顶点称为孤立点。与顶点 v 相关联的边的数目称为 v 的度（degree）。图2-8中 f 是一个孤立点，且图中各顶点的度分别为：

顶点	a	b	c	d	e	f
度	2	3	3	3	1	0

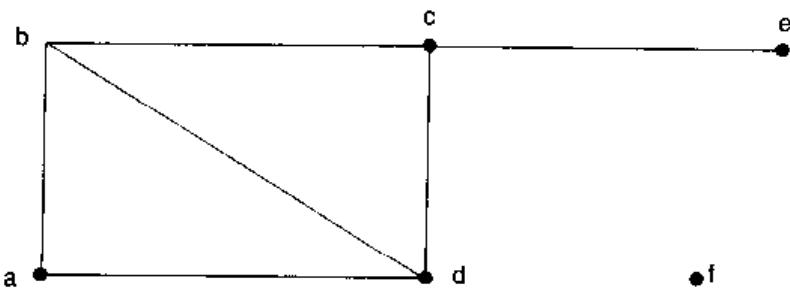


图2-8 一个图例

度为1的顶点称为悬挂点（pendant vertex），图2-8中 e 为悬挂点。通常用 n 和 m 分别表示顶点的数目和边的数目，我们感兴趣的是所有顶点的度的总和。由于每条边与两个顶点相关联，因此我们有如下结论：

1. 图的所有顶点的度的总和为 $2m$ ；
2. 图中度为奇数的顶点的个数通常为偶数。

正则图与完全图 若图中各顶点的度相同，则称其为正则图。一个含 n 个顶点的简单图，其最多可能含有 $n(n-1)/2$ 条边。一个图若含有所有可能的边，则称其为完全图。一个含有 n 个顶点（ $n(n-1)/2$ 条边）的完全图记为 K_n 。 K_n 是度为 $n-1$ 的正则图。见图2-9和图2-10。

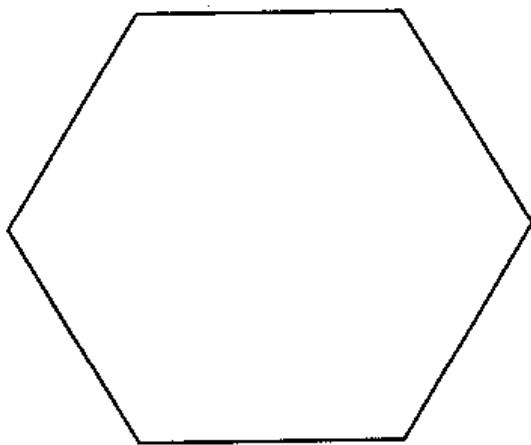
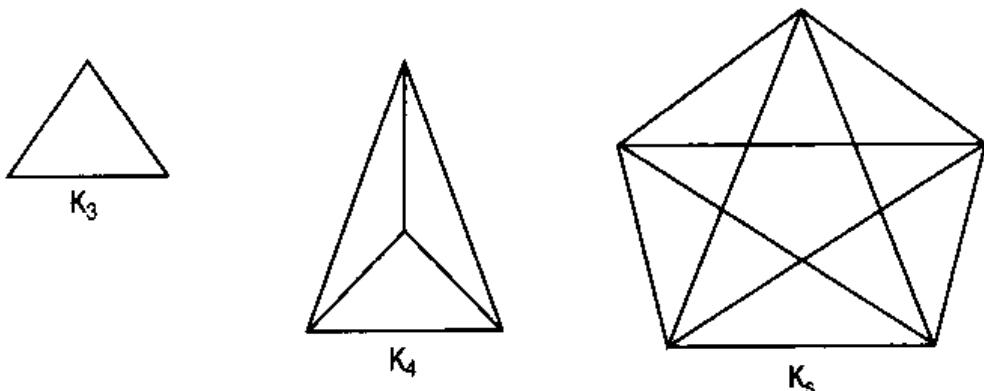


图2-9 度为2的正则图

通道、路、圈 假设 $G = (V, E)$ 是一个简单图， $v_1, v_2, v_3, v_4, \dots, v_k$ 是图 G 的顶点，且 v_i 与 v_{i+1} （ $1 \leq i < k$ ）相邻。若在序列中所有边仅出现一次则称序列 $v_1, v_2, v_3, v_4, \dots, v_k$ 为通道（walk）。 v_1 称为起点， v_k 称为终点。在通道的定义中可能存在下列情况：虽然 $i \neq j$ ，但是 $v_i = v_j$ 。如果 $v_1 = v_k$ ，

称为闭通道，否则称为开通道。在开通道中，若所有顶点仅出现一次，则称为路（path）。在闭通道中，若所有顶点仅出现一次，则称为圈（cycle or circuit），也就是说圈是一条闭路。



67

图2-10 一些完全图

在图2-11中， a,b,c,d,b,e,f 是一条通道，但它不是一条路，因为 b 点重复出现。 a,b,c,d 是一条由 a 到 d 的路； c,b,e,p,h,e,b,d 既不是一条路，也不是一条通道；边 (b,e) 重复出现两次，因此不是通道。 a,b,e,f,g,a 是一个圈。在圈中连接两个不相邻顶点的边称为弦（chord）。图2-11中， (b,f) 是圈 a,b,e,f,g,a 的一条弦。

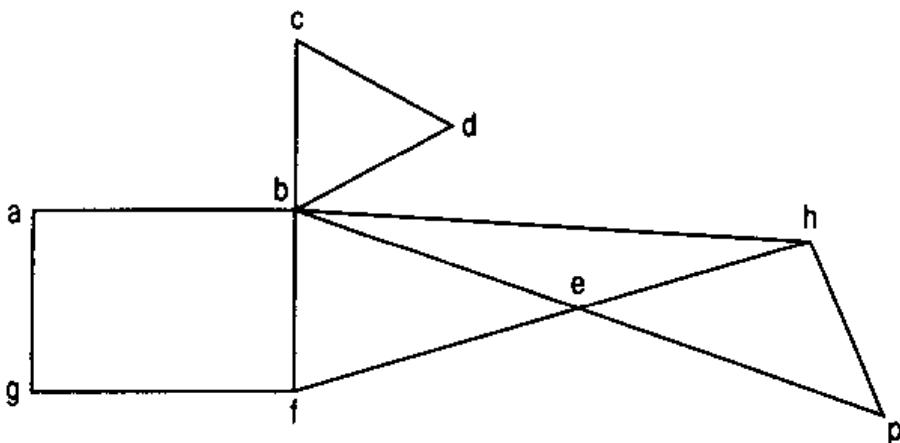


图2-11 路和通道的说明图

子图 假设 $G = (V, E)$ 是一个图， V' 是 V 的子集， E' 是 E 的子集且满足 E' 中所有的边只与 V' 中的顶点相关联，则称图 $G' = (V', E')$ 为 G 的子图。图2-12中 b 是 a 的子图。若图 G 的任何两个顶点 u 和 v 之间都有路存在，则称图 G 为连通图。对于一个非连通图，其最大的连通子图称为连通支（connected component）。若 V' 是 V 的子集， E' 是具有 V' 中两端顶点的图 G 的所有边的集合，则称 $G' = (V', E')$ 是由 V' 导出的图 G 的诱导子图。图2-12中 b 不是 a 的诱导子图； c 和 d 分别是 $\{1,5,4\}$ 和 $\{2,3,4\}$ 导出的 a 的诱导子图。两个子图如果没有公共边，则称为边不相交。图2-12中 c 和 d 是 a 的边不相交子图。通常有如下结论：

1. 图 G 是它自身的子图；
2. 图中任一顶点都可以看作具有此顶点的图的诱导子图；
3. 图中任一条边 (u, v) 都可以看作由 $\{u, v\}$ 导出的诱导子图；
4. “子图”关系是自反的、反对称的、传递的。

68

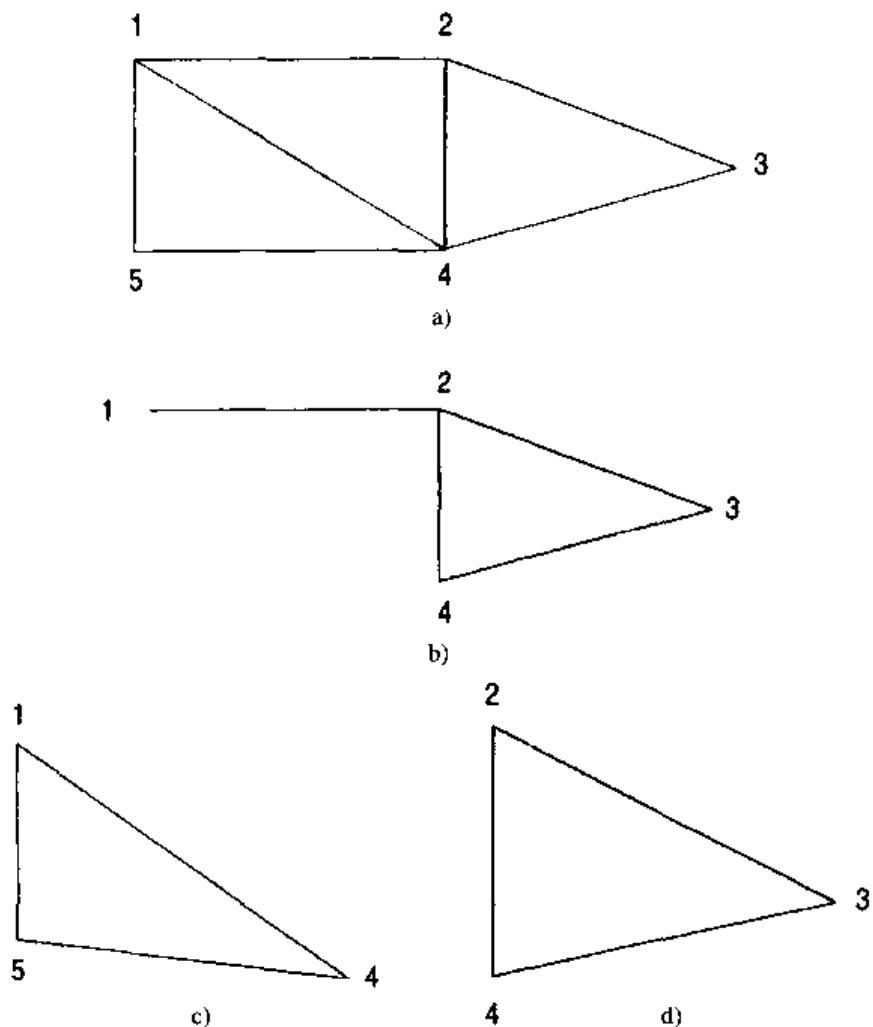


图 2-12

a) 图 G b) G 的子图 c) 由{1,4,5}导出的子图 d) 由{2,3,4}导出的子图

同胚 G 是一个图， v_1, v_2, v_3 是一条路，且 v_2 的度是2。 (v_1, v_2) 和 (v_2, v_3) 称为相继边 (series edge)。若用一条边 (v_1, v_3) 代替两条边 (v_1, v_2) 和 (v_2, v_3) 并且去掉顶点 v_2 ，则称为相继边融合 (merging of series edges)。若 (u, v) 是一条边，增加一个新的顶点 w 并且使之分别与 u, v 都相邻，同时去掉原始的边 (u, v) ，则称为度为2的顶点的插入。两个图 G_1 和 G_2 ，如果其中的一个图可以由另一个图经过有限个相继边的融合和(或)度为2的顶点的插入得到，则称它们是同胚的。

一个完全子图称为团 (clique)。一个团如果不是任何其他团的子图则称为极大团 (maximal clique)。对于图2-13a，图2-13b是其相应的极大团。

同构 对于图 $G = (V, E)$ 和 $G' = (V', E')$ ，如果在 V 与 V' 间存在一个一一映射，使得 V 中的两个顶点相邻的充要条件是相应的 V' 中的顶点也相邻，则称两个图是同构的。如果两个图是同构的，则有如下性质：

1. 有相同数目的顶点；
2. 有相同数目的边；
3. 对于给定的度，有相同数目的顶点。

上面的三个条件是必要但不充分的。寻找简单又有效的标准去考察两个图是否同构是一个有趣的研究课题。这个问题称为同构问题。

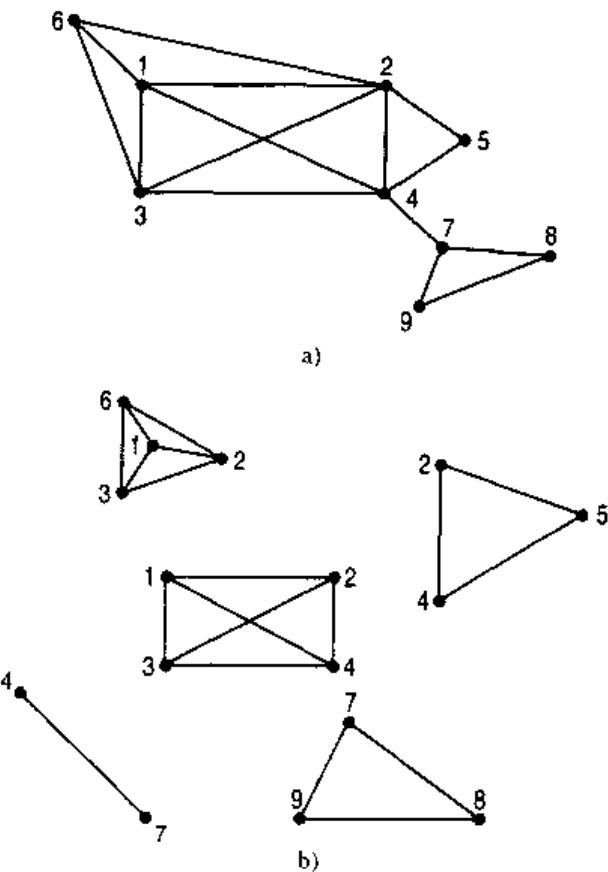


图 2-13

a) 图 G b) G 的极大团

2.3.2 欧拉图与哈密顿图

本节介绍两个特殊类型的图——欧拉图和哈密尔顿图，它们在各个领域都有广泛应用。

70

欧拉图是由欧拉给出的，这一图模型来源于著名的哥尼斯堡七桥问题。

对于图 $G = (V, E)$ ，一条遍历图的每条边恰好一次的闭通道称为欧拉闭迹。并不是每个图都含有欧拉闭迹。例如，四个顶点的完全图 K_4 不含有欧拉闭迹。含有欧拉闭迹的图称为欧拉图。

定理2-1 (欧拉定理) 当且仅当连通图的每个顶点的度是偶数时，此连通图是欧拉图。

定理的证明并不难。由于通道是闭合的且经过每条边恰好一次，无论何时当这条通道经过一条边到达一个顶点时，它必须要从这个顶点离开而经过另一条边。由这个事实我们可以证明上述定理。同样地，由欧拉闭迹的定义容易证明下面的定理。

定理2-2 当且仅当图的边集 E 可以分解成 $E = E_1 \cup E_2 \cup \dots \cup E_k$ 且每一个 E_i 由一个圈组成时，此图为欧拉图。

一个开通道仅一次遍历所有的边，称为欧拉开迹。一个图若含有欧拉开迹则称为一笔图。不难证明如下定理。

定理2-3 当且仅当图中含有两个奇数度的顶点时，此图为一笔图。

利用定理2-1设计一个串行多项式算法，可用来检验一个图是否为欧拉图。也可以通过验证每个顶点的度是否是偶数来判定图是否为欧拉图。如果每个顶点的度是偶数，我们可以判定该图是欧拉图。类似地，可以通过计算奇数度的顶点的个数来判定一个图是否为一笔图。

哈密顿图 一条闭合通道遍历图的每个顶点恰好一次，则称该图为哈密顿图 (Hamiltonian

circuit)。这个定义类似于欧拉闭迹的定义。欧拉闭迹是指遍历每条边恰好一次，而哈密顿圈是指遍历每个顶点恰好一次。 K_4 是哈密顿图但不是欧拉图。如果两个圈 C_1, C_2 恰有一个公共顶点，则 $C_1 \cup C_2$ 是欧拉闭迹不是哈密顿圈。连通图是否存在欧拉闭迹的充要条件已在定理2-1和定理2-2给出。然而，它是否存在哈密顿圈的充要条件仍然没有解决。尽管这个问题早在1859年就由爱尔兰数学家William Rowan Hamilton提出，但至今仍有待解决。

2.3.3 树

不含圈的连通图称为树。下面的叙述是等价的。

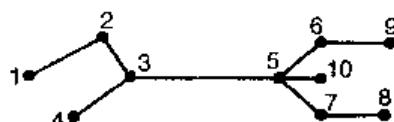
71

1. G 是一个树；
2. G 中的任何两点间恰有一条路；
3. G 是连通的且含有 n 个顶点和 $n-1$ 条边；
4. G 是极小连通图；
5. G 不含圈且有 n 个顶点和 $n-1$ 条边。

任何一个树至少有两个悬挂点。路的长度等于其边的数目。从顶点 v 到其他顶点的最长的路的长度称为 v 的离心率或直径，并记为 $E(v)$ 。

$$E(v) = \text{Max}\{d(v, u) | u \in V\}$$

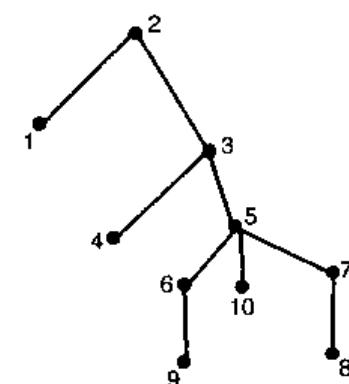
这里 $d(u, v)$ 表示从 v 到 u 的路的长度。在树 T 中，具有最小离心率的顶点称为从 v 到 u 树的中心。一个树含有一个或两个中心。图2-14a中给出了所有顶点的离心率。3和5有最小的离心率，因此图中有两个中心3和5。中心的离心率称为树的半径。树中最长路的长度称为树的直径。对于图2-14a中给定的树，其半径是3，直径是5。我们可以指定一个顶点作为树的根。如果一个顶点被指定为树的根，则称这个树为有根树。



顶点	1	2	3	4	5	6	7	8	9	10
离心率	5	4	3	4	3	4	4	5	5	4

a)

层数	顶点
0	2
1	1, 3
2	4, 5
3	6, 7, 10
4	8, 9



b)

图 2-14

a) 半径=3和直径=5的树 b) 有根树

72

考虑图2-14a中所给的树。如果指定2作为树根，就可重新描绘该树如图2-14b。作为有根树，每个顶点的层数这样定义：树根被指定为第0层，与根相邻的顶点称为根的孩子，它们被指定为第1层。树根又称为孩子的双亲。

在图2-14b描述的有根树中，1和3是2的孩子。如果顶点 v 位于 i 层，任何不是 v 的双亲且与它相邻的顶点 u 被指定在 $i+1$ 层，这样的顶点称为 v 的孩子。如果有根树的最大层数是 k ，那么它的高度（或深度）为 $k+1$ 。

图2-14b中树的高度为5。树的连通子图称为子树。图2-15展示了树及其子树。有根树通常用双亲关系来描述。假设 $T = (V, E)$ 是一个有根树，其根为 r ，我们可用数组PARENT(1:n)来表示（ n 是顶点数目）：

如果 i 不是树根，那么 $\text{PARENT}(i) = i$ 的双亲；

如果 r 是树根，那么 $\text{PARENT}(r) = r$ 。

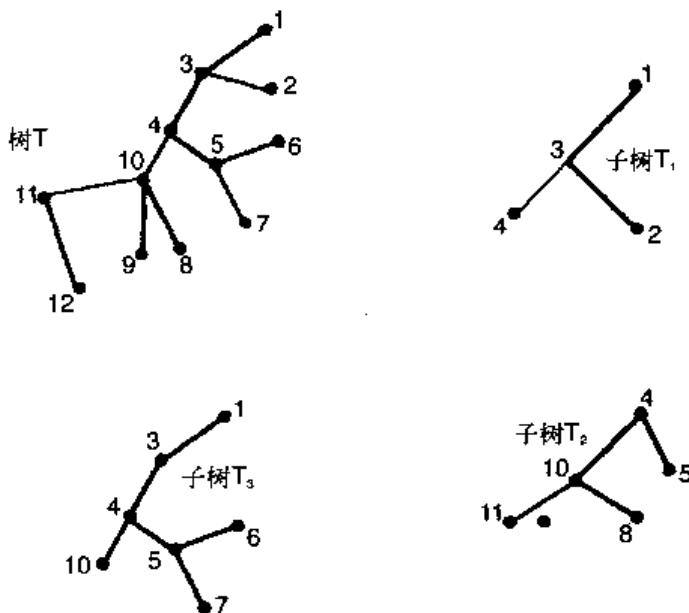


图2-15 树和它的一些子树

有些情况下定义PARENT为-1。图2-16展示了树及其PARENT表示。

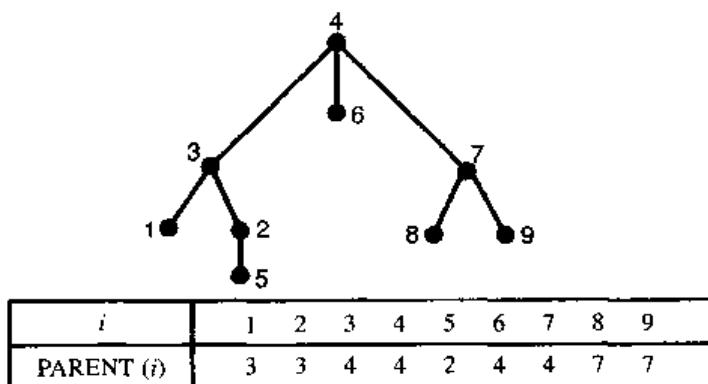


图2-16 树根及其PARENT表示

二叉树 有根树若其每一个顶点至多有两个孩子则称为二叉树。二叉树被广泛应用于计算

机科学中。这两个孩子通常称作左孩子和右孩子。若其没有顶点含有左孩子，则称为右斜 (right skewed) 二叉树。类似地可定义左斜二叉树。图2-17所示为左斜和右斜二叉树。考虑图2-18所示的二叉树：在第0层存在惟一的树根，第1层有2个顶点，第2层有4个顶点，第3层有8个顶点。通过观察有如下定理：

定理2-4

- 在一个二叉树中，第*i*层至多有 2^i 个顶点；
- 高度为*k*的树的顶点最大数目为 $2^k - 1$ 。

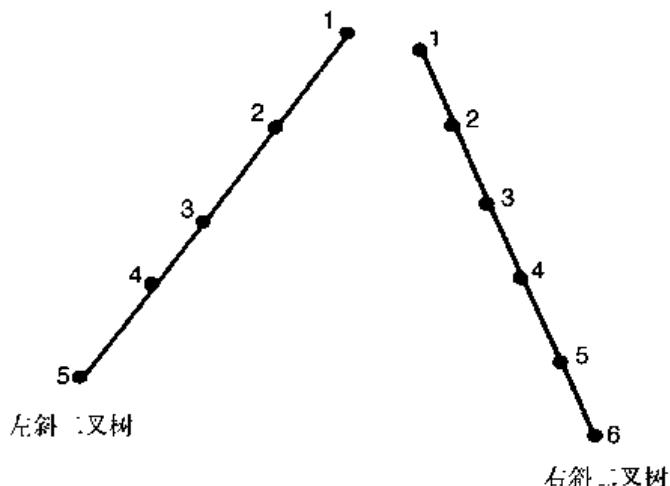


图2-17 斜二叉树

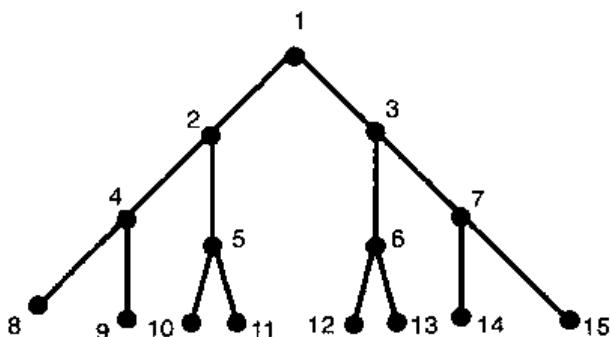


图2-18 满二叉树

证明：用归纳法证明a)。

当*i*=0时，结论是正确的。

假设第*i*层时结论正确，即第*i*层至多有 2^i 个顶点。由于第*i*层的每个顶点至多能产生2个孩子。因此第*i*+1层至多有 2^{i+1} 个顶点。由归纳法可知a)成立。

b) 的证明基于如下的数值结果：

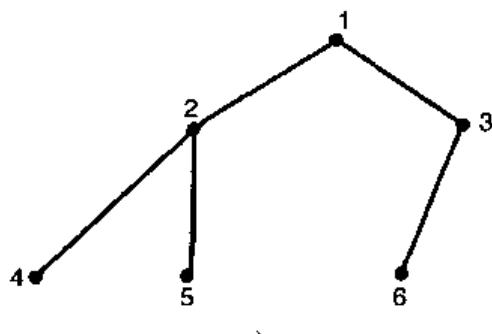
$$2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} = 2^k - 1$$

高度为*k*且有 $2^k - 1$ 个顶点的树称为满二叉树。满二叉树的顶点可用数字按序列编号：树根为1，第1层的两个顶点自左至右记为2和3；当对第*i*-1层的所有顶点编完号后，再用连续数字对第*i*层的顶点自左至右编号。图2-18给出了一个3层的满二叉树，并用连续数字对各个顶点编号。下面的定理对于满二叉树很容易证明。

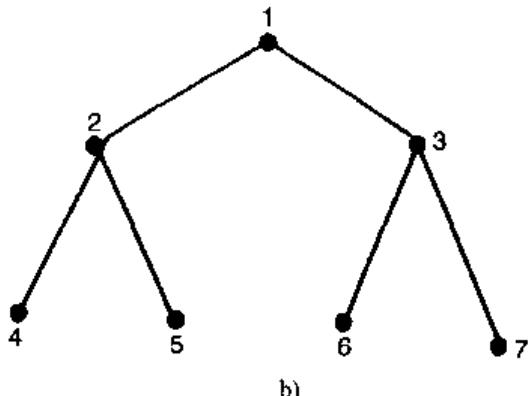
定理2-5 对于用连续数字编号的满二叉树而言，

- a) 第*i*个顶点的左孩子是 $2i$ ；
- b) 第*i*个顶点的右孩子是 $2i+1$ ；
- c) *i*的双亲是 $[i/2]$ 。

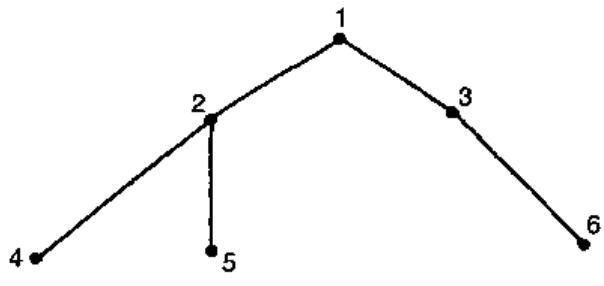
在满二叉树中，如果去掉某些具有最高数字编号的顶点，则称为完全二叉树。图2-19a所示为完全二叉树，图2-19b则是满二叉树，而图2-19c中是不完全二叉树。在完全二叉树中顶点编号是连续的，并且与同样高度的满二叉树一致。定理2-5对于完全二叉树亦成立。考虑一个二叉树，每个悬挂点给定一个正权，用 l_i 和 w_i 分别表示层数和悬挂点的权。考虑 $\sum l_i w_i$ ，它取遍了所有的悬挂点，这个和称为加权路径长度 (weighted path length)。例如图2-20中，6个悬挂点 d, q, g, p, r 和 f ，相应的权分别为5, 10, 7, 3, 4和8。权和层数如下表所示。



a)



b)



c)

图 2-19

a) 完全二叉树 b) 满二叉树 c) 不完全二叉树

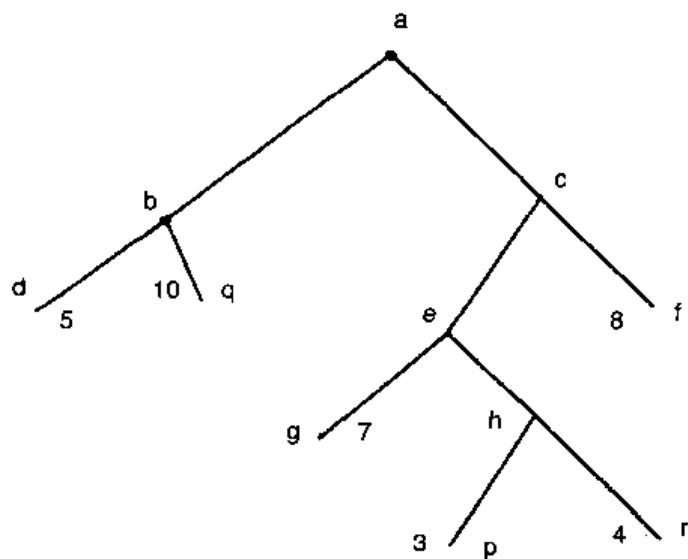


图2-20 加权路径长度=95的树

悬挂点	权	层数	$l_i w_i$
d	5	2	10
q	10	2	20
g	7	3	21
p	3	4	12
r	4	4	16
f	8	2	16
$\sum l_i w_i$			95

给定 n 个悬挂点及相应的权，构造一个具有最小加权路径长度的二叉树是一个有趣的问题。它在判定树以及优化程序结构等方面有许多应用。图2-21给出了几个二叉树及其加权路径长度。

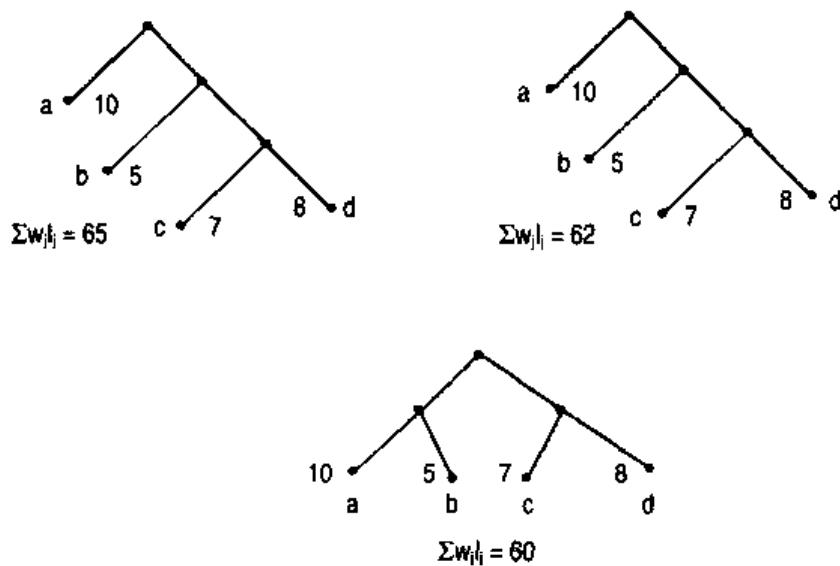
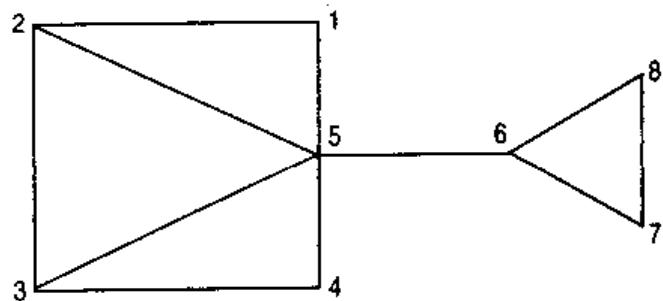


图2-21 二叉树及其加权路径长度

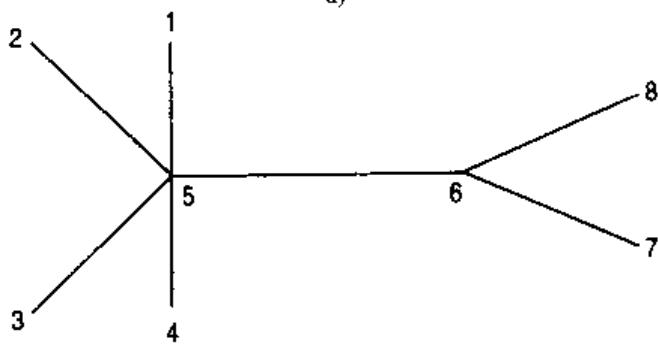
支撑树 $G = (V, E)$ 是一个连通的简单图，具有 G 所有顶点的无圈的连通子图 $T = (V, E')$ 称

为 G 的支撑树。图2-22a至d给出了图 G 和它的支撑树。

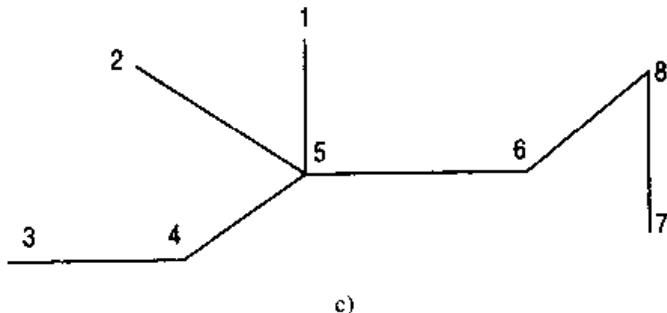
78



a)

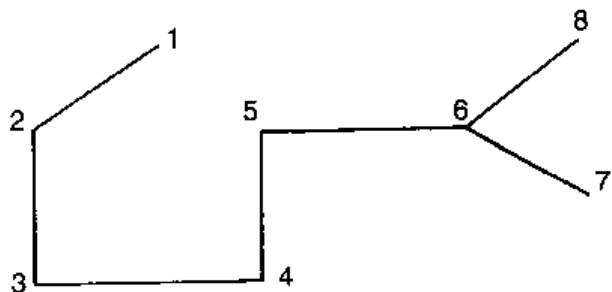


b)



c)

79



d)

图 2-22

a) 图 G b) 支撑树 T_1 c) 支撑树 T_2 d) 支撑树 T_3

图2-23a中， a,b,c,d,e 和 f 分别表示6个城市，城市之间的距离用边的权来表示。假设在城市中已经安装了通讯网络，并且认为在图2-23b中所示的支撑树的边之间建立的链接已经足够。在两个城市之间通讯电缆的安装费用与它们之间的距离成正比，即与边的权成正比，因此安装的总成本与边的权总和成正比。我们必须找到一个支撑树，使得其边的权总和最小。给定一个

无向图，寻找最小权支撑树是一个有趣的问题。最小权支撑树也称为最小成本支撑树 (minimum cost spanning tree) 或最短支撑树。每一个含 n 个顶点的连通图 G 都有一个含 $n-1$ 条边的支撑树 T ，这 $n-1$ 条边称为树枝。不在树中的边称为弦。如果图 G 是不连通的，它就有许多分支。在这种情形下，我们可以对每一个分支寻找相应的支撑子树，这些支撑子树的集合称为 G 的支撑森林。图 2-25 给出了在图 2-24 中所示的不连通图的支撑森林。

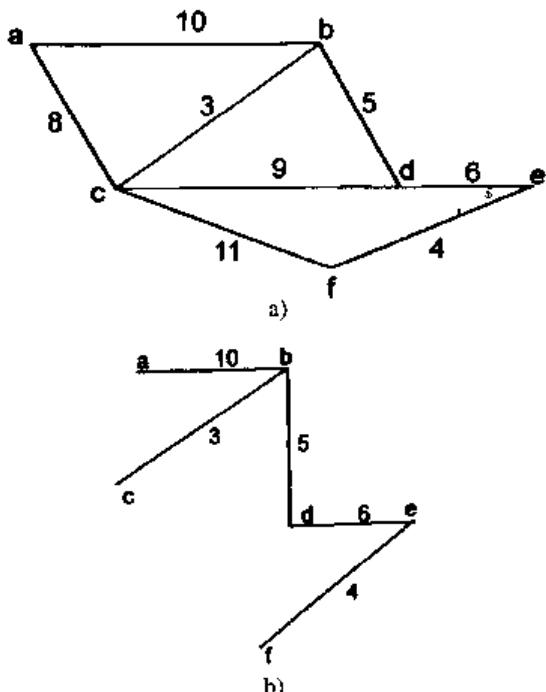
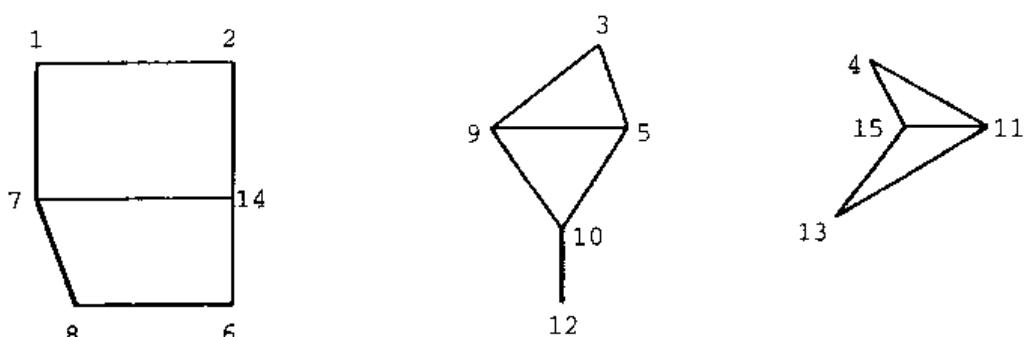
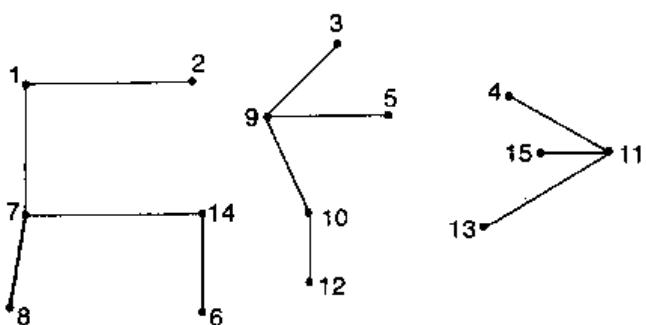


图 2-23

a) 城市和它们之间的距离 b) 六个城市的通讯链路，总成本为 $10 + 3 + 5 + 6 + 4 = 28$

图 2-24 不连通图 G 图 2-25 G 的支撑森林

假设图 G 有 n 个顶点, m 条边和 c 个分支。 G 的支撑森林有 $n-c$ 个树枝和 $m-n+c$ 条弦。我们定义图的秩(rank)为树枝的个数, 图的零维数(nullity)为弦的个数。在图2-25a中, $n=15$, $m=18$, $c=3$,

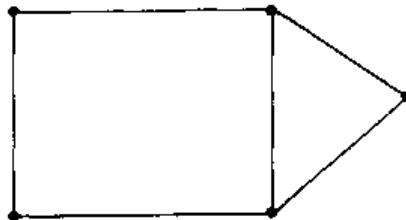
$$\text{秩} = n - c = 15 - 3 = 12$$

$$\text{零维数} = m - n + c = 18 - 15 + 3 = 6$$

注意秩与零维数之和等于边的数目。对于连通图而言, 其秩为 $n-1$, 零维数为 $m-n+1$ 。考虑连通图 G , T 是支撑树, e 是弦。如果把 e 放在 T 中, 则生成一个圈, 并设这个圈为

$$eb_1b_2\cdots b_t$$

这里 b_1, b_2, \dots, b_t 是树枝, e 是惟一的弦。这个圈称为基本圈(fundamental cycle)。总共有 $m-n+1$ 个基本圈。计算图的支撑树的数目也是一件很有趣的事。假设 G 是一个图, T 是它的支撑树, e 是一条弦, $eb_1b_2\cdots b_t$ 是基本圈。如果将 e 加到 T 中同时去掉 T 中的某条边 b_i , 则得到不同的支撑树(称为 $\{T_i(e)\}$)。对于每一条弦 e 我们可以找出这些支撑树 $\{T_i(e), i=1, 2, \dots, t\}$ 。对于给定的支撑树 T , 可以通过重复上述操作来得到一些支撑树。图2-26a、b、c给出了图 G 及其支



a)

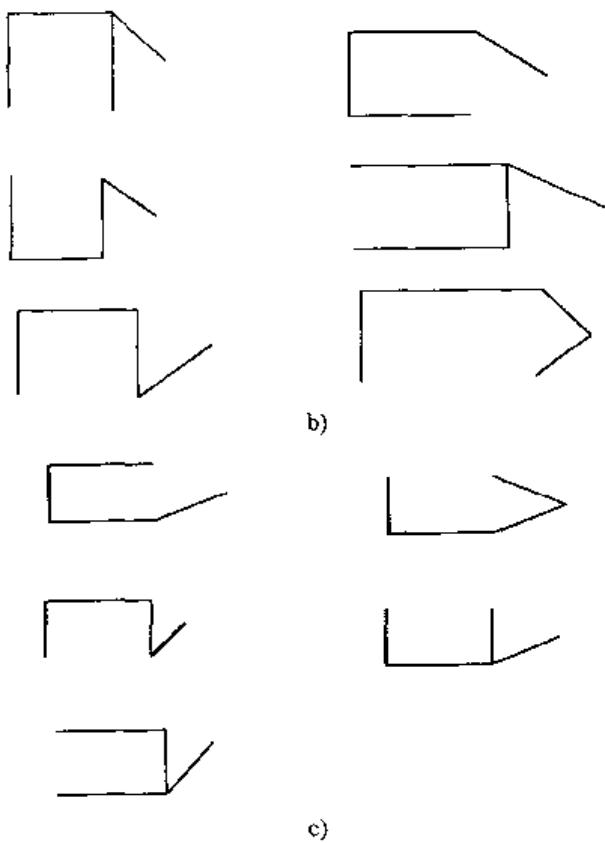


图 2-26

a) 图 G b) G 的一些支撑树 c) G 的另一些支撑树

撑树。在支撑树中加弦以及从基本圈中去掉树枝的操作称为圈互换。给定图G的任意两个支撑树 T_1 和 T_2 ，其中一个可由另一个经过有限次圈互换得到，并且圈互换的次数等于不同时在两个支撑树中的边的数目。

我们定义：

$$\begin{aligned} d(T_1, T_2) &= \text{由 } T_1 \text{ 得到 } T_2 \text{ 所需的圈互换的次数} \\ &= \text{在 } T_1 \text{ 中而不在 } T_2 \text{ 中的边的数目} \end{aligned}$$

由观察可知， $d(T_1, T_2)$ 满足如下性质：

1. $d(T_1, T_2) \geq 0$ 并且 $d(T_1, T_2) = 0$ (当且仅当 $T_1 = T_2$)；
2. $d(T_1, T_2) = d(T_2, T_1)$ ；
3. 对于任一不同于 T_1 或 T_2 的支撑树 T_3 ，满足 $d(T_1, T_2) \leq d(T_1, T_3) + d(T_3, T_2)$ 。

另外， $d(T_1, T_2)$ 不能超过图的秩或零维数。假设G是连通图， T 是G所有支撑树的集合，用 T 作为顶点集。可将图G表示成 $G_T = (T, E_T)$ ，当且仅当 $d(T_1, T_2) = 1$ 时， T 的两个顶点 T_1, T_2 之间用一条边相连。图 G_T 称为图G的树图 (tree graph)。

2.3.4 图的遍历

很多关于图的问题需要系统地访问图的每一个顶点。为了实现对顶点的遍历，下面给出两种方法：

1. 广度优先搜索 (BFS)；
2. 深度优先搜索 (DFS)。

为了遍历图，起始点是已知的。假定 v 是起始点，BFS由 v 点开始。访问完 v 点后，按照规定的顺序访问所有与 v 相邻的点 v_1, v_2, \dots, v_d 。访问完这些顶点后，再访问与这些顶点相邻的未被访问的顶点，重复这个过程直到所有的顶点都被访问。在DFS技术中，由 v 点开始访问与 v 相邻的顶点 v_1 ，再访问与 v_1 相邻而未被访问过的顶点。某种情况下，如果没有与 v_1 相邻且未被访问过的顶点，退到 v_1 并且访问与 v_1 相邻且未被访问的顶点，重复这个过程直到退回到 v ，并且访问所有与 v 相邻的顶点。遍历形成G的支撑树。图2-27a、b、c分别表示图G，图G的BFS支撑树和DFS支撑树。

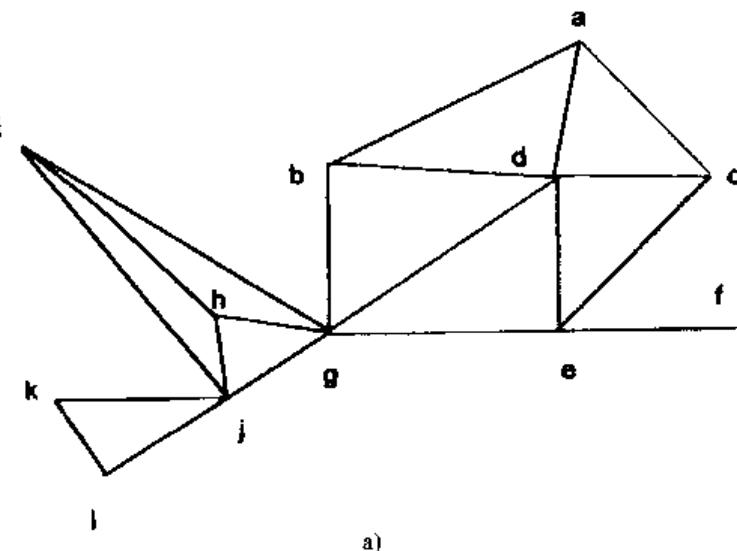


图 2-27

a) 图G

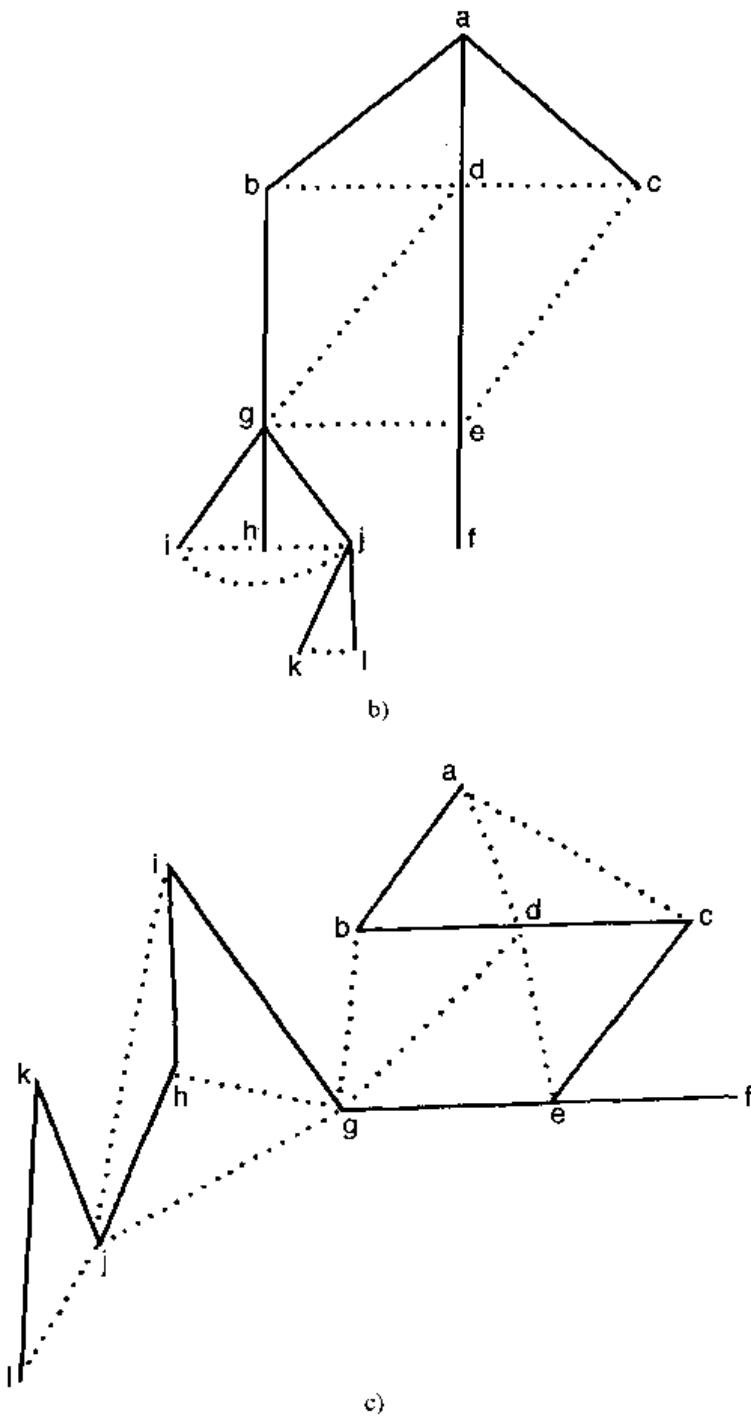


图2-27 (续)

b) BFS支撑树 c) DFS支撑树

在BFS支撑树中，虚线表示弦，实线表示树枝，而且每一条弦连接的顶点或者在同一层上，或者在连续层上。这个性质被有效地用于解决图论中的一些问题。

2.3.5 连通性

设 $G = (V, E)$ 是一个图， $v \in V$ ， $G - v$ 表示由 $V - \{v\}$ 导出的图。如图2-28a至c所示。设 $G =$

85 (V, E) 是一个有 k 个分支的图， $a \in V$ 。如果图 $G - a$ 的分支数多于 k 个，则称 a 为断点 (articulation point)。在连通图中，如果去掉某个点后图变得不连通了，就把该点称为断点。如图2-28a所

示，顶点4和7是两个断点。不含有断点的连通图称为2-连通图（biconnected graph）。任何圈是2-连通的。设 $G = (V, E)$ 是一个图， u, v 是 G 的同一分支的两个不相邻的顶点， S 是 V 的子集。如果去掉 S 后， u, v 位于不同的分支上，则称 S 是 $u-v$ 分离集（separator）。换句话说，如果 S 是 $u-v$ 分离集，那么不存在一条不经过 S 的顶点从 u 到 v 的路。对于 $u-v$ 分离集，若其不包含其他的 $u-v$ 分离集，则称为极小 $u-v$ 分离集。具有最少顶点数的 $u-v$ 分离集称为最小 $u-v$ 分离集。

$$S_1 = \{1, 2, 3, 4\} \quad S_2 = \{1, 2, 4\} \quad S_3 = \{2, 4\}$$

$$S_4 = \{2, 4, 8\} \quad S_5 = \{8\} \quad S_6 = \{5\}$$

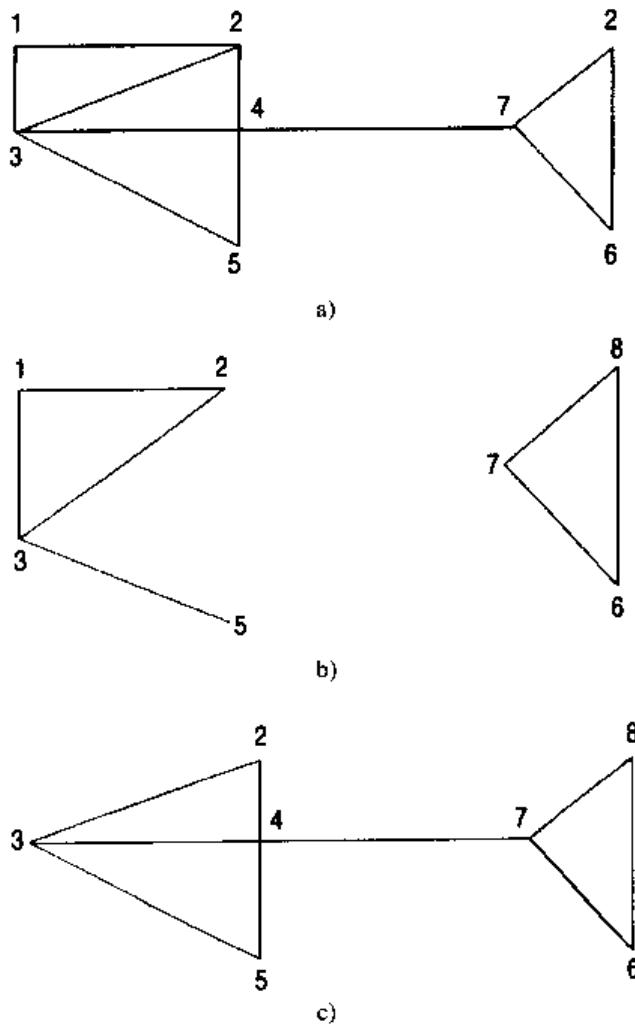


图 2-28

a) 图 G b) 图 $G-4$ c) 图 $G-1$

S_1, S_2, S_3, S_4, S_5 和 S_6 每个都是图 G （如图2-29）的7-6分离集。 S_3 是一个极小7-6分离集，同样， S_5 和 S_6 也是极小分离集。 $S_5 = \{8\}$ 和 $S_6 = \{5\}$ 是图 G 的两个最小7-6分离集。

设 $G = (V, E)$ 是一个连通图， S 是 V 的子集。如果 $G-S$ 是不连通的，则称 S 是 G 的分离集。如果对于任意 S 的子集 S' ， S' 不是分离集，则称 S 是极小分离集。具有最少顶点数的极小分离集称为最小分离集。最小分离集中顶点的个数称为图的顶点连通度。换句话说，连通图的顶点连通度等于使得余下的图成为不连图所需要去掉的最少顶点数为不连通图。注意，如果图含有断点，则顶点连通度是1。顶点连通度为1的图称为可分离图（separable graph）。不可分离图也叫做2-连通图。2-连通图的顶点连通度至少是2。顶点连通度至少是3的连通图称为3-连通图

(triconnected graph)。换句话说，对于3-连通图，去掉任何一个或两个顶点都不能使它变得不连通。由观察可知，任一3-连通图都是2-连通的，任一2-连通图都是连通的。顶点连通度至少为 k 的连通图称为 k -连通图(k 是正整数)。很容易得到下面的结论：

1. 当且仅当对于任意两个顶点 u, v ，由 u 到 v 至少存在两条内部点不相交的路时，连通图是2-连通的。
2. 当且仅当对于任意两个顶点 u, v ，由 u 到 v 至少存在三条内部点不相交的路时，连通图是3-连通的。
3. 当且仅当对于任意两个顶点 u, v ，由 u 到 v 至少存在 k 条内部点不相交的路时，连通图是 k -连通的。

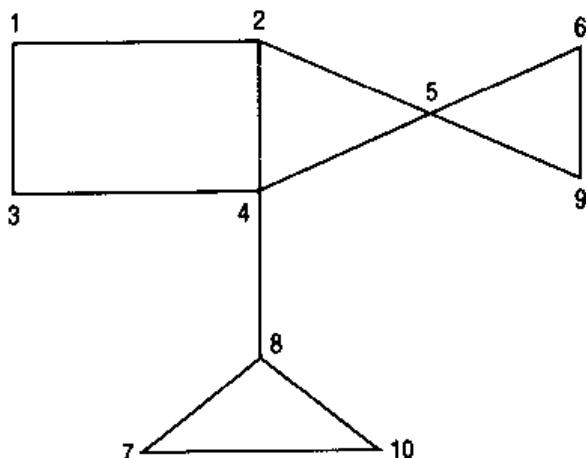
图2-29 图 G

图2-28a和图2-29中所示的图是连通的，但不是2-连通的。任一个圈 C_r 是2-连通的。图2-30所示是一个3-连通图。如果 d 表示图 G 中顶点的最小度，则 G 的顶点连通度至多是 d 。

87

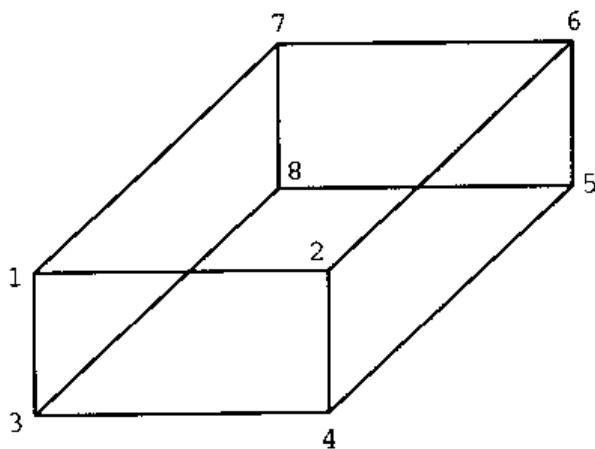


图2-30 3-连通图

不连通图有不止一个连通支。(一个连通支就是一个极大连通子图。)同样地，我们定义2-连通支是图的极大2-连通子图。如果图本身是2-连通的，则只有一个2-连通支。图2-31b给出了图2-31a的2-连通支。2-连通支也称为块(block)。如果图 G 中每一个块都是完全子图，则称该图为块图(block graph)。图2-32a所示为块图。由下列算法可找出图 G 的2-连通支：

1. 如果图 G 没有断点，则它自身是惟一的2-连通支；

2. 对于每一个断点 $a \in V$, 令 $V_1, V_2, V_3, \dots, V_s$ 分别表示 $G - \{a\}$ 不同分支的顶点集, G_i 表示由顶点 $V_i \cup \{a\}$ 导出的图 ($i = 1, 2, 3, \dots, s$)。

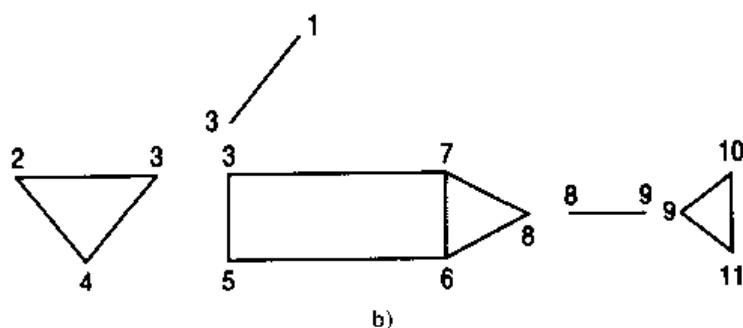
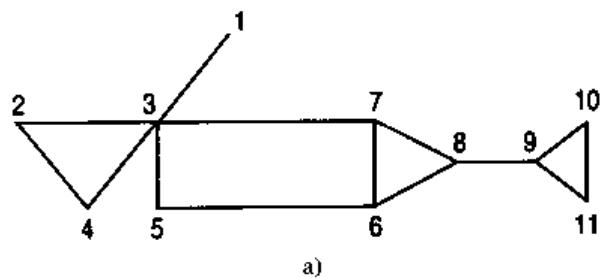


图 2-31

a) 图G是连通图但不是2-连通图 b) 图G的2-连通分支

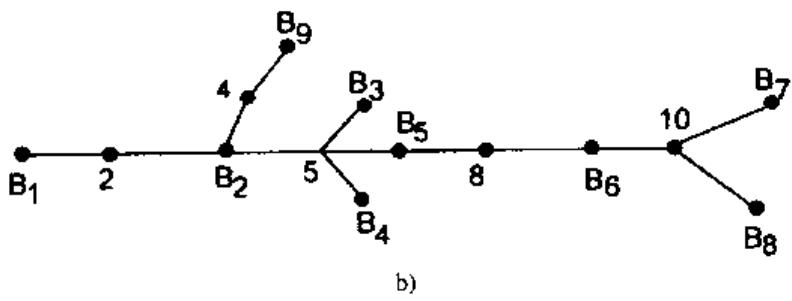
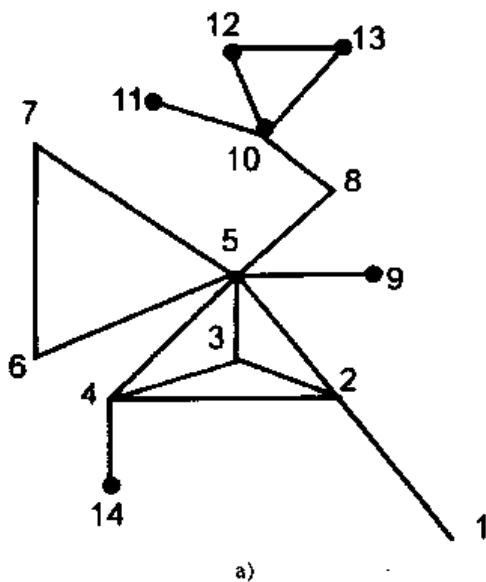


图 2-32

a) 块图 b) 图2-32a)中块图的BC树

所有的定义和过程都可以推广到 k -连通支 ($k > 2$)。块图可以用它的Block-Cut顶点树表示(BC树)。例如, 考虑图2-32a中所示的块图, 它的块为,

$$B_1 = \{1, 2\}, B_2 = \{2, 3, 4, 5\}, B_3 = \{5, 6, 7\}, B_4 = \{5, 9\}, B_5 = \{5, 8\}$$

$$B_6 = \{8, 10\}, B_7 = \{10, 11\}, B_8 = \{10, 12, 13\}, B_9 = \{4, 14\},$$

割点是2, 4, 5, 8, 10。BC-树的顶点集由块和割点组成, 仅当割点包含在某个块中时, 块和割点用一条边相连。图2-32b给出了图2-32a所示的块图的BC树。

2.3.6 可平面图

88 任何图都可以用多种方法来图示, 图2-33给出了图 $G = (V, E)$ 的几种图示, 其中

$$V = \{a, b, c, d, e, f, g, h\}$$

$$E = \{(a, b), (b, c), (c, d), (d, a), (e, f), (f, g), (g, h), (b, e), (a, e), (b, f), (c, g), (d, h)\}$$

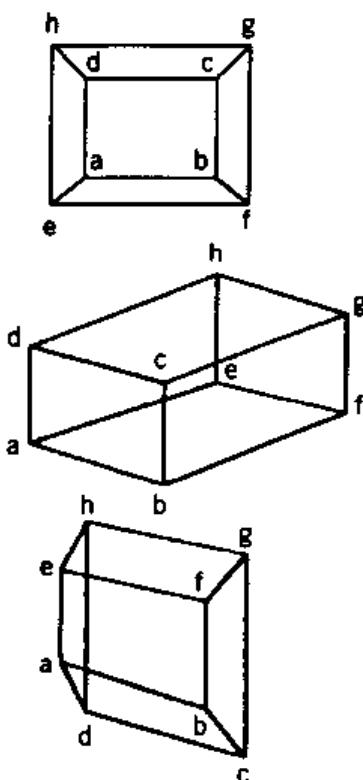
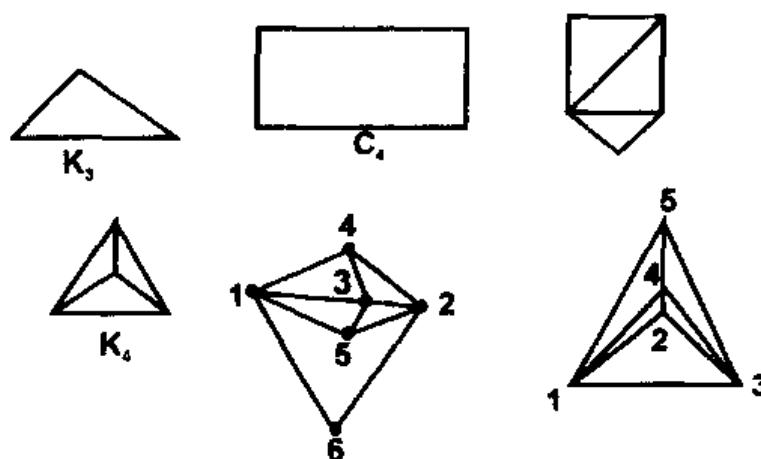


图2-33 同一个图的不同的图示

89 我们感兴趣的是在二维平面上描绘一个图, 使得任何两条边彼此不相交。很显然, 并不是所有的图都能如此。如果一个图可以在二维平面上描绘, 并使得任何两条边彼此不相交, 则称此图为可平面图。图2-34给出了可平面图的几个简单例子。如果将图 $G = (V, E)$ 的顶点集 V 分成两部分 V_1 和 V_2 , 使得图 G 的每一条边都连接 V_1 的一个顶点和 V_2 的一个顶点, 则称图 G 为二部图。注意二部图不含有奇数长度的圈。如果 V_1 的每一个顶点都与 V_2 的所有顶点相邻, 则此二部图称为完全二部图。完全二部图中如果 V_1 有 m 个顶点, V_2 有 n 个顶点, 则记为 $K_{m,n}$ 。注意, $K_{m,n}$ 含有 $m+n$ 个顶点和 mn 条边。波兰数学家库拉托斯基 (Kuratowski) 证明了如下两个图 (图2-35) 是不可平面图。

1. 含有5个顶点的完全图 (K_5) ;

2. 完全二部图 $K_{3,3}$ 。



90

图2-34 一些可平面图

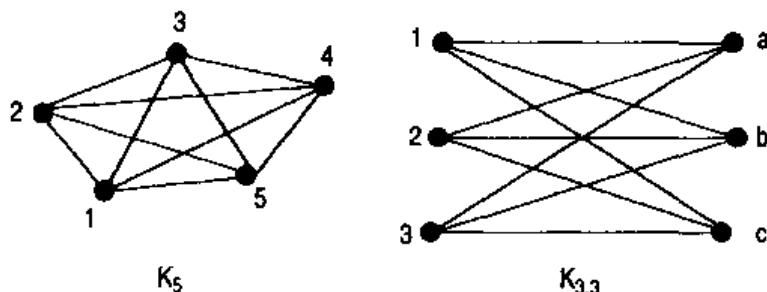


图2-35 库拉托斯基图

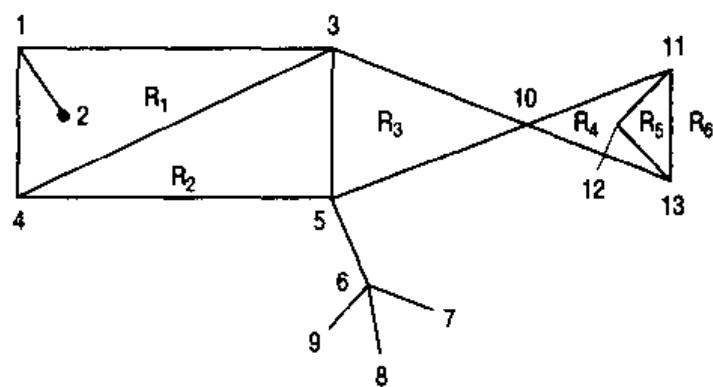
观察可知，库拉托斯基图是正则图，而且是不可平面图。我们还注意到，考虑顶点数和边数，库拉托斯基图是极小不可平面图。

考虑可平面图 G ，将其画在一个平面上（边不相交）。图 G 将平面分成若干个区域。如图2-36所示。 R_1, R_2, \dots, R_6 表示可平面图 G 在平面上所划分的区域。

很容易得到下面结论：

1. 任何一个简单可平面图都可以嵌入平面，且图的每一条边可用一条直线段来描绘；
2. 任何一个简单可平面图都可以嵌入平面，且任意指定的区域可以是无穷区域；
3. 任何一个简单可平面图都可以嵌入球面；可以嵌入球面的图是可平面图。

含有 n 个顶点和 m 条边的连通可平面图 G 能产生 $m-n+2$ 个区域。如图2-36， $n=13$ ， $m=17$ ，产生的区域数是 $m-n+2=17-13+2=6$ 。



91

图2-36 有6个区域的可平面图

同时还可证明，含有 n 个顶点和 m 条边的连通可平面图满足： $m \leq 3n - 6$ 。这是一个必要条件而不是充分条件。库拉托斯基的第二个图就满足这个条件，但它不是可平面图。图G是可平面图的充分必要条件是图G中不含有库拉托斯基的两个图或不含有与库拉托斯基的两个图同胚的图。将可平面图嵌入到平面，如果图的所有顶点都在外部区域上，则称此平面图为外可平面图。在外可平面图中，如果在任何两个不相邻的顶点之间增加一条边而使之不再是外平面图，则此外可平面图是极大外可平面图（mop）。哈密顿图就是mop图。

2.3.7 染色与独立集

染色是指将图的每个顶点染上颜色。许多研究人员已经研究了对图的边进行染色的问题，称为边染色（edge coloring）。对顶点进行染色称为顶点染色（vertex coloring）。在这里我们只考虑顶点染色，因此提到染色就是指顶点染色。如果任何两个相邻的顶点染不同的颜色，则称此染色为正常染色（perfect coloring）。

图2-37所示为图G和它的三种不同的正常染色。我们主要感兴趣的是用最小数目的颜色给一个图染色。这样的染色称为正常最小染色（perfect minimum coloring），最小染色的颜色数称为色数（chromatic number）。

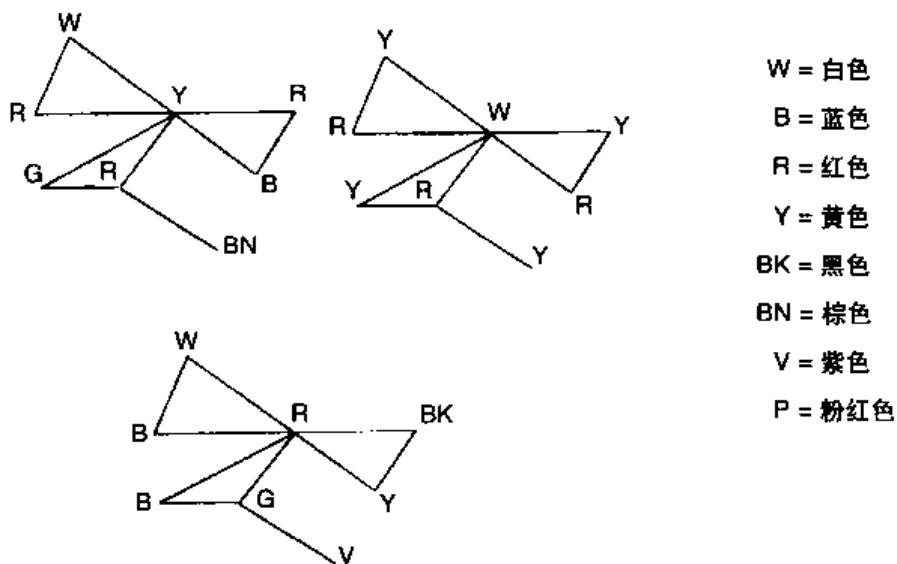


图2-37 图的三种正常染色

下表中列出了一些常见图的色数：

序号	图	色数
1	K_n	n
2	树	2
3	无边图	1
4	偶数长度的圈	2
5	奇数长度的图	3
6	二部图	2

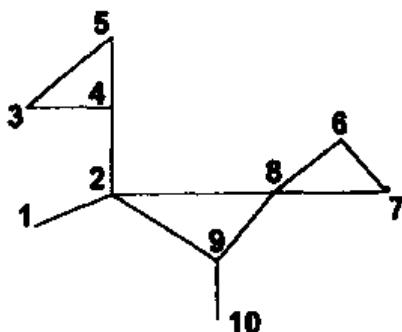
完全子图称为团。有 r 个顶点的团 C 称为 r -团。如果不存在包含团 C 的其他团，则称团 C 是极大的。顶点数最大的极大团称为最大团。最大团的顶点数称为团数。为了给一个 r -团染色，

我们需要 r 种颜色，因此色数 \geq 团数。

2.3.8 团覆盖

图 $G = (V, E)$ 的团覆盖是指将 V 划分成 V_1, V_2, \dots, V_k ，使得每个 V_i 都是一个团。图2-38给出了图 G 的两个不同的团覆盖。

团覆盖1	团覆盖2
$V_1 = \{1\}$	$V_1 = \{1, 2\}$
$V_2 = \{10\}$	$V_2 = \{3, 4, 5\}$
$V_3 = \{2, 8, 9\}$	$V_3 = \{6, 7, 8\}$
$V_4 = \{6, 7\}$	$V_4 = \{9, 10\}$
$V_5 = \{3, 4, 5\}$	



93

图2-38 团覆盖为 $\{1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}, \{9, 10\}$ 的图

团覆盖 $\{V_1, V_2, \dots, V_k\}$ 的容量指划分的集合数 k 。容量最小的团覆盖称为最小团覆盖。对于图 $G = (V, E)$ 的一个顶点集 X ，如果 X 中的任何两个顶点都不相邻，则称 X 为独立集。独立集也称为稳定集 (stable set)。对于任一不在 X 中的顶点 v ， $X \cup \{v\}$ 不是独立集，则称 X 是 V 的极大独立集。顶点数最大的极大独立集称为最大独立集。

对于图2-38，我们给出如下的独立集：

$$X_1 = \{1, 4, 8, 10\} \quad X_2 = \{1, 10, 6, 3\} \quad X_3 = \{8, 4\} \quad X_4 = \{1, 7, 10\}$$

其中 X_1 和 X_2 是最大独立集。最大独立集中的所有顶点用相同的颜色染色，团中的每一个顶点用不同的颜色染色。

2.3.9 交图

设 F 是一个子集族，对图 $G = (V, E)$ 构造如下：顶点集 V 与 F 中的子集一一对应。对于 V 中的两个顶点，当且仅当与它们相对应的两个子集有非空交集时，这两个顶点相邻。这样由 F 构造出来的图 G 称为 F 的交图。

设 $F = \{P_1, P_2, P_3, P_4, P_5\}$ ，这里 $P_1 = \{1, 2, 3\}$ ， $P_2 = \{2, 3, 4, 5, 6\}$ ， $P_3 = \{5, 6\}$ ， $P_4 = \{6, 7, 8, 9\}$ ， $P_5 = \{4, 10\}$ 。 F 的交图如图2-39所示。

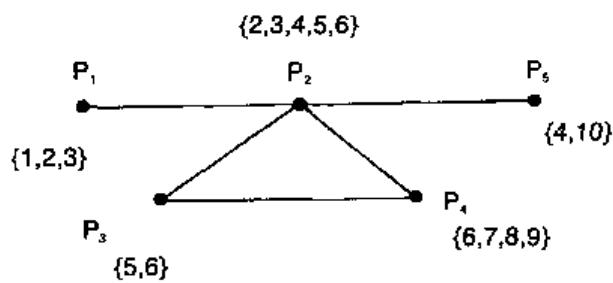


图2-39 $F = \{P_1, P_2, P_3, P_4, P_5\}$ 的交图

2.3.10 弦图

设 $\{v_1, v_2, \dots, v_i, v_j\}$ 是图的一个圈，连接两个不相邻的顶点 v_i 和 v_j 的任一条边称为弦（chord）。设 x 是图的一个顶点， $\text{ADJ}(x)$ 是与 x 相邻的所有顶点的集合，即一个团。我们称 x 是单纯的（simplicial）。如果图中每一个长度大于3的简单圈有一个弦，则称该图为弦图（chordal graph）。顶点序列 v_1, v_2, \dots, v_n 称为完美消去序列（PEO）或完美消去格式（PES）（当且仅当对所有 i ，由 $\{v_i, v_{i+1}, \dots, v_n\}$ 导出的图中的 v_i 是单纯的）。例如，图2-40给出了弦图 G 及一个PEO。在这个图中， $\{v_5, v_7, v_8\}$ 是一个 v_4-v_6 分离集， $\{v_5, v_7\}$ 是一个极小 v_4-v_6 分离集。假定 $A = \{1\}$, $B = \{1\}$, $C = \{1, 2\}$, $D = \{2, 3\}$, $E = \{3, 4\}$, $F = \{4, 5\}$, $G = \{3, 4, 5\}$, $H = \{5\}$ 。考虑子集族 $F = \{A, B, C, D, E, F, G, H\}$ ， F 的交图。如图2-40所示，其中 $v_1 = A$, $v_2 = B$, $v_3 = C$, $v_4 = D$, $v_5 = E$, $v_6 = F$, $v_7 = G$, $v_8 = H$ 。

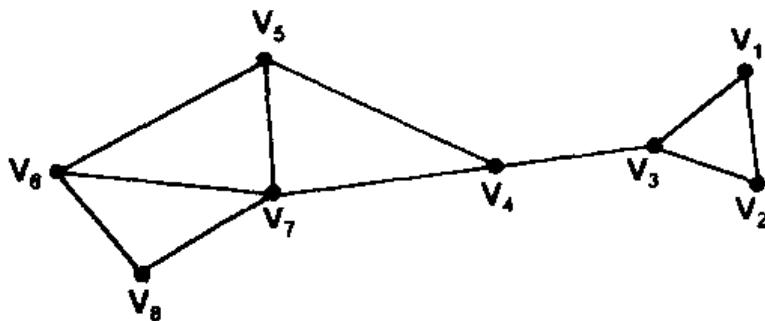


图2-40 弦图及PEO

弦图是树的子树的交图。存在一个无向树 T ，子树族 $F = \{S_1, S_2, \dots, S_r\}$ ，从 F 到 V 的双射并具有如下性质： G 中两个顶点相邻当且仅当相应的两个子树有公共顶点。给定一个弦图 G ，构造一个树使得 T 的顶点集同构于 G 的极大团集合。 T 的边构造如下：对于 G 中任一顶点 v ，与包含 v 的团相应的顶点导出 T 的一个子树，这样构造出来的树称为弦图 G 的团树。例如，考虑图2-41a所示的弦图 G ，图的顶点集 $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ ，极大团为 $C_1 = \{1, 2, 3\}$, $C_2 = \{2, 3, 4\}$, $C_3 = \{1, 5, 6\}$, $C_4 = \{5, 7\}$, $C_5 = \{1, 8, 9\}$, $C_6 = \{8, 9, 10\}$ 。 G 的团树如图2-41b所示。弦图广泛应用于各个领域，如稀疏线性方程组的求解、进化树的研究、设备定位问题等。关于弦图有如下性质：

定理2-6 下列叙述是等价的：

1. G 是弦图；
2. G 有一个PEO；
3. G 可通过下面的递归构造原理得到：
 - a. 从任一个作为基图的团开始，团是一个弦图；
 - b. 对于弦图 H ，增加一个新的顶点并使得它与 H 的一个团子图相邻；
4. G 的每一个极小分离集导出 G 的一个团；
5. G 是树的子树族的交图；
6. G 的每一个诱导子图或者是一个团，或者包含两个不相邻的单纯顶点；
7. 含有两个或更多顶点的图 G 的每一个连通诱导子图至多含有 $n-1$ 个团。

针对弦图识别的串行算法利用各种被称为PEO格式的过程来生成PEO。

下面我们给出一些文献中提及的PEO格式：

1. 字典顺序宽度优先搜索（LBFS）；
2. 最大基数搜索（MCS）；

3. 分支中的极大元素 (MEC)；
4. 分支中的最大基数邻域 (MCC)。

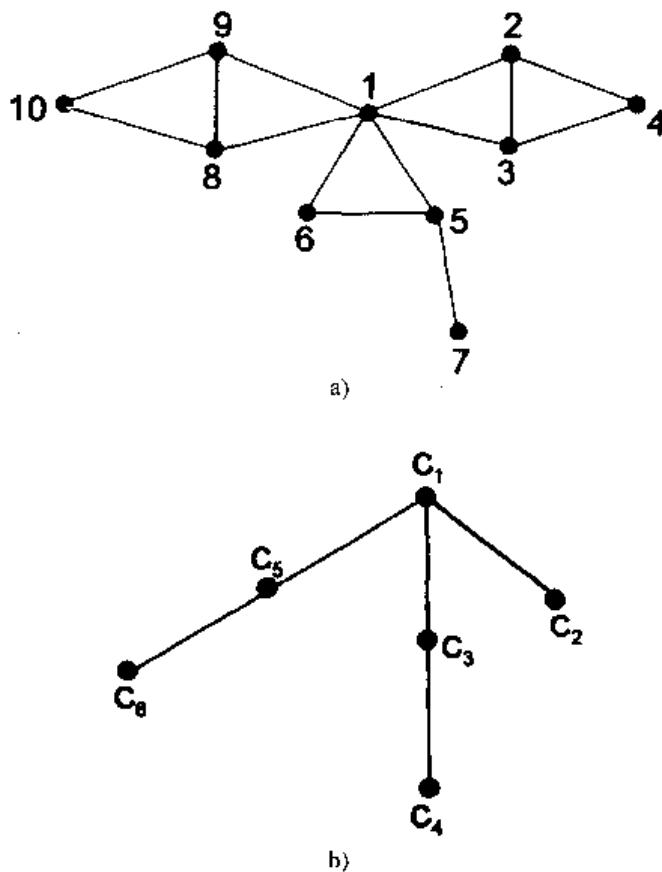


图 2-41

a) 弦图G b) G的团树

下面定理表明了识别弦图时PEO格式的重要性。

定理2-7 当且仅当由PEO格式LBFS、MCS、MEC、MCC产生的G的顶点序列是一个PEO时，图G是一个弦图。且测定图的弦所用的时间为 $O(m+n)$ 。

弦图的PEO已经成为串行算法的关键技术，而对于PEO的研究已经产生了许多重要的算法思想。然而并行算法的研究人员大都放弃了对PEO的使用。Chandrasekhar和Iyengar给出了利用极小团分离集识别弦图的NC并行算法，Edenbrandt也给出了类似的方法。而Naor和Schäffer则给出了利用PEO技术的NC并行算法，Dahlhaus和Karpinski也各自独立给出了同样的NC并行算法。这些算法都需要 $O(n^4)$ 个处理器。Klein的算法比较好，在CRCW PRAM模型中只需 $O(n)$ 个处理器，而执行时间为 $O(\log^2 n)$ 。96

