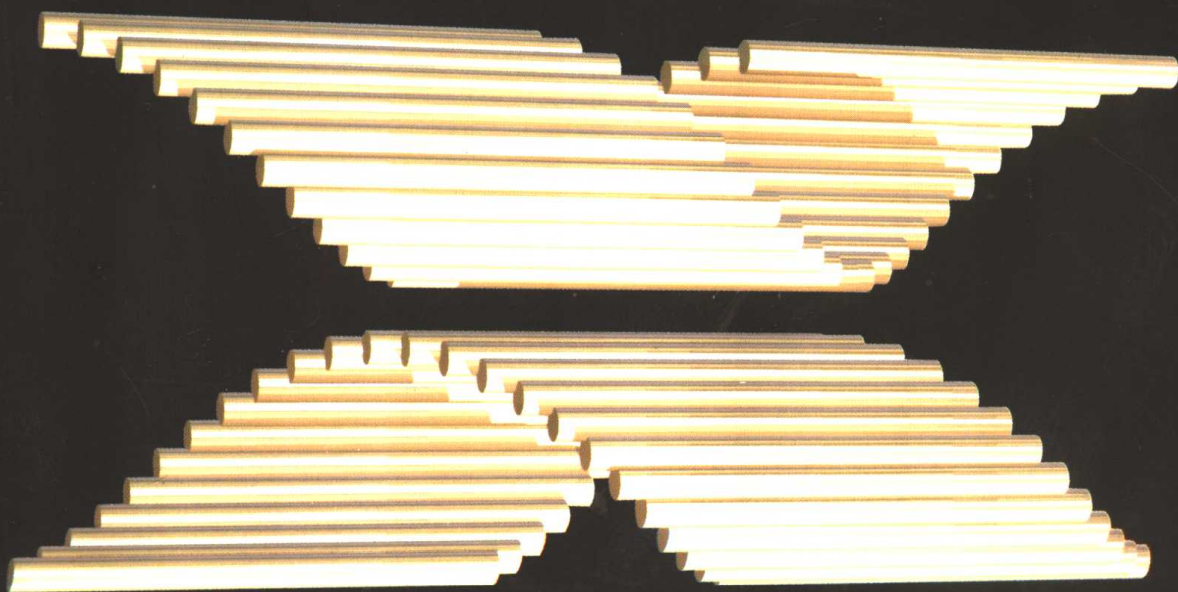


# MPI与OpenMP 并行程序设计

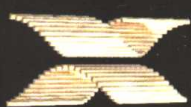
C语言版

Michael J. Quinn 著  
陈文光 武永卫 等译



**PARALLEL PROGRAMMING**  
in C with MPI and OpenMP

# MPI与OpenMP 并行程序设计 C语言版



本书是美国Oregon州立大学的Michael J. Quinn教授在多年讲授“并行程序设计”课程的基础上编写而成的,主要介绍用C语言,并结合使用MPI和OpenMP进行并行程序设计,内容包括并行体系结构、并行算法设计、消息传递编程、Eratosthenes筛法、Floyd算法、性能分析、矩阵向量乘法、文档分类、蒙特卡洛法、矩阵乘法、线性方程组求解、有限差分方法、排序、快速傅立叶变换、组合搜索、共享存储编程、融合OpenMP和MPI以及5个附录。

本书按授课方式安排章节,通过划分、通信、集聚和映射等四步的并行程序设计方法,来解决各种实际的并行性问题,使读者掌握系统化的并行程序设计方法,开发出高效的并行程序。

本书不仅是一本优秀的并行程序设计教材,对广大的相关专业人员也很有参考价值。

## 世界著名计算机教材精选

- 计算机网络 (第 4 版)  
Computer Networks (Fourth Edition)
- 数据结构 C++语言描述  
Data Structures with C++
- 计算机组织与结构: 性能设计 (第 4 版)  
Computer Organization and Architecture:  
Design for Performance (Fourth Edition)
- 数据库系统基础教程  
A First Course in Database Systems
- 面向对象系统分析与设计  
Object-Oriented Systems Analysis and Design
- 计算理论基础 (第 2 版)  
Elements of the Theory of Computation (Second Edition)
- 多媒体技术: 计算、通信和应用  
Multimedia: Computing, Communications & Applications
- 因特网和万维网的基本原理与技术  
In-line/On-line: Fundamentals of the Internet  
and the World Wide Web
- TCP/IP协议族  
TCP/IP Protocol Suite
- 通信网基本概念与主体结构  
Communication Networks: Fundamental Concepts and Key  
Architectures
- 计算机组成和设计 (硬件/软件接口)  
Computer Organization & Design (The Hardware/Software Interface)
- 分布式系统原理与范型  
Distributed Systems Principles and Paradigms
- 数据结构与抽象 (Java语言版)  
Data Structures and Abstractions with Java
- 安腾体系结构: 理解64位处理器和EPIC原理  
Itanium® Architecture for Programmers: Understanding 64-Bit  
Processors and EPIC Principles
- MPI与OpenMP并行程序设计 (C语言版)  
Parallel Programming in C with MPI and OpenMP

ISBN 7-302-09555-8



9 787302 095552 >

定价: 51.00元



<http://www.mheducation.com>



世界著名计算机教材精选

# MPI与OpenMP并行程序设计 (C语言版)

Michael J. Quinn 著

陈文光 武永卫 等 译

清华大学出版社  
北 京

Michael J. Quinn

**Parallel Programming in C with MPI and OpenMP**

EISBN: 0-07-282256-2

Copyright © 2004 by The McGraw-Hill Companies, Inc.

Original language published by The McGraw-Hill Companies, Inc. All Rights reserved. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Simplified Chinese translation edition is published and distributed exclusively by Tsinghua University Press under the authorization by McGraw-Hill Education (Asia) Co., within the territory of the People's Republic of China only, excluding Hong Kong, Macao SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书中文简体字翻译版由美国麦格劳-希尔教育出版(亚洲)公司授权清华大学出版社在中华人民共和国境内(不包括中国香港、澳门特别行政区和中国台湾)独家出版发行。未经许可之出口,视为违反著作权法,将受法律之制裁。未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

北京市版权局著作权合同登记号 图字 01-2003-7768 号

版权所有, 翻印必究。举报电话: 010-62782989 13901104297 13801310933

本书封面贴有 McGraw-Hill 公司防伪标签, 无标签者不得销售。

图书在版编目(CIP)数据

MPI 与 OpenMP 并行程序设计: C 语言版/奎因(Quinn, M. J.)著; 陈文光, 武永卫等译. —北京: 清华大学出版社, 2004.10

(世界著名计算机教材精选)

书名原文: Parallel Programming in C with MPI and OpenMP

ISBN 7-302-09555-8

I. M… II. ①奎… ②陈… ③武… III. 并行程序-程序设计-教材 IV. TP311.11

中国版本图书馆 CIP 数据核字(2004)第 095663 号

出版者: 清华大学出版社

<http://www.tup.com.cn>

社总机: 010-62770175

地址: 北京清华大学学研大厦

邮编: 100084

客户服务: 010-62776969

责任编辑: 龙啟铭

印刷者: 世界知识印刷厂

装订者: 三河市金元装订厂

发行者: 新华书店总店北京发行所

开本: 185×260 印张: 27.75 字数: 687 千字

版次: 2004 年 10 月第 1 版 2004 年 10 月第 1 次印刷

书号: ISBN 7-302-09555-8/TP·6645

印数: 1~3000

定价: 51.00 元

---

本书如存在文字不清、漏印以及缺页、倒页、脱页等印装质量问题, 请与清华大学出版社出版部联系调换。联系电话: (010) 62770175-3103 或 (010) 62795704



# 前 言

本书是用 C 语言进行 MPI 和 OpenMP 并行编程的实用教程，适用于大学高年级本科生、研究生以及利用本书进行自学的计算机专业人员。读者需要有很好的 C 语言编程经验，并学习过基础的算法分析课程。

对并行程序设计感兴趣的 Fortran 程序员也可以从本书中受益。尽管本书中的例子都是用 C 写的，但是使用 MPI 和 OpenMP 进行并行程序设计的基本概念对于 C 和 Fortran 来说是基本相同的。

在过去 20 年中，我为数以百计的本科生和研究生讲授了并行程序设计。在这个过程中，我逐步了解了人们在开始“用并行方式思考”和编写并行程序时所遇到的各种问题。逐步设计和实现的程序更容易让学生们受益。因此我的哲学是仅在需要时才引入新的功能，尽可能地在解决设计、实现和分析中的具体问题时引入新概念。

本书的前两章解释了并行计算的起源并对并行体系结构进行了综述。第 3 章介绍了 Foster 的并行算法设计方法论和几个应用该方法的实例。第 4、5、6、8 和 9 章介绍了如何使用该设计方法，为一系列从易到难的问题开发 MPI 程序。这些章节中用到的 27 个 MPI 函数是 MPI 函数库的一个子集，但已经足够为很多类型的实际应用设计并行程序。这些章节还介绍了能够简化矩阵和向量 I/O 的函数，附录 B 中给出了该 I/O 库的源代码。

第 4、5、6 和 8 章的程序在一个集群系统上进行了性能测试，并在书中给出了测试结果。由于新的处理器比本书中所用到的要快得多，读者可能会发现本书中用到的处理器已经落后了好几代。但是在书中给出测试结果并不是要让读者对计算速度感到吃惊，而是为了表明将串行程序性能、互连网络的通信和延迟等信息集成起来，可以相当准确地预测并行程序的性能。

第 7 章主要介绍了分析和预测并行系统性能所用到的 4 种度量：Amdahl 定律、Gustafson-Barsis 定律、Karp-Flatt 度量和等效度量。

第 10~16 章提供了更多的例子，表明如何分析问题并设计好的并行算法来解决问题。使用 MPI 来实现并行算法的任务则留给了读者。我介绍了蒙特卡洛法和并行随机数生成的相关问题。后面几章介绍了若干关键算法：矩阵乘法、高斯消去法、共轭梯度法、有限元方法、排序、FFT、回溯搜索、分支定界搜索以及 Alpha-Beta 搜索。

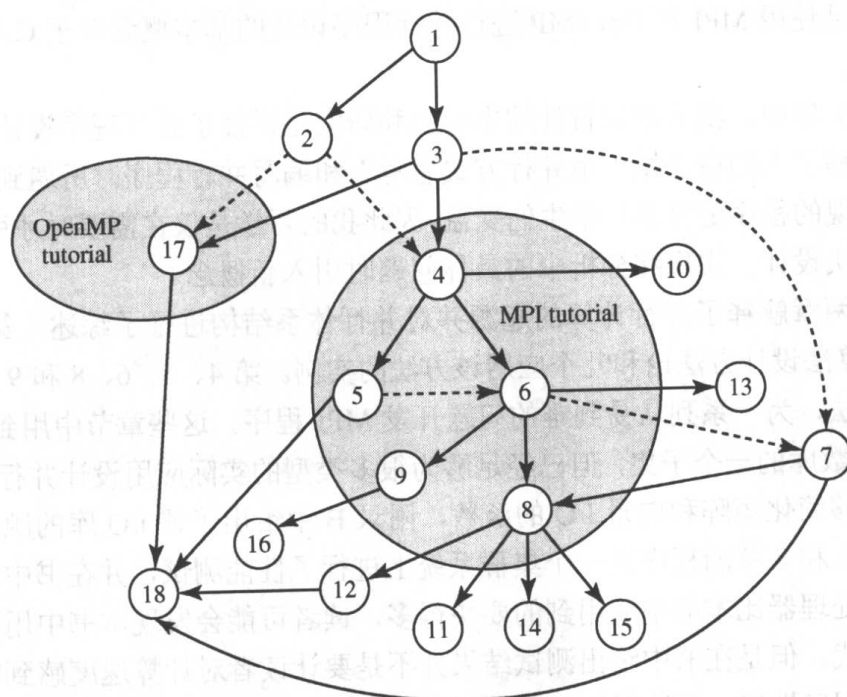
第 17、18 章介绍了新的共享内存编程标准 OpenMP。介绍了将串行程序段转化为并行程序段所需的 OpenMP 功能，并用两个实例讲解了如何将 MPI 程序转化为 MPI/OpenMP 混合程序。在多处理器集群系统上，混合程序比 MPI 程序能够得到更好的性能。

本书包含了超过一学期并行程序设计课程所需的内容。尽管并行程序设计比传统程序设计要困难得多，但也会带来更多的回报。即使在老师的指导与帮助下，大多数学生对于在多个处理器上执行单个任务仍然会觉得有些害怕。但是，当他们看到调试过的程序比“普通”C 程序要快得多的时候，这种害怕就转换成了真正的成就感。因此，编程练习应

该是本课程的中心内容。

幸运的是, 并行计算机比以前更容易获得了。如果无法使用商用并行计算机, 使用几台 PC 机、网络设备和自由软件搭建一个小型的集群系统是一件很容易的事情。

图前.1 说明了本书章节的阅读顺序。图中实线箭头代表强依赖关系, 虚线箭头代表弱依赖关系。按照章节顺序阅读本书将满足所有的章节依赖关系。但是, 如果你希望让学生尽快开始用 C 语言编写 MPI 程序, 那么你可能希望跳过第 2 章, 或仅讲授该章的 1~2 节内容。如果你希望集中讲授数值算法, 那么可以跳过第 5 章, 并用其他方式介绍函数 MPI\_Bcast。如果你希望学生从蒙特卡洛法开始, 那么可以在第 4 章后直接跳到第 10 章。如果你希望在 MPI 之前就讲授 OpenMP, 可以在第 3 章后直接跳到第 17 章。



图前.1 章节之间的依赖关系: 实线箭头代表强依赖关系, 虚线箭头代表弱依赖关系

感谢 McGraw-Hill 出版社的每位员工, 他们帮助我创造了这本书, 特别是 Betsy Jones, Michelle Flomenhoft 和 Kay Brimeyer。感谢他们的赞助、鼓励和帮助。同样感谢 Maggie Murphy 和 Interactive Composition Corporation 公司其他排版者所提供的帮助。

感谢本书的审稿者, 他们仔细阅读了我的手稿, 修正了其中的错误, 指出其中的弱点并建议补充内容。他们是: A. P. W. Bohm, Colorado 州立大学; Thomas Cormen, Dartmouth 学院; Narsingh Deo, Central Florida 大学; Philip J. Hatcher, New Hampshire 大学; Nickolas S. Jovanovic, Arkansas 大学 Little Rock 分校; Dinesh Mehta, Colorado 矿业学校; Zina Ben Miled, Indiana 大学-Purdue 大学; Paul E. Plassman, Pennsylvania 州立大学; Quinn O. Snell, Brigham Young 大学; Ashok Srinivasan, Florida 州立大学; Xian-He Sun, Illinois 理工学院; Virgil Wallentine, Kansas 州立大学; Bob Weems, Texas 大学 Arlington 分校; Kay Zemoudel, California 州立大学和 Jun Zhang, Kentucky 大学。

Oregon 州立大学的许多同事也给予了我不少帮助: Robin Landaw 和 Henri Jansen 分别帮助我了解了 Monte Carlo 算法和详细的平衡条件; 学生 Charles Sauerbier 和 Bernd Michael



---

Kelm; Tim Budd 教我如何把 PostScript 图插入到 LaTeX 文档中; Jalal Haddad 提供了技术支持。感谢他们的帮助!

最后,感谢我的妻子 Victoria。她鼓励我返回到教科书的写作中来。感谢那个令人鼓舞的圣诞节礼物:“Chicken Soup for the Writer’s Soul: Stories to Open the Heart and Rekindle the Spirit of Writers”。

# 译者序

并行处理是解决人类重大挑战问题的关键技术。随着集群技术和 SMP 系统的发展，并行处理在科学研究、工程计算以及商业计算等领域得到了越来越多的应用。并行程序设计一直是并行处理技术中的核心问题，人们也进行了非常多的尝试，提出了多种并行程序开发方法和并行程序设计语言。近年来，MPI 和 OpenMP 逐渐成为了并行程序设计的主流方式。

本书是介绍使用 MPI 和 OpenMP 进行并行程序设计的教材，作者是美国俄勒冈州立大学的 Michael J. Quinn 教授。我们觉得此书具有下面几个特点：

- 作为一本程序设计教材，本书没有简单地罗列所涉及的语句和函数，而是为每个语句和函数的“出场”都精心设计了实际问题，让读者能够在实际的上下文中了解这些语句和函数的目的和作用，以及在实际程序中的使用方式。这无疑将对读者理解相关的概念带来极大的帮助。
- 全书中贯穿使用了 Foster 提出的并行程序设计方法，即划分、通信、聚集和映射四步法。掌握系统化的并行程序设计方法，有助于读者养成良好的习惯，做出正确的设计决策，开发出高效率的并行程序。
- 并行程序的性能是非常重要的，一个低效的并行程序可能还没有串行程序速度快。本书专门介绍了并行程序性能量度模型和性能分析技术，并在多个章节中进行了实例分析，对并行程序的性能问题给予了足够的重视。
- 内容较新。尽管目前已经有较多讲授 MPI（消息传递接口）并行程序设计的教材，但本书不仅包括了 MPI 并行程序设计，还包括了 OpenMP 程序设计的内容。这是目前许多并行程序设计教材所不具备的。本书还介绍了 MPI/OpenMP 混合编程模式，该模式对于现有的 SMP（对称多处理器）集群系统具有重要意义。此外，处理器在未来一段时间内的发展趋势是在一个芯片上集成多个核心，研究 MPI/OpenMP 混合编程模式对采用多内核处理器构造的并行系统的适用性也是一个重要的研究课题。

本书由于篇幅所限，并未在所有有关问题上都进行详细的展开讲解，但每一章最后的参考文献为希望更加深入学习的读者提供了进一步阅读的建议。

因此我们认为，本书是一本很优秀的并行程序设计教材，适合学习并行程序设计的学生和计算机专业人士阅读。我们希望本书中文版的出版，能够将本书介绍给更多的中国读者，并为大家的阅读带来方便。同时，由于本书所涉及的领域相当广泛，译者水平有限，翻译中可能还存在不妥之处，敬请广大读者批评指正。

感谢清华大学计算机系的周立柱教授，将本书介绍给我们。感谢清华大学出版社的龙啟铭编辑，为本书的中文版的出版所付出的耐心和努力。

全书由陈文光、武永卫、陈永健、李建江、薛瑞尼、齐琳等译。清华大学计算机系都志辉进行了审读。

译者



# 目 录

<b>第 1 章 动机和历史</b> .....	1	2.2.5 超树形网络	24
1.1 概述	1	2.2.6 蝶形网络	25
1.2 现代科学方法	2	2.2.7 超立方体网络	26
1.3 超级计算的进化	3	2.2.8 混洗-交换网络	27
1.4 现代并行计算机	4	2.2.9 小结	28
1.4.1 Cosmic Cube 并行计算机	4	2.3 阵列处理机	29
1.4.2 商品化的并行计算机	5	2.3.1 体系结构与数据并行	29
1.4.3 Beowulf 系统	6	2.3.2 阵列处理机的性能	30
1.4.4 先进战略计算计划	6	2.3.3 处理器互连网络	31
1.5 寻找并行性	7	2.3.4 处理器的启动与阻塞	32
1.5.1 数据相关图	7	2.3.5 其他体系结构特点	33
1.5.2 数据并行性	8	2.3.6 阵列处理机的缺点	33
1.5.3 功能并行性	9	2.4 多处理器	33
1.5.4 流水线	9	2.4.1 集中式多处理器	34
1.5.5 计算规模的考虑因素	11	2.4.2 分布式多处理器	35
1.6 数据聚类	11	2.5 多计算机	38
1.7 为并行计算机编程	13	2.5.1 非对称多计算机	39
1.7.1 扩展编译器	13	2.5.2 对称多计算机	40
1.7.2 扩展串行编程语言	14	2.5.3 怎样的模型对商用集群 来说是最佳的	41
1.7.3 增加并行编程层	14	2.5.4 集群与工作站网络之间 的差异	42
1.7.4 创造一个并行语言	15	2.6 弗林分类法	42
1.7.5 现状	16	2.6.1 SISD	43
1.8 本章小结	16	2.6.2 SIMD	43
1.9 主要术语	16	2.6.3 MISD	43
1.10 参考文献	17	2.6.4 MIMD	45
1.11 练习题	18	2.7 本章小结	45
<b>第 2 章 并行体系结构</b> .....	21	2.8 主要术语	46
2.1 概述	21	2.9 参考文献	47
2.2 互连网络	21	2.10 练习题	47
2.2.1 共享介质与开关介质	22	<b>第 3 章 并行算法设计</b> .....	50
2.2.2 开关网络的拓扑结构	22	3.1 概述	50
2.2.3 二维网格形网络	23		
2.2.4 二叉树形网络	23		

3.2 任务/通道模型.....	50	4.4.2 MPI_Comm_rank 和 MPI_Comm_size 函数.....	80
3.3 Foster 的设计方法论.....	51	4.4.3 MPI_Finalize 函数.....	81
3.3.1 划分.....	52	4.4.4 编译 MPI 程序.....	81
3.3.2 通信.....	53	4.4.5 运行 MPI 程序.....	81
3.3.3 聚集.....	54	4.5 聚合通信简介.....	83
3.3.4 映射.....	55	MPI_Reduce 函数.....	84
3.4 边界值问题.....	58	4.6 检测并行性能.....	86
3.4.1 简介.....	58	4.6.1 MPI_Wtime 和 MPI_Wtick 函数.....	87
3.4.2 划分.....	59	4.6.2 MPI_Barrier 函数.....	87
3.4.3 通信.....	59	4.7 本章小结.....	88
3.4.4 聚集与映射.....	60	4.8 主要术语.....	89
3.4.5 分析.....	60	4.9 参考文献.....	89
3.5 找出最大值.....	60	4.10 练习题.....	89
3.5.1 简介.....	60		
3.5.2 划分.....	61	<b>第 5 章 Eratosthenes 筛法</b> .....	93
3.5.3 通信.....	61	5.1 概述.....	93
3.5.4 聚集与映射.....	64	5.2 串行算法.....	93
3.5.5 分析.....	65	5.3 并行性的来源.....	94
3.6 n-body 问题.....	65	5.4 数据分解方法.....	95
3.6.1 简介.....	65	5.4.1 交叉数据分解.....	95
3.6.2 划分.....	65	5.4.2 按块数据分解.....	95
3.6.3 通信.....	66	5.4.3 用于按块分解的宏.....	96
3.6.4 聚集与映射.....	67	5.4.4 局部下标还是全局下标.....	97
3.6.5 分析.....	67	5.4.5 块分解的结果.....	97
3.7 增加数据输入.....	68	5.5 开发并行算法.....	97
3.7.1 简介.....	68	函数 MPI_Bcast.....	98
3.7.2 通信.....	69	5.6 并行筛法算法的分析.....	99
3.7.3 分析.....	69	5.7 并行程序的说明.....	99
3.8 本章小结.....	70	5.8 测试.....	104
3.9 主要术语.....	70	5.9 改进.....	105
3.10 参考文献.....	71	5.9.1 删除偶数.....	105
3.11 练习题.....	71	5.9.2 消除广播.....	106
<b>第 4 章 消息传递编程</b> .....	74	5.9.3 循环的重新组织.....	106
4.1 概述.....	74	5.9.4 测试.....	106
4.2 消息传递模型.....	74	5.10 本章小结.....	108
4.3 MPI 接口.....	76	5.11 主要术语.....	108
4.4 电路可满足性问题.....	76	5.12 参考文献.....	108
4.4.1 MPI_Init 函数.....	80		



5.13 练习题	108	8.4 矩阵按行分解	145
<b>第 6 章 Floyd 算法</b>	111	8.4.1 设计与分析	145
6.1 概述	111	8.4.2 复制分块的向量	146
6.2 全点对最短路径问题	111	8.4.3 函数 MPI_Allgather	147
6.3 运行时创建数组	112	8.4.4 被复制向量的输入/输出	149
6.4 设计并行算法	113	8.4.5 编写并行程序	149
6.4.1 划分	113	8.4.6 测试	150
6.4.2 通信	114	8.5 矩阵按列分解	151
6.4.3 聚合和映射	115	8.5.1 设计与分析	151
6.4.4 矩阵的输入/输出	116	8.5.2 读取按列分解的矩阵	152
6.5 点对点通信	117	8.5.3 函数 MPI_Scatterv	153
6.5.1 函数 MPI_Send	118	8.5.4 打印输出按列分块矩阵	154
6.5.2 函数 MPI_Recv	119	8.5.5 函数 MPI_Gatherv	154
6.5.3 死锁	120	8.5.6 分发中间结果	155
6.6 并行程序的说明	121	8.5.7 函数 MPI_Alltoallv	156
6.7 分析和测试	123	8.5.8 编写并行程序	156
6.8 本章小结	124	8.5.9 测试	158
6.9 主要术语	125	8.6 棋盘式分解	159
6.10 参考文献	125	8.6.1 设计与分析	159
6.11 练习题	125	8.6.2 创建通信域	162
<b>第 7 章 性能分析</b>	128	8.6.3 函数 MPI_Dims_create	162
7.1 概述	128	8.6.4 函数 MPI_Cart_create	163
7.2 加速比和效率	128	8.6.5 读取棋盘式矩阵	163
7.3 Amdahl 定律	130	8.6.6 函数 MPI_Cart_rank	164
7.3.1 Amdahl 定律的局限	131	8.6.7 函数 MPI_Cart_coords	165
7.3.2 Amdahl 效应	132	8.6.8 函数 MPI_Comm_split	165
7.4 Gustafson-Barsis 定律	132	8.6.9 测试	166
7.5 Karp-Flatt 量度	134	8.7 本章小结	167
7.6 等效指标	136	8.8 主要术语	168
7.7 本章小结	139	8.9 参考文献	168
7.8 主要术语	140	8.10 练习题	169
7.9 参考文献	141	<b>第 9 章 文档分类</b>	173
7.10 练习题	141	9.1 概述	173
<b>第 8 章 矩阵向量乘法</b>	143	9.2 并行算法设计	173
8.1 概述	143	9.2.1 划分与通信	174
8.2 串行算法	143	9.2.2 聚集和映射	174
8.3 数据分解方式	144	9.2.3 管理者/工人模式	174
		9.2.4 管理进程	175

9.2.5 MPI_Abort 函数	176	10.4.3 拒绝法	202
9.2.6 工人进程	177	10.5 应用示例	204
9.2.7 建立一个只有工人的通信域	178	10.5.1 中子输运	204
9.3 非阻塞通信	179	10.5.2 二维板上一个点的温度	206
9.3.1 管理进程的通信	180	10.5.3 二维易辛模型	207
9.3.2 MPI_Irecv 函数	180	10.5.4 房间分配问题	209
9.3.3 MPI_Wait 函数	180	10.5.5 车库停车问题	212
9.3.4 工人的通信	180	10.5.6 交通环路	213
9.3.5 MPI_Isend 函数	181	10.6 本章小结	216
9.3.6 MPI_Probe 函数	181	10.7 主要术语	216
9.3.7 MPI_Get_count 函数	181	10.8 参考文献	217
9.4 文档分类的并行程序	181	10.9 练习题	218
9.5 算法改进	187	<b>第 11 章 矩阵乘法</b>	220
9.5.1 按组分配文档	187	11.1 概述	220
9.5.2 流水线处理	187	11.2 矩阵相乘的串行算法	220
9.5.3 MPI_Testsome 函数	189	11.2.1 基于行的迭代算法	220
9.6 本章小结	189	11.2.2 基于块的递归算法	222
9.7 主要术语	190	11.3 行块分解并行算法	224
9.8 参考文献	190	11.3.1 确定原始任务	224
9.9 练习题	190	11.3.2 聚合	224
<b>第 10 章 蒙特卡洛法</b>	193	11.3.3 通信和进一步的聚合	225
10.1 概述	193	11.3.4 分析	226
10.1.1 为什么蒙特卡洛法能奏效	195	11.4 Cannon 算法	227
10.1.2 蒙特卡洛法与并行计算	196	11.4.1 组合	227
10.2 串行随机数生成器	196	11.4.2 通信	228
10.2.1 线性同余法	197	11.4.3 分析	229
10.2.2 滞后形斐波那契生成器	197	11.5 本章小结	230
10.3 并行随机数产生器	198	11.6 主要术语	231
10.3.1 管理者-工人方法	198	11.7 参考文献	231
10.3.2 蛙跳方法	198	11.8 练习题	231
10.3.3 序列分割	199	<b>第 12 章 线性方程组求解</b>	233
10.3.4 参数化	199	12.1 概述	233
10.4 其他的随机数分布	200	12.2 基本术语	233
10.4.1 逆分布累积分布函数变换	200	12.3 回代法	234
10.4.2 Box-Muller 变换	201	12.3.1 串行算法	234
		12.3.2 面向行的并行算法	236
		12.3.3 面向列的并行算法	236

12.3.4 对比 .....	237	14.1 概述 .....	271
12.4 高斯消去法 .....	237	14.2 快速排序 .....	271
12.4.1 串行算法 .....	237	14.3 并行快速排序算法 .....	272
12.4.2 并行算法 .....	239	14.3.1 排序完毕的定义 .....	273
12.4.3 面向行的算法 .....	240	14.3.2 算法开发 .....	273
12.4.4 面向列的算法 .....	242	14.3.3 分析 .....	273
12.4.5 对比 .....	242	14.4 超级快速排序 .....	274
12.4.6 面向行的流水线算法 .....	243	14.4.1 算法描述 .....	274
12.5 迭代法 .....	244	14.4.2 等效分析 .....	275
12.6 共轭梯度法 .....	247	14.5 规则取样并行排序 .....	277
12.6.1 串行算法 .....	247	14.5.1 算法描述 .....	277
12.6.2 并行算法 .....	249	14.5.2 等效分析 .....	277
12.7 本章小结 .....	250	14.6 本章小结 .....	279
12.8 主要术语 .....	251	14.7 主要术语 .....	280
12.9 参考文献 .....	251	14.8 参考文献 .....	280
12.10 练习题 .....	252	14.9 练习题 .....	280
<b>第 13 章 有限差分方法 .....</b>	<b>254</b>	<b>第 15 章 快速傅立叶变换 .....</b>	<b>283</b>
13.1 概述 .....	254	15.1 概述 .....	283
13.2 偏微分等式 .....	255	15.2 傅立叶分析 .....	283
13.2.1 偏微分方程的分类 .....	255	15.3 离散傅立叶变换 .....	285
13.2.2 差分商 .....	256	15.3.1 离散傅立叶逆变换 .....	286
13.3 弦振荡问题 .....	257	15.3.2 应用示例: 多项式乘法 .....	286
13.3.1 导出方程 .....	257	15.4 快速傅立叶变换 .....	288
13.3.2 串程序 .....	259	15.5 并行程序设计 .....	291
13.3.3 并行程序设计 .....	260	15.5.1 分割与通信 .....	291
13.3.4 等效分析 .....	261	15.5.2 聚合与映射 .....	292
13.3.5 冗余计算 .....	262	15.5.3 等效分析 .....	292
13.4 稳定状态热量分布问题 .....	263	15.6 本章小结 .....	294
13.4.1 方程的导出 .....	263	15.7 主要术语 .....	294
13.4.2 串程序导出 .....	264	15.8 参考文献 .....	294
13.4.3 并行程序设计 .....	265	15.9 练习题 .....	294
13.4.4 等效分析 .....	266	<b>第 16 章 组合搜索 .....</b>	<b>296</b>
13.4.5 实现细节 .....	267	16.1 概述 .....	296
13.5 本章小结 .....	267	16.2 回溯搜索 .....	297
13.6 主要术语 .....	268	16.2.1 示例 .....	297
13.7 参考文献 .....	268	16.2.2 时间和空间复杂性 .....	298
13.8 练习题 .....	269	16.3 并行回溯算法 .....	298
<b>第 14 章 排序 .....</b>	<b>271</b>	16.4 分布式终止检测 .....	302

16.5 分支定界法.....	304	17.6 归约操作.....	332
16.5.1 示例.....	304	17.7 性能改善.....	333
16.5.2 串行算法.....	306	17.7.1 循环转化.....	333
16.5.3 分析.....	308	17.7.2 条件执行循环.....	334
16.6 并行分支定界法.....	308	17.7.3 循环调度.....	335
16.6.1 存储和共享待解的 子问题.....	308	17.8 更普遍的数据并行.....	336
16.6.2 效率.....	309	17.8.1 parallel 编译指导语句.....	338
16.6.3 停机条件.....	309	17.8.2 omp_get_thread_num 函数.....	339
16.7 搜索博弈树.....	312	17.8.3 omp_get_num_threads 函数.....	340
16.7.1 最大最小算法.....	312	17.8.4 编译指导语句 for.....	340
16.7.2 Alpha-Beta 剪枝.....	313	17.8.5 single 编译指导语句.....	342
16.7.3 Alpha-Beta 剪枝法 的改进.....	315	17.8.6 nowait 子句.....	342
16.8 并行 Alpha-Beta 搜索.....	316	17.9 功能并行.....	343
16.8.1 并行渴望搜索.....	316	17.9.1 parallel sections 编译 指导语句.....	343
16.8.2 并行子树估值.....	316	17.9.2 section 编译指导语句.....	344
16.8.3 分布式树搜索.....	317	17.9.3 sections 编译指导语句.....	344
16.9 本章小结.....	318	17.10 本章小结.....	345
16.10 主要术语.....	319	17.11 主要术语.....	347
16.11 参考文献.....	320	17.12 参考文献.....	347
16.12 练习题.....	320	17.13 练习题.....	347
<b>第 17 章 共享存储编程.....</b>	<b>323</b>	<b>第 18 章 融合 OpenMP 和 MPI.....</b>	<b>350</b>
17.1 概述.....	323	18.1 概述.....	350
17.2 共享存储模型.....	324	18.2 共轭梯度算法.....	351
17.3 对 for 循环的并行化.....	325	18.2.1 MPI 程序.....	351
17.3.1 parallel for 编译 指导语句.....	325	18.2.2 函数级程序轮廓刻画.....	354
17.3.2 omp_get_num_procs 函数.....	327	18.2.3 对函数 matrix_vector_ product 进行并行化.....	355
17.3.3 omp_set_num_threads 函数.....	327	18.2.4 测试程序.....	355
17.4 声明私有变量.....	328	18.3 Jacobi 方法.....	356
17.4.1 private 子句.....	328	18.3.1 MPI 程序轮廓刻画.....	356
17.4.2 firstprivate 子句.....	329	18.3.2 对函数 find_steady_state 并行化.....	357
17.4.3 lastprivate 子句.....	330	18.3.3 测试程序.....	359
17.5 临界区.....	330	18.4 本章小结.....	360
critical 编译指导语句.....	331	18.5 练习题.....	360

---

附录 A MPI 函数 .....	362	C.2.2 导致不准确结果的错误 .....	413
附录 B 工具函数 .....	393	C.2.3 组通信的优点 .....	413
B.1 MyMPI.h 头文件 .....	393	C.3 实用调试策略 .....	413
B.2 MyMPI.c 源文件 .....	394	附录 D 复数回顾 .....	415
附录 C 调试 MPI 程序 .....	412	附录 E OpenMP 函数 .....	418
C.1 概述 .....	412	参考文献 .....	420
C.2 MPI 程序常见错误 .....	412		
C.2.1 死锁错误 .....	412		



# 第 1 章 动机和历史

Well done is quickly done.

Caesar Augustus

## 1.1 概 述

你是那些认为运行速度还不够“快”的人吗？现在的计算机工作站比10年前的要快了100倍，但是计算科学家和工程师还需要更快的速度。他们已经对问题进行了大量的简化，但是在现有的计算机上仍然需要数小时、数日甚至数周的时间来完成他们的程序。

更快的计算机可以处理更加复杂的计算问题。假如你可以等待一个晚上来得到计算结果。当你的程序突然快了10倍的时候，过去一些无法忍受的计算任务现在变得可以接受了，过去需要大约一周时间的计算，现在可以用15小时左右来完成了。

当然，你可以等待CPU运行得更快。5年后的CPU将比现在快10倍（根据摩尔定理）。但另一方面，如果你可以等待5年的时间，你对速度的要求肯定没有那么高！并行计算现在是就获得更高性能的有效方法。

### 什么是并行计算

并行计算就是使用并行计算机来减少解决单个计算问题所需的时间。现在，并行计算被认为是科学家和工程师用来解决各种领域的问题的标准方法，如银河系的演变过程、气候模拟、飞行器设计以及分子动力学等。

### 什么是并行计算机

并行计算机是支持并行计算的多处理器计算机系统。多计算机（multicomputer）和集中式多处理器（centralized multiprocessors）是两种主要的并行计算机。多计算机是由多台计算机和互连网络组成的并行计算机。不同计算机上的处理器之间通过传递消息来相互通信。

相反，集中式多处理器（也称为对称多处理器系统，SMP 或 symmetrical mutltiprocessor）是集成的更加紧密的系统。系统中的所有CPU共享全局内存，并通过共享内存支持处理器之间的通信和同步。

在第2章，我们将研究集中式多处理器、多计算机以及其他类型的并行计算机。

### 什么是并行程序设计

并行程序设计是使用程序设计语言显式地说明计算中不同部分如何在不同处理器上同时执行。在本章的最后，我们将讨论各种并行程序设计语言。

## 并行程序设计真的有必要吗

人们投入了很多的精力来研究自动并行化技术, 目标是能够自动将 Fortran77 或 C 语言程序转换成在拥有大量处理器的并行计算机上可执行的高效并行代码。这是一个非常困难的问题, 尽管出现了一些试验性的并行化编译器, 但是迄今为止, 还没有出现任何成熟的商业产品。因此, 我们只能自己编写并行程序。

## 为什么应该使用 MPI 或 OpenMP 编程

MPI (Message Passing Interface, 消息传递接口) 是消息传递库的标准。几乎所有的并行计算机都支持该标准通信库。如果你希望在集群系统上运行 MPI 的话, 也可以使用免费的 MPI 库。如果使用 MPI 开发程序, 当你可以使用更新、更快的系统的时候, 可以重用该程序而无需重新编写。

并行计算机正越来越多地使用 SMP 系统来构建。在每个 SMP 系统内部, CPU 共享全局地址空间。尽管 MPI 是在不同 SMP 系统之间进行通信的良好方式, 但 OpenMP 在描述单个 SMP 节点内部的处理器之间通信上更加有效。在第 18 章, 我们将会看到, 在实际应用程序中采用 MPI/OpenMP 混合编程, 可以获得比仅使用 MPI 编程更好的性能。

通过本书的学习, 你可以学到一些关于并行计算机硬件的知识, 以及很多并行程序设计策略方面的知识, 包括并行算法设计和分析, 程序实现和调试, 以及测试和优化程序的方法。

# 1.2 现代科学方法

经典科学是基于观察、理论和实际实验的。通过对现象的观察, 总结出假说。科学家发展出理论来解释现象, 并设计实验来对理论进行测试。通常, 实验的结果会导致科学家对理论进行修正, 或是完全放弃。于是, 观察又成为中心的步骤。

经典科学的特征是模型和实际实验。例如, 很多物理学的学生利用纸带、滑块和气垫导轨研究过质量、力和加速度之间的关系。物理实验允许在实践中检验理论 (比如牛顿第一运动定律)。

现代科学的特征是模型、理论、实验和数值模拟。科学家们经常无法通过实际实验来验证理论, 因为实验过于昂贵、过于耗时或是根本无法实现。因此, 数值模拟成为了科学家们日益重要的工具, 如图 1.1 所示。科学家使用数值模拟来实现模型, 比较数值模拟的行为并从自然界得到的数据。科学家可以利用这些数据的差别来修订理论或是进行更多的观察。

许多重要的科学问题是非常复杂, 需要功能非常强大的计算机来进行数值模拟。这些复杂的问题, 经常视作为科学上的重大挑战问题。它们可以分为以下几类:

- (1) 量子化学、统计力学和相对论物理学;
- (2) 宇宙学和天体物理学;
- (3) 计算流体力学和湍流;
- (4) 材料设计和超导;

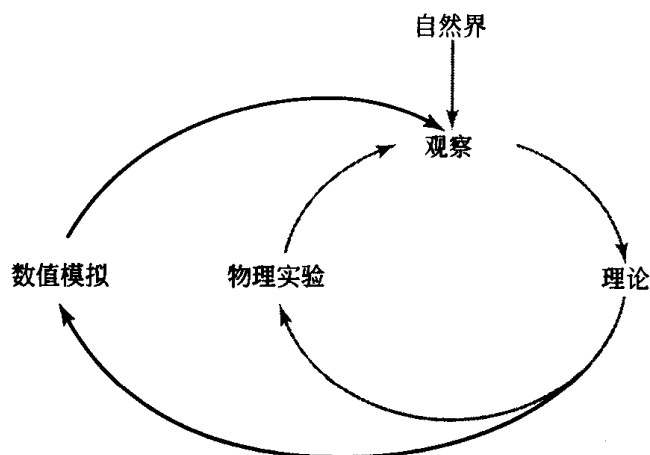


图 1.1 数值模拟方法的引入区分了经典科学方法和现代科学方法

- (5) 生物学、制药研究、基因组序列分析、基因工程、蛋白质折叠、酶活动和细胞建模；
- (6) 药物、人类骨骼和器官建模；
- (7) 全球天气和环境建模。

尽管这些重大挑战问题是在 20 世纪 80 年代才作为高性能计算进一步发展的推动力而出现的，但你也可以把整个电子计算的历史看作对高性能的追求的历史。

## 1.3 超级计算的进化

美国政府在开发和使用高性能计算机方面扮演了重要的角色。在第二次世界大战期间，美国军队为了加速弹道表的计算而投资研制了 ENIAC。在二战后的 30 年里，美国政府使用高性能计算机来设计核武器、破译密码，并进行其他与国家安全相关的应用。

超级计算机是所能建造的、功能最强大的计算机（随着计算机速度的增加，成为“超级计算机”的条件也会随之提高）。术语“超级计算机”是 1976 年随着 Cray-1 计算机的出现而得到广泛使用的。Cray-1 是流水线向量处理器，而不是多计算机系统，但是其计算能力可以超过每秒 1 亿次浮点运算。

超级计算机通常需要 1000 万美元以上。高成本一度意味着超级计算机几乎只能在政府所属的研究机构中找到，比如美国政府的 Los Alamos 国家实验室。

但是，随着时间的推移，超级计算机逐渐开始在政府部门以外出现了。20 世纪 70 年代末，超级计算机在资本密集型工业中开始得到应用，石油公司使用超级计算机寻找石油，汽车生产商开始使用超级计算机来改进其产品的燃油效率和安全性。

10 年后，全球数以百计的公司使用超级计算机来支持其业务，原因很简单：对很多业务来说，更快的计算能力意味着竞争优势。更快的碰撞模拟实验可以减少汽车制造商设计一辆新车所用的时间；更快的药物设计可以增加医药公司所持有的专利数。高速计算机甚至用于设计类似于一次性尿布这样的日常用品！

在过去 50 年中，计算速度有了飞速的增长。ENIAC 每秒可以执行约 350 次乘法，今天的超级计算机则要快上 10 亿倍，能够每秒执行数万亿次浮点运算。

单处理器的速度比 50 年前增长了大约 100 万倍。主要的性能增长归功于处理器时钟

频率的增加,使得单个操作能够更快地完成。其余的速度增长则要归功于更高的系统并行性:允许系统同时执行多个操作。计算机发展史的标志就是上面这两个方面的快速进步。以 Intel Pentium 4 CPU 为例,其时钟频率已经达到了 1 GHz,两个算术-逻辑单元以两倍核心处理器时钟频率的速度工作,并包含了必要的硬件以支持指令的乱序推测执行。

如果单处理器的性能仅提高了 100 万倍,现在的超级计算机是如何比 ENIAC 快上 10 亿倍的呢?答案非常简单:剩余的上千倍速度提高是通过把数千个处理器集成到一套计算机系统中实现的,即并行计算机。

因此,超级计算机的含义随着时间的推移发生了变化。1976 年,超级计算机是指诸如 Cray-1 之类的只有一个 CPU 的计算机,它拥有高性能流水线向量处理器和高速存储系统。今天,超级计算机是指拥有数千个处理器的并行计算机。

微处理器的发明是一个分水岭,引发了传统小型计算机和大型计算机的消亡,并促进了低成本并行计算机的开发。从 20 世纪 80 年代中期以来,微处理器厂商在基本维持价格不变的情况下,其最高端的处理器的性能每年大约增长 50%【90】。这种快速增长彻底改变了计算的面貌。基于微处理器的服务器代替了传统的基于门阵列或现成逻辑的小型计算机,甚至连大型机系统也开始基于微处理器构建。

## 1.4 现代并行计算机

只有在 20 世纪 70 年代晚期超大规模集成电路(VLSI)出现后,并行计算机才开始对广泛的用户变得有吸引力了。像 Cray-1 这样的超级计算机对于大多数组织来说过于昂贵。试验性的并行计算机要便宜一些,但仍然较贵,而且可靠性较差。VLSI 技术允许计算机设计师使用较少的芯片数,从而使得构建便宜可靠的并行系统成为可能。

### 1.4.1 Cosmic Cube 并行计算机

1981 年,加利福尼亚理工学院(Caltech)一个由 Charles Seitz 和 Geoffrey Fox 领导的研究组开始研制 Cosmic Cube 计算机(一台使用 64 个 Intel 8086 处理器构建的并行计算机)。他们选择了 Intel 8086 处理器,因为那是他们当时可以获得的惟一拥有浮点运算协处理器(Intel 8087)的处理器。完整的 64 CPU 系统在 1983 年 10 月开始运行,它展示了基于微处理器技术的并行计算的潜力。Cosmic Cube 计算机系统以 5~10 M 浮点运算的速度运行应用程序,是 DEC 公司的 VAX 11/780 速度的 5~10 倍,但其价格仅为 VAX 11/780 的一半。换句话说,这个研究组在自制的并行计算机上实现了比 VAX 高 10~20 倍的性能价格比。Cosmic Cube 也是非常可靠的,在第一年的运行中只出现过 2 次严重错误。

Intel 公司捐赠了 Cosmic Cube 所需的大部分硬件,并派其雇员 John Palmer 到加利福尼亚理工学院考察 Seitz 和 Fox 的工作。Palmer 对 Cosmic Cube 的印象极为深刻,于是他离开了 Intel 公司,创建自己的并行计算机公司 nCube。由 Justin Rattner 率领的 Intel 的第二个代表团,同样对 Cosmic Cube 印象深刻。Justin 后来成为 Intel 一个新成立部门 Intel Scientific Supercomputing 的技术负责人。

## 1.4.2 商品化的并行计算机

Bolt、Beranek 和 Newman (BBN) 以及 Denelcor 所制造的商业并行计算机在 Cosmic Cube 完成前就出现了。但是 Cosmic Cube 激发了新的制造并行计算机的浪潮。表 1.1 列出了部分并行计算机制造商。

表 1.1 在 1984~1993 年间开发商品化并行计算机的部分组织

公司	所在国家	成立时间	2001 年的状况
Sequent	美国	1984	被 IBM 收购
Intel	美国	1984	退出该行业
Meiko	英国	1985	破产
nCUBE	美国	1985	退出该行业
Parsytec	德国	1985	退出该行业
Alliant	美国	1985	破产
Encore	美国	1986	退出该行业
Floating Point Systems	美国	1986	被 Sun 公司收购
Myrias	加拿大	1987	退出该行业
Ametek	美国	1987	退出该行业
Silicon Graphics	美国	1988	正常运行
C-DAC	印度	1991	正常运行
Kendall Square Research	美国	1992	破产
IBM	美国	1993	正常运行
NEC	美国	1993	正常运行
Sun Microsystems	美国	1993	正常运行
Cray Research	美国	1993	正常运行 (更名为 Cray Inc.)

世界各地的公司都开始销售并行计算机。Intel 的 Supercomputing System Division 和一些小的创业公司, 如 Meiko、nCUBE 以及 Parsytec 处于领先地位, 一些大公司, 如 IBM、NEC 和 Sun 公司则在等待这一领域变得更加成熟。非常有趣的是, 以专用高性能流水线 CPU 闻名于世的 Cray Research 公司, 最终也于 1993 年推出了基于多处理器的并行计算机 T3D。

另外一些公司生产具有单个 CPU 和数千个 ALU 的并行计算机。其中最著名的是 Thinking Machines 公司生产的 Connection Machine。该系统于 1986 年首次交付使用, 它包括了 65 536 个 1 位的 ALU。

到 20 世纪 90 年代中期, 表 1.1 中的大部分公司或是离开了并行计算机产业, 或是破产, 或是被较大的公司兼并。尽管这个产业出现了轻度衰退, 领先的计算机制造商, 如 IBM、HP、IBM、DEC、SGI 和 Sun, 在 20 世纪 90 年代中期都有了并行计算机产品。

这些商品化的并行计算机的价格从数十万美元到数百万不等。与个人计算机相比, 并行计算机中的单位 CPU 价格更高, 原因是这些系统中包含对共享内存或处理器之间低延迟、高带宽通信网络的支持。

许多商业化的并行计算机支持高级并行程序设计语言和调试器, 但是这些并行系统底层硬件的发展太快, 使得系统程序员永远处于追赶新硬件的状态。因此, 商品化并行计算



机系统的系统编程工具非常原始, 结果是研究人员采用“最大公约数”的方法来进行并行程序设计, 一般是使用 C 语言或加上标准的并行消息通信库 (通常是 PVM 或 MPI)。生产商们集中精力在大规模商业市场, 而不是相对较小的科学计算市场。因此, 计算科学家在商业并行计算机上寻求最高性能的时候, 通常会觉得没有从生产商那里得到足够的服务, 所以他们采用了自己动手的态度。

### 1.4.3 Beowulf 系统

同时, 由于在工作和娱乐中的日益流行, 个人计算机 (PC) 成为了大宗商品市场, 其特征是性能的飞快增长和极低的利润率。成长的 PC 市场为并行计算的下一个突破搭建了舞台。

1994 年夏, 在 NASA (美国国家宇航局) 的 Goddard 宇航中心, Thomas Sterling 和 Don Becker 搭建了一台并行计算机, 系统完全使用商品化的硬件和可免费获得的软件。该系统名为 Beowulf, 包括 16 个 Intel DX4 处理器, 由多个 10Mbps 的以太网连接而成。该系统群运行的是 Linux 操作系统, 使用 GNU 编译器, 并且支持 MPI 并行程序设计, 所有这些软件都是可以免费获得的。

高性能计算研究者们很快就接受了 Beowulf 系统的哲学。在 Supercomputing'96 上, NASA 和美国能源部都展示了 Beowulf 集群系统, 其成本低于 5 万美元, 但可以在实际应用中达到 1Gflops 以上的速度。在 Supercomputing'97 上, 加利福尼亚理工学院展示了一个包含了 140 个节点的集群系统, 其求解 n-body 问题的速度超过了 10Gflops。

Beowulf 是由现成部件组建成的系统范例。与商业化的并行计算机不同, 集群系统通常不在计算速度和通信速度之间进行平衡: 与处理器速度相比, 通信网络的速度非常慢。但是, 对很多计算密集型应用来说, 集群可以有比商业化的并行计算机系统高得多得的性能价格比。最新的 CPU 通常在 PC 上最先得到应用, 而在数月后才能够商业化的并行计算机上使用, 因此可以创建使用最新 CPU 的集群系统。集群系统的另一个重大优点是入门成本较低, 这在学术机构非常受欢迎。

### 1.4.4 先进战略计算计划

同时, 美国政府建立了一个雄心勃勃的计划。该计划要建造 5 台超级计算机, 每台的造价都超过 1 亿美元。原因是布什总统于 1992 年签署了禁止地下核试验协议, 并于 1993 年由克林顿总统进行了进一步扩展。美国同时决定停止制造新的核武器。因此, 美国需要使其库存核武器超过其原有的生命周期。要保证库存核武器的安全性、可靠性和性能, 需要进行复杂的数值模拟。美国能源部的先进战略计算计划 (Advanced Strategic Computing Initiative, ASCI) 就是要开发一系列越来越快的计算系统来执行这些模拟。

这些超级计算机中的第一台 ASCI Red, 已经于 1997 年在美国 Sandia 国家实验室交付使用。该系统拥有 9 000 个 Intel Pentium II Xeon CPU, 是世界上首个实际运行速度超过 1Tflops 的超级计算机 (在交付这台系统后, Intel 退出了超级计算机行业)。ASCI 系列中的第二台系统 ASCI Blue Pacific 由 IBM 制造, 于 1998 年交付给了位于加州的 Lawrence Livermore 美国国家实验室。该系统拥有 5 856 个 PowerPC CPU, 实测计算速度超过 3Tflops。

2000 年, IBM 交付了第三台 ASCI 系统, 即 ASCI White。该系统安装在美国 Lawrence Livermore 国家实验室 (如图 1.2 所示)。ASCI White 系统实际上是由 3 个单独的系统组成, 以 SMP 系统为节点构成的多计算机。该系统包括 512 个节点, 每个节点是一套含有 16 个 PowerPC CPU 的 SMP 系统。8 192 个 CPU 的聚合速度的实测值超过了 10Tflops。

如果美国能源部保持前进的步伐, 即每 2 年将速度提高 3 倍, 那么在 2004 年将达到安装 100Tflops 系统的目标。

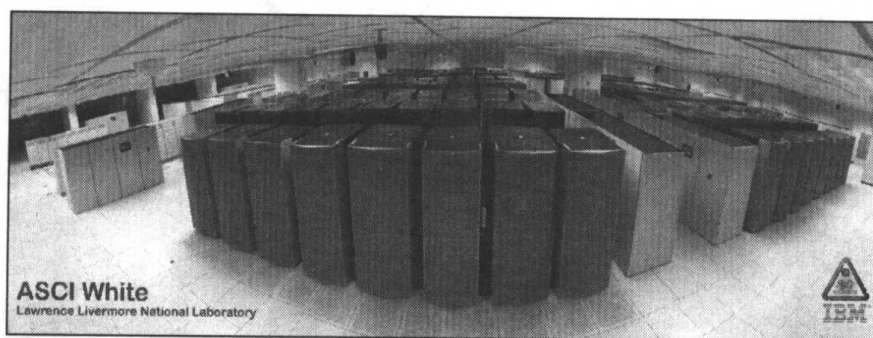


图 1.2 安装在美国 Lawrence Livermore 国家实验室的 ASCI White 超级计算机, 它拥有 8 192 个 PowerPC CPU, 并能够提供超过 10Tflops 的实测计算速度, 是 2000 年世界上最快的计算机 (图片由美国 Lawrence Livermore 国家实验室提供)

## 1.5 寻找并行性

正如我们所看到的那样, 并行计算机已经越来越容易获得。但是, 为了发挥多处理器的优点, 程序员或编译器必须能够识别出可以并行进行的操作。

### 1.5.1 数据相关图

识别并行性的形式化方法是画出数据相关图。数据相关图是一个有向图, 其中每个顶点代表一个要完成的任务, 从节点  $u$  到节点  $v$  的边表示任务  $u$  必须在任务  $v$  开始之前完成。我们称“任务  $v$  依赖于任务  $u$ ”。如果图中没有从  $u$  到  $v$  的路径, 那么这两个任务不相关, 可以同时进行。

作为一个例子, 让我们看看每周定期进行的房产地面维护工作。Allan 是 Speedy Landscape 公司的一个 8 人小组的组长, 如图 1.3 (a) 所示。他的目标是尽快完成 4 个主要任务: 割草、修剪草坪、给花园除草以及检查洒水装置。割草必须在检查洒水装置之前完成 (把 4 块草坪想象成共享变量, 它们只能在取值 “cut” 后才能取值 “wet and cut”)。同样, 修剪和除草也必须在检查洒水装置之前完成。但是, 割草、修剪和除草可以在同时进行。在工人们进入之前必须关闭安全系统, 并在离开时重新打开安全系统, 如图 1.3 (b) 所示。

Allan 了解每个任务的任务量和工人的能力, 因此他决定派 4 个工人割草, 派 2 个修剪草地, 另外 2 个在花园里除草, 如图 1.3 (c) 所示。

图 1.3 (c) 中出现了 3 种不同的任务模式。图 1.4 分别画出了这 3 种模式。圆圈内的标记代表了任务类型。具有相同标记的多个圆圈表示多个操作者进行相同操作的任务。

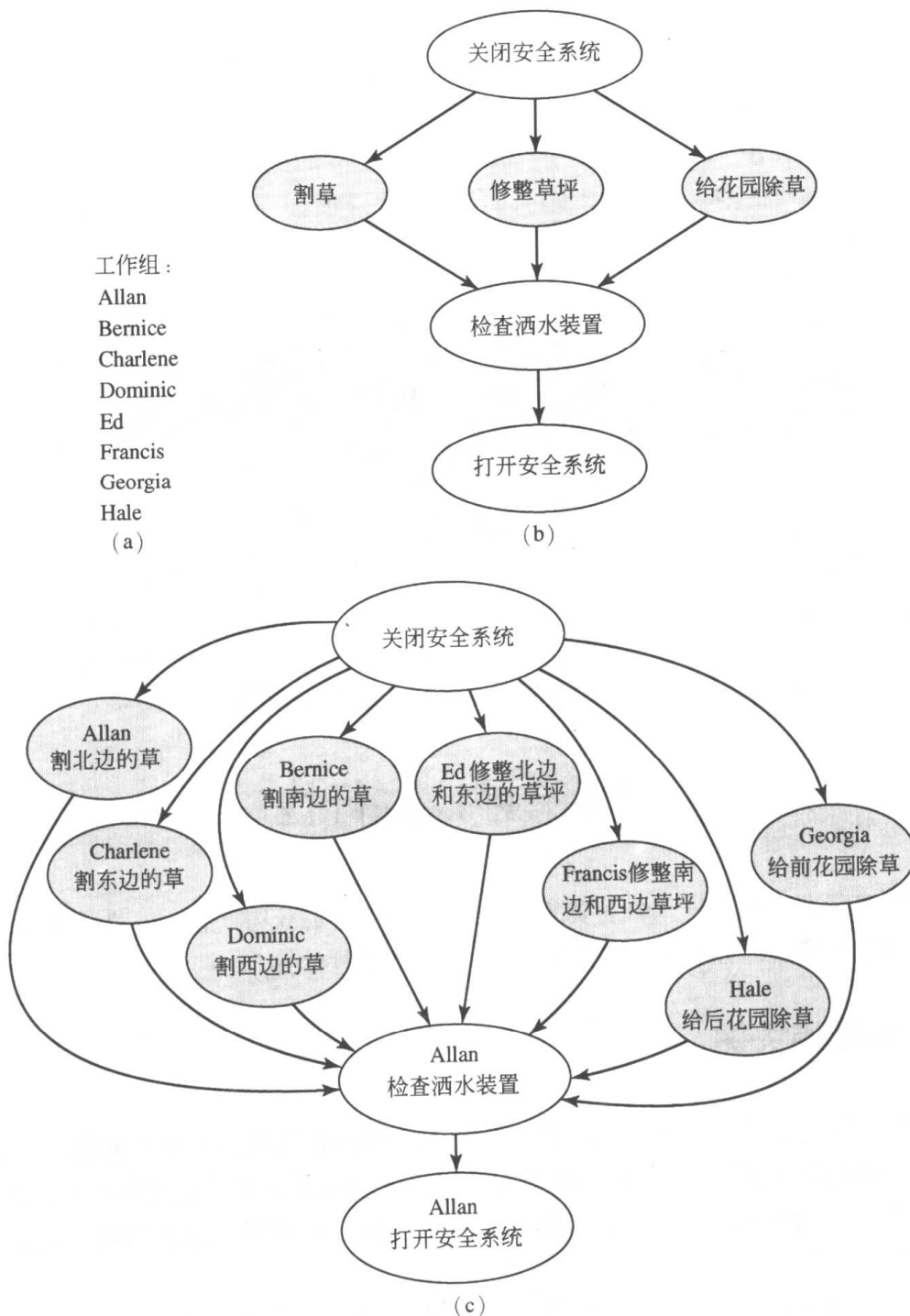


图 1.3 许多实际问题包含数据并行性、功能并行性以及任务之间的顺序限制。(a) 8 个工人的小组负责 Medici Manor 的地面维护工作；(b) 数据相关图表示了哪些任务必须要在其他任务开始之前完成，如果在节点  $u$  和  $v$  之间没有路径，说明这两个任务可以同时进行（功能并行）；(c) 较大的任务已经被分成了子任务，几个工人在不同的地方进行相同的动作（数据并行性）

## 1.5.2 数据并行性

在数据相关图中，如果有不相关的任务对数据集的不同元素进行相同的操作，我们称该数据相关图展现了数据并行性，如图 1.4 (a) 所示。

下面是一个串行程序中嵌入的细粒度数据并行性的例子：

```
for i ← 0 to 99 do
    a[i] ← b[i] + c[i]
endfor
```

相同的操作“加”在数组  $b$  和  $c$  的前 100 个元素上进行，并把结果存入了数组  $a$  的前 100 个元素。循环的 100 次迭代全部都可以同时执行。

### 1.5.3 功能并行性

在数据相关图中，如果有不相关的任务对数据集的不同元素进行不同的操作，我们称该数据相关图展现了功能并行性，如图 1.4 (b) 所示。

下面是一个串行程序中嵌入的细粒度功能并行性的例子：

```
a ← 2
b ← 3
m ← (a + b) / 2
s ← (a2 + b2) / 2
v ← s - m2
```

第 3 条和第 4 条语句，分别对  $m$  和  $s$  赋值，仅依赖于  $a$  和  $b$  的值，因此可以同时执行。

### 1.5.4 流水线

形状为简单路径或链的数据相关图，如图 1.4 (c) 所示。这意味着在仅处理单个问题的时候不存在并行性。但是，如果需要处理多个问题，且计算可以分成几个阶段，那么该计算能够支持与阶段数相同的并行性。流水线计算与一个产品装配流水线类似，如图 1.5 所示。在任何时候，每一阶段都进行计算中的一个特定部分，一个阶段的输出是下一个阶段的输入。

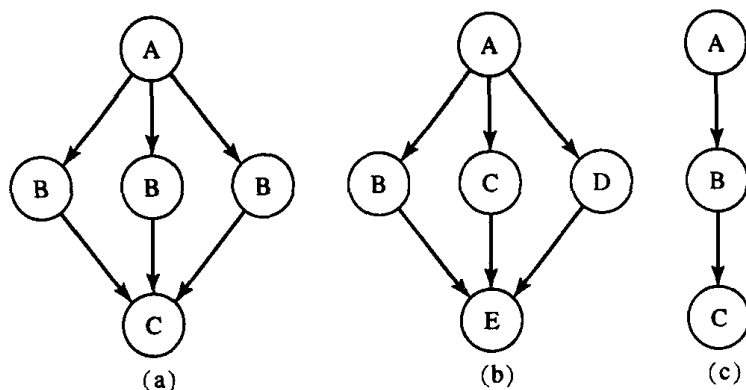


图 1.4 数据相关图中的并行性，节点代表任务，节点内的字母表示执行的操作，边表示任务间的相关性。(a) 具有数据并行性的图，不同的工人可以同时进行操作 B；(b) 具有功能并行性的图，执行 B、C 和 D 操作的任务可以同时进行；(c) 完全串行的相关图，但是如果每个任务都需要相同的时间，并且需要处理多个问题实例的时候，可以在处理第  $i$  个问题的操作 C 的同时，处理第  $i+1$  个问题的操作 B 以及第  $i+2$  个问题的操作 A。这种结构称为流水线

例如, 假如汽车需要 4 个装配阶段, 每阶段需要 1 小时。从一个空的装配线开始, 需要 4 个小时才能装配好第一辆汽车。此时, 装配线是处于满负荷状态的, 1 小时后就可以生产出第 2 辆车, 第  $k$  辆车可以在第  $k+3$  小时完成。

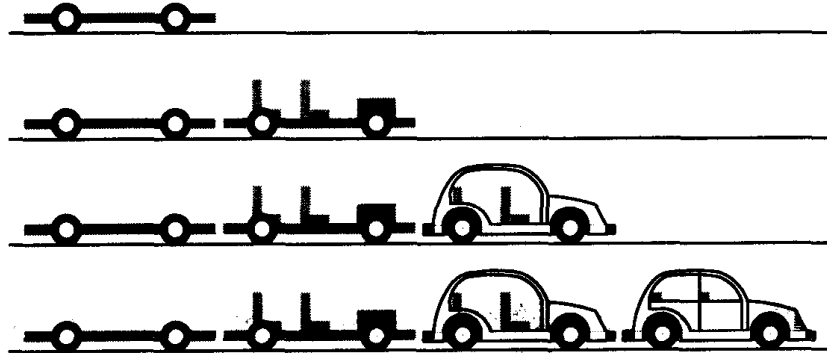


图 1.5 汽车装配生产线就是一个流水线范例

让我们考虑一个计算部分和和 for 循环的流水线实现:

$$p_0 \leftarrow a_0$$

$$p_1 \leftarrow a_0 + a_1$$

$$p_2 \leftarrow a_0 + a_1 + a_2$$

$$p_3 \leftarrow a_0 + a_1 + a_2 + a_3$$

这可以由下面的循环完成:

```
p[0] ← a[0]
for i ← 1 to 3 do
    p[i] ← p[i - 1] + a[i]
endfor
```

该循环不是数据并行的, 因为计算  $p[i]$  依赖于  $p[i-1]$  的值。但是, 我们可以将循环分成几个步骤:

```
p[0] ← a[0]
p[1] ← p[0] + a[1]
p[2] ← p[1] + a[2]
p[3] ← p[2] + a[3]
```

得到的流水线 (如图 1.6 所示) 可以用来计算多组数据的部分和。

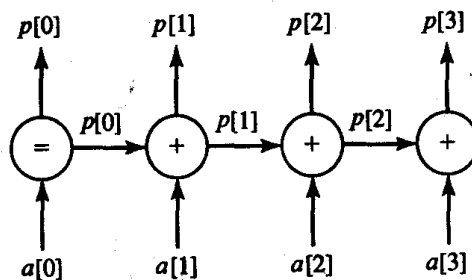


图 1.6 计算部分和的流水线。每个圆圈代表一个进程。最左边的阶段输入  $a[0]$ , 输出  $p[0]$ , 并将  $p[0]$  传递到下一阶段。所有其他阶段  $i$  输入  $a[i]$ , 并从上一处理器输入  $p[i-1]$ , 将两个输入相加, 输入到  $p[i]$

### 1.5.5 计算规模的考虑因素

上面所举出的关于数据并行性、功能并行性以及流水线的例子仅用于说明相关概念。在这些例子中所包含的操作非常少，以至于不值得在并行计算机的不同 CPU 上执行它们。在本书中，我们将寻找其他需要更多计算的问题，它们是值得使用并行计算的。

## 1.6 数据聚类

让我们考察一个计算密集的实际例子，并试图从中找到并行的机会。

现代计算机系统可以收集和保存非常大量的数据。例如，万维网包含了数以亿计的页面。人口普查数据也是一个非常巨大的数据集。使用计算机系统来找到数据中所包含的有意义的信息称为数据挖掘或科学数据分析。与 I/O 密集的“在线”数据检索相反，数据挖掘是一个计算密集的、“离线”操作。

多维的数据聚类是数据挖掘中非常重要的一项技术。数据聚类是指将数据分为由相似元素构成的组的过程。数据聚类可以让我们更容易地找到与所关心的数据项紧密相关的其他数据。

假设我们有  $N$  个文本文档。我们考察每个文档，估计其对  $D$  个不同主题的覆盖程度，并将其指定到  $K$  个组中，每个组由相似的文档构成，如图 1.7 所示。一个性能评估函数可以指出聚类的质量如何。我们的目标就是优化性能评估函数的值。

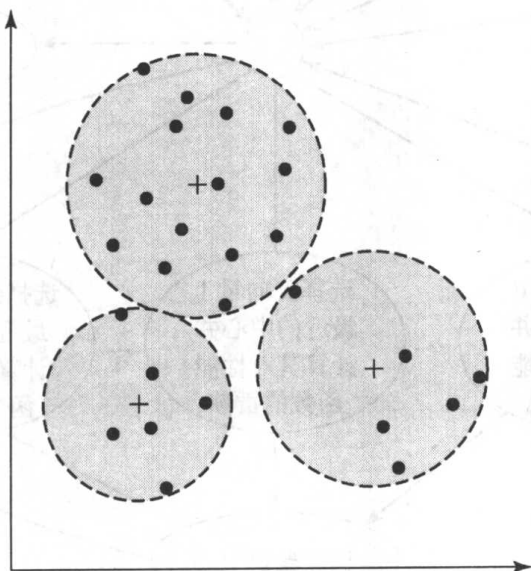


图 1.7  $N=20, D=2, K=3$  时的文档聚类示例。总共有 20 个文档（用黑点表示）。我们测量每个文档对 2 个主题的覆盖情况（因此每个文档可以由 2 维空间中的一个点来表示）。文档可以分为 3 组（以 + 为中心）。你能否找到更好的聚类方式

图 1.8 给出了一个解决数据聚类问题的串行算法的概要描述。如何使用并行性来提高算法的执行速度呢？



数据聚类:

- (1) 输入  $N$  个文档;
- (2) 对于每个文档, 根据其对  $D$  个主题的覆盖程度生成一个  $D$  维向量;
- (3) 使用随机采样得到  $K$  个中心的初始值;
- (4) 将下列步骤重复  $I$  次或性能评估函数收敛, 任一条件满足时退出循环;
  - (a) 对  $N$  个文档, 找到离其最近的中心并计算其对性能评估函数的贡献;
  - (b) 调整  $K$  个中心以改进性能评估函数的值;
- (5) 输出  $K$  个中心。

图 1.8 找到  $K$  个中心以最优地将  $N$  个文档分类的串行算法

分析的第一步是画出数据相关图。尽管我们可以将算法中的每一步映射到图中的一个节点, 但更好的方法是把算法中对每个文档或每个分组的每一个算法步骤都映射到图中的节点, 因为这种方法可以显现出更多的并行性。最终我们得到的数据相关图如图 1.9 所示。

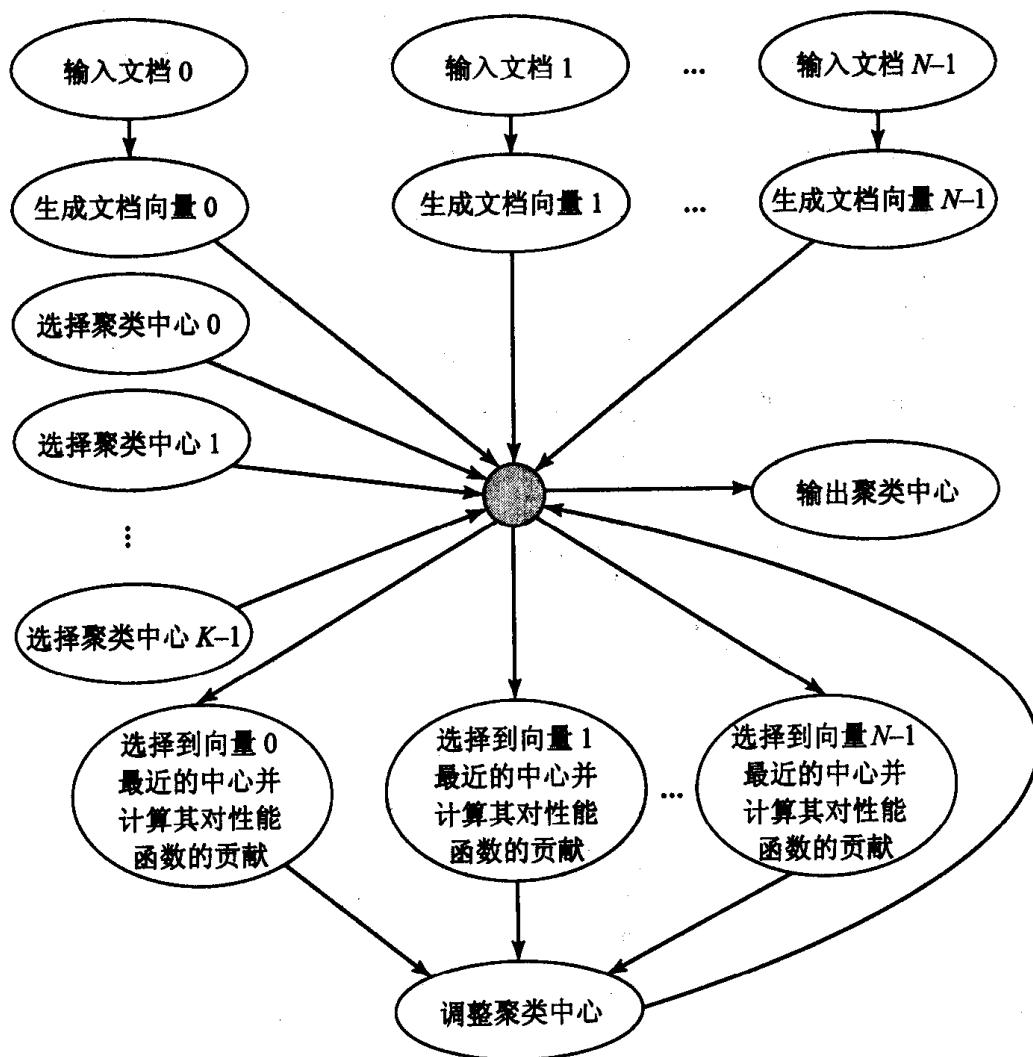


图 1.9 文档聚类算法的相关图。中间没有标签的节点是一个“空任务”，不包含任何操作。其目的是减少边的数量并使得图更加易读

一个好的数据相关图可以有助于更容易发现数据和功能并行性, 在本例中也确实如此。首先, 让我们列出数据并行性的可能性:

- 可以并行地输入每个文档;
- 可以并行地生成每个文档向量;
- 可以并行地产生数据聚类的中心;
- 可以并行地计算出离每个文档最近的中心, 以及每个文档对总性能评估函数的贡献。

然后, 我们来看看功能并行性。文档输入与向量生成、聚类中心生成是相关图中仅有的不相关节点集。因此这两项任务可以同时执行。

在识别出问题中的并行性后, 下一步是使用程序设计语言来实现算法。下面我们将介绍对并行计算机进行编程的多种方法。

## 1.7 为并行计算机编程

1988 年 McGraw 和 Axelrod 定义了为并行计算机开发应用程序的 4 条不同途径【85】:

- (1) 扩展现有的编译器以便将串行程序转化为并程序;
- (2) 扩展现有语言, 增加新的操作以允许用户表达并行性;
- (3) 在现有串行语言上增加一个并行语言层;
- (4) 定义全新的并行语言和编译系统。

让我们来考察一下这些候选方案的优缺点。

### 1.7.1 扩展编译器

解决并行编程的方法之一是开发并行化编译器, 使其能够发现和表达现有串行语言程序中的并行性。

有很多研究工作对如何并行执行函数语言程序或逻辑语言程序进行了探索。这两种语言的程序相对来说包含较多的并行性。但是吸引人们最多注意力的是命令式的 Fortran 语言。支持为 Fortran 语言开发并行化编译器的人们指出, 现存的 Fortran 代码代表了巨额的投资和人力成本。尽管并非所有这些代码都会从并行执行中受益, 但有一些组织(如美国国家能源部的国家实验室)还是希望能够提高许多复杂的 Fortran 程序的执行速度。自动并行化这些程序所节省的时间和人力使得这种方法非常有吸引力。另外, 并行编程比用 Fortran 编程更难, 这会进一步提高程序的开发成本。由于这些原因, 一些人认为让程序员继续使用简单的串行语言编写程序, 把并行化的工作留给编译器是有意义的。

在超过 20 年的时间里, 并行化编译器的开发一直是研究热点, 并出现了许多实验系统。类似于 Parallel Software 的公司已经开始出售可以将 Fortran 77 代码翻译为并行代码的编译器, 并同时支持消息传递和内存共享。

这种方法也存在批评者。例如, Hatcher 和 Quinn 指出, 使用串行命令式语言“将程序员与编译器陷入玩捉迷藏的游戏中。算法也许具有一定的内在并行性, 程序员将并行性隐藏在 DO 循环和其他控制语句的汪洋大海中, 然后让编译器必须将并行性找出来。程序员在使用传统的命令式语言的时候可能指定了不必要的串行性, 某些并行性可能无法挽回地丢失了”【49】。

对于这些顾虑的一种回答是, 允许程序员在串行程序上使用编译指导语句进行标注。这些指导语句提供了能够帮助编译器并行化程序段的相关信息。

### 1.7.2 扩展串行编程语言

与自动并行化编译器相比, 另一个保守得多的并行程序开发方法是扩展串行编程语言, 提供让用户能够创建和结束并行进程, 并提供进程同步和通信功能的函数, 还必须提供区分公共数据(所有进程共享)和私有数据(每个进程拥有一个副本)的手段。

因为只需开发一个子程序库即可, 所以扩展串行编程语言是最容易、最快捷、最便宜, 同时也是最流行(也许是由于上述原因)的并行程序设计方法。现有的语言及其编译器仍然可以原样使用。子程序库相对来说比较容易开发, 可以很快为新的并行计算机构造出来。例如, 几乎任何一种并行计算机上都能找到符合 MPI 标准的库。所以, 使用 MPI 开发的程序具有很好的可移植性。

让程序员访问到能够操纵并行处理器的底层函数, 为他们的程序开发提供了最大的灵活性。程序员可以在相同的编程环境中开发出差别很大的并行设计。

但是, 由于编译器没有参与并行代码的生成, 它无法指示程序中的错误。缺乏编译器的支持意味着程序员在开发并行程序的时候得不到帮助。其结果是并行程序令人吃惊地容易书写, 但是难于调试。

我们看看在 1988 年左右的时候, 并行程序设计先驱们的意见:

“突然间, 即使是让专门从事并行程序设计的、有经验的程序员来编写非常简单的任务, 也会不可避免地出现令人心烦、不可预测和令人迷惑的错误”(Robert Babb II)【6】。

“即使是很短的并行程序, 其行为也可能极为复杂。在特定的输入下, 一个程序可以正确地执行 1 次, 甚至 100 次, 都不能保证明天在相同的输入下能正确执行”(James R. McGraw 和 Timothy S. Axelrod)【85】。

### 1.7.3 增加并行编程层

我们可以把一个并行程序分为上下两层。下层包含计算的核心内容, 在其中进程操纵自己的数据并产生结果。一个现有的串行程序可以很好地表达这部分动作。上层控制着进程的创建和同步, 并负责将数据分布到各个进程中去。这部分动作可以用并行语言来描述(可以是可视化的编程语言)。编译器可以将这个两层的并行程序翻译为在并行计算机上执行的代码。

这种方法的两个示例是 CODE (Computationally Oriented Display Environment, 面向计算的显示环境) 和 Hence (Heterogeneous Network Computing Environment, 异构网络计算环境)。这些系统允许用户使用有向图来描述并行程序, 图中的节点代表串行过程, 边代表过程之间的相关性。

这种方法要求程序员学习和使用一种新的并行编程系统, 也许这就是为什么此方法没有能够在并行程序员中得到广泛注意的原因。尽管出现了一些研究原型系统, 据作者所知, 目前还没有基于这种方法的商业化系统。

### 1.7.4 创建一个并行语言

第4种方法是给程序员显式表示并行性的能力。

支持显式并行编程的方法之一是重新开发一个并行语言。并行编程语言 *occam* 是采用这种方法的一个著名例子。其语法与传统的命令式语言 (*imperative languages*) 有非常大的差别, 可以支持进程的并行和串行执行, 以及进程之间的自动通信和同步。

作为 Fortran 66 和 Fortran 77 的后继者, Fortran 90 是 ANSI 和 ISO 的标准程序设计语言 (在美国以外, Fortran 90 完全代替了 Fortran 77, 在美国国内, Fortran 90 被认为是一个附加标准)。Fortran 90 拥有了 Fortran 77 所没有的许多新特性, 包括数组操作。Fortran 90 可以在表达式中直接处理完整的多维数组。例如, 假定 A、B 和 C 是 10 行 20 列的数组, 我们希望把 A 和 B 相加, 把结果赋值给数组 C。在 Fortran 77 中, 我们需要写一个二重循环来完成这个操作。在 Fortran 90 中, 我们可以直接写下面的赋值语句:

```
C=A+B
```

HPF (High Performance Fortran, 高性能 Fortran) 是 Fortran 90 的扩展, 用多种方式来支持数据并程序。一个重要的扩展是 FORALL 语句, 它扩展了 Fortran 90 的数组操作, 允许程序员指定更加复杂的数据并行操作。HPF 的另一个关键功能是通过编译指导语句, 让程序员指定如何将数据映射到处理器上去。

由 Thinking Machines 公司开发的 C\* 语言, 扩展了 C 语言, 增加了 *shape* 语句, 用于指定并行数据的组织方式。*shape* 是类似于数组的对象, 但是数组里的元素只能被单独访问, 而 *shape* 中的多个元素却可以同时被操作, 例如:

```
shape [10][20] matrix;
```

建立了一个并行数据的模板。下面的声明:

```
float matrix a,b,c;
```

说明了变量 *a*、*b* 和 *c* 是 10×20 的类似于数组的对象, 可以对其进行并行操作。要把 *a* 和 *b* 相加并赋值给 *c*, 可以用:

```
with (matrix) {a = b + c; }
```

前面我们提到了对让程序员使用串行语言来编写具有内在并行性的算法, 然后让编译器来重新发现并行性的行为的可笑之处。支持显式并行性的语言改变了程序员和编译器的关系, 让他们从敌对的双方变成了同盟。

将并行语句添加到现存的程序语言中, 或是创造一个全新的语言都需要开发新的编译器。在一个语言标准已经被采用的情况下, 开发商通常还是需要若干年的时间才能在他们的并行系统上开发出高质量的编译器。

某些并行语言, 比如 C\*, 并没有被接受为标准。在这种情况下, 许多竞争厂商可能决定不在他们自己的系统上提供该语言的编译器。如果出现这种情况, 代码的可移植性会受到严重的损害。

另一个阻碍新程序语言采用的原因是用户的抵触。当新的并行语句加入到程序设计语言中的时候, 程序员必须学习如何使用这些新的语句。许多程序员并不情愿进行这种转换。

### 1.7.5 现状

尽管在并行化编译器和高层次并行编程语言方面的工作仍在继续, 但目前最流行的并行编程方法仍然是在现有串行语言上, 通过增加用函数调用或编译指导语句来表示低层语句。使用 C 语言、MPI 和 (或) OpenMP 就是采用这种方法的例子。低层并行编程可以产生具有高性能和好的移植性的并程序。与高层并行编程语言相比, 其不足之处在于编程和调试比较困难。

## 1.8 本章小结

由于国家安全因素, 美国政府在早期高性能计算的研究和开发中扮演了重要角色。后来, 以石油公司和汽车制造商为代表的资本密集型工业, 开始购买高性能计算系统。从 20 世纪 40 年代到现在, 超级计算机的性能提高了 10 亿倍, 其原因可以归结为时钟频率的提高、处理器内部并行度的提高以及使用多个处理器来更快地解决问题。现在的超级计算机是拥有成千上万个处理器的并行计算机系统。

很多并行计算机还不能叫做超级计算机。具有中等规模的并行计算机仍然可以用比单处理器系统快得多的速度来解决问题。并行计算机具有广泛的用途, 包括 Web 搜索引擎, 帮助华尔街的公司评估财务状况, 以及天气预报等。

如果某个组织能够通过减少解决问题所需的时间, 或是以相同的时间解决更复杂的问题来获得竞争优势, 那么并行计算是值得考虑的。很多问题在本质上是并行的, 应该用并行计算来解决。

商业化的并行计算机的价格通常在几十万美元到几百万美元之间。许多研究机构 and 大学正在利用商用的现成部件来构建并行计算机。尽管这些系统没有像商用并行计算机一样在计算性能和处理器间的通信方面实现很好的平衡, 但是这些系统的成本比较便宜。

尽管现在的并行计算机中的 CPU 速度是 1985 年时的 500 倍, 但并行编程环境却几乎没有什么改进。幸运的是, 出现了两个标准: MPI 和 OpenMP。MPI 支持消息传递方式的并行编程, 允许不共享内存的多个处理器合作进行并行计算。OpenMP 是一套编译指导语句, 帮助编译器生成利用 SMP 系统内多个处理器的多线程代码。MPI 和 OpenMP 还可以结合起来, 用于 SMP 集群系统的编程。

## 1.9 主要术语

ASCI  
centralized multiprocessor

先进计算计划  
集中式多处理器

COTS	商用的现成部件
data clustering	数据聚类
data dependence graph	数据相关图
data mining	数据挖掘
data parallelism	数据并行
functional parallelism	功能并行
gigaflops	每秒 10 亿次浮点运算
grand challenge problem	重大挑战问题
independent tasks	独立任务
megaflops	每秒百万次浮点运算
multicomputer	多计算机
multiprocessor	多处理器
parallel computer	并行计算机
parallel computing	并行计算
parallelize	并行化
parallel programming	并行程序设计
pipelined computation	流水线计算
scientific data analysis	科学数据分析
shape	C*中的数组的“型”
supercomputer	超级计算机
symmetrical multiprocessor (SMP)	对称多处理器
teraops	每秒万亿次浮点运算

## 1.10 参考文献

由 Kaufmann 和 Smarr 编写的“*Supercomputing and the Transformation of Science*”一书，是一本有趣易读的图书【60】。其内容是描述超级计算是如何激发新的科学发现的，书中有许多非常好的彩图。1994 年 4 月的 Communication of ACM 上的两篇文章给出了使用并行计算机来解决广泛的科学和工程问题的例子。Camp 等人描述了在美国 Sandia 国家实验室解决的各种问题【13】。Tentner 等人讨论了如何使用并行计算机来解决核反应堆设计中的各类问题【106】。Sabot 编辑的“*High Performance Computing: Problem Solving with Parallel and Vector Architectures*”【99】，给出了许多并行程序解决困难问题的例子，如天气预报，国际象棋以及抵押支持贷款（mortgage-backed financing）等。

在 Wilson 著作“*Practical Parallel Computing*”【116】的附录 B 中，读者可以看到更加详细的关于并行计算的历史。Patterson 和 Hennessy 在他们编写的教科书“*Computer Architecture: A Quantitative Approach*”【90】中也包括了关于并行计算历史的内容。

研究者们一直在探索支持对串行命令式程序的自动并行化编译技术。Bacon 等人写了一篇各种编译器优化的优化技术综述文章【7】。关于这个领域的比较新的专著是 Michael



Wolfe 的 “*High Performance Compilers for Parallel Computing*”【117】。

Kergommeaux 和 Codognet 给出了如何从逻辑程序中自动抽取并行性的技术的综述【20】。

20 世纪 90 年代初, HPF 是人们最为关注的并行程序设计语言。与其他并行编程语言不同, HPF 得到了相当数量的商业编译器的支持。想要更进一步了解该语言的读者, 请阅读 Koelbel 等人的 “*The High Performance Fortran Handbook*”【62】。

并行编程语言 Occam 值得特别注意, 因为它在某种程度上成为了 INMOS 公司的 Transputer 计算机的 “汇编语言”。Transputer 是一个为集成到并行计算机系统中而特别设计的单芯片计算机。Pountain 和 May 编写的 “*A Tutorial Introduction to Occam Programming*”【93】, 可提供关于 Occam 的进一步的信息。

Skillcorn 和 Talia 总结了并行计算的模型和语言【102】。根据所提供的抽象层次, 他们将并行程序设计模型分为 6 类, 并为每个层次的并行语言提供了使用示例。有趣的是, MPI 和 OpenMP 应该被归类于最低层, 因为所有与并行相关的部分都是显式指定的。

Jain 等人对数据聚类算法写了一篇综述【57】。

在 Moore 定律的原始公式中, Gordon Moore 观察到硅集成电路的逻辑密度与  $2^{t-1962}$  的曲线非常吻合, 其中  $t$  是年份【87】。换句话说, 密度每年增加一倍。实际的半导体位密度确实是按上述方式增长的, 到 20 世纪 70 年代末期, 密度翻倍的周期延长到了 18 个月, 此时 Moore 定律已经非常著名, 因此人们仅仅是将其进行了修正而没有重新命名。

## 1.11 练 习 题

1.1 与一些朋友进行下面的实验:

(a) 将一副牌洗乱后, 然后计算一个人要将该副牌排序为  $A\spadesuit, 2\spadesuit, \dots, K\spadesuit, A\heartsuit, 2\heartsuit, \dots, K\heartsuit, A\clubsuit, 2\clubsuit, \dots, K\clubsuit, A\diamondsuit, 2\diamondsuit, \dots, K\diamondsuit$  所需的时间?

(b)  $p$  个人给  $p$  副牌排序需要多长时间?

(c)  $p$  个人排序一副牌需要多长时间。尝试  $p=1, 2, \dots, 6$  的情况。你的算法是如何随  $p$  变化的?

1.2 假设你有一个可以在任意厚度的纸上打洞的打洞器。如果你将一张纸插入打洞器, 你会得到打了有一个洞的纸。如果在将纸插入打洞器前折叠一下的话, 你会得到有两个洞的纸。如果你只能使用打洞器一次, 为了得到有  $n$  个洞的纸, 你必须将纸折叠几次? 证明你的方法是正确的和最优的。

1.3 你需要用最快速度计算 1 000 个 4 位数的和。你手上有 1 000 个索引卡片, 每张卡片上都包括一个单独的数字, 你负责领导 1 000 个专业会计师, 每个人都有一个计算器。你可以在这些会计师中选取任意数量来为你服务。这些会计师坐在一个巨大的房间里, 每人有一张桌子。桌子的排列为 25 行 40 列。每个会计师可以把卡片传递给与她前后左右相邻的 4 个会计师。

(a) 描述一个快速的卡片分发算法。

(b) 描述将会计师所得到的中间结果累加成最终结果的快速算法。

(c) 给出能够使用的会计师的数量与总计算时间的函数关系的示意图。

(d) 如果可以使用的会计师数量为  $1, 2, \dots, 1000$ , 估计求  $10\,000$  个数的和所需的时间。将此函数关系的曲线也画在 (c) 所得到的图上。

(e) 解释为什么  $1000$  个会计师完成任务的速度不能比  $1$  个会计师快  $1000$  倍。

1.4 参考练习 1.3 中的将  $1000$  个数相加的问题, 找到另一种摆放桌子的方式以减少分发卡片和收集部分和所需的时间。描述新的桌子摆放方式, 新的通信模式以及分发卡片和收集部分和所需的时间估计。

1.5 给定一个可以分成  $m$  个子任务的任务, 每个需要  $1$  单位时间, 在  $m$  段流水线上处理  $n$  个任务需要多长时间?

1.6 一台复印机的送纸器存放着需要被复印的页面。假设送纸器在复印前需要  $5$  秒将一组新的页面装入, 在复印后需要  $10$  秒将原稿和复印的页面卸出。复印机在复印一组页面时, 第一页需要  $4$  秒, 然后每页需要  $1$  秒。当原稿长度增加时, 送纸器的最小容量是多少才能保证复印机的有效吞吐率达到每分钟  $40$  页?

1.7 今天所有的超级计算机是否都是并行计算机? 所有并行计算机是否都是超级计算机? 解释你的回答。

1.8 如果 CPU 的速度每  $5$  年增加  $10$  倍, 那么性能翻倍需要多长时间?

1.9 (a) 以 Caltech 的 Cosmic Cube 和在 Supercomputer'97 上展示的集群系统的性能为依据, 估算  $1984$  年到  $1997$  年之间的微处理器性能提高。

(b) 假设  $1984$  年到  $1997$  年之间的微处理器速度的变化速度是常数, 根据 (a) 得出的结论, 计算微处理器性能每年的提高速度?

1.10 当计算机的速度增加时, 算法的复杂性会变得更加重要。例如, 一位飞行器设计师每晚  $5$  点到第二天早上  $8$  点运行一个模拟程序。当用更快的计算机系统的时候, 可以在同样的时间内运行一个更大规模的模拟 (可以提供更加精确的结果)。假设现在的计算机可以在  $15$  个小时内解决规模为  $100\,000$  的问题。并假设执行时间是由 CPU 单独决定的, 即所有其他资源, 如 I/O 带宽和内存都不是性能的限制因素。那么在一台比现有系统快  $100$  倍的计算机上, 如果程序的时间复杂度如下所示, 那么在  $15$  小时内能够完成的问题规模是多大?

(a)  $\Theta(n)$

(b)  $\Theta(n \log_2 n)$

(c)  $\Theta(n^2)$

(d)  $\Theta(n^3)$

1.11 列举两个采用商用部件构建的集群系统与商业并行计算机相比的优点。说出一个商业并行计算机与商用部件构建的集群系统相比的优点。

1.12 对于下列每个活动, 列出完成该活动所需的任务, 并用数据相关图表示任务之间的依赖关系。每个活动至少需要包含  $6$  个任务。

(a) 建一栋房子。

(b) 做一顿意大利面条晚餐。

(c) 清扫一个公寓。

(d) 详细设计一辆汽车。

1.13 考虑图 1.10 所示的数据相关图，找出所有的数据并行性和功能并行性。

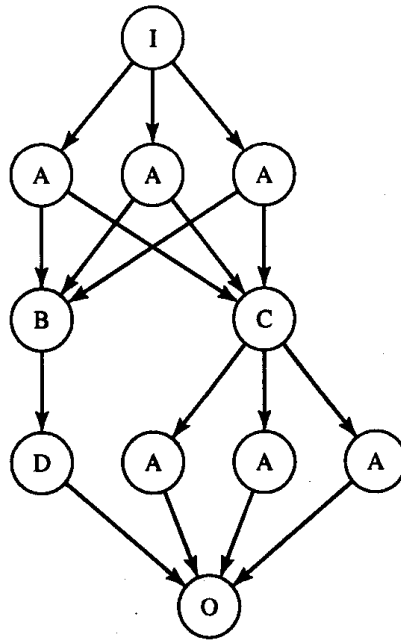


图 1.10 练习 1.13 所用的相关图

1.14 假定我们使用  $p$  个处理器来提高数据聚类算法的执行速度，对每  $N$  个文档生成  $D$  维向量。一种方法是给每个处理器预先分配  $N/p$  个文档。另一种方法是把文档放到一个列表中，然后让处理器从列表中获取文档并进行处理。讨论这两种方案的优点。

1.15 考虑本章讨论的数据聚类算法中的向量生成部分。假设执行这个步骤所需的时间是与文档大小成正比的，请提出一种将文档分配到处理器的方法，以避免给每个处理器预先分配  $N/p$  个文档，或让处理器从全局列表中获取未处理文档这两种方法的问题。

## 第 2 章 并行体系结构

What king marching to war against another king would not first sit down and consider whether with ten thousand men he could stand up to the other who advanced against him with twenty thousand?

Luke 14:31

### 2.1 概 述

从 20 世纪 60 年代早期到 90 年代中期，在差不多 30 年中，众多的科学家和工程师系统地研究了种类繁多的并行计算机体系结构。20 世纪 80 年代，并行计算机体系结构方面的发展达到了一个顶峰。当其他公司仍依赖于工作站和个人计算机中的通用 CPU 的时候，一些公司已开始利用刚刚才变得实用的超大规模集成电路（简称 VLSI）制造工艺来开发专门用于并行计算机的处理器。同时，计算机专家们正在就这样一个问题而激烈地辩论：占据主导地位的并行计算机系统应当顶多包含几十个性能强劲的处理器，还是应当包含成千上万能力稍逊的处理器呢？

今天，许多当年曾经被热烈辩论过的问题都已被解决。在现在已很难见到包含成千上万能力较差处理器的并行系统了。专门设计的处理器其性能已经无法跟上通用处理器的快速发展。因此，绝大多数目前的并行计算机都是用通用 CPU 搭建而成的。

本章中，我们将从并行系统中进行处理器连接的各种互连网络入手，对并行体系结构进行简短的概述；我们将介绍处理器阵列、多处理器和多计算机这三种在过去二十年中最为流行的并行计算机体系结构，并探讨组织一个常用集群（commodity cluster）（一种特殊的多计算机）的方法。我们将解释是什么使得一个常用集群有别于工作站网络（network of workstations）；我们还将介绍著名的有关串行和并行计算机的 Flynn 分类法，并简要介绍脉动式阵列（一种未被广泛采纳的重流水线体系结构）。

### 2.2 互 连 网 络

所有具有多个处理器的计算机都必须为处理器之间提供一种交互的方式。在一些系统中，处理器使用互连网络来访问共享内存。而在另外一些系统中，处理器使用互连网络来互传信息。本节略述两种主要的互连介质并介绍开关网络的几种流行拓扑结构。

## 2.2.1 共享介质与开关介质

并行计算机中的处理器可以通过共享介质或开关互连介质进行通信。共享介质允许在某一时刻仅仅一个消息被发送,如图 2.1 (a) 所示。通过共享介质,处理器得以广播它们的消息。每个处理器“监听”每一消息并接收发送至其上的消息。以太网便是为人所熟知的一种共享介质。

典型地,对某个共享介质访问的仲裁分散于处理器之间。在某个处理器发送消息之前,它会一直“监听”介质直到它未被任何处理器使用。这时,该处理器将尝试发送它的消息。如果两个处理器试图同时发送消息,则因为消息被混淆而必须重新发送。这两个处理器在等待一段随机的时间之后再一次尝试发送消息。消息冲突能够使一个负荷很重的共享介质的性能显著地下降。

与之相反,开关互连介质支持处理器对之间点对点的消息互传,如图 2.1 (b) 所示。每个处理器都有属于它自己的通向开关通信的一条路径。与共享介质相比,开关介质有两个重要的优势。它们支持在不同处理器对之间的多个消息并发传输,而且它们还支持互连网络的扩展,从而能够容纳更多数目的处理器。

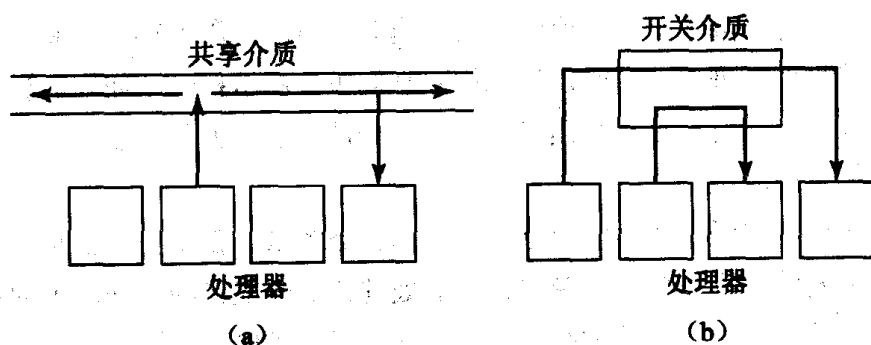


图 2.1 共享介质与开关介质的比较 (a) 任一时刻,共享介质仅仅允许一条消息被发送,每个处理器对每条消息都进行“监听”并接收目的地为它的那些消息;  
(b) 开关介质支持多个消息在不同处理器对中同时进行传送

## 2.2.2 开关网络的拓扑结构

可用图来表示一个开关网络。其中,图中的节点代表处理器和开关,图中的边代表通信路径。每个处理器都与某一开关相连,开关与处理器和其他开关相连。

在一个直接拓扑结构中,开关节点与处理器节点之间的比例为 1:1。每个开关节点都与一个处理器节点和一个或多个别的开关相连。在一个间接拓扑结构中,开关节点与处理器节点的比例大于 1:1,一些开关仅仅与别的开关相连。

我们根据实际硬件上实现高效并行算法时有助于理解其效力的原则来评价开关拓扑。这些原则是:

- 直径 一个网络的直径是两个开关节点之间的最大距离。直径小更好,这是因为直径决定了需要在随机节点对之间进行通信的并行算法复杂度的下界。
- 对分带宽 一个交叉网络的对分带宽是为了将该网络划分为两半而必须被删除

的边数的最小值。对分带宽越大越好，这是因为在需要大量数据移动的算法中，并行算法复杂度的下界为数据集的大小除以对分带宽。验证网络的对分带宽常常比走马观花的视觉观察更难，而后者可能使你相信。

- 每个开关节点的边数 最佳情况是每个开关节点的边数是一个与网络大小无关的常数。因为在这种情况下，网络更容易扩展到有大量节点的系统。
- 固定的边长 出于扩展性原因，最佳的情况为网络的节点和边能够布置于三维空间，使得最大边长是一个与网络大小无关常数。

许多开关网络的拓扑结构都被其他研究者分析过。我们将重点放在如下6种网络的拓扑结构上：二维网格、二叉树、超树、蝶形网络、超立方体以及混洗-交换网络。其中二维网格、超树、蝶形网络和超立方体都在商用并行计算机中出现过，而二叉树和洗牌-交换网络是设计领域中出现的有趣结构。

### 2.2.3 二维网格形网络

二维网格形网络(如图2.2所示)是一种直接拓扑结构，其中开关分布在一个二维的格子中，通信只能在相邻的开关之间进行。因此，网格内部的开关与四个其他开关通信。网络模型的一些变种允许网格边上的开关进行环绕连接。

让我们根据刚刚提出的四条准则来评价二维网格形网络。假设网格有 $n$ 个开关节点且没有环绕连接。当网格尽可能为正方形时，该网络的直径最小且对分带宽最大，在这种情况下，其直径和对分带宽均为 $\Theta(\sqrt{n})$ 。此时，每个开关的边数为一常数，允许我们构建任意大小的具有常数边长的网格。

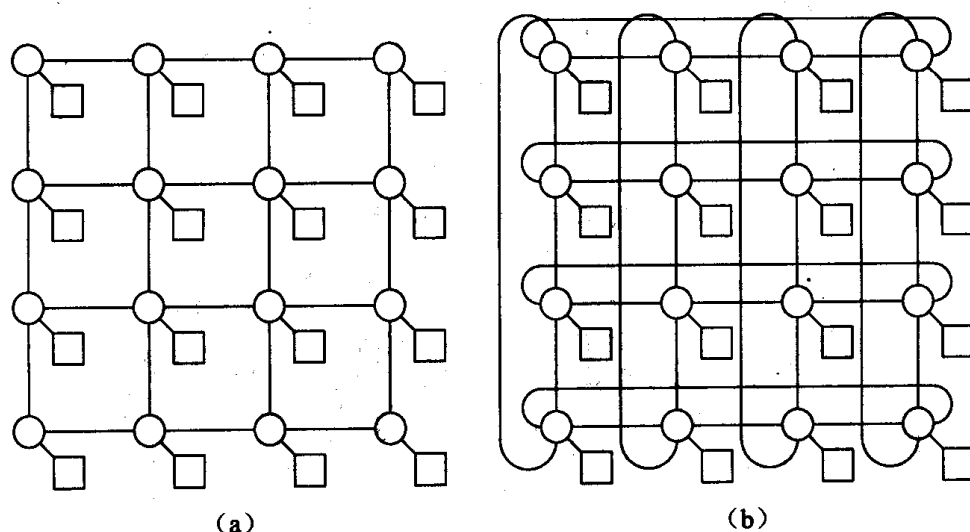


图 2.2 二维网格形网络的变形，圆圈代表开关，正方形代表处理器  
(a) 无环绕连接的情况；(b) 有环绕连接的情况

### 2.2.4 二叉树形网络

在二叉树形网络中，一个有着 $2n-1$ 个开关的二叉树支撑着 $n=2^d$ 个处理器节点间的通

信, 如图 2.3 所示。每个处理器节点与二叉树中的一叶子节点相连。因此, 二叉树网络是一个间接拓扑结构的例子, 其内部开关节点至少存在三个链接: 两个指向孩子节点, 一个指向父节点。

二叉树形开关网络具有较小的直径:  $2\log n$  (注: 在本书中,  $\log n$  意为  $\log_2 n$ ), 然而它的对分带宽是最小的, 仅为 1。假设节点分布在物理空间的话, 则在三维空间中不可能存在某个二叉树形网络开关的这样分布: 随着节点数目的增加, 最长边的长度总是小于一个特定的常数。

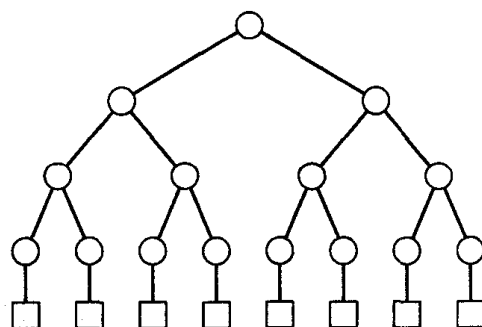


图 2.3 具有 8 个处理器节点和 15 个开关节点的二叉树形网络

## 2.2.5 超树形网络

超树为间接拓扑, 它拥有二叉树直径较小的优点且改善了其分带宽。想象一个度为  $k$  深度为  $d$  的超树形网络的最容易的方法是从两个不同的角度来观察该网络, 如图 2.4 所示。从前视图来看, 它像一个高度为  $d$  的  $k$  叉完全树, 如图 2.4 (a); 从侧视图来看, 它却像一个高度为  $d$  的倒立二叉树, 如图 2.4 (b)。将前视图与侧视图综合在一起就是一个完整的超树形网络。图 2.4 (c) 示意了一个度为 4 高为 2 的超树网络。

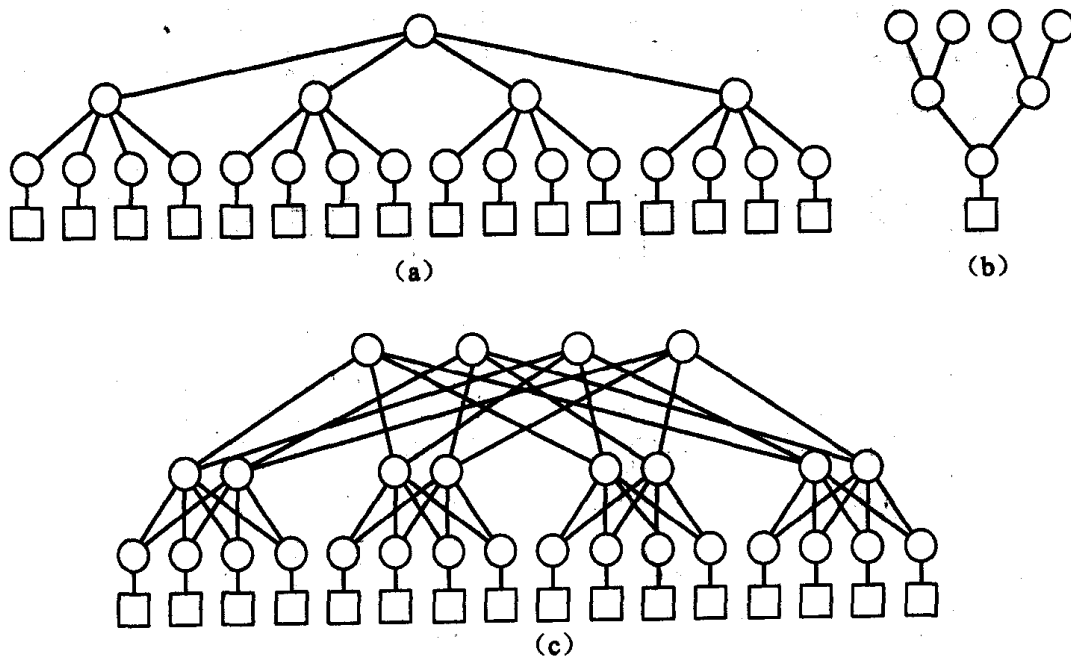


图 2.4 度为 4 深度为 2 的超树形网络。图中的圆环代表开关, 正方形代表处理器  
(a) 为前视图; (b) 为侧视图; (c) 为网络全貌

一个深度为  $d$  的 4 叉超树使用总数为  $2^d (2^{d+1} - 1)$  个交叉节点来调节  $4^d$  个处理器。该网络的直径为  $2d$ , 对分带宽为  $2^{d+1}$ 。每个开关节点的边数永远不会大于 6, 最大的边长是一个随网络大小而增加的函数。

## 2.2.6 蝶形网络

蝶形网络是一个通过  $n(\log n + 1)$  个开关节点连接  $n = 2^d$  个处理器节点的间接拓扑（如图 2.5 所示）。开关节点被划分为  $\log n + 1$  行或阶(rank)，其中每行或均含有  $n$  个节点。列序号从 0 到  $\log n$ ，尽管有时候第 0 阶和第  $d$  阶联合在一起，使得每个开关节点与别的 4 个开关节点相连。

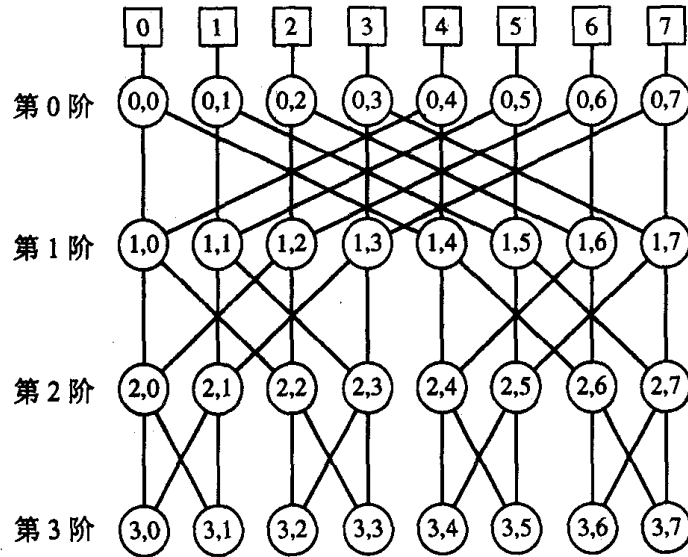


图 2.5 具有 8 个处理器节点和 32 个开关节点的蝶形网络

设  $\text{node}(i, j)$  表示第  $i$  列中的第  $j$  个节点，其中  $0 \leq i \leq d$  且  $0 \leq j \leq n$ ，则第  $i$  列（其中  $i > 0$ ）中节点  $\text{node}(i, j)$  与第  $i-1$  列中的两个节点  $\text{node}(i-1, j)$  与  $\text{node}(i-1, m)$  相连。其中  $m$  为将  $j$  的二进制表示中第  $i$  位取反的结果。比如， $\text{node}(2, 3)$  与  $\text{node}(1, 3)$  相连，同时它也与  $\text{node}(1, 1)$  相连。这是因为 1（其二进制表示为 001）是在 3 的二进制表示 011 中将第 2 个最有意义的位取反的结果。注意，如果  $\text{node}(i, j)$  与  $\text{node}(i-1, m)$  相连，则  $\text{node}(i, m)$  与  $\text{node}(i-1, j)$  相连。

无需太多的想象力，你便可以在由  $\text{node}(i, j)$ 、 $\text{node}(i-1, j)$ 、 $\text{node}(i, m)$  与  $\text{node}(i-1, m)$  连接而成的环路里看到蝴蝶的形状。蝶形网络由这样的蝴蝶图案组成，故因此而得名。随着列数的减少，这些蝴蝶翅膀的宽度呈指数增长。由于这个原因，最长网络边的长度随着网络节点数目的增加而增加。

假定在第 0 列的开关节点与  $\log n$  是相同的节点的话，则具有  $n$  个处理器节点的蝶形开关网络的直径为  $\log n$ ，其对分带宽为  $n/2$ 。

一个简单的算法使开关节点能够发送消息。每个开关节点都从消息中摘取定向位。如果定向为 0，则剩余的位将被发送至左链接；如果定向位为 1，则剩余的位将被发送至右链接。例如，假设在一个 8 处理器的网络中，第 2 号处理器想发送一个消息到第 5 号处理器，如图 2.6 所示。第 2 号处理器将目的地址 (101) 附着在消息的前端并将这个扩充后的消息插入到网络中去。第一个开关节点 ((0, 2) 节点) 接收到“101 消息”并发送“01 消息”到其右链接。第二个开关节点 ((1, 6) 节点) 接收到“01 消息”并发送“1 消息”到其左



链接。最后一个开关节点 ((2,4) 节点) 看到了“1 消息”并发送“消息”到其右链节链接。最终与第 5 号处理器相连的开关节点 ((3,5) 节点) 接收到了发自第 2 号处理器的消息。

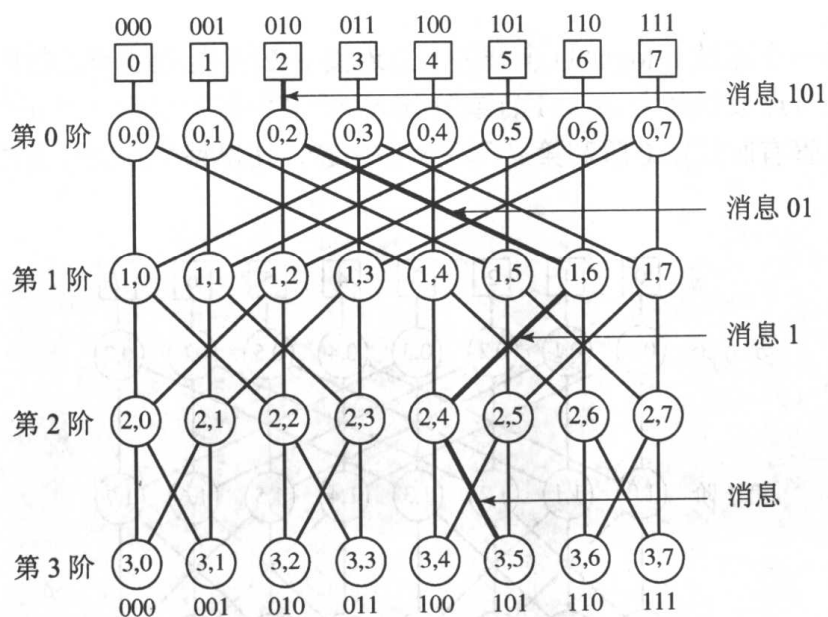


图 2.6 在蝶形网络中将一个消息从处理器 010 发送至处理器 101

## 2.2.7 超立方体网络

超立方体网络 (也称为二叉  $n$  立方体) 是一个每行开关节点都收缩为单节点的蝶形网络。一个二叉  $n$  立方体由  $n=2^d$  个处理器节点和相同数目的开关节点所组成。因此, 它是一个直接拓扑。处理器节点和与它们相连的开关节点都被标注为  $0, 1, \dots, 2^{d-1}$ ; 如果两开关的标注仅有一个二进制位存在差异, 则它们是相邻的。一个四维的超立方体如图 2.7 所示。

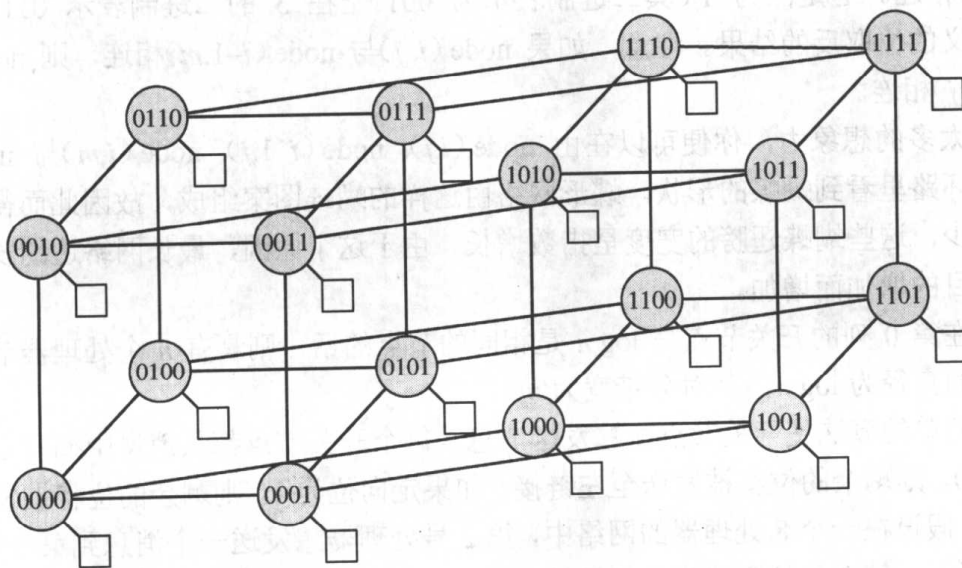


图 2.7 一个具有 16 个处理器节点和相同数目开关节点的超立方体网络, 图中圆环代表开关, 正方形代表处理器, 处理器/开关对共享相同的地址

具有  $n=2^d$  个开关节点的超立方体的直径为  $\log n$ ，对分带宽为  $n/2$ 。在不考虑另外两个评价因素的情况下，超立方体具有较小的直径和较大的对分带宽。在不计与处理器相连的边的情况下，每个开关节点的边数为  $\log n$ 。超立方体网络中最长边的长度随着网络中节点数目的增加而增加。

下面让我们探究一下如何实现超立方体网络中的消息传送。换一个角度来看图 2.7，注意边总是用来连接地址仅差一个二进制位的两个开关。例如，连接开关 0101 与开关 1101、0001、0111 和 0100 的链接。认识到这一点之后，我们便能够找到源开关与目的开关之间的一条最短路径。

假设我们想在一个四维超立方体中从开关 0101 发送一条消息至开关 0011，且地址在两个二进制位上存在差异，即在这两个开关之间最短路径的长度为 2，则存在一条如下的最短路径：

0101 → 0001 → 0011

你能从 0101 到 0011 长度为 2 的另外一条路径吗？改变对不同位取反的次序会生成一条不同的路径：

0101 → 0111 → 0011

## 2.2.8 混洗-交换网络

最后一个我们要评价的网络拓扑基于完美混洗思想。设想一副已经分好类的纸牌，如图 2.8(a)，首先将纸牌分成两等分，然后进行完美地混洗，如图 2.8(b) 所示。原始卡片次序的最终排列被称为一次完美的洗牌，如图 2.8(c) 所示。如果我们将每张牌的原始位置表示成一个二进制数的话，则每张牌的新位置就能够通过对该二进制数的一次向左轮转操作而被计算出来。换言之，我们对每一位向左移动一个位置，但最左边的那一位被旋转到最右边的位置上去。比如，牌 5 的最初位置为索引 5 (101)，轮转后的位置为索引 3 (011)。

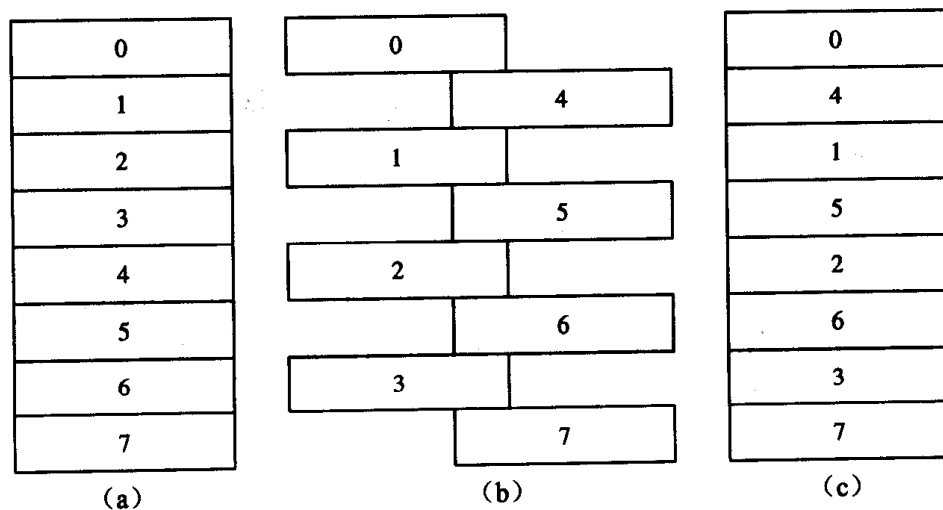


图 2.8 完美混洗的由来 (a) 一幅已经分好类的纸牌；(b) 纸牌被完美混洗；(c) 完美混洗后的结果

混洗-交换网络是一个有着  $n=2^d$  个处理器/开关对的直接拓扑。这些处理器/开关对被编号为  $0, 1, \dots, n-1$ 。在混洗-交换网络中，开关有两种连接，分别称为混洗和交换。交换连接

将序号差异存在于最低位上的开关对联系起来。每个混洗连接将开关  $i$  与开关  $j$  联系起来, 其中  $j$  是将  $i$  向左轮转一个位置后的结果。例如, 在一个 8 节点的网络中, 一个混洗连接将开关 5 (101) 与开关 3 (011) 联系起来。图 2.9 所示的是一个 16 节点的混洗-交换网络。

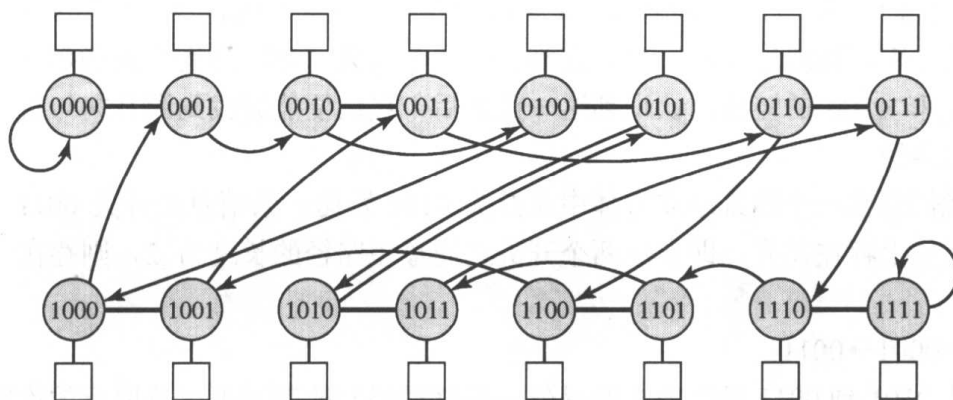


图 2.9 具有 16 个处理器节点和相同数目开关节点的一个混洗-交换网络, 圆环代表开关, 正方形代表处理器; 处理器/开关对共享相同的地址; 粗体黑线为双向交换连接, 箭头代表单向混洗连接

混洗-交换网络中的每个开关都有固定数目的边: 两个输入、两个输出 (不考虑与处理器的连接)。最长边的长度随网络大小的增加而增加。 $n$  个开关的混洗-交换网络的直径为  $2\log n - 1$ , 对分带宽约等于  $n/\log n$ 。

下面我们考虑混洗-交换网络上的寻径算法。最大长度的路径会经过  $\log n$  个交换连接和  $\log n - 1$  个混洗连接。最坏的情况是从开关 0 发送一条消息到开关  $n-1$  (反之亦然)。假定  $n=16$ , 从 0000 发送一条消息到 1111 需要  $2\log n - 1 = 7$  个步骤 (其中  $E$  和  $S$  分别指如下的交换和洗牌连接):

0000  $\xrightarrow{E}$  0001  $\xrightarrow{S}$  0010  $\xrightarrow{E}$  0011  $\xrightarrow{S}$  0110  $\xrightarrow{E}$  0111  $\xrightarrow{S}$  1110  $\xrightarrow{E}$  1111

设计这样一个算法并不难: 它总形成沿  $\log n - 1$  个混洗连接前行的路径, 但在低阶位不需被取反的情况下跳过交换连接。例如, 在下面的三次混洗连接和两次交换连接之后能够实现从 0011 发送一条消息到 0101:

0011  $\xrightarrow{E}$  0010  $\xrightarrow{S}$  0100  $\xrightarrow{E}$  0101  $\xrightarrow{S}$  1010  $\xrightarrow{S}$  0101

更完善的算法通过寻找源地址与目的地址之间的模式来减少混洗步数, 从而找到更短的路径。例如, 它能够识别出先前路径中最后的两次混洗是不必要的。

## 2.2.9 小结

表 2.1 对本节所描述的 6 种互连网络的特点进行了小结。在每方面都是最优的网络是不存在的。二维网络是仅有的一种随节点数目的增加而保持固定边长的网络, 但它也是惟一的一种无对数级直径的网络。蝶形和超立方体网络有较大的对分带宽, 但蝶形网络有  $\Theta(n \log n)$  个开关节点, 超立方体网络是仅有的一种每节点的边数不是常数的网络。混洗-

交换网络代表了设计的折中方案, 它的每个节点具有固定数目的边、较小的直径和较好的对分带宽。4-叉超树网络几乎在每个方面都要比二叉树形网络出众, 它具有很少的开关节点、小的直径和大的对分带宽。

表 2.1 6 种开关网络拓扑的属性。其中标注为“边/节点”的列指与开关节点相连的开关数目的最大值

	处理器节点数	交换网络节点数	直径	对分带宽	边/节点	边长为常数
二维网络	$n=d^2$	$n$	$2(\sqrt{n}-1)$	$\sqrt{n}$	4	是
二叉树	$n=2^d$	$2n-1$	$2\log n$	1	3	否
4-叉超树	$n=4^d$	$2n-\sqrt{n}$	$\log n$	$n/2$	6	否
蝶型网络	$n=2^d$	$n(\log n+1)$	$\log n$	$n/2$	4	否
超立方体	$n=2^d$	$n$	$\log n$	$n/2$	$\log n$	否
混洗交换网络	$n=2^k$	$n$	$2\log n-1$	$\approx n/\log n$	2	否

## 2.3 阵列处理机

向量计算机是一种指令集既包括向量也包括标量指令的计算机。一般地, 向量计算机有两种实现方式。在流水线向量计算机中, 矢量从内存流向 CPU, 流水线算术单元来执行向量操作。Cray-1 和 Cyber-205 (早期的超级计算机) 就是著名的流水线型矢量处理器。本书中我们不会再进一步讨论这些体系结构。

阵列处理机 (processor array) 是另一种向量计算机, 它由一台串行计算机连接到一些能够对不同数据执行同步操作的处理元件实现。许多早期的并行计算机发展成果促成了阵列处理机的构建。设计阵列处理机的一个原因是控制单元相对较高的成本【52】, 另一重要原因是注意到大量的科学计算是数据并程序【50】, 恰好可以由阵列处理机来实现并行性。

### 2.3.1 体系结构与数据并行

下面让我们看看通用阵列处理机的体系结构。它是由一个前端计算机控制的由简单处理元件所组成的集合, 如图 2.10 所示。

前端计算机是一个标准的单处理器。其主存保存正在执行的指令和被前端顺序处理的数据。阵列处理机被划分为许多单独的处理器-内存对。被并行处理的数据分布在这些内存中。为了执行某个并行操作, 前端计算机发射合适的指令到阵列处理机中的处理器, 它们对存储在其局部内存中的操作数同时执行这条指令。控制路径 (图 2.10 中用一条虚线来表示) 允许前端计算机将指令广播到后端处理器。

例如, 假设阵列处理机包含 1024 个处理器, 分别记为  $p_0, p_1, \dots, p_{1023}$ 。两个 1024 元素的向量  $A$  和  $B$  在处理器上分布, 处理器  $p_i$  的内存中包括  $a_i$  和  $b_i$ 。阵列处理机能够用一条指令实现  $A$  与  $B$  的向量加  $A+B$ , 这是因为每个处理器  $p_i$  可以从局部内存中取出属于自己的  $a_i$  和  $b_i$  的值, 并与所有其他处理器并行执行加法操作。

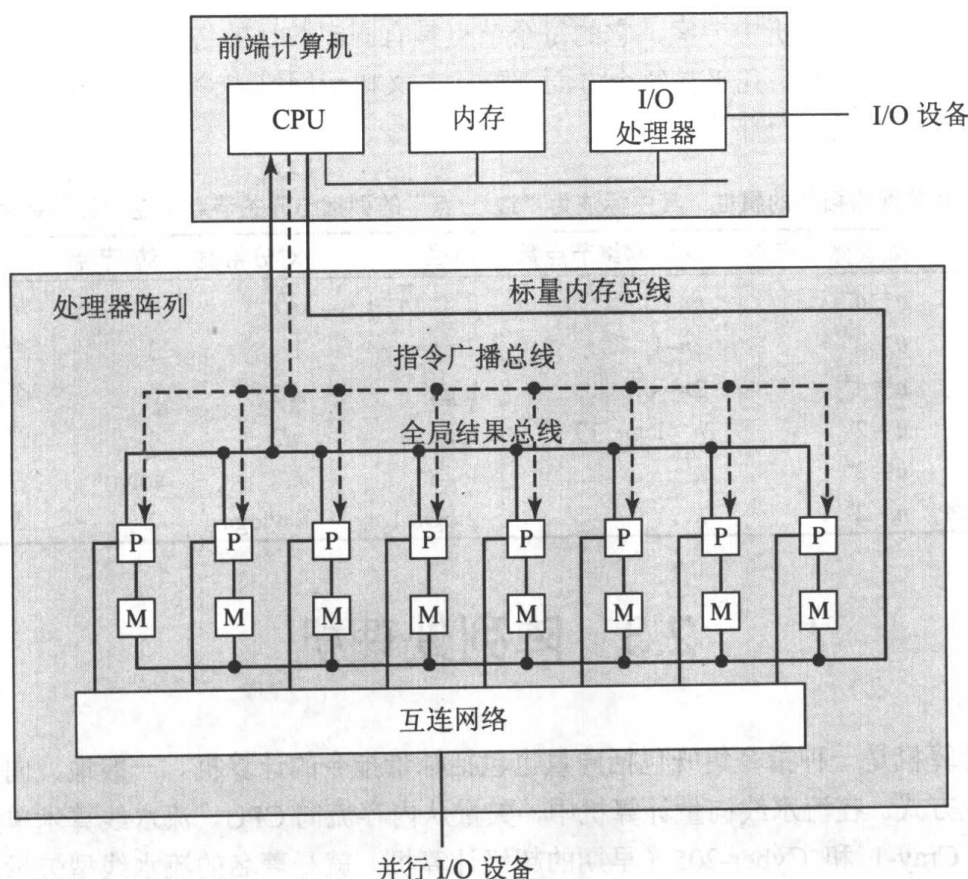


图 2.10 通用阵列处理机的体系结构。阵列处理机包含有许多基本处理器（用一个 P 标识的盒状物来表示）。每个处理器都有属于自己的内存（用一个 M 标识的盒状物来表示）

注意，将两个具有 100 个元素的向量（或任意两个长度小于等于 1 024 的向量）相加时，阵列处理机所消耗时间与将两个具有 1 024 个元素的向量相加所需的时间完全相同。在阵列处理机上执行任一特定指令所花时间与实际使用的处理器的个数无关。

如果程序员试图处理多于 1 024 个元素的向量时会发生什么呢？当向量所拥有的元素个数超过了阵列处理机中处理器的个数时，某些或所有处理器就需要存储和处理多个元素。例如，具有 10 000 个元素的向量能够被存储在具有 1 024 个处理器的系统上，其中 784 个处理器上分别存储 10 个元素，另外的 240 个处理器上分别存储 9 个元素： $784 \times 10 + 240 \times 9 = 10\,000$ 。根据体系结构和操作系统的不同，向量元素到处理器的映射可能需要由编程人员来管理，也可能无需编程人员的干预。例如，Thinking Machines 公司的 Connection Machine 的操作系统支持虚拟处理器的概念。编程人员在编写程序时，其中的向量和矩阵其元素个数可以超过阵列处理机所拥有处理器数目。将虚拟处理器映射到物理处理器的复杂工作由从前端发向阵列处理机的微码指令控制。

### 2.3.2 阵列处理机的性能

性能是表示单位时间内所能完成工作总量的一个量度。我们依据每秒执行的指令数来测量阵列处理机的性能。阵列处理机的性能依赖于其处理器的使用率，其所处理的数据结构的大小直接影响性能。当所有处理器都是活跃的且数据结构的大小是处理器个数的倍数

时, 阵列处理机的性能是最高的。

### 例 2.1

假设一个阵列处理机有 1024 个处理器。每个处理器能够在  $1\mu\text{s}$  内执行一对整数的加法操作。假设每个向量都被均衡地分配到处理器上, 则阵列处理机对两个具有 1024 个元素的整型向量相加时的性能是多少呢?

解:

被执行的整数操作的数目为 1024, 每个处理器执行一次整型加需要  $1\mu\text{s}$ 。

性能 =  $1024 \text{ 操作} / 1\mu\text{s} = 1.024 \times 10^9 \text{ 操作/s}$

### 例 2.2

假设阵列处理机有 512 个处理器, 每个处理器能够在  $1\mu\text{s}$  内执行一次整数加。假设每个向量都被均衡地分配到处理器上, 则阵列处理机对两个具有 600 个元素的整型向量相加时的性能是多少呢?

解:

被执行的整数操作的数目为 600, 由于  $600 > 512$ , 所以其中 88 个处理器必须进行两对整数的加操作, 其余 424 个处理器只进行一对整数的加操作。在其他 88 个处理器进行其第二对整数加操作的时候, 424 个处理器处于空闲状态。

性能 =  $600 \text{ 操作} / 2\mu\text{s} = 3 \times 10^8 \text{ 操作/s}$

## 2.3.3 处理器互连网络

当然, 典型的并行计算远比进行简单的两向量加要复杂得多。通常, 向量或矩阵元素的新值与其他元素的函数。例如, 在运用有限差分法来求解某个特定的差分方程的过程中:

$$a_i \leftarrow (a_{i-1} + a_{i+1}) / 2$$

处理器能够通过互连网络来传送数据, 从而将存储在不同处理器内存中的操作数集合在一起。最流行的用于阵列处理机的互连网络是二维网格形网络。除了前面提及的优点之外, 二维网格形网络有着可用 VLSI 相对直接实现的优势, 其中单个芯片就可含有大量的处理器, 如图 2.11 所示。

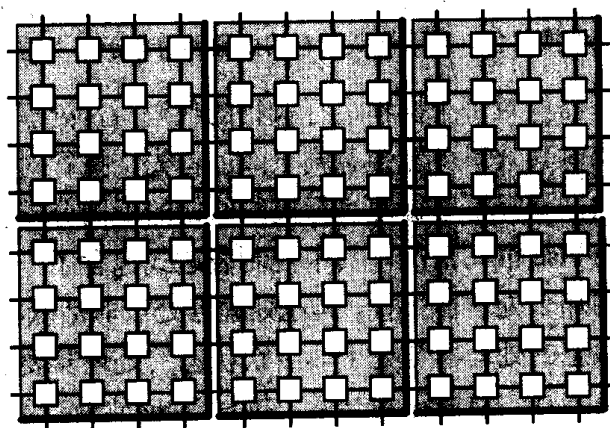


图 2.11 由二维网格形互连网络连接起来的  $8 \times 12$  阵列处理机如何安排才能使线的长度最小?  
单 VLSI 芯片含有分布在  $4 \times 4$  网格上的 16 个处理器元件。芯片按  $2 \times 3$  排列生成期望的 96 个处理器的阵列。本图仅展示互连网络, 并不包括处理器与前端计算机之间的连接



互连网络支持并发消息传递。例如,在如图 2.11 所示的二维网格形网络中,每个处理元件都能够同时将处理元件的某个值发送到它的“北边”。

### 2.3.4 处理器的启动与阻塞

处理器阵列展示了同步执行的特点,即所有处理器都步调一致地进行操作。然而,仅使用处理器的一个子集来执行某条指令是可能的。每个处理器均有掩码,通过它可使处理器不执行某些指令。当被操作数据项的数目不是处理器阵列大小的倍数时,采用掩码法是非常有用的。

掩码法使得处理器阵列能支持并行操作的条件执行。例如,假设整型向量  $A$  被分布在处理器阵列上,每个处理器分得一个元素。我们想将  $A$  中的每个非零值转化为 1,将每个零值转化为-1,如图 2.12 所示。首先,每个处理器都进行测试来看其上的  $A$  中元素的值是否为 0。如果是,则该处理器将对其掩码位进行设置,表示它不会执行下一指令。掩码位未被设置的处理器将其上的  $A$  中元素转化为 1。此时,掩码位被翻转,使得先前活跃的处理器的处理器变得不再活跃,反之亦然。现在掩码位未被设置的处理器将其上的  $A$  中元素转化为-1。最后,所有的掩码位都被抹去。

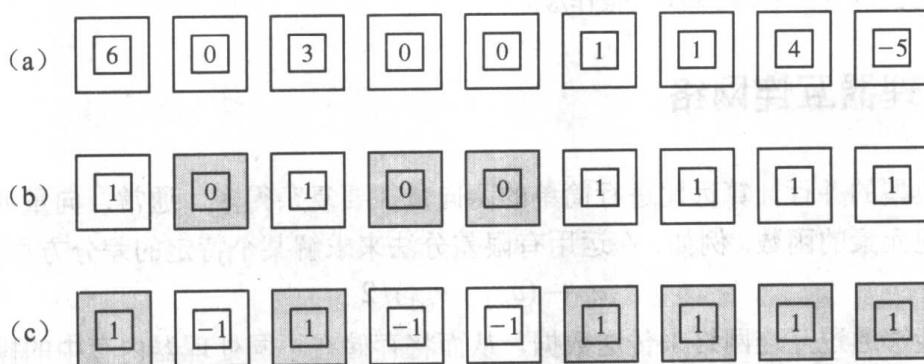


图 2.12 一条 if-then-else 语句的执行情况 (a) 具有 9 个元素的向量  $A$  分布于 9 个处理器的内存中,其中每个处理器分得一个元素。如果元素的值不为 0 的话,则将其修改为 1,否则将其修改为-1;

(b) 阴影暗示处理器的掩码位未被设置 (即处理器处于非活跃状态),这是由于这些处理器上  $A$  的元素为 0,其余处理器将其上  $A$  的元素设置为 1; (c) 处于活跃状态的处理器与处于非活跃状态的处理器切换角色。处于活跃状态的处理器将位于其上的  $A$  的元素设置为-1

虽然在执行每个处理器能对每个操作进行执行的代码时,处理器阵列是高效的,但是在程序进入条件执行代码时其效率可能会急剧下降。首先,执行对掩码位进行设置的测试会带来额外的开销;其次,不得不串行地执行控制结构的不同分支也导致效率的降低。例如,考虑一条并行 if-then-else 语句情况,其中控制表达式包含了一个并行变量。then 子句首先被其条件评判为真的那个处理器执行,其他处理器处于非活跃状态。接下来,为了 else 子句的执行,处于活跃状态的处理器与处于非活跃状态的处理器要进行角色转换。总的来说,考虑到对条件表达式进行估值所花的开销,执行 if-then-else 语句时系统的性能比在整个处理器阵列上执行并行操作的系统性能的一半还要低。

### 2.3.5 其他体系结构特点

数据通路允许前端计算机访问阵列处理机中每个内存位置。这种能力是重要的，因为它允许在串行代码中使用或定义并行变量。通过这种方式，阵列处理机可被视为前端计算机内存空间的扩展。

全局结果机制使来自阵列处理机的值能够被合并，并返回到前端计算机。计算全局“与”的能力是有价值的。例如，一些迭代程序可以持续运行直到矩阵中所有的值都已收敛。如果处理器阵列中的每个元素所对应的矩阵元素已经收敛时假定其对全局结果的贡献为1，否则为0，则当且仅当所有矩阵元素已经收敛时，处理器所有返回值的全局“与”返回1，这使得我们很容易确定程序是否应该继续下一次迭代。

### 2.3.6 阵列处理机的缺点

阵列处理机有几个明显的缺点，从而使得它们不像通用并行计算机那样具有吸引力。

(1) 许多问题都不能很好地映射为严格的数据并行解法。在阵列处理机体系结构上，这些问题不能有效地运行。

(2) 由于在某个时刻，阵列处理机仅仅能够执行一条指令，所以当程序进入条件执行并行代码时，计算机效率会下降。

(3) 阵列处理机在很大程度上是单用户系统，不容易处理多个用户要同时执行多个并程序情况。将后端处理器划分为相互独立的资源池或将每个处理器的内存划分为多个分区，需要在硬件和软件方面进行复杂地改进。

(4) 阵列处理机不适合于小规模的系统。为了让包含大量处理器的阵列处理机具有高性能，系统在前端与阵列处理机之间、处理元件之间以及处理元件与并行 I/O 设备之间需要高带宽的通信网络。当处理元件个数很多时，网络的开销可能仅仅是整个系统开销的一小部分。但是，当处理元件个数很少时，网络的开销可能占系统开销的大部分。尽管大规模的阵列处理机系统可以提供很高的性能价格比，但是具有较少处理器的入门级系统的性价比则无法与其他竞争系统相比。

(5) 由于阵列处理机使用定制的 VLSI，所以阵列处理机无法赶上通用 CPU 所表现出的性能和成本改进。随着时间的推移，生产定制处理器的公司与生产通用处理器的半导体生产厂家（例如，能够将数亿美元花费在新的芯片设计上并通过数以百万芯片的销售来分期回收投入的 Intel 公司）保持竞争会越来越困难（或根本不可能）。

(6) 建造阵列处理机的最初动机之一——控制单元相对高的价格——不再有效。在今天的 CPU 中，用于控制电路的芯片区总量相对较小。

由于上述所有原因，阵列处理机不再被认为是通用并行计算机的一种可行选择。

## 2.4 多 处 理 器

多处理器是具有共享内存的多 CPU 计算机。两个不同 CPU 上的相同地址表示相同的



内存位置。多处理器避免了阵列处理机所具有的三个问题。它们能够由通用 CPU 来构建，可以很好地支持多用户系统，并且遇到条件执行并行代码时并不会造成效率的降低。

我们将讨论多处理器的如下两种基本类型：集中式多处理器（其中的所有主存均位于同一处）和分布式多处理器（其中的主存分布于各处理器上）。

### 2.4.1 集中式多处理器

一个典型的单处理器使用总线将 CPU 与主存 I/O 处理器相连。Cache 用于减少 CPU 从内存中获取指令和数据时的等待时间，从而有助于保持 CPU 的工作效率。

集中式多处理器是单处理器的简单扩展。总线上连接附加的 CPU，所有的 CPU 共享同一主存，如图 2.13 所示。该体系结构也被称为内存一致性访问的多处理器（UMA）或对称多处理机（SMP），这是因为所有的内存都位于同一处且每个处理器访问所有内存的时间均相同。集中式多处理器是实用的，这是因为大而有效的指令和数据缓存减少了单个处理器在内存总线和内存上的负载，使得它们可以被多个处理器共享。尽管如此，由于内存总线的带宽的限制，SMP 系统的处理器的数目通常被限制在几十个。

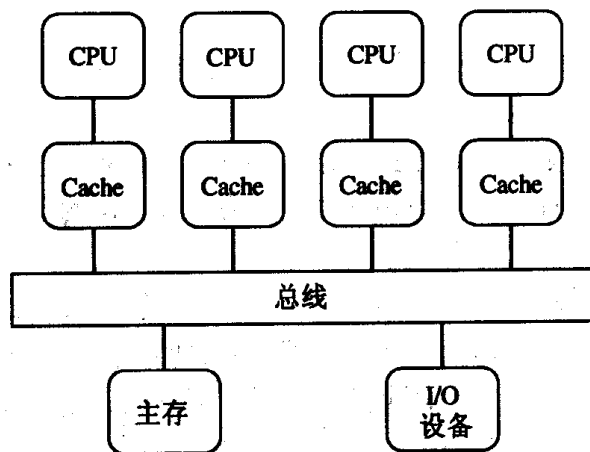


图 2.13 集中式多处理器的体系结构

私有数据是仅被一个处理器使用的数据，而共享数据则是被多个处理器所使用的数据。在集中式多处理器中，处理器通过共享数据来互相通信。例如，处理器可以一起执行链表上的所有任务。一个共享指针可以包含将被处理的下一表项的地址。每个处理器访问共享指针来确定它的下一任务并在其被另一处理器访问之前前移该指针。集中式多处理器的设计者必须解决与共享数据有关的如下两个问题：Cache 一致性问题 and 同步问题。

**Cache 一致性问题** 通过多个 Cache 对数据进行复制将减少处理器间访问共享数据时冲突情况的发生。然而，由于每个处理器对内存的观察是通过其 Cache 完成的，所以必须寻找一条途径来确保对相同的内存位置，不同处理器不会拥有不同的值。

图 2.14 展示了不同处理器如何会拥有不同的数值。在图 2.14 中，两个不同的 CPU，即 A 和 B，对同一个内存位置进行读操作，然后 CPU B 将一个新的值写回该内存位置。此刻，CPU A 的 Cache 中仍然保存着该内存位置的过时的缓存值。该现象称为 Cache 一致性问题。

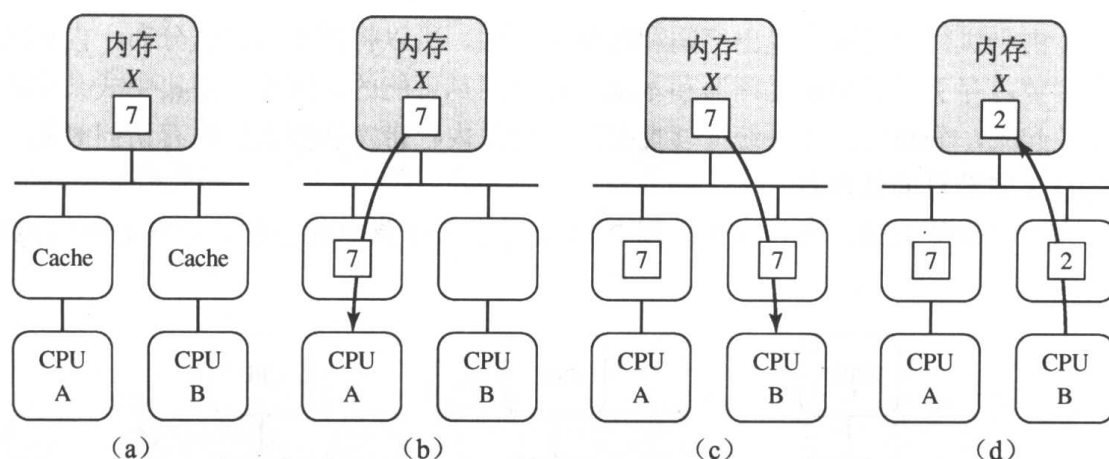


图 2.14 Cache 一致性的例子 (a) 内存位置  $X$  保存的值为 7; (b) CPU A 读  $X$ ,  $X$  的备份被存于 CPU A 的 Cache 中; (c) CPU B 读  $X$ ,  $X$  的备份被存于 CPU B 的 Cache 中; (d) CPU B 将 2 存入  $X$ , 内存位置  $X$  具有新的值, 该值在 CPU B 的 Cache 中也被修改。然而, CPU A 仍然有其 Cache 中  $X$  的一个旧值

通常使用“监听 (snoop)”协议用来保持集中式多处理器上的 Cache 一致性。每个处理器的缓存控制器对总线进行监听, 以识别哪个缓存块正被别的 CPU 请求。解决 Cache 一致性问题最为常见的方法是, 确保某个处理器在写某数据之前对其具有独占的 Cache 访问能力。在写操作发生之前, 被其他处理器缓存的该数据项的所有备份都被标为无效。此时, 该处理器执行写操作以修改在其缓存块及适当的内存位置中的值。当任何其他 CPU 试图从那个缓存块中读取某个内存位置时, 将发生一次缓存缺失, 从而迫使它从内存中重新获取修改后的值。此协议称为写无效协议 (write invalidate protocol)。

如果两个处理器同时试图向同一内存位置写值的话, 它们中仅仅有一个处理器赢得“竞争 (race)”。另一处理器的缓存块变为无效。“失败”了的处理器在能够进行其写操作之前, 它必须得到该数据的一个最新的备份 (修改后的值)。

**处理器同步** 处理器协同执行一个计算时需要多种多样的同步。互斥是“一种这样的状态, 在任何时刻最多一个处理器能够进行某一特定的行动”【116】。稍早的时候, 我们给出了多个处理器协同完成存于链表中的任务的例子。从该链表中获得下一任务并修改列表指针是需要互斥的。

阻塞式同步是常出现于共享内存程序中的另一种同步。阻塞式同步确保没有一个进程能够超越程序中设定的某个点 (该点称为“障碍点”) 而继续执行, 直至每个进程都到达该点之后。在程序执行的两阶段之间通常会使用阻塞式同步。

在大多数系统中, 执行同步功能的软件依赖于硬件支持的同步指令。在具有处理器数目较少的系统中, 最普通的硬件同步机制是不可中断指令或检索和改变某个值的一系列原子操作指令【90】。

## 2.4.2 分布式多处理器

共享内存总线的存在使得集中式多处理器中的 CPU 数目局限在几十个之内。另一种方法是将主存分布于各个处理器中, 从而生成一个访问本地内存要比访问非本地内存快很多

的系统。由于执行的程序呈现出空间和时间局部性，所以将指令和数据分布于内存之内，使得大多数内存引用发生在处理器和本地内存之间是可能的。因此，同集中式共享内存并行机比较起来，分布式内存系统能够拥有更高的总内存带宽和更低的内存访问时间，因此可以支持更多数目的处理器。

分布 I/O 也能够改善可扩展性。图 2.15 展示了一个常见的分布内存多 CPU 计算机的体系结构。

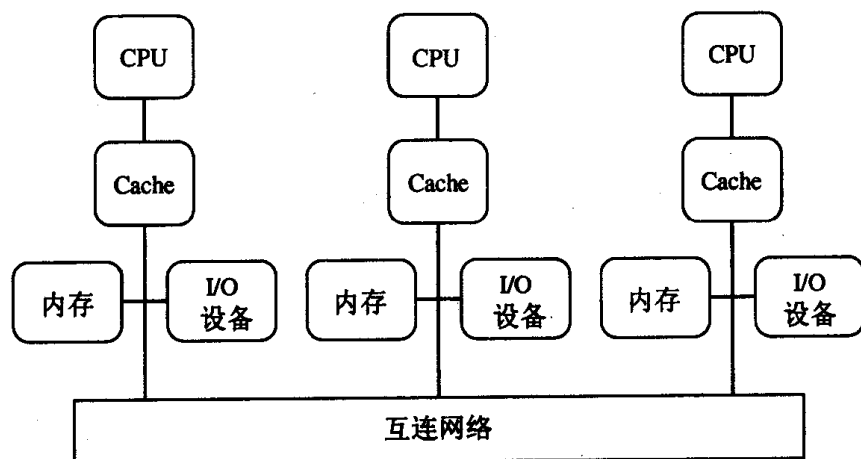


图 2.15 常见的分布内存多 CPU 计算机的体系结构：如果该计算机有一个单一的全局地址空间，则称为分布式多处理器；如果该计算机有分立的局部地址空间，则称为多计算机

如果分布的内存集合形成一个逻辑地址空间，则该并行计算机系统称为分布式多处理器。在分布式多处理器中，不同处理器上相同地址指的是同一内存位置。这种系统也称为非一致性内存访问多处理器（NUMA），这是因为根据被访问的地址是位于该处理器的局部存储中还是另一个处理器的局部存储中，内存的访问时间有相当大的变化。

**对 Cache 一致性的支持** 一些分布式多处理器，如 Cray T3D，并没有保证 Cache 一致性的硬件。在这样的计算机上，仅仅指令和私有数据能存放在处理器的 Cache 中。由于本地缓存访问与非本地内存访问之间存在的巨大时间差使得性能缺陷加剧。例如，在 Cray T3D 上一次非本地内存访问所花时间为 150 个时钟周期，但一次缓存的引用却只需两个时钟周期。由于这些原因，缓存一致性的硬件支持是十分有意义的。不幸的是，在集中式多处理器一节中所描述的监听方法随着处理器数目的增长，其可扩展性并不好，这是因为缓存控制器不能对共享存储总线进行简单的监听。需要一个更为复杂的协议来替换监听协议。

实现基于目录的协议是在分布式多处理器上实现缓存一致性的一种流行的方法。目录包含可能被缓存的每个内存块的共享信息。

对于每个缓存块，目录条目暗示了它是否：

- 未被缓存——当前未在任何处理器的 Cache 中；
- 共享——被一个或多个处理器缓存且内存中的备份是正确的；
- 独占——被某个写了该内存块的处理器所缓存，内存中的备份是陈旧的。

有必要追踪哪些处理器拥有某一缓存块的备份，在某个处理器将修改该内存块的值的时候可以将其他备份设为无效。如果处理器数目小于 128，使用位向量来存储处理器共享数

据块的信息是合理的。

为了避免对缓存目录的访问变成性能瓶颈,该目录本身也应该分布在该计算机的本地内存中。但是,目录的内容并未被复制:有关特定内存块的信息只在一个位置上出现。

**基于目录的协议实例** 现在让我们来看看基于目录的协议是如何工作的。考虑如图 2.16(a)所示的简单的分布内存多处理器。该并行计算机有 3 个 CPU,每个 CPU 有 Cache、内存和目录。所有处理器的内存形成了一个单一的地址空间且任一 CPU 都能够访问这地址空间中任意位置。整型变量  $X$  被存储在处理器 2 所控制的内存中,它当前的值为 7。处理器 2 有一个目录条目与包含  $X$  的缓存块相对应。该条目显示该内存块当前并未被缓存。

现在假设 CPU 0 试图读取  $X$  的值。而包含  $X$  的缓存块并未在 CPU 0 的 Cache 中。所以一个“读缺失”消息便从处理器 0 发送到处理器 2。包含  $X$  的缓存块的状态便被变为“共享”,位矢量也被修改以显示处理器 0 有缓存块的一个备份,最后该块被发送到处理器 0,如图 2.16(b)所示。

接下来, CPU 2 试图读取  $X$  的值,包含  $X$  的缓存块并未在 CPU 2 的缓存中。作为读缺失的一种结果,位矢量被修改以便表明处理器 2 也拥有该缓存块的一个备份,然后该缓存块被发送到处理器 2 的缓存中,如图 2.16(c)所示。

假设 CPU 0 现在将 6 写入  $X$ ,则一个“写缺失”消息由处理器 0 发送到处理器 2。目录控制器使当前位于 CPU 2 的缓存中包含  $X$  的那个缓存块的备份变为无效,修改位矢量以表明 CPU 2 不再拥有该缓存块的一个备份,最后将该缓存块的状态变为“独占”。图 2.16(d)显示该系统的最新状态。注意主存中  $X$  的值已过时。

如果 CPU 1 现在试图读取  $X$  (这将产生一次读缺失),则处理器 2 的目录控制器将发送一个“切换至共享”的消息到处理器 0。处理器 0 将该缓存块的一个备份传回处理器 2,在主存中保留一个更新后的备份。然后,这个最新的块被发送到处理器 1,图 2.16(e)所示。

假设对该缓存块进行的下一行动是 CPU 2 将值 5 写入  $X$ 。由于该块已不再存于其缓存中,所以它产生一个“写缺失”消息并传给目录控制器。目录控制器将无效消息发送给处理器 0 和处理器 1,它们都将其缓存中的该块清除。现在该块的状态被修改为被所有者 2 所“独占”。该块的一个备份被发送给 CPU 2 缓存,然后 CPU 2 对  $X$  的值进行修改,图 2.16(f)所示。

稍后, CPU 0 试图将值 4 写回  $X$ 。由于合适的块并不在它的缓存中,所以处理器 0 产生一个“写缺失”消息并发送到 CPU 2 的目录中。处理器 2 发送一个“取走 (take away)”消息给 CPU 2 的缓存控制器。然后,该缓存块被复制回内存,该块的状态保持为“独占”,但是位矢量被修改以表明拥有者现在为处理器 0。该块的一个备份被发送给 CPU 0 的缓存,从而  $X$  的值被修改,如图 2.16(g)所示。

最后,假定处理器 0 决定刷新包含  $X$  的那个缓存块。由于它对该块具有“独占”访问权,所以它必须将该块的内容复制回处理器 2 的内存中。最后的状态显示如图 2.16(h)所示。

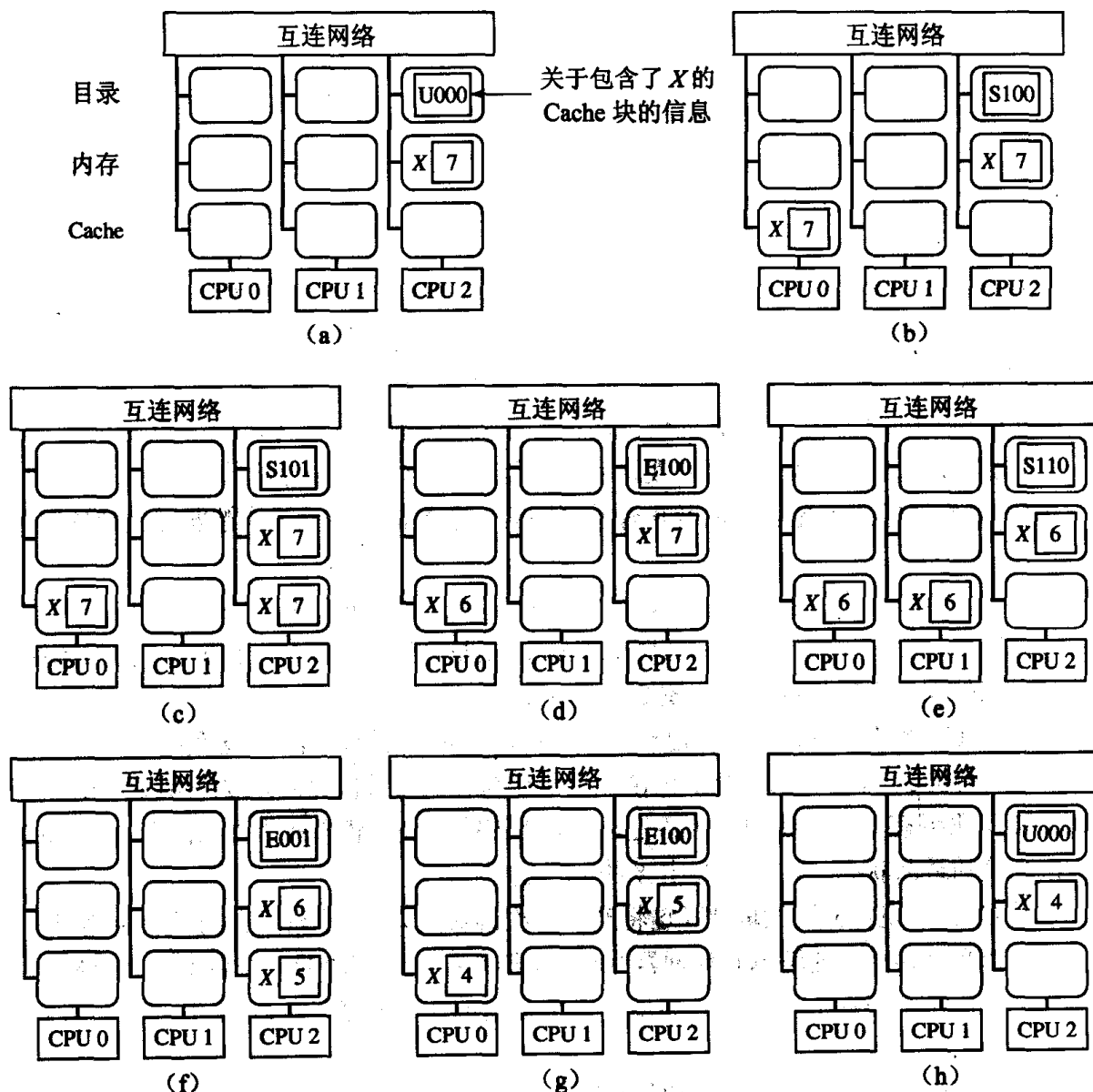


图 2.16 在分布式多处理器上实现 Cache 一致性的一种基于目录的协议 (a)  $X$  的值为 7, 包含  $X$  的块未被缓存; (b) CPU 0 读取  $X$  之后的状态; (c) CPU 2 读取  $X$  之后的状态; (d) CPU 0 将值 6 写回  $X$  之后的状态; (e) CPU 1 读取  $X$  之后的状态; (f) CPU 2 将值 5 写回  $X$  之后的状态; (g) CPU 0 将值 4 写回  $X$  之后的状态; (h) CPU 0 刷新包含  $X$  的那个缓存块之后的状态

## 2.5 多计算机

多计算机是另一种分布式内存的多 CPU 计算机 (如图 2.15 所示)。但是, 与具有单一全局地址空间的 NUMA 多处理器不同的是, 多计算机有独立的局部地址空间。每个处理器能直接访问的仅仅是属于它自己的局部存储空间。不同处理器上的相同地址指的是两个不同的物理内存空间。处理器之间没有共享的地址空间, 只能通过传递消息来进行交互, 因此不存在 Cache 一致性问题。

商业化的多计算机通常会提供定制的开关网络, 以提供低延迟、高带宽的处理器间通

道。商业化系统通常注意保持处理器速度与通信网络速度之间的平衡。与此相反的是，常用集群系统依赖于大批量生产的计算机、开关以及用于构建局域网的其他设备。这使得系统的成本较低，尽管其消息延迟更高且通信带宽更低。

### 2.5.1 非对称多计算机

早期的多计算机常常设计为非对称的，如图 2.17 所示。在某种程度上，它类似于阵列处理机。当后端处理器致力于“数字处理”的同时，前端计算机与用户和 I/O 设备进行交互。具有非对称设计的多计算机的两个范例是 Intel 的 iPSC 和 nCUBE/ten。其中，Intel 的 iPSC (c.1984) 由一个控制高达 128 个处理节点的 Cube Manager (前端节点) 构成。处理节点上运行着 NX 操作系统。NCUBE/ten (c.1985) 由一个运行 Intel 80286 主机 (前端) 组成，其上运行着 AXIS——一种商用的多程序操作系统。该主机控制高达 1024 个处理节点，这些处理节点运行一个非常小 (仅仅 4K 字节) 的 VERTEX 操作系统。

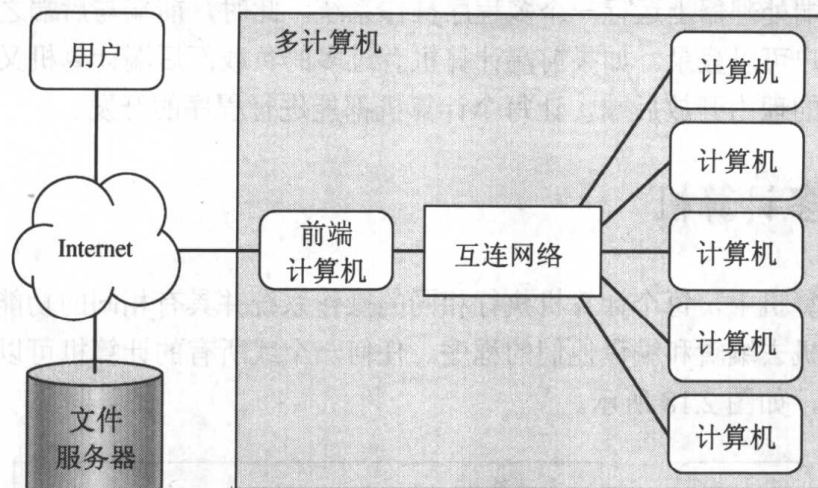


图 2.17 一个非对称多计算机：用户登录前端计算机，该前端计算机执行一个完全的多程序操作系统并提供程序开发所需的所有功能。后端计算机专门用于执行并程序。这些计算机可以执行一个简单的操作系统。在这种设计的一个变种里，前端可以由多台计算机构成

在非对称多计算机中，后端处理器专门用于执行并程序。这些处理器可以运行简单的操作系统 (如 VERTEX)，该操作系统不支持虚拟存储、I/O，甚至不支持多道程序。由于没有其他的处理器占用 CPU 周期或通过网络发送消息，所以易于理解、建模和调整并行应用的性能。这是一个显著的优点。另一个对于早期的商用多计算机也是非常重要的优点是，制造商能够比较容易地开发后端处理器所需的简单操作系统。

非对称多计算机也存在一些明显的缺点。首先是单点失效，即如果前端计算机出现了故障，那么整个并行计算机都将无法运转。

其次是前端计算机性能制约了并行机的可扩展性。用户登录前端计算机并用它来进行程序编辑与编译。前端发起在后端运行的并程序，同时还要负责所有的 I/O 操作。随着用户数目的增加，前端可能过载。在此期间，后端处理器的相当部分可能空闲。

多台前端计算机可以提供所需的额外计算资源，但它也带来了额外的复杂性。例如，用户如何知道登录哪个前端计算机？负载如何在前端计算机之间进行平衡？后端节点是静



态还是动态地被分配给特定的前端计算机?

