Parallel Computer Architecture
A Hardware / Software Approach (Second Edition)

并行计算机体系结构

硬件/软件结合的设计与分析

（原书第2版）

国外经典教材

Classical Texts From Top Universities

MORGAN KAUFMANN

David E. Culler
（美） Jaswinder Pal Singh 著
Anoop Gupta

李晓明 钱德沛 程旭 崔光佐 译

机械工业出版社
China Machine Press

国外经典教材 Classical Texts From Top Universities

并行计算机体系结构

硬件/软件结合的设计与分析（原书第2版）

Parallel Computer Architecture

A Hardware/Software Approach

(Second Edition)

作者简介

David E. Culler 加州大学伯克利分校计算机科学系教授。Culler 博士的研究领域包括计算机体系结构、通信、编程语言、操作系统和性能分析。他曾是伯克利工作站网络（NOW）项目的领导者，该项目点燃了当前在全球范围发展的高性能机群系统商业化的革命之火。Culler博士因他在快速通信的主动消息、LogP 并行性能模型、Split-C并行语言、TAM线程抽象机和数据流体系结构方面的研究而闻名于世。他曾被国家科学基金会授予过总统学术研究基金奖和总统青年研究人员奖。他于1989年获得麻省理工学院博士学位。目前任加州大学伯克利分校电气工程与计算机科学系的副主任，是千年（Millennium）项目的负责人，从事校园网范围的机群系统的研究。

Jaswinder Pal Singh 普林斯顿大学计算机科学系助理教授。Singh博士的研究领域是并行应用和多处理器系统，包括体系结构、软件和性能评价。他领导开发了在并行系统领域广为使用的SPLASH和SPLASH-2并行程序组件。他在斯坦福大学攻读硕士和博士学位期间，参加了 DASH 和 FLASH多处理器项目，是该项目应用开发方面的负责人。DASH项目开发的技术现在广泛地应用于商业产品中。在普林斯顿，他领导着PRISM项目组，该项目组以应用为驱动，致力于研究支持不同通信体系结构的编程模型，并将并行计算应用于各种不同的领域。他曾获得过总统科学家与工程师初期成就奖（PECASE）和 Sloan研究基金奖。

Anoop Gupta 斯坦福大学电气工程与计算机科学系副教授，微软高级研究员。Gupta 博士的研究领域包括计算机体系结构、操作系统、编程语言、性能调试工具和并行应用。他和John Hennessy 共同领导了斯坦福DASH机的设计和构建，DASH机是世界上最早的可扩展的分布式共享内存的多处理器之一。Gupta博士也参加了之后的FLASH项目。DASH项目开发的技术现在广泛地应用于商业产品中。Gupta 教授至今已在主要学术会议和刊物上发表了近100篇论文。Gupta教授曾获得过国家科学基金会授予的总统青年研究人员奖。他也是斯坦福大学的 Robert Noyce 学院院长。他于1986年获得卡内基梅隆大学的博士学位。

ISBN 7-111-07888-8

9 787111 078883

封面设计：陈子平

HZ BOOKS

华章图书

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037

购书热线：(010)68995259，8006100280（北京地区）

ISBN 7-111-07888-8/TP · 1404

定价：78.00 元

HZ BOOKS
华章教育
北京华章图文信息有限公司

国外经典教材

Classical Texts From Top Universities

（原书第2版）

并行计算机体系结构

硬件/软件结合的设计与分析

Parallel Computer Architecture

A Hardware/Software Approach

(Second Edition)

David E. Culler
（美） Jaswinder Pal Singh 著
Anoop Gupta
李晓明 钱德沛 程旭 崔光佐 译

机械工业出版社
China Machine Press

Parallel Computer Architecture: A Hardware/Software Approach, Second Edition
David E. Culler, Jaswinder Pal Singh and Anoop Gupta
ISBN: 1-55860-343-3
Copyright © 1996 by Morgan Kaufmann. All rights reserved.
Copyright © 2002 by Harcourt Asia Pte Ltd. All rights reserved.
Authorized Simplified Chinese translation edition published by the Proprietor.
ISBN 981-4134-95-3
Copyright © 2002 by Elsevier Science (Singapore) Pte Ltd. All rights reserved.
Elsevier Science (Singapore) Pte Ltd.
3 Killiney Road
#08-01 Winsland Hose I
Singapore 239519
Tel: (65) 6349-0200
Fax:(65) 6733-1817
First Published 2003
2003年初版
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan.

本书任何部分之文字及图片，如未获得本公司之书面同意，不得用任何方法抄袭、节录或翻印。本版仅限在中国境内（不包括香港特别行政区及台湾地区）出版及标价销售。未经许可之出口，是为违反著作权法，将受法律之制裁。

本书版权登记字：图字：01-2000-1859

图书在版编目（CIP）数据

并行计算机体系结构：硬件/软件结合的设计与分析（原书第2版）/（美）卡勒（Culler, D. E.）等著；李晓明等译. -北京：机械工业出版社，2002.10
（国外经典教材）
书名原文：Parallel Computer Architecture: A Hardware/Software Approach, Second Edition
ISBN 7-111-07888-8

Ⅰ.并… Ⅱ.①卡…②李… Ⅲ.并行计算机－计算机体系结构－教材 Ⅳ.TP338.6

中国版本图书馆CIP数据核字（2002）第008129号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：温丹丹
北京第二外国语学院印刷厂印刷·新华书店北京发行所发行
2003年1月第1版第1次印刷
787mm×1092mm 1/16·50.25印张
印数：0 001-5 000册
定价：78.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

出版者的话

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

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

机械工业出版社华章图文信息有限公司较早意识“出版要为教育服务”。自 1998 年始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与 Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann 等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出 Tanenbaum，Stroustrup，Kernighan，Jim Gray 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：针对本科生的核心课程，剔抉外版菁华而成“国外经典教材”系列；对影印版的教材，则单独开辟出“经典原版书库”；定位在高级教程和专业参考的“计算机科学丛书”还将保持原来的风格，继续出版新的品种。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

“国外经典教材”是响应教育部提出的使用外版教材的号召，为国内高校的计算机本科教学度身订造的。在广泛地征求并听取丛书的“专家指导委员会”的意见后，我们最终选定了这20多种篇幅内容适度、讲解鞭辟入里的教材，其中的大部分已经被M.I.T.、Stanford、U.C. Berkley、C.M.U.等世界名牌大学采用。丛书不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com
联系电话：(010) 68995265
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

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

序

十分高兴能为这本关于并行计算的新书作序。本书的内容令人兴奋，具有鲜明的时代特征。作者富有见解的思路，以及对各种计算机体系结构所做的系统化和定量化分析，将这本书与先前所有关于并行计算机体系结构的书区别开来。这种在开头4章里奠定的基本思路有三个主要的创新：它建立在近年来并行计算机体系结构的融合之上；它以应用作为驱动来评价和分析体系结构；它植根于一种坚实的性能评价方法论之中。

如同在第1章里所描述的一样，近年来共享存储和消息传递模式的融合，展现了一种在一个公共的框架下刻画和分析计算机体系结构的新机遇。基于这种融合，作者描述了4种根本的设计要点（对通信的抽象、程序设计模型、通信和复制的关系、性能的追求）。这种做法创建了一个用以讨论各种体系结构和具体实现的框架。在这个框架中，我们可以对不同体系结构的做法进行严格的比较和考察。

不理解应用和体系结构的相互作用，我们就不能理解多处理器的设计权衡和性能。为此，针对性能的提高，第2、3章介绍了应用程序并行化的过程以及一组并行程序。除了铺垫一个定量性评价体系结构及其实现的基础，这几章勾画了并行程序设计的过程和它带来的挑战，它们对于理解多处理器的性能是关键的。通过展示如何用一种并行的工作负载来评价体系结构，第4章进一步阐明了上述观念。作者还描述了评价并行计算机的复杂性，包括由机器规模和工作负载的扩展所带来的若干问题。这3章一起形成了其余各章论述的基础。

中小规模共享存储系统是目前的主导并行体系结构。任何对并行计算感兴趣的人，理解这类机器的原理和设计权衡是至关重要的。第5章描述了共享存储多重处理技术的主要概念：高速缓存的一致性、存储同一性和同步。然后作者描述了基于侦听的共享存储多处理器的详细设计，包括第6章两个详尽的案例分析。

设计多处理器系统，使之能够扩展到更多的节点，依然是多处理器体系结构领域最具有挑战性和争议的一个方面。第7章专门从消息传递到共享存储的设计领域讨论这个问题。第8章将这种讨论延伸，考察了目录方案的使用，它使得缓存一致性能够扩展到大量的处理节点。第8章还讨论了基于目录的一致性的基本问题，两个详尽的案例分析形成了本章的核心。在本书之前，针对基于目录的高速缓存一致性技术的商品计算机，还没有如此详细的和定量的分析。

用于多处理器系统的某些最重要的硬件和软件技术基本上是独立于体系结构细节的。因此，作者用3章的篇幅探讨了这些关键技术。第9章描述了存储系统中软件的影响、硬件的需求以及性能权衡，包括同一性问题和高速缓存的扩展使用。第10章考察了互连技术，这是任何多处理器系统都要有的一个关键组成部分。最后，第11章考察了用于时延包容的技术。从多个方面来讲，这是并行计算机的一个“普适”的设计问题。

作为结束，本书给出了关于未来硬件和软件挑战的一个富有见识的讨论。首先，作者讨

论了可能的发展情景，包括硬件和软件两个方面。然后作者在题为“遇到困难”的两段文字中转向潜在的障碍。最后，他们考察了可能的技术突破！我发现最后一章既有启发性，也有思想性。作者们不同的背景和互补的优势使得这一章既有洞察力，又有刺激性。

总而言之，这是一个关于多处理器系统设计领域的令人兴奋并且反映时代特征的探索性成果。体系结构设计方法融合的趋势同作者提出的框架的结合，使得我们可能建立一种公共的基础来考察多样化的现代并行体系结构。几年前，由于各种体系结构的设计方法相差太远，要写这样一本书是不可能的。类似地，如果没有注意定量的性能评测以及应用和体系结构的相互作用，这本书也就不会这么突出。而我们的作者恰好利用了这种技术的融合，并将注意力集中在应用驱动和针对性能的分析上，对并行计算机体系结构领域进行了一次独特的、富有见识的探索。这种做法，和作者们独特的优势和经验结合在一起，产生了这部论著，同其他任何并行体系结构的书相比，它体现了更深刻的认识。我向作者们表示祝贺，并向所有对并行处理技术的理论和实践，以及这些技术的现在和未来感兴趣的读者推荐此书。

John L. Hennessy，斯坦福大学腓特烈·埃蒙·特曼工程学院院长

译 者 序

翻译这本书的念头，源于1998年11月在美国弗罗里达州奥兰多市参加超级计算（Supercomputing）学术会议。当时，摩根考夫曼（Morgan Kaufmann）出版社的工作人员在现场展销图书。本书是其中之一。粗粗浏览，立刻被书中新颖的内容所吸引，当即买下一本。回过头看看，除了对上述印象有进一步的加深外，还发现它和以前类似题材的教材相比，语言也很有特色：在陈述种种技术要点时，还展开了大量的剖析性分析。我们感到，本书能使读者对技术的脉络有更深刻的把握。于是，在机械工业出版社华章公司的支持下，我们开始了本书的翻译工作。

这项工作基本上分为三个阶段。第一阶段，由李晓明负责翻译序、前言、第2、3、5、6章和附录，钱德沛翻译第4、9、10章，程旭翻译第1、7、8章，崔光佐翻译第11、12章。在统一了一些名词的译法后，在第二阶段由李晓明修改序、前言、第1、2、3、5、6章和附录，钱德沛修改第4、7、8、9、10章，崔光佐依然修改第11、12章。在最后审定阶段由李晓明负责第4、7、8、9、10、11、12章，钱德沛负责序、前言、第1、2、3、5、6章和附录。清样出来后主要由李晓明和钱德沛负责最后的校对。

尽管我们几个人近年来一直在从事和计算机体系结构有关的教学和科研工作，但坦率地说，做好这件事对我们依然是很难的。一方面，本书的内容十分广博，有一些是我们以前了解不多的；另一方面，本书的写作风格倾向于口语化，描述性强，因此在翻译过程中既要准确地表达其技术含义，又要尽量兼顾其表述风格，对我们是个很大的挑战。在一些难以兼顾的场合，我们采用了“译者注”的方式来处理，即在行文中基本遵照原文句子的结构，但会对其技术含义作进一步的解释。另外，在翻译过程中我们作了几次词汇译法的讨论和统一，在尽量采用国内有关计算机术语常用译法的同时，也对极少数术语作了我们认为更合适的处理。最典型的情况有两个，一是有关高速缓存访问的“miss”，以前常用的是“失效”，我们这里统一译成“扑空”；另一个是在讨论共享存储系统时常用的“coherence”和“consistency”，目前国内的译法都是“一致性”，但在本书中它们多次同时出现，翻译起来有明显困难，因此我们分别译成“一致性”和“同一性”。还有一个特别的词汇scalability（scale，scalable）。这是目前几乎到处都在用的词，基本含义是当系统规模扩大时其典型应用性能改善程度的度量，具体指在保证性能的条件下，典型应用的规模随计算系统的规模扩大而扩大的程度，隐含有两种因素（系统规模，应用性能）按比例变化的含义。在本书的翻译中，大多数场合采用了常用的“可扩展性”，但在需要强调其按比例变化含义的地方，有时也译成了“可扩放性”。最后，在翻译过程中我们得到了原著作者提供的勘误表，其内容也反映在此译本中。

历经近三年的时间，《*Parallel Computer Architecture: A Hardware/Software Approach, Second Edition*》中译本终于出版了，这其中还有不少人参加了相关的工作。北京大学和西安交大一些同学参加了本书翻译的一些前期工作和后期索引整理的工作，北京大学的黄蕊在文字校对中花了许多时间，为我们纠正了不少欠妥之处，在此我们一并表示感谢。

尽管如此，这个译本一定还会存在不少缺点、疏漏和蹩脚之处。我们欢迎读者指出问题，提出建议。为此，我们计划建立并维护一个相关网站 http：//lxm.cs.pku.edu.cn/pca/来反映读者的意见和建议，在上面提供相关章节和段落的更新或修改信息，以弥补现时由于我们的水平和时间所限可能造成的疏误。

译　者

2002 年 1 月

前　言

并行计算已经成为20世纪90年代计算技术的一个至关重要的组成部分。在接下来的20年里，它所产生的影响将有可能和微处理器在前20年里产生的影响相当。事实上，这两种技术有着深刻的联系，高度集成的微处理器和存储器芯片的发展使得多处理器系统日益具有更大的吸引力。多处理器系统已经代表着几乎所有计算市场层面的高性能端部分，从最快的超级计算机、最大的数据中心到部门级服务器，到单个的台式机。若干PC、工作站，甚至多处理器系统紧密集成起来，所形成的机群正作为可扩展因特网服务器出现。过去，计算机厂家在它们的产品系列中利用不同的技术和处理器体系结构来满足不断增加的系统性能要求。今天，产品中处处用到的都是相同的微处理器。在一个相当大的范围内，性能提高的基本手段是增加处理器的个数。这种性能扩展的经济效益使其极具吸引力。很快地，若干处理器将会集在一个芯片上，多处理器系统将会比今天更加流行。

尽管并行计算的学术历史在时间上不算短，在内容上也很丰富，但从根本上改变这一学科现状的是和商用技术的紧密结合。对新颖体系结构和特殊技术的强调已经让位于定量的分析，让位于在相同处理节点上实现不同的程序设计模型，让位于仔细的工程性权衡。我们写此书的目的就是让设计人员掌握这一类新型的多处理器系统的设计，从中小规模的并行台式计算机到高度并行的信息服务器和超级计算机，使他们理解根本的体系结构和软件问题，以及在设计中进行权衡的相关技术。同时，我们也希望为软件系统和应用的设计人员展示体系结构的可能发展方向，那些将决定硬件设计特定路线的动力，以及那些发展对面向性能程序设计上的影响。

近年来，在并行计算机体系结构领域最令人激动的进展是传统上完全不同的各种做法的融合，把共享存储、消息传递、数据并行和数据驱动等的计算都融合在一种公共的机器结构上。这种融合的原因部分在于共同的技术和经济力量的驱动，部分在于对并行软件的更好理解。它使得我们能开发一种公共的框架，在其中来理解和评估体系结构方面的权衡，而不是将注意力集中在各种奇特的设计和分类法上。再者，流行的并行程序设计模型在许多机器类型上都适用，这使得并行程序设计更加可移植，从而使我们得以发展有意义的标准测试和评估方法。这种领域的成熟使得硬件和软件相互作用的定量和定性的分析研究成为可能。事实上，领域的发展本身也要求我们这样做。针对一组对所有并行体系结构都很关键且涉及现代系统设计整个范围的基本问题——数据访问、通信性能、协同工作、有用语义的正确实现等。本书给出了旨在解决它们的硬件和软件技术，并考察了各种技术是如何相互作用的。仔细选择的、深入的案例分析提供了一种关于一般性原理的具体说明，展示了不同机制间的具体相互作用。

写这本书的动机之一是由于缺乏一种足够好的教材，供我们在伯克利、普林斯顿、斯坦福等校教学使用。有些教材以一种泛泛的方式将材料呈现出来，综述各种体系结构和研究成果，但没有深入分析它们，也没有提供一种现代工程的框架。另外一些教材集中在专门的项目上，但没有介绍在各种不同设计方案中体现的基本原理。在这个领域的研究报告提供了大

量想法和试验数据，但没能够提炼到一种有机构成的境界。在技术和体系结构融合的背景下，通过将注意力集中在最重要的问题上，而不是集中在将我们带到如今的丰富多彩的历史中，我们希望能够提供一种关于这个激动人心且迅速变化领域的更深刻、更清晰的理解。这是一个协作努力的结果，它反映在本书封面上我们的名字次序上。

本书的读者

本书的内容对多方面的读者都是很重要的，包括在计算机体系结构、系统软件和应用领域工作的研究人员、学生和工程技术人员。鉴于多处理器系统日益提高的重要性，这些内容和计算机系统结构设计师的相关性是明显的。芯片设计者必须理解什么能成为一个多处理器系统的有效基本模块。支配总线和存储系统设计的往往也是一些和并行性相关的问题。I/O系统的设计必须考虑具有可扩展性的高速网络、机群的构成，以及那些被多个处理器共享的设备。

系统软件——包括操作系统、编译器、程序设计语言、运行系统、性能调试工具——需要考虑新的情况，也将在并行计算机中获得新的发展机会。这样，理解体系结构的演化以及那些导致这种演化的力量是很重要的。在编译器和程序设计语言的研究与开发方面，针对并行计算的工作已经有相当一段时间。然而，体系结构和商用技术新的融合也许意味着编译和语言问题应该得到重新审视，需要在一个更一般的背景下讨论。硬件、操作系统和用户程序之间的传统边界也正在并行计算的意义下变化，为了更好的性能，程序经常要有对资源更直接的控制。

应用领域，诸如计算机图形学和多媒体、科学计算、计算机辅助设计、数据库、决策支持和事务处理，都可能出现一种巨大的转变。这种转变将是廉价的并行计算能够提供强大的计算能力的结果。然而，开发健壮的并行应用，在当前和未来多处理器系统上都能表现出好的加速比，是一个挑战性任务，而且它要求对系统相互作用和体系结构发展方向有深刻的理解。本书试图提供这样一种理解，促进应用领域和计算机系统结构之间的交流，从而使我们能设计出更好的体系结构——使程序设计更容易，性能更高和更健壮。

本书的组织

本书共有12章。第1章给出了并行体系结构的一个概貌。根据当前在工艺、体系结构和应用方面的趋势，它首先讨论了为什么并行计算机系统越来越重要。它简要介绍了那些影响了这个领域的各种多处理器体系结构（共享存储、消息传递、数据并行、数据流和脉动阵列），展现了工艺和体系结构的发展趋势是如何导致了一种领域的共识，即并行计算机系统应是由一种通信体系结构互连起来的一组通用处理节点。这种融合并不意味着创新的结束，恰恰相反，我们将看到一个迅速进展的时期。设计人员开始有了共同语言，相互交流，而不是路遇而无视之。为了理解多种通信体系结构和实现，第1章建立了一种层次式框架（包括程序设计模型、通信抽象、用户/系统界面和硬件/软件界面）。从这个框架中看这个领域的合流，第1章的最后部分展开了那些必须在各层界面中都要考虑的根本设计要点：命名、定序、复制和通信性能（开销、时延和带宽）。这些要点形成了一种贯穿本书其余部分的基本主线。第1章最后给出了若干历史文献。

第2章介绍了并行程序设计。描述了一组具有启发性的多处理器系统应用的例子，在本

书的其他部分也将用到它们。第2章展现了基于各种主要程序设计模型的并行程序，其中含有系统必须支持的基本成分。我们用案例分析的方法解释了在并行程序设计中进行分解、分配、协作和映射的步骤，且指出了这些步骤中关键的性能目标。

第3章给出了好的并行程序设计人员用于从底层体系结构中改进性能的基本技术。它提供了一种对硬件/软件权衡的理解，解释了什么样的性能特点能通过体系结构方法来加以考虑，什么样的性能特点必须或者由编译器或者由程序设计人员才能解决。和串行计算的一个类比是，体系结构不能将一个$O(n^2)$的算法变成一个$O(n\log n)$算法，但它能够改善对于那些公用存储访问模式的平均访问时间。第3章清楚地表明了那些存在于各种程序设计模型的核心算法和程序设计的挑战，也指出了和特定模型相关的若干问题。这一部分的内容表明了体系结构的进步除了提高性能外，还可能减轻并行程序设计的负担。程序设计技术在任何关于设计权衡的量化评估中都是一个关键的因素。在第3章的最后，把这些编程技术应用到典型应用程序中，给出了相应的高性能程序。

第4章讨论了在进行设计权衡时采用工作负载驱动评估方法的难题。即使对现代单处理器来说，体系结构的评估也是很困难的，通常我们只是针对一组固定的程序，在一定范围内考虑设计变化的影响，例如流水线或存储系统的组织等。在并行体系结构中，我们能够考虑变化的空间自由度要大得多。不同设计侧面之间的相互作用更加深刻，硬件和软件之间的相互作用更加重要，也在更大的范围里有影响。我们通常对机器和程序规模变化时的性能感兴趣，而改变其一往往都要影响另一方面。如果我们的评估方法不合适，就很容易导致片面的甚至是错误的结论。第4章讨论应用和体系结构的有关参数是如何相互作用的，它们应该怎样一起改变，同时还给出了将用于其后各章的基准测试程序。它提供了方法论指南，通过模拟来评估真实的机器和体系结构的思想。附录给出了若干关于并行性能基准测试的参考材料。

第5、6章是关于基于总线的对称共享存储多处理器（SMP）的一个完整介绍。除台式机外，这类系统几乎是所有现代商用机器的基础。第5章给出了一种关于“侦听”总线协议的高层逻辑设计。这种协议保证了在多个高速缓存之间自动复制数据的一致性。第5章还讨论了一个重要问题，即存储一致性问题。这个问题使我们开始理解对算法设计人员来说共享存储到底意味着什么。这一章讨论了多种设计选项，以及机器该如何针对在用户程序和操作系统中的典型存储访问模式进行优化。除了对SMP的概念性理解外，第5章还反映了并行软件牵涉的问题，包括应用软件和同步支持。

第6章进一步考察了协议的要点以及基于总线的多处理器系统的物理设计。它深入到用最新总线的多级高速缓存支持现代微处理器中出现的工程设计问题，这些高速缓存是高度流水线的。此外，还讨论了第5章提出的高层协议是如何在这些系统中实现并扩充的。第6章给出了在这一领域中有关设计要点的一个相当完整的介绍。其内容的重要性不仅由于这些小规模的设计形成了大规模设计的基石，还由于这里的许多概念在本书后面也会出现，只不过是在一个更大规模的意义上，带有更广泛的一些考虑面已。本章还包含了关于SGI Challenge和Sun Enterprise这两种服务器的独立案例分析。

第7~10章讨论的是可扩展多处理器体系结构。在当前，它们代表的是高端计算。随着技术的进步，它们也代表着未来中等水平的计算设施。

第7章展现了一类机器的硬件组织和体系结构，它们能够扩展到很大的配置。关键的概

念是网络事务，其重要性类似于第 5、6 章介绍小型设计中的总线事务，都是具有根本意义的。然而，在可扩展机器里，全局的仲裁和全局可见的信息不见了，而且可以有大量的网络事务待完成。第 7 章讨论了程序设计模型是如何通过网络事务的方法实现的，按照网络事务由直接硬件解释的程度，研究了一系列设计要点，包括对 nCUBE/2、Thinking Machine CM-5、Intel Paragon、Meiko CS-2、CRAY T3D 和 CRAY T3E 等系统的案例分析。结合 Myrinet NOW 和 DEC Memory Channel 的案例分析，本章在这个框架下还考察了现代机群。此外，对所有这些设计还进行了一个性能比较。

第 8 章将前面几章的结果综合起来，展现了如何在可扩展系统上通过自动硬件复制和高速缓存一致性，来实现一个共享的物理地址空间。这种样式的机器在业界日益流行。第 8 章全面研究了关于基于目录的高速缓存一致性协议和硬件设计备选方案，包括对 SGI Origin2000 和 Sequent NUMA-Q 的案例分析。它考察了在这些机器上工作负载的行为，进一步讨论了程序设计的内含和同步等问题。

第 9 章考察了针对共享地址空间系统的一系列备选方案，它们扩展了硬件/软件权衡的边界以获得更高的性能，降低硬件的成本和复杂性，或两者兼得。它讨论了放松存储同一性模型，由硬件在主存中一致复制数据的唯有高速缓存的存储器体系结构，以及基于软件的一致性复制。在写本书的时候，这里的许多内容正在经历一个从学术研究到商用产品的过渡阶段，它们的作用将随着机群技术的出现进一步明确。它揭示了若干在本书其他部分没有涉及但十分重要的设计概念。

第 10 章讨论可扩展的高性能通信网络的设计。通信网络是前面各章讨论的所有可扩展机器的基础，推迟到第 10 章来讲，是因为我们首先需要对驱动这些网络的处理器、存储系统和网络接口的设计有一个完整的了解。第 10 章建立了一个通用的框架，来理解网络中何处会出现硬件成本、传送延迟和带宽的限制等问题。针对这些性能价格比指标，它考察了各种路由技术、交换机设计和互连拓扑之间的权衡。这些权衡通过最近的一些设计的案例分析得到了具体体现。

基于有前面 10 章奠定的基础，第 11 章考察了一组交叉问题，它们涉及如何包容多处理器系统中出现的显著的时延而不至于影响总体性能。这些技术有两个基本的方面：让有用的工作覆盖时延，让传送的数据流水传送。这些技术的最简单的形式在本质上即批量传送，大量规则的数据序列流水传送，而且通常可以从处理器下载。其他的技术试图隐藏在多个独立的装载和存储操作中发生的时延。写时延利用弱同一性模型的特点来屏蔽，这种模型的基本出发点是认识到程序操作的序关系只是由程序中对共享存储的一个小的访问集合来表达的。读时延由隐式或显式的数据预取来屏蔽，在现代动态调度的处理器中，也可以通过前瞻技术来屏蔽。这其中有些技术还被扩展来隐藏同步时延。第 11 章对这些不同做法提供了一个透彻的分析，同时还考虑了对编译技术的影响，以及关于有效性的定量评估。

最后，第 12 章考察了那些有可能决定这个领域未来的技术、体系结构、软件系统和应用方面的发展趋势。从硬件/软件的观点，阐述了这个领域将如何演化，会遇到什么问题以及潜在的突破。

本书的使用

本书的这种组织方式是为了满足多方面读者的需要。它可以作为研究生教材、工程师的

专业参考书，以及一般的参考读物，对那些其工作日益和并行计算有关的人们都有帮助。如果深入到各个方面，本书所包含的材料足以用于一年的并行计算内容的学习，从各种机器的设计到并行程序设计的经验。然而，本书也能分成几部分来使用。

第1章旨在提供一个独立的关于并行计算机系统结构的理解，作为研究生或者大学高年级普通计算机系统结构课程的一部分是很合适的。对那些需要了解并行计算的术语和基本概念，从而理解这种技术将如何影响他们工作的工程管理人员或公司负责人，这一章也是有价值的。它清楚地告诉你当对于并行计算的兴趣和需要增加时该怎样进一步地学习。第1章还可以作为编译器、数据库、操作系统或程序设计课程在并行体系结构方面的基本背景。第1章和第12章一起构成了一个关于并行计算机体系结构领域的“外层框架”。

面向机器组织和设计的并行体系结构课程，除了第1章的概述外，可由第5、6、7、8、10章构成，它们是本书的核心。然而，和传统课程中的内容相比，这些章节在设计方面要深入得多。这是因为我们所用的材料有些以前没有发表过，而且没有以一种面向设计的框架来组织。这些材料提供了关于设计权衡的详细的定量性讨论。对于高速缓存一致性系统的正确性问题，第5、6章提出了关键的要求，展示了如何在日益复杂的设计中高性能地满足它们。第7章分析了可扩展机器，所采用的方式和通常商业做法和发表的研究成果都不一样，并且在这个框架中讨论了新兴的高性能机群。第8章描述主要的商用分布存储计算机中的高速缓存一致性协议，所采用的框架和细节层度也是在其他书中没有见到的。第10章是关于网络设计的一个简要而完整的讨论。这几章中的讨论足够深入，即使是有一定素养的系统设计人员，也能够从中获得一个新的理解和一个清晰的设计框架。贯穿这几章（还有第9章的开始部分）的，我们还有了关于存储同一性模型的一个严谨且实用的讨论，例如讨论同步操作的实现。第11章是关于日益重要的时延包容问题的，它可以作为这些关于机器组织和设计章节的补充。

这本教材为教学提供了令人兴奋的机会，在核心材料有机结合起来的基础上，从多个方向来加强基本并行体系结构课程都是可能的。首先，第2、3章透彻的处理使我们跨越了硬件/软件的边界。这就使学习体系结构的学生对设计决策可能带来的影响有更深刻的理解，以及了解并行程序设计到底是什么意思。这也使课程的吸引力扩大到包括操作系统、语言和应用在内的学生，他们可以从软件的观点来看体系结构的问题。第二个使得基本课程能得以加强的方向是硬件和软件设计决策的定量化性能分析。基于对第2、3章的理解，第4章、附录和后面几章中的“并行软件牵涉的问题”各小节将这一线索自始至终贯穿于核心机器设计材料。除了提供性能评价的方法论指导外，它们还提供了一个关键的视角，用以看待发表的结果。第三个方向是强调硬件/软件的权衡。这是由定量化分析所形成的一个基础性问题，在各章的同步和程序设计小节中得到了进一步论述。在第9章，这一问题更加明显，其中我们仔细研究了在提供一致的共享地址空间时责任的划分。在第11章所讨论的容许时延问题也是关于这一方面的。每一个方向都代表着一群专业人员，他们有日益增加的需要，来更深刻地理解如何对待并行体系结构。

本书也能作为需要实际操作的并行程序设计课程的主要教材。基于第1章的一般性介绍，第2、3章给出了一个坚实的框架，来理解并行程序的行为。通过第4章的工作负载分析以及第5、7、8、9章的“并行软件牵涉的问题”各小节，这一点进一步得到加强。这一部分的材料应该由用于课程的与并行程序设计环境相关的参考书来补充，例如MPI、并行线程或

者 HPF。第 6～8 章的案例分析提供了一个关于机器的彻底的讨论，学生很可能使用。第 11 章提供了一个方便的框架，来考察在并行程序设计中解决通信任务的最佳途径。

我们相信并行计算机体系结构在研究和实践方面都是一个令人兴奋的核心领域，它的重要性也会与日俱增。它已经达到这样的成熟程度，编写一本严肃的基于设计和工程原理的教科书是很有意义的。基于多年积累的丰富多样的思想和方法，这个领域正出现一种急剧融合的趋势。现在已经到了超越浏览各种机器的设计，进入理解基本设计原理的时候了。我们亲身经历了这个领域的融合过程；这本书来自我们的经验，希望它传达我们对于这个巨变和成长中的领域所感受到的某些兴奋之情。由于并行体系结构变化是如此迅速，案例分析、性能分析和工作负载需要定期地更新。除了辅助教学材料外，这本书的 Web 站点还将提供有关及时更新的材料。我们也希望你能够通过课程和商用开发的高质量产品，对这个站点做出贡献。

我们也欢迎读者指出任何错误或疏漏，以便在以后印刷时改正它们。为此，请发电子邮件到 pcabugs@mkp.com。同时，也请检查 www.mkp.com/pca 上的勘误表，看有关的错误是否已经公布和更正。

致谢

本书以各种形式已孕育了相当一段时间，而且得益于许多人的努力。它源于我们并行处理课程和研究课题的笔记和讲义。我们的学生和职员在整个过程中起到的作用是不可估量的。尽管这是第一版，但是随着这些材料被组织起来，手稿在 Web 上已存在多时了。鉴于 Web 的方式，我们不知道世界上哪些学校和研究机构在它们的教学和科研中用到了它，但我们收到了来自世界各地的建议。许多人直接或间接地，甚至默默地对它做出了贡献，因此我们在此对大家表示衷心的感谢。

许多学生通过他们的问题、想法、解答和项目改进了本书。我们要感谢选修下列课程的学生们：伯克利的 CS 258（并行处理器）和 CS 267（并行计算机的应用）课程，普林斯顿的 CS 598（并行计算机体系结构和程序设计）课程，以及斯坦福的 CS 315A（并行计算机体系结构和程序设计）和 CS 315B（并行程序设计实习）。其中特别要提到的是伯克利的 Andrea Arpaci-Dusseau，Remzi Arpaci-Dusseau，Brent Chun，Seth Goldstein，Alan Mainwaring，Rich Martin，Lok Tin Liu，Steve Lummetta，Chad Yoshikawa 和 Frederick Chun Bong Wong；普林斯顿的 Angelos Bilas，Liviu Iftode，Dongming Jiang，Steven Kleinstein，Sanjeev Kumar，Hongzhang Shan 和 Yuanyuan Zhou，还有斯坦福的 Cheng Chen，John Heinlein，Moriyoshi Ohara，Evan Torrie 和 Steven Cameron Woo。他们通过不懈的努力为本书提供了有价值的见解、数据和分析。其中 Jiang，Kumar，Ohara，Torrie，Wong 和 Woo 的贡献值得我们特别表示衷心的谢意。

许多在学术和业界的人在审阅我们的手稿时提供了无价的帮助，告诉我们一些原理在实际中是怎么发挥作用的，试用这本教材，对我们提出指导性意见。我们特别感谢 Sarita Adve，Arvind，Russell Clapp，Michel Dubois，Mike Galles，Kourosh Gharachorloo，Jim Gray，John Hennessy，Mark Hill，Phil Krueger，James Laudon，Edward Lazowska，Dan Lenoski，W.R. Michalson，Todd Mowry，Greg Papadopoulos，Dave Patterson，Randy Rettberg，Shuichi Sakai，Klaus Schauser，Ashok Singhal，Burton Smith，Jim Smith，Mark Smotherman，Per Stenstrom，Thorsten von Eicken，Maurice Wilkes，David Wood 和 Chengzhong Xu。感谢 John 和 Dave 对我们自始至终的指导。许多

人通过用这本书的某些部分教学对我们有所帮助，最早的有 Sarita Adve，Andrew Chien，Jim Demmel，Wallid Najjar，Constantine Polychronopoulos，Radhika Thekkath 和 Kathy Yelick。

我们也要感谢国家科学基金委员会、国防部高级研究计划局、能源部和若干企业。它们所支持的研究工作是本书材料的基础，也推动了并行计算领域的蓬勃发展。

我们感谢 Morgan Kaufmann 出版社的工作小组，他们管理本书出版的过程给人留下了深刻印象。Denise Penrose 承担了这个任务，以一种令人难以置信的精力、专注和热情领导了小组工作。和她一起工作是绝对快乐的。Elisabeth Beller 稳健地管理了整个生产过程。Meghan Keeffe 和 Jane Elliott 协调了审稿和图片的查找，解决了许多遗漏问题。一组校对人员对全书文字上的正确性提供了保证。还要感谢 Jennifer Mann，她在 Denise 之前负责本书出版的管理工作；感谢 Bruce Spatz，他在 Morgan Kaufmann 出版社从始至终指导了这本书的出版工作。

还必须感谢我们大学的职员 Gabriela Aranda，Ginny Hogan，Chris Kranz，Terry Lessard-Smith，Bob Miller，Thoi Nguyen，Matt Norcross，Charlie Orgish，Jim Roberts 和 Chris Tengi。他们在整个过程中提供了无数大大小小的帮助。

最重要的是，将最深的谢意、感激和爱献给我们的家庭。他们毫无保留的支持、耐心和智慧，贯穿于我们整个写作过程中。

目　录

出版者的话
专家指导委员会
序
译者序
前言
第1章　引论……1
1.1　为什么要用并行体系结构……3
1.1.1　计算机应用发展的趋势……4
1.1.2　微电子技术趋势……9
1.1.3　体系结构趋势……10
1.1.4　超级计算机……15
1.1.5　小结……17
1.2　并行体系结构的融合……18
1.2.1　通信体系结构……18
1.2.2　共享地址空间……20
1.2.3　消息传递……27
1.2.4　融合……29
1.2.5　数据并行处理……32
1.2.6　其他并行体系结构……34
1.2.7　一个通用并行体系结构……36
1.3　基本的设计问题……37
1.3.1　通信抽象……38
1.3.2　编程模型的要求……38
1.3.3　通信和复制……41
1.3.4　性能……42
1.3.5　小结……45
1.4　结论……45
1.5　历史资料……47
习题……50
第2章　并行程序……54
2.1　并行应用的案例分析……55
2.1.1　洋流的模拟……55
2.1.2　星系演化的模拟……56
2.1.3　用光线跟踪法来实现复杂场景的可视化……57
2.1.4　针对关联性的数据挖掘……58
2.2　并行化过程……58
2.2.1　程序并行化过程中的几个步骤……59
2.2.2　计算并行和数据并行……65
2.2.3　并行化过程的目标……65
2.3　一个例子程序的并行化……66
2.3.1　方程求解器的内核……66
2.3.2　分解……68
2.3.3　分配……71
2.3.4　在数据并行模型下的协调……72
2.3.5　在共享地址空间模型下的协调……73
2.3.6　在消息传递模型下的协调……78
2.4　结论……84
习题……84
第3章　面向性能的程序设计……88
3.1　划分阶段的性能问题……89
3.1.1　负载平衡和同步等待时间……90
3.1.2　减少固有的通信……95
3.1.3　减少额外的工作……98
3.1.4　小结……99
3.2　在多存储器系统中的数据访问和通信……100
3.2.1　看作扩展的存储层次结构的多处理器系统……100
3.2.2　在扩展的存储层次结构中的附加通信……101
3.2.3　用工作集的观点看附加的通信和数据的复制……102
3.3　性能的协调……103
3.3.1　减少附加通信……103
3.3.2　将通信结构化以降低代价……108
3.4　从处理器角度看到的性能因素……113
3.5　并行应用程序案例的深入分析……115
3.5.1　Ocean……116
3.5.2　Barnes-Hut……120
3.5.3　光线跟踪……125
3.5.4　数据挖掘……128
3.6　编程模型涉及的问题……131

3.6.1 命名 …… 132
3.6.2 复制 …… 132
3.6.3 通信的开销和粒度 …… 133
3.6.4 块数据传送 …… 134
3.6.5 同步 …… 134
3.6.6 硬件代价和设计复杂性 …… 135
3.6.7 性能模型 …… 135
3.6.8 小结 …… 136
3.7 结论 …… 136
习题 …… 137
第4章 工作负载驱动的性能评价 …… 142
4.1 改变工作负载和机器的规模 …… 144
4.1.1 多处理器性能的基本测量 …… 144
4.1.2 为什么要考虑扩放性 …… 145
4.1.3 扩放的关键问题 …… 148
4.1.4 扩放模型和加速比的测量 …… 148
4.1.5 扩放模型对方程求解器内核的影响 …… 151
4.1.6 扩放工作负载参数 …… 153
4.2 评价一台实际的机器 …… 154
4.2.1 使用微基准测试程序分离性能 …… 154
4.2.2 选择工作负载 …… 156
4.2.3 评价一台固定规模的机器 …… 158
4.2.4 改变机器的规模 …… 163
4.2.5 选择性能指标 …… 164
4.3 对一个体系结构概念或设计权衡的评估 …… 166
4.3.1 多处理器的模拟 …… 167
4.3.2 缩小模拟的问题和机器参数的规模 …… 168
4.3.3 处理参数空间：评价举例 …… 171
4.3.4 小结 …… 174
4.4 说明工作负载的特征 …… 175
4.4.1 工作负载案例分析 …… 175
4.4.2 工作负载的特征化 …… 182
4.5 结论 …… 188
习题 …… 189
第5章 共享存储的多处理器 …… 193
5.1 高速缓存的一致性 …… 196
5.1.1 高速缓存一致性问题 …… 196
5.1.2 通过总线侦听的高速缓存一致性 …… 199
5.2 存储同一性 …… 204
5.2.1 顺序同一性 …… 206
5.2.2 保证顺序同一性的充分条件 …… 208
5.3 总线侦听协议的设计空间 …… 210
5.3.1 一种三态(MSI)回写作废式协议 …… 211
5.3.2 一种四态(MESI)回写作废式协议 …… 215
5.3.3 一种四态(Dragon)回写更新式协议 …… 217
5.4 关于协议设计中若干折中的评估 …… 220
5.4.1 方法论 …… 221
5.4.2 在MESI协议下的带宽需求 …… 223
5.4.3 协议优化的影响 …… 224
5.4.4 高速缓存中存储块大小的权衡 …… 227
5.4.5 基于更新和基于作废协议的对比 …… 238
5.5 同步 …… 241
5.5.1 同步事件的组成部分 …… 242
5.5.2 用户和系统的角色 …… 243
5.5.3 互斥 …… 244
5.5.4 点对点事件同步 …… 254
5.5.5 全局(栅障)事件的同步 …… 255
5.5.6 同步问题小结 …… 259
5.6 对软件的影响 …… 259
5.7 结论 …… 264
习题 …… 265
第6章 基于侦听的多处理器的设计 …… 273
6.1 正确性需求 …… 273
6.2 基础设计：采用原子总线的单级高速缓存 …… 275
6.2.1 高速缓存控制器和标记的设计 …… 276
6.2.2 侦听结果的报告 …… 277
6.2.3 对回写的处理 …… 278
6.2.4 基础系统组织 …… 279
6.2.5 非原子性的状态转移 …… 279
6.2.6 串行化 …… 281
6.2.7 死锁 …… 282
6.2.8 活锁和挨饿 …… 283
6.2.9 原子操作的实现 …… 283
6.3 多级高速缓存层次结构 …… 284
6.3.1 包含性的维护 …… 285
6.3.2 在高速缓存层次结构中传播一致性的事务 …… 287

6.4 事务拆分型总线 …… 289
6.4.1 事务拆分型总线设计的一个例子 …… 290
6.4.2 总线设计和请求－响应的匹配 … 290
6.4.3 侦听结果和冲突的请求 …… 292
6.4.4 流控制 …… 293
6.4.5 一次缓存扑空的路线 …… 293
6.4.6 串行化和顺序同一性 …… 294
6.4.7 其他设计选择 …… 296
6.4.8 带有多级高速缓存的事务拆分型总线 …… 297
6.4.9 对一个处理器有多个待完成扑空的支持 …… 299
6.5 实例分析：SGI Challenge 和 Sun Enterprise 6000 …… 301
6.5.1 SGI Powerpath-2 系统总线 …… 302
6.5.2 SGI 处理器和内存子系统 …… 304
6.5.3 SGI I/O 子系统 …… 306
6.5.4 SGI Challenge 内存系统性能 …… 307
6.5.5 Sun Gigaplane 系统总线 …… 308
6.5.6 Sun 处理器和内存系统 …… 309
6.5.7 Sun I/O 子系统 …… 311
6.5.8 Sun Enterprise 内存系统性能 …… 311
6.5.9 应用程序性能 …… 312
6.6 高速缓存一致性的扩充 …… 314
6.6.1 共享缓存的设计 …… 314
6.6.2 虚拟标引缓存的一致性 …… 317
6.6.3 转换检测缓冲器的一致性 …… 318
6.6.4 环上基于侦听的高速缓存一致性 …… 320
6.6.5 在基于总线的系统中的数据和侦听带宽的扩展 …… 322
6.7 结论 …… 323
习题 …… 324
第 7 章 可扩展多处理器 …… 329
7.1 可扩展性 …… 331
7.1.1 带宽的可扩展性 …… 331
7.1.2 时延的可扩展性 …… 333
7.1.3 成本的可扩展性 …… 334
7.1.4 物理可扩展性 …… 336
7.1.5 通用并行体系结构的可扩展性 … 339
7.2 编程模型的实现 …… 340
7.2.1 基本的网络事务 …… 341
7.2.2 共享地址空间 …… 343
7.2.3 消息传递 …… 346
7.2.4 主动消息 …… 349
7.2.5 共同的挑战 …… 350
7.2.6 通信体系结构设计空间 …… 351
7.3 物理 DMA …… 352
7.3.1 节点到网络的接口 …… 352
7.3.2 通信抽象的实现 …… 354
7.3.3 案例分析：nCUBE/2 …… 354
7.3.4 典型的局域网接口 …… 355
7.4 用户级访问 …… 356
7.4.1 节点到网络的接口 …… 356
7.4.2 案例分析：Thinking Machines CM-5 …… 357
7.4.3 用户级的处理程序 …… 358
7.5 专用消息处理 …… 360
7.5.1 案例分析：Intel Paragon …… 362
7.5.2 案例分析：Meiko CS-2 …… 365
7.6 共享的物理地址空间 …… 367
7.6.1 案例分析：CRAY T3D …… 369
7.6.2 案例分析：CRAY T3E …… 371
7.6.3 小结 …… 372
7.7 工作站机群和工作站网络 …… 372
7.7.1 案例分析：Myrinet SBUS Lanai … 374
7.7.2 案例分析：PCI 存储器通道 …… 376
7.8 并行软件涉及的问题 …… 378
7.8.1 网络事务的性能 …… 378
7.8.2 共享地址空间操作 …… 382
7.8.3 消息传递操作 …… 383
7.8.4 应用层性能 …… 384
7.9 同步 …… 390
7.9.1 加锁算法 …… 390
7.9.2 栅障算法 …… 393
7.10 结论 …… 397
习题 …… 398
第 8 章 基于目录的高速缓存一致性 … 400
8.1 可扩展的高速缓存一致性 …… 403
8.2 基于目录方法概述 …… 404
8.2.1 简单目录方案的操作 …… 404
8.2.2 可扩展性 …… 407
8.2.3 组织目录表的其他方法 …… 408
8.3 目录协议和折中的评价 …… 412
8.3.1 目录方案的数据共享模式 …… 413

8.3.2 本地和远程通信流量 …… 418
8.3.3 高速缓存块尺寸的影响 …… 423
8.4 目录协议设计上的挑战性问题 …… 423
8.4.1 性能 …… 423
8.4.2 正确性 …… 427
8.5 基于存储器的目录协议：SGI 的 Origin 系统 …… 432
8.5.1 高速缓存一致性协议 …… 432
8.5.2 关于正确性问题 …… 438
8.5.3 目录结构的细节 …… 441
8.5.4 协议扩展 …… 442
8.5.5 Origin2000 硬件概述 …… 443
8.5.6 Hub 的实现 …… 445
8.5.7 性能特征 …… 447
8.6 基于高速缓存的目录协议：Sequent 的 NUMA-Q …… 451
8.6.1 高速缓存一致性协议 …… 452
8.6.2 关于正确性问题 …… 458
8.6.3 协议扩展 …… 459
8.6.4 NUMA-Q 硬件一览 …… 460
8.6.5 协议和 SMP 节点的交互 …… 462
8.6.6 IQ 链路的实现 …… 463
8.6.7 性能特征 …… 464
8.6.8 对比案例分析：HAL S1 多处理器 …… 466
8.7 性能参数和协议性能 …… 467
8.8 同步 …… 469
8.8.1 几种同步算法的性能 …… 470
8.8.2 实现原子性原语 …… 471
8.9 对并行软件的影响 …… 472
8.10 高级论题 …… 474
8.10.1 减少目录存储的开销 …… 474
8.10.2 层次式的一致性 …… 477
8.11 结论 …… 484
习题 …… 485
第 9 章 硬件/软件功能的折中 …… 491
9.1 放松的存储同一性模型 …… 492
9.1.1 系统规范说明 …… 496
9.1.2 程序员接口 …… 501
9.1.3 翻译机制 …… 504
9.1.4 真实的多处理器系统中的同一性模型 …… 504
9.2 克服容量限制 …… 505
9.2.1 第三层高速缓存 …… 505
9.2.2 惟有高速缓存的存储器体系结构 …… 506
9.3 降低硬件成本 …… 509
9.3.1 具有去耦辅助部件的硬件访问控制 …… 510
9.3.2 通过代码修改实现的访问控制 …… 510
9.3.3 基于页面的访问控制：共享虚拟存储器 …… 511
9.3.4 语言和编译器支持的访问控制 …… 520
9.4 综合：分类和简单的 COMA …… 522
9.4.1 综合：简单的 COMA 和 Stache …… 523
9.5 对并行软件的影响 …… 526
9.6 高级论题 …… 527
9.6.1 灵活性和 CC-NUMA 系统中的地址约束 …… 527
9.6.2 以软件实现放松的存储同一性 …… 528
9.7 结论 …… 533
习题 …… 533
第 10 章 互连网络设计 …… 539
10.1 基本定义 …… 540
10.2 基本的通信性能 …… 543
10.2.1 时延 …… 543
10.2.2 带宽 …… 547
10.3 组织结构 …… 549
10.3.1 链路 …… 550
10.3.2 交换机 …… 551
10.3.3 网络接口 …… 552
10.4 互连拓扑结构 …… 553
10.4.1 全连接网络 …… 553
10.4.2 线性阵列和环 …… 553
10.4.3 多维网格和多维花环 …… 554
10.4.4 树 …… 555
10.4.5 蝶网 …… 557
10.4.6 超立方体 …… 560
10.5 对网络拓扑设计折中的评价 …… 561
10.5.1 无负载时延 …… 562
10.5.2 负载情况下的时延 …… 565
10.6 路由 …… 569
10.6.1 路由机制 …… 569
10.6.2 确定性路由 …… 570
10.6.3 免死锁 …… 570
10.6.4 虚通道 …… 573

10.6.5 上行* - 下行* 路由 …… 574
10.6.6 折转模型路由 …… 575
10.6.7 自适应路由 …… 577
10.7 交换机的设计 …… 578
10.7.1 端口 …… 578
10.7.2 内部数据通路 …… 579
10.7.3 通道缓冲 …… 580
10.7.4 输出调度 …… 583
10.7.5 堆叠式维度交换机 …… 585
10.8 流控 …… 585
10.8.1 并行计算机网络与局域网、广域网的对照 …… 586
10.8.2 链路级的流控 …… 587
10.8.3 端到端的流控 …… 590
10.9 案例分析 …… 591
10.9.1 CRAY T3D 网络 …… 591
10.9.2 IBM SP-1 、SP-2 网络 …… 593
10.9.3 可扩展一致性接口 …… 595
10.9.4 SGI 的 Origin 网 …… 596
10.9.5 Myricom 网络 …… 597
10.10 结论 …… 598
习题 …… 598
第 11 章 时延的包容 …… 601
11.1 时延包容技术概述 …… 603
11.1.1 时延包容与通信流水线 …… 603
11.1.2 采用技术 …… 605
11.1.3 基本要求、优点与局限性 …… 607
11.2 显式消息传递中的时延包容 …… 612
11.2.1 通信结构 …… 612
11.2.2 块数据传送 …… 612
11.2.3 预通信 …… 613
11.2.4 跨越同一线程中的通信 …… 614
11.2.5 多线程技术 …… 614
11.3 共享地址空间中的时延包容 …… 614
11.4 共享地址空间中的数据成块传送 …… 615
11.4.1 技术和机制 …… 616
11.4.2 策略问题和折衷方案 …… 617
11.4.3 性能收益 …… 618
11.5 跨越长时延事件 …… 622
11.5.1 跨越写操作 …… 623
11.5.2 跨越读操作 …… 626
11.5.3 小结 …… 632
11.6 共享地址空间中的预通信 …… 632
11.6.1 没有共享数据高速缓存的共享地址空间 …… 632
11.6.2 缓存一致的共享地址空间 …… 634
11.6.3 性能收益 …… 641
11.6.4 小结 …… 645
11.7 共享地址空间中的多线程技术 …… 645
11.7.1 技术和机制 …… 646
11.7.2 性能收益 …… 656
11.7.3 阻塞方式的实现问题 …… 658
11.7.4 交替方式的实现问题 …… 660
11.7.5 在多发布处理器中集成多线程 …… 662
11.8 免锁定的缓存设计 …… 664
11.9 结论 …… 666
习题 …… 667
第 12 章 将来的发展方向 …… 673
12.1 技术与体系结构 …… 673
12.1.1 演变趋势 …… 674
12.1.2 遇到的阻碍 …… 676
12.1.3 潜在的突破 …… 679
12.2 应用程序和系统软件 …… 687
12.2.1 演变趋势 …… 687
12.2.2 遇到的困难 …… 690
12.2.3 潜在的突破 …… 691
附录 并行基准测试程序集 …… 692
参考文献 …… 696
索引 …… 722

第1章 引 论

过去十多年来，我们欣喜地见证了计算机系统性能和容量的爆炸性增长。促成这种巨大成功的关键是底层 VLSI 技术的迅猛发展，它使得时钟频率不断提升，单一芯片中集成的元器件数量越来越多。而成功的历程则是围绕计算机体系结构展开的，它将半导体工艺技术的原始潜能转化为计算机系统更高的性能和更大的容量。并行性的开发是其中的主要角色。较多的资源意味着可以按并行方式来同时执行更多的操作。并行计算机体系结构所研究的是怎样组织这些资源，才能使它们更好地一起工作。各种类型的计算机都是通过更加有效地利用并行性，来从现有的工艺技术中获取高的性能，并且开发并行性的层次也在不断提升。另一关键的角色是存储系统。数据处理的速度越来越快，它们必须存放于系统的某处。因而，并行处理的研究是与数据局部性和通信紧密相连的。在计算机系统不同层次的设计中，设计人员必须把握这些不断变化的关系，才能在不同时期的工艺技术和成本的约束下获得最高性能和最佳的可编程性。

并行性的概念适用于设计的各个层次。在本质上，它与所有其他的体系结构概念之间都存在相互影响，且对底层工艺技术有着独特的依赖性，因而从并行性的角度来理解计算机体系结构很有意义。特别是，在并行计算机系统设计的许多层次，都需要考虑局部性、带宽、时延、同步等基本问题。于是，就必须针对实际应用工作负载，从多方面进行权衡。

同一般设计问题一样，并行计算机体系结构也由一些功能单元及其形态来刻画。如下的定义是一个很好的概括（Almasi and Gottlieb 1989）：

并行计算机是“一组相互通信、相互协作以快速求解大型问题的处理单元”。

然而，这一简单定义却引发了许多问题。我们所说的一组单元到底有多少？每个处理单元的性能有多高？处理单元数是否能够以某种简单方式不断增大？这些单元相互之间如何通信和协作？处理器之间如何传送数据，提供哪种类型的互连方式，以及可以使用哪些操作来确定不同处理器完成动作的序列？对程序员而言，什么是硬件和软件提供的基本抽象？最后，上述这些因素如何影响性能？在对这些问题的回答中，我们将看到，从满足现代计算的需求而言，无论是少量处理单元的系统，还是中等或者大量处理单元的系统都有各自重要的作用。因而，完整地理解不同规模（从小规模到大规模）的并行机器是非常重要的。一些设计原则适用于所有规模的并行性；另一些原则却与特定约束，例如所有单元集中在芯片内、机箱内或超大型机器，有着密切关系。可以有把握地说并行计算机展示了一个丰富且多样的设计空间。这种多样性使得这一领域令人兴奋，但同时也意味着发展一个适用于理解多种设计选择的简明框架是非常重要的。

并行体系结构本身变化很快。历史上，并行机器通常表现为：在给定的工艺技术下，系统设计人员为追求最高性能而提出的创新性组织结构，它们通常依赖于特定编程模型。在许多情况下，一些标新立异的机器组织方法被认为是合理的，其理由是半导体工艺技术的发展就快到头了。事实说明，这些大胆的预测太言过其实了，例如逻辑单元密度、交换机速度一直在不断改进并且可以在更低的级别上发掘更加适度的并行性，这些技术都保证了处理器性

能持续不断地得以改进。尽管如此，实际应用对计算性能的需求增长还是超出了单个处理器可以提供的性能范围，因而多处理器系统在主流计算中占据着越来越重要的位置。一个重要的变化是，这些并行计算机的体系结构不再那么追求标新立异，即使当今的大规模并行机器也是利用工作站和个人计算机所用的基本部件建造而成的。它们遵从相同的工程原理和性能价格比权衡。此外，为了获得最大可能的性能，并行机器必须力求充分发掘它的每个组成部件的全部潜在的性能。因而，理解现代并行计算机体系结构就必然包括对工程设计权衡的深层次把握，而并非仅仅是根据描述机器结构的分类性术语。

并行体系结构将在信息处理中发挥越来越核心的作用。这个论点并非过分基于单个处理器性能将很快到头这一假设，而更主要是基于对今后系统设计的考虑，即随着芯片密度的增长，在一个芯片上集成多处理器将越来越具有吸引力。本书的目的就是阐明在多处理器层次的计算机设计的原理。本书将讨论有关各种系统部件（处理器、存储系统和网络）的设计问题及这些部件之间的关系。其中的一大要点是理解在并行机器演化中硬件和软件之间的职责划分。理解这一划分要求读者熟悉并行程序对机器的要求，各种机器设计因素之间的相互作用以及并行程序设计的实践。

学习计算机体系结构的过程通常被比喻为剥洋葱皮，这一类比对于并行计算机体系结构更是适合。在理解体系结构的每一层次时，我们都能发现一个具有许多相互作用的侧面的完整体系，这些侧面包括机器的结构、它们的抽象、它们所依赖的技术、演练它们的软件以及描述它们性能的模型。然而，如果我们深入钻研其中任何一个侧面，我们都会发现另一设计层次和新的一类相互作用情况。并行计算机体系结构的这种整体性和多层次性使得这一领域对学习和讲授都具有挑战性。人们不可避免地会感知这种一层接一层的结构。

本章为导论，主要介绍并行计算机体系结构的“表层”。它首先概要说明为什么并行机器将可能变得非常普及，从桌面型计算机到超级计算机，无处不在。另外，本章还将考察有关计算机的微电子工艺方面、体系结构方面和经济方面的趋势，这些趋势导致了计算机体系结构发展到今天的现状，同时它们也为预测未来的并行计算机体系结构提供了基础。本章1.1节重点讲述推动处理器性能迅猛发展的动力，以及整个计算机工业围绕通用微处理器进行的结构调整。这些动力包括对计算性能无止境的应用需求、VLSI芯片密度和集成水平的不断改进、体系结构在越来越高的层次利用并行性。

接着本节简要概括了几种重要的体系结构风格，它们丰富了体系结构研究的历史，并对从现代眼光来理解并行机器是非常有益的。在这种设计的多样性中，出现了一组共同的设计原则和权衡，它们也同样受到潜在工艺技术发展的驱动。这些动力正在迅速引导系统结构设计领域的融合，这一现象形成了本书的重点。1.2节纵观了传统的并行机器，包括共享存储型、消息传递型、数据并行型、脉动阵列型和数据流型，并举例说明这些不同类型的结构解决体系结构共同问题的不同策略。这些讨论展现了并行体系结构依赖于底层工艺技术，更重要的是，这些讨论证明了围绕微处理器的统治地位，以及目前并行体系结构已经发生的融合势态。

建立在这种融合之上，1.3节考察了贯穿并行机器设计的基本问题：作为通信和协作的基础，在机器级别可以对什么命名；执行通信和协作操作的时延；它们所要求的整体速率和带宽。这种从概念结构到性能要素的转化为并行机器结构的研究提供的不仅是定性描述，而是定量的框架。

有了上面对并行体系结构的广泛理解，后面各章将更加深入地研究技术细节。第2、3章详细考虑并行程序的结构和要求，是理解并行体系结构和应用之间相互作用的基础。第4章建立了一个基于应用要求和性能测量的评估设计决策的框架。第5、6章全面地研究了中小规模的商用多处理器（从几个到几十个处理器）的并行计算机体系结构。这里介绍的概念和结构奠定了在最后5章中讨论更大规模设计的基础。

1.1 为什么要用并行体系结构

计算机体系结构、微电子工艺技术以及应用需求竞相发展，并互相影响。并行计算机体系结构也不例外。在设计中，要考虑一个新的因素——处理器的个数——及在一定成本下对性能的追求驱动着设计。不管一个单处理器的性能如何，原则上讲，更高的性能可以通过利用多个这样的处理器来达到。所能获得的性能增益和附加的成本取决于许多因素，我们将在本书的其他部分经常讨论到这些因素。

为了更好地理解这种相互影响，让我们考虑一下多处理器构建模块的性能特征。图1-1[○]说明了不同类型的处理器的性能随着时间的变化（Hennessy and Jouppi 1991）。虚线范围以内的区域说明了处理器发展的趋势。尽管我们不应该草率地从这些有限的数据中得出什么定量的结论，但图1-1给出的基本信息是有价值的。

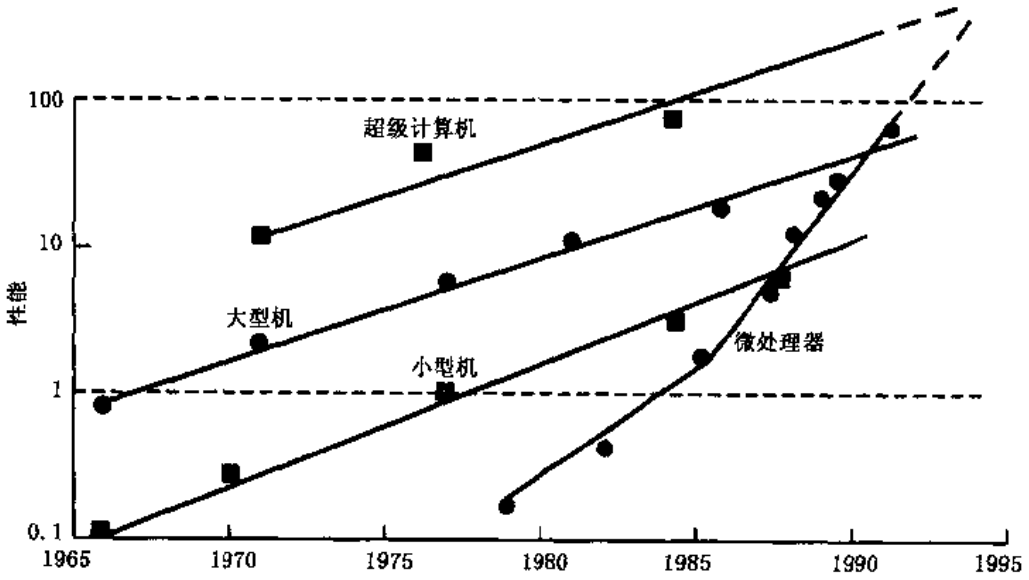


图1-1 微型机、小型机、大型机和超级计算机的性能趋势。从20世纪80年代中期以来，微处理器的性能以每年大约50%的速度增长。传统的大型机和超级计算机性能以每年大约25%的速度增长。结果我们看到适合并行计算机体系结构的处理器同时也是性能提高的领头羊。

来源：Hennessy and Jouppi 1991。

首先，高度集成的单芯片CMOS微处理器性能不断提高，并超过了规模更大，价格也更昂贵的对手。微处理器的性能以每年大约50%的速度增长。用小的、廉价的、大量生产的

○ 此图源于一篇很有影响力的论文（Hennessy and Jouppi 1991），旨在揭示计算机业所发生的巨大变化，由于跨越一个大的时间和市场范围，所用的性能度量有些难以理解。这项研究的数据来源于通用基准测试程序，例如，SPEC基准测试，它们常用来评估技术性计算应用的性能（Hennessy and Patterson 1996）。在论文发表后，微处理器的表现和预测的相符，而大型机和超级计算机经历了很大的危机，最后出现了用多个CMOS微处理器来实现系统的倾向。

处理器作为一个多处理器计算机系统基本单元的优势是明显的。尽管如此，适合并行计算机的处理器的性能曾经总是赶不上最快的单处理器系统性能的增长。但现在这种情况已经改变。和早期人们建造的各种不同规模的并行计算机相比，今天利用并行技术来获得高性能会越来越可行，因为我们现在有了更适合并行系统的基本处理器模块。

其次，可能是更基本的看法是“变化”。变化，甚至是剧烈的变化，是计算机体系结构的基本特征。不断变化的过程对研究计算机体系结构有深刻的影响，因为我们不但要理解系统是如何工作的，还要知道它将来是如何演变的和为什么会出现相应的变化。变化也是对我们写这本书的一个最大挑战，也是一个主要的促动因素。并行计算机体系结构领域已经相当成熟，到了需要从基本的工程原理和性能与成本定量评价的角度来研究的时机了。它们被归纳为事实和数据、测量、实际计算机的设计等几个方面。不幸的是，现有的数据和设计随着这个领域的发展将会过时。为了提供一个清晰的基础，本书提供的数据和实际的计算机设计都是20世纪90年代后期的。尽管如此，在分析具体的设计和权衡时所采用的评估方法是跨越年代的。

4

20世纪90年代后期是一个有特殊意义的时期。随着单个芯片的微处理器在计算的各个层面逐步占有统治地位，也随着并行计算在主流计算许多领域的普及，我们正处于技术重新融合的一个时期。当然，飞速的变化应该使我们谨慎地预测未来。这一节的剩余部分更深入的阐述了并行计算机体系结构进入计算的主流领域的力量和趋势。我们首先关注应用对计算性能增长的需求，然后关注努力满足这些需求的技术和体系结构的趋势。我们看到随着计算机的集成度越来越高，并行处理不仅具有天生的魅力，而且开发和利用的层次在不断提高。最后，本节将看一下并行性在最高性能计算机中的作用。

5

1.1.1 计算机应用发展的趋势

应用对计算机性能不断提高的要求，对计算的各个方面来说都是一个推动。硬件能力的提高使应用的新功能成为可能，于是对计算机体系结构提出了新的更多的要求。这种循环驱使大量的设计、工程和制造技术的进步，使得微处理器的性能得以呈指数级增长。而且它更强烈地推动并行计算机体系结构的发展，因为并行计算机体系结构要满足最高要求应用的需求。随着处理器性能每年50%的增长，在10年以后，有着上百个处理器的并行计算机将会被广泛使用；在20年后，有上千个处理器的并行计算机也会普及起来。

广泛的应用需求使计算机厂商提供各种不同性价比的机器。大量的机器和客户都在低端，要求最高的应用由高端服务器来解决。这个“平台金字塔”的一个效果是对性能的要求绝大多数在高端并且是通过少数重要的应用来体现。在微处理器以前，性能是通过新颖的电路和机器组织来实现。今天，为了获得比当代微处理器更高的性能，主选就是用多个处理器，而且最高要求的应用都是写成并行程序。这样，并行计算机体系结构和并行程序设计都是要获得高性能。

对体系结构设计者和应用开发者来说，一个关键点是如何才能用并行性提高应用的性能。我们可以定义 p 个处理器的加速比为

$$\text{加速比（}p\text{ 个处理器）}=\frac{\text{性能（}p\text{ 个处理器）}}{\text{性能（1 个处理器）}} \tag{1-1}$$

对于一个固定的问题，机器的性能和时间是成倒数关系的，所以我们有下面的公式：

$$\text{加速比}_{\text{固定问题}}(p\text{ 个处理器}) = \frac{\text{时间（1 个处理器）}}{\text{时间（}p\text{ 个处理器）}} \tag{1-2}$$

1．科学和工程计算

许多领域的进展都直接依赖于计算机性能水平的不断提高，这一点在计算科学和工程领域表现得尤为突出。在计算领域，计算机被用来模拟那些要么是不可能，要么要花很大代价才能观测到的物理现象。典型的例子包括建立全球长期气候变化的模型、星系的演化、物质的原子结构、发动机的燃烧效率、交通工具表面的气流、由于碰撞造成的损害和精密电子仪器的行为。计算建模，使我们可以更深入地分析有关的设计。计算机的性能和通过模拟所能研究的问题之间通常有一个直接的关系。图 1-2 总结了联邦科技政策办公室下属的物理、数学和工程科学委员会 1993 年的研究结果。它指出了解决一些重要的科学与工程问题所需的计算机速度和存储容量。即便处理器性能会大幅度提高，在不久的将来，也需要非常大规模的并行体系结构来处理这些问题。而再往后，我们会看到新的、更大的挑战。 6

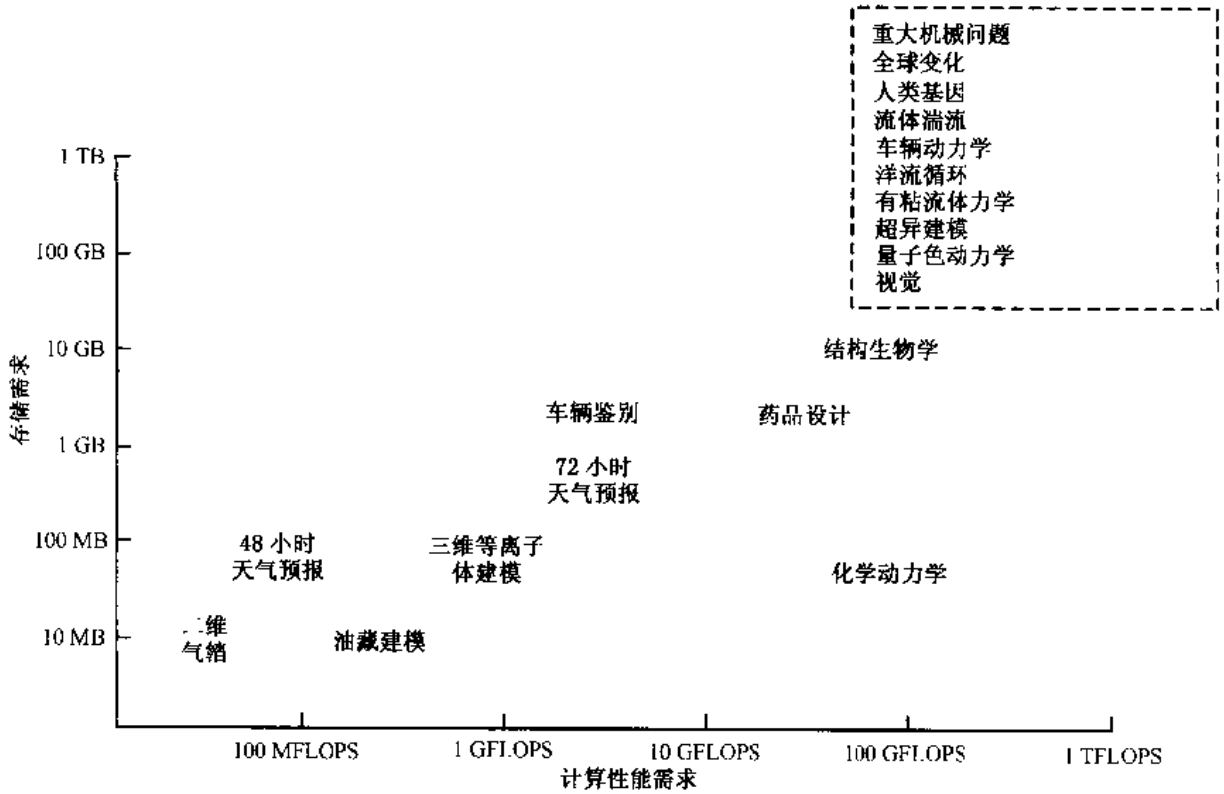


图 1-2 重大挑战性应用的需求。这是一类对计算机性能和存储容量有不同需求的重大的科学和工程问题。给定处理器性能和容量的指数增长，两个轴都和时间对应。在轴的右上端是美国高性能计算和通信（HPCC）计划指出的重大挑战性应用 7

并行体系结构已经成为科学计算的中流砥柱，包括物理、化学、材料科学、生物学、天文学、地球科学和其他科学。对物理现象建模的工具也被应用到工程应用中，包括石油（油藏模拟）、汽车（碰撞模拟、牵引分析、燃料效益）、航空（气流分析、引擎效率、结构力学、电磁学）、制药学（分子建模）和其他。在大多数这些应用中，要求计算的结果可视化，这也是并行计算的一个需求应用。

可视化成分使得传统领域，例如科学和工程更加接近娱乐业。在 1995 年，第一个计算机动画电影《玩具总动员》（*Toy Story*），是在由数百台 Sun 工作站组成的并行计算机系统上

制作的。这种应用成为可能的原因在于底层技术和体系结构已经越过了三个主要台阶：所需计算的成本已经减少到和拍电影的预算相当；单个计算机的性能已足够高；以及一定规模的并行性使得计算可以在可接受的时间内完成（在几百台计算机上计算几个月）。每个科学和工程应用在计算能力和成本方面都有一个类似的台阶，只有越过了这个台阶，用计算机来求解这样的问题才有生命力。

让我们用前述重大挑战问题中的一个例子来理解应用、体系结构和工艺技术在并行计算机中的相互影响。1995 年的一项研究（Pfeiffer et al. 1995）考察了多种并行计算机在各种应用上的有效性，这些应用中包括一个分子动力学程序包，称为 AMBER（Assisted Model Building through Energy Refinement）。AMBER 被广泛地用来模拟大的生物模型的运动，例如蛋白质和 DNA，它们是由残基（分别是氨基酸和核酸）序列组成，每个都由若干原子组成。原始代码是在 CRAY 超级向量计算机上开发的，这台计算机有定制的处理器，容量很大且昂贵的 SRAM 内存（替代高速缓存），以及可以在一个数据序列（称为向量）上完成算术运算和数据传送的机器指令。图 1-3 表示了这个程序在由 128 个 Intel 微处理器构成的并行系统（Intel Paragon）上的三种不同版本的加速比。具体的测试问题涉及到模拟一个溶于水的蛋白质。测试包括 99 个氨基酸和 3 375 个水分子，大约有 11 000 个原子。

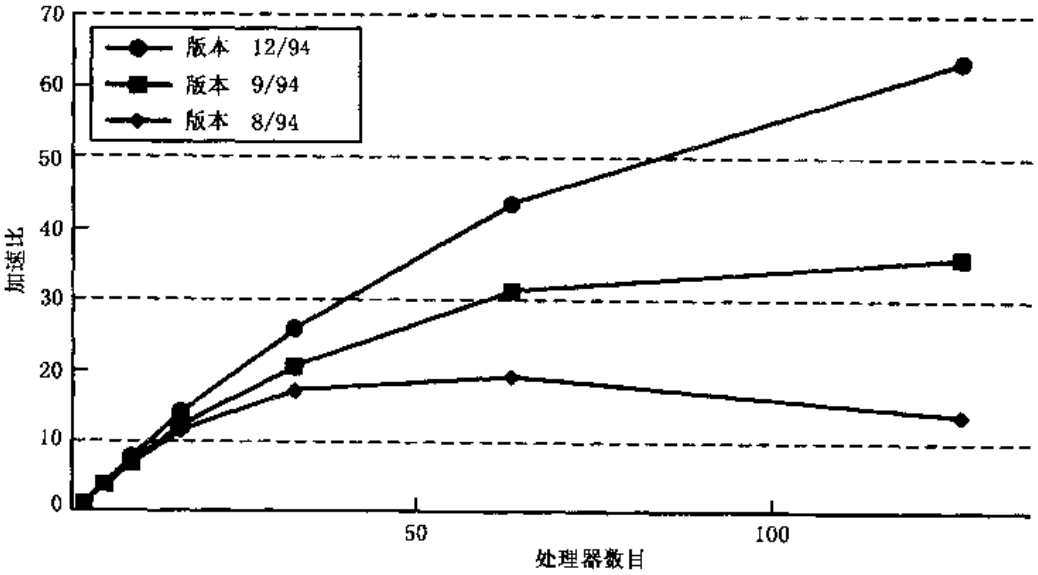


图 1-3 三种版本的并行程序的加速比。在 Intel Paragon 上执行分子动力学代码的三个版本的加速比曲线

这段代码最初的并行化（版本 8/94）对于小规模的系统配置有很好的加速比，但对大规模的系统加速比很小。稍做些努力，用第 2 章要讨论的技术对每个处理器的工作量进行平衡，应用的可扩展性就得到了大大提高（版本 9/94）。进一步的努力，将通信进行优化，我们得到了高可扩展性的程序（版本 12/94）。这 128 个处理器版本达到的性能是 406 MFLOPS；以前在 CRAY C90 向量处理器上获得的最好结果是 145 MFLOPS。同样的应用在 CRAY T3D 上（一个更高效的并行计算机），128 个处理器达到 891 MFLOPS。这种不断改进的过程在重大应用的并行化方面是相当典型的，反映了应用和体系结构之间的相互作用。应用程序的编写人员研究应用的特性，理解它对体系结构提出的要求以及如何在给定的计算机上改善性能。体系结构设计者也要研究不同的需求来理解如何设计指令集才能对给定应用的机器更有效。我

们所追求的理想是使应用程序的最终用户能够感受到这两方面努力带来的好处。

对性能不断提出新的要求是建模和模拟工作的一个自然趋势。例如，在电子 CAD 中，随着芯片上器件数目的增长，要模拟的内容显然越来越多。另外，设计复杂性的增加需要有更多的测试向量，并且由于高级功能在芯片内的组合，每个测试要花费更多的时钟周期。进而，由于生产的成本极高，需要对模拟结果有更大的信心。这种综合的效果是，设计一代新型微处理器对计算的要求不断提高，其速率甚至高于微处理器本身性能提高的速率。

2. 商业计算

商业计算在高端也要依赖并行体系结构。尽管并行性的规模不如科学计算，但其应用面要更宽。早在20世纪60年代中期，多处理器就在商业计算的高端市场上崭露头角。在这个领域，计算机系统的速度和容量直接转移到系统支持的商业规模中。一个商业企业所拥有的计算能力和它的事业规模的关系可以通过事务处理性能委员会（Transaction Processing Performance Council，TPC）的在线事务处理（OLTP）基准测试程序反映出来。这些基准测试程序能够以每分钟的事务量（tpm）表示的系统吞吐量来评估系统的性能。TPC-C 测试程序是一个融交互和批处理于一体的订购系统，包括一些实际应用中的特性，如事务队列，事务的取消和其他特性（Gray 1991）。基准测试程序包括显式的可扩展性的指标，从而使它更加实际，例如数据库的规模和用户终端的数量可以随 tpmC（在 TPC-C 上的 tpm）值的增加而增加。这样，一个更快的系统必须在一个更大的数据库上操作，并能给更多的用户提供服务才是合理的。 9

图 1-4 所示为 TPC 在 1996 年 3 月发布的一个结果，给出了一些系统的 tpmC 值。纵轴代表吞吐量，横轴代表处理器数目。这个数据反映了若干不同系统的情况，来源于不同的硬件和软件厂商。由于基准测试中所求解的问题随系统性能扩大，我们不能简单地比较时间。为此，我们用系统的吞吐量来比较，如例 1.1 所示。 10

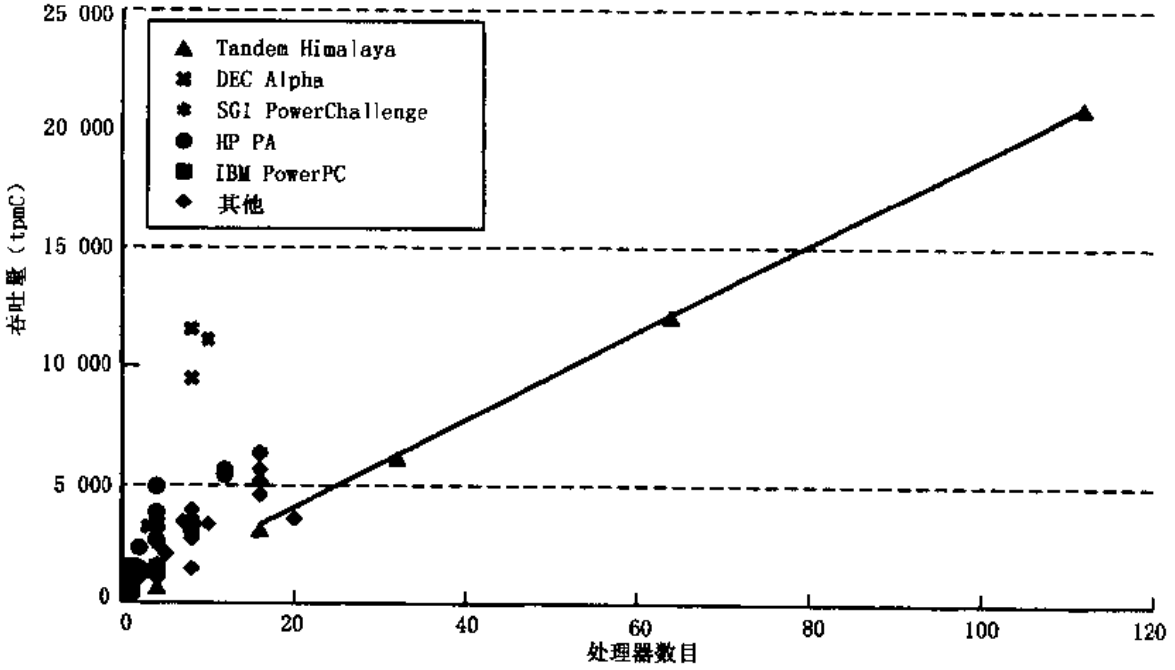


图 1-4 TPC-C 吞吐量和 TPC 上处理器的数量。1996 年 3 月 TPC 的报告表明了一系列系统的事务处理性能。图中给出了处理器的个数，并特别列出了 5 个主要的产品系列。尽管并行性高低不同，但所有主要的数据库厂家都用多处理器来作为获得高性能的方案

例 1.1 Tandem 公司的 Himalaya 和 IBM 的 PowerPC 系统的 tpmC 值如表所示。每个系统的加速比各是多少?

tpmC		
处理器数目	IBM RS6000 PowerPC	Himalaya K 10000
1	735	
4	1 438	
8	3 119	
16		3 043
32		6 067
64		12 021
112		20 918

解答：对于 IBM 系统，我们可以相对于单机系统给出加速比；对于 Tandem 的 Himalaya，我们只能以 16 个处理器的系统为基准来计算加速比。当从 1 个处理器到 4 个处理器时，IBM 的机器在并行数据库的实现上看来有很大的开销问题；然而，从 4 个到 8 个处理器时的可扩展性相当好（超线性）。Tandem 系统的可扩展性很好，但在 100 个处理器范围时加速比趋于扁平。■

$Speedup_{tpmC}$		
处理器数目	IBM RS6000 PowerPC	Himalaya K 10000
1	1	
4	1.96	
8	4.24	
16		1
32		1.99
64		3.95
112		6.87

可以从 TPC 的数据中观察出一些重要的信息。首先，并行体系结构的应用是普遍的，基本上所有提供数据库硬件或软件的厂商都用多处理器系统来作为得到高性能的手段。其次，不仅仅是大规模并行性是重要的，而且从有几十个处理器的适度规模的多处理器和即使只有两个或 4 个处理器的小规模多处理器，也都很重要。最后，即使是对于一类特定的系统，在一个特定的时间点所作的一套良好文档的测量也没法完全反映技术的情况。技术发展很快，系统需要时间来开发和布署，并且实际的系统还有一个生命周期的问题。这样，在任何时刻，来自厂商最好的系统处于它们生命期的不同点上。例如，在 1996 年 3 月的 TPC 报告中，DEC Alpha 和 IBM PowerPC 系统要比 Tandem Himalaya 系统新得多。因此，我们不可以轻率地做出结论，例如说 Tandem 的 Himalaya 系统由于其设计中的可扩展性问题，导致它的效率从根本上要低些。然而，我们可以说，即使是大规模的系统也要不断地跟踪工艺技术的发展来维持它的优势。

11

向并行程序设计的过渡，包括新算法的设计或对现有算法中通信和同步需求的关注，已经在计算的高性能端普遍展开。这种过渡在商业软件中也正在进行。典型地，工程和商业应用把目标定在适度规模的多处理器，这些多处理器统治着服务器市场。在商业界，所有主要的数据库厂商都在它们的高端产品中支持并行计算机。一些主要的数据库厂商还支持“什么

也不共享”的大规模并行机和通过高速网络连接的工作站（通常叫做机群）；另外，多处理器也在多道程序系统中用来提高吞吐量，甚至桌面机的应用也表现出多个并发进程的操作模式，例如多个活动窗口和守护程序。经常，一个用户会让他的任务在局域网的多个计算机上运行。所有的这些趋势显示了一种实实在在的应用需求，即需要不同规模的并行计算机体系结构。

1.1.2 微电子技术趋势

为了满足不断提高的性能要求，我们可以通过对底层技术和体系结构进展的理解来进一步明确并行的重要性。趋势表明通过“单机速度的提高来提供高性能”是困难的，而通过并行体系结构是可能的。另外，这些考察表明在并行计算机体系结构中采用的微电子技术和那些“串行”计算机中的很类似，例如预算如何在计算功能单元，开发局部性的高速缓存，以及在提供通信带宽的连线中分配芯片面积的资源。

主要的技术进步在于基本的 VLSI 的线宽稳步减小。这使得晶体管、门和电路越来越快，越来越小，所以在同样的面积内能容纳更多的单元。另外，可用的晶片尺寸在增长，所以有更多的空间可用。直觉上说，时钟频率的提高和线宽的减小是成正比的，而晶体管的数目和线宽的减小是平方增长的，而增加整个晶片的面积，晶体管数目还会更多。这样，从长远的观点看，通过同时用许多晶体管（即并行性）比通过提高单处理器时钟频率更能提高处理器的性能。

12

这种直觉可以通过对商用处理器的比较得出。图 1-5 显示了几种重要的微处理器家族的时钟频率和晶体管数量的增长。用于微处理器的时钟频率以每年大约 30% 的速度增长，晶体管的数量以每年大约 40% 的规模增长。这样，如果我们看一个芯片上的原始计算能力（每秒晶体管开关的总数），在过去的 20 年里，晶体管数量增长的贡献要比时钟频率提高的贡献高一个数量级[⊖]。在标准基准测试程序上微处理器性能的提高比时钟频率的提高要快。

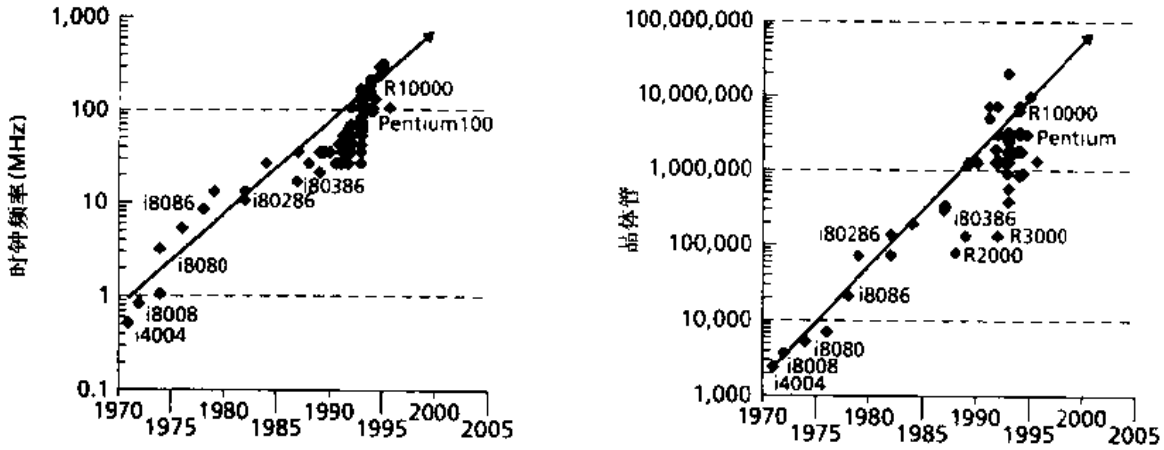


图 1-5 微处理器中逻辑单元的增长和时钟频率的提高。光刻技术、工艺技术、电路设计和数据通路设计的改进在逻辑单元的密度和时钟频率已经产生一个持续的提高

⊖ 为什么晶体管数量没有以时钟频率的平方关系增加，其原因是多方面的。其一是处理器大量的面积被连线占用，用来在片内分布控制、数据或者时钟信号（即芯片内的通信）。我们会看到通信问题在并行计算机系统结构的每个层次都会出现。

最常用的测量工作站性能的基准测试程序是 SPEC 程序集，它包含若干实际的整数程序和浮点运算程序（SPEC 1995）。SPEC 上整数测试程序的性能以每年大约 55% 的速度增长，而浮点性能以每年 75% 的速度增长。LINPACK 基准测试程序（Dongarra 1994）被广泛用来评价系统在数值应用上的性能。LINPACK 的浮点性能以每年超过 80% 的速度增长。因此，处理器越来越快的原因在很大程度上是由于更有效地利用了越来越多的计算资源。

对这些技术趋势的简单分析表明，作为系统基本单元的芯片还会提供更大的计算能力——在 2000 年将会有近 1 亿个晶体管。这种增长将会使我们能在芯片内放置更多的计算机系统的部件，包括内存和 I/O 支持，或者能在一个片上放多个处理器（Gwennap 1994a）。前者导致小的、更易封装的并行体系结构的构造模块，后者使单个芯片上有并行的体系结构（Gwennap 1994b）。这两种情形在商业上都有可能出现，例如我们看到片上系统（SOC）在嵌入式系统、便携式系统和低端 PC 产品中首先被使用。而在一个芯片上用多个处理器的实践正在数字信号处理领域普及起来（Feigel 1994）。

13

性能和容量之间的变化在内存技术中更为深刻。从 1980～1995 年，DRAM 芯片的容量增长了 1 000 倍，每 3 年翻 4 倍，而存储周期只缩短了一或两倍。在一台有 1 亿个晶体管的处理器中，我们期望有千兆位容量的 DRAM 芯片，但是处理器周期和内存周期的差距就更大了。这样，处理器对内存的带宽（每存储周期能给出的字节数）需求也就更大了。

内存操作的时延是由访问时间决定的，它小于存储周期，但是每个内存操作对应的处理器周期数还是很大并且在增加。为了降低处理器看见的存储访问平均时延，增加存储带宽，我们必须更有效地利用在处理器和 DRAM 存储器之间的存储层次结构。现代处理器基本上都有一级和二级高速缓存，而且系统的设计提供了加入外部高速缓存的可能。当我们谈到多处理器设计的时候，一个基本问题就是如何在多个处理器和多个存储模块之间安排一组高速缓存。例如，并行体系结构的一个直接的好处就是每级存储层次的总容量可以随着处理器数目的增加而增加，而这种容量的增加不会增加访问时间。

把这些结论扩展到磁盘，我们看到一个类似的情形。并行磁盘存储系统，例如 RAID，已经成为许多系统的标准配置。将大容量多级缓存用于文件或磁盘块访问的缓存在各种系统中也是屡见不鲜。

1.1.3 体系结构趋势

集成电路技术上的进步决定了什么是可能的，什么是不可能的；体系结构把技术的潜力转变为性能和容量。从根本上讲，利用更多的资源（例如，更多的晶体管）提高性能的两个途径是并行性和局部性。但是，这两个途径在对资源的占有上是竞争的关系。当多个操作并行执行时，程序执行所用的周期数减少了。但是，这需要资源来支持每个并发的活动。当对数据的引用越接近处理器，访问深层次存储的时延就可以避免，但也需要资源提供这种局部性。通常，最好的性能是通过某种中间路线来达到的；既开发一定的并行性，也利用一定的局部性。确实，我们可以从本书中看到并行性和局部性是如何在系统的各个层次（从一个芯片到大规模并行机器）相互作用的。在当前的微处理器上，晶片面积基本上是在高速缓存、处理部件和片外互连方面大致等量分配的。由于成本和性能的权衡不一样，大规模系统可能会有些不同，但基本问题是一样的。

14

1．微处理器设计趋势

研究一下微处理器的发展趋势有助于我们理解并行机，也能够使我们看到在常规计算机系统结构中开发并行性的根本性作用，还能使我们感到当前系统结构的趋势将导致多处理器的设计。（本书中关于处理器设计技术的讨论只是泛泛的，因为我们假定许多读者已经从其他传统的体系结构著作［Hennessy and Patterson 1996］中了解它们了。不过我们这里的讨论角度是独特的，有助于唤起读者的回忆。）

计算机体系结构的发展按照逻辑实现的技术可以分为4个阶段：电子管、晶体管、集成电路、大规模集成电路。本章的所有内容都属于第4阶段。很明显，在这个阶段有巨大的体系结构进展，那么接下来会是什么样呢？最突出的表现是由图1-6展示的并行性的发展。

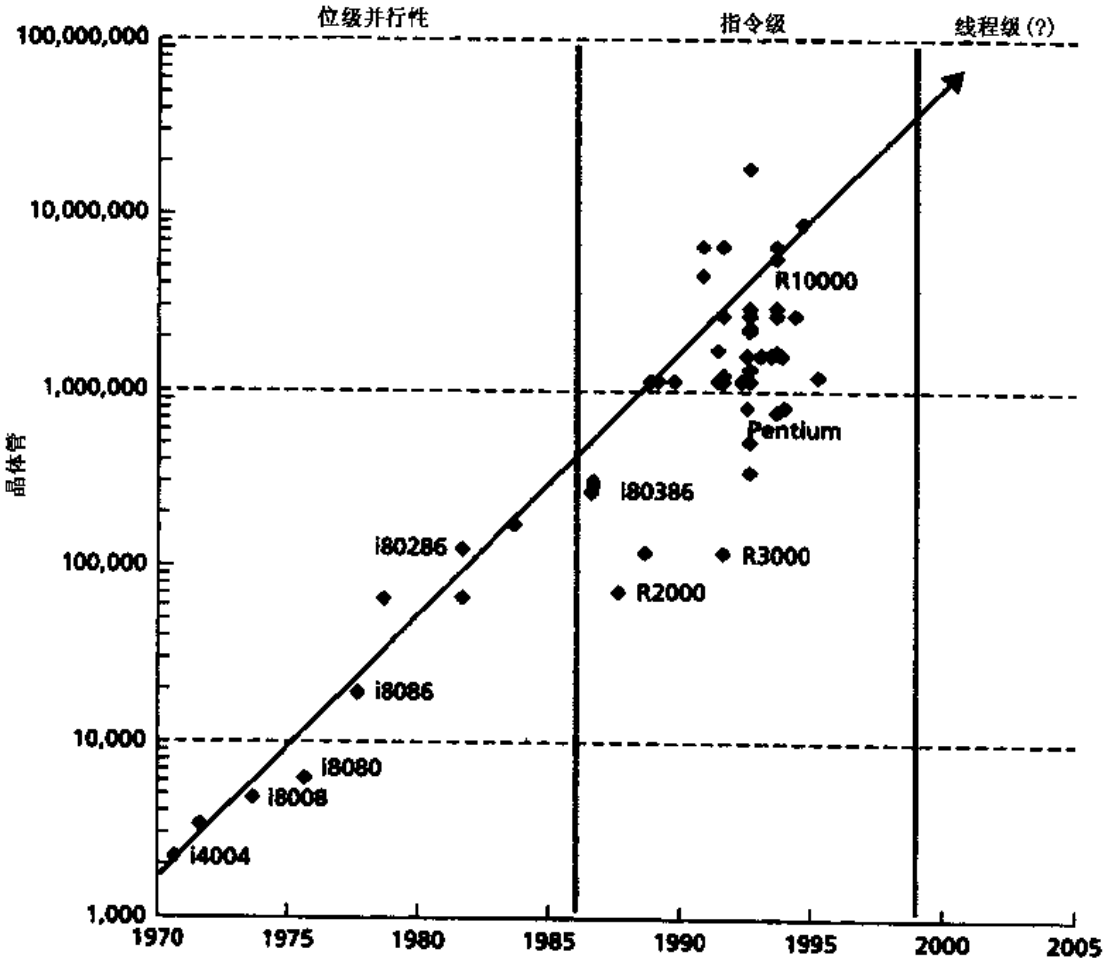


图1-6 25年来处理器上晶体管数目的变化。这个变化基本上遵循摩尔定律。定律说晶体管的数目每两年翻一番。通过过去来预测未来，我们可以推出未来10年会出现有5 000万到1亿个晶体管的处理器。图中也表现了在第4代技术（VLSI）中计算机体系结构不同设计的特色，反映出并行性层次的提高

一直到1986年，位级并行的进展都是居于支配地位的，4位的微处理器被8位微处理器取代，8位的又被16位的取代等。数据通路的加宽减少了32位操作所用的周期数。在20世纪80年代中期，当出现了32位的计算机时，这个趋势就减缓了，64位的操作只是在10年后部分地被采用。对更宽的数据的需求不是由性能引起的，而主要是为了改进浮点数的表示或更大的地址空间。随着对地址空间的需求以每年不到一位的速度增长，看来128位的操作

不会在最近出现。最早的微处理器从最容易的并行性形式（即每个操作中的位并行）中获益。图 1-1 所示的微处理器成长曲线中的拐点标志着 1986 年 32 位字操作，以及高速缓存广泛应用时期的到来。

20 世纪 80 年代中期到 90 年代中期是指令级并行的天下，多条指令的不同部分同时进行。全字操作意味着一条指令处理过程中的每个基本步骤（译码、整数运算、地址计算）分别都能在一个周期内完成；而在缓存中，大多数情况下取指和数据访问也可以在一个周期内完成。RISC 的研究表明了这一点，采用精心设计的指令集，可以让指令处理的每一步按流水线方式进行，从而几乎在每个周期都可以完成一条指令。这样，在指令执行之间的少量并行性得到了开发。虽然以流水线方式执行指令的技术不是新的，但它从来没有像现在这样和基本的集成电路工艺技术匹配得那么好。另外，编译技术的进步也使指令流水线更有效。

15

在 20 世纪 80 年代中期，基于微处理器的计算机由少量部件芯片构成，包括整数处理单元、浮点处理单元、缓存控制器和 SRAM。随着芯片容量的增加，这些部件集成在一个芯片上，就降低了它们之间的通信开销。这样，一个芯片就包含了用于整数算术、存储操作、转移操作和浮点操作的独立硬件。除了能够使每条指令流水执行外，一种令人感兴趣的想法是同时读多个指令，并把它们发送到不同的功能单元，这种形式又叫做超标量执行。它提供了一种利用不断增加的芯片资源的自然途径。功能单元越多，每次取的指令就越多，在每个周期就可以向功能单元发出更多的指令。

16

然而，在处理器中提高指令并行性的前提是处理器可以及时得到数据和指令，否则它就会闲下来。为了满足增加指令和数据带宽的需求，就要在芯片上放越来越大的缓存，这是要消耗晶体管数目的。一旦处理器和高速缓存在一个芯片上了，就有可能设计很宽的数据通路，从而满足多指令的带宽需求和每个周期的数据访问。尽管如此，随着每个周期发送指令数的增加，每次控制转移和缓存扑空的影响就会越来越明显。一个控制转移可能会引起处理器等待其流水线排空的时间，或称为时延，直到某条特殊的指令出现在流水线的末端，才能决定下面该做什么。类似地，当缓存扑空时从内存中读取数据也会使处理器暂停。

20 世纪 90 年代的处理器设计用到了许多复杂的指令处理机制，来减小超标量处理器中时延造成的性能降低。通过在控制转移前猜测控制流的方向，人们用精巧的转移预测技术来降低流水线的时延。更大、更复杂的缓存用来避免缓存扑空的时延。指令的动态调度允许指令变序发出和变序执行，因此如果一个指令遇到扑空，另一个指令可在它之前处理，只要这两个指令不依赖于指令的结果。在处理器内维护一个较大的等待分发的指令窗口，只要某条指令产生了一个新结果，多条等待着的指令就可能同时发向功能单元。这些复杂的技术可以使处理器容忍缓存扑空的时延和流水指令之间的依赖。然而，每个方法都需要大量片上硬件资源的支持，而且有很高的设计代价。

随着处理器集成度的增加，自然的问题就是在一个单线程控制流内的指令级并行能走多远？在什么情况下，问题的重点会转移到支持更高级别的并行，例如多进程或进程内控制的多线程，即线程级并行？通过计算机设计的模拟（Chang et al. 1991；Horst，Harris，and Jardine 1990；Lee、Kwok，and Briggs 1991；Melvin and Patt 1991），或者通过对程序固有性质的分析（Butler et al. 1991；Jouppi and Wall 1989；Johnson 1991；Smith，Johnson，and Horowitz 1989；Wall 1991），一些研究试图给出对这个问题第一部分的回答。在 Johnson 的书中（1991），对这个论题进行了完整的描述。机器设计的模拟通常显示 2 路超标量，即每周期发送两条指令，是非常有

17

效的，而4路超标量也会有明显的改善，但是再多了（例如8路超标量）就很少能提高性能。由于平均约每五条指令就有一次控制转移，设计的复杂性会急剧增长。

为了对潜在的最高加速比有一个估计，我们可以假想每周期发送多条指令，用一台理想的计算机来模拟指令的执行。在该计算机上，取指的带宽无限，程序可用的功能部件不受限制，并能完美地预测转移（后者较容易做到，因为程序执行的踪迹还记录了每个分支转移情况）。这些宽泛的假设使得每条指令都不会由于功能部件忙或处理器指令预测范围的限制而被延迟，还可以通过寄存器重命名技术来做到每条指令不被延迟发射。通过这种方法，指令不会因为更新了逻辑上前导的指令所使用的单元而被延迟。对寄存器或存储器单元的每次更新都被认为是引入了一个新的“名字”，在执行轨迹后续对该值的引用会访问这个新的名字。这样，程序执行的次序就只受到基本的数据相关性的约束，每条指令一旦其操作数可用就可以执行。图1-7总结了Johnson的结果。左面的柱状图表示没有指令被发射，接着是一条指令、两条指令等。Johnson的理想计算机保留了实际功能单元的时延，包括零发射周期代表的缓存扑空（其他的研究忽略了缓存效应或流水线时延，因此会得到更乐观的估计）。我们看到，即使有无限的计算机资源，完美的转移预测和重命名技术，在一个周期内发射指令的条数在90%情况下要少于4。基于此分布，我们可以得出右边的加速比图。最近的一项工作（Lam and Wilson 1992；Sohi，Breach，and Vijaykumar 1995）表明，要想得到更大量的并行，需要同时考虑多控制线索。虽然在指令级并行可能还会出现新的突破，但随着芯片容量的增加，向下一层次并行（并发多线程）的转移越来越有吸引力。

18

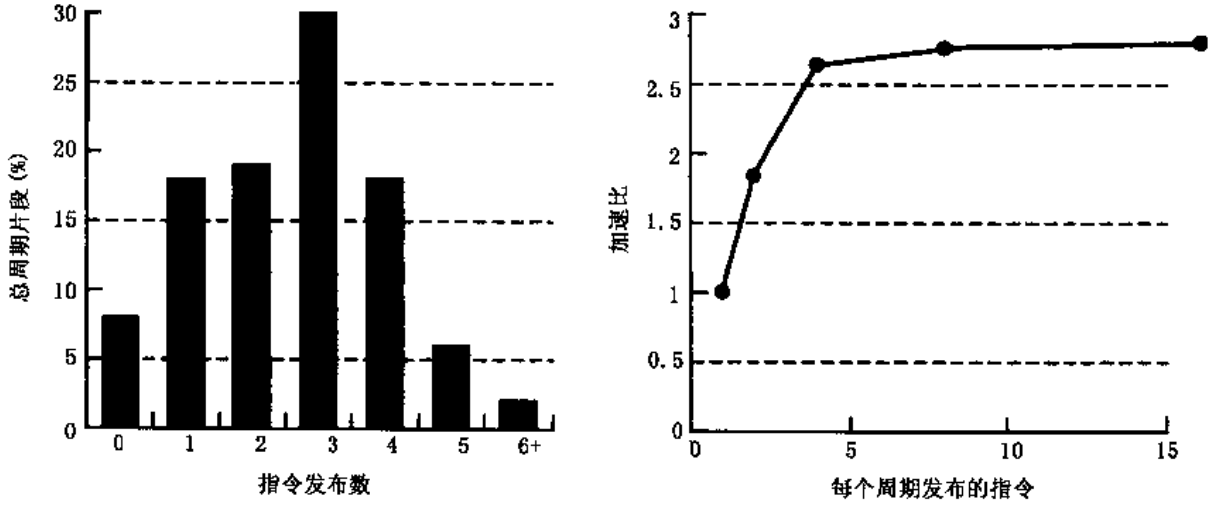


图1-7 理想超标量计算机内的指令级并行的分布和估计的加速比。图中表示了在理想超标量执行的情况下，包括无限处理资源和完美的转移预测，可用的指令级并行的分布和最大可能的加速比。数据是由Johnson（1991）的基准测试程序给出

2. 系统设计的趋势

在计算机系统中，面向线程级和进程级来开发并行性的趋势已有相当一段时间了。基于共享内存的计算机在20世纪80年代32位处理器出现时已经很流行了（Bell 1985）。图1-8显示出商用机器中处理器数目随时间的变化。这类基于总线的共享内存多处理器有效地提高了系统的性能。自从20世纪80年代后期，几乎每一种商用处理器都支持多处理器配置，在第5章我们会专门讨论这类系统。多处理器统治着服务器和企业计算的市场并已经向桌面系统进军。

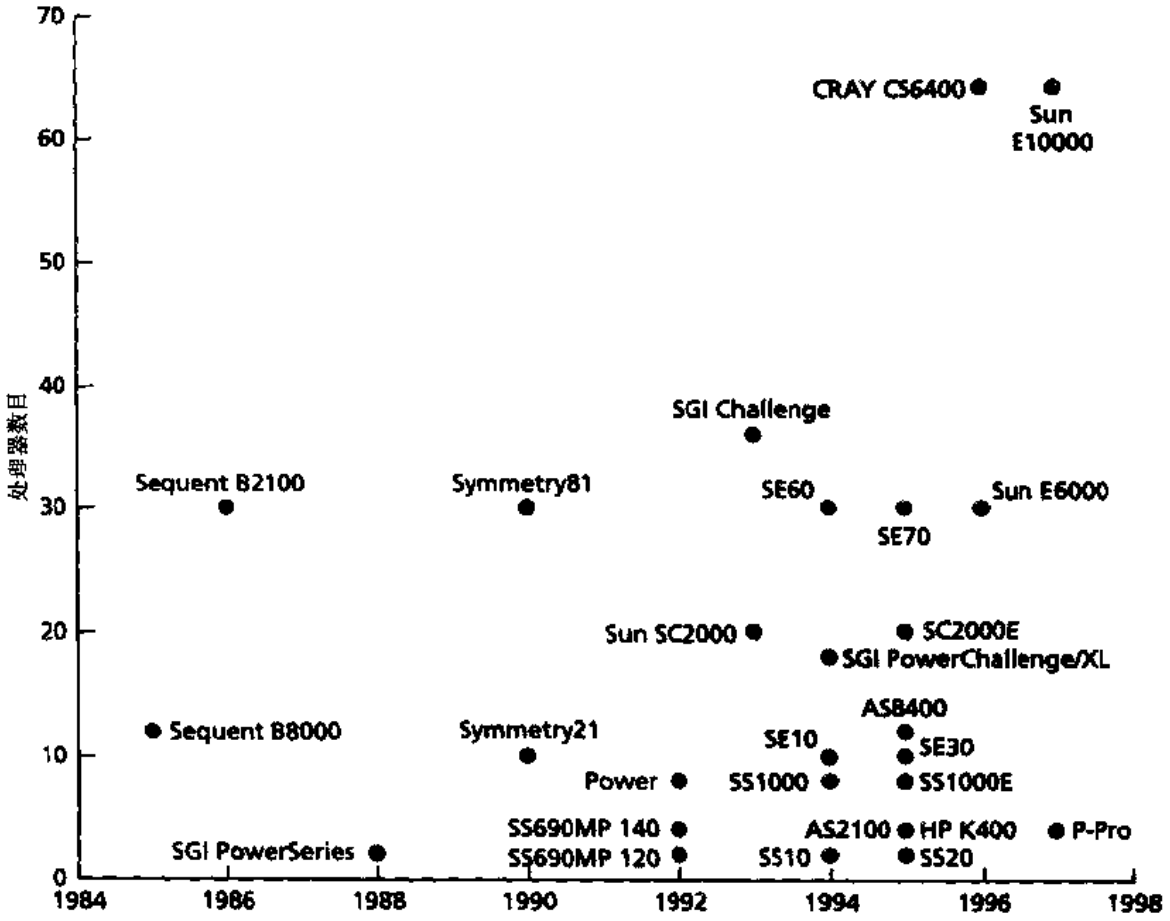


图 1-8 在商用基于共享内存系统中的处理器的数目。经过最开始的 10 到 20 路基于共享内存的 CISC 处理器，Sun、HP、DEC、SGI、IBM 和 CRI 公司就开始提供一定规模的基于 RISC 的 SMP 了，还有没在此显示的其他商用厂商，包括 NCR/ATT、Tandem 和 Pyramid

最早的多处理器系统是一些小公司研制的，如 Synapse（Nestle and Inselberg 1985），Encore（Schanin 1986），Flex（Matelan 1985），Sequent（Rodgers 1985）和 Myrias（Savage 1985）。其动因是希望在小型机市场占有一席之地。它们将 10 到 20 个微处理器组织在一起，在分时应用负载上能给出有竞争力的吞吐量。随着 Intel i80386 的出现，并作为处理器的基础，这些系统在商业上获得了成功，尤其是在事务处理方面。然而，高性能的 RISC 处理器在 20 世纪 80 年代后期阻止了 CISC 多处理器的进一步发展，并且完全占领了小型机市场。接着，一些大公司开始生产 RISC 微处理器系统，特别是作为服务器和大型机的代用品。这些设计使得带宽成为关键因素。在大多数的多处理器设计中，所有处理器都接到一个公共总线上。因为总线是固定带宽的，所以随着处理器的加快，总线只能支持少数处理器。20 世纪 90 年代初，共享内存总线技术有了巨大的进步，包括更快的电信号、更宽的数据通路、协议的流水执行和多通路等。这些都为带宽的提高做出了贡献，如图 1-9 所示。带宽的增加允许多处理器系统做到 10 到 20 个处理器，或者更多一些（Alexander et al. 1994；Cekleov et al. 1993；Fenwick et al. 1995；Frank，Burkhardt，and Rothnie 1993；Galles and Williams 1993；Godiwala and Maskas 1995）。

19

在图 1-8 中，20 世纪 90 年代中期的情况是很有趣的。不仅仅是基于总线的共享内存多处理器在工业界普遍存在，而且可以看见多种不同的规模。桌面系统和小型服务器通常支持

2~4个处理器，较大的服务器支持数十个，大型商业系统支持上百个。这种趋势越来越明显。1994年 Intel 定义了一个基于它的 Pentium 微处理器设计多处理器 PC 系统的标准（Slater 1994）。接下来的 Pentium Pro 微处理器允许 4 个处理器配置在一起，不需要任何额外的连接逻辑，总线驱动、仲裁电路等都在微处理器内部。这些发展为小规模多处理器的流行铺平了道路。另外，我们注意到在工业上的一个转变是，多处理器不仅由硬件供应商推动，软件供应商，特别是数据库公司，也推动多处理器的发展。将这种潮流和技术发展的趋势结合起来，现在的问题是片载多处理器系统何时会普及起来，而不是会不会普及的问题。 20

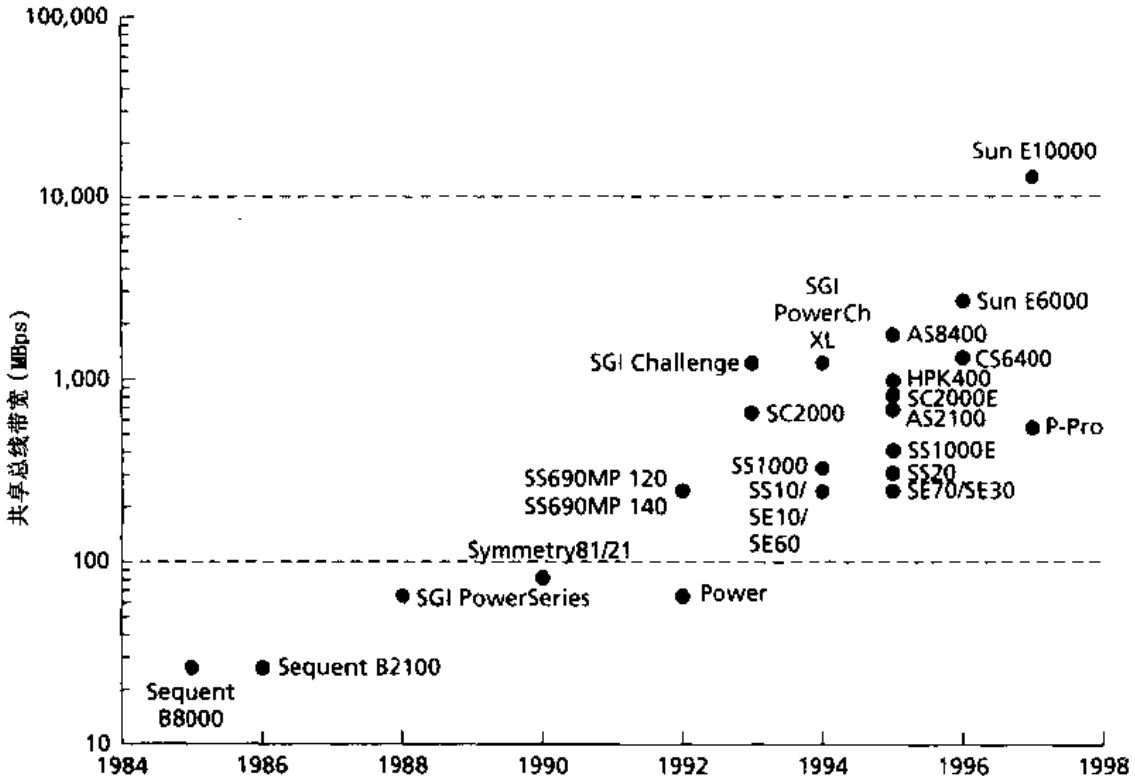


图 1-9 商用多处理器的共享内存的带宽。经过几年的缓慢增长，在 1991 年出现了一个内存总线设计的新时期，能够支持相当数目的高速微处理器

1.1.4 超级计算机

我们已经看到了在通用市场上驱动并行体系结构发展的动力。第二个具有影响力的动力来自超级计算领域，那里要追求最大的性能。尽管商业和信息处理的应用在增长，成为高端计算的一个重要部分，科学计算在历史上就是展示体系结构创新的领地。在 20 世纪 60 年代中期，它首先使用了流水线技术和动态指令调度，这些技术在现在都是通用技术。在 20 世纪 70 年代中期，在超级计算中居统治地位的是向量处理器，它的基本操作可以作用在一个数据序列上，而不是单个数据。向量操作使得在单个线程控制中能获得更大的并行性。另外，这些向量计算机的实现要用到高速、高价、高能耗的电路技术。 21

非稀疏的线性代数是 LINPACK 基准测试程序和科学计算的一个重要部分。尽管这个基准测试程序仅评估了科学计算一个很小的方面，它是这么多年来为数不多的在各种计算机上都可用的基准测试程序之一。图 1-10 显示了 LINPACK 在 CRAY 上的性能（August et al. 1989；

22 Russel 1978)，并和最快的基于微处理器的工作站和服务器进行了比较。对于每个系统，都给出了两个数据点。较低的针对 100×100 矩阵，高的针对 1 000×1 000 矩阵。在向量处理器中，单个处理器的性能提高是通过在一定程度上缩短周期时间，而主要是通过提高内存带宽来达到的。在微处理器系统中，我们看到的效果是提高时钟频率、片载流水线浮点单元、增加片载高速缓存的规模、增加片外二级高速缓存的规模，以及开发指令级并行性的综合效果。它们的差距在迅速减小。

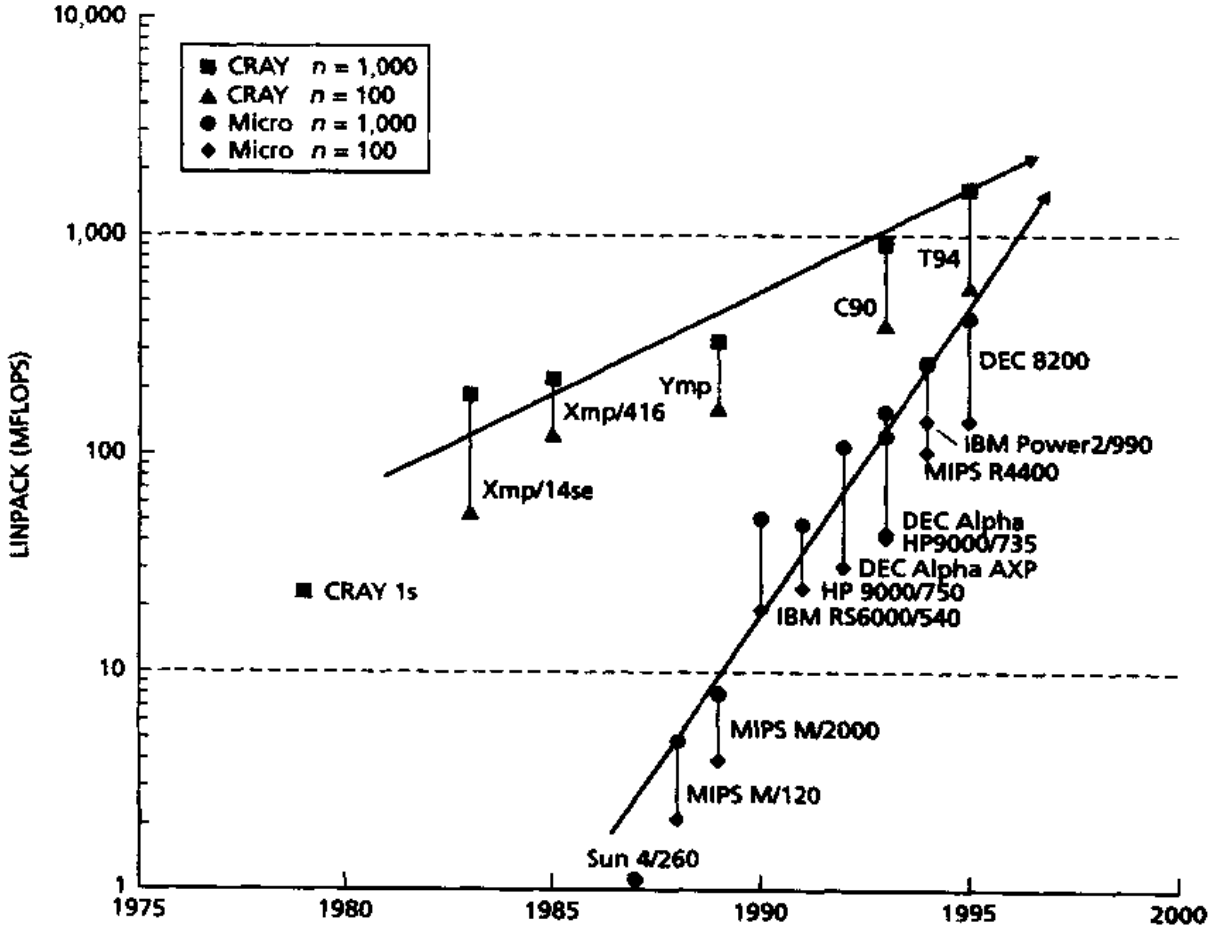


图 1-10 超级计算机的单处理器性能和基于微处理器的系统在 LINPACK 基准测试程序上的性能。图示是最快的向量计算机 CRAY 和最快的工作站在 100×100 和 1000×1000 矩阵上的性能比较

向量计算机和微处理器设计都可以采用多处理器体系结构，但规模是相当不同的。CRAY Xmp 最先采用了两个，然后是 4 个处理器，Ymp 用了 8 个，C90 用了 16 个，T94 为 32 个处理器。而基于微处理器的超级计算机最初就有约 100 个处理器，1990 年增长到 1000 个。这种含有大量处理器的系统被称为*大规模并行处理器*（MPP），紧紧跟踪微处理器的发展，通常只是比最先进的基于微处理器的工作站和个人计算机落后一两年。如图 1-11 所示，在 LINPACK 测试程序中占主要地位的是由大量较慢一些的微处理器构成的系统（注意在图 1-10 中用的是 MFLOPS，这里是 GFLOPS）。在比较完整的应用程序上，MPP 相比于传统向量计算机性能的优势不是很明显（Bailey et al.1994），这和程序语言、编译器和算法的不成熟有关；不过朝着 MPP 发展的趋势是有深刻影响的。当 CRAY 研究所于 1993 年公布了基于 DEC Alpha 微处理器的 T3D 时，这种趋势的重要性就更加明显了。

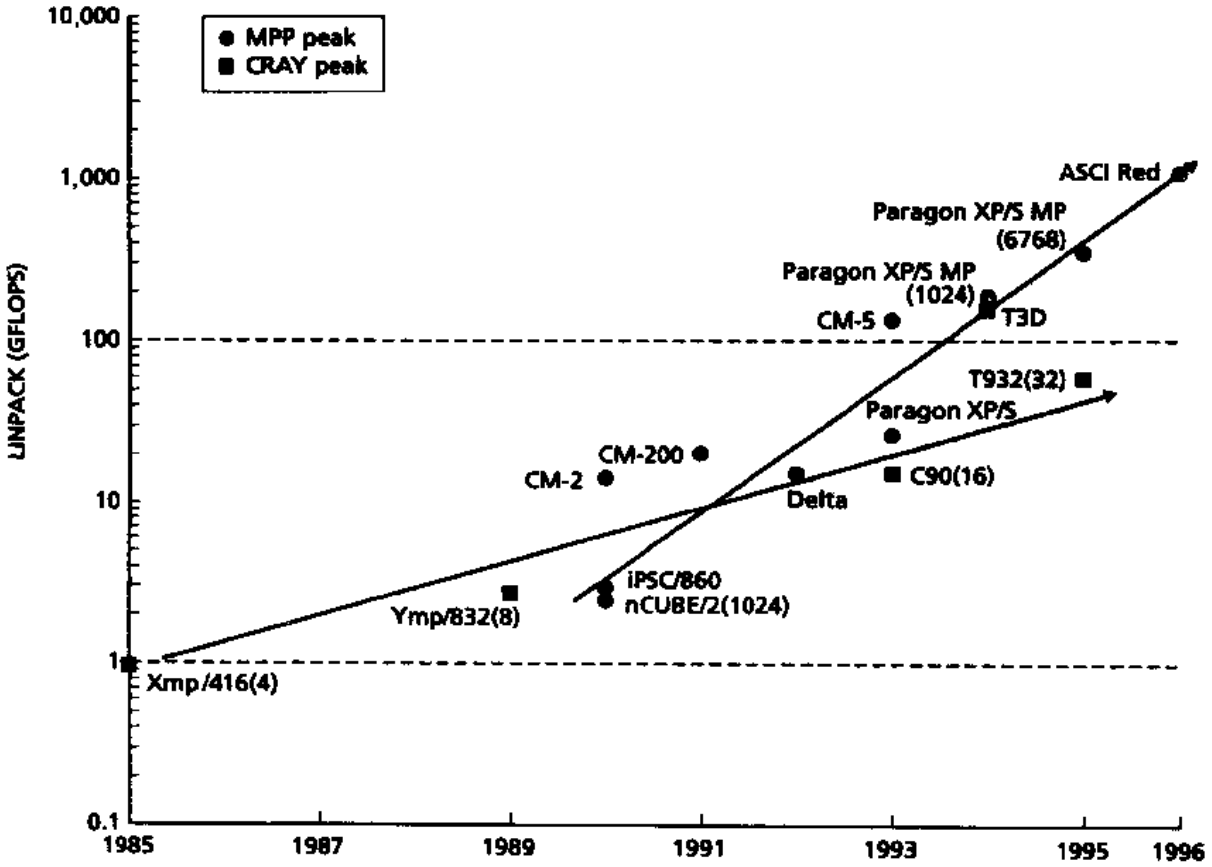


图 1-11 在超级计算机和 MPP 上 LINPACK 基准测试程序的性能峰值。图示为 CRAY 多处理器并行计算机和最快的 MPP 系统处理求解线形方程的峰值性能，用 GFLOPS 表示，注意图1-10 是通过 MFLOPS 表示

最近，LINPACK 基准测试程序被用来对世界上最快的计算机进行排序。图 1-12 显示了前 500 个系统的情况，包括多处理器并行向量处理器（PVP），MPP 和基于总线的共享内存多处理器（SMP）的数量。后两种都是基于微处理器的，趋势很明显。

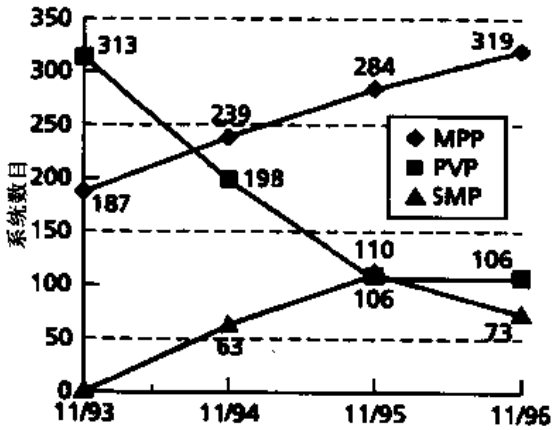


图 1-12 全球用于 500 台最快的计算机系统类型，并行向量处理器（PVP）已让位于基于微处理器的海量并行处理器和基于总线的对称共享存储多处理器

1.1.5 小结

通过从各个方面的考察，包括经济、技术、体系结构和应用需求，我们看到并行体系结

构日益重要。人们对性能强烈的追求，使得在计算机设计的不同层次和不同点上都在开发并行性。指令级并行性在现代高性能的处理器中开发。从根本上讲，在桌面系统之外都是多处理器，包括服务器、大型机和超级计算机。性能曲线的高端是海量并行处理器。在大规模并行系统的应用面正在扩大。本书的重点在于多处理器级别的并行性。我们将研究那些体现在不同规模并行机中的设计原理，从而来理解各种并行体系结构的谱系。

23 ~ 24

人们讨论并行计算机通常都是从处理器角度出发的，但如果你从内存的观点看，可能也会得到同样的结论。考虑要设计一个支持海量数据（即大规模问题的数据集）的简单的内存系统。计算机体系结构的一个规律是快速内存容量较小，容量大的内存慢。这是由许多因素造成的，包括地址解码时间、加长位线的延迟、密集存储单元的驱动较小，选择电路延迟等。结果就是内存系统应该是一个分层结构，层数越高的容量越大，速度越慢：平均来讲，如果局部性很好，层次存储系统就会很快。另一种办法是我们可以绕过物理规律的限制，用不同的处理器来读取彼此之间无关的小内存。当然，物理规律不是那么容易被躲过的。当处理器要读取非局部数据时，我们就要付出称为通信的代价，而当我们要协调许多处理器的动作时，我们也要付出同步操作的代价。

1.2 并行体系结构的融合

从历史上看来，并行机在几个阵营中同时发展。所以，大多数教科书关于这段历史，都并列地介绍。但是，仔细研究并行体系结构的发展，可以清楚地看到有关设计都被相似的技术发展和应用需求所影响。所以，在这个领域会发生体系结构的融合就不足为怪了。在这一节中，将介绍并行计算机体系结构的框架，以使读者对整个结构以及对这种融合的必然性有大概地认识。我们将从传统的阵营开始，简要地回顾并行机的发展历史，最后看它们是如何融为一体的。

1.2.1 通信体系结构

如果把并行计算机定义为“一组通过通信和协作来快速解决大型问题的处理单元”（Almasi and Gottlieb 1989），我们可以自然地把并行体系结构看成是传统计算机体系结构在通信和处理单元协作之上的扩展。从本质上来说，并行体系结构用通信结构扩展了以前的计算机体系机构的概念。计算机体系结构有两个不同侧面，其中之一就是定义关键的接口抽象，特别是软硬件边界和用户系统边界。并行体系结构定义了在边界上的操作，以及被操作的数据类型。另一方面就是用经济高效的方法来实现这些抽象。一个通信体系结构也包括这两方面。它定义了基本的通信和同步操作，并且定义了组织结构来实现这些操作。

25

图 1-13 表示了并行机中通信的框架。最上面一层是编程模型，这是程序员在编程时所见到的机器模型。每一种编程模型都指明了程序的各个部分如何交换信息和可以用来协调它的活动的同步操作。应用程序使用某种编程模型。最简单的编程模型是由大量独立的顺序程序组成的多道程序设计模型；在程序层不产生任何通信和协作。但有意思的是真正的并行编程模型，比如共享地址空间、消息传递和数据并行编程。我们可以直观地描述这些模型如下：

- 共享地址编程就像用公告牌发布消息一样，你可以将要他人知道的信息放在某个大家都知道的位置上。对每一个活动都可以通过加某某在做某某任务的注释来协调。
- 消息传递就像电话或信件一样，从发送者传到接收者。每一个消息的发送和接收都

有定义好的事件，这些事件是协调活动的基础。但是，没有可以公共访问的内存。

- 数据并行处理是一种更严格的协作方式。多个进程同时对一个数据集的不同元素进行操作，然后在全局交换信息。因为编程模型只是定义了全局的并行处理效果，这种全局数据的重组可以通过访问共享内存或消息传递来实现。

在本书的后面将更准确地定义这些编程模型；在这里，理解抽象的层次更加重要。

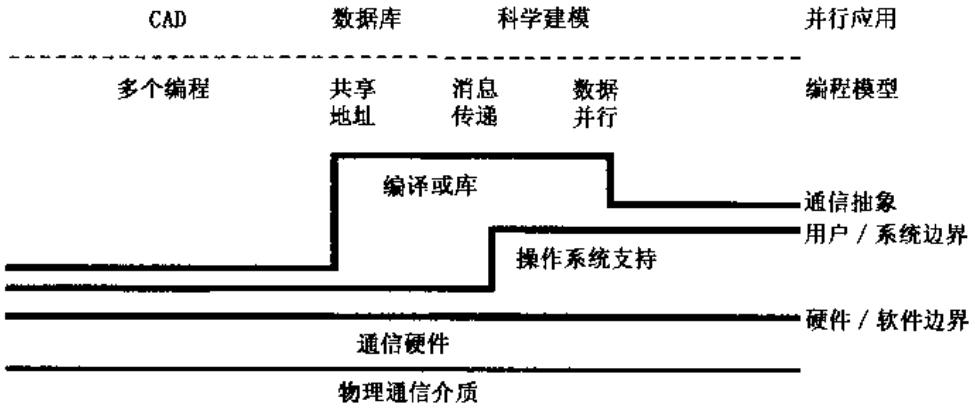


图 1-13 并行计算机体系结构的抽象层次。抽象的关键层在应用程序和实际硬件之间，应用根据编程模型编写，说明程序的各部分如何共享信息协调动作。提供通信和同步的特殊操作构成通信抽象，它是用户程序和系统实现之间的边界。这个抽象是通过编译程序或库程序实现的，使用硬件或操作系统提供的原语。操作系统则使用硬件原语。通信硬件的组成结构在连接机器的连线上有效地提供这些操作

编程模型通过用户层的系统通信原语实现，称之为通信抽象。一般说来，一个编程模型都是嵌入在一个并行语言或者编程环境中，所以要有映射将语言结构映射到特殊的系统原语上。这些用户层的原语可能直接由硬件操作系统或由针对具体机器的用户软件，将通信抽象映射到实际的硬件原语来实现。图 1-13 中线间的距离表示了层与层之间的映射可能简单，也可能非常复杂。比如，一台机器上的所有处理器用 load 和 store 指令来访问同一内存单元，这就可以实现访问共享内存。但是，消息传递有可能就需要包括库或系统调用来将消息写到缓冲区中或读取它。

通信体系结构定义了可供用户软件使用的通信操作、操作的格式和操作的数据类型，这非常像普通处理器的指令体系结构。我们注意到，即使在传统的指令集中，某些操作也可能是由软硬件共同来实现的，比如 load 指令需要操作系统的支持来处理缺页情况。通信体系结构使传统机器在支持通信方面也有所扩展。 26

长期以来人们一直都在争论，并行体系结构中每一抽象层应该包括什么，并且每一层的距离应该是多少。由于对底层技术和不同“易编程”评价标准的假设，这种争论更加激烈。图 1-13 中的软/硬件边界用平面来表示，说明可使用的硬件原语在不同的设计中或多或少有不同的复杂性。实际上，随着这个领域的逐渐成熟，这种情况更加突出。在很多早期设计中，物理硬件的设计是围绕特定编程模型的；硬件支持的通信原语在本质上是等同于编程模型的。这种“高层”并行体系结构方式导致了非常多的硬件种类。但是，随着人们对编程模型有了更好的理解，实现技术更加的成熟，编译器和运行库已经开始在编程模型和底层硬件间起到重要的桥梁作用。同时，在 1.1.2 节讨论的技术趋势起到了重要的影响作用。这导致了组织结构上的融合，那就是使用简单、通用的通信原语。 27

1.2.2～1.2.6 节介绍最广泛使用的编程模型，与编程模型对应的以前及现在的并行机器设计风格。由于历史上系统的设计大都是面向某种特例的编程模型，人们普遍将编程模型、通信抽象和机器组织合在一起统称为“体系结构”。比如，共享内存体系结构、消息传递体系结构等等。但今天的并行机中有很多共同点，并且很多机器支持多种编程模型，以前的方法已经不太合适了。知道体系结构的融合是如何出现的是非常重要的，所以在这节，我们将从传统的观点出发，回顾与每一种编程模型对应的机器设计，并解释它们的作用和影响它们的技术。在这里，我们并不是要给出这些机器设计的一种分类，而是要通过它们确定一些核心概念，正是这些概念形成了当前和将来评价机器权衡的基础。它还展示了微处理器和 DRAM 所主导的主要技术潮流对并行机设计的影响，这些技术自然地或强制地影响了许多基本的设计。特别地，在这里将讲解共享地址、消息传递、数据并行、数据流、脉动阵列。每一种方法，我们将介绍嵌入编程模型中的抽象、设计风格的原由和它适合的应用及规模。我们还将介绍背后的技术促动因素以及它们随时间的演变。这种演变反映在决定机器速度的机器的组织结构上。这些性能特点反过来又影响编程模型。这些简要评述所导致的就是在组织结构上的融合，将集中反映在 1.2.7 节中介绍的一种通用并行机上。

1.2.2 共享地址空间

并行机中最为重要的一种就是共享内存多处理器。这类结构的关键特性就是通过传统的内存访问指令（load 和 store）来实现通信。共享内存结构已经有很长的历史，至少从 1960 年起，大型机就开始使用了[○]，而在当前计算机的各个领域中几乎都可以看到它的身影。共享内存多处理器结构为多道程序和并行程序提供了更高的吞吐率。所以，你可以发现它有不同规模的应用，从几个到几百个处理器。这一节中将研究共享内存机器的通信体系结构，以及小规模到大规模应用中的一些关键组织问题。

28

这种机器的主要编程模型是在一个处理器上分时共享，还有就是用真正的并行性来代替分时。严格地说，一个进程是一个虚地址空间和一个或多个控制线程。我们可以配置进程使进程的部分地址空间共享，也就是像图 1-14 那样将它们映射到共用物理地址上。（一般说来，一个进程内的多个线程共享部分的地址空间）线程间的协作通过读写共享变量和指向共享地址的指针来实现。某个线程对一逻辑地址的写操作对其他线程的读操作来说是可见的。这种通信体系结构用传统的内存操作来实现通信和一些原子操作的同步。一般说来，即使完全独立的进程，也会共享部分核心地址空间，虽然这只是由操作系统代码来访问。不过，这种共享的地址空间的模型只是在操作系统内用来协调进程间的执行。

虽然共享内存可用于任意进程间通信，但大多数的并行程序在使用虚地址空间时都经过了仔细规划。它们通常有共享的代码映像、私有的栈和数据、进程或线程地址空间中共享的区域。在这种简单结构，程序中的私有变量在每一个进程中，共享变量在每一个线程中有相同的地址和意义。通常，都采用直接的并行策略。比如，每一个进程执行一个并行循环中的一个子集，或者组成一个进程池，从一个共享的队列中获取任务。第 2 章将深入讨论并行程序的结构。在这里，我们只是看一下这种重要的系统结构思路的基本发展和演化过程。

○ 有人认为 BINAC 是第一台多处理器，但它主要是为了提高系统的可靠性。它包含两个处理器，相互检查每条指令的执行结果。不过它们一致的时候太少，因此最终人们关闭了其中一个。

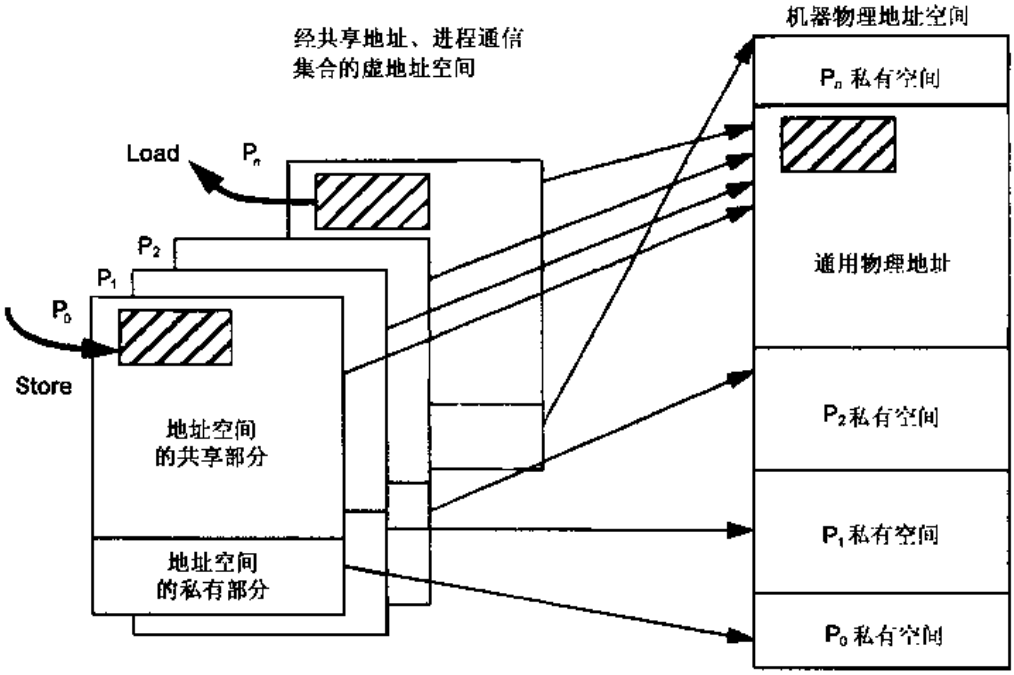


图 1-14 典型的共享内存并行程序的内存模型。很多进程有一个映射到它们的虚拟地址空间的共同物理地址区域、而且这个私有空间一般包含了栈和私有数据

在许多计算机中发现，共享内存多处理器的通信硬件只是对普通计算机内存系统的扩展。如图 1-15 那样，所有的计算机系统都允许处理器或某些 I/O 控制器通过一些硬件互连来访问某些内存模块。增加内存模块数可以扩展内存容量。由于取决于系统的具体组织策略，容量的增加不一定会增加可利用的内存带宽。I/O 能力可以通过向 I/O 控制器增加进一步的 I/O 控制器来扩大。有两种方法来扩大处理能力：用更快的处理器或用更多的处理器。在一个分时的工作负载下，应该通过增加系统的吞吐能力来增加处理能力。如果有更多的处理器，则它们同时运行就可以增加系统的吞吐量。如果单个应用通过多个线程来处理，则更多的处理器应该加速该应用。硬件原语实际上和通信抽象是一对一的，这些操作在编程模型中都可用。 29

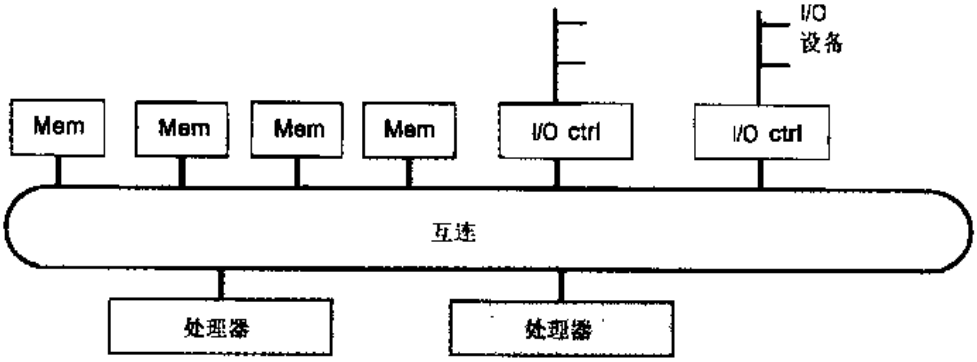


图 1-15 通过加入处理器模块把系统扩展成共享内存多处理器系统。很多系统包括一个或多个可以通过硬件的互连网络被处理器和 I/O 控制器访问的内存模块，典型的是总线、交叉开关或多级互联网络。存储器和 I/O 能力通过增加内存和 I/O 模块来增加其能力。共享存储机器允许通过增加处理器模块来增加处理能力，如阴影部分所示

在图 1-15 所示的总体框架中，随着底层技术的进步，共享内存机器发生了很多演变。早期的机器是“高端”主机配置（Lonergan and King 1961；Padegs 1981）。在技术方面，早期主机中的存储器和处理器相比是很慢的，所以通过多个存储体来交叉存放数据，即使对一个处理器来说，也可以得到足够的带宽；这就要求处理器和每一个存储体之间都要互连。在应用方面，这些系统主要为具有大量工作的吞吐量来设计。这样，为了满足 I/O 的工作负载要求，添加了多个 I/O 通道和设备。要求 I/O 通道直接访问每一个存储体。因此，这些系统一般是通过交叉开关来组织，把 CPU 和多个 I/O 通道连接到多个存储体，如图 1-16a 所示。增加处理器主要是要求扩展交换机；从处理器的一个端口访问内存单元的硬件结构和交换机的 I/O 方面是不必改变的。处理器的大小和代价限制这些早期的系统只能具有少量的处理器，但是当硬件的密度和代价改善的时候，就可以考虑更大规模的系统。扩展交叉开关的代价成为了限制的因素，并且在很多情况下它被多级互连网所替代，如图 1-16b 所示，其中代价随着端口数目面缓慢增加。带来这种好处的代价是，如果所有的端口一起使用，应用程序会感到延迟的增加和每个端口带宽的降低。直接通过每个处理器访问存储器的能力有多个好处：每个进程可以在任何处理器上运行或处理任何 I/O 事件，数据结构可以在操作系统内部进行共享。

30

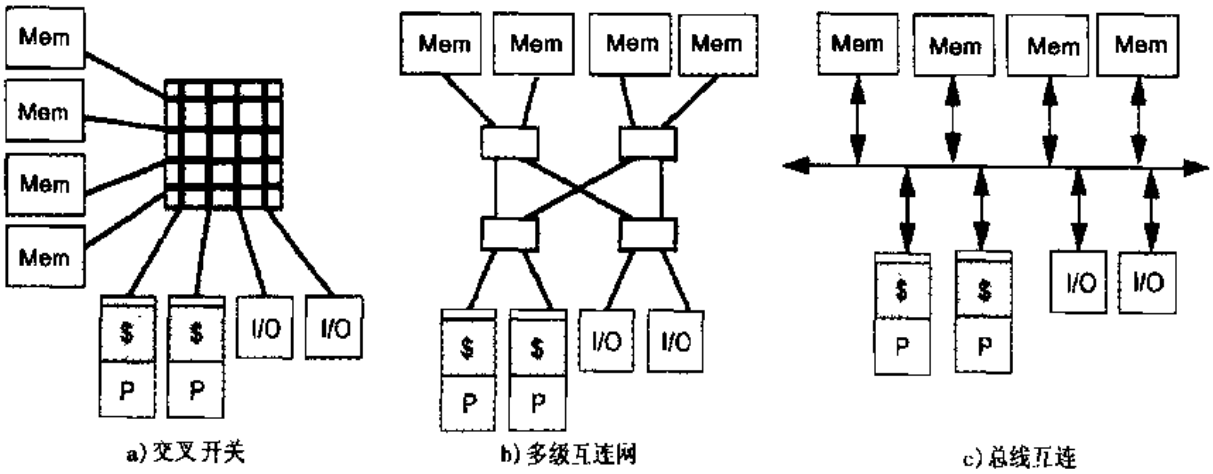


图 1-16 典型的共享存储多处理器互连模式。多个处理器和具有局部缓存（用$ 来表示）以及 I/O控制器可以通过交叉开关、多级互连网络或者总线进行互连，以增加多存储模块

随着 20 世纪 80 年代中期 32 位微处理器的出现，处理器、高速缓存、浮点和存储管理单元可以放置在一个单独的电路板上（Bell 1985），甚至可以在一个主板上安放两个处理器，共享存储多处理器系统开始得到广泛的使用。大多数中档机器，包括小型机、服务器、工作站以及个人电脑，都是通过一条中央的内存总线连接起来的，如图 1-16c 所示；而且总线还可以支持多个处理器。标准的总线访问机制允许任何处理器访问系统中的任何物理地址。就像基于交换机的设计，所有的存储单元和所有的处理器距离相等，所以所有的处理器在内存访问时具有相同的访问时间或延迟。这种配置通常被称作对称多处理器（SMP）[⊖]。SMP结构

31

⊖ SMP 这个术语在很多场合使用，但引起了一些歧义。到底是什么需要对称？从某种意义上讲，许多设计都是对称的。更加精确的刻画 SMP 意图的是共享存储多处理器，其中访问一个存储单元的代价对所有处理器都是相同的；即，它对存储器的访问代价是统一的。如果一个存储单元被缓存了，访问会快些，但缓存的访问时间对所有处理器也是相同的。

在并行程序和多道程序中均被广泛使用。基于总线的典型对称多处理器的组织在图 1-17 中示出。它描述了市场上出现的第一类高集成的 SMP 结构，图 1-18 表示了高端服务器的组织，把物理存储分配到处理器的各个模块，但是保持了对称访问。

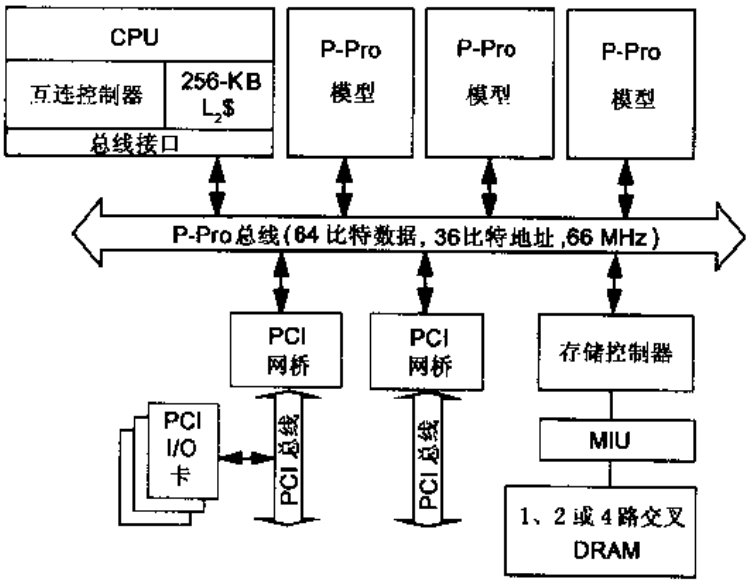


图 1-17a 包含 4 个 Intel Pentium Pro 处理器的“四封装”的物理和逻辑组织。在很多多处理器服务器中使用的 Intel 的 4 处理器 Pentium Pro 主板，表明了在大多数的小规模共享存储多处理器中所使用的主要设计部件。它的逻辑块图 a）表明它可以最多容纳 4 个处理器模块，每一个包含一个 Pentium Pro 处理器、一级缓存、翻译后援缓冲器、256 KB 的二级缓存、一个中断控制器，在一个单芯片上直接和 64 位内存总线连接的总线接口。该总线在 66 MHz 工作，内存事务流水化达到 528 MBps 的峰值带宽。一个双芯片的存储器控制器和 4 芯片的内存交叉单元（MIU）把总线和多存储体的 DRAM 进行互连。一个桥把内存总线和两个独立的 PCI 总线连接起来，包括了主机显示器、网络、SCSI 和低速 I/O 互连。Pentium Pro 模块包含支持多处理器通信体系结构的必要逻辑部件，包括所要求的内存和缓存一致性。Pentium Pro 的“四封装”和很多早期的 SMP 设计相似，但是却具有更高的集成性并且以更高的容量为目标。图 1-17b 示出了典型的 Pentium Pro SMP 的扩展，LX 系列的 HP 网络服务器

来源：经 Hewlett-Packard 公司允许。

32

限制处理器数目的因素，在基于总线和基于交换机的组织方法上是相当不同的。在交换上增加处理器是很昂贵的；然而，随着端口的增加，总的带宽增加了。在总线上增加一个处理器的代价比较小，但是总的带宽固定不变。在很多处理器之间分配固定的带宽限制了这种方法的实际可扩展性（图 1-9 中所示的是关键的总线带宽）。幸运的是，高速缓存减小了每个处理器的带宽要求，因为很多内存引用通过它得到了满足而不需要通过总线到达存储器。然而，在把数据复制到局部缓存之后，使得缓存“一致”就成为了潜在的难题，这将在第 5、6、8 章进行详细讨论。

从小规模的共享存储机器开始，如图 1-16 ~ 图 1-18 所示，我们可能会问要使得设计可以扩展到很多的处理器需要什么样的要求。基本的处理器部件很适合于这个任务，因为它小而经济，但是互连确实还存在问题。由于固定带宽的原因，总线不能够扩展。由于代价和端口数目的平方成正比增长，所以交叉开关也不能够很好的扩展。还存在一些其他的可扩展互连网络，比如当更多的处理器加入时总的带宽也可以增加，但是代价却没有太大增长。这时我们需要认真考虑时延方面的增长，因为处理器可能在数据从处理器到内存模块之间传送的

时候停止下来。如果访问的时延太大，处理器将花费很多的时间等待，这样多个处理器的好处就难以利用。

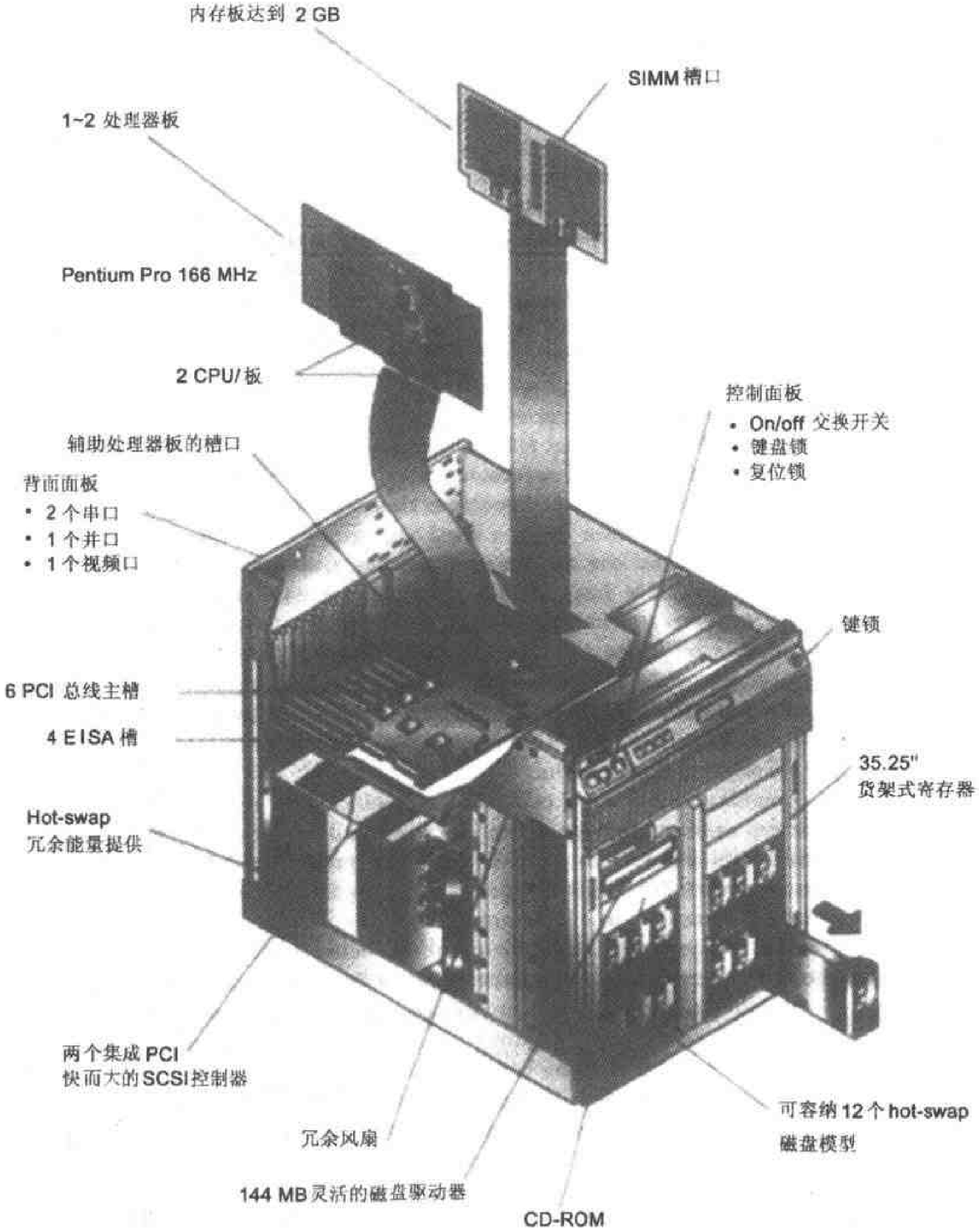


图 1-17b 包含四个 Pentium Pro 处理器“四封装”的物理组织

一个自然的方法是建造可扩展的共享存储机器，以维持如图 1-15 所示的统一的存储器访问方式（也称为“舞池”），并提供一个处理器和存储器之间的可扩展互连。每一个存储器访问都被翻译成网络上的消息事务，就像可以在 SMP 的设计中翻译成总线事务一样。这种

方法的主要不利之处是总的网络延迟在每个存储器访问的过程中形成，并且必须给每个处理器提供高的带宽。

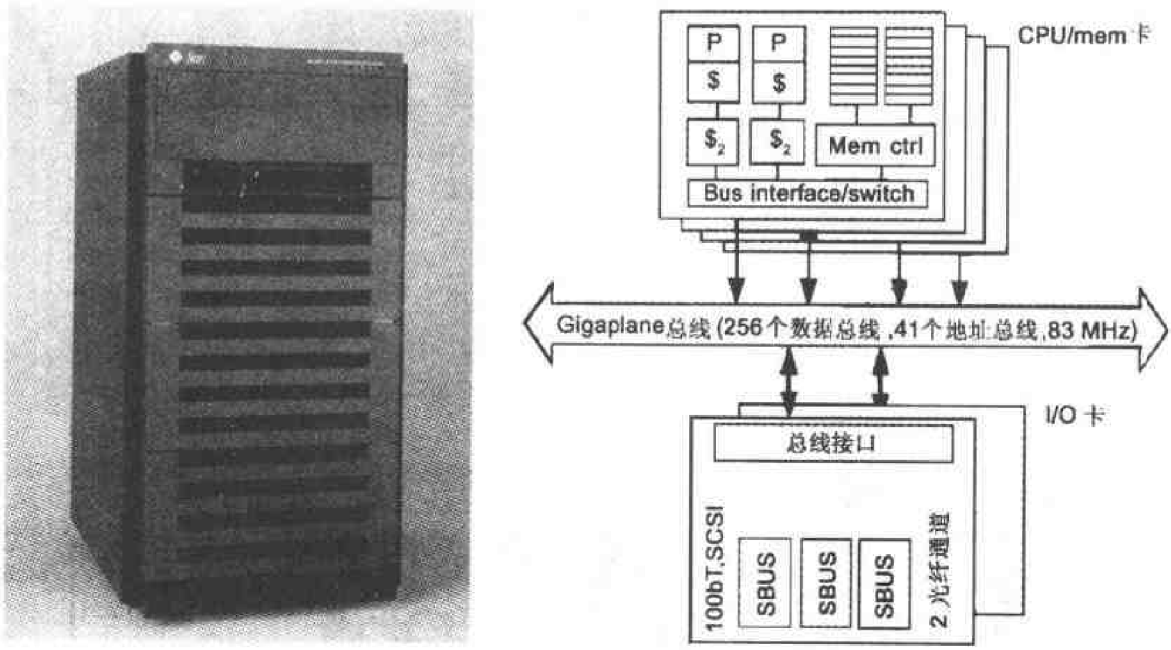


图 1-18 Sun Enterprise 服务器的物理和逻辑组织。基于 UltraSparc Enterprise 的多处理器服务器代表了一种较大规模的设计。该图说明了它的物理结构和逻辑组织。一个宽的（256 位）、高度流水的存储器总线提供了高达 2.5 GBps 的存储带宽。该设计使用了层次式的结构，其中每一个卡或者是一个完整的具有存储器的双处理器或者是完整的 I/O 系统。完整的配置每种类型支持 16 个卡，每种至少一个。CPU/内存包含两个 UltraSparc 处理器模块，每一个具有一个 16 KB 的一级缓存和一个 512 KB 的二级缓存，外加两个 512 位宽的存储体和内部交换机。这样，增加处理器增加了存储器的大小和存储交叉度。I/O 卡提供了用来 I/O 扩展的 3 个 SBUS 插槽，一个 SCSI 连接器，一个 100bT 的以太网端口，和两个电缆通道接口。一个典型的完整配置应该具有 24 个处理器和 6 个 I/O 卡。尽管存储体和处理器成对封装，所有的内存和所有的处理器距离相等，并且通过通用总线来访问，保证了 SMP 的特点。数据可以被放置在机器的任何位置，而不会有任何性能的影响

来源：照片的版权为 Sun 微系统公司所有，经过允许使用。

另一种可能的方法是把处理器和局部存储器构成的“节点”相连，就像图 1-19 所示的那样。在这种非统一的内存访问（NUMA）方法中，局部存储器控制器决定了是否执行局部存储器访问或者通过一个远程的存储控制器来执行消息事务。访问局部存储器比远端的速度更快（I/O 系统可以是每个节点的一部分，或者固定到特定的 I/O 节点中，这里没有显示出来）。访问私有数据，比如代码段或堆栈段，常常可以在本地执行，就像访问放在本地的共享数据。快速访问本地存储器的能力没有增加远程数据的访问时间，所以它减少了平均的访问时间，当访问的大部分是本地数据的时候尤为如此。同时，对网络的带宽要求也降低了。尽管纯粹共享的对称处理器要显得简单些，NUMA 方法在大规模的机器上已经成为主流；这是因为它潜在的性能优点，并且它很好地利用了主流的处理器存储系统技术。这种风格的一个例子是 CRAY T3E，如图 1-20 所示。这种机器反映了以下的观点：尽管每个处理器对每个存储体都可以进行访问，处理器间的存储分布是对编程者公开的。高速缓存只是用来存放来自局部存储器的数据（和指令），避免经常进行远程数据引用成为编程者的责任。SGI Origin 是这种具有同样组织结构的例子，但是它允许数据从任何存储器复制到任何缓存中，并且提

33
~
35

供了硬件支持以使得缓存保持缓存一致性而不必依靠互连所有模块的总线。在写本书的时候，随着两个公司的合并，这两种设计实际上正在走向统一。

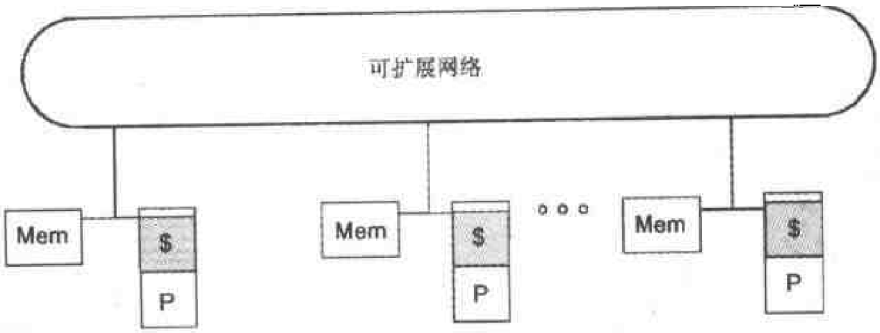


图 1-19 非统一的内存访问（NUMA）可扩展的共享内存多处理器组织。处理器和存储模块紧密集成，以使得对局部存储器的访问比对远端存储器的访问更快

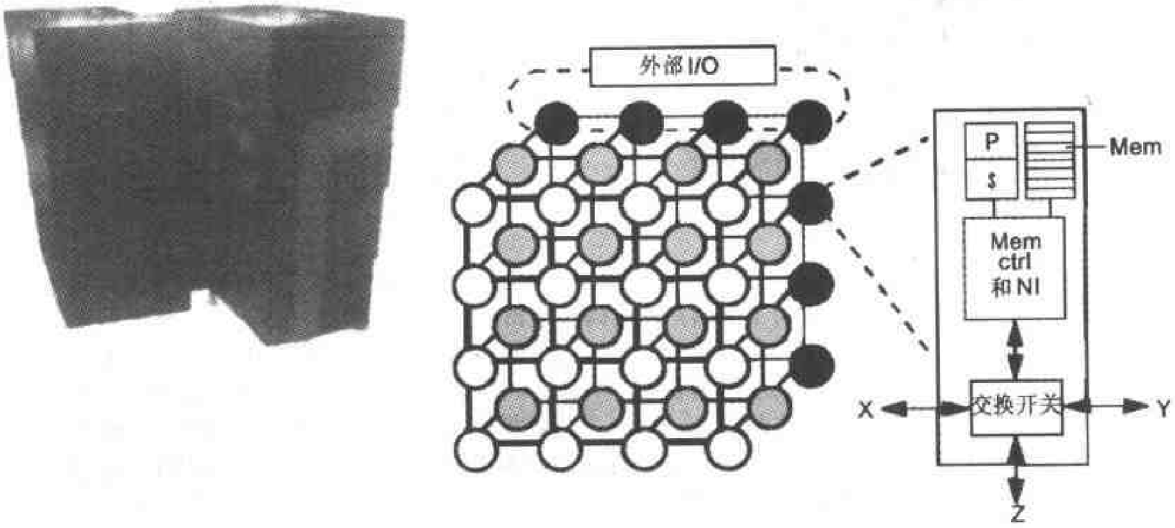


图 1-20 CRAY T3E 可扩展共享地址空间机器。CRAY T3E 可以扩展到多达上千的处理器，并可以支持全程共享地址空间。每一个节点包含了一个 DEC Alpha 的处理器、局部存储器、一个和存储控制器集成的网络接口以及一个网络交换机。机器组织成了一个三维的立方体，每个节点通过 650 MBps 的点到点链路连接到它的六个邻居。任何处理器可以读和写任何的内存位置；然而，NUMA 的特点在于它的通信结构和它的性能特征。需要一个短的指令序列来增加对远端存储器的寻址能力，使得可以通过传统的装入和存储来访问。内存控制器截获对远端存储器的访问，并代表局部处理器和远端节点的存储控制器进行一个消息事务。消息事务通过中间节点自动路由到目的节点，并且每一“跳”都有一个延迟。远端的数据没有被缓存，因为没有硬件机制保持它的一致性。（我们以后还将考察其他的设计，它们允许共享数据通过处理器缓存被拷贝）CRAY T3E 的 I/O 系统在立方体的表面进行分布，通过附加的 I/O 网络和外部世界相连

来源：CRAY Research 提供的照片。

36

总之，共享地址空间编程模型中的通信和协作包括对共享变量的读和写；这些操作直接映射到对全局共享地址空间进行装入和存储的操作，直接在硬件上通过对共享物理地址空间的访问来实现。编程模型和通信抽象和实际的硬件非常贴近。每一个处理器可以命名机器中的每个物理地址单元；一个进程可以在它的虚拟地址空间中命名它和其他部分所共享的数据。数据的传送可能以指令集的原始类型（字节、字等）为单位，也可能以缓存块为单位。每一个进程在它的虚拟地址空间内对它的地址执行内存操作；地址翻译过程确定了一个物理单元，可能是在本地，也可能在远端，还可能和其他进程共享。在每一种情况下，硬件直接

对它进行访问，而没有用户和系统软件的干预。地址翻译在共享地址空间内实现保护翻译，就像单处理器一样，进程只能访问它自己虚拟地址空间内的数据。

共享内存方法的有效性取决于存储器访问的时延，以及可以支持的数据传输的带宽。如同存储层次使得绑定到一个地址的数据能向处理器迁移那样，从存储地址空间的角度来表达通信可以使共享的数据向访问它的处理器迁移。然而，通过一个通用的互连网络来迁移和复制数据提出了一类特殊的难题。我们将看到为了在这样一个设计中得到可扩展性，整体解决方案，包括用来维护一致性共享内存抽象的硬件互连机制，必须能够很好地扩展。

1.2.3 消息传递

第二个重要的并行机家族，称为*消息传递体系结构*，它把整个计算机作为基本模块（包括微处理器、内存、I/O 系统），并用显式的 I/O 操作完成处理器之间的通信。消息传递机器的高层框图和 NUMA 共享内存的方式基本上是一样的，如图 1-19 所示。主要的区别就是通信是在 I/O 系统而不是在内存系统集成的。这样的设计和工作站机群有很多相同之处，只是节点间的联系要紧密，并且不是每个节点都有终端和键盘，互连网络的性能要比标准局域网高得多。处理器和网络之间的联系要比传统计算机中处理器和 I/O 的联系紧密得多，这是因为传统的计算机中处理器是和较慢的外设通信，而消息通信从根本上说是处理器和处理器之间的通信。

37

在消息传递中，程序模型和实际的硬件原语之间有很大的距离，所以用户通过操作系统或系统库来调用低层的通信操作。这样，我们的讨论从通信抽象开始并简要地看一下硬件是如何组织来支持这种抽象的。

在消息传递系统中，用户级通信的大多数操作是不同类型的发送和接收。在最简单的形式里，*发送*操作指明一个发送数据的局部数据缓冲区和一个接收进程（典型情况是在一个远程处理器中）。*接收*操作指明一个发送进程和一个局部数据缓冲区，以存放收到的数据。如图1-21所示，匹配的发送和接收就使得数据从一个进程传送到另一个进程。在大多数的消息传递系统中，发送操作可以在消息中指定一个消息标志，接收操作可以定义一个匹配规则

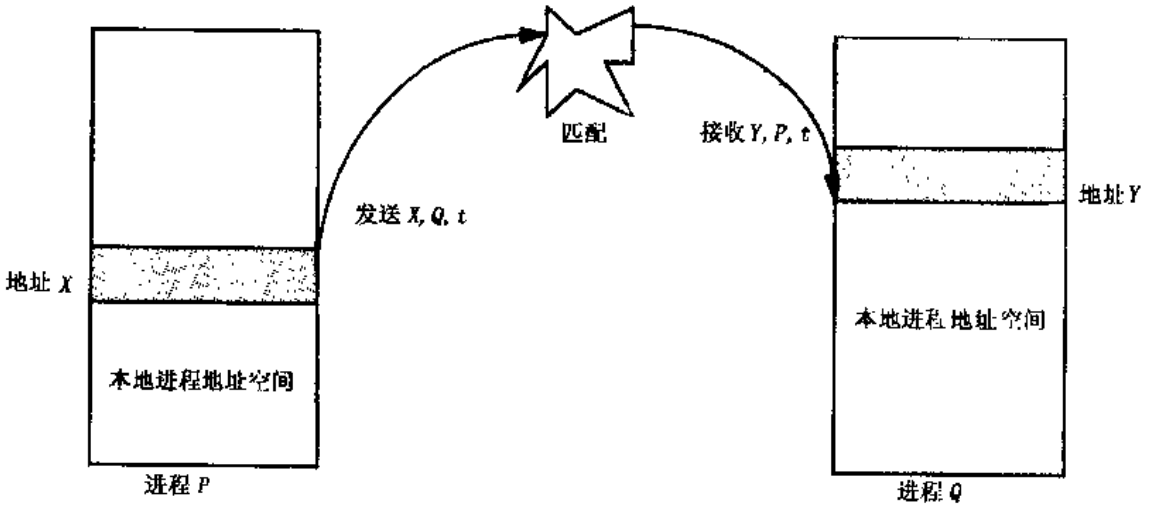


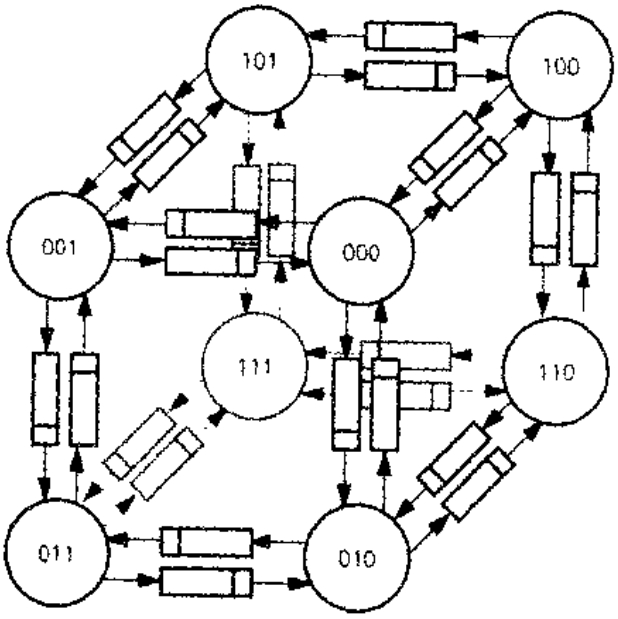
图 1-21 用户级发送/接收消息传递抽象。如果发送进程和接收进程匹配，就会出现一个从本地地址空间到另一个本地地址空间的数据传输

（比如，来自特定处理器的特定标志或来自任意处理器的任意标志）。这样，用户程序相当于用抽象的“进程-标志”来为自己本地的地址和数据项命名。进程间的发送和匹配接收同步完成并在内存中拷贝数据，每个进程都有自己的局部数据地址。这种同步事件有好几种形式，依赖于发送是否完成和接收是否在进行，何时用户的发送缓冲区可重用和请求已被接受。类似地，接收端可以等待用户发送或只是简单地应答。每种形式都有不同的语义和不同的实现要求。

消息传递在一组协作的顺序化进程之间，作为一种通信和同步的方法已有很长时间了，这些进程甚至在一个单处理器之中。一些重要的程序语言，如 CSP 和 Occam，和一些操作系统函数，如 socket 都是基于消息传递模型的。用消息传递方式写的程序通常是相当结构化的。在大多数情况下，所有的节点执行程序的同一拷贝、相同代码和各自的变量。进程通常用一个简单线性序中的元素来互相区分。

早期的消息传递机提供了硬件原语，和简单的用户层发送/接收通信的抽象很相似，只是稍微有些额外的限制。一个节点以一种规则的模式连接到固定的一个相邻节点集，所采用的点对点链接的行为类似于一个 FIFO（Seitz 1985）。这种设计如图 1-22 所示的小型三维立方体。许多早期的计算机是超立方体结构，有 2^n 个节点，每个节点都和 n 个二进制编址相差一位的节点相连。还有一些是网格结构，每个节点在二维或三维空间上和相邻节点相连。网络拓扑在早期消息传递的计算机上是重要的，因为每个节点只能和相邻的节点通信。数据传输涉及到发送方对链路写和接收方从链路上面读。FIFO 的容量很小，发送者通常在接收者开始接收数据之前不能完成发送，因此发送操作会被阻塞直到接收出现（按照现代的术语，两个事件同时发生，这被称为同步消息传递）。数据移动的细节被消息传递库隐藏而不为程序员所见，这种库构成发送与接收调用和硬件之间的一个软件支持[1]。

38



39

图 1-22 早期消息传递机的典型结构。通过 FIFO，每个节点和三个维上的邻近的节点相连

简单的FIFO设计很快被更通用和更健壮的设计取代了。新的设计在节点间采用了直

[1] 同步消息传递的动机不只是机器结构的问题，它也出现在重要的程序设计语言中，尤其是 CSP（Hoare 1978），这是由于它有着很好的理论性质。在微处理器时代的早期，这种技术路线集中地体现在一种单片机中，即 Transputer，在 INMOS 作为计算领域变革开发它的期间曾相当有名。

接存储器访问（DMA）传输。一个 DMA 设备允许在内存和 I/O 之间传输数据而不引起 CPU 的介入。DMA 允许无阻塞传输，发送者可以立刻完成操作并进行其他的计算。在接收方，通过 DMA 传输的消息被放到消息层的缓冲区中排队，直到匹配的处理进程取走这些数据，将它们拷贝到接收进程的缓冲区。

在早期的并行机中网络的物理拓扑对程序设计模型是十分重要的，以至于在并行算法中要指定特别的互连拓扑，如环、网格和立方体结构（Fox et al. 1988）。尽管如此，为了使计算机更加通用，消息层的设计者提供了在所有处理器之间通信的原语，而不仅仅是相邻节点。这是通过让数据和消息在网络中链路上转发实现的；很快，这种路由得到了硬件实现（见第 10 章）。所以每个节点由带有内存的处理器和交换机组成，交换机转发消息。尽管如此，在这种存储－转发的设计中，传输时间和经过每一跳的时间和跳数是相关的，所以问题的重点仍然是互连拓扑（一个简单的存储转发的例子请见习题 1.7）。

随着通用网络的引进，对网络拓扑的强调降低了，通用网络以在网络中路由器之间流水的方式处理消息（Barton, Crownie, and McLaren 1994; Bomans and Roose 1989; Dunigan 1988; Homewood and McLaren 1993; Leiserson et al. 1996; Pierce and Regnier 1994; von Eicken et al. 1992）。在大多数现代消息传递机器中，随跳步的增加所增加的延迟越来越短，以至于传输时间主要由将数据从处理器送到网络上的时间来决定，而和要走的距离关系不大（Groscup 1992; Homewood and McLaren 1993; Horiw et al. 1993; Pierce and Regnier 1994）。这就大大简化了程序模型；典型的，处理器被看作是一个有统一通信开销的线形序列。换句话说，通信抽象和如图 1-19 所示的结构相似。这种机器的一个重要例子是 IBM SP-2，如图 1-23 所示；它是由 RS6000 作为节点构建的，有可扩展的网络和通过专用处理器实现的网络接口。另一个设计是 Intel Paragon，如图 1-24 所示，它把网络接口更紧密的集成在 SMP 节点的处理器上，用一个处理器来专门支持消息传递。 40

消息传递机中的处理器只可以对它的本地存储器的单元命名，还可以通过标识数或路由对其他处理器命名。一个用户进程只能命名私有地址和其他进程，它可以用发送/接收调用来和其他进程通信。

1.2.4 融合

硬件和软件的进展将曾经是很清晰的共享内存和消息传递之间的界限变得模糊起来。首先，让我们看一下用户进程可见的通信操作：

- 大多数共享存储机器现在都支持传统的消息传递操作（发送/接收），其方式是通过共享的缓冲区存储。发送把数据或数据指针写到缓冲区；接收从共享的缓冲区读数据。标志和锁被用来控制对缓冲区的访问，例如通报消息的到来。
- 在消息传递机器中，一个用户进程可以构造一个全局地址空间，用指针来体现进程和该进程内的局部虚拟地址。对这样的全局地址的访问可以用软件通过一个显式的消息事务来完成。大多数的消息传递库允许进程接收任何进程的消息，所以每个进程都能为其他进程提供数据。一个逻辑的读操作的实现可以通过对包含相关对象的进程发消息，并接收一个响应。实际的消息可能是用户看不到的，可能是由编译器产生的用来访问共享变量的代码来完成。
- 在消息传递机器中，可以在页的级别实现一个共享的虚拟地址空间。一组进程有一

个共享的地址区，但是，对每个进程来说，只能访问本地的页。当发出的地址涉及一个远程的页时，缺页发生，引起操作系统来从远程节点将该页调来，并映射到用户的地址空间。

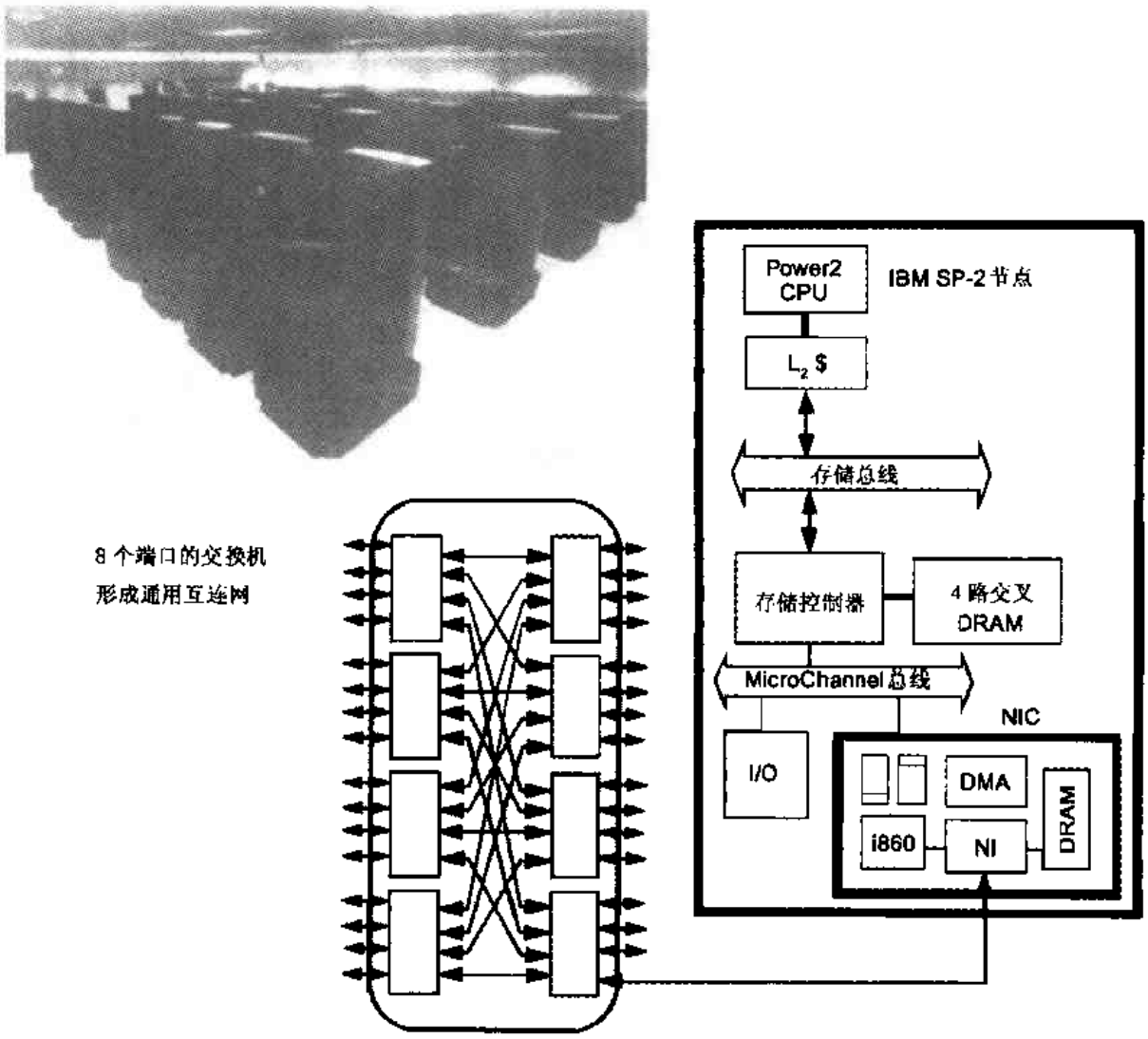


图 1-23 IBM SP-2 消息传递机。IBM SP-2 是一个可扩展的并行机，由 RS6000 构建。一个网络接口卡（NIC）被接到 MicroChannel I/O 总线。NIC 包括处理网络连接的驱动器和一个缓冲消息数据的内存，一个 DMA 设备和 i860 微处理器在内存和网络间传递数据。网络的结构类似蝴蝶。由 8×8 的交叉开关组成。在每个节点上的连接的带宽是 40 MBps，是 I/O 总线的最大容量。一些其他的计算机有相似的网络接口，但是直接接到内存总线而不是 I/O 总线

来源：Ray Mains Photography。

在机器组织的级别也出现了实质性的融合。现代消息传递体系结构在框图层次和如图 1-19 所示的可扩展 NUMA 设计在本质上是相同的。在共享内存的设计中，网络接口和缓存控制器或内存控制器集成在一起，使它能够观察到缓存扑空，从而通过消息事务来进行远程节点内存的访问。在消息传递的方式中，网络接口基本上是一种 I/O 设备。尽管如此，趋势是使它更深地集成到内存系统，并直接在用户地址空间传送数据。一些设计提供跨网络的 DMA 传输，从一台机器的内存直接传到另一台机器，所以网络接口要相当紧密地和内存系统集成起来。消息传递是通过远程内存拷贝来实现的（Barton，Crownie，and McLaren 1994）。

在一些设计中，一个完整的处理器用来辅助通信，与主处理器共享一个缓存一致的存储总线（Groscup 1992；Pierce and Regnier 1994）。从另外一个方面来看这种融合，可见所有大规模的共享存储操作在根本上都是由某种层次的消息事务来实现的。

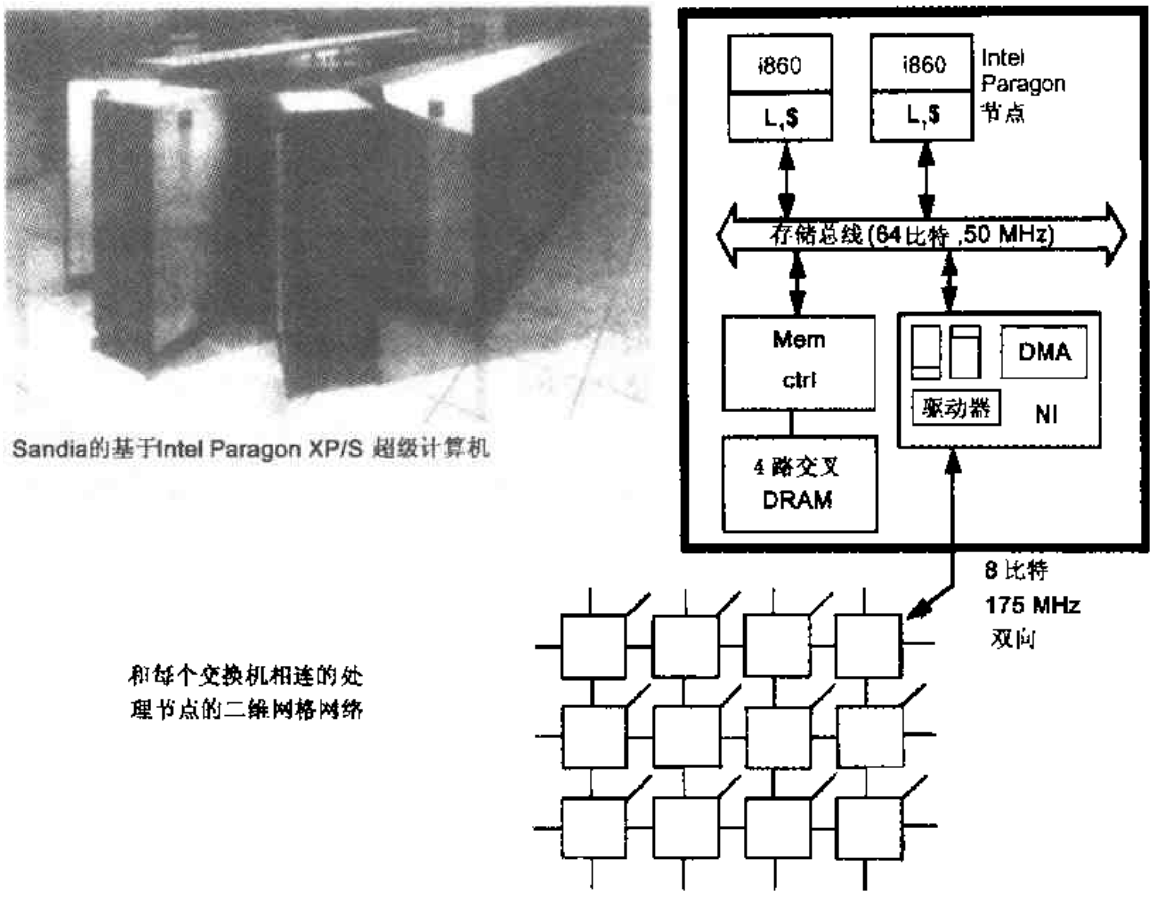


图 1-24 Intel Paragon。Intel Paragon 显示了节点上更紧密的连接。每个节点都是一个由两个或多个 i860 处理器及一块连接在高速缓存一致的内存总线上的网络接口芯片的 SMP。处理器之一用来提供网络服务，另外，节点上还有一个 DMA 引擎用来以较快的速率传递大块的数据。网络是一个 3D 网格结构，很像 CRAYT3E，每个方向上链路速度是 175 MBps
来源：Intel 公司的支持。

除了可扩展的消息传递和共享存储机器的融合外，基于交换的局域网，包括快速以太网、ATM、光通道和一些其他专有设计（Boden et al. 1995；Gillett 1996）已经出现，提供了可扩展的网络连接，这与传统并行机器提供的连接相近。这些新的网络被用来连接若干机器（本身可能是 SMP）构成机群，对于处理大型问题，它就像一个并行机；而对于多道程序，它就像是很多独立机器的组合。实质上，所有的 SMP 厂商都提供某种形式的网络机群方式来获得更高的可靠性。

41 ~ 43

总的来说，消息传递和共享地址空间代表两个完全不同的编程模型，每一个都对共享、通信和同步有明确的定义。尽管如此，底层的机器结构已经在向一种共同的组织结构融合，这种共同的组织结构就是一组完整的计算机，某种通信辅助部件将每个节点连接到一个可扩展的通信网络上。这样，考虑在这种通用框架上支持两种模型就是很自然的了。把通信辅助部件更紧的集成在内存系统上能减少网络事务的时延并能提高网络的带宽。我们会看到这些并理解它和缓存设计、地址映射、保护及体系结构的其他传统方面是如何互相作用的。

1.2.5 数据并行处理

另一种并行机称为处理器阵列、单指令多数据流计算机（SIMD），又叫做数据并行体系结构。名字的改变反映了用户层抽象和机器操作的一种分离。数据并行程序设计模型的关键特征是操作可以同时施行在一个大的规则数据结构（如一个数组或矩阵）的每一个元素上。就是一个操作在许多元素上一起执行。程序在逻辑上是单控制线程的，由一系列串行或并行的操作步骤构成。这种风格有许多新颖的设计，开发各种技术上的可能性，处理器技术的演进是其主导的推动力。

20世纪70年代早期有一篇很有影响的文章（Flynn 1972），提出了一种计算机的分类法，叫做 *Flynn 分类*。该分类法按照同时发出的指令数和指令所操作的数据元素的个数，将常规的串行计算机称为单指令单数据流（SISD），由多个常规处理器构成的并行计算机是多指令多数据流（MIMD）。那时革命性的一种机型是单指令多数据流（SIMD）。它的历史源于20世纪60年代中期，那时一个处理器就是一个满满的机柜，而取指的时间和硬件代价和执行实际指令的代价差不多。于是就产生了一个想法，将所有与指令有关的逻辑都集中到控制处理器中，而数据处理器只包含ALU、存储器和一些简单的互连逻辑。

在SIMD机器中，数据并行程序模型直接在硬件中实施（Ball et al. 1962; Bouknight et al.1972; Cornell 1972; Reddaway 1973; Slotnick, Borck, and McReynolds 1962; Slotnick 1967; Vick and Cornell 1978）。典型情况下，一个控制处理器把指令广播到各个数据处理单元（PE）阵列，这些处理单元形成一个规则的网格，如图1-25所示。人们发现，许多科学计算都表现为对一个数组或矩阵上的每一个元素做同样的操作；对每个元素的计算，经常只涉及到行或列中相邻的元素。这样，问题的并行数据就被分布到数据处理器的存储器中，而标量数据放在控制处理器的存储器中。控制处理器指挥数据处理器对各自本地数据的操作，或者是一起通信。例如，为了计算一个矩阵元素的周围四个邻居的平均值，数据矩阵的拷贝必须在PE阵列上的四个方向上做移位操作，还要做一次本地累加计算。数据PE通常包括一个条件标志，来决定是否放弃一个操作。在一些设计中，局部地址可以采用间接寻址方式，从而使得处理器在做相同的操作时用不同的地址。

44

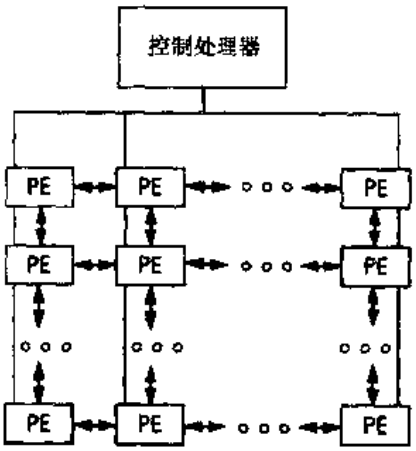


图1-25 典型的数据并行（SIMD）机的结构。通过单个控制处理器的控制以前后紧接的方式处理多个独立的PE的操作。传统上说，SIMD机提供了一个有限的PE间的规则互连，如Thinking Machines公司的Connection Machine和MasPar

处理器阵列在20世纪70年代中期随着向量处理器的发展衰落了。在向量计算机中，一个标量处理器和一些功能单元集成在一起用流水线处理数据。对存储器中任何位置的向量进

行操作的能力消除了将应用的数据结构映射到严格的互连结构上的必要，也大大简化了数据对准的问题。最早的向量处理器 CDC Star-100 提供了向量操作指令集，包括内存中的两个源向量并返回一个结果向量。只有向量元素在内存是连续存放时，机器才能全速执行，因此很多执行时间花在了矩阵的转置上。随着 CRAY-1 的出现，在 1976 年出现了戏剧性的变化。它扩展了用在 CDC 6600 和 CDC 7600 中的 load/store 结构（这种思路在现代 RISC 中再次被采用），将它推广到对向量的操作。存储器中的向量，可以是任何固定跳步的，可以通过 load 和 store 指令装入连续的向量寄存器中。操作在向量寄存器上完成。一个快速的标量处理器（以 80 MHz 运行，在当时是极高的频率）和向量操作结合在一起，并采用了大规模的半导体存储器（而不是磁芯存储器），这个机器一下子就统治了超级计算机世界。在以后的 20 年里，CRAY Research 通过提高向量存储器传输的带宽，增加处理器的数量，提高流水线的级数和向量寄存器的长度等技术，一直领导着超级计算机领域，其性能成长如图 1-10 和图 1-11所示。 45

随着 VLSI 使简单的 32 位处理器的实现成为可能，SIMD 的数据并行机在 80 年代中期获得了新生（Batcher 1974，1980；Hillis 1985；Nickolls 1990；Tucker and Robertson 1988）。数据并行结构中独特的一种做法是在每个芯片上放 32 个 1 位的处理单元；和邻接处理单元串行连接；在控制处理器中维持一个指令队列。通过这种方式，可以组织几千个位串处理单元。另外，人们认识到这种系统的利用率能够被大大提高，如果我们除了提供规则的网状相邻连接外，还提供一种通用的互连，允许任意通信模式在一次操作中完成，尽管这一次操作可能比较耗时（Hillis 1985；Hillis and Steele 1986；Nickolls 1990）。这种方法可以把整数运算和浮点运算的方法扩展到上千个处理器，看起来就像在一个虚拟的 PE 上执行这种操作。

技术因素不仅使位串设计吸引人，还提供了快速的、廉价的单芯片浮点单元，并迅速取代集成了快速浮点单元和高速缓存的微处理器。这就消除了合成时序逻辑的成本优势，使得用很少的完整的处理器组合可以提供同样的峰值性能。在数组上的计算给并行数据处理提供了大量的时间和空间局部性（如果计算正好映射到不多的处理器上），每个处理器处理一部分数据。高速缓存和本地内存用来放属于节点的局部数据，通信在处理边界数据时发生，也可能在全局的数据重组时发生。

于是，一方面在用户程序层次对大规模数据结构上的并行操作提供了解决一类重要问题的有效方法，另一方面针对数据并行程序设计模型的机器组织也在演化，朝着一种更一般的结构，即有多个协作的微处理器构成的结构发展；就像可扩展的共享内存和消息传递的机器一样。在这种演化的过程中，有几种设计保持了支持全局同步的专用网络。这种网络支持的一个简单例子是栅障，每个进程要在程序的某一点上等待，直到所有进程都到达此点（Horiw et al. 1993；Leiserson et al.1996；Kumar 1992；Kessler and Schwarzmeier 1993；Koeniger，Furtney，and Walker 1994）。实际上，SIMD 发展成为了 SPMD，其中每个处理器都执行相同的程序拷贝，从而在很大程度上和更加结构化的共享内存和消息传递编程形式融合到一起。 46

数据并行程序设计语言的实现通常是将每个处理器的局部地址空间看成是某个显式的全局地址空间的不同部分。数据结构是通过全局地址空间来组织的，其索引通过简单的变换映射到处理器和其局部偏移上。计算被组织成若干“块同步”阶段的序列，局部计算或全局通信阶段之间用全局同步栅障分开（Valiant 1990）。因为所有的处理器一起参与通信，都会知