路图 弦图的交图表示启发研究人员去研究更多类型的交图。Monma和Wei已经研究了几类起源于树的路族的交图。一条路如果把它看作是顶点的集合，则称为顶点路；两条顶点路如果有公共顶点，则称为有非空交集。一条路如果把它看作是边的集合，则称为边路；如果两条边路有公共边，则称它们有非空交集。在无向树中无向顶点路族的交图称为无向顶点路图或UV图。在Monma和Wei所用的术语中，树可以是有向的或者是有根有向的；图可以是有向的、无向的或有根有向的。基于此路图的可能类型有6种：

1. 有向顶点路图 (DV图)；

2. 无向顶点路图 (UV图)；
3. 有根有向顶点路图 (RDV图)；
4. 有向边路图 (DE图)；
5. 无向边路图 (UE图)；
6. 有根有向边路图 (RDE图)。

Monma和Wei证明这6种类型中的任一个图都可用适当类型的团树来表示，相应的团树的顶点用G的团标识。为了识别RDV图，Dictz给出了一个线性时间串行算法。为了识别UV图，Gavril给出了第一个关于交图表示的算法，执行时间为 $O(n^4)$ 。Schäffer给出了一个较快的识别UV图的算法，执行时间为 $O(mn)$ 。Xavier则给出了识别UV图的NC并行算法。

区间图 一个图如果是实轴上区间族的交图，则称为区间图。下面给出描述区间图特征的定理。

定理2-8 下面叙述是等价的：

- a. G是一个区间图；
- b. G是实轴上区间族的交图；
- c. G是一个弦图，且它的极大团可以被线性排序；对于G的每一个顶点x，包含x的极大团连续出现。

图2-42分别展示了一个区间图，它的区间表示以及极大团的线性序列。它的极大团分别是 $C_1 = \{a,b,c\}$, $C_2 = \{c,d,g\}$, $C_3 = \{c,e,g\}$, $C_4 = \{c,f,g\}$ 。Booth和Lucker证明了用串行算法识别一个区间图的时间为 $O(m+n)$ 。他们引入了称作PQ-树（见图2-43）的数据结构。PQ-树是一个非常有用的数据结构，它可用来表示区间图极大团的所有可能的线性定向。PQ-树是有根有向树，它有三种类型的节点：

1. 叶子：PQ-树的叶子是被操作的对象。在区间图识别中，叶子表示图的极大团。
2. P-节点：P-节点是内部节点，它至少有两个孩子并且用圈来表示。
3. Q-节点：Q-节点是内部节点，它至少有两个孩子并且用矩形来表示。

考虑两个PQ-树 T 和 T' ，如果存在一个由下面的变换组成的序列，这些变换使得 T 变成 T' ，则称 T 和 T' 等价，并记为 $T \equiv T'$ 。

1. 任意变换P-节点的孩子的顺序；
2. 颠倒Q-节点的孩子的顺序。

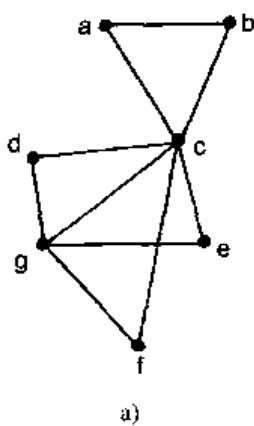


图 2-42

a) 区间图G

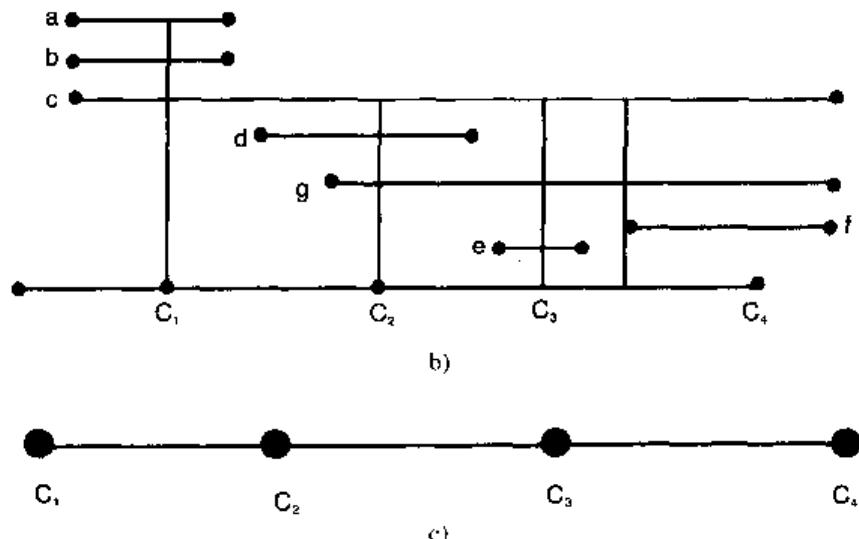


图2-42 (续)

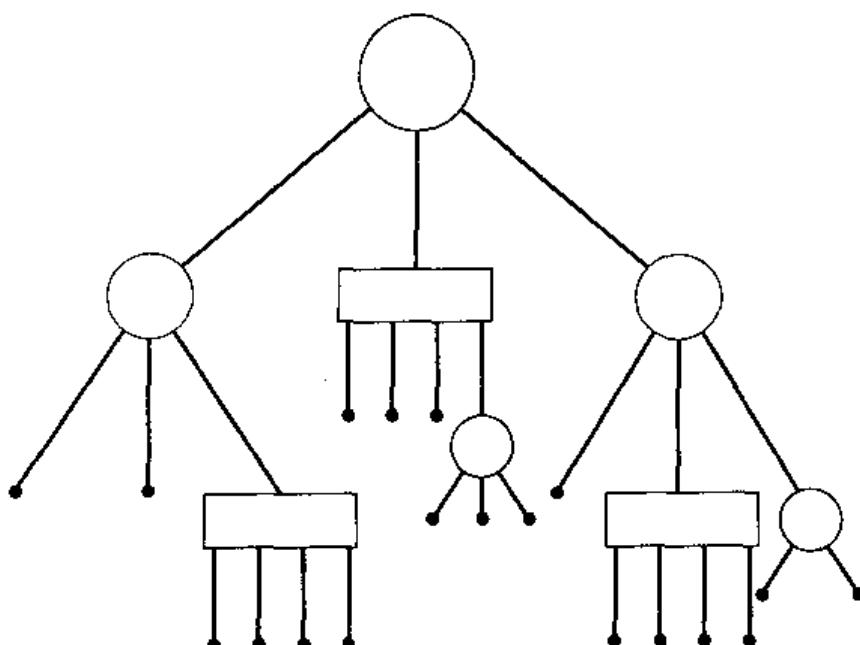
b) G 的区间表示 c) G 的极大团的线性序列

图2-43 PQ-树

显然， \equiv 表示等价关系。对于任何有序树 T ，其顶点 v 的边界 (frontier) 指叶子序列，这些叶子是顶点 v 的子代。在叶子序列中，叶子 a 先于叶子 b ，当且仅当 a 和 b 有公共祖先 c ，使得由 c 到 a 的路上 c 的孩子先于由 c 到 b 的路上 c 的孩子。换句话说，顶点 v 的边界是叶子序列，这些叶子被以 v 为根的子树遍历。如果 T 是一个有序树，则 $\text{FRONTIER}(T) = \text{FRONTIER}(\text{ROOT}(T))$ 。对于 PQ-树 T ，我们定义 $\text{CONSISTENT}(T) = \{\text{FRONTIER}(T'): T' \in T\}$ 。即树 T 的 CONSISTENT 是由 PQ-树 T 表示的所有可能的叶子序列的集合。很容易证明下面的定理。

定理2-9 如果 Z 和 Z' 是 PQ-树，当且仅当 $\text{CONSISTENT}(Z) = \text{CONSISTENT}(Z')$ 时， $Z \equiv Z'$ 。

假设 T 是 PQ-树，它的叶子节点是集合 C 的元素。 C' 是 C 的子集，Booth 和 Lueker 给出了一个 REDUCE 程序，使得 $\text{CONSISTENT}(\text{REDUCE}(T, C')) = \{\sigma \in \text{CONSISTENT}(T): C$ 的元素在 σ

中是连续的}。他们也证明了 n 个元素集 C_1, C_2, \dots, C_n 需要调用 REDUCE 程序 $O(p + \sum_{i=1}^n p_i)$ 次, 这里 $p = |\mathcal{C}|$, $p_i = |C_i|$ 。设 \mathcal{C} 是弦图 $G = (V, E)$ 的极大团的集合, $V = \{v_1, v_2, \dots, v_n\}$ 。假定包含 v_i 的极大团的集合用 C_i 表示。由定理 2-9 可知, 区间图的识别问题归结为 PQ-树的生成问题, 树的叶子取自 \mathcal{C} , 且对集合 C_1, C_2, \dots, C_n 需要调用 n 次 REDUCE 程序。极大团的数目至多是 n 。包含顶点 v 的团的数目至多是它的度数且图的所有顶点的度的总和是 $2m$ 。寻找弦图的极大团所用的串行时间是 $O(m+n)$, 因此, 识别区间图所用的串行时间是 $O(m+n)$ 。

Klein 和 Reif 设计了另一种运算——MREDUCE($\{\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_k\}$)。这个运算同时相当于所有的集合 \mathcal{C}_i , 可导出 PQ-树 T 。他们也给出了此问题的并行实现方法, 这里 \mathcal{C}_i 是两两不相交的。并行算法共需处理器数为 $O(n)$, 执行时间为 $O(\log n)$ 。

2.3.11 更多的交图

Scheinerman 刻画了交图类的特征。他定义了一种叫做顶点扩展的图运算。除了增加一个新的顶点 v 外, 在 v 点扩展的图与图 G 相同, 与此同时, v' 与 v 及 v 的每一个邻点都相邻。这里我们给出一个定理。
[100]

定理 2-10 F 是交图类, 当且仅当以下叙述是正确的:

- F 是遗传的, 即 F 中图的每一个诱导子图都在 F 中;
- 在顶点扩展运算下, F 是封闭的;
- 如果 G_1 和 G_2 都在 F 中, 则存在一个图 $G \in F$, 使得 G_1 和 G_2 是 G 的诱导子图。 G_1 和 G_2 的顶点集可以相交。

如果图 G 的每一个诱导子图 H 的最大独立集的顶点个数等于最小团覆盖的团的个数, 则称 G 是完美的 (perfect)。

一些交图如区间图、UV 图、RDV 图和弦图都是完美的, 本节我们将给出更多的交图。

圈图 圈的弦族的交图称为圈图。如何有效地识别圈图的问题多年来一直是一个难题, 但最后它几乎同时被三个独立的研究小组解决了。

奇偶图和距离遗传图 对于图 G 中的任一对顶点 a 和 b , 如果从 a 到 b 的所有导出路具有相同的长度, 则称 G 是距离遗传图。如果路的长度是偶数, 则称从 a 到 b 的导出路是偶长的 (even parity)。同样可以定义奇长的路 (odd parity)。在图 G 中, 对于任何两个顶点 a 和 b , 如果 a, b 间的所有导出路具有相同的奇偶性, 则称 G 是奇偶图。由定理 2-10, 上面定义的两个图都是交图。

圆弧图 圈的弧的集合的交图称为圆弧图。每一个区间图都是圆弧图。然而, 有些圆弧图甚至还不是弦图。

2.3.12 图的匹配问题

给定图 $G = (V, E)$, 边集 M 中的任何两条边都不与同一个顶点相关联, 则 M 称为匹配。如果 $(u, v) \in M$, 我们说 u 和 v 在 M 中是匹配的。如果一个匹配不包含在另一个匹配中, 则这个匹配称为极大匹配。边数最多的匹配称为最大匹配。图 $G = (V, E)$ 的匹配 M 如果匹配了图 G 中除了一个顶点外的所有顶点, 则称为完美匹配。当且仅当 $|M| = [n/2]$ 时, M 是完美匹配。当给图的边赋权时, 我们定义 M 中边的成本总和为匹配的成本 (Cost of the matching)。我们感兴趣的是

寻找最小（或最大）成本的完美匹配。匹配问题在许多方面都有应用。工人与工作相匹配，机器与部件相匹配，运动员与运动队相匹配等。

Wu和Manber已经引入了匹配问题的推广。设 $G = (V, E)$ 是一个加权图， G 中的路匹配是指具有不同端点的简单路的集合。匹配是路匹配的一个特例。在匹配中所有的路恰由一条边组成。

假定 P 是 G 的一个路匹配，如果 $|P| = \lfloor n/2 \rfloor$ ，则称路匹配 P 是完美的。如果 G 含有奇数个顶点，完美路匹配只剩一个顶点未匹配。如果 G 含有偶数个顶点， G 的所有顶点均在完美路匹配中所匹配。在 G 的路匹配中，如果任何两条路都是边不相交的，则称为边不相交的路匹配。Wu和Manber已经证明了下面这个引理。

引理 每一个无向图至少有一个边不相交的完美路匹配。

作为路匹配的一个应用，我们假定 G 是一个计算机网络模型，使得每一个顶点对应一台计算机，每一条边对应一条通讯链。每一条链上对应一个成本（如负载，运费，延迟）。进一步假定我们想在计算机间组织一场比赛，使得每一台计算机和另外一台配对共同完成某个竞赛。路表示与端点相应的计算机对。下面我们列出各种路匹配问题。

最小和路匹配 设 $G = (V, E)$ 是一个加权图， P 是 G 的一个完美路匹配。路的成本指路的所有边的权之和。 P 中路的所有成本之和称为 P 的成本和。求具有最小成本和的完美路匹配问题称为最小和路匹配问题。假定用 $O(n)$ 串行执行时间去寻找树的最小和路匹配。对于一般图，存在这样一个最小和路匹配，它含有每条至多有两条边的路。通过计算所有的最短路（至多两条边）以及求最小匹配可以得到最小和路匹配。

最小最大路匹配 完美路匹配 P 的最大成本是指 P 中路的最大成本。具有最小的最大成本的完美路匹配称为最小最大完美路匹配，这个问题也称为瓶颈问题（bottle-neck problem）。我们称边不相交的完美路匹配为 DP -匹配。Wu和Manber已经给出了一个求整数加权树的最小最大 DP -匹配的串行算法，其执行时间为 $O(n \log d \log w)$ ，这里 d 表示顶点的最大度， w 表示边的最大成本。Xavier(1995)给出了求树的最小最大 DP -匹配的并行算法。

2.3.13 图的中心

图 G 的两个顶点 u 和 v 的距离是指从 u 到 v 的路中的最小边数，记为 $d(u, v)$ 。如果 G 是加权图， $d(u, v)$ 表示从 u 到 v 的最短路的长度。顶点 v 的距离定义为从 v 到所有顶点的距离之和，记为 $d(v)$ ，即有

$$d(v) = \sum_{u \in V} d(v, u)$$

G 中具有最小距离的顶点称为 G 的重心（median）。每个树含有一个或两个重心，如果含有两个重心则这两个重心必相邻。

树 T 在顶点 v 的分枝指包含 v 且把 v 作为叶子节点的最大子树。在顶点 v 的所有分枝中，分枝所具有的最大边数称作分枝权重。记为 $B(v)$ 。也可以说，顶点 v 的分枝权重是 $T-v$ 的任一分枝中顶点的最大数目。如果 T 的某个顶点的分枝权重函数是最小的，这样的顶点称为 T 的形心（centroid）。对于任一树 T ，当且仅当一个顶点是形心时，它是树的重心。

求中心、重心和形心已经应用于许多关于设备定位（facility location）的问题中。例如，

假定图G的顶点表示一些村庄，边表示连接村庄的路，边的权表示两个村庄之间的距离。假定我们要挑选一个村庄来建造一所医院供所有村庄使用。如果我们选择G的中心作为医院的位置，我们可以减少病人移动的最大距离。如果图的顶点表示一个公司的各个分部，经理想找一个顶点作为年会的会议地点，那么感兴趣的会是图的重心，这样可以减少各个分部经理所付的总路费。Slater提议推广这些概念，使得设备定位在某个结构上而不是某个单一顶点上。他提议采用中心路结构。

中心路 从顶点 v 到一条路 P 的距离定义为

$$d(v, P) = \min\{d(v, u) / u \in P\}$$

路 P 的离心率定义为

$$e(P) = \max\{d(v, P) / v \in V\}$$

[103] 对于具有最小离心率的路 P ，如果具有相同离心率的其他路都长于 P ，则 P 称为中心路。路 P 的距离和定义为

$$d(P) = \sum_{v \in V} d(v, P)$$

具有最小距离和的路称为 G 的核心（core）。两个独立的研究小组已经给出了求树网络核心的NC并行算法。

2.3.14 控制理论

考虑图 $G = (V, E)$ ， D 是 G 的一个顶点集。如果每一个不在 D 中的顶点都与 D 中的某个元素相邻，则称 D 为图 G 的控制集。我们说 D 控制图 G 。考察图2-44，它的两个控制集为 $D_1 = \{1, 4, 7\}$ ， $D_2 = \{1, 5\}$ 。

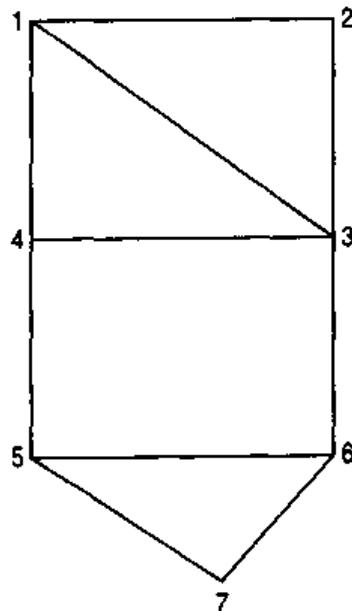


图2-44 图 G

这里有许多不同类型的应用。考虑一个公司，它有7个主管。如果主管 i 和 j 的人际关系较好，我们就在 i 与 j 之间连一条边。如图2-44所示，它表示公司7个主管之间的人际关系。

假定公司想组建一个委员会去执行一个计划，该计划需要公司7个主管的合作。而这个委员会的成员必须与不在委员会中的人有良好的人际关系，换句话说，对于不在委员会里的人，委员会里至少有一个人和他关系较好，因此这个委员会就是一个控制集。

对于一个控制集，如果它不包含其他控制集，则称之为极小控制集。含有最小顶点数的极小控制集称为最小控制集。最小控制集的顶点个数称为图的控制数（domination number）。我们注意到每一个极大独立集都是一个控制集。如果控制集是一个圈，则称之为控制圈（dominating cycle）。类似地，如果一个控制集是完全的（独立的），则称之为完全（独立）控制集。图 $G = (V, E)$ 的强控制数是指这样的一个最小整数 s ，它使得 G 的每一个含 s 个顶点的集合都是一个控制集。

D 是一个顶点集。如果 $V - D$ 中的每一个顶点 v 都与 D 中至少 k 个元素相邻，则称 D 为 k -控制集（这里 k 是正整数）。如果 (x, y) 是一条边，且满足 $\text{degree}(x) \geq \text{degree}(y)$ ，那么我们说 x 强控制 y ， y 弱控制 x 。强控制和弱控制的概念在文献中也出现过。不含孤立点的控制集称为全控制集。换句话说，对于 V 的一个子集 D ，如果 G 的每一个顶点与 D 中至少一个顶点相邻，则称 D 是全控制集。图 $G = (V, E)$ 的domatic划分是指 V 的这样一个划分，其中的每一个元素是 G 的控制集。图 G 的domatic划分中，元素的最大个数称为图 G 的domatic数。Cockayne和Hedetnemi (1977) 引入并研究了这个概念。Neeralagi (1988) 定义了图 $G = (V, E)$ 的奇（偶）控制集，它是顶点集 D ， $V - D$ 中的每一个顶点与 D 中的某一顶点之间的距离为奇（偶）数。

Slater, Bangar和Barkauskas研究了有效控制的概念。对于图 $G = (V, E)$ 的一个顶点集 D ，如果 $V - D$ 中的每一个顶点 u 恰好与 D 中的一个顶点相邻，则称 D 是有效控制集。Cockayne等人 (1988) 也研究了图的有效控制问题。

2.3.15 图论中的一些问题

这一节我们介绍图论中的一些问题，在文献中已研究过这些问题的算法。

1. **最大独立集问题：**求图 G 的最大独立集。
2. **最小染色问题：**求图的色数。
3. **最大团问题：**求给定图的最大团。
4. **最小团覆盖问题：**求给定图的最小团覆盖。
5. **同构问题：**给定两个图 G, G' ，判断 G 和 G' 是否同构。
6. **子图同构问题：**给定两个图 G, G' ，其中 G 的顶点数多于 G' 的顶点数，判断 G 是否有与 G' 同构的诱导子图。
7. **最大公共子图问题：**令 G 和 G' 是两个图，显然 G 和 G' 都有同构于 K_2 的子图。如果 G'' 是一个图，使得 G 与 G' 有同构于 G'' 的诱导子图，我们就说 G'' 是 G 和 G' 的公共子图。给定 G 和 G' ，求 G 和 G' 的包含最大顶点个数的公共子图。
8. **哈密顿问题：**判断一个图是否有哈密顿圈。
9. **控制问题：**给定一个图 G ，求极小控制集、全控制集、控制圈、连通控制集、控制团等。
10. **连通支问题：**给定一个图 G ，判断 G 是否连通，如果 G 是不连通的，求 G 的连通支。类似地，判断它是否是2-连通的，如果不是，求 G 的2-连通支。
11. **匹配问题：**给定一个图 G ，求最大匹配。
12. **路匹配问题：**求最小成本的路匹配问题是一类。在路匹配中，极小化路的最大权重是

另一类路匹配问题。

13. **最长路问题**: 在给定的图中, 求最长简单路。在树中, 最长路也称为直径。

14. **核心和双核心问题**: 加权图的核心是这样一条路, 它满足从这条路到所有顶点的距离之和最小。双核心是一对路, 它满足从其中任一条路到各顶点的距离之和最小。

15. **路覆盖问题**: 图 G 的路覆盖是指简单路的集合, 使得 G 的每一个顶点都在一条路上。求覆盖图的最小路数。

16. **识别问题**: 在设计有效算法过程中, 带有限制的图类是很重要的。这些带有限制的图类的结构性质可以被用来设计并行和串行算法。各种各样带限制的图类的识别问题在文献中受到相当的重视。

参考文献

106

- Bondy, J. A., and Murty, U. S. R. (1976) *Graph Theory with Applications*, North-Holland, New York.
- Joseph, J. J. (1972). *Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA.
- Johnson, D. S. (1985). The NP-completeness column: An ongoing guide. *J. Algorithms*, 6, 434–451.
- Klein, P. N. (1988). Efficient Parallel Algorithms for Planar, Chordal and Interval Graphs. TR 426 (Ph.D. Thesis) Laboratory for Computer Science, MIT, Cambridge, MA.
- Samy, A., Arumugam, G., Devasahayam, M. P., and Xavier, C. (1991). Algorithms for Intersection Graphs of Internally Disjoint Paths in Trees, *Proceedings of National Seminar on Theoretical Computer Science, IMSC*, Madras, India, Report 115, pp. 169–178.
- Berge, C. and Chvatal, C. eds. (1984). *Topics on Perfect Graphs*, *Annals of Discrete Mathematics*, 21.
- Sekharan, N. C. (1985). New Characterizations and Algorithmic Studies on Chordal Graphs and k-Trees, M.Sc. (Engg.) thesis, School of Automation, Indian Institute of Science, Bangalore.
- Chandrasekharan, N. and Iyengar, S. (1988). NC Algorithms for Recognizing Chordal Graphs and k-Trees, *IEEE Transactions on Computers*, 37(10).
- Joseph, J. A. (1992). *Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA.
- Kelly, D. (1985). Comparability graphs. In I. Rival, ed., *Graphs and Order*, pp. 3–40. D. Reidel, Dordrecht. (NATO ASI series C, v. 14.)
- Klein, P. N. (1988). Efficient Parallel Algorithms for Chordal graphs, *Proc. 29th IEEE Symposium on Foundation of Computer Section* pp. 150–161.
- Klein, P. N. (1988). Efficient Parallel Algorithms for Planar, Chordal and Interval Graphs, TR 426 (Ph.D. thesis), Laboratory for Computer Science, MIT, Cambridge, MA.
- Kozen, D., Vazirani, U. V., and Vazirani, V. V. (1985). NC Algorithms for Comparability Graphs, Interval Graphs and Unique Perfect Matchings. In *Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 496–503, New York and Berlin. (Springer Lecture Notes in Computer Science 206.)
- Moitra, A. and Iyengar, S. S. (1987). Parallel Algorithms for Some Computational Problems, *Advances in Computers*, 26, 93–153.
- Manacher, G. K. (1992) Chord Free Path Problems on Interval Graphs, *Computer Sci-*

- ence and Informatics, 22(2) 17–24.
- Mohring, R. H. (1984). Algorithmic Aspects of Comparability Graphs and Interval Graphs. In Ivan Rival, ed., *Graphs and Order*, pp. 41–101. D. Reidel, Dordrecht (Nato ASI series C, v. 147.)
- Xavier C. and Arumugam, G. (1994). Algorithms for Parity Path Problems in Some Classes of Graphs. *Computer Science and Informatics*, 24(4) 50–54.
- Xavier, C. (1995). Sequential and Parallel Algorithms for Some Graph Theoretic Problems (Ph.D. thesis). Madurai Kamaraj University.

第3章 并行算法设计环境

在串行计算中，一个处理器对存储在适当数据结构中的数据进行操作。这种数据变化的发生过程通常写成算法的形式。对并行计算来说，这种设计方式必须改变。在并行计算中，算法设计环境是完全不同的。本章介绍设计并行算法的各种环境。在设计串行算法的时候，如果时间复杂度为 $O(n)$ ，那么我们就认为这是一个好的算法。但在设计并行算法的时候，如果时间复杂度为 $O(n)$ ，那就不能认为是一个好的算法。时间复杂度为 $O(\log^k n)$ ，其中 k 为足够小的正整数，且用多项式数目的处理器的并行算法，就可以认为是好的算法。这些算法属于NC类。为了设计NC类的并行算法，我们在本章介绍特定的设计环境。如果想为一个新问题设计并行算法，就可以用本章介绍的任何一个设计环境来解决。即使不能用本章的任何设计环境来解决，我们也可以根据问题的性质设计新的环境。本章的目的仅仅是为了介绍一些著名的设计环境。

3.1 二叉树设计环境

二叉树的概念已经在第2章中介绍过。一个有 n 个叶节点的完全二叉树的高度为 $\lceil \log n \rceil$ 。我们可以在并行算法设计中利用这个性质。假定有 n 个数据，对应完全二叉树中的 n 个叶节点。我们可以并行地对它们进行处理，并在非叶节点得到部分结果。然后继续从底部向上走，并在 $\lceil \log n \rceil$ 时间内到达根节点。这些都会在本节给出。

n 个数求和 下面我们来看 n 个数求和的问题。在单个处理器上，此问题需要 $O(n)$ 的时间。

108

下面让我们来看看如何用更多的处理器来减少计算时间。假定 $n = 2^t$ 。当 $n = 8$ 时，把数据分成两组，即有

Group I: A(1) A(3) A(5) A(7)
Group II: A(2) A(4) A(6) A(8)

每组包含4个元素。因此，用 $n/2$ 个处理器（也就是4个处理器）。现在，相应的数据可以同时被这4个处理器相加，并分别存在 $A(1)$, $A(2)$, $A(3)$ 和 $A(4)$ 中。这些在图3-1中给出。利用下

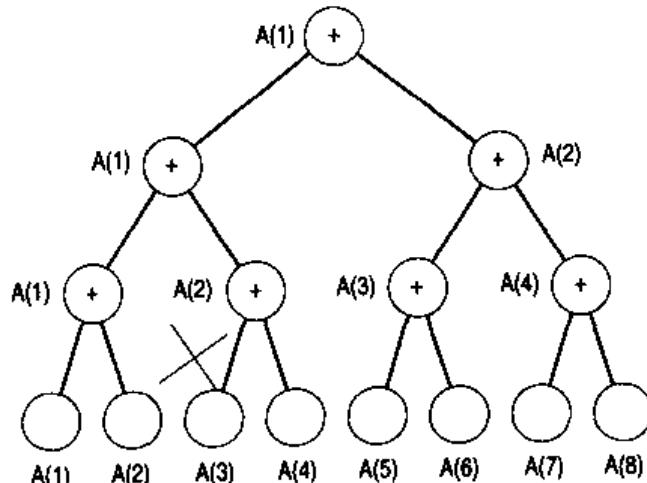


图3-1 $n = 8$ 个数的求和

面给出的数据，我们在图3-2a至图3-2d中演示了这个过程。

$A(1)$	$A(2)$	$A(3)$	$A(4)$	$A(5)$	$A(6)$	$A(7)$	$A(8)$
51	17	42	34	85	11	19	54

在第一阶段，进行如下操作：

$$A(1) \leftarrow A(1)+A(2) = 68$$

$$A(2) \leftarrow A(3)+A(4) = 76$$

$$A(3) \leftarrow A(5)+A(6) = 96$$

$$A(4) \leftarrow A(7)+A(8) = 73$$

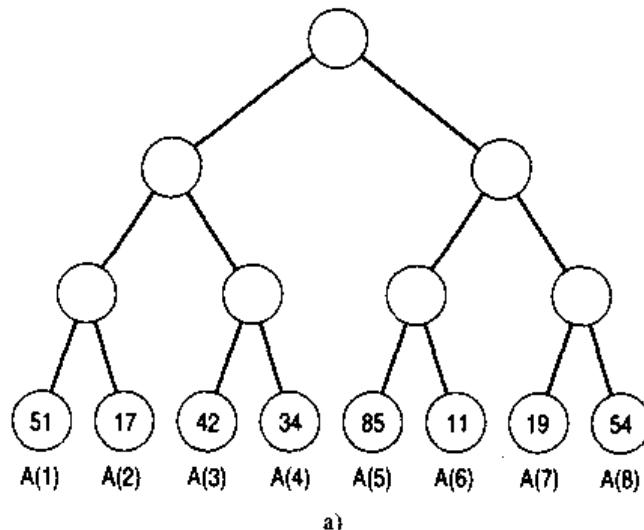
通常，对于 $i=1,\dots,4$ ，进行如下操作：

$$A(i) \leftarrow A(2i-1)+A(2i)$$

然后，在第二阶段，则进行：

$$A(1) \leftarrow A(1)+A(2) = 144$$

$$A(2) \leftarrow A(3)+A(4) = 169$$



109

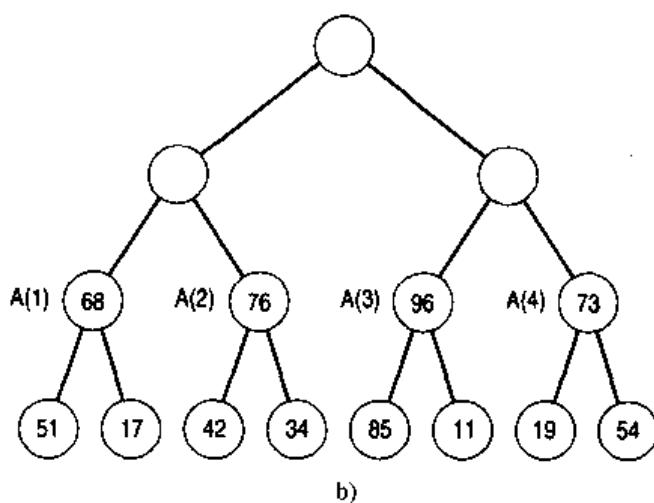


图 3-2

a) 初始值 b) 第一阶段

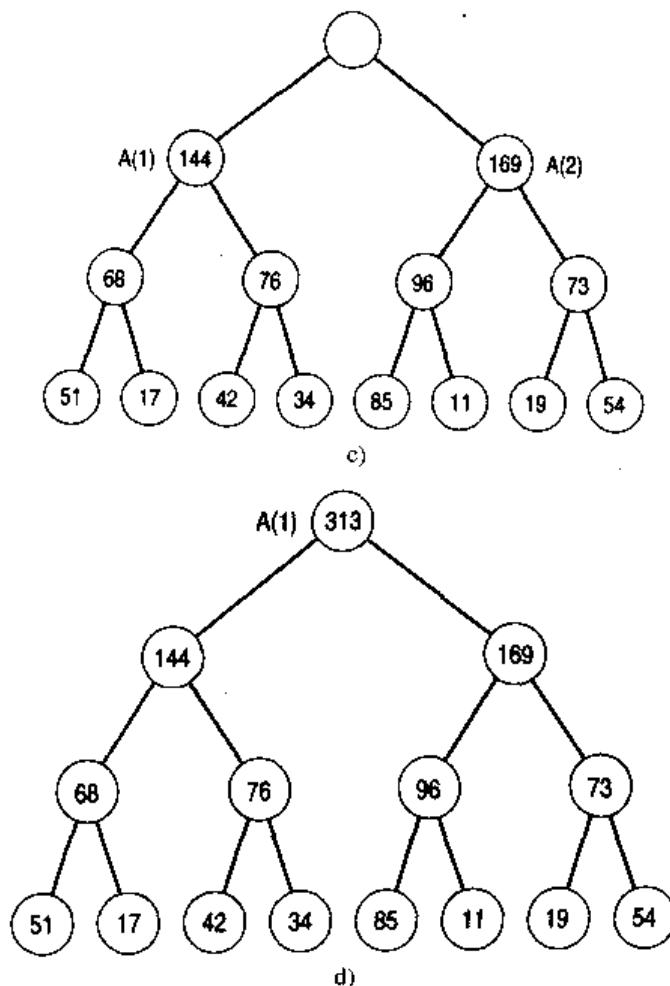


图3-2 (续)

c) 第一阶段 d) 第三阶段

也就是说，对 $i = 1, 2$ 计算：

$$A(i) \leftarrow A(2i-1) + A(2i)$$

在第三阶段，计算：

$$A(1) \leftarrow A(1) + A(2) = 313$$

这可以通过下面的并行算法实现：

算法SUM

输入：数组 $A(1:n)$, $n = 2^k$

输出：数组值的总和存储于 $A(1)$ 中

BEGIN

1. $p = n/2$
 2. While $p > 0$ do
 3. For $i = 1$ to p do in parallel
 4. $A(i) = A(2i-1) + A(2i)$
 5. End parallel
 6. $p = [p/2]$
 7. End while
- END

复杂度分析 p 是处理器的个数，开始的时候为 $n/2$ 个。 p 的值在每次while循环迭代时减半。因此，while循环重复 $\log n$ 次。另外，此算法既不用并发读，也不用并发写。这样，此算法可以在EREW PRAM模型中用 $O(n)$ 个处理器，并在 $O(\log n)$ 的时间内完成。

3.2 二倍增长

这是一个很有趣的设计方法，因为它只在设计并行算法才用到。如果有 n 个数据需要处理，那么开始的时候每个处理器覆盖一个数据，并等待合作。第一步时每个处理器中的数据为2，第二步变成4，第三步为8，以此类推。每一步都是把前一步的数据个数翻倍。

n 个数的求和过程可以看成二倍增长的例子。第一步，每个处理器求两个数的和：

$$\begin{aligned} P_1 &\text{求 } x_1 + x_2 \\ P_2 &\text{求 } x_3 + x_4 \\ \dots &\\ \dots &\\ P_{n/2} &\text{求 } x_{n-1} + x_n \end{aligned}$$

第二步：

$$\begin{aligned} P_1 &\text{求 } (x_1 + x_2) + (x_3 + x_4) \\ P_2 &\text{求 } (x_5 + x_6) + (x_7 + x_8) \\ \dots &\\ \dots &\\ P_{n/4} &\text{求 } (x_{n-3} + x_{n-2}) + (x_{n-1} + x_n) \end{aligned}$$

这里的每一步中每个处理器所处理的数据个数都是前一步的两倍。这就使得处理器可以在 $\log n$ 的时间内覆盖所有的 n 个数据。下面给出的“列表排序”(ranking a list)问题是另外一个这方面的例子。

3.3 指针跳转

很多典型问题都大量使用链接列表这种数据结构。如果想从表头到达表尾，在串行的情况下需要 $O(n)$ 的时间。而指针跳转技术却提供了一种可以在 $O(\log n)$ 时间内到达表尾的方法。我们在下面的例子中来说明：

列表排序问题 假定 $A(1:n)$ 为一个数组。这些数以某种顺序成为链接列表。LINK(i)表示 $A(i)$ 下面一个数的下标。LINK(3) = 7表示在链接列表中 $A(7)$ 是 $A(3)$ 的下一个。如果 $A(i)$ 是列表的最后一个数，那么LINK(i) = 0。变量HEAD包含第一个数的下标。一个数的排名定义为它和链表尾的距离。链表的最后一个数的排名为1，倒数第二个的排名为2，以此类推。链表的第一个数据排名为 n 。图3-3给出了有8个数的链表的例子。

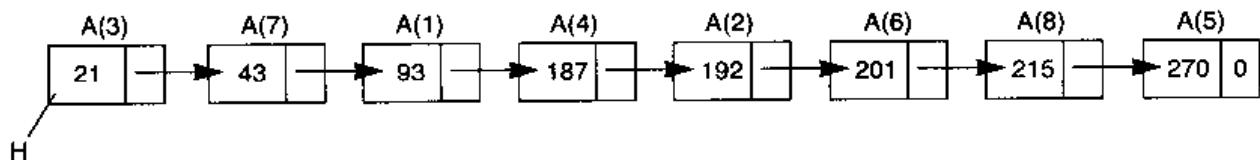


图3-3 8个数的链接列表

表3-1给出了图3-3中链接列表的数组表示。我们对找到每个数据的排名感兴趣。一个串行算法会从头到尾遍历整个列表，执行时间为 $O(n)$ 。其算法如下：

算法 Sequential List

输入： $A(1:n)$, $\text{LINK}(1:n)$, HEAD

输出： $\text{RANK}(1:n)$

BEGIN

$p = \text{HEAD}$

$r = n$

$\text{RANK}(p) = r$

Repeat

$p = \text{LINK}(p)$

$r = r - 1$

$\text{RANK}(p) = r$

until $\text{LINK}(p)$ is not equal to 0

END

表3-1 数组表示

NEAD = 3		
i	A	LINK
1	93	4
2	192	6
3	21	7
4	187	2
5	270	0
6	201	8
7	43	1
8	215	5

[13] 我们想为此问题设计一个并行算法。引入一个新变量 $\text{NEXT}(i)$ 。开始时 $\text{NEXT}(i) = \text{LINK}(i)$ 。也就是说， $\text{NEXT}(i)$ 开始的时候表示其右边邻居的下标。那么在下一步，会有：

$$\text{NEXT}(i) = \text{NEXT}(\text{NEXT}(i))$$

现在 $\text{NEXT}(i)$ 表示距离为2的数据下标。相似地，在下一阶段， $\text{NEXT}(i)$ 将表示距离为4的下标，即 $\text{NEXT}(i)$ 也是二倍增长的。因此，在第 $\lceil \log n \rceil$ 阶段，我们可以到达最后的数据，并完成最后的排名。完整的并行算法如下：

算法 List Ranking

输入： $A(1:n)$, $\text{LINK}(1:n)$, HEAD

输出： $\text{RANK}(1:n)$

BEGIN

1. For $i = 1$ to n do in parallel

$\text{RANK}(i) = 1$

$\text{NEXT}(i) = \text{LINK}(i)$

End Parallel

```

2. For  $k = 1$  to  $\lceil \log n \rceil$  do
  2a. For  $i = 1$  to  $n$  do in parallel
    If NEXT( $i$ ) is not zero
      RANK( $i$ ) = RANK( $i$ ) + RANK(NEXT( $i$ ))
      NEXT( $i$ ) = NEXT(NEXT( $i$ ))
    Endif
  End Parallel
Endfor
END

```

114

复杂度分析 这里第1步中， $O(n)$ 个处理器可以在 $O(1)$ 时间完成工作。第2步重复 $O(\log n)$ 次。第2a步需要 $O(n)$ 个处理器在 $O(1)$ 时间完成工作。因此，此并行算法利用 $O(n)$ 个处理器的总体复杂度为 $O(\log n)$ 。第1步可以用EREW PRAM模型实现，第2步也可以用EREW PRAM模型实现，因此所需要的计算模型为EREW PRAM。

下面用本节已经用过的例子来演示算法的工作过程。演示过程在图3-4a至3-4d中给出。

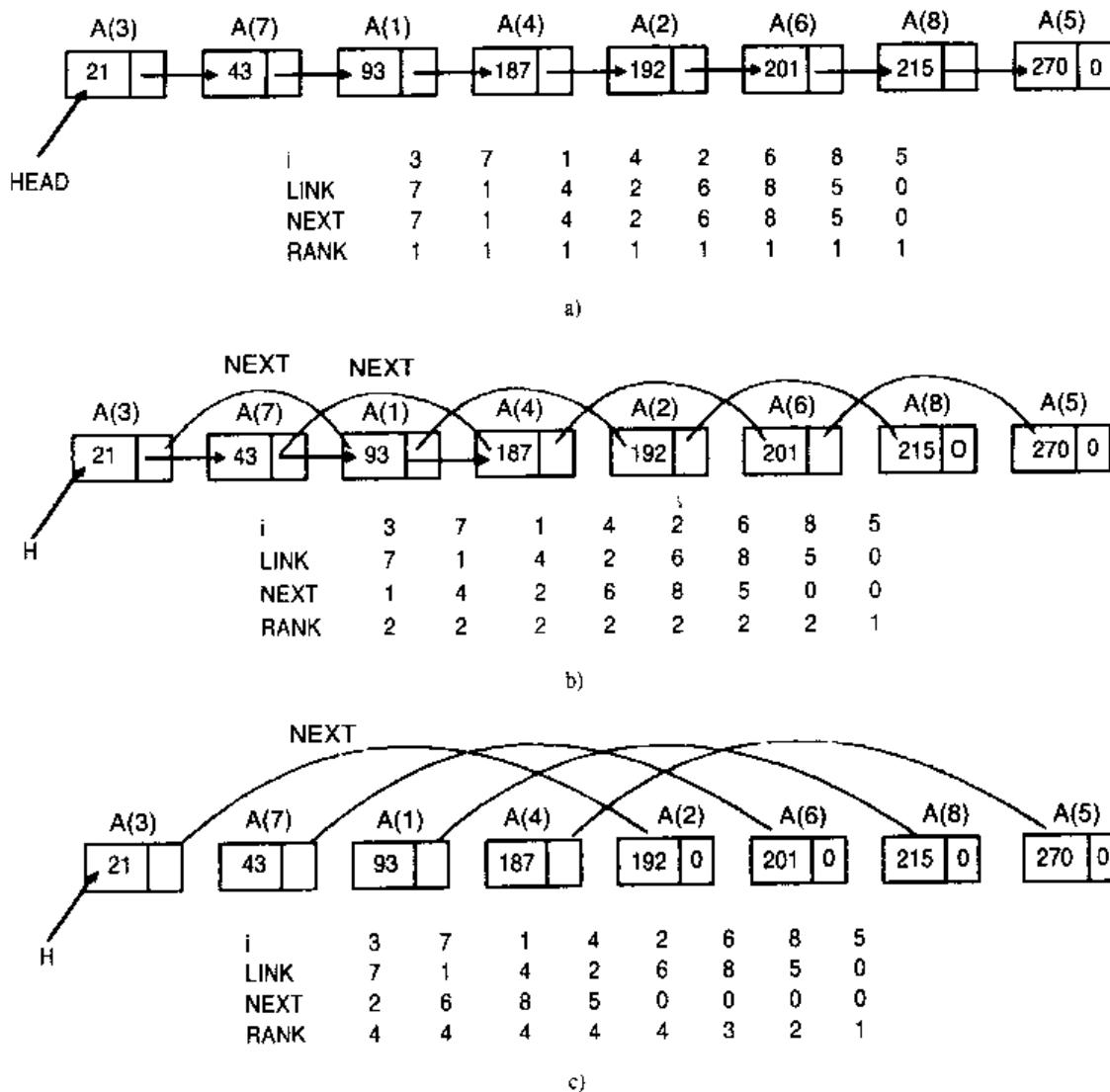


图 3-4

a) 初始阶段 b) 第一阶段 c) 第二阶段

115

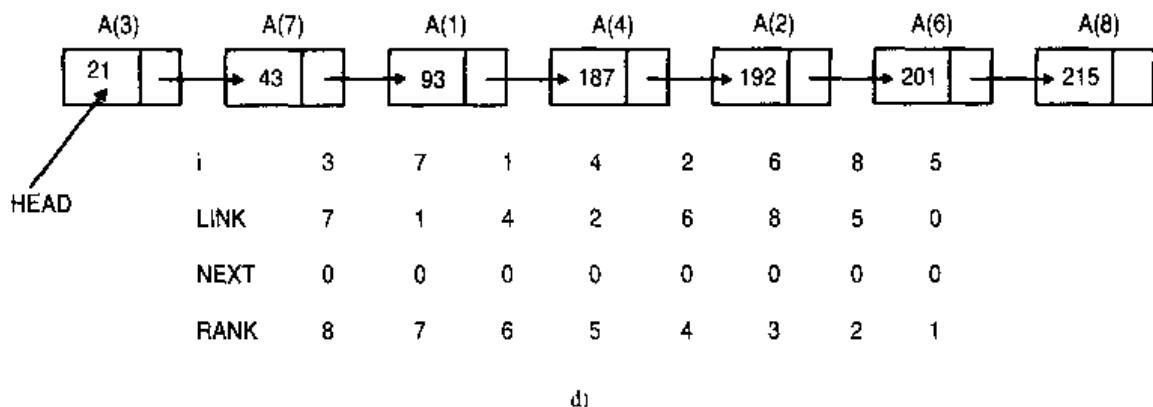


图3-4(续)

d) 第三阶段

3.4 分而治之

此方法中，一个问题被分成几个子问题。先求出这些子问题的解，然后进一步处理这些问题的解，从而得到整个问题的解。用求和算法来演示这个过程。如果 $A(1:n)$ 是一个数组，我们已经知道如何利用 $O(n)$ 个处理器在 $O(\log n)$ 时间内求得总和。在串行的情况下，用 $O(n)$ 的时间解决这个问题，此算法不是最优的。利用数据划分，我们设计了并行算法，可以用 $(n/\log n)$ 个处理器在 $O(\log n)$ 时间内完成。数组 A_1, A_2, \dots, A_n 被分成 $(n/\log n)$ 个组，每个包含 $\log n$ 个数组元素。为了简便，我们假定 $(n/\log n)$ 和 $\log n$ 是整数。设 $k = \log n$ 且 $r = n/\log n$ 。那么，有 $rk = n = 2^k$ 。下面给出分组：

组1: $A_1 A_2 A_3 \dots \dots \dots A_{\log n}$

组2: $A_{\log n+1} A_{\log n+2} \dots \dots \dots A_{2 \log n}$

组3: $A_{2 \log n+1} A_{2 \log n+2} \dots \dots \dots A_{3 \log n}$

.....

.....

组*i*: $A_{(i-1)\log n+1} A_{(i-1)\log n+2} \dots \dots \dots A_{i \log n}$

.....

组*r*: $A_{r(\log n+1)} A_{r(\log n+2)} \dots \dots \dots A_{r \log n}$

在每一组，有 $\log n$ 个数组元素，且共有 $r = n/\log n$ 个组。我们给每一组分配一个处理器，因此只需要 $(n/\log n)$ 个处理器。每个处理器得到 $\log n$ 个数组元素。现在，每个得到 $\log n$ 个数组元素的处理器 P_i 会在 $O(\log n)$ 时间内串行完成第*i*组数组元素的求和，并存入变量 B_i 。现在只有元素 $B_1, B_2, \dots, B_{n/\log n}$ 。利用3.1节中的算法，可以利用 $O(n/\log n)$ 个处理器在 $O(\log(n/\log n))$ 时间内完成求和。已知 $O(\log(n/\log n)) = O(\log n - \log \log n) = O(\log n)$ 。此算法利用 $O(n/\log n)$ 个处理器在 $O(\log n)$ 时间内完成求和，因此它是解决求和问题的最优算法。此算法正式给出如下：

算法Optimal-Sum

输入：数组 $A(1:n)$

输出：数组元素的和存于变量SUM中

1. For $i = 1$ to $n/\log n$ do in parallel
2. Using the sequential method find the sum of $A_{(i-1)\log n+1}, A_{(i-1)\log n+2}, \dots, A_{i\log n}$ and store the result in variable B_i
3. End parallel
4. Find the sum of $B_1, B_2, \dots, B_{n/\log n}$ and store in the variable SUM
5. End optimal sum

复杂度分析 上面的算法中，第2步利用串行算法在 $O(\log n)$ 时间内完成。所以，第1-3步可以用 $O(n/\log n)$ 个处理器在 $O(\log n)$ 时间内完成。第4步可以用第3.1节的并行算法在 $O(\log(n/\log n))$ 时间内在 EREW PRAM 模型下完成。因此，上述算法在 EREW PRAM 模型下利用 $O(n/\log n)$ 个处理器在 $O(\log n)$ 时间内完成。该算法是最优的。刚才解释的分而治之算法也可以用来求数组 $A(1:n)$ 中的最小/最大值。和上面给出的算法非常相似，可以在 EREW PRAM 模型下利用 $O(n/\log n)$ 个处理器在 $O(\log n)$ 时间内解决这个问题。

3.5 划分

在分而治之方法里面，我们把问题分成子问题，然后并发求解子问题，最后合并这些子问题的结果得到最后的结果。也就是，分而治之包含如下几步：

第一步：把问题分成子问题 P_1, P_2, \dots, P_r ，并发求解，得到解 S_1, S_2, \dots, S_r 。[117]

第二步：合并这些子问题的结果得到最后的结果。

在划分技术中，我们更多关注把问题分成子问题的过程。仔细地把问题分成子问题 P_1, P_2, \dots, P_r ，使得求解这些子问题的同时，原问题的解已经得到。换句话说，划分子问题的方法使得分而治之的第二步是不必要的。这可以通过合并的例子来说明。

合并问题 设 $A(1:n)$ 和 $B(1:n)$ 是两个排过序的数组，也就是 $A_1 < A_2 < \dots < A_n$ 且 $B_1 < B_2 < \dots < B_n$ 。我们想把这两个数组合并成一个数组 $C(1:2n)$ ，使得 $C_1 < C_2 < \dots < C_{2n}$ 。如果 $A = (1, 5, 9, 12)$ 且 $B = (2, 3, 15, 19)$ ，合并这两个数组可得到 $C = (1, 2, 3, 5, 9, 12, 15, 19)$ 。为了方便，设 $n = 2^k$ 和 $n/\log n \approx r$ ，且进一步假设 k 和 r 都是整数。我们知道下面的串行算法可以在 $O(n)$ 的时间内合并两个数组。

算法 Sequential Merge

输入：两个有序数组 $A(1:n)$ 和 $B(1:n)$

输出： $C(1:2n)$ ，且 $C_1 < C_2 < \dots < C_{2n}$

1. Let $A_{n+1} = B_{n+1} = \infty$
2. $i = 1; j = 1; k = 1$
3. While $k \leq 2n$ do
4. If $A_i < B_j$, then

$C_k = A_i$

$i = i + 1$

else

$C_k = B_j$

$j = j + 1$

```

    end if
5.  $k = k+1$ 
6. end while
7. End sequential merge

```

方法是把排序数组 A 分成 $r = n/\log n$ 个组，每组有 $k = \log n$ 个元素，如下：

组1:	$A_1 A_2 A_3 \dots \dots \dots A_k$
组2:	$A_{k+1} A_{k+2} A_{k+3} \dots \dots \dots A_{2k}$
.....	
组 i :	$A_{(i-1)k+1} A_{(i-1)k+2} \dots \dots \dots A_{ik}$
.....	
组 r :	$A_{(r-1)k+1} A_{(r-1)k+2} \dots \dots \dots A_{rk}$

118

现在我们要找到 r 个整数 $j(1), j(2), \dots, j(r)$ ，使得

$j(1)$ 是最大的下标， $A_k > B_{j(1)}$
.....
.....
$j(i)$ 是最大的下标， $A_{ik} > B_{j(i)}$
.....
.....
$j(r)$ 是最大的下标， $A_{rk} > B_{j(r)}$

这样就把数组 $B(1:n)$ 分成 r 个组，如下：

组1:	$B_1 B_2 \dots \dots \dots B_{j(1)}$
组2:	$B_{j(1)+1} B_{j(1)+2} \dots \dots \dots B_{j(2)}$
.....	
组 i :	$B_{j(i-1)+1} B_{j(i-1)+2} \dots \dots \dots B_{j(i)}$
.....	
组 r :	$B_{j(r-1)+1} B_{j(r-1)+2} \dots \dots \dots B_{j(r)}$

现在我们可以有如下结论： A 数组的第1组的每个成员都小于或等于 B 数组的第2, 3, … 组的每个成员。如果 A 数组的第1组和 B 数组的第1组单独合并，可以确保数组元素已经到达最后得到的排序数组 $C(1:n)$ 中的最后位置。这些对其他分组也成立。因此，我们把处理器 $i (1 \leq i \leq r)$ 分配给 A 数组的第 i 组和 B 数组的第 i 组，处理器 i 串行合并 A 数组的第 i 组和 B 数组的第 i 组。

现在正式给出这一并行算法：

算法Merge

输入：两个有序数组 $A(1:n)$ 和 $B(1:n)$ ，设 $n = 2^t$, $\log n = k$

输出：合并的数组 $C(1:2n)$

1. for $i = 1$ to r do in parallel
2. Using binary search find the index

$$j(i) = \text{Max } \{t, \text{ such that } A_{ik} > B_t\}$$

[119]

3. Using sequential method merge the two arrays

$$A((i-1)k + 1 : ik) \text{ and } B(j(i-1) + 1 : j(i))$$

4. End parallel

5. end Merge

为了演示上面的算法，考虑数组 $A = (1, 5, 15, 18, 19, 21, 23, 24, 27, 29, 30, 31, 32, 37, 42, 49)$ 和 $B = (2, 3, 4, 13, 15, 19, 20, 22, 28, 29, 38, 41, 42, 43, 48, 49)$ 。这里 $n = 16 = 2^4$, $\log n = k = 4$, $r = n/\log n = 4$ 。把 A 分成4个组($r = 4$)。

A	1, 5, 15, 18	19, 21, 23, 24	27, 29, 30, 31	32, 37, 42, 45
组	第1组	第2组	第3组	第4组

$j(i)$ 的值分别是 $j(1) = 5$, $j(2) = 8$, $j(3) = 10$ 和 $j(4) = 15$ 。把 B 数组的分组情况附在上表之后一并给出如下：

组	第1组	第2组	第3组	第4组
A	1, 5, 15, 18	19, 21, 23, 24	27, 29, 30, 31	32, 37, 42, 45
B	2, 3, 4, 13, 15	19, 20, 22	28, 29	38, 41, 42, 43, 48, 49

当我们按组合并数组 A 和 B 的子数组时，有：

分组1得到： $C(1:9) = (1, 2, 3, 4, 5, 13, 15, 15, 18)$

分组2得到： $C(10:16) = (19, 19, 20, 21, 22, 23, 24)$

分组3得到： $C(17:22) = (27, 28, 29, 29, 30, 31)$

分组4得到： $C(23:32) = (32, 37, 38, 41, 42, 42, 43, 48, 49, 49)$

因此，得到最后的数组：

$$C(1:32) = (1, 2, 3, 4, 5, 13, 15, 15, 18, 19, 19, 20, 21, 22, 23, 24, 27, 28, 29, 29, 30, 31, 32, 37, 38, 41, 42, 42, 43, 48, 49, 49)$$

复杂度分析 第2步可以用二分搜索在 $O(\log n)$ 时间内完成。第3步需要的时间取决于数组 A 和 B 的子数组。 $A((i-1)k+1:ik)$ 的大小为 k ; $B(j(i-1)+1:j(i))$ 的大小未知。如果其大小也是小于或等于 k ，那么算法的第3步可以在处理器 i 上用 $O(\log n)$ 时间串行完成。如果 $B(j(i-1)+1:j(i))$ 的大小大于 k ，我们必须先对数组 B 递归重复相同的划分技术，然后是 A 。因此，第3步可以在处理器 i 上用 $O(\log n)$ 时间串行完成。这样，此算法可以在EREW PRAM模型上用 $O(n/\log n)$ 个处理器在 $O(\log n)$ 时间内完成。

[120]

3.6 小结

本章中介绍了一些重要的设计环境。为了演示这些设计环境，我们设计了一些简单的并行算法。二叉树设计环境借用了二叉树的数据结构。数据进行适当的分布和处理可以获得高效的并行算法。另外一个指针跳转的设计环境在第3.2节中给出，并用适当的例子进行了完整的解释。我们还用简单有趣的例子对分而治之和划分技术进行了解释。

参考文献

- Akl, S. G. (1989) *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, NJ.
- Brassard, G and Bratley, P. (1988) *Algorithmics: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ.
- Dekel, E., Nassimi D., and Sahni, S. (1987) Parallel Matrix and Graph Algorithms, *Siam. J. Computing*, 10, 657–675.
- Joseph, J. A. (1992) *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA.
- Moitra, A. and Iyengar, S. S. (1987) Parallel Algorithms for Some Computational Problems in Advances in Computers, Academic Press, New York, pp. 93–153.
- Quinn, M. J. (1987) *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York.

习题

3.1 给定二维平面上 n 个点的集合 $S = \{p_1, p_2, p_3, \dots, p_n\}$ ， S 的平面凸包是能够包含这 n 个点的最小的凸多边形。请设计一个分而治之的并行算法来求把 n 个点连接起来得到凸包边界的序列。

121 3.2 考虑下面给出的求部分和的递归方法：

1. 给定数组 x_1, x_2, \dots, x_n ；
2. 求 $x_1, x_2, \dots, x_{n/2}$ 的部分和，并记作 $z_1, z_2, \dots, z_{n/2}$ 。其中 $z_i = x_1 + x_2 + \dots + x_i$ ，且 $i > n/2+1$ ；
3. 对于 $i < n/2$ ， $s_i = z_i$ ；
4. 对于 $i > n/2$ ， $s_i = z_i + z_n/2$ 。

利用上面的技术，设计一个非递归算法求数组的部分和。

3.3 设 A 为一个 $n \times n$ 的下三角矩阵，且 n 是2的幂次方。假定 A 是非奇异的。把 A 分成如下四块：

$$A = \begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix}$$

在上面的矩阵中， A_{11}, A_{21} 和 A_{22} 都是 $(n/2 \times n/2)$ 的矩阵。那么整个矩阵的逆如下：

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} & 0 \\ -A_{22}^{-1} A_{21} A_{11}^{-1} & A_{22}^{-1} \end{bmatrix}$$

利用上面的方法，设计一个分而治之的算法求下三角矩阵的逆。

3.4 A 是一个大小为 n 的布尔数组。我们要求最小的下标 i ，且 $A(i)$ 为真。

- a) 设计一个CREW PRAM模型下的并行算法;
b) 设计一个CRCW PRAM模型下的并行算法。
- 3.5 设 $X(1:n)$ 为大小为 n 的数组。数组元素 x_i 的左匹配是指数组元素 x_k （如果存在），其中 k 是最大的下标，且 $1 \leq k < i$, $x_k < x_i$ 。相似地，可以定义数组元素 x_i 的右匹配。请设计一个并行算法求数组每个元素的左右匹配。
- 3.6 请用划分技术设计一个算法求数组 $A(1:n)$ 的最小元素，并验证用 $O(n/\log n)$ 个处理器的时间复杂度为 $O(\log n)$ 。
- 3.7 对于 $n = 16$ ，画出在习题3.6中所设计的算法的二叉树表示。

[122]

第4章 简单并行算法

本章我们研究一些简单的问题，并演示怎样设计并行算法来解决这些问题。

4.1 向量内积

设 $a = (a_1, a_2, \dots, a_n)$ 和 $b = (b_1, b_2, \dots, b_n)$ 是两个向量，那么这两个向量的内积为 $a \cdot b = a_1b_1 + a_2b_2 + \dots + a_nb_n$ ，我们关注的是设计能够求出两个向量的内积的并行算法。这与求数组元素的和的算法类似。两个数组 $a[1:n]$ 和 $b[1:n]$ 都存在共享内存里。 $a \cdot b$ 最后的结果存在另外一个变量 c_1 中。

下面的算法描述了这种方法：

算法 Scalar Product

输入：数组 $a[1:n]$ 和 $b[1:n]$

输出：内积的值存于变量 c_1 中

BEGIN

1. For $i = 1$ to n do in parallel
2. $c_i = a_i * b_i$
3. End parallel
4. $p = n/2$
5. While $p > 0$ do
6. For $i = 1$ to p do in parallel
7. $c_i = c_i + c_{i+p}$
8. End parallel
9. $p = [p/2]$
10. End while

END

123

在此算法中，第1~3步需 $O(n)$ 个处理器，并在 $O(1)$ 时间内完成。第4~10步就是数组的求和过程。因此，该算法可以在EREW PRAM模型上用 $O(n)$ 个处理器在 $O(\log n)$ 时间内完成。结果将存在变量 c_1 中。利用第3章3.4节介绍的分而治之技术，此问题可以利用 $O(n/\log n)$ 个处理器在 $O(\log n)$ 时间内完成。这个问题作为习题留给读者。在下一节，我们演示一个更为流行的矩阵乘法并行算法的设计过程。

4.2 矩阵乘法

矩阵乘法在一些重要的问题中扮演着很重要的角色。如果 A 是 $m \times n$ 的矩阵， B 是 $n \times p$ 的矩阵，那么可以计算它们的乘积 $C = AB$ ，且其阶为 $m \times p$ 。矩阵 C 的第 i 行和第 j 列的元素为 $C(i, j)$ ，它通过矩阵 A 的第 i 行和矩阵 B 的第 j 列的内积得到。即：

$$C(i, j) = (a_{i1}, a_{i2}, \dots, a_{in}) \begin{pmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ b_{nj} \end{pmatrix}$$

$$C(i, j) = \sum_{k=1}^n a_{ik} b_{kj}$$

算法如下所示：

算法Matrix-Multiply

输入：矩阵A和B

输出：乘积矩阵C

BEGIN

1. For $i = 1$ to m do in parallel
2. For $j = 1$ to p do in parallel
3. Evaluate $C(i, j)$
4. End Parallel
5. End Parallel

END

这里对第3步必须进行进一步的解释。 $C(i, j)$ 的计算是两个向量的内积，它由下面的语句 [24] 段来完成。每个处理器都用一个本地临时变量 $T(1:n)$ 保存 $A(i, k)*B(k, j)$ 的值。

- 3.1 For $k = 1$ to n do in Parallel
- 3.2 $T(k) = A(i, k)*B(k, j)$
- 3.3 End parallel
- 3.4 $p = n/2$
- 3.5 While $p > 0$ do
 - 3.6 For $k = 1$ to p do in Parallel
 - 3.7 $T(k) = T(2k-1) + T(2k)$
 - 3.8 End Parallel
 - 3.9 $p = [p/2]$
- 3.10 End while
- 3.11 $C(i, j) = T(1)$

复杂度分析 上面的语句段包含两个部分。在第一部分先对 $A(i, k)*B(k, j)$ 进行计算，然后在第二部分把这些值相加得到 $C(i, j)$ 的值。本地临时变量 $T(1:n)$ 被用来保存相乘和相加的结果。在这一部分，第3.1 ~ 3.3步需用 $O(n)$ 个处理器，需要 $O(1)$ 的时间。第3.4 ~ 3.11步和SUM算法相同，因此需用 $O(n)$ 个处理器，需要 $O(\log n)$ 的时间。第3步被放在两层嵌套的并行循环里面，因此用 $O(mnp)$ 个处理器该算法的时间复杂度为 $O(\log n)$ 。特别是，当A和B为方阵时，用 $O(n^3)$ 个处理器算法的时间复杂度为 $O(\log n)$ 。在计算 $C(i, 1), C(i, 2), \dots, C(i, n)$ 时需要 $A(i, j)$ 。由于不同组的处理器并发计算 $C(i, 1), C(i, 2), \dots, C(i, n)$ ，因此多个处理器会同时读取 $A(i, j)$ 。所以上述算法需要CREW PRAM模型。如果采用第3章3.4节的分而治之技术，可以利用 $O(n^3/\log n)$ 个处

理器在 $O(\log n)$ 的时间内完成。下面一节给出的例子将解决一个更有趣的问题。

4.3 部分和

设 $A(1:n)$ 为 n 个数的数组。此数组的部分和定义为：

$$PS(i) = \sum_{j=1}^i A(j) \quad (1 \leq i \leq n)$$

我们用二叉树设计环境来计算部分和 $PS(i)$ ($1 \leq i \leq n$)。利用此算法求数组的和，只能得到 $PS(n)$ 。
[125] 我们需要修改此算法得到部分和。图4-1演示了对 $n = 8$ 的求和算法。 $S(i, j)$ 表示从左到右数第 i -层的第 j 个节点。开始，设定 $S(0, i) = A(i)$ ；然后，通过增加子节点得到 $S(1, i)$ 。在第一个时间片，得到 $S(1,1), S(1,2), S(1,3)$ 和 $S(1,4)$ 。在第二个时间片，我们得到 $S(2,1)$ 和 $S(2,2)$ 。在第三个时间片，我们得到 $S(3,1)$ 。整个过程从下向上在 $O(\log n)$ 时间内完成，其中 n 是数组的长度。 S 值的求解过程在图4-2a至图4-2d中演示。现在， $S(3,1)$ 是整个数组的元素的和。

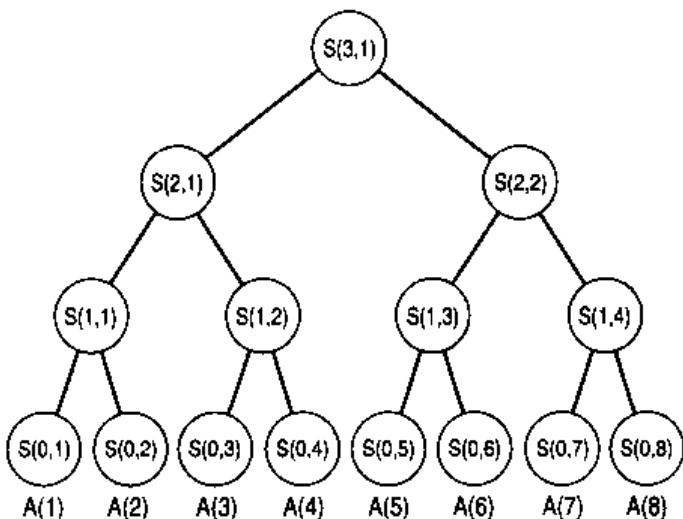


图4-1 树中的 $S(i, j)$ 值

$S(2,1)$ 是前四个元素的和。为了得到部分和，我们引入另一个变量 PS 。最后的部分和将是 $PS(0,1), PS(0,2), \dots, PS(0,n)$ 。这些结果可以通过遍历二叉树从根到叶的节点得到。从 S 的值得到 PS 值的过程如下：

从 $PS(3,1) = S(3,1)$ 开始，得到下一层：

$$PS(i, j) = \begin{cases} S(i, j), & j = 1 \\ PS(i+1, [j/2]) + S(i, j), & j \text{ 为奇数且 } j > 1 \\ PS(i+1, j/2), & j \text{ 为偶数} \end{cases}$$

我们用有8个元素的数组演示此过程如下：

i	1	2	3	4	5	6	7	8
$A(i)$	23	38	40	73	91	39	48	63

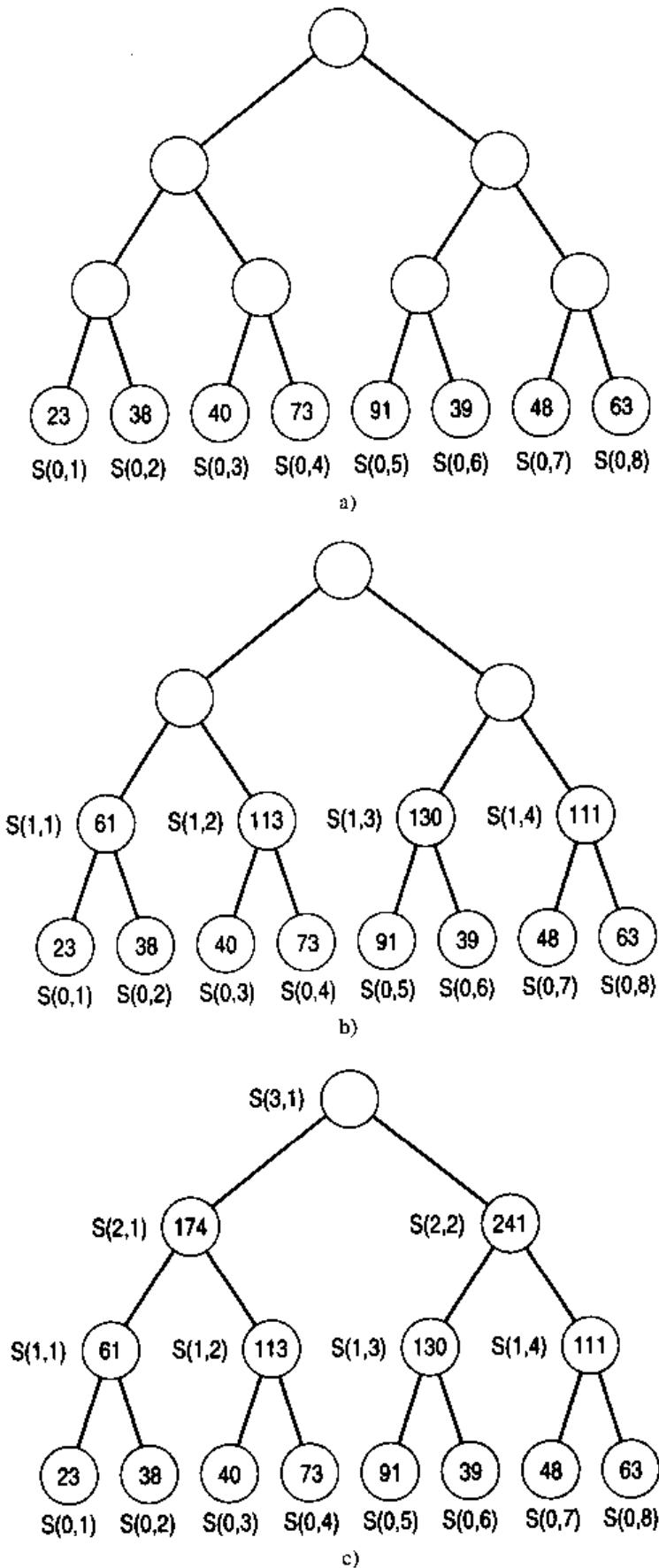


图 4-2

a) 初始阶段 b) 第一阶段 c) 第二阶段

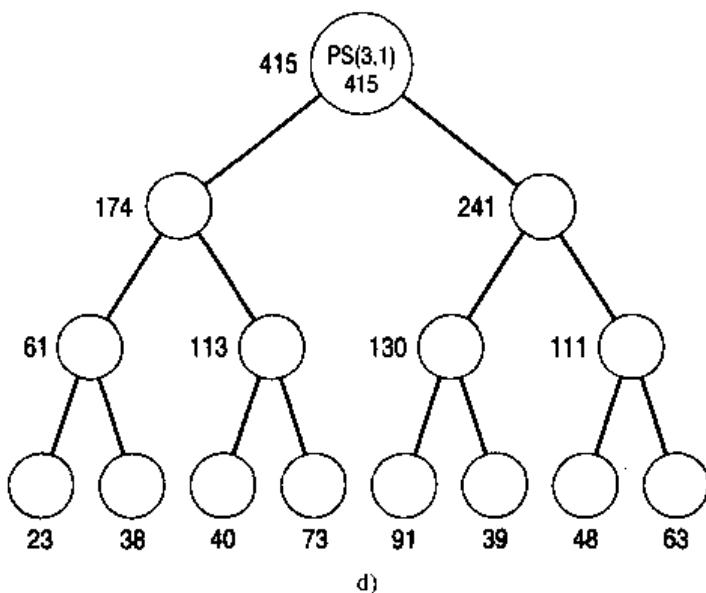


图4-2 (续)

d) 第三阶段

$S(i,j)$ 的值可以通过从下至上遍历二叉树得到。这在图4-2d中给出。在计算出 $S(i,j)$ 之后，根据上面的公式通过从上至下遍历二叉树得到 $PS(i,j)$ 。这在图4-3a到图4-3d中给出。 S 的值在每个节点的外部给出， PS 的值在每个节点的内部给出。在 $S(i,j)$ 的计算过程中，很容易发现可以并行计算 $PS(i,1), PS(i,2), \dots$ 。另外，可以发现 $PS(i,j)$ 被并发地用在计算 $PS(i-1,2j)$ 和 $PS(i-1,2j+1)$ 的过程中。因此需要CREW PRAM模型。算法如下：

127 算法Partial-Sum

输入：数组 $A(1:n)$

输出：部分和 $PS(0, 1:n)$

BEGIN (i denotes level number and p the number of nodes at that level.)

0. Copy $A(i)$ to $S(0, i)$ for $1 \leq i \leq n$ in parallel

1. $p = n$

2. For $i = 1$ to $\lceil \log n \rceil$ do

3. $p = \lceil p/2 \rceil$

4. For $j = 1$ to p do in parallel

5. $S(i,j) = S(i-2, 2j) + S(i-1, 2j-1)$

6. End Parallel

7. End for

8. $PS(\lceil \log n \rceil, 1) = S(\lceil \log n \rceil, 1)$

9. $h = 1$

10. For $i = \lceil \log n \rceil - 1$ down to 0 do

11. $p = 2p$

12. For $j = 1$ to p do in parallel

13. Case

$j = 1$: $PS(i,j) = S(i,j)$

$j = \text{even}$: $PS(i,j) = PS(i+1, j/2)$

```

Else:  $PS(i,j) = PS(i+1, [j/2]) + S(i,j)$  [129]
Endcase
14. End parallel
15. End for
END

```

130

复杂度分析 利用 $O(n)$ 个处理器，第0步需要 $O(1)$ 时间；利用 $O(n)$ 个处理器，第2到第7步需要 $O(\log n)$ 时间；利用 $O(n)$ 个处理器，第10步到第15步也需要 $O(\log n)$ 时间。因此，利用 $O(n)$ 个处理器，该算法的复杂度为 $O(\log n)$ 。从 $PS(i,j)$ 的公式可以看出， $PS(i,j)$ 被并行应用于 $PS(i-1, 2j-1)$ 和 $PS(i-1, 2j)$ 的计算过程中。因此需要并发读模型，但是整个程序不需要并发写，可以用CREW PRAM模型实现该算法。

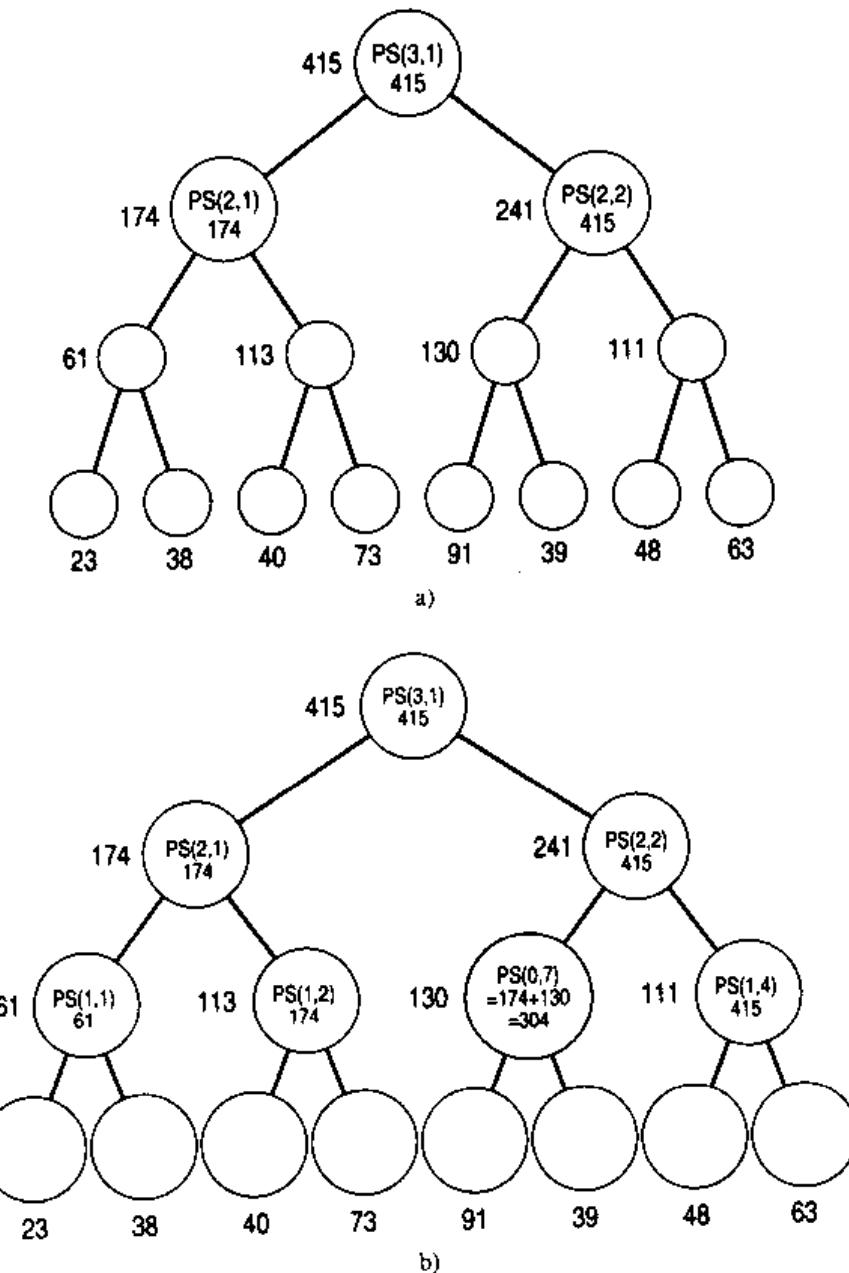


图 4-3

a) 第一阶段 b) 第二阶段

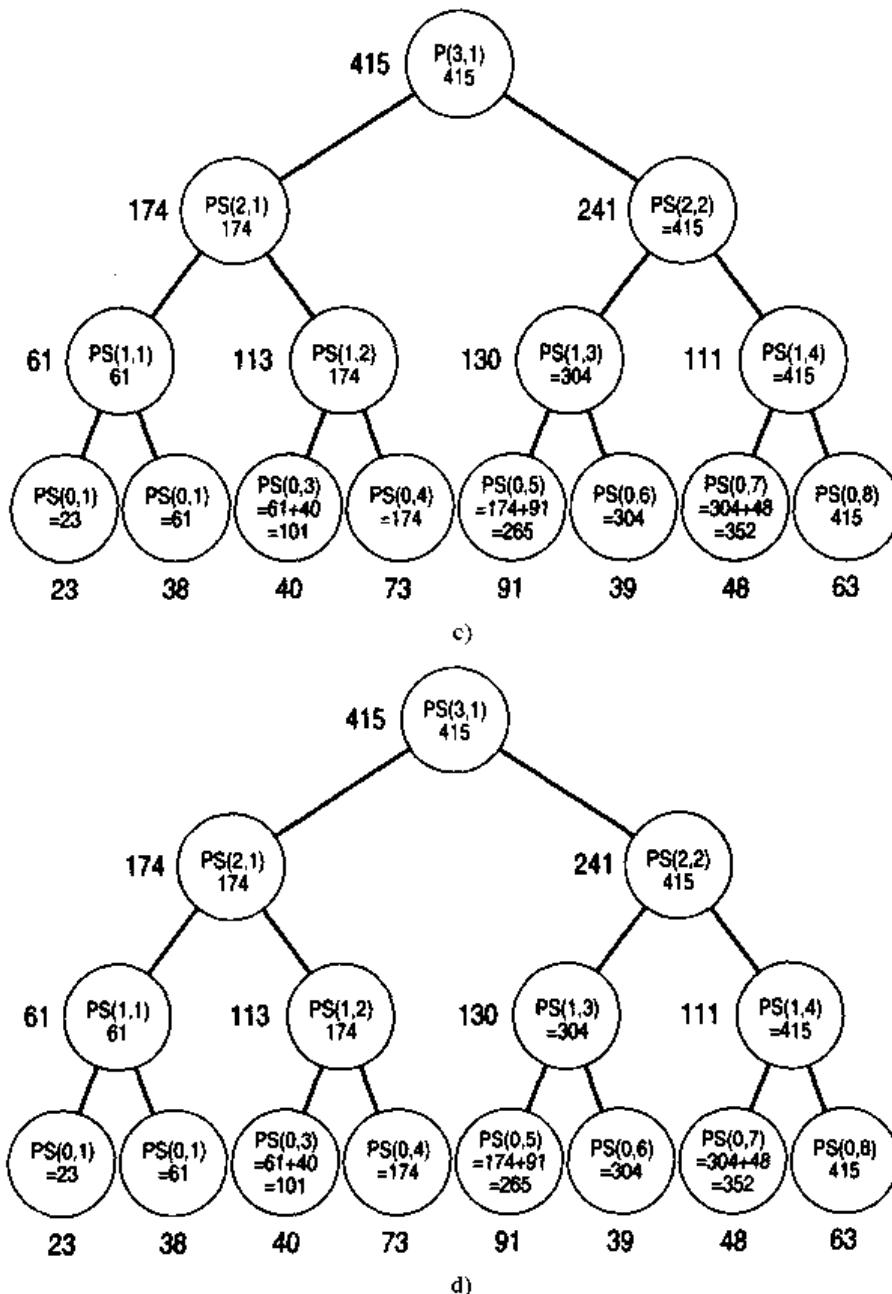


图4-3 (续)

c) 第三阶段 d) 最后结果

4.4 二项式系数

二项式系数 $\binom{n}{r}$ 由如下公式给出:

$$\binom{n}{r} = \frac{[n]}{[r](n-r)}$$

现在要求出所有的二项式系数:

$$\binom{0}{0}, \binom{n}{1}, \binom{n}{2}, \dots, \binom{n}{n}$$

我们可以把二项式系数写成三角表的形式。为了简便，用 C_{nr} 表示 $\binom{n}{r}$ 。

	C_{00}					
C_{10}	C_{11}					
C_{20}	C_{21}	C_{22}				
C_{30}	C_{31}	C_{32}	C_{33}			
C_{40}	C_{41}	C_{42}	C_{43}	C_{44}		
...	
...	