前端性能瓶颈的另一个解决方案是改进单个前端计算机的性能, 比如使用一个集中式的多处理器来替代单 CPU 系统。当然, 一个未被充分利用的多处理器前端的存在可能使那些希望在并行计算中使用前端某些计算能力的用户感到灰心。

非对称计算机的第三个缺点与程序调试有关。后端处理器上运行的简单操作系统虽然可能使程序性能易于被理解, 但也使得调试程序变得更困难。由于不支持 I/O 操作, 所以对于节点程序来说, 直接打印一条消息给用户是不可能的。节点程序必须发送一条消息给前端计算机, 然后前端计算机通过打印消息内容的方式能够将该消息传送给用户。

然而, 这又导致了非对称多计算机的第四个缺点。每个并行应用需要开发两个截然不同的程序: 前端程序和后端程序。前端程序负责与用户与文件系统交互, 传送数据至后端处理器以及将后端处理器上的结果发送给外部世界。后端程序负责执行算法中的计算密集型部分。为每个应用开发两个程序既冗长乏味又易于出错。

调试并行程序的困难使得我们希望在后端节点上提供全功能的 I/O 设备。一个简单的实现方式是在后端处理器上运行一个多程序操作系统。此时, 前端与后端之间的差异是仅在于哪个节点用户可以登录。如果前端计算机有过多的负载而后端计算机又未被充分利用的话, 我们有充分的理由开放后端, 让每个计算机都能进行程序的开发。

## 2.5.2 对称多计算机

在对称多计算机中, 每个计算机执行相同的操作系统并具有相同的功能。用户可以登录任何一个计算机去编辑和编译他们的程序。任何一个或所有的计算机可以专门用于某一并行程序的执行, 如图 2.18 所示。

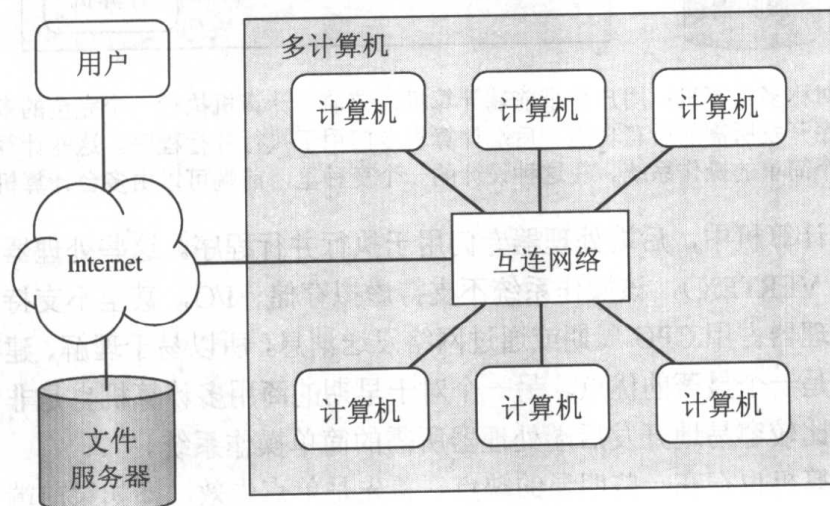


图 2.18 对称多计算机: 每个计算机都支持一个完全的、多程序的操作系统; 用户可以登录这些计算机中的任何一个; 任何一个或所有计算机可以用于执行并行程序

对称多计算机能够解决非对称多计算机所遇到的许多问题。例如, 它们能够减轻当把单一计算机用于程序开发所导致的性能瓶颈。如果某个计算机上的负载很重, 那么用户能够登录到其他计算机上。

对称多计算机对调试的支持较非对称多计算机要好。由于每个计算机都运行着一个功能完全的操作系统，所以每个处理器能够将调试消息写回给用户。

对称多计算机也消除了“前端/后端”编程问题。每个处理器执行相同的程序。当仅需要某个处理器执行某个特定操作的话，则可以很容易地使用 if 语句进行选择。

对称多计算机也存在以下一些缺点。

首先，某个用户能够登录该系统中任一节点（每个节点都拥有属于它自己的名字）的时候，更难于让人将该系统看做一台单一并行计算机。

其次，不存在一种简单的方法能够在所有处理器中平衡程序开发的负载。即便在登录前用户就检查这些计算机上的负载情况，但是这些负载是随时间而变化的。在没有软件支持的情况下，系统的负载很可能是不均衡的。

第三，当处理器必须与其他处理器竞争 CPU 周期、Cache 空间以及存储带宽时，要想实现并程序的高性能就更为困难了。Cache 是面向处理器而不是面向进程的。进程的上下文切换常常导致大量 Cache 缺失，使得性能下降。

### 2.5.3 怎样的模型对商用集群来说是最佳的

由于易于编程和调试，所以在每台计算机上执行一个功能齐全的多任务操作系统（如 Linux）并使每台计算机都能访问文件系统是有意义的。对称多计算机便具有该特征。

对于一给定应用，CPU 的性能在很大程度上依赖于其 Cache 命中率。如果该系统的主要目标是使单个并程序取得最大性能的话，在每个 CPU 上仅放一个用户进程是一个不错的主意。这是非对称安排的一个理由，其中绝大多数是不能进行程序开发的节点。

并程序的性能也受到网络速度的限制。因此，仅允许并行处理器访问处理器间互连网络是有意义的。用户应当通过另一通道访问前端计算机。

简而言之，商用集群的最佳排列可以是兼具非对称和对称设计属性的一个混合模型。图 2.19 举例说明了位于耶路撒冷的希伯来人大学的 ParPar 集群【24】。交换式以太网将前端计算机和 16 个后端计算机联系起来。与此同时，后端计算机还可独占地访问一条专用地高速数据网络。

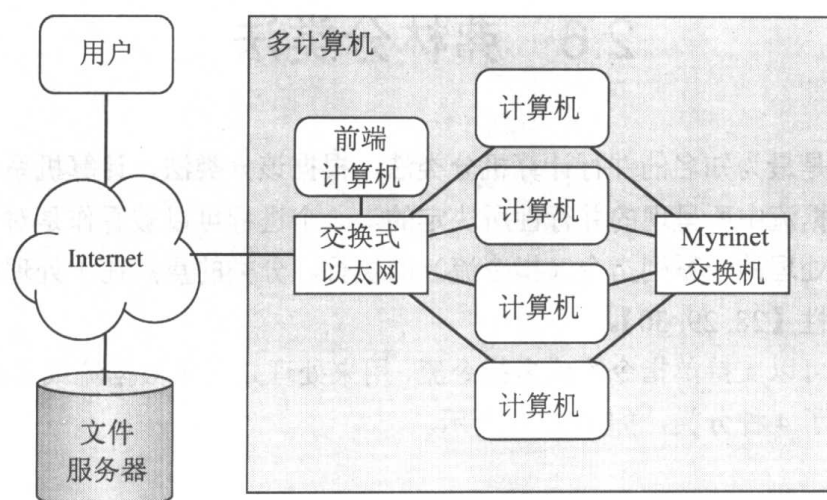


图 2.19 ParPar 集群混合了来自非对称和对称模型的特点



## 2.5.4 集群与工作站网络之间的差异

商用集群系统的组成部分——商用计算机和交换机在局域网中也能够找到。你既可以在集群上又可以在局域网中执行并行程序。那么，究竟什么系统应该被称为集群呢？

工作站网络是散布的计算机（通常位于用户桌面上）的集合。工作站常常由以太网（10Mbps）或快速以太网（100Mbps）相连。工作站的首要任务是为使用它的个人服务。执行并行任务仅是利用空闲 CPU 周期的方法。不同工作站可能有不同的操作系统和可执行程序。用户有权力关掉他们的工作站。因此，任务检查点和恢复系统是很重要的。

相反，商用集群是由批量生产的计算机系统和交换机所组成的专门用于执行并行任务的系统。这些计算机很可能只能通过网络来访问，也就是说这些计算机通常并没有显示器或键盘，其中某些系统可能不允许用户登录。所有的这些计算机都运行相同操作系统的相同版本并有相同的本地磁盘镜像。这个集群被作为一个实体来管理。

商用集群与工作站网络之间另一个关键的区别在于网络的速度。对于当今计算机的速度来说，以太网的速度太慢了，以至于不能被用作在商用集群下的网络。另外，必须使用开关网络而不是共享网络（即必须使用交换机而非集线器）。商用集群设计者可以选择如下三种流行的开关网络：快速以太网、千兆以太网及 Myrinet。表 2.2 对它们之间的差异进行了总结。

表 2.2 日常用集群（circa 2002）中三种开关网络的比较。每个节点的费用包括网络接口卡的费用和每个节点所分摊的开关费用

	延时	带宽	成本/节点
快速以太网	100 $\mu$ s	100Mbps	<100 美元
千兆以太网	100 $\mu$ s	1000Mbps	<1000 美元
Myrinet	7 $\mu$ s	1920Mbps	<2000 美元

## 2.6 弗林分类法

弗林分类法是最为知名的并行计算机分类法。根据该分类法，计算机系统的分类是由其在指令流和数据流中所呈现的并行性所决定的。一个进程可以被看作是对一系列操作数（数据流）进行处理的一系列指令（指令流）的执行。分类的重点在于处理指令流和数据流的硬件的多样性【28, 29, 30】。

计算机硬件可以支持单指令流或多指令流，用来处理单数据流或多数据流。因此，弗林分类法给出了 4 种分类，如图 2.20 所示。

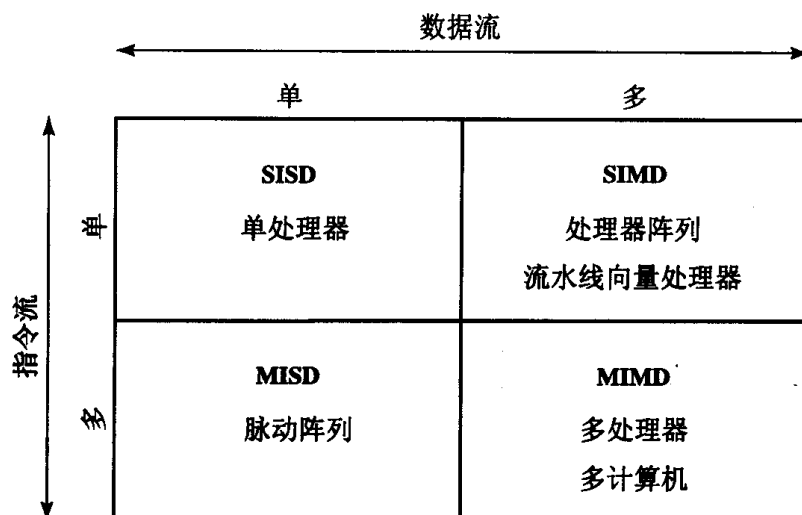


图 2.20 计算机体系结构的弗林分类法

### 2.6.1 SISD

SISD 计算机指的是具有单指令流单数据流的系统。单处理器属于该类。尽管它只有单个 CPU 执行单指令流，但现代的单处理器仍可能体现一些并发执行的特性。例如，超标量体系结构支持多个可被同时执行的无关操作的动态识别与选择。指令预取和指令的流水线执行也是体现现代 SISD 计算机中的并发执行特性的例子，虽然根据弗林分类法这些例子是处理的并发而不是执行的并发【30】。

### 2.6.2 SIMD

SIMD 系统是指只具有单个指令流但具有多个数据流的系统。处理器阵列和流水线向量计算机属于该类。正如我们所看到的那样，处理器阵列具有单个控制单元执行单个指令流，同时，多个附属处理器能够同时对不同数据元素执行相同的操作。流水线向量处理器依赖于非常快的时钟和一个或多个流水线功能部件来对数据集中的多个元素执行相同的操作。

### 2.6.3 MISD

MISD 计算机指的是具有多个指令流但只有一个数据流的系统。MISD 计算机是“多个独立的执行功能部件对单数据流执行操作并将结果从一个功能部件推进到下一个功能部件的一条流水线”【30】。

脉动阵列是 MISD 计算机的一个例子。来自于单词“systole”的该名字指的是心脏的收缩。脉动阵列是由能“抽取数据”的简单处理元件所组成的一个网络。

例如，考虑如图 2.21 所示的简单排序元件。排序工作分两步进行。在第一阶段中（如图 2.21 (a) 所示）它输入三个数据值。在第二阶段中（如图 2.21 (b) 所示），它输出最小

值、中间值和最大值。

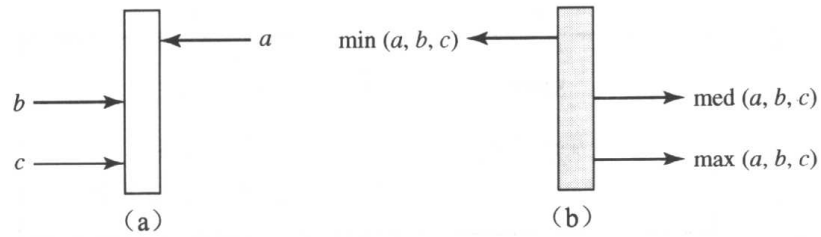


图 2.21 脉动分类部件 (a) 在第一个周期内, 部件输入控制字  $a$ 、 $b$  和  $c$ ; (b) 在第二个周期内, 部件沿着已设计好的通道输出这三个控制字的最小值、中值和最大值

通过将这些分类元件连接成一个线性数组, 我们能够创建硬件优先队列【70】, 如图 2.22 所示。该优先队列支持两个操作: 插入键和输出具有最小值的键。每个操作都要需要两个周期 (即常数)。

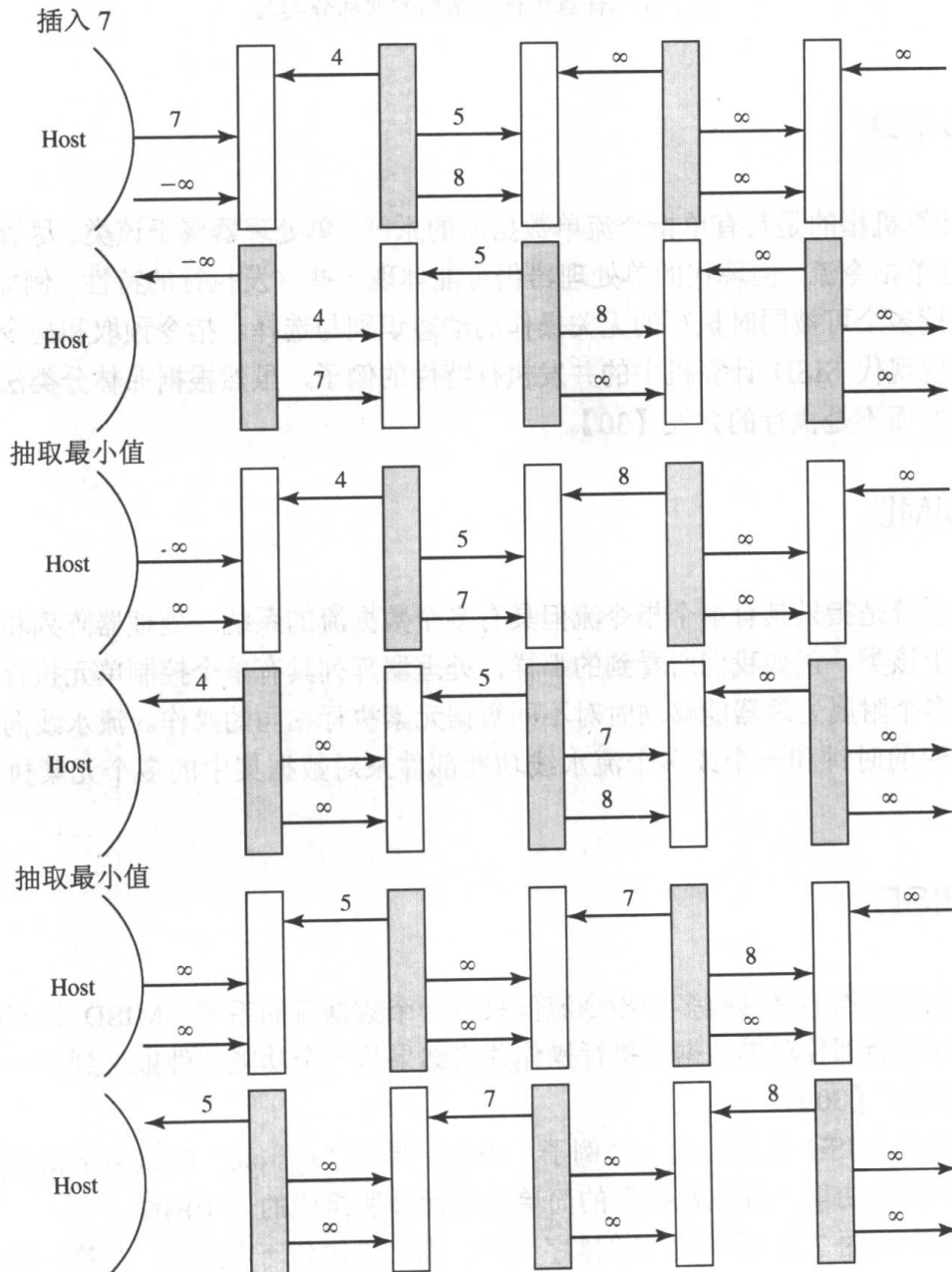


图 2.22 用脉动阵列实现的硬件优先队列的操作

为了插入键  $x$ , 在第一个周期内, 主机将  $x$  和  $-\infty$  插入优先队列的左端。在第二个周期内, 该优先队列在其左端输出  $-\infty$ 。主机丢弃该值。

为了抽取出最小的键值, 在第一个周期中, 主机插入两个  $\infty$  并在第二个周期中抽取出最小的键值。对于所有的操作, 在第一个时钟周期中,  $\infty$  被插入搏动阵列的右端。在第二个时钟周期中, 搏动阵列的右端应该输出两个  $\infty$ , 如果有一个  $\infty$ , 则优先队列已经溢出。

在这个例子中, 脉动阵列中的所有部件都是相同的, 但搏动阵列能够包含执行不同功能的各种部件, 这就是为什么它被恰当地称为“多指令”体系结构。

虽然人们已经开发出了几个基于 MISD 原理的商用并行系统, 但是它们都把解决诸如数字信号处理这样的特定应用作为目标。Flynn 和 Rudd 认为从熟悉的编程结构到 MISD 组织缺乏自然的映射, 因此抑制了人们对该体系结构的兴趣【30】。

## 2.6.4 MIMD

MIMD 计算机是具有多指令流和多数据流的系统。多处理器和多计算机属于此类。这两种体系结构都建立在多 CPU 基础之上。不同的 CPU 能够同时执行对不同数据流进行处理的不同指令流。

大多数当代的并行计算机属于 MIMD 计算机。因此, MIMD 设计对描述现代并行体系结构并不是特别有帮助。在本书的其余部分, 我们将使用本章引入的更为特殊的术语来描述并行体系结构。

## 2.7 本章小结

20 世纪 60 年代中期以来, 科学家和工程师已经设计和建造了各种各样的并行计算机。20 世纪 80 年代初, 首次出现了含有 8 个或更多处理器的商用并行计算机。通过对这段历史的回顾, 我们能够知道在通用计算平台方面, 为什么一些体系结构比其他体系结构更为成功。

控制单元相对较高的价格和数据并行在科学计算中的优势刺激了处理器阵列的构建。在超大规模集成电路制造工艺能够在一个芯片上集成 CPU 后, 处理器阵列的第一个优势, 即单一的控制单元变得不重要了。由于处理器阵列无法有效执行条件执行代码中的数据并行操作, 处理器阵列第二个优势也被削弱了。

同时, 处理器阵列的许多缺点随着时间的推移而变得越来越清晰。许多问题很难使用严格的数据并行解法。处理器阵列缺乏对多用户的良好支持, 而且不适合小规模系统, 这使得它们难以在市场的中低端与其他系统竞争。处理器阵列最明显的缺点是它们由定制的超大规模集成电路建造, 并且不能利用批量生产的 CPU 在性能和价格方面的改进。

由于上述这些原因, 在多 CPU 系统日益引人注目的时候, 处理器阵列却已引退。大量的商用并行计算机仅包含不多于几打的 CPU。这些相对小规模的系统足以满足绝大多数高性能计算的需求, 这是因为批量生产的 CPU 现在已经具有相当强的计算能力。

大多数小规模并行计算机采用集中式多处理器体系结构。多个处理器通常通过一条共

享总线访问公共的本地存储器。这些系统的设计师必须解决 Cache 一致性问题 and 同步问题。系统通常采用监听 Cache 和写无效协议, 在写操作发生的时候将陈旧的缓存块变为无效以确保 Cache 一致性。软件同步机制依赖于一条或多条硬件指令, 它们的功能是完成对内存位置的读和更新的原子操作。

对于包含 100 个或更多 CPU 的并行计算机来说, 为了给 CPU 提供充足的内存带宽, 需要采用某种形式的分布存储。在分布存储系统中, 每个 CPU 都有一个邻近的本地内存。当大多数内存引用访问的是 Cache 或最近的本地内存时, 聚合存储带宽可以达到很高, 并随着处理器的数目扩展。

分布存储并行机被划分为两类, 取决于它们支持单一的地址空间还是拥有多个独立的地址空间。在分布存储的多处理器中, 两个不同 CPU 的同一地址指的是并行机内位于某处的相同内存位置。在分布存储的多处理器中, 要实现 Cache 一致性会更加困难, 因为没有可供缓存控制器监听的共享总线。因此, 在这类系统中, 实现缓存一致性最常见的方法是采用基于目录的方案, 通过使用位向量来记录哪个 CPU 已经拥有哪个 Cache 块备份。

多计算机是具有多个独立地址空间的分布存储并行计算机。位于两个不同 CPU 上的相同地址指向不同的内存位置。由于没有共享内存, 所以不需要担心缓存一致性问题。对于处理器来说, 为了共享数据, 它们必须互相发送消息。数据项是否反映了最新的修改完全依赖于程序员。商业多计算机一般都具有定制的、高性能的处理器间通信网络和消息传递软件来确保处理器间低延迟、高带宽的通信。

常用集群是一种特殊的多计算机, 它由大批量生产的计算机和网络设备构成。常用集群能够结合最新的商用技术并利用这些产品具有吸引力的价格/性能比。由于常用集群通常拥有比商业并行计算机更快的 CPU 和更慢的网络, 所以在 CPU 与网络之间并不平衡, 因此适用的范围较窄。

Flynn 分类法是对并行计算机进行分类最为知名的方法之一。不幸的是绝大多数并行计算机属于同一类别 (MIMD), 从而限制了该分类法的用途。

## 2.8 主要术语

barrier synchronization

binary n-cube

cache coherence problem and protocols  
(directory-based, write invalidate) cache

commodity cluster

2-D mesh network

distributed multiprocessor

Flynn's taxonomy (SISD, SIMD, MISD, MIMD)

Interconnection media (shared, switched)

阻塞同步

二元 n-立方体

一致性问题 and 协议 (基于目录的 Cache 一致性协议, 写无效 Cache 一致性协议)

常用集群

二维网格形网络

分布式多处理器

Flynn 分类法 (单指令流单数据流、单指令流多数据流、多指令流单数据流、多指令流多数据流)

互连介质 (共享介质、开关介质)

interconnection networks (2-D mesh, binary tree, butterfly, hypercube, hypertree, shuffle-exchange)	互连网络(二维网格形网络、二叉树网络、蝶形网络、超立方体网络、超树网络、混洗-交换网络)
multicomputer (asymmetrical, symmetrical)	多计算机(非对称多计算机、对称多计算机)
multiprocessor (UMA, NUMA, SMP)	多处理器(一致性内存访问的多处理器、非一致性内存访问的多处理器、对称多处理器)
mutual exclusion	互斥
network attributes (bisection width, diameter, edge/node, edge length)	网络属性(对分带宽、直径、边/节点、边长)
network topology (direct, indirect)	网络拓扑(直接网络拓扑、间接网络拓扑)
perfect shuffle	完美混洗
performance	性能
private data	私有数据
ranks	阶
shared data	共享数据
systolic array	脉动阵列
vector computer (pipelined vector processor, processor array)	向量机(流水线向量处理器、处理器阵列)

## 2.9 参考文献

Patterson 和 Hennessy 所著 “*Computer Architecture: A Quantitative Approach*” 一书的第 8 章对多处理器体系结构进行了非常好的一般性介绍【90】。Nagendra 和 Rzymianowicz 对用于构建常用集群的高速网络进行了调查分析【88】。Feitelson 等人对 ParPar 的总的看法是: “仅仅使用常用构建的一种通用目的、适于多用户的 MPP 式的系统”【24】。

如果你想组装属于你自己的常用集群的话, 由 Sterling 等人所著的 “*Beowulf Cluster Computing with Linux*” 一书能够对运行 Linux 操作系统的常用 PC 集群的构造、管理和编程提供了实际的指导【105】。

## 2.10 练习题

2.1 画出具有两个、四个和八个节点的超立方体网络并确保你对其中的节点都进行了标注。请问具有  $n$  个节点的超立方体网络是具有  $2n$  个节点的超立方体网络的一个子图吗?

2.2 存在多少不同的方法来标注一个  $d$  维的超立方体?

2.3 令图中节点  $u$  和  $v$  之间的距离为从  $u$  到  $v$  最短路径的长度。已知一个  $d$  维超立方体和一指定的源节点  $s$ , 请问有多少个节点与  $s$  的距离为  $i$ , 其中  $0 \leq i \leq d$ 。

2.4 证明如果在一个超立方体中节点  $u$  与节点  $v$  的距离为  $i$ , 则存在  $i!$  条从  $u$  到  $v$ 、长度为  $i$  的路径 (虽然一些超立方体边可能出现在多条路径中)。

2.5 证明如果在一个超立方体中节点  $u$  与节点  $v$  的距离为  $i$ , 则存在  $i$  条从  $u$  到  $v$ 、长度为  $i$  且无共享边的路径。

2.6 证明超立方体没有奇数长度的环。

2.7 给出将消息从某个超立方体 (节点数为  $n$ ) 中节点  $u$  传送到节点  $v$ , 步数不超过  $\log n$  的算法。

2.8 画出具有 2 个、4 个和 8 个节点的均匀混洗网络并确保你对其中的节点都进行了标注。请问具有  $n$  个节点的超立方体网络是具有  $2n$  个节点的超立方体网络的一个子图吗?

2.9 已知一个具有  $2k$  个节点的均匀混洗网络, 请问在何种情况下节点  $u$  和节点  $v$  恰好存在  $2k-1$  个链接?

2.10 给出将消息从某个均匀混洗网络 (节点数为  $n$ ) 的节点  $u$  传送到节点  $v$ , 步数不超过  $2\log n-1$  的一个算法。

2.11 Omega 网络是基于完美混洗互连模式的一种间接拓扑【66】。图 2.23 说明了一个 8 处理器的 Omega 网络。考虑连接  $n=2^k$  个处理器的一个 Omega 网络。

- 该网络有多少开关部件?
- 该网络的直径为多少?
- 该网络的对分带宽多大?
- 每个开关节点的最大边数为多少?
- 随着节点数目的增加, 该网络的边长是否是常数?

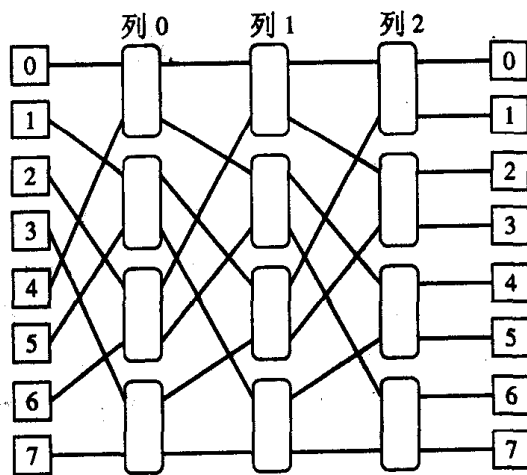


图 2.23 连接 8 处理器 (图中由正方形表示) 的 Omega 网络

2.12 假设  $n=2^k$  个处理器由 Omega 网络相连, 如图 2.23 所示。请设计一个算法将消息从处理器  $i$  传送给处理器  $j$  (提示: 用二进制数表示目标地址  $j$ )。

2.13 为什么处理器阵列非常适合数据并程序序的执行?

2.14 已知一个含 8 处理器部件的处理器阵列, 其中每个处理器每秒钟能够执行一千万次整数操作。请确定该处理器阵列在对两个整数矢量 (矢量长度为 1~50) 执行加法操作时的性能 (以百万次操作/秒为单位)。

2.15 估算处理器阵列执行具有  $k$  种情况的一条 case 语句时的效率。假设该 case 所包

含的所有指令都是并行指令而且所有指令的执行时间相同。

(a) 如果每条 case 语句所包含的指令数完全相同的话, 效率是多少?

(b) 如果第  $i$  条 case 语句有  $I_i$  条指令且一个处理部件在第  $i$  条 case 语句中处于活跃状态的概率为  $P_i$  的话, 请问效率是多少?

2.16 为什么在多处理器中需要大的数据与指令缓存?

2.17 为什么集中式多处理器中处理器的个数被限定在几打的范围之内?

2.18 基于目录的协议是分布存储多处理器上实现缓存一致性的一种很流行的方法。

(a) 为什么目录需要分布在各个多处理器的本地内存中?

(b) 为什么目录的内容不是重复的?

2.19 继续图 2.16 中对基于目录的 Cache 一致性协议的描述, 假设现在按下列次序出现五个操作: CPU 2 读取  $X$ , CPU 2 将数值 5 写入  $X$ , CPU 1 读取  $X$ , CPU 0 读取  $X$ , CPU 1 将数值 9 写入  $X$ 。请说明在每个操作后目录、Cache 和内存的状态。

2.20 请对 Flynn 分类中的每个类别进行一些研究工作并为每个类别找出至少一种商用计算机 (可以是现在已不存在的计算机, 但不能是本书中已经提到过的)。

2.21 继续图 2.22 中有关脉动优先队列操作的示例, 演示其处理如下 5 个请求时的状态变换: 插入 4、抽取最小值、插入 11、插入 9、抽取最小值。

2.22 请解释为何当代的超级计算机都是多计算机。



# 第 3 章 并行算法设计

From the highest to the humblest tasks,  
all are of equal honor; all have their part to play.

Winston Churchill

## 3.1 概 述

是开始进行并行算法设计的时候了！我们的方法基于 Ian Foster 所描述的任务/通道模型【31】。该模型促进了高效并行算法（特别是那些运行于分布存储并行计算机上的并行算法）的开发。

本章的前两节描述任务/通道模型和基于该模型设计并行算法的基本步骤。然后，我们将研究几个简单的问题。对于其中的每个问题，我们都将设计一个任务/通道并行算法并推导出该算法预期执行时间的表达式。在此过程中，我们的执行时间模型将逐步完善。

## 3.2 任务/通道模型

任务/通道模型将并行计算表示为一系列任务，任务之间通过使用通道发送消息进行相互交互，如图 3.1 所示。这里的任务指的是一个程序、其本地存储以及一组 I/O 端口。本地存储包含了程序的指令及其私有数据。任务能够通过输出端口将其本地数据值发送给其他任务。反过来，任务也能够通过输入端口接收来自其他任务的数据值。

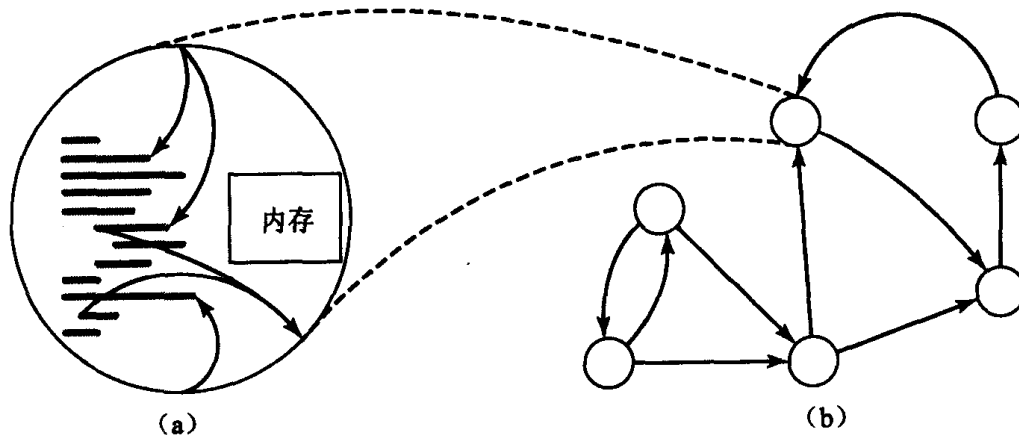


图 3.1 任务/通道编程模型

- (a) 任务由程序、本地存储以及一组 I/O 端口组成；
- (b) 并行计算可被视为一个有向图，其中顶点代表任务，有向边代表通信通道

通道是连接一个任务的输出端口与另一任务输入端口的一个消息队列。数据值按其通道另一端的输出端口中所放置的次序出现在输入端口里。

显然,只有在通道一端发送某个数据之后才能在通道的另一端接收该数据。如果一个任务试图在输入端接收某个值而没有任何可获得的值的话,则该任务必须等待直到该值出现为止。此时,我们称接收值的任务已被阻塞。反之,发送消息的进程从不阻塞,即使在这之前沿着同一通道发送的消息仍未被接收。换句话说,任务/通道模型中发送是异步操作,接收却是同步操作。

任务/通道模型中,对私有数据的本地访问很容易与发生在通道上的非本地访问区分开来。这是该模型的优点,因为我们应该考虑到本地数据访问比非本地数据访问要快得多。

我们所讨论的并行算法的执行时间是指任何其中一个并行任务处于活跃状态的时间长度。其中,启动时间指所有任务同时开始执行的时间,完成时间指的是最后一个任务停止的时间。

### 3.3 Foster 的设计方法论

Ian Foster 提出了一个分为四步的并行算法设计过程【31】。该过程将与机器相关的考虑延后,以促进可扩展并行算法的开发。我们将在本章和本书其余部分使用 Foster 设计方法,为各类应用程序开发并行算法。

Foster 设计方法中的四步分别被称为划分、通信、聚集和映射,如图 3.2 所示。本节中,我们将对每一个步骤进行解释,同时提供了一个检查表,以帮助你确定设计的质量。本节的解释比较理论化,但本章的其他部分将通过几个实例来说明该理论。

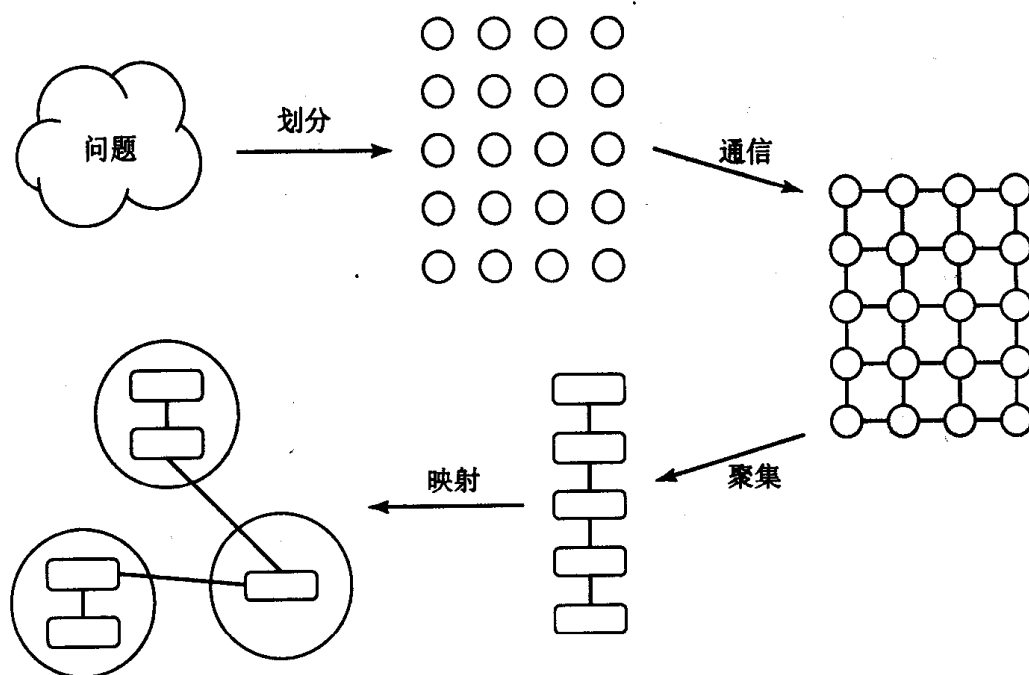


图 3.2 Foster 的并行算法设计方法

### 3.3.1 划分

在我们开始一个并行算法设计的时候, 我们常常试图发现尽可能多的并行性。划分是将计算和数据进行分片的过程。好的划分可以将计算和数据都分解成许多小片。为了实现这个目的, 我们要么采用以数据为中心的方法, 要么采用以计算为中心的方法。

域分解 (domain decomposition) 是一种并行算法设计方法, 其中, 我们首先将数据分解成片, 然后确定如何将计算与数据联系起来。通常我们主要考虑在程序中最大和最频繁访问的数据。

考虑图 3.3 所示的例子。其中, 三维矩阵是最大和最频繁访问的数据结构。我们可以将该矩阵划分为二维切片的集合, 从而得到一维原始任务 (primitive task) 的集合。我们也可以将给矩阵划分为一维切片的集合, 任务则相应地变为二维原始任务的集合。我们还可以将每个矩阵元素分别来考虑, 从而生成原始任务的一个三维集合。在这个步骤时, 通常我们要最大化原始任务的数目。因此, 三维划分方案较好。

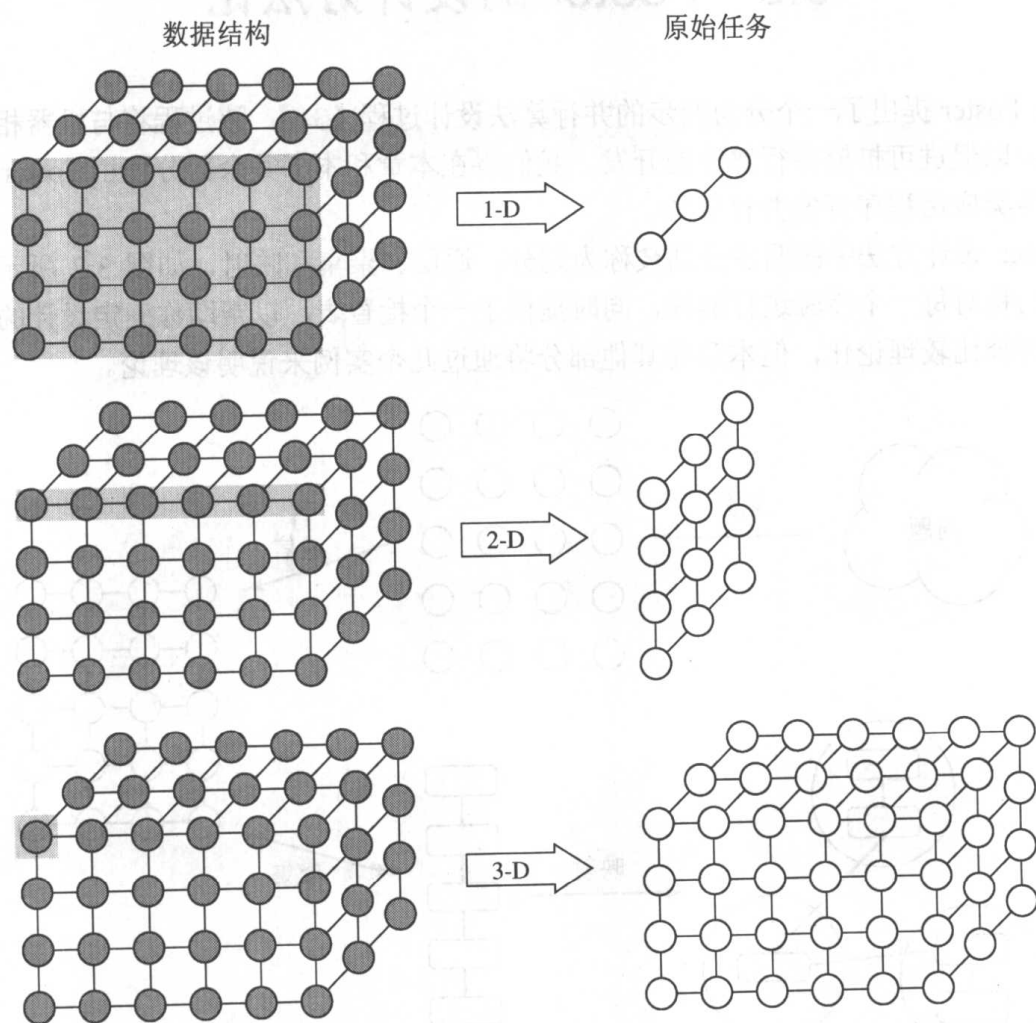


图 3.3 三维矩阵的三种域分解, 导致显著不同的原始任务的集合

功能分解 (functional decomposition) 是域分解的补充策略。其中, 首先我们将计算分解成片, 然后确定如何将数据项与计算联系起来。功能分解常常生成通过流水实现并发的

任务的集合。

例如，考虑支持交互式图像指导脑手术的一个高性能系统，如图 3.4 所示【37】。在手术开始之前，系统输入病人脑部的一系列 CT 扫描图像，构造出三维模型。在手术过程中，系统追踪手术器械的位置，将它们由物理坐标转换为图像坐标并在监视器上显示位于脑组织中的器械位置。该系统具有内在的并发性。在一个任务将图像由物理坐标转换为图像坐标的同时，第二个任务能够显示前一张图像，第三个任务能够为下一图像追踪手术器械的位置。

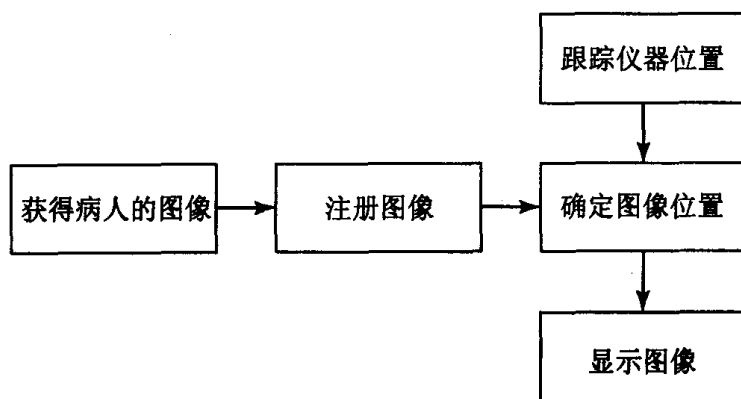


图 3.4 支持交互式图像指导脑手术的系统功能分解

无论我们选择哪种分解方法，都能够称这些片中的每一片为原始任务。我们的目的是尽可能多地识别原始任务，因为原始任务的数目是我们能够开发的并行性的上界。

我们能够使用下列检查表来评估一个划分的质量。最好的设计应该满足下列所有这些属性（Foster【31】）。

- 原始任务数至少要比目标并行计算机上的处理器数高一个数量级（如果该条件不能满足的话，则后面的设计将受到很大限制）。
- 冗余计算和冗余数据结构存储最小化（如果该条件不能满足的话，则该设计在问题规模增加的时候可能效果不佳）。
- 原始任务的大小大概相同（如果该条件不能满足的话，则难以平衡处理器之间的工作）。
- 任务数是问题规模的一个增量函数（如果该条件不能满足的话，则也许不能使用更多的处理器去求解更大规模的问题）。

### 3.3.2 通信

在识别出原始任务之后，下一步就是确定它们之间的通信模式。并行算法有两种通信模式：局部通信和全局通信。当一个任务为执行某个计算而需要来自少数其他任务的值的时候，我们创建从供应数据的任务到消费数据的任务的一条通道。这是一个局部通信的例子。

相反，全局通信存在于为执行计算而需要大量原始任务贡献数据的时候。全局通信的一个实例是计算原进程所拥有值的和。虽然，注意什么时候需要全局通信是重要的，但一

般来说在算法设计的这个阶段为它们规划通信通道并不会有什么帮助。

我们将任务之间的通信作为并行算法开销的一部分,因为串行算法并不需要进行这些通信。最小化并行开销是并行算法设计的一个重要目标。请牢记这一条,我们可以使用 Foster 的检查表来帮助我们评估并行算法的通信结构。

- 平衡任务间的通信操作。
- 每个任务仅仅与少量的邻居进行通信。
- 任务能够并发地执行它们的通信。
- 任务能够并发地执行它们的计算。

### 3.3.3 聚集

在并行算法设计的前两步中,我们的注意力集中在尽可能多地识别并行性。此时,我们的设计很可能无法在真实的并行计算机上高效执行。例如,如果任务数超过了处理器数目几个数量级,那么仅仅创建这些任务就将成为很大的开销。在设计最后两步中,我们需要考虑目标体系结构(如集中式多处理器或多计算机)。我们考虑怎样将原始任务合并成大的任务并映射到物理处理器上以减少并行开销的量。

聚集是为改善性能或简化编程而将任务合并为大的任务的过程。有些时候,我们希望任务的数目比执行并行算法的处理器数目要大。但大多数情况下,当开发 MPI 程序的时候,聚集的结果是给每个处理器分配一个任务。此时,任务到处理器的映射是非常简单的事情。

聚集的目标之一是降低通信开销。如果我们将相互通信的原始任务聚集起来的话,则它们之间的通信将被完全消除。这是因为被原始任务控制的数据值现在存在于合并后的任务的内存中,如图 3.5 (a) 所示。我们称它增加了并行算法的局部性。如果后面的任务由于等待前面的任务提供数据而无法并发地执行,那么聚集这些任务常常是不错的主意。

减少通信的另一种方法是合并发送消息的任务组与合并接收消息的任务组,从而减少发送消息的数目,如图 3.5 (b) 所示。发送少量长消息比发送总长度相同的大量短消息花费的时间更少,这是因为每次消息发送时都有一个消息启动开销,而该时间与消息的长度无关。

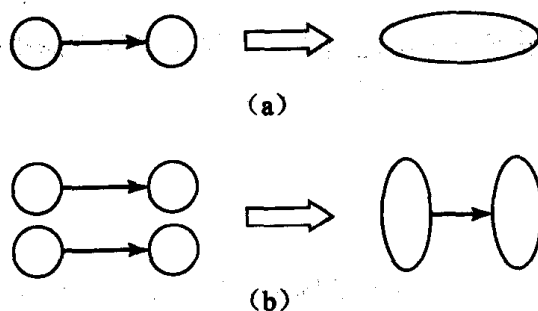


图 3.5 聚集任务能够消除通信或至少减少它们的开销。

(a) 将由通道相连的任务合并,可以增强并行算法的局部性并消除通信;

(b) 合并发送和合并接收任务可以减少消息传送的数目

聚集的第二个目标是要维持并行设计的可扩展性。我们希望确保对任务的合并不会影

响将我们的程序移植到有着更多处理器的计算机上。例如,假设我们正在开发一个对规模为  $8 \times 128 \times 256$  的三维矩阵进行处理的并程序。我们计划在具有 4 个 CPU 的集中式多处理器上执行该并行算法。假如我们将第二维和第三维聚集起来的话,该程序肯定能够在 4 个 CPU 上执行。每个任务将对一个  $2 \times 128 \times 256$  的子矩阵负责。在不改变设计的情况下,我们甚至可以在具有 8 个 CPU 的系统上执行该程序,此时每个任务将对一个  $1 \times 128 \times 256$  的子矩阵负责。然而,在不改变设计的情况下,我们却不能将该程序移植到超过 8 个 CPU 的并行计算机上,这可能导致对并行代码进行大量的改动。因此,对矩阵的第二维和第三维进行聚集的决定是目光短浅的做法。

聚集的第三个目标为减少软件工程上的开销。如果我们正在对一个串行程序进行并行化,聚集允许我们更多地利用现成的串行代码,以减少开发并程序的时间和成本。

我们可以使用 Foster 的检查表来评估聚集的质量。

- 聚集增加了并行算法的局部性。
- 复制的计算比它们所替代的通信花费的时间要少。
- 复制的数据总量足够小,使得算法具有可扩展性。
- 聚集后的任务有相似的计算和通信开销。
- 任务数是问题规模的一个增函数。
- 任务数要尽可能的少,但至少要与目标计算机中的处理器数目一样多。
- 合理权衡聚集所带来的好处与修改现有串行代码的开销。

### 3.3.4 映射

映射是将任务分配给处理器的过程。对于集中式多处理器系统来说,操作系统将自动地将进程映射到处理器上。因此假定目标系统为分布存储的并行计算机。

映射的目标是最大化处理器的利用率和最小化处理器之间的通信。处理器利用率是处理器用于执行用于求解问题的时间的平均百分比。当计算负载完全平衡时(即所有处理器在相同时刻开始和终止执行),处理器利用率取得最大值。反之,当某些处理器仍忙于处理的时候如果有些处理器处于空闲状态,处理器利用率将下降。

如果通道相连的两个任务被映射到不同的处理器,处理器间的通信将增加。而当通道相连的两个任务被映射到同一个处理器的时候,处理器之间的通信将减少。

例如,考虑图 3.6 所示的映射。图中 8 个任务被映射到 3 个处理器上。左边和右边的处理器分别对两个任务负责,而中间的处理器对 4 个任务负责。如果所有的处理器都有相同的速度且执行每个任务需要相同的时间的话,则中间的处理器将花费两倍于其他两个处理器的时间来执行任务。如果每个通道通信的数据总量相同的话,则中间的处理器也将对两倍于其他两个处理器的处理器间通信量负责。

增加处理器利用率和最小化处理器间通信常常是一对相互矛盾的目标。

例如,假设有  $p$  个处理器可用。将每个任务都映射到同一处理器可将处理器间的通信减少至零,但处理器的利用率也被减少到了  $1/p$ 。我们的目标是选择一个映射,该映射是在最大化利用率和最小化通信二者之间的一个合理的折衷点。

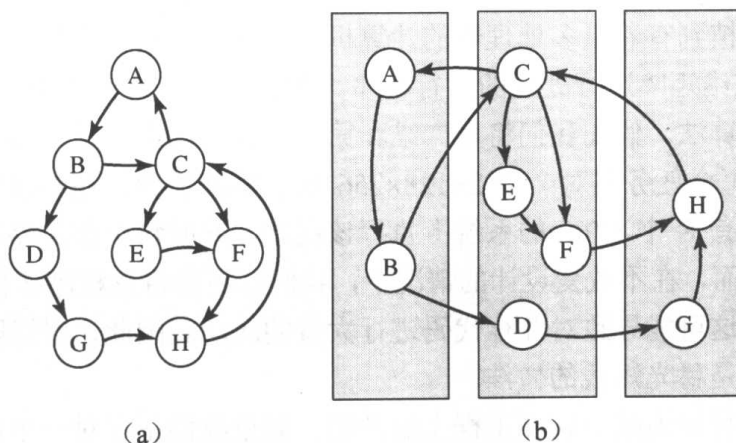


图 3.6 映射过程。(a) 任务/通道图；(b) 将任务映射到三个处理器上去。  
一些通道代表处理器内通信，而其他通道代表处理器间通信

不幸的是，寻找该映射问题的一个优化解是 NP 难题【38】，即不存在一个多项式时间算法能够将任务映射到处理器上使执行时间最小。因此，我们必须依赖于启发式算法，以得到较好的任务映射。

当使用域分解方法来划分问题的时候，在聚集后剩余的任务常常有相似的大小，即任务之间的计算负载相互均衡。如果任务之间的通信模式是规则的话，那么一个较好的策略是生成  $p$  个聚集后的任务，使得通信最小化且能够将这些任务中的每一个都映射到它自己的处理器上。

有时候，任务数是固定的且它们之间的通信模式也是规则的，但执行每个任务所需的时间却有着显著的不同。如果临近的任务有相似的计算需求的话，则在增加通信开销的前提下，将任务循环（或交叉）映射到处理器上能够得到均衡的计算负载。

一些问题在任务间产生无结构的通信模式。在这种情况下，映射的主要目标是使并行程序通信开销最小化。在程序开始运行前进行的静态负载均衡算法能够确定映射的策略。

到现在为止，我们一直都在讨论具有固定任务数的映射方法。当任务在运行时被创建和销毁或任务的通信或计算需求变化很大的时候，需要动态的负载均衡算法。在并行程序的执行过程中，动态的负载均衡算法会被不时调用。它分析当前的任务执行情况，并生成任务到处理器的新的映射。

最后，一些并行设计需要创建短期任务去完成特定的功能，任务之间并不相互通信。作为替代的是，每个任务被分配一个子问题去求解并返回子问题的解。任务调度算法可以是集中式也可以是分布的。

在集中式任务调度算法中，处理器池被分成一个管理者处理器和许多工人处理器。其中，管理者处理器维护了一个任务分配的列表。当某个工作处理器没有任何事的时候，它从管理者处申请一个任务。管理者则回复给它一个任务。工人完成任务后返回解并申请另一个任务。

管理者/工人类型任务调度的一个潜在问题是管理者可能成为瓶颈。在某种程度上，通过在某时刻分配多个任务或允许工作处理器在它们处理前面任务的同时预取任务的办法可以使该问题得到改善。

在分布式任务调度算法中，每个处理器都维护着一份其自己可用任务的列表。需要某种机制在处理器之间散布这些可用的任务。一些算法依赖于“推”策略。有太多可用任务的处理器将其上的一些任务发送到临近的处理器上。另外一些算法依赖于“拉”策略。没有任务的处理器向临近的处理器请求任务。分布式任务调度的挑战在于确定终止条件。未完成的任务被散布在处理器之间。对任何处理器来说，要知道什么时候所有处理器都完成了任务都是很困难的事。相反地，在管理者/工人类型算法中，管理者处理器总是确切地知道还有多少未完成的任务。

其他的任务调度算法综合了集中式和分布式算法。例如，两级管理者/工人策略有两级管理者。高级的管理者管理着一组低级管理者。每一个低级管理者将任务分配给它自己的工人组。管理者周期性地相互通信以平衡每个低级管理者所拥有的未分配任务的数目。

图 3.7 对并行算法的不同特点怎样导致不同的映射策略进行了总结。因为映射策略依赖于在并行算法设计过程的前几个步骤所作的决定，因此在设计过程中保持开放的思维是非常重要的。下列检查表可以帮助你确定是否得到了好的设计。

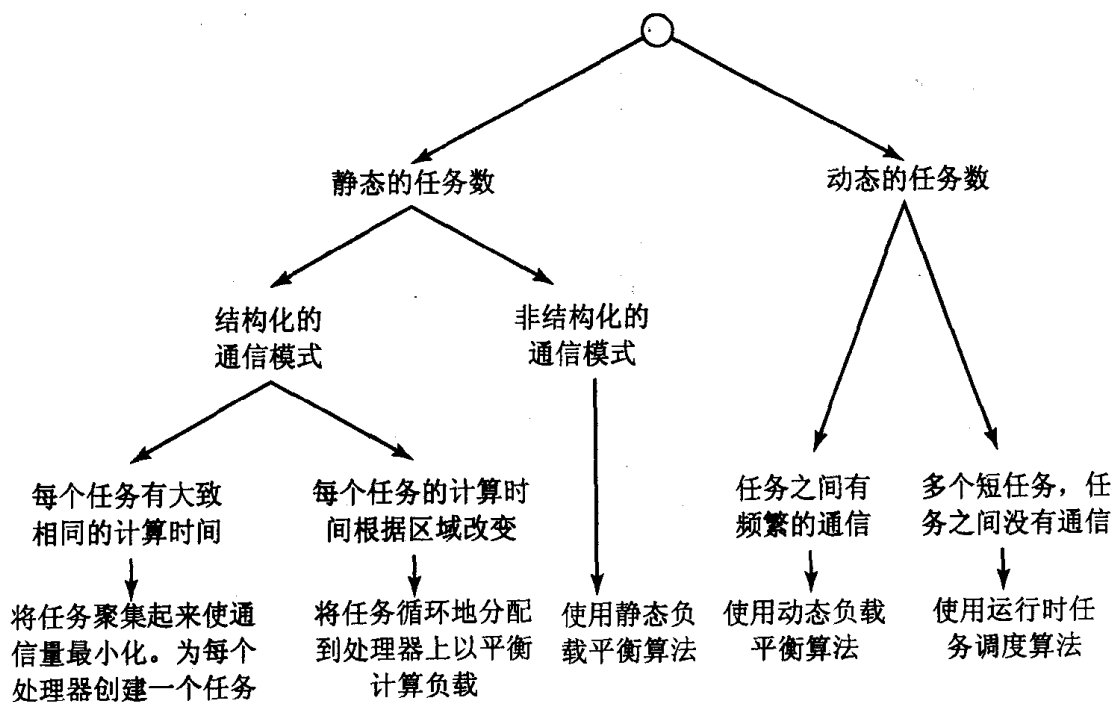


图 3.7 选择映射策略的决策树。最佳策略依赖于由划分、通信和聚集等步骤所形成的任务特征

- 已经考虑基于一个处理器对应一个任务和一个处理器对应多个任务的设计。
- 已经评估了静态和动态地将任务分配给处理器。
- 如果已经选择了将任务动态地分配给处理器，则管理者（任务分配者）不是性能的瓶颈。
- 如果已经选择了将任务静态地分配给处理器，则任务数与处理器个数的比例不低于 10:1。



## 3.4 边界值问题

### 3.4.1 简介

让我们将并行算法设计方法应用于一个简单然而实际的问题。如图 3.8 所示, 一根由同一材料制成的细棒被一块隔热体所包裹, 所以沿长度方向棒的温度变化是棒两端的热交换和沿长度方向的热传导共同的结果。棒长为 1。距离棒末端  $x$  处的初始温度为  $100\sin(\pi x)$ , 棒两端暴露于温度为  $0^\circ\text{C}$  的冰中。

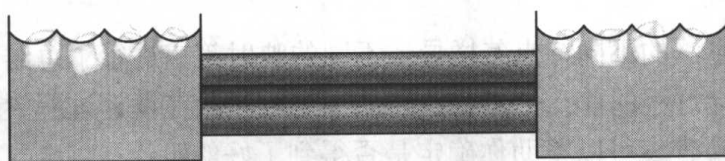


图 3.8 一条细棒 (深灰色) 悬于两块冰之间。棒的两端与冰水相接触。棒被一块厚厚的隔热体所包裹。可以用偏微分方程描述棒上任意一点处的温度的时间函数

随着时间的推进, 细棒最终冷却了下来。偏微分方程能够描述任何时刻、棒上任何一点处的温度。有限差分方法是在计算机上求解偏微分方程的一种方法。图 3.9 显示了棒冷却问题的一种有限差分近似。每一曲线都代表了某一时刻棒上的温度分布。曲线随时间的增加而下降。如果仔细观察, 你就能看到每条“曲线”实际上由 10 个线段组成, 在实际情况下, 温度的分布应该是平滑曲线。有限差分方法能够计算偏微分方程的近似解。

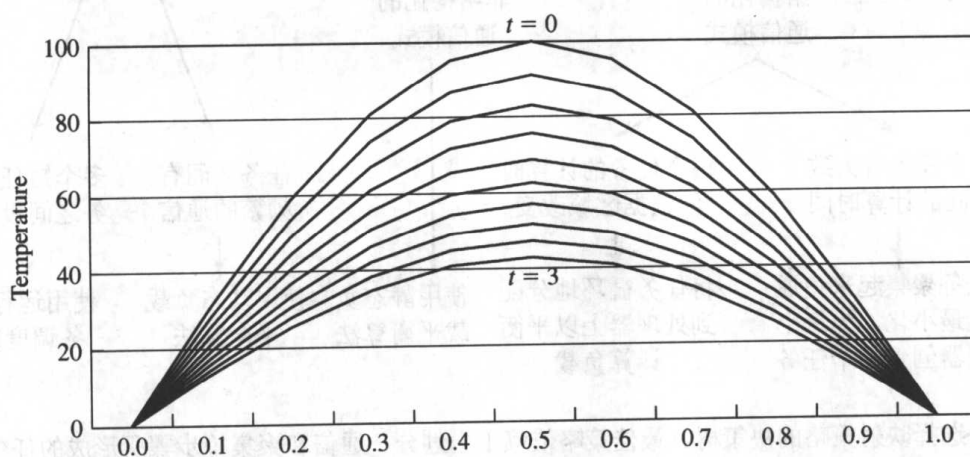


图 3.9 随着时间的推进, 棒冷却下来。有限差分方法能够找到某确定时间间隔棒中固定点数处的温度。减少空间和时间步长能够获得更为精确的解

求解该问题的有限差分方法将温度存储于一个二维矩阵中, 如图 3.10 所示。每行包含的是某时刻棒上温度的分布。棒被分为  $n$  段, 每段长  $h$ , 因此每行有  $n+1$  个元素。增加  $n$  可以减少近似误差。从 0 到  $T$  的时间段被划分为  $m$  个离散的实体, 每个长度为  $k$ , 因此二维矩阵含有  $m+1$  行。沿棒的长度方向初始温度分布可用最下一行中的点来表示。这些值是已知的。棒两端的温度可由网格的左边和右边来表示。这些值也是已知的。设  $u_{i,j}$  表示  $j$  时刻棒中  $i$  处的温度。

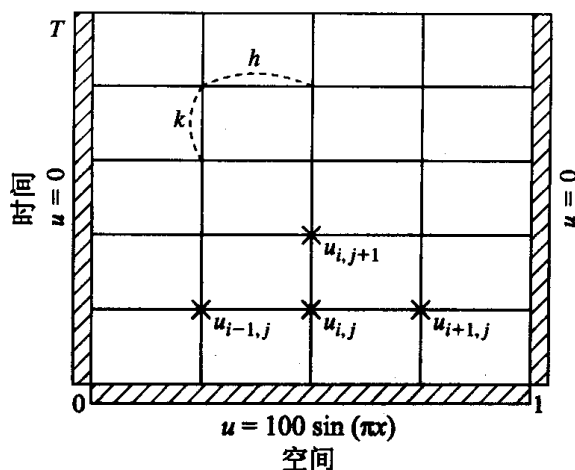


图 3.10 图 3.8 中出现的棒冷却问题中有限差分近似所用的数据结构。每个点  $u_{i,j}$  表示包含  $j$  时刻棒上  $i$  处温度的矩阵元素。棒的端点的温度总为 0。在 0 时刻，棒上  $x$  处的温度为  $100\sin(\pi x)$

在有限差分法中，算法随着时间推进，使用下列公式利用  $j$  时刻的值来计算  $j+1$  时刻的值：

$$u_{i,j+1} = ru_{i-1,j} + (1-2r)u_{i,j} + ru_{i+1,j}$$

其中  $r = k/h^2$ 。

### 3.4.2 划分

我们的第一步是进行划分。每个网格点处都有一个数据，这种情况很容易进行划分。我们每个网格点都与一个原始任务相联系，这将生成一个二维域分解。

### 3.4.3 通信

在确定了计算任务后，需要确定任务之间的通信模式。如果任务  $A$  需要来自任务  $B$  的值才能执行，那么需要建立一个从任务  $B$  到任务  $A$  的通道。由于计算  $u_{i,j+1}$  的任务需要  $u_{i-1,j}$ 、 $u_{i,j}$  和  $u_{i+1,j}$  的值，所以一般来说，每个任务都将有三个输入通道和三个输出通道，如图 3.11 (a) 所示，而边上的任务具有较少的通道。

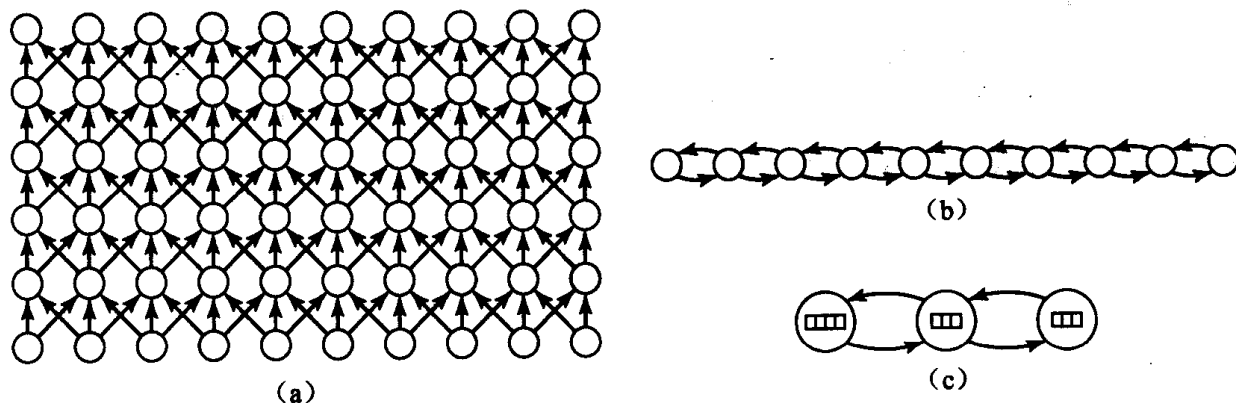


图 3.11 并行求解边界值问题的任务/通道图：(a) 第一个域分解将任务与将被计算的每个温度联系起来；(b) 进行第一次聚集之后，单个任务负责元素  $i$  在所有时间步的温度计算；(c) 进行第二次聚集之后，单个任务负责对棒上一组连续位置在所有时间步的温度计算

### 3.4.4 聚集与映射

即使能够获得足够的处理器，并发地计算图 3.11(a) 中的每个任务仍是不可能的。因为较后时间的任务依赖于较早任务的结果。从底部任务伸展到顶部任务的通道的垂直路径清楚地说明了这一点。如果必须被串行执行，保留多个任务的幻觉是毫无意义的。让我们聚集与棒上每一点有关联的所有任务（即图 3.11(a) 中位于相同列的任务）。

图 3.11(b) 中所示的任务/通道图得到了很大简化。现在，我们有一个一维的任务数组，每个任务仅仅与相邻的任务通信。每个任务对负责特定网格点上所有时间步的温度计算。

但是，即使是图 3.11(b) 中的任务数也可能大大多于我们可用的处理器数，因为在实际问题中棒的段数应该很大。我们可以使用图 3.7 中的决策树来决定映射策略。任务数目是静态的（左分支），它们之间的通信模式是规则的（左分支），且每个任务执行相同的计算（左分支）。因此，一个好的策略是为每个处理器创建一个任务，聚集原始任务使得计算负载均衡且通信最小。将棒的连续片断与每个任务相联系（如图 3.11(c) 所示），保留了任务之间简单的近邻通信，同时消除了单任务内部数据点之间进行的不必要的通信。

### 3.4.5 分析

棒已被划分为  $n$  片，每片的长度为  $h$ 。假设  $\chi$  表示在已知  $u_{i-1,j}$ 、 $u_{i,j}$  和  $u_{i+1,j}$  的情况下计算  $u_{i,j+1}$  所需的时间。使用单处理器修改  $n-1$  个棒的内部值需要的时间为  $(n-1)\chi$ 。因为该算法有  $m$  个时间步，所以执行串行算法的预期时间为  $m(n-1)\chi$ 。

现在让我们计算执行并行算法的预期时间。假设  $p$  为执行该算法的处理器数目。如果每个处理器所负责的长度相等，则每次迭代的计算时间为  $\chi((n-1)/p)$ 。然而，该并行算法包含了串行算法中没有的通信，因此我们必须计入通信带来的时间。一般来说，每个处理器必须将值发送给它的两个相邻处理器并从它们那里接收两个值。如果  $\lambda$  代表处理器将某值发送到另一个处理器或从另一个处理器接收到某值所需的时间，则对每次迭代来说通信将使得并行执行时间增加  $2\lambda$ 。在我们的任务/通道模型中，一个任务在一个时刻只能发送一条消息，但是在发送消息的同时它却可以接收一条消息。因此，该任务需要时间  $2\lambda$  将数据发送给它的两个邻居并同时从它的邻居处接收它所需的两个数据。

合并计算时间与通信时间，每次迭代的总并行执行时间为  $\chi((n-1)/p) + 2\lambda$ ，该算法所有  $m$  次迭代的并行执行的预计为  $m(\chi((n-1)/p) + 2\lambda)$ 。

## 3.5 找出最大值

### 3.5.1 简介

我们用于计算棒中温度分布（时间函数）的有限差分方法仅得到了偏微分方程的近似

解。我们使用有限差分或有限元方法来求解偏微分方程的原因在于，现实中的边界值问题非常复杂以至于无法获得解析解。

然而，在前一节中我们研究的热传导问题比较简单，可以获得解析解。这意味着我们能够确定沿棒  $m$  个点中每一点的计算解与真实解之间的差异。计算解  $x$  与真实解  $c$  之间的误差为  $|(x-c)/c|$ 。让我们增强并行算法以寻找最大误差。

给定  $a_0, a_1, a_2, \dots, a_{n-1}$  共  $n$  个值的一个集合和可结合二元算子  $\oplus$ ，归约 (reduction) 是计算  $a_0 \oplus a_1 \oplus a_2 \oplus \dots \oplus a_{n-1}$  的过程。加法是可结合二元算子的一个实例。因此求和操作  $a_0 + a_1 + a_2 + \dots + a_{n-1}$  为归约的一个实例。

你可能还没有认识到求最小值和求最大值是可结合的二元算子，因为在绝大多数编程语言中它们并未作为算子出现。然而，这两个可结合算子非常有用。例如，在我们正在考虑的问题中，我们就想找出一个集合的最大值。

由于对  $n$  个数的归约正好需要  $n-1$  次操作，在串行计算机上它的时间复杂度为  $\Theta(n)$ 。在并行机上我们怎样才能快速地执行归约操作呢？不失一般性，我们假设算子为加法操作，以便理解下面的解释。

### 3.5.2 划分

由于该序列有  $n$  个数值，所以我们将其分为  $n$  份。换句话说，我们要使划分尽可能地精细。如果我们将每份都与一个任务相联系的话，我们可以得到  $n$  个任务，每个任务对应一个数值。我们的目的是计算这  $n$  个数值之和。

### 3.5.3 通信

一个任务不能直接访问存储在另一任务内存中的数。为了计算这  $n$  个数之和，我们必须在任务之间建立通道。从任务  $A$  到任务  $B$  的通道允许任务  $B$  计算这两个任务所拥有值的和。我们希望通信和求和操作尽可能快地进行。在一个通信步中，每个任务既可发送也可接收一个消息。

在计算结束的时候，我们希望某个任务拥有这  $n$  个数的和，我们称该任务为根任务。我们由最原始的方法开始：除根任务之外的每一个任务都将其上的值发送给根任务，而根任务对所有的这些数进行相加，如图 3.12 (a) 所示。

如果将一个值传送给另一任务所需的时间为  $\lambda$ ，进行一次加法操作需要的时间为  $\chi$ ，则第一个并行算法需要的时间为  $(n-1)(\lambda+\chi)$ （通信时间为  $(n-1)\lambda$ ，这是因为根任务需要接收  $n-1$  个消息）。这实际上比串行算法还要慢。我们必须在通信与计算之间的取得更好的平衡。

如果两个任务协同执行归约又会怎样呢？假设我们有两个半根 (semi-root) 任务，则每个半根任务负责  $n/2$  个列表元素，如图 3.12 (b) 所示。现在，可以同时进行两个通信操作，且在每次通信之后，一次能够执行两个加法操作。在  $(n/2-1)(\lambda+\chi)$  时间内每个半根任务计算出各自一半元素的部分和。这时，其中一个半根任务将其部分和发送给另外一个半根任务。通过一个附加的通信/计算步，该任务可以得到  $n$  个数的总和。该并行算法的估

计执行时间为  $(n/2)(\lambda + \chi)$ 。

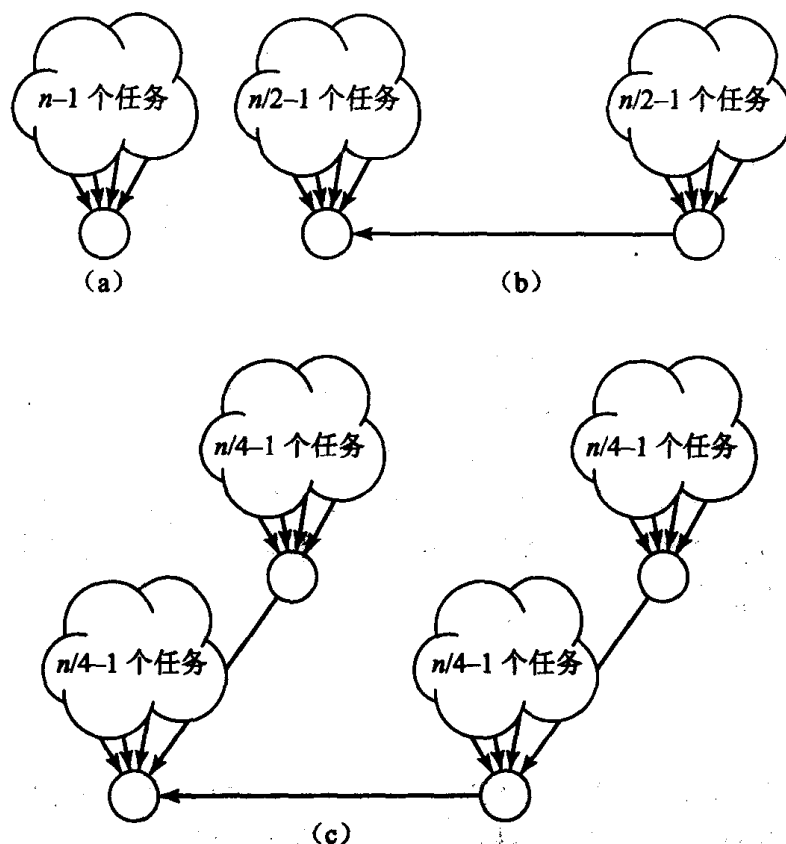


图 3.12 关于归约的有效并行算法的进化图。(a) 一个任务从其他  $n-1$  个任务处接收一系列元素并执行加法操作；(b) 两任务共同作用，每个任务都从  $n/2-1$  个其他任务处接收元素，在进行  $n/2$  次加法操作之后，一个任务将其上的部分和发送给第一个任务，与最初版本相比，计算时间几乎下降了一半；(c) 4 个任务协同工作。每个任务都从  $n/4-1$  个其他任务处接收元素。在进行  $n/4$  次加法操作之后，生成了 4 个部分和。在附加的两个通信/计算步内这 4 个部分和能够被合并

为什么不继续进行这个过程？如果有四个半根任务，其中每个负责半根任务负责  $n/4$  个列表元素又将发生什么呢？我们已经将通信和计算的并发度增加到 4。在计算出 4 个部分和之后，需要两个通信/计算步来求得总和。该算法的速度几乎是原算法的 4 倍。

如果我们将该想法扩展到极限的话，将有  $n/2$  个半根任务，每个任务负责两个列表元素。在该算法的第一步中，一半的任务发送消息到另一半任务。该步之后，接收到消息的任务能够同时将接收的数与自己拥有的数相加，从而将被加的数减少一半。

一个消息传递步足以将两个值合并为一个。两个消息传递步足以将四个值合并为两个。一般来说，在  $\log n$  个消息传递步内完成  $n$  个值的归约是可能的。图 3.13 图演示了具有 1、2、4 和 8 个节点的二项式树 (binomial tree)。在具有  $n=2^k$  个节点的树中，任何节点与位于左下角处根节点的最大距离为  $k=\log n$ 。二项式树是并行算法设计中最常见的通信模式之一。

图 3.14 展示了当通道结构为二项式树时，16 个任务如何在 4 个通信步内将它们的值进行合并。在第一步中，一半任务发送数值，一半任务接收数值。随后，发送数值的任务停止，算法对剩余的任务进行递归处理。剩余任务中的一半发送数值，同时剩余任务中的另一半接收数值，……直至只剩下一个任务。这个被称为根的任务得到本次归约的结果。

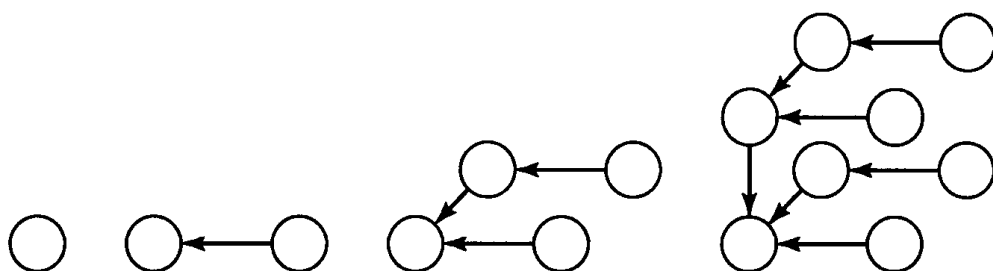


图 3.13 具有 1 个、2 个、4 个和 8 个节点的二项式树

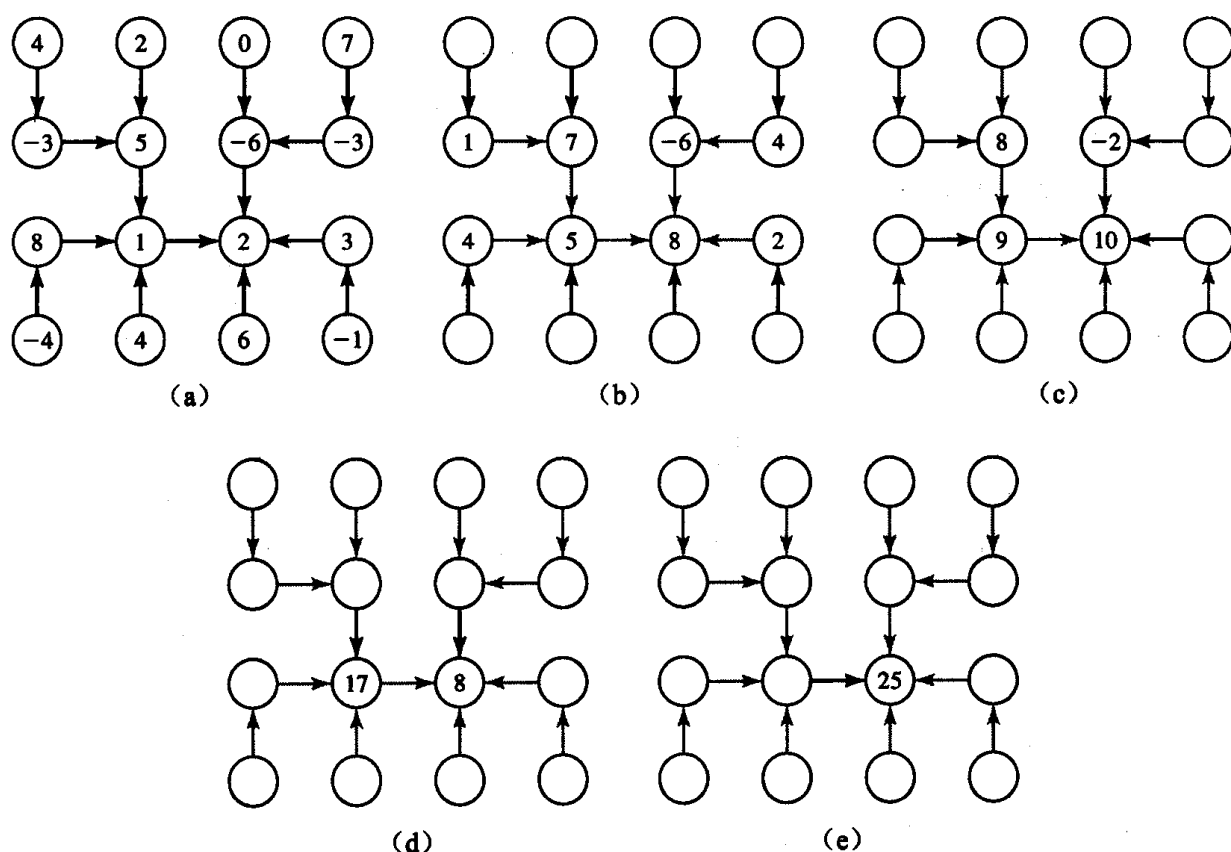


图 3.14 在对数时间内求全局和。(a) 结构为二项式树的任务/通道图，在被加数列表中每个整数对应一个任务；(b) 一半任务发送数值，剩余的一半任务接收数值并进行加法操作，随后发送数值的任务停止；(c) 四分之一的任务发送数值，四分之一的任务接收数值并进行加法操作，随后发送数值的任务停止；(d) 该过程递归进行，此时有两个发送数值的任务和两个接收数值的任务；(e) 在最后一步，一个任务发送数值，另一任务接收数值并执行加法操作，并得到全局和

如果任务数不是 2 的幂的话会怎样呢？在这种情况下，我们需要修改算法的第一步。假设任务数为  $n=2^k+r$ ，其中  $r<2^k$ 。在第一步中， $r$  个任务发送数值， $r$  个任务接收数值。当接收到数值之后，随后发送数值的  $r$  个任务停止。一旦完成了这一步，任务数变为  $2^k$  个，前面所述的算法可直接应用。

例如，考虑在 6 个任务中进行的归约，如图 3.15 所示。在第一步中，两个任务将数值发送给另外两个任务。此后，4 个任务拥有数值，因此归约便可在  $\log 4=2$  步中完成。

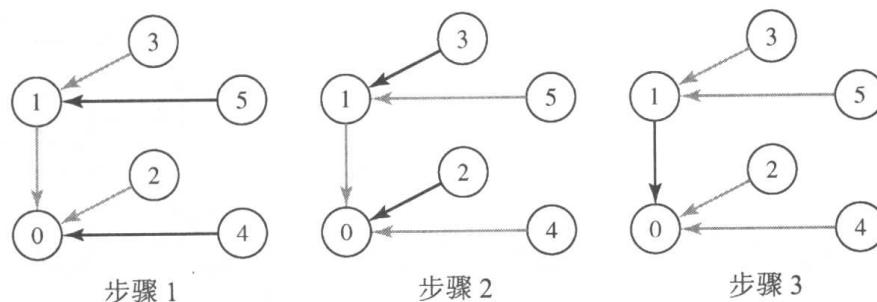


图 3.15 当任务数不是 2 的幂时进行归约的示例

我们看到如果任务数  $n$  是 2 的幂时, 归约能够在  $\log n$  次通信步内完成。如果  $n$  不是 2 的幂, 则需要  $\lceil \log n \rceil + 1$  个通信步。因此, 一般来说,  $n$  个任务进行归约所需的通信步为  $\lceil \log n \rceil$ 。

### 3.5.4 聚集与映射

图 3.16 (a) 复制了并行归约算法的通道/任务图。在将该算法实现为一个并程序之前, 我们需要将这  $n$  任务的图映射到  $p$  个处理器上去。为了简化我们的讨论, 假设  $p$  也是 2 的幂, 但  $p$  远小于  $n$ 。

任务数是静态的, 每个任务的计算量也是无足轻重的, 而且通信模式是规则的。使用如图 3.7 所示的映射决策树, 我们的结论是应该对任务进行聚集以使通信最小。我们可以通过为  $p$  个处理器中的每一个分配  $n/p$  个“叶子”任务来达到此目的, 如图 3.16 (b) 所示。

与对原始任务的聚集一样, 在一个物理处理器上保留多个相互通信的任务毫无意义。单个处理器上任务的目标是确定  $n/p$  个数值之和。因此我们并不需要  $n/p$  个原始任务, 使用一个具有  $n/p$  个数值的任务是更好的选择。结果如图 3.16 (c) 所示。该聚集的优点在于它能与在前一节中为边界值问题所选的聚集操作相匹配。这意味着我们能够将该增强功能很容易地添加到原有的并行算法中去。

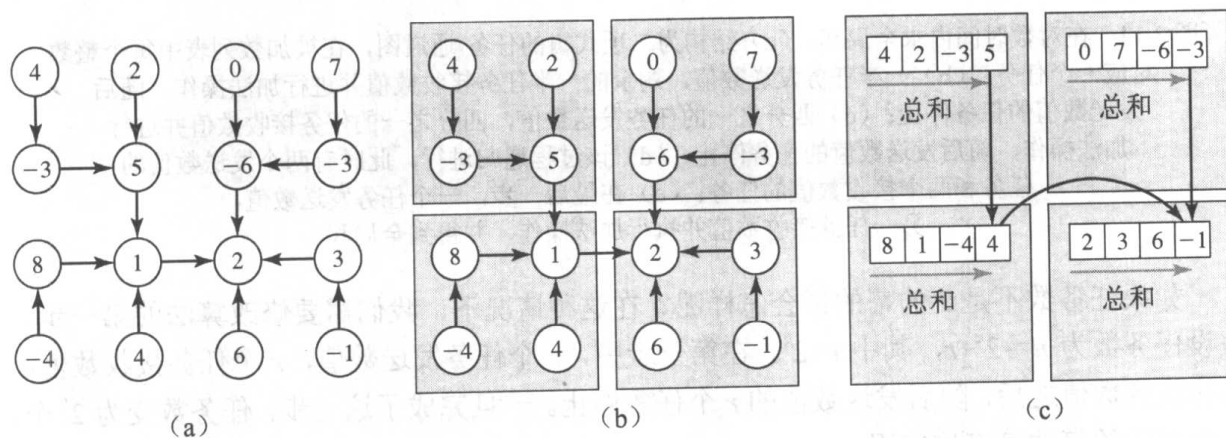


图 3.16 聚集的示例。(a) 并行归约算法的初始任务/通道图; (b) 16 个任务被映射到 4 个处理器上, 每个处理器上有相同的任务数, 处理器之间通信得到最小化; (c) 每个处理器上的 4 个任务被聚集为一个任务。在与其他任务进行通信之前, 每个任务使用串行算法求得本地的部分和

### 3.5.5 分析

现在，我们可以推导并行归约算法预计运行时间的表达式。定义如下的常数：

$\chi$ ：执行二元操作所需的时间；

$\lambda$ ：经由通道将一个整数从一个任务传送给另一任务所需的时间。

如果这  $n$  个整数被均匀地分布在  $p$  个任务中，则没有一个任务需要处理多于  $(n/p)$  个整数。由于所有的任务并发执行，因此所有任务计算其部分和所需的时间为：

$$(\lceil n/p \rceil - 1)\chi$$

分布在  $p$  个任务中的  $p$  个数值的归约能够在  $\lceil \log p \rceil$  个通信步内完成。接收数值的处理器必须不仅要等待消息的到达，而且还需要将它接收到的数值与其已拥有的数值进行相加。因此，每个归约步需要的时间为：

$$\lambda + \chi$$

由于有  $\lceil \log p \rceil$  个通信步，所以该并行算法总的执行时间为：

$$(\lceil n/p \rceil - 1)\chi + \lceil \log p \rceil (\lambda + \chi)$$

## 3.6 n-body 问题

### 3.6.1 简介

物理学中出现的一些问题可通过对数据集中所有物体对的计算进行求解。例如，在某些分子动力学问题中，作用在分子上的力可能存在库仑力或其他远范围的分量。在一个牛顿  $n$ -body 模拟过程中，重力有无限的作用范围。

求解这些问题的串行算法的每一次迭代的时间复杂度通常为  $\Theta(n^2)$ ，其中  $n$  为对象数目。对于  $n$ -body 问题，人们已经提出了许多具有更好的时间复杂度的算法。但我们更关心的是并行算法的开发，因此我们仅考察对每个物体对进行计算的串行算法，以便对其进行并行化。

为了给我们的讨论打下基础，假设我们正在求解一个  $n$ -body 问题。我们正模拟二维空间中不同质量的  $n$  个粒子的运动。在该算法的每次迭代过程中，已知所有其他粒子的位置，计算每个粒子的最新位置与速度，如图 3.17 所示。

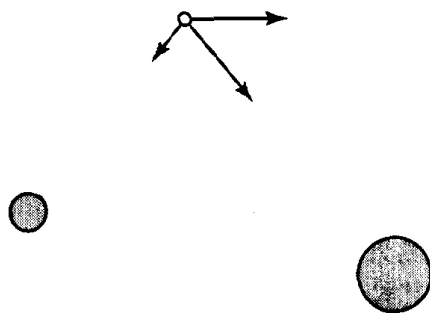


图 3.17 在  $n$ -body 问题中，每个粒子对其他每个粒子施加一个吸引力。在该二维示例中，白色的粒子有一个特定的位置与速度矢量（由黑色箭头表示）。它将来的位置受其他两个粒子所施加的吸引力影响

### 3.6.2 划分

第一步要做的工作是划分数据集。我们从假设每个粒子对应一个任务开始。为了计算一个粒子的最新位置，它需要知道所有其他粒子的位置。



### 3.6.3 通信

**gather** (收集) 为一全局通信操作, 它将散布于一组任务中的数据项收集到一个任务中, 如图 3.18 (a) 所示。与归约 (由数据元素计算一个单一的结果) 不同, **gather** 操作计算导致数据项的连接。**all-gather** (全部收集) 操作类似于 **gather**, 但在通信结束时每个任务均有整个数据集的一份副本, 如图 3.18 (b) 所示。

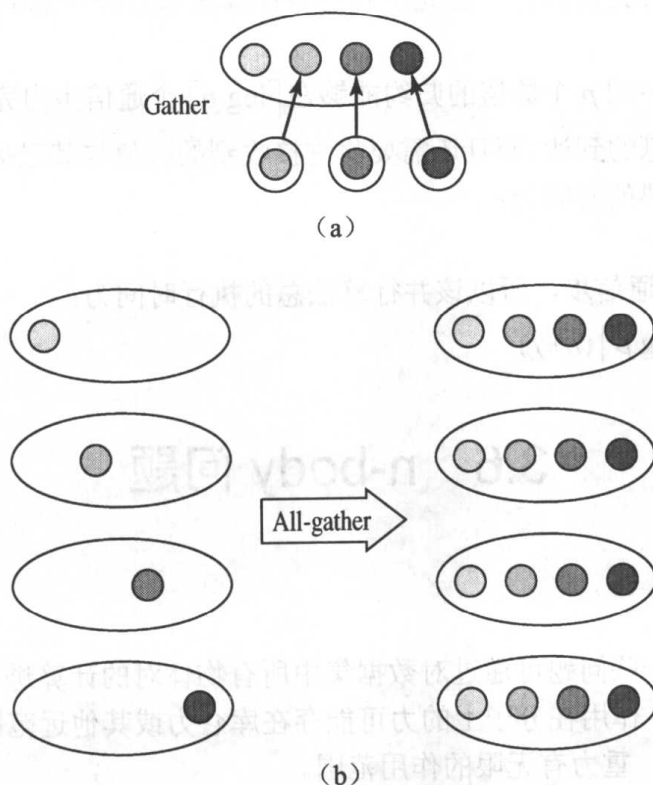


图 3.18 (a) **gather** 通信在一个任务上将数据项连接起来;  
(b) **all-gather** 通信在所有任务上建立起数据项的连接

在本例中, 我们希望更新每个粒子的位置, 因此需要一次 **all-gather** 通信。一个办法是: 在每对任务之间建立一条通道, 如图 3.19 所示。在每个通信步内, 每个任务将其矢量元素发送至另一个的任务。在  $n-1$  次通信步之后, 每个任务都有了所有其他粒子的位置, 可以计算其粒子的新位置和新速度矢量。

是否存在一种更快的方法使得所有的任务都能得到所有的数据值呢? 受并行归约算法的启示, 我们应当寻找在对数通信步内执行数据路由的方法。

我们通常使用自顶向下或自底向上的思路来考虑算法。在推导归约算法的过程中, 我们使用了自顶向下的思想。为了有所变化, 在这里我们尝试一种由底向上的办法。

假设仅仅有两个粒子。如果每个任务有一个

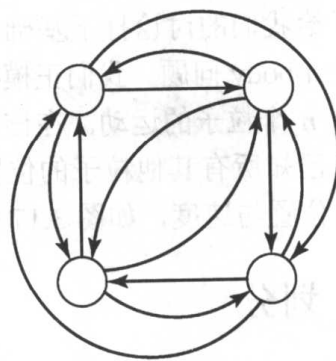


图 3.19 让所有数据值能被所有任务访问的一种方法是在每对任务之间建立一条通道

粒子, 则它们能够互换数值。每个任务通过一条通道发送一个数值并从另外一条通道接收一个数值。有四个粒子时情况如何呢? 在一个互换步之后, 任务 0 和任务 1 都能够获得粒子  $v_0$  和  $v_1$  的数值, 任务 2 和任务 3 都拥有粒子  $v_2$  和  $v_3$  的数值。如果现在任务 1 将其上的粒子对与任务 3 进行交换的同时, 任务 0 将其上的粒子对与任务 2 也进行交换, 则所有任务都将有所有四个粒子的数值。图 3.20 展示了该改进后算法对应的任务/通道图。

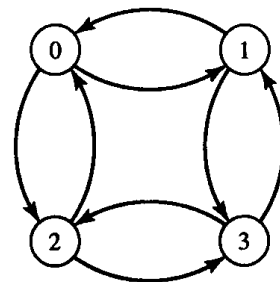


图 3.20 all-gather 通信需要每个任务仅有  $\log p$  个输出通道和  $\log p$  个输入通道

对数级的互换步是必需的且足以使得每个处理器获得原本属于其他处理器的数值。在第一个互换步内, 消息的长度为 1。在第二个互换步内, 消息的长度为 2。在第  $i$  个互换步内, 消息的长度为  $2^{i-1}$ 。

图 3.20 显示的任务/通道图是一个超立方体网络（在第 2 章中首次碰到）的例子。超立方体形式的任务/通道图常常出现于各种 all-to-all 数据开关的有效算法实现中。

### 3.6.4 聚集与映射

一般来说, 粒子数  $n$  要远远多于处理器数目  $p$ 。假设  $n$  是  $p$  的倍数, 我们为每个处理器分配一个任务并将  $n/p$  个粒子聚集为一个任务。则 all-gather 通信操作需要  $\log p$  个通信步。在第一步中, 消息的长度为  $n/p$ 。在第二步中, 消息的长度为  $2n/p$ , 等等。

### 3.6.5 分析

现在我们能够推导出该算法预期执行时间的表达式。在前面的例子中, 我们假设该算法消耗  $\lambda$  单元时间来发送一个消息。然而, 在这些示例中, 消息的长度总为 1。现在, 消息的长度要大得多。期望发送或接收一个消息所需的时间与消息长度无关是不现实的, 因此我们将为我们的消息传递时间公式增加一个新项。从现在开始,  $\lambda$  (延迟) 将表示初始化一条消息所需的时间。假设  $\beta$  (带宽) 表示单位时间内能够沿通道发送数据项的数目。发送一条包含  $n$  个数据项的消息需要的时间为  $\lambda + n/\beta$ , 如图 3.21 所示。注意, 随着带宽的增加, 通信时间将下降。

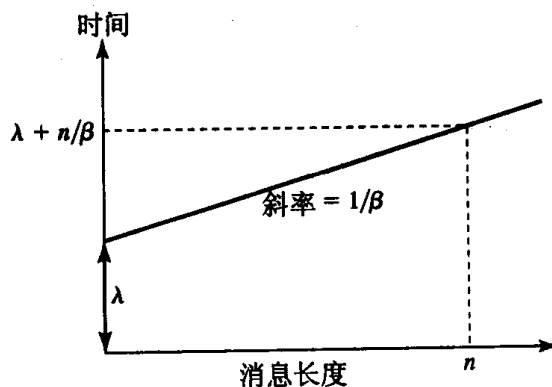


图 3.21 发送一条长度为  $n$  的消息所需的时间可用一线性函数  $\lambda + n/\beta$  来描述, Y 轴上的截距  $\lambda$  为消息延迟, 斜率的倒数  $\beta$  为通信系统的带宽

该算法每次迭代中通信时间为:

$$\sum_{i=1}^{\log p} \left( \lambda + \frac{2^{i-1}n}{\beta p} \right) = \lambda \log p + \frac{n(p-1)}{\beta p}$$

每个任务对  $n/p$  个列表元素的吸引力计算负责。假设该计算所需时间为  $\chi$ , 则该并行算法每次迭代所花的计算时间为  $\chi(n/p)$ 。

将该并行算法的通信时间与计算时间放在一起考虑的话, 我们可以推导出每次迭代的预计执行时间为:

$$\lambda \log p + n(p-1)(\beta p) + \chi(n/p)$$

## 3.7 增加数据输入

### 3.7.1 简介

大多数程序都输入和输出数据, 但我们所定义的任务/通道模型并没有解决数据的输入/输出问题。下面让我们考虑一下如何为我们刚刚开发的  $n$ -body 算法增加输入和输出。我们的方法是为基本模型增加 I/O 通道。

假设我们的并行算法将输入  $n$  个粒子的原始位置和速度矢量。商用并行计算机常有并行 I/O 系统, 但是商用的集群却常常使用外部的文件服务器来存储普通 Unix 文件。因此, 我们不考虑任何并行 I/O, 并假定一个单一的任务负责执行文件 I/O 操作。

我们在图 3.20 给出的  $n$ -body 问题的任务/通道图上增加代表文件 I/O 的新通道, 结果如图 3.22 所示。我们赋予执行文件 I/O 操作的任务一个明显的名字: I/O 任务。请注意我们并没有增加任务, 而只是将一些额外的责任分配给了任务 0。

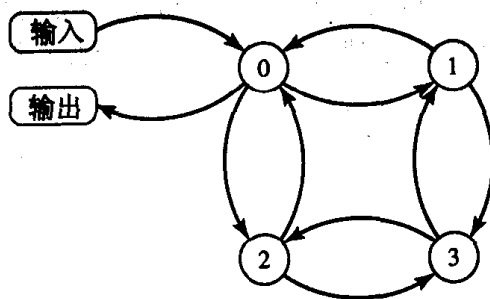


图 3.22 对该任务/通道图进行了扩展, 任务 0 负责 I/O。  
I/O 设备以圆框矩形的形状出现在该图中

I/O 任务需要打开数据文件并读取  $n$  个粒子的位置和速度。由于我们正在进行二维模拟, 所以一对坐标就能够识别出一个粒子的位置, 同时其速度能够通过另外一对值来表示。如果模拟输入或输出  $n$  个数据元素所需的时间为  $\lambda_{io} + n/\beta_{io}$ , 则读出所有  $n$  个粒子的位置和速度所需的时间为  $\lambda_{io} + 4n/\beta_{io}$ 。

### 3.7.2 通信

在 I/O 任务输入粒子的位置和速度之后,我们必须解决如下问题:如何将输入数据划分为数据片,使得分配给每个任务的子段都包含有  $n/p$  个元素?这里所使用的全局通信操作被称为 scatter (散发)。你能看出 scatter 操作像 gather 的反操作吗?

对于 I/O 任务来说,Scatter 粒子的一种方法是由 I/O 任务依次将  $n/p$  个粒子发送给每一个其他任务。换句话说,它将发送  $p-1$  个任务,每个任务的长度为  $4n/p$ ,需要的时间为:

$$(p-1)(\lambda+4n)(p\beta)$$

这并非是一个高效的算法,因为处理器之间的通信并不均衡。

通过使用本章中已经使用过好几次类似过程,我们可以推导出需要  $\log p$  个消息步的 scatter 操作。在第一步时, I/O 任务将一半数据发送到另一个任务。在第二步中,每个拥有一半数据的任务将四分之一数据发送给先前的非活跃任务。现在四个任务中的每一个都拥有四分之一数据。在第三步中,拥有四分之一数据的这四个任务又发送八分之一的数据到四个先前的非活跃任务,并依此类推。执行该算法所需的时间为:

$$\sum_{i=1}^{\log p} (\lambda + 4n/(2^i p\beta)) = \lambda \log p + 4n(p-1)/(\beta p)$$

现在我们已经了解到关于 scatter 算法的两种不同设计。在第一个算法中,一个任务串行地发送  $p-1$  条消息到其他任务。这需要时间  $(p-1)\lambda+4n(p-1)/(\beta p)$ 。

第二个算法大概需要执行  $\log p$  步(这并不准确,因为  $p$  可能并不是 2 的整数幂)。总的通信时间为  $4n(p-1)/(\beta p)$ 。该算法要优于第一个算法。

注意,两个算法中的数据传输时间(含有  $\beta$  的项)是相同的。在第一个算法中,每个粒子被直接传送给对其负责的任务。在第二个算法中,粒子被重复地移动。典型的粒子大约在  $\log p$  个消息中被传送。那么,为什么基于第二个算法的程序在传输数据上所花费的时间较基于第一个算法的程序要少呢?这是因为任务/通道模型支持多个任务的并发消息传递,只要这些消息传递使用不同的数据通道,而且没有两个活跃通道有相同的源或目的任务。这是对商用并行系统的合理假设,也是对集群系统的合理假设,其中每个处理器与具有足以支持处理器间许多并发消息的底板速度的某个开关直接相连。但它并不是对工作站网络的合理假设,因为工作站网络常常通过网络集线器或其他共享通信介质相连,在任意时刻仅支持单一消息传递。

### 3.7.3 分析

我们现在能够推导出并行 n-body 算法总期望执行时间的一个表达式。 $N$  个粒子的位置和速度的输入和输出是完全的串行操作,需耗时:

$$2(\lambda_{io}+4n/\beta_{io})$$

在算法开始时 scatter 粒子信息并在计算结束时, gather 粒子信息所需时间为:

$$2(\lambda \log p + 4n(p-1)/(\beta p))$$

并行算法的每次迭代需要一次有关粒子位置的 all-gather 通信。该算法的一个实现的执

行时间近似为:

$$\lambda \log p + 2n(p-1)/(\beta p)$$

最后, 每个处理器执行属于它的计算。每次迭代预期的执行时间为:

$$\chi \lceil (n/p) \rceil (n-1)$$

假设算法执行  $m$  次迭代, 则该并行计算期望的总执行时间为:

$$2(\lambda_{io} + 4n/\beta_{io}) + 2(\lambda \log p + 4n(p-1)/(\beta p)) + m(\lambda \log p + 2n(p-1)/(\beta p)) + \chi (n/p) (n-1)$$

## 3.8 本章小结

本章所描述的任务/通道模型将并行计算描述为一系列任务, 任务之间以及任务和 I/O 设备之间可以沿着通道发送消息进行交互。该模型是有用的, 因为它鼓励最大化本地计算和最小化通信的并行算法设计, 这些设计更适合于分布式并行计算机。

在使用任务/通道模型开发并行算法的过程中, 算法设计者一般需要划分计算和识别原始任务之间的通信, 将原始任务聚集为更大的任务并决定如何将任务映射到处理器。该过程的目标是通过在处理器间分布计算步的方法最大化处理器的利用率, 同时在最小化处理器间的通信。要达到一个目标, 必须要在一定程度上放弃另一个目标, 因此好的设计都必须在这两个目标之间达成平衡。

归约是对数据集进行可结合二元操作的应用。并行算法常常需要进行归约操作, 例如求解分布于所有任务中数值的总和。我们开发出了具有对数时间复杂度并行归约算法。归约的任务/通道图表现为二项式树。

我们还开发了一个有效的并行算法来完成 all-gather 操作, 它将从整个任务集收集来的数值连接起来, 并让每个任务都拥有一份副本。我们的算法仅需要对数级的通信步数。它依赖于一个形如超立方体的任务/通道图。

最后, 我们考虑了将单任务所拥有的数据 scatter 至任务集的问题及其将分布于任务集的数据收集到一个任务中的反问题。当通信时间主要由消息延迟决定时, 二项式树是适合于 scatter 与 gather 操作的任务/通道图结构。

## 3.9 主要术语

agglomeration

functional decomposition

primitive task

all-gather

gather

processor utilization

asynchronous

global communization

聚集

功能分解

原始任务

全收集

收集

处理器利用率

异步的

全局通信

reduction	归约
binomial tree	二项式树
increasing locality	增加局部性
scatter	散发
blocked task	阻塞式任务
local communication	局部通信
synchronous	同步的
channel	通道
mapping	映射
task	任务
domain decomposition	域分解
partitioning	划分

### 3.10 参 考 文 献

Foster 所著的 *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering* 一书的第一部分讲解了基于任务/通道模型进行并行算法设计的方法论【31】。该书对本章所描述的划分、通信、聚集与映射这四步设计过程进行了详细的论述，并包含了很多的案例研究，以阐明该方法。

Carriero 和 Gelernter 提出了另一种不同的并程序序设计观点。他们所著的书 *How to Write Parallel Programs: A First Course* 描述了如下三种并行算法设计范例：计算结果的并行性（result parallelism）、专家并行性（specialist parallelism）和议程并行性（agenda parallelism）【15】。从另一个角度来看并行算法是增强“并行思考”能力的一条好途径。

Valiant 提出了批量同步并行模型（bulk synchronous parallel (BSP) model），作为一种填补并行软件和硬件之间鸿沟的方式【107】。BSP 模型的设计目标是，它能够提供给并行算法设计者的好处与冯诺伊曼模型带给串行算法设计者的好处相同。BSP 计算由一系列的超步构成。每个超步包含一系列步骤，处理器在这些步骤中进行对本地数据的计算，随后是阻塞同步，进行非本地数据的交换。有关 BSP 的更多信息，请查询 BSP 全球网址：[www.bsp-worldwide.org](http://www.bsp-worldwide.org)。

### 3.11 练 习 题

3.1 请举出一个示例来说明提高处理器利用率是如何增加处理器间通信的。

3.2 计算变量  $n$  的下述取值所对应的  $\log n$ 、 $\lfloor \log n \rfloor$  和  $\lceil \log n \rceil$ ：

- (a) 3
- (b) 13
- (c) 32

(d) 123

(e) 321

3.3 就下列规模画出二项式树:

(a) 16 个节点

(b) 32 个节点

3.4 就下列规模画出超立方体并标注出节点:

(a) 16 个节点

(b) 32 个节点

3.5 给定一个四维超立方体, 画出四个不同的子图, 其中每个图都为一个具有 16 个顶点的二项式树。所有四棵树其树根都应同一超立方体节点。在每个图中, 请标出未被使用的超立方体的边。

3.6 举例说明如何在  $\lceil \log n \rceil$  通信步中执行一次归约操作。其中  $n$  取值分别为 7、11 和 21。

3.7 使用图 3.15 所阐明的通信模式作为指南, 编写一个 C 程序来描述参与归约的任务所执行的通信。给定任务数  $n$  和某任务特定的识别号  $i$ , 其中  $0 \leq i < n$ , 该程序应该打印由任务  $i$  发送和或接收的一系列消息。该消息列表应该表明所有发送消息的目标任务和所有接收消息的源任务。

例如, 在  $n=6$  和  $i=1$  情况下, 该程序的输出应该为:

从任务 5 处接收到的消息;

从任务 3 处接收到的消息;

发送到任务 0 的消息。

3.8 证明在任务/通道模型上执行一个  $n$  元素归约的时间复杂度为  $\Omega(\log n)$ 。

3.9 许多并行算法都需要一个广播步骤, 其中一个任务将其拥有的一个值发送给所有的任务。

(a) 使用本章所描述的任务/通道模型, 设计一个有效并行算法来实现广播操作。

(b) 证明 (a) 中你所设计的算法有最优时间复杂度。

3.10 我们所开发的 scatter 算法仅仅将大约  $n$  个值发送给  $p$  个任务, 而 all-gather 算法将  $n$  个值发送给  $p$  个任务。然而这两个算法都具有的时间复杂度  $\Theta(n + \log p)$ 。请解释。

3.11 设计一个并行算法来执行一次 all-to-all 的交换。有  $p$  个处理器, 其中  $p$  是 2 的幂。进程所处理的矢量长度为  $p$ 。设  $X_{i,j}$  表示处理器  $i$  所控制矢量的第  $j$  个元素。每个进程都从拥有矢量  $A$  开始, 交换的结果是得到矢量  $B$ , 其中  $B_{i,j} = A_{j,i}$ 。换句话说, 处理器  $i$  所拥有的矢量  $B$  为所有进程中矢量  $A$  中第  $i$  个元素的连接。你的算法的复杂度是多少?

3.12 冒泡排序算法通过对相邻元素进行重复比较的办法实现数组  $a[0], a[1], \dots, a[n-1]$  的排序。如果  $a[i] > a[i+1]$ , 则交换这两个元素。该过程持续进行直到  $a[0] < a[1] < \dots < a[n-1]$ 。使用任务/通道模型, 设计并行版的冒泡排序算法并为该并行冒泡排序算法画出如下两个任务/通道图: 第一个图显示原始任务; 第二个图应显示聚集后的任务。

3.13 黑白图被存储为 0 和 1 的  $n \times n$  数组。其中 1 代表物体, 0 代表物体之间的空白空间。组件标签问题是给每个物体分配一个惟一的正数。在该算法结束时, 每个 1 位像素都将有一个正整数标签。一对 1 位像素具有相同的标签, 当且仅当它们在同一组件 (物体)

中。如果1位像素由1位像素组成的路径连接,那么它们属于同一组件。如果两个1位像素在水平方向或垂直方向相邻,则称它们是连续的。使用任务/通道模型,设计一个并行算法来解决组件标签问题。并请画出如下两个任务/通道图。第一个图应显示原始任务,第二个图应显示聚集后的任务。

3.14 给定一个纵横字谜和一本字典,设计一个并行算法来寻找所有可能的填充方式,使得每个水平和垂直的字空间都包含有来自该字典中的单词。

3.15 给定一个有 $n$ 个记录的数组,每个记录都包含有一栋房屋的 $x$ 和 $y$ 坐标。另外给定一个火车站的 $x$ 和 $y$ 坐标。设计一个并行算法来寻找与火车站最近的那栋房屋(按乌鸦飞行的距离)。并请画出如下两个任务/通道图:第一个图应显示原始任务;第二个图应显示聚集后的任务。

3.16 字符串匹配问题是寻找特定子字符串(该子字符串被称为模式,而在另一字符串中被称为文本)所有可能的形式。设计一个并行算法来求解该字符串匹配问题。

3.17 重新考虑在上题中出现的字符串匹配问题。假设你仅仅对文本中该模式的第一次出现感兴趣,应该如何改变该并行算法的设计?

3.18 给定共 $n$ 个元素的一个序列 $a[0], a[1], \dots, a[n-1]$ ,所有元素具有不同的值。请设计一个并行算法来找出该序列中的第二大元素。

3.19 给定共 $n$ 个元素的一个序列 $a[0], a[1], \dots, a[n-1]$ 。请设计出一个并行算法来找出该序列中的第二大元素。注意,不同元素的值可以相同。



## 第 4 章 消息传递编程

The voice of Nature loudly cries  
And many a message from the kies,  
That something in us never dies.

Robert Burns

### 4.1 概 述

过去的 40 年间出现了很多并行编程语言，其中大多是人们通过简化并行管理的不同方面而提出的高级语言。尽管如此，到目前为止还没有一个高级并行编程语言为业界所广泛接受。然而，人们通过在 C 或者 Fortran 语言中增加进程间消息传递函数，完成了大多数的并行程序设计。MPI 标准是最流行的并行编程消息传递规范，几乎所有商业的并行机都支持它，同时也有众多支持 MPI 标准的开放软件库可供本土的商业集群使用。

从本章开始，我们将用连续三章的内容介绍基于 C 语言的并行 MPI 编程。本章将以电路可满足性问题为例子，来介绍如何设计、实现和分析一个简单的并行程序。我们首先来看以下几个函数。

- MPI\_Init: 初始化 MPI;
- MPI\_Comm\_rank: 确定进程的标识符;
- MPI\_Comm\_size: 确定进程数量;
- MPI\_Reduce: 执行归约操作;
- MPI\_Finalize: 结束 MPI;
- MPI\_Barrier: 执行一次同步操作;
- MPI\_Wtime: 确定当前时间;
- MPI\_Wtick: 获取计时器的精度。

### 4.2 消息传递模型

消息传递编程模型和我们在第 3 章描述的任务/管道模型相似，如图 4.1 所示。其底层硬件是一组处理器，每个处理器有自己的内存且只能直接访问本地的指令和数据。同时，一个互连网络支持各处理器之间进行消息传递。处理器 A 可以发送一个包含本地数据的消息给处理器 B，这样就实现了处理器 B 对非本地（即处理器 A）的数据的访问。

任务/管道模型中的任务对应着消息传递模型中的进程，互连网络的存在就意味着每两个处理器之间都有一个管道连通，也就是说每个处理器都可和其他任何一个处理器进行通

信。但是，我们仍然希望利用我们所掌握的策略来控制通信开销，从而避免破坏并行程序的效率。

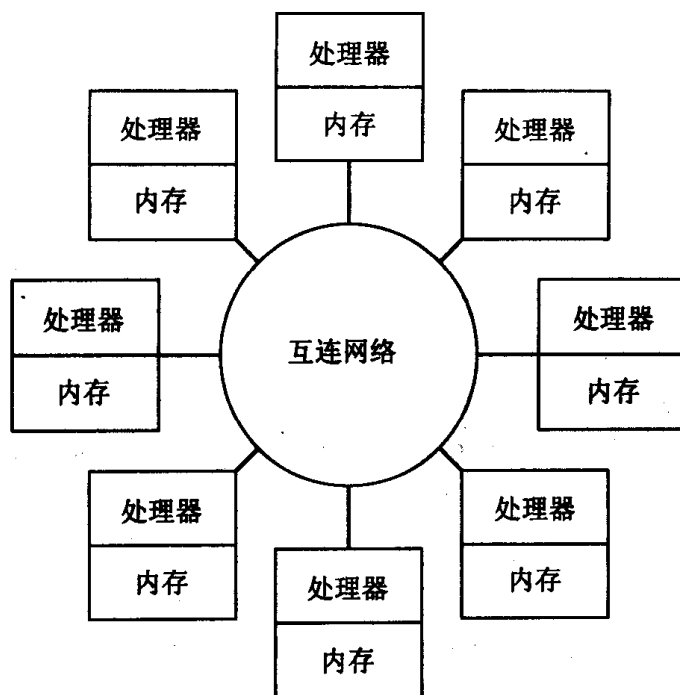


图 4.1 消息传递模型假设底层的消息传递模型是一组处理器，每个处理器有自己的本地内存，并且通过互连网络实现与其他处理器的消息传递

程序开始时，用户将指定并发的进程数，通常，在程序执行过程中，活动的进程数将保持不变。每一个进程执行着同一个程序，但是，由于每一个都有一个惟一的 ID 值，在程序展开之后不同的进程可以执行不同的操作。一个进程或执行针对其局部变量的操作，或与其他进程及 I/O 设备进行通信，两个过程可交替进行。

消息传递模型中一个很重要的概念是：进程传递消息的目的既在于相互通信，也在于彼此保持同步。当一个含有数据的消息从一个进程传递到另一个进程时，很明显，其作用在于通信。同时，一个消息也能起到同步的作用。只有在进程 A 向其发送了某个消息后，进程 B 才可能接收到来自 A 的消息。因此，收到消息的同时，进程 B 也获得了有关进程 A 的状态信息。正因为如此，一个空消息也是有其意义的。

并行计算的消息传递模型的提倡者强调，它与其他并行计算模型相比有几个优点。首先，消息传递程序在许多 MIMD（多指令多数据）架构的机器上能很好运行。由于多机系统不支持全局的地址空间，消息传递机制可以与它们很好的配合。同时，在多处理器计算机上，也可通过把共享变量用作消息缓存的办法，使得消息传递程序也可以运行。事实上，消息传递模型在快速的、可直接访问的本地存储和慢速的、间接访问远程存储上的差别，鼓励设计者开发最大化本地计算的同时，最小化通信。最终的程序在运行于多处理器计算机上时表现出较高的 Cache 命中率，从而能达到较高的性能。从另外一个角度讲，消息传递模型给多处理器计算机的程序员提供了可用于管理多层次存储的工具。

其次，调试消息传递模型的程序要比调试共享变量的程序更为简便。由于每个进程管理它自己的内存，某个进程就不可能意外地覆盖另外一个进程管理的变量，而这是共享变量程序的普遍错误所在。不确定的执行（例如，在多次程序执行过程中，不同的进程以不

同的顺序访问同一个资源) 将使调试过程复杂化。在消息传递模型中, 构建一个能确定执行的程序更为容易。

## 4.3 MPI 接口

20 世纪 80 年代末, 很多公司开始制造并出售多计算机系统。通常, 这种系统的编程环境由一种串行语言 (通常是 C 或 Fortran 语言) 以及一个使之能支持进程间通信的消息传递库扩展所组成。每一个供应商都有自己的函数调用接口, 这意味着一个为 Intel iPSC 开发的程序将不能在一个 nCUBE/10 上编译运行。程序员不满意于这种缺乏可移植性的做法, 于是几年后便制定了大量支持并行计算机的消息传递库的标准。

1989 年夏, 第一个版本的消息传递库在 Oak Ridge 国家实验室完成, 名为 PVM (并行虚拟机)。PVM 使得在异构的串行及并行计算机之间的程序的执行变得更为方便。原始的版本在 Oak Ridge 国家实验室内部使用, 并没有发布。开发小组的成员重写了两次并在 1993 年 3 月发布了第 3 版【39】。PVM 立即在并行程序员中广泛流传。

同时, 并行计算研究中心 (Center for Research on Parallel Computing) 与 1992 年 4 月赞助了分布式内存环境下消息传递标准研究小组组 (Workshop on Standards for Message Passing in a Distributed Memory Environment)。这个研究小组吸引了主要来自欧洲和美国的 40 个组织的 60 个人。大部分主要的多计算机供应商, 以及来自大学、政府实验室、工业界的研究人员都在其中。这个研究小组讨论了标准消息传递接口的基本特征, 并建立了一个工作小组来继续标准化进程。1992 年 11 月, 一个初步的草案问世。从 1992 年 11 月到 1994 年 4 月的消息传递接口论坛讨论并细化了草案。MPI 论坛并没有简单地采用一个现有的消息传递库接口 (如 PVM 或某个商业供应商的库), 而是试图寻找最佳的特征。该标准的 1.0 版出现于 1994 年 5 月, 通常称为 MPI。随后, 改进标准的工作继续进行, 尤其是并行 I/O 的加入, 以及与 Fortran 90 和 C++ 的绑定。MPI-2 于 1997 年 4 月形成。

今天, MPI 已经成为最流行的用于并行编程的消息传递库标准。在大部分的商用多计算机系统中都有 MPI。对于那些用现成系统构建他们自己的多计算机系统的人来说, 免费版的 MPI 库可以从 Argonne 国家实验室的网站或其他网站上下载。

尽管在不同的计算机之间, 特定程序的性能会有很大变化, 但使用 MPI 编写并行程序使你可以将它们移植到不同的并行计算机上。

## 4.4 电路可满足性问题

作为 MPI 的入门问题, 我们将设计一个程序, 用于计算图 4.2 中的电路是否是可满足的。换句话说, 就是什么样的输入组合 (如果存在) 可以使输出结果为 1? 这种电路可满足性问题在逻辑设备设计和验证中是很重要的。不幸的是, 这是一个 NP 完全问题, 意味着没有一个已知的多项式时间复杂度的算法可以解决这个问题【38】。

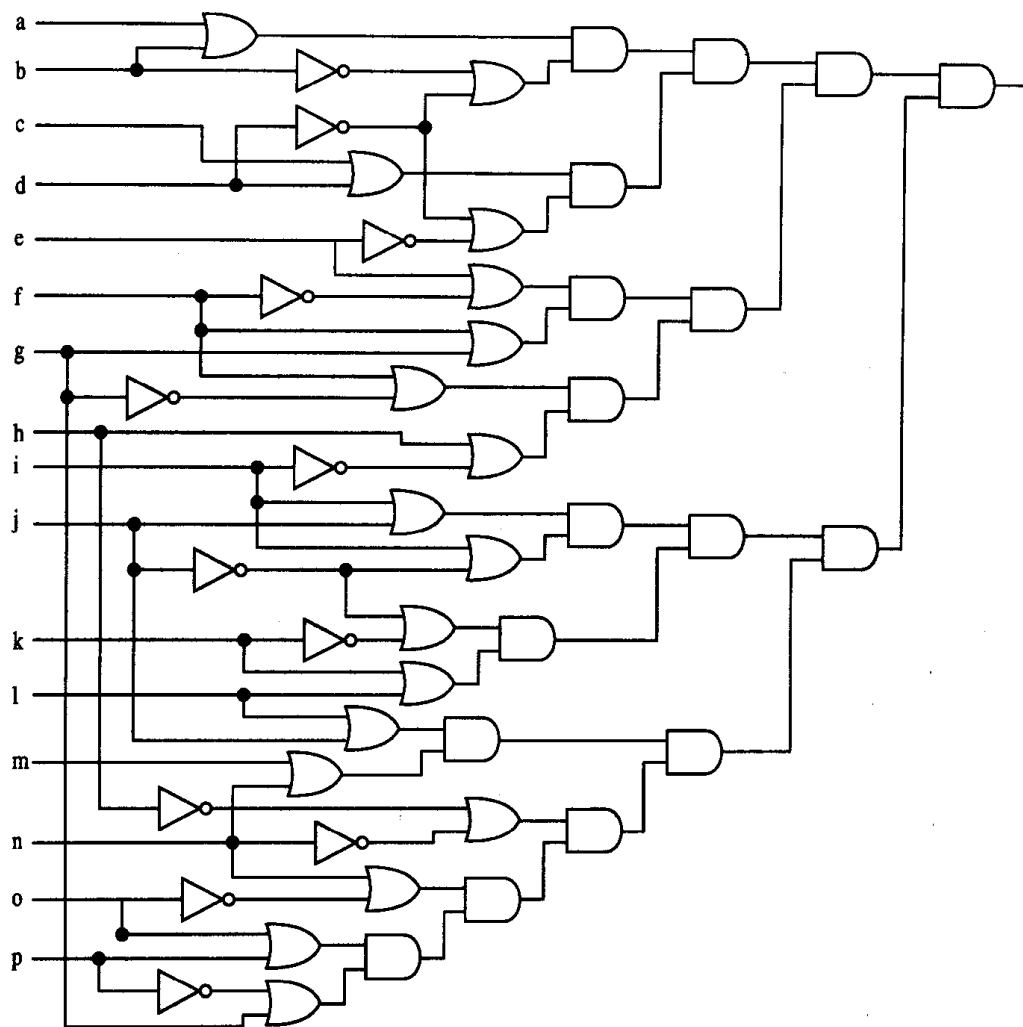


图 4.2 一个包含与、或、非门的电路。电路可满足性问题是判断是否有某种输入的组合可以使输出为 1

一种解决问题的方法是尝试每一种输入组合。该电路有 16 个输入，分别标记 a 到 p，每一个输入可以选两个值，0 和 1，因此有  $2^{16}=65\,536$  种输入组合。

正如我们在第 3 章中所了解到的，开发并行算法的第一个步骤是划分。并行性究竟在哪里？在这个问题中，可并行的部分很容易发现。我们需要检测 65 536 种输入组合的每一中来看它们的结果是否导致输出为 1。很自然，可以做功能分解。我们将每一种组合和一个任务相联系。如果一个任务发现其输入组合导致电路输入结果 1，它将打印该组合。因为所有的任务都是独立的，这个可满足性检测可以并行执行。

这是一个很合适的并行问题，因为在任务之间并不存在交互

图 4.3 是电路可满足性的任务/管道图。由于任务之间不需要任何通信，所以任务之间不存在管道（有些人称这种问题为令人满意的并行问题）。但是，任何一个任务都有可能产生结果，所以从每一个任务到输出器件都有一个管道。

我们的下一步工作是考虑聚合和映射。我们有一定数量的任务。任务之间没有通信。完成每

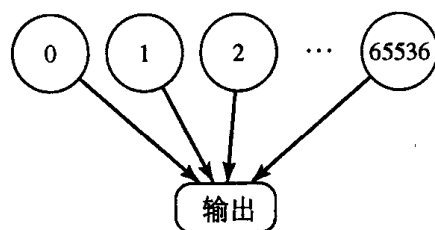


图 4.3 电路可满足性问题的任务/管道图

一个任务需要的时间并不确定。你知道为什么吗？几乎所有的任务都表示一个电路的位输入组合是不可满足的。对于一些位组合，我们可以很快发现电路是不可满足的。对于其他的就可能花较长的时间。通过使用图 3.7 中的决策树，我们知道一个好的策略是以交叉的形式将任务映射到处理器上的，其目的在于平衡计算负载。让我们看看如何使用这个策略来解决问题。

为了减少进程生成时间，我们想为每一个处理器生成一个进程。我们有  $n$  份工作需分配到  $p$  个进程中。一个环状（或交叉）分配将每隔  $p$  个任务循环分配给同一进程。例如，假设  $n=20$ ,  $p=6$ ，则进程 0 负责下标为 0、6、12 和 18 的工作；进程 1 负责下标 1、7、13 和 19；进程 2 负责下标 2、8 和 14；进程 3 负责下标 3、9 和 15；进程 4 负责下标 4、10 和 16；进程 5 负责下标 5、11 和 17。

形式化的表述是：如果有  $n$  份工作，分别标为 0, 1, ...,  $n-1$ ，它们以交叉的方式分配到下标为 0, 1, ...,  $p-1$  的  $p$  个进程中，则工作  $k$  将被分配到进程  $(k \bmod p)$  中。

在进入 C 代码前，我们总结一下程序的设计。我们要确定在图 4.2 中的电路是否是可满足的，要考虑 16 个布尔值输入的全部 65 536 种组合。这些组合要以交叉的方式分配到  $p$  个进程中。每个进程将依次检查每一种组合。如果进程发现一个使电路满足的输入，将打印这个输入组合。

现在让我们来具体分析它的 C 代码（整个程序如图 4.4 所示）。

```
/*
 * Circuit Satisfiability, Version 1
 *
 * This MPI program determines whether a circuit is
 * satisfiable, that is, whether there is a combination of
 * inputs that causes the output of the circuit to be 1.
 * The particular circuit being tested is "wired" into the
 * logic of function 'check_circuit'. All combinations of
 * inputs that satisfy the circuit are printed.
 *
 * Programmed by Michael J. Quinn
 *
 * Last modification: 3 September 2002
 */

#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i;
    int id; /* Process rank */
    int p; /* Number of processes */
    void check_circuit (int, int);

    MPI_Init (&argc, &argv);
```

```

MPI_Comm_rank (MPI_COMM_WORLD, &id);
MPI_Comm_size (MPI_COMM_WORLD, &p);

for (i = id; i < 65536; i += p)
    check_circuit (id, i);
printf ("Process %d is done\n", id);
fflush (stdout);
MPI_Finalize ();
return 0;
}

/* Return 1 if 'i'th bit of 'n' is 1; 0 otherwise */
#define EXTRACT_BIT(n,i) ((n&(1<<i)) ? 1:0)

void check_circuit (int id, int z) {
    int v[16]; /* Each element is a bit of 'z' */
    int i;

    for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);

    if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
        && (!v[3] || !v[4]) && (v[4] || !v[5])
        && (v[5] || !v[6]) && (v[5] || v[6])
        && (v[6] || !v[15]) && (v[7] || !v[8])
        && (!v[7] || !v[13]) && (v[8] || v[9])
        && (v[8] || !v[9]) && (!v[9] || !v[10])
        && (v[9] || v[11]) && (v[10] || v[11])
        && (v[12] || v[13]) && (v[13] || !v[14])
        && (v[14] || v[15])) {
        printf ("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d\n", id,
            v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],
            v[10],v[11],v[12],v[13],v[14],v[15]);
        fflush (stdout);
    }
}

```

图 4.4 用于解决电路可满足性的 MPI 程序 (第 1 版)

程序开始处是预处理指令, 用于包含 MPI 和标准 I/O 的头文件:

```

#include <mpi.h>
#include <stdio.h>

```

然后是 main 函数头。请注意, 我们包含了 argc 和 argv 参数, 它们将被传入用于初始化 MPI 的函数:

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

main 函数有 3 个标量变量。变量  $i$  是循环下标,  $id$  是进程 ID 号,  $p$  是当前进程的总数。注意, 如果有  $p$  个进程, 则 ID 号从 0 开始到  $p-1$ 。

每一个活动的 MPI 进程执行它自己的程序副本。这意味着对程序中声明的所有变量, 不论是外部变量 (在任何函数外声明的) 还是在函数内声明的自动变量, 每一个 MPI 进程有其自己的副本。

我们还引入了函数 `check_circuit` 的原型, 该函数将决定第  $i$  个输入组合是否满足电路。

#### 4.4.1 MPI\_Init 函数

被每一个 MPI 进程调用的第一个 MPI 函数都是 `MPI_Init`。该函数指示系统完成所有初始化工作, 以备对后续 MPI 库的调用进行处理。调用 `MPI_Init` 并不一定是程序中第一个执行语句。实际上, 它甚至不需要放在 `main` 函数中。惟一的要求是 `MPI_Init` 要在调用任何 MPI 函数之前调用。

注意, 所有的 MPI 标识符, 包括函数标识符, 都以 MPI 前缀开头, 后面紧跟一个大写字母和一系列小写字母以及下划线。所有的 MPI 常数都是以 MPI 开头的大写字母和下划线的字符串:

```
MPI_Init (&argc, &argv);
```

#### 4.4.2 MPI\_Comm\_rank 和 MPI\_Comm\_size 函数

当 MPI 初始化后, 每一个活动进程变成了一个叫做 `MPI_COMM_WORLD` 的通信域中的成员。一个通信域是一个不透明的对象, 提供了在进程之间传递消息的环境。`MPI_COMM_WORLD` 是一个你“免费”获得的默认通信域。对于大部分的程序而言, 这已经足够了。然而, 当你需要将进程划分到独立的通信组中时, 可以建立自己的通信域。具体做法将在第 8 章中讨论。

在一个通信域内的进程是有序的。一个进程的序号便是它在整个排序中的位置。在一个有  $p$  个进程的通信域中, 每一个进程有一个惟一的序号 (ID 号), 取值为  $0 \sim p-1$ 。可以使用进程的序号来决定它将负责计算和 (或) 数据集的哪一部分。

进程可以通过调用函数 `MPI_Comm_rank` 来确定它在通信域中的序号, 并调用 `MPI_Comm_size` 来确定一个通信域中的进程总数。

```
MPI_Comm_rank (MPI_COMM_WORLD, &id);  
MPI_Comm_size (MPI_COMM_WORLD, &p);
```

现在, MPI 进程知道它的序号以及进程总数, 可以检测在对电路的 65 536 个输入中它负责的部分:

```
for (i = id; i < 65536; i += p)
```

```
check_circuit (id, i);
```

在完成循环后, 该进程的工作也宣告结束, 它将打印一个提示完成的信息。我们在每一个 `printf` 后面都加一个 `fflush` 调用。这样, 即使并行程序崩溃, 输出缓存也将被清空并保证消息最终在标准输出上出现。

```
printf ("Process %d is done\n", id);  
fflush (stdout);
```

### 4.4.3 MPI\_Finalize 函数

在一个进程执行完其全部 MPI 库函数调用后, 将调用函数 `MPI_Finalize`, 从而让系统释放分配给 MPI 的资源 (例如内存等)。

```
MPI_Finalize();  
return 0;  
}
```

函数 `check_circuit` 收到进程的 ID 号和一个整数值  $z$ , 它首先将使用宏 `EXTRACT_BITS` 分解 16 个输入的值。元素  $v[0]$  对应输入  $a$ , 元素  $v[1]$  对应输入  $b$ , 等等。令  $z$  取值为 0~65535, 调用函数 `check_circuit` 将检测所有可能的  $2^{16}$  种组合。

在函数 `check_circuit` 确定了 16 个输入之后, 它检测其结果是否导致输出为 1。如果是, 进程将打印从  $a$  到  $p$  的值。

### 4.4.4 编译 MPI 程序

将这个程序录入到文件 `sat1.c` 后, 我们需要编译它。在不同的系统之间编译 MPI 程序的命令不同。这里是一个常见的命令行语法:

```
% mpicc -o sat1 sat1.c
```

通过这个命令, 系统将保存在 `sat1.c` 中的 MPI 程序编译并把可执行程序保存在 `sat1` 中。

### 4.4.5 运行 MPI 程序

典型的运行 MPI 程序的命令是 `mpirun`。 `-np` 标记表示要生成进程的个数。我们来查看当使用 1 个进程时的输出:

```
% mpirun -np 1 sat1  
0) 1010111110011001  
0) 0110111110011001  
0) 1110111110011001  
0) 101011111011001
```



```

0) 011011111011001
0) 111011111011001
0) 101011111011001
0) 011011111011001
0) 111011111011001

```

Process 0 is done

这个程序给出了满足电路的 9 种组合 (使之输出为 1)。例如, 输出的第一行表示当 a、c、e、f、g、h、l、m 和 p 为真 (取值为 1) 其他的变量为假 (取值为 0) 时, 电路输出得到满足。注意, 并程序在一个进程上的输出与解同一问题的串程序的输出一致, 因为单独的进程评估输入组合的顺序是和串程序一致的。

现在让我们来看一下用 2 个进程的程序执行结果:

```

% mpirun -np 2 sat1
0) 0110111110011001
0) 011011111011001
0) 011011111011001
1) 1010111110011001
1) 1110111110011001
1) 101011111011001
1) 111011111011001
1) 101011111011001
1) 111011111011001

```

Process 0 is done

Process 1 is done

2 个进程一共找到了 9 个解, 进程 0 找到了 3 个, 进程 1 找到了 6 个。

下面是使用 3 个进程的执行结果:

```

% mpirun -np 3 sat1
0) 0110111110011001
0) 111011111011001
2) 1010111110011001
1) 1110111110011001
1) 101011111011001
1) 011011111011001
0) 101011111011001
2) 011011111011001
2) 111011111011001

```

Process 1 is done

Process 2 is done

Process 0 is done

9 个解再次全部被找到。每一个进程找到了 3 个。请注意, 进程的输出顺序是随机的。

在标准输出上出现的顺序，只是部分地反映了在并行计算机内部输出事件的真实发生顺序。如果进程 A 打印两条信息到标准输出上，那么第一条将出现在第二条前面。但是，如果进程 A 先于进程 B 开始在标准输出上打印，并不能保证进程 A 的输出都出现在进程 B 之前。

如果简单认为信息出现的顺序是和 `printf` 语句执行的顺序一致，则会导致关于并行计算的错误结论，这会使得程序的调试更加困难。所以，请避免落入这个陷阱！

## 4.5 聚合通信简介

我们已经有了一个良好的开端，即已经完成并运行了第一个 MPI 程序。然而，我们可以容易地找到改进它的几种方法。比如，想知道使电路能够解决的解的个数怎么办？前面的例子只有 9 个解，所以很容易对它们计数，但如果有 99 个解呢？

在我们的下一个程序中，将给进程加入能够计算解的总数的功能。我们可以很容易用一个整型值来保存每个单独进程产生的解的个数，但是处理器之间必须协同计算以获得这些值的和。

一个组通信（Collective Communication）是一个通信操作，一组进程共同工作来分发或收集一组变量，其中所含的值的个数可以是一个或多个。在信息传递环境中，归约便是利用组通信的一种操作。

我们将修改第一个电路可满足性程序以计算电路的解的总数。新的 `main` 函数如图 4.5 所示。

```
/*
 * Circuit Satisfiability, Version 2
 *
 * This enhanced version of the program also prints the
 * total number of solutions.
 */

#include "mpi.h"
#include <stdio.h>

int main (int argc, char *argv[]) {
    int global_solutions; /* Total number of solutions */
    int i;
    int id; /* Process rank */
    int p; /* Number of processes */
    int solutions; /* Solutions found by this proc */
    int check_circuit (int, int);

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
```

```

MPI_Comm_size (MPI_COMM_WORLD, &p);
solutions = 0;
for (i = id; i < 65536; i += p)
    solutions += check_circuit (id, i);

MPI_Reduce (&solutions, &global_solutions, 1, MPI_INT, MPI_SUM, 0,
            MPI_COMM_WORLD);
printf ("Process %d is done\n", id);
fflush (stdout);
MPI_Finalize ();
if (id == 0) printf ("There are %d different solutions\n",
                    global_solutions);
return 0;
}

```

图 4.5 解决电路可满足性问题的 MPI 程序 (第 2 版)  
在这个版本的程序中, 所有进程一起来确定问题的解的个数

下面来看看对 `main` 函数所做的修改。首先, 引入了两个新的整型变量。整型值 `solutions` 跟踪了这个进程找到的解的个数。进程 0 (只有进程 0) 将使用整型变量 `global_solutions` 来保存所有 MPI 进程的找到值的总数。它将负责在程序执行结束时打印这个数字。

```

int solutions;
int global_solutions;

```

必须修改函数 `check_circuit`, 令其在找到满足电路的组合时返回 1。如果组合不满足电路它应该返回 0。这个修改是很小的, 我们后面将不再谈论。

```

int check_circuit (int, int);

```

修改了循环, 使之累加该进程找到的有效解的个数。

```

solutions = 0;
for (i = id; i < 65536; i += p)
    solutions += check_circuit (id, i);

```

图 4.5 是修改后的程序代码。

## MPI\_Reduce 函数

一个进程完成它的工作后, 就准备好了加入归约操作了。函数 `MPI_Reduce` 对在一个通信域中所有进程提交的值进行一个或多个归约操作。函数 `MPI_Reduce` 的函数声明如下:

```

int MPI_Reduce (
    void *operand, /* 第一个归约元素的地址*/
    void *result, /* 第一个归约结果的地址*/
    int count, /* 执行归约的次数*/

```

```

MPI_Datatype type, /* 元素类型*/
MPI_Op operator, /* 归约操作符*/
int root, /* 得到结果的进程*/
MPI_Comm comm) /* 通信域*/

```

我们来分别考查函数的每个参数。第3个参数 `count` 指出了要进行多少次归约。每一个进程提交 `count` 个值，每一个值都是一个用于不同归约的列表元素。

参数 `operand` 是一个输入参数。调用该函数的进程给出了执行第一个归约的元素的地址。如果 `count` 大于1，所有的归约列表元素占用一段连续的内存。

第4个参数 `type` 是一个输入参数，用于指定执行归约操作的元素类型。关于 MPI 常数和它们所代表的 C 类型如表 4.1 所示。

表 4.1 对应 C 数据类型的 MPI 常数

MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_SHORT	short
MPI_UNSIGNED	CHAR unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

第5个参数 `operator` 给出了要执行的归约的类型。所有内建的归约操作符如表 4.2 所示。

表 4.2 MPI 的内建归约操作符

MPI_BAND	按位与
MPI BOR	按位或
MPI_BXOR	按位异或
MPI LAND	逻辑与
MPI_LOR	逻辑或
MPI_LXOR	逻辑异或
MPI_MAX	最大值
MPI_MAXLOC	最大值和最大值所在位置
MPI_MIN	最小值
MPI_MINLOC	最小值和最小值所在位置
MPI_PROD	乘积
MPI_SUM	和

第6个参数 `root` 给出了将获得归约结果的进程的序号。

参数 `result` 指向第一个归约结果的地址。这个参数只对进程 `root` 有意义。

最后一个参数，`comm` 给出了通信域的名称——即参加归约的进程集合。

一个 MPI\_Reduce 调用的具体形式如下：

```
MPI_Reduce (&solutions, &global_solutions, 1, MPI_INT, MPI_SUM, 0,
            MPI_COMM_WORLD);
```

在这个函数返回后, 进程 0 使其 `global_solutions` 变量中包含所有进程的 `solutions` 变量的和。它在调用 `MPI_Finalize` 后打印该全局和, 因此将出现在程序输出的最后。

非常重要的一个问题是, 尽管只有一个进程 (在这个例子中是进程 0) 得到这个全局结果, 每一个进程都必须调用 `MPI_Reduce`。这并不奇怪! 在通信域中的每一个进程必须主动地加入归约过程——它们不能被进程 0 “召唤”。如果你编写的程序中, 只要有一个处于同一个通信域的进程没有调用 `MPI_Reduce` 或者其他的组通信函数, 这个程序就将在这个函数执行处被“挂起”而不能完成。

现在再回到 `MPI_Reduce`。注意, 我们通过条件限制了 `printf` 函数的执行, 因此只有进程 0 打印 `global_solutions` 的值。我们这样做出于两个原因: 首先, 只有进程 0 在其 `global_solutions` 变量中含有真实的全局和, 其他进程中的这个变量都是未定义的; 其次, 即使每一个进程都有 `global_solutions` 的正确副本, 你想读多少次结果呢? 一个进程打印结果就足够了。具体代码如下:

```
if (id==0) printf ("There are %d different solutions\n", global_solutions);
```

下面是使用 3 个进程的程序执行结果:

```
% mpirun -np 3 sat2
```

```
0) 0110111110011001
```

```
0) 1110111111011001
```

```
1) 1110111110011001
```

```
1) 1010111111011001
```

```
2) 1010111110011001
```

```
2) 0110111111011001
```

```
2) 1110111110111001
```

```
1) 0110111110111001
```

```
0) 1010111110111001
```

```
Process 1 is done
```

```
Process 2 is done
```

```
Process 0 is done
```

```
There are 9 different solutions
```

将这个结果和第一个程序在 3 个进程上运行的结果相比较, 尽管每一个进程找到结果的顺序和以前的相同, 但是进程之间打印结果的顺序却是不同的。

## 4.6 检测并行性能

至此, 我们完成并运行了一个并程序, 一个很自然的问题就是, 我们能否从并行中得到好处。也就是说, 我们是否能更快地得到结果?