道计算推进的全局状况，所以共享地址空间或消息传递都可以用来实现这种模式。例如，如果某一个计算阶段要做的是让每个处理器向自己“左边的”处理器的某个地址进行一个写操作，可以让每个处理器向左边发送，而从右边接收数据到目的地址中。类似地，如果每个处理器要进行一个读操作，可以首先通过每个处理器发送一个地址，然后送回一个数据来实现。实际上，如果我们考察现代数据并行语言编译器产生的代码，会发现它们和针对广泛用于共享存储和消息传递编程模型的结构化控制并行程序产生的代码基本上是相同的。机器结构上的融合伴随着机器使用方式的融合。

1.2.6 其他并行体系结构

20世纪80年代中期的复兴也使其他一些体系结构的方向得到了重视，学术界和工业界均对它们进行了相当多的研究。但它们在商业上都不如上述讨论的三类那么成功，因此很少在它们上面做并行程序设计。两个曾发展为完整的程序设计系统的是数据流结构和脉动阵列结构。两者在概念上对领域的发展有重要的价值。

1. *数据流体系结构*

数据流的计算模型试图使并行计算的基本方面在机器层显式化，而不利用有可能限制程序并行性的人为约束。它的想法是程序由一个基本数据依赖图来表示，如图1-26所示，而不是显式控制线程序列的固定组合。一个指令可能在获得了它的操作数后的任意时刻被执行。这个图可以在一个处理器集合上随意展开。每个节点指定一个要执行的操作和每个节点的结果地址。在原始形式下，一个数据流机里的单处理器就如一个循环流水线。从网路上来的一个消息或标识（token）由数据和一个地址或一个标记了目的节点的标签（tag）组成。这个标签被用来和相匹配的存储器里的内容比较。如果标签在存储器中，则该匹配的标识将被取出，该指令也将被发送执行。如果该标签不在存储器中，那么它将被放入存储器以等待与之匹配的另一部分。当一个结果被计算出来，一个新的保存了结果数据的消息或标识将被送到指令中指定的每个目的地。不管这些后续指令是在本地还是异地处理器上运行，都可以使用相同的机制。

47

数据流结构主要划分为两种，一种程序图是静态的，这种情况下每个节点代表一个原始的操作；另一种是动态的，这时每个节点能代表一个任意函数的调用，这个函数调用本身由一个图表示。在动态结构，又称加标签的标识结构中，通常是在标签中加入额外的上下文信息动态，而不是真正地修改程序图来实现函数调用时程序图的扩展。

数据流结构的关键特征是它能对在机器中任何地方进行的操作命名，对独立操作的同步的支持以及在机器层的动态调度。当数据流机器设计成熟到进入了由真正的高级并行语言设计的现实系统中时，一种更常规的结构出现了。典型情况是，并行性在程序中是作为并行程序调用或并行循环的结果而产生，因此将这些更大块的工作分派给各个处理器是很诱人的事。这导致了一类设计的产生，其组织形式和图1-19所示的NUMA设计差不多。关键的区别在于对大的、动态的控制线程集合的直接支持以及对将通信与线程的生成集成起来的直接支持。网络和处理器紧密集成；在许多设计中，“当前消息”放在特定寄存器中，该消息中会指明一个线程，通过硬件将该消息发送给线程。另外，许多设计在存储器特定区域提供额外的状态位来提供精细的同步（也就是以元素为单位的同步），而不是用锁来同步对整个数据结构的访问。特别地，每个消息都可以对使用本地寄存器和存储器的一块计算任务进行调度。

48

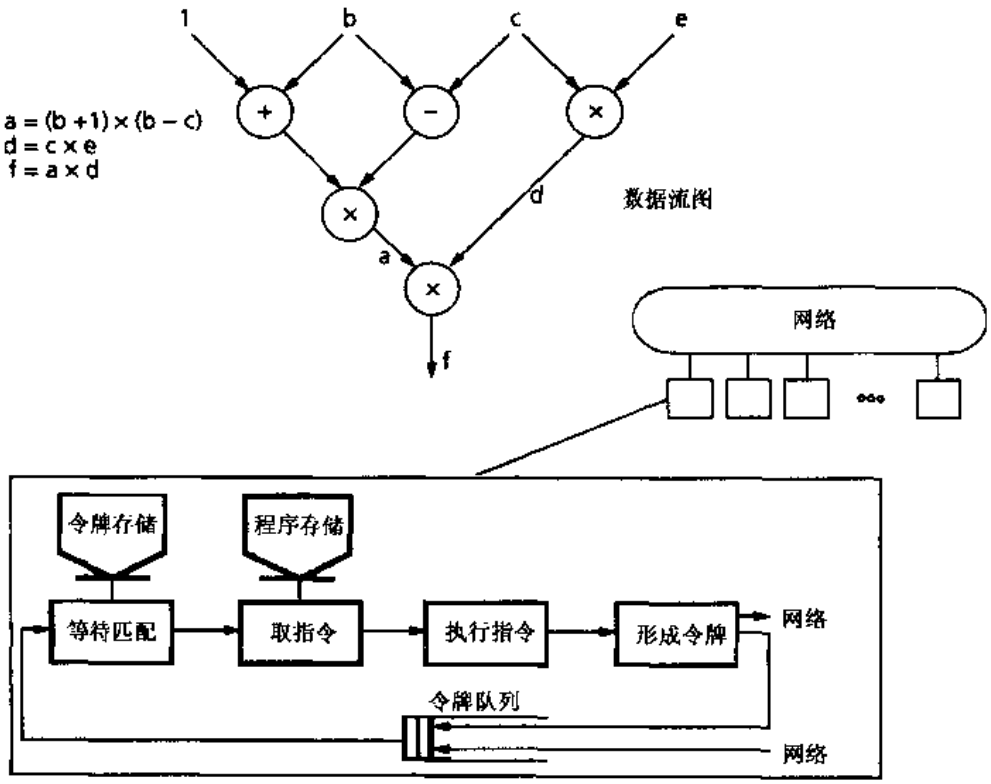


图 1-26 数据流图和基本执行流水线。图中的每个节点当它的输入中所需的操作数都已得到时被激发。它产生结果并将其放在它的输出中，这些输出将被交给图中与之相邻的节点。只有在数据匹配标识存在的情况下，程序执行流水线才能靠检测来实现这种激发规则，它取相应的指令、执行相应操作、并产生相应结果标识

与之相反，在共享内存机器中，一个普遍被接受的观点是一个静态或变化缓慢的进程的集合只在一个共享的地址空间内进行操作，于是编译器或程序会分配循环的迭代，保持一个共享的工作队列，从而将该程序中逻辑上的并行性映射到一个进程的集合。类似地，消息传递程序涉及一个静态或接近静态的进程集合，这些进程之间能彼此命名以进行通信。在数据并行结构中，编译器或定序器用分配一个规则循环嵌套迭代的办法将一个“虚拟处理器”操作的大集合映射到处理器上去。在数据流情形下，机器提供一种能力，来对一个很大的动态线程集合命名，这些线程可以被任意地映射到处理器上。典型情况下，这些机器也同时提供一个全局的地址空间。如同在消息传递和数据并行机器情形中一样，随着这种技术的成熟，数据流结构也经历了程序设计模型和硬件结构的逐步分离。

2. 脉动体系结构

另外一种新颖的思路是脉动体系结构，它试图用一组简单处理单元的阵列来代替一个完整的处理器；通过在这个阵列的单元之间仔细地编排数据流，达到非常高的系统吞吐率，同时对存储器带宽的需求也不高。这种设计和普通流水线的功能单元差别在于这里单元之间的排列结构可以是非线性的（例如，六角形的），处理单元间的通路可以是多方向的，每个处理单元可以有自己少量的本地的指令和数据存储器。这种设计和单指令流多数据流结构的区别是每个处理单元可以做不同的操作。

这种想法是借着超大规模集成电路可以提供廉价的专用芯片的机遇提出的。一个给定的算法能被直接表示为有某种规则互连关系的若干特定运算单元的集合。数据将在运算单元局

部状态的控制下像有规律的“心跳”那样在系统中通过。图 1-27 的例子是一个简单的线性阵列的计算过程。在每次“心跳”中，输入数据向右边前进一步，和一个本地的权相乘，并随着向右的移动放到输出队列上。脉动方法和消息传递、数据并行和数据流模型有一些共同的方面，但在一类特殊问题上有独特的性质。

49

$$y(i) = w1 \times x(i) + w2 \times x(i+1) + w3 \times x(i+2) + w4 \times x(i+3)$$

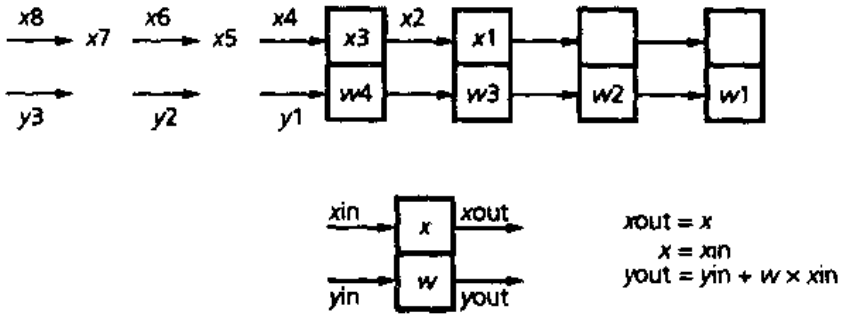


图 1-27 内积的脉动阵列计算，每个方块代表一个运行特殊功能的计算单元。时钟每跳一次，所有的单元接受输入、计算结果、产生输出。数据随着每次跳动经过脉动阵列

在实现中，比如 iWarp（Borkar et al.1990），通常可在节点提供很强的可编程性，以便各种不同的算法都能在相同的硬件上实现。关键的区别在于脉动阵列的网络是由若干专用的数据通道构成的，网络的结构直接对应待求解问题的通信模式，数据能通过这样的专用通道，在处理器的寄存器之间直接转换。对通信模式的全局认识可以减少竞争甚至避免死锁。脉动结构的主要特征是能够将高度特殊化的计算集成在简单、规则、高度局部性的通信模式中。

脉动算法也常常能用在普通的计算机上来求解问题，利用高速的栅障来表达一个个粗粒度的计算阶段。当逻辑脉动阵列的大的分区在各个进程中执行时，这种算法规则的、局部的通信模式产生良好的局部性，所需要的通信带宽很小，同步要求很简单。因而，这些算法被证明在各种并行机中都是有效的。

1.2.7 一个通用并行体系结构

回顾并行体系结构的主要进展，我们看到一个明显趋势，即可扩展的机器向一个如图 1-28所示的通用并行机器结构靠拢的趋势。这样的机器由一类原本完整的计算机构成，每个计算机带有一个或多个处理器和存储器，通过通信辅助部件（一个用于帮助产生外送消息或处理输入消息的控制器或辅助处理单元）和一个可扩展的通信网络相连。虽然这种领域内的整合看起来将缩小设计空间，但事实上，仍存在许多不同方案和大量的争论，焦点在于这个辅助部件中应该提供什么样的功能，它和处理器、存储系统、网络的界面是怎样的。这些是一个相似的组织结构内部的细节区别，记住这一点将有助于我们理解和评价重要结构上的权衡。

50

显然，不同的程序设计模型对通信辅助部件的设计有不同的需求；也决定着哪些操作是更经常的和需要优化的。在共享存储的情况下，辅助逻辑是和存储器系统紧密结合的，从而来获取可能需要和其他节点间进行交互的存储事件。辅助逻辑也必须能够接收消息并为其他节点执行存储操作和状态转换。在消息传递的情况下，不管在系统级还是用户级，通信都是

显式地发起，因此它不需要检查存储系统的事件，但需要迅速启动消息的传送以及对到来的消息进行响应。这个响应也许需要执行标签匹配、分配缓冲、开始数据传输或发布一个事件。数据并行结构和脉动结构都力图强调快速的全局同步，这可以直接在网络上或在辅助部件上实现。数据流结构则强调基于输入消息的、快速的、动态调度。脉动算法则着眼于在局部调度中利用全局模式的机会。即使存在这些不同点，看到所有这些方法的共同之处是很重要的，即作为特别的处理器事件的结果，它们都需要启动网络事务，而且它们都需要在远程节点上执行简单的操作以得到需要的结果。

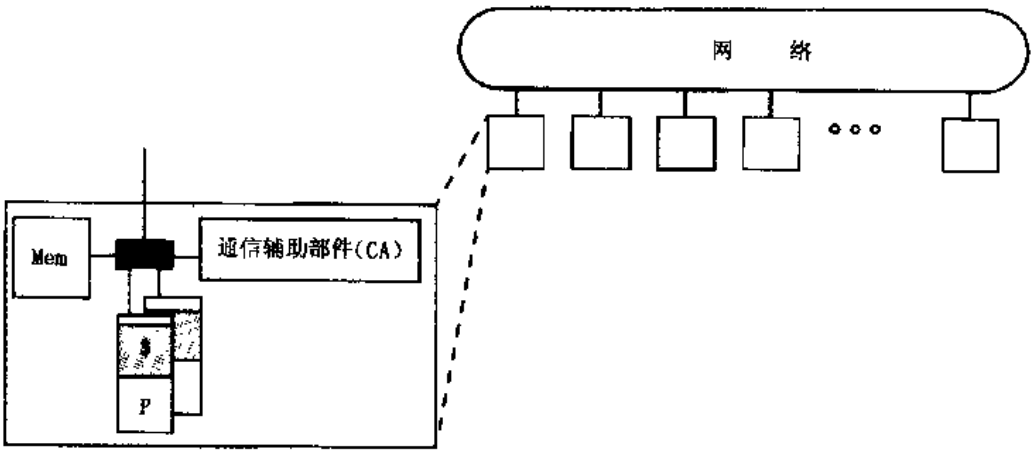


图 1-28 通用可扩展的多处理器组织结构。一个基本完整的计算机的集合，每个计算机包括一个或多个处理器和内存，通过一个通用、高性能、可扩展的网络互连。一般地，每个节点包含一个控制器用来辅助网络上的通信操作

51

我们同时看到，在并行程序设计环境逐渐成熟的过程中，程序设计模型和机器组织结构逐渐分离。例如，Fortran 90 和高性能 Fortran 提供了一种共享地址空间和数据并行程序设计模型，但它们实现在许多不同的机器上，有的只支持一个共享的物理地址空间，其他的只支持消息传递。虽然这些机器从组织结构上看起来类似，但由于在通信抽象层提供的通信和同步操作的不同以及这些操作在性能特点上的巨大不同，在不同类型机器上的编译实现技术是根本不同的。再如，流行的消息传递库，像并行虚拟机（PVM）和消息传递接口（MPI），在各种机器上都有实现，但在不同机器上库的实现是相当不同的。并行操作系统也是一样。

1.3 基本的设计问题

纵观并行体系结构发展至今的历程，我们需要用一种新的观点来看看如何组织这个领域的技术内容。传统的分类法，如 SIMD/MIMD，自从通用多处理器占据主流位置之后，就没有什么意义了。我们不能完全集中在编程模型上，因为很多情况下，很多完全不同的机器结构支持一种通用的编程模型。我们也不能只看硬件结构，因为相同的元素可以在不同的设计中以不同的方式来使用。我们应该做的是关注可能会对不同软件产生影响的机器结构方面的差异。特别是，我们特别考虑的问题是哪些因素会影响编译器从高级并行语言产生代码；哪些因素会影响函数库的作者编写出一个优化的库；哪些因素会影响应用程序使用低级并行语言来编写的方式。于是，我们就可以上从程序是如何使用机器，下从基本技术所能提供什么支持这两方面约束来考虑设计问题。

在本书中，我们通过图 1-13 所示的抽象层次来理解当代并行体系结构的指导性原则。从根本上讲，我们必须理解提供给用户层的通信抽象的操作，不同的编程模型是如何映射到这些原语上的以及这些原语是如何映射到实际硬件上的。过多地强调高级编程模型，而不注意它是如何映射到机器上的，就会偏离正确理解基本结构问题的方向，例如过分强调在每一种特定机器上的特殊硬件机制。

本节将更加强调通信抽象和编程模型的基本要求。它定义了更加形式化的关键概念，把众多的因素联系起来：数据的命名、定序、通信和复制。最后，它引入了用来解决设计权衡的基本性能模型。

52

1.3.1 通信抽象

通信抽象是编程模型和系统实现之间的关键接口。它的作用很像传统串行计算机体系结构中的指令集。从软件的观点看，它必须有一个精确的、定义良好的语义，这样才能使同一个程序在很多不同的实现上正确运行。而且，这个层次所提供的操作必须是简单的、代价明确的、且可以组合的实体，从而使得软件可以在性能上优化。从硬件的观点看，它也必须有一个定义良好的语义，这样机器的设计者才可以决定何处进行性能的优化，而不违反软件的假设。尽管抽象需要是精确的，但机器设计者希望它不要过分特殊，这样就不会妨碍增强性能的新技术的使用，也不会阻碍发掘更新技术的特性的努力。

通信抽象，事实上，是硬件和软件之间的一个合同，允许每一方具有足够的灵活性，并能够正确地协同工作。为了理解这种合同中的“术语”，我们需要更加仔细地看看编程模型的基本要求。

1.3.2 编程模型的要求

一个并行程序包含一个或多个控制流（线程），操作在相关的数据集合上。一个并行编程模型说明了什么样的数据可以在线程中命名，在所命名的数据上能够执行什么样的操作以及这些操作执行的次序。[⊖]

为了使这些问题具体化，考虑单处理器的编程模型。一个线程可以指定在它的虚拟地址空间中的存储单元，并且可以按名字访问机器的寄存器。在一些系统中，地址空间被明确划分成代码段、栈段、堆段等不同的部分；而另一些系统的地址空间采用的是统一的“扁平”结构。类似地，不同的编程模型提供了访问地址空间的不同方法；例如，一些方法允许指针和动态存储分配，另一些则不行。不管有多少不同的方式，基础都是由指令集所提供的、能在可命名单元上执行的操作。例如，在 RISC 机器中，线程可以从内存装入数据和存储数据到内存，但是只能对寄存器中的数据执行算术和比较操作。先前的指令集支持在存储器和寄存器上的算术操作。编译器在硬件和软件的边界上隐藏了这些差别，所以用户的编程模型是对保存数据的变量执行相应的操作。硬件把每一个操作的虚拟地址转换成物理地址。

存储器操作的次序是串行程序中的语句顺序。编程者的观点是在程序中从上到下、从左至右进行变量的读和修改。更加精确的是，从一个地址读出的值是在这个程序的顺序执行过程中，写到这个地址的最后数值。这个次序假设对于程序的逻辑是具有本质意义的。然而，

⊖ 名字（name）、操作（operation）和次序（order）是本书反复涉及的三个概念，它们大致上对应通常说的“操作数”、“操作码”和“操作序”。其中 name 和 order 常以动词的形式出现，就成了“命名”和“定序”。——译者注

读和写可能并没有完全按照在程序中的顺序被执行，因为编译器可能在把程序翻译成指令集的过程中进行了优化，硬件在执行指令的时候也可能进行了优化。但这些优化都要保证程序不能感到次序被改变了。编译器和硬件都要保持程序中的依赖次序，也就是说，如果对一个变量做了写操作，并且此后按照程序的顺序对它进行读操作，它们要保证后面的读操作用到了前面写的值，但是这些操作可能并没有真正发生在内存中；写操作的结果反映在内存中可能是后来的事。在没有写操作的干预下，一组读操作可能被完全重排序；而一般来说，只要保持对读的依赖关系，对不同地址的写操作也可以被重新排序。这种重排序在编译的级别发生，例如，编译器对寄存器的不同分配，对表达式的处理来提高流水线性能或者通过循环变换来减少开销并且改善数据访问的模式。重排序在机器的层次也会发生，例如，指令的流水执行，每周期发出多条指令，用写缓冲区来隐藏访存时延。我们依靠这些优化来提高性能。它们是有效的，因为程序要观察到写的效果，它就必须读此变量；这就产生了数据之间的依赖关系，而所有优化技术都是在保持这种依赖关系的前提下发挥作用的。这样，程序执行序看起来就被保持了，而实际的执行所体现的依赖次序要弱一些[⊖]。我们所处的这个计算机世界，所有的编程语言体现的都是在虚拟地址空间中变量上的顺序操作；但只要不改变程序执行的结果，系统往往都只遵循一个较弱的序关系[⊖]。 53

现在让我们回过来看并行编程模型。本章前面的一些讨论指出了命名、操作集合和其次序的不同地位。命名和操作的集合是编程模型的典型特征，但操作的序问题是关键性的。一个并行程序必须协调它的线程的活动以保证程序中的相关性，这就要求在基本操作中隐含的序不够充分的情况下进行显式的同步操作。作为系统结构师（和编译器作者），我们需要理解操作次序的特点，以看到我们为性能所能做到的优化“技巧”。我们可以将研究集中在共享地址和消息传递编程模型上，因为它们是用得最广泛的。其他的模型，如数据并行模型，经常是依据它们其中一个来实现的。

共享地址空间编程模型假设有一个或多个控制线程，每个线程在一个地址空间中进行操作，其中包含一个共享的区域，也可能包含一块只为该线程私有的区域。典型情况下，共享区域为所有的线程所共享。在私有地址上所定义的操作在共享地址空间上也同样有效；特别地，程序访问和更新共享变量的方式只是简单地将它们用在表达式和赋值语句中。

消息传递模型假设每一个进程都在自己的私有地址空间中进行操作，并且都能命名其他的进程。私有地址空间提供了按照程序顺序的一般的单处理器操作。附加的一些操作、发送和接收在局部地址空间和全局进程空间上进行。发送操作从局部地址空间向一个进程传送数据。接收操作从一个进程接收数据并存放到局部地址空间中。每一个发送/接收对是一个特定的点到点的同步操作。很多消息传递语言也提供全局的或集体的通信操作，例如广播。 54

1. 命名

编程模型中所采用的命名方法通过编程语言和编程环境提供给程序员。这是程序逻辑的基础。然而，命名的问题在通信结构的每一个层次都是很关键的。当然，一个可能的策略是使编程模型中使用的操作和用户/系统边界的通信抽象的操作——对应和下面的硬件原语—

⊖ 这种解释对系统程序员来说有点问题，例如，若一个变量实际上是某个部件上的一个控制寄存器问题就可能发生。在这种情况下，实际的程序语句序就必须被保持。这通常是通过将那样的变量说明为特殊变量来实现的；例如在C语言中用易失性类型修饰符。

⊖ 从而留出较大的优化空间。——译者注

一对应。然而，让编译器和库成为编程模型和通信抽象之间的一层翻译是可能的，还可以让操作系统来处理用户/系统边界上的某些操作。这些不同的技术路线使得系统结构设计师能考虑用硬件来直接实现那些常用、简单的操作，而用软件来实现或部分实现那些较复杂的操作。

针对共享存储和消息传递这两个基本的程序设计模型，让我们考虑命名问题在不同层次中的细节。首先，在共享存储模型中，程序中对共享变量的访问通常是被编译器映射到对共享虚拟地址空间的 load 和 store 指令中，就像对其他变量的访问一样。然而，这不是仅有的做法。编译器也可以为访问共享变量产生特定的代码序列。如果任何处理器都能对机器中的任何单元产生一个物理地址，并且用单个内存操作来访问那个单元，则称该机器支持*全局物理地址空间*。如果提供了一个全局物理地址空间，在机器上实现共享虚拟地址空间就变得很直接了：建立虚地址-物理地址映射，以使得共享虚拟地址映射到同一个物理存储位置上（即进程在它们的页表中有相同的项）。不过，由于翻译可能有不同的水平，因此也可能有其他的方法。我们称一个机器支持*独立的局部物理地址空间*，如果每一个处理器只能访问各自单独的地址空间。即使在这些机器上，也可以通过将局部于进程的虚拟地址映射到对应的物理地址上，来实现虚拟共享地址空间。那些非局部的虚拟地址没有映射，对它们的访问可能会产生一个缺页事件，从而操作系统可以介入来访问远程共享的数据。尽管这种方法可以提供同样的命名、操作和操作序，它确实在硬件/软件的边界需要不同的硬件要求。设计师的工作是要通过各层的系统实现去解决这些设计的权衡，以使得结果是有效的并且在可用的技术下，使得所针对的应用工作负载达到较高的性能价格比。

55

第二，消息传递操作可以直接通过硬件来实现，但是发送/接受操作的匹配以及缓冲方面更适合软件的实现。而用硬件来支持更加基本的数据传输原语。这样，在所有的并行机器中，消息传递模型是通过建构在简单的通信抽象之上的软件层来实现的。在用户/系统的界面上，一种方法是使所有的消息操作通过操作系统，就像它们是 I/O 操作一样。然而，消息操作的频率比 I/O 操作要高，因此用操作系统的支持来配置资源、优先级是有意义的，从而让硬件来直接支持高频率、简单的数据传输操作。在另一方面，我们可以考虑采用共享虚拟地址空间作为低级通信抽象，于是的发送和接收操作涉及了写和读共享缓冲区，加上相应的同步事件。

命名的问题出现在并行体系机构的每一个抽象层面，而不单单出现在编程模型中。作为设计师，我们需要根据通信抽象中所发生的操作的频率和类型去进行设计；理解在这个界面上的权衡就涉及到哪些直接由硬件实现，哪些应该由软件实现。

2. 操作

每一个编程模型定义了一个特定操作的集合，在一个能够被模型所命名的数据或对象上执行。在共享地址模型的例子中，这些包括了对共享变量的读和写以及各种用来同步线程的原子性读-改-写操作。对于消息传递来说，这些操作通过在私有（局部）地址和进程标识符上来进行发送和接收，如前面所描述的那样。程序中的每一个数据元素通过进程号和进程中的局部地址来命名。一个消息传递模型的确定义了某种全局地址空间，但没有在这种全局地址上的操作。操作可以被传递并由程序来解释，例如，模拟一个在消息传递之上的共享地址编程风格，但是它们不能在通信抽象上直接进行。作为设计师，我们需要明白每一层抽象所定义的操作。而且，我们要很清楚在每个抽象层操作的次序是如何规定的；通信在何处发

生；数据是如何复制的。

3. 定序

在并行体系机构的各个层次之间，操作之间次序的性质具有深刻的影响。例如，除了和显式的发送/接收相关的程序操作序外，消息传递模型对于不同进程的操作没有任何次序的假设，而共享地址模型必须指明进程看到其他进程所执行操作顺序的方式。操作序的问题非常重要，也相当不容易理解。在单处理器中为了获得高性能，我们用到许多技巧，包括降低编程者所假定顺序的严格性以得到性能增益、通过并行机制或开发局部性或两者兼用等。在多处理器系统中开发并行性和局部性更加重要。这样，我们需要理解有哪些新技巧可用。我们也需要考察哪些旧技巧还可以使用。我们可以在一个并行程序的每个进程上，从编译器和体系结构的层次执行传统的顺序优化吗？在哪里可以使用显式的同步操作，以允许放松普通操作上的次序限制？为了回答这些问题，我们需要进一步理解程序是如何使用通信抽象的，它们靠的是什么性质以及我们希望在什么样的机器结构中开发性能。

56

一种自然的方法是采用对应线程程序中的操作序。这是编程者对一个特定线程情况假定的。然而，如果多个线程使用同样的共享变量，应该如何假定操作的顺序呢？线程以不同的速度独立的执行自己的操作，所以“最近一次”的概念没有清楚的定义。如果我们把机器看作是在一个共同的、中心的存储器上进行操作的一些简单处理器，就可以合理地期望内存访问的全局序是各个程序访问序的某种任意的交织。实际中，我们不会这样造机器，但是它点明了模型中通过基本操作可能隐含的操作序。这种交织也是我们在运行有若干分时线程的单处理器上可能看到的，而且可能是很细的粒度。

当隐含的序不足以解决问题的时候，就要求有显式的同步操作。并行程序要求两种类型的同步：

- *互斥*保证在同一时刻特定数据上的特定操作只由一个线程或进程完成。我们可以想像有一个房间，要执行这些操作就必须进入这个房间，并且同时只能有一个进程在房间中。这通过在进入后锁门并且在它出去时开锁的方式来实现。如果多个进程同时到达门口，只有一个可以进入，其他的就必须等待直到那个进入的进程离开。允许进程进入房间的次序是无关紧要的，可能和具体执行程序的情况相关；关键是一次只能有一个进入。互斥操作倾向于把这些进程的执行串行化。
- *事件*被用来通知其他的进程已经达到了执行的一些点，使得它们知道已经满足一定的依赖关系，从而可以继续推进。这些操作就像在接力赛中，从一个选手中传到下一个入手中的接力棒一样，或者像发号者开枪以表示比赛的开始。如果一个进程写一个值，而其他的进程要读，就必须进行一个事件同步操作来表明该值可以读了。事件可以是点对点的，发生在两个进程之间；也可以是全局的，发生在所有的或者一组进程之间。

57

1.3.3 通信和复制

同并行体系结构诸层次有紧密联系的最后问题是通信和数据的复制。通信和复制从本质上是相关的。首先考虑消息传递操作。发送/接收操作对的效果是要从发送者的地址空间将数据拷贝到接收者的地址空间。这个传输是接收者访问相关数据的保证。发送者是数据的生产者时，它反映了信息从一个进程到另一个进程的*真通信*。当数据只不过是由于某种原因正

好放在发送方时，例如由于数据的初始配置，或者是由于数据集太大而没法放在任何单个节点上，这个传输只是在需要数据的地方复制。在这种情况下，进程之间实际上并不是通过数据传输将信息从一个进程送到了另一个进程。当数据被适当地复制或放置在有关的进程上时，就没有必要通过消息来传输它了。更重要的是，如果接收方反复使用数据，它就能在不用补充数据传送的情况下，重用其副本。发送方可能修改前面通信地址区的内容，而对接收方不会有什么影响。如果这些更新的效果需要通信，就必须发生补充性的传输。

现在考虑单处理器上通过缓存进行的常规数据访问。如果缓存不包括所希望的地址，就会发生扑空；并且，相应的数据块将从作为后备的存储器中发送出来，隐含地被复制到缓存中。当它在缓存的时候，如果处理器要重用数据，就不再需要内存传输了。在单处理器的例子中，处理器产生数据，并且被处理器所消耗；所以，只有数据不在缓存中或者是第一次访问该数据，才会发生和内存的“通信”。

进程之间的通信和存储层次结构之间的数据传送，在共享物理存储的系统中混在了一起。当一个节点访问某个地址时，只要物理的后援存储不是在该节点的本地，缓存扑空就会引起数据在机器之间传送，无论这个地址是私有的还是共享的，也无论这个传送是真通信的结果还仅仅是数据访问。机器的自然倾向是将数据的副本复制到访问该数据的处理器中缓存。当它在缓存中的时候，如果数据被重用，就不发生数据传送；这是其主要的优点。然而，当对共享数据的写操作发生时，为了保证后面读入到处理器并被复制到它们缓存的是新数据，而不是旧数据，就必须做一些处理。这会包括比简单的数据传送更多的东西。

为了弄清楚通信和复制的关系，区分几个经常在一起出现的概念也很重要。程序执行写操作时，它把数据值捆绑在该地址上；而读操作得到该地址上所绑定的数据值。数据存在于机器上的物理存储部件上。当数据从一个存储部件传送到另一个上时，就发生了*数据传输*；但这并不一定改变地址和值的绑定关系。同样的数据可能存在于多个物理的位置上，就像在单处理器的存储层次中，当时最靠近处理器的那个拷贝才是处理器所能看到的。如果它被更新，其他的隐含拷贝，包括实际的存储单元必须最终被更新。拷贝数据把一些新的地址和相同的值绑定起来。一般情况下，这将引起数据传送。一旦进行拷贝，这两套绑定将是完全独立的（不像在存储层次上所发生的隐含复制），所以对其中一些地址的更新不会影响其他的地址。当一个进程写的数据被其他进程读的时候，就将发生进程之间的*通信*。这也可以在整个机器的范围内引起数据传送，写或者读；或者，也可能因为其他的理由发生数据传送。通信可能包括建立新的绑定或者根据特别的通信抽象去做一些事情。

58

一般来说，复制可以避免不必要的通信，即避免了给数据的需要者传送自从上次访问后再没有改变的数据。在一个给定的通信体系结构层次上进行自动复制的能力，严重地依赖于该层的命名和次序的性质。而且，复制并不是万能的，它也需要数据传输。如果复制的数据不被使用，这显然就是一种浪费。我们将看到复制在并行计算机体系结构中所扮演的重要角色。

1.3.4 性能

在定义通信和协作的操作、数据类型和编址方式时，通信抽象指定了共享的目标是如何命名的，应该维持什么样的序性质以及怎样实现同步。不过，可以使用的原语性能特征决定它们如何被实际地使用。程序员和编译器开发者将在各种可能的情况下避免代价高的操作。

在评价结构上的权衡时，在可行的选择之间所做出的决定将最终依靠它们所提供的性能。这样，为了对并行计算机体系结构的基本问题进行介绍，我们在设计的很多层次上都需要一个框架。

从根本上说，有三个重要的衡量指标：时延，一个操作所花费的时间；带宽，单位时间里可以执行的操作次数；代价，这些操作对该程序的执行时间所产生的性能影响。在处理器每次只做一件事情的简单情况下，这些衡量指标是直接相关的：带宽（每秒操作数）是时延的倒数（每次操作的秒数），而代价是操作执行数目与时延的乘积。不过，现代的计算机系统同时可以做很多不同的操作，在这种情况下，这些性能指标之间的关系非常复杂。考虑下列一个基本的例子。

例 1.2 假定一个部件能在 100 ns 中完成一次特定的操作，显然，它支持每秒一千万次操作的带宽。但是如果部件内部分成 10 个相同的阶段流水作业，它能提供每秒 1 亿次操作的峰值带宽。操作发生的速率取决于最慢的阶段会被占用多久，比如 10 ns，而不是取决于一次操作的时延。对一个应用提供的带宽取决于该应用多么频繁地执行操作。如果应用每 200 ns 执行一次操作，传送的带宽是每秒 500 万个操作，而与部件是否流水化无关。当然，资源的使用通常是爆发式的，所以，当平均初始频率低的时候，流水线可能有利。如果应用在这个部件上执行 1 亿个操作，这些操作代价的范围是什么？ 59

解答： 用操作的次数乘以操作的延迟，将会得到上限 10 s。操作的次数除以峰值速率得到下限 1 s。当程序等待每个操作的完成时，前者是很准确的。后者假定操作和有用的工作完全重叠，所以代价只是启动操作的时间。假设在操作发送给部件后，在依赖操作的结果之前，程序平均可以做 50 ns 有用的工作，这样，应用的代价是每个操作 50 ns，其中 10 ns 来启动这个操作，40 ns 等待其完成，所以整个代价是 5 s。■

由于并行计算机体系结构的特征是通信，我们最关心的相关操作就是数据传输。我们可以通过推广上述流水线例子来理解数据传输操作的性能。

1. 数据传输时间

数据传输操作的时间一般用线性模型来描述：

$$\text{传输时间}\ (n) = T_0 + \frac{n}{B} \tag{1-3}$$

其中 n 为数据量（比如字节数），B 是以相同的单位在部件之间传送数据的传输率（Bps）。T_0 是常量，即启动的代价。这是一个非常方便的模型，可用来描述各种操作的性质，包括消息传递、内存访问、总线事务、向量操作。对于消息传递来说，启动代价可以看作是第一个数据位到达目的地所花的时间。对于内存操作来说，实际上是访问时间。对总线事务来说，它反映了总线仲裁和命令执行阶段。对任何流水线操作来说，包括对指令的流水处理以及向量操作，它是充满流水线的时间。

使用这个简单的模型，显然所得到的数据传输带宽就和传输的数据量有关。当传输量增加的时候，它就接近传输率 B，有时候被称为 r_∞。它达到这个速率的快慢取决于启动开销。不难发现，描述达到峰值带宽的一半时的中值点为：

$$n_{\frac{1}{2}} = T_0 B \tag{1-4}$$ 60

不够理想的是，这个线性模型没有说明下一个操作什么时候可以开始，也没有指出在传输时

可以进行什么样的其他有用工作。这些其他的因素由传输进行的方式来决定。

2. 开销和占用度

我们所感兴趣的数据传输是在并行机器的网络上发生的。处理器通过通信辅助部件来启动传输过程。这个操作的必要成分可以用下面的简单模型来描述：

$$\text{通信时间}(n) = \text{开销} + \text{占用度} + \text{网络延迟} \tag{1-5}$$

开销是指处理器用来启动传输的时间。这可能是一个固定的代价，例如处理器可能只需告诉通信辅助部件启动通信；也可能是 n 的线性函数，例如处理器必须把数据拷贝到通信辅助部件中。关键点在于处理器这时忙于通信事件；此时它不能做其他的有用工作或者启动其他的通信操作。通信时间的其余部分被认为是网络时延（network latency），这部分可以被其他处理器操作所隐藏。

占用度是指通过通信路径上最慢的部件时所用的时间。例如，遍历网络的每一个链接将会占用时间 n/B，B 是链接的带宽。数据将会占用其他的资源，包括缓冲区、交换机和其他的通信辅助部件。通信辅助部件经常是传输的瓶颈，它决定了占用的程度。占用度限制了通信操作启动的频度。在使用同样资源的相继的数据传输序列之前，后面的数据传输必须等待前面的传输不再占用关键的资源。如果在处理器和瓶颈之间还有缓冲，处理器可以以高于占用度倒数的频率发送一批突发式传输。然而，一旦缓冲区满，处理器就得降低到由占用度所决定的速率。一个新的传输只有在旧的传输已结束的情况下才能开始。

剩下的通信时间被计算到网络延迟（network delay）中，包括了一个数据位经路由穿过实际网络的时间以及很多其他的因素，比如穿过通信辅助部件的时间。从处理器的观点看，特定的硬件部件构成了网络延迟的主要组成部分。影响处理器的是在可以使用通信事件的结果之前必须等待多久以及在这个期间有多少时间可用来做别的事情，还有传输数据的频率。当然，设计网络和接口的任务和具体的部件非常相关，并且也关系到它们对处理器所观测到的性能方面的贡献。

[61] 在处理器发送请求并等待应答的简单情况下，把通信时间分解成三个组成部分来考虑没有什么实质的意义。所有关心的就是整个往返时间。然而，在流水线方式发送多个操作的情况下，每一个部件对所产生的性能都有特定的影响。

事实上，所有沿通信路径的单个部件都能通过其延迟和占用度来描述。网络延迟仅仅是沿通路延迟的和。网络占用度是沿通路的占有度的最大值。对于互连网络来说，因为很多传输能同时发生，所以就需要考虑进一步的因素。如果这两个传输同一时刻都去使用同一个资源（比如，它们同时使用同一条线），有一个就必须等待。对资源的竞争增加了平均通信时间。从处理器的观点来说，竞争表现为占用度的增加。系统中一些资源的占用情况由穿过它的一些传输所决定。

等式（1-5）是一个非常通用的模型。可以用来描述很多现代流水化程度很高的计算机系统中的数据传输。作为一个例子，考虑在扑空的情况下，在缓存和存储器之间移动一个块的时间。缓存控制器花费一段时间来检查标志，判断非命中情况并开始一个传输；这就是开销。如果系统中没有更低速的部件，占用度就是块的大小除以总线带宽。延迟包括仲裁的时间和访问总线的时间以及传送数据到存储器的时间。由于竞争，可能会引起更多的等待访问总线的时间或等待存储周期完成的时间。第二个明显的例子是把消息从一个处理器传递到另

一个处理器的时间。

3. 通信代价

我们关心的最基本的指标当然是程序执行通信的时间。一个联系程序特征和硬件性能的有用模型如下所示：

$$通信代价 = 频率 \times (通信时间 - 重叠部分) \tag{1-6}$$

通信的频率定义为程序中每工作单元通信操作的次数，它和很多编程因素（我们将在第2和3章讨论）以及很多的硬件设计因素有关。特别是，硬件可能限制了传送的能力，因此决定了信息的最小量。它可以自动复制数据，或者迁移到它被使用的地方。不过，由于数据必须被共享并且并行执行中有一些所固有的通信，因此处理机必须协调它们的工作。一般来说，对于一个支持高通信频率程序的机器来说，通信代价等式的其他部分所占比重必须比较小，即低负载、低网络时延以及小的占用度。对通信代价的关注决定了机器所能有效实现的编程模型以及它所能支持的应用空间。任何具有很好计算性能的并行计算机可以支持不频繁通信的程序，但是当通信的频率或通信量增加的时候，就会给通信体系结构带来很大的压力。 62

重叠部分是指和其他的有用工作同步进行的通信操作部分，包括计算和其他的通信。这样降低有效代价是可能的，因为很多的通信时间是由系统部件所做的工作占用而不是由处理器所做的工作占用，比如通信辅助部件、总线、网络或者远端的处理器和存储器。和其他工作的重叠通信是一种小规模的并行行为，这正是快速微处理器所开发的指令级并行性。实际上，我们可以利用一个程序中的并行性来隐藏通信的实际代价。

1.3.5 小结

命名、操作集合和操作序在每个抽象的层次都存在，它们是在并行体系结构中普遍适用的概念，并不只是在编程模型中。总的来说，翻译的级别或者运行时软件可以介入到编程模型和通信抽象中，而在这个抽象下面是关键的硬件抽象。在任何层次，通信和复制紧密地相关联。如果两个进程访问相同的数据，数据就需要在两者之间被传送或者被复制，以使得每个进程都能访问数据的拷贝。以一种有意义的方式在给定的抽象层次用同一个名字来指定两个不同物理位置的能力，取决于在这个层次进行命名和定序的方式。在任何涉及到数据传送的场合，我们都需要了解关于时延和带宽等性能指标；此外，还有这些额外开销量和占有度所施加的影响。作为体系结构师，我们需要按照通信抽象来进行操作频率和操作类型的设计，理解在这个边界处进行的权衡，包括硬件直接支持的和软件所支持的权衡。在每一个级别所采用的命名、操作集合以及定序对这些权衡有很大的影响，我们在这本书中将会经常看到。

1.4 结论

并行计算机体系结构是计算机体系结构发展史中一条重要的线索（本质上根植于计算的开始）。在这个历史中，它展现了一种新奇的甚至奇异的作用，通常是由于特定的更高水平的并行程序设计模型，并行计算机设计已经证明了结构的丰富多样性。不过，VLSI技术压倒一切的力量使得并行体系结构得到广泛的应用，已经把并行性进一步地推入了主流。所有

63 的现代微处理器系统内部都是高度并行的，在每个周期执行多个位并行指令，甚至在一致性相关的限制下进行指令的重排序，以额外的和硬件部件来减少通信代价的影响。这些多处理器系统已经成为计算机产业的性能和价格性能进步的主要方向。从功能最强的超级计算机到服务器再到台式机，我们看到把多个处理器集成到一个通信网络上，从而构成整个系统。这种技术，再加上不断成熟的编译技术，就产生了现代并行机器结构组织上的融合。关键的结构问题在于通信如何与存储器及 I/O 系统相结合；这些要点构成了除计算节点外的剩余部分。这种通信结构揭示了关于在硬件级别的命名方式，保证什么样的定序方式，同步操作如何执行；然而，从性能的角度看，我们必须理解通信操作内在的时延和带宽。这样，现代的并行体系结构就带有一种很强的工程味道，体现在对代价和性能权衡的量化分析。

本书提出的基本概念和并行结构的工程问题，跨越了很广泛的潜在设计空间，它们在今天和未来的计算中都会有重要的作用。计算机系统，不管是并行还是串行的，是根据要求和工作负载特点来设计的。对于传统的计算机来说，我们假设该行业的很多厂家能够很好地理解串行程序并且假设编程者都已经做到如何对它们进行编译，知道什么程度的优化是合理的。这样，我们可以非常容易地进行串行程序设计，针对具体的机器进行编译，通过运行程序和评估执行踪迹来得出结论。当我们努力通过体系结构的提高来提升性能时，我们首先假设程序是非常合理的。

并行计算机的情况相当不同。很少有对并行编程的总的理解，并且编程者和编译器的优化还有很大的空间，可以大大影响在机器级别表现出的程序特征。

第 2 章提供了并行程序的综述——它们的外在表现以及它们如何编写。第 3 章揭示了要构造一个好的并行程序程序员和编译器所必须解决的问题，也就是，能尽量的使用多处理器来形成一个体系结构度量的合理基础。最终，我们根据机器级别的程序特征来设计并行计算机，所以第 3 章的目标就是要建立程序和机器如何花费其时间之间的联系。事实上，第 2、3 章把我们在应用的级别对问题的总体理解带入到在通信抽象的层次对特性和操作频率的理解上。

第 4 章建立了一个工作负载驱动的并行设计评测环境的框架。首先必须解决两个相关的问题。第一，对于已经造好的并行机器来说，我们需要一个合适的性能评测方法。这可以通过独立考察机器的各个方面来决定，并且测试它们一起执行的情况。理解应用的特征对于理解负载和证机器性能的关系非常重要。第二，我们需要来评估设想的体系结构的新思想。新的想法需要通过模拟来评价，这就对什么可以被合理的执行加了很多的限制。而且，对应用特性的理解以及如何根据问题来扩展机器规模，对认识整个设计空间来说是很关键的。

64

第 5、6 章研究共享物理地址空间的对称多处理器系统的具体设计问题。在研究可扩展系统的设计之前，深入研究一下小规模的例子有很多重要的原因。第一，小规模的多处理器系统是并行系统最流行的形式；它们很可能是学生们最常见到的形式，也是很多软件开发者的目标，并且很多的专业设计人员都要处理这些问题。第二，在小规模系统中产生的问题会揭示出大规模系统中的一些问题，但是解决方案一般简单且易于把握。这样，这些章节就为在后 5 章讨论大规模系统提供了一个缩影。第三，小规模的多处理器设计是设计大规模机器的基本组成单元。将处理器－内存节点互连起来形成一个可扩展处理器的方案，在很大程度上受小规模机器的处理器、高速缓存和存储结构的限制。最后，小规模系统的设计问题本身是很引人入胜的。

第5、6章设计的基本构成元素是处理器和内存之间的共享总线。我们所要解决的基本问题是要保持高速缓存内容和提供给处理器的内存视图的一致性。总线是一个有力的机制。它通过单独的一组连线提供了任意两个单元之间的通信方式；而且，它可以作为一种广播的载体，因为这里只有一组连线，甚至通过线的“或”（OR）信号提供全程的状态。总线事务的性质在设计传统的高速缓存控制器的过程中被充分利用，以解决一致性的问题。第5章提出了逻辑级别的基于总线的缓存一致性技术，并提出了一些基本的设计选择。这些设计选择提供了如何把工作负载驱动的评测作为决策根据的一个示例。最后，第5章考察了影响软件级别的机器设计的并行编程的问题，特别是考虑到高速缓存对共享模式的影响以及健壮的同步例程的设计。第6章集中讨论组织结构和基于总线的高速缓存一致性机器的实现。它考察了很多更加先进的实现，尽量去减少时延并在保存内存的同一性视图的前提下增加带宽。

第7~11章将对可扩展性并行体系结构设计进行一个综合的考察。第7章阐述概念上的步骤，从将总线事务作为更高层次抽象的构造模块起，到将网络事务作为构造模块。为了进行综合的理解，在这个引论章节所讨论的通信抽象将从基本的网络事务开始。然后，这一章将进一步研究节点到网络接口的设计问题，使用的是一种案例分析的方法。 65

第8、9章深入讨论支持共享地址空间的可扩展机器的设计问题，包括物理地址空间的共享和在多个物理地址空间之上的虚拟地址空间的共享。中心问题是在保持内存同一性情况下的数据自动复制问题，并且避免性能方面的瓶颈。对全局物理地址空间的研究强调了能够提供有效、细粒度共享的全局物理地址空间。对全局虚拟地址空间的研究提供了对最大工作负载的最小硬件支持的准确理解。

第10章考虑可扩展网络设计本身的问题。和处理器、高速缓存、存储系统一样，网络设计空间也具有很多维度，一个设计决策经常包括这些维度之间的交互。这一章列出了可扩展互连结构设计的一些基本问题，示出了一般的设计选择并且评估了在第8、9章中所建立的相关要求。第11章在时延包容的概念下将前面4章的内容串了起来，包括大数据块传输、滞后写、提前读等在通信抽象中的作用。最后，第12章在技术、实现、经济趋势和预测这些并行体系结构领域的关键因素的整体影响下考虑这本书的整体概念。

1.5 历史资料

并行计算机体系结构具有丰富多彩的历史背景，并且和处理器、内存、网络技术的发展紧密相关。并行体系结构最先在1960年左右兴起。这是晶体管替代电子管和其他的复杂逻辑技术的关键时期。处理器越来越小且更容易管理。一个相对廉价、低耗费的存储技术出现了（磁芯存储器），计算机体系结构繁衍成有意义的“家族”。

随着大型机的出现，小规模的共享内存多处理器在这时起到了重要的商业角色。这些机器包括 Burroughs B5000（Lonergan and King 1961）和 D825（Anderson et al. 1962）以及 IBMSystem 360 机型 65 和 67（Padegs 1981）。支持多处理器的配置在 360 体系结构到 System370 系列的演进过程中成为关键的扩展。这包括原子存储操作和处理器间的中断。在科学计算的领域，共享内存的多处理器也非常普遍。CDC 6600 提供了非对称的共享内存组织把多个外设处理器和中心处理器相连，并且提供了一个双 CPU 配置。消息传递机器的起源可追溯到 RW400，始于 1960 年（Porter 1960）。数据并行机器也出现了，采用 Solomon 计算机的设计（Ball et al, 1962; Slotnick, Borck, and McReynolds 1962）。 66

到20世纪60年代末，在处理器内部发生了很大的革新，如使用流水线和功能单元的复制来实现并行性，而不是通过单纯的增加时钟频率来得到更高的性能。很多人认为这些努力即将会得不偿失，Illinois大学和Burroughs启动了一个重要的研究项目来设计和实现一个具有64个处理器的SIMD机器，叫做Illiac IV（Bouknight et al. 1972），基于早期的Solomon项目（尽管Amdahl持有相反的观点［Amdahl 1967］）。这个项目非常雄心勃勃，包括研究基本的硬件技术、体系结构、I/O设备、操作系统、编程语言和应用。1975年，一个具有较小的配置、16个处理器的系统开始运行，计算机业开始了巨大的变化。

首先，作为一个简单的线性阵列的中低速物理设备，存储的概念被革新，开始了虚存的概念和后来的高速缓存技术。Multics和它的前身（例如，Atlas and CTSS）把用户地址空间的概念和物理地址空间的概念区分开来。这就要求维护一些最新的映射机构，如TLB，来得到最合理的性能。EDSAC的设计者Maurice Wilkes，看到了组织可寻址存储的一种有力的技术，即高速缓存。这证明是对并行性应用的一个胜利。在360/85中引入高速缓存，得到了比360/91更高的性能，它具有更快的时钟频率、更快的存储器、具有动态调度的流水执行。在IBM 360/185中高速缓存技术得到了真正的商业应用，但是也为I/O控制器和协处理器带来了严重困难。如果地址被高速缓存，并且没有局限于特定的内存位置，其他的处理器和控制器如何定位合法的数据？一个解决方法是维护每一个高速缓存的目录，这种思想在近些年得到了重要的应用。

第二，存储技术本身随着半导体存储器替代磁芯存储器，发生了巨大的革命。开始，这个技术对小的高速缓存最为适用。其他的机器，比如CDC 7600，只是提供了一个单独的、快速的、可以精确寻址的小存储器。第三，集成电路兴起。总体的结果是单处理器系统在性能上有了巨大的进展，集成了Illiac IV系统中的很多并行特性。CDC STAR-100中的流水向量处理解决了Illiac中试图解决但由于数据移动操作的困难而没有最终解决的数学计算问题。最后引入了CRAY-1系统，由于精致的电路设计和RISC指令集，且具有令人吃惊的80 MHz时钟频率，并且通过向量寄存器来提高向量操作，提供了很高的峰值速率并保证了低启动代价。使用简单的向量处理和快速、昂贵的ECL电路相结合，在随后的15年中占据了高性能计算领域的领先位置。

67

不过20世纪70年代早期微处理器的出现发生了第4个戏剧性的变化。早期的微处理器性能非常低，但其改善是很迅速的，位片设计逐渐过渡到4位、8位、16位和全字设计。这种技术的潜力促使卡内基梅隆大学设计了使用流行的PDP-11小型计算机的LSI-11版本的大型共享存储器多处理器系统。这个工程经过了两个阶段。第一个阶段，叫做C.mmp（通过一个特殊设计的电路交换的交叉开关把16个处理器、存储器和输入输出设备连接起来），非常类似于图1-15中舞池设计（Wulf, Levin, and Person 1975）。第二个阶段，CM*，打算通过把14节点簇通过分组交换网络在NUMA配置中接在局部存储器上，形成一个具有100个处理机的系统（Swan, Fuller, and Siewiorek 1977; Swan et al.1977），如图1-19所示。

这种用许多小的微处理器构造系统的趋势从20世纪80年代早期到中期呈爆炸性增长的态势，导致了多个不同派别的出现。在共享存储器一方，大家注意到了总线的和高速缓存的结合使中等规模的多处理机非常有吸引力。总线的带宽虽然有限，但它是一个广播的介质。高速缓存将对带宽的需求滤掉一些，提供了处理机和存储系统之间的一个中介。加利福尼亚大学伯克利分校的研究工作（Goodman 1983; Hill et al. 1986）引入了一种基本总线协议的扩

展，允许总线维持一种一致的状态。随着32位微处理器的出现和个人电脑业的起飞，在这个方向上形成了若干小的公司，包括 Synapse（Nestle and Inselberg 1985），Sequent（Rodgers 1985），Encore（Bell 1985；Schanin 1986），Flex（Matelan 1985）等等。十年后，这种技术路线统治了服务器和高端工作站市场，并且在PC服务器和桌面系统中也有一定的地位。当高速RISC微处理器的性能优于由多个慢一些的处理器构成的系统时，这种方法曾经历过一段暂时的衰退。尽管RISC处理器也适合多处理机设计，但它们的带宽需要却严重的限制了扩展性，直到一代新型共享总线在20世纪90年代初期出现时才有了转机。

与此同时，消息传递方面有两个主要的研究工作一起开展。在加州理工学院，使用64个i8086/8087的微处理器组建了一个超立方体结构的系统（Seitz 1985；Athas and Seitz 1988）。根据这种设计，其他几个设计也在加州理工学院和JPL（Fox et al. 1988）进行，并且，至少两个公司把这种方法进行了商品化——Intel（iPSC系列）和Ametek。在英国的INMOS公司推动了一种更积极的方法，他们采用了Transputer的方式，也就是把4个通信通道直接集成到一个微处理器上。这种方法被nCUBE所采用，并使用在一系列大规模的消息传递的机器中。Intel公司把商品处理器方法继续推进，使用快速的i860替代i80386，在Delta中使用快速基于网格的互连结构代替了网络并且在Paragon中加入了专门的消息处理器。Meiko在计算领域从Transputer转移到i860。IBM也在Vulcan中用SP系列（实际上是RS6000工作站的群集）得到了商业方面的成功；之前他们研究了Vulcan中基于i860的设计。 68

数据并行系统在20世纪80年代早期，经过一段沉寂之后又出现了。这包括为图像处理服务的Batcher的MPP系统以及Hillis为AI应用所开发的Connection Machine（Hillis 1985）。关键的提高是提供了一种问题要求的通用互连而不是简单的基于网格的通信。这些想法随着Thinking Machines公司的出现开始了商业化，首先是CM-1机器，和Hillis的概念非常相近，然后是集成了大量位并行浮点部件的CM-2。而且，MasPar和Wavetracer将位串或稍宽一些的组织方式放到了廉价的系统中。

20世纪80年代早期出现了一种更加形式化的高度并行系统——脉动阵列机，假设很多简单的处理元素可以装配在一个单独的芯片上。这些数组将给传统的计算机系统提供便宜、高性能、特殊目的的附加物。在一定程度上，这些想法已经在数据并行机的程序中被实现。在CMU的iWARP项目中设计了一个更通用的，在与Intel公司的进一步合作中开发使用的小规模模块。另外，这些想法在快速图形学、压缩和制图芯片中也得到了应用。

由VLSI革命带来的可能性也激起人们研究一些更加激进的体系结构的概念，包括数据流结构（Dennis 1980；Gurd，Kerkham，and Watson 1985；Papadopoulos and Culler 1990；Arvind and Culler 1986）把网络和处理器的指令调度机制很好的集成起来。有人认为通过机器的快速动态调度可以隐藏长的通信延迟和大型机器的同步开销，因此大大简化了编程。这些思想的演进趋向于通过消息驱动的计算和消息传递机制相融合（Dally，Keen，and Noakes 1993）。

大规模的共享存储设计也出现了。IBM在RP-3（Pfister et al. 1985）中进行了高层面的研究，它通过蝴蝶网络互连很多的早期RISC处理器（801）来实现。这是以NYU的超机计算项目为基础（Gottlieb et al. 1983），在使用混合操作方面非常的新奇。BBN开发了两个大规模的设计，即使用Motorola 68000处理器的BBN Butterfly和使用88100的TC2000（Bolt Beranek and Newman 1989）。这些设计，在可扩展的前提下，考察了如何提供一个完全的高速缓存一致性的分布共享存储的问题。斯坦福大学的DASH项目提供了一个通过维护含有每个高速缓

存块存放位置的目录的缓存一致性分布的共享存储器（Lenoski et al. 1993; Lenoski et al. 1992）。SCI代表了标准化互连方式和缓存一致性协议（IEEE 1993）的设计。MIT的Alewife项目试图使对共享存储的硬件支持最小化（Agarwal et al. 1995），并且被威斯康辛（Wisconsin）大学的研究者推动（Wood et al. 1993）。Kendall Square Research KSR1项目（Frank, Burkhardt, and Rothnie 1993; Saavedra, Gains, and Carlton 1993）做进一步深入，甚至允许内存中的数据位置可以迁移。另外，Denelcor HEP通过在一个处理器的上交叉执行很多独立的线程努力隐藏远程内存时延的代价。

69

从20世纪90年代开始了这些不同派别之间的融合，这种融合被很多因素所推动。一是所有的方法都有同样的要求。它们都要求一个快速的、高性能的互连网络。它们从避免延迟和减少它们所发生的绝对时延方面来得到相应的好处并从隐藏尽可能多的通信开销方面得到了很大的好处。它们必须支持不同形式的同步。我们已经看到共享存储在Alewife（Agarwal et al. 1995）和FLASH（Kuskin et al. 1994）系统中努力把消息传递集成在一起，以取得好的性能，而应用的规则性可以提供大的传输量。我们已经看到数据并行设计把完整的商用处理器集成在CM-5中（Leiserson et al.1996），并且允许在用户级进行非常简单的消息处理，这提供了更有效的消息驱动的计算和共享存储（von Eicken et al. 1992; Spertus et al. 1993）。这里还有对快速全局同步的进一步支持。我们已经看到了在NUMA共享内存系统CRAY T3D（Kessler and Schwarzmeier 1993; Koeninger, Furtney, and Walker 1994）中的快速全局同步、消息队列、延迟隐藏技术以及Meiko CS-2（Barton, Crownie, and McLaren 1994; Homewood and McLaren 1993）所支持的消息传递在用户地址空间内提供了直接的虚存到虚存的传输。新的因素可以区别各个派别，如SP-1，SP-2使用完整的商用工作站节点，其他不同的工作站群集使用了刚出现的高带宽网络（Anderson, Culler, and Patterson 1995; Kung et al. 1989; Pfister 1995）。脆弱的存储系统集成的代价，相对不变的网络可靠性以及通用的系统要求使得这些系统和传统的消息传递机制更加一致，尽管将来的发展还并不清晰。

习题

1.1 对表1-1的数据按指数增长率外推，计算晶体管数量、晶片的尺寸、时钟频率的年增长率。同时从Web上获得最新的数据，看这种外推的结果和实际情况相比如何。

1.2 计算表1-2中所示各种计算机在基准测试程序上的性能年增长率。对你所看到的差别进行评述。

70

1.3 在评价性能的权衡时，我们通常要评估由于某种措施所导致的性能提高，或称加速比。形式上，我们有

$$\text{由于措施 } E \text{ 带来的加速比} = \frac{\text{时间（不用 } E\text{）}}{\text{时间（用 } E\text{）}} = \frac{\text{性能（用 } E\text{）}}{\text{性能（不用 } E\text{）}}$$

特别地，我们经常提到加速比是机器并行度的一个函数（比如，处理器的数目）。

假设给定一个程序完成一个固定量的工作，并且那个工作的某一部分 s 必须串行化，其他部分在 P 个处理器上能完全并行执行。假设 T_1 是一个处理器所花费的时间，试推导出 P 个处理器所用的时间 T_p。进而用这个结果来给出 P 个处理器潜在加速比的

上限（这就是常被称作 Amdahl 定律的变量［Amdahl 1967］）。并解释为什么它是上限。

1.4 给出一个如图 1-7 所示的表示可得并行性的直方图，其中 f_i 是在一个理想的机器上发出 i 条指令所用的周期数，试给出 Amdahl 定律的一种扩充形式，来评估在 k-发射超标量机器上的潜在加速比情况。把你的公式应用在图 1-7 的直方图上，产生在该图中所示的加速比曲线。

表 1-1 几种微处理器的基本参数

微处理器名	年	Die(mm²)	总晶体管数	时钟频率(MHz)
i4004	1971	9	2,300	0.5
i8008	1972	12.25	3,500	0.8
i8080	1974	20.25	5,000	3
M6800	1974	25	5,000	1
M68000	1979	43.56	68,000	12.5
i80286	1982	64	130,000	10
M68020	1984	84.64	180,000	25
i80386	1985	90.25	275,000	16
i80486	1988	160	1,200,000	50
MIPS R3000	1988	72	125,000	33
Motorola 68040	1989	126.4	1,200,000	25
Alpha 21064	1992	233.5	1,680,000	160
Pentium 66	1993	294	3,100,000	66.7
Alpha 21066	1994	209	1,750,000	133
MIPS R10000	1994	298	5,900,000	200
Alpha 21164	1995	298.7	9,300,000	300
UltracSparc	1995	315	3,800,000	167

表 1-2 领先的工作站的性能

机 器	年	SpecInt	SpecFP	LINPACK	n = 1,000	峰值 FP
Sun 4/260	1987	9	6	1.1	1.1	3.3
MIPS M/120	1988	13	10.2	2.1	4.8	6.7
MIPS M/2000	1989	18	21	3.9	7.9	10
IBM RS6000/540	1990	24	44	19	50	60
HP 9000/750	1991	51	101	24	47	66
DEC Alpha AXP	1992	80	180	30	107	150
DEC 7000/610	1993	132.6	200.1	44	156	200
AlphaServer 2100	1994	200	291	43	129	190

1.5 从网上找到最近的 TPC 性能数据，从系统配置、性能以及加速比等方面和图 1-4 中的

数据进行比较。

1.6 在消息传递模型中，给每一个进程有一个特定的变量或函数，来表示它在执行程序过程中的惟一的序号或编号。很多共享内存编程系统提供了一个 fetch&inc 操作，它读出一个内存单元的值，并自动对这个值做增量操作。写一个小的伪代码来表明如何使用 fetch&inc 操作来为每一个进程赋一个惟一的序号。你能使用一种简单的方式来决定组成共享内存并行程序的处理器数目吗？

1.7 为了沿着 H 个链路来移动 n 个字节的消息，在一个空载的存储-转发网络中需要 $H\frac{n}{W}+(H-1)R$ 的时间，其中 W 是原始的链路带宽，R 是每一跳的路由延迟。在一个具有直通路由的网络中，这需要 $n/W+(H-1)R$ 的时间。考虑一个 8×8 的包括 40 MBps 链路和路由器的网格，它具有 250 ns 的延迟。在这个网络中移动一个 64 字节的消息需要多少时间？256 个字节的消息呢？

1.8 考虑一个简单的 2D 有限差分模式，其中每一步矩阵中的每个点通过 4 个邻居的平均值进行更新 $A[i,j]=A[i,j]-w(A[i-1,j]+A[i+1,j]+A[i,j-1]+A[i,j+1])$。所有的值是 64 位的浮点数。假设每个处理器一个元素，总共有 $1\,024\times1\,024$ 个元素，则每一步要进行多少数据通信？解释这种计算如何映射到 64 个处理器上，以最小化的数据量进行通信。比较每一步必须进行的数据通信量。

71 ~ 72

1.9 考虑例 1.2 中所描述的简单流水线部件。假设应用在该部件突发的 m 个独立操作和延续 T ns 没有使用该部件的阶段中进行选择。开发一种描述基于这些参数的程序执行时间的表达式。对于 $T=100$ ns、200 ns、400 ns、800 ns，画出平均消息速率和 m 的函数关系，讨论其渐进特性。

1.10 说明如何从式（1-3）得出式（1-4）。

1.11 将式（1-3）看成是关于 n 的一个函数，给出它在横坐标轴上截距的含义。

1.12 如果我们考虑从内存装入一个高速缓存数据块，传输时间是在总线上传输数据的实际时间。起始时间包括得到总线访问权的时间、传输地址的时间、访问内存和在应答处理器之前把数据放进高速缓存中的时间。然而，在一个具有动态指令调度的现代的处理器中，开销可能只包括访问高速缓存检测扑空和把请求发送到总线上的部分。内存访问部分组成了时延，可以被指令的执行所隐藏，但不依靠装入的结果。假设我们有一个运行在 40 MHz，有 64 位宽总线的机器。它使用两个总线周期来为总线仲裁并提供地址。高速缓存线的大小是 32 字节，内存访问时间是 100 ns。读扑空的时延是多少？这种传输所能得到的带宽是多少？

1.13 假设 32 字节的线被传送给另一个处理器，并且通信结构的启动代价是 2 ms，数据传输带宽是 20 MBps。远程操作的总延迟是多少？

1.14 如果我们考虑向另一个处理器发送 n 个字节的消息，我们可能使用和习题 1.12 相同的模型。起始阶段可以被认为是发送 0 长度消息的时间；它包括了两个处理器间的软件开销、访问网络接口的开销以及实际穿越网络的时间。传输时间通常被路径上具有最小带宽的点所决定，也就是，瓶颈。假设我们有一个启动时延是 100 μs 的机器，非对称的峰值带宽是 80 MBps。达到峰值带宽的一半时，消息的大小是多少？

1.15 在一些情况下，式（1-6）被用来估计基于设计参数的数据传输性能。在其他场合中，

它作为把测试的结果拟合到直线上的实验工具来决定有效的起始和系统的峰值带宽。假设在传送一个消息之前，数据必须拷贝到一个缓冲区。基本消息时间如同习题1.14，但在100兆赫机器上拷贝的代价是每32位的字要花5个周期。给出用户层消息时间期望值的公式。将这种拷贝的代价和进入典型操作系统的固定代价相比较。 73

1.16 考虑100 MIPS的机器，运行如下的混合工作负载：50% ALU，20%加载，10%存储和20%转移。假设指令的扑空率是1%，数据扑空率是5%，高速缓存线的大小是32字节。为了计算的目的，处理一个存储扑空要求两个高速缓存线传输，一个装入最新更新的线，另一个替换脏的线。如果机器提供250 MBps的总线，那么在峰值带宽时可以容纳多少处理器？每个处理器的带宽要求是多少？

1.17 习题1.16只考察了平均带宽之和。但是，当总线饱和的时候，为了得到总线的访问权需要较长的时间，所以对处理器来说，内存系统比较慢。效果是使得系统中的所有处理器都慢下来，因此减小了它们对带宽的要求。让我们从其他方面试一试类似的计算。

假设指令的情况和扑空率同习题1.16中一样，但是忽略了MIPS，因为它依靠内存系统的性能。假设处理器在100 MHz运行，并且具有理想的CPI（具有一个完美的存储系统）。初启高速缓存扑空的代价是20个周期。你可以忽略存储的回写。（作为初学者，你可能想计算新机器的MIPS运算速度）假设存储系统（比如总线和存储控制器）在扑空时被利用。使用单处理器的内存系统 U_1 的利用率是多少？从这个结果来看，请估计，在满足处理器要求情况下可以支持的处理器数目。

1.18 我们知道，不管在总线上放多少处理器，它们所看到的带宽总和绝不会超过总线设计的总带宽。解释总线争用对处理器性能发生的影响，试将你的认识作形式化的表述。 74

第2章 并行程序

为了理解和评估在并行计算机系统设计时的一些决定，我们必须对软件在并行机器上运行的情况有所了解。基于对程序行为的理解，我们取得了单处理器系统设计中一些最重要的进展，包括存储层次结构和指令集合的设计。在多处理器的情形，这种理解会更加重要。这不仅是由于设计自由度增加了，从而要考虑的因素多起来；还由于和单处理器相比，应用程序和多处理器系统结构之间的不匹配所带来的性能损失要大得多[1]。

理解并行软件对算法设计人员、程序员和系统结构设计人员都是重要的。对算法设计人员来说，这种理解会帮助他们设计在真实并行计算机上能够有效运行的算法。对程序员来说，这种理解能使他们把握性能增益（或损失）的关键所在，以及如何能在一个系统上获得最好的性能。作为系统结构人员，它帮助我们理解机器所要承担的工作负载以及和它们相关的重要参量。并行软件和它的影响将在以下3章中讲到。本章描述由主流程序设计模型编写并行程序的过程。第3章讲述在这个过程中必须考虑的性能问题，探讨并行应用程序和体系结构之间相互作用的若干要素。基于对这种软硬件相互作用的影响，第4章给出了用并行程序作为负载来评估体系结构设计中诸多折中考虑的指南。除了对体系结构人员有帮助以外，这几章的材料对并行计算机的使用者也是有用的：第2、3章对程序员和算法设计人员是非常有用的，第4章对那些决定要购买何种类型机器的人有很大帮助。尽管如此，我们这里主要针对的还是体系结构人员，讨论他们在从事机器的细节设计之前应该理解的那些问题。

对串行计算机系统的设计师来说[2]，通常认为程序是一个给定的不变因素。这是因为，相关的领域已经成熟，有大量的程序早已存在，我们可以（或者必须）认为它们已不会再变了。于是我们可以针对这些程序的要求来优化机器的设计。尽管我们也知道程序员有可能进一步优化他们的代码（例如当高速缓存变大或者浮点支持有所改善时），但我们在评价新的设计时通常不预测软件的这些变化。编译器可能随着体系结构一起进化，但总认为源程序是不变的。然而，在并行体系结构方面，机器设计和并行软件进化之间的相互作用要强得多，也更具有动态性。由于并行计算旨在追求性能，程序设计的要义就是要充分利用机器所提供的手段来获取性能。并行性为程序设计提供了一个新的自由度（处理器的个数）并表现出较高的数据访问和协调的代价，这就给了程序员一个很宽的软件优化范围。即使作为体系结构设计师，我们也需要打开应用程序的“黑盒子”。对并行软件设计与编写过程中重要方面的理解（本章的内容）会帮助我们认识到体系结构的作用和局限。在下一章对性能问题更深入的考察将更多地讨论硬件/软件的权衡。

即使对应用问题有了准确的理解并有一个好的串行算法，要得到一个相应的并行程序以及它在一个多处理器系统上执行的行为特点，仍然不是一个简单的事情，通常是一个相当费力的过程。这一章给出了程序并行化过程的一般原则，并用实际例子对它们进行了诠释。首先，我们介绍了在下面两章里作为范例的4个实际问题。然后借助于这些例子，描述了开发

---

[1] 即通常所说的峰值性能和实际测得性能之间的差距，在多处理器系统中要比单处理器系统大得多，——译者注

[2] 要面对许多相互影响的因素，例如硬件条件、操作系统、编译技术、应用程序的写法等。——译者注

并行程序的4个步骤；随后是用例子说明如何用当前的主流程序设计模型来编写简单的并行程序。如第1章所讨论的，从程序设计的观点来看，具有统治地位的程序设计模型只剩下了三个：数据并行模型、共享地址空间或共享存储模型、在私有地址空间之间的消息通信模型。本章阐述了由这些模型提供的基本操作及其用法。这些讨论都还没有太涉及到对性能的考虑。通过理解了第3章所讨论的在程序并行化过程中的性能问题，我们将回过头来更详细地研究这4个应用案例，从而得到高性能的程序版本。

2.1 并行应用的案例分析

在前一章里，我们看到多处理器的应用范围很宽，从多道程序设计到商业计算、到所谓重大挑战的科学问题，其中要求最高的是那些来自科学和工程计算中的应用。在下面我们要研究的4个案例中，有两个源于科学计算，一个来自于计算机图形学，另一个属于商业计算。除了来自不同应用领域外，这些案例的选择还展示了许多在其他并行程序中也能观察到的重要行为特征。

第一个例子模拟洋流的运动，其做法是将这个问题离散化为规则的网格，然后求解在这个网格上的一组方程。这种技术在科学计算中是很常见的，所导致的是一组非常规则的通信模式。第二个例子研究代表科学计算的另一种重要形式。在这种形式中，计算的数据域表现为大量的、相互有影响的实体，离散分布在三维空间中，而且它们还由于这种相互作用而运动。这种称为n-体问题的例子在许多领域也是常见的，例如星体物理学中模拟银河系，化学和生物学中对蛋白质和其他分子的模拟以及对电磁相互作用的模拟等。对于解决这样的问题，层次式算法是很受欢迎的。而层次式算法在许多其他领域也有很好的应用。例如我们这里的层次式n-体算法，也可以用来解决计算机图形学中的重要问题以及某些有特殊难点的方程组类型。与第一个例子不同的是，这个例子所对应的是非规则、大范围和不可预测的通信模式。 76

第三个例子来自计算机图形学，是中等规模多处理器中一个很重要的应用。它以一个高度非规则且不可预知的访问模式遍历一个三维空间的场景，并且将它显示成一个二维图像。上述这三个案例是一个标准测试程序集的一部分（Singh，Weber，and Gupta 1992），在文献资料中被广泛用于对体系结构的评价，因此有许多关于它们的详细资料。在本书中，它们也将被用来解释在体系结构设计中所作的种种权衡。

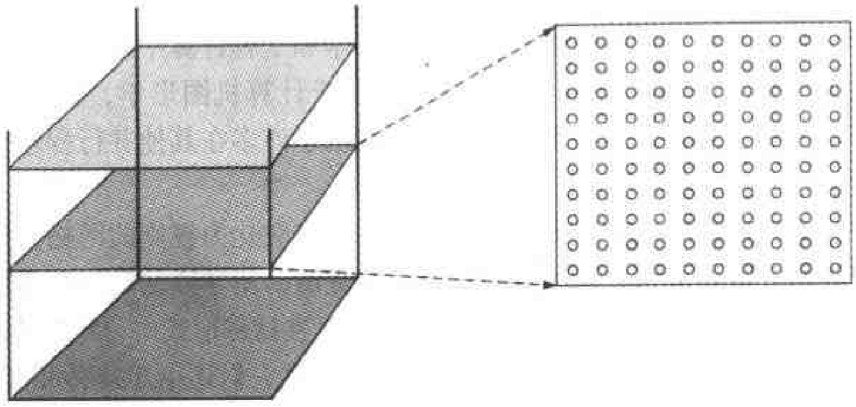
最后一个例子代表一类日益重要的商业应用，即对我们信息社会中产生的海量数据进行分析，以发现有用的知识、类别和趋势。这些信息处理应用是I/O密集型的，对它们的分析使我们能感到计算中I/O活动并行化的重要性。

2.1.1 洋流的模拟

为了建立地球气候的模型，理解大气和占地球表面四分之三的大洋的相互作用是很重要的。这个例子模拟大洋中水流的运动。这些洋流在若干物理力量的影响下发生和发展，这些物理的力量包括大气效果、风以及和大洋底的摩擦力。在洋壁附近，还有附加的“垂直”摩擦力，它导致旋涡流的产生。这个应用例子的目的是要模拟这些旋涡流在一段时间里的行为，理解它们和普通洋流的相互作用。

为洋流行为建立有效的模型是很复杂的：预报大洋在任何时刻的状态需要求解复杂的方程组，这只能由计算机来做。除此以外，我们还对这种洋流随时间的变化行为感兴趣。实际的物

理问题在空间[⊖]和时间上都是连续的，但为了能用计算机来模拟，我们在这两个方面都要做离散化。在空间上，通常用一个三维网格来作为海盆的模型。每个重要的变量，例如压力、速度和各种流，在网格的每一个点上都有一个值。不过在我们这里讨论的程序中没有用三维网格，而是用一组二维网格代之，其中每个二维网格对应于海盆在水平方向的一个截面（见图 2-1）。为简单起见，大洋被看成是一个长方形盆体，假定格点均匀分布。每一个变量因此也就由一个二维数组来表示，对应大洋的一个横截面。在时间上，我们将时间离散化为一个由有限个时间步构成的序列。在同一时间步中，运动方程在所有格点上求解，变量的状态得到更新；然后运动方程再针对下一时间步求解，如此重复。



a) 横截面　　b) 横截面的空间离散化

图 2-1　通过海盆的水平横截面和将它们形成规则网格的空间离散化

77 每个时间步本身由若干计算阶段构成。其中一些用前一时间步得到的结果，在所有网格点建立不同变量的值[⊖]。在另外一些阶段，完成该时间步内方程组的求解。在所有这些阶段，包括方程组求解器，都涉及到遍历相关数组所有的点，对它们的值进行处理。其中求解器阶段稍微复杂一些，我们在第 3 章将进一步详细讨论。

在表达一个固定大小的大洋区域时，在某一维上我们用的网格点越多，我们所做的离散化的空间分辨率就越精细，模拟也就越精确。对大西洋来说，大约 2 000 km × 2 000 km，用 100 × 100 点的网格意味着两点之间的距离为 20 km。这个分辨率并不很高，因此若希望得到较高的模拟精度，应该用更多的网格点。类似地，较短的时间步间隔也导致较高的模拟精度。例如，要模拟大洋 5 年的运动，每 8 个小时更新一次状态，我们大约需要 5 500 个时间步。对高精度的追求会导致很大的计算量，因此对并行处理的需求是明显的。

幸运的是，许多应用问题很自然地表现出大量的并发性：在一个时间步中，在网格点上建立变量初值的许多计算是相互独立的，因此可以并行进行，在一个阶段中处理不同的网格点或网格计算本身也可以并行地来做。例如，我们可以将大洋一个横截面分成若干不同的部分，让每个处理器负责其中一部分，完成其中网格点的有关计算（一种数据并行的做法）。

2.1.2 星系演化的模拟

78 第二个例子也是来自科学计算。它寻求理解银河系里星球随时间的演化过程。例如，我

⊖ 海盆——海洋底部形成的凹形体。——译者注

⊖ 例如按照物理学规律，在网格的点上维持不同物理量之间的一定关系。——译者注

们要研究当银河系碰撞时会发生什么，或者一个随机的星球集合是怎么变成了一个确定的银河形体的。这个问题涉及到模拟许多星体（这里是星体）在万有引力下的运动，这就是所谓 n-体问题。这个计算在空间是离散的，我们可以将每个星体看成是一个单独的体，或者通过抽样法用一个体来代表许多星球。这里我们同样将计算的时间离散化，在时间步序列上模拟星系的运动。在每一个时间步，我们计算由所有其他星球共同施加在每个星体上的万有引力，更新其位置、速度以及星体的其他属性。

计算星球间的力是一个时间步中最耗时的部分。计算力的一个简单方法是计算每对星体之间的相互作用。对 n 个星球来说，这具有 $O(n^2)$ 计算复杂性，因此对我们要模拟上百万个星体是不现实的。然而，利用对力学定律的认识，人们发明的层次型算法能够将计算复杂性减少到 $O(n\log n)$。这就使得在合理的时间里模拟有上百万个星体的问题变得可行，但这也只有用强大的多处理机才可能做到。层次型算法的基本思路是：由于万有引力的强度 $G\frac{m_1 m_2}{r^2}$ 随距离减弱，于是较远星球的影响较弱，因此不需要计算得像那些近的星球那么精确。这样，如果一组星球对一个给定的星球足够远，我们就可以考虑用在其中心的一个“等价”星体，来近似计算这一组星球对给定星球的效果。当然，这会稍微损失些精度（见图 2-2）。进一步来看，如果和给定星球的距离越远，用一个等价星球来近似的星球组就可以越大。事实上，由于许多物理相互作用的强度都随距离而减弱，这种思想所导致的层次型算法在计算的许多领域变得越来越流行了。

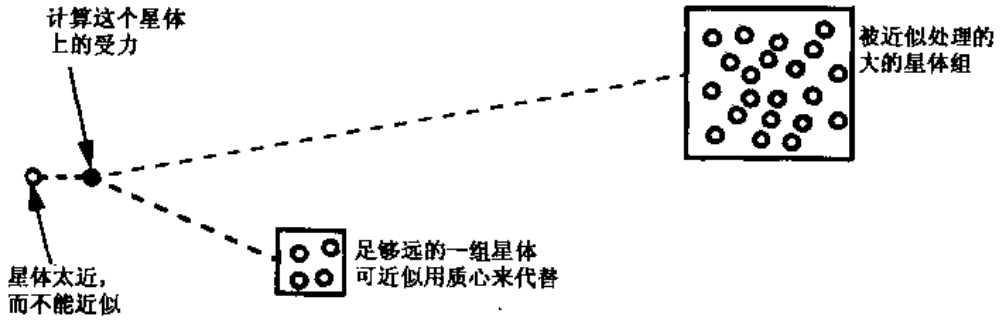


图 2-2 针对 n-体问题的层次型计算方法所带来的认识。对一个给定的星体组来说，一组足够远的星体组可以近似地被该组的质心来代替。距离越远，被近似处理的星体组就可以越大

在本书中，按这种思想计算受力作用的算法是 Barnes-Hut 算法（这个案例分析在文献中称为 Barnes-Hut，因此我们也用这个名字）。在 3.5.2 节里，我们将看到算法是如何工作的。由于星系在某些区域稠密，在另外一些区域稀疏，星体在空间的分布是高度非规则的。这种分布也随星系的演化而改变。这种层次型思路的精髓是：和星球稀疏的区域相比，在稠密区域的星球相互作用（以及和其质量中心的作用）较多，因此计算量较大。在一个时间步里，涉及各个星球的计算存在有大量的并发性。但它们的非规则和动态变化的特性，使得在一个并行体系结构上高效地开发出其中的并发性成为一个挑战。

2.1.3 用光线跟踪法来实现复杂场景的可视化

第三个例子研究的是计算机图形学中复杂场景的可视化。一个用来将这样的场景呈现为图像的常见技术称为*光线跟踪法*。场景被表示为三维空间中的一组物体，被显示的图像表示为一个二维像素阵列，其颜色、透明度、亮度值是要被计算的。所有像素一起表示这幅图，

图像的分辨率由在每一维上像素之间的距离决定。场景表现为从一个特定的视点或眼睛的位置看到的情景。光线从视点发出，通过图像平面的每一个像素进入场景。算法跟踪这些光线的路径，计算当它们碰上物体并从物体反射回来时它们的反射、折射以及和光照的相互作用，从而计算对应像素的颜色和亮度值。通过不同像素的光线是不同的，因此它们之间有明显的并行性。这个案例称为光线跟踪（Raytrace）。

2.1.4 针对关联性的数据挖掘

信息处理正迅速成为并行系统的一个主要市场。企业有许多关于顾客和产品的数据，要通过计算从它们中自动抽取有用的信息或“知识”。有关顾客数据库的一个例子可能包括决定不同人群的购买模式或者按照购买模式将顾客区分开来。这个过程称为*数据挖掘*（data mining）。它和标准数据库查询的不同之处在于其目的是要识别数据中隐含的趋势和模式，而不是通过直接的、显式查询命令简单地查找数据。例如，找到所有上周买了猫食的顾客不是数据挖掘；但是将顾客按照一定的关系分出层次来就是数据挖掘，这里的关系可能是年龄段、月收入以及在猫食、汽车和厨房用具方面的偏好等。

80

数据挖掘的一种形式是关联挖掘。这里，目标是要发现在已有信息中的相关性（比如在不同的顾客和他们的交易之间）来产生推导顾客行为的规则。例如，对每一笔交易来说，数据库可能存有该交易的购买物品清单。挖掘的目标可能是要决定经常同时购买物品集合之间的关联性。例如，设 S_1 和 S_2 是经常出现在顾客交易中的物品集合，给定在一个交易中出现了 S_2，求该交易中 S_1 被发现的条件概率 $P(S_1/S_2)$。如果这个概率高，那么在其购买的交易中有 S_2 的顾客很可能是 S_1 中物品的广告对象。

更具体一些考虑这个问题。我们有一个数据库，记录有顾客的购买交易。每笔交易有一个标识和一组属性，这里就是所购买的物品。关联挖掘的第一个目的是要考察这个数据库，来确定含有某 k 种物品的交易数，是否超过数据库中总交易数的某个百分比（出现频度）。在同一个交易中一起出现的若干物品（一定个数）称为一个*物品集*（itemset），出现频度超过给定阈值的物品集称为*高频物品集*（large itemset）。一旦找到含有 k 个元素的高频物品集，以及它们在数据库中出现的频率，决定它们之间的关联规则是相当容易的。因此我们考虑的问题就集中在发现含有 k 个元素的高频物品集和它们的频率上。数据库可能放在主存中，但更经常是在磁盘上。

解决这个问题的一个简单方法是首先决定含有 1 个元素的高频物品集。从它们开始，若干含有两个元素的候选物品集就能形成（我们注意到一个物品集能被称为高频的，仅当其所有子集都是高频的），然后就可以统计它们在交易数据库中的出现频率。这就得到所有含有两个元素的高频物品集。重复这个过程，我们就能得到 k-高频物品集。这里的并发性存在于考察（$k-1$）-高频物品集以决定 k-候选集的计算中，还存在于对每个候选物品集在交易数据库中出现次数的统计中。

2.2 并行化过程

这 4 个例子（洋流模拟、Barnes-Hut、光线跟踪和数据挖掘）含有大量的并发性。借助于它们，我们在本章和下一章里讨论创建有效的并行程序的过程。具体来讲，我们将假设串行算法是给定的，而且也许已描述成了一个串行程序，我们要将其并行化。在许多情况下

(如同在这些例子里)，对一个问题的最好串行算法本身是容易并行化的；在其他一些情形，串行算法不一定表现出足够的并行性，也许需要一个完全不同的算法。并行算法设计本身是一个丰富的领域，已经超出了本书的范围。然而，无论选什么串行算法作为基础，创建一个好的并行算法都是一个不容易的过程，我们必须理解这个过程，才能够编好并行计算机的程序，才能够以并行程序为基础对体系结构进行评估。

81

在高层，并行化的工作涉及识别那些可以用并行方式来做的工作，确定如何在处理节点上分配这些工作（也许还有数据），管理必要的数据访问，通信和同步。注意，这里的“工作”包括计算、数据访问和输入/输出活动。我们的目标是获取高性能，同时保持程序设计的开销和程序对资源的需求尽量低。特别地，我们希望相对于最好的串行程序得到好的加速比。这就要求我们在处理器之间保证一个平衡的负载分布，减少进程间耗时的通信，保持低的开销，包括通信、同步、并行性管理等。

创建一个并行程序的步骤可能是由程序员来完成，也可能由某一层系统软件在程序员和体系结构之间来完成。这里的层次包括编译、运行系统和操作系统。在理想的情形下，系统软件应该允许用户用他们觉得最方便的方式写程序（例如用某种高级语言编写的串行程序或者某种更高层次的问题描述），然后自动将这些程序转换为高效的并行程序来执行。尽管在并行化编译和程序设计语言方面已有许多研究成果，但最初设想很好的自动并行化目标目前尚未达到。在今天的实践中，这个过程的主要责任还在程序员身上，尽管编译和运行系统可能会提供一些帮助。不管这些责任是如何在这些并行化实体间划分的，所涉及的问题和权衡都类似，因此我们理解它们就变得很重要了。从下面的具体讨论来说，我们在大多数情况下将假定所有的决策都是要由程序员做出的。

现在让我们以一种更加结构化的方式考察并行化过程，看一看这其中有些什么实际的步骤。每一步将与获得高性能所需考虑的若干问题有关。这些性能要素在第3章会详细讨论，这里只是简单提及。

2.2.1 程序并行化过程中的几个步骤

为理解创建一个并行程序中的步骤，让我们首先定义三个重要的概念：任务、进程和处理器。任务是程序要完成的一个工作，其内容和大小是随意的，它是并行程序所能处理的并发性最小的单元；即一个任务只能由一个处理器执行，处理器之间的并发性只能在任务之间开发。在大洋例子中，我们可以将每个计算阶段的一个格点看成是一个任务，也可以将一行格点或者网格的某个子集看成是一个任务。我们还可以认为整个网格计算是一个任务。在 Barnes-Hut 例子中，任务可能是和一个星体相关的计算；在光线跟踪例子中，任务可能是一束光或一组光的跟踪计算；在数据挖掘中，则可能是检测某个物品集在一笔交易中的出现情况。到底是什么构成一个任务，在基本串行程序中是不说明的；它是并行化工具/人的选择，尽管它通常会和串行程序结构中某些自然的工作粒度匹配（例如，循环的一次迭代）。如果一个任务所涉及的工作量小，则称它为*细粒度*（fine-grained）任务；否则称为*粗粒度*（coarse-grained）任务。

82

进程（这里我们也称为*线程*）是一个完成任务的抽象实体[⊖]。一个并行程序由许多合作

⊖ 在第1章，我们用的是符合操作系统的关于进程的定义：一个地址空间和一个或多个控制线程共享该地址空间。因此，进程和线程是有区别的。为简化本章关于并行程序设计的讨论，我们不做这样的区别，但假设一个进程只有一个控制线程。

的进程构成，每个进程完成程序中任务的一个子集。通过某种分配机制，任务被分配给进程。例如，如果将洋流网格中一行格点的计算看成是一个任务，那么一种简单的分配方法可以是让每个进程负责同等数量相邻的若干行，因此将大洋的一个截面划分成和进程一样多的水平片段。在数据挖掘中，分配可能有不同的考虑，一种可能是将数据库的不同部分分给不同的进程；另一种可能是将候选物品集合分给进程来查询数据库。进程可能需要相互通信和同步来完成它们所分配的任务。最后，进程完成其任务的方式是通过在机器的物理处理器上执行。

从并行化的观点，理解进程和处理器的区别是很重要的。处理器是物理资源，进程是将多处理器抽象（或者说虚拟化）的一种方便的形式：我们通过进程，而不是物理处理器来写并行程序；然后是将进程映射到处理器。在一次程序的执行中，进程数不一定要等于处理器数。如果进程多，一个处理器有可能要执行多个进程；如果进程少，某些处理器则要闲置。

有了这些概念，从一个串行程序得到一个并行程序的工作由4个步骤构成，如图2-3所示。

1）将计算分解成任务。

2）将任务分配给进程。

3）在进程之间协调必要的数据访问、通信和同步。

83 4）将进程映射或绑定到处理器。

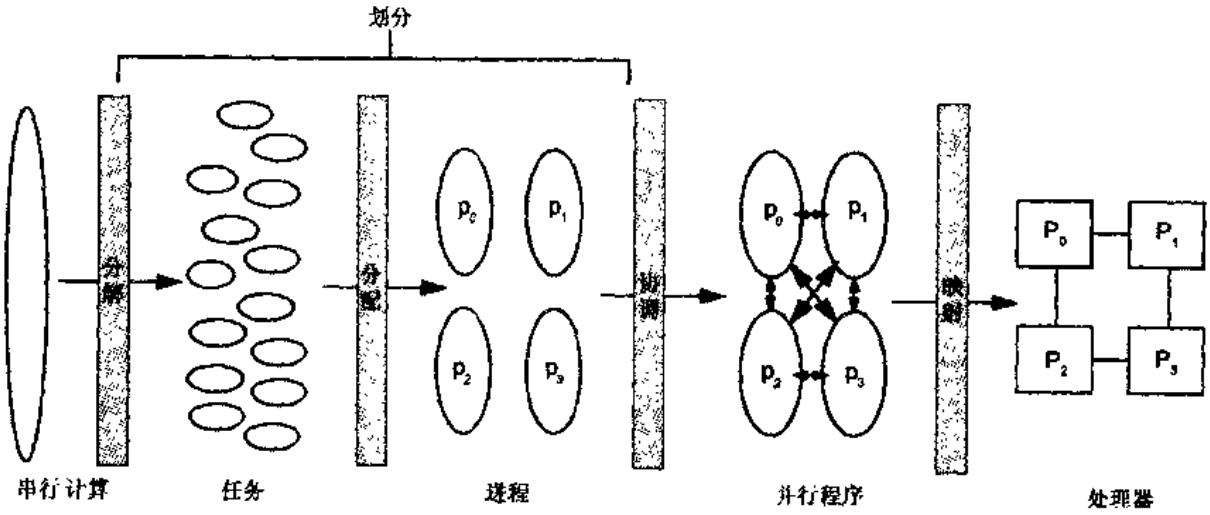


图2-3 并行化步骤以及任务、进程和处理器之间的关系。分解和分配阶段统称为划分。整合阶段协调进程之间（p）数据的访问、通信和同步，映射阶段将它们映射到物理处理器（P）

为方便起见，由于分解和分配把合作进程中的程序所做的工作进行了分割，它们一起被称为划分。下面让我们进一步考察这些步骤和相应的目标。

1．问题的分解

分解指的是将计算分为一组任务的集合。一般来说，任务可能是在程序的执行过程中动态形成的，在一个时间里的任务数可能随程序的执行变化。在一个时间上最大的任务数给出了能发挥作用进程数（因此也就是处理器数）的上限。因此，分解的主要目标就是要揭示足够的并行性，从而尽量保持所有的进程在所有时间都忙；同时要注意由此引起的任务管理开销不能太大（相对于所做的有用工作而言）。

有限的并发性是通过并行处理获得加速比的最基本限制。它不仅关系到待求解问题本身

有多少可用的并发性，还与在分解中多少并发性被揭示出来有关。可用并发性对并行处理效能的影响由 Amdahl 定律刻画，这是并行计算里不多的“定律”之一。如果一个程序执行的某些部分没有可用处理器数那么多的并发性，某些处理器将在那些部分闲置，加速比就会受到影响。从一种最简单的形式来看这个问题，考虑如果程序在单处理器上执行时间的某一部分 s 在本质上是串行的，即这一部分不可能并行。即使程序的其余部分能够得到充分的并行，在足够多的处理器上运行以至于执行时间可以忽略不计，但这一部分的串行时间依然保持。如果将串行执行时间规格化为 1，则并行程序的总执行时间将至少为 s，即加速比不会超过 $1/s$。例如，若 $s=0.2$（即程序执行的 20% 为串行），则无论用多少处理器，即使我们忽略所有其他开销，最大加速比也不能超过 1/0.2 或 5。例 2.1 给出了一个简单，但很实际的例子。 84

例 2.1 考虑某个含有两个阶段的程序。在第一阶段，一个操作独立地执行在一个 $n \times n$ 网格所有的点上，如同大洋的情形。在第二阶段，对 n^2 个网格点的值求和。如果我们有 p 个处理器，我们给每个处理器分配 n^2/p 个点，在第一阶段用 n^2/p 时间完成并行计算。在第二阶段，每个处理器可以将它的 n^2/p 个值加到全局和变量上。这种分配有什么问题？我们怎么能够揭示出更多的并发性？（暂时忽略数据访问和通信的开销）

解答： 为避免两个处理器同时修改全局和变量（见 2.3.5 节的互斥问题），全局和的累加过程就成了一个串行过程。这样，第二阶段实际上是串行的，不管 p 如何，都要花 n^2 时间。于是总时间为 $n^2/p+n^2$，而串行时间为 $2n^2$，因此加速比最多是

$$\frac{2n^2}{\frac{n^2}{p}+n^2}$$

或

$$\frac{2p}{p+1}$$

无论 p 多么大，加速最多是 2。

通过一个小小的技巧，我们可以揭示出更多的并发性。针对把每个值直接加到全局和上导致的串行化问题，考虑将计算过程的第二阶段（即求全局和）分成两个阶段。在重分后的第二阶段里，每个进程先独立地将它自己计算出来的值求和，形成一个局部和。然后，在第三阶段，进程将它们的局部和加到全局和上。重分后的第二阶段现在是全并行的；第三阶段如前一样是串行的，但只有 p 个操作，而不是 n。于是并行时间为 $n^2/p+n^2/p+p$，加速比的上界变为 $p \times 2n^2/(2n^2+p^2)$。如果 n 相对于 p 较大，这个加速比的极限几乎就是和进程数 p 成线性关系。图 2-4 指出了这种并发性的改进和影响。■

更一般地，给定一个划分和问题规模，我们可以为其构造一个*并发性态*（concurrency profile）曲线，它描绘了在应用问题中的给定的时间有多少可以并发执行的操作（或任务）。并发性态是问题、划分以及问题规模的函数。然而，它是独立于处理器数的，就好像假定处理器数无限的情形。它也独立于分配或协调。应用问题的并发性态有时可能很容易通过分析方法得到（如例 2.1 以及我们将要看到的习题 3.8 中的矩阵因子分解），但有时则可能相当不规则。例如，图 2-5 示出一个并发性态，反映一个并行事件驱动模拟对数字逻辑系统的综合。X 轴是时间，以被模拟电路的时钟周期数为单位。Y 轴是并发性的数量，在这里是电路 85

中在给定时间可被求值的逻辑门数，它是电路、输入值以及时间的函数。在不同的时钟周期上，不可预测的并发性分布范围很宽，有些周期几乎没有并发性。

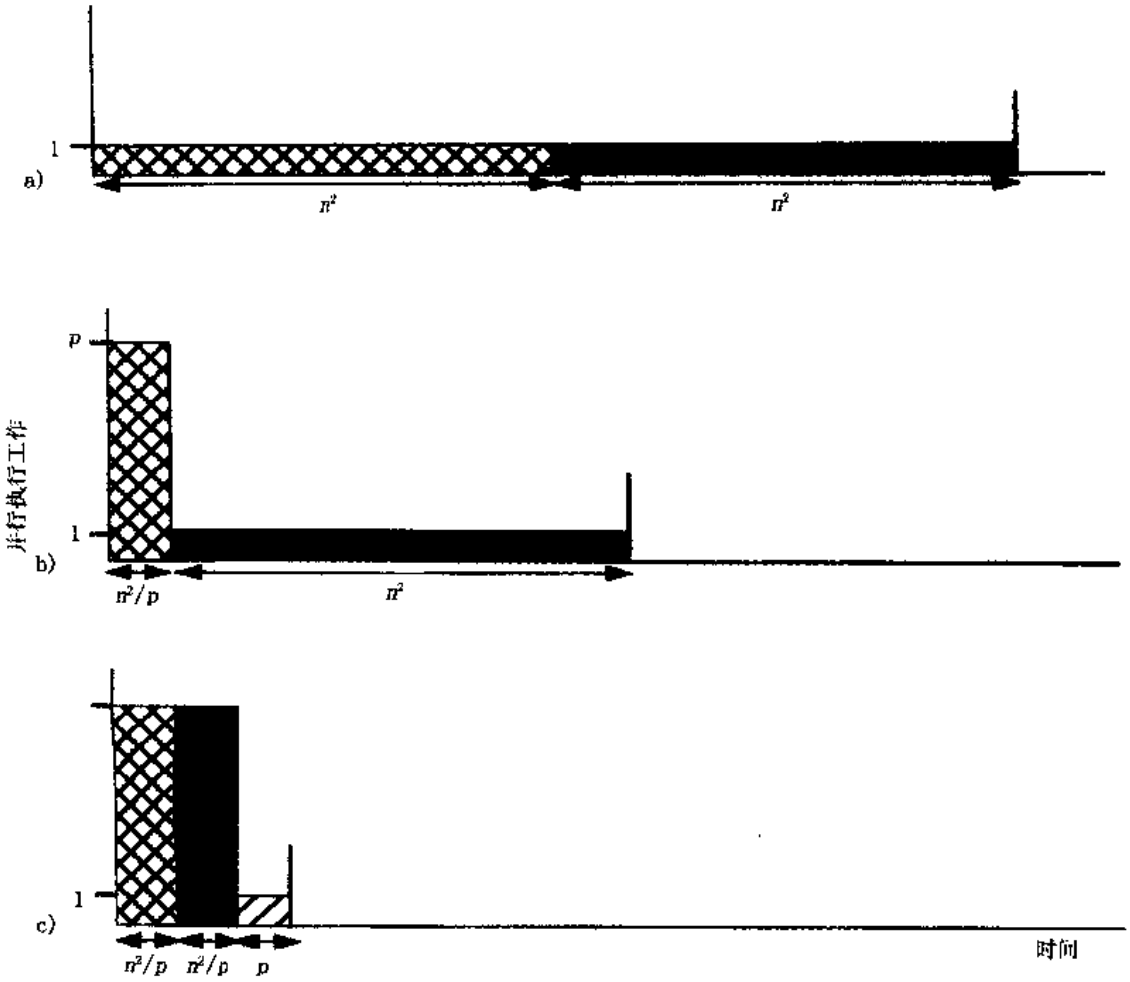


图 2-4 受限并发性影响的示例：a) 一个处理器；b) p 个处理器，n^2 操作的序列化；c) p 个处理器，p 个操作的串行化。x 轴是时间，y 轴是给定的时间里可并行的工作量（由分解表现出来的）。a) 所示的是单个处理器的性态。b) 表现的是例子中最初的情形，分两个阶段：一是全并发的，一是全串行化的。c) 所示的为改进版本，分为三个阶段：前两个是全并发的，最后一个是全串行化的，但其工作量要小得多（此时为 $O(p)$，前面是 $O(n)$）

并发性态曲线下的面积是总工作量；即所有操作或任务在单个处理器上执行所需的“时间”。水平方向的范围是在给定分解下，假定有无限多的处理器并且没有数据访问以及通信开销，最好的并行程序执行时间的下界。这个曲线下的面积除以水平范围，就给出在无限多处理器上可获得加速比的极限，这也就是该应用问题在时间上[⊖]的平均并发性。因此，Amdahl 定律的另一种形式为：

86

$$\text{加速比} \leq \frac{\text{并发性态曲线下的面积}}{\text{并发性态曲线的水平延展}}$$

对于 p 个处理器，如果 f_k 是有并发性为 k 的并发性态图中 x 轴上的点数，我们就可将 Amdahl 定律写成

⊖ 在这种分解方案下。——译者注

$$加速比(p) \leq \frac{\sum_{k=1}^{\infty} f_k k}{\sum_{k=1}^{\infty} f_k \left\lceil \frac{k}{p} \right\rceil} \tag{2-1}$$

容易看出，如果将总工作量

$$\sum_{k=1}^{\infty} f_k k$$

87

规格化到1，并且设 s 为其中串行部分所占的份额 s，那么在无穷多处理器情况下，加速比的上限是 $1/s$，随处理器数 p 变化的上限则为

$$\frac{1}{s + \frac{1-s}{p}}$$

事实上，并行处理可能带来各种不同的开销，它们不是用较多处理器就能消除的，Amdahl定律也可用来评价并行性的开销所带来的影响（不一定只是有限的并发性所带来的限制）。就我们这里来说，Amdahl定律量化了揭示足够多并发性的重要性，这是创建并行程序的第一步。

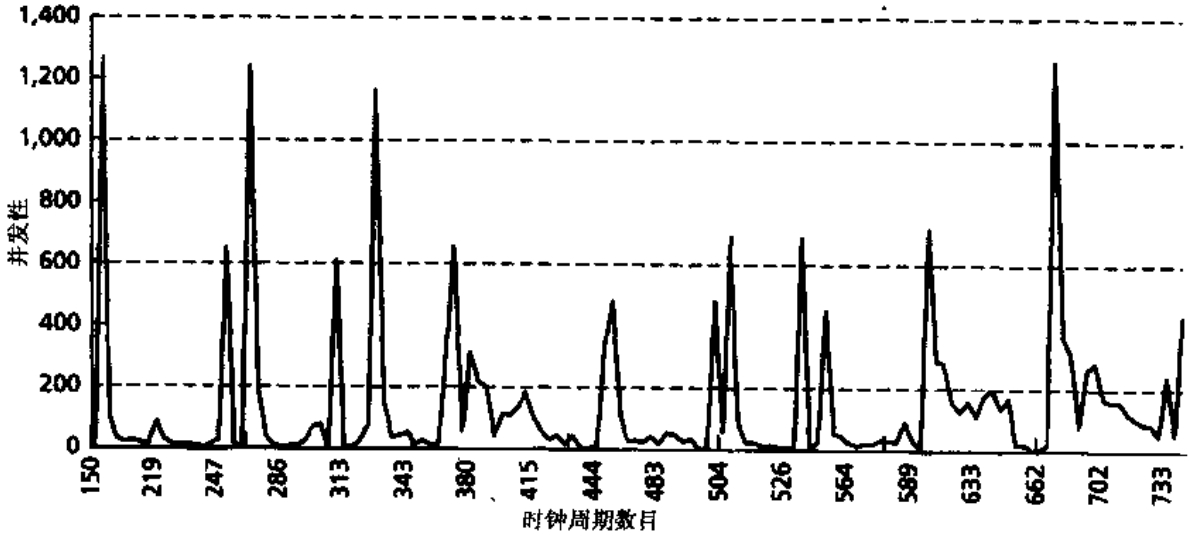


图2-5 一种时间分布，离散事件的逻辑模拟器的并发性态。被模拟的电路是一个简单的MIPS R6000微处理器。y轴表示在一个给定模拟时钟周期下可求值的逻辑元件数

2. 任务的分配

分配指的是确定一种将任务分给处理器上执行的机制。例如，在Barnes-Hut问题中哪个进程要负责计算哪些星体所受的力？在数据挖掘问题中，哪个进程要来统计哪些物品集在数据库的哪些部分的出现次数？

分配的基本性能目标是要在进程之间做负载平衡（Load balancing），要减少进程之间的通信量，还要减少运行时管理这种分配的开销。要平衡的负载包括计算、输入/输出以及数据访问和通信；在进程之间没有很好平衡的程序称为是负载不平衡的（Load imbalanced）。尤其是当进程运行在不同处理器上时，进程间通信的代价是很高的，并且如果任务分配算法很复杂，也可能在运行时产生明显开销。

同时取得这些性能目标看起来是不敢想像的。然而，许多程序本身所展现出来的特点使我们能够用一种相当结构化的方式来对其进行划分（即分解与分配）。例如，程序通常是分为阶段的，在一个阶段里的，用于分解的候选任务通常很容易识别出来，就像我们在案例研究中见到的一样。通过观察程序代码或对应用程序的某种高层理解，通常也能看出合适的任务分配方案。在不清楚的情形下，有些熟知的启发式技术常可以奏效。

如果一种计算分配在程序的开始或者是在刚刚读入和分析了输入数据后就完全确定了并且以后不再变化，则称它为*静态分配*（static assignment）或*预先确定的分配*（predetermined assignment）；如果计算任务在进程上的分配是在程序运行过程中确定的（也许是针对负载的不平衡进行调整），就称为是*动态分配*（dynamic assignment）。在第3章里，这两种情形我们都会见到。注意这里“静态”的意思和计算机科学中常用的“编译时”有所不同。编译时的分配，如果在运行时不改变，当然是静态的，但“静态”这个词在此的含义要更一般些。

分解与分配在程序并行化中是算法层次上的主要步骤。它们通常独立于下面的体系结构和程序设计模型，尽管有时在一定的系统上用某种原语的代价和复杂度可能影响分解与分配的决策。作为系统结构师，我们假定要在机器上运行的程序是基本上划分好了的。如果一个计算没有足够的并行性或者在进程间不平衡，我们无能为力；如果这个程序的通信过多，堵死了机器，我们也无能为力。作为程序员，我们通常首先注意独立于程序设计模型和结构的分解和分配，尽管在有些情况下程序设计模型和系统结构的特性会导致我们重新考虑已经作出的划分策略。

88

3. 通信与同步的引入（协调）

在协调这一步骤中，体系结构和程序设计模型以及程序语言本身都起很大的作用。为了执行分配给它们的任务，进程需要有命名和访问数据的机制以及和其他进程交换数据（通信）的机制，还要相互同步。协调指的是利用可用的机制来正确和高效地实现这些目标。同前面两个步骤相比，在这里所做的一些决定在很大程度上和程序设计模型及其基本操作的效率有关。在协调中所出现的问题包括如何组织数据结构；如何临时在进程中调度任务以开发利用数据的局部性；用显式通信还是隐式通信；用小消息还是大消息；如何准确地组织和表达进程间源于分配的通信和同步等等。程序设计语言的重要性既在于这是实际写程序的一步，还在于此时某些折中的效果在很大程度上受可用的语言机制和其开销的影响。

协调中的主要的性能目标是降低处理器之间的通信和同步开销，保持数据引用的局部性；如果一个任务为许多其他任务所依赖，则要安排它尽量早些完成，降低并行性管理的开销。系统结构师的工作是要提供适当的、效率高的基本操作，这些操作能简化协调的过程。在我们讲程序的编写过程时，有关协调的若干问题会得到进一步讨论。

4. 进程与处理器的映射

通过分解、分配和协调步骤所产生的协作的进程构成了一个能在现代系统上运行的完整的并行程序。这个程序本身有可能来控制进程到处理器的映射，否则的话，操作系统就要做这件事，从而得到一次实际的并行执行。映射通常针对具体的系统或程序设计环境。

在最简单的情形，机器中的处理器分成固定的子集（也可能整个机器就被看成是一个集合）。在一个子集上一次只执行一个程序。这称为机器的*空间共享*。程序可以将进程绑定或别（pin）到处理器上，保证它们在执行中不迁移；甚至还可以准确地控制进程和处理器的对

应关系，以保持在网络拓扑中通信的局部性。严格的空间共享机器，加上一些简单的时间共享机制，使多个应用来分享系统硬件资源的一个子集，是到目前为止大规模并行计算机系统的典型应用情况。在另一个极端，操作系统可能动态控制进程在什么时间运行在哪个处理器上，不让用户对映射有任何控制。这种方式有可能取得较好的总体资源共享和利用。每个处理器可以用普通的多道程序调度准则来管理从相同或不同程序中来的进程，进程还可能随调度安排在处理器之间移动。操作系统可能会扩展其调度规则，使之能够针对与多处理器特别有关的问题（例如，试图让一个进程尽量在同一个处理器上运行，这样这个进程能重用它在处理器高速缓存里的状态；试图在同一个时间调度来自同一个应用中的进程等）。事实上，大多数现代系统处于这两个极端之间：用户可以要系统保留某些特性，给用户程序一些映射的控制权，但操作系统也可以动态改变映射，以得到有效的资源管理。 89

在多道程序系统中的映射和相关的资源管理是一个活跃的研究领域。然而，我们这里的目标是要理解并行程序设计的最基本形式，因此为简单起见，我们假定单个并行程序对机器的资源有完全的控制。我们也假定进程数等于处理器数，且它们在程序执行的过程中都不改变。在缺省情况下，操作系统将在每个处理器上安排一个进程，没有特别的顺序。假定在执行期间没有进程在处理器之间迁移。由于这个原因，术语“进程”和“处理器”在本章的其他部分不加区别地使用。

2.2.2 计算并行和数据并行

如上所述的并行化过程集中在对计算或者操作的并行上，而不是针对数据。所分解和分配的是计算。然而，鉴于程序设计模型或性能的考虑，我们也可能要负责将数据分解和分配到进程上。事实上，许多重要的问题类型，工作和数据的分解有很强的联系，要区别它们是困难的，甚至是不必要的。大洋问题是一个好例子：每个通过大洋的截面网格表示为一个数组，我们可以将并行化看成是分解数组的数据，将不同的部分分配给不同的处理器。被分配了数据的进程然后负责和那些数据相关的计算；这称为是拥有者计算（owner compute）安排。在数据挖掘中也有类似的情形，我们可以认为数据库被分解和分配；当然，这里还有将物品集分配给进程的问题。若干语言系统，包括高性能 Fortran 标准（Koebel et al. 1994；高性能 Fortran 论坛 1993），允许程序员指定数据结构的分解和分配。计算的分配然后依照数据的分配，以拥有者计算的方式进行。然而，在计算和数据之间的区别在许多其他比较不规则的应用中是比较强的，例如我们将要讨论的 Barnes-Hut 和光线跟踪问题就是如此。由于以计算为中心的观点要更一般化，我们将保留这个观念并且考虑数据管理是协调步骤中的一部分。 90

2.2.3 并行化过程的目标

如前所述，用并行计算机的主要目标是要获得比最好的单处理器更好的性能，这种追求通常是通过加速比这个测度来体现的。前面讨论的程序并行化的4个步骤中，每一步在追求全局目标中都发挥有一定的作用，每一步都有自己的性能目标子集。它们总结在表2-1中，下一章会更详细地讨论它们。

创建一个有效的并行程序，除追求性能外，还要求评估代价。这除了机器本身值多少美元外，还必须考虑程序关于体系结构的资源需求（例如，它要用多少存储空间），还有开发出一个满意的程序所要花的气力。尽管代价和它们的影响常常要比性能更难量化，但它们是

非常重要的，一定不能忽略它们；事实上，我们经常决定用性能的损失来换取一些代价的减少。作为算法设计者，应该倾向于高性能、低资源要求的方案，同时还希望在程序设计上不需要投入过度的努力。作为系统结构师，在设计高性能系统的时候，除了追求低费用外，还要考虑对能够优化资源利用的算法的支持，考虑是否能够减少程序设计的努力。例如考虑下面两种体系结构的情况：一种随程序设计努力的增加，性能会逐步改善；另一种最终可能给出更好的性能，但即使是获得一般可接受的性能也要花极大的程序设计努力。人们可能更欢迎前者。

表 2-1 并行化进程的步骤和它们的目标

步骤	是否与体系结构有关	主要性能目标
分解	大多数情况下无关	揭示足够的并发性，但也不能太多
分配	大多数情况下无关	平衡负载，减少通信量
策划	有关	通过数据的局部性减少非本质的通信 减少处理器看得见的通信和同步开销 减少在共享资源上的串行化调度任务，尽早满足相关性要求
映射	有关	如果必要将相关的进程放到相同的处理器上在网络拓扑中开发局部性

91

我们可以应用对这个基本过程和它的目标的理解，来考察一个简单但详细的例子，针对在第 1 章介绍的三种主要现代程序设计模型（共享地址空间、消息传递和数据并行），看一看所得到的并行程序是什么样。这里，我们的注意力将集中在解释程序和编程的原语，而不考虑太多性能问题（那是第 3 章的论题）。

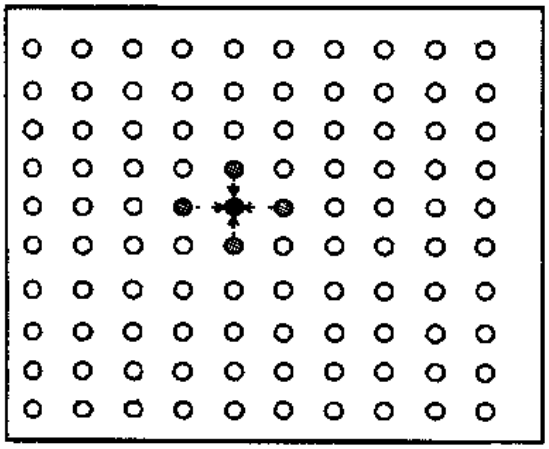
2.3 一个例子程序的并行化

在本章开始时介绍的 4 个案例分析所导致的并行程序都太复杂、太长，作为例子程序不合适。这一节里，我们给出 Ocean 问题求解核心的一个简化的版本，即它的方程求解器过程。我们用这个例子来解释在 3 种程序设计模型上如何分别实现一个并行程序。除了数据并行版本需要用一种高级数据并行语言外，这些并行程序并不是用什么特殊的语言写成的（有些语言可能提供一定的软件层次，将协调和通信的抽象对程序员屏蔽起来）。反之，它们用类似于 C 或 Pascal 的伪代码写成，用简单的并行性扩展来加强，以揭示共享地址空间或消息传递抽象必须提供的基本通信和同步原语。通过并行原语对标准串行程序设计语言进行扩充，也反映了当前实际并行程序设计的现状。

2.3.1 方程求解器的内核

方程求解器内核表示的是在一个网格上求解一个简单的偏微分方程，用的是所谓有限差分方法。它操作在一个规则的、含有 $(n+2)\times(n+2)$ 个元素的二维网格或数组上，例如海盆的一个水平截面。网格边界的行和列包含固定不变的边界值，内部的 $n\times n$ 个点由这个求解器从它们的初值开始来更新。计算进行若干次遍历。在每一次遍历中，它操作网格内部的所有 $n\times n$ 个点。对于每个点来说，它用一个加权平均值和包含它自己以及周围上下左

右四个点，来取代旧值（如图 2-6 所示）。这种更新在网格中就地完成，于是一个点的更新计算看到的是它上面点和左边点的新值，下面点和右边点的旧值。这种更新的形式称为 Gauss



更新每个内部点的表达式：

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

图 2-6 在简单方程求解器网格点的近邻更新。二维数组表示网格，按照图右边的公式，其中黑色的点 $A[i, j]$ 按照图中右边方程用它自己和周围四个点来更新

```
1.  int n;                                  /*size of matrix: (n + 2-by-n + 2) elements*/
2.  float **A, diff = 0;

3.  main()
4.  begin
5.    read(n) ;                             /*read input parameter: matrix size*/
6.    A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7.    initialize(A);                        /*initialize the matrix A somehow*/
8.    Solve (A);                            /*call the routine to solve equation*/
9.  end main

10. procedure Solve (A)                     /*solve the equation system*/
11.   float **A;                                /*A is an (n + 2)-by-(n + 2) array*/
12. begin
13.   int i, j, done = 0;
14.   float temp;
15.   while (!done) do                      /*outermost loop over sweeps*/
16.     diff = 0;                           /*initialize maximum difference to 0*/
17.     for i ← 1 to n do                   /*sweep over nonborder points of grid*/
18.       for j ← 1 to n do
19.         temp = A[i,j];                  /*save old value of element*/
20.         A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.           A[i,j+1] + A[i+1,j]); /*compute average*/
22.         diff += abs(A[i,j] - temp);
23.       end for
24.     end for
25.     if (diff/(n*n) < TOL) then done = 1;
26.   end while
27. end procedure
```

图 2-7 描述串行方程求解内核的伪码。在每一次遍历中要完成的主要计算量是 17～23 行的嵌套循环。这就是我们要并行化的（斜体字表示串行程序设计语言的关键字）

-Seidel 方法。在每一次遍历期间，这个内核也计算一个元素和它先前值的平均差。如果这个平均差小于某个预先定好的“容忍”参数，就说这个解已经收敛了，求解器就在这次遍历结束时退出。否则，它就进行下一次遍历，并进行收敛性测试。串行伪代码如图 2-7 所示。现在让我们分别针对不同的程序设计模型，一步步来将这个简单方程求解器转换成一个并行程序。对于所讨论的 3 种模型来说，分解和分配步骤基本上是相同的，因此对这两个步骤我们只是笼统地进行讨论。一旦我们进入了协调阶段，讨论将针对程序设计模型分别进行。

2.3.2 分解

有些程序的结构表现为循环或循环嵌套，对于这样的程序，一个简单的识别并发性的方法是从循环结构开始。一个个考察单独的循环或循环嵌套，看它们的迭代能不能并行完成，确定是否揭示了足够的并发性。然后可以寻找跨循环的并发性或者取一种不同的角度来做这项分解工作（如果必要的话）。让我们遵循这种基于程序结构的做法来分析图 2-7 中的程序。

从第 15 行开始，最外层 while 循环的每次迭代遍历了整个网格。由于在一次迭代中修改的数据要被下一次迭代访问，这些迭代显然是不独立的。考虑 17 ~ 24 行的循环嵌套，忽略含有 diff 的那些程序行。先看内层循环（从第 18 行开始的 j 循环）。这个循环的每次迭代读网格点 $A[i, j-1]$，其值是上一次迭代写入的。由此可看出这些迭代是顺序相关的，我们称这样的循环为*顺序循环*。由于 $i-1$ 行的元素是由第 $i-1$ 次迭代写入的，这个循环嵌套的外层循环也是顺序的。这样，对于这个循环和它们的相关性的简单分析，说明在这个程序中没有并发性存在。

通常，除了依赖程序的结构来发现并发性外，另外一种途径是回到所用的底层算法的基本依赖关系的分析中，而不管程序或循环的结构。在上面的方程求解器中，我们可以从单个网格点的粒度，来看在数据的产生和使用中发生的基本依赖关系。如前面所讨论的，由于计算是从左到右，从上到下进行的，在串行程序中计算一个网格点要用到它上面点和左边点的更新值。这种数据相关的模式如图 2-8 所示。其结果是沿着反对角线（西南到东北）的元素之间没有依赖关系，因此可被并行计算，而在下一排反对角线上的点依赖于上一排的若干点。从这个图，我们能够观察到在每一次遍历中要涉及 $O(n^2)$ 工作量，这有一种和 n 成

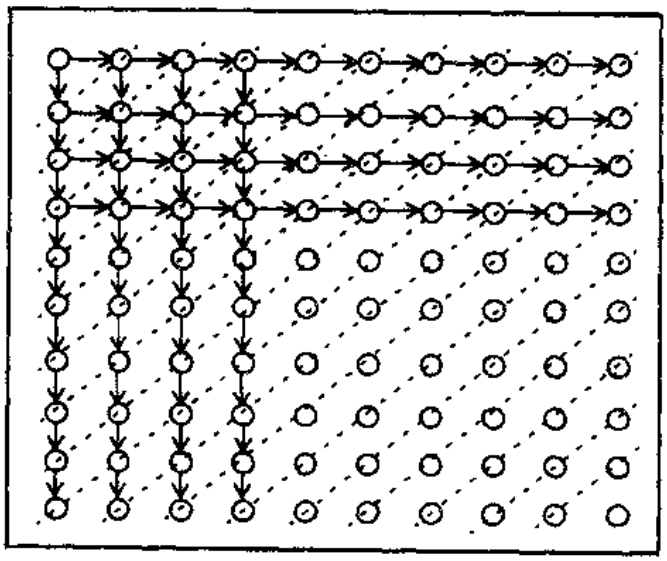


图 2-8 在 Gauss-Seidel 法中的依赖关系和并发性。带有箭头的横向和纵向的线指出数据的依赖关系；反对角线上的虚线所连接的点之间没有依赖关系，可以并行计算

一定关系的、沿着反对角线的固有并发性；也有一种和 n 成比例的、沿着对角线的顺序依赖关系。

假设我们要将网格点作为分解粒度，使得更新一个网格点是一个任务。我们能够开发在几个方面表现出来的并发性。首先，我们能够不动程序的循环结构，但加进点到点的同步以保证在一次遍历中一个点的新值被计算出来的时机先于它被下面点和右边点的引用。这样，该顺序程序的不同的循环嵌套，甚至不同的遍历都有可能在不同的元素上同时进行，只要元素级依赖关系不被违反。但是这种在网格点级别的同步开销可能太高。再者，我们可能改变循环结构：第一个 for 循环（第 17 行）可以是作用在反对角线的，而内层 for 循环作用在一条反对角线内的元素上。内层循环于是可被完全并行执行，使同步在外层 for 循环的迭代之间发生（以保守地保持跨越反对角线之间的依赖关系）。对这两种情况，通信的安排将是相当不同的；如果通信是显式消息的话，情况会更加明显。然而，这种做法也有问题。全局同步仍然很频繁——一条反对角线一次。除此以外，在并行内层循环的迭代次数随着不同的外层循环迭代随反对角线长短的变化而变化。由于同步的频繁、负载的不平衡以及程序设计的复杂，上面两种方法在现代系统结构中用得都不多。

第三个，也是最常用的办法是基于对问题本身的知识，不一定遵守顺序程序本身的依赖关系。对于 Gauss-Seidel 方法来说，串行算法中网格点更新的次序（即从左到右，从上到下）实际上不是很重要的，它不过是一种对于串行程序设计来说比较方便的一种次序而已。由于 Gauss-Seidel 方法不是一种精确解法（不同于高斯消去法），它只是要求迭代到收敛，我们可以用不同的次序来更新网格点，只要我们足够频繁地用到更新了的值[1]。这样一种常用的序称为红-黑序。其基本思想是将网格点分为交替的红点和黑点，如同棋盘那样（见图 2-9），相邻的点有不同的颜色。由于每个点只读它的四个近邻，为计算一个给定的红点，我们不需要任何其他红点的新值；我们只需要它上面和左边的黑点的新值（在标准遍历中），计算黑点的情形类似。因此我们可以将网格的遍历过程分为两个阶段，首先计算所有的红点，然后计算所有的黑点。在每一阶段内部，网格点之间没有依赖关系存在，于是我们可以并行计算所有 $n^2/2$ 个红点，做一次全局同步，然后并行计算所有 $n^2/2$ 个黑点。由于计算一个黑点并不需要所有红点都被计算出来，全局同步是保守的做法，可以被格点级的点到点同步来取代；不过全局同步做起来要方便得多。

92 ~ 95

由于红-黑序不同于我们最初的顺序序，它达到收敛的遍历次数可能会多，也可能会少。它也可能产生不同的网格点终值（尽管也在收敛范围）。在一次遍历中，红点更新时看不到任何黑点的新值，黑点却看得到它所有相邻红点的新值（从第一个阶段产生的），不仅是左边和上面的点。和老的序比起来，这种新的序的优劣取决于问题本身[2]。由于在一个阶段里没有依赖关系出现，红-黑序还有一个优点，即产生的值和收敛特性独立于参与计算的处理器数。如果串行程序本身也用红-黑序，那么并行化完全不会改变结果或者收敛特性，从而使得并行程序具有确定性。

红-黑序所产生的内核代码较长，不便于用来解释并行程序设计。让我们来考察一个简单但仍然常用的异步方法，它不需要将格点划分为红点和黑点。这个方法从根本上忽略在一

---

[1] 即使我们不用在当前遍历（即 while 循环迭代）中更新了的值，而是用在上一次遍历时产生的，系统也会收敛，只是慢些。这称为是 Jacobi 迭代法，不同于 Gauss-Seidel 迭代法。

[2] 即初始数据。——译者注

次遍历中格点之间的依赖关系。同前面的做法一样全局同步只是用于相继的遍历之间，但一个进程在一次遍历中不是从上到下，从左到右地依次更新格点。取而代之，在一次遍历内进程更新所有分配给它的格点值，访问有关的相邻点，不管那些点是否在本次遍历中已被相应进程更新。如果只有一个进程，这种方法就是最初的顺序。如果有多个进程参与，这个次序就是不可预测的；它取决于格点在进程之间的分配、所用到的进程数以及在运行时不同进程向前推进的相对速度。这个执行不再是确定性的，为达到收敛所需的遍历数可能就要取决于所参与计算的处理器数。然而，对于多数合理的分配来说，遍历数的变化将不会很大。

96

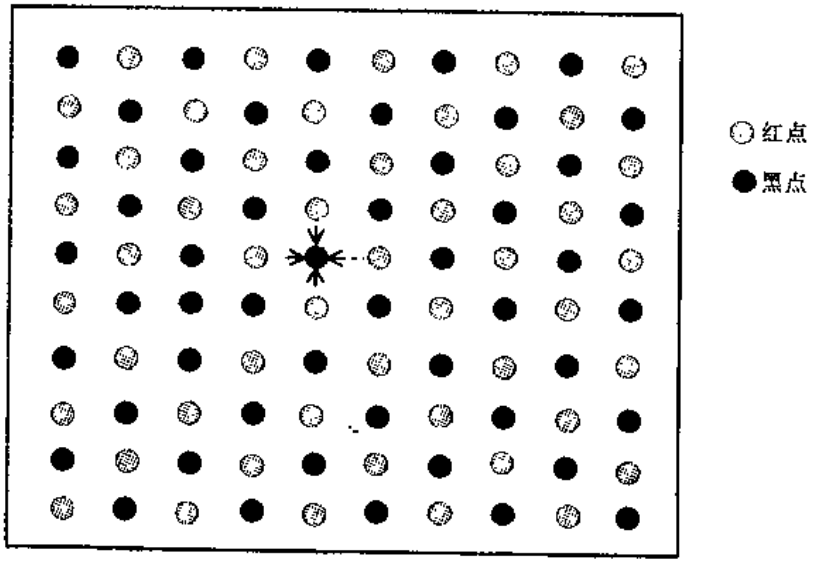


图 2-9 方程求解器的红黑序。在网格上的遍历被分为两次子遍历：第一次计算所有的红点，第二次计算所有的黑点。由于红点只依赖于黑点，反之亦然，在一次子遍历中就没有依赖关系了

如果决定将分解做到内层循环的迭代（即格点粒度），改写图 2-7 中的 15～26 行可得相应程序。图 2-10 用黑体指示了代码的变化：所做的是用 for_all 替换关键字 for。for_all循环告诉下面的硬件/软件系统，循环中的所有迭代都可并行执行，不要担心相关性问题，但它没讲分配。一个两层都是 for_all 的循环嵌套意味着所有迭代（这里是 $n \times n$ 次）都可并行执行。系统可以以任何方式来分配和安排并行化；程序对此没有任何限制。它

```
15. while (!done) do                          /*a sequential loop*/
16.   diff = 0;
17.   for_all i ← 1 to n do                   /*a parallel loop nest*/
18.     for_all j ← 1 to n do
19.       temp = A[i,j];
20.       A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.         A[i,j+1] + A[i+1,j]);
22.       diff += abs(A[i,j] - temp);
23.     end for_all
24.   end for_all
25.   if (diff/(n*n) < TOL) then done = 1;
26. end while
```

图 2-10 并行方程求解器内核，网格点分解，没有显式的分配。由于两个 for 循环都用 for_all 并行表示，分解的粒度就是单个网格点。除了这个变化外，代码的其他部分和串行代码一样

所要求的只是在 for_all 循环嵌套后有一个隐含的全局同步。

事实上，我们对这个计算的分解不仅可以落实到内层循环的单个迭代上，还可以落实到一组迭代上。注意到这里的计算分解和网格分解是相当对应的。假设我们要按行来分解网格点，从而一行上的工作成为一个不可分的任务，必须分配到同一个进程中。我们表达这种想法的方式可以是将 for_all 改回 for，从而将第 18 行的内层循环改成顺序循环，但保持第 17 行的循环为并行的 for_all。由此表现出来的并行性，即并发性的程度，就从问题本身所具有的 n^2 减到了 n：本来是 n^2 个独立的任务，每个具有一个时间单位；现在成了 n 个独立的任务，每个具有 n 个时间单位。如果每个任务在不同的处理器上执行，对于 n 个点，我们将有大约 $2n$ 字的通信量（访问那些由其他进程计算的格点），这导致通信与计算比为 $O(1)$。

2.3.3 分配

用基于行的分解，我们来看怎么将网格的行分配到进程上。最简单的做法是静态（事先确定的）分配，其中每个进程负责一个连续行块，如图 2-11 所示。第 i 行分给进程

$$\left\lfloor \frac{i}{p} \right\rfloor$$

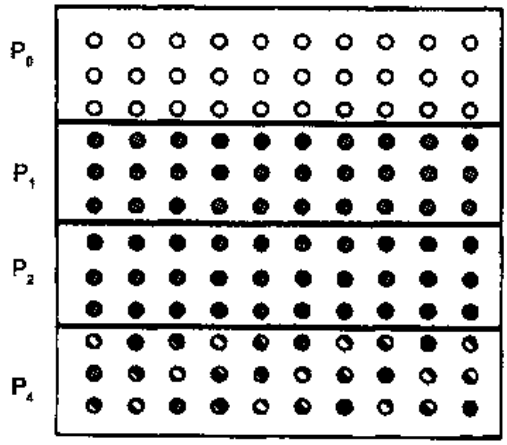


图 2-11 并行方程求解器的一种简单分配方案。每个处理器被分给网格上若干连续的、等量的行。在每次遍历中，处理器完成分给它的行上的元素的更新。图中只显示了那些在一次遍历中要更新的内部点

其中 p 是进程个数。这种方式称为块方式。另外一种静态分配方式是循环分配方式，网格中的行交替地在进程间分配（第 i，$i+p$ 行分配给进程 i，等等）。我们也可以考虑一种动态的分配方案，其中每个进程在完成一行的计算后去动态获取下一个尚未计算的行，这样就不可能事先确定那个进程计算哪些行。下面的讨论将基于静态块分配。由于每一行的工作量是统一的，只要网格内部的行数能被进程数整除，这种简单的划分就具有很好的负载平衡特性。我们看到，通过使任务变大，这种静态分配进一步将并行性（或并发度）从 n 降到了 p；通过将相邻的行分给同一个进程（块分配），从而减少了通信。通信和计算比现在是

$$O\left(\frac{p}{n}\right)$$

考察了分解和分配后，我们就要进入协调阶段了。这要求我们落实到程序设计模型。我们从一个高层的数据并行模型开始，然后看看两个主要的程序设计模型、共享地址空间和显式消息传递。数据并行模型和其他一些高层模型都可能编译到这两种模型上。

2.3.4 在数据并行模型下的协调

数据并行程序设计模型适合于上述方程求解器的内核。这是由于所涉及的计算可以自然地看成是一个单控制线索，在一个大数组数据结构上进行全局变换[⊖]（Hillis 1985；Hillis and Steele 1986）。从划分的角度看，面向计算和面向数据在相当程度上是等价的，一种简单的数据分解和分配就可以在进程之间得到很好的负载平衡，这种分配（划分）在形状上是很规则的，可以由简单表达式来描述。这种数据并行方程求解器的伪代码如图2-12所示。我们假设全局的变量声明（所有过程之外的）描述的是共享数据，所有其他数据（例如，过程栈上的数据）是私有于一个进程的。对于数组A那样的共享数据，存储分配通过G_MALLOC

```
1.   int n, nprocs;                /*grid size (n + 2-by-n + 2) and number of processes*/
2.   float **A, diff = 0;

3.   main()
4.   begin
5.     read(n); read(nprocs);; /*read input grid size and number of processes*/
6.     A ← G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
7.     initialize(A);            /*initialize the matrix A somehow*/
8.     Solve (A);                /*call the routine to solve equation*/
9.   end main

10.  procedure Solve(A)          /*solve the equation system*/
11.    float **A;                /*A is an (n + 2-by-n + 2) array*/
12.  begin
13.    int i, j, done = 0;
14.    float mydiff = 0, temp;
14a.   DECOMP A[BLOCK,*, nprocs];
15.    while (!done) do            /*outermost loop over sweeps*/
16.      mydiff = 0;               /*initialize maximum difference to 0*/
17.      for_all i ← 1 to n do  /*sweep over non-border points of grid*/
18.        for_all j ← 1 to n do
19.          temp = A[i,j];        /*save old value of element*/
20.          A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.            A[i,j+1] + A[i+1,j]);      /*compute average*/
22.          mydiff += abs(A[i,j] - temp);
23.        end for_all
24.      end for_all
24a.     REDUCE (mydiff, diff, ADD);
25.      if (diff/(n*n) < TOL) then done = 1;
26.    end while
27.  end procedure
```

图 2-12 描述数据并行方程求解器的伪码。和串行代码的区别见黑体字所示。斜黑体字所表示的是为获得并行性设计的成分。分解依然是落实到了单个元素，如嵌套的 for_all 循环所示。分配由 DECOMP 语句所指，将数据分成连续行块状（沿列方向做了划分，行方向上没划分）。REDUCE 语句将本地计算好的 mydiffs 求和到全局 diff 中。while 循环依然是串行的

⊖ 即对其中的元素做相同或类似的计算。——译者注

(全局 malloc) 动态实现，而不是一般的 malloc。G_MALLOC 在堆空间的一个共享区域为数据分配内存，这个空间可被任何进程访问和修改。除此以外，和串行程序相比主要不同（如黑体所示）在于 for_all 循环的使用，而不是 for 循环；还有 DECOMP 语句；每个进程都有的私有 mydiff 变量；以及 REDUCE 语句等。

我们已经看到了 for_all 循环说明迭代可被并行完成。除执行主控的进程外，并行进程隐含在数据并行模型中，且仅仅在这些并行循环时才活跃。DECOMP 语句有两个目的。首先，它说明了迭代在进程之间的分配（DECOMP 在这个意义下是一个不合适的字）。这里，它是［BLOCK，*，nprocs］分配，即第一维（行）在 nprocs 进程之间被划分为连续块，第二维不做划分。如果是［CYCLIC，*，nprocs］则意味着在 nprocs 进程之间对行做循环或交替划分，而［BLOCK，BLOCK，nprocs］则是做两维上的块划分，［*，CYCLIC，nprocs］则意味着对列做交替式划分。DECOMP 的第二个相关的目的是，它也说明了网格数据应该如何分配到分布存储机器的存储器中。（在当前多数数据并行语言中，依照拥有者计算原则，这被限制为和计算分配相同。对这个例子来说，数据分配和计算分配对应得相当好）。mydiff 变量用来让每个进程首先独立地计算它所拥有格点差值的和。然后，REDUCE 语句指示系统将所有部分 mydiff 值加到共享的 diff 变量里。如例 2.1 所讨论过的，这增加了并发性。REDUCE 操作实现一种规约，其中许多进程（在全局规约中是所有进程）一起在逻辑上共享的数据上完成满足结合律的操作（如相加、取最大值等）。结合律蕴含着操作的次序无关。这里进行的计算机浮点操作，由于舍入误差的累积和操作的次序有关，严格来讲是不满足结合律的。然而，差别不明显，于是通常忽略它们；尤其是在迭代计算的情形，本来结果就是近似的。规约操作可以用一种最适合底层体系结构方式的库来实现。

97 ~ 100

数据并行程序设计模型适合于说明在大型数组数据上进行规则计算时的划分和数据的分布（例如方程求解器内核或 Ocean 应用），但对较不规则的应用，适应性不一定总是很好；特别是有些应用，通信模式或者任务之间的工作分配随时间不可预测地变化（例如，考虑 Barnes-Hut 中的星体或 Raytrace 中的光线，如果给每个进程都分配相同数量的光线，会导致严重的负载不平衡）。下面讲更灵活的、层次较低的程序设计模型，其中进程是显式的，它们各自有自己的控制线程，当需要的时候会相互通信。

2.3.5 在共享地址空间模型下的协调

在共享地址空间，我们可以直接了当地将矩阵 A 声明成一个单一的共享数组——如同我们在数据并行模型所做的那样——进程可以通过读入和写出操作，用和串行程序同样的数组下标引用它所需的部分；通信随需要隐含产生。由于此时进程是显式的，我们需要机制来创建它们，并通过同步来协调它们，还要控制负载在进程之间的分配。我们用的原语是典型的低层程序设计环境，如 parmacs (Boyle et al.1987)，综合起来如表 2-2 所示。

101

表 2-2 关键共享地址空间的原语

原语名称	语 法	功 能
CREATE	CREATE(p,proc,args)	创建 p 个进程，用参数 args 从过程 proc 开始执行
G_MALLOC	G_MALLOC(size)	分配 size 字节的共享数据
LOCK	LOCK(name)	获得互斥访问权
UNLOCK	UNLOCK(name)	释放互斥访问权

（续）

原语名称	语　法	功　能
BARRIER	BARRIER(name, number)	在 number 个进程之间做全局同步：只有所有 number 个进程都达到后，才能通过 BARRIER
WAIT_FOR_END	WAIT_FOR_END(number)	等待 number 个进程一起终止
wait for flag	while (!flag); or WAIT(flag)	等待 flag 的出现（踏步或阻塞）；用于点对点事件同步
set flag	flag = 1; or SIGNAL(flag)	设置 flag；唤醒踏步或阻塞等在 flag 上的进程

在共享地址空间里的并行方程求解器的伪代码如图 2-13 所示。和并行性有关的特别原语如黑体所示。通常它们用库或宏来实现，每一个扩展成若干条指令来达到其目的。尽管 Solve 过程的代码和串行版本极端相似，我们还是有必要一步步进行讨论。

```
1.     int n, nprocs;          /*matrix dimension and number of processors to be used*/
2a.    float **A, diff;        /*A is global (shared) array representing the grid*/
                               /*diff is global (shared) maximum difference in current
                                 sweep*/
2b.    LOCKDEC(diff_lock);     /*declaration of lock to enforce mutual exclusion*/
2c.    BARDEC (bar1);          /*barrier declaration for global synchronization between
                                 sweeps*/

3.   main()
4.   begin
5.       read(n); read(nprocs);   /*read input matrix size and number of processes*/
6.       A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7.       initialize(A);           /*initialize A in an unspecified way*/
8a.      CREATE (nprocs-1, Solve, A);
8.       Solve(A);                /*main process becomes a worker too*/
8b.      WAIT_FOR_END (nprocs-1); /*wait for all child processes created to terminate*/
9.   end main

10.  procedure Solve(A)
11.    float **A;                          /*A is entire n+2-by-n+2 shared array,
                                             as in the sequential program*/
12.  begin
13.    int i,j, pid, done = 0;
14.    float temp, mydiff = 0;             /*private variables*/
14a.   int mymin = 1 + (pid * n/nprocs);   /*assume that n is exactly divisible by*/
14b.   int mymax = mymin + n/nprocs - 1    /*nprocs for simplicity here*/

15.    while (!done) do            /*outer loop over all diagonal elements*/
16.      mydiff = diff = 0;        /*set global diff to 0 (okay for all to do it)*/
```

图 2-13　在共享地址空间描述并行方程求解器的伪码。带有字母的语句行号表示在串行代码中没有的语句。语句行号的选取尽量和串行代码中的匹配。数据结构的设计不需要在串行程序的基础上有什么改变。计算进程通过 CREATE 调用创建，主进程在程序的结尾处用 WAIT_FOR_END 调用等待它们的完成。由于内层循环没有改动，分解是成行的；外层循环将数据行分给了进程。栅障用来将不同的遍历隔离开来（还将收敛测试和对 diff 的进一步更新隔离开来），锁用来保证对全局 diff 变量的互斥访问

```
16a.     BARRIER(bar1, nprocs);      /*ensure all reach here before anyone modifies diff*/
17.      for i ← mymin to mymax do /*for each of my rows*/
18.        for j ← 1 to n do        /*for all nonborder elements in that row*/
19.          temp = A[i,j];
20.          A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.            A[i,j+1] + A[i+1,j]);
22.          mydiff += abs(A[i,j] - temp);
23.        endfor
24.      endfor
25a.     LOCK(diff_lock);            /*update global diff if necessary*/
25b.     diff += mydiff;
25c.     UNLOCK(diff_lock);
25d.     BARRIER(bar1, nprocs);      /*ensure all reach here before checking if done*/
25e.     if (diff/(n*n) < TOL) then done = 1;      /*check convergence; all get
                                                     same answer*/
25f.     BARRIER(bar1, nprocs);
26.    endwhile
27.  end procedure
```

图2-13（续）

操作系统首先启动一个进程，从称为 main 的过程开始来执行这个程序。让我们称它为主进程。它读入输入数据，指定网格 A 的大小（记得输入 n 表示一个 $(n+2)\times(n+2)$ 网格，其中 $n\times n$ 点由求解器来更新）。它然后将网格 A 作为一个二维数组，在共享地址空间中，用 G_MALLOC 调用获得（见2.3.4节）存储空间，并将它初始化。对于那些没有动态在堆上分配空间的数据，对一个进程来讲哪些是共享的、哪些是私有的，不同的系统有不同的假设。让我们作如同先前数据并行例子中同样的假设。在所有过程外声明的数据，例如图2-13里的 nprocs 和 n 是共享的。在过程堆栈上的数据（例如 mymin，mymax，mydiff，temp，i，j）对于对应于该过程的进程是私有的，由常规 malloc 调用分配的数据也是私有的。（在有些情形，还可能显式说明私有数据，但本程序没有这种情况）。

分配了数据空间，初始化了网格，程序就准备开始对系统进行求解。它创建（nprocs－1）个“工作者”进程，它们在 Solve 过程开始执行。主进程然后也调用 Solve 过程，这样所有 nprocs 进程并行进入这个过程，它们是平等的伙伴。所有创建的进程执行相同的代码映像，直到它们从程序出来并终止。也就是说，我们用的是一种结构化的，单程序多数据（SPMD）的程序设计风格。但这不意味着它们完全以步步锁定的方式向前推进，或者执行同样的指令（如在单指令多数据（SIMD）模型那样）。一般来说，它们可能循着不同的控制路径在程序中通过。在进程之间的任务分配以及进程访问什么数据，是通过几个私有变量来实现的；对于不同的进程，那些变量取不同的值（例如，mymin 和 mymax）。对循环控制变量的处理也是控制计算任务分配的方式。例如，假定每一个进程在被创建时会在自己的私有地址空间自动地获得一个在0和 nprocs－1 之间的惟一进程标识（pid）[⊖]，然后它用这个 pid（见14a～14b行）来确定分配给它的行。进程通过调用同步原语来进行同步，下面将会马上讨论。

⊖ 通常，在进程开始时用一个调用来获得 pid。——译者注

为简单起见，我们假设内部行的数目是进程数 nprocs 的整倍数，这样每个进程所分得的行数相等。每个进程用它的私有变量 mymin 和 mymax 计算出它所分得数据块的第一行和最后一行的下标。然后进入实际求解方程的循环。

最外层 while 循环（第 15 行）仍然是作用于相继的网格遍历之上的。尽管这个循环的迭代是顺序进行的，但其中每次迭代或遍历本身可由所有进程来并行执行。是否要执行下一次遍历由每个进程或控制线程分别决定（通过设置 done 变量，判断 while (! done) 条件）。在这里安排的是让每一个进程都做同样的决定：这里完成的冗余工作和在处理器之间交流一个完成标志或 diff 值相比是很小的。

完成实际数据更新的代码（19～22 行）基本上和串行程序完全一样。除了循环控制语句的界以外（它控制任务的分配），惟一的区别是每个进程维护一个私有变量 mydiff。如同数据并行的例子，这个私有变量记录分配给它的格点的新旧值的总差别。它在本次遍历结束时加到共享的 diff 变量上，而不是让每个点的差值直接加到上面去。除了在 2.2.1 节（例 2.1）讨论的串行化和并发性原因外，所有进程重复修改和读同一个共享变量还要引起大量昂贵的通信，因此我们不应该让每个格点都这样做。

在程序的剩下部分，有意思的方面是同步，包括互斥和事件同步。首先，不同进程对共享变量的累加应该是互斥的。考虑一个处理器要将它的 mydiff 变量（假定在寄存器 r2 中）加到共享变量 diff 上要执行的指令序列（即执行源程序的语句 diff += mydiff）：

将 diff 装载到寄存器 r1 中；

将寄存器 r2 加到 r1 上；

将寄存器 r1 的值存到 diff。

假设最初变量 diff 的值为 0，每个进程的 mydiff 值是 1。在两个进程执行了这段代码后，我们期望 diff 的值为 2。然而，如果这两个进程以下面的方式交叉地执行了它们的操作，它却可能是 1。

P_1	P_2
r1←diff {P_1 在 n 中得到值 0}	
	r1←diff {P_2 也得到值 0}
r1←r1 + r2 {P_1 将 r1 设为 1}	r1←r1 + r2 {P_2 将 r1 设为 1}
diff←r1 {P_1 将 1 存储到 diff 中}	
	diff←r1 {P_2 也将 1 存储到 diff 中}

这并不是我们希望发生的。这里的问题在于一个进程（此处为 P_2）读一个逻辑上共享变量的 diff 值，发生在另一个进程（P_1）读 diff 并将 diff 写回之间。为了防止这种交叉的操作，我们希望不同进程的操作序列的执行相互具有原子性（即做到互斥）。这样一个要原子性执行的操作序列称为是临界区：一旦一个进程开始执行其中的第一条指令，任何其他进程都不能执行其相应临界区中的指令，直到前面的进程完成了它临界区中的最后一条指令。在第 25b 行左右的 LOCK-UNLOCK 对就是要保证由 diff += mydiff 构成的临界区的互斥。

102～105

像 diff_lock 这样的锁可以看作是一种共享的标记，代表一个排它的权利。通过 LOCK 原语获得这把锁给予一个进程执行其临界区的权利。具有锁的进程，在完成了它的临界区后，通过执行 UNLOCK 命令来释放它。此时取决于具体实现，该锁就可以被另外的进程获取，或由系统将它赋予某个进程。LOCK 和 UNLOCK 原语的实现必须保证互斥的体现。锁的

开销是很大的，如果多个进程试图在同一个时间访问一把锁，可能发生竞争和串行化。我们的 LOCK 原语用一把锁的名字作为参数，这样的好处是允许我们用不同的锁来保护不相关的临界区，减少竞争和串行化。

一旦一个进程将它的 mydiff 加到全局的 diff 后，它就应该等待，直到所有其他进程都完成了这个操作，diff 的值的确是有关所有各点的总的差值。这就要求全局的事件同步，这里用 BARRIER 实现。栅障操作取栅障名和同步所涉及的进程数为参数，所有进程都要执行。当一个进程调用栅障时，它登记它已到达程序这一点的事实；它将不再推进，直到所有进程都执行了相应的栅障操作。于是，BARRIER（name，p）的语义如下：等到所有 p 个进程都到达此处，然后继续推进。关于为什么需要另外两个栅障，习题 2.6 给出答案。

栅障通常用于将程序中的不同计算阶段分隔开来。例如，在 Barnes-Hut 星系模拟中，我们在每一时间步结束更新星体的位置后，在用这些位置做引力计算之前插入一个栅障，在数据挖掘中，我们可以在统计候选物品集合和用大物品集来产生狭义候选表之间用栅障。由于栅障实现的是全部对全部事件同步，它们通常是一个维持相互关系的保守方式；通常，并不是所有在栅障后的操作（或进程）都需要等待所有在栅障前的操作都完成后才能进行。在进程对或组之间更特定的事件同步可使得某些进程较早地通过它们的同步操作；然而，从程序设计的观点来看，和根据实际的依赖关系在进程之间用点到点同步相比，用一个栅障要方便得多。

当需要点到点同步的时候，在共享地址空间实现的一种方式是通过在信号灯上做等待和发信号操作，如同在操作系统中我们所熟悉的那样。在并行程序中更常用的方式是用普通共享变量作为标志来进行事件同步，如图 2-14 所示。由于 P_1 只是简单的在一个 while 循环兜圈子，等待标志变量置 1，使处理器在这个期间忙，我们称此为*踏步等待*（spin waiting）或*忙等待*（busy-waiting）。记得在信号灯的情形，等待进程是不原地踏步的，因此也就不消耗处理器资源；它将自己阻塞（挂起），等待另一个进程给出信号灯信号。 106

P_1	P_2
	A = 1;
	b: flag = 1;
a: while (flag is 0) do nothing;	
print A;	

图 2-14 用标志变量实现点到点事件同步。假设我们要做到在进程 P_2 到达程序中的某一点 *b* 之前，进程 P_1 不应通过程序中的另一点 *a*。假定变量 flag 和 *A* 在进程到达这种情况之前都初始化为 0。如果 P_1 到达语句 *a* 时 P_2 已经执行了 *b*，P_1 就能够通过 *a*。如果 P_2 还没有执行 *b*，P_1 就闲呆在 while 循环上，直到 P_2 到达 *b* 并将 flag 置 1，P_1 才能从循环上出来继续执行。如果我们假设 P_2 的写为 P_1 可见的序和 P_2 发出它们的序相同，这种同步就能保证 P_1 打印 *A* 时的值为 1

在进程子集的事件同步情形，或称*组事件同步*（group event synchroni zation），一个或多个进程可能要等待某个事件，一个或多个进程可能通知它们该事件的出现。组事件同步的实现可用普通共享变量作为标志或在进程子集之间用栅障。

返回图 2-13 的方程求解器，一旦某个进程通过了栅障，它就读 diff 的值，考察在所有格点上的平均差（diff / (n × n)）是否小于给定误差范围，来决定收敛性。如果收敛，它就置 done 标志，从 while 循环出去；否则，它就继续下一次遍历。

最后，在程序的结束部分（行 8b）由主进程调用的 WAIT_FOR_END 是一种多对一同步的特殊形式。通过它，主进程等待它所创建的工作者进程的结束。其他进程不调用 WAIT_FOR_END，只是隐式地参与这个同步，通过离开 Solve 过程（它们进入程序的入口点）时终止自己。

综合起来，对这个简单的方程求解器来说，在共享地址空间的并行程序在结构上和串行程序相差无几。在控制流的主要区别通过某些循环控制变量的改变来实现。其他的差别在于进程的创建，任务在它们之间的划分以及通过简单且通用的原语的同步。计算循环体基本上没有变化，主要的数据结构和对它们的引用也没什么变化。给定分解、分配和同步的策略，在这个例子中，插入必要的原语，作必要的修改以生成一个正确的并行程序是相当机械的。在分解和分配上的变化也容易体现，如例 2.2 所示。尽管许多简单程序在共享地址空间下有上述这些易于并行化的特点，我们在后面将看到，如果要追求更高的并行性能，或者是考虑更复杂的并行程序时，在串行程序上做更实质性的改变是需要的。

107

例 2.2 如果我们保留按行的分解但将行在进程之间的分配改为循环交叉方式的，同样是在共享地址空间，图 2-13 所示的方程求解器程序应该如何修改？

解答： 图 2-15 给出了相关的代码。所有的变化只是在第 17 行的循环控制表达式。同样的全局数据结构，同样的下标引用，其他部分保留原样。■

```
17. for i ← pid+1 to n by nprocs do        /*for my interleaved set of rows*/
18.   for j ← 1 to n do                     /*for all elements in that row*/
19.     temp = A[i,j];
20.     A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.       A[i,j+1] + A[i+1,j]);
22.     mydiff += abs(A[i,j] - temp);
23.   endfor
24. endfor
```

图 2-15 在共享地址空间中，基于行的求解器的循环分配方案。和图 2-13 所示的块划分相比，改变的仅仅是第 17 行的语句。数据结构和访问方式不需要有变化

2.3.6 在消息传递模型下的协调

我们现在考察并行求解器在私有地址空间之间进行显式消息传递的一种实现，分解和分配如前不变。由于没有了共享地址空间，我们不能简单地声明矩阵 A 是共享的，从而使进程能够像在顺序程序那样引用它的各个部分。这里，逻辑数据结构 A 必须由一组较小的，属于不同进程的数据结构来表示。这些数据结构在合作进程的私有地址空间中进行存储分配，和工作的分配保持某种一致性。特别地，分配有数组 A 中的若干行的进程将那些行作为一个数组分配在自己的私有地址空间中。

用于消息传递程序设计的一组简单原语如表 2-3 所示。图 2-16 所示的消息传递程序用到了其中一些原语，在结构上很像图 2-13 的共享地址空间程序（在 3.6 节，更复杂的程序将进一步地揭示区别）。这里，也有一个由操作系统启动的主进程，这个主进程创建 nprocs - 1 个其他进程和它协同工作。我们也假设每一个创建的进程自动地获取一个 0 到 nprocs - 1 之间的进程标识符（pid），CREATE 调用自动地将程序的输入参数（n 和 nprocs）传到每

108

个进程的地址空间[⊖]。Solve 过程的最外层循环（第 15 行）仍然是在网格上做遍历迭代，直到收敛；在每一次迭代，一个进程对分配给它的行进行计算，并按需要进行通信。这里主要的区别在于协调，在于用来表示逻辑上共享的矩阵 A 的数据结构，在于进程间的通信和同步是如何实现的。我们将主要关注这些差别。

表 2-3 若干基本的消息传递原语

原语名称	语 法	功 能
CREATE	CREATE(procedure)	创建从 procedure 开始执行的进程
SEND	SEND(src_addr, size, dest, tag)	从 src_addr 开始，向 dest 进程发送标识为 tag 的 size 字节
RECEIVE	RECEIVE(buffer_addr, size, src, tag)	从 src 进程接收一个标识为 tag 的消息，将其 size 个字节放到 buffer_addr 开始的缓冲区中
SEND_PROBE	SEND_PROBE(tag, dest)	检查标记为 tag 的消息是否已经送给了进程 dest（只用于异步消息传递，其含义取决于本章讨论的语义）
RECV_PROBE	RECV_PROBE(tag, src)	检查是否已经从 src 进程收到标记为 tag 的消息（只用于异步消息传递，其含义取决于语义）
BARRIER	BARRIER(name, number)	在 number 个进程之间做全局同步，如果 number 个进程没有到齐，任何一个都不能通过 BARRIER
WAIT_FOR_END	WAIT_FOR_END(number)	等待 number 个进程终止

应用问题的数组规模是（$n+2$）×（$n+2$），我们这里不可能将它表示为一个全局数组。在消息通信程序中，每个进程在其私有地址空间中分配一个大小为（nprocs/n + 2）×（n + 2）的数组 myA。这个数组表示所分配给它的矩阵 A 的 nprocs/n 行，加上边界的两行，以容纳从它的相邻划分来的边界数据（用于网格点的更新）。从其邻居来的边界行必须和它显式地通信，并拷贝到这些额外的或阴影（ghost）行，否则由于不在进程的空间中，它们的元素就不能被直接引用。阴影行的使用在于如果没有它们，通信的数据就得接收在特别为此创建的单独的有不同名字的一维数组中，这就使得数据的引用不可能像内循环 20 ~ 21 行那么简单。由于被通信的数据总是必须要拷贝到接收者的私有地址空间，通过将现有数据结构扩大一些，而不是分配一个新的，会使程序设计容易许多。 109

回顾第 1 章所提到的，在一个消息传递程序中，通信和同步都是基于两个原语：SEND 和 RECEIVE。程序中引起数据传送的事件是 SEND 操作，而不像在共享地址空间其数据传送通常由消费者或接收者用 read（load）指令引起。当一个消息到达了目的处理器，它要么保存在网络队列里或者暂时存放在一个系统缓冲区中，直到运行在目的处理器上的一个进程给出一个对应的 RECEIVE。通过这个 RECEIVE，进程将在网络队列里或系统缓冲区中的消息读入它自己私有（应用）地址空间的某个特定部分。RECEIVE 自己不引起任何跨越网络的数据传送。

⊖ 另外一种做法是用所谓“无主机”模型，没有一个主要的进程。所用的进程数在程序启动时告诉给系统。系统然后启动相应数量的进程，将代码分发到相应的处理节点。在程序中用不着 CREATE 原语；每个进程各自读程序的输入（n 和 nproc），但它们仍然要获得惟一的用户级标识号。

```
1.   int pid, n, nprocs;              /*process id, matrix dimension and number of
                                       processors to be used*/
2.   float **myA;
3.   main()
4.   begin
5.     read(n);  read(nprocs);        /*read input matrix size and number of processes*/
8a.    CREATE (nprocs-1, Solve);
8b.    Solve();                       /*main process becomes a worker too*/
8c.    WAIT_FOR_END (nprocs-1);       /*wait for all child processes created to terminate*/
9.   end main

10.  procedure Solve()
11.  begin
13.    int i,j, pid, n' = n/nprocs, done = 0;
14.    float temp, tempdiff, mydiff = 0;     /*private variables*/
6.   myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
                                       /*my assigned rows of A*/
7.   initialize(myA);                  /*initialize my rows of A, in an unspecified way*/

15.  while (!done) do
16.    mydiff = 0;                     /*set local diff to 0*/
16a.   if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
16b.   if (pid = nprocs-1) then
           SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
16c.   if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
16d.   if (pid != nprocs-1) then
           RECEIVE(&myA[n'+1,0],n*sizeof(float), pid+1,ROW);
                                       /*border rows of neighbors have now been copied
                                       into myA[0,*] and myA[n'+1,*]*/
17.    for i ← 1 to n' do              /*for each of my (nonghost) rows*/
18.      for j ← 1 to n do             /*for all nonborder elements in that row*/
19.        temp = myA[i,j];
20.        myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.          myA[i,j+1] + myA[i+1,j]);
22.        mydiff += abs(myA[i,j] - temp);
23.      endfor
24.    endfor

                                       /*communicate local diff values and determine if
                                       done; can be replaced by reduction and broadcast*/
25a.   if (pid != 0) then              /*process 0 holds global total diff*/
25b.     SEND(mydiff,sizeof(float),0,DIFF);
25c.     RECEIVE(done,sizeof(int),0,DONE);
25d.   else                            /*pid 0 does this*/
25e.     for i ← 1 to nprocs-1 do      /*for each other process*/
25f.       RECEIVE(tempdiff,sizeof(float),*,DIFF);
```

图 2-16 用显式消息传递方式描述并行方程求解器的伪码。现在，数据结构和它们下标的变化在并行代码中有所体现。每个进程有它自己的 myA 数据结构，表示分给它的网格部分，不同进程对 myA［i, j］的引用指的是逻辑上全局网格的不同部分。通信发生在 16a ~ 16d 和 25a ~ 25f 行。由于同步隐含在 send/receive 原语中，不再需要栅障和锁。还有几行代码用来协调用简单发送和接收实现的通信

```
25g.        mydiff += tempdiff;          /*accumulate into total*/
25h.      endfor
25i       if (mydiff/(n*n) < TOL) then   done = 1;
25j.      for i ← 1 to nprocs-1 do       /*for each other process*/
25k.        SEND(done,sizeof(int),i,DONE);
25l.      endfor
25m.    endif
26.  endwhile
27.  end procedure
```

图 2-16（续）

用在这个例子程序中的 SEND 和 RECEIVE 原语，假设了被传送的数据是在虚拟地址空间的一个连续的区域上。在我们简单的 SEND 调用中的参数是：被发送数据在发送进程的私有地址空间里的起始地址；消息的大小（字节数）；目的进程的 pid（不同于在共享地址空间的情形，我们现在必须要能够显式命名进程）；还有一个可选的和消息关联的标记或类型，用于在接收方匹配。RECEIVE 调用的参数是一个局部地址，从那里开始放置收到的数据；消息的大小；发送者的 pid；以及可选的消息标记或类型。所指定的发送者 pid 和消息标记，如果出现，被用来和已经接收到系统缓冲区的消息进行匹配，看哪一个和要接收的对应。这样的参数可以是通配的，可以和来自于任何源进程或任何消息标记匹配。SEND 和 RECEIVE 原语通常在一个针对特别的体系结构的库中实现，如共享地址空间的 BARRIER 和 LOCK 那样。常用于实际程序的消息传递原语是称为消息传递接口标准的一部分。消息传递接口简称为 MPI（不同层次的描述见 Pacheco 1996；MPI Forum 1993；Gropp、Lusk、Skjellum 1994）。一个重要的扩充是存储器非连续区域的传送，无论是含有规则的跨距（例如在地址 a 和 b 之间的每第 10 个字或每隔 6 个字取 4 个字），还是用索引向量来说明非结构化的地址，从中在发送方收集数据或在接收方分布数据。另一个进步是说明消息匹配标记的灵活性程度的提高，以及潜在的匹配复杂性。例如，进程可能被分为组，只在组内进行某种类型的消息通信，集体通信操作可以用如下描述的方式提供。

从语义上讲，在我们程序中能用的最简单形式的 SEND 和 RECEIVE 是所谓同步形式。一个同步的 SEND 操作把控制返回到调用进程，仅当对应的 RECEIVE 已经完成了。一个同步的 RECEIVE 返回控制，仅当数据已经收到了目的进程的地址空间中。用同步消息，我们在 16a ~ 16d 行的通信实现实际上是死锁的。所有进程首先发出它们的 SEND，然后停下来等对应的接收被执行，因此谁也不会开始真正执行它们的 RECEIVE！一般来说，如果我们不小心的话，同步消息在成对交换数据的情形下很容易死锁。避免这个问题的一种方式是让一些进程先做 SEND，后做 RECEIVE；而让另一些进程先做 RECEIVE，后做 SEND。另外的方式是用含有不同语义的发送和接受原语，后面将简要地讨论。

在每次迭代开始时，所有的通信是一次完成的，而不像在共享地址空间的情形，通信的单位是网格点。这里也可以一个网格点一个网格点地通信，但发送和接收操作的开销通常太大，这样做很不合理。作为结果，不同于共享空间的版本，消息传递程序是确定性的。尽管一个进程更新其边界行时它的邻居正在同一次遍历中计算，由于它们不在其地址空间邻居不会看到当前遍历中的更新。因此，一个进程所看到的邻居边界行是上一次遍历结束时的值。按照我们前面的讨论（红－黑序在这里会特别有用），这可能需要较多的遍历才能收敛。

110 ~ 112

一旦一个进程将它邻居的边界行收到它的阴影行，它就可以更新分配给它的格点，用的代码几乎和顺序程序和共享存储程序完全一样。（尽管我们用不同的名字 myA 来表示进程的局部数组，不同于顺序和共享地址空间程序的 A，这只是为了将它和逻辑上共享的网格 A 区别开来，A 在此只是概念上的；我们完全可以就用 A 来作为它的名字）循环的边界是不同的，每个进程都是从 1 到 nprocs/n（在代码中由 n' 代替），而不是像顺序程序中的从 0 到 n－1或者共享地址空间程序中的 mymin 到 mymax。实际上，用来引用 myA 的下标是局部下标；如果整个逻辑共享的网格 A 要作为一个共享数组引用的话，则必须用全局下标。例如，引用 myA[1, 1] 对不同的进程来说指的是逻辑共享网格 A 的不同元素[⊖]。在要用到全局索引的场合，使用局部索引空间有时可能会显得费解，如习题 2.7 所示。

和共享地址空间相比，这里的同步和它相当不同，包括将私有 mydiff 变量累加到逻辑上共享的 diff 变量上，以及随后的 done 条件的判断。假设我们用简单的同步发送和接收，阻塞进程直到操作完成，send/receive 匹配包含有一个同步事件，不需要什么特别的操作（像锁和栅障）或附加变量来实现同步。对于互斥来说，逻辑上共享的 diff 变量必须分配在某个进程的私有地址空间（这里是进程 0）。这个进程的标识必须为所有其他进程所知。每个进程将它的 mydiff 值送给进程 0，它将 mydiff 值加到逻辑上共享的 diff 中。由于只有进程 0 能够处理这个逻辑上共享的变量，互斥和序列化自然发生，不需要用锁。事实上，进程 0 可以就用它自己的 mydiff 变量作为全局的 diff，将其他进程送来的 mydiff 加到其上。

现在考虑用来确定 done 条件的全局事件同步。一旦进程 0 收到了所有其他进程的 mydiff 值并将它们累加了起来，它就测试 done 条件，然后将 done 变量送给所有其他进程，它们正用 receive 调用等着它。这里不需要有一个栅障，因为同步接收的完成隐含着进程 0 已经送出了 done 结果，因此所有进程的 mydiff 都已被累加起来了。所有进程这时就可以局部地测试 done 条件，来决定是否进行下一次遍历。当然，如果对程序设计更方便的话，我们也可以用消息来实现锁和栅障，尽管那意味着请求－回答通信而因此会有较多的消息往返。后面我们也会看到，比我们在此用的同步型更复杂的 send/receive 语义可能要求除消息本身以外的附加同步。

注意，当用点到点发送和接收作为通信操作时，用于累加和结束条件判定的代码已经扩展到了好几行。在实践上，程序设计环境常会为程序员提供库函数，如 REDUCE（将多个进程中的私有变量累加到某个进程的单个变量中）和 BROADCAST（从一个进程向所有其他进程发送），在进程中直接用它们能简化这些典型情况下的代码。用这些函数，图 2-16 中的 25a～25m行能由图 2-17 中的 5 行代替。系统可能提供特别的支持，来改善这些以及其他一些集体通信操作的性能（例如一对多、多对多以及所有对所有进程的通信）。例如，这种支持可能是将发送方的开销减少到只有一条消息。有时，这些操作可能就是在用户级别的库中，建立在通常的点对点发送和接收上，从而提供程序设计的方便。

113

最后，我们前面提过，不同的 SEND 和 RCCEIVE 操作版本有不同的语义，我们可以用它们来解决前面的死锁问题。让我们进一步考察这一点。这里的主要区别在于它们的完成语义——即，何时将控制返回到调用发送或接收的用户进程。这些语义影响操作所用到的数据结构或者缓冲区何时能被重用，而不影响正确性。两种主要的 SEND/RECEIVE 是同步和异

⊖ 原书为 row，有误。——译者注

步的；异步又分两种：阻塞和非阻塞。让我们考察这些情况，看它们如何用在我们的程序中。

同步的 SEND 和 RECEIVE 是我们以前所假定的，因为对程序员来说它们的语义最简单。同步的 SEND 把控制返回到调用进程，仅当在目的端的相应同步 RECEIVE 已经成功完成且向发送者发送一个确认。如果没收到这个确认，发送进程就不能执行 SEND 后面的任何代码。收到确认消息，意味着接收者已经从系统缓冲区将整个消息取到了应用空间，这样，SEND 的完成保证了（如果没有硬件错误）消息已被成功接收，所有相关的数据结构和缓冲区能被重用。

```
/*communicate local diff values and determine if done, using reduction and broadcast*/
25b.  REDUCE(0,mydiff,sizeof(float),ADD);
25c.  if (pid == 0) then
25i.   if (mydiff/(n*n) < TOL) then done = 1;
25k.   endif
25m.   BROADCAST(0,done,sizeof(int),DONE);
```

图 2-17 用 REDUCE 和 BROADCAST 实现的累加和收敛测试。REDUCE 函数的第一个参数是目的进程。在 REDUCE 的实现中，所有其他进程要对这个进程做一个发送操作，这个进程进行接收操作。下一个参数是要被归约的私有变量（所有进程都有），同时也是归约结果变量（对目的进程来说），第三个参数是这个变量所占的存储空间。最后一个变量是在归约中在这些变量上要实现的函数，类似地，BROADCAST 的第一个参数是发送进程标识符，这个进程进行发送操作，其他进程进行接收操作。第二个参数是要广播和接收的变量，第三个是其大小。最后一个是可选的消息类型

阻塞异步（或者就称为阻塞）SEND，把控制返回到调用进程时消息已经从发送应用程序的源数据结构中取走，因此已在系统的管理之下。这意味着当控制返回时，发送进程能修改源数据结构而不会影响消息。和同步 SEND 相比，这允许发送进程更早地向前推进，但控制的返回不保证消息已被或者将被送到适当的进程。要得到这样的保证需要进程之间额外的握手。阻塞的异步 RECEIVE 和同步的 RECEIVE 类似，返回时消息已经到了应用程序的地址空间。一旦返回，应用程序就能立即使用缓冲区中的数据。不同的是，阻塞 RECEIVE 不向发送者发送确认。 114

非阻塞异步（或简单地称为非阻塞的）SEND 和 RECEIVE 允许计算和消息传递之间最大的重叠，它返回调用进程最快。非阻塞 SEND 立即返回控制。非阻塞 RECEIVE 在简单地表达了接收意图后返回控制；消息的实际接收和在应用程序数据结构中的放置，在不确定的时间由系统基于提出的接受要求异步完成。然而，在非阻塞 SEND 和 RECEIVE 中，控制的返回不意味着任何事情，无论是消息的状态，还是它所用的应用程序的数据结构都不清楚，因此当需要的时候用户有责任来确定那些状态。系统常常提供单独的原语用来查询这些状态。这样，非阻塞消息典型地以两段式来使用：首先，SEND/RECEIVE 操作本身，然后在需要的时候进行检测。这样的检测（必须由消息传递库提供），可能阻塞直到所盼望的状态出现；还可能立即返回并报告所查到的状态。

选择哪一种语义的 SEND/RECEIVE 取决于程序要如何使用它的数据结构，我们通常是在程序设计的容易程度、程序的可移植性以及性能之间权衡。因为仅有私有地址空间，互斥自然不需要了，这些语义主要影响事件的同步。在方程求解器例子中，用异步的 SEND 和阻塞异步的 RECEIVE 将能避免死锁，因为此时进程能够越过 SEND，到达 RECEIVE。然而，如果用非阻塞的异步 RECEIVE，在实际用其所指定的数据结构之前，我们就得做检测。注意阻塞的 SEND/RECEIVE 等价于非阻塞的 SEND/RECEIVE，紧跟着一个阻塞的检测。 115

为了更好地体会共享地址空间和消息传递程序设计模型的区别，通过一个练习来将这个方程求解器的消息传递版本改写成循环分配数组的版本将是有教益的，如同我们在例2.2对共享地址空间版本所做的那样。在这种情形要考察的要点是，尽管两个消息传递版本表面上看起来相似，myA数据结构的含义将完全不同。在一种情形下它是这个全局数组的一个连续的部分，另一种情形下它是一组分开的行。只有通过对数据结构和通信模式仔细的审视，你才能确定一个给定的消息传递版本是如何对应于原始的顺序程序或者它的共享地址空间对应版本的。

2.4 结论

将一个顺序应用程序并行化的过程是相当有条理的：我们将程序要做的工作分解为任务；将这些任务分配给进程；协调数据访问、通信和进程之间的同步；可能还要将进程映射到处理器上。对许多应用来说，包括本章所用的简单方程求解程序，最初的分解和分配都是类似，甚至是完全一样的，因此与采用共享地址空间还是消息传递程序设计模型是无关的。区别是在协调部分，尤其是数据结构的组织和访问的方式以及通信和同步的实现方式。共享地址空间使我们用和串行程序同样的数据结构，也可以产生正确的并行程序。通信隐含在数据访问中，至少从正确性的保证来考虑，数据划分是不需要的。在消息通信的情形，我们必须从每个进程的私有数据结构来同步逻辑上共享的数据结构。通信是显式的，在进程的私有地址空间中需要进行数据的显式划分，进程必须能够相互用名字来通信。另一方面，共享地址空间程序要求有附加的同步原语（不同于读和写）来完成隐式通信；在许多消息传递系统中，同步功能包含在显式的发送和接收操作中。随着我们考察更复杂的应用程序的并行化，例如本章介绍的4种案例，我们将理解这些区别对于程序设计的难易程度以及由对高性能的追求所带来的各种考虑的影响。

这个简单的方程求解程序的若干并行版本在这里用来解释程序设计的原语。尽管这些版本的性能不会十分差（例如，我们用块划分，而不是用循环划分行来降低通信；通过首先累加到局部的mydiffs，然后再到全局的diff，大大地降低了通信和同步），程序仍有改进的潜力。在下一章，我们将注意力转移到并行程序的性能问题，将看到如何进行改进并看到所采取的改进方法会如何影响加到体系结构上的工作负载。

116

习题

2.1 举出两个例子，说明好的并行算法并不一定基于最好的串行算法，因为最好的串行算法体现不出足够的并行性。

2.2 在我们所讨论的4个案例中（Ocean、Barnes-Hut、Raytrace、Data Mining），哪些你认为更适合进行数据分解，从而在并行化中使用拥有者计算原则而不是计算分解？如果在其他情况下用严格的数据划分和拥有者计算原则，你认为问题出在哪里？

2.3 在共享存储空间，有两种主要的模型规范父子进程是如何相关联的。重量级的称为进程模型，当一个进程创建其他进程时，子进程得到父进程映像的一个拷贝；即，如果父进程分配了一个变量 x，那么子进程在它的地址空间也有变量 x，它初始化为父进程创建子进程时的值。然而，任何一个进程对它自己的 x 拷贝的修改只有自己所见。在轻量级的线程模型中，子进程或线程得到对父进程映像的指针，于是子进程和父进

程看到的是相同的存储单元。在这种模型下，除了在过程栈上的以外，由任何进程或线程分配的所有数据都是共享的。

1）考虑一个进程要在程序的不同部分访问它的进程标识数 pid，特别考虑在它调用的子程序中的引用（在调用链中）。用第一种模型，你打算如何实现这一要求？你需要用进程的私有数据，还是完全用全局共享数据？

2）用前一种模型（进程）写的程序可能依赖于这样的事实：子进程得到父进程数据结构的一个私有拷贝。你要做什么样的改变来把此程序移植到后一种（线程）模型，其进程在创建子进程后 i）只读数据结构 ii）既读还写？

2.4 经典的有限缓冲问题是点到点事件同步的一个范例。两个进程通过一个有限缓冲区通信。当缓冲区没满时，一个进程、生产者将数据项加到缓冲区；当缓冲区非空时，消费者从缓冲区读数据。如果消费者发现缓冲区空，它就必须等待生产者插入数据项。当生产者准备插入一个数据项时，它检查缓冲区是否满；若满，它就要等消费者从中取走一些项。如果缓冲区空，生产者要加一项，那么取决于实现情况；消费者可能正等通知，因此生产者要通知它。你能仅用点到点事件同步实现有限缓冲区吗？或也需要用互斥？设计一个实现，包括伪代码。

117

2.5 对于单处理器操作系统，你是用在一个标志上踏步等待，还是用进程阻塞来做进程间同步？在多处理器情形，你怎么考虑阻塞和踏步等待之间的权衡？

2.6 在共享存储空间版本的并行方程求解程序中（如图 2-13 所示），我们在一次 while 循环迭带中为什么需要第一个和第三个栅障（16a 行和 25f 行）？你能去掉它们而不加入任何其他同步吗，也许当做某些操作时需要做些改动？考虑所有可能的情况。

2.7 高斯消元法是一种求解联立线性方程组的著名技术。变量一个一个地消去，直到剩下一个变量，然后回带得到所有变量的值。在实践中，将方程组的未知数的系数表达成一个矩阵 A，首先将这个矩阵转换成一个上三角阵（在主对角线下的所有元素均为零），然后用回带。通过相继的变量消元将矩阵 A 转换成上三角阵。串行高斯消元法的伪代码如图 2-18 所示。k 循环的对角线元素称为主元，它所在的行称为主元行。

```
procedure Eliminate (A)          /*triangularize the matrix A*/
begin
for k ← 0 to n-1 do              /*loop over all diagonal (pivot) elements*/
  begin
    for j ← k+1 to n-1 do        /*for all elements in the row of, and to the right of,
                                  the pivot element*/
      A_k,j = A_k,j / A_k,k);    /*divide by pivot element*/
    A_k,k = 1;
    for i ← k+1 to n-1 do        /*for all rows below the pivot row*/
      for j ← k+1 to n-1 do      /*for all elements in the row*/
        A_i,j = A_i,j - A_i,k* A_k,j;
      endfor
      A_i,k = 0;
    endfor
  endfor
end procedure
```

图 2-18 描述串行高斯消元法的伪代码

1）画一个简单的图，解释矩阵元素之间的依赖关系。

2）假设一个按行的分解和按连续行块的分配，写一个共享存储的并行版本，用本章方程求解器所用的原语。

3）对相同的分解和分配，写一个消息传递版本；先用同步的消息传递，再用任何一种异步消息传递。

4）你能发现这种划分带来的明显性能问题吗？（我们将在下一章进一步讨论）

5）修改共享存储和消息传递版本，用行循环的分配方式。

6）讨论共享存储和消息传递版本在程序设计方面的权衡（编程的容易性和可能的主要性能差别）。

2.8 假设一个支持共享地址空间的系统不支持栅障，但只支持信号灯。即使全局事件同步也得通过信号灯或普通标志来构成。信号灯的使用可作如下考虑。假设进程 P_2 要向进程 P_1 指出（用信号灯）P_2 已经到达程序的 b 点，因此 P_1 可以推进通过 a 点（它正在那里等待）。P_1 到达 a 点后在一个信号灯上做一个等待操作（也称为 P 操作或 down 操作），P_2 到达 b 点后在同样的信号灯上发一个信号（做一个 V 操作或者 up 操作）。如果 P_1 在 P_2 到达 b 前到达 a，它就将自己挂起或阻塞，等待被 P_2 的信号操作唤醒。

118

1）在共享存储的并行高斯消元法中，你会怎么设计同步，针对 i）用标记 ii）用信号灯取代栅障？你能用点对点或组事件同步，而不用全局事件同步吗？

2）对方程求解器例子回答同样的问题。

2.9 在我们所讨论的这种直接的，基于循环的并行化高斯消元法中，并行性仅在最外层 k 循环迭代里得到开发。由于主元和它的行（称为主元行（pivot row））实际上是直接广播到所有需要它的进程，这称为是广播版本。高斯消元法也能并行化，以一种更激进的形式开发可用的并行性，甚至跨越外层循环迭代。在第 k 次迭代期间，分配有主元行的进程能简单地将主行送给下一个进程，而不是将它广播。这个进程能用主行立即更新它所分得的行，同时将它传送到下一个进程，等等。只要这个进程完成了顺序程序中循环的第 k 次迭代的计算，它就能立即做它的主元行的计算，对第（$k+1$）次迭代，不需要等所有其他进程都收到第 k 行，做它们在第 k 次迭代中的工作。然后进程也可以将这（$k+1$）行传送到下一个进程，它能够马上用上一进程，而不用等整个 k 次迭代的完成。多次 k 循环迭代就可同时进行，只要它们已被计算出来，矩阵的行就可以沿着处理器流水线向下传送，并且只要所需的行通过流水线到达后即开始计算。我们称此为流水形式的并行化。

1）写一个共享地址空间的伪代码，如图 2-13 的细节程度，实现在个别元素粒度的流水并行性。示出所有必要的同步。你需要栅障吗？

119

2）针对 1）所提的流水情形，写一个消息传递伪代码在图 2-16 的细节程度。假定仅有的通信原语是同步和异步（阻塞和非阻塞）的发送和接收。你会用发送和接收的哪种版本，为什么不选用其他的？

3）对比讨论基于循环和流水并行程序设计中的折中。

2.10 多播（从一个进程向一组其他进程发送消息）是在进程子集之间的一种有用的通信机制。

1）你如何用多播而不是广播来实现一个消息传递的高斯消元法数据循环分配的版本？设计一个多播原语并写出伪代码，比较两种版本程序设计的难易。
2）你认为哪一种性能会好些，为什么？
3）除多播外，你认为一个消息传递系统还应该支持什么样的组通信原语？给出计算的例子来说明它们的用处。

120

第3章　面向性能的程序设计

我们知道，用并行处理系统的目的是为了获得高性能。具体地理解了分解、分配和协调等步骤是怎么体现在实际运行在机器上的并行程序代码中的，我们就可以来考察那些限制并行程序性能的关键因素以及在许多问题中它们是如何表现出来的。我们将看到在程序设计过程中所做的决定是如何影响给体系结构的运行时特征以及体系结构的特征是如何影响程序设计决定的。理解程序设计技术和这些相互制约的关系，不但对并行软件设计人员，而且对系统结构人员都是重要的。除了将并行程序看成我们所设计系统的工作负载外，我们从中学会做硬件/软件之间的权衡。特别是，我们可以认识到系统结构能够有效地影响可编程性和性能的哪些方面，而其他方面最好留给软件。对于多处理器系统的性能来说，程序和系统的相互依赖关系要比单处理器情形所涉及的面更宽、更加复杂也更加重要；因此，对于我们设计高性能系统，要降低代价和程序设计劳作的目标来说，理解这种相互作用是很关键的。从第4章讨论工作负载驱动的系统结构评估的方法开始，这种理解将是我们贯穿全书的一种基本精神。

有关性能问题和并行软件技术的内涵是丰富的：不同的目标相互制约，更好地实现某个目标的技术可能使我们重新考虑用于解决另外目标的技术。这就是并行软件创建的引人入胜之处和挑战所在。如同在单处理器的情形一样，多数性能问题能够通过软件算法和程序设计技术，或者由体系结构技术来解决或者两者兼用。本章着重于性能问题和软件技术。至于体系结构方面的技术，将在本书的其余部分讨论，本章只是隐含地涉及一些要点。

尽管我们必须要考虑若干具有相互作用的性能问题，但它们不是同时来处理的。创建一个高性能程序的过程是逐步求精的过程。如第2章所讨论的，划分的步骤（分解和分配）常常是和底层系统结构或程序设计模型相独立，它们主要关心算法方面的问题，只依赖于问题固有的特性。特别地，这些步骤只是将多处理器简单地看成是一组相互通信的处理器。它们的目标是要解决进程间的工作负载平衡；减少程序中固有的进程间通信；减少为了计算和管理划分所带来的额外的工作。我们首先将注意力集中在这些和划分有关的问题上。

然后，我们看体系结构，考察它在协调和映射步骤中所涉及的新的性能问题。这意味着我们要认识到两种事实。第一，多处理器不仅是一组处理器，而且还是一组存储器，每个处理器可以将这存储器看成是一种扩充的存储器层次结构。在这些存储层次中数据的管理可能引起更多的数据在网络上传送，使得实际发生的通信要比并行程序划分所导致的固有通信要多。因此，实际发生的通信既依赖于划分，也依赖于程序访问的模式、数据引用的局部性和这种扩充的存储器层次结构的组织与管理相互作用。第二，由处理器所看到的通信代价（也就是通信在程序执行时间中的份额）不仅取决于通信量，还取决于它的结构以及和体系结构的相互作用。3.2节讨论通信、数据的局部性及其和扩充的存储结构之间的关系。然后在3.3节针对这些在协调和映射方面的主要性能问题，考察有关的软件技术：通过在扩充的存储器层次结构上开发数据的局部性；减少额外的通信；将通信结构化以减少其开销。

当然，在协调步骤中，我们必须面对系统结构的相互作用和通信代价的问题，这有时会使我们回过头去，重新修改我们的划分方法，这对于并行程序的提炼是一个重要的环节。尽管相互作用和权衡发生在我们所讨论的所有性能问题之间，本章尽量独立地考虑每个因素并在适当的场合指出各因素之间的折中之处。所有的例子都源于第 2 章介绍的 4 个案例分析，通过一些测试数据，我们可以解释一些具体程序设计技术的影响；所用的计算机是 SGI 的 Origin2000，一种物理上具有分布存储器、缓存一致性的机器（这种机器在第 8 章详细讨论）。在我们的讨论中，方程求解器内核也经常作为例子出现。在介绍提高性能的技术时，只要和它相关我们总是试图给出在它上面的应用；这样，在讨论的结束时，我们将会得到该求解器的一个高性能并行版本。

当考虑影响性能的因素时，我们将给出简单的分析表达式，来表达并行程序的加速比，解释每种性能因素是如何影响加速比公式的。然而，从系统结构的观点来看，一个更具体的考察性能的方式是，要从机器中每个处理器的角度看执行时间的不同组成部分——即多少时间花在执行指令上；多少时间花在访问扩充存储器层次结构的数据上；多少时间花在等待同步事件的发生上。事实上，执行时间的这些成分能直接映射到性能问题上，这些问题必须由软件在创建并行程序时考虑。考察性能的这种观点帮助我们非常具体地理解并行程序作为工作负载呈现给系统结构执行的情形，映射帮助我们理解程序设计技术如何能改变这个特性。这些主题在 3.4 节讨论。

122

一旦我们研究了性能问题和技术，我们可以以 4 个案例分析为例，考虑如何创建实际应用的高性能并行程序版本。在 3.5 节，我们对每一个案例依次应用并行化过程和提高性能的技术。它解释这些技术如何一起使用以及所导致的执行特征的范围，这些特征在执行时呈现给系统结构，反映在执行时间的多种性态中。我们还将考虑实际应用在两种底层程序设计模型之间的影响和折中，这两种程序设计模型即共享地址空间和显式消息传递。权衡的要点在于程序设计的难易和程序执行的性能，将在 3.6 节讨论。下面让我们从分解和分配中算法的性能问题开始。

3.1　划分阶段的性能问题

在分解和分配步骤中，我们可以将并行计算机系统简单地看作是一个相互合作的处理器集合，不用考虑程序设计模型和硬件系统组织。我们只需知道在处理器之间的通信开销是很大的。在算法方面的三个基本要素是：

- 平衡工作负载，减少花在等待同步事件上的时间。
- 减少通信开销。
- 减少由确定和管理分配所带来的附加工作。

不幸的是，即使这三种基本目标彼此之间也是有冲突的，必须进行权衡。例如，只要在一个局部存储器装下所需的数据，让程序在一个处理器上运行就能使通信最小，但这就产生了负载的最不平衡。在另一方面，使程序中每个基本操作都成为单独的任务，随机分配它们可能获得最接近优化的负载平衡，但通信和管理开销将大得惊人。进而，在许多复杂应用中，负载平衡和通信可能通过在确定分配方案中花较多的时间得以改善，但这意味着需要附加的工作。分解和分配的目的是在这些冲突的要求中获得一个较好的折中。我们将在案例分析和方程求解器内核中具体讨论。

3.1.1 负载平衡和同步等待时间

工作负载平衡最简单的理解是要让每个处理器尽量做同样多的工作。它进一步的含义则是揭示足够的并发性（见第 2 章对 Amdahl 定律的讨论），做适当的任务分配，降低操作的串行化的现象。在这种概念下，可能得到的加速比的简单上限如下：

123

$$\text{加速比}_{\text{问题}}(p) \leq \frac{\text{串行工作量}}{\text{任何处理器承担的最大工作量}}$$

“工作”在此的含义不仅是完成了多少计算，还有花多少时间完成它们，这也包含了数据访问和通信。

事实上，负载平衡要比简单地将工作均分复杂一些。负载平衡不仅要让不同的处理器做同样多的事情，而且它们也要同时工作。极端的情况是，工作尽管均摊在进程之间，但每一时刻只有一个进程活跃，因此根本不会有加速比！负载平衡的真正目标是要极小化进程花在等待同步点上的时间，包括在程序结束时隐含的同步。这也就涉及到要使进程操作的串行化现象最小，而由于互斥（等待进入临界区）或数据的依赖关系，这种串行化是不可避免的。在分配步骤，我们应该保证低串行化的可能性，而协调步骤则应该保证这种可能性成为现实。

平衡负载和减少同步等待时间的过程分为 4 步：

1）在分解中识别足够的并发性，减少 Amdahl 定律的影响。

2）决定管理并发性的方式（静态还是动态）。

3）确定并发性开发的粒度。

4）降低串行化和同步代价。

本节通过 4 个案例和其他应用中的例子，考察一些有关的技术。

1. 识别足够的并发性：数据并行和功能并行性

在对方程求解器内核并行化的时候，我们看到发现并发性有多种途径，可以通过考察程序的循环，还可以通过深入研究数据的基本相关性，也可以通过对所求解问题事务的进一步理解，发现具有更大并发性的算法。将循环并行化通常导致一种情形，即相似的（不一定是相同的）操作序列或功能，作用在一个大数据结构的元素上；如同在方程求解器内核所见到的那样。这称为是数据并行性，是一种更一般的并行性形式，它激发了如第 1 章所讨论的数据并行体系结构。Barnes-Hut 中计算不同粒子的作用力也是这方面的一个例子。

除了数据并行性外，应用程序通常也表现出功能并行：完全不同的计算可能并发地作用在相同或不同的数据上。功能并行常常也称为是控制并行或任务并行（我们也注意到这些术语在不同的场合含义可能有些不同）。例如，在 Ocean 例子中，为求解器建立一个方程系统要求许多不同种类的计算，这些计算作用在大洋的横截面上，每种计算要用到几个截面网格。在整个网格或数组级别分析相关性揭示出这些计算是相互独立的，可以并行执行。流水是另一种功能并行的形式，其中流水线的不同的功能或阶段并发作用在不同的数据上。例如，在为视频帧编码时，每一帧的每一块要通过几个阶段：预过滤、从时域到频域的卷积、量化和熵编码等。流水并行在这些阶段上就可使用（例如，为每个阶段安排几个进程、并发操作），而数据并行存在于帧之间、同一帧的块中，以及在一块上的操作内。

124

功能并行和数据并行在应用程序中常常是同时存在的，表现出一种并行性的层次结构，我们需要从中挑选（例如，在海洋（Ocean）例子中，有跨越网格计算的功能并行性和网格内部计算的数据并行性，在视频编码例子中也有类似情况）。数据或功能并行的正交层次在许多其他应用中也常见到；例如，在 VLSI 电路中的布线应用，在要被排布的线路之间，一条线路内的不同段之间以及每段的多种可能的布线方式，都表现出并行性（见图 3-1）。

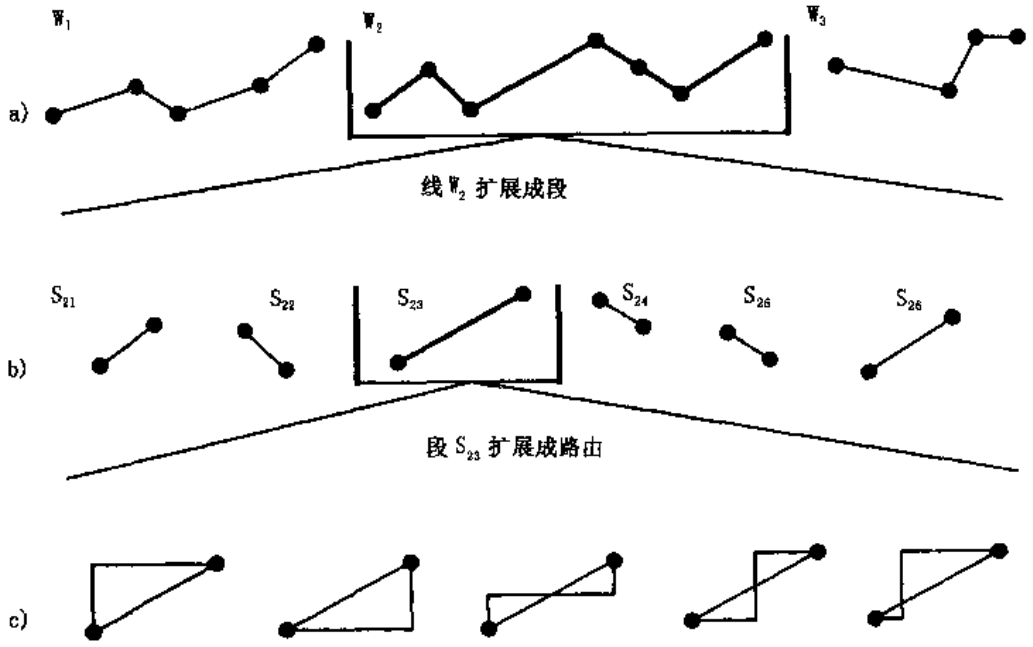


图 3-1　一种 VLSI 布线应用中的三种并行性：a）线间的并行性；b）同一条线内不同段之间的并行性；c）在同一段上的不同布线法之间的并行性。图中的实心圆圈表示由线路连接的焊点

可用的功能并行性通常是不大的，并且不会随着待求解问题的规模有大的变化。另一方面，数据并行性通常随数据集合的增大而增大。由于不同的功能涉及到不同的工作量，有不同的扩放特点，功能并行性常常也是更难以开发，难以做到工作负载平衡。按照我们松散的定义，大多数在大规模机器上运行的并行程序都是数据并行的，开发功能并行性主要是为了减少数据并行计算之间的全局同步量（如 3.5.1 节的海洋所示）。 125

通过识别应用中不同类型的并发性，我们常常会发现问题所表现出来的并行性比我们做工作负载平衡所需的要多。于是，在分解的下一步是通过确定任务的粒度来控制实际要用的并行性。然而，任务大小的选择也取决于我们期望如何来管理并发性，下面我们就来讨论这一点。