也可以用方形表的形式表示：

C_{00}	C_{10}	C_{20}	C_{30}	C_{40}	C_{50}	...
C_{10}	C_{21}	C_{32}	C_{43}	C_{54}
C_{20}	C_{31}	C_{42}	C_{53}
C_{30}	C_{41}	C_{52}
C_{40}	C_{51}
C_{50}

[131]

如果此二维数组用 P 表示，那么有 $P(i,j) = C_{(i-1),j}$, $i \geq 1, j \geq 0$ 。

由 $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$ ，我们得到 $P(i,j) = P(i-1,j) + P(i-1,j-1)$ 。在 $P(i-1,j)$ 上重复利用这一结果，得：

$$P(i,j) = \sum_{k=0}^j P(k,j-1)$$

我们得到如下表的 P 值：

C_{00}	C_{10}	C_{20}	C_{30}	C_{40}	C_{50}	
C_{10}	C_{21}	C_{32}	C_{43}	C_{54}		
C_{20}	C_{31}	C_{42}	C_{53}			
C_{30}	C_{41}	C_{52}				
C_{40}	C_{51}					
C_{50}						

(i,j)处的值通过累加前一列直到第 i 行位置的值得到。注意 $P(i,0) = C_{(i-1),0} = \binom{i-1}{0} = 1$ ，同时

$P(1,j) = C_{j1} = 1$ 。也就是说，第一行和第一列的所有位置的值都为1。整个过程先对第一列所有的位置写入1，然后求出第一列的部分和并对下一列重复上述过程。

算法Binomial-Coeff

输入：正整数 n

输出：二项式系数 $\binom{n}{0}, \binom{n}{1}, \dots, \binom{n}{n}$

BEGIN

1. For $i = 1$ to $n + 1$ do in Parallel

2. $P(i,0) = 1$
3. End parallel
4. For $j = 1$ to n do
5. Find the partial sums of the $(j-1)$ th column entries using PARTIAL SUM algorithm and store in j th column. That is,

$$P(i,j) = \sum_{k=1}^i P(k,j-1) \text{ where } i = 1 \text{ to } n-j+1$$

6. End for

[132] 7. OUTPUT the result as $P(n+1,0), P(n,1), P(n-1,2), \dots, \dots$

END

复杂度分析 第1~3步可以用 $O(n)$ 个处理器在 $O(1)$ 内完成。用 $O(n)$ 个处理器，第5步可以在 $O(\log n)$ 时间内完成。由于采用了部分和算法，CREW PRAM模型可以实现该算法。现在我们来看该算法如何工作。假定 $n = 6$ 是给定的输入，那么 P 表可以在6次迭代中形成。初始值在表4-1中给出。

表4-1 初始值

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

[133] 在第1次迭代中，第一列的值通过部分求和形成，如表4-2所示。之后的迭代在表4-3到表4-7中给出。在表4-7中，左下到右上的对角位置给出了 $\binom{6}{0}, \binom{6}{1}, \binom{6}{2}, \binom{6}{3}, \binom{6}{4}, \binom{6}{5}$ 和 $\binom{6}{6}$ 的值分别为1、6、15、20、15、6和1。

表4-2 第一阶段的值

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

表4-3 第二阶段的值

	0	1	2	3	4	5	6
1	1	1	1				
2	1	2	3				
3	1	3	6				
4	1	4	10				
5	1	5	15				
6	1	6					
7	1						

表4-4 第三阶段的值

	0	1	2	3	4	5	6
1	1	1	1	1			
2	1	2	3	4			
3	1	3	6	10			
4	1	4	10	20			
5	1	5	15				
6	1	6					
7	1						

表4-5 第四阶段的值

	0	1	2	3	4	5	6
1	1	1	1	1	1		
2	1	2	3	4	5		
3	1	3	6	10	15		
4	1	4	10	20			
5	1	5	15				
6	1	6					
7	1						

表4-6 第五阶段的值

	0	1	2	3	4	5	6
1	1	1	1	1	1	1	
2	1	2	3	4	5	6	
3	1	3	6	10	15		
4	1	4	10	20			
5	1	5	15				
6	1	6					
7	1						

表4-7 最后的值

	0	1	2	3	4	5	6
1	1	1	1	1	1	1	1
2	1	2	3	4	5	6	
3	1	3	6	10	15		
4	1	4	10	20			
5	1	5	15				
6	1	6					
7	1						

4.5 范围内最小值问题

设 $A(1:n)$ 为 n 个数的数组。在 EREW PRAM 模型中，用 $O(n)$ 个处理器可以在 $O(\log n)$ 时间内确定数组元素的最小值。同样的问题也可以在 CRCW PRAM 模型中用 $O(n^2)$ 个处理器在 $O(1)$ 时间内完成（参见第 3 章）。范围内最小值问题也可以表述如下：给定任意 i 和 j ，且 $1 \leq i < j \leq n$ ，求出 A_i, A_{i+1}, \dots, A_j 中最小的元素。

当 $i = 1$ 和 $j = n$ 时，范围内最小值问题化简为数组求最小值的简单问题。预先对数组进行处理并存储成适当的形式，这样可以高效地处理范围内最小值的问题。考虑数组 $A = (8, 3, 4, 5, 2, 11, 15, 17, 19, 7, 16, 5, 9, 10, 2, 8)$ ，一些范围内的最小值在表 4-8 中给出。

表4-8 一些范围最小值

范 围		元 素	最 小 值
i	j		
3	8	4, 5, 2, 11, 15, 17	2
9	12	19, 7, 16, 5	5
5	11	2, 11, 15, 17, 19, 7, 16	2

为了继续进行求解，我们定义两个术语，数组的前缀最小（prefix minima）和后缀最小（suffix minima）。对于数组 a_1, a_2, \dots, a_n ，其前缀最小为数组 $P = (p_1, p_2, \dots, p_n)$ ，其中 $p_i = \min\{a_1, a_2, \dots, a_i\}$ 。即 p_i 是数组前 i 个元素中的最小值。相似地，后缀最小定义为一个序列 $S = (s_1, s_2, \dots, s_n)$ ，其中 s_i 是数组最后 i 个元素中的最小值。

即：

$$\begin{aligned}
 s_1 &= \min\{a_n\} \\
 s_2 &= \min\{a_{n-1}, a_n\} \\
 s_3 &= \min\{a_{n-2}, a_{n-1}, a_n\} \\
 &\dots \\
 &\dots \\
 s_i &= \min\{a_{n-i+1}, \dots, a_n\}
 \end{aligned}$$

如果原来的数组为 $A = (8, 3, 4, 5, 2, 11)$ ，那么其前缀最小和后缀最小分别为 $P = (8, 3, 3, 2, 2)$ 和 $S = (11, 2, 2, 2, 2)$ 。对于有 n 个元素的数组 A （假定 $n = 2^k$ ），为了解决范围内最小值问题，我

们先构建一个有 n 个叶节点的完全二叉树。叶节点用 $(0,1), (0,2), (0,3), \dots, (0,n)$ 表示；下一层的节点用 $(1,1), (1,2), (1,3), \dots, (1, n/2)$ 表示；再下一层的节点用 $(2,1), (2,2), \dots, (2, n/4)$ 表示。所有节点都进行标记。这在图4-4中给出。

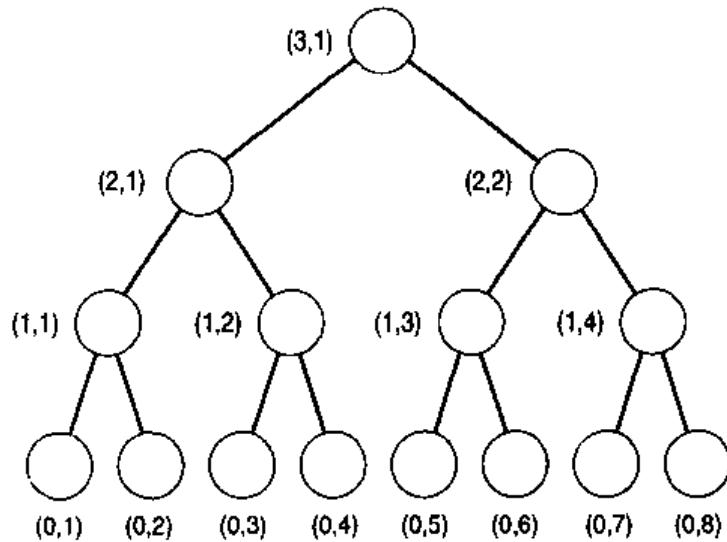


图4-4 节点作标记的完全二叉树

叶节点包含给定的数组 $A(1:n)$ 。也就是说，标记为 $(0,i)$ 的节点含有 A_i 的值。我们在图4-5中给出了有8个元素的数组 $A = (8,3,4,5,2,11,15,17)$ 对应的完全二叉树。

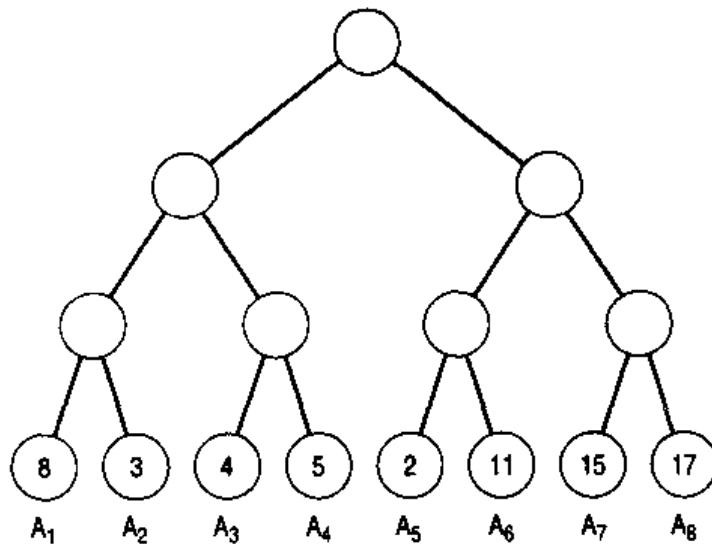


图4-5 初始值

137

每个内部节点 v 都包含有以 v 为根的子树的叶的前缀最小和后缀最小。例如，对于内部节点 $(2,2)$ ，以 $(2,2)$ 为根节点的子树的叶节点为 $(A_5, A_6, A_7, A_8) = (2, 11, 15, 17)$ 。这一子树的前缀最小和后缀最小分别为 $P = (2, 2, 2, 2)$ 和 $S = (17, 15, 11, 2)$ 。因此内部节点 $(2,2)$ 包含两个数组： $P = (2, 2, 2, 2)$ 和 $S = (17, 15, 11, 2)$ 。其他节点的内容在图4-6中给出。怎样确定 P 和 S 呢？现在让我们来看如何求出每个节点 (h,i) 的 P 和 S 数组。这要用从下至上的方式来完成。对第一层的所有节点， P 和 S 各有两个元素。如果 $(1,i)$ 是第一层的节点，它的左孩子为 $(0,2i-1)$ ，右孩子为 $(0,2i)$ ，它们分别包含 A_{2i-1} 和 A_{2i} 。

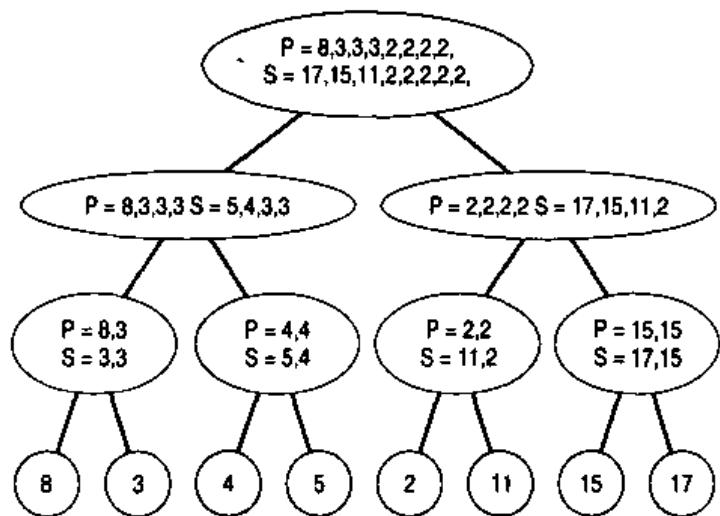


图4-6 P和S数组

设 $x = \min\{A_{2i-1}, A_{2i}\}$, 那么内部节点 $(0, i)$ 所对应的 P 和 S 如下:

$$S_{0,i} = A_{2i-1} x$$

当高度大于1的时候，节点 (h,i) 的左孩子为 $(h-1,2i-1)$ ，右孩子为 $(h-1,2i)$ 。确定 $P_{h,i}$ 的过程如下：

1. 通过连接两个列表 $P_{h-1, 2i-1}$ 和 $P_{h-1, 2i}$, 得到 $P_{h, i}$;
 2. $P_{h, i}$ 后半部分的每个元素 x 都被 x 和 $P_{h, i}$ 前半部分的最后一个元素中的最小值替代。

此过程可以用下面的具体例子来演示：

例 1

考慮兩組值：

$$P_{h-1,2i-1} = (7,7,4,4,4,3,3,3)$$

第一步，把两个数组连接得到 P_1 ：

$$P_{\text{c}} = (7,7,4,4,4,3,3,3 \quad \quad \quad 10,8,8,8,8,8,6,6)$$

现在，要用 x 和 $P_{k,i}$ 前半部分的最后一个元素中的最小值替代 x 。在这里，前半部分的最后一个元素是3。因此，第9个元素10将被 $\min\{10, 3\}$ 也就是3替代。相似地，第10个元素8将被3替代。最后，得到 $P_{k,i} = (7, 7, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3)$ 。

例2

假定

$$P_{k-1,2i-1} = (7,7,4,4,4,3,3,3)$$

在这种情况下，我们首先把两个数组连接，得到

$$P_{b,i} = (7,7,4,4,4,3,3,3 \quad \quad \quad 10,8,8,8,2,2,1,1)$$

现在，用 x 和 P_A 前半部分的最后一个元素中的最小值替代 x ，得到：

$$P_{\mu} = (7,7,4,4,4,3,3,3 \quad \quad \quad 3,3,3,3,2,2,1,1)$$

求数组S的方法也相似。对第一层的节点 $(1,i)$, 数组S为 $S_{1,i} = (A_{2,i}, x)$, 其中 $x = \text{Min}\{A_{2i+1}, A_{2i}\}$ 。然后, 对于高度 $h > 1$ 的节点, 首先连接 $S_{h-1,2i}$ 和 $S_{h-1,2i+1}$ 。注意, $S_{h-1,2i}$ 是 $S_{h,i}$ 的前半部分, $S_{h-1,2i+1}$ 是 $S_{h,i}$ 的后半部分。用 x 和 $P_{h,i}$ 前半部分的最后一个元素中的最小值替代 x 。

当所有节点 (h,i) 的前缀和后缀数组P和S都已知的时候, 范围内最小值问题可以在 $O(1)$ 时间内解决如下: 设 i 和 j 为两个整数, 且 $1 \leq i < j \leq n$ 。在完全二叉树中找到节点 $(0,i)$ 和 $(0,j)$ 的最低公共祖先节点 (h,k) 。由于这个树是完全的, 这可以在 $O(1)$ 时间内完成。我们确定数组 $S_{h-1,2k-1}$ 和 $P_{h-1,2k}$ 中的元素 $A_i, A_{i+1}, A_{i+2}, \dots, A_j$ 的最小值 m 。

假定 x 为数组 $S_{h-1,2k-1}$ 中节点 $(0,i)$ 对应的元素, y 为数组 $P_{h-1,2k}$ 中节点 $(0,j)$ 对应的元素。 $\text{Min}(x,y)$ 是元素 $A_i, A_{i+1}, A_{i+2}, \dots, A_j$ 的最小值。如果每个节点的P和S已知, 那么可以在 $O(1)$ 时间内完成范围内最小值问题。例如, 考虑图4-5中给出的数, 假定 $i = 3$ 和 $j = 7$, 那么这两个节点的最低公共祖先节点为根节点。所以, 我们必须考虑其左子节点的数组S和右子节点的数组P。它们是:

$$S = 5, 4, 3, 3$$

$$P = 2, 2, 2, 2$$

数组S中对应 i 的是第3个元素, 其值为3; 而数组P中对应 $j = 7$ 的也是第3个元素, 其值为2。因此 $\min\{3,2\}$ 的解为2。

复杂度分析 给定数组 $A(1:n)$, 利用 $O(n \log n)$ 个处理器, 我们可以在 $O(\log n)$ 的时间内完成求二叉树和每个节点的S和P值的问题。当完成前期处理工作之后, 对应任意 i 和 j ($1 \leq i < j \leq n$)的范围内最小值查询问题可以在 $O(1)$ 时间内完成。139

范围内最小值问题有很多应用领域。它可以应用于树的最低公共祖先节点的问题中 (见第5章)。

参考文献

- Cole, R. and Vishkin, U. (1989) Faster Optimal Prefix Sums and List Ranking, *Information and Computing*, **81**(3), 344–352.
- Kruskal, Rudolph C. L., and Snir, M. (1985) The Power of Parallel Prefix, *IEEE Transactions on Computers*, **C-34**(10), 965–968.
- Valiant, L. G., Skyum, S., Berkowitz, S., and Rackoff, C. (1983) Fast Parallel Computation of Polynomials Using Few Processors, *SIAM Journal of Computing*, **12**(4), 641–644.

习题

- 4.1 设 $A(0:n-1)$ 为一个排序的数组。给定下标 i , $A(i)$ 的左匹配是最大的整数 k , 其中 $0 \leq k < i$ 且 $A(k) < A(i)$ 。请设计一个算法, 利用 $O(n^2)$ 个处理器在 $O(1)$ 时间内求每一个数组元素的左匹配。
- 4.2 设 $A(0:n-1)$ 为一个整数数组。请设计一个算法找到最小的下标 k , 其中 $A(k)$ 是正数, 且算法的时间复杂度为 $O(\log n)$ 。再设计一个能在 $O(1)$ 时间内解决这个问题的算法。
- 4.3 设 X 为一个大小为 n 的布尔数组。请用CRCW PRAM模型设计一个算法, 在 $O(1)$ 时间内求出最大的下标 k , 使得 X_k 为假。
- 4.4 设 X 和 Y 为两个大小分别为 n 和 m 的数组。 $\text{rank}(a:X)$ 是数组 X 中小于或等于 a 的元素的个数。我们定义 $\text{rank}(X:Y)$ 为数组 $(r_1, r_2, r_3, \dots, r_n)$, 其中 $r_i = \text{rank}(x_i:Y)$ 。请设计一个并行算

法求出数组 $\text{rank}(X:Y)$ 。

- 4.5 如果对每一个 i 和 j , 都有 $A(i,j)$ 和 $A(j,i)$ 相等, 那么矩阵 A 为对称矩阵。
 a) 在 CREW PRAM 模型上设计一个算法检查一个矩阵是否为对称矩阵。你设计的算法的复杂度是多少?
 b) 在 CRCW PRAM 模型上设计一个算法检查一个矩阵是否为对称矩阵。你设计的算法的复杂度是多少?
- 4.6 请设计一个算法将两个 $n \times m$ 的矩阵相加, 并验证你的算法是否需用 $O(mn)$ 个处理器且时间复杂度为 $O(1)$ 。
- 4.7 如果一个矩阵的对角线以上的元素为零, 那么该矩阵就叫做下三角矩阵。给定一个矩阵 A , 我们想判断它是否为下三角矩阵。请在 CREW PRAM 模型上设计一个时间复杂度为 $O(1)$ 的算法。
- 4.8 给定阶为 $n \times n$ 的矩阵, 设计一个算法求行的所有元素的和。你的算法的时间和处理器复杂度是多少?
- 4.9 设 $A(0:n-1)$ 为一个大小为 n 的数组。范围内求和问题就是给定任意 i 和 j , 求 $A(i), A(i+1), \dots, A(j)$ 的和, 其中 $0 \leq i < j \leq n-1$ 。当 $i = 0$ 且 $j = n-1$ 的时候, 此问题成为普通的求和问题。请设计一个算法求对所有可能的 i 和 j 的值的范围和。
140
- 4.10 设 $A(1:n)$ 为一个数组, $B(1:m)$ 为另外一个数组, 其中 $m < n$ 。我们需要验证数组 B 是否为数组 A 的子数组。例如, 假定 $A = (2, 1, 7, 5, 120, 5, 2, 7, 1, 1, 8)$, $B = (5, 120, 5, 2)$, 那么 B 为 A 的子数组。如果 $B = (1, 2, 7, 5)$, 那么 B 不是 A 的子数组。
 a) 在 CREW PRAM 模型上设计一个算法验证数组 B 是否为数组 A 的子数组, 其时间复杂度为 $O(\log n)$;
 b) 对同一个问题在 CRCW PRAM 模型上设计一个并行算法。
- 4.11 设计一个并行算法, 输入是数组 $A(0:n-1)$, 求数组中给定数的出现次数。例如, 如果 $A = (1, 2, 3, 1, 4, 1, 2, 1, 3, 2, 3, 3)$, 那么 1 出现 4 次, 2 出现 3 次, 3 出现 4 次。设计一个并行算法, 将数组 $A(0:n-1)$ 和 x 作为输入, 求 x 在数组中的出现次数。
141
- 4.12 设 $A(0:n-1)$ 为一个数组, 数组的模式指数组中出现频率最高的数的值。如果数组 $A = (1, 2, 3, 4, 1, 5, 1, 7, 1, 9, 2, 8, 1, 3)$, 那么 1 出现 5 次, 2 出现 2 次, 3 出现 2 次, 4 出现 1 次。既然 1 出现的次数最多 (5 次), 则数组的模式为 1。请设计一个并行算法求给定数组的模式。
144

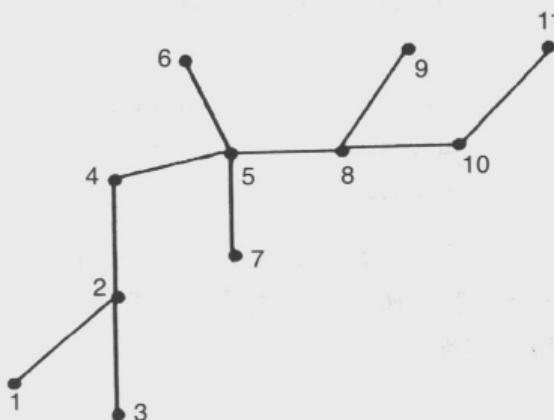
第二部分 图模型算法

第5章 树 算 法

本章将学习关于树的一些基本算法。我们已经学习了欧拉图和欧拉圈，树是不含圈的，树的悬挂点的度是奇数1，因此树不是欧拉图。但是我们可以用不同的方法给出树的欧拉圈的定义，并且学习判定树的欧拉圈的方法。

5.1 欧拉圈

设 $T = (V, E)$ 是一个图， T' 是由两个有向弧 $\langle u, v \rangle$ 和 $\langle v, u \rangle$ 代替 T 的每条边 (u, v) 而得到的有向图。 T' 是欧拉图， T' 的欧拉圈也称为 T 的欧拉圈。假定树用一个相邻列表来表示，例如，图5-1中的树的相邻列表如表5-1所示。



145

图5-1 树

表5-1 树的相邻列表

Vertex (v)	adj(v)
1	2
2	1,3,4
3	2
4	2,5
5	4,6,7,8
6	5
7	5
8	5,9,10
9	8
10	8,11
11	10

我们用一个有向弧列表来表示图的欧拉圈，欧拉圈为

$\{<1, 2><2, 3><3, 2><2, 4><4, 5><5, 6><6, 5><5, 7><7, 5><5, 8><8, 9><9, 8><8, 10><10, 11><11, 10><10, 8><8, 5><5, 4><4, 2><2, 1>\}$

这个列表可以用后继函数来表示。后继函数(Successor)是 T 的有向弧集合到它自身的一个双射。它可以完全确定一个欧拉圈。因此求图的欧拉圈的问题可以归结为求后继函数的问题。后继函数定义如下：考虑顶点5，与顶点5相邻的顶点有4, 6, 7和8。如果我们考察欧拉圈，会发现

$$\begin{aligned}\text{Successor } <4, 5> &= <5, 6> \\ \text{Successor } <6, 5> &= <5, 7> \\ \text{Successor } <7, 5> &= <5, 8> \\ \text{Successor } <8, 5> &= <5, 4>\end{aligned}$$

一般情况下，假定 v 是一个顶点， $u_0, u_1, u_2, \dots, u_{d-1}$ 是与 v 相邻的顶点，这里顶点 v 的度数是 d ，

[46]

我们有

$$\begin{aligned}\text{Successor}(<u_0, v>) &= <v, u_1> \\ \text{Successor}(<u_1, v>) &= <v, u_2> \\ \text{Successor}(<u_2, v>) &= <v, u_3> \\ \cdots & \\ \cdots & \\ \text{Successor}(<u_{d-2}, v>) &= <v, u_{d-1}> \\ \text{Successor}(<u_{d-1}, v>) &= <v, u_0>\end{aligned}$$

考虑带有某些附加指针且由相邻列表所表示的树。对于任意弧 $<u_i, v>$ ，在 $O(1)$ 时间内可以求出 $<v, u_{(i+1) \bmod d}>$ ，因此在下面的求树的欧拉圈的并行算法中，需 $O(n)$ 个处理器，执行时间为 $O(1)$ ，用EREW PRAM模型来实现。

算法Euler Circuit

输入：具有附加指针且由相邻列表所表示的树 T

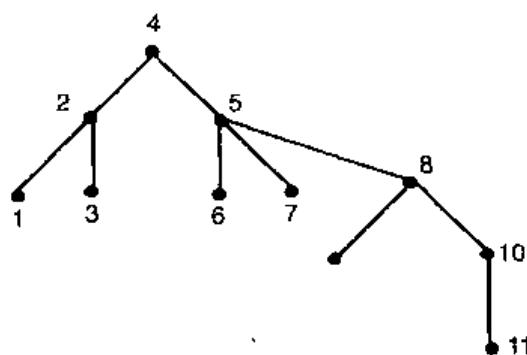
输出：每一个弧 $<u, v>$ 的后继函数 $\text{Successor}(<u, v>)$

1. For every arc $<u, v>$, do step 2 in parallel
2. $\text{Successor } (<u, v>) = <v, w>$, where w occurs next to u in the ordered list of vertices adjacent to v . If u appears last in the list of vertices adjacent to v , then w is the first node in the list

我们已设计了一个在 $O(1)$ 时间内求树的欧拉圈的算法。利用欧拉圈，可以解决许多有趣的问题。下面我们首先研究如何将一个树变成以给定顶点 v 为根的有根树。

5.2 给树加根

有根树是指以一个顶点为根，其他顶点为后代的树。一个有根树可以用它的双亲函数来表示。图5-1中所示的树 T ，它可以表示为以4为根的有根树，如图5-2所示。我们通常用双亲关系代替相邻列表来表示一个有根树，如表5-2所示。



147

图5-2 以4为根的树

表5-2 双亲关系

顶点	1	2	3	4	5	6	7	8	9	10	11
双亲	2	4	2	4	4	5	5	5	8	8	10

现在要从相邻列表出发来确定双亲函数，为此，我们利用欧拉圈。

给定的相邻关系是：给定 $p(2) = 4$; $\langle 2, 1 \rangle$ 出现在 $\langle 1, 2 \rangle$ 之前，有 $p(1) = 2$; $\langle 2, 3 \rangle$ 出现在 $\langle 3, 2 \rangle$ 之前，有 $p(2) = 2$; $\langle 4, 5 \rangle$ 出现在 $\langle 5, 4 \rangle$ 之前，则有 $p(5) = 4$ 。

现在，我们必须清楚如何确定一个弧出现在另一个弧之前。给每一个弧赋以权重1，求加权弧的前缀和。如果 $\langle u, v \rangle$ 的前缀和小于 $\langle v, u \rangle$ 的前缀和，则意味着 $\langle u, v \rangle$ 出现在前，因而有 $p(v) = u$ 。下面给出为树加根的并行算法。

算法Rooting

输入：

1. 由相邻列表定义的树 T
2. 由后继函数定义的 T 的欧拉环游
3. 特殊顶点

输出：对于每一个顶点 $v = r$ ，输出以 r 为根的有根树中的双亲 $p(v)$

1. Identify the last vertex v in the list of vertices adjacent to r and set successor $(\langle u, r \rangle) = 0$
2. Assign weight 1 to each arc $\langle u, v \rangle$
3. Find the prefix sum of the weights of the arcs in the list given by the successor function
4. For each arc $\langle u, v \rangle$, do the following in parallel: If the weight of $\langle u, v \rangle$ is less than the weight of $\langle v, u \rangle$, then set $p(v) = u$
5. End ROOTING

复杂度分析 第1~2步可在 $O(1)$ 时间内完成；第3步利用第3章中给出的前缀和算法，用 $O(n)$ 个处理器，执行时间为 $O(\log n)$ ；第4步用 $O(n)$ 个处理器，执行时间为 $O(1)$ ；因此该算法用 $O(n)$ 个处理器且总的执行时间为 $O(\log n)$ 。由于并行前缀和算法执行CREW PRAM 模型，因此该算法也执行CREW PRAM模型。

5.3 后序编号

本节给出欧拉圈算法的另一个应用。后序遍历（postorder traversal）法是顺序访问树的顶

148

点的方法。以 r 为根的树 T 的后序遍历由从左到右的 r 的子树的后序遍历组成，其中树根 r 位于最后。例如，如图5-3所示的简单树，4是树根，2、3和5是4的孩子，这个树的后序遍历是1, 2, 3, 6, 7, 11, 10, 8, 9, 5, 4。

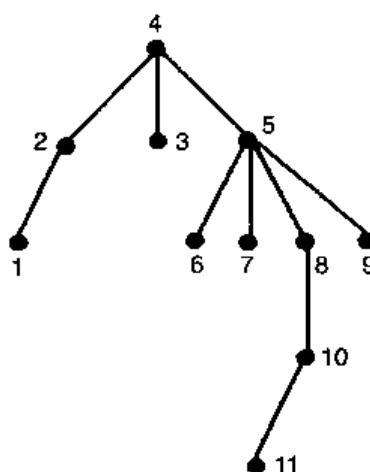


图5-3 树

后序编号是一个函数，它给出在后序遍历序列中顶点的排列。例如，图5-3中顶点的后序编号为： $\text{post}(1) = 1$; $\text{post}(2) = 2$; $\text{post}(3) = 3$; $\text{post}(6) = 4$; $\text{post}(7) = 5$; $\text{post}(11) = 6$; $\text{post}(10) = 7$; $\text{post}(8) = 8$; $\text{post}(9) = 9$; $\text{post}(5) = 10$; $\text{post}(4) = 11$ （见表5-3）。

表5-3 后序编号

v	1	2	3	4	5	6	7	8	9	10	11
$\text{post}(v)$	1	2	3	11	10	4	5	8	9	7	6

后序编号可以由欧拉圈求出。在欧拉圈中，只要沿着弧 $\langle v, p(v) \rangle$ 移动，都会遍历顶点 v 。这个过程由下面几步来描述：

1. 对于每一个弧 $\langle u, v \rangle$ ，如果 u 是 v 的双亲，则对弧 $\langle u, v \rangle$ 赋权重0；如果 v 是 u 的双亲，则对弧 $\langle u, v \rangle$ 赋权重1；
2. 对于由欧拉圈的后继函数确定的列表，执行加权弧的前缀和操作；
3. 对于每一个顶点 v ， $\text{post}(v)$ 是弧 $\langle v, p(v) \rangle$ 的前缀和；
4. 树根的后序编号为 n ，其中 n 是树的顶点个数。

对于图5-3所示的树，上述过程由表5-4给出。表5-4给出欧拉路，权重和前缀和。算法如下：

算法Postorder Numbering

输入：

1. 由双亲关系 $p(v)$ 给定的以 r 为根的有根树 $T = (V, E)$

2. T 的欧拉圈

输出：对于每一个顶点 v ，后序编号 $\text{post}(v)$

1. For every arc $\langle u, v \rangle$ do in parallel

If $u = p(v)$, assign the weight 0, else assign the weight 1 to the arc $\langle u, v \rangle$. End Parallel

2. Find the prefix sum of the list of weights specified by the successor function
3. For every vertex v do in parallel
 - post(v) = prefix sum of the arc $\langle v, p(v) \rangle$
 - End parallel
4. $p(r) = n$
5. End Post order Numbering

表5-4 后序编号作为前缀和

欧拉路	权重	前缀和
$\langle 4,2 \rangle$	0	0
$\langle 2,1 \rangle$	0	0
$\langle 1,2 \rangle$	1	$1 \rightarrow \text{post}(1) = 1$
$\langle 2,4 \rangle$	1	$2 \rightarrow \text{post}(2) = 2$
$\langle 4,3 \rangle$	0	2
$\langle 3,4 \rangle$	1	$3 \rightarrow \text{post}(3) = 3$
$\langle 4,5 \rangle$	0	3
$\langle 5,6 \rangle$	0	3
$\langle 6,5 \rangle$	1	$4 \rightarrow \text{post}(6) = 4$
$\langle 5,7 \rangle$	0	4
$\langle 7,5 \rangle$	1	$5 \rightarrow \text{post}(7) = 5$
$\langle 5,8 \rangle$	0	5
$\langle 8,10 \rangle$	0	5
$\langle 10,11 \rangle$	0	5
$\langle 11,10 \rangle$	1	$6 \rightarrow \text{post}(11) = 6$
$\langle 10,8 \rangle$	1	$7 \rightarrow \text{post}(10) = 7$
$\langle 8,5 \rangle$	1	$8 \rightarrow \text{post}(8) = 8$
$\langle 5,9 \rangle$	0	8
$\langle 9,5 \rangle$	1	$9 \rightarrow \text{post}(9) = 9$
$\langle 5,4 \rangle$	1	$10 \rightarrow \text{post}(5) = 10$

复杂度分析 算法所用的处理器数为 $O(n)$, 执行时间为 $O(\log n)$, 执行CREW PRAM模型。 [150]

5.4 后代个数

对于每一顶点, 它的后代个数可以由后序编号算法中确定的加权弧的前缀和得到。顶点 v 的后代个数等于以 v 为根的极大子树的顶点个数, 它是 $\langle p(v), v \rangle$ 的前缀和与 $\langle v, p(v) \rangle$ 的前缀和的差。例如, 图5-3所示的树, 根据前缀和的列表, 我们知道 $\langle 4, 5 \rangle$ 和 $\langle 5, 4 \rangle$ 的前缀和分别为 3 和 10, 因此顶点 5 的后代个数是 $10 - 3 = 7$, 也就是说以 5 为根的极大子树有 7 个顶点 (这 7 个顶点是 5, 6, 7, 8, 9, 10 和 11)。弧 $\langle 5, 8 \rangle$ 和 $\langle 8, 5 \rangle$ 的前缀和分别为 5 和 8, 因此顶点 8 的后代个数为 3 (分别为顶点 8, 10 和 11)。

5.5 顶点层数

根的层数为 0。遍历欧拉路。当经过弧 $\langle p(v), v \rangle$ 时, $p(v)$ 的层数比 v 的层数少 1; 而当经过弧 $\langle v, p(v) \rangle$ 时, v 的层数比 $p(v)$ 的层数多 1。因而我们给所有形如 $\langle p(v), v \rangle$ 的弧赋权重为 1, 给

所有形如 $\langle v, p(v) \rangle$ 的弧赋权重为-1。在由欧拉路确定的列表中执行前缀和，Level(v)就是弧 $\langle p(v), v \rangle$ 的前缀和。表5-5表示由图5-3所示的树中各顶点的层数。

表5-5 层数

欧拉路	权重	前缀和
$\langle 4,2 \rangle$	1	$1 \rightarrow \text{level}(2) = 1$
$\langle 2,1 \rangle$	1	$2 \rightarrow \text{level}(1) = 2$
$\langle 1,2 \rangle$	-1	1
$\langle 2,4 \rangle$	-1	0
$\langle 4,3 \rangle$	1	$1 \rightarrow \text{level}(3) = 1$
$\langle 3,4 \rangle$	-1	0
$\langle 4,5 \rangle$	1	$1 \rightarrow \text{level}(5) = 1$
$\langle 5,6 \rangle$	1	$2 \rightarrow \text{level}(6) = 2$
$\langle 6,5 \rangle$	-1	1
$\langle 5,7 \rangle$	1	$2 \rightarrow \text{level}(7) = 2$
$\langle 7,5 \rangle$	-1	1
$\langle 5,8 \rangle$	1	$2 \rightarrow \text{level}(8) = 2$
$\langle 8,10 \rangle$	1	$3 \rightarrow \text{level}(10) = 3$
$\langle 10,11 \rangle$	1	$4 \rightarrow \text{level}(11) = 4$
$\langle 11,10 \rangle$	-1	3
$\langle 10,8 \rangle$	-1	2
$\langle 8,5 \rangle$	-1	1
$\langle 5,9 \rangle$	1	$2 \rightarrow \text{level}(9) = 2$
$\langle 9,5 \rangle$	-1	1
$\langle 5,4 \rangle$	1	2

5.6 最低公共祖先

本节我们研究关于有根树的一个非常有趣且有用的问题。在以 r 为根的有根树 T 中，从顶点 v 到 r 路上的顶点称为 v 的祖先。给定两个顶点 u 和 v ，离根最远的 u 和 v 的公共祖先称为 u 和 v 的最低公共祖先，并记为LCA(u, v)。例如，图5-3所示的有根树中， $\text{LCA}(2,6) = 4$; $\text{LCA}(6,10) = 5$; $\text{LCA}(7,11) = 5$; $\text{LCA}(8,11) = 8$ 。

当 T 本身是一个简单路时，两个顶点 u 和 v 的最低公共祖先是 u 和 v 中离根较近的那个顶点。人们比较早地解决了完全二叉树的最低公共祖先问题。如果 T 是一个完全二叉树，首先给顶点标号，以便用于顺序遍历。如果 u 和 v 是两个顶点的标号，顶点 u 和 v 的最低公共祖先可通过下面步骤得到：

1. 将 u 和 v 分别表示成二进制数 x 和 y 。从左端开始比较 x 和 y ，假设他们在第 i 位首次出现不同值；

2. 将 x 和 y 从左端开始的前 $i-1$ 位数记为 $b_1 b_2 b_3 \dots b_{i-1}$ ；

3. u 和 v 的最低公共祖先标记为 $b_1 b_2 b_3 \dots b_{i-1} 100 \dots 0$ 。

例如： $u = 47$, $v = 61$, u 和 v 的二进制表示分别为

$$u = 101111$$

$$v = 111101$$

从左到右比较 u 和 v 的二进制表示，不同字位的位置在第二个，因此 $u = 47$ 和 $v = 61$ 的最低公共祖先的二进制表示为110 000，即 $\text{LCA}(47, 61) = 48$ 。

一般有根树的最低公共祖先 本节给出求一般有根树的最低公共祖先的方法。利用树的欧拉圈，首先求以根为起点的树的欧拉圈。这个欧拉圈肯定是弧的一个列表。在这个列表中，用顶点 u 来代替弧 $\langle u, v \rangle$ ，则在列表的起始位置恰好是树根。我们称这个列表为欧拉数组 A 。例如，图5-3给出的有根树的欧拉数组为 $A = (4, 2, 1, 2, 4, 3, 4, 5, 6, 5, 7, 5, 8, 10, 11, 10, 8, 5, 9, 5)$ 。欧拉数组 A 的一般形式为：

$$A = (a_1, a_2, \dots, a_m)$$

假定 $B = (b_1, b_2, \dots, b_m)$ ，这里 b_i 表示树 T 中顶点 a_i 的层数。对于图5-3所示的树，有

$$B = (0, 1, 2, 1, 0, 1, 0, 1, 2, 1, 2, 1, 2, 1, 2, 3, 4, 3, 2, 1, 2, 1)$$

进一步，我们还定义另外两个数组 $l(v)$ 和 $r(v)$ 。对于每一个顶点 v ， $l(v)$ 是指在数组 A 中 v 在最左边出现时的指标， $r(v)$ 是指在数组 A 中 v 在最右边出现时的指标。对于图5-3所示的树， $l(v)$ 和 $r(v)$ 由表5-6给出。

表5-6 $l(v)$ 和 $r(v)$ 的值

顶点 v	1	2	3	4	5	6	7	8	9	10	11
$l(v)$	3	2	6	1	8	9	11	13	19	14	15
$r(v)$	3	4	6	21	20	9	11	17	19	16	15

$$\text{Index} = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)$$

$$A = (4, 2, 1, 2, 4, 3, 4, 5, 6, 5, 7, 5, 8, 10, 11, 10, 8, 5, 9, 5, 4)$$

$$B = (0, 1, 2, 1, 0, 1, 0, 1, 2, 1, 2, 1, 2, 3, 4, 3, 2, 1, 2, 1, 0)$$

我们从以根为起点的欧拉路出发，执行深度优先搜索法。在执行过程中首先从 v 的双亲到 v ，利用最左边出现的 $a_i = v$ ，得到 $\text{level}(a_{i-1}) + 1 = \text{level}(a_i)$ 。类似地，仅当访问完以 v 为根的子树上的所有顶点后， v 才能返回到它的双亲。如果已经由 v 返回到 v 的双亲，就不会再回到 v 。因此，利用最右边出现的 $a_j = v$ ，有 $a_{j+1} = v$ 的双亲。所以，最右边出现的 $a_j = v$ ，当且仅当 $\text{level}(a_j) - 1 = \text{level}(a_{j+1})$ 。综上所述，我们有如下结论：

定理5-1

1. 给定数组 A ， $a_i = v$ 是在数组 A 中 v 的最左边出现，当且仅当 $\text{level}(a_{i-1}) + 1 = \text{level}(a_i)$ ；
2. $a_j = v$ 是数组 A 中 v 的最右边出现，当且仅当 $\text{level}(a_j) - 1 = \text{level}(a_{j+1})$ 。

由于在欧拉圈中采用深度优先搜索法，我们有如下结论：

定理5-2

1. u 是 v 的祖先，当且仅当 $l(u) < l(v) < r(u)$ ；
2. 如果 u 不是 v 的祖先且 v 不是 u 的祖先，我们称 u 和 v 是不相关的。 u 和 v 是不相关的，当且仅当要么 $r(u) < l(v)$ ，要么 $r(v) < l(u)$ 。

当 u 是 v 的祖先时，有 $\text{LCA}(u, v) = u$ ；当 u 和 v 不相关时，必须用数组 B 求 $\text{LCA}(u, v)$ 。由观察知 $\text{LCA}(u, v)$ 出现在 u 到 v 的路上。当 u 和 v 不相关时，不失一般性，假定 $r(u) < l(v)$ ，则 $\text{LCA}(u, v)$ 位于最右边出现的 u 与最左边出现的 v 之间的欧拉路中，这部分的欧拉路仅仅访问从 u 到 v 这条路上的顶点以及它们的后代。因此在这部分的欧拉路具有最小层数的顶点是 $\text{LCA}(u, v)$ 。因此

我们有如下结论：

定理5-3 如果 $r(u) < l(v)$, 那么 $\text{LCA}(u,v)$ 是在从 $r(u)$ 到 $l(v)$ 的区间上具有最小层数的顶点。

综上所述，我们得到下面的算法。

算法LCA

输入：

1. 基于双亲关系的有根树 T
2. 两个节点 u 和 v

输出： $\text{LCA}(u,v)$

1. Find the Euler path beginning from the root
2. In the Euler path, replace every arc (u, v) by vertex u and insert root at the beginning and get the array A
3. Find array $B = (b_1, b_2, \dots, b_n)$, where $b_i = \text{level number of } a_i$, the i th entry of A
4. For every vertex x of T . Find $l(x)$ and $r(x)$, using the arrays A and B
5. If $l(u) < l(v) < r(u)$, then set $\text{LCA}(u, v) = u$ and return
6. If $l(v) < l(u) < r(v)$, then set $\text{LCA}(u, v) = v$ and return
7. If $r(u) < l(v)$ set $i = r(u)$ and $j = l(v)$, else set $i = r(v)$ and $j = l(u)$
8. Choose the index k that is minimum in $\{b_i, b_{i+1}, b_{i+2}, \dots, b_j\}$
9. $\text{LCA}(u, v) = a_k$

End LCA

154

复杂度分析 求欧拉路和顶点的层数所用的时间为 $O(1)$ ，求 $l(x)$ 和 $r(x)$ 所用的时间也为 $O(1)$ 。这表明第1~4步的执行时间为 $O(1)$ ，第5~7步执行时间为 $O(1)$ ；第8步是范围最小问题，通过对数组的预处理，求范围最小所需时间也为 $O(1)$ ，但预处理需要处理器数为 $O(n \log n)$ ，执行时间为 $O(\log n)$ 。因此LCA问题的总复杂度是用 $O(n \log n)$ 个处理器，执行时间为 $O(\log n)$ 。

5.7 树收缩

树收缩是指收缩一个二叉树，并把它表示成仅含有三个顶点的完全二叉树的系统方法。这可以应用到算术表达式计算中。本节我们研究树收缩算法，下一节我们研究它在算术表达式中的应用。例如，考虑 $(2*x+4*y) / (2+a)$ ，这个算术表达式将被表示成如图5-4a所示的二叉树。每一个内部顶点表示一个运算，每一个叶子顶点表示一个运算数。

为了计算这个表达式，首先计算出 $2*x$, $4*y$ 和 $2+a$ 的表达式（见图5-4b）；然后将 $2*x$ 和 $4*y$ 加起来（见图5-4c）；最后， $2*x+4*y$ 被 $2+a$ 除（见图5-4c）。这里，原来的树被收缩成最后含有表达式值的树。为了收缩树，我们引入运算 *rake*。

rake运算 考虑二叉树 $T = (V, E)$ ，每个内部顶点恰有两个孩子。假设 v 是叶节点但不是根的孩子。记 $p(v)$ 表示 v 的双亲， $s(v)$ 表示 v 的兄弟。对 v 的 *rake* 运算（见图5-5）步骤如下：

1. 去掉顶点 v 和 $p(v)$ ；
2. 使得 $s(v)$ 为 $p(v)$ 的双亲的孩子。

155

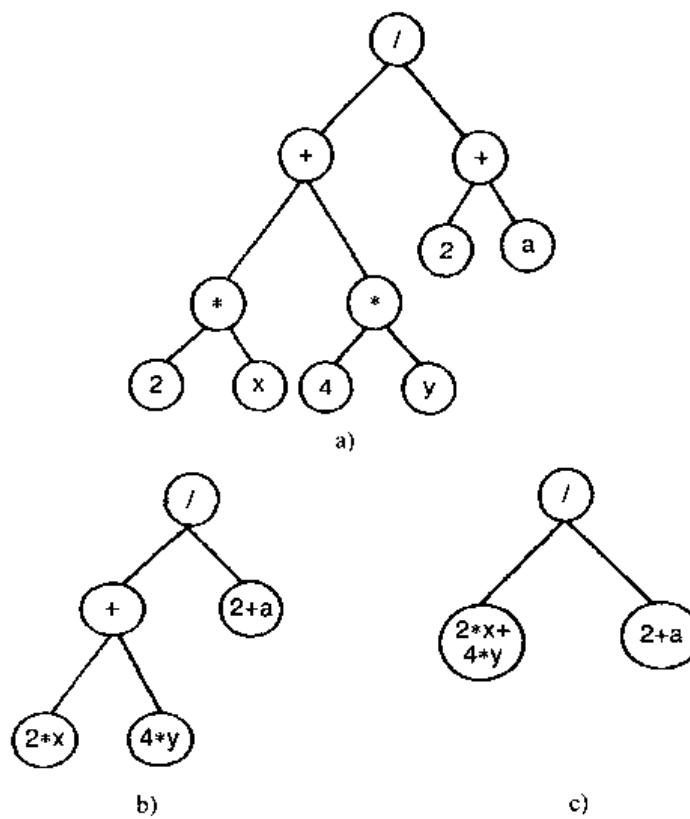


图 5-4

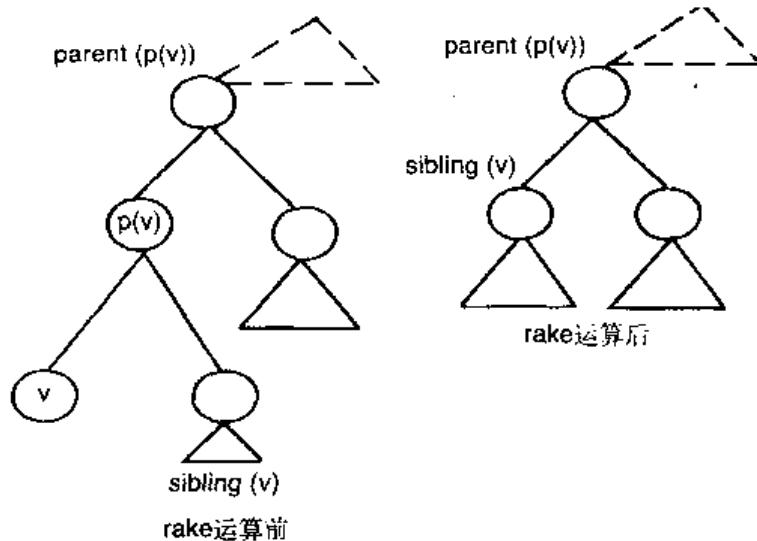
a) $(2*x+4*y) / (2+a)$ b) 收缩树 c) 收缩树

图 5-5 rake 运算

现在考察rake运算是否能够对许多叶节点同时执行。如果两个顶点 u 和 v 是兄弟，对 u 和 v 同时进行rake运算可能产生某些混乱。类似地，如果 u 的双亲是 v 的祖父母，同时执行rake运算也可能产生冲突。为了选择顶点使得可以同时对它们执行rake运算，去掉最左边和最右边的叶节点，然后从左到右对叶节点进行编号。假设它们分别记为 $L_1, L_2, L_3, \dots, L_n$ 。例如，在图5-6a所示的树中

$$L_1 = 5; L_2 = 8; L_3 = 12; L_4 = 13; L_5 = 10$$

记 L_{odd} 表示具有奇数编号的叶子，即 $L_1, L_3, L_5, L_7, \dots$ 。在 L_{odd} 的元素中，有些是它们相应的双亲的左孩子，有些是右孩子，我们对左孩子的元素做rake运算，然后再对其他元素做rake运算。

例如, 图5-6a所示的树中, $L_{\text{odd}} = (L_1, L_3, L_5) = (5, 12, 10)$ 。在 L_{odd} 中, 5是它的双亲的右孩子, 12和10是它们的双亲的左孩子, 我们首先同时对12和10做rake运算, 然后再对5做rake运算。这个过程用图5-7所示的树来描述。图5-7a表示有10个叶节点的树, 在第1次迭代中, L 有8个叶子:

$$L = (12, 14, 15, 5, 8, 16, 18, 19)$$

$$L_{\text{odd}} = (12, 15, 8, 18)$$

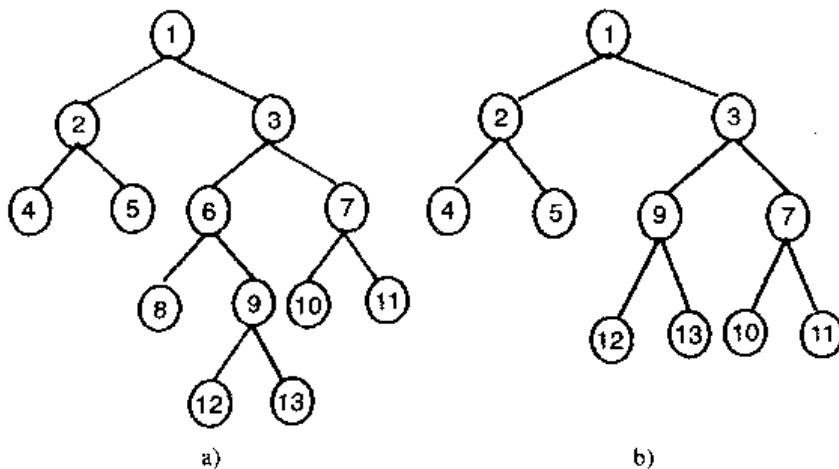


图 5-6

a) rake运算前 b) rake运算后

在 L_{odd} 的成员中, 12, 8和18是左孩子, 15是右孩子, 因此我们首先对12, 8和18同时执行rake运算, 然后再对15执行rake运算(见图5-7a, b和c)。在每个图中, 下一次将要进行rake运算的顶点用粗体表示。在第2次迭代中, L 中有4个元素:

$$L = (14, 5, 16, 19)$$

$$L_{\text{odd}} = (14, 16)$$

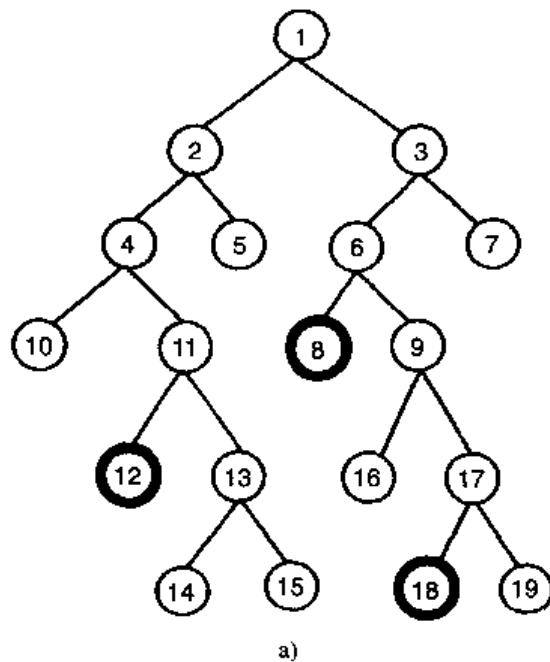


图 5-7

a) 树。第1次迭代, $L = (12, 14, 15, 5, 8, 16, 18, 19), L_{\text{odd}} = (12, 15, 8, 18)$

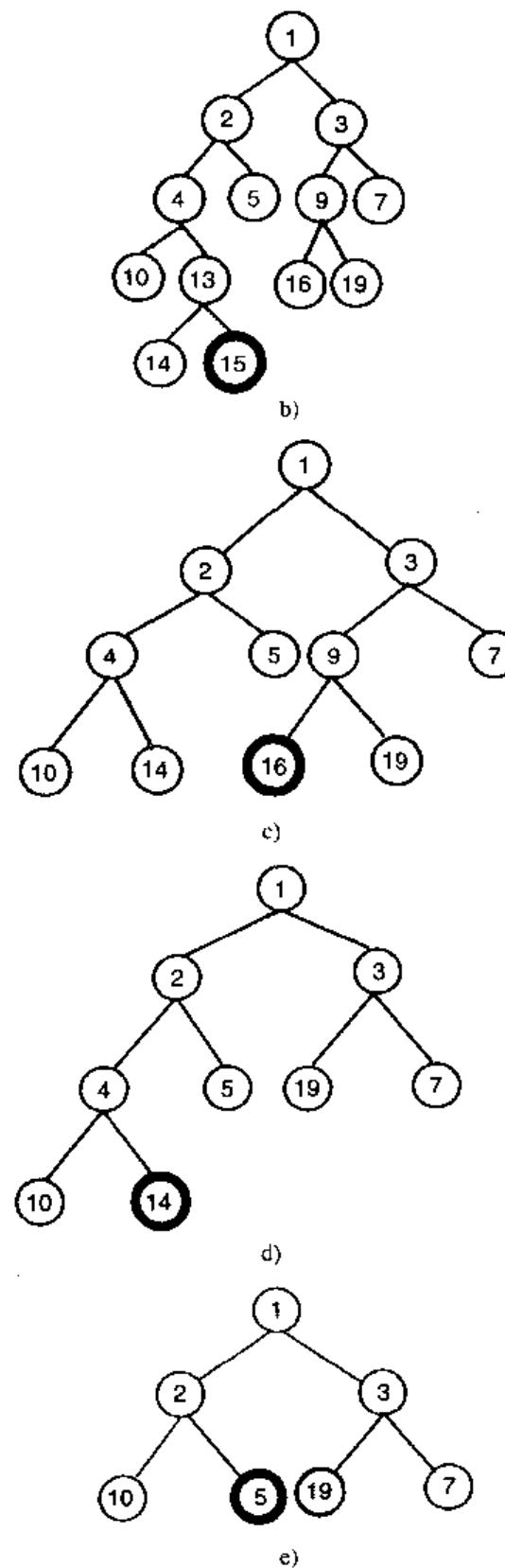


图5-7 (续)

- b) 对12, 8, 18同时做完rake运算后, 对15做rake运算 c) 对15做完rake运算后, 执行第2次迭代,
 $L = (14, 5, 16, 19), L_{odd} = (14, 16)$, 对左孩子16做rake运算, 然后对右孩子14做rake运算
 d) 对16做完rake运算后, 再对14做rake运算 e) 对14做完rake运算后, 第3步
 迭代中, 我们有新的值 $L = (5, 19), L_{odd} = (5)$, 对5做rake运算

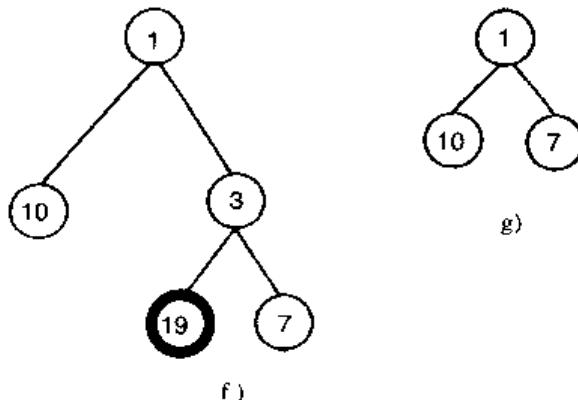


图5-7 (续)

f) 对5做完rake运算后，做最后一次迭代， $L = (19)$, $L_{\text{odd}} = (19)$ ，对19做rake运算

g) 对19做完rake运算后，我们得到仅含有三个节点的树

这里16是左孩子，14是右孩子。因此，我们首先对16做rake运算，然后再对14做rake运算（见图5-7c, d和e）。在第3次迭代中， $L = (5, 19)$, $L_{\text{odd}} = (5)$ ，我们仅对5做rake运算（见图5-7e和f）。在最后一次迭代中， $L = (19)$, $L_{\text{odd}} = (19)$ 。对19做完rake运算后，我们得到有三个节点的完全二叉树（见图5-7g）。

我们注意到，开始时 L 中有8个元素，经过每一次迭代， L 中的元素减半，因此如果最初 L 中有 n 个叶节点，我们得到有三个节点的完全二叉树共需要 $O(\log n)$ 次迭代，每一次迭代包含两步，每一步用 $O(n)$ 个处理器执行 $O(1)$ 时间。因此树的收缩算法总的时间复杂度是用 $O(n)$ 个处理器，执行时间为 $O(\log n)$ ，执行EREW PRAM模型。现在我们给出树的收缩算法的描述。

算法Tree Contraction

输入：

1. 有根二叉树 T ，其每一个非叶子节点恰有两个孩子
2. 对每个顶点 v 的 $p(v)$ 和 $s(v)$

输出：由原始树收缩得到的三节点完全树

1. Do the following $\log(n + 1)$ iterations

 1.1 Number the leaf nodes from left to right, leaving the leftmost and the rightmost node.

 Let L denote this ordered list of leaves, which are left children

 1.2 Apply the rake operation concurrently on all the nodes of L_{odd} which are right children

2. End Tree Contraction

现在我们研究树收缩算法的一个很重要的应用——算术表达式的计算（arithmetic expression evaluation）。

5.8 算术表达式的计算

我们已经清楚如何将一个算术表达式表示成二叉树（见图5-4）。从树的最底层计算表达式的值，这个bottom-up方法进行表达式计算所需的时间为 $O(n)$ 。本节我们用树收缩算法，在

$O(\log n)$ 时间内计算表达式，所用处理器数为 $O(n)$ 。为了求出表达式的值 $\text{val}(T)$ ，我们首先确定以内部节点 u 为根的子表达式的值 $\text{val}(u)$ ，每一节点 u 对应一个实数对 (p_u, q_u) 。

如果 u 是一个节点，它的孩子是 v 和 w ，则 $\text{val}(u) = \{p_v \cdot \text{val}(v) + q_v\} * \{p_w \cdot \text{val}(w) + q_w\}$ ，这里*表示节点 u 处的运算符(见图5-8)。例如，考虑如图5-9所示的树，表达式的值是 $\{8 * 5 + 3\} * \{(-2) * 1 + 20\} = (43) * (18) = 774$ 。[160]

对于一般的树，对每一个节点置 $(p, q) = (1, 0)$ 。当对 v 做rake运算时，在得到的树中我们要调整节点的 (p, q) 值，这样使得表达式最后的值不受影响。考虑图5-10所示的树，假定在节点 u 的运算是*，叶节点 u 含有一个常数 c 。

现在对节点 v 做rake运算，做完rake运算后，得到如图5-10b所示的树。 w 的值与rake运算之前的 X 的值相同，但是 (p_w, q_w) 变成了 (p'_w, q'_w) 。

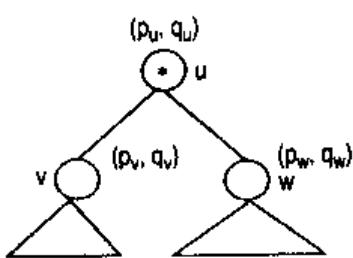
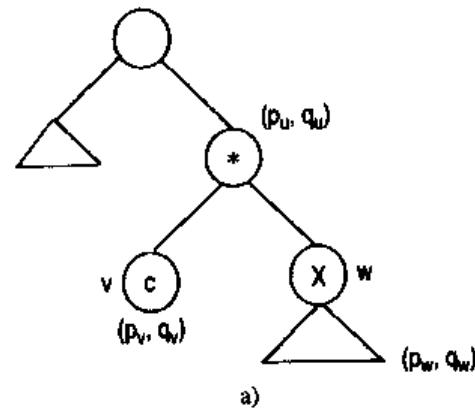
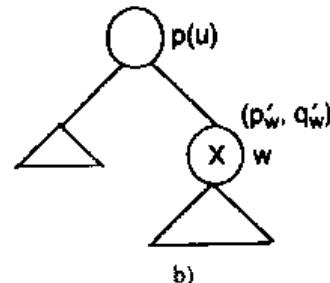


图5-8 有 (p, q) 值的树



a)



b)

图 5-10

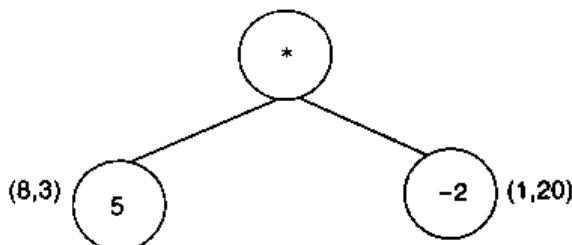


图5-9 有 (p, q) 值的小树

a) rake运算之前的树 b) 对节点 v 做
rake运算后的树

因此，在对 v 做rake运算时 X 的值不必已知。我们将选择 p'_w 和 q'_w ，使得表达式的值不受影响。在rake运算之前，节点 u 对其双亲的值所作的贡献为 $p_v * \text{val}(u) + q_v$ ，即

$$\begin{aligned} & p_u * \{p_v * c + q_v\} * \{p_w * X + q_w\} + q_u \\ &= \{p_u * (p_v * c + q_v)\} * p_w * X + \{p_u * (p_v * c + q_v)\} * q_w + q_u \end{aligned}$$

如果在rake运算后我们想保持同一个值，则有下列等式：

$$\begin{aligned} p'_w &= p_w * (p_v * c + q_v) * p_w \\ q'_w &= p_w * (p_v * c + q_v) * q_w + q_w \end{aligned}$$

在这个过程中 w 和 X 的值不受影响。

假定 u 含有 $+$ 运算，则类似地可以得到

$$\begin{aligned} p_w' &= p_u * p_w \\ q_w' &= p_u * (p_v * c + q_v) * q_w + q_u \end{aligned}$$

利用上面的公式， p_w' 和 q_w' 可在 $O(1)$ 时间内求出，因此利用改进的树收缩算法来计算算术表达式的值需 $O(n)$ 个处理器，执行时间为 $O(\log n)$ ，这里 n 表示表达式中运算数的个数。

这个过程用图5-11a~g来描述。图5-11a表示算术表达式，这类似于图5-7a中所示的树。用于图5-7a~g的rake运算，按相同顺序，可用于图5-11a~g。粗体节点表示下一次要做rake运算的节点。我们注意到，在rake运算后(p, q)改变了。在每一阶段，都可以验证树的值是2925。

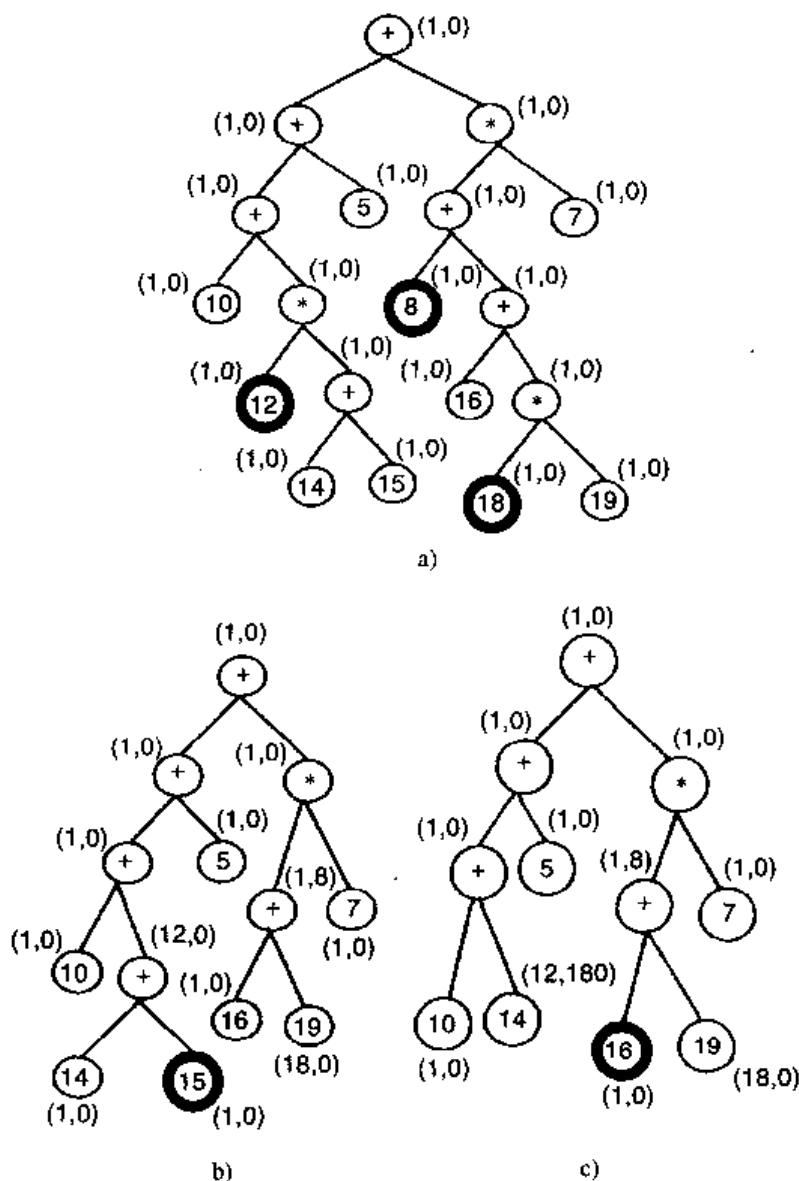


图 5-11

a) 算术表达式为 $\text{val}(T) = \{((10+12*(14+15))+5)+{(8+(16+18*19))*7}\}=2925$ 的树 T

b) 对12,8,18执行rake运算后 c) 对15执行rake运算后

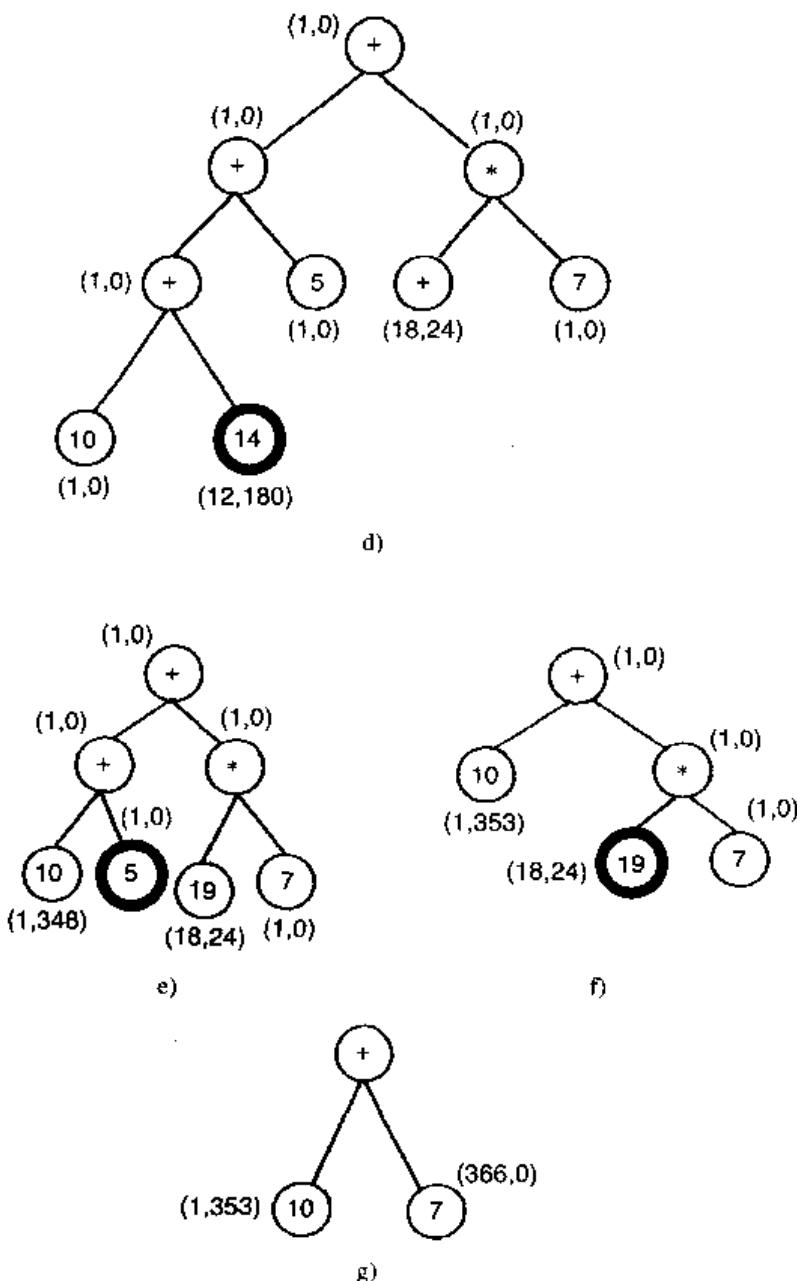


图5-11 (续)

d) 对16执行rake运算后 e) 对14执行rake运算后 f) 对5执行rake运算后

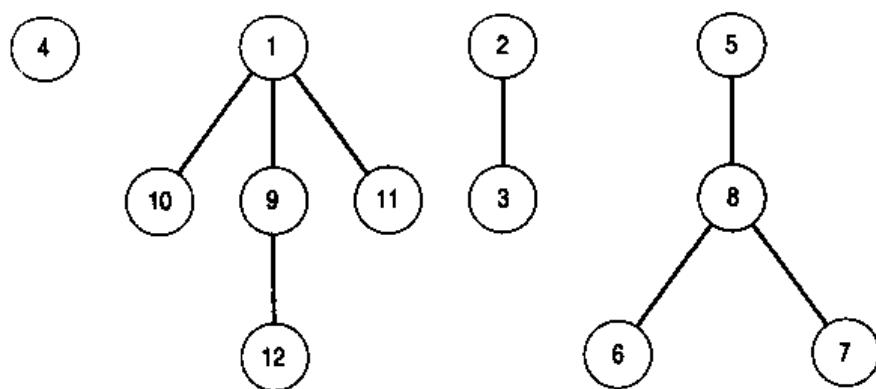
g) $\text{val}(T) = \{10+353\} + \{7*366\} = 2925$

5.9 森林求根问题

我们知道树是无圈的连通图，而无圈的图称作森林。换句话说，树的集合称作森林。图5-12表示有12个节点的森林。162

记 $R(v)$ 为树的根，其中节点 v 是该树的成员。表5-7中 R 的值为图5-12中各节点的 R 值。

我们将使用指针跳转法来设计一个算法，并利用这个算法来求每个节点的 $R(v)$ 值。对每个节点 v ，开始时并行地选取 $R(v) = \text{parent}(v)$ 。如果 $R(v)$ 是根，则必有 $R(R(v)) = R(v)$ 。直到我们得到 $R(R(v)) = R(v)$ ，并记 $R(v) = R(R(v))$ 。如图5-13所示的森林，表5-8给出了每一阶段的 $R(i)$ 的值。



163

图5-12 森林和它的双亲关系

表5-7 图5-12所示的森林的根数组

v	1	2	3	4	5	6	7	8	9	10	11	12
$R(v)$	1	2	2	4	5	5	5	5	1	1	1	1

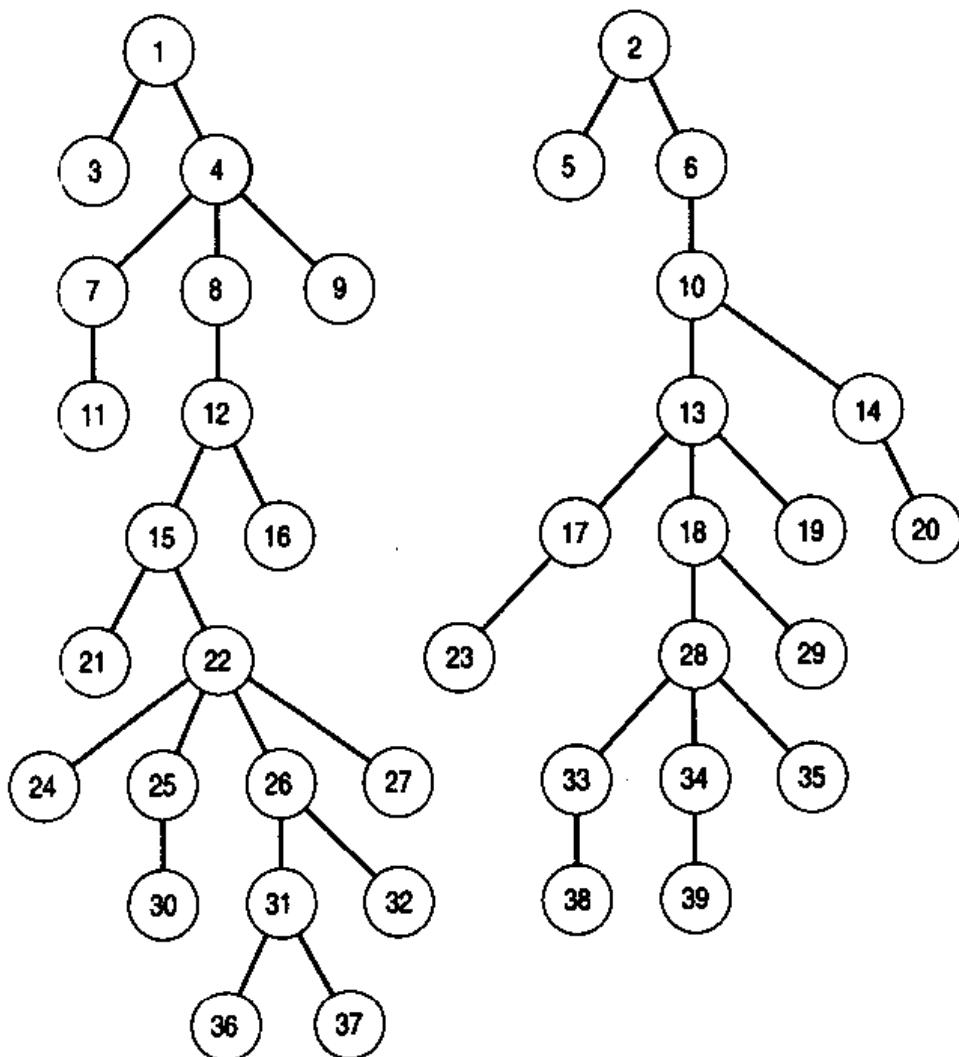


图5-13 含有39个节点的森林

表5-8 图5-13所示的森林求根问题

v	$R(v)$				v	$R(v)$			
	初始	阶段1	阶段2	阶段3		初始	阶段1	阶段2	阶段3
1	1	1	1	1	21	15	12	4	1
2	2	2	2	2	22	15	12	4	1
3	1	1	1	1	23	17	13	6	2
4	1	1	1	1	24	22	15	8	1
5	2	2	2	2	25	22	15	8	1
6	2	2	2	2	26	22	15	8	1
7	4	1	1	1	27	22	15	8	1
8	4	1	1	1	28	18	13	6	2
9	4	1	1	1	29	18	13	6	2
10	6	2	2	2	30	25	22	12	1
11	7	4	1	1	31	26	22	12	1
12	8	4	1	1	32	26	22	12	1
13	10	6	2	2	33	28	18	10	2
14	10	6	2	2	34	28	18	10	2
15	12	8	1	1	35	28	18	10	2
16	12	8	1	1	36	31	26	15	1
17	13	10	2	2	37	31	26	15	1
18	13	10	2	2	38	33	28	13	2
19	13	10	2	2	39	34	28	13	2
20	14	10	2	2					

算法Find-Root

输入：由数组PARENT(1:n)表示的森林

输出： $R(1:n)$

BEGIN

1. For $i = i$ to n do in parallel
2. $R(i) = \text{PARENT}(i)$
3. while $R(i)$ and $R(R(i))$ are not equal do
4. $R(i) = R(R(i))$
5. Endwhile
6. End Parallel

END

164

复杂度分析 显然所需的处理器数为 $O(n)$ 。第3~5步的For循环至多重复 $\lceil \log n \rceil$ 次，这是因为：开始时 $R(v) = \text{PARENT}(v)$ ，因此 $R(v)$ 给出与 v 距离为1的 v 的祖先。当我们对 $R(R(v))$ 赋值时，距离加倍。因此树的总高度就是被遍历的次数 $\lceil \log h \rceil$ ，这里 h 是树的高度。但是由于 $h < n$ ，因此算法的计算时间为 $O(\log n)$ 。当执行第4步时，有两个读操作，一个是 $R(i)$ ，另一个是 $R(R(i))$ 。对于两个不同的 i ， $R(i)$ 的值有可能相等，因此会发生并发读操作，而并发写操作不可能发生。因此这个算法执行CREW PRAM模型。

5.10 到根的路

作为指针跳转法的另一个例子，我们给出求任意一个树的每个节点与根之间路的问题。

假定 $T = (V, E)$ 是以 r 为根的有根树，并且用双亲关系来表示。

如果 v 不是树根， $\text{PARENT}(v)$ 表示树中节点 v 的双亲；如果 $v = r$ 是树根，则 $\text{PARENT}(v) = r$ 。

最初我们只有从每个节点到它双亲的长度为1的路，在下一阶段，所有的指向根方向长度至多为2的路被求出，然后所有的指向根方向长度至多为4的路被求出，继续这个过程，从每个节点到根的所有路经过 $\lceil \log h \rceil$ 步被求出，这里 h 是树的高度。路用变量 PATH 来表示， $\text{PATH}(v)$ 表示从节点 v 到根的路，它由一系列相邻节点组成。在第1阶段，假定

$$\text{PATH}(1) = (1, 2, 7)$$

$$\text{PATH}(7) = (7, 9, 3)$$

我们记

$$\text{PATH}(1) // \text{PATH}(7) = (1, 2, 7, 9, 3)$$

记号 // 用于连接具有公共端点的两条路，以形成一条较长的路。我们还定义另一变量 $\text{ANC}(v)$ （祖先）。最初有 $\text{ANC}(v) = \text{PARENT}(v)$ ，并且 $\text{PATH}(v) = (v, \text{PARENT}(v))$ 。

下一阶段，用下面的运算来求从 v 开始指向根方向的较长的路：

$$\text{PATH}(v) = \text{PATH}(v) // \text{PATH}(\text{ANC}(v))$$

更新 ANC 的值：

$$\text{ANC}(v) = \text{ANC}(\text{ANC}(v))$$

这样我们可以求出具有较长距离的祖先。注意 v 到 $\text{ANC}(v)$ 的路已经建立起来，我们又一次将路的长度加倍

$$\text{PATH}(v) = \text{PATH}(v) // \text{PATH}(\text{ANC}(v))$$

对每个节点，用 $\lceil \log h \rceil$ 步，这个过程给出从节点到根的路，这里 h 是树的高度。下面用一个例子来描述上述过程。考虑图5-14a所示的树，树的高度为4，这个树用双亲关系表示出来。最初我们给出路和祖先的值，如图5-14b。图5-14c，图5-14d，图5-14e分别表示经过第1, 2阶段后的位置。第2阶段后， $\text{PATH}(1), \text{PATH}(2), \text{PATH}(3), \text{PATH}(4), \text{PATH}(5), \text{PATH}(6), \text{PATH}(9), \text{PATH}(10)$ 和 $\text{PATH}(12)$ 的长度与第1阶段的值相同，因为它们相应的 ANC 的值都是根3。我们仅对节点 v 延长这条路；这里， v 满足 $\text{ANC}(v)$ 不是树根这一条件。并行算法如下：