### 4.6.1 MPI\_Wtime 和 MPI\_Wtick 函数

查看并行程序的执行时间是检测并行程序性能的一种方法，即计算从我们开始执行到程序结束所经历的秒数。在计算机产业界，这是最有效的评测标准。

然而，从我们的角度出发，我们更愿意关注更小范围的问题。一般地，我们会忽略初始化 MPI 进程、建立通信套接字以及在串行设备上执行 I/O 操作所消耗的时间。我们要计算并行程序在读取数据集和输出结果过程之间的性能，并与它所对应的串行程序进行比较。

MPI 提供了一个叫做 MPI\_Wtime 的函数，它返回从过去的某点时间到当前时间所消逝的时间秒数。函数 MPI\_Wtick 则返回 MPI\_Wtime 返回结果的精度。下面分别是这两个函数的声明：

```
double MPI_Wtime (void)
double MPI_Wtick (void)
```

为了测试某一段代码，可以在它的前后加上一对 MPI\_Wtime 的调用。两次调用的返回值的差就是消逝的时间秒数。

逻辑上来看，每一个 MPI 进程从同一时间开始，但实际上并非如此。MPI 进程在不同的处理器上可能相隔数秒才开始执行。这使得计时结果相差很大。例如，在第二个计算电路可满足性问题的程序中，进程调用 MPI\_Reduce 来获得解的总数。由于所有的进程必须加入这个通信函数，因此在所有的进程都达到这点之前没有进程可以结束。较早开始的进程可能要等一段时间，直至落后的进程赶上来。那么相对于落后的进程，这些进程将报告出长得多的计算时间。

### 4.6.2 MPI\_Barrier 函数

通过在第一次调用 MPI\_Wtime 之前引入阻塞同步，可以解决上面的问题。在一个阻塞处，任何进程都不能继续执行，直至所有的进程都到执行到该处。因此，阻塞保证了所有的进程基本上在同一个时间进入到待测试的代码部分。

下面是同步函数的原型：

```
int MPI_Barrier (MPI_Comm comm)
```

MPI\_Barrier 中惟一的参数指明了哪个通信域中的进程将参与同步。

可以通过在 main 函数中添加一个局部变量来测试电路可满足性程序：

```
double elapsed_time;
```

在初始化 MPI 后启动计时器：

```
MPI_Init (&argc, &argv);
MPI_Barrier (MPI_COMM_WORLD);
elapsed_time = - MPI_Wtime();
```

在调用 `MPI_Reduce` 之后停止计时器:

```
MPI_Reduce (&solutions, &global_solutions, 1, MPI_INT, MPI_SUM, 0,
            MPI_COMM_WORLD);
elapsed_time += MPI_Wtime();
```

由于不希望计算 I/O 时间, 因此需要注释掉在 `check_circuit` 内部对 `printf` 和 `fflush` 的调用。

现在我们做好测试的准备了。测试结果如图 4.6 中的实线所示。随着处理器的不断加入, 执行时间不断减少, 这是因为每一个处理器所要测试的电路变少了。图中的虚线表示理想化的执行时间, 即, 如果两个处理器执行该程序则时间减半, 如果三个处理器执行程序则时间变为  $1/3$ , 依此类推。实际执行时间比这个时间更长的原因是, 有一些时间消耗在程序结束处的归约操作。这个通信时间在串行程序中是没有的。随着处理器个数的增加, 这个额外开销也会加大。

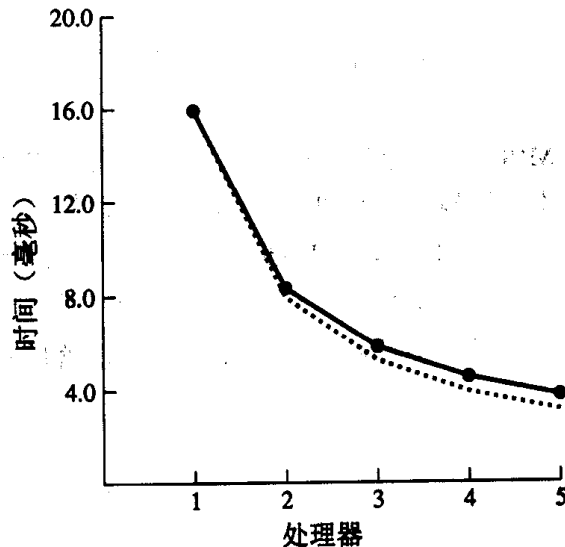


图 4.6 第二个电路可满足性程序在一个商用的集群上运行的平均执行时间, 这个集群采用了 450 MHz Pentium II 处理器。虚线代表一个“理想化”的加速比, 即  $p$  个处理器的执行速度是一个处理器的  $p$  倍

## 4.7 本章小结

使用任务/管道模型进行的并行算法设计, 可以很自然地通过消息传递编程模式来实现。消息传递模式允许程序员控制本地的内存使用, 并增加数据的局部性。正是由于无论是多计算机系统还是多处理器的性能优化, 保持数据引用的本地化特征都是一个重要的策略, 所以消息传递程序得以在较广范围的并行系统中高效运行。

在商用多计算机系统发展初期, 每一个制造商都有自己的消息传递库, 但是程序员需要一个消息传递的标准以增加程序的可移植性。消息传递接口 (MPI) 标准是企业 and 研究人员合作努力的结果。今天, 几乎每一种商用计算机都支持 MPI 函数。同样, 对构建商用集群者而言, MPI 库也是可以免费获得的。

本章开发并测设了一个并程序，可以解决电路可满足性问题。在这一过程中，我们使用了 MPI 库函数的一小部分。在后面的章节中，我们将引入很多其他的 MPI 函数。

## 4.8 主要术语

collective communication	组通信
communicator	通信域
cyclic (or interleaved)	环状 (或交叉) 分配
embarrassingly parallel	令人满意的并行
rank	序号

## 4.9 参考文献

自从计算机时代来临时，计算机科学家就一直在关注并行编程。实际上，在英国计算机协会的《计算机杂志》的创刊号中，其导言的题目就是“并行计算”【40】。这份期刊出版于 1958 年 4 月。

现今介绍使用 MPI 编程的书包括 Pacheco 的“*Parallel Programming with MPI*”【89】和 Gropp et al. 的“*Using MPI: Portable Parallel Programming with the Message-Passing Interface*”【45】。另外，Foster 在他的并行算法设计一书【31】中有一章是关于 MPI 的。

在“*Practical Parallel Programming*”一书的第 5 章中，Wilson 总结了三个不同的消息传递模型的特点，并特别指出了它们的缺点【116】。在第一个模型中，匿名的进程通过管道相连接。编程语言 Occam 就是基于这个模型的。在第二个模型中，进程被组织在一个规则的拓扑结构中，并且只能和它们的邻居或者控制进程通信。在第三个模型中，进程有自己的名字，每一个进程都可以和其他的进程通信。MPI 就属于这一类型。

## 4.10 练习题

4.1  $n$  份工作以交叉的方式被分配到  $p$  个进程中。

(a) 哪些工作被分配到进程  $p$ ，其中，

$$0 \leq k \leq p-1?$$

(b) 哪个进程负责工作  $j$ ，其中，

$$0 \leq j \leq n-1?$$

(c) 进程至多会分配到多少份工作？

(d) 给出含有最多工作份数的所有进程。

(e) 进程最少会分配到多少份工作？

(f) 给出含有最少工作份数的所有进程。



4.2 有 5 个无符号 8 位整数, 十进制下分别为 13、22、43、64、99, 给出下列归约操作的十进制计算结果:

- (a) 加
- (b) 乘
- (c) 最大值
- (d) 最小值
- (e) 按位或
- (f) 按位与
- (g) 逻辑或
- (h) 逻辑与

假设“与”和“或”操作与 C 语言中定义的一致。

4.3 修改函数 `check_circuit`, 使之在输入参数代表一个可满足电路时返回整型值 1, 如果输入参数不代表一个可满足电路时返回整型值 0。

4.4 (a) 分别用 1、2、...、8 个处理器测试第二个电路可满足性程序 (禁用打印语句)。对每一个处理器个数, 运行 5 次得到平均执行时间。

(b) 总结并解释你所观察到的结果。

4.5 本章中的电路可满足性程序在其 `check_circuit` 函数中以“硬连线”的方式写入了待测试的电路。说明如何修改电路可满足性程序从而检测一个从数据文件读入的电路。

(a) 如何在一个纯文本的文件中用“与”、“或”、“非”门表示一个电路, 使之可以用文本编辑器构造和查看?

(b) 你的程序如何解析这个文件?

(c) 描述你表示电路时所用到的数据结构。

4.6 编写 Kernighan 和 Ritchie 版的“hello, world”程序的并行版本。每一个进程都打印一个如下形式的消息:

```
hello, world, from process <i>
```

其中 `<i>` 是进程序号。

4.7 编写一个并行程序, 以如下的方式计算  $1+2+\dots+p$  之和: 每一个进程  $i$  给一个整数赋值  $i+1$ , 然后全部进程执行这些值的归约操作。进程 0 打印归约的结果。作为验证结果正确性的方法, 进程 0 还应该计算并打印  $p(p+1)/2$  的值。

4.8 一个素数是一个只能被正整数 1 和它本身整除的正整数。最小的 5 个素数是 2、3、5、7、11。有时两个连续的奇数都是素数。例如, 在 3、5、11 后面的奇整数都是素数。但是 7 后面的奇整数不是素数。编写一个并行程序判断, 对所有小于 1 000 000 的整数, 计算连续奇整数都是素数的情况的出现次数。

4.9 在两个连续的素数 2 和 3 之间的间隔是 1, 而在连续素数 7 和 11 之间的空间是 4。编写一个并行程序, 对所有小于 1 000 000 的整数, 求两连续素数之间间隔的最大值。

4.10 一个小规模的学校想给每一个现在的和将来的学生一个惟一的证件号。管理部门想使用 6 位数字, 但不确定是否够用, 已知一个“可接受的”证件号是有一些限制的。编写一个并行程序来计算不同的 6 位数的个数 (由 0~9 组合的数), 要求满足以下限制:

- 第一位数字不能为 0;
- 两个连续位上的数字不能相同;
- 各位数字之和不能是 7、11、13。

#### 4.11 定积分:

$$\int_0^1 \frac{4}{1+x^2} dx$$

的值是 $\pi$ 。我们可以使用数值积分的方法,通过计算在曲线下的图形的面积来计算 $\pi$ 。一个简单的方法是使用矩形规则(如图 4.7 所示)。将区间 $[0, 1]$ 分成  $k$  个等长子区间。用这些高度可以构造  $k$  个矩形。这些矩形的面积逼近曲线下图形的面积。当  $k$  增加时,估算精度随之增加。

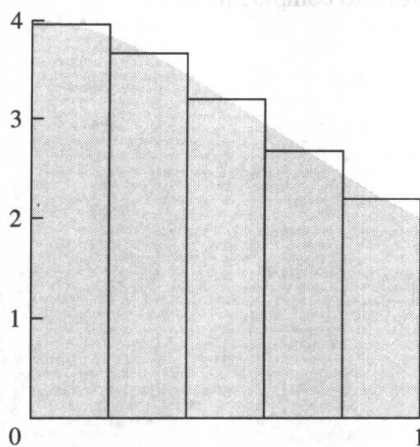


图 4.7 矩形规则是一个简单的估算曲线下图形的面积的方法  
在这个例子中,函数是  $4/(1+x^2)$ , 在 0 和 1 之间的曲线下的面积是 $\pi$

一个用矩形规则估算 $\pi$ 的 C 程序如图 4.8 所示。

```
/* This program computes pi using the rectangle rule. */
```

```
#define INTERVALS 1000000
```

```
int main (int argc, char *argv[])
```

```
{
```

```
    double area; /* Area under curve */
```

```
    double ysum; /* Sum of rectangle heights */
```

```
    double xi; /* Midpoint of interval */
```

```
    int i;
```

```
    ysum = 0.0;
```

```
    for (i = 0; i < INTERVALS; i++) {
```

```
        xi = (1.0/INTERVALS)*(i+0.5);
```

```
        ysum += 4.0/(1.0+xi*xi);
```

```
    }
```

```
    area = ysum * (1.0 / INTERVALS);
```

```
    printf ("Area is %13.11fn", area);
```

```
    return 0;
```

```
}
```

图 4.8 一个使用矩形规则计算 $\pi$ 值的 C 程序

(a) 编写一个使用矩形规则用 1 000 000 个区间的并行程序计算  $\pi$ 。

(b) 在不同个数的处理器上测试你的程序。

4.12 辛普生规则是一个比矩形规则更好的数值积分算法, 因为它能更快地收敛。假设我们要计算  $\int_a^b f(x)dx$ 。将区间  $[a,b]$  划分成  $n$  个子区间, 其中  $n$  是偶数。令  $x_i$  表示第  $i$  个区间的末尾, 其中  $1 \leq i \leq n$ , 令  $x_0$  表示第一个区间的开始。根据辛普生规则:

$$\int_a^b f(x)dx \approx \frac{1}{3n} \left[ f(x_0) - f(x_n) + \sum_{i=1}^{n/2} (4f(x_{2i-1}) + 2f(x_{2i})) \right]$$

一个用辛普生规则估算  $\pi$  的 C 程序如图 4.9 所示。

```
/* This program uses Simpson's Rule to compute pi. */
#define n 50
double f(int i) {
    double x;
    x = (double) i / (double) n;
    return 4.0 / (1.0 + x * x);
}
int main (int argc, char *argv[]) {
    double area;
    int i;
    area = f(0) - f(n);
    for (i = 1; i <= n/2; i++)
        area += 4.0*f(2*i-1) + 2*f(2*i);
    area /= (3.0 * n);
    printf ("Approximation of pi: %13.11f\n", area);
    return 0;
}
```

图 4.9 一个使用辛普生规则计算  $\pi$  值的 C 程序

(a) 使用辛普生规则, 编写一个并行程序计算  $\pi$  的值:  $f(x)=4/(1+x^2)$ ,  $a=0$ ,  $b=1$ ,  $n=50$ 。

(b) 在不同个数的处理器上测试你的程序。

# 第 5 章 Eratosthenes 筛法

He was not merely a chip of the old block, but the old block itself.

Edmund Burke

## 5.1 概 述

我们用 Eratosthenes 筛法作为例子来继续讨论 MPI 并行程序设计。在介绍串行算法后，我们使用领域分解方法得到一个数据并行的算法。在聚集阶段将考察几种数据划分方案。并行算法最后还需要一个广播的步骤，我们将学习如何使用 MPI 函数进行广播。

编码和对并行程序进行初步测试后，我们将学习提高性能的 3 种方法，包括使用冗余计算来减少进程通信时间以及改变计算的顺序来提高 Cache 命中率。对这些改进的测试结果表明，即使可以使用多处理器，尽量提高单处理器的性能仍然是非常重要的。

本章介绍了下面的 MPI 函数：

- MPI\_Bcast 在一个通信域中向所有进程发送一个消息。

## 5.2 串 行 算 法

我们的目标是开发一个并行的素数筛法程序。该算法的串行版本是由古希腊数学家 Eratosthenes（公元前 276~194）提出的。图 5.1 中是 Eratosthenes 筛法的伪代码。

1. 创建一个自然数 2, 3, 4, ...,  $n$  的列表，其中所有的自然数都没有被标记
2. 令  $k=2$ ，它是列表中第一个未被标记的数
3. 重复下面的步骤直到  $k^2 > n$  为止
  - (a) 被  $k^2$  和  $n$  之间的是  $k$  倍数的数都标记出来
  - (b) 找出比  $k$  大的未被标记的数中最小的那个，令  $k$  等于这个数
4. 列表中未被标记的数就是素数

图 5.1 Eratosthenes 筛法查找 2~ $n$  之间的素数

图 5.2 展示了一个该筛法的例子。为了找到 60 以内的素数，2、3、5 和 7 的倍数被作为合数标出。下一个质数是 11，它的平方是 121，大于 60，从而结束了筛法的循环。未被标记的数就是素数。

Eratosthenes 筛法在识别具有数百位数字的大素数时并不实用，因为其算法复杂度为  $O(n \ln \ln n)$ ，并且  $n$  是数字位数的指数。但是，改进了的筛法仍然是数论研究中的重要

工具。

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60

(a)

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60

(b)

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60

(c)

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60

(d)

图 5.2 Eratosthenes 筛法。本示例的目的是找出所有小于或等于 60 的素数：(a) 标出 4~60 之间所有是 2 的倍数的数；(b) 此时，下一个未标记的数为 3，标出 9~60 之间所有是 3 的倍数的数；(c) 此时，下一个未标记的数为 5，标出 25~60 之间所有是 5 的倍数的数；(d) 此时，下一个未标记的数为 7，标出 49~60 之间所有是 7 的倍数的数；由于 11 的平方为 121，而 121 大于 60，因此算法终止。所有未被标记的数就是素数

我们用 C 语言实现该算法的时候，可以使用包含  $n-1$  个字符的数组（具有下标 0, 1, ...,  $n-2$ ）来表示自然数 2, 3, ...,  $n$ 。数组第  $i$  个元素的布尔值表示自然数  $i+2$  是否被标记。

## 5.3 并行性的来源

应该如何划分这个算法呢？该算法的核心是在表示整数的数组中进行标记，因此可以进行域分解，将数分解成  $n-1$  个元素，并在每个元素上执行同样的任务。

关键的并行计算是步骤 3a，在该步骤中，某个特定的素数  $k$  的倍数将被标记为合数。对于代表整数  $j$  的单元，计算是非常直接的，如果  $j \bmod k = 0$ ，那么  $j$  是  $k$  的倍数，应该被标记。

如果原始任务代表整数，那么步骤 3b 中 repeat...until 循环中的每次迭代需要 2 次通信。每次循环迭代需要一次归约，以确定下一个  $k$  的值，然后还需要将  $k$  值广播出去使得所有任务都能够获得  $k$  的新取值。

我们设计中的下一个步骤将考虑如何将原始任务聚合成更加实际的任务，同时还能够保持足够的并行性。在最好的情况下，我们可以获得一个新的并行算法，其执行的计算操作和通信操作都比原有并行算法要少。

## 5.4 数据分解方法

在我们聚合了原始任务后，一个任务将负责一组数组元素。我们通常把数据元素的分组（作为划分、聚集和映射的结果）称作数据分解。

### 5.4.1 交叉数据分解

首先，让我们考虑对数组元素进行交叉数据分解：

■ 进程 0 负责自然数  $2, 2+p, 2+2p, \dots$

■ 进程 1 负责自然数  $3, 3+p, 3+3p, \dots$

.....

交叉分解的一个好处是，给定数组下标  $i$ ，很容易确定哪个进程负责该下标（进程  $i \bmod p$ ）。但对这个问题来说，交叉分解会在进程间引起显著的负载不均衡。例如，如果 2 个进程对 2 的倍数进行标记，那么进程 0 标记了  $\lfloor (n-1)/2 \rfloor$  个元素，但进程 1 什么也没有做。另一个不足之处是在实现步骤 3b 的时候，仍然需要进行一些归约和广播操作。

### 5.4.2 按块数据分解

另一个选择是按块数据分解。这意味着我们将数组分为  $p$  个连续的块，每个块的大小基本相等。如果数组元素  $n$  是  $p$  的倍数，那么分解方案是显而易见的。

如果  $n$  不是  $p$  的倍数，情况就要复杂一些。假设  $n=1024$ ，并且  $p=10$ 。在这种情况下  $1024/10=102.4$ 。如果我们给每个进程 102 个元素，最后会剩下 4 个元素。另一方面，每个进程也不可能拥有 103 个元素，因为数组并没有那么大。我们也不能简单地给前  $p-1$  个进程分配  $\lceil n/p \rceil$  个元素，然后将剩下的分配给最后一个进程，因为可能最后什么元素也没剩下（见习题 5.2）。不给某个进程分配数据会引起两个问题：第一，这将使得进程交换数据的程序逻辑变得复杂；第二，使得对通信网络的利用率降低。

我们所需要是一种能够平衡负载的按块分解的方法，给每个进程分配  $\lceil n/p \rceil$  或  $\lfloor n/p \rfloor$  个元素（如果  $n$  可以被  $p$  整除，每个进程可以划分  $n/p$  个元素）。下面我们考虑 2 种不同的实现方法。

第一种方法首先计算  $r = n \bmod p$ 。如果  $r=0$ ，那么  $n$  是  $p$  的倍数，每个进程应该分配  $n/p$  个元素。如果  $r>0$ ，那么前  $r$  个进程每个分配  $\lceil n/p \rceil$  个元素，后  $p-r$  个进程分配  $\lfloor n/p \rfloor$  个元素。

例如，当  $n=1024$ ，并且  $p=10$  时，前 4 个进程将分配到 103 个元素，后 6 个进程分配到 102 个元素。

在使用按块分解方法开发并行算法的时候通常需要回答两个问题：给定一个进程，其

控制的元素范围是什么？给定一个元素，谁是控制它的进程？

我们对第一种方案回答这两个问题。

假设  $n$  是元素数， $p$  是进程数。进程  $i$  控制的第一个元素是：

$$i \lfloor n/p \rfloor + \min(i, r)$$

进程  $i$  控制的最后一个元素是进程  $i+1$  控制的第一个元素的前一个元素：

$$(i+1) \lfloor n/p \rfloor + \min(i+1, r) - 1$$

对于特定的元素  $j$ ，控制它的进程是：

$$\min(\lfloor j / (\lfloor n/p \rfloor + 1) \rfloor, \lfloor (j-r) / \lfloor n/p \rfloor \rfloor)$$

这些表达式看起来都比较复杂。计算一个进程所控制的第一个和最后一个元素并不繁重，因为每个进程可以在算法开始时计算好并保存起来。但是，从某个元素的下标计算其控制进程却很可能是动态的，所以其复杂的表达式是令人不安的。

第二种按块分解的方案并不将所有较大的块划分给前面的进程，假设  $n$  是元素个数， $p$  是进程数。进程  $i$  控制的第一个元素是：

$$\lfloor i n/p \rfloor$$

进程  $i$  控制的最后一个元素是进程  $i+1$  所控制的第一个元素的前一个元素：

$$\lfloor (i+1) n/p \rfloor - 1$$

对于特定的元素  $j$ ，控制它的进程是：

$$\lfloor (p(j+1)-1)/n \rfloor$$

图 5.3 对比了上述两种按块分解的方案。

第二种方案在进行最常用的三个方案时所需的计算操作较少，尤其因为在 C 语言中的整数除法会自动向下取整，要优于第一种方案。在本章的剩余部分，我们将使用这一分解方案。

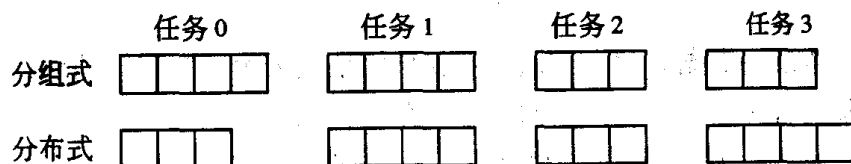


图 5.3 两种数据按块分解方案的示例。在本例中 14 个元素被分配到 4 个任务上。在第一个方案中较大的块分配给前面的任务；在第二种方案中，较大的数据块较均匀地分配给各个任务

### 5.4.3 用于按块分解的宏

我们定义 3 个 C 的宏，以便将来我们在编写并程序的时候使用按块分解来分布数据。

```
#define BLOCK_LOW(id,p,n) ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n)- 1)
#define BLOCK_SIZE(id,p,n) (BLOCK_LOW((id)+1)-BLOCK_LOW(id))
#define BLOCK_OWNER(index,p,n) (((p)*((index)+1)-1)/(n))
```

给定进程号  $id$ ，进程数  $p$ ，以及元素数  $n$ ，宏 `BLOCK_LOW` 定义了计算该进程控制的第一个元素下标的表达式。

给定同样的参数，宏 BLOCK\_HIGH 定义了计算该进程控制的最后一个元素下标的表达式。

给定数组下标，进程个数以及数组中元素的总数，宏 BLOCK\_OWNER 计算出控制该数组元素的进程号。

#### 5.4.4 局部下标还是全局下标

当我们把数组分解到各个任务中后，我们必须分清数组元素的局部下标和全局下标。

如图 5.4 所示，一个数组的 11 个元素被划分到 3 个任务中。每个任务控制 3 或 4 个元素，因此局部下标的范围是 0~2 或 0~3。但是每个局部数组也同时是更大的全局数组的一部分，其下标范围是 0~10。

当我们将串行程序并行化的时候，必须时刻牢记局部下标与全局下标之间的区别。串行程序总是使用全局下标来引用数组元素，但在并行代码中，我们必须将其替换为局部下标。

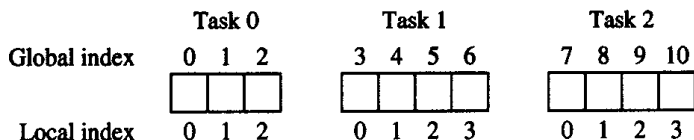


图 5.4 当数组分配到任务中后，必须区分数组元素的局部下标和全局下标。  
本图中一个有 11 个元素的数组被按块分解到 3 个任务中

#### 5.4.5 块分解的结果

按块分解方案是如何影响并行算法的实现的呢？

首先，注意到用筛法求  $n$  以内的素数时所用到的最大素数是  $\sqrt{n}$ 。如果第一个进程所控制的整数超过  $\sqrt{n}$ ，那么寻找  $k$  的下一个值就不需要任何通信了，这就节省了一步归约操作。但这个假设是否合理呢？第一个进程大约有  $n/p$  个元素，如果  $n/p > \sqrt{n}$ ，那么其控制的素数将超过  $\sqrt{n}$ 。 $n$  的值一般来说很大，所以这个假设是合理的。

按块分解的第 2 个好处是可以加快对  $k$  的倍数进行标记的过程。此时算法不必检查每个整数是否是  $k$  的倍数（这需要对每个素数进行  $n/p$  次取模操作），而是找到第一个  $k$  的倍数（记做  $j$ ）并标记它，然后标记  $j+k, j+2k$  等等，直到块的结束，总共进行  $(n/p)/k$  次赋值操作。换句话说，并行算法可以采用与串行算法类似的循环，这可以大大提高速度。

我们从这个例子中可以看到，通过按块分解，我们获得了既减少计算开销，也减少通信开销的结果。

### 5.5 开发并行算法

在决定了数据分解方案后，我们来看图 5.1 中的串行算法，看看如何将串行算法的每一步骤转换成相应的并行算法步骤。



第一步非常容易转换。在并行程序中每个进程可以创建自己所控制的自然数列表, 而不像串行程序中单个进程创建所有的自然数列表。这样每个进程的自然数列表仅包括  $\lceil n/p \rceil$  或  $\lfloor n/p \rfloor$  个布尔值。

随后, 每个进程需要知道  $k$  的值以在其控制的数组区域中标记  $k$  的倍数。因此, 并行程序中的每个进程都执行步骤 2。这是并行程序进行冗余执行的一个例子, 幸运的是, 冗余执行的工作量非常小。

步骤 3a 也很容易转换, 每个进程都负责标记在其控制区域内  $k^2 \sim n$  之间的  $k$  的倍数。在决定区域中的第一个  $k$  的倍数的位置的时候, 我们需要进行一定的计算, 但是随后的操作就是每隔  $k$  个元素进行一次标记了。

正如我们刚刚讨论过的, 当  $p < \sqrt{n}$  时, 仅有进程 0 需要对产生下一个  $k$  值负责。对于我们真正需要利用并行算法的场合,  $n$  的值应该足够大, 必然满足上述假设。如果进程 0 在步骤 3b 中负责找到下一个素数  $k$ , 那么, 所有其他进程必须接收到这个新的  $k$  值, 才能计算出 repeat...until 循环的结束条件表达式的值, 并决定是否进行下一步的循环迭代。

换句话说, 我们希望从进程 0 复制新的  $k$  值到其他进程的  $k$  的局部实例中。这个过程称为广播, MPI 提供了一个全局通信函数来完成这一功能。

## 函数 MPI\_Bcast

让我们看看 MPI\_Bcast 的函数声明, 其功能是从一个进程中将一个或多个具有相同类型的数据广播到同一通信域中的其他所有进程中。

```
int MPI_Bcast (
    void *buffer,           /* Addr of 1st broadcast element */
    int count,              /* # elements to broadcast */
    MPI_Datatype datatype,  /* Type of elements to broadcast */
    int root,               /* ID of process doing broadcast */
    MPI_Comm comm)          /* Communicator */
```

第 2 个参数 count, 说明了被广播的元素个数。每个调用这个函数的进程需要指定同样的 count 值。第 1 个参数 buffer, 是被广播数据的第一个元素的地址。此函数假定所有数据元素在内存中是连续的。第 3 个参数 datatype, 是一个 MPI 常数, 指出了被广播数据的类型。第 4 个参数 root, 是进行广播的进程的序号。最后, 第 5 个参数 comm 表示通信域, 即参与此次通信的进程组标识。

在我们的并行筛法算法中, 进程 0 需要广播一个整数  $k$  给其他进程, 其调用的形式如下:

```
MPI_Bcast (&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

在此函数执行后, 每个进程都将有一个最新的  $k$  值, 可用来计算 repeat...until 循环的结束条件。

在 repeat...until 循环执行后, 所有  $2 \sim n$  之间的素数都已被找到, 即在布尔数组中没有被标记的元素。其他程序可以利用找到的素数进行更有意义的工作。由于我们对并行编程

比对数论更感兴趣，我们就只进行一些简单的工作，比如计算  $2 \sim n$  之间的素数个数。

对每个进程来说，计算局部数组中的素数个数（取值为 0 的数组元素个数）是非常直接的。然后我们需要进行一次求和归约操作来将每个进程的素数个数累加起来，从而得到全局的总和。正如我们在前一章讲到的，可以使用函数 `MPI_Reduce` 来实现这个操作。

我们设计的并行算法的任务/通道图在图 5.5 中给出。

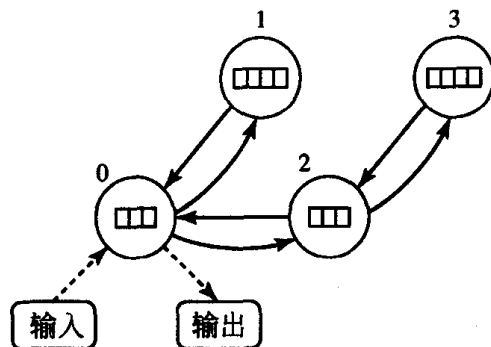


图 5.5 具有 4 个任务的并行 Eratosthenes 筛法算法的任务/通道图。虚线箭头表示使用的 I/O 通道。曲线箭头表示广播步骤的通道。直的实线箭头表示归约步骤的通道（参见图 3.12）

## 5.6 并行筛法算法的分析

在设计了并行算法后，我们来计算一下表示该算法大致执行时间的表达式。

令  $\chi$  代表标记一个为素数倍数的数组元素所需的时间。该时间不仅包括将 1 赋值给一个数组元素，还包括增量循环下标和测试循环结束条件所需的时间。串行算法的时间复杂度是  $\Theta(n \ln \ln n)$ 。我们可以通过执行串行筛法程序来确定  $\chi$  的值。换句话说，预期的串行算法的执行时间约为  $\chi n \ln \ln n$ 。

每个循环迭代仅进行一次广播操作，每次广播的开销近似为  $\lambda \lceil \log p \rceil$ ， $\lambda$  是消息的延迟时间。

循环会迭代多少次呢？在  $2 \sim n$  之间的素数个数约为  $n / \ln n$ 【11】。因此，循环迭代次数可以近似为  $\sqrt{n} / \ln \sqrt{n}$ 。

因此，并行算法的执行时间预计为：

$$\chi (n \ln \ln n) / p + (\sqrt{n} / \ln \sqrt{n}) \lambda \lceil \log p \rceil$$

## 5.7 并行政程的说明

图 5.6 是并行 Eratosthenes 筛法程序的完整代码文本。本节我们会对程序进行完整地说明。

我们从头文件引用开始。头文件 `MyMPI.h` 包含了我们开发的一些协助并行编程的工具宏和函数原型，我们将其引用到我们的程序中。程序中还定义了计算两个值的最小值的宏：

```

/*
 * Sieve of Eratosthenes
 */
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include "MyMPI.h"
#define MIN (a,b) ( (a) < (b) ? (a) : (b))
int main (int argc, char *argv[])
{
    int count; /* Local prime count */
    double elapsed_time; /* Parallel execution time */
    int first; /* Index of first multiple */
    int global_count; /* Global prime count */
    int high_value; /* Highest value on this proc */
    int i;
    int id; /* Process ID number */
    int index; /* Index of current prime */
    int low_value; /* Lowest value on this proc */
    char *marked; /* Portion of 2,...,'n' */
    int n; /* Sieving from 2, ..., 'n' */
    int p; /* Number of processes */
    int proc0_size; /* Size of proc 0's subarray */
    int prime; /* Current prime */
    int size; /* Elements in 'marked' */
    MPI_Init (&argc, &argv);

    /* Start the timer */
    MPI_Barrier (MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime ();
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    if (argc != 2) {
        if (!id) printf ("Command line: %s <m>\n", argv[0]);
        MPI_Finalize ();
        exit (1);
    }
    n = atoi (argv[1]);

    /* Figure out this process's share of the array, as
       well as the integers represented by the first and
       last array elements */
    low_value = 2 + BLOCK_LOW (id,p,n-1);
    high_value = 2 + BLOCK_HIGH (id,p,n-1);
    size = BLOCK_SIZE (id,p,n-1);

    /* Bail out if all the primes used for sieving are
       not all held by process 0 */

```

```

proc0_size = (n-1) / p;
if ((2 + proc0_size) < (int) sqrt ((double) n)) {
    if (!id) printf ("Too many processes\n");
    MPI_Finalize ();
    exit (1);
}
/* Allocate this process's share of the array. */
marked = (char *) malloc (size);
if (marked == NULL) {
    printf ("Cannot allocate enough memory\n");
    MPI_Finalize ();
    exit (1);
}
for (i = 0; i < size; i++) marked[i] = 0;
if (!id) index = 0;
prime = 2;
do {
    if (prime * prime > low_value)
        first = prime * prime - low_value;
    else {
        if (! (low_value % prime)) first = 0;
        else first = prime - (low_value % prime);
    }
    for (i = first; i < size; i += prime) marked[i] = 1;
    if (!id) {
        while (marked[++index]);
        prime = index + 2;
    }
    MPI_Bcast (&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
} while (prime * prime <= n);
count = 0;
for (i = 0; i < size; i++)
    if (!marked[i]) count++;
MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
            0, MPI_COMM_WORLD);
/* Stop the timer */
elapsed_time += MPI_Wtime ();
/* Print the results */
if (!id) {
    printf ("%d primes are less than or equal to %d\n",
            global_count, n);
    printf ("Total elapsed time: %10.6fn", elapsed_time);
}
MPI_Finalize ();
return 0;
}

```

图 5.6 Eratosthenes 筛法的 MPI 程序

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

用户需要在命令行中指定筛的上界, 如果这个参数没有指定, 程序将结束执行。在这种情况下, 每个进程都必须调用 `MPI_Finalize()`。如果命令行参数存在, 我们将该字符串转换为整数。

```
if (argc != 2){
    if (!id)printf ("Command line: %s <m>\n", argv[0]);
    MPI_Finalize();
    exit (1);
}
n = atoi(argv[1]);
```

程序将在 2 到  $n$  中找出所有的素数, 这意味着我们将检查  $n-1$  个整数的素数性。正如在前面讨论的一样, 我们将保存标记的数组划分成连续的块分给每个进程, 并使用宏来确定每个进程所控制的第一个和最后一个元素, 以及元素的个数。

```
low_value = 2 + BLOCK_LOW(id,p,n-1);
high_value = 2 + BLOCK_HIGH(id,p,n-1);
size = BLOCK_SIZE(id,p,n-1);
```

我们的算法只有在进程 0 所控制的最后一个元素下标的平方大于筛选的上界的时候才能正确运行, 因此我们加入代码检查此条件是否满足。如果条件不满足, 程序中止。

```
proc0_size = (n-1)/p;
if ((2 + proc0_size)< (int)sqrt((double)n)){
    if (!id)printf ("Too many processes\n");
    MPI_Finalize();
    exit (1);
}
```

现在我们需要在每个进程中给其负责的数组分配内存空间。因为在 C 语言中单个字节是最小的内存分配单位, 我们将数组的类型声明为 `char`。如果内存分配失败, 程序中退出。

```
marked = (char *)malloc (size);
if (marked == NULL){
    printf ("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit (1);
}
```

初始化数组, 使得所有数组元素的状态都是“未标记”。

我们已经完成了算法的步骤 1。幸运的是, 余下的步骤可以使用更少的代码来实现。比如, 步骤 2 仅需要 2 行代码。我们将从 2 的倍数开始筛选。整数 `prime` 是当前正被筛选的素数。整数 `index` 是 `prime` 在进程 0 中的数组下标。我们将 `index` 的初始化加上了条件, 目的是强调只有进程 0 需要使用这个变量。

```
if (!id) index = 0;
prime = 2;
```

现在我们到达了算法的核心部分，对应的是原算法的步骤 3。我们在 C 语言中使用 do...while 循环来实现 repeat...until 循环。

每个进程都要负责标记在其控制数组内的在 prime 的平方与  $n$  之间的所有 prime 的倍数。为了达到这个目的，我们必须确定需要被标记的第一个数组元素的下标。如果 prime 的平方比该进程所控制数组的第一个元素的值要大，我们就取这两个值的差作为第一个被标记的数组下标，否则我们就取 low\_value 除以 prime 的余数作为第一个被标记的数组下标。如果余数是 0，说明 low\_value 本身就是 prime 的倍数，是我们开始标记的地方，否则我们需要以该余数作为数组下标，找到第一个是 prime 倍数的元素。

```
if (prime * prime > low_value)
    first = prime * prime - low_value;
else {
    if (!(low_value % prime)) first = 0;
    else first = prime - (low_value % prime);
}
```

下面的 for 循环实际进行了筛选操作。每个进程都从第一个下标开始标记所有当前素数的倍数，直到数组的末尾。

```
for (i = first; i < size; i += prime) marked[i] = 1;
```

进程 0 通过在数组中寻找下一个未被标记的元素来确定下一个被筛选出的素数。

```
if (!id) {
    while (marked[++index]);
    prime = index + 2;
}
```

进程 0 将下一个素数的值广播给其他进程。

```
MPI_Bcast (&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

只要当前素数的平方小于或等于上界，这些进程继续筛选的过程：

```
} while (prime * prime <= n);
```

每个进程计算其局部数组中的素数个数：

```
count = 0;
for (i = 0; i < size; i++)
    if (!marked[i]) count++;
```

进程 0 计算素数个数总和，并将结果保存在进程 0 的变量 global\_count 中。

```
MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_
WORLD);
```

我们停止计时器。此时, `elapsed_time` 包含执行算法的时间, 不包括最初的 MPI 启动时间。

```
elapsed_time += MPI_Wtime();
```

进程 0 打印结果和执行时间。

```
if (!id){
    printf ("%d primes are less than or equal to %d\n", global_count, n);
    printf ("Total elapsed time: %10.6f\n", elapsed_time);
}
```

剩下的事情就是调用 `MPI_Finalize` 来关闭 MPI。

## 5.8 测 试

让我们看看上述并行算法在查找 1 亿以内的素数时的实际运行时间与我们的模型的对比。

并行程序的实际执行环境为使用 450MHz Pentium II 的商用集群系统, 每个 CPU 都通过快速以太网连接到 HP 的 Procurve 4108GL 交换机上。

首先, 我们通过集群系统上的一个 CPU 上执行程序的串行版本来确定  $\chi$  的值。串行程序执行了 24.9 秒, 因此:

$$\chi = 24.900s / (100000000 \ln \ln 100000000) \dots = 85.47 \text{ ns}$$

我们同时也确定  $\lambda$  的值, 通过在 2, ..., 8 个处理器上进行一系列的广播测试, 得到  $\lambda = 250\mu s$ 。

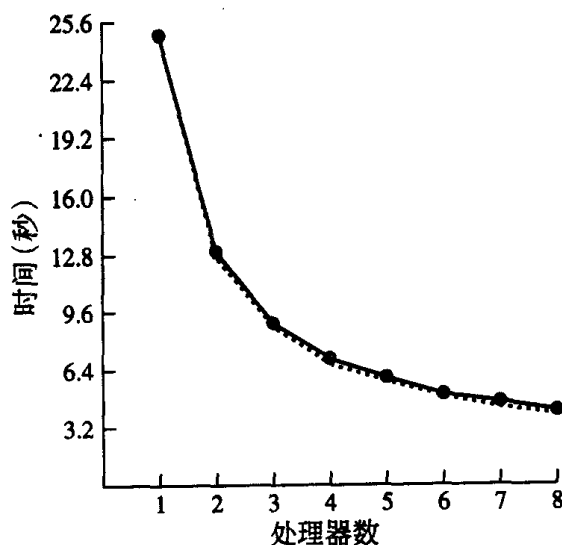


图 5.7 比较并行 Eratosthenes 筛法程序的预期执行时间 (虚线) 和实际执行时间 (实线)

将这些值带入我们先前得到的计算并行程序预期执行时间的公式, 有:

$$\chi (n \ln \ln n) / p + (\sqrt{n} / \ln \sqrt{n}) \lambda \log p = 24.900 / p + 0.2714 \lceil \log p \rceil s$$

我们的测试工作执行了并程序 40 次, 处理器的个数从 1~8, 对每个处理器数运行了 5 次程序, 并计算平均执行时间。图 5.7 比较了实验结果和我们的模型的预测结果。对 2, ..., 8 个处理器的平均误差大约为 4%。

## 5.9 改 进

尽管我们开发的并行筛法算法已经展示了良好的性能, 但我们还是可以通过对程序进行一定修改来显著地提高性能。本节我们讨论对并行筛法程序的 3 个改进。每个改进都是在前一个改进的基础上进行的。

### 5.9.1 删除偶数

既然 2 是仅有的偶素数, 把数组中一半空间用来存放偶数没有什么意义。将算法改为仅处理奇数可以将所需的空间减少一半, 并把标记特定素数倍数的速度提高一倍。经过这个改变, 串行算法的预期执行时间约为:

$$\chi(n \ln \ln n)/2$$

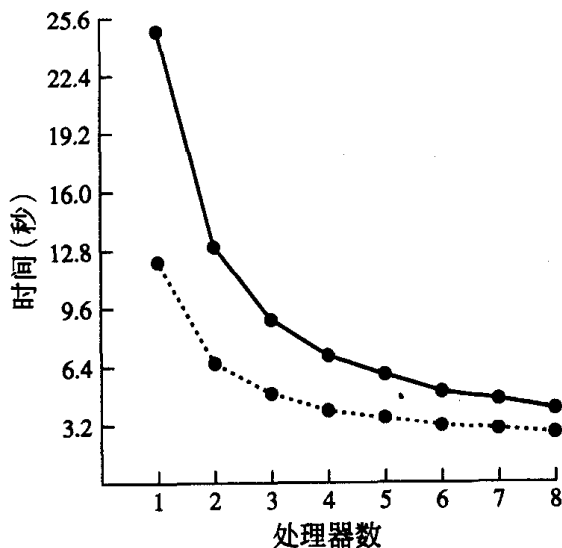


图 5.8 Eratosthenes 筛法的原有并行程序执行时间 (实线) 和改进的并行程序执行时间 (虚线)

预期的并行算法执行时间约为:

$$\chi(n \ln \ln n)/(2p) + (\sqrt{n}/\ln \sqrt{n})\lambda \log p$$

图 5.8 画出了对原有并行筛法算法和改进算法的性能测试结果, 测试内容是在 1, 2, ..., 8 个寻找 1 亿以内的素数。正如预计的那样, 至少在处理器个数较小的时候, 改进后的算法所需的时间大约是原有算法的一半。

实际上, 尽管改进后的算法在单处理器上可以比原有算法快几乎 1 倍, 但在 8 个处理器的时候仅仅是稍快一点。改进后的算法的计算量比原有算法要显著减少, 但通信开销还是一样大。当处理器个数增加后, 通信部分在整个执行时间中所占的比例会增加, 从而减



小了两个算法之间的性能差别。

### 5.9.2 消除广播

原有算法的步骤 3b 找出新的待筛素数  $k$ 。我们让一个进程计算  $k$ ，然后广播到其他进程的方法使这个步骤并行化。在程序执行的时候，这个广播操作重复了大约  $\sqrt{n}/\ln\sqrt{n}$  次。

为什么不让每个任务自己确定  $k$  的新值？在我们原有的数据分解方案中，这是不可能的，因为只有任务 0 控制着代表  $2\sim\sqrt{n}$  的整数的数组元素。但如果我们复制这些值的话，会得到什么结果？

假设每个任务除了分配到  $n/p$  个整数外，还拥有单独的数组，用于包含整数 3, 5, 7, ...,  $\lfloor\sqrt{n}\rfloor$ 。在寻找  $3\sim n$  的素数之前，每个任务可以先用串行算法计算出  $3\sim\sqrt{n}$  的素数。一旦这一步骤完成，每个任务就拥有了所有的数组，其中包含了所有  $3\sim\sqrt{n}$  的素数。现在这些任务可以在消除了广播步骤的情况下，对整个数组进行筛选。

如果下面条件成立，消除广播操作会提高并行程序的性能：

$$\begin{aligned} & (\sqrt{n}/\ln\sqrt{n})\lambda\lceil\log p\rceil > \chi\sqrt{n}\ln\ln\sqrt{n} \\ \Rightarrow & (\lambda\lceil\log p\rceil)/\ln\sqrt{n} > \chi\ln\ln\sqrt{n} \\ \Rightarrow & \lambda > \chi\ln\ln\sqrt{n}\ln\sqrt{n}/\lceil\log p\rceil \end{aligned}$$

并行程序的预计计算复杂度为：

$$\chi\left((n\ln\ln n)/(2p) + \sqrt{n}\ln\ln\sqrt{n}\right) + \lambda\lceil\log p\rceil$$

(最后一项代表进行求和归约所需的时间。)

### 5.9.3 循环的重新组织

并行筛法算法的大部分执行时间都用在对一个非常巨大的数组的分散的元素进行标记上了，造成了很差的 Cache 命中率。我们把算法的核心内容看成 2 个循环，外层循环为  $3\sim\sqrt{n}$  的素数之间迭代，内层循环在属于该进程的  $3\sim n$  的整数之间迭代。如果我们将内外层循环交换一下，就可以改进程序的 Cache 命中率。我们可以将数组的一部分放入 Cache，标记其中所有小于  $\sqrt{n}$  的素数的倍数，然后再读入数组的下一个部分，如图 5.9 所示。

### 5.9.4 测试

图 5.10 画出了原并行筛法程序和 3 个改进版本的执行时间，测试的内容仍然是在 1~8 个处理器上寻找小于 1 亿的所有素数，执行环境为使用 450MHz Pentium II 的商用集群系统，每个 CPU 都通过快速以太网连接到 HP 的 Procurve 4108GL 交换机上。

串行算法的执行时间与第一个并行算法在一个处理器上的执行时间是相同的。在 8 个处理器上，我们的并行筛法程序在实现了所有上述优化措施后比串行算法快了 72.8 倍。加速比中较大的部分 (9.8 倍) 应归功于消除了对偶数的存储和处理，以及通过重新组织循环来提高程序的 Cache 命中率，较小的部分 (7.4 倍) 归功于通过冗余计算到  $\sqrt{n}$  的素数来消

除广播以及使用 8 个处理器来进行计算。因此，我们从并行化之前对串行算法的改进中得到了更大的收获。

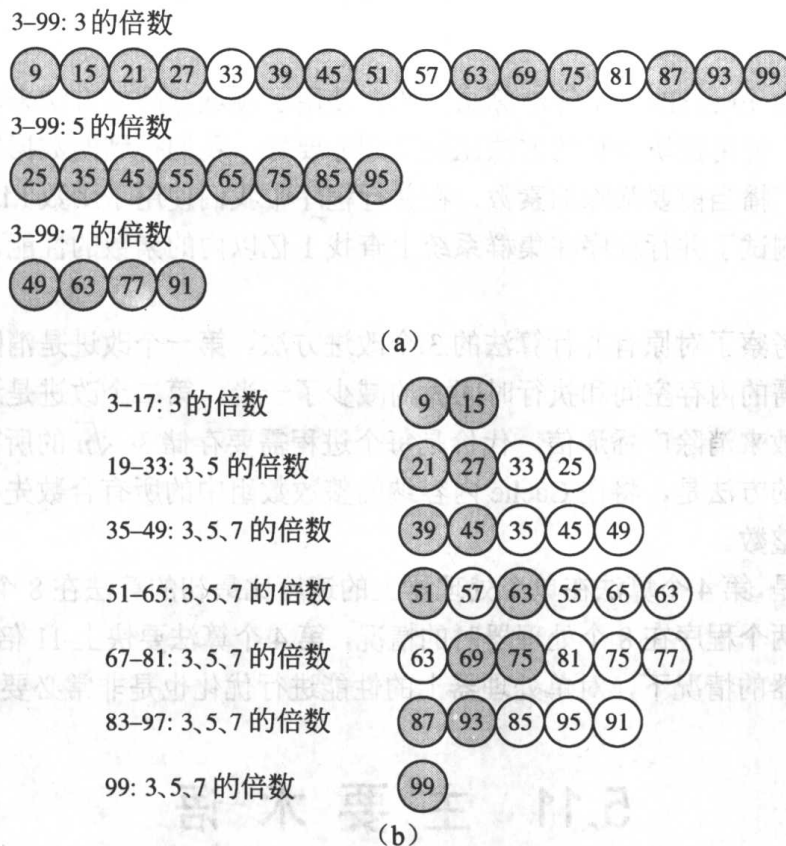


图 5.9 改变整数被标记的顺序可以极大地提高 Cache 命中率。在这个例子中，我们查找 3~99 之间的素数。假设 Cache 为 4 路的，每一路可以存放 4 个字节。一路会存放表示整数 3、5、7 和 9 的字节，下一路则存放表示整数 11、13、15 和 17 的字节等等。(a) 筛除一个素数的所有倍数，然后再筛除下一个素数。有阴影的圆圈表示 Cache 缺失，当循环的下一个迭代返回到较小的整数并开始筛除下一个素数的倍数的时候，这些整数已经不在 Cache 中了。(b) 每次在 Cache 内的 8 个整数中筛除所有素数的倍数，再处理下一组 8 个整数。有阴影的圆圈减少了，表明了 Cache 命中率的提高

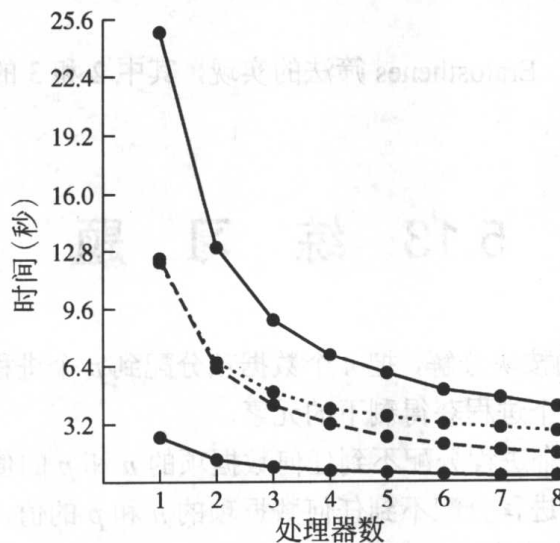


图 5.10 Eratosthenes 筛法的 4 个并行实现的执行时间，执行环境为由快速以太网连接的 450MHz Pentium II 集群。上面的实线代表最初的串行实现程序，点虚线表示消除了偶数的存储和处理后的程序，短线虚线是消除广播后的程序性能，下面的实现是进行了循环交换以提高 cache 命中率后的程序性能

## 5.10 本章小结

我们从串行的 Eratosthenes 筛法算法开始, 使用了领域分解的方法来确定其并行性。对这个算法来说, 使用按块分解的方法比交叉划分要好。我们得到的数据并行程序需要任务 0 向其他任务广播当前要筛除的素数, 在并行程序中我们使用了函数 MPI\_Bcast 来执行广播操作。我们测试了并行程序在集群系统上查找 1 亿以内的素数的性能, 得到了良好的结果。

我们进一步考察了对原有并行算法的 3 个改进方法。第一个改进是消除所有对偶数的操作, 把程序所需的内存空间和执行时间大约减少了一半。第二个改进是通过冗余计算下一个要筛除的素数来消除广播通信, 代价是每个进程需要存储  $3\sim\sqrt{n}$  的所有奇数。

第三个改进的方法是, 将在 Cache 内容纳的整数数组中的所有合数先标识出来, 然后再处理下一部分整数。

值得注意的是, 第 4 个算法在单个处理器上的速度比最初的算法在 8 个处理器上还快。如果我们比较这两个程序在 8 个处理器时的情况, 第 4 个算法要快上 11 倍。因此, 即使在使用多处理器的情况下, 对单处理器上的性能进行优化也是非常必要的。

## 5.11 主要术语

block decomposition 按块分解

data decomposition 数据分解

## 5.12 参考文献

Luo【76】提出了一个 Eratosthenes 筛法的实现, 其中 2 和 3 的倍数都不在需要被标记的整数数组中出现。

## 5.13 练习题

5.1 考察一个简单的按块分解, 把  $n$  个数据项分配到  $p$  个进程上。其中前  $p-1$  个进程获得  $\lceil n/p \rceil$  个元素, 最后一个进程获得剩下的元素。

(a) 找出使得最后一个进程分配不到任何数据项的  $n$  和  $p$  的值。

(b) 找出使得  $\lfloor p/2 \rfloor$  个进程分配不到任何数据项的  $n$  和  $p$  的值, 假设  $p>1$ 。

5.2 本章展示了两种按块进行数据分解的策略, 其中每个进程被分配到  $\lceil n/p \rceil$  或  $\lfloor n/p \rfloor$  个元素。对于下面的  $n$  和  $p$  的取值, 用表格说明这两种方案是如何分配数据的:

(a)  $n=15$  且  $p=4$

- (b)  $n = 15$  且  $p = 6$
- (c)  $n = 16$  且  $p = 5$
- (d)  $n = 18$  且  $p = 4$
- (e)  $n = 20$  且  $p = 6$
- (f)  $n = 23$  且  $p = 7$

5.3 使用第 5.6 节的分析模型预测最初的并行筛法程序在 1, 2, ..., 16 个处理器上的执行时间, 假设  $n=10^8$ ,  $\lambda=250\mu\text{s}$ , 且  $\chi=0.0855\mu\text{s}$ 。

5.4 使用第 5.9.1 节的模型预测第 2 个版本 (消除了偶数存储和标记) 的并行程序的性能, 并与表 5.1 第 2 栏得到的实际结果进行对比, 对 2, ..., 8 个处理器的预测的平均误差是多少?

**表 5.1 Eratosthenes 筛法的 4 个并行实现的平均执行时间 (单位是秒), 执行环境为由快速以太网连接的 450 MHz Pentium II 集群。筛法 1 为最初的并行实现程序, 筛法 2 是消除了偶数的存储和处理后的程序, 筛法 3 是在每个处理器上计算  $2\sim\sqrt{n}$  之间的素数以消除广播通信后的程序, 筛法 4 是进行了循环交换以提高 Cache 命中率后的程序**

处理器数	筛法 1	筛法 2	筛法 3	筛法 4
1	24.900	12.237	12.466	2.543
2	12.721	6.609	6.378	1.330
3	8.843	5.019	4.272	0.901
4	6.768	4.072	3.201	0.679
5	5.794	3.652	2.559	0.543
6	4.964	3.270	2.127	0.456
7	4.371	3.059	1.820	0.391
8	3.927	2.856	1.585	0.342

5.5 使用 5.9.2 节的分析模型预测第 3 个并行算法的性能。假设  $n=10^8$ ,  $\lambda=250\mu\text{s}$ , 且  $\chi=0.0855\mu\text{s}$ , 并与表 5.1 中筛法 3 的实际结果进行对比, 对 2, ..., 8 个处理器的预测的平均误差是多少?

5.6 修改并行 Eratosthenes 程序以实现 5.9 节所描述的第一个算法改进: 不给偶数分配内存。测试程序的性能, 并与最初的并行算法进行比较。

5.7 修改并行 Eratosthenes 程序以实现 5.9 节所描述的前两个算法改进: 不给偶数分配内存, 并通过在每个进程中计算  $3\sim\sqrt{n}$  的素数来消除对 MPI\_Bcast 的调用。测试程序的性能, 并与最初的并行算法进行比较。

5.8 修改并行 Eratosthenes 程序以实现 5.9 节所描述的所有三个算法改进。测试程序的性能, 并与最初的并行算法进行比较。

5.9 本章中开发的所有并行筛法算法都是通过域分解得到的。开发一个基于功能分解的并行筛法程序。假设有  $p$  个进程查到 2 到  $n$  之间的素数 (程序通过命令行来获得这些参数)。第一步, 每个进程独立地计算到  $\sqrt{n}$  的素数。第二步, 每个进程筛除  $2\sim\sqrt{n}$  之间的  $1/p$  份素数。第三步, 所有进程将自己的数组或归约到进程 0 上。最后一步, 进程 0 计算未被标记的数组元素个数, 并输出素数个数。

例如, 假设 3 个进程查找 1000 以内的素数。每个进程都分配一个具有 999 个元素的

数组, 表示 2~1000 的整数。每个进程单独计算  $\sqrt{1000}$  以内的素数: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31。进程 0 在自己的数组中筛除 2, 7, 17 和 29, 进程 1 筛除 3, 11, 19 和 31。进程 2 筛除 5, 13 和 23。

5.10 指出在上个练习中的并行程序与本章中所描述的最初并行算法相比的 3 个缺点。

5.11 最简单的调和数列是:

$$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \dots$$

$$\text{设 } S_n = \sum_{i=1}^n 1/i$$

(a) 写一个并行算法可以计算  $S_n$  到任意精度。例如  $S_7=2.592\,857\,142\,857$ , 即小数点后的精度为 12 位。进程 0 需要向用户查询 2 个参数  $n$  和  $d$ , 并将此参数广播给其他进程。所有进程应合作计算  $S_n$  到小数点后  $d$  位精度。在计算了  $S_n$  后, 进程 0 应该打印出该值。

(b) 使用不同个数的处理器测试此程序的性能, 计算  $S_{1\,000\,000}$  到 100 位精度。

# 第 6 章 Floyd 算法

Not once or twice in our rough island story,  
The path of duty was the path of glory.

Alfred, Lord Tennyson, Ode on the Death of the Duke of Wellington

## 6.1 概 述

旅行地图上经常会包含一些表格，说明两个城市之间的驾驶距离。代表城市  $A$  的行与代表城市  $B$  的列相交的单元内容是从城市  $A$  到  $B$  的最短路径的长度。对于较长的旅程，路径很可能经过表格中的其他城市。Floyd 算法是产生这类表格的经典算法。

在本章中我们将设计、分析、开发和测试并行的 Floyd 算法。首先，我们将开发一套从文件中读取矩阵并分配到 MPI 进程中的函数，以及从 MPI 进程中获取矩阵元素并打印它们的函数。

本章将讨论下面两个 MPI 函数：

- `MPI_Send`，允许一个进程向其他进程发送一条消息；
- `MPI_Recv`，允许一个进程接收另外一个进程发送的消息。

## 6.2 全点对最短路径问题

图是一个由  $V$ ,  $E$  组成的集合。其中  $V$  是节点的有限集， $E$  是节点间边的有限集。图 6.1 (a) 是图的一个图形方式的表示，其中节点用带有标号的圆圈表示，而边由圆圈之间的线表示。为了精确起见，图 6.1 (a) 是一个有向有权图。由于每条边都带有一个数值，

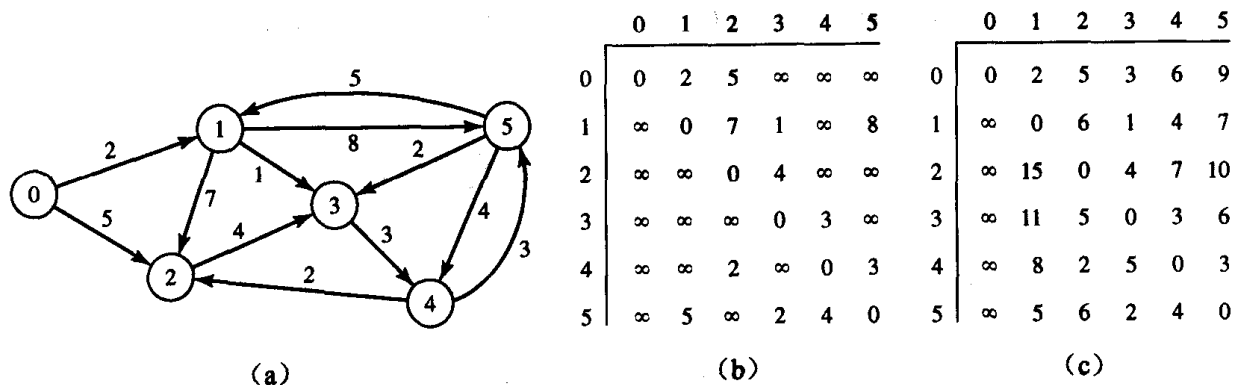


图 6.1 (a) 一个有向有权图；(b) 图的邻接矩阵表示：元素  $(i, j)$  表示从  $i$  到  $j$  的边的长度，不存在的边被认为具有无限的长度；(c) 全点对最短路径问题的解：元素  $(i, j)$  表示从节点  $i$  到节点  $j$  的最短路径长度，无穷大符号表示不存在路径

所以此图是有权图。边的权重可以有多种含义。在最短路径问题中, 权重表示距离。由于每条边都有方向 (由箭头表示), 所以此图为有向图。

给定这个有向有权图, 全点对最短路径问题就是要找到所有节点对之间的最短路径。路径的长度不是由其经过的边的条数, 而是由边的权重所决定。例如, 图 6.1 (a) 中节点 0 和节点 5 之间的最短路径长度是 9, 它经过了 4 条边 (0→1、1→3、3→4 和 4→5)。

如果我们要在计算机上解决这个问题, 就必须找到方便的方式来表示有向有权图。在这个应用中, 我们选择的数据结构为邻接矩阵, 因为它能够在常数时间内访问到任意的边, 而且其所需的存储空间也不比本问题的解所需要的更大。对于一个有  $n$  个节点的图, 邻接矩阵是一个  $n \times n$  的矩阵。在有权图中, 矩阵元素  $(i, j)$  的值是从节点  $i$  到节点  $j$  的边的权重。根据应用的不同, 不存在的边有不同的表达方式。在单源最短路径问题中, 不存在的边被赋为非常大的值 (比如在底层体系结构中所能表示的最大整数)。为方便起见, 我们使用符号  $\infty$  来表示这个非常大的值。图 6.1 (b) 就是在图 6.1 (a) 中用图形的方式表示的图的邻接矩阵表示。

当算法结束时, 矩阵包含了任意点对之间的最短路径。图 6.1 (c) 就是图 6.1 (a) 所表示的图的全点对最短路径问题的解。

40 多年前, Floyd 发明了一个解决此问题的  $\Theta(n^3)$  的算法。图 6.2 给出了 Floyd 算法。关于该算法的更多信息, 请参见 Cormen 等人的著作【18】。

#### Floyd 算法

输入:  $n$ —顶点数

$a[0..n-1, 0..n-1]$ —邻接矩阵

输出: 变换后的矩阵  $a$ , 其中包含最短路径长度

for  $k \leftarrow 0$  to  $n-1$

  for  $i \leftarrow 0$  to  $n-1$

    for  $j \leftarrow 0$  to  $n-1$

$a[i, j] \leftarrow \min(a[i, j], a[i, k] + a[k, j])$

    endfor

  endfor

endfor

图 6.2 Floyd 算法是一个  $\Theta(n^3)$  时间的算法, 用于解决全点对最短路径问题。

该算法将图的邻接矩阵表示转化为包含了任意点对之间的最短路径的矩阵

## 6.3 运行时创建数组

如果一个程序能够在运行时再指定数组的大小, 将更加实用, 因为当数组大小需要变化的时候不需要重新编译程序。

在 C 语言中分配一个一维数组是很容易的, 只需要定义一个指针标量, 并使用 malloc 语句从堆中分配内存即可。例如, 下面是分配一个一维具有  $n$  个元素的整数数组 (矩阵 A) 的方法:

```
int *A;
...
A =(int *)malloc(n * sizeof(int));
```

分配二维数组相对来说比较复杂, 因为 C 语言把二维数组看作是数组的数组。我们希望能够保证数组元素能否分配到连续的空间, 以便我们可以通过一个消息来发送或接受整个数组的内容。

图 6.3 是分配二维数组的一种方法。首先, 我们分配数组数据所需的内存, 然后分配一个指针数组, 最后将指针初始化。

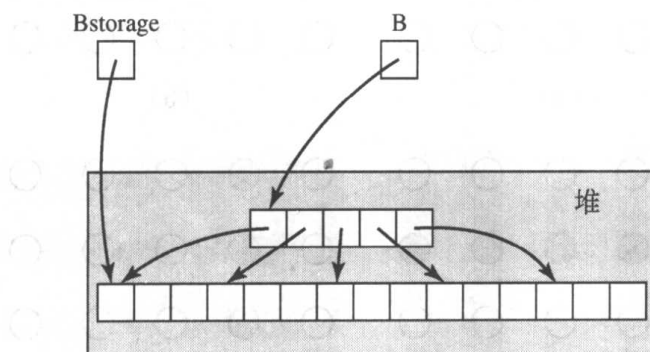


图 6.3 分配一个  $5 \times 3$  的矩阵是一个 3 步的过程: 首先, 从堆中分配 15 个矩阵数据所需的内存, 变量 `Bstorage` 指向这个内存块的开头; 其次, 从堆中分配 5 个行指针所需的内存, 变量 `B` 指向这个内存块的头; 最后, 初始化指针 `B[0], B[1], \dots, B[4]` 的值

例如, 下面的 C 代码分配 `B`, 二维的整数数组, 数组有 `m` 行和 `n` 列:

```
int **B, *Bstorage, i;
...
Bstorage =(int *)malloc(m * n * sizeof(int));
B =(int **)malloc(m * sizeof(int *));
for(i = 0; i < m; i++)
    B[i] = &Bstorage[i*n];
```

有多种方法对 `B` 的元素进行初始化。如果我们通过 `B[0][0]`、`B[0][1]` 等等来引用数组元素进行一系列初始化赋值操作, 就几乎不会犯什么错误。但是, 如果要对 `B` 的元素进行整体地初始化, 例如通过一个函数调用从数据文件中读入矩阵元素, 那么一定要记住使用 `Bstorage` 而不是 `B` 作为初始地址。

## 6.4 设计并行算法

### 6.4.1 划分

第一步是确定使用域分解还是功能分解。对本例来说, 选择是明显的。从图 6.2 的伪代码可以看出, 算法对同一个赋值语句执行了  $n^3$  次。除非我们再进行进一步分割这个语句, 否



则我们无法找出功能并行。相反地, 可以很容易地进行域分解。我们可以把矩阵  $A$  分成  $n^2$  个元素, 并对每个元素执行一个原始任务, 如图 6.4 (a) 所示。

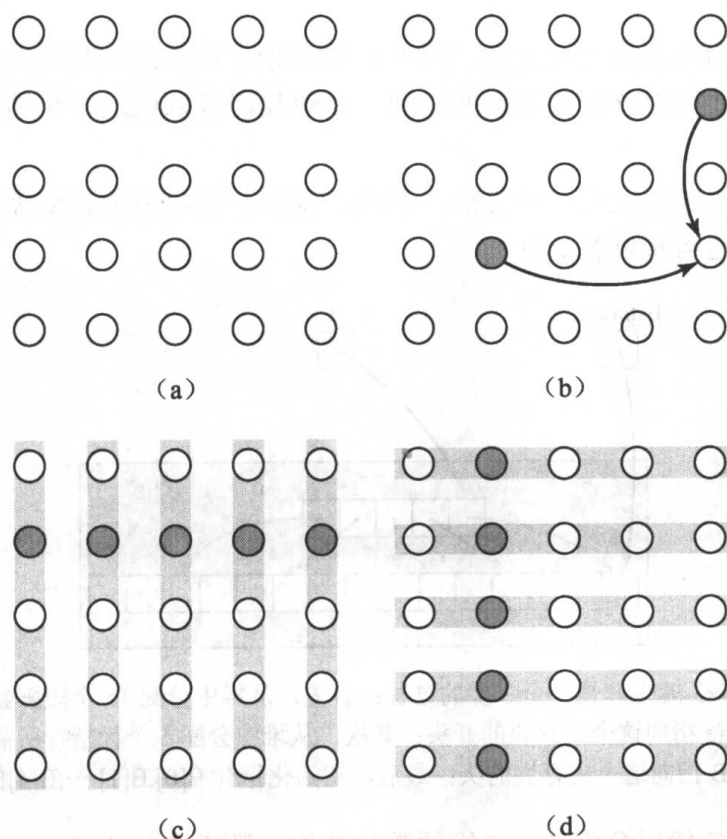


图 6.4 Floyd 算法中的分解和通信。(a) 每个元素所联系的一个原始任务; (b) 当  $k=1$  的时候更新  $a[3,4]$ ,  $a[3,4]$  的新值依赖于其原有取值和  $a[3,1]$  以及  $a[1,4]$  的取值; (c) 在第  $k$  次迭代时每个第  $k$  行的任务必须将其值广播到同一列的其他任务中; (d) 在第  $k$  次迭代时, 每个第  $k$  列的任务必须将其值广播到同一行的其他任务中, 在本图中,  $k=1$

## 6.4.2 通信

更新  $a[i,j]$  需要访问元素  $a[i,k]$  和  $a[k,j]$ 。例如, 图 6.4 (b) 展示了当  $k=1$  时更新  $a[3,4]$  所需要访问的元素。注意, 对任意给定的  $k$ , 第  $m$  列所关联的任务都需要元素  $a[k,m]$ 。类似地, 对任意给定的  $k$ , 第  $m$  行所关联的任务都需要元素  $a[m,k]$ 。这意味着在循环迭代  $k$  中, 第  $k$  行的每个元素都要被广播到同一列任务中去。同样, 第  $k$  列的每个元素也要被广播到同一行的任务中去。

$a$  中的元素能否同时被更新, 这是一个重要的问题。毕竟, 如果更新  $a[i,j]$  需要  $a[i,k]$  和  $a[k,j]$  的值, 是否我们应该先计算这些值?

答案是否定的。理由是,  $a[i,k]$  和  $a[k,j]$  的值在循环迭代  $k$  中并不改变。因为在循环迭代  $k$  中对  $a[i,k]$  的更新是按如下方式进行的:

$$a[i,k] \leftarrow \min(a[i,k], a[i,k] + a[k,j])$$

既然所有的值都是正的, 因此  $a[i, k]$  不会减少。同理, 对  $a[k, j]$  的更新是按如下方式进行的:

$$a[k, j] \leftarrow \min(a[k, j], a[k, k] + a[k, j])$$

$a[k, j]$  的值也不会减少。因此更新  $a[i, j]$  与更新  $a[i, k]$  和  $a[k, j]$  之间没有依赖关系。简单地说, 对于最外层  $k$  循环的每一个迭代来说, 我们可以进行广播然后并行更新  $a$  的所有元素。

### 6.4.3 聚合和映射

我们将使用图 3.7 的决策树来决定我们的聚合和映射策略。任务个数是静态的, 任务之间的通信模式是结构化的, 每个任务的计算时间是常数。因此, 我们应该将任务聚合起来, 以尽量减少通信开销, 每个 MPI 进程对应一个任务。

我们的目标就是将  $n^2$  个原始任务聚合成  $p$  个任务。两种自然的聚合方法是按行或按列对任务进行分组, 如图 6.5 所示。下面我们分别考察这两种聚合方法的效果。

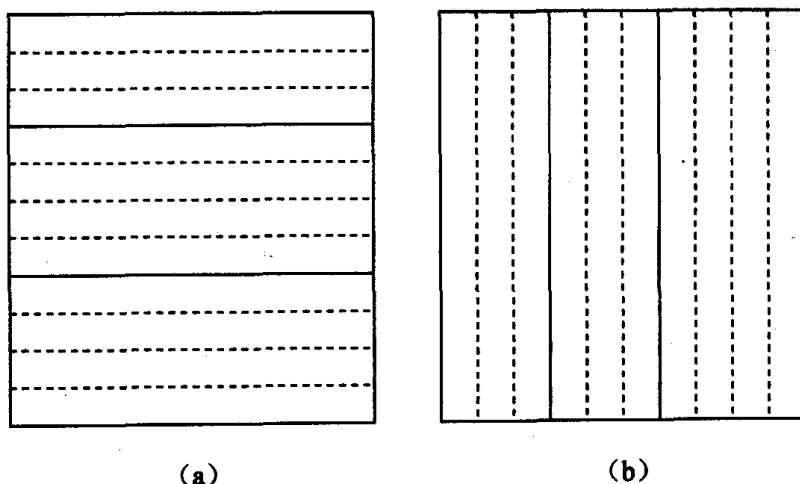


图 6.5 矩阵的两种数据分解方式。(a) 在按行块分解中, 每个进程负责一组连续的行, 本图中的 11 行被分解到了 3 个进程中; (b) 在按列块分解中, 每个进程负责一组连续的列, 本图中的 10 列被分解到了 3 个进程中

如果我们按行来聚合任务, 在原始任务中同一行内部进行的广播操作可以被消除, 如图 6.4(d) 所示, 因为这些数值都成为一个任务内的局部变量。通过聚合, 每个外层循环的迭代中需要把  $n$  个元素广播给其他任务, 每次广播需要的时间为  $\lceil \log p \rceil (\lambda + n/\beta)$ 。

如果我们按列来聚合任务, 在原始任务中同一行内部进行的广播操作可以被消除, 如图 6.4(c) 所示。通过聚合, 其每次外层循环迭代通信开销同样为  $\lceil \log p \rceil (\lambda + n/\beta)$ 。

(实际情况是, 我们没有考虑更好的聚合方法, 比如将原始任务按  $(n/\sqrt{p}) \times (n/\sqrt{p})$  的块进行分组。在第 8 章中, 当我们已经学习了更多的 MPI 函数的时候, 并将按照这种数据分解方案开发一个矩阵-向量乘法程序。)

为了在按行聚合和按列聚合之间进行选择, 我们需要考虑在算法的计算核心之外的

一些因素。并行程序必须从一个文件中输入距离矩阵。假设文件里的矩阵数据是按行排列的（文件中先是第一行的数据，然后是第二行的数据，等等）。在 C 语言中，矩阵在内存中也是按行排列的。因此，如果我们选择按行聚合的方式，将使得在进程之间分布这些矩阵行的数据的工作大为简化，输出结果矩阵也会比较容易。因此，我们选择按行分解方案。

#### 6.4.4 矩阵的输入/输出

现在我们必须决定如何支持矩阵的输入/输出。

首先，我们先看看如何从文件中读入距离矩阵。我们可以让每个进程都打开文件，找到恰当的位置，并读取属于该进程的那部分邻接矩阵数据。但是，我们还是让一个进程来负责从文件读入数据。在执行计算循环之前，该进程将读取矩阵数据并广播到其他进程中去。假定我们有  $p$  个进程，如果我们让进程  $p-1$  负责读入和分发矩阵数据，那么我们可以很容易地实现一个不需要附加的文件输入缓冲区的程序。

其原因如下：如果进程  $i$  控制第  $\lfloor in/p \rfloor$  行到第  $\lfloor (i+1)n/p \rfloor - 1$  行，那么进程  $p-1$  控制的行数为  $\lceil n/p \rceil$ （见练习题 6.1）。这意味着任何进程所控制的行数都不超过进程  $p-1$  所控制的行数。进程  $p-1$  可以用最终存放它所控制的  $\lceil n/p \rceil$  行的内存空间来作为读入其他进程所需数据时的缓冲区。

图 6.6 展示了这个方法是如何工作的。最后一个进程打开文件，读取进程 0 所需的数据，并把这些数据发送到进程 0。该进程对其他进程也重复上面的操作，最终读取自己所需的数据。

在附录 B 中给出了完整的 `read_row_striped_matrix` 函数。给定输入文件的名称，矩阵的数据类型以及通信域，它返回了：（1）指向一个指针数组的指针，允许矩阵元素通过 double-subscripting 访问；（2）指向包含实际数组元素地址的指针；（3）矩阵的大小。

我们所实现的 Floyd 算法将打印 2 次距离矩阵，一次是原始的距离，另一次是计算出最短路径。

进程 0 负责所有的打印至标准输出的操作，因此我们可以确定输出具有正确的顺序。进程 0 首先打印它自己的子矩阵，然后按顺序调用其他进程让他们发送自己的子矩阵。进程 0 将接收的每个子矩阵打印出来。

对进程  $1, 2, \dots, p-1$  来说，它们仅需要等待进程 0 的消息，然后将自己负责的子矩阵发送给进程 0 即可。

通过上述过程，我们可以保证进程 0 永远不会在同一时刻接收 1 个以上的子矩阵。进程 0 可以通过设置 `MPI_Recv` 中的进程号来区分消息的来源，那么，为什么我们不让每个进程主动将自己的子矩阵发送给进程 0 呢？原因是我们不想让进程 0 所在的处理器负载过重。每个进程都只有有限的带宽，如果进程 0 需要进程 1 的数据才能向下继续执行，我们不希望来自进程 1 的数据由于很多其他进程也同时发送的消息而被延迟。

附录 B 中给出了 `print_row_striped_matrix` 的源代码。

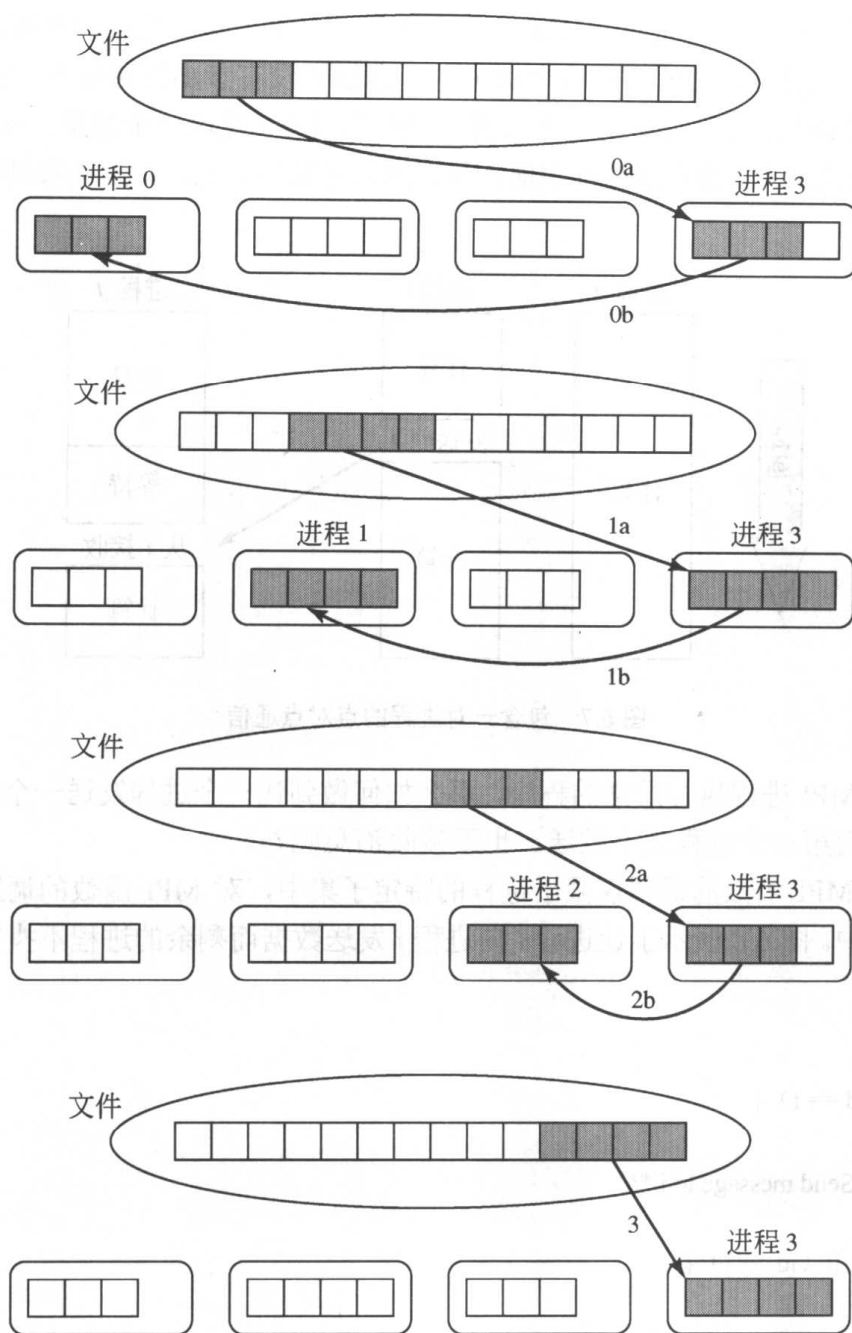


图 6.6 单进程管理文件输入的例子。在本图中共有 4 个进程，标号分别为 0、1、2 和 3。进程 3 打开文件读取数据。在步骤 0a，进程 3 读取进程 0 所需的数据，在步骤 0b 将数据发送给进程 0，在步骤 1 和步骤 2 分别为进程 1 和 2 进行同样的操作。在步骤 3，该进程输入自己的数据

## 6.5 点对点通信

在我们实现的从文件中读取数据的函数中，进程  $p-1$  读取连续的矩阵行数据，然后把这些数据直接发送到负责这些数据的进程中。在打印矩阵的函数中，每个进程（进程 0 除外）向进程 0 发送包含了其矩阵数据的消息。进程 0 接收这些消息并将这些数据打印到标准输出上。这些都是点对点通信的例子。

点对点通信包括了一对进程。相反，聚合通信则包含了这一组中的所有进程。

图 6.7 展示了点对点通信。在这个例子中，进程  $h$  没有参加到通信中，它继续执行操作其局部变量的语句。进程  $i$  进行本地计算，然后向进程  $j$  发送一个消息。在发送消息后，进程  $i$  继续自己的计算。进程  $j$  进行本地计算，然后阻塞住自己，直到接收到从进程  $i$  发来的消息。

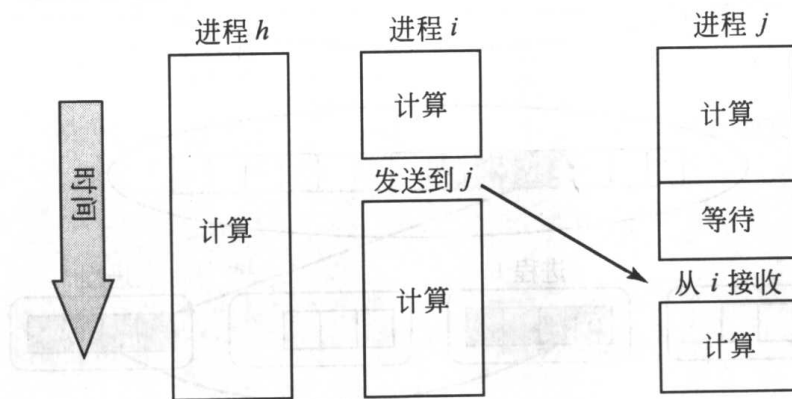


图 6.7 包含一对进程的点对点通信

如果每个 MPI 进程执行同样的程序，那么如何做到让一个进程发送一个消息，另一个进程接收消息而第三个进程既不发送、也不接收消息呢？

为了让对 MPI 函数的调用仅限于进程的特定子集中，对 MPI 函数的调用必须放在条件执行的代码中。图 6.8 演示了让进程  $i$  向进程  $j$  发送数据而剩余的进程不执行这些消息传递函数的方法。

```
...
if (id == i) {
    ...
    /* Send message to j */
    ...
} else if (id == j) {
    ...
    /* Receive message from i */
    ...
}
...
```

图 6.8 进行点对点通信的 MPI 函数经常出现在条件执行的代码中

现在，让我们看看这两个用于执行点对点通信的 MPI 函数的定义。

### 6.5.1 函数 MPI\_Send

发送进程调用函数 MPI\_Send:

```
int MPI_Send(
```

```
void *message,  
int count,  
MPI_Datatype datatype,  
int dest,  
int tag,  
MPI_Comm comm  
)
```

第 1 个参数 `message` 是被传送数据的起始地址。第 2 个参数 `count` 是被传送的数据项的个数。第 3 个参数 `datatype` 则说明了数据项的类型。第 4 个参数 `dest` 是接收消息的进程号。第 5 个参数 `tag` 是此消息的一个标记，可以用来说明消息的不同用途。最后一个参数 `comm` 是此消息所在的通信域。

函数 `MPI_Send` 会阻塞程序的执行，直到消息缓冲区重新为空。通常运行时系统会将消息复制到系统缓冲区中，使得 `MPI_Send` 可以返回调用者，但是运行时系统并非一定要进行这个操作。

## 6.5.2 函数 `MPI_Recv`

消息的接收进程调用函数 `MPI_Recv`:

```
int MPI_Recv(  
void *message,  
int count,  
MPI_Datatype datatype,  
int source,  
int tag,  
MPI_Comm comm,  
MPI_Status *status  
)
```

第 1 个参数 `message` 是接收数据要被存放的空间的起始地址。第 2 个参数 `count` 是接收进程所能够接收的最大数据项数。第 3 个参数 `datatype` 是数据项的类型。第 4 个参数 `source` 是发送消息的进程号。第 5 个参数 `tag` 指出了对此消息所预期的标记值。第 6 个参数 `comm` 是此消息所在的通信域。

注意第 7 个参数 `status` 出现在 `MPI_Recv` 中，但没有出现在 `MPI_Send` 中。在调用 `MPI_Recv` 之前，你需要分配一条类型为 `MPI_Status` 的记录，这是惟一的用戶可访问的 MPI 的数据结构，参数 `status` 就是指向这条记录的指针。

函数 `MPI_Recv` 会阻塞程序的执行直到消息接收完毕（或出现了使得函数返回的错误）。当函数 `MPI_Recv` 返回时，状态记录内部包含了刚刚结束的函数的状态信息，特别有以下几项：

- `Status->MPI_source` 此消息的发送进程号；
- `status->MPI_tag` 此消息的 `tag` 值；
- `status->MPI_ERROR` 错误状况。

为什么在 `MPI_Recv` 中已经指定了发送消息的进程号和 `tag` 的情况下, 还要再次从状态记录中查询这些信息呢? 原因是, 在指定参数的时候, 可以使用常数 `MPI_ANY_SOURCE` 来让接收进程接收从任何进程发来的消息。同样地, 可以让接收进程接收任何 `tag` 值地消息, 只要把 `MPI_Recv` 的第 5 个参数设为 `MPI_ANY_TAG`。在这些情况下, 需要查看状态记录以确定发送进程和/或消息的 `tag` 值。

### 6.5.3 死锁

“如果一个进程在等待一个永远不可能为真的条件, 则该进程处于死锁状态”【3】。在开发 MPI 程序的时候, 调用 `MPI_Send` 和 `MPI_Recv` 造成死锁的情况并不罕见。

比如考虑有 2 个进程, 其进程号分别为 0 和 1。每个进程都希望计算 `a` 和 `b` 的平均值。进程 0 有 `a` 的最新值, 进程 1 有 `b` 的最新值。进程 0 必须从进程 1 读取 `b`, 进程 1 也必须从进程 0 读取 `a`。考察下面的实现:

```
float a, b, c;
int id; /* Process rank */
MPI_Status status;
...
if(id == 0){
    MPI_Recv(&b, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD, &status);
    MPI_Send(&a, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
    c = (a + b) / 2.0;
} else if(id == 1){
    MPI_Recv(&a, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Send(&b, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
    c = (a + b) / 2.0;
}
```

在调用 `MPI_Send` 之前, 进程 0 在 `MPI_Recv` 内部阻塞, 等待进程 1 的消息的到达。同样, 进程 1 也在 `MPI_Recv` 内部阻塞, 等待进程 0 的消息的到达。这两个进程发生了死锁。

好吧, 这个错误相当明显 (尽管你也许会惊讶于在实际程序中这类错误会出现的有多么频繁), 让我们来看一个更加隐蔽的同样引起死锁的错误。

我们解决的问题还是同样的。进程 0 和 1 希望交换浮点数, 下面是代码:

```
float a, b, c;
int id; /* Process rank */
MPI_Status status;
...
if(id == 0){
    MPI_Send(&a, 1, MPI_FLOAT, 1, 1, MPI_COMM_WORLD);
    MPI_Recv(&b, 1, MPI_FLOAT, 1, 1, MPI_COMM_WORLD, &status);
    c = (a + b) / 2.0;
} else if(id == 1){
```

```

MPI_Send(&b, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
MPI_Recv(&a, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
c = (a + b) / 2.0;
}

```

现在两个进程都在接收数据前先发送数据，但它们仍然会死锁。你能看出错误来吗？进程 0 发送了一个 tag 为 1 的消息，并试图接收一个 tag 为 1 的消息。同时，进程 1 发送了一个 tag 为 0 的消息，并试图接收一个 tag 为 0 的消息。两个进程都在 MPI\_Recv 中被阻塞，因为谁都不能接受到具有正确 tag 的消息。

另一个常见的错误是发送进程将消息发送到错误的目标进程，或是接收进程试图从错误的源进程接收消息。

## 6.6 并行程序的说明

现在可以继续对 Floyd 算法的并行实现了。图 6.9 给出了我们的并行程序。

这里使用了一个 typedef 和一个宏来指出矩阵的类型。如果决定修改程序，在双精度浮点数的矩阵而不是现有的整数矩阵中计算最短路径，只需要修改两行程序，如下所示：

```

typedef double dtype;
#define MPI_TYPE MPI_DOUBLE

```

main 函数负责读取和打印初始的距离矩阵，调用最短路径函数，并打印结果矩阵。注意 main 函数检查矩阵以保证其为方矩阵。如果矩阵的行数不等于其列数，main 函数将调用 terminate 函数。terminate 函数将打印错误信息，关闭 MPI 并结束程序的执行。附录 B 中给出了 terminate 源代码。

现在来看真正实现了 Floyd 算法的函数。函数 compute\_shortest\_paths 有 4 个参数：进程号、进程数、指向该进程所负责的距离矩阵的指针以及矩阵的大小。

前面已经提到，要执行计算：

```

a[i][j] = MIN (a[i][j], a[i][k]+a[k][j]);

```

在第 k 个循环迭代的时候，第 k 行必须广播到所有的进程。

```

/*
 * Floyd's all-pairs shortest-path algorithm
 */
#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"
typedef int dtype;
#define MPI_TYPE MPI_INT
int main (int argc, char *argv[]) {
    dtype** a; /* Doubly-subscripted array */

```



```

dtype* storage; /* Local portion of array elements */
int i, j, k;
int id; /* Process rank */
int m; /* Rows in matrix */
int n; /* Columns in matrix */
int p; /* Number of processes */
void compute_shortest_paths (int, int, int**, int) ;
MPI_Init (&argc, &argv) ;
MPI_Comm_rank (MPI_COMM_WORLD, &id) ;
MPI_Comm_size (MPI_COMM_WORLD, &p) ;
read_row_striped_matrix (argv[1], (void *) &a,
    (void *) &storage, MPI_TYPE, &m, &n, MPI_COMM_WORLD) ;
if (m != n) terminate (id, "Matrix must be square\n") ;
print_row_striped_matrix ((void **) a, MPI_TYPE, m, n, MPI_COMM_WORLD) ;
compute_shortest_paths (id, p, (dtype **) a, n) ;
print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
    MPI_COMM_WORLD) ;
MPI_Finalize () ;
}

void compute_shortest_paths (int id, int p, dtype **a, int n)
{
    int i, j, k;
    int offset; /* Local index of broadcast row */
    int root; /* Process controlling row to be bcast */
    dtype* tmp; /* Holds the broadcast row */
    tmp = (dtype *) malloc (n * sizeof (dtype)) ;
    for (k = 0; k < n; k++) {
        root = BLOCK_OWNER (k, p, n) ;
        if (root == id) {
            offset = k - BLOCK_LOW (id, p, n) ;
            for (j = 0; j < n; j++)
                tmp[j] = a[offset][j];
        }
        MPI_Bcast (tmp, n, MPI_TYPE, root, MPI_COMM_WORLD) ;
        for (i = 0; i < BLOCK_SIZE (id, p, n) ; i++)
            for (j = 0; j < n; j++)
                a[i][j] = MIN (a[i][j], a[i][k] + tmp[j]) ;
    }
    free (tmp) ;
}

```

图 6.9 Floyd 算法的 MPI 实现

每个进程分配一个  $n$  个整数的数组 `tmp`，可以用于存放第  $k$  行的值。

与在串行算法中相同, 并行算法要循环  $n$  次。在每次迭代中, 各个进程都首先确定控制第  $k$  行数据的进程, 该进程是广播树的根节点。在调用了 `MPI_Bcast` 后, 每个进程都在 `tmp` 中存放了一份第  $k$  行数据的副本。因此, 上面所示的赋值操作转化为:

```
a[i][j] = MIN(a[i][j], a[i][k]+tmp[j]);
```

## 6.7 分析和测试

容易看出串行版本的 Floyd 算法的时间复杂度为  $\Theta(n^3)$ 。现在我们来分析一下我们的并行 Floyd 算法的复杂度。

最内层循环, 用来更新矩阵  $A$  的一行, 与串行算法中的最内层循环完全一样, 其时间复杂度为  $\Theta(n)$ 。给定一个矩阵  $A$  的按行块分解 (rowwise block-striped decomposition), 每个进程在中间循环最多执行  $\lceil n/p \rceil$  次迭代, 因此内层循环的复杂度为  $\Theta(n^2/p)$ 。

在中间循环的前面是广播操作。从一个处理器发送一个长度为  $n$  的消息到另一个处理器的时间复杂度为  $\Theta(n)$ 。由于广播到  $p$  个处理器需要  $\lceil \log p \rceil$  个消息传递步骤, 每个循环迭代的广播操作的总的时间复杂度为  $\Theta(n \log p)$ 。

并行算法最外层循环的每个循环迭代中必须计算新的根处理器, 需要常数时间。根处理器将矩阵  $A$  的相应行复制到数组 `tmp` 中, 需要  $\Theta(n)$  时间。最外层循环执行  $n$  次。

因此, 并行算法的总的时间复杂度为:

$$\Theta(n(1+n+n \log p + n^2/p)) = \Theta(n^3/p + n^2 \log p)$$

现在让我们来预测一下我们的并行算法在一个集群系统上的运行时间。并行程序需要  $n$  次广播, 每次广播有  $\lceil \log p \rceil$  步。每一步骤包括传送长度为  $4n$  的消息。因此并行程序的预期执行时间为:

$$n \lceil \log p \rceil (\lambda + 4n/\beta)$$

如果更新一个单元所需的平均时间为  $\chi$ , 那么并行程序的预期计算时间是  $n^2 \lceil \log p \rceil \chi$ 。

把计算时间和广播时间加起来可以得到并行算法预期执行时间的一个简单表达式:

$$n^2 \lceil n/p \rceil \chi + n \lceil \log p \rceil (\lambda + 4n/\beta)$$

但是, 这个表达式过高地估计了并行程序的执行时间, 因为其忽略了计算和通信之间可能有显著的重叠时间。

图 6.10 展示了在 4 个进程上执行的 Floyd 算法的前 4 次循环迭代。假设  $n \geq 16$ , 因此进程 0 是前 4 次迭代的根进程。在每次广播步骤中, 进程 0 向进程 2 和 1 发送消息。在它发起了这些消息后, 就可以开始更新自己所负责的矩阵数据, 通信和计算发生了重叠。

考察进程 1, 在收到进程 0 发来的第 0 行数据之前它不会开始更新自己所控制的矩阵。在第一次迭代的时候, 进程 1 必须等待消息的到来。但是这次延时使得进程 1 的计算阶段与进程 0 错开了。进程 1 完成自己的第 1 次循环迭代的时间要比进程 0 晚。在进程 0 发起第 2 行数据的传送的时候, 进程 1 还仍然在进行第 1 行数据的计算, 因此进程 1 在等待第 2 行数据的时候将不会等待同样长的时间。

在图 6.10 中, 每个循环迭代的计算时间超过了传送消息所需的时间。因此, 在第一个循环迭代后, 每个进程都花费同样的时间来等待或是发起消息:  $\lceil \log p \rceil \lambda$ 。

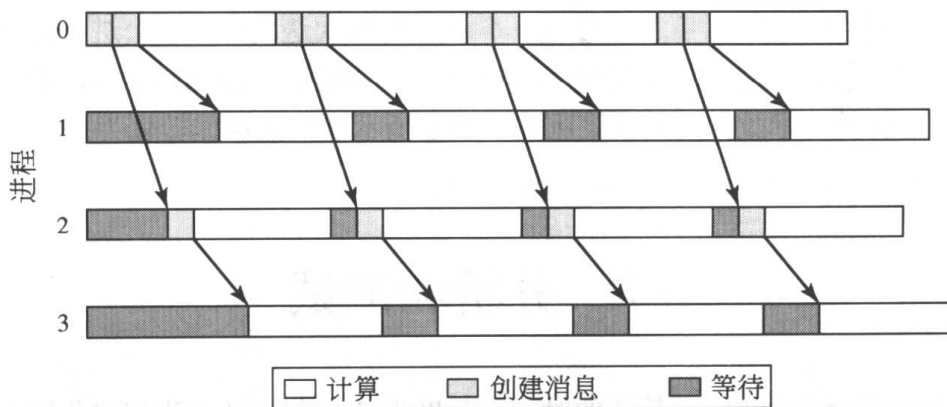


图 6.10 在执行并行 Floyd 算法的过程中, 在消息传递 (用箭头表示) 和计算之间有显著的重叠

如果  $\lceil \log p \rceil 4n/\beta < \lceil n/p \rceil n\chi$ , 在第一个循环迭代后的消息传递时间会完全被计算时间覆盖, 不应计入整个的执行时间中。在我们的集群系统上, 当  $n=1000$  的时候上述条件成立。因此, 估算并行程序时间的更准确的表达式为:

$$n^2 \lceil n/p \rceil \chi + n \lceil \log p \rceil \lambda + \lceil \log p \rceil 4n/\beta$$

图 6.11 画出了在解决规模为 1000 的问题的时候, 我们的并行程序在集群系统上的预期和实际执行时间, 其中  $\chi=25.5$  ns,  $\lambda=250\mu$ s,  $\beta=107$ 。在 2~7 个处理器上的预期和实际执行时间的平均误差为 3.8%。

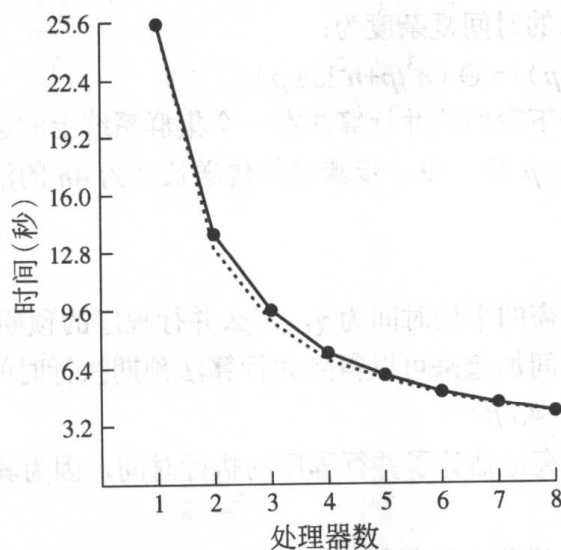


图 6.11 在集群系统上, 解决规模为 1000 的问题时, Floyd 算法并行实现的预期执行时间 (点线) 和实际执行时间 (实线)

## 6.8 本章小结

我们开发了用 C 语言和 MPI 开发了 Floyd 算法的并行版本。对于中等规模的任务, 并行程序在集群系统上得到了良好的加速比。我们的实现在处理器之间采用了点对点消息。我们介绍了函数 `MPI_Send` 和 `MPI_Recv` 来支持点对点消息。

我们同时也开始开发一个程序库，用于支持矩阵和向量在各种数据分解方案下的输入、输出和重新分布。本章用到的两个输入/输出函数基于矩阵的逐行块分解。函数 `read_row_striped_matrix` 从文件中读取矩阵并将元素分配到一组进程中去。函数 `print_row_striped_matrix` 将在一组进程上分布的矩阵元素打印出来。

## 6.9 主要术语

adjacency matrix	邻接矩阵
all-pairs shortest-path problem	全点对的最短路径问题
directed graph	有向图
graph	图
point-to-point communication	点对点通信
weighted graph	有权图

## 6.10 参考文献

Floyd 算法最早于 1962 年发表在 *Communications of the ACM* 【27】。该算法实际上是几个月前 *Journal of the ACM* 上发表的 Warshall 的传递闭包算法的一般化 【111】。

Foster 比较了两个版本的 Floyd 算法 【31】。第一种算法将同一行内的原始任务聚集起来，得到 rowwise block-striped 数据划分。第二种算法将 2 维数据块的原始任务聚集起来。下一章我们将介绍这种称为“棋盘式分解”的方法。Foster 指出第二种方法更加优越。

Grama 等也描述了基于棋盘数据分解方案的并行 Floyd 算法实现 【44】。

## 6.11 练习题

6.1 假如我们选择对  $n$  个元素（标号为  $0, 1, \dots, n-1$ ）按块聚集到  $p$  个进程上（标号为  $0, 1, \dots, p-1$ ），其中进程  $i$  负责元素  $\lfloor in/p \rfloor$  到  $\lfloor (i+1)n/p \rfloor - 1$ 。证明最后一个进程负责的元素个数为  $\lceil n/p \rceil$ 。

6.2 在图 6.6 中的文件输入的例子中，从进程 3 输入并传送数据比从进程 0 输入数据的优点在什么地方？

6.3 如果我们决定使用按列块划分来解决 Floyd 全点对最短路径问题，概述在并行实现中所需的修改？

6.4 如果我们决定使用按行交叉条分解，如图 12.3 (a) 所示，来解决 Floyd 全点对最短路径问题，概述在并行实现中所需的修改？

6.5 考虑基于另一种数据分解的 Floyd 算法。假设  $p$  是一个平方数且  $n$  是  $\sqrt{p}$  的倍数，每个进程负责一个  $A$  的方子矩阵，其大小为  $(n/\sqrt{p}) \times (n/\sqrt{p})$ 。

- (a) 描述最外层循环每次迭代所需的通信。
- (b) 用  $n$ 、 $p$ 、 $\lambda$  和  $\beta$  为参数描述并行算法的通信时间。
- (c) 将此通信时间与本章所开发的并行算法的通信时间进行比较。

6.6 如果我们用来测试并程序集群系统有 16 个 CPU, 估算在 16 个处理器上解决规模为 1000 的问题所需的时间。

6.7 假定我们仍然使用本章中用来进行测试的集群系统, 估算在 1, 2, ..., 8 个处理器上解决规模为 500 和 2000 的问题所需的时间。

6.8 假设发送一个  $n$  个字节消息所需的时间为  $\lambda + n/\beta$ 。写一个程序来实现 “ping pong” 测试, 以确定在你的并行系统上的  $\lambda$  (延迟) 和  $\beta$  (带宽) 的值。程序需要运行两个进程, 进程 0 记录时间并发消息给进程 1。进程 1 接收消息后, 马上将消息送回给进程 0。进程 0 接收消息并记录时间。这个过程所用的时间除以 2 就是一次消息传递所用的平均时间。将消息发送多次, 并发送不同大小的消息以得到足够的数据来估算  $\lambda$  和  $\beta$ 。

6.9 使用 MPI\_Send 和 MPI\_Recv 来设计你自己的 MPI\_Reduce。可以假设:

```
datatype = MPI_INT,
operator = MPI_SUM, and
comm = MPI_COMM_WORLD
```

比较你的实现与系统 MPI 库中真正的 MPI\_Reduce 的性能。

6.10 使用 MPI\_Send 和 MPI\_Recv 来设计你自己的 MPI\_Bcast。比较你的实现与系统 MPI 库中真正的 MPI\_Bcast 的性能。

6.11 对第 5 章开发的 Eratosthenes 筛法程序, Lester 建议使用发送和接收流水线来代替其中的广播操作以提高性能【71】。换句话说, 进程 0 不需要将下一个素数广播给其他进程, 而仅将其发送给进程 1, 进程 1 将接收该素数并传给进程 2, 依此类推。对于每个找到的素数, 每个进程所需的最大通信次数从  $\lceil \log p \rceil$  减少到了 2。

- (a) 实现一个新的并行 Eratosthenes 筛法, 将广播步骤替换成发送/接收步骤。
- (b) 使用第 5 章得出的分析模型预测改进后程序在执行时间上的改进。
- (c) 在并行计算机上测试原有算法和新算法。

6.12 编写两个并程序, 其输入为地形图, 输出为给定太阳位置时的阴影区域。

对每个程序来说输入和输出文件的格式都是相同的。输入文件是  $n \times n$  的矩阵, 代表在正方形地面上均匀分布的  $n^2$  个点。矩阵元素的值是 0~100 的整数, 代表相应位置的高度。

输出文件是  $n \times n$  的矩阵, 元素的值是 0 或 1。0 代表该位置有阳光照射, 1 代表该位置是阴影区域。

(a) 假定阳光是从西边照过来的。如果某个位置的西边有山挡住了阳光, 该位置就处于阴影区域。下面是如何计算哪些位置是在阴影中的: 首先根据定义最左边的行是阳光照射的。对于剩下的位置  $(i, j)$ , 如果  $(i, j-k)$  的高度比  $(i, j)$  的位置高  $4k$  以上 (包括  $4k$ ) 的话, 那么  $(i, j)$  处于阴影区。

例如, 下面是一个输入矩阵的例子 (方向字母是帮助你认清地图的方向, 但并不在输入文件中):

			N				
	0	0	5	3	2	0	
	5	7	11	10	8	4	
W	15	9	16	7	8	3	E
	22	15	4	11	7	2	
	10	2	10	9	7	2	
	0	0	5	7	4	0	
			S				

正确的输出矩阵是:

0	0	0	0	0	0
0	0	0	0	0	1
0	1	0	1	1	1
0	1	1	0	1	1
0	1	0	0	0	1
0	0	0	0	0	1

(b) 假设阳光是从西北方向照射过来的。如果某个位置的西北边有山挡住了阳光, 该位置就处于阴影区域。下面是如何计算哪些位置是在阴影中的: 首先根据定义最左边的行是阳光照射的。对于剩下的位置  $(i, j)$ , 如果  $(i-k, j-k)$  的高度比  $(i, j)$  的位置高  $4k$  以上 (包括  $4k$ ) 的话, 那么  $(i, j)$  处于阴影区。

例如, 给定与上面相同的输入矩阵, 下面是正确的输出矩阵:

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	1	0	1
0	1	1	0	1	1
0	1	1	1	1	1

6.13 1970 年, 普林斯顿的数学家 John Conway 发明了游戏 Life (生命)。Life 是细胞自动机的一个例子。它包括矩形的细胞网格, 每个细胞可以处于 2 个状态之一: 活和死。游戏由很多次迭代组成。每次迭代中, 如果一个死细胞正好有 3 个邻居, 则该死细胞变活。如果一个活细胞有 2 个或 3 个邻居, 则它保持不变。如果一个活细胞有 2 个以下或 3 个以上的邻居, 则该细胞变死。所有细胞都同时进行更新。图 6.12 展示了在一个小的细胞网格上的 Life 游戏的 3 次迭代过程。

写一个并行程序, 从  $m \times n$  的矩阵中读取游戏的初始状态, 然后进行 Life 游戏的  $j$  次迭代, 并每  $k$  次迭代打印出一次游戏的状态。 $j$  和  $k$  由程序的命令行给出。

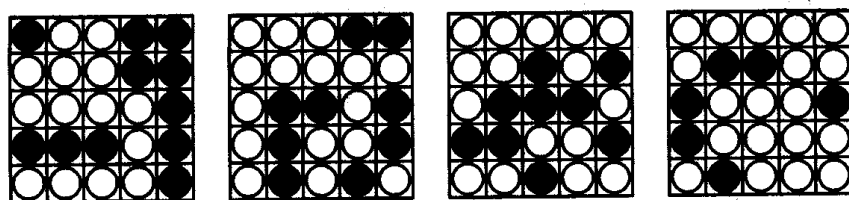


图 6.12 Conway 的 Life 游戏的一个初始状态和 3 次迭代过程

# 第7章 性能分析

The Highest and best form of efficiency is the spontaneous cooperation of a free people.

Woodrow Wilson

## 7.1 概 述

准确地预测你所设计并行算法的性能，可以帮助你决定是否要真正地花费时间来编码和调试这个算法。分析并行程序的运行时间可以帮助你了解提高性能的障碍，并预测通过增加处理器数能够提高性能的幅度。这一章将帮助你获得上面这些技能。

本章从推导一个并行程序所能达到的加速比的通用公式开始，然后介绍著名的性能预测公式：Amdahl 定律、Gustafson-Barsis 定律、Karp-Flatt 度量以及等效加速比度量。Amdahl 定律可以帮助你决定一个程序是否值得并行化。Gustafson-Barsis 定律提供了评估并行程序性能的方法。Karp-Flatt 度量可以帮助确定影响加速比的主要障碍是串行代码还是并行开销。等效加速比度量主要用来衡量在并行计算机上执行并行算法的可扩展性。它可以帮助你选择能够在处理器数目增加的情况下获得高性能的算法设计。

## 7.2 加速比和效率

我们设计和实现并行程序的目的是希望它们比相应的串行代码运行得更快。加速比 (speedup) 是串行程序执行时间和并行程序执行时间之比：

加速比 = 串行程序执行时间 / 并行程序执行时间

在我们研究过的例子当中，我们已经发现并行算法的操作可以分为 3 类：

- 必须串行执行的计算；
- 可以并行执行的计算；
- 并行开销（通信操作和冗余计算）。

根据上面的分类，我们可以得到一个关于加速比的简单的模型。让  $\psi(n, p)$  表示在  $p$  个处理器上解决规模为  $n$  的问题时的加速比， $\sigma(n)$  表示计算中内在的串行部分， $\phi(n)$  表示可以并行执行的计算， $\kappa(n, p)$  表示并行计算开销所需的时间。

一个在一个处理器上执行的串程序，一次只能执行一项计算操作，因此需要  $\sigma(n) + \phi(n)$  时间来执行所需的计算。串程序不需要处理器间通信，因此串程序执行时间的表达式中不含有  $\kappa(n, p)$ 。

现在我们来分析可能的最短并行执行时间。计算中内在的串行部分不能从并行化中获

益，在并程序中，不管可以使用多少个处理器，这部分的计算时间都是 $\sigma(n)$ 。在最好情况下，可并行计算的部分在 $p$ 个处理器上完全平均分配，其计算时间为 $\varphi(n)/p$ 。最后，我们必须加上并程序所需的处理器间通信所需的时间 $\kappa(n, p)$ 。

我们已经进行了最乐观的假设，即计算的并行部分可以在处理器上完全平均分配。如果实际情况并非如此，并行计算所需的时间将会更长，加速比则会更低。因此实际的加速比将会以我们刚才所定义的串行执行时间和并行执行时间之比为上界。下面是我们关于加速比的完整的表达式：

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

增加处理器会减少计算时间（通过将任务分配到多个处理器上），但是会增加通信时间。到某一点通信时间的增加会超过计算时间的减少量，如图 7.1 所示。此时，总的执行时间开始增加。由于加速比与执行时间成反比，加速比曲线出现了“拐点”并开始向下延伸。

并程序的效率是处理器利用率的度量。我们定义效率为加速比除以使用的处理器数：

$$\text{效率} = \text{串行执行时间} / (\text{使用的处理器数} \times \text{并行执行时间})$$

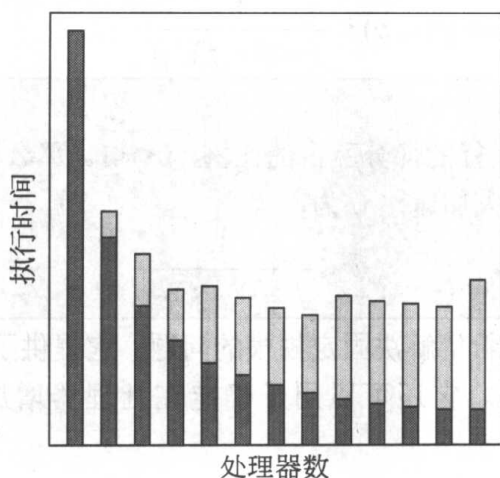


图 7.1 并行算法由计算部分和通信部分组成。计算部分（黑色条）是使用的处理器数的递减函数，通信部分（灰色条）是处理器数的增函数。对于任意固定规模的问题，都存在一个能够获得最小执行时间最优的处理器数

更为形式化地，使用 $\varepsilon(n, p)$ 表示在 $p$ 个处理器上解决规模为 $n$ 的并行计算效率。根据我们前面对加速比的定义：

$$\begin{aligned} \varepsilon(n, p) &\leq \frac{\sigma(n) + \varphi(n)}{p(\sigma(n) + \varphi(n)/p + \kappa(n, p))} \\ \Rightarrow \varepsilon(n, p) &\leq \frac{\sigma(n) + \varphi(n)}{p\sigma(n) + \varphi(n) + p\kappa(n, p)} \end{aligned}$$

由于所有项都大于或等于 0，我们得到  $0 \leq \varepsilon(n, p) \leq 1$ 。



## 7.3 Amdahl 定律

考察我们刚刚推导出的表达式。

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

因为  $\kappa(n, p) > 0$ ,

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)} \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

用  $f$  表示计算中的串行部分, 换句话说,  $f = \sigma(n) / (\sigma(n) + \varphi(n))$ , 于是,

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{\sigma(n)/f}{\sigma(n) + \sigma(n)(1/f - 1)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{1/f}{1 + (1/f - 1)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{1}{f + (1 - f)/p}$$

### Amdahl 定律

$f$  为计算中必须串行执行的部分所占的比例,  $0 \leq f \leq 1$ 。那么该计算在  $p$  个处理器的并行计算机上所能达到的最大加速比  $\psi$  为:

$$\psi \leq \frac{1}{f + (1 - f)/p}$$

Amdahl 定律基于我们希望解决固定规模的问题。它提供了用一定数量的处理器并行解决问题的加速比的上界。它还可以用于确定在处理器增加的情况下的渐进加速比 (asymptotic speedup)。

#### 例 1

假定我们试图确定是否值得为一个问题开发并程序。通过程序性能测试我们发现 90% 的时间用于执行可并行化的函数, 剩下的 10% 的时间用于必须在一个处理器上串行执行的函数。如果我们使用 8 个处理器来执行程序的话, 我们所期望的最大加速比是多少?

解:

由 Amdahl 定律:

$$\psi \leq \frac{1}{0.1 + (1 - 0.1)/8} \approx 4.7$$

我们期望的加速比为 4.7 或更少。

#### 例 2

如果一个并行程序中 25% 的操作必须串行执行, 那么所能达到的最大加速比是多少。

解:

可达到的最大加速比是：

$$\lim_{p \rightarrow \infty} \frac{1}{0.25 + (1 - 0.2)/p} = 4$$

### 例 3

假如我们实现了一个串行程序的并行版本，算法复杂度为  $\Theta(n^2)$ ， $n$  是数据集的大小。假定输入数据和输出结果所需的时间是：

$$(18000 + n) \mu\text{s}$$

这是程序的串行部分。程序的计算部分可以并行执行，其执行时间为：

$$(n^2/100) \mu\text{s}$$

当问题规模为 10 000 时，此并行程序所能达到的最大加速比是多少？

解：

由 Amdahl 定律：

$$\psi \leq \frac{(28\,000 + 1\,000\,000) \mu\text{sec}}{(28\,000 + 1\,000\,000/p) \mu\text{sec}}$$

图 7.2 中的虚线表示了 Amdahl 定律得出的加速比上界。

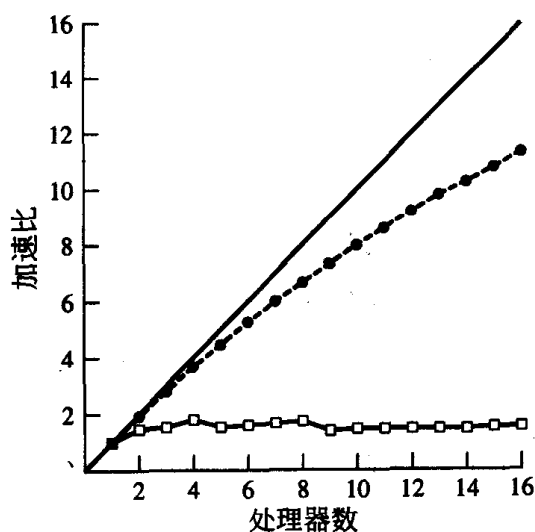


图 7.2 Amdahl 定律预测的加速比（虚线）比考虑了通信开销后所预测的加速比（实线）要高

### 7.3.1 Amdahl 定律的局限

Amdahl 定律忽略了引入并行性所带来的开销。让我们回到上一个例子。假设并行版本的程序由  $\lceil \log n \rceil$  个通信点，在这些通信点，通信时间为：

$$10\,000 \lceil \log p \rceil + (n/10) \mu\text{sec}$$

对于规模为 10 000 的问题，总的通信时间为：

$$14(10\,000 \lceil \log p \rceil + 1\,000) \mu\text{sec}$$

现在我们把所有因素都包括到加速比的计算公式中： $\sigma(n)$ 、 $\varphi(n)$  和  $\kappa(n, p)$ 。对在  $p$  个处理器上运行规模为 10 000 的问题，预计可达到的加速比为：

$$\psi \leq \frac{(28\,000 + 1\,000\,000)}{(42\,000 + 1\,000\,000/p + 140\,000 \lceil \log p \rceil)}$$

图 7.2 中的实线画出了使用这个更加全面的公式算出的加速比上限。将通信时间考虑到计算过程中, 能够更加准确地预测并程序的性能。

### 7.3.2 Amdahl 效应

通常  $\kappa(n, p)$  比  $\varphi(n)$  的复杂度要低, 对我们刚才所考虑的假想问题来说就是这样:  $\kappa(n, p) = \Theta(n \log n + n \log p)$ ,  $\varphi(n) = \Theta(n^2)$ 。问题规模增加时, 计算时间的增长比通信时间要快。因此, 对于固定数目的处理器来说, 加速比往往是问题规模的增函数。这称为 Amdahl 效应【42】。图 7.3 通过画出我们假想问题的加速比曲线来说明 Amdahl 效应。当问题规模  $n$  增加时, 加速比曲线的高度也同时增加。

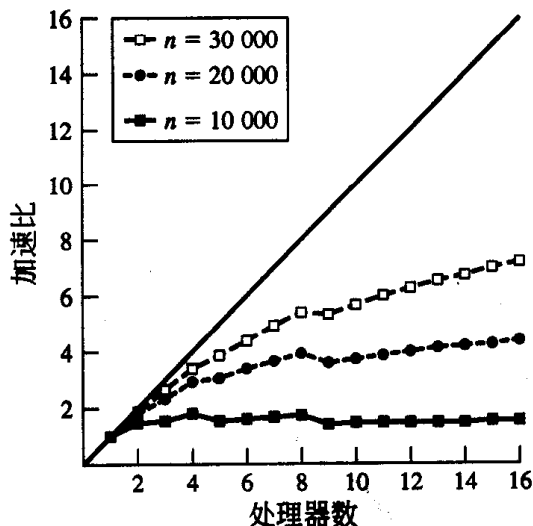


图 7.3 对于固定数量的处理器, 加速比往往是问题规模的增函数。这种现象称为 Amdahl 效应

## 7.4 Gustafson-Barsis 定律

Amdahl 定律假设并行计算的主要目标是缩短执行时间。它将问题规模当作一个常数, 并展示处理器个数的增加是如何减少执行时间的。

但是应用并行性的目标常常是在固定的时间内增加计算结果的准确性。例如, 一个研究超音速飞行器周围气流的工程师可能希望她的计算机能够在 1 小时内得出结果 (比如, 吃午饭的时间)。如果她可以使用一台具有多个处理器的计算机, 对她来说得到更详细的结果比以更快的时间得到同样的结果更有意义。

如果我们把时间作为常数而让问题规模随着处理器数增加的话会出现什么情况? 计算内在的串行部分所占地比例随着问题规模的增加而减小 (Amdahl 效应)。增加处理器数量使得我们增加问题规模, 减少内在的串行部分在计算中的比例, 并增加串行执行时间与

并行执行时间的比值（加速比）。

对于我们刚过那个推导出的加速比表达式，因为  $\kappa(n, p) \geq 0$ ,

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

用  $s$  表示在并行计算过程中花在内在的串行部分的时间比例。花在并行部分所占的时间比例就是  $1-s$ 。数学上表示为：

$$s = \frac{\sigma(n)}{\sigma(n) + \varphi(n)/p}$$

$$(1-s) = \frac{\varphi(n)}{\sigma(n) + \varphi(n)/p}$$

因此，

$$\sigma(n) = (\sigma(n) + \varphi(n)/p)s$$

$$\varphi(n) = (\sigma(n) + \varphi(n)/p)(1-s)p$$

所以，

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{(\sigma(n) + \varphi(n)/p)(s + (1-s)p)}{\sigma(n) + \varphi(n)/p}$$

$$\Rightarrow \psi(n, p) \leq s + (1-s)p$$

$$\Rightarrow \psi(n, p) \leq p + (1-p)s$$

#### Gustafson-Barsis 定律

给定一个并行程序在  $p$  个处理器上解决规模为  $n$  的问题，让  $s$  表示在总执行时间中串行代码的比例，则此程序可达到的最大加速比  $\psi$  为

$$\psi \leq p + (1-p)s$$

Amdahl 定律从串行程序开始，通过预测其在多个处理器上的计算速度来确定加速比。Gustafson-Barsis 定律则正好相反。它从并行程序出发，估计并行计算比在单处理器上执行同样的计算可以快多少。

在许多情况下，假设单个处理器比  $p$  个处理器慢  $p$  倍是过于乐观了。例如，假设在一台有 16 个处理器的并行计算机上解决问题，每个处理器有 1GB 的本地内存。假设数据集占用了 15GB，而并行计算机的聚合内存刚好可以存放数据集和程序的多份副本。如果我们试图在单个处理器上解决同样的问题，那么整个的数据集将不能放入主存之中。如果执行程序的工作集超过 1GB，系统将出现颠簸现象，所需的执行时间会大大超过在 16 个处理器上执行并行部分所需时间的 16 倍。

这就是为什么我们说 Gustafson-Barsis 定律中，加速比是并行计算的时间去除在单 CPU 上解决同样问题所需的时间，如果单 CPU 系统有足够的内存。我们把 Gustafson-Barsis 定律所预测的加速比称为比例加速比 (scaled speedup)，因为通过从并行计算而不是串行计算出发，它允许问题规模成为处理器个数的增函数。

**例 1**

一个应用程序在 64 个处理器上需要执行 220 秒。测试发现 5% 的时间花在程序的串行部分上。这个应用程序的加速比是多少？

解：

因为  $s=0.05$ ，因此在 64 个 CPU 上的加速比为：

$$\psi = 64 + (1-64)(0.05) = 64 - 3.15 = 60.85$$

**例 2**

Vicki 希望能够在有 16 384 个 CPU 的系统上，执行一个对其雇主非常重要的问题的时候可以达到 15 000 的加速比，以此来说明购买该 3000 万美元的超级计算机的正确性。要达到上述目标，执行时间中可以用于串行操作的最大比例是多少？

解：

根据 Gustafson-Barsis 定律：

$$\begin{aligned} 15\,000 &= 16\,384 - 16\,383s \\ \Rightarrow s &= 1\,384/16\,383 \\ \Rightarrow s &= 0.084 \end{aligned}$$

## 7.5 Karp-Flatt 量度

由于 Amdahl 定律和 Gustafson-Barsis 定律忽略了代表并行开销的项  $\kappa(n, p)$ ，它们可能会过高地估计加速比或比例加速比。Karp 和 Flatt 提出了另一个性能量度，称为实验决定的串行比例 (experimentally determined serial fraction)，为我们洞察程序的性能提供了有意义的工具【59】。

我们表示并行程序在  $p$  个处理器上的执行时间为：

$$T(n, p) = \sigma(n) + \varphi(n)/p + \kappa(n, p)$$

其中  $\sigma(n)$  是计算中内在的串行部分， $\varphi(n)$  是可以并行执行的部分， $\kappa(n, p)$  是处理器通信和同步，以及冗余计算的开销。串行程序没有任何处理器间通信和同步开销，所以执行时间为：

$$T(n, 1) = \sigma(n) + \varphi(n)$$

我们定义并行计算中实验确定的串行部分  $e$  为：

$$e = (\sigma(n) + \kappa(n, p)) / T(n, 1)$$

因此，

$$\sigma(n) + \kappa(n, p) = T(n, 1)e$$

我们可以将并行程序的执行时间表示为：

$$T(n, p) = T(n, 1)e + T(n, 1)(1-e)/p$$

我们使用  $\psi$  代表  $\psi(n, p)$ 。因为加速比  $\psi = T(n, 1)/T(n, p)$ ，有  $T(n, 1) = T(n, p)\psi$ 。

因此，

$$\begin{aligned}
T(n, p) &= T(n, p)\psi e + T(n, p)\psi(1-e)/p \\
\Rightarrow 1 &= \psi e + \psi(1-e)/p \\
\Rightarrow 1/\psi &= e + (1-e)/p \\
\Rightarrow 1/\psi &= e + 1/p - e/p \\
\Rightarrow 1/\psi &= e(1-1/p) + 1/p \\
\Rightarrow e &= \frac{1/\psi - 1/p}{1-1/p}
\end{aligned}$$

**Karp-Flatt 量度【59】**

给定一个在  $p$  个处理器上展示了加速比  $\psi$  的并行计算, 其中  $p > 1$ , 那么试验确定的串行比例  $e$  定义为:

$$e = \frac{1/\psi - 1/p}{1-1/p}$$

由于两个原因, 试验确定的串行比例是一个有用的量度。首先, 它考虑了并行开销  $[\kappa(n, p)]$ , 这是 Amdahl 定律和 Gustafson-Barsis 定律所忽略的。第二, 它可以帮助我们发现在我们的简单并程序执行模型中所忽略的其他开销的来源。例如, 我们假设  $p$  个处理器在执行计算的并行部分时比单处理器快  $p$  倍, 这就是为什么  $T(n, 1)$  中的  $(n)$  项在  $T(n, p)$  中成为了  $\varphi(n)/p$ 。这个假设忽略了在多个处理器上进行计算划分可能不均匀的事实。例如, 假定我们有 19 个相等大小并不可再分的任务片断, 其中每个需要 1 个单位时间来完成。如果我们有 6 个处理器可供使用, 那么其中一个处理器必须承担 4 个任务, 而其他处理器只需承担 3 个, 因此并行执行时间为 4, 而不是 19/6。

对于一个固定规模的问题, 并行计算执行的效率通常随着处理器个数的增加而下降。通过使用实验确定的串行部分, 我们可以确定并行效率的原因是 (1) 缺少并行性 (2) 算法或体系结构中增加的开销。

**例 1**

在 1, 2, ..., 8 个处理器上测试一个并程序得到下面的加速比结果:

$p$	2	3	4	5	6	7	8
$\psi$	1.82	2.50	3.08	3.57	4.00	4.38	4.71

并程序在 8 个处理器上加速比仅为 4.71 的主要原因是什么?

解:

使用我们推导出的公式, 我们可以计算实验决定的串行比例  $e$ :

$P$	2	3	4	5	6	7	8
$\psi$	1.82	2.50	3.08	3.57	4.00	4.38	4.71
$e$	0.10	0.10	0.10	0.10	0.10	0.10	0.10

由于实验确定的串行部分比例并没有随着处理器的个数增加而增加, 可以看出性能不好的主要原因是有限的并行性, 即大部分计算是本质上串行的。

**例 2**

在 1, 2, ..., 8 个处理器上测试一个并程序得到下面的加速比结果:

$p$	2	3	4	5	6	7	8
$\psi$	1.87	2.61	3.23	3.73	4.14	4.46	4.71

并行程序在 8 个处理器上加速比仅为 4.71 的主要原因是什么?

解:

计算程序运行时的实验决定的串行比例  $e$ :

$p$	2	3	4	5	6	7	8
$\psi$	1.87	2.61	3.23	3.73	4.14	4.46	4.71
$e$	0.070	0.075	0.080	0.085	0.090	0.095	0.1

由于实验确定的串行部分比例随着处理器的个数增加而稳定增加,可以看出加速比差的主要原因是并行开销。这可能是由于进程启动、通信或同步所需的时间,或是由于系统结构上的限制。

## 7.6 等效指标

我们把在并行计算机上执行的一个并行程序称作并行系统。并行系统的可扩展性是当处理器增加的时候系统性能随之增加的能力。

正如我们已经看到的,由于通信复杂度通常比计算复杂度要低,加速比(效率)通常是问题规模的增函数。我们称之为 Amdahl 效应。因此我们可以通过增加问题规模来保证在处理器个数增加的时候并行系统保持同样的效率。

等效关系形式化地描述了这些思想。我们从最初的加速比定义来推导等效关系:

$$\begin{aligned}
 \psi(n, p) &\leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n)/p + \kappa(n, p)} \\
 \Rightarrow \psi(n, p) &\leq \frac{p(\sigma(n) + \phi(n))}{p\sigma(n) + \phi(n) + p\kappa(n, p)} \\
 \Rightarrow \psi(n, p) &\leq \frac{p(\sigma(n) + \phi(n))}{\sigma(n) + \phi(n) + (p-1)\sigma(n) + p\kappa(n, p)}
 \end{aligned}$$

我们定义  $T_o(n, p)$  来代表所有进程花费在原有串行算法以外的操作的全部时间。其组成部分之一是  $p-1$  个进程花在程序的内在串行部分的时间,另一组成部分是  $p$  个进程花费在处理器间通信和冗余计算上的时间。因此  $T_o(n, p) = (p-1)\sigma(n) + p\kappa(n, p)$ 。将  $T_o(n, p)$  代入刚才的方程,我们得到:

$$\Rightarrow \psi(n, p) \leq \frac{p(\sigma(n) + \phi(n))}{\sigma(n) + \phi(n) + T_o(n, p)}$$

由于效率等于加速比除以  $p$ :

$$\varepsilon(n, p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n) + T_o(n, p)}$$



$$\Rightarrow \varepsilon(n, p) \leq \frac{1}{1 + \frac{T_o(n, p)}{\sigma(n) + \varphi(n)}}$$

$T(n, 1)$  代表串行执行时间:

$$\begin{aligned} \Rightarrow \varepsilon(n, p) &\leq \frac{1}{1 + T_o(n, p)/T(n, 1)} \\ \Rightarrow \frac{T_o(n, p)}{T(n, 1)} &\leq \frac{1 - \varepsilon(n, p)}{\varepsilon(n, p)} \\ \Rightarrow T(n, 1) &\geq \frac{\varepsilon(n, p)}{1 - \varepsilon(n, p)} T_o(n, p) \end{aligned}$$

如果我们希望维持恒定的效率, 那么分数,

$$\frac{\varepsilon(n, p)}{1 - \varepsilon(n, p)}$$

是一个常数, 公式可以简化为:

$$T(n, 1) \geq CT_o(n, p)$$

#### 等效关系【43】

假定一个并行系统具有效率  $\varepsilon(n, p)$ , 定义  $C = \varepsilon(n, p) / (1 - \varepsilon(n, p))$ , 并且  $T_o(n, p) = (p-1)\sigma(n) + p\kappa(n, p)$ 。为了在处理器个数增加的情况下维持效率不变,  $n$  必须增加使得下面的不等式成立:

$$T(n, 1) \geq CT_o(n, p)$$

我们可以使用并行系统的等效性来确定可以维持特定效率的处理器个数范围。由于在处理器个数增加时并行开销会增大, 要保持效率就需要增加问题的规模。我们设计的算法假定所需的数据结构可以放在内存里完成, 问题的规模受到主存容量的限制, 因此空间是我们进行分析时的一个限制性因素。

假定一个并行系统具有等加速比关系  $n \geq f(p)$ 。如果  $M(n)$  表示规模为  $n$  的问题所需的内存容量,  $M^{-1}(n) \geq f(p)$  表示为了保持效率不变所需的内存大小如何作为  $p$  函数增加。可用的内存总容量是处理器个数的线性函数, 因此  $M(f(p))/p$  表示了为了保持效率不变, 每个处理器所需的内存容量如何作为  $p$  的函数增加。我们称  $M(f(p))/p$  为可扩展性函数。

$M(f(p))/p$  的复杂度确定了可保持常数效率的处理器个数范围, 如图 7.4 所示。如果  $M(f(p))/p = 1$ , 每处理器所需内存为常数, 并行系统是完全可扩展的。如果  $M(f(p))/p = \Theta(p)$ , 每处理器所需内存按处理器个数的线性函数增长。当有足够可用内存时, 有可能通过增加问题规模来保持恒定的效率。但是, 由于每处理器所需内存随着  $p$  线性增长, 终究会到某一点会达到系统的内存实际容量。超过这一点后, 如果处理器个数继续增加, 系统将无法维持等效。

我们可以对  $M(f(p))/p = \Theta(\log p)$  和  $M(f(p))/p = \Theta(p \log p)$  的情况进行类似的分析。尽管函数中的常数项也必须加以考虑, 我们通常还是可以认为  $M(f(p))/p$  的复杂度越低, 并行系统的可扩展性就越好。



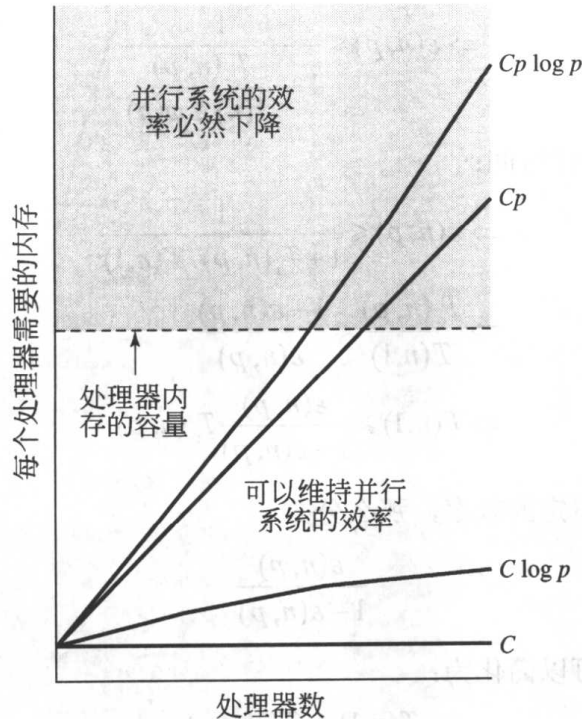


图 7.4 处理器个数增加时, 保持效率的方法是增加问题的规模。问题的最大规模受到可用内存容量的限制。可用内存容量是处理器个数的线性函数。从等效关系出发, 并考虑内存的需求为问题规模的函数, 我们可以确定为了维持效率, 每处理器所需的内存容量是如何作为  $p$  的函数增加的。此函数的复杂度越低, 并行系统就越可扩展

### 例 1 归约

第 3 章我们开发了并行归约算法。串行归约的计算复杂度是  $\Theta(n)$ 。归约步骤的复杂度为  $\Theta(\log p)$ , 每个都处理器参与了此步骤, 因此  $T_o(n, p) = \Theta(p \log p)$ 。 $\Theta$  记号忽略了常数, 但是我们假设其会被包含在效率常数  $C$  中。

因此归约算法的等效关系为:

$$n \geq Cp \log p$$

串行算法归约  $n$  个值, 因此  $M(n) = n$ , 所以:

$$M(Cp \log p) / p = Cp \log p / p = C \log p$$

我们通过思考来确认这个结果的意义。如果我们在  $p$  个处理器上对  $n$  个值进行求和归约, 每个处理器将  $n/p$  个值相加, 然后参加一个有  $\lceil \log p \rceil$  步的归约。如果我们将处理器个数和  $n$  的值都加倍, 每个处理器所需处理的值的个数仍然是  $n/p$  个。因此, 每个处理器用于求和的时间是相同的, 但是执行归约所需的步数从  $\lceil \log p \rceil$  增加到  $\lceil \log(2p) \rceil$ , 效率略微有所下降。为了保持效率不变, 我们在加倍处理器个数的同时必须把  $n$  的值增加到原来的 2 倍以上。可扩展性函数确认了我们的分析结果, 指出每处理器上的问题规模必须按  $\Theta(\log p)$  增加。

### 例 2 Floyd 算法

让我们确定在第 6 章开发的并行 Floyd 算法的等效函数。串行 Floyd 算法的复杂度为  $\Theta(n^3)$ 。在执行并行算法的时候,  $i$  个处理器中的每个处理器都花费  $\Theta(n^2 \log p)$  进行通信。因此等效关系为:

$$n^3 \geq C(pn^2 \log p) \Rightarrow n \geq Cp \log p$$

看起来这和我们在前一个例子得到的关系是相同的,但是我们必须考虑问题规模  $n$  与内存需求之间的关系。在 Floyd 算法中问题规模  $n$  所需要的内存容量为  $n^2$ ; 即  $M(n)=n^2$ 。此系统的可扩展性函数:

$$M(Cp \log p)/p = C^2 p^2 \log^2 p / p = C^2 p \log^2 p$$

与并行归约相比,此并行系统的可扩展性较差。

### 例3 有限差分方法

考虑解偏微分方程的有限差分法的并行算法(我们将在第13章详细讨论这一算法)。问题可以用  $n \times n$  的网格表示。每个处理器负责大小为  $(n/\sqrt{p}) \times (n/\sqrt{p})$  子网格,如图7.5所示。在算法的每次迭代中每个处理器发送边界上的值到其4个相邻处理器;这些通信操作所需的时间为每次迭代  $\Theta(n/\sqrt{p})$ 。

此问题的串行算法的时间复杂度为每次迭代  $\Theta(n^2)$ 。

此并行系统的等效关系为:

$$n^3 \geq Cp(n\sqrt{p}) \Rightarrow n \geq C\sqrt{p}$$

当问题规模为  $n$  时,网格中有  $n^2$  个元素,因此  $M(n)=n^2$  并且:

$$M(C\sqrt{p})/p = (C\sqrt{p})^2 / p = C^2 p / p = C^2$$

可扩展性函数是  $\Theta(1)$ , 即此并行系统是完全可扩展的。

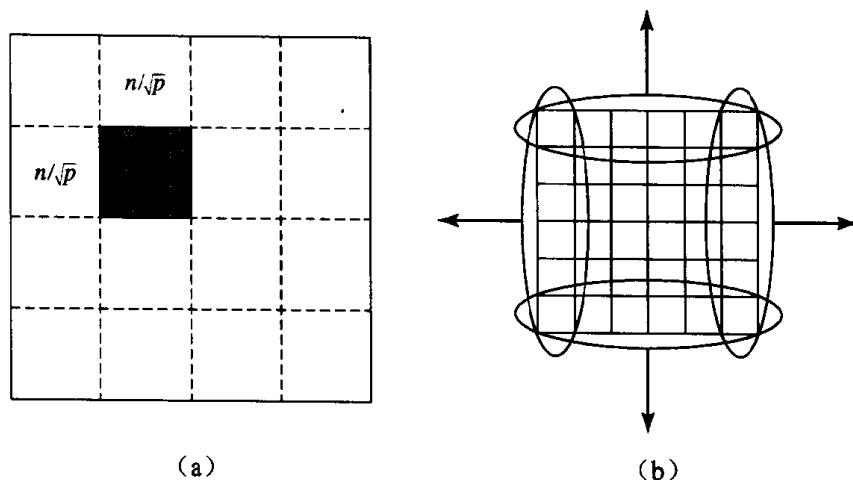


图7.5 划分并行有限差分算法。(a) 每个进程负责  $n \times n$  矩阵中  $(n \times \sqrt{p}) \times (n \times \sqrt{p})$  的数据块。  
(b) 在每次迭代中每个进程发送  $n/\sqrt{p}$  个边界值到其相邻的4个进程

## 7.7 本章小结

并行计算的目的是使用  $p$  个处理器执行程序,并获得比在单个处理器上高  $p$  倍的性能。顺序执行时间与并行执行时间之比称作加速比。

加速比=串行执行时间/并行执行时间

并行计算的效率(也称作处理器使用率)是加速比除以处理器数:

效率=加速比/ $p$

为了达到加速比  $p$ , 并行执行时间必须是串行程序的  $1/p$ 。因为仅有  $p$  个处理器, 这意味着每个处理器必须分配到同样数量的工作, 所有处理器在并行执行全过程中都必须在执行程序, 而且算法在并行化过程中不能引入额外的操作。换句话说, 当且仅当利用率是 100% 的时候加速比才等于  $p$ 。在实际中, 这几乎不可能实现。

为什么? 首先, 通常算法总有的某些部分无法在多个处理器上执行, 称作串行代码, 这使得我们无法让所有处理器总处于工作状态。

其次, 并行程序总需要处理器间的某些交互。这些通信操作并不包括在串行程序中。因此, 这些操作是算法并行化过程中引入的额外操作。

我们推导出了关于加速比的通用公式, 该公式包含了计算中内在的串行部分, 计算的并行部分以及并行开销 (通信操作和冗余计算)。我们同样讨论了分析并行程序性能的 4 个指标。

第 1 个指标, Amdahl 定律, 是向前看的定律。该定律依赖于对串行程序的评估来预测在并行计算系统上执行程序以加速程序中可并行部分的执行时, 所能达到的加速比上限。

第 2 个指标, Gustafson-Baris 定律, 是向后看的定律。该定律依赖于对并行程序的测试以预测程序在具有足够内存的串行处理器上的执行时间。由于处理器个数增加时可以增加问题的规模, 我们称 Gustafson-Baris 定律提供了比例加速比的估算。

第 3 个指标, Karp-Flat 量度, 考察了并行程序在解决固定规模问题时加速比。实验确定的串行比例可以用于预测程序在更多处理器上的性能。

第 4 个也是最后一个指标, 等效量度, 用于确定并行系统的可扩展性。在处理器个数增加时, 如果增加问题规模可以保持并行效率, 则该并行系统是完全可扩展的。由等效关系中推导出的可扩展性函数, 确定了要维持相同效率, 问题规模作为处理器个数的函数应该如何增长。

## 7.8 主要术语

Amdahl effect

Amdahl's Law

efficiency

experimentally determined serial fraction

Gustafson-Barsis's Law

isoefficiency relation

Karp-Flatt metric

parallel system

scalability

scalability function

scaled speedup

speedup

Amdahl 效应

Amdahl 定律

效率

实验确定的串行比例

Gustafson-Barsis 定律

等效关系

Karp-Flatt 量度

并行系统

可扩展性

可扩展性函数

比例加速比

加速比

## 7.9 参考文献

Gustafson, Montry 和 Benner【47】的文章不仅引入了比例加速比的概念,还首次报道了超过 1000 的比例加速比。该文章提供了在 MPP 系统(1024 个 CPU 的 nCUBE 多计算机)上获得最大加速比的方法。作者们因为此项工作而荣获 Gordon Bell 奖和 Karp 奖。

在 Grama 等人的著作 Introduction to Parallel Computing 中可以找到对等效量度的详细介绍【44】。请注意,在本书中没有采用他们对问题规模的定义。Grama 等人将问题规模定义为“在单个处理单元上最好的串行程序解决问题所需的基本计算步骤数”。换句话说,他们认为“问题规模”就是“串行执行时间”。我觉得这个定义与我们的直觉不符,因此在本书中没有采用。

## 7.10 练习题

7.1 使用 7.2 节中加速比的定义,证明存在  $p_0$ , 使得  $p > p_0 \Rightarrow \psi(n, p) < \psi(n, p_0)$ 。假定  $\kappa(n, p) = C \log p$ 。

7.2 由 7.2 节中效率的定义,证明  $p' > p \Rightarrow \varepsilon(n, p') \leq \varepsilon(n, p)$ 。

7.3 估计第 3.5 节中开发的并行归约算法在 1, 2, ..., 16 个处理器上可达到的加速比。假设  $n = 1\,000\,000$ ,  $\lambda = 100\mu s$ 。

7.4 对一个串行程序的测试 95% 的执行时间花费在一个可以并行化的函数中。程序的并行版本在 10 个处理器上执行所能达到的最大加速比是多少?

7.5 一个并行化程序 6% 的时间花费在单个处理器上执行的 I/O 函数中,要达到加速比 10,至少需要多少个处理器?

7.6 为了让一个并行程序达到 50 的加速比,能够用于执行其中内在串行部分的最大执行时间比例是多少?

7.7 Shaunna 的并行程序在 10 个处理器上达到了加速比 9。程序中内在串行操作在计算中所占的最大比例是多少?

7.8 Bradon 的并行程序在 16 个处理器上执行了 242 秒。通过测试,他发现有 9 秒钟花在单个处理器的初始化和退出上,另外 233 秒中所有处理器都处于工作状态。Brandon 的程序的加速比是多少?

7.9 Courtney 在 40 个处理器上测试了一个并行程序。她发现 99% 的时间用于执行并行代码,她的程序的加速比是多少?

7.10 6 个并行程序的执行时间,用 I-VI 表示,在 1-8 个处理器上进行了测试。下标表示了各个程序达到的加速比:

处理器数	加速比					
	I	II	III	IV	V	VI
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.67	1.89	1.89	1.96	1.74	1.94
3	2.14	2.63	2.68	2.88	2.30	2.82
4	2.50	3.23	3.39	3.67	2.74	3.65
5	2.78	3.68	4.03	4.46	3.09	4.42
6	3.00	4.00	4.62	5.22	3.38	5.15
7	3.18	4.22	5.15	5.93	3.62	5.84
8	3.33	4.35	5.63	6.25	3.81	6.50

对其中的每个程序, 选出最适合描述其在 16 个处理器上性能的陈述:

(a) 在 16 个处理器上的加速比至少比 8 个处理器上的加速比高 40%。

(b) 由于程序中的串行程序比例很大, 在 16 个处理器上的加速比至少比 8 个处理器上的加速比低 40%。

(c) 由于处理器增加时开销也会增大, 在 16 个处理器上的加速比至少比 8 个处理器上的加速比低 40%。

7.11 Amdahl 定律和 Gustafson-Barsis 定律都是从同一个加速比公式中推导出来的。但是当增加处理器个数  $p$  的时候, Amdahl 定律所预测的最大加速比收敛于  $1/f$ , 但 Gustafson-Barsis 定律预测的加速比没有上界, 请解释原因。

7.12 给定一个问题和你所需要的所有处理器, 你是否总能在给定的时间限制内解决问题?

7.13 让  $n \geq f(p)$  代表并行系统的等效关系,  $M(n)$  代表存储规模为  $n$  的问题所需的内存容量。使用可扩展性函数, 按可扩展性从高到低的顺序将下列并行系统排序。

(a)  $f(p) = Cp$  而且  $M(n) = n^2$

(b)  $f(p) = C\sqrt{p} \log p$  而且  $M(n) = n^2$

(c)  $f(p) = C\sqrt{p}$  而且  $M(n) = n^2$

(d)  $f(p) = Cp \log p$  而且  $M(n) = n^2$

(e)  $f(p) = Cp$  而且  $M(n) = n$

(f)  $f(p) = pC$  而且  $M(n) = n$ , 假设  $1 < C < 2$ 。

(g)  $f(p) = pC$  而且  $M(n) = n$ , 假设  $C > 2$ 。

# 第 8 章 矩阵向量乘法

Anarchy, anarchy! Show m a greater evil!

This is why cities tumble and the great housed rain down,

This is what scatters armies!