2. 确定如何管理并发性：静态分配和动态分配的对比

开发并发性的关键是希望能得到好的工作负载平衡，具体可以通过一种静态的或者事先确定的分配（第 2 章所介绍的）方法，或者采用动态的方法。静态分配通常就是构造一个任务向进程的映射算法，如同前面所讨论的简单方程求解内核的情形那样。任务（网格点或者行）在进程上的具体分配可能取决于问题规模、进程个数以及其他的参数，但一旦定下来了，在运行时分配就不再变化。由于分配是事先确定的，静态技术在运行时不会引起太多管理开销。不过，为了取得好的负载平衡，要求静态技术要么能够相当准确地预测不同任务中的相对工作量，或者能有足够多的任务来从统计上保证一种平衡的任务分配。除程序本身

外，其他环境条件也是很重要的。例如来自其他应用的干扰，虽然不影响处理器之间的关系，但限制了静态负载平衡方法的健壮性。

动态划分技术是在运行时对工作负载分布的情况进行调整，有两种形式。在半静态（semistatic）技术中，一个阶段计算的分配在那个阶段开始前通过一定的算法来决定，但要基于在运行时收集的工作负载分布信息周期性地重复计算分配方案，并据此重新调整负载的分配。例如，我们可以刻画（测出）每一个任务在一个阶段所完成的工作量，并用它作为下一次在这个阶段中工作量的估值。这种重新划分的技术在 Barnes-Hut 中（3.5.2 节）用来将星体分配给进程，在星系的演化过程的时间步之间采集有关数据，重新计算分配方案。星系是缓慢演化的，因此在星体之间的工作负载分布在相继的时间步里变化不多。图 3-2a 显示了在 Origin 2000 上对于 512 K 个粒子半静态划分和静态划分相比的优势，尽管周期性重新划分也会有些附加的计算代价。很清楚，性能的差别随使用的处理器数的增大而增大。

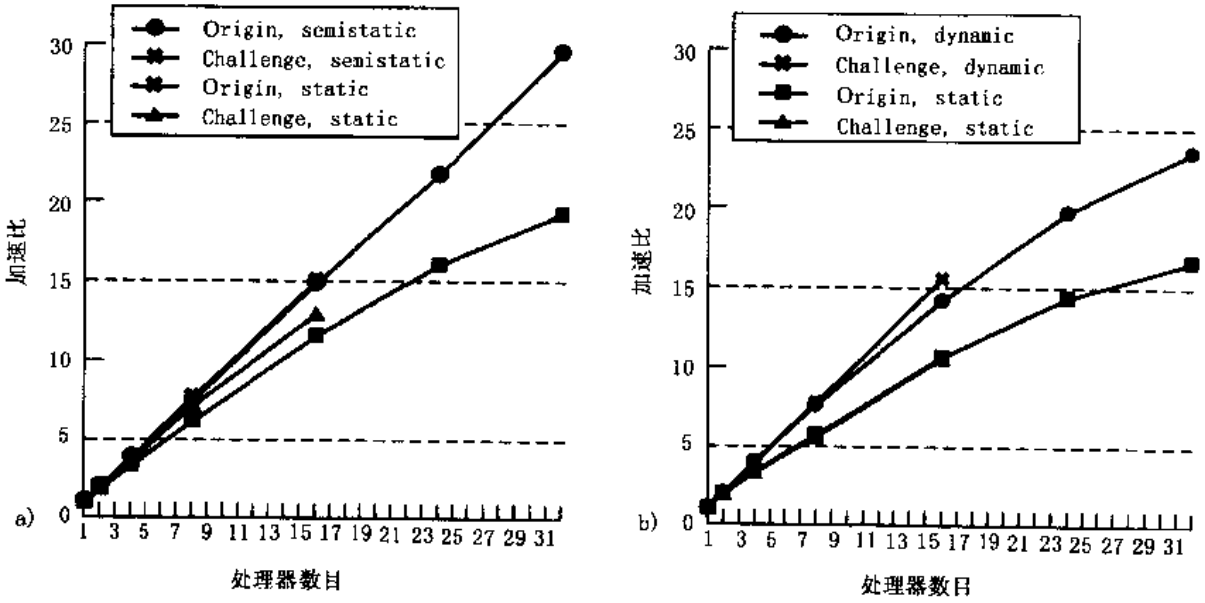


图 3-2 负载平衡动态划分的性能影响曲线。a）显示 Barnes-Hut 应用在静态划分和半静态划分情形下的加速比，b）显示 Raytrace 的加速比情况。即使在这些有大量并行性的应用中，和静态划分相比动态划分对改善负载平衡来说也有明显作用

126

第二种动态技术，动态任务调度（dynamic tasking），用来对付那些即使是周期性重新计算负载平衡的分配[⊖]，也很难决定负载分布的情况。例如，在 Raytrace 中，和每一条光线相关的计算量是不可能预测的。尽管渲染从不同的视点是重复的，但视点的变化不一定是渐变的。动态任务调度的做法将计算分成任务，并维护一个待执行的任务池（在 Raytrace 中，一个任务可能是一条光线或一组光线）。每个进程不断从这个任务池中取出任务并执行它（还可能将新的任务插入到池中）直到所有任务被执行完。当然，对任务池的管理要保证任务之间的相关性——例如，新插入的任务应该是随时可执行的。由于动态任务调度应用的广

⊖ 静态或者半静态分配的适用性不仅取决于程序的计算性质，还和程序同存储和通信系统的相互作用以及执行环境的可预见性有关。例如，即使负载从计算量来看是平衡的，但存储和通信停滞时间（由于缓存扑空、缺页或者冲突等原因）的不同可能会引起在同步点观察得到的不平衡。静态分配对分时或者异构系统也可能是不合适的。

泛性，让我们看看某些实现任务池的专门技术，图 3-2b 基于在 Origin2000 上的测试数据，显示了在 Raytrace 应用中，动态任务法相对于静态分配法将光线分给处理器的优势，其中的数据是一堆安排得像一堆葡萄的球体。

在共享地址空间系统做动态任务划分的一个简单例子是并行循环的自调度。循环计数器是执行该循环迭代的所有进程的共享变量。进程通过在该计数器上执行原子性的增量操作来获得一次循环迭代；参与工作的进程不断执行迭代并更新计数器，直到所有迭代完成。任务大小（粒度）可以通过一次取多个迭代来增加，即，在共享循环控制变量上加上一个比 1 大的数。然而，这样可能会增加负载的不平衡。有一种引导式的自调度方法（guided self-scheduling，Aiken and Nikolau 1988）：进程开始取较多的任务，随着循环的进展逐步减少，希望这样能够减少对共享计数器的访问次数而不影响负载平衡。

更一般的动态任务池通常用一组队列来实现，进程将任务插入这些队列并从这些队列中取出执行。这可能是单个集中式队列或者是一个分布队列的集合，典型的是每个进程一个队列，如图 3-3 所示。集中的队列简单，但每个进程访问相同的任务队列有不利之处，可能会增加通信和引起处理器争用队列访问的情况。而对队列的修改（添加任务或删除任务）必须要互斥，这就进一步增加了冲突，导致了操作的串行化。如果任务的粒度不大，使得进程每访问一次队列取走一个任务，干不了一会儿就又要回来访问队列，当进程数增加时，一个集中式的队列会很快成为性能的瓶颈。

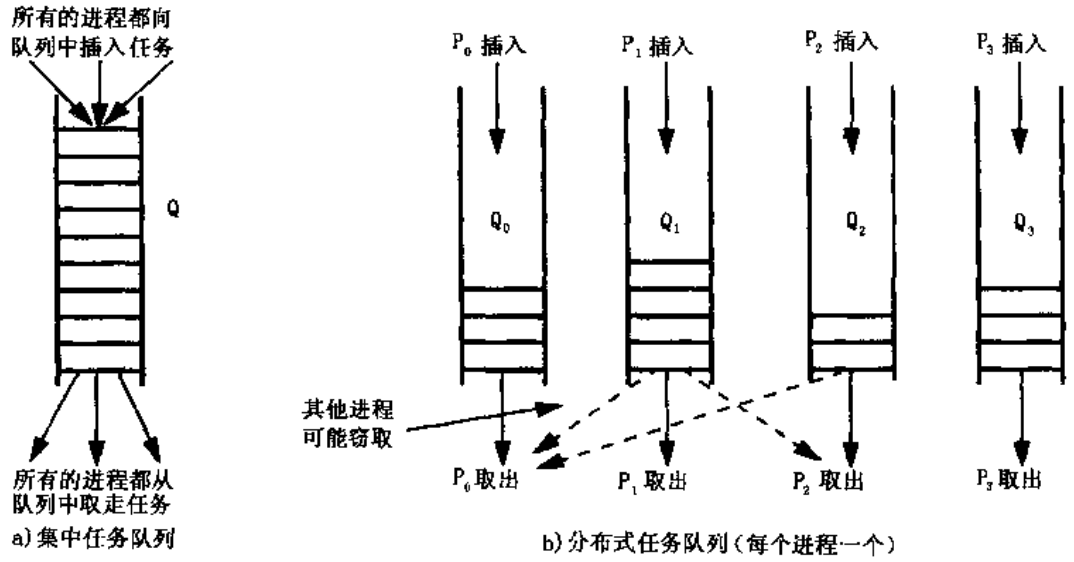


图 3-3　用任务队列组来实现动态任务池

对于分布式队列（distributed queue），每个进程最初在它本地的队列中会分得一些任务。这种初始的分配可以用某种智能化方法得到以减少进程间通信，这样可以提供比自调度和集中队列更多的控制。一个进程尽量从它自己的队列中移出和执行任务。如果它创建了任务，它将它们插入本地队列。当本地队列中没有任务时，它就查询其他进程的队列来从中获得任务，这就是一种称为任务窃取的机制。由于任务窃取（task stealing）隐含有通信，还可能产生竞争，在实现窃取过程中就提出了若干有趣的问题：例如，如何极小化窃取；从哪儿窃取；每次窃取多少任务和哪些任务等等。窃取还引入了重要的终止检测（termination detection）问题：如果所有任务都可能产生能动态插入队列中的其他任务，那么如何确定何时不

再窃取任务了，并假定所有任务都做完了？在实践中，简单的启发式算法能工作得很好，而一个完善的方案是相当复杂的，而且会引起很多通信（Dijkstra and Sholten 1968；Chandy and Misra 1988）。任务队列的方法用在共享地址空间和显式消息传递中。在共享地址空间中，队列是共享的数据结构，用锁来管理；在消息传递系统中，队列的拥有者为队列请求服务。

即使在不可预测且环境条件不清楚的条件下，动态技术通常也能给出好的负载平衡，但它们所导致的并行性管理代价要高得多。如果我们希望显式地控制某个任务必须在某个处理器上执行，动态任务技术也是不适合的[⊖]，从而可能增加通信，也可能会有损于对数据局部性的利用。因此，只要对一个应用和环境能提供好的负载平衡，静态技术通常是更可取的。

128

3. 确定任务的粒度

如果没有负载不平衡的问题出现（例如，在一个计算阶段开始时所有任务都是可以执行的），那么对于任务队列策略来说，负载的最大不平衡性等于最大任务的粒度。任务粒度，我们指和一个任务相关的工作量，用所执行的指令数，或者更确切些，用执行时间来度量。从实际开发并发性的角度看，对选择粒度的一般认识是细粒度或小任务有潜力得到更好地负载平衡（有更多的任务在进程间分配，因此有更多的并发性），但和粗粒度或大任务相比，它们导致较高的任务管理开销、更多的冲突、更多的进程间通信。下面我们来看看为什么，首先从动态任务排队的角度开始，因为在这种情况，定义和权衡比较清楚。

任务粒度和动态任务队列 这里，任务显式地定义为任务队列中的一项，因此任务的粒度就是和这一项相关的工作量。很明显，小任务所带来的管理开销较大（队列的管理）。至少对于集中式队列来说，更频繁的队列访问通常还会导致较大的竞争。最后，将一个任务分成两个较小的任务可能引起这两个任务在不同的处理器中执行，如果它们要访问同一个在逻辑上共享的数据，则会增加通信。

129

任务粒度和静态分配 对于静态的分配，任务在程序中不是显式的，因此什么叫并行操作的一个任务或者一个计算单元并不是那么清楚。例如，在方程求解器中，任务是数组的若干行，还是一行，还是一个元素？我们可以把任务定义为这样一种尽可能大的工作单位：即使任务在进程的分配改变了，实现该任务的代码不需要变[⊖]。和动态任务队列相比，由于没有队列的访问，静态分配中任务的大小对于任务管理开销的影响要小得多。通信和竞争受任务在进程上分配的影响，而和它们的大小关系不大。任务大小的主要影响通常在于负载不平衡和在处理器缓存中开发数据的局部性。

4. 减少操作序列化的现象

最后，要减小在同步点上操作的串行化。不管它是由于互斥还是任务间的依赖关系，我们必须仔细地考虑任务的分配和协调任务的同步和调度。对事件同步来说，过度串行化的一个例子是用比实际所需的更保守的同步方式，例如本来可以用点对点或组同步，但用了栅障同步。即使是用点对点同步，同步粒度的粗细也可能对串行化有影响。例如，一个进程和另一个进程的依赖关系本来只在矩阵的个别元素上，但却让它等待另一个做完整个一行矩阵元素的数据才可推进。然而，较细粒度的同步经常意味着较复杂的程序；它也隐含着要执行较多的同步操作（比方说，一个字一次，而不是在更大的数据结构上），这种开销可能要比在

⊖ 即程序员按照自己的考虑，安排好的任务到处理器的映射会得不到保证。——译者注

⊖ 这一段的意思是，如果将任务看成是一个由程序代码体现的计算量，那么该程序代码应该是独立于任务在进程中的分配的。这里有一种“封装”的含义。——译者注

串行化上的开销节省得多。于是，我们要折中权衡。

对于互斥，我们可以对不同的数据项用不同的锁来减少串行化；如果可能，就用较小的锁，不要经常来保护临界区。考虑前者，在一个数据库应用中，如果多个进程都能对记录访问，当我们更新某个记录的某个域时，我们可能要加锁。问题是如何组织锁的方法。每个进程一把锁，每个记录一把锁，还是每个域一把锁？粒度越细，竞争越低，但有更大的空间开销，并且锁的重用性也不好。一种中间的方案是用一个固定数量的锁，在记录之间共享，而在记录和锁之间用一个简单的哈希函数映射。减小串行化的另一个方法是在时间上错开对临界区的访问，即适当地安排计算，使得多个进程不要试图在同一时间访问相同的锁。

任务队列的实现提供了一个有趣的例子，使临界区更小并且不那么经常被访问。假定每个进程将一个任务加入一个队列，然后搜索这个队列找一个具有特别特点的任务，然后将它从队列中移出。任务的插入和删除可能需要是互斥的（如果它们发生在队列的不同端，也可能不需要互斥），但搜索不需要互斥。这样，我们就可以用两个临界区（一个插入，一个删除），而不是用单个临界区来包含操作的整个序列；用于搜索的代码不需要互斥。 130

更一般的情况，检测（读）一个被保护数据结构的状态通常不需要互斥；只是修改数据结构的时候需要互斥。如果经常发生的情况是检测而不是修改，例如在任务队列中搜索任务，我们就在检测的时候可以不用锁，只是当检测返回适当的条件时，我们就在修改前上锁并在临界区中重新检测（以保证状态没有发生变化）。除此以外，不用一把锁来管理整个队列，我们可以对每个队列元素用一把锁，从而队列中的不同部分的元素可以并行地插入和删除（没有串行化）。如同事件同步，在性能和程序设计的难易之间正确的权衡取决于不同的选择在一个系统中的代价和好处。

我们可以将加速比表达式如下扩充，反映负载的不平衡性和花在同步点的等待时间，其中分母中的 *max* 概括了所有进程：

$$\text{加速比}_{\text{问题}}(p) \leqslant \frac{\text{串行工作量}}{\max(\text{工作量}+\text{同步等待时间})}$$

通常，考虑平衡负载诸方面的因素是软件的责任。如果程序没有足够的并行性或者负载不平衡，一种体系结构所能发挥的作用并不多。然而，体系结构在某些方面能够起作用。首先，它可以对那些广泛用于并行软件（应用程序、库、操作系统）中的负载平衡技术（例如任务窃取）提供高效的支持。对一个远程任务队列窃取的访问通常是一个探测或者查询，涉及少量的数据传送以及可能的互斥。对细粒度通信和低开销、数据的互斥访问的支持越高效，就能使我们的任务变得越小，从而改进负载平衡。第二，体系结构能够使命名和访问逻辑上共享的数据（被窃取任务所需的）变得容易。第三，体系结构能对点对点同步提供高效的支持，使得人们更愿意使用这种同步，而不是保守的栅障同步，从而可以获得更好的负载平衡。

3.1.2 减少固有的通信

只要应用表现出足够的并发性，在概念上做到负载平衡是相当容易的：我们可以简单地让任务小一些，并利用动态任务分配。但对于负载平衡来说，也许它需要考虑的最重要的权衡是减少进程间的通信。将问题划分为多个任务通常意味着任务之间有通信的要求。如果这些任务分配给不同的进程，我们就在进程间（因此也在处理器间）引入了通信。本节的要点 131

是考虑在维持负载平衡的同时，减少那些并行程序之间固有的通信（即，一个进程产生另一个进程所需的数据），维持机器是一组合作进程的集合的观念。然而，在真实系统中，还会由其他原因导致通信，如3.2节所示。

要估计通信对程序性能的影响，绝对通信量往往不是最好的参数。人们更愿意考虑通信对计算的比值。它的定义是通信量（例如，以字节为单位）除以计算时间（或者考虑到影响时间的因素太多，则可以除以执行的指令数）。例如，1兆字节的通信相对于1 s的执行时间的影响就要比相对于1小时执行时间的影响要大得多。通信计算比可以按每个进程来算，也可以累计在所有进程上。

固有的通信计算比基本上是由任务在进程上的分配决定的。为了减少通信，我们应该尽量安排那些访问相同数据或要求频繁通信的任务分在同一个进程中。例如，在数据库应用中，如果访问同样数据库记录的查询和更新安排在同一个进程中，就能降低通信。

一个在实践中求取负载平衡和固有通信折中的划分的原则是区域分解（domain decomposition）。最初它是用在数据并行的科学计算问题中，如Ocean问题，但人们发现它也能用于许多其他领域。如果将操作作用其上的数据集合看成是一个物理区域，那么情形通常是，计算区域中的一个点所要求的信息，常常只需要从这个点的周围一个小的局部区域上获得；在需要从一个较大的范围获得信息的情况下，该范围中其他点对所求点的影响随它们之间距离的增加而降低。Barnes-Hut算法就是后者的典型例子。作为前者的例子，考虑一种视频应用，其中在视频编码和解码中用于运动估计的算法，只考察和当前像素接近的场景区域；类似地，在方程求解器内核中的一个数据点只需要直接访问它的4个相邻的点。在这些情况中的划分目标是要给每个进程分配数据域中的一个连续的区域，当然要注意负载平衡，将数据域安排成适当的形状，以至于进程所需的信息大多数可在它自己的所分的划分中得到满足。如图3-4所示，在许多这样的情况下，一个进程的通信需求和划分的边界的大小成正比，而计算量和整个划分成正比。于是通信计算比为表面积和体积之比（在三维情况下），或者是周长和面积之比（二维情况下）。通过增加数据集的规模（图中为 n^2）或者是减小处理器数（p），这个比可以减小。

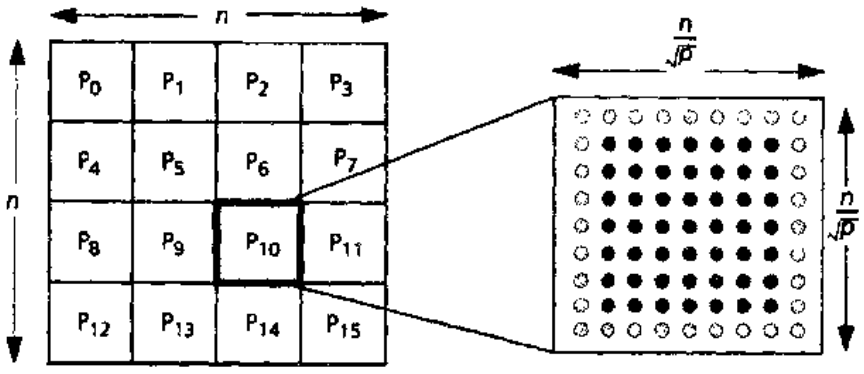


图3-4 在二维域分解中，通信量相对于计算量表现出的周长对面积的关系。所示的例子针对局部的算法，最近相邻信息交换算法类似于简单的方程求解器内核。网格上的每个点需要来自它相邻4个最近点的信息。这样，在处理器 P_{10} 的划分中的内部点不需要和划分外的任何点通信。处理器 P_{10} 的计算于是就和所有 n^2/p 个点的计算之和成比例，而通信和边界点的个数（即 $4n/\sqrt{p}$）成比例

当然，在域分解中理想的划分形状是和应用有关的，主要取决于信息的需求和域中点的

相关计算工作，对方程求解器内核来说，在第 2 章我们选择把网格划分为连续的行形成的块。图 3-5 示出将网格划分为正方形状的子块的情形，这会导致较低的通信计算比。随着处理器数相对于网格大小的增加，这种影响越来越大。因此我们在继续讨论性能的时候用这种划分方法（称为“块状划分”，前面的称为“条状划分”）。作为一个简单的练习，考虑一下如果我们将行以一种交替或循环的方式分给进程（行 i 分给进程 $i \bmod nprocs$），通信计算比会是什么。

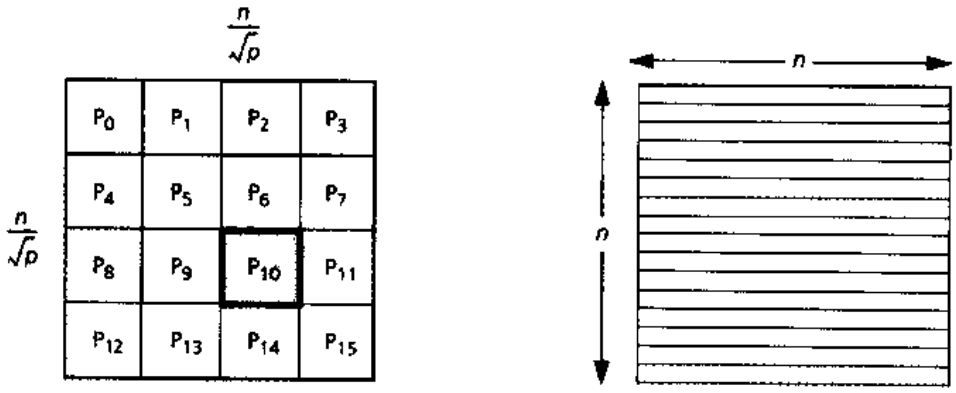


图 3-5　在一个规则的二维网格上，针对一种简单的近邻计算，可能有不同的域分解方案的选择。由于每个网格点的工作量是一样的，同样大小的划分给出好的负载平衡。但我们还可以有不同的选择。我们可以将网格元素划分成连续的条状的行（右边）或者尽量接近正方形的块状（左边）。于是在块状分解中周长对面积的比（即通信计算比）为

$$\frac{4 \times n/\sqrt{p}}{n^2/p} \text{或} \frac{4 \times \sqrt{p}}{n}$$

而在条状分解的情形是$\frac{2 \times n}{n^2/p}$或$\frac{2 \times p}{n}$。随着 p 的增加，和条状分解相比，块状分解导致的固有通信较少

如何找到一个合适的负载平衡的域分解，同时还是低通信的？这取决于计算的性质和它的可预测性，我们可以通过静态或半静态技术来实现。

- *静态，通过观察*。如同 Ocean 例子中的方程求解器内核。这要求计算的可预测性，并且通常导致规则形状的划分，如图 3-4 和图 3-5 所示。
- *静态，通过分析*。计算和通信的特点可能不仅取决于输入的规模，还取决于在运行时表现给程序的输入的结构，这就要求对输入进行分析。然而，划分可能只需要在分析输入后在实际计算开始前，一次性完成，因此我们仍然将这种情况说成是静态的。在航空和车辆模拟的应用中，我们常常在稀疏矩阵上进行计算，其中要做的划分就是这样的例子：矩阵的结构是固定的，但高度地非规则，因此要求复杂的图划分。另一个例子是数据挖掘。这里，我们可以将交易的数据库在进程之间静态地划分。由于和不同的物品集合相关的工作量是不相等的，数据库中记录的集合要在进程之间得到一种平衡分配就需要某些分析。一种简单的、通过观察将物品集合和数据库静态分配的结果能保持通信低，但不提供负载平衡。
- *半静态，周期性的重划分*。这种情况前面讨论过，像 Barnes-Hut 那样的应用，其特征是计算空间随时间缓慢变化。域分解对于减少通信仍然是重要的，我们在 3.5.2 节中的基于性态的 Barnes-Hut 案例分析时将会看到。
- *静态或半静态，动态任务窃取*。即使当计算高度不可预测，必须用动态任务窃取时，

域分解在开始将任务分到进程时可能是有用的。Raytrace 就是一个例子。这里有两个区域：要被表现的三维场景和二维图像平面。由于自然的任务是通过图像平面的光线，管理平面的区域划分要比管理场景本身容易得多。我们划分图像平面，很像方程求解器内核中的网格（如图 3-4 所示），图像的像素对应于网格中的点，将那些光线初始分给相应的进程。这是有用的，因为通过相邻像素上的光线趋于达到大量相同的场景数据。进程然后动态地窃取光线（像素）或者是光线组，以达到负载平衡。

132 ~ 134

当然，将计算空间划分成连续的子域给每一个进程的方法，并不是对所有应用性能都是适合的；习题 3.9 所示的高斯消去法就是一个例子。即便对于 Raytrace，将图像分成比处理器数多的块，以一种交叉的方式分给处理器可能更有利，这里的权衡在于增加了通信，但动态负载平衡可能会更好些。另外，同一应用的不同计算阶段也可能希望不同的划分。如上所述，尽管可用的技术是方方面面的，但还是存在像域分解这样的公共原理。例如，即使是在动态性很强的应用中为了负载平衡做任务窃取，我们可以通过每次以同样的次序来搜索其他队列来减少通信，或者一次窃取较大的任务或者多个任务来减少访问非本地队列的次数。

除了减少并行程序中的通信量，在处理器之间保持通信的平衡（不仅是计算平衡）也是重要的。由于通信是高代价的，通信的不平衡可能直接转换为处理器之间执行时间的不平衡。总之，在划分中的权衡到底倾向于负载平衡还是通信量，取决于在给定系统上的通信代价。如果将通信作为一个显式的性能代价参数，我们可进一步细化加速比上界为：

$$\text{加速比}_{\text{问题}}(p) \leqslant \frac{\text{串行工作量}}{\max(\text{工作量}+\text{同步等待时间}+\text{通信代价})}$$

比较先前的表达式，这个式子将通信从工作量中分离开来。工作量现在只包括执行的指令数加上本地数据访问的代价。

并行程序中的通信显然对体系结构有重要的影响。事实上，系统结构师考察应用的需要，来确定什么样的通信延迟和带宽值得花钱来解决（见习题 3.14）。例如，由一个机器所提供的带宽通常能够通过增加硬件（意味着增加钱）得到增加，但是否值得取决于应用是否经常会用到这些带宽。作为系统设计师，我们假设提交给我们的程序在负载平衡和通信要求方面是合理的，于是我们努力通过提供必要的支持来使它们的性能更好。让我们现在分析最后一个能够在划分阶段解决的，而不需要涉及底层体系结构的算法层次的问题。

3.1.3 减少额外的工作

前面关于域分解的讨论告诉我们，当一个计算是非规则时，计算一个好的分配，既提供负载平衡又减少通信的代价是相当高的。这种额外的工作在串行执行时并不需要，它是并行性的一种开销。考虑稀疏矩阵例子，前面在解释通过分析方法进行静态划分时讨论过。稀疏矩阵能够表示为一个图，每个节点代表矩阵的一行或者一列，如果矩阵的（i，j）元素非零，则在节点 i 和节点 j 之间存在一条边。划分的目标是分给每个进程一个节点集合使得计算是负载平衡的，同时跨划分边界的边数尽量少。人们已经研究出了许多聪明的划分技术，力求在负载平衡和降低通信需求方面取得较好的折中。通常，效果较好的划分算法本身的执行时间则会较多。在本章的后面，我们会看到这一点在 Barnes-Hut 中得到进一步体现。

135

除了划分，额外工作的另一个常见的来源是冗余计算：多个进程冗余地计算数值，而不

是由一个进程计算然后传给其他进程；在通信代价高时，这种情况也许是较好的折中。这样的例子包括，在计算机图形学应用中所有进程计算相同的投影表为自己所用，在科学计算中计算三角函数表也是这种情形。如果冗余计算的完成是在处理器否则就会空闲（由于负载不平衡）的时间内，其代价就可隐藏起来。

最后，协调并行程序的许多方面也涉及到额外的工作，诸如创建进程、管理动态任务、在机器上分配代码和数据、执行同步操作和并行控制指令、针对一个机器将通信适当地结构化、将通信的数据打包并从通信的消息中解包数据等等。例如，创建进程的高代价使得我们在程序的最开始创建一次，然后让它们执行任务直到结束，而不是由一个主进程随着代码的并行段落的进入和退出，随时创建和撤销进程（创建和归并 fork-join 方式，有时在轻量级线程情况下用到）。例如，在数据挖掘案例分析中（3.5.4 节），有大量的额外工作来做数据库的变换，以减小通信、同步和昂贵的输入/输出活动。

当作出划分决定时，我们必须仔细考虑在额外工作、负载平衡和通信之间的折中。通过使通信和任务管理更高效，系统结构能够帮助减少对额外工作的需要。基于这些算法层次的划分要素，加速比上界现在能表达为：

$$\text{加速比}_{\text{问题}}(p) \leqslant \frac{\text{串行工作量}}{\max(\text{工作量}+\text{同步等待时间}+\text{通信代价}+\text{额外工作})} \tag{3-1}$$

3.1.4　小结

要分析并行算法的性能，需要考虑多处理器系统特征和并行算法的特征两个方面。在过去，并行算法的分析集中在算法方面，例如划分和对拓扑结构的映射，没有考虑其他体系结构的相互作用。事实上，人们常用所谓并行随机存取存储机器（PRAM）（Fortuen，Wyllie 1978）模型来表示多处理器系统，并基于它来做算法分析。在 PRAM 最基本的形式中，假设数据的访问是没有代价的，不管它是对本地数据还是涉及到通信。也就是说，在加速比表达式中，通信代价为 0，工作量就是所执行的指令数：

136

$$\text{加速比-PRAM}_{\text{问题}}(p) \leqslant \frac{\text{串行指令数}}{\max(\text{指令数}+\text{同步等待时间}+\text{额外指令})} \tag{3-2}$$

这样的 PRAM 模型可以自然地对应共享地址空间的机器，其中所有的数据访问是没代价的。此时，在算法分析中有意义的性能要素是负载平衡（包括串行化）和额外工作。PRAM 算法设计的目标是揭示足够的并发性，从而使工作负载能很好地平衡而不引起太多的额外工作。

PRAM 模型对于发现一个算法中的并发性是有用的，这是在并行化中的第一步，但它不是在真实的并行系统上性能的一个实际的模型。这是由于 PRAM 模型忽略了通信，而通信很容易在并行执行代价中起支配作用，在通信上的不平衡能够导致指令执行的不平衡。事实上，在分析算法的时候不考虑通信会很容易导致不良的分解和分配选择，对协调效果的影响更大。最近人们开发的一些模型将通信代价作为一个显式的参数，算法设计人员能用它对算法的性能有更实际的评价（Valiant 1990；Culler et al. 1993）。在对通信代价有了更好的理解后，我们将回到这个话题上来。

本节对通信代价的处理方式，相对于真实系统来说从两个方面做了简化。首先，并行程序中固有的通信和它的划分不是仅有的重要通信形式：由于程序和它赖以运行的系统结构的

相互作用，大量非固有的或人工的通信可能发生。这样，我们到现在为止还没有对并行程序产生的通信量建立起一个满意的模型。再者，在公式（3-1）中的通信代价不仅是由所引起的通信量（不管是固有还是附加）所决定的，还和程序中通信的结构和它与机器中基本通信操作的代价的相互作用有关。附加通信和通信的结构是重要的性能要素，由于它们是和体系结构有关的，通常在协调步骤中讨论。为了理解它们，我们需要对并行系统结构和并行软件的某些关键的相互作用有更深的理解。

3.2 在多存储器系统中的数据访问和通信

在关于划分的讨论中，我们将多处理器系统看成一个相互协作进程的集合。然而，多处理器系统也是多存储器系统、多高速缓存系统。系统这些组成部分的作用对程序执行性能是十分重要的，并且其重要性与程序设计模型无关（尽管程序设计模型可能影响某些性能折中考虑的特性）。下面的讨论将主要集中在与这种多存储系统中的数据访问有关的问题上，看它们是如何影响性能的。在这里，我们从一种不同的角度来看多处理器系统。

137

3.2.1 看作扩展的存储层次结构的多处理器系统

从单个处理器的角度，可以将机器中所有的存储器（包括其他处理器中的高速缓存）看作一种扩充的存储层次结构。通信体系结构将这个层次在不同节点上的部分连到一起。在一个单处理器系统上，不同层次的存储设施的相互作用（例如，高速缓存大小、关联度、存储块大小）能引起对某些数据的访问比另外一些要快得多，还可能引起数据在不同层次之间的传送，其规模比程序本身要求的要多。类似地，在多处理器系统，这种扩展存储层次结构中的相互作用能引起较多通信（数据跨网络传送），也比并行程序本身所要求的多。由于通信是有代价的，在这种扩展存储层次开发数据的局部性就显得特别重要，既改善节点性能又减少节点间额外通信。

即使是在单处理器系统，处理器的性能很大程度上也是依赖于存储层次的性能。高速缓存的作用是极端重要的，不考虑它的影响，讨论性能就没有什么意义。我们可以通过完成一个程序的时间来看一个系统的性能，它包含两个成分：处理器忙于执行指令的时间和它等待来自存储系统的数据的时间。（输入/输出活动可以和数据访问一起考虑或者单独处理）

$$\text{程序执行时间}(1) = \text{CPU 繁忙时间}(1) + \text{CPU 等待数据访问时间}(1) \qquad (3\text{-}3)$$

作为系统结构设计师，我们经常将这个公式规格化，其方法是用执行的指令数除以各项，用时钟周期数来度量时间。于是我们就有了一个方便的、面向机器的性能度量指标，即每条指令所需的周期数（CPI），它由理想 CPI 加上每条指令数据访问停滞周期的平均数构成。在现代微处理器中，每周期发送 4 条指令，程序中的依赖关系可能将平均发送率限制到每周期 2.5 条指令或者说理想 CPI 为 0.4。如果这些指令的 1%引起缓存扑空，一个扑空平均引起 80 个周期的停滞，那么这些停滞将导致每个指令增加额外的 0.8 个周期。这样，处理器只是在 1/3 的时间里忙于做有用的工作！当然，另外的 2/3 时间事实上是“有用”的：它是花在和存储系统通信访问数据的时间。认识到这种数据访问的代价，我们就要优化程序或机器更高效地完成数据访问。例如，我们可以改变程序的数据布局以增加时间或空间局部性，或者我们提供更大的高速缓存或某种包容延迟的机制。

138

在多处理器中，一种理想的看待这种扩展存储层次的观点是，局部高速缓存层次连接到下一层次的一个中央存储器上。实际上，情况要复杂些。即使是有集中共享存储器的机器，在本地高速缓存外是多模块的存储和其他处理器的高速缓存。如果是物理上分布的存储器，主存的一部分也是局部的，大部分是远程的并且对一个处理器是远程的，对另一个则是局部的。

程序设计模型的不同反映出层次结构管理的不同。我们总是假设处理器中的寄存器由编译器来管理。我们也总是假设前几级高速缓存总是由硬件透明地管理。在共享地址空间模型中，数据在远程节点和本地节点之间的移动对用户程序来说也是透明的。无论管理情况如何，接近于处理器的存储级别提供较高的带宽和较低的数据访问延迟。同样，在这里我们可以通过改进扩展的存储结构或者程序的局部性来改善数据访问的性能。

开发局部性暴露出一种并行性的折中，类似于减少通信。并行性可能引起更多的处理器访问相同的数据，将数据移到每个处理器中，而每个处理器期望它自己的数据放在靠近它的地方。高性能并行程序除了很好的并行化外，需要从每个处理器获得高性能（通过在扩展存储层次中开发局部性）。

3.2.2 在扩展的存储层次结构中的附加通信

在这种扩展存储结构中，不能被本地（本节点）满足的数据访问产生通信。固有通信是其一部分：通过显式消息或者是读写操作，数据通过这个存储结构从一个处理器移动到另一个处理器。然而，在程序实际执行中所发生的通信量通常大于并行算法中的固有进程间通信量。额外的通信是人为的，来源于程序实现的具体情况和它和机器的扩展存储层次的相互作用。这种人为的通信有许多来源：

- *数据存储分配得不好*。一个节点要访问的数据可能分配在另一个节点的局部存储器中。无论数据是否被其他节点修改，对远程数据的访问引起通信。消除这样的数据传送可能有不同的方法，例如更好的任务分配或更好的数据分布或将数据复制在本地。
- *传送中的非必要数据*。在一次传送中，可能实际通信的数据多于所需的。例如，由于发送方不能精确确定要送什么数据，可能会保守地多送些数据，于是接收方可能用不了一条消息中的所有数据。类似地，如果数据以大于一个字的单位（例如，高速缓存数据块）隐式传送，块的一部分可能对请求者没用。这种附加通信可以通过较小的传送来消除。
- *由于其他系统粒度带来的不必要的传送*。在缓存一致性机器中，数据保持一致性的典型粒度大于一个字，为了保持数据的一致性可能会导致附加通信，在后面的章节我们将要讨论这个问题。
- *冗余数据通信*。数据可能多次通信（例如，每当数据值变化时），但只是最后的值被实际用到。在另一方面，由于难以确定，数据可能通信到已经有了最新值的一个进程。
- *有限的复制能力*。当数据要被进程多次访问时，通信的数据通常在本地复制以避免重复的通信。然而，在一个节点上复制的容量是有限的——不管是在高速缓存还是在主存——因此已经从进程A到进程B通信的数据可能在B的局部存储系统被替换掉，因此需要再次传送，即使A并没有修改也是如此。

139

相比之下，固有通信指的仅是程序本身所要求的数据传送，就好像是本地复制容量是无限的并且对于共享的数据已被更新或已被传送具有完整的知识。当我们深入讨论体系结构

时，将会对附加通信的来源有更好的理解。最后让我们再看看最后一种附加通信的来源（有限复制能力）它有特别深刻的影响。

3.2.3 用工作集的观点看附加的通信和数据的复制

在并行系统中，有限复制容量和附加通信的关系是相当基本的，就好像是单处理器系统中高速缓存大小和存储访问量的关系；只讲通信量而不提到数据复制的容量几乎是没有意义的。扩展存储层次的观点可用于分析这种关系。我们可以将多处理器看成是一个三层存储结构：局部高速缓存访问最快，本地存储差一些，任何远程存储要慢得多。我们可以认为每一层都是高速缓存，不管它是实际由硬件管理的，还是由系统或应用软件管理的。然后我们就能谈论任何一级的“扑空”，从而产生对下一级的流量，就像我们对单处理器那样。在任何一级，这流量的一部分源于冷启动失效，源于处理器对数据的首次访问。这一部分，在单处理器中也称为骤起流量，是独立于高速缓存大小的。这样的冷启动扑空随着程序的执行逐渐减小。然后是由于容量扑空的流量，它显然随高速缓存增大而减小。流量的第三部分可能是冲突扑空，它的减小要靠更大的关联度、更多的高速缓存块或者改变数据访问的模式。这三种扑空或流量在单处理器体系结构中称为 3C——冷启动、容量和冲突。在多处理器系统中，有第 4 个 C，即通信扑空，由处理器之间的固有通信或者某种前面所讨论的附加通信引起。如同冷启动扑空，通信扑空不随高速缓存大小的改变而减小。取决于空间局部性的情况，数据传送的粒度可能从正反两方面影响流量的每一个部分。

140

如果我们要确定当复制能力（即高速缓存大小）在相应层次增加，因为各种扑空并行程序执行产生的流量，我们可以得到如图 3-6 所示的曲线。该曲线有几个拐点。这些拐点对应于算法的工作集，它们和存储层次结构的对应级别相关。[⊖]对于第一级高速缓存，它们是算

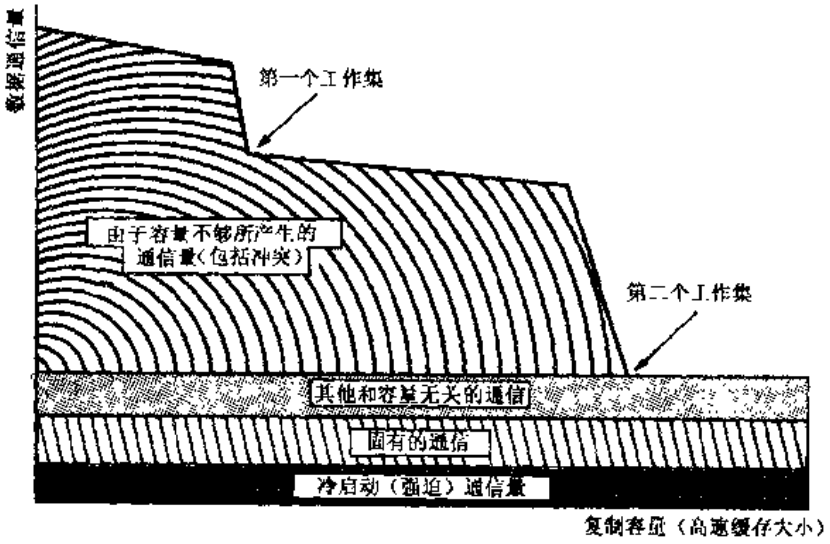


图 3-6　随高速缓存规模变化，在它和系统其他部分之间的数据流量，以及数据流量的不同成分。在总体流量曲线上的拐点指出了程序的工作集

⊖ 程序行为的工作集模型（Denning 1968）是基于程序数据引用的模式所表现出来的时间局部性。在这种模型下，一个程序（或者说是并行程序的一个进程）有一个数据集，它会在一段时间内重复使用后，才移动到其他的数据集上。在数据集之间的迁移可能是突然的，也可能是逐渐的。不管是哪种情况，在大多数时间下都有一个数据的“工作集”，处理器应该能够将其维持在存储层次体系的快速层上，来有效地利用它的速度。

法本身的工作集；对于其他层次，取决于其他层次是如何将数据引用过滤下来的，还取决于该层次是如何管理的。第一级缓存的曲线（假定全关联，1 个存储字的块大小）是算法的工作集曲线。

从任何这些扑空中产生的流量可能引起跨越机器互连网络的通信，例如，如果下一级存储恰好在远程节点上。类似地，如果这下级存储在本地的话，任何类型的扑空对本地流量也有影响，引起本地数据访问的开销。这样，可以期望许多用于减少附加通信的技术和单处理器上那些用于开发局部性的技术类似。进程运行在不同的处理器上，固有通信几乎总要在机器上产生实际的通信（除了所需的数据已在本地外，后面会见到这种情况）。这些扑空只能由算法中的逻辑共享模式来减少。除此以外，我们还要减少那些源于传送大小和有限的复制能力的附加通信，我们可以通过开发进程在扩展存储结构中的数据访问的空间和时间局部性来达到。改变分配和协调能够明显改变局部性特征，包括工作集曲线的形状。

最后，给定通信量，从处理器看它受通信结构的影响的代价。“结构”在此意味着消息的大小、通信突发的程度、通信代价是否能和其他的计算或通信重叠（这个问题在协调部分讨论）、还有通信模式和互连网络拓扑结构的匹配程度（这将在映射阶段讨论）。无论是固有的或者附加的，减少通信量都很重要，因为它减少了对系统和程序员降低通信代价的要求。前面我们从扩展的层次存储结构的观念了解了机器，以及它所引起的主要问题。下面我们来看如何从软件的角度考虑这些和体系结构有关的问题——即一旦划分完成，如何写出高效率的程序。

3.3　性能的协调

我们从讨论开发时间和空间局部性以降低附加的通信量开始，然后是通信的结构化，包括固有的和附加的，以减少通信开销。

3.3.1　减少附加通信

在消息传递模型中，通信和复制都是显式的，即使是附加通信，也是显式地编码在程序中。在共享存储地址空间中，由于附加通信隐含地发生在程序和机器组织之间，从体系结构的角度来看它们更令人感兴趣。特别值得关注的是有限的高速缓存大小和数据存储分配，通信和保持高速缓存一致性的粒度。我们因此用一个共享地址空间来解释开发局部性中的问题，其目的既在于改进节点性能又在于减少附加通信。

141 ~ 142

1. 开发时间局部性

称一个程序具有时间局部性，如果它在一段短时间里倾向于重复访问相同的存储单元。给定一个存储层次结构，开发时间局部性的目标是要构造算法使它的工作集和该层次结构的不同层次有很好的映射。对程序员来说，这通常意味着追求小的工作集，同时也不能小到因其他原因损失性能。有多种用于减小工作集的技术，其中之一是和减小固有通信相同的（将那些倾向于访问相同数据的任务分配到相同的处理器）这进一步说明了通信和局部性的关系。一旦分配完成，分给一个进程的计算任务就可能组织成使访问相同数据的任务在时间上调度得近一些，从而尽可能地重用数据，然后再移到其他数据，而不是在不同的数据块之间来回改变。

在同一个计算阶段中，当多种数据结构被访问时，我们必须决定哪一个是最重要的时间

局部性开发的对象。由于通信比局部访问代价要大，我们可能更应该开发非本地数据的时间局部性。考虑一种数据库应用，其中一个进程要将它的某种类型的记录和其他进程的所有记录相比较。这里有两种选择：1）对每一个它自己的记录，该进程遍历所有其他（非本地）记录，进行比较；2）对每一个非本地记录，进程遍历它自己的记录，进行比较。后者开发了非本地数据的时间局部性，因此可能得到较好的总体性能。例 3.1 针对我们方程求解器的内核讨论了时间局部性。

例 3.1　在方程求解器的内核中，时间局部性开发到什么程度？如何可能增加时间局部性？

解答：　方程求解器内核只是遍历单个数据结构。在一个进程划分内部的一个典型网格元素至少要被该进程在每次遍历时访问 5 次：计算它自己的新值至少要访问一次；计算它 4 个近邻的新值时每个至少访问一次。如果一个进程以行优先的方式遍历它的网格划分（即一行一行地从左到右进行遍历，如图 3-7a 所示），那么 $A[i, j]$ 可在更新同一行和它相关的 3 个元素时保证得到重用：$A[i, j-1]$，$A[i, j]$，$A[i, j+1]$。然而，在算 $A[i, j]$ 和 $A[i+1, j]$ 的新值之间，该进程要访问它的划分中的 3 个整行元素。如果这 3 行不能一次都装在高速缓存中，那么当需要它来计算 $A[i+1, j]$ 的时候，$A[i, j]$ 就不会在高速缓存中。

143

如果数据的后援存储是非本地的，就要导致附加通信。这个问题可通过改变元素计算的次序来改善，如图 3-7b 所示。从根本上讲，一个进程从左至右推进，不是遍历进程所得划分的整行的长度，而是只到某个长度 B，然后就移到下一行对应的部分。它遍历的方式通过每次遍历它的 $B \times B$ 子块来完成。块的大小 B 的选择在于划分中至少 3 个 B 长度的行能放到高速缓存中。当然，这改变了元素更新的顺序，从而也就可能改变收敛的性质（如果用红黑序，则可以保持收敛的性质）。■

a）没有分块的遍历模式　　b）B = 4 的分块遍历模式

图 3-7　在方程求解内核中为开发时间局部性的数据成组方式。图中反映了一个进程在一次迭代中遍历其划分的数据访问模式，箭头指向的线段表示网格点被访问的次序。更新颜色加深元素的子行要求对该行的访问，还要求对阴影部分子行的访问。更新下一子行（阴影部分的）第一个元素，要求再次访问颜色加深行的第一个元素，但这 3 个子行（加深的和阴影的）自从上一次加深元素访问时已经被访问过了。通过改变更新次序，成组的访问模式以一定比例改善了数据的重用性

这种技术称为成组技术，它将计算结构化使得它所访问数据的子集能够适应存储层次某

一级别，尽可能地使用那些数据，然后移到数据的下一个这样的子集。在方程求解器内核中，采用成组技术后扑空率的降低只是一个小的常数因子（大约是2）。这个降低只是当网格在进程划分的3个子行在高速缓存中容纳不下时才能发生，因此成组技术不总是有用的。然而，在诸如矩阵相乘或矩阵因子分解等线性代数计算中，使用成组技术非常成功。这样的计算通常有复杂性$O(n^{k+1})$，对应的数据规模为$O(n^k)$，因此每个数据项要被访问$O(n)$次。在这些情形下有效地用这种成组技术，如果组的大小是$B \times B$，则扑空率降低因子为B。这是很有意义的，因为多数数据访问都是非本地的。不足为奇，许多这种类型的重构技术也用来改善顺序程序中的时间局部性：例如，在顺序矩阵计算中，成组对于获得高性能也很关键，如习题3.10所示。用于时间局部性的技术可以用在有数据复制层次结构的任何一级（包括主存）无论是显式复制还是隐式复制。 144

时间局部性和程序中的数据访问模式对于并行体系结构有重要的影响。例如，它们帮助决定系统应该支持哪一个程序设计模型和通信抽象，这是我们在3.6节要考虑的问题。工作集的大小和变化规模，对于在存储层次的不同级别所需的复制容量有明显的影响，对于该层次中有意义的级别数也有影响。在缓存一致性共享存储空间中，工作集的大小和编配（即它们只包括本地数据，还是远程数据，还是两者都有）有助于确定在本地主存复制通信的数据是否有用，或是简单地依赖高速缓存的有用性；如果有用的话，该如何实现。在消息传递系统中，它们帮助我们确定什么数据该复制，如何管理复制。当然，确定存储层次的大小，不仅是个别应用的工作集，而是运行在机器上的整个工作负载和操作系统。对于硬件高速缓存来说，需要存放工作集的高速缓存的大小还取决于它的组织（关联度和块的大小）。

2．开发空间局部性

在我们的扩充存储层次结构中，一个级别和下一个级别以某种*数据传送粒度*交换数据。这个粒度可能是固定的（例如，一个高速缓存块或者主存的一页）也可能是灵活的（例如，显式的、用户控制的消息或者用户定义的对象）。它通常随着我们离处理器越远，变得越大，由于每次传送的延迟和固定的启动代价变得越来越大，因此应该在较大数据量中分摊。为了开发较大的通信和数据传输粒度，我们应该按照开发空间局部性的原则组织我们的代码和数据结构。[⊖]如果不这样做，就可能导致附加通信，例如传送是在远程节点之间发生并且是隐式的（以某种固定的粒度），如同共享存储系统那样。即便传送是显式的并且大小由用户来确定，不好的空间局部性可能导致代价更高的通信，因为较小的消息可能要被发送或者数据在被发送之前要被打包成连续的。如同在单处理器一样，差的空间局部性也可能导致高的TLB扑空频率。 145

在共享地址空间中，附加通信也可能源于空间局部性和另外两种重要的粒度的不匹配。一个是*存储分配粒度*，它是数据在本地存储器或者复制空间中*分配的粒度*（例如，主存中的一页）。这确定了数据能在物理的主存中分布的粒度；即，当数据通过操作系统以页为粒度分配时，我们不应该将一页的一部分分配到一个节点的存储器中，而另一部分分配到另一节点的存储器中。假如有两个经常被两个处理器访问的存储字落到了同一页上，这一页只能在一个处理器的局部存储器中；在这种情况下，容量或者冲突造成的缓存扑空对于另一个处理

⊖ 空间局部性原理讲的是：如果一个存储单元现在被访问，那么和它在地址空间上接近的存储单元很可能在最近要被访问。很清楚，在存储字的粒度上表现出来的空间局部性也可以看成是在高速缓存块或更大粒度上的时间局部性；即如果现在访问一个缓存块，那么在最近很可能还要访问它。

器要访问的字将产生通信。另一个重要的粒度是一致性粒度，其中在一致性共享地址空间中不相关，但碰巧都位于相同的一致性单元的字也可能引起附加通信。这个问题称为伪共享，将在第 5 章进一步讨论。

在共享地址空间中，类似于那些用在单处理器中的技术，开发空间局部性的所有这些方面所用的技术都可用，但有一个新的方面：我们应该力图保持一个处理器访问的数据在地址空间上靠近（连续），而不同处理器的数据分开。讨论共享地址空间中的空间局部性问题最好的方式是和某个特定的体系结构结合起来，我们将在第 5、8 章有这样的讨论。这里，作为说明的例子，我们看在方程求解器内核中，数据可以怎样重构，以便和存储分配的粒度相互作用得更好。

例 3.2　考虑一个共享地址空间的系统，其中主存物理上分布在节点上，主存分配的粒度是页（比如 4 KB）。假设一个给定的页只是在一个节点的存储器中分配。现在考虑用于方程求解器内核中的网格。这个存储分配的粒度引起了什么问题，它如何解决？

解答：在共享地址空间中，如同在串行程序，表示一个二维网格的自然数据结构是一个二维数组。在一个典型的程序设计语言中，一个二维数组数据结构以行优先或者列优先分配存储器。[1] 在图 3-8a 中从左至右的箭头表示以行优先方式分配时虚拟地址的连续性，我们下面也假设这种情况。由于和串行程序中使用的数据结构相同，一个二维共享数组在编程中有优势，但它在物理的分布存储器的机器上和存储器分配粒度相互作用很差。

146

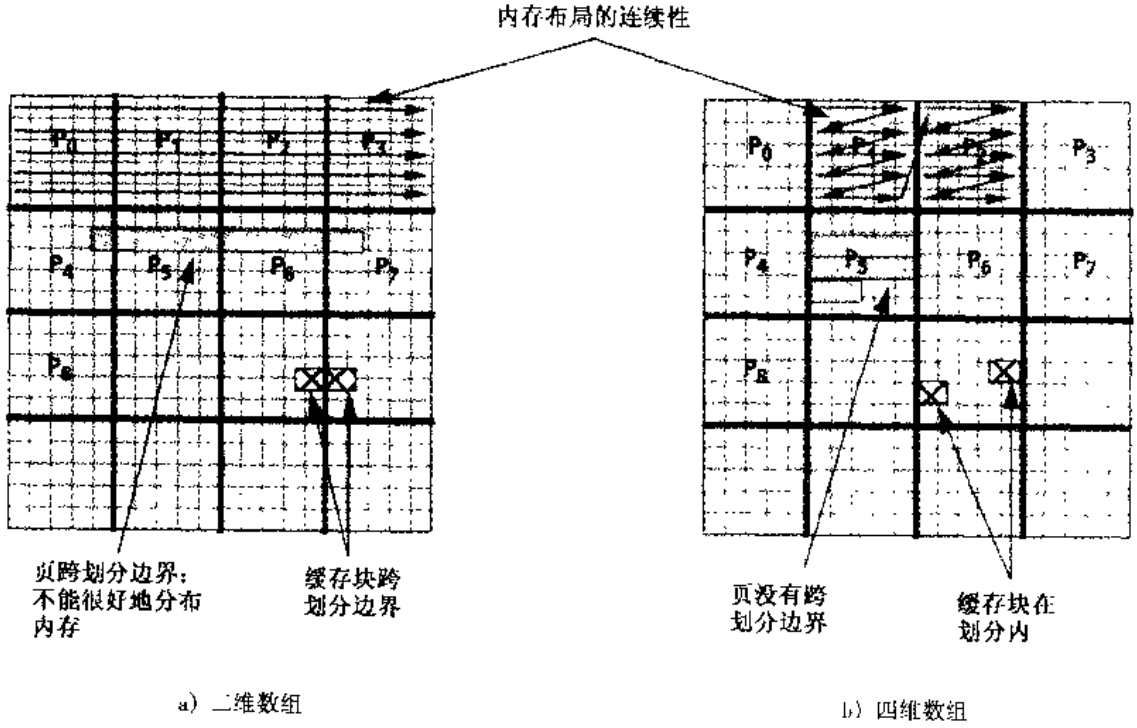


图 3-8　在共享地址空间中，二维和四维数组表示一个二维网格的情况

[1] 考虑对应一个二维矩阵的数组，第一维是行编号，第二维是列编号。行优先分配意味着第一行的所有元素在虚拟地址空间上是连续的，跟着是第二行的所有元素等。我们这里假定的 C 语言是行优先语言。列优先语言的一个例子是 Fortran。

考虑图3-8a中处理器P_5的划分。处理器的一个重要的工作集是它的整个划分，在每次遍历中扫过并且在不同的遍历中重用。如果它的划分不能装入处理器的高速缓存层次，我们希望它能放在本地存储器中，从而失效可以在本地得到满足。这里的问题是，划分中相继的子行在地址空间中不连续，由网格的整行长度分开（包含其他划分的子行）。如果一个划分的子行或者小于一页，或者不是页大小的倍数，或者和页的边界没有很好地对齐，就不可能在节点的存储器之间适当地分布数据。从两个（或多个）相邻划分来的子行将在同一页中，最好的情况不过是分配到有关处理器的某个本地存储器中。如果一个处理器的划分没法装入它的高速缓存中或者引起冲突扑空，每当访问在它自己划分中的网格元素时，就可能要通信，因为划分被分到别的处理器内存中了。

在这个例子中的解决方案中，用了一个高维数组来表示二维的网格。最常用的例子是用一个四维数组。在这种情况下，进程在概念上被安排成二维划分网格，如图3-8b所示。头两个下标指定划分或者说相关的进程；后两个下标表示在该划分中的子行和子列。例如，如果整个网格的大小为1 024×1 024个元素，有16个进程，那么每个划分将是大小为$1\,024/\sqrt{16}\times 1\,024/\sqrt{16}$的子网格，或者说有256×256个元素。在网格的四维数组表示中，数组有4×4×256×256个元素。这种高维表示的关键性质是每个进程所得的256×256个元素的划分现在在地址空间中是连续的（见图3-8b中虚拟存储空间的布局）。数据分布问题现在只能发生在整个划分的尾端，而不是每一子行，而且如果数据结构和页的边界对齐的话，根本不会有问题。不过，用高维数组写程序要复杂得多，特别是在邻近计算的情形，确定相邻进程划分中数组的下标（见习题3.16）。■

147

在针对空间局部性的数据结构设计上，更复杂的应用和数据结构可能显示出更重要的折中，这将在后面的章节中讨论。

进程访问模式中的空间局部性以及它们如何随问题规模和处理器数变化，在共享地址空间体系结构中，影响着所期望的各种粒度的大小——特别是存储分配的粒度、数据传送的粒度和一致性粒度。在消息传递系统中，它也影响是对小消息还是大消息提供支持的相对重要性。出于分摊硬件和传送代价的考虑，使我们倾向于用大的粒度，但太大的粒度也会引起性能问题，其中许多是与多处理器有关的。最后，访问的空间局部性影响高速缓存中冲突失效的出现。由于当下级存储器是非本地时，冲突扑空能产生附加通信，多处理器也使我们倾向于用更大的缓存关联度。对于多处理器的数据局部性支持来说，有许多代价、性能和可编程性之间的权衡，我们的选择最好是由应用的行为来引导。

最后，有趣的权衡经常出现在算法划分目标、实现因素和系统结构的相互作用之间；是那些相互作用产生了附加通信。这说明我们需要仔细考察权衡来获得在一个体系结构上最好的性能。让我们在例3.3中，用方程求解器内核来解释这种情况。

例3.3 基于到目前为止我们讨论的性能要素，对方程求解器内核来说，我们应该选择什么样的划分方式：块状划分还是条状划分？

解答： 如果我们只是考虑固有通信，我们已经知道块划分要优于条划分（见图3-5）。然而，即便只是简单的、二维的数组表示，条划分也有优点，它保持划分在地址空间上的连续。因此，它没有和空间局部性和机器粒度的相互作用有关的问题，例如前面谈到的存储分配粒度。当然，在块分配时遇到的这种特别的相互作用可以通过用高维数组的方式来解决。然而，一种更难解决的相互作用和通信的粒度有关。在子块分配中，考虑一个来自另一划分

148 的相邻元素，处于面向列的划分边界（见图 3-9）。如果通信的粒度大，那么当一个进程引用这个来自相邻划分的元素时，它将不仅取得那个元素，还将取得和它在同一通信单元的若干元素。不管是用二维还是四维表示，那些元素都不是本进程的相邻元素，因此是无用的而且浪费通信带宽。对于条状划分，没有面向列划分的边界；所引用的非本地元素仍然会引起它所在的行的其他元素被取过来，但现在这些元素的确是本进程划分的相邻元素。因此它们是有用的，并且事实上这种大的通信粒度导致了有价值的预取效果。总的来看，有许多应用和机器参数的组合，块划分所带来的性能损失（由于附加通信）要大于减少固有通信所带来的利益。我们可以想像条状划分在大多数情况下比二维数组的块划分性能会好一些，但它在有些情况下也比四维数组要好（在条状划分时，没有用四维数组的道理）。这样，附加通信可能使我们转回去修改划分方法——从块到条。图 3-10a 示出 Ocean 应用在 Origin2000 上的这种效果。通信的粒度越大、代价越大，则效果越明显。用软件支持共享地址空间模型的系统就是这种情形。图 3-10b 用方程求解器内核和更大的网格，表现了数据摆放的影响。注意，如果是按列做条状划分（数据保持行优先存放）而不是行，则所得到的是最差的情形。■

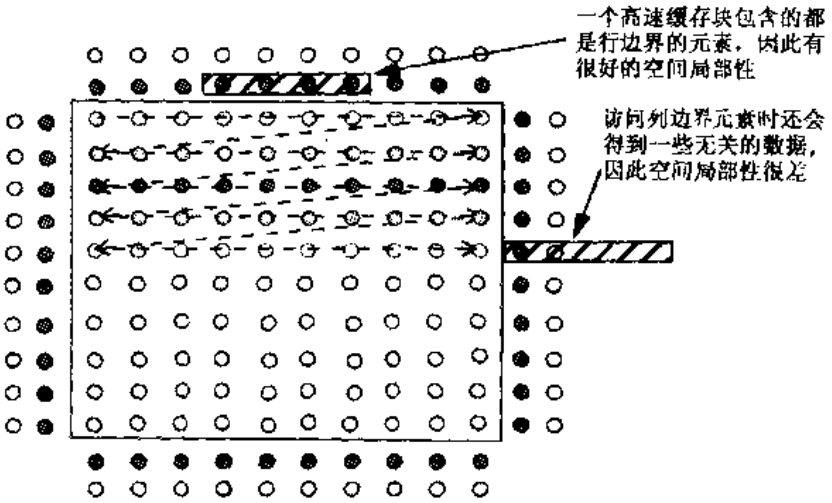


图 3-9 在方程求解器内核中访问非本地数据的空间局部性。图中只表示了一个处理器的划分和它周围的区域。阴影点是拥有该划分的处理器要访问的非局部点。抽取出来的矩形是高速缓存块，表示沿着行边界有很好的空间局部性，而列边界的局部性不好

149

3.3.2 将通信结构化以降低代价

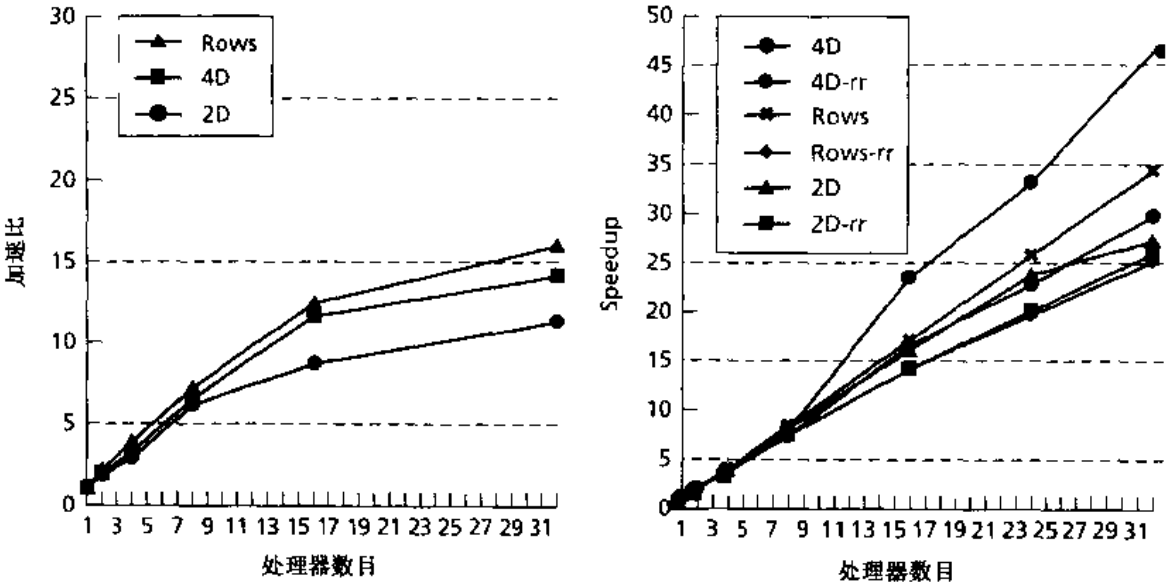
不管通信是固有的还是附加的，它对执行时间的影响取决于其形成消息的组织和结构化。一个小的通信计算比和一个大的通信计算比相比，可能对执行时间有更大的影响，如果后者的结构和系统交互要比前者好得多的话。为在真实机器上得到好的性能，这是一个重要的问题，也是我们要讨论的最后一个主要的性能要素。首先让我们看看通信的结构化是什么意思。

在第 1 章，给定程序发起通信或者消息操作的频率（显式消息或者由读和写操作隐含的消息），我们介绍了处理器看到的通信代价的一个模型。结合式（1-5）和式（1-6），代价 C 的模型是

$$C = \text{频率} \times \left(\text{开销} + \text{延迟} + \frac{\text{消息长度}}{\text{带宽}} + \text{竞争} - \text{重叠}\right)$$

或者是

$$C = f \times \left(o + l + \frac{n_c/m}{B} + t_c - Overlap\right) \tag{3-4}$$



a) 有 514 × 514 个网格的海洋　　b) 有 12K × 12K 网格的方程求解器内核

图 3-10　数据的结构化和空间局部性对性能的影响。所有的测试都是基于 SGI Origin 2000 的。"2D" 和 "4D" 分别表示二维和四维的数据结构，块状分配方式。"Rows" 对应在二维数组上的条状分配方案。在 b) 中，后缀 "rr" 表示数据页以轮循方式在物理存储器之间分配。没有 rr，意味着数据尽量放在分配的处理器的局部存储器中。从 a) 我们看到，条状分配比 2D 块状分配性能要好，这是因为空间局部性和长缓存块（在 Origin 2000 上是 128 个字节）的相互作用所引起的，条状分配甚至比 4D 块状分配还稍好一些，这是由于后者在访问列边界元素时的空间局部性不好。在 b) 中，虽然用了很大胆的通信结构，在主存中合适的数据分布对性能都是重要的，尽管对 2D 数组的块状分配效果不明显。在最好的情形，我们会看到超线性加速比现象，发生在处理器的数量足够并且处理器所得到的划分的规模（其重要的工作集）和它的高速缓存相适应的场合。这种差别在较保守的通信体系结构和较小的复制存储的机器上要大得多

其中 f 是程序中通信消息的频率；o 是在发送和接收处理初始化和消息接收的综合开销，假定没有和其他活动的冲突；l 是消息中第一位到达目的处理器或存储器的没有开销的延迟（假设没有冲突），它包含通过网络接口和辅助电路的延迟，以及网络线路上的延迟；n_c 是程序中数据的总通信量；m 是消息数（因此 n_c/m 是消息的平均长度）；B 是通路上的点到点通信带宽，不算处理器开销（即在第一位之后消息的其余部分到达目的地的速率，假定从源到目的的整个通路看起来是一条没有冲突的流水线）；t_c 是由资源冲突引起的时间；$Overlap$ 是能够被其他计算或其他通信重叠的通信量（即不在处理器执行的关键路径上的活动）。带宽 B 是第 1 章讨论过的总体容量的倒数。它可能受网络链路、网络接口或者通信辅助电路的限制。

关于通信代价的这个表达式可以代换到式（3-1）中，从而得到加速比的最终表达式。括号内的部分是我们的单向、单个消息的模型。如果消息是双向的，我们必须做适当调整。消息的代价（不算重叠部分），也称为时延。除了减少通信量 n_c，我们在结构化通信的目标可能包括 1）降低通信开销（$m \times o$）；2）降低延迟（$m \times l$）；3）降低冲突（$m \times t_c$）；4）让

通信和计算以及其他通信重叠以隐藏它的时延。下面针对各个方面，讨论有关的程序设计技术。

1. 降低开销

由于和初始化或处理一件消息的开销 o 通常是由硬件或系统软件固定的，减少由于通信开销所带来代价的方式就是将若干小消息组成一个大的消息——即减少消息频率。[⊖] 通过指定消息的大小，显式初始化的通信给程序员带来较大的灵活性（回顾 2.3.6 节描述的发送源语）。另一方面，通过读和写操作的隐式通信是程序没法直接控制的，如果有必要的话，系统必须负责将读和写整合成大消息。

150 ~ 151

在规则数据访问和通信模式的应用中，把消息做大是容易的。例如，在消息传递系统中，方程求解器划分成行，用一个消息发送一整行数据。但在那些非规则和不可预测通信模式的应用中，它可能是困难的，例如 Barnes-Hut 或 Raytrace。如我们在 3.6 节所见，它可能要求对并行算法有所改变或者额外的工作来确定哪些数据该合并，这就又要求在这种计算的代价和在开销方面的节省之间做权衡。可能需要用些计算来确定哪些数据该送到哪些进程，数据可能要收集起来，在发送方打包成消息，在接收方解包，散发到存储器的相关位置。

2. 降低延迟

通过辅助硬件和网络接口的延迟可以通过优化这些硬件部件来降低。对此，程序员没有什么可做的。考虑网络传播延迟——即通过网络线路本身的延迟。在没有冲突，且假设消息流水通过网络的情况下，一位通过网络的传播延迟 l 本身能被表达为 $h \times t_h$，其中 h 是相邻网络节点的跳步数或者是消息通过的交换机数，t_h 是通过一个网络跳步的延迟或者时延，包括链路和路由器或者交换机。和消息开销类似，t_h 也是由系统确定的，程序只能在减小 $f \times h \times t_h$ 中的 f 和 h 上做文章。（在存储-转发网络（相对于流水）中，t_h 则是整个消息经过一段跳步的时间，不只是一个数据位）

跳步数 h 可以通过把进程映射到处理器来减少，使得应用的进程间通信的拓扑和网络的物理拓扑对应。如何做到这一点取决于应用程序和网络的结构及其丰富程度；例如，近邻方程求解器内核（以及 Ocean 应用）会和网格互连的多处理器映射得相当好，但和单向环映射的不好。其他几个案例分析在通信模式上表现为非规则的。（我们在第 10 章考察在真实机器中使用的不同的拓扑，讨论它们的权衡。）

152

并行算法映射到网络拓扑的研究有很长时间了，由于人们曾想处理器数 p 会变得很大，差的映射会引起由于 $h \times t_h$ 的延迟在消息代价中起支配作用。拓扑的重要性在实践中取决于若干因素：t_h 项相对于消息出入网络开销 o 的大小；机器的处理节点数，它确定一个给定拓扑的最大跳步数 h；是否这台机器一次只用来运行一个应用，是以批处理方式还是在多个应用中间以多道程序运行。现在，网络拓扑在现代机器中，已不像原来看得那么重了，这是因为现代机器在 3 个方面的特点所导致的：开销的影响大大超过跳步时延（特别是在那些没有硬件支持共享地址空间的系统中），节点数通常并不是很大，机器常用作通用的、多道程序服务器。由于操作系统动态分配资源并且在运行时可能透明地改变进程向处理器的映射，

⊖ 一些显式消息传递系统提供代价不同的消息类型并且为程序提供不同的处理功能。

面向拓扑的程序设计在多道程序系统中可能没什么用。由于这些原因，同分解、分配和协调相比，人们现在对并行化的映射步骤关注不多。不过，随着技术和体系结构的演变，这种情况也可能改变。

3. 减少冲突

多处理器系统的通信机构由许多资源构成，包括网络连接和交换机、通信辅助电路、存储系统和网络接口。所有这些资源都有一种非零的占有度，即它们对任何请求的服务时间（即一个请求占有它的时间）都不是零。换句话说，它们对各种请求服务的带宽（占有度的倒数）是有限的。于是，如果若干消息争用一个资源，某些消息在得到服务的时候，另外一些消息必须等待，这样就增加了消息的时延减少了可用于单个消息的带宽。资源的占有度对消息代价的影响在没有冲突时也是有的；这是因为，消息通过一个资源的时间也是其延迟（或开销）的一部分，而它也可以引起冲突。资源的占有度甚至可能大于通过它的延迟。

冲突是一种特别难以把握的性能问题，这有几个原因。首先，在写并行程序的时候容易将它忽略；特别是如果它是由附加通信引起的，人们更不容易在写程序时考虑其影响。其次，它对性能的影响效果可能是难以捉摸的。如果 p 个处理器同时竞争占有度为 x 的资源，第一个获得该资源的引起 x 时延，最后一个的时延是 $p \times x$。除了会导致处理器有很大的停滞时间外，不同的处理器在停滞时间上的差别也可能导致较大的负载不平衡和同步等待时间。这样，由资源的占有度引起的冲突要比它所表现的非竞争时延危险得多。第三，在一种资源的冲突能影响其他资源，从而阻滞某些请求的处理，尽管那些请求可能并不需要那些被竞争的资源。这类似于多道高速公路上的单道出口冲突的情形，所引起的阻塞是整个高速公路。这种阻塞也影响那些并不需要该出口，只是要继续在高速路上行进的车辆，这是因为它们可能被堵在那些需要通过该出口的车的后面。这些车辆的聚集也占住了其他无关的资源（早先的出口），使它们也没法进入，从而最终堵塞高速公路。严重争用的情况能迅速使整个通信系统结构饱和。争用如此麻烦的最后一个原因是和第三个原因相关的：由于其效果可能表现在程序中和原始原因的不同点，冲突的根源可能特别难以识别（如果通信是隐式的情况更是如此）。

153

网络中的争用可以看成有两类：在网络的链路和交换机称为*网络争用*；在端点或进程节点称为*端点争用*。网络争用，如同网络延迟能通过进程映射以及在网络拓扑上适当地调度通信来减少。端点竞争在许多处理器要同时访问同一个处理器时发生（或者当通信事务和本地内存引用相干时发生）。当这种争用严重时，我们称那个处理节点或者资源为*热点*。让我们考察一个例子，看热点是怎么形成的以及如何通过软件来回避它。

前面讲过，在方程求解器内核中，有进程累计它们的部分和形成一个全局和的情况。形成全局和所导致的冲突可通过树形结构的通信来减少，而不是让所有进程将它们的结果直接送到最后拥有全局和的节点。图 3-11 示出这样一种用二叉树实现的多到一的通信结构。这个树（通常称为软件合并树）的节点是参与的进程。一个叶进程将它的更新结果送给其父节点，父节点将其子节点和自己的更新结果合并，再送给自己的父节点，如此进行直到更新结果达到根节点（拥有全局和的进程），所需步数为 $\log_2 p$ 步。一个类似的扇出二叉树能用来将数据从一个进程送给多个进程。由于其扩放性较好，基于树的方法用途很广，特别是在那些经常经历许多争用的同步源语中，例如栅障和一些其他通信模式的库例程。

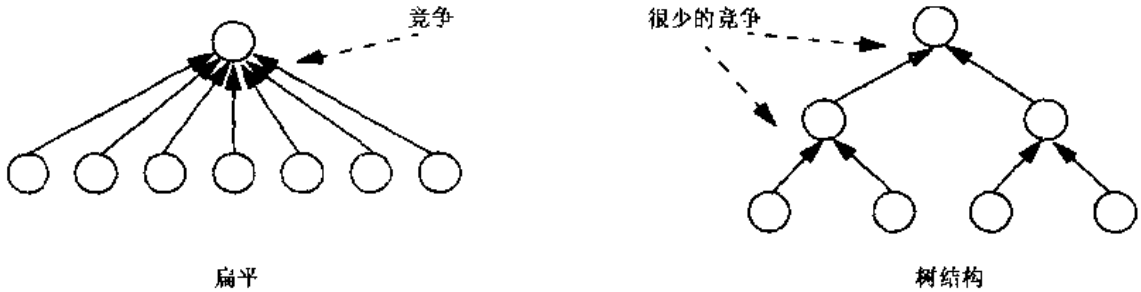


图 3-11 构成多对一通信的两种方式：扁平方式和二叉树方式。注意，在扁平方式目的处理器可能一次要接收 $p-1$ 个消息；而在二叉树方式没有处理器会收到多于两个的消息

通常，从程序设计的角度有两种思路来回避争用，一是避免太多进程和同一个进程通信；二是错开消息到达同一目的的时间，从而不对目的或资源形成太集中的冲击。当通信突发时（即程序在一段时间不通信，然后突然产生较大量的通信），常会产生争用，于是可以安排错开时间来降低突发性。然而，这要和将小消息合并成大消息的优点折中考虑。不幸的是，大消息有增加突发性的趋势。 154

4. 通信和计算的重叠以及其他通信的重叠

尽管我们做了种种降低开销和延迟的努力，第 1 章讨论的技术趋势指出端到端通信延迟相对于处理器周期来说可能依然是很大的。现在的情况已经是，即使在全硬件支持的共享数据空间中用高性能网络，通信延迟也是几百个处理器周期；而在消息传递机器上，这至少要高一个数量级，这是由于软件管理的开销 o 要更大。如果当通信发生时处理器维持一种空闲（停滞）状态，只是那些具有极低的通信计算比的程序才可能有好的并行性能。因此，大量通信的程序必须找出一种方式来隐藏进程关键路径上的通信延迟，方法是让通信和计算或者其他通信尽可能重叠，这样，系统必须提供必要的支持。

隐藏通信延迟的技术有着不同的、经常是互补的风格，我们将在第 11 章讨论。一种简单方法是尽量用较大的消息，通过大消息的流水传送，于是对第一个字有延迟，但隐藏了其余的后续字。另一种方法，我们可以称为预通信，即在需要数据之前就启动通信过程，从而当需要数据时，它可能已经到达了。第三种技术是在程序自然需要时启动通信，但在通信进行时，给处理器找些其他的事情做来隐藏通信代价，这些事情包括进程后面要做的某些计算或者通信。第四种称为多线程，是当通信事件发生时切换到不同的线程或进程。特定的技术和机制取决于通信抽象和所采取的方法，它们都要求程序本质上有比处理器数更高的并发性（也称为松弛度（slackness）），这样才能找到独立的工作来重叠通信延迟。 155

事实上并行体系一直集中在减少处理器看得见的通信代价：通过减少通信开销和延迟、增加带宽、减少占有度、提供机制来分散争用，用计算和通信来重叠通信等。后面的一些章节里用大量的篇幅来讨论这些问题——包括节点到网络接口的设计、通信辅助电路以及减小软件和硬件开销的协议（第 5～9 章）；网络拓扑的设计、原语操作和路由策略（和应用的通信模式很好地匹配）（第 10 章）；设计机制，隐藏处理器看到的通信开销（第 11 章）。激进的体系结构方法通常是昂贵的，因此，要使所带来的性能好处超过其代价，真实的程序能有效地利用它们至关重要。

3.4　从处理器角度看到的性能因素

为理解并行程序中不同的性能因素对体系结构的影响，从单个处理器的角度来看程序执行时间的不同成分是有用的——即有多少时间花在执行指令上；多少时间花在扩展的存储层次中访问数据；多少时间花在协调它和其他处理器的活动。时间的这些不同的部分能够相当直接地和本章所讨论的软件性能问题有关，它帮助我们将软件技术和硬件性能联系起来。这种观点也帮助我们理解并行程序作为表现给体系结构的工作负载的执行过程，这种观念对我们在下一章讨论负载驱动的体系结构的评估是很有用的。

在式（3-3）中，我们描述了花在单处理器上执行一个串行程序中的时间，包括实际执行指令（忙）的时间，停滞等待存储系统的时间（数据访问）；后者是一种“非理想”的因素，降低性能。图 3-12a 示出一个假想顺序程序的性态。在这种情况下，大约 80% 的执行时间花在执行指令上，只能通过改进算法或者处理器来减少。另外 20% 的时间花在存储系统上，可以通过改进局部性或存储系统来改善。

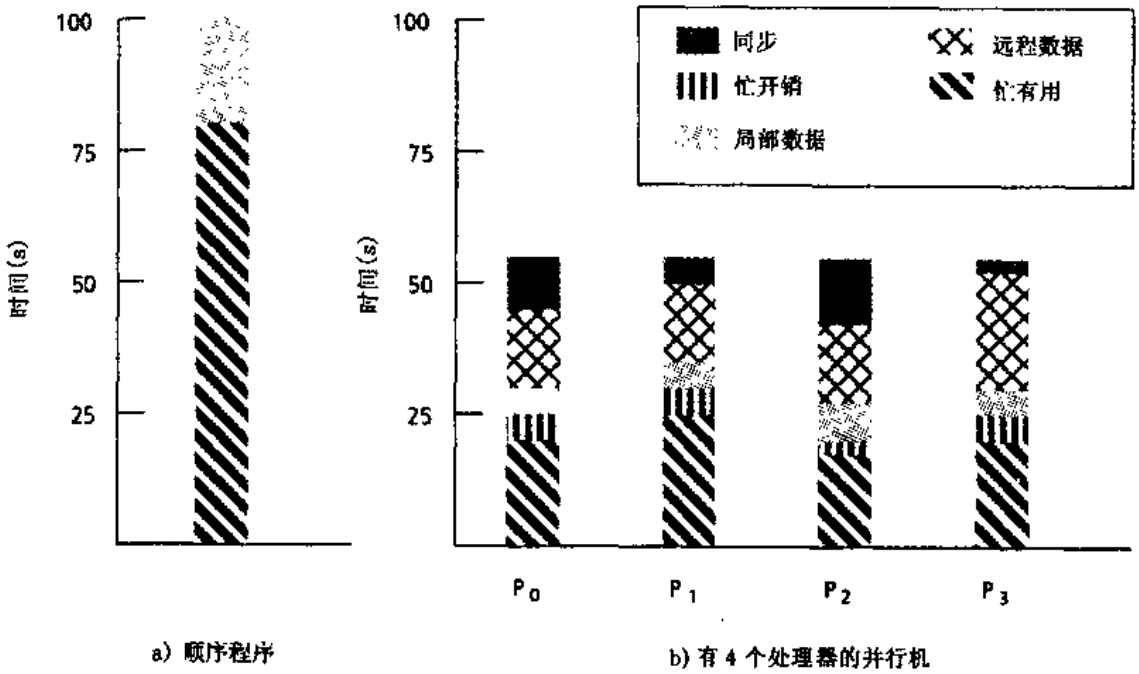


图 3-12　从单个处理器看到的执行时间的各个成分

在多处理器系统中，我们可取一种类似的观点，尽管这种非理想的因素要更多一些。这种观点涉及几种不同的程序设计模型：例如，停滞等待一个接收操作的完成很像停滞等待一个远程读操作的完成或者一个同步事件的发生。如果相同的程序并行化，在一个四处理器机器上运行，这时的执行时间形态看起来可能像图 3-12b 所示的一样。这个图假设在程序结束时有一个全局同步点，这样每一个进程在同一个时间终止。注意并行执行时间（55 s）比串行执行时间（100 s）的四分之一要大；即，我们只获得了 100/55 = 1.8 的加速比，而不是我们曾希望的 4 倍加速比。为什么会出现这种情况？有什么特别的软件或程序设计因素在其中起作用？这些可以通过从单个处理器的角度，考察并行执行时间的成分来确定。在我们通常的分布存储并行系统结构上，并行执行时间有 5 个分量：

- 忙有用。处理器花在执行指令上的时间，那些指令本来在串行程序中也是要执行的。假设一个直接从串行算法中导出的确定性的并行程序[1]，所有处理器的有用忙时间之和等于串行执行的有用忙时间。
- 忙开销。处理器花在执行那些在串行程序中不需要而只是在并行程序中需要的指令上的时间。这直接对应于并行程序中的额外工作部分。
- 数据局部。处理器等待数据引用在它自己的存储系统完成的时间；即，等待的引用不会产生和其他节点的通信。
- 数据远程。处理器等待数据通信的时间或来自其他（远程）处理节点，包括固有通信和附加通信。这代表处理器看到的通信代价。
- 同步。等待其他进程给出某个事件发生的信号，有了该信号，才能推进进程。这包括负载不平衡和程序中的串行化现象，还有实际花在执行同步操作和访问同步变量上的时间。当它等待的时候，一个处理器可能重复检测某个变量的值，直到改变（这就要执行指令）或者它停滞等待，这取决于同步的实现方式。[2]

156 ~ 157

同步、忙开销、远程数据访问分量是由于并行性所引入的开销，而且是在单处理器上运行的串行程序中所没有的。固有通信大多数包含在远程数据分量中，它的某些（通常很小）部分可能也在数据局部分量中体现出来。例如，分给处理器P的局部存储的数据可能要被另一个处理器Q更新，但在P引用它之前，异步地返回到P的存储器中（比如由于从Q中的替换）。在这种情况下，P可能看不到这个通信开销。最后，数据局部分量令人感兴趣，因为它是在串行程序和并行程序中都有的性能开销。对一个固定的问题或者输入数据集来说，开销分量趋向于随处理器数增加，这个分量可能减少：一个给定的处理器只负责整个计算的一部分，于是它可能只访问一部分数据，从而获得较好的高速缓存和存储性能。事实上，如果数据局部分量降低得足够多，它可能产生超线性的加速比，即使是对于确定性的并行程序（超线性加速比意味着比处理器数还要大的加速比）。图3-13概括了并行性因素之间的对应关系，它们被考虑的步骤以及处理器为中心的执行时间的分量。

158

用这些分量，我们可以进一步细化一个有固定问题的加速比模型，如式（3-5）所示，同样我们假设在程序结束时有一个全局同步。（否则，我们在分母中就要取进程的最大值，而不是任何一个进程的时间性态）

$$\mathrm{Speedup}_{problem}(p) = \frac{\mathrm{Busy}(1) + \mathrm{Data}_{local}(1)}{\overline{\mathrm{Busy}_{useful}(p)} + \overline{\mathrm{Data}_{local}(p)} + \overline{\mathrm{Synch}(p)} + \overline{\mathrm{Data}_{remote}(p)} + \overline{\mathrm{Busy}_{overhead}(p)}} \tag{3-5}$$

---

[1] 如果一个并行算法对给定输入数据集的计算结果总是相同的，独立于所用进程的个数或者事件的相对时序，那么并行算法是确定性的。更一般地，我们可以考虑算法中所有中间结果的计算是否是确定性的。一个非确定性的算法，其结果和参与计算的进程数以及相对事件的时序有关，一个例子是在图上的并行搜索，只要在某一条路线上找到了结果，算法就停止。非确定性算法会使得我们关于时间消耗的简单模型复杂起来，因为和串行程序相比，为达到结果并行程序可能做的有用功少一些。这种情形可能导致超线性加速比——即加速比大于处理器增加的个数。然而，并不是所有非确定性形式有这种好的结果。第2章介绍的红-黑方程求解法是确定性的算法，异步法不是。

[2] 同步引入的时间成分和其他类别重叠。例如，处理器首次访问同步变量的时间，或者说是消耗在通报同步事件上所花的时间，可能包含在同步时间中，或者在相关的数据访问时间中。这里，我们考虑的是后者。除此以外，如果一个处理器在等待一个事件发生时执行访问同步变量的指令，其消耗的时间可以定义为“忙开销”或者同步时间。鉴于其在本质上是负载不平衡的问题，在本书中将它归为同步时间。

在考虑这些性能问题时我们的目标是要使分母中的项尽量小，从而极小化并行执行时间(见图 3-13)。如同我们所见到的那样，程序员和体系结构师都有他们能发挥的作用。如果程序的负载平衡很不好，或者存在过分的额外工作，体系结构是无能为力的。然而，可以通过体系结构方法使得通信和同步更高效，来减少创建这种病态并行程序的可能性。体系结构还可能减少所发生的附加通信，提供方便的命名，从而灵活方便地安排分配机制，并且通过重叠通信和其他有用的工作使得隐藏通信代价成为可能。

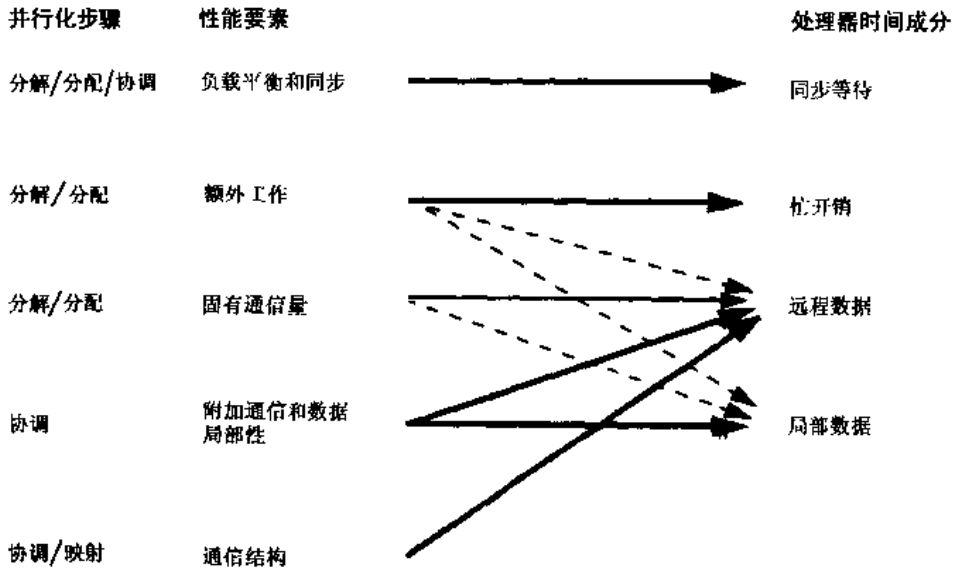


图 3-13　在并行化的若干要素和处理器所看到的执行时间成分之间的对应。粗体线表示直接的关系，虚线表示重要的副作用贡献。左边是并行化步骤和所涉及的最相关的并行化要素

159

3.5　并行应用程序案例的深入分析

前面我们在一般的意义下讨论了并行程序的主要性能要素，并将它们应用到了简单的方程求解器内核。我们现在可以考虑对于比较真实的应用程序，在真实的并行计算机系统上如何获得好的并行性能。特别地，我们现在回到第 2 章介绍的引起我们研究并行软件的 4 种应用案例，对每一个案例应用并行化过程的 4 个步骤并在每一步讨论所引起的主要性能问题。在这个过程中，我们可以理解不同的性能要素之间以及性能和程序设计便利之间的折中。在真实的机器上考察不同的应用呈现给一个并行体系结构的执行时间的不同成分，也能帮助我们看到工作负载特点的类型。当我们的讨论向本书的其他部分展开时，理解并行应用、软件技术和工作负载特点之间的关系将是非常重要的。

并行应用有各种类型和规模，它们具有非常不同的特点，要求在性能要素之间有不同的折中。我们所选的 4 个案例横跨计算机高性能应用的空间，尽管不可能覆盖整个范围，但为我们提供了一组相当好的代表。考察它们的并行化问题，特别是如何在协调步骤追求好的性能，我们将针对这样一种特定的体系结构：一种具有高速缓存一致性的共享地址空间多处理器，主存物理上分布在处理节点上。

对每个应用的讨论分为 4 个部分。第一部分给出串行算法和所使用的主要数据结构的详细描述。第二部分描述应用的划分（即计算的分解和它在进程上的分配），研究负载平衡、

通信规模和计算分配所用的开销等算法层次的性能问题。第三部分用于协调，它描述程序中空间和时间的局部性以及所用的同步和在同步点之间所完成的工作量。第四部分讨论对网络拓扑结构的映射。最后，作为具体解释，我们给出在一种符合所选风格的特定机器上，以一定的问题规模执行时间的各个组成部分。这种机器就是 32 节点的 SGI Origin2000。有用忙和开销忙成分在这个机器上的测量没法分开，局部数据和远程数据成分也没法分开，因此执行时间分成三个成分：忙、等待数据和同步。我们处理这些例子时，某些方面考虑了相当多的细节，这些细节在后面的章节中解释实验结果的时候是很重要的。

160

3.5.1 Ocean

Ocean 应用针对海盆洋流的模拟，在计算流体力学中有许多重要的应用和它相似。它的一些性质也代表了许多其他科学和商业上应用的情况，例如要遍历大的数据结构，在每一个数据点上做少量的计算等。在每一个通过海盆的水平截面，若干不同的变量被模拟，包括流量、温度、压力和磨擦力。每个变量的离散化，用一个规则的、统一的二维网格来表示，含有（$n+2$）×（$n+2$）个点。（这里用 $n+2$ 而不用 n 的道理在于需要考虑更新的内部格点数为 $n \times n$）。总的说来，这个应用要用到大约 25 个不同的网格数据结构。

1. 串行算法

在每一个断面的洋流被初始化后，应用的最外层循环在用户定义的时间步上推进。在每一时间步，首先是在网格上建立一定数据的值，然后是求解微分方程。一个时间步要完成 33 种不同的网格计算，每一个涉及一个或者少量的网格（变量）。典型网格计算包括对几个网格进行标量乘，然后将它们加到一起并将结果存到另一个网格中（例如，$A = \alpha_1 B + \alpha_2 C - \alpha_3 D$），遍历整个网格，求每个点的近邻平均值并将结果存到另一网格，而且在一个网格上用迭代法求解偏微分方程组等。

迭代的方程求解器用的是多重网格法。和我们讨论至今的简单方程求解器内核相比，这是一种复杂的但高效的变形。在简单的求解器中，每次遍历通过整个 $n \times n$ 网格（忽略边界的列和行）。另一方面，一个多重网格求解器在网格的一种层次结构上进行遍历。最初的 $n \times n$网格反映这个层次结构最精细的情况；在下一个较粗的层次中，从每个维删去间隔的网格点，从而得到 $n/2 \times n/2$，$n/4 \times n/4$ 大小的网格等等。求解器首先遍历最细的网格，相继的遍历是在粗的网格还是在细的网格，取决于上一次遍历的误差，当系统在最细的网格上满足某个收敛误差后，算法就终止。为了保持计算是确定性的并使它更高效，我们用红-黑序（见 2.3.2 节）。

2. 分解和分配

在同一个时间步里，Ocean 在两个层次表现出并发性：跨网格的计算（功能并行）和在一个网格内的计算（数据并行）。在相继的时间步之间几乎没有什么并行性。跨网格计算的并发性可以通过写下各种计算对网格的读写关系和分析它们之间在这一层次数据的相关性来发现。结果依赖关系结构和并发性如图 3-14 所示。很清楚，在网格之间的并发性不高（即没有足够的垂直段），用不了几个处理器。我们因此必须开发在网格内的数据并行性，需要决定什么样的功能和数据并行的组合是最好的。

161

在这个案例分析中，我们选择让所有进程在每个网格计算上进行合作，而不是将进程分给不同的网格来开发两种并行性。将功能并行和数据并行结合起来可以增加每个进程网格划

分的大小，从而降低通信计算比。然而，和不同的网格计算相联的工作是相当不同的并以不同的方式和问题规模相关，这就使负载平衡变得复杂。其次，在同一时间步由于不同的计算访问相同的网格，鉴于通信和数据局部性的原因，我们不想将同一个网格在进程之间划分成不同的样子以适应不同的计算。第三，所有的网格计算都是全数据并行的，在一个计算中所有的网格点几乎做同样的事情，因此我们能静态地将网格点分配到进程。尽管如此，知道哪些网格计算是独立的是有用的，因为它允许进程避免在它们之间做同步（见图 3-14）。

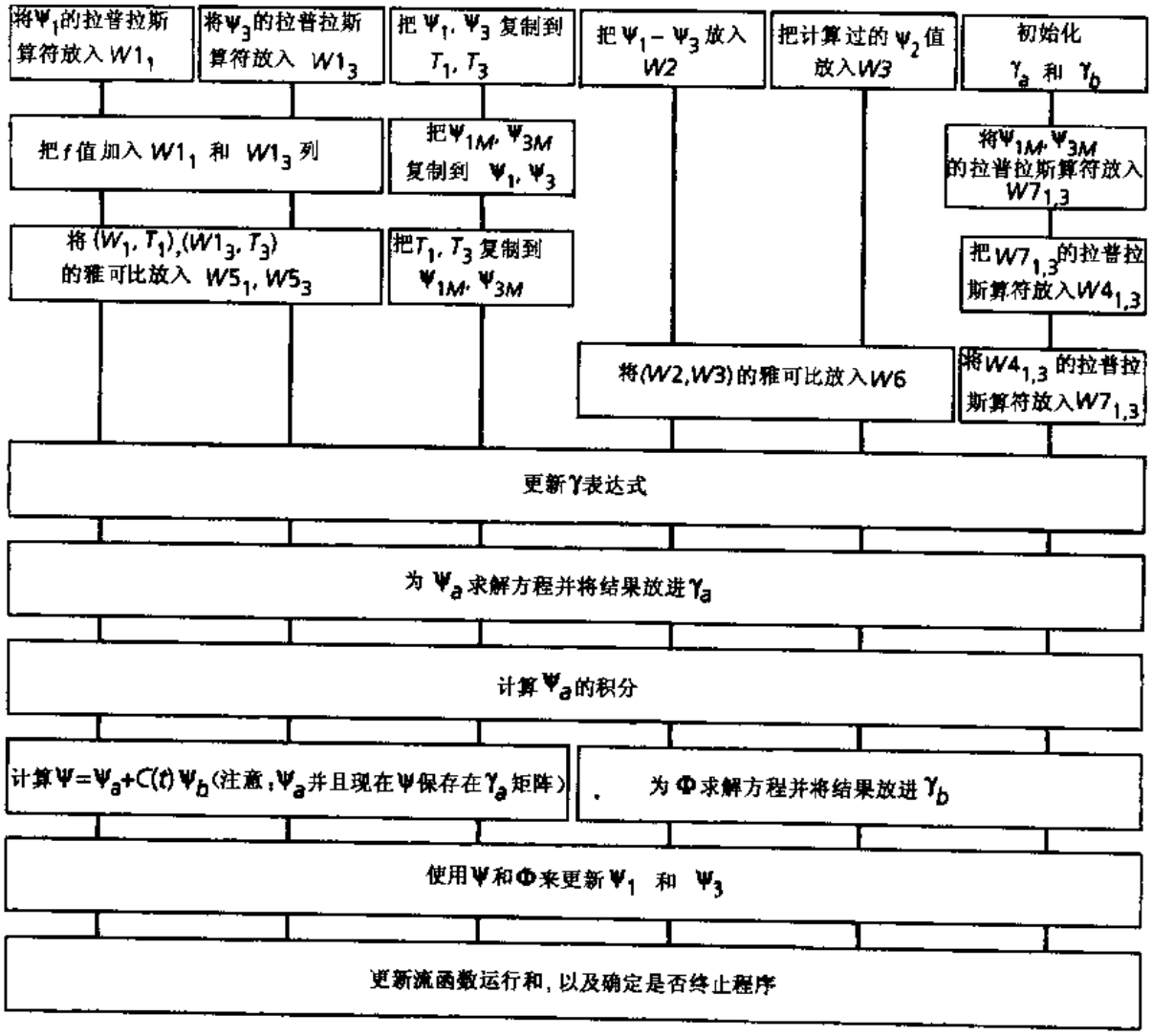


图 3-14　Ocean：在一个时间步中的不同阶段和网格计算之间的依赖关系。每个框是一个网格计算（或者是一对相似的计算）。通过竖线相连的计算相关，其他的（例如在同一行的）计算相互独立。并行程序将每一行看作是一个阶段，在阶段之间进行同步

涉及到的固有通信的问题和简单方程求解器的问题非常类似，因此我们对每个网格用一种块结构的区域划分。有一种麻烦的情况——在数据局部性和负载平衡方面的权衡，关系到有些计算中网格边界的那些点。内部的 $n \times n$ 个点做类似的事情，等量地分到所有进程中。完美的负载平衡也要求考虑边界点并在进程之间做等量分配，但通常它们做的事情较少。然而，通信和数据局部性要求边界点应该分给拥有邻近内部点的那些进程中，于是有一些进程就分不到边界点。我们就按照后面这种思路，允许稍微有点负载不平衡。

最后，让我们考察多重网格方程求解器。所有层次的网格按照相同的块状区域划分。然而，随着我们在这个层次结构上变粗粒度，每个进程的网格点数减少，于是在最高的层次上有些进程可能变得闲置起来。幸运的是，用在这些负载不平衡层次上的时间相对较小。由于每个进程要处理的点较少，通信计算比在较高的层次上也减少。这也说明了相对于最好的串行算法（这里是多重网格）测量加速比的重要性：和并行多重网格求解器相比，一种经典的，非层次式的并行迭代求解器在最初的（最细的）网格上可能给出较好的相对于自己的加速比（相对于单处理器完成相同的计算）。一般来说，低效的串行算法通常产生较好的自相对加速比，但这对最终用户来说不是有用的度量。

3. 协调

这里我们主要关心附加通信、数据的局部性和同步。首先让我们考虑和空间局部性有关的问题，然后是时间局部性，最后是同步。

空间局部性 在网格计算内部，和空间局部性相关的问题类似于3.3.1节的简单方程求解器内核中的问题。用一个四维数组来表示每一个网格。这样就有很好的空间局部性，特别是局部数据。对非本地数据的访问（那些邻居边界元素）沿行划分方向产生好的空间局部性，而列方向较差。简单求解器和完整的Ocean应用的一个主要差别是Ocean在每一时间步涉及33个不同的网格计算，每一个计算涉及25个网格中的一个或多个，因此我们会在网格之间碰到许多缓存冲突扑空。这些冲突扑空的减少可以通过让所分配的数组维数不是2的幂次（尽管程序用2的幂次网格），但是，将不同的网格安放得相对合适，来最小化冲突扑空不是一件容易的事情。第二个不同在于用了多重网格求解器。由于进程的划分在网格的层次结构的高层上有较少的网格点，空间局部性减小，因此在存储器之间以页面的粒度来适当地分布数据就变得更困难，尽管用了四维数组也帮助不大。

162
?
163

工作集和时间局部性 Ocean有一个复杂的工作集层次结构，它有6个工作集。前两个是由于在一个网格内用了近邻计算，因此和简单方程求解器内核的情况类似。如果高速缓存足够大，能够装下几个网格点使得某一个点一旦装进高速缓存，先是作为计算另一个点的右邻点，然后是计算它自己，最后是作为计算另一个点的左邻点，那么第一个工作集就被捕获了。第二个工作集包含进程划分的三个子行。当进程从一个子行回到下一子行的开始时，它可以重用上一子行的元素。

其余的工作集难以作为一个个工作集定义清楚，因此在工作集曲线中也不产生明显的拐点。第三个工作集是进程在多重求解器中一个网格的完整划分。这可以是在多重网格层次结构中的任何一层的划分，因此它不是一个真实意义上的工作集。第四个工作集是进程在网格层次中的若干层子网格的和（在极端情况下，就是所有层次）。第五个工作集使我们能在不同的网格计算，甚至计算阶段之间重用网格；这样它就要大到能够装得下几个网格在进程上的划分。最后一个工作集是进程在每个网格中所分得的所有数据，从而所有数据在不同的时间步之间能够重用。

对性能最重要的工作集是前三、四个，取决于多重网格求解器的情况。这些工作集中最大的一个随每个进程的数据集的变大而线性变大。这个变化率在有些应用中是常见的，重复通过它们的数据集，因此对于大的数据集合某些重要的工作集在本地缓存中放不下。幸运的是，在这些流场应用中大数据集使得在内存中以页面粒度分布数据变得容易，于是一个进程的工作集主要由本地数据构成，不需要通信。非局部数据引起的少量重用被捕获在前两个工

作集中。

同步　Ocean 用两种类型的同步。首先，全局栅障用来在计算阶段之间以及多重网格方程求解器的遍历之间，同步所有的进程（见图 3-14）。在几个阶段之间，我们可能用较细粒度（网格点的层次）的点对点同步，来获得某些阶段之间的重叠。然而，在这种情形下，重叠很可能太小，不足以平衡程序设计的复杂性和较多同步操作所带来的开销。第二种形式的同步是用锁来提供全局规约的互斥，例如，确定求解的收敛。同步点之间的工作量是比较大的，它和网格的进程划分的大小成比例。

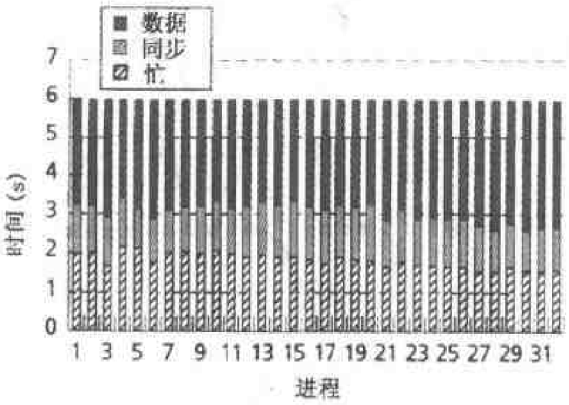
4．映射

给定邻近通信的模式，将进程映射到处理器的思路是：在网格上划分相邻的进程在网络拓扑上接近的处理器上运行。容易看到，我们的二维网格的子网格划分能够很好地映射到一个二维网格网络。然而，如同所有程序，进程到处理器的映射不是由程序来确定的，而是留给系统。

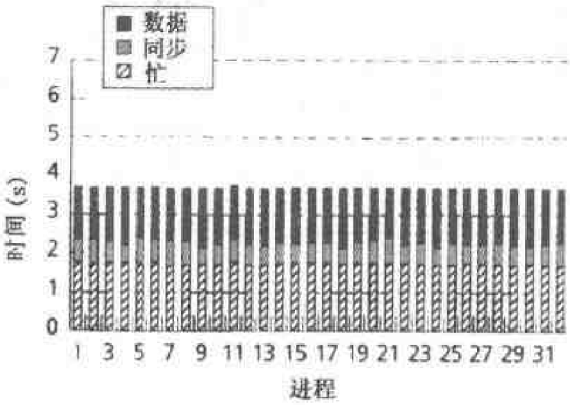
5．小结

对于许多要遍历规则数组的应用，Ocean 是一个很好的代表。对于问题规模是 $n \times n$ 个网格和 p 个处理器，计算通信比和 $n/\sqrt{p}$ 成比例；如果 n 相对于 p 并不是很大，负载平衡是好的；对给定的处理器数，并行效率随网格大小变化。由于在每个网格计算中，处理器通过网格中的那一部分，并且由于每次遍历对数据的操作不多，还由于跨网格的冲突扑空有很大的潜力，数据在存储器中的分布对于物理分布存储系统是非常重要的。

图 3-15 表示执行时间被分解为忙、等待同步点和等待数据访问的完成；具体问题是 1030 × 1030 网格的 Ocean，用 2D 和 4D 数组，执行在 SGI Origin2000 机器上。这个机器有很大的二级快存（4 MB），于是对于四维数组表示，每个处理器的划分能够较容易地放到其高速缓存中。和处理器数相比问题规模足够大，则固有通信计算比就相当低。主要的瓶颈在于等待栅障同步。问题规模小就会有较多通信，而大问题和适当的数据分布对局部存储系统要求



a) 二维数组



b) 四维数组

图 3-15　在 Origin2000 上 Ocean 问题的执行时间分解。每个网格的规模是 1026 × 1026，收敛误差是 10^{-3}。在（b）中用四维数组表示二维网格清楚地降低了停滞在存储系统上的时间（包括通信）。这里的等待时间很小，因为处理器的网格划分很好地适应了这个机器的 4 MB 二级高速缓存。如果高速缓存小一些或者网格大一些，消耗在（局部）数据等待上的时间就会大很多○

○　在这里以及后面的执行时间分解中，没有和图 3-12 中那样附加的终止栅障，使所有进程等待直到最后一个进程被完成才能结束。

高。对于二维数组，情况显然不同。由于数据在主存的分布困难，会经常发生冲突扑空，而许多这样的扑空不能在本地满足，导致较大的时延、竞争和较大的数据等待时间。

3.5.2 Barnes-Hut

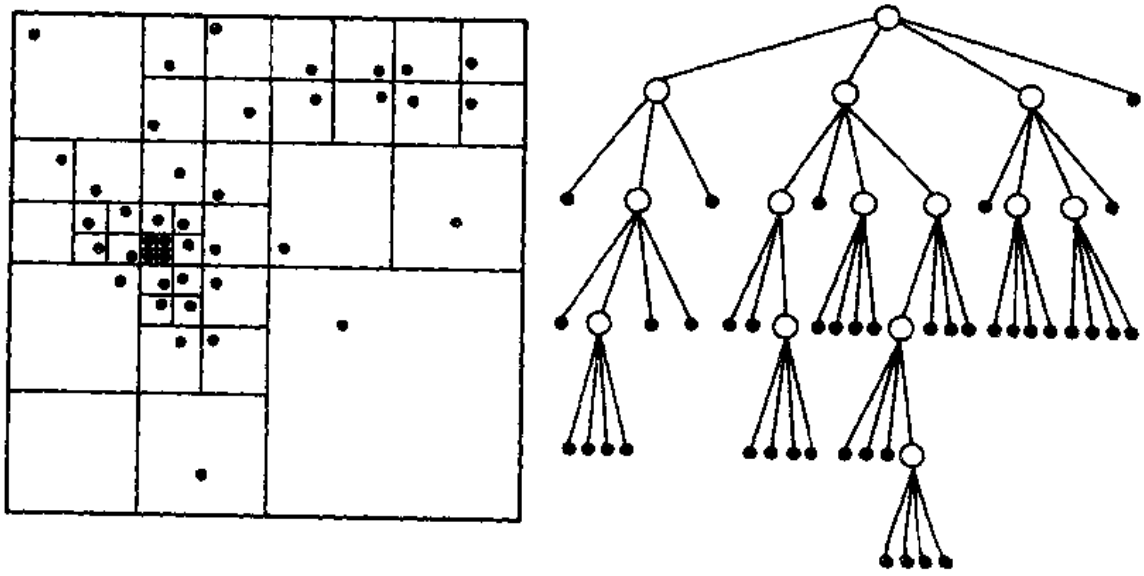
同 Ocean 模拟应用相比，星系演化的模拟表现出非规则和动态变化的行为特征。前面介绍过它解决的是 n-体问题，其中主要的计算内容是计算系统中 n 个对象相互的影响。用来计算星体所受引力的算法（Barnes-Hut）是一种高效的层次式方法。对于 n 体来说，时间复杂性为 $O(n \log n)$。

1. 串行算法

星系的模拟要经历许多时间步，每一步计算作用在每个星体上的合力，然后更新该星体的位置和其他属性。在 Barnes-Hut 方法中计算作用力的思想是基于：如果星体间相互作用力的大小随距离迅速减小（如同引力），一群星体的效果可以由一个等价的星体来近似，如果那一组星体距离作用点足够远的话。这个思想的层次式应用隐含着距离越远，能用一个星体来近似的群体就越大。

165 ~ 166

为了便利层次式的做法，Barnes-Hut 算法将包含有星系的三维空间表示为一棵树。树根表示包含所有星体的空间。这个树的建立是将星体加到最初为空的根节点上，然后一旦它含有了一定数量的星体（这里为 10），就将一个节点分为它的 8 个相同大小的子空间。这个结果是一个八叉树，它的内节点是表示空间的单元，叶节点包含一个个的星体。[○] 源于单元分割的空单元被忽略。这个树（也就是 Barnes-Hut 算法）因此具有适应性，它在星体密集的区域扩展有较多的层次。n-体问题一般是一个三维空间的问题，但为了解释方便起见，图 3-16 给出了一个小的二维例子，对应的树为四叉树。星体的位置随时间改变，因此这个树要在每



a) 空间域　　b) 4 叉树表示

图 3-16 Barnes-Hut：二维粒子的分布和相应的四叉树

[○] 八叉树是这样的一种数据结构，其中每个节点最多有 8 个子节点。在二维的情形，可以用四叉树，其中最大的子节点数不超过 4。

第3章　面向性能的程序设计

我们知道，用并行处理系统的目的是为了获得高性能。具体地理解了分解、分配和协调等步骤是怎么体现在实际运行在机器上的并行程序代码中的，我们就可以来考察那些限制并行程序性能的关键因素以及在许多问题中它们是如何表现出来的。我们将看到在程序设计过程中所做的决定是如何影响给体系结构的运行时特征以及体系结构的特征是如何影响程序设计决定的。理解程序设计技术和这些相互制约的关系，不但对并行软件设计人员，而且对系统结构人员都是重要的。除了将并行程序看成我们所设计系统的工作负载外，我们从中学会做硬件/软件之间的权衡。特别是，我们可以认识到系统结构能够有效地影响可编程性和性能的哪些方面，而其他方面最好留给软件。对于多处理器系统的性能来说，程序和系统的相互依赖关系要比单处理器情形所涉及的面更宽、更加复杂也更加重要；因此，对于我们设计高性能系统，要降低代价和程序设计劳作的目标来说，理解这种相互作用是很关键的。从第4章讨论工作负载驱动的系统结构评估的方法开始，这种理解将是我们贯穿全书的一种基本精神。

有关性能问题和并行软件技术的内涵是丰富的：不同的目标相互制约，更好地实现某个目标的技术可能使我们重新考虑用于解决另外目标的技术。这就是并行软件创建的引人入胜之处和挑战所在。如同在单处理器的情形一样，多数性能问题能够通过软件算法和程序设计技术，或者由体系结构技术来解决或者两者兼用。本章着重于性能问题和软件技术。至于体系结构方面的技术，将在本书的其余部分讨论，本章只是隐含地涉及一些要点。

尽管我们必须要考虑若干具有相互作用的性能问题，但它们不是同时来处理的。创建一个高性能程序的过程是逐步求精的过程。如第2章所讨论的，划分的步骤（分解和分配）常常是和底层系统结构或程序设计模型相独立，它们主要关心算法方面的问题，只依赖于问题固有的特性。特别地，这些步骤只是将多处理器简单地看成是一组相互通信的处理器。它们的目标是要解决进程间的工作负载平衡；减少程序中固有的进程间通信；减少为了计算和管理划分所带来的额外的工作。我们首先将注意力集中在这些和划分有关的问题上。

然后，我们看体系结构，考察它在协调和映射步骤中所涉及的新的性能问题。这意味着我们要认识到两种事实。第一，多处理器不仅是一组处理器，而且还是一组存储器，每个处理器可以将这存储器看成是一种扩充的存储器层次结构。在这些存储层次中数据的管理可能引起更多的数据在网络上传送，使得实际发生的通信要比并行程序划分所导致的固有通信要多。因此，实际发生的通信既依赖于划分，也依赖于程序访问的模式，数据引用的局部性和这种扩充的存储器层次结构的组织与管理相互作用。第二，由处理器所看到的通信代价（也就是通信在程序执行时间中的份额）不仅取决于通信量，还取决于它的结构以及和体系结构的相互作用。3.2节讨论通信、数据的局部性及其和扩充的存储结构之间的关系。然后在3.3节针对这些在协调和映射方面的主要性能问题，考察有关的软件技术：通过在扩充的存储器层次结构上开发数据的局部性；减少额外的通信；将通信结构化以减少其开销。

是不统一的，因此一个好的分配难以通过观察来发现。其次，星体的分配随时间步变化，意味着静态分配不大可能奏效。再者，由于在作用力的计算中信息的需要随距离递减，所有方向相同，减少进程间的通信要求划分在空间中连续并且在任何方向上不能有大小的倾向性。第四，在一个时间步里的不同阶段在体/单元之间有不同的负载分布要求，因此有不同的划分偏好。例如，在更新阶段的工作对于所有体都是统一的，而力的计算阶段显然不是。另一个要得到好性能的挑战是进程间所需的通信自然是细粒度和非规则的。

由于作用力的计算是最耗时的，我们重点讨论该阶段的划分。对于其他阶段，这个划分不随特殊需要改变，否则的话，重新划分和局部性的损失要比可能获得的好处要大；还由于类似的划分可能在其他阶段也工作得很好，例如对于树的构造和动量的计算阶段（尽管对于更新阶段不怎么好）。

在这个应用中，我们能够用基于性态的半静态划分，考虑下面这个事实：尽管星体的空间分布在模拟结束时可能与开始时有很大的不同，但它随时间的变化是缓慢的，在两个相继的时间步之间变化很小。在一个时间步里计算作用力的时候，我们记录下在该时间步中每个粒子所做的工作（即我们记下它和其他体或单元相互作用的次数）。然后我们把这个数作为下一时间步里和该粒子相关工作的一种度量。和代价高的相互作用计算相比，工作计数的开销是小的，它只涉及增加一个本地计数器。现在我们需要将这种负载平衡方法和分配技巧结合起来，那些技巧也达到通信的目标：保持划分在空间中的连续，不在任何方向上、大小上有偏向。我们简单地讨论两种技巧：第一种适合于许多非规则问题，第二种对我们这个应用有特别的针对性，也是我们程序所用的。

169

第一个技术，称为正交递归二分法（ORB），通过直接划分数据空间保持空间物理上的局部性。用前面提到的负载平衡度量，该空间被递归地分为两个带有相等工作量的矩形子空间，直到每个进程留有一个子空间（见图 3-18a）。开始，所有进程都和整个空间相关。每次分割一个空间的时候，和它相关的一半进程分给所得到的子空间。划分所取的笛卡尔方向通常随相继的分割交替改变，一个并行的中值计算方法用来确定在哪儿分割当前的子空间。一个单独的深度为 $\log p$ 的二叉树用来跟踪这些分割，实现 ORB。（这个应用中使用 ORB 的细节见［Salmon 1990］）

第二个技术，称为代价区（costzones），其基础是认识到在 Barnes-Hut 算法的树数据结构里已经有了一个空间分布星体的表示。这样，我们可以对这个数据结构本身做划分，从而达到划分空间的目的（见图 3-18b）。每个内部单元所存放的是和它所包含的所有星体相关的总代价。系统中总的工作量或者代价在进程之间分担，每个进程有一个连续的、相等区域的工作（例如，1000 个单位的总工作量在 10 个进程之间分配，1 ~ 100 单位区分给第一个进程，101 ~ 200 单位区分给第二个进程，等等）。树中的一个星体属于哪个代价区，可通过按先根序的遍历到该体的总代价确定。进程并行地遍历这个树，摘取属于它们代价区的星体。（细节见［Singh et al.1995］）代价区方法要比 ORB 容易实现得多。两种方法都导致类似的负载平衡和固有通信的性质，但代价区方法在共享存储空间中有较好的总体性能。这主要是由于花在划分阶段的时间本身（即计算这个划分）要小得多，这是额外工作对性能影响的一个例证。

3．协调

Barnes-Hut 的协调问题揭示了许多和 Ocean 的不同点，说明即使同是科学计算应用，也可能有相当不同的行为特征，这些差别对系统结构的研究是有用的。

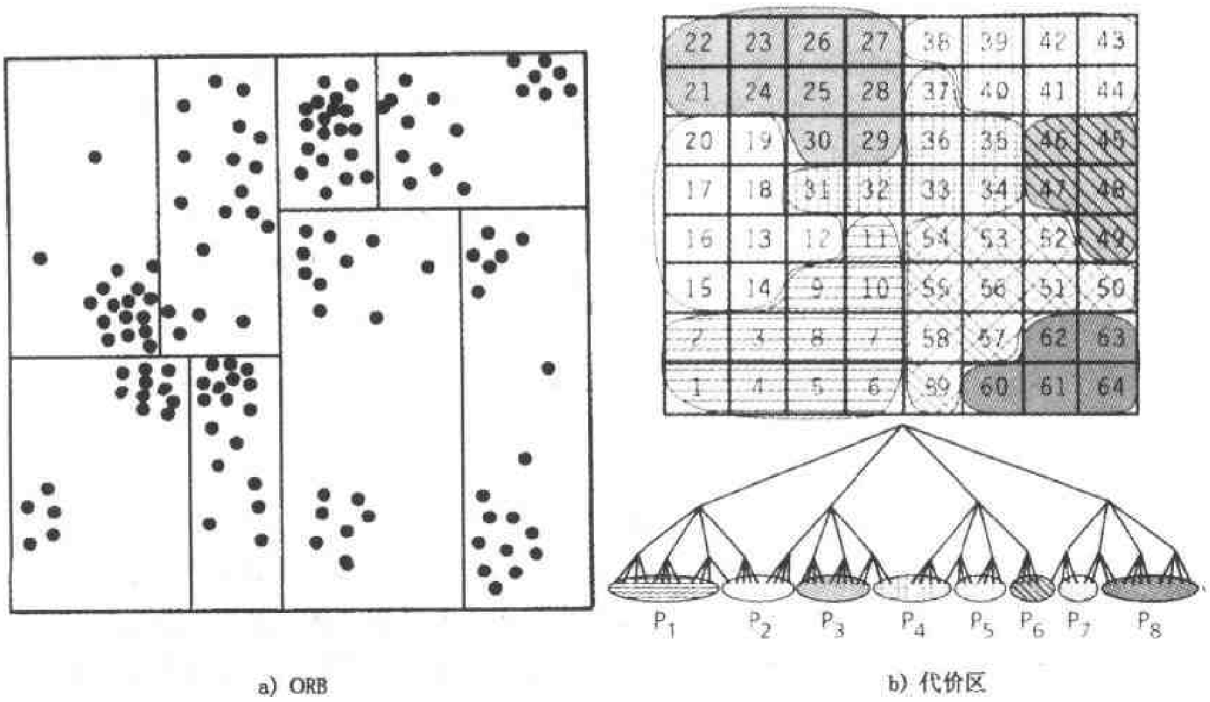


图 3-18　Barnes-Hut 的划分方案：ORB 和代价区法。ORB 直接将空间递归用二分法划分，代价区法的划分作用在树上。b）表示代价区法在树上的划分及其所导致的空间划分结果。和代价区法相比，ORB 得到的划分要规则些（矩形），但它们的通信和负载平衡性质相当类似

空间局部性　在共享地址空间，进程访问它在各个计算阶段所需的共享树的有关部分比较容易；但分布数据用来保持进程分配的星体和单元在它自己的主存就不如 Ocean 容易了。首先，数据得随不同时间步动态重新分布，这个代价很高。其次，逻辑的数据粒度（一个粒子/单元）要比物理存储分配的粒度（一个页面）小得多，分给相同进程的星体/单元在物理空间中是连续的，并不意味着它们在星体/单元数组的空间上连续。解决这些问题要求彻底规整存放星体和单元的数据结构：每个进程用单独的数组或表，在跨时间步当分配改变时进行修改，因此是和串行程序不同的数据结构。幸运的是，在这个应用中有足够的时间局部性，数据分布在共享数据空间不是那么重要（这也是不同于 Ocean 的）。除此以外，大多数高速缓存扑空是对于那些分给其他处理器的体和单元，因此数据分布本身不能使这些扑空局部化。我们因此简单地以一种轮循交替的方式将共享数据的页面分布在节点之间，不具体去管哪个节点得到哪个页面。

Ocean 中，大的高速缓存块能改善局部访问性能，只是受划分尺寸的限制；面这里多字的高速缓存块对开发空间局部性的帮助仅在于读一个粒子的偏移或动量数据涉及读多个双精度的数据。很大的传送粒度可能引起较多的碎片，因此预取的用处不大，原因和数据以页面分布的困难是相同的：不同于 Ocean，星体/单元在数组中的局部性和物理空间的局部性不匹配（分配是基于物理空间的），因此针对扑空从数组中取得多于一个星体/单元的数据可能是有害，而不是有益的。空间局部性取决于星体或单元结构的大小，不随星体数改进多少。

170 ~ 171

工作集和时间局部性　这个程序中的第一个工作集包含用于计算星体－星体或星体－单元之间力的数据。在遍历中，和下一个星体或单元的相互作用将重用这个数据。第二个工作集对性能来说是最重要的。它的构成包含所有与计算作用在单个星体上的作用力有关的数据，这

些数据在遍历树的时候碰到。由于划分的方式，下一个要计算受力的星体将在空间上和当前的星体靠近，因此通过树的遍历来计算作用在星体上的作用力将重用大部分数据。当我们从一个星体到另一个星体的时候，这个工作集的构成缓慢变化，但重用的量是很大的，于是结果工作集很小，尽管宏观上一个进程以非规则的方式访问了一个数量非常大的数据。在这个工作集中的许多数据源于其他进程的划分，大多数数据是非局部分配的。这样，对这个应用性能的关键是在共享数据（包括局部和非局部）上开发的时间局部性，而不像 Ocean 中是数据分布的。

算法的复杂性为

$$O\left(\frac{1}{\theta^2}\times n\log n\right)$$

从而这个工作集的期望大小和

$$O\left(\frac{1}{\theta^2}\times \log n\right)$$

成比例，尽管本应用的整个存储要求和 n 几乎呈线性关系：每个粒子访问树中大约这么多数据来计算作用于它的力。这里的比例系数不大，是从每个星体或单元在计算力的期间访问的数据量。由于这个工作集缓慢增长，很容易装到现代二级高速缓存中，我们不需要在主存中复制数据。在 Ocean 中，某些重要的工作集随数据集大小线性增长，我们不总是期望它们能放在高速缓存中；然而，适当的数据分配是容易的并能够保持多数高速缓存扑空在局部，因此即使在 Ocean 上我们也不需要在主存复制数据。

同步　在计算的一些阶段之间，例如树的建立和用树来计算受力为两个不同的阶段，栅障用来维护体和单元之间的相关性要求。至少对于我们假设的程序设计原语来说，相关性不可预测的特点使得在星体或单元粒度上用点到点同步替代栅障比较困难。用在一个时间步中的少量的栅障独立于问题大小或处理器数，只取决于阶段数。

172

在作用力的计算阶段本身，是不需要同步的。当应用中通信和共享模式为非规则时，它们是阶段结构化的。即尽管一个进程在作用力计算阶段要从许多其他进程读星体和单元数据，在本阶段写的一个星体结构的域（加速度和速度）不同于它所读出的（位移和质量）。位移只是在更新阶段的结束时才写入，质量在初始化后就不再改变。然而，在其他阶段程序可以用锁来互斥和用标记做点到点事件同步，其方式和 Ocean 相比要有意思得多。在树的构造阶段，当一个进程准备要增加一个星体或单元时，它必须首先得到对单元的互斥访问权，这是由于其他进程可能要在同一时间访问同一单元。这可以通过给每单元一加把锁来实现。计算一个单元质心的阶段本质上是一次从叶到根的上朔过程，每个单元动量的计算要从它的子单元的动量开始。点到点事件同步用标记实现，以保证父节点在子节点本身没被更新之前，不读取它的子节点的动量。这是一个多生产者、单消费者组同步的例子。在更新阶段内部没有同步。

在同步点之间的工作量是大的，特别是在力计算和更新阶段，分别为 $O\left(\frac{n\log n}{p}\right)$ 和 $O(n/p)$。在树的构造和质心计算阶段，锁住单元的需要使得这些阶段中同步点之间的工作量要小得多。

4．映射

非规则特性使这个应用较难在像网格那样的常见网络上映射得好，从而在网络节点的意

义上开发局部性较难。ORB 划分方案能够很自然地映射到一个超立方体拓扑结构上（将在第 10 章讨论），但这对网格或连通性差一些的网络并不怎么好。对于代价区划分来说，它能自然地映射到一个一维处理器结构上，但在许多其他网络拓扑上不容易保证通信的局部性。

5．小结

Barnes-Hut 应用展示了非规则、细粒度、随时间变化的通信和数据访问模式。在科学计算中，随着我们试图模拟更加复杂的自然现象，类似的情况日益普遍。仅仅从程序设计的角度，难以感受到对这个应用有效划分的方案，需要有来自应用领域的洞察力。这种洞察力能够使我们摆脱用完全动态的分配方法，例如任务队列和窃取。 173

图 3-19 示出这个应用在 32 处理器的 SGI Origin2000 上执行时间的分解。即使是静态的将星体指针数组划分到处理器，负载平衡也相当好，这恰恰是由于星体在数组中的位置和在物理空间中的位置没有什么关系。然而，由于大量的通信（固有和附加）产生于缺乏连续性的物理空间中，静态划分的数据访问代价是高的。半静态代价区划分大大减少了这种数据访问的代价，同时不损害负载平衡。

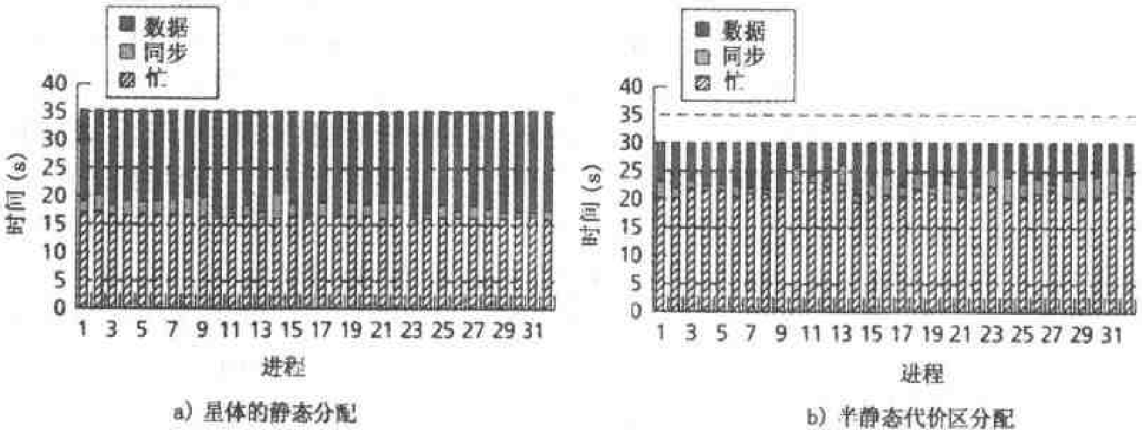


图 3-19　在 Origin2000 上 Barnes-Hut 对 512 K 个星体的执行时间分解。所用的静态分配是相当随机的，这样，在星体数相对于处理器数要大得多的情况下，负载就自然比较平衡了（由于大数定律）。这里静态分配较大的问题是，由于是随机的，分给处理器的粒子在空间中不聚在一起，因此通信对计算的比要比半静态方案大得多。这就是半静态方案的数据等待时间要小得多的原因。如果我们静态地将空间中连续的区域分配给进程，数据等待时间会小一些，但负载会不平衡，因此同步等待时间就会大了。即便是现在这种静态分配，随着星系的演化也不能保证分配会保持负载的平衡

3.5.3　光线跟踪

前面介绍过，在光线跟踪问题中，光线通过一个图像平面的像素射到一个三维场景，光线在不断反射和折射时的路径被跟踪，以计算对应像素的颜色和透明度。算法用了一种空间的层次表示法，称为层次化一致网格（HUG），在结构上类似于 Barnes-Hut 应用中的八叉树。树的根表示包含场景的整个空间，每个叶维持一个落到该叶范围内的对象基元的链表（每个叶的基元最大值和该树结构的其他一些方面由用户定义）。当跟踪一束光线时，层次网格或树使我们能够高效地掠过空间中的空区域，迅速找到下一个有意义的单元。 174

1．串行算法

给定一个视点，串行算法对图像平面的每一个像素点发出一个到达场景的射线。这些初始的射线称为原始射线。当该射线碰到第一个物体时（通过遍历层次统一网格找到），它首

先朝每一个光源反射，以决定它是否在该光源的阴影中。如果不是，该光源对它的颜色和亮度的贡献就可计算出来。这条光线还在场景中的物体上根据情况作反射和折射。每次反射和折射生成一条新的光线，然后递归地重复上述过程。这样，每个原始射线产生一个射线树。这个过程的终止条件是它们离开包含场景的空间或者遵照某种用户定义的标准（例如光线树中的最大层次数）。光线跟踪问题或者更一般地讲计算机图形学中的许多问题，表现出在执行时间和图像质量之间的几种权衡，人们研究了许多改善性能但不是十分影响图像质量的优化算法。

2. 分解和分配

有两种自然的做法在光线跟踪中开发并行性。一种是划分空间，从而也就是将场景中的物体分给进程，让一个进程计算光线在它自己的空间中的相互作用。这里分解的单位是一个子空间。当一条光线离开一个进程的子空间时，它将进入另一个子空间，并由负责该空间的进程来处理它。这称为是面向场景的做法。另一种面向光线的做法是在进程之间划分图像平面中的像素。一个进程对分给它的像素的光线负责，在它通过场景的整个路径中始终跟着一条光线，计算此条光线所产生的整个光线树的相互作用。这时，分解的单位是一条原始光线。如果有必要的话，分解还可能进行细化，可允许不同的进程处理由同一根原始光线产生的光线（即同一个光线树上的光线）。在面向场景的方法中，由于一个进程只接触它的子空间的场景数据和进入该子空间的光线，该方法保持更多的场景数据局部性。不过，面向光线的方法编起程序来要容易得多（特别是从一个循环在光线上的串行程序开始并行化），由于光线能独立处理不需要同步，而场景数据是只读的，因此在共享地址空间上低开销实现起来也是容易的。在消息传递模型，显式非局部场景数据的复制，做起来也很容易。这个程序因此用了面向光线的方法。对于 $n \times n$ 像素平面来说的并发性是 $O(n^2)$，通常来说是足够多的。

175

不幸的是，图像平面的静态块状划分将不是负载平衡的。从图像平面不同部分的光线可能遇到非常不同数量的反射，这意味着不同的光线可能对应有非常不同的工作量。鉴于工作量的分配高度不可预测，我们这里用一个分布式任务队列系统（每个进程一个队列），带有支持动态平衡负载的任务窃取功能。

为了确定如何开始将光线或者像素分给进程，让我们考虑通信的情况。由于场景数据是只读的，它不引起固有通信。如果我们在每个节点上复制整个场景数据，那么除任务窃取外就完全没有通信。然而，这种做法使我们不能显现比放在一个节点存储器大的场景，因此数据集的大小不能随处理器的增多而扩大。这里采用将场景数据分放在 p 个处理器上的做法。因为只有 $1/p$ 的场景放在一个节点，于是除了任务窃取外还要产生其他通信，而进程访问场景是广泛的、不可预测的。为了减少这种附加通信，我们希望进程尽可能重用场景数据，而不是随机地访问整个场景。为此，我们可以开发光线跟踪中的空间相关性：鉴于光线反射和折射的方式，从同一个视点通过相邻像素点的光线很可能经过场景中相似的部分，以相似的方式反射。这就提示我们应该在像素平面上用域分解法，将像素初始分配到任务队列中。由于相邻性或者光线的空间相关性在图像平面的所有方向起作用，面向块状的域分解效果会很好。这也减少图像、像素它们自己的通信。

给定 p 个进程，图像平面划分成 p 个矩形块，大小尽量接近。每个图像块或者划分进一步分为大小固定的正方形图像拼块，它是任务粒度和窃取的单位（见图 3-20 四进程的情形）。这些拼块任务最初插入相关的（拥有它的）处理器的任务队列，一个处理器按线扫描

次序来光线跟踪它自己划分中的拼块。当它完成了自己的分块后，就从仍然还在忙着的处理器那里窃取拼块任务。拼块大小的选择也是需要折中考虑的：大一些，可以通过空间相关性保持数据的局部性和减少对其他处理器队列的访问次数（两者都减少通信）；小一些，则可以通过足够小的任务来更好地保证负载平衡。我们也可能在开始的时候以某种在两个维都交织的方式将拼块分给进程（称为散发分解），从而追求在初始分配中改进负载的平衡，减少运行时任务的窃取，这时的代价是某些空间相关性的损失。

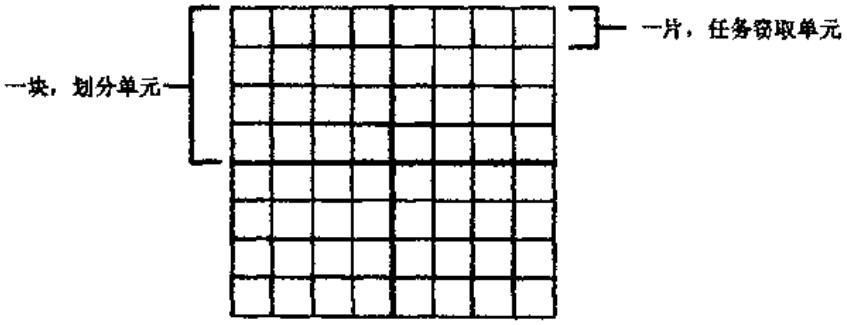


图3-20 针对四个处理器的系统，Raytrace的图像平面的划分。每一小片包含若干像素。每个进程分得一片连续的拼块。当进程完成了对分给它的块的处理，它可能从其他进程那里“窃取”小片来处理

3．协调

给定前面的分解和分配，让我们考察空间局部性、时间局部性和同步。

空间局部性 大多数共享数据的访问是针对场景数据的。然而，由于视点的改变和光线的反射不可预测，不可能将场景划分为若干部分，每一个只被（或者即使主要是）一个进程访问。除此以外，场景数据结构自然是比较小的，通过指针连在一起，因此很难在存储器中用页面的粒度分布它们。我们因此用一种含有场景数据页面的轮循布局来减少热点和冲突。图像数据也很小，我们试图将它所涉及的几个页面分配到不同的处理器，如同场景数据那样。先前描述的对图像平面的块状分配，在高速缓存块的粒度能够相当好地保持空间局部性，尽管它可能导致一些拼块边界局部性的损失（特别是任务窃取时）。图像平面的条状分解，从空间局部性的观点看起来要好些，但在利用场景的空间相关性方面并不怎么好。如同Ocean，最好的选择可能是和体系结构有关的，分配能够容易地参数化。场景数据上的空间局部性不是很高，对于大场景来说也没什么改进。 176

时间局部性 由于场景数据的只读性质，如果有无限的复制容量，那么只是在最初对非本地分配的数据引用时会引起通信。另一方面，如果是有限复制容量，数据可能要被替换，重新通信就成为可能。前面所描述的域分解和空间相关性方法增强了场景数据的时间局部性，减小了工作集的大小。然而，由于访问模式是如此的不可预测（鉴于光线的弹射），工作集相对还是较大并且定义的不好。注意，大多数被访问的场景数据，也就是工作集的内容，可能是非本地的。尽管如此，这个共享地址空间程序没有安排在主存复制数据，主要是由于工作集不那么明显，机器上的缓存已经较大，而在主存搞数据复制是有代价的，因此不清楚利益是否能大于开销。

同步和粒度 程序只用了一个栅障，安排在整个场景的渲染完成后，但在显示之前。锁用来保护任务队列和某些跟踪程序统计信息的全局变量。在同步点之间的工作是和一个光线拼块相关的工作，通常是相当大的。 177

4. 映射

由于 Raytrace 对于它的场景数据有非常不可预测的访问和通信模式，几乎没有通过映射来优化附加通信的空间。初始的分配将图像划分为二维块状网格，使得我们很自然将它映射到一个二维网格网络，但这种映射的效果不可能很明显。

5. 小结

这个应用的特点是有较大的工作集以及相对较差的空间局部性；只要有足够的场景复制容量，通信计算比就很低。图 3-21 显示在 Origin2000 上执行时间的各个成分，针对的是一种标准的数据，即场景为一些安排成堆的球，该图表现了任务窃取对于提高负载平衡的重要性，而负载平衡也意味着在栅障同步上等待时间的减少。由于任务窃取所发生的额外通信和同步是非常值得的。

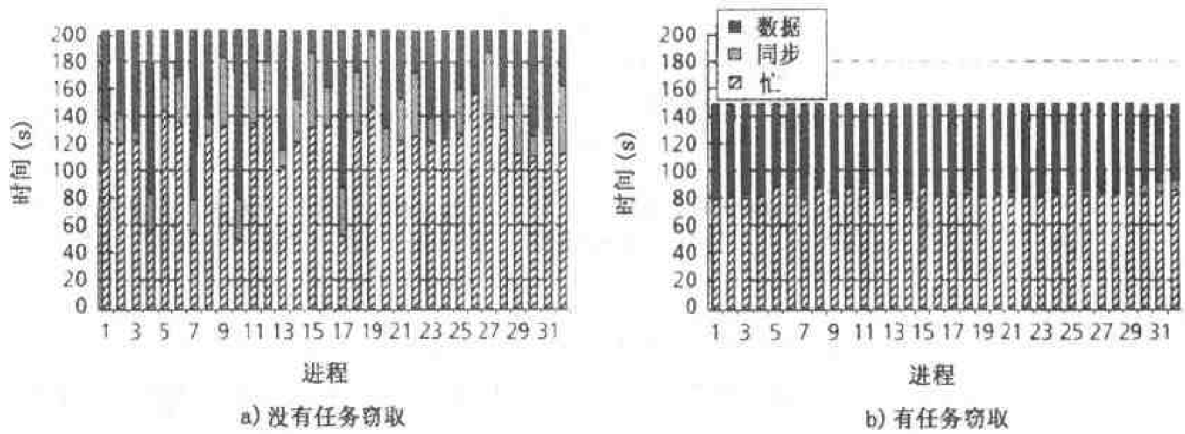


图 3-21 在 Origin 2000 上，Raytrace 算法在球数据上的执行时间分解。在这种高度不可预测的应用中，任务窃取对平衡负载（从而降低在栅障上的同步等待时间）来说显然是很重要的

3.5.4 数据挖掘

数据挖掘和前面提到的那些应用的一个关键区别在于它所要访问和处理的数据典型地驻留在磁盘上，而不是在内存中。由于磁盘访问的代价很高，减少其访问次数就变得非常重要。次数减少的另一好处是减少不同的处理器对磁盘控制器的争用冲突。从本质上看，用于减少磁盘访问代价的技巧和减少通信和存储访问的技巧是相同的。

在前面章节的介绍中，我们得到相联挖掘中的基本要点：如果大小为 k 的物品集合是大的（即它在交易中出现的次数超过某个阈值），那么这个项目集合的所有子集必然也是大的。作为解释，考虑一个数据库，包含五个物品（A、B、C、D 和 E）其中一个或者多个可能出现在一个特定的交易中。在一个交易中的物品按照词典次序存放。考虑 L_2，大小为 2 的物品集的集合，可能是 {AB，AC，AD，BC，BD，CD，DE}。在 L_2 中的物品集也是按词典次序排列。给定这个 L_2、L_2 的候选对象通过在 L_2 的元素上做连接操作得到——即取 L_2 中共享一个物品的物品集合对（比方说，AB 和 AC），将它们连接成一个大小为 3 的物品集合（这里是 ABC），也按词典次序排列。在这种情形下所导致的候选集合 C_3 是 {ABC，ABD，ACD，BCD}。在 C_3 的这些物品集合中，某些可能出现的频率足够高，从而要被放到 L_2 中，等等。一般来说，从 L_{k-1} 得到 C_k 的连接操作在 L_{k-1} 中找到物品集合对，它们的前 $k-2$ 个物品是相

同的，将它们连接以生成新的 C_k 中的物品集合。大小为 $k-1$ 的物品集合，那些有公共（$k-2$）大小前缀的，称为形成了一个等价类（例如，在这个例子中对于 $k=3$ 为 {AB，AC，AD}，{BC，BD}，{CD} 和 {DE}）。在从 L_{k-1} 形成 C_k 的过程中，只是那些在等价类中的物品集合需要一起考虑，这就大大减少了为确定 C_k 物品集合对比较的次数。 178

1．串行算法

相联挖掘简单的串行方法是首先遍历数据集合，记录所有大小为 1 的物品集合出现的频率，这样确定 L_1。从 L_1 我们能够构造候选表 C_2，然后再次遍历数据集合以发现 C_2 的哪些项出现的频率足够高以便可以放到 L_2 中去。从 L_2 我们能够构造 C_3，然后遍历数据集合来确定 L_3，如此下去，直到我们找到 L_k。尽管这个方法简单，它要求从数据库中读所有的交易 k 次，代价是很高的。

另一种串行算法寻求减少从大物品集 L_{k-1} 中计算候选表 C_k 的工作量，特别是减少为确定 C_k 中物品集合的频率必须从磁盘读出数据的次数。我们已经看到等价类能用来达到第一个目标。事实上，它们可用来构造一个方法，同时达到两个目标。这里的想法是变换数据在数据库中存放的方式。原来交易数据存放的形式是 {T_x，A，B，D，...}，其中 T_x 是交易标识，A、B、C 是交易中的物品；现在我们可以考虑让数据记录的形式为 {IS_x，T_1，T_2，T_3，...}，其中 IS_x 是一个物品集合，T_1，T_2 等是包含该物品集合的交易。即，数据库记录按物品集合来组织，而不是交易。如果大小为 $k-1$ 的大物品集（即 L_{k-1} 中的元素）在相同的 $k-2$ 等价类中被识别，那么计算候选表 C_k 只要求考察所有这样的物品集对。由于每个物品集合附有它的交易表，在 C_k 中每个结果物品集的大小能和从 L_{k-1} 物品集的对中构造 C_k 物品集表的同时计算出来，方法只是简单地计算在那个表对中的交易的交。 179

例 3.4　假设 {AB，1，3，5，8，9} 和 {AC，2，3，4，8，10} 是大小为 2 的大物品集，在相同的一等价类中（它们从 A 开始）。数据将怎么在磁盘和存储器中被访问的？

解答：包含物品集 ABC 的交易表是 {3，8}，因此物品集 ABC 的出现次数是 2。这意味着一旦完成了数据库变换，1-等价类识别出来了，对于一个1-等价类的其他计算就能够完成了（即发现所有大小为 k 的大物品集），不需要考虑任何其他 1-等价类的数据。如果一个 1－等价类能放在内存中，那么在数据库变换后，一个给定的数据项只需要从磁盘读一次，大大减少了昂贵 I/O 访问的次数。于是我们得到了一种成块以开发时间局部性的形式。■

2．分解和分配

这两种串行方法在并行化方面也是不同的，后者在这方面也有优势。为了并行化第一种方法，我们可能首先在处理器之间划分数据库。在每一步，一个处理器只是遍历在它的本地部分的数据库，来确定候选物品集合的部分出现的计数，在这一阶段不会引起通信和非本地磁盘访问。部分计数然后合成为全局计数，来确定哪些候选者是大的。这样，这个方法不仅要求在数据库上做多次遍历，而且要求进程间通信和每一遍历结束时同步。

在第二种方法中，那些有助于串行算法减少磁盘访问的等价类对并行化也是非常有用的。由于在每个 1-等价类上的计算是相互独立的，我们可以简单地在进程之间分 1-等价类，其后能够独立推进，其余的程序部分没有同步和通信。对应于一个等价类的物品集表（转换后的格式）能存放在进程本地磁盘，它分配有该等价类，因此在这以后再没有远程磁盘访问的需要。如同在串行算法中，在进行到下一个之前，每个进程能完成它所分得的等价类上

的工作，因此每个来自本地磁盘的物品集记录应该也只被读一次、作为它的等价类的一部分。

180 这里的挑战是要保证在进程之间等价类分配的负载平衡。负载平衡的一个简单的度量是基于等价类中的初始项来分配它们。然而，随着计算大小为 k 的物品集过程的展开，工作量的确定更接近在每一步所产生的大物品集的个数。对这个量或者某些其他更合适的量进行估计，用于启发式算法常常是可行的。否则，我们可能得归诸于动态任务和任务窃取，那就要大大破坏这个方法的简单性（即一旦进程分得了它们的初始等价类，它们不再需要通信、同步或者做远程磁盘访问）。

当然，这种做法的第一步是要计算 1-等价类和在它们中大小为 2 的大物品集，来作为并行分配的起始点。为了计算这些物品集合，我们最好用原始的面向交易的数据库形式，而不是变换后的版本，因此我们也不对数据库进行变换（见习题 3.18）。针对交易中的每一对物品，每个进程遍历数据库中属于它本地部分的交易，当发现其出现后，就在此物品对的本地计数器上加一（本地计数能维护成一个二维上三角矩阵，索引为物品）。本地计数值然后合并起来（这要涉及到进程间通信），从而就确定了大小为 2 的大物品集的个数。这些物品集然后划分成 1-等价类，按前面所描述的方式分给进程。

下一步是变换数据库，从最初的按交易的组织 $\{T_x, A, B, D, \ldots\}$ 到按物品集的组织 $\{IS_x, T_1, T_2, T_3 \ldots\}$，其中 IS_x 最初是大小为 2 的物品集。这可以通过两步来完成——一个本地步骤和一个通信步骤。在本地步骤，进程从它所分得的数据库部分中，针对大小为 2 的大物品集构造部分交易表。在通信步骤，根据 1-等价类的所属关系，一个进程（至少是概念上的）将它所得到的大小为 2 的物品集表“分发”给相关的进程；从其他进程“接收”分发给它的等价类表。并按词典存储次序，将从各个进程收到的部分表合并成本地表；在这之后，进程对于分给它的等价类就有了转换后的数据库。它现在就可以对它的每一个等价类，一步一步地计算大小为 k 的物品集，不需要通信、同步或者远程磁盘访问（如果不做任务窃取的话）。在计算结束时，大小为 k 的物品集合就存在于不同的进程之中。变换阶段的通信通常是本算法中代价最大的，很像进行一个矩阵的转置，不同的是在不同的进程对之间通信的大小可能是不同的。

3. 协调

181 在这种分配和分解下，让我们考察空间局部性、时间局部性和同步。

空间局部性　计算的组织将物品集合和交易按词典序的存放使得大多数通过数据的遍历只是简单地从前往后的遍历，表现出很好的可预测性和空间局部性。这对于从磁盘中读数据是特别重要的，这样可以在大量有用数据上分摊很高的磁盘读启动代价。

时间局部性　如前面所讨论的，每次在一个等价类上的操作很像成组操作，它的成功与否取决于那个等价类的数据是否能放在主存中。随着等价类计算的进行，大物品集合的数量变小，这样在主存中的重用更可能被开发出来。注意，这里我们更像是发掘在主存中的时间局部性，而不是缓存，尽管技术和目标在扩展存储层次的任何一层都是类似的。

同步　在算法的第一步（计算大小为 2 的大物品集合）之后，需要有一个栅障同步，保证将部分计数归约到全局计数，然后开始数据库的变换阶段。归约只是对大小为 2 的物品集合需要；在此之后，每个进程独立地继续计算它自己的等价类中的大小为 k 的大物品集合。如果用动态任务管理来做负载平衡，也许需要有进一步的同步。

4. 映射

变换数据库的通信是“全部对全部”：一个进程可能将不同的大小为 2 的物品集合和它们的部分交易表“分送”给不同的进程，同时可能从所有其他进程“接收”或者读得这样的表。将全部对全部的通信，以一种无冲突的方式映射到互连关系不是很多的网络拓扑（像网格或者环）是困难的。终点连接可以通过调度技术减少，例如可以使每个处理器 i 在步骤 j 和处理器 i xor j 交换数据，从而没有处理器或者节点被重载。

5. 小结

由于磁盘访问是主要的瓶颈，数据挖掘不同于其他几个应用实例，因此并行化技术主要瞄准磁盘访问的代价。前面已经分析过的技术将磁盘简单地看作是另一个、显式的扩展存储层次中的一个管理层次。这里负载平衡是特别重要的，考虑不好就可能抵消这个并行程序中某些局部性质所带来的好处。

3.6　编程模型涉及的问题

通过前面几章的讨论，我们已经看到并行程序的分解和分配通常是（不一定总是）独立于程序设计模型的，但协调步骤却和它高度相关。在第 1 章中，我们阐述了根本的设计问题，适用于任何通信体系结构层次和程序设计模型。我们了解了两种主要的程序设计模型，共享地址空间和在私有地址空间之间的显式消息传递；并由功能上的差别（例如命名、复制和同步）将它们从根本上区别开来。尽管这两种程序设计模型都能在任何通信抽象和硬件上实现，但在一个给定的层次所采取的关于这些功能性问题的做法影响（同时被影响）着性能诸要素，例如开销、时延和带宽。在此阶段，我们只是以一种抽象的形式来对待这些问题，不能体会它们和应用的相互作用，以及哪种程序设计模型在什么条件会有更好的影响。现在我们已经有了对几个典型并行应用的深入理解，理解了在协调阶段的性能问题，我们就能从应用和性能特点的角度来对程序设计模型进行比较。 182

假定有一个一般化的物理分布存储多处理器体系结构，下面我们用这些应用事例来解释上述问题。对于共享地址空间来说，我们假定对共享数据的读（装载）和写（存放）是用户能看到的仅有的通信机制，我们称这是读-写共享地址空间。当然，在实践上共享地址空间并不是除了这些原语外就不能有对显式消息传递的支撑，但我们现在忽略这种可能。共享地址空间模型可以用很多方法支持，包括通信抽象和硬件/软件接口等（回顾在第 1 章结尾部分的命名模型），它们在支持通信、复制和不同粒度的一致性上有不同的效率。这些因素影响着程序设计模型的成功与否，具体将在第 8、9 章详细讨论。这里，我们集中的看一看最常见的情况，其中缓存一致性共享地址空间在细粒度上得到高效支持——例如，对共享物理地址空间提供直接的硬件支持以及在高速缓存块固定粒度上的通信、复制和一致性。同时也将这种常见的情况和硬件支持的不带一致性复制的共享地址空间进行对比，正如 BBN Butterfly 和 CRAY T3D 和 T3E 机器所提供的。对于消息传递程序设计模型，我们将假设它也是通过通信抽象和硬件/软件接口得到高效支持的。

作为应用程序设计员，我们将程序设计模型看作是我们对于通信系统结构的窗口。程序设计模型之间的区别和它们如何实现的对程序设计的许多方面都有影响，包括程序设计的容易程度、通信结构、性能以及可扩展性等。除了功能方面（像命名、复制和同步），还有组织方面（像通信的粒度）和性能方面（像通信操作的端点开销），它们对不同的程序设计模

183 型各不相同，影响针对性能的程序设计。其他的性能方面（例如时延和可获得的带宽）主要取决于所用的网络和网络接口，可以认为是等价的。除此以外，在硬件开销上有差别；在要求高效支持抽象的复杂性有差别；在使我们能够推导和预测性能的容易程度上也有差别。下面我们一一进行讨论。首先考虑的三个方面——命名、复制和通信开销，表明读-写共享存储有优势；而其他方面——消息大小、同步、硬件或者设计代价、以及性能的可预测性——显式消息传递有好处。

3.6.1 命名

如我们所见，共享地址空间使程序员能容易地给逻辑上共享的数据命名，任何进程能够直接引用任何数据，其命名模型类似于单处理器。显式消息在这里是不必要的，一个进程不需要给其他进程命名或者需要知道哪个处理节点当前拥有它所需要的数据。在规则的、通信需求静态可知的应用中，诸如方程求解器内核和 Ocean，确定数据驻留在哪个进程的地址空间没什么困难，因此用显式消息也很容易。但对于非规则、不可预测的数据需求应用，要找到数据的拥有和使用之间的关系就是相当困难的，算法上和程序设计上都如此。一个例子是 Barnes-Hut，星体和树的单元的拥有关系是随时间变化的，因此一个进程为了计算它所负责的星体所受的力，需要遍历树的哪一部分，不是静态可确定的，确定要和哪个进程通信引起运行时的额外工作。在光线跟踪例子中，由一个进程负责的光线在场景数据中不可预测地反射，因此如果数据分布在进程的私有地址空间中，难以确定谁拥有所需的数据。这些困难可以被克服，但要求要么改变和增加算法的复杂性（例如，在 Barnes-Hut 的每一时间步加进一个额外的阶段，来计算谁需要什么数据，然后传送那些数据［Salmon 1990］或者在光线跟踪中改用面向场景的方法），在所有节点上复制整个共享数据结构（但这不是个扩放性好的办法），或者用软件模仿一种和特定应用相关的共享数据结构，将星体和单元或者场景数据散列到处理节点上。这些应用层的命名解决方法大大改变了程序的外观，通常是运行时开销最大的来源。它们在（Singh，Hennessy，and Gupta 1995；Singh，Gupta，and Levoy 1994；Warren and Salmon 1993）中有进一步的讨论。

3.6.2 复制

有几个方面的要素用来区别非局部数据复制的不同管理方式：1）谁负责复制，即制造
184 本地数据的拷贝？2）复制在本地存储层次中的什么地方发生？3）在复制存储区中数据以什么粒度分配空间？4）复制数据的值如何保持一致性？5）复制数据的替换如何管理？

对于消息传递模型中分离的、私有虚拟地址空间，复制通信数据的惟一方式是在应用程序中显式地将数据拷贝到进程的私有地址空间。复制的数据在新的地址空间显式地重新命名，因此对两个进程来说，虚拟地址和物理地址可能是不同的；从系统的角度来看，它们的拷贝相互之间没有任何关系。数据总是首先复制到主存（当拷贝发生时），进入处理器高速缓存的只有本地主存的数据。在本地存储器分配的粒度是可变的，它取决于用户。对于复制数据更新的保证要由程序通过显式消息来得到。我们即将讨论替换。

回顾在共享地址空间的情况，由于数据是通过普通处理器的读和写来访问的并且通信是隐式的，系统可能对用户透明地复制数据——不需要拷贝和在程序中显式重新命名——就像单处理器情况中的高速缓存。这就导致了很多可能性。例如，在共享物理地址空间系统，非

本地数据在访问时透明地进入处理器高速缓存子系统，不是复制在主存。复制以高速缓存块的细粒度发生，数据由硬件来保持一致性。其他系统可能先透明地在主存复制数据——或者通过附加的硬件支持以高速缓存块的粒度，或者通过系统软件以页面或者对象的粒度——而且可以保持一致性的方法和粒度有很多，将在第 9 章讨论。还有一些系统可能选择不支持透明的复制和/或一致性，而是让用户来管理（例如在 CRAY T3D 和 T3E 系统）。

最后，让我们考察由于有限容量所导致的本地复制数据的替换。替换的管理方式对所要复制的通信量有影响，每次涉及本地存储层次的一个层次。例如，硬件高速缓存动态管理替换，针对的是每一次引用并且空间粒度很细，因此高速缓存只需要像工作负载的活跃工作集那么大。在消息传递中，当用户来管理复制的时候，可以通过在应用中维护一个本地高速缓存数据结构获得一种类似的效果，可以看成是一个对非本地数据的硬件高速缓存。然面，因软件要查询和进行地址映射，管理这个缓存使程序设计变得复杂；同时当高速缓存失效时自然地要产生细粒度消息，从而引起运行时开销。另一方面，软件高速缓存能够做得很大并可以用一种和应用相关的方式管理。

典型地，消息传递程序对复制替换的管理通常缺少动态性。通信的数据在本地存储器有显式的拷贝并在程序的某些点显式地被清除；典型的是在计算的一个阶段之后，确定了在后面的一段时间里不再需要那些数据的时候。在某些非规则应用中，所需的复制可能要求大量的额外存储空间，例如 Barnes-Hut 和 Raytrace。对于像光线跟踪例子中场景那样的只读数据，许多消息传递程序简单地在每一个处理节点复制整个数据集合，这就解决了命名问题并几乎完全消除了通信，但也没有了在较大系统解决较大问题的能力。在 Barnes-Hut 应用中，人们提出过一种很有特点的方法（Salmon 1990），该方法中的进程首先在本地复制所有它所需要的数据（针对属于它的体），然后再开始受力的计算。这意味着在受力计算阶段没有通信，和进程分得的数据划分相比，此时在主存复制的数据量要大许多，从而肯定比活跃工作集要大得多（我们知道，这里的工作集只是计算一个星体受力所需要的数据）。这个工作集在典型的共享地址空间系统中能放在处理器高速缓存中，因此根本不需要在主存复制数据。在消息传递中，大量的复制可能限制方法的可扩展性。由于这些原因和其他一般性问题，对于非规则数据访问类型的应用，通过软件利用哈希方法来仿真共享地址空间，用一个固定大小的软件高速缓存（为保持一致性，该高速缓存内容在计算阶段的边界被清除）更动态地管理复制的做法正变得越来越流行。当消息传递系统对小消息越来越高效的时候，尤其如此。

185

3.6.3 通信的开销和粒度

启动和接收消息的开销在很大程度上受通信活动的完成在软硬件之间分工的影响，而在软件中，操作系统的影响会特别大。回顾在共享物理地址空间中，由于共享地址空间只是一个大的线性地址空间，底层的单处理器硬件机制足以完成地址翻译和保护（即使存储器物理上是分布的）。在一组实验中（Scales and Lam 1994），简单地用软件对共享数据做地址翻译，相对于硬件来说，降低 Barnes-Hut 的性能大约为 20%。开销的另外主要成分是缓冲区的管理：入向和出向的通信需要暂时缓存在网络接口中，以允许多个通信同时进行，在一个通信流水线里排队等待。在固定字粒度或者高速缓存块粒度的通信使得用硬件高效管理缓冲区变得容易。这些因素组合起来，在高速缓存一致性共享地址空间机器上，通信使每个高速缓存块的开销相当低（几个周期到几十个周期，取决于实现和通信辅助设施的集成）。另一方面，

如果空间局部性不好的话，固定大小块的自动传送可能导致大量的附加通信.

事实上，对比一个高速缓存一致性共享存储空间和那种提供透明的命名但没有一致性复制的系统（如 CRAY T3D 和 T3E），通信粒度的问题反映了一个重要的、难以把握的区别。对于前者，通信的完成是透明的，但其粒度比所引用的字要大（例如，高速缓存块）。于是通信的代价被分担了，不用程序员来保留所传送数据的其他部分的一致性问题，一切由系统负责。对于后者，复制和一致性是程序员的责任；于是当发生扑空时，系统只取要引用的字（否则，程序员的负担可能就太大了）。

186

在消息传递系统中，本地引用所带来的开销不比单处理器大。不过，通信消息是很灵活的，因此有很大的开销。消息类型的种类繁多，要求软件开销来对消息的类型编解码，这要通过发送和接收端点执行相应的处理程序来完成。灵活的消息长度、异步和非块化消息的使用使缓冲区管理变得复杂，因此常常要启用系统软件来临时存放消息。最后，在任意的地址空间之间发送显式消息，要求节点（或硬件支持）上的操作系统的干预来提供保护。特别是当操作系统必须被启用时，缓冲区的管理和保护的软件开销可能是很大的。最近大量的设计努力集中在将网络接口和消息传递机制合理化，以大大降低每一个消息的开销。这些方法会限制灵活性，将在第 7 章讨论。尽管如此，和硬件支持的读-写共享地址空间接口相比，每个消息的开销可能保持几倍大，这就限制了软件方法在那些自然产生细粒度通信的非规则应用中的有效性。

这三个问题（命名、复制和通信开销）表明了有高效支持的共享地址空间系统对于并行程序设计的优势。下面分析有利于消息传递的要点。

3.6.4 块数据传送

在硬件支持的高速缓存一致性的共享地址空间系统，通过读和写的隐式通信典型地引起一个消息的产生，这是针对每个引用或者至少对每个要求通信的高速缓存块发生的。通信通常由需要数据的进程发起，可能经历缓存扑空，我们称之为接收者发起的通信。硬件支持提供了高效的细粒度通信，但对于将大块的数据从一个进程传到另一个进程，每次和一个高速缓存块通信不是最高效的方法。我们更愿意通过用一个消息或者一组大消息来做数据通信，从而分摊开销和时延，这种方法称为*块数据传送*。

显式通信，如同消息传递，在选取消息的大小，在选择是否由接收方发起还是由发送方发起的通信的时候，允许较大的灵活性，这样自然地就使块传送成为可能。显式通信甚至能加到一种硬件一致性共享地址空间命名模型中，给程序员通信方法的选择并且在某些可预测通信的情形下，还可能由系统透明地在一个读写程序设计模型之下来做较粗粒度的通信。然而，由共享地址空间表现出的自然的通信结构是细粒度的，通常是接收方发起的。由于时延容忍技术的采用，在硬件支持的共享存储空间的块传送的优势多少有些复杂，但它显然是有优点的。

187

3.6.5 同步

在消息传递中，同步可以隐含在显式通信中；而在共享地址空间中，同步常常是显式的，是和隐式数据通信分离的。这种情形使我们在消息传递程序设计时基本上可以不考虑同步问题。互斥自动就达到了，不要用什么标志。这样做，那些难以捉摸的冒险条件和时序错误在消息传递中可能要少一些。除此以外，细粒度共享和复制的困难趋于使程序员用更多的

结构化，有时是更初等的算法来做简单的协调。然而，当用异步消息传递的时候，这个优势就不那么明显了；在异步消息传递的情况下，必须要用单独的事件同步来保证正确性。

3.6.6 硬件代价和设计复杂性

对于共享地址空间，得到所期望效果的硬件代价和设计时间要大于支持消息传递的模型。由于必须要看见所有存储事务，以确定何时非本地的高速缓存扑空发生，通信辅助电路的某些功能必须和处理节点相当紧密地集成。一种用硬件高速缓存实现透明复制和一致性保证的系统要求更多的硬件支持并实现相当复杂的一致性协议。在消息传递抽象中，辅助电路不需要看到存储引用并且可以不用那么紧密地集成，例如，可能就集成在I/O总线上。在第5、7和8章，我们要讨论支持不同抽象的实际的硬件代价和复杂性。

不过，代价和复杂性是要比辅助硬件成本和设计时间更复杂的问题。例如，同高速缓存一致性共享地址空间所发生的复制数据量相比，如果在一个消息传递程序中需要复制的量要大很多（由于不同的复制管理方式或者由于操作系统的复制），那么为这种复制所需的存储量应该和支持共享地址空间的硬件代价比较。同样的观念也适用于在一个机器上开发有效的程序所引发的代价和“设计时间”。协议的设计代价也随经验的增长降低。在实践上，成本和价格在很大程度上由销售量、工程设计经验和经营来决定，而不是纯技术的因素。 188

3.6.7 性能模型

最后，在针对一个体系结构设计并行程序的时候，我们希望至少有一个粗糙的性能模型。用这样的性能模型，我们能够预测一个程序的一种实现是不是要比另一种实现好，由此来引导通信结构的形成。一个性能模型有3个方面。首先，我们必须对机器的特点建模，例如，关键的系统操作粒度和基本事件（如通信消息）的代价。第二，我们必须建立应用特点的模型，例如，在并行程序中的基本事件的频率和突发性。第三，我们必须有一个能解析表达的或者数值模拟的性能模型，取上述两种特点的集合作为输入，预测执行时间。为系统特点建模通常并不是很困难，我们在本章已经看到了一个简单的通信代价模型。然而，建立应用特点的模型是相当困难的，特别是当应用是复杂和非规则时。当冲突是一个不可忽略的因素时，开发一个好的解析性能模型是困难的。正是由于建立应用特点模型所带来的困难，预测共享地址空间系统中的性能比消息传递要难，因为对于前者我们感兴趣的一些事件在程序中表达不出来。对于程序员来说，在消息传递中最基本的性能规则至少是清楚的；消息有代价；最好不要经常发送它们。在共享地址空间，特别是带有一致性复制的系统，正是由于它有使得程序设计容易的性质：命名、复制和一致性都是隐含的（即对程序员透明），因此难以确定有多少通信出现，何时出现，从而导致性能建模复杂化。附加通信也是隐含的，并且特别难以预测。（考虑那些产生通信的高速缓存映射冲突！）结果，对程序设计的指导也就模糊得多。我们只能讲：当必要时，通过数据布局来开发时间和空间局部性，以控制通信量的增大。这里的问题类似于在单处理器上的情形，隐含的高速缓存使得在单处理器上也难以预测性能，从而使用简单的冯诺依曼模型来分析计算机难以奏效了，因为该模型假设所有引用是同等代价的。然而，我们这里的问题要大得多，这是因为通信的代价要比单处理器上局部存储的访问代价大得多，于是有更大的发生竞争的机会。

3.6.8 小结

共享地址空间模型的隐式通信给我们带来的主要的潜在好处是程序设计容易以及细粒度数据共享方面的性能（至少当模型有硬件支持时）。显式通信的主要的好处，如在消息传递中，是块数据传送，在消息传递中包含同步、较好的性能导向和预测能力以及系统比较容易建造。

189

给定这些权衡，系统结构设计师要回答的问题是：

- 是否值得为共享存储提供硬件支持（即透明的命名）；软件支持是不是就够了或者由程序员来显式管理所有通信是不是本来就很容易？
- 如果共享地址空间是值得的，是不是也值得提供硬件支持来实现透明的复制和一致性？
- 如果对上面问题的任何一个回答是肯定的，隐式通信是不是就够了？或也要有处理可以使用的节点之间的显示消息传递的硬件支持？

对这些问题的回答取决于应用特点和代价。对任何一个问题的肯定回答自然导致其他问题，诸如这些特征应该怎么高效支持，在什么粒度等；这就又提出了代价、性能和程序设计权衡。这些问题将随着本书内容的进展而变得清晰起来。经验表明，随着应用变得更复杂和更非规则，透明命名和复制的有用性增加，这是倾向于共享存储的观点。然而，由于通信自然是细粒度的（特别是非规则应用），还由于大粒度通信和一致性引起的性能问题，支持共享空间就要求有大胆的通信系统结构，用硬件来支持大多数功能。许多计算机公司现在正建造这样的机器作为它们的高端系统。另一方面，便宜的工作站群或者多处理器群也正在逐步流行起来。这些系统通常是消息传递程序设计的，这是由于消息传递的性能模型比较好定义，可以用大消息来分摊开销，显式的控制以及不同的机器操作粒度对性能的相对影响不大。

3.7 结论

并行程序的特点对于设计多处理器体系结构有重要的影响。对于程序行为的某些关键的观察结果曾导致了单处理器计算中某些最重要的进步：在程序访问模式中时间和空间局部性的识别导致高速缓存的设计，指令使用情况的分析导致合理化指令集合的设计。在多处理器中，由于应用需求和体系结构所能提供的支持之间的不匹配所导致的性能的丢失要大得多，因此对运行在这样的机器上的并行程序和其他工作负载理解是更重要的。

从历史上看，许多不同的并行体系结构曾导致许多不同的程序设计风格，但有不好的可移植性。现在，体系结构的统一化趋势导致了一种共同的基础来开发可移植的软件环境和程序设计语言。对于共享存储空间和消息传递程序设计模型，尽管特定的粒度、性能特点和协调技术不同，我们考虑并行化进程的方式以及许多关键的性能问题在很大程度上是类似的。当我们分析共享地址空间和消息传递之间折中的时候，两者的技术特征在体系结构设计空间的不同部分不断丰富起来。

190

体系结构融合的另一个效果是有了一个更清楚的关于性能问题的理解，从而可以根据它来设计软件。历史上，理论并行算法研究的主要注意力是 PRAM 模型，它忽略数据访问和通信代价，只考虑负载平衡和额外工作（PRAM 模型的某些变形也涉及了当不同的处理器试图访问同样数据时的串行化效果）。PRAM 模型对于理解应用中的固有并发性是很有用的，这

是开发并行程序的第一个概念性步骤；然而，它没有考虑现代系统中重要的现实，诸如数据访问和通信代价常常在总执行时间中起支配作用。历史上，通信是单独处理的，对它的处理中的主要考虑是将通信映射到不同的网络拓扑上。随着我们对通信的重要性，以及在现代机器中一次通信中主要开销的理解更加清晰，发生了两件事情。首先，人们开发了有助于分析通信代价，从而改善通信结构的模型，例如大同步程序设计（BSP）模型（Valiant 1990）和 LogP 模型（Culler et al. 1993），希望取代 PRAM 来作为并行算法分析的基本模型。这些模型力图揭示和一次通信事件相联系的重要的代价（诸如时延、带宽或者开销）就像我们在本章所说的那样，允许一个算法设计人员来分别考虑它们，对比地进行并行算法的分析。BSP 模型还提供了一个漂亮的框架，能用来讨论通信和并行的性能。其次，对通信代价模型方面的强调，已经移到了通信消息端点的节点，因此在端点的消息和竞争数变得要比映射网络拓扑重要得多。事实上，BSP 和 LogP 模型完全不考虑网络拓扑，而是用一个常数值作为网络上延迟的模型！

人们盼望着有一个现实的系统结构模型，使得我们能在它上面设计和分析并行算法。BSP 和 LogP 是在这个方向上重要的进步。改变这些模型中的关键参数，我们就可能确定一个算法在体系结构的一个范围上表现得怎样，怎么能够最好地构成以适应不同的系统结构或者是可移植的性能。然而，比用几个参数建立体系结构模型更困难的是建立并行算法或者应用行为的模型，尤其是当它的结构是非规则时。这是建模方程的另外一个方面（Singh, Rothberg, and Gupta 1994）。这里的关键问题是：什么是通信计算比？它怎么随着系统复制容量的变化而变化？访问模式是如何同扩展存储层次的粒度相互作用的？通信的突发情况如何？这些怎么能够结合到性能模型中来考虑？能够概括真实应用中的这些特点，并将它们集成到如同 BSP 和 LogP 那样的机器模型的建模技术尚待开发出来。 191

本章讨论了并行程序某些关键的性能特性，以及它们和多处理器的扩展存储层次和通信结构所提供的基本功能之间的相互作用。这些特性包括负载平衡，通信计算比，协调通信的影响代价的各个方面，数据的局部性和它和复制容量以及存储分配、传送和一致性（产生附加通信）粒度的相互作用，通信抽象和机器可能支持的硬软件界面的影响。我们已经看到，性能的方方面面是相互制约的，设计好的并行程序的艺术在于在相互矛盾的要求中获得适当的折中。以高性能为目标的程序设计也是一个逐步求精的过程：在后面所发现的系统或者程序的特点，可能导致在前一个步骤中所做的决定被修改。将性能的潜力都发挥出来可能需要很大的努力，这取决于应用和系统两个方面。进而，不同技术一起发挥作用的程度和方式能够大大影响表现给体系结构的工作负载的特点。通过第 2 章的介绍，我们已经深入考察了 4 个应用实例，看到了这些问题如何在其中起作用。在本书的其他部分，当我们考虑体系结构的设计选择、权衡和评估时，我们还将更详细的讨论这些性能问题。然而，根据已经得到的并行程序设计的知识，我们现在已经知道了如何用这些程序作为工作负载来评估并行体系结构和权衡。

习题

3.1　对我们所描述的几种应用（Ocean、Barnes-Hut、Raytrace、Data Mining），在并行化过程中，其中哪些我们是分解数据、用拥有者计算规则来确定计算的发生，而不是直接分解计算？在其他的应用中，如果用严格的数据分布和拥有者计算规则，会有什么问题？

你怎么对待这些问题？

3.2 相对于全局任务队列来说，用分布式任务队列来实现负载平衡的优缺点是什么？在两种情况下，小任务都一定会增加通信、竞争和任务管理开销吗？

3.3 对于下面的表，从每一种存储系统流量（表的左边）向其解决技术（右边）画一条弧，所谓解决技术是在一个机器中（带有物理上分布存储的共享地址空间）降低流量源的最有效的办法。

192

存储系统流量的种类	解决技术
冷启动流量	用较大的高速缓存
固有通信	数据摆放
扑空后的额外数据通信	算法的重新组织
由于容量产生的通信	设计较大的高速缓存块
由于容量产生的局部流量	数据结构的重新组织

3.4 假设串行和并行程序都是确定性的，问在什么条件下，进程有效忙的时间之和不等于串行程序有效忙时间？给出一个例子。

3.5 作为层次式并行性的一个例子，考虑经常用于医疗诊断和经济预测的一个算法。这个算法通过一个网络或者图（如图 3-22 所示）传播信息，每个节点表示一个矩阵。弧对应于节点之间的依赖关系，也是信息流动的通道。这个算法从图的底部的节点开始，向上发展，在每个碰到的节点上做矩阵运算。它至少在两个层次上表现出并行性：在遍历中那些没有辈分关系的节点能并行计算；在一个节点内的矩阵计算也能并行化。你会如何并行化这个算法？这个网络或图的什么特点最能影响你的决定？什么是最重要的折中？

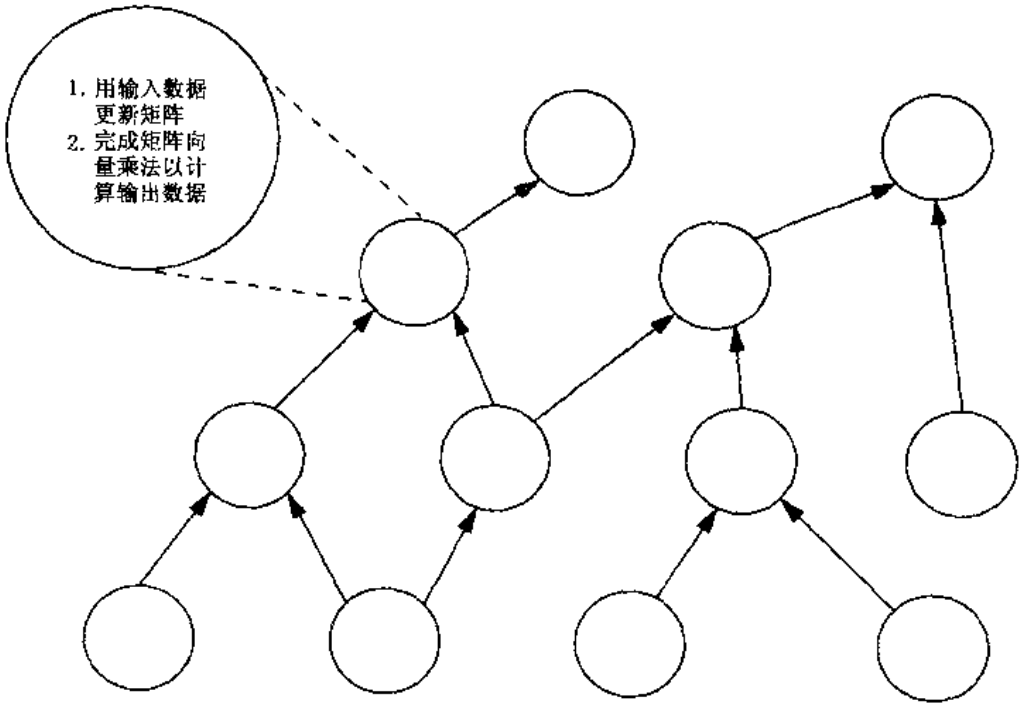


图 3-22 在一种图计算中的并行性层次。在图的节点内的工作由左边的扩展节点所示

3.6 为了解释并行性的层次，本章描述了一种应用，在 VLSI 芯片或板上布线连接点。有三种级别的并行性存在：线路之间、一个线路内的线段之间、对于给定的线段上可能的布线方式之间。在确定选取哪种级别时，权衡是什么？输入和机器的什么参数影响你的决定？对下面的情形你选取什么：30 根线路、24 个处理器、每线路有 5 段、每段有 10 种布线法、每条布线评估用同样多的时间？如果你必须选择一个级别，并且要用于所有情形，你会选什么？（你可以声明你的合理的假设来引导你的答案。）

3.7 如果 E 是通过并行化增强的算法的段落集合，f_k 是第 k 个段落在单处理器上串行执行的时间；s_k 是通过并行化在第 k 个片段上得到的加速比，推导出整体加速比表达式。将它应用于高斯消去法中以元素为粒度的广播方式。对于这个计算画出一个粗略的并发性形态（一个图，表明相对于时间的并发量，时间单位是一种逻辑的操作，比如更新内部的一个活跃元素）。假设 100×100 个元素的矩阵。估计加速比，忽略存储引用和通信代价。 193

3.8 考虑在习题 2.7 ~ 2.10 中讨论过的并行高斯消去算法。

1）画出一个并发性态图，表示“广播”版本随时间可用的并发性。假定对于网格元素每次更新是一个计算和时间单位。

2）对于 $n \times n$ 矩阵和 p 个进程，分析负载不平衡和通信量，假定给进程分配连续若干行。

3）对行向进程交织分配的情形做同样的分析。

4）对于流水版本做同样的分析，分解按照行来做。

3.9 高斯消去法中的并发性也能够增强，通过分解成个别元素，而不是行。为什么？

1）对于这种情形，画出一个针对广播版本的并发性形态。

2）一种二维散播（二维交织或者切饼（cookie-cutter））分配能用在个别元素的粒度，而不一定是行分配。分析在这种情形下的广播版本的负载不平衡和通信量。 194

3）分析负载不平衡和通信量，对于流水版本，假定二维元素的交织分配。和有相同分配的广播版本相比，负载不平衡和通信量有大的区别吗？

4）在本题和上一题中，所讨论的哪一个版本在一个真实机器上实际表现的最好，为什么？你能想出一种更好的分解和分配吗？

3.10 我们已经讨论了广泛用于线性代数算法，以开发时间局部性的成块技术（见 3.3.1 节）。考虑一种串行高斯消去程序。

1）用 $B \times B$ 块划分方式，写一个成块的串行版本。

2）对于原始（非成块）和成块的串行程序，给出一个有关 n 和 B 的读扑空率表达式。假设在非成块版本中，矩阵的一行不能放在高速缓存中，而在成块版本中 B 的选择是 $B \times B$ 块大约能占高速缓存的一半。忽略高速缓存冲突，只对矩阵元素的访问计数。对于大小为 16 KB 的高速缓存，$1\ 024 \times 1\ 024$ 个元素的矩阵和块大小 $B = 32$，什么是读失效率（在两种情形下）。假设没有跨块操作的块重用。如果读失效占用 50 个周期，什么是两种版本之间的性能差别（将每个网格点的更新计算计为一个周期，忽略写访问）？

3）你怎么来划分这个成块版本用来并行执行，假定以一种广播方式？写出伪代码，将一个块的计算看成是一个伪操作。

4）分析这种情况下的负载的不平衡和通信，和先前广播方式的划分方法进行比较。

5）考虑所有的性能问题（不仅仅是算法层次），在共享存储地址空间机器中，你会用最好的成块或非成块的版本用于并行广播方式吗？对消息传递系统你会用哪种？为什么？

6）考虑流水方式（成块或不成块），你会总体上用哪种方式来对付共享地址空间机器和消息传递机器？

3.11 终止检测是任务窃取的一个有趣的方面。考虑一种任务窃取的情形，其中进程随着计算的进行产生任务。设计一种好的任务分配方法（从哪儿取任务，如何将任务放进池中等），考虑某种好的终止检测启发式方法。对于你认为的终止检测方法需要的消息数，做最坏情况的复杂性分析。在实践中你会用哪一种方法？对于那种保证会正确工作和应该给出好性能的方法，写出伪代码。

[195]

3.12 考虑在并行中转置一个矩阵，从一个源矩阵到一个目的矩阵（即 $B[i, j] = A[i, j]$）。

1）你会如何在进程之间划分这两个矩阵？讨论一些可能性和折中。这对于共享地址空间和消息传递机器有区别吗？

2）为什么在矩阵转置中的进程间通信称为全部对全部的个性化通信？

3）对于并行矩阵转置，写出在共享地址空间和消息传递系统中简单的伪代码（只是实现转置的循环）。除了固有通信和负载平衡外，在每种情况，什么是你考虑的主要性能问题，你怎样对待它们？

4）对于这个并行矩阵转置，成块会有什么好处吗？在什么条件下？你如何对它进行分块？（没有必要写出完整代码）。如果有的话，高斯消去法和这里分块的区别是什么？

3.13 应用的通信需要，即使简单地表达为执行每条指令所要通信的字节数，也能帮助我们对背后的计算过程有一定的认识，来确定增加带宽和降低时延的影响。例如，快速傅立叶变换（FFT）是广泛用于信号处理和气候建模应用中的算法。一个简单的在 n 个数据点上的并行 FFT 在每个进程上有计算代价

$$O\left(\frac{n\log n}{p}\right)$$

和通信量 $O(n/p)$，其中 p 是进程个数。通信计算率因此为 $O(1/\log n)$。为简便起见，假设前面表达式中的所有常数为单位，我们要完成 $n = 1$ M（或 2^{20}）点的 FFT，在 $p = 1\,024$ 进程上。让一个字数据（FFT 的一个点）的平均通信时延是 200 个处理器周期，让任何节点和网络之间的通信带宽是 100 MBps。假设没有负载不平衡的问题，没有同步代价，忽略网络中的竞争。

1）没有时延隐藏，有多少执行时间是由于通信时延而导致进程停滞？

2）通信时延减半，对执行时间的影响如何？

3）没有时延隐藏，什么是节点对网络带宽的需求？

4）什么是节点对网络带宽的需求，假设所有时延都隐藏了，这个机器能满足它们吗？如果时延没被隐藏，那么什么（定性）是影响因素？

[196]

3.14 考虑利用复制来减少数据流量

1）哪种类型的数据（本地、非本地或者两者）能够构成相关的工作集在 i）消息传

递抽象中的主存？ⅱ）消息传递抽象中的处理器的高速缓存？ⅲ）在高速缓存一致性共享地址空间中处理器高速缓存？

2）有一种关于高速缓存一致性机器的提议，是要对细粒度、主存中一致性复制也提供硬件支持。你认为这值得吗？在什么情况下，什么是主要的缺点？对本章中的哪一个应用例子来说，这种设想有可能是有益的？

3.15 写一段在 p 个进程之间做归约和广播的伪代码：首先用线性的 $O(p)$ 方法，然后用一个基于树的 $O(\log p)$ 方法。分别给出对共享地址空间和消息传递的结果。

3.16 用一种四维数组表示网格，写出在共享地址空间的方程求解器内核，要求连续的划分的形状（例如，条状还是块状或者是网格每一维进程的个数）能够通过程序的输入来指定。

3.17 在将任务分给处理器后，处理器按某种时间顺序调度分给它的任务的问题依然存在。什么是这里涉及的要点？哪些是和单处理器相同的，哪些是不同的？构造例子，显示在不同情况下不好的调度的影响。

3.18 在数据挖掘的例子中，为什么大小为2的物品集合是从原始数据库格式中，而不是从转换后的格式中计算出来的？从计算复杂性的角度分析两种做法的不同。

3.19 假设你接到一个任务，要为某个大的出版商开发一个单词计数程序。你的工作环境是32个处理器的共享内存系统。给定的惟一接口是 get_words，它的参数是一个数组，返回时在数组中放着下一批需要计数的1 000个单词。每个处理器要做的主要工作大致为：

```
while(get_words(word)) {
   for (i=0; i<1000; i++) {
     if word[i] is in list
       increment its count
     else
       add word to list
   }
}
/*Once all words have been logged, the list should be printed out*/
```

试用伪代码给出一个并行程序的控制流和数据结构的一个详细的描述。你的方法应该试图极小化空间的使用、同步开销和存储延迟。求解这个问题有很大的灵活性，因此请陈述清楚你所作的各种假设和具体的设计决定。

第 4 章　工作负载驱动的性能评价

计算机体系结构领域变得越来越定量化，只有在对折中方案的详细评价完成后才能决定采用什么样的设计特性。系统一旦建立，理解折中方案的体系结构设计者和做出购买决定的用户都会对系统进行评价和比较。在单处理器设计中，有现存的机器及在其上面运行的应用这样丰富的基础，识别和评价折中方案的过程体现出从已知量出发的细致推断。设计者通过使用一些微基准测试程序——即一些强调机器特定性质的小程序，来分离机器的执行特征。一些流行的工作负载已经被编写在标准的基准测试程序集中了，比如，标准性能评价公司（Standard Performance Evaluation Corporation，SPEC）的工程计算类工作负载基准测试程序集（SPEC 1995），其测量是建立在一系列现存的不同设计方法之上的。基于测量、对新兴技术的评估以及应用需求的预期变化，设计者提出一些新的方法。对其中有前途的方法通过模拟来进行典型的评价。首先，要写一个模拟器。模拟器是一个程序，它模拟那些具有或者不具有所建议的特性的设计。其次，要选择一定数量的程序或者多道程序工作负载，它们可以是来自标准的基准测试程序集，或者是那些很可能要运行在该机器上的工作负载的代表。这些程序运行在模拟器上，机器特性对性能的影响也就决定了。特性的性能和实现该特性的硬件以及设计该特性的时间的预期成本一起，决定了是否要在机器中包括这种特性。模拟器要写得灵活，这样可以通过改变组成结构和性能方面的参数来理解它们的影响。

好的工作负载驱动的评价是一个困难而且耗时的过程，即使对于单处理器系统也是如此。当技术和使用模式发生变化时，工作负载需要更新。作为工业标准的基准测试程序集每隔几年就要修正一次。特别是，程序使用的输入数据集会影响到与系统的一些关键相互作用，并且决定是否能突出对系统的这些重要特性的评价。我们必须理解这些相互作用，并且反映到工作负载的使用中。比如，考虑到处理器速度的剧增和高速缓存尺寸的变化，从 SPEC92 到 SPEC95 的基准测试程序集的一个主要变化是使用了更大的输入数据集来加强存储器系统的测试。当然，精确的模拟器的开发和验证是很昂贵的，而且模拟程序的运行会消耗大量的计算时间，但是，这些努力仍是值得的，因为一个好的评价会产生好的设计。

随着多处理器体系结构的成熟、从一代机器到下一代机器之间有着更多的连续性，人们采用类似的定量的评价方法。尽管早期的并行机设计在很多方面像艺术品的大胆创作，很大程度上依赖于设计者的直觉，但现代的设计却包含了对于所建议的设计特性的大量的评价。这里，工作负载既用来评价真正的机器，也用来推断所建议的设计是否合理，通过软件模拟来探索可能的折中方案。对于多处理器，所感兴趣的工作负载可能是并行程序，也可能是串行和并行程序混合的多道程序。对于多处理器体系结构，评价是新工程方法的一个关键部分。在考察多处理器体系结构的核心或者本书中所评价的折中方案之前，理解评价的关键问题是非常重要的。

不幸的是，对于多处理器体系结构的工作负载驱动的评价工作比在单处理器情况下要更加困难，其原因有以下几点：

- 并行应用的不成熟性。对于多处理器，找到有“代表性”的工作负载并非易事，这

既因为它们的使用相对来说不成熟，又因为有很多新的行为特征要表示。

- *并行程序设计语言的不成熟性*。并行程序设计的软件模型还不稳定。基于不同的模型编写的程序会有完全不同的行为。
- *行为差异的敏感性*。不同的工作负载，甚至将相同的串行工作负载并行化时所做出的不同决策，都会呈现出差异巨大的体系结构执行特征。
- *新的自由度*。在体系结构中有一些新的自由度。最明显的是处理器的数量。其他的自由度包括扩展的存储器层次结构，特别是通信体系结构的组成结构和性能参数。与工作负载的自由度（即应用程序参数）和基本的单处理器节点一起，这些参数导致了试验设计的极其巨大的空间，特别是在通用的环境中评价一个概念或者折中方案，而不是评价某一台确定的机器时更是如此。通信的高代价使得性能对所有这些自由度之间的相互作用比在单处理器情况下更为敏感。也使得对如何遍历大的参数空间的理解更加重要。
- *模拟的限制*。用软件来模拟多处理器从而评价设计上的决策比模拟单处理器需要更多的资源。多处理器的模拟要消耗大量的内存和时间。因此，虽然我们希望探索的设计空间较大，但是，实际可能探索的空间往往要小得多，我们在决定对哪部分空间进行模拟时必须要做出细致的折中方案。

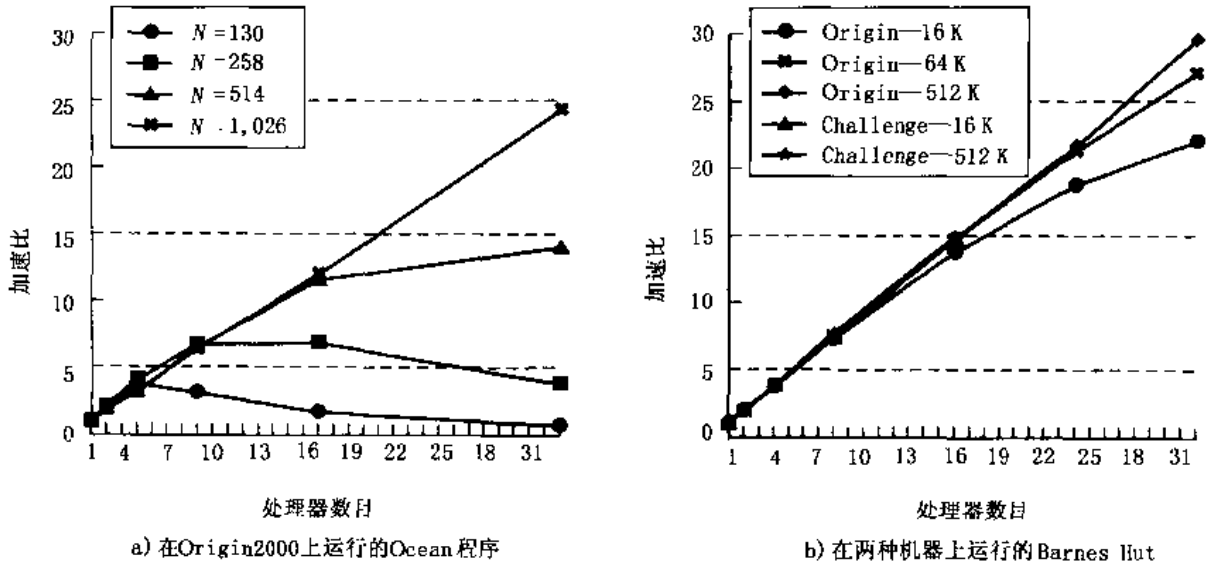
在应付这些困难时，我们对于第 2、3 章中的并行程序的理解是十分关键的。通过本章的学习，我们将了解到，要实现一个有效的模拟需要理解工作负载和体系结构两者的重要特性以及这些特性是如何相互作用的。特别是，应用程序参数和处理器的数量之间的关系决定了程序的基本特性，比如通信与计算比、负载平衡、时间和空间的局部性等。这些特性和扩展的存储器层次结构的参数之间相互作用，以与应用相关的显著方式影响着程序的性能（见图 4-1）。选择工作负载和机器的参数值（或者规模），理解它们规模变化之间的关系，是工作负载驱动的评价的一个关键方面，有深远的意义。它会影响我们为了充分覆盖行为特征所设计的试验方案，也会影响我们评价的结论。它还能够帮助我们限制试验的次数或必须考察的参数组合数。