算法 Tree-Path

输入： 给定由每个顶点 v 的 $\text{PARENT}(v)$ 组成的树 $T = (V, E)$

输出： 对每个 $v \in V$ 的 $\text{PATH}(v)$

BEGIN

1. For all $v \in V$ do in parallel

$$\text{ANC}(v) = \text{PARENT}(v) \quad \text{PATH}(v) = (v, \text{PARENT}(v))$$

End Parallel

2. Repeat the following steps $\lceil \log h \rceil$ times where h is the height of the tree. If h is not given its maximum value $n-1$ must be considered where n is number of vertices

2.1 For all $v \in V$ do in Parallel

2.2 If $\text{ANC}(v)$ is not root

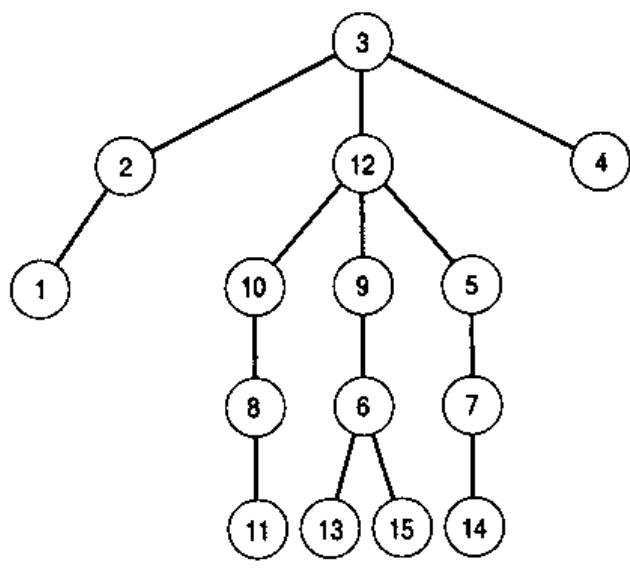
$$\text{PATH}(v) = \text{PATH}(v) // \text{PATH}(\text{ANC}(v))$$

$$\text{ANC}(v) = \text{ANC}(\text{ANC}(v))$$

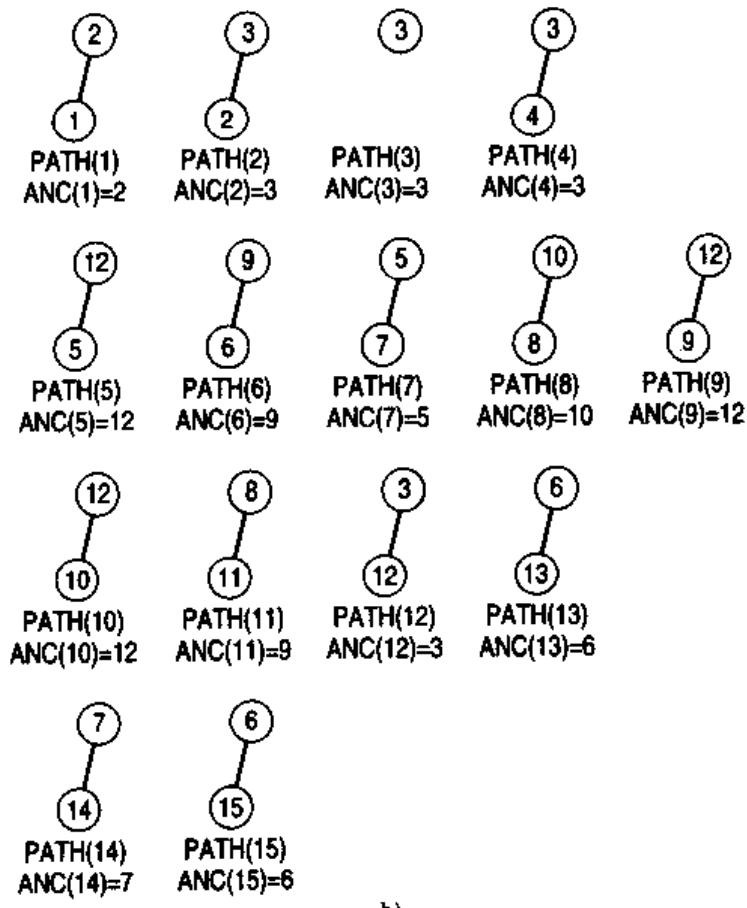
2.3 Endif

2.4 End Parallel

END Tree Path



a)



b)

图 5-14

a) 树 T b) 最初阶段的 T

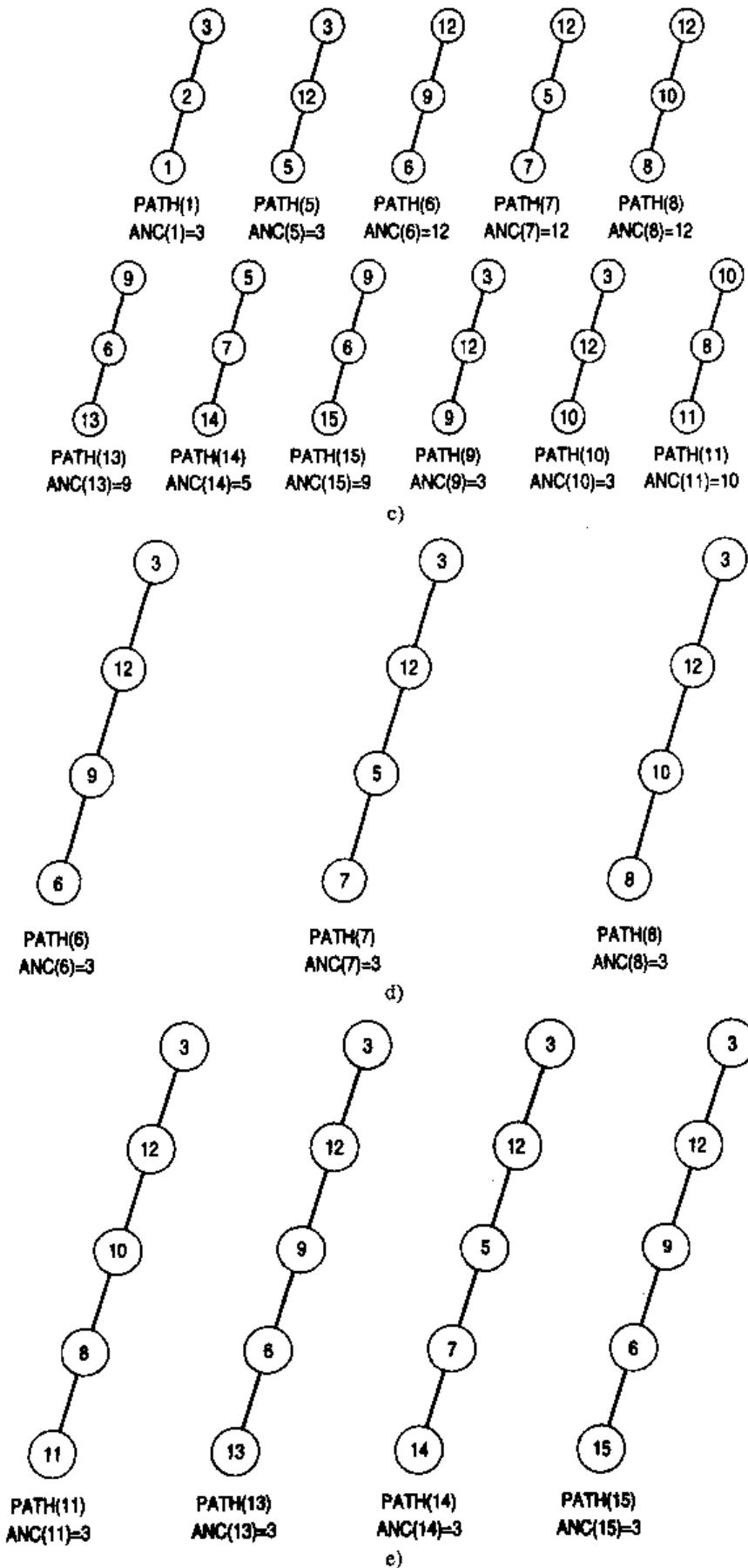


图5-14 (续)

c) 第1阶段的一些路 d) 第2阶段的一些路 e) 第2阶段的一些路

这个算法需 $O(n)$ 个处理器，执行时间为 $O(\log h)$ 。由于 $\text{ANC}(v)$ 可能对多个节点 v 而言其值相同，因此，该算法执行CREW PRAM模型。

5.11 树变为二叉树

表示树的常用数据结构是一维数组PARENT，它给出了顶点的双亲表示。168

如果 u 不是根， $\text{PARENT}(u) = u$ 的双亲；如果 u 是根， $\text{PARENT}(u) = u$ 。

例如，图5-15所示的树，表5-9给出了其双亲关系，但是这个数据结构并不能很有效地处理树的结构。因此我们需要将树转变成二叉树。只有转变成二叉树后，运算才能有效地执行。为了便于操作，我们用 $1, 2, \dots, n$ 来表示顶点。对于任一顶点 i ，它的孩子的指标从左到右依次增加，因此最左边的孩子有最小的指标。将一个树变成二叉树的方法如下：设 i 是一个顶点， i_1, i_2, \dots, i_k 是 i 的孩子，满足 $i_1 < i_2 < i_3 < \dots < i_k$ 。在二叉树中，

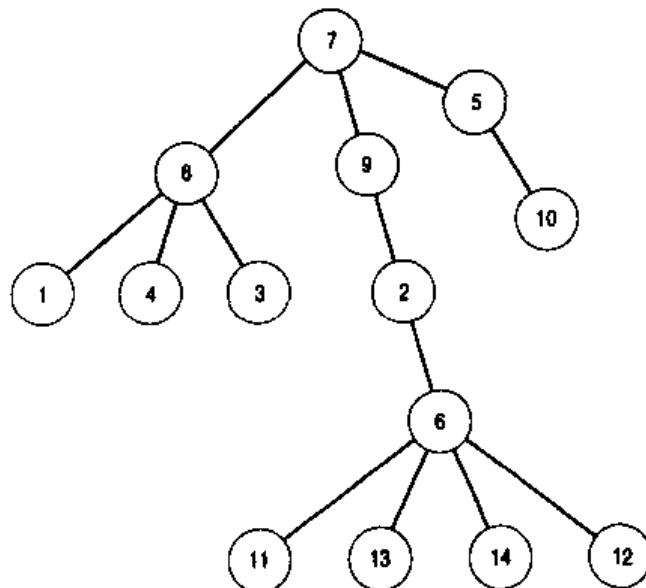
i 的左孩子是 i_1 ；

i_1 的右孩子是 i_2 ；

i_2 的右孩子是 i_3 ；

...

i_{k-1} 的右孩子是 i_k 。



169

170

图5-15 树及它的双亲关系

表5-9 双亲关系

顶点	双亲	顶点	双亲
1	8	8	7
2	9	9	7
3	8	10	5
4	8	11	6
5	7	12	6
6	2	13	6
7	7	14	6

上述过程见图5-16，二叉树用3个一维数组PARENT、LCHILD和RCHILD表示。

171

$LCHILD(i)$ = 二叉树中*i*的左孩子

$RCHILD(i)$ = 二叉树中*i*的右孩子

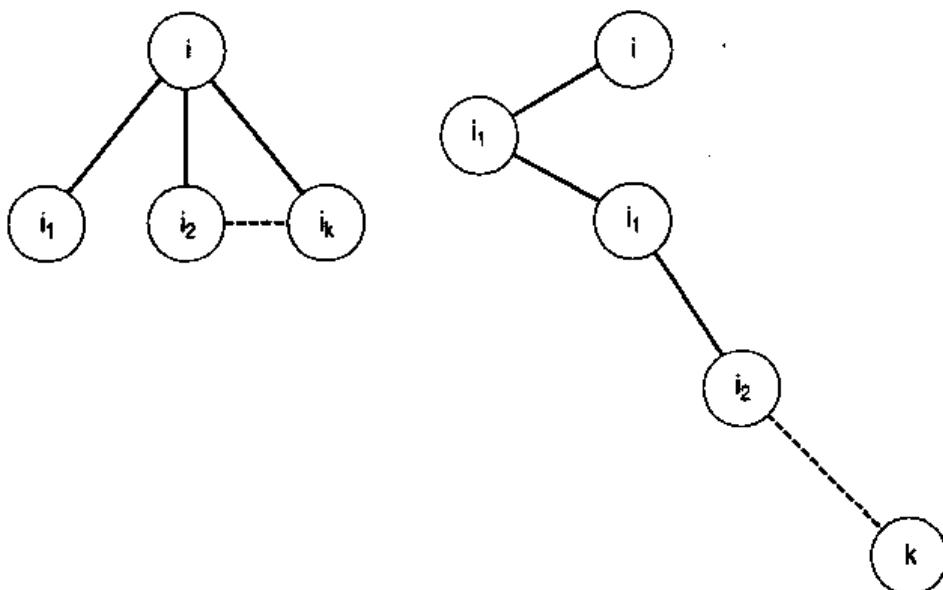


图5-16 树变为二叉树

当一个树转变成二叉树时，我们已经看到具有最小指标的*i*的孩子被选为*i*的左孩子。顶点*i*的右边是*i*的兄弟，具有大于*i*的最小指标。

$LCHILD(i)$ = *i*的具有最小指标的孩子

$RCHILD(i)$ = 具有大于*i*的最小指标的*i*的兄弟

更正式地，可写为：

$$LCHILD(i) = \begin{cases} j, & \text{如果 } j = \min \{k / \text{parent}(k) = i\} \\ 0, & \text{如果不存在顶点使得其双亲是 } i \end{cases}$$

$$RCHILD(i) = \begin{cases} j, & \text{如果 } j = \min \{k > i / \text{parent}(i) = \text{parent}(k)\} \\ 0, & \text{如果不存在顶点使得 } \text{parent}(i) = \text{parent}(k) \end{cases}$$

图5-17表示一个树及它的二叉树表示，表5-10给出了PARENT,LCHILD和RCHILD的值。

172

为了求出LCHILD和RCHILD的值，必须首先将顶点按升序排序，用这种方式使得LCHILD和RCHILD的值很容易求出。输出LCHILD和RCHILD数组的并行算法如下：

算法Binary-Tree

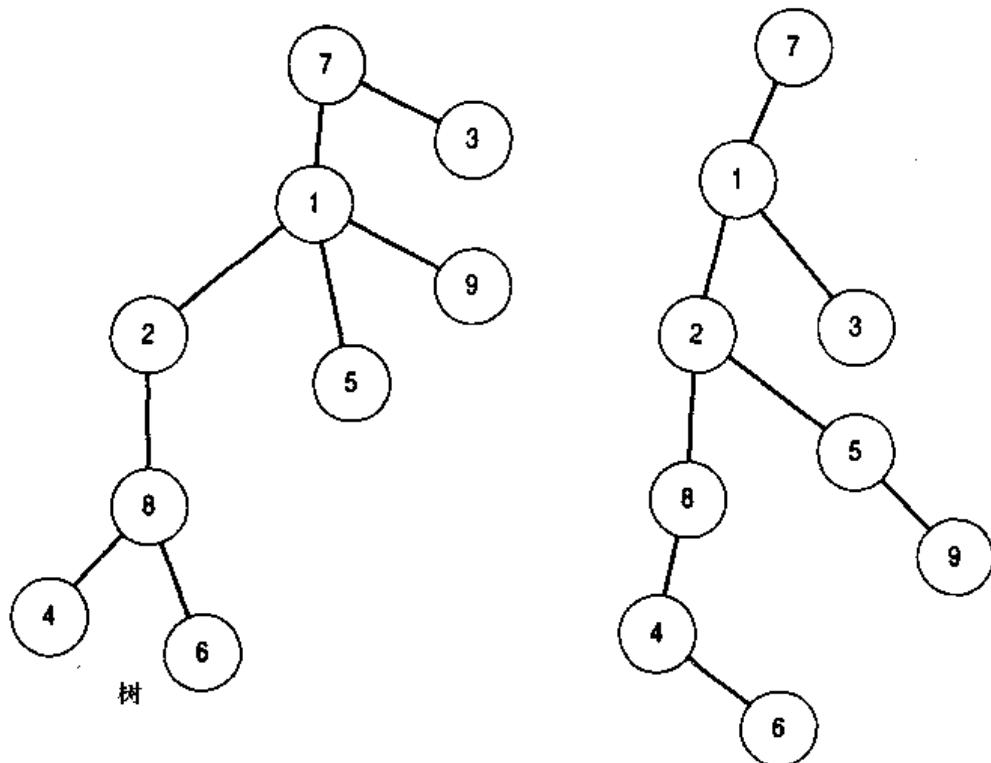
输入：PARENT(1:n)

输出：LCHILD(1:n), RCHILD(1:n)

BEGIN

1. Arrange the nodes in a convenient order so that the evaluation can be done efficiently
2. For $i = 1$ to n do in parallel
3. Find $LCHILD(i)$ and $RCHILD(i)$, using the formula given above

4. End Parallel
END



173

图5-17 树及其二叉树表示

表5-10 双亲、左孩子和右孩子

节点 <i>i</i>	PARENT(<i>i</i>)	LCHILD(<i>i</i>)	RCHILD(<i>i</i>)
1	7	2	3
2	1	8	5
3	7	0	0
4	8	0	6
5	1	0	9
6	8	0	0
7	7	1	0
8	2	4	0
9	1	0	0

174

上面的算法用 $O(n)$ 个处理器，执行时间为 $O(\log n)$ ，且执行 CREW PRAM 模型。

5.12 顶点直径

本节我们研究求一个树的所有顶点的直径的并行算法。树 T 的顶点 v 的直径是指从 v 到 T 的所有其他顶点的最长路的长度。即

$$\text{dia}(v) = \text{Max } \{d(v,w), w \in T\}$$

这里 $d(v,w)$ 表示从 v 到 w 的最短距离。

算法是根据树的如下两个性质设计的，首先我们给出这两个性质的描述和证明。

定理5-4 假定 $T = (V, E)$ 是一个树， u, v 是 T 的两个顶点且对每一个顶点 $i \in V$ ，有

$$\text{Max } \{d(i, u), d(i, v)\} \leq d(u, v)$$

则对任意的 $i \in V$ ，有 $\text{dia}(i) = \text{Max } \{d(i, u), d(i, v)\}$ 。特别地，从 u 到 v 的路是 T 中的最长路之一。

证明：由简单的观察可知 u 和 v 都是叶节点。假定 i 和 j 是 T 的两个任意的顶点，我们将证明

$$d(i, j) \leq \text{Max } \{d(i, u), d(i, v)\}$$

如果上式成立，因为 j 是 T 的任意顶点，所以等式 $\text{dia}(i) = \text{Max } \{d(i, u), d(i, v)\}$ 成立。现在来证明

$$d(i, j) \leq \text{Max } \{d(i, u), d(i, v)\}$$

假定 w 是从 u 到 v 的路中最靠近 j 的顶点，根据假定，我们有 $d(j, v) < d(u, v)$ 。因而有

$$\begin{aligned} d(j, w) &= d(j, v) - d(v, w) \leq d(u, v) - d(v, w) \\ &\leq d(u, w) + d(v, w) - d(v, w) \\ &= d(w, u) \end{aligned}$$

175 类似地，利用

$$d(j, u) \leq d(u, v)$$

我们可以证明

$$d(j, w) \leq d(w, v)$$

我们将利用这两个结论来完成证明：

$$d(j, w) \leq d(w, u)$$

$$d(j, w) \leq d(w, v)$$

情形1

假设从 u 到 v 的路与从 i 到 j 的路有一些公共顶点，并注意到

$$d(i, w) + d(w, u) = d(i, u)$$

$$d(i, w) + d(w, v) = d(i, v)$$

由前一种情形，有

$$\begin{aligned} d(i, j) &= d(i, w) + d(w, j) \\ &\leq d(i, w) + d(w, u) \\ &= d(i, u) \end{aligned}$$

由后一种情形，有

$$\begin{aligned} d(i, j) &= d(i, w) + d(w, j) \\ &\leq d(i, w) + d(w, v) \\ &= d(i, v) \end{aligned}$$

因此有

$$d(i, j) \leq \text{Max } \{d(i, u), d(i, v)\}$$

情形2

假设从 u 到 v 的路与从 i 到 j 的路没有公共顶点。假定 j' 是从 i 到 j 的路上距 w 最近的顶点。容易看出

$$d(j', j) \leq d(j, w) \leq d(w, u)$$

因而

$$\begin{aligned} d(i,j) &= d(i,j') + d(j',j) \\ &\leq d(i,j') + d(w,u) \\ &\leq d(i,j') + d(j',w) + d(w,u) \\ &= d(i,u) \end{aligned}$$

类似地，可以证明 $d(i,j) \leq d(i,v)$ 。因此有

$$d(i,j) \leq \text{Max } \{d(i,u), d(i,v)\}$$

定理得证。

定理5-5 假定 T 是一个树， u 是 T 的一个顶点，则对于任一满足 $\text{dia}(u) = d(u,k)$ 的顶点 k ， T 中有一条最长路，其中 k 是它的一个终点。

证明：考虑 T 是以 u 为根的有根树。按照定理中 k 的选取知， k 是一个叶节点，假定 v, w 是 T 的两个顶点， j_1 是 v 和 w 的最低公共祖先。

情形1

假定 j 是 k 的祖先，由 k 的选取有

$$\begin{aligned} d(j,v) &\leq d(j,k) \\ d(j,w) &\leq d(j,k) \end{aligned}$$

[177]

注意到

$$d(v,k) = d(v,j) + d(j,k)$$

或

$$d(w,k) = d(w,j) + d(j,k)$$

因此

$$\begin{aligned} d(v,w) &= d(v,j) + d(j,w) \\ &\leq \text{Max } \{d(v,k), d(w,k)\} \end{aligned}$$

情形2

假定 j 不是 k 的祖先， j_1 是 k 和 j 的最低公共祖先，有

$$d(v,j) < d(v,j_1)$$

和

$$d(w,j) < d(w,j_1)$$

又根据 k 的条件，进一步有

$$d(v,j_1) \leq d(j_1,k)$$

因而

$$\begin{aligned} d(v,w) &= d(v,j) + d(w,j) \\ &< d(v,j_1) + d(w,j_1) \\ &\leq d(k,j_1) + d(w,j_1) \\ &= d(w,k) \end{aligned}$$

我们也可以证明 $d(v,w) \leq d(v,k)$ 。因而有 $d(v,w) \leq \text{Max } \{d(w,k), d(v,k)\}$ 。

由情形1和情形2，可以得到

$$d(v,w) \leq \text{Max } \{d(w,k), d(v,k)\}$$

从上面不等式立即可得到定理的结论。

下面我们给出求树的所有顶点的直径的算法。

算法Diameter

输入：树 $T = (V, E)$

输出： $\text{dia}(i), i \in V$

BEGIN

1. Choose an arbitrary vertex $i \in V$ and compute $\{d(i, j) / j \in V\}$
2. Find the maximum of the set $\{d(i, j) / j \in V\}$ and the vertex u , such that $d(i, u)$ is the maximum. So, $\text{dia}(i) = d(i, u)$
3. Compute $\{d(u, w) / w \in V\}$
4. Choose the maximum of $\{d(u, w) / w \in V\}$ and find the vertex $v \in V$, such that $\text{dia}(u) = d(u, v)$
5. Compute $\{d(v, i) / i \in V\}$
6. For each vertex $j \in V$ do in parallel
7. $\text{dia}(j) = \text{Max} \{d(u, j), d(v, j)\}$
8. End parallel

END

此算法需用 $O(n/\log n)$ 个处理器，执行时间为 $O(\log n)$ 。

5.13 最远邻居

假定 $T = (V, E)$ 是一个树，且每一条边有一个正实数权重。如果 $\text{dia}(u) = d(u, v)$ ，则顶点 v 称为顶点 u 的最远邻居，这里 $d(u, v)$ 表示从 u 到 v 的路上所有边的权重之和， $\text{dia}(u)$ 表示 u 的直径。注意，一个顶点可能不止一个最远邻居。例如，图5-18所示的树，顶点 u 有三个最远邻居 x, y 和 z 。对于任何一个图，可以通过计算图的相邻矩阵的传递闭包（transitive closure）来计算每个顶点的最远邻居。传递闭包的计算所用的处理器数为 $O(n^3)$ ，执行时间为 $O(\log^2 n)$ 。

179

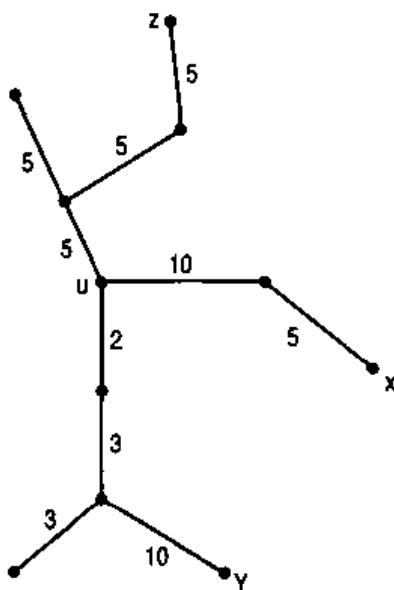


图5-18 u 的最远邻居 x, y 和 z

但是对于树来说，Ghosh和Maheswari (1992)已经证明，如果一个树的最长路位于顶点 u 和 v 之间，那么每个顶点的最远邻居或者是 u ，或者是 v 。利用这个性质，我们设计了一个有效的并行算法。

首先定义几个记号：

$P(u,v)$ 表示树中从 u 到 v 的惟一的一条路；

$D(u,v) = u$ 和 v 之间的距离

$=P(u,v)$ 的边的权重和；

$P(u,v) \cap P(x,y)$ 表示 $P(u,v)$ 和 $P(x,y)$ 公共的极长路。

树 T 的直径为树的最长路，即含有最大权重的路。一个树不止一条直径，但是都有相同的长度。下面给出一些有趣的性质。

定理5-6 如果 u_1 和 v_1 分别是 T 中 u 和 v 的最远邻居，则 $P(u,u_1)$ 和 $P(v,v_1)$ 必相交。

证明：假定 $P(u,u_1)$ 和 $P(v,v_1)$ 不相交（见图5-19）。考虑从 u 到 v 的路， $P(u,v_1)$ 经过 $P(u,x)$ ，然后分开，并且交 $P(v,v_1)$ 于 y 点，由观察得到

$$D(x,u_1) \geq D(x,y) + D(y,u_1)$$

$$D(y,v_1) \geq D(y,x) + D(x,v_1)$$

由这两个不等式，得：

$$D(x,y) < 0$$

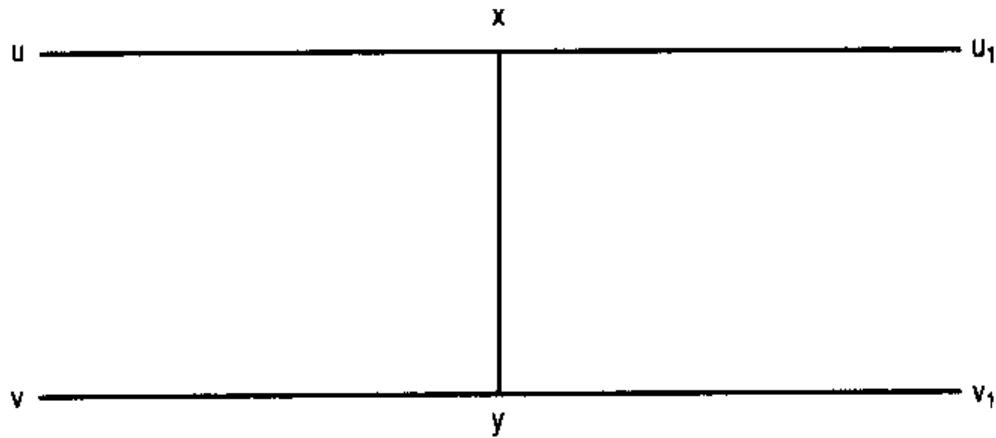


图5-19 $P(u, u_1)$ 和 $P(v, v_1)$ 不相交

这与已知相矛盾，因为每一条边都含有正的权重。因此 $P(u,u_1)$ 和 $P(v,v_1)$ 必相交。

推论 一个顶点和它的最远邻居间的路总是与树的所有直径相交。

定理5-7 假定 $P(u,v)$ 是树 $T = (V,E)$ 的一个直径，则对于任一顶点 $x \in V$ ，或者 u 是 x 的最远邻居，或者 v 是 x 的最远邻居。

证明：假定 $P(u,v)$ 是一直径， $x \in V$ 是任一顶点， y 是 x 最远邻居。我们只要证明

$$D(x,y) = D(x,u)$$

或

$$D(x,y) = D(x,v)$$

由推论， $P(x,y)$ 和 $P(u,v)$ 相交，记

$$P(x,y) \cap P(u,v) = P(p,q)$$

p 是 $P(u,v)$ 中的顶点，我们得到

$$D(q,v) > D(q,y)$$

因为 $P(u,v)$ 是直径，又因为 y 是 x 的最远邻居，有

$$D(q,y) > D(q,v)$$

因而得到

$$D(q,y) = D(q,v)$$

因此有

$$D(x,y) = D(x,v)$$

定理得证。

如果已知树 T 的直径 $P(u,v)$ ，通过比较 $D(x,u)$ 和 $D(x,v)$ ，可以求出 T 的每个顶点 x 的最远邻居，

因此，必须首先求直径，定理5-5给出了求直径的过程。下面我们给出求树的直径的过程：

1. x 是任意顶点；
2. y 是 x 的最远邻居；
3. z 是 y 的最远邻居；
4. $P(y,z)$ 是 T 的直径。

如果已经求出 $P(y,z)$ ，那么，对于任一顶点 $u \in V$ ， u 的最远邻居为

$$u = \begin{cases} y, & \text{如果 } D(y,u) > D(z,u) \\ z, & \text{否则} \end{cases}$$

可以证明上述方法需要 $O(n/\log n)$ 个处理器，执行时间为 $O(\log n)$ 。

参考文献

- Akl, S. G. (1989) *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, NJ.
- Samy, A. A., Arumugam, G., Devasahayam, M. and Xavier, C. (1991) Algorithms for Intersection Graphs of Internally Disjoint Paths in Trees, *Proceedings of National Seminar on Theoretical Computer Science*, IMSC, Madras, India, Report 115, pp. 169-178.
- Arnborg, S., Lagergren, J., and Seese, D. (1989) Problems Are Easy for Tree-Decomposable Graphs, in T. Lepistoo and A. Salomaa, ed., *15th ICALP*, pp. 335-351.
- Arikati, S. R., and Pandurangan, C. (1989) Linear Algorithms for Parity Path Problems on Circular Arc Graphs. Workshop on Algorithms and Data Structures, Ottawa, Canada.
- Albacea, E. A. (1994) Parallel Algorithm for Finding a Core of a Tree Network, *Information Processing Letters*, 51 223-226.
- Berge, C., and Chvatal, C. ed. (1984) Topics on Perfect Graphs, *Annals of Discrete Mathematics*, 21.
- Bondy, J. A., and Murty, U. S. R. (1976) *Graph Theory with Applications*, North-Holland, New York.
- Bandelt, H. J. and Mulder, H. M. (1986) Distance-Hereditary Graphs. *Journal of Combinatorial Theory, Series B*, 41, 182-208.

- Bouchet, (1988) Transforming Trees by Successive Local complementation. *Journal of Graph Theory*, **12**, 195–207.
- Chen, Z. Z. (1992) A Simple Parallel Algorithm for Computing the Diameters of All Vertices in a Tree and Its Applications. *Information Processing Letters*, **42**, 243–248.
- Ghosh, S. K., and Maheswari, A. (1992) An Optimal Parallel Algorithms for Computing the Furthest Neighbours in a Tree. *Information Processing Letters*, **44**, 155–160.
- Nykanen, M. and Ukkonen, E. (1994) Finding Lowest Common Ancestors in Arbitrarily Directed Trees. *Information Processing Letters*, **50**, 307–310.
- Wu, S. and Manber, U. (1988) Algorithms for Generalised Matching. Technical Report Tr 88-39, Department of Computer Science, University of Arizona.
- Wu, S. and Manber, U. (1992) Path Matching Problems. *Algorithmica*, **8**, 89–101.
- Xavier, C. and Arumugam, G. (1994) Algorithms for Parity Path Problems in Some Classes of Graphs. *Computer Science and Informatics*, **24(4)** 50–54.
- Xavier, C. (1995) Sequential and Parallel Algorithms for Some Graph Theoretic Problems (Ph.D. Thesis), Madurai Kamaraj University. Madurai, India.
- Yannakakis, M. (1981) Computing the Minimum Fill-In is NP-Complete. *SIAM J. Alg. and Disc. Methods*, **2**, 77–79.

习题

- 5.1 假定树 T 用它的相邻列表来表示。相邻列表仅是一个线性列表。设计一个算法来求欧拉环游。
- 5.2 树的前序遍历由根 r 的遍历及从左到右的根 r 的每一个子树的前序遍历组成。设计一个并行算法在 $O(\log n)$ 时间内得到每个顶点 v 的前序编号，利用 EREW PRAM 模型。
- 5.3 假定 $T = (V, E)$ 是一个树， $s(v)$ 表示 v 的下一个兄弟， $c(v)$ 表示 v 的第一个孩子。由这两个值，设计一个并行算法来求每个顶点 v 的双亲。
- 5.4 一个有根二叉树的中缀次序遍历先从根 r 的左子树的中缀次序遍历开始，接着是 r 的遍历，最后是 r 的右子树的中缀次序遍历。设计一个算法在 $O(\log n)$ 时间内求各个顶点的中缀次序编号。
- 5.5 一个二叉树中，各个顶点 v 有权重 $w(v)$ ，对于每个顶点 v ， $S(v)$ 表示以 v 为根的子树中所有顶点的权重和。设计一个算法求每个顶点 v 的 $S(v)$ 。
- 5.6 对于顶点 v ，如果去掉 v 后所得到的子树的大小小于或等于 n ，则称 v 为树 T 的形心。这里 n 表示树 T 的顶点数。设计一个算法在 $O(\log n)$ 时间内求 T 的形心。算法执行 EREW PRAM 模型。
- 5.7 假定 T 是一个树，我们想给树加根使得树的高度最小。设计一个并行算法来选取一个顶点 v ，并把它作为 T 的根，满足 T 具有最小高度。
- 5.8 假定 $T = (V, E)$ 是以 r 为根的有根树， $\text{size}(v)$ 表示以 r 为根的子树的顶点个数。设计一个并行算法求 $\text{size}(v)$ ， $v \in V$ 。
- 5.9 假定 $X(1:n) = (x_1, x_2, \dots, x_n)$ 是一列不同数的数组。如果 $i < j$, $a_i < a_j$ ，则称 (a_i, a_j) 是匹配对。设计一个并行算法来标识这个数组的所有匹配对。
- 5.10 假定 T 是一个有根树，对于每个顶点 v ， v 的后代个数记为 $\text{des}(v)$ 。设计一个并行算法来求每个顶点 v 的 $\text{des}(v)$ 。

第6章 图 算 法

许多实际生活情形用图来描述，甚至许多社会生活问题也用图来描述。例如一个机构里工作人员之间的人际关系可以用图来描述和研究。交通系统、商业系统、工作表、设备供给等问题都可以用图模型来描述和分析。计算机产生以后，图论研究得到飞速发展，图算法得到广泛应用。人们设计了各种图问题的有效算法。最近几年，关于图问题的并行算法的设计和分析成为一个非常有用的研究课题。

6.1 简单图算法

本节我们对许多简单图问题设计并行算法。首先考虑一些容易的图问题。让我们从确定顶点的度出发。考虑图问题时，我们假定数据结构在算法设计中扮演着重要角色。假定图用它的相邻矩阵表示。下面研究如何设计一个算法去求图的每个顶点的度。

定理6-1 假定 $G = (V, E)$ 是一个简单图， n 和 m 分别表示图的顶点和边的个数，图的顶点用整数 1 到 n 来表示。假定 A 是图 G 的相邻矩阵，则顶点 i 的度为

$$d(i) = \sum_{j=1}^n A(i, j)$$

证明：根据相邻矩阵的定义，

$$A(i, j) = \begin{cases} 1, & i \text{与 } j \text{ 相邻} \\ 0, & i \text{与 } j \text{ 不相邻} \end{cases}$$

因此， $\sum_{j=1}^n A(i, j)$ 给出了与 i 关联的边的数目。定理得证。

考虑图6-1的图，它的相邻矩阵为

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

顶点 i 的度是指相邻矩阵第 i 行的行元素和，即 $d(1) = 3$; $d(2) = 3$; $d(3) = 2$; $d(4) = 4$; $d(5) = 3$; $d(6) = 3$; $d(7) = 2$ 。下面的并行算法用来求图的每个顶点的度。

算法Degree

输入：相邻矩阵 $A(1:n, 1:n)$

输出：度数 $d(1:n)$

BEGIN

1. For $i = 1$ to n do in parallel

```

2.  $d(i) = \sum_{j=1}^n A(i,j)$ 
3. End parallel
END

```

185

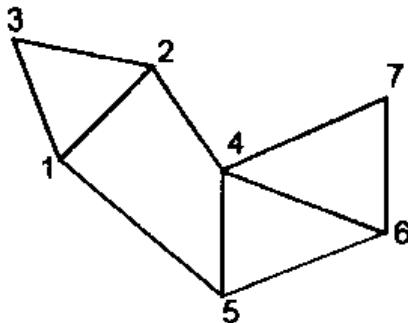


图6-1 图G

上面算法的第2步是求 n 个数的和，因而用 $O(n/\log n)$ 个处理器，执行时间为 $O(\log n)$ 。求图的所有顶点的度所用的处理器数为 $O(n^2/\log n)$ ，执行时间为 $O(\log n)$ 。注意第1~3步并行执行，用EREW PRAM模型。因此可用EREW模型来实现这个算法。

上面这个算法非常容易，但是经常被使用。现在我们讨论度算法的一个非常有趣的应用。

一个图称为欧拉图，当且仅当有一条闭通道经过图的每一条边且恰好经过一次。欧拉定理表明一个图是欧拉图，当且仅当它不含有奇数度的顶点。因此为了检验一个图是否是欧拉图，充要条件是检验它是否含有奇数度的顶点。下面的算法确定一个图是否是欧拉图。 eu 是一个布尔数组，如果顶点 v 的度是偶数，则 $eu(v) = \text{true}$ ；反之， $eu(v) = \text{false}$ 。如果对图的所有顶点 v 都有 $eu(v) = \text{true}$ ，则该图是欧拉图。

算法Euler

输入：图 $G = (V,E)$

输出：布尔变量EULER的值

BEGIN

1. Find the degree of each of the vertex of G
2. For each vertex v of G do in parallel
 3. If $d(v)$ is odd
 $eu(v) = \text{false}$
 else
 $eu(v) = \text{true}$
 endif
4. End Parallel
5. EULER = Boolean AND of all the $eu(v)$

END

上面算法中，第1步执行度算法，第2~4步所用的处理器数为 $O(n)$ ，执行时间为 $O(1)$ 。第5步执行 n 个布尔值的AND运算，这一步的复杂度依赖于PRAM模型：如果是EREW PRAM模型，则需要 $O(n)$ 个处理器，执行时间为 $O(\log n)$ ；如果是ERCW PRAM模型，则需要 $O(n)$ 个处理器，执行时间为 $O(1)$ 。这个算法的总的复杂度是用 $O(n^2)$ 个处理器，执行时间为 $O(\log n)$ 。

186

另一个度算法的应用是检验顶点集是否组成一个团。给定相邻矩阵 A , C 表示给定图 $G = (V, E)$ 的顶点子集。对任一顶点 v , $\sum_{u \in C} A(v, u)$ 给出了 C 中与 v 相邻的顶点数, 因此, C 是一个团, 当且仅当对 C 中的每个顶点 v , 有 $\sum_{u \in C} A(v, u) = |C| - 1$ 。下面算法能检验 C 是否形成一个团。在算法中, $dc(v)$ 表示 C 中与顶点 v 相邻的顶点个数。 C 是一个团当且仅当 $dc(v) = |C| - 1$, 因此, 如果 C 有 k 个顶点, 则 C 是一个团当且仅当对 C 中的每个顶点 v , $dc(v) = k - 1$ 。为了验证这个结论, 求 $d = \sum_{v \in C} dc(v)$ 。 d 等于 C 中边数的两倍。因此 C 是一个团当且仅当 $d = k(k - 1)$ 。现在给出这一算法:

算法Clique-EW

输入:

1. 相邻矩阵 A
2. 顶点集 C

输出: 布尔值CLIQUE

BEGIN (Let k denote $|C|$)

1. For each vertex $v \in C$ do in Parallel
 2. $dc(v) = \sum_{u \in C} A(u, v)$
 3. End Parallel
 4. $d = \sum_{v \in C} dc(v)$
 5. If $d = k(k - 1)$ then
 CLIQUE = True
else
 CLIQUE = False
- End if
- END

在这个算法中, 第1~3步用 $O(k^2)$ 个处理器, 执行时间为 $O(\log k)$; 第4步仅用 $O(k)$ 个处理器, 执行时间为 $O(\log k)$ 。因此, 这个算法采用EREW PRAM模型共需处理器数为 $O(k^2)$, 执行时间为 $O(\log k)$ 。如果PRAM模型是ERCW, 则需用处理器数为 $O(k^2)$, 执行时间为 $O(1)$ 。算法如下:

算法Clique-CW

输入:

1. 图 $G = (V, E)$ 的相邻矩阵 A
2. 顶点集 C

输出: 布尔值CLIQUE

BEGIN

Let k be the number of vertices in C

1. CLIQUE = True
2. For each $v, u \in C$ such that $v \neq u$ do in Parallel
3. If $A(v, u) = 0$ then

```

CLIQUE = False
End if
4. End Parallel
END

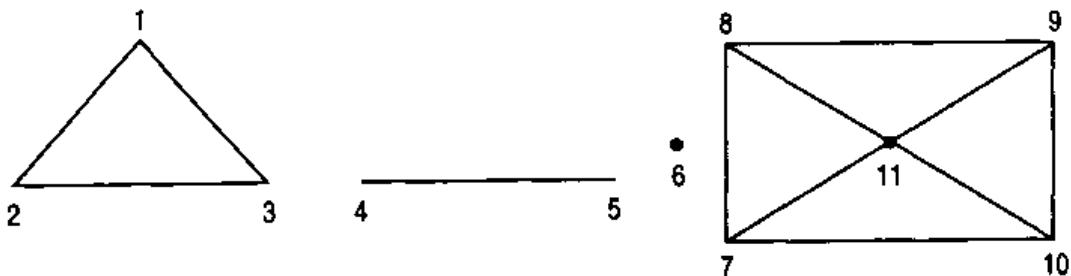
```

上面算法中，对于C的每一对顶点 u,v ，用独立的处理器验证是否有 $A(u,v) = 0$ 。如果有 $A(u,v) = 0$ ，则返回“false”值给布尔变量CLIQUE。这里，不止一个处理器试图写入CLIQUE的同一个位置。在并发写操作中，产生冲突时则可用任意冲突解决机制来解决。给定的顶点集可以在 $O(k^2)$ 个处理器上用 $O(1)$ 时间来检验是否它形成一个团，这里 k 是给定集合的顶点个数。现在我们讨论图论中的一个非常重要的问题。

6.2 并行连通度算法

广度优先搜索（Breadth-First Search）和深度优先搜索（Depth-First Search）是图的两个重要的搜索技术，它们也用来检验一个图是否是连通支。给定一个图，求出它的连通支是一个很有趣的问题。如图6-2所示，它有四个连通支。人们设计了许多并行算法来求无向图的连通支。所有这些算法都可以用下面两个主要策略对它们进行分类。

1. 所使用的基本技术；
2. 输入的格式。



[187]

[188]

图6-2 非连通树

目前的所有算法，使用的基本技术是下面三个中的一个：(1) 广度优先搜索(BFS)；(2) 传递闭包 (Transitive Closure)；(3) 顶点收缩 (Vertex Collapse)。输入的最常用的形式是相邻矩阵。相邻矩阵的广泛使用是因为它可以描述成图的理论问题并且可以看作矩阵问题来求解。对于稠密图这些策略可以很好的使用，但是对于稀疏图可能会产生无效算法。因此，我们经常输入相邻列表。而求连通支的大多数算法都是非常一般的算法，其中有一些算法适合稀疏图或稠密图。

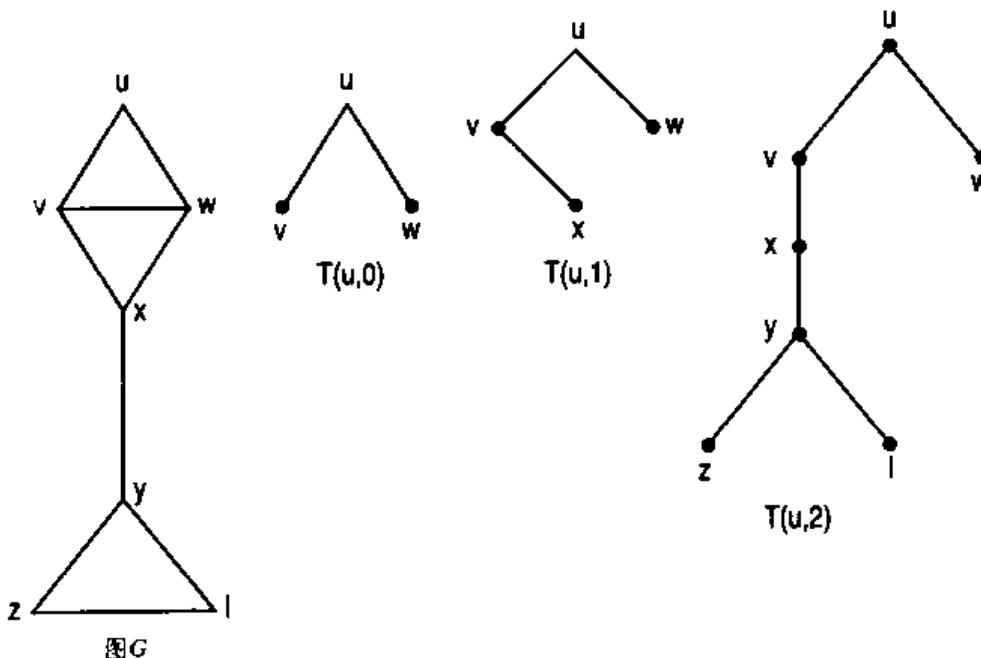
6.2.1 广度优先搜索 (BFS)

图的BFS首先访问一个顶点，记为 v ，访问完 v 后，再访问与 v 距离为1的所有顶点，然后是距离为2的所有顶点，如此继续下去，直到访问完所有的顶点。例如，考虑如图6-5a所示的图，如果BFS从1开始，首先访问1，然后访问2和3；由于4和5都与2和3相邻，再访问4和5，接下来访问6，没有更多的顶点与4和5相邻，因此访问与6相邻的点，即接下来访问7，然后访问与7相邻的8和9。为了了解如何并行执行BFS，下面介绍一些概念。

设 $G = (V, E)$ 是一个图， $x \in V$ ， k 是一整数且满足 $0 < k < \lceil \log n \rceil$ ，其中 n 是顶点个数。 $T(v, k)$

表示以 v 为根的树，它的所有的顶点从 v 开始经过一条小于或等于 2^k 长度的路到达，图6-3表示一个图及 $T(u,0)$, $T(u,1)$ 和 $T(u,2)$ 。又设 G 的子图 $T = (V_T, E_T)$ 是一个有根树，如果对于任意两个顶点 $u, v \in V_T$, $(u, v) \in E$, 要么它们位于 T 的同一层，要么位于 T 的相邻层，则称 T 具有BFS特性。图6-4描述了BFS特性。如果 $T(v,k)$ 具有BFS特性，则称 $T(v,k)$ 为 G 的 k -树，也称为 k -BFS树。下面我们考虑如何将BFS并行化。 $T(v,0)$ 以 v 作为根且以它的所有邻居作为孩子。对于 $T(v,0)$ 中的每一个顶点 u ，合并 $T(u,0)$ 到 $T(v,0)$ 中而得到 $T(v,1)$ 。一般地，对于任一个满足条件 $0 < k < \lceil \log n \rceil$ 的 k ，对于 $T(v,k-1)$ 最低层的每一个顶点 u ，合并 $T(u,k-1)$ 到 $T(v,k-1)$ 中，则 v 是所得树的根。下面的并行算法用来求任一顶点 v 的BFS支撑树 $T(v, \log n)$ 。

189



图G

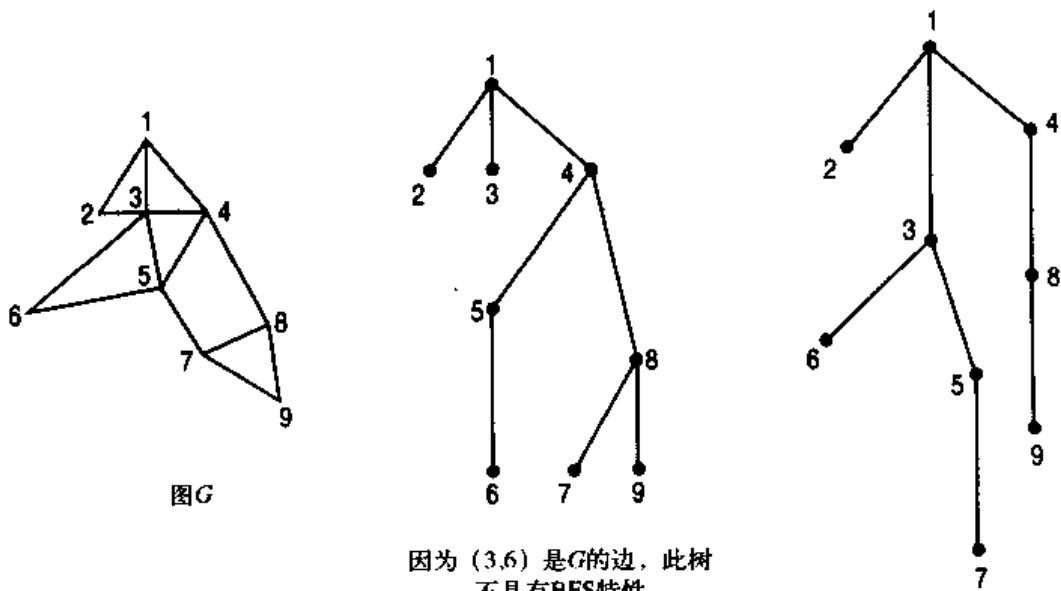
图6-3 图和 $T(u, i)$ 

图6-4 图和具有BFS特性的树

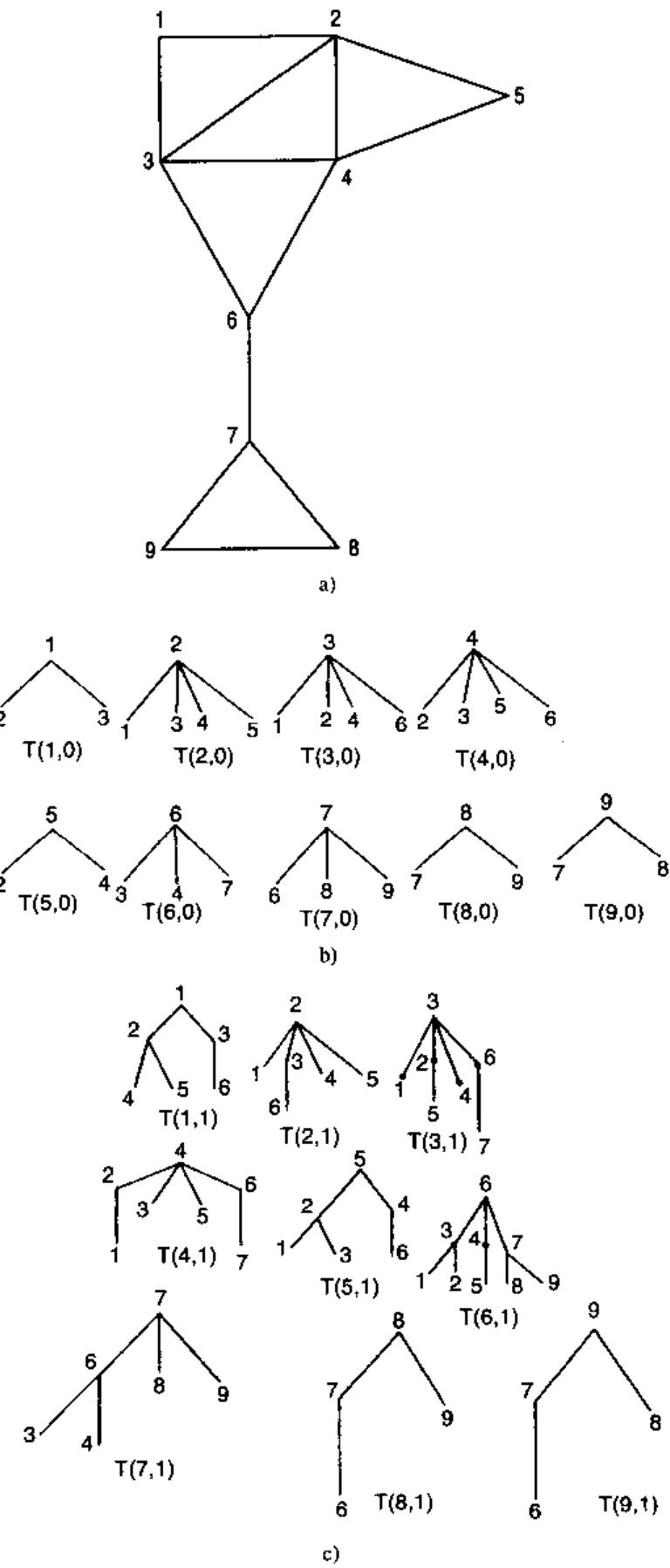


图 6-5

a) 无向图 G b) $T(v, 0)$, $v = 0, 1, 2, \dots, 9$ c) $T(v, 1)$, $v = 1, 2, \dots, 9$

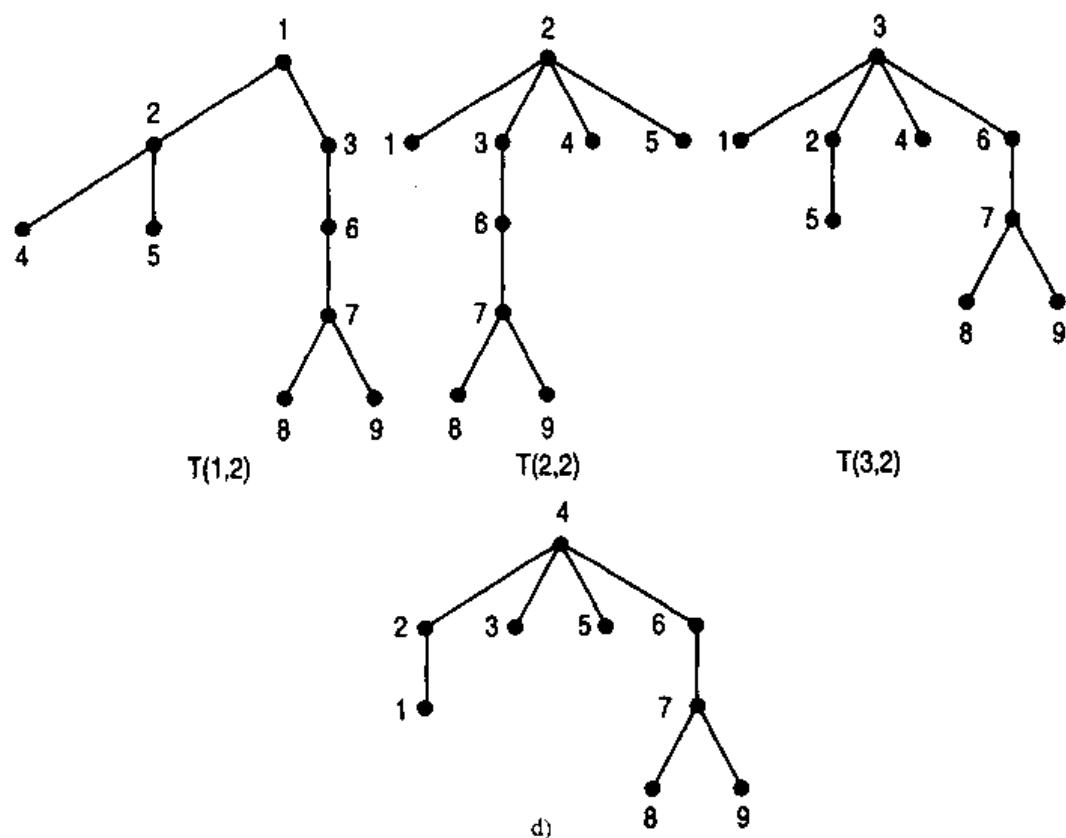


图6-5 (续)

d) $T(v, 2)$, $v = 1, 2, 3, 4$

190

算法BFS**输入：**图 $G = (V, E)$ **输出：**从每个顶点开始的 G 的 BFS 支撑树**BEGIN**1. For every vertex $v \in V$ do in ParallelFind $T(v, 0)$

End Parallel

2. $k = 0$ 3. While $k < \lceil \log n \rceil$ do $k = k + 1$ For every vertex $v \in V$ do in ParallelFor every vertex u at the lowest level of $T(v, k-1)$ do in ParallelMerge $T(u, k-1)$ with $T(v, k-1)$ such that v is the root of the resulting tree which is denoted by $T(v, k)$

End Parallel

End Parallel

END

图6-5a~d具体描述了上述算法。合并树 $T(u, k-1)$ 到 $T(v, k-1)$ 共需时间为 $O(\log n)$ ，算法的 while 循环共执行 $\log n$ 次，因此该算法共需时间为 $O(\log^2 n)$ 。在算法中有两个嵌套的并行循环，

可以证明合并操作需要 $O(n)$ 个处理器。因此算法共需要 $O(n^3)$ 个处理器。

6.2.2 利用BFS搜索连通支

设 $T(v, \lceil \log n \rceil)$ 是以 v 为根的 G 的 BFS 支撑树， $P(u, v)$ 表示 $T(v, \lceil \log n \rceil)$ 中 u 的双亲。如果 G 不是连通图，上述 BFS 算法的输出结果是什么？ $T(v, \lceil \log n \rceil)$ 将不是图 G 的支撑树或支撑森林， $T(v, \lceil \log n \rceil)$ 仅仅是包含 v 的 G 的连通支的支撑树。为简单起见，记 $\text{BFS}(v)$ 为 $T(v, \lceil \log n \rceil)$ 。如果 G 不是连通的， $P(u, v)$ 仅仅定义在从 v 可到达的顶点 u 上。如果从 v 不可到达 u ，则定义 $P(u, v) = \infty$ 。简而言之，

191

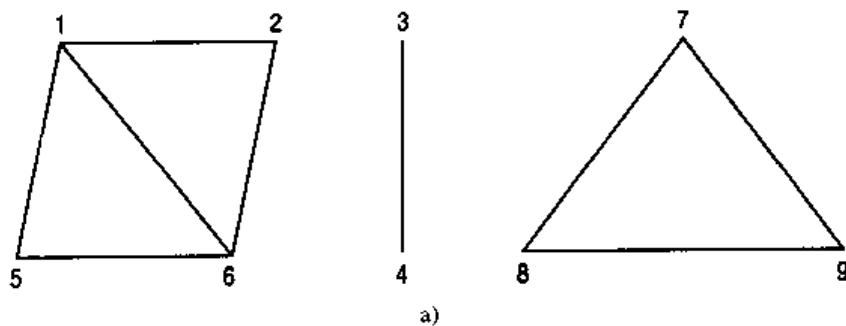
$$P(u, v) = \begin{cases} \text{BFS}(v) \text{ 中 } v \text{ 的双亲, 如果 } u \text{ 是 } \text{BFS}(v) \text{ 的顶点} \\ \infty, \text{ 否则} \end{cases}$$

为方便起见，我们记顶点为 $1, 2, 3, \dots, n$ ，定义连通度矩阵为

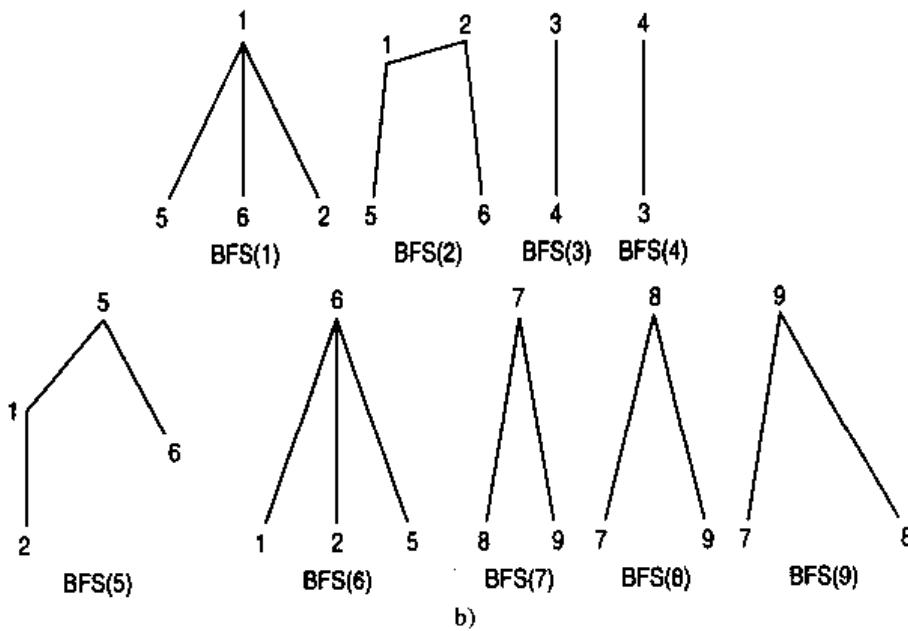
$$C(i, j) = \begin{cases} \infty, \text{ 如果 } P(i, j) = \infty \\ i, \text{ 否则} \end{cases}$$

在连通度矩阵中，顶点 j 的分支数是指包含 j 的分支的最小顶点，即 j 列的最小元素。考虑图 6-6a 和它们的 BFS 树，即图 6-6b，矩阵 $P(i, j)$ 和 $C(i, j)$ 如下：

192



a)



193

图 6-6
a) 图 G b) 得到的 BFS 树

194

P	1	2	3	4	5	6	7	8	9
1	1	2	∞	∞	5	6	∞	∞	∞
2	1	2	∞	∞	1	6	∞	∞	∞
3	∞	∞	3	4	∞	∞	∞	∞	∞
4	∞	∞	3	4	∞	∞	∞	∞	∞
5	1	1	∞	∞	5	6	∞	∞	∞
6	1	2	∞	∞	5	6	∞	∞	∞
7	∞	∞	∞	∞	∞	∞	7	8	9
8	∞	∞	∞	∞	∞	∞	7	8	9
9	∞	∞	∞	∞	∞	∞	7	8	9

C	1	2	3	4	5	6	7	8	9
1	1	1	∞	∞	1	1	∞	∞	∞
2	2	2	∞	∞	2	2	∞	∞	∞
3	∞	∞	3	3	∞	∞	∞	∞	∞
4	∞	∞	4	4	∞	∞	∞	∞	∞
5	5	5	∞	∞	5	5	∞	∞	∞
6	6	6	∞	∞	6	6	∞	∞	∞
7	∞	∞	∞	∞	∞	∞	7	7	7
8	∞	∞	∞	∞	∞	∞	8	8	8
9	∞	∞	∞	∞	∞	∞	9	9	9

顶点 j 的分支数是矩阵 C 的第 j 列的最小数。图 6-6a 中，

顶点:	1	2	3	4	5	6	7	8	9
分支数:	1	1	3	3	1	1	7	7	7

CREW 模型下基于上述方法的并行算法如下：

算法 CC-BFS

输入：图 $G = (V, E)$

输出：每个顶点 v 的分支数

BEGIN

1. Using ALGORITHM BFS construct the BFS trees $\text{BFS}(v)$

for each $v \in V$ in Parallel

2. For $i = 1$ to n do in Parallel

For $j = 1$ to n do in Parallel

If $P(i, j) = \infty$ then

$C(i, j) = \infty$

else

$C(i, j) = i$

End if

End Parallel

3. For $j = 1$ to n do in parallel

$\text{Comp}(j) = \text{Min} \{C(i, j), i = 1, 2, \dots, n\}$

```

    End Parallel
END

```

BFS算法的复杂度是用 $O(n^3)$ 个处理器，执行时间为 $O(\log^2 n)$ 。第2步用 $O(n^2)$ 个处理器，执行时间为 $O(1)$ 。取最小值所需处理器数为 $O(n)$ ，执行时间为 $O(\log n)$ ，所以，第3步需要 $O(n^2)$ 个处理器，执行时间为 $O(\log n)$ 。因此，上述算法用CREW PRAM模型共需 $O(n^3)$ 个处理器，执行时间为 $O(\log^2 n)$ 。

195

6.2.3 传递闭包矩阵

根据图 $G = (V, E)$ 的相邻矩阵 A ，通过计算 $(A + I)^n$ 可以得到传递闭包矩阵，并记 $R = (A + I)^n$ ，注意

$$R(i, j) = \begin{cases} 1, & \text{在 } G \text{ 的同一分支中, 如果 } i \text{ 与 } j \text{ 相邻} \\ 0, & \text{否则} \end{cases}$$

顶点 i 的分支数是使得 $R(i, j) = 1$ 的最小指标 j 。并行算法为：

算法CC-TRC

输入：图的相邻矩阵 $A(i, j)$

输出：每个顶点的分支数

BEGIN

1. Obtain the transitive closure matrix R from the adjacency matrix
2. For $i = 1$ to n do in Parallel

Comp(i) = Min $\{j / R(i, j) = 1\}$

End Parallel

END

算法的复杂度依赖于矩阵乘法的复杂度。对于CRCW PRAM模型，计算 R 需要 $O(n^3)$ 个处理器，共需时间为 $O(\log n)$ 。对于CREW PRAM模型，计算 R 需要 $O(n^3)$ 个处理器，共需时间为 $O(\log^2 n)$ 。由于第2步仅需 $O(n^2)$ 个处理器，执行时间为 $O(\log n)$ ，因此总的计算复杂度为：

模型	时间	处理器
CRCW	$O(\log n)$	$O(n^3)$
CREW	$O(\log^2 n)$	$O(n^3)$

6.2.4 顶点收缩

1976年，Hirschberg首先提出了求图的连通支的顶点收缩法。相邻的顶点合并组成一个“超点”。超点重复地连接成它们的超点，重复上面过程直到每一个分支用一个超点表示。已经证明，要达到最后阶段共需执行 $\lceil \log n \rceil$ 次重复操作。每一个过程由以下三步组成：

1. 找出每个顶点具有最低标号的相邻超点。
2. 每个超点的根与最低标号的相邻超点的根相连。
3. 所有新得到的相连的超点收缩成一个较大的超点。

196

最初，每一个顶点被看作超点， $\text{sup}(i)$ 表示 i 的超点， $\text{nhr}(i)$ 表示 i 的最低标号的相邻超点。

上面算法描述如下：

```

BEGIN
  1. For  $i = 1$  to  $n$  do
    sup( $i$ ) =  $i$ ;
  2. For  $k = 1$  to  $\lceil \log n \rceil$  do
    2(a) For  $i = 1$  to  $n$  do
      Compute the lowest numbered neighboring supernode and call it nbr( $i$ ). If  $i$  has no
      neighboring supernode assign nbr( $i$ ) = sup( $i$ )
      end for
    2(b) For  $i = 1$  to  $n$  do
      Among all vertices whose supernode is  $i$  choose the vertex  $j$ 
      with minimum nbr( $j$ ) value
      Assign nbr( $i$ ) = nbr( $j$ )
      If there is no such vertex assign nbr( $i$ ) = sup( $i$ )
      sup( $i$ ) = nbr( $i$ )
    2(c) For  $p = 1$  to  $\lceil \log n \rceil$  do
      For  $i = 1$  to  $n$  do
        nbr( $i$ ) = nbr(nbr( $i$ ))
      end for
    end for
  2(d) For  $i = 1$  to  $n$  do
    sup( $i$ ) = Min {nbr( $i$ ), sup(nbr( $i$ ))}
  end for
END

```

197

输入是图的相邻矩阵 $A[1:n, 1:n]$, 输出是 sup(i), 其最后给出了在含有 i 的分支中最低标号的顶点。算法的执行过程如图 6-7a ~ 6-7e 所示, 下面给出一个完整的并行算法, 执行 CREW PRAM 模型。

算法 CC-VC

输入: 相邻矩阵 $A[1:n, 1:n]$

输出: sup[1:n] 在最终结果中 sup(i) 是最小的顶点且 i 和 sup(i) 属于同一分支

BEGIN

```

  1. For  $i = 1$  to  $n$  do in parallel
    sup( $i$ ) =  $i$ 
  End Parallel
  2. For  $k = 1$  to  $\lceil \log n \rceil$  do
    2(a) For  $i = 1$  to  $n$  do in parallel
      nbr( $i$ ) = Min {sup( $j$ )/ $A(i, j) = 1$  and  $1 \leq j \leq n$ }
      If nbr( $i$ ) does not exist
        nbr( $i$ ) = sup( $i$ )
      end if
    end parallel

```

198

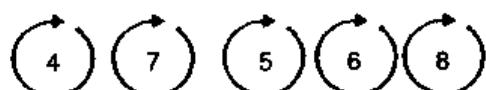
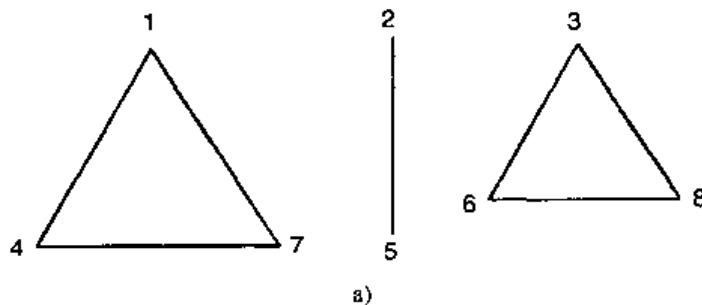
2(b) For $i = 1$ to n do in Parallel
 $\text{nbr}(i) = \text{Min} \{ \text{nbr}(j)/\text{sup}(j) = i \text{ and } \text{nbr}(j) = j \}$
 If no such vertex j exists then
 $\text{nbr}(i) = \text{sup}(i)$
 end if
 $\text{sup}(i) = \text{nbr}(i)$
 End Parallel

2(c) For $p = 1$ to $\lceil \log n \rceil$ do
 For $i = 1$ to n do in Parallel
 $\text{nbr}(i) = \text{nbr}(\text{nbr}(i))$
 end parallel
 end for

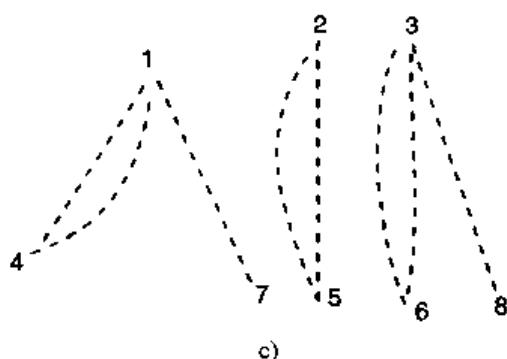
2(d) For $i = 1$ to n do in parallel
 $\text{sup}(i) = \text{Min} \{ \text{nbr}(i), \text{sup}(\text{nbr}(i)) \}$
 end parallel

END

199



b)



c)

图 6-7

a) 原始图 b) 初始 sup(i) = i c) 迭代一次, nbr(i)

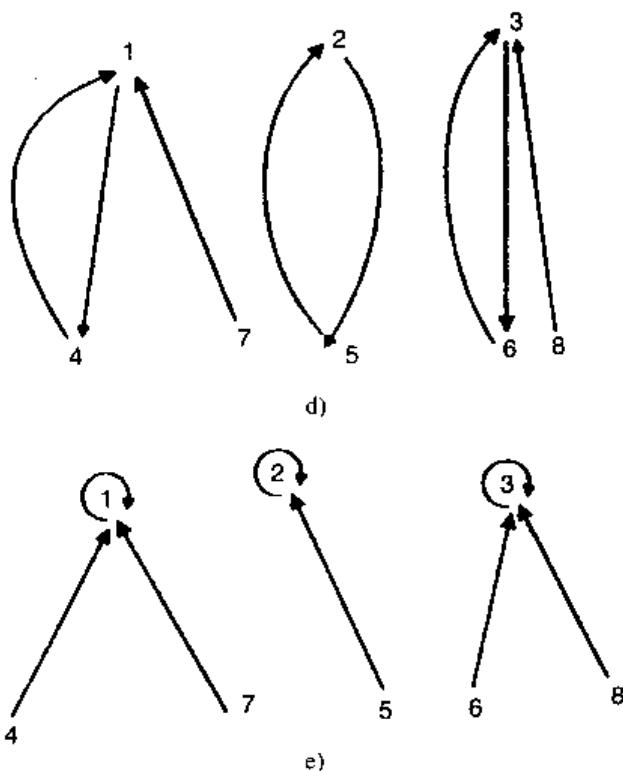


图6-7 (续)

d) 迭代一次, $\text{nbr}(i) = \text{sup}(i)$ e) 迭代一次, $\text{sup}(i)$

第1步用 $O(n)$ 个处理器, 执行时间为 $O(1)$ 。2(a)和2(b)步用 $O(n)$ 个处理器, 最少需执行时间为 $O(\log n)$ 。2(c)步也需要 $O(n)$ 个处理器, 执行时间为 $O(\log n)$ 。由于2(c)步重复执行 $\lceil \log n \rceil$ 次, 因而该步共用处理器数为 $O(n^2)$, 执行时间为 $O(\log^2 n)$ 。

Hirschberg等人(1979)认识到, 在时间复杂度没有提高的情形下, 可以减少所需要的处理器数, 从而可以提高原始的Hirschberg 算法的性能。新算法所用的处理器数为 $O(n[n/\log n])$, 时间为 $O(\log^2 n)$, 且执行SIMD-SM-R模型。

Chin等人(1981,1982) 考虑到迭代步仅仅用于非孤立的超点上, 从而提高了Hirschberg 算法的性能。这意味着每一次迭代后至少有两个因素使得起作用的顶点个数减少了。因此, 用 $O(n[n/\log n])$ 个处理器, 执行时间为 $O(\log^2 n)$, 且执行SIMD-SM-R模型。

Savage和Ja Ja (1981) 设计了两个并行连通度算法, 其中一个算法针对稠密图, 另一个针对稀疏图。Savage 和Ja Ja 观察到, 执行Hirschberg 算法时没有必要做 $\lceil \log n \rceil$ 次迭代, 直到连续两次迭代所得的结果相同即可停止。这时, 算法运行的时间是 $O(\log n * \min\{\log n, d/2\})$, 这里 d 是图的直径。当 $d < 2\log n$ 时 (例如稠密图), 算法比原始算法要快。对于稠密图, 算法需处理器数 $O(n^3/\log n)$, 执行时间是 $O(\log n \log d)$ 。

当图是稀疏图时, 通过组织, 我们输入相邻列表而不是相邻矩阵, 这样就减少了所需的处理器数。对Hirschberg 算法的重新组织产生了一个新的算法, 该算法所需的处理器数为 $O((m + n)\log n)$, 执行时间是 $O(\log^2 n)$, 这里 m 是图的边数。

Nath和Maheshwari (1982) 在一个比较弱的模型上考虑求连通支问题, 这个模型不允许不同的处理器对同一内存位置执行并发读操作。他们的算法是根据Hirschberg 等人(1979)的算法, 通过将中间的数据结构组成一个链, 并且保持多份数据拷贝, 从而避免了上述冲突。这个算

法用 n 个处理器，执行时间为 $O(\log^2 n)$ ，且执行SIMD-SM模型。

Shiloach和Vishkin (1982)考虑了一个较强的模型SIMD-SM-RW。他们将每个无向边看作两个有向边，将每个顶点和每个有向边放到一个处理器上。与Hirschberg算法相反，不再要求通过最低标号顶点来判断连通支。这个算法利用比较强的模型，不再将最小标号邻根连接起来，让顶点尝试同时“挂钩”(hooking)。这使得两个 $\log^2 n$ 因子中的一个被 $\log n$ 代替。重新组织使得在每次迭代中对于路的比较步仅用两次，而不是 $\log n$ 次，这样另一个 $\log^2 n$ 也被 $\log n$ 代替。

Awerbuch和Shiloach (1983)根据Shiloach和Vishkin (1982)的结果得到了利用 $2m$ 个处理器，执行时间为 $O(\log n)$ 且执行SIMD-SM-RW模型的改进型连通支算法。Wyllie (1979)利用顶点收缩得到了利用 $O(n + 2m)$ 个处理器，时间为 $O(\log^2 n)$ 且执行同步MIMD-TC-R模型的算法。稀疏图是非常有效的算法。通过把与 x 关联的任一条边变成与 y 关联的边，使顶点 x 收缩到顶点 y 。每条边被看作两个有向边，输入为相邻列表矩阵，且在 $O(1)$ 时间内完成转换，这样每个顶点都有一个有表头的循环双向链表。列表由两部分元素组成：有向边和哑元。边元素至少被一个哑元分开。这一有向元素结构允许两个顶点有效收缩成一个顶点，哑元的使用允许不同顶点的同时收缩。

6.3 2-连通支

对于连通图 G 的顶点 v ，如果 $G-v$ 是不连通的，则称 v 为 G 的断点 (articulation point)。如果一个图没有断点，则称为2-连通图。图6-8a是2-连通图；而图6-8b不是2-连通图，因为5是一个断点 (4是另一个断点)。一个极大诱导2-连通子图称为图的2-连通支。图6-8b有三个2-连通支，如图6-8c所示。检验一个图的2-连通度是一个有趣而简单的问题。对于每一个顶点 v ，利用连通支算法，需要验证 $G-v$ 是否是连通的。对每一个顶点 v ，如果结果是正的，则该图是2-连通的。

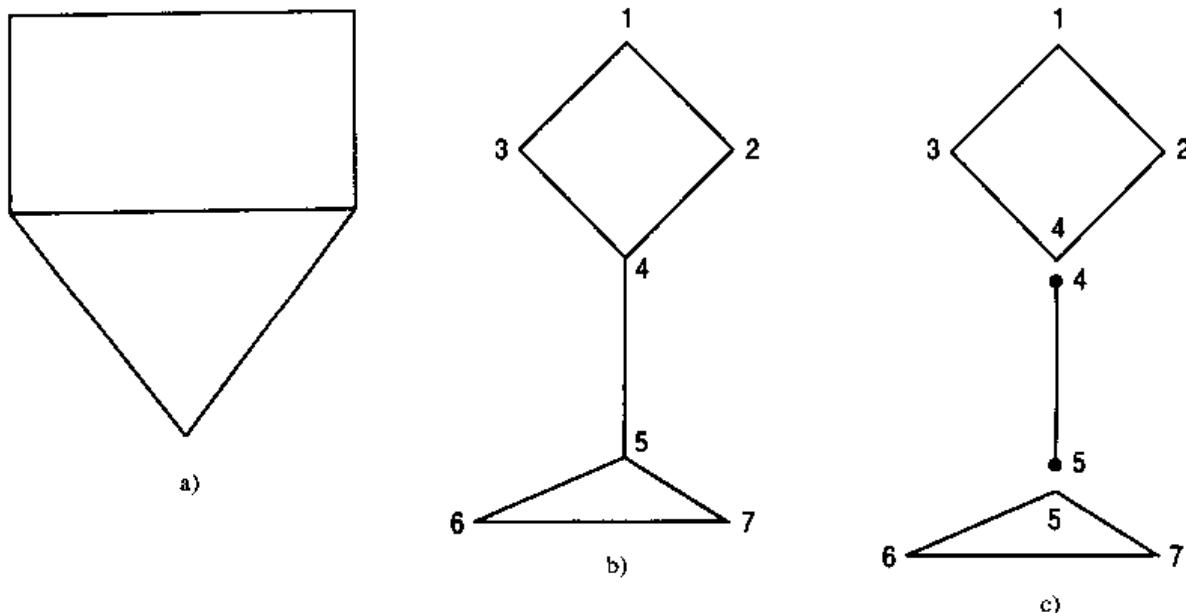


图 6-8

a) 2-连通图 b) 非双连通图 c) 图6-8b中的2-连通支

[202]

```

BEGIN
  For every  $v \in V$  do in parallel
    if  $G-v$  is connected
       $BCC(v) = 0$ 
    else
       $BCC(v) = 1$ 
    endif
  end parallel
   $BCC(G) = \sum_{v \in V} BCC(v)$ 
  If  $BCC(G) = 0$ , then  $G$  is biconnected, otherwise not biconnected
END