Sophocles, Antigone

## 8.1 概 述

矩阵向量乘法在许多解决实际问题中都有应用。例如，许多求解线性方程的迭代算法就依赖于矩阵向量乘法，第 12 章中的共轭梯度法就是这样一个算法。

神经网络也是矩阵向量乘法的一个典型应用。神经网络可广泛地应用于诸如手写识别、石油勘探、航班座位分配和信用卡检测【114】等许多领域。要得到一个  $k$  级神经网络的输出，最直接的方法就是完成  $k-1$  次矩阵向量乘。而且，神经网络的主要训练方法是反向传播算法，而这个算法的核心也是矩阵向量乘法【98】。

为了完成稠密矩阵与向量相乘的算法，在本章中我们将设计、分析并实现三个 MPI 程序并对它们进行测试。这三个设计基于三种不同的在 MPI 进程间分解矩阵及向量中元素的方式。这些不同的数据分解方式导致了进程间的通信模式不尽相同，这也就是说我们将使用不同的 MPI 函数来分别进行这些程序中所涉及的数据传递。所以这三个程序两两差别都很大。

在开发这三个程序的过程中，我们将逐渐引入下面 4 个功能强大的 MPI 通信函数：

- MPI\_Allgatherv, 数据全收集函数，不同的进程可以提供不同数量的元素；
- MPI\_Scatterv, 散发数据操作，不同的进程可以获得不同数目的元素；
- MPI\_Gatherv, 数据收集操作，从不同进程收集来的元素的个数可以不同；
- MPI\_Alltoall, 全交换操作，在所有进程间交换数据元素。

我们还用到了 5 个支持面向格状结构的通信域：

- MPI\_Dims\_create, 为平衡的笛卡儿进程网格创建新的维度；
- MPI\_Cart\_create, 创建一个笛卡儿拓扑通信域；
- MPI\_Cart\_coords, 返回笛卡儿进程网格中某个进程的坐标；
- MPI\_Cart\_rank, 返回笛卡儿进程网格中位于某个坐标的进程的进程号；
- MPI\_Comm\_split, 将一个通信域中的进程划分为一个或多个小组。

## 8.2 串 行 算 法

图 8.1 所示是向量和矩阵相乘的串行算法。又如图 8.2 所示，矩阵向量乘法是一系列

简单内积 (或点积) 的计算。两个  $n$  维向量的内积需要  $n$  次乘法和  $n-1$  次加法, 算法复杂度为  $\Theta(n)$ 。矩阵向量乘法需要完成  $m$  个内积计算, 所以算法复杂度为  $\Theta(mn)$ 。如果矩阵是方阵, 那么复杂度就变为  $\Theta(n^2)$ 。

输入:  $a[0..m-1, 0..n-1]$ —— $m \times n$  的矩阵

$b[0..n-1]$ —— $n$  维向量

输出:  $c[0..m-1]$ —— $m$  维向量

for  $i \leftarrow 0$  to  $m-1$

$c[i] \leftarrow 0$

for  $j \leftarrow 0$  to  $n-1$

$c[i] \leftarrow c[i] + a[i, j] \times b[j]$

endfor

endfor

图 8.1 串行矩阵向量乘法

$A$		$b$		$c$
2	1	3	4	0
5	-1	2	-2	4
0	3	4	1	2
2	3	1	-3	0

$\times$

3
1
4
0
3

$=$

19
34
25
13

图 8.2 矩阵向量乘法可被看作是一系列内积 (点积) 操作

例如,  $c_1 = 5 \times 3 + (-1) \times 1 + 2 \times 4 + (-2) \times 0 + 4 \times 3 = 34$

## 8.3 数据分解方式

我们使用域分解的策略开发该并程序。我们有很多种方法可以分解、聚合并映射矩阵和向量中的元素。不同数据分解方式会产生不同的并行算法。

我们有三种较为直接的分解矩阵的方法: 按行分解、按列分解和棋盘式分解, 如图 8.3 所示。假设待分解矩阵  $A$  的大小为  $m \times n$ 。我们已经见过按行分解的方法了: 在第 6 章 Floyd 算法的实现中, 我们通过这种方法在进程间分配矩阵的元素。在这种分解方法中,  $p$  个进程的每一个都会负责矩阵中相邻的  $\lfloor m/p \rfloor$  或  $\lceil m/p \rceil$  行元素。

按列分解与此类似, 只不过是矩阵按照列序号进行分解。 $p$  个进程的每一个都会负责矩阵中相邻的  $\lfloor n/p \rfloor$  或  $\lceil n/p \rceil$  列元素。

在棋盘式分解方式中, 所有进程构成一个虚拟网格, 矩阵按照按照这个网格分为二维的数据块。假设  $p$  个进程构成一个  $r$  行  $c$  列的网格, 则每个进程至多将负责  $\lceil m/r \rceil$  和  $\lceil n/c \rceil$  列的矩阵元素。

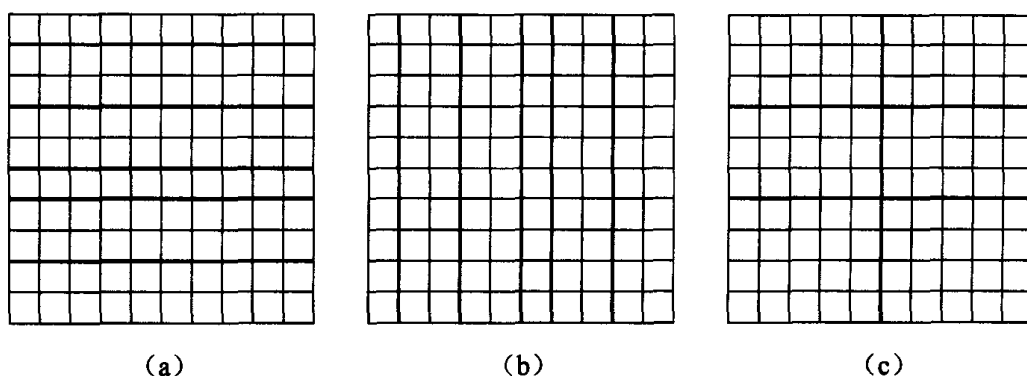


图 8.3 二维矩阵的三种分解方式。在这些例子中，一个  $10 \times 10$  的矩阵被分解到 6 个进程中  
(a) 按行分解；(b) 按列分解；(c) 棋盘式分解（进程被组织为一个  $3 \times 2$  的虚拟网格）

针对向量  $b$  和  $c$ ，我们有两种简单的分解方法。一是将向量元素复制，也就是每个进程都有此向量的一份副本；一是向量元素可以分配到几个或者所有的进程中去。将含有  $n$  个元素的向量块进行分解， $p$  个进程中的每一个都会负责原向量中相邻的  $\lfloor n/p \rfloor$  或  $\lceil n/p \rceil$  个元素。

为什么一个任务可以保存整个向量  $b$  和  $c$ ，却不能保存整个矩阵  $A$  呢？为了简化我们的讨论，假设  $m=n$ 。向量  $b$  和  $c$  只有  $n$  个元素，与矩阵  $A$  一行或一列的元素数目相同。若每个进程保存  $A$  的一行或一列和  $b$  与  $c$  的一个元素，其存储的量级为  $\Theta(n)$ 。若每个任务保存  $A$  的一行或一列和  $b$  与  $c$  的所有元素，存储的量级仍为  $\Theta(n)$ 。所以，不论向量是被复制到各进程还是被分解到各任务，存储的复杂度是相同的。

由于矩阵有三种分解方法，向量有两种分解方法，所以一共有六种组合。本章我们研究其中三种情况：(1) 矩阵按行分解，向量复制；(2) 矩阵按列分解，向量分块；(3) 矩阵棋盘式分解，向量分块并且只存在于进程网格的第一列进程中。

## 8.4 矩阵按行分解

### 8.4.1 设计与分析

本节我们将开发一个并行矩阵向量相乘算法。这个算法基于域分解，每个元任务与矩阵  $A$  的一行相关联。向量  $b$  和  $c$  被复制到所有的进程中。图 8.4 是此算法的一个高阶视图。为了计算内积，每个进程都需要一个行向量和一个列向量。任务  $i$  中有  $A$  的第  $i$  行和  $b$  的副本，这是用于完成内积计算的所有因素。完成内积计算后，任务  $i$  就得到了向量  $c$  的第  $i$  个元素。当然，不光是向量  $b$ ，向量  $c$  也需要被复制到各个进程中去。所以我们需要进行一次全收集操作以使各个任务都含有  $c$  中所有的元素，这个操作完毕后算法结束。

在稠密矩阵向量乘法中，完成内积的计算步数是相同的。所以我们可以将连续的几行所对应的元任务进行聚合并与一个进程相对应，这样就形成了矩阵按行分解，如图 8.3 (a) 所示。



如图 8.4 所示, 在内积计算结束时, 每个任务只计算出结果向量中的一个元素。如果采用按行分解, 每个进程 (包含几个元任务) 会得到结果向量中的几个元素。

如果  $m=n$ , 线性矩阵向量乘法的时间复杂度便为  $\Theta(n^2)$ 。下面来考察并行算法的复杂度。每个进程完成它所分配到  $A$  的那部分元素与向量  $b$  的乘法。任何进程分得的部分都不会超过  $\lceil n/p \rceil$  行。所以并行算法的乘法部分的复杂度是  $\Theta(n^2/p)$ 。

在第 3 章我们看到, 在一个高效的全收集通信中, 每个进程发送  $\lceil \log p \rceil$  个消息; 所传送的元素总数是  $n(p-1)/p$ , 其中  $p$  是 2 的幂。所以并行算法的通信复杂度是  $\Theta(\log p + n)$ 。

综合考虑算法的计算部分和后面的通信部分, 并行矩阵向量乘法的复杂度是  $\Theta(n^2/p + n + \log p)$ 。

下面来考察这个算法的等效率特性。串行算法的时间复杂度为  $\Theta(n^2)$ 。并行算法惟一的开销就是全收集操作。当  $n$  非常大的时候, 在全收集操作中消耗的数据传输延时将远远超过消息延迟。鉴于这个原因我们将通信的复杂度简化至  $\Theta(n)$ 。所以按行分解的矩阵向量乘法算法其等效率函数是:

$$n^2 \geq Cpn \Rightarrow n \geq Cp$$

如果问题规模为  $n$ , 矩阵有  $n^2$  个元素, 那么内存使用函数  $M(n) = n^2$ 。并行算法的可扩展函数为:

$$M(Cp)/p = C^2 p^2 / p = C^2 p$$

为了保持不变的效率, 每个处理器所使用的内存量必须随着处理器的数目的增加线性地增大。可见, 这种算法的可扩展性不好。

## 8.4.2 复制分块的向量

某个进程完成它所负责的那部分的矩阵向量相乘计算后便得到了结果向量  $c$  中的一块。我们必须把这个分块的向量传给到被复制的  $c$  向量中, 如图 8.5 所示。

下面考虑如何才能完成这个传递。首先, 每个进程必须分配一块内存来存放整个向量, 而不仅仅是它的一部分。所分配内存空间的大小依赖于元素的类型, 例如, 字符、整数、浮点数或双精度浮点数。第二, 进程必须将这些分块组合为一个完整的向量并共享组合的结果。幸好 MPI 库中有完成组合操作的函数。

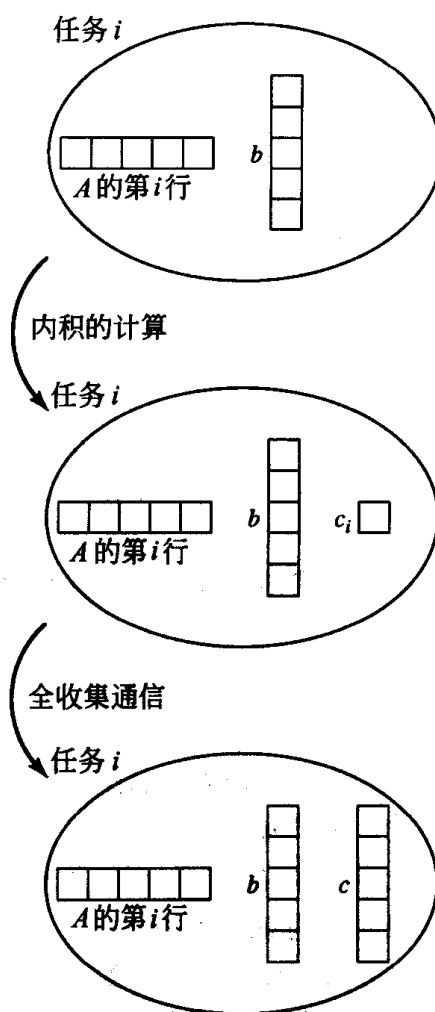


图 8.4 在我们所选择的域分解方式中, 每个任务都有矩阵的一行和输入向量的完整副本。一次内积计算将得到结果向量  $c$  中的一个元素。复制向量  $c$  需要一次全收集操作

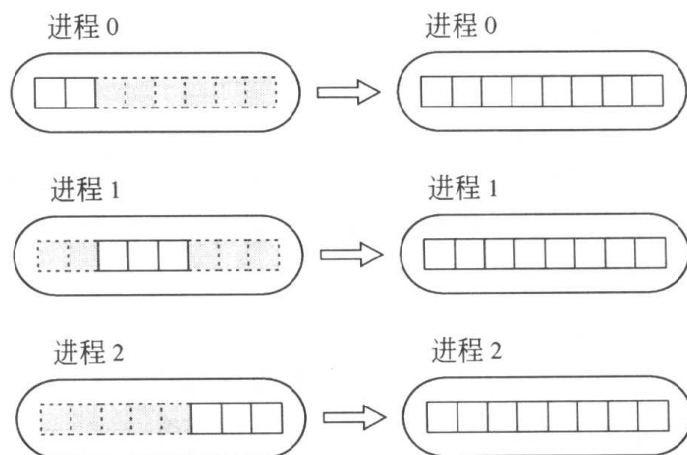


图 8.5 将向量的分块转换为被复制向量，其中该向量的分块分布在各个进程中。向量中的每个元素只存在于一个进程中。当向量被复制后情况则恰恰相反，每个进程将拥有完整的被复制向量

### 8.4.3 函数 MPI\_Allgatherv

一个全收集通信可以连接分布在一组进程中的向量数据块，并把结果向量复制至所有的进程。这个函数就是 MPI\_Allgatherv，如图 8.6 所示。

如果从每个进程收集同样数目的元素，较简单的 MPI\_Allgather 函数就非常合适。但在向量块分解方式中，只有当向量中的元素总数是进程总数的整数倍时，每个进程分到的向量元素才是相等的。显然这点不可能总能得到保证，所以我们使用 MPI\_Allgatherv。

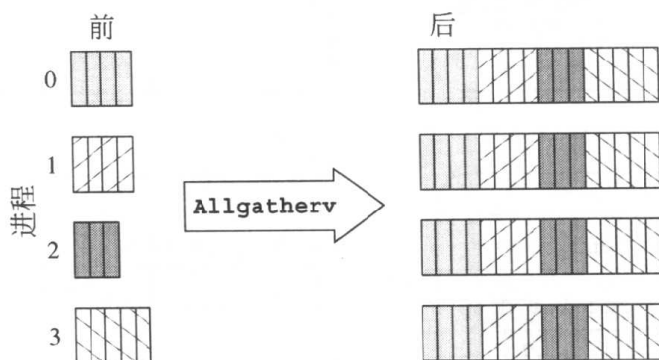


图 8.6 函数 MPI\_Allgatherv 负责从一个通信域的所有进程中收集数据项，并将其连接起来。

如果从每个进程收集的数据个数相同，也可以用稍微简单一点的 MPI\_Allgather 函数

函数声明：

```
int MPI_Allgatherv (void* send_buffer, int send_cnt,
    MPI_Datatype send_type, void* receive_buffer,
    int* receive_cnt, int* receive_disp,
    MPI_Datatype receive_type, MPI_Comm communicator)
```

除了第四个参数外其他参数都是输入参数：

- `send_buffer` 此进程要发送的数据的起始地址；
- `send_cnt` 此进程要发送的数据的个数；
- `receive_cnt` 包含要从每个进程（包括自身）接收的数据个数的数组；
- `receive_disp` 从每个进程接收的数据项在缓冲区中的偏移量；
- `receive_type` 待接收数据的数据类型；
- `communicator` 本操作所在通信域。

第四个参数，`receive_buffer`，是用来存放所要收集到的元素的缓冲区的起始地址。

图 8.7 描述了 `MPI_Allgatherv` 的工作原理。每个进程将 `send_cnt` 设置为它所负责的元素数目。数组 `receive_cnt` 由每个进程要接收的元素个数所组成，每个进程中该值是相同的。在这个例子中，每个进程按照进程的编号顺序对元素进行拼接，所以数组 `receive_disp` 中的值也是相同的。

`MPI_Allgatherv` 需要传递两个数组，每个都会涉及到所有的进程。第一个数组指出每个进程所贡献的元素个数，第二个数组指出这些元素在结果数组中的位置。

我们经常遇到被分块映射并顺序连接的数组。我们可以写一个函数来构造这两个数组，称之为 `create_mixed_xfer_arrays`，见附录 B。

此时，我们可以编写一个函数以完成分块向量到复制向量的传递，这也是矩阵向量乘法算法的最后一步。这个函数，即函数 `replicate_block_vector`，见附录 B。

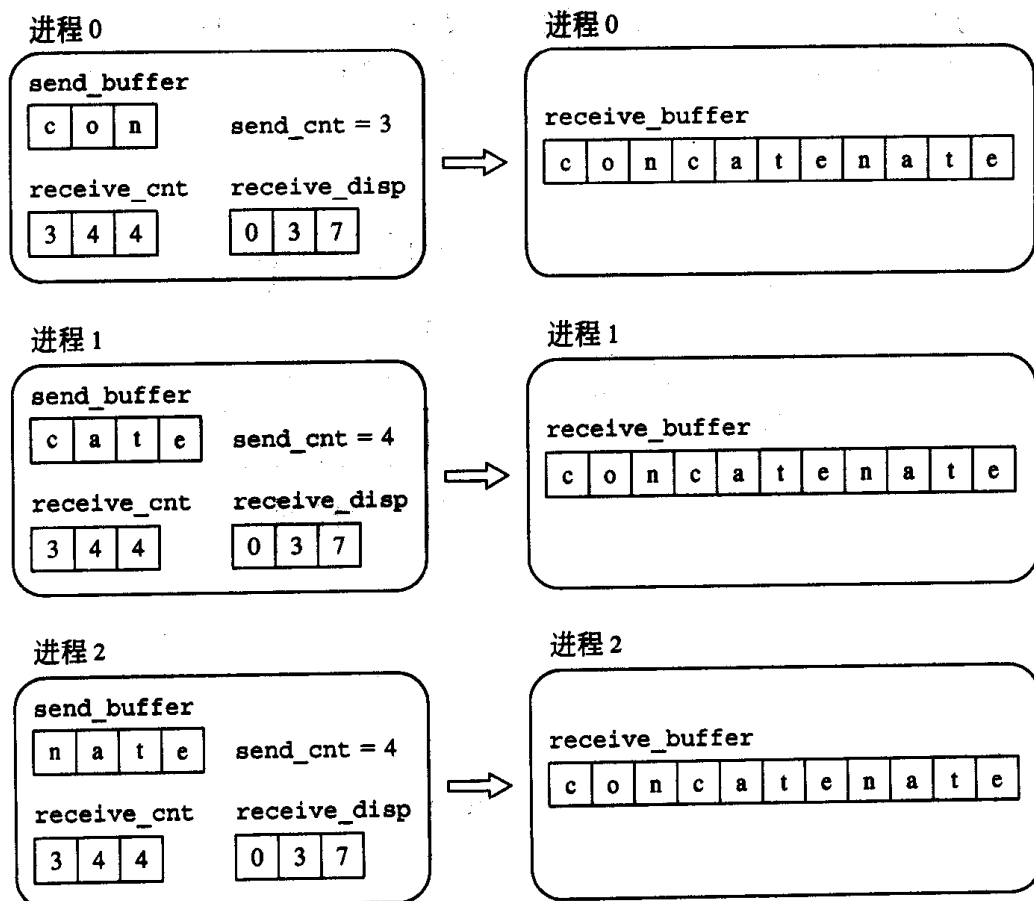


图 8.7 `MPI_Allgatherv` 直接进行连结时，进程初始化 `send_cnt`、`receive_cnt` 和 `receive_disp` 的实例

### 8.4.4 被复制向量的输入/输出

我们需要一个函数从文件中读取待复制的数组。假设这个文件由 `fwrite` 创建，并可由 `fread` 读取。文件开始是开头为整数  $n$ ，它表示向量中元素的个数，之后是  $n$  个元素。

进程  $p-1$  尝试打开并读取数据文件。如果可以打开，它就读取  $n$  并广播给其他进程。如果打开失败，就给其他进程广播 0，然后所有进程结束计算。如果打开成功，所有进程分配空间以存放此向量。之后进程  $p-1$  读取向量并广播给其他进程。`read_replicated_vector` 的源码见附录 B。

从并行编程的角度来看，打印复制向量的工作很简单。为了不让标准输出变得混乱，通常由一个进程完成所有的打印工作。因为每个进程都有此向量的一份副本，我们只需保证只有一个进程执行了 `printf` 即可。函数 `print_replicated_vector` 的源码见附录 B。

### 8.4.5 编写并程序序

所有的准备工作都做好了，现在我们可以编写矩阵向量乘法的并程序序了。再回顾一下图 8.4，图中总括了这个算法的主要步骤。完整的 C 代码如图 8.8 所示。

程序开头包含了标准头文件，另外还包含了我们开发的工具函数头文件 `MyMPI.h`。

```
/*
 * Matrix-vector multiplication, Version 1
 */
#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"
/* Change these two definitions when the matrix and vector
   element types change */
typedef double dtype;
#define mpitype MPI_DOUBLE
int main (int argc, char *argv[]) {
    dtype **a; /* First factor, a matrix */
    dtype *b; /* Second factor, a vector */
    dtype *c_block; /* Partial product vector */
    dtype *c; /* Replicated product vector */
    dtype *storage; /* Matrix elements stored here */
    int i, j; /* Loop indices */
    int id; /* Process ID number */
    int m; /* Rows in matrix */
    int n; /* Columns in matrix */
    int nprime; /* Elements in vector */
    int p; /* Number of processes */
    int rows; /* Number of rows on this process */
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
```

```

MPI_Comm_size (MPI_COMM_WORLD, &p);
read_row_striped_matrix (argv[1], (void *) &a,
    (void *) &storage, mpitype, &m, &n, MPI_COMM_WORLD);
rows = BLOCK_SIZE (id,p,m);
print_row_striped_matrix ((void **) a, mpitype, m, n, MPI_COMM_WORLD);
read_replicated_vector (argv[2], (void *) &b, mpitype,
    &nprime, MPI_COMM_WORLD);
print_replicated_vector (b, mpitype, nprime, MPI_COMM_WORLD);
c_block = (dtype *) malloc (rows * sizeof (dtype));
c = (dtype *) malloc (n * sizeof (dtype));
for (i = 0; i < rows; i++) {
    c_block[i] = 0.0;
    for (j = 0; j < n; j++)
        c_block[i] += a[i][j] * b[j];
}
replicate_block_vector (c_block, n, (void *) c, mpitype, MPI_COMM_WORLD);
print_replicated_vector (c, mpitype, n, MPI_COMM_WORLD);
MPI_Finalize ();
return 0;
}

```

图 8.8 矩阵向量乘法 (第 1 版)

为了保证在最小限度内编辑代码来实现改变矩阵和向量的类型, 我们使用 `dtype` 来表示矩阵和向量中数据的类型, 用 `mpitype` 作为 MPI 函数的类型标志符。在程序头部我们用 `typedef` 和一个宏定义确定 `dtype` 和 `mpitype` 的值。

MPI 初始化完毕后, 进程读取并打印矩阵  $A$  (见第 6 章), 同时读取并打印向量  $b$ 。

每个进程为它所分配到的结果向量  $c$  分配空间并完成内积计算。

此时, 每个进程都有一部分的  $c$ 。我们将  $c$  转化为一个完整向量并完成复制, 之后程序结束。

## 8.4.6 测试

下面我们推导一个适用于商用集群上并程序执行时间的表达式。令  $\chi$  表示内积计算循环中一次迭代所消耗的时间。我们可以将串行算法的执行时间除以  $n^2$  来得到  $\chi$ 。所以并行算法计算部分的执行时间是  $\chi n (n/p)$ 。

全收集归约操作中每个发送  $(\log p)$  个消息。每个消息的延迟为  $\lambda$ 。在全收集操作中总共传输的向量元素个数为  $n (2^{(\log p)} - 1) / 2^{(\log p)}$ 。每个向量元素是大小为 8 字节的双精度浮点数。所以全收集操作的执行时间为  $\lambda (\log p) + 8n ((2^{(\log p)} - 1) / 2^{(\log p)}) / \beta$ 。

测试集群配置: 450MHz PentiumII, 快速以太网参数:  $\chi=63.4\text{ns}$ ,  $\lambda=250\mu\text{s}$ ,  $\beta=106\text{bps}$ 。

表 8.1 中比较了矩阵向量乘法的实际执行时间和预测执行时间, 问题规模为 1000, 分别在 1, 2, ..., 8, 16 个处理器上进行测试。实际报告的时间是运行 100 次的平均时间。将总浮点操作次数  $(2n^2)$  除以执行时间, 再除以一百万得到 MFLOPS。此程序的加速情况如本章结尾图 8.20 所示。

表 8.1 按行块分解的矩阵向量乘法的预测时间和执行时间的比较。  
其中矩阵大小为  $1000 \times 1000$ ，向量中有 1000 个元素

处理器数	预期时间	实际时间	加速比	Mflops
1	0.0634	0.0634	1.00	31.6
2	0.0324	0.0327	1.94	61.2
3	0.0223	0.0227	2.79	88.1
4	0.0170	0.0178	3.56	112.4
5	0.0141	0.0152	4.16	131.6
6	0.0120	0.0133	4.76	150.4
7	0.0105	0.0122	5.19	163.9
8	0.0094	0.0111	5.70	180.2
16	0.0057	0.0072	8.79	277.8

## 8.5 矩阵按列分解

### 8.5.1 设计与分析

本节我们将设计另一种矩阵-向量相乘算法。假设任务  $i$  分配到矩阵  $A$  的第  $i$  列和向量  $b$  与  $c$  的第  $i$  个元素。图 8.9 是这个并行算法的结构。

计算过程从每个任务将它所拥有的  $A$  的列与  $b_i$  相乘开始，这将得到一个向量作为中间结果。计算任务结束时，第  $i$  个任务将只需保留一个元素  $c_i$ 。因而我们需要进行一次全交换通信：任务  $i$  所得到的部分结果元素  $j$  必须传递给任务  $j$ 。此时，每个任务  $i$  都有得到  $c_i$  的  $n$  个部分结果。

因为每个任务的计算和通信特征都是相同的，所以可以将它们组合成含有相同列数（或多一列或少一列）的更大的任务来保证负载平衡。所以我们将初始任务组合成为  $p$  个元任务，并把每个元任务映射为一个进程。

在前一节我们将矩阵  $A$  按行分解并分配给了各个进程，这叫做按行分解。现在我们使用按列分解的方法，将矩阵  $A$  中相邻的连续几列进行聚合，如图 8.3 (b) 所示。

下面分析这种算法的复杂度。假设  $m=n$ ，每个进程将它所分配到的部分矩阵  $A$  和部分向量  $b$  相乘。任何进程的计算量都不会超过  $A$  的  $\lceil n/p \rceil$  列和  $b$  的元素。因此开始的乘法部分的时间复杂度为  $\Theta(n(n/p)) = \Theta(n^2/p)$ 。全收集过程完成后，每个进程将从其他进程收集到的部分向量求和。一共有  $p$  个部分向量，每个的长度不会超过  $\lceil n/p \rceil$ 。这一步的时间复杂度为  $\Theta(n)$ 。因此，整个计算的时间复杂度为  $\Theta(n^2/p)$ 。

利用超立方通信模式，一个全交换在  $(\log p)$  步之内完成。在每一步，每个进程都向它的协作者发送  $n/2$  个数据，同样它也从其协作者处接收  $n/2$  个数据。在全交换交换中发送和接收的元素总数为  $n$ 。所以通信复杂度为  $\Theta(n \log p)$ 。

另一种完成全交换的方式是由每个进程向其他  $p-1$  个进程发送消息。每个消息只包含目标进程期望从源进程获得的数据。这种方式需要发出的消息总数是  $p-1$ ，但由每个进程传递的元素数量均不大于  $n$ 。这个算法的时间复杂度为  $\Theta(p+n)$ 。

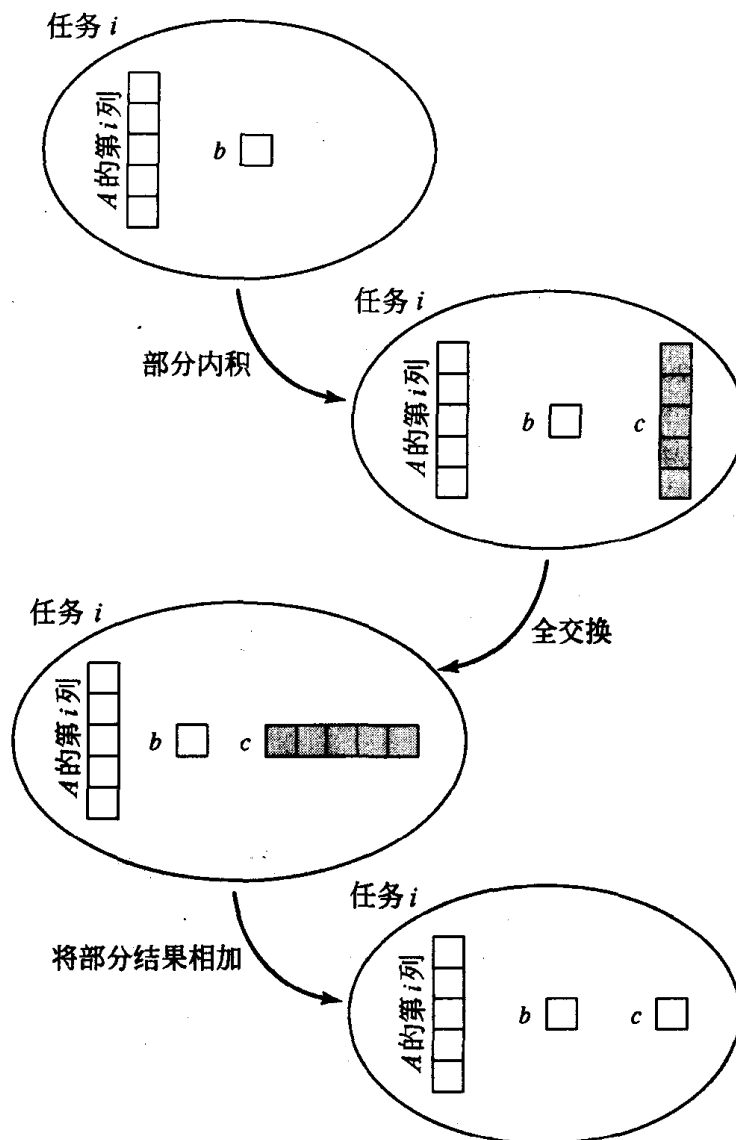


图 8.9 在本并行算法中, 每个任务负责矩阵中的一列和向量中的一个元素。  
通过全交换通信将各个数据项移动到相应的任务, 再将它们求和

综合此算法的计算阶段和全交换通信阶段, 它的总的复杂度为  $\Theta(n^2/p + n \log p)$  或  $\Theta(n^2/p + n + p)$ , 最终结果取决于采用哪种通信方式。

现在来考察这个并行算法的等效率特性。串行算法的时间复杂度是  $\Theta(n^2)$ 。并行算法的开销在于全交换操作。当  $n$  非常大的时候, 在全交换操作中消息传递的时间远远超过消息延迟。用第二种方法实现全交换, 时间复杂度为  $\Theta(n)$ , 所有进程都进行这一步。

所以基于按列分解的矩阵一向量相乘的并行算法, 其等效率性函数为:

$$n^2 \geq Cpn \Rightarrow n \geq Cp$$

这于我们使用按行分解得到的等效率函数是一样的。此算法的扩展性也不好, 为了保证效率不变, 内存的使用须随着处理器的增加而线性增大。

## 8.5.2 读取按列分解的矩阵

我们需要编写一个函数来读取矩阵数据文件 (在这个文件中, 矩阵按照行优先的规则

存放), 并将其按列分发到各个进程去。如果一个按行优先存放的矩阵有很多行, 而且要按照列块分解的方式进行分解, 那么分配到某个进程的数据在文件中是不连续的。实际上, 矩阵的每行都会被分发到所有的进程去。

我们仍然只让一个进程完成 I/O 操作, 如图 8.10 所示。第一步, 一个进程读取矩阵一行至一个临时缓冲区; 第二步, 这个进程分发缓冲区中的元素至所有进程。此后对矩阵的剩余行重复进行这两步操作。函数 `read_col_stripped_matrix` 的源码见附录 B。

函数 `read_col_stripped_matrix` 使用 MPI 库函数 `MPI_Scatterv` 在进程间散发行数据。下面我们进一步来研究一下这个函数。

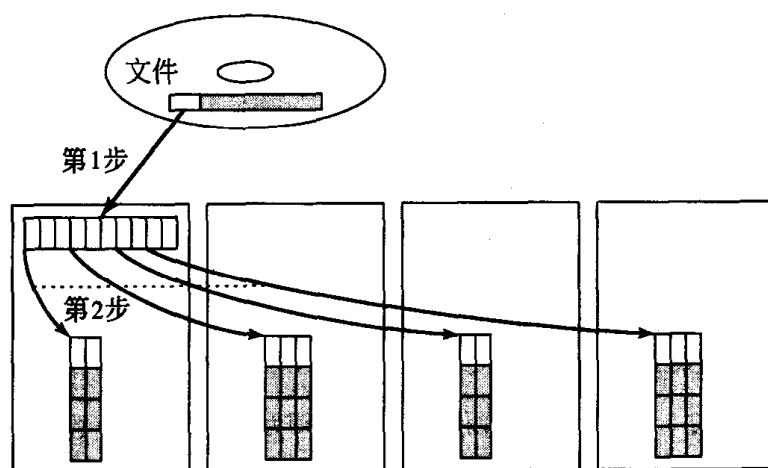


图 8.10 在按列分解算法中, 矩阵的每一行都分布在进程中。一个进程输入矩阵的一行 (第 1 步) 之后将其散发开来 (第 2 步)

### 8.5.3 函数 `MPI_Scatterv`

MPI 函数 `MPI_Scatterv`, 如图 8.11 所示, 通过一个根进程向在一个通信域内的所有进程 (包括自身) 分发一组连续的数据元素。

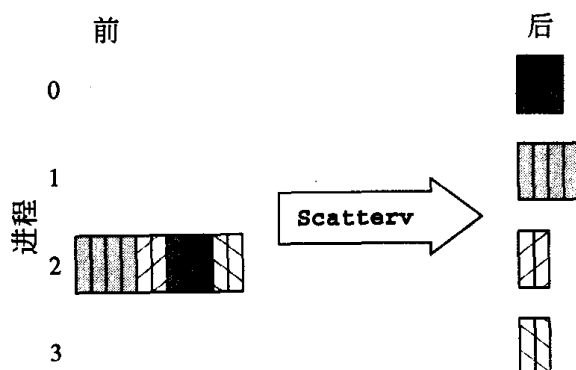


图 8.11 组通信函数 `MPI_Scatterv` 允许一个 MPI 进程将连续的数据项分解并把各个部分散发到通信域的其他进程去。如果向每个进程分发的数据项个数相同, 则相对简单的 `MPI_Scatter` 较为适合

`MPI_Scatterv` 的声明:



```
MPI_Scatterv(void *send_buffer, int* send_cnt,
             int* send_disp, MPI_Datatype send_type,
             void *recv_buffer, int recv_cnt,
             MPI_Datatype recv_type, int root, MPI_COMM communicator)
```

本函数有 9 个参数, 除第 5 个参数外全是输入变量:

- `send_buffer` 指向含有待分发元素缓冲区的指针。
- `send_cnt` 第  $i$  个元素是 `send_buffer` 中要发送到进程  $i$  的一连串数据的个数。
- `send_disp` 第  $i$  个元素是 `send_buffer` 中要发送到进程  $i$  的第一个元素在 `send_buffer` 中的偏移量。
- `send_type` `send_buffer` 中数据的类型。
- `recv_buffer` 指向本进程用于接收数据的缓冲区指针。
- `recv_cnt` 本进程要接收的数据个数。
- `recv_type` `recv_buffer` 的数据类型。
- `root` 分发数据进程的 ID;
- `communicator` 散发操作所在的通信域。

`MPI_Scatterv` 是一个组通信函数, 通信域中的所有进程都参与其执行。此函数要求所有进程初始化两个数组: 一个指出根进程向每个进程发送数据的个数, 一个指出散发数据在缓冲区中的偏移量。散发操作按照进程号的顺序进行: 进程 0 得到第一块, 进程 1 得到第二块, 以此类推。在收集操作中我们已经编写了 `create_mixed_xfer_arrays` 函数, 此处我们也可以直接使用。每个进程的元素数目和偏移量是相同的。

## 8.5.4 打印输出按列分块矩阵

我们需要设计一个函数完成按列分块矩阵的打印。为了保证数据以正确的顺序打印, 我们只允许一个进程完成所有的打印。为了打印一行, 此进程必须从其他所有进程收集该行的数据。所以此函数的数据流与 `read_col_stripped_matrix` 正好相反。函数 `print_col_stripped_matrix` 源码见附录 B。

函数 `print_col_stripped_matrix` 用到 MPI 库函数 `MPI_Gatherv` 来收集元素至进程 0, 然后进程 0 将其打印。下面我们来看看 `MPI_Gatherv`。

## 8.5.5 函数 `MPI_Gatherv`

MPI 组通信函数 `MPI_Gatherv` (如图 8.12 所示) 完成数据收集功能。

`MPI_Gatherv` 函数的声明:

```
MPI_Gatherv(void* send_buffer, int send_cnt,
            MPI_Datatype send_type, void *recv_buffer,
            int* recv_cnt, int* recv_disp, MPI_Datatype recv_type,
            int root, MPI_Comm communicator)
```

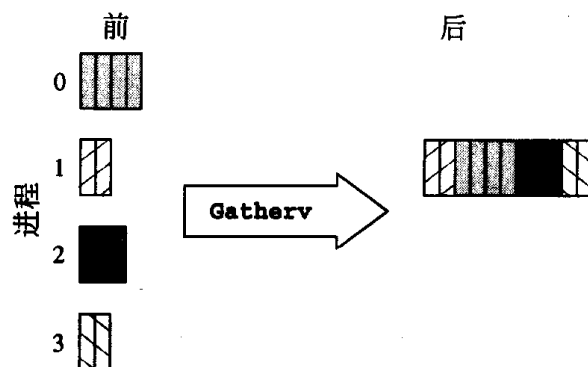


图 8.12 MPI\_Gatherv 可以将存储在通信域所有进程中的数据收集到某个进程中。

如果每个进程贡献的数据个数相同，可以使用较为简单的 MPI\_Gather 函数

此函数有 9 个参数，除了第 5 个参数外均为输入参数：

- send\_buffer 此进程被收集数据的起始地址；
- send\_cnt 本进程被收集的数据个数；
- send\_type send\_buffer 中数据的类型；
- recv\_buffer 根进程存放收集到数据的起始地址；
- recv\_cnt 这个数组的第  $i$  个元素表示从进程  $i$  收集的数据个数；
- recv\_disp 第  $i$  个元素是从进程  $i$  收集来数据的存放地址相对于 recv\_buffer 起始地址的偏移量；
- recv\_type recv\_buffer 的数据的类型；
- root 收集数据的根进程；
- communicator 收集操作所在的通信域。

### 8.5.6 分发中间结果

$m$  个内积操作得到了  $m$  个元素的结果向量  $c$ ：

$$c[0] = a[0,0]b[0] + a[0,1]b[1] + \cdots + a[0][n-1]b[n-1]$$

$$c[1] = a[1,0]b[0] + a[1,1]b[1] + \cdots + a[1][n-1]b[n-1]$$

...

$$c[m-1] = a[m-1,0]b[0] + a[m-1,1]b[1] + \cdots + a[m-1][n-1]b[n-1]$$

在域分解中，每个原始任务  $i$  分配到矩阵  $A$  的第  $i$  列和向量  $b$  的第  $i$  个元素。将列的每个元素与  $b_i$  相乘分别得到  $a[0,i]b[i]$ ,  $a[1,i]b[i]$ ,  $\cdots$ ,  $a[n-1,i]b[i]$ 。 $a[0,i]b[i]$  是组成  $c[0]$  的第  $i$  个元素， $a[1,i]b[i]$  是  $c[1]$  的第  $i$  个元素，依此类推。也就是说任务  $i$  进行的  $n$  个乘法所得到的  $n$  个元素不能相加得到内积。得到的第  $j$  个元素是用于计算  $c[j]$  的一部分 ( $0 \leq j < n$ )。

乘法完成之后，每个进程都要把它所不需要的  $n-1$  个结果分发到其他进程，同时向其他进程收集所需要的  $n-1$  个结果：这叫做全交换 (all-to-all exchange)。交换之后，任务  $i$  将它所有的  $n$  个元素（任务 0 生成的  $a[i,0]b[0]$ ，任务 1 生成的  $a[i,1]b[1]$ ）求和得到  $c[i]$ 。

在这种矩阵向量乘法的实现中，每个进程分配到  $A$  的某些列和  $b$  的对应元素。原则是

一样的：进程之间互相交换各自所不需要的和其所需要的数据。任务  $i$  收到  $\text{BLOCK\_SIZE}(i, p, n)$  个元素。交换之后，它有  $p$  个大小为  $\text{BLOCK\_SIZE}(i, p, n)$  的数组，将这些数组求和它便得到了  $c$  的一部分。

### 8.5.7 函数 MPI\_Alltoallv

MPI\_Alltoallv 可以完成在一个通信域的所有进程之间相互交换数据（如图 8.13 所示）。函数声明如下：

```
int MPI_Alltoallv(void *send_buffer, int *send_count,
                 int *send_displacement, MPI_Datatype send_type,
                 void *recv_buffer, int *recv_count,
                 int *recv_displacement, MPI_Datatype recv_type,
                 MPI_Comm communicator)
```

- send\_buffer 待交换数组的起始地址；
- send\_count 这是一个数组，它的第  $i$  个元素指定进程  $i$  的元素个数；
- send\_displacement 第  $i$  个元素为进程  $i$  的数据在 send\_buffer 的起始地址；
- send\_type send\_buffer 元素数据类型；
- recv\_buffer 接收数据（包括自己发送给自己的数据）缓冲区起始地址；
- recv\_count 这是一个数组，它的第  $i$  个元素表示本进程将要从进程  $i$  接收的数据个数；
- recv\_displacement 第  $i$  个元素表示从进程  $i$  接收的数据存入 recv\_buffer 的起始地址；
- recv\_type 在放入 recv\_buffer 前数据要被转换为此类型；
- communicator 指明参加全交换的进程集合。

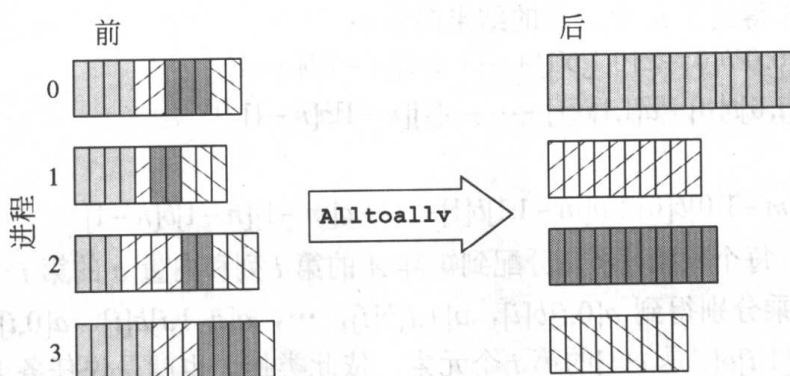


图 8.13 MPI\_Alltoallv 使通信域中的所有进程向所有进程收集数据。如果从任一进程传送到任意其他进程的数据个数均相同，则可以使用较为简单的 MPI\_Alltoall 函数

### 8.5.8 编写并行程序

现在我们已经可以编写第二个矩阵向量乘法的并程序了。源代码如图 8.14 所示。

```

/*
 * Matrix-vector multiplication, Version 2
 */
#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"
/* Change these two definitions when the matrix and vector
   element types change */

typedef double dtype;
#define mpitype MPI_DOUBLE

int main (int argc, char *argv[]) {
    dtype **a; /* The first factor, a matrix */
    dtype *b; /* The second factor, a vector */
    dtype *c; /* The product, a vector */
    dtype *c_part_out; /* Partial sums, sent */
    dtype *c_part_in; /* Partial sums, received */
    int *cnt_out; /* Elements sent to each proc */
    int *cnt_in; /* Elements received per proc */
    int *disp_out; /* Indices of sent elements */
    int *disp_in; /* Indices of received elements */
    int i, j; /* Loop indices */
    int id; /* Process ID number */
    int local_els; /* Cols of 'a' and elements of 'b' held by this process */
    int m; /* Rows in the matrix */
    int n; /* Columns in the matrix */
    int nprime; /* Size of the vector */
    int p; /* Number of processes */
    dtype *storage; /* This process's portion of 'a' */
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    read_col_stripped_matrix (argv[1], (void ***) &a,
        (void **) &storage, mpitype, &m, &n, MPI_COMM_WORLD);
    print_col_stripped_matrix ((void **) a, mpitype, m, n, MPI_COMM_WORLD);
    read_block_vector (argv[2], (void **) &b, mpitype,
        &nprime, MPI_COMM_WORLD);
    print_block_vector ((void *) b, mpitype, nprime, MPI_COMM_WORLD);

    /* Each process multiplies its columns of 'a' and vector
       'b', resulting in a partial sum of product 'c'. */
    c_part_out = (dtype *) my_malloc (id, n * sizeof (dtype));
    local_els = BLOCK_SIZE (id, p, n);
    for (i = 0; i < n; i++) {

```

```

        c_part_out[i] = 0.0;
        for (j = 0; j < local_els; j++)
            c_part_out[i] += a[i][j] * b[j];
    }
    create_mixed_xfer_arrays (id, p, n, &cnt_out, &disp_out);
    create_uniform_xfer_arrays (id, p, n, &cnt_in, &disp_in);
    c_part_in = (dtype*) my_malloc (id, p*local_els*sizeof (dtype));
    MPI_Alltoallv (c_part_out, cnt_out, disp_out, mpitype,
        c_part_in, cnt_in, disp_in, mpitype, MPI_COMM_WORLD);
    c = (dtype*) my_malloc (id, local_els * sizeof (dtype));
    for (i = 0; i < local_els; i++) {
        c[i] = 0.0;
        for (j = 0; j < p; j++)
            c[i] += c_part_in[i + j*local_els];
    }
    print_block_vector ((void *) c, mpitype, n, MPI_COMM_WORLD);
    MPI_Finalize ();
    return 0;
}

```

图 8.14 矩阵向量乘法并程序 (第 2 版)

MPI 的初始化完成之后, 调用函数 `read_col_striped_matrix` 从数据文件读取矩阵并分发到其他进程, 然后将矩阵打印输出。

类似地, 读取向量  $b$  并输出。

每个进程分配空间用来存储 `c_part_out`, 它是中间结果, 最终将被传递给其他进程。同样地, 要为 `c_part_in` 分配空间, 以接收从其他进程传递来的数据。

随后是真正的计算过程。每个进程用它拥有的部分矩阵 ( $n \times \text{local\_els}$ ) 乘以它的部分向量 (长度为 `local_els`), 得到长度为  $n$  的中间结果向量。

用于大小各异各个 `c_part_out`, 我们通过调用 `create_mixed_xfer_arrays` 来设置各自的数量和偏移量。相反, 所有用于输入的 `c_part_in` 大小都相同, 通过调用 `create_uniform_xfer_arrays` 来正确的初始化各自的数量和偏移量。最后由 `MPI_Alltoallv` 完成全交换通信, 将各个部分传递至目的地。

此时, 每个进程都有长度为 `local_els` 的  $n$  个数据。将其求和得到  $c$  的一个元素。

## 8.5.9 测试

下面我们来推导一个适用于计算商用集群上该并程序执行时间的表达式。如前, 令  $\chi$  表示内积计算循环中一次迭代所需要的时间。并行算法计算部分的时间是  $\chi n \lceil n/p \rceil$ 。

本算法要进行一次向量  $c$  部分结果的全交换操作。我们有两种方法实现全交换: 第一种是每个进程发送  $(\log p)$  个长度为  $n/2$  的消息, 总共传递的数据个数为  $n/2$ 。

第二种方法是每个进程向目的进程直接发送目的进程所需要的数据。每个进程需要发送  $p-1$  个消息，总共传递得数据个数为  $n(p-1)/p$ 。

当  $n$  较大的时候，消息的时延主要由消息传递所构成，所以第二种方法更有优势。令消息的延迟为  $\lambda$ ，传递一个字节耗时  $1/\beta$ ，完成一次双精度浮点数全收集将耗时  $(p-1)(\lambda+8n/(p\beta))$ 。

测试集群配置：450MHz PentiumII，快速以太网参数： $\lambda=63.4\text{ns}$ ， $\lambda=250\mu\text{s}$ ， $\beta=106\text{bps}$ 。

表 8.2 比较了矩阵向量乘法执行的实测时间和预测时间，问题规模为 1000，分别在 1, 2, ..., 8, 16 个处理器上测试。实测时间是运行 100 次的平均时间。此程序的加速情况见本章结尾的图 8.20。

表 8.2 第二个并行算法在 450MHz PII 商用集群上执行的预测时间和实测时间的比较

处理器数	预测时间	实际时间	加速比	Mflops
1	0.0634	0.0638	1.00	31.4
2	0.0324	0.0329	1.92	60.8
3	0.0222	0.0226	2.80	88.5
4	0.0172	0.0175	3.62	114.3
5	0.0143	0.0145	4.37	137.9
6	0.0125	0.0126	5.02	158.7
7	0.0113	0.0112	5.65	178.6
8	0.0104	0.0100	6.33	200.0
16	0.0085	0.0076	8.33	263.2

## 8.6 棋盘式分解

### 8.6.1 设计与分析

在这种域分解方式中，每个原始任务对应一个矩阵元素。负责元素  $a_{i,j}$  的任务将它与  $b_j$  相乘得到  $d_{i,j}$ 。结果向量的每个元素  $c_i$  等于  $\sum_{j=0}^{n-1} d_{i,j}$ 。也就是说，对第  $i$  行而言，将所有  $d_{i,j}$  求和得到向量  $c$  的第  $i$  个元素，如图 8.15 所示。

$$\begin{pmatrix} 4 & 5 & 3 \\ 6 & 2 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \Rightarrow \begin{pmatrix} 4 \times 1 + 5 \times 2 + 3 \times 3 \\ 6 \times 1 + 2 \times 2 + 1 \times 3 \end{pmatrix} \Rightarrow \begin{pmatrix} 23 \\ 13 \end{pmatrix}$$

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \Rightarrow \begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} \\ d_{1,0} & d_{1,1} & d_{1,2} \end{pmatrix} \Rightarrow \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$$

图 8.15 第三种算法将矩阵  $A$  的每个元素与一个原始（基本）任务相关联。

每个任务完成  $a_{i,j} \times b_j$ ，得到  $d_{i,j}$ 。将  $d_{i,j}$  按行归约即可得到向量  $c$  的元素

我们将基本任务规则地聚合成矩形块，并在任进程与块之间建立对应关系，如图 8.3 (c)。因为所有的块大小相同，每块内所要完成的工作量也基本相同，所以我们要设置块的大小以便于将一个任务映射给一个进程。可以将进程想像为一个二维的网格。向量  $b$  被分发到任务网格第一列的任务中，如图 8.16 所示。

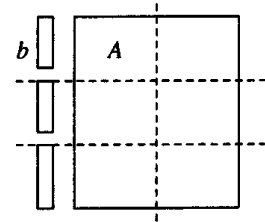


图 8.16 聚合完成后，任务形成一个 2 维网格，每个任务负责  $A$  的一块。本图显示了一个  $3 \times 2$  的任务网格。向量  $b$  被分解到任务网格中与第一列的各块所对应的进程中

现在我们来规划这个并行算法的三个重要步骤，以及实现这三个步骤所用的通信模式，如图 8.17 所示。与块  $A_{ij}$  关联的任务完成与子向量  $b_j$  的矩阵向量乘法。第一步，重新分发  $b$ ，保证每个任务得到  $b$  中相应的部分（后面会详细介绍如何分发）。第二步，每个任务完成各自部分的矩阵向量乘法。第三步，对任务网格每一行中的任务进行一次求和归约。在这之后，结果向量  $c$  就可从任务网格的第一列任务所对应的块中得到。

现在来看如何重新分发向量  $b$ 。假设  $p$  个任务被划分成  $k \times l$  网格。开始时， $b$  被分解到任务网格第一列的  $k$  个任务中。重新分发之后，每一个  $b$  分片被分发到对应行的  $l$  个任务中。

如果  $k=1$ ，重新分发很容易。如图 8.18 (a)。位于位置  $(i, 0)$  的一个任务将自己得到的部分  $b$  发送至  $(0, i)$ ，然后第一行的所有任务将自己部分的  $b$  广播至同列的其他进程。

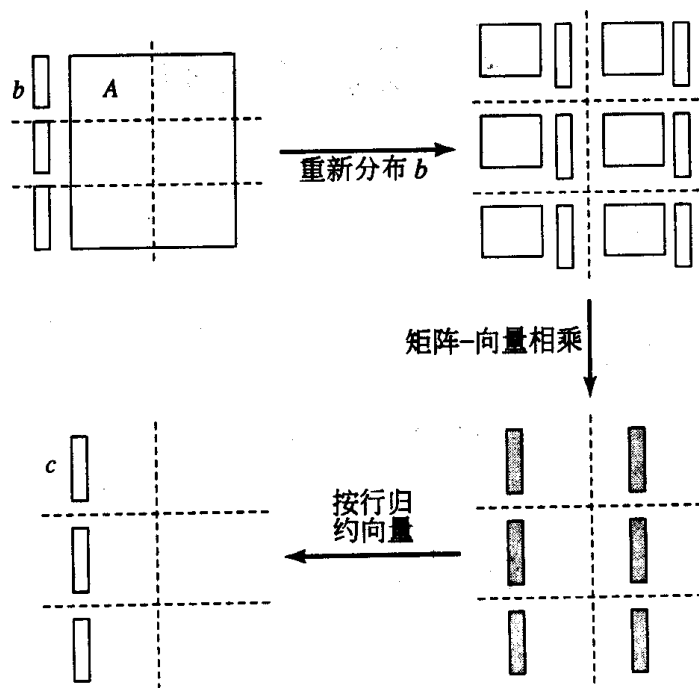


图 8.17 基于棋盘式分解的矩阵向量相乘算法的几个阶段。第一，将  $b$  向量散发到各任务中；第二，每个任务完成自己部分矩阵  $A$  和向量  $b$  的乘法；第三，每一行任务通过归约求和得到向量  $c$

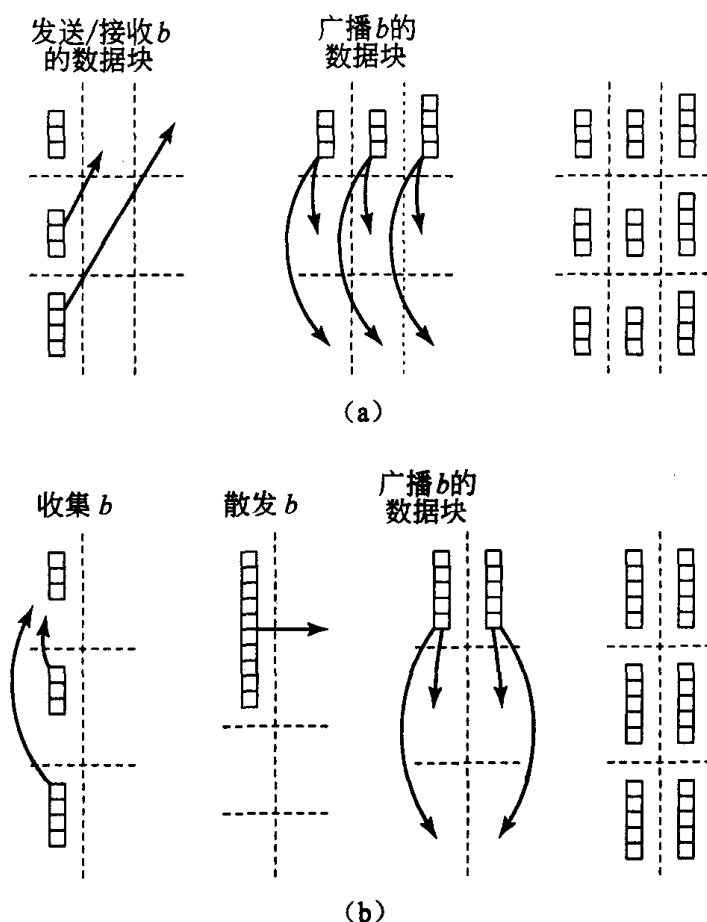


图 8.18 向量的重新分发 (a) 如果进程网格是方的, 算法将会比较简单。第一列的进程将自己部分的  $b$  发送至第一行的进程。然后第一行的进程将其广播到与各自同列的进程。(b) 进程网格不是方形的时候。首先, 第一列的进程将向量收集  $b$  至位于  $(0,0)$  的进程, 然后  $(0,0)$  出的进程再散发  $b$  至第一行的各个进程, 最后, 第一行的进程将相应部分的  $b$  广播至同列进程

如果  $k \neq 1$ , 重新分发就非常复杂了, 这是因为  $b$  分块的大小改变了, 如图 8.18 (b) 所示。在这种情况下, 我们将  $b$  的元素收集至任务  $(0,0)$ , 然后再将它散发到第一行的各个进程。最后, 第一行的进程将自己部分的  $b$  广播到同列的其他进程中。

下面分析此并行算法的复杂度。假设  $m=n$ ,  $p$  是一个平方数, 以便进程可以组织为方形网格 (毫无疑问, 这是最简单的假设。但如果网格是  $p \times 1$ , 分解就成了行块分解; 如果网格是  $1 \times p$ , 分解就成了列块分解。这两种方法的复杂度都已经讨论过了)。

每个进程负责一块矩阵, 最大为  $(n/\sqrt{p}) \times (n/\sqrt{p})$ 。所以矩阵向量乘法这一步的时间复杂度为  $\Theta(n^2/p)$ 。

如果  $p$  是一个平方数,  $b$  的重新分发将在两步内完成。首先, 第一列的所有进程发送相应部分的  $b$  至第一行, 时间复杂度为  $\Theta(n/\sqrt{p})$ 。然后, 第一行的所有进程将对应部分的  $b$  广播到同列的其他进程。广播的时间复杂度为  $\Theta(\log \sqrt{p} (n/\sqrt{p})) = \Theta(n \log p / \sqrt{p})$ 。

在矩阵向量乘法完成后, 进程按行进行求和归约。这个通信操作的时间复杂度为  $\Theta(\log \sqrt{p} (n/\sqrt{p})) = \Theta(n \log p / \sqrt{p})$ 。

综合以上几个步骤, 基于棋盘式分解的矩阵向量相乘算法的时间复杂度是  $\Theta(n^2/p + n \log p / \sqrt{p})$ 。



下面我们来研究此算法的等效率特性。其串行算法的时间复杂度为 $\Theta(n^2)$ 。并行算法的开销是 $p$ 次通信的复杂度, 即 $np \log p / \sqrt{p} = n \sqrt{p} \log p$ 。所以等效率函数为:

$$n^2 \geq Cn\sqrt{p} \log p \Rightarrow n \geq C\sqrt{p} \log p$$

因为 $M(n) = n^2$ , 所以可扩展函数为:

$$M(C\sqrt{p} \log p) / p = C^2 p \log^2 p / p = C^2 \log^2 p$$

这个并行算法比前两个的扩展性要好。

## 8.6.2 创建通信域

通信域是一个不透明的对象, 它为进程提供了消息传递的环境。到目前为止我们实现的所有 MPI 程序中, 组通信都会涉及到所有的进程, 我们可以使用默认的通信域, 即, `MPI_COMM_WORLD`。在棋盘式分解矩阵向量相乘算法的实现中, 有 4 个组通信函数涉及到子进程组:

- 如果 $p$ 不是平方数, 虚拟进程网格中第一列的进程参与收集 $b$ 的通信;
- 如果 $p$ 不是平方数, 虚拟进程网格中第一行的进程参与散发 $b$ 的通信;
- 第一行的每个进程广播 $b$ 的相应部分至与其同列的其他进程;
- 每行的所有进程完成独立的求和归约, 在第一列的进程中产生向量 $c$ 。

为了在某些组通信中只涉及原进程组中的部分进程, 我们需要创建新的通信域。

一个通信域由一个进程组, 上下文, 以及其他属性构成。进程拓扑是通信域的一个重要特征。拓扑可以为进程建立新的编址模式, 而不仅仅是使用进程编号。拓扑是虚拟的, 也就是说它并不依赖于处理器的实际连接方式。MPI 支持两种拓扑结构: 笛卡儿拓扑 (即网格) 和图拓扑。我们的程序需要建立笛卡儿拓扑通信域, 也就是由进程组成的二维虚拟网格。

## 8.6.3 函数 `MPI_Dims_create`

为了使算法具有最好的可扩展性, 所建立的虚拟进程网格最好接近方形。将笛卡儿网格节点数和网格的维数传递给 `MPI_Dims_create`, 它可以返回一个整数数组来指定每一维分别有多少个节点从而能最大程度地达到负载平衡。函数声明如下:

```
int MPI_Dims_create(int nodes, int dims, int *size)
```

三个参数的含义如下:

- `nodes` 输入参数, 网格中的进程数;
- `dims` 输入参数, 我们想要的网格维数;
- `size` 输入/输出参数, 网格中每一维的大小。在调用此函数之前, `size` 中的每个元素 (`size[0]`, ..., `size[dims-1]`) 必须进行初始化。如果 `size[i]=0`, 此函数就会决定这个维度的大小。如果 `size[i]>0`, 那么就是用户所指定的该维的大小。

例如, 如果要寻找含有 $p$ 个进程的二维网格, 就可以用代码完成这项工作:

```

int p;
int size[2];
...
size[0] = size[1] = 0;
MPI_Dims_create(p, 2, size);

```

函数 `MPI_Dims_create` 返回后, `size[0]`, `size[1]` 分别是网格的行数和列数。

## 8.6.4 函数 `MPI_Cart_create`

确定了虚拟网格的每一维的大小之后, 需要为这种拓扑建立通信域。组函数 `MPI_Cart_create` 可以完成此项任务, 其声明如下:

```

int MPI_Cart_create(
MPI_Comm old_comm, int dims, int *size, int *periodic,
int reorder, MPI_Comm *cart_comm)

```

本函数有以下 5 个参数。

- `old_comm`: 以前的通信域。这个通信域内所有的进程都必须调用此函数;
- `dims`: 网格维数;
- `size`: 长度为 `dims` 的数组, `size[j]` 是第  $j$  维的进程数;
- `periodic`: 长度为 `dims` 的数组, 如果第  $j$  维具有周期性 (通信在网格的边缘卷回), 那么 `periodic[j]` 为 1, 否则为 0;
- `reorder`: 标志信息, 指示进程是否能被重新编号。如果 `reorder` 为 0, 那么进程在新的通信域中将保留在旧的通信域 `old_comm` 中的进程号。

`MPI_Cart_create` 有一个输出参数。 `cart_comm` 将指向新的笛卡儿通信域。

下面来看看如何在我们的程序中使用这个函数。先前的通信域是 `MPI_COMM_WORLD`。网格是二维的, `MPI_Dims_create` 设置 `size` 为每一维的大小。我们不采用循环通信, 我们也不关心在新的通信域进程是否保持原来的顺序。由此我们得到下面的代码片段:

```

MPI_Comm cart_comm; /* Cartesian topology communicator */
int p;               /* Processes */
int periodic[2];     /* Message wraparound flags */
int size[2];         /* Size of each grid dimension */
...
size[0] = size[1] = 0;
MPI_Dims_create (p, 2, size);
periodic[0] = periodic[1] = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, size, periodic, 1, &cart_comm);

```

## 8.6.5 读取棋盘式矩阵

我们仍然只允许一个进程负责打开文件, 读取数据, 并分发至对应的进程。分发的方式与列块分解时使用的方式很像。区别在于不是将矩阵的每行分发到各个进程, 而是将每

行分发到一个进程的小集合——虚拟进程网格中处于同一行的进程。如图 8.19 所示。进程 0 负责矩阵输入。每读取一行，它就将其发送到进程网格相应行的第一个进程。接收进程收到此行的数据后就将其进一步分发到进程网格中与之位于同一行的其他进程中。

为了实现这个目标，我们还需要以下其他几个 MPI 函数。

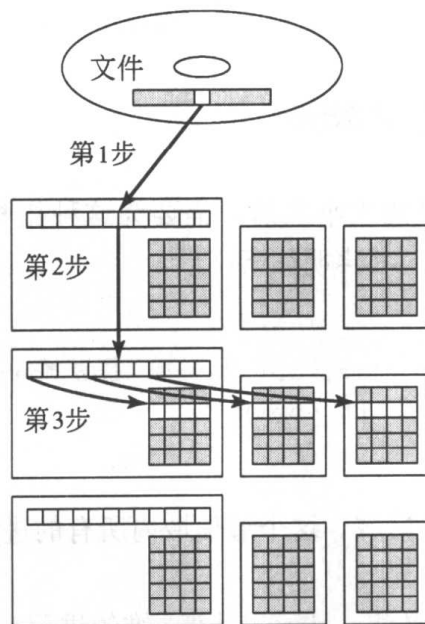


图 8.19 矩阵的棋盘式分解中，矩阵的每一行都散发在一些进程的子集合中。这个进程子集合占据着进程网格中的一行。在这个例子中，9 个进程被组织为 3×3 的网格。当前状态下，文件已经读入了大半。一个进程将下一列读入（第一步），然后将它传向虚拟网格对应行中第一列对应的进程（第二步）。最后这一列进程中的第一个进程负责把矩阵这一行的各个元素散发到同一行的其他进程中去（第三步）

### 8.6.6 函数 MPI\_Cart\_rank

为了将矩阵的一行发送到进程网格相应行的第一个进程中去，进程 0 必须知道它们的进程号。函数 MPI\_Cart\_rank 可以通过进程在网格中的坐标得到其进程号。其声明如下：

```
MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

`comm` 是一个输入参数，它指定了通信操作所在的通信域。`coords` 也是一个输入参数，指定了一个进程在虚拟网格中的坐标。`rank` 是一个输出参数，指定了 `comm` 中位于某个坐标的进程的进程号。

例如，假设虚拟网格有  $r$  行，而矩阵有  $m$  行。矩阵的第  $i$  行映射到进程网格中的第 `BLOCK_OWNER(i,r,m)` 行。下面的代码片段说明了进程 0 怎样确定接收此行的进程：

```
int dest_coord[2]; /* Coordinates of process receiving row */
int dest_id; /* Rank of process receiving row */
int grid_id; /* Rank of process in virtual grid */
int i;
```

```

...
for (i = 0; i < m; i++){
    dest_coord[0] = BLOCK_OWNER(i,r,m);
    dest_coord[1] = 0;
    dest_id = MPI_Cart_rank (grid_comm, dest_coord, dest_id)
    if (grid_id == 0){
        /* Read matrix row 'i' */
        ...
        /* Send matrix row 'i' to process 'dest_id' */
        ...
    } else if (grid_id == dest_id){
        /* Receive matrix row 'i' from process 0 */
        ...
    }
}
}

```

### 8.6.7 函数 MPI\_Cart\_coords

同样，进程也应该能确定自己在虚拟网格中的坐标。这项功能非常有用。例如，它可以指导进程为相应部分的矩阵和向量分配大小合适的内存空间；如果进程 0（读取矩阵的进程）正好是进程网格中某行的第一个进程，通过确定其自身的坐标可以避免它向自己发送消息。MPI\_Cart\_coords 可以完全胜任这个要求，其声明如下：

```
int MPI_Cart_coords (MPI_Comm comm, int rank, int dims, int *coords)
```

前三个参数是输入变量，包含调用者传入的信息。comm 是通信域，rank 是待确定其坐标的进程；dims 是虚拟网格的维数；最后一个参数作为输出参数，它将返回这个进程在虚拟网格中的坐标。

继续前面的例子，下面是进程号为 grid\_id 的进程如何在二维网格中确定其坐标的代码：

```

int grid_id;
MPI_Comm grid_comm;
int grid_coords[2];
...
MPI_Cart_coords (grid_comm, grid_id, 2, grid_coords);

```

### 8.6.8 函数 MPI\_Comm\_split

散发 (Scatter) 是一个组操作。为了仅在进程网格一行的的进程中分发输入矩阵某一行中的数据，我们必须把笛卡儿通信域按行分解为一个个独立的通信域。函数 MPI\_Comm\_split 将某个已存在的通信域中进程进一步划分为几组，并为每组建立一个新的

通信域。其声明如下:

```
int MPI_Comm_split(MPI_Comm old_comm, int partition, int new_rank,
                  MPI_Comm *new_comm)
```

前三个变量是输入参数:

- `old_comm` 待划分的进程所在的通信域。由于这个函数是一个组操作, 所以 `old_comm` 中的所有进程都要调用此函数;
- `partition` 分组数;
- `new_rank` 在新的通信域中本进程的位置。

函数通过参数 `new_comm` 返回指向新通信域的指针。

前面我们通过使用 `MPI_Cart_create` 创建了一个笛卡儿通信域, 从而将进程组织到了虚拟二维网格 `grid_comm` 中。各个进程还可以通过调用 `MPI_Cart_coords` 将其坐标存入数组 `grid_coords` 中。`grid_coords[0]` 是行号, `grid_coords[1]` 是列号。

现在我们使用 `MPI_Comm_split` 将进程网格按行分解。因为我们期望将同一行的各个进程划为一组, 所以我们将 `grid_coords[0]` 的值作为分组数, 同时以 `grid_coords[1]` 作为进程编号变量来按列为进程编号。

```
MPI_Comm grid_comm; /* 2-D process grid */
MPI_Comm grid_coords[2]; /* Location of process in grid */
MPI_Comm row_comm; /* Processes in same row */
MPI_Comm_split (grid_comm, grid_coords[0], grid_coords[1], &row_comm);
```

同样, 也可以用这个函数将笛卡儿通信域按列划分为新的通信域。

有了原先的通信域和这两个新的通信域, 我们就可以完成  $b$  的再分发和最终求和归约中所有的通信操作了。

### 8.6.9 测试

我们已经编写了基于棋盘式分解的矩阵向量乘法并程序。关键函数的编写见本章练习。

下面我们来研究本程序的性能模型, 我们将仍然使用与测试前两个程序所用的相同的集群系统。同时, 我们只考察  $p$  是平方数时的情形。

同样, 令  $\chi$  表示内积计算循环中一次进行迭代所需要的时间。每个进程负责的矩阵的最大尺寸为  $(n/\sqrt{p}) \times (n/\sqrt{p})$ 。所以我们估计并行算法计算部分消耗的时间是  $\chi (n/\sqrt{p}) \times (n/\sqrt{p})$ 。

在重新分发  $b$  过程的第一步, 网格中第一列的进程要向第一行的进程发送相应部分的  $A$ 。一个进程最多处理  $b$  的  $(n/\sqrt{p})$  个元素。所以发送或接收一个包含这些数据的消息耗时为  $\lambda + 8(n/\sqrt{p})/\beta$ 。在  $b$  向量重新分发过程的第二步中, 网格第一行的进程将相应部分的  $b$  广播到同列的其他进程, 耗时为  $\log \sqrt{p} (\lambda + 8(n/\sqrt{p})/\beta)$ 。

当所有进程完成自己部分的矩阵向量乘法运算之后, 每行的进程将通过归约得到各自负责的那一部分  $c$ 。由于通信时间较长, 可以忽略加法运算消耗的时间。通信时间与按列

广播相同:  $\log \sqrt{p} (\lambda + 8(n/\sqrt{p})/\beta)$ 。

集群配置: 450MHz PentiumII, 快速以太网参数:  $\chi=63.4\text{ns}$ ,  $\lambda=250\mu\text{s}$ ,  $\beta=106\text{bps}$ 。

表 8.3 比较了矩阵向量乘法的实际执行时间和预测的执行时间。问题规模为 1000, 分别在 1, 2, ..., 8, 16 个处理器上进行测试。实际时间是 100 次运行的平均时间。此程序与前两个程序的加速比如图 8.20 所示。

这种方法与前两种方法所发送的消息数目完全相同。主要的区别在于: 本算法每个进程得到的  $b$  和  $c$  的元素数是  $\Theta(n/\sqrt{p})$ , 而另两种方法则是  $\Theta(n)$ 。所以当处理器的数目变大的时候, 棋盘式分解比列块分解和行块分解的性能更好。我们的实验结果也证实了这一点。当处理器的数目是 1, 4 和 9 的时候, 棋盘式分解不如另两种方法, 但当处理器数增加到 16 的时候它远远超过了另两种方法。

表 8.3 棋盘式分解矩阵向量乘法算法的预测执行时间和实际执行时间的比较。

矩阵维  $1000 \times 1000$ , 向量有 1000 个元素

处理器数	预期时间	实际时间	加速比	Mflops
1	0.0634	0.0634	1.00	31.6
4	0.0178	0.0174	3.64	114.9
9	0.0097	0.0097	6.53	206.2
16	0.0062	0.0062	10.21	322.6

## 8.7 本章小结

本章我们设计、分析并测试了 3 个矩阵向量乘法的 MPI 并程序。三者分别基于按行分解, 按列分解和棋盘式分解。

我们分析了这三种算法的等效率特性。棋盘式分解算法等效率函数的性质最好, 也就是说它比另两种方法更适合处理器数目较多的情况。测试结果也证明了这一点, 如图 8.20 所示。

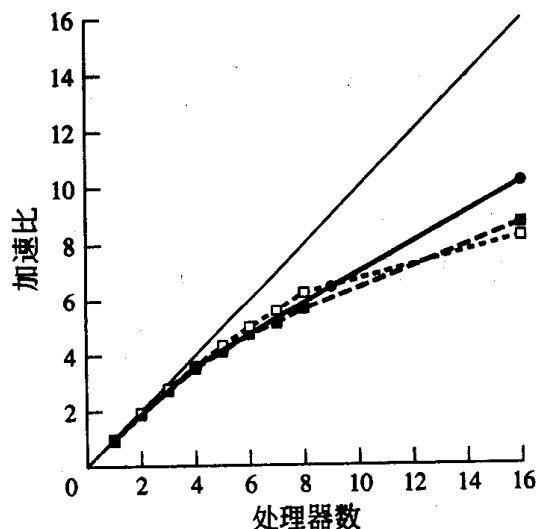


图 8.20 商用集群上三种算法处理  $1000 \times 1000$  矩阵与长为 1000 的向量乘法的加速情况。点线为按行分解加速情况, 短划线是按列分解的加速情况, 而实线则是棋盘式分解的加速情况

由于每种算法都基于不同的矩阵向量分解方式, 其产生的程序也有不同的通信方式。所以我们遇到了许多功能强大的 MPI 散发和收集函数。我们还学会了如何创建网格拓扑通信域, 以及如何将一个通信域内的函数划分为小组以构建新的通信域。

与完成同样功能的 C 程序相比, 并行程序和辅助函数的代码总长度要长很多。读取和分发矩阵向量数据在并行程序中要复杂的多, 除此之外二者实际用于计算的代码量不相上下。开发和调试并行程序是比较麻烦的, 这也就是为什么我们尽量使它们更一般化, 放入库中以便重用的原因了。我们也可以使用一些免费的库。ScaLAPACK 项目有一个庞大的与 MPI 兼容的支持科学和工程计算的函数包。ScaLAPACK 的详细信息请见参考文献。

## 8.8 主要术语

all-gather communication	全收集通信
all-to-all communication	全交换通信
attributes	属性
block-decomposed vector	分块的向量
checkerboard block decomposition	棋盘式分解
columnwise block-striped decomposition	按列块分解
communicator	通信域
context	上下文
distributed vector	分布式向量
process group	进程组
replicated vector	复制(冗余)向量
topology	拓扑

## 8.9 参考文献

介绍矩阵向量乘法的文献还有: Pacheco【89】(按行分解); Bertsekas 和 Teitsiklis【9】(按行和按列分解), Fox 等【33】(棋盘式分解), Grama 等【44】(按行和棋盘式分解)。

20 世纪 90 年代中期, 美国几个政府机构开始了 ScaLAPACK 项目。Oak Ridge 国家实验室、Rice 大学、加州大学伯克利分校、加州大学洛杉矶分校、伊利诺斯大学、田纳西大学诺克斯维尔分校都参与其中。这些组织开发了许多与 MPI 兼容的数值函数库。这些免费的库有许多功能, 包括矩阵和向量的基本运算, 线性方程求解, 特征值和特征向量求解, 以及迭代矩阵的预处理。更详细资料请参见 [www.netlib.org/scalapack/](http://www.netlib.org/scalapack/)。

Communications of the ACM 1994 年 3 月刊聚焦于人工智能, 其中有三篇关于神经网络的综述。

## 8.10 练习题

8.1 在你的并行计算机上测试本章的并程序，测试应包括矩阵和向量的读取时间。哪个程序的性能最好？为什么？

8.2 使用本章 8.4 节性能模型，估算第一个程序的执行时间，加速比和 MFLOPS。假设  $n=1,000$ ，处理器为 9,10,...,15 个。

8.3 编写矩阵向量乘法程序，其中矩阵按行分解，向量分块。假设矩阵和向量从数据文件中读取，文件格式与本章例子中用到的相同。程序执行结束时，结果向量  $c$  分块存在于各个进程中。

8.4 用另一种方法实现 `read_col_stripped_matrix`。在本章的函数中，由一个进程负责打开和读取文件。之后的过程要求与本章函数的实现不同，矩阵的分发将通过  $p-1$  次简单的发送-接收消息过程来完成。对于矩阵的每一行，读取进程要调用  $p-1$  次 `MPI_Send`，其他进程分别调用一次 `MPI_Recv`。所有进程分配空间都不大于  $n \lceil n/p \rceil$  个矩阵元素。

8.5 用另一种方法实现 `read_col_stripped_matrix`。不调用 MPI 函数，所有进程各自打开文件，只读取自己需要的部分。

8.6 编写一个计算矩阵向量相乘的程序，矩阵按列分解散发在各个进程中，向量被复制各个进程中。假设矩阵和向量从数据文件中读取，其格式与本章例子中的相同。程序执行结束时，结果向量  $c$  需要在各个进程中被复制。

8.7 假设 `grid_comm` 是具有笛卡儿拓扑结构(进程被组织成一个二维网格)的通信域。编写一段代码将进程网格按列划分。每个进程的 `col_comm` 变量应该是一个包括有调用进程和所有与之同列进程的通信域，而任何其他进程都不包含在该通信域中。

8.8 假设 `grid_comm` 是具有笛卡儿拓扑结构(进程被组织成一个二维网格)的通信域。编写一段代码，以说明怎样可以通过使用 `read_block_vector` 函数来打开包含向量的数据文件并将其分发到 `grid_comm` 的各个进程中去。其文件名为“Vector”，元素为双精度浮点数。

8.9 为棋盘式分解算法的矩阵向量乘法程序编写重新分发向量  $b$  的函数。假设向量有  $n$  个元素，进程组织成  $r \times c$  的网格。开始时， $b$  被分发到进程网格中第一列的各个进程中。 $(i, 0)$  处的进程负责从第 `BLOCK_LOW` ( $i, r, n$ ) 开始的 `BLOCK_SIZE` ( $i, r, n$ ) 个  $b$  的元素。重新分配后，第  $j$  列的每个进程负责从 `BLOCK_LOW` ( $j, c, n$ ) 开始的 `BLOCK_SIZE` ( $j, c, n$ ) 个  $b$  的元素。

(a) 假设  $p$  是平方数，即： $r=c$ ；

(b) 假设  $p$  不是平方数，即： $r \neq c$ ；

(c) 对  $p$  不作任何假设。

8.10 编写基于棋盘式分解的矩阵向量乘法程序。程序从文件读取矩阵及向量并将结果打印至标准输出。输入文件的名称作为命令行参数给出。

8.11 编写一个函数完成大小为  $n \times n$  的矩阵  $A$  的转置。假设调用前  $A$  按行分解位于  $p$  个进程中。调用之后， $A$  按列分解位于  $p$  个进程中。

8.12 二叉搜索树通过线性结构来组织  $n$  个值，它可以保证  $\Theta(\log n)$  的检索时间。如



果知道每个值的访问概率, 可以建立一棵优化的二叉搜索树以最小化搜索时间, 如图 8.21 所示。

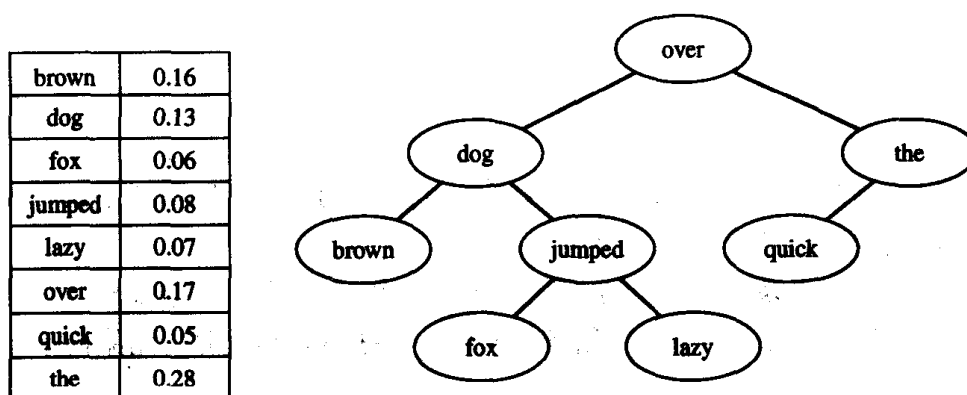


图 8.21 给定一组值和每个值的访问概率, 优化的二叉搜索树可以缩短检索时间

图 8.22 是用 C 编写的一个优化二叉搜索树的查找算法, 假设已经给定了一组概率值。程序的伪码来自于 Baase 和 Gelder 【5】; 详细情况可以查阅他们的教材。

```

/*
 * Given p[0], p[1], ..., p[N-1], the probability of each key
 * in an ordered list of keys being the target of a search,
 * this program uses dynamic programming to compute the
 * optimal binary search tree that minimizes the average
 * number of comparisons needed to find a key.
 *
 * Last modification: 12 September 2002
 */
#include <stdio.h>
#include <values.h>

main (int argc, char *argv[]) {
    float bestcost; /* Lowest cost subtree found so far */
    int bestroot; /* Root of lowest cost subtree */
    int high; /* Highest key in subtree */
    int i, j;
    int low; /* Lowest key in subtree */
    int n; /* Number of keys */
    int r; /* Possible subtree root */
    float rcost; /* Cost of subtree rooted by r */
    int **root; /* Best subtree roots */
    float **cost; /* Best subtree costs */
    float *p; /* Probability of each key */
    void alloc_matrix (void ***, int, int, int);
    void print_root (int **, int, int);
    /* Input the number of keys and probabilities */

```

```

scanf ("%d", &n) ;
p = (float *) malloc (n * sizeof (float)) ;
for (i = 0; i < n; i++)
scanf ("%f", &p[i]) ;
/* Find optimal binary search tree */
alloc_matrix ((void ***) &cost, n+1, n+1, sizeof (float)) ;
alloc_matrix ((void ***) &root, n+1, n+1, sizeof (int)) ;
for (low = n; low >= 0; low--) {
    cost[low][low] = 0.0;
    root[low][low] = low;
    for (high = low+1; high <= n; high++) {
        bestcost = MAXFLOAT;
        for (r = low; r < high; r++) {
            rcost = cost[low][r] + cost[r+1][high];
            for (j = low; j < high; j++) rcost += p[j];
            if (rcost < bestcost) {
                bestcost = rcost;
                bestroot = r;
            }
        }
        cost[low][high] = bestcost;
        root[low][high] = bestroot;
    }
}
/* Print structure of binary search tree */
print_root (root, 0, n-1) ;
}
/* Print the root of the subtree spanning keys
'low' through 'high' */
void print_root (int **root, int low, int high) {
printf ("Root of tree spanning %d-%d is %d\n",
    low, high, root[low][high+1]) ;
if (low < root[low][high+1]-1)
print_root (root, low, root[low][high+1]-1) ;
if (root[low][high+1] < high-1)
print_root (root, root[low][high+1]+1, high) ;
}
/* Allocate a two-dimensional array with 'm' rows and
'n' columns, where each entry occupies 'size' bytes */
void alloc_matrix (void ***a, int m, int n, int size)
{
    int i;
    void *storage;
    storage = (void *) malloc (m * n * size) ;
    *a = (void **) malloc (m * sizeof (void *)) ;

```

```
for (i = 0; i < m; i++) {  
    (*a) [i] = storage + i * n * size;  
}  
}
```

图 8.22 优化二叉搜索树查找算法的 C 语言实现

用分解-通信-组合-映射的设计方法对此程序进行并行化（提示：寻找一种允许多个进程同时计算的组合方式）。

# 第9章 文档分类

It is impossible to enjoy idling thoroughly unless one has plenty of work to do.

Jerome Klapka Jerome, *Idle Thoughts of an Idle Fellow*

## 9.1 概 述

万维网 (World wide web) 中包含着数以百万计的文本文档, 许多的问题都可以通过在其中查找合适的文档来得到解答。为了找到这些包含相关信息的文档, 我们需要求助于自动搜索引擎。为了简化文档内容与查询, 以及文档与文档之间的比较, 我们通常采用一个向量来说明文档中的内容。向量的每一维表示文档的内容和一个特定的概念 (可能用一个词或者一个短语来描述) 之间的匹配度。

在本章中我们来开发一个应用程序, 它读入一个关键词字典, 找到一批待处理的文本文档, 读取它们的内容, 为每个文档产生一个向量, 并最终将这些向量保存起来。与前几章中我们讨论过的问题不同, 这个问题更适宜于采用功能分解的分解模式。我们将开发一个具有管理者/工人模式的并行程序来解决这个实际问题。在开发并改进这个程序的过程中, 我们将学习下面几个新的 MPI 函数:

- MPI\_Irecv, 用来初始化一个非阻塞的接收;
- MPI\_Isend, 用来初始化一个非阻塞的发送;
- MPI\_Wait, 等待一次非阻塞通信结束;
- MPI\_Probe, 检查一个消息是否已经到达;
- MPI\_Get\_count, 获得一个消息的长度;
- MPI\_Testsome, 得到所有已完成非阻塞通信的信息。

## 9.2 并行算法设计

我们的目标是设计一个程序: 它读入一个字典, 在一个目录结构中查找并处理纯文本文件 (比如.html、.tex 和.txt 后缀的文件)。对每个纯文本文件, 程序将打开它, 读取其内容, 然后产生一个特征向量 (profile vector) 以记录字典中的每个词在该文档中出现的次数。最后这个程序把它处理过的每个纯文本文件的特征向量写入一个文件。在图 9.1 中, 这个处理过程被分解成彼此间存在数据依赖关系的 5 个子步骤。

## 9.2.1 划分与通信

虽然读取字典和识别文档可以并发进行,但为了更充分的开发并行性,我们还需要对任务进行更精细的划分。我们认为在整个处理流程中,读取文档与产生特征向量占用了绝大部分的计算时间,所以我们应该为每个文档产生两个任务:一个用于读取文档,一个用于生成特征向量,细化后的数据相关关系如图 9.2 所示。

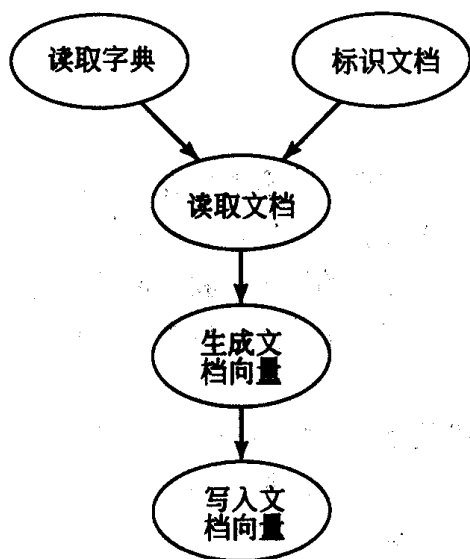


图 9.1 文本分类问题包含 5 个子任务

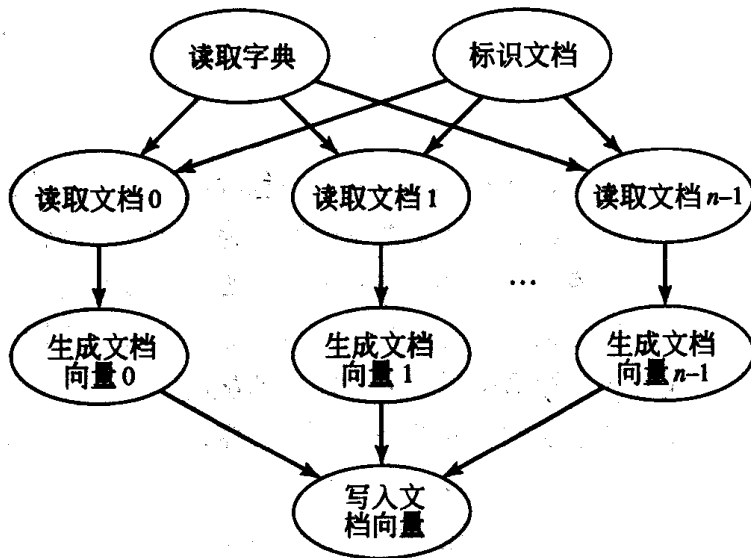


图 9.2 每个文档的读取和处理过程可以并行执行

自然,这个算法可以使用功能分解:每个操作映射为一个原始任务,每个原始任务和一个数据单元(一个文档)相关联。

## 9.2.2 聚集和映射

任务的数目在编译时无法确定,而各个任务之间不需要通信,完成不同任务(处理每个文档)所花费的时间可能会很大,因为文档的大小可能有很大的不同,而且某些类型的文档(比如.html 文件)要比其他文档(比如.txt 文件)更难处理。由于任务有这些特性,我们通过映射决策过程(如图 3.7 所示)最后决定:任务的映射应该在运行期完成。

## 9.2.3 管理者/工人模式

为了支持在运行期进行任务分配,我们将构造一个管理者/工人模式的并程序。其中的一个进程负责管理所有已分配和未分配的数据,我们称之为管理者,它将计算任务分配给其他的进程(称为工人),并从它们那里获得处理完毕的结果,如图 9.3 所示。

我们选择每次只分配一个任务给每个工人,这样做可以平衡工作负载。当一个工人完成了某个任务后,如果管理者没有更多的任务可以分配它,那么它的工作就结束了,此时,

所有其他的工人所剩的工作也都不会超过一个任务。

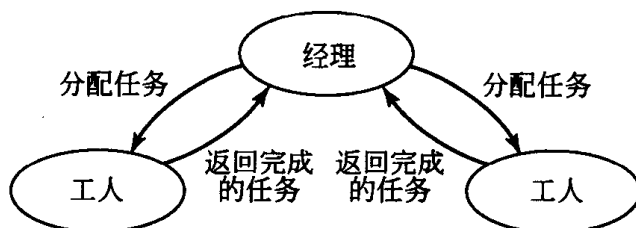


图 9.3 在采取管理者/工人模式的并行算法中，管理进程将任务分配给各个工人进程，并从工人进程处接收处理结果

每次只分配一个任务给工人的方法也有不足：这给并行算法引入了额外的通信开销，增加了执行时间同时降低了加速比。

到目前的讨论为止，我们编写的所有并行程序都是 SPMD 类型的（单程序流多数据流）。在 SPMD 程序中，每个进程执行相同的函数（尽管可能有一个指定的进程专门负责文件或用户输入/输出）。管理者/工人模式是一种与 SPMD 类型不同的新的并行程序类型。管理进程与工人进程扮演不同的角色。在一个实现管理者/工人模式的并行程序中，通常情况下在程序执行的开始就会有一个控制流的分离：管理进程执行一个函数，而工人进程执行另一个函数。

负载均衡是保证程序高效的一个重要因素，这也是我们为什么采用管理者/工人模式来设计该并行算法的原因。第一步是确定哪些任务需要由管理者来完成，哪些任务需要由工人进程来完成。因为文件名需要由管理者分配给工人，所以识别目标文档很显然应该是管理者的工作。词典的读取应该由工人来完成，因为这是构造特征向量的一部分。给定一个文件名，工人进程将读取相应的文件，并构造出文档的特征向量。最后，管理进程应该收集这些文档的特征向量，并把结果写入文件。

在图 9.3 所示的任务/通道图中，可以看到，在管理进程和每个工人进程间有一个交互的环路。管理者分配给工人一个任务，一段时间后，工人进程将完成了的任务返回给管理进程（或者仅仅简单地报告任务已经完成），此时，管理者可能会给工人进程分配新的任务。

这个交互的环路可能以管理者给工人消息作为起点，也可能相反。在我们的设计中，将由工人进程开始这个循环：它发送给管理进程一个消息，告诉管理进程它已经准备好接收任务。我们这样做是因为我们无法确定不同处理器上的 MPI 进程是否都已开始执行。这样，管理者只给它知道的那些准备好的工人进程分配任务。

## 9.2.4 管理进程

图 9.4 给出了管理进程的伪代码。管理进程首先在用户给出的目录中识别出  $n$  个纯文本文档。它从 0 号工人处接收  $k$  的值（ $k$  为每个文档其特征向量中元素的个数），因此它需要分配一个  $n \times k$  的矩阵  $s$  来存储它所接收到的文档特征向量。变量  $d$  用来记录已分配的文档数目，变量  $t$  用来记录已经结束的工人进程的数目。管理进程将这两个变量初始化为文档未被分配，工人进程尚未结束的状态。

```

manager (管理者)
local variables (局部变量)
 $a$ —array showing document assigned to each process (表示指定给每个进程的文档的数组)
 $d$ —document assigned (指定的文档)
 $j$ —ID of worker requesting document (请求文档的工人 ID)
 $k$ —document vector length (文档向量长度)
 $n$ —number of documents (文档数量)
 $p$ —total number of processes ( $p-1$  are workers) (总进程 (其中  $p-1$  个工人))
 $s$ —storage array containing document vectors (包含文档向量的存储数组)
 $t$ —terminated workers (终止的工人)
 $v$ —individual document vector (单个文档向量)
Identify  $n$  documents in user specified directory (在用户指定的目录中指定  $n$  个文档)
Receive dictionary size  $k$  from worker 0 (从工人 0 获得字典大小  $k$ )
Allocate  $s$  with dimension  $n \times k$  to store document vectors (为  $s$  分配  $n \times k$  的空间以存放文档向量)
 $d \leftarrow 0$ 
 $t \leftarrow 0$ 
repeat
    receive message from worker  $j$  (从工人  $j$  接收消息)
    if message contains document vector  $v$  (消息包含文档向量  $v$ )
         $s[a[j]] \leftarrow v$ 
    else
        {message is first request for work-do nothing (消息是最初的工作请求—什么也不做)}
    if  $d < n$  then
        send name of document  $d$  to worker  $j$  (发送文档  $d$  的名字给工人  $j$ )
         $a[j] \leftarrow d$ 
         $d \leftarrow d + 1$ 
    else
        send termination message to worker  $j$  (发送终止消息给工人  $j$ )
         $t \leftarrow t + 1$ 
    endif
until  $t = p - 1$ 
write  $s$  to output file (将  $s$  写到输出文件中)

```

图 9.4 文档分类程序中管理进程的伪代码

初始化之后, 管理进程将进入一个循环, 直到所有的工人进程都已经结束它才退出。在这个循环里, 它从各个工人进程那里接收消息。如果这个消息包含一个文档的特征向量, 那么它将这个向量存储在  $s$  中适当的位置。否则, 工人进程是在通过这个消息告诉管理进程它已经准备好接收新的文档 (这种情况每个工人进程只会发生一次)。如果还有未被处理过的文档, 管理者就将该文档的名字发送给工人, 同时在数组  $a$  中记录下已分配的文档, 并将  $d$  加一。如果所有文档都被处理过了, 管理者将发送结束消息给工人, 并将  $t$  加 1。

在退出循环后, 管理进程将  $s$  的内容写入一个文件。

## 9.2.5 MPI\_Abort 函数

在上面的代码中, 当管理进程识别出给定目录下的  $n$  个纯文本文件, 并且从工人进程 0 处接收到文档特征向量的大小  $k$  之后, 它需要分配一个大小为  $n \times k$  的矩阵来存放特征向

量。这个任务只由管理进程来完成，此刻工人进程在处理其他的事情。

如果空间分配失败，我们需要一个简单的方法来中止 MPI 程序的执行。MPI\_Abort 函数提供了这种功能，它的函数声明如下：

```
int MPI_Abort(MPI_Comm comm, int error_code)
```

MPI\_Abort 函数将尽力处理好 comm 给出的通信域中所有进程的退出事务，它将返回 error\_code 的值。

## 9.2.6 工人进程

现在我们来考虑工人进程。每个工人进程需要一份词典。一个解决方法是每个工人都打开词典文件，读取它的内容。另一个解决方法是由一个工人打开词典文件，读取它的内容，并将词典广播给其他的工人。如果并行计算机内部的广播带宽比并行计算机与文件服务器之间的带宽要大，第二种方法的性能更好，这也是我们所采用的方案。

```
worker (工人)
local variables (局部变量)
f—file name (文件名)
k—dictionary size (字典大小)
v—document vector (文档向量)
send first request for work to manager (发送最初的工作请求)
if worker 0 then
    read dictionary from file (从文件中读入字典)
endif
broadcast dictionary among workers (向其他工人广播)
build hash table from dictionary (根据字典构建哈希表)
if worker 0 then
    send dictionary size k to manager (将字典大小 k 发送给管理者)
endif
repeat
    Receive file name f from manager (从管理者那里接收文件名 f)
    if f indicates termination (f 表示终止)
        exit loop
    else
        read document from file f (从文件 f 中读入文档)
        generate document vector v (生成文档向量 v)
        send v to manager (把 v 发送给管理者)
    endif
forever
```

图 9.5 文档分类程序中，工人进程的伪代码

工人进程的伪代码如图 9.5 所示。一旦准备完毕，工人进程将通知管理者它已经准备就绪（技术上来说，这并不正确，因为此时工人还没有得到词典。但提早通知可以使两个过程能够相互重叠：发送消息并从管理者接收第一个文档的文件名的过程与建立词典的过



程)。由工人 0 来读取词典，所有的工人（除了管理者以外）都参加到广播词典的聚合通信操作中。每个工人由词典构造一个哈希表，这样可以使得词典的访问时间变为常数，从而提高文档处理的速度。工人 0 同时也将词典的大小发送给管理进程。

然后工人进程进入一个循环。它从管理进程接收消息，如果消息是文档的文件名，工人会读取相应的文件，构造文档的特征向量，并把这个特征向量发回给管理者，然后再次进入循环。如果工人从管理者收到一个结束消息，这表明已经没有未被处理的文档了，于是工人进程将停止执行。

我们可以用一个任务/通道图来描述这种管理者/工人的模式。有五个进程的情形由图 9.6 给出（一个管理进程，四个工人进程）。

我们需要确定哪一个进程是管理者。实际上这并无所谓，但为了描述起来方便，我们将指派 `MPI_COMM_WORLD` 通信域中编号为 0 的进程为管理进程，其他的进程为工人进程。

注意在我们的设计中，我们假设至少有两个进程：一个管理者和至少一个工人。在实现中我们需要确认至少有两个 MPI 进程在运行。

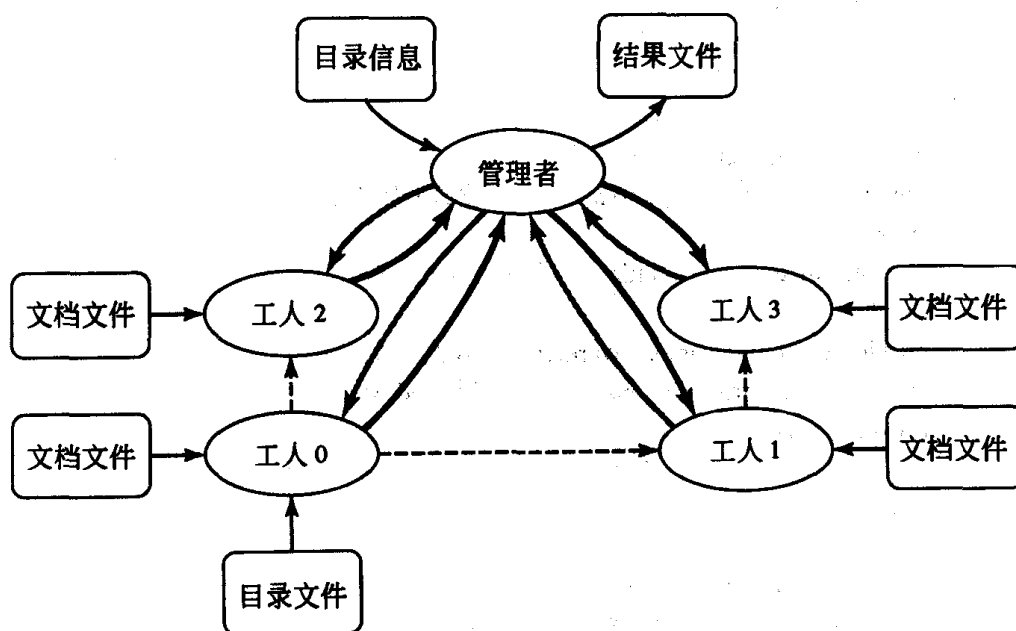


图 9.6 并行文档分类算法的任务/通道图。虚线箭头表示用来广播词典的通道，粗的灰箭头表示分发文档文件名的通道。粗的黑箭头用来表示将特征向量发回给管理者的通道

## 9.2.7 建立一个只有工人的通信域

在前面设计的并行算法中，当管理进程在目录中搜索纯文本文件的时候，词典需要在工人中进行广播。`MPI_Bcast` 是一个组通信操作，需要一个通信域中的所有的进程参与。

为了支持一个只涉及工人的广播，我们必须创建一个包括所有工人进程的新的通信域。在第八章中我们已经学习过如何使用 `MPI_Comm_split` 来将一个通信域划分为两个或者多个新的通信域。

在本例中，我们只是不希望把管理进程包含在这个新的通信域中。在管理进程中，我们可以通过传递常数 `MPI_UNDEFINED` 来作为参数 `split_key` 的方式将管理进程排斥在新的通信域外。这样，管理进程的返回值 `new_comm` 应该是 `MPI_COMM_NULL`。

我们可以用下面的代码来创建一个只包括工人的通信域：

```
int id;
MPI_Comm worker_comm;
...
if(!id)/* Manager */
    MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED, id, &worker_comm);
else /* Worker */
    MPI_Comm_split(MPI_COMM_WORLD, 0, id, &worker_comm);
```

## 9.3 非阻塞通信

管理进程的工作可以分为三个阶段。在第一阶段中，管理进程从用户指定的目录中寻找纯文本文件，从工人进程 0 接收词典大小，然后分配一个二位数组来存储文档的特征向量。在第二阶段中，管理者将文档分配给工人，并回收特征向量。在第三阶段中，管理进程将所有的文档特征向量存储到一个文件中。

现在我们集中考虑第一阶段的任务。管理进程需要搜索一个目录结构，并从工人进程 0 接收一个消息。这两个任务可以重叠的进行吗？

到目前为止，我们都采用 `MPI_Send` 和 `MPI_Recv` 来进行点到点的消息传递。这些是阻塞操作。`MPI_Send` 函数直到要发送的消息被复制到系统缓冲区或者消息已经被发送的时候才会返回（到调用者）。在这两种情况下，只要函数已经返回，你就可以重写消息缓冲区。同样的 `MPI_Recv` 直到消息被实际的写入用户指定的缓冲区后才会返回，只要函数已经返回，你就可以访问缓冲区中的消息。

阻塞式的发送和接收操作可能会限制并行政务的性能。比如说 `MPI_Send`，可能会有很多原因导致系统没有把消息复制到系统缓冲区，这种情况下，即使你并不打算马上重写消息缓冲区中的内容，函数在消息被发送出去之前会一直被阻塞。

在消息到达前就调用一个接收操作可能会节省时间，因为系统可以直接将消息的内容写入目标缓冲区，而不用先写入一个临时的系统缓冲区，这样可以减少一次复制操作。但采用 `MPI_Recv` 很难实现这种操作。如果这个函数被调用太早，调用进程会一直阻塞，直至消息到来；如果这个函数调用得太晚，消息已经被写入系统缓冲区，仍然需要一次复制。

幸运的是，MPI 库提供了非阻塞式发送和接收函数。调用 `MPI_Isend` 和 `MPI_Irecv` 函数仅仅会公布，或者说初始化，一些合适的通信操作（可以将“I”看成是“初始化”的缩写）。而用户进程将不能访问消息缓冲区，直至用户通过调用 `MPI_Wait` 显式地完成了通信操作。

公布消息，然后进行一些其他的计算或输入/输出操作，最后再完成消息操作。这样的模式可以在两种情况下节省时间。第一，它允许系统消除发送/接收消息时进行额外的消息副本。第二，它允许专门的通信协处理器（如果在并行计算机中存在的话）来完成通信的

相关操作，而将处理器专门用来处理仅涉及本地数据的计算操作。

### 9.3.1 管理进程的通信

在管理进程开始执行的时候，它便知道它需要从一个工人进程那里接收词典的大小。在它确定所要处理的文本文件的数量之前，它决不会用到这个值（当然，需要接收的消息长度越长，非阻塞接收的好处就越大）。下面先让我们来看看进行非阻塞通信所用到的两个函数。

### 9.3.2 MPI\_Irecv 函数

函数 MPI\_Irecv 的声明如下：

```
int MPI_Irecv(void *buffer, int cnt, MPI_Datatype dtype, int src, int tag,
              MPI_Comm comm, MPI_Request *handle)
```

前 6 个参数和 MPI\_Recv 的相同，但是，MPI\_Irecv 仅仅初始化接收操作。在与之对应的 MPI\_Wait 函数的调用返回之前，你将不能访问 buffer（来得到一个有意义的值）。MPI\_Irecv 函数返回时，handle 指向一个 MPI\_Request 对象，它代表了一个已经初始化了的通信操作。

需要指出的是，这个函数并不返回一个指向 MPI\_Status 对象的指针，因为实际的接收操作尚未完成。

### 9.3.3 MPI\_Wait 函数

下面是 MPI\_Wait 函数的声明：

```
int MPI_Wait(MPI_Request *handle, MPI_Status *status)
```

MPI\_Wait 会一直阻塞，直至参数 handle 所关联的操作完成。对发送操作来说，此时就可以向缓冲区写入新的值。而对接收操作来说，便可以从缓冲区中读取消息，而 status 所指向的 MPI\_Status 对象包含了所接收消息的信息。

### 9.3.4 工人的通信

现在我们来讨论工人进程进行通信时所需要的 MPI 支持。在分配第一个文档之前，每一个工人需要通知管理者它已经准备好。它可以先初始化这个发送操作，然后立即参与词典的广播通信，并建立哈希表。

工人需要从管理者接收文件名（完整的路径）。因为目录结构可以有不同的深度，所以没有办法预先知道这个文件名的长度。因此工人在实际读取文件名之前，最好能先确定它的长度。

下面给出了满足上述要求所用到的 3 个 MPI 函数。

### 9.3.5 MPI\_Isend 函数

```
int MPI_Isend(void *buffer, int cnt, MPI_Datatype dtype, int dest, int tag,
              MPI_Comm comm, MPI_Request *handle)
```

MPI\_Isend 函数初始化一个非阻塞的发送操作。前六个参数与 MPI\_Send 相应的参数含义相同，最后一个参数 handle 是一个输出参数，它指向一个不可见的 MPI\_Request 对象，这个对象由运行时系统构造和维护，用来表示通信请求。在对应的 MPI\_Wait 函数返回之前，请不要覆盖消息缓冲区中的内容。

### 9.3.6 MPI\_Probe 函数

```
int MPI_Probe(int src, int tag, MPI_Comm comm, MPI_status *status)
```

src 是消息源的编号，tag 是待接收消息的标签，comm 是通信域，而 status 是一个指向 MPI\_Status 对象的指针。在一个与 src 和 tag 相匹配的消息（到达并）可供接收之前，MPI\_Probe 会一直阻塞。它返回时，status 指针指向的对象将携带关于相应消息的信息，但实际上它并不接收该消息。

将 src 设置为 MPI\_ANY\_SOURCE 时，你可以检查从任意进程发送来的消息。将 tag 设置为 MPI\_ANY\_TAG，可以检查从 src 发送过来的任何消息，同时使用这两个值时，这个函数将会匹配所有收到的消息。

一般来说，最好将 src 和 tag 设置得越精确越好，以防止乱序到达的消息造成错误的匹配。在本例中，由于工人知道消息源，以及从管理进程发送来的消息的标签，所以没有必要使用上面的常数。

### 9.3.7 MPI\_Get\_count 函数

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype dtype, int *cnt)
```

status 是一个指向 MPI\_Status 对象的指针；dtype 是消息中元素的数据类型；cnt 是一个整型指针。通过调用 MPI\_Get\_count，可以通过 cnt 返回消息中元素的数目。

## 9.4 文档分类的并程序序

通过使用这些新的 MPI 函数，我们可以构造用于文档分类的并程序序，这一节中并不包含完整的程序：管理进程中的目录搜索和将特征向量写入文件的函数，工人进程中的哈希表构造函数和特征向量构造函数都被省略了。毕竟我们在本章中关心的是管理者/工人模式下的 MPI 程序的一般结构，以及如何在设计中应用本章介绍的新的 MPI 函数。

程序如图 9.7 所示。

```

/*
 * Document Classification Program
 */
#include <mpi.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <ftw.h>

#define DICT_SIZE_MSG 0 /* Msg has dictionary size */
#define FILE_NAME_MSG 1 /* Msg is file name */
#define VECTOR_MSG 2 /* Msg is profile */
#define EMPTY_MSG 3 /* Msg is empty */

#define DIR_ARG 1 /* Directory argument */
#define DICT_ARG 2 /* Dictionary argument */
#define RES_ARG 3 /* Results argument */

typedef unsigned char uchar;

int main (int argc, char *argv[]) {
    int id; /* Process rank */
    int p; /* Number of processes */
    MPI_Comm worker_comm; /* Workers-only communicator */
    void manager (int, char **, int);
    void worker (int, char **, MPI_Comm);
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    if (argc != 4) {
        if (!id) {
            printf ("Program needs three arguments:\n");
            printf ("%s <dir> <dict> <results>\n", argv[0]);
        }
    } else if (p < 2) {
        printf ("Program needs at least two processes\n");
    } else {
        if (!id) {
            MPI_Comm_split (MPI_COMM_WORLD, MPI_UNDEFINED, id, &worker_comm);
            manager (argc, argv, p);
        } else {
            MPI_Comm_split (MPI_COMM_WORLD, 0, id, &worker_comm);
            worker (argc, argv, worker_comm);
        }
    }
}

```

```

MPI_Finalize ( ) ;
return 0;
}
void manager (int argc, char *argv[], int p) {
    int assign_cnt; /* Docs assigned so far */
    int *assigned; /* Document assignments */
    uchar *buffer; /* Store profile vectors here */
    int dict_size; /* Dictionary entries */
    int file_cnt; /* Plain text files found */
    char **file_name; /* Stores file (path) names */
    int i;
    MPI_Request pending; /* Handle for recv request */
    int src; /* Message source process */
    MPI_Status status; /* Message status */
    int tag; /* Message tag */
    int terminated; /* Count of terminated procs */
    uchar **vector; /* Profile vector repository */

    void build_2d_array (int, int, uchar ***);
    void get_names (char *, char ***, int *);
    void write_profiles (char *, int, int, char **, uchar **);

    /* Put in request to receive dictionary size */
    MPI_Irecv (&dict_size, 1, MPI_INT, MPI_ANY_SOURCE, DICT_SIZE_MSG,
               MPI_COMM_WORLD, &pending);

    /* Collect the names of the documents to be profiled */
    get_names (argv[DIR_ARG], &file_name, &file_cnt);

    /* Wait for dictionary size to be received */
    MPI_Wait (&pending, &status);

    /* Set aside buffer to catch profiles from workers */
    buffer = (uchar *)
    malloc (dict_size * sizeof (MPI_UNSIGNED_CHAR));

    /* Set aside 2-D array to hold all profiles.
       Call MPI_Abort if the allocation fails. */
    build_2d_array (file_cnt, dict_size, &vector);

    /* Respond to requests by workers. */
    terminated = 0;
    assign_cnt = 0;
    assigned = (int *) malloc (p * sizeof (int));

    do {

```

```

/* Get profile from worker */
MPI_Recv (buffer, dict_size, MPI_UNSIGNED_CHAR,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
src = status.MPI_SOURCE;
tag = status.MPI_TAG;
if (tag == VECTOR_MSG) {
    for (i = 0; i < dict_size; i++)
        vector[assigned[src]][i] = buffer[i];
}
/* Assign more work or tell worker to stop. */
if (assign_cnt < file_cnt) {
    MPI_Send (file_name[assign_cnt],
              strlen (file_name[assign_cnt]) + 1,
              MPI_CHAR, src, FILE_NAME_MSG, MPI_COMM_WORLD);
    assigned[src] = assign_cnt;
    assign_cnt++;
} else {
    MPI_Send (NULL, 0, MPI_CHAR, src, FILE_NAME_MSG, MPI_COMM_WORLD);
    terminated++;
}
} while (terminated < (p-1));
write_profiles (argv[RES_ARG], file_cnt, dict_size, file_name, vector);
}

void worker (int argc, char *argv[], MPI_Comm worker_comm)
{
    char *buffer; /* Words in dictionary */
    hash_el **dict; /* Hash table of words */
    int dict_size; /* Profile vector size */
    long file_len; /* Chars in dictionary */
    int i;
    char *name; /* Name of plain text file */
    int name_len; /* Chars in file name */
    MPI_Request pending; /* Handle for MPI_Isend */
    uchar *profile; /* Document profile vector */
    MPI_Status status; /* Info about message */
    int worker_id; /* Rank in worker_comm */

    void build_hash_table (char *, int, hash_el ***, int *);
    void make_profile (char *, hash_el **, int, uchar *);
    void read_dictionary (char *, char **, long *);

    /* Worker gets its worker ID number */
    MPI_Comm_rank (worker_comm, &worker_id);

    /* Worker makes initial request for work */

```

```

MPI_Isend (NULL, 0, MPI_UNSIGNED_CHAR, 0, EMPTY_MSG,
           MPI_COMM_WORLD, &pending) ;

/* Read and broadcast dictionary file */
if (!worker_id)
    read_dictionary (argv[Dict_Arg], &buffer, &file_len) ;
MPI_Bcast (&file_len, 1, MPI_LONG, 0, worker_comm) ;
if (worker_id) buffer = (char *) malloc (file_len) ;
    MPI_Bcast (buffer, file_len, MPI_CHAR, 0, worker_comm) ;

/* Build hash table */
build_hash_table (buffer, file_len, &dict, &dict_size) ;
profile = (uchar *) malloc (dict_size * sizeof (uchar)) ;

/* Worker 0 sends msg to manager re: size of dictionary */
if (!worker_id) MPI_Send (&dict_size, 1, MPI_INT, 0,
                          Dict_Size_Msg, MPI_COMM_WORLD) ;

for (;;) {
    /* Find out length of file name */
    MPI_Probe (0, File_Name_Msg, MPI_COMM_WORLD, &status) ;
    MPI_Get_count (&status, MPI_CHAR, &name_len) ;

    /* Drop out if no more work */
    if (!name_len) break;
    name = (char *) malloc (name_len) ;
    MPI_Recv (name, name_len, MPI_CHAR, 0, File_Name_Msg,
              MPI_COMM_WORLD, &status) ;
    make_profile (name, dict, dict_size, profile) ;
    free (name) ;
    MPI_Send (profile, dict_size, MPI_UNSIGNED_CHAR, 0,
              Vector_Msg, MPI_COMM_WORLD) ;
}
}

```

图 9.7 文档分类的 MPI 程序

为了区分进程间发送和接收的四种消息，我们定义四个常量来作为这些消息的标签。使用消息标签有助于书写代码，同时也允许一个进程乱序地接收另外一个进程发送过来的消息。

比如工人进程 0 发送一个准备就绪的请求给管理者。然后它读取词典，广播它，处理词典并把词典的大小发送给管理者。对管理者来说，在处理工人进程的任务分配请求前，需要分配文档特征向量所需的内存，因此，它需要先接收词典的大小，然后再接收工人进程 0 发送过来的任务分配请求。我们可以给这两个消息分配不同的消息标签，从而实现乱序接收。我们用 Dict\_Size\_Msg 作为工人进程 0 发送给管理进程的词典大小的消息标签，而用 Empty\_Msg 来标识工人进程发送给管理进程的通知准备就绪的空消息标签。



管理进程用 `FILE_NAME_MSG` 作为消息标签来给工人进程分配文档, 而工人用 `VECTOR_MSG` 作为消息标签来将文档特征向量发回给管理者。

我们还为三个命令行参数定义了常量。`DIR_ARG` 是目录结构的根目录名, `DICT_ARG` 是词典的文件名, `RES_ARG` 是用于输出特征向量文件的名称。

`main` 函数包含了区分管理者和工人的角色的代码, 以及之前所有进程都要执行的初始化代码。这包括 `MPI` 的初始化, 获得进程标号和进程组的大小。

这个函数还检查了命令行参数, 以确定用户输入了正确的参数。如果参数数目不对, 程序将结束执行。这里还包含对进程数目的检查, 以保证至少有两个进程在运行: 如果没有工人进程, 将不能处理任何文档。当这些检查都通过后, 所有进程共同创建一个只包含工人进程的新通信域。然后管理者和工人的角色被区分: 进程 0 调用 `manager` 来完成管理者的任务, 而其余的进程调用 `worker` 来充当工人的角色。

只有一个进程执行 `manager` 函数, 现在回到图 9.4 来回忆一下它的结构。管理进程开始时试图接收一个包含字典大小信息的消息; 然后它调用 `get_names` 来构造字符串数组 `file_name`, 数组中的字符串是由命令行给定的目录下所有纯文本文件的文件名。这个函数返回后, `file_cnt` 的值是待处理文件的数目。

此时管理进程需要分配存储文档特征向量所需的内存空间, 向量的数目等于文档的数目 (`file_cnt`), 而向量的长度由词条的数量所决定。因此管理进程在这个接收操作完成之前必须等待。当包含词典大小的消息到达后, 词典分配一个二维数组来存放这些特征向量。

在进入函数的主循环之前, 管理进程需要初始化一些变量: 将已结束的进程和已分配的文档数均设为 0, 它还分配一个数组用来记录当前分配给各进程的文档。

进入主循环后, 管理进程将接收从工人进程发送过来的下一条消息。如果这个消息包含的是一个文档特征向量, 管理进程就保存它; 如果还有未分配的文档, 管理进程将文档的文件名发给工人进程, 并把已分配文档的数目加一。否则, 它发送一个空文件名给工人进程, 表示工人进程可以结束了, 同时将已结束的工人数目加一。这个循环将持续运行, 直到工人进程全部结束。

管理进程在收到所有的文档特征向量后才会退出主循环, 然后它将所有的向量写入用户通过命令行指定的文件中。

现在我们来分析 `worker` 函数, 如果你还不熟悉它的基本功能, 建议你回头阅读图 9.5 中给出伪代码。每个工人进程首先得到它在工人通信域的标志号。如果工人间不需要通信, 这一步并不是必须的。但在这个算法中, 由于工人 0 负责读词典文件并将它的内容广播给其他的工人进程, 所以工人进程需要知道它们在通信域中的编号。

在调用 `MPI_Comm_rank` 后, 每个工人进程向管理进程发送准备完备消息, 所用的消息标签 `EMPTY_MSG` 将示意管理进程这是工人进程的初始请求任务消息, 而不是一个包含文档特征向量的消息。

然后, 工人 0 读取词典文件, 并把它的内容广播给其他的工人进程。注意在广播词典前, 工人 0 先广播词典的大小, 这样, 其他的工作进程可以分配足够的空间来存放词典的内容。各工人进程从广播内容中提取词条的数目, 并把它们存入哈希表, 这样可以提高文档分类操作的处理速度 (可以以常数时间确定一个词是否在词典中)。

提取词典内容的工作结束后,工人进程知道了文档特征向量的大小,在我们的实现中,特征向量中对应每个词条的元素是一个无符号字符,这样,文档和词典条目之间的相关关系可以用一个 0~255 之间的整数所表示。每个工人进程进而分配存储向量所需的空間。工人 0 还需要把词典的大小发送给管理进程。

然后工人进程进入它的主循环。它们首先探测管理进程发送的包含待处理文件名字的消息,然后分配足够的空间来存放文件名,再调用 `MPI_Recv` 来实际接收消息以得到文件名。给定文件名和哈希表,函数 `make_profile` 构造一个文档特征向量,然后工人将这个向量发回给管理进程。当工人进程收到一个长度为 0 的文件名时,说明已经没有待处理的文档了,工人进程随即退出当前的函数。

## 9.5 算法改进

在这一节中,我们来讨论如何提高我们的并行文档分类程序的性能。

### 9.5.1 按组分配文档

在一些应用中,预先分配数据给进程的策略可能会导致不平衡的负载分配,而负载不平衡会导致处理器空闲,从而降低加速比。动态分配数据给进程的策略可以平衡负载,但另一方面,它也引入了处理器间额外的通信开销,这也会降低加速比。有些时候,最好的设计是选择一条中间路线:比如我们可以在管理进程分配任务给工人进程时,每次分配  $k$  个任务,而不是一个。

### 9.5.2 流水线处理

让我们来重新审视图 9.2 给出的任务图。如果一个进程从文件服务器读取词典文件的时间与得到  $k$  个任务同样快,这种情形下没有更多的改进空间:我们已经让进程并行地构造它们的哈希表。因此,对这个子任务我们不再多考虑。

另一方面,我们可以改进文档识别和写结果文件的任务。先前的设计中,我们假设几乎所有的处理时间都花在读文档文件和构造相应的文档特征向量上,而识别纯文本文件和写结果文件被设计为串行任务。在管理进程识别所有的文件以前,不会开始处理文档。但如果识别文件消耗的时间不能被忽略,同时进程数量非常多,我们的设计其可扩展性就不会非常好(Amdahl 定律)。

现在我们来重新考虑图 9.2 中的任务图,忽略词典构造的任务。如果我们将文本识别任务进一步细分会有什么效果?我们将得到如图 9.8 所示的新的任务图。从图中可以看出,当我们标识文档 1 时,我们可以开始读文档 0。更一般地说,当我们位于处理  $i$  号文档处理过程的某一阶段时,我们可以同时处理文档  $i-1$ ,  $i-2$  的后续阶段。这种情形称为流水线处理。

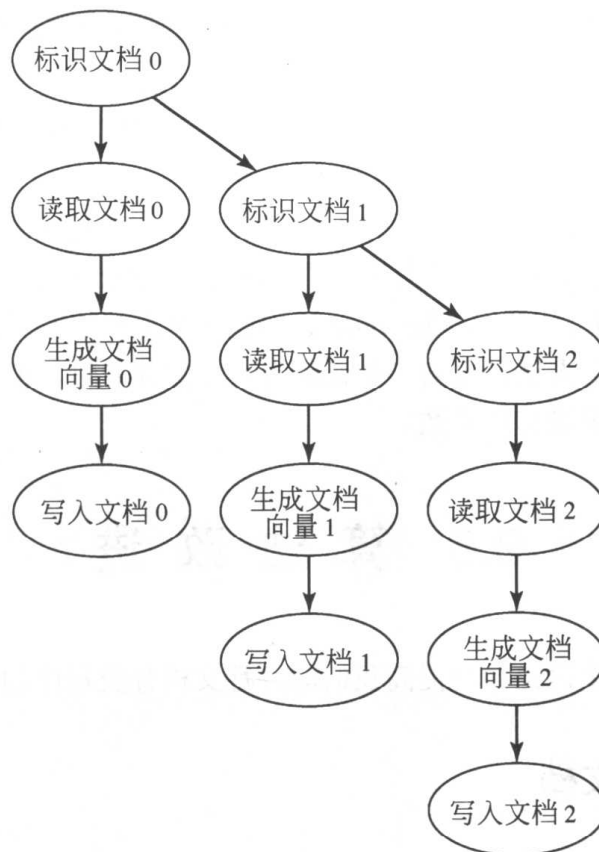


图 9.8 通过将文档标识任务划分成更小的基本任务，我们可以采用流水线的方法来处理文档。

在生成文档 0 的特征向量的同时，我们可以读取文档 1，并标识文档 2

对表现出具有功能并行性的并行算法，流水线可以显著的减少其执行时间。如图 9.9 所示，流水化的并行程序包含一个读进程，两个工人进程和一个写进程，非流水化的并行程序包含一个管理进程，三个工人进程，前者较后者具有更好的性能。

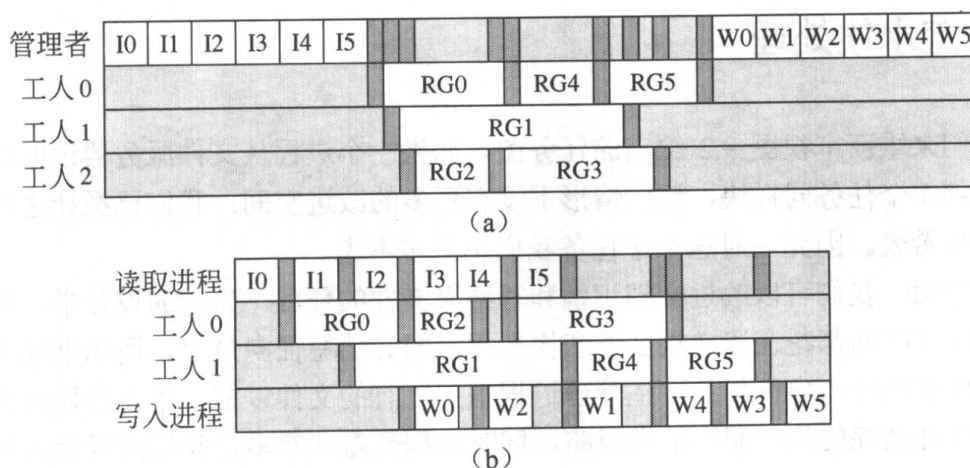


图 9.9 流水化处理可以缩短具有功能并行性并行程序的执行时间。图中  $I_i$  表示识别文件  $i$  的任务，

$RG_i$  表示读文件  $i$  并构造特征向量的任务， $W_i$  代表将文档  $i$  的特征向量写入文件的任务。

图中的灰黑色条表示通信时间。在图 (a) 中，管理进程集中识别所有的文本文件，

然后把它们分配给工人进程，最后在收集到所有的特征向量后才存入文件。

这就是我们先前开发的算法。图 (b) 是经过流水化了的版本。一个进程识别文本文件，

两个进程来读取文件并产生特征向量，而第四个进程负责将特征向量写入文件

采用流水线的方法来实现一个管理者/工人程序的缺点是它会使程序复杂化。在先前的实现中，管理进程在处理工人的请求前先标识所有的文本文件。假设我们要实现一个能使工人进程尽可能早地开始工作的管理者，也就是说，只要管理者标识出一个文件，接收到一个任务请求，它就把手头的文件发送出去。那么这就意味着，管理进程必须复用时间，以能同时标识文档并响应工人进程的请求。

基于同时完成文档标识和分派任务的初衷，下面是一种可能的实现：设  $j$  是未分配的任务数， $w$  是等待分配任务的工作进程数，如果  $j > 0$  且  $w > 0$ ，管理进程可以分配  $\min(j, w)$  个任务给工人进程。如果  $j > 0$ ，管理进程需要检查是否有来自于工人进程的消息，如果有则接收这些消息，这样就进入了  $j > 0$  且  $w > 0$  的状态。否则，管理进程需要提供更多的任务。

### 9.5.3 MPI\_Testsome 函数

为了实现上面的功能，我们需要非阻塞地检查一个或多个所期待的消息是否已经到达。MPI 库提供了 4 个函数来实现这种功能：MPI\_Test, MPI\_Testall, MPI\_Testany 和 MPI\_Testsome。这些函数都需要传入一些通过调用非阻塞接收函数而得到的 MPI\_Request 对象句柄（指针）。下面给出了 MPI\_Testsome 的用法，它最符合我们这个例子的需要。

管理进程对每个工人进程初始化一个非阻塞接收操作。它将这些操作返回的 MPI\_Request 对象指针存放在一个数组中，为了确定是否有消息从某个（或全部）工人进程到达，管理进程调用 MPI\_Testsome，这个调用返回有多少个感兴趣的消息已经到达。

函数 MPI\_Testsome 的声明如下：

```
int MPI_Testsome(int in_cnt, MPI_Request *handlearray, int *out_cnt,
                 int *index_array, MPI_Status *status_array)
```

传入参数 in\_cnt 是待查的非阻塞接收操作的数目，handlearray 中存放着 MPI\_Request 对象的句柄。out\_cnt 用于返回已经完成的通信操作的数目。index\_array 的前 out\_cnt 个元素包含着已经完成的通信操作所对应的 MPI\_Request 指针在 handlearray 中的下标，而 status\_array 的前 out\_cnt 个元素包含着这些操作所对应的状态信息。

## 9.6 本章小结

对于那些很难预先给各个任务分配工作的情况，管理者/工人模式可以很有效地保证计算过程中负载均衡分布。本章中，我们考虑了根据用户给定的词典对纯文本文件集合进行分类的问题。在这个问题中，文档的大小差异可能很大，因此有些文档地处理较为容易，而有些文档的处理则要困难一些。管理者/工人模式非常适合于这种情况。

在对这个应用的开发过程中，我们介绍了一些新的 MPI 函数的用法。比如为了便于在工人进程中广播词典，我们采用 MPI\_Comm\_split 函数来创建只含有工人的通信域。我们发现在许多情况下，通信都可以与计算或者其他的 I/O 操作相重叠。我们还介绍了非阻塞通信函数 MPI\_Isend 和 MPI\_Irecv 和辅助函数 MPI\_Wait。我们进一步讨论了工人进程在从

消息中读取实际的文件名之前先检查文件名长度的好处, `MPI_Probe` 和 `MPI_Get_count` 用来实现这样的操作。

我们考察了两种提高并程序性能的方法。在我们的例子中, 预先分配文档可能导致负载不均衡, 而如果以单个文档为单位动态分配文档则可能引入过多额外的处理器间通信。第一种提高性能的方法是在这两者之间求得折衷: 在一些应用中, 实行以较小的组为单位的动态分配可以取得最好的性能。

第二种方法是采用流水线设计来减少文本文档识别任务中的大的串行成分。如果对前面文档的处理可以和对后面文档的识别工作相重叠, 我们就可以减少总的执行时间。要实现这种类型的计算需要一个复杂得多的管理进程: 它必须能合理地分配自己的时间来同时完成文档的标识和给工人进程分配任务。为了实现这种时间复用, 管理进程需要非阻塞地测试从工人发送过来的消息, `MPI_Testsome` 函数提供了这种功能。

这一章完成了我们对 MPI 函数库的介绍。虽然整个库包括 100 多个函数, 但我们所介绍的 27 个函数已经提供了足够强大的功能, 它们对编写大部分的应用程序已经是绰绰有余了。

## 9.7 主要术语

block communication	阻塞通信
manager/worker paradigm	管理者/工人模式
SPMD programming	SPMD (单程序多数据) 程序
handle	句柄 (对象的指针)
nonblocking communication	非阻塞通信

## 9.8 参考文献

Wilkinson 和 Allen 所著的 “Parallel Programming” 很好地描述了管理者/工人模式的并行编程【115】。Carrero 和 Gelerntner 将管理者/工人模式 (他们称为主人/工人模式) 看作是 “日程式并行 (agenda parallelism)” 的一种体现, 这种并行方法所关注的是待处理的任任务【15】。

用一个向量来表示不同词语出现的频率, 虽然是一种旧的技术, 但仍然是一种最常用的文档表示法。

## 9.9 练习题

9.1 一个采用管理者/工人模式的程序中有一个管理进程,  $p-1$  个工人进程, 其中  $p \geq 2$ 。管理进程一次给工人分配一个任务。假设有  $k$  个需要完成的任务, 它们所需要的执行时间

分别为  $t_0, t_1, \dots, t_{k-1}$ 。假设分配任务和返回结果所消耗的时间都是 0。

(a) 什么情况下系统的效率达到最大? 这个效率的最大值是多少?

(b) 什么情况下系统的效率达到最小? 这个效率的最小值是多少?

9.2 解释为什么采用非阻塞发送和接收操作可以减少并行程序中的执行时间, 请给出两个理由。

9.3 在本章的文档分类程序中, `EMPTY_MSG` 标签并不是必须的, 请给出两种替代方法, 来使管理进程能够区分准备就绪的请求消息和后续的含有文档特征向量的消息。

9.4 在函数 `worker` 中, 解释为什么 `MPI_Isend` 不需要一个与之对应的 `MPI_Wait` 调用。

9.5 描述在 `manager` 函数中, 如何采用额外的非阻塞通信操作来提高性能?