200

本章的一个重要目标是揭示这些特性和参数之间的关键相互作用，说明它们的重要性，并指出一些重要的误区。尽管对评价而言，没有普遍适用的规则，但本章力图详细阐述一种通过模拟来评价真实机器和评估折中方案的方法。在本章最后在对几种工作负载的特征化过程中，以及在贯穿全书使用这些工作负载的例证性的评价中，我们遵循了这个方法。重要的是，我们不仅要进行良好的评价，而且还要理解评价研究的局限性，这样，我们在做出体系结构决策时才能正确地使用评价这一工具。

本章开头讨论了随着处理器数量增加，放大工作负载参数的基本问题，研究了性能指标和并行程序关键的固有行为特征的含义。在接下来的两节里，讨论了与扩展存储器层次结构的组成结构和性能参数间的相互作用，以及如何将这些相互作用结合到实际的实验设计中去，这两节考察了两种主要类型的评价。

4.2 节概述了一种评价真实机器的方法，它包括：首先理解我们可能会使用到的基准测试程序工作负载的类型和它们在这种评价中的角色——包括微基准测试程序、内核、应用程序和多道程序式的工作负载，还要理解选择它们时的期望的标准。然后，对于给定的工作负载，我们考察在评价一台给定的机器时如何选择工作负载的参数，说明重要的考虑和可能的



a) 在Origin2000上运行的Ocean程序　　b) 在两种机器上运行的Barnes Hut

图 4-1 应用程序参数对并行性能的影响。对 Ocean 而言，所显示的应用程序的参数是每一维网格点的个数（N），而在 Barnes-Hut 中，它是星体的数量。这些参数决定了要使用的数据集的大小。对于许多应用程序而言，比如 a）中的 Ocean，参数的作用是很明显的，至少是当数据集的尺寸相对于处理器的数量是足够大之前。对于最小的问题，从 4 个到 8 个甚至更多处理器的情况下，性能变得更差而不是更好。对于仅次于最小的问题，当处理器个数从 8 到 16 时，性能下降。对于最大的问题，性能大致随处理器个数线性增长，直到处理器个数达到 32。对于其他的应用，比如 b）中的 Barnes-Hut，数据集尺寸的影响要小得多

误区。这一节的最后将讨论可能用于解释和表示结果的各种指标。4.3 节把这种方法的讨论扩展到更具有挑战性的问题，即在更一般的情况下，通过模拟来评价体系结构方面的折中方案。

在理解了如何执行工作负载驱动的评价之后，我们进入 4.4 节，它提供了工作负载的相关特征，在本书所介绍的例证性评价中要用到这些特征。在附录中，还列出了一些用于并行计算的重要的、可以公开获得的工作负载集以及它们的原理。

4.1 改变工作负载和机器的规模

在考察规模性能模型和它们的含义之前，我们首先讨论一下在多处理器上的一些基本的性能测量，从而可以体会到适当的规模扩放的重要性。

4.1.1 多处理器性能的基本测量

假设我们已经选择了一个并行程序作为工作负载，并想利用它来评价一台机器。对于一台并行机，我们可以测量它的两个性能特征：绝对性能和并行性所带来的性能改进。后者的测量通常称作加速比，它已在第 1 章中定义，等于 p 个处理器上的绝对性能除以在一个单处理器上的性能。绝对性能（和代价一起）对于最终用户或者说机器的购买者来讲是最重要的。但是，它本身并不能说明性能的改善在多大程度上是来自于并行性的应用和通信体系结构的有效性，而不是来自于底层的单个处理器节点的性能。加速比可以告诉我们性能有多少是来自并行性的使用，但是值得注意的是，当计算比较慢时，通信代价显得不是那么重要，所以当单个节点的性能比较低时，很容易获得好的加速比。这两种指标都很重要，都应该被

201 ~ 202

测量。

对绝对性能的最好的测量是单位时间内完成的工作量。给定一个程序，要完成的工作量通常由程序的操作所依赖的输入配置来决定，这被称为问题规模（我们以后将会更精确地定义问题规模）。输入配置可能只是在程序最前面使用，也可能包括要到达“服务器”应用程序的一系列连续的输入的集合，比如一个处理银行事务的系统或是对来自传感器的输入做出响应的系统。假设输入配置和由此引起的工作量对于一组实验来讲是保持不变的，那么我们就可以把工作量看作是一个固定的参考点，测量执行时间并且将性能定义为相应的执行时间的倒数。

用户发现，在一些应用领域中，即使当输入配置固定不变时，使用一个显式的、领域特定的工作量的表示和使用每单位时间所完成的工作量这样显式的性能指标更为方便。例如，在一个事务处理系统中，指标可以是每分钟服务的事务的次数；在排序应用中，指标可以是每秒钟排序的关键字的数量；而在化学应用中，指标可以是每秒钟计算的结合的数量。但是，尽管可以显示地表示工作量，性能还是相对于特定的输入配置或者工作量来测量的，这些性能指标仍然都是从执行时间（和涉及到的应用事件的数量一起）的测量中得来的。给定了一个固定的已知的问题配置，这些领域特定的指标并没有显示出优于执行时间或者执行时间倒数的根本优势。实际上，我们必须要慎重，确保所使用的显式的工作量的测量从应用程序的角度上来看确实是有意义的测量，而不是我们可以用来骗人的东西。随着讨论的深入，还会进一步地讨论工作指标的理想特性，在 4.2.5 节中会考虑到关于指标的更为细节的问题。现在，让我们先集中精力评价一下由于并行性带来的绝对性能的改善，即因为使用了 p 个处理器而不是一个处理器所获得的加速比。

把执行时间作为我们的性能指标，在第 1 章中，我们看到了可以简单地在一个处理器和 p 个处理器上运行具有相同输入配置的程序，性能改善或加速比的测量是：

$$\frac{\text{时间（1 个处理器）}}{\text{时间（}p\text{ 个处理器）}}$$

以每秒钟的操作次数作为性能指标，我们可以按下式测量加速比：

$$\frac{\text{每秒钟的操作（}p\text{ 个处理器）}}{\text{每秒钟的操作（1 个处理器）}}$$

203

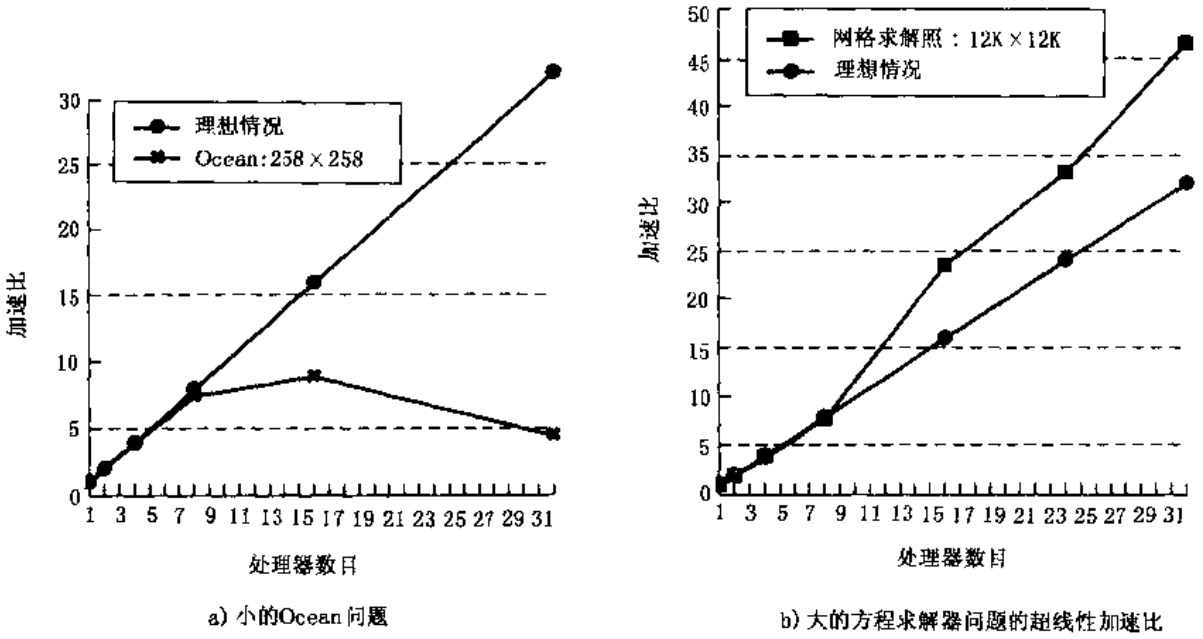
这里产生了一个问题，就是我们如何测量一个处理器的性能？例如，在一个处理器上运行最好的串行程序是否比在一个处理器上运行并行程序本身所得到的性能更为精确？这个问题其实是很容易回答的。随着处理器数量的变化，我们可以让这个问题运行在不同处理器数量的机器上，并且计算它们的加速比。那么为什么又要考虑扩放呢？

4.1.2　为什么要考虑扩放性

不幸的是，存在一些原因使得我们认为，把对固定问题规模的加速比测量作为评价各种不同规模的机器的并行性所带来的性能改善的惟一方法是不够的。

假设我们已经选定的固定问题规模相对较小，适用于只有几个处理器的机器。对于相同的问题规模，当我们增加了处理器的个数时，并行性所产生的额外开销（通信、负载失衡）相对于有效的计算就会增加。最终我们会到达某一点，问题规模对于评价现有的机器变得不

合情理的小。过高的额外开销会导致小得没有意义的加速比，其结果是由于使用了不合适的问题规模，因而不能反映出机器的能力（也就是说，问题对大型机器没有足够的并发性）。事实上，在有些点上使用较多的处理器甚至可能会损害性能，因为额外开销相对于有用的工作占据了主导的地位（参见图4-2a），用户不应该在那么大的机器上运行这类问题，所以用这样的问题来评价机器是不合适的。同样的情况，如果在大机器上运行耗时很少的问题也是不合适的。



a) 小的Ocean问题

b) 大的方程求解器问题的超线性加速比

图4-2 当处理器个数增加时，SGI Origin2000的加速比。a）显示了在Ocean应用中小问题规模的加速比。该问题规模显然非常适合于有8个处理器的机器。在稍稍超过16个处理器时，加速比会达到饱和，此时我们不清楚是否应该在这么大规模的机器上运行这样规模的问题。为评价具有32个或者更多处理器的机器而运行这样规模的问题显然是不行的！b）显示了对方程求解器内核的加速，说明了超线性加速出现的条件，即在16个处理器的系统中，处理器的工作集能够容纳于高速缓存之中，而在使用8个或者更少的处理器时，处理器的工作集就无法被高速缓存所容纳

另一方面，如果我们选择了一个适用于运行在有很多处理器的机器上的问题，我们在评价由于并行性而引起的性能改进时可能会碰到相反的问题。这个问题对于单处理器来讲可能太大了，因为它的数据已经大到无法放入一个单节点的内存中。在某些机器中，它可能不在一个单处理器上运行；在另外一些机器中，单处理器的执行会使磁盘遭受严重的反复存取的颠簸；而在其他的一些机器中，溢出的数据会被分配到扩展存储器层次结构的其他节点的内存中，导致很多人为的节点间的通信。当使用足够多的处理器时，数据就会容纳于它们的集合的内存中；如果数据分布得合理，就会消除人为的通信。在每个处理器上的计算会更加有效，结果是加速比要大大高于所使用的处理器的数量。一旦发生这种情况，当处理器的数量增加时，加速比会以更为平常的方式得到进一步的改善，但是相对于单处理器的加速仍然和处理器的数量呈超线性。

这种情况可能出现在存储器层次结构的任何层次上，而不仅仅是发生于主存。比如说，当每一个具有自己高速缓存层次结构的处理器被加入时，机器的集合高速缓存容量会增加。如果每个处理器的工作集随着数据集减少，当处理器数目增加时，处理器开始更加有效地使

204 ~ 205

用它们的高速缓存。图4-2b说明了方程求解器内核使用高速缓存容量的例子。这种因为存储系统的效果而产生的巨大的超线性加速比不是假的。确实，从用户的观点看，具有更多的、分布式的内存是并行系统胜过单处理器工作站的一个重要的优点，因为它能够使并行系统运行更大的问题，而且运行得更快。但是，超线性加速比使我们无法区别容量带来的效应和并行化所带来的通常意义上的改善，也不能帮助我们评价机器通信体系结构的有效性。

当处理器个数增加时，保持不变的问题规模的最后限制是它也许不能反映机器实际的使用情况。用户经常想使用更加强大的机器来解决更大的问题，而不是让同样的问题解得更快。在这样的情况下，因为在机器的实际使用中问题规模与机器规模一起增长，所以在评价机器时，也应该扩放问题的规模。这样的扩放可以克服刚才讨论的由规模不匹配而引出的问题，但是却失去了针对相同问题比较机器配置的简单性。

关于如何使问题规模适应机器规模的变化而改变，我们需要良好定义的规模扩放模型，这样我们才可以根据这些模型来评价机器。不管扩放模型如何，性能的度量总是单位时间内所完成的工作量。但是，如果问题规模被扩放，所做的工作并不是保持不变的，我们不能简单地通过比较执行时间来决定加速比。工作量必须被表示和测量。问题是如何去做？此外，我们还想理解扩放模型是如何影响程序特征的，比如通信与计算比率、负载平衡、扩展的存储器层次结构中的数据局部性等。为了简单起见，我们集中看一下单一的并行应用程序，而不是多道程序的工作负载。首先，我们需要清楚地定义那些曾被非正式使用的术语：扩放机器和问题的规模。

*扩放机器*的意思是使机器更加强大（或者相反）。可以通过使机器的任何一个成分，如单个处理器、高速缓存、内存、通信体系结构或者输入输出系统的能力更加强大，更加复杂或者更加快速的办法达到这个目的。一般来说，*机器规模*是刻画单个节点的处理能力、存储器层次结构、通信和输入输出能力的一个向量。机器的扩放包括改变向量中的某一个或者多个项。因为我们感兴趣的是并行性，所以我们就把机器规模定义为处理器的数量，而且我们假设对于每个节点，它的本地高速缓存、存储器系统以及每个节点的通信能力在机器扩放时保持不变。扩展一台机器意味着添加多个相同的节点。比如说，把一台有着 p 个处理器，$p \times m$MB 内存的机器扩展 k 倍，产生的是一台拥有 $k \times p$ 个处理器和 $k \times p \times m$ MB 内存的机器。

*问题的规模*指的是一个特定问题实例或者输入配置，它通常由输入参数向量而不是单个参数 n（如，一个 $n \times n$ 的 Ocean 网格或者 n 个粒子的 Barnes-Hut）指定。比如，在程序 Ocean 中，问题规模由向量 $V=(n, \varepsilon, \Delta t, T)$ 刻画，n 是网格每一维的尺寸（它说明了我们在表示海洋时的空间分辨率），ε 是用来决定多网格方程求解器收敛的容差系数，Δt 指时间分辨率（即时间步之间的物理时间），T 是所执行的时间步的数量。在一个事务处理系统中，问题规模由使用的终端个数、终端用户产生业务的速率、事务处理的混合等决定。问题规模是一个决定程序所做的工作量的主要因素。

必须把问题规模和数据集尺寸区别开来。*数据集尺寸*是在一个单处理器上运行程序所需要的存储器的数量。这本身不同于程序的*内存使用*，程序的内存使用是包括复制在内的并行程序使用的内存的总和。数据集的尺寸通常依赖于少数几个程序参数。比如，在 Ocean 中，数据集的尺寸完全是由网格规模 n 决定的，但是指令条数和执行时间则是依赖于其他的问题规模参数。因此，尽管问题规模向量 V 决定了应用程序的很多重要参数，比如它的数据

集尺寸、它执行的指令条数和它的执行时间，它还是有别于这些特征中的任何一个。

4.1.3 扩放的关键问题

[206] 给出了这些定义后，要扩展一个问题使之运行在较大的机器上，还要解决两个问题：

1）问题扩放应该在什么约束之下？为了定义一个扩放模型，当机器扩放时必须保证一些特性不变。这些特性或许包括数据集尺寸、每个处理器的内存使用、执行时间、每秒钟执行的事务处理的数量、分配给每个处理器的粒子数量或者矩阵行数。

2）问题如何扩放？也就是说，如何改变问题规模向量 V 中的参数来满足选定的约束？

为了简化讨论，我们先假设问题规模由单个参数 n 决定，然后考察在这个假设下的扩放模型及其影响。稍后，在 4.1.6 节中，我们将会考察扩放工作负载参数时这些参数之间相互参照的更为微妙的问题。

4.1.4 扩放模型和加速比的测量

用作扩放约束的基础特性可以被分成两类：*面向用户的特性*和*面向资源的特性*。面向用户的特性的例子是在 Barnes-Hut 中分给每个处理器的星体的数量；在矩阵乘法程序中为每个处理器的矩阵行数，在事务处理中为每个处理器向系统发出的事务的数量以及每个处理器执行的输入输出操作的数量。面向资源的约束的例子是执行时间和每个处理器使用的存储器总量。由于在不同的约束下执行扩放时，给定数量的处理器要完成的工作量是不同的，因此每一个这样的约束都定义了一个不同的扩放模型。究竟是面向用户还是面向资源的约束更加合适则取决于应用领域，构造基准测试程序的一项关键性任务是保证扩放约束对于所涉及的领域是有意义的。

在执行评价时，面向用户的约束通常更容易被遵循（比如，随着处理器数量的变化简单线性地改变粒子的数量）。但是，大规模的程序经常是在严格的资源约束之下运行的，资源约束在跨不同应用领域时更有普遍性（时间就是时间，存储器就是存储器，不管程序是处理粒子还是矩阵），因此我们将使用资源约束来说明扩放模型的效果。让我们针对为了在一台规模大 k 倍的机器上运行而扩展应用规模时的约束，考察一下三个最流行的面向资源的模型：*问题约束的*（PC）*扩放*，*时间约束的*（TC）*扩放*和*存储器约束的*（MC）*扩放*。

在 PC 扩放中，问题规模是固定的，也就是说，它根本没有被扩放，尽管前面也讨论过一些关于固定问题规模所带来的问题。不管机器上处理器的个数有多少，都使用相同的输入配置。在 TC 扩放中，完成程序所需要的墙钟（wall-clock）执行时间是固定的。问题被扩放，使得在大机器上新的问题的执行时间和在小机器上旧问题的执行时间相等（Gustafson 1988）。[207] 在 MC 扩放中，每个处理器使用的主存储器的数量是固定的。问题被扩放，使得新问题使用的主存正好是老问题的 k 倍（包括数据复制）。因此，如果老问题刚好能容纳于小机器的内存，那么新问题则应刚好能容纳于大机器的内存。

在某些领域里可能有一些更加适合的专门化的模型，比如，在商业在线事务处理基准测试程序中，事务处理委员会（TPC）规定了扩放的规则，即产生事务的用户终端数量和被访问的数据库的规模随被评价系统的“计算能力”成比例地扩放，其测量遵循指定的方式。这和 TC、MC 扩放模型一样，符合资源约束条件的扩放经常需要经过一些实验才能找到合适的输入，因为资源的使用并不一定是简单地随着输入参数而扩放。存储器的使用通常是可以预

见的，特别是如果没有在主存中复制的需要时更是如此，但是预见一个输入配置在 256 个处理器的机器上的执行时间和另外一个输入配置在 16 个处理器的机器上的执行时间相等却是十分困难的。我们将进一步研究 PC、TC 和 MC 的扩放模型，看看在这些模型下，“单位时间完成的工作量”，即加速比，被转换成了什么。

1. 问题约束的扩放

在 PC 扩放中的假设是，用户使用较大机器的目的是为了更快地解决相同问题，这是一种很寻常的情况。比如，如果一个视频压缩算法每秒钟只处理一帧图像，我们使用并行性的目标可能不是在 1 s 内压缩一个更大的图像，而是每秒钟压缩 30 帧图像，从而可以对这种大小的帧进行实时压缩。另外一个例子是，如果一个 VLSI 的布线工具要用一周的时间来布通一个复杂的芯片的版图，我们更感兴趣的可能是如何使用并行性来减少布线时间，而不是为一个更大的芯片布线。因为在工作/时间的性能定义中，有用的工作保持不变，加速比指标的公式可以简单地表示成：

$$\text{加速比}_{PC}\ (p\ \text{个处理器}) = \frac{1\ \text{个处理器用的时间}}{p\ \text{个处理器用的时间}} \tag{4-1}$$

2. 时间约束的扩放

这个模型假设用户有一定的时间可以用来等待程序的执行，他们想在这个固定的时间内解决尽可能大的问题。(想像一下那种愿意在计算中心购买 8 小时机时的用户或者那些愿意等待一个通宵完成程序的运行，但是需要在第二天早晨获得结果以便分析的用户)。尽管在 PC 扩放中问题规模固定不变，但执行时间是变化的，而在 TC 扩放中，问题规模是增加的，执行时间则保持不变。因为性能是工作量除以时间，而当系统扩放时时间保持不变，加速比可以由在固定的执行时间内完成的工作的增量来测定：

208

$$\text{加速比}_{TC}\ (p\ \text{个处理器}) = \frac{p\ \text{个处理器的工作}}{1\ \text{个处理器的工作}} \tag{4-2}$$

问题是如何测量工作，如果我们把它测量成在一个单处理器上问题配置的实际执行时间，那么我们可能不得不在机器的一个处理器上运行一个较大（扩展）规模的问题，这样才能获得分母[⊖]。不幸的是，这种情况可能行不通，或者要花很长的时间，或者根本不可能运行。

工作指标的理想特点是它应该容易测量，而且尽可能地与体系结构无关。它应该很容易用一个仅仅基于应用的解析表达式来建立模型，我们不应该进行额外的实验来测量问题被放大后的工作量，工作量的测量也应该与算法的串行时间复杂度成线性比例（见例 4.1）。

例 4.1　为什么线性扩放的特点对于工作指标很重要？

解答：如果我们希望理想的加速比（忽略存储器系统的人为效应）和处理器的数量成比例，那么线性扩放特点是很重要的。为了理解这一点，假设我们在一个矩阵乘法程序中把正方形矩阵的行数 n 作为工作的指标。让我们完全忽略掉存储器系统的相互作用，如果单处理器系统问题有 n_0 行，那么它的执行“时间”或者它需要执行的乘法操作的数量，和 n_0^3 成比例。因为问题是确定的，我们所能期望 p 个处理器在相同时间内的最好情况是执行 $n_0^3 \times p$

⊖ 原文是“分子”。——译者注

次操作，它对应（$n_0 \times \sqrt[3]{p}$）×（$n_0 \times \sqrt[3]{p}$）的矩阵。如果我们把矩阵行数作为工作的度量，那么根据式（4-2），即使在这种理想的情况下加速比是（$n_0 \times \sqrt[3]{p}$）/ n_0 或者说$\sqrt[3]{p}$，而不是 p。使用矩阵（n^2）中的点数作为工作的指标，从这点讲来，也是不合适的，因为它导致了时间约束的理想加速比为 $p^{2/3}$。但是，使用 n^3（乘法操作的次数）作为工作指标会得到理想的加速比 p，因为这种度量和矩阵乘法串行时间复杂度$O(n^3)$成线性比例。■

理想的工作量度量不仅要满足这些特点，而且从用户角度看应是一个直觉参数。比如，在使用一种叫基数排序的方法对整数的关键字进行排序时，串行的复杂度随待排序的关键字的个数线性增加，所以我们使用关键字作为工作的度量。但是在实际应用中是很难发现这种度量的，特别是当多个应用的参数都在扩放，并且以不同方式影响执行时间的时候更是如此。我们在实践中应如何测量工作呢？

如果无法找到一个具有理想特性的单个直觉参数，我们可以试着去发现一种容易从直觉参数推导，而且随着串行复杂度线性扩放的度量。一种流行的实现矩阵因子分解的 LINPACK 基准测试程序就是这么做的。大家都知道基准测试程序应该使用 $2n^3/3$ 次浮点操作来因子分解一个 $n \times n$ 的矩阵。其余的操作要么和它成比例，要么完全不占主导地位。如在例 4.1 中的矩阵乘法，操作的次数可以很容易从输入矩阵的维数 n 计算得到，且显然满足线性扩放的特性，所以在基准测试程序中，它被用作工作的度量。

209

真正的应用程序经常有多个参数要扩放、因此更加复杂。只要我们有一个良好定义的规则来同时扩放参数，我们就可以构造一个具有期望特性的解析形式的工作的度量。但是，这样的工作量计算可能不再是简单或者直观的，它们对评价者或者基准测试程序的提供者有很多要求。而且，在复杂的应用中解析形式的预测往往被简化（比如，它们通常是平均的情况或者它们没有反映出可能相当重要的“实现”的行为），因此所执行的指令或操作的实际增长率可能和预期的不同。

在这样的情况下，一种普遍适用的经验式的技术是运行串行程序，测量机器操作形式的工作量。如果能知道某一类型的高层次操作，比如粒子之间的相互作用，总是直接与串行复杂度成比例，那么，我们就可以计算出运行时执行的操作的数量。在更一般的情况下，我们可以尝试测量在一个单处理器上运行该问题所花费的时间，假设所有的存储器访问都是在高速缓存命中并且花相同的时间（比如，单个周期），因此消除了由存储器系统引起的人为因素。这种工作测量反映了在运行程序的时候哪些机器指令被实际执行了，同时避免了颠簸和超线性问题，我们把它称作具有完美存储器的执行时间（注意，它和在 3.4 节中介绍的串行忙-有用时间十分相近）。很多计算机拥有某种系统实用程序，允许对计算的情况进行收集，获得具有完美存储器情况的执行时间。如果没有这样的功能，我们必须求助于对某些高层操作发生次数的测量。

一旦我们有了工作量的测量，我们可以用式（4-2）计算 TC 扩放下的加速比。但是，要决定能产生期望的执行时间，因而满足 TC 扩放的输入配置，可能需要经过反复的求精

3．存储器约束的扩放

这个模型基于的假设是用户想在不考虑运行时间的情况下，运行尽可能大的程序而不使机器的内存溢出。比如，对于一个星体物理学家来讲，重要的是运行一个像 Barnes-Hut 这样的 n 个星体的模拟，该模拟包含了机器可以接受的最大数量的星体以便提高星体对宇宙采

样的分辨率。MC 扩放导致经常使用一个叫做*扩放的加速比*的性能改善指标，它被定义成在单个处理器上运行一个较大的（扩展的）问题花费的时间与该问题在扩展的机器上运行时间的比率。对于卖方来讲，这个指标通常是有诱惑力的，因为这样的加速比一般会很高。实际上，它是对一个非常大的问题测量问题约束的加速比，往往具有低的通信与计算比和充分的并发性，并且受益于由存储器和高速缓存容量产生的超线性效应。不管怎样，扩展了的问题并不是我们在 MC 扩放模型下运行在单处理器上的问题，所以这不是一个合适的加速比指标。

210

与前面的模型不同，在 MC 扩放模型下，工作和执行时间都不固定。作为惯例，使用工作除以时间作为性能指标，我们可以把加速比定义为：

$$\text{加速比}_{MC}\ (p\ \text{个处理器}) = \frac{p\ \text{个处理器的工作}}{p\ \text{个处理器的执行时间}} \times \frac{1\ \text{个处理器的执行时间}}{1\ \text{个处理器的工作}} = \frac{\text{工作的增量}}{\text{执行时间的增量}} \tag{4-3}$$

如果执行时间的增加仅仅是因为工作量增加，而不是因为并行性的额外开销所致，即如果不存在 MC 扩放中一般很少发生的存储器系统人为效应，那么加速比将是 P，这正是我们所希望得到的。工作量的测量和我们在前面讨论过的 *TC* 扩放的测量方法一样。

因为在 MC 扩放下数据集的尺寸比在其他模型下增长的快，所以并行额外开销的增长相对缓慢，加速比通常更好（忽略容量型人为效应）。MC 扩放确实是很多用户所希望的使用并行机的方式。但是，对于很多类型的应用来说，MC 扩放会导致一个严重的问题：执行时间（对于并行执行）会长得令人无法容忍。这个问题能出现在任何一个工作随问题规模的增长比存储器使用的增长快得多的应用之中（见例 4.2）。

例 4.2　矩阵分解是串行工作的增长比存储器使用的增长要快得多的简单例子。请说明在这种应用中，MC 扩放如何导致并行执行时间迅速增长。

解答：在矩阵分解中，对于一个 $n \times n$ 的矩阵，虽然数据集的尺寸和存储器的使用按 $O(n^2)$增长，在一个单处理器上的执行时间却以 $O(n^3)$ 增长。假设一个 10 000 × 10 000 的矩阵要使用大约 800 MB 的存储器，可以在一个单处理器的机器上用一小时完成矩阵的因子分解，那么考虑一个包含 1 000 个处理器的扩展的机器，在这台机器上，在 MC 扩放模型下，我们可以因子分解一个 320 000 × 320 000 的矩阵，因为几乎不需要在主存中进行复制。但是，并行程序的执行时间会增加到将近 32 个小时（即使假设有完美的 1 000 倍的加速比）。■

在 3 个模型中，时间约束的扩放越来越被认为是最可行的一个，但是，没有一个模型可以宣称自己是最适于所有应用和所有用户的。不同的用户有不同的目标，在不同的约束下工作，不可能在任何情况下都非常严格地遵循一个给定的模型。但不管怎样，对于在机器扩放条件下分析扩放的性能而言，这 3 个模型都是有用的综合的工具。

4.1.5　扩放模型对方程求解器内核的影响

我们先考察一个简单的例子——第 2 章的方程求解器内核，看一看它如何和不同的扩放模型相互作用，以及这些模型是如何影响它与体系结构相关的行为特征的。对于一个 $n \times n$ 的网格来说，简单方程求解器需要的内存为 $O(n^2)$。计算复杂度是 $O(n^2)$ 乘以最终达到收敛的迭代次数，我们可以保守地假设迭代次数是 $O(n)$（使数值从网格的一侧边界流动到另

211

外一侧所需要的迭代次数）。这导致了$O(n^3)$的串行计算的复杂度。

假设在任何情况下，由于并行性的加速比等于处理器的个数 p，考虑一下在 3 种扩放模型下的执行时间和存储器需求。对于 PC 扩放，因为相同的 $n \times n$ 网格在更多的处理器 p 之间分割，每个处理器的存储器需求会按 p 线性下降，执行时间也是一样。在 TC 扩放中，执行时间根据定义是固定不变的。假设是线性加速比，这就意味着如果扩展的网格规模是 $k \times k$，那么 $k^3/p = n^3$，所以 $k = n \times \sqrt[3]{p}$。因此每个处理器需要的内存为

$$\frac{k^2}{p} = \frac{n^2}{\sqrt[3]{p}}$$

它按照处理器的立方根减少。根据定义，使用 MC 扩放，每个处理器的存储器需求保持不变，为$O(n^2)$，这里单个处理器执行的基本网格是 $n \times n$。这意味着网格的整个规模增长了 p 倍，所以扩展的网格成为 $n\sqrt{p} \times n\sqrt{p}$而不是 $n \times n$。因为它现在需要 $n\sqrt{p}$次迭代才能收敛，所以串行的时间复杂度为 $O((n\sqrt{p})^3)$。这意味着即使假设由于并行化产生的加速比是理想的，在 p 个处理器上扩展了的问题的执行时间是

$$O\left(\frac{(n\sqrt{p})^3}{p}\right)$$

或者说是 $n^3\sqrt{p}$。因此，基本问题的并行执行时间是串行执行时间的$\sqrt{p}$倍。甚至在线性加速比的假设下，在一个处理器上需要运行一个小时的问题，在 *MC* 扩放模型下在一台具有 1 024个处理器的机器上要运行 32 个小时。那么，对于这个简单的方程求解器，MC 扩放下的执行时间会快速增加，在 TC 扩放下每个处理器对内存的需求减少。

让我们考虑一下不同的扩放模型对于并发性、通信与计算的比率、同步和 I/O 频度、时间和空间的局部性、消息的尺寸（在使用消息传递模型时）的效应。

这个内核中的并发性和网格点的数量成正比。它在 PC 扩放下保持不变；在 MC 扩放下的增长和 p 成正比；在 *TC* 扩放下的增长和$p^{0.67}$成正比。

通信与计算的比率是分配给每个处理器的网格分区的周长与面积之比，也就是说，它和每个处理器的点数（n^2/p）的平方根成反比。在 PC 扩放模型下，该比率按$\sqrt{p}$增长；在 MC 扩放模型下，分区大小不变，所以通信与计算的比率也不变。最后，在 TC 扩放模型下，因为一个处理器的分区的规模随着处理器数量的立方根减少，则该比率按 p 的六次方根增加。

方程求解器在每个网格扫描的结尾处同步一次来决定收敛与否。假设它在那个时刻也执行 I/O 操作，比如，在每次扫描的结尾输出最大误差。在 PC 扩放模型下，在一次给定的扫描中每个处理器完成的工作随着处理器数量的增加而线性递减，所以假设线性加速比，同步频率和输入输出频率随着 p 而线性增加。在 MC 扩放模型下，频率保持不变；在 TC 扩放模型下，它随着 p 的立方根增加。

[212]

这个方程求解器的重要工作集的尺寸正好是网格在一个处理器中的分区的大小，表示了它的时间局部性。因此，在 PC 扩放下，重要工作集的尺寸和对高速缓存的需求随着 p 而线性减少，在 MC 扩放中保持不变，在 TC 扩放中随着 p 的立方根减少。因此，在 TC 扩放中，尽管总的问题规模在增加，每个处理器的工作集的尺寸还是在减小。

方程求解器的空间局部性在一个处理器分区内部，在面向行的边界上是最好的，在面向

列的边界上是最差的。因此，它随着处理器的分区变小，随着面向列的边界相对于分区面积变大而减小，因此在 MC 扩放下它保持不变，在 PC 扩放下减少得很快，在 TC 扩放下减少得比较慢。

最后，在消息传递模型中，一个单个的消息很可能包含一个处理器分区的边界行或边界列，它是分区尺寸的平方根。因此，这里消息尺寸的扩放类似于通信与计算的比率的扩放。然而，一个进程发送的消息数量只是依赖于相邻的进程数目，而与 n、p 或者扩放模型无关。

从前面的讨论中我们可以很清楚地看到，只要存储器或者高速缓存的容量效应不是主要的，在 MC 扩放下，我们可以期望有最低的并行性额外开销和最高的加速比，在 TC 扩放下其次。在 PC 扩放下，我们应该能够预料到，至少一旦额外开销相对于有用的工作变得显著起来时，加速比会很快下降。还有一点很清楚，应用参数的选择和扩放模型在很大程度上影响基本的程序特性以及和扩展存储器层次结构在体系结构上的相互作用，比如空间和时间的局部性。除非已经知道一个特定的扩放模型刚好适合一个应用程序或者特别不适合，以所有这 3 种模型评价机器都是有用处的。在讨论实际的评价时，我们会更仔细地考察与体系结构参数的相互作用以及它们对评价的重要性（4.2 节和 4.3 节）。首先，我们先简要地看一看扩放的其他那些重要而精妙的方面：如何扩放应用的参数来满足给定的扩放模型的约束。

4.1.6　扩放工作负载参数

在讨论问题规模扩放的约束时，我们仅对单个应用参数 n 作了简化性的假设，但没有考察构成问题的规模向量的不同应用参数应该如何相互参照进行扩放来满足选定的约束条件。我们现在不考虑这个简单假设，比如，Ocean 应用程序有一个带 4 个参数的向量：n，ε，Δt，T。工作负载应该如何相互参照而扩放，这在 PC 扩放中不是问题；但是，它在 MC、TC 扩放中的确是一个问题。不同的参数彼此相关，不太可能扩放其中一个参数而不影响其余的参数，或者说不可能独立地扩放它们。比如，在 Barnes-Hut 应用的实际使用中，参数 θ（力的计算精度），Δt（时间步之间的物理时间间隔）应该随 n（星体的数量）的变化而扩放。所有这些参数决定了一个给定的执行特性，比如，Barnes-Hut 的执行时间的增长并不是简单的 $n\log n$，而是

213

$$\frac{1}{\theta^2\Delta t}n\log n$$

结果，在 TC 扩放模型下，星体数目 n 的增加没有仅仅由扩展 n 获得的那么大。

即使简单的方程求解器内核也有另外一个参数，ε，它是用来决定求解器收敛的容差系数。使该容差系数变小，正如实际应用中随着 n 的扩放所做的那样，就要增加为达到收敛所需要的迭代次数，因此，也就提高了执行时间；但是，它并没有影响对存储器的需求。和仅仅扩放 n 相比较，ε 和 n 一起扩展导致了在 TC 扩放下每个进程的网格尺寸、存储器需求、工作集尺寸减少得更快，通信与计算的比率在 TC 扩放中增加得更快，而在 MC 扩放中保持不变，甚至在 MC 扩放中执行时间增加的更快。作为一个使用工作负载的机器设计者，从应用程序用户的观点来理解参数之间的关系，并根据这种理解在评价中对参数进行扩放是十分重要的。否则，我们很容易获得一个不正确的关于体系结构的结论。

参数之间的实际关系和扩放它们的规则依赖于应用的领域。因为没有一个普遍适用的规则，这使得好的评价变得更加令人感兴趣。比如，在像 Barnes-Hut 和 Ocean 这样的通过使物理现象离散化而进行模拟的应用中，不同的应用参数通常精确地决定着模拟某些现象（比如星系的演化）过程中不同的误差来源。因此，整体扩放这些参数的合理规则是由扩放不同类型误差的策略所决定的。理想的情况是，基准测试程序集会描述扩放规则，甚至可以在应用中对它们编码，只留下一个像 n 这样的自由参数；所以对于作为基准测试程序使用者的设计者来说，他不必再操心学习这些东西。习题 4.12 和 4.13 说明了合适的应用程序扩放的重要性，它们显示了适当地扩放多个参数与仅仅扩放数据集尺寸参数 n 相比，往往会得出关于体系结构的定量的、某些时候甚至是定性的不同结论。

214

4.2 评价一台实际的机器

现在，我们已经理解了合适扩放的重要性以及问题和机器的规模对于基本行为特性和体系结构相互作用的效果，我们可以为两种主要类型的工作负载驱动的评价制定特定的指导方针，这两类评价是：评价一台实际的机器和在通用的环境中评价一个体系结构的概念或折中方案。评价一台实际的机器在很多方面是比较简单的：组成结构、粒度和机器的性能参数是固定的，我们所要考虑的是选择合适的工作负载和工作负载参数，而且我们并没有受到软件模拟的局限性的约束。本节提供了评价实际机器的一个范例性模板。我们从使用微基准测试程序分离性能特征入手，然后考虑为评价而选择工作负载所涉及的主要问题。紧接着介绍一旦选定了工作负载后评价机器的指导方针，首先是何时固定处理器的数量，然后是何时允许它变化。本节最后将讨论用于测量机器性能和表示评价结果的流行的指标。在评价一台实际的机器时的所有问题都和评价体系结构的概念或折中方案有关。

4.2.1 使用微基准测试程序分离性能

评价一台实际的机器的第一步是理解它的基本性能指标，也就是、程序设计模型提供的主要操作的性能特性、通信抽象（用户/系统接口）或者硬件/软件接口。这通常是通过使用小的、专门编写的被称作*微基准测试程序*（Saavedra，Caines，and Carlton 1993）的程序来完成的，微基准测试程序是为了分离这些性能特征（比如，延迟、带宽、额外开销等）而设计的。

在并行系统中使用了 5 种类型的微基准测试程序；前三个也可以用于单处理器的评价：

处理类微基准测试程序。测量处理器不涉及存储器访问的操作性能，比如算术操作、逻辑操作和转移。

本地存储器微基准测试程序。决定了本地节点内部的存储器层次结构各层的组成、时延、带宽，测量由不同层次满足的本地读和写操作的性能，包括那些导致 TLB 扑空和缺页的操作。

215

输入－输出微基准测试程序。测量 I/O 操作的特征，比如不同跨距和长度的磁盘读写。

通信微基准测试程序。测量数据通信操作，比如消息发送和接受或者不同类型的远程读写。

同步微基准测试程序。测量不同类型的同步操作的性能，比如加锁和栅障。

通信和同步微基准测试程序依赖于所使用的通信抽象或者程序设计模型。它们可能涉及

一个或者一对处理器，比如，一个单个的远程读扑空、一对发送/接收或者一个自由锁的获得；它们也可以是集合式的，比如广播、归约、全部到全部的通信、概率性通信模式、很多处理器竞争一个锁或者路障。可以设计不同的微基准测试程序来强调非竞争时延、带宽、额外开销和竞争。

出于测量的目的，微基准测试程序通常是采用重复的基本操作集合来实现的（比如，一行中有10 000个远程读操作）。它们通常具有一些简单的参数，这些参数可以变化以获得更加完整的特征；比如，集合通信微基准测试程序中涉及的处理器的数量，或者本地存储器微基准测试程序中连续读操作之间的跨距。图4-3说明了使用本地存储器微基准测试程序所获得的机器的典型情况。微基准测试程序的主要作用是分离和理解基本系统能力的性能。一个现在尚未实现的更具雄心的想法是，如果工作负载可以用不同的基本操作的加权之和来刻画，那么对于给定的负载一台的机器的性能可以根据它执行相应的微基准测试程序的性能做出预测。当我们在后续的章节中测量真实的系统时，我们将讨论一些特殊的微基准测试程序和它们的设计问题。

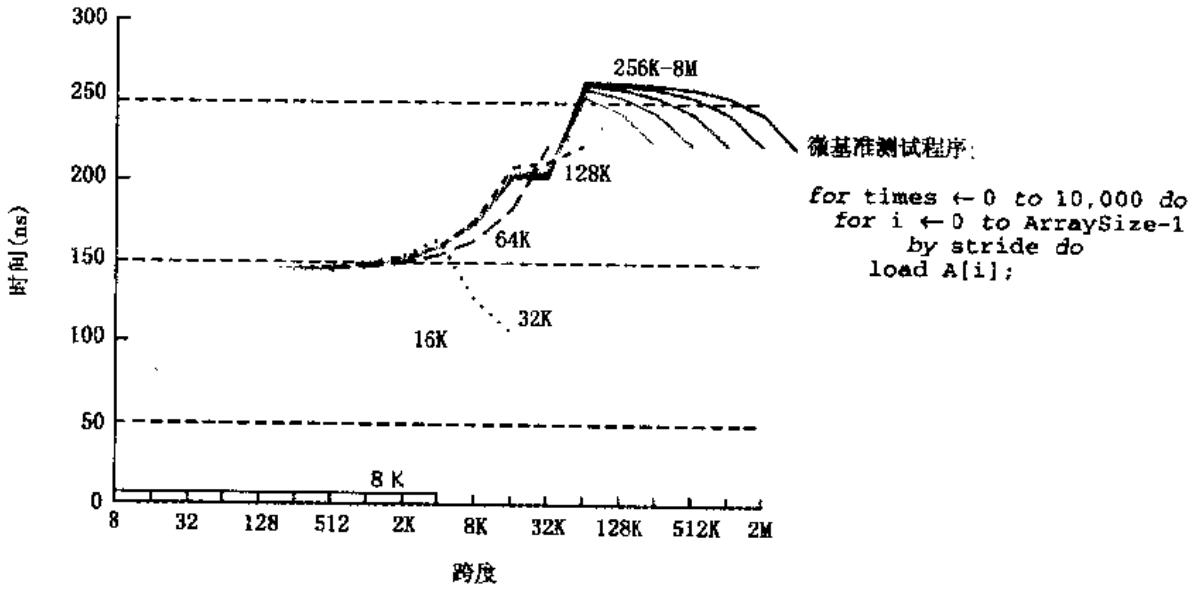


图4-3 在CRAY T3D多处理器的单个处理节点上微基准测试程序的实验结果。每个处理器有一个由本地存储器支持的小的单级高速缓存。微基准测试程序由读取本地数组的大量操作构成，y轴显示了每次读操作需要的时间，以ns为单位。x轴是在一次循环中连续读操作之间的跨距（即所访问的存储器单元的地址的差异）。不同的曲线对应所访问的数组的尺寸（ArraySize），并标注了数组的尺寸。当ArraySize小于8 KB时，数组可以容纳于处理器的高速缓存，这样所有的读操作都能命中，需要6.67 ns完成读操作。对于大一些的数组，我们能观察到高速缓存扑空的效应。平均访问时间是命中时间和扑空时间的加权和，直到跨距大于缓存块的大小（32个字或128个字节）而造成地址出界，使得每一次访问都扑空为止。曲线下一个上升是由于某些访问引起缺页而发生的，地址出界的原因是跨距足够大（16 KB），以至于每一次连续的访问都导致缺页。曲线最后一个上升是由于4体的主存中存储体的访问发生了冲突所致，当跨距为64 KB时，连续的访问总是落在相同的存储体上，而其余的存储体总是空闲的

分离了性能特征之后，下一步骤是使用更加实际的工作负载来评价机器。我们必须沿着三个主要的坐标：工作负载、它们的问题规模、处理器的数目（或者机器规模）进行遍历。

低级机器参数是固定的，我们首先来为评价选择工作负载。

4.2.2 选择工作负载

除了微基准测试程序以外，用于评价的工作负载可以按照真实性和复杂度上升的次序划分成三类：内核、完整的应用程序和多道程序的工作负载。每一类都有它自己的作用、优点和缺点。

内核是实际的应用程序中良好定义的部分，但是它们本身却不是完整的应用程序。其范围从简单内核（比如，一次矩阵转置或者一次相邻网格的扫描）到更为复杂的、占应用程序执行时间的主要部分的那些基本内核（比如矩阵因子分解和解偏微分方程用的迭代方法）。用于信息处理的内核的例子包括在决策支持应用中使用的复杂的数据库查询或者一组数据的排序。内核揭示了微基准测试程序中所没有的高层次的相互作用，结果，在某种程度上也失去了性能分离的作用。它们的关键特性是：它们那些与性能相关的特征，比如通信与计算比率、并行性和工作集，很容易理解并经常可以解析地决定，所以作为相互作用的结果观察到的性能可以根据这些特征来解释。

完整的应用程序由多个内核组成，并且展示了那些一个内核所不能展示的内核之间的高层次交互。完整的应用程序不同于内核，它们由用户运行以获得用户关心的答案。相同的大数据结构可以在一个应用程序中被多个内核以不同方式访问，而被不同的内核访问的不同的数据结构可能在存储器层次结构中产生相互的干涉。此外，对于孤立的内核最佳的数据结构在完整的应用程序中可能不是最好的。对于划分技术也是如此。比如，如果在一个应用程序中有两个独立内核，那么我们可以决定不把每个内核划分给所有的进程，而宁愿在它们之间共享进程。共享一个数据结构的不同内核可以有几种划分的方法，以求在它们不同的访问类型和通信模式之间得到某种平衡，产生最大的总体局部性。一个应用程序中多内核的存在带来了很多微妙的相互作用，一个完整的应用与性能相关的特性通常不能用解析的方法精确地决定。

216 ~ 217

多道程序工作负载是由多个在机器上一起运行的串行应用程序和并行应用程序构成的，不同的应用程序可以在时间上或者在空间上共享机器（比如，运行在机器处理器的不相交子集上的不同应用程序）或者同时具有时间共享和空间共享，它依赖于操作系统多道程序的规定。正如完整的应用程序因作为其组成成分的内核之间的高层次相互作用而变得复杂，多道程序的工作负载也涉及到多个完整应用程序本身之间的复杂相互作用。

随着我们从内核转移到完整的应用程序和多道程序工作负载，我们获得了真实性，这是非常重要的。很多致命的错误和性能问题不能被微基准测试程序甚至内核所揭示，但是却能被这些工作负载所发现。但是我们也失去了某些能力，包括：简明地描述工作负载的能力，明确地说明和解释结果的能力，孤立各种性能因素的能力。在极端情况下，多道程序工作负载不仅难以解释，而且设计都非常困难：比如哪些应用程序应该被包括在这种工作负载中？比例是什么？因为和操作系统之间有着精细的与定时相关的交互，从多道程序工作负载获得可重复的结果也是很困难的。每种类型的工作负载都有它们的作用，但是只能由完整的应用程序和多道程序工作负载反映出较高层次的相互作用这一事实（以及它们是由用户实际运行在机器上的工作负载的事实），这使得我们使用它们最终决定机器的整体性能变得十分重要。

让我们考察一下为评价而选择这样的工作负载（应用程序、多道程序负载、甚至是复杂

的内核）时期望的特性。这些特性包括应用领域的代表性、行为特性的覆盖性和足够的并发性。

1. 应用领域的代表性

如果我们作为要购买机器的用户正在进行评价，而且我们知道机器会仅仅被用来运行某些类型的应用程序的话，那么，选择一个有代表性的工作负载就是一件容易的事情。另一方面，如果我们用机器运行范围宽广的工作负载，或者如果我们是试图通过评价一台通用的机器从而为下一代机器的设计得到一些启示的设计者，我们应该选择能代表宽广领域的工作负载的混合体。

今天，并行计算的某些重要领域包括：模拟物理现象的科学应用；像计算机辅助设计、数字信号处理、汽车碰撞模拟、甚至是用于评价体系结构折中方案的模拟这样的工程应用；渲染场景或实体使之成为图像的图形化和可视化应用；像图像、视频和音频的分析和处理、语音和手写体文字识别这样的媒体处理应用；像数据库，数据挖掘和事务处理这样的信息管理应用；像航空公司员工调度和交通控制这样的优化应用；像专家系统和机器人这样的人工智能应用；多道程序工作负载；以及本身就是一个复杂的并行应用的多处理器操作系统。 218

2. 行为特性的覆盖性

工作负载可以在第3章所讨论的与性能相关的特征的整个范围之内发生实质性的变化。其结果是，评价存在的一个主要问题是，它很容易利用工作负载进行欺骗或被工作负载所误导。比如，一项研究可以选择那些强调体现了体系结构优点（比如，通信时延）的特性的工作负载，避免使用那些性能比较差的工作负载（比如，本地访问、竞争或者通信带宽）。对于通用的评价，重要的是使我们选择的工作负载在整体上能强调重要的性能特征的范围。比如，我们应该选择能覆盖低的和高的通信与计算比、小的和大的工作集、规则和不规则的访问模式和本地的、远距离的或者集合式通信的工作负载。如果我们对评价特殊的体系结构特征（比如，处理器之间的全部对全部通信的总带宽）特别感兴趣，那么我们至少应该选择一些强调这些特征的工作负载。

另外一个重要的问题是程序优化的级别。真实的并行程序并不总是像第3章讨论的那样为获得好的性能而进行了高度优化，第3章所讨论的优化不仅仅是针对现有的特定机器，而是甚至采用了降低通信与计算比率或者提高时间和空间局部性这样更加一般性的方法。没有高度优化的原因或许是因为优化程序所涉及的工作量比用户愿意付出的多，或许是因为程序是在自动并行工具的帮助下生成的。优化的级别很大程度影响着关键执行特征和突出体系结构能力的程度。我们应该特别注意4种重要类型的优化：

算法。任务的分解和分配可能不够最优（比如，比较一次网格计算是采用面向条的分配还是面向块的分配（见2.3.3节））；对数据局部性的算法上的增强（比如，阻塞）可能并没有被实现。

数据结构。使用的数据结构可能不是最优地与体系结构相互作用，增加了人为的通信，比如，比较在一个共享地址空间中是用二维数组还是四维数组来表示一个二维的网格（见3.3.1节）。 219

数据的规划、分布和对齐。即使使用了合适的数据结构，它们也可能没有被适当地分布或者没有合适地对齐页或者高速缓存的块，这会导致过高的本地流量或者人为通信。

通信和同步的协调。所产生的通信和同步结构可能不够优化，比如，在消息传递型系统

中发送小消息而不是大消息。

因为优化经常是自组织的，这些分类强制了某些结构。在合适的地方，我们应该将机器或特性的健壮性与具有不同级别优化的工作负载相比较。

3. 并发性

在工作负载中主要的性能瓶颈可能是计算负载的不平衡性，究其原因或者是分割方法所固有的特性、或者是协调同步的方法（比如，使用路障而不是点对点的同步）所致。如果这是事实，那么该工作负载可能对于评价该机器的通信体系结构不合适，因为体系结构对于这种瓶颈是无能为力的。甚至通信性能的巨大改进也不可能对整体性能有太大影响。为了评价通信体系结构，我们应该保证我们的工作负载和它们的问题规模呈现足够的并发性和负荷平衡。这里一个有用的概念是算法加速比，该加速比是假设所有的存储器访问和通信操作不花费时间（见第3章中PRAM体系结构模型的讨论）。通过完全忽略数据访问和通信的性能影响，算法加速比测量了工作负载的计算负载平衡性以及并行程序额外要做的工作。

一般来说，我们应该分离由于工作负载特征所引起的性能限制，对这些限制机器本身是无能为力的。另一件重要的事是，工作负载应该运行足够长的时间，使得它对于所评价的机器规模而言是真实的负载，尽管这一点和并发性通常更应该是输入问题规模的函数，而不是工作负载本身的函数。

连同前面讨论过的那些标准，人们已经为定义并行应用的标准基准测试程序集为方便工作负荷驱动的体系结构评价付出了巨大的努力。基准测试程序集覆盖了不同的应用领域具有不同的原则，其中一些在附录中将予以描述。尽管用于本书中例证性评价的工作负荷集是非常有限的，它们却是根据前述标准而选择的。现在，让我们假设一个特定的并行程序已经被选作工作负载，我们要看一看如何利用它来评价一台真实的机器。首先，我们使处理器的数量固定，这既简化了讨论，也能更加清楚的显现出关键的相互作用。然后，再改变处理器的数量。

220

4.2.3 评价一台固定规模的机器

固定了工作负载和机器的规模，我们只需要选择工作负载的参数。我们已经看到，对于固定数量的处理器改变问题规模会显著地影响所有重要的执行特征，从而影响评价的结果。实际上，它甚至会改变主要瓶颈的性质，即主要瓶颈是通信、负载失衡或本地数据访问。这告诉了我们非常重要但是经常被忽视的一点：在评价中仅仅使用惟一的问题规模往往是不够的，即使当处理器的数量固定时也是如此。

我们可以利用对于应用程序和体系结构的相互作用的理解来选择待研究问题的规模。我们的目标是对于真实固有行为以及体系结构方面的相互作用有足够的覆盖性，而同时又限制所需要的不同问题规模的数量。我们用一系列有计划的步骤来实现这一目标并显示在这一过程中只选择单一规模是错误的做法。我们的讨论会每次前进一步，在每一个步骤中，我们使用简单的方程求解器内核来定量地说明问题。为了得到定量的说明，我们假设正在评价一台有64个单处理器节点，每个节点有1 MB高速缓存和64 MB的主存，具有高速缓存一致性的共享地址空间的机器其步骤如下：

第1步：确定问题规模的范围

适用于某些幸运情况的一种选择问题规模的方法是要求更强的能力。我们进行的研究的

高层次目标有助于我们选择问题的规模。比如，我们可能知道机器的用户只对一些特定的问题规模感兴趣，这就简化了我们的任务。但是这种情况不常见，而且不是一个通用的方法。它不适用于方程求解器的内核。

对实际使用的知识可以帮助我们确定一个范围，低于此范围的问题对机器来说规模小得不实际，高于此范围的问题对于机器来说执行时间又太长或者用户不感兴趣。这对方程求解器的内核也不是特别有用，一旦我们确定了范围，就可以开始下一步了。

第 2 步：使用固有的行为特征

固有行为特征（比如通信与计算比和负载平衡）能帮助我们进一步限制范围并在已经选定的范围中选择问题规模。因为固有的通信与计算比常常随数据集规模的上升而降低，所以大问题可能没能足够地强调通信体系结构，至少对于固有的通信是如此；而小问题对通信的强调可能不具代表性，而且可能掩盖其他的瓶颈。因为并发性通常随着数据集规模的增加而增加，我们可以至少选择一些足够大但又不至于大到使得固有的通信变的太小的问题规模来达到负载平衡（见例 4.3）。问题规模也可以影响到应用在不同阶段所花费的执行时间，不同的阶段可能会有完全不同的负载平衡、同步和通信的特征。比如，在 Barnes-Hut 应用的实例研究中，较小的问题在树形成阶段花费了更多的时间，它不能很好地并行，与在实际中通常占主导地位的引力计算的阶段相比缺少一些期望的特性。因此我们应该小心不要选择没有代表性的场景。 221

例 4.3　你如何利用固有行为特性为方程求解器的内核选择问题规模的范围？

解答：对于这种内核，足够的工作量和负载平衡可能要求我们至少有 32×32 个点的分区，对于一台 64 个（8×8）处理器的机器，这意味着整个网格的规模至少是 256×256。这个网格规模对于 32×32 或者 1 K 个点的计算，需要在每次迭代中每个进程有 4×32 或者 128 个网格点的通信。如每个网格点有 5 次浮点运算和 8 个字节，固有的通信与计算比是每 5 次浮点运算 1 个字节。假设一个处理器能为该计算提供 200 MFLOPS（每秒 2 亿次浮点运算），这意味着需要 40MBps 的带宽。这对现代的多处理器网络来讲是相当小的，即使它是突发形式的通信也不算什么。我们假设低于 5 MBps 的通信对我们的系统来说很小，仅仅从固有特性的观点出发，不需要在每个处理器上运行大于 256×256 点（64 KB × 8 或者 512 KB 的数据）的问题，或者说不需要整体上大于 2 K × 2 K 的网格。■

像负载平衡和通信这样的固有特征随着问题规模而平滑地变化，所以要单独处理它们。可以选择少数几个跨越感兴趣范围的问题规模，如果它们的变化速率非常低，可能不需要选择很多种规模。实验显示，在大多数情况下 3 个左右是刚好的。比如，对于方程求解器的内核，我们可以选择 256×256，1 K × 1 K 和 2 K × 2 K 的网格。

另一方面，和体系结构发生相互作用的时间和空间局部性展示了它们在性能效果上的阈值，包括当问题规模变化时人为通信的产生。我们可能需要扩展对于问题规模的选择来获得对这些阈值的足够的覆盖。同时，阈值性质能帮助我们剪裁参数空间。选择问题规模的下一步骤是考察时间局部性和工作集。

第 3 步：使用时间局部性和工作集

工作集是否能被本地高速缓存或者复制存储器所容纳会显著地影响执行特征，比如本地存储器流量和人为的通信，即使在固有的通信、计算负载平衡变化不大时也是如此。在像 Raytrace（光线跟踪）这样的应用中，重要工作集都很大，而且主要是由分配到远程节点的 222

数据构成，所以由于复制容量有限所引起的人为通信可能在固有通信中占主导地位。这种人为通信随着问题规模的提高在增加，而不是减少。在其他应用中（比如 Ocean），不能容纳于高速缓存的工作集会显著地产生更多的本地存储器流量而不是人为的通信。如果在实际中这样的问题规模是现实的话，我们应该包括代表了重要工作集阈值的两侧（即能够或不能够被高速缓存所容纳）的问题规模。实际上，只要对应用程序来说是现实的，我们也应该包括对于机器而言非常大的问题规模，比如，一个几乎填满整个内存的问题规模，尽管从负荷平衡和固有通信的观点来看它可能是没有多大意义的。大的问题经常导致体系结构和操作系统之间的相互作用，比如 TLB 扑空、缺页以及由于缓存容量型扑空而引起的巨大流量，而较小的问题则不会产生这样的效果。例 4.4 和例 4.5 有助于说明如何基于工作集选择问题的规模。

例 4.4 假设问题规模和处理器的数量固定，对于我们的机器节点中最低层次的高速缓存（即距离处理器最远、距离存储器最近的高速缓存），一个应用具有如图 4-4a 所示的扑空率和高速缓存尺寸对照的曲线。如果 C 是高速缓存的尺寸，这个曲线如何影响我们对用于评价机器的问题规模的选择？

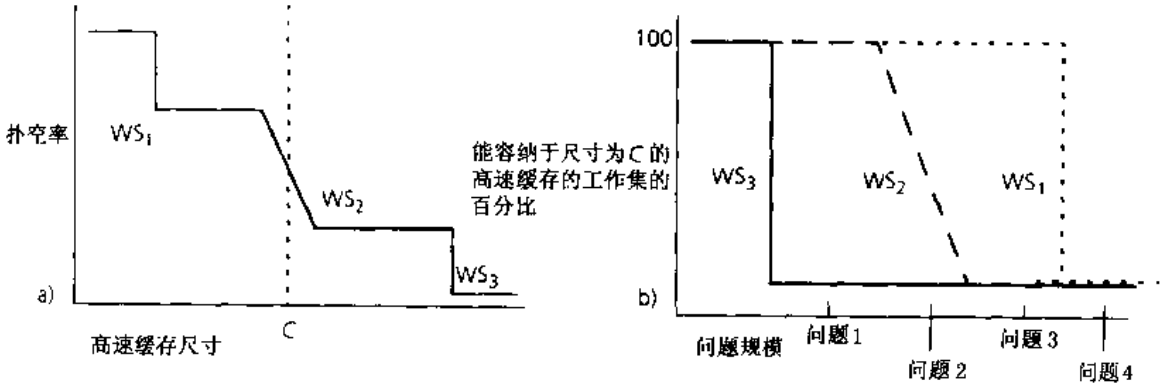


图 4-4 基于能容纳于高速缓存的工作集来选择问题的规模。在 a）中的图显示了对于固定的问题规模和所选定的处理器数量，扑空率和高速缓存尺寸对照的曲线。C 是所考虑的高速缓存或复制存储器的尺寸。这条曲线显示了三个拐点或者说工作集，两个定义得非常陡峭，另一个要缓和些。b）中的图表明，对于每一个工作集一条曲线描述了当问题规模提高时，它是否能容纳于尺寸为 C 的高速缓存。曲线的拐点表示了工作集合不再能被容纳的问题规模。可以看出规模为问题 1 的问题适合于 WS_1 和 WS_2 但是不适合于 WS_3。问题 2 适合于 WS_1，部分适合于 WS_2，但不适合于 WS_3。问题 3 只适合于 WS_1。问题 4 不适合于高速缓存中的任何工作集

解答： 从图 4-4a 中我们可以看到对于所给出的问题规模（和处理器的数量），第一个工作集能容纳于尺寸为 C 的高速缓存，第二个工作集只能被部分容纳，第三个则不能被容纳。每一个工作集按照它们自己的方式随问题规模而扩放，这种扩放决定了在何种问题规模下，该工作集不再能够被尺寸为 C 的高速缓存所容纳，因此决定了应该选择什么样的问题规模才能覆盖各种有代表性的情况。实际上，如果曲线真的包含陡峭的拐点，那么可以画一种不同类型的曲线，此时每个重要工作集对应一条曲线。图 4-4b 中的曲线描述了当问题的规模变化时，工作集是否能被尺寸为 C 的高速缓存所容纳。如果出现在曲线拐点处的问题规模处于我们所决定的实际的问题规模范围之内，那么应该保证把这个拐点两侧的问题规模包括在我们的选择之内。不这么做的结果是，可能会失去与存储器或者通信体系结构相关的重要效应。在这个例子中，拐点两侧的曲线是平滑的。这一事实说明，如果对高速缓存我们关心的只是扑空率，那么为了这个目的，只需要在每个拐点的每一侧选择一种问题规模，剪裁掉

其他的部分[○]。■

例 4.5　对于方程求解器，工作集如何影响我们对问题规模的选择？

解答：方程求解器最重要的工作集出现于当一个分区的两个子行能容纳于高速缓存以及当一个处理器的整个分区能放进高速缓存的时候。在这个简单内核中非常明确地定义了这两种情况。即使对于基于固有通信与计算的比率所选择的最大网格（2 K×2 K），每个处理器的数据集尺寸仅仅是 0.5MB，所以这两个工作集合都能自如地容纳于高速缓存（如果使用了一个四维数组的表示，基本上就没有因冲突而产生的扑空）。因此，可能还需要选择更大一些的问题规模。要使两个子行构成的第一个工作集超出一个 1 MB 的高速缓存，这意味着一个子行要有 64 K 个点，所以对于 64 个处理器的机器，整个的网格是 64 K×$\sqrt{64}$或者说 512 K 行（或列）。在这种情况下，每个处理器的数据集是 32 GB，大得有点不符合实际。但是，使另外一个重要的工作集（即一个处理器的整个分区）不能被一个 1 MB 的高速缓存所容纳却是现实的。这就导致了或者有很多本地存储器流量，或者有很多人为通信（如果数据没有被合适地存放），我们将表示这种情况。可以选择一个问题规模，如每个处理器 512×512 个点（2 MB）或者说整体 4 K×4 K 个点。这样做不会填满机器的存储器，所以可以再选择一种问题的规模，如整体 16 K×16 K 个点或者说每个处理器 32 MB。现在有 5 种问题规模：256×256，1 K×1 K，2 K×2 K，4 K×4 K 和 16 K×16 K。■

第 4 步：使用空间局部性

假设以共享地址空间实现的方程求解器内核中用于表示网格的数据结构是一个二维数组。作为它的重要的工作集，一个处理器的分区在跨越不同网格扫描时，可能已经不在它的高速缓存中了。即使高速缓存的容量足够大，高速缓存冲突仍可能相当频繁，因为一个处理器分区的子行在地址空间中是不连续的。在任何一种情况下，如果工作集不能被高速缓存所容纳，那么将一个处理器分区分配到具有分布式存储器机器的本地存储器中是很重要的。主存分配的粒度是页，通常为 4～16 KB。如果一个子行的尺寸小于页的尺寸，合适的分配是非常困难的，而且可能会导致很多人为的通信。但是，如果一个子行的尺寸是页尺寸的几倍，分配则不成问题，几乎不会有什么人为的通信。这两种情况都可能是符合实际的，所以我们应该试着来表示这两种情况。如果页的尺寸是 4 KB，已经选择的前三个问题的规模具有小于 4 KB 的子行，因此它们不能被合适地分布；后两个问题有大于或者等于 4 KB 的子行，因此只要网格对齐页的边界，它们就能够被合适地分布。所以，无需为此而扩展问题规模的集合。对于以一个四维数组表示的网格，处理器的网格分区在地址空间中是连续的，所以当分区大到必须分配时合适的分配是容易的。

223～224

一个更加严格的空间局部性相互作用的例子出现在不同的程序和体系结构的相互作用中。一个被称为 Radix 的程序是一个将在本章后面介绍的排序程序，被称为伪共享的体系结构的相互作用已经在第 3 章定义了，在第 5 章将要作进一步讨论。但是，在这里先看看其结果，从而阐明我们在选择问题规模时考虑空间相互作用的重要性是有益的。图 4-5 表明了运行在高速缓存一致的共享地址空间机器上的程序在使用相同数量即 p 个处理器的情况下，对于两种不同的问题规模 n（对256 K个整数和1 M个整数进行排序），其扑空率是如何随

○ 如果我们也关心除了扑空率之外同样受到尺寸影响的高速缓存的其他行为方面的话，剪裁掉扑空率曲线的扁平区域则可能是不适当的。我们在第 5 章讨论高速缓存一致性协议的折中方案时将会看到一个这样的例子。

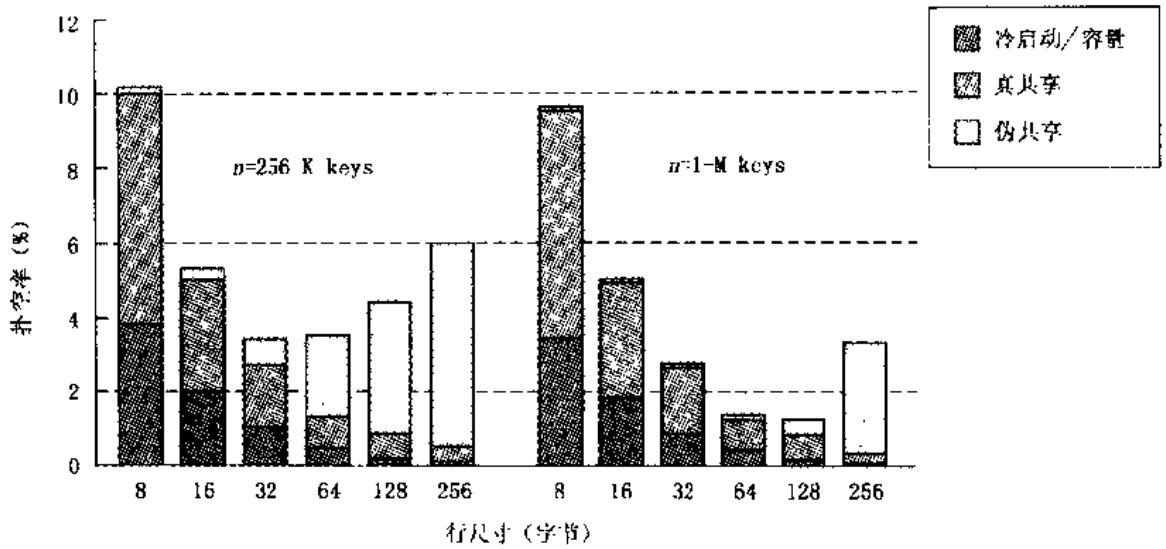


图 4-5 问题规模和处理器的数量对基数排序的空间局部性行为的影响。扑空率被分成冷启动/容量型扑空、真实共享（固有通信）扑空和由伪数据共享引起的扑空。对于给定的问题规模和处理器数量，当块的尺寸增加时，就会出现这么一个点，即文中讨论的临界率比块尺寸的多倍阈值要小，此时就会出现严重的伪共享。对于不同问题规模，这个阈值效应在不同块尺寸处发生。如果问题规模保持不变而处理器的数量变化时，也可以观察到类似的效应

225 高速缓存块的尺寸变化的。扑空率中的伪共享成分随着缓存块的尺寸而增加，当它变得显著时，会导致很多人为的通信破坏该应用的性能。如果给定机器的高速缓存块的尺寸，伪共享会不会破坏基数排序的性能，这取决于问题的规模（比较具有 64 个字节的块的直方条）。其结果是对于给定的高速缓存块尺寸，如果问题规模与处理器数量的比值小于某个阈值时，伪共享成分大；反之，当这个比值较大时，伪共享不显著。

很多应用在空间局部性与问题规模的相互作用中显示了这种阈值效应；在其他的应用程序中，特别是在很多非规则的应用程序如 Barnes-Hut 和 Raytrace 中，其数据结构和访问模式使得空间局部性不会随着问题规模的上升而增加很多。确认这种阈值的存在需要理解应用程序的局部性以及它和体系结构参数的相互作用，阐明评价中的一些微妙之处。

总结一下，简单的方程求解器说明了很多执行特征对问题规模的依赖，某些特征在和体系结构参数的相互作用中在阈值处呈现出拐点，而另一些则没有呈现。对于 $n \times n$ 的网格，p 个处理器，如果 n/p 的比率大，那么通信与计算比率就低，重要工作集不太可能被处理器的高速缓存所容纳，导致容量型扑空率高；但是即使是使用一个二维数组表示，空间局部性也很好。当 n/p 小时，情况则相反：会出现高的通信与计算的比率、差的空间局部性、伪共享（用二维数组表示），但几乎不存在本地容量型扑空。因此主要的性能瓶颈从第一种情况的本地访问转变为后者的通信。图 4-6 说明了对于 Ocean 应用程序的整体上的效应，Ocean 使用了类似于方程求解器的内核。

其他应用程序可能会呈现出对问题规模不同的特殊依赖性。尽管为评价机器而选择问题的规模并没有普遍适用的规则可循，方程求解器的内核只是一个很小的例子，但本章所给出的步骤提供了一个有用的方法，应该能保证从机器获得的结果并不是来自很容易从程序中去除掉的人为因素。如果我们要比较两台机器，选择能在两台机器上进行上述工作的问题规模是十分重要的。尽管要考虑的问题很多，实验表明，以一个应用程序评价规模固定的机器所

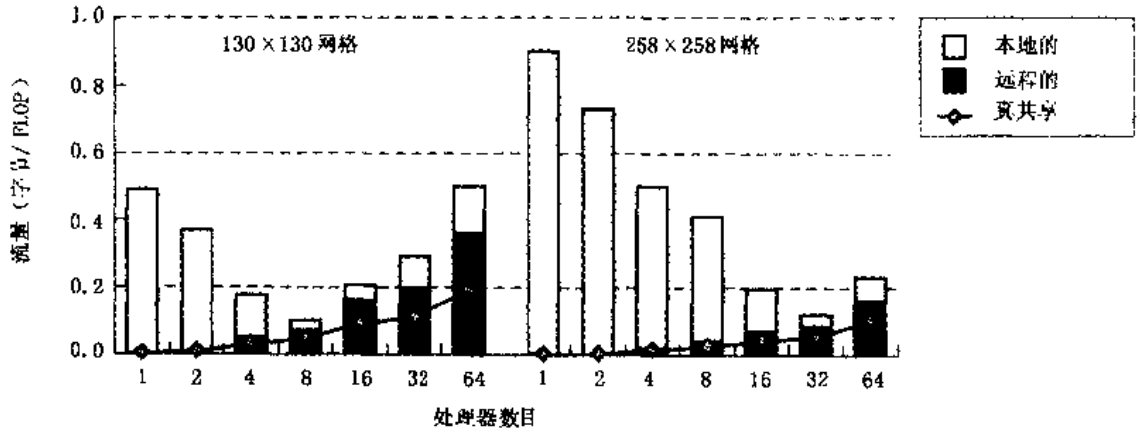


图 4-6　问题规模、处理器的数目、可被高速缓存容纳的工作集的效应。本图中显示了在共享地址空间中，Ocean 应用程序的存储器行为的效应。高速缓存扑空的流量（每次浮点运算或 FLOP 的字节数）被细分为本地的或者说节点内部的流量和远程的或者说经过网络的流量（即通信）。由真实数据共享（固有通信）所产生的流量也单独给出。远程流量随处理器的数目增加而上升，随着问题从小到大而减少。对于给定的问题规模，处理器数目增加时，工作集开始能被高速缓存所容纳，由本地扑空占主导地位转变为通信占主导地位。这个变化发生在较大问题规模和较大数量的处理器的情况下，因为工作集和 n^2/p 成比例。如果我们针对这两种问题规模，集中观察 8 个处理器的情况，我们可以看到对于小问题，流量主要是远程的（因为工作集能装入高速缓存），而对于较大的问题，流量主要是本地的

需要的问题规模的数目通常是相当小的，因为只有几个重要的阈值。

4.2.4　改变机器的规模

现在假设我们想要在处理器的数量变化时评价机器的性能。我们已经看到了在不同的扩放模型中如何扩放问题的规模，以及对由于并行性带来的性能改善应采用什么样的指标。剩下的问题是，在某种机器规模条件下，如何选择作为扩放开始点的问题规模。一个策略是从以前为固定数量的处理器所选择的问题规模开始，根据不同的扩放模型向上或向下扩放它们。我们可以将基本问题规模的范围缩小为三类：小、中、大，与三种扩放类型组合将会产生 9 组性能数据和加速比曲线。但是，可能需要注意当问题规模缩小时，它要保证能测试较小机器的能力。

另外一种策略是在一个单处理器上从一些精心选择的问题规模开始，然后在所有 3 个模型从下向上扩展它们，这里，选择 3 种单处理器问题的规模也是有道理的。小的问题应该是它的工作集能容纳于单处理器高速缓存的问题，这个问题在大机器上的问题约束（PC）扩放情况下不是很有用；但是在存储器约束（MC）扩放下或许甚至在时间约束（TC）扩放下它应该是没问题的。大问题应该是其重要工作集不能被单处理器的高速缓存所容纳的问题，如果这对于应用来说是切合实际的话。在 PC 扩放下，工作集合可能在某些点能够容纳于高速缓存（如果它随着处理器个数的增加而收缩）；而在 MC 扩放下，工作集不大可能被高速缓存所容纳，它可能不断地产生由容量型扑空造成的流量。大问题的一个合理选择是使它充满单个节点的几乎所有内存或者在节点上运行很长时间。因此，在 MC 扩放下，即使是在大系统中，它也会充满几乎整个内存。中等规模问题的选择是在大小两者之间做出一个明智的选择，如果可能它也要在单处理器上运行相当长的时间。一个突出的问题是，如何在不产生

226
227

超线性加速比问题的前提下，对于不能容纳于单个节点的内存的问题规模探索 PC 扩放。这里，一种解决方案是简单地选择这样一个问题规模，不是相对于单处理器，而是相对于多个处理器测量加速比；对于多个处理器而言，该问题确实能被存储器所容纳。

4.2.5 选择性能指标

在评价和比较机器时的一个重要的问题是应该使用的特定指标，正如没有适当地选择工作负载和参数很容易产生误导一样，没有用有意义的方法测量和表示结果也很容易得出错误的印象。总之，代价和性能都是比较机器和评价性能的重要指标。在评价机器随着资源（比如，处理器和存储器）的增加是否能很好地扩放时，我们关心的不仅仅是性能如何提高，而且也关心代价如何增加。即使加速比的提高比线性提高要小得多，如果运行程序所需要的资源的代价增加得没有加速比增加得那么快的话，那么使用较大的一些机器所花费的代价的确是值得的（Wood and Hill 1995）。总的来讲，一些对代价和性能的综合测量比简单的性能测量更合适。但是，我们可以分别测量代价和性能，代价在相当程度上依赖于市场，因此在这里主要关心的是性能测量的指标。

绝对性能和由于并行性产生的性能改善两者都是有用的指标，这里，我们考察一下使用这些指标来评价机器，特别是比较机器时的一些细致的问题并考虑其他一些基于处理速率（比如，每秒百万次浮点操作（megaflop））、资源使用、问题规模，而不是直接基于工作量和时间的指标的作用。某些指标显然是十分重要的，应该总是提供，而另外一些指标的使用却取决于我们以后要干什么以及我们所工作的环境。

1. 绝对性能

对于系统的用户来说，绝对性能是他最关心的性能指标。假设执行时间是我们关于绝对性能的指标，可以用不同的方法测量时间。首先，对于一个工作负载，在用户时间和墙钟时间之间要做出选择。用户时间是机器花在执行工作负载上的时间，不包括系统行为和其他可能分时共享机器的程序。墙钟时间是指运行工作负载而流逝的全部时间，包括所有与此相关的行为。其次，存在的另一个问题是在程序的全部进程中使用平均执行时间还是最大执行时间。

因为用户最终关心的是墙钟时间，在比较系统的时候，我们必须测量和表示它。但是，如果其他用户程序（不只是操作系统）作为多道程序和该程序的执行相互打扰的话，那么墙钟时间并不能帮助我们理解性能瓶颈。注意，在这种情况下，那个程序的用户时间也不会很有用，因为与不相关进程的交错执行会破坏程序的存储器系统的相互作用以及它的同步和负载平衡行为。因此，不管我们是否为了增强理解而提供更详细的信息，我们总是应该提交墙钟时间，同时描述执行的环境（批处理程序或多道程序）。如果我们想要理解一个特殊应用的性能，我们应该在只有操作系统介入的情况下独立地运行它。

228

同样，因为直到最后一个进程结束时，并行程序才结束，正是到达这一点的时间很重要，而不是进程的平均时间。用平均时间的概念容易忽略不平衡。当然，如果我们真正想理解性能瓶颈，我们会愿意看一看所有进程的执行性态（或者至少一个采样）分解成不同的时间成分的情况（如图 3-12）。执行时间的组成成分说明了为什么一个系统比另外一个系统性能好，工作负载对于研究是否合适（例如，未受到负载失衡的限制）。

2. 性能的改进或者加速

对任何一个扩放模型测量加速比时的一个问题是加速比的分子，即单个处理器的性能，应该实际测量什么。我们有 4 种选择：

1）在并行机器的一个处理器上并行程序的性能。

2）在并行机器的一个处理器上同一算法的串行实现的性能。

3）在并行机器的一个处理器上对于同一问题最优的串行算法和程序的性能。

4）在公认的一台标准机器上最佳串行程序的性能。

1）和 2）的区别在于，即使是在单处理器上运行并行程序也会产生额外的开销，因为它执行了同步操作、并行性管理指令或产生划分的代码或者甚至是为了省略这些操作所做的测试。这个额外开销有时候会相当显著。2）和 3）的区别在于最佳串行算法的并行化可能无法或难以有效地实现，所以并行程序所使用的算法会有别于最佳的串行算法。

从用户的观点看，使用 3）所定义的性能显然会导致比 1）和 2）更好和更加精确的加速比指标。但是，从体系结构设计者的观点看，在很多情况下使用定义 2）是可行的。定义 4）将机器的单处理器的性能融合进来，因此会产生和绝对性能相类似的比较指标。

229

3. 处理速率

一个经常引用的表现机器性能特征的指标是每单位时间内执行的计算机操作的数量（计算机操作与在应用层有意义的操作，比如事务处理或者化学结合不同）。经典的例子是用于浮点运算密集型程序的 MFLOPS（每秒百万次浮点操作）和用于一般程序的 MIPS（每秒百万条指令）。尽管它们在厂商的市场宣传材料中很是流行，但是已经有很多人写文章指出为什么它们不是好的通用性能指标，其基本的原因是，除非我们使用一个明确的、独立于机器的度量方法能够测量解决一个问题所需要的基本的 FLOP 或者指令的数目，而不是测量实际执行的 FLOP 或指令的数目，否则这些测量可以人为地膨胀：执行更多的 FLOP，花费很长时间的低级蛮干型算法可能反而会产生更高的 MFLOPS 速率。实际上，我们甚至可以通过在代码中插入无用但是廉价的操作来提高这样的指标。如果需要的操作数目是明确已知的，那么使用这些基于速率的指标和使用执行时间没有什么不同。MFLOPS 的其他一些问题包括：不同的浮点操作有不同的代价；即使在浮点操作密集型的应用中，现代的算法也要使用有很多整数操作的精巧的数据结构；这些指标还受到遗留的错误使用方式（比如，公布硬件的峰值速度而不是实际的应用达到的速度）的拖累。如果使用得合适，像 MFLOPS 和 MIPS 这样基于速率的指标对于理解基本硬件的能力还是有用的，但是若把它们作为机器性能的主要标志使用时，我们应该非常谨慎。

4. 利用率

体系结构设计者有时会以能否让处理引擎保持忙碌，不断地执行指令而不因为各种额外开销而停顿下来，作为衡量他的设计是否成功的标准。但是，现在有一点应该很清楚，处理器的利用率不是用户感兴趣的指标，也不是一个良好的惟一性能指标。它也可以被人为地扩大，有利于较慢的处理器，对于最终性能或者性能瓶颈也不能提供多少有用的信息。但是，作为决定是否要开始深入地探究程序或机器中的性能问题的出发点，它可能还是有用处的。同样的论点也适用于其他资源的利用率。利用率对于决定一台机器的设计是否在各种资源间平衡以及决定瓶颈所在是有用的，但是对于性能的测量和比较没有多少用处。

5. 问题规模

另外一个有趣的指标是能够获得特定的并行效率的给定应用的最小问题规模，它被定义为加速比除以处理器的数量（在给定的扩放模型下）。由于并行性产生的额外开销通常随着问题规模而相对减小，改进的通信体系结构的一个好处是运行较小问题的能力提高。在某种意义上，当处理器数量增加时保持并行效率不变产生了一种新的扩放模型，我们称之为效率约束扩放。当然，这个指标必须要小心使用，因为容量效应可能主导了通信的差别，而小问题可能无法突出系统的重要方面。并行效率是有用的，但不是一个通用的性能指标。

230

6. 性能改善的百分比

人们有时使用的一个用于评价因某个体系结构特性带来的性能改善的指标是执行时间或该特性所提供的加速比被改善的百分比。如果没有提到原来的并行性能（如原来的加速比），这种指标会在并行系统中起误导作用。比如，在一个 1 024 个处理器的系统中把加速比从 400 提高到 800 和把加速比从 1.1 提高到 2.2 两者的改进百分比相同，但是后者对于一个有 1 024个处理器的系统来讲，是没有什么意义的。如果问题的规模是导致加速比差的原因，那么用提高问题规模来产生很好的加速比却经常会严重地降低由特性所获得的改进。同样，这个指标是有价值的，但是必须由其他指标予以补充以避免误导。

总之，代价和性能都是需要考虑的重要因素。从用户的观点来比较机器，最感兴趣的性能指标是墙钟执行时间。但是，从系统设计师和力图理解程序的性能的编程人员，或者甚至从那些对机器性能有更为普遍性兴趣的用户的观点看，最好是看一下执行时间和加速比。这两种指标都应该在任何研究的结果中提交。理想的做法是，应该像在 3.4 节中讨论的那样，把执行时间分解成主要的组成成分。为了理解性能瓶颈，以每个进程为基础观察这种执行时间的成分分解，或将其看作平均及进程间分散程度的某种度量（简单的平均是不够的）是非常有用的。在评价变化对于通信体系结构的影响，或者比较基于相同底层节点的并行机器时，诸如实现一定目标所需的最小问题规模，这样的基于规模或者基于配置的指标是有用处的。像 MFLOPS、MIPS 和处理器利用率这样的指标可以用于特定的目的，但是仅用它们来表示性能则需要关于表示者的知识和完整性的很多假设，它们也受到遗留的不当使用方式的拖累。

4.3 对一个体系结构概念或设计权衡的评估

假设你是一个计算机公司的系统设计师，准备设计下一代的多处理器系统。你萌发了关于体系结构的一个新的概念，要决定是否把它包含在将要设计的机器中。对于上一代机器的性能和瓶颈，你可能有丰富的信息，这实际上可能是促使你进一步研究这个概念的最初动因。但是，你的概念和所拥有的数据并不全是新的。从上一代机器问世以来，从集成的层次到多处理器使用的高速缓存的规模以及组成结构，技术已经发生了变化。你将要使用的处理器可能不仅仅是快得多，而且也要复杂得多（比如，动态调度的四路指令发射和静态调度的单指令发射之比较），它的新的能力可能会影响你现在的新概念。操作系统也可能发生了变化，同样编译器、甚至是有意义的工作负载也都会发生变化，而且这些软件成分可能在机器实际建造和售出之前还会进一步变化。你所感兴趣的特性要求的硬件，特别是其设计时间的代价相当昂贵，而且你还要赶在截止期之前完成它。

231

在如此众多的巨大变化下，你所拥有的用于做出有关性能和代价决定的数据的有效性是令人怀疑的。你最多可以把它们和你的直觉揉合在一起使用，做出有见解、有素养的猜测。

但是如果特性的代价高，你可能希望再多做一些事情。你所能做的是构造一个能模拟你的系统的模拟器。你还需要将其他所有的东西——编译器、操作系统、处理器、技术和体系结构参数——固定在它们希望的配置上，只用你所感兴趣但尚不具备的特性来模拟系统，然后提交结果，判断它对性能的影响。然后，你或许应该检查一下你已经固定了的某些方面的敏感性，但是这些可能不容易预测。

为并行系统构造精确的模拟器是困难的。很多在扩展的存储器层次结构上的复杂交互很难被正确地模拟，特别是那些与资源占用和竞争有关的交互更是如此。处理器本身变得更加复杂，精确的模拟要求它们也必须在细节上被模拟。但是，即使你能够设计一个非常精确的模拟器来模拟你的设计，你仍然有一个大问题，即模拟是昂贵的，它要占用大量的存储器和时间，特别是当模拟大的问题和机器时。这意味着你不能模拟真实大小的问题和机器规模，你必须设法适度地缩小模拟的规模。

甚至连你的技术参数也可能是不固定的，你从一张白纸开始，想知道在不同的技术假设下你的概念工作得如何。现在，除了前面的工作负载、问题规模、机器规模几个坐标轴外，机器参数也可以变化。这些参数包括本地存储器层次结构中各层的规模和组成结构；分配、通信和一致性的粒度；时延、占用率和带宽等通信体系结构的性能特征。这些参数和那些工作负荷的参数一起产生了你必须遍历的巨大的参数空间。模拟的高代价和种种限制使得剪裁这个设计空间而又不失去太多的覆盖更加的重要。本节将讨论在模拟研究中，选择参数和剪裁设计空间的方法上的考虑，使用一个特别的评价作为例子。首先，让我们快速地看一下多处理器的模拟。 232

4.3.1 多处理器的模拟

尽管是要模拟多个进程和处理节点，模拟本身却可以在一个单处理器上运行。一个叫做访问生成器的部件扮演了并行机器中处理器的角色。它模拟了处理器的行为，产生对存储器的访问（同时带有表明访问是来自哪个处理器的进程标识）或者对模拟存储器系统和互连网络的模拟器的命令（比如发送或接收，见图 4-7）。如果是在单处理器上运行模拟，被模拟的不同进程分时共享该单处理器，由访问生成器进行调度。一个调度的例子是每当进程发出对

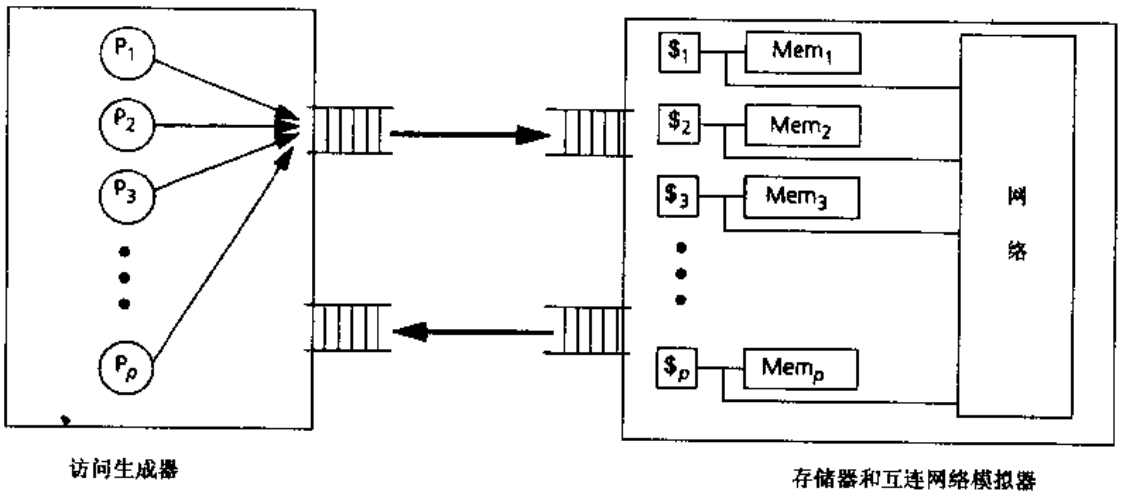


图 4-7 执行驱动的多处理器模拟。被模拟的处理器发出对存储器系统模拟器的访问，存储器系统模拟器模拟扩展的存储器层次结构并且向被模拟的处理器（访问生成器）反馈定时信息。$ 1，$ 2 等表示高速缓存

存储器系统的访问时就让它退出运行，允许另一个进程开始运行，直到那个进程产生了它的下一次访问。另外一个调度的例子是在每个模拟的时钟周期中重新调度一次进程。存储器系统模拟器模拟不同处理节点上所有的高速缓存、主存以及互连网络本身，它对数据通路、时延和竞争的模拟可以是任意复杂的。

访问生成器（处理器的模拟器）和存储器系统模拟器之间的耦合可以按照不同的方式组织，这取决于模拟所需要的精度和处理器模型的复杂程度。一个选择是轨迹驱动的模拟。在这种情况下，首先通过在一个系统上运行并行程序获得每个进程所执行的指令的轨迹，运行并行程序的这个系统可能与被评价的系统不同。这个轨迹替代了访问生成器：来自轨迹的指令被送入模拟目标多处理器扩展的存储器层次结构的模拟器。这里，耦合或者信息流只是单方向的：从访问生成器（这里只是一个轨迹）到存储器系统模拟器。

模拟的另一个更加流行的形式是执行驱动的模拟，它提供双向的耦合。在执行驱动的模拟中，当存储器系统模拟器接收到来自于访问生成器（现在是一个程序，而不是一个预先决定的轨迹）的访问或者命令时，它模拟在扩展存储器层次结构中所经过的访问路径，包括和其他访问的竞争，并向访问生成器返回满足该访问所花费的时间。和关于公平性及保持同步事件的语义的考虑一起，访问生成器程序，使用这个信息来决定下一次调度哪一个被模拟的进程以及何时从那个进程发出下一条指令。因此，从存储器系统模拟器到访问生成器发生了反馈，像在一台真实的机器中一样；这种反馈影响着后者的行为提供比轨迹驱动的模拟更高的精度。为了允许在事件和访问的模拟中最大限度的并行性，存储器系统和网络的大部分组成部件也被模拟成由模拟器调度的独立的相互通信的线程。模拟器维护一个全局的模拟时间，即被模拟的机器已经看到的虚拟时间，而不是模拟器本身已经运行的真实时间。这正是我们在决定被模拟的体系结构中工作负载的性能时要查看的时间和赖以做出调度决策的时间。除了时间以外，模拟器通常保存大量的关于各种有意义的事件的统计数据，这就提供了丰富的详尽的性能信息，这些信息在真实的系统中虽不是不可能获得，但至少也是难以得到的。但是，模拟的结果可能会因为缺少可信性而遭到玷污，因为它毕竟只是一次模拟。当必须要模拟复杂的、动态调度的、多指令发射的处理器时，精确的执行驱动的模拟也是要困难得多。习题4.9讨论了模拟技术中的一些折中。

4.3.2 缩小模拟的问题和机器参数的规模

如果知道了模拟是用软件实现的，涉及很多被非常频繁地重新调度的进程或者线程（精度要求越高，重新调度越频繁），模拟非常昂贵这一事实就并不令人感到惊奇了。对模拟本身的研究正在进行，使其加速，使用硬件仿真而不是软件模拟（Reinhardt et al. 1993; Goldschmidt 1993; Barroso et al. 1995），但是所取得的进展并不是那么显著，不能改变必须大大缩小参数的事实。

关于缩小问题和机器参数的一个棘手的问题是，我们希望运行较小问题的缩小规模的机器能代表运行较大问题的全规模的机器。不幸的是，没有保证实现这一点的规则可循。不管怎样它是一个重要的问题，因为这是大多数对体系结构折中的评价中存在的现实。我们至少应该理解这种扩放的局限性，认识哪些参数能被可信地缩小，而哪些不能，发展出一些能帮助我们避免主要误区的指导方针。让我们先来考察一下缩减问题规模和处理器的数量，然后解释一些和低层次的机器参数相关的难点。再说一遍，为具体起见，我们仍然将注意力集中

在高速缓存一致的共享地址空间的通信抽象上。

1．问题参数和处理器数量

先考虑问题参数。我们应该先找到那些如果存在就会严重影响模拟时间，但是却很少影响和并行性能相关的执行特征的问题参数。一个例子是很多科学计算，比如 Ocean 或者甚至 Barnes-Hut 所执行的时间步的数量，或者是在简单的方程求解器中的迭代次数。在时间步之间，操作的数据值会起很大的变化，但是行为特征不会变化太大。在这种情况下，我们的模拟可以只运行少数几个时间步[⊖]。

不幸的是，很多应用程序的参数要影响和并行性相关的执行特征。当缩小这些参数时，我们也必须减少处理器的数量，因为不这么做的话，我们可能会得到完全没有代表性的行为特征。但是，用具有代表性的方法做到这一点是很困难的，因为我们面临着很多约束，这些单个的约束都难以满足，要让它们彼此协调或许是根本不可能的。这些约束包括：

- 保持程序各阶段中花费的时间分布。用于执行不同类型的计算，比如 Barnes-Hut 中树的构造和力计算阶段，所花费的相对时间量最可能随着问题和机器的规模变化。
- 保持关键的行为特征。这包括通信与计算的比率、负载平衡、时间和空间局部性，它们全都会以不同的方式扩放！
- 保持应用程序参数之间的规模扩放关系。
- 保持竞争和通信的模式，这特别难，因为突发性是很难预见和控制的。

一个更加现实的目标是，当缩小规模的时候，不是保持真正的代表性，而是至少覆盖与研究最关心的行为特征相关的实际运行点的一个范围，避免不切实际的场景。因此，不能说缩小规模的模拟是有定量代表性的，但是可以用它们来获得见识和粗略的估计。有了这个更加适度的目标，让我们假设我们已经以某种合理的方法缩小了应用程序的参数和处理器的数量，现在来看看如何扩放机器的其他参数。 235

2．其他的机器参数

当问题与机器的规模缩小时，它们与低层次的机器参数的相互关系与全规模问题时不同。所以必须小心地调整这些参数。

考虑高速缓存或者复制存储器的尺寸。假定我们为方程求解器内核所能模拟的最大的问题和机器的配置是 512×512 的网格，在 16 个处理器（即每个处理器 128 KB）上运行。如果不缩小每个处理器的 1 MB 高速缓存的话，将无法表示重要工作集无法容纳于高速缓存的情况。关于扩放高速缓存的关键点在于，必须理解在所讨论的各个实际或不实际的工作点上，相关的工作集如何扩放（如图 4-4 所示），在此基础上对高速缓存进行扩放。一般来说，完全不调整高速缓存的大小或简单地根据数据集的尺寸或问题的尺寸按比例缩小高速缓存是不合适的，因为高速缓存的大小是与工作集的尺寸而不是与数据集或问题的尺寸最密切相关。例 4.6 和图 4-8 说明了如何根据给定问题的规模和机器的尺寸选择高速缓存的大小。我们还应该保证我们所模拟的高速缓存不会变得非常小，因为这样做会受到非代表性的映射和人为

⊖ 当然，我们现在应该从测量中忽略初始化和冷启动的阶段，因为在减少了时间步之后，它们在运行中的影响就要比在实际情况下大得多。如果我们希望在一段长的时间内行为会发生显著的变化，正像在 Barnes-Hut 或者其特征变化剧烈的应用中那样，那么，我们就可以从一台具有我们正在模拟的问题配置的真实机器的执行中周期性地卸出程序的状态，并且以这些卸出的状态作为输入数据集，启动几个样本模拟（在每个样本模拟中，不测量冷启动）。也可以用其他的采样技术来降低模拟的代价。

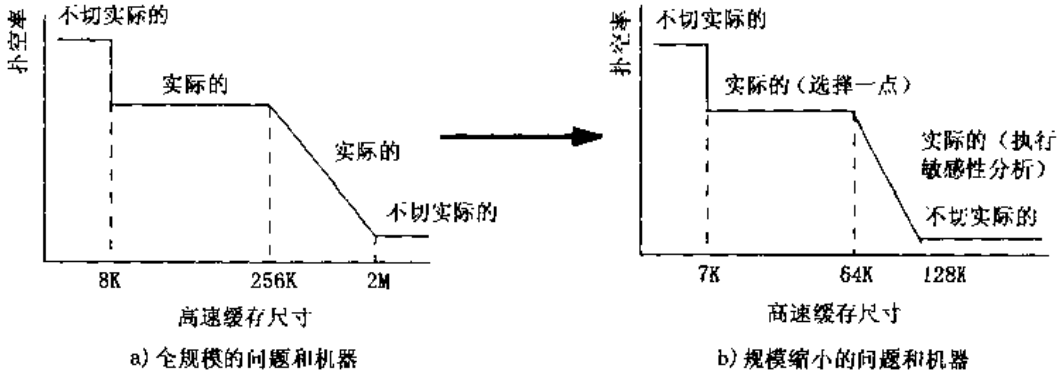


图 4-8　为缩小的问题和机器选择高速缓存的尺寸。a）基于我们对工作集的尺寸及扩放的理解，我们首先决定工作集曲线的哪些部分对于在具有全规模高速缓存的机器上运行全规模问题是实际的。b）然后我们计划或测量在当前模拟的较小规模的问题和机器上工作集曲线看起来是怎样的，剪掉相应的不实际部分，对实际的部分选取有代表性的工作点（高速缓存的尺寸），方法与例 4.4 所讨论的类似。对于不能被剪除的部分，我们可以进行必要的敏感性分析

碎片的影响。除处理器高速缓存之外，类似的论点也适用于复制存储器，包括那些仅保存通信数据的存储器。

例 4.6　在 Barnes-Hut 应用中，假定运行 $n = 1$ M 粒子的全规模问题时最重要的工作集的尺寸是 150 KB，又假定目标机的每个处理器有 1 MB 高速缓存，而你只能模拟具有 $n = 16$ K个粒子的执行情况。将高速缓存的尺寸按数据集的大小成比例地缩小是否合适？你如何选择高速缓存的尺寸？

解答：回忆在第 3 章中 Barnes-Hut 问题的最重要的工作集按 $\log n$ 扩放，这里 n 是粒子的数量并与数据集的尺寸成比例。150 KB 的工作集可以很容易地放入目标机的 1 MB 高速缓存。由于其增长速率缓慢，在真实问题中工作集总是可以存放在高速缓存中。如果在模拟中按数据集的大小成比例地缩小高速缓存的尺寸，我们得到的高速缓存的尺寸将是 1 MB × 16 K/1 M，或者说 16 KB。而被缩小了的问题的工作集的尺寸是

$$150\ \text{KB} \times \frac{\log 16\text{K}}{\log 1\text{M}}$$

或者说 70 KB，这显然不能放入缩小后的 16 KB 的高速缓存。所以，这种形式的高速缓存的扩放产生了不能代表真实情况的工作点。因为我们希望在实际中工作集能放入高速缓存，我们应该选择足够大的高速缓存尺寸，从而总是能容纳工作集。■

当我们涉及到存储器层次结构的更低层次的参数时，对它们进行有代表性的扩放变得愈发困难。例如，扩放和高速缓存的相联度之间的关系非常难以预测，通常我们所能做的是不去改变其相联度。这样做的主要的危险在于当高速缓存的尺寸缩到非常小时，仍保持一个直接映射的相联度非常容易受到映射冲突的影响，这在全规模的高速缓存不会发生。除非存在着近乎完美的空间局部性，否则与其他一些存储器和通信体系结构的组织结构参数，如数据分配的粒度、传输和一致性的粒度等的相互关系也是复杂而且不可预料的。但是保持这些参数不变在很多情况下会导致严重的、非代表性的人为效应。在本章的习题中我们将看到一些例子。最后，当通信的频度和模式改变时，在保持代表性的前提下对时延、占用率和带宽等性能参数的适当缩减也非常困难。

总之，模拟的最好途径是尽可能运行真实规模（如果规模不太大的话）的问题。当需要缩小规模时，为了保证能覆盖重要类型的工作点，我们应该注意那些已经讨论过的指导方针和可能存在的误区，同时在推广结论时要小心。降低规模的可信度依赖于我们对应用的理解。一般来说，对仅仅要理解某些体系结构特性是否有益而言，使用缩小规模的方案还是可以的，但是如果要力图用它们来得出关于全规模情况的精确的定量结论是危险的。

236 ~ 237

4.3.3　处理参数空间：评价举例

现在考虑在通用的场景下试图评价某个概念时所产生的巨大参数空间的问题。为了使讨论具体化，考察一个在模拟时我们可能进行的实际的评价。再次假定一台具有高速缓存一致特性的共享地址空间的机器，其存储器在物理上是分布的。其缺省的通信机制是通过取和存实现的以高速缓存的块为单位的隐式的通信，但是我们希望探索是否能以尺寸较大的消息作为通信单位，降低端点的通信开销的影响以及通信的延迟。所以我们希望了解对这样的体系结构增加显式地发送较大的消息的能力的效果，这种能力叫做*块传输*，程序除了使用以高速缓存块为单位的标准传输机制之外还可以使用块传输（从而合并了共享地址空间和消息传递这两个编程模型）。例如，在方程求解器中，进程可以用单个块传输向其相邻的进程发送它的分区的完整的边界子行或子列。

在为这样的评价选择工作负载时，至少应该选择一些其通信可以被组织成较大的消息的负载，例如方程求解器。更为困难的问题是如何覆盖参数空间。我们的目标是三重的：

1）*避免不切实际的执行特征*。应该避免那些能导致不真实行为的特征，即在机器的实际使用中不会碰到的行为的参数组合（或运行点）。

2）*获得对真实执行特征的良好覆盖*。应该尽量保证那些在实际应用中会出现的重要特征能被表示出来。

3）*剪裁参数空间*。即使是在参数值的实际子空间内，为了在不损失多少覆盖度的前提下节约时间和资源，同时也为了决定何时需要做直接的敏感性分析，也应该基于对应用的理解，在允许的情况下尽量剪去工作点。

我们能根据研究的目标、技术（或特定硬件构造模块的使用）对参数的限制和对参数相互作用的理解来剪裁参数空间。

让我们用方程求解器为例来体验一次选择参数的过程。虽然应该对参数逐个考察，但在较晚的阶段出现的问题可能迫使我们重新考虑较早时所做出的决定。首先选定问题的规模和处理器的数量，因为这些是最受模拟资源的限制的。

1．*问题规模和处理器数量*

在选择问题的规模和处理器数量时应该考虑程序的固有的特征，这些特征在关于实际的机器的评价和缩减模拟的规模的讨论中已经涉及。例如，如果问题足够大从而使通信与计算之比非常小的话，那么块传输无助于改善整体的性能；如果问题足够小，从而使负载失衡成为主要的瓶颈时，采用块传输也无济于事。

238

现在把方程求解器的问题规模固定为 514 × 514 的网格，处理器的数量为 16，然后考察如何选择其他参数。

2．*高速缓存/复制的尺寸*

按照惯例，我们根据对工作集曲线的了解来选择高速缓存的尺寸。如果给定了工作集曲

线以及工作集如何扩放的知识，则为给定的问题选择高速缓存的过程与例 4.7 所讨论的根据给定的高速缓存尺寸选择问题规模的过程相类似。

例 4.7　图 4-9 给出了方程求解器的良好定义的工作集。在这种情况下你如何选择高速缓存的尺寸？

解答：虽然重要工作集的尺寸通常取决于应用的参数和处理器的数量，但它们的性质和工作集曲线的形状并不随这些参数而变化。因为我们知道在方程求解器中各个重要工作集的尺寸以及它们如何随参数而变化，如果我们也知道对于目标机而言真实的高速缓存尺寸的范围，那么我们就能够了解 1）期望工作集在实际情况下能容纳于高速缓存是否是不切实际的；2）期望工作集不能容纳于高速缓存是否是不切实际的；3）期望工作集在参数值的某些实际组合下能容纳于高速缓存，而在另一些组合下不能被容纳[○]，这种期望是否是切合实际的。所以我们能分辨曲线的拐点之间哪些区域代表了实际的情况，哪些不能代表。对于给定的问题的规模和处理器的数量，我们可以使用（固定的）工作集曲线来选择能避免非代表性区域的高速缓存的尺寸，覆盖代表性的区域并通过从中选择单一高速缓存尺寸来剪裁扁平的区域（如果我们关心的只是高速缓存的扑空率）。■

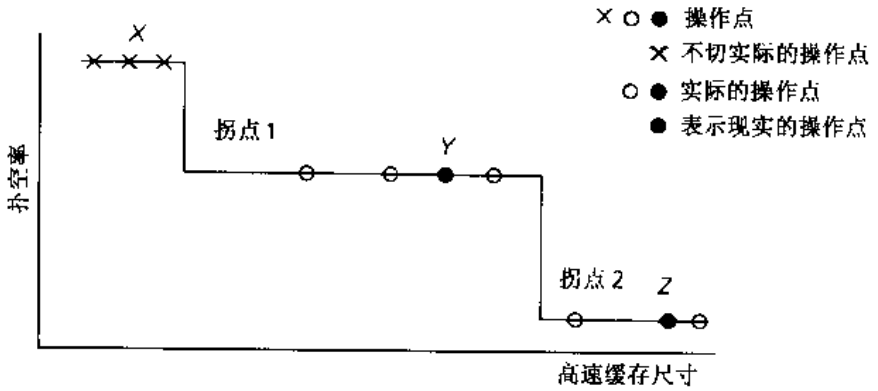


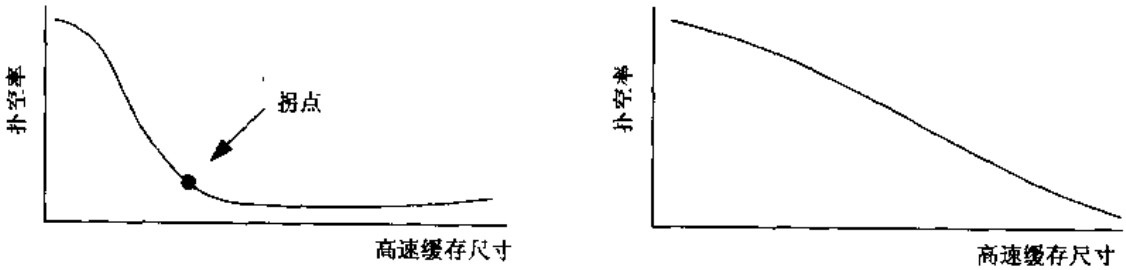
图 4-9　为使用方程求解器内核的评价来选择高速缓存的尺寸。拐点 1 大致对应一对 B 或 $n/\sqrt{p}$ 个元素的子行，取决于格的遍历是否被阻塞（对于 $B \times B$ 的块尺寸）。拐点 2 对应于一个处理器上的矩阵的分区（即数据集 n^2 除以 p）。后者的工作集根据 n 和 p 能否被高速缓存所容纳，所以 Y 和 Z 两者都是真实的工作点，必须被表示。对于第一种情况的工作集，如果遍历未被阻塞，它就不能被高速缓存容纳，但是在实际中我们知道有大的二级高速缓存，这种情况不大可能发生。如果遍历被阻塞，选择块尺寸 B 从而使第一种情况的工作集总是能被高速缓存所容纳。所以工作点 X 代表了不切实际的区域，因此被忽略。分块矩阵计算的情况在这方面与此类似

一个重要工作集是否能被容纳于高速缓存以一种有意思的方式严重影响块传输所带来的好处，其效应取决于工作集是否包含本地或非本地分配的数据。如果它主要由本地数据组成，正如数据适当分布的方程求解器的情况，但是它又不能被高速缓存所容纳，处理器将由于在本地存储器系统上受阻而花费更多的时间。其结果是通信时间变得相对地不重要，块传输没有多大帮助（块传输的数据对节点的本地流量的打扰更多，产生竞争）。然而，如果工作集主要包含非本地数据，我们将获得正面的效应：如果它们不能被高速缓存所容纳，就会有更多的通信，因此块传输就有更大的机会帮助性能的改善。

239

○ 给定尺寸的工作集能否被高速缓存所容纳除了取决于高速缓存的尺寸外，还取决于高速缓存的相联度和块的尺寸，但是如果我们假设至少 2 路的相联度的话，它在实际中通常不是主要的问题（我们在较晚些时候将看到这一点），所以我们现在忽略这些效应。

当然，工作集曲线并不总是包含由陡峭定义的拐点分隔的相对扁平区域。假如区域中有拐点，但是被拐点分隔的区域不是扁平的（见图 4-10a），若知道哪些区域是不切实际的，就能像以前那样剪切整个的区域。但是，如果区域是切合实际的但不是扁平的或在整个数据集能被高速缓存容纳之前不存在任何拐点（如图 4-10b），那么必须求助于敏感性分析，挑拣出靠近极端的点以及其间的某些点。再强调一下，适当的分析要求我们很好地理解应用的关键特征。



a) 拐点左边部分不是扁平的　　b) 对实际的高速缓存不存在扁平区域

图 4-10　不含被扁平区域所分隔的陡峭拐点的扑空率和高速缓存尺寸对照的曲线

剩下的问题是如何决定拐点的尺寸和拐点之间工作集曲线的形状。在简单的情况下，我们可以解析地完成这一任务。但是，算法复杂，常数因子难以预测，高速缓存的尺寸和相联度的效应难以分析。在这样的情况下，可以通过在给定问题规模和处理器数量的条件下测量（模拟）不同的高速缓存尺寸来获得曲线。所需要的模拟相对来说代价不大，因为工作集的尺寸并不依赖于详细的与定时相关的因素，例如时延、带宽、占用率和竞争，所以这些因素并不需要仔细地加以模拟（或根本无需模拟）。工作集如何随问题的规模和处理器的数量而变化也能够被适当地加以分析或测量。幸运的是，对增长速率的分析通常比预见常数因子要容易。而且，如果高速缓存足够大且结构合理的话（不是直接映射的高速缓存），像块尺寸和相联度这样低层次的问题通常也不会改变工作集（Woo et al. 1995）。 240

3. 高速缓存块的尺寸和相联度

除了问题规模、处理器数量和高速缓存的尺寸之外，高速缓存的块尺寸是决定块传输收益的另一个重要参数，但是，问题要稍微复杂一些。对于具有良好空间局部性的程序而言，大的高速缓存块本身的作用就类似于小的块传输，使得显式的块传输相对不那么有效。另一方面，如果空间局部性不好，那么当使用读和写通信时，由大高速缓存块引起的额外流量（由于碎片或伪共享）会消耗掉比必须消耗的多得多的带宽。对于块传输来说，差的空间局部性是否也会浪费带宽则取决于块传输的实现是将整个高速缓存块还是只将所需要的字经由网络流水传送。注意，块传输本身增加了带宽的需求，因为它要在较短的时间内完成相同量的通信（如果能做到的话）。所以，如果块传输是以高速缓存块流水传输实现的，而且空间局部性不好的话，当可用的带宽有限时，使用块传输可能非但无益反而有害，因为它可能增加对可用带宽的竞争。

幸运的是，由于当前技术的约束或可用于构造系统的模块的限制，通常能够限制高速缓存块尺寸增加的范围。比如，几乎所有的现代微处理器都支持 32～128 字节的高速缓存块，而我们可能已经选择了具有 64 字节高速缓存块的微处理器。当问题规模和高速缓存块尺寸的相互作用产生阈值时（比如，前面提到的基数排序的例子），我们应该保证覆盖阈值的

两侧。

高速缓存相联度的影响的程度却是很难预见的，实际的高速缓存的相联度一般较小（通常最多4路），所以能考虑的选择的数量也少。如果必须选择单一的相联度的话，最好避免使用直接映射高速缓存（至少在远离处理器的高速缓存层次结构的最底层是这样），除非知道所涉及的机器确实具有这样的高速缓存。

241

4. 通信体系结构的性能参数

在讨论了整个存储器层次结构的组成结构参数之后，让我们考虑一下通信体系结构的关键性能参数，即额外开销、网络延迟或通过时间以及它们如何影响块传输的收益。我们应该基于我们对所涉及的真实系统的期望来选择这些参数的基础值，而这种理解有助于我们决定哪些参数应该变化及如何变化。

一次高速缓存块的交换（扑空时）的额外开销成分越高，以较大的高速缓存块传输来构造通信从而降低额外开销就越重要。只要开始一次高速缓存块传输的额外开销不会高得抵消了收益，这样做总是对的，因为显式地启动一次块传输的额外开销可能比隐式地启动高速缓存块传输的额外开销要大。

出于同样的考虑，节点间的网络通信时间越长，通过大块传输获得的好处越大（对这一点有限制，我们在第11章详细考察块传输时会讨论它）。改变时延通常并不能产生出现拐点或阈值点的效果，所以为了考察时延的可能的范围，我们必须通过在范围内选择几个点进行敏感性分析。在实践中，我们通常根据所涉及的机器的目标时延来选择时延；比如，紧耦合的多处理器的时延一般比局域网上的工作站的时延要小得多。

可用带宽也是我们的块传输研究的一个重要问题。带宽也呈现很强的拐点效应，它实际上是饱和效应：或者有足够的可用带宽满足应用的需要，或者不能满足。如果能够满足，那么可用带宽是4倍还是10倍于需要都没什么关系。所以我们能够挑选一种小于需求的带宽和一种远高于需求的带宽。因为块传输研究对带宽特别敏感，我们也可以选择靠近边界线的带宽。在选择带宽的值时，我们应该仔细地考虑应用对带宽需求的突发性，虽然应用整体上的平均带宽需求可能不大，但是在它的突发通信期间仍然可能使较高的带宽饱和，导致竞争。

5. 重新修改选择

最后，我们可能经常会根据较晚的时候考虑的参数的相互作用修改我们早期做出的参数值的选择。比如，如果由于缺少模拟时间或资源，不得不使用较小的问题规模，可能试探使用非常小的高速缓存尺寸来表示重要工作集无法被高速缓存所容纳的真实场景。但是，选择非常小的高速缓存可能导致严重的人为效应，特别是如果我们使用直接映射高速缓存或大的高速缓存块尺寸的话（因为这将导致高速缓存中块的数量很少，从而产生很多碎片和映射冲突）。所以我们应该重新考虑我们为了表示这种情况所做出的问题规模和处理器数量的选择。

242

4.3.4 小结

前面的讨论说明，如果我们没有充分覆盖参数选择空间的话，评价研究的结果可能误导：我们可以很容易地选择参数和工作负载的一种组合（例如，相对较小的问题规模、大的高速缓存和小的高速缓存块尺寸），它能证明块传输这样的特性可以带来性能上的好处；我们也可以同样容易地选择另外的组合证明块传输没有优点。所以，在体系结构研究中综合可

靠的方法性的指导方针并且理解硬件和软件的有关相互作用是非常重要的。

尽管存在许许多多的相互作用，幸运的是我们能找到一些参数和性质，它们处于足够高的层次，能够推理它们不依赖于机器较低层次的定时细节，而且应用的关键行为特征又依赖于这些参数和性质。我们应该保证覆盖关于这些参数和性质的真实工作区域，即应用参数、处理器的数量、工作集和高速缓存/复制尺寸间的关系（也就是说，重要工作集是否能容纳于高速缓存）。基准测试程序集应该对它们的应用提供基本的特征，比如并发性、通信计算比、数据局部性以及它们对这些参数的依赖性，这样体系结构设计师无需再生成它们（Woo et al. 1995）。

在应用参数和体系结构参数的相互作用中寻找拐点和扁平区域也是重要的，因为这对覆盖和剪裁特别有用。最后，研究的高层次的目标和约束也能帮助我们剪裁参数空间。

关于工作负载驱动的评价的方法问题的讨论到此结束。在本章的剩余部分，将介绍本书中最常使用的其余的并行负载。它也说明了所有工作负载与方法有关的基本特征。

4.4 说明工作负载的特征

本书大量使用工作负载来定量地说明所讨论的体系结构上的折中，并评价用于开展案例分析的机器。主要为支持一致性共享地址空间的通信抽象而设计的系统在第5、6、8章中讨论，而消息传递型和非一致性共享地址空间系统在第7章中讨论。我们根据对应的程序设计模型编写了关于这些抽象的程序。因为这两种模型的程序的编写有很大不同（如第2章所述），而且某些重要的特征也是不同的，我们采用为一致性共享地址空间所编写的程序来说明工作负载的特征化。特别要指出的是，我们使用了6个以批处理模式运行（即一次运行一个）的不包含操作系统行为的并行应用和计算内核以及一个确实包含了操作系统行为的多道程序工作负荷。尽管我们使用的工作负载的数量不多，但这些应用代表了重要的计算类型，具有变化范围大的特征。 243

4.4.1 工作负载案例分析

我们用于共享地址空间体系结构的所有并行程序取自SPLASH-2应用集（见附录）。在前面的章节中，作为实例研究我们已经描述和使用了其中的3个（Ocean，Barnes-Hut and Raytrace）。这一节将简要地介绍一下我们将要使用但尚未讨论过的工作负载：LU、Radix、Radiosity和Multiprog。LU和Radix是计算内核；Radiosity是真正的应用；而Multiprog是一个多道程序工作负载。在4.4.2节中，我们将测量这些工作负载与方法有关的某些执行特征，包括数据访问的细目分类、通信与计算的比及其扩放、重要工作集的尺寸和扩放。我们使用这种特征化来为这些应用和后续几章的数据集合选择存储器系统的参数。

1. LU

密集LU因子分解是将一个密集矩阵A转换成为两个矩阵L和U的过程，L和U分别是下三角矩阵和上三角矩阵，它们的乘积等于A（即$A = LU$）[⊖]。它用于解线性方程组，在科学计算类应用以及像线性规划这样的优化方法中会碰到它。它是一个结构良好的内核，虽然

⊖ 如果一个矩阵的大部分元素非零，我们称其为密集矩阵（大多数元素都为零的矩阵叫做稀疏矩阵）。像L这样的下三角矩阵，其主对角线以上的元素均为零，而像U这样的上三角矩阵，其主对角线以下的元素都为零。（主对角线是从矩阵的左上角到右下角的对角线。）

不简单，但是为人们所熟悉且易于理解（Golub and Van Loan 1997）。

LU 因子分解的操作类似于高斯消去法，通过从一行中减去其他行的标量乘积而一次消去一个变量。LU 因子分解的计算复杂度是 $O(n^3)$，而数据集的大小是 $O(n^2)$。我们从第 3 章关于时间局部性的讨论中知道，这是通过分块发掘时间局部性的理想情况。事实上，我们使用分块的 LU 因子分解，不论是在串行还是并行的情况下，都远比非分块版本的效率要高。待因子分解的 $n \times n$ 的矩阵被划分成若干 $B \times B$ 大小的块，其思想是在移向下一块之前应尽可能多地重用一个块。我们现在认为矩阵是由 $n/B \times n/B$ 个块而不是 $n \times n$ 个元素组成，就像我们对元素所做的那样，一次消去和更新一块。此时对于小的 $B \times B$ 的块使用乘和求反这样的矩阵运算，而不是使用作用于元素的标量运算。图 4-11 显示了这种分块的 LU 因子分解的串行伪代码，它也定义了某些相关的术语。

244

```
for k ← 0 to N-1 do                          /*loop over all diagonal blocks*/
  factorize block A_k,k;
  for j ← k+1 to N-1 do                      /*for all blocks in the row of, and
                                              to the right of, this diagonal block*/
    A_k,j ← A_k,j * (A_k,k)^-1;              /*divide by diagonal block*/
    for i ← k+1 to N-1 do                    /*for all rows below this diagonal block*/
      for j ← k+1 to N-1 do                  /*for all blocks in the corresponding row*/
        A_i,j ← A_i,j - A_i,k* (A_k,j)^T;
      endfor
    endfor
  endfor
endfor
```

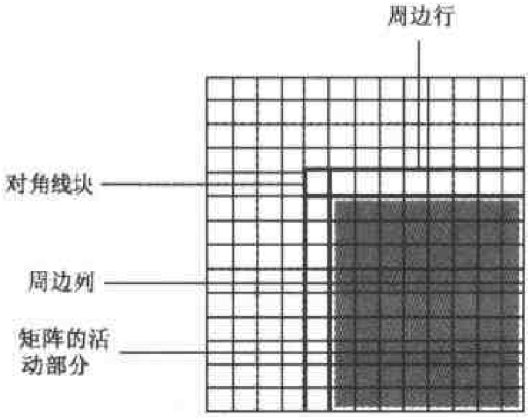


图 4-11　描述串行分块的密集 LU 因子分解的伪代码。N 是各维的块数（$N = n/B$），我们把矩阵看作 $N \times N$ 的块矩阵而不是 $n \times n$ 的元素矩阵。那么，$A_{i,j}$ 代表矩阵 A 的第 i 行和第 j 列的块。在最外层循环的第 k 次迭代中，我们称在 A 的主对角线上的块 A_k，k 为对角块，块的第 k 行和第 k 列分别为周边行和周边列。注意第 k 次迭代并不涉及矩阵前 $k-1$ 行和 $k-1$ 列中的任何块，也就是说，在当前最外层循环中，只有那些位于对角块右下方的正方形区域中的矩阵阴影部分是“活动”的。矩阵的其余部分已经在前面的迭代中计算过了，并且在后续的因子分解中也不再活动。在一个非分块的 LU 因子分解中，我们将类似地谈及对角线元素和元素的周边行和周边列

考虑一下分块的优点。如果我们不计算分块，处理器将计算一个元素，然后计算右边下一个分配的元素，如此进行直到当前行的尾，然后处理器将继续下一行。当它回到下一行的第一个活动的元素时，它将重新访问一个周边行元素（该元素在计算前一行的对应活动元素时已经使用过了）。但是，在此之前，计算已经流经了对应于矩阵整个行的数据，如果矩阵很大，那个周边行元素可能已经不再存在于高速缓存之中了。在分块的程序版本中，在图 4-11 的最内层循环各次迭代块层次上的计算中（即行计算 $A_{i,j} \leftarrow A_{i,j} - A^*_{i,k}\ (A_{k,j})^T$），我

245

们在回到先前访问过的数据之前在一个方向上仅仅前进 B 个元素，而那些先前访问过的元素仍然在高速缓存中而且可以重新使用。针对于各个 $B \times B$ 个块的操作（矩阵乘和因子分解）各自包含了 $O(B^3)$ 的计算和数据访问，每个块元素被访问 B 次。如果我们这样选择块的尺寸 B，使得一个块的 $B \times B$ 或者 B^2 个元素（加上某些其他数据）都能被高速缓存所容纳，那么在给定的块计算中，只有对元素的首次访问不会在高速缓存中命中，后续的访问都会在高速缓存中命中，这样在 B^3 次访问中有 B^2 次扑空，扑空率为 $1/B$。

在并行版本的程序中，我们把更新一个块的计算看作一个任务。图 4-12 提供了在一次最外层循环的迭代中块之间信息流的图示描述，它显示了在并行版的程序中我们如何将块（即任务）分配给处理器。因为计算的本质，朝向矩阵左上部分的块只在计算的最早几次最外层循环迭代中是活动的，而朝向矩阵右下部分的块却与相当得多的工作相关。所以把连续的行或块构成的正方形分配给进程（矩阵的简单的域分解）会导致不好的负载平衡。因此，我们在两个维度上让块在进程间交错，产生了一种叫做矩阵二维散布分解的划分：可以认为进程构成了 $\sqrt{p} \times \sqrt{p}$ 的网格，这个进程的网格就像糕饼切割器一样反复地压印在块矩阵上。进程负责对以这种方式分配给它的块进行计算，只有它能对这些块写入。这种交错缓解但没消除负载的不平衡，而分块保持了局部性也允许我们在消息传递型的系统中使用较大的数据传输。

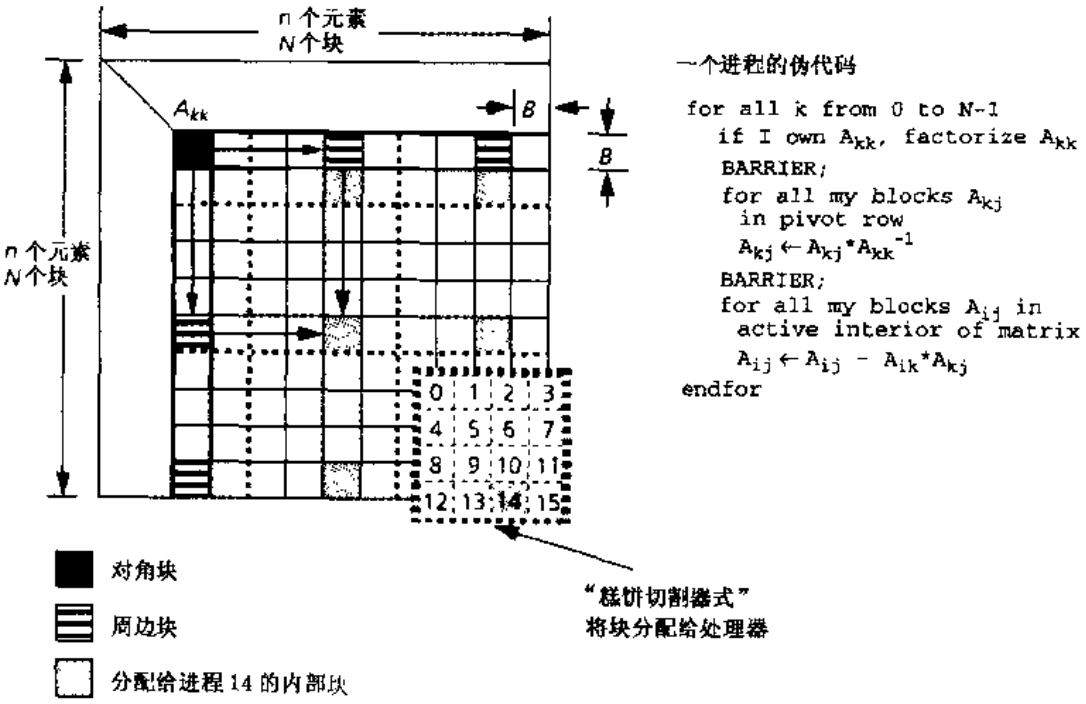


图 4-12　并行的分块式 LU 因子分解：信息的流、划分、并行伪代码。实线箭头显示了在一次外层（k）循环迭代中的信息流。在第一阶段，信息（数据）从对角块（它首先因子分解）流向周边行中的所有块。在第二阶段，矩阵活动部分的块需要来自周边行和周边列的对应元素

把计算分解成块而不是单个元素的缺点在于加大了任务的粒度并损害负载的平衡：因为块的数量比元素少，并发性降低了，每次迭代最大的负载失衡是与一个块而不是单个元素有关。在串行 LU 因子分解中，对块尺寸的惟一约束是：在一次块计算中用到的两个或三个块能容纳于高速缓存。在并行的情况下，理想的块尺寸 B 是由数据局部性和通信额外开销

（特别是在消息传递型的机器上）之间的折中而决定的，要降低通信开销倾向于较大的块；而另一方面，负载平衡则倾向于较小的块。所以理想的块尺寸取决于问题的规模、处理器的数量和其他体系结构参数。在实践中，16×16 或 32×32 个元素的块尺寸在大型的并行机器上工作得很好。

分块提供了块计算中数据的重用，数据也能跨越不同的块的计算而被重用。为了重用来自远地块的数据，我们可以显式地在主存中复制块并保持它们，或者在高速缓存一致性的机器上，依赖于足够大的高速缓存自动地完成这一点。但是，对改善性能而言，跨越块计算的重用不像块内的重用那样重要（显式的复制有其代价），所以在我们的程序中，我们并不在主存中显式地复制块。

至于空间局部性，因为现在分解的单位是二维的块，问题与 3.3.1 节讨论简单方程求解器内核时的问题很类似。所以我们用一个四维数组的数据结构来表示共享地址空间的矩阵，使得一块中的数据在地址空间中连续。前两个维度说明一个块，后两个维度说明块中的一个元素。这种表示方法允许我们以页的粒度在存储器中适当地分布块。（如果块比页小，我们可以用另外一个额外的数组维度来保证分配给一个进程的所有块在地址空间内是连续的。）然而，分块使得容量型扑空率足够的小，从而使 LU 因子分解时主存中的数据分布不成为主要的问题。使用高维数组保持一个块的数据连续的更重要的理由是，减少跨越一个块的子行和跨越块时的高速缓存的映射冲突，我们将在 5.6 节讨论这个问题。高速缓存冲突对于数组的尺寸和处理器的数量非常敏感，特别是对于直接映射的第一级高速缓存更是如此，它很容易抵消掉分块所带来的大部分好处。

246 ~ 247

并行 LU 因子分解没有使用加锁操作。使用了栅障操作分隔最外层循环迭代以及迭代中的不同阶段（例如，保证在使用一个周边行中的块之前先计算该行。）可以使用块级别的点对点同步来发掘更高的并发性，但是使用栅障使得编程容易得多。

2. Radix

程序 Radix 使用流行的基数排序法对一系列被称为键字的整数排序。假定有 n 个整数需要排序，每个整数的长度为 b 比特。该算法使用 r 个比特的基数，这里 r 由用户选择。这意味着一个 b 比特的键字可以被表示为一个包含 $\lceil b/r \rceil$ 个组的集合，每个组为 r 个比特（见图 4-13）。该算法经历 $\lceil b/r \rceil$ 个阶段或迭代。每个阶段从最低位的组开始，根据键字在所对应的 r 比特组中的值对键字排序，r 比特的组叫做一个数位[○]。在 $\lceil b/r \rceil$ 个阶段的末尾，键字被完全排序。在每个阶段中使用两个一维的大小为 n 的整数数组：其中一个叫做输入数组，存储该阶段输入的键字；另一个叫做输出数组，存储从该阶段输出的键字。一个阶段的输入数组是下一个阶段的输出数组，反之亦然。

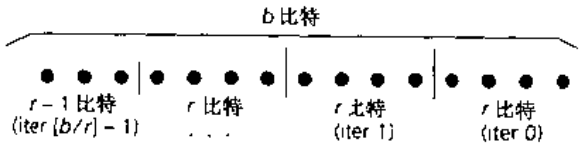


图 4-13　一个 b 比特的数字（键字）被分成 $\lceil b/r \rceil$ 个组，每组 r 个比特。radix 排序的首次迭代使用最低有效的 r 比特，依次类推

考虑在一个阶段内部的并行计算，它根据键字在某一特定数位上的值对所有的键字排序。并行算法把每个数组中 n 个键字对 p 个进程做划分，将前 n/p 个键字分配给进程 0，接

○ 从最低位而不是最高位的 r 比特组开始排序的原因在于它能导致一个“稳定”的排序，也就是说，具有相同值的键字在输出中的相对位置与它们在输入中的相对位置一样。

下来的 n/p 个键字分配给进程1，依次类推。分配给一个进程的那部分数组被放入对应的处理器的本地存储器中。在某个阶段分配给一个进程的输入数组中的 n/p 个键字叫做该阶段该进程的本地键字。在一个阶段中，进程执行下列步骤：

1）扫描 n/p 个本地键字，建立一个键字值的局部（每进程一个）直方图。该直方图有 2^r 个柱方项，这里 r 是数位中的比特数。如果碰到的键字在当前阶段的值为 i，那么直方图的第 i 个柱方项就加1。

2）当所有的进程都完成了步骤1）（由程序中的栅障同步决定）后，把局部直方图累计到一个全局的直方图去。如习题4.14所讨论的那样，这一操作通过一个并行的前序计算完成。全局直方图记录了对于当前数位的每个取值各存在多少个键字，还记录了对于每一个进程ID值 j，有多少个具有给定值的键字被ID值小于 j 的进程所拥有。

3）对 n/p 个本地键字做另一次扫描。对于每个键字，使用局部和全局直方图决定应该把该键字放到输出数组的哪一个（已排序）的部分去，并将键字的值写入输出数组的项中去。注意，要写入的数组项很可能是非本地的，这种可能性的期望值是 $(p-1)/p$（见图4-14）。这个步骤叫做置换步骤。 248

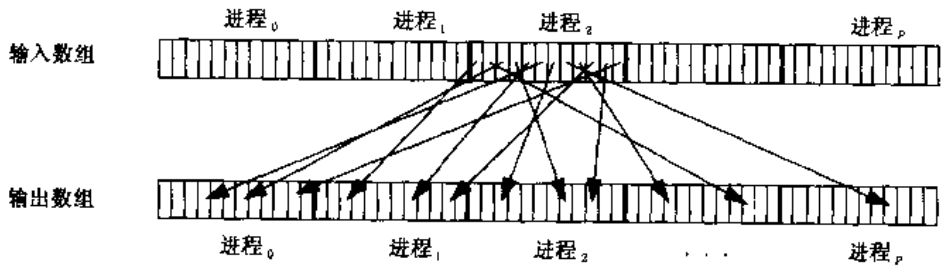


图4-14 radix排序阶段的置换步骤。在每个输入和输出数组中（它在后续的阶段中改变位置），分配给一个进程的键字（数组项）被分配到对应处理器的本地存储器中

读者可以在文献（Blelloch et al. 1991；Culler et al. 1993）中找到对radix排序算法及其实现的更详细的说明。在共享地址空间型的实现中，通信发生在置换阶段写入键字的时刻（或者，如果键字停留在写入者的高速缓存中，发生在下一次迭代的直方图生成阶段读出键字的时刻），也发生于从局部直方图构造全局直方图的时刻。与置换相关的通信是全部对全部的个人化通信（即每个进程与各个其他进程交换它的键字的不重叠的子集），但是，它是不规律和散布的，其精确的模式依赖于键字的分布。同步包括阶段之间的全局栅障和阶段内建立全局直方图的较细粒度的同步。后者可以采取互斥或者点对点的事件同步的形式，这取决于该阶段的实现（见习题4.14）。

3. Radiosity

Radiosity方法用于计算机图形学，用来计算包含散射表面的场景的全局照度。在层次型的radiosity方法中，一个场景最初被建模成包含了 k 个大的输入多边形，或者说片。例如，桌面或椅背可以是一个输入的片。我们要计算这些片中每对片之间光的传递相互作用。可以简单地认为这个算法完成以下功能：如果一对片之间光的传递比阈值强，其中之一（比如说大的那个）将被进一步分割，在由此产生的子片和其他片之间递归地计算光的相互作用。这个过程一直持续到所有片对之间的光传递都很低为止。所以，为了改善照度计算的精度，根据需要将片层次式地分割成子片。每一次分割产生4个子片，使每个片形成一个四叉树。如果最后不再分割的子片的数量是 n，那么对初始片数为 k 的情况，算法的计算复杂性是 249

$O(n+k^2)$。下面给出算法步骤的简单说明。读者能够在文献（Hanrahan，Salzman，and Aupperle 1991；Singh 1993）中找到算法的细节。

构成场景的输入片首先被插入一个二进制空间分割（BSP）树（Fuchs，Abram，and Grant 1983），它是一个便于片对之间可见性的高效计算的数据结构。初始时给每个输入片设立一个相互作用链表，该链表记录从该片能够看到的输入片，对这些片它必须计算相互作用。然后，通过下面的迭代算法计算照度：

1）对每一个输入片，计算由它的相互作用链表中所有的其他片而形成的它的照度，如果需要，将它或者其他片层次式地分割，递归地计算它们的相互作用（见图4-15）。

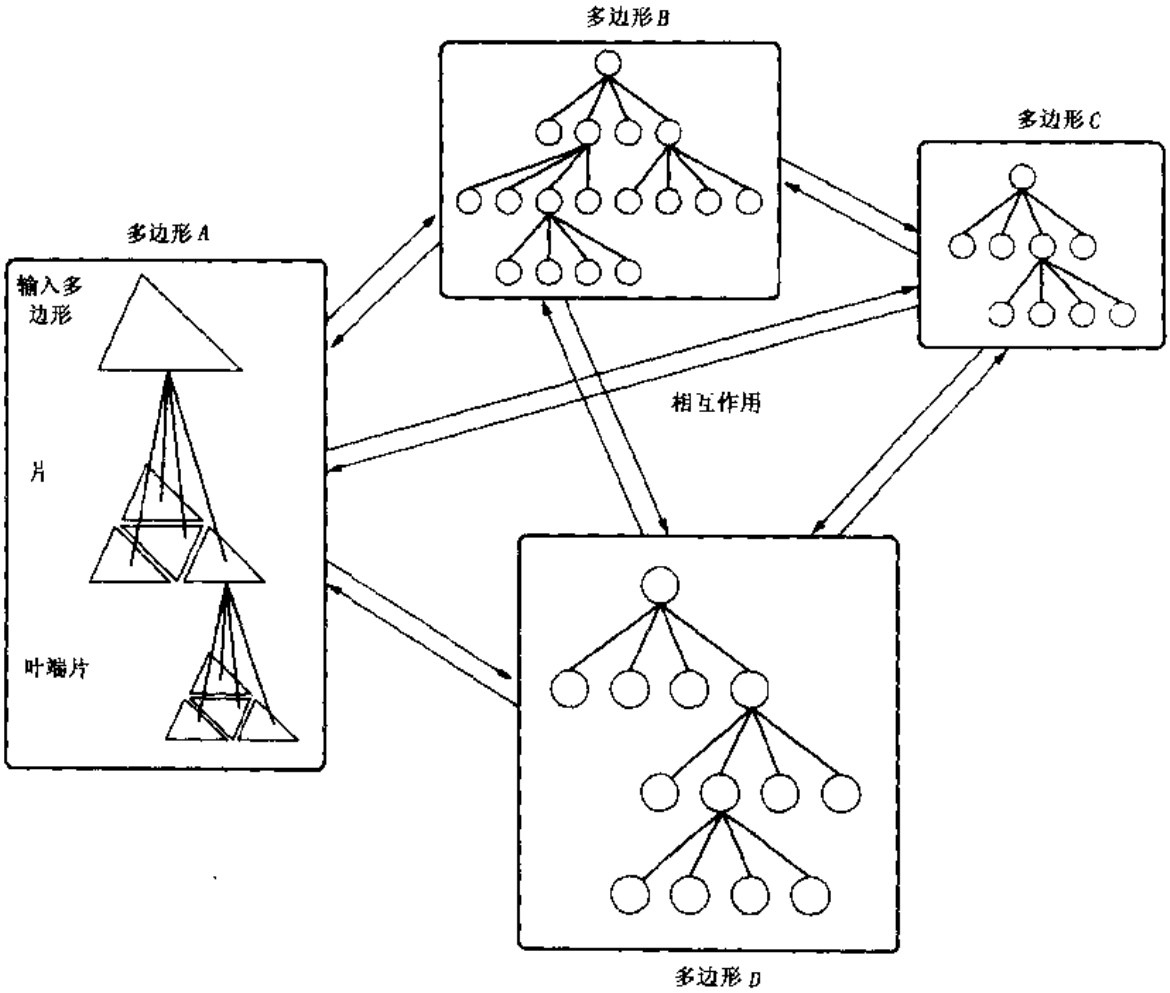


图4-15　随着照度计算的进展将输入多边形层次式地分割成四叉树。每一个输入多边形产生一个片的四叉树，该四叉树与其他四叉树的片相互作用

2）从位于四叉树的叶片处的片开始，把所有片的照度相加（由它们的面积加权）获得场景的总照度，将它与前一次迭代形成的照度比较，检查是否在确定的容差范围内收敛。如果照度未收敛，返回步骤1），否则的话，转向步骤3）。

3）为了显示对结果做光滑处理。

一次迭代的大部分时间花费在步骤1），因此让我们对它进一步考察。假定一个片 i 正在遍历其相互作用链表以计算它与其他片（四叉树节点）的相互作用。与另一个片（比如说 j）的相互作用包含这两个片的相互可见度计算以及它们之间的光的传递。（实际的光传递

是实际相互可见度与如果没有遮蔽因而相互完全可见而产生的光传递的乘积。）计算相互可见度包含从一个片到其他片遍历几次 BSP 树[⊖]；事实上，可见度计算占了整个执行时间的非常大的部分。如果相互作用的结果指出“源”片 i 应该被进一步分割，那么就对片 i 生成四个孩子，如果它们并没有因为先前的相互作用而已经存在；从片 i 的相互作用链表中去掉片 j 并将 j 加到 i 的每一个孩子的相互作用链表中去，这样这些相互作用会在以后计算。如果结果指出片 j 必须被进一步分割，那么片 j 在片 i 的相互作用链表中被 j 的孩子所替代。这意味着在片 i 和片 j 的每一个孩子之间将计算相互作用。这些相互作用本身可能又会引起进一步的分割，从而使得这个过程递归地继续下去（即如果片 j 的孩子在计算相互作用的过程中被再划分，片 i 最终要与片 j 之下的一棵片形成的树计算相互作用）。因为从一次分割产生的一个片的四个孩子在片 i 的相互作用链表中原地替换了其父亲，对由片 j 的后代组成的树的遍历是深度优先的。片 i 的相互作用链表被完全遍历之后才转向处理下一个片的相互作用链表（下一片可能是片 i 的后代或是另一个不同的片）。图 4-16 显示了这种层次式的相互作

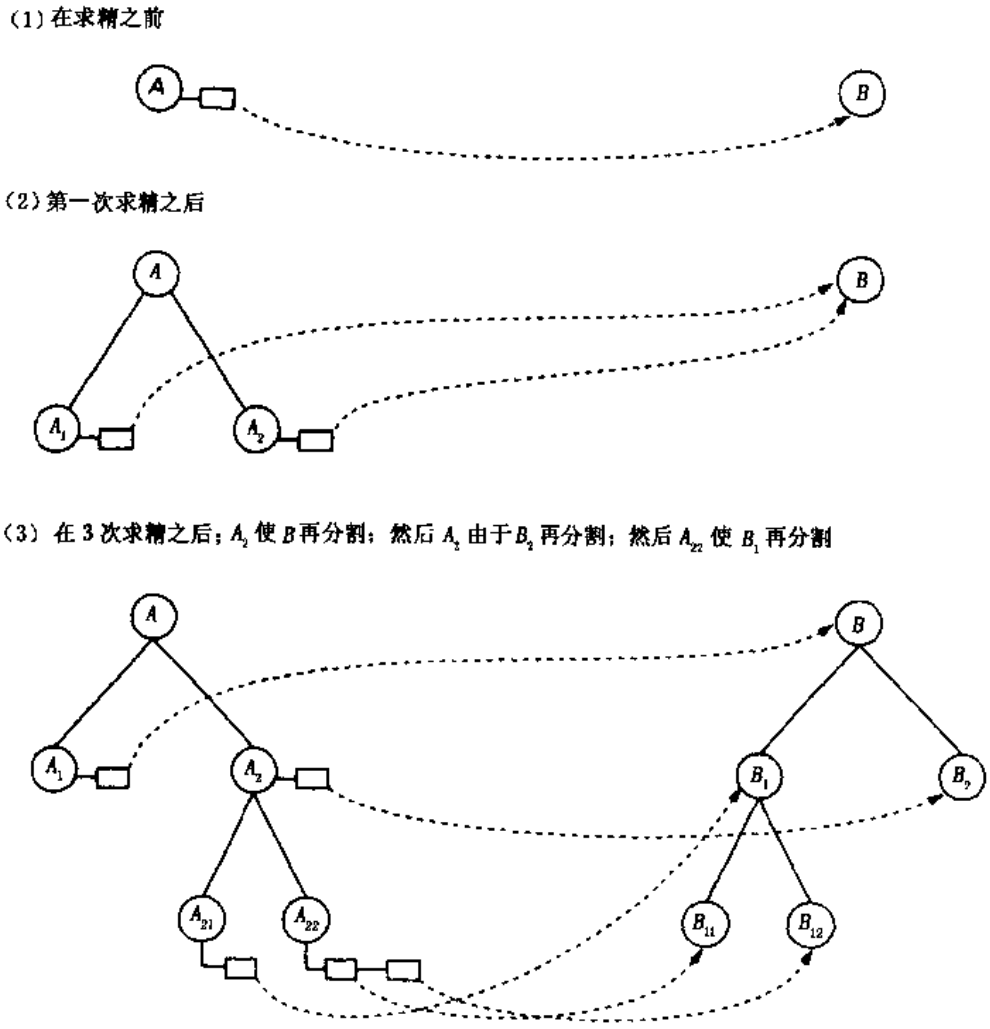


图 4-16　相互作用和相互作用链表的层次式求精。为了清楚起见，这里显示的是二叉树而不是四叉树，而且只显示了一个输入多边形的相互作用链表

⊖ 可见度的计算是在两个片之间概念性地发射一些射线并观察有多少射线未被场景中处于其间的其他干涉片遮蔽而到达目的地片。对每根这样的概念性射线，决定它是否被遮蔽的工作可以通过遍历从源到目的地片的 BSP 树而有效地完成。

用的不断求精过程。一轮迭代计算之后，所有的相互作用和求精都被完成，迭代算法的下一次迭代又以上一次迭代结尾形成的四叉树和相互作用链表开始。

在这个应用中可以找到三个层次的并行性：跨 k 个输入多边形，跨再分割这些多边形而形成的片，跨对一个片计算的相互作用。所有这三个层次都包含处理器之间的通信和同步。根据问题的规模、处理器的数量和机器的特性，我们把一个任务定义为一个片及其所有相互作用或者单个片对片的相互作用，从而获得最佳的性能。

因为计算和分割高度不可预见，我们必须使用任务队列和任务窃取来平衡工作负载。并行化的实现为每个处理器提供了它自己的任务队列。一个处理器的任务队列被初始化为初始可用的多边形-多边形相互作用的一个子集。当由于相互作用而分割一个片时，为这些子片而建立的新的任务被排入计算了相互作用并进行再分割的处理器的任务队列中。处理器从它的队列执行任务直到队列中没有任务为止。然后，它从其他处理器的队列中窃取任务执行。任务队列由加锁操作保护，当片被分割时，加锁提供了对片的互斥访问。（注意，分配给两个不同进程的两个片可能在它们的相互作用链表中有相同的片，所以两个进程可能会试图同时分割该片。）在一次迭代的两个步骤之间使用栅障同步。由于任务窃取和相互作用及再分割计算的次序，并行算法是非确定性的，它具有高度非结构化和不可预见的通信及数据访问的模式。

4．Multiprog

到目前为止我们讨论的工作负载仅仅包括一次运行一个程序的并行应用。但是，多处理器的通常用法，特别是小规模的共享地址空间多处理器，是作为多道程序工作负载的吞吐引擎。这类机器所支持的细粒度资源共享允许单一操作系统映像有效地服务于多个处理器。操作系统的活动通常是这样的工作负载的重要组成部分，而操作系统本身构成了重要的、复杂的并行应用。最后研究的一个工作负载是一个多道程序的（时分的）工作负载，由一系列串行应用和操作系统本身所组成。应用是两个 UNIX 文件压缩作业和两个并行编译，或者说 pmakes，在并行编译应用中，生成执行文件所需的多个文件被并行地编译和汇编。操作系统是由 Silicon Graphics 生产的 UNIX 版本，叫做 IRIX（版 5.2）。

4.4.2　工作负载的特征化

现在我们对所有我们提到过的工作负载的某些基本的特征定量化，包括被区分成读和写或共享和私有的数据访问、并发性和负载平衡、固有的通信与计算的比率及其扩放的方式、重要工作集的尺寸和扩放。与空间局部性相关的特征在后续章节中谈到特定的体系结构风格时再予以测量。在本节中，将给出并行应用在 16 个处理器上执行的定量特征数据和多道程序在 8 个处理器上执行的定量特征数据。我们还定性地或解析地讨论所涉及的特征如何随问题的规模而扩放，某些时候对它们进行测量。

250～253

1．数据访问和同步特征

表 4-1 总结了不同的工作负荷中基本的访问次数和同步事件（加锁和全局栅障）的动态频度。除非特别说明，输入数据集合符合本书中一直使用的缺省问题的尺寸。所选择的问题的规模足够大，可以对多至 64 个处理器的机器做实际评价；但又足够小，可以在合理的时间内被模拟。所以它们是处于在 64 处理器的机器上实际运行的数据集的小尺寸的一端，但是也很适合于较小规模的系统。

表 4-1　关于应用程序的综合统计数据

应用	输入数据集	指令总数（M）	总的 FLOPS（M）	访问总数（M）	读出总数（M）	写入总数（M）	共享读（M）	共享写（M）	栅障	加锁
LU	512×512 矩阵 16×16 块	489.52	92.20	151.07	103.09	47.99	92.79	44.74	66	0
Ocean	258×258 网格 容差 = 10^{-7} 4 个时间步	376.51	101.54	99.70	81.16	18.54	76.95	16.97	364	1 296
Barnes-Hut	16K 粒子 θ = 1.0 3 个时间步	2 002.74	239.24	720.13	406.84	313.29	225.04	93.23	7	34 516
Radix	256K 整数 radix = 1 024	14.02	—	5.27	2.90	2.37	1.34	0.81	11	16
Raytrace	汽车场景	833.35	—	290.35	210.03	80.31	161.10	22.35	0	94 456
Radiosity	室内场景	2 297.19	—	769.56	486.84	282.72	249.67	21.88	10	210 485
Multiprog 用户	SGI IRIX 5.2 两个 pmakes +	1 296.43	—	500.22	350.42	149.80	—	—	—	—
Multiprog 内核	两个 compress	668.10	—	212.58	178.14	34.44	—	—	—	621 505

注：对于并行程序，共享的读和写不过是指由应用进程发出的所有非堆栈的访问。所有这样的访问并不需要指向真正由多个进程共享的数据。工作负载 Multiprog 不是并行应用，所以它并不访问共享数据。表项中的短横线代表该项测量不适用或者对该应用没有做测量（例如，Radix 没有浮点数操作）。（M）表示在该行的测量是以百万为单位的

我们只是在父节点生成了子节点之后记录行为和定时统计数据。先前的访问（由主进程发出的）被模拟但并没有包括在统计数据之中。在大多数应用中，精确的测量在子进程被创建之后开始。但 Ocean 和 Barnes-Hut 两个程序是例外。在这两种情况下，我们能够利用机会，大大减少模拟所需要的时间步数（如 4.3.2 节所讨论的那样）；但是，我们必须忽略冷启动的扑空并允许应用在开始测量之前稳定下来。我们模拟很少几个时间步（对 Ocean 来说是 6 步，对 Barnes-Hut 来说是 5 步），在最初两个时间步之后就开始记录行为和定时统计数据。对于工作负载 Multiprog 来说，从靠近 pmake 开始位置的检查点收集统计数据，而对于所有其他的应用，仅仅考虑应用的数据访问；但对 Multiprog，还考虑指令访问的影响，进一步把内核的访问和用户应用的访问划分成独立的类。该表显示了不同的工作负载在把操作分为整数操作和浮点数操作、读和写、共享和私有等方面有很大的差别，表明它们很好地覆盖了这些坐标。

2．并发性和负载平衡

负载平衡的特征化是通过测量算法的加速比，也就是说，测量在 PRAM 体系结构模型上的加速比（第 3 章所讨论的）完成的，它假定数据访问和通信具有零时延（它们仅花费发出访问的指令所需的时间）。相对于理想加速比的偏离归咎于负载的不平衡、临界区的串行性以及由冗余计算和并行性管理所产生的额外的工作。

图 4-17 显示了使用缺省的数据集合，在多达 64 个处理器的系统上 6 个并行程序的算法加速比。其中 3 个程序（Barns-Hut、Ocean 和较小范围的 Raytrace）即使使用相对较小的数据集，在多达 64 个处理器的情况下仍有良好的加速比。这些程序的主要阶段是在大的数据集合（Barnes-Hut 中所有的粒子、Ocean 中的整个网格和 Raytrace 中的图像像素）上的数据并行。它们只是在某些全局归约操作和某些特殊阶段中受到有限的并行性和串行化的影响，而

254 ~ 255

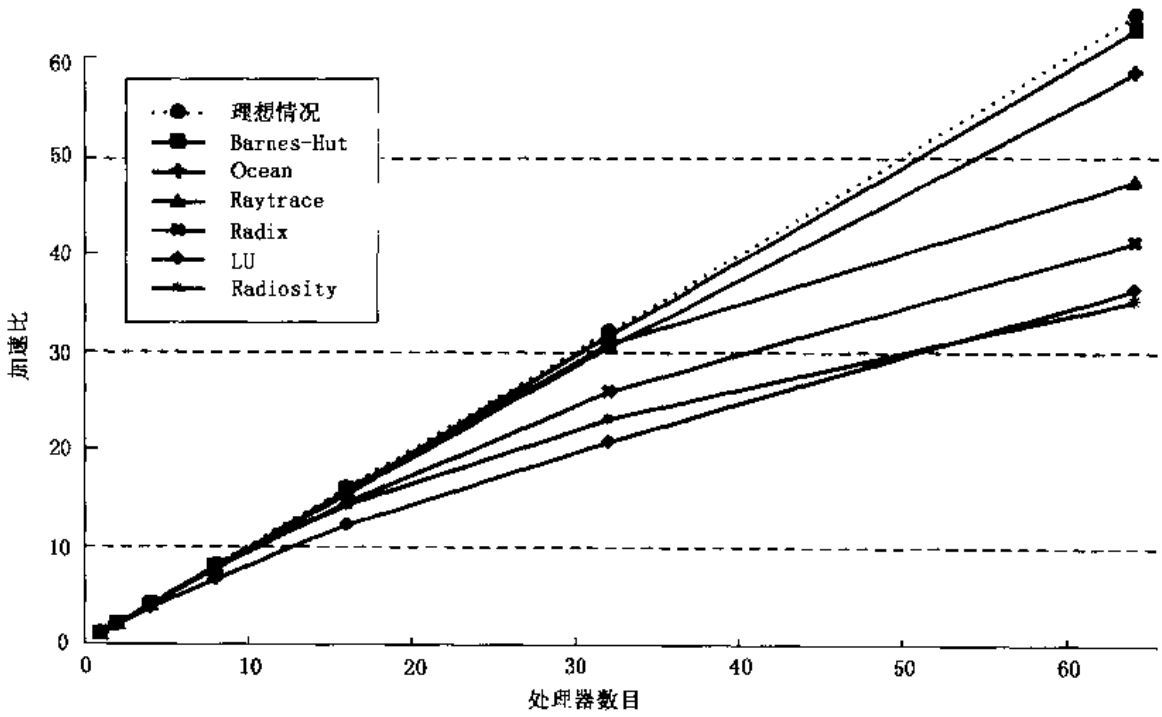


图 4-17　6 个并行应用的算法加速比。理想的加速比曲线表示在 p 个处理器情况下加速比为 p

那些特殊阶段就所执行的指令数而言并不占主导地位（例如，Barnes 中树的构造和靠近根的向上扫描的部分，Ocean 的多网格层次结构的较高的层次）。Raytrace 有一个产生麻烦的高度竞争的临界区，引起了串行化（事实上，这个临界区对于正确的执行并不一定是需要的，是为了记录某些重要的统计数据而使用它）。

所有 6 个程序在多达 16 至 32 个处理器的情况下都表现了良好的加速比。在处理器数量较多的情况下使用这些数据集加速比不是非常好的程序有 LU、Radiosity 和 Radix。在这几种情况下，其原因是输入数据集的尺寸而不是应用所固有的负载非平衡的特性。在 LU 中，尽管采用了面向块的分解，缺省的数据集导致在 64 个处理器的情况下负载显著的不平衡。较大的数据集（或较少的处理器）通过在分解的每一个步骤给每个处理器提供更多的块而降低不平衡性。对于 Radiosity，不平衡性也是由于使用了较小的数据集，虽然分析起来非常困难。最后，对于 Radix 在 64 个处理器时，较差的加速比是由将局部直方图累加到全局直方图时的前序计算所致（见 4.4.1 节），它不能被完全并行化。在这个前序计算上所花费的时间是 $O(\log p)$，而在其他阶段所花费的时间是 $O(n/p)$，因此当排序的键字的数量增加时，不平衡的阶段在整个工作中所占的比重将下降。所以，当选择较大的数据集时，即使是这 3 个程序也能被用来评价规模较大的机器。

256

我们已经满足了标准，这就是不要选择那些本质上不适合于所希望评价的机器规模的并行程序，并要懂得如何为现有规模机器上的程序选择合适的数据集。现在来考察一下固有的通信与计算的比以及程序的工作集尺寸。

3. 通信与计算的比

在通信与计算的比中，我们包括了固有的通信和一个字第一次被处理器访问时所产生的通信（即在测量启动之后发生的冷启动扑空）。在可能的条件下，数据是在物理分布的存储

器内适当地分布的，所以能够认为这种冷启动通信是本质性的而不是附加造成的。为了避免由于容量的限制或空间局部性不好所造成的附加的通信，我们模拟了每个处理器有无限多的高速缓存和每个高速缓存块由单个字组成的情况。我们所测量的通信与计算的比是所有的处理器平均来看，每条指令所通信的应用数据的字节数。对于浮点数密集型的应用（LU 和 Ocean），我们是使用每个浮点数操作（FLOP）所通信的字节而不是每条指令所通信的字节，因为 FLOP 的数量与指令的总数相比，对编译器的行为不那么敏感。

我们将首先观察一下对于表 4-1 所示的基本问题规模所测量到的通信与计算的比和所使用的处理器的数量的关系。这显示了在问题规模不变的前提下，该比值如何随处理器的数量而上升。然后，在任何可能的情况下，我们将解析地考察该比值是如何依赖于数据集的尺寸和处理器的数量的（表 4-2）。其他应用参数对通信与计算的比的影响将单独讨论，并且通常只是定性地讨论。

表 4-2　固有的通信与计算比的增长速率

应　用	增长速率
LU	$\sqrt{P}/\sqrt{DS}$
Ocean	$\sqrt{P}/\sqrt{DS}$
Barnes-Hut	大约$\sqrt{P}/\sqrt{DS}$
Radiosity	不可预见
Radix	$(P-1)/P$
Raytrace	不可预见

注：DS 是数据集的尺寸（以字节为单位），P 是处理器的数量。

图 4-18 给出了在基本的问题规模条件下，对我们的 6 个并行程序的测量结果。我们注意到的第一件事是平均的固有的通信与计算的比通常相当小。对于以 400 MIPS（每秒的百万指令）的速度工作的处理器来说，每条指令 0.1 字节的比率意味着大约 40 MBps 的数据流量，对于现代高性能的多处理器网络而言，这是相当小的。实际的流量比固有的流量要高得多，这既是由于附加造成的通信，又是由于在每次传输中都要有控制信息与数据一起传送。它指出，正是通信的突发性、通信的其他来源和通信的模式（比如，全部对全部或者长距离的通信）可能会引起通信带宽的问题。平均比率相当高的惟一的应用是 Radix，所以对于这个应用，在评价中仔细地模拟通信带宽就特别的重要。通信与计算的比低的一个原因在于，我们所使用的这些应用在被安排并行执行时已经很好地优化了。在实际中使用的应用，包括这些应用的其他版本，则可能呈现较高的通信与计算的比。 257

从该图我们观察到的下一个事实是不同的应用的通信与计算比的增长速率很不相同，显示了对这一行为特性有着的良好覆盖。表 4-2 以解析的方式总结了增长速率与处理器数量和数据集尺寸（未在图中显示）的关系。如果我们在某些应用（比如 Ocean）中使用不同的数据集的尺寸，通信与计算比将会有巨大的变化；但是在其他应用中，至少固有的通信不会有多大的变化。附加产生的通信则完全不同，在后面的章节中，我们将根据不同的体系结构的情况，考察由附加产生的通信所造成的通信流量。

显然，尽管增长速率是一个基本的参数，但重要的是应该认识到它并未揭示通信与计算比的表达式中的常数因子，在实践中，该因子比渐进线式的增长速率更为重要。例如，如果一个程序的通信与计算比仅随处理器的数量而对数式地增长，那么，以渐近线的增长的渐进 258

线方式看，它确实要比其比率按处理器数量平方根而增长的应用要小；但是，如果它的常数大得多的话，对于所有实际的机器规模，它的比率实际上要大得多。从图 4-18 中能够决定应用的常数因子。

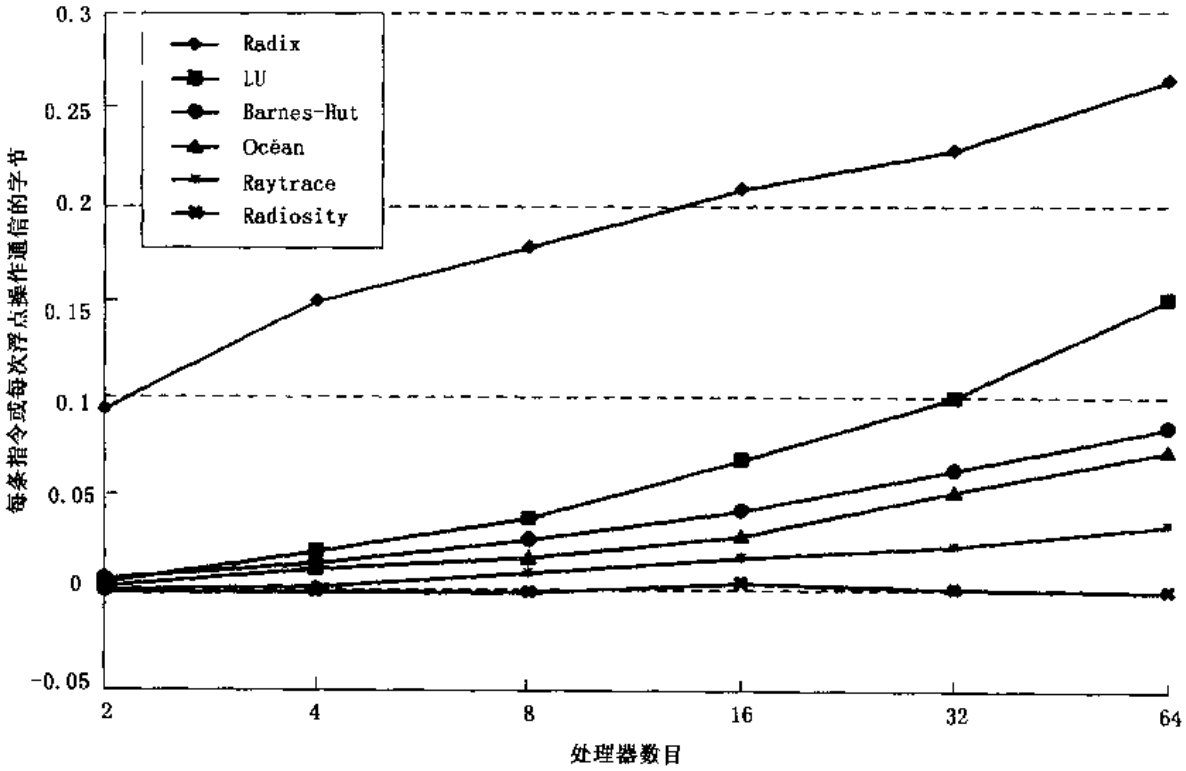


图 4-18　在 6 个并行程序中，对于基本的问题规模，通信与计算之比与处理器数量的对照

4．工作集的尺寸

测量一个程序的固有工作集的尺寸的最好办法是采用全相联的高速缓存和大小为一个字的高速缓存块，以不同尺寸的高速缓存模拟该程序的运行，发现扑空率和高速缓存尺寸的对照曲线的拐点。较小的相联度可能会使保持工作集所需要的高速缓存的尺寸大于固有的工作集的尺寸，使用多字组成的高速缓存块也会产生这一效果（由于高速缓存中的碎片所致）。在测量中，通过使每个处理器具有单级全相联高速缓存，采用最近最少使用（LRU）的高速缓存替换策略和 8 字节的高速缓存块，测量很接近固有工作集的尺寸。一般使用 2 的幂作为高速缓存的尺寸，但是为了识别拐点，在扑空率随高速缓存尺寸的变化更为显著的情况下，我们以更小的粒度改变高速缓存的尺寸。

图 4-19 显示了 6 个并行应用的扑空率与高速缓存尺寸的对照曲线，工作集被标注为 1 级工作集（L_1 WS）、2 级工作级（L_2 WS）等等。如我们在第 3 章所讨论的那样，像 Ocean 这样的应用有几个工作集，但我们的注意力集中在两个定义最清楚和最重要的工作集上。除了这些工作集以外，某些应用还有微小的、其尺寸不随问题规模和处理器数量而扩放的工作集；因此它们总是可以被容纳于最靠近处理器的高速缓存中，称这些工作集为 0 级工作集（L_0 WS）。典型地，这种工作集是由堆栈数据所组成，这种堆栈数据被程序作为对某一特定基本计算（例如 Barnes-Hut 中的粒子元的相互作用）的临时存储而使用并且在这些计算中反复使用。当这些工作集可以被观察到时，将它们在图中标出，但是不对它们作进一步的讨论。

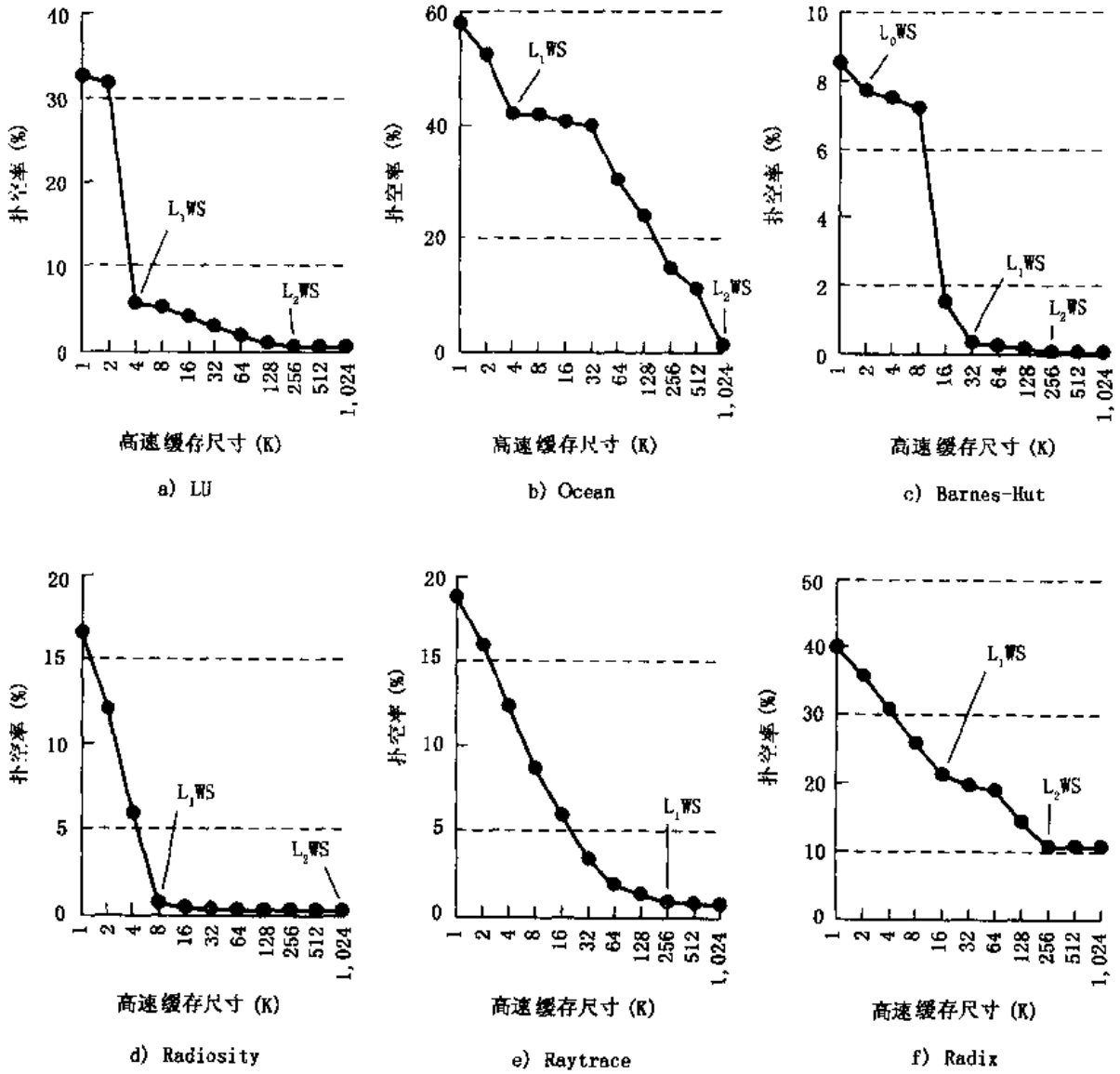


图 4-19　在 16 个处理器执行的情况下 6 个并行应用的工作集曲线。这些图显示了在每个处理器具有全相联第一级高速缓存和 8 字节的高速缓存块的情况下，扑空率和高速缓存尺寸的对照

我们看到在大多数情况下，工作集是非常明确地定义的。表 4-3 总结了对于不同的工作集其尺寸如何随应用参数和处理器数量而扩放；对于性能而言它们是否重要（至少在高效的高速缓存一致的机器上）以及对于合理的问题尺寸，是否能期望这样的工作集无法被现代的第二级高速缓存（具有合理的高速缓存相联度，至少高于直接映射）所容纳。其工作集在最后两列都为“是”的应用是 Ocean、Radix 和 Raytrace。回忆一下，在 Ocean 中，所有主要的计算流过一个或多个网格在一个进程中的分区。大的工作集包含整个网格在一个进程中的分区，它通过重用而受益。所以，这个大工作集是否能为现代的第二级高速缓存所容纳取决于网格的尺寸和处理器的数量。在 Radix 中，一个进程流过它的所有 n/p 个键字，同时大量地访问直方图数据结构（与所用的基数成比例）。所以，让高速缓存容纳直方图是重要的，但这个工作集没有被明确地定义，因为键字也同时流过。较大的工作集包含键字数据集在进程的整个划分，它或许能或许不能被高速缓存所容纳，这取决于参数 n 和 p。最后，在 Raytrace

260

表 4-3　SPLASH-2 测试程序集的重要工作集和它们的增长速率

程序	工作集 1	增长速率	重要?	实际中无法容纳于高速缓存?	工作集 2	增长速率	重要吗	实际中是否无法容纳于高速缓存
LU	一块	固定的(B)	是	否	DS 的划分	DS/P	否	是
Ocean	少数子行	$\sqrt{P}/\sqrt{DS}$	是	否	DS 的划分	DS/P	是	是
Barnes-Hut	一个星体的树状数据	$(\log DS)/\theta^2$	是	否	DS 的划分	DS/P	否	是
Radiosity	BSP 树	log(*polygons*)	是	否	非结构化	非结构化	否	是
Radix	直方图	Radix r	是	否	DS 的划分	DS/P	是	是
Raytrace	在光线之间重用的场景和网格数据	非结构化	是	是	—	—	—	—

注：DS 代表数据集的尺寸，P 是处理器的数量。

中，已经看到了工作集发散因而是不良定义的，所以可能变得很大，它取决于正在被跟踪的场景的特征和视点。对于其他的应用，我们期望对于实际的问题和机器的规模，重要工作集能够被高速缓存所容纳。在后续章节评价体系结构折中时，根据我们的方法，应该考虑这一点。特别是对于 Ocean、Radix 和 Raytrace，应该选择较大的工作集，覆盖能与不能被高速缓存容纳的两种场景；因为在实践中，这两种情况都存在。

261

4.5　结论

至此，对多处理器工作负载驱动式评价的主要问题已经有了很好的理解：选择工作负载，扩放问题和机器的规模，处理大的参数空间、选择度量指标。对每一个问题，都给出了一组应遵循的指导方针和步骤、对如何避免误区的理解以及对研究的局限性的理解。我们已经具备了在阅读本书的剩余部分时，做出对体系结构折中的定量说明的基础。本书中的实验是为了说明要点，而不是为了全面地评价那些折中，因为后者要求范围广泛得多的工作负载和参数变化。

我们已经看到了应该选择那些能够代表宽范围的应用、行为模式和优化级别的工作负载。尽管完整的应用和多道程序的工作负载是不可缺少的，但像微基准测试程序和内核程序这样较简单的工作负载也可以扮演重要的角色。

我们也看到了要实现适当的工作负载驱动式评价，要求既要理解工作负载的相关行为特性，也要理解这些特性和体系结构参数之间的相互作用。虽然这个问题是复杂的，但我们已经考察了处理巨大的参数空间的指导方针，以便对真实的机器以及体系结构上的折中进行评价，并且在仍然能覆盖各种真实情况的前提下对参数空间进行剪裁。

规模扩放的问题进一步强调了理解工作负载的相关性质的重要性，它影响所有重要的特征及相互作用。执行时间和存储器两者都可能成为扩放的约束，而应用通常有着一个以上决定关键执行特性的参数。我们基于对这些参数、它们的相互作用、它们对执行时间和存储器的需求的理解而扩放程序的规模。我们看到，由应用需求所驱动的实际的规模扩放模型在评价体系结构重要性方面得出的结论与仅仅扩放单个应用参数的初级模型的结论完全不同。事实上，规模扩放对于设计和评价都是重要的。在评价技术如何适应扩放（比如，相对于存储器和网络速度的处理器速度）的同时，理解应用的扩放以及它的内涵，对于决定未来机器合

适的资源分布是非常重要的（Rothberg，Singh，and Gupta 1993）。

在我们关于选择评价和结果表示的指标的讨论中也有很多有意思的问题。例如，了解执行时间（最好将每个处理器进一步细分为主要组成部件）和加速比都是应该加以表示的很有用的指标，而诸如 MFLOPS 或 MIPS 这样基于速率的指标或利用率指标对特定的目的而言，可能是有用的；但作为通用的指标，它们太容易受到问题的影响。

在本书的后续部分，将举例说明对具有共享地址空间的真实系统及体系结构折中的工作负载驱动式的评价，本章描述了用于该评价的主要工作负载，定量说明了它们的特征。（对于消息传递型的系统，将在第 7 章简要地考察一个标准的消息传递基准测试程序集，即 NAS 并行基准测试程序 II［NPB2］的性能。）现在，我们的基础已经足够牢固，可以进一步研究核心的体系结构和设计。 262

习题

4.1　1）假设为高速缓存一致的机器的通信体系结构建议了一种新的特性，你要对该特性进行评价研究。你的经理告诉你，你在评价中只能使用不超过 3 个并行程序，尽管这不符合你对问题的更好的判断，但你不得不服从。从本章和第 3 章所考察的 7 个并行程序（包括多道程序的工作负载）中，你将选择哪 3 个？为什么？

2）假如你知道该特性是为改善机器的通信带宽而设计的，你的选择会受到什么影响？

3）假如该特性是为了增加用于非本地分配的数据的有效复制存储，你将选择什么程序？

4.2　请说明与 MC 扩放相比较，TC 扩放的根本问题，试举例说明。

4.3　假定你必须评价系统的可扩展性。一种可能的方法是像本章所定义的那样测量不同规模扩放模型下的加速。另一个办法是决定要想达到 70% 并行效率，问题的规模应如何扩放。这两个方法的优点，特别是缺点和警示是什么？你实际将如何做？

4.4　你的经理要求你比较两类基于相同的单处理器节点的系统，但它们的通信体系结构有一些有趣的差别。经理告诉你，她仅仅关心 10 个特定的应用。她指示你，在为每个应用指定固定数量的处理器和固定的问题规模（你的选择）的前提下，只需要产生能说明哪个系统更好的单一数字形式的测量；尽管你认为对这些并行应用求平均不是一个好主意。在选定问题的规模之前你还要问她什么额外的问题？你将向她报告哪种平均值的测量？为什么？

4.5　一个系统经常会对某个应用呈现好的加速比，尽管它的通信体系结构对该应用并不是很合适。为什么会发生这样的事？除了加速比之外，你能否设计一种指标，它能够测量通信体系结构对于该应用的有效性。讨论设计这样一个指标可能碰到的问题和不同的方案。

4.6　你读到的一篇研究论文建议了一种通信体系结构机制，并告诉你，以一台具有 32 个处理器的机器的特定通信体系结构为基础，该机制对某些你关心的工作负载的性能改善达 40%。该信息是否足以使你决定将该机制包括到你将要设计的下一台机器中去？如果不是，请列出为什么不这样做的主要理由，并说明你还需要哪些其他的信息。假设你将要设计的机器也具有 32 个处理器。 263

4.7　假设你要设计一些实验来比较在一个共享地址空间机器上实现锁的不同方法，你想要

测量什么样的性能特性？你要设计什么样的微基准程序？你需要的专门性能测量是什么？然后对于全局栅障情况回答相同的问题。

4.8 你已经设计了一个方法，在第一章所讨论的 Intel Pentium Pro "quad" 这样的基于总线的共享存储器多处理器上，用软件透明地支持共享地址空间通信的抽象。在一个 4 处理器节点中，以硬件的高效性支持一致的共享存储器；而跨节点时，一致的共享存储器则是用软件以低得多的效率支持的。给定一组应用和感兴趣的问题规模，你将要写一个评价你的系统的研究报告。你可能希望进行什么样的性能比较来理解这个跨节点体系结构的有效性？你会设计什么样的实验？每一种类型的实验将告诉你什么？你将使用什么样的指标？你有 16 个基于总线的多处理器节点，每个节点上有 4 个处理器，总共有 64 个处理器。假设你使用问题约束的扩放，而且你已经选择了问题的规模。

4.9 如本章所讨论的，在实际研究体系结构折中时通常使用两种类型的模拟：轨迹驱动的和执行驱动的。在轨迹驱动的模拟和执行驱动的模拟之间的主要折中是什么？在什么条件下，你认为结果（比如，一个程序的执行时间）会显著不同。

4.10 考虑多处理器模拟的难度和精度。

1）考虑处理器、存储系统、网络、通信辅助部件、时延、带宽或者竞争，你认为系统哪个方面最难被精确地模拟？哪个方面相对来说容易模拟？在每一种情况下，主要的困难是什么？在这诸多方面，你认为哪一些最重要，应该非常精确地加以模拟？而哪一些可以牺牲一些模拟的精度？

2）当试图评价通信体系结构的折中的影响时，考虑适当地模拟处理器流水线的重要性。尽管很多现代的处理器是超标量的和动态调度的，一个单条指令发射的静态调度的处理器的模拟却要容易得多。假设你想要模拟的实际处理器主频为 200 MHz，具有双指令发射能力，但实现了每条指令 1.5 个周期（1.5 CPI）的完美存储器。你能否在一个致力于理解网络传送时延的改变对端性能的影响的研究中，把它模拟成一个具有 300 MH 主频、指令单发的处理器吗？主要要考虑的问题是什么？

4.11 考虑你所熟悉的在一个二维网格上的最近邻居网格计算的迭代，使用子块分割。假设我们使用四维数组的表示，其中前两维表示适当的分区，我们要评价的全规模问题是一个有双精度元素的 8 192 × 8 192 的网格，有 256 个处理器，每个处理器有 256 KB 的直接映射高速缓存，高速缓存块的尺寸为 128 字节。我们不能模拟这么大规模的问题，但是可以模拟一个 64 个处理器的 512 × 512 的网格问题。

264

1）你要选择的高速缓存的尺寸是什么？为什么这样选择？

2）列出选择太小的高速缓存会带来的危险。

3）你要选择什么样的高速缓存的块尺寸和相联性？会涉及到什么样的问题和禁忌？

4）在多大程度上你认为结果对于在更大机器上的全规模的问题有代表性？你会用这种配置去评价对某种通信体系结构的优化吗？可以评价该应用在机器上获得的加速比吗？

4.12 在像 Barnes-Hut 这样的星系模拟的科学应用中，影响扩放的关键问题是误差。这些应用经常模拟发生在自然界的物理现象，通过一些近似，用一个离散的模型来表示一个连续的现象并采用数值逼近的技术来解决问题。有几个应用参数代表了不同的近似的

源，因此也代表了模拟中的误差。比如，在 Barnes-Hut 中，粒子的数量 n 代表了对星系采样的精度（空间的离散化），时间步间隙 Δt 表示了时间离散化所做的近似，力计算精度参数 θ 决定了该计算的近似程度。运行较大问题的应用科学家的目标通常是在模拟中减少总体误差，并使它更加精确地反映被模拟的现象。尽管科学家调整不同的近似时没有通用的规则可循，但是对一个物理模拟来说，既具有直觉上的吸引力，又具广泛实践应用性的原则是：应该调整所有的误差源，这样它们的误差影响是相等的。

对于 Barnes-Hut 这样的星系模拟，星体物理学的研究（Hernquist 1987；Barnes 和 Hut 1989）表明，当某些误差影响不是完全独立时，下列规则在感兴趣的参数范围内是有效的：

- n：n 增加 s 倍会导致模拟误差降低$\sqrt{s}$倍。
- Δt：为了随时间而归并粒子轨道所使用的方法有一个 Δt^2 数量级的的全局误差。因此，误差降低$\sqrt{s}$倍（为了匹配 n 的 s 倍的增加）要求 Δt 降低$\sqrt[4]{s}$倍。这意味着要想让模拟持续保持不变的恒定量的物理时间，时间步数需要增加$\sqrt[4]{s}$倍。
- θ：在有实际意义的范围内，力的计算误差和 θ^2 成比例，因此，误差降低$\sqrt{s}$倍会要求 θ 降低$\sqrt{s}$倍。

265

假设在本习题中，某个规模的问题在 p 个处理器上的执行时间是在一个处理器上的执行时间的 $1/p$；也就是说，在问题约束扩放下，那个问题规模获得了完美的加速比。

1）如果 n 增加了 s 倍，你要如何调整其他的参数 θ 和 Δt？与只调整粒子的数量 n 的简单扩放对照，把这个规则称为真实扩放。

2）在串行程序中，数据集的尺寸和 n 成比例并和其他的参数无关。在共享地址空间中，在真实扩放和简单扩放下的存储器需求的增长是否不同（假定重要的工作集能容纳于高速缓存）？在消息传递型系统中会如何呢？

3）串行执行时间的增长大约是遵循

$$\frac{1}{\Delta t}\cdot\frac{1}{\theta^2}\cdot n\log n$$

（假设模拟固定量的物理时间）。如果 n 扩大 s 倍，假设完美加速比的情况，在真实扩放和简单扩放中，在 p 个处理器上的并行执行时间如何调整？

4）当处理器数量增长 k 倍，在 MC 扩放模型下，采用简单和真实两种扩放时，并行执行时间如何增长？如果在基本的机器上（未扩大规模之前）运行该问题需要一天，对于简单模型和真实模型下，在扩大的机器上运行该问题需要多长时间？

5）对于简单模型和真实模型，当处理器数目增加 k 倍，在共享地址空间中可以模拟的粒子数量在 TC 扩放下如何增长？

6）对于这个应用哪种扩放模型更加实际？MC 还是 TC？

4.13 对于 Barnes-Hut 这个例子，在真实和简单的 MC、TC 和 PC 扩放模型下，如何扩放下面的执行特征？

1）通信与计算的比率。首先假设它仅仅与 n 和处理器数量 p 有关，按照$\sqrt{p}/\sqrt{n}$变化，请在不同的模型下粗略的画出这个比值随处理器数量增长速率的曲线。然后说明

在不同的扩放模型下，其他参数可能产生的影响。

2）在共享地址空间的机器上为了获得好的性能所需要的不同工作集的规模和与此相适应的高速缓存的尺寸，请粗略地描绘在不同模型下，随着处理器数量的增加，最重要工作集的增长速率曲线。它在扩放方面补充了什么主要的方法上的结论？评价一下这些趋势和在消息传递型系统中局部基本树所需要的本地复制量的趋势有什么不同。

3）同步的频率（比如说每个单元计算一次），包括加锁和栅障两者。至少定量地加以描述。

266

4）输入/输出操作的平均频率和规模，假设每个处理器打印出所有分配给它的星体的位置 i）每十个时间步打印一次；ii）每隔固定量的模拟物理时间（比如，在星系演化中模拟时间中的每一年）。

5）在一致共享地址空间中的力的计算中，可能同时共享（访问）一个给定的星体数据的处理器的数量。（正如我们将在第 8 章中看到的，这个信息在为可扩展的共享地址空间的机器设计高速缓存一致性协议时有用处。）

6）在显式的消息传递型实现中消息的频率和尺寸。集中在力计算所需要的通信，假设每个处理器向其他每个处理器仅仅发送一条消息，该消息传送了后者在该时间步计算它的力需要从前者获得的数据。

4.14 基数分类应用需要并行的前序计算，从局部直方图计算出全局直方图。该计算的一个简化的版本如下。假设 p 个进程中的每一个进程有一个已经计算好的局部值（把这看作表示那个进程的直方图中对应于给定数位值的键字的数量）。我们的目标是计算一个有 p 个项的数组，项 i 是从进程 0 到进程 $i-1$ 所有局部值的总和。

1）描述并实现计算这个输出数组的最简单的线性方法。

2）现在设计一个具有较短关键路径的并行方法。（提示：你可以使用树结构。）分析每个方法所需要的时间。实现这两个方法，比较它们在你所选择的机器上的性能。在这里可以使用简化的例子或者更完整一些的例子，“本地值”实际上是一个数组，每个项是基数的一个数位，输出数组是二维的、由进程标识符和基数数位索引的数组。也就是说，更完整一些的例子对于每个基数数位都进行计算，而不是只计算一个基数数位。

267

3）讨论在后一个方法中你能够使用的协调同步的不同方法以及它们之间的折中。

第 5 章　共享存储的多处理器

并行体系结构最常见的形式是中小规模的多处理器，它提供全局物理地址空间，可以从任何处理器对等地访问全部主存。这种体系结构通常称为对称多处理器（SMP）。每个处理器均有自己的高速缓存，所有的处理器和存储模块均连接在同一个互连设备上，此互连设备通常是一共享总线。SMP 机器在服务器市场上占据着主导地位，在桌面计算机市场上也日益流行，同时它们还是构成大规模系统的重要组件。由于对存储器和处理器等资源的有效共享，这些 SMP 机器有很强的吞吐能力，在同时处理多个对存储器与 CPU 需求各异的串行作业时表现不凡。这种机器的各个处理器可以使用普通的装入和存储命令高效地访问共享数据，还可以自动地在各自的高速缓存中移动和复制共享数据。这些特性对并行程序设计颇具吸引力。另外，这些特性对操作系统也相当有用，使操作系统的不同进程可以共享数据结构，但在不同的处理器上运行。

从图 5-1 所示的通信体系结构的层次观点来看，这种共享地址空间的程序设计模型可以直接由硬件支持。用户进程可以用共享虚拟地址进行读写操作。这些操作通过对共享物理地址的装入和存储指令来实现。事实上，这里的程序设计模型和硬件操作之间的关系是如此密切，以至于它们通常都被简单地称为“共享存储”。一个消息传递的程序设计模型可由一个软件层来实现——典型的例子是某个运行库。这一软件层将共享地址空间的大部分划分给每个进程，当作它们的私有地址空间来处理，而将另外一些共享地址空间按照进程间的消息缓冲区进行管理。通过在这些缓冲区中做数据拷贝的方式来实现发送和接收操作。由于共享缓冲区地址的翻译和保护由硬件提供，这部分工作无须涉及操作系统。出于可移植性的考虑，大部分消息传递程序设计接口均在常见的 SMP 上得到实现。事实上，只要对共享总线和存储的争用不成为瓶颈，这种实现对消息传递程序带来的性能往往优于传统的分布式存储消息传递系统。这在很大程度上是由于没有涉及到操作系统。操作系统仍然用于输入/输出和对多道程序设计的支持。

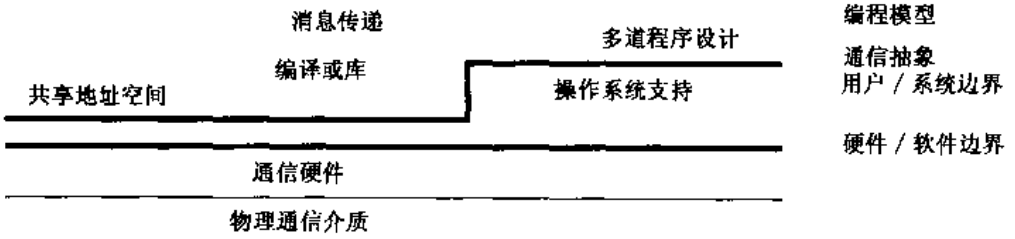


图 5-1　基于总线的 SMP 系统的通信体系结构抽象层次。共享地址空间由硬件直接支持，而消息传递由软件支持

由于所有的通信操作和本地计算均产生对共享地址空间中的存储访问，从系统设计者的观点来看，高层设计的关键问题在于对这种扩展存储层次的组织。通常多处理器的存储层次分为 4 类，如图 5-2 所示，这 4 类层次与可考虑到的多处理器规模大致对应。前 3 个对应的都是对称式多处理器（所有处理器到所有主存空间的访问时间都一样），而第 4 个不同。

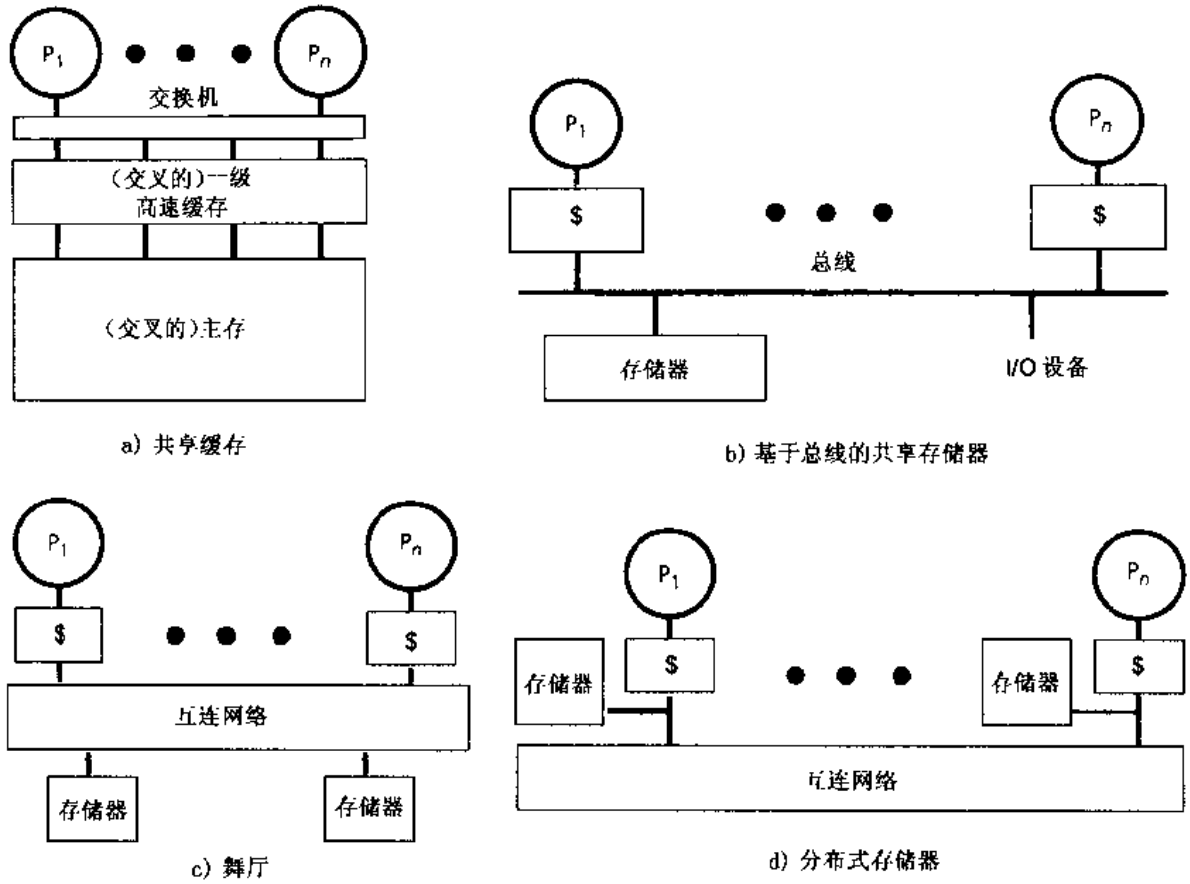


图 5-2 多处理器中常见的扩展存储层次结构

在共享高速缓存方式下（如图 5-2a 所示），互连设备位于处理器和共享的一级高速缓存之间，而此高速缓存连接到共享主存子系统上。高速缓存和主存系统都可以是按交叉方式组织的，从而可以增加有效带宽。这种方法适用于处理器数目较少的情况（2～8 个）。在 20 世纪 80 年代中期，在一块电路板上安放一对处理器是常见的。而现在，这可能在单片多处理器上实现，即几个处理器在同一块芯片上，共享芯片上的一级高速缓存。然而，这种策略仅适于很小的规模，因为处理器和它们共享的一级高速缓存之间的互连机构是决定高速缓存访问延迟的关键因素，还由于该高速缓存必须为多个处理器的同时访问提供很高的带宽。