```

为了检验 $G-v$ 的连通度，我们可以利用前一节描述的CC-VC算法，完整的算法如下：

算法BCC

输入：相邻矩阵 $A[i:n, i:n]$

输出：布尔值BCC。如果 G 是2-连通图， $BCC = \text{TRUE}$ ；反之， $BCC = \text{FALSE}$

BEGIN

```

1. For  $i = 1$  to  $n$  do in parallel
  1(a) Construct  $A_i [1:n-1, 1:n-1]$  from  $A$  by deleting the  $i$ th row and the  $i$ th column
  1(b) Using CC-VC algorithm verify whether  $A_i$  represents a connected graph
  1(c) If it represents a connected graph assign  $B(i) = 0$ 
    else
       $B(i) = 1$ 
    End if
  End Parallel
2.  $S = \sum_{i=1}^n B(i)$ 
3. If  $S = 0$  then  $BCC = \text{TRUE}$ 
  else
     $BCC = \text{FALSE}$ 
  End if
END

```

[203]

复杂度分析 第1步需处理器数 $O(n^2)$ ，执行时间为 $O(1)$ ；用CC-VC算法来验证连通度需 $O(n^2)$ 个处理器且执行时间为 $O(\log^2 n)$ 。第1步对 n 个顶点并行执行，因此第1步共需 $O(n^2)$ 个处理器且执行时间为 $O(\log^2 n)$ 。第2步共需 $O(n)$ 个处理器，执行时间为 $O(\log n)$ 。因此算法BCC共需处理器 $O(n^3)$ ，执行时间为 $O(\log^2 n)$ 。改进后的算法可以用来求图的所有2-连通支。

6.4 支撑树

支撑树在实际应用中非常有用。例如通讯网络，运输模型等。我们用顶点来表示城市，用边来表示连接城市之间的交通路线，支撑树在这个问题的研究中非常有用。当给边赋权后，

有极小总权重的图的支撑树非常有用，这样的支撑树称为图的最小成本支撑树。

本节我们求加权图的最小成本支撑树问题。下面给出三种方法来求一个连通简单加权图的最小成本支撑树。

1. Prim方法；
2. Kruskal方法；
3. Sollin方法。

在Prim方法中，支撑树的构造是从单一点出发，同时挑选一点并把它放到支撑树中，使得总权重最小。Kruskal方法是对边进行逐个挑选使得总成本最小。本节我们学习Sollin算法。

Sollin算法 一个图用二维数组COST描述为

$$\text{COST}(u, v) = \begin{cases} \text{边}(u, v) \text{ 的权重, 如果 } (u, v) \text{ 是一条边} \\ \infty, \text{ 如果 } u = v \\ \infty, \text{ 如果 } (u, v) \text{ 不是一条边} \end{cases}$$

204

Sollin算法将给定图的所有顶点看作孤立点，因此，图的 n 个顶点被看作 n 个树，每一个树只有一个顶点，记这 n 个树为 $T_{10}, T_{20}, \dots, T_{n0}$ 。它们组成一片森林

$$F_0 = \{T_{10}, T_{20}, \dots, T_{n0}\}$$

对于每一个树，我们选取最小成本边 (u, v) ，使得对于不同的 i 和 j ，有 $u \in T_{i0}, v \in T_{j0}$ 。当把这些挑选出来的边连成树时，可以得到，在下一次迭代中至多有一半数目的树。

一般地，在第 i 次迭代中，

$$F_i = \{T_{1i}, T_{2i}, \dots, T_{ni}\}$$

对于每一个树 T_{ji} ，我们必须选择最小成本边 (u, v) ，使得对不同的 k 和 j 有 $u \in T_{ki}, v \in T_{ji}$ 。利用这条边连接树 T_{ki} 和 T_{ji} ，得到的森林记为 F_{i+1} 。由上知森林 F_{i+1} 至多含有一半数目的 F_i 中的树。在某次迭代中，如果只得到一个树，就停止这个过程，这个树就是最小成本支撑树。由于树的数目在每一次迭代中依次减半，因此不到 $\log n$ 次迭代我们就可以得到最小成本支撑树。利用Sollin方法的并行算法如下：

算法Sollin

输入：用成本数组 $\text{COST}(u, v)$ 表示的图 $G = (V, E)$ ，这里 $u, v \in V$

输出： G 的最小成本支撑树

BEGIN

1. $F_0 = (V, \Phi)$
2. $i = 0$
3. While there is more than one tree in F_i do
4. For each tree T_j in forest F_i do in Parallel
5. Choose the minimum weight edge (u, v) joining some vertex u in T_j to a vertex v in some other tree T_k in forest F_i
6. Form the forest F_{i+1} by joining all T_j and T_k of F_i with the corresponding selected edges:
7. $i = i + 1$
8. End Parallel
9. End While

上面的算法并不完整，它需要我们解释第4~6步是如何并行执行的。Savage 和Ja Ja(1981)给出了下面的技术：森林 F_i 中有几个树，树用它的双亲关系表示，用另一个数据结构 $\text{NEAR}(u)$ 表示最靠近 u 的另一个树的顶点，数据结构 ROOT 则给出这个树的树根。并行执行如下：

算法Sollin-1

输入：由加权矩阵 $W(u,v)$ 表示的加权图 $G = (V,E)$

输出： G 的最小成本支撑树

205

BEGIN

1. For each $u \in V$ do in parallel
2. $\text{ROOT}(i) = i$
3. End parallel
4. Over = False
5. While not (over) do
6. For each $u \in V$ do in parallel
7. $\text{NEAR}(u) = v$ such that $\text{ROOT}(u)$ different from $\text{ROOT}(v)$ and $W(u,v) = \min\{W(u,w)/w \in V\}$
8. End parallel
9. For each component k of the forest F_i do in parallel
10. Choose a vertex u such that $W(u, \text{NEAR}(u))$ is minimum overall vertices of k
11. End parallel
12. Combine the new edges and create the new forest F_{i+1}
13. For each vertex $u \in V$ do in parallel
14. Search $\text{ROOT}(u)$
15. If $\text{ROOT}(u) = \text{ROOT}(v)$ for all $u, v \in V$ then over = true
16. End Parallel
17. End while

END

第1~3步是简单操作，可用 $O(n)$ 个处理器在 $O(1)$ 时间内完成。下一步迭代中的森林 F_{i+1} 至多含有一半数目的 F_i 中的树，因此，第5~17步的循环共执行 $\log n$ 次。第7步选择最近的节点，利用合适的数据结构，并保留按权重降序排列的边。运用二分搜索技术，第7步执行时间为 $O(\log n)$ 。第6~8步是并行循环，用 $O(n)$ 个处理器且执行时间为 $O(\log n)$ 。第9~11步是并行循环，第10步用 $O(n)$ 个处理器且执行时间为 $O(\log n)$ 。类似地，第12~16步是并行循环，第14~15步用Ja Ja给出的数据结构共需处理器数为 $O(n)$ ，执行时间为 $O(\log^2 n)$ 。因此Sollin算法共需 $O(n^2)$ 个处理器，并行执行时间为 $O(\log^2 n)$ 。执行的PRAM模型是CREW。

6.5 最短路问题

考虑加权有向图 $G = (V,E)$ ，每一条边对应一个非负权重，边 (i,j) 的权重记为 w_{ij} 。假定对所有的 $i \in V$, $w_{ii} = 0$ 。如果 (i,j) 不是一条边，定义 $w_{ij} = \infty$ 。路的长度是指这条路中所有边的权重和。顶点 i 到 j 的最短路是指从 i 到 j 具有最小长度的路。最短路用路的长度和沿着这条路的顶点的有序集合（或边的有序集合）表示。所有顶点对的最短路问题是求每个有序对 (i,j) 间的

206

最短路问题。下面的数据结构用来设计一个算法，该算法用于求所有顶点对的最短路问题。

定义 d_{ij} 表示*i*到*j*的最短路；

P_{ij} 表示*i*到*j*的最短路中*j*的祖先。

考虑图6-9，它可以用下面的加权矩阵(w_{ij})来表示：

W	a	b	c	d	e	f
a	0	20	12	∞	5	∞
b	20	0	5	∞	∞	∞
c	12	5	0	5	4	∞
d	∞	∞	5	0	15	∞
e	5	∞	4	15	0	6
f	∞	∞	∞	∞	∞	60

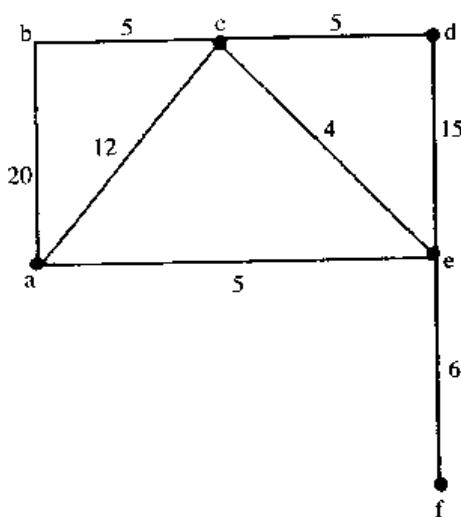


图6-9 加权图

任意两个顶点间的最短路用下面的矩阵表示：

d	a	b	c	d	e	f
a	0	14	9	14	5	11
b	14	0	5	10	9	15
c	9	5	0	5	4	10
d	14	10	5	0	9	15
e	5	9	4	9	0	6
f	11	15	10	15	6	0

207

由于这个例子对应的图是无向图，因而得到的矩阵是对称的。所有路列表如下：

Path (a,b) = $a \rightarrow e \rightarrow c \rightarrow b$

Path (a,c) = $a \rightarrow e \rightarrow c$

Path (a,d) = $a \rightarrow e \rightarrow c \rightarrow d$

Path (a,e) = $a \rightarrow e$

Path $(a,f) = a \rightarrow f$

Path $(b,c) = b \rightarrow c$

Path $(b,d) = b \rightarrow c \rightarrow d$

Path $(b,e) = b \rightarrow c \rightarrow e$

Path $(b,f) = b \rightarrow c \rightarrow e \rightarrow f$

Path $(c,d) = c \rightarrow d$

Path $(c,e) = c \rightarrow e$

Path $(c,f) = c \rightarrow e \rightarrow f$

Path $(d,e) = d \rightarrow c \rightarrow e$

Path $(d,f) = d \rightarrow c \rightarrow e \rightarrow f$

Path $(e,f) = e \rightarrow f$

从上面的路中我们可以求每一顶点对 (i, j) 的值 P_{ij} 。上面的数据结构可有效地用于下面的 Floyd Warshall 算法中。首先，我们有：

$$d_{ij} = \begin{cases} w_{ij}, & (i, j) \in E \\ \infty, & (i, j) \notin E \\ 0, & i = j \end{cases}$$

208 $P_{ij} = i$, 对所有的 i 和 j

上面的初始化意味着首先要考虑有向距离。现在来比较 d_{ij} 与从 i 到中间点 k 的距离和从 k 到 j 的距离之和的值，如果 d_{ij} 较大，取经过 k 的路。上面的更新结果可以通过选取每一个顶点 k 来修正。Floyd-Warshall 算法如下：

算法 Floyd-Warshall

输入：由加权矩阵 (W_{ij}) 表示的图

输出：矩阵 (d_{ij}) 和 (P_{ij})

BEGIN

1. For $i = 1$ to n do in parallel
2. For $j = 1$ to n do in parallel
3. $d_{ij} = w_{ij}$
4. $P_{ij} = i$
5. End parallel
6. End parallel
7. For $k = 1$ to n do
8. For each pair (i, j) where $0 < i, j \leq n$ and $i, j \neq k$ do in parallel
9. If $d_{ij} > d_{ik} + d_{kj}$ then

$$d_{ij} = d_{ik} + d_{kj}$$

$$P_{ij} = P_{kj}$$

endif

10. End parallel

11. End for

12. Output p and d matrices.

END

上面算法执行CREW PRAM模型，共需处理器数为 $O(n^2)$ ，执行时间为 $O(n)$ 。Gayraud和Authir(1991)又建议了另一个并行实现的算法，使用 P 个处理器且执行时间为 $O(n^3/p+np)$ 。在他们的算法中，考虑了面向行的技术。在第 k 步迭代中，矩阵(d_{ij})和(p_{ij})的第 i 行的更新由单独的过程LMOD(i, k)给出。第7~11步为

BEGIN

7. For $k = 1$ to n do
8. For $i = 1$ to n ($i \neq k$) do in parallel
9. LMOD(i, k)
10. End parallel
11. End for

END

过程LMOD如下：

过程LMOD(i, k)

BEGIN

For $j = 1$ to n , $j = i$ and $j = k$ do in parallel
 if $d_{ij} > d_{kj}$ then
 $d_{ij} = d_{ik} + d_{kj}$
 $P_{ij} = P_{kj}$
 end if

End Parallel

END

参考文献

- Samy, A. A., Arumugam, G., Devasahayam, M. P. and Xavier, C. (1991) Algorithms for Intersection Graphs of Internally Disjoint Paths in Trees, *Proceedings of National Seminar on Theoretical Computer Science*, IMSC, Madras, India, Report 115, pp. 169–178.
- Berge, C. and Chatval, C. (1984) *Topics on Perfect Graphs. Annals of Discrete Mathematics*, 21.
- Sekharan, N. C. (1985) New Characterizations and Algorithmic Studies on Chordal Graphs and k -Trees, (M.Sc. (Engg.) Thesis), School of Automation, Indian Institute of Science, Bangalore, India.
- Sekharan, N. C. and Iyengar, S. S. (1988) NC Algorithms for Recognizing Chordal Graphs and k -trees, *IEEE Transactions on Computers*, 37(10).
- Ja Ja, J. (1992) *Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA.
- Kelly, D. (1985) Comparability Graphs. In I. Rival, ed., *Graphs and Order*, D. Reidel, pp. 3–40. (NATO ASI series C, v14.)
- Klein, P. N. (1988) Efficient Parallel Algorithms for Chordal Graphs, Proc. 29th IEEE symposium on foundation of Computer Section (1988) pp. 150–161.
- Klein, P. N. (1988) Efficient Parallel Algorithms for Planar, Chordal and Interval

- Graphs, TR 426 (Ph.D. thesis) Laboratory for Computer Science, MIT, Cambridge, MA.
- Kozen, D., Vazirani, U. V. and Vazirani, V. V. (1985) NC Algorithms for Comparability Graphs, Interval Graphs and Unique Perfect Matchings. In *Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, 496–503, New York and Berlin, pp. 496–503. (Springer Lecture Notes in Computer Science, 206.)
- Moitra, A. and Iyengar, S. S. (1987) Parallel Algorithms for Some Computational Problems. *Advances in Computers* 26, Academic Press, New York, pp. 93–153.
- Manacher, G. K. (1992) Chord Free Path Problems on Interval Graphs, *Computer Science and Informatics*, 22(2), 17–24.
- Mohring, R. H. (1984) Algorithmic Aspects of Comparability Graphs and Interval Graphs. In I. Rival, ed., *Graphs and Order*, pp. 41–101, D. Reidel. (NATO ASI series C. v. 147.)
- Xavier, C. and Arumugam, G. (1994) Algorithms for Parity Path Problems in Some Classes of Graphs, *Computer Science and Informatics*, 24(4), 50–54.
- Xavier, C. (1995) Sequential and Parallel Algorithms for Some Graph Theoretic Problems (Ph.D. thesis), Madurai Kamaraj University, India.
- 习题**
- 6.1 拓扑排序定义如下：给定一个含 n 个顶点的有向无圈图 $G = (V, E)$ ，给每个顶点 v 标上记号 l , $l \in \{1, 2, \dots, n\}$ ，使得只要 $(u, v) \in E$ ，就有 $l(u) < l(v)$ 。设计一个并行算法对图 G 执行拓扑排序。
- 6.2 连通图 G 的迹是指序列 $W = \{e_1 = (v_0, v_1), e_2 = (v_1, v_2), \dots, e_k = (v_{k-1}, v_k)\}$ ，所有的边都不同。欧拉迹遍历 G 的每条边。设计一个算法求图 G 的欧拉迹。
- 6.3 $G = (V, E)$ 是一个二部图，最大的度是 $\Delta = 2^l$, l 是正整数。
- 如何将 G 分成两个二部图 G_1 和 G_2 ，其中每个最大度都是 $\Delta/2$ 。算法所用的时间为 $O(\log n)$ ，所用操作数为 $O(|E|+|V|)$ ；
 - G 的 k -边染色是指从颜色集 $\{1, 2, \dots, k\}$ 中选出颜色给每条边分配颜色，使得只要 e 和 g 有公共顶点，就有 $c(e) \neq c(g)$ 。利用(a)中算法求 G 的 Δ -边染色。
- 6.4 $G = (V, E)$ 是一个无向图且用它的相邻列表表示。欧拉划分是指将 E 分成边不相交的路（包括圈） $\{P_i\}$ ，使得每个奇数度的顶点恰是一条路（有两个不同端点）的端点，同时偶数度的顶点不能成为这样一条路的端点。如何在 $O(\log n)$ 时间内利用线性数目的操作得到一个欧拉划分。提示：从欧拉图入手。
- 6.5 假设给定了一个无向图 $G = (V, E)$ 及一个向量 D ，使得 $D(u) = D(v)$ ，当且仅当 u, v 在同一个连通支中。设计一个有效的并行算法来确定连通支数 k ，并在每个连通支中生成一个独立的顶点列表。
- 6.6 已知有3种不同的方法来描述图 $G = (V, E)$ ：相邻矩阵，相邻列表集合，边的无序集合。设计一个并行算法将一种表示转变成另一种表示，所用时间为 $O(\log n)$ ，这里 $|V| = n$ 。每一种情形中总的操作数是多少？
- 6.7 给定一个含 n 个顶点的无向连通图 G 。设计一个算法来求 G 的所有支撑树，所用的时间为 $O(\log n)$ ，所用的总操作数是 $O((n+m)\log n)$ ，这里 m 指边的个数。假定输入的是边的序列。不允许用优先CRCW PRAM模型。

- 6.8 对于无向图 $G = (V, E)$, 如果存在 V 的一个划分 $V = V_1 \cup V_2$, 使得每条边有一个端点在 V_1 中, 另一个端点在 V_2 中, 则 G 称为二部图。设计一个算法求这样一个二部图, 所用时间为 $O(\log n)$ 。

提示: 从求 G 的支撑树出发。

- 6.9 对于连通无向图 $G = (V, E)$ 的某个顶点 v , 如果去掉顶点 v , G 就不连通了, 称 v 为断点。

a. 证明: v 是断点当且仅当它属于不止一个块。

b. 设计一个有效的并行算法求 G 的所有断点。

- 6.10 假定 $G = (V, E)$ 是一个连通图, 对于 G 的某条边 e , 如果去掉这条边 G 就不连通了, 则称 e 为桥。设计一个并行算法求 G 的所有的桥。算法的复杂度应与其中一个连通支算法的复杂度匹配。

提示: 从求 G 的支撑树出发。

[212]

第7章 弦图的NC算法

如果并行算法的时间复杂度是 $O(\log^k n)$, 并且所用的处理器数是 n 的多项式, 这里 n 是输入的大小, k 是常数, 则该算法属于NC类型。本章我们研究弦图的NC算法。弦图在数据结构的分类中担当重要角色。如果输入的图是弦图, 一些NP完全问题可在线性时间内解决。我们在第2章已经介绍了弦图, 弦图是完美图的一个重要的子类。这里完美图是指这样一种图, 它的最大团的大小等于图的色数。弦图在许多领域都有应用, 其中包括高斯消元法、数据库等。首先列出弦图的一个性质。

定理7-1 图 $G = (V, E)$ 是弦图, 当且仅当 G 的每一个极小分离集导出 G 的一个团。

证明: 设 u, v 是弦图 $G = (V, E)$ 的两个不相邻的顶点, S 记为 u, v 的极小分离集, 如果 S 仅含有一个顶点, 则 S 导出一个团; 否则, 假定 x, y 是 S 中不相邻的顶点。由于 S 是极小 u, v 分离集, u, v, x, y 是不含弦的圈的四个顶点。这与 G 是弦图矛盾, 因此 S 的每一对顶点都相邻。 S 导出一个团。

反过来, 假定每一个极小分离集都是一个团。设 $v_1, v_2, \dots, v_k, v_1$ 是无弦的圈, v_1 和 v_k 是两个不相邻的顶点, 其极小分离集包含 v_2 以及顶点 v_4, v_5, \dots, v_k 中至少一个顶点。也就是说, 极小分离集含有顶点 v_2 和 v_i , 这里 $4 < i < k$ 。但是 v_2 和 v_i 是不相邻的, 因此极小分离集不能导出一个团。这与假设矛盾。

7.1 弦图判别

为了设计判别弦图的算法, 我们引入几个记号: 如果 u 和 v 是图 G 的两个不相邻的顶点, 记 $G_u = G - \text{adj}(u)$ 。

$C_{uv} = G_u$ 中包含 v 的分支;

$M_{uv} = \{x : x \in \text{adj}(u), x \text{与 } C_{uv} \text{ 中的某些顶点相邻}\}$ 。

考虑图7-1a所示的图, 图7-1b表示 G_u , 图7-1c表示包含顶点 v 的 G_u 的分支 C_{uv} 。

M_{uv} 表示与 C_{uv} 的顶点相邻的 $\text{adj}(u)$ 中的顶点集。在图7-1a中, $M_{uv} = \{6, 3\}$ 。 M_{uv} 是极小 $u-v$ 分离集。我们证明如下结论。

定理7-2 图 G 是弦图当且仅当对于每一对不相邻的顶点 u 和 v , M_{uv} 是团。

证明: 在证明结论之前, 首先证明 M_{uv} 是 $u-v$ 分离集, 并且证明如果 M_{uv} 含有一对不相邻的顶点, 则 G 含有一个长度至少为4的无弦圈。很明显, $G - M_{uv} - \{u\}$ 包含 C_{uv} , 任一个在 $G - M_{uv} - \{u\}$ 中而不在 C_{uv} 中的顶点 x 与 C_{uv} 中的顶点不相邻。 M_{uv} 分离 u 和 C_{uv} 中的所有顶点。特别地, M_{uv} 是 G 的 $u-v$ 分离集。现在证明如果 M_{uv} 含有一对不相邻的顶点, 则 G 含有一个长度至少为4的无弦圈。

假定 x 和 y 是 M_{uv} 中两个不相邻的顶点, x 和 y 分别与 C_{uv} 中顶点 r 和 s 相邻。假定 $ra_1a_2a_3 \cdots a_ps$ 是 C_{uv} 中的最短的无弦路, 则 $uxra_1a_2a_3 \cdots a_psyu$ 是 G 中长度至少为4的无弦圈。

现在我们来证明定理7-2。设 G 是弦图且包含不止一个团。假定 M_{uv} 不是团, 则它包含一对不相邻的顶点, 因此由前面的讨论知 G 包含一个长度至少为4的无弦圈。这与 G 是弦图矛盾。因此 M_{uv} 是团。

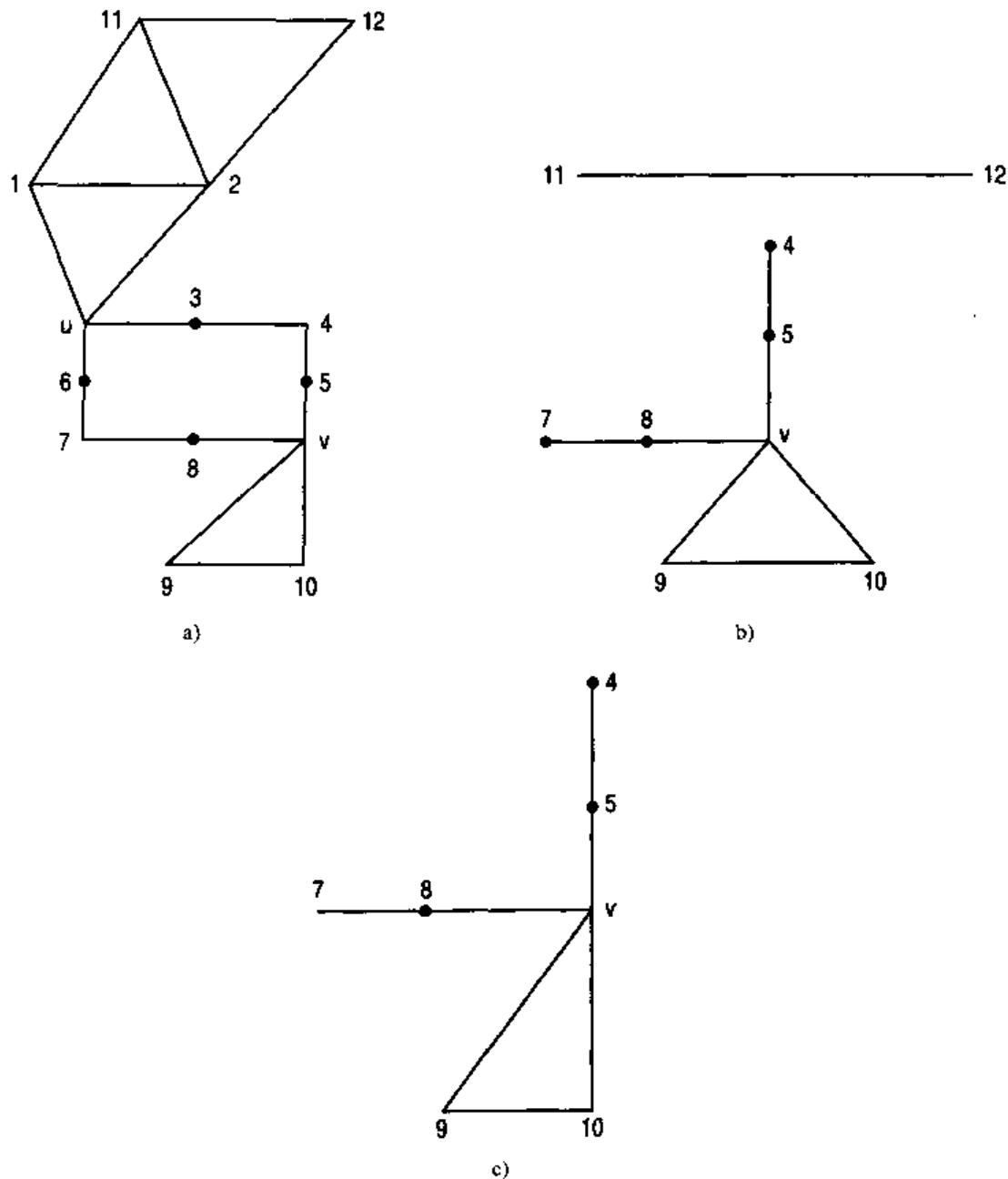


图 7-1

a) 图 G b) 图 G_{uv} c) C_{uv} = 包含 v 的 G_{uv} 的分支

反过来，假定对于每一对不相邻的顶点 u, v ， M_{uv} 是团，并假定 G 不是弦图，则存在一个无弦圈 $v_1, v_2, \dots, v_k, v_1$ ，令 $u = v_1$ ， $v = v_3$ ，则 M_{uv} 包含 $\{v_2, v_4\}$ ，且 M_{uv} 不是一个团。这与对于任何一对不相邻顶点 u, v ， M_{uv} 是团相矛盾。因此 G 是弦图。

为了检验一个图是否是弦图，对于每一对不相邻的顶点 u, v ，我们只要检验 M_{uv} 是否是一个团。如果对于每一对不相邻的 G 的顶点 u, v ， M_{uv} 刚好是一个团，我们可以得出结论： G 是弦图。否则 G 不是弦图。Iyengar(1985)给出了下面完整的算法。

算法Chordal Test—Chandra & Iyengar

输入：图 G

输出： G 是否为弦图

BEGIN

1. If G is a clique then output that G is chordal and terminate
2. For every pair of non adjacent vertices u, v do in Parallel
 - 2a. $H = G - ADJ(u)$
 - 2b. $C = \text{Component of } H \text{ containing } v$
 - 2c. $M = \Phi$
 - 2d. For each $x \in ADJ(u)$ and $y \in H$ do in parallel
 - If $(x, y) \in E$ then $M = M \cup \{x\}$
 - End parallel
- 2e. If M is a clique return CHORDAL = 1
3. End parallel
4. If for each pair of non adjacent vertices the value returned is 1 then
 - output that G is chordal
 - else
 - output that G is not chordal
 - end if

END

V 的不相邻的顶点对的个数是 $O(n)$ 。为了找出连通支，我们利用第6章给出的算法，该算法共需处理器数为 $O(n^2)$ ，执行时间为 $O(\log n)$ 。第2d步需要处理器数为 $O(n^2)$ 。因此这个算法共需处理器数为 $O(n^4)$ ，执行时间为 $O(\log n)$ 。PRAM模型应用CREW。这个用来判别弦图的方法也用来判别图的一类有趣的子类： k -树。

k -树 k -树是弦图一类非常重要的子类，下面给出定义和它的一些性质。

定义：如果一个图可由下面的递归构造得到，则称为 k -树。

1. 从 k 个元素组成的团作为一个基本图出发，这 k 个元素的团就是 k -树。
2. 对于任一 k -树 H ，增加一个新的顶点，并使得它与 H 中的一个 k 团的所有顶点相邻，而构成一个含 $k+1$ 个元素的团。

下面我们给出 k -树的特征定理。

定理7-3 下面的叙述是等价的：

1. $G = (V, E)$ 是 k -树；
2. (a) G 是连通的；
 (b) G 含有一个 k 团，但不含有 $(k+2)$ 团；
 (c) 每一个极小顶点分离集是一个 k 团。
3. (a) G 是一个弦图；
 (b) $|E| = k|V| - k(k+1)/2$ ；
 (c) G 含有一个 k 团，但不含有 $(k+2)$ 团。
4. (a) G 是连通的；
 (b) $|E| = k|V| - k(k+1)/2$ ；
 (c) 每一个极小顶点分离集是一个 k 团。

上面定理的证明留给读者自己完成。

Chandra和Iyengar的 k -树判别算法 判定 k -树的NC算法类似于判定弦图的NC算法，但不

同的是，它利用极小分离集。我们需要下面的基本结论。

定理7-4 G 是一个不完全的弦图， x 是 G 中一个单纯点（simplicial vertex）， $H = (U, F) = G - \{x\}$ ，则对于 H 中任何一对不相邻的顶点 u, v ，不存在包含 x 的极小 $u-v$ 分离集。

证明：如果 H 是一个团，则定理显然成立。现在假定 H 不是一个团， u 和 v 是两个不相邻的顶点， u 和 v 不可能都与 x 相邻，否则， u 和 v 必定相邻。因而， u 和 v 中至多有一个顶点与 x 相邻。分两种情形讨论。

情形1

不失一般性，假定 u 与 x 相邻， v 与 x 不相邻， S 是包含在 $\{x\} \cup \text{adj}(x) - \{u\}$ 中的极小 $u-v$ 分离集。 C_1 和 C_2 是分别包含 u 和 v 的 $G-S$ 的连通支，则 $C_1 \cap C_2 = \Phi$ 。因为 x 是一个单纯点，故 C_2 中没有顶点与 x 相邻，从而 C_2 中也没有顶点与 u 相邻，这是因为如果 C_2 中有一顶点 z 与 u 相邻，并且也有 z 与 x 不相邻，则在 $G-S$ 中 u 和 v 将有一条路相连。但是我们已经知道 C_2 中没有顶点与 x 相邻，因此 C_2 中的所有顶点都与 $\text{adj}\{x\} - \{u\}$ 的一个子集相邻。因而 u 和 v 将被 $R = (\text{adj}(x) - \{u\}) \cap \text{adj}(C_2)$ 分离。更进一步，很容易看出 R 也是惟一的一个包含在 $\text{adj}\{x\} \cup \{x\} - \{u\}$ 中的极小 $u-v$ 分离集。如果极小 $u-v$ 分离集 S 包含 x ，则它必定是 $\text{adj}(x) \cup \{x\} - \{u\}$ 的子集，但是我们知道在 $\text{adj}(x) \cup \{x\} - \{u\}$ 中惟一的极小 $u-v$ 分离集是 R 而其并不包含 x 。因而定理成立。217

情形2

u 和 v 都不与 x 相邻， S 是包含在 $\text{adj}\{x\} \cup \{x\}$ 中的极小 $u-v$ 分离集， C_1 和 C_2 是分别包含 u 和 v 的连通支，又因为在 C_1 或 C_2 中没有顶点与 x 相邻，因此包含在 $\text{adj}\{x\} \cup \{x\}$ 中的极小 $u-v$ 分离集 $\text{adj}(x) \cap \text{adj}(C_1)$ 和 $\text{adj}(x) \cap \text{adj}(C_2)$ ，它们都不含有 x ，由前面的情形我们知道定理成立。

下面我们给出关于 k -树的新的特征。

定理7-5 图 $G = (V, E)$ 是 k -树当且仅当

a: 对于 G 的每一对不相邻的顶点 u 和 v ，存在一个极小 $u-v$ 分离集，其中这个极小 $u-v$ 分离集是一个 k 团；

b: $m = kn - k(k+1)/2$ 。

证明：(必要性) 如果 G 是一个团，则它是 k -树。如果 G 不是一个团，则对于每一对不相邻的顶点 u 和 v ， G 的极小分离集是 k 团，更进一步有 $m = kn - k(k+1)/2$ 。

(充分性) 条件a可以导出 G 是弦图，因此我们只要证明 G 有一个 k 团，而不包含 $(k+2)$ 团。首先，如果 G 是一个团，则它不可能是 $(k+2)$ 团。所以假定 G 不是一个团，由于 G 含有一个极小分离集， G 必含有一个 k 团。我们通过对 G 的顶点数进行归纳证明。所有包含1, 2, 3, 4个顶点的图不含有 $(k+2)$ 团。假定顶点小于或等于 $n-1$ 的所有的满足条件a和b的图不含有团。对含有 n 个顶点的满足条件a和b的图进行讨论，由于 G 是弦图，在 G 中存在一个单纯点 x 。令 $H = (U, F) = G - \{x\}$ ， $W = \{x\} \cup \text{adj}(x)$ 。于是 $|U| = n-1$ ， $|W| < n-1$ ，否则， G 是一个团。现在考虑如下两种情形。

情形1

假定 H 是 $(n-1)$ 团， H 满足条件a和b。由归纳假设， H 不含有 $(k+2)$ 团，由于 G 不是一个团，在 U 中至少存在一个顶点与 x 不相邻，更进一步，所有极小 $z-x$ 分离集只能是 $\text{adj}(x)$ ，这里 $z \notin \text{adj}(x)$ 。由于 G 满足条件a， $\text{adj}(x)$ 必定是一个 k 团，由此推出在 G 中至少有一个 k 团，而不含有 $(k+2)$ 团。218

情形2

H 不是一个团，在 G 中，对于任何一对不相邻的顶点 $w, z \in V$ ，存在一个极小分离集，它是

k 团。因为 x 不可能包含在任一个极小 $w-z$ 分离集中，所以 H 满足条件a。从极大团 W 开始，考虑图 G 的递归构造。假设 y 是后来被加入到这个递归构造过程的顶点， y 与 x 不相邻，但与在 $\text{adj}(x)$ 中的一个团相邻，更进一步，任何极小 $x-y$ 分离集都包含在 $\text{adj}(x)$ 中。如果 $|\text{adj}(x)| < k$ 或者 $|\text{adj}(y)| < k$ ，则在 G 中存在一个顶点数小于 k 的极小 $x-y$ 分离集，这与假设矛盾。如果 $|\text{adj}(x)|$ 和 $|\text{adj}(y)|$ 都小于 k ，可得到类似的矛盾，因而有 $|\text{adj}(x)| = |\text{adj}(y)| = k$ 。因此， H 有 $m-k = k(n-1)-k(k+1)/2$ 条边，所以 H 也满足条件b。由归纳假设， H 不包含 $(k+2)$ 团。故 G 也不含有 $(k+2)$ 团。

下面的串行算法用来检测图 G 是否为 k -树，算法的内容类似于Chordal Test 算法的内容。

算法 k -tree Test—Chandra and Iyengar

输入： n 个顶点和 m 条边的图 G

输出：图 G 是否为 k -树

BEGIN

1. If the equation $m = kn - k(k+1)/2$ does not have a positive integer root $< n$ for k , G is not a k tree, Terminate
- End if
2. Let $k < n$ be a positive integer root of the above equation
3. For every non adjacent unordered pair of vertices u and v of V do
4. $G_u = G - \text{ADJ}(u)$
5. $C_u =$ the component of G_u containing v
6. $M_{uv} = \{x : x \in \text{ADJ}(u) \text{ and } x \text{ is adjacent to some vertex in } C_u\}$
7. If M_{uv} induces a k -cliques in G then continue
- else
- G is not a k tree
- terminate
- End if
8. End for
9. G is a k tree

END

上面算法的正确性由下面定理来确保。

定理7-6 图 G 是 k -树当且仅当：

1. $m = kn - k(k+1)/2$;
2. 对于图 G 中所有不相邻的顶点对 u 和 v ，算法中的 M_{uv} 导出一个 k 团。

证明：(充分性) 很容易看出 M_{uv} 是 G 中的一个极小 $u-v$ 分离集，更进一步，定理7-5的条件被满足，由定理7-5， G 是 k -树。

(必要性) 假设 G 是 k -树，则显然有 $m = kn - k(k+1)/2$ ，因此 k 是满足上述方程的正整数解。更进一步， G 不是一个团，很容易证明上述方程至多有一个小于 n 的正整数根。现在，假定对于一对不相邻的顶点 u, v ， M_{uv} 不是 k 团，那么，因为 M_{uv} 是一个极小分离集，因此它不是 k -树，这与假设矛盾。

下面的定理可直接证明。

定理7-7 算法 k -tree Test 可正确检测图 G 是否是 k -树。

k -tree Test串行算法可以并行化为NC算法。我们有下面结论。

定理7-8 存在检测 k -树的并行算法，该算法共需处理器数为 $O(n^4)$ ，执行时间为 $O(\log n)$ 。

Naor, Naor和Schaffer算法 Naor, Naor和Schaffer (1989) 主要从事关于弦图的并行算法研究，并设计了关于弦图的一些NC算法。我们从他们关于弦图的判别理论出发。他们算法的主要优点在于算法的正确性很容易从弦图的定义中得到，即图不含有长度大于3的诱导无弦圈。另一个优点是在稀疏图上，它利用渐进减少的处理器。算法中我们将利用下面一些记号。

$$N(v) = \text{与 } v \text{ 相邻的顶点集合};$$

$$G-v = V-\{v\} \text{ 的诱导图};$$

$$\text{如果 } W \subseteq V, G-W = V-W \text{ 的诱导图}.$$

并行算法由下面的定理给出。

定理7-9 图 $G = (V, E)$ 不是一个弦图当且仅当它含有一个顶点 v ，且具有下列性质： $(G-v)-N(v)$ 的一个连通支与 $N(v)$ 的两个彼此不相邻的顶点相邻。

证明：假定 $G = (V, E)$ 是一个图，存在一个顶点 v 使得 $w_1, w_2 \in N(v)$ ，并且它们都与 $(G-v)-N(v)$ 的一个连通支相邻， w_1, w_2 不相邻（见图7-2）。设 u_1, u_2 是该连通支的顶点，且 w_1, w_2 分别与 u_1, u_2 相邻， w_1, v 和 w_2 属于一个长度大于3的无弦圈，因此 G 不是一个弦图。

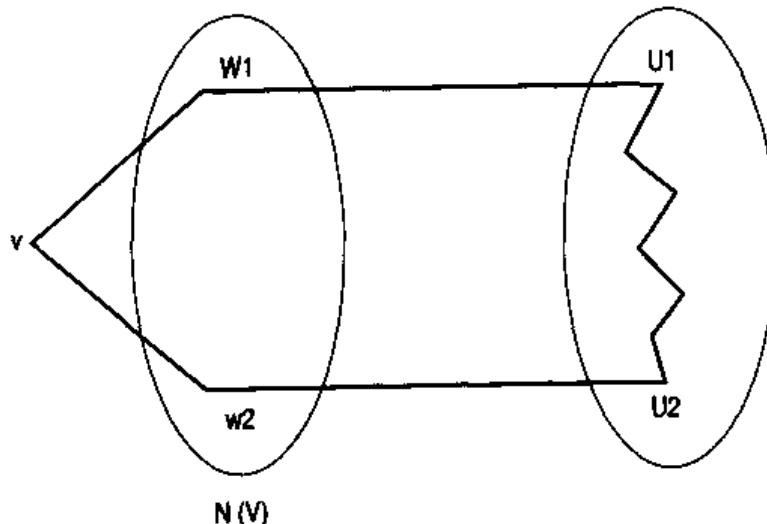


图7-2 圈的形成

反过来，假定 G 不是一个弦图， $v_1, v_2, \dots, v_k, v_1$ 导出一个长度为 $k > 4$ 的无弦圈。考虑顶点 v_1 ，在图 $(G-v_1)-N(v_1)$ 中， v_3, v_4, \dots, v_{k-1} 位于相同的连通支上， v_2, v_k 属于 $N(v_1)$ 并且它们都与 $(G-v_1)-N(v_1)$ 的连通支相邻，该连通支包含 v_3, v_4, \dots, v_{k-1} ，而 v_2 与 v_k 不相邻。因此定理得证。

根据定理7-9，为了检验图 G 是否是弦图，我们可以检验图 G 中是否存在一点 v 满足定理7-9的条件。对于每一个顶点 v ，首先必须求出 $(G-v)-N(v)$ 以及它的连通支。然后对于 $N(v)$ 中每一对不相邻的顶点 (u, w) ，检验是否有某个连通支与 u 和 w 都相邻。如果这样的连通支存在的话，我们可以得出结论，即 G 不是弦图。并行算法如下：

算法Chordal Test—Naor, Naor and Schaffer

输入：简单无向图 $G = (V, E)$

输出： G 是否为弦图

```

BEGIN
  1. (a) For each  $v \in V$  do in parallel
    (b) Find all the connected components of  $(G - v) - N(v)$  and Number them
    (c) End parallel
  2. (a) For every  $u, w \in V$  do in parallel
    (b) If  $(u, w)$  is not an edge but  $(v, u)$  and  $(v, w)$  are edges then select the pair  $(u, w)$ 
    (c) End parallel
  3. (a) For each  $v \in V$  do in parallel
    (b) For each  $u \in N(v)$  do in parallel
    (c) Compute a sorted list of connected components computed in step 1b to which  $u$  is
        adjacent
    (d) End parallel
    (e) End parallel
  4. (a) For every vertex  $v \in V$  do in parallel
    (b) For every pair of vertices  $u, w \in N(v)$  such that  $(u, w)$  is not adjacent do in parallel
    (c) Verify if there is an entry in the list of connected components to which both  $u$  and  $w$ 
        are adjacent. If so, conclude  $G$  is not chordal
    (d) End parallel
    (e) End parallel
  5. (a) If step 4c does not declare that  $G$  is not chordal for any pair  $(u, w)$  and  $v \in V$  then
      conclude that  $G$  is chordal
END

```

复杂度分析 第1步可以利用第6章中的连通支算法，用CREW PRAM模型，需处理器

222 $O(n^2)$ ，执行时间为 $O(\log^2 n)$ 。在第1步中我们对所有 n 个顶点并行执行，因此第1步共需处理器 $O(n^3)$ ，执行时间为 $O(\log n)$ 。第2步的复杂度依赖于图的数据结构，如果考虑到它的相邻矩阵，第2步可表示如下：

```

If  $(A(u, w) = 0 \text{ and } A(v, u) = 1 \text{ and } A(v, w) = 1)$  then select
  the pair  $(u, w)$ 
end if

```

这个过程共需时间为 $O(1)$ 。如果一个图用相邻列表来表示，第2(b)步共需处理器 $O(m)$ ，执行时间为 $O(\log n)$ (m 指图 G 中边的个数)。因此，第2步共需处理器数为 $O(mn)$ ，时间为 $O(\log n)$ 。第3(c)步用一个处理器，所需时间为 $O(\log^2 n)$ ；第3(b) ~ (d)是并行循环，循环执行的次数是顶点 v 的度数，又因为所有顶点的度和为 $2m$ ，因此第3步所需的处理器数为 $O(mn)$ ，执行时间为 $O(\log^2 n)$ 。第4步所需的处理器数为 $O(mn^2)$ ，执行时间为 $O(\log n)$ ：每一个三元数组 (v, u, w) 使得 v 与 u 相邻， w 与 u 彼此不相邻。给第3步中每对 (u, w) 得到的连通支列表的每一个分支分配一个处理器。对所有顶点对 (u, w) ，连通支列表的总的长度为 $O(mn)$ 。对于这个列表中的每一个元素至多分配 $n-2$ 个处理器，一个处理器对应 w 的每次选取。总的处理器数是 $O(mn^3)$ 。对于三元数组 (v, u, w) ，与元素 C 相应的处理器在 (v, w) 的分支列表中对 C 执行二分搜索，并把结果记录下来。(如果求出 C 则结果为1，否则结果为0)。对于每一个固定的 v ，与含有 v 并把 v 作为第一个元

素的三元数组所有的处理器执行的结果做布尔OR运算。由第4步，我们得到 n 个布尔值。第5步中，我们对这 n 个布尔值执行布尔OR运算，用 $O(n)$ 个处理器，执行时间为 $O(\log n)$ 。因此，该算法所用的总的处理器数为 $O(mn^2)$ ，执行时间为 $O(\log^2 n)$ ，且执行CREW PRAM模型。

Naor, Naor 和 Schaffer (1980) 对于下列问题也设计了NC并行算法：

1. 列出弦图的所有极大团；
2. 求出弦图的最优染色；
3. 求出弦图的树表示；
4. 求出弦图的最大独立集；
5. 求出弦图的最小团覆盖。

7.2 弦图的极大团

首先让我们来看一个求弦图的所有极大团的算法，一个弦图至多有 n 个极大团。首先定义双团 (bi-clique)。

如果图 $G = (V, E)$ 的顶点集 V 可以分成两个集合 A 和 B ，使得由 A 和 B 所导出的图都是团，我们称 G 为双团。[223]我们用分而治之法来求一般弦图 G 的所有极大团。如果 V 被分成任意两个大小几乎相等的不相交的子集，可递归地计算由 A 和 B 导出的图的极大团，于是对 G_A 和 G_B 的每一对极大团 p 诱导子图 $G_{P \cup Q}$ 的双团。有上面讨论可知，用来求极大团的算法也用来计算双团的所有极大团。

对于弦图 $G = (V, E)$ ，我们感兴趣的是计算所有的极大团。 $V = A \cup B$ 是图 G 的顶点的一个划分，满足 $|A|$ 和 $|B|$ 几乎相等。 P 和 Q 分别是 G_A 和 G_B 的极大团，这里 G_A 表示由 A 导出的图， $P \cup Q$ 生成一个双团记为 $G_{P \cup Q}$ 。对于任一顶点 $v \in P$ ，记 $N_Q(v) = \{w \in Q : (v, w) \text{ 是 } P \cup Q \text{ 中的一条边}\}$ 。同样地，对于任意顶点 $v \in Q$ ，记 $N_P(v) = \{w \in P : (v, w) \text{ 是 } P \cup Q \text{ 中的一条边}\}$ 。

有趣的是 $\{N_Q(v) : v \in P\}$ 可以用包含关系进行排序。即对于任意两个顶点 $v_1, v_2 \in P$ ，要么 $N_Q(v_1) \subseteq N_Q(v_2)$ ，要么 $N_Q(v_2) \subseteq N_Q(v_1)$ 。

定理7-10 可利用集合包含关系对集合 $\{N_Q(v) : v \in P\}$ 排序。

证明：反证。假定 $u, v \in P$ ，满足 $N_Q(u) \not\subseteq N_Q(v)$ 且 $N_Q(v) \not\subseteq N_Q(u)$ 。又假定 $w_1 \in N_Q(u)$ 但 $w_1 \notin N_Q(v)$ 。同样地， $w_2 \in N_Q(v)$ 但 $w_2 \notin N_Q(u)$ 。由于 $w_1, w_2 \in Q$ ，它们是相邻的。

又由于 $u, v \in P$ ，它们也是相邻的，见图7-3。但是根据 w_1, w_2 的选取，有 u 与 w_2 不相邻， v 与 w_1 不相邻。因此， $u-v-w_1-w_2$ 导出一个无弦图。这与 G 是弦图矛盾。定理得证。

推论7-1 假定 $C = C_p \cup C_q$ 是弦图双团 $G_{P \cup Q}$ 的任一极大团，这里 $C_p \cap P$ 与 $C_q \cap Q$ 都非空集，则 C_p 是集合 $\{N_p(x) | x \in C_q\}$ 的最小集， C_q 是集合 $\{N_q(x) | x \in C_p\}$ 的最小集。[224]

算法Biclique-MC

输入：图 G ， $A \subseteq V$ 且 $B \subseteq V$

$$A \cup B = V, A \cap B = \emptyset$$

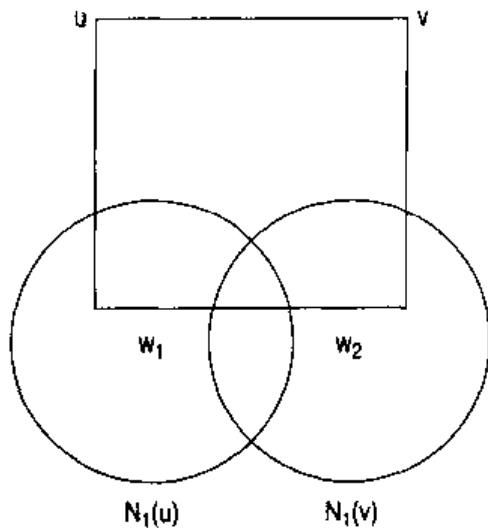


图7-3 圈的形成

团 P_A

团 Q_B

输出：由 $G_{P \cup Q}$ 产生的团列表， $G_{P \cup Q}$ 是 G 的极大团

BEGIN

1. Compute $\{N_Q(v) : v \in P\}$
2. Sort these sets with set inclusion and sort the vertices of P according to this order
3. Compute $\{N_P(u) : u \in Q\}$
4. Sort these vertices with set inclusion and sort the vertices of Q according to this order
5. Find the vertices v_i in the list P such that $N_Q(v_i)$ is a proper superset of $N_Q(v_{i+1})$
6. Each vertex v chosen in step 5 yields a different maximal clique $Q_A(v) \cup Q_B(v)$. $Q_A(v)$ consists of v and every vertex of P with the same or larger neighbor set in Q . That is,

$$Q_A(v) = \{v\} \cup \{w : N_Q(w) \supseteq N_Q(v)\}$$

Given $Q_A(v)$, we can find $Q_B(v)$ by Corollary 7.11 we may also need to include the cliques P and Q

7. Decide which maximal cliques of the biclique $G_{P \cup Q}$ are maximal cliques in G

END

在讨论上述算法的复杂度之前，我们首先来看一看如何执行第7步。定义一个记号：如果 $v \notin P \cup Q$, $u(P, v) = P$ 的顶点排序列表中最高顶点，使得 $N_Q(u(P, v)) \subseteq N_Q(v)$ 。对于满足上述条件的顶点可能存在。类似地，我们也可定义 $u(Q, v)$ 。

定理7-11 假设 C 是 $G_{P \cup Q}$ 的一个极大团，且满足

1. $C \cap P \neq \emptyset$;
2. $C \cap Q \neq \emptyset$;
3. 对于任一顶点 $v \notin P \cup Q$, C 包含 $u(P, v)$ 或包含 $u(Q, v)$ 。

则 C 也是 G 的极大团。

读者可参照Naor, Naor和Schaffer(1989)给出定理7-11的证明。

根据上面的定理，可以执行算法Bi-Clique-MC 的第7步。下面我们讨论算法Bi-Clique-MC 的复杂度。

复杂度分析 第1~4步需用 $O(n^2)$ 个处理器，执行时间为 $O(\log n)$ ；第5步用同样多的处理器且执行 $O(1)$ 时间；第6步中， $Q_A(v)$ 是 P 的前缀（prefix）， $Q_A(v)$ 的大小是 P 中 v 的秩， $Q_B(v)$ 与 $N_Q(v)$ 相同，因而对于每一个团 $Q(v)$ ，用一个处理器的执行时间为 $O(\log n)$ 。对于 $Q(v)$ 中每一个元素，我们设计用一个处理器来输出结果，因此执行时间为 $O(\log n)$ ，处理器数为 $O(m + n)$ 。计算 $u(P, v)$ 和 $u(Q, v)$ 所需的处理器数为 $O(n^2)$ ，执行时间为 $O(\log n)$ 。因此第7步所需的处理器数为 $O(n^2)$ ，执行时间为 $O(\log n)$ 。

我们已经了解了求双团的所有极大团的过程，下面来描述求一个图的所有极大团的NC算法。

算法Max-Cliques

输入：图 $G = (V, E)$

输出：图 G 的极大团列表

BEGIN

1. Let $V = A \cup B$,

where $A \cap B = \Phi$ and $|A| \approx |B|$ (approx.)

2. Compute all the maximal cliques of G_A and G_B recursively

3. For every pair of maximal cliques P of G_A and Q of G_B compute all the maximal cliques of G in the induced subgraph $G_{P \cup Q}$ which is a bi-clique

4. Eliminate duplicate cliques occurring in more than one bi-clique

END

由于在每一次递归调用中顶点的数目减半，因而总的递归调用次数为 $O(\log n)$ 。

在每一次递归调用中我们利用Bi-Clque-MC算法，去掉完全相同的团需用 $O(n^4)$ 个处理器，总的执行时间为 $O(\log^2 n)$ 。因此计算弦图的极大团用 $O(n^4)$ 个处理器，共需执行时间为 $O(\log^2 n)$ 。

弦图的一些子类 C. Xavier 等人 (1990) 已经定义了树的路簇的两类交图。[226] 如果树 T 中不存在这样一个顶点，它是路簇中至少一条路的内部顶点，这样的路簇称为是完美的 (perfect)。树的完美顶点路簇的交图称为完美顶点路图或 PV 图。如果树 T 中不存在这样一条边，它位于路簇中至少一条路中，则称这样的路簇是紧的 (compact)。树的紧顶点路簇的交图称为紧顶点路图或 CV 图。由观察可知，CV 图是 RD 图和 PV 图的子集，也是 DV 图的子集。

7.3 CV图的特征

由定义知，PV图是UV图，PV图不必是RDV图，RDV图不必是PV图，CV图不必是区间图，每一个区间图不必是CV图。下面我们给出CV图的几个特征。

定理7-12 对于每一个图 G 下面的叙述是等价的。

- a. G 是树 T_1 中边不相交的路簇 F 的交图；
- b. G 是树 T_2 中边不相交的子树簇 F 的交图；
- c. 存在一个树 T 满足 $V(T) = C(G)$ ，使得 $F = \{T[C_v(G)]\}, v \in V\}$ 是树 T 的边不相交的路簇；
- d. G 不包含 $K_4 - e$ 或 $C_n (n \geq 4)$ 作为它的诱导子图。注意 $K_4 - e$ 由 K_4 去掉一条边 e 得到，其中 K_4 是含 4 个顶点的完全图； C_n 是含 n 个顶点的圈；
- e. G 是一个块图。

定理的证明留给读者作为练习。

假定 G 是一个 CV 图， C 是 G 的一个分离团。记 $G_i = G[V_i \cup C]$ 是分离子图， $1 < i < r$, $r \geq 2$ 。

命题7-1 $W(G_i)$ 是一个独立集， $1 < i < r$ 。

证明：很显然， $W(G_i)$ 是非空的。如果 $|W(G_i)| > 2$ ，则不难证明 G_i 含有 $K_4 - e$ 作为它的诱导子图，这与 G 是一个块图矛盾。[227]

推论7-2 如果图 G 的每一个分离子图都是 CV 图，则图 G 是 CV 图。

证明：假定每一个分离子图 G_i , $1 < i < r$, 都是 CV 图。因此每一个 G_i 都是块图。又根据命题7-1，每一个 $W(G_i)$ 都是独立集，因此 G 是一个块图， G 是 CV 图。

CV图判别算法 我们提供了一个线性时间算法来判别 CV 图。如果图是 CV 图，那么就可以构造一个交图模型。

算法CV test

输入：用相邻列表表示的图 $G = (V, E)$

输出：如果图 G 不是一个CV图，输出“No”，否则输出图 G 的一个CV团树 T

BEGIN

1. If G is not a block graph, then output ‘No’
2. Find all clique of G . Let C_1, C_2, \dots, C_n be the cliques of G
3. Find the set $\{C_{i_1}, C_{i_2}, \dots, C_{i_n}\}$ of cliques containing v_i ,
- 1 $\leq i \leq n$
4. $T = T(V_0, E_0)$, where V_0 is the set of all clique of G and
 $E_0 = \Phi$
5. For $i = 1$ to n do
6. $T = T(V_i, E_i)$, where $V_i = V_0$ and
7. $E_i = E_{i-1} \cup \{G_i, G_j, 1 < j < r_i - 1\}$
8. If $r_i > 1$ return else $E_i = E_{i-1}$
9. End for

END

7.4 路图判别

本节我们介绍C. Xavier 提出的判别UV图（路图）的一个NC并行算法。我们采用与弦图判别完全不同的策略。从下面两个简单的事事实谈起。

事实7-1 如果 G 是一个UV图， v 是 G 的一个顶点，与包含顶点 v 的团相对应的团树 T 的顶点导出 T 中的一条路。

事实7-2 在一个区间图中，极大团可以被线性排序且对于任一顶点，包含 v 的团在线性排序中是连续的。换句话说，区间图的团树是一条路，且对于任一顶点 v ，与包含 v 的团相对应的顶点组成一条子路。

如果 G 是UV图，根据定理7-14可以证明，包含顶点 v 的极大团的并(union)导出一个区间图。对于 G 的每一个顶点 v ，我们可以并行验证包含 v 的极大团的并是否是区间图。如果对于某个顶点，包含它的极大团的并不是区间图，我们可以得出 G 不是UV图。如果对于任一顶点 v ，包含它的极大团的并都是区间图，我们可以得出相应的区间图的PQ树表示公式。对于这些PQ树表示，我们试图给出图 G 的团树表示公式，如果不能写出表示公式，便可以得到图 G 不是UV图。7.4.1节介绍一些基本概念和几个简单的事事实，这些事实对于算法设计是必须的；7.4.2节主要介绍算法总框架；后面几节主要给出具体算法并讨论算法的正确性与复杂度。

7.4.1 一些概念和事实

我们首先给出几个记号，接着考虑极大团，因而以下所说的团仅指极大团。假定 G 是一个弦图， T 是它的团树表示，如果大写的阿拉伯字母 C 表示 G 的团，那么相应的小写 c 表示树中对应于 C 的顶点。

如果 a 是 G 的顶点，则 C_a 表示包含 a 的图 G 的团簇，即

$$C_a = \{C : C \text{ 是图 } G \text{ 的团且 } a \in C\}$$

令 G_a 表示 C_a 中所有团的并导出的图。如果 a 和 b 是 G 的两个顶点， $C_{a+b} = C_a \cup C_b$ ， $C_{ab} = C_a \cap C_b$ 。在 C_{a+b} 和 C_{ab} 中所有团的并导出的图分别记 G_{a+b} 和 G_{ab} 。

考虑图7-4a所示的图 G , 它的团为 $C_1 = \{a,b,c,d\}$, $C_2 = \{e,d\}$, $C_3 = \{d,f,g\}$, $C_4 = \{d,g,h\}$, $C_5 = \{s,h,r\}$, $C_6 = \{h,r,p\}$, $C_7 = \{p,q\}$ 。

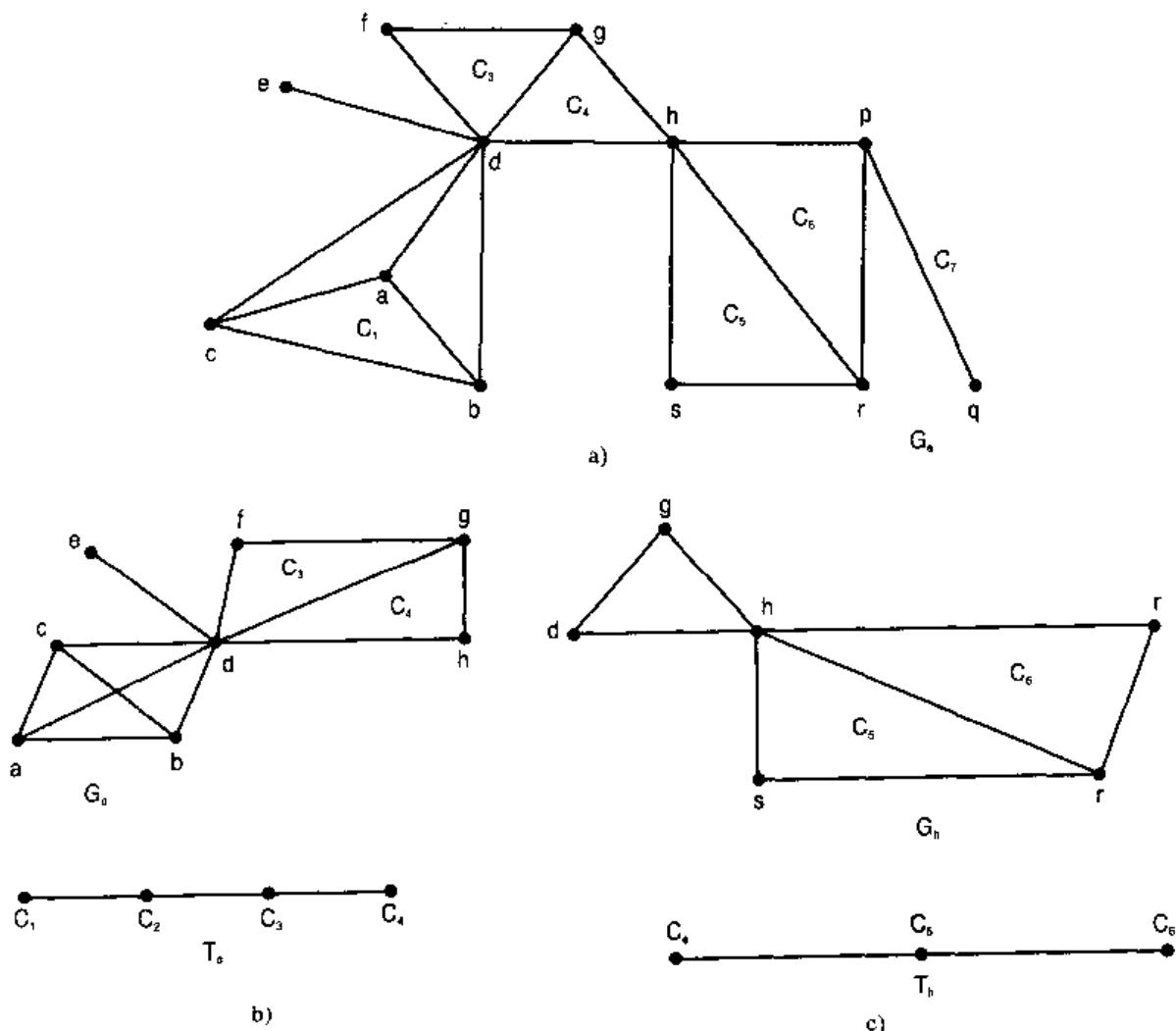


图 7-4

a) 弦图 G 的表示 G_a 。b) 图 G_a 和它的团树 T_a 。c) 图 G_b 和它的团树 T_b

$$C = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$$

由于顶点 a 仅在 C_1 中, 因此 $C_a = \{C_1\}$ 。同样地, $C_b = \{C_1\}$, $C_c = \{C_1\}$ 。 d 包含在 C_1, C_2, C_3 和 C_4 中, 因此 $C_d = \{C_1, C_2, C_3, C_4\}$ 。类似地, $C_e = \{C_2\}$, $C_f = \{C_3\}$, $C_g = \{C_3, C_4\}$, $C_h = \{C_4, C_5, C_6\}$, $C_s = \{C_5, C_6\}$, $C_{d+h} = \{C_1, C_2, C_3, C_4, C_5, C_6\}$, $C_{d+h} = \{C_4\}$ 。

我们首先证明关于弦图的一般性结论, 这些结论直接或间接地用于算法设计中。

定理7-13 假定 G 是一个弦图, T 是它的团树表示, 则 T 的每一个非叶子顶点表示 G 的一个分离团。

证明: 设 c 是 T 的非叶子顶点, 并且使得 c_1, c_2 都与 c 相邻。 C, C_1, C_2 是这样的团, 满足任一个团都不包含在另外两个团中。因此存在顶点 a 和 b , 使得

$$a \in C_1, \text{ 但 } a \notin C, \text{ 因而有 } a \notin C_2;$$

$$b \in C_2, \text{ 但 } b \notin C, \text{ 因而有 } b \notin C_1.$$

我们证明 G 中从 a 到 b 的每一条路必经过 C 的一个顶点。假设 $P = a_1, a_2, a_3, \dots, a_k$ 是任一条从 a

到 b 的路，这里 $a = a_1, b = a_k$ 。记 S_1, S_2, \dots, S_k 是与顶点 a_1, a_2, \dots, a_k 相对应的 T 的子树。

记 $S = S_1 \cup S_2 \cup \dots \cup S_k$ ，则 S 是 T 的子树， c_1 和 c_2 都在 S 中，因而 T 中从 c_1 到 c_2 的一条路完全落在 S 中，因此 $c \in S$ 。这意味着存在某个 i ，使得 $c \in S_i$ 。因此 $a_i \in C$ 。由于 P 是任意的，每一条从 a 到 b 的路必经过 C 的一个顶点。这证明了 C 分离 a 和 b ，因此 C 是一个分离团。

现在我们来证明一个主要结论，它是下面算法设计的基础。

定理7-14 如果 G 是一个UV图，则对于 G 的任一顶点 a ， G_a 是一个区间图。

[231] 在证明这个定理之前，我们用一个例子来说明这个结论。考虑图7-5a所示的弦图。这个图只

有6个团，分别是 $C_1 = \{a, b, f, g\}$ ， $C_2 = \{b, f, g, h\}$ ， $C_3 = \{b, g, h, k\}$ ， $C_4 = \{f, g, h, d\}$ ， $C_5 = \{f, h, l\}$ ， $C_6 = \{p, l\}$ 。这里 g 包含在 C_1, C_2, C_3 和 C_4 中，因此 $C_g = \{C_1, C_2, C_3, C_4\}$ 。图7-5b表示 G_g ，不难验证， G_g 不是一个区间图。因此，根据定理7-14， G 不是一个UV图。

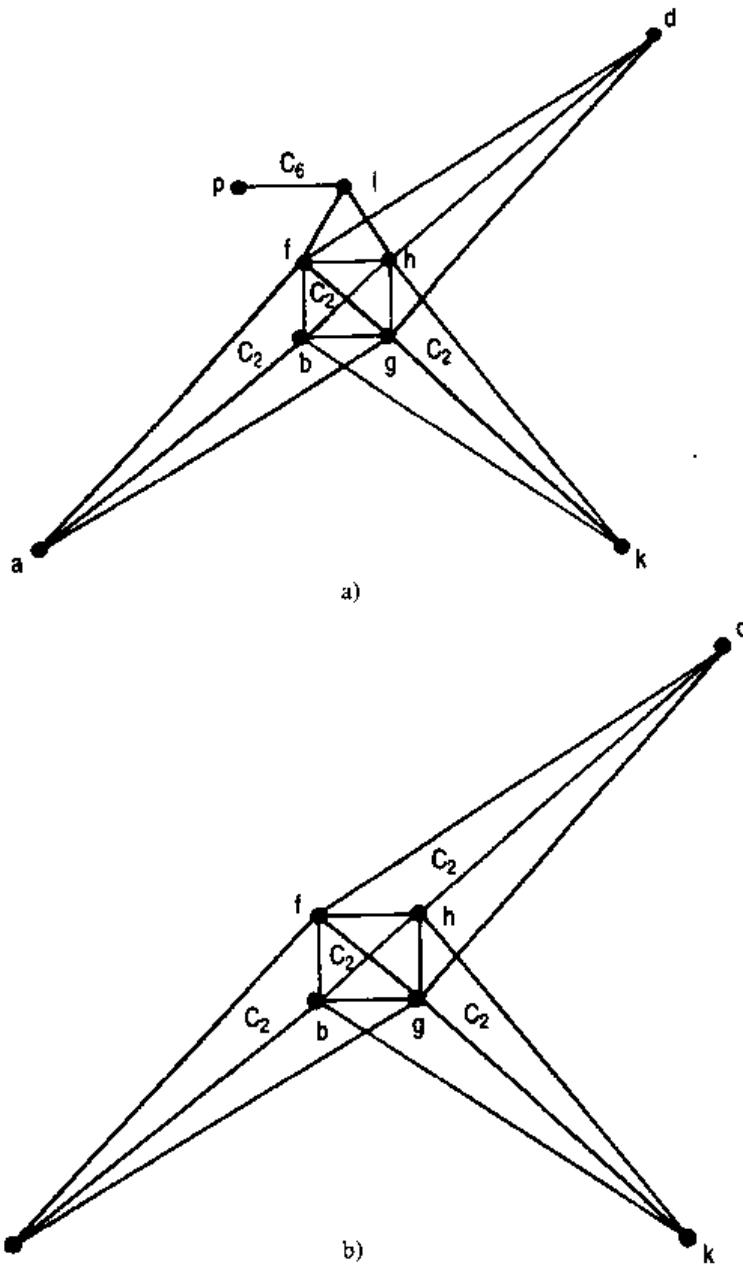


图 7-5

a) 用来验证定理7-19的弦图 G b) 图 G_g 不是区间图

证明：设 $P_a = C_1, C_2, \dots, C_k$ 是 T 中相应于顶点 a 的一条路，则 $G_a = C_1 \cup C_2 \cup \dots \cup C_k$ 。 G_a 的团是 C_1, C_2, \dots, C_k 。已知 C_1, C_2, \dots, C_k 是线性排序，且对于 G_a 中任一顶点 b ， G_a 中包含 b 的团连续出现。假定 b 是 G_a 的任一顶点， G 中包含 b 的团组成 T 中一条路，记这条路为 $P_b = c'_1, c'_2, \dots, c'_r$ 。

由于 $b \in G_a$ ， $P_a \cap P_b$ 非空，又因为 T 是一个树， P_a 和 P_b 是它的两条路且 $P_a \cap P_b$ 非空， $P_a \cap P_b$ 是 P_a 和 P_b 的子路。特别地， $P_a \cap P_b$ 是 P_a 的子路，记 $P_a \cap P_b = c_i, c_{i+1}, \dots, c_m$ ，这意味着包含 b 的 G_a 的团是 C_i, C_{i+1}, \dots, C_m 。因此 G_a 是一个区间图。

定理7-14反过来不成立。考虑图7-6a，它的6个团分别为 $C_0 = \{a, b, d, e, f\}$ ， $C_1 = \{a, b, d, e, h\}$ ， $C_2 = \{a, b, d, i\}$ ， $C_3 = \{a, b, d, e, f, j\}$ ， $C_4 = \{a, e, l\}$ ， $C_5 = \{a, l, q\}$ ， $C_6 = \{e, f, r\}$ 。

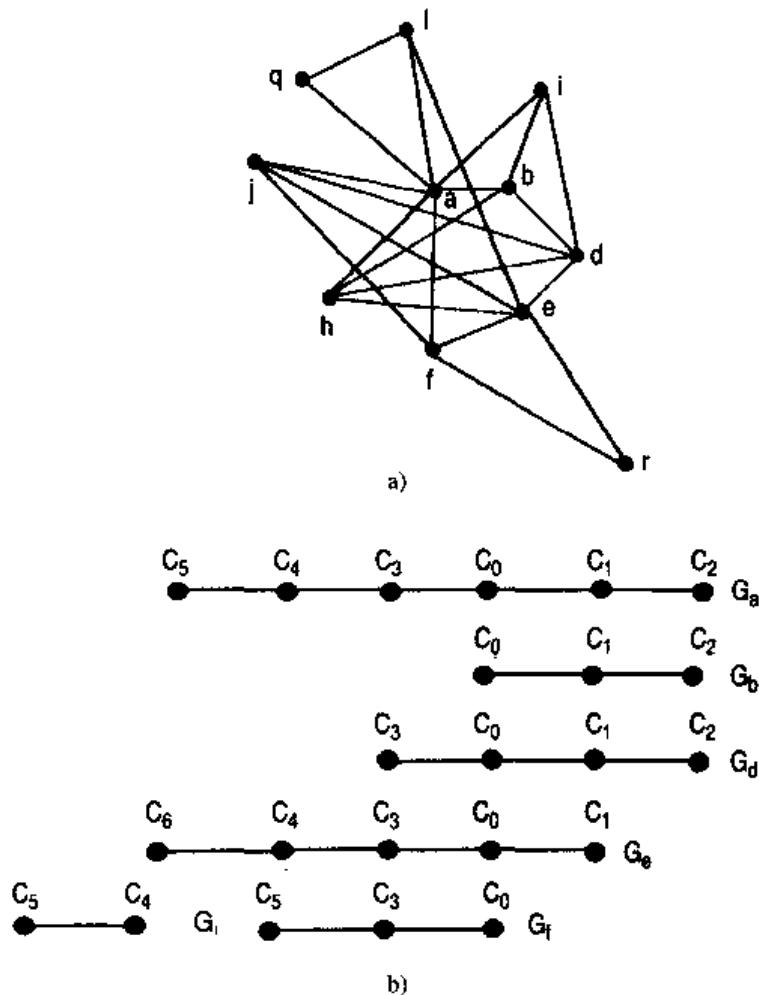


图 7-6

a) 弦图 G 不是 UV 图 b) G_a, G_b 等的区间表示

$$C_a = \{C_0, C_1, C_2, C_3, C_4, C_5\}$$

$$C_b = \{C_0, C_1, C_2\}$$

$$C_d = \{C_0, C_1, C_2, C_3\}$$

$$C_e = \{C_0, C_1, C_3, C_4, C_6\}$$

$$C_f = \{C_0, C_1, C_6\}$$

$$C_l = \{C_4, C_5\}$$

$$C_q = \{C_5\}$$

$$\begin{aligned}C_1 &= \{C_6\} \\C_2 &= \{C_1\} \\C_3 &= \{C_2\} \\C_4 &= \{C_3\}\end{aligned}$$

图7-6b描绘 $G_a, G_b, G_c, G_d, G_e, G_f$ 的区间表示，因此对于 G 的每个顶点 v , G_v 是区间图。我们可以证明图 G 不是UV图。

推论7-3 如果 a, b 是UV图 G 的两个顶点，则 G_{ab} 是区间图。

证明： G_a 是一区间图， G_{ab} 是 C_{ab} 中包含 b 的团的并的导出图。因而，根据定理7-14， G_{ab} 是区间图。

推论7-4 如果 G 是一个UV图， a_1, a_2, \dots, a_k 和 b 是 G 的顶点，且 $G_{a_1+a_2+\dots+a_k}$ 是UV图，则 $G_{(a_1+a_2+\dots+a_k)b}$ 是区间图。

7.4.2 算法概述

本节我们介绍算法是如何实现的。给定一个一般图 G ，利用Klein算法验证图 G 是否是弦图。如果 G 不是弦图，则 G 不是UV图。如果 G 是弦图，我们列出 G 的所有的团，并且对 G 的每一个顶点 a ，求出 G_a 。对于每一个顶点 a ，我们并行验证 G_a 是否是区间图。根据定理7-14，对于 G 的每一个顶点 a ，如果 G_a 不是区间图，则可知 G 不是UV图。如果 G_a 是区间图，可以求出 G_a 的团树表示以及它们的线性排序。假定 T_a 是由团的线性排序组成的一条路，如果 C_1, C_2, \dots, C_t 是区间图 G_a 的团的线性排序，则 T_a 表示路 c_1, c_2, \dots, c_t 。

记 $\{a_1, a_2, \dots, a_n\}$ 是 G 的顶点集， n_{a_i} 为包含 a_i 的团的个数。假定 a_1, a_2, \dots, a_n 按这样方法排序使得

$$n_{a_1} > n_{a_2} > \dots > n_{a_n}$$

开始我们使用 $\frac{n}{2}$ 个处理器并行处理，每个处理器做连接两个图 G_{a_i} 的操作并且验证它们的并是否是UV图。一个处理器执行 $G_{a_1+a_2}$ 并且验证它是否是UV图，另一个处理器执行 $G_{a_3+a_4}$ 并且验证它是否是UV图，等等。即如下操作：

```
For i = 1 to n/2 do in parallel
    Check whether  $G_{(a_{2i-1}+a_{2i})}$  is a UV graph
End parallel
```

当 $n = 8$ 时，这个过程如图7-7所示。上述过程可描述成如下的NC-UV(G)算法。

算法NC-UV(G)

输入：无向简单连通图 G

输出： G 是否为UV图

1: Check if G is chordal. If G is not chordal abort the algorithm because G is not a UV graph

2: List the cliques of G and also list G_a for every vertex a of G

3: Arrange the vertices of G as a_1, a_2, \dots, a_n , such that

$$n_{a_1} > n_{a_2} > n_{a_3} > \dots > n_{a_n}$$

Assign $k_0 = n$

Rename G_{a_i} as G_i for $1 \leq i \leq n$

4:

4.1 For $j = 1$ to $\lceil \log n \rceil$ do4.2 $k_j = k_{j-1}/2$ 4.3 For $i = 1$ to k_j , do in parallel4.4 PROCESS ($2i-1, 2i$)

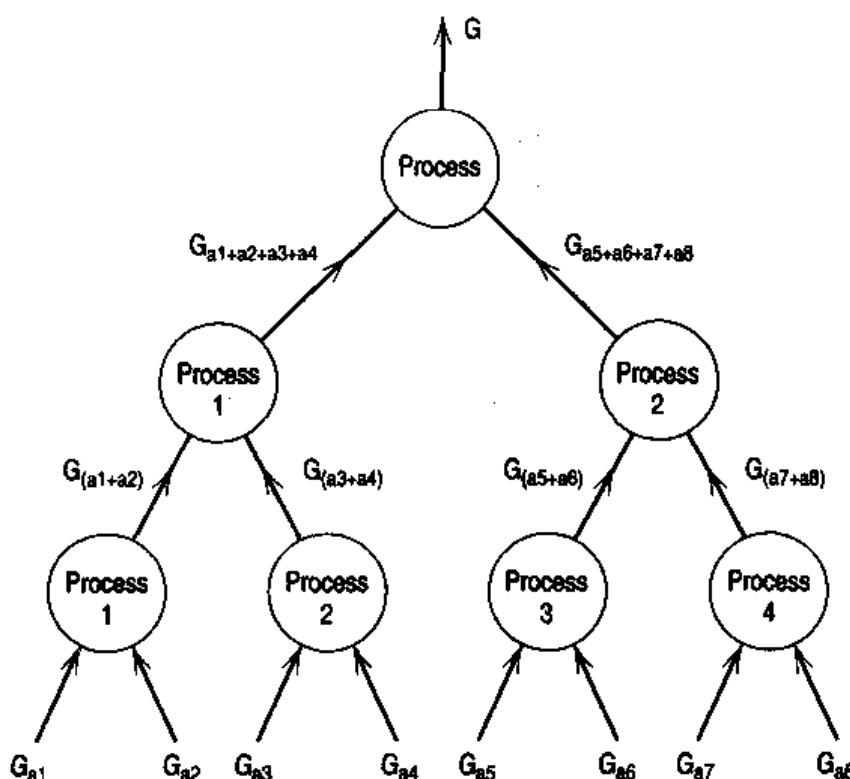
PROCESS (i, j) is a procedure which takes the UV graphs G_i and G_j , and checks whether $G_i \cup G_j = G_{i+j}$ is a UV graph. If G_{i+j} is a UV graph it returns the clique tree representation T_{i+j} of G_{i+j} .