9.6 在本章开发的文档分类程序中, 需要一个工人读词典并将词典的内容广播给其他的工人进程。然后所有的工人进程根据词典的内容构造哈希表。还有另外一种做法: 由一个工人读取词典, 构造哈希表, 然后将哈希表广播给其他的工人进程。解释这种做法在实际中需要解决的基本问题, 并给出一种解决办法 (假设哈希表通过链表的方式来解决哈希冲突: 即将具有相同哈希值的元素组织成一个链表)。

9.7 采用管理者/工人方模式来写一个完成矩阵-向量积的程序。管理进程需要完成下面的任务: 从文件读取向量, 将它给每个工人分发一份; 然后管理者需要从文件中读取矩阵, 并根据工人的请求将矩阵的行分配给各个工人进程。对分配的每一行, 管理进程需要回收一个结果向量的元素。当所有结果都收到后, 管理者需要将结果向量从标准输出上输出。

9.8 采用管理者/工人方式来设计一个并行程序, 求解下面方程的最小正数解:

$$f(x) = -2 + \sin x + \sin^2 x + \sin^3 x + \dots + \sin^{1000} x$$

这个根  $r$  在  $[0, 1]$  内, 而且具有惟一性。程序需要将  $[0, 1]$  分成若干个子区间, 并对每个子区间创建一个任务。每个任务计算它所对应区间的端点的函数值  $f(x)$ 。其中某个任务会发现它的端点函数值一个为负, 一个为正。算法进而在这个子区间上继续求精: 将这个区间再划分为子区间, 当子区间的大小满足小于  $10^{-11}$  时, 程序结束, 并打印出最后结果。

9.9 写一个并行程序来显示曼德勃罗集, 如图 9.10 所示。一个曼德勃罗集是复平面上的点集, 这些点满足如下的条件: 在根据某个函数递推的计算数列时, 数列的元素虽然或增或减, 但不会超过某个极限 (这种性质称为半稳定)。本例中, 我们采用下面的递推公式:

$$z_{k+1} = z_k^2 + c$$

其中  $z_0=0$ ,  $c$  是一个复数, 表示复平面上的一个点。上式中, 数列  $z$  的第  $k+1$  个值由  $z$  的第  $k$  个值及  $c$  导出。

$z$  的模是它对应的向量的长度, 比如  $z=a+bi$ , 则  $z$  的模为  $\sqrt{a^2+b^2}$ 。如果  $z$  的模大于等于 2, 数列的后续值将不再有界, 此时我们便知道  $c$  不是曼德勃罗集中的点。如果我们迭代  $n$  次后  $z_n$  依然小于 2, 我们就判定  $c$  在曼德勃罗集中。

本题要求计算一个复平面上均匀分布于正方形区域内的  $600 \times 600$  个点的曼德勃罗集, 这个正方形区域的左下角和右上角坐标分别是  $-1.5-i$  和  $1+i$ 。令  $n=1000$ , 根据定义, 如果  $z_{1000} < 2$ , 你需要将点  $c$  作为曼德勃罗集中的元素并将它显示出来。

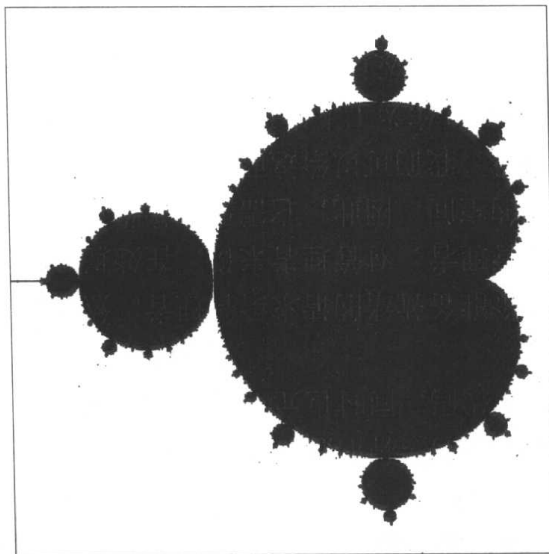


图 9.10 曼德勃罗集是分形的一个例子

在这个图中，矩形的左下角表示复数 $-1.5+i$ 。右上角表示 $0.5+i$ ，黑色的点表示集合中的点

9.10 完数指的是一类正整数，它等于它的所有正因子的和（除了它本身）。数集中的前两个完数是 6 和 28：

$$6=1+2+3$$

$$28=1+2+4+7+14$$

希腊数学家欧几里德（大约公元前 300 年）指出如果  $2^n-1$  是一个质数，那么  $(2^n-1)2^{n-1}$  是一个完数。比如  $2^2-1$  是质数，因此  $(2^2-1)2^1=6$ ，是一个完数。写一个并行程序来求出前 8 个完数。

# 第 10 章 蒙特卡洛法

O! mang a shaft at random sent  
Finds mark the archer little meant!  
And many a word, at random spoken,  
May soothe or wound a heart that's broken!  
Sir Walter Scott, The Lord of the Isles

## 10.1 概 述

蒙特卡洛法是一个通过统计采样来解决问题的算法。这个名称来自于摩纳哥的胜地——赌城蒙特卡洛。虽然这个领域的早期工作开始于 19 世纪，但蒙特卡洛法的第一个重要的应用是在二次世界大战期间原子弹的研制中。

对大于六维的任意函数的积分计算，蒙特卡洛法是惟一实用的方法。它还有许多其他的应用，包括预测道琼斯工业指数的走向，解偏微分方程，锐化卫星图片，对细胞群体建模，以及在多项式时间界里找到 NP 问题的近似解。

为了描述蒙特卡洛法，我们先来看一个物理类比的例子。我们来求  $\pi$  的值。我们知道一个直径为  $D$  的圆的面积是  $\pi D^2/4$ 。同时我们知道一个边长为  $D$  的正方形的面积为  $D^2$ 。假设我们在一个  $D \times D$  的饼盘中放置一个直径为  $D$  的饼盘，然后把这两个盘放在雨中。过几个小时后，我们取回盘，分别衡量两个盘中的水量。圆盘中的水量与两个盘总的水量之比应该接近  $\pi/4$ ：

$$\frac{\pi D^2/4}{D^2} = \frac{\pi}{4}$$

我们用随机数来进行相同的估计（这个例子说明了蒙特卡罗法的原理，但需要记住当维数较小的时候数值积分是一种更好的策略）。图 10.1 描述了在一个单位正方形中嵌入一个半径为 1 的 1/4 圆的情况。一个半径为 1 的圆的面积是  $\pi$ ，因此 1/4 圆的面积是  $\pi/4$ 。我们来产生一系列数对  $(x, y)$ ，其中  $x$  和  $y$  都是区间  $[0, 1]$  上的均匀分布的随机变量。每对数对应着单位正方形内的一个点。我们对落在 1/4 圆区域内的点所占的比例  $f$  进行计算，也就是那些满足  $x^2 + y^2 \leq 1$  的点。因为  $f \approx \pi/4$ ，我们知道  $4f \approx \pi$ 。

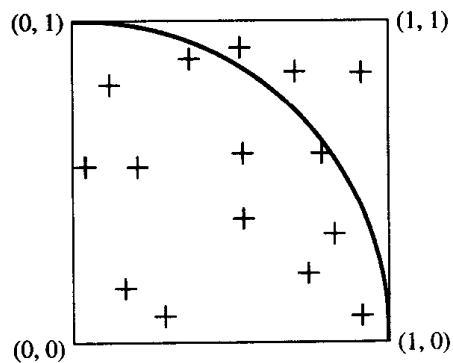


图 10.1 用蒙特卡洛法来估计  $\pi$  的值。图中 1/4 圆的面积是  $\pi/4$ 。在这个图中，在单位正方形中随机选择的 15 个点中有 12 个落在这个 1/4 圆中。因此  $\pi/4$  的估计值是 0.8，也就是估计  $\pi$  为 3.2



应用上面的方法,我们实现了一个用来计算  $\pi$  的 C 程序,如图 10.2 所示。表 10.1 显示了计算得到的  $\pi$  值与  $\pi$  的真实值之间的绝对误差随着样本数目  $n$  的增加而慢慢减小(给出估计值  $e$  和精确值  $a$ , 绝对误差定义为  $|e-a|/a$ )。对于蒙特卡洛法,可以用函数  $1/(2\sqrt{n})$  来近似估计它的绝对误差。

表 10.1 采样点的数量增加时,估计解的精确度也提高

采样数	估计的 $\pi$ 值	差错	$1/(2\sqrt{n})$
10	2.40000	0.23606	0.15811
100	3.36000	0.06952	0.05000
1,000	3.14400	0.00077	0.01581
10,000	3.13920	0.00076	0.00500
100,000	3.14132	0.00009	0.00158
1,000,000	3.14006	0.00049	0.00050
10,000,000	3.14136	0.00007	0.00016
100,000,000	3.14154	0.00002	0.00005
1,000,000,000	3.14155	0.00001	0.00002

```

/*
 * This C program uses the Monte Carlo method to
 * compute the value of pi.
 */
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    int count; /* Points inside circle */
    int i;
    int n; /* Number of samples */
    double pi; /* Estimate of pi */
    unsigned short xi[3]; /* Random number seed */
    double x, y; /* Point's coordinates */
    if (argc != 5) {
        printf ("Correct command line: "),
        printf ("%s <# samples> <seed0> <seed1> <seed2>\n", argv[0]);
        return -1;
    }
    n = atoi(argv[1]);
    for (i = 0; i < 3; i++)
        xi[i] = atoi(argv[i+2]);
    count = 0;
    for (i = 0; i < n; i++) {
        x = erand48(xi);
        y = erand48(xi);
        if (x*x+y*y <= 1.0) count++;
    }
    pi = 4.0 * (double) count / (double) n;
    printf ("Samples: %d Estimate of pi: %7.5fn", n, pi);
}

```

图 10.2 用蒙特卡洛法计算  $\pi$  的 C 程序。解的精度依赖于采样的数量和伪随机数产生器的质量

### 10.1.1 为什么蒙特卡洛法能奏效

回顾一下均值定理:

$$I = \int_a^b f(x) dx = (b-a)\bar{f}$$

其中  $\bar{f}$  是函数  $f(x)$  在区间  $[a, b]$  上的均值, 如图 10.3 所示。

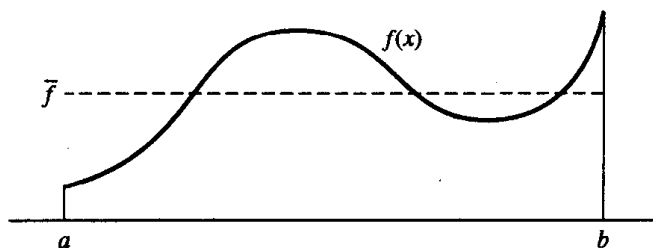


图 10.3 根据中值定理我们知道曲线  $f(x)$  下的区域面积等于  $\bar{f}$  的区域面积,  
 $\bar{f}$  等于  $f(x)$  在区间  $[a, b]$  上的中值

蒙特卡洛法通过对区间  $[a, b]$  上均匀分布的  $n$  个随机点的值  $f(x_i)$  进行计算的方式来估计  $I$  的值。它们的期望值  $\frac{1}{n} \sum_{i=0}^{n-1} f(x_i)$  就是  $\bar{f}$ 。因此有:

$$I = \int_a^b f(x) dx = (b-a)\bar{f} \approx (b-a) \frac{1}{n} \sum_{i=0}^{n-1} f(x_i)$$

让我们来看如何将这个结论用于已经讨论过的计算  $\pi$  的算法。我们知道半径为 1 的  $1/4$  圆的面积与边长为 1 的正方形的面积比为  $\pi/4$ 。考虑图 10.4(a) 中的曲面。当  $x^2 + y^2 \leq 1$  时曲面的高为 1, 否则为 0。将一个计数器初始化为 0, 然后产生位于单位正方形内的随机点  $(x, y)$ , 如果  $x^2 + y^2 \leq 1$ , 那么将计数器加 1, 否则不加。这样我们就对这个曲面进行取样。

如果在  $n$  次取样后, 我们将计数器的值除以  $n$ , 我们就得到了一个平均值。这个均值的期望是  $\pi/4$ , 如图 10.4b 所示。因此将这个均值乘以 4 就得到了  $\pi$  的蒙特卡洛估计值。

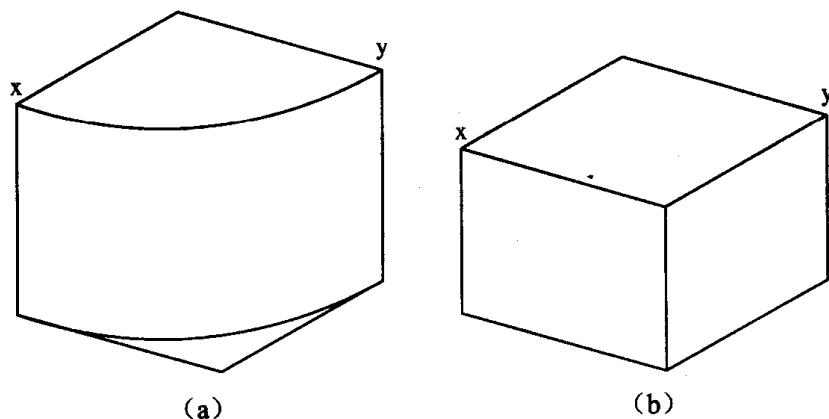


图 10.4 由均值定理我们知道, 这些曲面包围的体积相同: (a) 在边界为  $0 \leq x, y \leq 1$  的正方形中, 当  $x^2 + y^2 \leq 1$  时曲面的高为 1, 否则为 0; (b) 在由边界  $0 \leq x, y \leq 1$  所定义的正方形中, 曲面的高度为  $\pi/4$

重要的是, 用蒙特卡洛估计的  $I$  值, 其绝对误差正比于  $1/\sqrt{n}$ 。这个收敛速率与被积函

数的维数没有关系。这与确定性的数值积分方法比如辛普生规则形成了鲜明的对比：后者的收敛率随维数增加而降低。这就是为什么对六维以上的被积函数进行积分时，蒙特卡洛法要优于确定性的数值积分方法。

### 10.1.2 蒙特卡洛法与并行计算

蒙特卡洛算法通常很容易并行化。许多并行蒙特卡洛程序的处理器间通信量都可以被忽略。这种情况下，可以用  $p$  个处理器来将估计值求解加速  $p$  倍，或者将估计误差降低  $\sqrt{p}$ 。用另外一种方式来说，可以用  $p$  个处理器来将结果的方差降低  $p$  倍。

当然，上面的结论都基于一个共同的假设：随机数在统计上具有独立性。实现并行蒙特卡洛法的一个主要挑战就是如何开发一个好的并行随机数生成器。一个为大多数人接受的论断是：有一半的超级计算机的计算能力都用来进行蒙特卡洛计算。因此理解怎样才是一个好的并行随机数生成器就变得非常重要。所以，我们先对串行随机数生成器进行一个简短的回顾。

## 10.2 串行随机数生成器

从技术上来说，在今天的计算机上看到的随机数生成器都是伪随机数生成器，因为它们的操作都是确定性的，因此他们产生的序列都是可预测的。在最好的情况下，这些序列是对真正的随机序列的一个合理的近似。然而，由于“伪随机数产生器”这个术语太长太绕口，我们决定还是使用简单的词语。在本章的其余部分，“随机数产生器”所指的是“伪随机数产生器”。

Coddinton【17】指出一个理想的随机数产生器所产生的序列应当具备以下 10 个性质：

- 它是均匀分布的，即所有可能的数的出现可能性都相同；
- 各数之间不相关；
- 不出现环，也就是说，数字不会出现重复；
- 它满足所有随机性的统计测试；
- 可以再生：序列可以反复产生；
- 与机器无关。也就是说，在所有的计算机上，这个产生器将产生相同的序列；
- 可以通过改变一个初始“种子”值的方法来改变它；
- 可以很容易的将它分成许多相互独立的子序列；
- 它可以很快的产生随机数；
- 产生器只消耗有限的计算机存储空间。

没有一个随机数生成器能满足上述的所有要求。比如，由于计算机依赖于有限精度的算术运算，因而随机数生成器只能有有限个状态，所以最终它会回到一个已经到过的状态，运行到这一步，它已构成了一个环，之后它产生的数开始与原来的相同。一个随机数生成器的周期就是这个环的长度。

同样，如果我们要求这个序列是可以重复产生的，这些数就不可能是完全不相关的。

我们可以期待的最好情况是这种相关性非常小,以至于对计算的结果不会产生明显的影响。

通常情况下,我们要在随机数生成器的速度与产生的数的质量之间做权衡。由于在程序中产生随机数所需要的时间通常只占总计算时间的很小一部分,速度的重要性比质量要低得多。

接下来的章节中,我们来考虑两类重要的随机数生成器:线性同余与滞后的斐波那契产生器。

### 10.2.1 线性同余法

线性同余法已经有 50 多年历史,而且仍然是最常用的一种方法。线性同余随机数生成器根据下面的公式产生一个随机整数序列  $X_i$ :

$$X_i = (a \times X_{i-1} + c) \bmod M$$

其中  $a$  称为乘数,  $c$  称为偏移量,  $M$  称为模。在一些实现中  $c=0$ 。这时的生成器称为乘法同余生成器。为了保证产生序列有较长的周期和较好的随机性,这三个数必须仔细挑选。最大周期设为  $M$ ,那么对一个 32 比特的整数而言最大周期是  $2^{32}$ ,即大约 40 亿。对如今的每秒执行数十亿条指令的计算机来说,这个周期太小了。一个满足质量要求的生成器至少要有 48 位。

这个生成器产生的序列的值依赖于初始值  $X_0$ ,这个值通常称为种子,它通常由用户来提供。

线性同余法也可以用来产生浮点随机数。由于生成器可以产生  $0 \sim M-1$  之间的整数,将它们除以  $M$  就和产生区间  $[0,1]$  上的浮点数  $x_i$ 。

对于线性同余法的缺点我们已经有很清楚的认识了:它所产生的数的最低位是相关的(尤其是当模  $M$  是 2 的幂时)。如果你在一个  $k$  维超立方体中产生一系列点  $(x_i, x_{i+1}, \dots, x_{i+k-1})$ ,你会得到一个格子(lattice)结构【79】。当随机数的维数增加时,这个问题变得越发明显,它会影响高维模拟的质量。

尽管如此,精度在 48 位以上的线性同余产生器,如果对参数精心挑选,它可以“在所有已知的应用上都表现得很好,至少在串行计算机上如此”【17】。

### 10.2.2 滞后形斐波那契生成器

滞后形斐波那契生成器可以产生具有非常长周期的序列,而且生成速度也能得到保证,因而它们开始逐渐流行起来了。它根据下面的公式来产生一个序列  $X_i$ :

$$X_i = X_{i-p} \star X_{i-q}$$

其中  $p$  和  $q$  是滞后系数,  $p > q$ ,而  $\star$  是任意的二元算术操作。常用的  $\star$  操作包括模  $M$  加法、模  $M$  减法、模  $M$  乘法和按位的异或操作。对加法和减法,  $X_i$  可以是整数或者浮点数,如果序列包含浮点数,则  $M=1$ 。如果  $\star$  操作是乘法,那么序列必须完全由奇数构成。

与线性同余产生器只需要一个种子不同,滞后的斐波那契产生器需要  $p$  个种子  $X_0, X_1, \dots, X_{p-1}$ 。精心挑选  $p, q, M$  和各个种子,可以产生一个具有很长周期及良好随机性

的序列。如果  $X_i$  有  $b$  位, 那么异或操作可以达到的最大周期为  $2^p-1$ , 加法和减法可以达到的最大周期为  $(2^p-1)2^{b-1}$ , 乘法为  $(2^p-1)2^{b-3}$ 。增大最大滞后系数  $p$  会增加所需的存储量, 但会增大最大周期。

C 语言函数 `random` 就是一个默认滞后系数为 31 的加法滞后式斐波那契生成器。Coddington 指出, 这个滞后系数太小了, 他建议  $(p, q)$  至少应设为 (1279, 1063)。

## 10.3 并行随机数产生器

并行蒙特卡洛法依赖于我们是否能产生大量高质量随机数序列。除了上一节中讨论的串行随机数生成器应具备的性质外, 一个理想的并行随机数产生器还需要具有下列性质:

- 不同序列的数之间没有相关性;
  - 可扩展性: 也就是可以适用于大量的进程同时运行, 每个进程产生各自序列的情况;
  - 局部性: 一个进程能够在不进行进程间通信的情况下产生一个新的随机数序列。
- 这一节中我们来讨论 4 种由串行随机数产生器构造并行随机数产生器的技术。

### 10.3.1 管理者-工人方法

Group 等人【45】描述了一种构造并行随机数产生器的方法, 它具有管理者-生产者的结构。管理者产生随机数, 并将这些随机数分发给需要它们的工人进程。这种方法有两个主要的缺陷。

一些随机数产生器产生的序列在长距离上有相关性。由于每个进程都在同一个序列上进行采样, 因此有可能使得这种原来序列中的长距离的相关性转化为并行序列中的短距离相关性。

这种管理者-工人结构无法扩展到任意数目的进程, 因此缺乏可扩展性。这样, 在随机数的产生速度与随机数的使用速度之间可能很难找到平衡。它显然没有不具备局部性, 与此相反, 它所需要的通信相当频繁。

上述的缺陷都很致命, 因此这种方法已经不再流行。下面我们来讨论由每个进程独立产生随机数序列的方法。

### 10.3.2 蛙跳方法

蛙跳方法与轮循地将数据分发给任务的方法类似。假设我们的并行蒙特卡洛法由  $p$  个进程执行, 所有的进程使用相同的串行随机数产生器。编号为  $r$  的进程从  $X_r$  开始, 每隔  $p$  个数取走一个数:

$$X_r, X_{r+p}, X_{r+2p}, \dots$$

图 10.5 给出了编号为 2 的进程所使用的序列中的元素。7 个进程并行执行, 每个进程产生自己的随机数序列。



图 10.5 7 个进程中的 2 号进程采用蛙跳方法来产生随机数

我们可以很容易在一个线性同余生成器的基础上使用蛙跳方法。跳过  $p$  个元素可以通过替换  $a$  和  $c$  来完成：将  $a$  换成  $a^p$ ，而将  $c$  换成  $c(a^p - 1) / (a - 1)$ 。Makino 在【77】中给出了用蛙跳方法来改造滞后的斐波那契产生器的方法。

蒙特卡洛算法经常需要产生多维随机数。比如，在 10.1 节给出的估计  $\pi$  的例子中，我们产生的是一个坐标对。如果我们想让并行算法产生与串行算法相同的坐标对，需要产生  $(X_{2r}, X_{2r+1}, X_{2r+2p}, X_{2r+2p+1}, \dots)$ ，而不是  $(X_r, X_{r+p}, X_{r+2p}, X_{r+3p}, \dots)$ 。这是一种最直接的对蛙跳方法的修改方法，如图 10.6 所示。

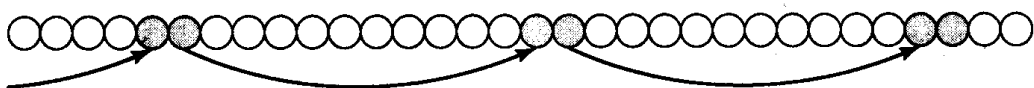


图 10.6 6 个进程中的 2 号进程使用修改过的蛙跳方法产生随机数对

蛙跳方法的缺点是，即使原来的串行随机数产生器的序列元素有很低的相关性，对于某些  $p$  值，蛙跳方法产生的子序列仍会产生相关。当我们采用线性同余产生器， $p$  是 2 的幂并且模  $M$  也是 2 的幂时，这尤其可能发生。即使不是这种情形，蛙跳方法也可能将原来的长距离相关变成并行序列内的短距离相关。

蛙跳方法的另外一个缺点是它不支持动态生成新的随机数序列。

### 10.3.3 序列分割

序列分割类似于对数据进行块状划分。假设一个随机数生成器的周期为  $P$ ，序列的前  $P$  个元素被分成大小相同的块，每个进程分一个块，如图 10.7 所示。



图 10.7 在序列分割方法中，每个进程分配到一组在原来序列中连续的随机数

这种方法有个缺点，每个进程必须从序列中与自己对的位置开始。这可能需要很长的时间。但另外一方面，这个操作只需要在算法的初始化阶段进行一次。在这之后，每个进程按顺序逐个产生元素。

模为 2 的幂的线性同余生成器具有长距离的相关性，由于不同进程所产生的序列在原序列中的距离很远，因此不同进程的序列之间可能会有相关性。

通过修改算法，序列分割可以支持动态新序列的产生。比如，一个进程可以通过将自己的随机数分半的方法来产生一个新序列。

### 10.3.4 参数化

第四种实现并行随机数生成器的方法是在每个进程中都运行一个串行的随机数生成器，但确保每个生成器都产生不同的随机数序列。这可以通过给每个生成器不同的初始化

参数来实现, 如图 10.8 所示。

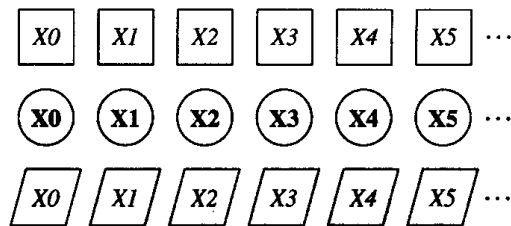


图 10.8 通过在初始化串行随机数生成器的时候给出不同的参数, 通常可以使每个进程拥有自己的序列

具有不同偏移量的线性同余生成器可以产生出不同的序列。Percus 和 Kalos 给出了一种选择偏移量的方法, 这种方法可以很好的生成多达 100 个随机序列【91】。

滞后的斐波那契生成器特别适合这种方法。给每个进程提供不同的滞后常数使得每个进程可以产生不同的随机数序列。显而易见, 滞后常数内部或者滞后常数之间的相关性是致命的。一种用来产生这些初始化常数的方法是采用另外一个滞后的斐波那契产生器来产生所需的种子。这个过程可以用蛙跳方法, 或者是序列分割来确保它们的初始化常数取不同的值。

滞后的斐波那契生成器能产生的不同序列的数量是非常惊人的【82】。比如 SPRNG 库中所提供的默认的乘法滞后的斐波那契产生器能支持大约  $2^{1008}$  个不同的序列, 从而在并行程序执行的过程中可以产生出足够多的新序列【83】。

## 10.4 其他的随机数分布

目前我们讨论的随机数产生器都遵循均匀分布, 可是有时候我们需要产生出满足其他分布的随机数。

### 10.4.1 逆分布累积分布函数变换

我们用  $u$  表示  $[0, 1]$  上均匀分布的一个采样。

假设我们想产生一个满足概率密度函数  $f(x)$  的随机变量, 如果我们能确定它的分布函数  $F(x)$  并对它求逆, 则  $F^{-1}(u)$  就是一个满足概率密度函数  $f(x)$  的随机变量, 如图 10.9 所示。

下面我们用一个指数分布的例子来说明这种方法。

**指数分布:** 放射性原子的衰变, 中子在固体中碰到原子前的运动距离, 以及服务中心里下一个顾客到达所需的时间, 都可用满足指数概率分布的随机变量来对其进行建模。

期望值为  $m$  的指数概率密度函数为  $f(x) = (1/m)e^{-x/m}$ 。将这个函数积分, 可以得到它的概率分布函数  $F(x) = 1 - e^{-x/m}$ 。对  $F(x)$  求逆, 得到:  $F^{-1}(u) = -m \ln(1-u)$ 。由于  $u$  在 0 与 1 之间呈均匀分布, 因此  $u$  和  $1-u$  没有什么区别, 所以  $F^{-1}(u) = -m \ln u$  是一个满足指数分布的随机变量, 且期望值为  $m$ 。

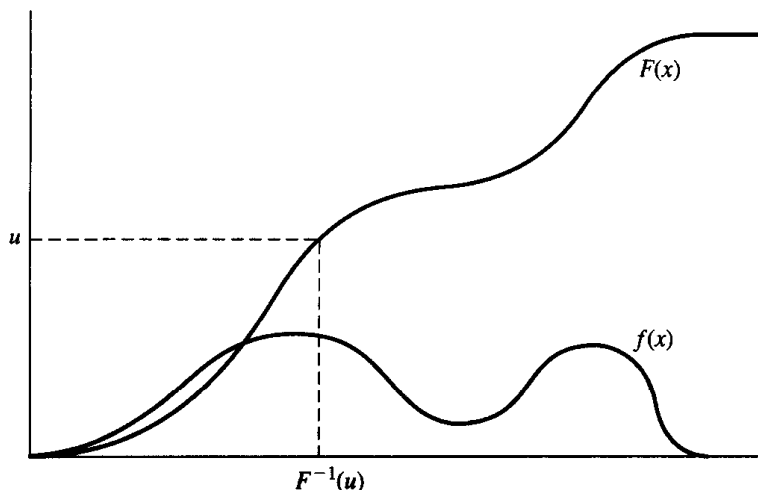


图 10.9 给出一个概率密度函数  $f(x)$ ，它的概率分布函数是  $F(x)$ ， $u$  是一个均匀分布的采样，则  $F^{-1}(u)$  就是  $f(x)$  的一个采样

### 例 1

为一个期望值为 3 的指数分布产生 4 个样本。

解：

我们先产生 4 个满足均匀分布的样本：

0.540 0.619 0.462 0.095

对每个值取自然对数并乘以 -3：

$-3\ln(0.540)$   $-3\ln(0.619)$   $-3\ln(0.462)$   $-3\ln(0.095)$

这样就得到了期望值为 3 的指数分布的 4 个样本：

1.850 1.440 2.317 7.072

### 例 2

一个模拟的时间步长为 1 秒。某个特殊事件发生的概率满足期望值为 5 的指数分布。那么下一个时间步里发生这个事件的概率是多少？我们如何确定在下一个时间步里这个事件是不是会发生？

解：

下一个时间步里这个事件的发生概率为 1/5。为了决定在下一个时间步内这个事件是否会发生，我们用  $[0, 1]$  上的均匀分布来产生一个随机数，如果它比 1/5 小，这个事件判定为发生。

## 10.4.2 Box-Muller 变换

对标准正态分布函数（高斯分布），我们无法给出其分布函数的逆函数：

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

幸运的是，Box-Muller 变换可以用来从一对均匀分布的变量  $u_1$  和  $u_2$  得到一对标准正态分布变量  $g_1$  和  $g_2$  【65】：



```

repeat
     $v_1 \leftarrow 2u_1 - 1$ 
     $v_2 \leftarrow 2u_2 - 1$ 
     $r \leftarrow v_1^2 + v_2^2$ 
until  $r > 0$  and  $r < 1$ 
 $f \leftarrow \sqrt{-2 \ln r / r}$ 
 $g_1 \leftarrow f v_1$ 
 $g_2 \leftarrow f v_2$ 

```

**例 1**

生成期望值为 0, 标准方差为 1 的正态分布的 4 个样本如下。

$u_1$	$u_2$	$v_1$	$v_2$	$r$	$f$	$g_1$	$g_2$
0.234	0.784	-0.532	0.568	0.605	1.290	-0.686	0.732
0.824	0.039	0.648	-0.921	1.269			
0.430	0.176	-0.140	-0.648	0.439	1.935	-0.271	-1.254

解:

从均匀随机样本 0.234 和 0.784 我们可以得到两个正态分布的样本 -0.686 与 0.732。下一对均匀分布样本 0.824 和 0.039 下  $r > 1$ , 因此我们必须舍弃这对样本, 再产生一对。0.430 和 0.176 产生的正态样本为 -0.271 与 -1.254。

**例 2**

生成期望值为 8, 标准方差为 2 的正态分布的 4 个样本。

解:

我们对 Box-Muller 变换进行修改, 将赋值操作:

$$g_1 \leftarrow f v_1$$

替换为:

$$g_1 \leftarrow 2f v_1 + 8$$

同样对  $g_2$  作相应的修改, 算法产生的变量如下所示。

$u_1$	$u_2$	$v_1$	$v_2$	$r$	$f$	$g_1$	$g_2$
0.017	0.262	-0.965	-0.475	1.158			
0.832	0.743	0.663	0.486	0.676	1.075	9.426	9.045
0.670	0.439	0.339	-0.122	0.130	5.602	11.800	6.630

由均匀分布的样本 0.017 和 0.262 得到的  $r$  值太大, 我们必须舍弃这个样本。0.832 与 0.743 得到正态样本 9.462 和 9.045, 而样本 0.670 与 0.439 得到正态样本 11.800 和 6.630。

你可以用 Box-Muller 变换来得到一个能返回单独的正态分布样本的函数。在对这个函数的第 1 次、3 次、5 次等调用时, 它进行 Box-Muller 变换, 存储  $g_2$ , 返回  $g_1$ ; 在第 2 次、第 4 次、第 6 次等调用时, 这个函数返回前一次调用所存储的  $g_2$ 。

### 10.4.3 拒绝法

拒绝法由 John von Neumann 首先提出, 运用拒绝法我们对不能积分和不能通过解析方法求逆的概率密度函数  $f(x)$  进行处理从而得到它的样本。假设我们可以为另一个概率密

度函数  $h(x)$  产生样本, 而且我们可以找到一个常数  $\delta$ , 使得对所有  $x$  有  $f(x) \leq \delta h(x)$ , 如见图 10.10 所示, 那么我们可以用下面的方法来产生  $f$  的样本: 我们先产生  $h$  的一个样本  $x_i$ , 和另外一个满足均匀分布的样本  $u_i$ , 如果有  $u_i \delta h(x_i) \leq f(x_i)$ , 那么我们接受  $x_i$  做为  $f(x)$  的样本, 并将它返回; 否则, 我们对另外一对  $x_i$  和  $u_i$  进行测试, 这个过程会一直重复直到产生所需要的样本。

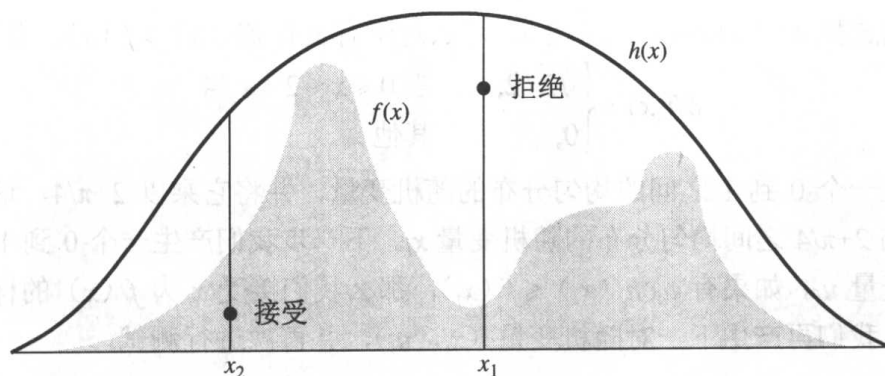


图 10.10 拒绝法使得我们可以从一个概率密度函数产生样本。我们先从概率密度函数  $h(x)$  产生样本  $x_i$ , 从  $(0,1)$  上的均匀概率密度函数产生样本  $u_i$ 。在这个图中,  $u_1=0.8$ ,  $u_1 \delta h(x_1) \leq f(x_1)$ , 所以我们拒绝样本  $x_1$ 。另外一个样本  $u_2=0.15$ ,  $u_2 \delta h(x_2) \leq f(x_2)$ , 所以我们接受样本  $x_2$

$(x_i, u_i \delta h(x_i))$  点对曲线  $\delta h(x)$  下的区域进行均匀采样。由于我们只接受那些在曲线  $f(x)$  下的点, 最终得到的序列  $x_i$  符合概率密度函数  $f(x)$ 。

当  $f(x)$  与  $\delta h(x)$  的相对误差很小时, 拒绝方法能最好地工作。两者曲线之间区域越大, 所选的随机数被拒绝的可能性就越大, 这会使整个过程变慢。当维数增加时, 拒绝法的效率会急剧下降。比如, 对一个一维积分, 假设 75% 的随机数被接受。如果当维数增加时, 单个维度上的效率保持不变, 那么对一个 6 维积分, 它的效率是  $(0.75)^6$ , 即大约 18%。

#### 例

一个随机变量具有下面的概率密度函数:

$$f(x) = \begin{cases} \sin x, & \text{当 } 0 \leq x \leq \pi/4; \\ (-4x + \pi + 8)/(8\sqrt{2}), & \text{当 } \pi/4 < x \leq 2 + \pi/4; \\ 0, & \text{其他} \end{cases}$$

这个概率密度函数如图 10.11 所示。

#### 解:

我们可以用拒绝法来为这个分布产生随机变量。首先我们需要找出  $\delta$  和  $h(x)$ , 以使得对所有的  $x$  有  $f(x) \leq \delta h(x)$ 。注意当  $x$  的值在 0 到  $2 + \pi/4$  之间时, 概率密度函数大于 0, 并且它有最大值  $\sqrt{2}/2$ , 我们选择均匀分布的概率密度函数来作为  $h(x)$ :

$$h(x) = \begin{cases} 1/(2 + \pi/4), & \text{当 } 0 \leq x \leq 2 + \pi/4 \\ 0, & \text{其他} \end{cases}$$

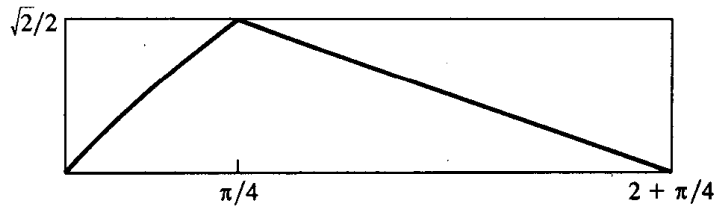


图 10.11 用一个均匀分布的随机变量和拒绝法来产生一个分段的概率密度函数的样本

如果我们选择  $\delta = (2 + \pi/4) (\sqrt{2}/2)$ , 那么对所有  $x$  有  $\delta h(x) \geq f(x)$ , 有下面的公式:

$$\delta h(x) = \begin{cases} \sqrt{2}/2, & \text{当 } 0 \leq x \leq 2 + \pi/4 \\ 0, & \text{其他} \end{cases}$$

我们产生一个 0 到 1 之间的均匀分布的随机变量, 并将它乘以  $2 + \pi/4$ , 这样我们就得到了一个 0 到  $2 + \pi/4$  之间均匀分布的随机变量  $x_i$ 。下一步我们产生一个 0 到 1 之间的均匀分布的随机变量  $u_i$ 。如果有  $u_i \delta h(x_i) \leq f(x_i)$ , 那么我们接受  $x_i$  为  $f(x)$  的样本, 并将它返回; 否则, 我们再产生下一对随机变量  $(x_i, u_i)$ , 并再次进行测试。

$x_i$	$u_i$	$u_i \delta h(x_i)$	$f(x_i)$	Outcome
0.860	0.975	0.689	0.681	Reject
1.518	0.357	0.252	0.448	Accept
0.357	0.920	0.650	0.349	Reject
1.306	0.272	0.192	0.523	Accept

## 10.5 应用示例

这一节中将考察 5 个应用, 以介绍蒙特卡洛法在各个领域中的应用概况。

### 10.5.1 中子输运

我们来考虑一个经过简化的两维中子输运模型, 如图 10.12 所示。一个中子源向一个厚度为  $H$ , 无限高的均质板发射中子。一个中子可能被这个板反射, 吸收或者击穿。我们希望计算这些事件发生的频率, 并研究它们和厚度  $H$  的函数关系。

我们用两个常数用来描述中子与板的相互作用。一个是捕获横截面  $C_c$ , 另一个是散射横截面  $C_s$ 。总的横截面是  $C = C_c + C_s$ 。

一个中子在与一个原子发生相互作用前在板中运动的距离  $L$  服从期望值为  $1/C$  的指数分布。如我们在前一节中所见, 如果  $u$  是一个  $[0, 1]$  上的均匀分布随机变量, 那么公式:

$$L = -\frac{1}{C} \ln u$$

是一个满足要求的指数分布的随机变量。

当一个中子在板中与一个原子发生相互作用, 被反弹的概率是  $C_s/C$ , 而被吸收的概率是  $C_c/C$ 。我们可以用一个  $[0, 1]$  上的均匀分布随机变量来决定一个中子-原子发生相互作用的情形。

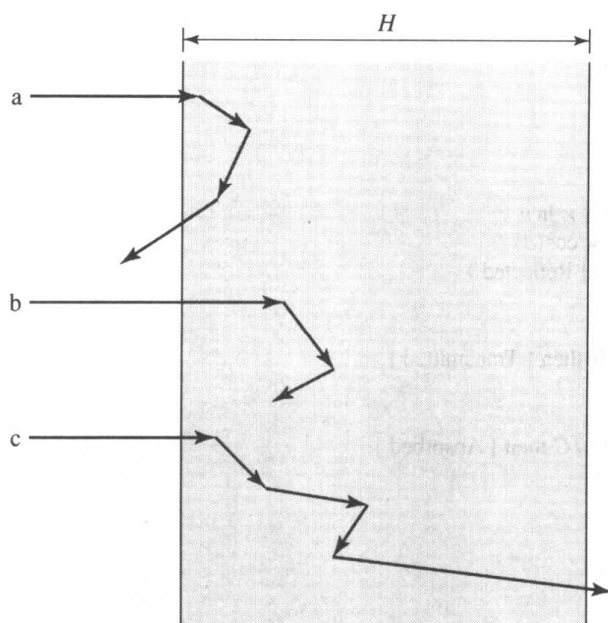


图 10.12 一个进入均匀介质的中子可能会：(a) 被反射；(b) 被吸收；(c) 穿透

如果中子发生散射，它在任何方向的移动具有相同的概率。因此它的新运动方向（用弧度表示）可以表示为  $0$  到  $\pi$  之间的一个均匀分布随机变量（由于板的高度无限，所以我们不必区分上和下）。给定方向  $D$ ，中子在发生碰撞之间在  $x$  方向运动的实际距离为  $L\cos D$ 。

对中子运动的仿真会一直进行，直到下面的事件发生：

- (1) 中子被一个原子吸收；
- (2) 中子的  $x$  坐标小于  $0$ ，表示中子被板反射了；
- (3) 中子的  $x$  坐标大于  $H$ ，表示中子穿透了板。

用蒙特卡洛法对中子输运进行仿真的伪代码在图 10.13 中给出。注意 `while` 循环的每次迭代所对应的时间间隔并不一样，仿真所基于的是从一个事件（一次相互作用）到下一个事件。这种不断增长的伪时间称为蒙特卡洛时间。

$C$ —mean distance between neutron/atom interactions is  $1/C$ （中子/原子作用之间的平均距离）

$C_s$ —scattering component of  $C$ （ $C$  的散射部分）

$C_a$ —absorbing component of  $C$ （ $C$  的吸收部分）

$H$ —thickness of plate（板的厚度）

$L$ —distance neutron travels before collision（碰撞前中子移动的距离）

$d$ —direction of neutron（measured in radians between  $0$  and  $\pi$ ）中子的方向（由  $0$  到  $\pi$  之间的弧度值表示）

$u$ —uniform random number（正态随机数）

$x$ —position of particle in plate（ $0 \leq x < H$ ）板中粒子的位置）

$n$ —number of samples（样本数）

$a$ —true while particle still bouncing（当粒子处于跳跃状态的时候为真）

$r, b, t$ —counts of reflected, absorbed, transmitted neutrons（被反射、吸收和穿透的中子数）

```

begin
  r, b, t ← 0
  for i ← 1 to n do
    d ← 0
    x ← 0
    a ← true
    while a do
      L ← -(1/C) × ln u
      x ← x + L × cos(d)
      if x < 0 then { Reflected }
        r ← r + 1
        a ← false
      else if x ≥ H then { Transmitted }
        t ← t + 1
        a ← false
      else if u < Cc/C then { Absorbed }
        b ← b + 1
        a ← false
      else
        d ← u × π
      endif
    endwhile
  endfor
  print r/n, a/n, t/n
end

```

图 10.13 用蒙特卡洛法对 neutron 输运进行仿真的伪代码

## 10.5.2 二维板上一个点的温度

给定一个均质的薄板。我们希望能计算出板上某个特定点的稳态温度。板的顶部和底部绝缘，而除板的边缘外，板上任意点的温度完全由它周围的点的温度决定。板的边缘温度固定。

板内的温度分布可以用 Laplace 方程来描述： $\nabla^2 T=0$ ，它表示一个点的温度是它周围点的温度的平均值。

一种求解 Laplace 方程的数值方法是对问题进行离散化，即将板划分为一个二维网格。这种情形下，一个点的温度是它左边，右边，上边和下边四个点温度的均值（我们可以想像它为东、南、西和北 4 个方向）。

用蒙特卡洛法我们可以求解一个特定点  $S$  的温度。方法如下：随机选择它 4 个邻点中的一个，将它的温度加到一个累加器中，当我们完成  $n$  次随机采样后，我们将这个累加和除以  $n$ ，就得到了  $S$  的温度。这个平均值的期望值为  $(T_n + T_s + T_e + T_w) / 4$ 。

当然，我们并不知道邻点的温度，但我们可以用相同的方法来求出它们的温度。递归地调用这个方法，我们便完成了板上的一个随机游走。这个递归和随机游走过程会最终结束，因为板的边缘温度是已知的。

图 10.14 所示是蒙特卡洛算法的结果。我们从  $S$  开始，然后随机选择一个方向来移动（北、南、西或者东）。这个随机移动一直持续到我们碰到板的边为止。在这个点我们将它的温度加到累加器，然后继续迭代。每次循环中我们也能确定到目前为止所有的随机游走遇到的边的平均温度。当这个平均温度收敛时算法结束。

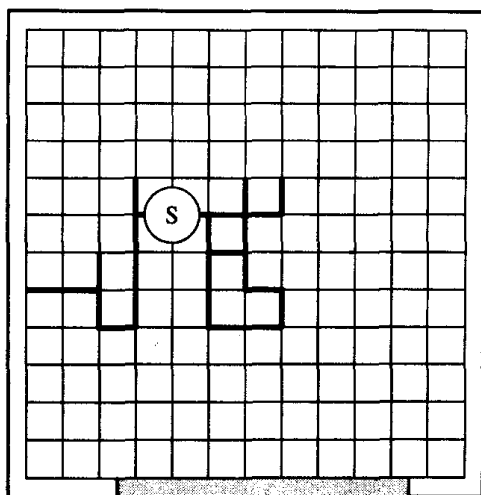


图 10.14 采用随机游走来估计板上  $S$  点的温度。边缘温度固定。与 U 形的白色条相接触的边缘点的温度为 0，而和灰色条相接触的边缘点温度为 100。

从  $S$  开始的随机游走用粗线来画出，结束时得到的温度是 0，它被累加到采样结果中

### 10.5.3 二维易辛模型

二维易辛模型可以被用来模拟简单磁体的行为，也可以用于模拟其他的现象，如图 10.15 所示。问题域是一个正方形的格。每个格点称为一个场点。每个场点  $\sigma_k$  有一个相应的磁矩。磁矩有两种可能：向上和向下。我们将向上赋为  $1/2$ ，向下为  $-1/2$ 。系统的能量由下面的函数给出：

$$E(\sigma) = -\sum_{i,j} J \sigma_i \sigma_j - B \sum_i \sigma_i$$

其中的第一个求和是针对最近的点进行的， $J$  是一个常量，用来表示磁矩之间相互作用的强度， $B$  是另外一个常量，它和外部磁场有关。

我们的目标是估计每个粒子的比热，这可以类比为对系统所有可能的格局进行积分。给定温度  $T$  和波尔兹曼常数  $k$ ，所有可能格局的概率密度函数为：

$$\mu(\sigma) = \frac{e^{-E(\sigma)/kT}}{Z(T)}$$

其中  $Z(T)$  是所有状态的加权和。

不幸的是，我们很难对  $\mu$  的分布进行采样。这里，我们用一个随机采样  $x_i$  来表示一种格局。格局的数量非常巨大，即使对小的格网也是如此。比如我们例子中  $20 \times 20$  的格网有 400 个点，每个点有两种可能状态，所以所有可能格局的数量为  $2^{400}$  种。由于概率密度函数是一个反指数函数，所以大多数状态的概率都极小。所以如果对格局进行均匀采样，将不太可能会经历到那些高概率的格局从而得到一个好的积分估计值。因此，我们必须对倾向于具有较高概率的格局分布进行采样。大都会算法 (Metropolis Algorithm) 就用于产生这样的采样点。

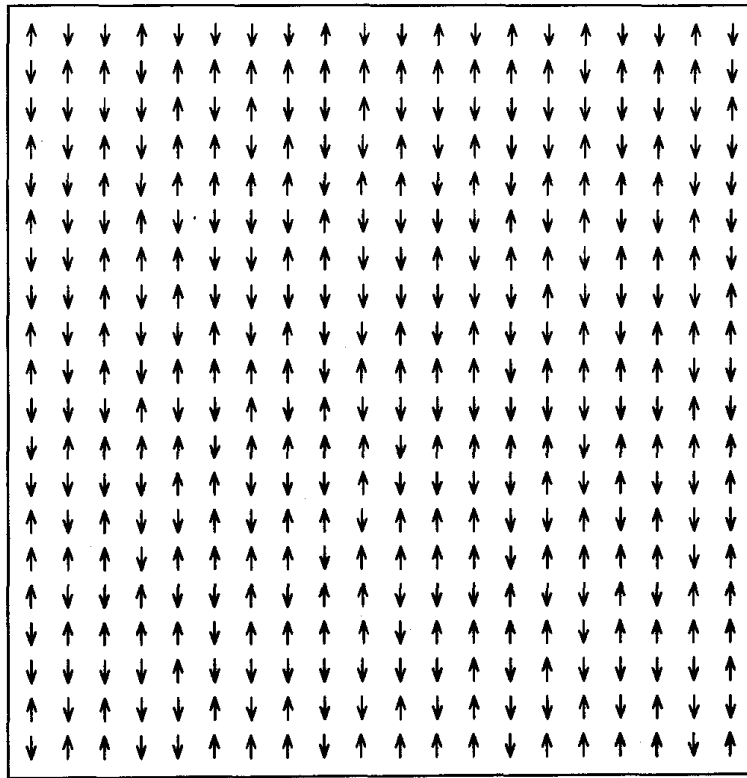


图 10.15 一个 20×20 的易辛模型。400 个点中每一个都有相应的磁矩，或者向上，或者向下。系统的能量是这些旋转的函数。这个模型有  $2^{400}$  种可能的状态。进入每个状态的概率由系统的当前状态和系统温度所决定

大都会算法根据当前的格局  $x_i$  (当前的随机采样) 来产生下一个格局  $x_{i+1}$  (下一个随机采样)。给定  $x_i$ ，算法产生一个相邻的格局  $x'$ ，如果有  $E(x') < E(x_i)$ ，则  $x_{i+1} = x'$ 。如果有  $E(x') > E(x_i)$ ，则  $x_{i+1}$  以概率  $e^{-[E(x') - E(x_i)]/kT}$  等于  $x'$ ，否则  $x_{i+1} = x_i$ 。我们称这个随机采样序列为马尔科夫链，表示在可能的格局空间中的一个随机游走。当用于易辛模型时，大都会算法的形式如图 10.16 所示。

虽然大都会算法产生的序列在短距离内相关性很高，但如果用这个算法产生足够多的采样，它能很好的覆盖整个概率分布函数的范围。

我们怎样保证大都会算法访问的马尔可夫链与概率密度函数相匹配呢？一种方法是满足所谓的详细平衡条件。令  $P(x_i)$  表示位于  $x_i$  格局的概率， $P(x_j|x_i)$  表示从格局  $x_i$  转移到格局  $x_j$  的概率，则详细平衡条件由下面的公式满足：

$$P(x_1)P(x_2|x_1) = P(x_2)P(x_1|x_2)$$

$k$ —Boltzmann's constant (玻尔兹曼常数)

$T$ —Temperature (温度)

$E$ —Energy function (能量函数)

$\Delta$ —change in energy (能量的变化)

$\rho$ —Probability of changing to state  $x'$  (转换到状态  $x'$  的概率)

$u$ —Uniform random variable (正态随机变量)

```

begin
   $x_0 \leftarrow$  Initial state of model
   $i \leftarrow 0$ 
  repeat
     $\sigma \leftarrow$  randomly selected site (from uniform distribution)
     $x' \leftarrow$  Identical to  $x_i$  except spin at  $\sigma$  is reversed
     $\Delta \leftarrow E(x') - E(x_i)$ 
    if  $\Delta < 0$  then
       $\rho \leftarrow 1$ 
    else
       $\rho \leftarrow e^{-\Delta/kT}$ 
    endif
    if  $u < \rho$  then
       $x_{i+1} \leftarrow x'$ 
    else
       $x_{i+1} \leftarrow x_i$ 
    endif
     $i \leftarrow i + 1$ 
  forever
end

```

图 10.16 应用大都会算法于易辛模型

假设  $E(x') > E(x_i)$ 。则下列条件成立的话，大都会算法将满足详细平衡条件：

$$\begin{aligned}
 P(x_i)P(x' | x_i) &= P(x')P(x_i | x') \\
 \Rightarrow \frac{e^{-E(x_i)/kT}}{Z(T)} \times e^{-[E(x') - E(x_i)]/kT} &= \frac{e^{-E(x')/kT}}{Z(T)} \times 1 \\
 \Rightarrow \frac{e^{-E(x')/kT}}{Z(T)} &= \frac{e^{-E(x')/kT}}{Z(T)}
 \end{aligned}$$

这个等式在  $E(x') \leq E(x_i)$  时也同样满足。因此大都会算法满足详细平衡条件。

### 10.5.4 房间分配问题

设大学新生的数目为偶数  $n$ ，我们的目标是将他们分配到  $n/2$  个房间，并且要使学生间的冲突最小。每个学生已经完成了一个调查表，计算机程序根据这些调查表来产生一张冲突表，也就是说，表中  $(i, j)$  的值表示学生  $i$  和  $j$  发生冲突的可能性（注意  $[i, j]$  的值与  $[j, i]$  的值相同）。我们采用一种称为模拟退火的方法来解决这个问题。

物理退火过程是将一种固体加热到融化，然后将它慢慢冷却。物理退火的目的是想得到一种坚固的，具有规整结构的无瑕疵晶体。当物质温度较高的时候，它的原子处于高能量状态，因此可以更容易的重新排列。当温度开始下降时，原子的能量开始降低，重新排列也变得更为困难。慢速的冷却使得物质达到一种最小能量状态，这种状态就是它的结晶状态。

模拟退火是对物理退火过程的一种模仿，它用来解决一类组合优化问题。这个优化问题的一个解对应着物质的一种状态，解所对应的目标函数的值对应着该状态的能量，问题的最优解也就对应着最低能量状态。

模拟退火是一种迭代算法。每次迭代中，当前的解将随机的改变为它邻域中的一个替代解。如果新的解目标函数值比当前解的要小，那么新的解变成当前解。否则，新的解以概率  $e^{-\Delta/T}$  替代当前解，其中  $\Delta$  是两个目标函数值的差，而  $T$  是当前温度。



为什么我们想让一个比现在差的解替代我们已经找到的解？原因是解空间中通常有局部最小值。我们不希望算法太快地停止在这些局部最小值上。当温度很高时，算法可以较容易地从局部最小（如图 10.17 (a) 所示）中跳出来。但当温度降低后，这种可能性开始减小，如图 10.17 (b) 所示。

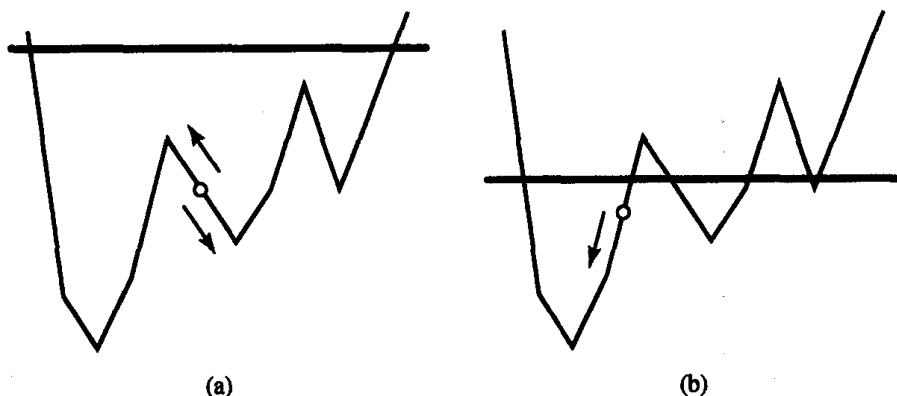


图 10.17 模拟退火总是允许转移到一个具有较低开销的解。而转移到一个开销较高的解的概率随温度下降而降低。(a) 高温时，很可能会转移到一个开销较高的解  
(b) 当温度较低时，转移到一个开销较高的解变得不太可能

请注意模拟退火算法与大都会算法紧密相连。他们都使用相同的概率函数来决定是否要改变到一个高能量状态。不同之处是，模拟退火算法中我们是想找到一个函数的最小值，而不是计算一个积分。

为了用模拟退火解决一个实际问题，我们需要：

- 知道如何表示解；
- 决定开销函数；
- 确定如何从已知解产生一个新的相邻解；
- 设计一个冷却函数。

下面我们在房间分配问题中实施这些步骤。开始的时候我们有一个不相容矩阵  $D$ ，矩阵的元素  $d_{i,j}$  为 0 到 10 之间的浮点数，表示学生  $i$  和  $j$  之间的冲突可能性，注意这个矩阵是对称的，有  $d_{i,j}=d_{j,i}$ 。

问题的一个解是一种将  $n$  个学生分配到  $n/2$  个房间的分配方案。我们用一个数组来记录这些分配方案。数组里的每个元素  $a_i$  是一个  $0 \sim n/2-1$  之间的整数，它表示学生  $i$  所分配到的房间号。数组  $a$  中，0 到  $n/2-1$  之间的每个整数都刚好出现两次。

开销函数就是这种分配方案下的不相容指数和。用  $r_i$  表示学生  $i$  的室友，则开销函数定义为：

$$\sum_{i=0}^{n-1} d_{i,r_i}$$

我们可以通过随机选择两个学生，交换他们的房间来产生下一个新的解。

最后，我们需要选择一个温度函数。温度函数的选择会对算法的性能产生很大的影响。一个较差的函数可能会导致模拟退火算法找到较差的解，也可能使程序耗费很长的时间，或者两种情况会同时出现。

对这个问题，我们选择一个简单的几何温度函数：

$$T_0=1$$

$$T_{i+1}=0.999T_i$$

图 10.18 描述了用模拟退火算法来求解房间分配问题时的收敛情况，温度函数为上面的几何温度函数。两个算法都得到了相同的解，但以  $T_0=10$  开始的算法的迭代次数是以  $T_0=0$  开始的算法的两倍。

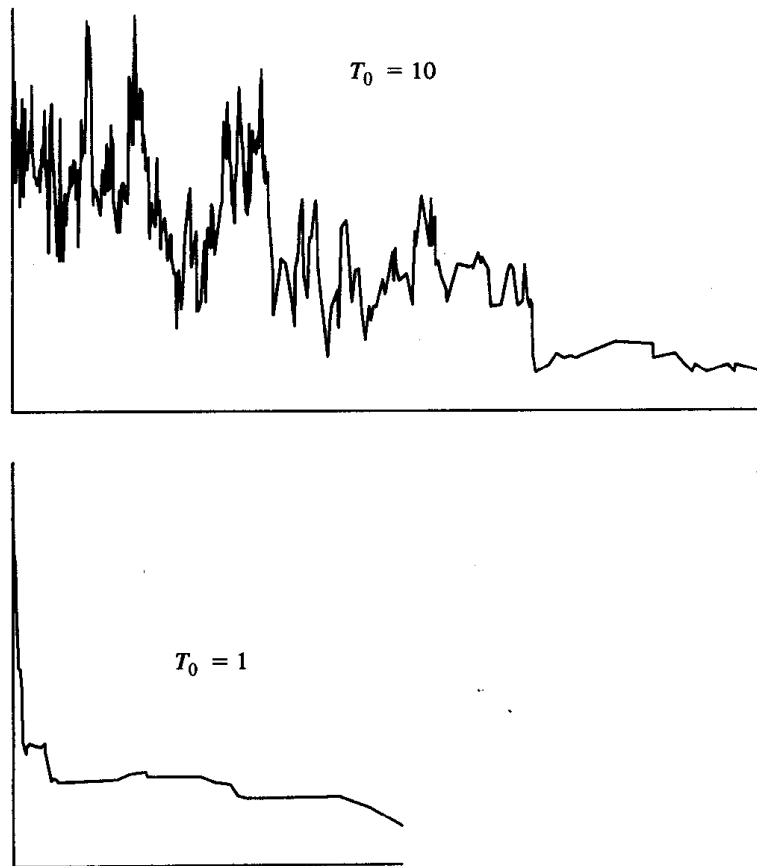


图 10.18 用模拟退火算法求解房间分配问题的收敛。两个算法中几何温度函数都是  $T_{i+1}=0.999T_i$ ，它们都收敛到了最优解。但是，当初始温度较高时，收敛速度会变慢

退火算法的伪代码如图 10.19 所示。

模拟退火并不能保证找到最优解。事实上，对同样的算法，使用不同的随机数序列时，可能收敛到不同的解。因此，可以将相同的算法执行多次，不过每次采用不同的随机数种子。很明显我们可以采用并行计算机来加速总的执行过程。

$a[0 \cdots n-1]$ — $n$ -element array containing room assignments (保存房间分配方案的  $n$  个元素的数组)

$c1, c2$ —two persons involved in possible room swap (可能的房间交换所涉及的 2 个人)

$d[0 \cdots n-1, 0 \cdots n-1]$ — $n \times n$  matrix containing roommate incompatibilities (保存不相容室友信息的  $n \times n$  矩阵)

$sum$ —sum of dislikes of best solution found so far (目前找到的最佳解的冲突数)

$new\_sum$ —sum of dislikes of newly generated solution (新产生解的冲突数)

$t$ —temperature (温度)

```

begin
  Randomly assign students to rooms
  sum ← 0
  for i ← 0 to n - 1 do
    for j ← 0 to n - 1 do
      if a[i] = a[j] then
        sum ← sum + d[i][j]
      endif
    endfor
  endfor
  t ← 1
  i ← 0
  while i < 1000 do {Stop if no changes for 1000 iterations}
    repeat
      c1 ← ⌊u × n⌋
      c2 ← ⌊u × n⌋
      until a[c1] ≠ a[c2]
      Compute new_sum assuming c1 and c2 swap rooms
      if new_sum < sum or u ≤ e(sum-new_sum)/t then
        Swap room assignments for c1 and c2
        sum ← new_sum
        i ← 0
      else i ← i + 1
      endif
      t ← 0.999 × t
    endwhile
    print a and sum
  end
end

```

图 10.19 用模拟退火来解决房间分配问题

### 10.5.5 车库停车问题

一个车库有  $S$  个车位。连续两辆车到达车库门的时间间隔是一个满足泊松分布的随机变量，均值为  $A$  分钟。如果一辆车到达车库时有空闲车位，它开进其中的一个空车位。车在车库中停放的时间是一个满足正态分布的随机变量，均值为  $M$  分钟，标准方差为  $M/4$  分钟。如果一辆车到达时车库已满，这个车就会开走。这个问题中，我们想确定车库的稳态特性：车库中停放的车平均数量，因为车库满而开走的车的数量。

我们把时间表示成一个实数变量  $t$ ，单位为分钟。当模拟开始时， $t=0$ 。

我们用一个有  $S$  个元素的数组  $G$  来模拟车库的车位。其中元素  $G_i$  中的值为车位  $i$  空闲的时刻。模拟开始时，对所有的  $i$ ， $0 \leq i < S$ ，有  $G_i=0$ 。

我们规定模拟开始的时刻是第一辆车到达的时刻，即第一辆车到达的时刻为 0。

由于车到达的时间间隔可以用一个泊松分布来描述，所以车辆到达的时间间隔是一个均值为  $A$  的指数分布，就如我们在前面的章节所讲，可以用表达式  $-A \ln u$  来确定下一辆车的到达时刻，其中  $u$  是一个  $[0, 1]$  上均匀分布的随机变量。

我们用这个时间间隔来对  $t$  递增，并寻找一个可用的车位，也就是，一个满足  $G_i \leq t$  的车位。

当我们将一辆车停入车位  $i$ ，必须设置  $G_i$ ，来反映这辆车离开车库的时刻。因为它是一个正态分布，所以我们可以采用 10.4 节中描述的 Box-Muller 变换。

### 10.5.6 交通环路

一个交通环路（也称为环路）是一种在交叉路口控制交通的方法，它不需要使用信号灯，在欧洲和美国的东北部比较常见。它使得多辆向同方向行驶的车可以同时移动。

图 10.20 给出了一个简单的环路。车辆从 4 个方向进入环路，分别标为 N、W、S 和 E。所有的车以逆时针方向在环路内运动。

在我们的模拟中，我们将交通环路分成 16 个区域。在一个单独的时间步里，环内的所有车辆按逆时针方向运动到下一个区域（或者从四个出口之一退出）。环内的车辆比想进入环的车辆有优先权，因此在一个时间步里，环内的车辆从不会停顿。

当一辆车想进入环内时，必须要满足没有环内的车想进入对应的区域。在图 10.20 中，等待在 N 和 S 的车可以在下一时刻进入环路，因为这样不会引起潜在的冲突。而等待在 W 的车也可以进入环路，因为此时在  $W_c$  的车正要离开环路。然而，等在 E 的车不能进入环，因为  $E_c$  处的车将待在这个区域，而且它有优先权。

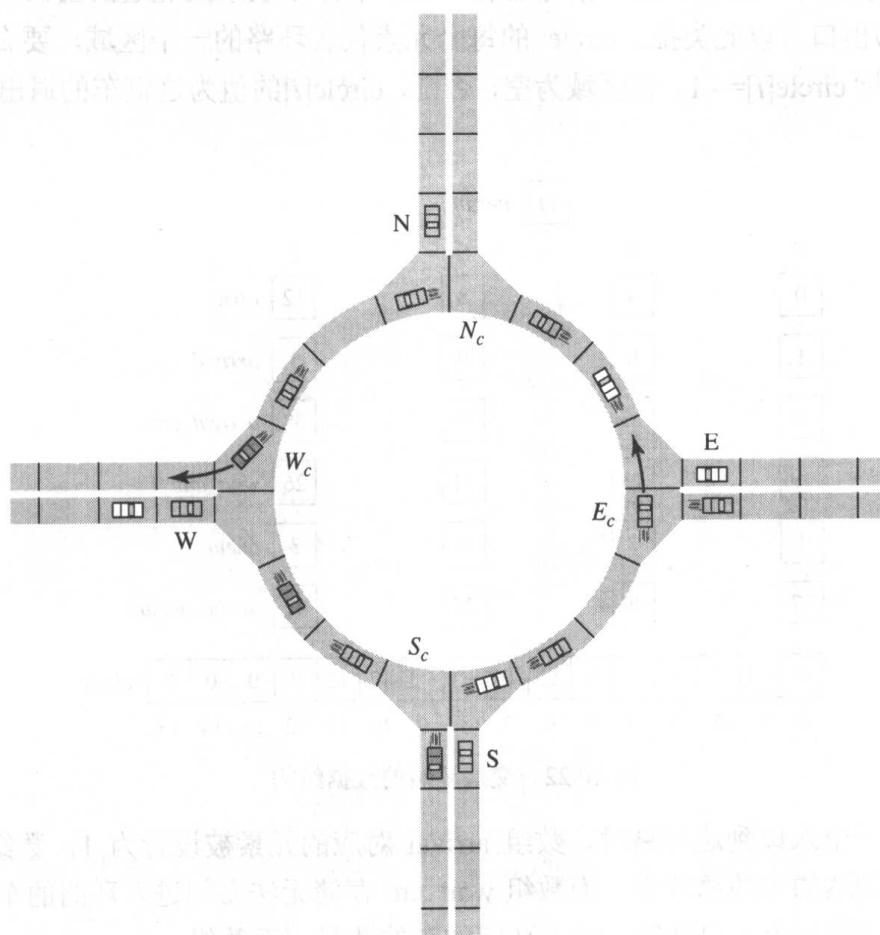


图 10.20 一个环路模型环路内的车按逆时针方向运动，并且比想进入环的车有优先权

为了完善我们的模型，我们需要知道车辆到达四个入口的频率，如图 10.21 所示。一辆车在某个特定时刻到达一个入口的概率满足一个均值为  $m$  的指数分布。数组  $f$  给出了四个入口处车辆到达的平均时间间隔。而矩阵  $D$  的元素  $d_{i,j}$  是一辆从  $i$  进入的车会从  $j$  开出的

概率。比如，一辆车从 E 开入，从 S 开出的概率为 0.20。

		$f$					
			D	N	W	S	E
N	3	N	0.1	0.2	0.5	0.2	
W	3	W	0.2	0.1	0.3	0.4	
S	4	S	0.5	0.1	0.1	0.3	
E	2	E	0.3	0.4	0.2	0.1	

图 10.21 与环路问题相关的概率数组元素  $f_i$  是车辆到达入口  $i$  的平均间隔时间，矩阵元素  $d_{i,j}$  是一辆从  $i$  进入的车会从  $j$  出口离开的概率

我们的目标是构造一个对这个环路的模拟，以回答下面的两个问题：

- (1) 对 4 个入口，一辆车在开入环路前需要等待的概率是多少？
- (2) 对 4 个入口，等待进入环路的车辆的平均队长是多少？

使用 8 个基本的数组就足够进行模拟并存储问题的答案了，如图 10.22 所示。环路本身用 `circle` 来表示，这是一个用 16 个整型元素的数组实现的环形缓冲区。`offset` 数组给出了 `circle` 数组中与每个入口和出口相对应的下标。下标 0 表示最北边的出口/入口，下标 4 是西边的入口/出口，以此类推。`circle` 的每个元素代表环路的一个区域，要么为空，要么有一辆车。如果 `circle[i] = -1`，该区域为空，否则，`circle[i]` 的值为这辆车的退出口口（0、4、8 或者 12）。

71 iteration															
N				W				S				E			
0				4				8				12	offset		
1				0				0				1	arrival		
26				23				22				38	arrival_cnt		
19				13				11				26	wait_cnt		
1				2				0				3	queue		
37				49				20				41	queue_accum		
4	-1	4	-1	8	8	12	0	-1	-1	12	12	8	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

图 10.22 交通环路的数据结构

当车辆从一个入口到达环路时，数组 `arrival` 对应的元素被设置为 1，数组 `arrival_cnt` 包含每个入口到达的车的总数量，而数组 `wait_cnt` 存储无法立刻进入环路的车的数量。数组 `queue` 用来存放每个入口处等待进入环路的车的数量，而数组 `queue_accum` 是一个总数量，即在模拟的所有时刻的 `queue` 的值的和。

交通环路模拟的伪代码如图 10.23 所示。每个时间步的模拟被分成 3 个阶段。首先，新的车辆到达环路。然后，环内的车辆向前运动（未在图 10.22 中给出的数组 `new_circle` 用于这个阶段的计算），那些已经到达它们的退出点的车辆被从环中移出。需要注意的是我们不需要模拟从环退出的车道的情况。最后，如果环内有空区域，相应的车辆就进入环路。

## Traffic Circle Simulation 交通环路模拟

Data structures Representing the Traffic Circle 表示环路的数据结构

*circle*[0...15]—Current state of traffic circle 环路当前状态*new\_circle*[0...15]—Next state of traffic circle 环路下一个状态

Data Structures Representing the Four Entrances 表示四个入口的数据结构

*offset*[0...3]—Each entrance's location (index) in traffic circle 每个入口在环路中的位置(下标)*arrival*[0...3]—1 if a car arrived this time step 如果一辆车到达了时间步, 置为 1*wait\_cnt*[0...3]—Number of cars that have had to wait 等待的车数量*arrival*[0...3]—Total number of cars that have arrived 到达的车的总数*queue*[0...3]—Number of cars waiting to enter circle 等待进入环路的车的数量*queue\_accum*[0...3]—Accumulated queue size over all time steps 所有时间步的累计队列大小

```

begin
  for i ← 0 to 15 do
    circle[i] ← -1
  endfor
  for i ← 0 to 3 do
    arrival_cnt[i], wait_cnt[i], queue[i], queue_accum[i] ← 0
  endfor
  for iteration ← 0 to requested_iterations
    { New cars arrive at entrances }
    for i ← 0 to 3 do
      if  $u \leq 1/f[i]$  then {  $u$  is a uniform random number }
        arrival[i] ← 1
        arrival_cnt[i] ← arrival_cnt[i] + 1
      else arrival[i] ← 0
      endif
    endfor
    { Cars inside circle advance simultaneously }
    for i ← 0 to 15 do
      j ← (i + 1) mod 16
      if circle[i] = -1 or circle[i] = j then new_circle[j] ← -1
      else new_circle[j] ← circle[i]
      endif
    endfor
    circle ← new_circle
    { Cars enter circle }
    for i ← 0 to 3 do
      if circle[offset[i]] = -1 then
        { There is space for car to enter }
        if queue[i] > 0 then
          { Car waiting in queue enters circle }
          queue[i] ← queue[i] - 1
          circle[offset[i]] ← Choose_Exit(i)
        else if arrival[i] > 0
          { Newly arrived car enters circle }
          arrival[i] ← 0
          circle[offset[i]] ← Choose_Exit(i)
        endif
      endif
      if arrival[i] > 0 then
        { Newly arrived car queues up }
        wait_cnt[i] ← wait_cnt[i] + 1
        queue[i] ← queue[i] + 1
      endif
    endfor
    for i ← 0 to 15 do
      queue_accum[i] ← queue_accum[i] + queue[i]
    endfor
  endfor { iteration }
end

```

图 10.23 模拟交通环路的伪代码

当一辆车进入环路后, 模拟必须决定它什么时候退出。可以产生一个均匀分布的随机变量来访问矩阵  $D$ 。在伪代码中, 这个步骤通过对函数 `ChooseExit` 的调用来完成。我们过一个例子来说明这个过程。假设一辆车正在从西边进入环路, 而且我们产生了一个随机数 0.55。我们查找  $D$  的  $W$  行, 直到前面的元素的和超过 0.55。我们的例子中, 第一个元素是 0.2, 比 0.55 小, 这意味着目标不是北边的出口, 第二个元素为 0.1, 这样累加和为 0.3, 也比 0.55 小, 因此西边也不是目的地, 再加上第三个元素 0.3, 这样总和为 0.6, 超过了 0.55, 所以南边是它的出口。因此这辆车从西边入口进入 (偏移量为 4), 从南边的出口离开 (偏移量 8)。因此我们进行赋值操作: `circle[4] ← 8`。

当环路模拟开始时, 环路内没有车辆, 延迟时间也最小。当模拟继续进行, 交通堵塞开始出现并逐渐消失。模拟继续进行, 直到问题的答案收敛为止。

## 10.6 本章小结

蒙特卡洛法通过统计采样的方法来为一大类问题寻找适当的解, 其中两个重要的应用是数值积分和模拟。当积分的维数大于 6 时, 蒙特卡洛法比确定性数值积分算法要更好。在实际中, 很多具有随机行为特性系统中出现的问题很难得到解析解。对于这些问题, 蒙特卡洛法可以提供一个好的工具来产生近似解。

为了产生可靠的答案, 蒙特卡洛法需要有一个好的随机序列。一个采用 32 位整数随机数产生器的最大周期是  $2^{32}$ , 大约是 40 亿。对现代的计算机来说, 这个周期太小了。你必须保证的随机数生成器至少有 48 位的精度。

有时一个通常情况下表现良好的随机数产生器可能并不适合某些特定的应用, 如果你有一个关键应用, 你最好用两种不同的随机数生成器来将它运行两次, 分析它们的结果是否相同。

为了在并行计算机上产生随机数, 人们已经提出了很多的方法, 包括蛙跳方法, 序列分割, 以及进程各自维护其独立序列等。

最常用的随机数产生器产生一个满足均匀分布的伪随机数序列。蒙特卡洛法经常需要产生其他分布的随机数。对指数分布和正态分布, 有很直接的方法从均匀分布得到所需的采样。而拒绝法使我们可以产生其他分布的随机数。

为了描述解决问题的实际方法, 我们考察了蒙特卡洛法的六个应用。在对这些问题求解的过程中, 我们介绍了两个重要的算法。大都市算法对产生高维空间中的样本是一个特别好的方法, 而模拟退火方法可以为组合优化问题找到近似解。

## 10.7 主要术语

detailed balance condition

详细平衡条件

multiplicative congruential generator

乘法同余产生器

seed	种子
linear congruential generator	线性同余产生器
simulated annealing	模拟退火
Markov chain	马尔可夫链
period	周期
site	点
Metropolis algorithm	大都会算法
pseudo-random number generator	伪随机数产生器
uniform distribution	均匀分布
Monte Carlo method	蒙特卡洛法
Monte Carlo time	蒙特卡洛时间
random walk	随机游走

## 10.8 参 考 文 献

对蒙特卡洛法和大都会算法的一个简明易懂的介绍可以参见由 R.Landau 和 Paez 写的 “*Computational Physics: Problem Solving with Computers*”【65】。与之相对, D.Landau 和 Binder 的 “*A Guide to Monte Carlo Simulations in Statistical Physics*” 则对蒙特卡洛模拟的设计实现和结果分析有更严格的描述【64】。在 Kalos 和 Whitlock 的 “*Monte Carlo Methods*”【58】中, 我第一次看到了“饼盘中的雨滴”的比喻。

Lehmer 在 1951【69】发表了线性同余方法。有段时间, 就称它为“Lehmer 算法”。寻找更长周期的线性同余方法的工作不断继续。Wu 给出了一个采用大质数模  $2^{61}-1$  的乘法同余产生器, 以及四种乘数形式【118】。然而, L'Ecuyer 和 Simard 指出, 这个产生器产生的随机数的二进制形式中的 1 的个数并不独立【68】。

为了产生非均匀分布的随机数, 人们已经提出了许多算法。Wallace 提出了不依赖均匀分布的产生正态和指数分布的一种快速方法【110】, Leva 给出了一种产生正态分布随机变量的快速方法, 这种方法对每个随机数平均只需要 0.012 个对数操作【72】。Marsaglia 和 Tsang 给出了一种生成正态、指数和其他随机变量的快速方法【80】。

Mascagni 对采用参数化方法, 而不是序列分割方法生成并行随机数序列的方法做了一个概述【84】, 他的论文包含了有用的早期工作的文献。

SPRNG 库, 可扩展的并行随机数产生器, 由 Mascagni 和 Srinivasan 在 *ACM Transaction on Mathematical Software* 上做了一个简单介绍, 可以从佛罗里达州立大学免费得到, 网址为 <http://sprng.cs.fsu.edu>。

交通环路问题是从 Manno 的 “*Introduction to the Monte Carlo Method*” 中的例子演变而来。



## 10.9 练 习 题

10.1 假设你采用蒙特卡洛法来计算一个积分。方法和 10.1 节中的计算  $\pi$  的方法相同，只是要积分的函数有 20 维，而不是 2 维。如果你在使用一个线性同余随机数产生器，那么你需要留意什么问题？

10.2 本书中没有讨论的一种并行随机数产生器采用了如下的方法：每个进程都采用相同的线性同余产生器（具有相同的乘法系数，偏移量和模）。只是每个进程的种子  $X_0$  不同。这种方法可能的主要问题是什么？

10.3 用 Box-Muller 变换写一个 C 程序，来返回满足正态分布的双精度浮点数。

10.4 一个直径为  $d$  的圆柱形洞是从一个边长为  $s$  的正方体中钻出来的，如图 10.24 所示，圆柱的中线与正方体的一条对角线重合。写一个程序来求出当  $s=2$ ,  $d=0.3$  时，正方体残余的体积（要求 5 位精度）。提示：点  $(x_1, y_1, z_1)$  到直线  $x=y=z$  的距离为：

$$\sin \left[ \cos^{-1} \left( \frac{x_1 + y_1 + z_1}{\sqrt{3} \sqrt{x_1^2 + y_1^2 + z_1^2}} \right) \right] \sqrt{x_1^2 + y_1^2 + z_1^2}$$

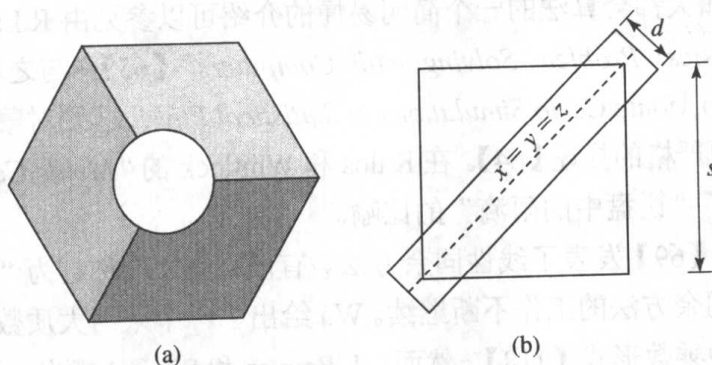


图 10.24 从两种角度观察题 10.4 中所描述的物体。(a) 从正方体的一个角往下看，我们可以看到这个圆柱体直穿到正方体对面的角 (b) 从正方体的侧面看，我们可以看到与直线  $x=y=z$  距离在  $d/2$  以内的物质都被去掉了

10.5 写一个程序来求下面的积分，保留 5 位的精度。

$$\int_{x=0}^4 \int_{y=0}^3 \int_{z=0}^2 4x^2 + xy^2 + 5y + yz + 6zdzdydx$$

10.6 写一个程序来求下面的积分，保留 5 位的精度。

$$\int_{x=0}^4 \int_{y=0}^3 \int_{z=0}^2 4x^2 + xy^2 + 5y + yz + 6zdzdydx$$

10.7 一个放射性原子的平均生存时间为  $m$  个时间单位。在任意时间单位里这个放射性原子衰变的概率为  $(1/m) e^{-1/m}$ 。给出在一个包含 100 000 个放射性原子的组内，每个时间步里发生衰变的原子数目。 $m=250$ ，计算开始的 1 000 个时间步。

10.8 实现一个解决 10.5.1 节中的中子输运问题的并程序。其中  $C_c=0.3$ ,  $C_s=0.7$ 。

对  $H=1, 2, 3, \dots, 10$ , 分别确定吸收, 反射和穿透的概率。对每个  $H$ , 你的结果必须基于至少 10 000 000 个中子。

10.9 实现一个解决 10.5.2 节中的稳态温度问题的并程序。假设正方形板被划分为  $20 \times 20$  的小块, 假设板的三个边温度为 0 度, 第四边的温度为 100 度。计算板中央的温度, 保留三位有效数字。

10.10 编写实现 10.5.3 节中易辛模型的并程序。目标是得到一个  $100 \times 100$  的系统在 1 000 000 次迭代后的能量水平。设  $J=1$ ,  $B=0$ ,  $kT=1$ 。系统的初始状态  $x_0$  中, 使每个点  $\sigma_k$  都向上。对每一对水平或者垂直相邻的点计算  $\sum_{i,j} J \sigma_i \sigma_j$ 。要求试验重复 1 000 次。

10.11 实现一个解决 10.5.4 节中的房间分配问题的并程序。假设  $n=20$ ,  $T=1$ 。采用一个随机数产生器来构造  $D$  矩阵。每个元素必须是 0~10 之间满足均匀分布的一个随机变量。每个进程需要对相同的  $D$  来求解问题, 但要求随机数生成器的种子不同。

10.12 实现一个解决 10.5.5 节中的车库问题的并程序。假设  $S=80$ ,  $A=3$ ,  $M=240$ 。确定车库中车的平均数量, 以及稳态下 ( $t \rightarrow \infty$ ) 由于车库满而一辆车必须开走的概率。

10.13 实现一个解决 10.5.6 节中的交通环路问题的并程序。用这个程序回答下面的两个问题:

- (a) 对每个入口, 稳态下一辆车在进入环路前必须等待的概率;
- (b) 对每个入口, 稳态下等待进入环路的车辆队列的平均长度。

# 第 11 章 矩 阵 乘 法

We go on multiplying our conveniences only to multiply our cares. We increase our possessions only to the enlargement of our anxieties.

Anna C. Brackett, *The Technique of Rest*

## 11.1 概 述

想想在计算机科学课程中我们曾多么频繁地讨论到矩阵乘法！然而，可笑的是在科学和工程问题中几乎不会遇到大矩阵的相乘。在下面两个领域中则会用到矩阵乘法运算。首先，计算化学家通常用状态来描述化学系统中所遇到的问题。如果每个指标对应一个基本状态，那么由指标构成的矩阵就会逼近系统的哈密尔顿函数。基本状态间的转换可以通过矩阵相乘来实现。另外一个例子是在信号处理中的某些变换的实现依赖于大矩阵的乘法。

本章介绍两种串行矩阵相乘算法并探索两种不同的并行化方法。在第 11.2 节，我们将回顾标准的串行算法。通过绘制性能随矩阵尺寸变化的图表，我们可以看到当第二个源矩阵不能放进 Cache 时性能将显著下降。之后我们介绍一种矩阵乘法的递归实现算法。这种算法将原始矩阵的分块并相乘，可以保持很高的 Cache 命中率。

在第 11.3 节中，我们将设计一个按行分解矩阵的并行算法，导出这种算法执行时间的期望值，并分析其等效率特性。在第 11.4 节，我们使用同样的设计分析方法研究基于棋盘式分解的并行算法。

## 11.2 矩阵相乘的串行算法

### 11.2.1 基于行的迭代算法

一个大小为  $l \times m$  的矩阵  $A$  与大小为  $m \times n$  的矩阵  $B$  之积是一个  $l \times n$  的矩阵  $C$ ，它的元素有如下的定义：

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} b_{k,j}$$

图 11.1 所示为矩阵相乘的串行算法。这个算法需要进行  $lmn$  次加法和同样次数的乘法。所以两个  $n \times n$  矩阵用串行算法相乘的时间复杂度为  $\Theta(n^3)$ 。人们也开发了其他一些时间复杂度较低的算法，比如 Strassen 算法。本章的所有算法都是对直接相乘算法的并

行化。

这个算法非常容易实现。我们在 Beowulf 集群的一个节点上测试了这个算法的 C 语言版本，每个节点是拥有 933MHz PIII，256K 二级 Cache 的 Linux 机器。图 11.2 是测试的结果。对较小的矩阵，执行速度为 220Mflops，但对较大的矩阵只有 80 Mflops。是什么原因导致性能的下降的呢？

基于行的矩阵相乘

Input:

$a[0..l-1, 0..m-1]$

$b[0..m-1, 0..n-1]$

Output:

$c[0..l-1, 0..n-1]$

for  $i \leftarrow 0$  to  $l-1$

for  $j \leftarrow 0$  to  $n-1$

$c[i, j] \leftarrow 0$

for  $k \leftarrow 0$  to  $m-1$

$c[i, j] \leftarrow c[i, j] + a[i, k] \times b[k, j]$

endfor

endfor

endfor

图 11.1 基于行的迭代式矩阵乘法

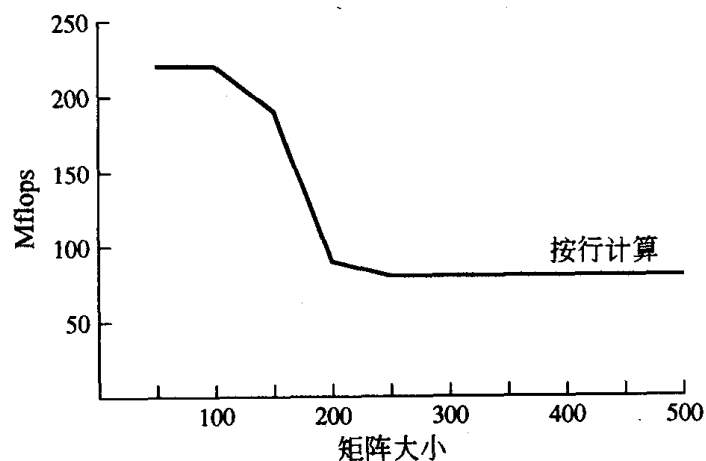


图 11.2 基于行的迭代式矩阵乘法的性能。测试平台：933MHz PentiumIII 处理器。

一旦 Cache 中放不下矩阵  $B$  时，此算法的性能急剧下降

参看图 11.3。每次外层迭代都要读取  $B$  的所有元素。如果对于 Cache 而言， $B$  太大的话，后面读入元素就会覆盖先前已经读入 Cache 中的元素，也就意味着进行下一次的外层迭代时，需要将  $B$  全部重新读入。因此，当矩阵的规模达到某个阈值时，Cache 的命中率会显著下降，从而降低了 CPU 的性能。

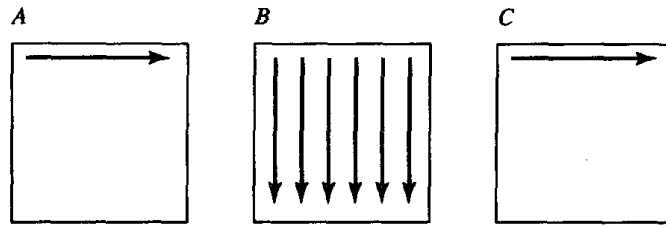


图 11.3 在  $i$  为索引的循环的一次迭代中, 需要读取  $A$  的第  $i$  行和整个  $B$  矩阵, 并得到  $C$  的第  $i$  行

我们测试的 CPU 中有 256KB 的 Cache。进行双精度浮点数相乘时, 每个矩阵元素占 8 个字节。所以 Cache 中最多可以同时放下 32 768 个矩阵元素。32 768 的平方根约为 181。这个算法的性能测试结果也反映出, 当  $n \leq 150$  时, Cache 的命中率远高于当  $n \geq 200$  时的命中率。

## 11.2.2 基于块的递归算法

当矩阵  $A$  的列数等于  $B$  的行数时, 二者才可以相乘。

假设  $A$  有  $l$  行  $m$  列,  $B$  有  $m$  行  $n$  列。我们把  $A$  分成 4 个小矩阵:

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

同时也把  $B$  分成 4 个小矩阵:

$$B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

同时使得  $A_{00}$  和  $A_{10}$  的行数分别等于  $B_{00}$  和  $B_{01}$  的列数, 那么矩阵的积为:

$$C = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix}$$

其中  $A_{ik}B_{kj}$  表示分块块矩阵相乘,  $+$  表示矩阵加法。

我们的目标是计算  $C=AB$ 。如果  $B$  太大不能放进 Cache 中时, 可以先将其分块再计算  $C$ 。如果分块矩阵  $B_{ij}$  仍与 Cache 不匹配, 我们可以递归地对矩阵进行分解, 直到分块能放进 Cache 为止。图 11.4 是递归算法的 C 语言实现。

```
double a[N][N], b[N][N], c[N][N];
void mm (int crow, int ccol, /* Corner of C block */
        int arow, int acol, /* Corner of A block */
        int brow, int bcol, /* Corner of B block */
        int l, /* Block A is l x m */
        int m, /* Block B is m x n */
        int n) /* Block C is l x n */
{
    int lhalf[3], mhalf[3], nhalf[3]; /* Quadrant sizes */
```

```

int i, j, k; double *aptr, *bptr, *cptr;
if (m * n > THRESHOLD) {
    /* B doesn't fit in cache---multiply blocks of A, B */
    lhalf[0] = 0; lhalf[1] = 1/2; lhalf[2] = 1 - 1/2;
    mhalf[0] = 0; mhalf[1] = m/2; mhalf[2] = m - m/2;
    nhalf[0] = 0; nhalf[1] = n/2; nhalf[2] = n - n/2;
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            for (k = 0; k < 2; k++)
                mm (crow+lhalf[i], ccol+mhalf[j],
                    arow+lhalf[i], acol+mhalf[k],
                    brow+mhalf[k], bcol+nhalf[j],
                    lhalf[i+1], mhalf[k+1], nhalf[j+1]);
} else {
    /* B fits in cache --- do standard multiply */
    for (i = 0; i < 1; i++)
        for (j = 0; j < n; j++) {
            cptr = &c[crow+i][ccol+j];
            aptr = &a[arow+i][acol];
            bptr = &b[brow][bcol+j];
            for (k = 0; k < m; k++) {
                *cptr += *(aptr++) * *bptr; bptr += N;
            }
        }
}
}

```

图 11.4 基于块的递归矩阵乘法的 C 语言实现。这个函数第一次调用的形式为 `mm(0, 0, 0, 0, 0, 0, N, N, N)`

图 11.5 描述了递归算法是如何工作的。在这个例子中，矩阵 *B* 对于 Cache 来说太大了，所以被分成 4 部分。但是每个分块仍然太大，所以程序进行了第二次递归。

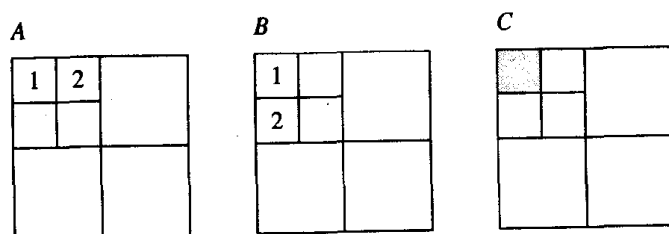


图 11.5 递归式矩阵乘法将矩阵分解为更小的块，直至这些块能放进 Cache 为止。

本例中通过两次递归才得到了适合于 Cache 的分块。

矩阵 *C* 的每块都是两个分块矩阵乘积的和

在与前面完全相同的环境下,我们测试了递归式矩阵乘法用 C 语言实现的程序。图 11.6 所示为两次测试的实验结果。即使矩阵的规模远远超过了 Cache 的容量,递归算法一直保持着很高的性能。

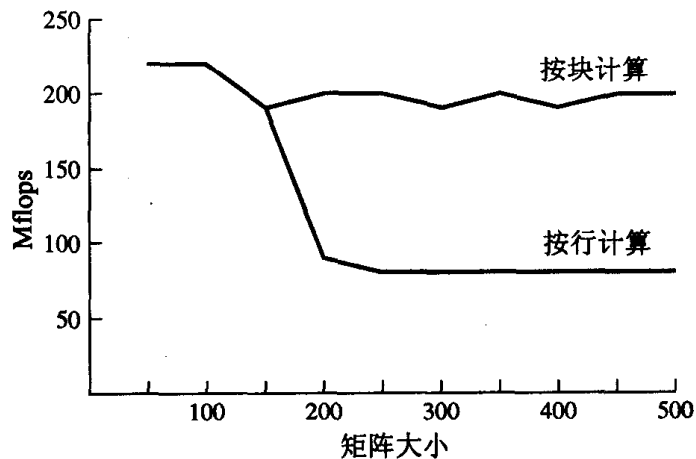


图 11.6 在一台 PIII 933MHz 的计算机上,两个串行矩阵乘法算法的性能。  
基于块分解的算法一直保持着很高的 Cache 命中率,比基于行算法的性能要好得多

## 11.3 行块分解并行算法

本节我们开发一个按行分块的并行矩阵相乘算法。

### 11.3.1 确定原始任务

相乘得到的矩阵  $C$  中,每个元素都是矩阵  $A$  与  $B$  中元素的函数。只要  $A$  和  $B$  在计算的时候不改变, $C$  的各个元素就可以并行地计算。作为并行设计的第一步,我们将  $C$  的每个元素与一个原始任务相关联。

完成这些任务到底需要那些元素?为了计算  $c_{i,j}$ ,需要计算  $A$  的第  $i$  行和  $B$  的第  $j$  列的内积(即点积)。

### 11.3.2 聚合

我们可以根据数据的相关性来组合任务。非常自然地,我们可以将  $C$  按行或按列进行聚合,因为它们共享  $A$  的某一行或  $B$  的某一列。基于这种设计的算法很简单。我们选择按行来对任务进行聚合。

如果我们对所有的矩阵都采用相同的组合方法,情况就会更简单。这样的话,一次矩阵相乘的结果可以直接作为另一次矩阵相乘的源矩阵。我们假设,每个任务负责  $A$ 、 $B$ 、 $C$  的相应行。

下面考察任务  $i$  利用  $A$  的第  $i$  行, $B$  的第  $i$  行和  $C$  的第  $i$  行能完成什么工作。如前所述:

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} b_{k,j}$$

利用  $A$ 、 $B$  的第  $i$  行可以计算  $c_{i,0}$  的一个因子  $a_{i,0}b_{0,0}$ ，也可以计算  $c_{i,1}$  的一个因子  $a_{i,0}b_{1,0}$ ，依此类推。也就是说，这个任务通过  $n$  个乘法可以得到  $C$  矩阵第  $i$  行元素的一个因子。

从前一节我们可以知道， $C$  的第  $i$  行是  $A$  的第  $i$  行和矩阵  $B$  的积。所以每个任务必须访问  $B$  的所有行才能得到结果。

### 11.3.3 通信和进一步的聚合

如果我们将任务组织为环状结构，每个进程将自己所拥有的  $B$  的一行传递给环上的下一个任务，经过  $m$  次迭代，每个任务将都拥有完整的  $B$ 。

图 11.7 描述了这个过程。这里我们假设  $C$  有 4 行。

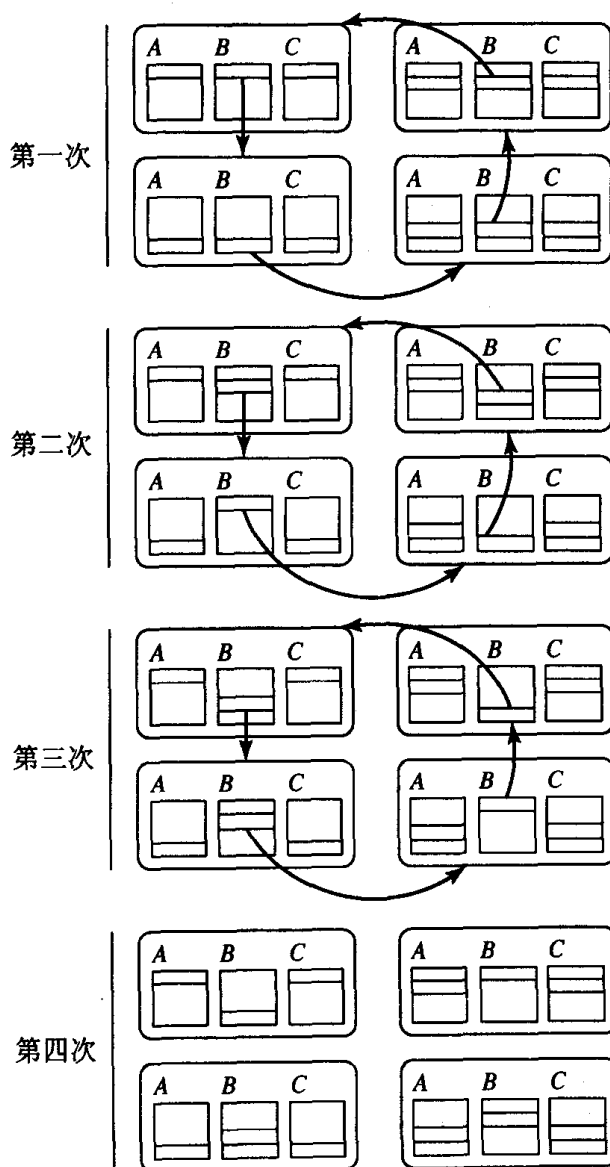


图 11.7 面向行的并行矩阵乘法算法中  $B$  的通信。每个任务负责  $A$ 、 $B$  和  $C$  的一行。

如果  $B$  有  $m$  行，经过  $m-1$  次通信每个任务都访问过了完整的  $B$



我们的并行算法中实际执行计算的进程数可能远小于目标矩阵的行数,所以我们可以考虑进一步进行聚合。使用我们刚刚发明的环状通信模式,我们可以使用按行分块的聚合模式。

### 11.3.4 分析

为了简化分析,假设  $A$ 、 $B$ 、 $C$  都是  $n \times n$  方阵,  $p$  为活动的进程数,而且  $n$  是  $p$  的整数倍。在整个算法中,每个进程控制  $A$  和  $C$  的  $n/p$  行。矩阵  $B$  被按行分割为大小为  $n/p$  的块,各个分块将如图 11.7 所示那样在进程之间传递。

算法一开始,每个进程将它负责的那一部分  $C$  矩阵(大小为  $(n/p) \times n$ )初始化为 0。在每次迭代中,每个进程将它控制的  $A$  部分的分块(大小为  $(n/p) \times (n/p)$ )和  $B$  的分块(大小为  $(n/p) \times n$ )相乘。同时,将得到的大小为  $(n/p) \times n$  结果与它对应的  $C$  部分相加。令  $\chi$  为内积计算中加-乘所需要的时间,那么每次迭代耗时为:

$$\chi(n/p)(n/p)n = \chi n^3 / p^2$$

每次迭代中每个进程必须将自己部分的  $B$  传给给环上的下一个进程。如果在计算之后才进行通信,那么发送这些消息会为每次迭代增加长度为  $\lambda + (n/p)n/\beta$  的延时。从环上的上一个进程接收消息则会在同时发生。

此算法需迭代  $p$  次。总共的计算和通信时间为:

$$p[\chi n^3 / p^2 + \lambda + n^2 / (p\beta)] = \chi n^3 / p + p\lambda + n^2 / \beta$$

我们仔细检查一下这个表达式。串行算法的执行时间为  $\chi n^3$ 。由于计算被平均的分配到  $p$  个进程,所以并行算法的纯计算时间为  $\chi n^3 / p$ 。

每个进程发送  $p$  个消息,所以  $p\lambda$  这一项是合理的。最后,每个进程将  $B$  的一部分发送给(每次一部分)后继进程,所以  $n^2/\beta$  是必须的。

下面我们来确定按行分块并行矩阵乘法的等效率特性。串行算法的时间复杂度为  $\Theta(n^3)$ 。并行算法的通信复杂度为  $\Theta(n^2)$ 。将原有的复杂度乘以处理器的数目  $p$  得到开销:  $T_o(n, p) = \Theta(pn^2)$ 。所以行块分解的矩阵乘法的等效性关系为:

$$n^3 \geq Cpn^2 \Rightarrow n \geq Cp$$

内存使用函数  $M(n) = n^2$ 。为了保持不变的效率,每个进程的内存使用量必须增加:

$$M(Cp) / p = C^2 p^2 / p = C^2 p$$

可见,这个算法的可扩展性不好。

最后,在这个算法中,将通信和计算重叠起来是很有希望的改进方式。如果在计算的时候有足够的内存接收新的一部分  $B$ ,那么每个进程可以在乘法之前对发送和接收进行初始化。因为通信复杂度是  $\Theta(n^2)$ ,计算复杂度是  $\Theta(n^3)$ 。所以当矩阵的规模较大的时候,通信时间可以为计算时间完全掩盖(通信初始化时间不包括在内),这样会显著提高加速比。

我们能做得更好一些么?

考虑上面的基于行并行算法中的计算/通信比。在  $p$  个进程上进行两个  $n \times n$  矩阵的乘法,而且  $n$  是  $p$  的倍数。每个进程迭代  $p$  次,每次迭代都完成  $A$  的一个大小为  $(n/p) \times (n/p)$  的子矩阵与  $B$  的大小为  $(n/p) \times (n/p)$  的子矩阵的乘法。因为矩阵乘法中夹杂着通信( $B$  的元素在进程间传递),所以  $B$  中的每个元素所对应的计算量为:

$$\frac{2n^3/p^2}{n^2/p} = \frac{2n}{p}$$

这个比率很小，这是由于  $B$  的子矩阵其列数是行数的  $p$  倍。

下一节我们要开发一个算法以提高计算通信比率。

## 11.4 Cannon 算法

本节我们要开发一个基于棋盘式分解的并行算法。这个算法通常称作 Cannon 算法【14】。

### 11.4.1 组合

因为基于行的并行算法中  $B$  的分块又短又粗-列数是行数的  $p$  倍，所以  $B$  中每个元素平均分得的计算量很低。

计算  $c_{i,j}$  的任务需要访问  $A$  的第  $i$  行和  $B$  的第  $i$  列。在基于行的聚合方式中，每个进程负责计算  $C$  的一行中的所有元素，也就是说需要访问整个矩阵  $B$ ，如图 11.8(a) 所示。

相反，如果将进程进行聚合，使其只负责  $C$  中的一个方块（或近似方块），每个进程需要访问的  $B$  的元素数目就能显著地减少。

下面计算一下这种方法到底有多大改善。为了简化计算，我们假设  $A$ 、 $B$ 、 $C$  都是  $n \times n$  方阵，且  $p$  是平方数， $n$  整除于  $\sqrt{p}$ 。每个进程负责计算  $C$  中大小为  $(n/\sqrt{p}) \times (n/\sqrt{p})$  的一块。为了计算这些元素，进程需要访问  $A$  中的  $n/\sqrt{p}$  行和  $B$  中的  $n/\sqrt{p}$  列，如图 11.8(b) 所示。

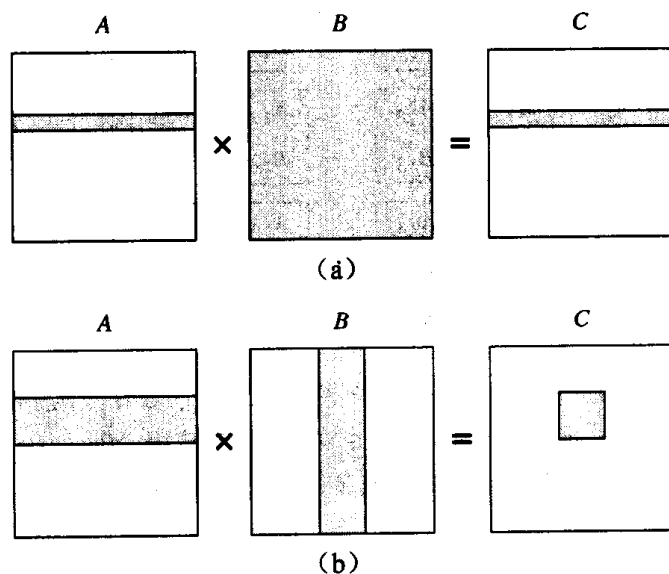


图 11.8 两种并行算法下用来计算单个进程所负责的  $C$  矩阵部分  $A$  和  $B$  的元素个数的比较。

- (a) 在基于行的算法中，每个进程负责计算  $C$  中的  $n/p$  行，这需要访问  $A$  中的  $n/p$  行和  $B$  中的每个元素； (b) 在 Cannon 算法中，每个进程计算大小为  $(n/\sqrt{p}) \times (n/\sqrt{p})$  的  $C$  的分块，这需要访问  $A$  中的  $n/\sqrt{p}$  行和  $B$  中的  $n/\sqrt{p}$  列

每个进程需要完成的计算仍然是  $2n^3/p$ 。每个进程需要访问的元素数是  $2n(n/\sqrt{p})$ 。计算量与通信量之比为：

$$\frac{2n^3/p}{2n^2/\sqrt{p}} = \frac{n}{\sqrt{p}}$$

下面计算在什么条件下 Cannon 算法的计算量/通信量比值高于基于行的算法：

$$\frac{n}{\sqrt{p}} > \frac{2n}{p} \Rightarrow \sqrt{p} > 2 \Rightarrow p > 4$$

我们能够期待，当进程数大于 4 的时候，Cannon 算法的效果更好。

## 11.4.2 通信

既然已经明确了棋盘式分解法的潜力很大，下面就让我们来挖掘这些潜力。首先，我们看一下  $A$  和  $B$  是怎样分布在各个进程中的，如图 11.9 (a) 所示。进程  $P_{i,j}$  包含有块  $A_{i,j}$  和  $B_{i,j}$ ，并负责计算  $C_{i,j}$  这一块。除了主对角线上的进程外，其他进程所含有的  $A$  矩阵块和  $B$  矩阵块并不能直接相乘。

我们需要循环移动各个分块以便  $P_{i,j}$  包含一对可以直接计算  $C_{i,j}$  的分块。图 11.9 (b) 所示为一种方法。进程网格中第  $i$  行的进程将  $A$  的分块循环左移  $i$  个位置，第  $j$  列的进程将  $B$  的分块循环上移  $j$  个位置。现在结果就符合条件了：每个进程所包含的  $A$  和  $B$  的分块可以直接相乘以得到其所对应的  $C$  分块的部分结果。

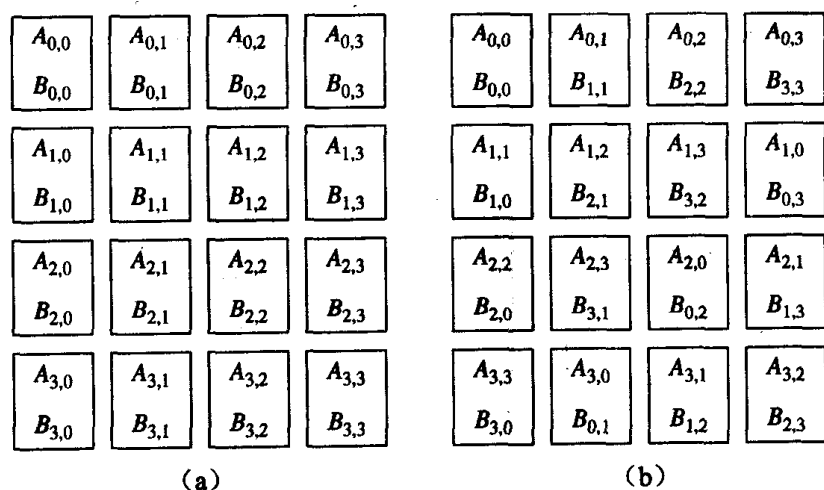


图 11.9 矩阵乘法中块的位置。(a) 初始化块分布。进程  $P_{i,j}$  包含块  $A_{i,j}$  和  $B_{i,j}$ 。块矩阵乘法计算所有匹配的  $A_{i,k}B_{k,j}$ 。可以发现只有在主对角线上的进程 ( $P_{0,0}$ 、 $P_{1,1}$ 、 $P_{2,2}$  和  $P_{3,3}$ ) 才是匹配的。(b) 并行算法将第  $i$  行的  $A$  循环左移  $i$  个位置，将第  $j$  列的  $B$  循环上移  $j$  个位置。现在每个进程  $P_{i,j}$  都有一对可以直接相乘的块

进程网格的大小为  $\sqrt{p} \times \sqrt{p}$ 。初始化过程中重新排列  $A$  和  $B$  的块。之后，棋盘式并行矩阵乘法的计算需要  $\sqrt{p}$  步。每个进程计算自己的两个  $A$  和  $B$  分块的乘积并将结果加到  $C$  的部分和上，然后把自己的  $A$  矩阵块传递给左边的进程并从右边的进程接收  $A$  矩阵块；并把自己的  $B$  矩阵块传递给上面的进程并从下面的进程接收  $B$  矩阵块。图 11.10 以大小

为  $4 \times 4$  的进程网格中的  $P_{1,2}$  进程为例, 解释了这种块循环的操作。

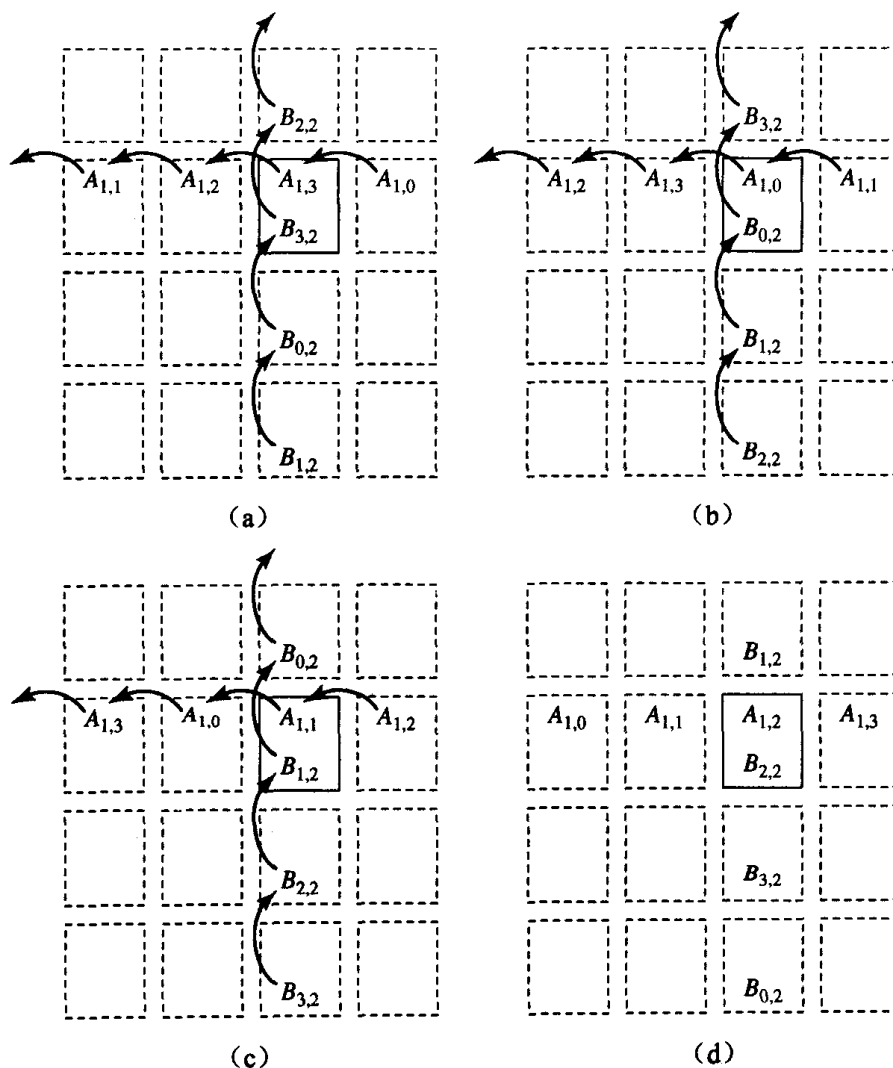


图 11.10 从进程  $P_{1,2}$  的角度说明 Cannon 算法。进程组织成一个 2 维网格, 每个进程已经获得进行第一次迭代计算所需要的块。(a) 第一次分块相乘。每次完成相乘操作后,  $P_{1,2}$  把自己的  $A$  矩阵块传递给左边的进程并从右边的进程接收新的  $A$  矩阵块; 同时把自己的  $B$  矩阵块传递给上面的进程并从下面的进程接收新的  $B$  矩阵块。(b) 第二次分块相乘。(c) 第三次分块相乘。(d) 第四次分块相乘

### 11.4.3 分析

本小节我们要导出 Cannon 算法执行时间的表达式。为了简化分析, 假设  $A$ ,  $B$  和  $C$  均为  $n \times n$  方阵,  $p$  (活动进程数) 是平方数且  $n$  是  $\sqrt{p}$  的整数倍。每个进程负责计算大小为  $(n/\sqrt{p}) \times (n/\sqrt{p})$  的  $C$  矩阵的一块。

首先考虑计算时间。算法开始后, 进程将自己部分的  $C$  块初始化为 0。每次迭代都将大小为  $(n/\sqrt{p}) \times (n/\sqrt{p})$  的  $A$  分块和大小为  $(n/\sqrt{p}) \times (n/\sqrt{p})$  的  $B$  分块相乘, 把得到的结果与  $C$  的部分结果相加。我们令  $\chi$  为内积计算中一次加一乘操作所需要的时间, 那

么每次迭代中计算所消耗的时间为:

$$\chi(n/\sqrt{p})^3 = \chi n^3 / p^{3/2}$$

共有  $\sqrt{p}$  次迭代, 所以全部的计算时间为:

$$\sqrt{p} \chi n^3 / p^{3/2} = \chi n^3 / p$$

再来考虑通信开销。在第一次迭代前, 每个进程必须将自己的  $A$  分块和  $B$  分块发送至相应的进程, 并接收自己需要的  $A$  分块和  $B$  分块。假设消息的发送和接收可以同时执行, 但一次只允许发送或接收一条消息。同时我们令  $1/\beta$  为传递一个矩阵元素所消耗的时间。那么初始化块分布的时间为:

$$2\left(\lambda + \frac{n^2}{p\beta}\right)$$

每次迭代中 (共有  $\sqrt{p}$  次迭代) 进程都必须发送和接收  $A$  和  $B$  的分块。这些步骤总共耗时:

$$2\sqrt{p}\left(\lambda + \frac{n^2}{p\beta}\right)$$

将这三项求和, 就得到了 Cannon 算法总执行时间的表达式:

$$\chi n^3 / p + 2(\sqrt{p} + 1)\left(\lambda + \frac{n^2}{p\beta}\right)$$

Cannon 算法的等效率特性怎么样呢? 串行算法的时间复杂度为  $\Theta(n^3)$ 。并行算法的通信复杂度为  $\Theta(n^2/p)$ 。将通信复杂度乘以进程数  $p$  得到总开销:  $T_o(n, p) = \Theta(\sqrt{pn^2})$ 。

所以行块矩阵乘法算法的等效率关系为:

$$n^3 \geq C\sqrt{pn^2} \Rightarrow n \geq C\sqrt{p}$$

而  $M(n) = n^2$ , 所以可扩展性函数为:

$$M(C\sqrt{p})/p = C^2 p / p = C^2$$

随着进程数的增加, 消耗相同的内存我们仍能保持效率不变, 所以 Cannon 算法具有很高的可扩展性。

同前面基于行的算法一样, Cannon 算法中计算和通信可以相互重叠, 这为优化提供了很大空间。如果在计算当前矩阵块相乘的时候还有足够的内存来存放新的  $A$  矩阵块和  $B$  矩阵块, 那么在进行乘法迭代前就可以对块的发送和接收进行初始化。在乘法结束后和下一次迭代开始前, 进程检查消息是否接收完毕。因为通信的复杂度是  $\Theta(n^2)$ , 计算的复杂度是  $\Theta(n^3)$ , 所以当矩阵规模足够大的时候, 通信可以完全为与计算所覆盖。

## 11.5 本章小结

本章我们开发了两种矩阵乘法的并行算法。第一种是基于矩阵按行分块, 第二种 (即 Cannon 算法) 是基于矩阵棋盘式分块。两个算法都把计算均匀地分解到各个进程中。Cannon 算法的通信量较小。等效率性分析也显示了 Cannon 算法有很好的可扩展性, 这是前一种

算法所不具备的。如果有足够的内存空间,两种算法都可以获益于通信/计算的重叠。

我们还探讨了串行矩阵相乘算法的性能问题。直接的计算方法其内存使用模式有这样的效果:当第二个源矩阵与 Cache 不匹配的时候,程序的 Cache 命中率就会很低。我们介绍了递归分解的矩阵相乘算法,如果矩阵太大不能放入高速缓存时,我们便递归地将其分解成小块后再进行计算。可以看到即使在矩阵规模超过 Cache 大小限制的时候,程序的 CPU 性能仍然非常高。

为了获得最好的性能,矩阵相乘的并行程序必须依赖于高速的串行矩阵相乘函数,本章中的递归算法就是一个很好的例子。

## 11.6 主要术语

Cannon's algorithm      Cannon 算法

## 11.7 参考文献

本章我们说明了递归矩阵相乘算法是如何提高 Cache 命中率的。Gustavson【48】指出递归变换是用于解决密集型线性代数问题的常用而有效的分块技术。

## 11.8 练习题

$$11.1 \quad \text{假设 } A = \begin{pmatrix} 1 & 2 & -3 & -2 \\ 4 & 1 & -1 & 3 \\ 3 & 2 & 1 & -4 \end{pmatrix} \text{ 和 } B = \begin{pmatrix} -2 & -3 \\ 3 & 2 \\ 4 & -1 \\ 1 & -4 \end{pmatrix}$$

(a) 求  $C=AB$ 。

(b) 考虑子矩阵

$$\begin{aligned} A_{00} &= \begin{pmatrix} 1 & 2 \end{pmatrix} & A_{01} &= \begin{pmatrix} -3 & -2 \end{pmatrix} & B_{00} &= \begin{pmatrix} -2 \\ 3 \end{pmatrix} & B_{01} &= \begin{pmatrix} -3 \\ 2 \end{pmatrix} \\ A_{10} &= \begin{pmatrix} 4 & 1 \\ 3 & 2 \end{pmatrix} & A_{11} &= \begin{pmatrix} -1 & 3 \\ 1 & -4 \end{pmatrix} \text{ 和 } & B_{10} &= \begin{pmatrix} 4 \\ 1 \end{pmatrix} & B_{11} &= \begin{pmatrix} -1 \\ -4 \end{pmatrix} \end{aligned}$$

$$\text{计算 } C = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix}.$$

并给出每块的计算结果。

11.2 在按行分块的矩阵相乘并行算法中,每个进程在自己部分的  $A$  矩阵和整个  $B$  相乘操作结束后宣告完成。如果在所有进程间复制  $B$ , 算法会简单很多。但这种方法的有什么

问题呢?

11.3 在对自己部分的  $A$  分块和  $B$  分块进行相乘运算的时候, 按行分块算法和 Cannon 算法都可以调用那个串行递归矩阵相乘算法。

(a) 为什么 Cannon 算法与按行分块算法相比, 与串行递归矩阵相乘算法能匹配得更好呢?

(b) 根据 (a) 中的问题, 修改串行递归矩阵相乘算法。

11.4 考虑本章两个算法中将计算与通信重叠起来所能得到的优化。假设  $p=16$ ,  $\beta=1.5 \times 10^6/\text{s}$ ,  $\lambda=250 \mu\text{s}$ ,  $\chi=10\text{ns}$ 。

(a)  $n$  为何值时按行分块算法中每次迭代中通信时间小于计算时间?

(b)  $n$  为何值时 Cannon 算法中每次迭代中通信时间将小于计算时间?

11.5 编写程序以实现 11.3 节中介绍的矩阵相乘并行算法。程序从文件中读取源矩阵, 并将结果写入文件, 文件名通过命令行参数确定。假设矩阵元素为双精度浮点数, 按照第 6 章介绍的形式存放于文件中: 先是矩阵行数和列数  $m, n$ , 然后是  $mn$  个双精度浮点数。

(a) 分别在 1, 2, 3, 4,  $\dots, p$  个处理器上, 用规模为 100、200、400 和 800 的方阵时进行测试, 忽略 I/O 时间。在同一个图上绘制 4 条加速曲线。

(b) 分别在 1, 2, 3, 4,  $\dots, p$  个处理器上, 用规模为 100、200、400 和 800 的方阵时进行测试, 考虑 I/O 时间。在同一个图上绘制 4 条加速曲线。

11.6 编写程序实现 11.4 节中的 Cannon 算法, 假设程序中的进程数是一个平方数。程序从文件中读取源矩阵, 并将结果写入文件, 文件名通过命令行参数确定。假设矩阵元素为双精度浮点数, 按照第 6 章介绍的方法存放在文件中: 先是矩阵行数和列数  $m$  和  $n$ , 然后是  $mn$  个双精度浮点数。

(a) 分别在 1, 4, 9,  $\dots, p$  个处理器上, 用规模为 100、200、400 和 800 的方阵时进行测试, 忽略 I/O 时间。在同一个图上绘制 4 条加速曲线。

(b) 分别在 1, 4, 9,  $\dots, p$  个处理器上, 用规模为 100、200、400 和 800 的方阵时进行测试, 考虑 I/O 时间。在同一个图上绘制 4 条加速曲线。

11.7 设计一个可以处理进程数为非平方数的 Cannon 算法。

11.8 编写基于 Cannon 算法的并程序, 要求即使进程数不是平方数, 也得把所有进程都利用到。程序从文件中读取源矩阵, 并将结果写入到文件, 文件名由命令行参数给定。假设矩阵元素为双精度浮点数, 按照第 6 章介绍的方法存放在文件中: 先是矩阵行数和列数  $m, n$ , 然后是  $mn$  个双精度浮点数。

(a) 分别在 1, 2, 3, 4,  $\dots, p$  个处理器上, 当矩阵规模为 100、200、400 和 800 的方阵时进行测试, 忽略 I/O 时间。在同一个图上绘制 4 条加速曲线。

(b) 分别在 1, 2, 3, 4,  $\dots, p$  个处理器上, 当矩阵规模为 100、200、400 和 800 的方阵时进行测试, 考虑 I/O 时间。在同一个图上绘制 4 条加速曲线。

# 第 12 章 线性方程组求解

Concern for man himself and his fate must always form the chief interest of all technical endeavors, concern for the great unsolved problems of the organization of labor and the distribution of goods—in order that the creations of our mind shall be a blessing and not a curse to mankind. Never forget this in the midst of your diagrams and equations.

Albert Einstein, Address at the California Institute of Technology, 1931

## 12.1 概 述

许多科学和工程的问题都具备线性方程组的形式。下面列出了一些产生这类问题的领域：

- 结构分析（土木工程）
- 热传导应用（机械工程）
- 电力网格分析（电气工程）
- 生产规划（经济学）
- 回归分析（统计学）

因为由实际问题导出的线性方程组常常非常庞大，所以我们应该学习如何在并行计算机上对它们进行有效地求解。

在 12.2 节我们定义了在以后章节中所使用到的基本术语。在 12.3 节和 12.4 节我们研究密集型线性方程组的直接求解方法。我们从分析上三角方程组开始，这类方程组可以用回代算法求解。然后我们再来研究如何求解密集型线性方程组：我们先利用高斯消去法把密集型线性方程组变换成上三角型方程组，然后再用回代算法进行求解。在开发并行高斯消去算法的过程中，我们将介绍一种被称为巡回赛的归约操作，并且说明如何在 MPI 中实现它。

通过对偏微分方程进行离散化，我们经常会得到稀疏矩阵线性方程组。对于这类方程组，迭代法比高斯消去法更合适。迭代法通过产生一个不断逼近结果向量的近似序列来求解线性方程组。我们在 12.5 节介绍 Jacobi 方法和 Gauss-Seidel 法。这些方法的收敛速度比较缓慢。相反，在 12.6 节中我们将介绍共轭梯度法，其收敛速度要快得多。

对于以上三种串行的算法：回代法、高斯消去法和共轭梯度法，我们基于不同数据分解方式分别开发了一对并行算法。在全部三种情况中，我们都会遇到适合每种分解办法的具体情况。

## 12.2 基本术语

拥有  $n$  个变量  $x_0, x_1, \dots, x_{n-1}$  的线性方程，可以表示为如下的形式：

$$a_0x_0 + a_1x_1 + \dots + a_{n-1}x_{n-1} = b$$



其中  $a_0, a_1, \dots, a_{n-1}$  和  $b$  是常数。

一个以  $x_0, x_1, \dots, x_{n-1}$  为变量的有限个线性方程所组成的集, 称为线性方程组或者线性系统。数集  $s_0, s_1, \dots, s_{n-1}$  称为一个线性方程组的解, 当且仅当通过替换  $x_0=s_0, x_1=s_1, \dots, x_{n-1}=s_{n-1}$  之后该线性系统中的每一个方程都得到满足。

一个包含  $n$  个线性方程且有  $n$  个变量的方程组:

$$\begin{aligned} a_{0,0}x_0 + a_{0,1}x_1 + \dots + a_{0,n-1}x_{n-1} &= b_0 \\ a_{1,0}x_0 + a_{1,1}x_1 + \dots + a_{1,n-1}x_{n-1} &= b_1 \\ \dots & \\ a_{n-1,0}x_0 + a_{n-1,1}x_1 + \dots + a_{n-1,n-1}x_{n-1} &= b_{n-1} \end{aligned}$$

通常表示为  $Ax=b$ , 其中  $A$  是一个  $n \times n$  的矩阵, 它包含所有的  $a_{i,j}$ , 而  $x$  和  $b$  是长度为  $n$  的向量, 分别用来储存各个  $x_i$  和  $b_i$ 。

一个  $n \times n$  的矩阵  $A$  在满足如下条件时, 称为对称带状矩阵, 其中  $w$  称为半带宽:

$$i-j > w \Rightarrow a_{i,j} = 0 \text{ 且 } j-i > w \Rightarrow a_{i,j} = 0$$

换句话说,  $A$  中的所有非零元素或者存在于主对角线上, 或者存在于主对角线上方或下方最近的  $w$  条对角线上。

当:

$$i > j \Rightarrow a_{i,j} = 0$$

$n \times n$  的矩阵  $A$  是上三角矩阵。

当:

$$i < j \Rightarrow a_{i,j} = 0$$

$n \times n$  矩阵  $A$  是下三角矩阵。

当:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, 0 \leq i < n$$

矩阵是严格对角占势的。

当  $a_{i,j} = a_{j,i}$ , 其中  $0 \leq i, j \leq n-1$ ,  $n \times n$  矩阵  $A$  是对称的。

当对每一个非零向量  $x$  和它的转置  $x^T$ , 积  $x^T A x > 0$ ,  $n \times n$  矩阵  $A$  是正定的。

## 12.3 回代法

回代法是用来求解线性方程组  $Ax=b$  的一种算法, 其中  $A$  是上三角矩阵。在这一节当中, 我们将研究串行回代算法并且在多处理器上通过不同的数据划分方法来研究它的性能。

### 12.3.1 串行算法

我们首先来看一个回代法的示例。

假设我们想求解下面这个方程组:

$$\begin{aligned}
 1x_0 + 1x_1 - 1x_2 + 4x_3 &= 8 \\
 -2x_1 - 3x_2 + 1x_3 &= 5 \\
 2x_2 - 3x_3 &= 0 \\
 2x_3 &= 4
 \end{aligned}$$

我们直接解最后一个方程，因为它仅有一个未知数。确定  $x_3=2$  之后，我们可以通过消掉其他方程中包含  $x_3$  的各项并相应地调整它们的  $b$  值来进行化简：

$$\begin{aligned}
 1x_0 + 1x_1 - 1x_2 &= 0 \\
 -2x_1 - 3x_2 &= 3 \\
 2x_2 &= 6 \\
 2x_3 &= 4
 \end{aligned}$$

现在，第三个方程中只有一个未知数。通过一个简单的除法我们可以得到  $x_2=3$ 。同一步类似，我们利用这个信息简化它上方的两个方程：

$$\begin{aligned}
 1x_0 + 1x_1 &= 3 \\
 -2x_1 &= 12 \\
 2x_2 &= 6 \\
 2x_3 &= 4
 \end{aligned}$$

我们已经简化了第二个方程，使它只含一个未知数。并且用  $a_{1,1}$  去除  $b_1$  得到  $x_1=-6$ ，然后从  $b_0$  中减去  $x_1 \times a_{0,1}$ ，我们有：

$$\begin{aligned}
 1x_0 &= 9 \\
 -2x_1 &= 12 \\
 2x_2 &= 6 \\
 2x_3 &= 4
 \end{aligned}$$

很容易得出  $x_0=9$ 。

串行回代算法的伪代码如图 12.1 所示。这个算法的时间复杂度为  $\Theta(n^2)$ 。

下面让我们考虑如何将这个算法并行化。我们首先画一张数据相关图，每个初始矩阵和向量中的元素用一个点来表示，每次给向量  $b$  的元素分配新值时都在图中增加一个点。一条从  $u$  点到  $v$  点的弧表示使用  $u$  的值来计算新的  $v$  值。数据相关图如图 12.2 所示。我们

```

a[0..n-1, 0..n-1] — coefficient matrix
b[0..n-1] — constant vector
x[0..n-1] — solution vector

for i ← n-1 down to 1 do
  x[i] ← b[i]/a[i, i]
  for j ← 0 to i-1 do
    b[j] ← b[j] - x[i] × a[j, i]
    a[j, i] ← 0 {This line is optional}
  endfor
endfor

```

图 12.1 回代算法求解  $Ax=b$  中的  $x$ ，  
其中  $A$  是一个上三角矩阵

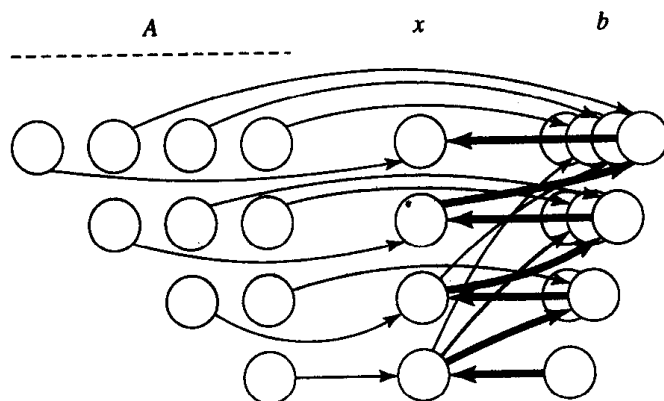


图 12.2 回代算法的数据相关图。随着算法的执行，  
 $b$  中元素的值不断改变。迭代的圆圈表明计算  
新值需要用到前一个值

用粗线表示图中的关键路径。从关键路径中我们可以很明显地看出, 向量  $x$  中的元素一次只能计算出一个。换句话说, 外层 for 循环并不能被并行执行。

然而, 我们可以并行执行内层的 for 循环。 $b_j$  的每一个新值都只依赖于它的前一个值、 $x_i$  的值和  $a_{j,i}$  的值。

### 12.3.2 面向行的并行算法

假设我们把  $A$  的每一行和相应的  $x$  和  $b$  的元素与一个原始任务相关联。当进行第  $i$  次迭代的时候, 与第  $j$  行相关的任务需要计算出  $b_j$  的新值, 这就意味着它需要访问当前的  $x_i$  和  $a_{j,i}$  的值。由于它控制  $A$  的第  $j$  行, 它可以直接访问  $a_{j,i}$ 。然而, 它不能直接访问  $x_i$ , 除非  $i=j$ 。因此任务  $i$  必须首先计算  $x_i$ , 然后将这个值广播到其他所有的任务中去。

让我们确定一下这种并行回代算法的时间复杂度。假设我们将原始任务汇集成  $p$  个更大的任务 (每进程一个任务), 进程  $k$  将控制满足  $i \bmod p = k$  的所有  $i$  对应的行。我们称这种方式为行交叉带状分解, 如图 12.3 (a) 所示。

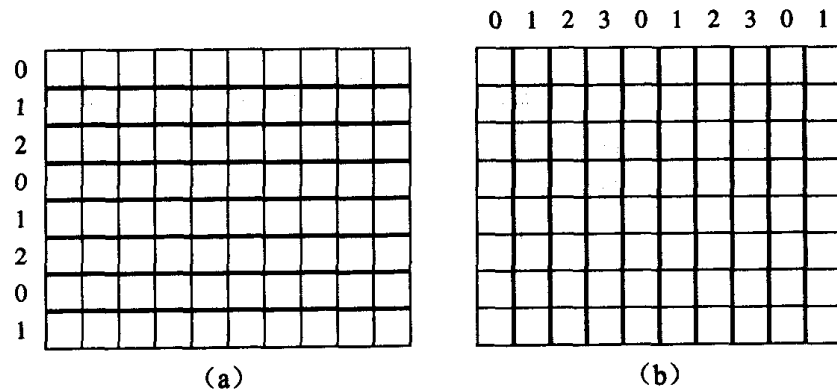


图 12.3 两种新的分解二维矩阵的方法 (前三种见图 8.3)。(a) 行交叉带状分解。

这个例子中 8 行被分解到 3 个进程中。(b) 列交叉带状分解。这个例子中 10 列被分解到 4 个进程中

在算法运行过程中, 任何进程执行第  $j$  次循环所进行的平均迭代次数大约为  $n/(2p)$ 。由于算法有  $n-1$  次迭代, 所以并行算法的计算复杂度为  $\Theta(n^2/p)$ 。

在每一次迭代过程中, 控制第  $i$  行的进程把  $x_i$  广播给其他进程。因为算法有  $n-1$  次迭代, 所以总的消息延迟是  $\Theta(n \log p)$ 。因为所有消息中只包含一个元素, 所以总消息传递时间也是  $\Theta(n \log p)$ 。

推导这一算法的等效关系和可扩展性函数在本章的末尾留作练习。

### 12.3.3 面向列的并行算法

在另一种设计中, 我们为  $A$  的每列关联一个原始任务。我们假设任务  $j$  ( $0 \leq j \leq n$ ) 负责  $A$  的第  $j$  列和  $x_j$ 。算法开始时, 任务  $n-1$  还负责向量  $b$ 。

我们以交叉的方式聚集任务, 形成一种矩阵按列交叉带状分解, 如图 12.3 (b) 所示。我们可以确定基于这种分解的并行算法的复杂度。

当第  $i$  次迭代时进程  $i$  负责计算  $x_i$  并更新向量  $b$ 。在第一次迭代中 ( $i=n-1$ )，进程  $n-1 \bmod p$  已经拥有  $A$  的第  $n-1$  列和向量  $b$ ，所以它可以在不进行任何通信的情况下算出  $x_{n-1}$  并且更新  $b$ 。然而，在第二次迭代中是需要通信的，进程  $n-2 \bmod p$  拥有  $A$  的第  $n-2$  列，但它没有  $b$  的副本 (除非  $p=1$ )。在第一次迭代时负责更新  $b$  的进程必须把它的  $n-1$  个元素传递给后续进程。

对于外层循环的每次迭代，一个进程负责计算  $x_i$  并更新  $b$ 。这里并没有并发的计算，因此并行算法的计算复杂度与串行算法的计算复杂度一样，都是  $\Theta(n^2)$ 。在两次迭代之间， $b$  的元素必须在两个进程间传递。平均传送的元素个数约为  $n/2$ 。由于有  $n-1$  次迭代，所以总通信延迟为  $\Theta(n)$ ，而总的消息传输时间为  $\Theta(n^2)$ 。

### 12.3.4 对比

面向行的并行回代算法，其计算时间复杂度为  $\Theta(n^2/p)$ ，消息传输时间为  $\Theta(n \log p)$ 。而面向列的算法的计算时间复杂度为  $\Theta(n^2)$ ，消息传输时间为  $\Theta(n^2)$ 。对于任意给定的  $p$ ，随着  $n$  的无限增长，面向行的算法最终会被证实具有更短的执行时间。

面向行的算法的总消息延迟是  $\Theta(n \log p)$ ，而面向列的算法总消息延迟为  $\Theta(n)$ 。因此对于任何给定的  $n$ ，随着  $p$  的无限增长，面向列的算法将最终被证实具有更短的执行时间。

我们可以画图来描述对于不同的  $n$  和  $p$  的值，两种算法中哪种在什么情况下占优。结果如图 12.4 所示。面向行的算法把计算负载分担到各个进程上，但是需要  $n$  步广播。因此当  $n$  较大， $p$  较小的时候占优。与之相反，面向列的算法虽然在计算中没有并行性，但是整个过程只需要发送  $n$  条点到点的消息。这使得它在  $n$  相对较小， $p$  相对较大的时候占优。

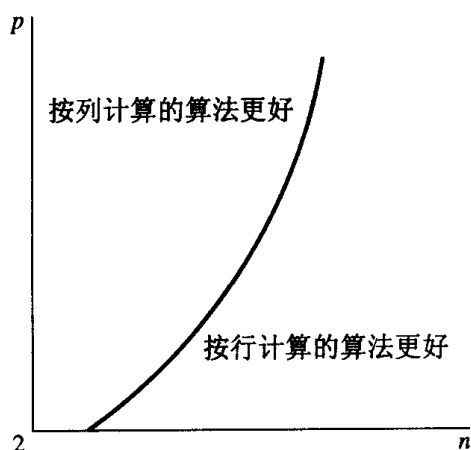


图 12.4 并行回代算法的实现中，面向行设计与面向列设计的比较两种设计的优劣依  $n$  和  $p$  的具体情况而定

## 12.4 高斯消去法

### 12.4.1 串行算法

高斯消去法是用来求解系数矩阵中非零元素任意分布的线性方程组的著名算法。高斯消去法把  $Ax=b$  归约为上三角方程组  $Tx=c$ ，这样可以利用回代算法求解  $x$ 。

我们在不改变解的值情况下对线性方程组执行三种操作【4】：

- 方程的每项乘以非零的常数；
- 交换两个方程的顺序；
- 把一个方程的倍数加到另一个方程上。

因此, 我们可以把线性方程组的任何一行替换为这一行与方程组中任意行的非零倍数之和。

我们看一个例子。这里有一个密集型线性方程组, 我们希望把它变成上三角形式:

$$\begin{array}{rrrrr} 4x_0 & +6x_1 & +2x_2 & -2x_3 & = & 8 \\ 2x_0 & & +5x_2 & -2x_3 & = & 4 \\ -4x_0 & -3x_1 & -5x_2 & +4x_3 & = & 1 \\ 8x_0 & +18x_1 & -2x_2 & +3x_3 & = & 40 \end{array}$$

系数  $a_{1,0}=2$ ,  $a_{0,0}=4$ 。2 除以 4 商为 0.5。如果用第 1 行与第 0 行的 -0.5 倍之和代替第 1 行, 那么第 1 行的第 1 项将变为 0。同样, 如果我们用第 2 行与第 0 行的 1 倍之和替换第 2 行, 第 2 行的第 1 项也会变成 0。用第 3 行和第 0 行的 -2 倍的和替换第 3 行, 则第 3 行的第 1 项变为 0:

$$\begin{array}{rrrrr} 4x_0 & +6x_1 & +2x_2 & -2x_3 & = & 8 \\ & -3x_1 & +4x_2 & -1x_3 & = & 0 \\ & +3x_1 & -3x_2 & +2x_3 & = & 9 \\ & +6x_1 & -6x_2 & +7x_3 & = & 24 \end{array}$$

既然已经使系数  $a_{0,0}$  下方的所有系数都变成了 0, 那么就把注意力转移到  $a_{1,1}$  下面的各列。我们用第 2 行与第 1 行的 1 倍之和替换第 2 行, 用第 3 行与第 1 行 2 倍之和替换第 3 行, 结果如下:

$$\begin{array}{rrrrr} 4x_0 & +6x_1 & +2x_2 & -2x_3 & = & 8 \\ & -3x_1 & +4x_2 & -1x_3 & = & 0 \\ & & +1x_2 & +1x_3 & = & 9 \\ & & +2x_2 & +5x_3 & = & 24 \end{array}$$

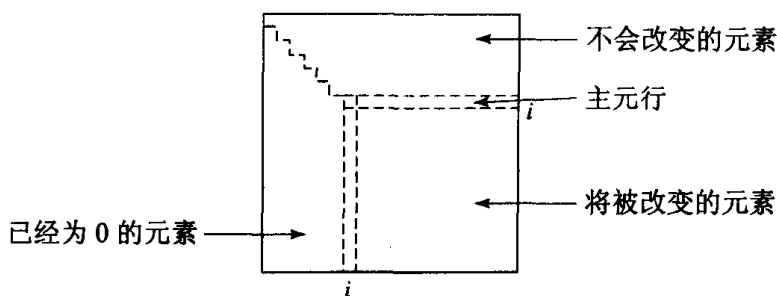
最后, 需要使  $a_{2,2}$  下方的系数变为 0。用第 3 行和第 2 行的 -2 倍的和替换第 3 行:

$$\begin{array}{rrrrr} 4x_0 & +6x_1 & +2x_2 & -2x_3 & = & 8 \\ & -3x_1 & +4x_2 & -1x_3 & = & 0 \\ & & +1x_2 & +1x_3 & = & 9 \\ & & & +3x_3 & = & 6 \end{array}$$

至此, 我们已经将密集的线性方程组转换成了上三角方程组。这时可以使用回代算法将方程组变成对角形式, 从而得到解向量。

图 12.5 说明了算法一次迭代的过程。对角线以下和第  $i$  列左边的所有非零元素已经被消去。在第  $i$  步, 通过用第  $j$  行与第  $i$  行的  $-a_{ji}/a_{ii}$  倍之和替换每一个第  $j$  行 (其中  $i+1 \leq j \leq n$ ), 我们可以消去第  $i$  列的对角线位置以下所有非零的元素。经过  $n-1$  次这样的迭代之后, 线性方程组就具备上三角形式了。

在上面描述的直接高斯消去法中, 在进行第  $i$  次迭代时, 第  $i$  行便成为了枢轴, 我们将使用这一行把第  $i$  列在对角线位置以下的非零元素全部变为 0。然而, 如果核心元素  $a_{i,i}$  接近 0, 用它做为除数所得到的结果将有严重的舍入误差。因此, 这种方法在数字计算机上的数值稳定性较差。

图 12.5 高斯消去法第  $i$  次迭代使得第  $i$  列里位置  $i$  以下的所有元素变为 0。

我们将从第  $i$  行下的每一行  $j$  中减去第  $i$  行的某个倍数。

这个倍数选择的原则是执行减法后第  $j$  行中第  $i$  列的元素变为 0

幸运的是，高斯算法的一个简单变形——部分选主元高斯消去法，可以产生可靠的结果。在部分选主元的高斯消去法的第  $i$  步，我们在第  $i$  行到第  $n-1$  行中寻找第  $i$  列元素的绝对值最大的行并将这一行与第  $i$  行交换（变成枢轴）。一旦这些都完成了，算法利用枢轴行（现在保存为第  $i$  行）的倍数把第  $i+1$  行到第  $n-1$  行中的第  $i$  列上的所有非零元归约成零，如图 12.6 所示。

在图 12.7 中显示了高斯消去法的一个串行实现，先部分选主元然后使用回代算法。这个算法有两个值得注意的特性。第一，注意没有独立的数组来储存向量  $b$ 。因为对于  $b$  的元素的操作与对于  $A$  的元素的操作是相同的，我们用  $b$  邻接  $A$  创建一个  $n$  行  $n+1$  列的增广矩阵就可以了。因此，在这个算法中数组  $a$  代表这个增广矩阵。

第二，注意到算法在每次迭代中间接地访问枢轴行，而不是直接地将交换枢轴行与第  $i$  行相交换。数组元素  $loc[i]$  记录了第  $i$  次迭代中所用的枢轴行的下标。

## 12.4.2 并行算法

让我们来研究高斯消去法是否适合并行化。其串行算法要求大约  $2n^3/3$  次浮点操作【95】。大多数这种操作发生在最内层的 for 循环内部。算法的数据相关性

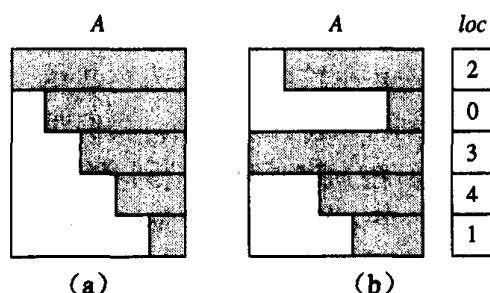


图 12.6 高斯消去法和部分选主元高斯消去法的比较

```

for i ← 0 to n - 1
  loc[i] ← i
endfor

for i ← 0 to n - 1
  {找到主元行 picked}
  magnitude ← 0
  for j ← i to n - 1
    if |a[loc[j], i]| > magnitude
      magnitude ← |a[loc[j], i]|
      picked ← j
    endif
  endfor
  tmp ← loc[i]
  loc[i] ← loc[picked]
  loc[picked] ← tmp

  {将第 i 列中位于未标记行中的元素消为 0}
  for j ← i + 1 to n - 1
    t ← a[loc[j], i] / a[loc[i], i]
    for k ← i + 1 to n + 1
      a[loc[j], k] ← a[loc[j], k] - a[loc[i], k] × t
    endfor
  endfor
endfor

{回代}
for i ← n - 1 down to 0
  x[i] ← a[loc[i], n] / a[loc[i], i]
  for j ← 0 to i - 1
    a[loc[j], n] ← a[loc[j], n] - x[i] × a[loc[j], i]
  endfor
endfor

```

图 12.7 先部分选主元然后回代的串行高斯消去算法

研究也表明以位于最内层下标为  $k$  的 for 循环和位于中间层的下标为  $j$  的 for 循环都可以拿来并行执行。换句话说,一旦找到枢轴行,对所有未被标记的行的修改就可以同时进行。在每一行内,一旦系数  $a[loc[j], i]/a[loc[i], i]$  被计算出来了,那么这一行内从位置  $i+1$  开始的  $n-1$  个元素的修改工作也可以同时进行。因此这个算法较适合于并行处理。

我们将基于两种不同的数据分解方式对算法进行并行化。

### 12.4.3 面向行的算法

我们把  $A$  的每一行和  $b$  与  $x$  中相应的元素关联到为原始任务。如果我们检查第  $i$  次迭代的数据相关性,我们发现为了决定枢轴行,我们需要对分布在不同任务中的第  $i$  列的各个元素进行归约。

我们把决定枢轴行过程称为巡回赛 (tournament), 因为与枢轴行的第  $i$  列中存储的具体数值 (胜利者的得分) 相比, 我们对枢轴行的位置 (胜利者的身份) 更感兴趣。

我们如何在 MPI 中实现巡回赛操作呢? 一种方式是进行两次全归约操作。在第一次全归约中, 每一个未标记行的对应任务将生成该行中第  $i$  列元素的绝对值。(如果该行先前曾被用作枢轴行, 其任务将返回 0, 以保证这一行不会再次被选中。) 在第一次全归约后, 每一个任务都知道了所有任务生成的最大值。现在我们进行第二次归约。每个任务用它的值与优胜值比较。如果它的值与优胜值匹配, 则输出它的 ID 值; 否则输出 -1。通过再次进行 maximum 操作符下的全归约之后, 每个任务便知道了一个含有最大值的任务的身份 (我们说一个任务而不是这个任务, 是因为含有这个最大值的任务数可能不止一个)。

虽然这样可以完成工作, 但是连续进行两次全归约则显得很浪费。幸运的是, MPI 提供了一种在一次全归约中实现巡回赛的方法。对于一个由点对组成的序列  $(v_0, i_0), (v_1, i_1), \dots, (v_{p-1}, i_{p-1})$ , 通过使用 MPI\_MAXLOC 操作符可以找到  $v_0, v_1, \dots, v_{p-1}$  中的最大值  $v_k$  并且返回  $(v_k, i_k)$ 。

为了在归约操作中使用 MPI\_MAXLOC (或者与它类似的 MPI\_MINLOC), 你必须提供一种用于表示 (数值, 下标) 的数据类型。MPI 提供了 6 种预先定义的用于表示 (数值, 下标) 的数据类型, 如表 12.1 所示。

表 12.1 MPI 中用于表示 (数值, 下标) 对的数据类型。注意下标必须为 int 类型的变量

MPI 数据类型	含义
MPI_2INT	两个整数
MPI_DOUBLE_INT	双精度数加整数
MPI_FLOAT_INT	浮点数加整数
MPI_LONG_INT	长整数加整数
MPI_LONG_DOUBLE_INT	长双精度数加整数
MPI_SHORT_INT	短整数加整数

为了存放 (数值, 下标) 对, 同样需要在 C 中创建一个结构。我们把该结构传送到归约函数。下面的代码展示了如何在高斯消去法的并行实现中利用这个特性:

```
struct {
```

```

double value;
int index;
} local, global;
...
local.value = fabs(a[j][i]);
local.index = j;
...
MPI_Allreduce (&local, &global, 1, MPI_DOUBLE_INT,
               MPI_MAX_LOC, MPI_COMM_WORLD);

```

通信域中的每一个进程通过 `local` 结构把它的（数值，下标）对传递给 `MPI_Allreduce`。当函数返回时，最大值和相关的下标将存放在 `global` 结构中。

在第  $i$  次迭代进行的过程中，枢轴行的确定需要两个步骤。首先，每个进程在它所负责的未标记的行中寻找在第  $i$  列数值最大的那一行。这一步的时间复杂度为  $\Theta(n/p)$ 。第二，进程加入寻找枢轴行的巡回赛。巡回赛的时间复杂度为  $\Theta(\log p)$ 。

在每次迭代过程中，这一步都是首先要完成的，之后还有一个涉及通信的任务，如图 12.8 所示。为了计算  $a[j, k]$  的新值，其对应的任务需要访问  $a[j, i]$ 、 $a[picked, i]$  和  $a[picked, k]$  的值。每个任务至少分配到  $A$  的一行，所以这个任务既然控制了  $a[j, k]$ ，那么也就控制着  $a[j, i]$ 。但是  $a[picked, i]$  和  $a[picked, k]$  的值可能为另一个任务所控制。因此，我们还需要做一次广播。

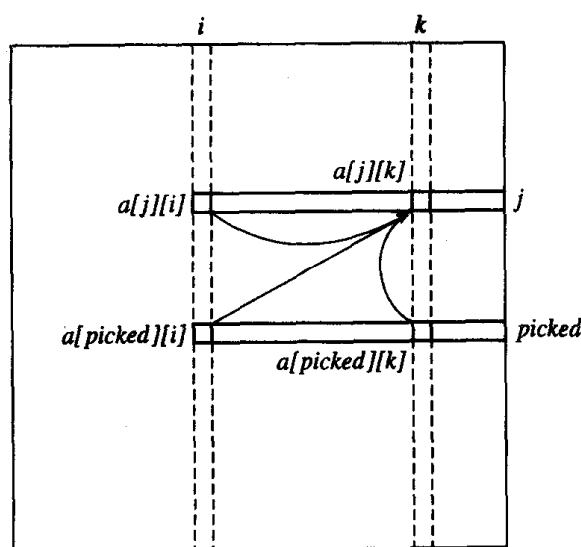


图 12.8 要更新  $a[j][k]$  的值，需要获得  $a[j][i]$ 、 $a[picked][i]$  和  $a[picked][k]$  的值

控制着 *picked* 行的任务在  $k$  下标的 `for` 循环的每次迭代中，都得把  $a[picked, k]$  广播给其他任务，但是这样会导致每次迭代都要广播  $\Theta(n)$  步。在 `for` 循环进行之前进行广播会是个更合理的选择。换句话说，*picked* 行的第  $i$  到第  $n$  个元素应当被立即广播到其他任务中。广播消息中元素的平均数量约为  $n/2$ 。因此广播的消息延迟是  $\Theta(\log p)$ ，而广播的消息传输时间为  $\Theta(n \log p)$ 。

综合考虑两步通信的总开销，我们发现面向行的并行高斯消去算法的总消息延迟为  $\Theta(n \log p)$ ，而总的消息传输时间为  $\Theta(n^2 \log p)$ 。

我们还仍需判断如何把原始任务聚集成更大的任务，以能关联到各个 MPI 进程。使用



按行分块看起来是一个很好的策略。使用部分选主元意味着（没有其他信息的情况下）在某次迭代中，任何未标记行被选为枢轴的概率均相同。在算法的执行过程中，每个进程上未被标记的行数将保持平衡，同时并行算法的计算复杂度为  $\Theta(n^3/p)$ 。

让我们判定这种并行高斯消去算法的等效率特性。 $p$  个进程上总的通信开销是  $\Theta(n^2 p \log p)$ 。因此我们有：

$$n^3 \geq C n^2 p \log p \Rightarrow n \geq C p \log p$$

下面我们找出这个并行系统的可扩展性函数。因为  $M(n) = n^2$ ，所以有：

$$M(C p \log p) / p = C^2 p^2 \log^2 p / p = C^2 p \log^2 p$$

可见，这个算法的可扩展性比较差。

#### 12.4.4 面向列的算法

下面来研究一下另外一种基于高斯消去法的并行化方案。我们把一个原始任务同  $A$  的每一列相关联，同时将向量  $b$  与另一个原始任务相关联。

在算法的第  $i$  次迭代中， $A$  中第  $i$  列对应的任务负责寻找绝对值最大的候选元素。它只需要考虑当前尚未被用做枢轴的那些行。所以，每个任务都需要有一份  $loc$  数组的副本。

在一次迭代中，面向列的算法在识别枢轴上所花费的时间为  $\Theta(n)$ 。

在负责第  $i$  列的任务识别到枢轴之后，它必须把该枢轴的标志和对应行的元素值广播给其他任务，这些任务需要这个信息以完成对各自元素的更新工作。这一步的消息延迟为  $\Theta(\log p)$ ，消息传递时间则为  $\Theta(n \log p)$ 。对整个算法而言，总的消息延迟为  $\Theta(n \log p)$ ，而总的消息传输时间为  $\Theta(n^2 \log p)$ 。

如果我们以交叉的方式对原始任务进行聚集，我们最后将得到一个矩阵  $A$  的基于列的交叉带状分解。该分解确保了在算法执行过程中工作量平衡地分布在各个任务上。

每次迭代的过程中，每个进程都所负担的计算量基本上相等。所以该并行算法的计算复杂度为  $\Theta(n^3/p)$ 。

面向列的算法其等效率函数与面向行的算法一样，因而它并不是高度可扩展的。

#### 12.4.5 对比

面向行的和面向列的高斯消去法，都将以  $j$  和  $k$  为索引的双层嵌套循环内部的计算工作均匀地划分给了各个进程。面向行的算法需要进程负责把枢轴广播给其他进程，而面向列的算法需要进程负责在第  $i$  次迭代中把第  $i$  列的信息广播给其他进程。因此，从这两方面来看，这两个并行算法执行时间的期望值应大致相同。

因此，这两个算法的最重要的不同点在于对枢轴的识别。面向行的算法以一次全归约通信的开销在进程之间对寻找枢轴的工作进行了分解。面向列的算法串行地实现了枢轴的定位，而不需要任何通信。因此面向行的算法在  $n$  相对较大而  $p$  相对较小的情况下占优，而面向列的算法在  $p$  相对较大而  $n$  相对较小的情况下占优。注意到在前面讨论面向行和面向列的回代算法时我们也得到了相似的结论。

这两种算法的可扩展性都不大好。我们需要找到一个进一步减少通信开销的方法。

### 12.4.6 面向行的流水线算法

我们讨论的面向行和面向列的算法都整齐地将并程序的执行划分为通信和计算两个阶段,从这个意义上说他们都是同步的。我们考虑面向行的算法。首先进程都加入一个巡回赛来确定枢轴。然后控制枢轴的进程把它广播给其他的进程。在这个广播步骤之后,所有这些进程使用枢轴来对各自控制的子矩阵的部分进行归约。这个步骤一旦完成,这些进程将再次加入巡回赛以确定下一个枢轴。

这种同步的方法其缺点在于进程在广播步骤中不执行任何计算,而且这些广播的累积时间复杂度是  $\Theta(n^2/p)$ , 如此大的复杂度导致这种并行算法的可扩展性很差。

我们必须找到一种能将通信时间与计算时间重叠起来的方法。如果事先就知道迭代  $i$  的枢轴的话这是可以实现的。前面我们曾经引入了按行部分选主元法来确保数字的稳定性,但是这样做不可能预测出迭代  $i$  中作为枢轴的行。如果不是对矩阵的行做部分选主元,而是对它的列做会怎么样呢? 在第  $i$  次迭代中考察第  $i$  行并找出绝对值最大的元素并把这个元素作为枢轴元素,然后归约系数矩阵中的  $i+1$  到  $n-1$  行,并把它们中包含枢轴元素的那一列的元素全部清零。基于列的部分选主元高斯消去法的伪代码如图 12.9 所示。

让我们来从基于列的部分选主元串行算法设计一种并行算法。我们选择对增广矩阵进行按行交叉分解,并把进程组织为一个逻辑上的环状结构。

当算法开始执行时,进程 0 搜索第 0 行,确定绝对值最大的元素所在的列。搜索完成后,它就向任务 1 发送一个包含枢轴元素下标和第 0 行所有元素和消息。在消息发送的过程中,进程 0 可以同时对它上面的那部分增广矩阵的其余行进行归约。

进程 1 一直等待,直到收到进程 0 发送来的包含第 0 行元素的消息。收到以后,它立刻将这个信息传给进程 2。然后再根据第 0 行中的元素和枢轴元素的信息对它那部分矩阵进行归约。这个时候它就能确定第 1 行的枢轴元素。确定了枢轴元素后它把第 1 行所有元素和新选择的枢轴发送给进程 2。在发送这些消息的同时,进程 1 可以使用第 1 行来归约属于它的那部分矩阵的各行。

第 0 行从进程 0 被发送到进程 1,然后又从进程 1 被发送到进程 2,如此下去直到到达进程  $p-1$ 。第 1 行从进程 1 到进程 2,再从进程 2 到进程 3,沿着进程的逻辑环发送,直到到达进程 0。一个进程发送的每一行都沿着环路往下传直到到达该进程的前一个进程。

```

for  $i \leftarrow 0$  to  $n$ 
     $loc[i] \leftarrow i$ 
endfor
for  $i \leftarrow 0$  to  $n-1$ 
    {找到主元行  $picked$ }
     $magnitude \leftarrow 0$ 
    for  $j \leftarrow i$  to  $n-1$ 
        if  $|a[i, loc[j]]| > magnitude$ 
             $magnitude \leftarrow |a[i, loc[j]]|$   $picked \leftarrow j$ 
        endif
    endfor
endfor

```

```

    tmp ← loc[i]
    loc[i] ← loc[picked]
    loc[picked] ← tmp
    {将第 loc[i] 列中位于第 i+1 行到第 n-1 行的元素消为 0}
    for j ← i+1 to n-1
        t ← a[j, loc[i]] / a[i, loc[i]]
        for k ← i to n-1
            a[j, loc[k]] ← a[j, loc[k]] - a[i, loc[k]] × t
        endfor
    endfor
    {回代}
    for i ← n-1 down to 0
        x[loc[i]] ← a[i, n] / a[i, loc[i]]
        for j ← 0 to i-1 do
            a[j, n] ← a[j, n] - x[loc[i]] × a[j, loc[i]]
        endfor
    endfor

```

图 12.9 部分选主元的串行高斯消去法, 以及随后的回代过程

我们前面实现的两种并行高斯消去算法都依赖于广播操作。作为替代, 本算法使用了沿着进程环发送一系列点到点消息的通信模式。为什么这种方法更好呢? 通过对消息流进行流水线化的操作, 使得并行算法有两个明确的优点。首先, 它有利于进行异步计算: 进程可以一得到枢轴行以后对进程上负责的那部分增广矩阵进行归约。其次, 它使得进程将通信时间和计算时间有效地重叠了起来。

因为总的归约时间是  $\Theta(n^3/p)$ , 而总的消息传输时间是  $\Theta(n^2)$ , 所以如果  $n$  足够大, 我们就可以很合理地假设传输一行元素的时间与进行矩阵元素归约的时间可以相互重叠。同时消息启动时间与计算是不能相互重叠的。由于一个进程必须发送  $n-1$  条消息, 所以本算法的总的通信时间是  $\Theta(n)$ 。

让我们来确定该并行系统的等效率特性。串行执行时间是  $\Theta(n^3)$ , 并行化的开销是  $\Theta(np)$ 。因此:

$$n^3 \geq Cnp \Rightarrow n \geq \sqrt{Cp}$$

由于  $M(n) = n^2$ , 所以可扩展性函数为:

$$M(\sqrt{Cp}) / p = Cp / p = C$$

假设  $n$  足够大, 以使得消息传输时间基本上为计算时间所覆盖, 那么这个并行系统就有着完美的可扩展性。

## 12.5 迭代法

首先高斯消去然后进行回代是一种直接求解线性方程组的方法。算法通过执行步数事先可以确定的操作, 最后求出解向量。

当线性方程组的系数矩阵是密集矩阵的时候，高斯消去法效果很好。然而，如果我们用高斯消去求解一个稀疏的线性方程组（相对而言非零元素非常少），系数矩阵会逐渐为非零元素所填充。图 12.10 用一个较小的系统（9×9）为例说明了这个现象（当矩阵大小增加时，这种填充现象更富戏剧性，而且由非零元素所组成的斜对角线之间的距离还会增加）。元素填充是不合需要的，因为它增大了对存储需求并且增加了总的操作次数。

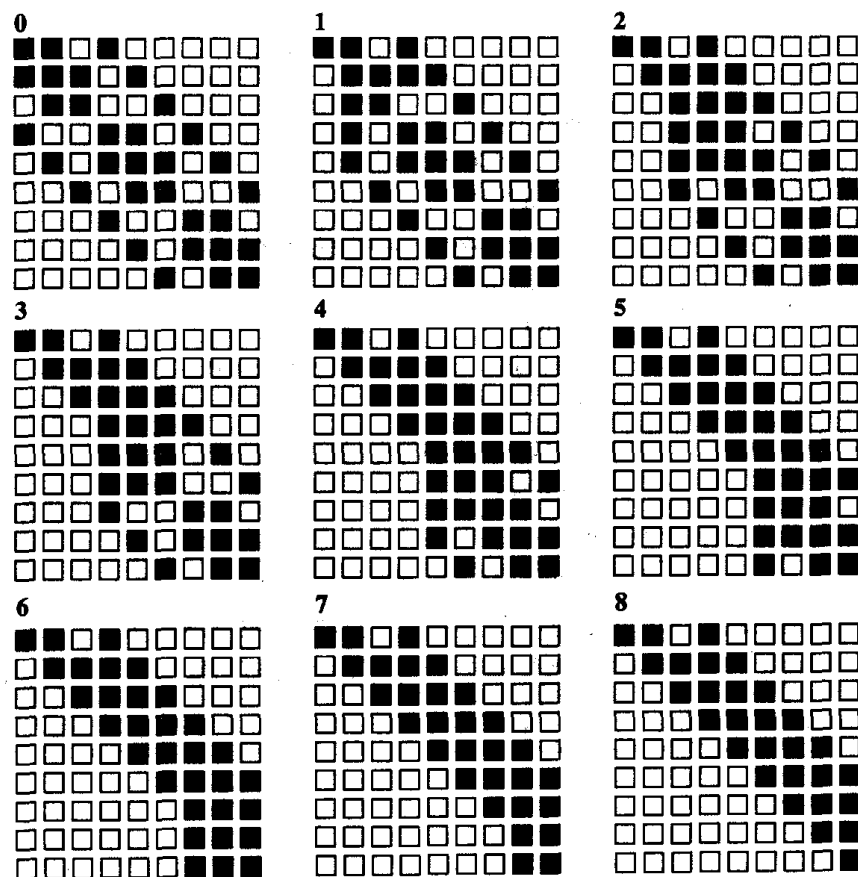


图 12.10 用高斯消去法求解稀疏线性方程组的效果。标记为 0 的矩阵是算法开始时系统的状态。

黑色方块代表非零系数；白色方块代表零元素。标记 1 到 8 的矩阵显示了进行每一次迭代后矩阵系数的变化

迭代法是一种用一组近似值来逼近真实解的算法。典型的迭代法都比直接求解所需要的存储量更少。由于避免了对零元的操作，这些算法还能进一步减少计算量。它们通常也都适于并行化。这一节我们考虑两个用于求解线性方程组的简单迭代法。

假设我们要求解线性方程组  $Ax=b$ ，其中  $A$  对角线上的元素都是非零元；也就是说，对于  $0 \leq i < n$ ， $a_{ii} \neq 0$ 。Jacobi 法开始用一个初始向量  $x^0$  作为近似解。它重复地根据当前近似值  $x^k$  计算新的近似值  $x^{k+1}$ ，所用的公式是：

$$x_i^{k+1} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^k \right)$$

图 12.11 是 Jacobi 迭代法的伪代码。

```

 $a[0..n-1, 0..n-1]$  — 系统矩阵
 $b[0..n-1]$  — 常数向量
 $new[0..n-1]$  — 结果向量的新值
 $sum$  — 部分结果累计
 $x[0..n-1]$  — 结果向量
for  $i \leftarrow 0$  to  $n-1$  do
     $x[i] \leftarrow 0$ 
endfor
repeat
    for  $j \leftarrow 0$  to  $n-1$  do
         $sum \leftarrow 0$ 
        for  $k \leftarrow 0$  to  $n-1$  do
            if  $k \neq j$  then
                 $sum \leftarrow sum + a[j,k] \times x[k]$ 
            endif
        endfor
         $new[j] \leftarrow (1/a[j,j]) \times (b[j] - sum)$ 
    endfor
    for  $j \leftarrow 0$  to  $n-1$  do
         $x[j] \leftarrow new[j]$ 
    endfor
until values in  $x$  converge

```

图 12.11 Jacobi 法是用于求解线性方程组  $Ax=b$  的迭代方法，它要求系数矩阵中主对角元均为非零元素

图 12.12 举例说明了用 Jacobi 法成功地求解一个有两个未知变量的方程组的过程中解向量的变化。

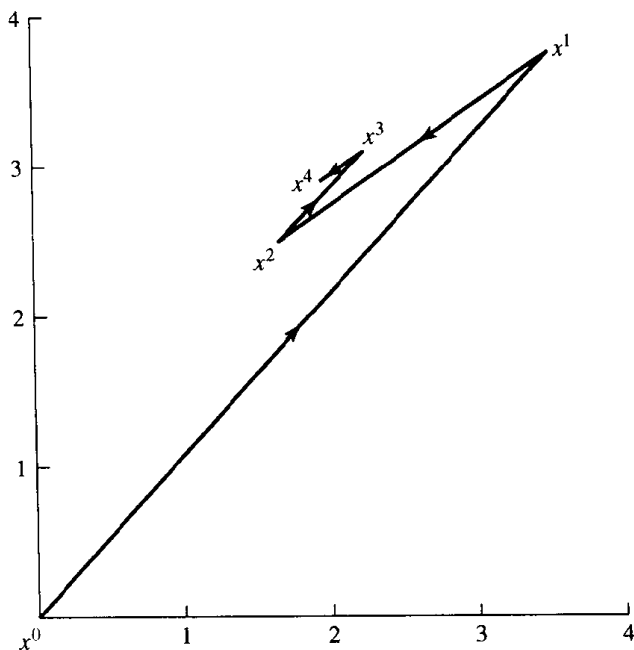


图 12.12 向量点  $x^1, x^2, x^3, x^4$  是 Jacobi 法在求解有方程组  $2x+y=7$  和  $x+3y=11$  的方程组时产生的，初始向量为  $x^0 = (0,0)$ 。 $x$  的值成功地收敛到解向量  $(2,3)$

注意, Jacobi 法中从  $x^k$  计算  $x^{k+1}$  的过程可以很好地并行化:  $x^{k+1}$  的每一个新元素均由  $x^k$  的值通过计算得到。如果我们每次都用最近一次得出的  $x_i$  的值来求解的话, 收敛速度会更快。要完成这一功能, 我们可以在伪代码中把公式:

$$new[j] \leftarrow (1/a[j, j]) \times (b[j] - sum)$$

替换为:

$$x[j] \leftarrow (1/a[j, j]) \times (b[j] - sum)$$

同时删去将向量  $new$  复制到向量  $x$  的 for 循环。如此改变后得到的算法称为 Gauss-Seidel 法。

前面我们提到一个矩阵如果严格对角占优的话, 必须满足:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, 0 \leq i < n$$

如果系数矩阵  $A$  是严格对角占优的, Jacobi 法和 Gauss-Seidel 法都能在任意初始向量  $x^0$  下收敛到  $Ax=b$  的解。

即使 Jacobi 法和 Gauss-Seidel 法确保能够收敛到一个解, 它们的收敛速度也太慢以至于并不适合实际使用。出于这个原因, 我们将不会开发这两个方法的并行算法。在第 12.6 节我们会提出一个收敛性好得多的迭代算法。

## 12.6 共轭梯度法

前面提到过, 一个  $n \times n$  的矩阵  $A$  是正定矩阵的充要条件是: 对每个非零向量  $x$  及其转置  $x^T$ , 都满足  $x^T Ax > 0$ 。如果  $A$  是对称的且正定的, 则使函数:

$$q(x) = \frac{1}{2} x^T Ax - x^T b + c$$

取得最小值的  $x$  向量, 就是  $Ax=b$  的解。共轭梯度法是众多通过求  $q(x)$  最小值来求解  $Ax=b$  的算法中的一种。如果忽略舍入误差, 共轭梯度法能够确保通过  $n$  次以内的迭代便可收敛到方程组的解。

### 12.6.1 串行算法

共轭梯度法的迭代形式是:

$$x(t) = x(t-1) + s(t)d(t)$$

向量  $x$  的新值是关于原向量  $x$ , 步长  $s$  和方向向量  $d$  的函数。

第一次迭代之前, 必须先设置  $x(0)$ 、 $d(0)$  和  $g(0)$  的初始值。在我们的算法实现中,  $x(0)$  和  $d(0)$  都被初始化为零向量,  $g(0)$  初始化成  $-b$ 。在进行第  $t$  次迭代时, 我们经过 4 个步骤计算出  $x(t)$ 。

第 1 步计算梯度:

$$g(t) \leftarrow Ax(t-1) - b$$

第 2 步计算方向向量:

$$d(t) \leftarrow -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$$

这里  $g(t)^T g(t)$  代表向量  $g(t)$  的转置与其自身的内积。

第 3 步计算步长:

$$s(t) \leftarrow -\frac{d(t)^T g(t)}{d(t)^T A d(t)}$$

第 4 步计算新的近似值:

$$x(t) \leftarrow x(t-1) + s(t)d(t)$$

共轭梯度法的伪代码见图 12.13。

图 12.14 说明了给定了前面 Jacobi 法 (见图 12.12) 中的示例系统 (两个方程组成的线性方程组), 共轭梯度法通过两次迭代得到解。

```

for i ← 0 to n - 1 do
    d[i] ← 0
    x[i] ← 0
    g[i] ← -b[i]
endfor
for j ← 1 to n do
    d1 ← Inner Product(g, g)
    g ← Matrix Vector Product (A, x)
    for i ← 0 to n - 1 do
        g[i] ← g[i] - b[i]
    endfor
    n1 ← Inner Product (g, g)
    if n1 < ε break endif
    for i ← 0 to n - 1
        d[i] ← -g[i] + (n1/d1) × d[i]
    endfor
    n2 ← Inner Product(d, g)
    t ← Matrix Vector Product (A, d)
    d2 ← Inner Product(d, t)
    s ← -n2/d2
    for i ← 0 to n - 1
        x[i] ← x[i] + s × d[i]
    endfor
endfor

```

图 12.13 串行共轭梯度算法

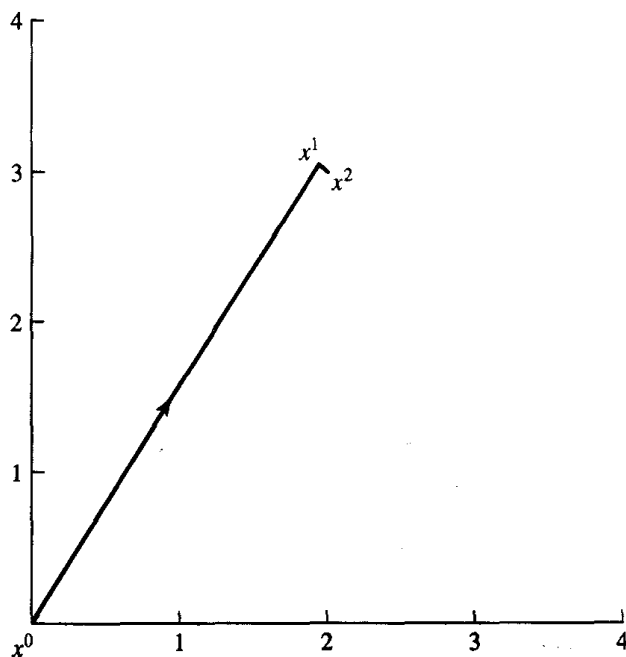


图 12.14 共轭梯度法需要两次迭代来求解  $2x+y=7$  和  $x+3y=11$  组成的方程组，解向量是  $(2, 3)$

假设矩阵  $A$  是对称带状矩阵，半带宽为  $w$ ，如图 12.15(a) 所示。这种情况下，寻找  $A$  中某列与另一个向量内积的时间复杂度是  $\Theta(w)$ ，因此矩阵-向量相乘步骤的时间复杂度为  $\Theta(nw)$ 。其他的向量操作，包括内积（点积）操作，时间复杂度均为  $\Theta(n)$ 。

## 12.6.2 并行算法

我们已经在第 8 章中讨论了完成矩阵-向量相乘的并行算法。为了利用矩阵  $A$  是带状矩阵的特点，我们需要对这个算法进行修改。具体的说，进程将只存储  $A$  的各行中非零的那部分，如图 12.15(b) 所示。这既可以节省内存又能使算法的执行速度更快，但是这也意味着我们需对矩阵向量相乘算法中的各种下标进行修改。

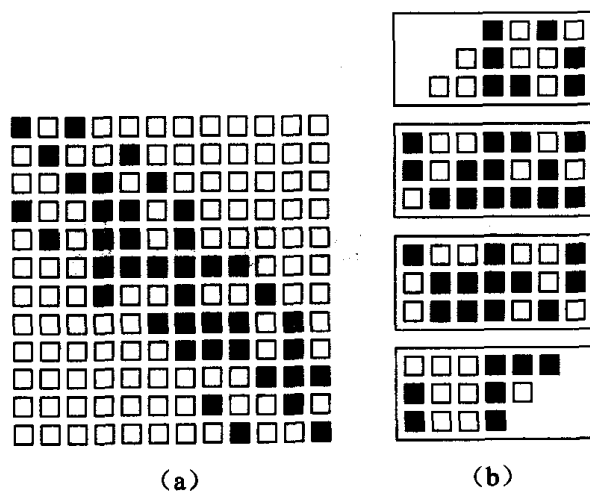


图 12.15 对一个带状矩阵的按行分块。(a) 一个半带宽为 3 的对称带状矩阵。

所有的非零元（用黑色方格表示）都在主对角线或者距主对角线距离不超过 3 的对角线上。

(b) 将这个矩阵存储在四个进程中。由于矩阵的半带宽为 3，每一行可以用  $7 = 2 \times 3 + 1$  个元素表示。

注意每一行的第 4 个元素（也就是中间的元素）为矩阵主对角线上的元素



假设我们对  $A$  做按行分块, 同时为所有的向量保留副本。这时  $A$  与某个向量相乘时不需要任何通信过程, 但是需要一个全收集的通信来对结果向量进行复制。该并行算法总的时间复杂度为  $\Theta(n^2w/p + n\log p)$ 。

另一方面, 如果我们对向量做块分解, 则需要在矩阵向量相乘之前进行一次全收集通信。但是之后无需复制结果向量块。这种方法的总的时间复杂度与第一种相同, 即为  $\Theta(n^2w/p + n\log p)$ 。

我们来看看这两种不同的数据分布方式是如何影响算法中其他部分的复杂度的。首先考虑将向量复制在各个进程上的情况。由于每个进程都有每个向量的完整副本, 那么在每个循环的每次迭代中, 它都必须更新这些向量。修改向量  $g$ 、 $x$  和  $d$  的工作通过一个循环来完成, 这个循环的并行时间复杂度是  $\Theta(n)$ 。同样, 对两个  $n$  维向量做内积需要的时间也是  $\Theta(n)$ 。

现在我们考虑对向量做块分解的情况。这种情况下把一个向量初始化成 0 或用一个向量减去另外一个向量所需要的时间都是  $\Theta(n/p)$ 。另一方面, 执行一个内积运算需要每个进程找到其子向量的内积, 然后进行求和归约。这时内积操作的复杂度将会是  $\Theta(n/p + \log p)$ 。

如果固定  $p$  而增加  $n$ , 最后计算时间将成为决定性因素。在这种情况下对向量进行块分解并分配到各个进程的方法占优。如果固定  $n$  而增加  $p$ , 最后通信时间将成为主导因素, 这种情况下复制向量从而避免所有通信的方法占优。当  $n$  决定  $p$  时, 图 12.16 说明了两种数据分布方案各自占优的区域。

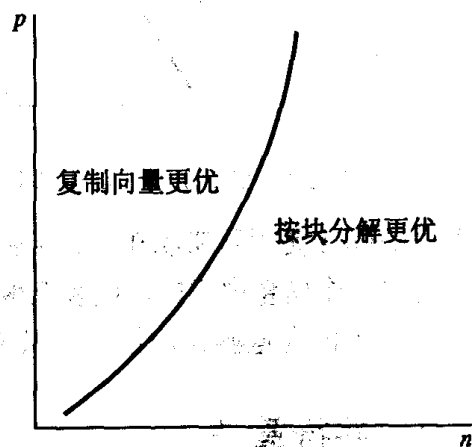


图 12.16 两种内积 (点积) 计算方式的比较。

当  $n \rightarrow \infty$  时, 在进程之间分配向量元素, 执行部分内积运算, 然后执行一个求和归约的过程相对复制向量并让每个进程执行串行算法而言会更快。然而,  $n$  值比较小的时候执行归约相比分布地进行计算所节省的时间要多, 因而这种情况下复制向量是更好的办法

## 12.7 本章小结

这一章中我们分别考察了直接和间接求解线性方程组的方法。设计了回代算法, 高斯消去法和共轭梯度法的并行版本。针对每种算法我们都各自开发了采用不同数据分布方式的两种实现。我们还发现每种算法中, 没有任何一个实现绝对占优。究竟哪一种数据分布方式下算法更快, 取决于问题的大小、可用处理器的数量、处理器的速度以及通信网络的速度。

## 12.8 主要术语

augmented matrix	增广矩阵
Gauss-Seidel method	Gauss-Seidel 法
solution	解
back substitution	回代
iterative method	迭代法
sparse matrix	稀疏矩阵
banded matrix	带状矩阵
Jacobi method	Jacobi 法
strictly diagonally dominant	严格对角占优
columnwise and rowwise interleaved striped decompositions	按列和按行交叉带状分解
linear equation	线性方程
symmetric	对称
linear system	线性系统
symmetrically banded	对称带状
lower triangular	下三角
system of linear equations	线性方程组
conjugate gradient method	共轭梯度法
partial pivoting	部分选主元
tournament	巡回赛
direct method	直接法
positive definite matrix	正定矩阵
upper triangular	上三角
Gaussian elimination	高斯消去
pivot row	枢轴

## 12.9 参考文献

Bertsekas 和 Tsitsiklis 的教材【9】是本章内容的主要来源。他们讨论了线性方程组的求解, 非线性问题, 最短路径问题和网络流量问题等算法。参考 Golub 和 Ortega【41】以得到一个关于共轭梯度法如何工作的详细的数学解释。

其他阐述并行数值算法的图书包括 Dongarra et al. 的【22】和 Fox et al. 的【33】。

Gallivan et al. 对密集线性代数计算的并行算法进行了综述【36】。

## 12.10 练习题

12.1 使用回代法求解第 12.4.1 节给出的上三角方程组。

12.2 (a) 推导第 12.3.2 节中面向行的并行回代算法的等效率关系和可扩展性函数。

(b) 设计一种使用流水线来对通信和计算进行重叠的并行回代算法。分析你设计的算法的时间复杂度, 并且确定其等效率关系和可扩展性函数。

12.3 向前取代(前代)是一种类似于回代的算法, 它用于求解下三角方程组。用伪代码写一个串行前代算法。

12.4 基于按行部分选主元的高斯消去然后回代的方法, 编写一个 C 程序来求解线性方程组  $Ax=b$ 。程序需要的方程组从文件输入, 该文件包括一个双精度数矩阵, 格式与第 6 和第 8 章所用的存储矩阵的格式一样。文件的前两个元素是两个整数, 第一个是  $n$  的值, 第二个值为  $n+1$ 。文件其余部分包括  $n(n+1)$  个双精度数, 分别对应  $A$  和  $b$  的元素, 按如下顺序存储:

$$\begin{aligned} &a_{0,0}, a_{0,1}, \dots, a_{0,n-1}, b_0, \\ &a_{1,0}, a_{1,1}, \dots, a_{1,n-1}, b_1, \dots, \\ &a_{n-1,0}, a_{n-1,1}, \dots, a_{n-1,n-1}, b_{n-1} \end{aligned}$$

12.5 使用上题的 C 程序作为起点, 用按行部分选主元法实现三个并行算法以求解线性方程组  $Ax=b$ , 程序必须从一个文件中读入方程组。数据文件的格式与练习 12.4 一样。程序要把结果向量  $x$  在标准输出上输出。

(a) 对增广矩阵  $Ab$  实行按行分块。

(b) 对增广矩阵  $Ab$  实行按列交叉带状分解, 程序通过广播的方式完成对列的传输。

(c) 对增广矩阵  $Ab$  实行按列交叉带状分解, 程序使用流水线把通信和计算重叠起来。

12.6 (a) 推导 12.4 节中设计的面向行的高斯消去程序执行时间的期望值。

(b) 推导 12.4 节中设计的面向列的高斯消去程序执行时间的期望值。

(c) 使用你的并行计算机的参数, 画一个类似于图 12.4 的图形, 以确定这两种设计所适用的  $n$  和  $p$  值的范围。

12.7 基于按列部分选主元的高斯消去然后回代的方法, 编写一个 C 程序来求解线性方程组  $Ax=b$ 。程序所要处理的方程组从文件输入。该文件包含一个双精度数矩阵, 格式与第 6 和第 8 章存储矩阵的格式一样。文件的前两个元素是两个整数, 第一个是  $n$  的值, 第二个值为  $n+1$ 。文件其余部分包括  $n(n+1)$  个双精度数, 分别对应了  $A$  和  $b$  的元素, 按如下顺序存储:

$$\begin{aligned} &a_{0,0}, a_{0,1}, \dots, a_{0,n-1}, b_0, \\ &a_{1,0}, a_{1,1}, \dots, a_{1,n-1}, b_1, \dots, \\ &a_{n-1,0}, a_{n-1,1}, \dots, a_{n-1,n-1}, b_{n-1} \end{aligned}$$

12.8 为了求解线性方程组  $Ax=b$ , 编写一个流水线化的并行程序, 以实现按列部分选

主元的高斯消去法。要求将增广矩阵  $Ab$  按照基于行的交叉带状分解方式进行划分。方程组从文件读入，其数据文件的格式与练习 12.7 的一样。程序要把结果向量  $x$  从标准输出上输出。

12.9 设计一个基于增广矩阵  $Ab$  棋盘式分解的高斯消去算法。确定该算法的等效率关系和可扩展性函数。

12.10 实现一个并行程序求解线性方程组  $Ax=b$ ，要求对增广矩阵  $Ab$  进行棋盘式分解。程序所需方程组从文件输入，该文件包括一个双精度数矩阵，格式与第 6 和第 8 章中用来存储矩阵的格式一样。程序要把结果向量  $x$  在标准输出上输出。

12.11 实现一个 C 程序通过共轭梯度法求解线性方程组  $Ax=b$ 。可以确定  $A$  是一个对称正定矩阵。程序所需的方程组从文件中输入。该文件包含一个双精度数矩阵，格式与第 6 和第 8 章中用来存储矩阵的格式一样。文件的前两个元素是两个整数，第一个是  $n$  的值，第二个值为  $n+1$ 。文件其余部分包括  $n(n+1)$  个双精度数，分别对应  $A$  和  $b$  的元素，按如下的顺序存储：

$$\begin{aligned} &a_{0,0}, a_{0,1}, \dots, a_{0,n-1}, b_0, \\ &a_{1,0}, a_{1,1}, \dots, a_{1,n-1}, b_1, \dots, \\ &a_{n-1,0}, a_{n-1,1}, \dots, a_{n-1,n-1}, b_{n-1} \end{aligned}$$

要求程序最终把结果向量  $x$  在标准输出上输出。

12.12 我们在第 6 章中首先使用的文件格式是基于矩阵是密集矩阵这一假设的。这个练习的目的在于开发一种用于存储对称带状矩阵的新的文件格式。

(a) 设计一种文件格式存储对称带状矩阵，文件的大小要与矩阵的行数与半带宽之积成比例。

(b) 实现一个 C 程序使用共轭梯度法求解稀疏线性方程组  $Ax=b$ ，可以确定  $A$  是一个对称正定矩阵。程序中的矩阵  $A$  从文件读入，使用你自己设计的文件格式。向量  $b$  从另一个文件读入。程序要把结果向量在标准输出上输出。

(c) 以该 C 程序为起点，实现一个共轭梯度法的并行程序（假定向量在进程间复制）。在你的并行计算机上用不同处理器数和不同的问题规模测试你的程序。

(d) 以该 C 程序为起点，实现一个共轭梯度法的并行程序（假定在进程间进行向量分块）。在你的并行计算机上用不同处理器数和不同的问题规模测试你的程序。

# 第 13 章 有限差分方法

Big words do not smite like war-clubs,  
Boastful breath is not a bow-string,  
Taunts are not so sharp as arrows,  
Deeds are better things than words are,  
Actions mightier than boastings.