在基于总线的共享存储方式下（如图 5-2b 所示），互连设备是共享总线，它位于处理器的私有高速缓存（或高速缓存的某种层次结构）和共享主存子系统之间。这种方式广泛用于中小规模的多处理器之上，典型的中小规模多处理器通常包括 20 或 30 个处理器。目前出售的并行计算机主要是这种形式。在现代的多处理器设计中，投入的大量设计工作都是为了支持“高速缓存一致性”的共享存储系统。例如，Intel 的 Pentium Pro 处理器可以直接挂在共享总线上，无须任何“粘接逻辑”。低成本的基于总线的机器常常使用这些处理器，极大地推广了基于总线的共享存储方式的使用。这种机器的扩展限制主要来自于共享总线和存储系统的带宽限制。

最后的两种方式可扩展到许多处理节点。“舞厅”（dancehall）方式也将互连设备放在高速缓存与主存之间，但这时的互连设备不一定只是总线，它可以是可扩展的端到端互连网络，同时存储器划分为许多逻辑模块，这些模块连接到互连设备中逻辑上不同的位置（如图

5-2c 所示)，这种方式是对称的，即所有主存模块到所有处理器的距离相同。但这种方式也有局限性，即存储器到所有处理器的距离太远。特别是在规模较大的系统中，为了使得所有的处理器能访问所有的存储模块，必须经过互连网络中的若干“跳步”。第 4 种方式是分布存储方式，它不是对称性的。可扩展的互连设备位于处理节点之间，但每个节点均在全局存储中拥有自己的一部分本地存储，对这部分主存的访问速度更快（如图 5-2d 所示）。通过在分布的数据中发掘局部性，大部分高速缓存失效可以在本地存储中满足，无须跨越网络。这种设计对可扩展的多处理器最具吸引力。本书中后面的若干章节会集中于这一主题。当然，也可以将上述方式组合在一种机器设计中——例如，设计一台分布式存储的机器，其节点是基于总线的 SMP；或设计一台机器，其处理器共享的高速缓存不是一级高速缓存。

在种种情况下，高速缓存总是扮演着重要角色，对处理器而言，高速缓存可以减少平均数据访问时间，对共享的互连设备和存储系统而言，它可以减少每个处理器需要的通信带宽。带宽需求的减少是因为某个处理器提出的数据访问要求可在高速缓存中得到满足，无须使用互连设备。除了共享高速缓存方式之外，其他方式下每个处理器至少有一级私有的高速缓存，这带来了一个非常关键的问题——*高速缓存一致性问题*。当同一个存储块的副本出现在一个或多个处理器的高速缓存中时，就会产生这样一个问题：如果某个处理器写入并因而修改了此存储块，那么除非采用特殊的操作，否则其他处理器在访问此存储块时，实际上访问的还是自己高速缓存中保存的旧内容。

269 ~ 271

当前，大多数小规模的多处理器使用共享总线互连设备，每个处理器有自己的高速缓存，同时这些处理器共享集中式主存；而舞厅和共享高速缓存方式只是用在特定的条件下。随着技术的演化，具体的系统组织方式可能会发生变化。然而，基于总线的组织方式和分布式存储组织方式是当前最流行的方式，它们体现了解决高速缓存一致性问题的两种基本方法。根据互连设备本质特性的不同，一种方法适用的情况是：互连设备上的操作对所有的处理器均是可见的（类似总线）；在另一种情况下，互连设备不是集中式的，端到端的操作仅对于其参与者可见。本章主要讨论协议的逻辑设计，这种协议应能充分利用总线的基本属性解决高速缓存一致性的问题。与这些高速缓存一致性技术的硬件实现相关的设计问题在下一章介绍。可扩展的分布式存储多处理器的基本设计在第 7 章介绍，随后的第 8、9 章涵盖了特定于可扩展高速缓存一致性的问题。

5.1 节详细地描述了共享存储体系结构的高速缓存一致性问题，并给出一个最简单的例子，称为*侦听式*（snooping）高速缓存一致性协议。一致性不仅是一个关键性的硬件设计概念，而且是存储抽象的直观概念中不可或缺的部分。然而在存储行为方面，并行软件常常需要依赖比这种一致性更强的假设。5.2 节对第 1 章提出的序的概念作进一步讨论，引入存储同一性（consistency）的概念，这一概念定义了共享地址空间的语义。在计算机体系结构和编译器设计中，这一问题越来越重要；近来大多数指令集系统结构的参考手册中，有很大篇幅在介绍存储同一性模型。在有了存储抽象和相关的概念之后，5.3 节详细介绍了一些更实用的侦听协议，同时向读者展示这些协议如何满足高速缓存一致性条件和有用的存储同一性模型。协议的操作通过在逻辑状态变迁层次得以描述。对这一层次上的几个设计方面的权衡考虑有一个量化的评估，使用的评估技术在 5.4 节中介绍，这一技术涉及到第 4 章中谈到的工作负载驱动评估方法的若干要素。

本章后面的部分探讨具有高速缓存一致性的共享存储系统对于在其上面运行的软件的影

272 响。5.5 节讨论低层的同步操作如何利用具有高速缓存一致性的多处理器上可用的硬件原语以及如何改造锁和栅障算法，使机器能更好地得到利用。5.6 节讨论并行程序设计的影响，特别讨论了如何利用数据的时间和空间局部性，来减少高速缓存扑空和共享总线上的通信量。

5.1 高速缓存的一致性

直觉上，存储行为的模型应该是这样的：它提供一组保存数值的单元，当读取某个单元时，应返回写入此单元的最新值。这是存储系统抽象的基本属性。在串行程序中我们依赖于这一属性，利用存储器将从程序中某个位置产生的值传递到其他使用此值的地方。当使用共享地址空间，在运行于一个处理器上的线程或进程之间进行数据通信时，我们同样依赖于存储系统的这一属性。读操作返回的是最新写入所读取单元的值，与此值由哪个进程写入无关。由于所有的进程都通过相同的高速缓存层次访问存储器，不会有诸如高速缓存一致性问题的发生。当两个进程运行于共享同一存储系统的不同处理器上时，我们也希望能依赖于这样的存储属性。也就是说，含有多个进程的程序在物理上不同的处理器上运行，或者在同一个处理器上（以交叉方式或以多道程序方式）运行时，结果不应该有区别。然而，当两个进程通过不同的高速缓存访问存储器时会存在这样的危险：一个进程在其高速缓存中看到的是新值，而另一个看到的还是旧值。

5.1.1 高速缓存一致性问题

多处理器中的高速缓存一致性问题是普遍存在的，而且对机器性能有重大影响。例 5.1 描述是这方面的一个情况。

例 5.1 图 5-3 显示了 3 个带有高速缓存的处理器，经总线连接到共享主存。处理器有一个对单元 u 的访问序列。首先，处理器 P_1 从主存中读取 u，将 u 值的副本放在其高速缓存中。然后处理器 P_3 从主存中读取 u，将 u 值的副本放在自己的高速缓存中；然后 P_3 对 u

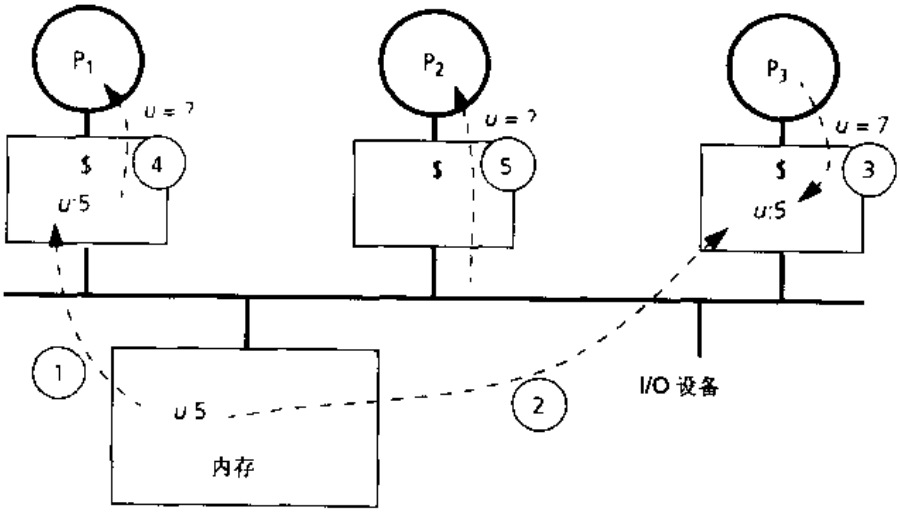


图 5-3 高速缓存一致性例子。此图显示 3 个带有高速缓存的处理器，经总线连接到共享主存。u 是存储器中的一个单元，这些处理器对其进行读写操作。读写操作的顺序由表现操作的弧上所标的数字给出。容易看到，当 P_3 将 u 的值由 5 改为 7 时，如果不采用什么特别的措施，P_1 从自己的高速缓存中读的 u 值是过时的，而 P_2 从主存中读的值也是旧的

写入，将其中的值由5改为7。在直写式高速缓存中，这会导致主存中相应位置的更新；然而，当P_1再次读取u时（第4个操作），不幸的是，它从自己高速缓存中读得旧值5，而不是从主存中读得新值7。这就是高速缓存一致性的问题。在回写式高速缓存中情况又如何呢？

解答：在回写式高速缓存中情况更糟糕。P_3的写操作仅仅在保存单元u的高速缓存块的相关标志位作了标记，而不会立即修改主存。仅在此高速缓存块从P_3的高速缓存中替换出来时，其内容才会写回主存中。因此，不仅P_1读的是旧值，而且当P_2读u时（第5个操作），由于在高速缓存扑空它会从主存中读取，但这时读取的也是旧值5，而不是新值7。最后，如果多个处理器在其回写式高速缓存中将不同的值写入单元u时，最终到达主存的值由包含u的高速缓存块的替换顺序决定，而与各处理器对u进行写操作的顺序无关。■

显然，例5.1描述的行为与我们对存储系统应有行为的认识相抵触。事实上，即使在单处理器上，当执行I/O操作时也会产生高速缓存一致性问题。大部分I/O传输由直接存储访问（DMA）设备完成，DMA设备在存储器与外设之间移动数据不会涉及处理器。当DMA设备写入主存的某个单元时，除非采取特殊的操作，否则，如果此单元先前在某个处理器的高速缓存中，则此处理器看到的仍然是旧值。在回写式高速缓存中，DMA设备可能会从主存中读取某个单元的旧值，因为最新的值可能还在处理器的高速缓存中。由于I/O操作远不如存储操作频繁，在单处理器系统中采用了某些粗粒度的解决方案。例如，存储空间中用于I/O的段可以标记为“不可用于高速缓存”（即这段存储空间的内容不能进入处理器的高速缓存），或者在访问用于与I/O设备通信的存储段时，处理器总使用不通过高速缓存的载入和存储操作。对于每次传输较大数据块的I/O设备，例如硬盘，操作系统会提供支持以确保数据的一致性。在许多系统中，在I/O操作进行之前，数据传输涉及到的存储页会由操作系统根据处理器的高速缓存内容进行刷新。在另一些系统中，所有的I/O操作均会经过处理器的高速缓存层次，由此来维护一致性。当然，这种策略有其消极影响，因为处理器不会马上用到的数据也可能被记录在高速缓存中了。幸运的是，用于解决多处理器高速缓存一致性的技术和支持同样可以解决I/O一致性问题。现在，几乎所有的多处理器均提供对高速缓存一致性支持。

273
274

在多处理器中，不同进程读取和写入共享变量是经常性事件，因为这是并行应用程序的多个进程之间通信的基本方法。因此，我们不希望禁止在高速缓存中存有共享数据，也不希望在引用所有共享变量时必须调用操作系统的相应操作。这时，高速缓存一致性需要作为一项基本的硬件设计任务来完成；例如，当某个共享单元被修改时，此位置在高速缓存中的旧副本（如例5.1中P_1高速缓存里u的副本）必须淘汰，要么使此值无效，要么用新值更新。事实上，操作系统自身也会从这种透明的、硬件支持的数据结构一致性中获益匪浅。

在探索可提供一致性的技术之前，有必要更精确地定义一致性的概念。我们的直观概念“每个读操作必须返回最后写入此位置的值”在并行体系结构中有些问题，因为“最后”一词没有很好地定义。两个不同的处理器可能在同一时刻对同一存储单元进行写操作；或者对于同一个存储单元，一个处理器可能在另一个处理器刚刚发出写操作后马上发出读操作，但由于光速或其他什么因素的影响，写者的更新根本来不及传播到读者。[○]即使在顺序程序的

○ 这里强调从处理器看到的操作的发出时间，而不是完成时间，见后面关于操作发出的定义。——译者注

情况下，“最后”一词也不是时序或物理概念，它指的是程序定义的操作次序中的“最后”。现在，我们可以暂时将进程所对应的程序操作次序看作是在机器语言程序中存储操作发生的次序。在5.2节中，关于程序操作序这一概念有进一步的详尽阐述。在并行情况下，问题在于程序操作序是由每个独立进程中的操作定义的，要定义一致性存储系统的语义，我们需要先形成关于一组程序操作序的概念。

我们首先回顾一下单处理器存储系统环境中某些术语的定义。对于存储操作，我们指的是一次单独的读取（载入）、写入（存储），或对存储单元的“读-改-写”访问。对于完成多个读取和写入操作的指令，例如那些出现在许多复杂指令集中的复杂指令，可以将其分解为多个存储操作。指令中的这些存储操作的执行次序由此指令确定。指令中的这些存储操作可看作以彼此相关的指定次序自动执行；也就是说，一个操作的各种影响应发生在下一操作的任何影响之前。称一个存储操作被发出，指的是它离开了处理器的内部环境并呈现给存储系统之时，这里提到的存储系统包括高速缓存、写缓冲区、总线和存储模块。关于序的一个十分重要的问题是，处理器观察存储系统状态的惟一方法是通过发出存储操作（例如，读操作）；这样，我们称一个存储操作相对于这个处理器执行了，只要它能够感受到所发出的存储操作的效果。特别地，我们若称“一个写操作相对于这个处理器执行了”，则意味着该处理器后续的读操作能够返回由此写操作产生的值，或者返回由此写操作之后的另一个写操作产生的值。称“一个读操作相对于这个处理器执行了”，意味着该处理器后续的写入操作不会影响此读操作的返回值。注意，在这两种情况下我们都没有指明要访问存储芯片上的具体物理单元，也没有具体要求硬件上的哪些部分改变它们的状态。此外我们注意到，在串行情况下“后续”一词的含义是明确的，因为读和写操作的次序是由程序中语句的次序决定的。

275

关于存储操作“发出”和“相对于一个处理器执行了”的定义也可应用于并行情况；我们只需将定义中的“这个处理器”替换为“某个处理器”。问题在于，“后续”和“最后”这两个术语的含义出了一些问题，因为我们此时没有一个程序操作序：每个进程有自己的程序操作序，在访问存储系统时这些程序操作序会交织在一起。要澄清我们对一致性存储系统的概念，一种方法是设想在没有高速缓存的单一共享存储环境下会发生什么情况。对存储单元的每个写入和读取操作均会访问主存中的物理位置。这时的操作“相对于所有的处理器都执行了”，于是被称为完成了。因此，存储器就可以在所有处理器对一个单元发出的读写操作上强加一个序。进一步，在这个整体序中，来自某个处理器的读写操作应该遵从自己的程序操作序。那么，在这种情况下，主存单元就是一个硬件上的自然参照点，可确定跨处理器访问此单元的操作顺序。没有理由认为不同处理器必须按照特定的方式交替访问存储系统，因此任何交替访问均是合理的，只要此访问序列能够体现每个处理器的程序操作序即可。不过我们要假定一些基本的公平性原则，即最终每个处理器的操作都应该完成。我们的直观概念“最后的”可看作是在保持这些性质的一个假想串行序中最近的元素，“后续的”可以类似地定义。由于这一串行序必须保持一致性，所有处理器看到对某个单元的写操作应该是相同序的（如果它们愿意看的话，即读这一单元的值），这一点很重要。

对每个单元的操作都有一个总体的、串行的操作序是我们希望在任何一致性存储系统中见到的。当然，总体的序无须在执行程序时真正建立。特别是在具有高速缓存的系统中，我们不希望主存看到所有的存储操作，希望尽量避免串行化。我们仅希望程序行为就好像某种序存在一样。

使用更加形式化的方法，我们说多处理器存储系统是一致的，如果某个程序的任何执行结果都满足下列条件：对于任何单元，有可能建立一个假想的操作序列（也就是说，将所有进程发出的读写操作排成一个全序），此序列与执行结果一致，并且在此序列中：

1）任何特定进程发出的操作所表现出来的序和该进程向存储系统发出它们的序相同，并且

276

2）每个读操作返回的值是对相应单元按串行顺序写入的最后一个值。

关于一致性的这一定义中隐含了两个性质：写传播意味着写操作的效果对其他进程可见；写串行化意味着对于某个单元的所有写操作（来自相同的或不同的进程）而言，所有进程都是以相同顺序看到这些操作。例如，写串行化意味着，如果 P_1 进程对某个单元的读操作先看到由写操作 $w1$（由 P_2 完成）产生的值，再看到由 $w2$（P_3 中的操作）产生的值，那么 P_4（也可以是 P_2 或 P_3）的读操作看到的应该是同样的顺序。没有必要将写串行化的概念推广到读操作上，因为读操作的结果只有发出此操作的进程可见。

程序的结果可以看作是程序中读操作的返回值，也许可能增广到程序结束时对所有单元的一组隐含的读操作。从程序结果中，我们无法确定机器执行这些操作的真正次序，也无法精确获知数据位是何时更改的，只能知道程序外在的执行序。幸运的是，这也就够了，因为这就是处理器能够得到的所有信息。当讨论存储同一性模型时，这一概念会显得更加重要。

5.1.2 通过总线侦听的高速缓存一致性

定义过存储一致性的性质后，让我们来考察解决高速缓存一致性问题的技术。例如在图 5-3 中，我们如何保证 P_1 和 P_2 能够看到 P_3 写的值？实际上，高速缓存一致性的一种非常简单而漂亮的解决方法可以从总线的根本性质中得到。总线是连接几台设备的单独一组导线，其中每一设备都可以观察到每一个总线事务，诸如在共享总线上的每一次读或写操作。当一个处理器向它的高速缓存发出请求时，高速缓存控制器检查其高速缓存的当前状态并采取适当的措施，这可以包括产生总线事务以访问存储器。通过让所有的高速缓存控制器“侦听”总线并监管其上发生的事务，一致性得到维持，如图 5-4 所示（Goodman 1983）。如果一个总线事务和某个高速缓存控制器相关的话，例如在该高速缓存中含有与该总线事务相关的一个

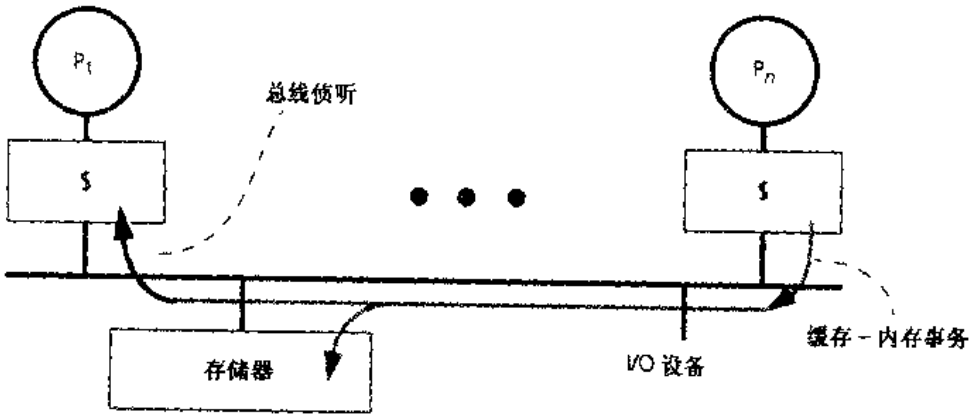


图 5-4 一个侦听高速缓存一致性的多处理器。带有各自私有高速缓存的多个处理器连接到一个共享总线上。每个处理器的高速缓存控制器时刻在总线上“侦听”，观察和它相关的总线事务并相应地更新其状态以保持高速缓存的一致性。灰色箭头表示总线上出现的一个事务并被主存接受，这和单处理器系统一样。黑色箭头指示的是“侦听”

存储块的拷贝，该控制器可能采取行动。这样的话，如果 P_1 看到了 P_3 的写操作，它可能采取行动，比如使这个拷贝无效或使它更新。实际上，由于高速缓存中数据的分放和替换是以高速缓存块（通常为几个字长）的粒度进行管理的，并且访问高速缓存扑空时要从存储器获取一整个高速缓存块的数据，因此通常高速缓存一致性也是以高速缓存块的粒度来维持的。换句话说，在高速缓存中，要么整个高速缓存块处于有效态，或者整个高速缓存块都处于无效态。因此，高速缓存块是高速缓存中数据分配的粒度，是高速缓存间数据传输的粒度，也是缓存一致性管理的粒度。

下面我们看支持一致性的总线的最关键性质。首先，总线上出现的所有事务对高速缓存控制器来说是可见的。其次，它们对所有控制器以同样的顺序（也就是它们出现在总线上的顺序）可见。一个一致性协议必须保证为响应存储操作的所有“必要的”事务都真正出现在总线上，并且当控制器发现一个相关的事务时将做出适当的动作。

保持一致性最简单的例子是一个含有单级直写高速缓存的系统。这基本上就是在 20 世纪 80 年代中期那些最初的基于总线的商用 SMP 所遵从的方法。在那样的系统中，每一个写操作引发总线上的一个写事务，于是每一个高速缓存控制器将观察到每一次写操作（于是实现了写传播）。如果某个侦听高速缓存有这个块的一个拷贝，它或者使这个拷贝无效，或者更新该拷贝。前者被称为基于作废的协议，后者被称为基于更新的协议。不管何种情况，任何处理器随后访问这一块数据时它将看到最新的值，或者通过一次缓存扑空转而从主存得到，或者因为被更新的值在它自己的高速缓存中。主存中含有的总是有效的数据，于是当发现总线上的一个读操作时，高速缓存不需要采取任何行动。例 5.2 显示了图 5-3 中的一致性问题是如何被直写缓存解决的。

例 5.2　考虑图 5-3 中的情形。假设为直写缓存，试说明使用基于作废的协议时，总线怎样可以被用来提供一致性。

解答：当处理器 P_3 把 7 写到 u 单元时，P_3 的高速缓存控制器生成一个总线过程以更新存储器的内容。P_1 的高速缓存控制器发现此总线事务是和它相关的并且是一个写事务，于是就作废它其中含有 u 的数据块。主存控制器将 u 的值更新为 7。以后从处理器 P_1 和 P_2 发出的读 u 的请求（动作 4 和 5）均将在它们私有的高速缓存中扑空，并将从主存获得正确的值 7。■

277 ~ 278

确定一个总线事务是否与某个高速缓存相关从本质上说和对从处理器发出的一个请求进行标识匹配是完全一样的。采取的动作可以是作废或更新那块高速缓存块的内容或状态并且（或者）将那个高速缓存块的最新值从高速缓存中传送到总线上。

侦听式高速缓存一致性协议将在单处理器中也有体现的计算机体系结构的两个基本方面结合到了一起：总线事务和与高速缓存块相关的状态转换图。前面提过，一个总线事务由三个阶段组成：仲裁、命令/地址、数据。在仲裁阶段，想发起一次事务的设备发出它们的总线请求，总线仲裁器从中选择一个并且给出相应的准许信号作为响应。收到准许信号后，被选中的设备将命令，比如读或写以及相关的信号放到总线的命令和地址线上，所有的设备都会看到这个地址。在一个单处理器中，其中一个设备知道由它对这个特别的地址负责。对一个读事务，地址阶段接下来就是数据传输。根据数据是在地址阶段期间传输的还是在地址阶段之后传输的，写事务对不同的总线将有所不同。对大多数总线来说，一个响应设备能够发出并维持一个等待信号直到数据就绪为止。这个等待信号不同于其他的总线信号，因为它是

在各个处理器之间线“或”（wired-OR）的，即任何设备发出该信号都得到逻辑1。发起者不需要知道哪一个响应设备正在参与数据传输，而只需知道有这样一个设备并且它是否就绪即可。

计算机体系结构的第二个基本方面是，每一个单处理器高速缓存中的数据块，除了标记和数据外，还有一个状态和它联系，用来表明该数据块的特征（例如，无效、有效、脏）。高速缓存的策略由*高速缓存块的状态转换图*来定义，它是一个指示数据块的特征如何改变的有限状态自动机。高速缓存块的转变发生在访问该数据块或者访问一个映射到和此数据块相同的高速缓存线的一个地址上。（我们用高速缓存块来表示实际数据，用缓存线来表示在高速缓存硬件中固定的存储位置，这和主存中的页和页帧区别是类似的。）虽然只有实际存在于高速缓存线中的数据块才具有硬件状态信息，逻辑上，所有不在缓存中的数据块均可以被看作或者处于一个特殊的“不存在”状态，或者处于“无效态”。在一个单处理器系统中，对一个直写、写不分配的高速缓存（Hennessy and Patterson 1996）仅仅需要两个状态：有效态和无效态。起初，所有的数据块都是无效的。当一个处理器对高速缓存的读操作扑空时，系统要生成一个总线事务来从存储器中调入该块，该块被标记为有效。写操作生成一个总线事务来更新存储器，如果高速缓存块处于有效态，它们也更新该块。写操作不改变数据块的状态。如果一个数据块被替换了，它可以被标记为无效直到存储器提供新的数据块使它成为有效的。一个回写缓存的每一条缓存线需要一个额外的状态，用来标记一个“脏”的或者说已被修改的数据块。

在一个多处理器系统中，一个数据块在每个高速缓存中都有一个状态，这些高速缓存状态根据状态转换图而改变。这样，我们可以把一个数据块的缓存状态看作一个由 p 个状态而不是单独一个状态组成的向量，这里 p 是高速缓存的个数。高速缓存状态由 p 个分离的自动状态机来控制，由高速缓存控制器来实施。控制状态转换的状态机或状态转换图对所有的数据块和所有的高速缓存是完全相同的，但不同高速缓存中数据块的当前状态是不同的。像以前一样，如果一个数据块在高速缓存中不存在，我们可以假设它处于一个特殊的“不存在”状态或无效态。 279

在一个侦听式高速缓存一致性模式中，每个高速缓存控制器接收两套输入：由处理器发出的存储器请求以及由总线侦听器通知关于从其他高速缓存发起的总线事务。不管对应哪一种，根据当前状态和状态转换图，控制器都可以更新高速缓存中相关块的状态。控制器也可能采取一个行动。例如，它以被请求的数据来响应处理器的要求，潜在地生成新的总线事务以获得数据。它通过更新自己的状态来响应总线事务，有时候也干预事务的完成。这样来说，侦听式协议是一种分布式算法，由一组相互作用的有限状态自动机来表示。它由下面的部分确定：

- 和存在于局部高速缓存的存储块相联系的一组状态。
- 状态转换图，它以当前状态和处理器请求或观察到的总线事务为输入，并产生缓存块的下一状态作为输出。
- 和每一次状态转换相关的动作，它们部分是由总线、高速缓存和处理器的设计所定义的一组动作决定的。

一个数据块的不同状态自动机由总线事务来协调。

在图5-5中，状态转换图描述了一个简单的基于作废的协议，它针对是一致的直写、写

不分配的高速缓存。像在单处理器的例子中一样，每一个缓存块仅有两个状态：无效态（I）和有效态（V）（假设“不存在”态和无效态是一样的）。转换由引起转换的输入和由转换产生的输出来标记。例如，当一个控制器看到处理器传来的一个读操作在缓存中扑空时，就生成一个 BusRd 事务，在此事务完成时，该数据块转换为有效态。每当控制器看到处理器对某单元的一个写操作时，就生成一个总线事务来更新主存中的那个单元，但并不改变状态。对单处理器状态图的关键扩充在于，当一个总线侦听器看到总线上有一个对自己局部高速缓存的存储块的写过程时，控制器将那个数据块的状态置为无效，从而有效的废除了它的拷贝。（图 5-5 中用虚线显示此总线引起的事务）。以此扩展，如果任何处理器生成一个对缓存在其他所有处理器的数据块的写操作，那些处理器将使它们的拷贝无效。这样来说，一个数据块的多个同时出现的读操作可以共存而不会生成总线事务或无效，但写操作将废除所有其他被缓存的拷贝。

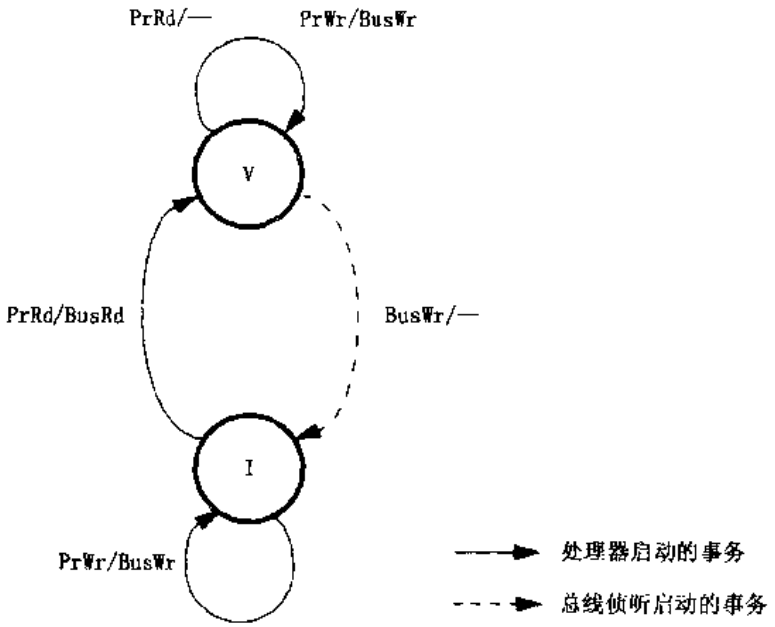


图 5-5　带有直写、写不分配高速缓存的多处理器的侦听一致性。直观上看，有两种状态，有效态（V）和无效态（I）。*A*/*B*（例如，PrRd/BusRd）表示如果 *A* 被观察到，那么将生成事务 *B*。从处理器的角度看，请求可以是读（PrRd）或写（PrWr）。从总线的角度来看，高速缓存控制器可以观察/生成总线读（BusRd）或总线写（BusWr）的事务

为看到这种简单的直写作废协议是如何提供一致性的，我们需要说明在此协议下程序的任何一次执行所含对一个单元的存储操作，都可以构造一个满足程序语句序和写串行化条件的完全序列。就目前的讨论，让我们假设总线事务和存储器操作都是原子的。也就是说，总线上一次只有一个事务在进行：一旦一个请求被放到总线上，此总线过程的所有阶段（包括数据响应），都在任何从其他处理器发出的请求被允许放到总线上之前完成（这种过程具有原子性的总线被称为*原子总线*）。同样，处理器要等待前面的存储操作完成才能发出下一个存储操作。对单级高速缓存来说，我们也很自然的假设作废被应用于高速缓存，从而写操作本身在总线事务期间完成。（本章将保持这些假设，但在第 6 章，当我们更细致地来看协议的实现和用更高的并行性来研究高性能设计时，这些假设将被放宽。）最后，我们可以假设存储器处理写和读操作的顺序和它们被提交到总线上的顺序一致。

在直写协议中，所有的写操作都在总线上出现。因为一次只进行一个总线事务；不管在哪次执行中，所有的对某一单元的写操作都按它们出现在共享总线上的顺序串行化（一致地），该顺序被称为总线顺序。因为每个侦听高速缓存控制器是在总线事务期间执行作废操作，于是作废操作是按总线顺序由高速缓存控制器来执行的。

280 ~ 281

处理器通过读操作“看到”写操作，于是对写串行化我们必须保证从所有处理器传来的读操作以串行化的顺序看到这些写操作。然而，对一个单元的读并不是完全串行化的，因为命中的读操作可以在它们的高速缓存中被独立地、并行地执行而不会生成总线事务。为了显示读操作如何可以被插入写操作的串行序列，考虑下面的情况。一个到达总线上的读操作（读扑空）由总线和写操作一起来进行串行化；因此它将获得按总线顺序最近写到相应单元的值。不出现在总线上的仅有的存储器操作是读命中操作。在这个例子中，所读的值是由同一处理器上对此单元最近的一次写操作或者通过它的最近的读扑空（以程序顺序）放到高速缓存中的。因为这两种值的来源都出现在总线上，读命中操作同样也看到在一致的总线顺序中产生的值。这样看来，在此协议下，总线顺序和程序顺序一起提供了足够的约束以满足一致性要求。

更一般地，我们可以通过观察下面的由协议强加的偏序来构造一个（假想的）满足一致性的全序：

- 如果操作是由同一个处理器发出的并且在程序顺序中存储器操作 M_2 在存储器操作 M_1 之后，则 M_2 是 M_1 的后继。
- 如果读操作生成一个过程在写操作 W 生成的事务之后，则该读操作是 W 的后继。
- 如果一个读或写操作 M 生成一个总线事务并且一个写操作的总线事务在 M 的总线事务之后，则该写操作是 M 的后继。
- 如果一个读操作并不生成总线事务（是一次命中）且并没有被另一个总线事务将其与写操作分离，该写操作是该读操作的后继。

任何保持这种偏序的串行顺序都是一致的。“后继”序关系是可传递的。在图 5-6 中勾画了这种偏序的一个例子，其中和写操作相关的总线事务分隔各自的程序顺序。虽然总线可能建立一个特别的顺序，但偏序并不限制从不同处理器发出的、发生在两个写过程之间的读操作总线事务的次序。实际上，只要遵循程序的顺序，此分隔中两个写操作之间的任何交错的读操作都是一个有效的串行顺序。

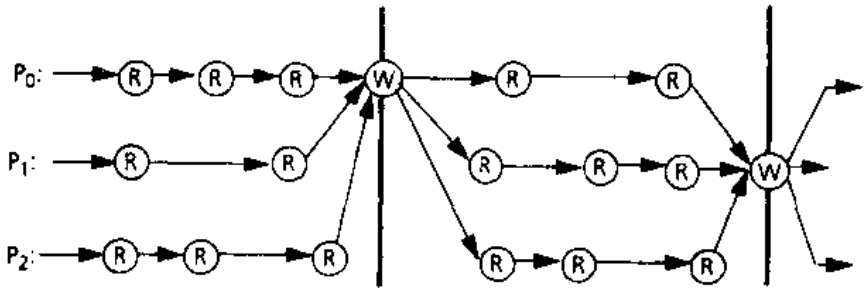


图 5-6　按照直写作废协议的一次执行的存储操作偏序图。写总线事务定义了一个全局事件序列，其间各个处理器按照程序操作序执行读操作。在保证各自的序的前提下，处理器之间操作的交织得到的任何全序是和该执行结果一致的

当然，这种简单的直写方法的问题在于每一个存放指令都要发到存储器，这就是大多数

微处理器使用回写高速缓存的原因（至少在接近总线的这一层）。此问题在多处理器的环境下被加重了，因为从每一个处理器发出的每一个存储命令消耗了珍贵的共享总线的带宽，从而造成可扩展性不高，正如例 5.3 所示。

例 5.3　考虑一个以 200 MHz 运行的超标量 RISC 处理器每一个时钟周期发出两条指令。假设此处理器的平均 CPI（每条指令的时钟数）为 1，15%的指令是存储指令，并且每一次存储写 8 字节的数据。在没有达到饱和状态时，一个 1 GBps 的总线可以支持多少个处理器？

解答：单个处理器每秒钟将产生 3 000 万次存储指令（每条指令 0.15 个存储 × 每周期 1 条指令 × 每秒钟 1 000 000/200 个周期），因此总的直写带宽是每秒每处理器 240 MB 数据。即使忽略地址和其他信息并忽略读扑空，一条每秒 1 *GB* 的总线只能支持 4 个左右处理器。■

对大多数应用来说，一个回写高速缓存将吸收绝大多数写操作。不过，如果写操作不达到存储器，它们就不会产生总线事务，从而就不清楚其他高速缓存怎么能够观察到这些修改并且保证写的传播。同时，如果允许对不同高速缓存的写操作并发，对那些写操作的一种定序机制也不是一目了然的。我们需要某种更复杂的高速缓存一致性协议来使得“关键的”事件为其他高速缓存可见并且保证写的串行化。

针对回写高速缓存协议的设计空间是相当大的。在研究它之前，让我们先回到在本章引言部分提到的更一般的定序问题，考察由存储同一性模型确定的共享地址空间的语义。

5.2　存储同一性

如果信息是由一个处理器对某个单元写入，而另一个处理器从中读出这样的方式来得以传递的话，那么我们前面所关注的一致性将是非常重要的。最终，写在一个单元的数据将对所有的读取者都会是可见的；但这种一致性并没有指明所写入的数据何时成为可见的。通常，在编写一个并行程序时，我们希望确保一个读操作能够返回一个特定的写操作的值；也就是说，我们希望在写和读之间建立一种序。典型地，我们使用某种形式的事件同步来表达这种依赖关系，并且为此用到的存储地址不止一个。

282 ? 283

比如，考虑图 5-7 中的代码段被处理器 P_1 和处理器 P_2 执行，我们在第 2 章讨论在一个共享地址空间中的点对点事件同步时见过。很明显，在这个程序中，当共享变量 flag 值为 1 时，一直处于空闲状态的处理器 P_2 才能跳出循环，并接着输出变量 A 的值。因为变量 A 的值先于 flag 已经被处理器 P_1 更新了，所以输出的值为 1。在这种情形下，我们通过对另一个地址空间（如 flag）的访问来维持不同处理器访问同一地址空间（如 A）的某种期望顺序。特别是，假定了 P_1 对变量 A 的写操作在它对 flag 的写操作之前就可见了，并且 P_2 对变量 flag 的读操作使它跳出 while 循环，在它对 A 的读操作之前就完成了（一个输出操作就是一个读操作）。前述一致性概念在此并没有隐含着处理器 P_1 和 P_2 对不同地址单元的

P_1	P_2
/*Assume initial value of A and flag is 0*/	
A = 1;	while (flag == 0); /*spin idly*/
flag = 1;	print A;

图 5-7　通过标记变量做事件同步的需求。此图显示两个处理器并发执行两个不同的代码段。从程序员的直觉来看，输出的 A 值必须是 1，因为根据程序语句序，如果进程 P_2 看到 flag = 1，那么它必须也能看到 A = 1

访问顺序，一致性在此仅仅要求 A 的当前值最终对处理器 P_2 是可见的，而不一定是在 flag 的新值被观察到之前。

程序员可能试图通过使用栅障或者其他显式的事件同步方式（如图 5-8 所示）来避免这个问题。我们期望上例中的变量 A 的值在栅障前被设定为 1，其后的输出也为 1。但是这个解决方案存在两个潜在的问题。首先，我们给栅障的语义增加了一个假设，即不仅进程在运行到栅障时必须等待所有其他进程的到来，而且进程在运行到栅障前发出的所有写操作在运行到栅障时要对其他所有进程可见。第二个问题是：一个栅障常常是通过对普通的共享变量（比如，图 5-8 中的 b1）的读写操作来实现的，而并不需要专门的硬件支持。既然这样，那么从机器的角度看，上述方法仅仅是对经过编译的代码中不同的共享变量的访问，而并不是专门的栅障操作。但是，存储同一性却根本没有涉及到这些访问的顺序问题。

```
P1                                  P2
/*Assume initial value of A is 0*/
A = 1;                  . . .
- - - BARRIER(b1)- - - - - - BARRIER(b1)- - - - - - -
                                    print A;
```

图 5-8　用栅障来显式同步，保证对同一个单元不同访问的一定顺序。类似于图 5-7，程序员期望输出 A 的值为 1，由于通过了栅障意味着 P_1 对 A 的写操作已经完成，从而为 P_2 可见

显然，我们希望一个存储系统能提供更深刻的东西，而不仅仅只是对每个单元都能“返回最后一次写操作的值”。为了建立不同的进程对相同单元（例如 A）的访问顺序，我们有时希望一个存储系统遵从同一进程对不同的存储单元（A 和 flag 或者 A 和 b1）执行读写操作的顺序。一致性也没有涉及到对不同存储单元的写操作成为可见的顺序问题。同样地，一致性也没有涉及 P_2 对不同存储单元的读操作相对于 P_1 所见到的顺序。由此，存储同一性本身并没有防止在以上每个例子中输出结果为 0，当然这不是程序员所期望的。

在其他情况下，程序员的主观意图可能并不是很明确。考虑图 5-9 中的例子，进程 P_1 执行的存储访问是通常的写操作，而且 A 和 B 并没有作为标记变量或者是同步变量来使用。如果变量 B 输出值为 2，那么，从直觉上我们会认为变量 A 的输出值为 1 吗？不管答案如何，这两条输出语句都对不同的存储地址执行了读操作，并且一致性概念也没有讲 P_1 的不同写操作对 P_2 是可见的顺序。这个例子实际上是 Dekker 算法（Tanenbaum and Woodhull 1997）中的一个程序段，用来决定两个进程谁先到达一个临界点，成为互斥得到保证的一个步骤。这个算法依赖于这样的假设：一个进程对不同存储地址的写操作为其他进程可见的顺序和那些写操作在程序中出现的顺序相同。很明显，我们需要某些规则，而不仅仅是前面讨论过的一致性来给共享存储地址空间一个明确的语义，也就是定义一个序模型；依照该模型，程序员能推断他们程序的执行结果及其正确性。

```
P1                                  P2
/*Assume initial values of A and B are 0*/
(1a) A = 1;                         (2a) print B;
(1b) B = 2;                         (2b) print A;
```

图 5-9　没有同步的访问序。这里，由于既没有用标记，也没有用显式事件同步，程序员的意图不太清楚

在共享存储空间上，多个进程对存储器的不同单元做并发的读写操作，每个进程都会看到一个这些操作被完成的序。一个存储同一性模型规定了对这种序的若干约束。值得一提的是，这里涉及的并发存储操作既包括对相同单元的，也包括对不同单元的；既可以来自同一进程，也可以来自不同进程。在这个意义上，存储同一性包含了一致性。

284 ~ 285

5.2.1 顺序同一性

在第1章关于通信体系结构基本设计的讨论中，1.4节描述了关于共享地址空间的一个序模型：对于一个多线程程序在单处理器环境中任意交织执行的性质所做的任何推断，在多处理器环境下，线程分到不同处理器上并行运行时应该继续有效。所以，在一个进程中的数据访问顺序也就是这个程序的执行顺序，并且在不同进程间的数据访问顺序也就是某种程序执行顺序的交织。也就是说，在多处理器的情形不应该使进程在共享地址空间中看到任何交织执行都不可能产生的值。这种直观模型被 Lamport 形式化为顺序同一性模型（Sequential Consistency，SC），其精确定义如下（Lamport 1979）[1]：

称一个多处理器是顺序同一的，如果程序在它上面的任何一次执行的结果都和其中所有操作按某种顺序执行的结果一致，并且在这种顺序执行中每个处理器所完成的操作的顺序和它的程序执行顺序一致。

图5-10描述了一种由顺序同一性系统呈现给程序员的存储抽象（Adve and Charachorloo 1996）。除了现在是针对多存储单元外，它同我们曾用来介绍存储同一性的机器模型是相似的。尽管在真实机器里，主存可能被分布在多个处理器中，并且每个处理器有它自己的高速缓存和缓冲区，多个进程在这里看起来是共享一个单一的逻辑存储空间。每个进程依照程序原序执行并完成一次次存储操作；即一个存储操作要待相同进程内的前一操作完成后才开始执行。另外，公共存储区按照某种任意（但希望是公平）的调度方式，交叉响应来自不同进程的操作请求。存储操作是这个交叉顺序中的原子操作，即每个存储操作应该是全局性的

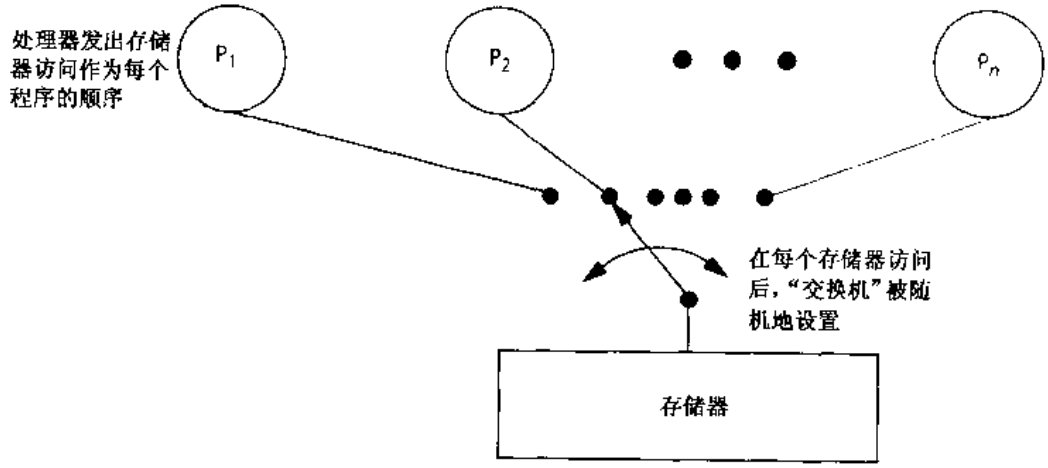


图5-10 在顺序同一性模型下程序员所看到的存储子系统的抽象。这个模型对程序员完全隐藏了背后存储系统硬件的并发性（例如，可能的分布式主存储器、高速缓存和写缓冲区等）

[1] 在软件系统中有两个相关的概念，一是对于数据库并发更新的可串行性（Papadimitriou 1979）；二是关于并发对象的可线性（Herlihy and Wing 1987）。

(对于所有的进程)，一个操作完成后下一个操作才开始执行。

如同一致性，在这里存储操作具体以何种顺序来执行甚至完成是不重要的。对于顺序同一性来说，真正重要的是存储操作的完成满足上述约束条件。在图5-9的例子中，按照SC模型，(A，B)结果值为(0，2)是不允许的，这和我们直观上的认识一致，否则处理器P_1对A和B执行的写操作就不符合程序执行序了。然而，这些存储操作可能确实以1b、1a、2b、2a这样的顺序来执行并完成。它们实际完成的顺序不同于程序执行序是没有关系的，因为执行结果(1，2)与存储操作按照程序执行序来执行并完成的结果是相同的。另一方面，因为产生了在SC模型下不允许的结果(0，2)，所以实际的执行顺序1b、2a、2b、1a不是顺序同一性的。在习题5.6中，还有一些说明顺序同一性的例子。值得注意的是SC模型并不意味着可以不同步了。其原因是SC模型只是允许任意交错的不同进程的操作在单个指令的粒度上执行。如果我们想要在一个进程中的多个存储操作之间维护原子性(互斥)，或者如果我们想要在交错进程间强加某种约束，那么同步是必需的。 286

术语“程序执行序”(也称程序原序)也具有某些需要斟酌的细节。从直观上看，某个进程的程序执行序仅仅是源程序中语句执行的顺序；更明确地说，它是编译器按照直截了当的方式所产生的汇编代码中存储操作的发生顺序。但这不一定是优化编译器产生的硬件存储操作顺序，因为这个编译器可能对存储操作重新排序(遵循特定的约束条件，比如对相同存储地址的存储依赖性)。程序员的头脑中有的是源程序中语句执行的顺序，但是处理器只关注机器指令的执行顺序。事实上，在并行计算机体系结构的每一个接口上都有一个“程序执行序”，特别是在程序员可见的程序设计模型接口上和硬件/软件接口上，并且这个序模型是可以分别定义的。因为程序员总以源程序来推断程序执行的行为，所以在讨论存储同一性模型时，使用源程序来定义程序执行序是有意义的；也就是说，我们所关心的同一性模型由程序设计语言和执行系统呈现给程序员。

实现SC模型，要求系统(包括软件和硬件)体现先前定义的直观上约束条件。这里实际上是两个条件。一个是程序执行序的要求：一个进程的存储操作为自己以及别的进程成为可见的顺序必须符合程序执行序。另一个约束是基于操作的原子性假设来确保所有操作的全序，或者说在进程间交叉执行的情况，对所有进程都是一样的。也就是说，一个所有进程都能看到的操作的完成应该在总体顺序中的下一个操作开始执行前(不管是由哪个进程来执行)。对第二个约束值得注意的是要使写操作表现出原子性，尤其是在一个多副本的存储系统中，一个写操作需要影响到相关的所有副本。在先前定义的顺序同一性中，写操作原子性的要求，意味着一个写操作在所有操作全序中的执行位置应该对所有处理器都是一样的。这就保证了当一个处理器看到自己执行的一个写操作的效果后，在别的处理器见到这个写操作产生的新值前，这个处理器所做的任何事情(例如另一个写操作)对其他处理器都是不可见的。从效果上看，SC模型要求的写操作原子性扩展了一致性要求的写操作的串行化：即写操作的串行化要求同一个存储单元的一系列写操作在所有处理器看来都以相同的顺序发生，而写操作的原子性要求所有的写操作(对于任何存储地址)在所有处理器看来是以相同的顺序发生的。例5.4表明了写操作原子性的重要性。 287

例5.4 考虑图5-11中的三个进程。指出如果写操作的原子性得不到保证，顺序同一性就做不到。

解答： 因为处理器P_2直到变量A值为1时才开始运行，并接着置B的值为1，而处理

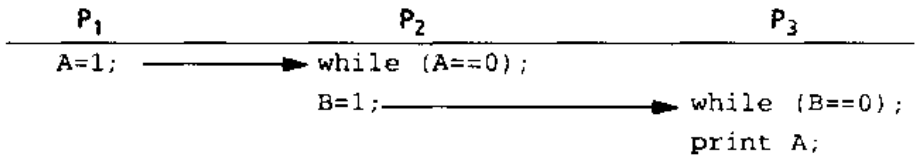


图 5-11 写操作原子性对顺序同一性的重要性的示例

器 P_3 直到变量 B 值为 1 时才开始运行，并接着读出变量 A 的值，由传递性我们可以推断出处理器 P_3 应该发现变量 A 的值为 1。如果处理器 P_2 被允许越过对变量 A 的读操作，在确保处理器 P_3 已经可见变量 A 的新值前对变量 B 执行写操作，那么处理器 P_3 就可能读出变量 B 的新值和变量 A 的旧值（比如，从它的高速缓存中），这样也就和我们对顺序同一性的认识不相符。■

更形式化地说，每个进程的程序执行序在所有操作集上强加了一个偏序；也就是说，它在自己所执行的操作（所有操作的一个子集）上强加了一个序。不同进程操作的交织执行就定义了一个全序。因为 SC 模型并没有定义具体的交织方式，满足每个进程操作偏序的总体程序执行序可能会很多。因此我们有下列定义：

- *顺序同一性的执行*。如果程序的一次执行产生的结果与前面定义的任意一种可能的总体顺序（交织产生的）产生的结果一样，那么程序的这次执行就称为是顺序同一的。也就是说，存在这么一个总体顺序，或者说是程序的一种交叉执行，能产生与实际执行相同的结果。
- *顺序同一性的系统*。如果在一个系统上的任何可能的执行都是顺序同一的，那么这个系统就是顺序同一的。

288

5.2.2 保证顺序同一性的充分条件

讨论了定义和高层需求，让我们来看看多处理器的实现怎么能够满足 SC。我们可能定义一组充分条件，通过它们使得顺序同一性在多处理器中得到保证，无论是基于总线还是分布存储的，也不管是否有高速缓存的一致性。下面这些条件，最初源于（Dubois, Scheurich, and Briggs 1986; Scheurich and Dubois 1987）是相对简单的：

1）每个进程按照程序执行序发出存储操作。

2）发出写操作后，进程要等待写的完成，才能发出它的下一个操作。

3）发出读操作后，进程不仅要等待读的完成，还要等待产生所读的数据的那个写操作的完成，才能发出它的下一个操作。也就是说，如果该写操作对这个处理器来说是完成了（即返回了所写的值），那么这个处理器应该等待该写操作对于所有处理器都完成了。

第三个条件保证了写操作的原子性，要求是相当高的。由于读操作必须等待逻辑上先前的写操作变得全局可见，它不是一个简单的局部限制。我们注意到这些是充分的、但不是必要的条件。在许多情形，顺序同一性可以在较低的串行化要求下也能得到保证，后面将会看到。

源程序定义了一种程序的执行顺序，重要的是编译器不应该改变程序呈现给硬件（处理器）存储器操作的次序。否则，从程序员所看到的顺序同一性就可能被破坏，即使还不一定涉及到硬件层次。但是，在编译和处理器中广为采用的许多优化技术违反这个充分条件。例

如，在一个进程内对不同存储单元的访问重新排序是编译器一贯做的事情，因此处理器就可能给出和程序员所看到的不同的访问顺序。显式并行程序用的是单处理器的编译器，它只关心保持对相同单元的相关性。那些极大改善性能的高级编译优化技术，诸如公共子表达式的删除、常数传播、寄存器分配和循环变换，如循环拆分、循环反序和循环封锁（Wolfe 1989）等等，能改变对不同单元访问的次序，甚至消除一些存储操作[○]。在实践上，为了限制这些编译优化，多线程和并行程序对那些用来维持某种序要求的变量或存储引用进行注释。一种特别严格的例子是在变量声明时赋予其 volatile 属性，它告诉编译器不要对该变量进行寄存器分配，也不要改变在该变量上的存储操作相对于它前后操作的次序。例 5.5 是关于这些要点的一个解释。

289

例 5.5 在图 5-7 中，不同的存储操作的序会如何影响串行程序的语义（只一个进程运行）、在多处理器上并行程序的语义以及在一个多线程程序上的语义（其中两个线程在同一个处理器上交叉执行）？你怎样解决这个问题？

解答：编译可以改变对 A 和 flag 写操作次序，并对串行程序没有影响。然而，这可能违反我们对于并行程序和并发（或多线程的）单处理器程序行为的直观认识。对后者，上下文切换可能在两个重排序的写之间发生，于是切换进来的进程可能看见对 flag 的更新但看不到对 A 的更新。如果编译器改变了读 flag 和 A 的次序，类似违背直觉的情况也会出现。对许多编译来说，通过声明变量 flag 为 volatile integer，而不是简单的声明为 integer，就可以避免变序的情形。还有一些其他的解决方案，将在第 9 章讨论。■

即使编译器保持了程序指令的执行顺序，现代处理器所用的一些复杂的机制，诸如写缓冲、交叉存储、流水线和乱序执行技术（Hennessy and Patterson 1996）等，也会使得从进程看到的存储操作的发出、执行和/或完成不同于程序中所描述的次序。和编译优化类似，这些系统结构方面的优化对于串行程序是能正常工作的，这是因为在那些程序中序的表现只要求对相同存储单元的访问的相关性得到保证，如图 5-12 所示。在并行程序中的问题是，一个进程对不同共享变量操作的乱序处理能被其他进程检测到。

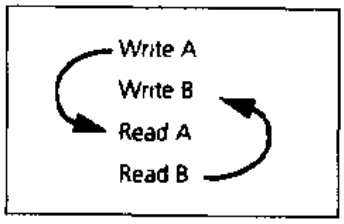


图 5-12 维持一个串行程序在单处理器上执行的操作序。只是对应于两个相关弧的序必须被维持。前面两个、后面两个和中间两个操作可以换序

执行在多处理器中，上述保持 SC 的充分条件是一个相当强的要求，它使编译器变序和乱序执行技术的使用受到了限制。若干弱化一些的同一性模型和技术提了出来，满足 SC 的要求，但放松了充分条件。在第 9 章讨论可扩展共享存储系统时将会考察这些方法。对于本章来说，假定编译器不改变存储操作的顺序，从而处理器所看到的程序的顺序和程序员所看到的是相同的。在硬件方面，假设必须满足充分条件。为此，我们需要有一个处理器能检测

○ 我们注意到，现代编译器用来消除一些存储操作的寄存器分配，不仅会影响存储同一性，还会影响一致性。对于图 5-7 中的标记同步例子来说，如果编译器在 P_2 中将变量 flag 分到了寄存器中，可能会导致进程永远踏步等待下去：高速缓存一致性硬件只是更新或者作废存储器和高速缓存中的内容，而不涉及到寄存器，于是违反了一致性所要求的写传播性质。

它的写操作完成，从而可以继续向下执行的机制（读操作完成的检测是容易的；当数据返回到处理器时，读就完成了）。还需要机制来保证写操作的原子性。对于本章所考虑的所有协议和系统来说，我们将看到它们如何满足一致性（包括写操作的串行化），如何满足顺序同一性（特别地，如何检测写操作的完成、写原子性的保证）以及在仍然满足这些充分条件下，可以进行什么样的简化处理。

对基于总线的机器来说，共享总线所带来的必然的操作串行化对存储器操作的定序是很有用的。容易验证，前面讨论过的两状态直写作废协议实际上相当容易地提供了顺序同一性——不仅是一致性。将关于一致性的论点扩充的关键是注意到对所有存储单元的读和写扑空（不仅是对个别单元），都在总线上串行化了。当读操作得到了一个写操作的值，那个写操作肯定已经就完成了（由于它引起了一个前面的总线事务），这样就保证了写操作的原子性。当对任何处理器进行一个写操作的时候，所有先前的写操作已经以它们访问总线的次序完成了。

5.3　总线侦听协议的设计空间

基于侦听的高速缓存一致性的精彩之处在于，解决一个困难问题的整个机制可以归结到对在系统中自然发生的若干事件进行少量的额外解释。处理器完全不需改变。在程序中不需要插入显式的一致性操作。通过扩充高速缓存控制器的功能，利用总线的性质，程序中固有的读和写隐含地保持了高速缓存的一致性，由总线提供的串行化维护了同一性。每个高速缓存控制器观察并解释由其他高速缓存产生的总线事务，并相应维护自己的内部状态。我们最初的针对直写缓存的设计不是很高效的，但我们现在可以来研究侦听式协议的设计空间，高效利用有限的共享总线带宽。所有这些都用回写高速缓存，允许多个处理器并发地在它们的局部缓存中写不同的存储块，不用任何总线事务。这样，为保证足够的信息在总线上传送，需要引入额外的措施来维护一致性。

290 ~ 291

回顾单处理器的回写高速缓存，处理器写扑空引起缓存从存储器读进一个存储块，更新其中的一个字，将这一块置成已修改（或者脏）状态，以便在替换时写回存储器。在多处理器的情形，这个已修改状态也由协议用来指出一个缓存对某一存储块的单独拥有权。一般来说，我们称某个高速缓存是一存储块的拥有者，如果当对该块有请求时必须由它提供数据（Sweazey and Smith 1986）。称一个高速缓存独立地拥有某一存储块的一个拷贝，如果它是含有该块有效拷贝的惟一高速缓存（主存可能有也可能没有一个有效的拷贝）。这种排它性隐含着该高速缓存可以修改这一存储块而不需要通知其他高速缓存。如果一个高速缓存对一存储块的拥有没有排他性，那么它就应该首先产生一个通知其他高速缓存的总线事务，才能向该块写入新值。试图做写操作的这个高速缓存可能有一份有效的块，但由于要产生一个总线事务，它也称为写扑空，就好像要写入一个不存在或者无效的块一样。如果高速缓存里的这一块处于修改状态，那么显然它是拥有者，并且有排他性。（区别拥有权和排他性的必要性很快就会很清楚了。）

在作废协议中发生一次写扑空时，一种特殊形式的总线事务，称为排他读，用来告诉其他高速缓存将发生一次写操作，并且获得对于该块的单独拥有权。这就是将该块以修改状态放入高速缓存，然后就可以写了。多个处理器不可能并发写同一个存储块，不然会引起非一致的值。由这样的写操作产生的排他读总线事务将由总线来串行化，因此一次只有一个能获

得对该块的单独拥有权。高速缓存一致性的动作由这两种类型的过程驱动：读和排他读。最终，当一个已修改块从高速缓存中被替换时，数据被写回存储器，但这个事件不是由一个对该块的存储操作引起的，对协议几乎是随机性的。不处于修改状态的块在替换时不需要回写，简单地丢掉即可，这是因为存储器有最新的拷贝。许多协议都是针对回写高速缓存设计的，我们来考察一些基本的可选方案。

我们也考虑基于更新的协议。回忆在基于更新的协议中，只要一个共享单元被一个处理器写，它的值就在所有含有该存储块的处理器的高速缓存中更新[⊖]。这样，当这些处理器接着访问这一存储块时，它们就可直接从它们的低时延高速缓存中得到。所有其他处理器的高速缓存的更新由一次总线过程完成，这样当有多个共享者时就节省了带宽。相比之下，对于基于作废的协议，一个处理器在执行写操作时，所有含有相应存储块的处理器高速缓存的状态被置成无效的，于是这些处理器在下一次读的时候必须通过一次扑空，从而也是一次总线事务来获得那一块。不过对执行该写操作的处理器来说，在其他处理器访问之前，对那一存储块后继的写操作不会引起总线事务（在更新协议时是会引起的）。在一个处理器连续向一个存储块做写操作，中间没有其他处理器访问该块的情形，这是有吸引力的。具体的折中要复杂许多，它们取决于机器的工作负载情况。对此，在 5.4 节有量化的讨论。一般来说，基于作废的策略要更健强，因此许多厂家都提供作为缺省协议。有些厂家提供更新协议作为备选，用于某些对应于特定数据结构或页的存储块。

292

对于协议（更新或作废）和高速缓存策略所作的选择直接影响状态的选择、状态转换图和相关动作的选择。对于计算机体系结构设计师来说，在这一层次的设计任务有很大的灵活性。我们这里不一一列举所有的可能性，下面通过考虑三种常见的一致性协议来体会有关的设计思路。

5.3.1 一种三态（MSI）回写作废式协议

我们这里考虑的第一个协议是针对回写缓存、基于作废的协议。它和在 Silicon Graphics 4D 系列多处理器中所用的很相似（Baskett，Jermoluk，and Solomon 1988）。这个协议利用任何回写协议都要有的三个状态，来区别有效存储块的未修改（干净）和已修改（脏）。具体来说，三个状态分别为*已修改的*（M），*共享的*（S）和*无效的*（I）。无效的意义是明显的。共享意味着该存储块在这个高速缓存中存在但未修改，主存中有最新的值，其他高速缓存有可能有最新的拷贝（共享）。已修改的，也称脏的，意味着只有这个缓存有一个该存储块的有效拷贝，主存中的拷贝是过时的。在对一个共享或者无效块做写操作，并将其置成已修改状态之前，所有其他拷贝必须通过一个排他读总线事务作废。除了引起作废外，这个总线事务还用来为写操作定序，从而保证写操作对其他高速缓存是可见的（写传播）。

处理器发出两种类型的请求：读（PrRd）和写（PrWr）。所读和所写的，可能是存在于高速缓存中的一个存储块，也可能是高速缓存中不存在的。对于后者的情形，高速缓存中的某一块就要被新请求的块所替换，如果所存在的块处于已修改状态，它的内容就要被写回主存。

⊖ 这是一种写广播的情形。人们也研究过读广播的设计，其中一个高速缓存在总线上看到一个读操作后将自己含有的一个已修改存储块送上总线，所有其他高速缓存中的拷贝也得到更新。

我们假定总线允许下面的过程：

- 总线读（BusRd）：这个过程由引起高速缓存扑空的 PrRd 产生，处理器指望一个数据返回作为结果。高速缓存控制器将地址放在总线上，请求一个它不想修改的拷贝。存储系统（可能是另外的高速缓存）提供数据。
- 总线排他读（BusRdX）：这个过程由 PrWr 产生，这个 PrWr 要读的一个存储块，要么不在高速缓存或者在高速缓存但不处于已修改状态。高速缓存控制器将地址放到总线上，请求一个它要修改的排他拷贝。存储系统（可能是从另一个高速缓存）提供数据。所有其他高速缓存被作废。一旦高速缓存得到了这个排他拷贝，写就可以在高速缓存里完成。处理器可能要求一个认可来作为这个过程的结果。
- 总线回写（BusWB）：这个过程由高速缓存控制器在回写时产生；处理器不知道它的发生也不指望有一个响应。高速缓存控制器将地址和存储块的内容放到总线上。主存被更新。

293

总线排他读（有时称为读后拥有）是仅有的特别和高速缓存一致性有关的过程。为支持回写协议所需要的新动作是，除了改变高速缓存块的状态外，一个高速缓存控制器能够根据一个观察到的总线事务，从它的高速缓存中将一个被引用的存储块提出来，放在总线上，而不是让主存来提供数据。当然，高速缓存控制器也能启动如上所述的总线事务，提供回写数据，或者拾取出存储系统提供的数据。

1. 状态转换

图 5-13 中的状态转换图表达了在这种侦听式协议中一个存储块的各种情况。其中的状态是这样组织的，越是接近顶部的状态，表示对应的存储块和处理器绑定得越紧。处理器读一个无效的（或者不存在的）存储块，引起一个 BusRd 事务来解决这次扑空。新装入的存储块在发出请求的缓存中被提升，在这个状态图中，即从无效到共享状态，无论其他高速缓存是否也有一个拷贝。任何有着这一存储块（处于共享状态）的高速缓存观察到这一 BusRd，但是不采取任何行动，而让主存储器提供数据响应。然而，如果一个高速缓存有着处于已修

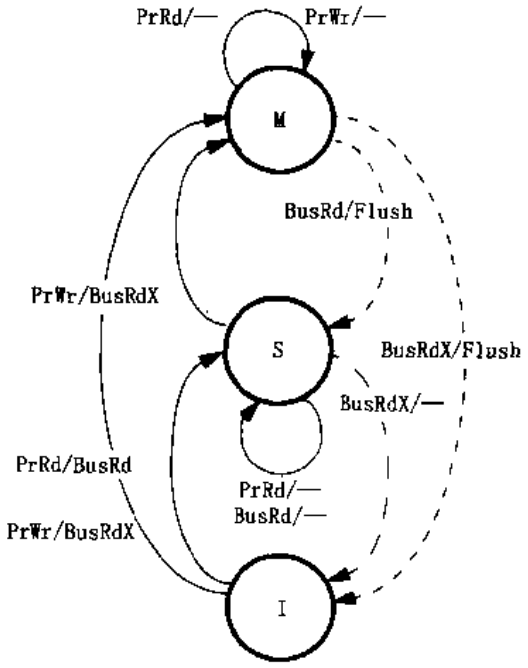


图 5-13 基本三态作废式协议。M、S 和 I 分别表示已修改、共享和无效状态。记号 *A/B* 表示如果控制器从处理器或总线方观察到了事件 *A*，那么除了状态改变外，它要产生一个总线事务或者动作 *B*。"—"表示空动作。由所观察到的总线事务所引起的转移用虚线弧表示，由于本地处理器动作所引起的转移用实线弧表示。如果有多个 *A/B* 和一个弧相联，指的是多种输入能引起相同的状态转换。为完整性起见，我们应该对每个状态和每个可观察到的事件说明相应的动作。如果有些转移没有显示出来，这里表示要么不关心，要么没有什么动作需要实施。为简单起见，替换和它们可能引起的回写在图中没有示出

改状态的存储块（只可能有一个），当它观察到总线上一个 BusRd 事务，它就必须参加这个事务（因为主存的块是过时的）。这个高速缓存将数据送到总线上替代存储器，并且将它的拷贝降级为共享状态（见图 5-13）。存储器和高速缓存都接受这个存储块。这个操作的完成可以通过在 BusRd 期间直接跨总线的高速缓存——高速缓存传送或者通过在 BusRd 事务中给出一个出错信号，产生一个写过程来更新存储器。在后者的情形，发起的高速缓存将最终重新产生它的请求，从存储器中得到这一块。（也可能让总线上的数据只被发请求的高速缓存，而不被存储器接收，使存储器内容仍然是陈旧的，但这要求有更多的状态［Sweazey and Smith 1986］。）

对一个无效的块做写操作导致一个写扑空，处理的过程首先是将整个块装入，然后修改它其中的有关字节。这个写扑空产生一个排他总线读事务，引起所有其他高速缓存拷贝被作废，从而保证发请求的高速缓存对于该存储块具有单独拥有权。由这个排他读所返回的数据块被提升到已修改状态，其中相关的字节然后得到更新。如果另一个高速缓存后来请求排他的访问，那么作为对它的 BusRdX 事务的响应，在将它的排他拷贝放到总线上后，这个块就要被作废（降级到无效状态）。

最有意思的状态转换发生在对一个共享块进行写操作的时候。如先前所讨论的，这基本上可以当作写扑空来处理，用一个排他读总线事务来获得排他的所有权；在本书中我们称它为写扑空。由于数据已经在缓存中了，在排他读里返回的数据在这种情况下可以忽略，这一点不像对无效或者不存在块进行写操作的情形。事实上，在总线协议中一种常见的减少数据流量的优化是引入一种新的事务，称为总线升级或者 BusUpgr。一个 BusUpgr 得到排他的所有权，就像 BusRdX 那样让其他拷贝作废，但它不引起主存或者其他设备进行数据响应。不管是 BusUpgr 还是 BusRdX（让我们还是假设 BusRdX），在发请求的高速缓存中的块都转换到已修改状态。在这个已修改状态下，对于那个块的其他的写操作都不产生额外的总线事务。

294
~
295

从高速缓存里替换一个存储块，逻辑上就是使那个块降级为无效的（不存在的）。一次替换因此就引起两个块的状态机在高速缓存中改变状态：被替换的块从它的当前状态到无效，新带入的块从无效（不存在）到它的新状态。后者状态的改变不能在前者之前发生，这要求在实现中采取一些措施。如果被替换的块处于已修改状态，这个从 M 到 I 的替换转移产生一个回写事务。其他的高速缓存在这个过程上不采取什么特殊行动。如果这个被替换的块处于共享或者无效状态，那么它就不会引起任何总线事务。为简单起见，替换在状态图中没有表示出来。

注意，为了能完整地说明这个协议，对于每一个状态我们必须有外向的弧，带有对应于所有可观察事件的标记（从处理器和总线一边的输入），还必须表示相对于它们的动作。当然，这些动作和状态转移有时候可能是空的，在那种情况下，可以要么显式说明空动作（见图 5-13 中的状态 S 和 M，或者可以简单地从图中略去这些弧（见状态 I）。进而，由于我们将不存在状态当作无效处理，当扑空时一个新存储块被带进高速缓存的时候，状态转移的处理就好像该高速缓存块的先前状态是无效的一样。例 5.6 说明状态转换图的解释方式。

例 5.6 利用 MSI 协议，说明图 5-3 所示情形的状态转移和总线事务。

解答：结果如图 5-14 所示。■

对于回写协议来说，在存储器被实际更新以前，一个块能被多次写。一个读操作所得的

处理器动作	在 P_1 状态	在 P_2 状态	在 P_3 状态	总线动作	提供数据的实体
1. P_1 reads u	S	–	–	BusRd	Memory
2. P_3 reads u	S	–	S	BusRd	Memory
3 P_3 writes u	I	–	M	BusRdX	Memory
4. P_1 reads u	S	–	S	BusRd	P_3 cache
5. P_2 reads u	S	S	S	BusRd	Memory

图 5-14 和图 5-3 中的处理器事务对应的三态作废协议执行情况。此图表现了在每次处理器动作结束时相关存储块的状态，所产生的总线事务以及提供数据的实体

数据可能不是来自存储器，而是来自一个高速缓存。事实上，可能是这个读而不是一个替换引起存储器被更新。除此以外，写命中不出现在总线上，因此相对于其他处理器来说，写的概念有些不同。事实上，说正在进行一个写操作，意味着这个写被“弄得可见”。对一个共享的或者无效块的写的可见是通过它所触发的总线排他读事务。写者在这个事务后将“观察到”在它缓存里的数据。这个写为其他处理器可见，是通过排他读所产生的作废，那些处理器在实际看到所写的值之前将经历一次缓存扑空。对一个已修改块的写命中对其他处理器是可见的，但同样只是在通过总线事务的一次扑空之后才能观察到。这样，在 MSI 协议中，当 BusRdX 事务发生时，对一个未修改块的写就被完成或者是可见了；对一个已修改块来说，当这个块在写者的高速缓存中被更新时，写的效果就可见了。

2. 对一致性的保证

在回写协议中，由于读和写的发生都可以不产生总线事务，它是否能满足一致性条件并不是显而易见的，尽管一致性要比顺序同一性要求弱得多。让我们首先考察一致性。从前面的讨论来看，写操作效果的传播是没有问题的，因此让我们集中考虑写的串行化问题。排他读事务保证了当一个块实际写入缓存的时候，该缓存有着惟一的拷贝，就像在直写协议中的一个写事务。紧跟着的，是在缓存中的一个写，发生在缓存控制器处理任何其他总线事务之前，因此它的序对所有处理器来说（包括写者），相对于其他总线过程都是相同的。和直写协议的惟一区别是，针对一个存储单元的操作序列，不是所有写都会产生总线事务。然而，这里的关键点是，如果对某个块的两次操作的确出现在总线上了，那么只有一个处理器能完成这样的写命中；即那个最近完成对那个块排他读总线事务 w 的处理器 P。在串行化中，这个写命中的顺序因此出现在（以程序执行序）w 和下一次对于该块的总线事务之间。处理器 P 的读操作将清楚地看见它们（相对于其他的写）以这种序出现。至于另一个处理器的一个读操作，对那个块至少有一个总线事务，将读的完成和这些写命中的完成分开。这样一个总线事务保证了那个读也以同样的顺序看到这些写。这样，所有处理器的读看到所有写的顺序是相同的。

3. 对顺序同一性的保证

为了分析 SC 是如何得到满足的，让我们首先看看定义本身，看看所有存储操作的一种全局的交叉执行怎么能够构造出来。如同直写缓存，关于总线的串行仲裁结果事实上定义了一个针对所有存储块的总线事务的全序，不只是关于某个块的操作序。所有的缓存控制器观察到的读和排他读总线事务的序都是相同的，并且以这个序完成作废。在相继的总线事务之间，每个处理器以程序中的次序完成一系列存储操作（读和写命中）。这样，一个程序的任何执行定义了一个自然的偏序：

296 ~ 297

存储操作 M_j 是操作 M_i 的后继，如果 1）这些操作由相同的处理器发出并且 M_j 在程序中出现在 M_i 之后，或者 2）M_j 产生一个跟在 M_i 存储操作后面的总线事务。

这个偏序从图形上看来类似于图 5-6，不同之处在于在一个段中的本地序列除了有读外还有写，并且排他读和读总线事务在建立这个序中都起了重要作用。在总线事务之间，任何来自于不同处理器的局部操作（命中）序列的交叉导致一个统一的全序。对于出现在总线事务之间同一段中的写，一个处理器将观察到其他处理器的写操作，其次序依从它所产生的总线事务，而它自己的写的序是程序中的序。

我们还可以从充分条件的角度看到 SC 是如何被满足的。写完成的检测是当排他读总线事务出现在总线上且这个写在缓存中进行时做出的。读完成的条件，提供了写操作的原子性，它被满足是因为一个读要么 1）引起一个总线事务，跟在其值正在被返回的读的后面，在这种情况下，这个写必须在这个读之前全局完成；2）由同一个处理器以程序执行序跟着这个读；或者 3）随着在完成写操作的同一个处理器上以程序执行序发生，在这种情况下，该处理器已经等着该写操作的全局完成（在可见之前）。这样，所有的充分条件都很容易得到了保证。在第 6 章讨论协议的实现时我们还要回到这个问题上来。

4. 低层设计的选择要素

为了了解在协议中某些隐含的设计决策，让我们更仔细地考察当针对一个存储块的 BusRd 被观察到时从 M 状态发生的转移。在图 5-13 中，我们转移到状态 S 并且将存储块的内容放到总线上。尽管将内容放到总线上是必须的，我们仍可以考虑转移到状态 I，这样完全放弃这个块。转移到 S 而不是 I 的选择反映了设计者认识：和另外的处理器对该存储块做写操作相比，最初的这个处理器更可能继续读这个块。直觉上看，这个认识对于大多数读数据是成立的，而且这正是许多程序中常见的。然而，一种不成立的常见情形是对于在进程之间来回传送信息的一个标记或者缓冲区：一个处理器写入，另一个处理器读出且修改，然后第一个处理器读并修改，如此等等。对一个共享计数器的累加表现了类似的跨处理器的迁移行为。这种将赌注压在读共享的问题是每一个写都要首先产生一个作废，因此增加了它的时延。的确，用在早期 Synapse 多处理器中的一致性协议采取的是不同的选择，在 BusRd 上从 M 直接转移到 I 状态，这种做法的基本出发点是认为上述那种迁移模式会经常发生。某些机器（Sequent Symmetry model B 和 MIT Alewife）试图在这种迁移型访问模式被观察到时对这种协议做些适应性改造（Cox，Fowler 1993；Dahlgren，Dubois，and Stenstrom 1994）。在本章的后面将会看到，这些选择可能影响存储系统的性能。 298

5.3.2　一种四态（MESI）回写作废式协议

如果我们考虑一个串行程序运行在多处理器上的情形，我们就可能产生对 MSI 协议的一种顾虑。事实上，这种多道程序的用法是在小规模多处理器系统上最常见的工作负载情况。当进程读进并且修改一个数据项，尽管并没有其他处理器来共享这个数据，但在 MSI 协议中还是要产生两个总线事务。第一个是 BusRd，将存储块置成 S 状态，第二个是一个 BusRdX（或者 BusUpgr），将 S 转换成 M 状态。通过增加一个状态，指出这个块是惟一的（排他的）拷贝，但没被修改，并且装入这个块时使它处于这个状态，我们能够省去后面一次总线事务，因为这个状态指出没有其他处理器的缓存也含有这一块。这个新的状态，称为干净的独

占或者非拥有的独占（甚至就简单称“独占”），指出一种在共享和已修改之间的情况 它是独占的，因此不同于共享状态，缓存能够执行写操作并且转移到已修改状态，而不需要进一步的总线事务；但它又不隐含拥有权（存储器里也有一个有效拷贝），因此不同于已修改状态，缓存在观察到关于那个块的请求时不需要答复。这种 MESI 协议的各种变形用于许多现代微处理器中，包括 Intel Pentium、PowerPC 601 以及用于 SGI Challenge 多处理器的 MIPS R4400。这个协议首先是由 UIUC 的研究人员发表的（Papamacros and Patel 1984），因而常称为 Illinois 协议（Archibald and Baer 1986）。

MESI 协议于是由四个状态构成：已修改（M）或者脏的、干净的独占（E）、共享（S）和无效（I）。M 和 I 的语义和先前的一样。E，干净的独占或者独占状态，意味着只有一个缓存有此块的一个拷贝，并且是没被修改的（即主存相应内容是最新的）。S 意味着在它们的缓存中潜在有两个或者多个处理器有这个存储块，处于未被修改的状态。所需的总线事务和动作非常类似于 MSI 协议的情形。

1．状态转换

当一个存储块首次被某个处理器读进来时，如果一个有效的拷贝存在于另一个缓存中，和通常一样，它就要以 S 状态进入这个处理器的缓存。不过，如果这时没有其他的缓存有拷贝（例如，在顺序应用程序执行的情形），它就要以 E 状态进入缓存。当这一块被同一个处理器写的时候，由于没有其他的缓存有拷贝，它就可以直接从 E 状态进入 M 状态，而不产生总线事务。如果另一个缓存在同时获得了一个拷贝，这一块的状态就要被侦听协议从 E 降级到 S。

299

这个协议对于总线的物理互连加进了一个新的要求。一个附加的共享信号（S）要被缓存控制器用来在 BusRd 上确定是否有其他缓存当前持有数据。在总线事务的地址阶段，所有缓存确定它们是否含有所要求的存储块，若如此，就给出这个共享信号。这个信号是一个线或连接，因此提出请求的控制器能够观察到是不是有其他处理器缓存含有所请求的存储块，从而能够决定是将装入的存储块置成 E 还是 S 状态。

图 5-15 表示了 MESI 协议的一个状态转换图，我们还是假设不用 BusUpgr 事务。记号 BusRd（S）表示由共享信号 S 引起的总线读事务；BusRd（$\overline{S}$）意味着 S 没有作用。BusRd 则意味着我们不关心 S 在那个过程中的值。无论处于什么状态，对于一个块的写将它提升到 M 状态，但如果它是 E 状态的话，就不会有总线事务。观察到一个 BusRd，处理器会将相关存储块的状态从 E 降级到 S，因为现在有另一个拷贝存在了。和通常一样，观察到 BusRd 将从 M 到 S 状态，还要将存储块的内容送到总线上；同样，这个块只可能被请求缓存拾取，而不是被主存，但这可能要求超出 MESI 以外附加的状态。（第五个称为被拥有的状态可以被加进来，它指出即使存在其他共享的拷贝，这个缓存（而不是主存）要在观察到相关的总线事务时负责提供数据。这就导致一种 5 状态 MOESI 协议［Sweazey and Smith 1986］）。注意，即使没有其他拷贝存在，一个块也可能处于 S 状态，这是由于拷贝可能被替换（S→I）而不需要通知其他拷贝。满足一致性和顺序同一性的论点和在 MSI 协议中的相同。

2．低层设计的选择要素

关于基于总线的协议的一个有趣的问题是，当发生一个 BusRd 事务时，如果存储器和另一个缓存都有拷贝，谁应该提供数据块。在 MESI 协议的原始（Illinois）版本中，提供数据的是缓存，这称为是一种缓存到缓存共享的技术。采用这种做法的出发点是：缓存是 SRAM，

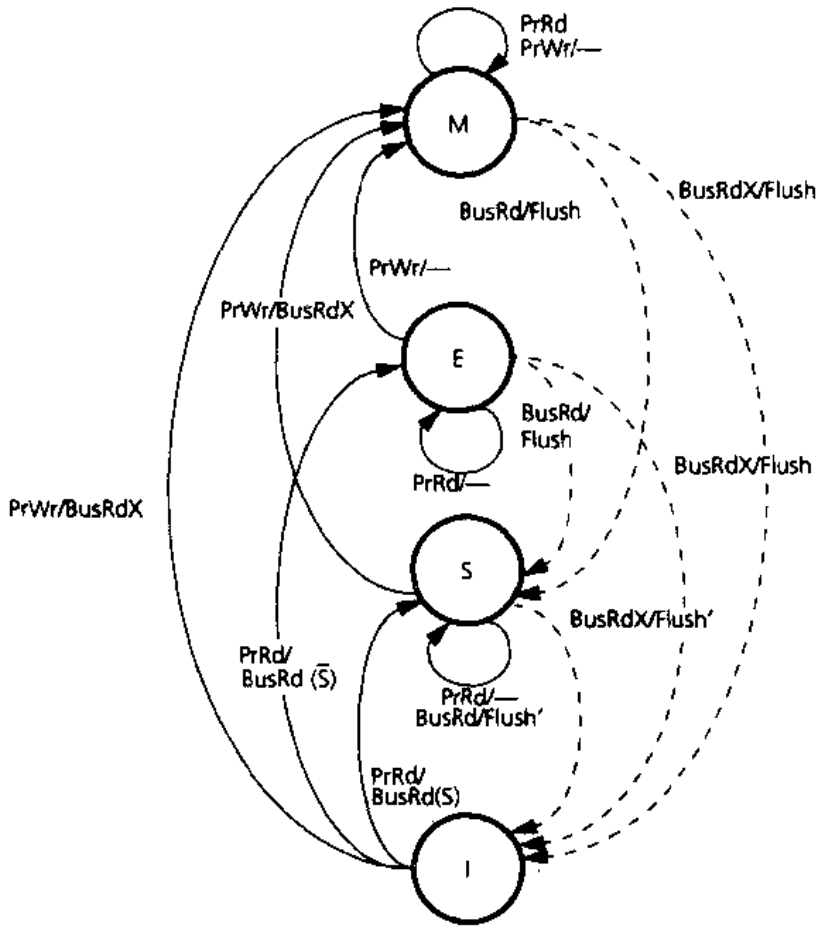


图 5-15 Illinois MESI 协议的状态转换图。MESI 是已修改（M）或脏的、独占（E）、共享（S）和无效（I）状态的英文缩写。这里用的记号和图 5-13 相同。E 状态有助于在串行程序数据不共享时降低总线上的流量。只要可能，Illinois 版本的 MESI 协议都让高速缓存，而不是存储器为 BusRd 和 BusRdX 事务提供数据。由于多个处理器可能在它们的缓存中有相同的存储块，我们需要选出一个来向总线提供数据。图中 Flush' 只是针对相关的那个处理器而言的；其他处理器做它们常规的动作（作废或没动作）。一般来说，状态转换图中的 Flush' 指出相关的存储块被清除，其条件是用到了缓存到缓存的共享，并且清除是由负责提供数据的缓存来实行的

存储器是 DRAM，前者能更快地提供数据。然而，这个优势在现代基于总线的机器中不一定存在，让另一个缓存来提供数据可能比直接从存储器提供的代价要更高。缓存到缓存共享也增加了基于总线协议的复杂性：主存必须等待，直到它能肯定没有缓存将提供数据后才能驱动总线，并且如果多个缓存都有这个数据，那么就要有一个选择算法来确定哪一个应该提供数据。另一方面，这个技术对于物理上分布存储的多处理器系统是有用的（如我们在第 8 章会看到的），因为从近处的缓存获取数据要比从远处的存储器快得多。特别是，在由 SMP 节点构成的网络型机器中，数据请求者的 SMP 节点中的缓存可能提供数据，这种缓存到缓存共享的方式可能就特别重要了。基于这种考虑，斯坦福 DASH 多处理器（Lenoski et al. 1993）用的就是这种缓存到缓存的传送机制。

5.3.3 一种四态（Dragon）回写更新式协议

现在让我们考察一种回写缓存上基于更新的协议。这个协议最初是由 Xerox PARC 的研

300-301

究人员提出来的，背景是他们的 Dragon 多处理器系统（McCreight 1984；Thacker，Stewart，and Satterthwaite 1988）。该协议的一个增强版本用在 Sun SparcServer 多处理器系统中（Catanzaro 1997）。

Dragon 协议由四个状态构成：干净的独占（E）、干净的共享（Sc）、共享已修改（Sm）和已修改（M）。干净的独占（简称独占）的意图和含义与前面相同：只有一个缓存（这个缓存）有这一存储块的一个拷贝，并且还没有被修改（即，主存的内容也是有效的）。干净的共享意味着，潜在地有两个或者多个缓存（包括这个）有这一存储块，主存可能有但不一定是最新的。共享已修改意味着潜在地有两个或者多个缓存有这一存储块，主存不是最新的，并且当这个块要被从缓存中替换出去时，这个缓存有责任对主存进行更新（即这个缓存是拥有者）。一个存储块在一个时间只能在一个缓存里是 Sm 状态。不过，一个存储块在一个缓存是 Sm 状态，同时在其他缓存是 Sc 状态的情况是相当可能的，或者没有缓存是 Sm 状态，但有一些是 Sc 状态。这就是为什么当在一个缓存里发现某个存储块是 Sc 状态时，不能断定存储器里的内容是否是最新的道理；还取决于这个存储块是否在其他缓存里处于 Sm 状态。和先前一样，M 表示独占的拥有权：这个存储块已经修改了（脏）并且只是出现在这个缓存里，主存的内容是陈旧的，一旦发生替换，这个缓存有责任提供数据来更新存储器的内容。注意，这里没有在前面的协议中具有的显式的无效（I）状态。这是因为 Dragon 是一个基于更新的协议；它总是保持缓存中块的内容是最新的，因此如果标记匹配成功，则总是可以用缓存中的数据。然而，如果一个存储块根本就不在缓存中，它就可以被想像为一种特别的无效或者不存在状态。○

处理器请求，总线事务以及 Dragon 协议的动作都类似于 Illinois MESI 协议。处理器仍然假设为只发出读（PrRd）和写（PrWr）请求。不过，由于没有一个无效状态，为了说明当标记匹配不成功时的动作，加上两种请求类型：处理器读扑空（PrRdMiss）和写扑空（PrWrMiss）。至于总线事务，有总线读（BusRd）、总线回写（BusWB）和一个新的称为总线更新（BusUpd）的事务。BusRd 和 BusWB 事务有着通常的语义。BusUpd 事务将这个处理器写的特定的字（或者字节）广播到总线上，从而所有其他处理器的缓存能更新它们自己。这里广播的只是被修改的内容，而不是整个存储块。这种处理方式是指望总线带宽能得到更高效的利用（见习题 5.4，其中谈到这一种愿望不一定总是奏效的）。如同 MESI 协议，为了支持 E 状态缓存控制器需要用到一个共享信号（S）。最后，协议提出的惟一新要求是，当一个相关的 BusUpd 事务将数据广播到总线上时，缓存控制器要能够用总线上的内容来更新一个本地缓存的存储块（标记为 Update 动作）。

302

1. 状态转换

图 5-16 所示的是 Dragon 更新协议的状态转换图。按照处理器为中心的观点，能够用在一个缓存发生读扑空、写（命中或者扑空）或者替换所采取的动作（读命中没有动作）来解释这个图。

- 读扑空。产生一个 BusRd 事务。取决于共享信号（S）的状态，读到的这个存储块装入缓存后，被置成 E 或者 Sc 状态。如果这个块在某个其他缓存里是 M 或 Sm 状态，

○ 逻辑上讲，还有另一个状态，但它只是用来自启这个协议。一个“扑空模式”位用在每条缓存线上，强制导致一次扑空。初始化软件在开始将数据读入缓存中来时，这一“扑空模式”位是激活的，以保证第一次读产生扑空。在这第一次扑空后，“扑空模式”位关闭，缓存正常运行。

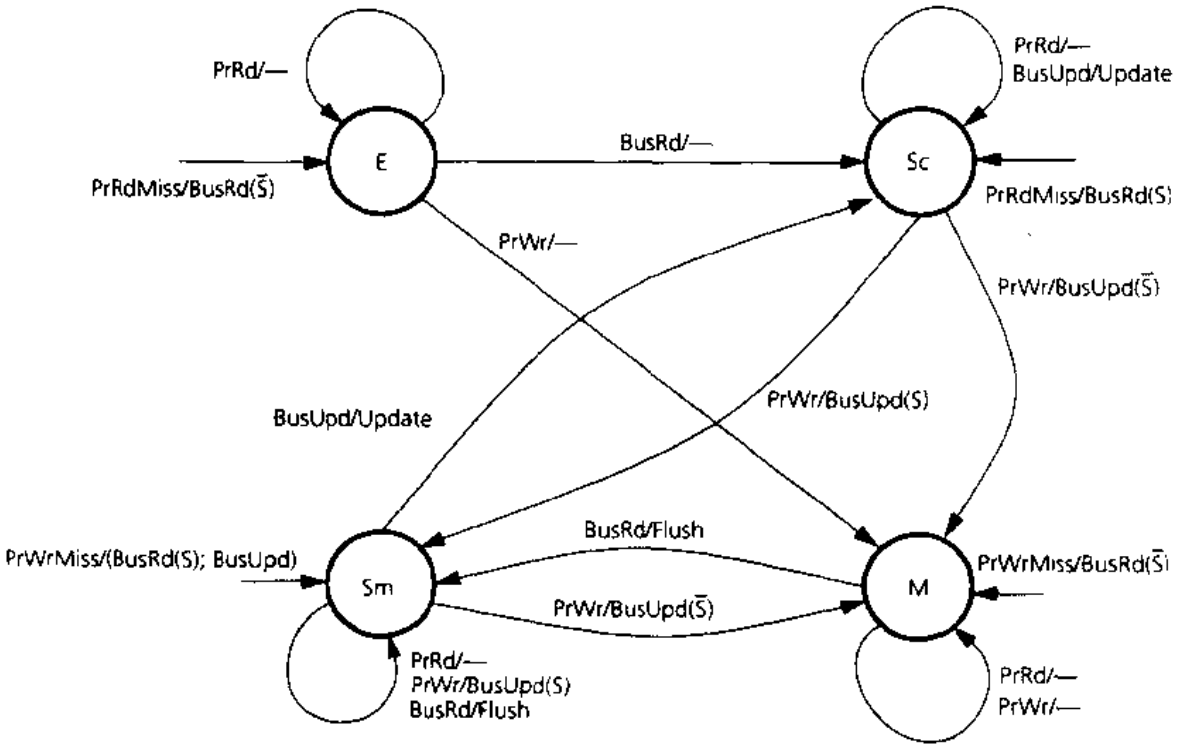


图 5-16 Dragon 更新协议的状态转换图。四个状态是独占（E）、干净的共享（Sc）、已修改共享（Sm）、和已修改（M）。因为更新协议总是保持使缓存中的数据为最新，这里没有无效状态（I）

那么缓存就要给出这个共享信号并且将最新的数据提供到总线上，该存储块以 Sc 状态装入本地缓存。如果它在其他缓存里是 M 状态，它就要改变为 Sm。如果它在其他缓存是 Sc 状态，则存储器提供数据，它以 Sc 状态装入。如果所有其他缓存都没有相应拷贝，则共享信号线保持为无效的，数据由主存提供，这个块以 E 状态装入本地缓存。

- 写。如果这个块在本地缓存中的状态是 M，那么不需要有什么动作。如果是 E，那么就变到 M 状态，也没有什么动作。然而，如果是在 Sc 或者 Sm 状态，就要产生一个 BusUpd 事务。如果其他某个缓存有一个拷贝，它们就要置共享信号，更新它们拷贝中的相关字节，并且如果必要的话要将状态改为 Sc。本地缓存也更新它的拷贝，如果必要的话要将状态改为 Sm。主存不被更新。如果没有其他缓存有该数据的拷贝，则共享信号保持无效，本地缓存更新状态变为 M。最后，如果在写操作的时候这个存储块在缓存中不存在，这个写就简单地处理为一次读扑空过程，跟着是一个写过程。这样，就产生了第一个 BusRd。如果这个块也在其他缓存中，就要产生一个 BusUpd，存储块以 Sm 状态装入本地缓存；否则，它就以 M 状态装入。
- 替换。在一次替换时（图中没有显示相应的弧），只是当相关的存储块处于 M 或者 Sm 状态上，我们才通过一个总线过程将存储块写回存储器。如果它是 Sc 状态，那么或者某个其他缓存有它在 Sm 状态，或者谁也没有。在这种情况下它在存储器中就是有效的。

303

针对一个我们已经熟悉情形，例 5.7 解释了这些状态转换过程。

例 5.7 用 Dragon 更新协议，针对图 5-3 所示的情形，指出状态转换和总线事务。

解答：结果如图5-17所示。我们可以看到在更新协议中，关于处理器的动作3和4只有一个字被送到总线上，整个存储块在基于作废的协议中被传送两次。当然，我们也可以构造一种情况，其中作废协议的表现要好于更新协议。在5.4节，我们将讨论详细的权衡。■

处理器动作	在 P_1 状态	在 P_2 状态	在 P_3 状态	总线动作	提供数据的实体
1 P_1 reads u	E	—	—	BusRd	Memory
2. P_3 reads u	Sc	—	Sc	BusRd	Memory
3 P_3 writes u	Sc	—	Sm	BusUpd	P_3 cache
4 P_1 reads u	Sc	—	Sm	null	—
5. P_2 reads u	Sc	Sc	Sm	BusRd	P_3 cache

图5-17 Dragon更新协议对于图5-3中处理器的操作所产生的动作。此图表明在每次处理器动作结束时相关存储块的状态所产生的总线事务以及提供数据的实体

2. 低层设计的选择要素

同样，在这个协议中也有许多隐含的设计抉择。例如，共享已修改状态是有可能取消的。事实上，用在DEC Firefly多处理器系统中的更新协议就是这么做的。这里的道理是每当BusUpd事务发生时，和其他持有该存储块的缓存一道，主存也可以更新它的内容；因此，干净的共享就足够了，已修改共享就不再需要。而Dragon协议是基于这样的假设：更新SRAM缓存要比更新DRAM主存快得多，因此在所有BusUpd事务上等待主存的更新就不合适了。另一个微妙的因素和缓存替换时所取的动作有关。当一个干净的共享存储块被替换时，应该通过一个总线事务让其他缓存得知这个替换吗？这样做的理由可能是，如果只有一个缓存持有该存储块的一个拷贝，它就可以将其状态改为独享的或者已修改的。这样做的好处是，这个由替换引起的总线事务可能不在一次存储器操作的关键路径上，而它所省下来的后来的总线事务可能就在关键路径上。

由于在更新协议中所有写操作都会出现在总线上，对于一条原子型总线来说，写串行化、写完成的检测以及写操作的原子性都是相当明确的，就像在直写缓存的情形那样。然而，对于含有作废和更新两种功能的协议，我们必须考虑许多精细的实现问题和竞争条件，即便是原子型总线和单级缓存也是如此。将在第6章中讨论这一层次的协议和硬件设计以及带有流水线总线、多级缓存层次的更实际的情形、还有能够变序完成存储操作的硬件技术。尽管如此，基于到目前为止所考虑的状态图的层次，也能够量化许多协议的权衡考虑。

5.4 关于协议设计中若干折中的评估

如同任何其他复杂系统，一个多处理系统的设计要做出许多相互关联的决定。即使处理器已经被选定了，也必须决定系统所要支持的最大处理器的个数、缓存层次结构的各种参数（例如，层次数，每级缓存的大小、关联度、块的大小以及是采用直写还是回写策略等等）、总线的设计（例如，数据和地址总线的宽度、总线协议）、存储系统的设计（例如，是否采用多模块交叉存储技术、存储模块的宽度、内部缓存的大小）、还有I/O子系统的设计。这其中许多因素和单处理器系统所要考虑的类似（Smith 1982），但矛盾可能进一步突出了。例如，直写缓存对于多处理器来说可能是个不好的选择，因为总线带宽是由多个处理器共享的，存储器可能需要在更大程度上交叉，因为它要服务于多个处理器的缓存扑空。较大的缓存关联度在减少冲突扑空（要产生总线传输）方面也可能是有用的。

缓存一致性协议对于多处理器来说是一个关键的新设计考虑。它包括协议类型（作废还是更新）、协议状态和动作以及低层实现的权衡。协议的决定和所有其他设计因素都有关系。另一方面，协议影响系统部件的时延和带宽特征被强调的程度；另外，除存储组织和通信体系结构外，性能特征也影响协议的选择。如第 4 章所讨论的，这些设计的决定需要针对实际程序的行为来评估。这样的评估在 20 世纪 80 年代后期是很时兴的，尽管用的是一些不成熟的并行程序作为负载（Archibald and Baer 1986；Agarwal and Gupta 1988；Eggers and Katz 1988，1989a，1989b）。

304 ~ 305

在真实的系统设计中要做出决定，部分是艺术，部分是科学。艺术源于过去的经验、直觉和设计者的审美观，科学是基于工作负载驱动的评估。设计目标通常就是满足代价和性能指标，达到一个平衡的系统，从而没有个别资源成为性能瓶颈，同时每个资源只有很小的富裕能力。这一节通过第 4 章介绍的负载驱动评估方法，讨论一些关键的协议权衡。

5.4.1 方法论

我们的基本策略如下。如同第 4 章所描述的，让工作负载在一个多处理器体系结构的模拟器上执行。通过观察在模拟器中遇到的状态转换，能够确定各种事件的频率，诸如缓存扑空和总线事务。然后可以从其他设计参数，例如时延和带宽需求的角度来评估协议选择的效果。

按照第 4 章的方法选择参数，通过一组程序在四状态 Illinois MESI 协议上的执行，本节首先建立起基本的状态转换特性。然后解释如何用这些测得的结果，针对上述协议例子来获得一个关于设计权衡初步的量化分析，诸如 MESI 协议中独占状态的使用和 S→M 转换中 BusUpgr 事务（而不是 BusRdX 事务）的使用。本节还解释更传统的设计问题，诸如缓存块的大小（既是一致性也是通信的粒度）如何影响应用对于时延和带宽需要的。为了理解这个效果，我们将缓存扑空分为几种情况，诸如冷启动扑空、容量扑空和共享扑空，考察缓存块的大小对于不同情况的效果，并从应用特征的角度来解释结果。最后，也是从时延和带宽影响的角度，通过这种对应用的理解，来解释在基于作废和基于更新协议之间的权衡。

本节的分析是基于各种重要事件的频率，而不是基于它们所花的绝对时间（即性能）。这种做法在缓存体系结构的研究中是通行的，因为这样的结果不依赖于具体的系统实现和工艺的假设。然而，它应该只被看作是一种初步的分析，这是由于许多可能在真实系统中影响性能权衡的细节因素被抽象掉了。例如，记录状态转换的情况为计算扑空率和总线事务提供了一个方式，但如果没有真实的时延、开销和占用度的值，我们就不能将这些量转换成作用在系统上实际的带宽需求。为了得到带宽需求的一个估计，我们可能人为地假设每一次引用都花同样多的周期数完成。然而，带宽需求本身并不直接对应性能，只是通过增加竞争扑空的代价发生间接的影响。竞争是很难估计的，因为它不仅取决于所用的时序参数，还取决于流量的突发性，而这些在频率测量中是不可能捕获的。竞争、时序，以及性能也受到和低层硬件结构（例如队列和缓冲）相互作用和策略的影响。

306

用在本节的模拟不涉及竞争。它们只用一个简单的 PRAM 代价模型：所有存储操作的完成时间都假定是相同的（这里是单周期），无论它们是命中还是不命中缓存。对于这一点有三个主要原因。首先，这里关注的主要是理解协议固有的行为和与频率有关的权衡，不在于性能。第二，由于对不同的缓存块的大小和组织进行实验，不管具体的参数选择如何，希望从模拟器运行应用程序产生的引用交织是相同的；也就是说，所有协议和块的大小应该看到

相同的引用轨迹。由于这里用的是执行驱动，而不是踪迹驱动，只有使得模拟中所有的存储操作有同样代价，这才是有可能。否则，如果一次引用对于一个小缓存扑空，但对于一个大缓存命中的话，那么它在两种情形下表现出两种不同的延迟。于是，要确定哪些效果是协议固有的，哪些是由于所选择参数所导致的，就不是件容易的事情。第三，包含竞争效果的真实模拟要花更多的时间。用这种简单模型来采集频率的缺点是这种时序模型可能影响某些我们观察到的值；不过，这个影响对于我们研究的应用来说不大。

这里用来解释问题的工作负载是6个并行程序（来自SPLASH-2）和在第3、4章描述的一个多道程序。这些并行程序以批处理模式运行，对机器是独占的方式，并且在模拟中不包括操作系统的活动。所用的应用程序的数量相对较小，但这些应用主要是用来做解释用的，如第4章所描述的那样；这里所强调的是所选择的程序要能代表重要的计算类型，有着广泛变化的特性。这些应用中基本操作的频率如表5-1所示。现在更进一步来研究它们，以评价在缓存一致性协议中的设计权衡。

表5-1　由应用程序发生的每1 000次数据存储访问所导致的状态转换

应用			至				
			NP	I	E	S	M
Barnes-Hut	从	NP	0	0	0.0011	0.0362	0.0035
		I	0.0201	0	0.0001	0.1856	0.0010
		E	0.0000	0.0000	0.0153	0.0002	0.0010
		S	0.0029	0.2130	0	97.1712	0.1253
		M	0.0013	0.0010	0	0.1277	902.782
LU	从	NP	0	0	0.0000	0.6593	0.0011
		I	0.0000	0	0	0.0002	0.0003
		E	0.0000	0	0.4454	0.0004	0.2164
		S	0.0339	0.0001	0	302.702	0.0000
		M	0.0001	0.0007	0	0.2164	697.129
Ocean	从	NP	0	0	1.2484	0.9565	1.6787
		I	0.6362	0	0	1.8676	0.0015
		E	0.2040	0	14.0040	0.0240	0.9955
		S	0.4175	2.4994	0	134.716	2.2392
		M	2.6259	0.0015	0	2.2996	843.565
Radiosity	从	NP	0	0	0.0068	0.2581	0.0354
		I	0.0262	0	0	0.5766	0.0324
		E	0	0.0003	0.0241	0.0001	0.0060
		S	0.0092	0.7264	0	162.569	0.2768
		M	0.0219	0.0305	0	0.3125	839.507
Radix	从	NP	0	0	0.004746	3.524705	11.41111
		I	0.130988	0	0	1.108079	4.57868
		E	0.000759	0.002848	0.080301	0	0.00019
		S	0.029804	1.120988	0	178.1932	0.817818
		M	0.044232	11.53127	0	4.03157	802.282
Raytrace	从	NP	0	0	1.3358	0.15486	0.0026
		I	0.0242	0	0.0000	0.3403	0.0000
		E	0.8663	0	29.0187	0.3639	0.0175
		S	1.1181	0.3740	0	310.949	0.2898
		M	0.0559	0.0001	0	0.2970	661.011

（续）

应用			至				
			NP	I	E	S	M
Multiprog User Data References	从	NP	0	0	0.1675	0.5253	0.1843
		I	0.2619	0	0.0007	0.0072	0.0013
		E	0.0729	0.0008	11.6629	0.0221	0.0680
		S	0.3062	0.2787	0	214.6523	0.2570
		M	0.2134	0.1196	0	0.3732	772.7819
Multiprog User Instruction References	从	NP	0	0	3.2709	15.7722	0
		I	0	0	0	0	0
		E	1.3029	0	46.7898	1.8961	0
		S	16.9032	0	0	981.2618	0
		M	0	0	0	0	0
Multiprog Kernel Data References	从	NP	0	0	1.0241	1.7209	4.0793
		I	1.3950	0	0.0079	1.1495	0.1153
		E	0.5511	0.0063	55.7680	0.0999	0.3352
		S	1.2740	2.0514	0	393.5066	1.7800
		M	3.1827	0.3551	0	2.0732	542.4318
Multiprog Kernel Instruction References	从	NP	0	0	2.1799	26.5124	0
		I	0	0	0	0	0
		E	0.8829	0	5.2156	1.2223	0
		S	24.6963	0	0	1 075.2158	0
		M	0	0	0	0	0

注：除 Multiprog 用 8 个处理器外，这些数据都假设 16 个处理器、1 MB 四路组相联高速缓存、64 字节高速缓存块，Illinois MESI 一致性协议。

5.4.2　在 MESI 协议下的带宽需求

如第 4 章所述，我们从容量为 1 MB、每个处理器只有一级缓存的情况开始。对于问题的缺省规模来说，这种缓存能够容纳其重要的工作集，对于所有应用这是现实的情形。我们用四路组相联（LRU 替换）以减少冲突扑空并取高速缓存块的大小为 64 字节，这是实用的。让这些工作负载通过模拟 Illinois MESI 协议的高速缓存模拟器，我们得到如表 5-1 所示的状态转换频率。其中数据指的是由处理器发出的每 1000 次访问所导致的状态转换数。注意，在这个表中，我们引入了一个新的状态 NP（未现态）。它帮助表现当缓存扑空时的状态转换，即一个缓存块被替换（创建一个从 I、E、S、M 到 NP 的转换），一个新的块被带进来（创建一个从 NP 到 I、E、S、M 的转换）。尽管我们表现的是每 1000 次访问的情形，但状态转换的总和可以大于 1000，这是因为有些访问会引起多个状态转换。例如，一次写扑空，除了在其他缓存中引起的作废转换（I/E/S/M→I）外，还可以在本地缓存引起两个转换（例如，对于旧缓存块是 S→NP，对于新进来的缓存块是 NP→M）。这个状态转换频率数据对于回答“如果……怎样”类型的问题是很有用的。例 5.8 告诉我们如何来确定这些负载带给存储系统的带宽需求。

307 ~ 309

例 5.8　假设整数密集型应用以每处理器 200 MIPS 持续运行，浮点密集应用为 200 MFLOPS。假设缓存块的传送涉及 64 字节的数据（在数据总线）和每个总线事务涉及 6 字节的命令和地址（在地址总线），问每个处理器产生的流量是多少？

解答：第一步是计算每条指令的流量。对于每个可能的状态转换，我们确定所进行的总

线动作，从而得知和每一事务相关联的流量。例如，M→NP 转换指示的是，由于扑空一个已修改的高速缓存块要被回写。类似地，S→M 转换表示一个更新请求要被放到总线上。作为对一种总线事务（例如，M→S 或 M→I 事务）的响应，将一个已修改的缓存块刷新也导致一个 BusWB 事务。所有可能转换的总线事务如表 5-2 所示。除 BusUpgr 只是产生地址流量外，所有事务都产生 6 字节地址总线流量和 64 字节数据流量。[⊖]■

表 5-2　Illinois MESI 协议中响应状态转移的总线活动

		至				
		NP	I	E	S	M
从	NP	—	—	BusRd	BusRd	BusRdX
	I	—	—	BusRd	BusRd	BusRdX
	E	—	—	—	—	—
	S	—	—	Not possible	—	BusUpgr
	M	BusWB	BusWB	Not possible	BusWB	—

我们现在能计算所产生的流量。用表 5-2，我们可以将表 5-1 中的每 1000 次存储访问的状态转移变换到每 1000 次存储访问的总线事务，并通过乘以每个事务的流量，将这些转换为地址和数据流量。然后，用表 4-1 中存储访问的频率，我们可以将它转换为每条指令的流量。最后，乘以假设的处理速率，我们就得到关于每个应用的地址和数据的带宽需求。这个计算方法对于每个应用的结果对应于图 5-18 直方图中每一组的最左项。

先前例子中的计算给出了平均带宽需求，但假设了总线带宽足以使处理器以全速来执行它们的任务（实际上，带宽局限可能使处理器和事件都慢下来，从而反过来导致单位时间的流量降低）。这个计算，对于得到一个系统在不饱和的前提下所能支持的处理器数提供了一个有用的基础。例如，像 SGI Challenge 这样的机器，有 1.2 GBps 数据带宽，对于我们的那些应用来说（除 Radix 外），总线提供了足以支持 16 个处理器的平均带宽。一种典型的经验规则是留出 50%的“余量”，来应付数据传送的突发性。如果 Ocean 和 Multiprog 工作负载被排除在外，这个总线则可能支持到 32 个处理器。如果带宽不足以支持某个应用，这个应用的执行就要慢下来。这样，我们可以预料到 Radix 加速比的曲线随处理器数的增加会很快扁平下来。通常，一个多处理器系统用于各种不同的工作负载，其中许多对带宽的要求不高，因此设计者要选择支持一种处理器规模，能在要求最高的应用上使总线处于充分利用的状态。

5.4.3　协议优化的影响

给定这种基本的设计要点，我们就能在常用机器参数假设下来评估协议的权衡了。下面就是一个例子。

例 5.9　在本章里，我们已经描述了两种作废式协议——基本三态 MSI 协议和 Illinois MESI 协议。关键的区别在于 MESI 协议包含了独占状态。这个 E 状态对带宽的节省到底有多

⊖ 对于 Multiprog 工作负载，为了提高模拟的速度，在把指令传送到 1 MB 的统一的指令和数据高速缓存之前用了一个 32 KB 的指令高速缓存，作为一个过滤器。对于指令访问的状态转换频率的计算只是基于在 L_1 指令缓存中扑空的访问。这种过滤不影响我们计算数据流量的方式，但它意味着指令流量的计算是不同的。除此以外，对 Multiprog，我们分别给出内核指令、内核数据访问、用户指令和用户数据访问。一次访问可能产生多种用户数据和内核数据类型的状态转移。例如，如果一条内核指令扑空引起一个已修改的用户数据块的回写，那么我们将有针对内核指令的转移 NP→E/S 以及针对用户数据访问的转移 M→NP。

大呢？

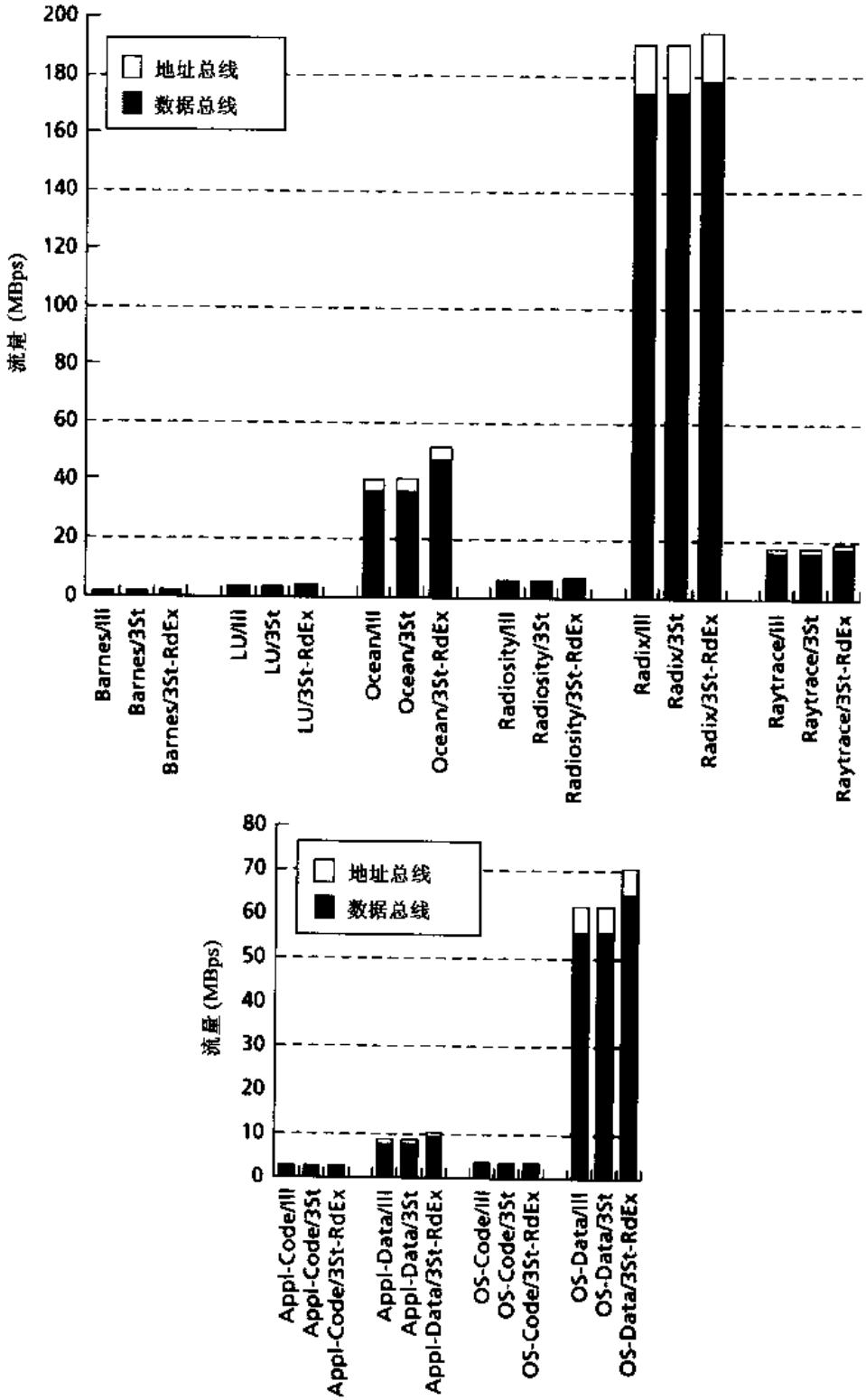


图 5-18 假定处理器计算性能为 200 MIPS/MFLOPS，且每处理器有 1 MB 高速缓存，这里表现的是各种应用从每个处理器产生的带宽需求。上边的直方图表示并行程序的数据，下边的图表示从 Multiprog 的模拟得到的数据。数据流量和地址（包括命令）流量分别表示。每一组中最左边的项表示 Illinois MESI 协议（Ill）；中间项表示的是基本三态作废协议（3St），即没有 E 状态；最右边的项表示在 S→M 转换时用 BusRdX 而不是 BusUpgr 的三态协议（3St-RdEx）

解答：E状态的主要优点是当从E→M转换时没有流量产生。而一个三态协议会产生一个BusUpgr事务，以获得对于存储块的独占拥有权。计算带宽的节省，我们只要在表5-2中对于E→M转换放一个BusUpgr，然后和以前一样重新计算流量，图5-18各组的中项就是所得到的带宽需求。■

310 - 312

例5.9说明了对于一个比较复杂的设计，直觉的推理可能经不起工作负载的量化测试。这对于Multiprog也是对的，尽管它基本上是由顺序程序构成，看起来应该是受益最大的。这种收益微不足道的主要原因是表5-1中E→M转换的份额相当小（即由于读扑空以独占状态装载的存储块并不经常在同一状态被写）。另外，在三态协议中对于S→M转换所需要的BusUpgr事务，只是用到6个字节的地址流量，而没有数据流量。例5.10考察了BusUpgr事务的优点。

例5.10 回顾即使在三态MSI协议中，一个对于共享状态存储块的写操作在总线上会产生一个BusUpgr请求，而不是BusRdX。这是节省带宽的，因为对BusUpgr没有数据传送的需要。但如我们将要看到的，它使实现复杂了。这里的问题是，这种额外的复杂性能使我们节省多少带宽？

解答：要计算不太复杂的实现和一个三态协议的带宽，我们只要在表5-2中对E→M和S→M转换放上BusRdX（在三态MSI协议，这都是S→M转移），然后重新计算带宽数。结果由图5-18直方图各组最右项所示。尽管对大多数应用来说，带宽的差别是不大的，但Ocean和Multiprog内核数据访问表明，这个差别可能大到10%～20%。■

在带宽需求方面的差别对性能的影响取决于总线事务是如何实现的。然而，这种高层分析指出了在何处需要有更细致的评估。

最后，就像在第4章所讨论的，对于所用的输入数据集的大小，在较小的缓存上也运行Ocean、Raytrace和Radix应用是重要的，因为那可以模拟重要的工作集在缓存层次中放不下的情形。我们在此用64KB的高速缓存，对于这些问题的规模，除了最大的工作集外，它能适应所有其他的工作集。对于这种情形粗略的状态转换数据如表5-3所示，图5-19示出按处理器的带宽需求。就像所看到的，如果由于容量性扑空导致一个关键的工作集不能放到处理器的缓存中，对总线带宽的需求可能大大增加。1.2 GBps的总线现在对于Ocean和Radix只能勉强支持4个处理器，而对Raytrace可支持到16个处理器。

表5-3 在高速缓存容量较小的情况下，应用程序每发出1000次存储访问所导致的状态转换

应用			至				
			NP	I	E	S	M
Ocean	从	NP	0	0	26.2491	2.6030	15.1459
		I	1.3305	0	0	0.3012	0.0008
		E	21.1804	0.2976	452.580	0.4489	4.3216
		S	2.4632	1.3333	0	113.257	1.1112
		M	19.0240	0.0015	0	1.5543	387.780
Radix	从	NP	0	0	9.440787	2.557865	27.36084
		I	4.354862	0	0.00057	0.157565	1.499903
		E	8.148377	0.001329	140.9295	0.012339	0.126621
		S	3.825407	0.481427	0	102.4144	0.484464
		M	23.03084	5.629429	0	2.069604	717.1426

（续）

应用			至				
			NP	I	E	S	M
Raytrace	从	NP	0	0	7.2642	3.9742	0.1305
		I	0.0526	0	0.0003	0.2799	0.0000
		E	6.4119	0	131.944	0.7973	0.0496
		S	4.6768	0.3329	0	205.994	0.2835
		M	0.1812	0.0001	0	0.2837	660.753

注：这里的数据假设 16 个处理器、64 KB 四路组相联高速缓存、64 字节的高速缓存块，Illinois MESI 一致性协议。

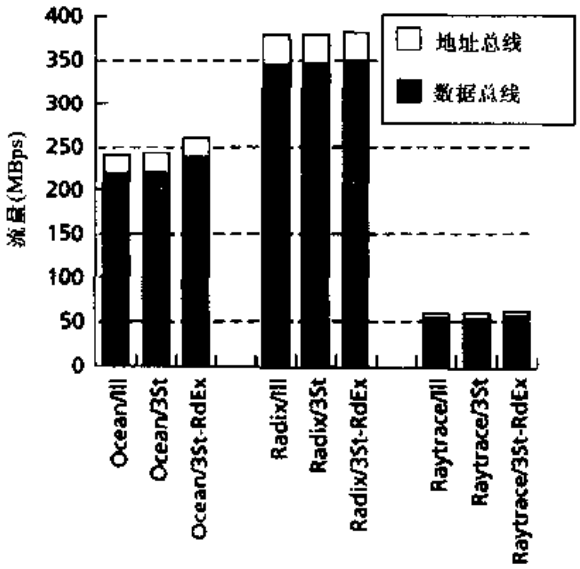


图 5-19 假定处理器计算性能为 200 MIPS/MFLOPS，且每处理器有 64 KB 高速缓存，这里表现的是各种应用从每个处理器产生的带宽需求。数据流量和地址（包括命令）流量分别表示。每一组中最左边的项表示 Illinois MESI 协议；中间项表示的是基本三态作废协议，即没有 E 状态（如 5.3.1 节所述）；最右边的项表示在 S→M 转换时用 BusRdX，而不是 BusUpgr 的三态协议

5.4.4 高速缓存中存储块大小的权衡

在所有现代计算机中，缓存的组织是一个关键的性能因素，对于多处理器尤其如此。在单处理器场合，缓存扑空典型地被分为“3C”，即强制（compulsory）扑空，容量（capacity）扑空和冲突（conflict）扑空（Hill and Smith 1989；Hennessy and Patterson 1996）。强制性扑空，也称为冷启动扑空，发生在处理器对一个存储块的首次引用时。容量扑空是由于在程序执行过程中处理器所要引用的所有块没法放到高速缓存中（即使是全相联），因此某些块要被替换，后来还要再被访问。冲突或碰撞扑空发生在非全相联的缓存中，当一个程序引用的应映射到同一缓存块的集合在一组中装不下的时候；它们在全相联缓存中是不会发生的。许多研究成果都是针对缓存大小、相联度和缓存块大小是如何影响这几种扑空的。

从系统结构来说，容量扑空可以通过加大缓存来减少，冲突扑空可以通过增加缓存的相联度或者增加可以映射的线数来减少（增加缓存规模，减少块大小）。冷启动扑空只能通过增加块的大小来减少，因为那样一次冷启动扑空将带进来较多的数据，那些数据可能紧接着被访问到。在单处理器中缓存设计的挑战性在于这些因素相互影响。例如，对于固定的缓存容量，增加块的大小将减少块的数量，因此减少冷启动扑空可能的代价是增加冲突扑空。同样，在缓存组织上的变化可能影响扑空所带来的惩罚或者命中时间，从而可能影响到处理器的周期。

313 314

缓存一致性多处理器引入第四种扑空：一致性扑空。它们发生在数据块在多个缓存中的共享。这类扑空又分两种：真共享扑空和伪共享扑空。真共享扑空发生在由一个处理器写的数据字要被另一个处理器读或者写的时候。伪共享扑空指的是，不同的数据字被不同的处理器访问，但它们处于同一个存储（缓存）块中，并且至少有一个访问是写。缓存块的大小不仅是对主存进行数据访问的单位（粒度），它也是相容性的粒度。即一个处理器的写，另外处理器缓存中的整个块都被作废，而不仅是写操作涉及的那个存储字。

更精确地讲，真共享扑空发生在一个处理器向缓存中的某些字做写操作时，作废另一处理器中缓存的相应块，在那之后后者要从修改的字中读。之所以称为是“真”共享扑空，是因为这个扑空真正涉及到了共享的数据；无论和机器组织或者粒度有什么相互作用，对这种扑空的处理对于程序的正确性是必须的。另一方面，当一个处理器向一个缓存块中写入某个字，然后另一个处理器读（或者写）同一缓存块的另一个字，即使没有有用的数据在处理器之间交流，块的作废和随后的缓存扑空也会发生。这样的扑空就称为伪共享扑空（Dubois et al. 1993）。随着缓存块的增大，处于同一存储块中不同的变量被不同的处理器访问的概率增加。如果对这些变量的访问是写操作，则伪共享扑空的可能性也就增加。如果存储块的大小只是一个字，仍然可能有真共享扑空，但不会有伪共享扑空。工艺的进步趋向于较大的缓存块（例如，DRAM 的组织和访问模式以及需要通过分摊开销以获得高的数据传送带宽），因此理解伪共享扑空的潜在影响以及如何能避免它们是重要的。

315

真共享扑空是给定的并行分解和分配所固有的，因此如同冷启动扑空，减少它们的惟一办法是增加存储块的大小和增加通信数据的空间局部性。另一方面，由于是体系结构的相互作用所引起的，伪共享扑空是第 3 章所讨论的附加通信的例子。同真共享和冷启动扑空相比，伪共享扑空能够通过减小块的大小来减少，还可以通过软件（调整）和硬件方面的一些其他优化措施来减少，对此，在后面将会有讨论。这样，决定最好的缓存块的大小就具有根本性的意义了，而这件事只能通过针对真实程序的评测才能解决。

1. 缓存扑空的一种分类

图 5-20 中的流程图给出了一个具体的算法来对于缓存一致性多处理器中的缓存扑空进行分类[1]。理解其中的细节现在还不是很重要，对于本章后面的内容来说，只要理解前面的定义就够了，但对细节的理解能够加强我们的观念，并且是一个有用的练习。在这个算法中，一个存储块在缓存中的生存时间定义为它在缓存中处于有效状态的时间区间，也就是说从引起它装载进缓存的扑空开始到它被作废，替换或者程序结束。当一次扑空出现的时候，还说不清楚它应该归于哪一类；只是当相应的存储块被替换或者作废时，我们才能明确，因为只有在那个时刻我们才知道在该存储块生存期是否发生了真共享或者仅仅是伪共享。让我们首先考虑简单的情形。情形 1 和 2 是直接的冷启动扑空，发生在事先没被写过的块上。情形 7 和 8 反映了在一个块上的伪共享和真共享，被共享的那个块在缓存中先被作废，然后被另一块替换。共享类型的确定是通过考察在生存期作废后是否有特定的字被修改了。情形 9 是简单的容量性（或者冲突）扑空，这是由于存储块先被替换并且自从上次访问后其中的字还没有被修改。所有其他的情形反映的是一些组合因素的扑空。例如，由于这个处理器以前

[1] 在这种分类中，我们不区别容量扑空和冲突扑空，主要考虑到它们都是可用资源（缓存组或整个缓存）用尽后表现出来的现象，而强调它们之间的差别并不给多处理器问题的研究增加什么有意义的结论。

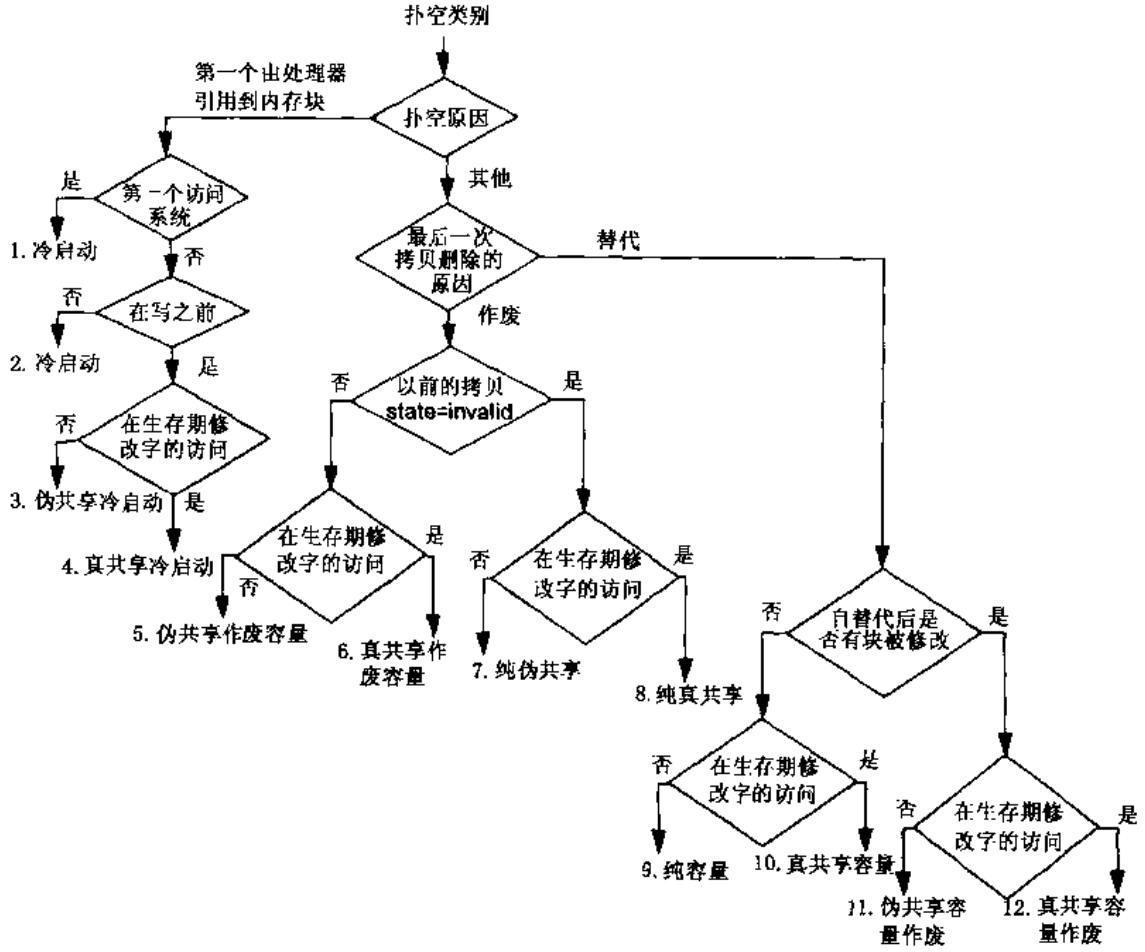


图 5-20　共享存储多处理器高速缓存扑空的一种分类。按照这种分类法，四种基本的缓存扑空是冷启动扑空、容量扑空、真共享扑空和伪共享扑空（冲突扑空在这里被看成是容量扑空）。由于一次扑空可能有多种原因，人们还提出了一些混合类型。例如，一个存储块可能先在处理器 *A* 的缓存中被替换，然后由处理器 *B* 写，然后又由 *A* 读回去，于是造成了一次容量-cum-作废伪/真共享扑空。这在分类中可能被称为“伪/真共享容量-作废”，由于共享优先级高，还由于替换在作废之前发生（图中的情形 11 和 12）。如果存储块先在 *A* 的缓存中作废，然后这个无效块被替换，然后再被 *A* 读出，它就要被称为“伪/真共享作废-容量”（情形 6 和 7）。按照我们给出的四种类型来说，上面这些都属于真或伪共享扑空。注意：图中的问题“已修改的字在生命周期被访问过?”问的是自从上一次“根本的一致性”扑空后已修改的存储块是否在当前生命周期里被访问过了，其中“根本的一致性”对应于第 4、6、8、10 和 12 类。这只有在当前生命期结束时才能决定

从来没有访问过这个块，情形 4 和 5 是冷启动扑空；然而，一些其他的处理器写过这个块，因此也有共享（假或真）。类似地，我们可能在那些曾经由于容量或者冲突被替换的存储块上发生假的或真共享。例如，如果一个扑空的发生是由于伪共享和容量问题，那么通过减小块的大小可能消除伪共享，但可能并不能消除扑空。另一方面，共享扑空在某种意义上要比容量扑空更基本，这是因为共享扑空的存在是与缓存容量无关的，因此我们在多原因扑空中给予它们优先级。在我们的分类结果中，所有名称带有真共享的扑空被称为根本性一致性扑空。即使缓存有无限大、单字块、所有数据预装在缓存中了（即没有了冷启动扑空）它们还是会出现的。例 5.11 解释了扑空分类中的这些定义。

316
317

例 5.11　假设三个处理器 P_1、P_2、P_3，发出存储器操作，如表 5-4 中的头几列所示（第

一列指出虚拟时间或步子)。用扑空分类算法对最后一列的扑空进行分类。假定每个处理器的缓存只有一个4字的缓存块，所有缓存最初都是空的。

解答：结果如表5-4所示。■

表5-4　对源于三个处理器的访问流产生的扑空分类

次数	P_1	P_2	P_3	扑空类别
1	ld w0		ld w2	P_1 和 P_3 扑空；类别将在后面说明
2			st w2	$P_{1,1}$：纯冷扑空，$P_{3,2}$：提升
3		ld w1		P_2 扑空；类别将在后面说明
4		ld w2	ld w7	P_2 命中；P_3 扑空；$P_{3,1}$：冷扑空
5	ld w5			P_1 扑空
6		ld w6		P_2 扑空；$P_{2,3}$：冷真共享扑空（w2被访问）
7		st w6		$P_{1,5}$：冷扑空；$P_{2,7}$：提升；$P_{3,4}$：纯冷扑空
8	ld w5			P_1 扑空
9	ld w6		ld w2	P_1 命中；P_3 扑空
10	ld w2	ld w1		P_1，P_2 扑空；$P_{1,8}$：纯真共享扑空；$P_{2,6}$：冷扑空
11	st w5			P_1 扑空；$P_{1,10}$：纯真共享扑空
12			st w2	$P_{2,10}$：容量型扑空；$P_{3,11}$：提升
13			ld w7	P_3 扑空；$P_{3,9}$：容量型扑空
14			ld w2	P_3 扑空；$P_{3,13}$：作废容量型假共享扑空
15	ld w0			P_1 扑空；$P_{1,11}$：容量型扑空

注：如果在同一行列有多次引用，我们假定 P_1 在 P_2 之前，P_2 在 P_3 之前。记号 ld/st wi 表示对字 i 的装入/存放。w1 到 w4 在相同的缓存块，等等。记号 $P_{i,j}$ 指向由处理器 i 在第 j 行发出的存储访问

2．缓存块的大小对于扑空率的影响

将图5-20的算法应用于一种负载的模拟运行结果，我们可以确定各种扑空发生在程序中的频率，以及这些频率如何随着缓存组织的变化而改变，例如块大小的变化。图5-21表示各种应用运行在16个处理器上扑空的分解，这些处理器都有一个1 MB的4路组相联缓存，缓存块的大小在8～256字节之间变化。直方图表现4种基本的扑空：冷启动扑空（情形1和2），容量（包括冲突）扑空（情形9），真共享扑空（情形4，6，8，10，12），以及伪共享扑空（情形3，5，7和11）。除此以外，它们还表现了升级的频率——也就是那些发现块在缓存中处于共享状态的写操作。更新不同于其他类型的扑空，由于缓存已经有了有效的数据，只是需要独享的所有权。虽然没有包含在图5-20的分类方案中，它们仍然被看作是扑空，这是由于它们在处理器互连机构上产生流量，从而可能阻滞处理器的执行。

对于每个独立的应用，扑空特征随块大小变化，其情况和我们对程序和扑空类别的理解一致。冷启动、容量和真共享扑空倾向于随块的变大而减少，这是因为随着扑空所带进来的附加数据在该块被替换前被访问。在所有情形下，真共享在扑空中占有明显比例，于是即便是理想的、无限大的缓存，扑空率和总线带宽将都不会为零。然而，整个特征对于不同的程序大相径庭。例如，真共享部分的大小变化范围可能是很大的。某些应用在伪共享中表现出随块大小实质性的增加，而另一些则几乎没有。进一步看，这个图只是表现了针对缺省数据集的情况。在实际中，得到关于一个应用程序的伪共享或空间局部性之前，考察结果随输入数据规模和处理器数的扩展变化是重要的（见第4章）。下面让我们来研究应用的性质，它们反映了在机器层次看到的扑空特征方面的差别，这使我们能够定性地理解扩展的效果。

318

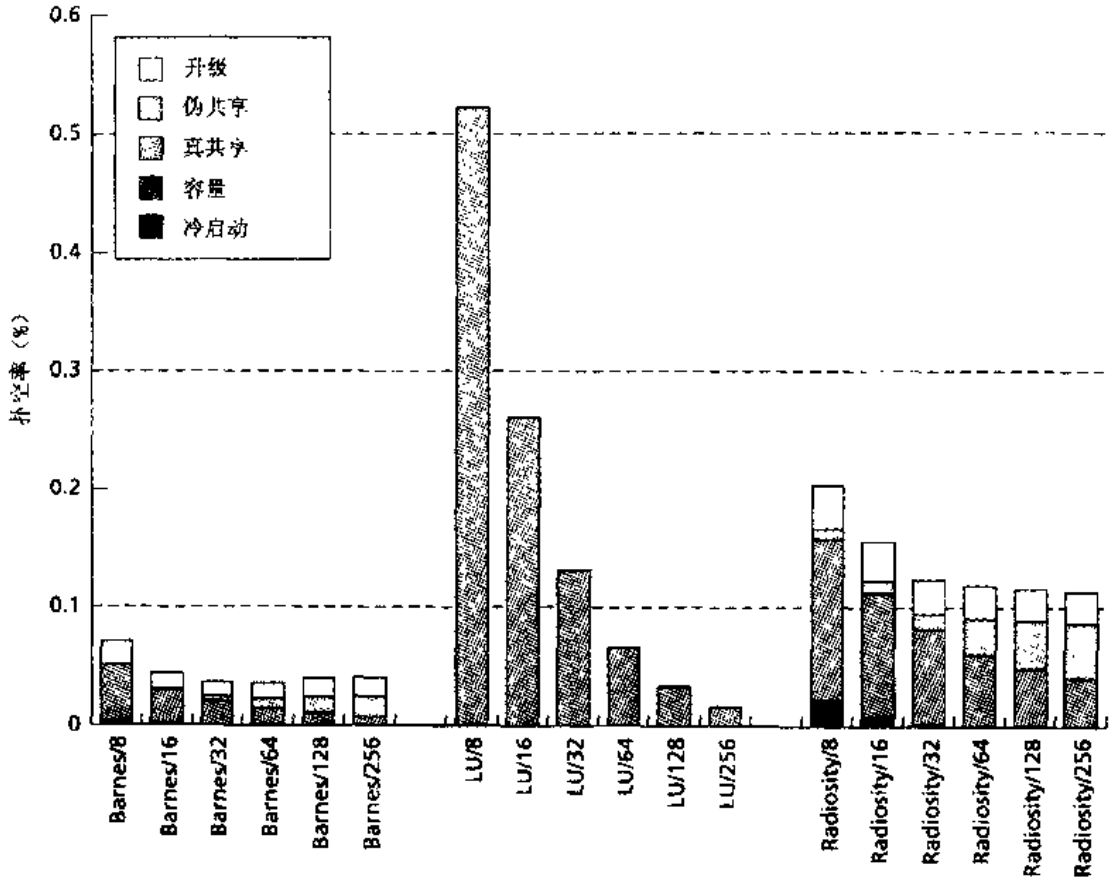


图 5-21a　在 1 MB 缓存的处理器上，Barnes-Hut、LU、Radiosity 应用在不同缓存块大小下的扑空情况。冲突扑空含在容量扑空中。应用不同，扑空的情形变化很大，但我们还是能够看到一些共同的特点。冷启动扑空和容量扑空随缓存块大小的增加很快下降，这是由于空间局部性得到了利用。真共享扑空也随之减少，但伪共享扑空增加。对小缓存块来说，伪共享扑空的成分通常较小，但有时会很快增加。更新所导致的情形是直方图顶部的非阴影部分，可以忽略不计

3．同应用程序结构的关系

含有多个字的缓存块通过预取所访问地址附近的数据，使程序的空间局部性得到利用。当然，超出某个点，较大的缓存块反而会对性能有不好的影响，原因在于：1）预取了不需要的数据；2）引起冲突扑空增加，这是由于对固定的缓存规模来说，这样就减少了缓存块的个数；3）引起伪共享扑空增加。并行程序中的空间局部性一般低于串行程序，这是因为当一个存储块带到缓存时，其中有些数据可能属于另一个处理器，并且将不会被完成扑空的处理器用到。作为一个极端的例子，某些并行程序将数组中相邻元素赋给不同的处理器，虽然可能取得好的负载平衡，但在进程中就大大地降低了程序的空间局部性。

图 5-21 中的数据表示，即使在并行情况下，LU 和 Ocean 有好的空间局部性并且没有伪共享。扑空率的许多成分随缓存块大小的增加按比例下降，伪共享扑空基本上是不存在的。这在很大程度上是由于这些基于数组的代码用到了和体系结构相匹配的数据结构，如同第 3、4 章所讨论的那样。例如，Ocean 中的一个网格不是由一个二维数组（二维数组在面向列划分的边界引起严重的伪共享），而是由一个四维数组来表示的：一个二维块的数组，每一个块自己也是一个二维数组。这样通过程序或者编译的结构化，保证了多数访问是单位跨距的，覆盖大量连接数据，于是表现出很好的结果。

319
320

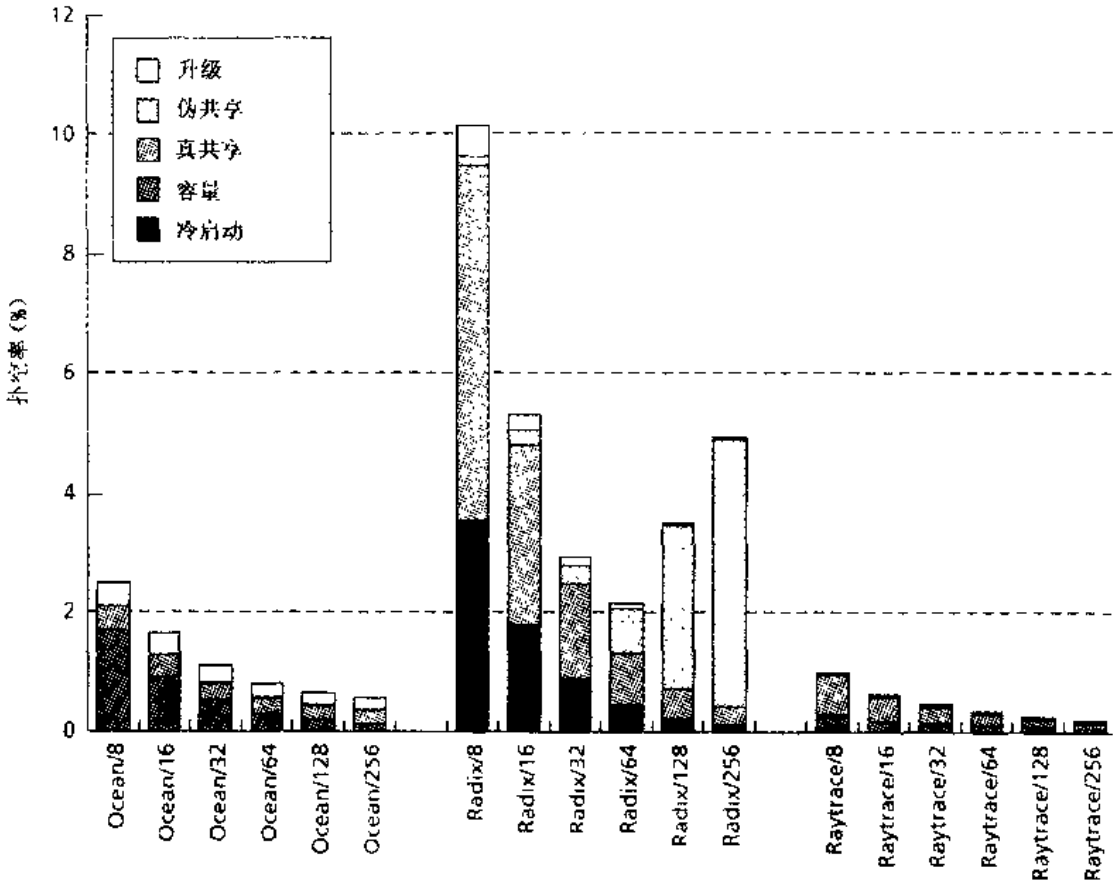


图 5-21b 在 1 MB 缓存的处理器上，Ocean、Radix、Raytrasce 应用在不同缓存块大小下的扑空情况

在 Ocean 中，容量性扑空是值得注意的，但它们只是反映在一个进程划分的内部元素上，因此它们有非常好的空间局部性。和 LU 的一个区别是，Ocean 中的真共享扑空展示不出这样好的空间局部性。大多数真共享扑空反映在相邻划分的边界元素上。在面向行的边界安排下，由于被访问的数据在地址空间中是连续的，于是能表现出好的空间局部性。然而，当一个处理器访问面向列边界的一个元素时，要涉及到它的相邻划分的内部元素的一整个缓存块，其中大部分数据将不被使用，因此就浪费了。由于容量性扑空对于这个问题和这种机器配置来说不是很大的，整个空间局部性的限制在于真通信的要求。在 LU 中，即使真通信也是每次 $B \times B$ 连续块，因此空间局部性即使在真共享扑空情况下也是很好的。

321

至于扩展性，当问题的规模和处理器个数增加时，这两个应用的空间局部性都保持得相当好，没有伪共享（至少在划分变得过分小之前）。即使对于大于 256 字节的缓存块，至少对于 LU 应该是这样的。在 Ocean 中，容量相对于真通信扑空（因此也就是空间局部性）的变化在很大程度上取决于数据规模和处理器数的相对变化关系。

图形应用程序 Raytrace 所表现出来的伪共享基本可以忽略，但它在空间局部性方面表现得明显的不好。伪共享小是因为主要的数据结构（构成场景的多边形的集合）是只读的。读写共享只是发生在图像平面数据结构和任务队列上，但那是在控制范围内的并且对大多数问题来说相比很少。这种真共享扑空率可通过增加缓存块的大小来减少。至于较差的容量扑空的空间局部性问题（尽管在这种配置中总的量是小的），其原因是对于多边形集合的访问模式是相当任意的，主要由于一束光线在其中反射的对象集合不可预测。在扩展性方面，随着

问题规模的扩大（大多数情形下意味着更多的多边形），基本的效果可能是更大的容量性扑空率；在单个部分内的空间局部性应该没什么变化。除能在图像平面和任务队列中的数据结构中看到更多的共享外，从许多方面来看，处理器数增大的效果类似于问题规模的变小。

Barnes-Hut 和 Radiosity 应用表现出中等的空间局部性和伪共享。这些应用用到了复杂的数据结构，包括树型编码的空间信息和数据，其中分配给每个处理器的记录在存储器中不是连续的。例如，Barnes-Hut 的操作作用在存放在数组中的粒子记录上，随着应用程序的执行，粒子在物理空间中运动，粒子记录可能要重新分配给不同的处理器，数组中相邻的粒子在一段时间后很可能属于不同的处理器。空间局部性在一个粒子记录内体现得很好，但在跨记录之间体现得就不好。在缓存块较大的情况下，伪共享成为一个问题，其原因是多方面的。首先，不同的处理器可能对共享同一个缓存块的不同记录进行写操作。其次，一个粒子的数据结构（记录）包含有某些数据域，要被在本阶段拥有它的处理器修改（例如，在计算阶段作用在粒子上的力），还包含有数据域要被其他处理器读，但在本阶段不被修改（例如，当前粒子的位置）。由于这两个域可能落入同一个缓存块，就要导致伪共享。通过按照数据域的访问模式拆分粒子数据结构，有可能消除这样的伪共享；但由于扑空率总体上很小，人们并没有这样做。随着问题的规模和处理器个数的变化，Barnes-Hut 的扑空率情况也不会有太大变化。这是由于工作集改变得很慢（和粒子数的对数成比例，这一点和 Ocean、Raytrace 不同），空间局部性是由一个粒子记录的大小确定的，因此保持不变。同时，伪共享的来源对于处理器数来说并不敏感。Radiosity 要复杂得多，对于较大的数据集合和较多的处理器，它的行为难以推测；惟一的办法是收集表现趋势的试验数据。 322

Radix 显示的共享情形最差，即使对于 1 MB 的缓存来说，它不仅有很高的扑空率（由于冷启动和真共享扑空），而且由于在 128 字节或更大缓存块情形的伪共享扑空变得更差。Radix 中的这种伪共享效果如第 4 章所示。现在来考察它的具体情况。考虑要对 256 K 个键排序，使用 1 024 为基和 16 个处理器。平均来说，这导致每个基每个处理器有 16 个键（64 字节数据），然后它们写到一个全局数组的一个连续的部分，起始点不可预测。在这个数组中相邻的 64 字节区域被不同的处理器写。如果缓存块大于 64 字节，伪共享的可能性显然就很大。随着问题规模的增加，我们将清楚地看到伪共享变小。增加处理器数的效果恰好是相反的。Radix 突出地表明，仅仅通过看问题的规模和处理器数来得到关于伪共享和空间局部性的结论是不够的。重要的是要理解这些结果是如何依赖于实验中所选择的关键参数，并且这些参数在实际中可能如何变化。

Multiprog 工作负载在 1 MB 缓存上运行的数据如图 5-22 所示。其中分别表示了用户代码、用户数据、内核代码和内核数据。对于代码来说，只有冷启动和容量性扑空。进而看到操作系统中数据引用的空间局部性并不怎么好。在某种程度上，这对于应用程序的数据也如此，这是因为 gcc（在 Multiprog 中引起扑空的主要应用）用到了大量的链表，而这种数据结构的空间局部性不好。有意思的是，尽管我们只是运行串行程序，我们仍然看到了真共享扑空。这是由于进程的迁移所导致的（操作系统为资源管理所作的工作），进程从一个处理器迁移到另一个处理器，引用它曾经在另一个处理器写入的存储块。冷启动和容量扑空中的空间局部性是合理的，对于内核数据来说真共享扑空一点也不减少。这种情形的一个原因可能是操作系统本身还不是一个好的并行程序。 323

最后，让我们考察 Ocean、Radix 和 Raytrace 在 64 KB 缓存上的行为。所导致的扑空率如

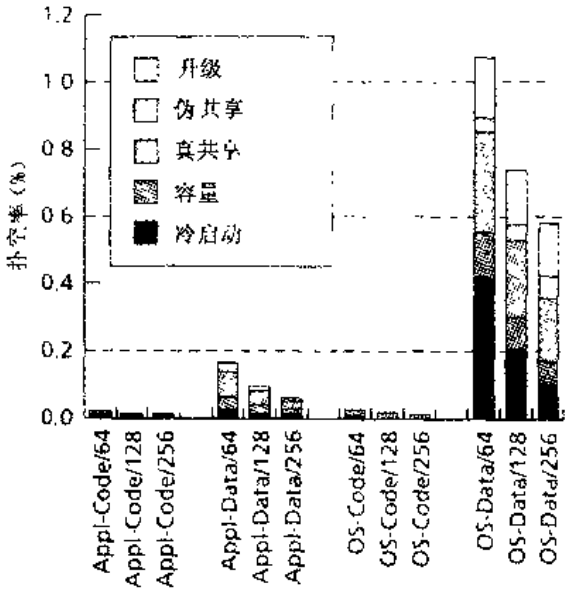


图 5-22　在 1 MB 缓存的处理器上，Multiprog 应用在不同缓存块大小下的扑空情况。这里的结果基于 1 MB 缓存，对真共享扑空来说，空间局部性对这个应用要比对操作系统好得多

图 5-23 所示，正如所预想的，整体扑空率较高并且容量性扑空增加较明显。缓存块大小对于真共享和伪共享扑空的效果和 1 MB 缓存相比没有实质性的不同，这是由于这些性质是直接针对程序的并行分配和性能调试策略的，和缓存的大小关系不大。不过，容量性扑空的行

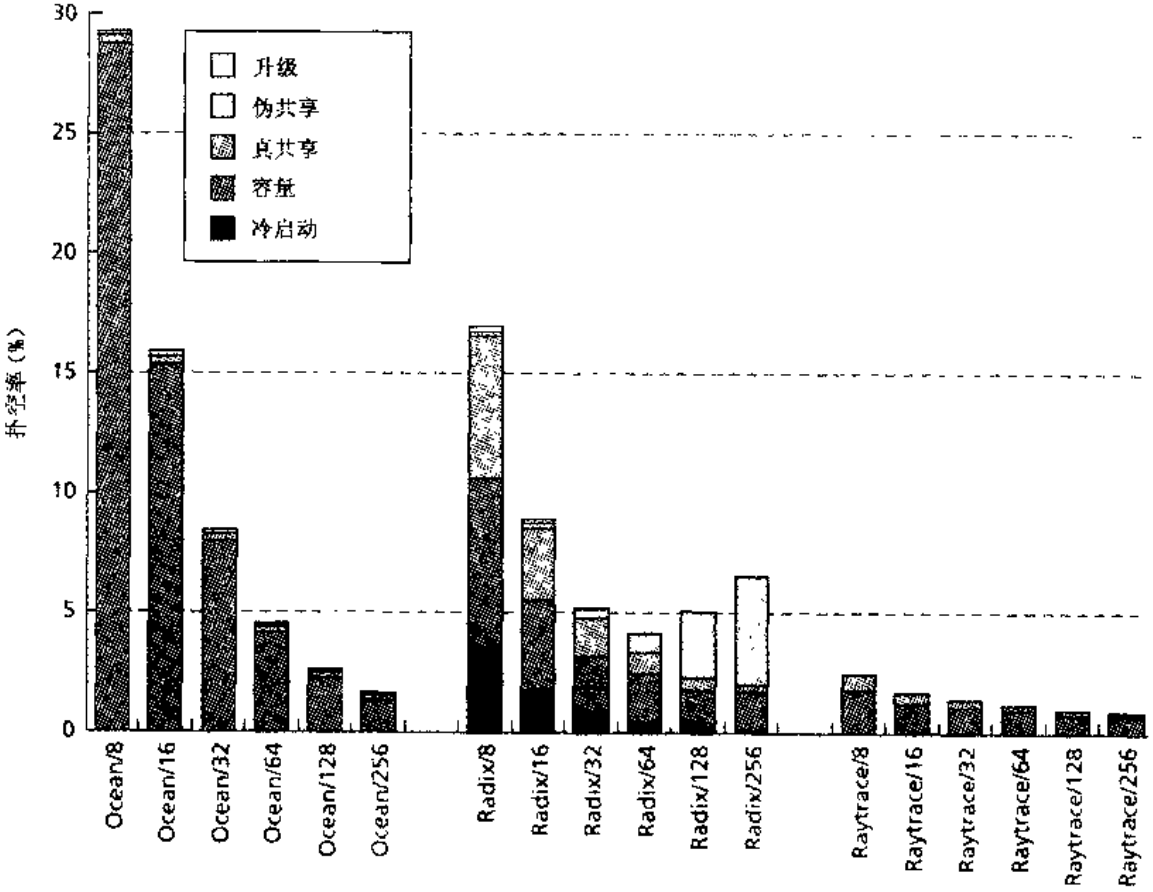


图 5-23　在 64 KB 缓存上，应用程序在不同缓存块大小下的扑空情况。容量扑空现在在整个扑空率中占有较大的比例。随缓存块大小的增加，不同的应用表现出的容量性扑空率下降的情况也不同

为对于整个扑空率的影响要大得多。例如，在 Ocean 中，容量性扑空对于共享扑空具有支配地位；由于它们有好得多的空间局部性，和 1 MB 缓存相比整个扑空率随块大小的增加下降很快。（在小缓存中用很大的块可能有问题，存储块可能由于冲突在处理器还没有机会访问其中所有的字时就被替换出去了。）在 Raytrace 中，容量扑空和真共享扑空相比有较差的空间局部性，因此对于较小缓存用较大块的总体效果看来要差些。文献中有对于其他应用程序的伪共享和空间局部性的研究结果（Torrellas，Lam，and Hennessy 1994；Jeremiassen and Eggers 1991；Woo et al. 1995）。

对于多数应用来说，较大的缓存块会减少扑空率，但在我们考虑的大小范围内，大缓存块有两个重要的缺点。首先，它们会增加每次扑空的开销，这是由于有更多的数据要在总线上传送（尽管有些技术能缓解这种情形，例如关键字重启技术，只要所需的存储字到达就允许处理器继续向下执行）。其次，如果不是整个块都有用的话，它们增加流量，从而也增加了竞争。

4. 缓存块大小对于总线流量的影响

让我们简略地考察缓存块大小对于总线流量（而不是扑空率）的影响。尽管扑空数和所产生的总流量显然是相关的，它们对于所观察到的性能的影响可能是相当不同的。尽管现代处理器常通过用其他活动来覆盖，努力隐藏扑空所带来的延迟，扑空所带来的代价还是可能直接影响到性能。另一方面，流量影响性能是间接的，主要是由于所产生的竞争增加了其他扑空的代价。例如，一个应用程序的扑空率可能通过增加缓存块的大小得到了明显的减少，但总线的流量增加了 50%；如果该应用程序最初只用到总线能力和存储带宽的 10%，这可能就是一个合理的折中。增加总线和存储的利用率到 15% 不可能明显增加扑空延迟。然而，如果应用程序最初要用 75% 的总线和存储带宽，那么增大块的大小可能就是一个不好的主意。

图 5-24 表示我们应用程序随缓存块大小变化的总的总线流量，单位是每条指令的字节数或者每 FLOP 的字节数。从这个图中可以看到三个要点。首先，流量的行为和扑空率表现得相当不同。只有 LU 表现出随块大小单调递减的总流量。多数其他应用随块大小的变化看到的是两倍或者三倍的流量。其次，除 Radix 外，其他应用的总流量需求仍然是小的，即使块大小为 256 字节。Radix 对于带宽较大的需求（对于 128 字节的缓存块，假设一个持续 200 MIPS 的处理器，大约每个处理器 650 MBps）反映了它在大缓存块上的伪共享问题。第三，对于每一次总线事务或者扑空，地址和命令流量的固定开销是小缓存块情形总流量的一个重要组成部分。因此，尽管实际应用数据流量通常随块大小的增加而增加，由于空间局部性比较差，总流量经常在 16 ~ 32 字节，而不是 8 字节时得到极小化，这是由于改善的扑空率分摊了开销。

324
~
325

图 5-25 表示 Multiprog 的流量数据。缓存块从 64 ~ 128 字节变化所引起的流量增加是小的，跳到 256 字节时就很显著了（主要是由于内核数据的引用）。最后，图 5-26 表示 64 KB 缓存对于这三个相关应用的流量结果。对 Ocean 来说，即使 64 字节和 128 字节的缓存块也不是那么差，这是由于好的空间局部性使得起支配作用的容量扑空较小。

5. 缓解大缓存块的缺点

现在的处理器倾向于用较大的缓存块。这个趋势是由处理器性能和存储器访问时间之间的差距变大带来的。较大的缓存块，在较大量数据时能够分摊总线事务和存储访问的代价。

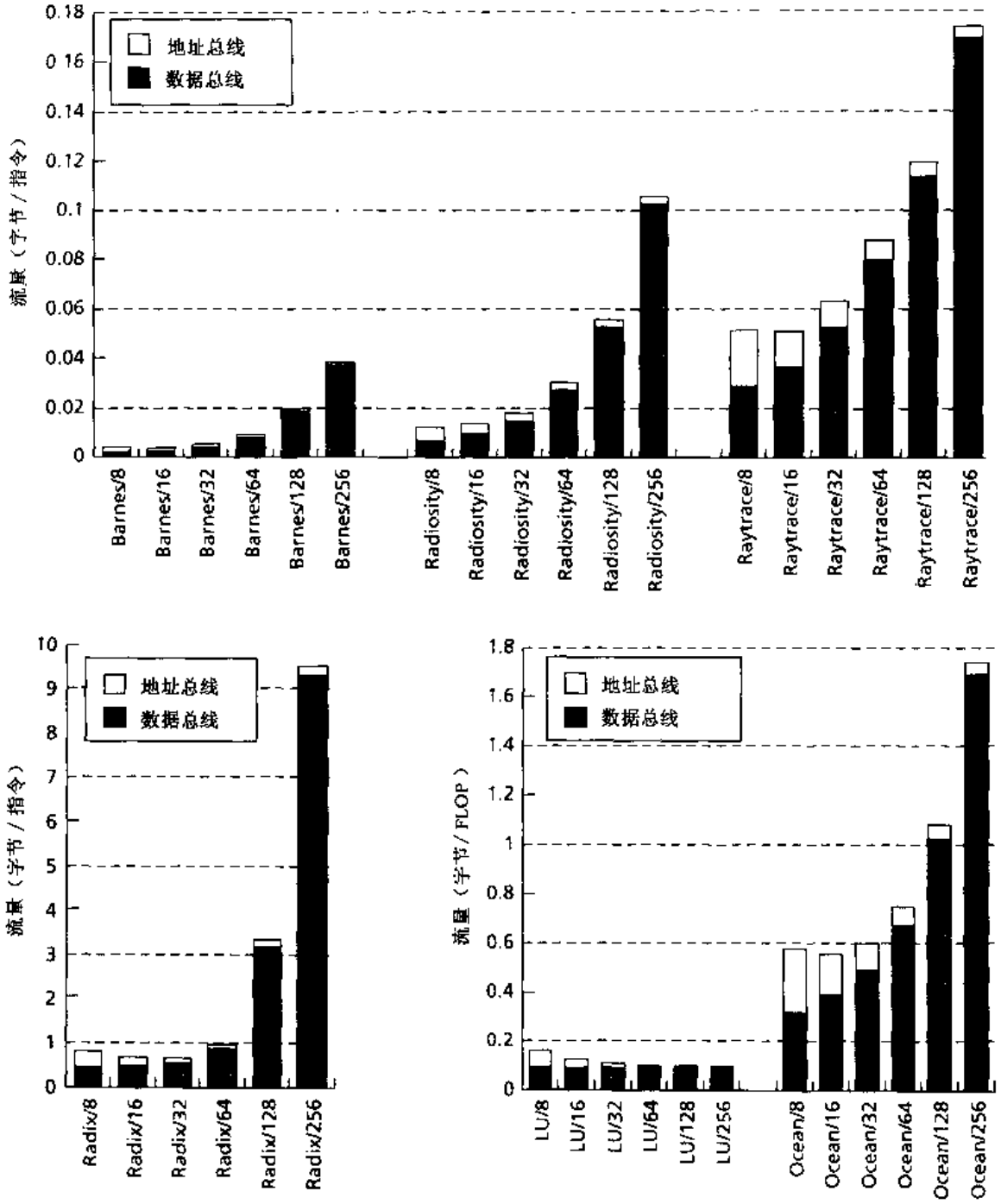


图 5-24 每处理器 1 MB 缓存情形，流量（每指令字节数或每 FLOP 字节数）随缓存块大小变化的情况。当通信扑空起支配作用时，数据流量随缓存块变化增加得相当快；LU 是个例外，它在所有类型的扑空上空间局部性都很好。地址（包括命令）总线流量倾向于随缓存块大小递减，这是由于扑空率和传送的缓存块数都减少了

处理器和存储器芯片的密度的增加使得用很大的一级和二级缓存成为可能，较大缓存块所预取的数据带来的好处可能掩盖由此带来的冲突扑空增加的问题。然而，这个趋势对多处理器设计来说可能预示着不妙，因为伪共享变成了一个较大的问题。幸运的是，我们有一些硬件和软件的机制能用来对付大缓存块的不良影响。

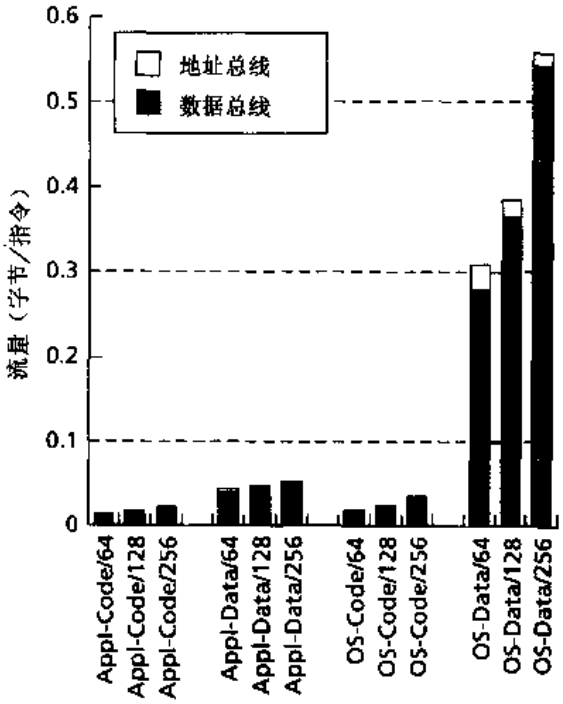


图 5-25 1 MB 缓存情形，Multiprog 应用中流量（每指令字节数）随缓存块大小变化的情况。从操作系统内核来的数据流量随缓存块变化增加得相当快

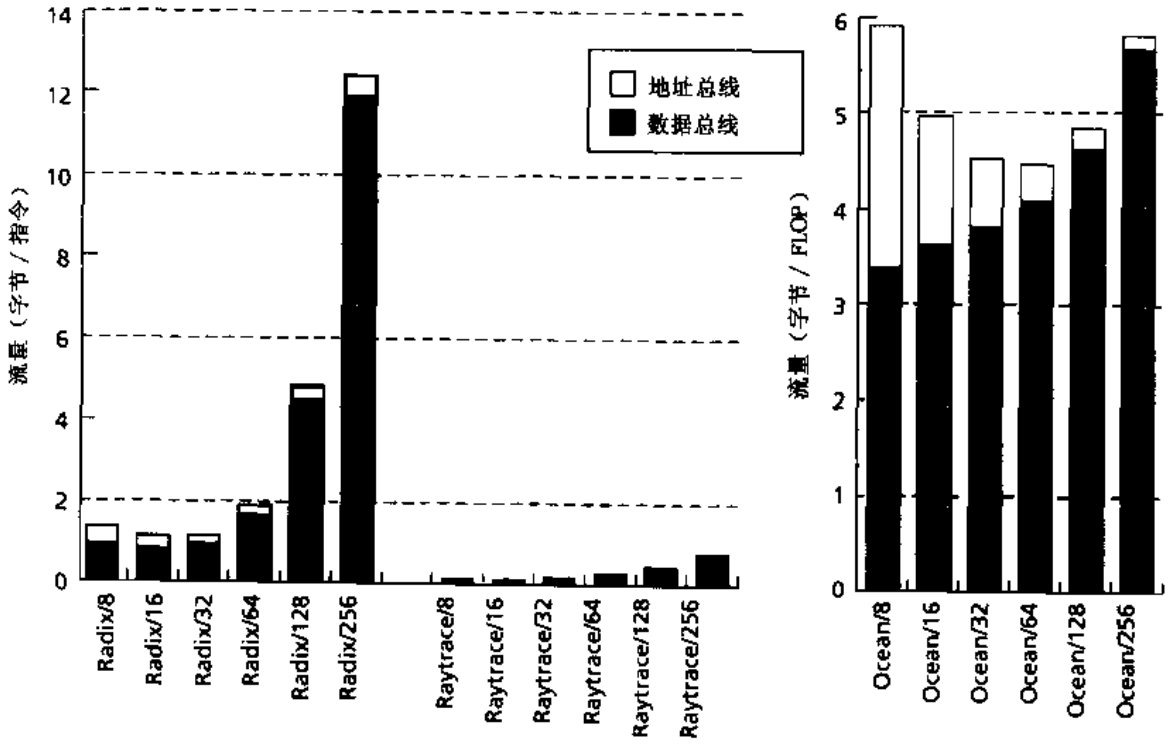


图 5-26 每个处理器 64 KB 缓存情形，流量（每指令字节数或每 FLOP 字节数）随缓存块大小变化的情况。对 Ocean 来说，流量的增加要比 1 MB 缓存情形慢，这是由于现在起支配作用的容量扑空表现了很好的空间局部性（进程在子网格上的遍历）。然而，一旦引起伪共享的缓存块阈值被超过，Radix 的流量就增加得很快

在本章的后面，我们对减少伪共享，针对一致性扑空改进局部性的软件技术有进一步的详细讨论。它们本质上涉及数据结构的组织或者任务的分配，使得由不同进程访问的数据不在共享地址空间细粒度交织存放。一个例子是用高维数组，使得存储块或划分整个是连续

326 ~ 328

的。人们也开发了若干编译技术来自动地进行数据布局，从而减少伪共享（Jeremiassen and Eggers 1991）

由于伪共享是由一致性的粒度过大引起的，减少它但仍然利用空间局部性的方法是在数据传送时用大的存储块，但在一致性管理中用较小的单位。一种自然的硬件机制是用存储子块。每个缓存块有一个地址标记，同时对每一个子块还有不同的状态位。一个子块可能是有效的，而其他的子块同时可能是无效的或者脏的。这种技术用在许多单处理器系统中以在替换时减少写回存储器的数据量，或者在读扑空时减少存储器访问时间（一旦有关子块到达就重启处理器，称为关键字重启）。为了避免伪共享，一个处理器所做的写操作可能会作废另一个处理器的子块，但保持其他的子块有效。从另一个角度来实现这种思路，也可以用较小的缓存，但扑空发生时系统可以多预取一些存储块。还有人提出过可调整块大小缓存的建议（Dubnicki and LeBlanc 1992）。这些做法的缺点是增加状态和复杂性，不太适用商用缓存的设计。

一种更精细的硬件技术是延迟从一个处理器发出的作废操作的传播或者应用，等着有多个写操作后一起来作废。延迟作废操作，将一组操作集中起来一次完成，以减少对那些块的干涉读扑空的出现。然而，这种技术可能以一种微妙的方式改变存储器同一性模型。在第9章，当我们考虑可扩展机器的弱同一性模型的时候，会有进一步的讨论。另一个减少伪共享的硬件技术是用基于更新的协议。

5.4.5 基于更新和基于作废协议的对比

对一个缓存中某个块的写操作应该引起其他缓存中的拷贝更新还是作废？这一直是争论不休的话题。不同的厂家有不同的态度并且实际上对不同的设计用了不同的做法。这种矛盾的起因是由于基于更新和基于作废协议的相对性能在很大程度上取决于应用程序负载所表现出来的共享的模式，以及各种背后操作的代价。从直觉上看，如果处理器在更新之前使用数据，并可能希望要在将来看到新的值，更新就会比作废表现出更好的性能。然而，如果持有老数据的处理器再也不会用它了，更新就是不必要的，所引起的流量只是消耗了互连和控制器资源。作废则会清除旧的拷贝，消除了明显的共享[○]。这种更新协议的“收集鼠”现象在多道程序下特别令人难受，串行进程在操作系统的控制下在处理器之间迁移，于是无用的更新在不再运行该进程的处理器的缓存中进行。如例5.12所示，我们容易构造出一种方案明显优于另一种方案的情形。

329

例5.12 考虑下述两个程序访问模式：

- 模式1：重复 k 次；处理器1向变量 V 写入一个新的值，处理器2到处理器从 P 都读 V 的值。这表现了一种单生产者多消费者机制可能出现的情形，例如，在一到多事件同步中处理器访问一个高度争用的标记变量。
- 模式2：重复 k 次；处理器1向变量 V 写 M 次，然后处理器2读 V 的值。这表示了一种在处理器对之间可能出现的一种共享模式，其中第一个处理器连续计算并累加一个变量的值，当累加完成后，另一个处理器读这个值。

用缓存扑空数和总线流量来评价，基于更新和基于作废协议的相对代价是什么？假设一

○ 这种共享是不必要的。——译者注

个作废/升级过程要消耗 6 个字节（5 个字节地址，1 个字节命令），更新需 14 个字节（6 个字节地址和命令，8 个字节为更新的数据），且一个常规的缓存扑空要 70 个字节（6 个字节地址和命令，加上 64 个字节的数据，对应一个缓存块）。还假设 $P = 16$、$M = 10$、$k = 10$，所有缓存最初都是空的。

解答：对于模式 1 用更新方案，在所有 P 个处理器上的第一次迭代将引起一个常规的缓存扑空（包括处理器 1 在写操作的时候）加上由于写的一次更新。在后续的 $k-1$ 次迭代中不会产生更多的扑空，并且每次迭代只有一次更新产生。这样总体上将看到扑空 = P = 16；流量 = $P \times$ RdMiss + $(k-1) \times$ Update = $16 \times 70 + 10 \times 14 = 1\,260$ 字节。

对于作废方案，所有 P 个处理器在第一次迭代将引起一个常规缓存扑空。在后续 $k-1$ 次迭代中，处理器 1 将产生一次升级，但所有其他的处理器将经历一次读扑空。这样，将升级算作扑空，整个我们将看到扑空 = $p + (k-1) \times p = 16 + 9 \times 16 = 160$，其中 151 个是读扑空，9 个是升级；流量 = 读扑空 × RdMiss + $(k-1) \times$ Upgrade = $151 \times 70 + 9 \times 6 = 10\,624$ 字节。

对于模式 2 用更新方案，首次迭代将发生两次常规缓存扑空，处理器 1 一次，处理器 2 一次。在后续 $k-1$ 次迭代中，将没有更多的扑空产生，但在每次迭代会产生 M 次更新。这样，总体上我们将看到扑空 = 2；流量 = $2 \times$ RdMiss + M × $(k-1) \times$ Update = $2 \times 70 + 10 \times 9 \times 14 = 1\,400$ 字节。

对于作废方案，首次迭代将发生两次常规缓存扑空。在后续 $k-1$ 次迭代中，每次迭代会产生一次升级（对于第一个写）加上一次常规读扑空。这样，将升级算作扑空，总体上我们将看到扑空 = $2 + (k-1) \times 2 = 2 + 9 = 11$；流量 = 扑空 × RdMiss + $(k-1) \times$ Upgrade = $11 \times 70 + 9 \times 6 = 824$ 字节。■

这些例子说明，有可能设计出方案来，使更新和作废协议双方的优点都体现出来。这样方案的成功将取决于它们的代价和对于真实并行程序和负载的共享模式。我们下面先简略地探讨一下设计选项，然后用负载驱动方法来进行评估。

1. 基于更新和基于作废协议的结合

一种能够体现两种类型协议优点的方式是在硬件上对两者都支持，在页面粒度动态决定对给定的一个页一致性是通过更新还是通过作废来维护。关于协议选择的决定可以通过系统调用来指出。这种方案的主要优点是它们相对比较容易支持；它们利用 TLB 来向一致性子系统的其他部分指明用哪一种协议。这种方案的主要缺点是它们给程序员增加了为页或者数据结构选择协议的负担。由于控制的粒度较粗，也使得这种决定不容易作出，因为适应不同协议的数据结构可能落在相同的页面上。 [330]

一种替代的方法是通过在运行时观察共享的行为，在缓存块的粒度来选择协议。理想情况下，对每一个写，希望能够看到未来所有处理器要对当前缓存块的引用，然后再决定是否作废其他的拷贝或者更新。由于这个信息显然是得不到的，还由于有缓存替换和伪共享带来的严重波动，需要一种更实际的方案。

所谓竞争性方案，是基于运行时观察到的模式，从硬件上在作废和更新之间改变对于一个存储块的协议。这种方案的关键属性是，如果对一个缓存块作了错误的决定，由于那个错误决定所带来的损失应该保持有界并且很小（Karlin et al 1986）。例如，如果一个块当前用更新模式，一旦一个处理器连续向它写，但没有其他处理器从中读，它就不应该保留在那个模式中。

有一类方案，旨在限定更新协议的损失，原理如下（Grahn、Stenstrom and Dubois 1995）。从 5.3.3 节介绍的基本 Dragon 更新协议开始，让每一缓存块和一个向下计数器联系起来。只要一个缓存块被本地处理器访问，那一块的计数值就被置一个阈值 k。每当一个块收到一次更新，计数器就递减。如果计数器到了 0，这个块就在本地被作废。本地作废的后果是下一次一个更新在总线上产生时，它可能找不到持有一个有效拷贝的缓存；在这种情形，这个块将切换到已修改状态（如 Dragon 协议那样）并且将停止产生更新。如果现在某个其他处理器访问这一块，它就要再次被切换到共享状态，这个混合协议会又开始产生更新。

一个在 Sun SparcCenter 2000 中实现的相关做法是以某种概率有选择地作废，而不是更新，这个概率是在配置机器时建立的（Catanzaro 1997）。还可以有一些其他混合的做法。例如，在一级缓存上用一种基于作废的协议，在二级缓存上缺省地用基于更新的协议。然而，如果对于一级缓存中的一个有效块二级缓存收到了第二次更新，那么这个块也要在二级缓存中作废。这样当一个块在所有二级缓存中都作废时，对该块的写就不再引起更新。

331

2. 利用工作负载驱动的评估

为了评价我们前面介绍的作废，更新和混合协议的折中，图 5-27 按类表示四种应用的扑空率，用 1 MB 四路组相联缓存和 64 字节的块。所用的混合协议是刚才描述的基于阈值的方案。我们看到，对于明显容量性扑空的应用，扑空有时在使用更新协议时增加。这是合乎情理的，因为这个协议（用 LRU 替换算法）将数据保持在处理器缓存中，那些数据要被作废协议去掉。对于那些具有明显真共享或者伪共享扑空率的应用程序，这些类别随着更新协议减少：在一个写更新后，持有该块的其他的缓存能够访问它们而不会扑空。从总体上看，对于这三种情形来说更新协议看起来是优越的，而混合协议的优越性居中。然而，那种没有显示在这个图中的类型是对于这些协议的升级或者更新操作。这个数据表示在图 5-28 中。注意，

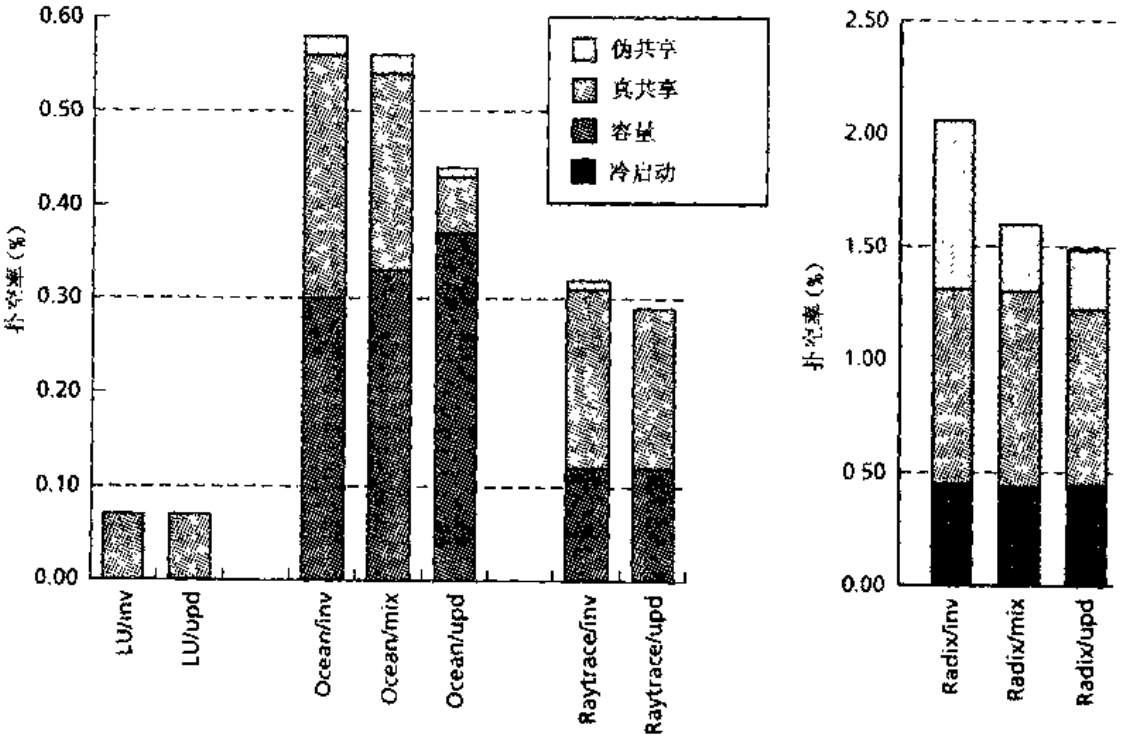


图 5-27　扑空率及其在作废、更新和混合协议上的分解。这里的数据假设 1 MB 缓存、64 字节缓存块、4 路组相联、对混合协议阈值 $k=4$

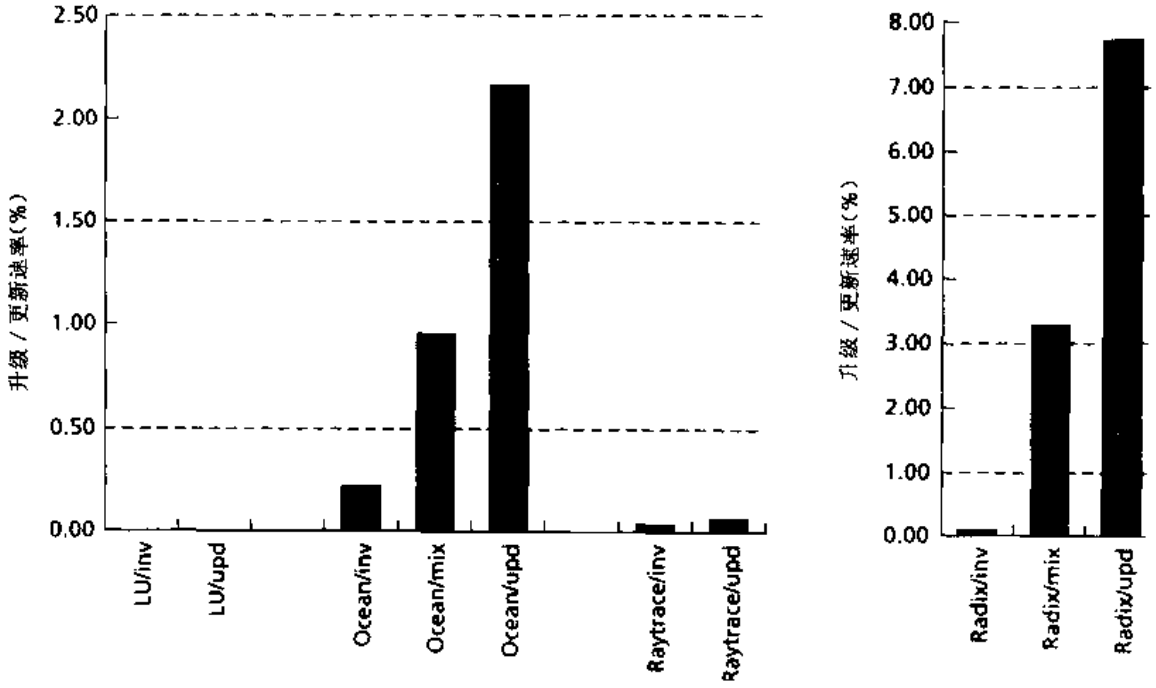


图 5-28　作废、更新协议和混合协议上的升级和更新率。这里的数据假设 1 MB 缓存，64 字节缓存块、4 路组相联、对混合协议阈值 $k=4$。数据率是相对于总的存储访问数

由于更新操作大概 4 倍于扑空，这些图的比例已经改变了。由于在机器中处理它们的方式可能不同，将这些操作和其他扑空分别开来是有用的。更新是一个单字的写操作，而不是整个缓存块的传送。因为数据是从它所产生的地方被推出来的，它可能在被需要之前到达消费者。即使对于生产者来说，更新和升级操作的时延和扑空的时延相比，可能不那么关键，这是因为它很容易被安排在处理器的关键路径之外（见第 11 章）。

不幸的是，和更新相联的流量是相当大的。这主要是由于同一个处理器对相同的块在一个读之前发出的多个写操作都要产生更新。而对作废协议来说，第一个写可能引起作废，但其他的可以简单地累积在本地块中，在回写或者向别的缓存提供数据时以一次总线事务传送完成（见例 5.12）。这种增加的流量引起冲突，能够大大增加扑空的代价。复杂的更新方案可能试图延迟更新的时机，以取得一种类似的效果（通过合并写缓冲中的写操作）或者用其他的技术来减少流量和改进性能（Dahlgren 1995）。然而，所增加的带宽要求支持更新所带来的复杂性、缓存块变大的趋势，以及多道顺序程序负载的收集鼠现象，使得基于更新的协议在业界用得越来越少。我们在第 8 章会看到，更新协议对于可扩展缓存一致性系统结构还有一些其他的问题，这进一步使得它对多处理器系统设计者的吸引力越来越低。

讨论了如何保持数据的一致性，我们现在来考虑在基于总线的多处理器中同步是如何管理的。

332 ~ 333

5.5　同步

在多处理器中，硬件和软件一种关键的相互作用表现在对同步操作的支持上。同步操作包含互斥、点对点事件和全局事件。在过去的若干年里，关于要多少硬件支持以及到底应该提供什么样的硬件原语来支持这些同步操作，有相当多的争论。而且结论不仅随时间变化，

还随技术工艺和设计风格的变化而变化。硬件支持有速度的优越性，但将功能由软件实现在代价、灵活性和对不同情况的适应性方面有优越性。Dijkstra（1965）和 Knuth（1966）的经典工作表明，只通过原子性的读和写操作就能够提供互斥（假设一个顺序同一的存储器）。然而，所有实际应用中的同步方法都依赖于某种原子性读－改－写操作的硬件支持，其中一个存储单元的值的读、修改和写回被原子性地完成，中间没有其他处理器对该单元的访问。从简单到复杂的各种同步算法都可以通过软件用这样的原语来实现。

指令集的历史也给我们提供了硬件对同步支持的演化过程。在 IBM370 指令集中的一个关键的增强就是包含了一个复杂的原子指令（比较并交换指令）来在单处理器或者多处理器系统支持并发程序设计中的同步。这种比较并交换指令将一个存储单元的值和一个寄存器的值对比，如果相等，则将该存储单元的值和另一个寄存器的值交换。Intel x86 允许给任何指令设置一个前缀，相当于加锁，使它成为原子性的；由于源和目的操作数是存储单元，许多这样的指令能用来实现各种原子操作，甚至涉及到不只一个存储单元。高级语言系统结构的支持者们还提出了用户层次的同步操作（例如锁和栅障）也应该直接由硬件来支持，而不仅是原子性的读－改－写原语；这就是说，同步“算法”本身应该由硬件实现。对于这个问题，在精简指令集的讨论中是很活跃的，主要由于 RISC 中存储访问的操作只涉及一个存储操作数的简单的装入和存放。Sparc 的方法是提供涉及寄存器和一个存储单元的原子操作，用一个简单的交换（将寄存器内容和存储单元内容原子性交换）和比较并交换。MIPS 在早期的指令集中没有安排原子性原语，IBM 的 Power 最初在 RS6000 也是如此。最终包含到 MIPS 中的原语是一个新颖的组合，涉及一个特别的装入和一个条件存放，将在本节后面讨论，它使得各种高层次的读-改-写操作都可以形成，不需要硬件一个个实现它们。从本质上讲，这一对指令，而不是单个指令能用来实现原子性的交换或者更复杂些的原子操作。这种做法后来也被 PowerPC 和 DEC Alpha 系统结构采用，现在是相当流行的。如同我们将要看到的那样，同步引出了一系列跨通信体系结构各个层次的折中。不仅许多高层的操作和低层的原语能由硬件来实现，应用所提出的同步要求也是变化很大的。

334

本节的要点是讨论同步操作如何能通过软件算法和硬件原语的组合，在一种基于总线的缓存一致的多处理器上实现。特别地，讨论互斥的实现，通过加锁-解锁对，通过标记实现点对点事件同步，通过栅障实现全局事件同步。让我们从考虑同步事件的各个成分开始。这将清楚地表明在硬件直接支持高层互斥和事件操作是困难的，而且可能使得实现很不灵活。然后，给定硬件只对最基本的原子操作提供支持，我们能够考察用户软件和系统软件在同步操作中的作用，最后较详细地考虑硬件和软件的设计权衡。

5.5.1 同步事件的组成部分

一个同步事件主要由三个成分组成：

1）*获取方法*：指的是进程用来获得同步权的方式（这里“同步”包括进入临界区或者向前推进，以通过事件同步机构）。

2）*等待算法*：这是进程用来等待同步可用的方法；例如，如果一个进程试图获取一把锁但该锁被别的进程占有着，或者想推进通过一个事件但事件还没有发生。

3）*释放方法*：这是进程用来使其他进程推进通过一个同步事件的方法；例如，Unlock 操作的一种实现，最后到达栅障的进程释放等待进程的方法，或者是通知一个等待在点对点

事件上的进程事件已经到达的方法。

等待算法的选择基本上是独立于同步类型的。有两种主要的选择：忙等待和阻塞。忙等待意味着进程在一个循环中不断测试某个变量是否改变了值。由另一个进程做的同步事件的释放改变这个变量的值，让等待进程可以向前推进。在阻塞情形，进程不执行循环，而是简单地阻塞（挂起）它自己，如果需要等待的话，还要释放处理器。当它所等待的释放信号出现时，它将被唤醒并且进入就绪状态。忙等待和阻塞相比的利弊是显而易见的。阻塞的开销较大，这是由于挂起和重启一个进程涉及到操作系统（挂起和重启一个线程涉及一个线程包的运行支持），但这种做法使得处理器能为其他进程或线程所用。忙等待避免了挂起的开销，但在等待时消耗了处理器和缓存带宽资源。阻塞机制的功能要比忙等待强，这是因为如果不允许正在等待的进程或者线程运行，忙等待就永远不会结束[○]。在等待周期比较短的场合，忙等待可能会好一些；而如果等待时间比较长或者还有其他进程要运行，阻塞可能是一种更好的选择。也可以用一些混合的方法，例如让进程忙等待一会儿，如果超出了某个时间阈值，进程就阻塞，让其他进程投入运行（一种两阶段等待算法）。 335

用硬件实现高层同步操作的困难不在于获取或释放部分，而在于等待算法。这样，对于获取和释放方法的关键之处在于提供硬件支持，而用软件将三者结合起来是有道理的。不过，如何从硬件/软件相互作用方面实现忙等待中的循环操作仍然留有微妙但十分重要的问题。

5.5.2 用户和系统的角色

谁应该负责实现诸如锁和栅障之类高层同步操作的具体内容？典型地，程序员要用锁、事件或者更高层的操作，而不需要关心它们的内部实现。实现留给系统，系统来决定提供多少硬件支持，哪些功能用软件来实现。人们已经开发出了利用简单原子交换原语的软件同步算法，其性能可以和全硬件实现相媲美，而且它们所带来的灵活性和对硬件的简化是很有吸引力的。和系统设计的其他方面一样，由硬件提供更快操作的实用性取决于这些操作在应用中出现的频度。这样，我们又一次体会到最好的答案的作出在于对应用行为的最好的理解。

同步构造的软件实现通常包含在系统库中。好的同步库设计是相当具有挑战性的。一种潜在的复杂因素在于同一种同步（锁，栅障），甚至相同的同步变量，可能在不同的时间用于非常不同的运行条件。例如，对一把锁的访问可能是低冲突的（较少的处理器，其中可能只有一个在同一时间试图获取这把锁），也可能是高冲突的（许多处理器同时试图获取这把锁）。不同的情形提出了不同的性能要求。在高争用情况下，大多数进程将等待一段时间，对锁算法一个关键的需求是要提供高的加锁－开锁传送带宽；在低争用的情况，关键目标是对锁的获取提供低延迟。不同的算法可能更好地满足不同的需求，于是我们必须或者找到一个好的折中算法，或者对不同的同步提供不同的算法供用户选择。如果我们运气好，一种灵活的库可能在运行时根据具体情况，选择最好的实现。不同的同步算法也可能依赖于不同的基本硬件原语，这样，某些算法可能特别适合一个特定的机器，而对其他机器并不适合。在多道程序情况下，进程调度和其他资源交互可能改变并行程序中进程的同步行为。和在专用条 336

○ 这种进程或线程得不到资源的问题在多处理器情形实际上要简单些。当进程在一个处理器上分时的情况下，没有强占的忙等待肯定是个问题。如果每个进程或线程有它自己的处理器，就肯定没有问题。在多处理器上的多道程序环境可以看作是在上述两种情况之间。

件下表现出低时延高带宽的简单算法相比，考虑到多道程序效果的复杂算法可能在实践中给出更好的性能。所有这些因素使得同步成为硬件/软件相互作用的一个焦点。

5.5.3 互斥

有许多算法可用于实现互斥（加锁-开锁）操作。简单的算法通常在对锁的争用很小的情况下是快的，但在高争用情况下则效率不高，而复杂的算法对于争用处理得较好，但在低争用时的代价较高。在关于硬件锁的一个简略讨论后，本节描述基于存储器的一个最简单的软件锁算法，该算法用了一种原子交换指令。之后，我们讨论这些简单的算法如何可以不用原子性交换指令，而用专门的装载-加锁和条件存储指令对来实现，以及有关的利弊。然后，我们会看看更复杂一些的算法，它们都可以用上述任何实现原子操作的方法来构成。

1. 硬件锁

尽管目前锁操作在基于总线的机器上不那么流行，但锁操作能够完全用硬件支持。在某些较老的机器上，一种做法是在总线上用一组锁线路，其中每一条线代表一把锁。持有锁的处理器负责设置该线的相应状态，等待该锁的处理器要等待这条线的释放。当有多个请求者时，一种优先级线路决定下一次该哪个处理器得到这把锁。然而，这种做法相当不灵活，因为在同一时间只有一个有限量的锁可用，并且等待算法是固定的（典型的是某种形式的忙等待，超时放弃控制）。通常，只是操作系统为一些特定的目的用这些硬件锁，这些目的之一就是在存储器中实现大量的软件锁。CRAY Xmp 用的就是这种做法的一种有趣的变形　一组寄存器在处理器之间共享，包括若干固定数量的锁寄存器。尽管体系结构使得将寄存器分给用户进程成为可能，但这样的寄存器太少，在一种通用计算环境下这样做很不方便，因此在实践中这些锁寄存器主要用来在存储器中实现高层的锁。

337

2. 简单的软件锁算法

考虑一种为一个临界区代码提供原子性的锁操作。对于其获取方法，一个试图获得一把锁的进程必须检测锁是否是自由的；如果是的话，还要声明对于该锁的占有权。锁的状态可以存在于一个二进制变量中，0 表示自由，1 表示忙。考虑锁的获取操作的一种简单方法是，试图获得该锁的进程应该检测这个变量是否为 0，若是则将它置 1，这样就将该锁标为忙；如果是 1（忙），则进程应该按照一定的等待算法等待该变量变成 0。解锁（开锁）操作应该简单地将该变量置 0（释放方法）。下面是试图实现上述想法的汇编层次的指令。（在我们的伪汇编记号中，如果有第一个操作数，则它总表示操作的结果。）

```
lock:  ld    register, location  /*copy location to register*/
       cmp   register, #0        /*compare with 0*/
       bnz   lock                /*if not 0, try again*/
       st    location, #1        /*store 1 into location to mark it locked*/
       ret                       /*return control to caller of lock*/
```

and

```
unlock: st location, #0          /*write 0 to location*/
        ret                      /*return control to caller*/
```

这个锁过程要为它后面的临界区提供原子性，但它自己却保证不了自己的原子性。为认识到这一点，假定锁变量最初为 0，两个进程 P_0 和 P_1 执行上面的锁操作。进程 P_0 读到了 0，