4.5 If $G_{(2i-1)+2i}$ is not a UV graph the fact is informed and the entire algorithm is aborted and terminated because, in this case G is not a UV graph. If $G_{(2i-1)+2i}$ is a UV graph rename it as G , for the next iteration

4.6 End parallel

4.7 Next j 5: Conclude that G is a UV graph

End NC-UV



235

图7-7 $n = 8$ 时的算法过程

7.4.3 两个UV图的并

本节我们设计程序PRECESS(i, j), PRECESS(i, j)用于算法NC-UV中, 它对两个UV图 G_i 和 G_j 进行操作, 同时还需输入它的团树表示 T_i 和 T_j , 验证 $G_i \cup G_j$ 是否是UV图。如果 $G_i \cup G_j$ 是UV图, 则返回它的团树表示, 否则退出算法。我们注意到 $G_i \cup G_j$ 是 G 的某些团的并导出的图。 G_i 或 G_j 的团恰是 G 的团, 我们记

$C_i = G_i$ 的团簇;

$C_j = G_j$ 的团簇。

由于 $G_i \cap G_j$ 是 G_i 的诱导子图, 因而也是UV图。在 G_i 中, 去掉所有与 $C_i \cap C_j$ 的元素对应的顶点以及与顶点相关联的所有的边, 最后得到 T_i 的子树簇。我们试图将 T_i 的每一个这样的子树与 T_j 连接起来并构造 $G_i \cup G_j$ 的团树。首先给出算法 PRECESS(i, j)。

程序 PROCESS(i, j)

输入: 两个UV图 G_i 和 G_j , 以及它们相应的团树表示 T_i 和 T_j

输出: 1. $G_i \cup G_j$ 是否为UV图

[236]

2. 如果 $G_i \cup G_j$ 是UV图, 输出该图的团树模型

1. For every $C \in C_i \cap C_j$, do in parallel

2. Let c_1, c_2, \dots, c_s be the nodes adjacent to c in T_i . Such that $c_t \notin C_i \cap C_j$ ($i < t < s$)

3. Remove c from T_j . This removal causes a forest of s subtrees of T_j . Consider these trees as subtrees with roots c_1, c_2, \dots, c_s , respectively

4. End Parallel

5. Let R = set of subtrees (rooted) of T_j left out after steps 1 to 4

6. For every element ST of R do the operation which where going ATTACH(ST, T_i) explained in step-7

7. ATTACH(ST, T_i). Let c' be the root of ST . We try to join ST to T_i by creating an edge between c' and a node c of T_i , where $C' \cap G_i$ abort because $G_i \cap G_j$ is not a UV graph

8. End for

9. End PROCESS

我们将在下面对 ATTACH(ST, T_i) 作详细解释。首先列出一些附加结论。

改进UV图的团树 假定 G 是UV图, T 是它的团树表示, C 是 G 的一个团, v 是 C 的一个顶点。我们首先设计如下一个过程。

为了验证团树 T 能否被改进, 使得 c 是 T 中对应于 v 的一条路的端点, 改进的 T 依然是UV图 G 的团树表示。下面我们详细解释这个验证过程。

如果 c 已经是 T 的叶节点或者已经是 v 的一条路的端点, 则已经完成解释。下面我们假定 c 不是叶节点且在 T 的一条含 v 的路中, c 是它的中间顶点。考虑以 c 为根的有根树 T 。设 T 中一条含 v 的路为 $c c_{i-1} \cdots c_2 c_1 c c'_1 c'_2 \cdots c'_r$ 。

情形1

假定存在整数 i 和 j ($1 < i < l, 1 < j < r$), 使得 $(C_i \cap C) \setminus C_j \neq \emptyset, (C_j \cap C) \setminus C_i \neq \emptyset$ 。考虑 C 作为一个分离团, 团 C_i 和 C_j 位于 C 的两侧 (定理7-13), 因此我们不能重新调整 T 使得 c 是 v 的路的叶节点, 并且 T 仍然是UV图 G 的团树表示。可以证明: 如果 c 是 T 中 v 的路的端点, 要么 c_i 必须位于从 c 到 c_j 的路上, 要么 c'_j 必须位于从 c 到 c_i 的路上。

假定

$$a \in (C_i \cap C) \setminus C_j$$

$$b \in (C_j \cap C) \setminus C_i$$

由于 a 的路经过 c_i 和 c 而不经过 c'_j , 我们不能重新调整使得 c'_j 位于从 c 到 c_i 的路上。类似地,

由于 b 的路经过 c 和 c' 而不经过 c_i ，我们不能重新调整使得 c_i 位于从 c 到 c' 的路上。因此 T 不能按要求重新安排。

考虑图7-8所示的图 G 。团树 T 给定，假定我们设法改进 T ，使得 c 成为相应于顶点 b 的树的端点。我们有：

$$(C_1 \cap C) \setminus C_2 = \{f\};$$

$$(C_2 \cap C) \setminus C_1 = \{d\}.$$

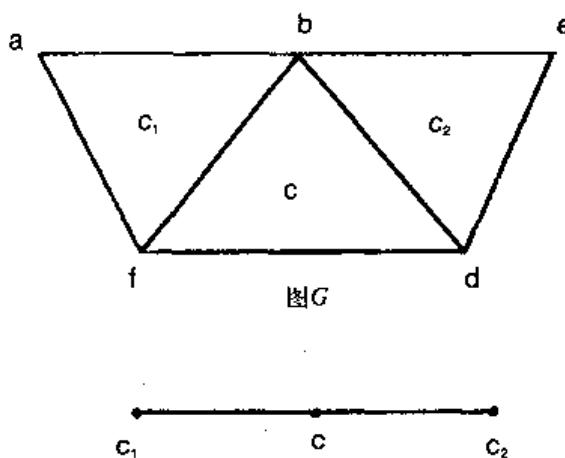


图7-8 用来描述MODIFY定义的图 G

因此，我们不能改进团树，使得 c 成为相应于顶点 b 的路的端点。

情形2

假定 $(C_i \cap C) \subseteq C_1$ 。这时，根据分离团 C 、 C_i 控制 C_1 。现在我们打算从 c 上去掉 c'_i ，并使 c'_i 成为 c_i 的孩子。首先，需要验证某些事实。假定 C_i 与 C_1 的每个孩子都不相邻，这时，很容易从 c 上去掉 c'_i 并使之成为 c_i 的孩子。假定 c_{i+1} 是 c_i 的孩子且 $C_{i+1} \cap C_1 \neq \emptyset$ ，并注意 $C_{i+1} \cap C_i \subseteq C$ ，这时，如果想使 c'_i 成为 c_i 的孩子，我们必须去掉 c_{i+1} 并使它成为其他点的孩子。由于 $C_{i+1} \cap C_i \neq \emptyset$ ， $C_{i+1} \cap C_i \subseteq C$ ，我们必须设法使 c_{i+1} 成为 c 的孩子。仅当 $C_{i+1} \cap C_i \subseteq C$ 时才能这样做。因此，对于 c 的每一个孩子 c_{i+1} ，验证是否有 $C_{i+1} \cap C_i \subseteq C$ 成立。如果成立，从 c_i 上去掉 c_{i+1} ，并使它成为 c 的孩子。去掉所有这样的 c_{i+1} 后，我们从 c 上去掉 c'_i ，并使它成为 c_i 的孩子。对于某个孩子 c_{i+1} ，如果 $C_{i+1} \cap C_1 \neq \emptyset$ ，且 $C_{i+1} \cap C_i$ 不包含在 C ，我们就不能从 c_i 上去掉 c_{i+1} ，因此不可能使 c'_i 成为 c_i 的孩子。又因为 C_i 控制 C_1 ，我们不能用这样的方式改进路，即使 c'_i 位于 c 到 c_i 的路上。这也证明了我们不能改进树使得 c 是 T 中含 v 的路的端点。

考虑图7-9所示的UV图，这个图包含6个团，分别为

$$C_0 = \{a, b, d, f\}, C_1 = \{b, d, f, g\}, C_2 = \{d, f, h\}, C_3 = \{d, f, i\}, C_4 = \{d, i, j\}, C_5 = \{f, k\}.$$

图7-9b给出了这个UV图的团树表示 T ，在团树中，相应于 d 的路是 C_2, C_1, C_0, C_3, C_4 。假定我们对改进的树 T 感兴趣，改进的树 T 满足 T 中关于 d 的路以 C_0 为端点。

注意，相对于分离团 C_0 ， C_1 ， C_2 ， C_3 位于分离团 C_0 的一侧，而 C_4 位于 C_0 的另一侧。同样， C_1 ， C_2 ， C_3 位于分离团 C_0 的一侧，而 C_4 位于 C_0 的另一侧。上述情形不满足情形1。然而我们发现， $C_3 \cap C_0 \subseteq C_2 \cap C_0$ ，即相应于分离团 C_0 ， $C_2 \supseteq C_3$ 。因此可以使 C_3 成为 C_2 的孩子。为此， C_3 必须和 C_2 的每个孩子都不相邻。这里 C_5 是 C_2 的孩子，但是， $C_5 \cap C_3 \neq \emptyset$ 。因而，如果我们想使 C_3 是 C_2 的孩子，就必须把 C_5 移到其他位置。所幸的是， $C_5 \cap C_2 \subseteq C_0$ 。因此，我们从 C_2 上去掉

掉 C_5 并使它成为 C_0 的孩子，然后从 C_0 上去掉 C_5 并使它成为 C_2 的孩子。图7-9就是改进的树。我们注意到在这个改进的树中， C_0 是 d 的路的端点，并且改进的树 T 依然是UV图 G 的团树。

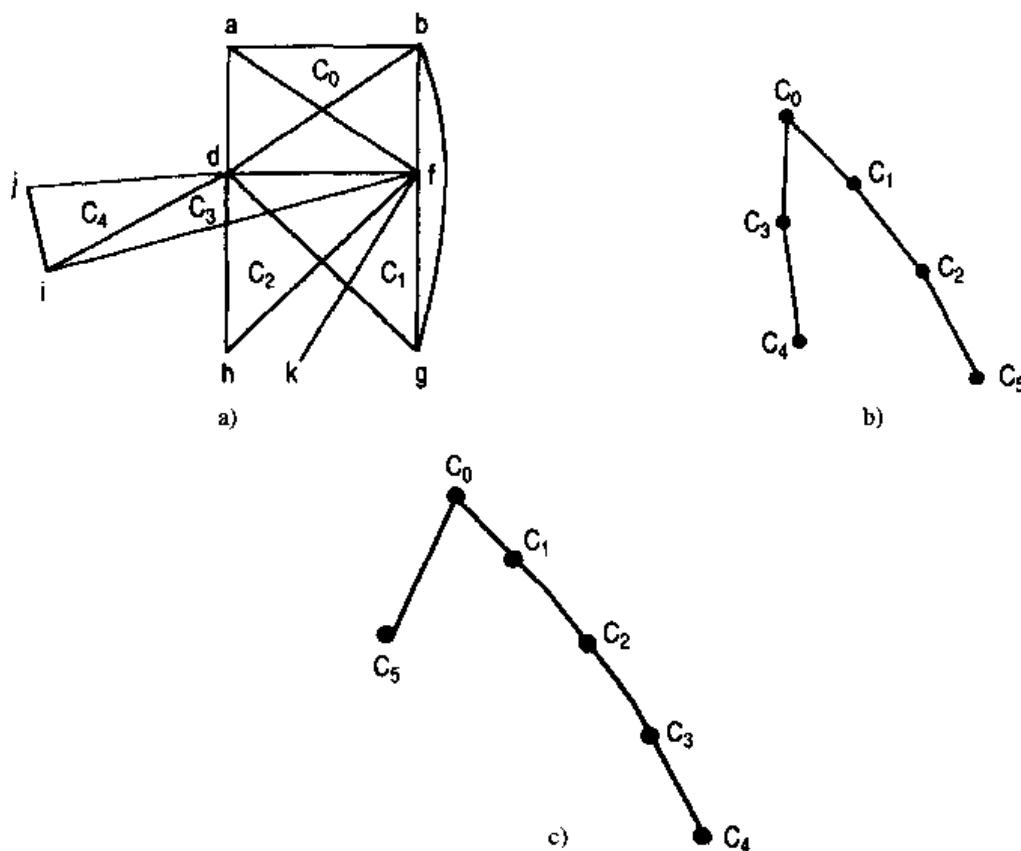


图 7-9

a) 情形2中的UV图 b) G 的团树 c) 改进的团树

情形3 关于分离团 C_i ，假定 C_i 控制 C_j 。这与情形2类似。

情形4 假定顶点 c 不满足情形1、2、3。由于不满足情形1，则对于任意的 i 和 j ($1 \leq i < l$,

240 $1 \leq j < r$)，要么 $C_i > C_j$ ，要么 $C_j > C_i$ 。然而，在 $i = 1, j = r$ 的情形下，我们有 $C_1 > C_r$ 。类似地，当 $i = 1, j = l$ 时， $C_1 > C_l$ 。在这种情形下，设法按线性顺序排列 $C_1C_2 \cdots C_lC_1C_2 \cdots C_r$ ，重新命名这些团为 $B_1B_2 \cdots B_lB_{l+1} \cdots B_{l+r}$ ，这里 $B_i = C_i$ ($1 \leq i \leq l$)， $B_{l+j} = C_j$ ($1 \leq j \leq r$)。

现在我们用下列方法对这个新的序列进行重排，满足

$$|B_1 \cap Cl| > |B_2 \cap Cl| > |B_3 \cap Cl| > \cdots > |B_{l+r} \cap Cl|$$

现在验证对于任一对满足 $1 \leq i < j \leq l+r$ 的整数 i, j ，分离团 C 是否有 $B_i > B_j$ 。如果 $B_i > B_1 > B_2 > \cdots > B_{l+r}$ ，我们重新构造 v 的路为 $cb_1b_2 \cdots b_{l+r}$ 。

由于不满足图 G 的其他顶点的路的性质，要首先改进树使得 $cb_1b_2 \cdots b_{l+r}$ 是 v 的路，做法如下：

1. 去掉边 $cc_1, c_1c_2, \dots, c_{l-1}c_l$ ；
2. 去掉边 $c'c'_1, c'_1c'_2, \dots, c'_{r-1}c'_r$ ；
3. 增加新的边 $cb_1, b_1b_2, b_2b_3, \dots, b_{l+r-1}b_{l+r}$ 。

不难验证，经过上述改进后， T 依然是一个树。我们要验证，对于UV图 G ， T 是否依然满足团树性质。为此，对于每一个 i ($1 \leq i \leq l+r$)，都要进行验证。假定 $B_{i-1} = c$ ，如果 $B_{i-1} > B_i$ ，仅当 B_i 与 b_{i-1} 的每一个孩子 b 都相离时，可以使 b_i 成为 b_{i-1} 的孩子。一旦 B_i 与 b_{i-1} 的一个孩子 b 相邻，就

必须设法去掉 b , 并使它成为其他顶点的孩子。最初 b_i 和 b_{i+1} 从 c 的两边来。 $B_i \cap B \neq \Phi$ 意味着 $C \cap B \neq \Phi$ 。因此仅使 b 成为 c 的孩子。为此, 要求 $B \cap B_{i+1} \subseteq C$ 。只有当 $B_i \cap B \neq \Phi$ 时, 检验 $B \cap B_{i+1} \subseteq C$ 是否成立。如果成立, 让 b 成为 c 的孩子; 否则, T 不能被改进成所希望的那样。基于上述讨论, 程序MODIFY可描述如下:

程序Modify (G, T, C, v)

输入:

1. UV图 G
2. G 的团树表示 T
3. G 的团 C
4. C 的顶点 v

输出: T 的改进形式使得 T 中 v 的路包含 c 作为它的端点, 而且对于UV图 G , T 依然满足团树性质。如果 T 不能改进成所要求的那样, 则返回原来的 T 并作出通知

- 1: Verify if c is already an end node of the path of v in T . If true return T , else let the path of v in T be $c_i c_{i+1} c_{i+2} \cdots c_r c_1 c c' c_2 \cdots c'_r$. Consider T as a rooted tree with root c
- 2: For every $i \in \{1, 2, 3, \dots, l\}$ and $j \in \{1, 2, \dots, r\}$ do the following in parallel

If $(C_i \cap C) \setminus C_j \neq \Phi$
and $(C_j \cap C) \setminus C_i \neq \Phi$

then return with a message that T cannot be modified

- 3: If $C_i \cap C \subseteq C_j$, then

- 3.1 For every child c_{i+1} of c_i do in parallel
- 3.2 If $C_{i+1} \cap C_j \neq \Phi$ then
- 3.3 If $(C_{i+1} \cap C_j) \setminus C \neq \Phi$ then Proceed to step 4

ELSE

- 3.4 Delete the edge $c_{i+1}c_i$ and create a new edge $c_{i+1}c$ so that c_{i+1} is a new child of c
- End if
- End if
- End parallel

- 3.5 Delete the edge $c'_j c$ and create a new edge $c'_j c_i$ so that c'_j becomes a new child of c_i

- 3.6 Return T successfully

End if

- 4: If $C_i \cap C \subseteq C_j$, then

- 4.1 For every child C_{i+1} of C_i do in parallel
- 4.2 If $C_{i+1} \cap C_j \neq \Phi$ then
- 4.3 If $(C_{i+1} \cap C_j) \setminus C \neq \Phi$ then Proceed to step 5

ELSE

- 4.4 Delete the edge $c_{i+1}c_i$ and create a new edge $c_{i+1}c$ so that c_{i+1} becomes a new child of c
- End if
- End if
- End parallel

4.5 Delete the edge $c_i c$ and create a new edge $c_i c'$, so that c_i becomes a new child of c' .

4.6 Return T successfully

End if

5:

5.1 Form duplicate copies of the cliques $C_1 C_2 \cdots C_l C_1, C_2 \cdots C_r$ with new names $B_1, B_2, \dots, B_l B_{l+1} B_{l+2} \cdots B_{l+r}$, so that $B_i = C_i (1 \leq i \leq l)$ and $B_{l+j} = C_j (1 \leq j \leq r)$. Let the corresponding nodes in T be denoted by b_1, b_2, \dots, b_{l+r} , respectively

5.2 Sort the cliques such that

$$|B_1 \cap C| > |B_2 \cap C| > \dots > |B_{l+r} \cap C|$$

5.3 For $i = 1$ to l and for $j = 1$ to r do in parallel

Remove edges c_{i-1} and $c'_{j-1} c_j$. Here C_0 means C . While removing keep log of the operation because if we are not able to modify T as desired we have to return the original T with failure message

End parallel

5.4 For $i = 1$ to $l+r$ do in parallel

Create a new edge $b_{i-1} b_i$ and reconstruct the tree T . Assume the notation b_0 for C

End parallel

5.5 For $i = 1$ to $l+r$ do in parallel

5.6 For every child B of B_{i-1} do in parallel

5.7 If $B \cap B_i \neq \Phi$ then

5.7.1 If $B \cap B_{i-1} \subseteq C$ then

5.7.2 Remove edge bb_{i-1} and create a new edge cb

5.7.2 ELSE

5.7.3 Report that T cannot be modified as desired undo the change done in steps 5.3 and

5.4 and return the original T without any modification

5.7.4 End if

End if

5.8 End parallel

5.9 End parallel

5.10 Return the modified tree T in success

现在我们重点来设计程序ATTACH(ST, T_i)，在程序PROCESS(i, j)的第7步使用了这个程序。记 G_{sr} 为由 ST 的顶点表示的团的并的导出图。

用 $W(ST)$ 表示 $G_{sr} \cap G_i$ 。在树 T_i 中， ST 的一个顶点与 $T_i \setminus ST$ 的一个顶点相邻， ST 的顶点 c 与 $T_i \setminus ST$ 的顶点 c' 相邻。因而有 $W(ST) \subseteq C$ ， $W(ST) \subseteq C'$ 。顶点 c' 也是 T_i 的顶点。设法使 ST 的顶点 c 与 T_i 的顶点 c' 相邻。然而，如果 c 是 T_i 的顶点且满足 $W(ST) \subseteq C$ ，那么我们设法使 c 与 c' 相邻，并使 ST 与 T_i 相连。

记 $N_{sr} = \{c : c \text{ 是 } T_i \text{ 的顶点且 } W(ST) \subseteq C\}$ ， N_{sr} 是 T_i 中 ST 的所有可能的邻点集合。由上面讨论知 $c \in N_{sr}$ 。因而对于每一个子树 ST ， N_{sr} 非空。通过增加一条边 $(c'c)$ ， ST 可与 T_i 的顶点 c' 相邻当且仅当 T_i 被改进，且 c 是关于 $W(ST)$ 的每一个顶点的路的端点。对于 $W(ST)$ 的每一个顶点可以并行验证这个结论。对于 $W(ST)$ 的某一个顶点，如果 T_i 不能被改进，那么顶点 c 不能与 ST 的顶

点 c 相邻。对于 N_{ST} 的每一个顶点 c ，可以并行验证是否 c 能与 c' 相邻。对于 N_{ST} 的顶点 c ，如果可能的话，可以立即通过增加一条新的边(cc')来连接 ST 与 T_i 。如果不可能使 c' 与 N_{ST} 的某个元素相邻，可以得出结论： ST 不能与 T_i 相连。因而 $G_i \cup G_j$ 不是UV图。下面这个程序目的是将 ST 连到 T_i 上。

程序Attach(ST,T_i)

1. Let G_{ST} be the graph induced by the union of cliques represented by the nodes of ST
2. $W(ST) = G_{ST} \cap G_i$
3. $N_{ST} = \Phi$
4. For every node c of T_i do in parallel
5. If $W(ST) \subseteq C$ then
 - $N_{ST} = N_{ST} \cup \{c\}$
 - End if
6. End parallel
7. For every $c \in N_{ST}$ do in parallel
8. For every $v \in W(ST)$ do in parallel
9. $MODIFY(G_i, T_i, c, v)$
 - If the procedure MODIFY returns with failure then exit
10. End parallel
11. In loop of steps 8 to 10, if the MODIFY is in failure for any one of the vertex v then
 - remove c from N_{ST}
12. End parallel
13. End parallel
14. If $N_{ST} \neq \Phi$ then choose any node c from N_{ST} and make c' adjacent to c
 - Else
 - Abort the algorithm because we cannot attach ST to T_i and Hence $G_i \cup G_j$ is not a UV graph
15. End ATTACH

243

上面这个程序将子树 ST 与 T_i 相连。在程序PROCESS(i, j)中，第6~8步调用与 R 相连的每一个子树 ST 。然而，对于 R 的所有元素不能并行执行上述操作。对于所有互不相交的子树 ST ，我们试图并行执行Attach操作。这样，第6~9步FOR循环至多执行 $\lceil \log n \rceil$ 次。

7.4.4 正确性和复杂度

本节我们证明算法的正确性并且分析时间和处理器的复杂度。

定理7-15 在NC-UV(G)算法的第4.5步中，如果 $G_{(2^{i-1}+2^j)}$ 不是UV图，那么 G 不是UV图。

证明：由于UV图是树的路簇的交图，UV图具有遗传性，即UV图的每一个诱导子图也是UV图， $G_{(2^{i-1}+2^j)}$ 也是UV图。证毕。

定理7-16 如果在NC-UV(G)算法的第4.1~4.7步的FOR-NEXT循环共执行 $\lceil \log n \rceil$ 次，并且成功地转到执行第5步而没有异常退出，那么 G 是UV图。

证明：用 G_1, G_2, \dots, G_r 来表示NC-UV(G)算法第3步中得到的 $G_{a_1}, G_{a_2}, \dots, G_{a_n}$ ，这里 G 的顶点为 $\{a_1, a_2, \dots, a_n\}$ 。在第一次执行完时，有 $[n/2]$ 个 G 的诱导子图，每一个子图都是UV图。在第4.5步的循环的最后一次迭代中，按照 $G_{(2i-1)*2}$ 的重新命名规则， G_{i+2} 就是 G ，因而 G 是UV图。

定理7-17 在程序PROCESS(i, j)的第7步中，如果我们不能将 ST 与 T_i 相连，那么 G 不是UV图。

证明：在程序ATTACH(ST, T_i)中， N_{ST} 是 ST 的所有可能的邻点集合。在程序ATTACH的第13步中仅当 $N_{ST} = \Phi$ 时，我们判断 ST 不能与 T_i 相连，这意味着不能改进 T_i 并且使得 ST 是某个顶点 c 的孩子，这里 $W(ST) \subseteq C$ 。如果 c_i 是另外一个满足 $W(ST) \subseteq C_i$ 的顶点，那么至少存在 $W(ST)$ 的一个顶点 u ，该顶点不在 C_i 中。这样，如果 ST 与 c_i 相连，那么 u 的路根本不能进入 T_i 。因此，不能将 ST 与 c_i 相连，其中 $W(ST) \subseteq C_i$ 。证毕。

定理7-18 在程序MODIFY的第2步中，如果我们返回 T 并带有信息： T 不能被改进。我们不能改进 T 使得 c 是 v 的路的端点， T 仍然是UV图 G 的团树。

证明：由于 $(C_i \cap C) \setminus C_i \neq \Phi$, $(C_j \cap C) \setminus C_j \neq \Phi$, c_i 和 c_j 位于分离团 C 的两边，因而 c 位于从 c_i 到 c_j 的路上。因此，我们不能改进 T 使得 c 是 v 的路的端点， T 仍然是UV图 G 的团树。

定理7-19 第3.5和第4.5步的操作并不改变 T ，以致于不遵从UV图 G 的团树性质。

证明：除了记号外，第3.5与4.5步是一样的。因而，这里我们仅对第3.5步讨论。对第4.5步同样成立。在第3.5步中，我们从 c 上去掉 c'_i ，并使 c'_i 与 c 相邻。这样改变并不影响 $C \setminus C_i$ 中的点。如果 $v \in C \setminus C_i$ ，那么这种改变根本不违背条件。因此可以充分验证对于 $C \setminus C_i$ 中的点，新的树是否满足UV团树条件。

第3.4步中我们去掉了所有与 C_i 有非空交的 c_i 的孩子。因此，对于 $C \setminus C_i$ 中的所有顶点的终点是 C_i 。因而，通过使 c'_i 成为 c_i 的孩子，对于 $C \setminus C_i$ 的所有顶点，路的性质满足。

定理7-20 在程序MODIFY中的第5.7.3步中，如果返回 T ，则 T 不能被改进，使得 C 是 v 的路的端点，且对于 G 中所有顶点， T 依然满足路的性质。

证明：由于经过第5.3和5.4步，对任何 i 和 j ($1 \leq i < j \leq r$)有 $B_i \subset B_j$ 。在这种情形下，我们设法去掉所有与 B_i 相连的 B_{i+1} 的孩子。由于我们不能去掉 B_{i+1} 的某些孩子 B ，在第5.7.3步中返回 T 。假定 $v \in B \cap B_i$ ，那么 $v \in B, B_i, B_{i+1}$ ，这三者不能在同一条路中。证毕。

定理7-21 算法NC-UV(G)可正确地判别UV图 G 。

证明：如果 G 不是弦图，第1步告诉我们 G 不是UV图。否则，对于 G 的顶点 a ，如果 G_a 不是区间图，那么 G 不是UV图。如果对于 G 的每一个顶点 a ， G_a 都是区间图，那么我们设法构造 G 的团树。如果第5步完成的话，那么 G 就是UV图（定理7-16）。如果不能完成第5步，在第4.5步中异常退出，这种情形下， G 不是UV图（定理7-17）。对于任一输入图，算法要么达到第5步，要么在第4.5步中异常退出。因而这个算法可正确地判定UV图。

时间和处理器复杂度 现在我们来分析该算法的时间和处理器复杂度。定理7-22给出了一个范围，这对复杂度分析是非常有用的。

定理7-22 假定 $G = (V, E)$ 是一个弦图， n_v 表示包含 v 的团的个数，那么

$$\sum n_v = \sum |c| \leq m + n$$

这里 m 和 n 分别表示 G 的边数和顶点数。

证明：定义一个矩阵 $A(v, c)$ ，这里 $v \in V, c \in C$ ，且满足

$$A(v, c) = \begin{cases} 0, v \notin c \\ 1, v \in c \end{cases}$$

对于G的任一个团c，有

$$|c| = \sum_{v \in c} A(v, c)$$

类似地，对于任一顶点 $v \in V$ ，有

$$n_v = \sum_{c \ni v} A(v, c)$$

因而有

$$\sum_{c \in C} |c| = \sum_{v \in V} \sum_{c \ni v} A(v, c) = \sum_{v \in V} n_v$$

又因为 $\sum_{c \in C} |c| \leq m + n$ ，因而定理得证。

定理7-23 程序MODIFY共需执行时间为 $O(\log n)$ ，处理器数为 $O(m+n)$ 。

[246]

证明：由于在弦图中至多有 n 个团，因而第2步所需处理器数为 $O(n)$ ，执行时间为 $O(1)$ ；第3步和第4步也需 $O(n)$ 个处理器，执行时间为 $O(1)$ ；在第5步中，第5.2步中 B_i 排序共需执行时间 $O(\log n)$ ，处理器数为 $O(n \log n)$ 。第5.6~5.8步共需时间 $O(1)$ ，处理器数 $O(n)$ 。第5.5~5.9步是一个循环，完全包含循环5.6~5.8，在5.6步中提到的 $B_{i,j}$ 至多有 $|B_{i,j}|$ 个孩子。在第5.5步中提到的 $1+r$ 是团的个数的界，因此第5.5~5.9步共需处理器数有一个界 $\sum_{c \in C} |c| \leq m + n$ （定理7-22），因而5.5~5.9步共需处理器数为 $O(m+n)$ ，执行时间为 $O(1)$ 。所以程序MODIFY共需执行时间为 $O(\log n)$ ，处理器数为 $O(m+n)$ 。证毕。

定理7-24 程序ATTACH(ST,T)共需执行时间 $O(\log n)$ ，处理器数为 $O((m+n)^2)$ 。

证明：第1、2、3步共需常数执行时间，第4~6步共需执行时间为 $O(1)$ ，处理器数为 $O(n)$ 。注意到 $|N_{st}| \leq |C|$ ， $|W(ST)| \leq |V|$ 。因此，根据定理7-22，每一次并行执行MODIFY共需 $O(m+n)$ 个处理器。根据定理7-23，第7~13步共需执行时间 $O(\log n)$ ，处理器数为 $O((m+n)^2)$ 。所以程序ATTACH共需执行时间 $O(\log n)$ ，处理器数为 $O((m+n)^2)$ 。

定理7-25 程序PROCESS(i,j)共需执行时间 $O(\log^2 n)$ ，处理器数为 $O(m^2 n)$ 。

证明：第1~4步共需执行时间 $O(1)$ ，处理器数为 $O(n)$ 。我们已经证明 R 至多含有 $\lceil \log n \rceil$ 个子树，每一个子树互不相交。因此，根据定理7-24，PROCESS(i,j)共需执行时间 $O(\log^2 n)$ ，处理器数为 $O(m^2 n)$ 。

定理7-26 判定UV图的问题是NC的。

证明：算法NC-UV(G)第1步利用Klein算法，Klein算法是NC的；第2步利用Klein的区间图判别算法；第3步所用的执行时间为 $O(\log n)$ ，处理器数为 $O(n \log n)$ ；第4步中，第4.3~4.6步共需执行时间 $O(\log^2 n)$ ，处理器数为 $O(m^2 n^2)$ （定理7-23，定理7-24和定理7-25）。第4.1~4.7步For循环需要执行 $\lceil \log n \rceil$ 次。因此该算法总的复杂度是执行时间为 $O(\log^3 n)$ ，处理器数为 $O(m^3)$ ，且执行CREW PRAM模型。

参考文献

Chandrasekharan, N. (1985) New Characterizations and Algorithmic Studies On Chordal Graphs and k -Trees, (M.Sc.(Engg.) Thesis), School of Automation, Indian Institute of Sciences, Bangalore, India.

[247]

Chandrasekharan, N. and Iyengar, S. S. (1988) NC Algorithms for Recognizing Chordal Graphs and k Trees, *IEEE Transactions on Computers*, 37(10).

- Chandrasekharan, R. and Tamir, A. (1982) Polynomially Bounded Algorithms for Locating p -Centres on a Tree, *Math. Programming*, **22**, 304–315.
- Naor, J., Naor, M. and Scjaffer, A. (1987) Fast Parallel Algorithm for Chordal Graphs, in *Proc. Symp. Theory Comput.*, New York.
- Proskurowski, (1980) k -trees: Representations and Distances, Tech. Rep., CIS-TR-80-5, University of Oregon.
- Xavier, C., (1995) Sequential and Parallel Algorithms for Some Graph Theoretic Problems, Ph.D. Thesis, Madurai-Kamaraj University, Madurai, India.

第三部分 数组处理算法

第8章 搜索与合并

8.1 串行搜索

假定 $A = (a_1, a_2, a_3, \dots, a_n)$ 是一个数据数组且满足 $a_1 < a_2 < \dots < a_n$ 。给定一个元素 x ，我们感兴趣的是求出数组下标 k 使得 $a_k \leq x < a_{k+1}$ 。在串行计算中，既可以对每一个 k 验证 a_k 和 a_{k+1} 的值，也可以使用二分搜索技术。算法如下：

算法 Sequential Search

输入：1. 数据数组 $a_1 < a_2 < \dots < a_n$
2. 元素 x
输出：下标 k 且 $a_k \leq x < a_{k+1}$
1. Set $a_0 = -\infty$ and $a_{n+1} = +\infty$
2. For $k = 0$ to n do
3. If $a_k \leq x$ and $x < a_{k+1}$ then return (k)
 else continue
4. Next k
5. END

上面的串行搜索方法对数组的数据逐个进行比较。因此串行执行时间为 $O(n)$ 。下面的二分搜索法可以在 $O(\log n)$ 的时间里解决搜索问题。

算法 Binary Search

输入：1. 数据数组 $a_1 < a_2 < a_3 < \dots < a_n$
2. 元素 x

输出：下标 k 且 $a_k \leq x < a_{k+1}$

1. $a_0 = -\infty, a_{n+1} = +\infty$
left = 1, right = n
2. While left \leq right repeat
 mid = [left + right]/2

Case:

$x < a_{\text{mid}}$: right = mid - 1

$x = a_{\text{mid}}$: return (mid)

$x > a_{\text{mid}}$: left = mid + 1

End Case

3. End While

4. return (right)

5. END

为了演示二分搜索的过程，我们以数组 $A = (20, 24, 25, 29, 32, 35, 39, 85)$ 为例，设 $x = 22$ 。

初始化：left = 1, right = 8 且 mid = 4；

比较 $x = 22$ 和 $a_{mid} = 29$ ，

有 $x < a_{mid}$ 。因此，令 $right = mid - 1 = 3$ 。

迭代1：left = 1, right = 3 且 mid = 2；

比较 $x = 22$ 和 $a_{mid} = 24$ ，

有 $x < a_{mid}$ 。因此，令 $right = mid - 1 = 1$ 。

迭代2：left = 1, right = 1 且 mid = 1；

比较 $x = 22$ 和 $a_{mid} = 20$ ，

有 $x > a_{mid}$ 。因此，令 $left = mid + 1 = 2$ 。

现在有 $left = 2, right = 1$ 。

While 循环 $left < right$ 不满足条件为真，

因此 While 循环结束，返回 $right = 1$ 。

对同一个数组，我们在下表中用另外两个 x 的值进一步说明算法的执行过程。

$x = 20$			
迭代	左	右	中
0	1	8	4
1	1	3	2
2	1	1	1
返回 1			

$x = 100$			
迭代	左	右	中
0	1	8	4
1	5	8	6
2	7	8	7
3	8	8	8
4	9	8	
返回 8			

8.2 CREW PRAM模型下的并行搜索

在CREW PRAM模型里面， $a_k \leq x < a_{k+1}$ 的验证操作可以对所有的 k 值同时进行。即处理器 P_0 进行 $a_0 \leq x < a_1$ 的操作；处理器 P_1 进行 $a_1 \leq x < a_2$ 的操作；处理器 P_2 进行 $a_2 \leq x < a_3$ 的操作；以此类推。

验证 $a_i \leq x < a_{i+1}$ 的处理器 P_i 把结果 i 写入全局变量 RESULT 中。算法如下：

输入： 1. 数组 $A = (a_1, a_2, a_3, \dots, a_n)$, $a_1 < a_2 < a_3 < \dots < a_n$
2. 值 x

输出： $Result = k$, $a_k \leq x < a_{k+1}$

0: $a_0 = -\infty, a_{n+1} = +\infty$

1: For $k = 0$ to n do in parallel

2: If $a_k \leq x$ and $x < a_{k+1}$ then

set RESULT = k

endif

3: End Parallel

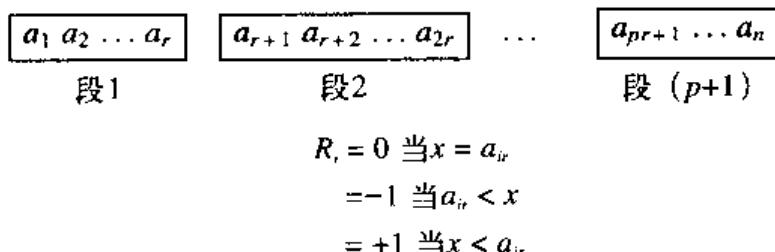
4: END

复杂度分析 在上面的算法中，并行循环需要有 $n+1$ 个处理器同时执行。第2步的时间复杂度是 $O(1)$ 。因此该算法需 $O(n)$ 个处理器，可以在 $O(1)$ 的时间内完成。由于数组 A 满足条件 $a_1 < a_2 < a_3 < \dots < a_n$ ，只有一个处理器能够在第2步成功设置RESULT的值，所以要允许绝对写操作。而处理器 P_i 和处理器 P_{i-1} 都要用到 a_i ，并发读操作是必要的。因而CREW PRAM计算模型可实现该算法。

8.3 更多数据的并行搜索

当数组的长度比处理器的数目多的时候，上述算法不能在 $O(1)$ 时间内完成。考虑数组 $A = (a_1, a_2, a_3, \dots, a_n)$ 其中 $a_1 < a_2 < a_3 < \dots < a_n$ 。给定一个值 x ，处理器数目为 p 。我们下面对二分搜索法进行扩展。

在二分搜索中，整个数组被分成两个子数组，数组的中间元素与 x 进行比较。在本节我们先把数组分成 $p+1$ 个几乎大小相同的子数组并比较由此得到的 p 个内值。为了方便起见，假定 $n = (p+1)r$ 。因此，当把数组分成 $p+1$ 个子段时，每一段将有 r 个元素。



如果 $R_i = 0$ ，那么返回值 $i * r$ 。如果对所有的 i 有 $R_i \neq 0$ ，那么选择满足 $R_{k-1} = -1$ 且 $R_k = +1$ 条件的第 k 段，则 x 将落在第 k 段内。在第 k 段重复上述过程。该算法正式给出如下：

算法 Parallel Search

输入： 1. 数组 $A = (a_1, a_2, a_3, \dots, a_n)$ 。
2. 值 x

输出：下标 k 且 $a_k \leq x < a_{k+1}$

1. p is the number of processor variable. If $n < p$ then use the algorithm Search-CREW algorithm and solve $O(1)$ time
2. If $n > p$ then without loss of generality assume that $n = r(p+1)$ for an integer value r
3. For $i = 1$ to p do in parallel
 - Case:
 - $a_i = x: R_i = 0$
 - $a_i < x: R_i = -1$
 - $a_i > x: R_i = +1$
 - End case
4. End Parallel
5. For $i = 1$ to p do in parallel
 - If $R_i = 0$ then
 - return ($i * r$)
 - End if

254

6. End Parallel
7. Set $R_0 = -1$ and $R_{p+1} = 1$
8. For $k = 1$ to $p + 1$ do in parallel
9. If $R_k = +1$ and $R_{k-1} = -1$ then choose
the k th segment $(a_{ik-1}, a_{ik-1}, \dots, a_i)$ and do the search operation recursively on this segment
- Endif
10. End Parallel
11. END

在上述算法中，每执行一个递归步，问题的规模就以 $p+1$ 的因子减小。

8.4 无序数组搜索

设 $A = (a_1, a_2, a_3, \dots, a_n)$ 是一个没有排序的数组。给定一个数 x ，我们的问题是求出满足 $a_k = x$ 的数组的下标 k 。如果满足条件的 k 不存在，那么返回的结果是 $n+1$ 。以下是解决此问题的串行算法。

算法 Unsorted Search

输入： 1. 数组 $A = (a_1, a_2, a_3, \dots, a_n)$
2. 值 x

输出： 下标 k 且 $a_k = x$ 。如果没有这样的 k 存在则返回 $n+1$

1. $k = 0$
2. For $k = 1$ to n do
3. If $a_k = x$ then exit for loop
4. Next k
5. return (k)
6. END

在串行计算机上，该算法可以在 $O(n)$ 的时间内完成。它可以直接并行化。第 2 ~ 4 步的 for-next 循环可以转换成并行循环。如果使用 $O(n)$ 个处理器，这样得到的并行算法可以在 $O(1)$ 的时间内完成。

8.5 秩合并

如果 $A = (a_1, a_2, a_3, \dots, a_m)$ 和 $B = (b_1, b_2, b_3, \dots, b_n)$ 是两个有序的数组，合并数组 A 和 B 意味着形成一个新的包含 A 和 B 的 $(m+n)$ 个元素的有序数组。例如， $A = (2, 4, 11, 12, 14, 35, 95, 99)$ ， $B = (6, 7, 9, 25, 26, 31, 42, 85, 87, 102, 105)$ ，合并两个数组后得到的新数组 $C = (2, 4, 6, 7, 9, 11, 12, 14, 25, 26, 31, 35, 42, 85, 87, 95, 99, 102, 105)$ 。

串行合并算法通过遍历两个数组，把数组元素存到 C 中。开始的时候，我们应该让指针 i 指向数组 A 的第一个元素，指针 j 指向数组 B 的第一个元素。即 $i = 1$ 且 $j = 1$ 。将要存入元素的数组 C 的下标记为 k 。比较 a_i 和 b_j ，其中小的一个存入 c_k 。相应的指针指向下一个元素。该串行算法如下所示：

串行算法Merge

输入：大小分别为 m 和 n 的有序数组 A 和 B

输出：合并的有序数组 $C = (c_1, c_2, c_3, \dots, c_{m+n})$

1. Set $a_{m+1} = b_{n+1} = +\infty$

2. Set $i = 1, j = 1, k = 1$

3. While $k < m+n$ do

4. If $a_i < b_j$ then

$c_k = a_i$ and $i = i+1$

else

$c_k = b_j$ and $j = j+1$

endif

5. $k = k + 1$

6. End while

7. END

上面的串行算法可以在 $O(m+n)$ 的时间内完成。它是内在串行的。为了将其并行化，我们先介绍一些符号。设 $A = (a_1, a_2, a_3, \dots, a_m)$ 和 $B = (b_1, b_2, b_3, \dots, b_n)$ 是两个数组， x 表示一个数。我们定义 $\text{rank}(x:A)$ 为数组 A 中小于等于 x 的元素个数。设 $A = (2, 4, 11, 12, 14, 35, 95, 99)$, $B = (6, 7, 9, 25, 26, 31, 42, 85, 87, 102, 105)$ ，那么 $\text{rank}(6:A) = 2$, $\text{rank}(7:A) = 2$, $\text{rank}(9:A) = 2$, $\text{rank}(25:A) = 5$ 。

我们进一步定义 $\text{rank}(B:A)$ 为数组 $(r_1, r_2, r_3, \dots, r_m)$ ，其中 $r_i = \text{rank}(b_i:A)$ 。对上面给出的数组 A 和 B :

$$\text{rank}(B:A) = (2, 2, 2, 5, 5, 6, 6, 6, 8, 8)$$

$$\text{rank}(B:B) = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)$$

$$\text{rank}(A:B) = (0, 0, 3, 3, 3, 6, 9, 9)$$

$$\text{rank}(A:A) = (1, 2, 3, 4, 5, 6, 7, 8)$$

$\text{rank}(x:A \cup B)$ 是 $A \cup B$ 中小于等于 x 的元素个数。因此当 A 和 B 交集为空集时， $\text{rank}(x:A \cup B) = \text{rank}(x:A) + \text{rank}(x:B)$ 。

$$\begin{aligned}\text{rank}(A:A \cup B) &= (1, 2, 3, 4, 5, 6, 7, 8) + (0, 0, 3, 3, 3, 6, 9, 9) \\ &= (1, 2, 6, 7, 8, 12, 16, 17)\end{aligned}$$

$$\begin{aligned}\text{rank}(B:A \cup B) &= \text{rank}(B:A) + \text{rank}(B:B) \\ &= (2, 2, 2, 5, 5, 6, 6, 6, 8, 8) + (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11) \\ &= (3, 4, 5, 9, 10, 11, 13, 14, 15, 18, 19)\end{aligned}$$

256

我们有

$$\text{rank}(A:A \cup B) = (1, 2, 6, 7, 8, 12, 16, 17)$$

$$\text{rank}(B:A \cup B) = (3, 4, 5, 9, 10, 11, 13, 14, 15, 18, 19)$$

从上面的两个数组中，可以发现每一个数组元素在数组 C 中的最后位置。由 $\text{rank}(A:A \cup B)$ ，可以看出 $\text{rank}(a_5:A \cup B) = 8$ 。这表示在数组 $A \cup B$ 中有8个元素小于或等于 a_5 。所以， a_5 一定位于合并后的数组 C 中的第8个位置。相似地，由于 $\text{rank}(a_6:A \cup B) = 12$ ， a_6 会赋值给 C_{12} 。如果 RA

$= \text{rank}(A:A \cup B) = (1, 2, 6, 7, 8, 12, 16, 17)$, 那么有 $C(RA_i) = A_i$ 。举例来说:

$$\begin{aligned} C(1) &= a_1 = 2, & C(2) &= a_2 = 4, & C(6) &= a_3 = 11 \\ C(7) &= a_4 = 12, & C(8) &= a_5 = 14, & C(12) &= a_6 = 35 \\ C(16) &= a_7 = 95, & C(17) &= a_8 = 99 \end{aligned}$$

如果 $RB = \text{rank}(B:A \cup B)$, 那么有 $C(RB_i) = B_i$ 。

$RB = \text{rank}(B:A \cup B) = (3, 4, 5, 9, 10, 11, 13, 14, 15, 18, 19)$ 。因而有

$$\begin{aligned} C(3) &= b_1 = 6, & C(4) &= b_2 = 7, & C(5) &= b_3 = 9 \\ C(9) &= b_4 = 25, & C(10) &= b_5 = 26, & C(11) &= b_6 = 31 \\ C(13) &= b_7 = 42, & C(14) &= b_8 = 85, & C(15) &= b_9 = 87 \\ C(18) &= b_{10} = 102, & C(19) &= b_{11} = 105 \end{aligned}$$

该算法正式表述如下:

算法Merging and Ranking

输入: 1. 数组 $A = (a_1, a_2, \dots, a_m)$, $a_1 < a_2 < a_3 < \dots < a_m$

2. 数组 $B = (b_1, b_2, \dots, b_n)$, $b_1 < b_2 < b_3 < \dots < b_n$

输出: 合并的数组 $C = (C(1), C(2), C(3), \dots, C(m+n))$, $C(1) < C(2) < C(3) \dots < C(m+n)$

1. For $i \in \{1, 2, \dots, m\}$ and $j \in \{1, 2, \dots, n\}$ do in parallel
2. Find rank ($a_i:A$) and Find rank ($b_j:B$)
3. Find rank ($a_i:B$) and Find rank ($b_j:A$)
4. End parallel
5. Denote rank ($A:B$) and rank ($B:A$) as explained earlier
6. $RA = \text{rank}(A:A) + \text{rank}(A:B)$
7. $RB = \text{rank}(B:A) + \text{rank}(B:B)$
8. For $i = 1$ to m do in parallel
9. $C(RA_i) = A_i$
10. End parallel
11. For $i = 1$ to n do in parallel
12. $C(RB_i) = B_i$
13. End parallel
14. END

复杂度分析 由于 A 和 B 是有序数组, 我们可以用二分搜索法利用一个处理器在 $O(\log m)$ 时间内完成求 $\text{rank}(x:A)$ 。相似地, 求 $\text{rank}(x:B)$ 可以在 $O(\log n)$ 时间内完成。所以, 第1~4步需用 $O(m+n)$ 个处理器, 在 $O(\log n)$ 时间内完成 (假定 $n \geq m$)。第8~10步和第11~13步用 $O(n)$ 个处理器可以在 $O(1)$ 时间内完成。所以该算法需用 $O(m+n)$ 个处理器, 在 $O(\log n)$ 时间内完成。第2~3步涉及并发读操作, 而绝对写操作也是必要的。因此, 该算法可以在CREW PRAM模型中实现。

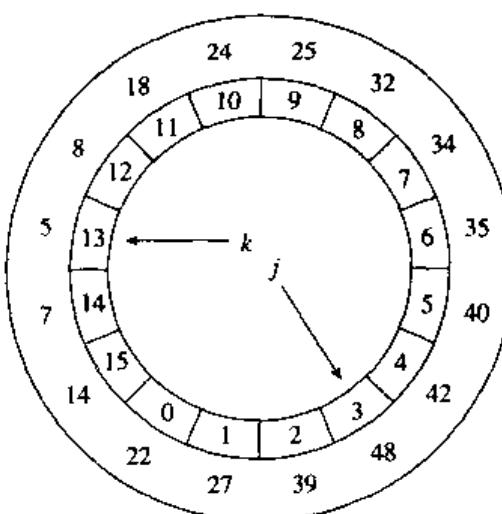
8.6 双调合并

如果存在两个整数 j 和 k 使得 $(a_{j+1}, a_{j+2}, \dots, a_t)$ 和 $(a_{(k+1)\text{mod}(n)}, a_{(k+2)\text{mod}(n)}, \dots, a_j)$ 是两个单调序列, 一个单

调上升，另一个单调下降，那么我们称数组 $A = (a_0, a_1, a_2, \dots, a_{n-1})$ 为双调数组或双调序列。例如，序列 $A = (22, 27, 39, 48, 42, 40, 35, 34, 32, 25, 24, 18, 8, 5, 7, 14)$ 是一个双调序列。为了便于理解，此序列如下表所示：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$A(i)$	22	27	39	48	42	40	35	34	32	25	24	18	8	5	7	14

如果我们把这些数沿着一个圆圈的周边排列，那么这些数可以分成两部分：一部分单调上升，另一部分单调下降。258



设 $A(0:2n-1)$ 为一双调序列。我们定义：

$$L_i = \min\{a_i, a_{i+n}\} \quad 0 \leq i < n$$

$$R_i = \max\{a_i, a_{i+n}\} \quad 0 \leq i < n$$

则有如下的结论：

1. $L_0, L_1, L_2, \dots, L_{n-1}$ 是一个双调序列；
2. $R_0, R_1, R_2, \dots, R_{n-1}$ 是一个双调序列；
3. 每一个 L_i 都小于或等于 R_i 。

为了说明这一点，考察本节所用的数组 A 。

i	a_i	a_i	i
0	25	32	8
1	27	25	9
2	39	24	10
3	48	18	11
4	42	8	12
5	40	5	13
6	35	7	14
7	34	14	15

259

i	L_i	R_i	i
0	25	32	8
1	25	27	9
2	24	39	10
3	18	48	11
4	8	42	12
5	5	40	13
6	7	35	14
7	14	34	15

上面的结果被用于合并两个有序数组。我们可以进行如下操作：

1. 设 $A(0:n-1)$ 和 $B(0:n-1)$ 是两个有序数组。把数组 B 反转，即 $(b_0, b_1, b_2, \dots, b_{n-1}) = (b_{n-1}, b_{n-2}, \dots, b_2, b_1, b_0)$ 。现在 $A(0:n-1)$ 是单调上升序列， $B(0:n-1)$ 是单调下降序列。那么我们可以用数组 B 扩展数组 A ，即 $A(n:2n-1) = B(0:n-1)$ 。由此， $A(0:2n-1)$ 是双调序列。

2. 对 $i = 0, 1, 2, \dots, (n-1)$ ，进行如下的操作

$$a_i = \text{Min}\{a_i, a_{i+n}\}$$

$$a_{i+n} = \text{Max}\{a_i, a_{i+n}\}$$

该操作可以通过下面的“if”语句完成：

if $a_i > a_{i+n}$ then interchange a_i and a_{i+n}

通过这些操作，我们得到：

$$(L_0, L_1, L_2, \dots, L_{n-1}) = (a_0, a_1, a_2, \dots, a_{n-1})$$

$$(R_0, R_1, R_2, \dots, R_{n-1}) = (a_n, a_{n+1}, a_{n+2}, \dots, a_{2n-1})$$

因而有如下的结论：

1. $A(0:n-1)$ 为一双调序列；

2. $A(n:2n-1)$ 为一双调序列；

3. 对 $0 < i < n$ 和 $n < j < 2n-1$ ，每一个 a_i 都小于等于 a_j 。即 $A(0:n-1)$ 的每一个元素都小于等于 $A(n:2n-1)$ 。

基于上面的结论，子数组 $A(0:n-1)$ 和 $A(n:2n-1)$ 可以分别处理。这样， $A(0:2n-1)$ 将给出最后的合并数组。下面正式给出算法描述。

算法 Bitonic-Merge

输入：有序序列 $A(0:n-1)$ 和 $B(0:n-1)$

输出：合并的序列 $A(0:2n-1)$

1. Copy $A(n:2n-1) = B(0:n-1)$

2. For $i = 0$ to $n-1$ do in parallel

3. If $a_i > a_{i+n}$ then

Interchange a_i and a_{i+n}

Endif

4. End parallel

5. Do the above operations recursively on $A(0:n-1)$ and $A(n:2n-1)$ separately in parallel
and make them sorted sequence

6. End

260

复杂度分析 在第2~4步， $O(n)$ 个处理器可以在 $O(1)$ 时间内完成。每一次迭代，数组的大小减半，所以一共有 $O(\log n)$ 个递归步。因此，在EREW PRAM模型中，此算法可以在 $O(n)$ 个处理器上用 $O(\log n)$ 的时间完成。

参考文献

- Cole, R., Parallel Merge Sort, *SIAM J. Computing*, 17(4):770–785, 1988.
- Knuth, D., *Sorting and Searching*, Addison-Wesley, Reading, MA 1973.
- Kruskal, C., Searching, Merging and Sorting in Parallel Computation, *IEEE Transactions on Computers*, C-32(10):942–946, 1983.
- Shiloach, Y., and U. Vishkin, Finding the maximum merging, and sorting in a Parallel Computation model, *Journal of Algorithms*, 2(1):85–102, 1981.

第9章 排序算法

排序一般是按某种顺序把事物进行排列。把文件中的记录 R_1, R_2, \dots, R_n 进行排序就是为符号 $1, 2, \dots, n$ 确定一个置换 π ，使得对 $i = 1, 2, \dots, n-1$ 有 $k_{\pi(i)} < k_{\pi(i+1)}$ 成立，其中的 k_i 是记录 R_i 的键值。如果 A 是一个数组，那么 $A_{\pi(1)} < A_{\pi(2)} < \dots < A_{\pi(n)}$ 。除了一般的商业应用，排序还有很多应用领域，下面我们给出一些例子：

- 假定有10 000个随机样本且其中很多是相同的。我们的目的是要将其重新排列，使得所有相同的值相邻。此问题称为“归属”(togetherness)问题。
- 搜寻某个特别的记录只有在文件中记录排序的情况下才比较容易。
- 计算机使用中的“文件编辑”操作。
- 计算机制造商估计超过25%的计算机运行时间被用来进行排序运算。

排序算法的复杂度 排序算法的复杂度是参加排序的记录数目 n 的函数。下面给出的是在排序过程中的基本操作：

1. 两个键值的比较；
2. 记录的交换；
3. 在一个临时位置对记录进行赋值。

正常情况下，由于其他操作的次数几乎是比较次数的常数倍，复杂度函数只计算比较的次数。在学习并行排序算法之前，让我们先回顾几个串行排序算法。

262 9.1 串行排序算法

到现在为止，研究者设计了多个串行排序算法，其中冒泡排序是最流行的。

9.1.1 冒泡排序

冒泡排序是常用的排序方法。想对数 x_1, x_2, \dots, x_n 进行排序，如果 $x_i > x_{i+1}$ ，则交换 x_i 和 x_{i+1} ，其中 $i = 1, 2, \dots, k$ ， $k = n, n-1, \dots, 2$ 。因而比较操作的次数为：

$$1+2+3+\dots+(n-1)=n(n-1)/2=O(n^2)$$

在此算法中，比较操作的次数和数据集合无关。冒泡排序是最简单的排序算法，而且可以并行化。考虑有8个数的冒泡排序：

$A_0 \ A_1 \ A_2 \ A_3 \ A_4 \ A_5 \ A_6 \ A_7$

如果用冒泡排序，则先比较 A_0 和 A_1 ，然后是 A_2 和 A_3 ， A_4 和 A_5 ，等等。如果用4个处理器，那么每个处理器可以分到两个数：

$P_0: A_0 \ \& \ A_1$	$P_1: A_2 \ \& \ A_3$	$P_2: A_4 \ \& \ A_5$	$P_3: A_6 \ \& \ A_7$
-----------------------	-----------------------	-----------------------	-----------------------

每一个处理器比较自己的两个数，如果需要就进行交换。即处理器 P 进行如下操作：

如果 $A_{2i} > A_{2i+1}$ ，那么交换 A_{2i} 和 A_{2i+1} 。对 $i = 0, 1, 2, 3$ ，上述操作在4个处理器中并行进行（我

们称为第0步)。

在下面一步(第1步),用3个处理器且数的分配如下:

$$\begin{array}{ccc} A_1 & \& A_2 & A_3 & \& A_4 & A_5 & \& A_6 \\ P_0 & & P_1 & & & P_2 \end{array}$$

这样,每个处理器比较 A_{2i+1} 和 A_{2i+2} ,如果 $A_{2i+1} > A_{2i+2}$,那么交换。下一步(第2步),数的分配又一次出现和第0步相同的情况;而第3步和第1步类似。

此并行算法可以正式表述如下:

算法Bubble-Sort

输入: $A(0:n-1)$

输出: 有序数组 $A(0:n-1)$

263

1. For $k = 0$ to $n-2$
2. If k is even then
 - (a) for $i = 0$ to $(n/2)-1$ do in parallel
 - (b) If $A_{2i} > A_{2i+1}$ then interchange them
 - (c) End parallel
- Else
 - (d) For $i = 0$ to $(n/2)-2$ do in parallel
 - (e) If $A_{2i+1} > A_{2i+2}$ then interchange them
 - (f) End parallel
- Endif
3. Next k

复杂度分析 第1~3步循环体的执行次数是 $n-1$ 次,因此执行时间为 $O(n)$ 。第2步中,奇数步需要 $(n/2)-2$ 个处理器而偶数步需要 $(n/2)-1$ 个,所以需要 $O(n)$ 个处理器。很清楚可以看出,EREW PRAM计算模型是可以满足要求的。

9.1.2 插入排序

如果前面几个记录已经是排序的形式,那么未排序的记录可以插入已排序的记录中的适当位置。此方法称作插入排序。给定数 A_1, A_2, \dots, A_n ,其中

$$A_1 \leq A_2 \leq \cdots \leq A_{j-1}, 1 \leq j \leq n$$

此方法建议在 A_i 和 A_{j-1} 之间的合适位置插入 A_j ,由此我们得到 $A_1 \leq A_2 \leq \cdots \leq A_j$ 。上面的假定在给定数列中对 $j=2$ 是成立的,对 $j=2, 3, \dots, n$,重复上面的步骤,就可以完成排序。

下面是插入排序的串行算法:

算法Insertion Sort

BEGIN

1. For $j = 2, 3, 4, \dots, n$
2. $i = j-1, T = A_j$
3. If $T > A_i$ go to step 6
4. $A_{i+1} = A_i; i = i-1$

5. If $i > 0$ go to step 3

6. $A_{i+h} = T$

7. Next j

END

总的计算时间是 $O(n^2)$ ，因此这并不是一个好的算法。

对直接插入排序法的改进 直接插入涉及两个基本操作：(i) 扫描有序数列 A_1, A_2, \dots, A_{j-1} ，求出小于等于 A_j 的最大数；(ii) 在特定的位置插入 A_j 。在最坏的情况下， A_i 和 A_j 之间的扫描需要 j 个比较。如果采用二分搜索技术，可以减少到 $\log j$ 个比较。但是如果想减少插入的时间，给定数列的数据结构必须改变。如果采用列表数据结构，就不能用二分搜索技术扫描有序列表，因此不能提高总的计算时间。

9.1.3 Shell递减步长排序

如果每一对数都进行比较并排序，那么总的排序时间不可能比 $O(n^2)$ 少。因此考虑其他的技术是必要的。Shell于1959年提出了一个新的排序技术，被称为递减步长排序法。

先给定一个整数步长 h ($0 < h < n$)，然后把给定的数分成 h 个集合，如下所示：

$$\text{集合 } 1 = \{A_1, A_{h+1}, A_{2h+1}, \dots\}$$

$$\text{集合 } 2 = \{A_2, A_{h+2}, A_{2h+2}, \dots\}$$

...

...

$$\text{集合 } h = \{A_h, A_{2h}, A_{3h}, \dots\}$$

每一组集合用直接插入排序法进行排序，然后把步长 h 减小，重复前面的步骤。对逐渐减小的步长 $h_t, h_{t-1}, h_{t-2}, \dots, h_3, h_2, h_1$ 进行上面的操作，最后完成排序。这意味着 $n < h_t < h_{t-1} < h_{t-2} \dots < h_3 < h_2 < h_1 = 1$ 。此算法如下所示：

算法 Shellsort

1. For $s = t, t-1, \dots, 3, 2, 1$ do
2. $h = h_s$
3. For $j = h + 1$ to n do
4. $i = j-h; T = A_j$
5. If $A_i < T$ then go to step 8
6. $A_{i+h} = A_i; i = i-h$
7. If $i > 0$ go to step 5
8. $A_{i+h} = T$
9. Next j
10. Next s

复杂度分析 下面是此算法的一些参数：

1. t ，步长个数；

2. 步长 $h_t, h_{t-1}, h_{t-2}, \dots, h_3, h_2, h_1 = 1$ 。

对计算时间的分析会引出一些很吸引人但目前尚未解决的数学问题；对很大的 N ，没有人

能确定可能最好的步长序列。

Hunt仅仅考虑了两个步长，结果表明计算时间为 $O(n^{5/3})$ 。对直接插入排序的 $O(n^2)$ 计算时间而言， $O(n^{5/3})$ 已经是很大的改进。由于对比较小的 N （如16），直接插入排序是最好的方法，我们选定 $h = N/16$ 并且对不同的 N 运行Shell的算法。我们记录了计算时间并分析了所得到的变化图。

如果步长 $h_s, h_{s+1}, h_{s+2}, \dots, h_3, h_2, h_1$ 对 $s = 2, 3, \dots, t$ 满足 h_{s+1} 可整除 h_s ，我们说步长序列满足整除条件。计算时间不可能小于 $O(n^{3/2})$ 。Papernov和Stasevich的研究表明，如果对 $s = 2, 3, \dots, t$ ，整除条件不能满足，那么Shell的算法会更有效。步长 $h_s = 2^{s-1}$ ($s = 1, 2, 3, \dots, t$) 满足整除条件，而 $h_s = 2^s - 1$ ($s = 1, 2, 3, \dots, t$) 则不满足。

Pratt在1969年发现了对这一问题的改进。如果所选择的步长都是 $2p3q$ 且比 N 小的形式，那么计算时间为 $O(N(\log N)^2)$ 。只有在 N 很大的情况下才能改进效率。

1971年，斯坦福大学的Peterson 和Russell对选择Shell算法的步长进行了大量的实验。他们采用了很多步长经验值。经验值当然不能穷尽所有的可能性，他们没有发现选择最好的步长序列的有力论据。一般下面的步长选择方法被认为能够使Shell算法更有效率。设 $h_1 = 1$ ， $h_{s+1} = 3h_s + 1$ ，直到 $h_s + 2 > n$ 时停止。

9.1.4 堆排序

本节我们学习一种新的能够允许向一个集合插入数据并高效求出最大数的数据结构。允许上述两种操作的数据结构称为优先队列 (priority queue)。很多算法需要用到优先队列，因此实现这些操作的高效方法会很有用。由于可以高效插入新数据，我们首先想到的也许是队列；但是求出最大的数据却需要扫描整个队列。另外一个建议是采用串行存放的有序链表。但是插入操作需要把整个链表的所有数据移动一遍。我们需要能够高效进行上述两种操作的数据结构。

堆是每一节点的值至少大于等于其子节点的值（如果存在）的完全二叉树。此定义暗示堆的根结点的值就是最大值。如果所有的数据各不相同，那么根结点包含最大的数据。

大于等于关系可以取反，使得父节点包含的数据至少小于子节点。这时，根节点包含最小的数据。但是由于历史传统的原因，我们假定越大的数据越接近根。因此可以通过移动数据的位置并保留二叉树的结构使得任何存在某种顺序的二叉树满足堆的性质。然而，经常遇到的情况是：给定 n 个数据，如 n 个整数，我们可以自由选择任意形状看上去最合理的二叉树。在这种情况下，我们可以选择完全二叉树。这也是我们在堆的定义中坚持用完全二叉树的原因。

现在考虑给定在 $A(1:n)$ 中的 n 个整数如何形成堆。一种策略是确定怎样把一个数据插入已经存在的堆中。如果可以完成这一操作，那么我们可以应用该算法 n 次，首先把数据插入一个空堆里，然后继续重复操作直到所有 n 个数据都插完。解决方法很简单，可以先把一个新数据放到堆的“底部”，然后将该数据和它的父节点、祖父节点、曾祖父节点等比较，直到它小于等于其中的一个节点为止。过程 INSERT 描述了这一过程的全部细节。

过程INSERT ((A,n))

$j \leftarrow n$

$i \leftarrow [n/2]$

```

item = A(n)
while i > 0 and A(i) < item do
  A(j) = A(i)// move the parent down
  j ← i
  i ← [i/2] // the parent of A(i) is at A([i/2])
End while
A(j) = item // a place for A(n) is found

```

从描述可以很清楚地看出，INSERT的时间会变化。在最好的情况下，新的数据一开始就被放在正确的位置，不需要移动任何数据。在最坏的情况下，while 循环的次数和堆的层数成正比。 $A(1:n)$ 中的 n 个数据可以通过下面的程序段转换成一个堆（即完全二叉树）。

```

for i ← 2 to n do
  Call INSERT (A,i)
End do

```

复杂度分析 在INSERT过程中，生成堆的数据集合最坏是把数据按升序插入。每一个新的数据都会上升变成新的根节点。

对于 $1 < i < [\log(n+1)]$ 的完全二叉树，其第 i 层最多有 2^{i-1} 个节点。对一个在第 i 层的节点，其到根节点的距离为 $i-1$ 。因此用INSERT过程生成堆的最坏情况下的时间复杂度为 $O(n \log n)$ 。

一个关于INSERT的令人惊奇的事实是对 n 个随机输入数据，其平均复杂度比最坏的情况要快，接近 $O(n)$ 而不是 $O(n \log n)$ 。这表示每一个新的数据只是引起常数次的数据移动。证明INSERT的这个性质很复杂，因此我们在此不给出证明。

另外一个生成堆的算法具有很好的性质，即其最坏的情况比 $n-1$ 次的INSERT调用快一个数量级。这种减少通过采用把 $A(1:n)$ 看作完全二叉树，并从树的叶结点到根节点逐层进行操作的算法达到。在每一层，左右子树都有可能不符合堆的性质。因此给出一种把只有根节点不符合堆性质的二叉树转换成堆的方法就足够了。过程ADJUST就是为根在位置 i 的子树进行这一操作的算法。这一算法假定此二叉树为前面讨论的二叉树的子树。

过程ADJUST ((A, i, n))

/* 根在 $A(2*i)$ 和 $A(2*i+1)$ 的完全二叉树与 $A(i)$ 结合生成一个简单堆， $1 < i < n$ 。节点值在1和 n 之间*/

```

j = 2*i;
item ← A(i)
while j ≤ n do
  If j < n and A(j) < A(j+1) then
    j = j+1
  endif
  If item ≥ A(j) then
    exit while
  else
    A([j/2]) = A(j)
    j = 2*j
  endif
endif

```

```

    endif
End while
A([j/2]) ← item
END

```

268

给定 $A(1:n)$ 中的 n 个数据，我们可以通过调用 ADJUST 生成一个堆。很容易看出叶节点已经是堆。因此我们可以把叶节点的父节点调用 ADJUST 作为开始，然后逐层上升，直到根节点。

过程HEAPIFY

```

BEGIN
  for i = [n/2] to 1 by step -1 do
    call ADJUST (A,i,n)
  Next i
END

```

我们已经把堆作为一种每一节点至少不小于其子节点的数据结构进行了讨论。现在我们给出利用堆数据结构进行排序的算法：

```

过程Heapsort((A, n))
call HEAPIFY
for i = n to 2 by step -1 do
  t = A(i)
  A(i) = A(1)
  A(1) = t
  call ADJUST (A,i-1)
  Next i

```

虽然只需要调用 $O(n)$ 次 HEAPIFY，ADJUST 的每一次调用可能需要 $O(\log n)$ 次操作。因此最坏情况下的时间复杂度为 $O(n \log n)$ 。注意对存储的需求，除了 $A(1:n)$ 外，仅针对一些简单变量。

9.2 合并排序

合并可以有效地应用于排序。考虑数组 $x(1:n)$ ，如果单独把 x_1 看作一个数组，那么该数组是已排序的；对于 x_2 也一样。那么，我们现在可以合并数组 x_1 和 x_2 ，得到已排序的数组 x_1, x_2 。相似地，可以合并已排序的数组 x_1, x_2 和 x_3, x_4 ，得到已排序数组 x_1, x_2, x_3, x_4 。

我们以 16 个数的数组为例子来进行说明：

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}
71	81	83	85	23	19	64	19	36	53	87	48	8	96	10	0

269

成对进行合并：

合并 x_1 和 x_2 ，得到 y_1, y_2

合并 x_3 和 x_4 ，得到 y_3, y_4

...

...

合并 x_{15} 和 x_{16} , 得到 $y_{15} y_{16}$

得到下面的表:

y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}	y_{16}
71	81	83	85	23	19	64	19	36	53	87	48	8	96	10	0

成对进行合并:

合并 $y_1 y_2$ 和 $y_3 y_4$, 得到 $x_1 x_2 x_3 x_4$

...

...

合并 $y_{13} y_{14}$ 和 $y_{15} y_{16}$, 得到 $x_{13} x_{14} x_{15} x_{16}$

我们得到下面的数列: 71, 81, 83, 85, 19, 23, 64, 36, 48, 53, 87, 0, 8, 10, 96。继续进行合并, 最后得到一个简单的有序数列。进行合并的数组对的选择方法可以改进以加快排序进程, 并为进一步的改进创造条件。改进的数组对的选择方法可以是 $(x_1 x_n), (x_2 x_{n-1}), \dots$, 排序的元素分别存储为 $(y_1 y_n), (y_2 y_{n-1}), \dots$ 。此方法叫做直接合并排序, 其特点是每次合并的数组元素数目是预先确定的, 即 $1, 2, 2^2, 2^3, 2^4, \dots$, 因而合并迭代的次数为 $\log n$ 。如果合并操作并行执行, 在CREW PRAM模型下, 我们可以在 $O(\log^2 n)$ 的时间内完成排序。

9.3 排序网络

270 Knuth发现超过25%的CPU时间用在排序上。排序已经变得如此重要, 因此, 现在是考虑用专用硬件实现数据排序功能的时候。本节我们假定两个基本排序硬件可以对两个数据进行排序。这在图9-1中给出。

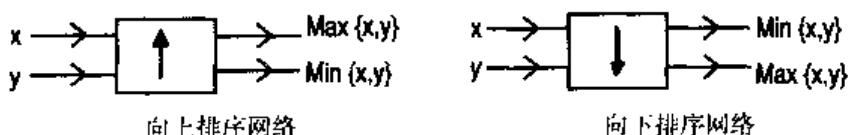


图9-1 基本排序网络

如果要对四个数 x_1, x_2, x_3, x_4 进行排序, 可以用基本排序网络设计一个新的排序网络。如图9-2所示。

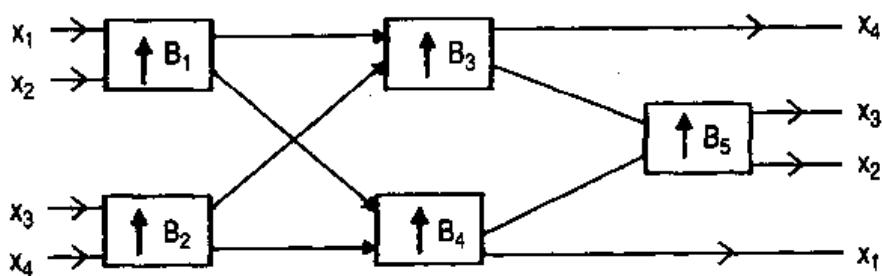


图9-2 对四个元素进行排序的排序网络

在图9-2中, x_1 和 x_2 用基本排序网络 B_1 进行排序, 同时用另外一个基本排序网络 B_2 对 x_3 和 x_4

进行排序。 B_1 和 B_2 较大的输出元素用 B_3 进行排序； B_3 输出的较大元素很明显是最大的元素，称作 x_4 ； B_1 和 B_2 较小的输出元素用 B_4 进行排序； B_4 输出的较小元素很明显是最小的元素，称作 x_0 ； B_3 较小的输出元素和 B_4 较大的输出元素用 B_5 排序，称作 x_2 和 x_1 。因此输出的数组 $(x_0, x_1, x_2, x_3, x_4)$ 是已排序的数组。

双调排序网络 如果 $A(0:n-1)$ 是双调数列，那么可以像双调合并一样对它进行排序。当 $A(0:n-1)$ 是给定的双调数列时，我们可以考虑下面的两个子数列：

$$A\left(0 : \frac{n}{2} - 1\right)$$

和

$$A\left(\frac{n}{2} : n - 1\right)$$

设

$$L_i = \min\left\{a_i, a_{i+\frac{n}{2}}\right\}, \quad 0 \leq i < \frac{n}{2}$$

$$R_i = \max\left\{a_i, a_{i+\frac{n}{2}}\right\}, \quad 0 \leq i < \frac{n}{2}$$

正如我们已经研究过的， L 和 R 会各自形成一个双调数列且 L 的每一个元素都小于等于 R 。如果 L 和 R 分别进行排序，把 R 的输出放在 L 后面，我们将得到已排序的数列。

能够对有 n 个元素的双调数列进行排序的网络（用 $B(n)$ 表示），称作大小为 n 的双调排序网络。正如上面所解释的，双调排序网络 $B(n)$ 可以用两个 $B(n/2)$ 的双调排序网络和一些基本排序网络来设计，双调排序网络 $B(8)$ 在图9-3中给出。

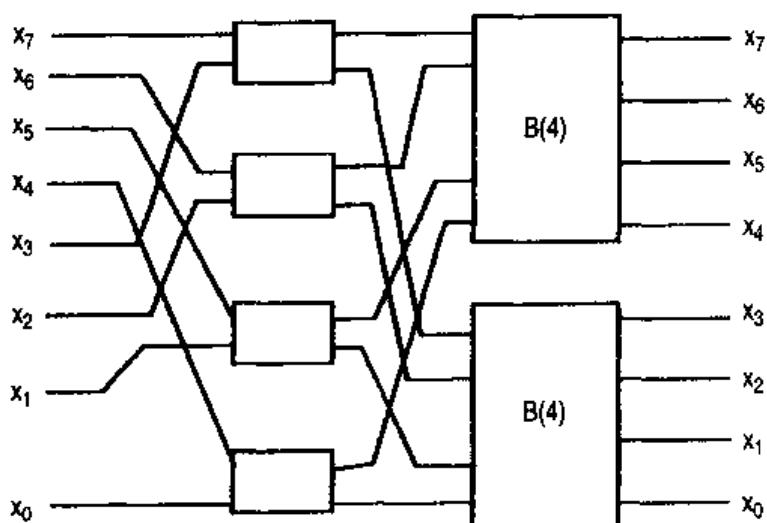


图9-3 双调排序网络 $B(8)$

参考文献

Ajtai, M., Komlos, J. and Szemerédi, E. (1983) Sorting in $c \log n$ Parallel Steps, *Combinatorica*, 3(1), 1–19.

Azar, Y. and Vishkum, U. (1987) Tight Comparison Bounds on the Complexity of Par-

- allel Sorting, *SIAM Journal of Computing*, **16**(3), 458–464.
- Batcher, K. (1968) Sorting Networks and Their Applications, *AFIPS Spring JOINT Computing Conference*, Atlantic City, NJ, pp. 307–314.
- Bilardi and Nicolau, A. (1989) Adaptive Bitonic Sorting, *SIAM J. Computing*, **18**(2), 216–228.
- Cole, R. (1988) Parallel Merge Sort, *SIAM J. Computing*, **17**(4), 770–785.
- Knuth, D. (1973) *Sorting and Searching*, Addison-Wesley, Reading, MA.