Henry Wadsworth Longfellow, *The Song of Hiawatha*

## 13.1 概 述

常微分方程是一个包含有单个变量函数导数的等式。偏微分方程 (PDE) 是一个包含有两个或更多个变量的函数导数的等式。许多科学家和工程师研究的对象能够用偏微分方程来建模。下面是一些例子：

- 飞行器的双翼上的气流
- 人体内的血液循环
- 海洋中水流的循环 (如图 13.1 所示)

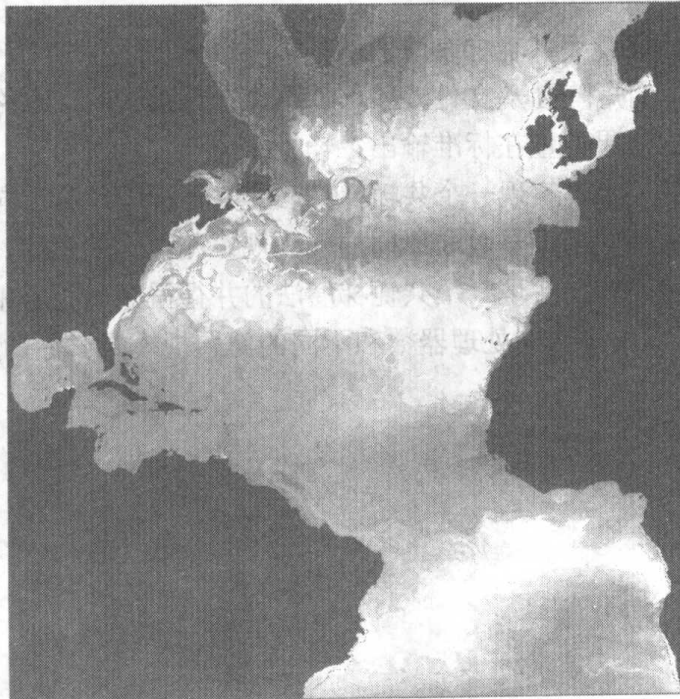


图 13.1 海洋表面的温度，来自一个高解析度 ( $1/12^\circ$ ，平均大约 6km 的网格空间) 北大西洋的有限差分数值计算，它使用的是迈阿密等密度海洋模型 (MICOM) (资料来源于迈阿密大学海洋和大气科学罗森斯代尔学院的科特西 MICOM 组)

- 桥梁负重时的变形
- 暴风雨的演变
- 大厦在地震袭击中的垂直震荡
- 玩具的强度
- CPU 散热器上的温度分布
- 低音的震动

通过使用图解法来获得一些简单的偏微分方程的解析解是可能的。但一般说来，我们不可能获得解析解，我们只能通过数值（计算）的方法找到一个近似解。这些数值的方法经常需要消耗大量的 CPU 时间。这就是为什么值得探索通过并行方法解决偏微分方程的原因。

数值地解决偏微分方程的方法主要是有限元方法和有限差分方法。本章主要涉及有限差分方法。

有限差分方法把偏微分方程转换为一个矩阵等式。就像我们在前一章看到的那样，有限差分方法产生的矩阵是一个稀疏矩阵（典型情况下，每行只有很少的几个元素非零）。有限差分方法的实现主要属于两个较大的范畴，这主要决定于它们如何表示稀疏矩阵。基于矩阵的实现办法准确地表达矩阵，使用可高效访问非零元素的数据结构。上一章中我们证明了如何用迭代法求解这些形式的线性方程组。在不用矩阵的实现方法中，我们将间接地访问矩阵的元素。这一章我们主要集中讨论不用矩阵的有限差分算法的实现上。

我们首先定义线性二阶偏微分方程。线性二阶偏微分方程可分为三类，每一类有不同的解法。我们也将介绍如何用差分商来近似一个连续函数在某点上的一阶和二阶导数。

我们将通过研究两个有限差分的案例——弦振荡问题和稳态热扩散问题——来展示程序并行化的技术。

## 13.2 偏微分等式

### 13.2.1 偏微分方程的分类

正如我们已经注意到的，偏微分方程是一个包含有两个或更多个变量导数的方程式。例如，假设  $u$  是  $x$  和  $y$  的函数： $u=f(x,y)$ 。我们把  $u$  对  $x$  的偏导数记作  $u_x$ ；类似的，我们记  $u$  对  $y$  的偏导数为  $u_y$ 。由于只取了一次导数，因而这些偏导数是一阶的。如果我们取  $k$  次导数，我们就称它有  $k$  阶。下面是三个  $u$  的二阶偏导数： $u_{xx}$ ， $u_{xy}$  和  $u_{yy}$ 。

二阶偏微分方程至多含有二阶的偏导数。二阶偏微分方程是最常用的用来解决物理科学和工程中的问题的偏微分方程。

线性二阶偏微分方程有如下的形式：

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H \quad (13.1)$$

其中  $A$ 、 $B$ 、 $C$ 、 $D$ 、 $E$ 、 $F$ 、 $G$  和  $H$  仅是  $x$  和  $y$  的函数。

下面是线性二阶偏微分方程的例子：

$$4u_x x + 6xyu_{xy} = 0$$

$$\pi u_{xy} + x^2 u_{yy} = \sin(xy)$$

下面是非线性的二阶偏导数的例子:

$$u_{xx}^2 + u_{yy} = 0$$

$$uu_{xy} + \sin(xy)u_{yy} = x + y$$

第一个方程不是一个线性的二阶的偏微分方程因为  $u_{xx}$  项是平方项; 第二不是线性的二阶偏微分方程是因为有  $u_{xy}$  项与  $u$  的相乘项。

基于 (13.1) 等式中的  $A$ 、 $B$ 、 $C$  的不同值, 我们可以把线性二阶偏微分方程分为三类:

- 椭圆偏微分方程: 它们满足  $B^2 - AC < 0$ ;
- 抛物线偏微分方程: 它们满足  $B^2 - AC = 0$ ;
- 双曲线偏微分方程: 它们满足  $B^2 - AC > 0$ 。

每一类都有一个著名的代表式。

**泊松等式**  $u_{xx} + u_{yy} = f(x, y)$  是一个椭圆偏微分方程的例子。它来自于对电场, 磁场以及重力场内势能问题的研究, 热量/电量在均匀导体内稳态的分布情况的研究, 以及一些液体流动和转动问题。当  $f(x, y) = 0$  时, 泊松等式又称为拉普拉斯等式。

**热传递方程**  $ku_{xx} = u_t$  是抛物线偏微分方程的例子。热传递方程来自于对固体中热传递的研究。液体和气体的扩散的等式同热传递方程等式有一样的结果, 但是这些情况下它被称作扩散方程。

**波方程**  $c^2 u_{xx} = u_{tt}$  是双曲线偏微分方程的例子。波方程来自于对波传播以及弦和膜振荡现象的研究。

## 13.2.2 差分商

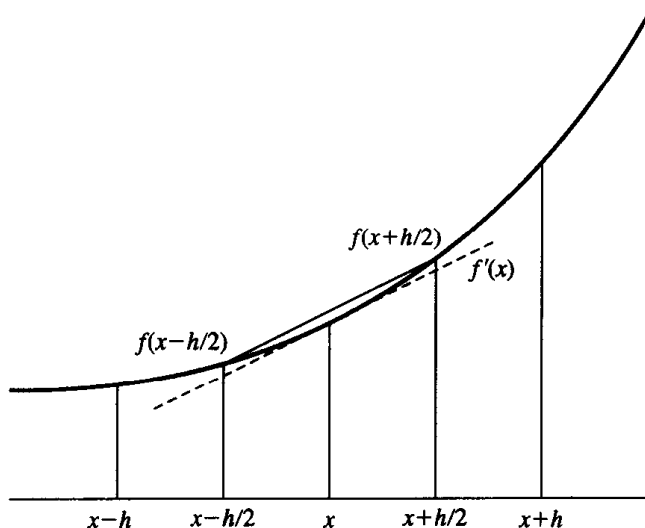
虽然我们用不同的算法来分别解决椭圆, 抛物线, 双曲线的偏微分方程, 但是所有的有限差分方法都是通过把变量 (经常是时间和空间) 分成很多离散的间隔来近似求解的。为了描述这个过程, 我们来考虑如何对一阶和二阶导数进行近似。

考虑图 13.2 中的函数  $f$  (我们假设  $f$  是一个连续的处处可导的函数)。我们要计算  $f$  在一个特定点  $x$  的一阶和二阶导数。对  $f'(x)$  的一个合理的近似为:

$$f'(x) \approx \frac{f(x+h/2) - f(x-h/2)}{h}$$

通过减小  $h$  我们能进一步减少近似值的误差。我们使用同样的公式来近似  $f''(x) = f'(f'(x))$ :

$$\begin{aligned} f''(x) &\approx \frac{\frac{f(x+h/2-h/2) - f(x+h/2-h/2)}{h} - \frac{f(x-h/2+h/2) - f(x-h/2-h/2)}{h}}{h} \\ &\approx \frac{f(x+h) - f(x) - (f(x) - f(x-h))}{h^2} \\ &\approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \end{aligned}$$

图 13.2 对在  $x$  点函数  $f$  的导数进行近似

## 13.3 弦振荡问题

作为我们学习有限差分方法的第一个例子，我们来考虑一个双曲线偏微分方程的示例。这一节只是简明地介绍一下这个算法发展的概况；如果需要，请查阅 Plybon【92】以获取更多的信息。

### 13.3.1 导出方程

观察图 13.3。我们的目标是对振荡弦进行建模（例如吉他弦）。更进一步说，我们试图基于弦当前的位置，确定某一个特定时刻时弦的位置。

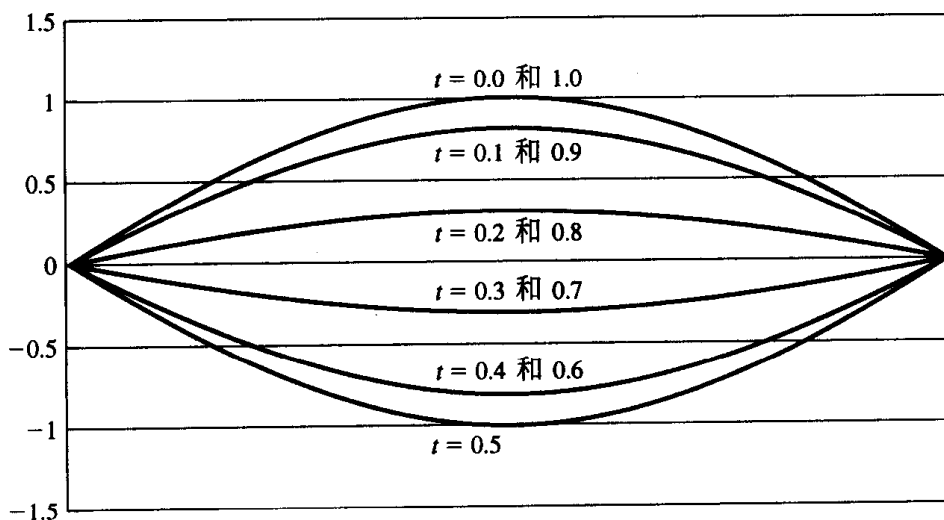


图 13.3 弦随时间的变化所做的运动

弦的端点是固定的。我们用变量  $x$  代表两个端点连线上的位置。左端点是  $x=0$  的点，右端点对应  $x=1$ 。从而  $0 \leq x \leq 1$ 。



我们用变量  $t$  表示时间。弦的初始位置是在时间为 0 时它的位置, 所以  $t \geq 0$ 。

函数  $u(x, t)$  描述弦的点  $x$  在时间  $t$  的偏移。

我们使用几个方程来对这个问题进行建模。第一个方程是一个线性二阶偏微分方程, 它表示位移随时间的变化:

$$4u_{xx} = u_{tt} \quad 0 < x < 1, \quad 0 < t$$

第二个方程表示弦的端点被固定的约束条件:

$$u(0, t) = u(1, t) = 0 \quad \text{其中 } t > 0$$

第三个和第四个方程分别描述了弦初始时的位置和状态信息:

$$u(x, 0) = \sin(\pi x), \quad u_t(x, 0) = 0 \quad \text{其中 } 0 \leq x \leq 1$$

如果对我们的问题进行归类, 那么它是一个波方程的实例, 下面是波方程的一般形式:

$$c^2 u_{xx} = u_{tt} \quad 0 \leq x \leq a, t \geq 0$$

$$u(x, 0) = F(x) \quad \text{和} \quad u_t(x, 0) = G(x) \quad \text{在 } [0, a] \text{ 上}$$

$$u(0, t) = u(a, t) = 0$$

一般地, 我们需要找到  $x$  在  $0 \sim a$  之间, 时间在  $0 \sim T$  之间的所有解。我们把空间划分为  $n$  个部分, 时间划分为  $m$  个部分, 我们定义  $h = a/n$ ,  $k = T/m$ 。也就是说,  $k$  是时间的步数而  $h$  是空间的步数。如果时间的步数  $k$  太大, 我们的离散化将会变得不准确, 从而算法会变得不稳定(例如, 我们的近似解和实际解之间的差别会随着时间步数增多而变得越来越大)。当系数  $kc/h > 1$  时会出现这样的情况。另一方面, 如果时间步太小, 舍弃的误差会不断积累, 这种情况也会使估计的精度降低。这种情况在系数满足  $kc/h < 1$  时会出现。所以, 最好的精度是在  $kc/h = 1$  时获得的。

在我们确保  $kc/h \leq 1$  后, 我们进一步定义:

$$x_i = ih \quad i = 0, 1, \dots, n$$

$$t_j = jk \quad j = 0, 1, \dots, m$$

现在我们可以定义  $u_{i,j} = u(x_i, t_j)$  为弦在位置  $x_i$  和时间  $t_j$  时的位移, 如图 13.4 所示。

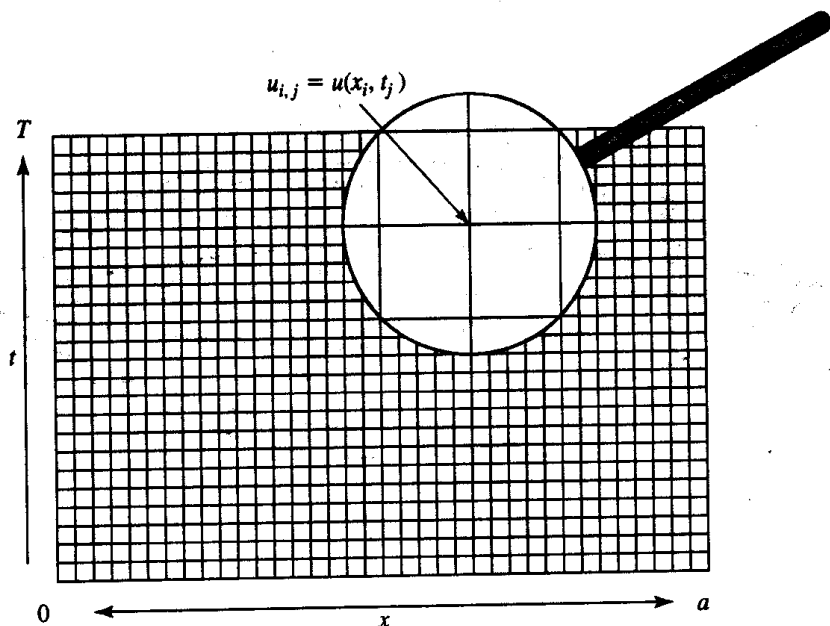


图 13.4 通过将空间和时间分成离散的间隔我们得到了矩形网格。边界条件则位于矩形的底边和两条垂直边。每个交点  $u_{i,j}$  表示  $u$  对  $x_i$  和  $t_j$  的近似。换句话说, 就是弦在某个时刻下某点的位移

### 13.3.2 串行程序

使用我们在 14.2 推导的公式, 我们能够得到对二阶偏导数  $u_{xx}$  的近似:

$$\begin{aligned} u_{xx}(x_i, t_j) &\approx \frac{u(x_i + h, t_j) - 2u(x_i, t_j) + u(x_i - h, t_j))}{h^2} \\ &\approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \end{aligned}$$

类似地, 我们能够得到近似的二阶偏导数  $u_{tt}$ :

$$\begin{aligned} u_{tt}(x_i, t_j) &\approx \frac{u(x_i, t_j + k) - 2u(x_i, t_j) + u(x_i, t_j - k))}{k^2} \\ &\approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} \end{aligned}$$

这些近似方法能够通过替换返回到波方程。经过一系列的进一步的近似和细化 (请参阅 Plyon 【92】以获取更详细的信息), 我们可以得到图 13.5 所示的 C 程序。注意, 在把图 13.4 转变成图 13.5 的矩阵  $u$  的时候, 对下标进行了转置。

```
/* Sequential solution to string vibration problem */
#include <stdio.h>
#include <math.h>
#define F(x) sin(3.14159*(x)) /* Initial string position */
#define G(x) 0.0 /* Initial string velocity */
#define a 1.0 /* Length of string */
#define c 2.0 /* String-related constant */
#define m 20 /* Discrete time intervals */
#define n 8 /* Discrete space intervals */
#define T 1.0 /* End time of simulation */
int main (int argc, char *argv[])
{
    double h; /* Space interval length */
    int i, j;
    double k; /* Time interval length */
    double L; /* Computed coefficient */
    double u[m+1][n+1]; /* String displacements */
    h = a / n;
    k = T / m;
    L = (k*c/h)*(k*c/h);
    for (j = 0; j <= m; j++) u[j][0] = u[j][n] = 0;
    for (i = 1; i < n; i++) u[0][i] = F(i*h);
    for (i = 1; i < n; i++)
        u[1][i] = (L/2.0)*(u[0][i+1] + u[0][i-1]) +
            (1.0 - L) * u[0][i] + k * G(i*h);
```

```

for (j = 1; j < m; j++)
    for (i = 1; i < n; i++)
        u[j+1][i] = 2.0*(1.0-L)*u[j][i] +
            L*(u[j][i+1] + u[j][i-1]) - u[j-1][i];
for (j = 0; j <= m; j++) {
    for (i = 0; i <= n; i++) printf ("%6.3f", u[j][i]);
    putchar ('\n');
}
return 0;
}

```

图 13.5 实现解决弦振荡问题有限差分算法的 C 程序 (由 Plybon【92】的伪码所改编)

### 13.3.3 并程序序设计

按照惯例, 我们首先设计原始任务, 确认它们之间的数据通信模式, 然后寻找将这些任务聚集起来的方法。最精细粒度下的基本任务是计算矩阵  $u$  中每个元素的值。参看图 13.5 中的 C 代码, 我们发现  $u[j+1][i]$  的值依赖于  $u[j][i-1]$ 、 $u[j][i]$ 、 $u[j][i+1]$  以及  $u[j-1][i]$  的值。对于一个具体的  $u[j+1][i]$ , 其依赖关系如图 13.6 所示。如果我们画出完整的任务/通道图, 它将会看起来类似于图 3.11(a) (尽管不完全相同)。

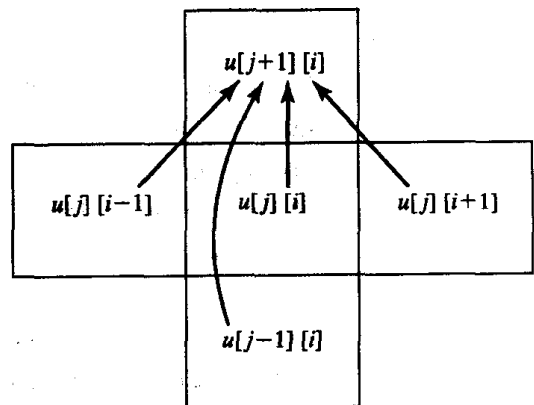


图 13.6  $u[j+1][i]$  的值依赖于  $u[j][i-1]$ 、 $u[j][i]$ 、 $u[j][i+1]$  和  $u[j-1][i]$  的值

对一个特定点在各个时刻的位移的计算, 其本质上是串行的:  $u[j+1][i]$  的值有赖于  $u[j][i]$  和  $u[j-1][i]$ 。由于这个原因, 我们需要把所有关于  $x_i$  的任务聚集起来。在这个点上, 所有的通信都是集中在相邻的任务间。如果我们进一步对弦上连续区域的任务进行聚集, 任务之间的通信将被最小化。

假设弦上的  $n+1$  个位置被分配到了  $p$  个进程中。我们来考虑在计算矩阵的第  $j+1$  列时发生的通信, 见图 13.7(a), 进程  $q$  负责计算 4 个不同  $i$  所对应的  $u[j+1][i]$ 。它无需任何通信就能够计算灰色格子中的值。然而, 它在从临近进程中得到数值后才能计算黑色格子的值。在图 13.7(b) 中, 我们说明了进程  $q$  如何同进程  $q-1$  和进程  $q+1$  交换数据。接收到这些值后, 黑色的格子才能被赋值。

如果同一循环同时更新边缘上的格子和内部格子的值, 那么并行程序的编写将会变得更加简单。给进程  $q$  额外分配两个列就可以做到这一点。这些列用于将接收邻近进程的数据 (每次迭代对应一列元素)。幻影点 (Ghost Point) 用来存放邻近进程数据的副本。图 13.7(c) 所示的是当  $n=16$  和  $p=4$  时, 若假设循环的两个边界之间无需通信, 我们应当如何设置幻影点。

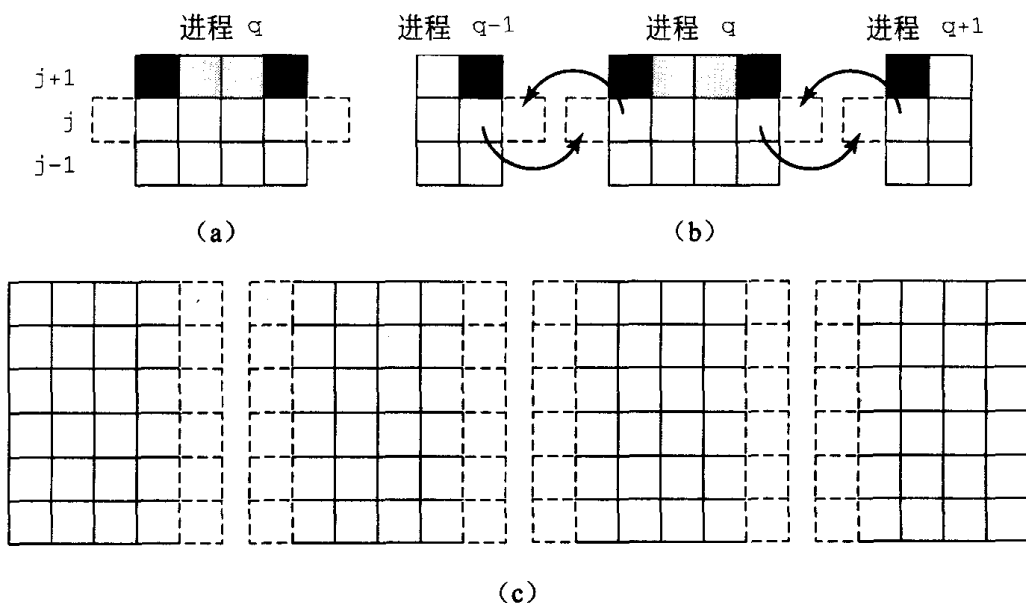


图 13.7 幻影点简化了有限差分并程序。 (a) 当计算第  $j+1$  行时, 进程  $q$  可以计算用于填入灰格子的数值, 但是它需要从邻近的进程里获取一些数值才能填充那些黑色的格子。

(b) 每个进程把它边缘上的值送到其邻居中。每个进程接收这些值并放入幻影点。这样幻影点内的就含有了有效数据的副本。(c) 通过将幻影点看做是数据向量中额外的行或列, 编程会变得更简单。在接收到这些值之后, 一个简单的 for 循环就可以更新  $j+1$  行中的每一个值

在进行第  $j+1$  行迭代运算的过程中, 每个进程将其边界上的各个数值发送给相应的邻居, 同时也接收邻居第  $j$  行的边界值。在接收到该值并存入幻影点后, 通过一个简单的 for 循环进程就可以计算其第  $j+1$  行的值了。

### 13.3.4 等效分析

计算每个元素值所消耗的计算时间是恒定的, 所以串行化算法每一次迭代的时间复杂度为  $\Theta(n)$ 。如果各个元素平均地分配在  $p$  个进程上, 那么并行算法每一次迭代的时间复杂度变为  $\Theta(n/p)$ 。

每一次迭代中, 一般而言进程必须发送消息给它的两个邻居, 每一个消息的程度为 1, 然后从它们的邻居进程处接收消息。这些发送接收消耗的通信时间复杂度为  $\Theta(1)$ 。从而整个并行算法的通信开销是  $\Theta(p)$ 。

该算法的等效关系是:

$$n \geq Cp$$

我们的方法中使用了大约  $nm$  个元素来存储弦上每点在每一时刻的位置, 然而由于模拟过程在时间上在不断推进, 我们也可以设计一种方案使之仅使用  $3n$  个内存单元。这时  $M(n) = n$ 。可扩展性的函数即为:

$$M(C_p)/p = C_p/p = C$$

可见, 我们所设计的算法具有非常优秀的可扩展性。

### 13.3.5 冗余计算

我们怎样做才能进一步减小通信开销呢？由于各个进程发送的消息只含有一个数据项，因而通信时间主要由消息延迟所占据。其实我们如果发送两个数据项，所消耗的通信时间基本一样。下面我们来挖掘在发送多个数据项时程序有何潜力。

观察图 13.8。其中图 13.8 (a) 部分描述了在最初的算法设计中将发生些什么。白色的方块代表内部的进程所负责的格子。虚线的白色方块是幻影点。当进程收到幻影点中的真实数据时，它就能计算出灰色方格中的值，也就是下一时刻它所负责部份的所有值。

图 13.8 (b) 部份描述了如果我们增加幻影点的数目至 2 个格子时我们能够做什么。每个进程现在发送两个值到与它相邻的进程。当它从它的两个邻居各收到一对值的时候，它能将整个模拟过程在时间上向前推进两个时间步。在第一个时间步内，它计算出它负责的所有格子内的值，以及两边的幻影点内的数据值（灰色的虚线方盒子）。后面一项属于冗余计算，因为这些幻影点的值同时也由它的邻居进程计算出来了。有了这些冗余的数据，进程能够进一步计算第二个时间步的值，无需传递任何消息。

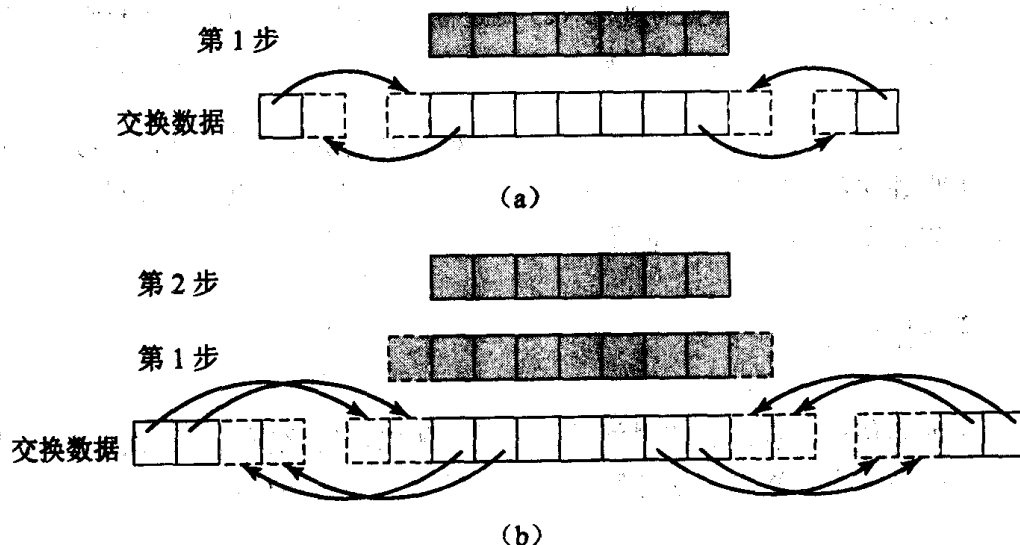


图 13.8 通过引入冗余计算我们能减少通信时间。(a) 传递边界上的一个元素能够使得算法在每次通信后只前进一个时间步。(b) 如果传递两个边界元素，那么算法能够在每次通信后在时间上前进两个时间步，这一过程引入了两次额外计算

增加幻影点的数目有三个影响：增加消息长度，减少消息频率和增加冗余计算。我们试图归纳每一次迭代的并行开销关于幻影点的数目的函数关系。如果每一个边界上有  $k$  个幻影点，那么邻居进程之间每  $k$  次迭代后会相互交换长度为  $k$  的消息。同时引入的冗余计算的次数是：

$$\sum_{i=1}^{k-1} i = k(k-1)/2$$

每个进程每次迭代的并行开销则为：

$$\frac{2(\lambda + k/\beta) + \chi k(k-1)/2}{k} = \frac{2\lambda}{k} + \frac{2}{\beta} + \frac{\chi(k-1)}{2}$$

可见，消息延迟与  $k$  成反比，然而计算时间随着  $k$  的增长平方地增加。所以在典型情况下，并行开销函数将有图 13.9 所描述的特性。使得该函数最小化的  $k$  的值依赖于  $\lambda$  和  $\chi$  的值。

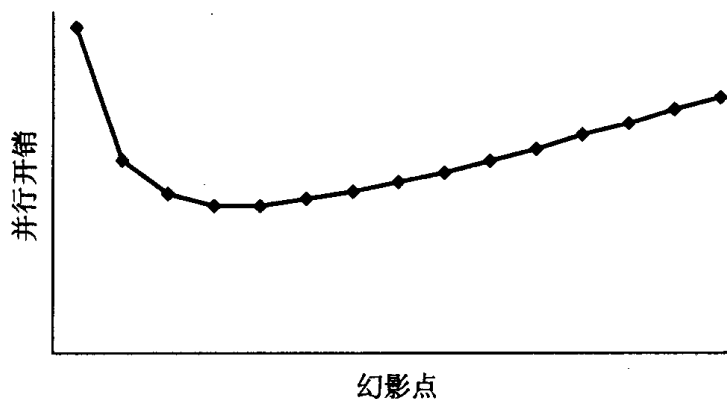


图 13.9 适当增加幻影点的个数可以使并行开销降到最小，之后如果增加幻影点的个数，冗余计算开销将超出所减少的通信开销

## 13.4 稳定状态热量分布问题

作为我们第二个研究的实例，我们考虑一个用来解决薄盘上稳态热分布问题程序的并行化（本节将遵循 Plybon【92】中的符号）。

### 13.4.1 方程的导出

这个问题对应的有限差分方程本质上是 Poisson 方程：

$$u_{xx} + u_{yy} = f(x, y), \quad 0 \leq x \leq a, \quad 0 \leq y \leq b$$

我们为这个问题加上边界条件：

$$u(x, 0) = G_1(x) \text{ 和 } u(x, b) = G_2(x) \quad 0 \leq x \leq a$$

$$u(0, y) = G_3(y) \text{ 和 } u(a, y) = G_4(y) \quad 0 \leq y \leq b$$

由于是矩形区域，所以我们也称它为狄利克问题。如果函数  $G_1$ 、 $G_2$ 、 $G_3$  和  $G_4$  在边界上连续而且函数  $f$  在矩形范围内连续，那么这个问题只有惟一解。

正如我们在前一个例子中学到的，我们将创建一个二维网格。然而，这里对网格的解释却不大一样。在振荡弦问题中，网格的每个格点  $(x_i, t_j)$  表示弦上的  $x_i$  点，在  $t_j$  时刻的位移。在这个例子中，每个格点  $(x_i, x_j)$  对应的数值表示矩形中该坐标点的稳态解。在上一个例子中，对每一个格点我们只计算了一次它的值。这个例子中，我们会反复更新内部格点的值直至它们收敛。

### 13.4.2 串行程序导出

我们把  $x$  维度分成  $n$  份, 而把  $y$  维度分成  $m$  份。我们定义  $h=x/n$  和  $k=y/m$ 。

使用 14.2 节中推导的二阶导数的近似结果, 我们知道:

$$\begin{aligned} u_{xx}(x_i, y_j) &\approx \frac{u(x_i + h, y_j) - 2u(x_i, y_j) + u(x_i - h, y_j)}{h^2} \\ &\approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \end{aligned}$$

类似地,

$$u_{yy}(x_i, y_j) \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2}$$

把这些近似值代入 Poisson 方程, 得到:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} = f(x_i, y_j)$$

假设  $\lambda=k/h$ 。在一系列进一步的近似操作后 (具体细节见 Plybon 【92】), 我们能够得到一个计算每一个格点值的新公式:

$$w_{i,j} = \frac{\lambda^2(w_{i+1,j} + w_{i-1,j}) + w_{i,j+1} + w_{i,j-1} - k^2 f_{i,j}}{2(1 + \lambda^2)}$$

现在看看我们要解决的这个具体问题。一个薄铁盘三边都被蒸汽所环绕 (摄氏 100 度), 第四边接触着冰块 (摄氏 0 度)。一个绝热的东西覆盖在盘子的顶部和底部。我们的目标是求出盘上各点的稳态温度分布。

既然各点均匀分布, 所以  $h=k$  并且  $\lambda=1$ 。由于盘面是绝热的, 所以除了边缘之外, 任何其他地方都不与它进行热量交换。这意味着  $f_{i,j}=0$ 。所以通过对这个线性二阶偏微分方程进行有限差分近似, 我们得到:

$$w_{i,j} = \frac{w_{i+1,j} + w_{i-1,j} + w_{i,j+1} + w_{i,j-1}}{4}$$

从所有的  $w_{i,j}$  的初值开始, 我们根据前一步的估计值通过迭代计算得到新的估计, 直到这些值收敛。根据第  $i$  次迭代的结果计算第  $i+1$  步迭代的方法叫做 Jacobi 方法 【56】。

(注意这个算法同我们在第 12 章遇到的算法一样。) 实现这个稳态热量分布问题求解的 C 程序见图 13.10 所示。

```
/* Sequential Solution to Steady-State Heat Problem */
#include <math.h>
#define N 100
#define EPSILON 0.01
int main (int argc, char *argv[])
{
    double diff; /* Change in value */
    int i, j;
```

```

double mean; /* Average boundary value */
double u[N][N]; /* Old values */
double w[N][N]; /* New values */
/* Set boundary values and compute mean boundary value */
mean = 0.0;
for (i = 0; i < N; i++) {
    u[i][0] = u[i][N-1] = u[0][i] = 100.0;
    u[N-1][i] = 0.0;
    mean += u[i][0] + u[i][N-1] + u[0][i] + u[N-1][i];
}
mean /= (4.0 * N);
/* Initialize interior values */
for (i = 1; i < N-1; i++)
    for (j = 1; j < N-1; j++) u[i][j] = mean;
/* Compute steady-state solution */
for (;;) {
    diff = 0.0;
    for (i = 1; i < N-1; i++)
        for (j = 1; j < N-1; j++) {
            w[i][j] = (u[i-1][j] + u[i+1][j] +
                       u[i][j-1] + u[i][j+1])/4.0;
            if (fabs(w[i][j] - u[i][j]) > diff)
                diff = fabs(w[i][j] - u[i][j]);
        }
    if (diff <= EPSILON) break;
    for (i = 1; i < N-1; i++)
        for (j = 1; j < N-1; j++) u[i][j] = w[i][j];
}
/* Print solution */
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
        printf("%6.2f", u[i][j]);
    putchar ('\n');
}
}

```

图 13.10 使用 Jacobi 迭代求解稳态热分布问题的 C 程序

### 13.4.3 并行程序设计

我们可以将对每个  $w[i,j]$  的计算看作是原始任务。在 Jacobi 方法中各个更新操作是完全并行的。为了计算  $w[i,j]$  的值，每个任务需要从北、南、东和西 4 个邻居处获得  $u$  的值，



如图 13.11 所示。

我们需要对原始任务进行聚集, 然后把聚集后的任务分配给各个并行进程。我们最好以哪种方式聚集各个原始任务呢? 如果每个进程负责一个矩形区域, 那么我们能够使用区域内  $u$  值直接计算矩形内部的  $w$  元素。计算矩形边缘上的  $w$  元素时需要其他进程的数据。

我们可以在进程所负责的区域边沿上引入幻影点。在幻影点接收到其他进程传来的数据后, 就可以通过一个单一的双重循环计算出所有  $w$  的值。

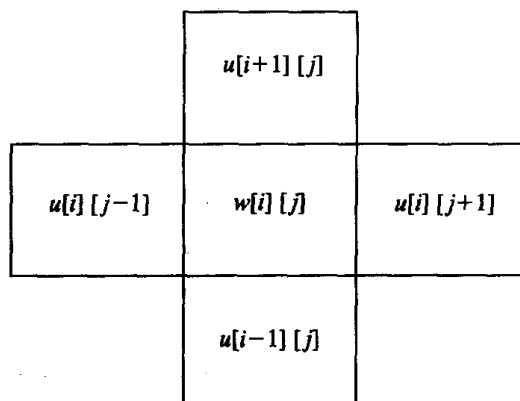


图 13.11  $w[i][j]$  的值依赖于  $u[i-1][j]$ 、 $u[i][j+1]$ 、 $u[i+1][j]$  和  $u[i][j-1]$  的值

我们可以选择按行分块 (见图 13.12 (a) 所示)。在这种分割模式下, 每个位于内部的进程与同它的相邻的两个进程交换数据。另外很明显我们也可以进行棋盘式分块。在这种情况下, 每一个内部的进程同其他四个相邻进程交换数据。

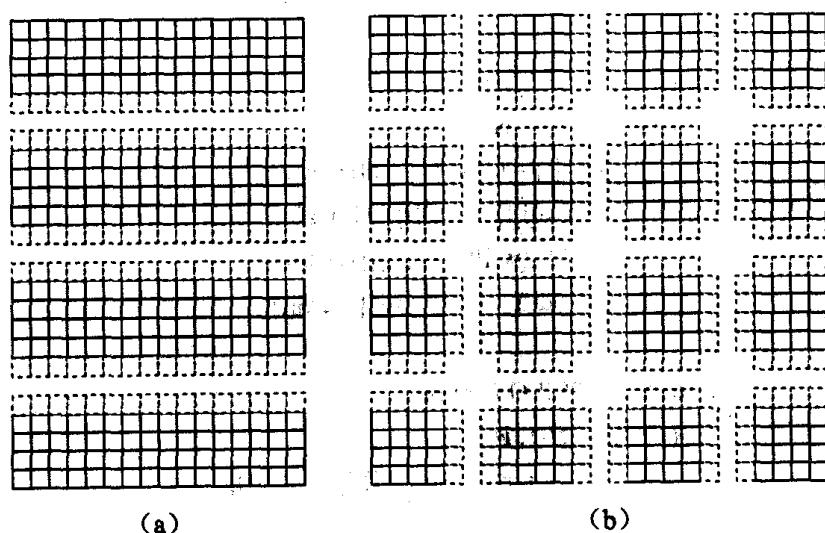


图 13.12 解决二维的稳态热量分布问题的几种可能的数据分割方式。(a)  $16 \times 16$  的网格经过按行分解映射到 4 个进程上。每一个进程负责包含  $(n/p) \times n$  个点的区域。虚线的格子为幻影点。(b) 使用棋盘式分割方式将  $16 \times 16$  的网格映射到 16 个进程上。每个进程管理一个大小为  $(n/\sqrt{p}) \times (n/\sqrt{p})$  的区域。虚线的格子为幻影点。

### 13.4.4 等效分析

假设我们的计算对象是一个大小为  $n \times n$  的网格。由于每一个网点的所消耗的时间是常数, 因而串行程序每一次迭代的计算复杂度是  $\Theta(n^2)$ 。

我们考虑按行分割。  $p$  个进程中, 每一个进程负责大小为  $(n/p) \times n$  的部分网格。在每次迭代中, 每一个内部进程必须分别向与之相邻的两个进程发送长度为  $n$  个值的消息, 并且从各个邻居处分别接收  $n$  个值, 所以通信复杂度为  $\Theta(n)$ 。而整个并行程序一次迭代的通信开销为  $\Theta(np)$ 。

基于按行分块的算法，其等效函数为：

$$n^2 \geq Cnp \Rightarrow n \geq Cp$$

由于  $M(n) = n^2$ ，所以可扩展性函数是：

$$M(Cp)/p = C^2 p^2 / p = C^2 p$$

所以它的可扩展性并不很好。

现在我们来考察棋盘分割方式。共有  $p$  个进程，每个进程管理一个大小约为  $(n/\sqrt{p}) \times (n/\sqrt{p})$  的区域。在每次迭代中，内部进程需向它的各个邻居发送  $n/\sqrt{p}$  个数，然后分别从它们那儿接收  $n/\sqrt{p}$  个值，因而通信的复杂度为  $\Theta(n/\sqrt{p})$ 。并程序每一次迭代的通信开销是  $\Theta(n\sqrt{p})$ 。

基于棋盘分割法的程序，其等效函数为：

$$n^2 \geq Cn\sqrt{p} \Rightarrow n \geq C\sqrt{p}$$

计算可扩展性函数：

$$M(C\sqrt{p})/p = C^2 p / p = C^2$$

可见，它具有高度的可扩展性。

### 13.4.5 实现细节

在二维分块的边界上放置幻影点意味着消息传递操作需要有额外的复制步骤。在 C 语言中，二维数组以行为主存放。顶行和底行的幻影点在内存位于连续的位置，但是列的幻影点则相反。由于这些元素在内存中的位置不连续，因而不能将包含幻影点消息的内容直接复制到幻影点中。你需要将接收到的消息存储在临时的缓存中，然后逐个复制到幻影点中。类似地，在向邻居进程发送它所需要的列幻影点时也需要一个临时的缓存来整合这些待发送的数据。

## 13.5 本章小结

偏微分方程是一个包含有两个或多个变量函数导数的等式。科学家和工程师广泛地使用偏微分方程来对物理系统进行建模。从现实问题归纳得到的偏微分方程都过于复杂，以至于我们得不到解析解。取而代之的是，科学家和工程师使用计算机通过数值方法来求解偏微分方程。

两个最常用的求解偏微分方程的数值方法是有限元法和有限差分两种方法。基于矩阵的有限差分法需要明确地表示矩阵，其中使用了可高效访问矩阵非零元素的数据结构。不基于矩阵的实现则不需要明确地表示出矩阵。这一章中我们设计并分析了不基于矩阵的有限差分算法的并程序实现。

线性二阶偏微分方程可分为椭圆型，抛物线型和双曲线型三种。不同类型的偏微分方

程有不同的算法,但是这些算法之间具有相似性。在我们的例子中,我们首先研究了波方程求解(抛物线偏微分方程的实例)和热方程的解(椭圆偏微分方程的实例)。典型的双曲线偏微分方程应用并不能很好的得到并行化。对于每一个具体例子,我们都运用了标准的并行算法设计方法。我们首先确认原始任务和它们间的通信模式。我们然后选择在最小化通信量和最大化利用率上进行折衷以确定如何进行聚集。

在本章的两个例子中,我们都使用了幻影点来存储我们从其他的进程接收到的值。当进程收到幻影点中的数据后,通过同一段代码可以对它负责的所有的格点进行更新。

我们也探索了如何增加幻影点以及发送额外的数值,从而通过增加冗余计算减少通信的频率。幻影点的最佳值取决于网络延迟,带宽以及计算单个格点的值所消耗的时间。

## 13.6 主要术语

Dirichlet problem	狄利克问题
Jacobi method	Jacobi 方法
ordinary differential equation	普通微分方程
diffusion equation	扩散方程
heat equation	热方程
partial differential equation	偏微分方程
finite difference method	有限差分方法
Laplace equation	拉普拉斯方程
Poisson equation	泊松方程
finite element method	有限元方法
linear second-order partial differential equation	线性二阶偏微分方程
wave equation	波方程
ghost points	幻影点

## 13.7 参考文献

由 G. D. Smith 编写的 *Numerical Solution of Partial Differential Equations: Finite Difference Methods* 一书详细论述了有限差分法【104】。Plybon 编写的 *An Introduction to Applied Numerical Analysis* 书中有一章讲述了利用有限差分法求解偏微分方程【92】。作为对这个领域的快速入门,你会发现它比 Smith 的专论容易懂得多。

M. J. Fagan 编写的 *Finite Element Analysis: Theory and Practice*【23】全面地描述了有限元法。

大多数有限元模型和某些有限差分模型使用的是非常规的网格。为了最小化通信量并平衡计算量,人们针对非常规网格的分解问题做了很多研究。Unstructured Scientific Computation on Scalable Multiprocessors 可以做为在并行计算机上解决非规则问题的一本入

门书【86】。

位于普渡的一个小组开发了 PELLPACK, 这是一个求解由偏微分方程描述的物体模型的环境。在 PELLPACK 的图形交互界面后面是多于 100 万行的代码。请查阅 Houstis et al.【53】以获得对这个问题求解环境的深入介绍。

## 13.8 练 习 题

13.1 函数  $u=f(x, y)$  有两个自变量, 它有两个一阶的偏导数  $u_x$  和  $u_y$ 。解释为什么  $u$  只有三个不同的二阶偏导数, 而不是 4 个 ( $2 \times 2$ )。

13.2 编写一个波方程求解串程序的并行版本。用不同的  $n$  和  $p$  的值测试你的程序。画出三维图以展示加速比对于  $n$  和  $p$  的函数关系。

13.3 本练习假设你在求解热方程时, 将矩阵分割为  $(n/\sqrt{p}) \times (n/\sqrt{p})$  的区域并分配到各个进程。设  $k = (n/\sqrt{p})$ 。每个进程上的  $u$  矩阵和  $w$  矩阵的大小均为  $(k+2) \times (k+2)$  (如图 13.13 所示)。 $u$  中额外的行和列提供了元素和幻影点的存储空间。 $w$  和  $u$  的大小相同, 这就允许我们在两个矩阵中使用同样的指标完成对有限差分网格中同一点的引用。

编写一段 C 代码, 用于进程间相互通信以更新它们邻居的幻影点。

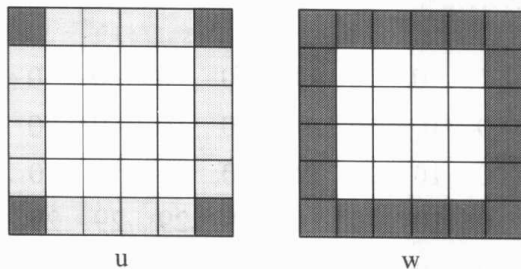


图 13.13 解决热方程的并程序序中, 一个进程负责网格中大小为  $4 \times 4$  的分块。

我们设  $u$  和  $w$  网格分别有 6 行和 6 列。其内部的白色方块包含了进程所负责的部分格点。

其额额外的行和列用来存储  $u$  的幻影点 (灰色)。矩阵  $u$  中的炭黑色的元素并未使用。

$w$  内部的白色方块用来保存新近计算得到的值; 炭黑色方块表示并未使用的空间

13.4 编写一个解决热方程串程序的并行版本。假设将矩阵按行分块后分配到各进程。用不同的变量值  $n$  和  $p$  测试你的程序。画出三维图以展示加速比对于  $n$  和  $p$  的函数关系。

13.5 编写解决热方程串程序的并行版本。假设对矩阵进行二维分块后分配到各个进程。用不同的变量值  $n$  和  $p$  测试你的程序。画出三维图以展示加速比对于  $n$  和  $p$  的函数关系。

13.6 分析在解决热方程的并程序序中利用冗余计算来减少通信开销的效率。假设每一个进程负责形为  $(n/\sqrt{p}) \times (n/\sqrt{p})$  的区域。

(a) 假设  $n$  是一个以  $\sqrt{p}$  为因子的整数。求出每一次迭代的平均通信开销, 通信开销将包括冗余计算所消耗的时间。你的答案应该是每次迭代的平均通信开销对于  $k, \lambda, \beta$  和  $\chi$  的函数。

(b) 假设  $n=5000$ ,  $p=16$ ,  $\chi=10$  纳秒,  $\lambda=100$  微秒,  $\beta=5 \times 10^6$  元素/秒。画出通信开销对于幻影带宽度  $k$  变化的函数关系, 其中  $1 \leq k \leq 10$ 。

13.7 写一个程序来解决组成标号问题。一个二值图像用由 0 和 1 组成的  $n \times n$  数组来表示。1 代表物体而 0 表示物体之间的空位。组成标号问题是将各个物体用不同的正整数进行标号。当程序结束时, 每一个值为 1 的像素点将会有有一个正的标号。如果一对像素点的值均为 1 并且它们属于同一个物体, 它们将有相同的标号。值为 1 的各个像素属于同一个物体, 当且仅当它们可以被一串值为 1 像素点相连。如果两个 1 像素点或者在水平方向上或者在垂直方向上彼此相邻, 那么就称它们是连续的。例如, 给定图像:

1	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
0	1	1	0	1	1	1	1
0	0	0	0	1	0	0	1
1	1	1	0	1	1	0	1
1	1	1	1	0	1	1	1
0	0	0	0	0	0	1	1

它的一种正确的 (当然不是惟一的) 输出:

1	0	0	0	0	0	0	0
0	10	0	10	0	0	0	0
0	10	10	10	0	0	0	0
0	10	10	0	29	29	29	29
0	0	0	0	29	0	0	29
41	41	41	0	29	29	0	29
41	41	41	41	0	29	29	29
0	0	0	0	0	0	29	29

注意在输入的图像为零的点在输出中值仍然是 0。如果输出图像的两个位置有相同的正值, 那么就意味着在输入图像中这两点间有一条 1 像素点的路径。

# 第 14 章 排 序

Had I been present at the creation, I would have given some useful hints for the better ordering of the universe.

**Reaction of Alfonso X to a description  
of the intricacies of the Ptolemaic system**

## 14.1 概 述

给定  $n$  个数  $\{a_0, a_1, a_2, \dots, a_{n-1}\}$ , 排序问题就是找到一个序列  $\{a'_0, a'_1, a'_2, \dots, a'_{n-1}\}$ , 使得  $a'_0 \leq a'_1 \leq a'_2 \leq \dots \leq a'_{n-1}$ 。排序是串行计算机上最常见的工作。许多算法都包含着排序功能以便使稍后的人们能更有效地进行访问信息。

用来进行排列的数字通常是数据集合的一部分, 我们称之为记录 (records)。在每个记录里, 用来进行排列的值称为键 (key)。记录里的其他数据叫做卫星数据 (satellite data)。即将访问的数据通常是卫星数据, 而被排序的是键值, 所以整个记录都要被重新排列。如果卫星数据很少, 排序的过程中整个记录都要被移动; 如果卫星数据很多, 实际排序的是指向记录的指针。本章的重点是如何对数列进行排序, 而将与卫星数据有关的问题作为实现的细节来处理。

研究者已经开发出了许多并行排序算法。然而, 这些算法的绝大多数是为理论化的并行计算模型或专用硬件设计的。对于在通用并行机上实现有效的排序, 这些算法的帮助都很小。

本章的重点在于设计适应于多 CPU 计算机的方法。我们需要从另两个方面限制我们讨论的范围: 首先, 我们考虑内排序 (internal sorts) —— 排序数据列足够小, 可以在主存中完成。相反, 外排序 (external sorts) 则对数据量太大不能一次装入主存的数列排序。其次, 我们这里考虑的排序是基于数据比较的。比如, 基数 (Radix) 排序就不是通过数据比较排序的。

本章概括了快速排序 (QuickSort) 的工作原理并开发出三种并行快速排序算法。假设目标机都是现代多计算机系统, 任何两个处理器间的时延和带宽都相同。

## 14.2 快 速 排 序

串行快速排序算法对大家来说已经非常熟悉了, 这里只做简单的回顾。更加深入的复习可以参考 Cormen 等【18】, Baase 和 Van Gelder【5】, 或其他算法分析教科书。

大约 40 年前 C.A.R.Hoare 发明了快速排序【51】, 它是一个依赖于键值比较从而对无序序列进行递归排序的算法。它从传入的数列中选一个作为中点。它把这列数分成两个子

列：“低列” (low list) 包含小于或等于中点的数，“高列” (high list) 包括大于中点的数。它递归调用自身对这两个子列排序 (如果一个子列没有数字，此次调用就被忽略)。函数返回关于由低列，中点和高列组成的有序序列。

例如，图 14.1 是对 {79, 17, 14, 65, 89, 4, 95, 22, 63, 11} 进行快速排序的过程。选择列表第一个元素作为中点。以 79 作为中点，低列为 {17, 14, 65, 4, 22, 63, 11}，高列是 {89, 95}。然后递归调用自身处理这两个子列。

处理低列 {17, 14, 65, 4, 22, 63, 11} 的递归执行先把 17 作为中点删除，又得到一个低列 {14, 4, 11} 和高列 {22, 63, 65}。同样，再次递归调用自身处理新的子列。

由于中点元素的删除保证数列的长度是不断减小的，所以函数一定会结束。如果某个子列没有元素，就没有必要对其排序。如果只有一个元素进行快速排序，这个元素就是中点，函数可以简单的返回这个值作为排序后的结果。

每个函数调用都返回排序后的两个子列和中点组成的新序列。例如，节点  $Q(17, 14, 65, 4, 22, 63, 11)$ 。调用  $Q(14, 4, 11)$  返回 {4, 11, 14}，中点值为 17，调用  $Q(65, 22, 63)$  返回 {22, 63, 65}。链接这些列，函数返回 {4, 11, 14, 17, 22, 63, 65}。

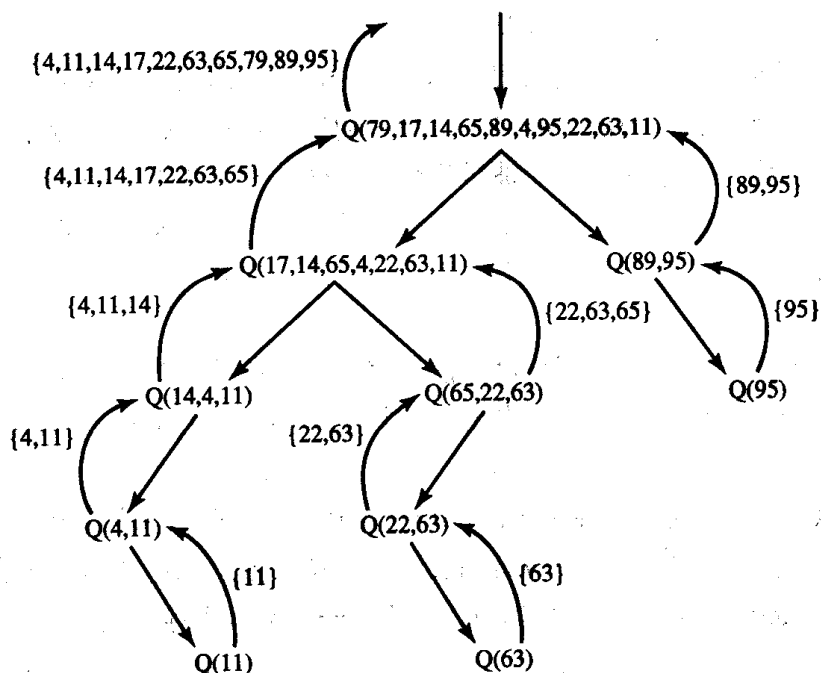


图 14.1 快速排序 10 个元素的列表。Q 代表对快速排序的调用。此算法移除列表的第一个元素，用它来做中点将列表分成两部分。再递归调用自身来处理两个子列。  
(如果列表为空则调用会被忽略)。函数返回由低列，中点，高列组合生成的新序列

### 14.3 并行快速排序算法

快速排序作为并行排序算法的起点有两个原因：首先，在一般情况下，快速排序是公认的基于键值比较排序中速度最快的算法。我们更愿意将并行算法建立在最快的串行算法之上。其次，快速排序天生具有并行性。当快速排序递归调用自身的时候，两个调用可以独立地执行。

### 14.3.1 排序完毕的定义

我们需要一种适用于商用集群和多机系统的算法。在深入探讨之前，必须对多计算机系统的排序作出明确的定义。我们可以这么说，算法开始前待排序列表在某个处理机的主存中，算法结束后排序过的列表还在同一处理机的主存中。这种定义的问题在于它不允许问题的最大规模随着处理器数目的增加而增大。

我们要为多计算机系统中并行排序给出一个不同的定义。我们假定待排序列表的值均匀的分布在各个处理器的主存中。算法结束后，(1) 驻留在每个处理器主存中的值都被排序好；(2)  $P_i$  列表的最后一个元素应不大于  $P_{i+1}$  列表的第一个元素，其中  $0 \leq i \leq p-2$ 。注意排序后的值并不一要求均匀地分布在处理器中。

### 14.3.2 算法开发

首先想象一下并行快速排序算法如何工作。由于快速排序算法会调用自身两次，在调用图中，“叶子”数为 2 的幂（忽略空列表而放弃的调用）。所以，我们假设活动进程数也是 2 的幂。

在图 14.2 中，待排序地数列分布在各个进程的内存中。从一个进程选择一个中点并广播它，如图 14.2 (a) 所示。每个进程把待排序的数列分成两个子列：小于等于中点的和大于中点的。进程列表中上半部分进程将自己的“低列”发送给下半部分的合作者进程，并从那里接收一个“高列”，如图 14.2 (b) 所示。现在进程列表中上半部分进程的值都大于中点，下半部分进程的值小于或等于中点，如图 14.2 (c) 所示。

然后进程分为两组，算法递归执行。每组进程选择一个中点并广播，如图 14.2 (c) 所示。进程分解自己的列表并与其合作进程交换，如图 14.2 (d) 所示。

经过  $\log p$  次递归后，每个进程所有的待排序值都与其他进程的待排序值完全脱离。也就是说，进程  $i$  的最大值小于进程  $i+1$  的最小值。每个进程可用顺序快速排序（或者其他方法）对自己部分的数据排序，之后并行快速排序结束。

### 14.3.3 分析

如果来实现这个算法，它的工作情况如何呢？执行时间随着第一个程序的启动而开始，随着最后一个程序的结束而停止。所以，为了让所有进程几乎同时结束，让它们拥有相同的工作量就非常重要。在本算法中，工作量与进程所控制的元素数目有关系。

不幸的是此算法在平衡列表大小方面表现很糟糕。例如，在图 14.1 的例子中。初始的列表分解产生了一个长度为 7 和一个长度为 2 的两个子列。如果中点值等于中值，列表就可被分成相等的两部分。但这样的话我们首先要费不少力气排序来确定中值。所以取中值做中点是不实际的。

如果选择列表的任意一个元素做中点，这个值接近列表中值的可能性就更大，自然平衡情况就会好一些。这个发现正是另一种并行快速排序算法：超级快速排序(hyperquicksort)



的起因。

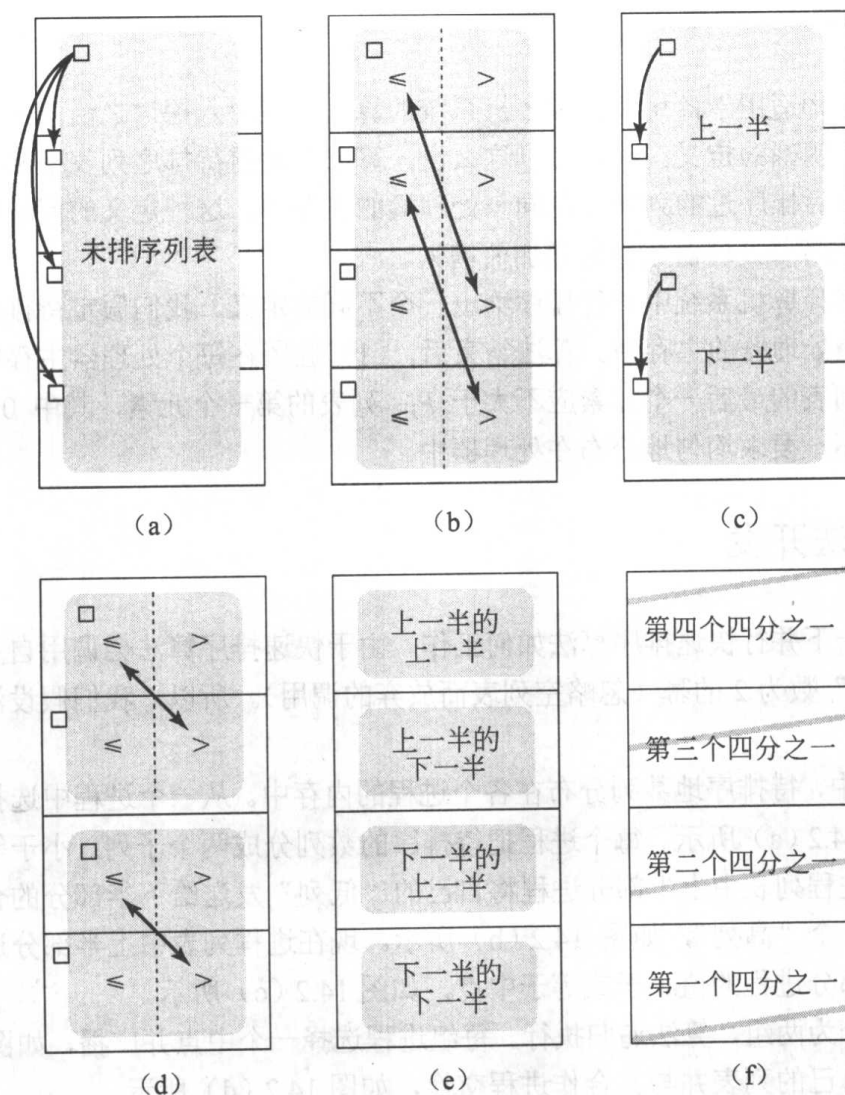


图 14.2 并行快速排序的高层视图。(a) 开始, 待排序值分布在所有进程的内存中。

选择某个值作为中点, 并广播到其他进程。(b) 进程利用中点分解成“下半部分”(lower half) 和“上半部分”(upper half)。上半部分进程与下半部分的合作者交换数据。

(c) 算法递归。从每半部分各选一个值做中点, 并广播到本部分的其他进程。

(d) 如同 (b), 进程利用中点进行分解。上半部分进程用较大的值与下半部分进程较小的值交换。(e) 此时, 进程  $i$  的最大值小于进程  $i+1$  的最小值。

(f) 所有进程用快速排序对自己控制的元素排序。列表排序完毕

## 14.4 超级快速排序

### 14.4.1 算法描述

Wagar 发明的超级快速排序【108】从每个进程用普通快速排序排序自己部分的列表开始, 也就是从第一种并行快速排序结束的时候开始。这个时候, 并行排序定义的第一点已

经满足,但不满足第二点。

为了满足第二点必须在进程间移动数据。像第一种算法一样,使用中点值将数据分为两组:下半部分(lower half)和上半部分(upper half)。因为每个进程里的列表已经排过序,负责提供中值的进程用自己列表的中值作为中点值。这个值比从列表中任选一个值作为中点更加接近实际列表的中值。

超级快速排序下面的三个步骤与前面已经开发出来的并行快速排序算法是一样的。选择中点的进程将中点值广播到所有进程。每个进程根据中值将自己的列表分成不大于中值的低列和大于中值的高列。上半部分的进程用自己的低列交换下半部分中其合作者的高列。

现在给超级快速排序增加一步。交换之后,每个进程都有自己本身的一个有序子列和一个从合作者接收来的有序子列。将这两个子列合并并得到一个有序列表。这一步结束后每个进程的列表必须有序,因为递归的时候,两个进程需要选择其中值作为中点值。

经过  $\log p$  次这样的分解-合并步骤,原始的  $p$  个进程的超立方体被分割成  $\log p$  个单进程超立方体,并且第二个条件已被满足。因为在分解-合并的过程中,每个进程反复的合并子列并保持子列的有序性,所以在算法最后不需要再次调用快速排序。图 14.3 是超级快速排序的示例。

超级快速排序假设进程数是 2 的幂。如果将进程组织成超立方体,就可以为超级快速排序建立通信方式,也就是只有邻接进程才能通信,如图 14.4 所示。正因为如此,超级快速排序非常适合将处理器组织成超立方体的第一代多计算机系统,如 Intel iPSC 和 nCUBE/ten。

## 14.4.2 等效分析

现在来分析超级快速排序的等效率性。假设  $p$  个进程对  $n$  个元素进行排序,其中  $n \gg p$ 。算法开始时,每个进程最多有  $(n/p)$  个元素。第一次快速排序算法的时间复杂度为  $\Theta[(n/p) \log(n/p)]$ 。假设每个进程有  $n/2p$  个值并且在分解-合并的步骤中传递  $n/2p$  个数据,将两个子列合并成一个有序序列所需要比较的次数为  $n/p$ 。分解-合并的操作分别在  $\log p$ ,  $(\log p)-1, \dots, 1$  维的超立方体上进行,分解-合并这个步骤所需要比较的次数为  $\Theta((n/p) \log p)$ 。整个算法需要比较的次数是  $\Theta[(n/p)(\log n + \log p)]$ 。

如果进程构成逻辑上的  $d$  维超立方体,广播分解值(即中点)所用的通信时间是  $\Theta(d)$ 。然而由于  $n \gg p$ ,与进程消耗在交换列表元素的时间相比广播的时间可以忽略。假设每个进程每次迭代都传递一半的数据,向合作者发送  $n/2p$  个数据和接收  $n/2p$  个数据耗时  $\Theta(n/p)$ 。共有  $\log p$  次迭代,从而分解-合并步骤的通信时间为  $\Theta(n \log p / p)$ 。由于最初的快速排序不需要进程间通信,这个值也就是超级快速排序算法所有的通信复杂度。

快速排序的时间复杂度是  $n \log n$ 。超级快速排序的通信开销是时间复杂度的  $p$  倍,也就是  $\Theta(n \log p)$ 。所以超级快速排序的等效率函数为:

$$n \log n \geq C n \log p \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C$$

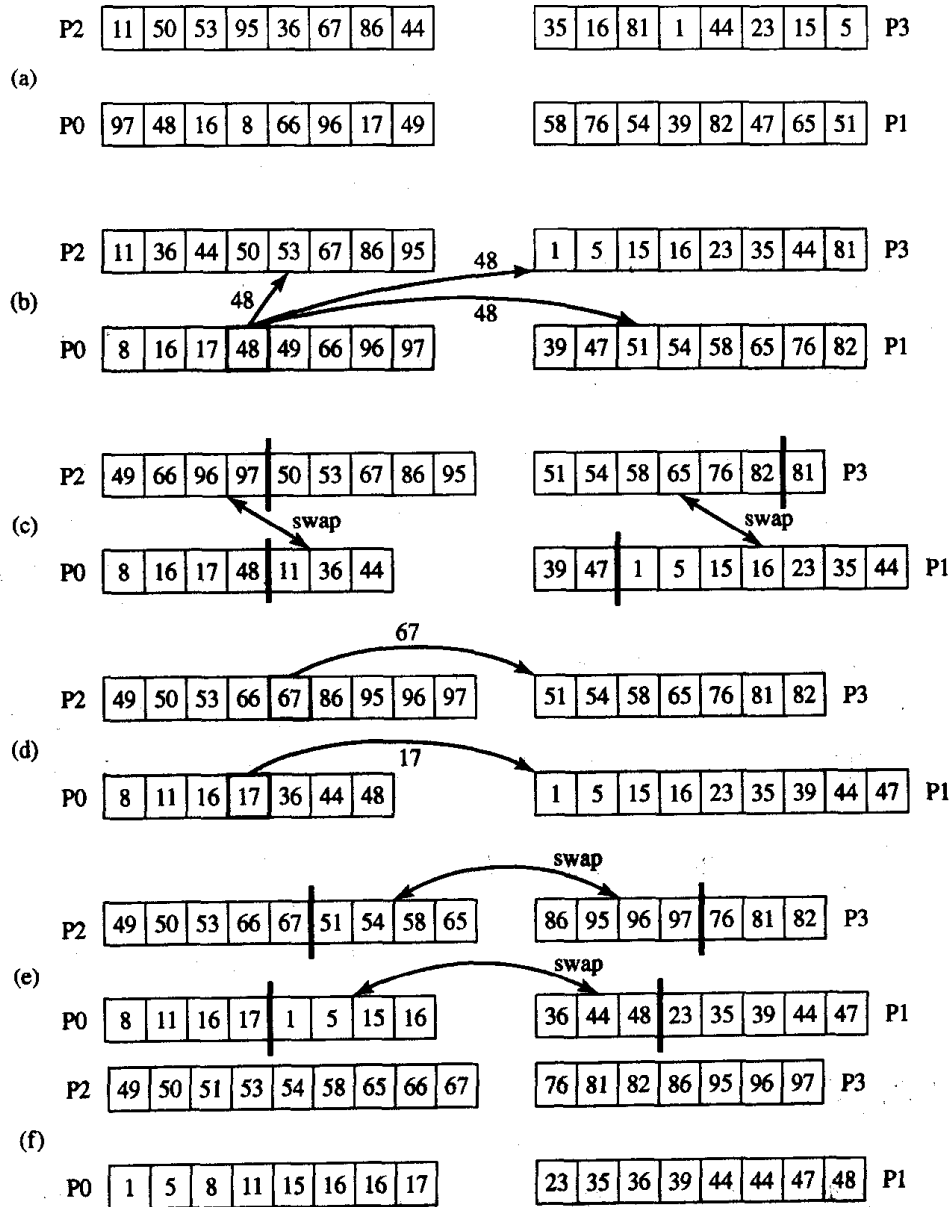


图 14.3 超级快速排序图示。4 个进程组成二维超立方体，对 32 个数字进行排序。

- (a) 初始状态，每个进程有 8 个数；(b) 每个进程用快速排序排序自己的列表。  
 进程 0 广播它的中值 48 到其他进程；(c) 超立方体下半部分的进程将大于 48 的值发送给上半部分的进程，上半部分的进程将小于或等于 48 的值发送到下半部分进程；  
 (d) 每个进程将自己原有的数和接收来的数合并。进程 0 广播其中值至进程 1，进程 2 广播其中值至进程 3；(e) 进程在超立方体的另一维上交换数据；  
 (f) 每个进程将自己原有的数和接收来的数合并。此时排序完毕

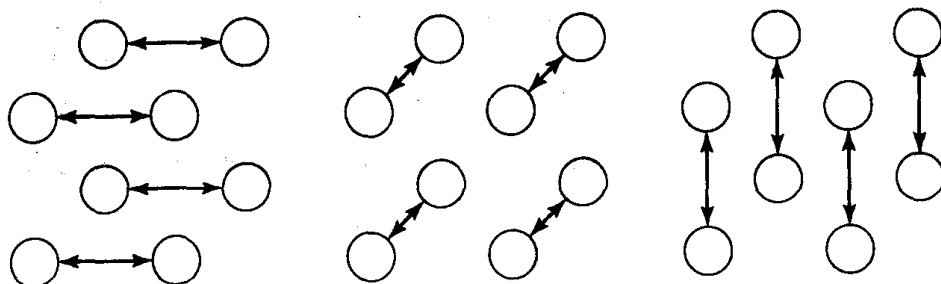


图 14.4 超级快速排序算法的通信模式。本例中有 8 个处理器，算法需进行 3 次分解-合并的步骤

这个问题对内存的消耗是线性的, 即  $M(n) = n$ 。所以超级快速排序的可扩展性函数是  $p^{C-1}$ 。 $C$  的值决定了并行系统的可扩展性。如果  $C > 2$ , 可扩展性就很低。

还有一个因素使得超级快速排序的可扩展性变得更糟。我们在分析中假设: 某个进程选择的中值就是实际的中值, 每个进程在迭代完毕分别向合作者发送并接收  $n/2p$  个元素。实际上, 中值元素并不是真正的中值, 进程间的负载是不平衡的。元素较多的进程在通信、归并上的耗时更多。这种不平衡随着处理器数目的增加而加剧, 因为每个处理器中的列表长度变小。如果列表较短, 从处理器中选择的中值有可能与实际中值差别更大。

简而言之, 超级快速排序的两个缺点限制了它的使用。首先, 一个值从一个进程传递到另一个进程的次数为  $(\log p)/2$  次。这个通信开销限制了此算法的可扩展性。我们可以通过寻找一种直接将数据发送到目的进程的方法来减小这个开销。其次, 选择分解值 (即中点) 的方法会导致进程负载的不平衡。如果可以从所有进程取样, 就能更好的使列表元素均匀分布。这两种想法都体现在第三种也是最后一种并行算法上: 规则取样并行排序 (Parallel sorting by regular sampling, PSRS)。

## 14.5 规则取样并行排序

由 Li 等发明的规则取样并行排序【74】与超级快速排序相比有三个优点: 保持进程中列表的大小平衡, 避免键值的重复通信, 不要求进程数为 2 的整幂。

### 14.5.1 算法描述

PSRS 算法共有 4 个阶段, 如图 14.5 所示。假设在  $p$  个进程上对  $n$  个数进行排序。第一阶段, 每个进程用串行快速排序对自己的元素进行排列 (不超过  $(n/p)$  个)。进程按照  $0, n/p^2, 2n/p^2, \dots, (p-1)(n/p^2)$  的顺序从自身有序的列表中本地规则取样。

第二阶段, 一个进程收集并排序这些本地规则样本。从有序的样本序列中选择  $p-1$  个中值, 序号为  $p+(p/2)-1, 2p+(p/2)-1, \dots, (p-1)p+(p/2)$ 。此时, 所有进程用这  $p-1$  个中值将其列表分解为  $p$  个片断。

第三阶段, 进程  $i$  保留第  $i$  个片断, 将第  $j$  个片断发送给进程  $j$  的第  $i$  部分,  $j \neq i$ 。

第四阶段, 所有进程合并其  $p$  个片断成一个列表。这个列表里的值与其他进程的数据表完全脱离。此步结束元素就排列完毕。

Li 等【74】已经证明 PSRS 算法在第四步进程合并的元素数小于  $2n/p$ , 也就是进程所拥有元素数的两倍。实际实验显示, 如果从一个统一的随机分布中选择数据, 片断的规模最大只会超过平均规模  $n/p$  百分之几。

### 14.5.2 等效分析

现在来分析 PSRS 算法的等效率性。假设  $p$  个进程对  $n$  个元素进行排序, 其中  $n \gg p$ 。

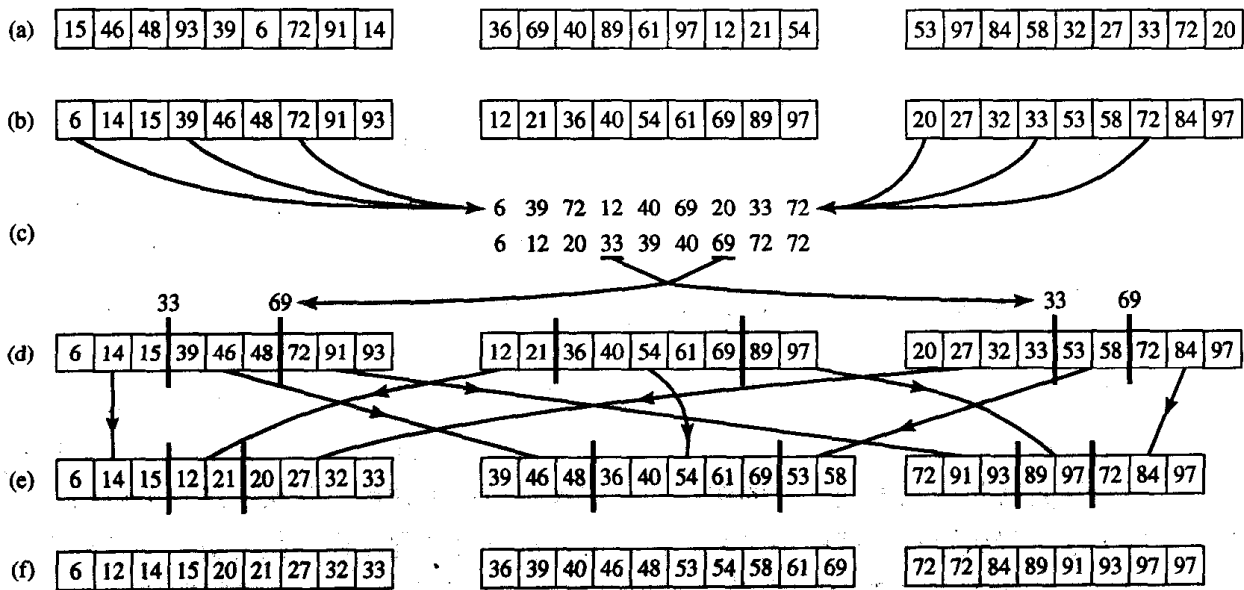


图 14.5 3 个进程用 PSRS 算法排序 27 个元素。(a) 初始的待排序的 27 个元素分解到 3 个进程;

(b) 各进程对自己的数据表进行快速排序; (c) 各进程从自己有序列表中规则取样。

某个进程收集这些样本, 将其排序, 然后把中值广播到其他进程; (d) 进程根据

(c) 中计算得到的中值将自己的列表分为 3 部分; (e) 所有进程进行一次

全交换通信将自己的有序子列发送到正确的进程去; (f) 各进程合并其有序子列。

第一阶段, 各进程完成  $n/p$  个元素的快速排序。这一步的时间复杂度是  $\Theta[(n/p) \log(n/p)]$ 。第一步结束后, 某个进程从其余  $p-1$  个进程各接收  $p$  个规则样本。相对而言, 这一步传递的数据很少, 消息延迟是主要因素。所以收集操作的通信复杂度为  $\Theta(\log p)$ 。

在 PSRS 算法的第二阶段, 收集数据的进程对  $p^2$  个元素排序。排序具有时间复杂度  $\Theta(p^2 \log p^2) = \Theta(p^2 \log p)$ 。排序进程广播  $p-1$  个中点到其他进程。因为只有  $p-1$  个值参与通信, 消息时延是主要因素, 所以通信复杂度为  $\Theta(\log p)$ 。

在第三步, 各进程依据中点值将自己的列表分成  $p$  部分。然后所有进程进行一次全交换。在这个全交换通信中, 每个进程发送和接收  $p-1$  个消息。假设列表大小平衡, 每个进程发送的元素数约为  $(p-1)n/p^2$ , 接近  $n/p$ 。由于  $n \gg p$ , 所以消息很长, 传递消息主要是数据的传输时间而不是延迟。所以执行全交换通信来完成进程发送接收  $p-1$  个消息是很必要的。这样的话, 列表元素就只需传递一次——直接到达需要它的进程。假设处理器互联网络支持  $p$  个消息同时传递。也就是说, 互联网络的容量随着处理器数目的增加而增加 (在第 2 章我们看到, 4 路胖树结构的等分带宽随着处理器数目的增加线性的增加)。在这种假设的前提下, 通信总的时间复杂度为  $\Theta(n/p)$ 。

在算法的第四步, 各进程对  $p$  个有序子列进行归并。假设列表大小是平衡的 (实验显示这个假设符合实际的), 归并所需的时间是  $\Theta[(n/p) \log p]$ 。

PSRS 算法总的计算时间复杂度为:

$$\Theta[(n/p) \log(n/p) + p^2 \log p + (n/p) \log p]$$

由于  $n \gg p$ , 对规则样本的排序时间可以忽略。第四阶段中归并阶段的系数比第一阶段中快速排序阶段的系数要高, 所以必须为第四步包含  $\Theta[(n/p) \log p]$  这一项。所以整体计

算复杂度是:

$$\Theta[(n/p)(\log n + \log p)]$$

假设并行系统的通信能力随着  $p$  线性增加, 总的通信复杂度为:

$$\Theta(\log p + n/p)$$

同样, 由于  $n \gg p$ , 通信时间主要为进程发送数据子列到其他进程所占据, 所以可以简化通信复杂度为:

$$\Theta(n/p)$$

系统的并行开销是通信复杂度的  $p$  倍, 即  $\Theta(n)$ 。再加上  $p$  倍的并行归并阶段的复杂度, 即  $\Theta(n \log p)$ 。PSRS 的等效率函数为:

$$n \log n \geq C n \log p \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C$$

由于  $M(n) = n$ , 可扩展性函数为:

$$p^C / p = p^{C-1}$$

这与超级快速排序的可扩展性函数是一样的。然而, 因为每个进程所拥有键的数目非常平衡, 所以 PSRS 算法可以获得比超级快速排序更好的加速效果。

## 14.6 本章小结

排序对常规计算机和并行计算机而言都非常重要。本章介绍了三种基于快速排序的并行算法, 它们适合于在多机系统和多处理器系统实现。

第一种算法反复的分解元素列表, 在进程对之间进行数据交换, 直到各个进程的子列不再重叠。不幸的是, 这种算法在平衡进程间数值方面的表现很糟糕。

超级快速排序保留了递归划分和子列交换的思想。但是它把快速排序从算法结束移到算法开始, 使得中点值的选择更加合理。超级快速排序算法的发明受到 20 世纪 80 年代多计算机系统处理器的超立方体结构的启发。在这些系统里, 发送消息所需的时间直接与发送进程和接受进程间的“跳步”(hops)成比例。超级快速排序可以实现所有的消息只在邻接处理器间传递, 可以优化超立方体的通信时间。因为超级快速排序依赖于一个单独的进程为整个立方体(或子立方体)选择中点, 随着处理器数目的增加, 中点的选取质量随之下降。当中点值逐渐偏离实际中值, 进程的负载就不再平衡, 降低了性能。

规则取样并行排序(PSRS)通过从所有进程的规则样本中选择中点的方法解决了超级快速排序负载不平衡的问题。还有一个优点是它的全交换通信可以让每个元素只移动一次(而不是  $\log p$  次)。这非常适合现在基于开关的集群, 在这种集群上任何一对处理器间发送消息时间基本相同。

## 14.7 主要术语

external sort	外排序
hyperquicksort	超级快速排序
internal sort	内排序
Key	键
parallel sorting by regular sampling	规则取样并行排序
Record	记录
satellite data	卫星数据
sorting problem	排序问题

## 14.8 参考文献

已经有许多研究工作研究并行排序算法。实际上已经有一本这方面的专著：Akl 著的 *Parallel Sorting Algorithm* 【1】。

1968 年, Batcher 给出了一种叫 “bitonic merge” 的并行算法【8】。这个由  $n/2$  个比较器组成的网络可以在  $\Theta(\log^2 n)$  的时间内对  $n$  个元素进行排序。然而基于 “bitonic merge” 的算法在多处理器计算机和多计算机系统上的可扩展性很差【44】。所以, 本章没有讨论它。

许多作者建议多处理器排序算法包含两个步骤: 第一步各进程完成自身列表的排序, 第二步进程互相协作将有序子列进行归并。相关文献可以参考 Francis 和 Mathieson【35】, Quinn【94】, Wheat 和 Evans【113】的著作。

Quinn【94】讨论了在 UMA 多处理器上的并行外层排序 (shell sort)。Fox【33】和 Grama【44】也介绍了一种并行外层排序算法。这个算法不是串行算法严格的并行化, 但具有同样的特征。

## 14.9 练习题

14.1 稳定排序 (stable sorting) 算法满足保持原始键值的顺序不变。快速排序是稳定排序么?

14.2 在有 16 个处理器的多计算机系统上进行排序, 键长为 4 字节。假设比较两个键需 70ns, 消息时延 200 $\mu$ s, 消息带宽 107 字节/秒。

(a) 进行以下的计算。采用第一种并行排序算法, 确定经过  $\log p$  次列表分解后任意进程中列表最大长度的期望值,  $1 \leq p \leq 16$ 。在每种情况下将期望最大长度与  $\lceil n/p \rceil$  比较。

(b) 预测第一种并行排序算法下分别在 1, 2, ..., 16 个处理器上完成 1 亿个键的排序可以达到的加速效果。

14.3 在有 16 个处理器的多计算机系统中进行排序，键长为 4 字节。假设比较两个键需 70ns，消息时延 200 $\mu$ s，消息带宽 10<sup>7</sup> b/s。

(a) 进行以下的计算。采用超级快速排序算法，确定经过  $\log p$  次列表分解后任意进程中列表最大长度的期望值， $1 \leq p \leq 16$ 。在每种情况下将期望最大长度与  $\lceil n/p \rceil$  进行比较。

(b) 预测超级快速排序算法下分别在 1, 2, ..., 16 个处理器上完成 1 亿个键的排序可以达到的加速效果。

14.4 一般情况下，完成  $n$  个键的排序顺序快速排序需要比较  $1.386n \log n \sim 2.846n$  次【5】。要在一多计算机系统中对键长为 4 字节的数据排序。假设比较两个键需 70ns，消息时延 200 $\mu$ s，消息带宽 10<sup>7</sup> b/s。PSRS 算法一般可以均匀地分解键值，所以最后一步每个进程最多合并  $\lceil 1.03n/p \rceil$  个键值。预测 PSRS 算法下分别在 1, 2, ..., 16 个处理器上完成 1 亿个键的排序可以达到的加速效果。

14.5 对于超级快速排序和 PSRS 算法的分析都是基于  $n$  个键值初始状态下随机分布的前提，即可能是  $n!$  组合中的任一种情况。假设列表的初始状态已经有序，即  $a_0 \leq a_1 \leq a_2 \leq \dots \leq a_{n-1}$ 。哪种算法受到的影响更小一些？为什么？

14.6 在 PSRS 算法的最后一步，每个进程必须合并  $p$  个有序子列，每个子列长度约为  $n/p^2$ 。（需要合并的元素的总数约为  $n/p$ ，最大不会超过  $2n/p$ ）。给出能在  $\Theta[(n/p) \log p]$  内完成合并的算法和数据结构。

14.7 (a) 实现超级快速排序并行算法。

(b) 根据你系统的  $\chi$ ,  $\lambda$  和  $\beta$  的值，预测在不同处理器数  $p$  和不同问题规模  $n$  下你的程序的加速情况。

(c) 在  $p$  和  $n$  相同的情况下测试你的程序。

(d) 预测时间和实验结果的误差有多大？确定你的公式中最大的误差源。

14.8 (a) 实现 PSRS 并行算法。

(b) 根据你系统的  $\chi$ ,  $\lambda$  和  $\beta$  的值，预测在不同处理器数  $p$  和不同问题规模  $n$  下你的程序的加速情况。

(c) 在  $p$  和  $n$  相同的情况下测试你的程序。

(d) 预测和实验结果的误差有多大？确定你的公式中最大的误差源。

14.9 本章主要集中讨论了快速排序的并行实现，其复杂度为  $\Theta(n \log n)$ 。归并排序 (Mergesort) 是另一个具有同样复杂度的有名的递归排序算法。下面是归并排序的伪码，它的核心函数是用于归并两个有序子列的函数 Merge。

```
Mergesort (list):
  if length (list) = 1 then
    return list
  else
    part1 ← Mergesort (first half of list)
    part2 ← Mergesort (remainder of list)
    return Merge (part1, part2)
  endif
```



如果需要归并排序更详细的信息请参看 Cormen【18】、Baase 和 Van Gelder【5】，或其他算法分析教科书。

(a) 设计一个适合多计算机系统的并行归并排序算法。假设，算法开始时待排序数据在位于某个处理器的内存中，在算法结束时有序列表仍在一个处理器的内存中，处理器数目  $p$  是 2 的幂。

(b) 求出算法的复杂度。

(c) 求算法的等效率函数。

(d) 编写程序实现你的并行算法。对  $p$  和  $n$  的不同组合进行测试。

14.10 假设  $n$  个键值均匀的分布在区间  $[0, k]$  内。桶排序 (Bucket sort) 将区间  $[0, k]$  均匀的分成  $j$  个部分，每个部分称为一个“桶” (bucket)。各键根据自己的值放入某个桶中。所有的键都放入桶中后，再对桶中元素排序。一旦桶中的元素都排序完毕，整个序列排序也就结束。

(a) 设计一个并行桶排序算法。开始  $n$  个键分布在  $p$  个进程中， $[0, k]$  分成  $p$  个桶。第一步，每个进程把自己  $n/p$  个键值分成  $p$  组，每个桶对应一个组。第二步，每个进程负责一个桶，执行一个全交换通信将各组都发送到正确的进程去。第三步，各进程对自己桶中的键值进行排序。

(b) 求出算法的时间复杂度。

(c) 求算法的等效率函数。

(d) 编写程序实现你的并行桶排序算法。对  $p$  和  $n$  的不同组合进行测试。

14.11 编写一个并行程序，从分布在  $p$  个进程的  $n$  个元素中找到第  $k$  大的值。

14.12 某文件包含  $n$  个长度为 4 字节的整数。编写一个并行程序判断哪个数在文件中出现的频率最高。

# 第 15 章 快速傅立叶变换

The meeting of two personalities is like the contact of of two chemical substances; if there is any reaction, both are transformed.

Carl Gustav Jung, *Modern Man in Search of a Soul*

## 15.1 概 述

离散傅立叶变换在科学研究和工程中有很多应用。例如，它经常被用于数字信号处理的应用，诸如语音识别和图像处理。离散傅立叶变换的直接实现其时间复杂度是  $\Theta(n^2)$ 。快速傅立叶变换是一个复杂度为  $\Theta(n \log n)$  的进行离散傅立叶变换的算法，而且并行化非常简单。

在这一章中我们通过一个来自语音识别的例子来说明离散傅立叶变换是如何工作的。我们将形式化地说明离散傅立叶变换及其逆变换。接下来我们给出快速傅立叶变化的算法并说明如何在多计算机系统上实现该算法。

## 15.2 傅立叶分析

傅立叶分析使用一系列正弦曲线 ( $\sin$  和  $\cos$ ) 函数表示一个连续函数。我们可以将离散傅立叶变换看成是一个函数，它将一个时间上的函数序列  $\{f(k)\}$  映射到频率上的函数序列  $\{F(j)\}$ 。序列  $\{f(k)\}$  表示一组信号采样的时间函数。序列  $\{F(j)\}$  表示傅立叶系数的分布，这个系数是关于频率的函数。我们可以使用  $\{F(j)\}$  来计算采样信号的各个分量。

图 15.1 展示了这一过程。我们从图 15.1 (a) 开始，图中给出了  $\{f(k)\}$  的图像，以及 16 个从时间 0 到时间  $2\pi$  的采样。图 15.1 (b) 是  $\{F(j)\}$  的图像，由 16 个点的复数序列所构成，它表示了其在频率上的分布。从  $\{F(j)\}$  的非零元素中我们可以获得产生信号的频率，其中“频率”是指在时间 0 和时间  $2\pi$  之间正弦波完成的完整周期的个数。非零的实部对应余弦函数，非零的虚部对应正弦函数。从图 15.1 (b) 中我们可以看出有频率为 2 和 5 的非零实部以及频率为 1 和 2 的非零虚部。因此产生信号的函数形为：

$$s_1 \sin x + c_2 \cos(2x) + s_2 \sin(2x) + c_5 \cos(5x)$$

对于每一个频率，我们将在图 15.1 (b) 左侧的幅值除以 8 (采样点个数 16 的一半) 来得到不同的正弦分量的系数。频率 1 的分量是  $16i$ 。16 除以 8 得到  $\sin x$  的系数为 2。频率 2 的分量是  $-8-16i$ 。-8 除以 8 得到  $\cos(2x)$  的系数为 -1，函数  $\sin(2x)$  的系数为 -2。我们使用同样的方法计算  $\cos(5x)$  的系数为 0.5。产生信号的 4 个分量为：

$$2\sin x - \cos(2x) - 2\sin(2x) + 0.5\cos(5x)$$

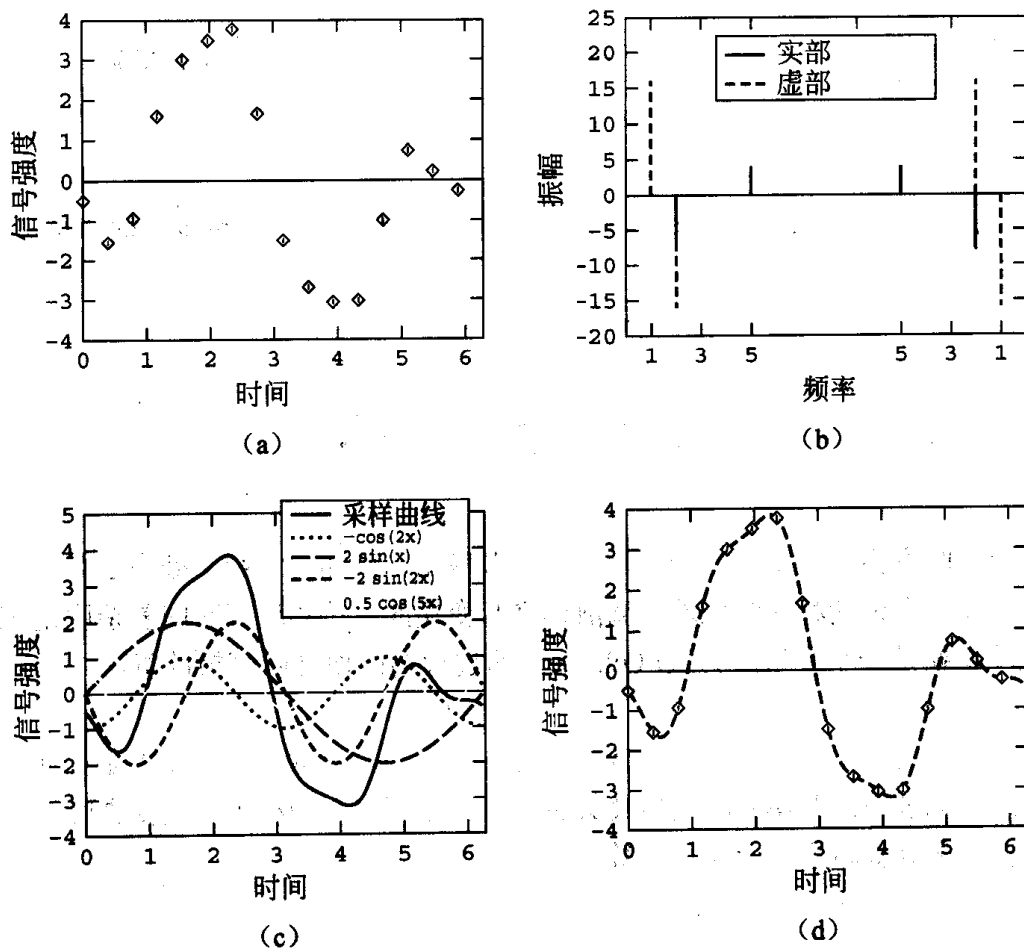


图 15.1 离散傅立叶变换举例。(a) 一组 16 个数据点, 是在时间区间 0 到不超过  $2\pi$  内的信号强度的采样; (b) 离散傅立叶变换的结果即各个正弦和余弦分量的频率和幅值; (c) 四个分量的图像和它们的和的图像; (d) 原连续函数的图像和原始的 16 个采样点

在图 15.1 (c) 中画出了不同频率分量的图像和它们的和函数, 这是一个连续函数。在图 15.1 (d) 中画出了原函数以及采样点。

现在我们来学习傅立叶分析是如何在语音识别中使用的。大多数语音分析是通过研究语音信号的频谱参数完成的。

将复杂的语音信号分解成周期性循环的频率分量是信号处理的中心工作, 并且这已被证明是可行的, 因为 (1) 正余弦是线性的物理 (电子) 系统的“自然语言”; (2) 共振现象是关系发声的最主要因素; (3) 语音声音是由语音基频及其和声组成的; (4) 人耳会进行某种形式的频谱分析。另外, 正弦信号 (以及一些其他的指数信号) 可以互不干扰的在线性系统中相加 (“叠加”); 这样我们可独立地对输入的信号按照频率分析分解成不同的正弦分量, 即 “正交信号”【67】。

离散傅立叶变换可以用于将数字化的人类语音采样信号转换成一张二维图像, 如图 15.2 所示。图中给出了检测到的信号频率关于时间的变化。每一个垂直方向的窄带以灰色阴影表示一个检测到的频率的幅值。当人们说话时, 语音信号发生变化, 信号的频率也变化。这样的频率图可以作为语音识别系统的输入, 识别系统将用模式识别的方式识别出发音的元素。

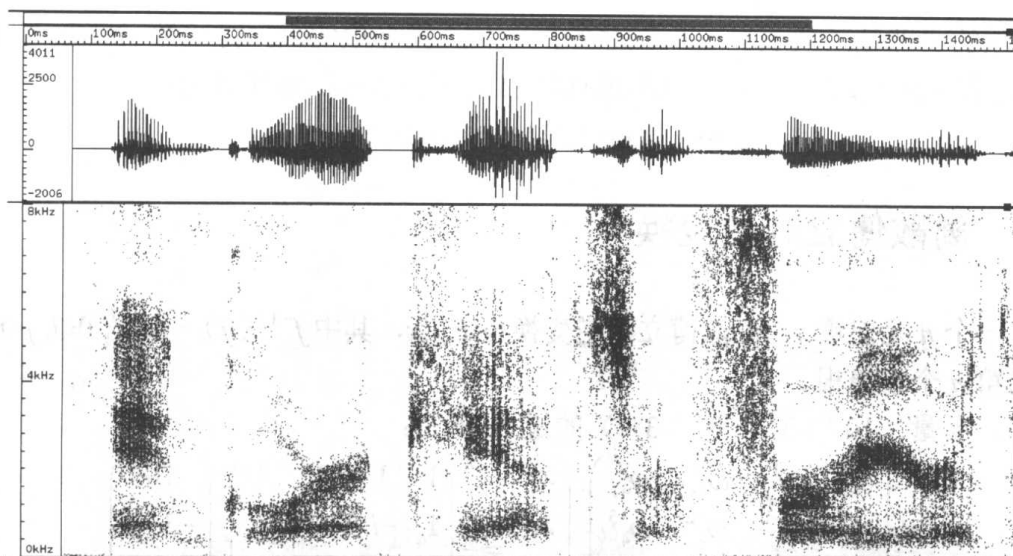


图 15.2 图的上半部分显示了输出信号强度随时间变化的情况，  
下半部分显示了频率和振幅随时间变化的情况

### 15.3 离散傅立叶变换

给定一个  $n$  元向量  $x$ ，离散傅立叶变换 (DFT) 是一个矩阵-向量的乘积  $F_n x$ ，其中  $f_{i,j} = \omega_n^{ij}$ ， $0 \leq i, j < n$ ， $\omega_n$  是单位元的第  $n$  个本原根 (有关复数的内容请参看附录 D)。

例如，计算向量  $(2, 3)$  的离散傅立叶变换，我们需要知道  $\omega_2$  单位元的本原平方根。单位元的本原平方根是  $-1$ 。则  $(2, 3)$  的离散傅立叶变换 (DFT) 结果是：

$$\begin{pmatrix} \omega_2^{0 \times 0} & \omega_2^{0 \times 1} \\ \omega_2^{1 \times 0} & \omega_2^{1 \times 1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 5 \\ -1 \end{pmatrix}$$

现在我们计算向量  $(1, 2, 4, 3)$  的 DFT。我们将使用单位元的本原四次方根，即  $i$ 。

$$\begin{pmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^1 & \omega_4^2 & \omega_4^3 \\ \omega_4^0 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ \omega_4^0 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 10 \\ -3-i \\ 0 \\ -3+i \end{pmatrix}$$

我们回到在前一节中提到的例子。有 16 个复数代表在时间区间 0 到  $2\pi$  的信号强度。为了简化表述，我们只显示每个数字的 3 位有效数字。

$$\begin{aligned} &(-0.500, -1.55, -0.939, 1.60, 3.00, 3.51, 3.77, 1.66, \\ &-1.50, -2.70, -3.06, -3.02, -1.00, 0.736, 0.232, -0.250) \end{aligned}$$

这个向量的 DFT 是：

$$(0, 16i, -8-16i, 0, 0, 4, 0, 0, 0, 0, 0, 4, 0, 0, -8+16i, -16i)$$

为了得到组成该信号的正余弦分量的系数，我们查看在变换后的序列中的前半非零元素 (从位置 9 到 15 的项只是从 1 到 7 的项的共轭，即虚部反向)。如果我们从 0 开始计

数, 第  $k$  项的实部是函数  $\cos(kx)$  的系数的 8 倍, 第  $k$  项的虚部是函数  $\sin(kx)$  的系数的 8 倍 (8 是采样点个数的一半)。则组成曲线的正弦和余弦函数组合为:

$$2\sin(x) - \cos(2x) - 2\sin(2x) + 0.5\cos(5x)$$

### 15.3.1 离散傅立叶逆变换

给定一个  $n$  元向量  $x$ , 离散傅立叶逆变换是  $F_n^{-1}x$ , 其中  $f^{-1}(ij) = \omega_n^{-ij}$ ,  $0 \leq i, j < n$ ,  $\omega_n$  是单位元的第  $n$  个本原根。

例如, 向量  $(10, -3-i, 0, -3+i)$  的逆 DFT 是:

$$\begin{aligned} \frac{1}{4} \begin{pmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^{-1} & \omega_4^{-2} & \omega_4^{-3} \\ \omega_4^0 & \omega_4^{-2} & \omega_4^{-4} & \omega_4^{-6} \\ \omega_4^0 & \omega_4^{-3} & \omega_4^{-6} & \omega_4^{-9} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} &= \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 10 \\ -3-i \\ 0 \\ -3+i \end{pmatrix} \\ &= \frac{1}{4} \begin{pmatrix} 4 \\ 8 \\ 16 \\ 12 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 4 \\ 3 \end{pmatrix} \end{aligned}$$

### 15.3.2 应用示例: 多项式乘法

我们使用 DFT 和逆 DFT 来进行多项式乘法。首先, 我们需要理解 DFT 和逆 DFT 做了些什么。DFT 计算了一个多项式在单位元的  $n$  次方根的  $n$  个复数值。让我们来看一下其中的原因。如果  $f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1x + a_0$  是一个  $n$  阶多项式,  $\omega$  是单位元的  $n$  次本原根, 则:

$$\begin{pmatrix} f(\omega^0) \\ f(\omega^1) \\ \dots \\ f(\omega^{n-1}) \end{pmatrix} = F \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix}$$

因为  $f(\omega^i) = a_0 + a_1\omega^i + a_2\omega^{2i} + \dots + a_{n-1}\omega^{(n-1)i}$ , 其中  $0 \leq i < n$ 。

逆 DFT 用一个多项式在  $n$  个复数单位元  $n$  次根上的值计算多项式的系数。

现在, 假设我们想相乘两个多项式:

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \text{ 和 } q(x) = \sum_{i=0}^{n-1} b_i x^i$$

这两个  $n-1$  阶的多项式的乘积是一个  $(2n-2)$  阶多项式:

$$p(x)q(x) = \sum_{i=0}^{2n-2} \sum_{j=0}^i a_j b_{i-j} x^i$$

我们可以通过计算原来的两个多项式的参数向量的卷积, 来得到结果多项式  $p(x)q(x)$

的系数。

例如，两个多项式相乘：

$$p(x) = 2x^3 - 4x^2 + 5x - 1$$

$$q(x) = x^3 + 2x^2 + 3x + 2$$

得到

$$r(x) = a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

我们计算参数向量的卷积：

$$a_6 = 2 \times 1 = 2$$

$$a_5 = 2 \times 2 + (-4) \times 1 = 0$$

$$a_4 = 2 \times 3 + (-4) \times 2 + 5 \times 1 = 3$$

$$a_3 = 2 \times 2 + (-4) \times 3 + 5 \times 2 + (-1) \times 1 = 1$$

$$a_2 = (-4) \times 2 + 5 \times 3 + (-1) \times 2 = 5$$

$$a_1 = 5 \times 2 + (-1) \times 3 = 7$$

$$a_0 = (-1) \times 2 = -2$$

结果为：

$$r(x) = 2x^6 + 3x^4 + x^3 + 5x^2 + 7x - 2$$

另一个相乘两个  $n-1$  阶多项式的方法是在单位元的  $n$  个  $n$  次方根计算它们的值，并在这些点按照元素相乘两个多项式的值，对结果作插值来得到多项式的系数。我们将这个方法应用于上面的例子。

首先我们在  $p(x)$  的系数上执行 DFT。按照从低到高的顺序列出系数。由于多项式是 3 阶的，所以后面的 4 个系数是 0。为了简化表述，我们只显示了小数点后两位数字。

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix} \begin{pmatrix} 1 \\ 5 \\ -4 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 1.12 + 0.95i \\ 3 + 3i \\ -3.12 + 8.95i \\ -12 \\ -3.12 - 8.95i \\ 3 - 3i \\ 1.12 - 0.95i \end{pmatrix}$$

然后我们对  $q(x)$  的系数做 DFT。这次也只显示了小数点后两位数字。

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & \omega^1 & \omega^4 & 1 & \omega^4 & \omega^1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 2 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 8 \\ 3.41 + 4.83i \\ 2i \\ 0.59 + 0.83i \\ 0 \\ 0.59 - 0.83i \\ -2i \\ 3.41 - 4.83i \end{pmatrix}$$

现在在这 8 个点上执行一个按元素的乘法:

$$\begin{pmatrix} 2 \\ 1.12+0.95i \\ 3+3i \\ -3.12+8.95i \\ -12 \\ -3.12-8.95i \\ 3-3i \\ 1.12-0.95i \end{pmatrix} \begin{pmatrix} 8 \\ 3.41+4.83i \\ 2i \\ 0.59+0.83i \\ 0 \\ 0.59-0.83i \\ -2i \\ 3.41-4.83i \end{pmatrix} = \begin{pmatrix} 16 \\ -0.76+8.66i \\ -6+6i \\ -9.25+2.66i \\ 0 \\ -9.25-2.66i \\ -6-6i \\ -0.76-8.66i \end{pmatrix}$$

最后在乘积结果向量上做逆 DFT。注意我们已经将有负指数的  $\omega$  替换成等值的正指数。例如, 当  $\omega$  是单位元的 8 次方根时,  $\omega^{-1}=\omega^7$ ,  $\omega^{-2}=\omega^6$ 。这里是逆 DFT:

$$\frac{1}{8} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \end{pmatrix} \begin{pmatrix} 16 \\ -0.76+8.66i \\ -6+6i \\ -9.25+2.66i \\ 0 \\ -9.25-2.66i \\ -6-6i \\ -0.76-8.66i \end{pmatrix} = \begin{pmatrix} -2 \\ 7 \\ 5 \\ 1 \\ 3 \\ 0 \\ 2 \\ 0 \end{pmatrix}$$

通过逆 DFT 得到的向量包含了从阶数由低到高的乘积多项式的各个系数。即,

$$r(x) = 2x^6 + 3x^4 + x^3 + 5x^2 + 7x - 2$$

## 15.4 快速傅立叶变换

在 15.3 节中我们展示了如何使用 DFT 和逆 DFT 完成两个多项式的相乘。可是我们为什么要使用这么复杂的算法来做卷积或者多项式相乘计算呢? 直接计算的可以在时间  $\Theta(n^2)$  内完成。原因在于我们不需要用矩阵向量乘法来执行 DFT 和逆 DFT。存在一种复杂度为  $\Theta(n \log n)$  的算法, 而且 (很幸运的是) 它易于并行化。这个改进的算法名为快速傅立叶变换 (FFT)。

FFT 使用“分而治之”的策略来计算一个  $n$  阶多项式的  $n$  阶 DFT 系数的值。定义  $n$  为 2 的整数次幂, 为了计算一个  $n$  阶多项式  $f(x)$ , 算法定义了两个新的  $n/2$  阶多项式。函数  $f^{[0]}(x)$  包含  $f(x)$  中的  $x$  的偶次幂项, 函数  $f^{[1]}(x)$  包含  $f(x)$  中的  $x$  的奇次幂项。

$$f^{[0]} = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1}$$

$$f^{[1]} = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}$$

注意到  $f(x) = f^{[0]}(x^2) + xf^{[1]}(x^2)$ , 因此在  $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$  点计算  $f(x)$  的值的问题转化为计算  $f^{[0]}$  和  $f^{[1]}$  在  $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$  点的问题, 然后计算  $f(x) =$

$f^{[0]}(x^2) + xf^{[1]}(x^2)$ 。

**等分引理** 如果  $n$  是一个正偶数，则单位元的  $n$  个  $n$  次方复根的平方分别等于单位元的  $n/2$  个  $n/2$  次方复根。

**证明** 参见附录 D。

根据等分引理，我们知道点集  $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$  只有  $n/2$  个不同的值。换句话说，为了计算多项式  $f(x)$  的  $n$  个  $n$  次方复根我们只需要计算多项式  $f^{[0]}(x)$  和  $f^{[1]}(x)$  在单位元的  $n/2$  个  $n/2$  次方复根。因此“分而治之”的策略可以辅助我们的计算。最自然的表达 FFT 算法的“分而治之”策略的是在图 15.3 中通过递归实现的伪代码。这个算法的时间复杂度很容易判断。令  $T(n)$  表示对  $n$  阶多项式使用 FFT 的时间，其中  $n$  是 2 的次幂。

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$$

参数 $n$	$a[0 \dots (n-1)]$	$a$ 的元素个数 系数
局部变量 $\omega_n$		单位元的 $n$ 次原根
$\omega$		当前点的估值多项式
$a^{[0]}$		偶数项系数
$a^{[1]}$		奇数项系数
$y$		变换结果
$y^{[0]}$		$a^{[0]}$ 的 FFT 结果
$y^{[1]}$		$a^{[1]}$ 的 FFT 结果

```

if  $n = 1$  then return  $a$ 
else
   $\omega_n \leftarrow e^{2\pi i/n}$ 
   $\omega \leftarrow 1$ 
   $a^{[0]} \leftarrow (a[0], a[2], \dots, a[n-2])$ 
   $a^{[1]} \leftarrow (a[1], a[3], \dots, a[n-1])$ 
   $y^{[0]} \leftarrow \text{Recursive\_FFT}(a^{[0]}, n/2)$ 
   $y^{[1]} \leftarrow \text{Recursive\_FFT}(a^{[1]}, n/2)$ 
  for  $k \leftarrow 0$  to  $n/2 - 1$  do
     $y[k] \leftarrow y^{[0]}[k] + \omega \times y^{[1]}[k]$ 
     $y[k + n/2] \leftarrow y^{[0]}[k] - \omega \times y^{[1]}[k]$ 
     $\omega \leftarrow \omega \times \omega_n$ 
  endfor
  return  $y$ 
endif

```

图 15.3 快速傅立叶变换算法的递归串行实现（资料来源于 Cormen et al. 【18】）

尽管递归形式的 FFT 算法是（相对）容易理解的，我们仍有两个原因使得我们开发一种迭代的 FFT 算法。首先，一个优秀的迭代版的 FFT 算法可以执行更少的下标运算并避免在每一次迭代的 for 循环中对  $\omega_n^k y^{[1]}(k)$  的双重计算。其次，迭代形式的串行算法更容易并行化。

图 15.4 给出了由递归算法演变的迭代算法。在图 15.4 (a) 中，我们看到一个递归算法如何对一个 4 元向量进行变换。每一个圆角矩形代表一个 `fft` 的函数调用。圆括号中是待变换的向量。函数不断将向量二分并对每一半递归调用直到向量长度为 1。单个数值的 DFT 是其本身（请记住 FFT 只是 DFT 的一个快速实现）。粗曲线箭头代表每一次函数调用的返回值。函数将收到的两个长度为  $i$  的向量组合起来并返回长度为  $2i$  的向量。对输入向量 (1, 2, 4, 3) 执行 FFT 得到结果 (10,  $-3-i$ , 0,  $-3+i$ )。



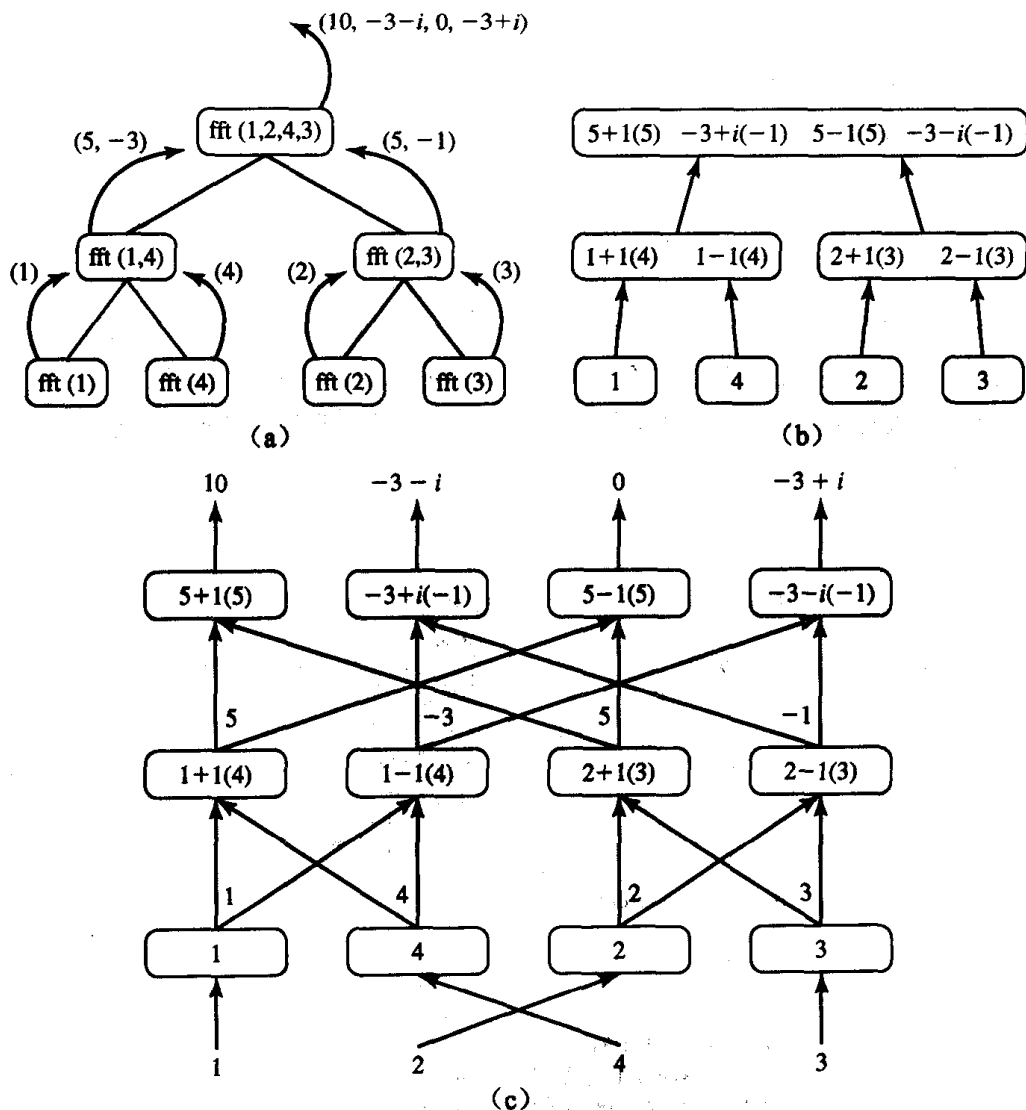


图 15.4 由递归算法到迭代算法的演变。(a) FFT 的递归实现。

(b) 决定每个函数调用中执行哪个计算。(c) 数据流的走向

在图 15.4(b) 中, 我们研究函数内部并决定每次调用执行哪些操作。具有格式  $a+b(c)$  和  $a-b(c)$  的表达式对应的伪代码语句为:

$$y[k] \leftarrow y^{[0]}[k] + \omega \times y^{[1]}[k]$$

$$y[k+n/2] \leftarrow y^{[0]}[k] - \omega \times y^{[1]}[k]$$

图 15.4(c) 显示了数据的流向。算法开始时各个元素按顺序排列。在输入向量中下标为  $i$  的元素被移到下标为  $\text{rev}(i)$  处,  $\text{rev}(i)$  表示将  $i$  的二进制位按倒序排列得到的值。例如, 数值 2 开始位于 01 下标处, 移到 10 下标处。数值 4 开始位于 10 下标处, 移到 01 下标处。数值 0 (在 00 处) 和数值 3 (在 11 处) 不动。在算法的第一阶段寻找单个值的 DFT, 仅仅是将原来的值传递过去。在剩下的步骤中, 每次计算一个新的值都依赖于前一步的两个值。数据流形成蝶形模式。

我们可以从图 15.4 中直接得到迭代算法。在初始排列之后, 算法将迭代  $\log n$  次。每次迭代对应图 15.4(c) 中的一个水平层次。在一次迭代中算法更新每一个下标的值。图 15.5 所示的算法达到和递归算法同样的时间复杂度:  $\Theta(n \log n)$ 。使用临时变量  $t$  将复数乘

法次数减少了将近一半。

```

Iterative_FFT(a,n):
  参数      n      a 的元素个数
            a[0... (n-1)] 系数
  局部变量   $\omega_d$     单位元的  $n$  次原根
             $\omega$       当前点的估值多项式
            y        变换结果

  y  $\leftarrow$  Bit_Reverse_Permutation(a)
  for j  $\leftarrow$  1 to log n
    d  $\leftarrow$  2j
     $\omega_d \leftarrow e^{2\pi i/d}$ 
     $\omega \leftarrow 1$ 
    for k  $\leftarrow$  0 to d/2 - 1
      for m  $\leftarrow$  k to n-1 step d
        t  $\leftarrow$   $\omega \times y[m + d/2]$ 
        x  $\leftarrow$  y[k]
        y[k]  $\leftarrow$  x + t
        y[k + d/2]  $\leftarrow$  x - t
      endfor
    endfor
     $\omega \leftarrow \omega \times \omega_d$ 
  endfor
  return y

```

图 15.5 快速傅立叶变换算法的串行迭代实现（资料来源于 Cormen et al. 【18】）

## 15.5 并行程序设计

### 15.5.1 分割与通信

迭代算法的效率是我们设计一个用于多计算机系统的并行 FFT 函数的出发点。我们将进行域分解, 让每一个任务负责输入向量  $a$  的一个元素和相对应的输出向量  $y$  的一个元素, 如图 15.6 所示。

算法的第一步是排列向量  $a$  的元素。每一个元素  $a[i]$  被复制到  $y[j]$ , 其中  $j$  是通过倒排  $i$  的位得到的。例如, 当  $n=8$ , 我们有:

```

001  $\rightarrow$  100
010  $\rightarrow$  010
011  $\rightarrow$  110
...
110  $\rightarrow$  011
111  $\rightarrow$  111

```

我们为此初始通信画出管道图。

函数的主循环有  $\log n$  次迭代。在每一次迭代中, 每一个任务根据上一步它的  $y[k]$  和  $y[k+m/2]$  或  $y[k-m/2]$  中的一个计算其  $y[k]$  的新值。任务/管道图显示出其蝶形通信模式。

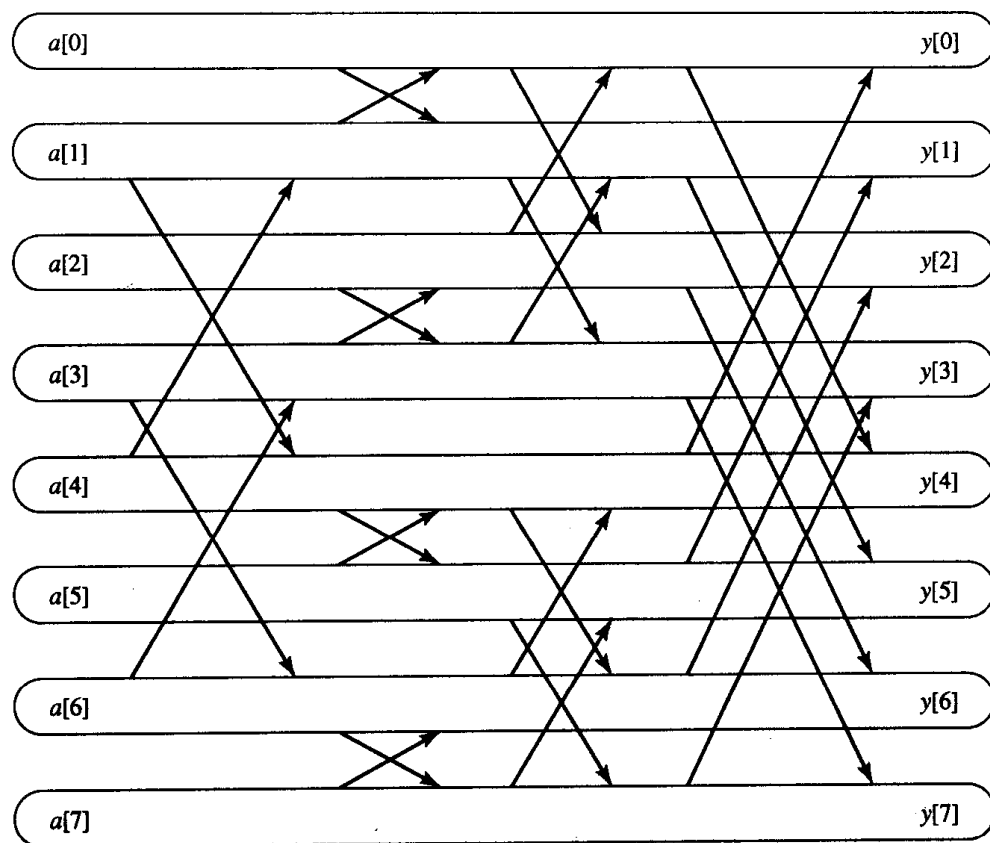


图 15.6 当  $n=8$  时的 FFT 算法的任务/管道图。任务由长的圆角矩形表示，算法的每一步需要的管道如图所示

### 15.5.2 聚合与映射

根据向量中的相邻元素将原始任务聚合起来可以减少一些通信。例如， $n=16$  并且  $p=4$ ，则进程 0 负责下标为 0, 1, 2, 3 的系数；进程 1 负责下标为 4, 5, 6, 7 的系数。

我们可以画出另外一幅任务/管道模型图来描述任务之间的通信模式。在图 15.7 中，每一个聚合的任务（进程）都表示为一个灰色的矩形。每个进程负责两个复数向量。第一个向量  $a$  中包含一段连续的输入系数。第二个向量  $y$  保存中间变量。在计算结束时，数组  $y$  包含变换后的一组连续的元素值。

这个并行算法有三个阶段。在第一阶段中进程对  $a$  的元素进行排列。这是一个所有进程之间相互通信的实例。在第二阶段进程执行 FFT 的前  $\log n - \log p$  次迭代，这个过程需要进行复数乘法、加法与减法运算。此时不需要传递消息。在第三阶段进程执行 FFT 的最后的  $\log p$  次迭代，要进行  $y$  之间的元素交换与乘法、加法、减法。我们可以将进程的组织看成一个逻辑上的超立方体。在每一个最后的  $\log p$  次迭代中，超立方体不同维的一对进程进行数据交换。

### 15.5.3 等效分析

每一进程执行等额的一部分计算。因为串行算法的计算复杂度是  $\Theta(n \log n)$ ，因此并行算法的计算复杂度是  $\Theta(n \log n/p)$ 。

每一个进程至多负责  $a$  的  $(n/p)$  个元素。我们假设进程组织成逻辑上的超立方体。全部进程通信以超立方体每一维之间一系列交换来实现；这样的时间复杂度为  $\Theta[(n/p) \log p]$ 。在每一个进程中该进程和超立方体中的相邻进程之间交换  $n/p$  个值的迭代操作有  $\log p$  次。这些交换的时间复杂度为  $\Theta[(n/p) \log p]$ 。基于以上假设，算法的总的通信复杂度为  $\Theta[(n/p) \log p]$ 。

现在我们来确定并行算法的等效。串行算法的时间复杂度为  $\Theta(n \log n)$ 。并行开销是  $p$  倍的通信复杂度。因此等效方程是：

$$n \log n \geq C n \log p \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C$$

FFT 算法的可扩展性和超快速排序以及 PSRS 算法的可扩展性相似。

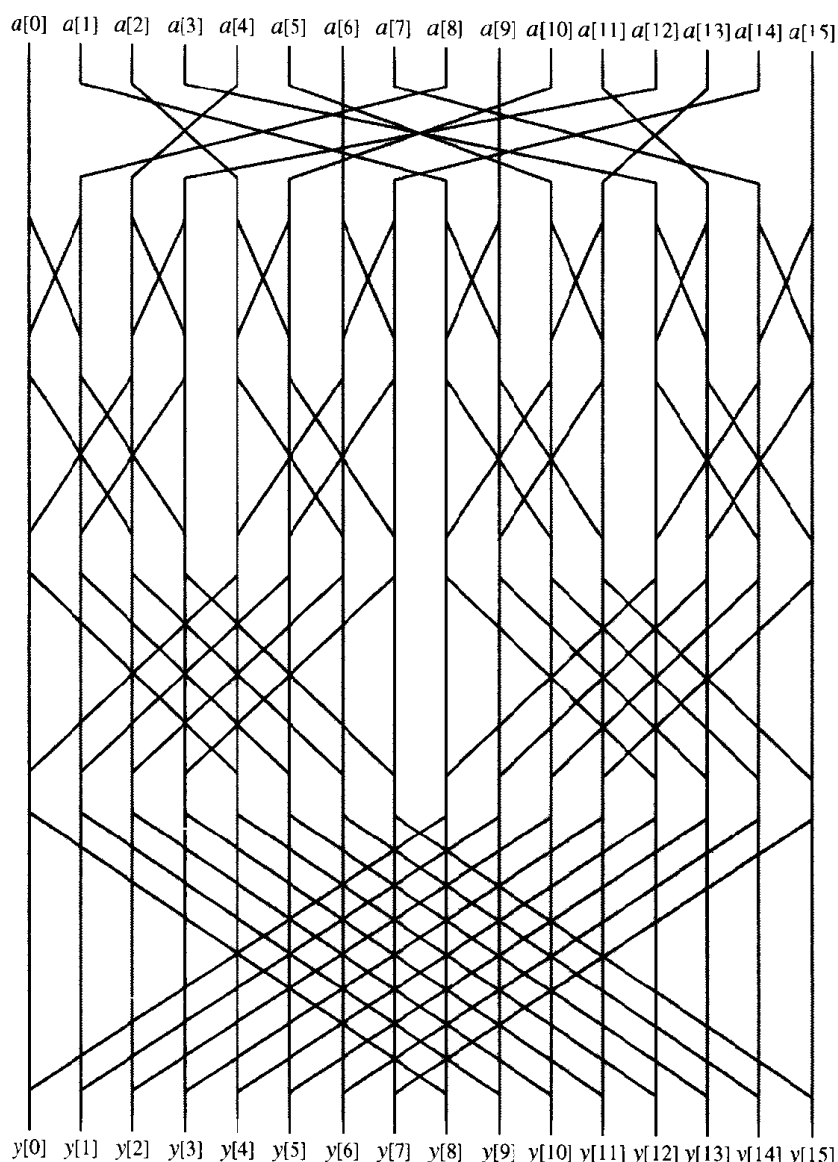


图 15.7 假设我们已经实现了在一个  $p$  进程多计算机上的  $n$  元素 FFT，这样在计算的开始时每一个进程包含一组连续的输入元素，在计算结束时每一个进程包含一组连续的输出元素。输入元素的初始计算需要一个全部进程之间的通信，在 FFT 中的开始的  $\log n - \log p$  次迭代不需要通信，在 FFT 中的最后的  $\log p$  次迭代需要每个进程与超立方体中某相邻维的进程交换数据。本图说明了  $n=16$  且  $p=4$  时的情况

## 15.6 本章小结

离散傅立叶变换在科学和工程应用中都有很重要的作用。快速傅立叶算法的意义在于两个方面。首先,它是离散傅立叶变换算法的复杂度为  $\Theta(n \log n)$  实现,而原始的现实事件复杂度为  $\Theta(n^2)$ 。其次,快速傅立叶变换易于并行化。

在本章中我们给出了离散傅立叶变换算法并演示了一组离散数据如何用一个连续函数来建模,连续函数表示为正弦和余弦函数的和。我们还介绍了离散傅立叶逆变换并演示了离散傅立叶变换和离散傅立叶逆变换是如何计算多项式乘法的。我们开发了递归的和迭代的快速傅立叶变换算法实现,并且给出了一个在多计算机上快速傅立叶变换的实现。

## 15.7 主要术语

Butterfly Pattern

Fourier Analysis

discrete Fourier transform

frequency

transform

fast Fourier transform

halving lemma

蝶形模式

傅立叶分析

离散傅立叶逆变换

频率

变换

快速傅立叶变换

二分引理

## 15.8 参考文献

用计算机科学的方式描述串行快速傅立叶变换算法的文献有 Baase 和 Van Gelder 【5】以及 Cormen et al. 【18】。

关于其他的并行快速傅立叶变换算法请参看 Foster 【31】、Grama et al. 【44】以及 Wilkinson 和 Allen 【115】。

## 15.9 练习题

15.1 以  $re^{i\theta}$  的形式表示以下复数。

(a) 1

(b)  $i$

(c)  $4 + 3i$

(d)  $-2 - 5i$

(e)  $2 - i$

15.2 对在  $2, \dots, 5$  范围内的每一个值  $n$ , 给出单位元的主  $n$  次方复根, 以  $x+iy$  和  $re^{i\theta}$  的形式给出。

15.3 对以下的每一个向量, 给出 DFT 的结果。

(a)  $(7, 11)$

(b)  $(13, 17, 19, 23)$

(c)  $(2, 1, 3, 7, 5, 4, 0, 6)$

15.4 对以下的每一个向量, 给出逆 DFT 的结果。

(a)  $(3, -2)$

(b)  $(10, -2 + 2i, -2, -2 - 2i)$

(c)  $(14, -3 - 4i, 1 - i, -1 + 3i, 0, -1 - 3i, 1 + i, -3 + 4i)$

15.5 基于本章讨论的设计实现一个并行 FFT 程序。针对不同的  $n$  和  $p$  测试你的程序。

15.6 实现一个逆 FFT 算法的串程序。

15.7 实现一个逆 FFT 算法的并程序。在不同个数的处理器上以不同大小的向量测试你的程序。

15.8 不计入开始时的全部进程之间的通信, 则快速傅立叶变换算法的主体为一个蝶形通信模式。给出在前面的章节中同样使用蝶形通信模式的并行算法。

15.9 FFT 算法的可扩展性是和超快速排序算法的可扩展性相似的。解释两种算法的相似性。

# 第 16 章 组 合 搜 索

Attempt the end, and never stand to doubt;

Nothing's so hard but search will find it out.