认为锁是自由的，于是它通过了转移指令。它的下一步是置该变量为1，将它标记为忙；但在它做这件事之前，进程P_i读到了变量为0，认为锁是自由的，也通过了转移指令。于是我们现在有了两个进程，同时通过了锁，进入到同一个临界区，但这正是我们希望用锁来避免的。让存储指令紧跟着装载指令也无济于事。由两个指令构成的序列——读（测试）锁变量的值以检测它的状态，若它是自由的则向它写（置）1——不是原子性的，并且没有任何措施来防止这些操作在不同的进程中交织执行。我们需要的是一种方法，它可以原子性地测试一个变量的值，并且如果测试成功就将它置成另一个值（即，原子性地读并且有条件地修改一个存储单元），并且然后无论这个原子序列执行成功与否都要返回。向用户进程提供这种原子性的一种方式是将锁代码放在操作系统中，通过一个系统调用来访问它，但这种开销很大并且操作系统自己怎么支持锁也是个问题。另一种可能性是在锁代码序列前后用一把硬件锁，但这要求有硬件锁，并且在现代处理器上显得慢。 338

对于这种锁问题，一种高效的通用解决方案是在处理器的指令集合中支持一种原子性的读-修改-写指令。典型的做法有一种原子交换指令：由指令给出的一个存储单元的值被读到一个寄存器中，并且另一个值被写到该单元中。操作是原子性的，其间不允许有其他对该单元的访问。这种操作有许多变形，通过所存储的值的特点提供不同的灵活性。一种适用于互斥的简单例子是测试并设置指令。在这种情况下，存储单元的值被读到一个特殊的寄存器，同时常数1被写入该单元，操作原子性完成。这种测试并设置指令的成功通过考察寄存器中的值来确定。如果是0，则指令成功。如果是1，就是不成功的；由这测试并设置指令写入存储器的1和其中原有的值是相同的，因此也没什么害处（1和0是典型值，其他常数也是可用的）。给定这样一条指令，记为t&s，可以写出如下加锁和解锁代码：

```
lock:   t&s register, location
                          /*copy location to reg, and set location to 1*/
        bnz register, lock /*compare old value returned with 0*/
                           /*if not 0, i.e., lock already busy, so try again*/
        ret               /*return control to caller of lock*/
```

和

```
unlock:st location, #0    /*write 0 to location*/
        ret               /*return control to caller*/
```

在这种锁的实现中，程序不断试着用测试并设置指令来获取该锁，直到寄存器中出现由测试并设置留下的0值，表示在测试的时候锁是自由的（当时测试并设置指令也将存储单元置1了，从而获得了它）。解锁操作简单地将和锁相连的存储单元置0，指出这把锁现在自由了，从而使得任何进程在其后的锁操作能成功。一种简单的互斥结构于是用软件实现了，依赖的硬件支持是一条原子性的测试并设置指令。

这种原子性指令还有一些更复杂的变形，如我们将会看到的，它们在不同的软件同步算法中可能用到。一个例子是交换指令。像测试并设置指令一样，这种指令将特定存储单元的值读入特定的寄存器，但将该寄存器开始的任意值写到该存储单元中。也就是说，指令将存储单元和寄存器的值做了交换。很清楚，用交换指令可以像前面一样来实现一把锁，只要我们用0和1并且保证寄存器的值在交换指令执行前为1；如果由交换指令留在寄存器中的值为0，则加锁成功。 339

另一个例子是一类取并操作（fetch&op）指令。这样的指令也是指定一个存储单元和一

个寄存器。它原子性地将存储单元的值读入寄存器并且将一个值（由这个取并操作指令指定的一个操作，作用在读出来的值所产生的）写入该单元。取并操作指令的最简单形式是取并加1（fetch & increment）和取并减1（fetch & decrement），将当前值增减1。取并加（fetch & add）指令则用另外一个操作数，可能来源于一个寄存器或者立即数，加到该单元先前的值上去。一种更复杂的原语是比较并交换（compare & swap）指令。它取两个寄存器和一个存储单元（即，它是一个3操作数指令，RISC体系结构通常不支持）；它将存储单元的值和第一个寄存器比较，如果相等，就将存储单元内容和第二个寄存器内容交换。

3、简单锁的性能

图5-29示出SGI Challenge[○]中的一种简单的测试并设置锁的性能。性能是通过在一个循环中重复执行下述微测试程序来得到的：

```
lock(L);
critical-section(c);
unlock(L);
```

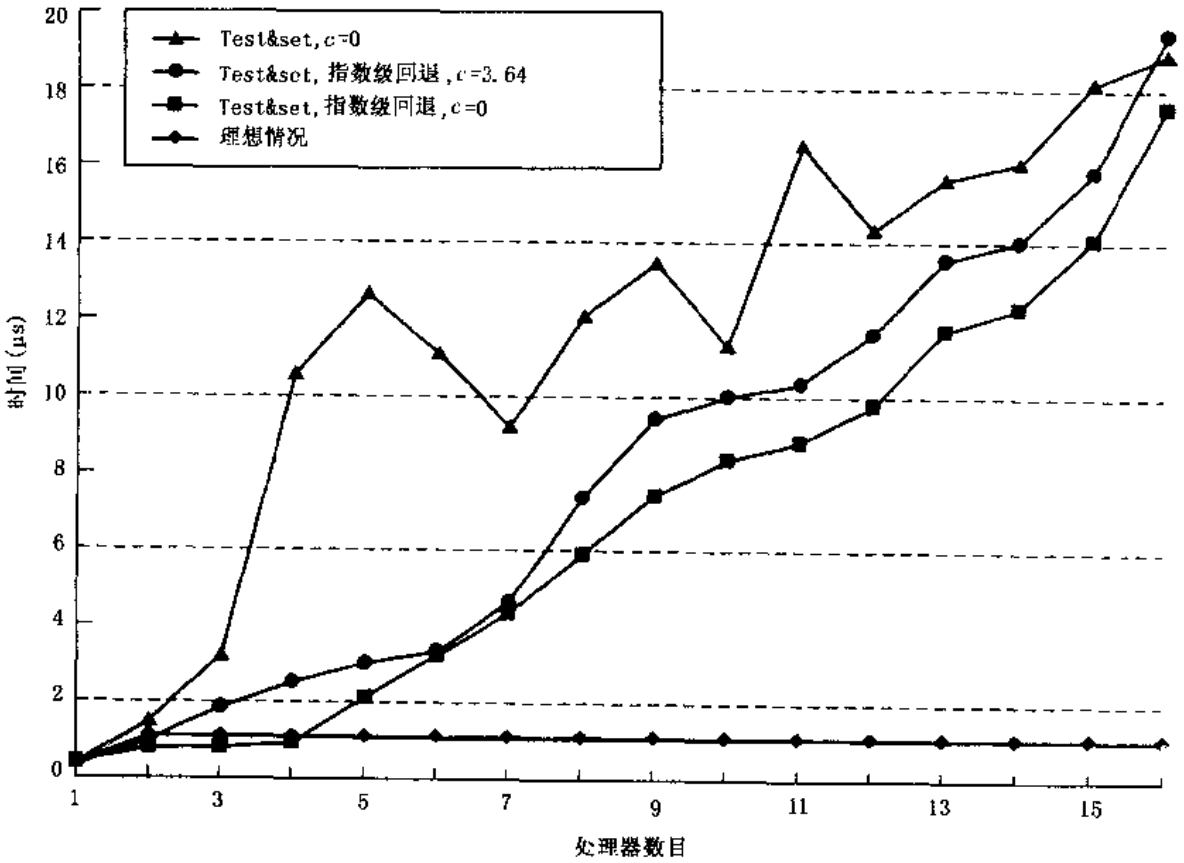


图5-29　在SGI Challenge上，随处理器个数的增加，一种综合测试并设置（test & set）锁的性能。y轴是每加锁－解锁操作对所花的时间，不包括临界区中的c微秒延迟。最上面一条曲线的非规则性是由于处理器的争用对时间的依赖性所引起的

○ 事实上，用于本章来说明同步性能的SGI Challenge上的处理器并不提供测试并设置指令。它所用的是另外一种原语，本节后面将会谈到。对于这些实验来说，我们用其他一些原语综合出一种在行为上类似于测试并设置指令的实现。关于一些较早的机器，例如Sequent Symmetry，人们得到过用基于真实测试并设置指令构成的锁的结果（Granuke and Thakkar 1990；Mellor-Crummey and Scott 1991）。

其中 c 是一个延迟参数，决定临界区的大小（这里它只是一个延迟，不做什么实际的事情）。这个测试程序的配置使得无论处理器的个数多少，锁调用的总数不变，这对应于一个固定的任务数在一个集中式的任务队列中要被处理，独立于参与工作的处理器的个数。性能评价的单位是每次锁的传递所花的时间，即所有处理器执行这个测试程序所花的总时间除以锁被获得的次数。在临界区中所花的总时间（即 c 乘以锁被获得的次数）要从总执行时间中减掉，从而得到锁传递所花的时间（也就是包括了锁操作引起的竞争）。所有测量单位为 μs。

图 5-29 中上面的曲线是用测试并设置锁、临界区很小的情况（暂时忽略带有回退的曲线）。理想情况下，我们希望看到每次获取锁的时间独立于参与竞争的处理器的个数，每次锁转移只有一次总线活动，如图中标有“理想情况”的曲线所示。然而，图中的情形显然表示了性能随处理器个数的增加而降低。

这里的问题在于等待方法期间所产生的流量：每次试图测试锁是否自由，无论成功与否，都产生一个对于含有该锁变量的缓存块的写操作（由于它用一个测试并设置指令，向该单元写 1）；由于这个块当前在某个别的处理器的缓存中（当它做测试并设置时向该块写过后），于是每一个写产生一个总线事务来作废该块先前的拥有者。这样，所有处理器不断发出总线事务，甚至在等待算法期间也消耗珍贵的总线带宽。随着处理器数的增加，从而也是测试并设置指令和总线过程的频度增加，所导致的竞争大大减慢了锁的转移。在实际中，它将影响在临界区中所做的工作。在总线上的大量竞争以及所导致的获得锁的时间依赖性引起测试时间随处理器数，甚至同样的处理器数但不同的执行实例，大幅度的变化。图 5-29 中所表现的是一组特殊的、有代表性的执行，针对不同的处理器数的结果。

340 ~ 341

4. 简单锁算法的增强

我们可以通过两个简单的措施来减缓这种流量。首先，我们可以减少进程在等待时发出测试并设置指令的频率；其次，我们可以让进程只是在读操作时忙等待，这样在锁被释放前它们就不产生作废和扑空。这两种做法称为带有回退（backoff）的测试并设置锁和测试-测试并设置（test- and -test & set）锁。

带有回退的测试并设置锁 回退的基本思想是让进程在一次试图获取锁的操作失败后插入一个延迟。在测试并设置指令之间的延迟不可以太长；否则当锁自由了，处理器还可能处于空闲状态。但这个延迟又应该足够长，以使得流量真正降低下来。一个自然的问题是这个延迟量应该是固定还是变化。实验结果显示，好的结果是让延迟按“指数规律”变化；即第一次尝试后的延迟是一个小常数 k，它随次数按几何级数增加，这样在第 i 次尝试后，它就是 $k \times c^i$，其中 c 是另一个常数。这样的锁称为带有指数回退的测试并设置锁。图 5-29 也示出了两种这样的情况，不同之处在于临界区的大小，所选的 k 对于性能测试来说是最好的。性能得到了改进，但可扩展性仍然不怎么好，这是由于在释放和获取时还是有大量的流量相干。在文献（Granuke and Thakkar 1990；Mellor-Crummey and Scott 1991）中有关于一些较早机器用真实测试并设置指令带有回退的性能结果。习题 5.14 还讨论了为什么用回退方法时非空临界区的性能要比空临界区的性能差。

测试-测试并设置锁 算法的一个更精细的变化是让它在忙等待时用一些不产生这么多总线流量的指令。进程忙等待时重复执行一个标准装入指令，而不是测试并设置，将锁变量读入，直到它从 1（加锁）变到 0（解锁）。在缓存一致性机器中，所有处理器都可以在缓存

中进行这样的读操作，而不产生总线流量，这是由于每个进程在第一次读的时候得到了该锁变量的一个缓存拷贝。当这把锁释放时，所有等待的进程缓存中的拷贝都被作废，下一次读就要产生一次读扑空。等待进程之后会发现锁已经可用了，然后就执行测试并设置指令来实际试图获取该锁。其中之一将成功，其他的将失败，并返回到基于读的等待方法。这样的测试-测试并设置锁能大大减少总线流量。

342

5. 锁的性能目标

在考察更复杂的锁算法和原语之前，有必要清楚地指出对于锁的性能目标，并且回顾一下我们关心锁的哪些表现。目标包括：

- 低时延。如果一把锁是自由的并且没有其他处理器同时试图获取它，一个处理器应该能够以低时延获得它。
- 低流量。如果许多或者所有处理器同时试图获取一把锁，它们应该能够一个接一个得到该锁，尽量不产生流量或总线事务。如前面所讨论过的，除了不相关的过程争用总线外（包括在临界区中的），由于高流量所导致的竞争能减慢锁的获取。
- 可扩展性。时延和流量都不应该随处理器数的变化而变化太大。然而，由于在基于总线的 SMP 中处理器的数目不会太大，重要的不是渐近可扩展性，而是在一定实际范围的可扩展性。
- 低存储代价。一把锁所需的信息应该很少并且不应该随处理器数增加很多。
- 公平性。理想上，处理器应该以它们发出请求的次序获得一把锁。至少，应该避免饥饿或者明显的不公平。由于饥饿的可能性通常不大，公平性的重要性要和它对性能的影响权衡考虑。

考虑简单的原子交换或者测试并设置锁。如果相同的处理器在没有竞争的情况下重复获取一把锁，它的时延是很小的，这是由于所执行的指令数很小并且锁变量将在处理器的缓存中。然而，我们已经看到在许多处理器竞争的情况下锁会产生大量的总线流量。这种锁的性能随着处理器数的增加扩充性很差。存储代价低（只是一个变量）并且不随处理器数变化。这种锁没有任何措施来保证公平性，运气不好的处理器可能挨饿。带有回退的测试并设置锁在非竞争情况下有同样的时延，产生较少的流量，可扩展性稍好一些，不取更多的存储并且公平性也没改善。和简单测试并设置锁相比，测试-测试并设置锁无冲突的时延稍大一些（即使在没有竞争的情况下，它也要多做一个读操作），但产生的总线流量要小得多，而且可扩展性也好些。它要求的存储也可以忽略，也不保证公平性。（习题 5.12 让你计算测试-测试并设置类型的锁在不同场合下总线事务的次数，还有所花的时间。）

343

在这种测试-测试并设置锁中，由于测试并设置操作（也就是对应一个总线事务）只是在处理器注意到锁可用，并且它在忙等待一个缓存块失败的情况下才进行，所以没有回退的需要。然而这种锁也有问题，即当锁被释放时，所有等待的进程几乎会同时冲出来，进行它们的读扑空，执行它们的测试并设置指令。对于这些读扑空的总线事务可以组合在一种聪明的总线协议中；然而，每个测试并设置指令本身产生作废和后续扑空，对于 p 个处理器每个要求一次锁，会导致 $O(p^2)$ 总线流量。在发出测试并设置之前插入一个随机的延迟至少能够有助于拉开这些测试并设置指令，但它将增加在无竞争情况下获取锁的时延。测试-测试并设置方式曾经是当时一个比较大的进步，但后来人们又设计了更好的硬件原语和更好的算法来减缓它的流量问题。

6. 改进的硬件原语：Load-Locked，Store-Conditional

除了在忙等待中用读代替读－改－写外，我们还想尝试让读－改－写失败的进程不产生作废。而且，希望有这样一个原语，能够实现多种原子性的读－改－写操作，诸如测试并设置，取并操作，比较并交换等，而不是要用单独的指令分别实现。在现代微处理器中，有一种用得越来越多的方式，能达到这两个目标。它的基本精神是用一对特殊指令，而不是单个读－改－写指令来实现对一个变量的原子性访问（让我们称这个变量为同步变量）。第一条指令，通常称为 Load-Locked（装入－加锁）或者 Load-Linked（装入－链接）指令（LL），将同步变量装入一个寄存器。它后面可以跟着任意的指令，这些指令完成对该寄存器中的值操作，即对应读－改－写的修改部分。这个序列的最后一条指令是第二个特殊指令，称为条件存放（store-conditional）。它试图将寄存器的值写回去（同步变量），条件是当且仅当没有其他处理器在本处理器完成了 LL 后，对该单元（或者缓存块）进行写操作。这样，如果条件存放成功，它意味着装入－加锁、条件存放（LL-SC）指令已经原子性地对该变量进行了读、可能的修改以及回写操作。如果条件存放检测到其间对该变量或者缓存块发生了一次写，它就不会试图将值回写（或者产生任何作废）。这意味着对于该变量的原子操作失败，必须从 LL 再开始尝试。条件存放的成功或者失败由条件码或者返回值来确定。LL 和条件存放是如何实现的将留在后面讨论；现在，我们关心它们的语义和性能。

利用 LL-SC 来实现原子操作，前面那种简单的加锁和解锁算法能如下实现。其中 reg 1 是存储单元当前值装入的寄存器，reg 2 持有要被存回的值（如同测试并设置，对于一次加锁操作，reg 2 可以就是 1）。

344

```
lock:   ll   reg1, location     /*load-locked the location to reg1*/
        bnz  reg1, lock         /*if location was locked (nonzero),
                                  try again*/
        sc   location, reg2     /*store reg2 conditionally into location*/
        beqz lock               /*if store-conditional failed, start again*/
        ret                     /*return control to caller of lock*/
```

和

```
unlock: st location, #0         /*write 0 to location*/
        ret                     /*return control to caller*/
```

多个处理器可能同时执行 LL，但只有第一个将它的条件存放指令放到总线上的才可获得成功。这个处理器将成功地获得这把锁，其他的将失败，不得不重新从 LL-SC 开始。注意，条件存放失败的途径有两个，一是它在试图访问总线之前就检测到其间发生了一次写；二是在它试图访问总线，但发现某个其他处理器的条件存储已经先到达那儿了。当然，如果当进程执行 LL 时该单元是 1（非 0），它将把 1 装入 reg 1，并且将从 LL 重新开始，根本不去尝试条件存放。

值得注意的是，LL 本身不是一种加锁操作，条件存放本身也不是解锁。首先，LL 的完成本身不意味着获得了排他的访问权；事实上，LL 和条件存放一起用来实现一种加锁的操作。其次，即使 LL-SC 对成功了，也不保证在它们之间的指令（如果有的话）的执行是原子性的，因此事实上这些指令不构成一个临界区。一个成功的 LL-SC 指令对惟一能保证的是在 LL 和条件存放之间没有发生对于同步变量的写。事实上，由于在 LL 和条件存放之间的指令是无条件执行的，但如果条件存放失败应该是不可见的，所以重要的是它们不修改任何其他

重要的状态。典型地，这些指令只是操作同步变量装入的寄存器（例如，完成取并操作中的操作部分）不修改程序中任何其他的变量（修改这个寄存器是没有问题的，因为这个寄存器总是要由下一次 LL 重新装入）。显式支持 LL-SC 的微处理器厂家鼓励软件编写人员遵循这种要求，并且还经常说明哪些指令在其间可用，能够保证它们所实现的 LL-SC 操作的正确性。在 LL 和条件存放之间的指令数也应该很小，以减少由于别的处理器插入其间的写导致的条件存放失败的可能性。尽管 LL 和条件存放不构成一个加锁－解锁对，它们可以直接用来实现在共享数据结构上的某些原子操作。例如，如果所希望的功能是在一个全局变量上的一个小操作（例如一个计数器或者全局和），和在变量更新周围做加锁和解锁相比，用一种自然的指令序（LL、寄存器操作、条件存放、测试）实现起来会显得更有道理。

345

如同 test-and-test&set 指令，当 LL 指出当前锁被别人持有时，用 LL-SC 建立起来的锁在等待算法期间不会产生总线流量。比 test-and-test&set 优越之处在于它在一次获得锁的尝试失败后也不产生作废（即一次失败的条件存放）。然而，当锁被释放时，在一个装载－加锁操作循环上的踏步等待的处理器很可能会在该单元上扑空，并且向总线产生读事务。在这之后，对给定锁的一次获取只会由条件存放成功的处理器产生一个作废，但这会再次作废所有的缓存。即使和 test-and-test&set 相比，流量大大地减少，并且没有读－改－写总线事务，但流量依然随处理器数线性增加，即每次获取锁意味着 $O(p)$ 次总线事务。由于在一个上了锁的单元上踏步等待总是通过读来进行的（装载－加锁操作），不可能有什么类似的情形可以进一步改进 test-and-test&set 指令的性能。然而，在 LL 和条件装入之间可以用回退以减少突发性流量。

简单的 LL-SC 锁在时延和存储要求方面也是低的，但它也不是一种公平的锁并且所引起的流量也不是最小的。更高级的锁算法能够既提供公平性，也减少流量。它们可以用原子性的读－改－写指令来实现，也可以用由 LL-SC 综合得到的等价语义的原子操作实现。当然，这两种做法的优点是不同的。让我们考虑这样两个算法，它们适合于基总线的机器。

7. 高级的锁算法

特别地，当用一种类似于测试并设置的原子交换指令（不是 LL-SC）来实现锁的时候，我们希望当锁被释放时只有一个进程实际上试图去获取该锁（而不是像所有前面的算法那样，让所有进程都冲出来做测试并设置，并且发出作废信号）。更希望的是当一把锁释放时只有一个进程发生读扑空（对 LL-SC 也希望如此）。加号锁达到了第一个目的；基于数组的锁两个目标都达到了，代价是需要稍多一点空间。和所有前面的锁不同，这样两种锁是公平的并且以 FIFO 序让处理器得到锁。

加号锁 加号锁的操作就像在餐馆排队买三明治，或者像银行的出纳队列的票号系统。需要获得锁的进程取一个票号，然后在一个全局 now-serving（现在服务）号上忙等待（就像我们在买三明治队列上刻意观察的 LED 显示屏上的数字）直到 now-serving 号等于自己所得的票号。要释放一把锁，进程只要将 now-serving 号加 1，于是下一个等待的进程能够获得该锁。所需的原子性原语是，当进程首次到达锁操作，要从一个共享计数器上获得它的票号时用到这个原语。由于只有票号等于 now-serving 的过程在看到锁被释放后试图进入临界区因此在实际获得锁时不需要做原子操作。这样，获取方法是 fetch&increment，等待算法是用忙等待的方式检查 new-serving 是否等于自己的票号，释放方法是增加 now-serving 。这种锁的不竞争时延大约和 test-and-test& set 锁相等，但所产生的流量要小得多。尽管每一个进程在到达该锁时都做取并加 1 操作（假想所有进程不是同时到达），在锁的一次释放后测试并设置就不必要

346

了，而那通常更趋向于同时性，并且竞争性要严重得多。由于进程是以它们执行 fetch&increment 操作的次序得到该锁的，加号锁对存储的需求是固定的和小量的，并且它还是公平的。

加号锁所需的取并加 1 操作可以用 LL-SC 实现。然而，由于简单的 LL-SC 锁已经避免了多个处理器在锁被释放后试图获取它时发出作废信号，在流量上加号锁和简单的 LL-SC 锁没有大的区别。（简单的 LL-SC 锁稍微差一些，这是由于当一个处理器在它的条件存放上成功时，另外一个作废和一组读扑空会出现。）在这两种锁上面的关键区别在于公平性。

像简单的 LL-SC 锁，在释放中加号锁也有一个读流量的问题。原因是所有进程在相同的变量（now-serving）上踏步等待。当那个变量被释放写入后，所有处理器的缓存中的拷贝被作废，它们都要引发一次读扑空。这些读扑空在有些总线上可能结合起来，但如果没有结合功能或者结合失败，则会引起不必要的流量。一种降低这种突发性读扑空流量的方式是引入一种形式的回退。我们不想用指数性回退，因为在锁被释放时，我们不想让所有处理器都回退，从而过一会儿谁都不想尝试获取它。一种有前途的技术是让每个处理器在试图读 now-serving 计数器时回退一段正比于它期望轮到它所需的时间——即正比于它的票号和它上一次读到的 now-serving 值的差。作为另一种方案，基于数组的锁完全消除了这种在释放时额外的读流量，办法是让每个进程在不同的单元上循环。

基于数组的锁 这里的想法是用一种 fetch&increment，来获得一个惟一的单元而不是值，然后在这个单元上做忙等待。如果有 p 个进程可能争用一把锁，那么这把锁的数据结构包含一个含有 p 个单元的数组，供进程在上面循环等待，理想情况是不同的单元分布在不同的存储块上，以避免伪共享。这里，获取方法就是用一个 fetch&increment 操作来获得该向量中下一个可用的单元（带回卷），等待方法则是在这个单元上循环，释放方法是向下一个单元写入一个表示“解锁”的值。这种释放使得在下一个单元上等待的处理器发生缓存块的作废；它后面的读扑空告诉它，它已经得到了这把锁。如同加号锁的情形，在扑空后不需要做 test&set，这是由于当锁被释放时只有一个进程被通知。显然，这种锁也是 FIFO，因此是公平的。它的无竞争时延可能类似于 test-and-test&set 锁（一种 fetch&increment，跟着一个对所分配数组单元的读），并且它显然潜在地要比加号锁更具有可扩展性，这主要是因为只有一个进程发生读扑空。基于同样的原因，不同于加号锁，它不需要任何形式的回退来减少流量。在基于总线的机器上，它的惟一缺点是它用 $O(p)$ 空间，而不是 $O(1)$，但对于较小的 p 和比例常数，这通常不是一个很重要的缺点。对于分布存储机器，它有一个潜在的缺点，我们将在第 7 章讨论它和克服它的锁算法。 347

8、性能

让我们简略地考察 SGI Challenge 上不同的锁的性能，如图 5-30 所示。所有的锁用 LL-SC 实现（由于 Challenge 只是提供这些，而没有提供原子性指令）。所得到的结果是基于前面提到过的微基准测试程序的一个更参数化的版本，其中一个进程不仅可以在临界区插入延迟，在锁的释放和它的下一次尝试之间也可以插入（如同真实程序会发生的）。即，代码是一个以下面为主要部分的循环：

```
lock(L);
critical_section(c);
unlock(L);
delay(d);
```

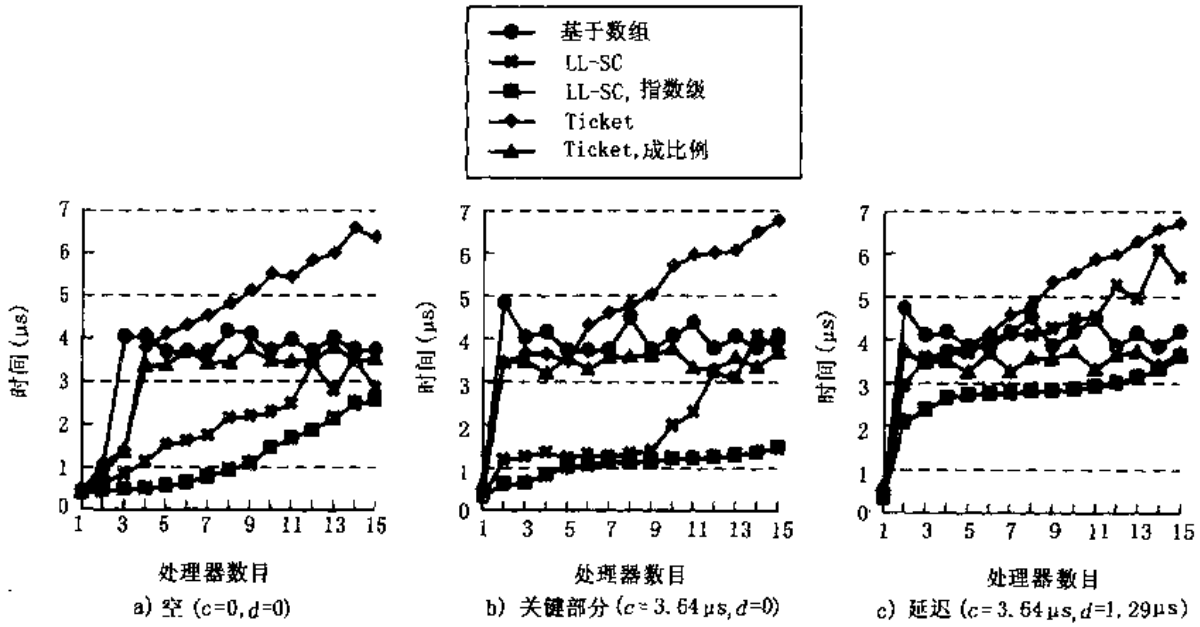


图 5-30　对于三种不同的情形，SGI Challenge 上锁的性能

考虑三种情形：1）$c=0$，$d=0$；2）$c=3.64\mu s$，$d=0$；3）$c=3.64\mu s$，$d=1.29\mu s$。它们分别称为空临界区情形、非空临界区情形和带有延迟的非空临界区情形。延迟 c 和 d 以处理器周期数为单位插入到代码中，转换为这里的微秒数。和先前讨论过的一样，延迟 c 和 d（乘以每个进程获取锁的次数）要从总的时间中减掉，它只是来测量一定数量的锁的获取和释放所花的时间（见习题 5.15）。

考虑空临界区的情形。比较图 5-30 和图 5-29，首先观察到的是所有其他锁的确都比测试并设置锁要好，这是已估计到的[○]。第二个观察是，简单的 LL-SC 锁实际上看来性能要好于比较复杂的加号锁和基于数组的锁。对这些锁来说，它们所遇到的竞争不像 test&set 锁遇到的那么多，性能主要由释放和成功获取之间的总线过程数来决定。尤其对于较少处理器的情况，LL-SC 锁表现特别好的原因是它们的不公平性，体系结构的相互作用利用了这种不公平性！特别是，当一个进程通过一个写操作释放了一把锁，紧跟着用一个读（LL）试图进行下一次获取，这个读和后续的条件存放可能在它的缓存中成功，先于其他处理器能越过总线来读这个存储块。（SGI Challenge 上的偏向实际上要更严重，这是由于释放的处理器能够从它的写缓冲区中来满足它的下一次读，甚至在对应于释放写的排他读达到总线之前。）锁的传递很快并且性能也很好，但同样的处理器可能重复获得自己释放的锁。随着处理器数和对总线竞争的增加，最后释放者的条件存储成功获得总线的机会减少，因此自传送的可能性减少。除此以外，总线流量会由于作废和读扑空增加，因此每次锁转移的时间增加。指数型回退有助于减少流量的突发性，从而也减缓了扩展的速率。非空的临界区（$c=3.64$，$d=0$）对此会有进一步的帮助。

对于在临界区内外都有延迟的情形（$c=3.64$，$d=1.29$），即使处理器数不多，LL-SC锁

○ 这里的测试并设置是通过 LL-SC 如下模拟的：每当一个条件存放失败，就向同一存储块的另一个变量做一个写操作，于是就好像 test&set 那样引起作废。这种模拟的方法可能导致的性能要比真正用 test&set 原语差，但它反映了趋势。

的性能也不怎么好。这是由于处理器在释放锁后，试图再次获取锁之前需要等待，很可能使得其他等待着的处理器先于它获得了该锁。自我传递不太可能，因此即使在两个处理器的情况，锁的传递也是比较慢的。有意思的是，在处理器数量少，解锁加锁之间有延迟 d 时，回退的使用表现的性能特别差。这是由于当一个处理器释放了锁，在等待 d 的期间，所有其他处理器都在回退周期，根本还没有试图来取锁。在 $d=0$ 情形下，释放锁的处理器马上又获取锁，这种情况在处理器数较少时特别明显。从上面可以看到，回退技术必须慎用才可能成功。

348 - 349

考虑另外一些锁。它们是公平的，因此每次都传递到不同的处理器，且在传递的关键路径上涉及到总线事务。因此，即使只有两个处理器，它们都以一个跳转开始。每次传递都在关键路径上大约跳三个总线事务。时间方面的实际区别取决于产生哪种总线事务以及它们有多少时延能被隐藏起来，不被处理器可见。不带回退的加号锁可扩展性相对较差：当所有处理器试图读 now-serving 计数器时，在释放和由正确处理器读得之间的总线事务的期望数是 $p/2$，导致在锁传递的关键路径上的线性性能损失。采用比例回退技术，正确的处理器可能是在释放后第一个发出读的处理器，这样每次传递的时间就是常数，不随 p 变化。由于只有正确的处理器发出一个读，基于数组的锁也有较好的可扩展性。

这些结果表现了在确定锁的性能方面体系结构诸要素之间相互作用的重要性。它们还表明，只要总线有足够带宽，简单的 LL-SC 锁在总线上的性能会相当好。在这个特定的机器上，因为总线带宽相当高，非公平 LL-SC 锁的性能对于多于 16 处理器的情形，不会比更复杂的锁差多少。当采用回退技术来降低流量后，简单 LL-SC 锁在所有情形下都有最好的平均锁传递时间。然而，这些结果也表明了在评估同步算法方面有效实验方法的难度和重要性。空临界区展示了某些有趣的效果，但有意义的比较取决于在实践（真实应用）中同步模式的具体情况。例如，临界区和延迟大小，对于自传递的作用，对比较公平和不公平锁来说，影响是实质性的。空情形不具有真实代表性，但有重要的方法论意义。人们还做过一个实验，用 LL-SC 同时通过附加变量来保证处理器（公平性）获得锁的循环，表明其性能类似于加号锁。这证实了非公平和自传递在处理器不多时的确是较好性能的原因。特别是，如果期望公平，带有比例回退的加号锁和基于数组的锁在基于总线的机器上表现得很好。

9. 免锁的、非阻塞的以及免等待的同步

当机器用在多道程序设计的环境下时，涉及到同步的还有对其他一些性能的考虑。在这种环境下，其他进程会不时执行一段时间；即使我们独占这个机器，后台驻留程序也会周期运行，进程会发生缺页，会出现 I/O 中断以及进程调度器只能基于应用需求不完整的信息来作出调度决定。这些事件能引起进程推进速度在很大的范围内变化。一个重要的问题是，当其中一个进程被减慢，并行程序作为一个整体会如何减慢。对于传统的锁来说，这个问题可能很严重：如果一个进程持有一把锁，在它的临界区中停止或者减慢，所有其他进程可能都要等待。这个问题在操作系统调度器的研究方面受到了广泛的关注。在有些情况下，人们试图避免剥夺持有锁的进程的控制权。另外一些研究认为基于锁的操作不健全，应该避免；例如，如果一个进程在持有锁时死掉，其他进程就都没法推进。人们观察到，大多数加锁-解锁操作都用来支持在典型数据结构和对象上的操作，那些结构是由多个进程共享的，例如，更新一个共享的计数器或者操作一个共享的队列。这些在数据结构上的高层操作能够直接用原子原语实现，而不要用锁，如同先前讨论 LL-SC 所提到的那样。

350

一个共享数据结构被称为是*免锁的*，如果在它上面定义的操作不要求在多条指令上互斥。如果在数据结构上的操作，能够保证某个进程在有限时间里，即使其他进程停止也能完成它的操作，这种数据结构就是*非阻塞的*。如果操作能担保每一个进程都能在有限的时间里完成其操作，该数据结构就是*免等待的*（Herlihy 1993）。针对这种数据结构的理论和实践，人们做过一些研究，包括实现它们的基本原语（Herlihy 1988）、将顺序操作转换为非阻塞并发操作的通用技术、各种专门的免锁数据结构（Valois 1995; Michael and Scott 1996）、操作系统实现以及对系统结构支持的提议等等。这里基本的出发点是要实现对共享数据结构的更新，先读出其中一部分，做一拷贝，更新该拷贝，然后在没有已经发生冲突更新的情况下才执行一个操作，以确认这个改变（这会使我们想到 LL-SC）。作为一个简单的例子，考虑一个共享计数器。计数器读到一个寄存器中，给寄存器拷贝加上一个值，结果放到第二个寄存器。然后，只是当共享计数器的值和原来是相同时，才用一个比较/交换来更新共享的计数器。对于更复杂的链表数据结构，就要创建一个新的元素，如果插入仍然有效，才将它链到共享表中。这些技术用来限制共享数据结构处于非一致状态的窗口的大小，于是它们增强了健壮性；当然，要把它们做得高效可能是困难的。

根据用不同的原子交换操作来实现同步变量访问的时间复杂性，人们发现了这些原子交换操作的一些性质。特别是，人们发现像 test&set 和 fetch&op 那样的简单操作不足以保证由处理器访问一个同步变量的时间独立于处理器个数，而更复杂一些的操作，例如比较/交换和交换两个存储单元的值，就能够达到这种保证（Herlihy 1988）。

[351] 讨论了基于总线机器的互斥方式后，下面来考虑点对点事件同步，然后是栅障事件同步。

5.5.4　点对点事件同步

在一个并行程序中的点对点同步，通常的实现方式是用普通变量作为标记，在上面做忙等待。如果我们要用阻塞，不用忙等待，则我们能用信号灯，就像在并发程序设计和操作系统中所用的那样（Tanenbaum and Woodhull 1997）。

1. *软件算法*

标记是控制变量，专门用来通报同步事件的出现，而不是要传递值。如果两个进程在共享变量 a 上有一种生产者－消费者关系，那么就能使用如下一个标记来管理同步变量：

P_1	P_2
a = f(x);　/*set a*/	while (flag is 0) do nothing;
flag = 1;	b = g(a); /*use a*/

如果我们知道，变量 a 初始为某个值（比如，0），将被这个生产事件变到一个我们感兴趣的一个新值，那么我们能够用 a 本身作为同步变量，如下所示：

P_1	P_2
a = f(x);　/*set a*/	while (a is 0) do nothing;
	b = g(a); /*use a*/

这就消除了对一个单独标记变量的需要，节省了对那个变量的读和写，可能的代价是程序的可读性和可维护性。

2. *硬件支持：满－空位*

这种特别标记值的想法在有些研究型机器中得到进一步发展（尽管主要在物理分布存储

的机器里)，以提供对细粒度生产者－消费者同步的硬件支持。让存储器的每个字都和称为满－空位的状态相联。这一位被置1，表示“满”，即该字装有新数据（即写操作上），置0，如果这个字被消费该数据的处理器“腾空”（即读操作后）。字级别的生产者－消费者同步然后可以如下完成。当生产者进程要向单元中写，如果看到满－空位是空，就将它置满。消费者要读，如果看到满，就置空。硬件保证带有对满－空位操作的读和写的原子性。给定满－空位，我们前面的例子就可以写成如下没有踏步循环的样子：

352

P_1	P_2
a = f(x); /*set a*/	b = g(a); /*use a*/

满－空位引起关于灵活性的一些问题。例如，它们难以对付单生产者－多消费者同步，或者生产者在消费行为之前多次更新值的情况。还有，所有读和写都应该用满－空位，还是只是编译到特殊指令的操作用？后者要求在语言和编译器中有支持，但前者将同步强加到所有单元访问中（例如，它不允许迭带方程求解器中的异步放松，见第2章），局限性太大。鉴于这些原因以及硬件代价，在大多数商用机器中满-空位都没有受到青睐。

3．中断

另一种重要的事件是中断，主要源于需要处理器关注的I/O设备。在单处理器机器中，中断该谁来处理不是一个问题，但在SMP中，任何处理器都可能承担中断任务。除此以外，还有一个处理器向另一个处理器发中断的情况。在早期的SMP设计中，提供特殊硬件来管理每个处理器上进程的优先级，将I/O中断送到低优先级的处理器上。这样的做法证明价值不大，多数现代机器用的是简单的仲裁策略。除此以外，通常存在一种存储映射的中断控制区，于是可以在内核层次，通过在相关的地址上写入中断信息，使得任何处理器能中断任何其他处理器。

5.5.5 全局（栅障）事件的同步

最后，我们考察一下在基于总线机器上的栅障同步问题。栅障软件算法的实现通常都用锁、共享计数器和标记单元。首先我们看 p 个进程之间的一种简单的栅障，称为集中式栅障，它只用一把锁、一个计数器和一个标记位。

1．集中式软件栅障

用一个共享的计数器来记录到达栅障的进程数，每一个到达的进程使它加1。这些加1操作必须是互斥的。在对该计数器作了加1操作后，进程检查计数值是否等于 p，即它是否是最后一个到达的进程。如果不是，它就进入忙等待状态，等在和栅障相关的标记位上；如果是，它就置标记位，释放 $p-1$ 个等待着的进程。于是，我们可能提出栅障算法的一个简单实现如下：

353

```
struct bar_type {
  int counter;
  struct lock_type lock;
  int flag = 0;
} bar_name;

BARRIER (bar_name, p)
{
```

```
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)
      bar_name.flag = 0;                /*reset flag if first to reach*/
    mycount = bar_name.counter++;       /*mycount is a private variable*/
    UNLOCK(bar_name.lock);
    if (mycount == p) {                 /*last to arrive*/
      bar_name.counter = 0;             /*reset counter for next barrier*/
      bar_name.flag = 1;                /*release waiting processes*/
    }
    else
      while (bar_name.flag == 0) {}; /*busy-wait for release*/
  }
```

2. 带有感应逆转的集中式栅障

能看出上面的栅障有什么问题吗？我们会发现有一个问题。它发生在同样栅障变量上连续进行栅障操作的时候——例如，如果每个处理器执行下面的代码：

```
some computation...
BARRIER(bar1, p);
some more computation...
BARRIER(bar1, p);
```

第一个进入栅障的进程第二次进入时重新初始化栅障计数器，因此没问题。问题在于标记位。为了从第一个栅障出来，进程踏步在标记上直到它为 1。看到标记为 1 的进程将离开栅障，进行后面的计算，再次进入栅障。然而，假设一个处理器 P_x 没有来得及看见标记变成了 1，其他进程又进入了第二个栅障；这种情况是可能的，例如由于等待时间太长而被操作系统交换出去了。当它再次被交换进来时，它就要继续等在标记位上。同时，其他进程可能已经第二次进入栅障，它们中的第一个将把标记置 0。现在，这个标记位只有当所有进程都第二次进入栅障后才能置 1，但这不可能发生，因为 P_x 没办法离开第一次栅障的反复循环。

354

如何解决这个问题？我们要做的是防止一个进程在所有其他进程都离开先前的栅障操作之前再次进入这个栅障。一个办法是用另一个计数器来记录离开了栅障的进程，在这个计数器未达到 p 之前不让进程在新的栅障操作中对标记复位。然而，对这个计数器的操作会引起进一步的时延和争用。另一方面，在目前的框架下面，我们没办法等所有进程都到达栅障后再将标记置 0，因为我们是将它置 1 来释放进程的。一个较好的解决方案是避免显示重置标记的值，而让进程等待标记在后面的栅障事例中获得一个不同的释放值。例如，进程可能在一次事例中等待标记变成 1，在下一次事例中等待变成 0。可以用一个私有变量来跟踪在本次事例中要等待的值。根据栅障的语义，一个进程不能超越其他进程多出一个栅障，于是我们只需要两个值（0，1）在其间交换。因此我们称这种方法为*感应逆转*（sense reversal）。现在，在前面的例子中，当第一个进程到达栅障时，标记不需要复位；相反，停在老栅障上的进程仍然等待标记达到老的释放值，而进入新事例的进程等待另外的释放值。当所有进程到达了新栅障事例，标记的值只改变一次，因此在老事例上的进程看到之前将不会改变。下面是一段对应的代码：

```
BARRIER (bar_name, p)
{
   local_sense = !(local_sense);      /*toggle private sense variable*/
```

```
    LOCK(bar_name.lock);
    mycount = bar_name.counter++;        /*mycount is a private variable*/
    if (bar_name.counter == p) {         /*last to arrive*/
      UNLOCK(bar_name.lock);
      bar_name.counter = 0;              /*reset counter for next barrier*/
      bar_name.flag = local_sense;       /*release waiting processes*/
    }
    else {
      UNLOCK(bar_name.lock);
      while (bar_name.flag != local_sense) {};  /*busy-wait for
                                                  release*/
    }
  }
```

注意在计数器增量后锁不是立刻被释放的，锁的释放只是在条件被求值后发生；这里的原因在习题 5.18 中可以发现。我们现在有了一个正确的栅障了，可以连续重用任何次数。剩下的问题就是我们下面要考虑的性能。(注意，保护计数器增量的 LOCK/UNLOCK 可以被简单的 LL-SC 或其他原子增量操作替代，效率更高。) 355

3. 性能

我们所追求的栅障性能指标类似于锁，包括：

- 低延迟。(关键路径的长度要小) p 个处理器通过栅障所需的相关操作的链和总线事务应该尽量小。
- 低流量。由于栅障是全局操作，很可能许多处理器会试图在同一时间执行一个栅障算法。这个算法应该减少总线事务的总数（无论是否在关键路径上），从而减少可能的争用。
- 可扩展性。时延和流量应该只随处理器数缓慢增加。
- 低存储代价。我们当然希望存储代价低。
- 公平性。我们应该避免同一个处理器总是最后一个离开栅障的情形（或者我们可能要保持 FIFO 序）。

在前面所描述的集中式栅障中，每个处理器访问一次锁，因此关键路径的长度至少和 p 成比例。下面考虑总线流量。为了完成它的操作，一个涉及 p 个处理器的集中式栅障要做 $2p$ 次总线事务来获得锁和对计数器加一，两次总线事务让最后一个处理器对计数器复位，写入释放标记，另外还需 $p-1$ 次总线事务来在标记作废后再读它。注意这要好于 p 个处理器要获取一把 test-and-test&set 锁的情况；因为在后面一种情况下，每次释放（共有 p 次）都引起作废，结果是 $O(p)$ 进程要在此又一次完成 test&set 操作，于是就导致 $O(p^2)$ 总线事务。然而，如果许多处理器同时到达栅障，源于这些竞争性总线事务的竞争可能是很大的，因此，栅障的代价可能很高。

4. 栅障算法针对总线的改进

集中式栅障的部分问题是所有处理器争用相同的锁和标记变量。为对付这样的问题，我们可以构造一种栅障，只引起少数处理器争用相同的变量。例如，处理器能通过一种软件结合树来指示它们到达了栅障（见 3.3.2 节）。例如在一个二元结合树中，只有两个处理器在树的节点上相互通报它们的到来。这样，只有两个处理器会访问一个给定变量。在有着多重并行通路的分布式网络中，诸如那些可扩展机器中所有的，一个结合树能比集中式栅障性能 356

好得多。这是由于两对不同的处理器能在网络的不同部分并行通信。然而，对于像总线那样的集中式互连来说，即使处理器对通过不同的变量通信，它们都要产生总线事务，于是就要串行化，且在这条总线上争用。由于含有 p 叶节点的二元树大约有 $2p$ 个节点，一个结合树所需要的总线事务数和集中式栅障类似。结合树方式还有较高的时延，这是因为除了它也要求 $O(p)$ 串行化总线事务外，即使没有总线串行化，每个处理器还至少等待 $\log p$ 步才能从叶节点到达树的根，每一步都有实质性的工作。结合树在总线上的优点是它不用锁，而只是简单的读写操作，如果总线上的处理器数目很大，这可能对它的较大的非竞争时延是个补偿。然而，如图 5-31 所示，简单集中式栅障在总线上表现很好。图 5-31 中示出的其他一些栅障将在第 7 章讨论可扩展机器时和树栅障一并讨论。

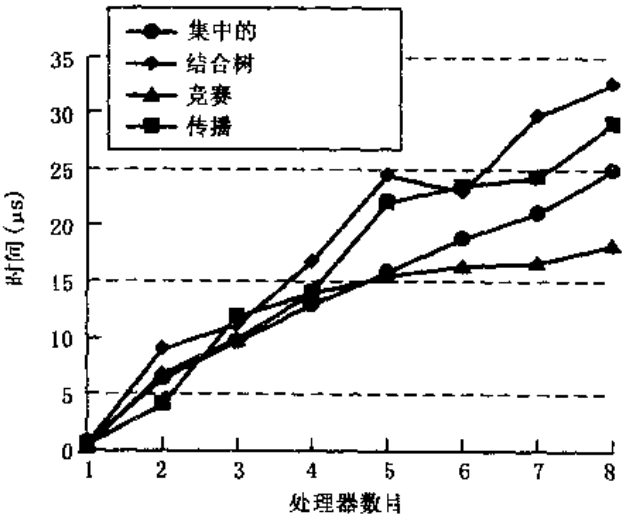


图 5-31 SGI Challenge 上一些栅障的性能。这里性能的度量是在一个循环中连续执行许多栅障，每次通过栅障的平均时间（在栅障之间没有工作或延迟）。结合树栅障在其关键路径上有较高的时延，这使得在总线上它的性能不好，因为在这种情况下它的流量和争用优势体现不出来

5. 硬件原语

由于集中式栅障用锁和普通的读和写，所需的硬件原语取决于所用的锁算法。如果一个机器不能很好支持原语，结合树栅障也可以在基于总线的机器上用得很好。

357

一个特别的总线原语可以用来减少在集中式栅障中读扑空引起的总线事务的数量（也对处理器踏步等待同一变量的高度争用的锁有用）。这个优化利用了如下事实，当一个变量在释放作废后，所有处理器发出对它的读扑空。不同于所有处理器发出分别的读扑空总线事务，一个处理器能够管理总线，在看到对同一单元的读扑空（由另一处理器发出，首先到达总线）的响应后，将自己的读扑空在到达总线前取消，然后简单地从总线上取得返回数据。在最好的情况下，这种做法能将读扑空总线事务从 p 降到 1。

6. 硬件栅障

如果提供一个单独的同步总线，如讨论锁时提到的，它也可以用来支持栅障。这会将流量和冲突从主要总线上分离出来，从而得到较高性能的栅障。从概念上讲，一条“线与”线路就足够了。当到达栅障时，处理器将它对这条线的输入置 1，然后等到输出为 1 后再向前推进（在实际中，重用栅障要求的线不只一根）。这样一种单独的栅障硬件机制在栅障频率很高的情况下特别有用，这种情况可能出现在由并行化编译生成的程序的内层循环里，在每次内层循环完成后都需要一次全局同步。不过，它的价值在实际中尚不清楚，而且在机器中的处理器只有部分参与栅障时难于管理。例如，不容易动态地改变参与栅障的处理器的数量，在操作系统使得进程在处理器之间迁移时也不容易改动参与处理器的配置。如果多个参

与进程运行在同一个处理器上也会造成麻烦。因此，当前总线型多处理器不倾向于提供特别的硬件支持，而是用锁和共享变量来构造软件栅障。

5.5.6 同步问题小结

有些基于总线的机器对诸如锁和栅障等同步操作提供直接的硬件支持。然而，关于灵活性的考虑导致大多数现代设计人员只在硬件上提供简单的原子操作，通过用它们来构成软件库，来得到高层同步操作。应用程序人员通常用这些库，可以不需要了解那些由机器支持的底层原子操作。这些原子操作可能是通过单条指令实现的，或者通过仔细推敲的读写指令对来实现的，如同装载-加锁和条件存储。后者有较大的灵活性，这使得它们日益流行。我们已经看到了一些同步原语，算法和系统结构细节之间的相互作用。在后面几章中，当我们讨论可扩展共享地址空间机器时，这种相互作用会进一步体现出来。 358

5.6 对软件的影响

迄今为止，我们一直关注着基于总线的、具有高速缓存一致性的多处理器的高层体系结构的问题，以及工作负载的特性对体系结构和协议折中的影响。现在，我们转回来探究这些小规模机器的体系结构是如何影响并行软件的。具体来说，我们不再是在固定工作负载的条件下研究提高机器性能或协议的方法，而是在给定机器配置情况下去探究怎样提高并行程序的性能。改进同步算法，以减少通信量和时延就是其中一个方面，下面先让我们更一般地看看并行程序设计的过程。

关于负载平衡和固有通信的一般性技术在第 3 章中已讨论过了，这些技术也适用于一致性高速缓存的机器中。除此以外，在这类机器有一种一般性划分原则，即在对计算任务的分配时，力图使得只有一个处理器对一组数据做写操作（至少在单个计算阶段中）。在很多计算问题中，处理器要读某一共享数据结构，但写另一个数据结构，因此这种原则可以应用于很多计算领域。例如，在 Raytrace 应用中，处理器读的是场景，写的是图像。常常可以有两种选择，一是划分计算任务以至于不同处理器写的是不相交的数据结构，但从共享数据结构中读；或是从不相交的数据结构中读，但写到同一块共享存储区。在所有其他因素相同的情况下（诸如负载均衡，程序设计的复杂度），我们认为通常应该避免写共享。写共享不仅能引起作废，从而导致缓存扑空和流量增加，而且如果不同的处理器写同一字，很可能这样的写操作必须被诸如锁一样的同步机制保护起来，所带来的开销于是更大。

通信的结构比较单纯：有一个集中式存储器，没必要用显式的存储器到存储器的数据传送，因而所有的通信都是通过 load 和 store 指令隐式完成的，这样的指令会导致缓存块的传送。映射不是一个问题（除了尽量使进程在不同处理器间少做迁移），它由操作系统全权处理。人们最关心的问题是在性能协调步骤中利用数据局部性和控制附加的通信，特别是利用时间和空间局部性来减少高速缓存扑空，从而减少时延、通信量和在共享总线上的争用。

由于主存是集中式的，时间局部性体现在处理器的高速缓存中。第 3 章介绍了基于总线机器的工作集曲线，其特性如图 5-32 所示。所有和容量有关的扑空都要出现在同一个总线上，从而要反映到存储器中，其代价和一致性扑空一样高。即使有无限大的高速缓存，另外三类扑空也会产生总线流量。开发时间局部性的主要目的就是让工作集能放到高速缓存的层次结构中，所用的技术和第 3 章中讨论过的相同。 359

对空间局部性来说，一个集中式的存储器使数据在主存中的分布和存储分配是无关紧要

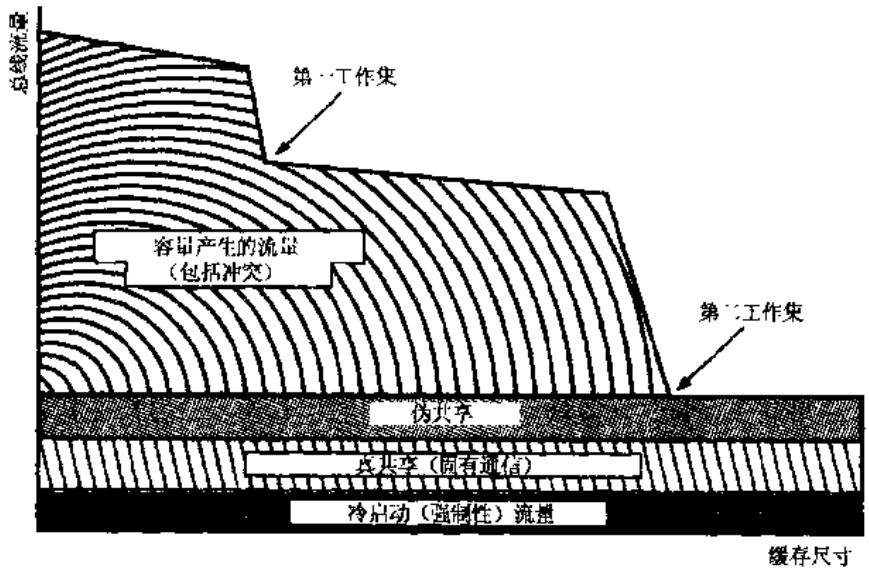


图 5-32 在共享总线上的数据流量及其各种成分随缓存大小的变化。拐点指出程序的工作集

的（仅仅在存储器不同模块之间安排数据的交叉存取以减少争用可能是应该考虑的，这一点和单处理器情形一样）。空间局部性不好的效果是存储碎片（也就是将不必要的数据取到了缓存块中）和伪共享。这里的原因是通信和一致性的粒度都是缓存块，要比一个字大。前者引起存储碎片，后者引起伪共享。（这里我们假定没有用到消除伪共享的一些技术，例如设置子块的脏位，因为它们在大多数实际机器中都没采用。）我们考察一些减轻这些问题的方法和有效地利用大缓存块预取的结果，以及通过更好的数据的空间组织去减轻缓存冲突的一些方法。在程序员的“技巧锦囊”中能发现很多这样的方法。下面所列出的是一些最通行的。

- *任务的分配应减少访问模式的空间交叉*。任务的分配应该是让每个处理器去访问一片相邻的数据区。例如，把一个 n 个元素数组的计算分配给 p 个处理器处理，最好的分配是让每个处理器访问 n/p 个相邻的元素，而不是对元素进行精细地交叉分配。这样提高了数据的空间局部性，减少了缓存块的伪共享。当然负载均衡或其他一些约束可能迫使我们不能这样做。
- *组织数据以减少访问模式的空间交叉*。在第 3 章中我们有一个在方程求解器内核中的例子，那里我们用高维的数组使一个数组在一个处理器中的分配在地址空间上是连续的，于是在物理上分布的存储器中，本地分配能按页的粒度来进行。这种方法也可以帮助减少伪共享，数据传送的存储碎片和冲突扑空，如图 5-33 和图 5-34 所示，从而减少扑空和总线上的流量。一个比网格元素大的缓存块可能跨越面向列的划分的边界，如图 5-33a 所示。如果一缓存块比两个网格元素还大，它就可能引起由伪共享导致的通信。如果我们暂且假定在算法中不存在固有的数据通信，这一点是很容易看到的；例如，假设在每一次遍历中一个进程只是简单地把一常量加到该进程所分配到的网格元素上，而不是执行一次相邻元素的计算。现在，即使缓存块有两个网格元素那么大（或更大），跨越一个划分的边界，由于不同的处理器向其中不同的字做写操作，也将会出现伪共享。这也将导致在通信中的存储碎片，因为一个读其边界元素但扑空了的进程会去取在同一缓存块、但在其他处理器划分中的其他元素，但那些数据是它不需要的。图 5-34 解释了冲突扑空的问题。在以上所有

360

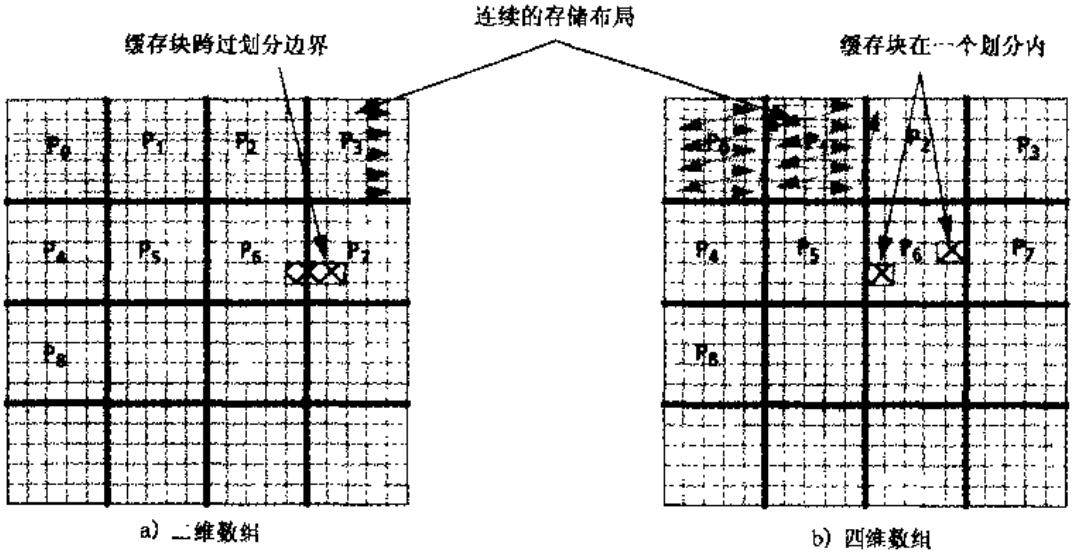


图 5-33　利用高维数组保持划分在地址空间中的连续，以减少伪共享和存储碎片。在二维数组的情形中，跨划分边界的缓存块存储碎片（扑空导致带入其他处理器划分中的无用的数据）和伪共享。四维数组表示使得划分连续，从而缓解了这些问题

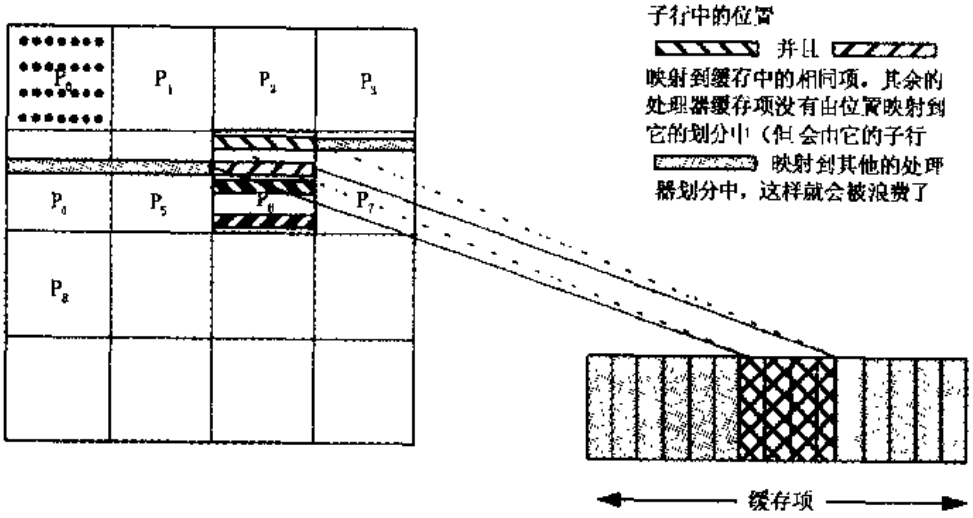


图 5-34　由二维数组表示在一个直接映射缓存中引起的高速缓存映射冲突。这个图表示最坏的情况，在处理器划分中相继两个子行的间隔（即二维数组中一行的大小）正好等于缓存的大小，于是相继的两个子行映射到缓存中的相同的位置。对每一子行的访问将替换出前面的子行。处理器在它的划分上做下一次遍历时，它会在它引用的每个缓存块上扑空，即便缓存大得能装下整个划分也是如此。受网格大小、处理器数和缓存大小的影响，我们可能遇到许多不如最坏情况差的中间状况。由于缓存大小是 2 的幂次，将待分配的数组的维展定为 2 的幂次是不利的

情况下的问题都是划分的不连续性。于是，一个简单的数据结构变换（如图 5-33b 所示）就帮助我们在方程求解器内核中解决了所有与空间局部性相关的问题。图 5-35 表示的是在 SGI Challenge 上，针对 Ocean 和 LU 应用用高维数组表示网格或分块矩阵对性能的影响。冲突和伪共享对于单处理器和多处理器性能影响的差别在此是很明显的。

- 防止冲突扑空。针对发生在网格求解器的冲突扑空，图 5-34 表明了由于高速缓存的

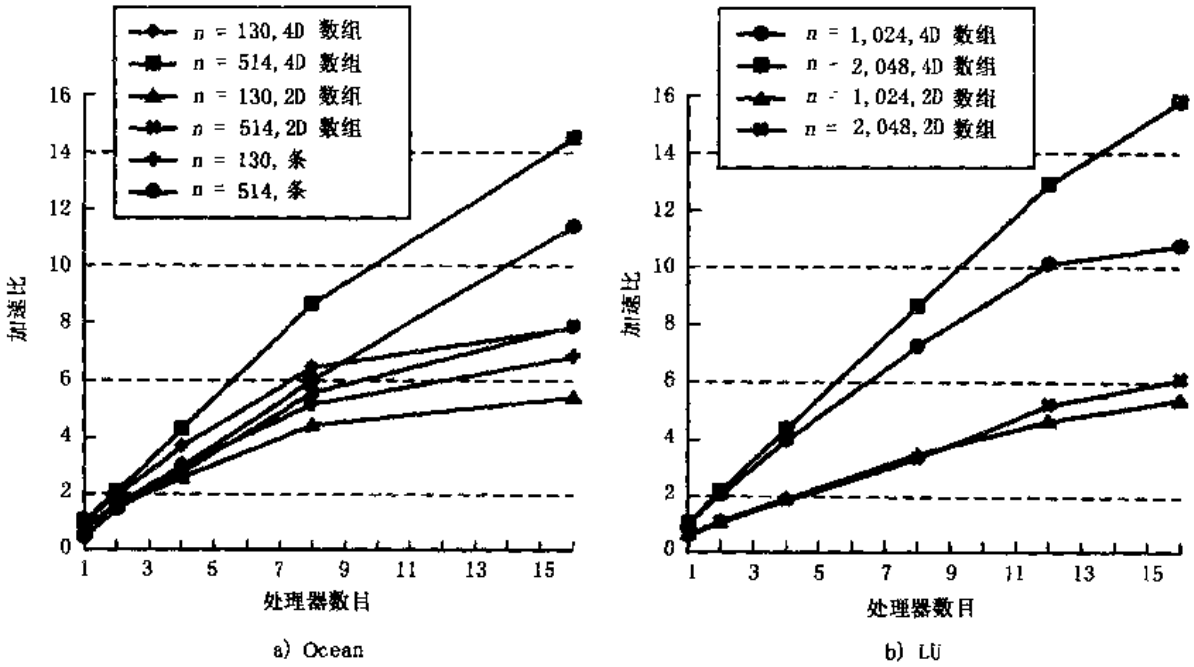


图 5-35 在 SGI Challenge 上用四维数组和二维数组表示矩阵数据对性能的影响。所显示的数据针对 Ocean 和 LU 的不同的问题规模、对于 Ocean，"条状"表示按连续行的条状划分（其中二维或四维数组无关紧要），其他情形表示类方块状划分

大小通常是 2 的整次幂，当数组的存储分配规模也是 2 的幂时高速缓存冲突的严重情况。于是，在应用中即使数组的逻辑大小是 2 的整次幂，我们也常常分配一个更大的数组，但只访问其中有效的部分。不过这种策略会受到物理上分布存储分配页面粒度（也是 2 的整次幂）的影响，所以我们必须小心。在此例中的高速缓存映射冲突发生在一个以可预取模式访问的单一数据结构中，因而能在一定数据结构下得到缓解。当映射冲突发生在不同的主数据结构之间时，映射冲突是很难避免的（例如，通过由 Ocean 应用程序使用的不同网格），这种情况可能只利用随意填充和对准来缓解。然而，在一个共享地址空间上，当它们发生在一些看似无害的共享变量和数据结构上时（程序员通常不太注意这些变量和数据结构），常常是特别隐伏的。例如，在一个直接映射的高速缓存中，一个频繁被访问的指向重要数据结构的指针可能会与一个在同一计算中也被频繁访问的标量变量发生缓存冲突，导致很大的通信量。幸运的是，在现代两级高速缓存中（容量大并且是组相联）这些问题出现很少。总之，如果不把注意力放在减少冲突扑空上，那么在开发局部性上的努力就会白费了。

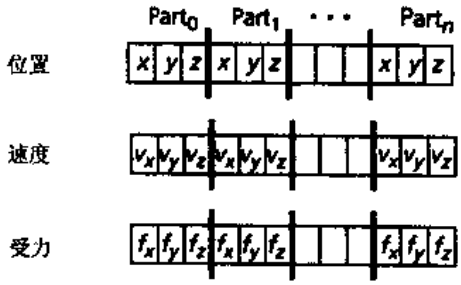
- 每个处理器都用单独的堆。这是值得考虑的一种方法，每个处理器（或进程）都有自己独立的堆用来动态地进行数据分配。否则，如果一个程序要访问很多非常小的存储区，被不同处理器使用的数据就可能落在同一缓存块中。
- 通过数据拷贝来提高空间局部性。如果一个处理器要反复用到一组本来不是连续分配的数据，我们可以考虑做适当的数据拷贝，使得它们在相应的计算周期中是连续分配的，从而提高数据的空间局部性，减少缓存冲突。这里要注意的是，拷贝数据需要访问主存因而有一定的开销；同时，如果那些数据可能都驻留在高速缓存中，这种方法就无效了。例如，在分块矩阵的因子分解或乘法的计算中，用二维数组表示矩阵，

361 ~ 363

矩阵中的每一分块在地址空间中是不连续的（就像在方程求解器内核中的一个划分）。然而，二维表示能使程序设计更简单。因此一种常见的做法是用二维数组，但在相关的计算阶段将分给其他处理器的数据拷贝到自己的一片连续的临时数据区中，来减少冲突扑空。当然，拷贝的代价需要和冲突减少的利益权衡。在有关粒子的应用中，当一个粒子从一个处理器的划分移动到另一处理器时，将和粒子相关的数据移动到一个连续、紧凑的数据区，可以提高空间局部性。

- *填充数组*。并行程序设计者经常创建一些以进程标识号为索引的数组。例如，为了跟踪负载平衡的情况，可能要维持一个 p 元整型数组，数组的每一项记录了相应处理器已经完成的任务数。由于缓存的一个块可能容纳这个数组中的大部分元素，而这些元素经常要被不同的处理器更新，所以伪共享就是一个严重的问题。一个解决的办法是，用一些无用元素填充在数组的有用项之间，使得有用元素之间的数组段和缓存的块一样大（预见到程序可能在不同的机器上运行，也可以填充得更大一些），然后让数组和一个缓存块对齐。但是填充许多大数组将导致存储的大量浪费，而且导致数据传输中的存储碎片。一个更好的策略是把分给一个进程的这种性质的变量集中到一个记录中，将该记录填充至一缓存块，然后创建一个以这样的记录为元素、以进程标识符为索引的数组。
- *决定怎样组织记录数组*。假设我们要表示一些逻辑记录，如在 Barnes-Hut 引力模拟中的粒子。我们应该是用一个数组来表示粒子集，数组有 n 项，每一项为一个粒子的记录，包含粒子的位置、速度、受力、质量等域，如图 5-36a 所示的那样呢？还是把它们表示成若干 n 元数组，每个数组对应一个域，如图 5-36b 所示的那样？为 CRAY 那样传统的向量机写的程序常按一个对象的性质或域分类，每一类用一个数组（向量）表示——实际上，甚至对域中的每个物理维（x、y 或 z）都分别用一个数组（向量）表示。当按域访问数据时，如访问所有粒子的速度，由于访问跨距是存储单元，减少了存储体冲突，因而提高了向量操作的性能。但是，在缓存一致性的多处理器中，需要考虑一些新的折中，最好的数据组织方法取决于访问模式。

a) 按粒子组织



b) 按属性或域的粒子组织

图 5-36　基于记录数据的不同组织方式

在 Barnes-Hut 应用中，粒子状态的更新和受力计算阶段揭示了一个引人注目的冲突现象。先考虑更新阶段。在这一阶段处理器读写被分配的所有粒子的位置域和速度域，然而被分配的粒子在同一粒子数组中是不连续的。假定每个域或性质对应一个 n（粒子数）元数组。一个双精度三维的位置（或速度）数据占 24 个字节，故多个这样的数据能对应一个高速缓存块。由于在数组中邻近的粒子可能被不同的处理器读和写，因而将引起伪共享。在这个阶段，用一个记录数组，数组中的每一项记录了粒子的所有信息，这样组织数据比按域组织数据要好。

现在我们考虑受力计算阶段。假定我们是用一个记录数组来组织数据，每个粒子的所有数据在其中的一个记录中。为了计算一个粒子的受力，一个处理器要读其他许多粒子的位置数据，然后更新被处理粒子的受力值。然而，一个粒子的受力值和位置值可能存在于同一缓存块中。在更新受力域时，可能会同时作废该粒子在其他处理器的缓存中的位置值（这个粒子的记录在其他处理器缓存中的存在是由伪共享所导致的，其他处理器要用到该粒子的位置数据，而位置数据在这个计算阶段是不被修改的）。在这种情况下，如果我们把以粒子记录为单元的数组分解为两个大小都为 n 的数组，一个以位置域（或其他性质域）为单元，一个以受力域为单元。可以如前所述填充受力数组，减少跨粒子的伪共享。总之，把一个记录数组分解为两个，一个数组的每项记录是在一个计算阶段中只读的记录域，另一个数组的每项记录是在同一阶段中要更新的记录域。不同的情况或计算阶段可能规定不同的数据组织，但最终的结果要根据对性能起支配作用的模式和计算阶段。

- 数组的对准。结合前述的技术，将数组对准到缓存块边界能得到更多的好处。例如，一个缓存块的大小为 64 字节，一个记录域为 8 字节，我们用一个记录数组存放粒子数据，数组的每一项是一个记录，该记录有 x、y、z、fx、fy、fz 6 个域。为了避免跨粒子的伪共享，我们为每一个记录填充两个 8 字节的虚记录域，使每个记录正好用一缓存块存放。但如果数组在虚拟地址空间中页面偏移 32 字节的地方起始，即使利用填充法也会导致伪共享，因为每个粒子的数据将跨存在两个缓存块中。即使 malloc 调用不能返回和页或缓存块对齐的数据，数据的对准也是容易办到的，只要在调用 malloc 函数时多要求一点额外存储，然后调整数组的起始地址即可。

364
~
365

通过以上的讨论，数据的组织、对准和填充等技术对于开发空间局部性，减少伪共享和冲突扑空都是很重要的。有经验的程序员，甚至一些编译器都使用这些方法。如第 3 章所讨论的，我们知道局部性和附加通信的问题比固有数据通信对性能更重要。这些问题可能使我们为了某个应用而重新考虑算法的划分决策（回顾在第 3 章 3.1.2 节中讨论的方程求解器问题，对比条状和块状两种划分，或参看图 5-35a）。

5.7 结论

对称共享存储的多处理器是工作站和个人电脑的自然扩展。一个串行应用程序可以照常在其上运行，同时享受到更大的处理器时间片、更大的共享主存和在这种机器中典型具备的 I/O 能力所带来的好处。运行并行应用也相对是容易的，因为对所有共享数据处理器可以通过普通的存取指令直接访问。在这样的机器上也容易从事渐进的并行化工作，即可以选择串行应用中的计算密集部分首先并行化，其效果当然要服从 Amdahl 定律。对多道程序的工作

负载，它优势的关键是对资源共享的细粒度，按这种粒度，系统的资源能在不同应用进程之间共享，并且由不同的操作系统共享，因此能很容易地为每个应用表现出一种熟悉的、单一系统映像。在时间上，处理器和/或主存页面时常被重新分配给不同的应用进程；在空间上，主存可以以页面为单位分配给不同的应用进程。因为这些吸引人的特点，所有大型的计算机系统厂商，从工作站供应商 Sun、Silicon Graphics、Hewlett-Packard、Digital、IBM 到个人电脑供应商 Intel，Compaq 都在生产和销售这样的机器。事实上，对于一些大的工作站供应商，这样的多处理器机器在它们的销售额和纯利中都占相当一部分，因为在这些高端机器上有更高的利润。

设计对称多处理器的关键技术难题是共享存储系统的组织和实现。这样的存储系统除了要满足常规的存储访问外，还要用来在处理器之间的通信。目前大多数小规模并行机用系统总线作为通信的互连，故问题转变为在处理器的私有缓存中保持共享数据的一致性。系统设计者有很多有效的选择，其中包括与缓存块相关联的状态集，所用到的总线事务处理和动作，缓存块大小的选择，使用更新还是作废策略。但设计者关键的任务是依据预期的数据共享模式，在工作负载的高效执行和使任务实现更加容易之间做出选择。另一个难题是高效的同步技术的设计和实现，要求即有高性能也有灵活性。 366

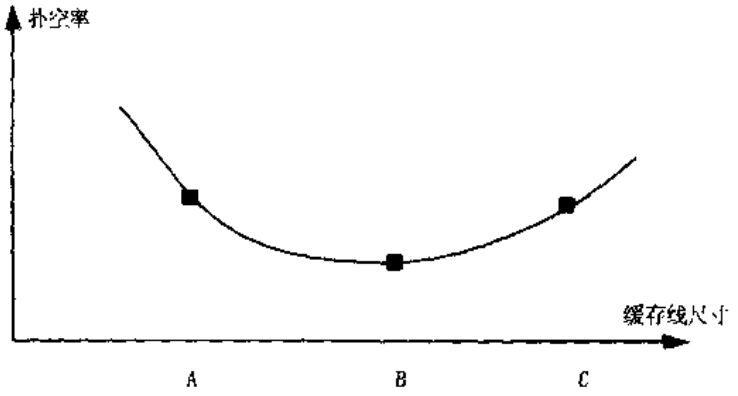
由于处理器、存储器、集成电路和封装技术的持续高速发展，人们关心小规模多处理器的未来和各种相关的设计问题。我们依据以下 3 个理由预料小规模多处理器将继续成为重要的发展方向。首先，它们提供了一个很吸引人的性能价格比。个人或小团体都能承受它们，把它们作为一个共享资源或一个计算或文件服务器。其次，当今微处理器的设计为多处理器作好了准备，在设计人员开始设计下一代多处理器的时候，他们是知道微处理器的未来趋势的，因此在新发布的微处理器和用其构成多处理器之间就不会有太大的延迟。如同在第 1 章所见，Intel Pentium Pro 微处理器就可以直接插在共享总线上。其三，一些必要软件支持正迅速成熟起来。例如，大多数计算机系统厂商都有支持它们自己基于总线的多处理器的并行操作系统版本。随着集成度的提高，在一块芯片上集成多个处理器将更加吸引人。尽管最佳的设计点会随技术和工艺的发展变化，但在本章中讨论的设计问题是基本的，无论技术如何发展它们将会保持其重要性。

本章在逻辑层研究了基于总线多处理器的设计的一些关键方面，包括缓存块状态的转换和完整的（原子的）总线事务。在这一层上，设计和实现表现在对传统缓存控制器的扩展。然而，很多在设计上的困难和很多优化与改进的机会出现在下一层协议设计和更具体的“物理”层中。下一章将讨论深入一层的设计，讨论基于总线缓存一致性多处理器的设计和体系结构，包括一些自然的推广。

习题

5.1 处理器寄存器是否也有和缓存一致性类似的问题？假定在硬件上不能保证寄存器的同一性，目前的系统怎样保证程序所期望的语义？

5.2 下图表示了在一个多处理器中一个应用程序的扑空率与缓存块大小的函数曲线。与预计的一样，曲线是 U 状。考虑在曲线上的 A、B、C 三点。指出在什么环境下，其中一点可能是机器的敏感操作点（也就是说，在这一点比其他两点机器有更好的性能）。对于单处理器，曲线形状和位置有怎样的变化？ 367



5.3 假设一个基于总线共享存储器的处理器的平均数据存储通信量：私有读——70%，私有写——20%，共享读——8%，共享写——2%。同时 50% 的指令（32 位）是存和取。有一个 32 KB 的数据/指令缓存，私有数据的命中率是 97%，共享数据是 95%，指令是 98.5%。缓存线为 16 字节。

我们希望在 64 条数据线和 32 条地址线的总线上放置尽可能多的处理器。一个处理器的时钟是总线时钟的两倍，不考虑存储器的影响，处理器的 CPI 是 2.0。如果我们采用 a）写分配策略的直写缓存，总线最大能支持多少处理器？b）如果是回写呢？忽略缓存一致性流量和总线争用。在回写缓存中因扑空取一新块来替换一脏块的概率是 0.3。对于读，存储器在收到地址后两个周期给出数据。对于写，地址和数据同时发给存储器。假定总线是原子的，处理器扑空损失正好等于每个扑空所需的总线周期数。

5.4 下面列出了三个存储器访问流，比较执行它们的开销，运行在基于总线机器上 a）Illinois MESI 协议，b）Dragon 协议。根据访问流的特点和相关协议解释执行的不同。

stream 1: *r1 w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3*
stream 2: *r1 r2 r3 w1 w2 w3 r1 r2 r3 w3 w1*
stream 3: *r1 r2 r3 r3 w1 w1 w1 w1 w2 w3*

368

所有访问都是针对同一个单元的：*r/w* 指明读或写，后面的数字表示发出操作的处理器。假定所有缓存初始为空，用下面的代价模型：读/写缓存命中要花费 1 个周期，扑空带来在总线上的事务（BusUpgr，BusUpd）要花费 60 个周期，扑空带来的传送整个缓存块要花费 90 个周期。假定所有缓存都是写分配的。

5.5 1）随着扑空时延的增加，更新协议和作废协议哪一个更可取？为什么？

2）在一个多级缓存层次结构中，你是将更新传播到第一级缓存还是仅仅只到第二级？说明折中方案。

3）为什么在目前基于更新的一致性在多处理器计算服务上对典型的多道程序工作负载不是一个好方法？

4）为了提供一个更新协议作为选择，一些机器在页中为软件提供了协议类型控制；也就是说，给定的页能保持一致地使用更新方式或作废方式。另一种不是基于页的控制方法是为将导致更新而不是作废的写提供特殊的操作码。评价优劣。

5.6 下面给出了一些代码段，在顺序同一性下哪些结果是可能的（或不可能的）。假定在代码到达前，所有变量初始化为 0。

1）

P_1	P_2	P_3
A = 1	u = A	v = B
	B = 1	w = A

2）

P_1	P_2	P_3	P_4
A = 1	u = A	B = 1	w = B
	v = B		x = A

3）在下面的序列中，用虚线框起的操作是同一指令的一部分：fetch&increment。然后假定它们是独立的指令。针对这两种情况分别回答上面的问题。

P_1	P_2
u = A	v = A
A = u + 1	A = v + 1

369

5.7　1）在 5.2.2 节中提到过一种由于写缓冲区的使用所导致的重新定序问题。在单处理器中并发程序环境下它会不会是一个问题？如果是，你如何去预防它？如果不是，为什么？

2）按照程序执行序，对于同一个存储单元，同一个处理器发出的读能否在前面的写之前完成（例如，如果写被放置在写缓冲区中但对其他处理器尚不可见）但仍然提供一个一致的存储系统？如果能，读返回值是什么？如果不能，为什么？这样做能否仍然保证顺序同一性？

3）如果我们只关心一致性，不关心顺序同一性（SC），我们能否说处理器通过了写操作写就完成了？

5.8　顺序同一性（SC）的充分条件是必要的吗？将那些约束放松：1）尽可能少约束，2）采用一种适当的中间方式。评价在实现复杂度上的效果。

5.9　考虑下列 SC 的充分条件：

- 每个进程按程序执行序发出存储请求。
- 一个读或写操作发出后，发出操作的进程将在发出下一操作前等待当前操作的完成。
- 在一个处理器 P_j 能够返回一个被另外处理器 P_i 写的值前，所有关于 P_i 的在其发出写操作之前完成的操作也必须关于 P_j 完成。

这些条件能否真正保证 SC 执行？如果能，为什么；如果不能，构造一个反例，说明为什么在本章中列出的条件是充分的。[提示：想想这些条件与本章中的那些条件有何不同]

5.10　考虑一个四个处理器基于总线的多处理器使用 Illinois MESI 协议。每个处理器执行一个 test&set 锁去获得访问一个空临界区。假定 test&set 指令总是发到总线上并且同处理一般读过程的时间开销一样。初始时处理器 1 拥有锁，处理器 2、3、4 在自己的缓存中踏步等待锁被释放。每个处理器一旦得到锁就退出程序。只考虑总线对上锁和解锁操作的处理：

1）从初态到终态所执行的最少总线事务是多少？

2）最多总线事务数是多少？

3）假设用 Dragon 协议，1）和 2）又如何？

5.11 在锁中指数式回退的主要优缺点是什么？考虑 test&set 锁，test-and-test&set 锁，票号锁，基于数组的锁。如果 LL-SC 被用来代替原子指令，情况会如何变化？

370

5.12 假设所有 16 个处理器同时争用一个 test-and-test&set 锁（每个处理器只有一次）。假定在 0 时刻所有处理器在自己的缓存中的锁上踏步等待并且因为一个释放而被作废。

1）如果所有临界区为空（也就是说，每个处理器在 LOCK 和 UNLOCK 之间什么都不做），到所有处理器获得锁将有多少次总线事务？

2）假设总线是公平的（即总是在服务新请求之前先服务挂起的请求），每个总线事务的开销是 50 个周期，到第一个处理器能获得和释放锁要多长时间？到最后一个处理器获得和释放要多长时间？

3）在一个不公平的总线上，如果想让你希望的处理器获得优先权而不管请求的顺序，最多能做到什么程度？

4）能否有一个不同于公平总线的总线仲裁方案用来提高性能？

5）如果用来实现锁的变量没有被缓存起来，一个 test-and-test&set 锁仍比一个 test&set 锁产生的通信量少吗？为什么？

5.13 对于同习题 5.12 的 2）一样配置的机器，假设是公平总线，使用一个加号锁，第一个和最后一个处理器需要多少个总线事务和多长时间来获得和释放锁？如果使用基于数组的锁，情况又如何？

5.14 考虑图 5-29 中使用指数型回退的 test&set 设置锁的性能曲线，为什么针对非空临界区的曲线比针对空临界区的曲线差？

5.15 1）在我们锁的实验中，为什么我们安排解锁后的延迟 d 后要比临界区延迟 c 小？如果 d 比 c 大将产生怎样的问题？[提示：画出两个处理器的执行时序图。]

2）如果我们用大得多的数值 c 和 d，比较锁算法会有怎样的结果？

5.16 1）分别用 i）fetch&increment 和 ii）LL-SC 来写出实现票号锁和基于数组锁的伪代码（高层表示加上汇编）。

2）假定你不用 fetch&increment 原语而用 fetch&store（一种简单的原子交换）。用这个原语能实现基于数组的锁吗？描述你得到的锁算法。

5.17 用 LL-SC 实现一个 compare&swap 操作。

5.18 考虑在 5.5.5 节中描述的具有感应交替功能的栅障算法，如果把 UNLOCK 语句放在计数器增值语句后，而不是在 if 条件的每个分支后，有问题吗？问题是什么？

371

5.19 假设有一个机器在每个机器字中都提供了满-空位硬件支持[○]。这类机器允许下列 C 函数：

`ST_Special (loc, val)` 把 `val` 写到数据单元 `loc` 并且置满空位。如果满空位已经被设置，则发出自陷信号。

`int LD_Special (loc)` 等待直到 `loc` 的满空位被设置，读数据，清满空位，返回结果数据。

○ 可以想像满-空位是一个特别的二进制位置满空位意指让其表示“满”，清满-空位意指让其表示“空”。——译者注

利用上述原语写一个 C 函数 swap（i，j），它能够原子性地交换 A［i］和 A［j］的内容。你应允许很高的并发性（如果多个处理器要交换一对不相干的存储单元，它们应能并发进行）并且必须要避免死锁。

5.20 原子操作 fetch&increment 能被用于实现栅障、信号灯和其他同步机制。fetch-and-add 的语义是将第二个参数加到第一个参数所指的存储单元并且返回该存储单元在加之前的值。用 fetch-and-add 原语去实现栅障操作适合于共享存储的多处理器。为了使用栅障，一个处理器必须执行 BARRIER（BAR，N），这里 BAR 是栅障的名字，N 是需要到达栅障的进程数。假定在 BAR 的使用中 N 都是同一数值。栅障将能够支持下面的代码：

```
while (condition) {
  Compute for a while
  BARRIER(BAR, N);
    }
```

下面是实现这种栅障的一种方法：

```
BARRIER(Var B: BarVariable, N: integer)
{
  if (fetch-and-add(B, 1) = N-1) then
    B := 0;
  else
    while (B != 0) do {};
}
```

这段代码中的问题是什么？写出避免了这些问题的 BARRIER 函数。

5.21 考虑下面同步原语 BARRIER 的实现，它用在应用中每个计算阶段的尾部。假设 bar.releasing 和 bar.count 的初始值为 0，bar.lock 初始时是开锁状态。

```
struct bar_struct {
  LOCKDEC(lock);
  int count, releasing;
} bar;
...
BARRIER(N)
{
  LOCK(bar.lock);
  bar.count++;

  if (bar.count == N) {
    bar.releasing = 1;
    bar.count--;
  } else {
    UNLOCK(bar.lock);
     while (! bar.releasing)
       ;
    LOCK(bar.lock);
    bar.count--;
    if (bar.count == 0) {
    bar.releasing = 0;
    }
```

372

```
        }
        UNLOCK(bar.lock);
    }
```

1）本代码不能提供一个正确的栅障。描述这个实现中的问题。

2）尽量少改动本例中的代码，使其能提供一个正确的栅障实现。指明改动的地方或仔细描述改动。

5.22 考虑具有移动性的数据：在处理器之间来回窜的共享数据，每个处理器读，然后在其他处理器读之前写。在标准 MESI 协议下，读扑空和写都能导致总线事务。

1）根据表 5-1 中列出的数据，估计用升级（BusUpgr）代替 BusRdX 后能省下的最大带宽。

2）有可能增强缓存块和状态转换图的状态，使紧接在同一块写操作后的读操作能被承认，使具有移动性的块能在第一次读扑空时直接把独享状态引进缓存中（而不是共享状态）。试给出附加状态的建议和状态转换表的扩充。根据表 5-1、5-2、5-3 中的数据，计算能得到的带宽节省。除了带宽节省以外还有其他好处吗？指出在程序中哪些地方具有移动性的协议可能有害于性能。

373

5.23 通过在更新时适当地更新主存，Firefly 更新协议删除了在 Dragon 协议中现有的 Sm 状态。我们能否通过合并状态 E 和 M，进一步减少 Dragon 和 Firefly 协议中的状态？有哪些折中的考虑？

5.24 人们发现处理器有时只写一个字到一缓存块中。为了优化这种情况，改变在所有情况下都用回写缓存的方式，就提出一个协议，它有下面的一些特点：1）在初始写一块时，处理器写直达到总线并且该块以一种新的状态——保留状态放置在缓存中；2）在写一个保留状态块时，缓存块转到已修改状态，用回写代替直写。

1）画出这种协议的状态变化，状态为 INVALID、SHARED、RESERVED、MODIFIED。图中要能显示对每一状态 BusRd、BusWr、ProcWr 和 ProcRd 的情况。指明处理器在斜杠（如 BusWr/WriteBlock）后的动作。由于采用整字和整块写，要指明刷新字（FlushWord）和刷新块（FlushBlock）。

2）如何区分这种协议与 4 状态的 Illinois 协议？

3）简明地描述你为什么认为这种协议没有用在像 SGI Challenge 一样的系统上。

5.25 考虑一个处理器写一个被很多处理器共享的块的情况（因而使它们的缓存作废）。如果这个缓存块在随后被其他处理器反复读，每次都将在该块上扑空。研究者提出了一个广播读的策略，一个处理器读该块时，所有其他的被作废的处理器将该块读到它们的二级高速缓存中。你认为这是一个好的协议扩展吗？至少给出两条理由说明你的选择，而且至少有一条是从反面说明。

5.26 在下面三个处理器的访问流中按图 5-20 中的类别把扑空分类（按照表 5-4 的格式）。假设每个处理器的缓存仅仅由一个 4 字的缓存块构成，字 w0 到 w3 在一个缓存块中，字 w4 到 w7 在另一缓存块中。

操作数	P_1	P_2	P_3
1	st w0		st w7
2	ld w6	ld w2	
3		ld w7	

（续）

操作数	P_1	P_2	P_3
4	ld w2	ld w0	
5		st w2	
6	ld w2		
7	st w2	ld w5	ld w5
8	st w5		
9		ld w3	ld w7
10		ld w6	ld w2
11		ld w2	st w7
12	ld w7		
13	ld w2		
14		ld w5	
15			ld w2

374

5.27 给你一个基于总线共享存储的机器。假设处理器有一个 32 字节的缓存块，A 是一个数组，其中的元素是 4 字节长的整数。考虑下面的循环：

```
for i ← 0 to 16
  for j ← 0 to 255 {
    A[j] ← do_something(A[j]);
```

1）在什么情况下最好使用动态调度的循环？

2）在什么情况下最好使用静态调度的循环？

3）对于动态调度的内循环，每一次一个处理器要迭代多少次？

5.28 如果你正在写一个图像处理程序，用一个二维像素数组表示图像。在计算中的基本迭代如下：

```
for i = 1 to 1024
  for j = 1 to 1024
    newA[i,j] = (A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j])/4;
```

假设 A 是一个以 4 字节单精度浮点数为元素的数组，并且按行存储（即 A [i, j] 和 A [i, j +1] 在存储中的地址是相邻的），A 的起始地址是 0。你正写的这个代码是支持 32 个处理器的。每个处理器有一个 32 KB 的直接映射缓存，缓存块的大小为 64 字节。

1）首先试着按交错分配法给每个处理器分配数组中的 32 行。你期望的计算和总线通信量的比率是多少（固有的或附加的）？假设每一次循环是 4 个计算单位，忽略所有其他的控制和赋值操作，声明你用到的其他假设。

2）如果分配给每个处理器的数组行是相邻的，回答和 1）同样的问题？

375

3）如果分配连续的数组列给每个处理器，回答和 1）同样的问题？

4）如果 A 的起始地址为 32，而不是 0，用 3）中的分配法，计算和所产生的通信量的比率有变化吗？如果有，是变大还是变小，为什么？如果没有，为什么？

5.29 下面是用一个 $O(N^2)$ 算法的简化的 n 体代码（即计算分子之间的相互作用），估计在稳定状态每一时间步的扑空数。再用本章中讨论的提高空间局部性和减少伪共享的方法改写代码。试着按你期望的处理器数和缓存块大小重构。假设 16 个处理器，1 MB 直接映射缓存，缓存块为 64 个字节。估算重构后代码的扑空数。声明你做的所有

假设。

```
typedef struct moltype {
  double x_pos, y_pos, z_pos;    /*position components*/
  double x_vel, y_vel, z_vel;    /*velocity components*/
  double x_f, y_f, z_f;          /*force components*/
  } molecule;

#define numMols 4096
#define numProcs 16
molecule mol[numMols]

main()
{
... declarations ...
  for (time=0; time < endTime; time++)
    for (i=myPID; i < numMols; i+=numProcs)
    {
      for (j=0; j < numMols; j++)
      {
        x_f[i] += x_fn(position of mols i & j);
        y_f[i] += y_fn(position of mols i & j);
        z_f[i] += z_fn(position of mols i & j);
      }
      barrier(numProcs);
      for (i=myPID; i < numMols; i += numProcs)
      {
        write velocity and position components
        of mol[i] based on force on mol[i];
      }
      barrier(numProcs);
    }
}
```

376

第6章　基于侦听的多处理器的设计

我们看到，市场上对称多处理器在性能、价格和规模上有很大的差异，这些差异主要不是由于高速缓存一致性协议的不同选择上，而是在于支持协议逻辑操作的组织结构的设计和实现上。人们对协议诸方面的权衡有了很好的理解；大多机器使用的都是上一章所描述协议的某种变形。然而，协议所导致的时延和带宽依赖于总线设计、高速缓存的设计以及和存储器系统的整合，还取决于系统工程的开销。本章考察基于侦听的一致性高速缓存的对称多处理器的详细物理设计问题。

虽然从第5章我们看到一致性协议对应的抽象状态转换图是相当的简单，但是在实现层会有一些很微妙的问题要解决。实现都必须争取达到至少三个相关的目标：正确性、高性能和最少的额外硬件。正确性问题产生的原因是抽象层认为原子性的活动到硬件层并不一定是原子性的。性能问题产生的主要原因是使存取操作流水化，允许同一时间有多个操作待完成（使用存储器的不同部件），而不是必须等待上一操作完成才能开始下一操作。不幸的是，正是由于这些事件关系错综复杂，使得正确性往往难以得到保证。因为一致性硬件中难以察觉的差错，一些商用系统，包括含有片上一致性控制器的微处理器的发布时间都大大推迟了。总体来说，缓存一致性多处理器中通信辅助部件（控制器）的设计提出了一系列挑战，其形式和复杂性和现代处理器的设计不相上下，表现在大量的待完成指令和乱序执行。我们需要深入一层来考察基于侦听多处理器的设计，理解状态转换图中所表达的实际需求。

本章首先列举缓存一致性存储系统主要的正确性要求。在6.2节，我们给出一个含有单级缓存和单事务原子总线的基础设计，简述在处理单个总线事务时的关键事件。在正确性讨论中这一节假定的是作废协议，但有关要点也可以直接应用于更新协议中。6.3节将这个设计扩展到多级高速缓存，展示了协议事件是如何在层次之间进行传播的。6.4节将这个基础设计扩展到使用事务拆分型总线的情形。在这样的总线中，一个总线事务被分为请求和响应两个阶段，这两个阶段都要涉及总线仲裁，因此多个事务在总线上可以同时处于待完成状态，能够以流水方式处理。然后，我们讨论多级高速缓存和事务拆分型结合的情形。从这一设计点来看，支持源于多个处理器的多个待完成扑空只是迈出了一小步，这是因为所有事务已经能够流水化处理，一些事务能够并发发生。在此当中潜在的本质挑战是要能保证由一致性和存储器同一性模型所要求的操作序的体现。随着设计复杂性的增加，如何做到这一点，也是在这些章节中讨论的。

一旦理解了一般情况下的关键设计要点，就能着手研究具体设计的细节。6.5节给出了两个案例：SGI Challenge和Sun Enterprise，用微基准测试程序和我们自己的例子应用程序来阐述它们的性能。最后，6.6节考察了若干将上述技术在功能和规模上扩展所涉及的进一步问题。

6.1　正确性需求

高速缓存一致性的存储系统理所应当要满足一致性的要求，并且要保持由存储器同一性

模型表达的语义。尤其是对一致性来说，必须要能够找到过时的副本，并且在写操作发生时使其作废或更新，还要提供写操作的串行化。如果要保持顺序同一性，那么应当提供写原子性和检测写操作完成的能力。另外，设计应该满足任何协议实现都要具有的性质，即要避免死锁和活锁，消除挨饿或使挨饿发生的可能性极小。最后，设计还要应付控制之外的错误情况（如奇偶校验错），并且尽量从错误中恢复。

死锁发生在操作仍然待完成但所有系统活动均已停止时。多个并发的实体争相获得共享资源，并以不可剥夺方式占有，产生资源依赖环，这样就导致潜在的死锁产生。图 6-1 所示交叉路口的交通是一简单类比。在此交通例子中，实体是车辆，资源是道路。每辆车需要两个道资源才能通过交叉路口，缺一不可，但每辆车都拥有一个道的资源，彼此互不相让。

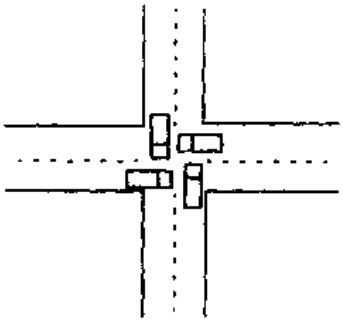


图 6-1　在十字路口的死锁。四辆车到达一个十字路口，每辆车占一条道。由于每辆车都占有着另一辆车得以前进所需的资源，它们相互阻塞。即使每辆车都让其右边的那一辆，该十字路口还是死锁。为消除这个死锁，某些车必须后退，让其他车前进，从而它自己才能前进

在计算机系统中，典型的实体是控制器，资源是缓冲区。例如，图 6-2a 所示，有两个控制器 A 和 B 通过缓冲区通信。A 的输入缓冲区已满，而且 A 拒绝接受任何来的请求信号，除非 B 接受了来自 A 的一个请求（从而释放 A 的缓冲区，A 才能接受来自其他控制器的请求信号）。但是 B 的输入缓冲区也满了，除非 A 能接受来自 B 的一个请求，否则拒绝任何来的请求信号。两个控制器都不能接受到请求，因此死锁形成。图 6-2b 所示的三个处理器例子可用于说明一般的多于两个控制器的情形。为防止死锁，根本上是要避免这种依赖环或者当其产生时将它消除。

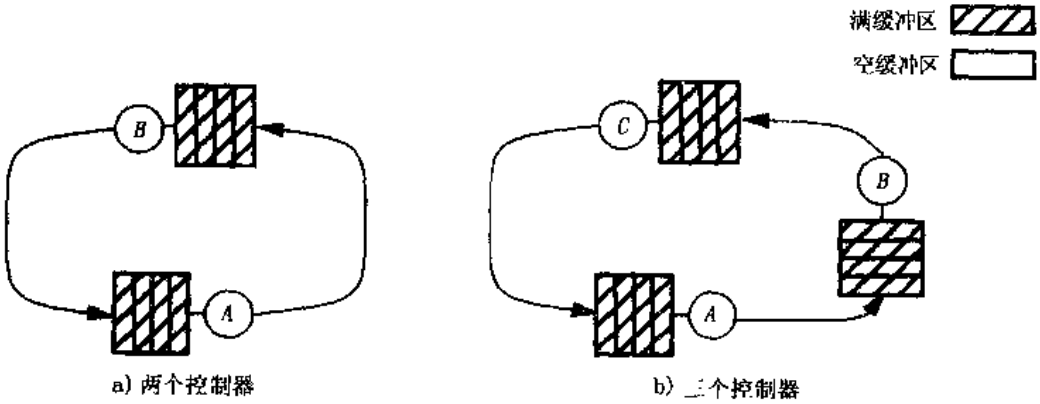


图 6-2　在计算机系统中的死锁。如果相互独立的控制器通过有限缓冲区相互通信的话，死锁很容易发生。如果在通信图上可能形成环路，那么每个控制器可能都会停滞，等待它前面的控制器释放资源

若尽管计算过程仍在系统中进行，但任何处理器的计算却不能得到进一步的推进，则称该系统处于活锁中。继续考虑上面的交通例子，任一车辆可能原路返回，以清空交叉路口，然后再试图前进。然而，如果车辆都同时重复前进后退，那将有大量的活动但最终不断重复

停止在同一位置，而无任何实质性的进展。在计算机系统中，活锁典型发生于多个独立控制器竞争某一共享资源时，当任何一方未完成当前操作对该资源的使用时，另一方就把资源剥夺过去。

378 ~ 379

挨饿现象不会停止全部的进展，但这是一种极度不公的现象，即一个或多个处理器毫无进展而其他处理器不断获得机会。例如，在交通例子中，活锁问题可以通过简易的优先级安排来解决。如果向北去的车辆比向东去的车辆优先级高，后者必须后退让前者在后者重试之前通过交叉路口。同样，南向的车辆可以比西向的车辆优先级高。[○] 不幸的是，这样却不能解决挨饿现象：在繁忙拥挤的交通中，东向车辆可能因为总有新的北向车辆准备通过而总不能通过路口。北向的车辆不断前进而东向的车辆处于挨饿状态。可能的解决办法是设置一仲裁者（警察或交通灯）公平的安排资源使用。这个比喻很容易扩展到计算机系统中。

一般而言，挨饿的可能性被认为比死锁和活锁危害要小。挨饿不会导致整个系统停止前进而且也不是永久状态。就是说过去一段时间内处于挨饿的处理器并不意味着将来所有时间内都会挨饿（在某些时候，北向的车将会清空了，东向的车辆就可以通过路口）。事实上，和这种无监控的交通例子相比，挨饿现象更不可能产生于计算机系统中，因为挨饿是和时间相关的，而必要的时序条件通常不会持久。在基于总线的系统中，挨饿现象易被消除，只要使用合理的总线仲裁设备和使用先进先出队列方式访问硬件资源就行了。然而，在后面的章节讨论可扩展系统的时候，会看到完全消除挨饿现象会大大增加协议的复杂度，降低常见情况下事务的推进速度。因此，虽然几乎所有的系统都在尽力减少挨饿现象发生的可能，但是许多系统并没有完全消除挨饿。

6.2 基础设计：采用原子总线的单级高速缓存

在第5章中，我们讨论了高速缓存一致性协议如何保证写串行化、如何满足顺序同一性的充分条件。我们假设总线是原子性的，给定进程的操作相对于其他进程也是原子的，而且产生总线事务的存储器操作，从发出到完成，也是原子的，即使来自不同的处理器。本节中，我们的假设会稍微实际些。每个处理器仍只有单级高速缓存，总线事务是原子不可分的。高速缓存在执行一存储操作所包含的一系列步骤时，它可以使处理器停滞；从而一个进程内的操作相对都是原子不可分的。除上述外，再没有其他假设了。本节讨论在这样的系统中实现侦听和状态转换所引起的基本问题和种种权衡，以及在提供写串行化、检测写完成和保证写原子性所引起的新问题。随后，我们将讨论更大胆的系统设计，包括先前讨论过的更复杂的高速缓存层次结构和更复杂的总线。不过，我们的讨论都是基于回写高速缓存的，至少最接近总线的缓存是如此，这样可以减少总线的流量。

380

即使是对简单的单级高速缓存和原子总线的例子，也必须做出一些设计上的选择。首先，给定处理器和总线端的侦听部件都需要访问高速缓存中的标记，我们应该如何设计标记和控制器呢？其次，从高速缓存控制器得到的侦听的结果需要作为总线事务的一部分体现出来，这又如何和何时办到？第三，即使总线有原子不可分性，满足处理器的存储器操作的若干动作还要使用其他资源（如高速缓存控制器），而这些活动并非原子不可分，从而引入可能的竞争条件。在这种非原子性条件下，我们如何来设计高速缓存控制器的协议状态机

○ 这里在方向上的描述和图6-1不符。如果以图6-1为准，应该是东向比北向的优先级高、西向比南向的优先级高。——译者注

呢？这样对写串行化、检测写完成和写原子性会产生什么新的问题呢？关于死锁、活锁和挨饿又会产生什么新问题呢？最后，从高速缓存中回写也能引入有意思的竞争条件，我们必须有支持原子不可分的读-改-写操作的机制。下面逐个来考虑这些问题。

6.2.1 高速缓存控制器和标记的设计

首先考虑传统单处理器的高速缓存，它由包含数据块、标记和状态位的存储阵列、比较器、控制器和总线接口部件组成。当处理器执行对高速缓存的操作时，地址的一部分用来访问可能含有相应存储块的一高速缓存组。标记和其他的地址位进行比较，确定寻址的存储块是否确实存在。然后进行适当的数据操作和修改状态位。例如，在一个干净的高速缓存块上的写命中，会导致一个字被更新并且将状态设置为已修改状态。高速缓存控制器按顺序进行它的存储阵列的读和写。如果操作需要将存储块从高速缓存传送到存储器（或者反之），则高速缓存控制器要发起一个总线操作。该总线操作要求总线接口部件执行一系列步骤，典型步骤如下：1）发出总线请求信号；2）等待总线的认可；3）驱动地址和命令；4）等待命令被相关设备接收；5）传送数据。高速缓存控制器所采取的一系列动作用有限状态机实现，一个总线事务中的若干步骤的实现也是如此。注意，不要将这里的状态机和缓存存储块所遵循的协议中的状态转移图相混淆。

为支持侦听一致性协议，必须对基本的单处理器高速缓存控制器的设计有所扩充。首先，高速缓存控制器除了响应处理器的操作外，还必须能监控总线上的操作。最简单方法是把高速缓存看作有两个控制器：总线端控制器和处理器端控制器，分别控制来自相应端的外部事件。任何情况下，当操作发生时，控制器都要访问高速缓存的标记。每个总线事务中，总线端控制器必定要捕获来自总线端的地址，并使用该地址进行标记对比。如果对比失败（一次侦听扑空），则不采取任何行动：即该总线操作和本高速缓存无关。如果侦听“命中”，则控制器可能会因为高速缓存一致性的要求卷入到该总线事务中。这可能涉及到对状态位的读-改-写操作或者是将一个存储块放到总线上（或者两者兼有）。

381

如果高速缓存只有一组标记，则很难允许这两个控制器对标记的同时访问。于是，在总线过程中，处理器将被锁在外面，不能访问高速缓存，这将降低处理器的性能。如果给处理器较高的优先级，于是侦听控制器必须在获得了标记访问权后才能进行总线过程，从而导致有效总线带宽的降低。为缓解这一问题，一致性高速缓存的设计对标记和状态可以使用双端口的随机存取存储器或者对每个块的标记和状态进行拷贝。高速缓存的数据部分不被复制，因为对它的访问不那么经常。如果采用双标记的方式，两套标记的内容是完全相同的，一套用于处理器端控制器查询，另一套用于总线端控制器侦听（如图 6-3 所示）。两个控制器可以同时读取标记和进行检验。当然，一旦某一存储块的标记或状态更改（如当状态改变或写入新的存储块时），最终两个版本都要修改，因此控制器之一可能被锁住一段时间。机器设计师们可以考虑用一些技巧来缩短控制器被锁住的时间。例如，上述例子中处理器端的标记不是在总线端标记更新时立即更新的，而是仅在后来高速缓存中数据被修改时更新。标记更新的频率也远远小于标记查询的频率，从而可以认为总线端标记更新和处理器高速缓存访问冲突较小。

另一个对单处理器中高速缓存的主要扩充是现在控制器不仅可以是总线事务的发起者，还可以作为总线事务的应答者。传统应答设备，如存储器模块控制器，它要监控总线，看一

个过程是否和某个特定的地址子集相关，且可能在若干“等待”周期后对相关读写操作做出应答。传统应答设备甚至可以把数据放到总线上。高速缓存控制器的做法与之类似，只是高速缓存控制器不对固定地址子集应答，而是对任何事务都要监控总线进行标记对比来决定是否相关。对基于更新的协议来说，控制器也可能侦听总线传出的新数据。多数现代微处理器已经实现此种增强型高速缓存控制器，因而用它们能很方便地构成多处理器。

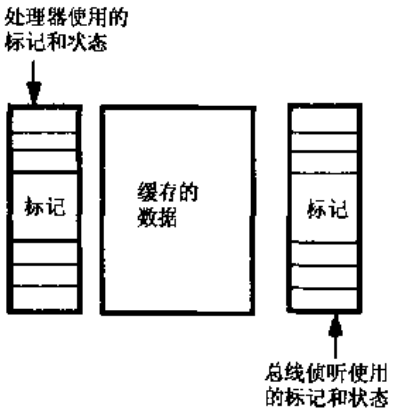


图 6-3 单级侦听缓存的组织。对于单级高速缓存，用双拷贝标记和状态组来减少冲突。处理器用一组，总线侦听器用另一组。不过，对缓存内容或状态的任何改变都要同时更新两组标记

6.2.2 侦听结果的报告

侦听也给总线事务引入了新的内容。在传统单处理器系统的总线事务中，某一设备（发起者）将地址放到总线上，所有其他的设备监控地址，其中有一设备（应答者）发现地址和它相关。然后数据在两设备之间传输。应答者发出“线或”信号来确认自己的作用；如果在一定时间后没有设备应答，则产生总线错误。对侦听高速缓存来说，每个高速缓存要用它的标记和地址对比，从各高速缓存侦听的总结果必须在总线事务继续之前让总线得知。特别地，这种侦听结果的一个功能就是通知主存是否要响应请求信号，或者某个高速缓存是否持有该存储块的已修改版本，从而必须采用另一行动。这里的问题是，什么时候将侦听结果报告到总线上，且以何种形式？

首先让我们着眼于“何时”这个问题。显而易见，令人满意的就是使延迟时间尽可能的少，使得主存能迅速决定如何行动[1]。下面是三个主要选项：

1）保证在一定的时间内产生侦听结果，通常指当地址出现在总线上后某个固定的时钟周期内。一般而言，这需要双套标记，因为处理器（一般具有优先级）在总线事务发生时会多次访问标记。即使有了双套标记，由于当处理器更新标记时两套标记都不能被访问，我们在完成侦听所需的时延上还需要采取一种保守的态度；例如在 MESI 协议由执行（E）向修改(M)状态转移时就是如此[2]。采取这种做法的优点在于主存设计不受影响，而且高速缓存

[1] 注意，在原子型总线上，我们有办法让系统对侦听时延不那么敏感。由于在任何时间只能有一个总线事务（原著中误为“存储事务”。——译者注）待完成，主存能够开始访问存储块，不管最后是它还是缓存要提供数据；否则主存子系统就可能空闲。然而，我们后面要讨论到，减小这个延迟对事务拆分型总线十分重要。在那种情况下，多个事务可能并发地在总线上待完成，于是存储子系统可能正在服务于另一个请求，为它（而不是高速缓存）提供数据。

[2] 有趣的是，在我们所描述的基本三态作废协议中，如果没有相应的总线事务的卷入，缓存块的状态是不会更新的。这通常会给标记的更新留有足够的时间。

到高速缓存之间的联系非常简单。而缺点是需要额外的硬件和可能较长的侦听时延。一种由4个 Pentium Pro 构成的多处理器就是用的这种方案，它能在必要时延长或推迟侦听阶段（见第8章），HP公司的商业服务器（Chan et al. 1993）和 Sun Enterprise 也是如此。

382 ~ 383

2）设计能够支持可变延迟的侦听方案。主存首先假设的是某一个高速缓存将提供数据，但提供的时机是在所有其他高速缓存控制器都完成侦听并且指出不提供数据以后。一定的握手协议是需要的，但高速缓存控制器不用担心标记访问冲突会影响及时的查询，且设计者也不用在侦听结果延迟时间的估计上采用保守的态度。SGI Challenge 多处理器使用的方法是这种做法的一种变形，即存储子系统先是针对请求取出数据，然后停滞，直到侦听完成后再根据情况相应动作（Galles and Williams 1993）。

3）第三种可能的做法是让主存子系统对每一存储块维护一个标记位，来指示该块是否被某一高速缓存修改。用这个方法，主存子系统不会被迫依靠侦听结果来决定下一步行动。这种方法的缺点是对主存子系统增加了额外的复杂性。

侦听的结果以何种格式报告到总线上呢？对 MESI 设计来说，请求的高速缓存控制器需要知道被请求的存储块是否在其他处理器的高速缓存中，这样才能决定以独享（E）还是共享（S）状态来装入该存储块。另外，存储系统也要知道是否有一高速缓存将该块置为修改状态：如果有，存储器则不用应答。一种合理的办法是使用三组线或信号，两个用来报告侦听结果的情况，另一个用作指示侦听结果是否有效。第一个信号指示某个处理器的高速缓存中（除请求处理器外）含有该存储块的副本。第二个信号指出该存储块在某个处理器的高速缓存中处于修改状态。我们不需要知道是哪个高速缓存，因为高速缓存本身知道该采取什么行动。第三个信号是禁止信号，它的出现表示还有高速缓存没有完成侦听。当第三个信号未出现时，请求者和存储器能够安全地检验其他两个信号。MESI 协议的完整的 Illinois 版本要更加复杂些，这是因为即使在共享状态下，存储块更优先地从其他高速缓存中取出而不是取自存储器。如果多个高速缓存均有副本，优先级机制决定从哪个高速缓存中提取数据。这也是为何大多数使用 MESI 协议的商业机型限制了高速缓存到高速缓存的传输。Silicon Graphics Challenge 和 Sun Enterprise 在数据在某一高速缓存中处于修改状态时，才使用高速缓存到高速缓存的传输，此时只有一个提供数据者。Challenge 在高速缓存到高速缓存传输中更新存储器，而 Enterprise 不更新存储器，并且使用第5章中讨论过的 MOESI 协议所含的第五个状态，即拥有状态。

6.2.3 对回写的处理

回写的实现比较复杂，因为回写过程涉及到一个要进入缓存的块和一个要被替换出的缓存块（已修改），因此包括两个总线事务。一般来说，为了使处理器在引起回写的缓存扑空发生后能尽可能快的继续工作，我们愿意推迟回写的处理，而先来处理引起回写的扑空。这一做法给我们提出了两个要求。其一，需要机器提供额外的存储空间一个（回写缓冲区）当新存储块写入高速缓存而总线还未能响应第二个事务时，被覆盖的存储块能暂时保存在回写缓冲区中。其二，在完成回写前，也许能发生总线事务，包含正在回写的存储块的地址。在这种情况下，控制器一定要从回写缓冲区中提取数据，且取消刚才未完成的回写总线请求。这就需要有地址比较器来监视该回写缓冲区。由第8章可知，在物理上分布存储的机器中，回写的正确实现有更进一步的问题。

384

6.2.4 基础系统组织

图6-4所示的是我们得到的基础侦听结构的框图。每个处理器有单级回写式高速缓存。高速缓存是双标记的，从而总线端控制器和处理器端控制器可并行的进行标记对比。处理器端控制器通过将地址和命令放到总线上来启动一个事务。在回写事务中，数据从回写缓冲区传输。在读事务中，数据被捕获在数据缓冲区内。总线端控制器侦听回写标记和高速缓存标记。总线仲裁器以一种全序安排总线上运行的请求信号。对于每个总线事务来说，请求阶段中的地址和命令以这种全序来驱动侦听查找。线或侦听结果用于向发起者确认所有的高速缓存都已看到请求并采取了相关行动。

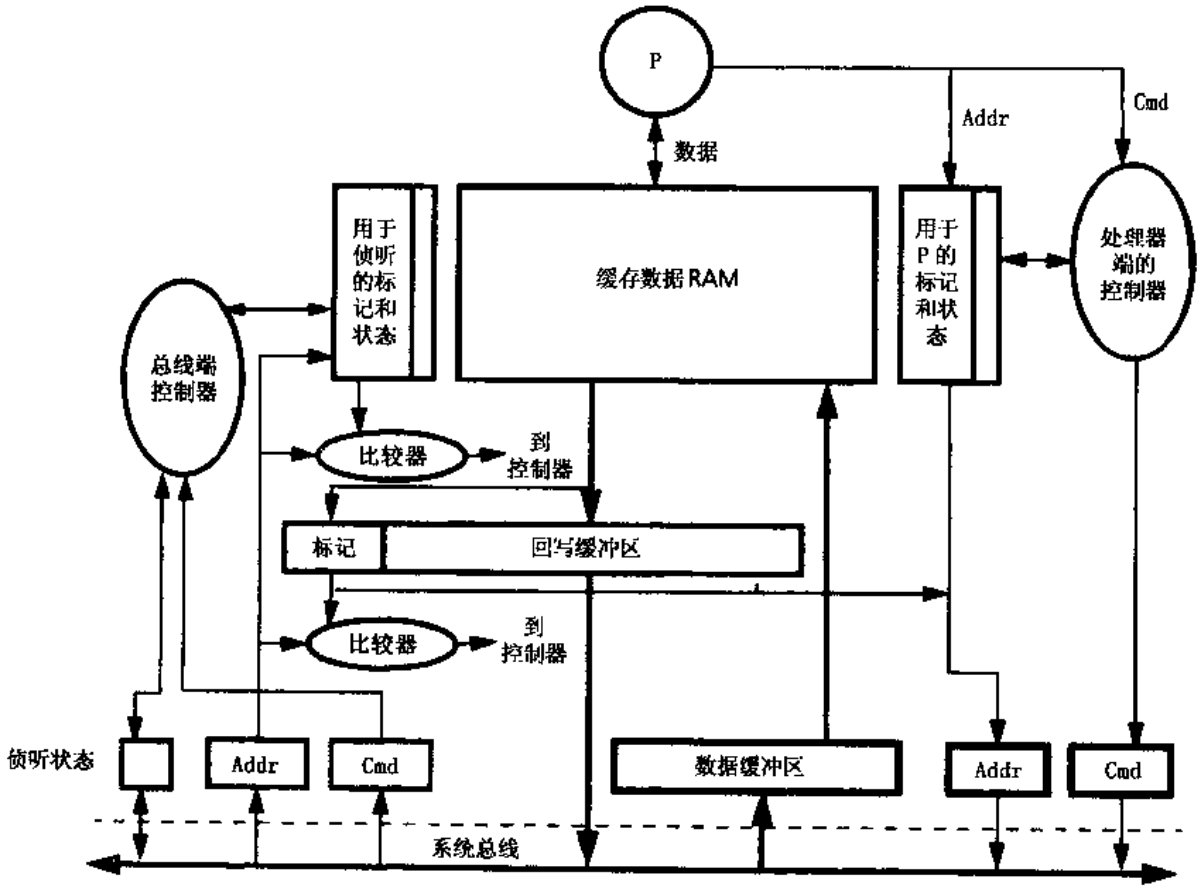


图6-4 基础机器的侦听高速缓存的设计。假设每个处理器有一个单级回写高速缓存，用一种作废协议，处理器只能有一个待完成的存储请求，系统总线是原子的。为保持简单，我们没有画出总线仲裁逻辑和所需的某些低级信号和缓冲区。也没有画出在总线方控制器和处理器方控制器之间所需的协调信号

用这种简单设计，让我们来考虑进一步的正确性问题，这些问题需要对状态机和协议进行扩充，或者需要在实现中有些特别的措施。它们包括非原子性的状态转移、为达到一致性和同一性的序列化、死锁、活锁和挨饿现象。

6.2.5 非原子性的状态转移

在第5章的状态转移图中，状态转移和相关活动都被假设为同时发生或至少是原子不可分的。事实上，由处理器产生的请求要花一段时间才能完成，常常包括一个总线事务。虽然

在我们的简单系统中，一个总线事务是原子不可分的，但这仅仅是要满足处理器请求一系列活动的其中之一。这一系列活动包括查询高速缓存的标记、总线仲裁、由其他控制器对相应高速缓存采取行动以及由请求处理器的控制器在总线事务结束时采取的行动（可能包括实际地将数据写入存储块中）。总的来说，这一系列活动不是原子不可分的，即使是使用原子总线，来自不同处理器的多个请求可能同时在系统的各部分活动，或许当处理器（控制器）P有一请求等待完成（如等待获得总线访问权）而来自另一处理器的请求出现在总线上，并且需要由P提供服务，甚至可能是和P请求待完成一样的存储块。这类复杂问题将在例6.1中加以说明。

例6.1 假定处理器P_1和P_2在它们的缓存中共享存储块A，并且同时对A发出一个写操作。说明当P_2的事务出现在总线上时，P_1的待完成请求是如何等待总线的，解决这种复杂情况可能有些什么办法。

解答： 这里是一种可能的情形。P_1的写操作会检查它的高速缓存，确定在实际向块中写数据之前，它需要将存储块的状态从共享提升到已修改，并且发出一个升级总线请求。同时，P_2已经发出了一个类似的升级或者排他读A的过程，它可能首先通过仲裁得到总线。P_1的控制器会看到总线事务，必须将A的状态从共享降级到无效。否则，当P_2的事务完成后，A在P_2的高速缓存中为已修改状态，在P_1的高速缓存中为共享状态，这是违反协议规定的。但现在使P_1待完成的升级总线请求不再是合适的，必须用排他读请求替换。这样，控制器也必须要能够将它自己待完成的请求的地址和从总线上侦听得到的地址对比，并且在必要的时候修改前者。（如果在协议中没有升级过程，并且即使在对共享状态的存储块做写操作时也用排他读，那么在这种情况下即便存储块的状态改变了，请求也没必要改变。因此，当评价协议优化的复杂性时，我们应该考虑这些实现方面的需求。）■

385 ~ 386

要处理状态转移中非原子性现象，而且有时需要根据所观察到的事件来修改请求和行动，一种方便的方法就是使用中间或过渡状态扩展协议状态图（原协议状态我们已深入讨论过，如MESI，其状态都认为是稳态）。例如，可用单独的一个状态来指示升级请求正待完成。图6-5给出了MESI协议的一种状态扩展图。为应答处理器的写操作，高速缓存控制器通过确认总线请求和转移到中间的S→M状态来开始总线仲裁。当总线仲裁器确认了给该设备的总线授予（BusGrant）信号，则从中间状态转移出去。此时，总线升级事务（BusUpgr）放到总线上，高速缓存中相应块的状态得到更新。然而，当处于S→M状态下，如果总线上观察到对该存储块的总线读请求（BusRdx）或者总线升级请求（BusUpgr），那么控制器将视该存储块在此过程和转移I→M状态之前已被设置为无效。（我们可以撤回总线请求，转移到I状态，让挂起的PrWr再度得到处理。）在处理器从无效状态读数据时，控制器转到中间状态（I→S，E）；而下一个要转移到的稳态由读操作得到总线后共享信号线的值决定。这些中间状态通常在高速缓存块的状态位中未加以编码，所编码的仍然是稳定的MESI状态。这是因为要表示可能出现处于暂时状态的高速缓存块需在每个高速缓存块中扩充，实在过于浪费。这些状态可以通过状态位及控制器的状态共同反映出。可是当我们考虑高速缓存允许多个待完成过程时，就必须对可能处于过渡状态的高速缓存（多个）块加以明确表示。

协议中的状态数的扩充加大了证明实现过程正确性和测试设计的难度，因此设计者也寻求避免过渡状态的机制。例如Sun Enterprise没有使用MESI协议中的BusUpgr事务，而是使用BusRdx事务中侦听结果来消除冗余的数据传输。回顾在总线写操作中，包含该存储块的

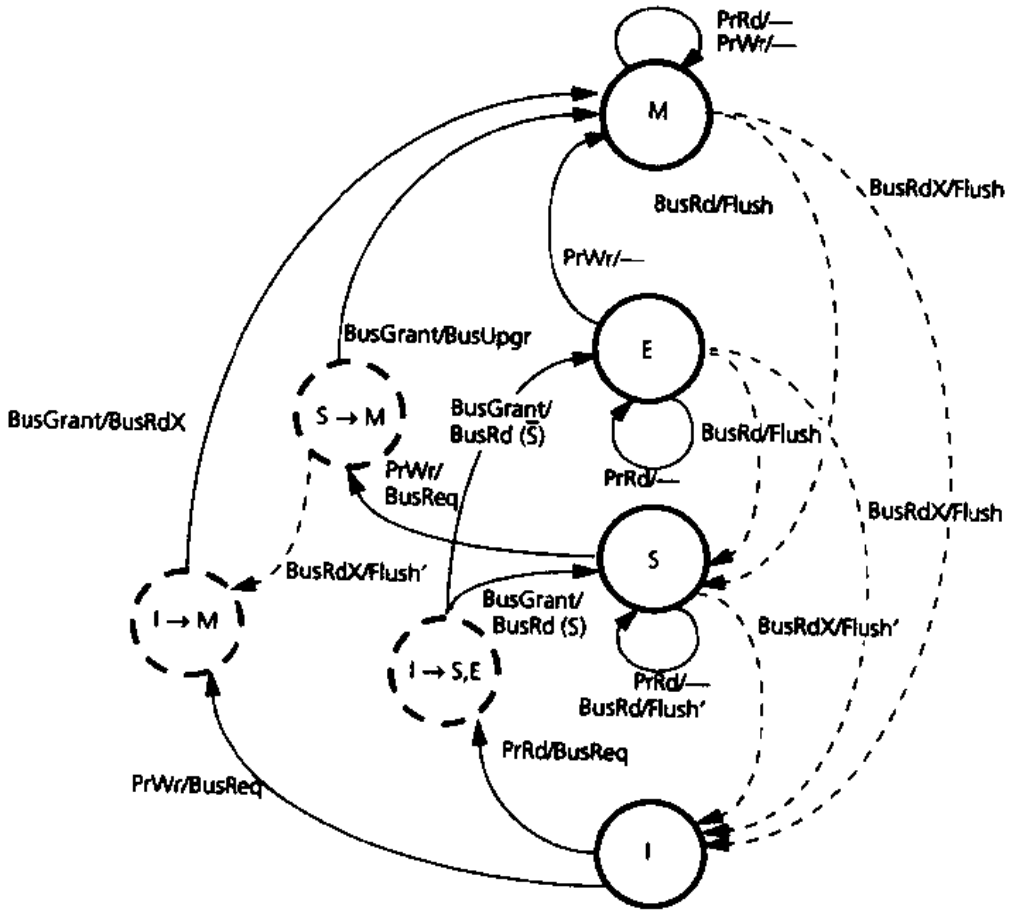


图 6-5 扩充的 MESI 协议状态图，指出了总线获取中的过渡状态。缓存控制器在其请求进行仲裁时监测总线。一个有冲突的事务可能改变稳态之间的转换

高速缓存要将副本置为无效。如果某一高速缓存包含修改状态下的存储块，它就会给出脏信号线，从而阻止存储器提供数据，它自己将数据送到总线上。这里没有用到共享信号线。当总线事务在总线实际进行时，此方法将使发出总线读请求的处理器侦听自身的标记。如果高速缓存中该存储块还处在有效状态下，它就给出共享信号来禁止主存。因为它已经含有有效存储块，该块在任何其他高速缓存中都不可能处于已修改状态，且事务中的数据阶段被忽略。高速缓存控制器不需要过渡状态，因为不管发生什么，它只要做一件事——将 BusRdX 事务放到总线上。

6.2.6 串行化

鉴于不同处理器发出的存储操作的非原子性，在处理器和高速缓存的握手机制当中一定要注意保持由总线事务的串行化所决定的顺序。对于读操作，处理器需要操作结果。为获得写操作上更高的性能，一种有吸引力的做法是当高速缓存控制器获取存储块的独享权时，让处理器在缓存块更新的时候继续执行一些有用的指令，还可能通过一个总线事务装入存储块的剩余部分。这里的问题是，在处理器向高速缓存发出写操作和高速缓存控制器获得排他读（或者升级）事务总线所有权之间有一个空当。正如我们所知，其他总线事务（包括写操作）可能出现在这一空档中，这就可能改变高速缓存中该存储块或其他块的状态。这将使一致性

387
388

写串行化（如果这个事务针对同一存储块）和 SC（如果事务针对不同块）复杂起来。为提供写操作串行化或 SC，这些过程都一定要让处理器看起来是出现在写操作之前，因为这就是它们通过总线得到串行化且告知其他处理器的方式。为保守起见，高速缓存控制器应该不允许发出写操作的处理器，在排他读事务出现在总线上且使得写操作对其他处理器可见之前，认为写已经完成了从而可以去执行程序中在写操作之后的其他操作。

事实上，高速缓存不用等到排他读事务完成之后，即等到其他高速缓存中的其他副本实际被作废后，才允许处理器继续工作：事务一旦出现在总线上，只要传输中存储块的访问能处理合理，处理器就可以为读、写命中服务。5.2 节给出的一致性和顺序同一性的关键论点，在于所有的高速缓存控制器以同样的顺序观察由写操作产生的独享所有权事务（BusRdx 或 BusUpgr），以及独享所有权事务完成后立即将数据写入高速缓存中。一旦总线事务开始，在我们的基本设计中写者认为所有其他的高速缓存将在其他事务发生之前将其副本作废。我们称写操作*被提交了*，指的是在总线顺序中该写操作的位置已完全确定，与进一步的活动无关。写者不可能确切知道在其他处理器的局部程序顺序中何处会被插入作废操作；它只知道不论操作产生下一步的总线事务如何，插入作废都在此之前且所有处理器以相同的顺序插入作废。同样，写者在其后的本地高速缓存命中的顺序也仅仅在下一个总线事务中才可见。这些为保证一致性和 SC 必须维护的序关系是很重要的，它允许写者能在达到 SC 的充分条件下用提交代替实际完成。事实上，这个基本的观察结论就是有可能用流水的方式实现高速缓存的一致性和顺序同一性的关键；这里的流水包括总线、多层存储器和每处理器的多个待完成操作。写原子性和先前 5.3 节阐述相同。

操作串行化的讨论产生一个重要的但有点难以说清的观点。写串行化和写原子性对于何时把数据写回到存储器，或者何时存储器单元被更新没有任何影响。任何读或写操作如果产生要被覆盖的脏块，就会导致一次回写。回写也是总线事务，但它们不需要按照顺序进行。另一方面，一次写操作并不一定导致新的值出现在总线上，即使是扑空了：它会产生一次排他读。对程序来说重要的是新值绑定到地址上的时间。写操作完成，就是说一旦总线读事务（BusRdx）或者总线升级事务（BusUpgr）发生，就会返回刚才写进去或者在其后写进去的值。通过作废旧的高速缓存块，保证了所有返回旧值的读操作在这个事务之前发生。发起这个事务的控制器能保证新的值在总线事务后写入高速缓存且无任何其他的存取操作干扰。

389

6.2.7　死锁

一种如同存储操作中请求-应答那样的两阶段协议表现出了协议层死锁的一种形式，有时称作“取死锁”（Leiserson et al. 1996），它不仅仅是缓冲区使用的问题。当一实体试图发出请求信号，它需要为接踵而来的事务服务。在带原子总线的 SMP 中，当高速缓存控制器等待总线时将产生下列情况：高速缓存控制器既要继续侦听、又要处理请求信号，而请求信号可能会要求控制器把存储块传输到总线上。如果两个控制器各含有一待完成事务，而这两个事务都要求对方应答却同时拒绝处理请求信号，这样系统就可能死锁。例如，假设总线上出现对 *B* 块的 BusRd，同时某个处理器 P 对另外一块 *A* 的排他读的请求信号被总线接受。若 P_i 有 *B* 块的已修改副本，等待它控制器就应当在等待获得总线时给当前总线事务提供数据（并不需要原子总线的仲裁），且改变修改状态为共享状态。否则，当前总线事务就会等待 P_i 控制器而 P_i 控制器又在等待总线事务释放总线。

6.2.8 活锁和挨饿

在基于作废协议的高速缓存一致性存储系统中，典型潜在的活锁问题是由所有处理器试图同时写入同一存储单元而产生的。设想一下，开始任何处理器在其高速缓存中均无该单元的副本。某一处理器的写操作有下列非原子不可分的系列活动：它的高速缓存获得对应存储块的独享所有权（即它将其他副本作废，获得修改状态的存储块）；处理器中状态机确认高速缓存中该块处于合理状态下；状态机再进行写操作。除非对处理器和高速缓存握手的设计非常严密，不然很有可能出现存储块以修改状态进入了高速缓存，但处理器还没有完成写，该存储块又被其他的处理器的 BusRdX 请求作废了。处理器写操作再次失败，如此可能总重复下去。为避免活锁，一个已获得独享所有权的写操作一定要在所有权被拿走之前能够完成。

当处理器争用总线时，有可能一些处理器反复获得总线而另一些却由于不能获得总线而挨饿。挨饿现象可通过在总线仲裁和其他方面采用先来先服务的原则避免。然而，这往往要另加缓冲设备，因此有时就采用启发式技术来减少可能发生的挨饿现象。例如，记录下该请求被拒绝的次数，一旦超过某个阈值，则拒绝其他请求；直到该请求服务完成，才服务其他新请求，或者可以提高该请求的优先级。 390

6.2.9 原子操作的实现

在讨论更实际的体系结构之前，我们对基本体系结构还应了解的最后一项是原子不可分的读－改－写指令的实现，诸如 test&set 和 fetch&op 以及能合成原子操作的 LL-SC[⊖] 原语(参看第 5.5 节)。

考虑简单的 test&set 指令，该指令包含读成分（test）和写成分（set）。第一个问题就是 test&set（加锁）指令的变量是否可放入高速缓存中从而可以在处理器的高速缓存中执行；如果不能放入，则原子操作在主存里执行。5.5 节中有关同步的讨论假设了锁变量可以缓存。其优点是允许对局部性的利用，从而当同一个处理器反复需要该锁时降低时延和流量；锁变量以已修改状态保留在高速缓存中，不产生作废和扑空。在锁的状态不满足要求时，处理器还可以在其高速缓存中踏步等待，从而减少无用的总线流量。然而，在存储器上执行锁操作能加快锁从一处理器传到另一处理器。对于可高速缓存的锁，忙等待下的处理器首先被置为无效，然后试着从其他处理器的高速缓存或者主存来访问锁。对于不能高速缓存的锁，锁的释放直接反映到存储器（不需要作废什么），且在作废操作到达存储器之前，正在等待的处理器的下一个读操作可能也正在抵达存储器的路上，因此可以以很低的时延从存储器获得锁。总的看来，流量和局部性的因素占支配地位，因此锁变量多为可高速缓存的，从而处理器在忙等待时可不必访问总线。

如果可缓存的 test&set 命令的实现不能由高速缓存本身完成，一种自然的方法是使用两个总线事务：读总线事务处理 test 部分和写总线事务处理 set 部分。保证这个序列原子不可分的办法之一是在读事务时锁住总线直到写过程完成，使其他处理器不能在两个事务之间访问总线。对于原子总线来说，实现这一点非常容易，但对于事务拆分型总线则有较大难度：锁住总线不仅会有损效率，而且若某一事务不放弃总线就不能立即满足的话，则可能导致死锁。

幸运的是我们有更佳的方法。研究一个回写式高速缓存的基于作废的协议。处理器真正

⊖ 即装入链接－条件存储。——译者注

要做的是获得高速缓存块的所有权（如通过发出一次排他读总线事务），然后执行高速缓存中读成分和写成分，只要是两者之间不放弃块所有权就可以；即使算是非原子的总线，新来的从总线到那一存储块的访问也被禁止且一直等到数据写入高速缓存为止。更加复杂的原子操作，如 fetch&op 也都要保留独占所有权直至操作结束。

[391]

Compare&swap 是一更难实现的原子指令。它要求在一存取指令中指定三个操作数：存储单元、用于比较的寄存器以及要与存储单元交换的数值/寄存器。RISC 指令系统一般不包括该指令。

实现 LL-SC 需要些特殊的支持。典型的实现方法是在每个处理器中使用硬件锁标记和锁存地址寄存器。LL 操作读存储块的同时也设置锁标记和将块地址放入锁存地址寄存器。新的来自总线的作废（或更新）请求信号和锁存地址匹配，匹配成功（称冲突写）则复位锁标记。条件存储检查锁标记，判断是否发生冲突写；如果标记已经复位，则检查失败，否则则检查成功。如果锁变量从高速缓存被替换，锁标记同样被复位（条件存储失败）。这是因为处理器无法看到对该变量的作废或更新操作。最后，锁标记还可能在上下文切换的时候复位，因为在 LL 和它的条件存储之间的上下文切换可能错误地使得老进程的 LL 导致切换进来的新进程的条件存储执行的成功。

在实现 LL-SC 中为避免活锁问题又产生了新的细节问题：第一，实际上我们不应允许占有锁变量的高速缓存块的替换发生在 LL 和 SC 之间。替换会清除锁标记且形成处理器不断试图执行 SC 但总不成功的情况。由于在 LL 和 SC 操作之间可能有不断的替换发生。为了禁止和取指相冲突的替换，我们可以使用拆分型指令和数据高速缓存或者组相联的统一高速缓存。对于和其他数据引用的冲突，常见的解决办法是简单在 LL 和条件存储之间禁止涉及存储器的指令。隐藏时延（如乱序问题）的技术可能将问题复杂化，因为程序代码不在 LL 和 SC 之间的存储操作到执行时，则可能在两者之间。简单的解决办法就是不允许重排存储操作跨 LL 或 SC 操作发生。

第二种活锁的潜在情况发生在两个进程连续在条件存储上失败，而且每个进程失败的条件存储使其他进程的存储块作废或者更新，从而清除了锁标记。如果这种错误情况继续存在，那两个进程都不能成功。这也就是为什么不把条件存储视为简单的写操作和失败时也不能发出作废或更新命令的重要原因。

和实现原子的读-改-写指令相比，LL-SC 在效率上有问题，这是因为 LL 和 SC 即便成功，但都可能产生缓存扑空。当 LL 装入处于共享状态下的存储块时，就会发生这种情况，导致两个扑空，而不是一个。为提高性能，我们可能希望在 LL 执行时以独占或修改状态获得（或预取）存储块，从而条件存储除非失败否则会扑空。然而，这又产生了第二种活锁情况：将其他副本作废来获得独享所有权，因此如果不能保证这处理器条件存储成功，那其他处理器的条件存储也将失败。如果引入这种优化，则应该在失败的操作之间引入某种形式的回退，以减少（尽管不能完全消除）活锁的可能性。

[392]

6.3 多级高速缓存层次结构

上一节给出的简单设计是可以说明一定问题的，但它有两个简化的假设并不适用于大多数现代系统：即单级高速缓存和原子总线。这一节将放弃第一个假设，来研究所导致的设计问题。

自 20 世纪 90 年代早期开始，微处理器的设计趋势一直是两级高速缓存，第一级在片

内，第二级高级缓存容量要大些，可能在片内也可能在片外[⊖]。不少系统也使用片内第二级高速缓存和片外第三级高速缓存。多级高速缓存的层次结构看来使一致性问题更加复杂，因为由处理器对第一级高速缓存做的修改，可能对负责总线操作的更低级的高速缓存不可见，且第一级高速缓存也不能直接看见总线事务。然而，为达到高速缓存一致性的基本机制可以自然地扩展到多层高速缓存上。让我们看看图 6-6 所示的两级高速缓存具体的层次结构；到多级情形的扩展就显而易见了。

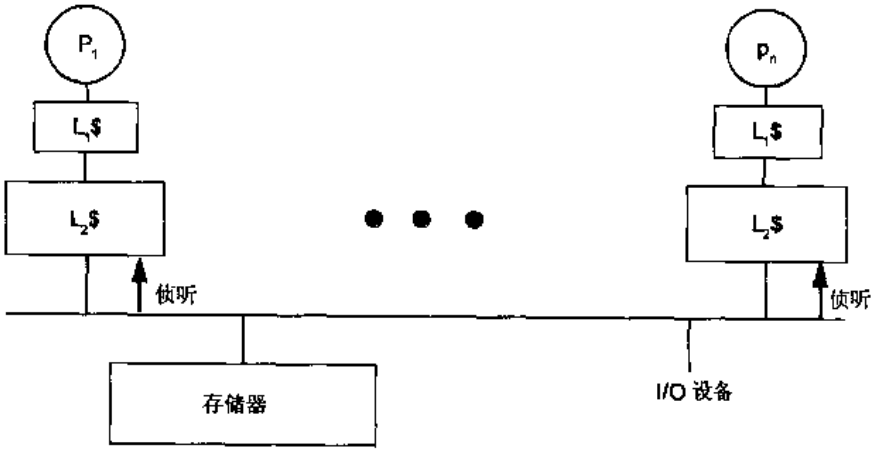


图 6-6 一种基于总线、包含两级高速缓存的处理器的机器

处理多级高速缓存的一种显然的方法是高速缓存层次结构中每级都有相应独立的总线侦听硬件。由于种种原因，此方法不太理想。首先，L_1 高速缓存总是在处理器芯片内，片内侦听器要占用珍贵的芯片管脚来监听共享总线上的地址。第二，做双份标记以允许处理器和侦听器同时访问，也会消耗过多的芯片资源。第三，由于大多时候出现在 L_1 高速缓存的存储块也出现在 L_2 高速缓存上，因此 L_1 和 L_2 侦听有重复操作；因此，L_1 高速缓存的侦听是不必要的。

实用的解决办法就是基于对上面最后一条的认识。当使用多级高速缓存时，设计者要确保所谓包含特性，要求如下：

1）如果一存储块在 L_1 高速缓存内，那它也必在 L_2 高速缓存内。换句话说，L_1 高速缓存的内容是 L_2 高速缓存的子集。

2）如果该存储块在 L_1 高速缓存中是拥有状态（如 MESI 或 MOESI 中的修改状态，Dragon 中的共享修改状态和 MOESI 中的拥有状态），那么它在 L_2 高速缓存一定是已修改状态。 393

第一条要求保证任何与 L_1 高速缓存相关的总线事务必与 L_2 高速缓存相关，因此让 L_2 高速缓存控制器侦听总线就足够了。第二条保证如果一个总线事务向 L_1 高速缓存或 L_2 高速缓存请求某个已修改状态的存储块，L_2 高速缓存能知道这个存储块就在它那里。

6.3.1 包含性的维护

包含性的维护并非小事。有三个方面要考虑到。第一，与 L_2 高速缓存有关的处理器引用

⊖ HP 的 PA-RISC 微处理器是一个特例，在许多其他厂家追求较小的片上一级缓存后，它在多年里一直用一个大容量的片外一级缓存。

能使它改变状态和执行替换，这就需要以保证包含性的方式来处理。第二，总线过程引起 L_2 高速缓存改变状态和输送存储区块，它们要反映到第一级。最后，已修改状态必须要传播到 L_2 高速缓存。

乍一看，由于所有 L_1 高速缓存扑空都会到 L_2 高速缓存中去找，似乎包含性能自动满足。然而问题是在扑空时，两个高速缓存可以选择不同的存储块和数据来替换。只是在某种高速缓存的组合配置下，包含性才能自动维持。研究一下如果不采取特殊措施，在何种情况下典型的高速缓存层次结构会导致包含性被破坏，是一个很有趣的练习（Baer and Wang 1988）。在知道如何保持包含性之前，先研究上面这个问题。为表达方便，假设 L_1 高速缓存的相联度是 a_1，组数是 n_1，存储块大小是 b_1，总容量 $S_1 = a_1 \times b_1 \times n_1$。相应 L_2 高速缓存的参数是 a_2，n_2，b_2 及 s_2。我们还假设参数均是 2 的幂次。

- *基于历史的替换策略，组相联 L_1 高速缓存。*基于存储块访问的历史纪录的替换算法，如 LRU（最近最少使用），其问题在于 L_1 高速缓存看到的访问历史和 L_2 高速缓存及其他级看到的不同，因为处理器的所有存储访问都要察看 L_1，但并不一定都到更低层的高速缓存。假设 L_1 是两路组相联，采用 LRU 替换策略，L_1 和 L_2 高速缓存大小相同（$b_1 = b_2$），L_2 是第一级的 k 倍（$n_2 = k \times n_1$）。容易看出在此简单例子中不能保持包含性。假设三个不同的存储块 m_1，m_2 和 m_3，映像在第一级的同一组内；若 m_1 和 m_2 都处在 L_1 内该组的两个有效位置上，且出现在 L_2 高速缓存中。现在看看当处理器引用 m_3，在与 m_1 和 m_2 冲突时，会发生什么事情。它要导致替换 m_1 和 m_2 之一，这不仅涉及 L_1 高速缓存，而且也涉及 L_2 高速缓存。因为 L_2 高速缓存对 L_1 高速缓存的访问史一无所知，而这个访问史决定了 L_1 高速缓存是替换 m_1 还是 m_2；因此容易发现有可能 L_2 高速缓存替换了 m_1 和 m_2 之一，而 L_1 高速缓存替换的却是另外一块。如果 L_2 是直接映像方式，甚至是双路组相联，且 m_1 和 m_2 在同一组，不同的替换也会发生。事实上，将该例子的情形推广可见，如果 L_1 不是直接映像方式，但采用 LRU 替换策略，不管 L_2 的相联度、存储块的大小还是整个 L_2 的大小如何，包含性都可能被破坏。
- *在一个层次有多个高速缓存。*当第一级高速缓存将数据和指令分开，在发生替换时可能出现类似的问题，即便是采用直接映像方式，且被一体化的第二级缓存支持。首先假设 L_2 高速缓存也是直接映像，在 L_2 高速缓存发生冲突的指令块 m_1 和数据块 m_2 因进入不同的高速缓存中而在 L_1 不发生冲突。如果在引用 m_1 时 m_2 在 L_2 高速缓存中，m_2 将在 L_2 高速缓存中被替换，但在 L_1 数据高速缓存中保留，这就违反了包含性。这就说明，如果在一级有多个独立的高速缓存，即使下面的高速缓存是一体化的，且有很高的相联度，包含性仍无法得到保证（见习题 6.72）。
- *不同的存储块大小。*最后，不同的存储块大小也会破坏包含性。考虑某种使用直接映像方式的系统，一体化的 L_1 和 L_2 高速缓存（$a_1 = a_2 = 1$），存储块大小分别为一个字和两个字（$b_1 = 1$，$b_2 = 2$），组数分别为 4 和 8（$n_1 = 4$，$n_2 = 8$）。因此，L_1 的大小是 4 个字，存储字单元 0，4，8，……映射到组 0。单元 1，5，9，……映射到组 1，依次类推。L_2 的大小是 16 个字，字单元 0 和 1，16 和 17，32 和 33，……映射到

[394]

组 0；字单元 2 和 3，18 和 19，34 和 35，……映射到组 1，依次类推。现在易看到 L_1 高速缓存可以同时包含单元 0 和单元 17 的字（它们分别映射到组 0 和组 1），而 L_2 高速缓存因为两字映射到同一组（第 0 组）却不是相继的字（于是存储块大小为两个字无济于事），所以不能同时包含它们。正如所示的那样，即使 L_2 高速缓存有更大容量或者更高的相联数，只要存储块大小不同，包含性就可能被破坏。而且我们也已看到了当 L_1 高速缓存有更高相联度时出现的问题。

幸运的是，在最常见的一种情形包含性是自动保持的。那是这样一种情况：L_1 高速缓存采用直接映像方式（$a_1 = 1$）；L_2 高速缓存可以是直接映像，也可是组相联方式（$a_2 \geqslant 1$），采用任何一种替换算法（如 LRU、FIFO、随机），只要求新的存储块同时被放到 L_1 高速缓存和 L_2 高速缓存，块的大小相同（$b_1 = b_2$），并且 L_1 高速缓存的组数小于或等于 L_2 高速缓存的组数（$n_1 \leqslant n_2$）。使用这种配置就是解决包含性问题的一种流行方法。 395

然而，许多实际使用的高速缓存配置并不能自动在替换时保持包含性。解决的办法是通过在高速缓存层次结构中扩充用于传播一致性事件的机制，使包含性得以保持。一旦 L_2 高速缓存内的存储块被替换，该块地址则传到 L_1 高速缓存，将该块作废或者送出（如果已被修改）。若 $b_2 > b_1$，则可能有多个存储块受到影响。

还需要加强处理总线事务和处理写操作的能力。考虑 L_2 高速缓存能见到的总线事务。有些和 L_2 高速缓存有关的过程也和 L_1 高速缓存相关，因此需要广播到 L_1 高速缓存去。例如，如果 L_2 高速缓存内存储块因某一总线事务（如 BusRdx）被作废，如果数据也在 L_1 高速缓存，那作废也要广播到 L_1 高速缓存。有几种方法可以做到此点。其一是所有与 L_2 高速缓存相关的过程均通知 L_1 高速缓存，由 L_1 高速缓存来忽略地址与其任何标记都不相匹配的事务。这种方法会发送给 L_1 高速缓存大量无用的干扰，而且由于高速缓存的标记不能被处理器访问而降低效率。更好点的解决办法是在 L_2 高速缓存内存储块都设置一新状态（称为“包含位”），记录该块是否也在 L_1 高速缓存内。这种方法用一点额外的硬件资源和复杂性代价，能适当地过滤对 L_1 高速缓存的干扰。

最后，当 L_1 高速缓存写命中时，修改操作要传播给 L_2 高速缓存，从而必要时 L_2 高速缓存能够提供给总线最新数据。一种方法是使得 L_1 高速缓存直写。这样做的优点是单周期的写操作易于实现（Hennessy and Patterson 1996）。可是写操作会消耗 L_2 高速缓存的部分带宽。为避免处理器停转，L_1 高速缓存和 L_2 高速缓存之间需要一写缓冲区。另一种满足需求的方法是 L_1 高速缓存采用回写式，因为 L_2 高速缓存的数据并不需要立即更新而是 L_2 高速缓存要知道 L_1 高速缓存是何时有最近的数据，这样增强了 L_2 高速缓存内存储块的状态信息，使得存储块可以置为“修改了但仍是陈旧的（modified-but-stale）”状态。L_2 高速缓存内存储块可以作为一致性协议中已修改的块，但如果需要传到总线上，数据则从 L_1 高速缓存中提取。（设置 modified-but-stale 状态的简单方法是既置修改位也置作废位）。直写式和回写式的 L_1 高速缓存都被许多基于总线的多处理器使用。保持包含性的详细资料可见（Baer and Wang 1988）。

6.3.2 在高速缓存层次结构中传播一致性的事务

假定我们有包含关系，并且可以根据需要传播作废信号以及发出请求到 L_1 高速缓存， 396

现在看有关的过程如何能够在缓存层次结构上下渗透。层次内的协议通过向下渗透（离开处理器），直到碰到一个持有被请求的处于适当状态的存储块或者到达总线，来处理响应器的请求。对这些处理器请求的回应沿着缓存层次向上传送，随着向处理器方向的推进，依次对缓存进行更新。对读操作的应答，将数据装载到层次中的每个处于共享或者独享状态的缓存中，而对于排他读的响应，要装入除了最内层（L_1 高速缓存）外的所有处于已修改但陈旧状态的层次中。在最内层缓存，排他读数据以已修改状态装入，就好像是写入了新数据，这就是最新的拷贝。

从总线来的请求从外部界面（总线）向上渗透，修改一路上缓存块的状态。有些请求要使得一个存储块送回到总线上，这样的请求可以分成两种，一种是“送回请求”，它也使得该块被作废；另一种是“反拷贝请求”，不要求作废。这些请求向上渗透，直到遇到已修改的拷贝，在这一点，为其外部界面的请求产生一个响应。对于简单的作废，没有必要让总线事务停滞直到所有的拷贝被作废。最底层的缓存控制器（最接近总线的）能够看见这个事务在总线上的出现，对请求者来说，这可以保证作废将会以适当的次序完成。一旦作废请求出现在总线上，对这种作废的响应可能通过它自己的总线界面被送到请求处理器，这样就没有响应在目的缓存层次中产生。所要求的只是要在进来的作废信号和其他通过缓存层次的事务之间保持某种次序，我们将在介绍事务拆分型总线时进一步讨论，这种类型的总线允许多个事务在同一时间处于待完成状态。

有趣的是，在多层缓存的情况下，双标记不是那么关键。L_2 高速缓存的作用相当于是 L_1 高速缓存的一个过滤器，筛取从总线来的不相关的过程，从而 L_1 高速缓存的标记几乎全被处理器可用。类似地，由于 L_1 高速缓存可以看作是 L_2 高速缓存和处理器之间的一个过滤器（期望能满足大多数处理器的请求），L_2 标记几乎全部为总线侦听器的查询所用（图6-7）。尽管如此，许多机器在多层缓存设计中依然保持双标记。

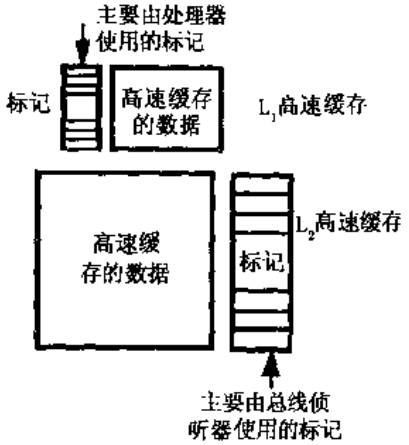


图 6-7　两级侦听式高速缓存的组织。对每个缓存来说只需要一组标记

对于在一个时刻总线上只能有一个事务进行的设计，只要包含关系得以保持，主要的正确性问题并不由于多层次的应用改变多少。必要的事务在层次的上下传播，总线事务可能停滞，直到必要的传播发生。当然，保持总线直到得到一个响应所导致性能的惩罚是更加重的，因此我们试图将这些操作分开。在沿着这条路线进一步下去之前，让我们去掉第二个简化假设，即总线是原子性的假设，考察更大胆些的事务拆分型总线。为简单起见，首先回到单级缓存的情形，然后再考虑多级缓存的层次结构。

397

6.4 事务拆分型总线

对于原子型总线来说，在地址从总线上取走后，直到存储系统或者另一个缓存提供数据响应之前，总线的线路是闲置的，因此原子型总线大大限制了可以发挥出来的总线带宽。在事务拆分型总线的情形，要求得到响应的总线事务被分成两个独立的子事务——一个请求事务和一个响应事务。其他的总线事务（或者子事务）允许在这两个子事务之间发生，这样在产生对一个请求的响应的时候，总线就可能被其他活动利用。在总线和缓存控制器之间设置缓冲，使得总线上有多个活动同时存在，等待来自控制器的侦听和/或数据响应。当然，这里的优点是可以通过总线操作的流水，能更有效地利用总线，从而有更多的处理器能共享相同的总线。缺点是增加了复杂性。

作为请求-响应关系的例子，一个 BusRd 事务现在就是需要一个数据响应的请求。BusUpgr 不需要数据响应，但它要求一个认可信号来指出它已经被确认了从而已经被安排在待完成的序列中。为保证这个认可信号不作为一个单独的事务出现在总线上，它通常是在得到相对于 BusUpgr 请求的总线控制权后被直接送给请求处理器。BusRdX 需要一个数据响应以及一个提交的确认；典型地，它们作为数据响应的部分出现。最后，回写通常没有响应。

拆分型总线所带来的主要新问题有：

1）在侦听和服务一个前面的请求完成之前，一个新的请求可能出现在总线上。特别地，有冲突的请求（两个要求同一存储块，至少一个是写操作）可能在总线上同时处于待完成状态，这是一种必须非常仔细处理的情形。注意这是不同于先前用原子型总线的活动，但宏观上可能出现的非原子性情形。在那里，相互冲突的请求，在它获得总线之前，可以被一个缓存控制器看到，于是就可能在放上总线之前做适当地修改。而这里，两个请求的子事务都已经出现在总线上了。例 6.2 给出了其中的差别。 398

2）在总线和缓存控制器之间，请求和数据响应缓冲区的大小通常是固定的，并且也很小，于是我们必须面对缓冲器充满的问题，要么想办法不让其发生，要么有一个方法来处理它的发生。由于它影响了通过系统的总线事务流，称为是流控问题。

3）由于来自总线的请求被缓冲了，我们需要重新考虑何时、如何在总线上产生侦听响应和数据响应。例如，它们的顺序是否按请求出现在总线上，侦听和数据是不是同一个响应过程的不同部分？

例 6.2 考虑前面两个处理器 P_1 和 P_2 的例子，存储块的缓存状态是共享，并且同时决定对它进行写操作（例 6.1）。说明拆分型总线可能会引入哪些在原子型总线不会出现的复杂性。

解答： 对于事务拆分型总线来说，P_1 和 P_2 可能产生 BusUpgr 请求，这些请求在后续周期中得到总线。例如，P_2 可能在缓存中查询到 P_1 的请求，并且在测得会发生冲突之前得到总线。如果它们两者都假设它们已经获得了排他的所有权，这个协议就出问题了，因为此时两个处理器都认为它们拥有处于已修改状态的存储块。在原子总线上，这种事情是不会发生的，因为第一个 BusUpgr 事务会在第二个处理器得到总线以前完成（包括侦听、响应等阶段），于是这第二个处理器就必须将它的请求从 BusUpgr 变到 BusRdX。（注意，即使在例 6.1 中讨论的原子总线出问题的情况下，也是只有一个处理器中的存储块处于已修改状态，其他处理器中存储块为共享状态。）■

事务拆分、缓存一致性总线的设计有很大的空间，并且许多改进正在工业界进行。也许，从一致性协议的角度看，最关键的问题是如何建立一个序以及报告侦听结果的时机。它们是请求阶段的一部分还是响应阶段的一部分？所采取的立场事实上会影响到对冲突操作的处理方式，即，前面描述的第一个主要问题。关于流控的决定（以及冲突操作）是受同时在总线上允许的未完成请求数影响的。通常，较大的未完成请求数使总线的利用率较高，但要求更多的缓冲和设计复杂性。剩下的高层设计决策是数据响应是否需要以和请求发出相同的顺序返回。Intel Pentium Pro 和 DEC Turbo Laser 总线是“保序”的例子，而 SGI Challenge 和 Sun Enterprise 总线允许变序的响应。后者对于存储访问时间差异的容忍更强（由于存储模块的冲突或者页外的 DRAM 访问，存储器可能会先满足后提交的请求），但更加复杂。让我们先完整地考察一个具体的例子，看这些问题是如何解决的，然后讨论可能的其他方案。

[399]

6.4.1 事务拆分型总线设计的一个例子

这个例子主要基于 SGI Challenge 的总线体系结构，即 Powerpath-2。它在三个设计问题上采取如下做法。对有冲突的请求的处理非常简单或者说保守：这种设计不允许对一个存储块同时有多个请求在总线上处于待完成状态。事实上，它只允许在总线上最多同时有 8 个待完成请求，这样就使得必要的冲突检测可行。在总线和缓存控制器之间提供少量缓冲，对于缓冲的流量控制通过总线上的否认回答，即 NACK 线来实现。即，如果在看到一个请求或者响应事务时缓冲区是满的（它一旦出现在总线上就能被检测出来），这个事务就要被拒绝，给出 NACK 信号；这就显示该事务无效，要发起者重新再试。最后，允许响应的次序和最初请求出现在总线上的次序不同。在这种设计中，一致性过程中全（总线）序的建立发生在请求阶段；然而，从缓存控制器来的侦听结果作为响应阶段的一部分和数据一起（如果有的话）出现在总线上。

现在进一步考察这个总线体系结构设计的例子。首先考虑高层总线设计以及响应如何同请求匹配起来，然后深入看看流控和侦听结果的问题。最后，考察一个请求通过系统的通路，包括如何避免相互竞争的请求在总线上同时呈现待完成状态的。

6.4.2 总线设计和请求－响应的匹配

事务拆分型总线的设计在本质上是用两条分离的总线，一条是请求总线用于命令和地址，一条是用于数据的响应总线。请求总线提供请求的类型（例如，BusRd、BusWB）和目的地址。由于响应可以以和请求不同的次序到达，应该有一个方式来让返回的响应和它们的待完成的请求匹配。当仲裁器将总线控制权给予一个请求（命令-地址对）时，这个请求还得到一个惟一的标记（由于这个设计允许 8 个待完成请求，这个标记为 3 位）。响应包含数据总线上的数据和 3 位宽的标记总线上的原始请求标记。标记的使用意味着响应不需要用地址总线，这就使得它们可为其他请求所用。因此地址和数据总线就能分别仲裁。仲裁以及流控和侦听结果也都有分别的总线线路。

在这个设计中，缓存块是 128 字节（1 024 位），数据总线是 256 位宽，于是 4 个总线周期和一个 1 周期的往返时间要用在响应阶段。下面会讨论一种统一的流水策略，因此请求阶段也是 5 个总线周期：仲裁、冲突解决、地址、译码和确认。总的来说，一个完整的请求-响应过程要用到这其中至少 3 个周期的活动——至少是地址请求阶段（用到地址总线），数

[400]

据请求阶段（用到数据总线仲裁逻辑，为响应子事务获得数据总线的访问），以及一个数据传送或响应（用到数据总线）。三种不同的存储器操作可能同时出现在三个不同的周期中。这种基本流水策略是若干高层设计决策的基础。

为理解这种策略，让我们考虑一个读操作的全过程，如图 6-8 所示。我们从地址请求阶段开始。在请求仲裁周期，缓存控制器向总线提交请求。在请求解决周期，要考虑所有请求，但只有一个会被受理，并分配一个标记。胜者在下面的地址周期驱动地址线，然后所有控制器用一个周期来对它译码并察看缓存标记，查看是不是有侦听命中（侦听结果将在后面提交到总线上）。这时，缓存控制器可以做相应的动作，使这些操作对处理器可见。对于 BusRd，一个独享的存储块降级到共享；对于 BusRdX 或 BusUpgr，存储块就要被作废。在所有情况下，如果一个缓存拥有的存储块处于脏状态，就需要它在响应阶段将那个块送上总线。如果某个缓存控制器在地址阶段不能完成侦听，并且采取必要的行动（比如，如果它没能得到缓存标记的访问），它可以在确认周期冻结这一阶段的继续，直到完成它的侦听。（在确认周期，前一次存储操作的第一次数据传送周期可以进行，占用数据总线的 4 个周期；见图 6-8）。

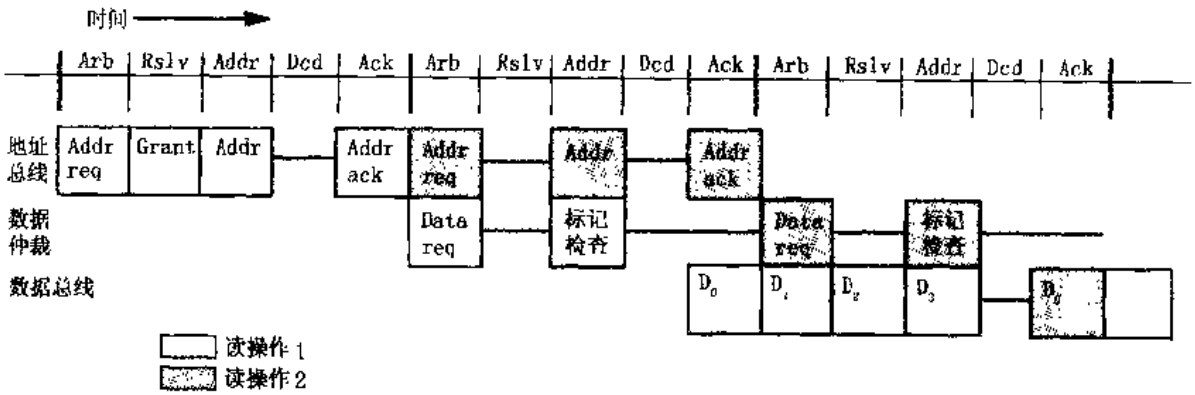


图 6-8　事务拆分型总线的完整的读事务。一对相继的读操作在相继的阶段完成，由阴影框区分。每一个阶段由五个特定的周期组成：仲裁、冲突解决、地址、译码、确认。事务被分为三个阶段：地址请求（用到地址总线）、数据请求（用到数据总线仲裁和相应逻辑）、数据响应（用到数据总线）

在整个总线事务的地址周期之后，我们就知道了是该存储器还是该缓存提出数据响应。响应者可能在下 5 个周期阶段的仲裁周期请求数据总线。（注意在这个周期请求者也在地址总线上发出一个新的请求。）数据总线仲裁在下一个周期解决，并且在这个地址周期可以检查标记。如果目标准备好了，数据传送就从确认周期开始，并在下 3 个周期继续（即进入数据传送或响应阶段）。在一个来回后，就可以开始下一次数据传送（其仲裁并行进行）。缓存块的共享状态（侦听结果）被带到响应阶段，当数据在缓存中被更新后，设置状态位。

401

如前面所讨论的，回写（BusWB）只有请求阶段。它们要一起用到地址和数据线，这样就必须仲裁对这两种资源的同时使用。最后，更新（BusUpgr）用来获取对一个块的排他所有权，由于不需要在总线上有数据响应，也只有一个请求部分。对于通过一个写操作产生了 BusUpgr 的处理器，在这个 BusUpgr 出现在总线后，将得到一个由它自己的总线控制器发出的响应，指出这个写操作已被受理并且已安排在总线操作的序列中。

为了跟踪总线上 8 个已受理的请求，每个缓存控制器维护一个有 8 个条目的表，称为请求表（见图 6-9）。只要一个新的请求发到总线上，它就以同样的索引被加到所有的请求表

中，作为仲裁过程的一部分。索引是在仲裁期间分给那个请求的3位标记。（请求也被单独缓冲在缓存层次中）请求表项包含和请求相联的存储块的地址、请求类型、存储块在本地缓存中的状态（如果它已经被确定了的话），以及其他几位。由于请求表是全相联的，因此要考察所有请求表项和请求的匹配，包括本地处理器发出的请求和在总线上看到的其他请求（用地址域）和响应（用标记）。当对于某个请求的响应出现在总线上时，相应的请求表项就被释放。只是在此时，和请求相联的3位标记值才由总线仲裁器重新分配，因此在请求表里没有冲突。

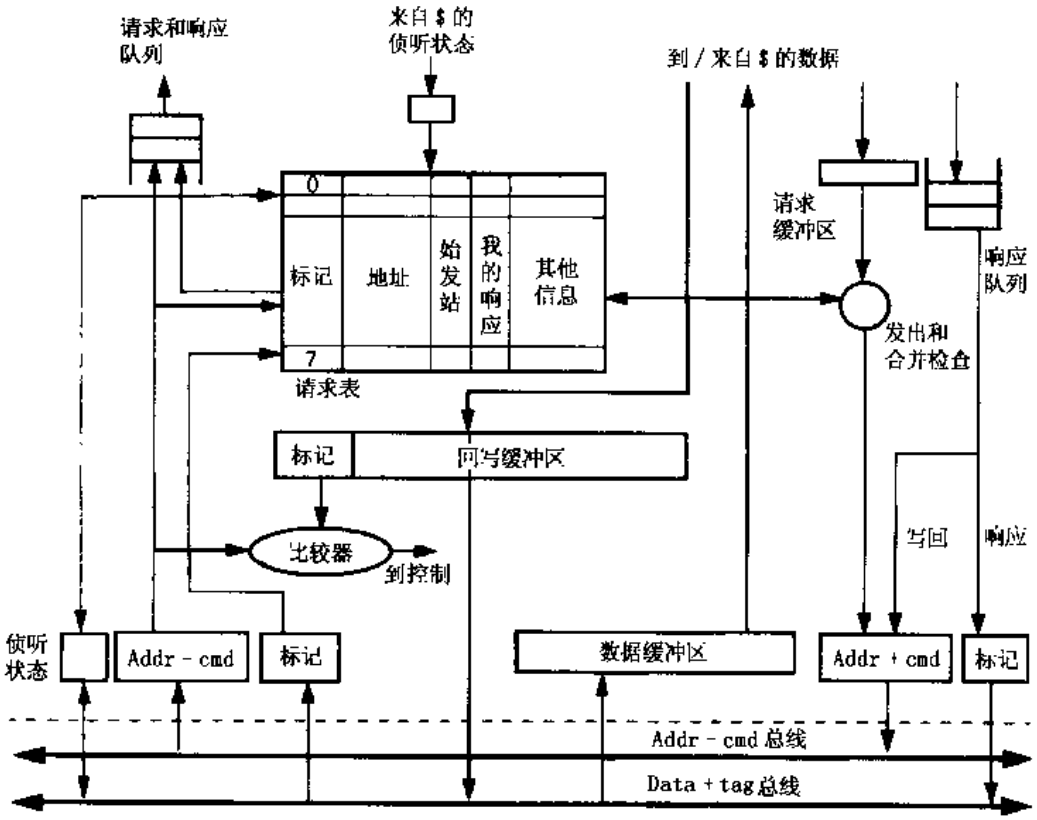


图6-9 图6-4所示的总线接口逻辑的扩充，以完成事务拆分型总线的功能。关键的增加是一个有8个条目的请求表，跟踪所有在总线上待完成的请求。当一个新请求发到总线上，它就被加到所有处理器请求表中的相同的索引处。请求表服务于多个目的，包括请求归并和保证对任何给定的存储块来说只能有一个请求待完成

6.4.3 侦听结果和冲突的请求

如同SGI Challenge，这个设计例子用的是可变延迟侦听法。前边讨论过，总线的侦听部分由三根线或线构成：共享、脏和禁止（它延长当前响应阶段的期间）。当在地址请求阶段末尾确定出哪个模块应该给出数据响应，在数据准备好和响应者获得总线访问之前可能还有许多周期。在这个期间，侦听响应保持在请求表中，其他的请求和响应有可能发生。为了简化请求和侦听结果的匹配，在这个设计中，当所有的控制器看到对一个请求实际的响应被放到总线上时，即在响应阶段，它们就将侦听结果呈现到总线上。回写和升级请求没有数据响应，它们也不要求侦听响应。

避免请求冲突是容易的：由于每个控制器都在它的请求表中有一个记录，表示那些发送

到总线上但未完成的事务，它就不会发出对某个还有待完成事务存储块的请求。这样，即使总线是流水操作，对各个存储单元的操作也是串行化的，同原子情形一样。写操作的确认在请求阶段期间，它影响着串行化。

402 ~ 403

6.4.4 流控制

除了对来源于总线的入向请求外，在系统的其他部分也可能需要流控。除了前面讨论过的回写缓冲区外，缓存子系统有一个缓冲区，其中可以存放对它请求的响应。如果处理器或者缓存每次只允许一个待完成请求，如我们隐含假设的那样，这个响应缓冲区的深度只有一项。由于响应缓冲区项不仅含有一个地址，还包含一个缓存块数据，因此容量就比较大，于是缓冲区项数通常都是很少的。缓存控制器通过限制它所有的待完成请求数来进行流控，使得对每个响应都有缓冲空间。

主存也是需要流控的。除请求本身外，8 个中的每个挂起请求会产生一个主存必须接受的回写。由于回写事务不需要响应，它们可能在总线上迅速连续发生，可能使主存子系统的缓冲区溢出。

由于总线允许对于不同部分进行独立仲裁，SGI Challenge 为总线的地址和数据提供分别的 NACK 线。在一个请求或者一个响应子事务到达它的确认周期和完成之前，主存或任何其他处理器能给出一个 NACK 信号，例如，如果它发现其缓冲区满。然后这个子事务在每个地方被取消，而且必须重新开始。在 Challenge 中，一种常用的选择方案是让那个子事务的请求者做周期性的尝试，直到成功。回退和优先级技术能用来减少失败的重试对带宽的消耗，避免“挨饿”进程。对于遇到缓冲区满的数据传送，Sun Enterprise 用另一种有趣的方案。在这种情形下，接受者（它不能在第一次尝试时接纳数据）当有足够缓冲时启动重试过程。最初的提供者只是简单地看着总线上重试事务，把数据放到总线上。Enterprise 总线的操作保证了当数据到达时，在目的缓冲区的空间是可用的。这就保证了数据传送的成功，只需一次重试总线事务。

6.4.5 一次缓存扑空的路线

给定这个例子设计，我们可以来考察如何处理各种请求，以及会出现什么竞争条件。首先看看这种情况，一个处理器在缓存中有一个读扑空，于是应该产生一个 BusRd 事务的请求部分。这个请求首先检查当前在请求表中挂起的项。如果发现一个地址匹配，取决于挂起请求的性质，它可能做两种不同的动作。

1）如果当前的请求是针对相同块的，这对处理器来说是一个好消息：这个请求不需要送上总线，而可以在对先前请求的响应出现在总线上时直接获得数据。为做到这一点，我们在请求表中的每一项加上两位，这表示：我希望获得这个请求的数据响应吗？我是这个请求的最初产生者吗？在我们的情形中，这些位被分别置 1 和 0。第一位的目的是明显的；第二位的目的是要帮助确定在哪个状态下（独享还是共享）这个数据响应被装入。如果一个处理器不是最初的请求者，那么它必须在它从总线获得相应数据时，在侦听总线上声明共享线；从而所有缓存都将这一存储块以共享态装入，而不是独享。如果某个处理器是最初的请求者，它从总线得到响应后不会给出共享信号；如果共享信号根本没有出现，那它就以独享态装入该存储块。

404

2）如果前面的请求和 BusRd 冲突（例如 BusRdX），控制器就必须保持请求，直到它看到在总线上有一个对先前请求的响应，而且在那之后才尝试这个请求。处理器方的控制器通常对此负责。

如果控制器在请求表中找不到匹配项，它就可以直接向总线发请求。不过，它必须关注前面提过的那种竞争条件。当控制器首先检查请求表时，它可能找不到冲突请求，因此它可能请求总线仲裁。然而，在它被赋予总线控制权之前，一个冲突请求可能出现在总线上，然后它可能被赋予紧接着的下一次总线使用权。由于这个设计不允许总线上有冲突请求，当控制器看到在它前面的时隙中正好有一个冲突请求的时候，它应该 1）向总线发出一个空请求（没有动作的请求），以占据它已被赋予的时隙；2）从后面的仲裁中退出，直到对这个冲突请求的响应产生。

假设处理器来管理在总线上发出 BusRd 请求，其他缓存控制器和主存控制器应该干什么？请求一旦出现在总线上，就进入所有缓存控制器的请求表，包括发出请求的那一个。控制器开始检查它的缓存，看有没有被请求的存储块。主存子系统根本不知道这个块是否在某个处理器缓存中是脏的，因此独立地开始取出这一存储块。现在要考虑三种不同的情况。

1）某个缓存可能发现它的相应存储块处于修改状态，于是可能在主存响应之前得到总线，从而产生一个响应。在总线上一看到这个响应，主存就放弃它所启动的取数据动作，等待着这一存储块的缓存控制器将数据装载进来，其状态取决于侦听线上的值。如果一个缓存控制器在响应出现在总线上之前没有完成侦听，它就会保持禁止线的作用状态，响应周期就被延长（即将呆在总线上）。由于这个块在缓存中是脏的，主存也接收这个响应。如果主存没有所需的缓冲空间，它就给出用于流控的 NACK 信号，控制器有责任保持存储块的脏态并在以后重试这个响应事务。

405

2）在持有脏块的缓存控制器完成它的侦听和/或获取总线之前，主存可能要取数据并获得总线。持有脏块的控制器将首先置禁止线，直到它完成它的侦听；然后置脏线并且释放禁止线，这就告诉存储器，它有最新的拷贝，而且存储器不应该将数据放到总线上。观察到这脏线后，存储器取消它的响应事务，从而不将数据放到总线上。带有脏块的缓存稍后将获得总线并将数据响应放到总线上。

3）最简单的情形是，没有缓存有脏块。主存将获得总线并产生这个响应。没有完成它们侦听的缓存控制器在看到来自存储器的响应后将给出禁止信号，但一旦它们取消，存储器就可以提供数据。（在这个系统中，缓存到缓存之间的共享技术没有用于共享态的数据）

处理器写操作的处理类似于读。如果进行写操作的处理器在它自己的缓存中没有发现处于有效状态的数据，就会产生 BusRdX。和前面一样，它检查请求表，然后到总线上。除了主存不会接受来自另一个缓存的数据响应（由于它将被写者再次修改）以及没有其他处理器能抓住这个数据外，各个方面都和总线读相同。如果被写的存储块是有效的但处于共享状态，BusUpgr 就会被发出。这不要求有响应事务（当前有效的存储块在主存，也在写者的缓存）；然而，如果任何其他处理器正准备对同一存储块发 BusRdX，它现在就要将请求转换为 BusRdX，这和原子总线是一样的。

6.4.6　串行化和顺序同一性

考虑对单个存储单元操作的串行化问题。如果出现在总线上的一个请求子事务是读，在

这个读后面出现在总线上的写应该不能改变由这个读返回的值。尽管在总线上有多个待完成的事务，由于对相同单元的冲突请求不允许同时出现在总线上，做到这一点也是不难的；因此读响应子事务将优先于这个写请求，它将在写操作能影响缓存的值之前完成。如果出现在总线上的事务是由一个写操作产生的 BusRdX 或者 BusUpgr，请求缓存将对缓存阵列完成这个写，在响应阶段后和发出任何其他存储操作之前发生；从任何其他处理器来的对该存储块其后（冲突的）的读只允许在该写的响应阶段之后的总线上发生，因此它们保证得到的是新数据。（回顾前面讨论过的，写操作的响应阶段可能是总线上的一个单独的操作，如同 BusRdX，或者是在请求赢得仲裁后隐式产生的，如同 BusUpgr。）

现在考虑对不同单元操作的串行化所涉及的顺序同一性问题。总线过程逻辑全序的建立是通过对地址总线请求所赋予的序来确立的。一旦 BusRdX 或 BusUpgr 得到了总线，相联的写就被认可了。然而，对于总线上有多个待完成请求的情况，作废也被缓冲起来，可能在它们被实际应用到缓存上之前还有一段时间（不同于原子总线，那里作废是立即发生）写操作的认可不担保由这个写所产生的值已为所有其他处理器可见；只有实际的完成才能保证。（针对一个处理器的动作对该处理器有保证）需要有进一步的机制来保证所需的序在总线和处理器之间能够维持。例 6.3 将这一点具体化了。 406

例 6.3　考虑如下所示的两个代码段。在 SC 要求下，（A，B）的什么结果是不允许的？假设每个处理器只有一级缓存，总线上有多个待完成事务，没有特别的机制来保持总线和缓存或者处理器之间的次序，表明如何能得到不允许的结果。假定一种基于作废的协议，在两个缓存中 A 和 B 的初值都是 0。

P_1	P_2
A = 1	rd B
B = 1	rd A

P_1	P_2
A = 1	B = 1
rd B	rd A

解答：在左边的第一个例子中，SC 条件下不允许的结果是（A，B）=（0，1）。然而，考虑下面的情形。P_1 对 A 的写认可，于是它继续写 B（在修订后的 SC 充分条件之下）。在 A 之前，B 的作废应用到 P_2 的缓存，这是由于它们在缓冲区中重新定序。P_2 在 B 上引起一个读扑空，得到新的值 1。然而，A 的作废仍然是在缓冲区中，即使在 P_2 发出读 A 之前也应用不到 P_2 的缓存。读 A 是一个命中，它的完成使 A 从缓存中得到旧值 0。

右边的例子对于破坏 SC 并不需要作废的重新排序。不允许的结果是（0，0）。然而，考虑下面的情形。P_1 发出并认可它对 A 的写，然后完成对 B 的读，读进旧值 0。P_2 然后写 B 并认可，因此 P_2 继续读 A。对 B 的写出现在总线上（认可）是在 A 的写之后，因此它们应该以这种次序串行化，P_2 应该读到 A 的新值。然而，对应于 P_1 写 A 的作废仍在 P_2 的输入缓冲区，还没有被应用到 P_2 的缓存中。P_2 看到 A 上的一个读命中，并完成返回 A 的旧值 0。■

如果我们用认可代替完成，并且允许在总线和处理器之间有多个待完成操作缓冲起来，此时为保持顺序同一性必须的关键性质是：不应该允许一个处理器在先前的写（总线序）为它所见之前，实际看到的是由于当前写所产生的新值。有两种保持这种性质的方式：不让某些从总线到缓存的输入过程在输入队列中重定序；允许重定序，但其后要保证重要的序在机器中的必要点得到维持。下面简单考察这两种做法。 407

一种遵循第一种策略简单的方法是保证所有从总线来的事务（作废、读扑空答复、写提

交确认等）以 FIFO 序传播到处理器。然而，这样一种严格的定序是不必要的。针对一种基于作废的协议，考虑维持刚才描述过的所期望的性质。这里，有两种方式让新的值读到缓存中，并且使它为处理器可读而不引起另外的总线操作。一种是通过一个读扑空，另一个是通过那个处理器的一个写。另一方面，一个处理器可以通过其他处理器的作废操作，来看见那些处理器的写（即使这个值在本地还不可用）。为了使写操作定义成前面那样提供新值的操作，它们必须在操作前出现在总线上（在写命中的情形，则是一个先前发自那个处理器的总线事务）。因此，在相关事务出现在总线上的时候，也就是当它的响应回来时，这些写所产生的作废就已经在输入队列中了，或者兑现到缓存中了。因此，我们需要保证的是一个答复（读扑空或者写提交确认）在总线和缓存之间不越过一个作废，即所有先前的作废都兑现，先于缓存接收到这个答复。

注意，进入队列中的作废可能会被重新定序。这是因为对应于一个作废的新值只有通过相应的读扑空时才被看见，而对读扑空的答复不允许相对先前的作废重新定序。另一方面，在一种基于更新的协议下，一旦进来的更新被兑现，就能够看见相应写的新值。这意味着，不仅答复不应该超越更新，更新之间也不应该相互超越。

一个备选方案是允许从总线来的入向事务在它们到达缓存的路上随机重新定序，但要保证的是：如果一个来自本地处理器的操作使它能看见一个新值，则在这个操作被完成之前，所有事先认可的写被兑现到缓存中（通过从入向队列提供服务）。毕竟，重要的不是作废或者更新被兑现的次序，而是相应的新值能被处理器看到的顺序。有两种自然的方法能达到这一点。其一是每当处理器试图完成一个向缓存写入新值的操作时，处理从队列中来的入向作废和更新。在基于作废的协议，这意味着要首先处理队列，先于允许处理器来完成一个读扑空或者一个要产生总线过程的写；在基于更新的协议中，它意味着对每次读命中都要服务。另一个方式是当一个处理器真正快要访问一个值的时候（完成一个读命中或扑空）来处理队列，如果一个新值（即一个答复或一个自从上次队列被服务后的一次更新）的确兑现到了缓存。从总线到缓存的操作重新定序，以及一个新值被兑现到缓存的事实意味着，作废或者更新可能在队列中，对应于先前和那个新值有关的写；这些写现在应该在读能够完成之前兑现。这些技术不会引起如例6.3所讲的不希望的结果，见本章的习题。这些习题可能有助于使这些技术更具体化。正如我们即将能看到的，将这些技术扩充到多层缓存结构是相当自然的。

408

不管用哪种方法，写的原子性由总线的广播特性自然地得到保证。这个总线隐含着写操作被认可的顺序对所有处理器都是相同的，并且在写对于所有处理器都认可之前，一个读没法看到由它产生的值。（注意到写处理器也在本地保证这一点）。用前面这些技术，我们就能用完成来替代提交，从而保证原子性。对于事务拆分型总线的另一些主要的正确性问题——死锁、活锁、挨饿——在我们介绍了多层缓存之后会有讨论。首先，让我们看看在事务拆分型总线上协议设计的一些方法。

6.4.7 其他设计选择

针对请求－响应的定序、请求冲突的处理和流控，人们还研究了其他一些方法，不同于我们用作例子的事务拆分型总线设计。例如，保证响应在总线上产生的次序和请求的次序一致——缓存控制器倾向于此——将简化设计。为了请求－响应的匹配，全相联的请求表可以被一个简单的 FIFO 缓冲区代替（如果不允许冲突的请求，全相联查询可能依然是需要的）。

如前所述，只有当一个请求实际出现在总线上时，它才被放到FIFO中，保证所有实体（处理器和存储系统）对未决请求都有完全相同的看法。缓存控制器和存储系统按FIFO序处理请求。当响应出现的时候（如前面的设计），如果其他的处理器还没有完成它们的侦听，它们就要置禁止线，延长其事务的周期，即侦听依然和响应一起报告出来。区别在于，尽管一个处理器在它的缓存中有一脏块，但存储器先产生了一个响应。在先前无序设计的情况下，有着脏块的缓存控制器释放禁止线，置脏线，当它从缓存获得了数据后再次参与总线的仲裁。但现在为了保持FIFO序，这个响应必须放到总线上，先于针对任何后来请求的响应。于是带有脏块的控制器不释放禁止线，而是延长当前的总线事务，直到它从其缓存中得到那一块数据提供给总线。实现这一点和任何其他实体访问总线无关，因此没有死锁问题。

尽管FIFO请求-响应序是比较简单，它可能有性能问题。考虑带有交叉存储系统的多处理器。假设有三个请求 A，B，C 依次发到总线上，A 和 B 到同一个存储模块，C 到另一个存储模块。迫使系统按序产生响应意味着 C 要等待 A 和 B 的处理完成，尽管由于 A 和 B 发生模块冲突，C 的数据先于 B 可用。主存的行为是允许变序响应主要的动机，因为缓存不管怎样都可能是按序对请求做响应的。 409

让响应保持其顺序，也使得处理总线上同时存在的对同一存储块的冲突请求更容易些，这样就消除了对全相联请求表查询的需要，也增加了带宽。假设两个BusRdX请求连续发给一个存储块。如前所述，发出后面一个请求的控制器在看到前面的请求后将要作废它的存储块。对于事务拆分型总线来说，问题在于发出前面请求的控制器，在它所等待的数据响应之前看到了后面的请求。控制器不能由于这个后面的请求简单地作废它的块，因为那一存储块正在去往发出前面请求的控制器的路上，它自己的写应该先于一个请除或作废之前完成。对于乱序响应来说，允许这种冲突请求可能是困难的。对于保序响应，前面的请求者知道它的响应会首先出现在总线上，因此这实际上是一种性能优化的机会。前面请求的控制器对后面请求的反应是简单地注意到后者是待完成的。当它的响应块到达时，它就更新要写入的字，并且“顺手”将修改后的存储块送回到总线上，作为对后面请求的响应，使它自己的块无效。这种优化降低了在写-写伪共享下往复转换一个存储块的时延。

如果从请求到侦听结果的延迟是固定的，就可以允许冲突请求的存在，甚至不需要数据响应的保序。然而，由于对一个存储块的冲突请求也要到存储器中的同样一个队列中，对这些请求的数据响应本身通常就是按序出现的，于是它们可以由上述快捷的方法来处理（在Sun Enterprise系统中就是这么做的）。

事实上，只要一个明确定义的序能在请求过程之间识别出来，它们甚至不需要顺序地发到同一个的总线上。例如，Sun SparcCenter 2000用两个不同的事务拆分型总线，CRAY 6400用四个，以改善大配置系统的带宽。于是就可以在一个周期中发出多个请求。同时，在总线之间建立起一个简单的优先级，从而在并发的请求中间定义了一个逻辑顺序。

6.4.8 带有多级高速缓存的事务拆分型总线

现在可以将基本协议上的两种主要的增强技术结合起来：多级缓存和事务拆分型总线。我们考察的设计是一个（类似于Challenge）事务拆分型总线和两级缓存层次结构。其中的问题和解决方案可以推广到更深的层次。我们已经看到了请求、响应和作废沿这个层次结构传播的基本问题。我们需要对付的新的关键问题是，一个请求传播通过控制器要占用相当多的

410 总线周期。在这个期间，我们必须允许其他的过程也在层次中上下传播。为了允许单个单元（例如，控制器和缓存）以它们自己的速度运行的同时维持高带宽，也在层次的级别之间布置了队列。然而，这就引起了关于死锁和串行化的一系列问题。

一个简单的多级缓存组织如图 6-10 所示，假设一个处理器同时只能有一个待完成请求，于是在处理器和第一级缓存之间就没有队列。这种队列结构要关心的一个问题是死锁。为了避免前面所讨论过的取数死锁问题，L_2 缓存在有请求未完成时需要能缓冲入向请求或响应，从而使总线得以自由。对于每个处理器有一个待完成请求的情形，在总线和 L_2 缓存之间的入向队列要足够大，能够容纳从其他处理器来的待完成请求数加上对它自己请求的一个响应。这覆盖了所有请求都到达同一个缓存且这个缓存有一个请求待完成的情况。如果队列比这种情况的要小，在队列满了不足以接收新的请求时，总线请求就应该被给予 NACK 信号。这种说法适用于带有事务拆分型总线的单级或多级缓存的系统。在总线到 L_2 之间和在 L_2 到 L_1 中的一个时隙预留给对处理器待完成请求的响应，以便每个处理器总能排干它的待完成响应。如果用 NACK，总线仲裁需要含有一种机制，诸如一种简单的优先级方案，来保证在严重的竞争条件下使系统仍能向前推进。

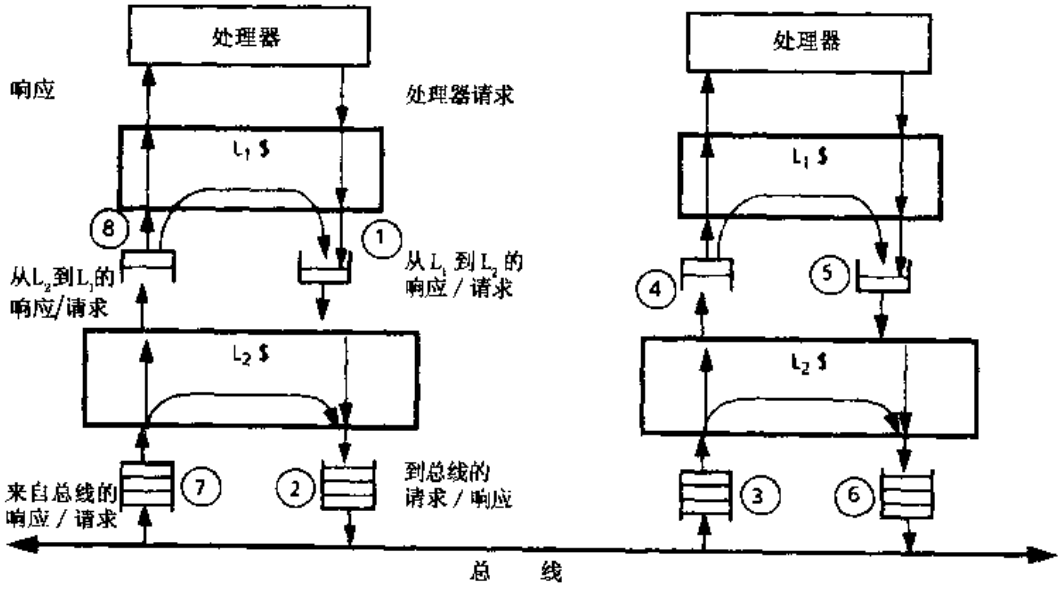


图 6-10 可能存在于多级缓存层次结构内部的队列。每个层次有一个自上的输入队列和一个自下的输入队列。一个操作可能产生对相邻层的请求或者响应。例如，一个在 L_1 缓存扑空的读请求被送到 L_2 缓存①。如果也扑空，总线上就会出现一个请求②。在输入队列中该读请求被所有其他缓存控制器捕获③。假设存储块当前以修改状态出现在另一个处理器的 L_1 缓存中，这个请求就在它的 L_1 服务中排队④。L_1 将该存储块降到共享状态，并送到 L_2 缓存中⑤，由它再送上总线⑥。这个响应被请求者捕获⑦并送给它的 $L_1$⑧，在那里向处理器提供所要求的存储字

除了取数死锁外，经典的缓冲区死锁也可能在多层缓存层次之内发生。例如，假设在 L_1 和 L_2 缓存之间每个方向都有一个队列，两者都是回写缓存的，每个队列能持有一个数据项。有可能 $L_1 \rightarrow L_2$ 队列持有一个出向读请求，它能够在 L_2 中被满足，但要对 L_1 产生一个回答；$L_2 \rightarrow L_1$ 队列持有一个入向读请求，它能够在 L_1 缓存中被满足，但要对 L_2 产生一个回答。现在有了一个典型的循环缓冲依赖关系，因此死锁。注意，这个问题只出现在这样的层

次结构中：上层的缓存（靠近处理器），而不是最靠近总线的，是一个回写缓存。否则，入向请求不产生来自高层缓存的答复，于是就没有循环，也就没有缓冲区死锁问题（注意，作废的确认被总线自己隐含完成，不需要来自缓存的确认）。

在多层回写缓存层次系统中，一种对付这种缓冲区死锁问题的硬件方法是限制从处理器来的待完成请求数，在每个缓存对入向请求和响应提供足够的带宽。然而，这要求用到大量的器件并且不是可扩展的。每个请求可能需要两个出向缓冲区项——一个是为请求，另一个是为它可能产生的回写。对于允许大量待完成总线事务的情况，入向缓冲区也可能需要有许多项。另外一种方法是用一种通用的死锁回避技术，针对有限缓冲的情形，我们将在第7章和物理上分布存储的系统一并讨论，在那种情况下，这个问题更加明确。基本思想是将流过缓冲区和通信介质的操作分离成请求和响应。一个操作被分为响应，如果它不产生进一步的操作，只是简单地被其目的地吸收。一个请求可能产生一个响应，但没有操作能产生请求（尽管在这种情况下，一个请求如果不在最初层次产生一个响应，则可能传送到层次的下一个级别，初始化一个新的请求-响应对，将自己结束）。针对这种分类，我们可以避免死锁，如果我们在每个方向为请求和响应提供单独的队列，保证响应总被从队列上抽取出来，这样也允许请求向前推进。在第7章讨论了这个技术后，我们在练习中将它应用到这个特别的带有多层回写缓存的情形。

还有其他一些潜在的死锁情况。例如，如果总线上的待完成过程数小于缓存允许的待完成请求数，处理器缓存来的一个响应可能需要先于从它发出的新的外向请求到达总线。否则，存在的请求可能总不会得到满足，于是就不会有进展。出向队列或者队列组必须支持在必要时响应能够绕过请求。