习题

- 9.1 证明任意双调数列满足惟一交叉属性 (unique crossover property)。
- 9.2 设 X 为一数组。设计并行算法用 x_0 划分数组，使得划分后 x_0 存在 x_i 所在的位置，且新的数组元素 $x_0, x_1, x_2, \dots, x_n$ 满足
- $$x_i < x_k, i < k$$
- $$x_k < x_j, j > k$$
- 9.3 设计快速排序算法的非递归版本。
- 9.4 画出对具有64个元素的数组进行排序的排序网络。
- 9.5 在长度是 2^t 的双调数列中，求出具有惟一交叉属性的水平切割所需要的比较操作的次数。

第四部分 数值算法

第10章 代数方程和矩阵

10.1 代数方程

考虑代数方程 $f(x) = 0$ 。如果 $f(a)$ 和 $f(b)$ 符号相反，那么在 a 和 b 之间存在一个根。例如方程 $x^2 - 3x + 2 = 0$

$$f(0) = 2 \quad \text{正的}$$

$$f(1.5) = -0.25 \quad \text{负的}$$

因此，方程 $x^2 - 3x + 2 = 0$ 在 0 和 1.5 之间有一个实根。

10.1.1 几何解释

考虑方程 $f(x) = 0$ ，我们在二维平面内画出曲线 $y = f(x)$ 。曲线与 x 轴的交点就是方程 $f(x) = 0$ 的根。例如方程 $x^2 - 3x + 2 = 0$ ，相应的曲线 $y = x^2 - 3x + 2$ 在图 10-1 中给出。[277]

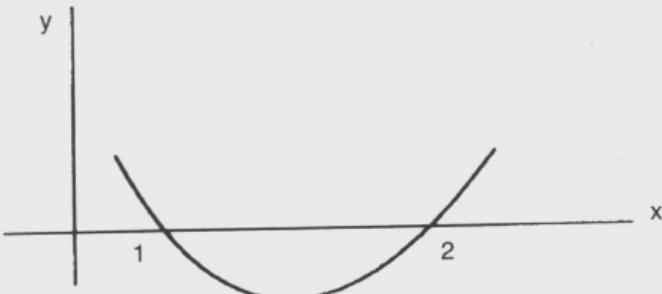


图 10-1 曲线 $y = x^2 - 3x + 2$

图中曲线与 x 轴有两个交点： $x = 1$ 和 $x = 2$ 。这两个交点的值就是方程 $x^2 - 3x + 2 = 0$ 的根。

为了求一个代数方程的实根，人们设计了多种迭代法。这些方法从根的一个近似值 x_0 出发，通过近似迭代公式求得一个比 x_0 更接近根 x 的值。这样的公式叫作迭代公式，其形式为 $x_{k+1} = f(x_k)$ 。

通过代入 $k = 0, 1, 2, \dots$ ，根的近似值得到逐步改进，并在精度达到要求的情况下终止这一过程。下面给出几种迭代法：

解方程 $f(x) = 0$ 的迭代法

编 号	方 法	迭代公式
1	牛顿-拉弗森方法	$x_{k+1} = x_k - f(x_k)/f'(x_k)$
2	Von Mises 公式	$x_{k+1} = x_k - f(x_k)/f'(x_0)$

(续)

编 号	方 法	迭代公式
3	弦法或割线法	$x_{t+1} = (x_t f(x_0) - x_0 f(x_t)) / (f(x_0) - f(x_t))$
4	试位法	$x_{t+1} = (x_{t-1} f(x_t) - x_t f(x_{t-1})) / (f(x_t) - f(x_{t-1}))$
5	逐次逼近法	如果给定方程 $f(x) = 0$, 将其改写成 $x = \Phi(x)$, 那么迭代公式为 $x_{t+1} = \Phi(x_t)$

上述五种方法是内在串行的。另外一个称作对分法的迭代法可以发掘一些内在的并行性。

10.1.2 对分法

278 考虑方程 $f(x) = 0$ 。 a 和 b 是两个点且 $f(a)$ 和 $f(b)$ 符号相反, 那么在 a 和 b 之间 $f(x) = 0$ 有一个根。考虑 a 和 b 间的中点: $m = (a+b)/2$ 。

已知 $f(a)$ 和 $f(b)$ 符号相反, 如果 $f(a)$ 和 $f(m)$ 也符号相反, 那么 $f(x) = 0$ 在 a 和 m 之间有一个根。区间 (a,m) 的长度是 (a,b) 的一半。如果 $f(a)$ 和 $f(m)$ 的符号不相反, 那么 $f(m)$ 和 $f(b)$ 符号相反。这样根在 m 和 b 之间。

这样我们就把区间 (a,b) 分成两个长度相同的子区间 (a,m) 和 (m,b) , 而且在其中一个子区间内有根存在。如果重复这一过程, 那么含有根的区间长度每次都减少一半。当区间长度很小的时候, 值 m 就被认为是方程的根。

我们先给出这一问题的串行算法。假定区间的初始值 a 和 b 已知, 且 $f(a)$ 和 $f(b)$ 符号相反(即 $f(a) \times f(b) < 0$)。

算法 Bisection-SEQ

BEGIN

1. While $|a-b| > \epsilon$ do
2. $m = (a+b)/2$
3. If $f(a)f(m) < 0$ then
 - $b = m$
 - else
 - $a = m$
- Endif
4. End While
5. Return $(a+b)/2$

END

在这里, ϵ 是结果所需要的精度。如果 $1/2^n < \epsilon/b - 1 < 1/2^{n-1}$, 算法 Bisection-SEQ 的 While 循环重复执行 n 次。下面的算法用并行方式来确定代数方程的根, 并要用到 CRCW PRAM 模型。

假定我们只有 p 个处理器, 那么区间 (a,b) 被划分成 p 个子区间, 每个子区间分配一个处理器。各处理器判断根是否落在自己的子区间内, 并把该子区间看作区间 (a,b) 。这样做可以提高计算时间 p 倍。算法如下:

算法 Bisection-PARALLEL-1

输入: 给定函数 $f(x)$, 精度 ϵ

输出：根

1. While $|b-a| > \epsilon$ do
2. $x_0 = a$
3. For $i = 1$ to p do in Parallel
4. $x_i = x_{i-1} + i(b-a)/p$
5. If $f(x_{i-1})f(x_i) < 0$ then
6. $a = x_{i-1}$
7. $b = x_i$
8. endif
9. End Parallel
10. End While
11. root = $(a+b)/2$
12. End

279

复杂度分析 算法的第5~6步中，会有1个以上的处理器同时对 a 和 b 赋值。因此，实现该算法需要允许并发写操作的PRAM模型。 x_i 的值既会被第*i*个处理器用到，也会被第*i*+1个处理器用到，因此也需要允许并发读操作。这样，为了实现该算法，需要用CRCW PRAM模型。

10.2 矩阵的行列式

考虑矩阵

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

下面是矩阵的初等行变换：

变换1：矩阵一行的所有元素都乘以一个常数。如果第*i*行乘以常数*k*，我们写做 $R_i \leftarrow kR_i$ 。

变换2：矩阵一行的所有元素都乘以一个常数，并用另外一行减它。即 R_i 被 $R_i - kR_j$ 替代。

我们写做 $R_i \leftarrow R_i - kR_j$ 。

变换3：两行互换。如果 R_i 和 R_j 互换，我们写做 $R_i \leftrightarrow R_j$ 。

完成变换1之后，矩阵行列式的值相当于乘以*k*；变换2对矩阵行列式的值没有影响；变换3改变矩阵行列式值的符号。

280

变换	对行列式的影响
1	乘以 <i>k</i>
2	无影响
3	符号取反

考虑第一列对角线下面的所有元素为零的矩阵：

$$|A| = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & a_{32} & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

$$= a_{11} \begin{bmatrix} a_{22} & a_{23} & a_{24} & \cdots & a_{2n} \\ a_{32} & a_{33} & a_{34} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n2} & a_{n3} & a_{n4} & \cdots & a_{nn} \end{bmatrix}$$

主元消去法 对矩阵

$$|A| = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

281 进行行变换

$$R_2 \leftarrow R_2 - \frac{a_{21}}{a_{11}} R_1$$

如果对矩阵A进行上面的变换，那么 a_{21} 会变成零。同样，如果进行如下变换：

$$R_i \leftarrow R_i - \frac{a_{ii}}{a_{11}} R_1, i = 2, 3, \dots, n$$

那么第一列对角线下面的所有元素都变为零。这些变换对矩阵A行列式的值没有影响。经过上面的变换， a_{ij} 会变成 $a_{ij} - (a_{ii}/a_{11})a_{1j}$ 。如仍用 a_{ij} 表示，变换后的矩阵如下：

$$|A| = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

$$= a_{11} \begin{bmatrix} a_{22} & a_{23} & a_{24} & \cdots & a_{2n} \\ a_{32} & a_{33} & a_{34} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n2} & a_{n3} & a_{n4} & \cdots & a_{nn} \end{bmatrix}$$

我们对化简的矩阵重复上面的过程，得到矩阵的行列式：

$$|A| = a_{11} a_{22} \begin{bmatrix} a_{33} & a_{34} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ a_{n3} & a_{n4} & \cdots & a_{nn} \end{bmatrix}$$

A 是一个 $n \times n$ 的矩阵，通过第一步，我们把它化简成一个 $(n-1) \times (n-1)$ 的矩阵。现在更进一步化简成 $(n-2) \times (n-2)$ 的矩阵。

重复上述过程，可以把矩阵化简成一个 1×1 的矩阵。所以，最后行列式的值为 $|A| = a_{11} a_{22} \cdots a_{nn}$ 。

算法设计 设 A 为一给定矩阵，

1. 对 $i = 2, 3, 4, \dots, n$ ，进行行变换：

$$R_i \leftarrow R_i - \frac{a_{i1}}{a_{11}} R_1$$

使得第一列对角线下面的元素都变成零。

2. 对 $i = 3, 4, \dots, n$ ，进行行变换：

$$R_i \leftarrow R_i - \frac{a_{i2}}{a_{22}} R_2$$

282

使得第二列对角线下面的元素都变成零。

3. 对 $i = 4, \dots, n$ ，进行行变换：

$$R_i \leftarrow R_i - \frac{a_{i3}}{a_{33}} R_3$$

使得第三列对角线下面的元素都变成零。

一般为了把第 k 列对角线下面的元素变成零，进行如下行变换：

$$R_i \leftarrow R_i - \frac{a_{ik}}{a_{kk}} R_k, \quad i = k+1, k+2, \dots, n$$

对 $k = 1, 2, 3, \dots, n-1$ ，进行上述变换，使得矩阵对角线下面的所有元素都变成零。最后得到矩阵的行列式：

$$|A| = a_{11} a_{22} \cdots a_{nn}$$

下面的程序段就可以完成所需要的行变换：

```
ratio = a_{ii}/a_{11}
For j = 1 to n
  a_{ij} = a_{ij} - ratio*a_{1j}
next j
```

为了让第 k 列对角线下面的元素变成零，就要对 $i = k+1, \dots, n$ 重复上述变换。完整的算法如下：

算法 Sequential Pivotal Condensation

输入：给定矩阵 $A(1:n, 1:n)$

输出：行列式的值

1. For $k = 1$ to $n-1$
2. For $i = k+1$ to n

3. ratio = a_{kk}/a_{kk}
4. Do the row operation $R_i \leftarrow \text{ratio} * R_i$
5. next i
6. next k
7. Determinant $a_{11} * a_{22} * \dots * a_{nn}$
8. END

在上面的串行算法中，第2~5步的for循环使得第 k 列对角线下面的元素变成零。在此循环
[283] 中，任意两个迭代之间是互相独立的，因而可以由不同的处理器并行执行。所有的处理器都要用到第 k 行的元素，所以实现此算法需要用到支持并发读的PRAM模型。第4步的行变换可以由下面的for循环完成：

- 4.1 For $j = 1$ to n
- 4.2 $a_{ij} = a_{ij} - \text{ratio} * a_{kj}$
- 4.3 next j

也可以由 n 个不同的处理器并行完成：

- 4.1 For $j = 1$ to n do in Parallel
- 4.2 $a_{ij} = a_{ij} - \text{ratio} * a_{kj}$
- 4.3 End Parallel

第7步需要 $O(n)$ 个处理器，且在 $O(\log n)$ 时间内并行完成。完整的并行算法如下：

算法Determinant

输入： $A(1:n, 1:n)$

输出： $\det(A)$

BEGIN

1. For $k = 1$ to $n-1$
2. For $i = k+1$ to n do in parallel
3. ratio = a_{kk}/a_{kk}
4. For $j = 1$ to n do in Parallel
5. $a_{ij} = a_{ij} - \text{ratio} * a_{kj}$
6. End Parallel
7. End Parallel
8. Next k
9. $\det = a_{11} * a_{22} * \dots * a_{nn}$

END

第1~8步的循环执行次数为 $n-1$ 。因此计算时间为 $O(n)$ 。在第2~7步和第4~6步有两个并行嵌套循环，因此需要 $O(n^2)$ 个处理器。第9步与求和算法相似，需要 $O(n)$ 个处理器，且在 $O(\log n)$ 时间内完成。所以上述并行算法需要 $O(n^2)$ 个处理器，在 $O(n)$ 时间内完成。

[284] 10.3 线性方程组

线性方程组一般表示形式为：

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= a_{1,n+1} \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= a_{2,n+1} \\ \cdots &\quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ \cdots &\quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= a_{n,n+1} \end{aligned}$$

线性方程组的解是指满足上面方程的一组值 x_1, x_2, \dots, x_n 。

如果用下面的矩阵表示:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

则此矩阵叫做系数矩阵 (coefficient matrix)。而向量

$$\begin{bmatrix} a_{1,n+1} \\ a_{2,n+1} \\ a_{3,n+1} \\ \cdots \\ \cdots \\ a_{n,n+1} \end{bmatrix}$$

叫做右端向量 (right-hand-side vector)。在一些特殊情况下, 可以直接求得方程组的解。

例1

如果一个方阵除了对角线元素非零之外, 其他非对角线元素都为零, 则叫做对角矩阵 [285] (diagonal matrix)。假定系数矩阵是对角矩阵。即系数矩阵的形式如下:

$$\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ \cdots & \cdots & a_{33} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

那么对应的方程组满足下面形式:

$$\begin{aligned} a_{11}x_1 &= a_{1,n+1} \\ a_{22}x_2 &= a_{2,n+1} \\ \cdots &\quad \cdots \quad \cdots \\ \cdots &\quad \cdots \quad \cdots \\ a_{nn}x_n &= a_{n,n+1} \end{aligned}$$

这个方程组的解可以直接写成:

$$x_1 = \frac{a_{1,n+1}}{a_{11}}, \quad x_2 = \frac{a_{2,n+1}}{a_{22}}, \quad \dots, \quad x_n = \frac{a_{n,n+1}}{a_{nn}}$$

例2

如果一个矩阵对角线之上的所有元素都是零, 则称作下三角矩阵 (lower triangular)。假定系数矩阵为下三角矩阵, 即有如下形式:

286

$$\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ \cdots & \cdots & a_{33} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

相应的方程组为:

$$\begin{aligned} a_{11}x_1 &= a_{1,n+1} \\ a_{21}x_1 + a_{22}x_2 &= a_{2,n+1} \\ \cdots &\cdots \cdots \cdots \cdots \cdots \cdots \cdots \\ \cdots &\cdots \cdots \cdots \cdots \cdots \cdots \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= a_{n,n+1} \end{aligned}$$

从第一个方程可求得 $x_1 = a_{1,n+1}/a_{11}$ 。求出 x_1 后, 将其代入第二个方程, 我们可以求出 x_2 :

$$x_2 = \frac{1}{a_{22}}(a_{2,n+1} - a_{21}x_1)$$

求出 x_2 后, 我们可以用下面的等式求出 x_3 :

$$x_3 = \frac{1}{a_{33}}(a_{3,n+1} - a_{31}x_1 - a_{32}x_2)$$

类似地, 我们可以求出 x_4, x_5, \dots, x_n 。上面的方法称作向前消去法 (forward substitution method)。

例3

假定系数矩阵是一个上三角矩阵。那么方程组有下面的形式:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= a_{1,n+1} \\ a_{22}x_2 + \cdots + a_{2n}x_n &= a_{2,n+1} \\ \cdots &\cdots \cdots \cdots \cdots \cdots \cdots \cdots \\ a_{nn}x_n &= a_{n,n+1} \end{aligned}$$

我们可以从最后一个方程开始。由最后一个方程, 我们可以求出 x_n 的值:

$$x_n = \frac{a_{n,n+1}}{a_{nn}}$$

接着利用第 $(n-1)$ 个方程求出 x_{n-1} :

$$x_n = \frac{1}{a_{n-1,n-1}} (a_{n-1,n+1} - a_{n-1,n} x_n)$$

一般在求出 $x_n, x_{n-1}, \dots, x_{k+1}$ 之后，我们再求 x_k 。

这样，我们可以求出所有的 x_i 。此方法称作向后消去法（backward substitution）。

10.3.1 高斯消元法

287

考虑方程组：

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = a_{1,n+1}$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = a_{2,n+1}$$

...

...

$$a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = a_{n,n+1}$$

可以用下面的矩阵形式表示：

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{1,n+1} \\ a_{2,n+1} \\ a_{3,n+1} \\ \cdots \\ \cdots \\ a_{n,n+1} \end{bmatrix}$$

对上面的矩阵进行主元消去法的行变换，可以化简成下面的形式：

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & 0 & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

这样，我们就可以用向后消去法进行求解。此算法主要由以下三步组成：

1. 读矩阵 A ；
2. 将矩阵化成上三角形式；
3. 用向后消去法进行求解。

我们已经在主元消去法中对矩阵化简成上三角形式进行了讨论。该操作用到CREW PRAM模型，需要 $O(n^2)$ 个处理器，且在 $O(n)$ 时间内完成。在向后消去法中， x_k 只能在求出 $x_n, x_{n-1}, \dots, x_{k+1}$ 之后进行求解。

288

求解 x_i 的运算是内在串行的。每一个 x_i 需要 $O(n)$ 个处理器，且在 $O(\log n)$ 时间内完成。因此高斯消元法可以在CREW PRAM模型中用 $O(n^2)$ 个处理器并在 $O(n)$ 时间内实现。

部分选主元 在高斯消元法中，对第 k 行所进行的行变换是先用主元 a_{kk} 除，再用 a_{ik} 乘。如果 a_{kk} 是零或者在量级上相对比较小，那么此过程将会导致诸如截断误差或舍入误差等方面的

计算错误。为了避免这一点，我们可以对第 k 列的对角元素和对角线下面的元素进行选择。如果 a_{kk} 是最大的元素，我们可以交换第 k 行和第1行，然后继续进行消元过程。此方法叫做部分选主元法 (partial pivoting)。在前面算法DETERMINANT解释过的消元过程中，在第1步和第2步之间我们要查找第 k 列对角线下和对角线的元素，以便找到最大的元素 a_{kk} 。然后我们要把第 k 列和第1列互换。在 $O(n)$ 个处理器上，选择最大元素的运算可以在 $\log(n-k+1)$ 时间内完成。所以，如果在高斯消元法中加入部分选主元，在CREW PRAM模型中需要用 $O(n^2)$ 个处理器，且在 $O(n \log n)$ 时间内完成。

Sameh和Kuck以经典的Givens旋转矩阵分解法为基础给出了一个并行算法。

10.3.2 Givens旋转

考虑一个常见的 n 阶稠密方阵。设三个整数 $i, j, k < n$ 。假定在第 i 行和第 j 行中前面的 $(k-1)$ 列元素都是零，而第 k 列元素都非零。即有下面的形式：

$$\begin{bmatrix} * & * & * & \cdots & \cdots & \cdots & \cdots & * \\ \cdots & \cdots \\ \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 0 & * & * & \cdots & * \\ \cdots & \cdots \\ \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 0 & * & * & \cdots & * \\ \cdots & \cdots & \cdots & \cdots & \cdots & * & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & * & \cdots & \cdots & \cdots \end{bmatrix}$$

↑
第 k 列

上面矩阵形式中的*表示非零元素。Givens旋转法就是使得下标为 (j,k) 的元素为零的方法。利用Givens旋转法，我们可以把矩阵对角线之下的元素变为零。下面我们来看一下Givens旋转法的具体过程。考虑下面的矩阵：

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ \cdots & 0 \\ \cdots & 0 \\ 0 & 0 & 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \cdots & \cdots \\ \cdots & \cdots \\ 0 & 0 & 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \cdots & \cdots \\ 0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 1 \end{bmatrix}$$

↑ ↑
第 i 列 第 j 列

其中 $c = \frac{a_{ik}}{\sqrt{a_{ik}^2 + a_{jk}^2}}$, $s = \frac{a_{jk}}{\sqrt{a_{ik}^2 + a_{jk}^2}}$ 。矩阵 G 称作 Givens 旋转矩阵 (Givens rotation matrix)。

下面给出矩阵乘积 GA 的一些有趣的性质。

定理 10-1

1. G 是一个正交矩阵;
2. 对 $p = i$ 和 $p = j$, 有 $GA(p, q) = a_{pq}$;
3. 对 $q < k$, 有 $GA(i, q) = GA(j, q) = 0$;
4. $GA(j, k) = 0$ 。

证明: 从欧几里德范数可以验证 G 是正交矩阵。如果 $p = i$ 且 $p = j$, 那么 G 的第 p 行和单位矩阵第 p 行相同。当此行用在矩阵乘积 GA 时, 并不影响 A 的值。因为当 $q < k$ 时, $a_{iq} = 0$ 且 $a_{jq} = 0$ 。

$$\begin{aligned} GA(j, k) &= \sum_{i=1}^n g_{ji} a_{ik} \\ &= \sum_{i=i,j} g_{ji} a_{ik} \\ &= -sa_{ik} + ca_{jk} \\ &= 0 \end{aligned}$$

选择矩阵 A 的两行, 我们可以应用 Givens 旋转, 使得矩阵下标为 (j, k) 的元素的值为零。特别要注意的是 Givens 旋转只影响所涉及的行。选择不同的矩阵行对, 我们可以并行应用 Givens 旋转, 并使得对角线下面的元素为零。如果想把第 k 行对角线下面的所有元素都变成零, 我们可以将第 $(k+1), (k+2), \dots, n$ 行和第 $1, 2, \dots, k$ 行成对组合, 然后并行应用 Givens 旋转。

BEGIN

For $j = k+1$ to n do in Parallel

(i is a number from 1 to k and distinct for each j .)

Apply Given's rotation for i th and j th row

End parallel

END

上面的算法段使得对角线下面的矩阵元素变成零。我们来分析 Givens 旋转的复杂度。设有两个矩阵:

$$A = (a_{ij}) \quad G = (g_{ij})$$

矩阵乘积 GA 中只有矩阵 A 的第 i 行和第 j 行改变, 而其他的元素没有变化。我们要求 GA 并把结果存在相同的变量中。Givens 旋转只影响矩阵 A 的下列元素:

$$\begin{aligned} a_{ik}, a_{i,k+1}, \dots, a_{in} \\ a_{jk}, a_{j,k+1}, \dots, a_{jn} \end{aligned}$$

因此不需要求解矩阵乘积 GA , 只需求 GA 的这些元素就足够了。由于 GA 的第 i 行是:

$$(GA)_{it} = \sum_{q=i,j} g_{iq} a_{qt}, \text{ 其中 } t > k$$

同样,

$$(GA)_{jt} = \sum_{q=i,j} g_{jq} a_{qt}, \text{ 其中 } t > k$$

因此，对每一个 $t > k$ ， $(GA)_n$ 和 $(GA)_{\bar{n}}$ 可以在 $O(1)$ 时间内求出。

下面的程序段也应用了Givens旋转：

1. For $t = k$ to n do in Parallel

2. $a_n \leftarrow g_{ii} a_n + g_{ij} a_{ji}$

3. $a_{ji} \leftarrow g_{ji} a_{ji} + g_{ii} a_{ii}$

4. End Parallel

如果用 $O(n)$ 个处理器，一对行的Givens旋转可以在 $O(1)$ 时间内完成。一矩阵行所有对角线之下的元素可以同时用 $\lceil n/2 \rceil$ 对矩阵行的Givens旋转变换成零。所以，用 $O(n^2)$ 个处理器把一矩阵行所有对角线以下的元素变成零的时间为 $O(1)$ 。用 $O(n^2)$ 个处理器把一个矩阵化简成下对角矩阵的时间为 $O(n)$ 。

10.4 傅里叶变换

负数的平方根在实数域中没有定义。数学家们定义了虚数 $i = \sqrt{-1}$ ，并且发展了称为复分析的理论。令人惊讶的是这一理论在实际中有大量应用，如数字图像处理等。

[292] $1^{1/2}$ 有两个根+1和-1。同样地， $1^{1/3}$ 有三个复根。推广到 $1^{1/n}$ ，它有 n 个根，将其分别记为 $\omega, \omega^2, \omega^3, \dots, \omega^{n-1}, 1$ ，它们称为 n 次单位根，其中 ω 称为单位原根。可以很容易求出：

$$\omega^k = \cos \frac{2k\pi}{n} + i \sin \frac{2k\pi}{n}$$

当 $n = 3$ 时，相应的三个单位根为：

$$\begin{aligned}\omega &= \cos \frac{2 \cdot 1 \cdot \pi}{3} + i \sin \frac{2 \cdot 1 \cdot \pi}{3} \\&= \cos 120^\circ + i \sin 120^\circ \\&= \frac{-1}{2} + i \frac{\sqrt{3}}{2} \\ \omega^2 &= \cos \frac{2 \cdot 2 \cdot \pi}{3} + i \sin \frac{2 \cdot 2 \cdot \pi}{3} \\&= \cos 240^\circ + i \sin 240^\circ \\&= \frac{-1}{2} - i \frac{\sqrt{3}}{2} \\ \omega^3 &= \cos 2\pi + i \sin 2\pi \\&= 1\end{aligned}$$

因此3次单位根为：

$$\omega = \frac{-1}{2} + i \frac{\sqrt{3}}{2}$$

$$\omega^2 = \frac{-1}{2} - i \frac{\sqrt{3}}{2}$$

$$\omega^3 = 1$$

类似地, $1^{1/4}$ 的四个单位根分别为:

$$\omega = \cos \frac{2\pi}{4} + i \sin \frac{2\pi}{4} = \cos \frac{\pi}{2} + i \sin \frac{\pi}{2} = i$$

$$\omega^2 = \cos \pi + i \sin \pi = -1$$

$$\omega^3 = \cos \frac{3\pi}{2} + i \sin \frac{3\pi}{2} = -i$$

$$\omega^4 = \cos 2\pi + i \sin 2\pi = 1$$

因此, $i, -1, -i$ 和 1 是 4 次单位根。令 W 为 $n \times n$ 矩阵, 行和列的标记分别从 0 到 $n-1$, 且 [293]
 $W(i,j) = \omega^i$, 这里 ω 表示 n 次单位原根。

即有:

$$W(0,0) = \omega^0 = 1$$

$$W(0,1) = \omega^0 = 1$$

$$W(0,2) = \omega^0 = 1$$

更一般地,

$$W(0,j) = 1, j = 0, 1, 2, \dots, n-1$$

$$W(i,0) = 1, i = 0, 1, 2, \dots, n-1$$

$$W(1,1) = \omega^1$$

$$W(1,2) = \omega^2$$

$$W(1,3) = \omega^3$$

矩阵 W 表示如下:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 & \cdots & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^4 & \cdots & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^8 & \cdots & \cdots & \omega^{2(n-1)} \\ \cdots & \cdots \\ \cdots & \cdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \cdots & \cdots & \cdots & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

由于 $\omega^n = 1$, W 可用 $\omega, \omega^2, \omega^3, \dots, \omega^{n-1}, 1$ 来表示:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & \cdots & \cdots & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \cdots & \cdots & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \cdots & \cdots & \cdots & \omega^{n-2} \\ \cdots & \cdots \\ \cdots & \cdots \end{bmatrix}$$

假定

$$X = [x_0 \ x_1 \ \cdots \ x_{n-1}]$$

为 n 维向量。向量 X 的离散傅里叶变换为向量 $Y = WX$. [294]

例

取 $n = 3$, 我们已经计算出 $\omega = -1/2 + (\sqrt{3}/2)i$, $\omega^2 = -1/2 - (\sqrt{3}/2)i$ 和 $\omega^3 = 1$, 矩阵 W 为

$$W = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega & \omega^2 \\ 1 & \omega^2 & \omega \end{bmatrix}$$

假设

$$X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

向量 X 的傅里叶变换为:

$$Y = WX = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega & \omega^2 \\ 1 & \omega^2 & \omega \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

即:

$$\begin{aligned} Y_0 &= x_0 + x_1 + x_2 \\ Y_1 &= x_0 + \omega x_1 + \omega^2 x_2 \\ Y_2 &= x_0 + \omega^2 x_1 + \omega x_2 \end{aligned}$$

如果

$$X = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

则 X 的傅里叶变换为:

$$Y = \begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \end{bmatrix}$$

295

其中

$$\begin{aligned} Y_0 &= x_0 + x_1 + x_2 = 1 - 1 + 0 = 0 \\ Y_1 &= x_0 + \omega x_1 + \omega^2 x_2 \\ &= 1 - \omega \\ &= 1 + \frac{1}{2} - \frac{\sqrt{3}}{2} i \\ &= \frac{3}{2} - \frac{\sqrt{3}}{2} i \\ Y_2 &= x_0 + \omega^2 x_1 + \omega x_2 \\ &= 1 - \omega^2 \\ &= 1 + \frac{1}{2} + \frac{\sqrt{3}}{2} i \\ &= \frac{3}{2} + \frac{\sqrt{3}}{2} i \end{aligned}$$

因此向量 X 的傅里叶变换为:

$$Y = \begin{bmatrix} 0 \\ \frac{3}{2} - \frac{\sqrt{3}}{2}i \\ \frac{3}{2} + \frac{\sqrt{3}}{2}i \end{bmatrix}$$

现在我们设计一个并行算法来求给定向量 X 的傅里叶变换。给定 n , 可以利用下面公式计算 ω^k 的值。[296]

$$\omega^k = \cos \frac{2k\pi}{n} + i \sin \frac{2k\pi}{n} \quad k = 0, 1, 2, \dots, n-1$$

利用 n 个处理器并行计算 ω^k , 而矩阵 W 的元素用 n^2 个处理器并行处理, 即:

$$W(i, j) = \omega^{ij}, \quad 0 \leq i, j \leq n-1$$

因此, 如果用 n^2 个处理器, 则矩阵 W 就可以在 $O(1)$ 时间内构造出来。傅里叶变换是 $n \times n$ 矩阵 W 与向量 X 的乘积。如果用通常的矩阵乘法, 上述运算需要 $O(n^3)$ 个处理器, 执行时间为 $O(\log n)$ 。并行算法如下:

算法Fourier

输入: $X(0:n-1)$

输出: $Y(0:n-1)$

1. For $k = 0$ to $n-1$ in parallel
2. $\omega^k = \cos 2\pi k/n + i \sin 2\pi k/n$
3. End parallel
4. For $i = 0$ to $n-1$ do in parallel
5. For $j = 0$ to $n-1$ do in parallel
6. $W(i, j) = \omega^{ij}$
7. End parallel
8. End parallel
9. For $i = 0$ to $n-1$ do in parallel
10. $Y_i = \sum_{j=0}^{n-1} W(i, j) * x_j$
11. End parallel

上述算法中第10步是求和算法, 需要 $O(n)$ 个处理器, 执行时间为 $O(\log n)$ 。因此上述算法需要 $O(n^2)$ 个处理器, 总的执行时间为 $O(\log n)$ 。

傅里叶变换有效并行算法 本节我们设计分治 (divide-and-conquer) 并行算法, 该算法在同一时间内可以用较少的处理器来求傅里叶变换。即算法所用时间为 $O(\log n)$, 处理器数为 $O(n \log n)$ 。

在离散傅里叶变换的有效并行算法设计中, 矩阵 W 的结构和 $\omega^n = 1$ 的性质起到了很关键的作用。下面列出对不同的 n , 矩阵 W 的值。当 $n = 4$ 时,

[297]

$$W = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^3 & \omega \end{bmatrix}$$

当 $n = 8$ 时,

$$W = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{bmatrix}$$

在偶数行利用 $\omega^4 = -1$ 的性质, 有

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & -1 & -\omega & -\omega^2 & -\omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & -\omega^2 & \omega & -1 & -\omega^3 & \omega^2 & -\omega \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\omega & \omega^2 & -\omega^3 & -1 & \omega & -\omega^2 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & -\omega^3 & -\omega^2 & -\omega & -1 & \omega^3 & \omega^2 & \omega \end{bmatrix}$$

注意:

1. 当 i 为偶数且 $0 \leq i \leq n-1$ 时, $W(i,j) = W(i,2j)$ ($0 \leq j \leq n/2$);
2. 当 i 为奇数且 $0 \leq i \leq n-1$ 时, $W(i,j) = -W(i,2j)$ ($0 \leq j \leq n/2$)。

计算 $Y = WX$, 便有如下的结论:

1. 当 i 为偶数且 $0 \leq i \leq n-1$ 时,

[298]

$$y_i = \sum_{j=0}^{n/2-1} W(i,j)(x_j + x_{j+n/2})$$

2. 当 i 为奇数且 $0 \leq i \leq n-1$ 时,

$$y_i = \sum_{j=0}^{n/2-1} W(i,j)(x_j - x_{j+n/2})$$

已知 ω^2 是 $n/2$ 次单位原根, 即 $n/2$ 次单位根为 $1, \omega^2, \omega^4, \dots, \omega^{n-2}$ 。又假定 i 是偶数且满足 $0 \leq i \leq n-1$, 令 $i = 2t$, $0 \leq t \leq n/2$ 。

$$\begin{aligned} y_i &= \sum_{j=0}^{n/2-1} W(i,j) * (x_j + x_{j+n/2}) \\ &= \sum_{j=0}^{n/2-1} \omega^{ij} * (x_j + x_{j+n/2}) \end{aligned}$$

$$y_{2t} = (\omega^2)^t * (x_i + x_j + n/2)$$

$$y_{2t} = \sum_{j=0}^{n/2-1} (\omega^2)^j * (x_i + x_j + n/2)$$

可以看出

$$\begin{bmatrix} Y_0 \\ Y_2 \\ Y_4 \\ \vdots \\ Y_{n-2} \end{bmatrix}$$

是如下向量的傅里叶变换：

$$\begin{bmatrix} x_0 + x_{n/2} \\ x_1 + x_{n/2+1} \\ \cdots \\ \cdots \\ \cdots \\ x_j + x_{j+n/2} \\ x_{n/2-1} + x_{n-1} \end{bmatrix}$$

类似地，当*i*是奇数且满足 $0 \leq i \leq n-1$ 时，令*i*= $2t+1$, $0 \leq t \leq n/2$ ，可以推导出：

$$y_i = \sum_{j=0}^{n/2-1} (\omega^2)^j * \omega^j (x_j + x_{j+n/2})$$

这表明

$$\begin{bmatrix} Y_1 \\ Y_3 \\ Y_5 \\ \vdots \\ Y_{n-1} \end{bmatrix}$$

是如下向量的傅里叶变换：

$$\begin{bmatrix} x_0 + x_{n/2} \\ \omega(x_1 - x_{n/2+1}) \\ \omega^2(x_2 - x_{n/2+2}) \\ \cdots \\ \cdots \\ \omega^{n/2-1}(x_{n/2-1}) + x_{n-1} \end{bmatrix}$$

计算完下面两个傅里叶变换

$$\begin{bmatrix} Y_0 \\ Y_2 \\ \vdots \\ Y_{n-2} \end{bmatrix}$$

和

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ \vdots \\ Y_{n-1} \end{bmatrix}$$

[300] 我们可以计算向量 $\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$ 的傅里叶变换

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

当 $n = 2$ 时，向量

$$\begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

的傅里叶变换可以直接计算。

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0 + x_1 \\ x_0 - x_1 \end{bmatrix}$$

下面给出计算 n 维向量傅里叶变换的有效分治并行算法。

算法 Efficient-FOURIER

输入： $x(0:n)$

输出： $Y(0:n-1)$

1. If $n = 2$ then

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0 + x_1 \\ x_0 - x_1 \end{bmatrix} \text{ and exit}$$

2. Recursively evaluate the Fourier transform of the vector

$$\begin{bmatrix} x_0 + x_{n/2} \\ x_1 + x_{n/2+1} \\ \dots \\ \dots \\ \dots \\ x_j + x_{j+n/2} \\ x_{n/2-i} + x_{n-1} \end{bmatrix} \text{ and store as } \begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ \vdots \\ Y_{n/2} \end{bmatrix}$$

[301]

3. Recursively evaluate the Fourier transform of the vector

$$\begin{bmatrix} x_0 + x_{n/2} \\ \omega(x_1 - x_{n/2+1}x_{n/2}) \\ \omega^2(x_1 - x_{n/2+1}) \\ \cdots \\ \cdots \\ \cdots \\ x_j + x_{j+n/2} \\ \omega_{n/2-1}(x_{n/2-1} + x_{n-1}) \end{bmatrix} \text{ and store as } \begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ \vdots \\ Y_s \\ \vdots \\ Y_{n-1} \end{bmatrix}$$

4. Output the vector $Y(0:n-1)$

复杂度分析 上面算法的第2步与第3步是彼此独立的，可以并行执行。整个过程是将一个 n 维向量分为两个 $n/2$ 维向量，且以递归方式执行。因此整个过程要用 $O(n \log n)$ 个处理器，执行时间为 $O(\log n)$ 。EREW PRAM模型可以实现此算法。

现在，我们来看一个有趣且常用的应用。

10.5 多项式乘法

考虑 n 次多项式 $a(x)$

$$a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

假定 $a_n \neq 0$ （否则为 $n-1$ 次多项式）。

本节主要讨论次数分别为 n 和 m 的两个多项式的乘积。

$$\begin{aligned} a(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_nx^n \\ b(x) &= b_0 + b_1x + b_2x^2 + \cdots + b_mx^m \end{aligned}$$

乘积 $a(x)b(x)$ 是 $(m+n)$ 次多项式，设 $a(x)b(x) = c(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{m+n}x^{m+n}$ 。

这里

$$\begin{aligned} c_0 &= a_0b_0 \\ c_1 &= a_0b_1 + a_1b_0 \\ c_2 &= a_0b_2 + a_1b_1 + a_2b_0 \\ c_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 \\ &\cdots \\ &\cdots \end{aligned}$$

一般地，

$$c_k = \sum_{j=0}^k a_j b_{k-j}$$

这里 $0 \leq k \leq m+n$ ，并且假定当 $i > n$ 时， $a_i = 0$ ；当 $j > m$ 时， $b_j = 0$ 。如果由关于 c_k 的公式来直接求解问题，每一个 c_k 可以用 $m+n$ 个处理器组并行计算，每一组同时需要 k 个处理器，执行时间为 $O(\log k)$ 。由于 k 的最大值为 $m+n$ ，上述方法共需要 $O((m+n)^2)$ 个处理器，执行时间为 $O(\log(m+n))$ 。

算法ST-POLY-MUL

输入： $A(0:n)$ and $B(0:m)$

输出: $C(0:m+n)$

BEGIN

1. Assign $A(n+1:n+m) = 0$
2. Assign $B(m+1:n+m) = 0$
3. For $k = 0$ to $m+n$ do in parallel
4. Evaluate $c_k = \sum_{j=0}^k a_j b_{k-j}$
5. End parallel

傅里叶变换可以有效地计算两个多项式的乘积, 仅需要处理器数为 $O((m+n)\log(m+n))$, 执行时间为 $O(\log(m+n))$ 。读者很容易证明用EREW PRAM模型可以成功地实现该算法。

多项式乘法有效算法 前面我们已经讨论了由 k 个元素组成的向量的有效傅里叶变换算法。

303 $a(x)$ 和 $b(x)$ 分别是 n 次和 m 次多项式, 分别用数组 $a[0:n]$ 和 $b[0:m]$ 来表示。设 N 是一个 2 的幂次方的整数, 且 $n+m < N \leq 2(n+m)$ 。假定 $a[n+1:N-1] = b[m+1:N-1] = 0$ 。下面我们计算向量 a 和 b 的傅里叶变换 $Y = Wa$ 和 $Z = Wb$ 。

通过观察可得到下面结果:

1. $y_i = a(\omega^i), 0 \leq i \leq N-1$;
2. $z_i = b(\omega^i), 0 \leq i \leq N-1$ 。

记 $u = (u_0, u_1, \dots, u_{N-1})^T$, 这里 $u_i = y_i z_i$ 。如果 $u = Wc$, 其中 $c = [c_0, c_1, \dots, c_{n+m-1}]$, 则 $u(\omega^i) = a(\omega^i)b(\omega^i)$ 。可以证明如果 c 被确定, 则 c 是多项式 a 与 b 的乘积, 且向量 c 的前 $m+n$ 个元素可以看作乘积多项式 $c(x)$ 的系数。

算法 EFFICIENT-POLY-MUL

输入: $a[0:n], b[0:m]$

输出: $c[0:m+n]$

1. Let N be the integer which is power of 2 such that $(m+n) < N \leq 2(m+n)$
2. Assign $a[n+1:N] = b[m+1:N] = 0$
3. Compute the Fourier transform of a , $Y = Wa$ using the efficient algorithm
4. Compute the Fourier transform of b , $Z = Wb$ using the efficient algorithm
5. For $i = 1$ to N do in parallel
6. $u_i = y_i z_i$
7. End parallel
8. Evaluate the inverse Fourier transform of u . i.e determine the vector c such that $u = Wc$
9. $c[0:m+n]$ is the coefficient vector of the product polynomial $c(x)$

END

计算傅里叶变换的有效并行算法共需处理器数为 $O(N \log N)$, 执行时间为 $O(\log N)$ 。同样,

304 计算傅里叶逆变换也需 $O(N \log N)$ 个处理器, 执行时间也为 $O(\log N)$ 。由于 N 的选取, 我们有如下结论:

如果 $a(x)$ 和 $b(x)$ 分别是 n 次和 m 次多项式, 它们的乘积多项式 $c(x) = a(x)b(x)$ 可用 $O((m+n)\log(m+n))$ 个处理器并行处理, 所用时间为 $O(\log(m+n))$ 。

读者可以证明EREW PRAM模型可有效地实现这个算法。

设 $a[0:n], b[0:m]$ 是两个向量。 a 与 b 的卷积记为 $a \times b$, 并定义为向量 $c[0:m+n]$, 这里

$c[0:m+n]$ 是 a 与 b 所表示的多项式的乘积，且 $c(m+n+1) = 0$ 。卷积有着广泛的应用。两个多项式卷积计算的复杂度与多项式乘法的复杂度相同。

10.6 矩阵求逆

高斯-约当(Gauss-Jordan) 消元法可以用来计算矩阵的逆。

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

考虑矩阵

$$A^{-1}A = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

将 A 代入，得：

$$A^{-1} \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

下面用高斯-约当法使等式左边的矩阵 A 变成单位阵，同样的行变换用于等式右端矩阵，得到

$$A^{-1} \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

305

即等式右端矩阵为 A^{-1} 。该算法包括以下几步：

1. 读入矩阵 A ；
2. 右端矩阵记为 B ，开始时令 B 为单位阵；
3. 对 A 作行变换并且对 B 作同样的行变换，最后将 A 化为单位阵。

将 A 通过行变换化为单位阵的算法类似于高斯消元法。相应的并行算法如下：

算法MATRIX-INVERSE

输入：矩阵 $A(1:n, 1:n)$

输出：矩阵 $A^{-1}(1:n, 1:n)$

BEGIN

1. Assign $A^{-1} = \text{identity matrix}$
2. For $k = 1$ to n
3. For $i = 1$ to n but $i \neq k$ do in parallel
4. do row operation for the matrices A and A^{-1}

$$R_i \leftarrow R_i - (a_{ik}/a_{kk})R_k$$

5. End parallel
6. Next k

END

在上述算法中，第1步可以用 $O(n^2)$ 个处理器来执行，所用的时间为 $O(1)$ 。第4步的行变换需用 $O(n)$ 个处理器，执行时间为 $O(1)$ ，因此第3~5步在单位时间内需用 $O(n^2)$ 个处理器来执行。总的复杂度为用 $O(n^3)$ 个处理器，执行时间为 $O(n)$ 。可以用CREW PRAM模型实现。

三角形矩阵的逆

考虑如下 n 阶下三角方阵，为方便起见，这里 n 是2的幂次。

306

$$\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

将矩阵分成四个子矩阵，每一个子矩阵的阶为 $(n/2) \times (n/2)$ 。

$$A = \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix}$$

由于 A 是非奇异的，因而 A_1, A_3 也是非奇异的。我们首先计算 A_1^{-1} 和 A_3^{-1} 。

$$\begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix} \begin{bmatrix} A_1^{-1} & 0 \\ -A_3^{-1}A_2A_1^{-1} & A_3^{-1} \end{bmatrix} = \begin{bmatrix} I_{n/2} & 0 \\ 0 & I_{n/2} \end{bmatrix} = I_n$$

因此有

$$A^{-1} = \begin{bmatrix} A_1^{-1} & 0 \\ -A_3^{-1}A_2A_1^{-1} & A_3^{-1} \end{bmatrix}$$

综上所述，求矩阵 A 的逆可分成以下几步：

1. 计算 A_1^{-1} 和 A_3^{-1} 的值；
2. 计算 $A_2A_1^{-1}$ ；

3. 计算 $-A_3^{-1}A_2A_1^{-1}$ 。

为了计算 A_1^{-1} 和 A_3^{-1} 的值，我们采用递归方式。在每一次递归调用时，矩阵的阶被一分为二，因而总的递归步是 $\log n$ 。在每一步中我们要计算两个较小矩阵的逆，也要计算如第2, 3步所示的两个较小矩阵的乘积。最著名的计算两个 $n \times n$ 矩阵乘积的并行算法需要 $O(n^3)$ 个处理器，所用的时间为 $O(\log n)$ ，且用CREW PRAM模型来实现。

如果用 $T(n)$ 表示求 n 阶矩阵的逆的并行算法的执行时间，则有

$$\begin{aligned} T(n) &= 2T(n/2) + 3O(\log n) \\ &= 2\{2T(n/4) + 3O(\log n)\} + 3O(\log n) \end{aligned}$$

根据这个递推式，可以求得 $T(n) = O(\log^2 n)$ 。因而上述并行算法需 $O(n^3)$ 个处理器，执行时间为 $O(\log^2 n)$ 。

算法Triangular-Matrix-Inverse

输入：下三角矩阵 $A(1:n, 1:n)$

输出： $A^{-1}(1:n, 1:n)$

[307]

1. Split A as $A = \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix}$
2. Evaluate recursively A^{-1} and A_3^{-1}
3. Evaluate $A_2A_1^{-1}$
4. Evaluate $-A_3^{-1}A_2A_1^{-1}$
5. $A^{-1} = \begin{bmatrix} A_1^{-1} & 0 \\ -A_3^{-1} & A_2A_1^{-1} & A_3^{-1} \end{bmatrix}$
6. return A^{-1}

10.7 Toeplitz矩阵

现在考虑另外一类有趣的矩阵，它们经常出现在信号处理的应用中，这类矩阵叫做Toeplitz矩阵。本节我们设计求三角形Toeplitz矩阵的逆的算法。考虑如下矩阵：

$$\begin{bmatrix} a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_1 & a_0 \\ a_n & a_{n-1} & a_{n-2} & \cdots & a_2 & a_1 \\ a_{n+1} & a_n & a_{n-1} & \cdots & a_3 & a_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{2n-2} & a_{2n-3} & a_{2n-4} & \cdots & a_n & a_{n-1} \end{bmatrix}$$

上面这个矩阵仅有 $(2n-2)$ 个不同的元素，矩阵满足如下条件：

$$A(i,j) = A(i+1, j+1) \quad i, j = 1, 2, \dots, n-1$$

所有的对角元素都相等。我们可以用一个 $2n-2$ 阶的向量来代替 $n \times n$ 的Toeplitz矩阵，此向量为

$$A = [a_0, a_1, a_2, \dots, a_{2n-2}]^T$$

以下为Toeplitz矩阵的一个例子：

29	13	8	9	5
15	29	13	8	9
1	15	29	13	8
-1	1	15	29	13
-11	-1	1	15	29

[308] 这是一个 5×5 的Toeplitz矩阵，可用向量 $[5, 9, 8, 13, 29, 15, 1, -1, 11]$ 来表示。Toeplitz矩阵的特殊结构能够用来设计有效的并行算法。考虑一般的矩阵 $A(0:n-1, 0:n-1)$ 和向量 $x = (x_0, x_1, \dots, x_{n-1})$ 。用直接矩阵乘法算法计算 Ax 如下：

算法 Matrix-Vector-Product

输入： $A(0:n-1, 0:n-1), x(0:n-1)$

输出：The product Ax denoted by the vector $y(0:n-1)$

1. For $i = 0$ to $n-1$ do in parallel

$$2. \quad y_i = \sum_{j=0}^{n-1} a_{ij} x_j$$

3. End parallel

上面的直接算法需 $O(n^2)$ 个处理器，执行时间为 $O(\log n)$ 。Toeplitz矩阵特殊的性质可以减少所需处理器的个数。假定 A 是一个 $n \times n$ 的Toeplitz矩阵，它也可以表示为一个向量，记矩阵 A 的向量表示为

$$a = [a_0, a_1, a_2, \dots, a_{2n-2}]$$

定理10-2 假定 $a = [a_0, a_1, a_2, \dots, a_{2n-2}]$ 是 n 阶Toeplitz方阵 A 的向量表示， $x = (x_0, x_1, \dots, x_{n-1})$ 是一向量。如果 a 和 x 的卷积是 $y = ax$ ，则有

$$\begin{bmatrix} y_{n-1} \\ y_n \\ \vdots \\ y_{2n-2} \end{bmatrix} = Ax$$

证明：在Toeplitz矩阵 A 中，第 k 行是 $a_{k+n-1} \cdots a_{k+1} a_k$ 。由矩阵乘积 $C = Ax$ ，得

$$c_k = a_{k+n-1} x_0 + a_{k+n-2} x_1 + \cdots + a_k x_{n-1}$$

根据卷积定义，有

$$y_{k+n-1} = a_{k+n-1} x_0 + a_{k+n-2} x_1 + \cdots + a_k x_k$$

证毕。

[309] 由上面定理知，Toeplitz矩阵与向量的乘积等于两个向量的卷积，因而，我们有如下定理。

定理10-3 $n \times n$ 阶Toeplitz矩阵与 n 维向量的乘积计算所需的处理器为 $O(n \log n)$ ，执行时间为 $O(\log n)$ 。

下三角Toeplitz矩阵

考虑下三角Toeplitz矩阵

$$\begin{bmatrix} a_0 & 0 & 0 & \cdots & 0 \\ a_1 & a_0 & 0 & \cdots & 0 \\ a_2 & a_1 & a_0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 \end{bmatrix}$$

Toeplitz矩阵可以用它的第一列 $[a_0, a_1, a_2, \dots, a_{n-1}]$ 唯一表示。我们利用分治技术求 A 的逆。将 A 分成四块：

$$\begin{bmatrix} A_1 & 0 \\ A_2 & A_1 \end{bmatrix}$$

上面这样分块是可能的，其中 A_1 是下三角Toeplitz矩阵， A_2 是Toeplitz矩阵。

利用下三角矩阵的逆的结论，有

$$A^{-1} = \begin{bmatrix} A_1^{-1} & 0 \\ -A_1^{-1}A_2A_1^{-1} & A_1^{-1} \end{bmatrix}$$

首先我们要递归地计算 A_1 的逆（也是一个下三角的Toeplitz矩阵）， A_1 是Toeplitz下三角矩阵，我们求出 A_1^{-1} 的第一列。求出 A_1^{-1} 后，再计算 $A_1^{-1}A_2A_1^{-1}$ 的第一列。这个过程表示如下。

假定 $e = (1, 0, 0, \dots, 0)^T$ 。首先计算 A_1^{-1} ，给出 A_1^{-1} 的第一列；然后计算 $A_2(A_1^{-1}e)$ ，这是Toeplitz矩阵和向量的乘积；然后计算 $A_1^{-1}(A_2(A_1^{-1}e))$ 。由结合性，我们有

$$A_1^{-1}A_2A_1^{-1}e = A_1^{-1}(A_2(A_1^{-1}e))$$

算法Tria-Toeplitz

输入： $A[0:n-1]$

输出： $A^{-1}[0:n-1]$

1. Divide the matrix A into the form $\begin{bmatrix} A_1 & 0 \\ A_2 & A_1 \end{bmatrix}$

2. Evaluate A_1^{-1} recursively

3. Let $e = [1, 0, 0, 0 \dots]^T$

4. Evaluate $(A_1^{-1}e)$

5. Evaluate $A_2(A_1^{-1}e)$

6. Evaluate $A_1^{-1}(A_2(A_1^{-1}e))$

7. $A^{-1}[0:n/2-1] = (A_1^{-1}e)$

8. $A^{-1}[n/2:n-1] = A_1^{-1}(A_2(A_1^{-1}e))$

END

310

在上面算法中，矩阵的阶在每一次递归调用中减半，因此共需执行 $O(\log n)$ 个递归步。第4、5、6步用来计算Toeplitz矩阵和一个向量的乘积，因而可以利用EREW PRAM 模型来实

现，共需处理器数为 $O(n \log n)$ ，执行时间为 $O(\log n)$ 。这样，共需 $O(\log n)$ 个递归步来计算下三角 Toeplitz 矩阵的逆，利用 EREW PRAM 模型共需处理器数为 $O(n \log n)$ ，执行时间为 $O(\log^2 n)$ 。

10.8 三对角方程组

在一些科学实践中，线性方程组通常具有下面的形式：

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= b_3 \\ a_{43}x_3 + a_{44}x_4 + a_{45}x_5 &= b_4 \\ &\dots \\ &\dots \\ a_{n-1,n-2}x_{n-2} + a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n &= b_{n-1} \\ a_{n,n-1}x_{n-1} + a_{nn}x_n &= b_n \end{aligned}$$

一般地，第 i 个方程有如下形式：

$$a_{i,i-1}x_{i-1} + a_{ii}x_i + a_{i,i+1}x_{i+1} = b_i,$$

这里 $1 < i < n$ 。当 $i = 1$ 或 $i = n$ 时，方程只有两项。下面我们来看一个 $n = 6$ 的例子：

$$\begin{aligned} 3x_1 + 2x_2 &= 1 \\ x_1 + 2x_2 + 4x_3 &= 3 \\ 12x_2 + 3x_3 + 2x_4 &= 8 \\ 7x_3 - 3x_4 + 12x_5 &= 2 \\ 2x_4 + 5x_5 + x_6 &= 4 \\ x_5 - x_6 &= 12 \end{aligned}$$

这里，方程组可以用矩阵表示为：

$$\left(\begin{array}{cccccc} 3 & 2 & 0 & 0 & 0 & 0 \\ 1 & 2 & 4 & 0 & 0 & 0 \\ 0 & 12 & 3 & 2 & 0 & 0 \\ 0 & 0 & 7 & -3 & 12 & 0 \\ 0 & 0 & 0 & 2 & 5 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{array} \right) \left(\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array} \right) = \left(\begin{array}{c} 1 \\ 3 \\ 8 \\ 2 \\ 4 \\ 12 \end{array} \right)$$

注意，系数矩阵的非零元素只出现在对角线及次对角线上，这样的矩阵称为三对角阵，这样的方程组称为三对角方程组。

10.8.1 高斯消元法

求一个三对角方程组的解，我们通常可以利用高斯消元法消去下三角元素。由于在每一列中只有一个下三角元素，因而只需消去这一个下三角元素，然后利用回代法求解。利用高斯消元法求三对角方程组的算法如下：

算法Tridiagonal-G

输入: $a(1:n, 1:n)$, $b(1:n)$

输出: $x(1:n)$

1. Use Gauss elimination method to reduce the matrix a into an upper triangular matrix
2. Use Back substitution method now to solve the system

复杂度分析 对于每一个行运算, 所需时间为 $O(n)$ 。算法第1步所需时间为 $O(n^2)$, 回代所需时间为 $O(n^2)$ 。因此在串行方式下该算法共需时间为 $O(n^2)$ 。利用CREW PRAM模型, 共需 $O(n)$ 个处理器, 执行时间为 $O(n)$ 。[312]

高斯消元法具有内在串行性, 即只有在完成第一列的行运算后才能进行第二列。以此类推, 此方法不可能同时进行各列的行运算, 因此不适合并行计算机。下面给出求解三对角线性方程组的另一种并行化方法。

10.8.2 奇偶约化法

奇偶约化法是将一个方程组分成两个可以分别并行求解的子方程组的方法。下面我们给出一个有16个未知数的三对角方程组的例子。假定未知数分别为 $x_1, x_2, x_3, \dots, x_{16}$ 。给定含16个未知数的16元方程组, 我们对系数矩阵作一些算术运算, 当这些算术运算完成后, 这个方程组被分成如下两个子方程组:

子方程组1:

未知数: $x_1, x_3, x_5, \dots, x_{15}$

方程组: 含8个未知数的8阶方程组;

子方程组2:

未知数: $x_2, x_4, x_6, \dots, x_{16}$

方程组: 含8个未知数的8阶方程组。

如果这些子方程组可以单独求解, 我们可以得到原来方程组的解 $x_1, x_2, x_3, \dots, x_{16}$ 。

为了求每一个子方程组的解, 我们利用相同策略, 即做某些算术运算使得子方程组1分成两个分别含有4个未知数的更小的方程组, 继续这个分化过程, 直到得到一个仅含有2个未知数的方程组, 并可在 $O(1)$ 内求解。[313]

下面我们更详细地描述计算过程。给定方程组

$$\begin{aligned}
 b_1x_1 + c_1x_2 &= r_1 \\
 a_2x_1 + b_2x_2 + c_2x_3 &= r_2 \\
 a_3x_2 + b_3x_3 + c_3x_4 &= r_3 \\
 a_4x_3 + b_4x_4 + c_4x_5 &= r_4 \\
 &\cdots \\
 &\cdots \\
 a_ix_{i-1} + b_ix_i + c_ix_{i+1} &= r_i \\
 &\cdots \\
 &\cdots \\
 a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n &= r_{n-1} \\
 a_nx_{n-1} + b_nx_n &= r_n
 \end{aligned}$$

可以表示成

$$\begin{aligned} b_1x_1 + c_1x_2 &= r_1 \\ a_1x_{i-1} + b_1x_i + c_1x_{i+1} &= r_i \\ (i = 2, 3, \dots, (n-1)) \\ a_nx_{n-1} + b_nx_n &= r_n \end{aligned}$$

如果我们设定哑元 $x_0 = 0, x_{n+1} = 0$, 则有:

$$a_ix_{i-1} + b_ix_i + c_ix_{i+1} = r_i,$$

这里 $1 \leq i \leq n$ 。

第 $i-1$ 个方程, 第 i 个方程, 第 $i+1$ 个方程分别为:

$$\begin{aligned} a_{i-1}x_{i-2} + b_{i-1}x_{i-1} + c_{i-1}x_i &= r_{i-1} \\ a_ix_{i-1} + b_ix_i + c_ix_{i+1} &= r_i \\ a_{i+1}x_i + b_{i+1}x_{i+1} + c_{i+1}x_{i+2} &= r_{i+1} \end{aligned}$$

314 由第 $i-1$ 个方程, 得:

$$x_{i-1} = f(x_i, x_{i-2})$$

这里, f 是 x_i 和 x_{i-2} 的函数。

类似地, 由第 $i+1$ 个方程得:

$$x_{i+1} = g(x_i, x_{i+2})$$

这里, g 是 x_i 和 x_{i+2} 的函数。

将 x_{i-1} 和 x_{i+1} 代入到第 i 个方程中, 可得到关于 x_{i-2}, x_i, x_{i+2} 的方程。

重置这些方程的系数为 a_i, b_i, c_i , 右边项为 r_i 。引入哑元 $x_{-1} = 0, x_{n+2} = 0$, 则有

$$a_ix_{i-2} + b_ix_i + c_ix_{i+2} = r_i, \quad 1 \leq i \leq n.$$

即

$$\begin{aligned} a_1x_{-1} + b_1x_1 + c_1x_3 &= r_1 \\ a_2x_0 + b_2x_2 + c_2x_4 &= r_2 \\ a_3x_1 + b_3x_3 + c_3x_5 &= r_3 \\ a_4x_2 + b_4x_4 + c_4x_6 &= r_4 \\ a_5x_3 + b_5x_5 + c_5x_7 &= r_5 \\ a_6x_4 + b_6x_6 + c_6x_8 &= r_6 \\ a_7x_5 + b_7x_7 + c_7x_9 &= r_7 \\ a_8x_6 + b_8x_8 + c_8x_{10} &= r_8 \end{aligned}$$

上面这个方程组被分成两个子方程组。方程 $1, 3, 5, \dots$ 是由 $x_1, x_3, x_5, \dots, x_9$ 组成的方程组, 即子方程组1。方程 $2, 4, 6, \dots$ 是由 $x_2, x_4, x_6, \dots, x_{10}$ 组成的方程组, 即子方程组2。如下所示:

子方程组1

$$a_1x_{-1} + b_1x_1 + c_1x_3 = r_1$$

$$a_3x_1 + b_3x_3 + c_3x_5 = r_3$$

子方程组2

$$a_2x_0 + b_2x_2 + c_2x_4 = r_2$$

$$a_4x_2 + b_4x_4 + c_4x_6 = r_4$$

$$\begin{array}{ll}
 a_5x_3 + b_5x_5 + c_5x_7 = r_5 & a_6x_4 + b_6x_6 + c_6x_8 = r_6 \\
 \cdots & \cdots \\
 \cdots & \cdots \\
 \cdots & \cdots
 \end{array}$$

例：运用奇偶约化法求下面线性方程组的解。

$$\begin{array}{ll}
 4x_1 + x_2 = 2 & (i) \\
 4x_1 + 11x_2 - 5x_3 = 7 & (ii) \\
 2x_2 + 14x_3 - 6x_4 = 13 & (iii) \\
 5x_3 + 18x_4 = 24 & (iv)
 \end{array}$$

315

这里有四个方程，四个未知量。对方程(i)，要用方程(ii)中的 x_2 的值来代替(i)中 x_2 的值。
方程(ii)为

$$\begin{aligned}
 4x_1 + 11x_2 - 5x_3 &= 7 \\
 x_2 &= \frac{1}{11}(7 - 4x_1 + 5x_3)
 \end{aligned}$$

代入方程(i)中，有

$$4x_1 + \frac{1}{11}(7 - 4x_1 + 5x_3) = 2$$

即

$$\frac{40}{11}x_1 + \frac{5}{11}x_3 = \frac{15}{11}$$

即

$$40x_1 + 5x_3 = 15$$

也可以写成

$$8x_1 + x_3 = 3 \quad (v)$$

这是方程(i)新的表示形式。

从方程(i)中解得 x_1 ，从方程(iii)中解得 x_3 ，再将 x_1 和 x_3 代入(ii)中，可得新方程(ii)。即

由(i) $x_1 = (2 - x_2)/4$

由(iii) $x_3 = (13 - 2x_2 + 6x_4)/14$

将 x_1 和 x_3 代入(ii)中，得新方程

$$\begin{aligned}
 (2 - x_2) + 11x_2 - \frac{5}{14}(13 - 2x_2 + 6x_4) &= 7 \\
 10x_2 - 2x_4 &= 9 \quad (vi)
 \end{aligned}$$

316

从方程(ii)中解得 x_2 ，从方程(iv)中解得 x_4 ，将 x_2 和 x_4 代入(iii)中，得：

$$-24x_1 + 547x_3 = 651 \quad (vii)$$

将方程 (iv) 中的 x_3 用 (iii) 中的 x_3 来代替, 得:

$$-10x_2 + 282x_4 = 271 \quad (\text{viii})$$

方程 (v) 和 (vii) 构成子方程组1, 方程 (vi) 和 (viii) 构成子方程组2。

子方程组1	子方程组2
$8x_1 + x_3 = 3$	$10x_2 - 2x_4 = 9$
$-24x_1 + 547x_3 = 651$	$-10x_2 + 282x_4 = 271$

上面每个方程组都含有两个未知数和两个方程, 因此可以立即得到各方程组的解。

子方程组1	子方程组2
$x_1 = \frac{9}{40}$	$x_2 = \frac{11}{10}$
$x_3 = \frac{6}{5}$	$x_4 = 1$

把这两组解合并, 得到原方程组的解。

$x_1 = \frac{9}{40}$	$x_2 = \frac{11}{10}$	$x_3 = \frac{6}{5}$	$x_4 = 1$
----------------------	-----------------------	---------------------	-----------

上面的方法适合并行实现。用 $O(n^2)$ 个处理器将方程组分成子方程组, 执行时间是常数时间。对子方程组继续上面的执行过程, 分成只含有2个未知数的若干个子方程组共需 $O(\log n)$ 步。因此总的并行执行时间是 $O(\log n)$, 处理器数为 $O(n^2)$, 且用CREW PRAM 模型来实现。

参考文献

317

- Ames, W. F. (1977) *Numerical Methods for Partial Differential Equations*, Academic Press, New York.
- Bini, D. (1984) Parallel Solution of Certain Toeplitz Linear Systems, *SIAM J. of Comput.*, 13(2), 268–276.
- Borodin, A. and Munro, I. (1975) *The Computational Complexity of Algebraic and Numeric Problems*, Elsevier, New York.
- Borodin, A. J. Von Zur Gathen and Hopcroft, J. H. (1975) *Fast Parallel Matrix and gcd Computation*, Elsevier, New York.
- Chin-Wen, H. and Lee, R. D. T. (1990) A Parallel Algorithm for Solving Sparse Triangular Systems, *IEEE Trans. on Computers*, 39.
- Codenotti, B. and Favati, (1987) Iterative Methods for the Parallel Solution of Linear Systems, *Comput. Math. Applic.*, 13(7), 631–633.
- Young, D. M. (1971) *Iterative Solution of Large Linear Systems*, Academic Press, New York and London.
- Hellar, D. (1978) A Survey of Parallel Algorithms in Numerical Linear Algebra, *SIAM Review*, 20(4), 740–777.
- Evans, D. J. (1982) *Parallel Processing Systems an Advanced Course*, Cambridge University Press, New York.
- Faddeeva, V. N. (1959) *Computational Methods of Linear Algebra*, Dover Publications, New York.
- Gallivan, K. A., Plemons, R. J., and Sameh, Parallel Algorithm for Dense Linear Algebra Computations, *SIAM Rev.*, 32, 54–135.

Miranker, W. L. (1971) A Survey of Parallelism in Numerical Analysis", *SIAM Review*, **13(4)**, 524–547.

Sameh, A. H. and Kuck, D. J. (1977) A Parallel QR Algorithm for Symmetric Tridiagonal Matrices, *IEEE Trans. on Computers*, **C-26**, 147–153.

Stone, H. (1973) An Efficient Parallel Algorithm for the Solution of Tridiagonal Linear System of Equations, *J. ACM*, **20**, 27–38.

习题

10.1 考虑矩阵递归方程

$$Y_1 = B_1;$$

$$Y_i = A_i Y_{i-1} + B_i \quad (2 \leq i \leq n)$$

这里 A_i , B_i 是给定的 $m \times m$ 矩阵。设计一个时间复杂度为 $O(\log n \log m)$ 的并行算法来比较所有的 Y_i 。

- 10.2 证明：两个 $n \times n$ 下三角 Toeplitz 矩阵相乘所得的矩阵仍是下三角 Toeplitz 矩阵。
- 10.3 设计一个时间复杂度为 $O(\log n)$ 的算法来计算两个下三角 Toeplitz 矩阵的乘积。[318]
- 10.4 两个任意的 Toeplitz 矩阵相乘仍是一个 Toeplitz 矩阵吗？设计一个算法来计算两个 Toeplitz 矩阵的乘积。

第11章 微分与积分

计算机起初是用于数值计算，在几乎所有的科学与工程学分支中，数值计算占主导位置。在本章中，我们学习一些重要的数值方法并设计相应有效的并行算法。首先，我们给出最基本但又非常重要的概念——微分。

11.1 微分

考虑实值函数 $f(x)$ ，在点 $x_0, x_1, x_2, \dots, x_n$ 处相应的函数值为 $f_0, f_1, f_2, \dots, f_n$ 。

x 值	x_0	x_1	x_2		x_{n-1}	x_n
f 值	f_0	f_1	f_2		f_{n-1}	f_n

考虑如下泰勒级数

$$f(x_i + h) = f(x_i) + \frac{h}{1} f'(x_i) + \frac{h^2}{2} f''(x_i) + \dots$$

$$f_{i+1} = f(x_i) + \frac{h}{1} f'(x_i) + \frac{h^2}{2} f''(x_i) + \dots$$

当 h 非常小时， h^2 近似为零，因而有

$$f_{i+1} = f_i + h f'_i$$

这里 f'_i 表示 $f'(x_i)$ ，有

$$f'_i = \frac{f_{i+1} - f_i}{h}$$

利用这个公式可并行求出 $f'_0, f'_1, f'_2, \dots, f'_{n-1}$ 。并行算法如下：

算法 Differentiation

输入： $x(0:n), f(0:n), h$

输出： $f'(0:n-1)$

BEGIN

1. For $i = 0$ to $n-1$ do in parallel
2. $f'_i = (f_{i+1} - f_i)/h$
3. End parallel

END

复杂度分析 上面的算法执行时间为 $O(1)$ ，处理器数为 $O(n)$ ，且为 CREW PRAM 模型。并发读操作是必要的，这是因为

处理器 P_i 计算 f'_i ；

处理器 P_{i-1} 计算 $f_{i-1}^{'}$, 并且用到 f_i 。

对于一阶导数, 还有许多其他公式, 这里给出常用的一个公式:

$$f_i^{'} = \frac{f_{i+1} - f_{i-1}}{2h}$$

对于函数的两阶、三阶或更高阶导数, 我们给出如下的一些公式:

$$\begin{aligned} f_i^{''} &= \frac{1}{h^2} \{f_{i-1} - 2f_i + f_{i+1}\} \\ f_i^{'''} &= \frac{1}{12h^2} \{11f_{i-1} - 20f_i + 6f_{i+1} + 4f_{i+2} - f_{i+3}\} \\ f_i^{'''} &= \frac{1}{h^3} \{-f_{i-1} + 3f_i - 3f_{i+1} + f_{i+2}\} \\ f_i^{'''} &= \frac{1}{2h^3} \{-f_{i-2} + 2f_{i-1} - 2f_{i+1} + f_{i+2}\} \\ f_i^{'''} &= \frac{1}{h^4} \{f_{i-2} - 4f_{i-1} + 6f_i - 4f_{i+1} + f_{i+2}\} \end{aligned}$$

[320]

利用上面这些导数公式, 很容易设计并行算法来计算高阶导数。

11.2 偏微分

假定 $U(x,y)$ 是关于 x 和 y 的函数, 要用到下面一些记号:

U_x : U 对 x 的偏导数;

U_y : U 对 y 的偏导数;

U_{xx} : U_x 对 x 的偏导数;

U_{yy} : U_y 对 y 的偏导数;

U_{xy} : U_y 对 x 的偏导数;

U_{yx} : U_x 对 y 的偏导数。

同样地, 我们可以定义记号 U_{yy} , U_{xx} 等。一般情况下, 求导数的常用公式为:

$$y_i^{'} = \frac{y_{i+1} - y_i}{h}$$

利用上面的公式来求 U_x , U_y :

$$U_x(x_i, y_i) = \frac{U(x_{i+1}, y_i) - U(x_i, y_i)}{h}$$

这里 $h = x_{i+1} - x_i$ 。

在上面公式中, 注意到 y_i 没有改变。同样地, U_y 表示为

$$\frac{U(x_i, y_{i+1}) - U(x_i, y_i)}{k}$$

这里 $k = y_{i+1} - y_i$ 。

我们还可以利用三点中心公式

$$y_i^{'} = \frac{y_{i+1} - y_{i-1}}{2h}$$

[321]

得到下面关于 U_x 和 U_y 的表达式

$$U_x(x_i, y_i) = \frac{U(x_{i+1}, y_i) - U(x_{i-1}, y_i)}{2h}$$

$$U_y(x_i, y_i) = \frac{U(x_i, y_{i+1}) - U(x_i, y_{i-1})}{2k}$$

为简单起见，上面四个公式可表示为：

$$U_{x,i,j} = \frac{U_{i+1,j} - U_{i-1,j}}{2h} \quad (11.1)$$

$$U_{y,i,j} = \frac{U_{i,j+1} - U_{i,j-1}}{2k} \quad (11.2)$$

$$U_{x,i,j} = \frac{U_{i+1,j} - U_{i-1,j}}{2h} \quad (11.3)$$

$$U_{y,i,j} = \frac{U_{i,j+1} - U_{i,j-1}}{2k} \quad (11.4)$$

在导出高阶偏导数公式之前，我们来观察二维空间内关键点的情况。

考虑点 (x_i, y_j) ，它的邻点如图11-1和图11-2所示。

[322]

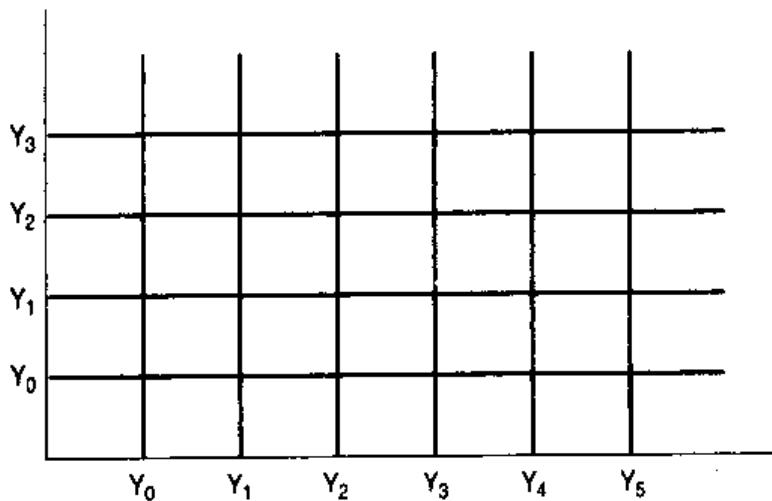


图11-1 二维网格

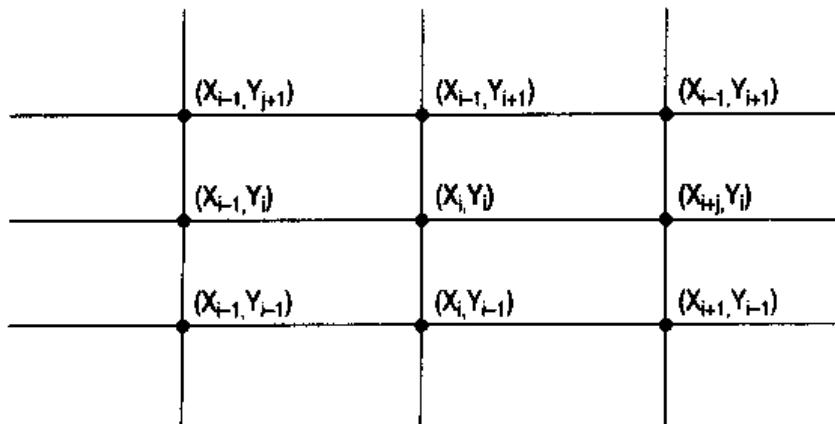


图11-2 网格中的点

下面我们推导 U_{xy} 的公式。由前面定义可知， U_{xy} 表示 U 对 x 的偏导数，应用公式(11.3)可得：

$$U_{xyj} = \frac{U_{i+1,j} - U_{i-1,j}}{2h}$$

再应用公式(11.4)，得：

$$\begin{aligned} U_{j+1,j} &= \frac{U_{i+1,j+1} - U_{i+1,j-1}}{2k} \\ U_{i-1,j} &= \frac{U_{i-1,j+1} - U_{i-1,j-1}}{2k} \end{aligned}$$

应用上面等式的结果，可得：

$$U_{xyj} = \frac{1}{4hk}(U_{i+1,j+1} + U_{i-1,j-1} - U_{i+1,j-1} - U_{i-1,j+1})$$

图11-3可以帮助理解和记忆这个公式。图中，右上角和左下角的值相加，减去另外两个角的值，再除以 $4hk$ 。类似地，我们可以导出如下两个公式：

$$\begin{aligned} U_{xxy} &= \frac{U_{i+1,j} - 2U_i + U_{i-1,j}}{h^2} \\ U_{yyj} &= \frac{U_{i,j+1} - 2U_i + U_{i,j-1}}{k^2} \end{aligned}$$

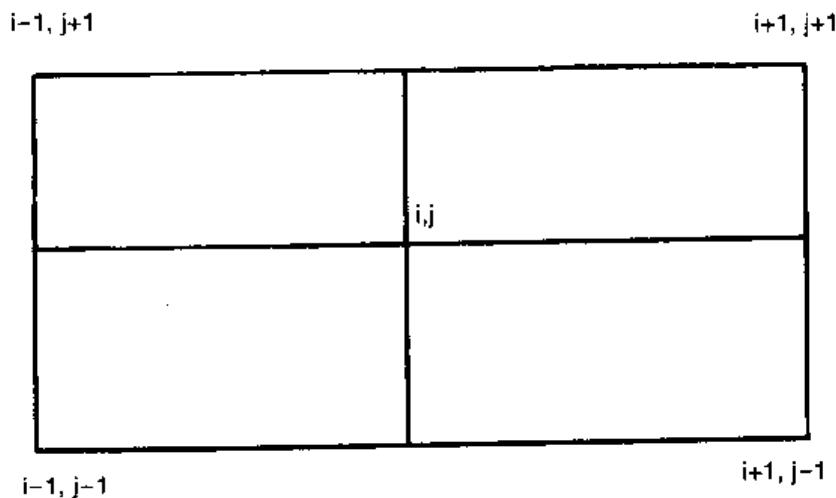


图11-3 U_{xy} 的计算图示

在许多问题中，常见到表达式 $U_{xx} + U_{yy}$ 。我们把它记为 $\nabla^2 U$ ，并称之为 U 的拉普拉斯算符， ∇^2 称为拉普拉斯算子(Laplacian operator)。

$$\nabla^2 U = U_{xx} + U_{yy}$$

利用 U_{xij} 和 U_{yyj} 的值，我们可以得出 $\nabla^2 U_j$ 的公式。选取 $h = k$ ，得：

$$\nabla^2 U_j = \frac{1}{h^2}(U_{i+1,j} + U_{i,j+1} + U_{i-1,j} + U_{i,j-1} - 4U_i) \quad (11.5)$$

对于下列给定的函数，我们可以求出 $\nabla^2 U$ 在(1.5, 2.0)处的值。

Y/X	1.0	1.5	2.0	2.5
1.0	12	15	15	16
1.5	13	14	16	19
2.0	21	23	27	35
2.5	31	34	37	38

$\nabla^2 U_{ij}$ 的公式图解见图11-4，图中四边的值相加再减去中心值的4倍。

$$\nabla^2 U_{ij} = \frac{1}{h^2} (U_{i+1,j} + U_{i,j+1} + U_{i-1,j} + U_{i,j-1} - 4U_{ij})$$

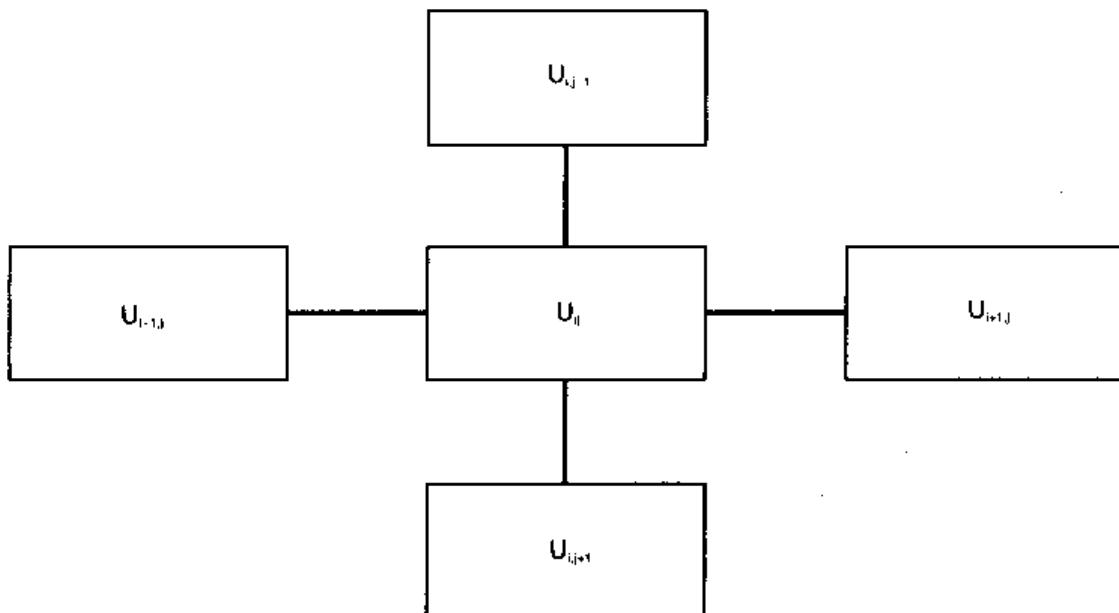


图11-4 拉普拉斯算子

$$\nabla^2 U_{22} = 15 + 23 + 13 + 16 - 4 \times 14 = 11$$

$$\nabla^2 U_{32} = 15 + 27 + 14 + 19 - 4 \times 16 = 11$$

我们可以设计并行算法来计算 U_{ij} ($1 \leq i \leq m, 1 \leq j \leq n$)。由上面公式可知，对所有的 i 和 j 均可实现并行计算。

算法Laplace

输入： $U[0:m, 0:n]$

输出： $\nabla^2 U[1:m-1, 1:n-1]$

1. For $i = 1$ to $m-1$ do in parallel
2. For $j = i$ to $n-1$ do in parallel
3. Calculate $\nabla^2 U_{ij}$, using the formula
4. End parallel
5. End parallel

[325]

上述算法共需处理器数为 $O(nm)$ ，执行时间为 $O(1)$ 。由公式知， $U_{i+1,j}$ 被用于计算 $\nabla^2 U_{ij}$ ，同时也要计算 $\nabla^2 U_{i+2,j}$ 。因此，需要用并发读PRAM模型，即CREW PRAM模型来实现。

11.3 定积分

考虑实值函数 $y = f(x)$, 定义域为 $[a,b]$ 。积分 $\int_a^b f(x)dx$ 表示函数 $f(x)$ 与 x 轴及 $x = a, x = b$ 所围区域的面积。

当 $(b-a)$ 非常小时, 这个面积近似等于上下底分别为 $f(a), f(b)$, 高为 $(b-a)$ 的梯形面积。因此有:

$$\text{area} = \frac{(b-a)(f(a)+f(b))}{2}$$

即

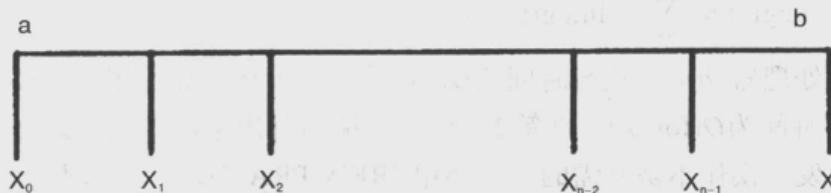
$$\int_a^{a+h} f(x)dx = \frac{h}{2}(f(a) + f(a+h))$$

这里的 h 非常小, 这个公式称为梯形法则。仅当 h 非常小时, 这个法则给出合理的值。如图11-6所示。

现在假定我们要计算积分 $\int_a^b f(x)dx$, 这里 $(b-a)$ 并不很小。我们将区间 $[a,b]$ 分成若干个相等的小子区间:

$$a = x_0 \leq x_1 \leq x_2 \leq \cdots \leq x_n = b$$

其中 $x_{i+1} - x_i = (b-a)/n = h$ ($0 \leq i \leq n$) (见图11-5)。要使 $h = (b-a)/n$ 足够小, n 必须取足够大。



[326]

图11-5 区间 $[a,b]$ 被分成 n 等分

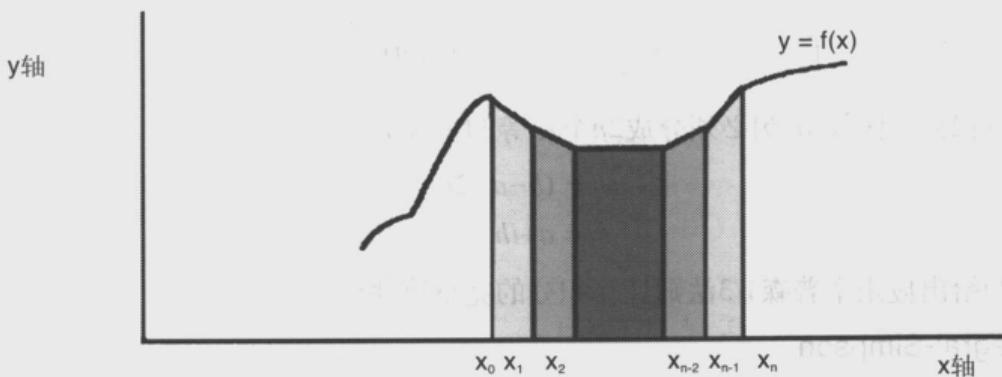


图11-6 梯形法则

$$\int_{x_i}^{x_{i+1}} f(x)dx = \frac{x_{i+1} - x_i}{2} (f(x_{i+1}) + f(x_i))$$

$$= \frac{h}{2} (f(x_i) + f(x_i + h))$$

$$\int_a^b f(x)dx = \int_{x_0}^{x_n} f(x)dx$$

$$\begin{aligned}
 &= \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} + \dots + \int_{x_{n-1}}^{x_n} f(x)dx \\
 &= \frac{h}{2} \{f(x_0) + f(x_1)\} + \frac{h}{2} \{f(x_1) + f(x_2)\} + \dots \\
 &\quad + \frac{h}{2} \{f(x_{n-1}) + f(x_n)\}
 \end{aligned}$$

如图11-6所示。完整的算法如下：

算法 Integral-Trapezoidal

输入：函数 $f(x)$, a, b, n

输出：Integral = $\int_a^b f(x)dx$

1. $h = (b-a)/n$
2. For $i = 0$ to $n-1$ do in parallel
3. $x_i = a + ih$
4. $\text{Integral}(i) = \frac{h}{2} \{f(x_i) + f(x_i + h)\}$
5. End parallel
6. Evaluate integral = $\sum_{i=0}^{n-1} \text{integral}(i)$

[327]

第2~5步共需处理器 $O(n)$, 执行时间为 $O(1)$ 。第6步求和运算占用了主要时间，所需处理器数为 $O(n)$, 执行时间为 $O(\log n)$ 。在第2~5步中 h 被同时用于所有的处理器，因此并发读操作是必需的，而并发写操作不是必需的。需要用CREW PRAM模型来实现。

为了求小子区间的积分值，我们给出一些近似公式。其中，辛普森1/3法则比较好。

辛普森1/3公式为：

$$\int_{x_i}^{x_i+2h} f(x)dx = \frac{h}{3} (f(x_i) + 4f(x_i + h) + f(x_i + 2h))$$

为了计算积分，区间 $[a, b]$ 必须分成 $2n$ 个相等的子区间。

$$h = (b-a)/2n$$

$$x_i = a + ih$$

下面我们给出应用辛普森1/3法则计算积分的完整算法：

算法 Integral-Simpson

输入：函数 $f(x)$, a, b, n (其中 n 为偶数)

输出：积分 $\int_a^b f(x)dx$

1. $h = (b-a)/2n$
2. For each $i \in \{0, 2, 4, 6, 8, \dots, (n-2)\}$ do in parallel
3. $x_i = a + ih$
4. $\text{Integral}(i) = h/3 \{f(x_i) + 4f(x_i + h) + f(x_i + 2h)\}$
5. End parallel
6. Evaluate Integral = $\sum \text{Integral}(i)$, where sum is taken over all $i \in \{0, 2, 4, \dots, (n-2)\}$

很容易判断出此算法所用的处理器数为 $O(n)$ ，总的执行时间为 $O(\log n)$ ，且用了CREW PRAM模型来实现。

11.4 插值

已知函数 $f(x)$ ，以及其在点 $x_0, x_1, x_2, \dots, x_n$ 处的函数值 $f(x_0), f(x_1), f(x_2), \dots, f(x_n)$ 。点 x 不同于 $x_0, x_1, x_2, \dots, x_n$ ，且 $f(x)$ 未知。328

由已知的函数值 $f(x_0), f(x_1), f(x_2), \dots, f(x_n)$ 来求 $f(x)$ 的近似值，这种方法称为插值。

时间(秒)	0	10	32	128
温度(℃)	15	20	30	50

上表表示不同时间以及相应的温度值。开始时温度为 15℃，当温度达到 20℃ 时，时间是 10 秒；当温度达到 30℃ 时，时间是 32 秒；当温度达到 50℃ 时，时间是 128 秒。我们感兴趣的是当时间为 100 秒时的温度，即当 $x = 100$ 时 $f(100)$ 的值。

x	x_0	x_1	x_2	x_3
(时间)	0	10	32	128
$f(x)$	15	20	30	50
(温度)	f_0	f_1	f_2	f_3

11.4.1 线性插值

设 x_0, x_1 是两个点， f_0, f_1 是相应的函数值， x 是介于 x_0 和 x_1 之间的任一点。我们要从 $f(x_0)$ 和 $f(x_1)$ 来求 $f(x)$ 。

考虑泰勒级数：

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x_1 - x_0) + \dots$$
329

仅考虑最前面两项，则：

$$f_1 = f_0 + f'(x_0)(x_1 - x_0)$$

因而有 $f'(x_0) = (f_1 - f_0)/(x_1 - x_0) + \dots$

x 点处的泰勒级数为：

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x_1 - x_0) + \dots$$

仅考虑最前面两项，则：

$$f(x) = f_0 + f'(x_0)(x - x_0) = f_0 + \frac{(f_1 - f_0)(x - x_0)}{(x_1 - x_0)}$$

设 $(x - x_0)/(x_1 - x_0) = p$ ，则有：

$$\begin{aligned} f(x) &= f_0 + (f_1 - f_0)p \\ &= (1 - p)f_0 + pf_1 \end{aligned}$$

上面的公式称为线性插值公式。 x 介于 x_0 与 x_1 之间， p 是一个非负分数，即 p 满足： $0 \leq p \leq 1$ 。

计算 给定 x_0, x_1, f_0, f_1 ，给定 $x, f(x)$ 未知。首先求 $p = (x-x_0)/(x_1-x_0)$ ，然后使用 $f(x) = (1-p)f_0 + pf_1$ ，求出 $f(x)$ 。例如，考虑如下函数：

x_i	7	19
f_i	15	35

假定求值 $f(10)$ 。

这里

$$x_0 = 7, \quad x_1 = 19$$

$$f_0 = 15, \quad f_1 = 35$$

$$x = 10$$

[330]

因此有

$$p = (x-x_0)/(x_1-x_0) = (10-7)/(19-7) = 0.25$$

$$1-p = 0.75$$

$$f(10) = (1-p)f_0 + pf_1 = 20$$

11.4.2 二次插值

设 x_{i-1}, x_i, x_{i+1} 是三个点且满足

$$x_i - x_{i-1} = x_{i+1} - x_i = h$$

f_{i-1}, f_i, f_{i+1} 是相应的函数值。 x 是介于 x_i 与 x_{i+1} 之间的点。

由泰勒级数

$$f(x) = f(x_i) + \frac{f'(x_i)}{1!}(x - x_i) + \frac{f''(x_i)}{2!}(x - x_i)^2 + \dots$$

只取前三项，并且用下面的式子来代替 $f(x_i)$ 和 $f'(x_i)$ 。

$$f'(x_i) = \frac{f_{i+1} - f_{i-1}}{2h}$$

$$f''(x_i) = \frac{1}{h^2}(f_{i-1} - 2f_i + f_{i+1})$$

得：

$$f(x) = \frac{-p(1-p)}{2}f_{i-1} + (1-p^2)f_i + \frac{p(1+p)}{2}f_{i+1}$$

这里 $p = (x-x_i)/h$ 。我们称其为二次插值公式。

11.4.3 拉格朗日插值

假定函数 $f(x)$ 在一些给定的点 x 上的值是已知的（区间不必相等）。

x 值	x_0	x_1	x_2	\dots	x_n
$f(x)$ 值	f_0	f_1	f_2	\dots	f_n

[331]

函数值在点 $x_0, x_1, x_2, \dots, x_n$ 处是已知的，我们设法构造一个关于 x 的 n 次多项式 $f(x)$ ，且满足 $f(x_i) = f_i$ (已知)。

考虑多项式

$$P_k(x) = \frac{(x - x_0)(x - x_1) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_0)(x_k - x_1) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}$$

注意

$$P_k(x_i) = \begin{cases} 0 & \text{if } i \neq k \\ 1 & \text{if } i = k \end{cases}$$

这里 $k = 0, 1, 2, \dots, n$ 。我们得到如下多项式：

$$\begin{aligned} f(x) &= f_0 P_0(x) + f_1 P_1(x) + \cdots + f_n P_n(x) \\ f(x_i) &= f_i (0 \leq i \leq n) \end{aligned}$$

这个公式称为拉格朗日插值公式。

算法设计 n 是插值点个数， f 在插值点处的值已知。

插值点： x_1, x_2, \dots, x_n

相应的 f 值： f_1, f_2, \dots, f_n

想要求函数 f 在 y 点的值 $f(y)$ 。在下面算法中用 f_y 表示 $f(y)$ 。拉格朗日插值算法如下：

算法Lagrange

输入： $x(1:n), f(1:n), y$

输出： f_y

1. For $k = 1$ to n
2. $P_k = 1$
3. For $i = 1$ to n do
4. If $i = k$ go to step 6
5. $P_k = P_k * (y - x_i) / (x_i - x_k)$
6. next i
7. next k
8. $f_y = 0$
9. For $i = 1$ to n do
10. $f_y = f_y + P_i f_i$
11. next i
12. END

[332]

拉格朗日插值并行算法 上面算法的第1~7步计算 $P_k(0 \leq k \leq n)$ 的值，且可以并行化。每个 P_k 值的计算需要 $O(\log n)$ 时间，处理器数为 $O(n)$ ，并用EREW PRAM 模型来实现。第9~11步计算 f_y 的值，也需要 $O(n)$ 个处理器，执行时间也是 $O(\log n)$ 。拉格朗日插值并行实现的总时间为 $O(\log n)$ ，处理器数为 $O(n^2)$ 。

参考文献

Atkinson, F. S. (1970) Numerical Methods That Work, Harper and Row, New York.