Robert Herrick, "Seek and Find," Hesperides

## 16.1 概 述

组合算法在离散、有限的数学结构上进行计算。组合搜索是“在定义好的问题空间中找到一个或多个最优或次优解”【109】，在广泛的领域中得到应用，例如：

- 在 VLSI 布线时最小化连线所占的芯片面积
- 规划机器人手臂的运动以最小化总的移动距离
- 为航班分配人员
- 定理证明
- 游戏

有两种类型的组合搜索问题。解决决策问题的算法试图找到满足所有约束条件的解决方案。对于决策问题的回答或者为“是”，意味着该问题有解，或者为“否”，意味着无解。下面是决策问题的一个例子：“是否存在一条路线使得机器人手臂可以经过每个钻孔点而且移动的总长度小于 15 米？”解决优化问题的算法则必须要找到最小化（或最大化）目标函数的解。下面是一个优化问题的例子：“找到机器人手臂访问每个钻孔点的最短路径”。

本章讨论了用于解决决策问题和优化问题的 4 种组合搜索算法，包括分治法、回溯法、分支定界法以及 Alpha-Beta 剪枝法。不同的算法用于在不同的搜索树中进行查找。在所有情况下，搜索树的根节点表示最初要解决的问题，但根据搜索树的类型，可以有多种多样的非终端节点。一个 AND 节点代表一个问题或子问题，该问题仅当其所有子问题都得到解决时其本身才被解决。一个 OR 节点代表一个问题或子问题，该问题在其任意子问题得到解决时其本身就得到解决。在 AND 树中的每个非终端节点都是一个 AND 节点，如图 16.1 (a) 所示。分治法所对应的搜索树是一颗 AND 树，因为问题的解决可以通过其所有

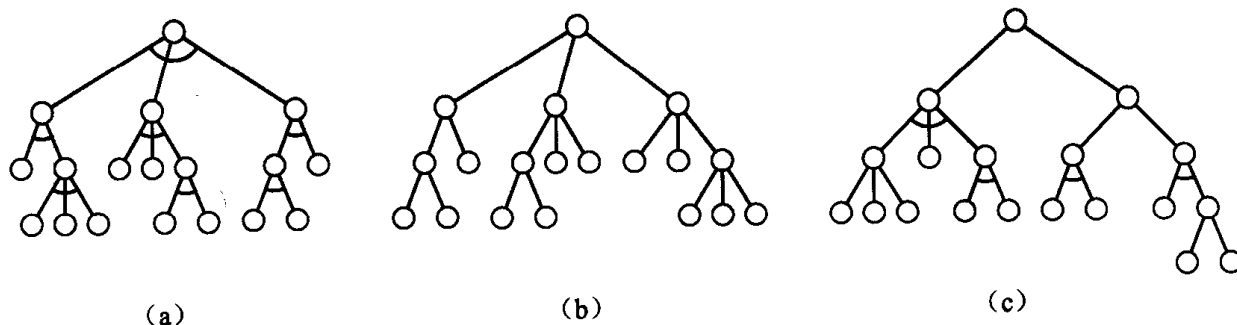


图 16.1 搜索问题可以用树来表示：(a) AND 树；(b) OR 树；(c) AND/OR 树

子问题解决的组合而获得。在 OR 树中，每个非终端节点也是一个 OR 树，如图 16.1 (b) 所示。AND/OR 树是由 AND 非终端节点和 OR 非终端节点来表示的，如图 16.1 (c) 所示。游戏树就是 AND/OR 树的示例。

## 16.2 回溯搜索

回溯搜索是使用深度优先搜索的组合优化问题的另一种可行解法。给定原始问题（状态空间的根节点），回溯生成其孩子节点并选择其中一个继续搜索。回溯法在选定的节点上递归地使用相同的方法。如果搜索遇到了一个无法扩展的节点，例如，一个“死节点”（dead end），或是所有的孩子的子树都已被搜索过，那么就回溯到上一个节点。

### 16.2.1 示例

我们以一个生成纵横字谜的问题为例。给定一个空白的纵横字谜（见图 16.2 (a)）以及一个包含了词和短语的字典，我们的目标是将字母指定到空白格子中使得每行和每列中的字母包含字典中的词和短语，如图 16.2 (b)。这是一个决策问题：我们在试图回答“是否能够用给定字典里的词来填充这个给定的纵横字谜模式？”

空白纵横字谜中的每个数字对应于一个水平词的起始，或是一个垂直词的起始，或是同时是两个方向的词的起始。我们用“不完全词”表示还没有被完全确定的词；即至少有一个字母还没有被确定的词。

1	2	3		4	5	6
7				8		
9			10			
		11				
12	13				14	15
16				17		
18				19		

(a)

1	2	3		4	5	6
U	M	P		G	I	N
7	P	O	E	8	E	W
9	S	P	A	10	R	O
			11	C	O	B
12	P	13	R	O	D	I
16	S	A	C		17	L
18	I	N	K		19	S
						P
						A

(b)

图 16.2 纵横字谜问题是从一个空白字谜模板和一个包含了词和短语的字典中生成一个纵横字谜的解。  
(a) 空白字谜模板 (b) 一个纵横字谜的解。这个解是使用 Crossword Weaver 生成的 ([www.CrosswordWeaver.com](http://www.CrosswordWeaver.com))

为了填充字谜，我们使用下面的搜索策略：找出字谜中的最长不完全词，并在字典中查找具有该长度的词或短语。如果有多个词符合条件，我们随意选取一个。随后的步骤中，我们找到至少有一个确定字母的最长不完全词，并从词典中查找具有相同长度并且与已确定字母匹配的词。如果有多个词或词组满足要求，我们还是任选一个。我们按照上述方式



继续下去,直到没有不完全词为止。

使用上面的标准,我们可以定义填充不完全词的顺序。下面是一个满足标准的顺序:  
3 DOWN, 9 ACROSS, 4 DOWN, 12 ACROSS, 1 DOWN, 2 DOWN, 5 DOWN, 6 DOWN,  
10 DOWN, 12 DOWN, 13 DOWN, 14 DOWN, 15 DOWN。

选词的过程可以用状态空间树来表示。在树的根部是空的纵横字谜。根的孩子代表所有可以用于填充的不完全词 3 DOWN 的长度为 7 个字母的词。树中的每个节点代表在当前已确定词的情况下,将字典中的一个词指定给一个不完全词。如果需要  $d$  次指定才能完全填充空白的格子,则树的深度为  $d$ 。每个树的叶子节点都代表一个纵横字谜问题的解,因此状态空间树是 OR 树的一个例子。

我们可以通过在状态空间树上进行回溯搜索来查找纵横字谜问题的解。回溯搜索得名于在遇到死节点时,算法会回溯到上一层来寻找其他可行解。

我们看看回溯搜索时如何用来解决图 16.2 中的纵横字谜问题的。图 16.3 为搜索的图示。

第 1 步是为 3 DOWN 找到一个 7 字母词。在字典中查找后我们选择了 TROLLEY。

第 2 步是考虑 9 ACROSS。从字典中找第 3 个字母是 O 的 7 字母词,我们找到了 CLOSETS。

第 3 步是为 4 DOWN 找一个第 3 个字母是 E 的 7 字母词。假设我们在字典中无法找到这个词。在这种情况下,我们必须回溯,为 9 ACROSS 找另外一个第 3 个字母是 O 的 7 字母词。这一次,我们选择了 CROQUET。

在填充了 9 ACROSS 后,我们回到 4 DOWN,找第 3 个字母是 U 的 7 字母词。字典中的 TRUMPED 满足这个条件。

搜索过程继续进行,直到没有不完全词或是字典中的所有可能词已被用尽。

## 16.2.2 时间和空间复杂性

如果树的平均分支因子为  $b$ ,那么在最坏情况下搜索深度为  $k$  的树需要考察的节点个数为

$$1 + b + b^2 + \dots + b^k = \frac{b^{k+1} - b}{b - 1} + 1 = \Theta(b^k)$$

换句话说,在最坏情况下,对状态空间的回溯搜索需要指数时间。

但是,回溯搜索所需的内存容量搜索深度的线性函数,即  $\Theta(k)$ ,因为在内存中仅需要维护状态空间树中每层当前的选择。因此,回溯搜索在求解问题规模上的限制因素是计算机的速度而不是内存容量。

## 16.3 并行回溯算法

既然我们是在处理一个在最坏情况下具有指数时间的算法,其中应该有很多并行性的机会。我们应该如何进行并行回溯搜索呢?

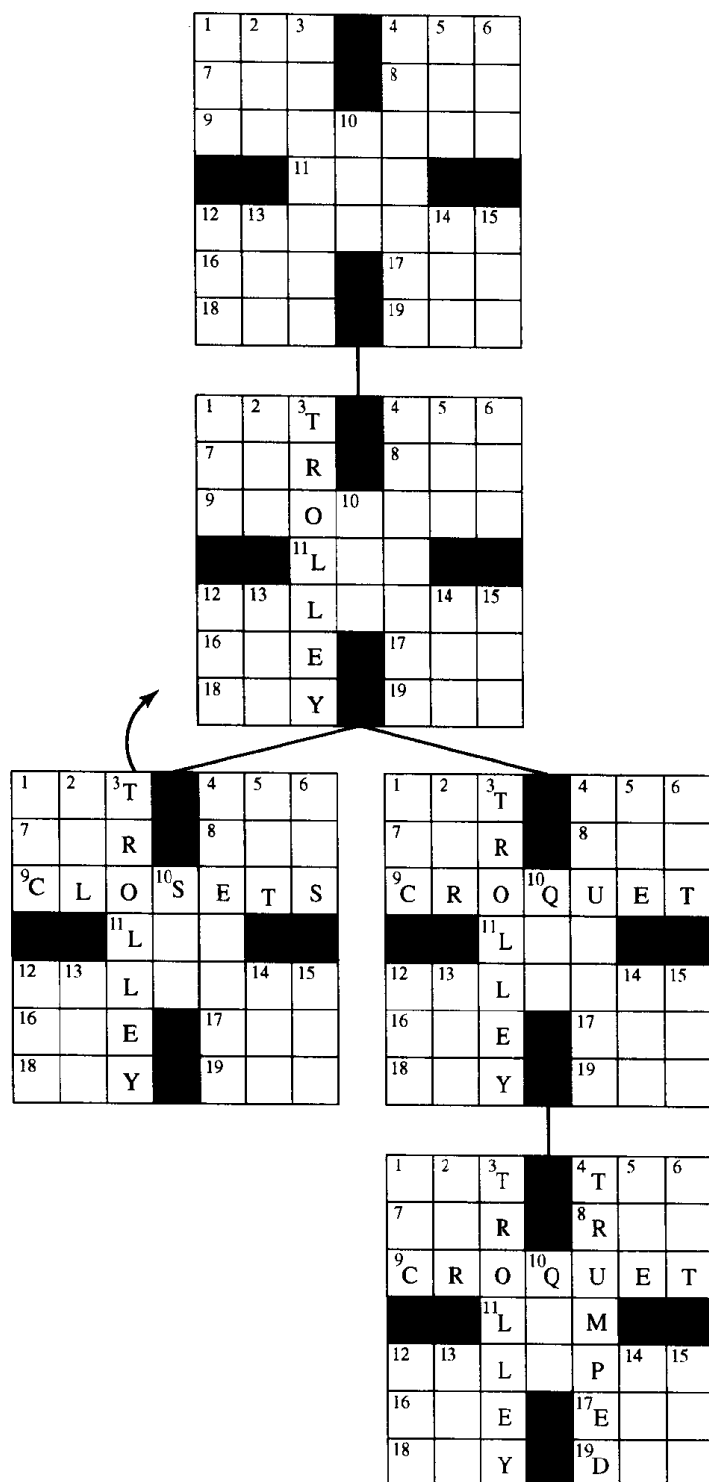


图 16.3 一个纵横字谜问题的状态空间树的回溯搜索法的开始步骤

一个显而易见的策略是将子树搜索分配给各个进程，如图 16.4 所示。假定树有分支因子  $b$  并且进程数为  $b^k$ 。每个进程搜索都搜索到状态空间树的第  $k$  层，然后就仅搜索第  $k$  层的一个节点的子树。如果搜索深度  $d$  比  $2k$  更大，那么每个进程进行前  $k$  层搜索所需的时间相对比较小，整个并行算法可以达到较好的加速比。

如果不存在  $k$  使得  $p=b^k$ ，串行搜索可以搜索到状态空间树的第  $m$  层，每个进程可以承担第  $m$  层节点的部分子树的搜索。

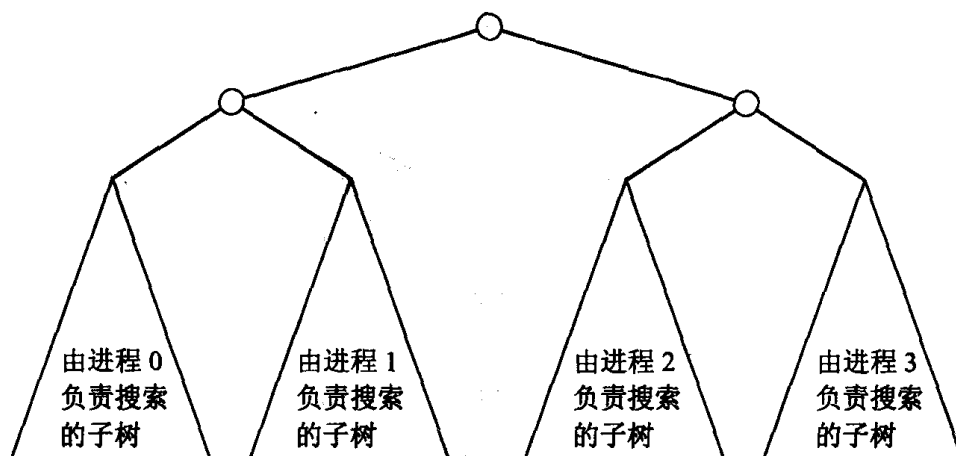


图 16.4 如果  $p=b^k$ ，那么每个进程可以搜索到状态空间树的第  $k$  层，然后仅搜索第  $k$  层某一个节点的子树。如果所有子树具有相同数目的节点，那么这是一个好的策略。如果子树的大小有显著区别，这个策略就是一个差的策略

例如，假定状态空间树的分支因子为 3，我们可以搜索 10 层深。进一步假设我们希望用 5 个进程来执行搜索。如果并行搜索从状态空间的第 0 层开始，那么只有一个节点需要搜索，也仅有一个进程需要工作，得到的加速比为 1。

如果并行搜索从第 1 层开始，一个有 3 个节点需要搜索，可以使用 3 个进程。扩展根节点所需的时间可以忽略，因此我们得到的加速比非常接近于 33。

如果从第 2 层开始进行并行搜索，一个有 9 个节点需要搜索。4 个进程每个搜索 2 棵子树，1 个进程搜索剩下的 1 个，得到的加速比接近  $9/2=4.5$ 。

不存在  $k$  的值使得  $3^k$  是 5 的倍数。但是，我们搜索得越深，就生成了越多的节点，意味着我们可以更加均匀地将他们分配给进程，从而提高加速比。另一方面，搜索得越深也意味着每个进程花在产生开头层次的时间越多。这些计算是冗余的，增加了整个计算中串行计算的比例。图 16.5 画出了并行计算开始的状态空间层次与最好加速比之间的关系。可以看出，最优加速比在很大的深度范围内保持着高水平。

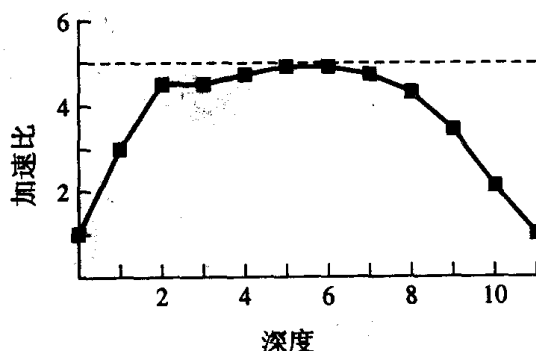


图 16.5 在分支因子为 3 深度为 10 的状态空间树上，用 5 个进程使用回溯搜索法所能达到的最大加速比与开始进行并行搜索的层数之间的函数关系

不幸的是，大多数情况下状态空间树不是平衡的。一些子树比另外一些子树具有多得多的节点。例如，在纵横字谜游戏中，一些初期选择的词会导致死节点并比另外一些词要回溯得早得多。因此我们需要一个能够在不平衡树的情况下也能得到较好性能的算法。

我们的方法是让串行搜索进行到足够深的层次，让每个并行任务负责大量的子树。这



设为 0, 这个变量用来表示在状态空间树中, 本进程在第 `cutoff_depth` 层所遇到的节点个数。每个进程然后调用函数 `Parallel_Backtrack`, 参数为 `board` 和 `level`。进程搜索整个树直到 `cutoff_depth`。每个进程在分配给自己的以第 `cutoff_depth` 层节点为根的子树上进行回溯搜索。

## 16.4 分布式终止检测

注意到图 16.7 中的每个进行并行回溯搜索的进程都仅在搜索完了自己所分配到的子树后才会停止。换句话说, 该算法找出了所有的解。当使用回溯算法来解优化问题时, 进程必须找出所有解然后从中选择最优的。

但是, 有些时候我们仅需要一个解就可以了。在这种情况下, 我们希望所有进程可以在一个进程找到了一个解后能够尽快停止。那么进程如何知道什么时候停止搜索呢? 如果我们希望一个进程在完全搜索其所分配的状态树空间之前就停止, 找到解的进程必须向其他进程直接或间接地发送一个消息, 所有进程也必须周期性地检查消息的到来。一种增加周期性的消息检查的方法是让进程在搜索到某一特定层次 (如第 `cutoff_depth` 层) 的时候查看消息。函数 `MPI_Iprobe` 可以用来实现这种检查, 因为该函数可以以非阻塞的方式让进程确定是否有消息到达。将这个功能增加到函数 `Parallel_Backtrack` 中是十分容易的。

一个简单的 (但是不正确的) 终止程序的方法是让找到解的进程发送一个消息到所有其他进程。一个进程在发生下列事件之一时就停止:

- 该进程找到了一个解并已向其他进程发送了消息。
- 该进程从其他进程接收到一个消息。
- 该进程已经完全搜索了所分配到的状态空间树。

不幸的是, 如果一个进程先调用了 `MPI_Finalize`, 然后其他活动进程又试图向它发消息, 我们会得到一个运行时错误。我们上面所描述的方法可能会导致这个错误。这是如何发生的呢?

下面是可能会导致这个运行时错误的一个场景。假设进程 A 找到了一个解, 发送消息给其他进程, 并调用 `MPI_Finalize` 结束。同时, 进程 B 也发现了另一个解, 并在接到进程 A 发来的消息前向其他进程发送了消息。如果进程 B 试图在进程 A 调用了 `MPI_Finalize` 后向进程 A 发送消息, 那么就会产生一个运行时错误。

我们必须保证所有进程都处于不活动状态并且没有消息处在传送过程中, 才能够允许各个进程调用 `MPI_Finalize`。这称为分布式终止检测问题。大约 20 年前, Dijkstra, Seijen 和 Gasteren 发明解决这个问题的方法【21】。

图 16.8 表示了他们的算法。进程组织成一个逻辑环, 如图 16.8(a) 所示。一个进程 (在本例中是进程 0) 通过向在环上的后继节点发送一个令牌来检测系统的状态。如果令牌返回了进程 0, 它将能够确定全体进程是否可以安全结束。

每个进程都有一个颜色和消息计数。当一个进程开始执行的时候, 它是白色的并且消息计数为 0。一个进程在发送和接受消息的时候变成黑色。一个进程发送了一个消息后, 其消息计数会增加。当进程接收消息时, 其消息计数会减少。

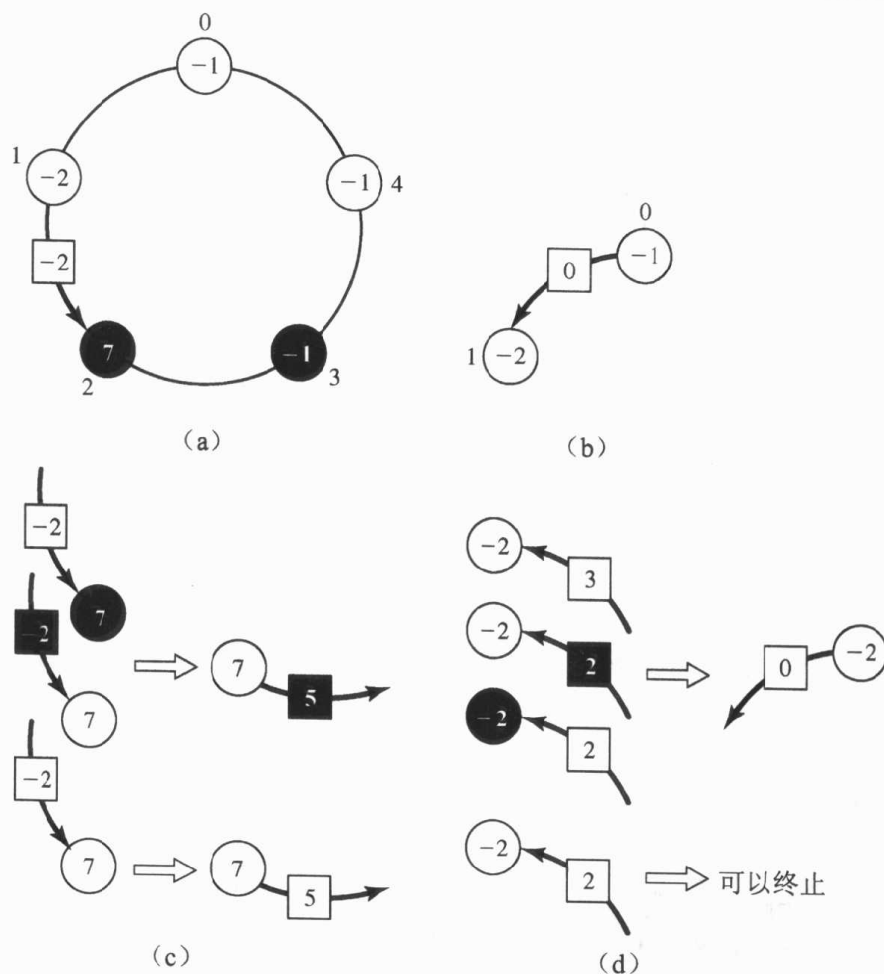


图 16.8 Dijkstra 等的算法用来解决分布式终止问题。(a) 一个令牌(正方形)在进程(圆圈)的逻辑环上传递。(b) 进程 0 发起了一次探测。(c) 一个中间进程修改了令牌并将其继续传递下去。(d) 令牌返回进程 0

Dijkstra 等的算法的主要思想是,如果所有进程都是白色的并且所有的消息计数值的和为 0, 则我们知道系统中没有正在传递的消息, 可以将进程停止。

传递的令牌也有颜色和计数。当进程 0 发起检测的时候, 令牌是白色的, 计数值为 0, 如图 16.8 (b) 所示。

现在让我们看看中间进程是如何处理令牌的, 如图 16.8 (c) 所示。当一个进程接到令牌后, 它将自己的消息计数累加到令牌的消息计数上。如果进程正处于活动状态, 它将令牌保留直到自己处于不活动状态。此时, 如果进程是黑色的, 它将令牌也变成黑色的。否则, 该进程不改变令牌的颜色。该进程将自己的颜色变成白色并将更新的令牌发送给其后继进程。

最终令牌会回到进程 0, 如图 16.8 (d) 所示。如果令牌是白色的, 进程是白色的, 并且令牌的计数与进程 0 的消息计数之和为 0, 那么系统处于静止状态, 可以安全地结束进程(可以通过让进程 0 向其他进程发送退出消息来完成)。否则, 进程 0 必须再次对进程环进行检测。

对于我们的并行回溯算法, 我们可以实现如下的分布终止检测算法: 所有进程在开始搜索的时候消息计数为 0。当一个进程发现了一个解的时候, 它发送一个“找到解”的消息给进程 0 并将自身的消息计数设为 1。当进程 0 接收到“找到解”的消息后, 它将自己

的消息计数减 1。如果进程 0 自身找到了解,或是从其他进程收到了“找到解”的消息后,进程 0 发起一个分布终结检测过程,初始化一个令牌并把它传递给进程环中的后继。

如果进程正在主动地搜索,在收到令牌后该进程会迅速停止搜索,因为接收到令牌说明其他进程已经找到了解。因此我们不必担心进程会由于令牌而被挂起。中间进程只是在恰当的情况下修改令牌的颜色和计数,并将其传递下去。注意进程在收到或发送令牌的时候不需修改其自身的消息计数。

当进程 0 接到令牌并确定系统处于静止状态时,它发送一个“结束”消息给所有其他进程,并调用 `MPI_Finalize` 和 `exit`,其他进程在接到“结束”消息后也调用 `MPI_Finalize` 和 `exit`。

## 16.5 分支定界法

分支定界法是回溯法的变种,通过考虑部分解的最优性来避免搜索不可能是最优的解。

### 16.5.1 示例

作为分支定界法的一个例子,我们考察 8-puzzle 游戏,如图 16.9 所示。8-puzzle 是 Sam Loyd 于 1878 年发明的著名的 15-puzzle 游戏的一个简化版本。8-puzzle 包括 8 块瓷砖,放在  $3 \times 3$  的板上,其中 8 个位置上各有 1 块瓷砖,还有一个空位。游戏的目标是不断使用与空位相邻的瓷砖去填补空位,直到所有瓷砖都按行排好序。与孩子们在夏令营中随意移动瓷砖以达到正确的排列不同,我们的目标是使用最少的步骤来得到结果——一个优化问题。

我们可以使用状态空间树来表示可以从最初位置到达的板的位置,如图 16.10 所示。一种解决问题的方法是对状态空间树采用广度优先法搜索,直到找到目标状态。但是我们的目标是考察尽量少的候选移动。如果我们能够把每个状态与在该状态下达到目标所需的最少步骤的估计联系起来,我们就可以达到搜索更少的节点的目的。

一个评估函数将已经移动的步数与每块瓷砖到其正确位置的 Manhattan 距离相加,如图 16.11 所示。给定这样的函数,我们可以将我们的搜索集中到最有希望的移动上。我们总是从具有最小评估函数值的节点继续我们的搜索。如果 2 个或多个节点具有相同的评估值,我们选择距离根节点距离最远的节点。如果有 2 个或多个节点具有相同的评估函数值而且距根节点也一样远,我们可以任意选取一个。

图 16.12 中给出了用分支定界法解一个 8-puzzle 游戏的例子的过程。我们看看如何为每个节点指定沿着该节点搜索下去所得解的开销的下界。我们要查看的第一个节点是树的根。在根所代表的状态中,瓷砖 2, 3, 5, 6 不在正确的位置上,到其正确位置的距离分别

1	2	3
4	5	6
7	8	空位

图 16.9 8-puzzle 的目标状态。  
8-puzzle 是 Sam Loyd 于 1878 年  
发明的著名的 15-puzzle  
游戏的一个简化版本

为 1, 2, 1 和 1 (计算 Manhattan 距离)。距离的和为  $1+2+1+1=5$ 。由于现在只进行了 0 次移动, 根节点的解的开销下界为 5 (5 只是下界, 不是真实的界, 因此问题的解所实际需要的步骤可能大约 5)。

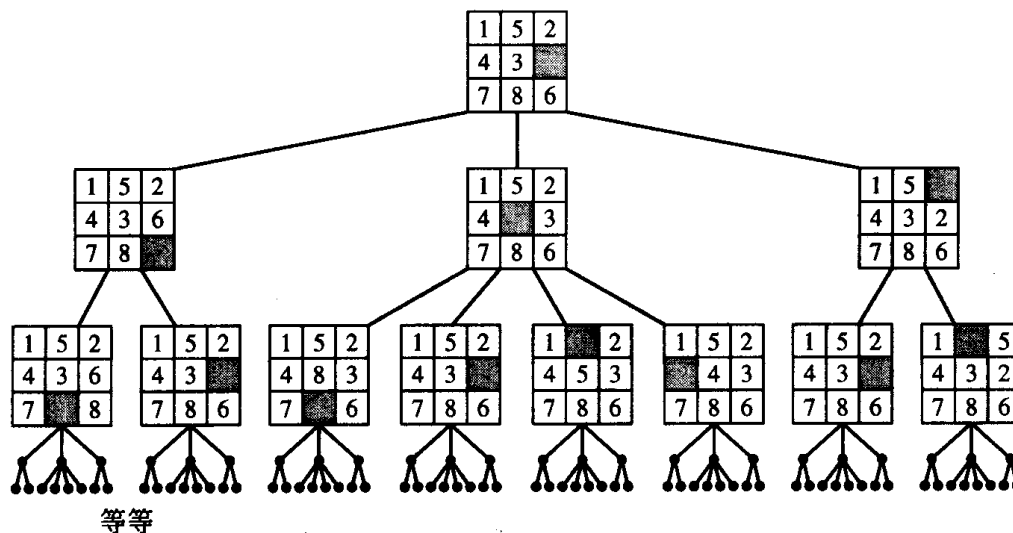


图 16.10 8-puzzle 游戏搜索的部分状态空间树

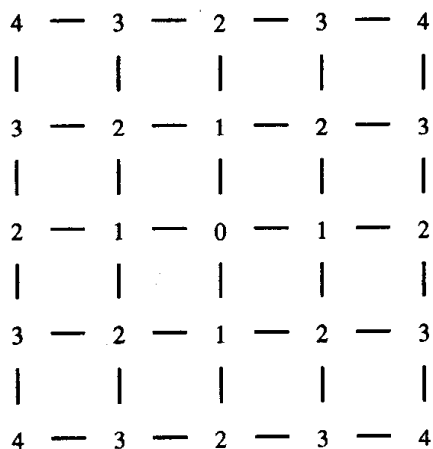


图 16.11 一对点之间的 Manhattan 距离是指这一对点之间使用水平或垂直移动所形成的路径中最短的。  
(想象在矩形的街道网格中从一个十字路口走到另一个十字路口。)用形式化的方法描述,  
具有坐标  $(x_1, y_1)$  和  $(x_2, y_2)$  的两个点之间的距离是  $|x_1 - x_2| + |y_1 - y_2|$

现在让我们考虑根节点的左孩子。瓷砖 2, 3 和 5 到正确位置的距离分别为 1, 2 和 1。总是  $1+2+1=4$ 。由于已经进行了一次移动 (该节点在状态空间的第 1 层), 使用该移动的所有解的开销下界为 5。

最后, 让我们考虑根节点的左孩子的左孩子。瓷砖 5, 2, 3 和 8 不在正确位置上, 距正确位置的距离总和为 5。由于此时已经进行了 2 次移动, 使用这两次移动的解的开销下界为 7。

最佳情况优先搜索 (the best-first search) 集中搜索具有最小下界的节点。实际搜索发现了这个问题解, 该解仅需 5 次移动。此时我们已经不需要再寻找更好的解了。从我们得到的下界得知, 其他解至少需要 7 步。注意分支定界法搜索的节点数要远远少于广度优先



搜索。

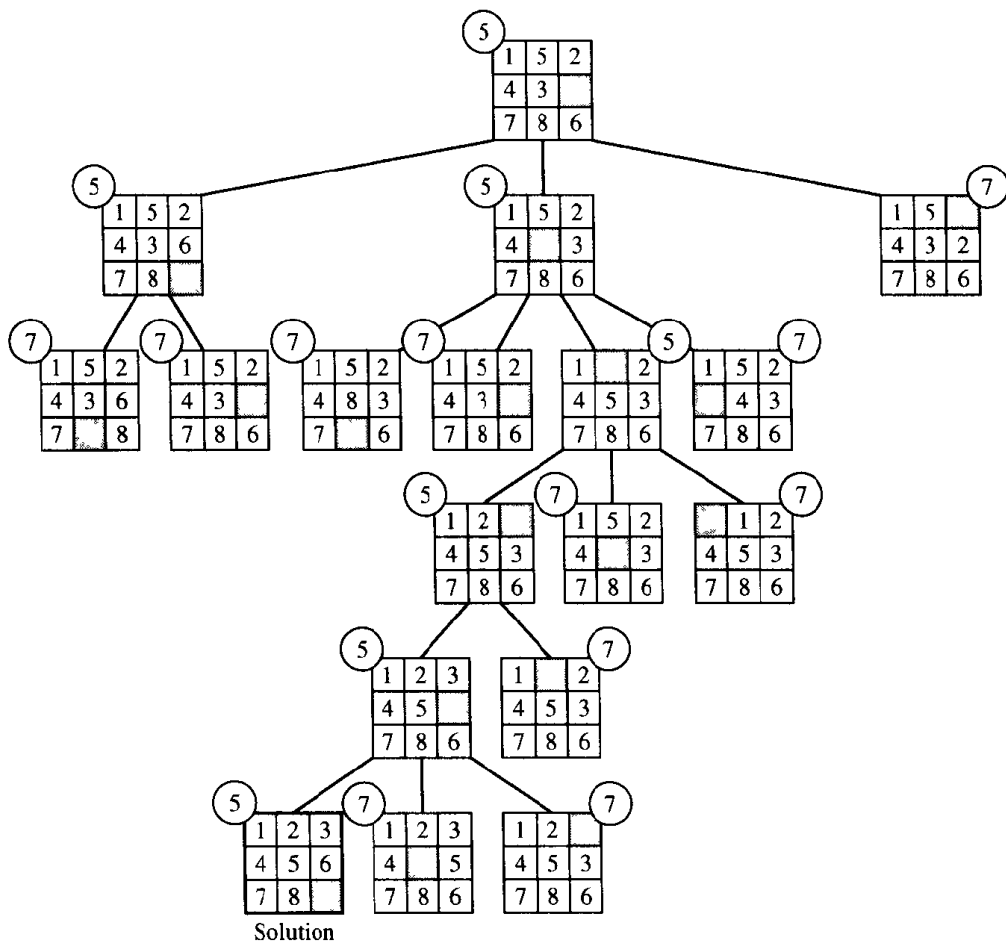


图 16.12 使用最好情况优先的分支定界法搜索 8-puzzle 问题的解。  
实际搜索过的节点构成了一棵高度不平衡的数

## 16.5.2 串行算法

现在我们已经看过了一个实际的例子，让我们开发一个更加通用的分支定界法。给定一个初始问题和要被优化的目标函数  $f$ ，分支定界法将问题分解为 2 个或多个子问题的集合。每个子问题由 1 个或多个约束标识。我们重复分解过程直到每个子问题都被分解、解决或是证明不可能包含原问题的最优解。

在 8-puzzle 的例子中，问题是将各个瓷砖按顺序排列。目标函数  $f$  是排序所需要的移动次数。如果瓷砖已经是按行排序的，问题就得到了解决。否则，分支定界法将问题分解为若干个子问题，每个子问题对应一个合法的移动。对瓷砖的移动可以表示为增加约束条件。

正如我们在 8-puzzle 中看到的，我们使用状态空间树来表示对原问题的分解过程。树的节点对应于分解后的自刎体，树的边对应于分解过程。原问题是树的根节点。树的叶子节点表示被解决的子问题或是未经进一步分解就被废弃的子问题。

分支定界法的目标就是仅搜索树中的少数节点来解决问题。假设我们希望得到具有最

小开销的解  $f^*$ 。对分解的每个子问题都计算定界函数  $g$ ，该函数代表在该子问题的约束下，该子问题的解所可能达到的最小开销。在从根节点到终端节点的任意路径上，下界都是非递减的。另外，对于代表问题可行解的叶子节点，该节点的下界函数  $g(x)$  的取值必须与目标函数  $f(x)$  相同。代表不可行解的叶子节点的下界函数值是无穷大。图 16.13 是状态空间树的另一个例子。节点内的数是相应子问题的下界。代表可行解的节点用加重的圆圈表示。该问题的最优解的开销为 18，即  $f^*$  的值是 18。

在分支定界法执行的任何时候都存在一些被生成但没有被考察的子问题。搜索策略决定了从未被考察问题中选择问题进行考察的顺序。最佳优先策略选取具有最小下界的候选子问题。在最小下界相同时，选取在状态空间树中最深的子问题（即具有最多约束的子问题）。如果深度也相同，就任意选取一个。图 16.13 给出了最佳优先搜索策略考察节点的顺序。

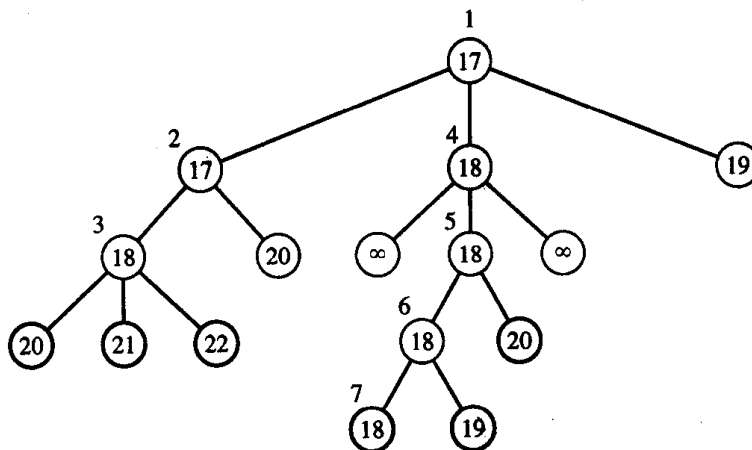


图 16.13 状态空间树的另一个例子。节点内的数是相应子问题的下界。代表可行解的节点用加重的圆圈表示。该问题的最优解的开销为 18。圆圈边上的数字代表使用最佳优先搜索策略时所考察节点的顺序

图 16.14 给出了一个采用最佳优先策略的通用串行分支定界法算法的伪代码。最佳优

```

initial — 初始问题
q — 优先队列，可以在该队列上进行初始化、插入和删除最小元素等操作
s — 迄今为止找到的最优解
u — 状态空间树节点
v — 具有附加约束的新节点

Initialize (q)
Insert (q, initial)
repeat
    u ← Delete_Min(q)
    if u 是一个解 then
        Print_Solution (u)
        Halt
    else
        for i ← 1 to Possible_Constraints(u) do
            给 u 增加约束 i，并创建 v
            Insert(q, v)
        endfor
    endif
forever

```

图 16.14 采用最佳优先策略的通用串行分支定界算法

先搜索采用了优先队列抽象数据类型。该数据类型支持的操作包括 Insert (插入), Delete\_Min (删除最小值) 或 Delete\_Max (删除最大值)。程序员通常用堆来实现优先队列。

### 16.5.3 分析

分支定界法的目标是通过使用下界函数来去除那些不可能达到最优解的节点, 减少必须考察的节点个数。在最坏情况下, 下界函数会使得算法在状态空间树上进行广度优先搜索, 而没有对子树进行任何修剪。如果最优解在状态空间树的第  $k$  层上, 平均分支因子是  $b$ , 那么最佳优先分支定界法的最坏时间复杂度为  $\Theta(b^k)$ 。

现在让我们考虑采用最佳优先策略的串行分支定界算法的空间复杂性。注意到优先队列保存了所有未经考察的子问题。如果状态空间树的平均分支因子为  $b$ , 则算法每在优先队列中删除一个节点, 就会平均插入  $b$  个节点。在最坏情况下, 搜索深度为  $k$  时所需的内存为  $\Theta(b^k)$ 。使用最佳优先分支定界法所能解决的问题规模通常会受限于内存空间。

## 16.6 并行分支定界法

在这一节中我们将开发一个适合在多计算机和分布式多处理器系统上实现的并行分支定界算法。

### 16.6.1 存储和共享待解的子问题

串行算法将所有待解的子问题都存放在一个优先队列中。在分布内存的计算机上维护单一的优先队列是不现实的; 将一个待解的状态空间树节点发送给另一个处理器, 并接受该节点的解或是该节点的孩子节点所需的通信开销要大大超过进行本地计算所需的时间。但是如果让一个处理器进行所有的优先队列操作也同样会产生性能瓶颈, 会对能够用来解决问题的处理器个数产生限制。在一个处理器上维护单一的优先队列会使得我们无法在处理器个数增加的时候扩展问题的规模。

由于上述原因, 我们决定让每个进程维护自己的优先队列, 用于存放待解的子问题。在每个拥有非空优先队列的进程的每次迭代都从队列中删除具有最小下界的待解子问题。如果子问题不是问题的解, 它将被分解为  $k$  个子问题 (注意, 尽管每个进程都不断迭代进行同样的一系列操作, 进程之间并没有同步)。如果一个进程将问题分解为  $b$  个子问题, 那么它将这些新的子问题也放入到优先队列中。

偶尔某些进程也会将待解的子问题发送到其他进程 (图 16.15 (a))。在程序执行的开始, 进程 0 包含了在其优先队列中的原始问题。其他进程的优先队列此时为空, 处于不工作状态。在进程 0 分发了一个待解子问题后, 有两个进程处于活动状态。经过另一个分发过程后, 4 个进程拥有待解子问题。如果合理地分发待解子问题的话, 仅需  $\lceil \log p \rceil$  步就可以让所有进程都拥有待解子问题。

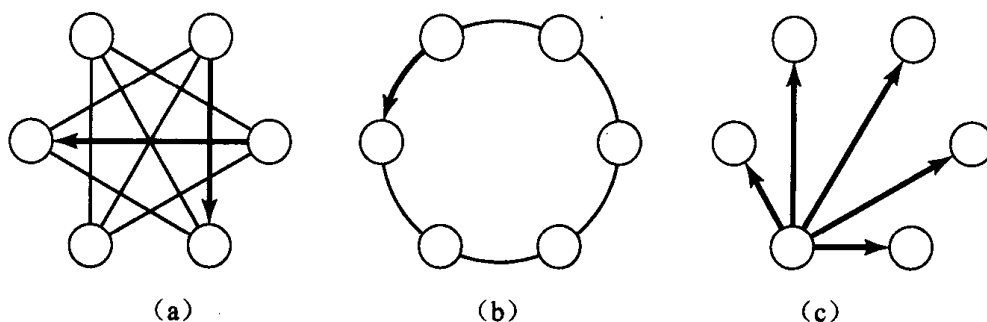


图 16.15 并行分支定界算法使用 3 种类型的消息。(a) 进程发送 UnexaminedSubproblem (待解子问题) 到其他进程。每个消息包括一个发送进程从自己的优先队列中删除的待解子问题。接收进程将接受到的待解子问题插入到自己的优先队列中。(b) 进程形成了一个逻辑环来传递令牌以进行分布终止检测。(c) 当确定其他进程可以安全退出的时候, 进程 0 发送一个终止消息到所有其他进程

## 16.6.2 效率

要找到问题的解并保证解的最优性, 必须满足两个条件: 第一, 至少有一个解节点需要被考察 (因此状态空间树中所有该节点的祖先也会被考察); 第二, 必须考察所有在状态空间树中下界比最优解小的节点。算法的执行时间由最后出现的事件决定, 而最后出现的事件是由进程数和状态空间树的形状决定的。

采用最佳优先策略的串行分支定界算法依赖于一个单独的优先队列, 在给定定界函数  $g$  的情况下, 考察尽量少的节点。该算法总是考察具有最小下界的节点, 因此一遇到解节点算法就可以停止执行, 因为没有具有更小下界的节点存在。

相反地, 采用最佳优先策略的并行分支定界算法可能会考察不必要的节点。因为每个进程只是考察局部最优的节点-即在本地优先队列中具有最小下界的节点。在众多进程中, 可以保证有一个进程在考察全局最优的节点, 但其他进程就不一定了。如果一个进程考察了一个下界大于最优解的节点, 那么这部分工作是无用的, 整个的并行计算效率也会随之下降。在并行算法的执行过程中, 进程会将未解子问题发送出去, 原因就是这样的会在所有进程上更好地分发低下界的子问题。另一方面, 传递未解子问题也增加了并行算法的通信开销。

## 16.6.3 停机条件

分支定界算法的分布式终止检测比回溯法更加复杂。在回溯搜索的情况下, 我们仅是寻找任意解。现在我们是在寻找一个最优解。进程所发现的第一个解不一定是最优解。因此只有在下列两个条件都满足的情况下, 我们才能够停止算法 (1) 找到解 (2) 确认没有更优解存在。假设我们在解决一个最小化问题, 即试图得到一个成本最小的解。如果已知最优解的成本小于或等于其他所有未解子问题的成本的下界, 则满足上面的两个条件。

可以通过修改 Dijkstra 等人的分布式终结检测算法来解决我们的问题。在一个进程接受了一条消息或处理了一个具有比已知最优解更小的下界的未解子问题之后, 其颜色变为黑色。一个进程如何知道已知最优解? 我们通过在终止令牌的后面附加一段信息来达到这

个目的, 如图 16.15 (b) 所示。

在原算法中终止令牌包括一个计数和一个颜色。现在我们再增加 2 个字段: 已知最优解的成本和解本身 (即达到解的移动步骤)。当一个进程接收到了令牌, 它更新颜色和计数字段, 并检查自己所发现的局部最优解是否比令牌上的更优。如果是, 它用自己的解来更新令牌上的解和成本字段。最后, 进程比较已知最优解的成本与其自身优先队列中的第一个待解子问题的下界, 如果待解子问题的下界大于或等于已知最优解的成本, 则不需要再继续考察该子问题以及优先队列中的其他节点, 因为它们不可能得出比已知最优解更好的解。在这种情况下, 进程应该重新初始化 (排空) 其优先队列。

通过上述修改, 进程 0 仍然使用同样的检查来确定并行算法应该何时停止。如果进程 0 在收到白色令牌的时候是白色的, 并且如果令牌的计数与进程 0 的消息计数之和为 0, 那么所有的计算工作都已停止。进程 0 可以发送一个终止消息到所有其他进程 (图 6.15 (c)), 所有进程调用 `MPI_Finalize` 和 `exit` 退出。

令牌在环上传递的目的有两个。其最初的目标是在进程间同步已知最优解的值。最终, 每个进程在发现自己无法找到更优解的时候将自己的优先队列排空。此时令牌的目标是进行分布终止检测: 保证所有进程是不活动的并且所有包含未解子问题的消息都已送达。

图 16.6 给出了采用最佳优先的并行分支定界算法的伪代码。

#### 常数

*Comm\_Interval* — 通信步之间的时间

*Termination* — 结束消息标志

*Token* — 令牌消息标志

*Unexamined\_Subproblem* — 包含待解子问题消息的标志

#### Functions:

*Current\_Time()* — 墙钟时间

*Delete\_Min()* — 从优先队列中删除具有最小下界的子问题

*First\_Element()* — 从优先队列中返回第一个元素, 但不从队列中删除它

*Initialize()* — 将优先队列的长度设为 0

*Insert()* — 在优先队列中插入子问题

*Is\_Empty()* — 如果优先队列为空, 返回“真”值

*Lower\_Bound()* — 返回待解子问题的下界

#### 变量

*color* — 进程颜色 (用于终止检测)

*global\_c* — 当前最优解的成本

*id* — 进程号

*initial* — 初始问题

*last\_comm* — 最后一次通信的时间

*local\_c* — 本进程目前最优解的成本

*local\_s* — 本进程目前找到的最优解

*msg\_count* — 发出的消息数减去收到的消息数

*q* — 优先队列

*token* — 在环上传送的用于终止检测的令牌

*u* — 状态空间树节点

*v* — 具有附加约束的新节点

#### Parallel Best-First Branch and Bound (minimization):

*Initialize (q)*

if *id* = 0 then

*Insert (q, initial)*

*token.c* ← ∞

*token.color* ← WHITE

```

    token.count ← 0
    Send token to successor process
endif
local_c ← ∞
best_soln ← ∞
last_comm ← Current_Time()
msg_count ← 0
color ← WHITE
repeat
    if Is_Empty(q) or (Current_Time() - last_comm > Comm_Interval) then
        BandB_Communication()
        last_comm ← Current_Time()
    else if not Is_Empty(q) then
        u ← Delete_Min(q)
        if Lower_Bound(u) < best_c then
            color ← BLACK
            if u is a solution then
                if Lower_Bound(u) < global_c then
                    local_s ← u
                    local_c ← Lower_Bound(local_s)
                endif
            else
                for i ← 1 to Possible_Constraints(u) do
                    Add constraint i to u, creating v
                    if Lower_Bound(v) < global_c then
                        Insert(q, v)
                    endif
                endfor
            endif
        endif
    endif
endrepeat
forever

BandBCommunication():
if there is a pending message with a Termination tag then Halt endif
if there is a pending message with a Token tag then
    Receive message containing token
    if local_c < token.c then
        token.c ← local_c
        token.s ← local_s
    endif
    if token.c ≤ Lower_Bound(First_Element(q)) then Initialize(q) endif
    global_c ← token.c
    if id = 0 then
        if (color = WHITE) and (token.color = WHITE) and
           (token.count + msg_count = 0) then
            Send messages with a Termination tag to all other processes
            Halt
        else
            token.color ← WHITE
            token.count ← 0
        endif
    else
        if color = BLACK then token.color ← BLACK
            token.count ← token.count + msg_count
        endif
        Send token to successor
        color ← WHITE
    endif
    while there are pending messages with tag Unexamined_Subproblem do
        Receive message with unexamined subproblem u
        msg_count ← msg_count - 1
        color ← BLACK
        if Lower_Bound(u) < global_c then Insert(q, u)
    endwhile
endwhile

```

```

endwhile
if there is more than one unexamined subproblem in  $q$  then
  Send unexamined subproblem to another process
   $msg\_count \leftarrow msg\_count + 1$ 
   $color \leftarrow BLACK$ 
endif
return

```

图 16.16 并行分支定界算法

## 16.7 搜索博弈树

进行两人零和 (zero-sum) 游戏, 如国际象棋、checkers 以及围棋的最成功的计算机程序都是基于穷举搜索法的。这些算法考虑到一系列移动和对方的移动, 评估得到的棋局的局势, 然后回溯达到最佳的最初移动。

### 16.7.1 最大最小算法

给定一个游戏, 最大最小算法可以用来决定最佳策略。图 16.17 (a) 给出了一个假想

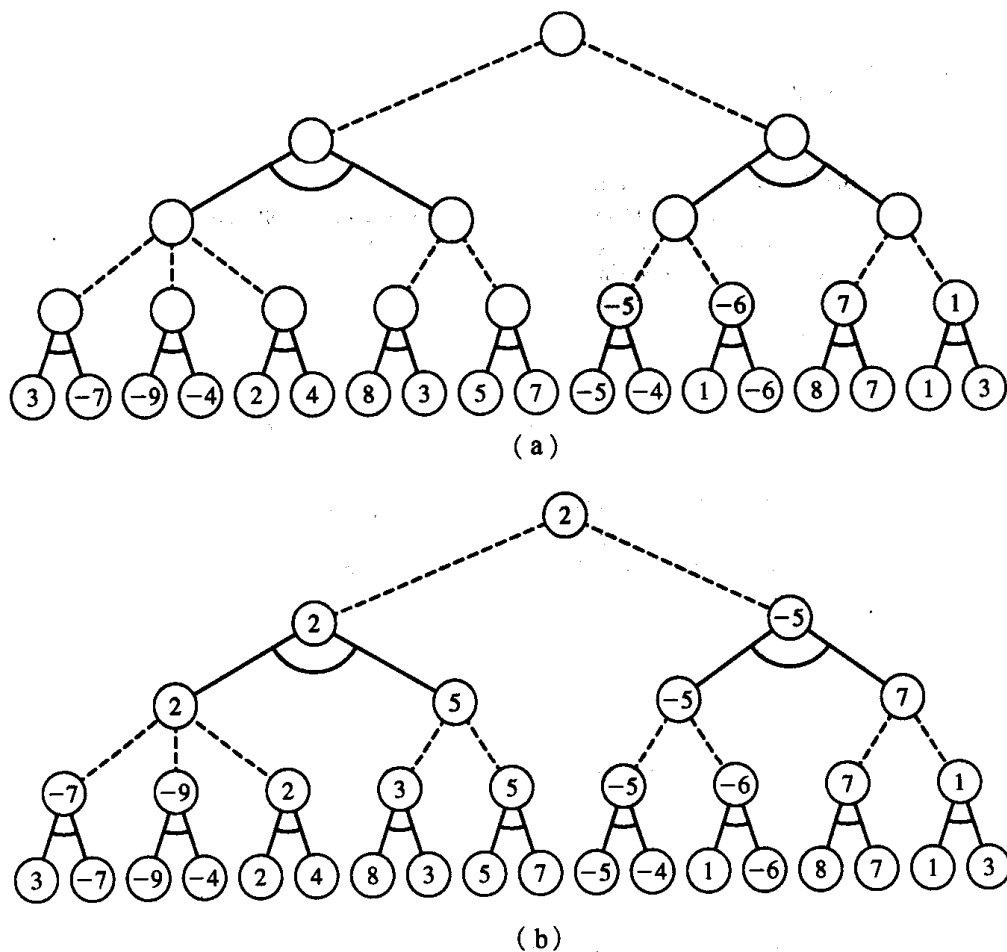


图 16.17 (a) 一颗博弈树, 虚线表示第一个玩家可进行的移动, 实线表示第二个玩家可进行的移动; (b) 填入了内部节点值的同一棵博弈树, 如果选择第一个可能的移动, 第一个玩家可以保证得到至少为 2 的结果。

的关于钱的游戏的博弈树。虚线代表第一个玩家的移动；实线代表第二个玩家的移动。树根表示游戏的初始状态。博弈树的树叶表示游戏的结果。内部节点表示中间状态。结果用第一个玩家所取得的优势来表示。因此正数表示第一个玩家赢得的钱数，负数表示他/她输掉的钱数。算法假定第二个玩家试图最小化第一个玩家的收获，而第一个玩家试图最大化自己的收获。图 16.17 (b) 是同样的树，但是填充了中间节点。对第一个游戏玩家来说这个游戏的价值是 2。如果第一个玩家采用最大最小策略，他/她保证可以至少获得 2 美元。

博弈树是 AND/OR 树的例子。先走的玩家会对树进行评估，AND 节点代表后走玩家开始移动的起始位置。先走的玩家必须考虑到对手所有可能的移动来保护自己。OR 节点代表该先走玩家移动了。先走的玩家如果已经找到了一个好的移动，就不需要考虑所有可能的移动了。

类似于国际象棋这样的复杂游戏其博弈树非常复杂，无法进行精确的评估。例如 de Groot 估计在国际象棋的博弈树中大约有  $38^{84}$  个位置【19】。因此当前的国际象棋程序仅能考察一定步数内的棋步，并估计每个棋盘位置的值。当然，评估函数是不精确的。如果存在精确的评估函数，就不需要进行搜索了。

正如我们所看到的，从一个位置到走几步之后所有可能的移动可以用一棵博弈树来表示。我们可以通过对树的叶子节点使用评估函数，然后向上回溯整个最大最小树上每个节点的评估函数值。如果是后走玩家的非终结节点，其评估值是其所有孩子中最小的。如果是先走玩家的移动，评估值为其所有孩子中最大的。给定一个博弈树，其中每个位置可以有  $b$  个合法的移动，很容易看出深度为  $d$  的博弈树的最大最小搜索需要考察  $b^d$  个叶子。

## 16.7.2 Alpha-Beta 剪枝

作为一个规律，搜索的深度越大，得到结果的质量就越好。这就是为什么 Alpha-Beta 剪枝是有价值的。Alpha-Beta 剪枝，也是一种分支定界法，避免了对不可能影响结果的子树的估值。因此它允许在同样的时间里进行更深的搜索。

图 16.18 给出了 Alpha-Beta 剪枝算法，拥有 4 个参数： $pos$ ，游戏的当前位置； $\alpha$  和  $\beta$ ，搜索的范围； $depth$ ，搜索的深度。函数返回位置  $pos$  的最大最小值。游戏最初的位置是 MAX-MODE，每个 MAX-NODE 的孩子是 MIN-NODE。每个 MIN-NODE 的孩子是 MAX-NODE。

为了说明 Alpha-Beta 算法，考虑图 16.19 中的博弈树。该博弈树与图 16.17 的游戏相同，不过不包括 Alpha-Beta 剪枝算法不搜索的节点。在算法开始时， $\alpha = -\infty$  并且  $\beta = \infty$ 。算法采用先序遍历博弈树的节点（即深度优先）； $\alpha$  和  $\beta$  在搜索过程中逐步收敛。

图 16.19 中用加重线画出的节点表示剪枝操作（消除子树搜索）的发生地。为了考察剪枝发生的条件，让我们看看搜索树的任意内部节点。当搜索达到该节点时，我们知道对先走的玩家经过某些移动步骤后得到的评估值至少为  $\alpha$ 。我们还知道对手的正确应对可以保证先走的玩家得到评估值不超过  $\beta$ 。因此  $\alpha$  和  $\beta$  定义了搜索的窗口。

如果内部节点  $pos$  是 MAX-NODE，那么它对应先走玩家的移动。如果  $val$ ，位置  $pos$  的博弈树评估值比  $\alpha$  要大，那么  $\alpha$  变成  $val$ ，意味着已经为先走玩家找到了更好的移动。

类似地，如果内部节点  $pos$  是 MIN-MODE，那么它对应后走玩家的移动。如果  $val$  位



置  $pos$  的博弈树评估值比  $\beta$  要大, 那么  $\beta$  变成  $val$ , 意味着已经为后走玩家找到了更好的移动。

常数

$max.c$  — 最大可能移动数

Alpha\_Beta ( $pos, \alpha, \beta, depth$ ):

参数

$pos$  — 位置

$\alpha$  — 剪枝下界

$\beta$  — 剪枝上界

$depth$  — 搜索深度

变量

$c[1..max.c]$  —  $pos$  在游戏树中的孩子

$cutoff$  — 如果可以剪枝, 则取值为“真”

$i$  — 在合法移动中迭代

$val$  — 搜索的返回值

$width$  — 合法移动数

```
begin
  if  $depth \leq 0$  then
    return (Evaluate( $pos$ )) {Evaluate terminal node}
  endif
   $width \leftarrow$  Generate_Moves( $pos$ )
  if  $width = 0$  then
    return (Evaluate( $pos$ )) {No legal moves}
  endif
   $cutoff \leftarrow$  FALSE
   $i \leftarrow 1$ 
  while ( $i \leq width$ ) and ( $cutoff =$  FALSE) do
     $val \leftarrow$  Alpha_Beta( $c[i], \alpha, \beta, depth-1$ )
    if Max_Node( $pos$ ) and  $val > \alpha$  then
       $\alpha \leftarrow val$ 
    elseif Min_Node( $pos$ ) and  $val < \beta$  then
       $\beta \leftarrow val$ 
    endif
    if  $\alpha > \beta$  then
       $cutoff \leftarrow$  TRUE
    endif
     $i \leftarrow i + 1$ 
  endwhile
  if Max_Node( $pos$ ) then return  $\alpha$ 
  else return  $\beta$ 
endif
end
```

图 16.18 串行 Alpha-Beta 剪枝算法

但是如果在任何时候我们发现  $\alpha$  超过了  $\beta$  的值, 那么就没有必要再搜索下去, 因为游戏者会阻止游戏走到当前考虑的位置。

例如, 考虑图 16.19 中标号为 A 的节点。搜索其第一个孩子返回的值为 3, 比  $\beta$  的值 2 还要大。对后走玩家来说, 到达这个位置毫无好处, 因为有另一条游戏路径可以保证返回值不大于 2。因此没有必要再搜索这个位置。

Alpha-Beta 剪枝到底再多程度上减少了对叶子节点的搜索? 算法对于完全排序博弈数具有最好的修剪效果, 即在每个位置总是优先考察最好的移动, 如图 16.20 所示。对一个具有深度  $k$ , 分支因子  $b$  的完全排序博弈树, Slagle 和 Dixon【103】已经证明 Alpha-Beta

算法所检查的叶子节点个数为：

$$Opt(b, d) = b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$$

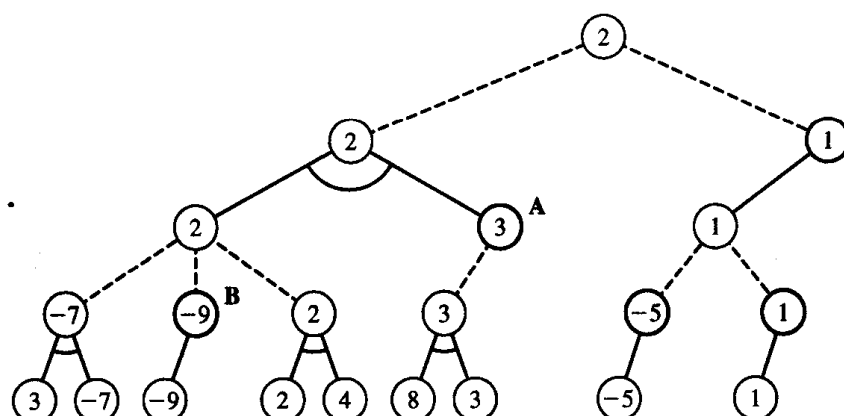


图 16.19 Alpha-Beta 剪枝的图示。每个节点内部的数字是该位置的评估值。对于叶子节点，评估函数计算出该点值。对于内部节点，值是通过其孩子节点的值计算出来的。突出显示的圆圈代表了进行剪枝的节点。请注意与最大最小搜索（图 16.17）相比，减少搜索的节点个数

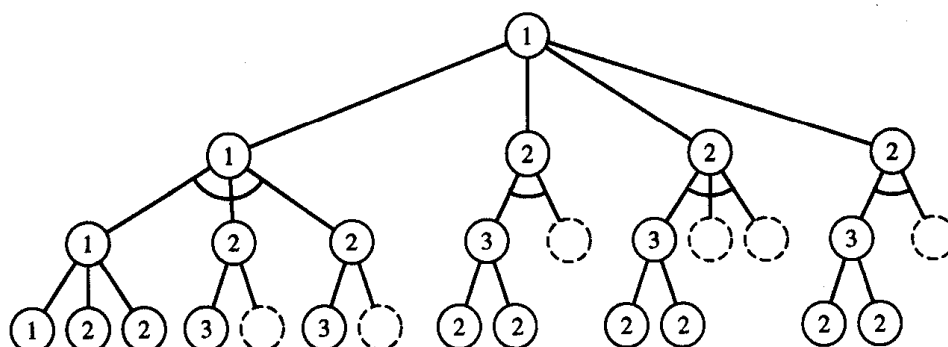


图 16.20 完全排序游戏树的 Alpha-Beta 剪枝。节点内的数字表示其分类：类型 1、类型 2 或类型 3。树的根是类型 1 的节点。类型 1 节点的第一个孩子是类型 1 节点。所有类型 1 节点的其他孩子是类型 2 节点。类型 2 节点的第 1 个孩子是类型 3 节点；所有类型 2 节点其他孩子都被剪枝掉。所有类型 3 节点的孩子是类型 2 节点

换句话说，在最好情况下 Alpha-Beta 剪枝法搜索的节点数不超过最大最小算法所搜索节点数的平方根的两倍。

对深度为  $d$  的博弈树的搜索算法的有效分支因子是算法评估过的叶子节点的  $d$  次方根。在搜索完全排序博弈树时，Alpha-Beta 搜索将有效分支因子从  $b$  降低到了  $\sqrt[d]{b}$ 。换句话说，在最好的情况下 Alpha-Beta 剪枝在同样时间内可以比最大最小算法搜索深一倍的博弈树。实验证明，在博弈树是完全排序的情况下，Alpha-Beta 算法通常搜索不超过 50% 的总节点。因此在实际应用中，Alpha-Beta 剪枝法展现了比最大最小算法高得多的性能。

### 16.7.3 Alpha-Beta 剪枝法的改进

对 Alpha-Beta 剪枝法的两个常见的改进是渴望搜索 (Aspiration Search) 和逐层加深 (Iterative Deepening)。渴望搜索是估计在博弈树根节点的评估函数值  $v$ ，并计算该评估值的可能误差  $e$ ，并把初始窗口设为  $(v-e, v+e)$  进行 Alpha-Beta 剪枝搜索。如果博弈树的值

确实在  $(v-e, v+e)$  中, 那么算法将会比使用  $(-\infty, \infty)$  作为初始窗口时更快地结束。如果博弈树的值比  $v-e$  小, 搜索的结果将返回  $v-e$ , 我们必须使用  $(-\infty, v-e)$  作为初始窗口再次调用 Alpha-Beta 剪枝搜索。类似地, 如果博弈数的值大于  $v+e$ , 搜索返回  $v+e$ , 我们必须再次进行搜索, 使用  $(v+e, \infty)$ 。

另一个标准 Alpha-Beta 算法的变种称做逐层加深。博弈树的每一层都称做 ply, 对应于一个玩家的移动。渴望搜索使用  $(d-1)$  层的搜索来为  $d$  层搜索做准备。这个技术具有 3 个优点。首先, 该技术允许控制搜索所需的时间, 搜索可以向更深的层次推进, 直到预先分配的时间用完。其次, 第  $(d-1)$  层搜索的结果可以用来改进第  $d$  层节点的排序, 使得节点顺序更接近于完美排序, 从而增加剪枝的效果。最后, 第  $(d-1)$  层搜索的结果可以用于作为第  $d$  层搜索的窗口的中心。

## 16.8 并行 Alpha-Beta 搜索

Alpha-Beta 搜索有很多可以进行并行执行的操作。一种方法是对位置估值和移动产生进行并行化, 例如专用的国际象棋机 HITECH, 它具有 64 个处理器, 按  $8 \times 8$  阵列排列。但是这种方法所能达到的加速比受到上面这些操作内的并行性的限制。

要进一步改进加速比就需要对搜索过程进行并行化。这种方法被开发 IBM Deep Blue (深蓝计算机) 的团队所采用。IBM Deep Blue 是一个 32 个节点的 RS/6000 多计算机, 并包含 192 个 VLSI 国际象棋处理器, 具有每秒搜索 1 亿个位置的能力, 并在 1997 年与国际象棋世界冠军 Gary Kasparov 的 6 盘棋的比赛中以  $3 \frac{1}{2}$  比  $2 \frac{1}{2}$  取胜。

### 16.8.1 并行渴望搜索

对 Alpha-Beta 剪枝搜索的一个直接的并行化方法就是进行并行的渴望搜索。如果有 3 个处理器可用, 那么可以给每个处理器分配下面的窗口之一:  $(-\infty, v-e)$ ,  $(v-e, v+e)$ ,  $(v+e, \infty)$ 。在理想的情况下搜索  $(v-e, v+e)$  的处理器会得到结果, 但是在任何情况下并行搜索的速度都比单个处理器搜索  $(-\infty, \infty)$  要快。我们还可以通过搜索更加狭窄的窗口来使用更多的处理器。

通过对国际象棋游戏进行并行渴望搜索实验, 我们得到两个结论。第一, 不管有多少处理器可用, 预期的最大加速比通常为 5 或 6。这是因为  $Opt(b, d)$  是 Alpha-Beta 搜索的开销的下界, 即使当  $\alpha$  和  $\beta$  的最初设置与搜索的返回值相同时也是如此。第二, 并行渴望搜索在使用 2 或 3 个处理器的时候有时会出现超线性加速比。

### 16.8.2 并行子树估值

另一种方法是让各个处理器并行搜索不同的子树。当采用这种方法的时候, 我们必须考虑两种开销。搜索开销是指由于引入并行性所导致的搜索节点个数的增加。通信开销是指用于协调各个进程进行搜索所需的开销。通过让每个进程都知道当前的搜索窗口  $(\alpha, \beta)$ ,

可以将搜索开销转化为通信开销。通过允许各个处理器搜索过时的搜索窗口，可以将通信开销转化为搜索开销。

例如，考虑下面的并行 Alpha-Beta 搜索方法。从树根处划分博弈树，并给每个处理器分配相同大小的子树。让每个处理器在其子树上执行 Alpha-Beta 搜索。每个处理器都以  $(-\infty, \infty)$  为初始窗口搜索，并且处理器不通知其他处理器自己的搜索窗口的改变。显然这种方法使得通信开销降到了最低。这种方法能达到的加速比是多少？

Slagle 和 Dixon 的研究表明，在深度为  $d$ 、分支因子为  $b$  的完全排序的博弈树中，Alpha-Beta 搜索所考察的节点个数为  $Opt(b, d)$ 。我们使用这个公式来确定对该博弈树的第一个子树的搜索所需考察的节点个数为  $Opt(b, d-1)$ 。

上面的结果意味着考察完全排序博弈树的第一个子树所需要的时间是不成比例的。例如，对于一个分支因子为 38 的完全排序博弈树进行 10 层搜索，所需搜索的最少节点个数为 158 470 335。对于其第一个子树来说，所需搜索的最少节点个数为 81 320 303。根据 Amdahl 定律，如果只有一个处理器来对一个子树进行搜索，我们得到的加速比会小于 2。

另外，由于每个处理器的搜索的初始窗口为  $(-\infty, \infty)$ ，并行算法能够剪枝掉的子树没有串行算法多。完全消除了通信开销的后果是产生了显著的搜索开销。

让我们来看看另一个极端。如何才能完全消除搜索开销？我们假定博弈树是完全排序的，如图 16.20 所示。如果我们希望消除搜索开销，则必须保证并行算法能够像串行算法一样剪枝掉同样的节点。首先考虑搜索类型 1 节点的子树。类型 1 节点的第一个孩子是类型 1 节点；剩余的孩子是类型 2 节点。搜索类型 2 节点的子树，需要得到更新的  $\alpha$  和  $\beta$  的值，以便剪枝掉除了第一个子树外的其他所有子树。为了得到更新的值，对类型 2 节点的搜索必须在对类型 1 节点子树的搜索完成并返回  $\alpha$  和  $\beta$  值后才能开始。但是当  $\alpha$  和  $\beta$  的值更新以后，所有类型 2 节点可以被同时搜索而无需处理器间通信。

在实际情况下，搜索树并不是完全排序的，但是上面的研究已经表明如果能够在得到更加准确的界限信息后再开始某些节点的搜索，可以显著减少并行 Alpha-Beta 剪枝算法的搜索开销。这实际是我们下一个算法的基础。

### 16.8.3 分布式树搜索

Ferguson 和 Korf【26】开发了一个并行树搜索算法称做分布式树搜索 (Distributed Tree Search, DTS)。该算法在博弈树的评估中可以获得很好的加速比。DTS 算法可以用于解决很多树搜索问题，我们将主要描述如何将使用算法进行并行 Alpha-Beta 搜索。

DTS 的执行方式是将进程分配给搜索树中的节点。每个进程控制 1 个或多个物理处理器。当算法开始执行的时候，一个称做根进程的进程，被分配到搜索树的根节点上。该进程控制所有的物理处理器进行搜索。

如果一个进程被分配到一个非终端节点上，它通过对合法移动进行估值来产生该节点的孩子节点。该进程根据一定的处理器分配策略将处理器指定给孩子节点。所采用的分支定界策略与上一节最后我们讨论的算法非常相似。当搜索一个类型 1 节点的时候，所有处理器都被指定到其最左孩子上。在从以最左孩子为根的子树搜索中返回更新的定界值后，处理器按照广度优先的原则分配给该类型 1 节点的其他孩子节点。此时，每个孩子节点都

分配到了一个进程, 并至少分配到一个处理器。父进程的操作挂起, 直到接收到其他进程发来的消息 (从其父节点或是子节点)。

当一个进程被分配到一个终端节点上时, 它将该节点的值和分配给该节点的处理器返回给父进程, 然后就结束自己。

第一个完成了对分配到的子树搜索任务的子进程会发一个包含了搜索结果值和  $\alpha$ 、 $\beta$  的值给父进程。同时该子进程还将自己所用的处理器集合返回给父进程, 并退出。父进程在收到子进程发来的消息后被唤醒。它重新将被释放的处理器分配给仍在搜索的其他子进程。父进程还会将更新的  $\alpha$  和  $\beta$  值发送给其子进程。在快的进程和慢的进程之间进行的处理器重新分配会得到很好的负载平衡结果。注意在这个方案中子进程被父进程唤醒, 并得到附加的处理器。在重新分配了处理器后, 父进程挂起直到接收到另一个消息。当所有子进程都结束后, 父进程返回  $\alpha$ 、 $\beta$  和处理器集给它的父进程并结束。当根进程结束时, 算法执行完毕。

有 3 个实现上的细节可以提高 DTS 算法的性能。首先, 每个被阻塞的进程应该与它的一个子进程共享一个物理处理器。用这种方式所有处理器都处于工作状态。其次, 当父进程被唤醒时, 它应该比在搜索树上更深的节点上工作的进程具有更高的优先级。第三, 如果搜索中遇到只有一个处理器分配给一个节点的时候, 则控制该处理器的进程应该执行标准的串行 Alpha-Beta 搜索算法。

给定一个具有分支因子  $b$  的博弈树, 如果 Alpha-Beta 算法搜索的有效分支因子为  $b^x$  ( $0.5 \leq x \leq 1$ ), 那么具有  $p$  个处理器的 DTS 算法在使用广度优先策略的时候将达到加速比  $O(p^x)$ 。

为了测试 DTS 算法, Ferguson 和 Korf【26】实现了游戏 Othello。其节点排序函数达到了约  $b^{0.66}$  的有效分支因子。该程序用 DTS 算法实现了并行 Alpha-Beta 搜索。通过在第一代小型计算机上执行该程序, 在 32 个处理器上得到的加速比为 12。

## 16.9 本章小结

组合搜索算法被用于在有限离散的数学结构上为各种决策和优化问题寻求答案。对组合搜索问题的一种区分方法是按照其搜索的状态树空间进行分类。分治法在 AND 树上进行遍历; 只有当所有孩子的解都找到时, 问题或子问题的解才被找到。回溯法和分支定界法在 OR 树上进行遍历; 不需要搜索每个子问题就可以得到问题的解。两人对弈游戏可以使用 AND/OR 树来表示, 它结合了 AND 节点和 OR 节点。

并行分治法的一种方法是让一个进程负责将问题 (或子问题) 分解成小片, 并合并子问题的解。使用这种方式能够得到加速比受到传播和合并开销的限制。相反地, 如果原问题和最终解都被分解到各个处理器上, 那么并行分治法的效率可以大大提高。但是, 在这种情况下保持处理器间的负载平衡是一个困难的问题。

回溯法采用深度优先策略来搜索状态空间树, 可以用来得到问题的一个解或所有解。但是回溯法没有利用与问题相关的知识来避免搜索无解的子树。该算法的优点是仅需要与搜索深度成线性关系的存储空间。由于状态空间树通常是不平衡的, 并行回溯算法的主要

挑战是让每个进程承担同样的工作量。我们讨论将给每个进程分配多个子树的策略, 这增大了每个进程搜索数目近似相等的节点的概率。

为了保障并行回溯算法在结束的时候不发生运行时错误, 需要进行分布终止探测。Dijkstra 等人的算法允许在  $\Theta(p)$  时间内完成这一工作。

串行的分支定界算法比深度优先或广度优先等穷举法能够以快得多的速度找到组合优化问题的解, 因为分支定界法跳过了对不可能产生问题的解的子树的搜索。但是, 由于分支定界法所实际搜索的状态空间树的形状是不规则的, 因此很难将处理器分配给子树并获得平衡的负载。并行分支定界算法设计者所面临的基本问题是让并行算法集中搜索串行算法所搜索的节点以保证处理器的高效率。

Alpha-beta 剪枝算法是用于评估博弈树的算法。在最好情况下, 该算法可以让计算机搜索的深度比使用最大最小算法多 1 倍。通过使用渴望搜索算法和逐层加深, 算法的性能可以进一步得到提高。我们考察了若干种并行化 Alpha-Beta 搜索的方法: 并行的移动生成和评估, 并行渴望搜索, 以及对独立子树的并行搜索。如果要将并行算法扩展到 MPP 计算机上, 只有第三种方法看起来能够提供足够的并行性。最小化通信开销会造成无法接受的搜索开销, 反之亦然。分布式树搜索 (DTS) 算法将处理器分配到搜索上, 并使得通信开销和搜索开销都保持在合理的水平上。

## 16.10 主要术语

Alpha-Beta pruning

AND tree, AND/OR tree, OR tree

aspiration search

backtrack

bound-and-branch strategy

branch-and-bound search

combinatorial algorithm

combinatorial search

communication overhead

decision problem

distributed termination detection problem

divide and conquer

effective branching factor

game tree

iterative deepening

minimax

optimization problem

perfectly ordered game tree

ply

Alpha-Beta 剪枝

AND 树、AND/OR 树、OR 树

渴望搜索

回溯

分支定界策略

分支定界搜索

组合算法

组合搜索

通信开销

决策问题

分布终止检测问题

分治法

有效分支因子

博弈树

迭代深化

最大最小

优化问题

完全排序博弈树

层

pruning  
search overhead  
state space tree

剪枝  
搜索开销  
状态空间树

## 16.11 参 考 文 献

Andrews【3】和 Grama 等人【44】提出了其他分布终止检测策略。

Lai 和 Sahni【63】以及 Quinn 和 Deo【96】描述了在分支定界法可以得到超线性加速比（使用  $p$  个处理器的时候加速比超过  $p$ ）的条件。

Marsland 和 Campbell【81】发表了在强排序博弈树上进行并行搜索的早期文献综述。该文比本章的相关内容要详细得多，并描述了另外一些有趣的并行 Alpha-Beta 搜索的变种。

Huntbach 和 Burton【55】描述了在虚拟树计算机上如何实现并行 Alpha-Beta 搜索。Schaeffer【100】也描述了在计算机网络上实现并行 Alpha-Beta 搜索的方法。

Bhattacharya 和 Bagchi【10】描述了 SSS\* 搜索算法的并行化。他们的工作是基于本章描述的 DTS 算法的。

Felten 和 Otto【25】在 nCUBE 3200 多计算机上实现了并行国际象棋程序，在 256 个处理器上得到的加速比为 100。他们的程序与 DTS 非常相似，但是包括了其他的改进。

Hsu 写了一本书，记录了 IBM 的计算机科学家们研制的 Deep Blue 计算机的有关情况。该并行国际象棋计算机在 1997 年击败了 Gary Kasparov【54】。

## 16.12 练 习 题

16.1 给定一个分治法，其复杂度由下面的递归关系表示：

$$T(n) = \Theta(n) + kT\left(\frac{n}{k}\right)$$

$$T(1) = \Theta(1)$$

请推导并行分支定界算法的复杂度下界，假设问题分解步骤（复杂度为  $\Theta(n)$ ）不能被并行化。

16.2 第 16.3.1 节提出了在纵横字谜游戏中填充不完整词时确定顺序的一种方法：选择至少有一个已填充字母的最长不完整词。请提出一个更好的排序方法。你的排序方法会怎样对图 16.2 (a) 中的词进行排序？解释为什么你的方法很可能会产生一个具有较少节点的搜索空间树。

16.3 以 8-puzzle 为例，解释为什么在问题空间树上进行简单地回溯搜索很可能得不到解。提出一个修改方案以保证得到解。

16.4 在图 16.12 的状态空间树中，解释为什么根节点的右孩子的下界值为 7。

16.5 图 16.12 演示了使用分支定界搜索解 8-puzzle 问题的过程。另一种策略是使用广度优先搜索。假设子问题的顺序不变，使用广度优先法找到问题的解需要考察多少个节点？

16.6 选取具有相同最小下界的待解子问题来验证最佳优先分支定界搜索所使用的深度启发式规则。

16.7 考虑下面并行执行最佳优先分支定界算法的方法：平均分配根节点的孩子给各个进程。如果没有足够多的孩子，将根节点的孙子节点或曾孙节点平均分配到进程上。每个进程在所分配到的子树上执行最佳优先分支定界算法。当一个进程找到解后，将结果的值广播给其他进程。当所有进程都没有能够导致更优解的待解子问题的时候，整个算法结束。

与本书中所使用的算法比较，解释这个算法的优点和缺点。这个算法是否能够达到更好的性能？解释你的理由。

16.8 如果存在完美的估值函数，就不需要对博弈树进行搜索。解释你的理由。

16.9 使用最大最小算法对图 16.21 所示的博弈树进行评估。

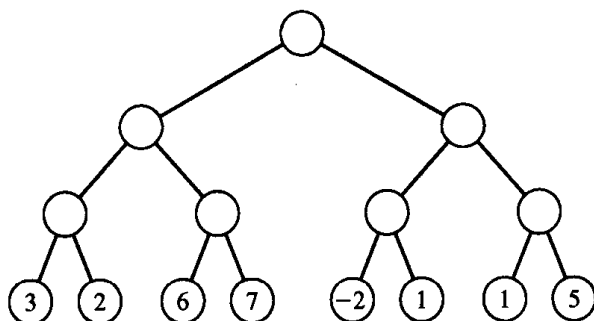


图 16.21 一棵博弈树

16.10 解释为什么 Alpha-Beta 算法在图 16.19 的博弈树中对标号为 B 的节点进行剪枝。

16.11 使用 Alpha-Beta 算法对图 16.21 中的博弈树进行评估。

16.12 将图 16.20 中的完全排序博弈树扩展一层，并表示第 4 层的节点是如何被剪枝的。假定分支因子为 2。

16.13 解释为什么说 Alpha-Beta 搜索只是 DTS 的一个特例。

16.14 改进 Alpha-Beta 算法的剪枝效率对 DTS 算法在应用中的加速比会有什么影响？

16.15  $N$  皇后问题是在  $N \times N$  的国际象棋棋盘上放置  $N$  个皇后，使得皇后之间不互相攻击。图 16.22 给出了 4 皇后问题的一个解。写一个并行程序计算  $N$  皇后问题的解的个数， $N$  由命令行输入。测试不同的  $N$  和  $p$  的值，并画出作为  $N$  和  $p$  的函数的程序加速比曲线。

16.16 写一个并行程序，对从命令行输入的  $N$ ，给出  $N$  皇后问题的一个解，在打印出这个解后程序就结束。测试不同的  $N$  和  $p$  的值，并画出作为  $N$  和  $p$  的函数的程序加速比曲线。

16.17 图 16.23 给出了一个游戏棋盘。这个棋盘包含 21 个洞。棋局开始的时候，除了中间的洞外（用黑色表示），

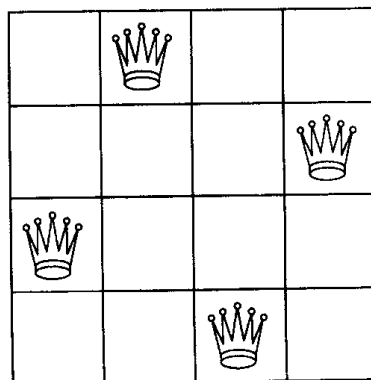


图 16.22 4 皇后问题的一个解



每个洞中都有一个桩子。桩子可以像跳棋一样移动和被吃，即一个桩子可以在直线上越过另一个桩子移动到一个空着的洞中，被越过的桩子被吃掉，从而被从棋盘上取走。跳的方向包括水平方向、垂直方向和对角方向。

游戏的目标是不断去除桩子，直到只在棋盘中间剩下一个桩子。对于有 20 个桩子的初始情况，需要 19 次移动来使得棋盘上只剩下一个桩子。

写一个并行程序找出上述游戏的一个解。对不同的  $p$  测试你的程序，并画出作为  $p$  的函数的程序加速比。

16.18 Sam Loyd 发明的 15-puzzle 游戏是本章介绍的 8-puzzle 问题的一个更大的版本。15 块瓷砖，编号为 1 到 15，和一个洞在一个  $4 \times 4$  的网格上排列。写一个并行程序，其输入为任意混乱的 15-puzzle 棋局排列，程序应找到将瓷砖重新排序所需的最小移动序列。对至少 5 个棋局测试你的程序，每个棋局都应该至少需要 6 次移动才能完成。对每个棋局，画出作为  $p$  的函数的加速比曲线。

16.19 写一个程序能够和人对弈 Othello（也称作 Reversi）游戏。

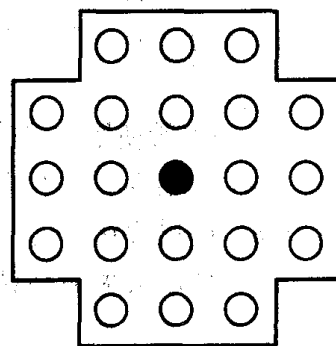


图 16.23 练习 16.17 图示

# 第 17 章 共享存储编程

not what we give, but what we share—

For the gift without the giver is bare;

Who gives himself with alms feeds three—

Himself, his hungering neighbor, and me.

**James Russell Lowell, The Vision of Sir Launfal**

## 17.1 概 述

在 20 世纪 80 年代，拥有少量处理器的商用多处理机便需要花费数十万美元。如今，尽管那些拥有庞大数目处理器的多处理机仍然非常昂贵，但是小规模系统的价格已经降低了。Dell, Gateway 以及其他一些公司以低于五千美元的价格出售双处理器计算机，你还可以以低于两万美元的价格买到拥有四个处理器的多处理机。

你可以用 MPI 在多处理机上编写并程序，然而你还可以用专门为共享存储环境量身打造的编程语言来得到更优的性能。OpenMP 是作为共享存储标准而问世的。它是为在多处理机上编写并程序而设计的一个应用编程接口。它包括一套编译指导语句和一个用来支持它的函数库。OpenMP 是通过与标准 Fortran, C 和 C++ 结合来工作的。

本章介绍如何用 OpenMP 在共享存储环境下编写并程序。你可能有两种具体情况。也许你所拥有的并行计算机只是一台多处理机。在这种情况下，你也许应更倾向于使用 OpenMP 来编写程序，而不是使用 MPI。

另一种情况，你可能有一台包括许多节点的多计算机，其中每一个节点都是一个多个处理机。这是一种相当流行的组建拥有上百甚至上千个处理器的大型多计算机的方法。比如（2002 年左右）：

- IBM 的 RS/6000 SP 系统包括 512 个节点。每一个节点最多可有 16 个处理器。
- 富士通的 AP3000 系列超级计算机包括 1024 个节点，每个节点有一至两个 UltraSPARC 处理器。
- Dell 的高性能计算机群包括 64 个节点，每个节点都是一个拥有两个 Pentium III 处理器的多处理机。

在本章中你将领会到共享存储编程模式和消息传递模式有何不同，并且学到足够多的 OpenMP 编译指导语句以及函数调用从而能够对各种不同的 C 代码段并行化。

本章将介绍一套功能强大的 OpenMP 编译指导语句：

- parallel, 用在一个代码段之前，指示这段代码将被多个线程并行执行
- for, 用在一个 for 循环之前，每个循环之间必须无相关性，从而可以被分到不同的线程中并行执行

- parallel for, 是 parallel 和 for 这两个编译指导语句的结合
- sections, 出现在一系列可能会被并行执行的代码段之前
- parallel sections, 是 parallel 和 sections 这两个编译指导语句的结合
- critical, 用在一段代码临界区之前
- single, 出现在一段将只被单个线程执行的代码段之前

你还会接触到 4 个重要的 OpenMP 函数:

- omp\_get\_num\_procs, 返回运行本线程的多处理机的处理器个数
- omp\_get\_num\_threads, 返回当前并行区中活动的线程个数
- omp\_get\_thread\_num, 返回线程号
- omp\_set\_num\_threads, 修改并行执行代码时线程的个数

## 17.2 共享存储模型

共享存储模型 (如图 17.1 所示) 是对在第 2.4 节中介绍的一般的集中式多处理机的抽象。其底层硬件为一系列处理器, 这些处理器都访问同一个共享存储器。由于所有处理器可以访问内存中的同一个位置, 因而它们可以通过共享变量进行交互和同步。

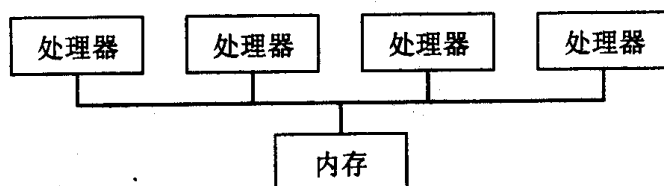


图 17.1 并行计算的共享存储模型示意图。各个处理器之间通过共享变量来实现交互与同步

在共享存储的并行程序中, 标准的并行模式为 fork/join 式并行。当程序开始执行的时候只有一个叫做主线程的线程存在, 如图 17.2 所示。主线程执行算法的顺序部分。当遇到

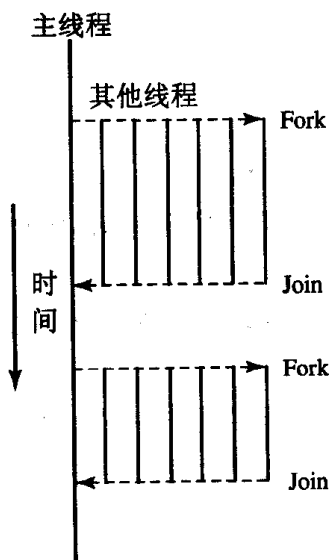


图 17.2 在共享存储的并行程序中, 标准的并行模式为 fork/join 式并行。当程序开始执行的时候只有一个叫做主线程的线程存在。主线程执行程序串行部分。它通过派生出其他的线程来执行其他的并行部分。当重新执行程序串行部分时, 这些线程将终止

需要进行并行运算时，主线程派生出（创建或者唤醒）一些附加线程。在并行区域内，主线程和这些派生的线程协同工作。在并行代码段结束时，派生的线程退出或者挂起，同时控制流回到单独的主线程手中，我们称为会合。

共享存储模型和消息传递模型的一个关键区别在于消息传递模型中的所有进程存活于整个程序的执行过程中，而在共享存储模型中，在程序的开始和结束时存活着的线程数均为一，而在整个程序执行过程中线程数会动态变化。

同时我们也可以将顺序执行的程序看作是共享存储模型程序的一种特殊情况，即：没有 `fork/join` 的形式。无论是仅仅在一个循环并行执行中使用了一个 `fork/join` 的并行程序，还是那些大部分代码段都被并行执行的程序，它们都属于共享存储模型的并行程序。因此，共享存储模型支持增量并行化，就是说一次操作并行化程序中的一段代码，我们进行多次这样的操作从而可以将整个顺序程序转化为并行程序。

支持增量并行化是共享存储模型相对于消息传递模型而言最大的优势。我们可以分析得到顺序程序的轮廓，按照时间开销对程序的各个模块进行排序，然后从最费时的模块开始逐个进行分析，接着对适于并行执行的模块进行并行化，直到我们不能再获得更高的性能为止。

相反，对于消息传递模型程序来说，它们没有共享存储器来存放变量，且并行进程在整个程序的执行过程中存活。由顺序程序到并行程序的转化不可能是增量式的，因为这个转变的跨度非常大，而不是通过很多步较小的努力能完成的。

在本章中你将看到一些非常复杂的顺序代码模块并学习如何将它们转化成为并行代码区域。

## 17.3 对 for 循环的并行化

在 C 程序中固有的并行操作常常以 `for` 循环的形式表现。OpenMP 可以方便对其进行指示一个 `for` 循环的迭代可以被并行地执行。比如，参考下面的循环，它在用 MPI 实现的 Eratosthenes 筛法程序中占据了很大份额的执行时间：

```
for (i=first; i<size; i+=prime) marked[i]=1
```

很明显，在这个循环的每次迭代之间不存在相关性。我们应该怎样将它转化为并行循环呢？在 OpenMP 中，我们只需简单的告诉编译器一个 `for` 循环可以被并行执行；编译器会负责生成派生和会合线程以及调度并行迭代的代码，并将循环的迭代分配给线程。

### 17.3.1 parallel for 编译指导语句

C 或 C++ 语言编译器指导语句在英语中记作 `pragma`。`pragma` 这个词是“`pragmatic information`”的简写形式。编译指导语句起着与编译器交互信息的作用。编译指导语句提供的信息不是必需的，因此编译器可以在忽略它的情况下仍然生成正确的目标程序。但是这些信息无疑可以辅助编译器进行程序优化。

就像为预处理器提供信息的其他代码行一样, 编译指导语句以#开头。编译指导语句在 C 或 C++ 程序中的文法如下:

```
#pragma omp <rest of pragma>
```

我们要学习的第一个编译指导语句是 `parallel for`。其最简单的形式为:

```
#pragma omp parallel for
```

当这一行下面紧跟着一个 `for` 循环的时候, 它将指示编译器将 `for` 循环并行化:

```
#pragma omp parallel for
    for (i=first; i<size; i+=prime) marked[i]=1
```

为了使编译器能够成功的将顺序执行的循环转化为并行执行, 在分析控制子句时运行系统必须能够得到所需信息以确定循环迭代的次数。因此 `for` 循环的控制子句必须具备规范格式, 如图 17.3 所示。并且, `for` 循环中不能包含允许循环提前退出的语句。比如, 语句 `break`、`return`、`exit`、`goto` 以及此类循环外的标记。然而这里允许语句 `continue` 的存在, 因为它的执行不会影响到迭代的次数。

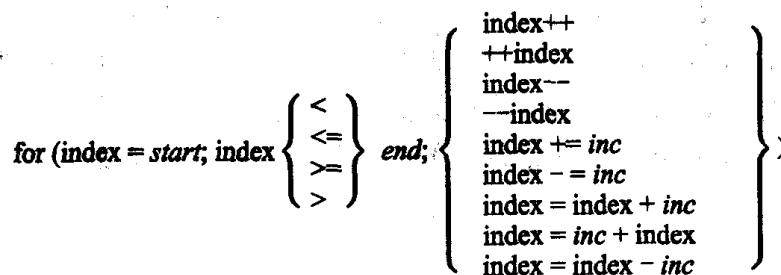


图 17.3 `for` 循环必须具有规范格式

如下所示 `for` 循环:

```
for (i=first; i<size; i+=prime) marked[i]=1
```

它遵循了前述标准: 控制子句具备规范格式并且在循环体中不存在提前退出。于是编译器可以生成让循环迭代并行执行的代码。

在 `for` 循环并行执行的过程中, 主线程创建若干派生线程, 所有这些线程协同工作共同完成循环的所有迭代。每个线程有各自的执行现场, 也就是上下文: 一个囊括所有这个线程将访问的变量的地址空间。执行现场包括静态变量, 堆中动态分配的数据结构以及运行时堆栈中的变量。

执行现场包括线程本身的运行时堆栈, 这个堆栈保存着调用函数的框架信息。其他变量或者是共享的, 或者是私有的。共享变量在所有线程的执行现场中的地址都是相同的。所有线程都可以对共享变量进行访问。私有变量在各个线程的执行现场中的地址不同。一个线程可以访问它自己的私有变量, 但是不能访问其他线程的私有变量。

在 `parallel for` 编译指导语句中, 变量默认设置为共享, 而循环编号变量除外, 它是私有变量。

图 17.4 所示为共享变量和私有变量。在图示例子中 for 循环的迭代被分配到两个线程中进行。循环编号 *i* 为私有变量——每个线程拥有一份自己的副本。剩下的变量 *b* 和变量 *cptr*，以及堆中分配的数据，均为共享变量。

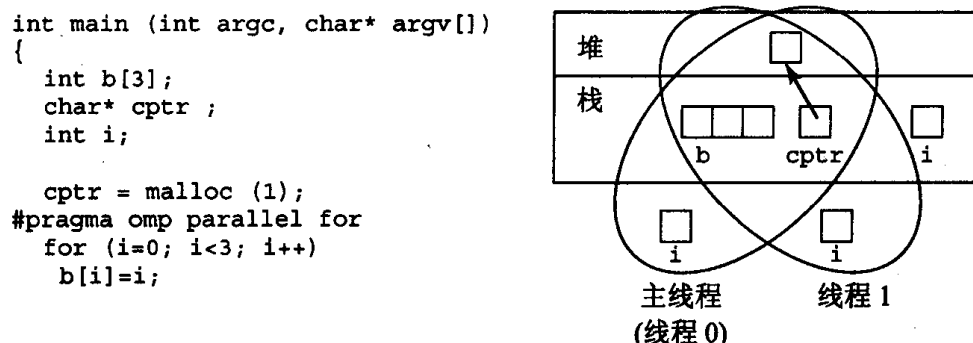


图 17.4 并行执行 for 循环时，编号 *i* 是私有变量，而 *b*、*cptr* 和堆数据被共享

运行时系统是如何获得所需创建的线程个数呢？环境变量 `OMP_NUM_THREADS` 的值提供了代码并行区的默认线程数。在 Unix 中可以用 `printenv` 命令查看这个变量的值，还可以用 `setenv` 命令改变此值。

另外一种策略是：将线程数设置为多处理器的 CPU 个数。为此，下面我们来认识一些 OpenMP 的函数。

### 17.3.2 omp\_get\_num\_procs 函数

函数 `omp_get_num_procs` 返回并行程序中的可用的物理处理器的个数。函数的声明如下：

```
int omp_get_num_procs (void)
```

这个函数返回一个整数，该数有可能小于多处理机的处理器的物理个数，这取决于运行时系统所允许的进程可进行的处理器访问。

### 17.3.3 omp\_set\_num\_threads 函数

函数 `omp_set_num_threads` 用参数值来设置代码并行区中活动的线程个数。函数的声明如下：

```
void omp_set_num_threads (int t)
```

因为这个函数可以在程序的多处调用，你可以根据并行的粒度大小或者代码段的其他特点来设置动态的并行度。

我们可以把当前可用的处理器数目直接作为所要设置的线程数目：

```
int t;
...
```

```
t=omp_get_num_procs ();  
omp_set_num_threads (t)
```

## 17.4 声明私有变量

在第二个例子中, 我们可以设计一个较为复杂的循环结构。下面就是 Floyd 算法的 MPI 实现中负责计算的核心部分:

```
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)  
    for (j = 0; j < n; j++)  
        a[i][j] = MIN(a[i][j], a[i][k] + tmp[j]);
```

在我们先前对该算法的分析中, 我们认为两个循环都可以并行地执行。那我们应当选择哪一个呢? 如果将内部循环并行化, 那么程序将在外重循环的每次迭代中都进行派生和会合操作。如果这样, 那么派生-会合操作的开销可能会远大于将内部的  $n$  重循环在多个线程之间并行化所节省的时间。反之, 如果我们对外层循环进行并行化, 将仅仅引入一次派生-会合操作的开销。

粒度是相邻通信过程或者同步操作间隔内进行的运算量。大致上讲, 增加粒度的大小可以改善并程序的性能。将外层循环进行并行化可以使程序有更大的粒度。因而我们将使用这种方式。

我们可以很简单地指导编译器对变量  $i$  索引的循环进行并行化。然而, 我们必须注意各个并行线程所访问的变量。按照默认情况, 所有的变量将被设定为共享的, 除了循环的索引变量  $i$ 。这样做可以使得线程之间相互通信较为简便, 但是也有可能引起一些问题。

让我们来仔细考察当多个线程对循环的各次迭代进行并行处理时的具体情形。我们本想使针对  $i$  值的每个线程遍历  $j$  的  $n$  个取值。但是, 所有的线程将都试图初始化共享变量  $j$  并对它进行递增操作。这就意味着线程将很有可能不能遍历所有  $n$  值对应的迭代。

试想这样的情形, 多个线程试图并行执行以  $i$  编号的不同的循环迭代。对于每个以  $i$  编号的外层循环迭代, 我们希望每个线程能做完以  $j$  编号的  $n$  个内层循环迭代。然而, 所有的线程都试图对同一个共享的变量  $j$  初始化和进行增量运算——于是很有可能某些线程不能完整地执行所有  $n$  个迭代。

解决办法很明显, 我们应当将  $j$  也设为私有变量。

### 17.4.1 private 子句

子句是编译指导语句可选的附加部分。private 子句指导编译器将一个或若干个变量私有化。语法如下:

```
private (<variable list>
```

指导语句告诉编译器为每个执行这条指导语句的代码段的线程分配一个这个变量的私有副本。如下例一个并行 for 循环。变量  $j$  的私有副本只有在 for 循环内部才能访问到。

在循环的入口和出口处此变量都是未定义的。

一个使用 `private` 子句的双层嵌套循环的 OpenMP 实现如下：

```
#pragma omp parallel for private(j)
  for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
    for (j = 0; j < n; j++)
      a[i][j] = MIN(a[i][j], a[i][k] + tmp[j]);
```

即使  $j$  在并行的 `for` 循环区之前已经被提前赋值了，也没有一个线程能够访问到那个值。同样的，无论在并行执行 `for` 循环时线程为  $j$  赋了什么值，共享的  $j$  的值不会受到影响。另一方面，当进入并行结构的时候一个私有变量的默认值为未定义，当并行结构结束的时候这个变量的值也是未定义。

由于削减了共享变量和与其对应的私有变量之间不必要的复制，私有变量的默认情况（在循环入口和出口设为未定义）将节省执行时间。

## 17.4.2 firstprivate 子句

有时我们希望一个私有变量继承共享变量的值。比如，下面的代码段：

```
x[0] = complex_function();
for (i = 0; i < n; i++) {
  for (j = 1; j < 4; j++)
    x[j] = g(i, x[j-1]);
  answer[i] = x[1] - x[3];
}
```

假设函数  $g$  没有副作用，当把  $x$  设为私有变量的时候，我们可以让每个外层循环的迭代被并行执行。然而， $x[0]$  在外层 `for` 循环之前已经被初始化了并且在内层循环的第一次迭代中被引用了。把对  $x[0]$  的初始化移到外层 `for` 循环里是不现实的，因为那样将非常费时。不过，我们可以让每个线程的数组元素  $x[0]$  的私有副本继承在主线程中所对应的共享变量被赋的值。

`firstprivate` 子句有以下语法：

```
firstprivate (<variable list>)
```

它将指导编译器在进入循环时创建各个私有变量，并使得它们与由主线程所控制的相应变量有相同的值。

下面是正确使用该子句的并行循环代码：

```
x[0] = complex_function();
#pragma omp parallel for private(j) firstprivate(x)
  for (i = 0; i < n; i++) {
    for (j = 1; j < 4; j++)
      x[j] = g(i, x[j-1]);
    answer[i] = x[1] - x[3];
  }
```



请注意, 在 `firstprivate` 变量表中, 各个变量的值将在线程创建时初始化, 而非进行每次迭代时均初始化。如果某个线程执行了该循环的多次迭代, 并对其中的某个变量进行了改动, 那么之后的各次迭代所得到的该变量的值将是改动后的值。

### 17.4.3 `lastprivate` 子句

循环的顺序末次迭代是指当循环被串行执行时的最后一次迭代。`lastprivate` 子句将指示编译器产生可以在循环的并行执行结束时将其私有变量复制回主线程中的对应变量的代码。

例如, 如果我们将下面的代码并行化:

```
for (i = 0; i < n; i++) { x[0] = 1.0;
    for (j = 1; j < 4; j++)
        x[j] = x[j-1] * (i+1);
    sum of powers[i] = x[0] + x[1] + x[2] + x[3];
}n cubed = x[3];
```

在这个循环串行执行至最后一个循环时, `x[3]` 将被赋值为  $n^3$ 。为了使这个值能在 `for` 循环外访问到, 我们必须将 `x` 声明为 `lastprivate` 类型的变量。这时并行版本的代码:

```
#pragma omp parallel for private(j) lastprivate(x)
for (i = 0; i < n; i++) { x[0] = 1.0;
    for (j = 1; j < 4; j++)
        x[j] = x[j-1] * (i+1);
    sum of powers[i] = x[0] + x[1] + x[2] + x[3];
}n cubed = x[3];
```

一个 `parallel for` 编译指导语句可以同时包含 `firstprivate` 和 `lastprivate` 子句。如果该指导语句包含了这两个子句, 那么这两个子句中也可以包含同样的元素。

## 17.5 临界区

让我们来考察一段通过矩形法则的数值积分方法来估算  $\pi$  值的 C 程序。

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

这个例子与前面的 for 循环并不相同，这次的循环不同次的迭代之间并不相互独立。每次迭代都会读取并更新变量 *area* 的值。如果我们简单的这样并行化循环：

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x); /* Race condition! */
}
pi = area / n;
```

我们很可能将不能获得正确的结果，这是由于赋值操作并不能保证原子性（也就是可划分的）。这是一种竞争状况，在这种情况下由于多个线程访问共享变量时，计算会呈现非确定性的特征。

如图 17.5 所示，假设线程 A 和线程 B 在同时地进行同一循环的不同迭代。线程 A 读取了当前的 *area* 变量的值，并且计算和：

$$\text{area} + 4.0/(1.0 + x*x)$$

在它将该计算结果写回时，线程 B 读取了 *area* 的值。然后线程 A 将 *area* 的值进行更新。线程 B 之后计算和并将结果写回。这时 *area* 变量的值就是错误的。

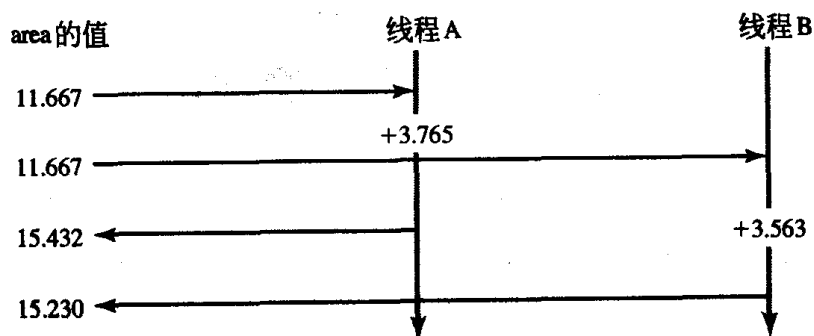


图 17.5 冲突条件的示例。每个线程均对 *area* 变量进行操作。然而，线程 B 在线程 A 写回新的 *area* 数值之前便获得了它的值。所以 *area* 最终的值是错误的。如果线程 B 在线程 A 更新 *area* 之后才读取它的值，那么 *area* 最终的值就是正确的。总之，如果缺乏临界区就会导致不确定的计算结果

对 *area* 变量的读取和更改必须放在同一个临界区内部，同一时刻仅能有一个线程执行这里的代码。

## critical 编译指导语句

在 OpenMP 中我们可以使用下面的编译指导语句来定义一个代码临界区：

```
#pragma omp critical
```

这个编译指导语句将告诉编译器生成相应代码, 使得试图执行该段代码的线程之间互斥。

加上临界区指导语句后, 我们的代码如下:

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    #pragma omp critical
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

这时我们的 C/OpenMP 程序段将能产生正确的结果。for 循环的各个迭代将在不同的线程中完成, 而同一时刻只能有一个线程执行赋值语句并更新 *area* 的值。然而, 这段程序的加速比会不尽如人意。由于每次只允许一个线程进行访问临界区, 那么临界区将成为 for 循环中的一段串行代码。执行这些语句的时间并不能忽略不计。所以根据 Amdahl 定律, 我们可以知道临界区将决定 for 循环的并行化的加速比上界。

当然, 我们刚才试图实现的仅仅是一个加法归约操作。在下一部分中我们将学到一种更有效的完成归约的方法。

## 17.6 归约操作

归约操作非常普遍, 所用在 OpenMP 中我们可以在 `parallel for` 编译指导语句中加入一个归约子句来实现归约。我们要做的仅仅是指定归约操作的类型和要归约的元素, OpenMP 将负责诸如将部分和存储到私有变量中并在循环结束时将各个部分和相加等具体的操作。

归约子句有如下的语法:

```
reduction(<op>:<variable>)
```

这里的 `<op>` 属于在表 17.1 中列出的归约操作符, `<variable>` 是用来完成归约操作的共享变量名。

下面是一段用归约子句代替临界区实现的计算  $\pi$  值的 C 程序:

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x) reduction(+:area)
for (i = 0; i < n; i++) {
```

```

    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;

```

表 17.2 将我们运用矩形法计算 $\pi$ 值的两个程序进行了比较。我们设  $n$  为 100 000 并将程序在一台 Sun 公司的企业服务器 4000 上运行。使用了归约操作的程序很明显的比使用临界区的程序要好。当只有单个活动线程时,前者更快,并且当活动线程数增加时,其程序执行时间逐渐减少。

表 17.1 OpenMP 支持的 C/C++ 归约操作符

操作符	含义	允许的类型	初始值
+	和	浮点数、整数	0
*	积	浮点数、整数	0
&	逐位与	整数	所有位都为 1
	逐位或	整数	0
^	逐位互斥或	整数	0
&&	逻辑与	整数	1
	逻辑或	整数	0

表 17.2 在 Sun 公司的企业服务器 4000 上两个通过矩形法计算 $\pi$ 值程序的运行时间

线程数	程序执行时间(s)	
	使用 critical	使用 reduction
1	0.0780	0.0273
2	0.1510	0.0146
3	0.3400	0.0105
4	0.3608	0.0086
5	0.4710	0.0076

## 17.7 性能改善

有时候把顺序执行的 for 循环转化成并行却增加了执行时间。在这个部分我们将学习三种改善并行循环性能的方法。

### 17.7.1 循环转化

来看下面一段代码:

```

for (i = 1; i < m; i++)
    for (j = 0; j < n; j++)
        a[i][j] = 2 * a[i-1][j];

```

数据相关性图可以帮助我们理解这段代码内的数据相关关系。如图 17.6 所示, 由于行与行间具有相关性, 两个行是不可能同时更新的。然而, 两个列的更新是可以同时进行的。这说明以  $j$  编号的循环可以被并行执行, 而以  $i$  编号的循环就不可以。

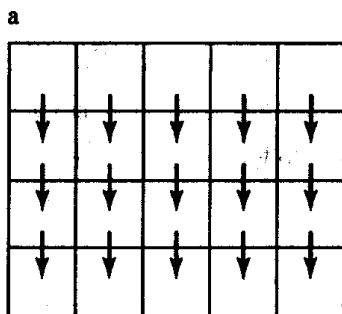


图 17.6 一个嵌套循环的数据相关性图, 列可以同时更新, 而行则不行

如果在内层循环的前面插入 `parallel for` 编译指导语句, 并行程序虽然会正确执行, 但是性能的表现却令人不满, 因为这将会导致  $m-1$  次线程派生-会合 (`fork/join`), 即外层循环每迭代一次就执行一次派生-会合 (`fork/join`)。

然而, 如果将循环转换一下:

```
#pragma parallel for private(i)
for (j = 0; j < n; j++)
    for (i = 1; i < m; i++)
        a[i][j] = 2 * a[i-1][j];
```

这样只需要一次 `fork/join` (在外层循环的外面)。数据相关性并没有改变; 以  $j$  编号的循环迭代之间依然不存在数据相关性。从这个方面上讲, 我们确实改进了程序。

然而, 我们还必须认识到代码转化对高速缓存命中率的影响。就这个例子来说, 每个线程以  $a$  的一列为单位执行, 而非  $a$  的一行。因为  $C$  矩阵是以行为单位的顺序存储的, 循环转化将降低高速缓存命中率, 这依赖于  $m, n$ , 活动的线程数和底层系统的体系结构。

## 17.7.2 条件执行循环

如果循环的迭代次数不多, 那么花在线程派生-会合 (`fork/join`) 上的时间将超过将迭代分配到多线程中去执行而节省的时间。比如, 前面讲过的矩形规则的并行实现:

```
area = 0.0;
#pragma omp parallel for private(x) reduction (+:area)
for (i = 0; i < n; i++) {
    x = (i + 0.5)/n;
    area += 4.0 / (1.0 + x * x);
}
pi = area / n;
```

表 17.3 给出了不同的  $n$  值和线程数下这段程序在一台 Sun 公司企业服务器 4000 上的平均执行时间。正如你所看到的, 当  $n$  为 100 时, 顺序执行时间非常短, 增加工作线程数

只会导致整个执行时间增长。当  $n$  为 100 000 时，在四个线程上执行的并行程序相对于顺序程序得到了 3.16 的加速比。

表 17.3 在一台 Sun 公司企业服务器 4000 上用矩形规则计算  $\pi$  值的并行 C 程序的执行时间，它是矩形数和线程数的函数

线程数	执行时间(s)	
	$n=100$	$n=100\ 000$
1	0.964	27.288
2	1.436	14.598
3	1.732	10.506
4	1.990	8.648

使用 if 子句，我们可以指导编译器插入控制代码以确定在运行时是否将执行并行化。它的语法如下：

```
if (<scalar expression>)
```

如果标量表达式的值为真，循环将并行执行。否则，它将被顺序执行。

从下面用矩形规则计算  $\pi$  的例子可以看到在并行程序中如何在 parallel for 编译指导语句里加入 if 子句：

```
#pragma omp parallel for private(x) reduction(+:area) if(n > 5000)
for (i = 0; i < n; i++) {
    ...
}
```

在这个例子中，只有当  $n > 5000$  时循环才会被分配到多个线程中执行。

### 17.7.3 循环调度

在一些循环中，执行不同的迭代所需的时间区别相当大。比如，下面这个初始化一个上三角矩阵的两层循环：

```
for (i = 0; i < n; i++)
    for (j = i; j < n; j++)
        a[i][j] = alpha_omega(i, j);
```

假设迭代之间没有数据相关性，则我们并行化外层循环来获得最小的派生-会合 (fork/join) 开销。如果每次对函数 `alpha_omega` 的调用所花的时间均相同，则外层循环的第一次迭代（当  $i$  为 0 时）的运行时间相当于最后一次迭代（当  $n$  为  $n-1$  时）运行时间的  $n$  倍。将内外层循环互换也不能解决这种不平衡。

假设  $n$  次迭代被分配到  $t$  个线程中，如果每个线程被分到连续的  $(n/t)$  或  $(n/t)$  次迭代，我们将会得到较低的并行执行效率，因为有些线程完成这些迭代的速度比其他线程要快。

`schedual` 子句使得我们可以指定如何调度循环迭代，即如何将各次迭代在线程间分配。在静态调度中，循环迭代被执行之前已经被分配到各个线程了。如果使用动态调度，那么在循环执行开始时将只分配了循环中的一部分迭代。线程在完成了分配给它们的迭代后可

以获得额外的工作。所有迭代均被分配到线程后, 分配过程才算结束。静态调度开销小但是会有明显的负载不平衡现象。动态调度开销相对大, 但是可以减少负载的不平衡。

无论是静态还是动态调度, 线程都会分配到循环中一段连续的迭代。增大数据块大小会降低程序开销同时提高高速缓存命中率。减小数据块大小可以得到更好的负载平衡效果。

调度子句的语法如下:

```
schedule(<type> [, <chunk>])
```

换句话说, 我们必须指定调度的类型, 而不一定要给出数据块的大小。根据这两个参数我们可以很容易地描述多种调度类型:

- `schedule (static)` 静态地分配约  $n/t$  个连续迭代到每个线程;
- `schedule (static, C)` 将数据块静态地轮换分配给各个任务, 每个数据块包括  $C$  个连续的迭代;
- `schedule (dynamic)` 动态地将迭代逐个分配到各个线程;
- `schedule (dynamic, C)` 给各个任务动态分配任务块, 每个任务块包含  $C$  个迭代;
- `schedule (guided, C)` 一种采用指导性的启发式自调度方法。开始时每个任务会分配到较大的迭代块, 之后任务每次请求新的迭代时会被分配到大小递减的迭代块。迭代块的大小将指数地下降到  $C$ , 这是迭代块大小的下界;
- `schedule (guided)` 进行指导性自调度, 块最小为 1;
- `schedule (runtime)` 在运行时根据环境变量 `OMP_SCHEDULE` 确定调度类型。比如, Unix 命令。

```
setenv OMP_SCHEDULE "static,1"
```

可设置运行时调度类型为轮换式分配。

当 `parallel for` 编译指导语句中不包含 `schedule` 子句时, 大部分运行时系统默认为采用简单静态调度, 将连续的循环迭代块分配给各个任务。

回到我们最初的例子上来, 运行时外层 `for` 循环的任何一次迭代都是可预测的。循环迭代的轮换分配在各个线程间平衡了工作负载:

```
#pragma omp parallel for private(j) schedule(static,1)
for (i = 0; i < n; i++)
    for (j = i; j < n; j++)
        a[i][j] = alpha_omega(i,j);
```

增大数据块大小可以在牺牲负载平衡的条件下提高高速缓存命中率。数据块大小的最佳值与具体的系统情况有关。

## 17.8 更普遍的数据并行

到这里为止我们把主要注意力集中在了简单 `for` 循环的并行化上。他们或许是程序中最常见的并行部分了, 尤其是在用 MPI 写的程序中。然而, 我们不该忽略对其他并行性的

利用。在这个部分中，我们来考察两个非 for 循环的数据并行实例。

首先来考虑一个用来执行一连串任务的算法。这个算法与我们在第 9 章中设计了解决文档分类问题的算法有些类似。在设计后者的过程中，我们用的是消息传递模式。因为在这种模式中没有共享存储，我们用一个单独的进程作为“管理者”，负责维护整个任务列表。“工人”进程用来完成各项任务，当准备好可以开始完成另一项任务的时候它们便发信息给“管理者”。

而共享存储模式中的每个线程都可以访问到相同的“待完成”工作表，所以并不需要单独的管理线程。

接下来的代码段来自于用于完成存储在“待完成”列表中的任务的程序（见图 17.7）：

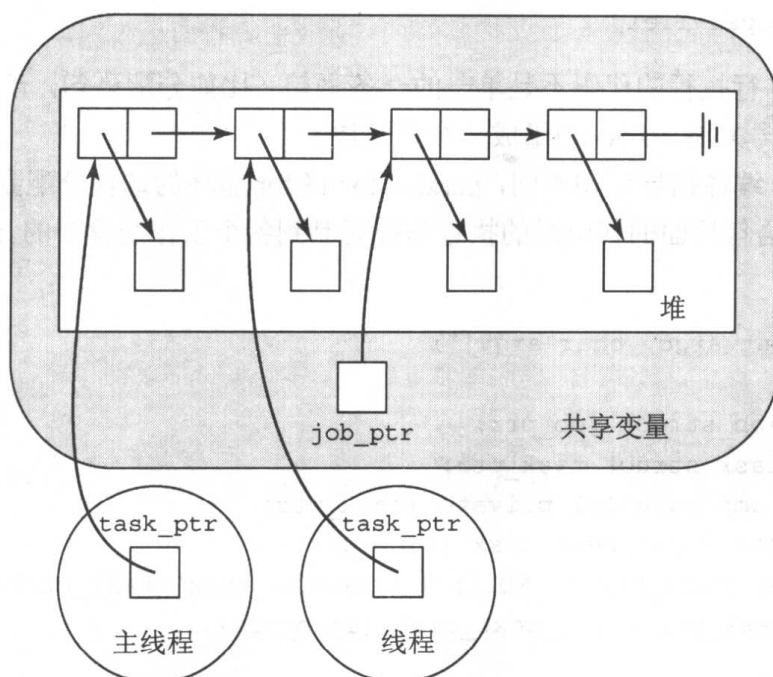


图 17.7 两个线程针对单一的工作列表进行工作  
变量 `job_ptr` 必须是共享的，而变量 `task_ptr` 必须是私有的

```
int main (int argc, char argv[])
{
    struct job_struct job_ptr;
    struct task_struct task_ptr;
    ...
    task_ptr = get_next_task (&job_ptr);
    while (task_ptr != NULL) { complete_task (task_ptr);
        task_ptr = get_next_task (&job_ptr);
    }...
}

char get_next_task(struct job_struct job_ptr) { struct task_struct answer;
    if (job_ptr == NULL) answer = NULL;
    else { answer = (job_ptr)->task;
        job_ptr = (job_ptr)->next;
```



```

    }return answer;
}

```

如何让这个算法并行执行起来？我们希望所有的线程都做同样的事情：不断的从列表中取任务并完成它，直至列表中所有任务都被取完为止。我们必须确认没有两个线程在做列表中的同一项任务。换句话说，函数 `get_next_task` 的执行必须具备原子性。

## 17.8.1 parallel 编译指导语句

`parallel` 编译指导语句出现在一段需要被所有线程执行的代码前面。其语法如下：

```
#pragma omp parallel
```

如果我们要并行执行的代码不是单一的一条语句（比如分配语句，`if` 语句或者 `for` 循环），可以用大括号来将一组语句组成一个代码块。

与 `parallel for` 编译指导语句不同，`parallel` 将 `for` 循环的迭代分配到工作线程中，而 `parallel` 编译指导语句后面的代码段的执行是被复制到各个工作线程中的。`main` 函数部分如下所示：

```

int main (int argc, char argv[])
{
    struct job_struct job_ptr;
    struct task_struct task_ptr;
    #pragma omp parallel private (task_ptr)
    { task_ptr = get_next_task (&job_ptr);
      while (task_ptr != NULL) { complete_task (task_ptr);
        task_ptr = get_next_task (&job_ptr);
      }
    }
}

```

现在需要确定函数 `get_next_task` 的执行是独立的。否则，允许两个线程同时执行函数 `get_next_task` 会导致不止一个线程返回相同的 `task_ptr` 值。

我们用 `critical` 编译指导语句来确保临界区代码被互斥地执行。下面是重写的 `get_next_task` 函数：

```

char get_next_task(struct job_struct job_ptr) {
    struct task_struct answer;
    #pragma omp critical
    {
        if (job_ptr == NULL) answer = NULL;
        else { answer = (job_ptr)->task;
              job_ptr = (job_ptr)->next;
            }
    }
    return answer;
}

```

## 17.8.2 omp\_get\_thread\_num 函数

本章的前面部分我们用矩形规则计算 $\pi$ 值。在第 10 章我们用蒙特卡洛法计算了 $\pi$ 值。这种方法（如图 17.8 所示）是在单元格中生成一对点（坐标在 0 和 1 之间）。我们对落在单位圆内的点进行计数，并用它来计算落在单位圆内点的所占的比例。这个比值的期望值为 $\pi/4$ ；因此将这个分数乘以 4 便大致得到了 $\pi$ 的值。

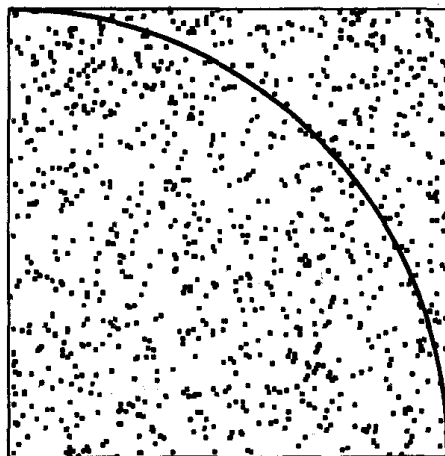


图 17.8 用蒙特卡洛方法计算 $\pi$ 值的例子。在这个例子中我们在坐标为 0 到 1 间生成了 1000 对点。773 对落在了单位圆里，于是我们得到的 $\pi$ 值约为  $4(773/1000)$ ，即 3.092

下面是 C 代码实现的这个算法：

```
int count; /* Points inside unit circle */
unsigned short xi[3]; /* Random number seed */
int i;
int samples; /* Points to generate */
double x, y; /* Coordinates of point */
samples = atoi (argv[1]);
xi[0] = atoi (argv[2]);
xi[1] = atoi (argv[3]);
xi[2] = atoi (argv[4]);
count = 0;
for (i = 0; i < samples; i++) {
    x = erand48(xi);
    y = erand48(xi);
    if (x*x+y*y <= 1.0) count++;
}
printf ("Estimate of pi: %7.5f\n", 4.0*count/samples);
```

如果希望通过使用多线程加速执行过程，我们必须确认每个线程都在生成不同的随机数流。否则，若每个线程都生成了相同的  $(x,y)$ ，不可能通过使用多线程而增加结果的精确度。因此  $xi$  必须为私有变量，并且每个线程给  $xi$  数组的初始值必须不同。这意味着我们要找到区别各个线程的方法。

在 OpenMP 中多处理机的每个线程有不同的线程号。我们可以通过函数 `omp_get_thread_num` 得到这个值。该函数的声明如下:

```
int omp_get_thread_num (void)
```

如果有  $t$  个工作线程, 线程号从 0 到  $t-1$ 。主线程的线程号通常为 0。向 `xi[2]` 分配线程号使得每个线程可以得到不同的随机数。

### 17.8.3 omp\_get\_num\_threads 函数

为了在各个线程之间分配迭代, 我们必须知道活动线程的总数。函数 `omp_get_num_threads` 的声明如下:

```
int omp_get_num_threads (void)
```

这个函数返回当前并行区域内活动线程的总数。我们使用这个信息以及线程各自的编号在线程之间分配循环的各次迭代。

每一个线程在其内部的一个私有变量上积累它数到的位于单位圆内的点数。当一个线程完成它自己的循环后, 它在一个临界区内将它的部分和累加到 `count` 变量上。

用 OpenMP 来实现 Monte Carlo 法求解  $\pi$  值的算法如图 17.9 所示。

### 17.8.4 编译指导语句 for

当并行化 `for` 循环时, `parallel` 编译指导语句同样可以派上用场。看下面的两层嵌套循环:

```
for (i = 0; i < m; i++) {  
    low = a[i];  
    high = b[i];  
    if (low > high) {  
        printf ("Exiting during iteration %d\n", i);  
        break;  
    }  
    for (j = low; j < high; j++)  
        c[j] = (c[j] - a[i])/b[i];  
}
```

因为包含有 `break` 语句, 所以外层循环迭代不能并行执行。如果在以  $j$  编号的循环前加上 `parallel for` 编译指导语句, 会导致每执行一次外层迭代线程就执行一次派生-会合操作。我们应该避免这种开销。前面我们学习过如何通过转化循环顺序来解决这个问题, 但是由于本例的数据相关性这个方法无法使用。

如果在以  $i$  编号的循环前加上 `parallel` 编译指导语句, 就可以只执行派生会合-操作一次。默认方式下, 所有的线程执行这段代码所描述的工作。当然, 我们希望内层循环迭代能够由多个线程协同完成。`for` 编译指导语句可以完成这个工作:

```

/*
 * OpenMP implementation of Monte Carlo pi-finding algorithm
 */
#include <stdio.h>
int main (int argc, char *argv[]){
    int count;          /* Points inside unit circle */
    int i;
    int local_count;     /* This thread's subtotal */
    int samples;         /* Points to generate */
    unsigned short xi[3]; /* Random number seed */
    int t;               /* Number of threads */
    int tid;             /* Thread id */
    double x, y; /* Coordinates of point */

    /* Number of points and number of threads are
       command-line arguments */
    samples = atoi(argv[1]);
    omp_set_num_threads (atoi(argv[2]));
    count = 0;
    #pragma omp parallel private(xi,t,i,x,y,local_count)
    {
        local_count = 0;
        xi[0] = atoi(argv[3]);
        xi[1] = atoi(argv[4]);
        xi[2] = tid = omp_get_thread_num();
        t = omp_get_num_threads();

        for (i = tid; i < samples; i += t) {
            x = erand48(xi);
            y = erand48(xi);
            if (x*x+y*y <= 1.0) local_count++;
        }
    }
    #pragma omp critical
        count += local_count;
    printf ("Estimate of pi: %7.5f\n", 4.0*count/samples);
}

```

图 17.9 C/OpenMP 程序用蒙特卡洛方法计算 $\pi$  值

```
#pragma omp for
```

加上这条编译指导语句后，代码段如下：

```

#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        printf ("Exiting during iteration %d\n", i);
        break;
    }
    #pragma omp for
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}

```

然而, 我们的工作尚未结束。

### 17.8.5 single 编译指导语句

我们对以  $j$  编号的循环进行了并行化。那么外层循环内的其他代码呢? 我们当然不想一再的看见错误信息。

`single` 编译指导语句告诉编译器应该只有一个线程执行它后面的这个代码段。语法如下:

```
#pragma omp single
```

加上这条编译指导语句后代码如下:

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
#pragma omp single
        printf ("Exiting during iteration %d\n", i);
        break;
    }
#pragma omp for
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}
```

现在这段代码可以正确执行了, 但是性能还有待改进。

### 17.8.6 nowait 子句

编译器在每个 `parallel for` 语句后面都会放置一个同步屏障。对于刚才这个例子, 同步屏障是必要的, 因为我们需要在每个线程准备开始下个以  $i$  编号的迭代之前确认它们已经完成了前一个迭代。否则, 线程有可能会改变 `low` 或 `high` 的值, 从而改变由其他线程执行的以  $j$  编号的循环的迭代数。

另一方面, 如果设置 `low` 和 `high` 为私有变量, 则没有必要在以  $j$  为编号的循环出口处放置同步屏障。`parallel for` 编译指导语句中的 `nowait` 子句告诉编译器无需在并行 `for` 循环的出口处放置同步屏障。

设置 `low` 和 `high` 为私有变量并加上 `nowait` 子句后, 例程的最终版本为:

```
#pragma omp parallel private(i,j,low,high)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
```

```
    if (low > high) {  
#pragma omp single  
        printf ("Exiting during iteration %d\n", i);  
        break;  
    }  
#pragma omp for nowait  
    for (j = low; j < high; j++)  
        c[j] = (c[j] - a[i])/b[i];  
}
```

## 17.9 功 能 并 行

到这里为止我们一直把注意力放在开发数据并行性方面。程序的另一种并行性为功能并行。OpenMP 使得我们能够为不同的线程分配不同部分的代码。

来看下面的代码段：

```
v = alpha();  
w = beta();  
x = gamma(v, w);  
y = delta();  
printf ("%6.2f\n", epsilon(x,y));
```

如果所有这些函数之间都没有副作用，我们可以用图 17.10 来表示数据间的相关性。很明显函数 alpha, beta 和 delta 可以并行执行。如果同时执行这些函数，就没有其他的未开发的并行性了，因为函数 gamma 只能在函数 alpha 和 beta 后面调用，函数 epsilon 则必须要在 gamma 后面调用。

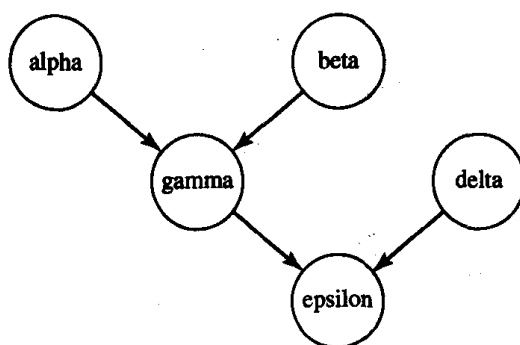


图 17.10 第 17.9 节代码段的数据相关图

### 17.9.1 parallel sections 编译指导语句

在由  $k$  块代码组成的一个代码块前加上 parallel sections 编译指导语句，我们可以让  $k$  个线程同时执行这  $k$  个块的工作。语法如下：

```
#pragma omp parallel sections
```

## 17.9.2 section 编译指导语句

section 编译指导语句放在 parallel sections 编译指导语句后的代码块里的每个小块前面 (parallel sections 编译指导语句后面的第一个 section 编译指导语句可以省略)。

这上面的例子中, 可以同时调用 alpha、beta 和 delta 函数。在对代码的并行过程中, 我们用一对大括号来建立一个包括这三个任务语句的代码块 (一个任务语句代表了一个代码块, 因此一个包括三个任务语句的代码块相当于一个包含了三段代码的代码块)。

```
#pragma omp parallel sections
{
#pragma omp section /* This pragma optional */
    v = alpha();
#pragma omp section
    w = beta();
#pragma omp section
    y = delta();
}
x = gamma(v, w);
printf ("%6.2f\n", epsilon(x,y));
```

注意, 我们重新安排了三条赋值语句, 以使得这三个工作得以并行执行。

## 17.9.3 sections 编译指导语句

让我们再来看一下图 17.10 中所示的数据相关性。在这段代码中还可以用另一种方法来开发它的功能并行性。就像前面说过的, 如果我们让函数 alpha、beta 和 delta 并行执行, 我们就失去了进一步开发功能并行的机会。然而, 如果我们只让函数 alpha 和 beta 并行执行, 则当他们返回时我们便可以让函数 gamma 和 delta 并行执行。

这样程序中就有了两个不同的并行区, 并且它们前后相连。为了减少派生-会合操作的开销, 我们可以把所有四个任务语句放在 parallel 编译指导语句后面的同一个块中, 然后用 sections 编译指导语句来指定第一对和第二对并行执行的函数。

sections 编译指导语句的语法如下:

```
#pragma omp sections
```

它出现在一个 parallel 代码块中。其意义相当于我们前面讲过的 parallel sections 编译指导语句。

下面是另一种用 sections 编译指导语句表现功能并行的方法:

```
#pragma omp parallel
{
```

```
#pragma omp sections
{
    #pragma omp section /* This pragma optional */
        v = alpha();
    #pragma omp section
        w = beta();
}
#pragma omp sections
{
    #pragma omp section /* This pragma optional */
        x = gamma(v, w);
    #pragma omp section
        y = delta();
}
}
printf ("%6.2f\n", epsilon(x,y));
```

从一个方面看这个方法比前面讲的方法要好，因为它有两个并行区，每个需要两个线程。第一种方法只有一个需要三个线程的并行区。如果系统有两个处理器，则第二种方法能得到更高的性能。不过这些都取决于单个函数的执行时间。

## 17.10 本章小结

OpenMP 是用于共享存储并行程序设计的应用程序接口。共享存储模型依赖于派生-会合类型的并行性。你可以把一个共享存储程序的执行看作若干段串行代码与并行代码的交替执行。主线程执行所有的串行代码。当遇到一个并行区时它派生出其他线程。所有线程之间通过共享变量进行通信。当并行代码区结束时，这些线程之间进行同步操作并结束，程序恢复到只由一个主线程执行。

本章介绍了 OpenMP 编译指导语句及其子句，它们可以把一个串行 C 程序转化为一个在多处理器上运行的并行程序。我们首先考虑的是 for 循环的并行化，C 程序中的数据并行性通常是以 for 循环的形式表达的。我们用 parallel for 编译指导语句来告诉编译器哪些循环的迭代可以并行执行。满足一定条件的 for 循环才可以并行执行。控制子句必须足够简单，这样运行时系统才能在循环执行前确定有多少次迭代。循环中不能存在 break 语句，goto 语句，以及其他能够让循环提早结束的语句。

我们同样也讨论了用 parallel sections 编译指导语句来开发功能上的并行性。这条编译指导语句出现在一个包含了若干个小代码段的代码段前面，这个代码段中的每一个小代码段代表了一个独立的任务，可以与其他小代码段并行执行。

编译指导语句 parallel 出现在一段将被所有线程并行执行的代码前面。当所有线程执行同样的代码时，这就是和大多数用 MPI 实现的程序所体现的并行模式一样的 SPMD 型



并行。由编译指导语句 `parallel` 标记的代码段内可以出现编译指导语句 `for` 或 `sections`，从而指导编译器开发程序的数据并行性或功能并行性。

我们还可以使用编译指导语句来指定并行代码段内必须串行执行的部分。编译指导语句 `critical` 用来指示一个代码段为代码临界区，临界区中的代码被强制互斥地执行。编译指导语句 `single` 用来指示一段必须仅由一个线程执行的代码。

我们通过给编译指导语句附加一些子句来向编译器传达附加的信息。子句 `private` 使得每个线程拥有一份自己的变量列表。通过使用子句 `firstprivate` 和/或 `lastprivate`，变量的值可以在原始变量和私有变量间进行复制。采用 `reduction` 子句时编译器能够生成有效代码从而在一个并行循环中进行归约操作。子句 `schedule` 让用户自己确定如何将循环迭代分配给各个任务。子句 `if` 让系统在运行时决定一个结构应该被串行执行还是并行执行。子句 `nowait` 在并行结构的结尾处用来消除同步等待带来的额外开销。

虽然我们是在特定的编译指导语句环境中引入子句进行介绍的，但是大多数子句可以应用在几乎所有编译指导语句中。表 17.4 中列出了本章中我们介绍过的子句，以及它们可以从属于的编译指导语句。

表 17.4 各种子句分别能应用于何种编译指导语句

编译指导语句	允许的子句
<code>critical</code>	None
<code>for</code>	<code>firstprivate</code> , <code>lastprivate</code> , <code>nowait</code> , <code>private</code> , <code>reduction</code> , <code>schedule</code>
<code>parallel</code>	<code>firstprivate</code> , <code>if</code> , <code>lastprivate</code> , <code>private</code> , <code>reduction</code>
<code>parallel for</code>	<code>firstprivate</code> , <code>if</code> , <code>lastprivate</code> , <code>private</code> , <code>reduction</code> , <code>schedule</code>
<code>parallel sections</code>	<code>firstprivate</code> , <code>if</code> , <code>lastprivate</code> , <code>private</code> , <code>reduction</code>
<code>sections</code>	<code>firstprivate</code> , <code>lastprivate</code> , <code>nowait</code> , <code>private</code> , <code>reduction</code>
<code>single</code>	<code>firstprivate</code> , <code>nowait</code> , <code>private</code>

我们考察了不同的提高 `for` 循环并行化性能的方法。这些方法包括互换多层循环内外层顺序，条件化对循环的并行操作，以及改变循环迭代的调度策略。

表 17.5 中将 OpenMP 和 MPI 进行了对比。这两种编程环境都可以应用于多处理器。MPI 适合于在多机环境上应用。而由于 OpenMP 有共享变量，因而并不适合这种一般的无共享内存的多机系统结构。对于编程人员而言，用 MPI 更易于对存储层次进行控制。另一方面，OpenMP 由于支持程序的并行程度递增，因而在这方面有着更突出的优势。而且，用 MPI 编写的并行程序通常比其串行版本要长很多。与此不同，用 OpenMP 编写的并行程序不会比串行版本长多少。

表 17.5 OpenMP 和 MPI 之间的比较

特点	OpenMP	MPI
适合多处理器	是	是
适合多计算机	否	是
支持增量式并行化	是	否
最少附加代码	是	否
显式控制内存层次	否	是

## 17.11 主要术语

canonical shape	规范格式
grain size	粒度
race condition	竞争(状态)
chunk	数据块
guided self-scheduling	指导性自调度
reduction variable	归约变量
clause	子句
incremental parallelization	增量并行化
schedule	调度
critical section	临界区
master thread	主线程
sequentially last iteration	串行执行的最后一次迭代
dynamic schedule	动态调度
pragma	编译指导语句
shared variable	共享变量
exection context	执行现场
private clause	private 子句
static schedule	静态调度
fork/join parallelism	派生-会合型并行
private variable	私有变量

## 17.12 参考文献

OpenMP 官方网站的地址是 [www.OpenMP.org](http://www.OpenMP.org)。您可以从这个站点下载 C/C++ 和 Fortran 版本的 OpenMP 规范。

Chandra 等人所著的“Parallel Programming in OpenMP”对共享存储应用编程接口进行了很好介绍。它对 OpenMP 的特征进行了更宽更深的讲解。它同时也讨论了如何对 OpenMP 代码进行性能调试。

## 17.13 练习题

17.1 在本章介绍的 4 个 OpenMP 函数中, 哪两个与 MPI 函数最相似? 请指出相应的 MPI 函数的名称。

17.2 对下面各段代码, 用 OpenMP 编译指导语句将循环并行化, 或者解释为什么这段代码不适合并行执行。

```
a. for (i = 0; i < (int) sqrt(x); i++) {  
    a[i] = 2.3 * i;  
    if (i < 10) b[i] = a[i];  
}  
b. flag = 0;  
   for (i = 0; (i < n) & (!flag); i++) {  
       a[i] = 2.3 * i;  
       if (a[i] < b[i]) flag = 1;  
   }  
c. for (i = 0; i < n; i++)  
    a[i] = foo(i);  
d. for (i = 0; i < n; i++) {  
    a[i] = foo(i);  
    if (a[i] < b[i]) a[i] = b[i];  
}  
e. for (i = 0; i < n; i++) {  
    a[i] = foo(i);  
    if (a[i] < b[i]) break;  
}  
f. dotp = 0;  
   for (i = 0; i < n; i++)  
       dotp += a[i] * b[i];  
g. for (i = k; i < 2*k; i++)  
    a[i] = a[i] + a[i-k];  
h. for (i = k; i < n; i++)  
    a[i] = b * a[i-k];
```

17.3 假设 OpenMP 没有 reduction 子句。指出如何通过增加一个私有变量和 critical 编译指导语句来执行有效的并行归约。针对 17.5 节的估算 $\pi$  值程序具体说明你的方法。

17.4 第 17.7.3 节中讨论了在初始化上三角矩阵时, 为了平衡线程间的工作负载而采用的任务轮换调度方法。请解释为什么从 1 开始增大数据块的尺寸可以提高高速缓存的命中率。

17.5 给出一个简单的并行 for 循环, 使之用静态轮换调度策略实现比采用将包含若干次迭代的数据块依次分配给任务 (每个任务仅分得一个数据块) 的策略执行的更快。这个 for 循环不能为嵌套循环。

17.6 请自己举一个并行 for 循环的例子, 使之用动态调度策略比静态调度策略执行得更快。

17.7 在 17.8 节中我们编写了一段代码让多个线程共同完成一个“待完成工作”的列表。请解释当函数 get\_next\_task 不是互斥执行时两个线程将有可能执行同一个任务。

17.8 图 17.9 描述了用蒙特卡洛算法计算 $\pi$  的 C/OpenMP 程序。其中 for 循环的迭代被显式分配到各个线程中。请为这段代码另写一个版本, 用编译指导语句 for 来指示运行

时系统如何分配循环迭代。用不同的  $n$  值（采样数目）和  $t$  值（线程数目）测试你的程序。

17.9 用 OpenMP 编译指导语句最大限度地开发下列 Winograd 矩阵相乘算法（由 Baase 和 Van Gelder 改写）中的并行性。

```
for (i = 0; i < m; i++) {
    rowterm[i] = 0.0;
    for (j = 0; j < p; j++)
        rowterm[i] += a[i][2*j] * a[i][2*j+1];
}
for (i = 0; i < q; i++) {
    colterm[i] = 0.0;
    for (j = 0; j < p; j++)
        colterm[i] += b[2*j][i] * b[2*j+1][i];
}
```

17.10 用 OpenMP 编译指导语句来实现 Sieve of Eratosthenes 并行程序。用不同的  $n$  值和  $t$  值（线程数）测试你的程序。

17.11 用 OpenMP 编译指导语句实现 Floyd's Algorithm 的并行程序。用不同的  $n$  值和  $t$  值（线程数）测试你的程序。

17.12 用 OpenMP 编译指导语句实现矩阵向量相乘算法（如图 8.1 所示）的并行版本。用不同的  $n$  值和  $t$  值（线程数）测试你的程序。

17.13 用 OpenMP 编译指导语句实现一个并行程序，用螺旋线高丝消去法求解线性方程密集系统并在最后使用带入法。用不同的  $n$  值和  $t$  值（线程数）测试你的程序。

17.14 假设系数矩阵  $A$  是对称带状矩阵，用 OpenMP 编译指导语句实现一个实现共轭梯度法（如图 12.13 所示）的并行程序。用不同的  $n$  值和  $t$  值（线程数）测试你的程序。

17.15 用 OpenMP 编译指导语句实现一个用规则样本进行并行排序的程序。用不同的  $n$  值和  $t$  值（线程数）测试你的程序。

17.16 用 OpenMP 编译指导语句实现一个解决 15-puzzle（第 17 章）问题的并行程序。针对问题的不同随机程度，用不同的  $n$  值和  $t$  值（线程数）测试你的程序。

# 第 18 章 融合 OpenMP 和 MPI

The good things in life are not to be had singly, but come to us with a mixture.

Charles Labm, *That You Must Love Me and Love My Dog*

## 18.1 概 述

大多数拥有成百上千个 CPU 的商用多计算机实际上是通过将多处理机集中起来而得到的，很多商业集群是由双 CPU 甚至四 CPU 节点组成的。基于以上原因我们需要知道如何将 MPI 程序改写成由 MPI 和 OpenMP 混合编写的程序，以能适用于由多处理器计算机组成的多机集群。

当然，我们可以在系统的每个 CPU 上创建一个 MPI 进程，如图 18.1 (a) 所示，从而在多处理器集群上执行仅由 MPI 实现的并程序。在这种情况下，即使有一些 MPI 进程在发生同一个多处理器计算机上，所有的进程间的交互也都是通过消息传递方式进行的。然而，某些情况更适合混合模式的并程序，如图 18.1 (b) 所示。在混合模式下，每个多处理器上运行有一个 MPI 进程。在代码的并行区内 MPI 进程派生出若干个线程占据着这个多处理器计算机的各个 CPU，这些线程将通过共享变量进行交互。

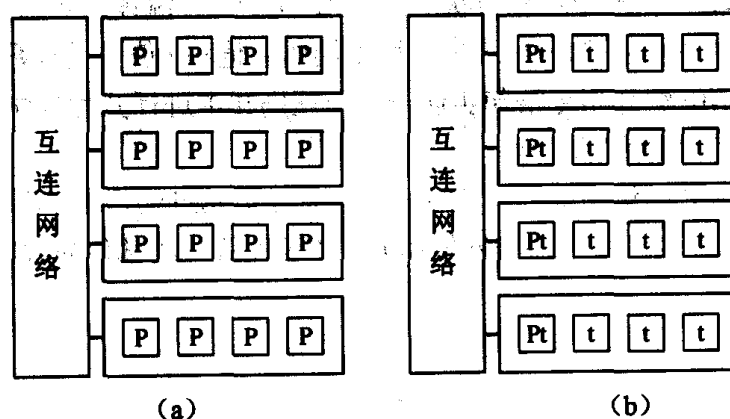


图 18.1 在多处理器机群上执行并行算法的两种方法。(a) 为每个 CPU 创建一个 MPI 进程 P；  
(b) 为每个多处理器创建一个 MPI 进程 P，并创建占用 CPU 的线程 t

在很多情况下，使用 MPI 和 OpenMP 混合编程的程序比单纯用 MPI 编写的程序执行效率更高。

有时混合编程模式的程序效率更高是因为它们有着更少的通信开销。假设有一个由  $m$  个多处理器计算机组成的集群，每台计算机有  $k$  个 CPU。我们在这样的集群上运行一个程序。为了利用每一个 CPU，用 MPI 编写的程序必须创建  $mk$  个进程。在通信过程中， $mk$  个进程一直是存活的。另一方面，混合编程模式下的程序只需创建  $m$  个进程。在代码的并行区，工作负载被分配到每台计算机的  $k$  个线程中。因此每个 CPU 也都被利用起来了。然

而，在通信过程中，只存在  $m$  个进程。与完全由 MPI 程序书写的程序相比，混合编程模式程序减少了通信开销，从而能得到更高的加速比。

另一种混合编程模式程序能获得更高加速比的可能情形是我们是否可以用轻量级的线程，而非重量级进程将一些计算并行化。请参考下面的例子。

假设我们正在并行化一个串行运行时间为 100 秒的程序。这个程序花在执行内部串行操作的时间为 5 秒（占总执行时间的 5%）；花在可被完全并行化的部分上的操作时间为 90 秒（占总执行时间的 90%）。我们将程序的这个部分改写为并行代码以得到线性加速比。

剩下的 5 秒（5% 的执行时间）对应的是那些本可以并行执行但是会引入很大通信开销的操作。由于 MPI 函数调用所需要的时间太多，以至于我们不将这些操作并行执行，而是将在各个 MPI 进程中均保留它们。

然而，我们假设在共享存储环境中，这些操作的并行化将是可行的。进一步假设如果将这些操作在两个 CPU 上执行时，并行开销可以忽略。

下面我们来计算一下程序所能得到的加速比，假设我们在一个由 8 个双处理器节点组成的集群上执行该程序。一共有 16 个 MPI 进程时，最大加速比为（使用 Amdahl 规则）：

$$\frac{1}{(0.10 + 0.90/16)} = 6.4$$

另一种方法，我们可以在 8 个 MPI 进程上执行这段程序并且允许每个进程内使用双线程。对于那 90% 完全可以并行化的代码，由 16 个线程来执行这些操作执行时间将为原来的  $1/16$ 。对于那 5% 的在节点间复制的代码，由两个线程执行这些操作速度将提升两倍。剩下的 5% 的串行代码执行效率不变。因此可以得到的最大加速比为：

$$\frac{1}{0.05 + 0.05/2 + 0.90/16} = 7.6$$

在这种情况下混合编程模式程序要比单纯用 MPI 编写的程序快 19%。

当一些 MPI 进程空闲而其他进程忙碌的时候，混合编程模式也可以体现它的优势。假设一段应用程序在一个多处理器计算机的集群上执行。在其中一台计算机上，3 个 MPI 进程在等待消息，而第 4 个进程正在忙碌。如果忙碌的进程能够利用那些空闲的 CPU 来执行一些并行操作使执行速度更快，那么为此派生出一些线程将是值得的。

在本章中我们将通过两个例子学习程序是如何由将单纯的 MPI 程序转化为由 MPI 和 OpenMP 混合模式编写的程序。第一个例子是在第 12 章中介绍过的共轭梯度算法的 MPI 实现。第二个例子是在第 13 章中介绍过的 Jacobi 方法的 MPI 实现。在这两个例子中对原来的由 MPI 实现的 C 程序进行相对较小的改写就能将它转化为 MPI 和 OpenMP 混合模式的 C 程序，从而获得加速比的明显提高。

## 18.2 共轭梯度算法

### 18.2.1 MPI 程序

图 18.2 中所示的程序是用共轭梯度算法来解线性方程  $Ax = b$  的具体实现，其中系数矩阵  $A$  是正定的。这段程序基于对矩阵  $A$  在进程间进行行分解的设计。程序将向量  $b$  和其他

所有向量复制到各个进程。

```

/* Conjugate Gradient Method in MPI */

#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
#include "MyMPI.h"
main (int argc, char *argv[])
{
    double **a;          /* Solving Ax = b for x */
    double *astorage;    /* Holds elements of A */
    double *b;           /* Constant vector */
    double *x;           /* Solution vector */
    int p;               /* MPI Processes */
    int id;              /* Process rank */
    int m;               /* Rows in A */
    int n;               /* Columns in A */
    int n1;              /* Elements in b */

    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    read_block_row_matrix (id, p, argv[1], (void *) &a,
        (void *) &astorage, MPI_DOUBLE, &m, &n);
    n1 = read_replicated_vector (id, p, argv[2],
        (void **) &b, MPI_DOUBLE);
    if ((m != n) || (n != n1)) {
        if (!id)
            printf ("Incompatible dimensions (%dx%d) x (%d)\n",
                m, n, n1);
    } else {
        x = (double *) malloc (n * sizeof(double));
        cg (p, id, a, b, x, n);
        print_replicated_vector (id, p, x, MPI_DOUBLE, n);
    }
    MPI_Finalize();
}

#define EPSILON 1.0e-10 /* Convergence criterion */
double *piece;         /* Temp storage allocated once */

/* Conjugate gradient method solves ax = b for x */

cg (int p, int id, double **a, double *b, double *x, int n)
{
    int i, it;          /* Loop indices */
    double *d;          /* Gradient vector */
    double *g;          /* Gradient vector */
    double denom1, denom2, num1,
        num2, s, *tmpvec; /* Temporaries */

    double dot_product (double, double, int);
    void matrix_vector_product (int, int, int, double, double *, double *);

    /* Initialize gradient vectors */

    d = (double *) malloc (n * sizeof(double));
    g = (double *) malloc (n * sizeof(double));
    tmpvec = (double *) malloc (n * sizeof(double));
    piece = (double *) malloc (BLOCK_SIZE(id,p,n) *
        sizeof(double));

```

```

    for (i = 0; i < n; i++) {
        d[i] = 0.0;
        x[i] = 0.0;
        g[i] = -b[i];
    }

    /* Algorithm converges in n or fewer iterations */

    for (it = 0; it < n; it++) {
        denom1 = dot_product (g, g, n);
        matrix_vector_product (id, p, n, a, x, g);
        for (i = 0; i < n; i++)
            g[i] -= b[i];
        num1 = dot_product (g, g, n);