411
412

除死锁外，对于这些队列结构的一个考虑是维护顺序同一性。对于多级缓存来说，重要的是总线不要等待一个作废信号一直上朔到第一级缓存，返回一个答复；反过来它应该考虑放上总线后所提交的写，这样的写位于最低级缓存的输入队列中。提交和完成的分离在这种情形更加明显。然而，我们所讨论过的对于单级缓存的技术在此自然地得到扩充：我们只要简单地将它们应用到缓存层次的每一级。这样，在一种基于作废的协议中，第一个技术扩充到保证在层次的每一级答复不要针对那一级的入向队列中的作废重定序（为这个目的，从低级缓存到高一级缓存的答复也被当成答复处理）。第二个技术的扩充体现在，在对某一层次的入向作废兑现之前，不让出向存储操作推进超过该存储层次；或者是排空对一个层次的入向作废，如果一个答复自从上一次排空已经在那一级兑现的话。

6.4.9 对一个处理器有多个待完成扑空的支持

尽管我们已经考察了事务拆分型总线，到目前为止我们还是隐含地假设了一个给定的处理器在一个时刻只能有一个待完成的存储请求。这个假设对现代处理器来说是简化的，即使是单处理器系统，现在的处理器都允许多个待完成请求以包容缓存扑空的时延。一方面，允许处理器有多个待完成引用会改善性能，但它也将语义复杂化了，使得来自同一处理器的存储访问可能在存储系统中完成的序不同于它们发出的序。

多个待完成引用的一个例子是用写缓冲区。由于希望让处理器在发出了第一个写操作后继续进行其他的计算甚至存储操作，我们将这个写操作放到写缓冲区。直到写被串行化，它不应该可见；否则，它可能违反写串行化和一致性。一个可能性是将它写到本地缓存，但在

独占所有权得到之前不使它可用（即在此之前不让缓存响应关于它的请求）。更通常的一个办法是将它保持在写缓冲区中，只是当得到了独占所有权才将其放入缓存。

多数处理器用更大胆的方式使用写缓冲区，处理器不停地快速发出一个写序列到写缓冲区。在单处理器，只要读操作检测写缓冲区以满足相关性，这个方法就非常有效。在多处理器中的问题是，通常在关于一个存储块的独占所有权过程被放到总线上，因此被串行化之前，不能允许处理器推进让存储操作超越这个写。然而，有些特别的情形，处理器能够发出一个写序列，并认为它们是完成了，不需停滞。一个例子是是否能确定写操作所针对的存储块处于已修改状态。然后它们可以在处理器和缓存之间被缓冲，只要缓存在服务来自总线方的请求（对一致性来说是相同的块，对 SC 来说是任何块）之前处理这些写操作。存在一个重要的特殊情况，写序列能被缓冲而不管缓存的状态：写都是针对同一个存储块，没有来自那个处理器的其他存储操作插在这些写之间。当控制器正在得到读排他事务的总线时，这些写可能被归并。当那个事务出现的时候，它立刻使整个写序列可见。这个行为如同在这个总线事务后但在下一个总线事务之前写在本地命中并完成。注意，回写序列没有问题，协议不要求它们有什么顺序。

413

更一般地讲，为了满足顺序同一性的充分条件，一个有能力跨越待完成的写，甚至读操作的处理器提出了这样的问题：哪个实体应该等待来“发出”一个操作，直到程序原序前面的那个完成。强迫处理器自己等待，可能会丧失所有由复杂的处理器机制带来的好处（诸如写缓冲和乱序执行）。相反，由于这里的问题是可见度，持有待完成操作的缓冲区（诸如写缓冲或者是在动态调度乱序执行处理器中的重定序缓冲）可以为此目的服务。处理器能够紧接着前一个操作发出下一个操作，缓冲区承担的责任有两方面：一是担保在适当的时间之前写操作不被存储器和互连系统可见（即不将它们发到外部可见的存储系统）；二是读操作不允许乱序完成（相对于待完成写提交的次序），即使处理器可能乱序发出并执行它们。单处理器中提供精确中断的机制通常可用于这里的缓冲区，在后面的章节中将讨论它们。当然，某些较简单的处理器，不支持操作推进跨越读或写，能使顺序同一性的维护比较容易些。在第 9 章考察同一性模型的时候，还会进一步讨论多个待完成引用对存储同一性模型的语义影响。

从设计的角度来看，最有效地开拓多重待完成引用要求缓存允许在同一时间有多个缓存扑空待完成，从而这些扑空的时延能被重叠。这反过来要求缓存或者某些辅助数据结构来跟踪待完成的扑空；由于响应可能乱序返回，这可能是相当复杂的。允许多重待完成扑空的缓存称为免锁定缓存（Kroft 1981），相对于阻塞缓存，其只允许一个待完成扑空。在第 11 章中，讨论包容时延措施的时候，也将讨论免锁定缓存的设计。

最后，考虑事务拆分型总线和多级缓存层次之间的相互作用，以及避免死锁的需求。给定一个支持事务拆分型总线和多级缓存的设计，为支持每个处理器有多个待完成操作所需的扩充并不多，且多数只是出于性能的考虑。我们只是需要在处理器到总线之间提供较深的请求队列（在图 6-10 中，请求队列向下指），因此多个待完成操作能够缓冲起来，于是处理器或者缓存能够不停滞。更深的响应队列和较多的回写缓冲也可能是有用的，这时系统能承受更多的并发性。只要死锁的处理是通过不同于响应的请求来完成的，并且为它们提供逻辑上分开的缓冲，这些队列的准确长度对正确性的影响都不大。我们看到这里做的改变是相当小的，其原因是免锁定缓存本身对同一块响应的任务做了复杂的融合请求和管理，因此对于缓存以及它下面的总线子系统来说，只看到对于不同块的多个请求来自该处理器。某些潜在的

414

取死锁情形可能会暴露出来，它们在每处理器只有一个待完成请求情形不会出现；例如，我们现在可能看到这种情形，从所有处理器来的待完成请求的总数多于总线能够接受的数量；因此，需要保证响应能顺便旁路掉一些请求。尽管如此，我们讨论过的支持多个待完成过程的技术（针对事务拆分型总线）使系统的其余部分能够对付源于一个处理器的多个请求，而不产生死锁。

6.5 实例分析：SGI Challenge 和 Sun Enterprise 6000

前面的部分论述了一般性的设计和实现，接下来将讨论两个具体的基于总线的多处理器系统——SGI Challenge 和 Sun Enterprise 6000。我们将更多地集中于讨论这些实际系统的组织和工程方面的问题，而较少地注重它们的逻辑，考察它们在若干问题上采取的不同做法。

SGI Challenge 被设计成为能支持到 36 个 MIPS R4400 处理器（最高能达到 2.7 GFLOPS）或 18 个 MIPS R8000 处理器（最高能达到 5.4 GFLOPS）。两者都使用 Powerpath-2 总线，最高带宽 1.2 GBps。此系统能支持高达 16 GB 的 8-通路交叉存取主存储器，多达 4 个 PowerChannel-2 I/O 总线（每个总线支持最高带宽 320 MBps 和多以太网连接、VME/SCSI 总线、图形卡以及其他外围设备）。全部磁盘存储容量能达到几万亿字节。操作系统使用 IRIX（它是 SVR4 UNIX 的一个变型。具有对称多处理器内核，能把操作系统的任务分配到任一个处理器上进行）。图 6-11 给出了 SGI Challenge 系统的高层组织结构。

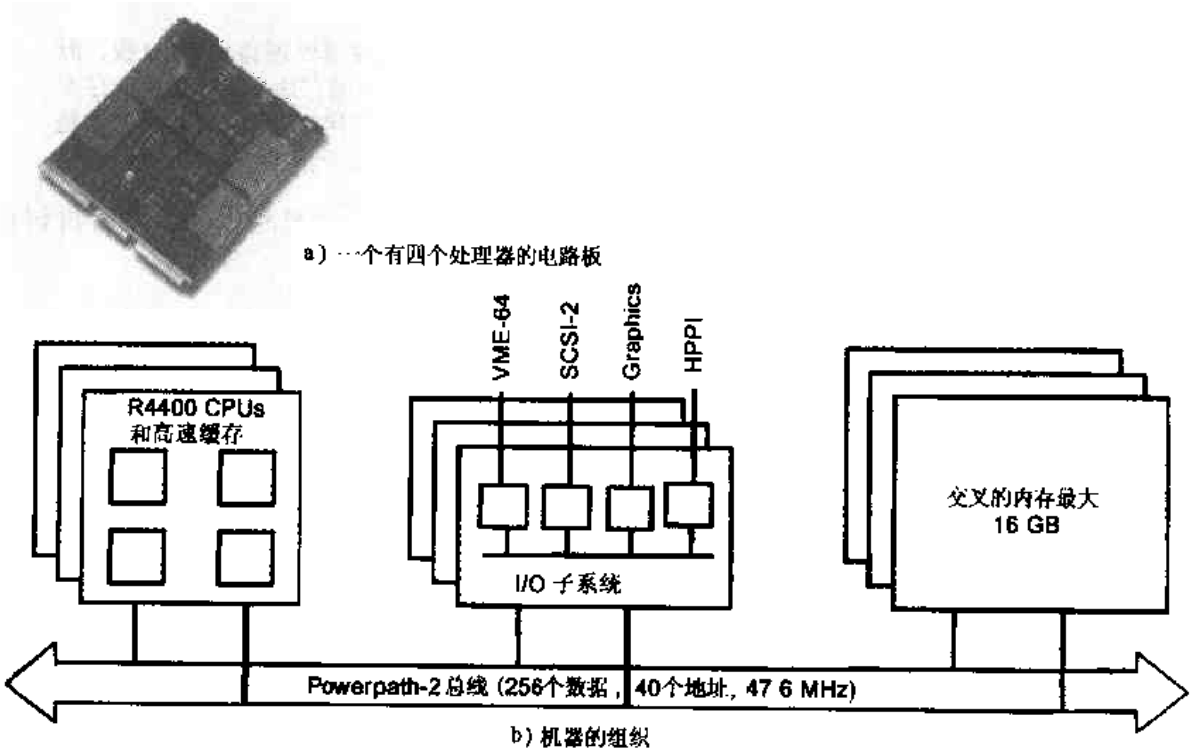


图 6-11 SGI Challenge 多处理器 每个板有 4 个处理器，9 个总线槽共可容纳 36 个处理器。此系统能支持高达 16 GB 的 8 通路交错主存储器。I/O 板提供一个独立的 320 MBps 的 I/O 总线，通过它与其他标准总线和设备相接。系统总线有一个独立的 40 位地址通路、一个独立的 256 位数据通路以及命令通路与其他信号的通路，支持最高带宽 1.2 GBps。总线是事务拆分型的，在给定的时间内可以有多达 8 个请求在总线上等待处理

来源：CHALLENGE 是 Silicon Graphics Inc. 公司的注册商标。

Sun Enterprise 6000 被设计成能支持多达 30 个 UltraSparc 处理器（最高能达到 9 GFLOPS），支持 Gigaplane 系统总线（总线支持最高带宽 2.67 GBps），支持高达 30 GB 的 16 通路交叉存取存储器。机器有 16 条槽，能混合装备处理板和 I/O 板，且每种板至少需有一个。每个处理板有两个 CPU 模块，两个最高达 1 GB 的 512 位宽的内存模块（因此，系统的内存容量和带宽受限于处理器数目）。虽然特定的内存在物理位置上与一特定的处理器对相连，但由于对内存的存取都是通过系统总线进行的，故各内存的存取时间都是一致的。每个内存都有特定的地址，它所处的电路板就叫做此特定地址的主板（home board）。每个 I/O 板都提供两个独立的 64 位 × 25 MHz 的 SBUS I/O 总线，因此，系统的 I/O 带宽受限于 I/O 板的数量。所有磁盘存储容量能达到数十万亿字节。操作系统使用 Solaris UNIX。图 6-12 给出了 Sun Enterprise 系统的高层结构。

415 ~ 416

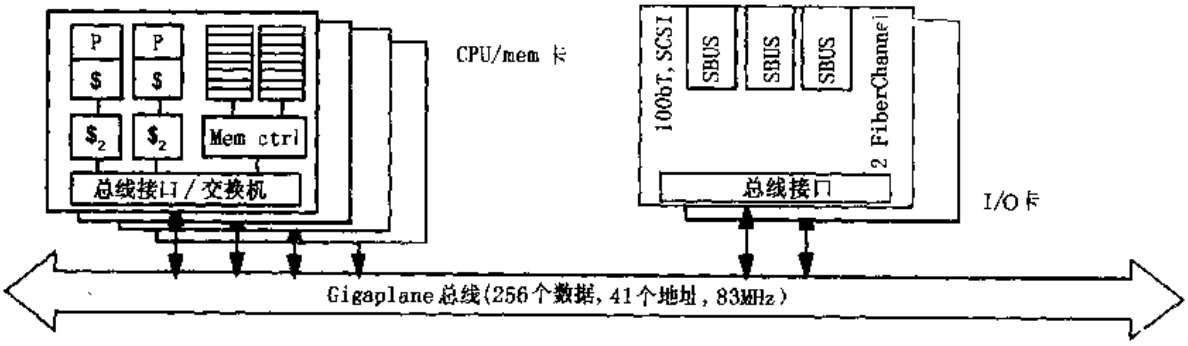


图 6-12 Sun Enterprise 6000 多处理器。系统提供了 16 条总线槽，能容纳处理器板或 I/O 板，但每种都至少得存在一个。处理器板能容纳两个处理器和两个内存库，对所有板内存库都能被一致地存取。I/O 板为多种独立的外部设备总线建立了接口，因此看起来就像是系统总线的高速缓存控制器。事务拆分型总线允许有多达 112 个请求在总线上等待处理

下面几小节将描述 SCI Challenge 的体系结构，并给出它的一些性能特征，然后再讨论 Sun Enterprise 6000。

6.5.1 SGI Powerpath-2 系统总线

系统总线是系统内各部件间相互连接的核心。因此，它的设计要考虑所有其他部件的请求，相应地，一旦采用了某种设计方案，也会影响其他部件的设计。总线设计方案的选择要考虑如下因素：多选还是非多选的地址和数据总线、较宽的数据总线（如 256 位或 128 位）还是较窄的数据总线（如 64 位）、总线的时钟频率（它受所用的信号发送技术、总线长度、槽的数目等的影响）、事务拆分型还是原子型设计、流控策略等等。Powerpath-2 系统总线是非多选的，它有 256 位宽的数据部分、独立的 40 位宽的地址部分、以及命令部分和其他信号部分。它采用的时钟频率是 47.6 MHz，它是事务拆分型设计，并支持 8 个等待的读请求。然而宽数据通路意味着其他设备连接到总线的硬件开销是较大的（连接要求多个位片芯片做接口），好处是一个适中的时钟频率也能得到 1.2 GBps 的高带宽。总线支持 16 个槽，其中的 9 个能装备四处理器板，从而达到 36 个处理器的配置。总线的宽度也影响和受影响于其他的设计。例如，最接近总线的高速缓存（在这指二级高速缓存）的块大小是 128 字节，隐含着整个块能在 4 个总线时钟周期内被总线传输；由于两次传输之间死循环的影响，块越小，总线流程效率越低或设计更复杂。另外，由于总线连接大，也要求板相当的大。总线接口分

布在板边缘，占了大约 20% 的面积，这很自然地要在每个板上多放几个处理器。

让我们更仔细地看一下 Powerpath-2 总线的设计。总线共包括 329 个（位）信号：256 位数据，8 位数据奇偶校验，40 位地址，8 位命令，2 位地址 + 命令奇偶校验，8 位数据资源 ID，和 7 个其他信号。总线上事务的类型少，所有的事务处理都花费 5 个周期，这和前面讨论过的例子一致。所有的总线控制器专用芯片（ASIC）都同步地执行如下的五态自动机：仲裁、解决、写址、译码、确认。但没有事务发生时，每个总线控制器都进入两态空转自动机。这种更小的两态空转自动机使得每个新请求都能立刻得到仲裁，而不像等待五态自动机的仲裁要经过更长的时间（至少需要两个状态，如果只有一个，将会阻止不同的请求者驱动仲裁进入下面的周期）。图 6-13 给出了基本总线协议的状态机。 417

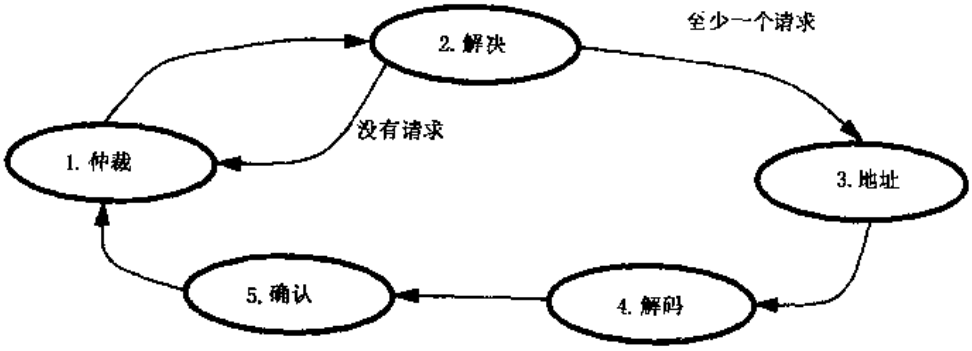


图 6-13 Powerpath-2 总线状态转换图。连到系统总线上的所有板的总线接口都按图所示同步地进行五态循环，这也是所有总线上的地址和数据事务的周期。但当总线进入空转态时，它只在状态 1 和 2 间循环

由于总线是事务拆分型的设计，地址总线和数据总线必须被独立地仲裁。在仲裁周期，使用 48 位地址线 + 命令线。这些线中的低 16 位被 16 块板使用（1 位/板）用于请求数据线，中间的 16 位用于地址线仲裁（对那些既要求地址线也要求数据线的事务，相关的位都被置高），高 16 位用于紧迫的或高优先级的请求。紧迫的请求要防止挨饿；例如，当一个处理器等待存取总线时超时。通过设置请求的优先程度，设计者可以灵活地考虑问题，使得对一些请求的服务因为性能因素可以比其他请求优先（例如读比写优先），但又肯定不会使任何请求者挨饿。

图 6-14 是图 6-8 的扩展，给出了过程的各周期（包括被驱动的各种总线）和它们的语义。在仲裁周期的末尾，所有的总线接口控制器捕捉到地址线 + 命令线的 48 位的状态，从而获悉了所有的总线请求。使用分布式体系结构方案，每个控制器能看到所有的总线请求，并在“消解”周期独立地处理同一个获胜者。虽然分布式体系结构消耗了更多的 ASIC 门资源，但它节省了集中控制器把获胜者经总线授权线传达给每一位的等待时间。 418

在“写址”周期，地址总线的获胜者用相关信息来驱动地址总线和命令总线。同时，数据总线的获胜者根据响应来驱动数据资源 ID 线。（数据资源 ID 是一个 3 位的全局标签，在读请求刚生成时，被赋予此标签。标签的使用请见 6.4.2 节）。

在“译码”周期，没有信号在地址总线上被驱动。每个总线接口槽各自决定如何对事务做出响应。例如，如果事务是回写，而当前内存系统没有足够的缓冲资源来接纳数据，那么在下个周期这个过程就会被否定回答（或拒绝），以待将来重试。另外，所有的接口槽都有准备提供适当的高速缓存一致性信息。

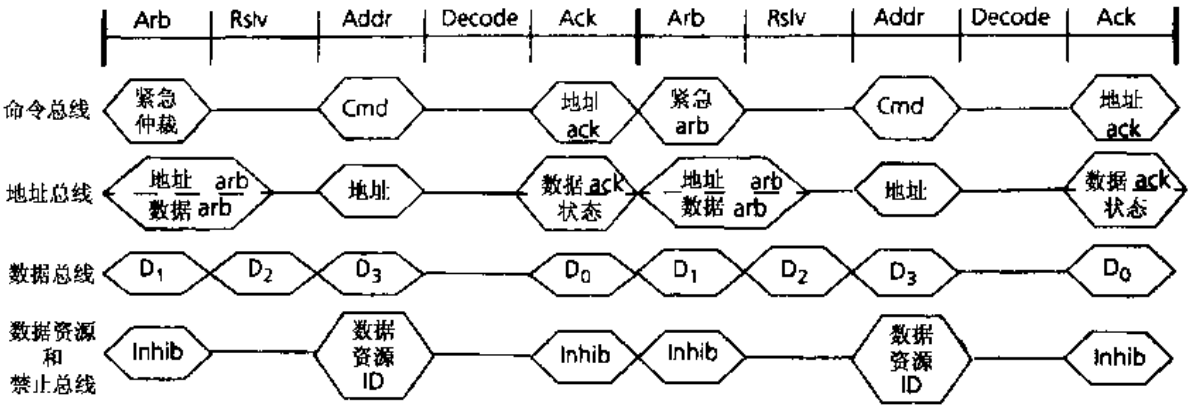


图 6-14 Powerpath-2 总线时序图。在仲裁周期，48 位地址总线＋命令总线传达了来自 16 条总线槽的关于数据事务、地址事务和紧迫事务的请求。每个总线接口用一个公共的算法独立地决定仲裁的结果。就地址请求而言，地址＋命令在“写址”周期被传送，请求可能会在“确认”周期被“否定回答”。类似地，就数据请求而言，相关标签（即数据资源 ID）会在“写址”周期被传送，请求可能会在“确认”周期被“否定回答”或数据在接着的 D0～D3 周期（D0 即“确认”周期）被传送

在“确认”周期，每个总线接口槽都对数据事务或地址事务做出响应。48 位地址总线＋命令总线如下使用：高 16 线表明响应槽上的设备是否因为缓存空间不足而拒绝地址总线事务，类似地，中间的 16 线针对数据总线事务。最低 16 线表明在数据总线上传输的块在高速缓存中的状态（在或不在）。这些线有助于决定将被载到请求的处理器中的数据块的状态（如排他或共享）。最后，假如某一个处理器通过这一周期没有完成侦听，它就会申明使用相关的禁止线（数据资源 ID 线在“接受”周期和“仲裁”周期加倍为抑制线），它会持续地声明直到它完成侦听。如果侦听表明了一个“干净”的高速缓存块，侦听节点就会释放禁止线，并允许请求节点接受内存响应。如果侦听节点表明了一个“脏”的高速缓存块，则节点重新仲裁数据总线，提供数据的最新版本，然后才释放禁止总线。

[419]

对于数据总线事务，一旦一条槽成为主槽，128 字节的高速缓存块数据将会在连续的 4 个周期内通过 256 位宽的数据通路。这 4 个周期序列起始于“确认”周期，终止于下一个循环的“地址”周期。由于 256 位宽的数据通路只用了 5 个周期中的 4 个，因此数据线的最高使用率为 80%。尽管如此，从某种意义上说，这是一种最好的办法，因为 Powerpath-2 总线使用的信号发送技术要求在不同的控制器驱动总线之间，要给与一个周期的回转时间。

6.5.2 SGI 处理器和内存子系统

在这个体系结构中，每个板有多个处理器。为了减少总线接口的开销，许多总线接口芯片被处理器间共享。图 6-15 给出了处理器板的高层结构。

处理器板使用 3 个不同类型的芯片来与总线接口，并支持高速缓存一致性。有一个单一的 A-芯片作为处理器与地址线的接口。它包含了用于分布式体系结构的逻辑仲裁，8 个条目的请求表用于存储当前总线上待处理的事务（见 6.4 节）以及其他的控制逻辑用来决定事务何时在总线上发布，怎样响应它们。它把总线上观察到的请求传到 CC-芯片（每个处理器一个），CC-芯片使用一个复制的标签集来决定内存块在本地高速缓存器中的存在性，并把它传回给 A-芯片。所有来自处理器中的请求流经 CC-芯片到达 A-芯片，然后由 A-芯片把它放到总线上。有 4 个位片 D-芯片与 256 位宽的数据总线接口，它们非常简单，并在处理器间共享；

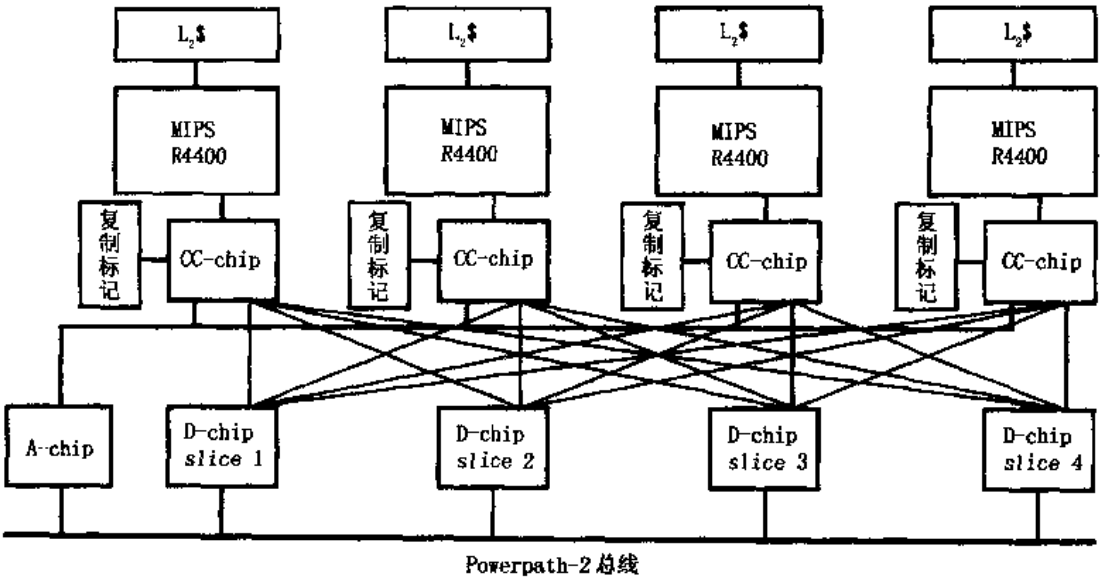


图 6-15　SGI Challenge 处理板的组织和芯片分割。为了支持 36 个处理器而使用较少的总线槽，每块板安排了 4 个处理器。为了保持一致性和与总线接口，每个处理器都有一个高速缓存一致性（CC）芯片，一个共享的 A-芯片来跟踪来自/去往所有芯片的请求以及和地址总线的接口，4 个共享的位片 D-芯片与 256 位宽数据线接口

它们提供有限的缓冲能力，并简单地把数据在和每个处理器（高速缓存）相联系的总线和处理器的 CC-芯片间传送。

Challenge 的主存子系统使用高速缓存把地址扇出到 576 位宽的内部 DRAM 总线，576 位包括 512 位的数据，64 位的错误校验码（ECC），实现一位纠错和两位检错。快速页模式的存取使得 128 字节的高速缓存块在两个内存周期被读入，数据缓冲器把响应传入 256 位宽的系统数据总线。当地址在总线上出现并过了 12 个总线周期（大约 250 ns）后，响应数据出现在数据总线上。一个单一的内存板能容纳 2 GB 的内存，支持一个两通路的交叉存取存储器系统，它的饱和系统总线带宽能达到 1.2 GBps。

假定主存系统占用的原始等待时间粗略计为 250 ns，我们不妨来看看二级缓存扑空时处理器的整个等待时间。在 Challenge 系统中，这个数字接近 1μs。使请求第一次出现在总线上花了大约 300 ns，其间包括处理器认识到一级缓存扑空、二级缓存扑空，并把请求经过 CC-芯片传到 A-芯片。完整的高速缓存数据块穿过总线到达 D-芯片另外花了大约 400 ns，其间包括 3 个总线周期等待请求过程进入“写址”阶段，12 个总线周期（大约 250 ns）存取主存，以及 5 个总线周期等待数据事务在总线上发送数据。最后，再经过 300 ns，数据流经 D-芯片、CC-芯片、64 位接口进入处理器芯片（对 128 个字节的高速缓存块 16 个周期）；在这儿，数据被载入主高速缓存，然后重启处理器流水线㊀㊁。

为了保持高速缓存一致性，SGI Challenge 缺省使用 Illinois MESI 协议，它也支持更新事务。高速缓存一致性协议和事务拆分型总线的相互作用见 6.4 节中的描述。

420 ~ 421

㊀ 注意，在往返路径上，内存请求都穿过非同步的边界，从而增加了双倍的同步器延迟，在每个方向上大约为 30 ns。非耦合的优点就是处理器和系统总线能运行不同的时钟频率，从而允许引入更高时钟频率的处理器而保持总线时钟频率。当然，这种开销就属于额外开销。

㊁ 更新的处理器，如 MIPS R10000，允许处理器在收到所需的关键字后，就能重起流程，而不用等待完整的高速缓存块全部到达。这种关键字重起机制降低了扑空延迟。

6.5.3 SGI I/O 子系统

为了支持多处理器提供的强大计算能力，在提供匹配的 I/O 能力时必须格外的小心。SGI Challenge 提供了可扩放的 I/O 性能，它允许多个 I/O 卡插在系统总线上，每个卡提供一个 320 MBps 的本地 HIO I/O 总线。多种不同的 ASIC 作为 I/O 总线和标准一致设备（如 Ethernet、VME、SCSI、HPPI），非标准一致设备（如 SGI Graphics）的接口。

图 6-16 给出了 SGI Challenge 的 PowerChannel-2 I/O 子系统的高层结构。总线是一种 64 位宽的多选地址/数据总线，与系统总线运行相同的时钟频率。它支持读事务的分离，每个设备可以有 4 个待完成事务。和系统总线不同，它使用集中式仲裁，节省等待时间。然而，仲裁是流水进行的，使总线带宽不致被浪费。由于 HIO 总线支持多个不同的事务长度（它不要求每个事务都处理一个完整的高速缓存块的数据），因此事务在请求时，要表明它们的长度，仲裁器使用此信息来保证总线得到更高效的利用。接到较窄 HIO 总线的 ASIC 比直接接口到较宽的系统总线上的 ASIC 相对便宜。另外需要接口到系统总线的公共机制通过这种方式被若干 ASIC 电路共享。

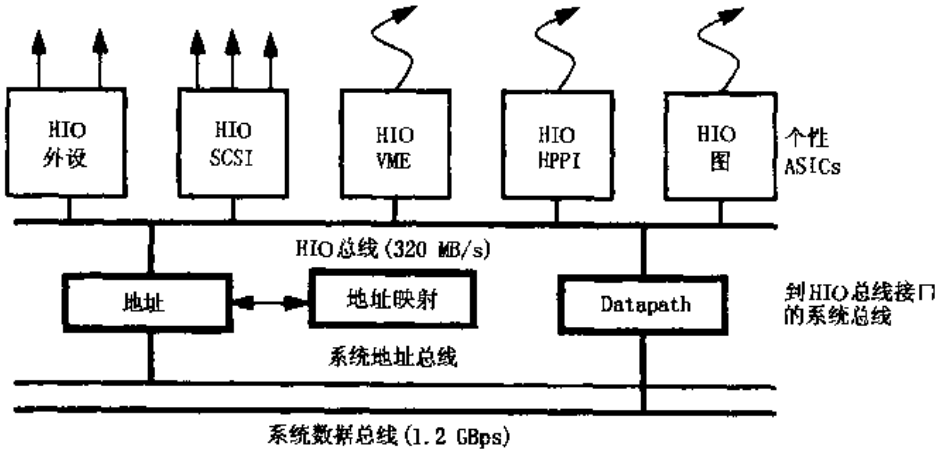


图 6-16 SGI Challenge PowerChannel-2 I/O 子系统的高层组织结构。每个 I/O 板提供了 Powerpath-2 系统总线和内部的具有最高带宽为 320 MBps 的 64 位宽的 HIO I/O 总线的接口。较窄的 HIO 总线降低了与它接口的开销，支持大量的专门 ASIC，它相应的支持标准总线和外部设备

HIO 接口芯片可以请求对 DMA 的读/写，使用全 40 位系统总线传输自/至系统内存的任何位置，使用系统接口中的映射资源提出对地址翻译的请求，要求中断处理器或响应处理器的 I/O（PIO）读。系统总线提供 DMA 读响应，对 I/O 设备提供地址翻译的结果，并传送到要读它们的 PIO。

在系统的其余部分（处理器板和主内存板）看来，在 I/O 板上的系统总线接口是一个清晰的接口；它工作起来就像一个处理器板。因而，当一个 DMA 读请求通过系统总线接口到达系统总线时，它就变成了一个 Powerpath-2 读，就像系统总线可能做的一样。类似地，当一个完全的高速缓存块 DMA 写发生时，它在总线上变成了一个特定的写块事务，使所有处理器高速缓存中的拷贝都变得无效（不仅仅更新内存）。我们需要一个特别的事务来进行处理，因为即使此块在一个处理器高速缓存中是脏块，我们也不想在这种情况下把它写回。

422

为了支持部分缓存块 DMA 写，必须小心地把数据一致地合并到内存中。为了支持这些

部分块 DMA 写，系统总线接口包含一个全相联的、含有 4 个存储块的高速缓存，来侦听 Powerpath-2 系统总线。高速缓存块的状态只有两种：作废或被修改。刚开始发出部分块 DMA 写时，此块被放进这个特殊的缓存，置以被修改状态，并使所有处理器高速缓存中的块拷贝都变得无效。以后的部分块 DMA 写如果命中此缓存，就不必再到系统总线，因而增强了系统总线的工作效率。在以下情况，修改的块状态将置为作废，并进入系统总线：1）有任何系统总线事务存取此块；2）有其他部分块 DMA 写过程将使此块替换出这个缓存；3）有任何 HIO 总线读事务存取此块。虽然块 DMA 读也可能会使用这 4 块高速缓存，但设计者感到部分块 DMA 读比较稀少，从优化中得到的收益很小。

在系统总线接口中的地址映射 RAM 为 I/O 设备提供了通常的地址翻译来访问主存。例如，经常要把小地址空间（如 VME-24 或 VME-32）映射到 Powerpath-2 总线的 40 位物理地址空间。有两种映射：一级映射和二级映射。一级映射仅仅返回地址映射 RAM 的 8 K 条目中的一条，每一条目对应物理内存中的 2 MB。在二级映射方案中，映射条目指向主存的页表。每一 4 K 页在二级表中有自己的表目，于是虚页能被正确地映射到物理页。注意 PIO 请求（来自处理器）到 I/O 设备有一个相似的翻译问题，它不使用地址映射 RAM 而直接通过专门的 ASIC 接口芯片来处理。

最后我们来看流控。所有从 I/O 接口到 Powerpath-2 系统总线的请求都是隐示地流控；即 HIO 接口如果没有给响应保留缓冲空间，就不会向系统总线发出读请求。类似地，HIO 仲裁器不会把 HIO 总线交给请求者，除非系统接口有空间来接纳事务。例外的是，PIO 能不经请求地从处理器到达 I/O 设备，它们需要进行显示地流控。 423

在 Challenge 系统中使用显示流控方案目的是使 PIO 看起来经过了请求才到达 HIO 接口 ASIC。在复位后，HIO 接口芯片（如 HIO-VME，HIO-HPPI）使用特定的称作 IncPIO 的请求将它们可得到的 PIO 缓冲空间传给系统总线接口。系统总线接口为每一个 HIO 设备维护一个独立的计数器。每次当一个 PIO 被传到一个特定的设备时，相关的计数减少。每次设备重试一个 PIO，它就发另一个 IncPIO 请求来增加计数器。如果系统总线接口收到一个关于设备的 PIO 请求却没有足够的可利用缓冲空间，它就会在系统总线上拒绝（NACK）这个请求，于是这个请求就必须在以后重新提出。

6.5.4 SGI Challenge 内存系统性能

各种级别的 SGI Challenge 内存系统的存取时间能用第 4 章介绍的简易的读微基准测试程序来得到。微基准测试程序方案测试出在一特定的跨距内，读给定大小的一组元素花费的平均存取时间。图 6-17 给出了在一定范围的大小和跨距内读操作的时间。每条曲线给出了对一定的大小受跨距影响的平均时间。小于 32 KB 的数组完全适用于一级缓存。二级缓存的访问大约用 75 ns，在 16 字节跨距处的拐点表明二级缓存和一级缓存之间的转变大小是 16 字节。第二个凸起部分给出了 TLB 扑空的额外代价大约是 140 ns，并揭示了页的大小大约是 8 KB。（你能想出为什么随着跨距的进一步增大每次扑空的时间反而回落了呢?）从 2 MB 数组开始，在 1 MB 的二级缓存中发生访问扑空，我们看到在二级缓存控制器、Powerpath 总线控制器和 DRAM 存取的共同作用下，一次访问时间大约为 1 150 ns。与前面的讨论相符，实现从请求到回应共 13 个周期的最小的总线协议使得这个时间略低于 300 ns。TLB 扑空在 1 150 ns 中增加了大约 200 ns。一个简易的乒乓微基准测试程序（在此方案中，有一对节点

都受某一标记控制而不断运转，当轮到自己时，就设置标记把信号发给对方）给出了6.2 μs的往返时间。

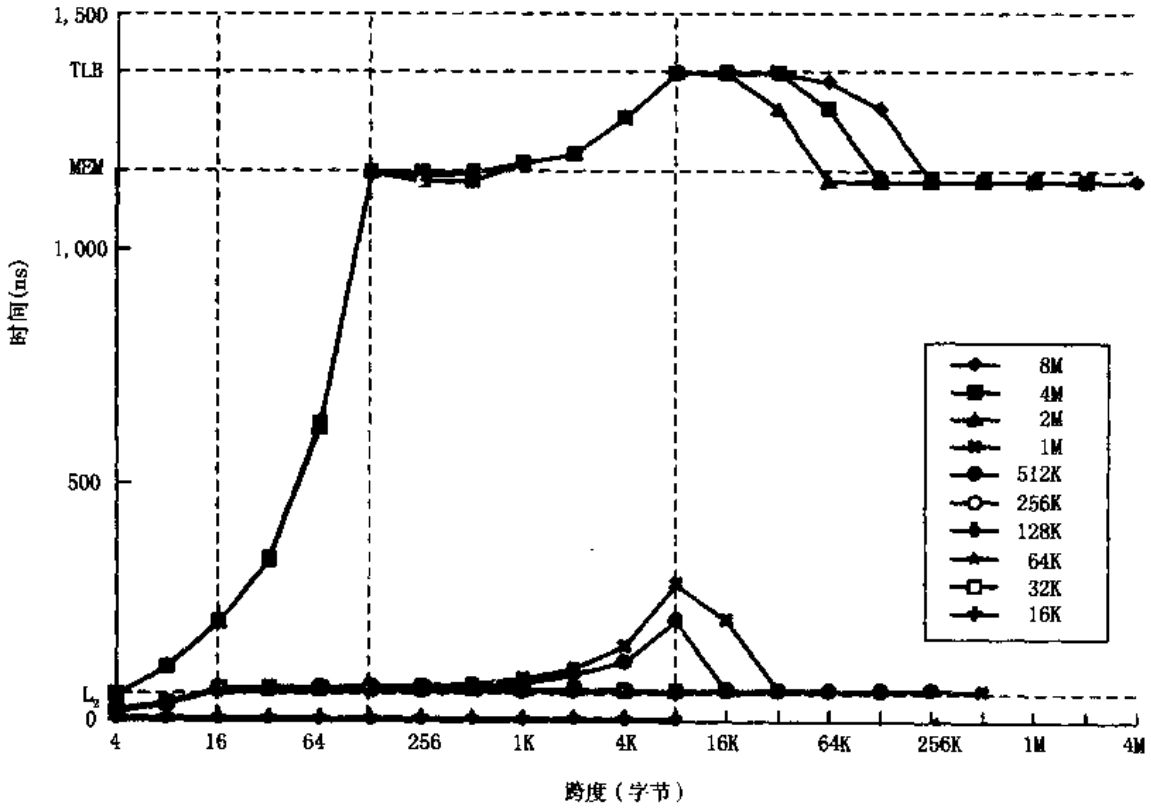


图 6-17 SGI Challenge 读微基准测试程序的结果。每条曲线都针对图表符号中所示的一组有相同尺寸的元素。数组尺寸从 32 K 到 256 K 的数据点非常接近，很不容易区分

6.5.5 Sun Gigaplane 系统总线

Sun Gigaplane 也是非多选的事务拆分型总线，有 256 位数据线，41 位物理地址线，但采用 83.5 MHz 的时钟频率。它是一种中心平面设计，一条总线贯穿和接口汇集，允许双面插板，而不是单面的底板。总线全长 18 英寸，故每一边能插入 8 个板，且在每两个板间留有两英寸的散热空间，连接口间有 1 英寸的空间。它与 Powerpath-2 总线最大的不同在于，它支持多达 112 个等待处理的事务。每个板高达 7 个，因此它是为能支持多个等待事务的设备而设计的，例如免锁定的高速缓存。这种电气上和机制上的设计允许处理模块和 I/O 模块的热插拔。

总线共包括 388 个信号：256 个数据、32 个校验码、43 个地址（带有奇偶校验）、7 个 ID 标记、18 个仲裁以及许多的配置信号。电气上的设计实现了数据传送间没有死循环的回转。它把重点放在了实现尽可能少的操作延迟上（如图 6-18 所示），它的协议与 SGI Challenge 有很大的不同。一个著名的基于冲突的推测仲裁技术被用于减少总线仲裁的开销。当请求者对地址总线进行仲裁时，如果总线自从前一个周期后尚未被调度使用，那么在仲裁周期，它就试探性地把自己的请求驱动到地址总线上。如果在那个周期没有其他的请求者，它就赢得了仲裁并将地址送出去，于是它继续余下的事务。如果发生了请求冲突，赢得仲裁的请求者只简单地把地址在下个周期重新驱动一遍，和常规的仲裁一样。

424 ~ 425

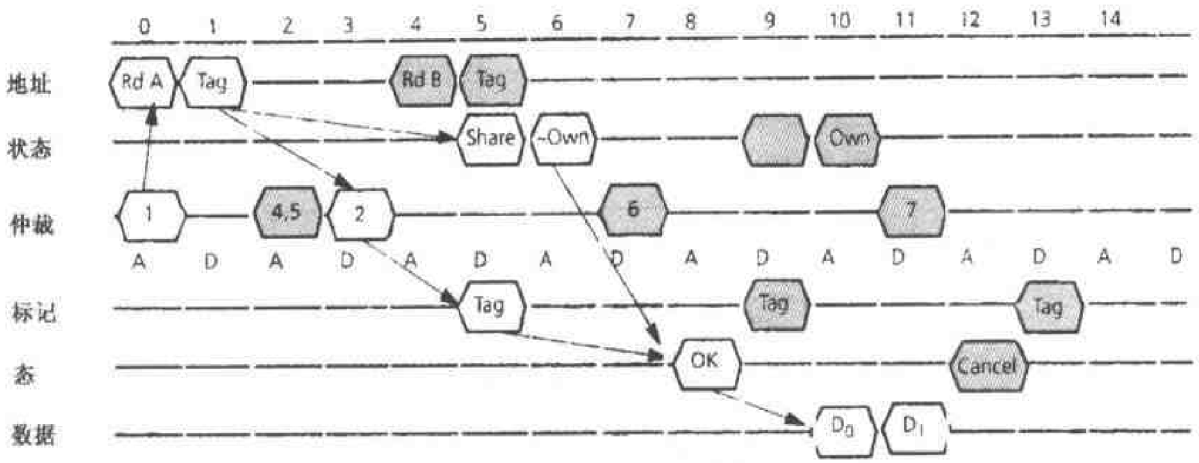


图 6-18 Sun Gigaplane 对采用快速地址仲裁的 BusRd 的信号时序。正如所示，在流水线时隙上，有两个 BusRd 事务，一个不带阴影，另一个有阴影。水平粗线显示了总线的不同组件，垂直点线划分了周期。总线仲裁组件下的“A”和“D”表明了地址仲裁周期和数据仲裁周期。箭头显示了第一个 BusRd 事务的路径。板 1 初始化了一个采用快速仲裁的读事务（地址在同一个周期被成功地驱动），然后被主板 2 响应。侦听结果表明没有高速缓存容纳此块，于是主板把结果驱动到数据线上。对于第二个 BusRd 事务，板 4 和板 5 在地址总线仲裁期间发生冲突，板 4 获胜并初始化了一个读事务。主板 6 对数据总线进行仲裁，由于侦听结果表明有高速缓存含有此块，故取消了响应。最后板 7 上的此高速缓存以数据做出响应。板 5 的重试事务没有显示

426

有关请求的 7 位标记在紧跟着“写址”周期后的周期内被放到地址总线上（如图 6-18 所示）。侦听的状态与地址阶段相联，与数据阶段无关。在“写址”周期后，过了 5 个周期，所有的板都在状态总线上声明自己的侦听信号（共享、被拥有、被映射、忽略）。在此期间，响应内存地址（主板）的板会在“写址”周期 3 个周期后，侦听结果出来之前，请求数据总线。DRAM 访问也能被试探性地启动。当主板得到仲裁后，它必须在两个周期后给出标记总线的电平，告知所有设备处理数据正在到来。在驱动标记 3 个周期后，即驱动数据的前两个周期，主板驱动了一个状态信号，用于表明如果某个高速缓存拥有这个块（在侦听状态时得到），数据传送就会被取消。拥有者通过仲裁数据总线把数据放到总线上，驱动标记，启动数据。图 6-18 给出了另一个读过程（灰色），它在仲裁时遇到冲突，于是地址由常规的槽提供。对过程的侦听表明拥有者是一个高速缓存，故主板取消了数据传送。然后，这个变速缓存仲裁数据总线，驱动相应的数据。

像 SGI Challenge 一样，出现在地址总线上的 BusRdX 请求发出作废命令，此命令被高速缓存子系统以先进先出的方式进行处理。这样，也不必要求对完成“作废”工作的显示的认可。为了保持连续的一致性，仍有必要在允许写处理器在写之后继续从事内存操作之前获得对地址总线的仲裁。[⊖]

6.5.6 Sun 处理器和内存系统

如图 6-19 所示，在 Sun Enterprise 中，每个处理板有两个处理器（每个处理器有一个外部的二级缓存）和两个内存模块（通过交叉开关相连）。UltraSparc 内的数据线被缓冲来驱动称为 UPA（通用端口体系结构）的内部总线（具有 1.3 GBps 的内部带宽）。到内存的通路很

⊖ Sparc V9 规范弱化了同一性模型，以使处理器能使用写缓存，我们将在第 9 章更深入地讨论这个问题。

宽，读一个完全的64字节的高速缓存块只要一个内存周期或两个总线周期。地址控制器使UPA协议和Gigaplane协议相匹配，并实行高速缓存一致性协议，提供缓冲，跟踪潜在的大量的待完成的事务。它为二级缓存保有一组备份的标记，称为D-标记。为了保证高速缓存一致性，即使是从处理器到本地内存模块的访问也要经过地址控制器。

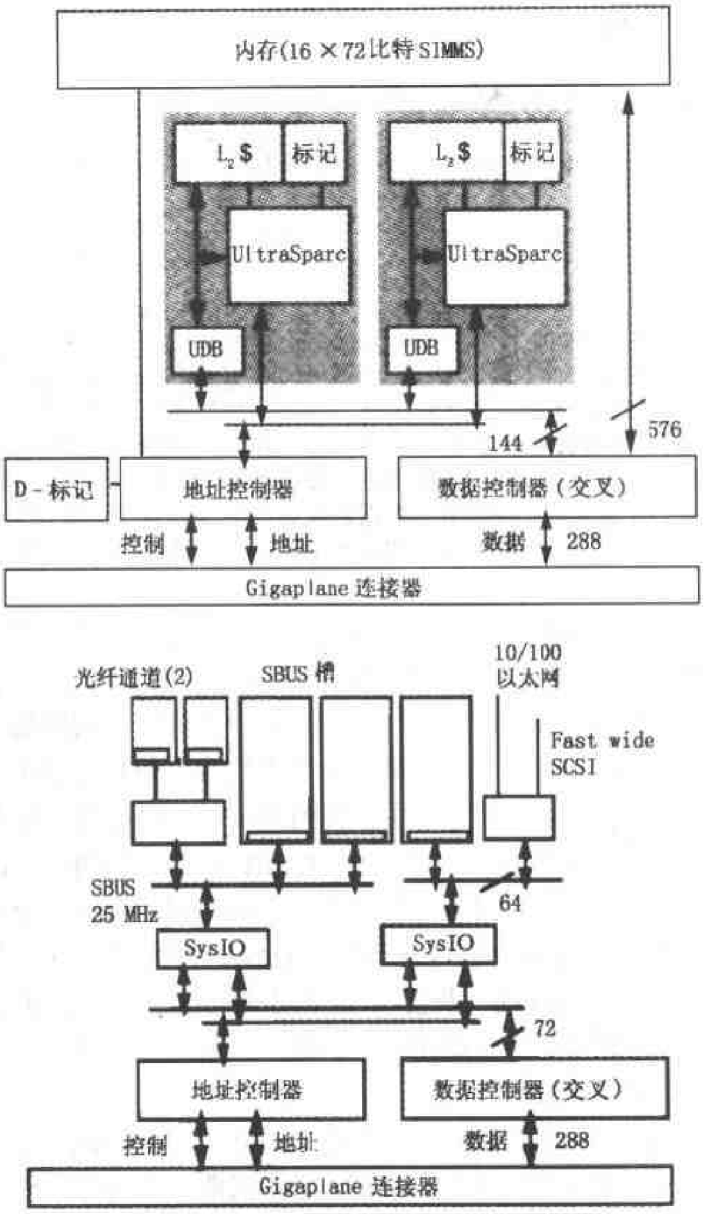


图6-19 Sun Enterprise处理板和I/O板的组织结构。处理板（上）包括两个UltraSparc模块（带有在内部总线上的二级缓存）和两个宽内存模块（通过两个ASIC与系统总线接口）。地址控制器使两个总线协议相匹配，并实现了高速缓存一致性协议。数据控制器本质上就是一个交叉开关。I/O板（下）使用两个相同的ASIC与两个I/O控制器接口。SysIO ASIC看起来就像一个遵从一致性协议的单块高速缓存。另一方面，它们支持独立的I/O总线，并与FiberChannel、Ethernet、SCSI接口

虽然UltraSparc在二级缓存里实现了5态的MOESI协议，D-标记只使用了3个状态：被拥有、共享、作废。它们基本上综合了在Gigaplane级上一致处理的状态。特别地，地址控制器需要知道二级缓存是否有一个块的拷贝以及此块是否是一个排他的拷贝。它不必知道这个块是否是干净的或脏的。例如，在进行BusRd时，如果存储块在二级缓存中的状态是如下

427
~
428

三个之一：已修改、被拥有（最近一次修改后已被发送过）、排他的（不能共享读和不能修改），块需要被送到总线上；因而，D-标记代表的仅仅是被拥有状态。这能使当 UltraSparc 把块由排他的改为已修改时，不用告知地址控制器。当发生由作废、被拥有、共享到已修改的状态转变时，需要告知 UltraSparc，以便它初始化一个总线事务。

6.5.7 Sun I/O 子系统

一个 Enterprise I/O 板与处理板使用一样的总线接口 ASIC，但内部总线只有一半宽，并且没有内存通路。在外看来，I/O 板就像处理板一样只做高速缓存块级的事务，为了简化主系统总线的设计。SysIO ASIC 实现了一个单块高速缓存代表 I/O 设备，并遵循一致性协议。在内支持两个独立的 64 位 25 MHz 的 SBUS。其中一个支持两个专用的 FiberChannel 模块，它提供大磁盘存储阵列的冗余的、高带宽的互连。另一个支持专用以太网和快速的宽 SCSI 连接。另外，3 个 SBUS 接口卡能被插入这两条总线以支持仲裁外设，包括 622 MBps ATM 接口。I/O 带宽、与外设的连接、I/O 子系统的开销决定了 I/O 卡的数目。

6.5.8 Sun Enterprise 内存系统性能

图6-20给出了用读微基准测试得到的Sun Enterprise的各种等级的存取时间。小于或等

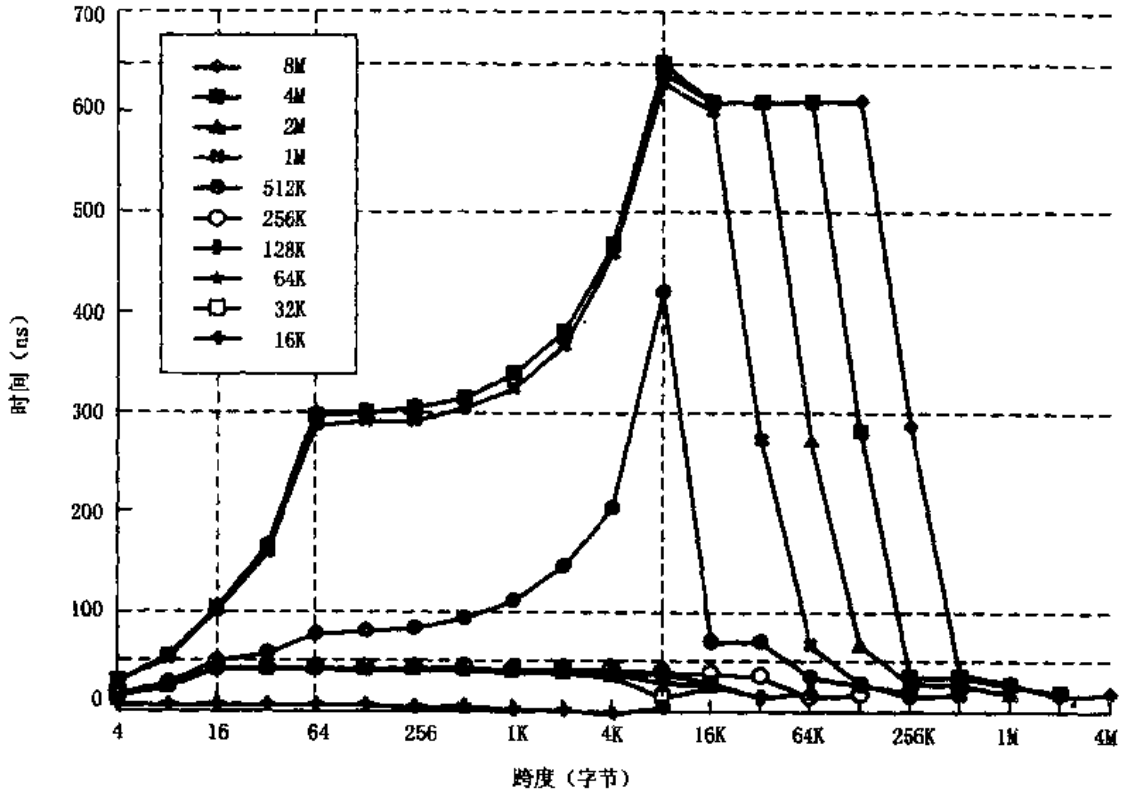


图 6-20 Sun Enterprise 在读操作微基准程序上的测试结果。每条曲线对应不同的规模，具体见图例所示

于 16 KB 的数组完全适用于一级缓存。二级缓存的存取时间大致是 40 ns，拐点表明二级缓存和一级缓存之间的转变大小是 16 字节。对于存取二级缓存未中的 1 MB 的数组，我们看到由于二级缓存控制器，总线协议和 DRAM 存取共同作用的结果，存取时间大约为 300 ns。

83.5 MHz 下共 11 个周期的最小的总线协议占用了这段时间内的 130 ns。由于机器有软件 TLB 管理器，所以 TLB 未中增加了大约 340 ns 未中的代价。一个简易的乒乓微基准测试（在此方案中，有一对节点，都受某一标记控制而不断运转，当轮到自己时，就设置标记发给对方信号）给出了 1.7 μs 的往返时间，相当于访问内存 5 次。

6.5.9 应用程序性能

现在我们已对机器和它们的微基准测试性能有了一定的理解，接下来检验一下对于并行应用程序性能的提高。本书不说明商用机器绝对的应用性能，而着重在并行性带来的性能提高。以 SGI Challenge 为例解，让我们先看看应用程序的加速比，接着是性能随规模的扩放情况。

1. 应用程序的加速比

图 6-21 给出了 6 个并行程序（在两组不同数据规模上）得到的加速比。我们可以看到，除了基数排序内核外，多数程序的加速比都不错。通过分析排序执行时间的细节，能够发现绝大部分时间都花在等待数据访问上。在排序算法的交换阶段，数据流量和一致性流量都很大，使共享总线不堪重负，所导致的访问冲突极大地破坏了性能。冲突在数据访问时间内导致了严重的负载不平衡，从而时间花在了等待全局栅障上，即使繁忙时间被很好地平衡。不幸的是，增加问题的规模不能对冲突有很大地减轻，由于通讯和计算的比以及所产生的带宽要求，使得交换问题独立于数据规模的大小（见第 4 章 4.4.1 节）。图示的基数为 256，它要得到最好的性能与两个问题大小因素有关，不仅仅是处理器的数目。Barnes-Hut、Raytrace 和 Radiosity 即使在相对小的输入规模情况下也加速得非常好。LU 也是如此，它在 16 个处理器下对较小规格问题的瓶颈主要是由于矩阵因子分解过程造成的负载不平衡。最后，Ocean 的瓶颈既是由高的通讯和计算比，也由一些分区具有较少的相邻者造成的不平衡产生的，两个问题可以由运行大的数据集来减轻。

429
~
430

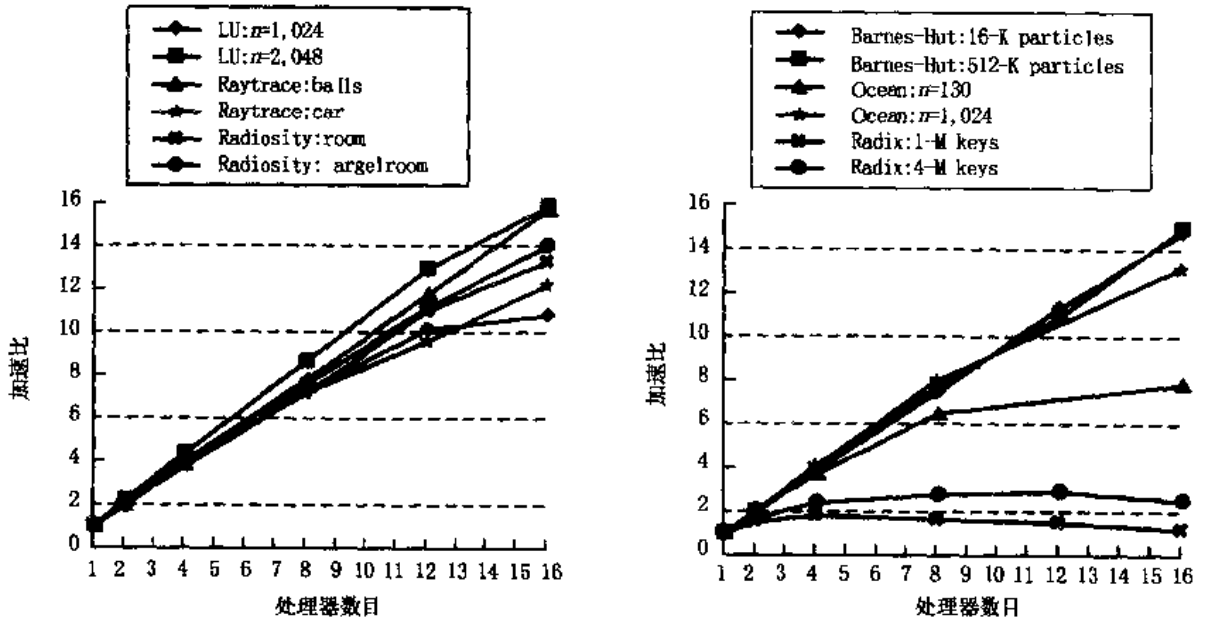


图 6-21　SGI Challenge 上 6 个并行应用程序的加速比。成组的 LU 因子分解的块尺寸是 32×32

2. 性能随规模的扩放

现在让我们考察一下规模扩放对一些应用程序的影响。根据第 4 章的讨论，我们看看不同的规模扩放模型下得到的加速比，以及完成的工作和使用的数据集大小的变化。图 6-22 给出了对于 Barnes-Hut 和 Ocean 应用程序的结果。其中 Naive TC（时间受限）或 Naive MC（内存受限）扩放指的是仅仅改变决定数据集大小的参数（星体或网格点的数目 n），而不改

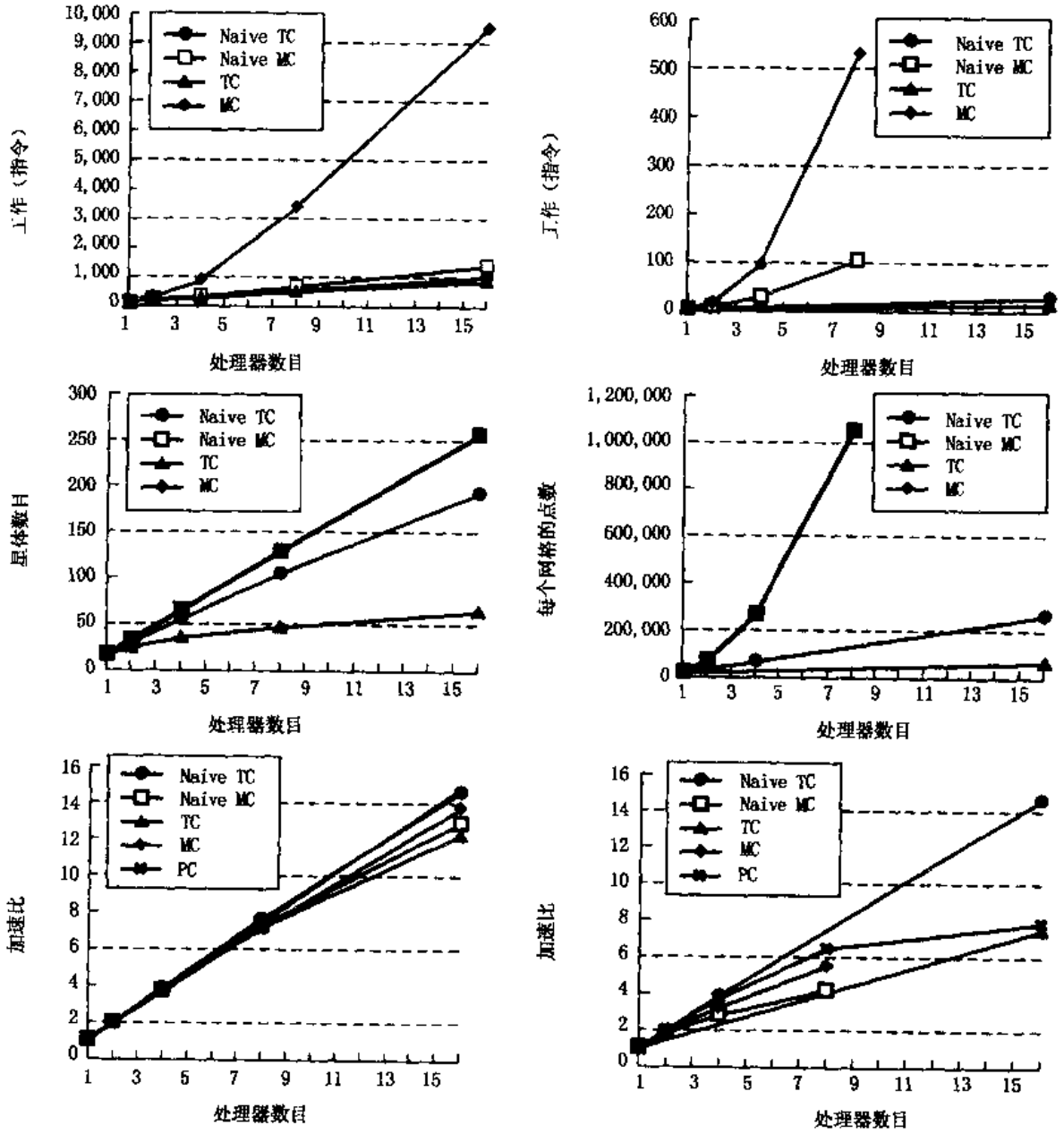


图 6-22 SGI Challenge 上 Barnes-Hut（左）和 Ocean（右）应用程序的缩放结果。图给出了在不同的扩放模型下完成的工作、数据集的大小（测量主体或网格点的数目）、加速比。PC、TC 和 MC 分别指的是问题约束、时间约束、内存约束的缩放。对 Barnes-Hut 来说，问题尺寸的基准是 16 K 星体；对 Ocean 来说是 130×130 网格。最上的一组图显示出对于这两个应用程序在 Realistic MC 缩放下需要解决的工作都增长的非常快。中间的一组图显示出在 MC 下或 Naive TC 下，数据集大小的增长比在 realistic TC 下增长的更快。缩放模型在加速比方面对于 Ocean 的影响比 Barnes-Hut 大得多，主要是因为在 Ocean 里通信和计算比更强烈地依靠数据集大小和处理器数目

变其他应用程序参数（精确性或时段数目）。很明显，在 Realistic MC 扩放下，对于这两个应用程序要完成的工作都比线性地改变处理器的数目快得多，于是并行执行时间增长得非常快。能在 TC 扩放下被模拟的星体或网格点的数目比在 MC 下增长慢得多，也比 Naive TC 扩放下的增长慢得多，它仅仅只是一个应用程序参数的改变。扩放其他应用程序参数使得完成的工作和执行时间增长，使得增加 n 更加困难。

431 ~ 432

在不同的扩放模型下加速比的测量和第 4 章描述的一样。现考虑 Barnes-Hut 星系的模拟，它的加速比对于各种扩放模型下的机器尺寸都很好。通过检验主要的性能因素可以解释差别。在引力计算方面的通信和计算比主要依靠星体的数目。另外一个影响性能的重要因素为两个量的比，即在引力计算阶段的工作量和在建立计算树阶段的工作量的比，前者容易通过并行得到好的加速比，后者难以并行。这个比率倾向于在作用力计算方面有更高的精度，即更小的 θ。然而，更小的 θ（从更小地程度上说，是更大的 n）意味着增长每个处理器工作集的尺寸（Singh，Hennessy，and Gupta 1993）。重要的工作集即使在扩放下也可能继续适用大的二级缓存，但是在一个单处理器上，改变 θ 的扩放后的问题，相对于基准问题而言，也许会降低一级缓存的性能。这些因素就可以解释为什么 Naive TC 比 Realistic TC 加速得更好：工作集的行为越好，通信和计算比就越好（因为当 θ 和 Δt 不变时，n 增长得更快）。

Ocean 的加速比在不同模型下很不相同。在这儿，主要的控制因素还是通信和计算比、工作集大小、在不同情况下的时间花费等。然而，所有的效果更强烈地依靠网格的尺寸（与处理器数目有关）。在 MC 扩放下，通信和计算比不随使用的处理器数目而变化，于是我们希望得到更好的加速比。但随着处理器的变化，出现了两个效果。第一，随着处理器的网格划分在地址空间上的距离增大，跨网格的冲突扑空增加。第二，在求解器中，更多的时间花在多重网格的较高层次上，降低了并行性能。当精确性和时段间隔被细化后，后一个影响被减轻了（至少有益于并行加速比），于是实际的 MC 比简单 MC 稍好一点。在简单 TC 下，网格尺寸的增长速度不足以引起主要的冲突问题，但足以使通信和计算比不发生明显的增长，故加速比挺好。实际的 TC 使网格尺寸增长较慢，从而加大了通信和计算比，导致较低的加速比。很明显，许多效果在决定扩放下的并行性能时扮演了重要的角色，并且对某个应用程序最合适的扩放模型的选择影响对机器的评估。

6.6 高速缓存一致性的扩充

在这两章里描述的基于侦听获得缓存一致性的技术可以在许多方向上扩充。这一节考察几个重要的情况：利用共享缓存的向下扩展、带有虚拟索引缓存和 TLB 的功能性扩放和利用非总线互连的向上扩放。

433

6.6.1 共享缓存的设计

将处理器组织在一起，让它们共享存储层次结构的一个层次（例如一级或二级缓存），对于共享存储多处理器来说是一个有潜在吸引力的考虑。尤其是考虑到封装时，若能将多个处理器放在同一个芯片上，这种想法会更有价值。和处理器在各自存储层次中有自己的缓存相比，将处理器集中起来有多种潜在的好处。这些好处，如同后面将要讨论的弊病一样，当共享任何一层存储时都会体现出来，但在处理器间共享的一级缓存的情形表现得最突出。在一层中的一组处理器间共享一个高速缓存的好处归纳为如下几点：

- 它消除了在这一级做缓存一致性协议的必要。特别是，如果一级缓存由所有处理器共享，那么就没有多份缓存块的存在，从而也就根本没有了一致性问题。
- 它降低了处理器组内部通信的时延。处理器之间通信的时延和它们通信时相遇在存储器层次的哪一层有关。当共享一级缓存时，通信时延可能只要 2 ~ 10 个处理器时钟周期，而在主存相遇的情形，时延要大许多倍（见关于 Challenge 和 Enterprise 的案例分析）。降低了时延就可能使不同处理器上执行的任务能在更细的粒度上共享数据。
- 一旦处理器在访问一个数据时发生扑空，而后将它带到共享缓存来，在同一组中的其他处理器当需要该数据时可能会发现它已经在缓存了，于是不会形成新的扑空。这称为跨处理器的数据预取。对于私有缓存来说，每个处理器会引发分别的扑空。扑空的数量减少了，也就降低了对下级存储和互连带宽的要求。
- 它允许更有效地利用较大的缓存块。即使一个组中的不同处理器访问一个缓存块的不同字，空间局部性也被开发出来。除此以外，由于在这一层次没有一致性协议，也就没有了伪共享问题。例如，考虑两个处理器 P_1，P_2 相继对一个大数组的各个元素依次交替进行写操作，考虑一级共享缓存和私有一级缓存的差别。
- 在一个组中的工作集（代码或数据）可能重叠很多，如果每个共享缓存都要保持其处理器的整个工作集，这就允许共享缓存的大小比私有缓存之和小。缓存容量的减小对单芯片的多处理器是特别有意义的，因为在这样的场合硅面积是一个重要的限制。 434
- 它提高了缓存硬件的利用率。共享缓存不会因为一个处理器停滞就闲起来，而是可以为组中的其他处理器服务。
- 成组的做法和层次封装技术适相应得很好（机柜，板，多芯片模块和芯片），且使我们能够有效地利用新出现的封装技术来获得更高的计算密度（每单元面积的计算能力）。

共享一级缓存时，处理器如图6-23那样通过一个交换机连接到一个共享缓存上。这个交

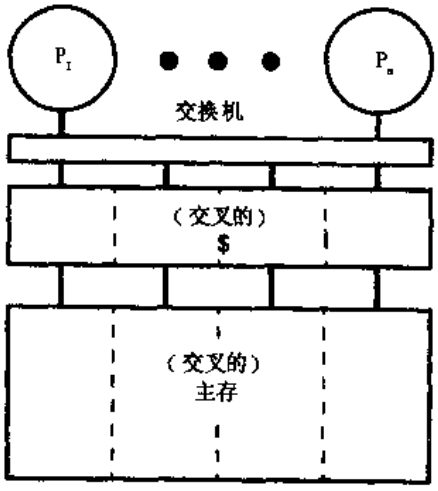


图 6-23　多处理器共享一级缓存的一般性结构。互连设置在处理器和一级缓存之间。在这种设计中，共享的缓存直接由主存支持，缓存和主存都可以分别交叉来提供高带宽。当然，由于共享缓存或交换机带宽的限制，这种设计只适合于较少的处理器数。另外较大的交换机会有较大的时延

换机可能是一个总线，但更可能是一个交叉开关，让从不同处理器来的缓存访问能够并行处理。类似地，为了支持多处理器带来的带宽要求，缓存和主存都按交叉方案组织。这种共享缓存系统的一个早期例子是设计于 20 世纪 80 年代初期的 Alliant FX-8 机。该系统可以最多含有 8 个特定的处理器，每个处理器是 68020 指令集的一个流水实现，并增强有向量指令，时钟周期 170 ns。所有 8 个处理器用一个交叉开关连接到一个 512 K 的 4 路交叉缓存。这个缓

存的块为32字节，回写型，直接映像，免锁定，允许每个处理器有两个待完成扑空。缓存到处理器带宽是每指令周期8个64位的字。

早期还有一种共享缓存的不同用法，出现在Encore Multimax中，和FX-8是同时代的产品。Multimax是一种侦听缓存一致性多处理器，但每个缓存支持两个处理器（在一对处理器之间不需要一致性）。当时Encore的动机是要降低侦听硬件的费用，努力在非常慢的多CPI的处理器条件下提高缓存的利用率。

435

当前，人们在单片多处理器的背景下研究一级缓存的共享问题，4～8个处理器共享一个片载一级缓存。这样的系统本身就是一个多处理器，也可以用作更大系统的基本模块，这样的大系统要在单片共享缓存组之间维持一致性。随着工艺的进步，一个芯片上的三极管数以数千万或亿计，这种方式越来越具有吸引力。由于在一个芯片内的处理器之间通信和同步开销是相当低的，用这样的芯片做成的工作站将能够对细粒度或粗粒度并行性都给出很高的性能。问题是和用更多的三极管做出更复杂的处理器相比，这种做法是否更加有效。

不尽人意的是，共享缓存也有若干弱点，带来若干挑战：

- 共享缓存需要满足来自多个处理器的带宽要求，这限制了组的大小。对一级缓存来说这个问题特别突出，因此只可能有很少的处理器。如何能提供所需的带宽是单片多处理器系统设计中最大的挑战。
- 由于互连的关系，对共享缓存的命中时延通常要高于私有缓存。对共享一级缓存来说，在处理器和缓存之间放一个交换机意味着机器周期拖长，或者在处理器流水线中装入指令中要增加额外的延迟时隙。前者造成的速度慢是显然的。虽然编译器有可能在装入延迟时隙中调度一些不相关的指令，成功与否取决于具体的应用程序。特别是对于那些没有许多指令级并行性的程序，减速是不可避免的。共享缓存引起的竞争使本已增加的命中时延更加严重，相应地，扑空时延也会由于共享增加。
- 鉴于前面的原因，设计一个有效的共享缓存要比设计私有缓存复杂许多。
- 虽然一个共享的缓存不需要如相应私有缓存之和那么大，但和单个私有缓存相比，它仍然要大得多，因此就会慢得多。对一级缓存来说，这就可能会加长机器时钟周期，或者要多个处理器周期才能完成缓存访问。
- 重叠工作集（或者是建设性干扰）的反面是共享缓存的性能受到伤害，由于跨越源于不同处理器的引用流的缓存冲突（破坏性干扰）。当一个共享缓存多处理器用来执行没有数据共享的负载（例如，并行编译或者数据库和事务处理工作负载），缓存中在不同处理器所需的数据集之间的干扰可能严重伤害性能。在科学计算中，性能是第一位的，许多程序试图管理它们对每个处理器缓存的使用，使得它们访问的数组在缓存中不发生干扰。所有这些程序员或编译的努力在共享缓存系统中很容易就白费了。同私有缓存相比，共享缓存可能要求更高的相联度，这也可能增加它们的访问时间。

436

- 最后，在目前情况来看，共享一级缓存不符合当前用商品微处理器技术建造成本效益合算的并行机器的趋势。

由于许多微处理器已经对第一级缓存提供侦听支持，一种有吸引力的做法可能是让处理器有私有的一级缓存，而在第二级缓存共享。这种做法“软化”了共享一级缓存的优缺点，可能总体上是一种不错的折中。这个共享缓存可能比较大，以减少破坏性干扰。在实践中，

封装方面的考虑对于共享缓存的决定也有很大影响。

6.6.2 虚拟标引缓存的一致性

回顾在单处理器系统结构中在物理的和虚拟的索引缓存之间的权衡，即，用物理地址还是虚拟地址对缓存索引。对于物理索引的一级缓存来说，允许缓存索引和地址转换并行进行，就要求缓存有很小或者很高的相联度。这保证了如果使用页染色方式，那些在转换下不改变的位——$\log_2$（Page_size）位或者更少些——足以索引缓存（Hennessy and Patterson 1996）。随着片载一级缓存变大，虚拟标引缓存变得越来越有吸引力。然而，这也有它们自己的问题。首先，不同的处理器可能用同样的虚拟地址来引用不同地址空间中不相关的数据。这可以通过在切换上下文时清除整个缓存来处理，或者通过使地址空间标识符（ASID）标记和缓存块相联，除虚拟地址标记外。对缓存一致性更严重的问题是同义：不同的虚拟页，来自相同或不同的进程，为了共享指向同一个物理页。对于虚拟地址缓存来说，相同的（共享的）物理存储块能够读到两个不同的缓存块、不同的索引。如我们所知，这对单处理器是一个问题，它也扩展到多处理器的缓存一致性，如果一个处理器写一个存储块，用一个虚拟地址别名表示，另一个读他，用不同别名的表示，那么如果简单地将虚拟地址放到总线上，侦听它们，对这个共享物理页的写就不会为后来的处理器可见。将虚拟地址放到总线上还有另外一个问题：它要求 I/O 设备和存储器做从虚拟地址到物理地址的变换，由于它们处理的是物理地址。然而，将物理地址放到总线上似乎要求相反的变换在一次侦听期间查找虚拟标引的缓存，不管怎样它自己解决不了由不同表示产生的别名一致性问题。

别名问题可以通过软件方法，限制其使用来避免。例如，可以让别名有同样的页颜色，即，如果这些多于 $\log_2$（Page_size）位，就让用于索引缓存的部分是相同的。另一个方法是，当引用相同页的时候，可以要求进程用相同的虚拟地址，如同在 SPUR 研究项目中那样（Hill et al. 1986）。

437

人们也提出过复杂的缓存设计来从硬件的角度解决不同表示的问题（Goodman 1987）。其思想是处理器用虚拟地址来进行缓存查询，但将物理地址放到总线上让其他缓存和设备来侦听。这要求提供能实现下述功能的机制：1）如果查找虚拟地址失败了，能用物理地址来查询缓存（届时物理地址可用）或者如果检测到那个块曾被由一个别名访问带到了缓存；2）保证相同的物理块在两个不同的虚拟地址下不会同时出现在相同的缓存中；3）将一个侦听的物理地址转换到一个有效的虚拟地址来查询侦听缓存。实现这些目标的一个办法是让缓存对它们的块维护虚拟和物理两种标签（和状态），分别由虚拟和物理地址来标引，让一个块的两个标签相互指向（即分别存放相应的物理地址和虚拟地址；如图 6-24 所示）。缓存数据阵列本身用虚拟标引（或者是物理标记项中的指针，在侦听的情形，它们是一样的）。让我们看看在一个高层中这种组织是如何提供所需机制的。

处理器用它的虚拟地址查询缓存，同时如果需要的话，虚拟地址到物理地址的转换由存储管理单元来完成。如果用虚拟地址的查询成功，一切都好。如果它失败了，转换得到的物理地址就用来查找物理标签；如果这命中了，就可以通过物理标签中的指针来找到这一块。如下所述，这就达到了第一个目标。一个虚拟的扑空但物理的命中检测了一个别名的可能性，因为物理块可能已被通过不同的虚拟地址带进来了。在一个直接映像缓存中，它一定是通过一个不同的虚拟地址带到了缓存，因此为简单起见让我们假设直接映像缓存。包含在物

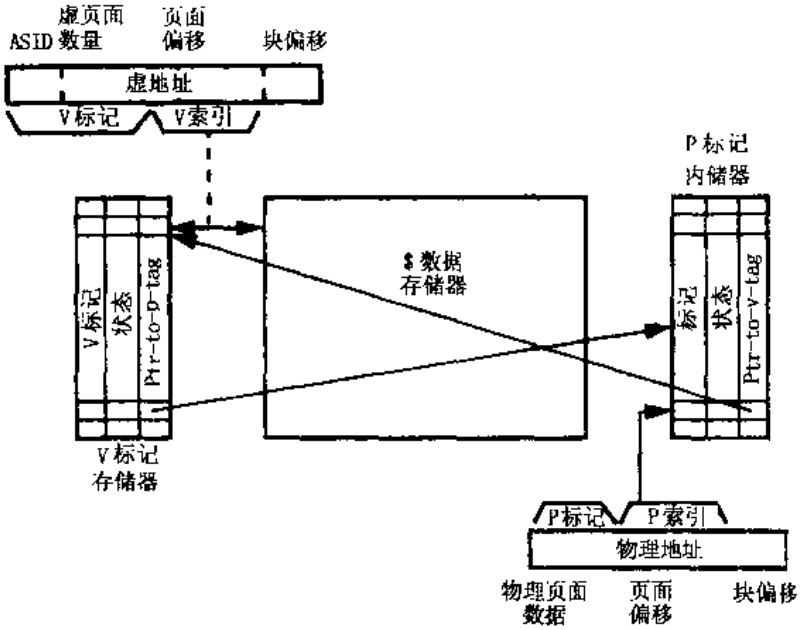


图 6-24 一种双标记的虚拟寻址的缓存的组织。左边的 V 标记存储器服务于 CPU，且由虚拟地址索引。右边的 P 标记存储器用于总线侦听，由物理地址索引。存储块的内容根据 V 标记的索引存放，相应的 P 标记和 V 标记条目相互指向，以处理对缓存的更新

理标签中的指针（别名虚拟标引）现在所指的缓存块不同于当前虚拟标引。我们需要使当前的虚拟标引指向这个物理数据，让虚拟地址和物理地址统一，以消除别名。当前被别名虚拟标引的物理块，被拷贝过来后，取代当前虚拟标引的块（如果必要，它会被回写），因此对当前虚拟标引的引用将从此立刻命中。在别名虚拟标引的块被置成作废或者不可访问，因此数据现在只能通过当前虚拟标引来访问（或者在侦听情形，借助于物理标记中的指针通过物理地址），但不能通过别名虚拟标引。对这个别名后续的访问将在它的虚拟地址查询上扑空，并且必须完成同样的过程。因此，在任何给定时刻，给定的物理块只是在缓存中一个（虚拟标引的）单元有效，于是实现了第二个目标。注意，如果虚拟地址和物理地址查询都失败了（真的缓存扑空），可能需要多到两次的回写。带入缓存的新块将被放在由当前虚拟（不是物理）地址确定的标引上，虚拟和物理的标记和状态将要适当地更新，相互指向。

不管是回写、读扑空、或者排他读或更新，放到总线上的地址总是物理地址。用从总线来的物理地址侦听是容易的。由于所需的信息已经在那里，不需要显式逆向转换。查询物理的标记以检查块的存在，从它所包含的指针找到数据。如果必须采取什么动作，由物理标记项所指的虚拟标记状态也要被更新。这样一种缓存系统如何操作的进一步的细节见（Goodman 1987）。这种做法也被扩展到多级缓存，此时它更具吸引力：L1 缓存用虚拟标记以加速缓存的访问，L2 缓存用物理标记以便利侦听和避免跨处理器的别名（Wang，Baer and Levy 1989）。

6.6.3 转换检测缓冲器的一致性

简单地讲，处理器的转换检测缓冲器（Translation lookaside Buffer，TLB）只不过是关于页表条目（PTE）的一个高速缓存，用于虚拟地址到物理地址的转换。由于实际的数据共享或者进程迁移，一个 PTE 可能出现在多个处理器的 TLB 中。PTE 可能被修改——例如，当存储

页被交换出去，或者它的保护被改变——导致和缓存一致性问题相似的情况。

438 ~ 439

有多种用于TLB一致性的方案。由于TLB一致性操作要比一般缓存一致性操作少得多，流行的做法是通过操作系统的软件方案。具体的方案取决于PTE是直接由硬件装入TLB，还是由软件控制；还取决于TLB和操作系统如何实现的其他一些因素。硬件方案也用于某些系统中，特别是当TLB操作为软件不可见时。这一节对4种TLB一致性方法给出一个简略的概要：虚拟地址缓存、软件TLB击落、地址空间标识符（ASID）、硬件TLB一致性。进一步的细节可见（Thompson et al. 1988；Rosenburg 1989；Teller 1990）以及其中的参考文献。

TLB，以及由此而来的TLB一致性问题，能够通过用虚拟地址缓存完全避免。地址变换现在只在缓存扑空上需要，因此特别是如果缓存扑空率低，我们可以直接用页表。当页表项被访问的时候带入普通的数据缓存，因此由缓存一致性机制保持一致性。然而，当一个物理页被交换出存储器时，或者它的保护改变了，这些都是缓存一致性硬件看不见的，因此必须由操作系统将PTE从所有处理器的虚拟地址缓存中清除出去。除此以外，虚拟地址缓存的一致性问题必须要解决。这个方法在SPUR研究项目中得到了探讨（Hill et al. 1986；Wood et al. 1986）。

第二个方法称为TLB击落。有许多变形，取决于硬件支持的程度，通常包括对处理器之间中断的支持，个别TLB项的作废。这种TLB一致性过程由一个称为发起者的处理器在它改变那些可能被其他TLB缓存的PTE时启动。由于对PTE的变化必须由操作系统来做，操作系统知道哪些PTE正在被改变，它也可能知道哪些其他的处理器可能将它们缓存到它们的TLB中。（保守地，由于那些项可能已经被替换了）操作系统内核锁住正被改变的PTE（或者相关的页表段，取决于锁的粒度），向其他被认为有拷贝的处理器发中断。一旦被中断，接受者屏蔽中断，察看正被修改的页表项（在共享存储中），在本地从它们的TLB中作废这些项。发起者等它们完成，也许通过轮询共享存储单元，然后打开页表段的锁。在Mach操作系统中，人们用了一种不同的、但要更复杂些的击落算法（Black 等 1989）。

某些处理器家族，尤其是Silicon Graphics的MIPS家族，用软件装入的TLB而不是硬件装入的TLB，这意味着操作系统不仅涉及到PTE的修改，还涉及在扑空时将PTE装入TLB。在这些情况下，由于进程迁移带来的处理器私有页面的一致性问题可以用第三种方法来解决，即ASID，它避免了中断和TLB击落。每个TLB项都有一个ASID域与之相联，以避免在上下文交换时清除整个TLB（就好像进程标识符用在虚拟地址缓存一样）。然而，在TLB的情形，ASID像是由操作系统以处理器为单位动态分配的标签，用一个自由池，当TLB项被替换时，让它们返回其中；它们不是在整个生命周期都和进程相联的。IRIX 5.2操作系统用ASID的方法如下。操作系统为每个进程维护一个数组，跟踪分配给系统中每个处理器的那个进程的ASID。当一个进程修改PTE时，对所有其他处理器那个进程的ASID被置零。这保证了当进程迁移到另外的处理器时，进程将发现它的ASID为零；因此内核就要分配给进程一个新的ASID值，这样就防止了用过时的TLB项。对于由进程真共享的页的TLB一致性用TLB击落来完成。

440

最后，某些处理器家族提供硬件指令来作废其他处理器的TLB。在PowerPC家族中（Weiss and Smith 1994），“TLB作废项”指令（tlbie）在总线上广播页地址，从而其他处理器上的侦听硬件能自动地作废相应的TLB项，而不中断处理器。处理PTE变化的算法是简单的：操作系统首先改变页表，然后发出一个针对这些变化的PTE的tlbie指令。如果TLB是

硬件装入的（如在 PowerPC 中那样），OS 不知道哪些其他的 TLB 可能缓存这个 PTE，因此这个作废必须广播到所有处理器。广播是很适合于总线的，但对于分布式网络的可扩放的系统来说却是不希望广播的，我们将在后边的章节讨论。

6.6.4 环上基于侦听的高速缓存一致性

由于基于总线的缓存一致性多处理器的规模受总线的限制，很自然我们就会问侦听式一致性怎么能够扩展到其他的、局限性小一些的互连结构。总线的一个直接扩展是环。和总线（所有模块都接在同一组线上）不同，环上每个模块都和相邻的两个模块相连。和总线相同的方面是，环固有地支持基于广播的通信，从一致性角度看它是一个有意思的互连网络。从一个节点到另一个节点的事务沿着环的链向下传递，由于目的节点的平均距离是环长的一半，应答的一种简单且自然的做法是让它传播通过环的剩余部分，返回发送者。事实上，用硬件组织通信结构的一种自然方法是，让发送者将事务放到环上，当事务通过时，让其他节点审查（侦听）此事务是否与自己有关。给定这种广播和侦听的基础结构，我们可以在环上提供侦听缓存一致性，即使存储器在物理上是分布在环的节点上。由于多个数据传送过程可能在环上同时进行，模块看到这些事务的时间不同，顺序可能潜在地不同，环上的串行化和顺序一致性要比总线上的复杂些。

除了用分布式存储器外，环相对于总线的优点还有较短的、点对点链接使它们能够以很高的时钟频率工作。例如，IEEE 可扩展一致性接口（SCI）传输标准（Gustavson 1992；IEEE 1993）就是基于 500 MHz，16 位宽的点对点连接。线性点对点特点也使得链路深度流水成为可能，即在前面的位没有到达目的节点之前新的位能够发送到线路上。这个特征允许链路做得长一些，同时不影响它们的吞吐量。环的一个缺点是通信时延高，高于总线，并且随着环上处理器个数增加而线性增长（平均来说，在单向环上，要达到目标，需要穿越 $p/2$ 跳；双向环上减半）。

441

由于环是一个广播介质，侦听缓存一致性协议可以很自然地在其上实现。早期的一种基于环的侦听缓存一致性机器是 Kendall Square Research 的 KSR1（Frank，Burkhardt，and Rothnie 1993）。最近一些的商用机器，诸如 Sequent NUMA-Q 和 Convex 的 Exemplar 家族（Convex 1993；Thekkath et al. 1997），用环作为二级互连，将多处理器节点连在一起。（这两种系统在环互连上都是用目录协议，而不是侦听，因此对它们的讨论将推迟到第 8 章，专门介绍那些协议。在 NUMA-Q 中，节点内的互连是总线；在 Exemplar 中，它是具有丰富连接关系，低时延的交叉开关。）多伦多大学的 Hector 系统（Vranesic et al. 1991；Faras，Vranesic and Stumm 1992）是一个基于环的研究原型系统。

图 6-25 示出一个用环连接的多处理器组织。典型地，环和物理上分布的存储一起使用，但存储器可能在逻辑上仍然是共享的。每个节点的构成包含有一个处理器、它的私有缓存、全局主存的一部分、还有一个环接口。这个环接口有一个从环上来的输入线，一组组织成 FIFO 缓冲区的锁存器，以及一个到环上的输出线。在每个环时钟周期，锁存器的内容向前移位，于是整个环就像一个环形流水线。锁存器的主要功能是将通过的信息持有足够长的时间，让接口来决定是否将它向前发送到下一个节点。一个信息可能被从环上取出，通过将锁存器的内容存入本地缓冲存储器，并向这个锁存器写入一个空时隙标记。如果一个节点要将事务放到环上，它要等待一个空时隙通过，并填充它。当然，我们希望在每个接口上的锁存

器数尽量小，以减少事务通过环的时延。

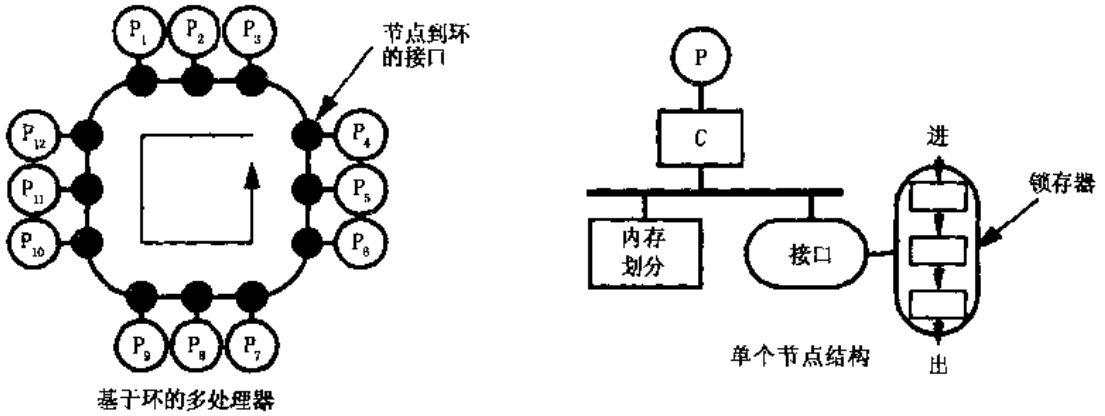


图 6-25　一种单环多处理器的组织结构

需要有一种机制，称为环访问控制机制，来确定一个节点何时能够将它的事务放到环上。它的复杂性在于：环上的数据通路要比在它上面传输的事务窄得多。结果，事务在环上需要多个相继的时隙。进一步，环上的事务（消息）本身大小也可能不一样。例如，请求消息短，只包含命令和地址，而数据应答消息存储块的内容要长得多。最后的复杂因素是，访问环的仲裁逻辑必须是分布式的，这是因为它不同于总线，没有全局可见的线路。

有三种主要的环访问控制（即仲裁）方法：令牌传递环、寄存器插入环和时隙环。在令牌传递环中，一种特别的位模式，称为令牌，在环上传递，只有当前占有这个令牌的节点才能在环上发送。这种情况仲裁是容易的，但缺点是同时只有一个节点能发起一个事务，尽管环上的其他节点可能传递的是空槽，这就导致带宽的浪费。寄存器插入环为 IEEE SCI 标准所选。这里，在环的接口输入和输出阶段之间，当本地节点在发送的时候，用一个旁路 FIFO 来缓冲输入消息（带有反向流控以避免溢出）。当本地节点完成时，旁路 FIFO 的内容就被转发到输出链路，本地节点不允许再发送，直到旁路 FIFO 为空。多个节点可以同时发送，环的各个部分当它们的旁路 FIFO 溢出时就要停滞。最后，在时隙化的环中，环被分为事务时隙，带有类型标签（针对不同大小的事物，如请求和数据应答），这些时隙在环上不断循环。一个准备好了发送消息的处理器等待所需类型的时隙的到来（由槽头的一位指出），然后插入它的消息。这里的“时隙”实际上是一串空时间片段，这个串的长度取决于消息的类型。时隙化的环可能会限制环带宽的利用率，用硬件混合不同类型的时隙，可能和给定负载的实际流量模式不匹配。然而，对于一个给定的一致性协议，人们对消息类型混合的情况了解得较好，因此在实践中几乎没有什么带宽浪费（Barroso and Dubois 1993，1995）。

也许我们会感到广播和侦听在像环这样的点到点互连上会浪费带宽，实际中不一定如此。在环上，和平均随机点对点消息相比，广播只须两倍的带宽，这是由于对两个随机选择的点来说，前者平均要穿越半个环。除此以外，只是请求消息需要广播（读扑空、写扑空、更新请求），而它们是短的；数据应答消息由数据源放到环上，运转到请求节点上停止。

442 ~ 443

考虑在基于环的侦听协议中的一次读扑空。如果我们关心的一个存储块所分配的存储单元（称为宿主存储器）不在本地节点，读请求就被放到总线上。如果宿主是本地的，我们还要确定这个块在其他节点是否脏，在这种情况，本地存储器不应该响应，一个请求应该放到总线上。一个简单的方案是将所有扑空都放到总线上，如 Sun Enterprise，它是一个基于总线

的设计，但存储在物理上是分布的。另一种方案是，对宿主存储器中的每一块都维护一个脏位。如果这一块在其他某个节点的缓存中为脏，这一位被置位。如果这一位置位，请求就送到环上。读请求现在沿着环循环。所有节点都检测它，宿主节点或者脏节点会作出响应（如果宿主不是本地，宿主节点就用脏位来决定它是否应该对请求做响应）。排他读和更新过程也出现在环上，其他节点侦听这些请求，必要时作废它们的存储块。当回到请求者时，请求和响应事务就从环上删除。请求返回到请求节点作为确认。当多个节点试图并发地写一个相同的块，赢者是首先到达该块的当前拥有者的节点（即如果那块在主存有效，则是宿主节点，否则是脏节点）；其他节点被隐式或显式送一个 NACK，然后它们必须重试。

从实现的角度看，环上侦听协议的一个主要难点在于所强加的实时限制：在环接口中的侦听器考察并对所有通过的消息作出反应，不能有过多的延迟或内部排队。这对于寄存器插入环是困难的，因为许多短请求消息可能在环上相邻。对于高速运行的环，请求可能相互挨得太近，侦听器难以在固定的时间里做出响应。这个问题在时隙环中得到简化，短请求消息和长数据响应消息（后者是点对点，不需缓存查找）的选择和布置能保证请求型消息相互有一定的距离（Barroso and Dubois 1995）。例如，时隙可以成组为帧，每个帧可以组织成请求时隙，紧跟着响应时隙。尽管如此，如同总线，环的带宽最终还是受控制器或缓存的侦听带宽限制，而不是受互连上的原始数据传送带宽限制。

环的串行化和顺序同一性要比总线复杂些，这是由于存在这样的可能性：环上不同点的处理器会看到环上一对消息有不同的顺序（取决于它们在环的什么地方，相对于哪些事务发送者）。用作废协议可以简化这个问题，由于写只是引起排他读事务放到环上，而不是数据本身，除宿主节点外的所有节点都只要简单作废它们的拷贝。宿主节点能确定环上出现冲突的消息，并采取特定的行动，但这不会在协议中增加太多的暂态过程。

444

6.6.5 在基于总线的系统中的数据和侦听带宽的扩展

有一些方法可用来增加 SMP 设计中的带宽，同时保持基于总线做法的简单性。在事务拆分型总线中，我们看到仲裁、地址阶段和数据阶段的流水线使得它们能同时进行。事实上，扩展数据带宽事实上是较容易的；挑战在于拓展侦听的带宽。

首先考虑扩大数据带宽。缓存块的尺寸和描述它们的地址相比要大些[⊖]。增加数据带宽最直接的方法是使数据总线的宽度扩大。我们在 Sun Enterprise 和 SGI Challenge 的设计中看到了这一点，两者都用 256 位宽的数据总线。在 Enterprise 中，64 字节的缓存块只要两个周期就可以传送了。这种做法的不利之处是代价：随着总线的变宽，它要用较大的连接器，占据电路板上更多的空间，需要更大的功率。它显然将这种风格的设计推向极限，由于一个高效的、统一的流水线要求侦听操作必须在两个周期完成，侦听操作是需要被所有的缓存看见并确认的。一种更突出的做法是用点到点交叉开关代替总线的数据部分，直接将每个处理器－存储器模块相互连接。这种做法认识到了只是事务的地址部分需要广播到所有的节点，来确定一致性操作和数据源（即存储器或缓存）。这种做法在 IBM 基于 PowerPC 的 RS6000 G30 多处理器中被采纳。一个总线被用于地址和侦听结果，但交叉开关用来移动实际的数据。交叉开关中的单个通路不需要太宽，因为多个传送能同时发生。

⊖ 例如缓存块可能为 16 字节，而地址为 8 字节。——译者注

在基于总线系统中提高带宽的一个笨办法是简单地用多条总线，包括地址总线，如6.4.7节所提到的那样。事实上，这个做法提供了一种基础性贡献，它允许侦听带宽也能扩展。为了将侦听带宽扩展到在每地址周期一个一致性结果之外，必须有多个同时的侦听操作。一旦有了多个地址总线，数据总线问题可以通过数据总线，交叉开关或者任何其他机制来处理。同一性是容易的。地址空间的不同部分用不同的总线；典型地，每条总线将为特定的存储模块服务，因此一个给定的地址总是用相同的总线。多个地址总线似乎违反了用于保证顺序同一性关键的机制：对地址总线的序列化仲裁。然而，我们记得，顺序同一性要求的是一个逻辑的总序，而不是一个严格的地址事件的发生时间序。一种静态的序在逻辑上赋予参与的总线操作序列：一个地址操作 i 逻辑上在 j 之前，如果它在 j 之前出现（按照总线时钟）或者它们在相同的时钟周期发生，但 i 发生在较低编号的总线上。这种多总线的方法用在 Sun SparcCenter 2000 中，它提供两条事务拆分型总线，每条和 SparcStation 1000 中用的一样，可以扩展到30个处理器。CRAY CS6400 用4条这样的总线，可以扩展到64个处理器。每个缓存控制器侦听所有总线，按照缓存一致性协议作出响应。Sun Enterprise 10000，比本章讨论的 Sun Enterprise 6000 后发布的机器，将多个地址总线和数据交叉开关结合起来，可以扩展到64个处理器。每个板上有4个250 MHz 的处理器，4个存储模块（每个最多1 GB），以及两个独立的 SBUS I/O 总线。16个这样的板通过 16×16 数据交叉开关连接起来，144位宽的通路，以及4条和每个板上4个模块相联的地址总线。总体上来说，这就提供了12.6 GBps 的数据带宽和一个高达250 MHz 的侦听速率。 445

6.7 结论

本章探讨的设计问题是具有基础意义的；虽然所用的都是小规模并行的例子，但其中的原理在中等规模并行的情形也有指导作用。当然，最优的设计方案可能改变。例如，尽管当前不那么流行，但在一定条件下，在某个层次共享缓存可能变得相当有吸引力，这种条件就是多模块封装技术变得便宜起来，或者多个处理器出现在单个芯片上。

处理器通过共享总线互连，尽管是一个非常有效的机制，但随着处理器的数目或者速度的增加，受到明显的带宽限制。尽管如此，至少在小规模上，系统设计师们肯定还会不断去寻找新方法，从总线型设计中“榨取”更大的数据带宽和侦听带宽，来充分地利用基于广播方法的简单性。然而，建造可扩放缓存一致性机器的一般方法是让存储器物理上分布在节点上，用一种可缩放的互连，以及不依赖于侦听的一致性协议。这个方向是后面几章的主题。随着处理器变得比总线和侦听带宽快，这种思路在较小规模也可能有用。对于总线结构来说，肯定在一定的时间里还会有重要的作用，但现在难以预测它们的将来，以及在什么样的规模上还会用它们。虽然总会有各种变化和进步，但我们针对总线在这两章里讨论的问题，大多数都是独立于工艺的，对所有设计缓存一致性共享存储系统结构都至关重要，不管用何种互连方式。这些问题包括存储层次内的互连，缓存一致性问题和在状态转换层次的各种缓存一致性协议，以及当处理许多并发过程时所引起的关键正确性和实现问题。用于解决这些正确性和实现问题的具体做法可能改变，但这些问题本身、围绕它们所作的权衡以及基本的出发点是具有基础意义的，可以外推。进而，对于本书后面要讨论的大规模系统的设计来说，这些基于总线的设计提供了基本的元素。

习题

6.1 考虑两个机器 M_1，M_2。M_1 是一个 4 处理器共享 L_1 高速缓存机器，M_2 是一个 4 处理器基于总线的侦听缓存机器。M_1 有一个共享的 1 MB 两路组相联高速缓存，64 字节的块；M_2 中的每个处理器有一个 256 KB 的直接映像高速缓存，64 字节块。M_2 用 Illinois MESI 一致性协议。考虑如下代码：

446

```
double A[1024,1024]; /* row-major; 8-byte elems */
double C[4096];
double B[1024,1024];

for (i=0; i<1024; i+=1) /* loop-1 */
  for (j=myPID; j<1024; j+=numPEs)
    {
     B[i,j] = (A[i+1,j] + A[i-1,j] +
           A[i,j+1] + A[i,j-1]) / 4.0;
    }
for (i=myPID; i<1024; i+=numPEs) /* loop-2 */
  for (j=0; j<1024; j+=1)
    {
     A[i,j] = (B[i+1,j] + B[i-1,j] +
           B[i,j+1] + B[i,j-1]) / 4.0;
}
```
⊖

1）假设数组 A 从 16 进制地址（0x）0 开始，数组 C 从0x300 000开始，数组 B 从0x308 000开始。所有高速缓存都初始化为空。每个处理器执行上面的代码，和 4 个处理器对应，myPID 在 0 到 3 之间变化。针对两个循环嵌套，分别计算 M_1 的扑空。对 M_2 也做一遍，指明你所作的任何假设。

2）如果没有数组 C，简单地说明你的 1）中答案会如何改变。指明你所作的任何其他假设。

3）从这个练习中，我们了解到共享高速缓存结构的什么优缺点？

6.2 假设你从前面的章节和 5.4 节的数据中已经有了关于 Barnes-Hut、Ocean、Raytrace 和 Multiprog 负载的知识。考虑一个 4 处理器，共享 4 MB 高速缓存的机器和一个 4 处理器，基于总线侦听的机器，每处理器 1 MB 高速缓存，讨论这些应用在它们上执行的情况。用模拟来验证你的直觉可能是有用的。指明任何相关的假定。

6.3 和共享一级高速缓存相比，私有一级高速缓存但共享二级高速缓存有什么优缺点？评价现代处理器的设计中，例如 MIPS R10000 和 IBM/Motorola 的 PowerPC 620，是如何体现或回避这种趋势的。在这些设计中封装技术的影响是什么？

6.4 用 6.3 节关于高速缓存包含的术语，假设 L_1 和 L_2 是两路组相联，$n_2 > n_1$，$b_1 = b_2$，替换策略是 FIFO，不是 LRU。包含关系自然成立吗？如果替换是随机的或基于循环计数器又如何？

447

⊖ 原著中将 A [i+1, j] 和 B [i-1, j] 误为 A [i+i, j] 和 B [i+i, j]。——译者注

6.5 针对下述情形，给出一个表现高速缓存包含性质被违反的存储引用流：

1）L_1 高速缓存是 32 字节，两路组相联，8 字节缓存块，LRU 替换。L_2 缓存是 128 字节，4 路组相联，8 字节缓存块，LRU 替换。

2）L_1 高速缓存是 32 字节，两路组相联，8 字节缓存块，LRU 替换。L_2 缓存是 128 字节，两路组相联，16 字节缓存块，LRU 替换。

6.6 对于下面的系统，说明高速缓存是否自然地提供包含：如果不，说明问题在哪里或者给出一个违反包含的例子。

1）L_1：8 KB 直接映像基本指令高速缓存，32 字节块大小；8 KB 直接映像基本数据高速缓存，直写，32 字节块。

L_2：4 MB 4 路组相联统一二级高速缓存，32 字节块大小。

2）L_1：16 KB 直接映像统一基本高速缓存，直写，32 字节块大小。

L_2：4 MB 4 路组相联统一二级高速缓存，64 字节块大小。

6.7 回顾 6.3 节中关于高速缓存包含性质的讨论。

1）在通常情形，包含能相当自然地被满足。这情形是，L_1 高速缓存是直接映像（$a_1=1$），L_2 可以是直接映像或组相联（$a_2 \geq 1$），带有任何替换策略（例如，LRU、FIFO、随机），只要带进来的新块放到 L_1 和 L_2 高速缓存，块的大小是相同的（$b_1=b_2$），L_1 高速缓存中的组数等于或小于 L_2 的（$n_1 \leq n_2$）。说明这为什么是对的。

2）我们的讨论称，用一个统一高速缓存支撑在某一级的多个高速缓存所产生的问题并没有通过使那个统一高速缓存相联而得到解决。用一个简单的例子，有直接映像指令和数据高速缓存并由一个统一的、两路组相联高速缓存支持，说明这是对的。

6.8 假设每个处理器有分开的指令和数据高速缓存，且没有指令扑空。还假定，当活跃时，每 3 个时钟周期处理器发出一个数据高速缓存请求，扑空率是 1%、扑空时延是 30 周期。假设读标记要 1 个时钟周期，但修改标记要两个时钟周期。

1）如果用带有一组高速缓存标记的单级数据高速缓存，试量化由于高速缓存标记争用的性能损失。假设总线过程要求侦听每 5 个时钟周期发生一次，它们的 10%将高速缓存的一个块作废。还假设侦听的优先级高于处理器访问标记的优先级。做粗略地计算。然后通过建立一个排队模型或写一个简单的模拟器，检查你的答案的精确度。

448

2）如果对处理器和侦听用分开的标记集，标记竞争的性能损失如何？

3）一般来讲，你会将访问标记的优先级给与处理器引用还是总线侦听？

6.9 SGI Challenge 多处理器的设计者们考虑了下面的总线控制器优化，以更好地利用交叉存储和总线带宽。如果控制器发现对一个给定的存储模块（它可以通过请求表来确定），一个请求已经是待完成的，它就不发出那个请求，直到那个模块的先前请求被满足。讨论这种优化潜在的问题，Challenge 设计中的什么特征允许这种优化？

6.10 1）尽管 Challenge 支持 MESI 协议的状态，它不支持最初 Illinois MESI 协议中的高速缓存到高速缓存的传送特征。

ⅰ）讨论这种选择的可能原因。

ⅱ）扩充 Challenge 实现，以支持高速缓存到高速缓存的传送。如果有的话，描述

总线上所需的额外信号，不要忘了公平性问题。

2）尽管 Challenge MESI 协议有四个状态，和高速缓存控制器芯片一起存放的标记只跟踪三个状态（I、S、E+M）。解释为什么这依然可以正确工作。你认为他们为什么做这种优化？

3）针对 SparcCenter 的多总线体系结构和 SGI Challenge 的单-快-宽总线体系结构，讨论它们之间代价、性能、实现和扩展性方面的权衡，以及任何程序语义和死锁的影响。

6.11　1）Challenge 的主存，甚至在它决定出一个数据是否在其他处理器的高速缓存中脏之前，可以推测式地启动该数据的读取过程。利用表 5-1 的数据，估计无用主存访问的比例。基于那个数据，你支持这种优化吗？这个数据从方法论上是充分的吗？请解释。

2）Challenge 的总线接口支持请求的融合。这样，如果多个处理器在等待同一个存储模块，那么当数据出现在总线上时，它们都可以从总线上抓住数据。这个特征在实现基于踏步等待的同步原语时特别有用。对于测试-测试和设置锁，计算有这种优化和没有这种优化时总线上最小的流量。假设有 4 个处理器，每个获得锁一次然后做一次开锁，最初没有处理器在其高速缓存中有包含锁变量的存储块。

6.12　SGI Challenge 总线允许 8 个待完成事务。设计人员是怎么得到这个数的？试建议一个通用公式，能够基于总线的参数给出应该支持的待完成过程数。用如下参数：

449

P——处理器数；

M——存储器模块数；

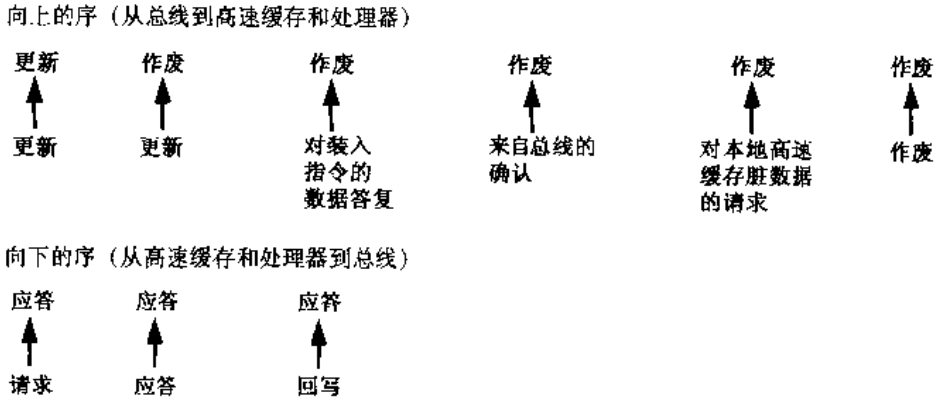
L——平均存储时延（周期）；

B——高速缓存块大小（字节）；

W——数据总线宽度（字节）。

定义其他你认为重要的参数。保持你的公式简单，讲清楚任何假设，说明你的决定的正确性。

6.13　在一个两级高速缓存系统中，考虑从总线到高速缓存层次的入向事务（请求和响应）和从高速缓存层次到总线的出向事务。为了保证当作废被提前确认（一旦它们出现在网络上）时的顺序同一性（SC），对于下图所描述的，必须在每一个方向上维护什么序的限制，哪些可以被重定序？对于一级高速缓存的入向事务回答同样的问题。假设每个处理器同时只有一个待完成请求，但总线是拆分型的。



6.14　在 6.4 节讨论的拆分型解决方案中，取决于处理器到高速缓存的界面，可能一个作废请求立刻在数据响应后到来，因此这个块，在处理器有机会实际访问高速缓存块满足它的请求之前就被作废。这为什么可能是个问题，你如何解决它？

6.15　当支持免锁高速缓存时，一个设计人员建议，我们也可以在事务拆分型总线接口中的请求表中加更多的项。这个想法好吗？是否能得到较大收益？

6.16　应用 6.4.6 节中描述的不同的技术在维持例 6.3 中有多个待完成总线事务的 SC，使你自己相信它们的确能解决问题。在什么条件下一种方案要好于其他的？ 450

6.17　假设一种类似于 Powerpath-2 那样的系统总线，如 6.5 节所讨论的。假设 200 MIPS/200 MFLOPS 处理器，1 MB 高速缓存和 64 字节高速缓存块，对于表 5-1 中的每个应用，计算总线带宽，用

1）Illinois MESI 协议

2）Dragon 协议

3）Illinois MESI 协议，假设 256 字节的高速缓存块

对每个部分 1），2），3）分别计算地址 + 命令总线的利用率和数据总线的利用率。阐明所有假设。

4）针对 SparcCenter XDBus，比较 1）和 2）两个部分，它有 64 位宽的多选地址和数据信号。假设总线在 100 MHz 上运行，发送地址信息要两个总线周期，64 字节的数据要 9 个总线周期。

6.18　在 6.4.8 节提出的关于多级高速缓存死锁的一个解决方案是使所有队列足够深，能够容纳所有的入向请求和响应。这些队列可以小一些吗？若如此，为什么？讨论为什么让队列的深度大于死锁考虑所需的深度可能是有益处的。

6.19　6.3 节给出的一致性协议假设两级高速缓存。如果有三级或更多的高速缓存层次又会如何？试将 Illinois MESI 协议扩充到一个三级高速缓存的中间级：列出任何附加的状态或所需的动作，给出状态转换图。

6.20　图 6-26 表示用在 Mach 操作系统中的 TLB 击落算法的细节（Black et al.1989）。对每个处理器来说，维护下面的基本数据结构：一个“活跃”标识，指出处理器是否正活跃地用着页表；一个 TLB 清理通知的队列，指出虚拟地址的范围，其映射要被改变；一个指出当前活跃页表的表（即那些其 PTE 当前可能被存到处理器的 TLB 中的进程）。对每一个页表，当一个处理器在改变页表时必须持有一把踏步等待锁和一组处理器，在其上这个页表当前是活跃的。虽然基本击落方法是简单的，实际的实现需要仔细安排有关步骤的顺序和数据结构的加锁。

1）为什么页表项的修改时间要在对 TLB 一致性的处理器发送中断或者作废消息之前？

2）为什么图 6-26 中击落的发起者要在获得页表锁之前屏蔽处理器之间的中断（IPI）并清除它自己的活跃标记？你能想到图中存在的任何死锁条件吗，若如此，怎样解决它们？

3）和 Mach 算法有关的一个问题是，当发起者改变页表时，它使所有的响应者忙等待。这里的原因是它的设计思想，在 TLB 替换时，只要脏位被设置了，就用微处理器自主回写整个 TLB 项到相应的 PTE。这样，如果其他处理器被允许用页表，当发起者正在改变它的时候，一个从这些处理器来的自主的回写可能覆盖新的变 451

化。你会怎样设计 TLB 硬件和/或算法，使得响应者不要忙等待？

4）在什么情况下，清除整个 TLB 要比有选择地使 TLB 的某些项无效来得更好？

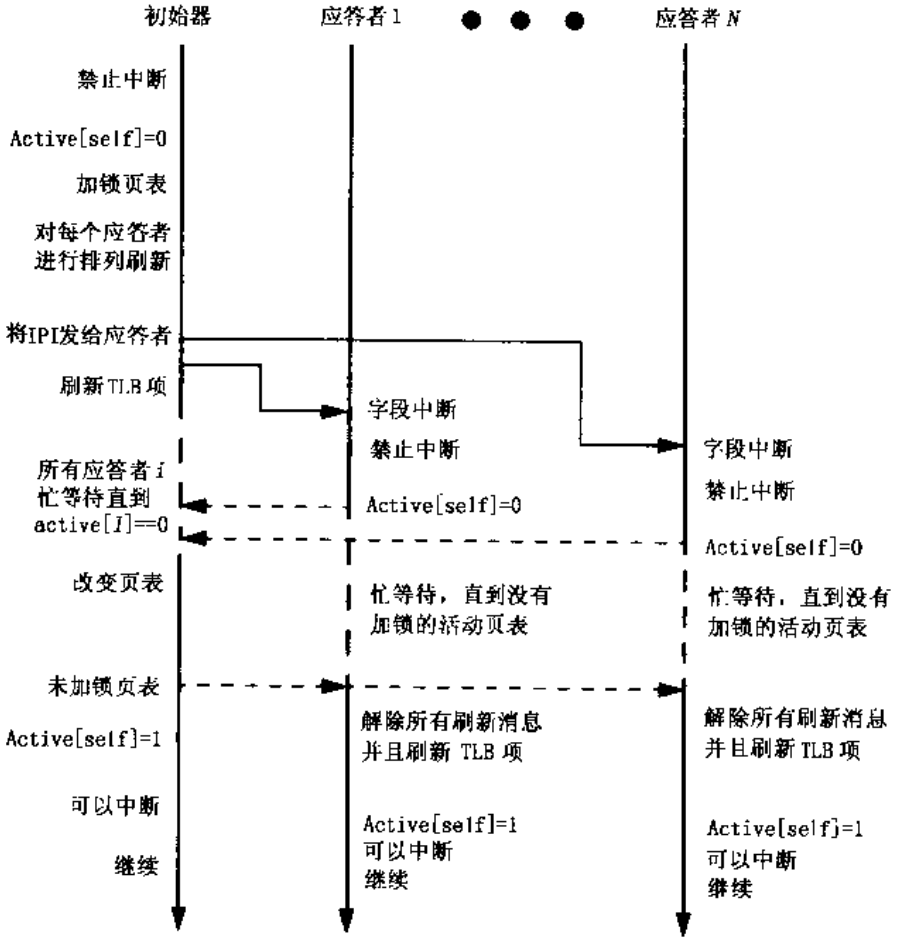


图 6-26 Mach TLB 的击落算法。发起者是改变某个页表的处理器，响应者是所有其他处理器，它们的 TLB 中可能高速缓存有发起者所改变页表的一些条目

452

第7章　可扩展多处理器

本章，我们开始研究机器的设计问题，这些机器在实际中能够扩展到具有几百甚至上千个处理器。可扩展性在设计的各个层次上都具有深刻的含义。对于初学者来说，构造一个大规模的配置必须在物理上可能和在技术上可行。增加处理器显然可以增加系统的潜在计算能力，但为了真正实现这种潜力，就要求系统在各方面是可扩展的。特别是，存储器的带宽应能够根据处理器数量进行扩展。一种自然的解决方案正如我们的通用多处理器那样，将存储器与处理器一起分配，这样每个处理器就能够直接访问局部存储器。但是，连接这些节点的通信网络必须能够在合理的延迟条件下，提供可扩展的带宽。此外，用来在系统间传输数据的协议也必须可扩展，同步技术也必须如此。在可扩展网络上运行可扩展性协议，系统中可能同时产生非常多的事务，所以我们不能依靠全局信息建立次序或仲裁资源的使用。所以，为了获得可扩展的应用性能，必须把扩展性当作涉及系统设计的每一层的“垂直”问题来处理。下面让我们考虑几个熟悉的设计要点来具体说明可扩展性问题。

在第5、6章中描述的小规模共享存储器的机器可以被看做是一种极端情况。一条典型的共享总线最大长度只有1~2英尺，有固定数目的插槽和固定的最大带宽，所以它从根本上限制了规模。到通信介质的接口是存储器接口的扩展，使用额外的控制线和控制状态来支持一致性协议。通过对总线的仲裁来建立全局的次序，在任何时刻处于未决状态的事务数有限。通过标准的虚-实地址转换机制，强化了所有的通信操作的保护。系统中的处理器间完全信任，被看做是一个运行于所有处理器的单一操作系统的控制之下，具有公共的系统数据结构。通信介质被包括在机箱物理结构之中，因此绝对安全。典型的情况是，当任何处理器失效，系统就进行重启。而在编程模型和基础硬件之间几乎没有任何软件形式的干预。因此，在系统设计的每一个层次，决策是基于其下层的可扩展限制以及部件间紧耦合的假设。可扩展的机器比基于总线的共享存储器多处理器的耦合度要低，而且我们必须重新考虑处理器和其他处理器以及存储器的交互方式。

在另一个极端，可以考虑在一个局域网或甚至一个广域网上的传统工作站。这里，对物理规模没有明显的限制，并且系统的处理器间没有什么信任度。到通信介质的典型接口是硬件级的标准外部接口，操作系统介入到用户级原语和网络之间，进行保护和访问控制。每一个处理器都由它自己的操作系统进行控制，而对其他的处理器不信任。这里不存在操作的全局次序，所以很难达成一致。通信介质独立于各个节点之外，所以具有潜在的不安全性。每个工作站自身可能会出错并能够独立地重启，除非它正在为其他工作站提供服务。不管采用什么编程模型，在基础硬件和任何用户级的通信操作之间有一层软件存在，这就使得通信延迟相当大，而通信带宽低。由于通信操作由软件来完成，所以对未决的事务的数目以及事务的含义没有明确的限制。在系统设计的每一个层次，假设和其他节点间的通信很慢而且不可靠。所以，即使当多处理器在共同解决同一个问题时，也很难发掘问题内在的耦合性和信任性来得到更高的系统性能。

在这两个极端情况之间有一系列合理的和有意义的设计选择，其中某些可以体现在现行

的商用大规模并行计算机和正在出现的并行计算机群之中。很多大规模并行处理器（MPP）使用复杂的封装和快速的专用网络，以使非常多的处理器可以在有限的空间内获得高带宽和低时延的通信。其他的可扩展机器基本上是使用传统的计算机作为节点，这些节点采用大致标准的接口和快速网络进行连接。在这两种情况下，存在许多物理上的安全措施，以及高度信任或基本自治之间的选择。

第1章中的通用多处理器提供了一个理解可扩展设计的有用框架：机器由基本上完整的计算节点组成，每一个节点具有存储器子系统和一个或多个处理器，由可扩展网络相互连接。节点到网络的接口特性是可扩展系统设计中最关键的问题之一。它允许很多可能的方案，这些方案在处理器和存储器子系统与网络的耦合程度以及网络接口本身的处理能力上有所不同。这些问题影响了节点间的信任度和通信原语的性能特征，进而决定了不同的编程模型在机器上实现的效率。

我们在本章的目标是理解可扩展机器的一系列通信体系结构的设计折中。例如，我们希望理解，追求更加专用的或追求更加面向市售商品的做法，对节点到网络间接口的能力、对有效支持不同的编程模型的能力以及对规模上的限制又有什么样的影响。我们从7.1节开始讨论可扩展性的一般性讨论，考察它对系统设计在带宽、时延、成本和物理结构方面的要求。这一讨论提供了对近年来的一些大规模机器的基本介绍，同时也允许我们根据节点到网络接口“另一侧面”发展一种可扩展网络的抽象观点。在本章和下两章完整地形成对网络的基本需求之后，我们将在第10章对可扩展网络的设计做深入的研究。

454

在7.2节我们将集中讨论，在大规模并行机环境中，如何根据通信原语来实现编程模型的问题。关键的概念是网络事务，对于可扩展网络而言，它和前一章所研究的总线事务相类似。通过研究一个非常抽象的概念——网络事务，我们将考察共享地址空间和消息传递机制是如何通过构造于网络事务之上的网络协议而实现的。

本章的其余部分将考察一系列的重要设计问题，硬件对网络事务中的信息直接解释的程度逐步提高。从一般意义上说，对网络事务的解释和对指令集的解释很类似。原理上非常有限的解释已经足够，但在实践中，更加延伸的解释对性能来讲是重要的。7.3节考察了本质上没有对消息事务进行解释的情况，它被看作是在操作系统的控制之下，通过物理的直接存储器访问（DMA）操作盲目地传送到存储器中去的比特序列。这是一个非常重要的设计问题，因为它代表了许多早期的MPP机和最为现代的局域网接口。

7.4节考虑了更加激进的设计策略，这里消息可以从用户层送到用户层，无须操作系统的干预。这至少要求网络事务携带一个用户/系统的标识符，该标识符由源通信辅助部件产生，由目标通信辅助部件所解释。这个小的改变产生了用户虚拟网络的概念，就像虚拟存储一样，它必须提供一种保护模式并提供一个用来共享底层物理资源的框架。一个特别关键的问题是用户级别的消息接受，因为对作为目的地的用户线程来说，消息的到达在本质上是异步的。

7.5节集中讨论能够提供全局虚地址空间的设计问题。这要求在目的地进行大量的解释，因为它需要执行虚-实地址转换，实施所需要的数据传输并负责通知。典型地，这些设计使用一个专用的消息或通信处理器（CP），以使得对网络事务的详尽解释可以在没有机器设计细节的情况下执行。7.6节考虑了对全局物理地址空间的特殊支持。在这种情况下，通信辅助部件和存储器子系统紧密集成；典型的情况是，它是一个支持有限网络事务集合的专

455

用设备。对全局物理地址空间的支持使得我们在原理上能够像设计小规模共享存储器的机器那样进行设计。然而，通过以可扩展模式在一致的高速缓存进行自动的共享数据复制比基于总线的情况更加复杂，我们将在第8、9章集中讨论这个话题。

7.1 可扩展性

一个设计怎样才能称为“可扩展”的？几乎所有的计算机都允许系统的能力以某些形式扩展，比如增加存储器、I/O卡、磁盘或升级处理器，但使这种增长有很大的限制。一个可扩展系统应尽量避免对添加系统资源的固有的设计限制。在实践中，一个系统可能具有相当大的可扩展性，尽管在任何时候都会由于经济方面的限制，不可能实现任意大规模的配置。当足够大的配置可以被建立的时候，可扩展的问题实际上已经变成了增加系统能力后所增加的成本和对应用所带来的性能增长的比例关系上。在实际中，并没有完美的可扩展性设计，所以我们的目标是要明白如何设计一个能够有效地扩展到大量处理器的系统。特别地，我们将自顶向下考察可扩展性的四个方面。首先，随着处理器的增加，系统的带宽和吞吐能力如何相应增加？理想情况下，吞吐能力应该和处理器数目成正比。第二，延迟或每个操作的时间如何增加？理想情况下它们应该是常量。第三，系统的成本如何增加？最后一点，我们如何封装和装配系统？

很容易知道，第6章中所讨论的基于总线的多处理器在这四个方面都不能很好地扩展，其理由是相互关联的。在这些设计中，多个处理器和存储器模块通过一组导线（即总线）连接。当一个模块驱动总线时，其他模块均不能再驱动它。这样，当更多处理器加入时总线带宽不能够增加；在某一点，总线会饱和。尽管接受这些缺陷，我们还是可以考虑通过一条总线连接许多处理器来构建机器，或许会相信随着处理器的增加，每个处理器对带宽的要求会减少。不幸的是，总线的时钟周期是由驱动一个值进入导线并使它被总线上的每个模块所采样的时间决定的，时钟周期会随总线上模块的数目和导线长度而增长。所以，较大的总线会有较长的时延和较小的集合带宽。事实上，信号的质量会随着线上连接器的数目的增加而衰减，所以对任何总线技术来讲，对能够插入的处理器的插槽数和最大线长有硬性的限制。在这些限制之内，基于总线的设计具有良好的成本扩展性，因为添加处理器和存储器增加的代价只是这些模块的成本。不幸的是，这种简单的分析忽视了一种情况：即使是最小的配置也要负担支持最大机器配置所需要的基础设施的固定成本；总线、机柜、电源以及其他的部件也必须按照最大配置来选择。最起码，一个可扩展的设计必须克服这些限制。集合带宽必须能够随着处理器数目的增加而增长，执行一个操作的时间不应该随着机器规模的扩展而有很大增加；而且大规模配置的构造必须是实际的而且是合算的。如果设计能够按比例缩小也是很有价值的，如此，小配置就是合算的。 456

7.1.1 带宽的可扩展性

从根本上说，如果大量的处理器要和许多其他的处理器或存储器同时交换信息，它们之间必须用很多条独立的导线进行连接。这样，可扩展机器必须像图7-1所抽象描述的那样进行组织；大量的处理器模块和存储器模块经由很多交换机用独立的导线（或链路）连接在一起。我们在非常通用的意义上使用交换机这个术语，指把有限的输入和有限的输出进行连接的设备。在内部，这样一个交换机可以用总线、交叉开关或甚至是一组特定的多路复用器来

实现。我们称输出（或输入）的数量为交换机的度。对总线来说，前面所讨论的物理和电气的约束决定了它的度。在某一时刻，只有一个输入能向输出传送信息。交叉开关允许每一个输入连接到一个不同的输出，而其度数被内部交叉点矩阵的成本和复杂度所限制。随着端口数量的增加，多路复用器的成本和时延都会快速增加。所以，交换机在规模上有限制，但可以互连以形成大的配置，也就是形成网络。除了输入和输出之间的物理互连外，还必须有某种形式的控制器来决定在每个特定时刻将哪个输入和哪个输出相连。本质上，一个可扩展网络类似于一个道路系统，导线对应街道，开关对应交叉路口，以简单的办法决定在每个路口上哪辆车可以通过。如果措施得当的话，大量的车辆就可以同时驶向它们的目的地，并迅速到达那里。

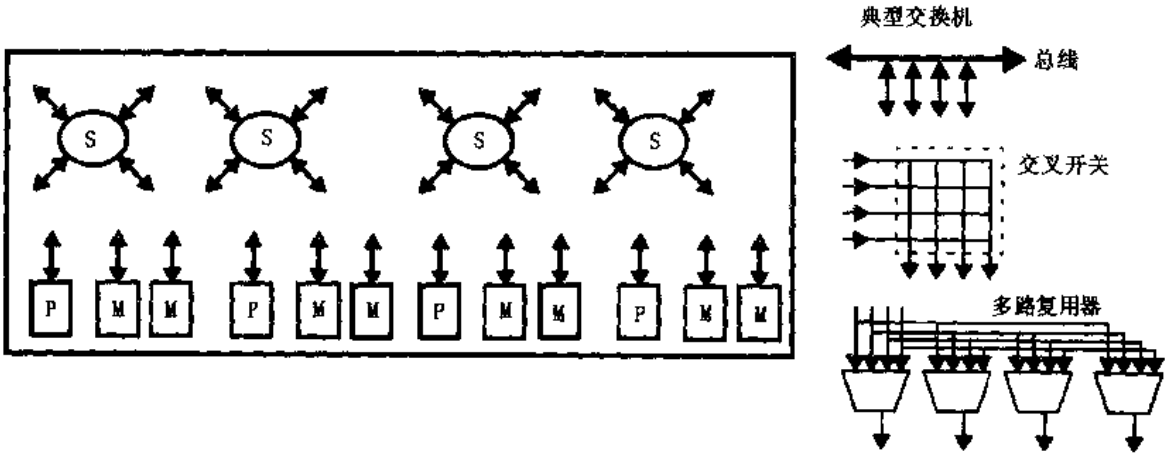


图 7-1　可扩展机器的抽象视图。大量的处理器（P）和存储器（M）模块由独立的导线（或链路），经由大量交换机模块（S）连接。每个交换机具有有限的度数。一个单独的交换机可以通过总线、交叉开关、多路复用器或其他的输入输出之间的受控连接来实现

按照我们的定义，一个基本的基于总线的 SMP 系统包括一条连接所有处理器和存储器的总线，以及把这些部件与外设相连的基于总线的交换机的简单层次结构。总线中的控制通路是专用的，与某个输入的事务相关的地址被广播给所有的输出，应答信号决定了哪个输出参与该事务。网络交换机是一种更加通用的装置，这里，输入提供的信息足以使交换机控制器决定合适的输出，而不必询问所有的节点。一对模块通过经由一系列网络交换机的路由连接。

可扩展机器的最通用的结构用图 7-2 中的体系结构来说明，其中把一个或多个处理器与一个或多个存储器模块以及一个通信辅助部件封装在一起，成为一个容易复制的单元，称其为一个节点。典型的节点的内部交换机是高性能总线。系统也可以通过“舞池”的方式构成，也就是处理器节点通过网络和存储器节点分开，如图 7-3 中所示。在每种情况下，都存在大量的可选择的交换机设计方式、互连网络拓扑结构和路由算法，这些将在第 10 章中进行研究。可扩展网络的一个关键特性是它提供了大量独立的节点间通信路径，这样当节点增加时，带宽就可以相应增加。理想情况下，节点间传输数据的时延、节点的成本不应该随节点数量的增加而上升；但是，正如我们即将讨论的，时延和成本的某些增加是不可避免的。

如果像图 7-3 所示的那样，存储器模块位于互连网络的另一侧，对网络带宽的需求随处理器的数量线性增长，即使在进程间没有通信发生也是如此。提供足够可扩展带宽可能对计算性能的可扩展性是不够的，因为随着处理器数量的增加，存取时延也相应增加。通过把存储器分布到各个处理器，所有的进程能够以固定的时延访问本地存储器，与处理器的数量无

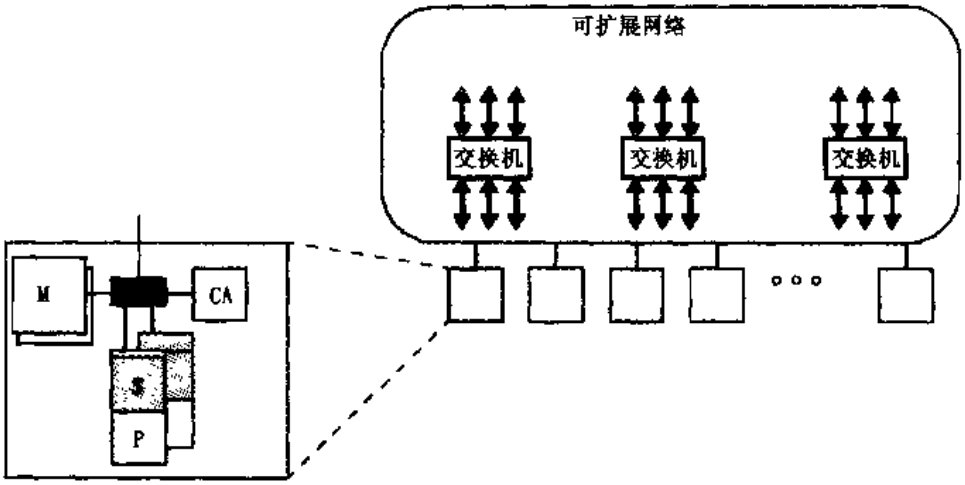


图 7-2　通用的分布存储器多处理器的组成。一组拥有自己的处理器和存储器的本质上完整的计算机，通过通用的、高性能的、可扩展的互连网络进行通信。典型地，每个节点包含一个能够辅助产生和接受通信操作的控制器

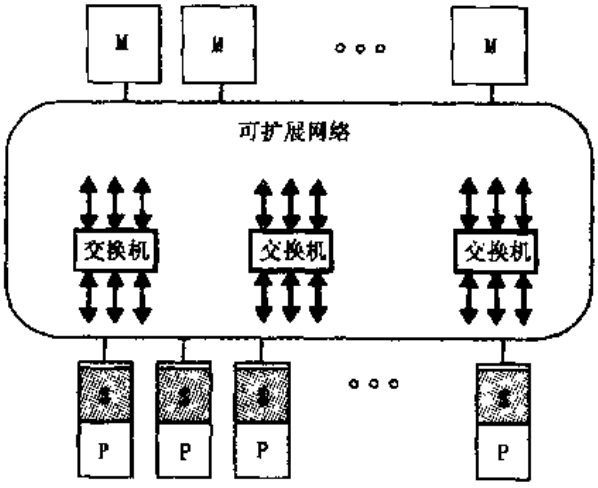


图 7-3　"舞池"式的多处理器组成结构。处理器经由可扩展的互连网络来访问存储器模块。即使处理器完全独立没有任何通信和共享，对网络的带宽要求也会随着处理器的数目增加而线性增长

关；所以，系统的计算性能能够完美地扩展，至少在进程间没有通信的简单情况下是如此。网络必须满足与实际通信以及信息共享相关的需求。在进程间确实通信的更有意义的情况下，计算性能如何扩展取决于网络本身的扩展性、通信体系结构的效率以及程序通信的方式。

为了实现可扩展的带宽，必须抛弃基于总线的设计中的几个关键假设，也就是说，只有有限数量的并发事务，它们是通过集中仲裁决定处理次序，而且是全局可见的。相反，非常多的并发事务可能使用不同的连线。它们独立地产生，没有全局的仲裁机制。一次事务处理的效果（比如状态变化）只被该事务处理涉及的节点直接可见（通过其他的事务处理的传播，其效果对其他节点最终也将是可见的）。尽管可以把信息广播给所有节点，*广播带宽*（即广播可以被执行的速率）并不随节点的增加而增加。这样，在大的系统中，广播不能经常使用。

457 ~ 459

7.1.2　时延的可扩展性

我们可以延伸可扩展网络的抽象模型，来考虑通信时延的基本要素。总的来讲，在两个节点间传输 n 个字节的时间是：

$$T(n) = \text{额外开销} + \text{通道时间} + \text{路由延迟} \tag{7-1}$$

这里，额外开销是开始和结束传输的处理时间，通道时间是 n/B（这里 B 是指“最窄通道”的带宽），路由延迟是路由步数（或跳数）和传输的字节数的函数 $f(H,n)$。

处理开销可能是固定的，或者当处理器必须要为传输而拷贝数据时，随 n 增加。对并行机器中使用的大部分网络来说，路由器一跳的延迟是固定的，独立于传输数据量的大小，因为消息直通穿过一系列交换机[1]。与此相比较，传统的数据通信网络在每一级队数据进行“存储－转发”，在每一跳产生与传输尺寸成正比的延迟[2]。在大规模并行机中，存储-转发路由是不实际的。因为网络交换机的度数有限，节点间的平均路由距离一定会随着节点数的增加而增加。所以，通信时延将随着规模的增加而上升。然而，当交换机足够快而且互连拓扑结构比较合理的时候，时延的上升和额外开销以及传输时间比起来相对较小（见例 7.1）。

例 7.1 很多经典的网络采用固定度数的交换机来构建，比如对 n 个节点的配置或拓扑结构来说，从任何网络输入到任何网络输出的距离是 $\log_2 n$，而交换机的总数是 $\alpha n\log n$，α 是小的常数。假设额外开销是每个消息 1 μs，链路的带宽是 64 MBps，路由器的延迟是每跳 200 ns。当系统从 64 节点扩展到 1 024 节点时，传输 128 个字节所用的时间将增加多少？

解答： 在 64 节点时，需要 6 跳，所以

$$T_{64}(128) = 1.0\ \mu s + \frac{128\ B}{64\ B/\mu s} + 6\times 0.200\ \mu s = 4.2\ \mu s$$

对一个 1 024 节点的配置，它将增加到 5 μs。这样在机器规模扩大 16 倍后，时延增加不到 20%。即使对于这么小的传输量，存储-转发模型将在每一个跳增加 2 μs（用来缓冲 128 字节的时间）。所以，时延将会是：

460

$$T_{64}^{sf}(128) = 1.0\ \mu s + \left(\frac{128\ B}{64\ B/\mu s} + 0.200\mu s\right) \times 6 = 14.2\ \mu s\ (64\text{ 个节点})$$

$$T_{1\,024}^{sf}(128) = 1.0\ \mu s + \left(\frac{128\ B}{64\ B/\mu s} + 0.200\ \mu s\right) \times 10 = 23\ \mu s\ (1\,024\text{ 个节点})\ \blacksquare$$

实际上，在例 7.1 中忽略了带宽和时延之间存在的一个重要联系。如果两个传输涉及到网络中相同的节点或者利用了相同的连线，由于竞争共享资源的缘故，一个传输可能会使另一个传输延迟。当使用更多的可用带宽时，发生竞争的概率和时延的期望值都会增加。此外，网络中将形成等待队列，进一步增加时延期望值。这种基本的饱和现象在任何资源共享的情况下都会发生。设计一个好网络的目的之一是为了保证这些与负载相关的时延在实际发生通信模式中不至于太大。但是，当很多处理器同时向一个节点传输数据时，就不可能没有竞争。该问题必须通过在较高级别，使用第 3 章所讨论的技术平衡通信负载来解决。

7.1.3 成本的可扩展性

对大的机器来说，成本的可扩展也非常重要。一般来说，我们认为它是一个固定的系统

---

[1] 可以把消息看作一列火车，火车头在沿铁轨的每个交叉点做出决定，而后面的车厢只是跟着走；当火车头做出新的拐弯决定时，某些车厢可能还在通过以前的交叉点。

[2] 存储-转发的方法类似于火车到达车站，整列火车必须停靠在一个站台上，装卸货物、乘客上下车之后，才能重新出发。所以，某节车厢在车站上停靠的时间线性正比于火车的长度。

基本配置的成本与系统添加的处理器和存储器的增量成本之和：

成本（p,m）= 固定成本 + 增量成本（p,m）

固定成本和增量成本两者都很重要。例如，在基于总线的机器中，固定成本典型地包括机箱、电源以及支持一个全配置的总线。和单处理器相比，一个小规模配置是不太合算的，但是鼓励扩展，因为增加处理器的增量成本是固定的，并且常常要比单机的处理器便宜很多。（有趣的是，对大多数商用的基于总线机器来说，典型配置一般只是最大配置的一半。这即充分的补偿了基本硬件的固定开销，并且还在总线上保留了“扩展的余地”。提供较大规模 SMP 系统机器的厂商（比如说，20 个或更多处理器配置的），也经常提供带有扩展槽的低端配置。）我们要强调成本方程式中存储器所增加的代价，因为存储器经常占据很大一部分系统成本，并且一台并行机器不一定要有 P 倍于单处理器的存储器。

经验表明，可扩展的机器一定要支持多样性的配置，不一定非得是大型机和巨型机。所以，对可扩展机器来讲，基于总线的机器这种“前面一笔付清（Pay-up-front）”模型就不太划算了。相反，基本硬件必须是模块化的，这样当加入更多的处理器时，再增加相应的电源和机柜以及更多的网络设施。对于具有好的带宽和时延扩展性的网络来讲，网络的成本将以高于节点数的线性函数的速度增长；实际中，这个增长的速率并不是太大的负担，正如例 7.2 中所说明的那样。 461

例 7.2 在很多网络中，对于 n 个节点，网络交换机的数量以 $n\log n$ 的规模增长。假设在 64 个节点的系统中，系统的成本在处理器、存储器、网络之间等量平衡，那么在一个 1 024节点的系统中有多大部分的成本用在网络上（假设每个处理器使用同样大小的存储器）？

解答： 我们可以将系统的成本标准化为 64 节点系统的单个处理器的成本。在 1 024 的大配置中，每个处理器所具有的路由器数量将会是小型系统中的 10/6 倍。所以，假设网络的成本正比于路由器的数量，在 1 024 节点的系统中，每个处理器的标准成本是：

1 个处理器 × 0.33 + 1 个存储器 × 0.33 + 10/6 个路由器 × 0.33 = 1.22

系统规模扩大 16 倍，网络所占的成本部分从 33% 增加到 45%。（实际中，其他的因素如线长的增加可能使网络成本的增加稍快于交换机的增长。）■

带宽如何随端口数的增加而增长，成本如何增长，网络延迟如何增长，在不同网络设计中也有很大不同，但它们无疑都要增加。对这三个因素有很多细微的权衡，而大部分情况下，带宽增加越大，时延的增加就越小，而成本的增加就越大。好的设计包括很多折中。最终，这些要根据目标负载的应用需求来决定。

最后，当我们考虑成本的问题时，很自然会问大规模并行机器是不是合算，或者是不是仅仅是一种能够达到高性能的手段。对效率的标准定义（效率（p）= 加速比（p）/p）反映了只有当所有的处理器在所有时间都被充分利用时，并行机器才是最有效率的。这个以处理器为中心的观念忽视了这样的事实：系统成本很大一部分在其他的部分，尤其在存储器系统中（Wood and Hill 1995）。假如我们以和加速比类似的方式定义了系统的成本可扩展参数成本上升比（costup），即 Costup（p）= cost（p）/cost（1），我们就会看到并行计算的性价比是好的，也就是说，当加速比（p）> 成本上升比（p）时，它具有较小的性能价格比。所以，在一个真实应用场景中，我们要考虑解决问题所需要的整体系统代价。

7.1.4 物理可扩展性

尽管普遍认为应该使用模块化结构来构建大规模的机器，但是在物理尺寸的特殊要求方面却很少有一致的意见，比如说节点的紧凑程度、连线长度的选择、时钟的策略等等。在某些商用机器中，单个的节点占用稍稍多于一个微处理器的空间；而在另一些情况下，一个节点将会占用主板的很大一部分甚至是占据整个工作站机箱。在某些机器中，线的长度不会超过几英寸；而在其他一些机器中，线长可以达到数英尺。有些机器中，链路只有 1 比特宽，但在其他一些机器中，有 8 比特或者 16 比特宽。总的来说，链路的速度随长度的增加而降低，并且每一种链路技术由于能耗需求和信噪比的因素都有自己的最大长度的限制。支持很长距离连接的技术，比如光纤，每条链路的收发器和连接器都有比较高的固定成本。因此，节点在物理空间上的紧密排放对扩展性有很大的好处。另一方面，松散的封装允许更多地采用市售部件，因此有利于降低工程上的努力，也降低了在大型并行计算机中采用新的微处理器和存储器技术的时间滞后性。因此，松散封装就有了更好的技术可扩展性。物理封装策略间的众多复杂的折中产生了众多的设计，所以最好看一些具体例子。这些例子也有助于使扩展的其他方面更为具体。

462

1. 芯片级集成

一些设计把通信体系结构直接集成在处理器芯片上。nCUBE/2 是这种密集封装节点方法的很好的代表，尽管这种机器本身已经是相当老。这种高度集成的节点方式在 MIT 的 J-machine（Dally，Keen，and Noakes 1993）和很多其他研究性机器及嵌入式系统中也被采用。在芯片密度继续增加的情况下，这种设计风格将会变得更加流行。

在 nCUBE 中，每个节点拥有处理器、存储器控制器、网络接口和网络路由器，集成在单个芯片上。节点芯片直接和 DRAM 芯片以及 14 个双向网络链路相连，所有的部件都集成在一个几平方英寸的板卡上，图 7-4 显示了它的真实尺寸。网络链路形成了直接和其他节点连接的位串行的通道○以及一个到 I/O 系统的双向通道。28 条线中的每一条在节点芯片上都有一个专用的 DMA 设备。节点插到主板的插槽上，每块主板 64 个节点。而主板插到无源的连线背板上，每个处理器芯片和其他 log n 个处理器之间形成了直接的节点对节点的线对。每个处理器一个的 I/O 链路从主机架引出，连接到包含一个节点芯片（连接到 8 个处理器）和一个 I/O 设备的 I/O 节点上。最大的配置是 8 096 个处理器，1991 年建造了具有 2 048 个节点的机器。在所有配置中，系统运行在 40 MHz 的单时钟下。因为有些线需要横跨整个机器的宽度，所以密集封装对于限制最大线长而言是关键的。

nCUBE/2 应该被理解为当时的一项重要设计。节点芯片大约包含 50 万个晶体管，在当时是很多的。处理器类似于一个精简的 VAX 机器，运行在 20 MHz 的频率下，具有 64 位的整数运算和 64 位 IEEE 浮点运算，峰值运算速度 7.5 MIPS 和 2.4 MFLOPS 的双精度运算。而通信支持部件占用了同代单处理器本来应该用于高速缓存的硅片面积；指令高速缓存只有 128 字节，数据高速缓存可以容纳 8 个 64 位的操作数。网络链路的宽度为 1 位，每个 DMA 通道运行在 2.22 MBps。

463

○ nCUBE 节点以超立方体结构连接。一个超立方体，或称 n 立方体，是图 7-4 所示的立方体的一般化形式，在一个 n 节点的配置中每个节点直接和 logn 个其他节点连接，所以，13 条链路可以支持多达 8096 个节点的设计方案。

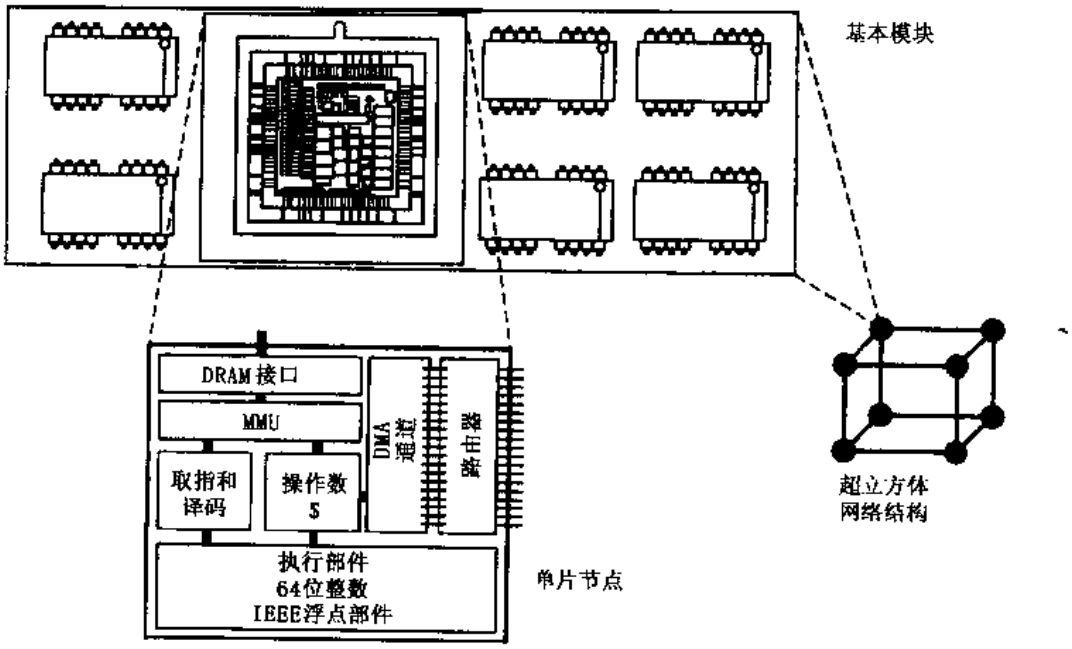


图 7-4　nCUBE/2 机的组成结构。这种设计基于一个紧凑的包含单片节点的模块，该模块包括处理器、存储器接口、网络交换机和网络接口，直接与 DRAM 芯片和其他的节点相连

就时延可扩展而言，nCUBE/2 比我们的直通模型要复杂一点。一条消息可以包含任意数量的 32 位的字，发往任何目的节点，其中第一个字是目的节点的地址。消息以一系列 36 位的块的形式被路由到目的节点，每个块中 32 位是数据，另加 4 位是奇偶校验位，每一跳的路由时延是 44 个周期（2.2 μs），传输时间是每字 36 个周期。对 n 个节点的网络，跳数最多为 $\log n$，平均距离是它的一半。作为对照，J-machine 把节点组织成三维的网格（实际上是三维花环），这样每个节点就用很短的连线和六个相邻节点连接。链路相对较宽（8 位）。这种组织结构中最大跳数大约是$\frac{3}{2}\sqrt[3]{n}$，平均长度是它的一半。

464

2．板级集成

大规模机器设计的最常用的硬件策略使用标准的处理器部件并将它们在板级集成，获得中等密度的封装。这种方法的代表包括 Caltech 的超立方体机器（Seitz 1985）和 Intel 的 iPSC 和 iPSC/2，它们本质上是在每个节点中放置了一个早期个人计算机的内核。Thinking Machines 的 CM-5 复制了 Sun Sparc Station1 工作站的核心部件；CM-500 复制了 Sun Sparc Station10；而 CRAY T3D 和 T3E 基本上复制了 DEC Alpha 工作站的内核。大多数近代的机器在一块主板上放置几个处理器，在某些情况下，每个主板包含一台基于总线的多处理器。例如，Intel ASCI Red 机器就有 4 000 个以上的 Pentium Pro 双路多处理器。

在 1993 年左右，Thinking Machines 的 CM-5 是板级方法的一个很好的代表。图 7-5 显示了 CM-5 的基本硬件组成。节点基本上由一个当时的工作站的部件所构成，即一个 33 MHz 的 Sparc 微处理器、它的外部浮点部件以及连接到基于 MBUS 的存储器系统的高速缓存控制器[⊖]。网络接口是Sparc MBUS上的一个附加的专用集成电路。每一个节点与两个数据网络、一

⊖ CM-5 使用专用的存储器控制器，该控制器包含专用的、存储器映射的向量加速器。这种设计源自 CM-2 的 SIMD 风格，并且是伴随物理机器的伸缩而来的。

个控制网络、一个诊断网络相连接。网络组织成4叉的树，其叶子是处理节点。一块板包含4个节点和一个把这些节点连结成为树的第一层的网络交换机。为了提供可扩展的带宽，CM-5使用了一种多根树，或称为胖树（将在第10章详细讨论），它在每一层具有同样数量的网络交换机。每个板包含了构成16个节点的网络的第2层的4个网络交换机中的一个。网络树的更高的层次则位于额外的网络板上，这些板通过电缆连在一起。一个机柜上安装几块板，但是对于具有几千个节点规模的系统来说，就要进一步把多个机柜用大的线束连接起来。此外，使用几个机架的路由器来完成整个的互连网络。网络中的链路宽度为4位，时钟频率40 MHz，提供的峰值带宽为12 MBps。路由延迟是每跳10个周期，最多会有$2\log_4 n$跳。

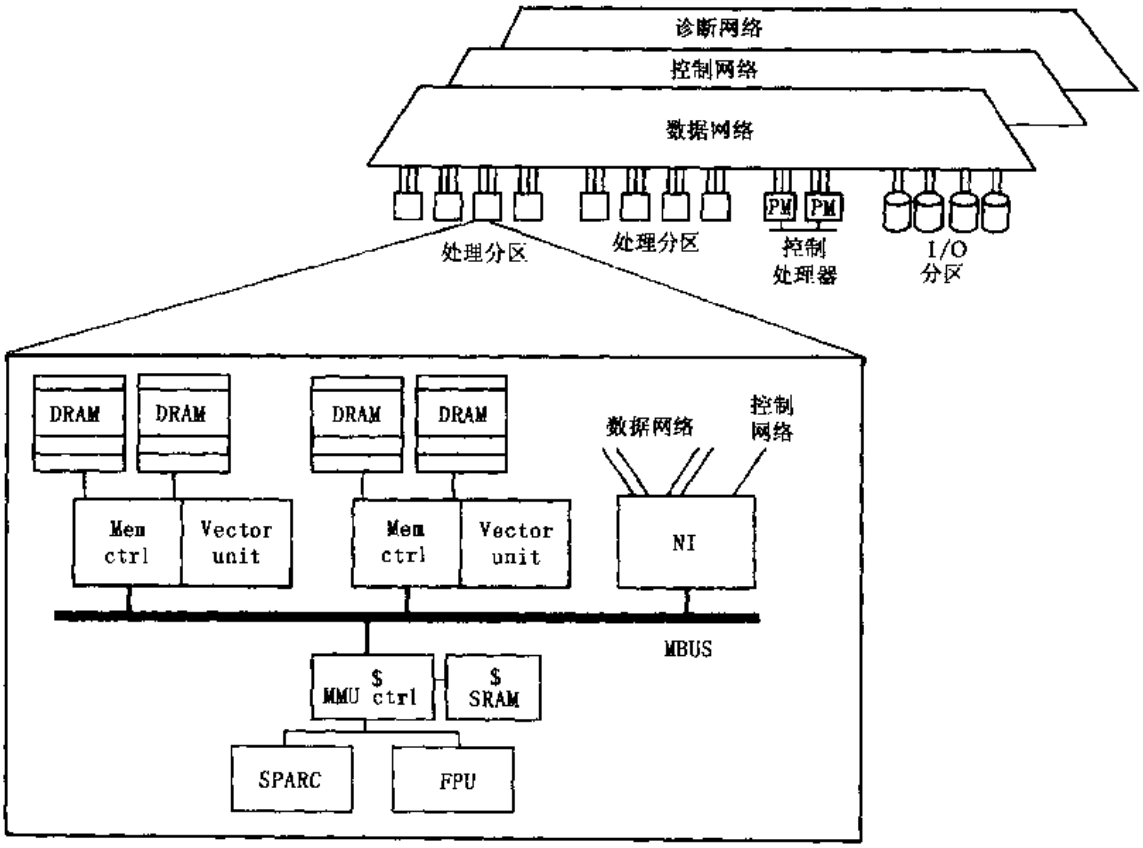


图7-5　CM-5机的组织结构。每一个节点是一个重新封装的Sparc Station芯片组（包括处理器、MMU、高速缓存、存储器控制器和DRAM）和一个MBUS上的网络接口芯片。网络（数据、控制、诊断）形成了一个“可扩展的背板”，把计算分区和I/O节点连接起来

CM-5网络提供了一种可扩展的背板，以支持多个独立的用户划分以及一些I/O设备。尽管存储器被分配给了每个处理器，I/O却采用了“舞厅”的方式来解决。处理器通过网络以便一致地访问专用的I/O节点集合。其他机器使用了类似的板级集成方法，如Intel的Paragon和CRAY T3D和T3E，把主板以类似网格的形式连接以保持连线较短，并且使用更宽更快的链路（Dally 1990b）。I/O节点通常位于网格的面或占用了立方体的内面。

465

3．系统级集成

为了减少采用最新的微处理器和操作系统的工程化的时间，一些现代的大规模机器使用了较松散的装配技术。IBM可扩展并行系统设计（SP-1和SP-2）就是很好的代表；它把多个近乎完整的RS6000工作站装入一个机柜。因为完整的标准系统用作节点，通信辅助部件和

网络界面就成为了系统插卡的一部分。对 IBM SP 来说，是使用微通道适配卡连接到位于机柜的基座的交换机上。每条网络链路工作于 40 MBps。一个机柜装有 8 到 16 个节点，所以通过连接很多个机柜，其中包括一些交换机机柜来完成大的系统配置。由于每个节点都是一个完整的系统，可以选择将磁盘和其他 I/O 设备分布到整个机器上。在 SP 系统中，计算节点的磁盘通常只用作交换空间和临时存储。大部分的 I/O 设备集中在专用的 I/O 节点上。这种设计方式允许节点间一定程度的异构性，因为只要网络接口卡可以插入就行了。例如，SP 系统支持多种工作站模型作为节点，甚至包括 SMP 系统节点。

466

为大规模并行机开发的高速网络技术已经被为数众多的局域网所采用。比如，ATM（异步传输模式）就是一个可扩展的、基于交换机的网络，支持 155 MBps 和 622 MBps 链路。还有连接多个 100 MBps 环路的 FDDI 网络以及基于交换机的光纤通道（FiberChannels）和 HPPI。很多厂商提供了基于交换的 100 MBps 的以太网，基于交换的千兆以太网正在出现。此外，很多高带宽、低时延的系统域网络（SAN）可以在较短的物理距离上运行，它们已经被商品化，如 Myrinet、SCI 和 ServerNet 等。这些网络和传统的大规模多处理器网络非常相似并且允许传统的计算机系统通过一个可扩展、低时延的互连网络集成为一个“机群”。在很多情况下，作为节点的单台机器是小规模的多处理器。因为节点是完备的、独立的系统，这种方法被广泛使用来提供高可用性的服务，比如数据库，当一个节点失效时，它的那部分工作由其他节点代为完成。

7.1.5　通用并行体系结构的可扩展性

在几个公司日常生产具有成百上千高性能的处理器系统的今天，大规模并行机工程实现上的挑战已经被广泛认识。在物理上密集集成和使用新的微处理器技术的工程化所需时间之间是有矛盾的，因而产生了很多种装配的解决方案。这些方案概念上的组成结构和图 7-2 所示的通用并行体系结构是相同的。可扩展互连网络的设计是并行体系结构的一个重要而有趣的子领域，在过去几年来有了巨大的进步，对此，我们将在第 10 章中进行讨论。已经推出了一些好的商品化的网络产品，它们提供了高的带宽和低的时延，而且时延随着系统规模扩展增长缓慢。而且，这些网络提供了可扩展的集合带宽，允许大量的传输同时进行。它们的硬件传输错误率很低，有时和现代的总线错误率相当，所以是健壮的。这些设计中的每一种都因为带宽、时延、成本等因素而具有固有的扩展限制。但是从工程的观点来看，今天对构建一个机器的规模限制主要来自经济上的可行性。

在实际中，最大目标规模对于评价一个设计权衡来说非常重要。它决定了设计的级别和系统每一个方面的工程努力。例如，为了构造非常大的系统，达到高的封装密度和高度的模块化所付出的工程努力，对于中等的系统来说就不太合算了，因为不太复杂的方案就可以满足需要。一个实际设计寻求计算性能、通信性能、机器建造的当时成本之间的平衡。比如，较好的网络性能或更高的物理密度可以通过把网络和处理器更紧密的集成来实现。然而，这可能会提高成本，也可能由于增加了处理器到存储器的时延或延长了设计时间而牺牲了性能，所以，可能不是最有效的选择。由于处理器的性能随着时间不断快速提高，在技术曲线的较后位置开始的一个更为基本的设计，可以实现较高的计算性能，但通信性能可能较低。正像设计的所有方面一样，问题在于如何平衡。

467

大规模并行机的一个共同特点是允许很多传输同时进行，本质上没有全局的信息和全局

的仲裁，通信时间主要来自于节点到网络的接口。这些问题主导了我们从体系结构角度出发的思考。单单硬件能力的可扩展是不够的；整个系统解决方案必须可以扩展，包括实现编程模型的协议以及操作系统所提供的能力，比如进程调度、存储管理、I/O 等。即使硬件本身能很好地扩展，由于锁的竞争锁和应用中资源共享所导致的串行化甚至操作系统本身，也可能会限制系统有用的扩展。假如以性价比好的方式，能够满足将系统实际扩展到要求规模的工程上，还必须保证支持目标编程模型的通信和同步操作的可扩展性，同时维持一个足够小的固定性价比。

7.2 编程模型的实现

本节，我们将考察如何在大的分布存储器型机器上实现编程模型。从历史角度看，这些机器和基于消息传递的编程模型密切相关，但是共享地址空间的编程模型正变得越来越重要。第 1 章介绍了通信抽象的概念，它定义了提供给用户的通信原语集合。这些可以由硬件直接实现，或由系统软件实现，或者是两者的结合，我们所熟悉的图 7-6 说明了这一点。它把我们的注意力集中到支持通信的节点结构方面。在小规模的共享存储器的机器中，通信抽象被当作存储器接口的扩展由硬件直接支持。在一致的共享存储器抽象模型中，根据由状态机集合所定义的特殊协议，装入和存储操作通过一系列基本的总线事务来实现。

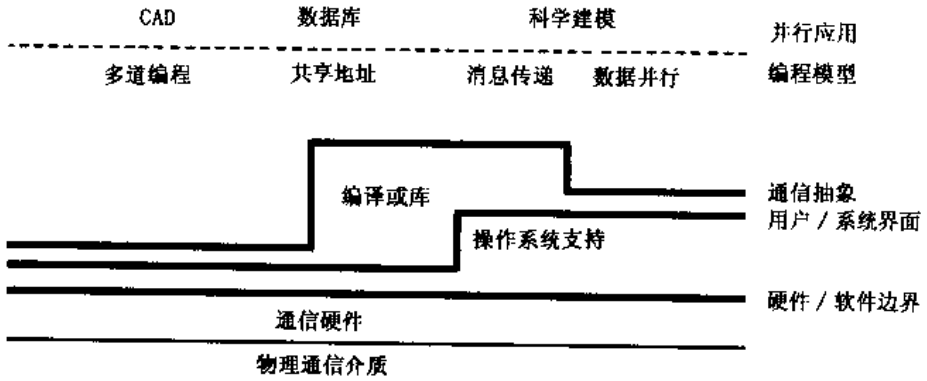


图 7-6 并行体系结构的层次。这个图表示了抽象的关键层，以及实现每一层的系统设计的各个方面

在大规模并行机中，编程模型用类似的方式来实现，只是基本的事件是跨越网络的事务，即网络事务，而不是总线事务。一个网络事务是从源输出缓冲区到目的输入缓冲区的单向传输，并在目的节点引发了某种从源节点不能直接看到的动作。如图 7-7 所示。这种动作可能非常简单（比如，把数据存入可以访问的位置，或者一个有限状态机发生一次状态转换），或者可能更具有一般性（比如，执行一个消息处理例程）。网络事务的效果只能通过其他事务观察到。传统的、大规模机器的硬件所直接支持的通信抽象隐藏在厂商的消息传递库之下，但是一个趋势是，用户的应用越来越可以访问较低层的抽象。

总线和网络事务间的差别有着意义深远的结果。潜在的设计空间比我们在第 5 章中所看到的还要大，在那里总线提供了串行和广播的方式，而基本事件在传统的总线事务中变化很小。但在大规模的机器中，基本网络事务本身变化很大，这些事务在端点如何被驱动和如何被解释也是如此。它们可以对处理器表达为 I/O 操作并完全由软件来驱动，或者可以集成到存储器系统中被专门的硬件控制器所驱动。已经设计和实现了很多种类的大规模机器，强调

468 ~ 469

了一系列编程模型，在硬件/软件边界上使用了一系列原语，提供了不同程度的硬件直接支持和系统软件干预。

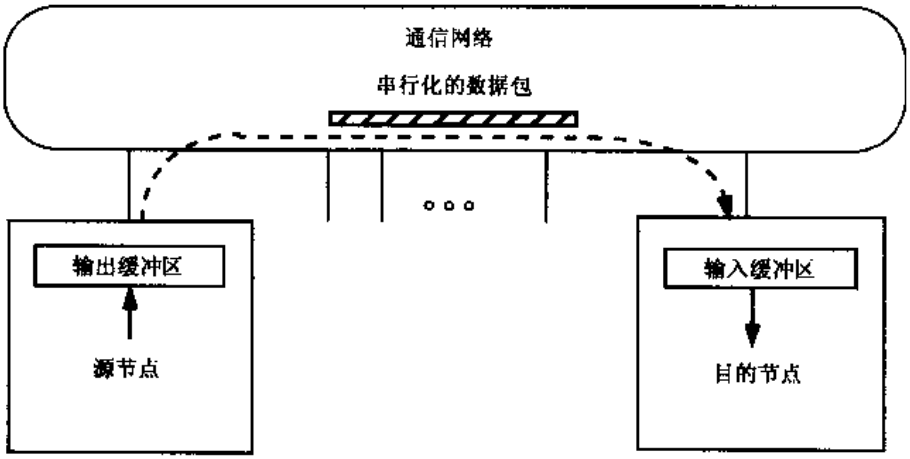


图 7-7　网络事务原语。一个从源节点输出缓冲区到指定的目的节点输入缓冲区的单向的信息传输，引起目的端的某种动作

为了弄明白这些设计的多样性，我们将循序渐进地展开。本节首先更加精确地定义网络事务并将其和一个总线事务仔细地做对比。然后概括了利用这些原语实现共享存储器和消息传递抽象所涉及的方面，但不要受到网络事务本身无数实现方式的阻碍。在后面的章节中，将系统地考察实现网络事务以及建立在它们之上的编程模型的设计选择空间。

7.2.1　基本的网络事务

为了理解网络事务中所包含的内容，让我们先再次考察总线事务，两者之间有很多相似性。在开始一个总线事务之前，作为虚-实地址翻译的一部分，先进行保护检查。总线事务中的信息格式由总线的物理连线所决定，即包括数据线、地址线和命令线。将要传送到总线上的信息在它们可以被送到总线之前，在特定的输出寄存器中放置（包括地址、命令、数据寄存器）。一个总线事务以对介质的仲裁开始。大多数总线使用全局的仲裁策略：一个请求总线事务的处理器在总线请求线上给出信号电平，并等待相应的总线许可信号。事务的目的地址隐含在地址中。总线上的每个模块都配置成能够对应一组物理地址。所有的模块检查地址信息，其中有一个会对该事务做出响应（如果没有模块响应，总线控制器将检测到超时信号并中止该事务）。每一个模块包括一组输入寄存器，能够缓冲它可能做出响应的所有请求。每一个总线事务包括一个请求和随后的一个响应。在读的情况下，响应包括数据和一个相关的结束信号；对于写，则只有完成确认。在每一种情况下，源和目的都被通知事务的结束。在很多总线中，根据一个良好定义的调度来保证每一个事务的结束。主要的差异是从目的地返回响应时间的长短。在事务拆分型总线中，事务的响应阶段可以要求重新仲裁并且有可能以不同于请求的顺序来执行。

在事务拆分型（和每个操作包括多个总线事务的一致性协议）中必须注意避免死锁，因为在总线上的一个模块可能在同时请求和服务于事务。模块必须继续为总线请求提供服务，并且在它力图提出自己的请求时接受应答。总线设计保证了这一点，对任何可能放置在总线

470

上的事务来讲，在目的节点要有足够的输入缓冲空间来接受事务。这可以通过为最坏的情况提供足够的资源的保守方式来解决，或者通过添加否认信号（NACK）以乐观的方式实现。在这两种情况下，解决方案都相对简单，因为总线上很少有并发的通信同时进行，源和目的直接用线耦合。

对总线的讨论也引发了网络事务中的一些问题。这些问题包括保护、格式、输出缓冲、介质仲裁、目的地命名及路由、输入缓冲、动作、结束检测、事务排序，死锁避免以及传输保证。网络事务和总线事务的基本区别是，前者的源与目的是非耦合的；也就是说，它们之间可能没有直接的连线，在系统中也没有对资源的全局仲裁。在同一个时刻，对所有的模块没有全局的信息可用，大量的事务可能同时进行。这些基本差别使得上述问题具有与总线情况非常不同的性质，下面让我们依次考虑每一个问题。

- 保护。当部件的数目较大，部件间的耦合较为松散，单个部件更加复杂时，限制每个部件对其他部件的正确操作的信任程度可能是有意义的。尽管在一个基于总线的系统中，处理器在把事务放到总线之前进行所有的保护检查；但在一个可扩展系统中，单个部件将经常对网络事务进行检查，这样一个错误的程序或硬件部件的失效就不会导致系统其他部件的崩溃。
- 格式。大多数网络链路较窄，所以和一个事务相关的信息以串行流方式传输。典型的链路的宽度只有几位（1～16 位）。事务的格式由信息在传输链路上串行的方式来决定，而不像总线方式中以并行的方式传输，其格式由物理连线决定。这样，设计中就有了很大的灵活性。我们可以把网络事务的信息看作内部装有更多信息的一个信封。信封包括和物理网络相关的信息，用来把包从源发送到目的端口。这非常像总线事务中的命令和地址部分，告诉涉及的所有方面如何处理事务。一些网络被设计成只能发送固定大小的包；而其他的一般可以发送变长包。信封中经常进一步包含其他信封。通信辅助部件可以把用户信息封装到和远处的通信辅助部件密切相关的信封中，然后将它再封装在物理网络的信封中。这种把信息包放置在更大信封中的概念和传统网络协议栈的封装很相似。这提供了一种对通信子系统的层次进行抽象的方式。

471

- 输出缓冲。源必须提供容纳将要被串行送到链路上信息的存储空间，这可能在寄存器、FIFO 或者存储器中。当事务是固定格式时，它可以像总线的输出缓冲区一样简单。因为网络事务是单向的并且具有流水的潜力，所以希望提供一个输出寄存器的队列。如果包的格式在某个尺寸范围内变化时，可以采取一种简单的方法，允许输出缓冲区中的每个项的尺寸可变。假如一个包很长，典型的做法是，输出控制器包含一个描述符缓冲区，描述符指向存储器中的数据。然后，它分阶段把存储器中的包的一部分传送到小的输出缓冲区中进而送到链路上，经常使用 DMA 的传输方式。
- 介质仲裁。对网络的访问没有全局的仲裁，可以同时启动很多网络事务（在类总线的网络，比如以太网中，具有对一个或少量同时发生的事务分布式的仲裁）。启动一次网络事务提出了对源和目的间通信路径资源以及对目的资源的隐式请求。这些资源可能是和其他的事务共享的。在源端进行局部的仲裁，来决定是否应该开始该事务。然而，这并不意味着通往目的节点的所有需要的资源已经预留了；资源是在消息前进时逐步分配的。

- 目的地命名和路由。源必须能够指定足够的信息来使得事务可以被引导到合适的目的地。这和总线形成了对比，总线方式中，源只是简单地把地址放置到连线上，由目的地来选择是否应该接受这个请求。有很多不同的路由指定和执行的方式，但是基本上，源执行从目的地的逻辑名到某种形式的目的物理地址的转换。
- 输入缓冲。在目的地，网络事务上的信息一定要从物理链路传送到一些存储单元中。与输出缓冲区一样，这可能是简单的寄存器或队列，或可能直接传到存储器中。关键的差别是，事务可能来自很多个源；与此对照，源对于它能发出多少事务有完全的控制。在某种意义上，输入缓冲区是由很多远地处理器共享的资源；对它如何管理以及在它被装满后该怎么办是我们稍后要讨论的关键问题。
- 动作。在目的地发生的动作可能非常简单，比如，一次存储器访问，它也可能相当复杂。不管是哪种情况，它都会产生一个响应。
- 结束检测。源具有事务已经被发送到网络上的指示，但是通常没有表明它们已经到达了目的地的指示。结束状态必须通过响应、确认或某种进一步的事务来获得。 472
- 事务排序。尽管总线提供了很强的对事务进行排序的特性，但在网络中排序的能力很弱。即使在一个具有多个未决事务的分裂事务总线上，我们也可以依靠对地址总线的串行仲裁提供全局的次序。某些网络保证，从给定的源到单一目的地的一系列事务，可以在目的地顺序地接收到；其他的网络甚至对这种有限的保证都无法做到。无论是哪种情况，没有节点可以察觉到全局的顺序性。在大规模机器上实现编程模型时，必须通过网络事务强加次序的约束。
- 死锁避免。大多数现代网络在网络模块能够在持续接受事务的情况下无死锁。在网络上，这可能需要限制允许的路由或采取其他的特殊预防措施，正像我们将在第10章讨论的。当然，我们必须小心，我们为了实现编程模型而使用网络事务不要引入死锁。特别地，当我们等待，不能够产生事务时，通常需要能继续接受进入的事务。除了同时出现的事务数目很多和没有全程仲裁或立即的反馈以外，这种情况非常类似于“分裂事务”总线。
- 传输保证。可扩展网络设计中一个基本的决策是当目的缓冲器满时的行为。这显然是一个基于端到端的问题，因为对于源来说，要想在试图开始一个事务前，了解目的端的输入缓冲是否可用并不是一件容易的事情。对于网络本身来说，这也是一个基于链路到链路的问题。我们有两个基本的选择：当缓冲区满时抛弃信息或者推迟传输直到有空间时为止。第一种方法要求能够检测到缓冲区满并且重试的方法；第二种方法则要求一个流控机制并能导致事务堆积。将在本章后面考察这两种方法。

总而言之，网络事务是一个从源输出缓冲区到指定的目的地的输入缓冲区的单向信息传输过程，使得目的端发生某种动作。让我们接着考虑，根据这些原语，实现常用的编程模型中的通信抽象时所涉及的问题。

7.2.2 共享地址空间

实现共享地址空间的通信抽象根本上要求一个双向的请求-响应协议，如图7-8所抽象地说明的那样。一个全局的地址被分解成一个模块号和一个局部地址。对于一个读操作，请求发送给指定的模块，请求读取所期望的地址并且指定足够的信息来把结果通过一次响应网

473 络事务返回给请求者。一个写操作与读操作很相似，只是数据通过地址和命令送到指定的模块，而应答不过是把写操作已经执行的确认返回给请求者。响应则通知源，请求已经被接受或者被服务，这取决于它在远程事件之前还是之后被产生。这种响应是保证合适的事务次序的基本需要。

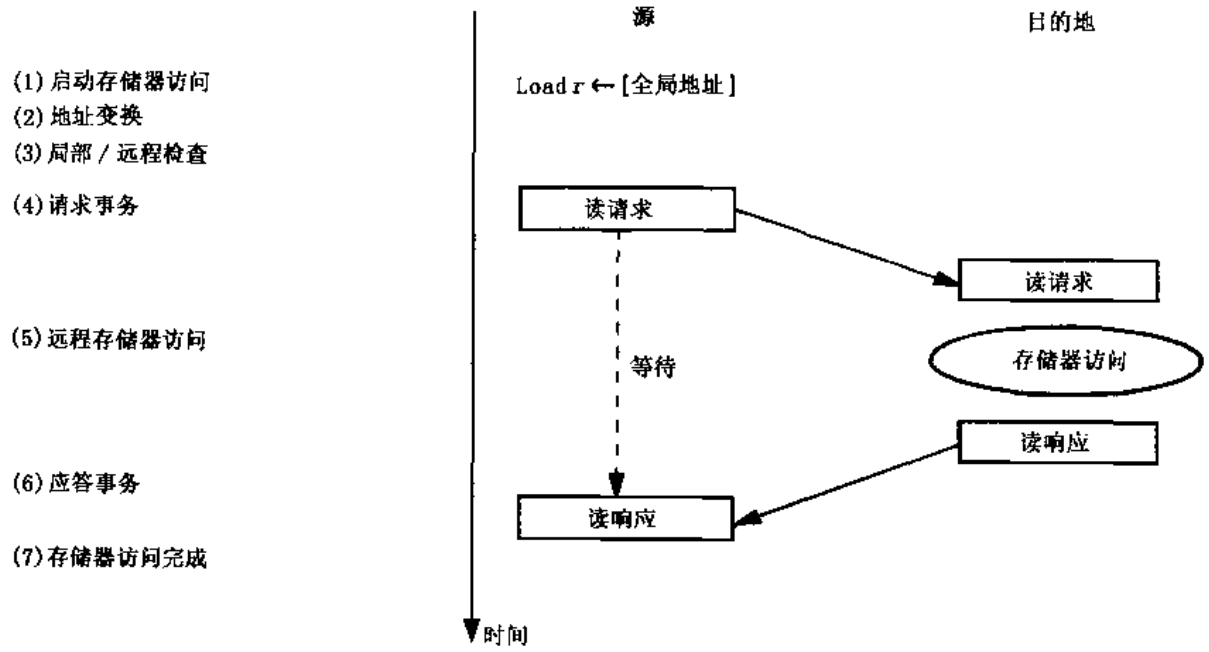


图 7-8 共享地址空间通信抽象。该图说明了基于基本的网络事务，对大规模机器中读操作的剖析。1）源处理器启动对全局地址的存储器访问。2）全局地址被翻译成节点号（或路由）和该节点上的局部地址。3）执行一个检查以决定该地址对于发出的处理器是否是局部的。4）如果不是，就执行一个读请求事务，把请求发送给指定处理器，5）访问指定的地址，6）通过一次应答事务把值返回给原来的节点，7）存储器访问完成

一个读请求通常具有简单的固定格式，描述要读的地址和返回的信息。写确认也比较简单。如果只支持固定尺寸的传输（即一个字或一个高速缓存块），读响应和写请求也是简单的固定格式。这可以很容易地扩展以支持部分字的传输，比如引入字节使能；然而，任意长度的传输要求一个更加通用的格式。对于固定格式传输来讲，输出缓冲通常和总线的情况一样。地址、数据和命令在输出寄存器上顺序分级，串行化地送上链路。

目的地名通常由地址翻译过程的结果决定，它把一个全局地址转化为一个模块名（或一个到该模块的路由）以及一个模块的局部地址。成功的翻译通常意味着获得对指定目的模块的访问；然而，源必须仍然要获得对物理网络和输入缓冲区的访问。因为大量的节点可能对同一个目的端发出请求，并且在源和目的之间没有全局的仲裁和直接的耦合；请求的存储需求的和可能会超过节点的输入缓冲区。请求可以被处理的速率只是单个节点的处理速率，所以请求可能在网络中堆积，甚至一直堆积到源。另一种做法，请求可能被丢掉，要求重传的机制。因为当节点在发送的时候，网络可能无法接受一个请求，每一个节点即使在它不能够向网络注入自己的请求时，也必须能够接受应答和请求，以使得网络上的包得以继续前进。这是在前一章节已经看到的读死锁的一个更加普遍的形式。这种输入缓冲区问题和取死锁问题在很多不同的通信抽象中都有发生，所以在考察消息传递抽象的协议后，对其进行更详细的讨论。

474

当支持一个共享地址空间的抽象时，我们需要询问它是否是一致的，支持什么样的存储器同一性模型。本章，我们考虑不自动通过高速缓存复制数据的设计；第 8、9 章将集中讨论这个问题。因此，每一个远程读和写到达那个具有被访问地址的节点并对该位置进行读写，一致性是由穿越网络和访问存储器的自然的串行化实现的。一个重要问题是，远端节点来的访问要和本地节点的访问相一致。这样，如果共享的数据在本地高速缓存，远端访问的处理就要在这个节点满足高速缓存的一致性。

在可扩展机器中得到顺序的一致性比在基于总线的设计中更具有挑战性，因为互连网络没有把对不同节点存储器的访问串行化。而且，因为网络事务的时延较大，我们将会尽可能的将它隐藏起来。特别是，在不等待返回完成确认而开始多个写事务，是非常诱人的。为了了解这将如何破坏一致性模型，考虑我们所熟悉的基于标记的代码段在具有物理的分布存储器但无高速缓存的多处理器上执行的情况。变量 A 和 flag 分配在两个不同的处理节点上，如图7-9a所示。由于网络的延迟，处理器P_2看到的对A和flag的写入次序可能和它们产生的顺

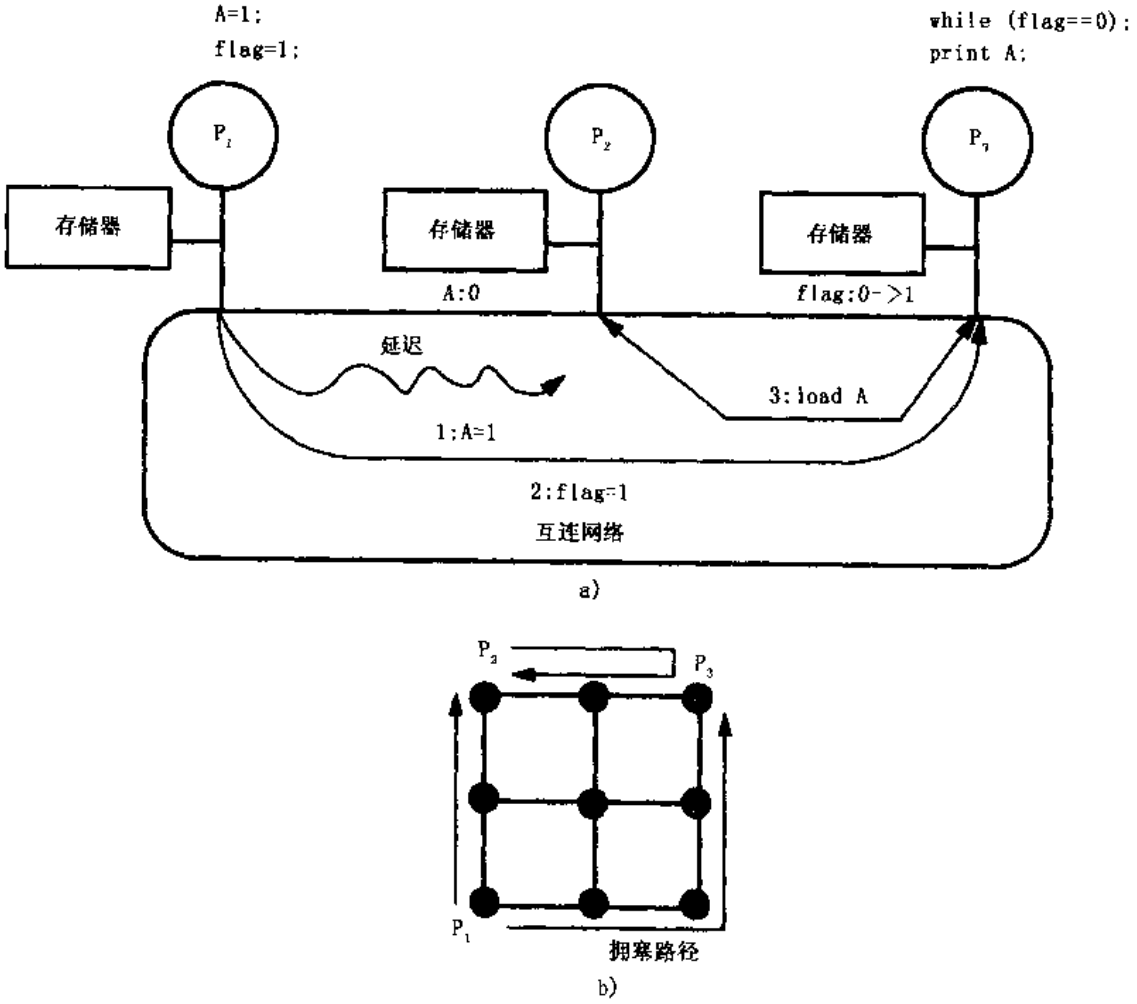


图 7-9　对于共享标记的存储器访问的可能的重排序。假设网络保持一个点到点的次序。处理器没有高速缓存如图中 a) 部分所示。变量 A 假设分配在 P_2 的存储器内，而变量 flag 假设在 P_3 的存储器内被分配。假设所有的处理器不会被存储指令暂停，就像大部分的单处理器系统那样。很容易看到，假如从 P_1 到 P_2 的链路上有网络阻塞，P_3 可能从 P_1 得到更新过的 flag，而从 P_2 读到 A 的旧值（A =0）。这种情况很容易发生，就像图中 b) 部分所示的那样，消息在 2D 网格中总是先走 *X* 方向路由，再走 *Y* 方向

序相反。在每一对节点间保证包的点到点的顺序并不能解决这个问题，因为可能要涉及到多对的节点。因为使用网络的不同路径而造成的可能的重排序情况在图 7-9b 中显示。写操作需要确认的原因之一就是要解决这个问题。这种结构的一种正确的实现方法是在发出 flag 的写之前等待写 A 操作的完成。通过使用写的完成和读应答，满足顺序一致性的充分条件就是简单的了。更深层的设计问题是如何在减小等待时间的情况下来满足这些条件，做法是判断写操作已经提交，并且对所有的处理器来讲，它似乎已经被执行了。

7.2.3 消息传递

在消息传递模型中的一个发送/接收对在概念上是一个从源用户进程指定的源区域到目的用户进程所指定的目的区域的单向传送。而且，它包含了两个进程间的成对同步事件。在第 2 章，我们注意到基本的消息传递抽象中的重要语义的变型，比如同步和异步消息发送。用户级的消息传递模型根据基本网络事务实现，不同的同步语义有相当不同的实现（即不同的网络事务协议）。在大多数早期的大规模机器中，这些协议隐藏在厂商的内核和库软件中。在较现代的机器中，原语事务变得对外可见，以便能够支持更多的编程模型。

475
476

本章使用和消息传递接口（MPI）相关的概念和术语。MPI 把发送或者接受调用的返回和一个消息操作的完成区别开来。一旦一个相匹配的接收被执行，一个同步发送也就完成了，源数据缓冲区就可以被重用，并且保证数据已经到达了目标的接收缓冲区。一个带缓冲的发送一旦在源发送缓冲区可以重用时就结束了，不管相对应的接收操作是否已经开始；数据可能已经被传输或者可能在系统中某处被缓存[○]。缓冲的发送结束和接收进程是异步的。当消息数据已经到达接收端的目的缓冲区时，一次接收过程完成。一个阻塞式发送或接收函数，只有当消息操作完成时才返回。一个非阻塞的函数则立即返回，不管消息完成与否，并且进一步调用探测函数来检测是否完成。协议只关心消息操作和完成情况，而和函数是否阻塞无关。

为了理解从用户消息传递操作到机器的网络事务原语的映射，让我们先考虑同步消息。执行源进程的处理器知道匹配的接收是否已经执行的惟一方法是，信息通过一种显式的事务被传送。从而，同步消息操作可以通过一个三阶段的网络事务协议来实现，如图 7-10 所示。这个协议是用于一次发送者启动的传输。发送操作使得一个“准备好发送”信号传送给目的地，携带了源进程和标记信息。发送者然后等待，直到对应的“准备好接收”信号到达。远端的动作是检查一个局部表，从而决定对应的接收是否已经执行。假如还没有执行，就把“准备好发送”信息记录在表中，以等待匹配的接收。假如发现了一个匹配的接收，就产生一个“准备好接收”的响应。接收操作检查同一个表。如果匹配的发送还没有在那里记录，则接收就被记录下来，包括目的数据地址。假如一个匹配的发送已经存在，接收产生一个“准备好接收”事务。当一个“准备好接收”到达了发送者，它就可以开始一次数据传输。假设网

477

○ 标准的 MPI 模式是缓冲模式和同步模式的混合，于是给实现带来很大的自由，但给程序员很少的保证。实现可以自由选择对数据进行缓冲，但是不能假定它一定这样做。因此，当发送完成时，发送缓冲区可以被重用，但不能假设接收者已经到达了接收调用的地方。也不能假设发送缓冲能够打破两个节点由于互相发送然后再接收信息所造成的死锁，非阻塞的发送可以用来避免死锁，即使在同步发送的情况下也可以。就绪模式的发送是同步模式的一种更强的变型，当消息到达目的端时如果接收还没有执行就将报错。因为得到非局部进程状态的惟一方式是通过消息的交换，需要用一个显式的消息事件来表明是否已经就绪。在宣布就绪的接收和传送同步消息之间的竞争条件和共享地址空间情况的标记的例子非常相似。

络是可靠的，一旦所有的数据已经传输，发送操作就结束了。接收操作在数据到达后就会结束。注意，根据这个协议，源和目的节点两者在实际数据传输发生时，知道源和目的地缓冲区的局部地址。“准备好”事务是小的、固定格式的包，而数据是可变长传送的。

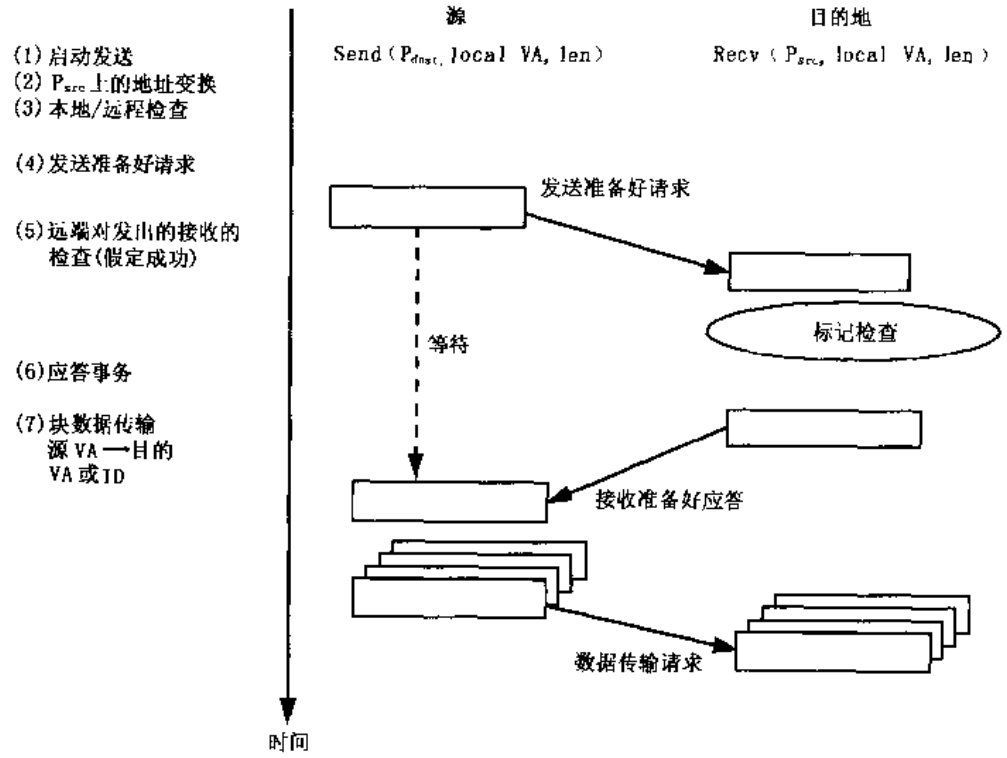


图 7-10 同步消息传递协议。该图说明了根据基本网络事务实现同步发送/接收对的三次握手

在很多的消息传递系统中，和同步消息相联系的匹配规则有相当大的限制性，接收显式地指定了发送进程。这就允许了另一种接收者启动的协议，这里，匹配表在发送者处维护，只需要两次网络事务（“接收准备好”和“数据传送”）。

如图 7-11 所建议的那样，带缓冲的发送通过一个“乐观”的单阶段协议简单地实现。发送操作在一次大的事务中传输源数据，使用一个含有匹配信息（比如源进程和标记）和长度信息的信封。目的端将信封剥离，检查它的内部表，决定一个匹配的接收是否已经发出。假如有匹配接收，它就可以向指定的接收地址发送数据。假如没有匹配接收，目的端将为整个消息分配存储空间并且把它们放入临时的缓冲区。当后来发出匹配的接收时，消息就被拷贝到指定的目的区域，清空缓冲区。 478

这个简单的协议代表了一类问题。首先，消息数据的合适的目的地址，直到检查了进程和标记信息并询问了匹配表之后，才能够决定。这些是代价很高的操作，一般由软件来执行。同时，消息数据通过网络以一个较高的速率流入。一种方法是先把数据接收到一个临时输入缓冲区，然后把它们拷贝到合适的目的地。当然，这就引入了存储-转发延迟并消耗了相当数量的处理器资源。

和这个乐观方法相关的第二个问题与针对共享地址空间抽象的输入缓冲区的问题非常相似，因为在数据传输前没有“准备好接收”的握手。事实上，问题在很多方面放大了。首先，传输量更大了，所以在目的端所需要的存储空间总量要求相当大。第二，缓冲存储空间

的大小依靠程序的行为，它不只是多个发送和接收者之间速率不匹配的结果，也不是数据恰好在接收者就绪之前到达的定时不匹配。多个进程可能选择向单一进程发送很多消息，该进程可能恰好要推迟很久才接收它们。从概念上来讲，异步消息传递模式假设在通常的程序数据结构外有无限大小的存储空间。消息数据一直存储到接收事件发出并执行。而且，必须允许阻塞的异步发送结束，以避免像成对交换这样简单的通信模式所引起的死锁。要允许程序通过这个发送而继续执行，直到接收发出。我们的乐观协议在消息传递层没有区分临时的接收端缓冲和拖长的数据累积。