Andrew, L. (1985) Elementary Partial Differential Equations with Boundary Value

- Problems, Saunders College Publishing, Philadelphia.
- Chandy, K. M., and Taylor, S. (1992) An Introduction to Parallel Programming, Jones and Bartlette, Boston.
- Cooley, J. W., and Turkey, J. W. (1965) An Algorithm for the Machine Calculations of Complex Fourier Series, *Mathematics of Computations*, **19**, 297–301.
- P. J. Davis, and Rabinowitz, P. (1967) *Numerical Integration*, Blaisdell, Waltham, MA.
- Dongara, J., Duff, I., Sorensen, D., and Vander Vorst, H. (1991) *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia.
- Forsythe, G. E. and Moler, C. B. (1967) *Computer Solutions of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ.
- D. Kahaner, Moler, C., and Nash, S. (1989) *Numerical Methods and Software*, Prentice-Hall, Englewood Cliffs, NJ.
- Knuth, D. (1973) *Sorting and Searching*, Addison-Wesley, Reading, MA.
- O'Neill, M. A. (1988) Faster than Fast Fourier, *Byte*, **13**(4), 229–230.
- Pinsky, M. A. (1991) *Partial Differential Equations and Boundary Value Problems with Applications*, McGraw-Hill, New York.
- Wilkinson, J. H. (1963) *Rounding Errors in Algebraic Process*, Prentice-Hall, Englewood Cliffs, NJ.
- Wilkinson, J. H. (1965) *The Algebraic Eigenvalue Problem*, Oxford University Press, London.

习题

- 333 11.1 用Monte Carlo法设计一个算法，求从 a 到 b 的函数积分。
 11.2 用差分法设计一个求函数微分的算法。
 11.3 什么是牛顿插值？求其并行复杂度。
 11.4 设计一个并行过程，用把 (a,b) 分成 n 个区间的五点中心公式求从 a 到 b 的函数积分。
 11.5 设计一个并行过程，用差分法求函数的三阶导数。

第12章 微分方程

12.1 欧拉公式

考虑如下微分方程：

$$\frac{dy}{dx} = f(x, y)$$

初始条件为 $y(x_0) = y_0$ 。点 $x_0, x_1, x_2, \dots, x_n$ 满足 $x_{i+1} = x_i + h$ ，且 $y_i = y(x_i)$ 。由于初值给定，我们把这一问题称为初值问题。

由泰勒级数

$$y_{i+1} = y_i + \frac{h}{1!} y'_i + \frac{h^2}{2!} y''_i + \dots$$

当 h 非常小时， h^2 可以忽略不计，因而有

$$y_{i+1} = y_i + hy'_i$$

给定 $y' = f(x, y)$ ，由上式，可得：

$$y_{i+1} = y_i + hf_i$$

335

这里 f_i 表示 $f(x_i, y_i)$ 。

上式称为求解初值问题的欧拉公式。由于 y_0 给定，使用欧拉公式可求得 y_1 ，即 $y_1 = y_0 + hf_0$ ；再由 y_1 及欧拉公式可求得 y_2 ，即 $y_2 = y_1 + hf_1$ 。重复这个过程，我们依次可以求出 $y_3, y_4, y_5, \dots, y_n$ 。由以上讨论可知，这个方法是内在串行的，将其并行化是不可能的。同样，对于初值问题，存在许多算法其本身具有内在串行性。下面我们考虑偏微分方程。

12.2 偏微分方程

设 x, y 是两个变量，由函数 $U(x, y)$ 及 U 对 x, y 的偏导数组成的方程称为偏微分方程。例如：

$$U_x + U_y + x^2 \sin x U_y = e^{2x} \cos^2(xy)$$

是偏微分方程。偏微分方程中最高阶导数的阶数称为偏微分方程的阶。在研究中，二阶偏微分方程是最重要的一类偏微分方程。在许多物理和工程计算中经常遇到求解二阶偏微分方程的问题。

二阶偏微分方程的一般形式为：

$$AU_{xx} + BU_{xy} + CU_{yy} = F$$

这里 F 是 x, y, U, U_x, U_y 的函数。

根据 A, B, C 的值，上述方程可分为如下三种类型：

如果 $B^2 - 4AC < 0$ ，方程称为椭圆方程；

如果 $B^2 - 4AC = 0$ ，方程称为抛物方程；

如果 $B^2 - 4AC > 0$, 方程称为双曲方程。

一些标准的二阶偏微分方程如下表所示:

1	波动方程	$U_{yy} - a^2 U_{xx} = 0$	$A \approx -a^2, B = 0, C = 0,$ $B^2 - 4AC = 4a^2 > 0,$ 双曲型
2	热传导方程	$kU_{xx} = U_y,$	$A = k, B = 0, C = 0,$ $B^2 - 4AC = 0,$ 抛物型
3	拉普拉斯方程	$U_{yy} + U_{xx} = 0$	$A = 1, B = 0, C = 1,$ $B^2 - 4AC = -4 < 0,$ 椭圆型
4	泊松方程	$U_{yy} + U_{xx} = F(x, y)$	椭圆型

为了描述这个分类, 让我们把下面方程归类为抛物型, 椭圆型和双曲型。

$$1. 2f_{xx} - 4f_{xy} + f_{yy} = (1+x^2)(1-y^2)$$

$$2. (x^2/4)f_{xx} + \sqrt{1-y^2}f_{xy} = (1+x^2)f_x f_y, \quad x > 0, 0 < y < 1$$

$$3. (1-x^2)U_{xx} + 2yU_{xy} + U_{yy} = 0$$

首先, 考虑方程

$$2f_{xx} - 4f_{xy} + f_{yy} = (1+x^2)(1-y^2)$$

这里 $A = 2, B = -4, C = 1, B^2 - 4AC = 16 - 8 = 8 > 0$ 。因此它是双曲方程。

现在考虑第二个方程

$$(x^2/4)f_{xx} + \sqrt{1-y^2}f_{xy} = (1+x^2)f_x f_y$$

这里 A 和 B 不是常数而是 x, y 的函数, 其中 $A = x^2/4, B = \sqrt{1-y^2}, C = 1, B^2 - 4AC = 1-y^2-x^2$ 。

因此当 $x^2+y^2 > 1$, 方程是椭圆型的; 当 $x^2+y^2 = 1$, 方程是抛物型的; 当 $x^2+y^2 < 1$, 方程是双曲型的。

为了对这个方程有更好的认识, 我们考虑二维平面上的单位圆。

给定 $0 < y < 1$, 区域位于 y 轴和垂线 $y = 1$ 之间。在单位圆内部, 方程是双曲型的; 在单位圆外部, 方程是椭圆型的; 在单位圆上, 方程是抛物型的。

考虑第三个方程 $(1-x^2)U_{xx} + 2yU_{xy} + U_{yy} = 0$, 这里 $A = 1-x^2, B = 2y, C = 1, B^2 - 4AC = 4(x^2+y^2-1)$ 。当 $x^2+y^2-1 < 0$ 时, 方程是椭圆型的; 当 $x^2+y^2-1 = 0$ 时, 方程是抛物型的; 当 $x^2+y^2-1 > 0$ 时, 方程是双曲型的。

12.3 抛物方程

最常见的抛物方程是热传导方程, 形如

$$CU_{xx} = U_y$$

我们在矩形区域 $[a_1, a_2] \times [b_1, b_2]$ 内解上述方程。将 $[a_1, a_2]$ 分成 m 个相等的子区间, $[b_1, b_2]$ 分成

n 个相等的子区间。令

$$x_i = a_1 + ih, \quad y_j = b_1 + jk$$

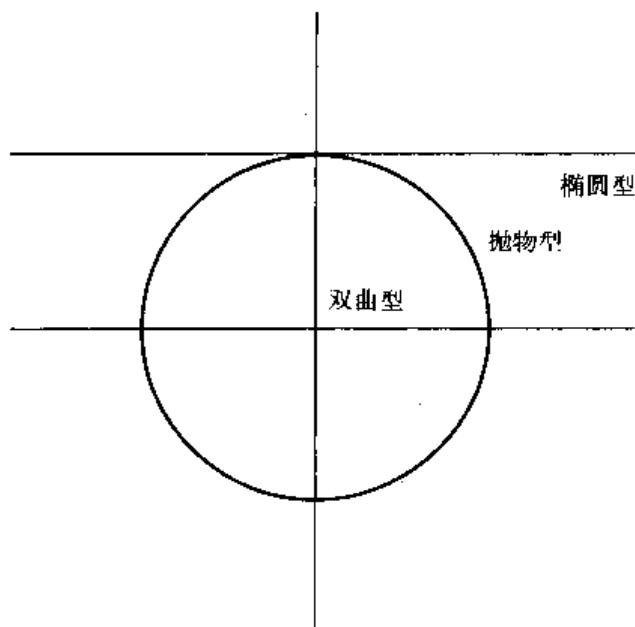


图12-1 区域及方程类型

这里

$$h = \frac{a_2 - a_1}{m}, \quad k = \frac{b_2 - b_1}{n}$$

如图12-2所示。为方便起见，记作

$$U_{ij} = U(x_i, y_j)$$

我们是要求 U_{ij} ，其中 $1 \leq i \leq m, 1 \leq j \leq n$ 。

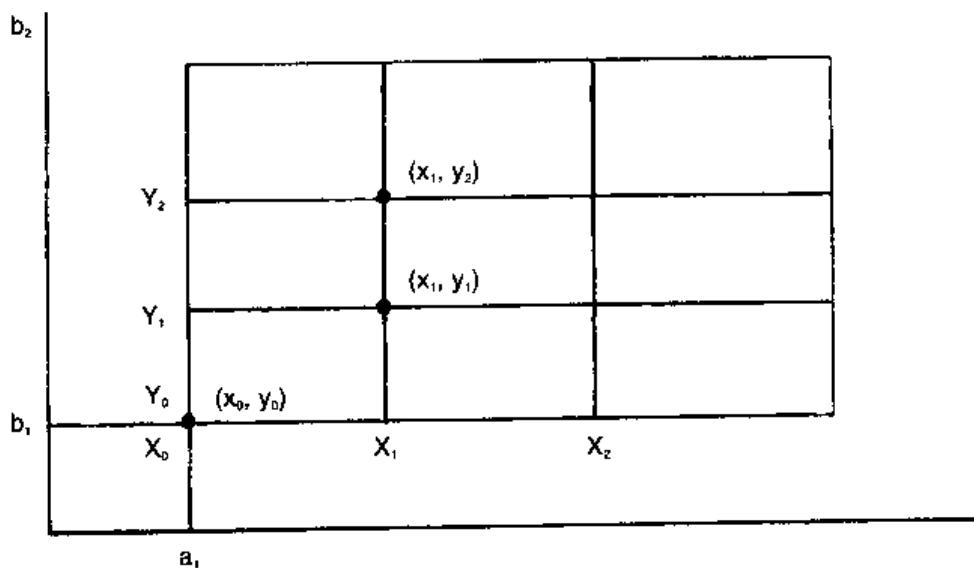


图12-2 二维网格

12.3.1 施密特法(求解抛物方程)

338 给定抛物方程

$$CU_{xx} = U_y$$

应用微分公式, 得:

$$U_{i,j+1} = (1 - 2r)U_j + r(U_{i-1,j} + U_{i+1,j})$$

其中 $r = ck/h^2$ 。上述表达式称为施密特公式。如图12-3所示。

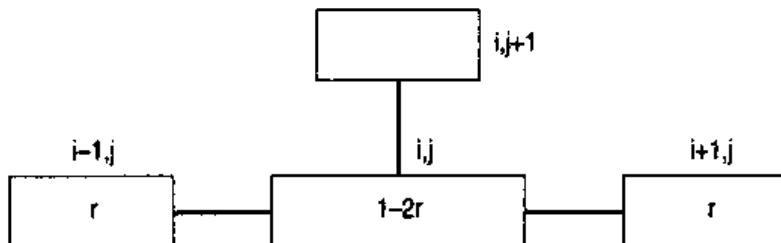


图12-3 施密特法

例 求解网格内点中的抛物方程 $U_y = 1/20U_{xx}$, 且有 $h = 0.2$, $k = 0.2$, $x \in [0,1]$, $y \in [0,1]$ 。

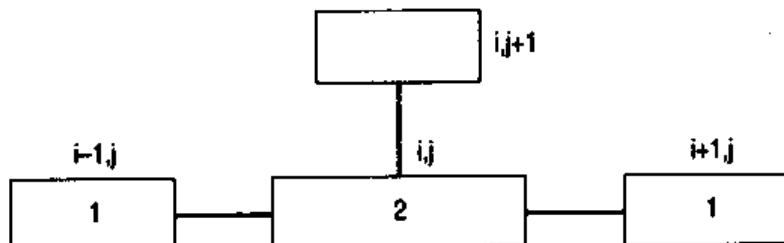


图12-4 施密特公式

$$U_y = 1/20U_{xx}$$

这里 $c = 1/20$, 初始条件为:

$$U(x,0) = 1+x^2$$

$$x_0 = 0, h = 0.2, y_0 = 0, k = 0.2$$

$$r = ck/h^2 = 1/4$$

施密特公式为

$$\begin{aligned} U_{i,j+1} &= (1 - 2r)U_j + r(U_{i-1,j} + U_{i+1,j}) \\ &= 1/2U_j + 1/4(U_{i-1,j} + U_{i+1,j}) \end{aligned}$$

即:

$$U_{i,j+1} = 1/4(2U_j + U_{i-1,j} + U_{i+1,j})$$

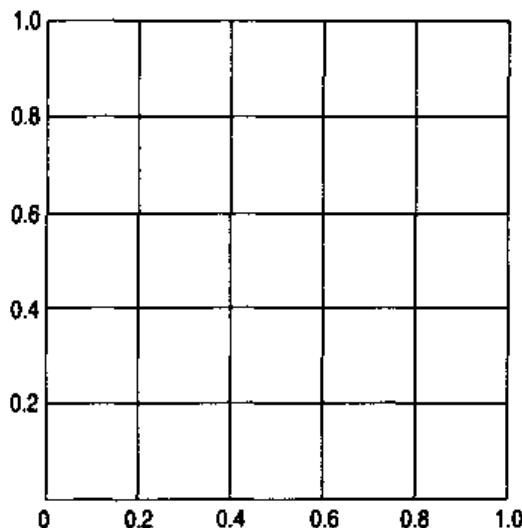
340 上述公式如图12-4所示。在 x 轴上取节点分别为 $x = 0, 0.2, 0.4, 0.6, 0.8, 1.0$ 。在 y 轴上取节点分别为 $y = 0, 0.2, 0.4, 0.6, 0.8, 1.0$ 。网格如图12-5所示。

使用给定的初始条件 $U(x,0) = 1 + x^2$, 我们可以计算 $y = 0$ 上所有点的值。

$$U_{0,0} = U(0,0) = 1 \quad U_{3,0} = U(0.6,0) = 1.36$$

$$U_{1,0} = U(0.2,0) = 1.04 \quad U_{4,0} = U(0.8,0) = 1.64$$

$$U_{2,0} = U(0.4,0) = 1.16 \quad U_{5,0} = U(1.0,0) = 2$$



[341]

图12-5 节点与网格

利用这些点的值，并结合施密特公式

$$U_{i,j+1} = 1/4[U_{i-1,j} + 2U_g + U_{i+1,j}]$$

可计算 $U_{1,1}, U_{2,1}, U_{3,1}, U_{4,1}$ 。

求 $U_{1,1}$ (见图12-6a):

$$\begin{aligned} U_{1,1} &= 1/4\{U_{0,0} + 2U_{1,0} + U_{2,0}\} \\ &= 1/4\{1 + 2(1.04) + 1.16\} \\ &= 1.06 \end{aligned}$$

求 $U_{2,1}$ (见图12-6b):

$$\begin{aligned} U_{2,1} &= 1/4\{U_{1,0} + 2U_{2,0} + U_{3,0}\} \\ &= 1/4\{1.04 + 2(1.16) + 1.36\} \\ &= 1.18 \end{aligned}$$

求 $U_{3,1}$ (见图12-6c):

$$\begin{aligned} U_{3,1} &= 1/4\{U_{2,0} + 2U_{3,0} + U_{4,0}\} \\ &= 1/4\{1.16 + 2(1.36) + 1.64\} \\ &= 1.38 \end{aligned}$$

求 $U_{4,1}$ (见图12-6d):

$$\begin{aligned} U_{4,1} &= 1/4\{U_{3,0} + 2U_{4,0} + U_{5,0}\} \\ &= 1/4\{1.16 + 2(1.36) + 1.64\} + 2 \\ &= 1.66 \end{aligned}$$

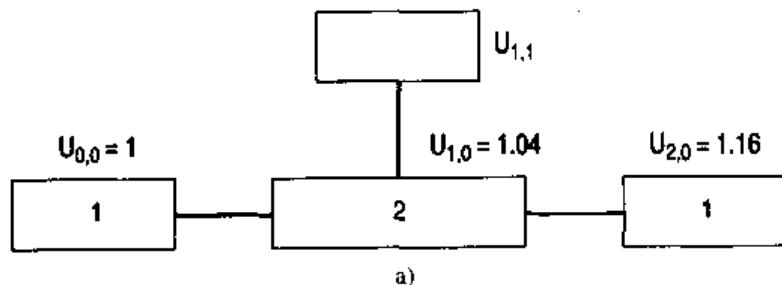


图 12-6
a) $U_{1,1}$ 的计算

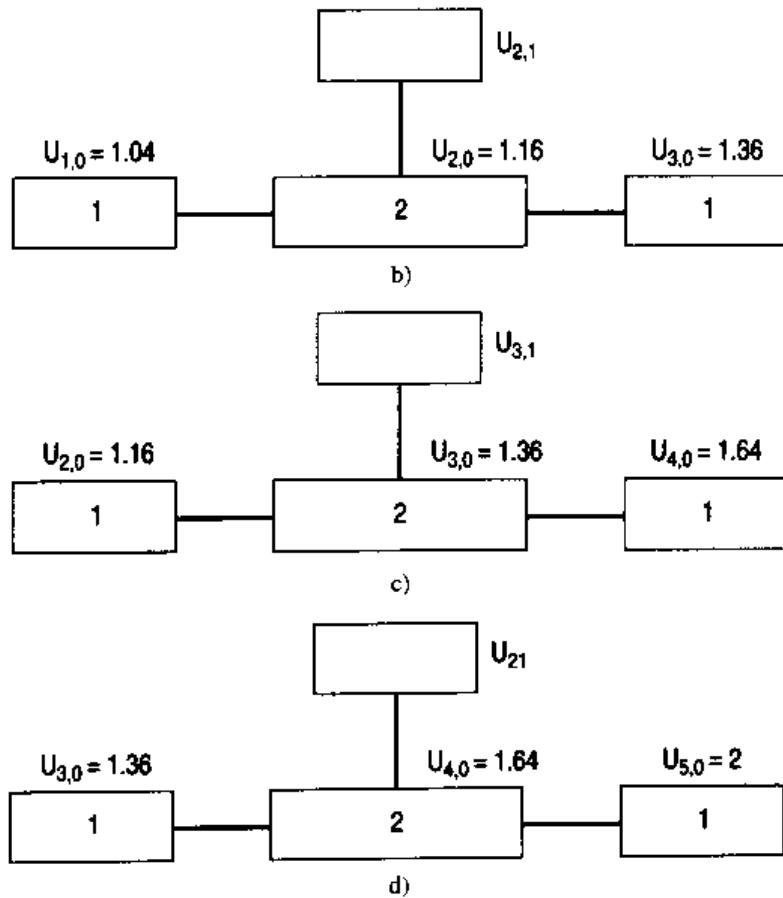


图12-6 (续)

b) $U_{i,1}$ 的计算 c) $U_{3,1}$ 的计算 d) $U_{4,0}$ 的计算求出 $U_{i,1}$ ($i = 1, 2, 3, 4$) 后，我们来求 $U_{i,2}$ ($i = 2, 3$)。求 $U_{2,2}$ (见图12-7a):

$$\begin{aligned} U_{2,2} &= 1/4 \{ U_{1,1} + 2U_{2,1} + U_{3,1} \} \\ &= 1/4 \{ 1.06 + 2(1.18) + 1.38 \} \\ &\approx 1.2 \end{aligned}$$

342

求 $U_{3,2}$ (见图12-7b):

$$\begin{aligned} U_{3,2} &= 1/4 \{ U_{2,1} + 2U_{3,1} + U_{4,1} \} \\ &= 1/4 \{ 1.18 + 2(1.38) + 1.66 \} \\ &\approx 1.4 \end{aligned}$$

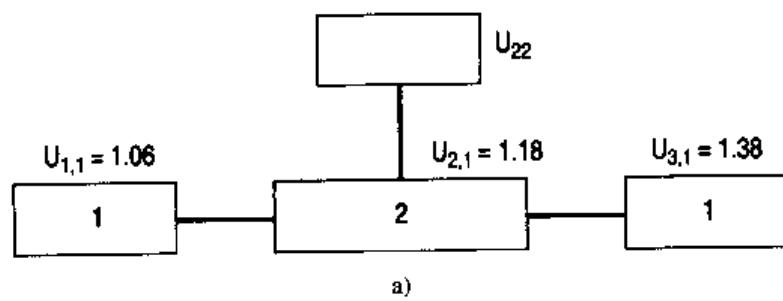


图 12-7

a) $U_{i,2}$ 的计算

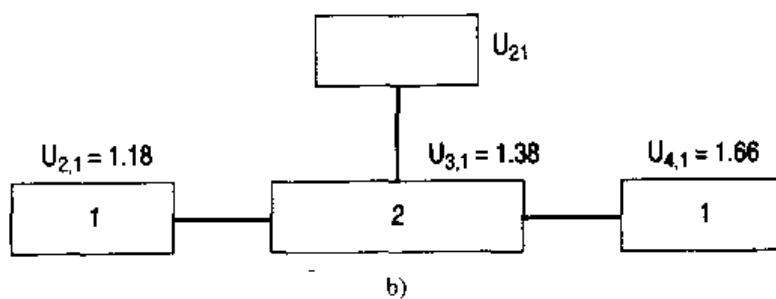


图12-7 (续)

b) $U_{3,2}$ 的计算

下面给出求解热传导方程的算法:

$$\begin{aligned} CU_y &= U_{xx} \\ y(x, b_1) &= f(x) \\ U(a_1, y) &= g_1(y) \\ U(a_2, y) &= g_2(y) \end{aligned}$$

算法Heat Flow

1. Define $f(x)$, $g_1(y)$, $g_2(y)$
2. Read a_1 , a_2 , b_1 , b_2 , c , m , n
3. $h = (a_2 - a_1)/m$, $k = (b_2 - b_1)/n$
4. $r = ck/h^2$
5. $x_0 = a_1$, $y_0 = b_1$, $U_0 = f(x_0)$

求初值:

6. For $i = 1$ to m
7. $x_i = x_0 + ih$
8. $U_{i,0} = f(x_i)$
9. next i
10. For $j = 1$ to n
11. $y_j = y_0 + jk$
12. $U_{0,j} = g_1(y_j)$
13. $U_{m,j} = g_2(y_j)$
14. next j

343

启用施密特公式:

15. For $j = 1$ to n
16. For $i = 0$ to $m-2$
17. $U_{i+1,j} = (1-2r)U_{i,j} + r(U_{i-1,j} + U_{i+1,j})$
18. next i
19. next j

施密特法的并行实现

上述串行算法中, 第6步到第9步的for循环可以并行化如下:

6. For $i = 1$ to m do in parallel
7. $x_i = x_0 + ih$
8. $U_{i,0} = f(x_i)$
9. End parallel

同样，第10步到第14步也可以并行化如下：

```

10. For  $j = 1$  to  $n$  do in parallel
11.  $y_j = y_0 + jk$ 
12.  $U_{0,j} = g_1(y_j)$ 
13.  $U_{m,j} = g_2(y_j)$ 
14. End parallel
END

```

施密特公式应用是从第15步开始的。由于所有的*i*可并行化，因而第17步可并行计算。第16步到第18步可写成：

```

16. For  $i = 0$  to  $m-2$  in Parallel
17.  $U_{i+1,j} = (1-2r)U_{i,j} + r(U_{i-1,j} + U_{i+1,j})$ 
18. End parallel

```

在施密特法中，求第*j*+1层的*U*值只用到第*j*层的*U*值，即*U*值是一个接一个串行计算的，因此第15步的for循环不可能并行化。

在上面的讨论中，第6~9步和第10~14步分别用的处理器数为*O(m)*和*O(n)*，执行时间均为*O(1)*。
[344] 第15~20步所用的处理器数为*O(m)*，执行时间为*O(n)*。

12.3.2 Laasonen法(求解抛物方程)

Laasonen法对*U*_{*y*}用如下公式：

$$U_{y,t,j} = \frac{U_{t,j} - U_{t,j-1}}{k}$$

对给定的抛物方程（热传导方程），使用这个公式，可得：

$$r(U_{t-1,j} + U_{t+1,j}) - (1+2r)U_{t,j} = -U_{t,j-1}$$

这里 $r = ck/h^2$

这个公式称为Laasonen法。如果对所有的*i*值，*U*_{*t*,*j*-1}已知，则用上面的公式可得到关于*U*_{*t*,*j*}，*U*_{*t*,*j*+1},…的线性代数方程组。通过求解方程组，可得到*U*_{*t*,*j*}，*U*_{*t*,*j*+1},…的值。

例 求解抛物方程

$$\begin{aligned} 20U_y &= U_{xx}U(x,0) = 1+x^2, & 0 < x < 0.6 \\ U(0,y) &= 1(y \geq 0), & U(0.6,y) = 1.36(y \geq 0) \\ h &= 0.2, & k = 0.2 \end{aligned}$$

给定的方程为

$$U_y = \frac{1}{20}U_{xx}$$

[345]

这里

$$\begin{aligned} c &= \frac{1}{20}, \quad k = 0.2, \quad h = 0.2 \\ r &= \frac{ck}{h^2} = \frac{0.2}{20(0.04)} = \frac{1}{4} \end{aligned}$$

因此, Laasonen公式变为

$$\frac{1}{4}(U_{i-1,j} + U_{i+1,j}) - \frac{3}{2}U_{ij} = -U_{i,j-1}$$

即

$$U_{i-1,j} - 6U_{ij} + U_{i+1,j} = -4U_{i,j-1}$$

考虑图12-8所示的网格点, 沿着 $x=0$ 和 $x=1$ 上的网格点的值全都为零(给定的)。沿着 x 轴上的网格点的值可由函数 $U(x,0) = 1 + x^2$ 计算得到。

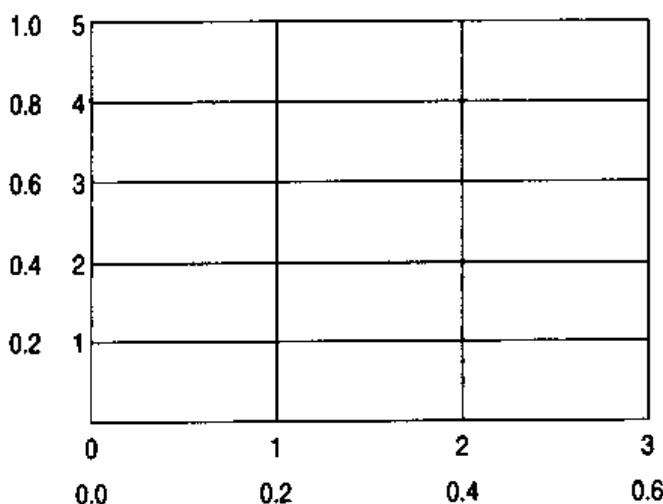


图12-8 网格点

$$U_{1,0} = U(0.2,0) = 1.04$$

$$U_{2,0} = U(0.4,0) = 1.16$$

$$U_{3,0} = U(0.6,0) = 1.36$$

还有

$$U_{0,j} = 1, j = 0, 1, 2, \dots$$

$$U_{3,j} = 1.36, j = 0, 1, 2, \dots$$

我们已求得网格中左侧、右侧和下边界的 U 值。在上面的公式中, 取 $i = 1, j = 1$, 则可得(见图12-9):

$$U_{0,1} - 6U_{1,1} + U_{2,1} = -4U_{1,0}$$

346

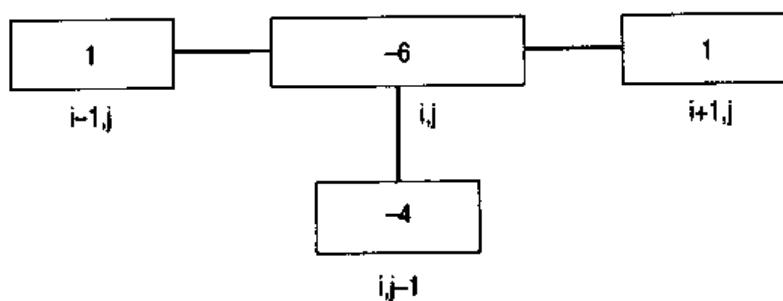


图12-9 Laasonen公式

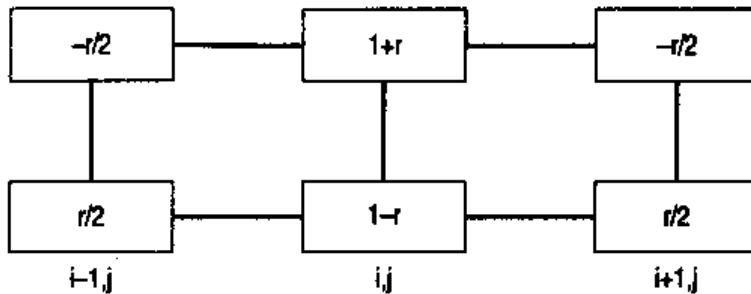


图12-10 Crank-Nicolson法

在上面的公式中，取*i*=2, *j*=1，则可得：

$$U_{1,1} - 6U_{2,1} + U_{3,1} = -4U_{2,0}$$

又有 $U_{0,1} = 1, U_{3,1} = 1.36, U_{1,0} = 1.04, U_{2,0} = 1.16$, 则上面两个方程变为：

$$-6U_{1,1} + U_{2,1} = 5.16$$

$$U_{1,1} - 6U_{2,1} = -6$$

求解并可得： $U_{1,1} = -1.056, U_{2,1} = 1.176$ 。

同样，取(*i,j*)=(1,2), (*i,j*)=(2,2), 可得到关于 $U_{1,2}$ 和 $U_{2,2}$ 的线性代数方程组。通过求解可得 $U_{1,2}$ 和 $U_{2,2}$ 的值。重复以上过程可求得 $U_{1,j}$ 和 $U_{2,j}$ (*j*>2)的值。

12.3.3 Crank-Nicolson法

根据施密特法，我们有 $r(U_{i+1,j} - 2U_{i,j} + U_{i-1,j}) = U_{i,j+1} - U_{i,j}$ 。

根据Laasonen法，如果用点(*i,j*+1)来代替(*i,j*)，则有：

$$r(U_{i+1,j+1} - 2U_{i,j+1} + U_{i-1,j+1}) = U_{i,j+1} - U_{i,j}$$

上述方程右边的值等于两个方程左边值的平均值，则有：

$$\frac{r}{2}(U_{i+1,j} - 2U_{i,j} + U_{i-1,j}) + \frac{r}{2}(U_{i+1,j+1} - 2U_{i,j+1} + U_{i-1,j+1}) = (U_{i,j+1} - U_{i,j})$$

和

$$\frac{r}{2}(U_{i-1,j} + U_{i+1,j}) + (1-r)U_{i,j} = \frac{r}{2}(U_{i-1,j+1} + U_{i+1,j+1}) + (1+r)U_{i,j+1}$$

上述方法叫做Crank-Nicolson法。用这个方程可以组成一个线性代数方程组。

在给定的热传导方程中，使用公式

$$U_{i,j} = \frac{U_{i,j+1} - U_{i,j-1}}{k}$$

可以推导出施密特法。使用公式

$$U_{i,j} = \frac{U_{i,j} - U_{i,j-1}}{k}$$

可以得到Laasonen公式。利用平均值技术，可以得到Crank-Nicolson法。我们可以由 $U_{i,j}, U_{i-1,j}, U_{i+1,j}$ 等来计算 $U_{i,j+1}$ 。 $U_{i,j}$ (*i*=0,1,2)位于第*j*层，利用这三种方法和第*j*层上的值，可以求出第*j*+1层上的值。因此这三种方法都是两层的。施密特法给出 $U_{i,j+1}$ 的显式公式，其他两种方法给出在

第 $j+1$ 层上的 U 的隐式关系式。通过这些关系式，我们可以构造线性代数方程组并求出在第 $j+1$ 层上 U 的值。因此，施密特法称为显式方法，而Laasonen和Crank-Nicholson法称为隐式方法。

12.3.4 三层差分法

考虑三点微分公式

$$U_{j,j,j} = \frac{U_{i,j+1} - U_{i,j} - 1}{2k}$$

对于抛物方程 $CU_y = U_\infty$ ，使用这个公式，有

$$\frac{2ck}{h^2} \{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}\} = U_{i,j+1} - U_{i,j-1}$$

348

这个公式称为Richardson公式。在这个公式中要计算第 $j+1$ 层上点的值，就要用到第 j 层和第 $j-1$ 层的值。因而Richardson公式是一个三层差分法。

考虑Richardson公式

$$U_{i,j+1} = U_{i,j-1} + 2r\{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}\}$$

我们用如下 $U_{i,j}$ 的表达式来代替等式右边 $U_{i,j}$ 的值。

$$U_{i,j} = \frac{U_{i,j-1} + U_{i,j+1}}{2}$$

得到：

$$\begin{aligned} U_{i,j+1} &= U_{i,j-1} + 2r\{U_{i-1,j} - U_{i,j-1} - U_{i,j+1} + U_{i+1,j}\} \\ (1+2)U_{i,j+1} &= (1-2)U_{i,j} + 2r\{U_{i-1,j} + U_{i+1,j}\} \\ U_{i,j+1} &= \frac{1-2r}{1+2r} U_{i,j-1} + \frac{2r}{1+2r} \{U_{i-1,j} + U_{i+1,j}\} \end{aligned}$$

这个公式称为Dufort-Frankel法。注意Richardson 和Dufort-Frankel法都是显式方法。

下面我们设计一个算法来求解热传导方程

$$CU_y = U_\infty$$

用Dufort-Frankel法求解，假定初始条件已经给出：

$$U(x, b_1) = f(x)$$

$$U(a_1, y) = g_1(y)$$

$$U(a_2, y) = g_2(y)$$

由于Dufort-Frankel法是三层差分法，在计算 $U_{i,2}$ ($i = 1, 2, \dots$)之前，我们需要知道 $U_{i,0}$, $U_{i,1}$ 的值。用施密特法求 $U_{i,0}$ 的值。算法如下：

算法Dufort-Frankel

1. define $f(x)$
2. define $g_1(x)$
3. define $g_2(x)$
4. read $a_1, a_2, b_1, b_2, m, n, c$

5. $h = (a_2 - a_1)/m, k = (b_2 - b_1)/n$
 6. $r = ck/h^2$
 7. $x_0 = a_1; y_0 = b_1; U_{m,0} = g_2(y_0); U_{0,0} = f(x_0)$

/*计算 $U_{i,0}$ 的初值*/

8. For $i = 1$ to m
 9. $x_i = x_0 + ih$
 10. $U_{i,0} = f(x_i)$
 11. next i

/*根据已知条件计算 $U_{0,j}$ 和 $U_{m,j}$ */

12. For $j = 1$ to n
 13. $y_j = y_0 + jk$
 14. $U_{0,j} = g_1(y_j)$
 15. $U_{m,j} = g_2(y_j)$
 16. next j

用施密特法计算 $U_{i,j}$:

17. For $i = 1$ to $m-1$
 18. $U_{i,1} = (1-2r)U_{i,0} + r(U_{i-1,0} + U_{i+1,0})$
 19. next i

启用Dufort-Frankel法:

20. For $j = 1$ to $n-1$
 21. For $i = 1$ to $m-1$
 22. $U_{i,j+1} = ((1-2r))U_{i,j-1} + (2r/(1+2r))\{U_{i-1,j} + U_{i+1,j}\}$
 23. Next i
 24. Next j

END

Dufort-Frankel法的并行化 在串行算法中, 第8~11步可以并行化, 第12~16步也可以并行化, 因而第8~16步可以写为:

8. For $j = 1$ to n do in parallel
 9. $x_i = x_0 + ih$
 10. $U_{i,0} = f(x_i)$
 11. End parallel
 12. For $j = 1$ to n do in parallel
 13. $y_j = y_0 + jk$
 14. $U_{0,j} = g_1(y_j)$
 15. $U_{m,j} = g_2(y_j)$
 16. End parallel

350 第17~19步用施密特法计算 $U_{i,j}$ 的值, 也可以将其并行化为:

17. For $i = 1$ to $m-1$ do in parallel

$$18. U_{i,j+1} = (1-2r)U_{i,j} + r\{U_{i-1,j} + U_{i+1,j}\}$$

19. End parallel

Dufort-Frankel法利用 U 在第 j 层和第 $j-1$ 层的值来计算 $U_{i,j+1}$ 。第22步也可以并行化。但第20步的for语句不能转换成并行语句。第20~25步可以重新写为：

20. For $j = 1$ to $n-1$

21. For $i = 1$ to $m-1$ do in parallel

$$22. U_{i,j+1} = ((1-2r)/(1+2r))U_{i,j-1} + (2r/(1+2r))\{U_{i-1,j} + U_{i+1,j}\}$$

23. End parallel

24. Next j

25. Stop

END

在并行操作中，第8~11步、第12~16步、第17~19步所用的处理器数分别为 $O(m)$ 、 $O(n)$ 、 $O(m)$ ，执行时间为 $O(1)$ 。第20~26步所用的处理器数为 $O(m)$ ，执行时间为 $O(n)$ 。第22步利用 $U_{i,j-1}$ 来计算 $U_{i,j+1}$ 。由于这一步是并行操作，要用CREW PRAM模型来实现，因此这一算法采用CREW PRAM模型。

参考文献

- Action, F. S. (1970) *Numerical Methods That Work*, Harper and Row, New York.
- Andrew, L. (1985) *Elementary Partial Differential Equations with Boundary Value Problems*, Saunders College Publishing, Philadelphia.
- Chandy, K. M., and Taylor, S. (1992) *An Introduction to Parallel Programming*, Jones and Bartlett, Boston.
- Cooley, J. W., and Turkey, J. W. (1965) An Algorithm for the Machine Calculations of Complex Fourier Series, *Mathematics of Computations*, 19:297–301.
- Davis, P. J. and Rabinowitz, P. (1967) *Numerical Integration*, Blaisdell, Waltham, MA.
- Dongara, J., Duff, I., Sorensen, D. and Vander Vorst, H. (1991) *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia.
- Forsythe, G. E. and Moler, C. B. (1967) *Computer Solutions of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ.
- Kahaner, D., Moler, C. and Nash, S. (1989) *Numerical Methods and Software*, Prentice-Hall, Englewood Cliffs, NJ.
- Knuth, D. (1973) *Sorting and Searching*, Addison-Wesley, Reading, MA.
- O'Neill, M. A. (1988) Faster than Fast Fourier, *Byte*, 13(4):292–230.
- Pinsky, M. A. (1991) *Partial Differential Equations and Boundary Value Problems with Applications*, McGraw-Hill, New York.
- Wilkinson, J. H. (1963) *Rounding Errors in Algebraic Process*, Prentice-Hall, Englewood Cliffs, NJ.
- Wilkinson, J. H. (1965) *The Algebraic EigenValue Problem*, Oxford University Press, London.

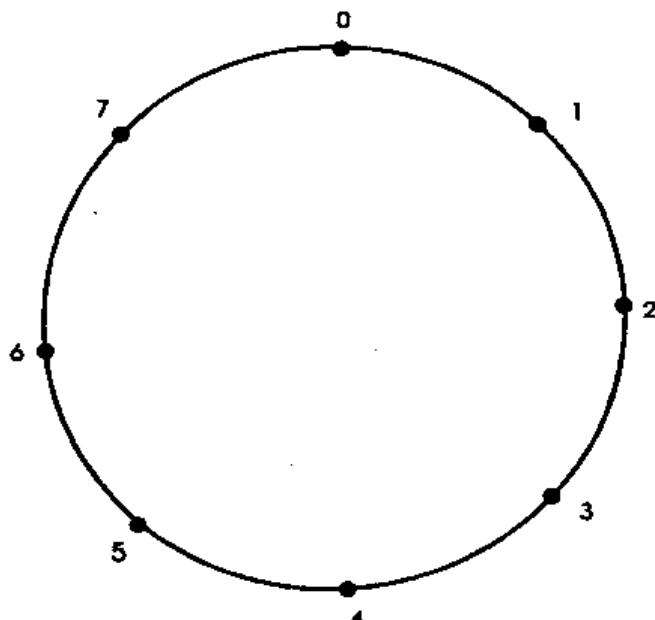
部分习题解答

第1章

1.7 环状网

如果把线性数组网络的两端连接起来，就得到如图A-1所示的环状网。

对于一个有 n 个节点的环，其直径为 $n/2$ ，且是度为2的正则图。



图A-1 环状网

[352]

1.8 弦环网络

在环状网中增加额外的连接，使得图变成度为3,4,⋯的正则图，就得到弦环网络。例如，一个度为3的弦环网络在图A-2中给出。

在这个例子中，有两个节点标记为 $0, 1, 2, 3, \dots, (2^4-1)$ 。每个奇数节点*i*都和一个偶数节点*j*连接，其中：

$$j = i+3(\bmod 2^4)$$

例如：

1和4连接；

3和6连接；

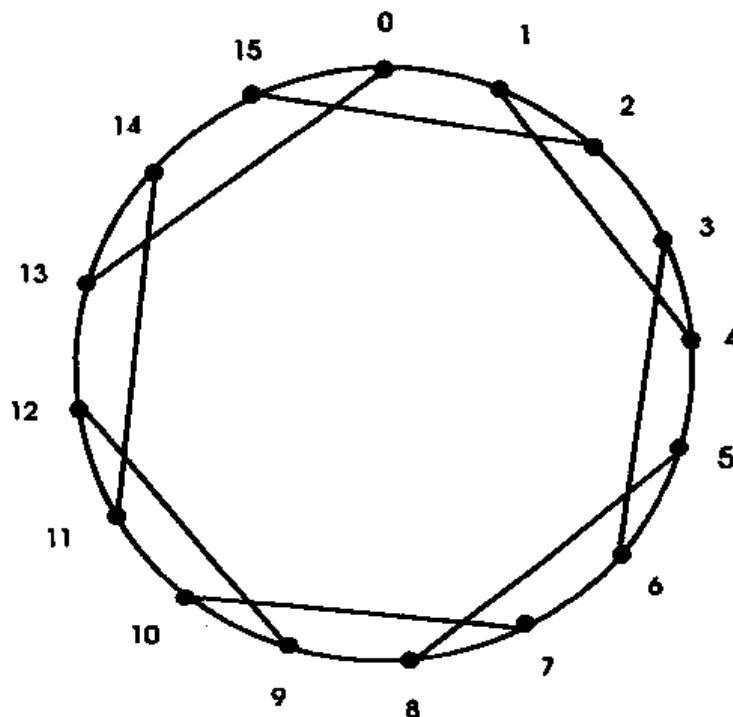
5和8连接；

7和10连接；

9和12连接；

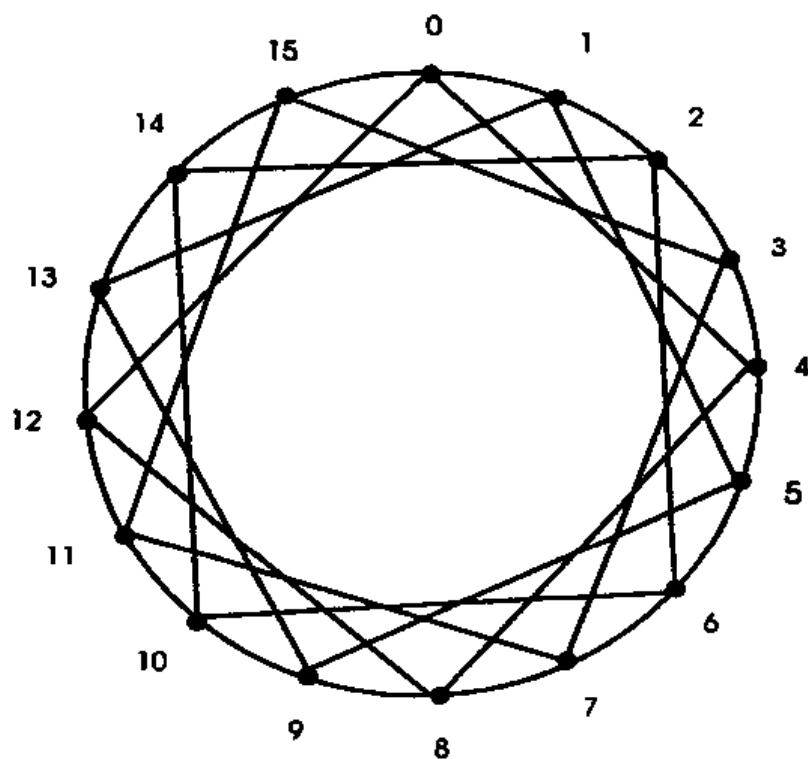
11和0连接;

14和2连接。



图A-2 度为3的弦环网络

一个度为4的弦环网络在图A-3中给出。这可以通过在环状网中增加额外的度得到。每个节点*i*都和*i+4(mod 4)*相连。

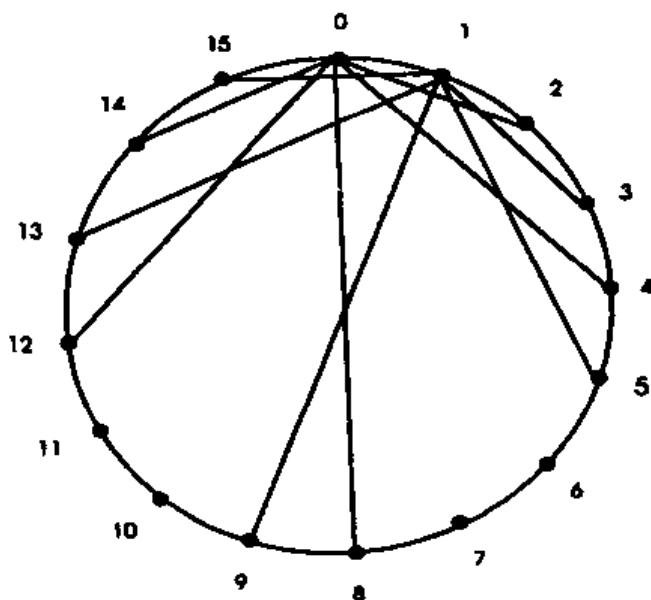


图A-3 度为4的弦环网络

1.9 Barrel Shifter 网络

Barrel Shifter 网络是通过在环状网中增加以下额外的连接得到的：

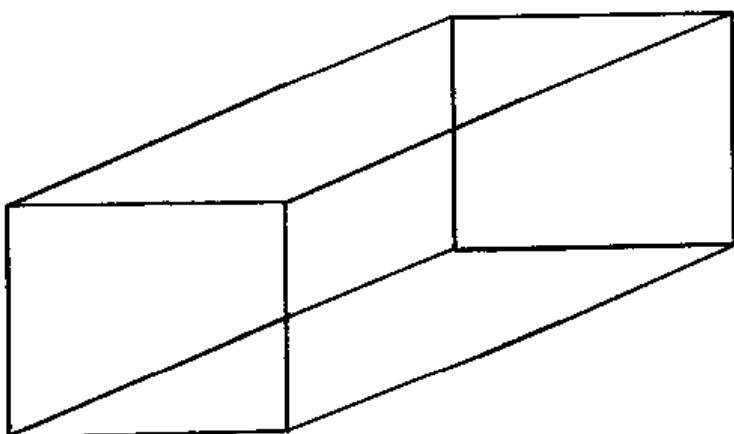
每个节点 i 都和与其距离为2的幂次方的节点 j 连接。即节点 i 和所有与其距离为2, 4, 8, 16, …的节点连接。例如，考虑有16个分别标记为0, 1, 2, 3, …, 15的节点的环状网。其中，节点0已经与1和15连接；节点2和14与0的距离为2，因此节点0与2和14连接；节点4和12与0的距离为4，因此节点0与4和12连接；0和8之间的距离为8，因此它们之间相连接。因此在 Barrel Shifter 网络中，节点0与节点1, 2, 4, 8, 12, 14和15相连接。相似地，节点1与节点2, 3, 5, 9, 13, 15和0相连接。这些在图A-4中给出。



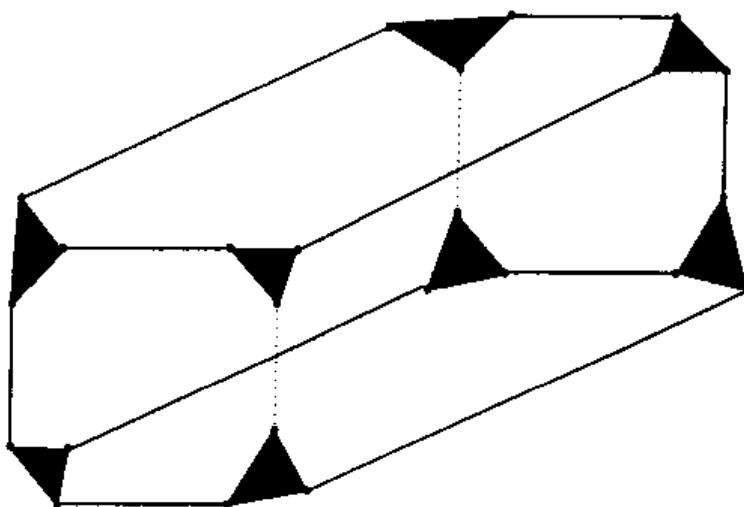
图A-4 有15个节点的Barrel Shifter网络（只给出了节点0和1的连接）

1.12 立方体连接环

通过把 k -立方体中的每个节点替换成 k -环，就得到了 k -立方体连接环(k -CCC)。例如，考虑图A-5中给出的3-立方体。通过把3-立方体中的每个节点替换成3-环，就得到了3-立方体连接环(3-CCC)。如图A-6所示。



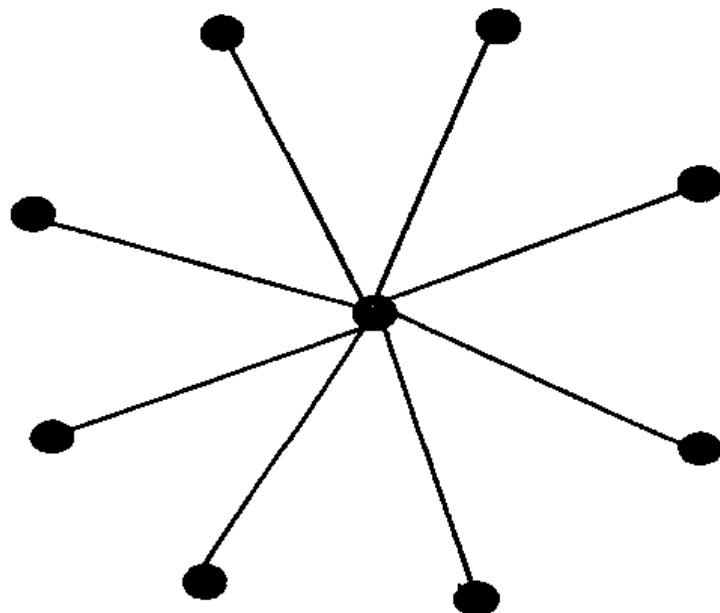
图A-5 3-立方体



图A-6 3-立方体连接环

1.13 星形网络

星形网络以一个节点作为中心节点，并与所有其他节点直接连接。如图A-7所示。星形网络的直径为2，这与节点的个数无关。



356

图A-7 星形网络

第3章

3.4b) 我们给由 n 个处理器所组成的每个组分配一个数据。 i 组得到 $A(i)$ 并检查 $A(i)$ 是否为真。如果 $A(i)$ 为真，那么对于所有的 $j > i$ ， $A(j)$ 为假。只有最小下标对应的 $A(i)$ 仍然为真。

算法Smallest Index_CRCW

输入：布尔数组 $A(1:n)$

输出：使 $A(k)$ 为真的最小下标 k

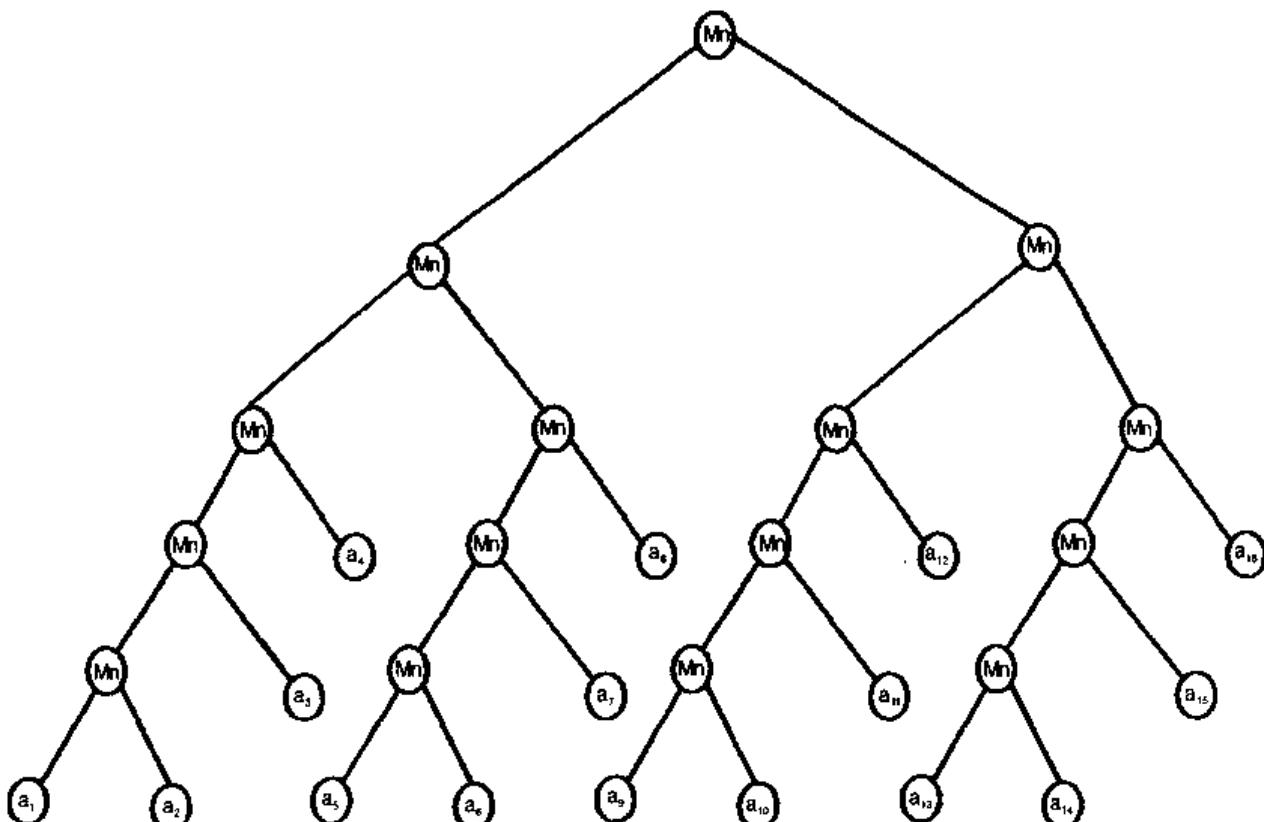
1. Copy $A(1:n)$ to $B(1:n)$
2. For $i = 1$ to n do in parallel
 3. if $B(i) = \text{true}$ then
 - For $j = i + 1$ to n in parallel
 - $B(j) = \text{false}$
 - End parallel
 4. End parallel
5. For $i = 1$ to k
6. if $a(i) = \text{true}$
 - return(k)
7. End parallel
8. END

复杂度分析 第1步需要 $O(n)$ 个处理器，执行时间为 $O(1)$ 。第2~4步需要 $O(n^2)$ 个处理器，执行时间为 $O(1)$ 。第5~7步需要 $O(n)$ 个处理器，执行时间为 $O(1)$ 。第3步中的 $B(j) = \text{false}$ 需要并发写操作。因此该算法需要用CRCW PRAM模型。在CRCW PRAM模型中，需要 $O(n^2)$ 个处理器，执行时间为 $O(1)$ 。

3.7 当 $n = 16$ 时，数组 $A(1:16)$ 被分成 $\log n$ 个组。

组1	组2	组3	组4
$A(1:4)$	$A(5:8)$	$A(9:12)$	$A(13:16)$

在每一组内，每个处理器用串行方式求最小值，然后用并行方式求。如图A-8所示。



图A-8 数组的最小值

第4章

357

4.3 假定存在一个数组 $\text{count}(1:n)$, 其中

$$\begin{aligned}\text{Count}(i) &= 1, A(i) > X \\ &= 0, \text{否则}\end{aligned}$$

现在计算Count数组元素的和, 就得到 $A(1:n)$ 中大于 X 的元素个数。算法如下:

算法Greater Count

输入: $A(1:n), X$

输出: $K =$ 大于 X 的元素个数

1. For $i = 1$ to n do in parallel
2. if $A(i) > X$ then
- count(i) = 1
- else
- count(i) = 0
- endif
3. End parallel
4. Find the sum of the count (1: n) array and store the values in k
5. return k
6. END

358

复杂度分析 第1~3步需要 $O(n)$ 个处理器, 执行时间为 $O(1)$ 。第4步是求和算法, 因此需要 $O(n^2)$ 个处理器, 执行时间为 $O(\log n)$ 。这样, 此算法需要 $O(n^2)$ 个处理器, 执行时间为 $O(\log n)$ 。

4.5 对称矩阵算法

输入: $A(1:n, 1:n)$

输出: A 是否为对称矩阵

1. For $i = 1$ to n do in parallel
2. For $j = 1$ to i do in parallel
3. If $A(i,j)$ and $A(j,i)$ are not equal then conclude that A is not symmetric
- Abort the loop
- endif
4. End parallel
5. End parallel
6. Conclude that A is Symmetric if the loop was not aborted in step 3
7. END

359

360

复杂度分析 该算法在CRCW PRAM模型下, 需要 $O(n^2)$ 个处理器, 时间复杂度为 $O(1)$ 。

索引

索引中标注的页码为英文原书页码，与书中边栏的页码一致。

A

- Absolute speedup (绝对加速比), 52
- Algebraic equations (代数方程), 277
- Alpha (Alpha处理器), 4
- ALU (ALU), 3
- Arithmetic expression (算术表达式), 160-162
- Arithmetic logic unit (算术逻辑单元), 3
- Arrangement graph (排列图), 43
- Array (数组), 59
- Array processor (阵列处理器), 16
- Articulation point (断点), 85
- Artificial intelligence (人工智能), 4
- Assignment statement (赋值语句), 45, 46
- Associative memory processor (相联存储处理器), 27
- Asymptotic speed up (渐近加速比), 53
- Atoms (原子), 61

B

- Backward substitution (向后消去), 287
- Barrel shifter network (Barrel Shifter 网络), 57, 354
- BC-tree (BC树), 88
- BFS (BFS), 83, 188
- Bi-clique (双团), 223
- Biconnected graph (2-连通图), 86
- Binary tree (二叉树)
 - complete (完全的), 75
 - definition (定义), 29
 - model (模型), 74, 108
- Binomial co-efficient (二项式系数), 131
- Biconnected components (2-连通支), 201
- Bi-partite graph (二部图), 90
- Bisection method (对分法), 278
- Block (块), 87
- Block-cut vertex tree (Block-cut 顶点树), 88
- Block graph (块图), 87
- Boolean AND (布尔与), 47-50
- Branch (分枝), 103
- Branch weight (分枝权重), 103

B

- Breadth first search (广度优先搜索), 83

- Bubble sort (冒泡排序), 263

C

- CDC-Cyber (CDC-Cyber), 15
- Center (中心), 72, 103
- Central path (中心路), 103
- Centroid (形心), 103
- Chandra algorithm (Chandra算法), 216
- Character recognition (字符识别), 6
- Children (孩子), 73
- Chord's method (弦法), 278
- Chordal ring network (弦环网络), 58, 353
- Chordal graph (弦图), 94, 213
- Chromatic number (色数), 92
- Chromatic polynomial (色多项式), 92, 93
- Circle graph (圈图), 101
- Circuit integration (集成电路), 4
- Circular arc graph (圆弧图), 101
- Circular list (循环列表), 64
- Circuit-Hamiltonian (哈密顿圈), 71
- Clique (团), 69, 93
- Clique algorithm (团算法), 187
- Clique covering (团覆盖), 93
- Coloring (染色), 92
- Complete binary tree (完全二叉树), 75
- Complete domination (完全控制), 105
- Component (分支), 68
- Computing time (计算时间), 50
- Connected component (连通支), 68, 191
- Connected graph (连通图), 71
- Connectivity (连通度), 85
- Connectivity algorithm (连通度算法), 188
- Continuous list (连续列表), 63
- Contraction of tree (树收缩), 155
- Control unit (控制单元), 3
- Core (核心), 103
- Cost normalized speed up (成本规范化加速比), 53
- Crank Nicolson method (Crank Nicolson法), 347

Cray XMP (Cray XMP), 4, 15

CRCW (CRCW), 43

CREW (CREW), 43

Cube connected cycles (立方体连接环), 355

CV graph (CV图), 227

Cycle (圈), 66

D

DAG (DAG), 29

Data dependency (数据相关性), 11

Data structure (数据结构), 59

DE graph (DE图), 97

Decomposability (可分解性), 11

Definite integrals (定积分), 326

Degree (度), 66

Degree algorithms (度算法), 184

Depth first search (深度优先搜索), 83

Descendents of a vertex (顶点后代), 151

Determinant (行列式), 278

DFS (DFS), 83, 188, 191

Diagonal matrix (对角矩阵), 285

Diameter (直径), 72, 175

Differential equations (微分方程), 335

Differentiation-partial (偏微分), 321

Differentiation (微分), 319-321

Directed acyclic graph (有向无环图), 29

Distributed processing (分布式处理), 14

Divide and conquer (分治), 116

Domatic number (domatic数), 105

Dominating set (控制集), 104

Doubly linked list (双向链表), 64

Dufort Frankel method (Dufort Frankel法), 349

Duncan's taxonomy (Duncan分类), 25

DV graph (DV图), 97

E

Eccentricity (离心率), 72

Ed Roberts (Ed Roberts), 4

Edge coloring (边染色), 92

Edge set (边集), 66

Efficiency(效率), 51, 54

Efficient domination (有效控制), 105

Elliptic equations (椭圆方程), 336

Equations (方程), 277

ERCW (ERCW), 43

EREW (EREW), 43

Erlangen (Erlangen), 20, 23

Eta (Eta), 10, 15

Euler (欧拉),

formula (欧拉公式), 335

graph (欧拉图), 70

graph algorithm (欧拉图算法), 186

line (欧拉闭迹), 70,

tour (欧拉环游), 145

walk (欧拉闭迹), 70

Expert system (专家系统), 6

F

False position (试位法), 278

Finite graph (有限图), 66

First generation computer (第一代计算机), 4

Floyd-Warshall algorithm (Floyd-Warshall算法), 209

Flynn's classification (Flynn分类), 20

For-loop (For循环), 46, 47

Forest (森林), 78

Forward substitution (向前消去), 287

Fourier transforms (傅里叶变换), 292

Frontier (边界), 97

Fujitsu (富士通), 15

Full binary tree (满二叉树), 75

Nearest neighbors (最近邻居), 179

G

Gauss elimination (高斯消元法), 287, 312

Geometric interpretation (几何解释), 277-278

Gilio taxonomy (Gilio分类), 24

Givens rotation (Givens旋转), 289

Graph (图)

biconnected graph (2-连通图), 86

bipartite (二部图), 90

chordal (弦图), 94

circle (圆图), 101

circular-arc (圆弧图), 101

connected (连通图), 71

distance hereditary (距离遗传图), 101

Euler (欧拉图), 70

Hamiltonian (哈密顿图), 71

interval (区间图), 97

k-connected (k -连通图), 87

Kuratowski graphs (库拉托斯基图), 90

mop (极大外可平面图), 92

outerplaner (外可平面图), 92

planner (外可平面图), 88
 regular (正则图), 66
 separable (可分离图), 86
 triconnected (3-连通图), 87
 algorithms (图算法), 184
 unicursal (一笔图), 71
 Growing by doubling (二倍增长), 112

H

Hajime Karatsu (Hajime Karatsu), 4
 Hamiltonian (哈密顿)
 circuit (哈密顿圈), 71
 graph (哈密顿图), 70, 71
 Handler (Handler), 20, 23
 Hazuhiro Fuchi (Hazuhiro Fuchi), 5
 Heapsort (堆排序), 266-269
 Heat flow equation (热传导方程), 336, 343
 Hideo Aiso (Hideo Aiso), 5
 High performance computer (高性能计算机), 14
 Hitachi (日立), 15
 Homeomorphism (同胚), 69
 HP superchip (HP超级芯片), 5
 Hwang-Brigg's taxonomy (Hwang-Brigg分类), 24
 Hyperbolic equations (双曲方程), 336
 Hypercube (超立方体), 33

I

IBM 3090/400/VF (IBM 3090/400/VF), 15
 ICOT (ICOT), 5
 If structure (If结构), 46
 Illiac iv (Illiad iv), 15
 Image processing (图像处理), 6-7
 Independent domination (独立控制), 105
 Independent set (独立集), 94
 Index (下标), 62
 Induced subgraph (诱导子图), 68
 Infinite graph (无限图), 66
 Infix (中缀), 60
 Inherently sequential (内在串行), 13
 Initial value problem (初值问题), 335
 Input unit (输入单元), 3
 Insertion sort (插入排序), 264-265
 Integrals (积分), 326
 Integrals-definite (定积分), 326
 Intel (Intel), 5

Intelligent machine (智能机器), 6, 9
 Interval graph (区间图), 97
 Interpolation
 Lagrange's (拉格朗日插值), 331
 linear (线性插值), 329
 quadratic (二次插值), 330
 Intersection graph (交图), 94
 Inverse of matrix (矩阵的逆), 305
 Isolated vertex (孤立点), 66
 Isomorphism (同构), 69
 Iyengar's algorithm (Iyengar算法), 216

K

k-connected graph (*k*-连通图), 87
k-star graph (*k*-星形图), 41
k-tree (*k*-树), 216
k-cubes (*k*-立方体), 33
 Key board (键盘), 3
 Koneigsberg bridge (哥尼斯堡七桥), 70
 Kruskal's algorithm (Kruskal算法), 204
 Kuratowski graphs (库拉托斯基图), 90

L

Laasonen method (Laasonen法), 344
 Lagrange's interpolation (拉格朗日插值), 331
 Laplace equation (拉普拉斯方程), 336
 Laplacian (拉普拉斯算符), 325
 LBFS (字典顺序广度优先搜索), 96
 LCA (最低公共祖先), 151
 Left child (左孩子), 75
 Level number (层数), 73
 Level of a vertex (顶点层数), 151
 Levels of parallelism (并行的层次), 18
 Linear equations (线性方程), 284
 Linear interpolation (线性插值), 330
 Linked list (链接列表), 61
 List (列表)
 circular (循环), 64
 doubly linked (双向链接), 64
 linked (链接), 64
 list (列表), 59
 ranking (排序), 113
 Loosely coupled (松耦合), 27
 Lowest common ancestor (最低公共祖先), 151-154

M

Magnetic disk (磁盘), 3

Magnetic tapes (磁带), 3
 Manber Udi (Manber Udi), 102
 Matching (匹配), 101
 Matrix (矩阵)
 inverse (逆矩阵), 305
 multiplication (矩阵乘法), 124
 toeplitz (Toeplitz矩阵), 308
 triangular (三角形矩阵), 306
 Maximum matching (最大匹配), 101
 MCC (MCC), 96
 MCS (MCS), 96
 MEC (MEC), 96
 Median (重心), 103
 Medical diagnosis (医疗诊断), 6, 9
 Memory unit (存储单元), 3
 Mergesort (合并排序), 266-270
 Merging (合并)
 bitonic (双调), 258-260
 ranking (秩合并), 255-257
 Mesh network (网格网络), 40
 Message broadcasting (消息广播), 57
 Metal oxide (金属氧化物), 5
 Micro processor (微处理器), 4
 MIMD (MIMD), 22
 Minimal dominating set (极小控制集), 105
 Minimal separator (极小分离集), 86
 Minimum coloring (最小染色), 92
 Minimum cost spanning tree (最小成本支撑树), 78
 Minimum dominating set (最小控制集), 105
 Mop graph (极大外可平面图), 92
 Mostech (Mostech), 5
 Motorola (摩托罗拉), 5
 MPP (MPP), 25
 Multi computer (多计算机), 14
 Multi processor (多处理器), 14

N

Naor algorithm (Naor算法), 220
 NC algorithm (NC 算法), 213
 Network (网络)
 barrel shifter (barrel shifter), 354
 chord ring (弦环), 353
 model (模型), 32
 ring (环网络), 352
 sorting (排序), 270-272
 star (星形), 356

Newton-Raphson (牛顿-拉弗森方法), 278
 Non-planar graph (不可平面图), 90
 Null graph (零图), 66
 Nullity (零维数), 78

O

Odd-even reduction method (奇偶约化法), 313-317
 Outerplanar graph (外可平面图), 92
 Output unit (输出单元), 3

P

Paper tapes (纸带), 3
 Parabolic equation (抛物方程), 336, 337
 Paradigms (环境), 108
 Parallel computers (并行计算机), 10
 Parallel edge (平行边), 66
 Parent (双亲), 73, 75
 Parity graph (奇偶图), 101
 Partial differential equations (偏微分方程), 321, 336
 Partial sums (部分和), 125
 Partitioning (划分), 117
 Path (路), 66
 length (路长), 76
 matching (路匹配), 102
 root (根), 165-168
 graph (路图), 97
 graph recognition (路图识别), 228
 Pattern recognition (模式识别), 6-7
 PC (PC), 4
 Pendant vertex (悬挂点), 66
 Pentium Pro (Pentium Pro), 4
 PEO (PEO), 94
 PEPE (PEPE), 25
 Perfect coloring (正常染色), 92
 Perfect elimination scheme (完美消去格式), 94
 Performance metrics (性能尺度), 50
 Personal computer (个人计算机), 4
 Pipe lining (流水线), 13
 Pivotal condensation (选主元归约), 281
 Planar graph (可平面图), 88
 Plotters (绘图仪), 3
 Pointer jumping (指针跳转), 112
 Poisson equation (泊松方程), 336
 Polynomial multiplication (多项式乘法), 302
 Post order numbering (后序编号), 148
 Postfix (后缀), 60

PQ-tree (PQ树), 98

PRAM (PRAM模型), 43

Prefix (前缀), 60

Prefix minima (前缀最小), 135

Preorder numbering (前序编号), 179

Prim's algorithm (Prim算法), 204

Printer (打印机), 3

Punched cards (穿孔卡片), 3

Pyramid network (金字塔网络), 41

Q

Quadratic interpolation (二次插值), 331

Queue (队列), 59

R

Radius (半径), 72

Rake operation (Rake运算), 156

Range minima (范围内最小值问题), 134

Rank (秩), 78

Ranking (秩, 排序), 255-257

RDE graph (有根有向边路图), 97

RDV graph (有根有向点路图), 97

Real speed up (实际加速比), 52

Recognition algorithm (识别算法), 106

Recognition of k -tree (k -树判别), 217

Reduction machines (规约机), 28

Region (区域), 90

Regular falsi (试位法), 278

Regular graph (正则图), 66

Relative speedup (相对加速比), 52

Richardson's formula (Richardson公式), 348

Right child (右孩子), 75

Ring network (环状网), 57, 352

RISC (RISC), 15

Root (根), 73

Root finding (求根), 162-165

Rooted tree (有根树), 73

Rooting a tree (给树加根), 147

S

Scalability (可扩展性), 55

Scalar product (内积), 123

Schaffer's algorithm (Schaffer算法), 220

Schmidt method (施密特法), 338

Searching (搜索)

binary (二分), 251

CREW PRAM (CREW PRAM模型), 252

more data (更多数据), 253

sequential (串行), 251

unsorted (未排序的), 255

Secant method (割线法), 278

Second generation computer (第二代计算机), 4

Self edge (环), 66

Separable graph (可分离图), 86

Separator (分离集), 86

Sequential computer (串行计算机), 10

Shared memory model (共享存储器模型), 22

Shell's sort (Shell排序), 265

Shifter network (Shifter 网络), 354

Shortest path problem (最短路问题), 206

Simple algorithm (简单算法), 47, 123

Simple graph (简单图), 66

Simplicial vertex (单纯顶点), 94

Simpson (辛普森), 328

Skewed binary tree (斜二叉树), 74

Slater (Slater), 103

Sollin's algorithm (Sollin算法), 204

Sorting (排序), 262

bubble (冒泡), 263

heap (堆), 266-269

insertion (插入), 264-265

merge (合并), 269-270

sequential (串行), 262

shell (shell), 265

networks (网络), 270-272

Spanning forest (支撑森林), 78

Spanning tree (支撑树), 77, 204

Speech recognition (语音识别), 6, 8

Speed up (加速比), 51

asymptotic (渐近加速比), 53

absolute (绝对加速比), 52

real (实际加速比), 52

relative (相对加速比), 52

Stack (堆栈), 59

Star graph (星形图), 41

Star network (星形网络), 356

STARAN (STARAN), 25

STMD (STMD), 20

STSD (STSD), 20

Subgraph (子图), 68

induced (诱导子图), 68

Subtree (子树), 73

Successive approximation (逐次逼近法), 278

Suffix minima (后缀最小), 135

Summation (求和), 108

Sun Wu (Sun Wu), 102

Symmetric matrix (对称矩阵), 359

Synchronous (同步), 25

Systolic arrays (脉动阵列), 26

T

Taxonomy (分类), 20

Taylor's series (泰勒级数), 319

Third generation computer (第三代计算机), 4

TI-ASC (TI-ASC), 15

Tightly coupled (紧耦合), 27

Toeplitz matrix (Toeplitz矩阵), 308

Tohru Moto Oka (Tohru Moto Oka), 4

Topology of network (网络拓扑), 32

Transitive closure matrix (传递闭包矩阵), 196

Trapezoidal rule (梯形法则), 327

Tree (树), 65, 71

algorithms (树算法), 145

contraction (树收缩), 155

conversion (树转化), 168

graph (树图), 83

rooting (求树根), 147

rooted (有根树), 73

Triangular matrix (三角形矩阵), 286, 306

Triconnected graph (3-连通图), 87

Tridiagonal system (三对角方程组), 311

U

Udi Manber (Udi Manber), 102

UE graph (UE图), 97

Unicursal graph (一笔图), 71

UV graph (UV图), 97, 228

V

Vector processor (向量处理器), 26

Vertex coloring (顶点染色), 92

Vertex connectivity (顶点连通度), 86

Vertex expansion (顶点扩展), 100

Vertex set (顶点集), 66

VLSI (VLSI), 4

Von-Mises formula (Von-Mises公式), 278

Von Neumann (冯·诺依曼), 3

W

Walk (通道), 66

Warshall's algorithm (Warshall算法), 209

Wave equation (波动方程), 336

Weather forecasting (天气预报), 6, 8

Weighted path length (加权路径长度), 76

While-loop (While循环), 46

Wu Sun (Wu Sun), 102

Z

ZMOB (ZMOB), 16