        /* When g is sufficiently close to 0, time to halt */

        if (num1 < EPSILON) break;

        for (i = 0; i < n; i++)
            d[i] = -g[i] + (num1/denom1) * d[i];
        num2 = dot_product (d, g, n);
        matrix_vector_product (id, p, n, a, d, tmpvec);
        denom2 = dot_product (d, tmpvec, n);
        s = -num2 / denom2;
        for (i = 0; i < n; i++) x[i] += s * d[i];
    }
}
/*
 * Return the dot product of two vectors
 */

double dot_product (double *a, double *b, int n)
{
    int i;
    double answer;

    answer = 0.0;
    for (i = 0; i < n; i++)
        answer += a[i] * b[i];
    return answer;
}
/*
 * Compute the product of matrix a and vector b and
 * store the result in vector c.
 */

void matrix_vector_product (int id, int p, int n,
                           double **a, double *b, double *c)
{
    int i, j;
    double tmp; /* Accumulates sum */

    for (i = 0; i < BLOCK_SIZE(id,p,n); i++) {
        tmp = 0.0;
        for (j = 0; j < n; j++)
            tmp += a[i][j] * b[j];
        piece[i] = tmp;
    }
    new_replicate_block_vector (id, p, piece, n,
                                (void *) c, MPI_DOUBLE);
}

```

图 18.2 通过 MPI 实现的共轭梯度法



main 函数按照通常顺序依次调用 MPI 启动函数, 然后用两个在前面章节中开发的函数从文件中读取矩阵  $A$  和向量  $b$ 。如果矩阵  $A$  不是方块矩阵, 或者  $A$  的列向量数与  $b$  的行向量数不匹配, 则算法结束。假设矩阵为  $n \times n$  矩阵, 向量有  $n$  个元素, main 函数为所求向量  $x$  分配空间, 并调用函数 `cg` 来解方程  $Ax = b$ , 最后打印结果。

如第 12 章中所述, 函数 `cg` 是共轭梯度算法实现的核心。代码中有对向量的初始化, 加, 减等操作, 但是点积 (内积) 和矩阵向量乘都是通过函数调用实现的。共轭梯度算法是一个交互式的算法, 它连续生成与目标向量越来越相似的向量。当结果收敛或者重复了  $n$  次的时候, 算法就结束。算法通常通过远少于  $n$  次的迭代便可找到最终解。

传入两个向量参数, 函数 `dot_product` 便返回一个双精度标量值作为两个向量的点积。因为所有的向量都被复制到各个进程, 每个进程无需进行通信便可以得到计算任何一个点积的值。这个函数的时间复杂度为  $\Theta(n)$ 。

为函数 `matrix_vector_product` 传入一个矩阵和一个向量两个参数时, 它将返回这两者相乘所得到的向量。向量在各个进程间被复制, 而矩阵的行向量被划分为连续的块并按块分配到各个进程中。因此通过将矩阵的各行与这个向量相乘, 我们将在各个进程上得到结果的对应分块。这个函数的计算复杂度为  $\Theta((n^2)/p)$ 。

为了复制结果向量, 函数 `matrix_vector_product` 调用了函数 `new_replicate_block_vector`。与函数 `replicate_block_vector` 不同, 函数 `new_replicate_block_vector` 不为复制的向量分配空间, 而是通过调用函数传进了一个内存指针, 以获知被复制的向量的存储位置。用这个新的函数可以节省时间, 因为它节省了函数 `matrix_vector_product` 中复制向量到指定位置而引起的开销。复制向量中的一块可以用一个 `all-gather` 操作来实现, 其时间复杂度为  $\Theta(\log p + n)$ 。

我们看到共轭梯度方法中计算最密集的部分在矩阵向量乘积这个部分。并且, 在这个算法的 `while` 循环中惟一需要进行通信操作的地方也出现在矩阵向量乘积的求解过程中。

## 18.2.2 函数级程序轮廓刻画

我们要对整个程序进行刻画, 首先就是要发现程序中有待进一步发掘的并行性到底藏于何处。

我们在程序内部插入 `MPI_Wtime` 的调用来监测程序内部每一个主要函数 (`cg`、`dot_product` 和 `matrix_vector_product`) 的执行时间。我们记录了问题规模为 768 时, 分别在集群上的 1 个 CPU 和 8 个 CPU 上执行时每个函数所消耗的平均时间。表 18.1 所示为这一测试的结果 (图标中函数 `cg` 的执行时间中刨除了它所调用的其他函数所花费的时间)。

表 18.1 共轭梯度法在 1 个和 8 个 CPU 的机器上运行情况

函数	1 个 CPU	8 个 CPU
<code>matrix_vector_product</code>	99.55%	97.49%
<code>dot_product</code>	0.19	1.06
<code>cg</code>	0.25	1.44

事实上就像你看到的, 函数 `matrix_vector_product` 占据了所有的执行时间。这是一个

有意义的结果，因为它也是这个算法中具有最高计算复杂度的部分。这个函数应当是我们并行化工作的重点。

### 18.2.3 对函数 `matrix_vector_product` 进行并行化

函数 `matrix_vector_product` 里存在有嵌套的 `for` 循环。我们通过对最外层循环的并行化来增大程序的并行粒度。由某个  $i$  值索引的最外层循环负责计算结果向量的元素  $i$ 。在循环中会读取一些矩阵和向量的值，而待写的值只有 `tmp`、 $j$  和 `piece[i]`。当每个线程都有一个 `tmp` 和  $j$  的私有副本时，外层循环的各次迭代就可以被并行执行。

我们可以用编译指导语句 `parallel for` 来并行化由  $i$  索引的循环。这个 `for` 循环的索引变量  $i$  被默认为是私有的。正如我们曾经提到的，每个线程都要有自己的私有副本 `tmp` 和  $j$ ，所以我们声明它们为私有变量。完整的编译指导语句为：

```
#pragma omp parallel for private(j,tmp)
```

我们还可以让用户自己来指定每个进程中活动的线程个数。为此，我们在 `main` 函数中加入一个调用 `omp_set_num_threads`。它将来自于命令行。我们将这个语句放在 `main` 函数中 `MPI_Comm_rank` 调用的后面：

```
omp_set_num_threads(atoi(argv[3]));
```

这样我们仅加入了两行代码便将纯粹用 MPI 编写的程序转化为用 MPI 和 OpenMP 混合编写的程序了。

### 18.2.4 测试程序

现在让我们在一个包含有四个双处理器计算机的常用集群上来对我们的程序进行测试。首先我们运行原始程序。当它分别在 1、2、3 和 4 个进程（1~4 个 CPU）上运行时，每个进程都在不同的节点上。通过这种方式我们最大限度的利用了存储器与 CPU 间的带宽。当我们在五个 CPU 上运行 5 个进程的时候，有两个进程是在同一个节点上的。当我们在八个进程运行时，每个 CPU 上都有一个进程，也就是每个计算机（节点）上是两个进程。

现在让我们在一个商用的机群上测试我们的程序。首先我们运行该程序。当它在 1、2、3 和 4 个进程（1~4 个 CPU）上运行时，各个进程对应不同的 CPU。这样我们最大化了 CPU 和内存之间的带宽。当我们在 4 个 CPU 上运行 5 个 MPI 进程时，将有 2 个 MPI 进程共享 1 个 CPU。当我们使用 8 个进程时，进程将被分配到了各个可用的 CPU 上，每台机器对应 2 个进程。

在我们的 MPI 和 OpenMP 混合程序的测试中，我们在每台计算机上只创建一个 MPI 进程。在第一个实验中我们在一台计算机的一个进程里创建了两个线程。在第二个实验中我们在两台计算机上运行了 2 个进程，共 4 个线程。在最后一个实验中运行了 4 个进程，共 8 个线程，即每个可用 CPU 上都分配有一个线程。换句话说，我们在 2、4、6、和 8 个 CPU 上运行了 MPI 和 OpenMP 混合模式编写的程序。

图 18.3 描述了测试结果。图中的每一个时间值代表程序 5 次执行时间的平均值。用 2 个 CPU 的时候原始程序比混合程序执行速度要快。这个结果很有意义, 因为 2 个 OpenMP 线程在同一台计算机上执行。执行 OpenMP 线程的 CPU 会有更低的存储带宽和更低的高速缓存命中率。由于同样的原因, 我们预期在 4 个 CPU 上用单纯 MPI 编写的程序执行得更快, 而事实也是如此。然而, 当我们在 6 个 CPU 上实验时, 计算机开始被一对一对的 MPI 进程所共用, 上述的优点就不存在了。于是混合编程模式程序低通信开销的优势显现了出来。从最后的几个数值点可以看出, 4 个 MPI 进程, 每个进程两个线程的程序, 其执行时间比使用 8 个 MPI 进程的程序执行时间减少了 27%。

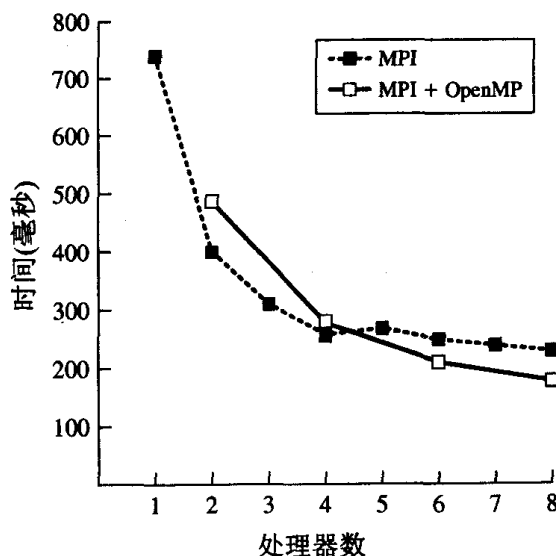


图 18.3 求解一个有 768 个方程的稠密系统的共轭梯度算法, 其原 MPI 程序和混合编写程序的测试结果。所有时间的单位都是毫秒。目标机器是一台有四个双处理器节点的商用集群

## 18.3 Jacobi 方法

### 18.3.1 MPI 程序轮廓刻画

下面我们以一个更加复杂的例子作为第二个学习范例。在第 13 章中已经介绍过, 我们用 MPI 实现了 Jacobi 方法解决恒定热量分布问题的 C 程序。为了让这个例子更易于理解, 我们选择矩阵按列分块的分解方式来表示这个有限差分网格。程序的执行分为三个阶段。函数 `initialize_mesh` 负责将矩阵的某一部分分到相对应的进程, 并且对边界条件和内部数值进行初始化。函数 `find_steady_state` 实现了求解偏微分方程的 Jacobi 方法。直至当网格点的值都收敛时迭代结束。函数 `print_solution` 将每个网格点的数值在标准输出上输出。

第一步我们将分别在由 1 个和 4 个 CPU 组成的商用集群上对并程序的执行进行刻画。测试结果在表 18.2 中进行了汇总。主要的执行时间消耗在了函数 `find_steady_state` 上。因此我们将主要针对这个函数进行并行化, 如图 18.4 所示。

表 18.2 对 Jacobi 方法 C/MPI 程序实现的特征刻画

函数	1 个 CPU	4 个 CPU
initialize_mesh	0.01%	0.03%
find_steady_state	98.48%	93.49%
print_solution	1.51%	6.48%

```

int find_steady_state (int p, int id, int my_rows,
                      double **u, double **w)
{
    double diff;
    double global_diff;
    int its;
    MPI_Status status;
    int i, j;

    its = 0;
    for (;;) {
        if (id > 0)
            MPI_Send (u[1], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD);
        if (id < p-1) {
            MPI_Send (u[my_rows-2], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD);
            MPI_Recv (u[my_rows-1], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD,
                     &status);
        }
        if (id > 0)
            MPI_Recv (u[0], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD, &status);
        diff = 0.0;
        for (i = 1; i < my_rows-1; i++)
            for (j = 1; j < N-1; j++) {
                w[i][j] = (u[i-1][j] + u[i+1][j] +
                           u[i][j-1] + u[i][j+1])/4.0;
                if (fabs(w[i][j] - u[i][j]) > diff)
                    diff = fabs(w[i][j] - u[i][j]);
            }
        for (i = 1; i < my_rows-1; i++)
            for (j = 1; j < N-1; j++)
                u[i][j] = w[i][j];
        MPI_Allreduce (&diff, &global_diff, 1, MPI_DOUBLE, MPI_MAX,
                       MPI_COMM_WORLD);
        if (global_diff <= EPSILON) break;
        its++;
    }
}

```

图 18.4 程序执行时间的 90%都为函数 find\_steady\_state 所消耗，  
我们将在这里通过多线程寻找并开发更多的并行性

### 18.3.2 对函数 find\_steady\_state 并行化

除了两段初始化代码和函数返回语句，函数 find\_steady\_state 主要由一个 for 循环构成。我们不能将这个 for 循环并行化，这是由于它并不具备规范格式：它包含一个 break 语句；并且调用了 MPI 函数。然而最重要的原因在于，循环的不同迭代间有数据相关性。每个迭代依赖于前一个迭代的计算结果。因此我们只能在外层循环迭代的内部寻找并行性。

我们将注意力转移至第一个由 i 索引的 for 循环。这个循环在其进程所拥有的矩阵的行

向量内进行迭代, 通过计算  $u$  的元素得到  $w$  的元素。这些任务的各个赋值语句间是互相独立的, 并且可以同时执行。

然而, 当  $w[i][j]$  的绝对值比当前  $diff$  的值大时, 便需要更新  $diff$  的值。如果我们希望多线程引用共享变量  $diff$ , 则需要将  $if$  语句放在一个临界区内。但是这将降低加速比。

因此我们将引入一个叫作  $tdiff$  的新的私有变量。每个线程都在运行由  $i$  索引的  $for$  循环前将自己的  $tdiff$  副本初始化为 0, 并且将  $tdiff$  与  $w[i][j]$  的每个值进行比较。因为不同的线程都是对不同的  $w$  值和私有的  $tdiff$  元素进行赋值, 我们可以用编译指导语句  $for$  来指示将  $i$  索引的  $for$  循环并行执行。

第二个由  $i$  索引的  $for$  循环把  $w$  的元素复制到  $u$  的相应的元素。我们将编译指导语句  $for$  放在这个循环的前面以指导编译器将其并行执行。

在第二个由  $i$  索引的  $for$  循环之后我们建立一个临界区, 在这个临界区内每个线程将其自己的  $tdiff$  值与公共变量  $diff$  的值进行比较, 并且在  $tdiff$  较大的时候将  $diff$  值更新为  $tdiff$  值。

注意到创建私有变量  $tdiff$  使得我们能够确立一种解决方案, 在这个方案中的各个线程里, Jacobi 算方法的每次迭代仅需进入临界区一次。如果所有线程在原始的嵌套循环中都引用  $diff$ , 每个线程中的每次迭代都将  $i*j$  次进入临界区。

我们用大括号来创建一个代码段, 这段代码中包含这两个  $for$  编译指导语句, 编译指导语句  $critical$  以及初始化私有变量  $tdiff$  的语句。在这段代码的头部我们插入编译指导语句  $parallel$ 。图 18.5 为函数 `find_steady_state` 的新版本。

```
int find_steady_state (int p, int id, int my_rows,
                      double **u, double **w)
{
    double    diff;
    double    global_diff;
    int       i, j;
    int       its;
    double    tdiff;
    MPI_Status status;

    its = 0;
    for (;;) {
        if (id > 0)
            MPI_Send (u[1], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD);
        if (id < p-1) {
            MPI_Send (u[my_rows-2], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD);
            MPI_Recv (u[my_rows-1], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD,
                     &status);
        }
        if (id > 0)
            MPI_Recv (u[0], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD, &status);
        diff = 0.0;
        #pragma omp parallel private (i, j, tdiff)
        {
            tdiff = 0.0;
            #pragma omp for
            for (i = 1; i < my_rows-1; i++)
                for (j = 1; j < N-1; j++) {
                    w[i][j] = (u[i-1][j] + u[i+1][j] +
                               u[i][j-1] + u[i][j+1])/4.0;
                    if (fabs(w[i][j] - u[i][j]) > tdiff)
                        tdiff = fabs(w[i][j] - u[i][j]);
                }
        }
    }
}
```

```

#pragma omp for nowait
    for (i = 1; i < my_rows-1; i++)
        for (j = 1; j < N-1; j++)
            u[i][j] = w[i][j];
#pragma omp critical
    if (tdiff > diff) diff = tdiff;
}
    MPI_Allreduce (&diff, &global_diff, 1, MPI_DOUBLE, MPI_MAX,
        MPI_COMM_WORLD);
    if (global_diff <= EPSILON) break;
    its++;
}
return its;
}

```

图 18.5 加入 OpenMP 编译指导语句后的 find\_steady\_state 函数

同样，为了使得用户可以通过命令行指明在每个 MPI 进程中存活的线程个数，我们改写 main 函数，在其内部加入下面一条语句：

```
omp_set_num_threads (atoi(argv[1]));
```

### 18.3.3 测试程序

在包含 4 个双处理器节点的商用集群上，我们对原始程序和对应的混合编程模式程序进行了测试。首先我们运行了仅用 MPI 实现的 C 程序。在节点间分配进程的方式与前面讲过的例子一样。当我们在 1 到 4 个 CPU 上执行程序时，每个进程分配到不同的节点以最大程度地利用存储器与 CPU 间的带宽。当我们在 5 个 CPU 上执行 5 个进程的时候，其中的两个进程被分配到同一个节点上。在 8 个进程的情况下，每个进程都被分配到 1 个 CPU 上，相当于每台计算机上分得两个进程。

在对 MPI/OpenMP 混合编程进行测试时，我们在每一个节点上仅创建一个 MPI 进程。每个进程内有两个线程。因此图中的 4 个数据点分别代表了 2、4、6 和 8 个 CPU。

图 18.6 所示为测试结果。图中的每个时间点代表这个程序 5 次执行得到的平均值。我

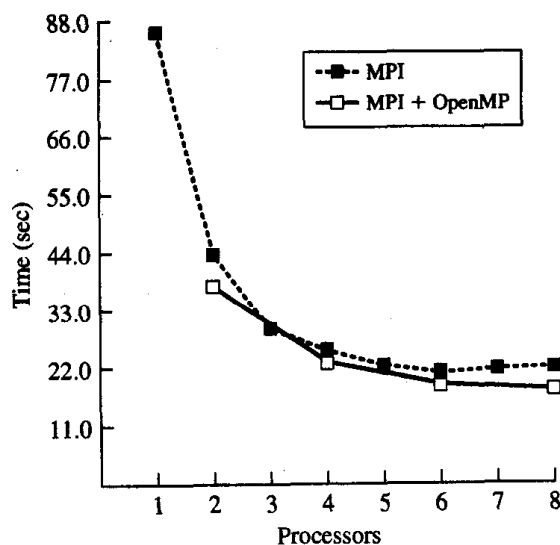


图 18.6 使用 Jacobi 方法求解规模为  $200 \times 200$  的稳态热方程，其原始的 MPI 实现与混合编程实现的程序测试结果。所有的时间值单位均为秒。目标机器是一含有 4 台双处理器计算机的商用集群

们注意到在相同的 CPU 数目下, 所有混合模式程序都比原始程序执行得快。这是因为混合编程模式的计算量-通信量比值更高。在每次元素通信中所更新的网格点数都是混合编程模式中每个节点的两倍。更低的通信开销导致了更高的加速比。在 8 个 CPU 上混合编程模式的程序比单纯使用 MPI 的并行程序要快 19%。

## 18.4 本章小结

许多当代的并行机, 包括大部分世界上最快的计算机系统, 都是由多处理器计算机集合在一起组成的。尽管在这类系统上用单纯的 MPI 编写并行程序是可行的, 但是你会经常发现用 MPI 和 OpenMP 混合编程可以得到更高的性能。MPI 可以解决多处理机间的粗粒度通信, 而 OpenMP 提供的轻量级线程可以很好地解决每个多处理器计算机内部各处理器间的交互。

在本章中我们考察了两个用 MPI 实现的 C 程序并将它们改写为适合在多处理器计算机集群上执行的混合编程模式的程序。这种转化的第一步工作是对原并行程序的轮廓进行分析, 以发现最耗时的函数。这些函数将是我们进一步并行化的工作重点。通常情况下, 不需在原始程序中加入很多函数调用或者编译指导语句, 便可将其转化为 OpenMP 和 MPI 混合编程模式下的程序。

## 18.5 练习题

18.1 在第 9 章中的文档分类函数中, 哪些最有可能适合用 OpenMP 进行并行化? (考虑程序中的所有函数, 而不光是图 9.7 中列出的那些) 请进一步证实你的结论。

18.2 假设你的学校的计算中心有一个由多个多处理器计算机节点组成的集群。进一步假设, 大部分 CPU 资源都在执行用 MPI 库编写的实现蒙特卡洛法的 C 程序。计算中心的负责人希望程序能够执行得更快。应该如何将原始程序转化为混合编程模式的程序, 从而获得性能上的显著提高? 用实验证实你的观点。

18.3 改写实现 Floyd 算法的程序 (如图 6.9 所示), 使之变成包含 OpenMP 编译指导语句的混合编程模式的程序。将线程个数设置为程序中可用处理器的个数, 对比你的程序和原始程序的加速比。

18.4 将图 8.8 中的矩阵向量相乘程序转化为包含 OpenMP 编译指导语句的混合编程模式的程序。将线程个数设置为程序中可用的处理器个数。对比你的程序和原始程序的加速比。

18.5 将图 8.14 中的矩阵向量相乘程序转化为包含 OpenMP 编译指导语句的混合编程模式的程序。将线程个数设置为程序中可用处理器的个数。对比你的程序和原始程序的加速比。

18.6 用 OpenMP 和 MPI 混合模式编写一段 C 程序, 先用高斯消去法, 然后用后替代法来求解密度系统线性方程组。用不同的  $n$  和  $p$  值来测试你的程序, 其中  $p$  值指使用的多

处理器计算机的节点个数。

18.7 编写一段混合编程模式的程序来实现规则取样并行排序。用不同的  $n$  和  $p$  值来测试你的程序，其中  $p$  值指使用的多处理器计算机的节点个数。

18.8 编写一段混合编程模式的程序来实现快速傅立叶变换。用不同的  $n$  和  $p$  值来测试你的程序，其中  $p$  值指使用的多处理器计算机的节点个数。

18.9 编写一段混合编程模式的程序来解  $n$ -皇后问题。用不同的  $n$  和  $p$  值来测试你的程序，其中  $p$  值指使用的多处理器计算机的节点个数。

18.10 编写一段混合编程模式的程序来解 15-puzzle 问题。用不同的  $n$  和  $p$  值来测试你的程序，其中  $p$  值指使用的多处理器计算机节点的个数。



# 附录 A MPI 函数

本附录介绍了 MPI-1 标准中的所有函数，其中，参数用下面 3 个标记表示：

- IN（输入参数）——调用函数将提供该值；
- OUT（输出参数）——函数本身将对这个值进行改动；
- IN/OUT（输入/输出参数）——调用者和被调用函数都可以对这个值进行修改。

```
int MPI_Abort (
    MPI_Comm comm, /* IN - Communicator */
    int error_code /* IN - Error code */
)
```

MPI\_Abort 将尽力终止通信域中的所有进程，并返回错误码。

---

```
int MPI_Address (
    void *location, /* IN - A location in 'offsets' */
    MPI_Aint *offsets /* IN - Array of addresses */
)
```

MPI\_Address 返回 location 在 offsets 中的字节地址。在构造派生数据类型的时候，此函数非常有用。

---

```
int MPI_Allgather (
    void *send_buffer, /* IN - Send buffer */
    int send_cnt, /* IN - Elements in send buffer */
    MPI_Datatype send_dtype, /* IN - Send buffer element type */
    void *recv_buffer, /* OUT - Receive buffer */
    int recv_cnt, /* IN - Elements gathered from each process */
    MPI_Datatype recv_dtype, /* IN - Receive buffer element type */
    MPI_Comm comm /* IN - Communicator */
)
```

MPI\_Allgather 是一个完成全收集（all-gather）操作的组通信函数。所有进程从通信域中的每个进程接收 send\_cnt 个元素。函数返回时，这些元素首尾相连的结果将存在于所有进程的 recv\_buffer 中。如果不同的进程提供的元素数目不同，或者元素并不按进程号的顺序首尾相联，就需要用 MPI\_Allgatherv 函数。

---

```
int MPI_Allgatherv (
    void *send_buffer, /* IN - Send buffer */
    int send_cnt, /* IN - Number of elements sent by this process */
    MPI_Datatype send_dtype, /* IN - Send buffer element type */
    void *recv_buffer, /* OUT - Receive buffer */
    MPI_Aint *recv_counts, /* IN - Number of elements received by each process */
    MPI_Datatype *recv_dtypes /* IN - Receive buffer element type */
)
```

---

```

int *recv_cnts,    /* IN - Group-sized array. Entry j is
                    number of elements to receive from process j */
int *recv_disp,    /* IN - Group-sized array. Entry j is
                    the offset from the start of
                    recv_buffer where the elements
                    received from process j should be put */
MPI_Datatype recv_dtype, /* IN - Receive buffer element type */
MPI_Comm comm      /* IN - Communicator */
)

```

**MPI\_Allgatherv** 是一个用来完成全收集操作的组通信函数。各进程提供的数据数目可能不同。**recv\_disp** 数组指出每个进程的数据应该放在 **recv\_buffer** 的位置，这些数据块并不一定要按照进程号的顺序放置。函数返回时，收集到的元素存在于每个进程的 **recv\_buffer**。如果所有进程提供相同数量的元素并且数据按照进程号的顺序放置，可以使用较为简单的 **MPI\_Allgather** 函数。

---

```

int MPI_Allreduce (
    void *send_buffer, /* IN - Send buffer */
    void *recv_buffer, /* OUT - Receive buffer. */
    int cnt,           /* IN - Number of elements to reduce */
    MPI_Datatype dtype, /* IN - Element type */
    MPI_Op op,         /* IN - Reduction operator */
    MPI_Comm comm      /* IN - Communicator */
)

```

**MPI\_Allreduce** 是一个完成 **cnt** 个归约操作的组通信函数。函数返回后，所有进程都含有归约的结果。如果只有一个进程需要归约结果，可以使用函数 **MPI\_Reduce**。

---

```

int MPI_Alltoall (
    void *send_buffer, /* IN - Send buffer */
    int send_cnt,      /* IN - Elements sent to each process */
    MPI_Datatype send_dtype, /* IN - Sent element type */
    void *recv_buffer, /* OUT - Receive buffer */
    int recv_cnt,      /* IN - Elements received from each process */
    MPI_Datatype recv_dtype, /* IN - Received element type */
    MPI_Comm comm      /* IN - Communicator */
)

```

**MPI\_Alltoall** 执行在通信域内的全交换 (all-to-all exchange) 操作。每个进程发送相同数目的元素给所有其他的进程，并从包括自身的每个进程中接收相同数目的元素。如果任何一个进程发送给任何其他进程的元素数不同，可以使用更通用的 **MPI\_Alltoallv** 函数。

---

```

int MPI_Alltoallv (
    void *send_buffer, /* IN - Send buffer */
    int *send_cnts, /* IN - Group-sized array. Entry j
                    indicates number of elements of

```

```

        send_buffer to send to process j */
int *send_disp, /* IN - Group-sized array. Entry j
                indicates the displacement from
                the start of send_buffer of the
                elements sent to process j */
MPI_Datatype send_dtype, /* IN - Send buffer element type */
void *recv_buffer, /* OUT - Receive buffer */
int *recv_cnts, /* IN - Group-sized array. Entry j is
                the number of elements being
                received from process j. */
int *recv_disp, /* IN - Group-sized array. Entry j is
                the displacement from the
                start of recv_buffer where the
                elements received from process
                j should be stored. */
MPI_Datatype recv_dtype, /* IN - Receive buffer element type */
MPI_Comm comm, /* IN - Communicator */
)

```

**MPI\_Alltoallv** 完成数据全交换。如果每个进程所提供的待交换元素的数目相同, 并且接收数据按照进程号相连接, 则可以使用简单一点的 **MPI\_Alltoall**。

```

int MPI_Attr_delete (
    MPI_Comm comm, /* IN - Communicator */
    int key /* IN - Attribute identifier */
)

```

**MPI\_Attr\_delete** 删除 key 所对应的缓冲属性。

```

int MPI_Attr_get (
    MPI_Comm comm, /* IN - Communicator */
    int key, /* IN - Attribute identifier */
    void *attr, /* OUT - Pointer to attribute */
    int *flag /* OUT - Existence flag */
)

```

**MPI\_Attr\_get** 返回指针 attr, 指向缓冲属性中具有 key 标志的属性。检索结果是否成功由 flag 的值确定。如果检索成功其值为 1, 否则为 0。

```

int MPI_Attr_put (
    MPI_Comm comm, /* IN - Communicator */
    int key, /* IN - Attributed identifier */
    void *attr /* IN - Pointer to attribute */
)

```

**MPI\_Attr\_put** 在整数 key 与 attr 所指的属性记录之间建立关联。

```

int MPI_Barrier (

```

---

```

    MPI_Comm    /* IN - Communicator */
)

```

**MPI\_Barrier** 是一个组通信函数, 用来完成指定通信域中所有进程在某一点的阻塞同步操作。

---

```

int MPI_Bcast (
    void *buffer,      /* IN/OUT - Message address */
    int cnt,           /* IN - Elements in message */
    MPI_Datatype dtype, /* IN - Element type */
    int root,          /* IN - Rank of root process */
    MPI_Comm comm      /* IN - Communicator */
)

```

**MPI\_Bcast** 是一个组通信操作, 用来完成一个进程向通信域中所有其他进程广播消息。参数 **root** 是广播消息的进程的编号。

---

```

int MPI_Bsend (
    void *buffer,      /* IN - Send buffer */
    int cnt,           /* IN - Number of elements in send buffer */
    MPI_Datatype dtype, /* IN - Type of elements in send buffer */
    int dest,          /* IN - Destination process */
    int tag,           /* IN - Message tag */
    MPI_Comm comm      /* IN - Communicator */
)

```

**MPI\_Bsend** 实现缓存模式的发送操作。如果某个进程执行缓存模式发送操作但没有相匹配的接收操作, MPI 必须将待发送的消息进行缓冲, 以便使源进程能继续执行。如果缓冲空间不足的话, 会导致错误的发生。可以通过调用 **MPI\_Buffer\_attach** 来控制可用的缓存空间的大小。

---

```

int MPI_Bsend_init (
    void *buffer,      /* IN - Send buffer */
    int cnt,           /* IN - Elements in message */
    MPI_Datatype dtype, /* IN - Type of elements */
    int dest,          /* IN - Destination process rank */
    int tag,           /* IN - Message identifier */
    MPI_Comm comm,     /* IN - Communicator */
    MPI_Request *handle /* IN - Request handle */
)

```

**MPI\_Bsend\_init** 为缓存模式发送操作创建一个持久的通信请求。当程序中反复调用参数相同的通信操作时, 此函数很有用。调用 **MPI\_Start** 才真正初始化消息发送。

---

```

int MPI_Buffer_attach (
    void *buffer, /* IN - Message buffer */
    int size      /* IN - Number of bytes in message buffer */
)

```

)

MPI\_Buffer\_attach 可以加载一定大小的内存, 以用于缓存模式下对消息进行缓冲。通常, buffer 指向调用 malloc 在堆中分配的空间。

```
int MPI_Buffer_detach (
    void *buffer_addr, /* OUT - Buffer address */
    int *size           /* OUT - Buffer size */
)
```

MPI\_Buffer\_detach 用于在缓存模式下, 将与当前消息相关的缓冲区卸载。函数返回被卸载的缓存区的地址和长度 (以字节为单位)。

```
int MPI_Cancel (
    MPI_Request *handle /* IN - Pending message handle */
)
```

MPI\_Cancel 用于取消那些未决的非阻塞发送或接收操作。此函数将立即返回, 所以真正的取消操作将在稍后发生。

```
int MPI_Cart_coords (
    MPI_Comm comm, /* IN - Cartesian communicator */
    int rank,      /* IN - Process rank */
    int dims,      /* IN - Dimensions */
    int *coords    /* OUT - Coordinates */
)
```

传递笛卡尔通信域和进程号, MPI\_Cart\_coords 返回此进程在笛卡尔坐标中的位置。

```
int MPI_Cart_create (
    MPI_Comm old_comm, /* IN - Old communicator */
    int dims,          /* IN - Grid dimensions */
    int *size,         /* IN - Array of size dims. Entry j
                        is the number of processes in dimension j. */
    int *periodic,     /* IN - Array of size dims. Entry j
                        indicates if dimension i is periodic (wraps around). */
    int reorder,       /* IN - Ranking reorder allowed */
    MPI_Comm *cart_comm /* OUT - Cartesian communicator */
)
```

MPI\_Cart\_create 返回一个通信域指针, 此通信域包含了进程集合构成的笛卡尔拓扑结构的信息。如果 reorder 为 false, 进程在新通信域中的进程号和旧通信域中的进程号相同。否则, 系统可以重排进程号以便将进程更有效地映射在处理器中。如果指定的笛卡尔网格比旧的通信域大, 那么将会导致错误发生。如果网格比现在的组小, 有些进程便会返回 MPI\_COMM\_NULL。

```
int MPI_Cart_get (
```

```

MPI_Comm comm, /* IN - Cartesian communicator */
int dims,      /* IN - Grid dimensions */
int *size,     /* OUT - Size of each dimension */
int *period,   /* OUT - Periodicity of each dimension */
int *coord     /* OUT - Coordinates of calling process */
)

```

向 `MPI_Cart_get` 传递笛卡尔通信域句柄和网格维数，它将返回每一维的大小，每一维是否有周期性（回卷）和调用此函数的进程的坐标。

```

int MPI_Cart_map (
MPI_Comm comm, /* IN - Cartesian communicator */
int dims,      /* IN - Grid dimensions */
int *size,     /* IN - Size of each grid dimension */
int *periodic, /* IN - Periodicity of each dimension */
int *new_rank  /* OUT - "Optimized" process rank */
)

```

向 `MPI_Cart_map` 传递笛卡尔通信域，网格维数，每一维的大小和每一维的周期性信息，它返回调用进程的“优化”后的进程号至 `new_rank`。

```

int MPI_Cart_rank (
MPI_Comm comm, /* IN - Cartesian communicator */
int *coords,   /* IN - Process coordinates */
int *rank      /* OUT - Process rank */
)

```

向 `MPI_Cart_rank` 传递笛卡尔通信域和进程的坐标，它将返回此进程的进程号。

```

int MPI_Cart_shift (
MPI_Comm comm, /* IN - Cartesian communicator */
int shift_dim, /* IN - Dimension of shift */
int direction, /* IN - >0 up; <0 down */
int *src,      /* OUT - Source of received message */
int *dest      /* OUT - Destination of sent message */
)

```

如果一个进程要在网格的某一维上进行发送-接收操作，那么它可以调用 `MPI_Cart_shift` 以得到消息源进程和目的进程的必要信息。如果不采取循环方式并且超出了轮换的范围，`src` 和（或）`dest` 将被置为 `MPI_PROC_NULL`。

```

int MPI_Cart_sub (
MPI_Comm comm, /* IN - Cartesian communicator */
int *free,     /* IN - Array of size dimensions.
                Entry free[i] is 1 if coord i can vary, 0 otherwise. */
MPI_Comm *new_comm /* OUT - Handle to new communicator */
)

```

**MPI\_Cart\_sub** 将笛卡尔网格分成几个维度较小的网格。新网格的维度等于 **free** 数组中值为 0 的元素的个数。函数返回一个包含调用进程在内的新通信域 **new\_comm**。

```
int MPI_Cartdim_get (
    MPI_Comm comm, /* IN - Cartesian communicator */
    int *dims      /* OUT - Dimensions */
)
```

**MPI\_Cartdim\_get** 返回一个笛卡尔通信域的维数。

```
int MPI_Comm_compare (
    MPI_Comm comm1, /* IN - First communicator */
    MPI_Comm comm2, /* IN - Second communicator */
    int *result /* OUT - Result of comparison */
)
```

**MPI\_Comm\_compare** 对两个通信域进行比较。比较的结果由 **result** 返回：如果环境上下文和其中的组是相同的，将返回 **MPI\_IDENT**；如果上下文不同，但组中的进程以及进程序号相同，将返回 **MPI\_CONGRUENT**；如果上下文和进程号不同但组中的进程相同，将返回 **MPI\_SIMILAR**；否则将返回 **MPI\_UNEQUAL**。

```
int MPI_Comm_create (
    MPI_Comm old_comm, /* IN - Old communicator */
    MPI_Group group,   /* IN - Process group */
    MPI_Comm *new_comm /* OUT - New communicator */
)
```

组函数调用 **MPI\_Comm\_create** 根据 **group** 中的进程创建新通信域。

```
int MPI_Comm_dup (
    MPI_Comm old_comm, /* IN - Old communicator */
    MPI_Comm *new_comm /* OUT - New communicator */
)
```

组函数调用 **MPI\_Comm\_dup** 复制一个通信域，两个通信域组相同但上下文不同。

```
int MPI_Comm_free (
    MPI_Comm* comm /* IN - Communicator */
)
```

组函数调用 **MPI\_Comm\_free** 释放与 **comm** 相关联的资源。

```
int MPI_Comm_group (
    MPI_Comm comm, /* IN - Communicator */
    MPI_Group *group /* OUT - Process group */
)
```

---

**MPI\_Comm\_group** 返回与 comm 关联的进程组。

---

```
int MPI_Comm_rank (
    MPI_Comm comm, /* IN - Communicator */
    int *rank /* OUT - Rank of calling process */
)
```

---

**MPI\_Comm\_rank** 返回调用进程在通信域中的进程号。

---

```
int MPI_Comm_remote_group (
    MPI_Comm comm, /* IN - Handle to inter-communicator */
    MPI_Group *procs /* OUT - Handle to remote group */
)
```

---

向 **MPI\_Comm\_remote\_group** 传递域间通信句柄，它将返回对应的远端的进程组。

---

```
int MPI_Comm_remote_size (
    MPI_Comm comm, /* IN - Handle to inter-communicator */
    int *size /* OUT - Remote group size */
)
```

---

向 **MPI\_Comm\_remote\_size** 传递域间通信句柄，它将返回对应的远端进程组中的进程数目。

---

```
int MPI_Comm_size (
    MPI_Comm comm, /* IN - Communicator */
    int *size /* OUT - Number of procs in communicator */
)
```

---

**MPI\_Comm\_size** 返回通信域中的进程数。

---

```
int MPI_Comm_split (
    MPI_Comm old_comm, /* IN - Old communicator */
    int partition, /* IN - Partition number */
    int new_rank, /* IN - Ranking value */
    MPI_Comm *new_comm /* OUT - New communicator */
)
```

---

组函数调用 **MPI\_Comm\_split** 将一个通信域 (**old\_comm**) 中的进程划分为一个或几个小进程组。**partition** 值相同的进程置于同一个小组中。在每个小组中，进程根据 **new\_rank** 的值进行编号，而 **old\_comm** 中进程的关联将被打破。此函数返回调用进程所属的新通信域的指针。

---

```
int MPI_Comm_test_inter (
    MPI_Comm comm, /* IN - Communicator */
    int *flag /* OUT - Result of test */
)
```



向 `MPI_Comm_test_inter` 传递通信域句柄 `comm`。如果 `comm` 是域间通信句柄则 `flag` 为真, 否则为假。

---

```
int MPI_Dims_create (
    int nodes, /* IN - Number of grid nodes */
    int dims, /* IN - Number of dimensions */
    int *size /* OUT - Size of each dimension */
)
```

向 `MPI_Dims_create` 传递笛卡尔网格中的节点数和维数, 它将返回一个记录着网格中每维的节点数的整数数组, 使各维的大小尽量平衡。

---

```
int MPI_Errhandler_create (
    MPI_Handler_function
    *eh_func, /* IN - Error handler function */
    MPI_Errhandler *eh /* OUT - Handle to error handler */
)
```

通过调用函数 `MPI_Errhandler_create` 用户可将 `eh_func` 注册为 MPI 异常处理函数, 并返回指向错误处理函数的指针。这是一个不透明的对象。

用户创建的错误处理函数必须是 `MPI_Handler_function` 类型的 C 函数, 定义为:

```
typedef void (MPI_Handler_function) (MPI_Comm *, int *, ...);
```

第一个参数是用到的通信域。第二个参数是导致异常的 MPI 函数返回的错误码。其他参数的类型和含义根据实现的具体情况而定。

---

```
int MPI_Errhandler_free (
    MPI_Errhandler *eh /* IN - Handle to error handler */
)
```

`MPI_Errhandler_free` 将原来与 `eh` 关联的错误处理函数与 `MPI_ERRHANDLER_NULL` 替换。

---

```
int MPI_Errhandler_get (
    MPI_Comm comm, /* IN - Communicator */
    MPI_Errhandler *eh_func /* IN - Error handler function */
)
```

`MPI_Errhandler_get` 用于获得与通信域 `comm` 相关联的错误处理函数。

---

```
int MPI_Errhandler_set (
    MPI_Comm comm, /* IN - Communicator */
    MPI_Errhandler eh /* IN - Error handler */
)
```

`MPI_Errhandler_set` 将错误处理句柄 `eh` 与通信域 `comm` 关联。

---

```
int MPI_Error_class (
```

```

    int code, /* IN - Error code */
    int *class /* OUT - Error class */
)

```

错误码的值决定于具体的 MPI 实现。错误类型是 MPI 标准的一部分。向 `MPI_Error_class` 传递错误码，它将通过第二个参数返回与之对应的错误类别。

```

int MPI_Error_string (
    int err_code, /* IN - Error code */
    char *err_string, /* OUT - Error string */
    int *err_string_length /* OUT - Length of error string */
)

```

向 `MPI_Error_string` 传递一个错误码或者错误类别，它返回与之对应的错误字符串，和字符串的长度。在调用此函数之前需为 `err_string` 分配长度至少为 `MPI_MAX_ERROR_STRING` 字节的空间。

```

int MPI_Finalize (void)

```

**MPI\_Finalize** 关闭 MPI 执行环境。所有进程在退出前必须调用此函数。

```

int MPI_Gather (
    void *send_buffer, /* IN - Send buffer */
    int send_cnt, /* IN - Elements in send buffer */
    MPI_Datatype send_dtype, /* IN - Send buffer element type */
    void *recv_buffer, /* OUT - Receive buffer */
    int recv_cnt, /* IN - Number of elements gathered from each process */
    MPI_Datatype recv_dtype, /* IN - Receive buffer element type */
    int root, /* IN - Rank of gathering process */
    MPI_Comm comm /* IN - Communicator */
)

```

**MPI\_Gather** 是一个组通信函数，它用来完成收集操作。根进程从通信域中的所有进程（包括自身）处各接收 `send_cnt` 个元素。函数返回的结果包含有接收到的所有元素的连接，它将被存放在 `recv_buffer` 中。如果各进程提供的元素个数不同或者元素不按进程号的次序相连接，则可以使用 **MPI\_Gatherv**。

```

int MPI_Gatherv (
    void *send_buffer, /* IN - Send buffer */
    int send_cnt, /* IN - Elements in send buffer */
    MPI_Datatype send_dtype, /* IN - Send buffer element type */
    void *recv_buffer, /* OUT - Address of receive buffer */
    int *recv_cnt, /* IN - Elements to gather from each process */
    int *displacements, /* IN - Displacement in recv_buffer
                        of elements gathered from each process */
    MPI_Datatype recv_dtype, /* IN - Receive buffer element type */
)

```

```

    int root,                /* IN - Rank of gathering process */
    MPI_Comm comm            /* IN - Communicator */
)

```

**MPI\_Gatherv** 是完成收集操作的组通信函数。根进程从进程 *i* 收集(包括自身) **send\_cnt[i]** 个元素。它把从进程 *i* 收集到的元素连续地放入从 **recv\_buffer** 的第 **displacements[i]** 个位置开始的地方。函数返回时, **recv\_buffer** 将存放有这些元素的连接。如果各进程提供的元素个数相同且元素按进程号连接, 可以使用较为简单的 **MPI\_Gather** 函数。

```

int MPI_Get_count (
    MPI_Status *status, /* IN - Result of receive */
    MPI_Datatype dtype, /* IN - Type of elements received */
    int* cnt            /* OUT - Count of elements received */
)

```

向 **MPI\_Get\_count** 传递一个包含有接收操作结果信息的句柄和接收元素的类型信息, 它将返回实际接收到的元素(不按字节计算)的个数。

```

int MPI_Get_elements (
    MPI_Status *status, /* IN - Result of receive */
    MPI_Datatype dtype, /* IN - Type of elements received */
    int* pe_cnt         /* OUT - Count of elements received */
)

```

向 **MPI\_Get\_elements** 传递一个与接收操作相关联的状态句柄和接收元素的类型信息, 它将返回实际收到的元素个数。

```

int MPI_Get_processor_name (
    char *name, /* OUT - Processor name */
    int *length /* OUT - Length of processor name */
)

```

**MPI\_Get\_processor\_name** 返回调用进程所在处理器的主机名。调用此函数之前要为 **name** 分配空间, 长度应为 **MPI\_MAX\_PROCESSOR\_NAME**。

```

int MPI_Get_version (
    int *major, /* OUT - Major version number (1 or 2) */
    int *minor /* OUT - Minor version number */
)

```

**MPI\_Get\_version** 返回 MPI 主版本号 and 次版本号。

```

int MPI_Graph_create (
    MPI_Comm old_comm, /* IN - Old communicator */
    int n, /* IN - Nodes in process graph */
    int *degree, /* IN - Array of size n with vertex degree
                  stored indirectly. Entry 0 is

```

```

        degree of vertex 0. For all other
        vertices i, degree[i]-degree[i-1]
        is degree of vertex i. */
    int *edge, /* IN - Array with rest of edge info. Entry
                i is destination of edge i. */
    int reorder, /* IN - Ranks changeable (logical) */
    MPI_Comm *graph_comm /* OUT - Graph communicator */
)

```

**MPI\_Graph\_create** 返回一个指向新通信域的指针。这个通信域包含了由一组进程形成的有向图的信息。如果 **reorder** 为 **false**，进程号则不可更改。否则，该函数将修改进程的编号以提高效率。每条有向边的起点由 **degree** 数组决定，终点在 **edges** 数组中。

```

int MPI_Graph_get (
    MPI_Comm comm, /* IN - Graph communicator */
    int n,          /* IN - Number of vertices in graph */
    int m,          /* IN - Number of edges in graph */
    int *index,     /* OUT - Index information */
    int *edge       /* OUT - Edge information */
)

```

向 **MPI\_Graph\_get** 传递具有图拓扑的通信域，以及此通信域中的顶点数（进程数）和边数（两个进程间的联系），它将返回数组 **index** 和 **edge** 以表示整个图的结构。图结构见 **MPI\_Graph\_create** 中的描述。

```

int MPI_Graph_map (
    MPI_Comm comm, /* IN - Graph communicator */
    int n,         /* IN - Vertices in graph */
    int *index,    /* IN - Index information */
    int *edge,     /* IN - Edge information */
    int *new_rank  /* OUT - New rank of process */
)

```

给定图通信域内的连接关系的联系，组函数 **MPI\_Graph\_map** 将试图优化进程在各个处理器中的分布。最后一个参数返回调用进程的新进程号。如果调用进程不是图通信域的一部分则返回 **MPI\_UNDEFINED**。

```

int MPI_Graph_neighbors (
    MPI_Comm comm, /* IN - Graph communicator */
    int rank,      /* IN - Process rank */
    int max_neighbors, /* IN - Max number of neighbors */
    int *neighbors /* OUT - Ranks of neighbors */
)

```

**MPI\_Graph\_neighbors** 通过参数表中的最后一个参数返回某个进程邻居进程的编号（此

邻居列表是用来创建图通信域的 edge 数组的一部分)。

```
int MPI_Graph_neighbors_count (
    MPI_Comm comm, /* IN - Graph communicator */
    int rank,      /* IN - Process rank */
    int *neighbors /* OUT - Number of neighbors */
)
```

**MPI\_Graph\_neighbors\_count** 返回指定通信域中某个指定进程的邻居数目。

```
int MPI_Graphdims_get (
    MPI_Comm comm, /* IN - Graph communicator */
    int *vertices, /* OUT - Vertices in the graph */
    int *edges     /* OUT - Edges in the graph */
)
```

向 **MPI\_Graphdims\_get** 传递一个图通信域，它将返回通信域中的顶点数（进程数）以及这些进程间的有向边的条数。

```
int MPI_Group_compare (
    MPI_Group group1, /* IN - First process group */
    MPI_Group group2, /* IN - Second process group */
    int *result       /* OUT - Result of comparison */
)
```

**MPI\_Group\_compare** 对两个进程组进行比较。如果两个组中的进程和进程号都相同，则返回 **MPI\_IDENT**；如果两组进程相同但进程号不同，则返回 **MPI\_SIMILAR**；否则返回 **MPI\_UNEQUAL**。

```
int MPI_Group_difference (
    MPI_Group group1, /* IN - First group */
    MPI_Group group2, /* IN - Second group */
    MPI_Group *group_diff /* OUT - Difference */
)
```

传递两个进程组作为参数，**MPI\_Group\_difference** 将产生一个新进程组，其中包含所有在第一个进程组中而不在第二个进程组中的进程。进程的编号与其在第一组中的编号相同。

```
int MPI_Group_excl (
    MPI_Group group, /* IN - Existing process group */
    int excl_num,    /* IN - Number of processes to exclude */
    int *excl_ranks, /* IN - Ranks of excluded processes */
    MPI_Group *new;  /* OUT - New group */
)
```

**MPI\_Group\_excl** 从一个已有的组中排除某些特定进程号的进程并创建新进程组。新组

中的进程编号与各自在原先组中的编号相同。

```
int MPI_Group_free (
    MPI_Group *group /* IN - Process group */
)
```

**MPI\_Group\_free** 试图将一个组内的对象释放并将该组的句柄改写为 **MPI\_GROUP\_NULL**。当的确知道所有用到此组的操作全部结束后，组内的对象才会最终释放。

```
int MPI_Group_incl (
    MPI_Group old; /* IN - Existing process group */
    int new_size, /* IN - Number of procs in new group */
    int *old_ranks, /* IN - Order of procs in new group */
    MPI_Group *new; /* OUT - New process group */
)
```

**MPI\_Group\_incl** 从一个已有进程组中创建新的进程组。新组可能比旧组小，其大小由参数 **new\_size** 决定。只有在 **old\_ranks** 中标出的进程才会在新组中出现。**old\_ranks** 中的进程的编号决定了各个进程在新组中的编号。**old\_ranks** 中的第一个进程在新组中的进程号为 0。

```
int MPI_Group_intersection (
    MPI_Group group1, /* IN - Group 1 */
    MPI_Group group2, /* IN - Group 2 */
    MPI_Group *new_group /* OUT - Intersection group */
)
```

**MPI\_Group\_intersection** 根据传递的两个进程组取得它们的交集以产生一个新的进程组。新进程组中各个进程的编号与它们在第一个组中的编号相同。

```
int MPI_Group_range_excl (
    MPI_Group group, /* IN - Existing process group */
    int n, /* IN - Ranges to evaluate */
    int range[][3], /* IN - Ranges of processes to exclude */
    MPI_Group *new /* OUT - New process group */
)
```

**MPI\_Group\_range\_excl** 根据一个旧组和  $n$  个范围限定符产生一个新组。每个范围限定符由一个起始进程号，终止进程号和步长组成。例如，{5,11,3}表示进程号为 5、8 和 11 的进程。新组将包含旧组中所有出现在这些范围内的进程。进程的编号与它们在旧组中的编号相同。

```
int MPI_Group_range_incl (
    MPI_Group old, /* IN - Existing process group */
    int n, /* IN - Ranges to evaluate */
    int range[][3], /* IN - Ranges of processes to include */

```

```

    MPI_Group *new    /* OUT - New process group */
)

```

**MPI\_Group\_range\_incl** 根据一个旧组和  $n$  个范围限定符产生一个新组。每个范围限定符由一个起始进程号, 终止进程号和步长组成。例如, {5,11,3} 表示进程号为 5,8,11 的进程。新组将包含旧组中所有出现在这些范围内的进程。进程的编号与它们在旧组中的编号相同。

```

int MPI_Group_rank (
    MPI_Group group, /* IN - Process group */
    int *rank        /* OUT - Rank of process */
)

```

向 **MPI\_Group\_rank** 传递一个进程组句柄, 它将返回调用进程在那个组中的编号。

```

int MPI_Group_size (
    MPI_Group group, /* IN - Process group */
    int *size /* OUT - Size of group */
)

```

向 **MPI\_Group\_size** 传递一个进程组句柄, 它返回该组中的进程数。

```

int MPI_Group_translate_ranks (
    MPI_Group group1, /* IN - First group */
    int n,            /* IN - Number of ranks to compare */
    int *rank1,       /* IN - Valid ranks in first group */
    MPI_Group group2, /* IN - Second group */
    int *rank2        /* OUT - Ranks in second group */
)

```

**MPI\_Group\_translate\_ranks** 可以确定一个组的各个进程在另一个组中的进程号。参数  $n$  表示需要进行比较的次数。数组 **rank1** 列举了 **group1** 的进程号。函数通过 **rank2** 返回这些进程在 **group2** 中相应的进程号。例如, 假设 **group1** 中进程号为 3 的进程在 **group2** 的进程号为 4, 那么如果 **rank1[i]=3**, 将有 **rank2[i]=4**。

```

int MPI_Group_union (
    MPI_Group group1, /* IN - First group */
    MPI_Group group2, /* IN - Second group */
    MPI_Group *new_group /* OUT - Union of two groups */
)

```

**MPI\_Group\_union** 返回由前两个参数确定的进程组的并集。进程按照先第一组进程, 而后第二组中不在第一组出现的进程的顺序进行编号。

```

int MPI_IbSEND (
    void *buffer, /* IN - Message buffer */
    int cnt,      /* IN - Elements in message */
    MPI_Datatype dtype, /* IN - Element type */

```

---

```

    int dest,          /* IN - Rank of destination process */
    int tag,           /* IN - Message identifier */
    MPI_Comm comm,     /* IN - Communicator */
    MPI_Request *handle /* OUT - Handle to request */
)

```

**MPI\_Ibsend** 完成对非阻塞缓冲式发送过程的初始化。可以用 **request** 句柄查询发送状态或者等待其结束。由于操作是非阻塞式的，因而在发送未完成前不要访问消息缓冲区。

---

```

int MPI_Init (
    int *argc, /* IN - First parameter to function main */
    char ***argv /* IN - Second parameter to function main */
)

```

**MPI\_Init** 对并行环境进行初始化。MPI 程序必须在调用其他 MPI 函数前调用 **MPI\_Init**。(惟一的例外是 **MPI\_Initialized**)。多次调用 **MPI\_Init** 是错误的做法。同时，我们向它传递 **main** 函数中的 **argc** 和 **argv** 指针。

---

```

int MPI_Init_thread (
    int *argc, /* IN - First parameter to function main */
    char ***argv, /* IN - Second parameter to function main */
    int desired, /* IN - Desired level of thread support */
    int *provided /* OUT - Provided level of thread support */
)

```

**MPI\_Init\_thread** 用 **MPI\_Init** 同样的方式初始化 MPI，另外它还初始化线程环境。如果调用了 **MPI\_Init\_thread**，我们就可以省略对 **MPI\_Init** 的调用。前两个参数与 **MPI\_Init** 相同。第三个参数是期望得到对线程的支持层次：

- (1) **MPI\_THREAD\_SINGLE**——单线程执行。
  - (2) **MPI\_THREAD\_FUNNELED**——如果进程是多线程的，只有主线程可以调用 MPI 函数。
  - (3) **MPI\_THREAD\_SERIALIZED**——多个线程都可以调用 MPI 函数，但这些调用是顺序的。
  - (4) **MPI\_THREAD\_MULTIPLE**——多个线程可以同时调用 MPI 函数。
- 更大的数代表更高的支持层次。函数通过最后一个参数返回系统所提供的线程支持层次。MPI 的具体实现并不一定要对线程提供支持。
- 

```

int MPI_Initialized (
    int* flag /* OUT - Indicates if MPI has been initialized */
)

```

向 **MPI\_Initialized** 传递一个整数指针，如果 **MPI\_Init** 已被调用过，则此整数被置为真，否则为假。它是惟一可在 **MPI\_Init** 调用之前调用的 MPI 函数。

---

```

int MPI_Intercomm_create (

```



```

MPI_Comm local_comm,      /* IN - Local communicator */
int local_leader,         /* IN - Rank of local "leader" */
MPI_Comm remote_comm,    /* IN - Remote communicator */
int remote_leader,       /* IN - Rank of remote "leader" */
int tag,                 /* IN - Intercomm identifier */
MPI_Comm *new_comm       /* OUT - Inter-communicator */
)

```

组函数 `MPI_Intercomm_create` 根据一个已有的本地通信域(此进程是该通信域的成员)和一个远程通信域(此进程不是其成员)创建一个新的组间通信域。`tag` 用来消除同时创建多个组间通信域时与组间通信域相关联的消息歧义。它所生成的组间通信域将仍有“本地”和“远程”的概念。至少,每个组中的进程都可以与其他进程通信。

```

int MPI_Intercomm_merge (
    MPI_Comm inter, /* IN - Handle to inter-communicator */
    int high,       /* IN - "High" group indicator */
    MPI_Comm *intra /* OUT - Handle to intracommunicator */
)

```

组函数调用 `MPI_Intercomm_merge` 将组间通信域转化为组内通信域。组间通信域中某一组的所有进程需将 `high` 设为 `true`, 而另一组将 `high` 设为 `false`。在决定进程在新通信域中的进程号的时候,系统将“低”进程排在“高”进程之前。

```

int MPI_Iprobe (
    int src,           /* IN - Rank of sending process */
    int tag,          /* IN - Incoming message tag */
    MPI_Comm comm,    /* IN - Communicator */
    int *flag,        /* OUT - Success flag */
    MPI_Status *status /* OUT - Pointer to status object */
)

```

`MPI_Iprobe` 是非阻塞函数,它将非阻塞地检查消息是否到达但不接收该消息。函数返回时,如果特定来源和特定标志的消息可以被接收则 `flag` 为真,否则 `flag` 为假。如果 `flag` 为真,有关消息的信息可以通过状态指针获得。如果要根据接收消息的大小针对性地分配接收缓冲区,此函数将非常有用。`MPI_ANY_SOURCE` 允许探测来自任何源进程的消息,`MPI_ANY_TAG` 允许探测包含任何标志的消息。如果程序需要阻塞地探测消息是否到达,可以用 `MPI_Probe`。

```

int MPI_Irecv (
    void *buffer,      /* OUT - Address of receive buffer */
    int cnt,          /* IN - Elements to receive */
    MPI_Datatype dtype, /* IN - Type of message elements */
    int src,          /* IN - Source process of message */
    int tag,          /* IN - Message ID */
    MPI_Comm comm,    /* IN - Communicator */
)

```

---

```

    MPI_Request *handle /* OUT - Request handle */
)

```

**MPI\_Irecv** 实现了非阻塞方式，或者说是立即方式的接收。它向运行系统发出接收请求，然后立即返回调用函数。除非已经调用 **MPI\_Wait** 完成了接收操作，否则不要访问接收缓冲区。

---

```

int MPI_Irsend (
    void *buffer,          /* IN - Message buffer */
    int cnt,               /* IN - Elements in message */
    MPI_Datatype dtype,    /* IN - Element type */
    int dest,              /* IN - Rank of destination process */
    int tag,               /* IN - Message identifier */
    MPI_Comm comm,         /* IN - Communicator */
    MPI_Request *handle /* OUT - Handle to request */
)

```

**MPI\_Irsend** 对一个非阻塞式的就绪发送进行初始化。用请求句柄可查询发送状态或者等待其结束。由于此调用是非阻塞式的，在发送完成之前不要访问消息缓冲区。

---

```

int MPI_Isend (
    void *buffer,          /* IN - Message buffer */
    int cnt,               /* IN - Elements in message */
    MPI_Datatype dtype,    /* IN - Type of elements */
    int dest,              /* IN - Destination process */
    int tag,               /* IN - Message identifier */
    MPI_Comm comm,         /* IN - Communicator */
    MPI_Request *handle /* OUT - Request handle */
)

```

**MPI\_Isend** 是一个非阻塞式的，即立即式的通信函数。它向运行系统发出通信请求后立即返回调用过程，并初始化一个不透明的对象指针。此指针包含了该未完成发送的信息。必须通过将此句柄传递给其他函数，如 **MPI\_Wait**，以完成发送操作。直到那时，才可以修改发送缓冲区的内容。

---

```

int MPI_Issend (
    void *buffer,          /* IN - Message buffer */
    int cnt,               /* IN - Elements in message */
    MPI_Datatype dtype,    /* IN - Element type */
    int dest,              /* IN - Rank of destination process */
    int tag,               /* IN - Message identifier */
    MPI_Comm comm,         /* IN - Communicator */
    MPI_Request *handle /* OUT - Handle to request */
)

```

**MPI\_Issend** 对一个同步模式的立即（非阻塞）发送操作进行初始化。直到相应的接收

操作开始之后, 调用函数才会返回。通过请求句柄我们可以获得发送操作的状态或者等待其结束。由于此调用是非阻塞式的, 所有在发送完成之前不要访问消息缓冲区。

```
int MPI_Keyval_create (
    MPI_Copy_function *copy_fn, /* IN - Ptr to copy attribute function */
    MPI_Delete_function *del_fn, /* IN - Ptr to delete attribute func */
    int *key, /* OUT - Ptr to attribute key */
    void *extra /* IN - Extra info for callbacks */
)
```

**MPI\_Keyval\_create** 创建一个可由 key 值索引的新属性键。

```
int MPI_Keyval_free (
    int *keyval /* IN - Key value */
)
```

**MPI\_Keyval\_free** 试图释放一个由整数 key 关联的属性键, 并将 keyval 的值置为 MPI\_KEYVAL\_INVALID。

```
int MPI_Op_create (
    MPI_User_function *assoc_func, /* IN - Associative function */
    int commutative, /* IN - Commutativity flag (logical) */
    MPI_Op *op /* OUT - Op handle */
)
```

你可以定义自己的全局归约操作并用函数 **MPI\_Op\_create** 将其绑定至一个操作句柄, 以在函数 **MPI\_Reduce**, **MPI\_Allreduce**, **MPI\_Reduce\_scatter** 和 **MPI\_Scan** 中使用。该全局操作必须满足结合率。如果它也满足交换率, 那么需将第二个参数置为 true。函数通过第三个参数返回 op 句柄。

全局操作函数的 ANSI-C 原型为:

```
typedef void MPI_User_function(void *in_vector,
    void *in_out_vector, int *length, MPI_Datatype *dtype);
```

调用函数时,  $u[0], u[1], \dots, u[length-1]$  表示 in\_vector 的元素,  $v[0], v[1], \dots, v[length-1]$  表示 in\_out\_vector 的元素;  $w[0], w[1], \dots, w[length-1]$  表示函数返回时 in\_out\_vector 中的元素。最后, 用  $\oplus$  表示满足结合率的操作。那么, w 的元素可通过如下方式得到:  $w[i] = u[i] \oplus v[i]$ , 其中  $i < length$ 。也就是说, 函数的结果覆盖了 in\_out\_vector 的值。

```
int MPI_Op_free (
    MPI_Op *op /* IN - Handle to a user-defined operation */
)
```

**MPI\_Op\_free** 试图将一个用户定义的操作释放掉, 它将 op 赋值为 MPI\_OP\_NULL。

```
int MPI_Pack (
    void *in_buffer, /* IN - Original message buffer */
    int elements, /* IN - Elements in message buffer */
    MPI_Datatype *dtype, /* IN - Data type of message buffer */
    void *out_buffer, /* OUT - Packed message buffer */
    int *out_elements, /* OUT - Elements in packed message buffer */
    MPI_Comm comm /* IN - Communication handle */
)
```

```

MPI_Datatype dtype, /* IN - Type of elements */
void *out_buffer,   /* OUT - Packed message buffer */
int out_size,       /* IN - Bytes in out_buffer */
int *offset,        /* IN/OUT - Index in out_buffer where
                    packing starts/ends */
MPI_Comm comm /* IN - Communicator used in subsequent send */
)

```

**MPI\_Pack** 可将不连续数据在发送前打包到一块连续的缓冲区中。消息被接收后也必须被解包。打包和解包的一种替代方法是使用派生的数据类型。打包和解包的另一个作用是可以避免用到系统的缓冲机制。

```

int MPI_Pack_size (
    int cnt,           /* IN - Elements in message */
    MPI_Datatype dtype, /* IN - Element type */
    MPI_Comm comm,     /* IN - Communicator */
    int *bound         /* OUT - Upper bound on packed message size */
)

```

**MPI\_Pack\_size** 按字节计算被打包消息所占的空间。在 **MPI\_Pack** 之前调用此函数，可以确定需为打包消息开辟多大的缓冲区。

```

int MPI_Probe (
    int src,           /* IN - Rank of message source */
    int tag,           /* IN - Incoming message tag */
    MPI_Comm comm,     /* IN - Communicator */
    MPI_Status *status /* OUT - Pointer to status object */
)

```

**MPI\_Probe** 用来探测某个消息是否到达，但它并不接收该消息。如果要根据接收消息的大小分配接收缓冲区此函数非常有用。**MPI\_ANY\_SOURCE** 允许探测来自任何进程的消息，**MPI\_ANY\_TAG** 允许探测携带有任何标志的消息。此函数将一直阻塞直至与各个参数均匹配的消息到来。如果要非阻塞地探测消息是否已经到来，可以使用 **MPI\_Iprobe**。

```

int MPI_Recv (
    void* buffer,      /* OUT - Receive buffer */
    int cnt,           /* IN - Max number of elements to receive */
    MPI_Datatype dtype, /* IN - Type of message elements */
    int src,           /* IN - Source process of message */
    int tag,           /* IN - Message ID */
    MPI_Comm comm,     /* IN - Communicator */
    MPI_Status *status /* OUT - Result of receive */
)

```

**MPI\_Recv** 实现阻塞式接收。如果接收成功，则当程序从 **MPI\_Recv** 返回时，**buffer** 中

含有接收到的消息。

```
int MPI_Recv_init (
    void *buffer,          /* OUT - Receive buffer */
    int cnt,              /* IN - Max number of elements to receive */
    MPI_Datatype dtype,   /* IN - Type of element */
    int src,              /* IN - Rank of message source */
    int tag,              /* IN - Message identification */
    MPI_Comm comm,        /* IN - Communicator */
    MPI_Request *handle /* OUT - Request handle */
)
```

**MPI\_Recv\_init** 为标准模式接收操作创建一个持久的通信请求。当程序中反复调用参数相同的接收操作时此函数很有用。我们可以通过调用 **MPI\_Start** 来开始真正的消息接收过程。

```
int MPI_Reduce (
    void *send_buffer, /* IN - Send buffer */
    void *recv_buffer, /* OUT - Receive buffer. Only root
                        process gets results. */
    int cnt,          /* IN - Number elements to reduce */
    MPI_Datatype dtype, /* IN - Element type */
    MPI_Op op,        /* IN - Reduction operator */
    int root,         /* IN - Rank of root process */
    MPI_Comm comm     /* IN - Communicator */
)
```

**MPI\_Reduce** 是组通信函数，用来完成 cnt 个归约。函数返回时，进程号为 root 的进程将拥有归约的结果。如果想让所有进程都拥有归约的结果，请使用 **MPI\_Allreduce**。

```
int MPI_Reduce_scatter (
    void *send_buffer, /* IN - Send buffer */
    void *recv_buffer, /* OUT - Receive buffer */
    int *recv_cnts, /* IN - Group-sized array. Entry j
                    is the number of result elements to send process j. */
    MPI_Datatype dtype, /* IN - Element type */
    MPI_Op op, /* IN - MPI reduction operator */
    MPI_Comm comm /* IN - Communicator */
)
```

**MPI\_Reduce\_scatter** 是组通信函数，完成归约之后将对结果元素进行散发操作。归约的元素数目是 **recv\_cnts** 数组所有元素值的和。

```
int MPI_Request_free (
    MPI_Request *handle /* IN - Message request handle */
)
```

`MPI_Request_free` 用于请求释放与持久通信关联的 `MPI_Request` 对象。任何与此请求对象关联的通信将在系统释放此对象前结束。

```
int MPI_Rsend (
    void *buffer,          /* IN - Message buffer */
    int cnt,               /* IN - Elements in message */
    MPI_Datatype dtype,    /* IN - Type of elements */
    int dest,              /* IN - Destination process */
    int tag,               /* IN - Message identifier */
    MPI_Comm comm          /* IN - Communicator */
);
```

`MPI_Rsend` 实现就绪模式的发送操作。在就绪模式中，只有与其对应的接收操作开始之后，发送才能启动。如果匹配的接收操作还未开始，便会产生错误。就绪模式发送操作在某些系统中可以消除握手操作而提高性能。函数返回后，发送缓冲区可被再用。

```
int MPI_Rsend_init (
    void *buffer,          /* IN - Send buffer */
    int cnt,               /* IN - Elements in message */
    MPI_Datatype dtype,    /* IN - Type of elements */
    int dest,              /* IN - Rank of destination */
    int tag,               /* IN - Message identifier */
    MPI_Comm comm,         /* IN - Communicator */
    MPI_Request *handle    /* OUT - Request handle */
);
```

`MPI_Rsend_init` 为就绪模式发送操作创建一个持久的通信请求。当程序中反复调用参数相同的通信操作时此函数很有用。`MPI_Start` 用于开启真正的消息发送操作。

```
int MPI_Scan (
    void *send_buffer,     /* IN - Send buffer */
    void *recv_buffer,     /* OUT - Receive buffer */
    int cnt,               /* IN - Size of send buffer */
    MPI_Datatype dtype,    /* IN - Type of sent elements */
    MPI_Op op,             /* IN - MPI operator */
    MPI_Comm comm          /* IN - Communicator */
);
```

`MPI_Scan` 是一个并行前缀操作符。函数返回后，`recv_buffer` 中的每个元素都是将 `MPI` 操作符 `op` 作用于通信域内与之具有相同或者较小编号的相似进程中 `send_buffer` 的对应元素所得到的结果。

```
int MPI_Scatter (
    void *send_buffer,     /* IN - Send buffer */
    int send_cnt,          /* IN - Elements sent each process */
    MPI_Datatype send_dtype, /* IN - Type of sent elements */
    MPI_Comm comm          /* IN - Communicator */
);
```

```

void *recv_buffer,      /* OUT - Address of receive buffer */
int recv_cnt, /* IN - Number of elements this process receives */
MPI_Datatype recv_dtype, /* IN - Type of received elements */
int root,          /* IN - Rank of sending process */
MPI_Comm comm      /* IN - Communicator */
)

```

**MPI\_Scatter** 是用于完成散发操作的组通信函数：根进程上的一组元素被均分成数块，每一块被发送到通信域的一个进程去（包括根进程）。如果各个分块的大小不等，则可用更一般的 **MPI\_Scatterv**。

```

int MPI_Scatterv (
    void *send_buffer, /* IN - Send buffer */
    int *send_cnts, /* IN - Number of elements to send to each process */
    int *send_disp, /* IN - Element i is the offset in
                     send_buffer of the first data
                     element going to process i */
    MPI_Datatype send_dtype, /* IN - Type of sent elements */
    void *recv_buffer, /* OUT - Receive buffer */
    int recv_cnt, /* IN - Number of elements this process will receive */
    MPI_Datatype recv_dtype, /* IN - Type of received elements */
    int root, /* IN - Rank of sending process */
    MPI_Comm comm /* IN - Communicator */
)

```

**MPI\_Scatterv** 是完成散发的组通信函数：根进程上的一组元素被均分成数块，每一块发送到通信域的一个进程去（包括根进程）。**send\_cnts** 数组指出发送到每个进程的元素数，**send\_disp** 数组指定给每个进程的块在 **send\_buffer** 中的偏移量。如果元素分块大小相等而且按进程顺序发送，则可使用 **MPI\_Scatter**。

```

int MPI_Send (
    void *buffer,      /* IN - Message buffer */
    int cnt,          /* IN - Elements in message */
    MPI_Datatype dtype, /* IN - Type of elements */
    int dest,         /* IN - Destination process */
    int tag,          /* IN - Message identifier */
    MPI_Comm comm     /* IN - Communicator */
)

```

**MPI\_Send** 用于完成阻塞式发送操作：当函数返回时，消息缓冲区可被立即重用。**MPI** 运行系统或者将消息复制到系统缓冲区，或者将其直接复制到相应的接收缓冲区。如果运行系统不对输出消息进行缓冲，那么 **MPI\_Send** 直到消息发送到接收进程之后才会返回。

```

int MPI_Send_init (
    void *buffer,      /* IN - Send buffer */
    int cnt,          /* IN - Elements in message */

```

```

MPI_Datatype dtype, /* IN - Type of elements */
int dest,           /* IN - Rank of destination */
int tag,            /* IN - Message identifier */
MPI_Comm comm,      /* IN - Communicator */
MPI_Request *handle /* OUT - Request handle */
)

```

**MPI\_Send\_init** 为标准模式发送操作创建一个持久的通信请求。当程序中反复调用参数相同的通信操作时此函数很有用。**MPI\_Start** 用于启动真正的消息发送过程。

```

int MPI_Sendrecv (
    void *send_buffer, /* IN - Send buffer */
    int send_cnt,      /* IN - Elements to send */
    MPI_Datatype send_dtype, /* IN - Outgoing message element type */
    int dest, /* IN - Destination process for outgoing message */
    int send_tag, /* IN - Outgoing message ID */
    void *recv_buffer, /* OUT - Receive buffer */
    int recv_cnt,      /* IN - Elements to receive */
    MPI_Datatype recv_dtype, /* IN - Incoming message element type */
    int src, /* IN - Source process of incoming message */
    int recv_tag, /* IN - Incoming message ID */
    MPI_Comm comm, /* IN - Communicator */
    MPI_Status *status /* OUT - Status of received message */
)

```

**MPI\_Sendrecv** 把向一个进程发送消息和从那个进程或其他进程接收消息的操作综合在一个函数里。当进程形成环状而且每个进程都向邻居发送消息时，它将非常有用。如果使用普通的发送接收操作，就需要仔细安排以免发生死锁。用 **MPI\_Sendrecv** 可替换一对 **send-receive** 调用同时保证不会发生死锁。**MPI\_Sendrecv** 是阻塞的发送和接收操作。发送缓冲区和接收缓存区必须分离。如果想让二者使用同一个缓冲区，可以用 **MPI\_Sendrecv\_replace**。

```

int MPI_Sendrecv_replace (
    void *buffer, /* IN/OUT - Message buffer */
    int cnt,      /* IN - Elements in message */
    MPI_Datatype dtype, /* IN - Type of message elements */
    int dest, /* IN - Destination process */
    int send_tag, /* IN - Sent message ID */
    int src, /* IN - Source process */
    int recv_tag, /* IN - Received message ID */
    MPI_Comm comm, /* IN - Communicator */
    MPI_Status *status /* OUT - Received message status */
)

```

**MPI\_Sendrecv\_replace** 完成阻塞式发送-接收操作。它与 **MPI\_Sendrecv** 很像，但包含发



送消息的缓冲区正是接收消息存放的地方。发送和接收消息的长度必须相同。

```
int MPI_Ssend (
    void *buffer,          /* IN - Send buffer */
    int cnt,               /* IN - Number of elements to send */
    MPI_Datatype dtype,    /* IN - Type of elements */
    int dest,              /* IN - Destination process */
    int tag,               /* IN - Message tag */
    MPI_Comm comm          /* IN - Communicator */
)
```

**MPI\_Ssend** 实现同步模式发送操作, 只有当匹配接收操作已经启动而且开始接收消息后, 发送函数才能返回。**MPI\_Ssend** 的结束标志着发送缓冲区可被重用。

```
int MPI_Ssend_init (
    void *buffer,          /* IN - Send buffer */
    int cnt,               /* IN - Elements in message */
    MPI_Datatype dtype,    /* IN - Type of elements */
    int dest,              /* IN - Rank of destination */
    int tag,               /* IN - Message identifier */
    MPI_Comm comm,         /* IN - Communicator */
    MPI_Request *handle    /* IN - Request handle */
)
```

**MPI\_Ssend\_init** 为同步模式发送操作创建一个持久的通信请求。当程序中反复调用参数相同的通信操作时此函数很有用。**MPI\_Start** 开启真正的消息发送过程。

```
int MPI_Start (
    MPI_Request *handle    /* IN - Request handle */
)
```

调用函数 **MPI\_Start** 来初始化该句柄关联的持久通信请求 (发送或者接收)。

```
int MPI_Startall (
    int size, /* IN - Elements in request_array */
    MPI_Request *requests /* IN-Array of pointers to
                           communication objects */
)
```

调用带有 **size** 个通信句柄的 **MPI\_Startall** 与调用 **size** 个按任意顺序传递 **requests** 中句柄的 **MPI\_Start** 是等效的。

```
int MPI_Test (
    MPI_Request *handle, /* IN - Persistent request handle */
    int *flag,           /* OUT - Completion flag */
    MPI_Status *status   /* OUT - Results of communication */
)
```

**MPI\_Test** 确定与某个通信请求相关联的操作是否已完成。如果操作已经完成，函数返回的 **flag** 值为真，否则为假。如果是一个接收操作结束了，可以访问 **status** 来获取消息的来源 (**status->MPI\_SOURCE**)，消息标志 (**status->MPI\_TAG**) 和错误码 (**status->MPI\_ERROR**)。

```
int MPI_Test_cancelled (
    MPI_Status *handle, /* IN - Communication handle */
    int *flag           /* OUT - Result flag */
)
```

对于传入的通信对象句柄，如果通信被成功地取消，函数将返回 **flag** 为真，否则为假。

```
int MPI_Testall (
    int cnt,                /* IN - Requests to test */
    MPI_Request *handle, /* IN - Array of request handles */
    int *flag,              /* OUT - Result flag */
    MPI_Status *status      /* OUT - Array of status info */
)
```

向 **MPI\_Testall** 传递 **cnt** 个通信请求句柄。当且仅当所有的通信都已完成，它返回的 **flag** 才为真。如果 **flag** 为真，**status** 数组的元素将含有通信结束的信息。

```
int MPI_Testany (
    int cnt,                /* IN - Number of requests to check */
    MPI_Request *handle, /* IN - Handles to request objects */
    int *index,            /* OUT - Index of a completed communication */
    int *flag,             /* OUT - Result flag */
    MPI_Status *status      /* OUT - Status information */
)
```

**MPI\_Testany** 检查 **cnt** 个通信中是否至少有一个已经完成。如果函数返回时 **flag** 为假，则说明没有任何通信操作已完成。如果 **flag** 为真，则至少有一个通信已经完成。**index** 是已完成的操作的句柄在 **handle** 数组中的位置。如果是一个接收操作，附加信息可通过 **status** 获取。

```
int MPI_Testsome (
    int in_cnt,                /* IN - Number of requests to test */
    MPI_Request *handlearray, /* IN - Array of request handles */
    int *out_cnt,              /* OUT - Number of completed requests */
    int *index_array,          /* OUT - Array of request indices */
    MPI_Status *status_array   /* OUT - Array of status records */
)
```

**MPI\_Testsome** 返回所有已完成的通信操作的信息。数组 **index\_array** 指出 **handlearray**

数组中哪些请求句柄是已完成的通信。接收操作的其他信息可通过 `status_array` 获取。

```
int MPI_Topo_test (
    MPI_Comm comm, /* IN - Communicator */
    int *topology /* OUT - Communicator's topology */
)
```

`MPI_Topo_test` 返回通信域的拓扑类型。`topolog` 可能的返回类型为: `MPI_GRAPH`, 对应图拓扑; `MPI_CART`, 对应笛卡尔拓扑; `MPI_UNDEFINED`, 对应无拓扑。

```
int MPI_Type_commit (
    MPI_Datatype *dtype /* IN - Derived datatype object */
)
```

`MPI_Type_commit` 提交一个派生数据类型, 以便在通信中使用它。参见 `MPI_Type_free`。

```
int MPI_Type_contiguous (
    int cnt, /* IN - Copies to make */
    MPI_Datatype old_dtype, /* IN - Old datatype */
    MPI_Datatype *new_dtype /* OUT - New datatype */
)
```

`MPI_Type_contiguous` 创建一个新数据类型, 它是由 `cnt` 个 `old_dtype` 联合组成的。

```
int MPI_Type_count (
    MPI_Datatype dtype, /* IN - Datatype */
    int *cnt /* OUT - Top-level entry count */
)
```

`MPI_Type_count` 返回数据类型 `dtype` 中“顶层”项的数目。

```
int MPI_Type_extent (
    MPI_Datatype dtype, /* IN - Datatype */
    MPI_Aint *extent /* OUT - Extent of dtype */
)
```

`MPI_Type_extent` 返回数据类型 `dtype` 所占的空间。也就是该数据类型的一个实例在消息中所占的字节数, 同时它也等于为了满足底层硬件数据对其需要, 对该数据类型元素通过适当的舍入后其所占的字节数。

```
int MPI_Type_free (
    MPI_Datatype *dtype /* IN - Derived datatype */
)
```

`MPI_Type_free` 用于释放 `dtype` 关联的数据类型对象, 并设置 `dtype` 为 `MPI_DATATYPE_NULL`。任何还未结束并且涉及到 `dtype` 的通信都会结束。见 `MPI_Type_commit`。

```
int MPI_Type_hindexed (
```

---

```

int cnt,                /* IN - Number of blocks */
int *block_len_array, /* IN - Elements in each block */
MPI_Aint *disp_array, /* IN - Block byte displacements */
MPI_Datatype old,      /* IN - Handle to old datatype */
MPI_Datatype *new      /* OUT - Handle to new datatype */
)

```

**MPI\_Type\_hindexed** 与 **MPI\_Type\_indexed** 完全一样，只是 **disp\_array** 中给出的块偏移量是按字节计算的。

---

```

int MPI_Type_hvector (
    int cnt,          /* IN - Number of blocks */
    int len,          /* IN - Number of elements per block */
    MPI_Aint stride, /* IN - Displacement in bytes between
                        start of each block */
    MPI_Datatype old, /* IN - Handle to old datatype */
    MPI_Datatype *new /* OUT - Handle to new datatype */
)

```

**MPI\_Type\_hvector** 通过重复一个现有数据类型来创建新的数据类型。新数据类型由 **cnt** 个块组成。每块包含 **block\_length** 个 **old** 副本。块间的距离（按字节计算）是 **stride**。

```

int MPI_Type_indexed (
    int cnt, /* IN - Number of blocks in 'new' */
    int *block_len, /* IN - Array indicating copies of
                     old datatype in each block */
    int *disp,      /* IN - Block displacements array */
    MPI_Datatype old, /* IN - Handle to old datatype */
    MPI_Datatype *new /* OUT - Handle to new datatype */
)

```

**MPI\_Type\_indexed** 把一个或多个块连接起来，形成新的派生数据类型 **new**，其中每一块包含一个或几个旧数据类型 **old**。整数 **cnt** 是要连接的块数。数组 **block\_len** 指出每块中 **old** 实例的数目。数组 **disp** 指出每块的偏移量，按 **old** 扩展的整数倍计算。

---

```

int MPI_Type_lb (
    MPI_Datatype dtype, /* IN - Handle to datatype */
    MPI_Aint *lb        /* OUT - Lower bound's displacement */
)

```

**MPI\_Type\_lb** 返回数据类型 **dtype** 的末端相对于起点的偏移量（对齐到字节）。

```

int MPI_Type_size (
    MPI_Datatype dtype, /* IN - Handle to datatype */
    int *size           /* OUT - Total size of type signature entries */
)

```

数据类型的“类型签名”(type signature)是该类型所包含的一系列基本类型。`MPI_Type_size` 返回 `dtype` 类型签名项的字节数,也就是包含一个此类型的数据元素的消息所占的字节数。

```
int MPI_Type_struct (
    int cnt,                /* IN - Number of blocks in 'new' */
    int *block_len,         /* IN - Elements in each block */
    MPI_Aint *disp,         /* IN - Displacement of each block */
    MPI_Datatype *dtype,    /* IN - Array of datatype handles */
    MPI_Datatype *new       /* OUT - Handle to new datatype */
)
```

`MPI_Type_struct` 构建一个包括 `cnt` 块的新数据类型。块 `i` 包含 `block_len[i]` 个与 `dtype[i]` 对应的数据类型。块 `i` 的偏移量由 `disp[i]` 指定。

```
int MPI_Type_ub (
    MPI_Datatype dtype, /* IN - Handle to datatype */
    MPI_Aint *ub        /* OUT - Upper bound's displacement */
)
```

`MPI_Type_ub` 返回数据类型 `dtype` 的上界相对于起点的偏移量(以字节为单位)。

```
int MPI_Type_vector (
    int cnt,                /* IN - Number of blocks */
    int block_length,       /* IN - Elements in each block */
    int stride,             /* IN - Elements between start of each block */
    MPI_Datatype old_dtype, /* IN - Handle to old datatype */
    MPI_Datatype *new_dtype /* OUT - New datatype's handle */
)
```

`MPI_Type_vector` 通过重复一个现有数据类型来创建新的数据类型。新数据类型由 `cnt` 块组成。每块包含 `block_length` 个 `old_dtype` 副本。块间的距离(按 `old_dtype` 扩展的整数倍计算)是 `stride`。

```
int MPI_Unpack (
    void *in_buffer, /* IN - Input buffer */
    int len,         /* IN - Length of input buffer */
    int *position,    /* IN/OUT - Position in 'in_buffer' */
    void *out_buffer, /* OUT - Output buffer */
    int out_cnt,      /* IN - Number of items to unpack */
    MPI_Datatype dtype, /* IN - Handle to
    MPI_Comm comm      /* IN - Communicator */
)
```

`MPI_Unpack` 将一个长为 `len` 字节的消息从 `in_buffer` 解包至 `out_buffer`。在函数执行的开始, `position` 是打包消息的第一个字节在 `in_buffer` 的索引。函数返回

时, `position` 是解包消息第一个字节的索引。`position` 参数允许多个消息打包进一个包单元, 然后发送出去, 再独立解包。

```

• int MPI_Wait (
    MPI_Request *handle, /* IN - Request handle */
    MPI_Status *status /* OUT - Result of communication */
)

```

`MPI_Wait` 用于完成任何非阻塞操作。如果针对一个发送操作, 函数将一直等到消息被运行系统缓冲或发送后才会返回。此时, 发送缓冲区可被重用。如果针对的是一个接收操作, 它将一直等到消息被复制至接收缓冲区后才返回。

```

int MPI_Waitall (
    int cnt, /* IN - Number of comms to wait on */
    MPI_Request *handle, /* IN - Request handles array */
    MPI_Status *status /* OUT - Status of completed comms */
)

```

`MPI_Waitall` 将一直阻塞到与存储在 `handle_array` 的句柄相关联的 `cnt` 个通信操作全部结束。当函数返回时, `status` 数组包含所有已完成通信的信息。

```

int MPI_Waitany (
    int cnt, /* IN - Number of comms to check */
    MPI_Request *handle, /* IN - Array of request handles */
    int *index, /* OUT - Index of completed comm */
    MPI_Status *status /* OUT - Status of completed comm */
)

```

`MPI_Waitany` 将一直阻塞, 直至指定的一组通信操作中有一个完成。当它返回时, `index` 是已完成通信在 `handle` 中的索引, `status` 指向已完成通信的状态记录。

```

int MPI_Waitsome (
    int in_cnt, /* IN - Number of comms to check */
    MPI_Request *handle, /* IN - Array of request handles */
    int *out_cnt, /* OUT - Number of completed ops */
    int *index_array, /* OUT - Array of completed ops */
    MPI_Status *status /* OUT - Array of status info */
)

```

`MPI_Waitsome` 将一直阻塞, 直至指定的一组通信操作中的一个或多个通信操作已经完成。当它返回时, `out_cnt` 是已完成的操作的数量。`index` 的前 `out_cnt` 个元素是已完成操作在 `handle` 中的位置, `status` 的前 `out_cnt` 个元素是这些通信的状态对象。

```

double MPI_Wtick (void)

```

`MPI_Wtick` 返回一个双精度浮点数, 表明函数 `MPI_Wtime` 中相邻两个滴答间所经历的秒数。例如, 如果时钟按微秒增加, 则 `MPI_Wtick` 将返回  $10^{-6}$ 。

```
double MPI_Wtime (void)
```

**MPI\_Wtime** 返回一个双精度浮点数, 表示从过去的某个时刻到现在流逝的秒数。“过去的某个时刻”在进程的生命中不会改变。所以一块代码的执行时间可以通过在其前后分别调用 **MPI\_Wtime** 得到的数据做差得到。

# 附录 B 工具函数

## B.1 MyMPI.h 头文件

```
/* MyMPI.h
*
* Header file for a library of matrix/vector
* input/output/redistribution functions.
*
* Programmed by Michael J. Quinn
*
* Last modification: 4 September 2002
*/
/***** MACROS *****/
#define DATA_MSG 0
#define PROMPT_MSG 1
#define RESPONSE_MSG 2

#define OPEN_FILE_ERROR -1
#define MALLOC_ERROR -2
#define TYPE_ERROR -3

#define MIN(a,b) ((a)<(b)?(a):(b))
#define BLOCK_LOW(id,p,n) ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id,p,n) (BLOCK_HIGH(id,p,n)-BLOCK_LOW(id,p,n)+1)
#define BLOCK_OWNER(j,p,n) (((p)*((j)+1)-1)/(n))
#define PTR_SIZE (sizeof(void*))
#define CEILING(i,j) (((i)+(j)-1)/(j))

/***** MISCELLANEOUS FUNCTIONS *****/
void terminate (int, char *);
/***** DATA DISTRIBUTION FUNCTIONS *****/
void replicate_block_vector (void *, int, void *, MPI_Datatype, MPI_Comm);
void create_mixed_xfer_arrays (int, int, int, int**, int**);
void create_uniform_xfer_arrays (int, int, int, int**, int**);

/***** INPUT FUNCTIONS *****/
void read_checkerboard_matrix (char *, void **, void **,
    MPI_Datatype, int *, int *, MPI_Comm);
```



```

void read_col_stripped_matrix (char *, void ***, void **,
    MPI_Datatype, int *, int *, MPI_Comm);
void read_row_stripped_matrix (char *, void ***, void **,
    MPI_Datatype, int *, int *, MPI_Comm);
void read_block_vector (char *, void **, MPI_Datatype, int *, MPI_Comm);
void read_replicated_vector (char *, void **, MPI_Datatype, int *, MPI_Comm);

/***** OUTPUT FUNCTIONS *****/
void print_checkerboard_matrix (void **, MPI_Datatype, int, int, MPI_Comm);
void print_col_stripped_matrix (void **, MPI_Datatype, int, int, MPI_Comm);
void print_row_stripped_matrix (void **, MPI_Datatype, int, int, MPI_Comm);
void print_block_vector (void *, MPI_Datatype, int, MPI_Comm);
void print_replicated_vector (void *, MPI_Datatype, int, MPI_Comm);

```

## B.2 MyMPI.c 源文件

```

/*
 * MyMPI.c -- A library of matrix/vector
 * input/output/redistribution functions
 *
 * Programmed by Michael J. Quinn
 *
 * Last modification: 4 September 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include "MyMPI.h"

/***** MISCELLANEOUS FUNCTIONS *****/
/*
 * Given MPI_Datatype 't', function 'get_size' returns the
 * size of a single datum of that data type.
 */

int get_size (MPI_Datatype t) {
    if (t == MPI_BYTE) return sizeof(char);
    if (t == MPI_DOUBLE) return sizeof(double);
    if (t == MPI_FLOAT) return sizeof(float);
    if (t == MPI_INT) return sizeof(int);
    printf ("Error: Unrecognized argument to 'get_size'\n");
    fflush (stdout);
}

```

```
MPI_Abort (MPI_COMM_WORLD, TYPE_ERROR);
}

/*
 * Function 'my_malloc' is called when a process wants
 * to allocate some space from the heap. If the memory
 * allocation fails, the process prints an error message
 * and then aborts execution of the program.
 */

void *my_malloc (
    int id, /* IN - Process rank */
    int bytes) /* IN - Bytes to allocate */
{
    void *buffer;
    if ((buffer = malloc ((size_t) bytes)) == NULL) {
        printf ("Error: Malloc failed for process %d\n", id);
        fflush (stdout);
        MPI_Abort (MPI_COMM_WORLD, MALLOC_ERROR);
    }
    return buffer;
}

/*
 * Function 'terminate' is called when the program should
 * not continue execution, due to an error condition that
 * all of the processes are aware of. Process 0 prints the
 * error message passed as an argument to the function.
 *
 * All processes must invoke this function together!
 */

void terminate (
    int id, /* IN - Process rank */
    char *error_message) /* IN - Message to print */
{
    if (!id) {
        printf ("Error: %s\n", error_message);
        fflush (stdout);
    }
    MPI_Finalize();
    exit (-1);
}
```

```

/***** DATA DISTRIBUTION FUNCTIONS *****/
/*
 * This function creates the count and displacement arrays
 * needed by scatter and gather functions, when the number
 * of elements send/received to/from other processes
 * varies.
 */

void create_mixed_xfer_arrays (
    int id, /* IN - Process rank */
    int p, /* IN - Number of processes */
    int n, /* IN - Total number of elements */
    int **count, /* OUT - Array of counts */
    int **disp) /* OUT - Array of displacements */
{
    int i;
    *count = my_malloc (id, p * sizeof(int));
    *disp = my_malloc (id, p * sizeof(int));
    (*count)[0] = BLOCK_SIZE(0,p,n);
    (*disp)[0] = 0;
    for (i = 1; i < p; i++) {
        (*disp)[i] = (*disp)[i-1] + (*count)[i-1];
        (*count)[i] = BLOCK_SIZE(i,p,n);
    }
}

/*
 * This function creates the count and displacement arrays
 * needed in an all-to-all exchange, when a process gets
 * the same number of elements from every other process.
 */

void create_uniform_xfer_arrays (
    int id, /* IN - Process rank */
    int p, /* IN - Number of processes */
    int n, /* IN - Number of elements */
    int **count, /* OUT - Array of counts */
    int **disp) /* OUT - Array of displacements */
{
    int i;
    *count = my_malloc (id, p * sizeof(int));
    *disp = my_malloc (id, p * sizeof(int));
    (*count)[0] = BLOCK_SIZE(id,p,n);
    (*disp)[0] = 0;
    for (i = 1; i < p; i++) {
        (*disp)[i] = (*disp)[i-1] + (*count)[i-1];
    }
}

```

```

        (*count)[i] = BLOCK_SIZE(id,p,n);
    }
}

/*
 * This function is used to transform a vector from a
 * block distribution to a replicated distribution within a
 * communicator.
 */

void replicate_block_vector (
    void *ablock, /* IN - Block-distributed vector */
    int n, /* IN - Elements in vector */
    void *arep, /* OUT - Replicated vector */
    MPI_Datatype dtype, /* IN - Element type */
    MPI_Comm comm) /* IN - Communicator */
{
    int *cnt; /* Elements contributed by each process */
    int *disp; /* Displacement in concatenated array */
    int id; /* Process id */
    int p; /* Processes in communicator */

    MPI_Comm_size (comm, &p);
    MPI_Comm_rank (comm, &id);
    create_mixed_xfer_arrays (id, p, n, &cnt, &disp);
    MPI_Allgatherv (ablock, cnt[id], dtype, arep, cnt, disp, dtype, comm);
    free (cnt);
    free (disp);
}

/***** INPUT FUNCTIONS *****/
/*
 * Function 'read_checkerboard_matrix' reads a matrix from
 * a file. The first two elements of the file are integers
 * whose values are the dimensions of the matrix ('m' rows
 * and 'n' columns). What follows are 'm'*'n' values
 * representing the matrix elements stored in row-major
 * order. This function allocates blocks of the matrix to
 * the MPI processes.
 *
 * The number of processes must be a square number.
 */
void read_checkerboard_matrix (
    char *s, /* IN - File name */
    void ***subs, /* OUT - 2D array */

```

```

void **storage, /* OUT - Array elements */
MPI_Datatype dtype, /* IN - Element type */
int *m, /* OUT - Array rows */
int *n, /* OUT - Array cols */
MPI_Comm grid_comm) /* IN - Communicator */
{
    void *buffer; /* File buffer */
    int coords[2]; /* Coords of proc receiving next row of matrix */
    int datum_size; /* Bytes per element */
    int dest_id; /* Rank of receiving proc */
    int grid_coord[2]; /* Process coords */
    int grid_id; /* Process rank */
    int grid_period[2]; /* Wraparound */
    int grid_size[2]; /* Dimensions of grid */
    int i, j, k;
    FILE *infileptr; /* Input file pointer */
    void *laddr; /* Used when proc 0 gets row */
    int local_cols; /* Matrix cols on this proc */
    int local_rows; /* Matrix rows on this proc */
    void **lptr; /* Pointer into 'subs' */
    int p; /* Number of processes */
    void *raddr; /* Address of first element to send */
    void *rptra; /* Pointer into 'storage' */
    MPI_Status status; /* Results of read */
    MPI_Comm_rank (grid_comm, &grid_id);
    MPI_Comm_size (grid_comm, &p);
    datum_size = get_size (dtype);

    /* Process 0 opens file, gets number of rows and
       number of cols, and broadcasts this information
       to the other processes. */
    if (grid_id == 0) {
        infileptr = fopen (s, "r");
        if (infileptr == NULL) *m = 0;
        else {
            fread (m, sizeof(int), 1, infileptr);
            fread (n, sizeof(int), 1, infileptr);
        }
    }

    MPI_Bcast (m, 1, MPI_INT, 0, grid_comm);
    if (!(*m)) MPI_Abort (MPI_COMM_WORLD, OPEN_FILE_ERROR);
    MPI_Bcast (n, 1, MPI_INT, 0, grid_comm);

    /* Each process determines the size of the submatrix
       it is responsible for. */

```

```

MPI_Cart_get (grid_comm, 2, grid_size, grid_period, grid_coord);
local_rows = BLOCK_SIZE(grid_coord[0],grid_size[0],*m);
local_cols = BLOCK_SIZE(grid_coord[1],grid_size[1],*n);

/* Dynamically allocate two-dimensional matrix 'subs' */
*storage = my_malloc (grid_id,
local_rows * local_cols * datum_size);
*subs = (void **) my_malloc (grid_id,local_rows*PTR_SIZE);
lptr = (void *) *subs;
rptr = (void *) *storage;
for (i = 0; i < local_rows; i++) {
    *(lptr++) = (void *) rptr;
    rptr += local_cols * datum_size;
}

/* Grid process 0 reads in the matrix one row at a time
and distributes each row among the MPI processes. */
if (grid_id == 0)
    buffer = my_malloc (grid_id, *n * datum_size);

/* For each row of processes in the process grid... */
for (i = 0; i < grid_size[0]; i++) {
    coords[0] = i;

    /* For each matrix row controlled by this proc row...*/
    for (j = 0; j < BLOCK_SIZE(i,grid_size[0],*m); j++) {

        /* Read in a row of the matrix */
        if (grid_id == 0) {
            fread (buffer, datum_size, *n, infileptr);
        }

        /* Distribute it among processes in the grid row */
        for (k = 0; k < grid_size[1]; k++) {
            coords[1] = k;
            /* Find address of first element to send */
            raddr = buffer +
                BLOCK_LOW(k,grid_size[1],*n) * datum_size;

            /* Determine the grid ID of the process getting the subrow */
            MPI_Cart_rank (grid_comm, coords, &dest_id);

            /* Process 0 is responsible for sending...*/
            if (grid_id == 0) {
                /* It is sending (copying) to itself */

```

```

        if (dest_id == 0) {
            laddr = (*subs)[j];
            memcpy (laddr, raddr, local_cols * datum_size);

            /* It is sending to another process */
        } else {
            MPI_Send (raddr,
                     BLOCK_SIZE(k,grid_size[1],*n), dtype,
                     dest_id, 0, grid_comm);
        }

        /* Process 'dest_id' is responsible for receiving... */
    } else if (grid_id == dest_id) {
        MPI_Recv ((*subs)[j], local_cols, dtype, 0,
                  0, grid_comm,&status);
    }
}

}

if (grid_id == 0) free (buffer);
}

/*
 * Function 'read_col_stripped_matrix' reads a matrix from a
 * file. The first two elements of the file are integers
 * whose values are the dimensions of the matrix ('m' rows
 * and 'n' columns). What follows are 'm'*'n' values
 * representing the matrix elements stored in row-major
 * order. This function allocates blocks of columns of the
 * matrix to the MPI processes.
 */
void read_col_stripped_matrix (
    char *s, /* IN - File name */
    void ***subs, /* OUT - 2-D array */
    void **storage, /* OUT - Array elements */
    MPI_Datatype dtype, /* IN - Element type */
    int *m, /* OUT - Rows */
    int *n, /* OUT - Cols */
    MPI_Comm comm) /* IN - Communicator */
{
    void *buffer; /* File buffer */
    int datum_size; /* Size of matrix element */
    int i, j;
    int id; /* Process rank */
    FILE *infileptr; /* Input file ptr */

```

```

int local_cols; /* Cols on this process */
void **lptr; /* Pointer into 'subs' */
void *rptr; /* Pointer into 'storage' */
int p; /* Number of processes */
int *send_count; /* Each proc's count */
int *send_disp; /* Each proc's displacement */

MPI_Comm_size (comm, &p);
MPI_Comm_rank (comm, &id);
datum_size = get_size (dtype);

/* Process p-1 opens file, gets number of rows and
   cols, and broadcasts this info to other procs. */
if (id == (p-1)) {
    infilepath = fopen (s, "r");
    if (infilepath == NULL) *m = 0;
    else {
        fread (m, sizeof(int), 1, infilepath);
        fread (n, sizeof(int), 1, infilepath);
    }
}

MPI_Bcast (m, 1, MPI_INT, p-1, comm);

if (!(*m)) MPI_Abort (comm, OPEN_FILE_ERROR);
MPI_Bcast (n, 1, MPI_INT, p-1, comm);
local_cols = BLOCK_SIZE(id,p,*n);

/* Dynamically allocate two-dimensional matrix 'subs' */
*storage = my_malloc (id, *m * local_cols * datum_size);
*subs = (void **) my_malloc (id, *m * PTR_SIZE);
lptr = (void *) *subs;
rptr = (void *) *storage;
for (i = 0; i < *m; i++) {
    *(lptr++) = (void *) rptr;
    rptr += local_cols * datum_size;
}

/* Process p-1 reads in the matrix one row at a time and
   distributes each row among the MPI processes. */
if (id == (p-1)) buffer = my_malloc (id, *n * datum_size);
create_mixed_xfer_arrays (id,p,*n,&send_count,&send_disp);
for (i = 0; i < *m; i++) {
    if (id == (p-1))
        fread (buffer, datum_size, *n, infilepath);
    MPI_Scatterv (buffer, send_count, send_disp, dtype,

```



```

        (*storage)+i*local_cols*datum_size, local_cols,
        dtype, p-1, comm);
    }
    free (send_count);
    free (send_disp);
    if (id == (p-1)) free (buffer);
}

/*
 * Process p-1 opens a file and inputs a two-dimensional
 * matrix, reading and distributing blocks of rows to the
 * other processes.
 */
void read_row_striped_matrix (
    char *s, /* IN - File name */
    void ***subs, /* OUT - 2D submatrix indices */
    void **storage, /* OUT - Submatrix stored here */
    MPI_Datatype dtype, /* IN - Matrix element type */
    int *m, /* OUT - Matrix rows */
    int *n, /* OUT - Matrix cols */
    MPI_Comm comm) /* IN - Communicator */
{
    int datum_size; /* Size of matrix element */
    int i;
    int id; /* Process rank */
    FILE *infileptr; /* Input file pointer */
    int local_rows; /* Rows on this proc */
    void **lptr; /* Pointer into 'subs' */
    int p; /* Number of processes */
    void *rptr; /* Pointer into 'storage' */
    MPI_Status status; /* Result of receive */
    int x; /* Result of read */

    MPI_Comm_size (comm, &p);
    MPI_Comm_rank (comm, &id);
    datum_size = get_size (dtype);

    /* Process p-1 opens file, reads size of matrix,
       and broadcasts matrix dimensions to other procs */
    if (id == (p-1)) {
        infileptr = fopen (s, "r");
        if (infileptr == NULL) *m = 0;
        else {
            fread (m, sizeof(int), 1, infileptr);
            fread (n, sizeof(int), 1, infileptr);

```

```

    }
}

MPI_Bcast (m, 1, MPI_INT, p-1, comm);
if (!(*m)) MPI_Abort (MPI_COMM_WORLD, OPEN_FILE_ERROR);
MPI_Bcast (n, 1, MPI_INT, p-1, comm);
local_rows = BLOCK_SIZE(id,p,*m);

/* Dynamically allocate matrix. Allow double subscripting
   through 'a'. */
*storage = (void *) my_malloc (id,
local_rows * *n * datum_size);
*subs = (void **) my_malloc (id, local_rows * PTR_SIZE);
lptr = (void *) &(*subs[0]);
rptr = (void *) *storage;
for (i = 0; i < local_rows; i++) {
*(lptr++)= (void *) rptr;
rptr += *n * datum_size;
}

/* Process p-1 reads blocks of rows from file and
   sends each block to the correct destination process.
   The last block it keeps. */
if (id == (p-1)) {
    for (i = 0; i < p-1; i++) {
        x = fread (*storage, datum_size,
            BLOCK_SIZE(i,p,*m) * *n, infileptr);
        MPI_Send (*storage, BLOCK_SIZE(i,p,*m) * *n, dtype,
            i, DATA_MSG, comm);
    }
    x = fread (*storage, datum_size, local_rows * *n, infileptr);
    fclose (infileptr);
} else
    MPI_Recv (*storage, local_rows * *n, dtype, p-1, DATA_MSG, comm, &status);
}

/*
* Open a file containing a vector, read its contents,
* and distribute the elements by block among the
* processes in a communicator.
*/
void read_block_vector (
    char *s, /* IN - File name */
    void **v, /* OUT - Subvector */
    MPI_Datatype dtype, /* IN - Element type */
    int *n, /* OUT - Vector length */

```

```

MPI_Comm comm) /* IN - Communicator */
{
    int datum_size; /* Bytes per element */
    int i;
    FILE *infileptr; /* Input file pointer */
    int local_els; /* Elements on this proc */
    MPI_Status status; /* Result of receive */
    int id; /* Process rank */
    int p; /* Number of processes */
    int x; /* Result of read */
    datum_size = get_size (dtype);
    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &id);
    /* Process p-1 opens file, determines number of vector
       elements, and broadcasts this value to the other processes. */
    if (id == (p-1)) {
        infileptr = fopen (s, "r");
        if (infileptr == NULL) *n = 0;
        else fread (n, sizeof(int), 1, infileptr);
    }
    MPI_Bcast (n, 1, MPI_INT, p-1, comm);
    if (! *n) {
        if (!id) {
            printf ("Input file '%s' cannot be opened\n", s);
            fflush (stdout);
        }
    }

    /* Block mapping of vector elements to processes */
    local_els = BLOCK_SIZE(id,p,*n);

    /* Dynamically allocate vector. */
    *v = my_malloc (id, local_els * datum_size);
    if (id == (p-1)) {
        for (i = 0; i < p-1; i++) {
            x = fread (*v, datum_size, BLOCK_SIZE(i,p,*n), infileptr);
            MPI_Send (*v, BLOCK_SIZE(i,p,*n), dtype, i, DATA_MSG, comm);
        }
        x = fread (*v, datum_size, BLOCK_SIZE(id,p,*n), infileptr);
        fclose (infileptr);
    } else {
        MPI_Recv (*v, BLOCK_SIZE(id,p,*n), dtype, p-1, DATA_MSG,
                  comm, &status);
    }
}

```

/\* Open a file containing a vector, read its contents,  
and replicate the vector among all processes in a communicator. \*/

```
void read_replicated_vector (
    char *s, /* IN - File name */
    void **v, /* OUT - Vector */
    MPI_Datatype dtype, /* IN - Vector type */
    int *n, /* OUT - Vector length */
    MPI_Comm comm) /* IN - Communicator */
{
    int datum_size; /* Bytes per vector element */
    int i;
    int id; /* Process rank */
    FILE *infileptr; /* Input file pointer */
    int p; /* Number of processes */

    MPI_Comm_rank (comm, &id);
    MPI_Comm_size (comm, &p);
    datum_size = get_size (dtype);
    if (id == (p-1)) {
        infileptr = fopen (s, "r");
        if (infileptr == NULL) *n = 0;
        else fread (n, sizeof(int), 1, infileptr);
    }
    MPI_Bcast (n, 1, MPI_INT, p-1, MPI_COMM_WORLD);
    if (! *n) terminate (id, "Cannot open vector file");
    *v = my_malloc (id, *n * datum_size);
    if (id == (p-1)) {
        fread (*v, datum_size, *n, infileptr);
        fclose (infileptr);
    }
    MPI_Bcast (*v, *n, dtype, p-1, MPI_COMM_WORLD);
}
```

/\* \*\*\*\*\* OUTPUT FUNCTIONS \*\*\*\*\* \*/

/\*

\* Print elements of a doubly subscripted array.

\*/

```
void print_submatrix (
    void **a, /* OUT - Doubly subscripted array */
    MPI_Datatype dtype, /* OUT - Type of array elements */
    int rows, /* OUT - Matrix rows */
    int cols) /* OUT - Matrix cols */
{
```

```

    int i, j;
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            if (dtype == MPI_DOUBLE)
                printf ("%6.3f ", ((double **)a)[i][j]);
            else {
                if (dtype == MPI_FLOAT)
                    printf ("%6.3f ", ((float **)a)[i][j]);
                else if (dtype == MPI_INT)
                    printf ("%6d ", ((int **)a)[i][j]);
            }
        }
        putchar ('\n');
    }
}

/*
 * Print elements of a singly subscripted array.
 */
void print_subvector (
    void *a, /* IN - Array pointer */
    MPI_Datatype dtype, /* IN - Array type */
    int n) /* IN - Array size */
{
    int i;
    for (i = 0; i < n; i++) {
        if (dtype == MPI_DOUBLE)
            printf ("%6.3f ", ((double *)a)[i]);
        else {
            if (dtype == MPI_FLOAT)
                printf ("%6.3f ", ((float *)a)[i]);
            else if (dtype == MPI_INT)
                printf ("%6d ", ((int *)a)[i]);
        }
    }
}

/*
 * Print a matrix distributed checkerboard fashion among
 * the processes in a communicator.
 */
void print_checkerboard_matrix (
    void **a, /* IN -2D matrix */
    MPI_Datatype dtype, /* IN -Matrix element type */
    int m, /* IN -Matrix rows */

```

```

int n, /* IN -Matrix columns */
MPI_Comm grid_comm) /* IN - Communicator */
{
    void *buffer; /* Room to hold 1 matrix row */
    int coords[2]; /* Grid coords of process
    sending elements */
    int datum_size; /* Bytes per matrix element */
    int els; /* Elements received */
    int grid_coords[2]; /* Coords of this process */
    int grid_id; /* Process rank in grid */
    int grid_period[2]; /* Wraparound */
    int grid_size[2]; /* Dims of process grid */
    int i, j, k;
    void *laddr; /* Where to put subrow */
    int local_cols; /* Matrix cols on this proc */
    int p; /* Number of processes */
    int src; /* ID of proc with subrow */

    MPI_Status status; /* Result of receive */
    MPI_Comm_rank (grid_comm, &grid_id);
    MPI_Comm_size (grid_comm, &p);
    datum_size = get_size (dtype);
    MPI_Cart_get (grid_comm, 2, grid_size, grid_period, grid_coords);
    local_cols = BLOCK_SIZE(grid_coords[1],grid_size[1],n);

    if (!grid_id)
        buffer = my_malloc (grid_id, n*datum_size);

    /* For each row of the process grid */
    for (i = 0; i < grid_size[0]; i++) {
        coords[0] = i;

        /* For each matrix row controlled by the process row */
        for (j = 0; j < BLOCK_SIZE(i,grid_size[0],m); j++) {

            /* Collect the matrix row on grid process 0 and print it. */
            if (!grid_id) {
                for (k = 0; k < grid_size[1]; k++) {
                    coords[1] = k;
                    MPI_Cart_rank (grid_comm, coords, &src);
                    els = BLOCK_SIZE(k,grid_size[1],n);
                    laddr = buffer + BLOCK_LOW(k,grid_size[1],n) * datum_size;
                    if (src == 0) {
                        memcpy (laddr, a[j], els * datum_size);
                    } else {

```

```

        MPI_Recv(laddr, els, dtype, src, 0, grid_comm, &status);
    }
}
print_subvector (buffer, dtype, n);
putchar ('\n');
} else if (grid_coords[0] == i) {
    MPI_Send (a[j], local_cols, dtype, 0, 0, grid_comm);
}
}
}
if (!grid_id) {
    free (buffer);
    putchar ('\n');
}
}

/*
 * Print a matrix that has a columnwise-block-striped data
 * decomposition among the elements of a communicator.
 */
void print_col_striped_matrix (
    void **a, /* IN - 2D array */
    MPI_Datatype dtype, /* IN - Type of matrix elements */
    int m, /* IN - Matrix rows */
    int n, /* IN - Matrix cols */
    MPI_Comm comm) /* IN - Communicator */
{
    MPI_Status status; /* Result of receive */
    int datum_size; /* Bytes per matrix element */
    void *buffer; /* Enough room to hold 1 row */
    int i, j;
    int id; /* Process rank */
    int p; /* Number of processes */
    int* rec_count; /* Elements received per proc */
    int* rec_disp; /* Offset of each proc's block */

    MPI_Comm_rank (comm, &id);
    MPI_Comm_size (comm, &p);
    datum_size = get_size (dtype);
    create_mixed_xfer_arrays (id, p, n, &rec_count, &rec_disp);

    if (!id)
        buffer = my_malloc (id, n*datum_size);

    for (i = 0; i < m; i++) {

```

```

    MPI_Gatherv (a[i], BLOCK_SIZE(id,p,n), dtype, buffer,
        rec_count, rec_disp, dtype, 0, MPI_COMM_WORLD);
    if (!id) {
        print_subvector (buffer, dtype, n);
        putchar ('\n');
    }
}

free (rec_count);
free (rec_disp);
if (!id) {
    free (buffer);
    putchar ('\n');
}
}

/*
 * Print a matrix that is distributed in row-striped
 * fashion among the processes in a communicator.
 */
void print_row_striped_matrix (
    void **a, /* IN - 2D array */
    MPI_Datatype dtype, /* IN - Matrix element type */
    int m, /* IN - Matrix rows */
    int n, /* IN - Matrix cols */
    MPI_Comm comm) /* IN - Communicator */
{
    MPI_Status status; /* Result of receive */
    void *bstorage; /* Elements received from
another process */
    void **b; /* 2D array indexing into 'bstorage' */
    int datum_size; /* Bytes per element */
    int i;
    int id; /* Process rank */
    int local_rows; /* This proc's rows */
    int max_block_size; /* Most matrix rows held by any process */
    int prompt; /* Dummy variable */
    int p; /* Number of processes */

    MPI_Comm_rank (comm, &id);
    MPI_Comm_size (comm, &p);
    local_rows = BLOCK_SIZE(id,p,m);

    if (!id) {
        print_submatrix (a, dtype, local_rows, n);
        if (p > 1) {

```



```

    datum_size = get_size (dtype);
    max_block_size = BLOCK_SIZE(p-1,p,m);
    bstorage = my_malloc (id, max_block_size * n * datum_size);
    b = (void **) my_malloc (id, max_block_size * datum_size);
    b[0] = bstorage;
    for (i = 1; i < max_block_size; i++) {
        b[i] = b[i-1] + n * datum_size;
    }
    for (i = 1; i < p; i++) {
        MPI_Send (&prompt, 1, MPI_INT, i, PROMPT_MSG,
                  MPI_COMM_WORLD);
        MPI_Recv (bstorage, BLOCK_SIZE(i,p,m)*n, dtype,
                  i, RESPONSE_MSG, MPI_COMM_WORLD, &status);
        print_submatrix (b, dtype, BLOCK_SIZE(i,p,m), n);
    }
    free (b);
    free (bstorage);
}

putchar ('\n');
} else {
    MPI_Recv (&prompt, 1, MPI_INT, 0, PROMPT_MSG,
              MPI_COMM_WORLD, &status);
    MPI_Send (*a, local_rows * n, dtype, 0, RESPONSE_MSG,
              MPI_COMM_WORLD);
}

}

/*
 * Print a vector that is block distributed among the
 * processes in a communicator.
 */
void print_block_vector (
    void *v, /* IN - Address of vector */
    MPI_Datatype dtype, /* IN - Vector element type */
    int n, /* IN - Elements in vector */
    MPI_Comm comm) /* IN - Communicator */
{
    int datum_size; /* Bytes per vector element */
    int i;
    int prompt; /* Dummy variable */
    MPI_Status status; /* Result of receive */
    void *tmp; /* Other process's subvector */
    int id; /* Process rank */
    int p; /* Number of processes */
    MPI_Comm_size (comm, &p);

```

```

MPI_Comm_rank (comm, &id);
datum_size = get_size (dtype);

if (!id) {
    print_subvector (v, dtype, BLOCK_SIZE(id,p,n));
    if (p > 1) {
        tmp = my_malloc (id,BLOCK_SIZE(p-1,p,n)*datum_size);
        for (i = 1; i < p; i++) {
            MPI_Send (&prompt, 1, MPI_INT, i, PROMPT_MSG, comm);
            MPI_Recv (tmp, BLOCK_SIZE(i,p,n), dtype, i,
                RESPONSE_MSG, comm, &status);
            print_subvector (tmp, dtype, BLOCK_SIZE(i,p,n));
        }
        free (tmp);
    }
    printf ("\n\n");
} else {
    MPI_Recv (&prompt, 1, MPI_INT, 0, PROMPT_MSG, comm, &status);
    MPI_Send (v, BLOCK_SIZE(id,p,n), dtype, 0,
        RESPONSE_MSG, comm);
}
}

/*
 * Print a vector that is replicated among the processes
 * in a communicator.
 */
void print_replicated_vector (
    void *v, /* IN - Address of vector */
    MPI_Datatype dtype, /* IN - Vector element type */
    int n, /* IN - Elements in vector */
    MPI_Comm comm) /* IN - Communicator */
{
    int id; /* Process rank */
    MPI_Comm_rank (comm, &id);
    if (!id) {
        print_subvector (v, dtype, n);
        printf ("\n\n");
    }
}

```

# 附录 C 调试 MPI 程序

## C.1 概 述

编程是一项容易出错的活动。固然认真细致的设计是开发正确程序过程中最重要的一步，但是实际上，每个程序员编写的程序都需要调试。许多程序员通过源码调试器进行调试工作，还有的通过使用 `printf` 进行错误排查。任何一种方法所消耗的时间都依赖于程序员的技巧和程序的复杂程度。

调试并行程序比调试串行程序更困难。首先，我们会遇到更多可能导致错误的原因。多个进程同时进行计算，相互之间通过消息传递来交互信息。其次，并行调试器不如串行调试器完善。并行程序员通常没有好的工具。

本附录列出 MPI 程序中常见的错误，还有一些调试 MPI 程序的经验方法。

## C.2 MPI 程序常见错误

### C.2.1 死锁错误

如果一个进程“为了等待一个永远不可能满足的条件而阻塞”【3】，那么这个进程就陷入了死锁（deadlock）。如果其中的一个或几个进程发生死锁，MPI 程序就不会结束。MPI 程序中的死锁通常可以分为以下几类：

**死锁错误 1：**只有一个进程调用组通信函数。例如，只有根进程调用了 `MPI_Reduce` 或 `MPI_Bcast`。

**防止措施：**不要在条件语句的分支中调用组通信函数。如果必须要在条件语句块调用组通信函数，请确保条件表达式在所有进程的计算结果都是相同的。这样的话，要么所有进程都执行包含组通信函数的代码块，要么都不执行。

**死锁错误 2：**两个或多个进程交换数据，在调用发送函数之前都调用了阻塞接收函数，如 `MPI_Recv`。

**防止措施：**有多种方法防止这类错误。第一，可以组织程序使其在调用 `MPI_Recv` 前调用 `MPI_Send` 或其他消息发送函数。第二，用 `MPI_Sendrecv` 替换 `MPI_Send` 和 `MPI_Recv`，它可以保证不会发生死锁。第三，用非阻塞函数 `MPI_Irecv` 替换阻塞式函数 `MPI_Recv`，并将相应的 `MPI_Wait` 紧跟在 `MPI_Send` 之后。

**死锁错误 3：**一个进程从一个从不向它发送消息的进程接收数据，导致死锁。

**防止措施：**如果通信域中没有与此进程号对应的进程，MPI 运行系统会捕获此错误。然而，如果进程号在可接收范围内，那么它就不起什么作用了。避免此类错误最好的方法

是在可能的时候使用组通信函数。如果必须使用点对点通信，请务必保证采取简单的通信模式。

**死锁错误 4:** 进程试图从自身接收数据。

**防止措施:** 对源码简单的检查即可以清除这类错误。也可以在每个接收函数之前进程运行时检查。

## C.2.2 导致不准确结果的错误

**错误原因 1:** 发送和接收类型不匹配。例如，发送进程元素类型为 `MPI_INT` 的消息放入缓冲区，而接收进程按照 `MPI_FLOAT` 读取消息。

**防止措施:** 组织程序确保每个消息发送函数都有一个与之匹配的消息接收函数。保证可以方便地确定与每个发送相对应的接收。仔细检查以确保每对函数的消息长度和元素类型都相同。

**错误原因 2:** 错误填写的参数。将 `MPI_Reduce` 的第一个和第二个参数写反就是一个例子。

**防止措施:** 大多数 MPI 函数都有很多参数。最好的办法就是在用到某个 MPI 函数时参考附录 A 的函数声明以保证参数位置正确。

## C.2.3 组通信的优点

点对点通信（如发送和接收）比组通信（如广播和归约）更容易产生错误。在组通信中，通常所有的进程都位于程序中同一个位置。所有进程都在源码的同一行启动此函数。所以所有的参数也都相同。如果一个进程能正确调用该函数，则所有调用都正确。

相反，考虑点对点通信。在大多数本地通信中，发送和接收分别调用不同的 MPI 函数，这为错误填写不匹配的参数以及其他错误的产生创造了条件。有可能的情况比如：弄错源和目的，弄错一个或几个参数的类型，弄错传递的数据元素的个数，弄错消息标志等等。

所以，我们应该合理地使用组通信函数。

# C.3 实用调试策略

- 如果并程序只在一个进程上运行，首先必须保证用一个进程运行的版本能够正确工作。这样可以测试程序的许多功能，比如 I/O。更重要的是，你可以利用串行调试器设置断点，测试数值等等。
- 如果程序在一个进程上能正确工作，那么再在允许程序所有功能都可被测试的最小进程数下工作。通常，2 到 3 个进程就足够了。
- 在允许所有功能都可被测试的最小问题规模下运行程序。例如，当编写一个解线性方程组的程序时， $4 \times 4$  的方程组和  $1024 \times 1024$  的方程组所能测试的功能是一样的。在较小的问题规模下，可以使用 `printf` 查看整个数据结构。此外，由于程序

的输出很少, 所以输出的结果也比较容易理解。

- 在每个 `printf` 之后调用 `fflush(stdout)`。否则, 在程序出错或发生死锁之前我们可能将得不到所有的输出。
- 对于点对点消息, 打印发送数据和接收数据以保证二者匹配。
- 从任意一个进程接收到的消息必须遵守时间顺序, 而从不同进程接收的消息可以不遵守时间顺序。不要认为如果来自 X 进程的消息比来自 Y 进程的消息更早出现, X 的消息就会先于 Y 的消息被打印出来。先按照进程号处理所有消息, 然后利用 Unix 的 `sort` 以根据进程号对程序的输出做排序。将进程输出按照进程号组织在一起进行分析, 这才是你惟一可做的。
- 首先调试程序的初始化部分以保证能正确地创建所有的数据结构。
- 检查程序以确保每个处理器所有访问本地数据时用到的本地索引都是正确的。
- 调试程序的时候, 不要为了优化性能而进行消息合并或使用复杂的数据结构。首先保证逻辑正确性, 然后再考虑合并消息或其他性能优化步骤。

## 附录 D 复数回顾

本附录回顾了如何进行复数计算。本文大部分参考了 Weaver 的论述【112】。

复数是一个有序实数对，用  $(x, y)$  表示，其中称  $x$  为复数的实部，称  $y$  为复数的虚部。两个复数  $(x_1, y_1)$  和  $(x_2, y_2)$  相等，当且仅当  $x_1 = x_2$  且  $y_1 = y_2$ 。

令  $z_1 = (x_1, y_1)$ ， $z_2 = (x_2, y_2)$  为两个复数，则其和为：

$$z_1 + z_2 = (x_1 + x_2, y_1 + y_2)$$

其积为：

$$z_1 z_2 = (x_1 x_2 - y_1 y_2, x_1 y_2 + y_1 x_2)$$

复数乘法满足交换率，结合率和分配率。

任意实数  $x$  可用实数  $(x, 0)$  表示。

三个特殊复数是：零元素，单元素和虚单位元。

零元素，用  $0$  表示，为复数  $(0, 0)$ 。

任意复数  $z$  和  $0$  的和都为  $z$ ：

$$z + 0 = (x, y) + (0, 0) = (x + 0, y + 0) = (x, y) = z$$

任意复数  $z$  和  $0$  的积都为  $0$ ：

$$z0 = (x, y)(0, 0) = (x \times 0 - y \times 0, x \times 0 + y \times 0) = (0, 0) = 0$$

单位元素，用  $1$  表示，复数为  $(1, 0)$ 。

任意复数  $z$  和  $1$  的积都为  $z$ ：

$$z \times 1 = (x, y)(1, 0) = (x \times 1 - y \times 0, 1 \times y + 0 \times x) = (x, y) = z$$

虚单位元，用  $i$  表示，复数为  $(0, 1)$ 。虚部单位元素是  $-1$  的平方根：

$$i^2 = (0, 1)(0, 1) = (0 \times 0 - 1 \times 1, 0 \times 1 + 1 \times 0) = (-1, 0) = -1$$

### 定理 D.1

任意复数  $z = (x, y)$  可用  $x + iy$  表示。

证明：

实数  $x = (x, 0)$ 。实数  $y$  和虚部单位元素  $i$  的积为：

$$iy = (0, 1)(y, 0) = (0 \times y - 1 \times 0, 0 \times 0 + y \times 1) = (0, y)$$

所以，

$$x + iy = (x, 0) + (0, y) = (x, y)$$

如图 D.1 所示，我们将复数  $z$  表示为  $x + iy$ ，横轴对应  $z$  的实部，竖轴对应  $z$  的虚部。

复数  $z$  可以看作是长为  $r$  角度为  $\theta$  的向量， $\theta$  按照从正实轴逆时针旋转的度数来度量。

记为：

$$x = r \cos \theta$$

$$y = r \sin \theta$$

根据这个公式可得： $z = x + iy = r(\cos \theta + i \sin \theta)$

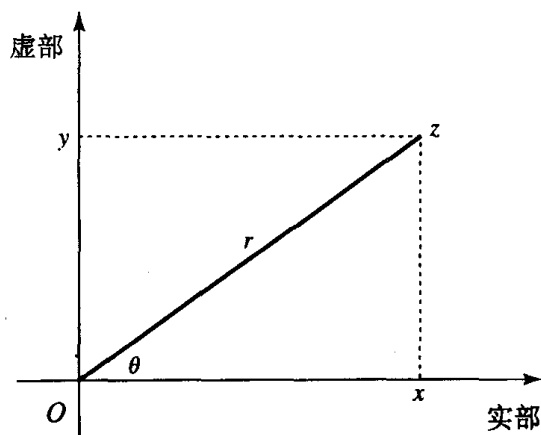


图 D.1 任意复数  $z$  可用有序实数对  $(x, y)$  来表示,  $x$  为实部,  $y$  为虚部。还可以用长为  $r$  角度为  $\theta$  的向量表示,  $\theta$  的大小按照从正实轴逆时针旋转的度数来度量

在学习离散傅里叶变换的时候需要用指数的形式表示  $z$ , 也就是下面得到的。根据泰勒公式展开:

$$\sin \theta = \theta - \frac{\theta^3}{3} + \frac{\theta^5}{5} - \frac{\theta^7}{7} + L$$

$$\cos \theta = 1 - \frac{\theta^2}{2} + \frac{\theta^4}{4} - \frac{\theta^6}{6} + L$$

$$e^{i\theta} = 1 + i\theta - \frac{\theta^2}{2} - \frac{i\theta^3}{3} + \frac{\theta^4}{4} + \frac{i\theta^5}{5} + L$$

$$= \left( 1 - \frac{\theta^2}{2} + \frac{\theta^4}{4} - \frac{\theta^6}{6} + L \right) + i \left( \theta - \frac{\theta^3}{3} + \frac{\theta^5}{5} - \frac{\theta^7}{7} + L \right)$$

结合这些公式有:

$$e^{i\theta} = \cos \theta + i \sin \theta$$

和

$$e^{-i\theta} = \cos \theta - i \sin \theta$$

因为  $z = x + iy = r(\cos \theta + i \sin \theta)$ , 所以:

$$z = r e^{i\theta}$$

是复数的另一种表示法。

指数形式复数的一个特性就是它可以简化乘除法。令  $z_1 = r_1 e^{i\theta_1}$ ,  $z_2 = r_2 e^{i\theta_2}$  为两个复数,

那么:

$$z_1 z_2 = r_1 e^{i\theta_1} r_2 e^{i\theta_2} = r_1 r_2 e^{i(\theta_1 + \theta_2)}$$

$$z_1 / z_2 = r_1 e^{i\theta_1} / r_2 e^{i\theta_2} = (r_1 / r_2) e^{i(\theta_1 - \theta_2)}$$

单位 1 的  $n$  次复根是复数  $\omega$ , 即  $\omega^n = 1$ 。

单位 1 的  $n$  次复根共有  $n$  个, 即  $e^{2\pi i k / n}$ , 其中  $k = 1, 2, \dots, n$ 。

用  $\omega_n$  来表示复数  $e^{2\pi i / n}$ , 它是单位 1 的主  $n$  次方根。

图 D.2 解释了单位 1 的 8 次方根和它的幂, 即它的其他几个 8 次方根。

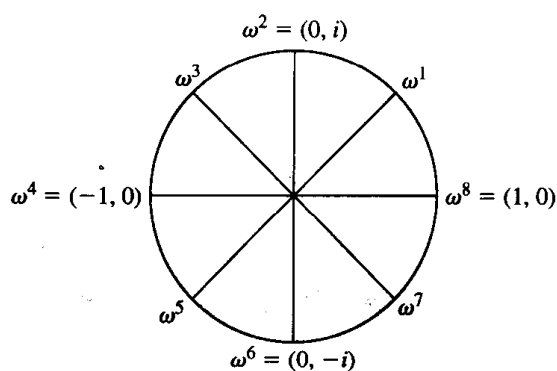


图 D.2 单位 1 的主 8 次方根和它的幂

**引理 D.1**

(消去原理) 对任意整数  $n \geq 0$ ,  $k \geq 0$  和  $d > 0$ , 有  $\omega_{dn}^{dk} = \omega_n^k$ 。

证明:

$$\omega_{dn}^{dk} = (e^{2\pi i / dn})^{dk} = (e^{2\pi i / n})^k = \omega_n^k$$

**推论 D.1**

对任意偶数  $n > 0$ , 有  $\omega_n^{n/2} = \omega_2 = -1$ 。

证明:

$$\omega_n^{n/2} = \omega_{(n/2)2}^{(n/2)1} = \omega_2^1 = \omega_2 = -1。$$

**引理 D.2**

(等分引理) 如果  $n$  为正偶数, 单位 1 的  $n$  个  $n$  次方根的平方与它的  $(n/2)$  个  $n/2$  次方根是相同的。

证明:

根据消去原理可知, 如果  $k$  非负, 则  $(\omega_n^k)^2 = \omega_{n/2}^k$ 。如果将所有  $n$  次方根平方, 就分别得到  $(n/2)$  次方根两次, 因为:

$$(\omega_n^{k+n/2})^2 = \omega_n^{2k+n} = \omega_n^{2k} \omega_n^n = \omega_n^{2k} = (\omega_n^k)^2$$



# 附录 E OpenMP 函数

本附录描述了 OpenMP 标准中所有的 c/c++函数。每个函数的参数个数不多于一个。所有的参数都是输入参数，其值均由调用函数提供。所有结果均通过函数返回值的方式返回。

```
int omp_get_dynamic(void)
```

如果支持动态线程，`omp_get_dynamic` 将返回 1，否则将返回 0。

---

```
int omp_get_max_threads(void)
```

`omp_get_max_threads` 返回运行系统所允许的程序可以创建的线程数目最大值。

---

```
int omp_get_nested(void)
```

如果允许并行嵌套，`omp_get_nested` 将返回 1，否则将返回 0。目前所有的 OpenMP 实现在默认情况下都不允许并行嵌套。

---

```
int omp_get_num_procs(void)
```

`omp_get_num_procs` 返回并行环境中可用的处理器数。

---

```
int omp_get_num_threads(void)
```

`omp_get_num_threads` 返回当前活动的线程数。如果此函数在串行部分被调用，它将返回 1。

---

```
int omp_get_thread_num(void)
```

`omp_get_thread_num` 返回线程的标志数。如果有  $t$  个活动线程，线程的标志数将为  $0 \sim t-1$ 。

---

```
int omp_in_parallel(void)
```

如果在一个被并行化了的代码块范围内被调用，`omp_in_parallel` 将返回 1，否则将返回 0。

---

```
void omp_set_dynamic( int k /* 1 = ON, 0 = FALSE */)
```

`omp_set_dynamic` 可以使能或禁止动态线程。如果动态线程被置为有效，运行系统可能会调整活动的线程数目以匹配可用物理处理器数。如果想明确知道进入并行块后创建了多少线程，我们可以将动态线程设为无效。

---

```
void omp_set_nested(int k /* 1 = enable; 0 = disable */) 
```

`omp_set_nested` 可以使能或禁止并行嵌套。目前的 OpenMP 实现只支持一层并行化，默认情况下并行嵌套是被禁止了的，即使激活它也不起作用。所以，此函数在目前的 OpenMP 实现中没有意义。

```
void omp_set_num_threads(int t /* 期望的线程数 */)
```

`omp_set_num_threads` 设置后续的并行块所期望的线程数。线程数可能超过可用的处理器数，此时多个线程会被映射到同一个处理器上。此函数必须在串行部分被调用。

# 参 考 文 献

1. Akl, S. G. *Parallel Sorting Algorithms*. Orlando, FL: Academic Press, 1985.
2. Amdahl, G. "Validity of the single processor approach to achieving large scale computing capabilities." In *AFIPS Conference Proceedings*, Vol. 30, pages 483–485, Washington, D.C.: Thompson Books, April 1967.
3. Andrews, Gregory R. *Concurrent Programming: Principles and Practice*. Redwood City, CA: Benjamin/Cummings, 1991.
4. Anton, H. *Elementary Linear Algebra*. 3d ed. New York: John Wiley & Sons, 1981.
5. Baase, Sara, and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. 3d ed. Reading, MA: Addison-Wesley, 2000.
6. Babb, Robert G. II. Introduction. In Robert G. Babb II, (ed.), *Programming Parallel Processors*, pages 1–6. Reading, MA: Addison-Wesley, 1988.
7. Bacon, David F., Susan L. Graham, and Oliver J. Sharp. "Compiler transformations for high-performance computing." *ACM Computing Surveys* 26(4):345–420, December 1994.
8. Batcher, K. E. "Sorting networks and their applications." In *Proceedings of the AFIPS Spring Joint Computer Conference*, Vol. 32, pages 307–314. Reston, VA: AFIPS Press, 1968.
9. Bertsekas, D. P., and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
10. Bhattacharya, S., and A. Bagchi. "Searching game trees in parallel using SSS\*." In *Proceedings AAAI-90*, pages 42–47. The AAAI Press, 1990.
11. Bressoud, David M. *Factorization and Primality Testing*. New York: Springer-Verlag, 1989.
12. Browne, James C., Syed I. Hyder, Jack Dongarra, Keith Moore, and Peter Newton. "Visual programming and debugging for parallel computing." *IEEE Parallel & Distributed Technology* 3(1):75–83, Spring 1995.
13. Camp, W. J., S. J. Plimpton, B. A. Hendrickson, and R. W. Leland. "Massively parallel methods for engineering and science problems." *Communications of the ACM* 37(4):30–41, April 1994.
14. Cannon, L. E. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, Bozeman, MT, 1969.
15. Carriero, Nicholas, and David Gelerntner. *How to Write Parallel Programs: A First Course*. Cambridge, MA: The MIT Press, 1990.
16. Chandra, Rohit, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and

- Ramesh Menon. *Parallel Programming in OpenMP*. San Francisco: Morgan Kaufmann, 2001.
17. Coddington, Paul D. "Random number generators for parallel computers." Technical report, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, April 1997.
  18. Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 2d ed. Cambridge, MA: The MIT Press, 2001.
  19. de Groot, A. D. *Thought and Choice in Chess*. The Hague: Mouton, 1965.
  20. de Kergommeaux, Jacques Chassin, and Philippe Codognet. "Parallel logical programming systems." *ACM Computing Surveys* 26(3): 295–336, September 1994.
  21. Dijkstra, E. W., W. H. Seijen, and A. J. M. V. Gasteren. "Derivation of a termination detection algorithm for a distributed computation." *Information Processing Letters* 16(5):217–219, 1983.
  22. Dongarra, J. J., I. S. Duff, D. C. Sorenson, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Philadelphia: SIAM Press, 1991.
  23. Fagan, M. J. *Finite Element Analysis: Theory and Practice*. Singapore: Longman Scientific & Technical, 1992.
  24. Feitelson, Dror G., Anat Barat, Gabriel Benhanokh, David Er-El, Yoav Etsion, Avi Kavas, Tomer Klainer, Uri Lublin, and Marc A. Volovic. "The ParPar system: A software MPP." In Rajkumar Buyya (ed.), *High Performance Cluster Computing: Architectures and Systems*. Vol. 1, pages 754–770. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.
  25. Felten, E. W., and S. W. Otto. "A highly parallel chess program." In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 1001–1009. ICOT, 1988.
  26. Ferguson, C., and R. E. Korf. "Distributed tree search and its application to alpha-beta pruning." In *Proceedings AAAI-88*, pages 128–132, 1988.
  27. Floyd, R. W. Algorithm 97: Shortest path. *Communications of the ACM* 5(6):345, June 1962.
  28. Flynn, Michael J. "Very high-speed computing systems." *Proceedings of the IEEE* 54(12): 1901–1909, December 1966.
  29. Flynn, Michael J. "Some computer organizations and their effectiveness." *IEEE Transactions on Computers* C-21(9):948–960, September 1972.
  30. Flynn, Michael J., and Kevin W. Rudd. "Parallel architectures." *ACM Computing Surveys* 28(1):67–70, March 1996.
  31. Foster, Ian. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Reading, MA: Addison-Wesley, 1995.
  32. Foster, Ian, and K. M. Chandy. "Fortran M: A language for modular parallel programming." *Journal of Parallel and Distributed Computing* 25(1), 1995.

33. Fox, G. C., M. A. Johnson, G. A. Syzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. Solving Problems on Concurrent Processors. Vol. 1. Englewood Cliffs, NJ: Prentice-Hall, 1988.
34. Fox, Geoffrey C., Roy D. Williams, and Paul C. Messina. *Parallel Computing Works*. San Francisco: Morgan Kaufmann, 1994.
35. Francis, R. S., and I. D. Mathieson. "A benchmark parallel sort for shared memory multiprocessors." *IEEE Transactions on Computers* C-37(12):1619–1626, December 1988.
36. Gallivan, K. A., R. J. Plemmons, and A. H. Sameh. "Parallel algorithms for dense linear algebra computations." *SIAM Review* 32:54–135, March 1990.
37. Galloway, Robert L., W. Andrew Bass, and Christopher E. Hockey. "Task-oriented asymmetric multiprocessing for interactive image-guided surgery." *Parallel Computing* 24(9–10):1323–1343, September 1998.
38. Garey, Michael R., and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman, 1979.
39. Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: The MIT Press, 1994.
40. Gill, S. "Parallel programming." *Computer Journal* 1(1):2–10, April 1958.
41. Golub, Gene, and James M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Boston, MA: Academic Press, 1993.
42. Goodman, S. E., and S. T. Hedetniemi. *Introduction to the Design and Analysis of Algorithms*. New York: McGraw-Hill, 1977.
43. Grama, Ananth Y., Anshul Gupta, and Vipin Kumar. "Isoefficiency: Measuring the scalability of parallel algorithms and architectures." *IEEE Parallel & Distributed Technology* 1(3):12–21, August 1993.
44. Grama, Ananth, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. 2d ed. Harlow, England: Addison-Wesley, 2003.
45. Gropp, William, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: The MIT Press, 1994.
46. Gustafson, John L. "Reevaluating Amdahl's law." *Communications of the ACM* 31(5):532–533, May 1988.
47. Gustafson, John L., Gary R. Montry, and Robert E. Benner. "Development of parallel methods for a 1024-processor hypercube." *SIAM Journal on Scientific and Statistical Computing* 9(4):609–638, March 1988.
48. Gustafson, F. G. "Recursion leads to automatic variable blocking for dense linear-algebra algorithms." *IBM Journal of Research and Development* 41(6), 1999.
49. Hatcher, Philip J., and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. Cambridge, MA: The MIT Press, 1991.

50. Hillis, W. Daniel, and Guy L. Steele, Jr. "Data parallel algorithms." *Communications of the ACM* 29(12):1170–1183, December 1986.
51. Hoare, C. A. R. "Quicksort." *Computer Journal* 5(1):10–15, 1962.
52. Hockney, R. W., and C. R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Bristol: Adam Hilger Ltd, 1981.
53. Houstis, E. N., J. R. Rice, S. Weerawarana, A. C. Catlin, P. Papachiou, K.-Y. Wang, and M. Gaitatzes. "PELLPACK: A problem-solving environment for PDE-based applications on multicomputer platforms." *ACM Transactions on Mathematical Software* 24(1):30–73, March 1998.
54. Hsu, Feng Hsiung. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton, NJ: Princeton University Press, 2002.
55. Huntbach, M. M., and F. W. Burton. "Alpha-beta search on virtual tree machines." *Information Sciences* 44:3–17, 1988.
56. Jacobi, C. G. J. "Über eine neue auflösungsart der bei der methode der kleinsten quadrate vorkommenden linearen gleichungen." *Astr. Nachr.* 22(523):297–306, 1845.
57. Jain, A. K., M. N. Murty, and P. J. Flynn. "Data clustering: A review." *ACM Computing Surveys* 31(3):264–323, 1999.
58. Kalos, Malvin H., and Paula A. Whitlock. *Monte Carlo Methods, Volume 1: Basics*. New York: John Wiley & Sons, 1986.
59. Karp, Alan H., and Horace P. Flatt. "Measuring parallel processor performance." *Communications of the ACM* 33(5):539–543, May 1990.
60. Kauffmann, William J. III, and Larry L. Smarr. *Supercomputing and the Transformation of Science*. New York: Scientific American Library, 1993.
61. Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall PTR, 1988.
62. Koelbel, Charles H., David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Cambridge, MA: The MIT Press, 1994.
63. Lai, T. H., and S. Sahni. "Anomalies in parallel branch-and-bound algorithms." *Communications of the ACM* 27(6):594–602, June 1984.
64. Landau, David P., and Kurt Binder. *A Guide to Monte Carlo Simulations in Statistical Physics*. Cambridge: Cambridge University Press, 2000.
65. Landau, Rubin H., and Manuel J. Paez. *Computational Physics: Problem Solving with Computers*. New York: John Wiley & Sons, 1997.
66. Lawrie, Duncan H. "Access and alignment of data in an array processor." *IEEE Transactions on Computers* C-24(12):1145–1155, December 1975.
67. Lea, W. A. "Speech recognition: Past, present, and future." In *Trends in Speech Recognition*. Englewood Cliffs, NJ: Prentice-Hall, 1980.
68. L'Ecuyer, Pierre, and Richard Simard. "Beware of the linear congruential generators

- with multipliers of the form  $a = \pm 2^q \pm 2^r$ ." *ACM Transactions on Mathematical Software* 25(3):367–374, September 1999.
69. Lehmer, Derrick Henry. "Mathematical methods in large scale computing units." In *Proceedings of the Second Symposium on Large-Scale Digital Calculating Machinery*, pages 141–146. Cambridge, MA: Harvard University Press, 1951.
  70. Leiserson, Charles E. *Area-Efficient VLSI Computation*. Cambridge, MA: The MIT Press, 1983.
  71. Lester, Bruce P. *The Art of Parallel Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1993.
  72. Leva, Joseph L. "A fast normal random number generator." *ACM Transactions on Mathematical Software* 18(4):449–453, December 1992.
  73. Levin, E. "Grand challenges to computational science." *Communications of the ACM* 32(12):1456–1457, December 1989.
  74. Li, X., P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. "On the versatility of parallel sorting by regular sampling." *Parallel Computing* 19:1079–1193, 1993.
  75. Luhn, H. P. "The automatic creation of literature abstracts." *IBM Journal of Research and Development* 2(2):159–165, 317, April 1958.
  76. Luo, Xuedong. "A practical sieve algorithm for finding prime numbers." *Communications of the ACM* 32(3):344–346, March 1989.
  77. Makino, Jun. "Lagged-Fibonacci random number generators on parallel computers." *Parallel Computing* 20(9):1357–1367, 1994.
  78. Manno, István. *Introduction to the Monte-Carlo Method*. Akadémiai Kiadó, Budapest, Hungary, 1999.
  79. Marsaglia, George. "Random numbers fall mainly in the planes." *Proceedings of the National Academy of Sciences of the United States of America* 62:25–28, 1968.
  80. Marsaglia, George, and Wai Wan Tsang. "The Monty Python method for generating random variables." *ACM Transactions on Mathematical Software* 24(3):341–350, September 1998.
  81. Marsland, T. A., and M. Campbell. "Parallel search of strongly ordered game trees." *Computing Surveys* 14(4):533–551, December 1982.
  82. Mascagni, M., S. A. Cuccaro, D. V. Pryor, and M. L. Robinson. "A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator." *Computational Physics* 19:211–219, 1995.
  83. Mascagni, M., and A. Srinivasan. "Algorithm 806. SPRNG: A scalable library for pseudorandom number generation." *ACM Transactions on Mathematical Software* 26(4):618–619, December 2000.
  84. Mascagni, Michael. "Some methods of parallel pseudorandom number generation." In *Algorithms for Parallel Processing*, New York: Springer-Verlag, 1999.
  85. McGraw, James R., and Timothy S. Axelrod. "Exploiting multiprocessors: Issues and

- options." In Robert G. Babb II, (ed.), *Programming Parallel Processors*, pages 7–25. Reading, MA: Addison-Wesley, 1988.
86. Mehrotra, Piyush, Joel Saltz, and Robert Voigt, editors. *Unstructured Scientific Computation on Scalable Multiprocessors*. Cambridge, MA: The MIT Press, 1992.
87. Moore, Gordon. "Cramming more components onto integrated circuits." *Electronics Magazine* 38(8):114–117, April 1965.
88. Nagendra, Bhavana, and Lars Rzymianowicz. "High speed networks." In Rajkumar Buyya, (ed.), *High Performance Cluster Computing: Architectures and Systems*. Vol. 1, pages 204–245. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.
89. Pacheco, Peter S. *Parallel Programming with MPI*. San Francisco: Morgan Kaufmann, 1997.
90. Patterson, David A., and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. 2d ed. San Francisco: Morgan Kaufmann, 1996.
91. Percus, Ora E., and Malvin H. Kalos. "Random number generators for MIMD parallel processors." *Journal of Parallel and Distributed Computing* 6:477–497, 1989.
92. Plybon, Benjamin F. *An Introduction to Applied Numerical Analysis*. Boston, MA: PWS-Kent Publishing Company, 1992.
93. Pountain, Dick, and David May. *A Tutorial Introduction to Occam Programming*. Oxford: BSP Professional Books, 1987.
94. Quinn, Michael J. "Parallel sorting algorithms for tightly coupled multiprocessors." *Parallel Computing* 6:349–357, 1988.
95. Quinn, Michael J. *Parallel Computing: Theory and Practice*. 2d ed. New York: McGraw-Hill, 1994.
96. Quinn, Michael J., and Narsingh Deo. "An upper bound for the speedup of parallel best-bound branch-and-bound algorithms." *BIT* 26(1):35–43, March 1986.
97. Reingold, E. M., J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Englewood Cliffs, NJ: Prentice-Hall, 1977.
98. Rumelhart, David E., Bernard Widrow, and Michael A. Lehr. "The basic ideas in neural networks." *Communications of the ACM* 37(3):87–92, March 1994.
99. Sabot, Gary W. (ed.). *High Performance Computing: Problem Solving with Parallel and Vector Architectures*. Reading, MA: Addison-Wesley, 1995.
100. Schaeffer, J. "Distributed game-tree searching." *Journal of Parallel and Distributed Computing* 6:90–114, 1989.
101. Shi, H., and J. Schaeffer. "Parallel sorting by regular sampling." *Journal of Parallel and Distributed Computing* 14:361–372, 1992.
102. Skillicorn, David B., and Domenico Talia. "Models and languages for parallel computation." *ACM Computing Surveys* 30(2):123–169, June 1998.
103. Slagle, J. R., and J. K. Dixon. "Experiments with some programs that search game trees." *Journal of the ACM* 16(2):189–207, April 1969.



104. Smith, G. D. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford: Oxford University Press, 1985.
105. Sterling, Thomas, (ed.). *Beowulf Cluster Computing with Linux*. Cambridge, MA: The MIT Press, 2002.
106. Tentner, A. M., R. N. Blomquist, T. R. Canfield, P. L. Garner, E. M. Gelbard, K. C. Gross, M. Minkoff, and R. A. Valentin. "Advances in parallel computing for reactor analysis and safety." *Communications of the ACM* 37(4):54–64, 1994.
107. Valiant, Leslie G. "A bridging model for parallel computation." *Communications of the ACM* 33(8), August 1990.
108. Wagar, Bruce. "Hyperquicksort: A fast sorting algorithm for hypercubes." In *Hypercube Multiprocessors 1987*, pages 292–299. Philadelphia, PA: SIAM, 1987.
109. Wah, B. W., G. Li, and C.-F. Yu. "Multiprocessing of combinatorial search problems." *Computer* 18(6):93–108, June 1985.
110. Wallace, C. S. "Fast pseudorandom generators for normal and exponential variables." *ACM Transactions on Mathematical Software* 22(1):119–127, March 1990.
111. Warshall, S. "A theorem on boolean matrices." *Journal of the ACM* 9(1):11–12, January 1962.
112. Weaver, H. J. *Applications of Discrete and Continuous Fourier Analysis*. New York: John Wiley & Sons, 1983.
113. Wheat, M., and D. J. Evans. "An efficient parallel sorting algorithm for shared memory multiprocessors." *Parallel Computing* 18:91–102, 1992.
114. Widrow, Bernard, David E. Rumelhart, and Michael A. Lehr. "Neural networks: Applications in industry, business, and science." *Communications of the ACM* 37(3):93–105, March 1994.
115. Wilkinson, Barry, and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, NJ: Prentice-Hall, 1999.
116. Wilson, Gregory V. *Practical Parallel Programming*. Cambridge, MA: The MIT Press, 1995.
117. Wolfe, Michael. *High Performance Compilers for Parallel Computing*. Redwood City, CA: Addison-Wesley, 1996.
118. Wu, Pei-Chi. "Multiplicative, congruential random-number generators with multiplier  $\pm 2^{k_1} \pm 2^{k_2}$  and modulus  $2^p - 1$ ." *ACM Transactions on Mathematical Software* 23(2):255–265, June 1997.