479

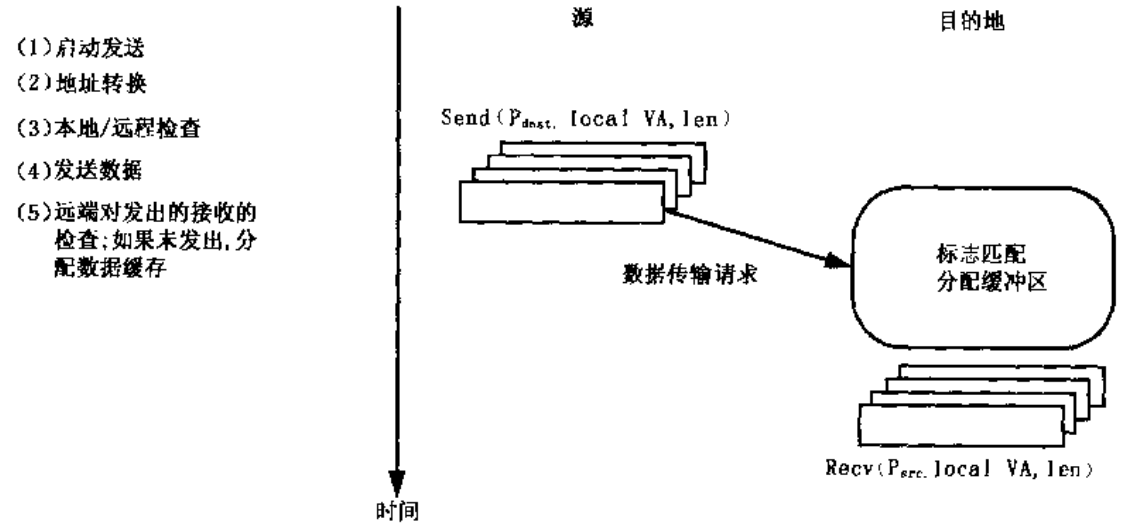


图 7-11 异步（乐观的）消息传递协议。该图说明了一个用于异步消息传递的简单的单阶段乐观协议，源只是简单地把数据传送给目的端，而不关心目的端是否有存储空间来容纳数据

一个更加健壮的消息传递系统使用一个针对大块传送的三阶段协议，如图 7-12 所示。发送端用信封发出一个“准备好发送”，但是把数据缓冲在发送端直到目的端能够接收到为止。目的端在有足够的缓冲空间，或者当一个配对的接收已经执行时发出一个“准备好接收”，以使得传输可以发生于正确的目标区域。注意在这种情况下，源和目的地址在实际的数据传输发生之前已经被传输双方所知。对于握手可能成为实际数据传输的主要代价的短消息来说，可以使用一个简单的信用方案。每一个进程为每个可能会向它发送短消息的进程设置一定数量的空间。当发送了一个短消息后，发送者局部地扣减它的目的信用，直到接到短消息已经被接收的确认。以这种方式，一个短消息通常无需等待握手的往返延迟就可以发出。以后使用结束确认来决定什么时候可以在无需握手的情况下发送其他的短消息。

480

至于共享地址空间，这里的设计问题与源和目的地址是物理地址还是虚拟地址有关。在每一端的虚-实映射可以作为发送和接收调用的一部分来执行，允许通信辅助部件交换可用作 DMA 传输的物理地址。当然，页必须在数据传送入存储器的时候保持常驻。然而，驻留的时间可以限制在将要开始握手到传输结束为止。另外，非常长的传输可以在源处进行分段，以使用很少的驻留页。而作为另一种选择，临时缓冲区可以驻留，处理器依靠这种驻留从源到目的区域拷贝数据。解决这个问题主要依靠通信辅助部件的能力，这将在下面进行讨论。

简而言之，发送/接收的消息传递抽象在逻辑上是一个单向的传输，源和目的地址由两个参加的进程单独指定，并且任意数量的数据可以在其中任何数据被接受之前发送。这类和

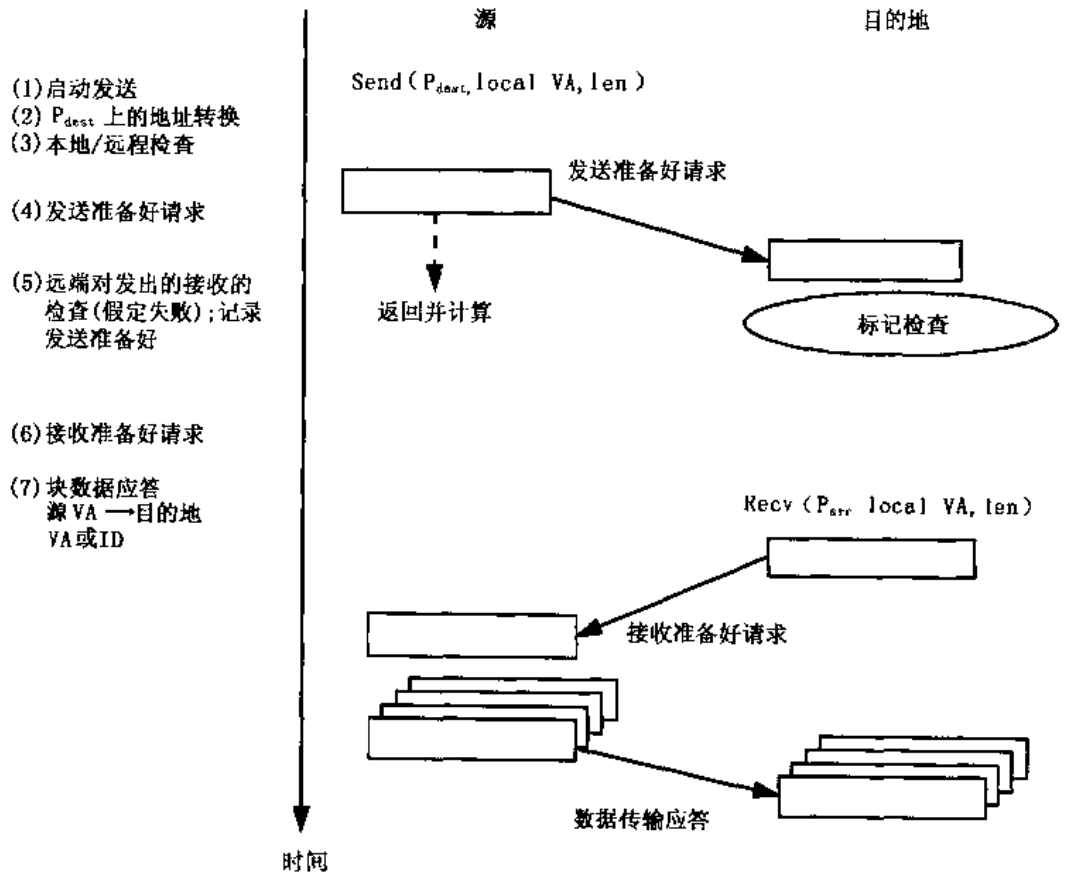


图7-12 保守的异步消息传递协议。该图说明了一个1加2阶段的保守异步消息传输协议。数据在源处保持，直到匹配的接收执行，使目的地址在数据被递交之前已知

基本网络事务相关的抽象，在实现时要使用三阶段的协议来管理两端的缓冲区。尽管通过某些形式的流控制可以安全地使用一个乐观的单阶段协议。

7.2.4 主动消息

尽管共享地址空间和消息传递成为现代并行机器的主要编程模型，提供一个和这些模型所依赖的网络事务相近的通信抽象也是可能的。最为广泛使用的低级通信抽象是主动消息机制（von Eicken et al. 1992）。主动消息以一种本质上是受限的远程过程调用的形式形成了请求和应答事务。每一个消息标识一个目的节点的处理例程，在消息到达时被调用来处理事务。一个典型的请求包括目标处理器地址，在那个处理器上的消息处理例程标识符，还有在源处理器寄存器中作为参数传递给处理例程的少量数据字。一个优化的指令序列通过一个通信辅助部件把消息发送到网络上。在目的端的处理器，一个优化的指令序列从网络上抽取消息并调用处理例程，针对消息数据执行一个简单的操作并发出响应，该响应标明了源处理器上的响应处理例程。通过构造实现适当协议的处理例程，可以在主动消息原语的基础上建立更高级的编程模式（Tucker and Mainwaring 1994; Shah et al. 1998）。

481

消息到达（即调用处理例程）通知，可以通过中断或激励一个线程的方式提供，但它必须作为发出一个主动消息的一部分，以便使这样一个低层的通信操作在没有任何缓冲的情况下达到无死锁。当力图发送一个请求的时候，网络可能会满，处理器应该允许调用针对进入

的消息的处理例程，以继续执行。所以，通过一个称作轮询的空消息事件，可以显式地调用处理例程进行网络服务。与中断和线程不同，这种方式允许处理例程和目的进程同步执行。

批量传输可以与主动消息方法相结合，这可以通过为事务安排相关的数据缓冲区（作为请求的一部分，提供了一个指向源数据缓冲区的指针，缓冲区被拷贝到了目的端，指向目的缓冲区的指针被提供给了处理例程），或者存储器到存储器的拷贝优先于处理例程的调用（Mainwaring and Culler 1996）。

7.2.5 共同的挑战

在大规模系统中实现编程模型的固有挑战在于每个处理器只了解系统状态的一部分，而非常多的网络事务可以同时进行，并且事务的源和目的之间并无耦合。每一个节点必须从自己的点到点事件中推断系统的相关状态。在这样的情况下，一组源节点在发现问题之前，可能已经使一个目的节点过载严重。而且，因为涉及的时延很大，我们试图使用乐观协议和大尺寸的传输。这些都会增加目的端的过载的可能性。而且，用来实现编程模型的协议对于一个操作往往需要多个网络事务。所有的这些问题都应该被考虑到，以保证在没有全局仲裁的情况下程序可以向前推进。这些问题和基于总线的设计中遇到的问题非常相似，但是不能够依靠对总线的约束，即少量的处理器、有限数目的未决事务以及总的全局的排序来解决。

1. 输入缓冲区溢出

考虑一下对远端节点输入缓冲区的竞争问题。为了保持讨论的简单性，暂时假设在完全可靠的网络上的固定格式的传输。输入缓冲区的管理是简单的：一个队列就足够了，每一个进入的事务放在下一个空闲的队列槽中。然而，很可能有大量处理器同时请求同一个模块。如果这个模块具有固定的输入缓冲区容量，在一个大的系统中，它可能会过载。这种情况类似于在网络中对有限缓冲区的竞争，并且可以以一种类似的方式来处理。一种解决方案是使输入缓冲区变大，并且为每个源预留一部分。当源端耗尽了自己在目的端的份额后，就必须限制自己的请求。这就带来一个问题，源怎么知道它是否还能得到空间？必须有一些流控信息从目的端传回到发送者。这是一个设计方面的问题，可以通过确认每一个事务或者在更高的层次将确认和协议耦合来解决（例如，回答指示处理请求的完成）。

482

另一种在可靠的网络中常用的方案是，让目的端在其输入缓冲区满的时候，简单地拒绝接受进入的事务。当然，数据无处可去，所以它会滞留在网络中一段时间。正向满的缓冲区发送的网络交换机将处于一种不能以包到达速度转交包的状态。由于它的缓冲有限，它最终会拒绝接受输入包。这种现象称作反向压力。假如目的端的过载持续了足够长的时间，阻塞的块将会在反向通往所有源的路径上形成一棵树。这些源感到从过载的目的端传来的反向压力，被迫放慢速度，以使得向目的端发送的源端的速率之和低于目的端接收的速率。

我们可能担心系统在这种情况下会崩溃。一般来说，网络以无死锁的方式来构建——即只要消息在目的端可以被移去，消息就应该可以前进（Dally and Seitz 1987）。所以会产生进展。问题在于，网络在阻塞的情况下，不是向过载的目的端方向前进的消息也会被阻塞。因此，在这种阻塞建立的时候，总的通信时延将会急剧增加。

一个可靠网络上的反向压力在处理节点和网络间建立了一个有趣的“契约”。从源的观点看，假如网络接收了一个事务，它就会保证事务最终会被传送到目的端。但是，事务可能不会在任意长的时间内接收，并且在这段时间内源必须继续接受进入的事务。

或者，可以以另一种方式构建网络，使得目的端可以通知源端它的输入缓冲区的状态。这通常是通过在反向预留一个特殊的确认路径来实现。当目的端接受一个事务的时候，它会对源端明确地发出确认；假如它丢弃了该事务，它就会发送一个否认，通知源以后重发。如以太网、FDDI和ATM这样的局域网，在没有足够的空间缓冲事务时，会采取更加原始的方法将其抛弃。源依靠超时来决定事务可能已被丢弃并且重发。

2. 取死锁

输入缓冲区问题在请求-应答协议的情况下，显示了额外的复杂性，这是共享地址空间所固有的并存在于消息传递的实现中。在一个可靠的网络中，当处理器试图启动一个请求事务时，由于目的节点竞争和/或网络竞争，网络可能会拒绝接受它。为了使得网络无死锁，就要求源即使在不能开始自己事务的情况下继续接受事务。然而，进入的事务可能是一个请求，它将会产生一个响应。这个响应在网络满载的情况下无法进行。 483

对这个取死锁问题的一个通常解决方法是，为请求和响应提供两个逻辑上独立的通信网络。这可以通过两个物理网络或在单一网络中具有单独的输出和输入缓冲的独立虚通道来实现。尽管在因试图发送请求而阻塞时必须继续接受响应，但响应可以在不启动进一步的事务的情况下完成。因此，响应事务最终将有所进展。这意味着，进入的请求最终将被服务，阻塞的请求最终将继续执行。

另一个解决方案是保证当一个事务被启动时，在目的端输入缓冲空间总是可以使用，这是通过限制未决事务的数量实现的。在一个请求响应协议中，限制任何处理器未决请求的数量的方法是简单的；维护一个计数器，每个响应将计数器减一，允许发出一个新的请求。标准的阻塞读和写通过在结束当前请求之前等待响应来实现。然而，如果有 P 个处理器，每个处理器最多有 k 个未决请求，很有可能所有这 kP 个请求指向同一个模块。对于发往同一个目的地的 $k(P-1)$ 个未决请求和目的节点发出的对请求的响应，空间应该可用。显然，可以使用的输入缓冲区最终将限制系统的可扩展性。因为网络总是把事务吸纳在目的地可用的输入缓存中，所以能保证请求事务的进展。当节点在它的未决事务信用已经耗尽时试图产生请求就导致了取死锁的问题。为了接收它自己的响应，它必须对进入的事务服务，使进一步的事务得以产生。进入的请求能被服务，因为请求者保证为响应预留了输入缓存空间。所以，即使节点在力图提交响应时，仅仅将进入的事务入队并忽略，也能够保证向前的进展。

最后，我们可以采用在分裂事务总线所使用的方法，当输入缓冲区满的时候，否认（NACK）事务。当然，否认可以任意推迟。这里假设网络可靠的传送事务和否认，但是目的节点为了腾出输入缓冲空间可能选择抛弃它们。响应从来不需要被否认，因为它们被其目的节点吸纳，该节点是与该响应对应的请求的源并为响应留有输入缓冲空间。当试图启动一个请求时阻塞，我们需要接受和吸纳响应并否认请求。可以假设在NACK的目的端输入缓冲区是可用的，因为它只是简单地使用为预期的响应预留的空间。只要每个节点为请求提供了一定输入缓冲空间，我们就可以保证最终一些请求将成功，并且系统不会活锁。需要进一步的预防措施来减少饿死的概率。 484

7.2.6 通信体系结构设计空间

在本章的其余部分，将考察大规模分布存储器型机器的重要设计问题。回忆一下，如图7-13所建议的，我们通用的大规模机器体系结构包含相当标准的节点结构，由硬件的通信辅

助部件所增强。关键的设计问题是网络事务中信息被通信辅助部件直接解释，无需节点处理器的参与。为了解释进入的信息，必须指定其格式，就像在构造一个解释器（即处理器）之前要定义一个指令集的格式一样。事务的格式化必须与地址转换、目的路由选择和介质仲裁一起，由源通信辅助部件部分地执行。所以，在产生网络事务时源通信辅助部件执行的处理和目的地执行的处理一起，实现了提交给节点的低层硬件通信原语的语义。实现所希望的编程模型的任何其他处理，由节点处理器在用户级或系统级进行。

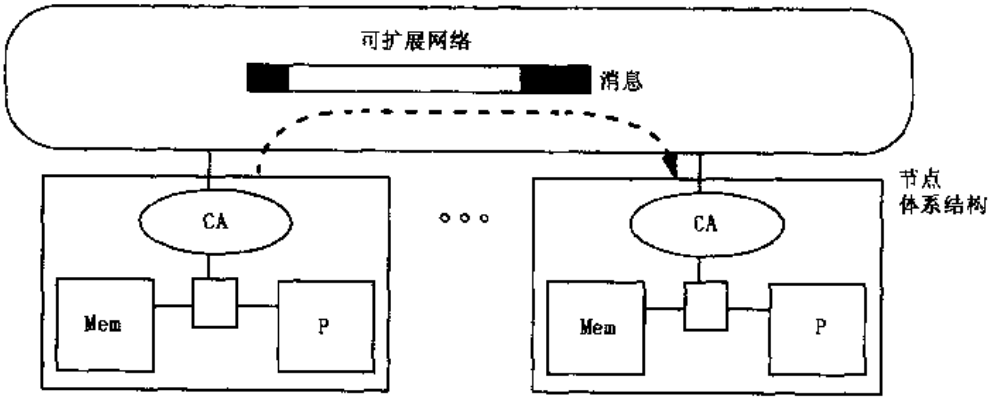


图 7-13 通用的大规模体系结构中的网络事务处理。一个网络事务是一个单向的从源端输出缓冲区到目的端输入缓冲区的数据传输，引起在目的端发生某种动作，这种动作的发生对源端不是直接可见的。源通信辅助部件（CA）对事务做格式化并且通过网络将其送出。目的通信辅助部件必须解释该事务并且引发适当的动作。这种解释的本质是可扩展多处理器设计方面的关键问题

485

确立两个通信辅助部件执行的网络事务处理本质的地位，对设计的其余部分有着深远的含义，包括如何实施输入缓冲，如何加强保护，数据在节点内拷贝几次，地址如何翻译等等。对进入的事务的最小解释就是对它完全不做解释。它被看作原始的物理二进制串，只是简单地被放置在存储器或寄存器中。更多的特殊解释提供了用户级的消息、一个全局的虚拟地址空间或者甚至一个全局的物理地址空间。在后续几节，将依次考察每一个部分。我们将考察一些使用了相关设计的重要机器进行案例分析。

7.3 物理 DMA

本节考虑不对网络事务中的信息进行解释的设计。这种方法在大多数早期的基于消息传递的机器上很典型，包括 nCUBE10 和 nCUBE/2、Intel 的 iPSC、iPSC/2 和 iPSC860、Delta、Ametek 以及 IBM SP-1。另外，大多数的局域网的接口也遵循这一设计。硬件可以十分简单，并且用户通信抽象非常通用，只是典型的额外开销太大。

7.3.1 节点到网络的接口

硬件本质上支持物理的直接存储器存取（DMA），如图 7-14 所示。一个 DMA 设备或 DMA 通道带有它的地址和长度寄存器、状态（例如，传输就绪、接收就绪）以及中断使能。该设备或者是存储器映射的，或者提供特权指令来访问这些寄存器。地址是物理地址[⊖]，所

⊖ 在 Sun 工作站和服务器中使用的 SBUS 是一个例外。它提供虚 DMA，允许 I/O 设备对虚地址而不是实地址操作。

以在网络事务中传输的是连续的存储器区段。发送通常会导致陷入操作系统。特权软件会提供源地址翻译，把逻辑节点转换成物理路由，对物理介质进行仲裁，访问物理设备。通常，数据会被拷贝到内核区，这样，可以建立包括路由信息和其他信息的封装。封装的其他部分，例如，错误检测位可以由通信辅助部件产生。操作系统内核选择适当的通道，把通道地址设置为消息物理地址并且设置计数。(另一种方案是内核建立一个包括这些信息的描述符，并把它发到临时队列中去。) DMA引擎把消息放到网络上。当传输结束时，输出通道就绪标记置位，产生一个中断，如果中断没有被屏蔽的话。消息通过网络到达目的地，目的节点的输入通道的DMA必须被启动，允许消息流过网络进入节点。(如果在启动输入通道时发生延迟，或者如果消息在网络中与其他使用同一链路的消息碰撞，消息一般会停留在网络中) 通常，输入通道的地址寄存器中预先设置了数据将要存入区域的基地址。DMA会把网络上到达的数据字送入存储器。消息尾部会设置输入就绪位并产生一个中断，除非中断被屏蔽。为了避免死锁，必须激活输入信道来接收消息并排空网络，即使在忙碌的输出通道上有输出消息需要发送。

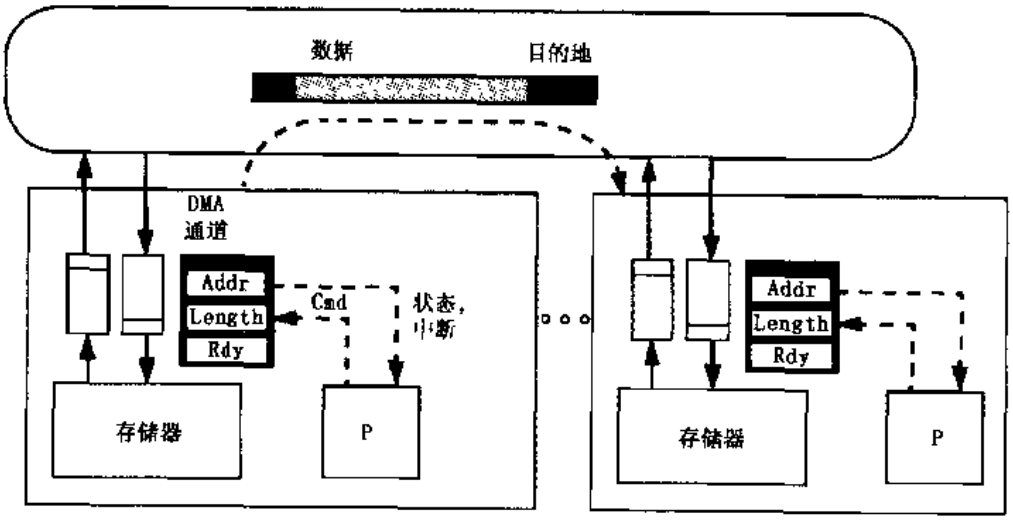


图7-14 通信辅助部件中对单纯物理的DMA的硬件支持。单纯的物理DMA是最小化的解释。它允许目标通信辅助部件仅仅把事务数据存入存储器，在那里由处理器解释。因为事务的类型没有事先决定，所以使用系统内核缓冲区存储，事务的处理通常会涉及现场切换和一次或多次拷贝过程

这个方法的关键特征在于目的处理器启动一次从网络到存储器的DMA传输，下一个进入的网络事务只是盲目地放到指定的存储器区域。当系统在目的地端设置输入DMA时，它无法知道下一个消息是用户消息还是系统消息。它只是盲目地传入预先指定的物理区域。消息到达通常会引起一个中断，这样，特权软件可以检查消息，处理它或者把它交给适当的用户进程。节点处理器上的系统软件解释网络事务并且给用户提供一个清晰的抽象。

一个降低通信额外开销的可能的方法是允许对DMA设备的用户级访问。如果DMA设备是存储器映射的，正如大多数的情况，问题在于如何设置用户的虚拟地址空间以覆盖包含了设备控制寄存器的I/O空间。但是，采用这种方法，保护域和资源共享的级别相当粗糙。当前用户得到整个机器。如果用户错误地使用网络，那么操作系统除了重启计算机之外，无法进行干预。这个设计已经用于一些试验性的场合，但不是很健壮，这使得并行机变得像昂贵的个人计算机。

486 ~ 487

7.3.2 通信抽象的实现

因为在这些机器中的硬件通信辅助部件是相对基本的，关键问题是如何将新接收的网络事务以一种健壮的、被保护的方式递交给用户进程。这就是编程模型和通信原语的连接点。最常用的方法就是直接在内核支持消息传递的抽象。网络事务到达时触发一个中断。网络事务中进程标识符和标记被解析，在图 7-10 或图 7-12 的过程中发生一个协议动作。例如，如果已经发出了匹配的接收，那么数据就会被直接拷贝到用户存储器空间。如果未发出，那么内核就会提供缓冲或在目的用户空间分配存储空间来缓冲数据，直到匹配的接收被执行为止。另一种方法是，用户进程可以预先分配缓冲并且通知内核它要在何处接收消息。一些消息传递层允许接收者在缓冲区外直接操作，而不是把数据接收到自己的地址空间。

让内核软件提供一种全局虚拟地址空间的用户级抽象也是可能的。在这种情况下，读或写请求或者直接通过系统调用发出，或者通过对远程逻辑页的存取的陷阱而发出。源节点的内核发送请求并处理响应。目标节点的内核从网络事务中提取用户进程、命令和目标虚拟地址并且执行读写操作（如图 7-8 所示），发出响应。当然，与这样的共享地址空间抽象的实现相关的额外开销是相当大的，特别是对一次一字的操作而言。通过成块数据的传输可以获得更高的效率，它使得这种共享地址空间的方法和消息传递一样有竞争力。已经建立了许多软件共享虚拟存储器的系统，多数在工作站机群上，但是主要的努力在于自动复制、减少通信量。这些将在第 9 章详细论述。

在内核和用户间的其他连接是可能的。例如，内核可以提供一个用户级输入队列的抽象，简单地把消息附加到适当的队列中，在队列溢出时遵循某些良好定义的策略（Brewer 等 1995）。

7.3.3 案例分析：nCUBE/2

nCUBE/2 是关于物理 DMA 型机器的典型代表。网络接口的组成结构如图 7-15 所示，每一个 DMA 输出通道驱动一个输出端口，并且每一个输入通道和一个输入端口相连。这个机器是直接网络的一个例子，在直接网络中，数据通过中间节点从源传送到目的地。交换机把网络事务从输入端口转发到输出端口。网络接口检查到达各个输入端口的消息封装、决定消息是否以本节点为目的地。如果是，就激活输入 DMA 将消息取到存储器中。否则，将消息转发到适当的输出端口[○]。逐链路的流控保证这种传递是可靠的。用户程序被分配到连续的子立方体中，路由方式使得子立方体的链路仅仅由立方体内的流量使用；因此对于空间共享的机器，用户程序不能相互干扰。nCUBE/2 的一个特殊之处在于输入通道中没有计数寄存器，所以源节点和目的节点的内核必须保证进入的消息的长度不能超过存储器输入缓冲区。

488

为了帮助解释网络事务和不经拷贝将用户数据传送到希望的存储区域，在 nCUBE/2 中，有可能在一次连续的传输中传输一系列消息段。在目标节点，输入 DMA 会在每一个逻辑段处停下来。因此，目标节点可以中断，检测第一个段，决定将其余段引向何方；例如，通过查找一个接收表。但是，这种能力代价很高，因为对每一个逻辑段都需要一个启动 DMA 和中断（或忙等待）。

○ 这个路由步骤是 n 立方体的互连网络拓扑与节点设计关联的基本所在。正如我们在第 10 章将要讨论的，输出端口是由本地节点地址和消息目的地址的第一个不同位的位置决定的。

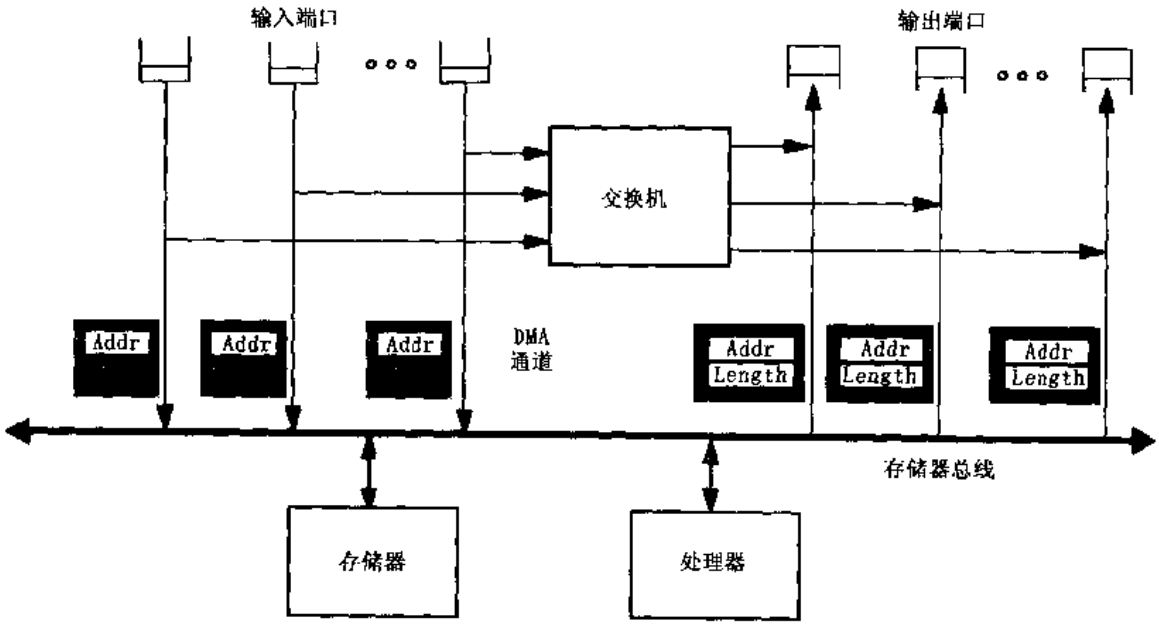


图 7-15　nCUBE/2 的网络接口组成结构。多个 DMA 通道直接驱动从存储器到网络或从网络到存储器的网络事务。输入通道把数据存放进存储器，地址由处理器指定，与内容无关。为了避免拷贝，机器允许多个消息段合成一个单独的单元通过网络传输。一个典型的方法是处理器向通信辅助部件提供一个输入 DMA 描述符的队列，每个描述符包含存储器缓冲区的地址和长度。当网络事务到达时，从队列弹出一个描述符，数据被放入相关的存储缓冲区

489

最好的策略是，在内核一级，当输出缓冲区正被注入输出端口时为每一个输入通道保留一个输入缓冲区（von Eiken et al. 1992）。典型地，每个消息将包含一个消息头，允许内核根据消息类型分发消息，并采取适当的动作来处理这一消息；例如执行标记匹配或把消息拷贝到用户数据空间。

在这个平台上最有成效并具有最细致的文档的通信抽象是主动消息（von Eiken et al. 1992）。用户消息的第一个字包含了处理该消息的用户例程的地址。消息的到达触发一个中断，在栈上有被中断的用户地址，内核执行一个中断返回，切换到消息处理例程。采用这种方法，可以在 13 μs 中将一个消息传到网络上（16 条指令，包括 18 次存储器访问，花费 260 个周期），用 15 μs 从网络抽取消息（18 条指令，包括 26 次存储器访问，花费 300 个周期）。将此与厂商消息传递库所需要的 150 μs 的启动时间相比，反映出消息传递编程模型中用户级操作和硬件基本操作之间的差距。厂商消息传递层使用乐观的单向协议，但是需要匹配和缓冲区管理。

7.3.4　典型的局域网接口

nCUBE/2 中的简单 DMA 控制器对于并行机是典型的，但质量上和那些外部设备和局域网的 DMA 控制器是不同的。注意，每一个 DMA 通道都能胜任一个单一连续的传输。几条简短的指令就完成了下一次输入或输出操作的通道地址和通道限制的设置。传统的 DMA 控制器提供了把大量的传输链接起来的能力。为了启动一次输出 DMA，把 DMA 描述符链接到输出 DMA 队列中去。外设控制器轮询这个队列，发出 DMA 操作并在操作完成时通知处理器。

大多数的 LAN 控制器，包括 Ethernet LANCE、Sun ATM 适配器以及其他适配器，提供一

个发送描述符队列和一个接收描述符队列。（这里还有每种描述符的一个自由链表。典型做法是，队列和它的自由列表组合成一个单一环。）内核在存储器中建立了输出消息，并且建立了一个包括输出信息的地址、长度以及某些控制信息的发送描述符。在某些控制器中，一个单一消息可以用一系列描述符来描述，所以控制器能从不同的存储器区域得到包的封装和数据。典型地，控制器有一个通往网络的端口，所以它能把消息推到网线上。对于以太网和令牌环网，每个控制器在消息经过的时候检查，所以事务说明的是目标地址而不是路由。

490

输入端更加有趣。每个接收描述符有一个目标缓冲区地址。当一个消息到达时，从队列中弹出一个缓冲区描述符，启动一次 DMA 传送，把消息数据装入存储器的相关区域。如果没有可用的描述符，就丢掉消息，更高层的协议必须重传（就好像消息在传输中被破坏了）。大多数设备有可配置的中断逻辑，所以每次消息到达的时候产生一个中断。操作系统的驱动程序管理这些输入和输出队列。使用这样的设备，即使是一次较小的传输，其设置所需的指令的数量也是相当大的，部分是由于要形成描述符以及控制器之间的握手。

7.4 用户级访问

对进入的网络事务的最基本的硬件解释区分用户消息和系统消息，并且无需操作系统介入，把用户的消息递交给用户程序。每个网络事务携带一个用户/系统的标记位，在消息到达的时候由通信辅助部件检查。此外，应该能在用户级将用户消息插入到网络中去；通信辅助部件在生成事务时自动插入用户标记。事实上，这种设计观点提供了一种用户级的网络端口，即一种无需系统介入就能够被写入和读出的网络访问路径。

7.4.1 节点到网络的接口

图7-16显示了一种典型的支持用户级网络访问的并行机组成结构。地址空间的一个区域被映射到网络的输入和输出端口及状态寄存器中，如图 7-17 所示。处理器可以通过向输出端口写入目的节点号和数据产生一个网络事务。通信辅助部件执行保护检测，把逻辑目的节

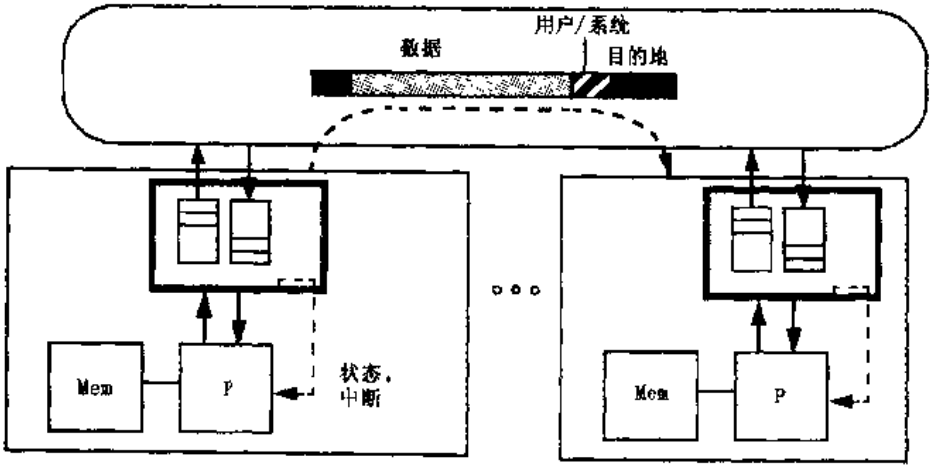


图 7-16 在通信辅助部件中对用户级网络端口的硬件支持。网络事务被区分为系统和用户两类。通信辅助部件向系统和用户提供先进先出的（FIFO）网络输入输出队列。通信辅助部件在发送用户消息时，给它打上标记；在接收到消息时，检查事务的类型。用户消息会停留在用户输入 FIFO 中，直到用户应用将它取出。系统事务引起一个中断，由系统以特权方式对其进行处理。由于缺少用户级中断的支持，产生中断的用户事务被当作特殊的系统事务处理

点号转换成物理地址或路由，并且进行介质仲裁。它还插入消息类型和任何差错校验信息。当消息到达的时候，一个系统消息会触发一个中断，系统会把它从网络中抽取出来，而一个用户消息会留在输入队列中，直到用户进程从网络读取它，将它从队列中弹出。如果网络阻塞，向网络写消息的企图就会失败，用户进程必须能继续从网络抽取信息，以便能向前推进。由于当前的微处理器不支持用户级中断，一个产生中断的用户消息会被通信辅助部件当作系统消息处理，系统迅速地将控制转交给用户级的处理例程。

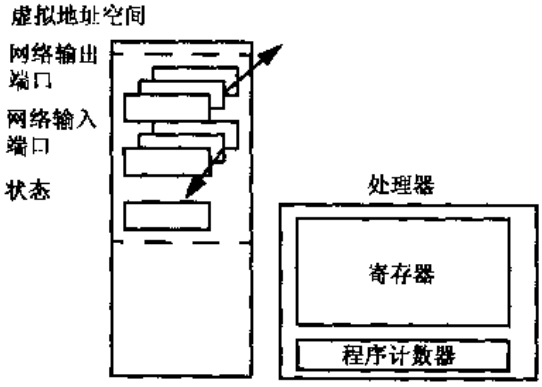


图 7-17 用户级网络端口的典型体系结构。除了指令集体系结构和存储器空间所提供的存储，在用户虚拟地址空间的一个区域提供了网络输出端口、输入端口和状态寄存器的访问。通过写入和读出这些端口，加上检测状态寄存器，来启动和接收网络事务

这种设计点意味着通信原语允许在网络中的进程状态部分，即已经离开了源，但还没有到达目的地。所以，如果构成一个并行程序的用户进程集合是根据时间片调度的，那么程序切换时中途的消息集合也需要保护。在程序恢复时，它们将被重新插入到网络或目的输入队列中。

491 ~ 492

7.4.2 案例分析：Thinking Machines CM-5

真正支持用户级网络的最早的商用计算机是 1991 年由 Thinking Machines Corporation 发明的 CM-5。通信辅助部件包含在一块网络接口（NI）芯片中，该芯片连接在存储器总线上，就好像它是一个附加的存储器控制器，如图 7-5 所示。NI 对两个数据网络和一个“控制网络”都提供了输入和输出 FIFO，控制网络专门完成像栅障、广播、规约、扫描这样的全局操作。通过把网络输入输出端口和一些状态寄存器映射到地址空间，如图 7-17 所示，通信辅助部件可以为处理器所用。内核可以访问所有的 FIFO 和寄存器，而用户进程只能访问用户 FIFO 和状态。在这两种情况下，都是通过使用传统的装入和存储指令，对通信辅助部件的寄存器读写来启动和完成通信操作的。此外，通信辅助部件可以产生中断。在 CM-5 中，每次网络事务包括一个小的标记，通信辅助部件维护一个表，用来指出哪些标记应该引起中断。（所有的系统标记引起中断。）

在 CM-5 中，可以在 1.5 μs（50 个周期）内将五个字的消息写入网络，用 1.6 μs 将它读出。此外，穿过无载网络的时延从邻近节点间的 3 μs 到跨越 1 024 个节点的 5 μs 不等。用户级向量中断耗费 10 μs。如果几个消息快速相继到达，用户级的处理例程可能会处理几个消息。将消息传入或传出网络接口的时间由在存储器总线上所花费的时间决定的，因为这些操作都是作为不经高速缓存的读写执行的。如果消息的数据从寄存器出发，它可以作为一系列的缓冲存储写入网络接口。但是，它后面必须紧跟着一个读操作来检查消息是否被接受，在这点上，要经历写时延，因为要再次将写缓冲发给 NI。

如果消息数据从存储器发出，并且要被存到存储器而不是寄存器中，评价是否使用DMA将数据传入或传出NI是有趣的。关键的资源是存储器总线。当使用传统的存储器操作访问用户级的网络端口时，消息的每个字先被读到寄存器中，然后被写到NI或存储器中。如果数据不能被高速缓存，那么网络事务中每个数据字涉及4次总线操作。如果存储器数据可以被高速缓存，那么处理器和存储器的传输作为高速缓存块传输执行，当数据在高速缓存中时，这种传输可以避免。但是，NI读和写依然存在。对于DMA传输，使用阵发传输模式，数据在网络事务的两个端点只经过存储器总线一次，然而，仍然必须把DMA描述符写入NI。如果消息数据不处于存储器能被高速缓存的区域，采用DMA的性能优势就会丧失殆尽，因此，NI所执行的DMA传输必须要和处理器的高速缓存一致。所以，节点存储器体系结构支持一致的高速缓存十分关键。在接收端，NI必须根据网络事务中的信息和NI内部状态启动DMA，否则，我们又将面临单纯物理DMA的问题。这将导致我们对网络事务做额外的解释，以便通信辅助部件抽取地址字段。我们在7.5节将进一步考虑这种方法。

493

CM-5中的两个数据网络提供了取死锁问题的一个简单解决方案：一个网络用于请求，一个网络用于响应（Leiserson et al. 1996）。当请求阻塞时，节点继续接收到来的应答和请求，这可能会产生向外的应答。当发送应答时阻塞，只从网络接受到来的应答。最终应答会成功，允许请求继续。另一种方法是，可以在每个节点提供缓冲区，以某种端到端的流控来保证缓冲区不会溢出。当用户程序试图从输入队列弹出消息时被中断，系统会提取消息的其余部分并在恢复程序执行之前将消息放回到输入队首。

7.4.3 用户级的处理程序

几种实验性的体系结构研究了用户级网络端口和处理器更紧密集成方案，包括Manchester Dataflow Machine（Gurd, Kerkham, and Watson 1985）、Sigma-1（Shimada, Hiraki, and Nishida 1984）、iWARP（Borkar et al. 1990）、Monsoon（Papadopoulos and Culler 1990）、EM-4（Sakai, Kodama, and Yamaguchi 1991）和J-machine（Dally, Keen, and Noakes 1993）。关键的差别在于网络中的输入和输出端口是处理器的寄存器，如图7-18所示，而不是特殊的存储器区域。这大大改变了节点的工程实现，因为这样，通信辅助部件本质上成了处理器的一个功能部件。各种操作的时延大大降低，因为是使用寄存器到寄存器指令把数据移入和移出网络的。对存储

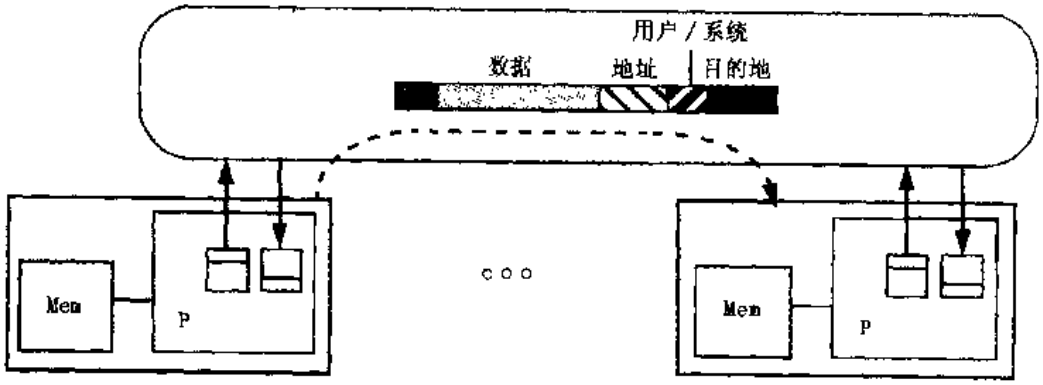


图7-18 通信辅助部件中对用户级处理例程的硬件支持。用户级处理例程所要求的基本支持是通信辅助部件能判定网络事务发往用户进程并直接将它提交给那个进程。这或者意味着每个用户进程拥有一组逻辑FIFO，或者一组FIFO被所有用户进程分时共享

器总线的带宽需求也降低了，并且通信支持部件的设计与存储器系统的设计分离了。但是，处理器要介入每一次网络事务。大规模的数据传输消耗处理器周期，而且有可能污染处理器的高速缓存。

有趣的是，这些实验性的机器以完全不同的途径到达了一个类似的设计点。CMU 和 Intel 联合开发的 iWARP 机（Borkar et al. 1990）把主寄存器文件中的两个寄存器绑定为网络输入和输出端口的头。在消息从网络流入时，处理器可能会逐字访问消息。此外，消息也可以被 DMA 控制器伪脱机送到存储器中。处理器通过说明一个消息标记来指明它希望通过此端口寄存器访问哪个消息，很像一个传统的接收调用中所做的那样。其他消息通过 DMA 控制器写入存储器，DMA 控制器用一个输入缓冲队列来指明目的地址。引导一个进入和离开的消息通过寄存器文件的额外的硬件机制是受到脉动算法的启发，在脉动算法情况中，数据流被泵过高度规则的处理器流水线，在数据流经过时，做少量的计算。通过在网络接口中解释标记，或者以 iWARP 的术语虚拟通道，基于存储器的消息流没有受到基于寄存器的消息流的打扰。与此对照，在 CM-5 风格的设计中，所有的消息在单一的输入缓冲中交错。

MIT 和 Motorola 提出的 *T 机（Nikhil, Papadopoulos, and Arvind 1993）为用户级消息处理例程提供了一个更通用的体系结构。它扩展了 Motorala 的 88110 RISC 微处理器，使它包含了一个包括一组寄存器的网络功能部件，很像浮点部件。在这个设计中，一个多字的输出消息在一组输出寄存器中组成，通过一条特殊指令，网络功能部件把它发送出去。几个输出寄存器组构成一个队列，发送使队列指针向前移动，对用户呈现下一个可用的组。功能部件的状态位指出是否有一个输出寄存器组可用，这些状态位可以直接在转移指令中使用。也有几个输入消息寄存器组，所以当一个消息到达时，会被装入一个输入寄存器组中，状态位被置位或者产生一个中断。为了允许处理器快速转移到消息的第一个字所说明的地址，提供了额外的硬件支持。

*T 设计大量借鉴了以前的支持消息驱动执行和数据流体系结构的计划，特别是 J-machine（Dally, Keen, and Noakes 1993）、Monsoon（Papadopoulos and Culler 1990）和 EM-4（Sakai, Kodama and Yamaguchi 1991）。这些早期的设计使用了相当不寻常的处理器体系结构，所以通信辅助部件没有很明白地说明。J-machine 的设计提供了两个执行现场，每一个现场都有一个程序计数器和一个小的寄存器集。“系统”执行现场的优先权高于“用户”现场。指令集包括分段的存储器模型，一个段是一个特殊的在片上的消息输入队列。每个现场也有一个消息输出端口。网络事务的第一个字被指定为消息处理例程的地址。当用户现场空闲并且一个消息在输入队列中时，消息头自动地装入程序计数器并为访问消息的其他部分设置一个地址寄存器。处理例程在挂起或结束之前必须从输入缓存抽取消息。一个系统级的消息的到来会抢占用户现场，并启动一个系统处理例程。

494
495

在 Monsoon 的设计中，网络事务具有固定的格式，并且规定包括处理例程的地址、数据帧的地址和一个 64 位的值。处理器支持这种小消息的大队列。基本的指令调度和消息处理机制深度集合在一起。在每个取指周期，一个消息弹出队列，消息的第一个字指明的指令被执行。指令具有 $1+x$ 的地址格式并且指明一个相对于帧地址的偏移量，那里有第 2 个操作数。每个帧的位置包含一个存在位，指明这一位置是满还是空。如果为空，消息中的数据字就会被写入指定的位置（类似存储累加器指令）。如果位置非空，它的值会被取出，对两个操作数执行操作，产生携带运算结果的一个或多个消息，发到一个本地队列或者跨越网络的

队列。在较早的更传统的数据流机器中，网络事务携带一个指令地址和一个标记，标记用于在一个相关匹配中定位第 2 个操作数，而不是简单的相对于帧的寻址。较晚一些的机器（Nikhil and Arvind 1989；Grafe and Hoch 1990；Culler et al. 1991；Sakai、Kodama and Yamaguchi 1991）对每一个消息出队和匹配操作都执行一个指令序列。

7.5 专用消息处理

基于分布存储器的大规模计算机的第三种重要的设计风格寻求使用专用硬件资源对网络事务进行复杂的处理，但不在硬件设计中加入解释功能。解释是通过在通信处理器（CP）上的软件执行的，专用通信处理器直接对网络接口操作。有了这种能力，很自然就想到把与消息传递抽象有关协议处理转移到 CP 中。CP 能执行缓冲、匹配、拷贝和确认操作。支持一个全局的地址空间，CP 代表请求节点执行远程读操作也成为合理的。CP 之间可以互相协作，提供把全局地址空间中的数据从一个区域移到另一个区域这样的通用的能力。CP 能够提供同步操作或数据移动和同步的组合，比如写数据和标记设置或数据入队列。本节考察这类机器的基本的组织特性，以便理解关键的设计问题。作为案例分析，我们会详细考察两台机器——Intel Paragon 和 Meiko CS-2。

496

这种设计的通用组成结构如图 7-19 所示，这里计算处理器（P）和通信处理器（CP）是对称的，两者都在存储器总线上。即以基于总线的 SMP 作为节点的基本部分（如第 5 章所概述的），扩充一个与前两节的叙述类似的基本网络接口。SMP 节点的一个处理器用软件专门设计，作为专用 CP。另一种组成结构是把通信处理器嵌入在网络接口中，如图 7-20 所示。这两种结构有不同的时延、带宽和成本权衡。后面将会说明。从概念上说，它们是非常类似的。通信处理器通常运行于系统优先级，使机器设计者不再受到前面讨论过的与用户级网络接口相关问题的困扰。这两个处理器通过共享存储器通信，通常采用命令队列和应答区的形式，所以特权级的变换本质上是区域交接的一部分，没有什么代价。因为设计假定系统级的

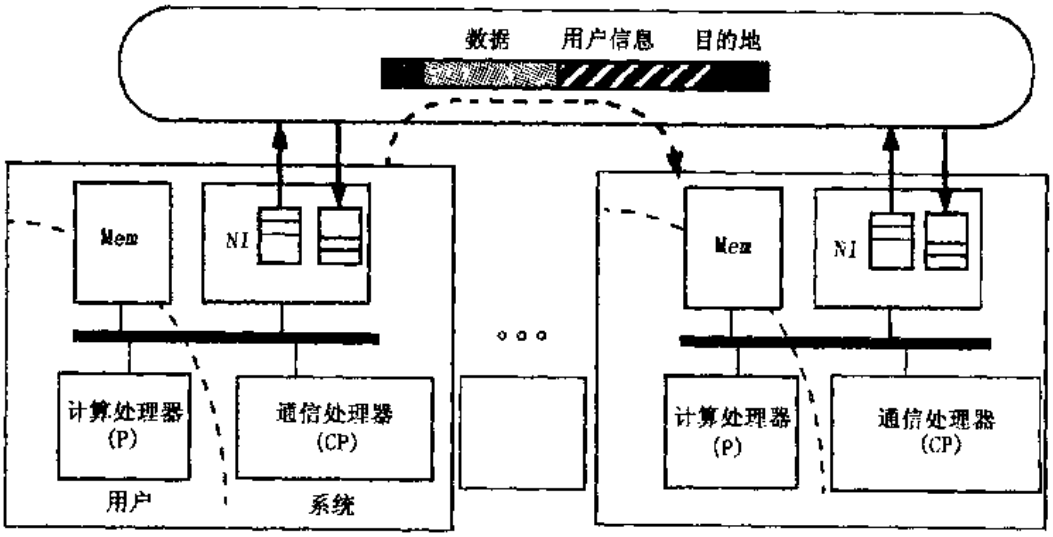


图 7-19 具有对称处理器的专用消息处理的机器组成结构。每个节点在共享存储器总线上有一个与主处理器对称的处理器，专用于启动和处理网络事务。因为是专用的，所以它总是运行于系统模式，所以通过存储器传送数据隐含着跨越保护边界。CP 能对事务的内容提供任何额外的保护性检查

497

处理器负责管理网络事务，所以它允许逐字地访问网络接口的 FIFO 和 DMA。通信处理器能够检查消息各部分，决定采取何种动作。CP 可以轮询网络和命令队列，来执行通信进程。

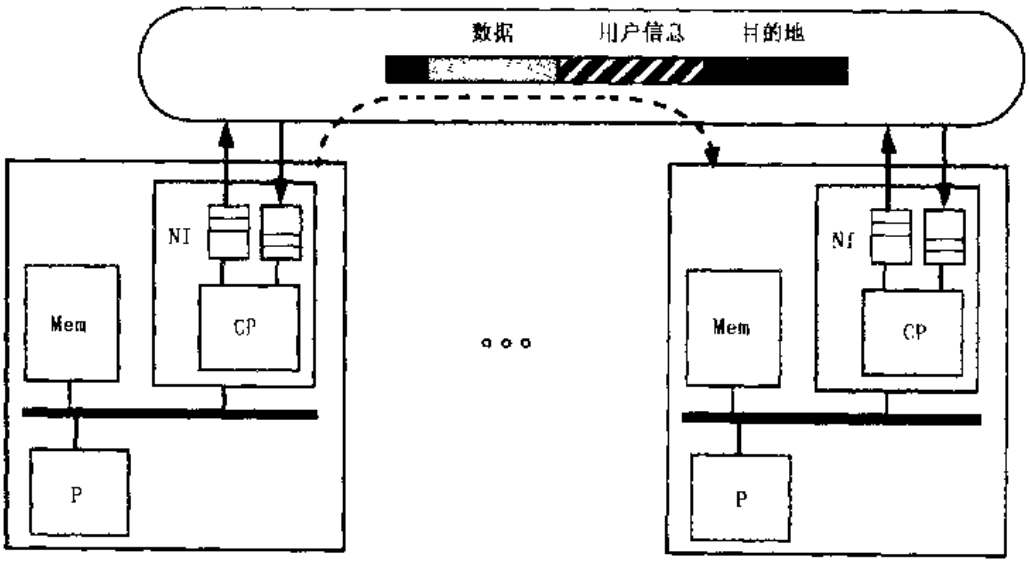


图 7-20 具有嵌入式处理器的专用消息处理的机器组成结构。通信辅助部件中包含一个专用的、可编程的、嵌入在网络接口的 CP。可以和网络之间有直接路径，不需要和主处理器共享存储器总线

CP 给计算处理器提供网络接口的一个非常清晰的抽象。物理网络操作的所有细节（例如，硬件的输入/输出缓冲、状态寄存器和路由表示）都被隐藏起来。一个消息的发送是简单地把消息或指向消息的指针写入共享存储器。这两个处理器间交换控制信息是使用我们熟悉的共享存储器同步原语，例如标记和锁。到来的消息直接由 CP 提交到存储器，同时通过共享变量通知计算处理器。通过精心设计的用户级抽象，数据可以直接放入用户地址空间。一个简单的低层抽象给并行程序的每个用户进程提供一个逻辑的输入和输出队列。在这种情况下，网络中的消息流动如图 7-21 所示。

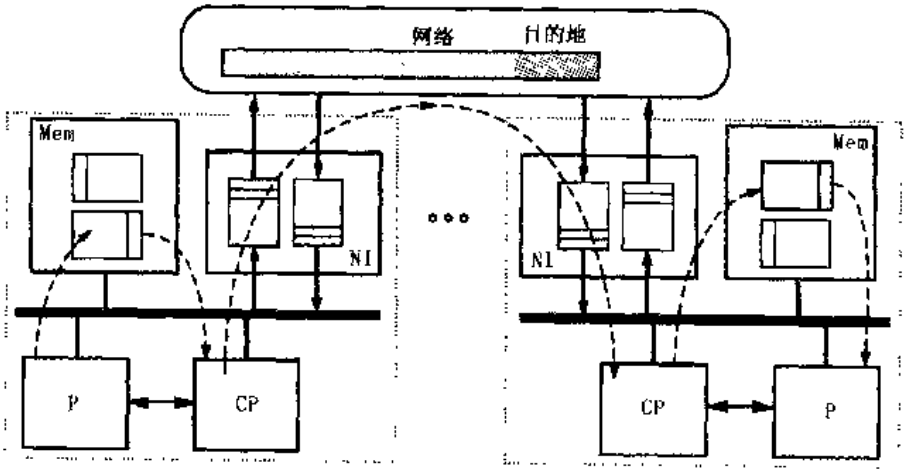


图 7-21 具有对称通信处理器时的网络事务流。在主处理器和通信处理器[⊖]之间的缓存到缓存传输中，每个网络事务流过存储器，或至少通过存储器总线。它也跨越 CP 和网络接口之间的存储器总线

⊖ 原文是存储器处理器。——译者注

这种好处并不是没有代价的。因为在节点中，计算处理器和 CP 间的通信通过共享存储器，所以通信的性能受到高速缓存一致性协议效率的严重影响。在第 5 章中，我们知道这些协议主要是为了避免两个工作于共享数据结构不相交区域的处理器间不必要的通信而设计的。而共享的通信队列是非常不同的情形。生产者写入数据，设置标记；消费者一定要看到标记更新，读取数据并清除标记。最终，生产者看到标记被清除，再写入数据。所有的数据应该以最小的时延从生产者移至消费者。我们将会看到传统一致性效率不高，使这一时延变得显著。例如，在生产者写入一个新数据项之前，在消费者高速缓存中的旧数据的副本必须作废。能想像一个更新协议或甚至没做高速缓存的写，能够避开这种情况；但如果这样，每个字，而不是每个高速缓存块，都会产生一次总线事务。

第二个问题是 CP 执行的功能是高度并发的。它要同时面对来自计算处理器的请求、网络上到达的消息和发往网络的消息。把所有这些事件折叠成一个串行分支循环，每次只能处理一个事件。这会严重削弱硬件的处理速度。

最后，CP 直接把消息传送到存储器的能力并没有完全消除用户级取死锁的可能性，尽管它能保证当应用死锁时，物理资源不会停止工作。用户应用可能需要提供某种额外流控。

7.5.1 案例分析：Intel Paragon

为了这些一般性的论点具体化，看一下它们如何体现在这类机器的一个重要代表——Intel Paragon 中的，Intel Paragon 于 1992 年首次推出。每个节点都是一个共享存储器的多处理器，包含两个或更多个 50 MHz i860XP 处理器、一个网络接口（NI）芯片、16 MB 或 32 MB 存储器，由一条工作于 400 MBps 的 64 位高速缓存一致的存储器总线连结在一起，如图 7-22 所示。另外，在存储器和网络间成组数据传输提供了两个 DMA 引擎（一个用于发送，另一个用于接收）。DMA传输操作于高速缓存一致性协议之下，在缓冲区溢出或内容跟不上时，

498 ~ 499

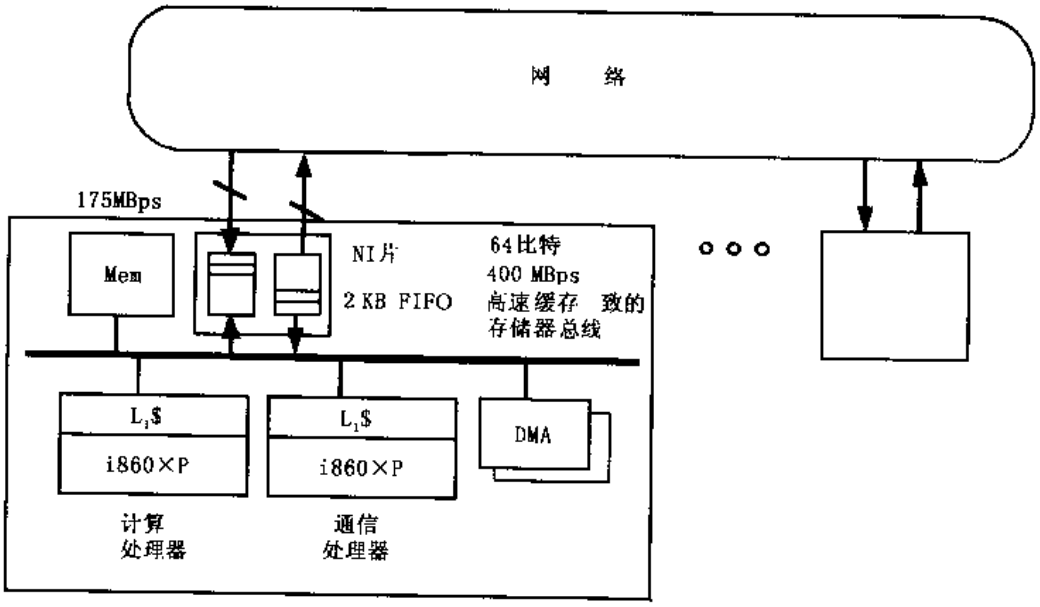


图 7-22 Intel Paragon 机的组成结构。机器的每个节点包括一个专用的、与挂在高速缓存一致的存储器总线的计算处理器相同的 CP，它与快速可靠的网络之间有一个简单的系统级接口，并有两个高速缓存一致的 DMA 引擎照料 NIC 的流控

由网络接口对 DMA 调速。一个处理器被指定为 CP，处理网络事务和消息传递协议，其他的是用作通用计算的计算处理器。通过添加 SCSI、Ethernet 和 HPPI 连接的 I/O 子卡，形成"I/O"节点。

i860XP 通常使用回写高速缓存，也可以在软件和硬件控制下配置成使用直写和一次写策略。写缓冲可以容纳两个连续的存储，避免写扑空时的暂停。i860XP 的高速缓存控制器实现了第 5 章讨论的 MESI（修改、执行、共享、作废）高速缓存一致性协议的变型。外部总线接口也支持 3 级地址流水（即 3 个未决总线周期）和突发模式，传输长度为 2 或 4，速率为 400 MBps。

NI 芯片把一个 64 位同步的存储器总线连接到 16 位的异步（自定时）网络链路。一个 2-KB 的发送 FIFO（tx）和一个 2-KB 的接收 FIFO（rx）用于节点和全双工 175 MBps 的网络链路间的速率匹配。接收 FIFO 的头和发送 FIFO 的尾可以作为存储器映射的 NI 芯片 I/O 寄存器被节点访问。此外，状态寄存器包含一些标记位，反映 FIFO 满、FIFO 空、FIFO 接近满或接近空等状态以及包尾标记出现的情况。在这些标记位之一被置位时，NI 芯片可以选择产生一个中断。对 NI 芯片的 FIFO 的读写是不经过高速缓存的，每次必须读写双字（64 位）。消息的第一个字必须包含路由信息（2D 网格中的 X-Y 位移量），但是硬件对消息的格式不做任何限制。特别地，它不区分用户和系统信息。此外，NI 芯片还进行奇偶校验和 CRC 校验的检查，保证端对端的数据完整性。 500

两个 DMA 引擎，一个用于发送，一个用于接收，可以以 400 MBps 的速度在存储器和 NI 芯片之间传送连续的数据块。存储器区域用对齐于 32 字节边界的物理地址访问，长度为 64 字节到 16 KB（一个 DMA 页），是 32 字节（一个高速缓存块）的整倍数。在 DMA 传输中，DMA 引擎侦听处理器高速缓存来保证一致性。硬件流控防止 DMA 上溢或下溢 NI 芯片的 FIFO 队列。如果输出缓冲满，发送 DMA 会暂停并释放总线。类似地，接收 DMA 在输入缓冲空时也会暂停。总线仲裁器给予 DMA 引擎的优先权高于处理器。一次 DMA 传输的启动是使用 `stio` 指令，把地址和长度写到存储器映射的 DMA 寄存器中。结束时，DMA 在引擎状态寄存器中设置一个标记位并可以选择地产生一个中断。

以这种硬件配置，可以在 10 μs（500 周期）多一点的时间内，把一个小于两个高速缓存块的消息（7 个字）从一个计算处理器的寄存器传到另一个等待的计算处理器的寄存器。这个时间可以被划分成几乎相等的三部分处理器到处理器的传输：计算处理器通过总线到 CP，CP 通过网络到 CP，CP 通过远程节点的总线到计算处理器。令人吃惊的是，高速缓存一致的存储器总线上的两个处理器间的传输和经由网络的两个 CP 间的传输有着相同的时延，特别是处理器和网络接口之间的传输还涉及同一总线上的传输。

让我们更仔细地看一下这个情况。一个 i860 处理器可以用两条 4 字的存储指令把寄存器中的数据写入到高速缓存块中。假定该块的一部分用作满-空标记。在这种典型情况下，对这个块的最后一个操作是消费者的一个清除标记存储指令；采用 Paragon 的 MESI 协议，这个写操作直写到存储器，使生产者的块作废，并使消费者处于独占状态。生产者读取这个标记，发现在生产者的高速缓存中标记扑空，于是从存储器读这个块，并把消费者的块降级为共享态。第一个存储指令直写并作废了消费者的块，但是它使生产者的块处于共享态，因为在执行写操作时检测到了共享。第二个存储指令也是直写，但是因为没有共享者，它使生产者处于独占态。消费者最终读这个标记，扑空，接着读入整个块。所以，一次高速缓存

块传输要求4次总线事务。(通过使用一个额外的标记，允许生产者检测几个块为空，可以将此降为3次总线事务。这留作习题。)数据以一系列不经高速缓存的双字存储被写入网络接口，它们都经过写缓冲流水作业；但是，它们确实涉及多次总线传输。在写数据之前，CP需要检查在输出缓冲区中是否还有空间来写数据。为此，CP检查处理器状态字中对应可屏蔽“输出缓冲空”中断的一个状态位，而不是付出一次不经高速缓存的读的代价。在接收端，CP读取一个类似的“输入非空”状态位，然后用一系列非高速缓存的装入来读取消息数据。在一个无负载网络上，每个实际的NI到NI的传输只需大约250 ns加上每跳40 ns的时间。

501

对于存储器到存储器的块传送，需要CP做一些额外的工作，启动源于用户源区域的发送DMA。这大约需要2 μs的时间（100周期）。DMA以400 MBps的突发方式传送到2 048字节的网络输出缓冲区。当输出缓冲区满时，DMA暂停直到一些高速缓存块被传到网络上。在接收端，CP检测到消息的到来，读取包含目的存储器地址的前几个字，启动接收DMA把消息的其余部分送入存储器。对于大批的传输，在接收DMA引擎正把数据移入存储器时，发送DMA引擎正在把数据移出存储器，传输的部分占据它们之间的缓冲区和网络链路。此刻，消息在网络链路上以175 MBps的速率向前传输。发送DMA引擎和接收DMA引擎周期性地介入，以400 MBps的突发速率把数据移进和移出缓冲区。

对CP的要求显示，它负责对CP必须采取动作的大量的独立事件做出响应。这些包括当前用户程序将消息写入共享队列，计算处理器内核程序把消息写入类似的“系统队列”，网络把消息送入NI输入缓冲区，网络接受消息后NI输出缓冲区变为空，发送DMA引擎结束以及接收DMA引擎结束。CP的带宽由它检测各种事件并根据各种事件而分支所花费的时间决定。当处理其中任何一个事件时，所有其他事件被有效地锁在外面。引入DMA引擎这样的额外硬件来最小化处理任何特定事件的工作量，允许数据以一种完全流水的方式从源存储区域（寄存器或存储器）流动到目的存储区域。但是，通信速率（每秒消息数）依然被CP的串行分支循环所限制。此外，CP上保持数据流动，避免死锁，避免网络饥饿的软件是相当复杂的。逻辑上讲，它包括一系列协同的独立线程，但是它们都塞进一个单一串行分支循环中，该循环跟踪每个部分操作的状态。我们在下一个案例中研究这个并发问题。

ASCI Red Machine采用了Paragon的基本体系结构，ASCI Red Machine是第一台达到持续的TFLOPS（每秒一万亿次浮点操作）的机器。它包含4 536个节点，每个节点有两个200-MHz的Pentium Pro处理器和64 MB的存储器。它采用具有400 MBps的链路的升级的Paragon网络，依然使用网格拓扑结构。机器分布在85个机柜中，占地1 600平方英尺，耗电800 kW。其中40个节点提供对大规模RAID存储系统的I/O访问，32个节点提供对各个独立节点上的轻量内核操作的操作系统服务，16个节点提供“热”备份。

502

许多机群设计采用SMP系统节点作为基本的构造块，使用高性能的LAN或SAN相连。这种方法或者专门用一个处理器来处理消息，或由消息流量请求一个处理器承担这个责任。一个关键的区别在于像Paragon所使用的这样的网络，在进入的事务未被节点处理之前会停顿，所有的通信（包括系统消息）都停止前进。所以，指定一个处理器来处理消息提供了一个更为健壮的设计。（在一些特殊情形，例如试图创造LINPACK基准测试程序的记录时，Paragon和ASCI Red甚至把两个处理器都用于用户计算。）机群通常依赖其他的机制，例如网络接口卡中的专用处理，来保持通信流动，这将在7.7节中讨论。

7.5.2 案例分析：Meiko CS-2

Meiko CS-2 提供了具有非对称 CP 的代表性具体设计。CP 与网络接口紧密集成并有一条专用路径通到网络。节点体系结构基本上是典型的 Sun SparcStation 10 体系结构，在 MBUS 上有两个标准的超标量 Sparc 模块，每个模块包含一个在片一级高速缓存 L_1 和一个模块上的二级高速缓存 L_2。经由总线适配器可以访问 MBUS 上的以太网、SBUS 和 SCSI 连接器来提供 I/O。（节点体系结构的一种高性能改型包括两个 Fujitsu μVP 向量部件，它们共享一个 3 端口存储器系统。第 3 个端口连接 MBUS，和基本节点一样，MBUS 上有两个计算处理器和通信模块 ）通信模块的功能可以是 MBUS 上另一个处理器模块或存储器模块，这取决于它的操作。网络链路在每个方向提供 50 MBps 的带宽。这台机器在网络事务的解释和通信处理中的并发支持方面有独特的做法。

Meiko CS-2 中，一个网络事务是一个传过网络并由远程 CP 直接执行的代码序列。网络是电路交换的，这意味着一个通道被建立，并在网络事务执行过程中保持开放。如果通道被建立，并且事务成功地进行到结束，一个确认使通道关闭。如果不能建立连接，或发生了 CRC 错误，或远程执行超时，或条件操作失败，会返回一个 NACK。网络事务的控制流是具有条件中止的顺序代码，没有分支。超时的典型原因是远端的缺页。能被包括在网络事务中的操作包括远地存储器的读、写、读-修改-写，设置事件、简单测试，DMA 传输以及简单的应答传输。因此，网络事务中信息的格式相当多。它包括一个现场标识符、一个起始符、一系列具体格式的操作和一个结尾符。事务有 40 ~ 320 个字节长。在考察机器组成结构之后，再来看网络事务所支持的操作。

503

基于前面的讨论，把 CP 分解成几个独立的处理器是有意义的，如图 7-23 所示。一个命令处理器（P_{cmd}）等待用户或系统发出的通信的命令并对它执行。因为它是存储器总线上的设备，它可以直接响应对它负责的地址的读和写。而不像常规的处理器那样要轮询一个共享存储器单元。它通过把路由信息和数据推给输出处理器（P_{out}），或者把数据从存储器传送到

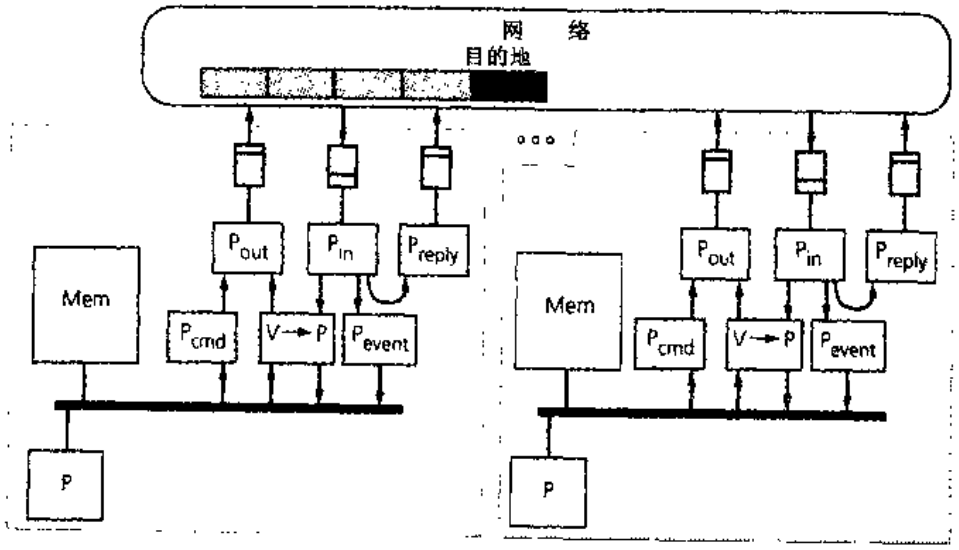


图 7-23　带有多个专由通信处理器的 Meiko CS-2 概念性结构。生成和处理网络事务的每一个具体侧面都结合到独立运行的硬件功能部件之中

输出处理器完成它的工作。它可能需要负责虚实（V→P）地址转换的部件的帮助。它也提供用户到用户通信所需要的保护检查。输出处理器 P_{out} 监视网络输出 FIFO 的状态，把网络事务递交到网络。输入处理器（P_{in}）等待网络事务的到达并执行它。可能会把数据送入存储器，向事件处理器（P_{event}）发出一条命令，指示事务的结束；也可能对应答处理器（P_{reply}）发出应答操作，后者的操作和输出处理器非常类似。

Meiko CS-2 提供这些独立的功能，虽然它们是作为一个叫做 elan 的微程序处理器上的分时复用的线程运行的（Homewood and McLaren 1993）。这使得逻辑处理器之间的通信十分简单，并提供了一个清晰的概念性结构，但是它并没有真正使所有的信息流平滑地处理。图 7-24描述了实际的 elan 的功能组成结构。计算处理器通过一条交换指令（swap）向命令处理

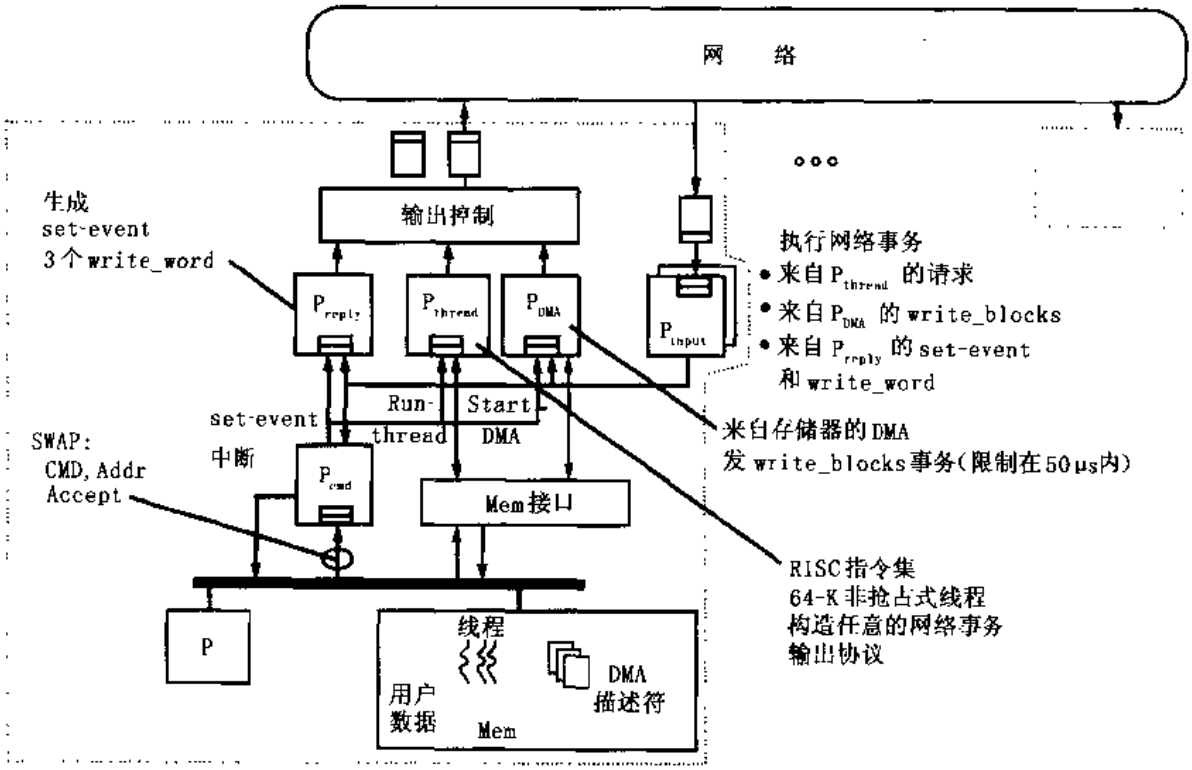


图 7-24 Meiko CS-2 机的组成结构。通信辅助部件以单个微程序处理器上的时间片的形式，提供 5 个简单的处理器，其中的 4 个专门用于特定的功能：从主机接收命令，从网络接受事务，执行 DMA 传输，发出应答。另外一个线程处理器执行用户级代码以产生网络事务并向其他处理器发出请求

器发出一个命令，交换指令交换寄存器和存储器单元的值。存储器单元被映射到命令处理器输入队列的头。返回给处理器的值指明进队命令是否成功，或者队列是否已经满了。给命令处理器的值包含命令类型和虚拟地址。命令处理器支持三种命令：start-DMA，这种情况下，地址指向一个 DMA 描述符；set-event，这种情况下，地址指向一个简单的事件数据结构；start-thread，这种情况下，地址指向线程的第一条指令。DMA 处理器从存储器中读数据并产生一系列网络事务，把数据存储到远程节点。命令处理器也执行事件操作，包括更新一个小的事件数据结构和对主处理器发出一个中断，以便来唤醒一个休眠的线程。start-thread 命令被发送给一个简单的 RISC 线程处理器，该线程处理器执行任意的指令序列来构造和发出网络事务。网络事务由输入处理器解释，它可能引起线程执行，启动 DMA，发出应

答或设置事件。应答是一个简单的事件设置（set-event）操作，具有一个可选的3个数据字的写。

504 ~ 505

为了使机器操作更加具体，下面考虑几个简单的操作。假设一个用户进程想把数据写进同一个并行程序的另一个进程的地址空间。两个进程共享一个通信现场的能力提供了保护。源计算处理器建立了一个DMA描述符并发出一个启动DMA（start-DMA）命令。源DMA处理器读这个描述符，并把数据作为一系列的块传送，每个块都包括从存储器中读取的32个字节的数据，并形成一个写块（write_block）的网络事务。远程节点的输入处理器会收到并执行一系列write_block事务，每一个事务都包括一个用户虚拟存储器地址和要写到那个地址去的数据。从远端的地址空间读取一块数据更加棘手。首先在本地CP上启动一个线程，该线程发出一个start-DMA事务。远程节点的输入处理器把start-DMA及其描述符传递给远程节点的DMA处理器，后者会从存储器中读取数据并通过一系列的write_block事务将数据返回。为了检测结束，可以增加一个事件设置（set-event）操作。

为了支持直接的用户到用户的传输，Meiko CS-2的通信处理器含有自己的页表。主处理器的操作系统将这个页表与正常页表保持一致。如果CP碰到缺页，就会产生一个中断，操作系统调入页并更新页表。

这个设计的主要缺点是线程处理器相当慢，并且是非抢占式调度的。这使得除了一些琐碎的处理之外，很难把其他处理放到线程处理器中去执行。此外，网络事务提供的操作集也不是足够强大，不能用单一的网络事务构造有效的远程进队操作（Schauser and Scheiman 1995）。

7.6 共享的物理地址空间

本节考察可扩展多处理器的通信体系结构的第4种主要设计风格——共享的物理地址空间。它直接建立在较小规模的共享存储器的机器上，并提供了相同的通信原语：装入、存储、对共享存储器的原子操作。许多机器把这种方法扩展到大规模的系统，包括CM*、C.mmp、NYU Ultracomputer、BBN Butterfly、IBM RP3、Denelcor HEP-1、BBN TC2000和CRAY T3D（Scott 1996）。大多数的早期设计采用舞厅组织结构，在存储器和处理器之间设置互连网络，而大多数较晚的设计采用分布式存储器结构。通信辅助部件把总线事务转换为网络事务。网络事务是非常特殊的，因为它只是说明预先定义的存储器操作集合，并由远程节点的通信辅助部件直接解释。

图7-25显示了一个大规模分布共享物理地址空间的机器的通用组成结构。我们最好认为通信辅助部件构成一个*伪存储器模块*和一个*伪处理器*，集成为处理器-存储器连接。例如，考虑节点上的处理器执行的一条load指令。片上的存储器管理部件（MMU）把虚拟地址转换成一个全局的物理地址，提交给存储器系统。如果该物理地址落在发出请求的节点，那么存储器简单地送回指定单元的内容。否则，通信辅助部件就会访问远程节点，表现得像一个存储器模块。伪存储器控制器接受存储器总线上的读事务，从全局地址中提取远程节点号，对远程节点发送一个网络事务，访问所希望的存储器单元。注意，此刻，load指令会停留在存储器操作的地址阶段和数据阶段之间。远程的通信辅助部件接收网络事务，读指定的存储器单元，并对发出请求的原始节点送回响应事务。远程的通信辅助部件在对存储器发出代理读请求时，对该节点的存储器来说就像一个伪处理器。需要指出的重要的一点是当伪处理器

506

试图代表一个远程节点访问存储器时，当地的主处理器可能也暂停在它自己的远程 load 指令的中间。支持一个未决操作的简单存储器总线不足以应付这个任务。必须有两条独立的通往存储器的路径，或者总线必须支持乱序结束的分裂阶段操作。最终，响应事务到达原始的伪存储器控制器。它就像一个慢的存储器模块一样完成这个存储器读操作。

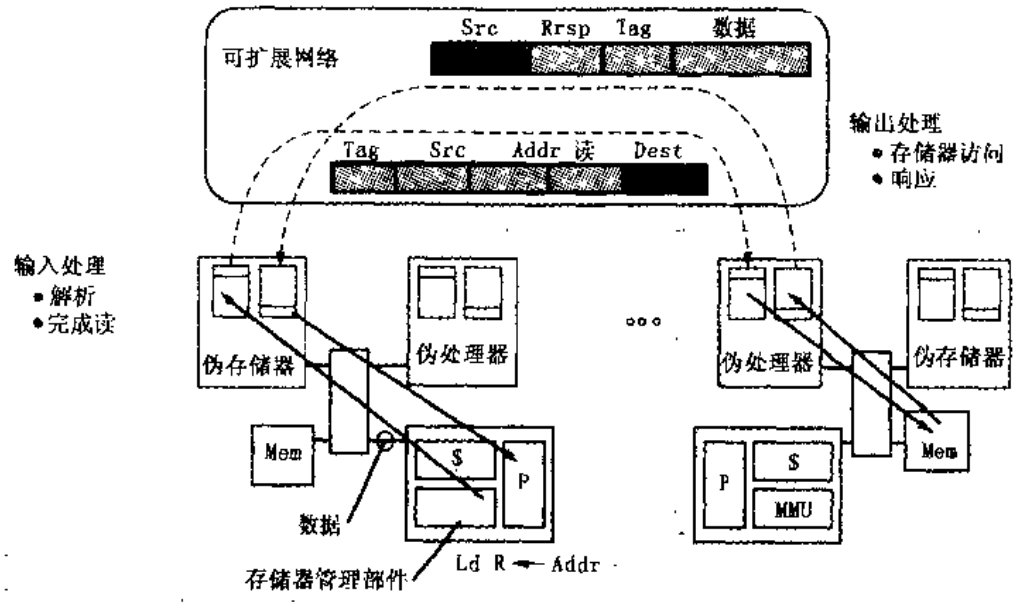


图 7-25 共享物理存储器的组织结构。在可扩展共享物理地址空间的机器中，网络事务由常规的存储器指令产生。它有一组固定的格式，由通信辅助部件的硬件直接解释。操作是请求-响应类型，多数系统为了避免取指死锁提供了两套不同的网络。通信体系结构在发出事务的节点扮演伪存储器模块的角色，在接收节点扮演伪处理器。远程存储器操作被发出方的伪存储器单元接收，携带了一个与远程节点的请求-响应事务。源总线事务在网络事务请求、远程存储器存取和网络响应事务过程这段时间内保持开放。所以，通信辅助部件必须能够访问远程节点的本地存储器，即使那个节点的处理器暂停在它自己的存储器操作中间

在下一章要深入讨论的一个关键论题是共享存储器单元的高速缓存问题。在大多数现代微处理器中，地址的高速缓存能力是由所在页的页表项的一个字段决定的。当该单元被访问时，从 TLB 中抽取该子段。在我们的讨论中，重要的是区分两个正交的概念。一个地址，或者是进程私有，或者是多个进程共享，对于处理器而言它或者是本地的物理地址，或者是远程的物理地址。很明显，进程私有，对进程所在处理器而言本地物理的地址必须被高速缓存，这不需要特殊的硬件支持。物理远程的私有数据也可以被高速缓存，虽然这要求通信辅助部件支持高速缓存块事务，而不是单个的字。对远程块高速缓存不需要处理器有任何变化，因为远程存储器访问等同于本地存储器访问，只不过慢一些。但是，只要进程保持不变，也不存在高速缓存一致性问题，因为没有其他进程访问私有数据。如果物理本地并且逻辑共享的数据在本地缓存，那么只要求伪处理器在代表远程节点执行存储器写入时，将高速缓存的数据作废。如果以回写模式缓存共享数据，那么伪处理器必须能腾空高速缓存。最自然的方法是把伪处理器集成在高速缓存一致的存储器总线上，但是总线必须是分裂阶段的，以便为伪处理器保留一些待完成事务。最后一个选择对共享的远程数据进行高速缓存。访问、传送、远程块在本地高速缓存的放置所要求的硬件支持已经由前面那些选择所覆盖。新

的问题是如何是使块在各个高速缓存中的大量复本保持一致。我们必须应付这个带有复制的分布式共享存储器的一致性模型。这些问题要求更多的设计上的考虑，我们将在下两章中专门讨论。对共享的远程数据实行高速缓存，以便在多数情况下可以本地访问它们，从性能角度考虑显然是很有吸引力的。

7.6.1 案例分析：CRAY T3D

CRAY T3D 提供了一个共享全局物理地址空间的设计范例。该设计遵循图 7-13 中的指导原则，经由支持请求和响应事务独立提交的可扩展网络，伪存储器控制器和伪处理器提供远程存储器访问。有硬件直接解释的 7 种特定网络事务格式。但是，该设计以几种有效的途径扩展了基本的共享物理地址空间的方法。T3D 系统可以扩展到 2 048 个节点，每个节点配有一个 150 MHz 的双发指令 DEC Alpha 21064 的微处理器和 64 MB 的存储器，如图 7-26 所示。DEC Alpha 体系结构作为并行体系结构的一个基本构件块使用（Digital Equipment Corporation 1992），21 064 的几个特征显著影响了 T3D 的设计。在这个案例分析中，我们首先看一下处理器本身和局部存储器的突出特征。然后，我们讨论一下围绕基本处理器构造，提供共享物理地址空间、时延容忍、块传输、同步和快速消息传递的辅助部件。有时，CRAY 的设计者称它为环绕着实现并行处理能力的常规微处理器的一个“外壳”支持电路。

507 ~ 508

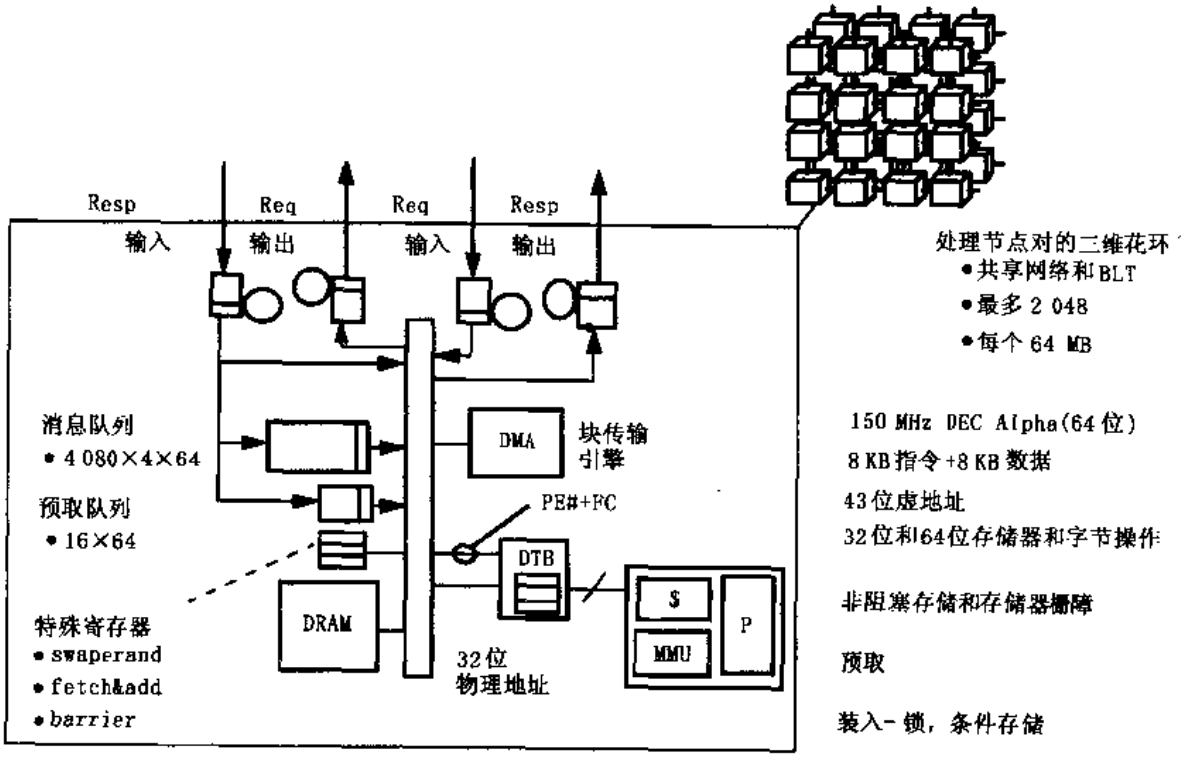


图 7-26 CRAY T3D 的机器结构。每个节点包含一个精巧的通信辅助部件，它包括共享物理地址空间要求的伪存储器和伪处理器功能。此外，提供了一组外部段寄存器（DTB），扩展了机器有限的物理地址空间。预取队列通过显式的预读，支持时延隐藏。消息队列支持与消息传递模式相关的事件。DMA 部件提供块传输能力，支持特殊的点和全局的同步操作。机器按 3 维花环组织，正如它的名字所言

Alpha 21064 具有片载 8 KB 数据和 8 KB 指令高速缓存以及对外部二级高速缓存 L_2 的支持。在 T3D 设计中，取消了 L_2 高速缓存以降低访问主存的时间。在高速缓存扑空时，处理 509

器会暂停，所以降低扑空时延直接提高了可用带宽。CRAY 的设计偏向于典型向量代码的存取模式，这种模式会扫过大片的存储器区域。在 T3D 上，测量到的因扑空而访问存储器的装入（load）指令访问时间是 155 ns（23 个周期），与此比较，在具有 512 KB L_2 高速缓存和相同时钟频率的DEC Alpha工作站是300 ns（45个周期）。（如果访问跨DRAM的页，CRAY T3D 的访问时间增加到 255 ns。）对于校准全局存储器访问的性能，这个测量是非常有用的。

根据 Alpha 体系结构，Alpha 21064 提供了一个 43 位的虚拟地址空间，但是，物理地址空间只有 32 位。因为虚拟地址到物理地址转换在片上进行，所以存储器和通信辅助部件收到的是物理地址。对于一个最大配置的 2048 个节点的系统，如果每个节点都有 64 MB 存储器，需要 37 位的全局物理地址。为了放大每个节点上的物理地址空间，T3D 提供了一个外部寄存器组，叫做 DTB Annex，用 5 位物理地址来选择包含 21 位节点号的寄存器，和另外 27 位本地物理地址相拼接，形成完整的全局物理地址[⊖]。annex 的寄存器还包含说明访问类型的额外的域，例如，是否可以高速缓存。annex 的 0 号寄存器总是指向本地节点。T3D 使用 Alpha 的装入-锁（load-lock）和条件存储（store-conditional）指令读出和写入 annex 寄存器。更新一个 annex 寄存器需要 23 个周期，好像一个对片外存储器的访问，后面可以紧跟一条使用 annex 寄存器的装入（load）或存储（store）指令。

对全局地址空间一个单元的读或写通过一个短的指令序列实现。首先，全局虚拟地址的处理器号部分被抽取出来，并存入一个 annex 寄存器。然后，构造一个临时虚拟地址，其高位指向这个 annex 寄存器，低位说明节点内的地址。最后，对这个虚拟地址执行一条装入或存储指令。装入指令需要 610 ns（91 个周期），不包括 annex 设置和地址处理。（如果远程 DRAM 访问跨页，这个数字增加 100 ns（15 个周期）。此外，如果导致取整个高速缓存块，增加 785 ~ 885 ns。）

记住，虚拟-物理地址转换发生在发出读的处理器上。建立页表，从而使寄存器号在虚拟地址到物理地址转换中简单地保持。这样，产生的物理地址在远程节点才有意义，并行程序中所有进程的物理偏移量是相同的（不支持分页）。此外，需要确保并行程序的所有进程把它们的堆扩展到同一长度。

510

Alpha 21064 只提供非阻塞的存储。在一条存储指令之后，无须等待存储完成，后续执行就可以继续。写指令在写缓冲区缓冲，写缓冲区深度为 4，每个表项中可以合并多达 32 个字节的写数据。可以由几条存储指令待完成。提供“存储器栅障”指令来保证在其他执行开始之前写已经完成。Alpha 的非阻塞写允许远程的存储重叠，从而对远程存储器的写能达到高带宽。每 250 ns 可以从写缓冲发出多达一个高速缓存块的远程的写，提供从本地高速缓存到远程存储器的 120 MBps 的传输带宽。单块的远程写需要一系列操作，包括发出存储，通过存储器栅障指令把存储推出写缓冲区，然后等待网络接口的一个完成标记。这需要 900 ns 的时间，外加 annex 的建立和地址运算时间。

Alpha 提供了一个特殊的预取指令，用来鼓励存储器系统将重要的数据移近处理器。这在 T3D 中被用来隐藏远程的读时延。在片外提供了一个 16 个字的预取队列。预取从存储器

⊖ 这种情况并不罕见，C.mmp 使用类似的技巧来克服其构造模块 LSI-11 寻址能力不足的问题。由此产生的问题带来了一个归功于 Gordon Bell 和 Bill Wulf 的著名的论点：一个体系结构准一难以克服的毛病是寻址空间太小。

中读取数据到队列中。读取队列是弹出它的头位置的字。发出预取的指令和存储指令一样，只需要几个周期。弹出队列操作需要 23 个周期，是片外操作的典型值。如果 8 个字被预取，然后弹出，网络的时延就完全被隐藏起来，每个字的实际时延将小于 300 ns。

T3D 也提供块传送引擎，它能够在本地节点和远程节点之间双向移动块或规则跨距的数据。从远程节点读时，块传输带宽的峰值为 140 MBps；对远程节点的写，峰值带宽为 90 MBps。但是，使用块传输引擎需要进入内核，提供虚拟地址到物理地址的转换。所以，对于传输不超过 64 KB 的数据，预取提供了更好的性能，非阻塞存储对任何长度都更快。块传输引擎的主要优势是能够重叠通信和计算。因为处理器和块传输引擎竞争同一存储器带宽，这种能力受到了某种程度的限制。

T3D 的通信辅助部件也提供对同步的特殊支持。首先，有一个专用的网络提供全局或操作和全局与操作，主要用于栅障指令。这允许处理器发出一个表明它们碰到了栅障标记，继续执行，然后在离开前等待所有处理器的进入。每个节点都有一组外部同步寄存器，用来支持原子的 swap 和 fetch&inc。也存在用户级的消息队列操作，使消息入队，或者调用一个远程节点的线程。不幸的是，这两种操作都会引起远程内核的陷入，所以这两种操作分别花费 25 μs 和 70 μs 的时间。与之比较，使用 fetch&inc 操作在存储器中建立一个队列，允许在 3μs 内将一个 4 字的消息入队，在 1.5 μs 内出队。

511

7.6.2 案例分析：CRAY T3E

CRAY T3D 之后的 CRAY T3E（Scott 1996）阐明了大规模系统设计折中。设计的两个驱动力是在节点中使用更强大、更现代的处理器和简化外部电路。CRAY T3D 为支持类似的功能使用了许多复杂的机制，每个机制都有不同的优点和缺点。T3E 采用 300 MHz 的、4 发指令的 Alpha 21164 的处理器，带有相当大（96 KB）的二级在片高速缓存。因为 L_2 高速缓存在片上，不能像 T3D 那样取消它。但是，T3E 放弃了在典型的基于 Alpha 21164 的工作站具有的板上三级高速缓存。各种远程访问机制被合并为单一的外部寄存器的概念。此外，使用虚地址访问远程存储器，由远程通信辅助部件把虚拟地址转化为物理地址。

一个用户进程可以访问一组 512 个 64 位的 E-寄存器。处理器可以对存储器空间特定区域使用常规的装入和存储，读出或写入 E-寄存器的内容。还提供了一些操作，从全局存储器中取数据送入一个 E-寄存器，将 E-寄存器的内容送入全局存储器，执行 E-寄存器和全局存储器之间的读-修改-写操作。把远程数据装入 E-寄存器包括三步。首先，在 E-寄存器中构造该处理器的全局虚拟地址部分。其次，通过把命令向存储器的特定区域存储而发出 get 命令，命令的存储使用的一个地址字段说明了 get 命令，另一个字段指定目的数据 E-寄存器。命令存储的数据说明了一个相对于地址 E-寄存器的偏移量。命令的存储产生的副作用是，执行了远程的读，数据被装入目的 E-寄存器。最后，数据通过读 E-寄存器被读入处理器。远程的 put 命令的过程与之类似，只是要写的数据放置在数据 E-寄存器中，该 E-寄存器是由 put 命令的存储指明的。利用对地址寄存器操作的副作用产生对数据寄存器的读出和写入的方法可以追溯到 CDC 6600（Thornton 1964），虽然它几乎已经被遗忘了。

通过对每个 E-寄存器附加一个满-空位，在 E-寄存器中提供了预取队列的功能。发出一系列的 get 命令，每一个 get 都把相应的目的 E-寄存器置为空。当 get 完成时，寄存器被置为满。如果处理器试图读一个为空的 E-寄存器，存储器操作被暂停，直到 get 结束。允许以单

个操作将4或8个字的向量传过E-寄存器集，来提供块传输引擎的功能。这还有提供有效的汇集操作的额外的好处。

T3E的改进大大简化了机器的代码生成，带来几个性能上的优点；但是，它们决不是均匀的。因为用了更快的处理器和更大的在片高速缓存，T3E的计算性能比T3D有显著提高。另一方面，远程读的时延是T3D的两倍多，从600 ns增长到大约1500 ns。这种增长是因为I_2高速缓存扑空和远程地址转换引起的。两种机器的远程写时延基本相同。预取的代价大约改善了两倍，得到每130 ns读一个字的速率。每个存储器模块每67 ns可服务一次读操作。非阻塞写在两台机器上的性能基本相同。T3E的块传输能力远比T3D强。可以获得300MBps以上的带宽而没有大的块传输引擎启动代价。成块写的带宽超出300MBps，是T3D的三倍。

512

7.6.3 小结

在现代大规模的并行机中，存在不同程度的网络事务的硬件解释。这些变化产生计算处理器在执行通信操作时非常不同的额外开销，以及通信辅助部件对实际网络所增加的非常不同的时延。通过限制通信事务的集合，使通信辅助部件专门解释这些事务，并把通信辅助部件和节点的存储器系统紧密集成在一起，可以使额外开销和时延有实质性的降低。专门的硬件也可以提供通信辅助部件所需要的并发性，使它能以高带宽处理几个同时的事件流。

7.7 工作站机群和工作站网络

随着商品化微处理器、存储器器件、甚至工作站操作系统在现代大规模并行机中的使用，与并行机中所使用的很类似的可扩展通信网络已经可以用于有限的局域网中。这样自然引起一个问题，在什么程度上，工作站网络（NOW）和工作站机群能汇合到一起？在回答这个问题之前，需要一点背景知识。

传统上说，使用专用互连连接完整计算机称为*机群*（clusters），用于多道程序的工作负荷，改善系统可用性（Kronenberg，Levy，and Strecker 1986；Pfister et al. 1985）。在多道程序的机群中，一台前端机在一组计算服务器和大量的终端或远程机器上的用户之间，通常起到中介的作用。前端机跟踪机群节点的负荷并把任务调度到负荷最轻的节点上。典型地，机群中的所有计算机在功能上设置成等同的，它们有相同的指令集、相同的操作系统和相同的文件系统访问。在较老的系统中，例如Vax.VMS机群（Kronenberg，Levy，and Strecker 1986），这是通过把每台机器连接到一组公共的磁盘来实现的。最近，这种单一系统映像通常通过在网络上安装公共的文件系统实现。通过共享功能等价的机器池，对大量独立任务可以达到更好的利用率。

513

*可用性机群*寻求降低大型关键系统的失效时间，例如，重要的在线数据库和事务处理系统。结构上讲，它们和多道程序的机群有很多共同点。一个非常常见的做法是用一对SMP系统运行具有共享磁盘组的数据库系统的相同副本。如果主系统由于硬件或软件问题失效，操作就会快速切换到备份的系统上去。提供这种共享磁盘能力的实际互连可以是对磁盘的双路访问或某种专用网络。

机群越来越多地被用做并行机，经常称其为*工作站网络*（NOW）。对机群的一个主要影响是出现了流行的公共领域软件，例如Condor（Litzkow，Livny，and Mutka 1988）和PVM（Geist et al. 1994），这些软件允许用户在一组机器上分布作业，或在由任意的局域网甚至广域网连

接的大量的机器上运行并行程序。尽管通信的性能不好，对于小的传输的典型时延在毫秒级甚至更高，集合带宽经常小于 1 MBps，但是这些工具为一类具有很高计算通信比的问题提供了一个价廉的手段。

显示了机群在大型并行机方面的潜力的技术突破是可扩展的、低时延的互连，质量上类似于并行机所用的互连网络，但按照局域网部署。在三个基本的方向上出现了几种可能的网络。传统局域网通常是具有固定带宽的共享总线（例如以太网）或环（如令牌环和 FDDI），或者是专用点对点的连接（例如 HPPI）。为了支持大量的快速的机器提供可扩展的带宽，趋势是采用基于交换的局域网（例如，HPPI 交换机、FDDI 交换机［Lukowsky and Polit 1997］和 FiberChannels）。一个重要的发展是电信产业开发的，作为交换局域网的 ATM（异步传输模式）标准的广泛采用。几家公司提供了具有 16 个 155-MBps 端口（19.4 MBps）链路带宽的 ATM 交换机。可以将它们级连，构成更大的网络。在 ATM 方式下，一个可变长度长的消息以一系列 53 个字节的码元（48 字节的数据，5 字节的路由信息）的形式，在预先分配的叫做“虚电路”的路径上传输。我们将在第 10 章更详细地考察这些网络技术。根据 7.1 节所发展的模型，当前的 ATM 交换机的典型路由延迟在无载网络中大约是 10 μs，虽然某些还要更高。第二种主要的标准化努力的代表是可伸缩一致性互连（scalabe coherent interconnect，SCI）提供的标准，它包括物理层标准和一个特殊的分布式高速缓存一致性策略。第三种是大量应用的快速交换以太网和千兆以太网标准。

目前出现了一个强烈的趋势，就是把在 MPP 系统中使用的专用网络发展成为能在较大区域连接大量独立的工作站或 PC 机的网络。这方面的例子从包括 Tandem Corporation 的 ServerNet（Horst 1995）和 Myrinet（Borden et al. 1995）。Myrinet 交换机提供 8 个端口，每个端口 160 MBps，可以用规则或不规则的拓扑把交换机级连，形成大的网络。它传输可变长的数据包，每跳的路由延迟是 350 ns。使用链路级的流控，避免在出现竞争时丢数据包。 514

正如更紧密集成的并行机的情况一样，正在出现的 NOW 和机群中的硬件原语还是一个开放的论题，需要许多的争论。在这些先进网络上的传统 TCP/IP 通信抽象表现出大的额外开销（毫秒级或更高）（Keeton，Anderson，and Patterson 1995），在许多情况下比常规以太网还要大。使用 TCP/IP 仅仅以 20MBps 传输就需要一个非常快的处理器。但是，带宽确实随机器数增加而扩展，至少在没有什么竞争的情况下。已经建议了几种效率更高协议，包括主动消息（Anderson，Culler，and Patterson 1995；von Eicken et al. 1992；von Eicken，Basu，and Buch 1995）和反射存储器（Gillett 1996；Gillett and Kaufmann 1997）。如前所述，主动消息提供了用户级的网络事务。反射存储器允许对特定的存储器区的写表现得就像写入了远程处理器的区域；没有读远程数据的能力。但是，在存在潜在的不可靠性情况下支持一个真正的共享物理地址空间，仍然是一个未解决的问题。一个中间的策略是把连接一组相互通信的进程的逻辑网络连接看作一个全相连的队列的组。每个进程具有一个通信端点，包括发送队列、接收队列和一些状态信息，比如在消息到达时是否提交通知。每个进程都可以对任何接收队列递交消息，即把一个具有适当的目的标识符的消息放入它的发送队列即可。这种方法正在被 Intel、Microsoft、Compag 所领导的业界论坛标准化，其名称是虚拟接口体系结构（Dunning et al. 1998）。其基础是几个研究项目，包括 Berkeley 大学的 NOW，Cornell 大学的 UNET，Illinois 大学的 FM 和 Princeton 大学的 SHRIMP。

在机群和 NOW 中的通信接口的硬件支持和网络事务的解释覆盖了在前面讨论过的大部

分的设计观点。但是，因为网络是接插到现存的机器，而不是在板级或芯片级集成到系统里，所以，它必须与 I/O 总线而不是与存储器总线或靠近处理器的地方与系统接口。在这个领域，也有相当多的创新。已经开发了几种相对快速的 I/O 总线，它们保持高速缓存的一致性，最值得一提的是 PCI。正在试验把网络通过图形总线集成（Martin1994；Banks and Prudence 1993），还有 SIMM 附加（Minnich，Burns，and Hady 1995）。

一个推动机群进步的重要技术力量就是出现了相对便宜的 SMP 构造模块。例如，通过很小的努力就能把几十个 4-Pentium Pro 处理器的商品化服务器集合起来，产生一个相当大的并行机。在高端，大多数非常大型的并行机是采用最大可用的商品 SMP 优化构造的机群。例如，在 1997～1998 年美国能源部作为加速战略计算计划（Accelerated Strategic Computing Initiative）的一部分购买的 Intel 机，就是由 4 536 个双 Pentium Pro 模块组成的。IBM 机可以是 512 个 4 路的 PowerPC 604，后升级为 8 路的 PowerPc 630。SGI/CRAY 机最初是用许多 HPPI 6400 链路互连的 16 个 32 路的 Origin，最终集成为更大的高速缓存一致的部件，我们会在第 8 章讨论。

515

7.7.1 案例分析：Myrinet SBUS Lanai

图 7-27 显示了一个正在出现的 NOW 的实例。一组 UltraSparc 工作站经由智能网络接口卡（NIC），使用 Myricom 的 Myrinet 可扩展网络集成起来。让我们从基本的硬件操作开始向上考察。该网络说明了相对于密集封装的并行机网络或者分布范围广的局域网（LAN），什么是系统域网（SAN）。链路是并行的铜绞线（18 位宽），几十英尺长，这取决于链路的速度和电缆的类型。通信辅助部件采用专用的通信处理器，类似于 Meiko CS-2 和 IBM SP-2 的方案。NIC 包含一个内嵌的 Lanai 处理器，用来控制主机和网络间的消息流。在机群设计中一个关键的不同之处是 NIC 上有一个相当尺寸的 SRAM 存储器。所有主机和网络间的消息数据都要经过 NIC 存储器。该存储器也用作 Lanai 处理器的指令和数据存储器。在 NIC 上有 3 个 DMA 引擎，一个用作网络输入，一个用作网络输出，一个用于主机和 NIC 存储器之间的传输。主机的处理器可以使用常规的装入和存储指令访问地址空间特定区域，即通过程序 I/O 来读写 NIC 存储器。NIC 处理器使用 DMA 访问主机的存储器。内核划定一块 NIC 可以访问的主机存储器区域。对于短的传输，主机直接把数据移入移出 NIC 是效率最高的，而对于长的传输，主机最好把地址写入 NIC 存储器，让 NIC 拾取这些地址，用它们来设置 DMA 传输。Lanai 处理器可以读写网络的 FIFO 队列，或通过 DMA 操作驱动 FIFO 和 NIC 存储器之间的传输。

Lanai 执行的固件程序主要通过协调 DMA 传输来管理数据流，响应主机写入的命令和从网络到达的包。典型地，命令写入 NIC 存储器，被 NIC 的处理器拾取。NIC 根据要求从主机传输数据，把它们推入网络。Myricom 网络采用基于源的路由，所以数据包的头包括了给通往目的地路径上每个网络交换机的简单路由信息。目的 NIC 把数据包接收到 NIC 存储器。它可以检查事务中的信息并按要求处理它，提供通信抽象。

NIC 的实现有 4 个基本部件：总线接口、链路接口、SRAM 和 Lanai 芯片，在芯片上有处理器、DMA 引擎和链路 FIFO。链路接口把片上的 CMOS 信号转换成长距离绞线上传输的差分信号。设计的一个关键方面是到 NIC 存储器的带宽。3 个 DMA 引擎和处理器共享内部总线，这在 Lanai 片内实现。网络 DMA 引擎要求 320 MBps 带宽，主机 DMA 在 SBUS 上要求短的 100 MBps 阵发带宽，在 PCI 总线上要求长的 133 MBps 突发带宽。固件的设计目标就是能

516

保持 3 种 DMA 引擎同时工作；但是，有一点复杂，因为一旦启动了 DMA 引擎，它的可用带宽及它的执行速率就因为与 SRAM 的竞争而显著减低了。

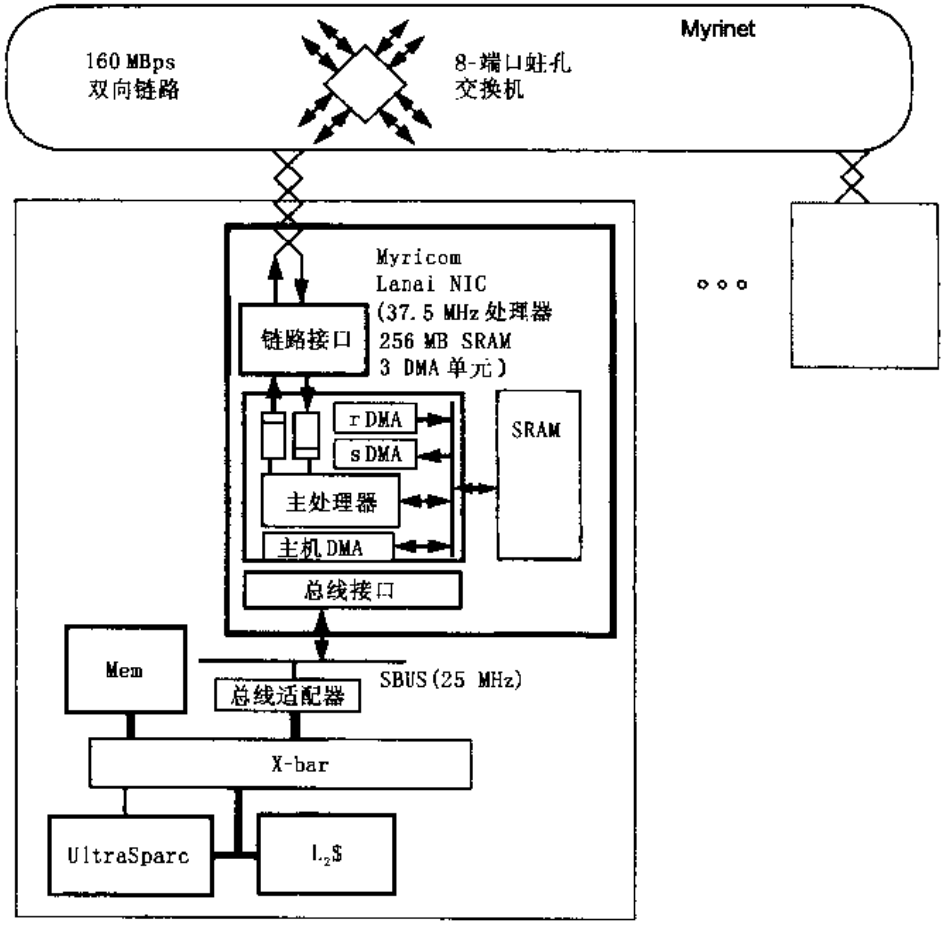


图 7-27　采用 Myrinet 和内嵌处理器的专用消息处理的 NOW 组织结构。尽管机群的节点是完整的常规计算机，但在与可扩展、低时延网络的接口中可以提供一个复杂的通信辅助部件。典型地，网络接口挂在常规的 I/O 总线上，但是厂商越来越倾向于通过节点体系结构提供更紧密的集成手段。在很多情况下，通信辅助部件提供了网络事务的专门处理

典型地，NIC 存储器逻辑上划分成一组功能不同的区域，包括指令存储区、内部数据结构、消息队列和数据传输缓冲区。NIC 存储器空间的每一个页由主机的虚拟存储器系统独立映射。所以，只有内核才能访问 NIC 处理器的代码和数据空间。其余的通信空间可以被划分为几个不相交的区域。通过控制这些区域的映射，几个用户进程可以有驻留在 NIC 上的通信端点，每个都有消息队列和相关的数据传输区（Chun，Mainwaring，and Culler 1998）。此外，主机的一组存储器帧可以被映射为 NIC 可以访问的 I/O 空间。因此，几个用户进程能具有向 NIC 写消息，从 NIC 直接读消息的能力，或者写入消息的描述符，描述符包含通过卡指向待 DMA 传输的数据的指针。这些通信端点可以像传统的虚拟存储器一样管理，这样写入一个端点，就使它驻留在 NIC 中。NIC 的固件负责把多个通信端点的消息多路复用到实际的网络链路上。它发现何时一个消息被写入发送队列时，把用户的目的地址翻译成经由网络到达目的节点的路径以及那个节点上目的端点的标识符，形成一个数据包。此外，它可以把一个源标识符放在包的头部，该源标识符在目的地检查。NIC 固件检查每个到来的消息的头部。如

517

果它的目的地是一个驻留的端点，那么，把消息直接存入相关的接收缓冲区；对于块传输，如果目的区域匹配，可以通过 DMA 把数据传入主机存储器。如果上述条件不满足，或消息被破坏，或违犯了保护检查，就给源端发送 NACK。通知管理端点映射和数据缓冲空间的驱动程序，在消息成功的重发之前修补环境。

7.7.2 案例分析：PCI 存储器通道

第二种重要有代表性的机群通信辅助部件设计是由 DEC 公司发展起来的存储器通道（Gillett 1996），该方案基于 Encore 的反射存储器和虚拟存储器映射的通信方面的研究成果（Blumrich et al. 1994；Dubnicki et al. 1996）。这个方法寻求提供有限形式的共享物理地址空间，而不是把整个伪存储器设备和伪处理器集成到该节点的存储器系统中，它还保留了机群的某些自治和独立失效的特征。和其他机群一样，通信辅助部件包含在插在常规节点扩充总线上的网络接口卡之中，在这里是 PCI 总线。

反射存储器背后的基本思想是在两个进程的地址空间区，即发送区和接收区间建立一种连接，如图7-28所示。源端写入发送区的数据被反射到目的端的接收区。通常，一个进程集

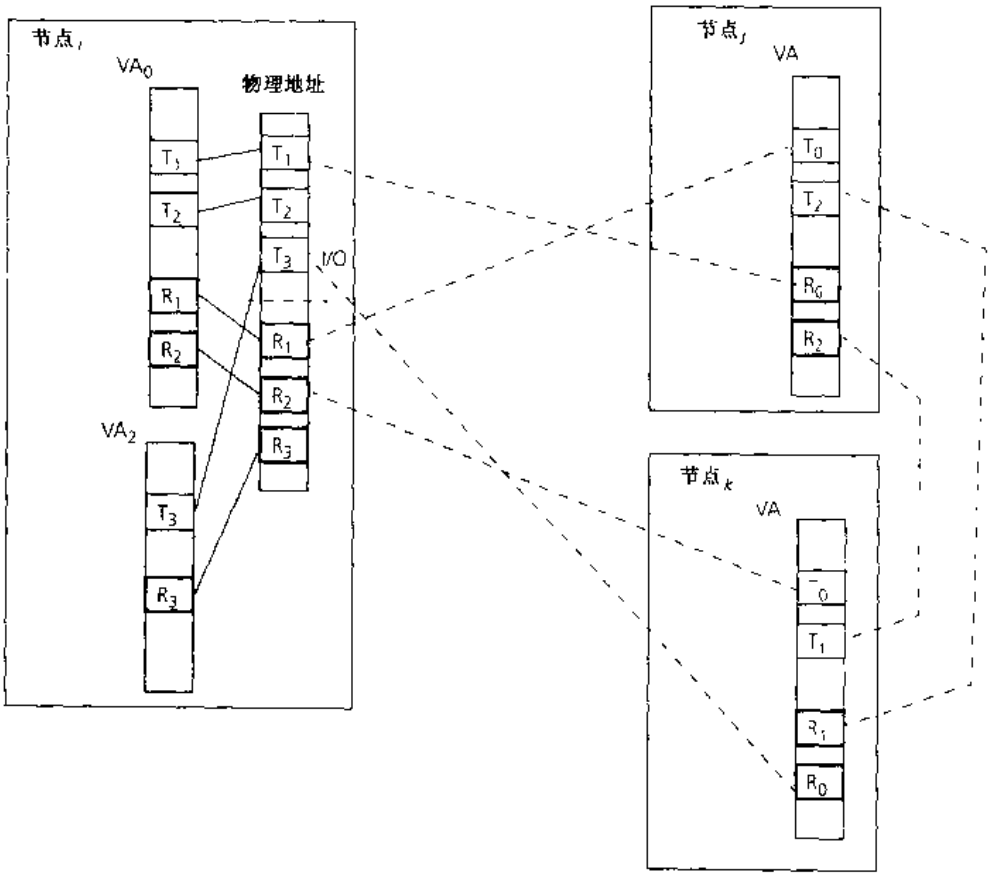


图 7-28 典型的反射地址空间组织结构。节点（VA）进程的虚拟地址空间中的发送区域被映射到与 NIC 相关的物理地址空间。接收区域设在存储器中，建立在 NIC 内的映射，这样就可以通过 DMA 把相关的数据传入主机存储器。这里，节点 i 有两个通信进程，建立了与其他两个节点上的进程的反射存储器连接。对发送区域的写入产生了对 NIC 的存储器事务。它接受写数据，建立一个数据包，其包头标示了接收的页面和偏移量，把数据包引导到目的节点。在从网络接收到一个数据包时，NIC 检查包头，并用 DMA 把数据传输到对应的接收页面。也可以选择让到达的数据包页产生一个中断。一般来说，用户进程扫描接收区域做相应更新。为了支持消息传递，接受页面包含消息队列

合会有一个全连接的发送-接收区对的集合。一个节点的发送区从映射到 NIC 的那部分物理地址空间中分配，而接收区则通过一个特殊的内核调用锁定在存储器中，NIC 经过配置，可以用 DMA 的方式把数据传给接收区。此外，源和目的进程必须在源发送区和目的接收区之间建立一个连接。典型的做法是给连接附加一个关键字，并把各个区域绑定到那个关键字中。区域包含整数个页面。

518
519

DEC 存储器通道是一个基于 PCI 的 NIC，典型地用于基于 Alpha 的 SMP 系统，如图 7-29 所示（Gillet 1996）。除了一般的发送和接收 FIFO 之外，它还包含了一张页控制表（PCT）、一个接收 DMA 引擎和发送与接收控制器。用一系列存储指令把一块数据写入发送区中。Alpha 的写缓冲器会力图合并对一个高速缓存块的更新，所以，发送控制器将典型地看到一个高速缓存块的写操作。给控制器的地址的高位部分是帧号，用来访问 PCT，以获取相关接收区的一个描述符（即目的节点或通往节点的路由、接收帧号、相关的控制位）。这些信息和源信息一起放入数据包头，通过网络发送到目标节点。接收控制器提取接收帧号，以此作为PCT索引地址。在检查数据包的完整性和验证源端之后，数据经过DMA传入存储器。接收

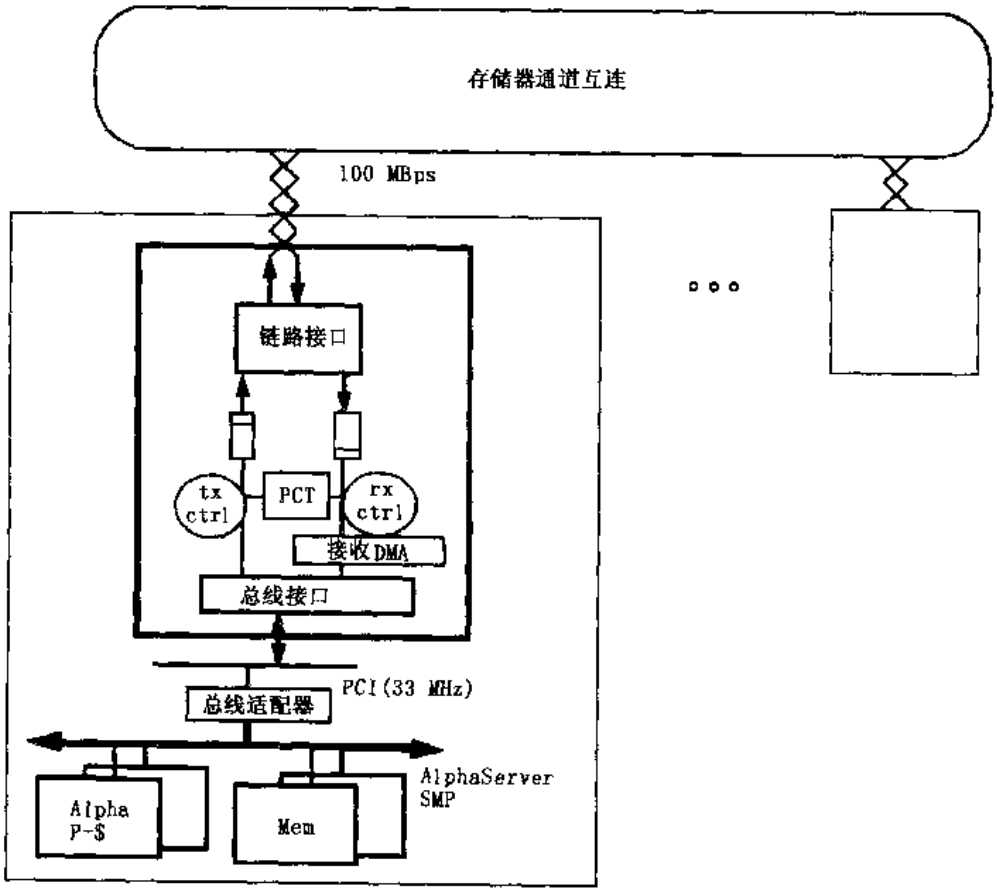


图 7-29　DEC 存储通道的硬件组织结构。一个存储器通道机群由一组带有 PCI 存储器通道适配器的 AlphaServer SMP 组成。适配器包含一个页控制表，用于完成本地发送区到远程接收区和本地接收区到锁定物理页面帧的映射。发送控制器（tx ctrl）接受 PCI 的写事务，用存储地址和页控制表（PCT）表项的内容来构造数据包头，把含有待写数据的数据包存放到发送 FIFO。接收控制器（rx ctrl）在启动一次把数据包数据传到主机的 DMA 传输之前，先检查 CRC 并分析收到的数据包的头。必要时，还可产生中断。在最初的机型，存储器通道的互连是 100 MBps 的共享总线，但它很有可能被交换所取代。每个典型节点都是一个相当大的 SMP AlphaServer

区可以是高速缓存的主机存储器，因为跨越存储器总线的传输是缓存一致的。如果需要，写完成之后发一个中断。

和共享物理地址空间一样，该方案允许数据在节点之间传输，即简单地存储源数据，从目的端能读出它。但是，地址空间的使用很受限制，因为传输到特定节点的数据必须放在特定的发送页和接收页中。这不同于任一进程都可读写任一共享地址空间的方案。典型情况是，共享的数据结构并不是放在通信区中的。相反，该区域为消息缓冲器专用。数据从逻辑共享的数据结构中读出，通过逻辑的存储通道发送给请求进程。因此，通信抽象成为真正的基于存储器的消息传递。没有读取远程地址空间的机制，进程只能读取已被写入的数据。为了更好地支持共享存储的“单一写，任意读”，DEC 存储器通道允许发送区向一组接收区多点广播，包括源节点上的回送区。为了方便构造分布式数据结构，特别是分布式锁存管理器，发送区操作间的先进先出顺序被保存起来。因为反射存储器不是简单地扩展了节点存储系统，而是建立在它之上，因此，一旦 NIC 接收到了数据，对发送区的写操作就结束了。为了确定写操作的实际发生，主机会检查 NIC 中的状态寄存器。

最初的存储器通道接口在两台 300 MHz 的 DEC Alpha 服务器之间达到了 2.9 μs 的单向通信时延和 64 MBps 的传输带宽（Lawton et al. 1996）。存储器通道上传输一条小 MPI 消息的单向时延约为 7 μs，长消息传输能达到的最大带宽为 61 MBps。用 TruCluster 存储器通道软件（Cardoza，Glover，and Snaman 1996）时，获取和释放一个非竞争的自旋锁分别需要大约 130 μs和 120 μs。

普林斯顿的 SHRIMP 设计方案（Blumrich et al. 1994，Dubnicki et al. 1996）扩展了反射存储器模型，以更好地支持消息传递。它允许由目的端的寄存器来决定接收偏移地址，这样，连续的数据包便可排队进入接收区。此外，可以对一个发送区执行一组写，发送区的一个区段可发送到接收区。

520 ~ 521

7.8 并行软件涉及的问题

至此，我们已经看到现代可扩展分布存储器型机器的设计频谱，根据交付给应用的通信性能，可巩固我们对这些设计折中的影响的理解。本节中，我们将通过三个不同层次的微基准测试程序来检验通信性能。第一组微基准测试程序采用了一种类似于基于用户到用户的基本网络事务的通信抽象；第二组用的是共享地址空间；第三组则采用标准的 MPI 消息传递抽象。在做出比较时，我们可以发现不同的组织方式和用于实现通信抽象的协议的不同效果。

7.8.1 网络事务的性能

很多因素相互影响，共同决定着单个网络事务端到端的时延以及建立在它之上的通信抽象。测量每个通信操作的时间时，我们观察到这些相互作用的累积效果。一般而言，测量到的时间会比通过把每一个单个硬件部件的时间加到一起得到的总和大。作为体系结构设计者，我们想知道每一个部件对性能的影响度；但是，程序关心的是这个累积效果，包括那些不可避免地使之变慢的细微的相互作用。

我们将用一个经验方案来判断几台我们作为案例分析目的机器的通信性能。使用一个简单的用户级通信抽象——主动消息，作为研究的基础。我们不仅想测量总的消息时间，还想知道数据传输方程（式（1-5））中的额外开销时间部分，还有占用和延迟的时间部分。

这个用主动消息的微基准测试程序是一个简单的回送检测：远程处理器提供网络服务并发回应答。这可排除当一个请求到达时处理器正在处理其他问题而引起的时间偏差。此外，既然只关心节点到网络的接口，我们只挑选在物理网络上相邻的处理器。由于这些机器之间没有一个全局的同步时钟，处理器之间的“时钟扭斜”会轻易地超过单个消息的传递时间尺度，因此，所有的测量都在源处理器上执行。请求-响应事务的往返时间除以2，获得端到端的消息传递时间。但是，这个单向消息传递时间可分为三个不同的阶段，如图7-30所示。处理器注入一个消息时，和通信辅助部件进行接口占用了不少时钟周期。我们称之为发送额外开销，因为这些时间不能用于有用的计算。类似地，目的处理器用了几个时钟周期去取出或处理消息，称之为接收额外开销。除额外开销外的消息传递时间称为通信时延。按照第3章中的通信时间表达式，它包含了传输时延和不能与发送、接收的额外开销重叠的占用时间。它很可能被其他的有效工作或其他消息的处理所隐蔽，我们将在第11章进行详尽的讨论。 522

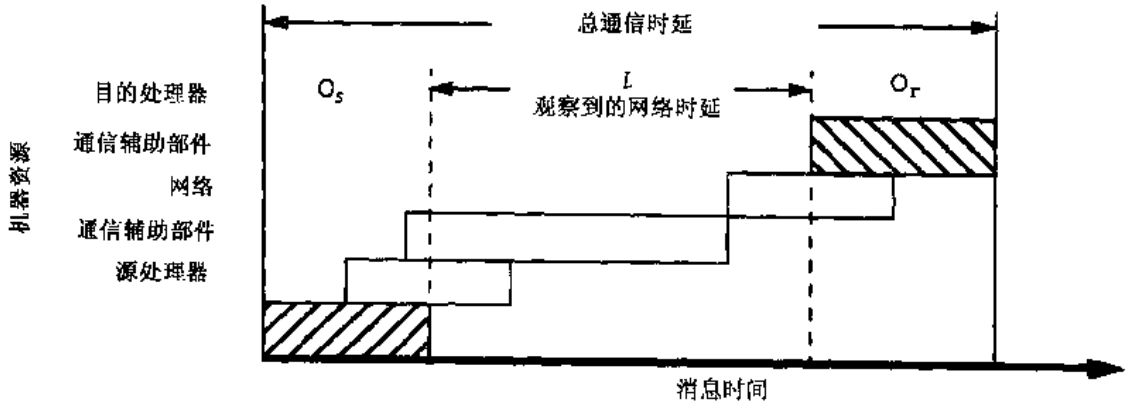


图7-30 消息传递时间分为发送额外开销、网络时延、接收额外开销。图中描述的是与我们的通信操作基本数据传输模型相关的机器操作。源端处理器用了 O_s 把消息交给通信辅助部件，这段时间内，处理器无法做其他有用的工作；类似地，目的端用了 O_r 的额外开销来取出消息；实际的信息传输涉及通信辅助部件、网络接口以及网络链路和交换机。从处理器角度看，这些子成分是不可区分的。处理器经历了不能由它自己的时延开销所覆盖的那部分传输时间，这段时间内处理器可以完成其他有用的工作。处理器也经历了最大消息传输率，这是根据消息间的最小平均时间间隔（或称为间隙）说明的

在我们的微基准测试程序中，我们期望能够确定这三部分时间长度。但是处理器无法区分出通信辅助部件花费的时间和实际互连链路与交换机所消耗的时间。事实上，当处理器还在检查状态字的时候，通信辅助部件可能已经开始把消息推送到网络上了，但是这个工作被额外开销隐蔽了。

图7-31左边的图显示了比较单向的主动消息在以下几种不同的系统上的传输时间：Thinking Machines 的 CM-5（7.4.2节）、Intel 的 Paragon（7.5.1节）、Meiko CS-2（7.5.2节）、NOW Ultra 的工作站群（7.7.1节）、CRAY T3D（7.6.1节）。直方图显示出单向传输的一条小消息（5个字）的传输时延的三个部分，包括发送端的处理开销（O_s）、接收端的处理开销（O_r）、剩下的通信时延（L）。右边的直方图（g）显示出请求-响应操作流水序列中每个消息的时间。例如，在 Paragon 机上，单个消息端到端的时延大约为10 μs，但是，消息阵发传输时，每个消息的时延大约为7.5 μs，或者说其传输速率为每秒1/7.5 μs = 133 000个消息。让我们更详细地考察每一个成分。

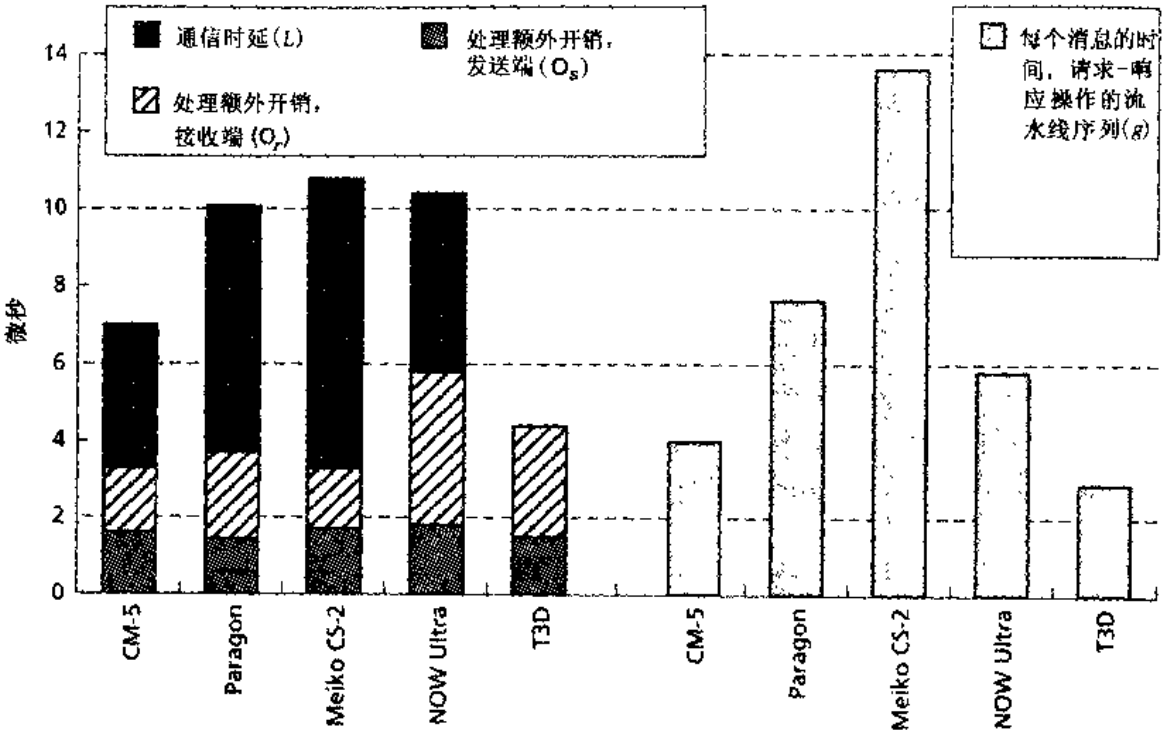


图 7-31 使用主动消息的网络事务级的性能比较。左边的直方图示意了本章描述的 5 种机器各自的总的时延，分成发送额外开销、接收额外时延和网络时延三个部分。其中后者主要由通信辅助部件占用时间所主导，但包括网络延迟。右边的直方图显示了这些机器传输一连串的小消息时每个消息的时间，叫做间隙。它主要由通信辅助部件的占用时间所决定

523 5 种设计的发送额外开销基本一致，但是，各个系统对之的决定因素却不尽相同。在 CM-5 系统上，发送额外开销由数据非高速缓存写操作和网络接口状态的非高速缓存读操作决定，读状态可判定消息是否被接受。此外，状态还可指示进入的消息是否到达，这样的安排是方便的，因为即使节点无法发送消息，它也必须接收消息。不幸的是，这两个网络要求两个状态读操作（较晚的机器型号 CM-500 则提供了更为有效的状态信息，结果大大地降低了发送开销）。Paragon 系统中，发送额外开销由把消息写入计算处理器和通信（或消息）处理器（CP）所共享的缓冲区的时间所决定。计算处理器能够用两条 4 字的存储指令写入消息，同时在缓冲项中设置“消息存在”的标记，这是 i860 所独有的，因此，它的发送开销也惊人地高！其原因是对于我们前面讨论过的生产者-消费者情况，节点内基于总线的高速
524 缓存一致性协议的低效率。计算处理器必须在重新填充之前从通信处理器那夺走缓存块。在 Meiko CS-2 中，消息在结合高速缓存的存储器中建立，然后，用一条交换指令把指向消息（和命令）的指针排入 NI 的队列。Spare 指令系统中提供了这条指令，支持同步操作。在这里，它提供了一个把消息传递给 NI 并获取标记该操作成功与否的状态字的方法。不幸的是，交换操作比基本的非高速缓存的存储器操作慢了很多。在 NOW 系统中，发送额外开销由一系列双字的非高速缓存的存储操作和一个跨越 I/O 总线到 NI 存储器的非高速缓存的读操作所决定。令人吃惊的是，这和更为紧密集成的设计有着相同的实际代价。

通过对非高速缓存的操作、扑空和同步指令的时间开销的比较，我们发现了一件很重要的事情：这些体系机构优化时很少顾及的，一般被认为是不常发生的事件，其实才是影响通信性能的关键。允许事务提交给存储层次结构下一层次之前，高速缓存控制器所花费的时间

甚至在总线协议中占主导地位。比较各种设计方案中的接收额外开销，我们发现从网络处理器到计算处理器的高速缓存至高速缓存传输，比从 CM-5 和 CS-2 存储器总线上的网络接口的非高速缓存读出操作的代价更高。但是，在 NOW 系统中，I/O 总线上的非高速缓存的读操作有更高的时间开销。

机器的几个方面都会影响时延成分，包括通信辅助部件或网络接口中的处理时间、通过通道把消息发到链路上的时间和网络传输的延迟。我们研究的系统其各自的决定因素不同。CM-5 链路上的传输速率为 20 MBps（带宽为 4 位，频率为 40 MHz）。这样，单线传输一条 40 个字节外加一个含有路由信息、循环冗余码和消息类型的数据包头封装的时延大约为 2.5 μs。每个路由器增加一个大约 200 ns 的延迟，机器最多有 $2\log_4 N$ 跳。网络接口的占用和线路的占用基本相同，因为它只是一个从线路上取走或向线路上放入数据包的简单设备。

在 Paragon 系统中，时延由源端和目的端的通信处理器的处理过程所决定。在 175 MBps 的链路上，链路的占用仅为大约 300 ns。每跳的路由延迟也很小，但是，大型系统上总的延迟可能比链路占用要大的多，因为跳数可达 $2\sqrt{N}$。其主导因素是在源端经由存储器总线把消息写入网络接口和从目的端读取消息的过程对通信辅助部件（CP）的占用。这些步骤花了时延中的 4 μs。因此，取消通信路径上的通信处理器可以明显地减少时延（Krishnamurthy et al. 1996）。让人吃惊的是，这样做并不会增加多少发送和接收额外开销，对网络接口写入或读出消息本质上和高速缓存到高速缓存传输的时间开销一样。但是，由于网络接口并未提供充足的解释来强化保护以及保证消息经由网络向前传递，所以取消通信处理器在实践中是不可行的。

Meiko 系统中有一个非常大的时延成分。网络提供了 40 MBps 的链路和类似于 CM-5 的拓扑结构，因此它只占其中的很小一部分。通信处理器和网络紧密耦合在一块芯片中。时延大部分由通信处理器访问系统存储器引起。回想一下前面讲到的计算处理器为通信处理器提供消息指针。通信处理器执行 DMA 操作从存储器中取出消息。目的端则把消息写入一个共享队列。这个系统的一个不寻常的特点是：网络上的电路在进行网络传输时是保持的，目的端为源端提供一个确认信息。这种机制用来通知源通信处理器它是否成功地在目的端接收消息队列中获得了一把锁。这样，尽管这部分时延很长，但难以通过流水的通信操作加以隐藏，因为在通信期间，源端和目的端的通信处理器被占用。

525

NOW 系统中，通信系统各部分的时延分布相当均匀。其链路的传输速率为 160 MBps，使得链路占用度比较小，路由延迟也很普通，每跳时延为 350 ns。数据由主机送入网络接口，并直接从网络接口中读出。时间主要耗在传输事务两端进行的消息队列处理和网络接口存储器与网络之间的 DMA 传输操作上。这样，两个网络接口与网络各自大约花费了 4 μs 时延的三分之一。

CRAY T3D 以每个处理器分配一个消息队列的形式为用户级消息提供硬件支持。它采用了 DEC Alpha 的用来扩展指令系统的特权子例程（PAL 代码）的技术。一个 4 字的消息在寄存器中形成，然后发出一个 PAL 调用来发送消息。消息在目的端放入处理器的用户级消息队列中；目的端处理器被中断，控制权返回给查询该消息队列的用户应用线程或一个特殊的消息处理线程。插入消息的发送额外开销仅为 0.8 μs；但是，中断的额外开销为 25 μs，切换到消息处理例程的额外开销为 33 μs（Arpaci et al. 1995）。使用为原子操作而提供的 fetch&increment 寄存器，可以不经中断和线程切换把数据包插入消息队列。推进队列指针的

fetch&increment 和消息的写操作用了 1.5 μs，而根据消息分支和非高速缓存的数据读操作花了 2.9 μs。

在 Paragon、Meiko 和 NOW 机器中，每个节点上都有一个完整的操作系统。节点之间使用消息协同，提供一个单一系统映像、在节点集合上运行并行程序。通信处理器允许来自多用户进程的消息多路复共享的网络，同时把来自共享网络的消息分发到正确的目的进程中去。它还提供了流控和差错检验。MPP 系统依靠物理上单个机箱的完整性来提供高可靠性的操作。因此，只要其中有一个用户进程失败，并行程序的其他进程也被迫中止。当一个节点瘫痪，它所在那部分机器要重启。更为松散集成的 NOW 系统则必须应付由硬件出错、软件出错、物理断开引起的单个节点的失效。和 MPP 系统一样，操作系统之间协同控制并行程序的进程。一个节点失效时，系统对其余节点重新配置，甩掉故障节点继续工作。

[526]

7.8.2 共享地址空间操作

比较案例分析中这几种机器的通信体系结构对共享地址空间的读与操作很有用。这些操作可以很容易地在用户级消息抽象之上实现，但是这样就放弃了对这些简单而又常用的操作进行优化的机会。另外，在通信辅助部件没有提供足够的解释能力的系统中，当远端存储器被访问时远端处理器会被占用。在具有专用通信处理器的系统上，通信处理器可以潜在地服务于节点的存储器请求，避免计算处理器被占用。在支持共享物理地址的机器上，硬件直接支持这种存储器请求的服务。

图 7-32 显示了这几种系统远端地址单元的读操作性能（Krishnamurthy et al. 1996）。左边的直方图给出总的读操作时间，分为三个部分：发出读指令的额外开销、通信时延和远程处

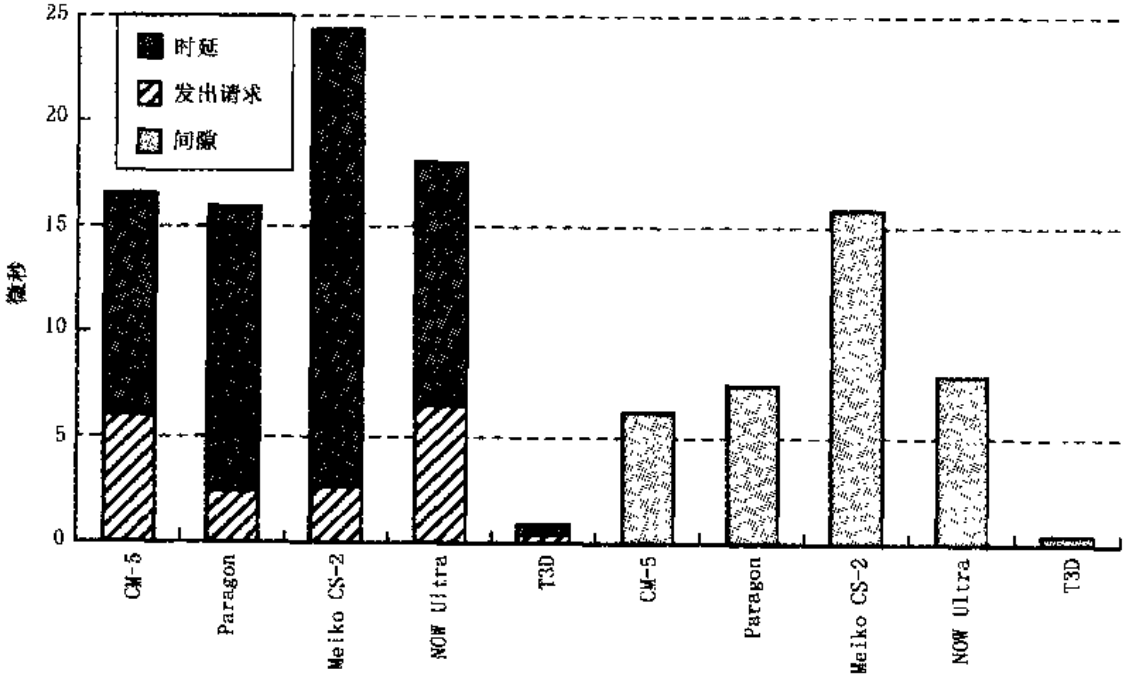


图 7-32 共享地址读操作的性能比较。对这 5 种案例分析平台，左边的直方图表明了执行远程读操作的总的时间，该时间被分成发出请求时间和完成读的时间。其余时间与其他有效任务时间相重叠，第 11 章中将作深入的讨论。右边的直方图表示的是连续的两个读操作之间的最小时间间隔，它是最大传输速率的倒数

理时间。右边的直方图显示了在稳定状态下，两个读操作之间的最小时间间隔。对 CM-5 系统而言，由于远程处理器必须处理网络事务，因此无法对其进行优化。在 Paragon 系统中，远程通信处理器处理读请求并应答。其通信处理器要从网络接口读取消息，处理它，然后向网络接口写一个应答，因此其远端处理耗时非常显著。此外，通信处理器还要进行保护检查和虚拟地址到物理地址的转换。在运行 OSF1/AD 操作系统的 Paragon 系统上，通信处理器在核心态下运行，对物理地址进行操作，因此，它用软件对请求的地址执行页表查找（请求的地址未必是当前运行进程的）。Meiko CS-2 系统提供读和写网络事务，其源端到目的端的电路保持连接，直到读响应或写确认返回。专用的远程处理用一个硬件页表来完成远端节点上的虚拟地址到物理地址的转换。这样，读时延比一对消息的确小了不少但仍相当显著。如果远端通信处理器执行读操作，时延还要增加 8 μs。NOW 系统有一个很小的性能优势，因为它没有使用远端处理器。如在 Paragon 系统中，消息处理器必须进行保护检查和地址转换，这在嵌入式通信处理器上是非常慢的。此外，从网络接口访问远端存储器涉及 DMA 操作，代价很高。在所有这些系统中，远程存储器操作的专用处理的主要优点是，操作的性能不会依赖于对进入的请求进行服务的远端计算处理器。 527

从图中可看出，T3D 系统通过提供专用的硬件来支持共享地址空间，结果使其性能提高了一个数量级！把远程读写操作交给附加的硬件实现，发出操作的时间开销仅为 400 ns。另外，请求的传送、远程服务和应答传送另外花费了 450 ns。对一连串的读操作，用硬件预取队列可进一步提高其性能。发出预取指令和从队列弹出只需大约 200 ns。时延被一系列预取分摊，当有 8 个或更多预取操作时，时延几乎完全被隐藏。

7.8.3 消息传递操作

让我们看看几种大型机上消息传递的性能测量结果。我们前面已经讨论了，消息传递操作的最常用的性能模型就是用来计算发送 n 个字节的总时间的线性模型。其线性关系如下：

$$T(n) = T_0 + \frac{n}{B} \tag{7-2}$$

启动开销 T_0 逻辑上是发送 0 字节的时间，B 是渐进带宽。数据传输速率可简单地计算为 $BW(n) = n/T(n)$。同样地，传输时间用两个参数来描述：r_∞ 和 $n_{\frac{1}{2}}$，分别表示渐进带宽和得到一半带宽传输尺寸（即半能力点）。

启动开销反映协议的执行以及建立数据传输所要求的任何缓冲管理和匹配。渐进带宽反映了系统上端到端的数据注入速率。 528

尽管这个模型易于理解，而且作为编程指导很有用，但是它却提出了体系结构评价中的几个难题。与网络事务一样，除非有一个全局时钟，否则总的消息时间很难测量，因为发送在一个处理器执行，而接收在另一个处理器进行。这个问题通常用测量一个应答测试的时间来解决，一个处理器发送数据，然后等待，直到接收到一个消息。但是，只有在消息到达之前远端就发出接收，该方案才能产生一个可靠的测量，否则，测到的其实是远端节点在向发出接收前消耗的时间。如果接收是预先发出的，那检测到的就不是完整的接收时间，无法反映出由于匹配成功而缓冲数据的开销。最后，测量的时间也不是与消息的大小成线性关系，因此为了使数据和直线拟合，产生了一个和实际的启动开销没多大关系的参数 T_0。一般来

说，n 的值比较小的时候会有一段平直区间，以致从这种拟合中得到的启动开销比发送 0 个字节的消息的开销还小，甚至有可能为负数。这些与方法相关的问题在较老的机器上不存在，因为它们的启动开销很大，基于软件的消息传递实现也很简单。

表 7-1 列出了一段时期以来的几种重要的大型并行机商用消息传递库的启动开销和渐进带宽。（表中还另加了第 6 章讨论过的两个小型 SMP 系统的相关信息。）从表中可以看出，不到十年，启动开销 T_0 下降了一个数量级，它本质上与时钟周期的改进相符，这一点从表的中间那列可以很容易地看出。表的右面几列说明，启动开销的改进跟不上浮点运算性能和带宽的改进速度。注意，启动开销与只有几微秒或几分之一微秒的通信网络硬件时延不在一个数量级上。它由每个消息事务的处理开销、协议和用于提供消息传递抽象的同步语义所需要的处理决定。

表 7-1　消息传递的启动开销和渐进带宽

机器	年	T_0 (μs)	最大带宽 (MBps)	每个 T_0 的周期数	每个处理器的 MFLOPS	每个 T_0 的浮点操作	$n_{1/2}$
iPSC/2	1988	700	2.7	5 600	1	700	1 400
nCUBE/2	1990	160	2	3 000	2	300	300
iPSC/860	1991	160	2	6 400	40	6 400	320
CM-5	1992	95	8	3 135	20	1 900	760
SP-1	1993	50	25	2 500	100	5 000	1 250
Meiko CS-2*	1993	83	43	7 470	24	1 992	3 560
Paragon	1994	30	175	1 500	50	1 500	7 240
SP-2	1994	35	40	3 960	200	7 000	2 400
CRAY T3D（PVM）	1994	21	27	3 150	94	1 974	1 502
NOW	1996	16	38	2 672	180	2 880	4 200
SGI Power-Challenge	1995	10	64	900	308	3 080	800
Sun E6000	1996	11	160	1 760	180	1 980	2 100

图 7-33 显示在几种机器上作为消息尺寸函数、通过应答测试测得的单向通信时间。在这个区间段图中，启动开销的差别是明显的，即小消息对应的非线性段。带宽由线的斜率给出。带宽可以从与之等价的 BW（n）的对应图示（如图 7-34 所示）中看得更清楚。在解释消息传递的带宽数据时必须警惕，因为成对传输带宽只反映了点到点的数据率。例如，我们知道，对基于总线的机器而言，如果多对节点之间同时进行通信，一般是不会保持这个带宽的。在第 10 章中我们将看到，有的网络可以提供比其他网络更高的集合带宽。特别是，Paragon 的数据对于涉及多对节点的许多通信模式而言是过于乐观的，而其他的大型系统能对大多数的模式保持成对通信带宽。

7.8.4　应用层性能

计算机系统各方面的端到端效应一起决定了应用层获得的性能。这一层的分析对终端用户最有用，通常是采购决策的基础。它也是评价体系结构折中方案的最终基点，但是，这就要求找出机器和程序中那些因素导致了累积的性能效应。一般地，这种映射涉及对应用的剖

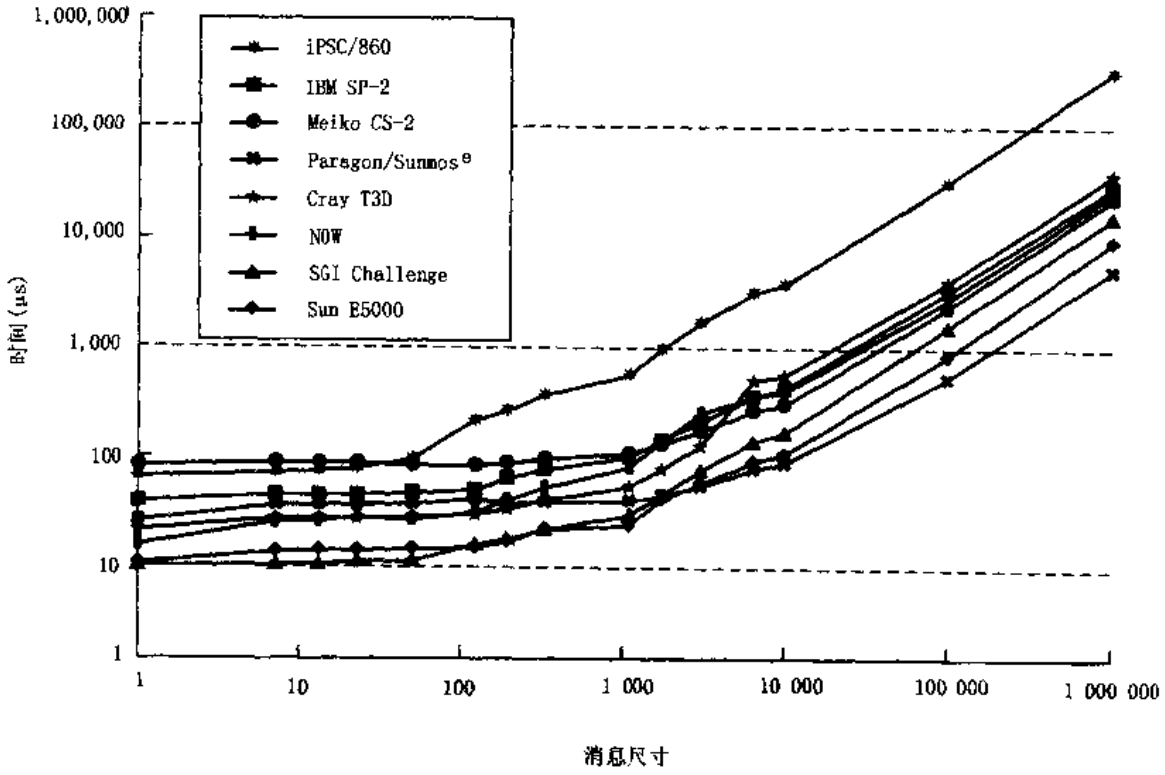


图 7-33 消息传递操作的时间与消息尺寸的关系。不同的消息传递的实现显示了几乎一个数量级的启动开销的差别，对于一个范围之内的小尺寸消息，这个开销是常数，导致单个消息时间曲线的显著非线性。在大尺寸消息区段，时间与消息尺寸成近似的线性关系，曲线的斜率表示带宽

⊖ 在这个基准测试中用的是 Sunmos 操作系统

析，分离出主要时间开销之所在，提取出该应用的使用特征以决定其需求。本节简要地比较在我们研究的几种机型上运行 NPB2 测试程序集（NAS 基准测试程序 1998）的两个 NAS 并行基准程序所得到的性能。

LU 基准测试程序以差分方法的上三角块和下三角块近似分解，解决用于流体动力学模型的三维可压缩 Navier-Stokes 方程的有限差分离散化问题。LU 因子分解形式作为放松因子，用对称连续超松弛（SSOR）方案来解。该基准测试程序基于使用 Intel 的消息传递库的 1991 年 NAS 参考实现，NX（Barszcz et al. 1993）。它要求有 2 的幂次方个处理器。把坐标的前两维重复对分得到三维数据网格的二维分区，然后按 x、y 坐标交替分配到处理器，直到所有的处理器都被分配，结果形成垂直笔状的网格分区。每支笔可看作水平片的堆砌。基于点的操作的次序构成 SSOR 沿对角线上的处理过程，从一给定的 z 平面的一角向其对角的相同 z 平面扫描，接着处理下一个 z 平面。这样构造的对角线流水方法被发明者称为“波前”法（Barszcz et al. 1993）。软件流水线用了相对很少的时间来装满和清空，得到很好的平衡。完成了所有与相邻分区发生作用的对角线上的计算后，发生分区边界的数据通信。结果导致了相对较多的小尺寸通信。但是，总的通信量比计算开销还是要小，使得这种并行 LU 方案效率相对更高。松弛段内高速缓存块复用率很高。

图7-35显示了对于稀疏LU在A类输入（在64 × 64 × 64网格上的200次迭代）时，IBM SP-2（宽节点）、CRAY T3D 和 UltraSparc 机群（NOW）各自的加速比。加速比以 4 个处理器的性能为基点进行了标准化，图 7-35 的右边显示了 4 个处理器时的各自的性能。之所以选

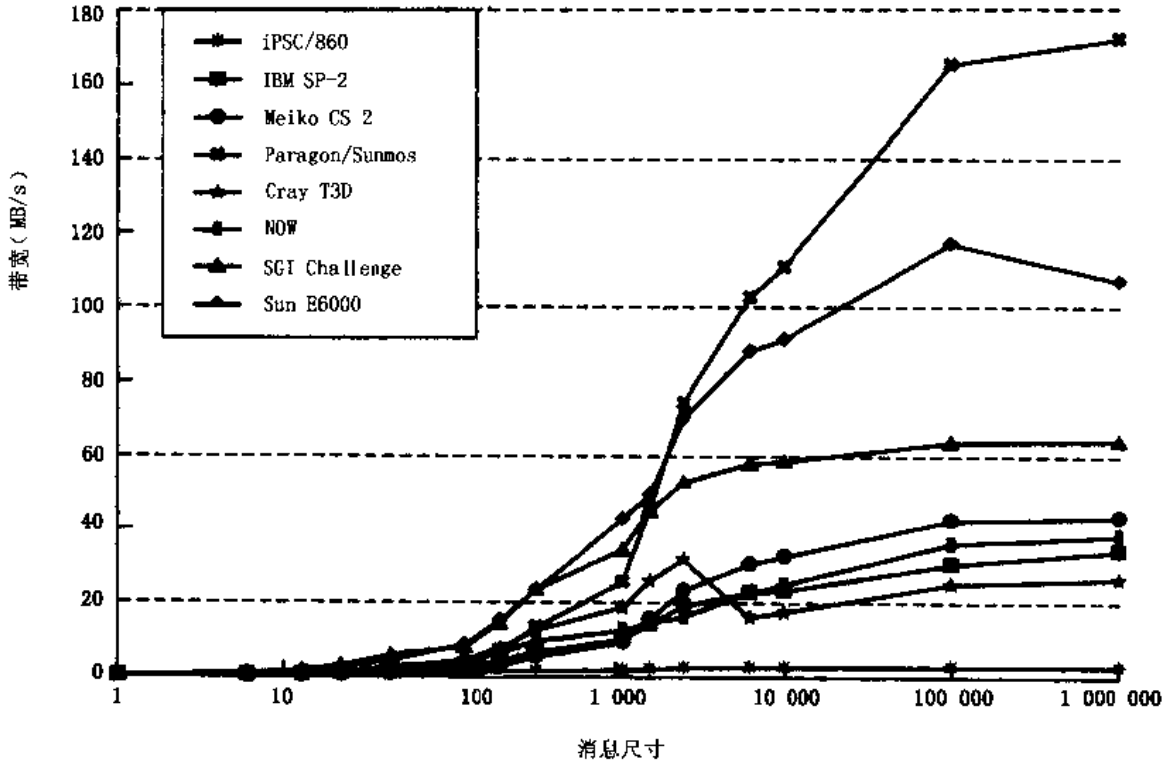


图 7-34 带宽与消息尺寸的关系。可扩展型机器在大消息传输时，通常提供每个节点每秒几十兆字节的带宽，虽然 Paragon 设计使其带宽达到了每秒几百兆字节。带宽主要受限于存储器系统，包括内部总线，传送数据的 DMA 支持的能力。小型的共享存储器设计在几个传送同时发生时，呈现了出自高速缓存的存储器副本所要求的点到点带宽

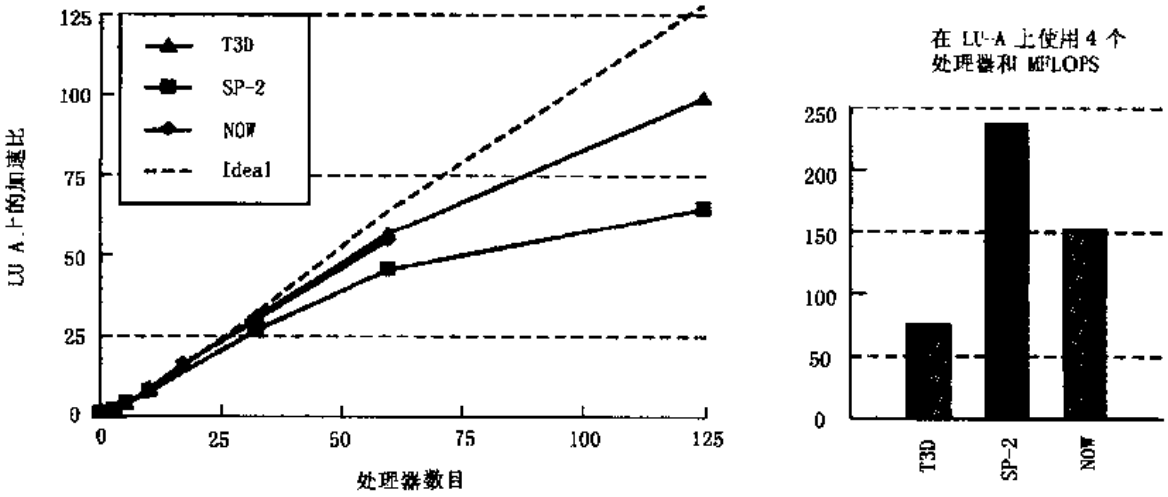


图 7-35 在稀疏 LU NAS 并行基准测试程序上的应用性能。左图显示了 IBM SP-2、CRAY T3D 和 NOW UltraSparc 机群运行在稀疏 LU 基准测试程序的可扩展性，以 4 个处理器的性能为基点进行标准化。右边表示 3 个系统的 4 个处理器时的基础性能

择 4 个处理器为基点，是因为这是 T3D 解决该问题的最小节点数。从图 7-35 中我们可以看出，处理器个数超过 100 时扩展性还很好，而且 T3D 和 NOW 两者的扩展性都比 SP-2 要好。这和处理器的性能与小消息传输性能之比相一致。

BT算法解三组不相关的方程组——首先 x 方向，然后 y 方向，最后 z 方向。这些方程组是块对角方程，块大小是 5×5，用多分区方案可解出（Bruno and Cappello 1988）。多分区方案提供了很好的负荷平衡，使用粗粒度通信。每个处理器负责几个网格点（单元）的不相连字块。这些单元的分配使得在按行求解阶段的每个方向上，属于某个特定处理器的单元将沿解的该方向均匀分布。这样，每个处理器在按行求解整个过程当中一直做有用的工作，而不是在开始工作前被迫等待来自其他处理器的行的部分解。此外，一个单元的信息直到它所处理的线性方程组各个部分都被解出之后才传给下一个处理器。所以，保持了一个大的通信的粒度，使得传送的消息更少。BT代码要求平方数个处理器。代码已经写入，因此如果一个给定并行平台只允许给一个作业分配2的幂次方个处理器，那些不需要的处理器被认为是闲置的，在进行运算时被忽略，但计算MFLOPS速率时却要把它们计算在内。 529 ~ 532

图7-36显示了IBM SP-2、CRAY T3D和UltraSparc NOW系统在A类问题上BT基准测试程序的可扩展性。这里的加速比以25个处理器的性能为基点进行标准化。图7-36右部所示即为25个处理器的性能，之所以这样标准化，是因为这是能获得性能数据的最小T3D配置。这三种平台的可扩展性都不错，尽管SP-2还是相对差一点，但它单个节点的性能却高了很多。

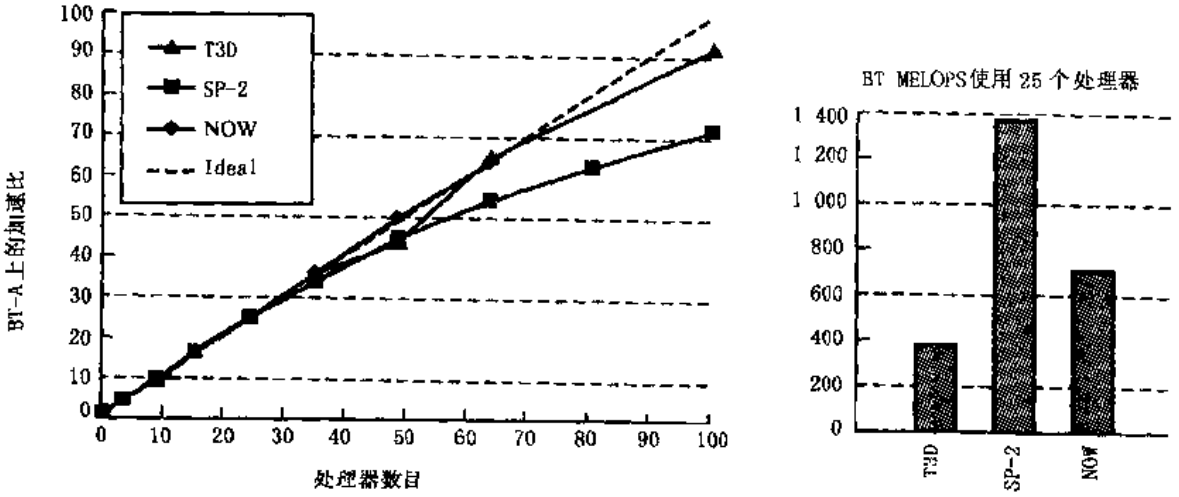


图7-36 在BT NAS并行基准测试程序集上的应用性能。此图表现的是IBM SP-2、CRAY T3D、UltraSparc机群在BT基准测试集上的可扩展性。这3个系统的基本性能如右图所示

为了理解为什么机器的扩展性总是低于理想情况，我们需要更进一步地研究应用的特征，尤其是通信特征。为了把注意力集中到应用的制约因素部分，我们研究最外层循环的一个迭代。典型地，应用在初始化之后都会执行几百个这种迭代。这次，我们不使用前面几章的模拟方法来考察通信特征，我们通过运行用经过修改的MPI消息传递库编写的程序，来收集这个信息。消息尺寸的直方图统计是一个有用的特征。每发送一个消息，与其尺寸范围对应的计数器就加一。表7-2的上半部分对所有处理器的结果进行了总结，其中问题的规模固定，处理器的个数从4扩展到64。对每一种配置而言，给出了消息个数非零的消息尺寸，以及对应尺寸的消息个数和这些消息中传送数据的总量的估计值。我们看到，这个应用中基本上传送了三种不同尺寸的消息，但是这些尺寸和发送频率随处理器的数量变化很大，消息 533

尺寸和发送频率是消息传递程序的两个典型特征。调试好的程序高度结构化，通信使用也非常精简节约。此外，由于问题规模保持固定不变，处理器的数量增加时，每个处理器负责的问题部分就相应减少。因为传送数据的尺寸由程序如何编写所决定，而与机器参数无关，配置变化时，数据尺寸会发生显著变化。即在小型配置上程序发送几个非常大的消息，而在大型配置上它发送许多相对较小的消息。当处理器的个数增加 16 倍，总的通信量几乎增加 8 倍。这个增量当然是影响加速比曲线的一个因素。此外，启动开销较大的机器受消息尺寸减小的影响越严重。

表 7-2 一定数量范围的处理器上执行 BT 算法 A 类总题的一次迭代的通信特征

	4 处理器			16 处理器			36 处理器			64 处理器		
	消息尺寸（KB）	消息	总数据传输量（KB）	消息尺寸（KB）	消息	总数据传输量（KB）	消息尺寸（KB）	消息	总数据传输量（KB）	消息尺寸（KB）	消息	总数据传输量（KB）
	43.5+	12	513	11.5+	144	1 652	4.5+	540	2 505	3+	1 344	4 266
	81.5+	24	1 916	61+	96	5 742	29+	540	15 425	19+	1 344	25 266
	261+	12	3 062	69+	144	9 738	45+	216	9 545	35.5+	384	13 406
总通信量（KB）			5 491			17 132			27 475			42 938
一次迭代的时间（s）			5.43			1.46			0.67			0.38
平均带宽（MB/s）			1.0			11.5			40.0			110.3
每个处理器的平均带宽（MB/s）			0.25			0.72			1.11			1.72

观察在固定问题规模的可扩展规则下，每个处理器的工作量随扩展而变化非常重要。这里我们观察通信方面的情况，但是，它一样改变了高速缓存的行为和一些其他的因素。表 7-2的下半部分描述了该应用的平均通信需求。用总的消息量除以 UltraSparc 机群上每次迭代的时间[⊖]，得到可提供的平均消息数据带宽。的确，通信率确实按比例增长，机器配置每增加 16 倍，通信率就增加 100 倍以上。进一步除以处理器的个数，我们发现每个处理器的平均带宽很显著，但并不是非常大。与我们在 5.4 小节中模拟 SMP 时观察到的共享地址空间应用的速率差不多。但是，根据平均通信需求来决定设计方案必须极其谨慎，因为通信很可能有突发事件。通常，计算过程会不时地被突发的通信事件打断。对 BT 基准测试程序上的应用而言，通过定期的采样消息直方图的快照，可以得到瞬间通信的行为。图 7-37 显示了执行该程序的 64 个处理器中之一的几次迭代的结果。每一个采样间隔内，直方条表示该段时间间隔内所传输的最大消息的尺寸。对该应用而言，通信的剖视轮廓在所有的处理器上都相似，因为它对所有的处理器采用阶段同步方式，所有处理器在本地计算和通信阶段间交替切换。从图中可清楚地看到三种消息尺寸，它们以有规律的模式重复出现，两个较小的尺寸比较大的尺寸出现得更为频繁。总体来说，白的空间的确比暗的空间要多，因此，平均通信带宽与其说表明了通信期间的通信比率，不如说表明了通信对计算的比例。

如果把式（3-1）提供的框架应用于 NAS 并行基准测试程序的加速比检测，我们发现，随着处理器的数量的增加，所有的并行性代价都随处理器的数量而增加。增加了额外的工作，通信量增加了，通信的代价也增加了，等待时间也增加了。虽然如此，这些基准测试程

⊖ 执行时间是表中特定平台的惟一一个特征数据。所有的其他通信特性都是关于程序的，因此在任意的平台上测量到的结果都一样。

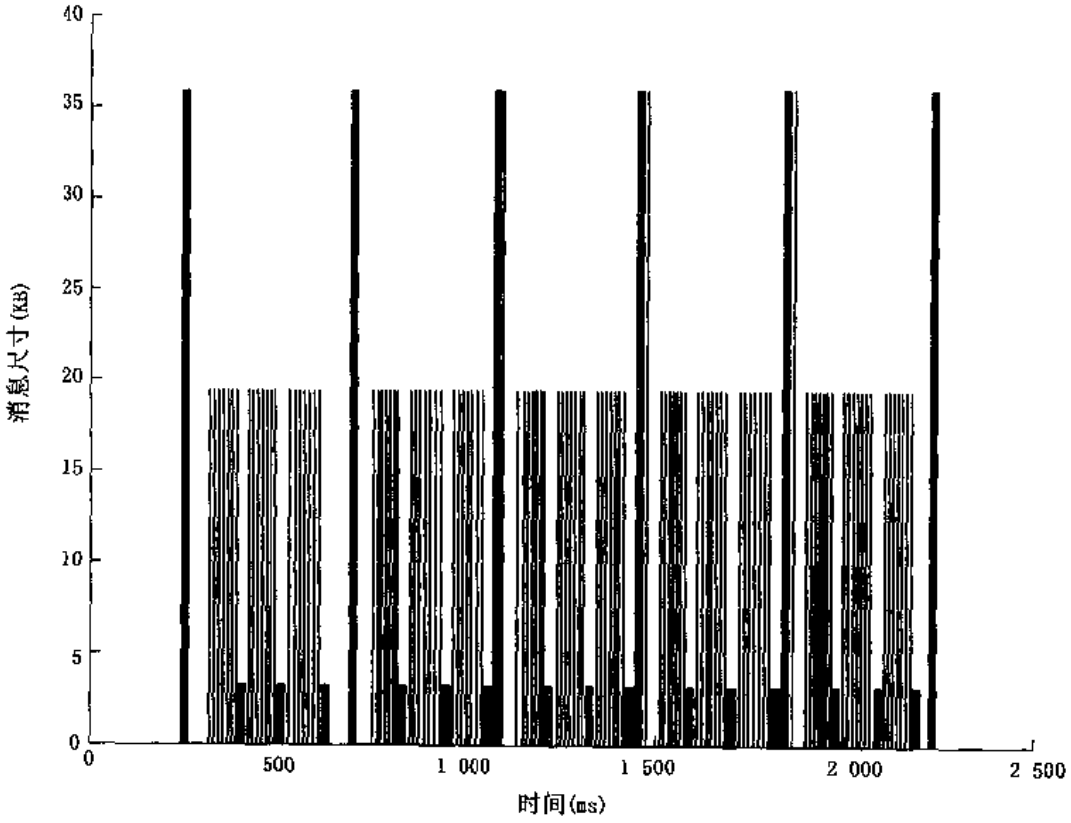


图 7-37 在 64 个处理器上 BT-A 运行中消息随时间变化的轮廓。该程序使用被修改过的 MPI 库执行，该 MPI 库能定期采样通信直方图。图中显示了每一个采样间隔内一个特定的处理器发出的最大的消息。很明显，通信是有规律的、突发的

序中的几个却在有相当大数量处理器时获得了完美的加速比。原因是，固定规模的问题分布到大量的处理器上时，计算工作效率更高。特别是，图 3-6 所示的工作集的行为有非常显著的影响，尽管程序用的是消息传递的编程模式。这个效应对LU的影响可以从图7-38中看出。

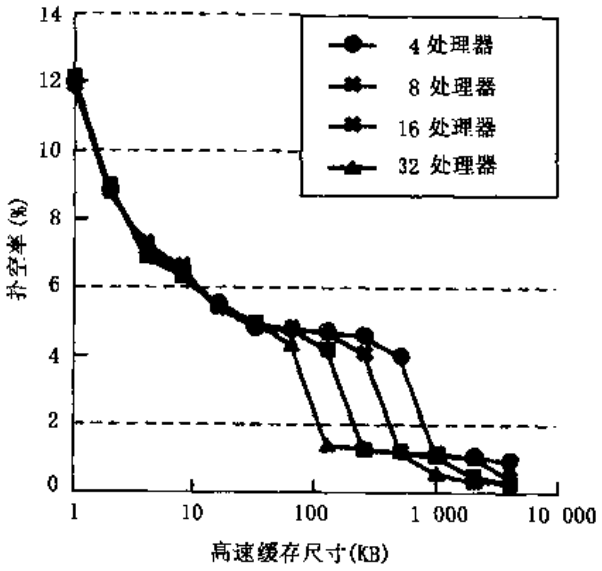


图 7-38 一定规模范围的机器执行 NAS 并行基准测试程序 LU 的工作集曲线。随着 CPS 的扩展，由于数据集分布到不同数量的节点上，使得不同规模的系统对应的曲线也不同。在大高速缓存情况下，对应曲线的分离通过减少计算花费的时间而改善观察到的加速比。但是，在 20 世纪 80 年代中期普通的小高速缓存配置上，这个效果没有出现，而恰恰出现了与之相反的情形

图中的每一条曲线表示了在一个执行该并行程序的典型节点上，作为高速缓存尺寸的函数的高速缓冲扑空率。这个结果的获得是通过并行地运行该程序，收集每个节点上高速缓冲轨迹，把该轨迹提交给一个高速缓存模拟器，该模拟器模拟了具有 64 字节块的各种不同大小的全相联缓冲器。观察的关键点是每种规模的系统都有不同的工作集剖视轮廓。当是 4 个处理器时，拐点出现在高速缓存大小为 512 KB 处，但是在 32 个处理器时，拐点出现在 64 KB 处。因此，在具有 256 KB 高速缓存的机器上，其配置由 4 个处理器扩展到 16 个或更多个处理器时，扑空率从 5%降到 1%。的确，随着配置的扩展，全部处理器的总计算时间下降了，因此，尽管通信花费的时间增加了，还是能获得完美的加速比。但这个效果在 SP-2 上却不突出，因为其基本节点对于不能为高速缓存所容纳的数据的操作进行了优化。

533 ~ 537

7.9 同步

在大型的分布存储器型机器上软硬件结合实现同步操作主要关心的问题是可扩展性。对于消息传递的编程模型，互斥是基本假定，因为每个进程对其局部地址空间有着独占性的访问。点到点事件隐含在每个消息操作中，更令人感兴趣的情况是根据点到点的消息来指挥全局的或组的同步。这里一个重要的问题是平衡：重要的是用于实现同步的通信模式在节点间必须平衡，以便实现高的消息速率和高效率的同步。在极端情况下，我们应该避免让所有的进程同时与同一个进程通信或等待同一个进程。消息传递层的机器设计者和实现者都力图使这种环境下的消息速率最大化，但是只有程序能够缓解其负载不平衡。全局同步的其他问题都与共享地址空间中的问题相类似。

在共享地址空间中，互斥和点到点事务问题和第 5 章讨论的问题本质上是一样的。和小规模共享存储器型机器一样，可扩展机器的趋势是在基本的交换原语之上，用软件构造用户级同步操作（如加锁和栅障）。但是，两个主要的区别会影响算法的选择。首先，互连网不是集中式的，而是含有很多并行的通路。一方面，这意味着几组不相交的处理器可以通过完全不相交的通路并行相互协调；另一方面，它使得同步原语的实现更加复杂。其次，物理上分布的存储器使得适当地在存储器中分配同步变量变得很重要。它的重要性取决于该机器是否高速缓存非局部共享数据，显然，对于本章所描述的不对非局部共享数据做高速缓存的机器更为重要。本节中介绍几种新的加锁和栅障的算法，它们适合于具有物理的分布存储器并且互连的机型，我们将从讨论用于共享存储器机型的算法开始。在下一章一旦研究了可扩展的高速缓存一致的系统之后，就返回这点进行比较。下面开始讨论加锁算法。

7.9.1 加锁算法

第 5 章 5.5 节中介绍了基本的 test&set 锁、带退回的 test&set 锁、test-and-test&set 锁、标签锁、基于阵列的锁。这些加锁对应的总线流通量和公平性依次减少，但通常，其对应的额外开销也会增加。例如，当一个锁被释放后，标签锁只允许一个进程发出一条 test&set，但是所有的处理器通过一个扑空都得知锁已经被释放，随后的一个读扑空决定谁来发出 test&set。基于阵列的锁解决这个问题的办法是：让每个进程在不同的地址单元等待，释放锁的进程通过写入对应的单元，只通知一个进程锁被释放。

538

但是，基于阵列的锁对具有物理上分布的存储器的可扩展机器有两个潜在的问题。第

一，每把锁需要与处理器数量成比例的存储空间。第二，无法预先知道一个进程将自旋于哪一个地址单元，因为这是在运行时通过 fetch&increment 操作决定的，这一点对远程数据非高速缓冲的机器尤为重要。这样，就不可能把同步变量分配在绕其自旋的进程的局部存储空间内（实际上，第 5 章涉及的所有的锁都有这个问题）。在不具有一致的高速缓冲的分布式存储器机器中，如 CRAY T3D 和 T3E，这是一个很严重的问题，因为进程将在远程地址单元上自旋，导致大量紊乱的通信和竞争。幸运的是，存在一个软件加锁算法，它既可以减少需求空间，又保证所有自旋操作都针对本地分配的变量。这种叫做软件排队锁的锁是当初 Wisconsin Multicube 项目（Goodman，Vernon，and Woest 1989）建议的全硬件锁的软件实现。其思想是每个锁维护一个等待者的分布链表或队列。表中头节点代表持有锁的进程，其他每一个节点表示一个等待该锁的进程，而且该节点分配在对应进程的本地存储器空间内。一个节点指向紧跟在其后试图获取该锁的进程（节点）。还有一个尾指针指向队列的最后那个节点，即最后那个希望获取该锁的节点。让我们首先通过图观察在进程获得和释放锁时队列如何变化，然后将分析获取和释放锁的方法所对应的代码。

假设图 7-39 中的锁最初空闲。当进程 A 试图获取该锁时，它得到了，此时对应的队列如图 7-39a 所示。在图 7-39b 中，进程 B 试图获取该锁，因此它被放入队列当中，尾指针现在指向它。在图 7-39c 中，对试图获取该锁的进程 C 也做类似的处理。在 A 进程持有锁的时候，B 进程和 C 进程绕与其队列节点相连的局部标记自旋。在图 7-39d 中，进程 A 释放该锁。它改写与 B 节点相关联的标记来唤醒队列中下一个进程，即进程 B，然后离开队列。现在 B 持有该锁，处于队列的头部。尾指针保持不变。在图 7-39e 中，类似地，B 释放锁，把锁传给 C，因此 C 既是队头又是队尾。如果 C 在另一个进程试图获取锁之前释放锁，那么该锁指针将为空（NULL），锁也再一次空闲。以这种方式，按试图获取锁的次序先进先出地把锁赋予进程。下面将定义获取锁的次序。

539

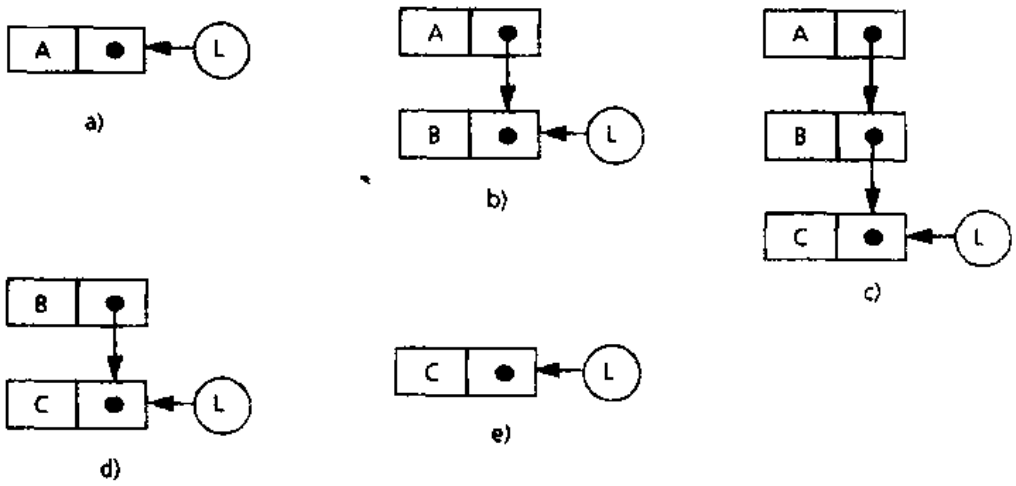


图 7-39　进程申请获取和释放锁时对应的队列状态。随着新的等待进程加入队尾，队列增长。锁被释放时，通知队头的下一个等待者。等待者总是自旋于局部地址单元

获取和释放锁的方法对应的代码如图 7-40 所示。根据所需要的原语，关键是保证改变尾指针操作的原子性。在获取锁的方法中，申请进程要改变锁指针以指向自身节点。它用取并存储（fetch&store）原子操作完成这项任务，该原子操作有两个操作数：它返回第一个操

作数（这里是当前的尾指针）的当前值，然后将它的值设置为第二个操作数的值，只有当成功后才返回。不同进程的 fetch&store 原子操作成功的次序定义了进程获取锁的次序。

```
struct node {
   struct node *next;
   int locked;
} *mynode, *prev_node;
shared struct node *Lock;

lock (Lock, mynode) {
   mynode->next = NULL;                    /*使我成为队中最后一个*/
   prev_node = fetch&store(Lock, mynode);
                                           /*当前指向以前的队尾；原子地把 prev_node 设置为
                                           Lock 指针，设置 Lock 使其指向我的节点，这样我
                                           就成了队尾。*/

   if (prev_node != NULL) {                /*如果在我进入队列时，我不是惟一的一个，就是说，
                                           其他在队列中的进程仍保持着锁*/

      mynode->locked = TRUE;               /*Lock 被其他进程锁定 */
      prev_node->next = mynode;            /*把我连到队列 */
      while (mynode->locked) {};           /*忙等待直到我得到锁 */
   }
}

unlock (Lock, mynode) {
   if (mynode->next == NULL) {             /*似乎没人等着释放 */
      if compare&swap(Lock,mynode,NULL)/*确实没人等着释放 */
        return;                            /*即，Lock 指向我，将 Lock 设置为
                                           NULL，返回 */
      while (mynode->next == NULL);        /*如果我到达这里时，某人正好在队上，并使我的
                                           c&s 失败，我必须等待，直到它们在我把锁给他们
                                           之前，把我的next指针设置为指向它们*/
   }
   mynode->next->locked = FALSE;           /*有人等着锁，释放 */
}
```

图 7-40　软件的排队锁算法。锁数据是一个长度等于该锁上等待者的数量的链表。一个请求锁的节点把对应的表目原子地加入到链表的尾部，然后绕其本地表目自旋，直到前面的申请者解锁后通知它为止

在释放锁的方法中，我们要保持检查操作的原子性：检查释放进程是否为队列中的最后一个进程；如果是，就把锁指针置为 NULL。用比较并交换（compare&swap）原子操作可以实现它，该操作带有 3 个操作数：它比较前两个操作数（这里为尾指针和指向该释放锁进程的指针），如果它们相等，把第一个操作数（尾指针）的值设置为第 3 个操作数（这里为 NULL），然后返回 TRUE；如果它们不相等，则什么也不做，返回 FALSE。置锁指针为 NULL 的操作必须和比较操作保持原子性，否则别的进程可能插进来并且把它自己加入队列，这时还把锁指针置为 NULL 就不对了。回想一下，在第 5 章中，compare&swap 操作很难用单条机器指令来实现，因为它需要在一条存储器指令中有 3 个操作数（但是这个功能可以用加锁载入和条件存储指令来实现）。不用 compare&swap 原语，而只用 fetch&store 原语也可以实现这个排队锁，但是这样实现起来更加复杂（它允许队列被断开，然后将其修复），而且会失去锁授权的先进先出的特征（Michael and Scott 1996）。

显然，软件排队锁需要只是对应于等待或参与该锁的进程数的空间，而不是与程序中的进程数对应的空间。它正是那些以分布存储器支持共享地址空间，但没有一致的高速缓存的

机器选用的锁（kägi，Burger，and Goodman 1997）。

540 ~ 541

7.9.2 栅障算法

在消息传递和共享地址空间两种模式中，像栅障这样的全局事务是一个关键点。一个值得争论的问题是：是否有必要用硬件来支持全局操作，或是否基于点到点操作的成熟软件算法就足够了。CM-5 系统代表了该选择范围的一端：用特殊的“控制”网络在机器的子树上提供栅障、规约、广播和其他一些全局操作。CRAY T3D 也为栅障提供硬件支持。因为很容易构造自旋于局部变量或只使用点对点消息的栅障，所以很多可扩展系统根本不为栅障提供特殊的支持，而是在软件库上构建栅障。

在基于总线型机器的集中式栅障中，所有的处理器在用信号通知它们的到达时，使用同一把锁对同一个计数器加 1，而且在同一个标记变量上等待，直到它们被释放为止。在大型机上，允许所有的处理器访问同一把锁和读写同一个变量会导致大量的通信和竞争。对没有一致的高速缓存的机器，特别如此，因为当几个处理器绕它自旋而又没有对它进行高速缓存时，变量很快成了一个热点。

可以用一种更为分布式的方法来实现到达和离开，这里不要求所有的进程都必须访问同一个变量或锁。到达或释放的协同可以分阶段或轮次进行，在每个轮次中，处理器子集间协同，这样，经历了几个轮次之后，所有的处理器都达到了同步。不同的处理器子集之间的协同可并行地进行，它们之间不需要串行化。在基于总线的机器上，把必须的协同动作分散也没有什么问题，因为总线总是把那些需要通信的动作串行化；但是，这种方法对于具有分布式的存储器和互联网络的机器非常重要，在这类机器上不同子集可以在网络的不同部分进行协同。共享地址空间中应用的技术近乎反映了消息传递方案的本质。下面分析几种这样的分布式栅障算法。

1. 软件合并树

正如第 3 章中避免出现热点的提议，一个简单的协调进程的到达或释放的分布式方法是使用树结构（见图 7-41）。到达树被处理器用来指示它们到达一个栅障。这里用计数器构成的树代替了集中式栅障中的单个锁和计数器。这里的树可以有任何选定的度数或分支因子，比如说 k。在最简单的情况下，树的每个叶子表示参与该栅障的一个进程。当一个进程到达该栅障时，在其相连父节点的计数器上执行一个 fetch&increment 操作（或者给其父节点发一个消息），指示它的到达。然后，它检测 fetch&increment 操作的返回值，判定这是否为其最后一个兄弟节点的到达。如果不是，它对到达的处理工作到此结束，简单地等待释放。如果

542

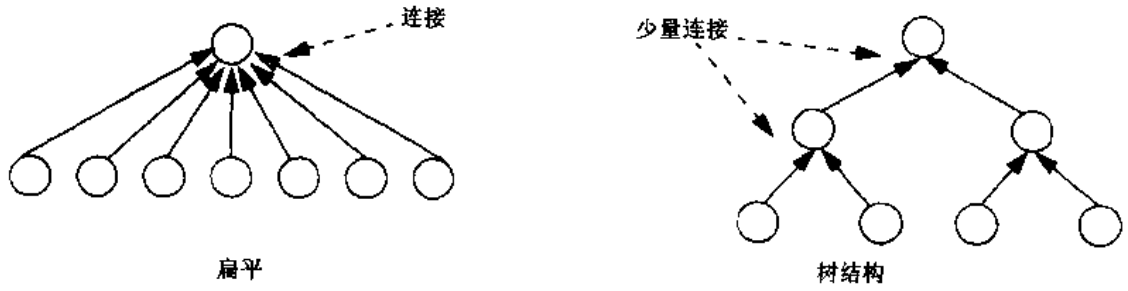


图 7-41　用一棵二叉到达树替代一个扁平到达的结构。对于大规模机器来说，用小分叉的树结构能够利用互联网中许多通路，以避免串行化

是的话，它被推选为高于该树一层中作为其兄弟节点的代表，因此在那层的计数器上执行一次 fetch&increment 操作。按这种方法，每个树节点当其子节点所代表的所有进程都到达时，仅发送一个代表进程到其下一个更高层中。对一个度数为 k 的树而言，它总共有 $\log_k p$ 层，因此能用 $\log_k p$ 步来完成 p 个进程到达的通知。如果进程子树处在网络的不同部分，并且树节点的计数器变量适当地分布在存储器中，没有祖先-子孙关系的节点上的 fetch&increment 操作根本不用串行化。

类似的树结构也可以用于释放事件，因此所有的处理器不会在同一个标记上忙等待。也就是说，最后一个到达栅障的进程设置与树的根节点相联的释放标记，此时该标记上只有 $k-1$ 个进程在忙等待。然后，这 k 个进程中的每个进程都在树的下一层设置一个释放标记，对那个标记有其他 $k-1$ 个进程在等待，这样沿树下降，直到所有的进程都被释放（类似地，也可以沿着树向下发送消息）。根据相关或串行的操作（或网络事务）的数量，栅障的关键路径长度是 $O(\log_k p)$，与此比较，集中式路障的关键路径长度为 $O(p)$，在集中式总线上的任何栅障的关键路径都为 $O(p)$。图 7-42 给出了一个带有检测反转的简单合并树栅障的代码。

```
struct tree_node {
   int count = 0;                    /*技术其初始化为0*/
   int local_sense;                  /*实现检测反转的释放标志*/
   struct tree_node *parent;
}

struct tree_node tree[P];            /*每个元素（节点）被分配在不同的
                                     存储器*/
private int sense = 1;
private struct tree_node *myleaf;    /*指向该进程在树种的叶节点*/

barrier () {
   barrier_helper(myleaf);
   sense = !(sense);                 /*为下一次栅障调用反转检测*/
}

barrier_helper(struct tree_node *mynode) {
   if (fetch&increment (mynode->count) == k-1){     /*最后一个到达节点*/
     if (mynode->parent != NULL)
       barrier_helper(mynode->parent);              /*升到父节点*/
     mynode->count = 0;                             /*为下一次设置*/
     mynode->local_sense = !(mynode->local_sense); /*释放*/
   endif
   while (sense != mynode->local_sense) [];         /*忙等待*/
```

图 7-42　具有检测反转的软件合并树栅障算法。每次使用该栅障，标记的检测被反转，这样就不用将它复位

尽管这种树型栅障把流量分布到互连网络中去了，但是它与没有高速缓存远程共享数据的机器上的简单锁有同样的问题：处理器自旋所绕的变量并不一定分配在其局部存储器中。多个处理器绕着同一个变量自旋时，哪一个处理器到达树的更高层并绕那里的变量自旋，取决于处理器到达栅障并执行其 fetch&increment 指令的次序，而这一点是无法预测的。这就导

致了自旋时的大量的网络流量。

2. 带局部自旋的树状栅障

有两种方法可确保处理器绕一个局部变量进行自旋。一个是基于处理器的标识符和参与栅障的进程数量来预先决定哪个处理器上升到树中对应节点的父节点层。在这种情况下，既然自旋所绕的标记可以分配在自旋处理器的局部存储器中，而不是分配在升到父节点层的处理器的局部存储器中，因此使用二叉树将使得局部自旋更为容易。事实上，在这种情况下，即使没有任何像 fetch&increment 这样的原子操作，只要有下面的简单读写操作也一样可以实现栅障。对到达事务而言，节点的进程之一在到达时只绕该节点相联的到达标记自旋，与该节点相关的另一个进程在到达时简单地对那个标记写入。当另一个进程上升到父节点层时，担当自旋角色的进程则绕与该节点相联的释放标记自旋。这种静态二叉树栅障在文献中被称为“竞赛栅障”，因为到达树中每一步有一个进程退出比赛。(作为一个练习，思考一下怎样改进这一策略使之对参与的进程数不是 2 的幂次方也一样适用，使用非二叉树。)

确保局部自旋的另一个方法是用 p 节点树实现参与进程数为 p 的栅障，树的每个节点（叶节点或内部节点）分配给惟一的进程。唤醒树和到达树可以相同，当然也可以用不同的分叉因子的不同树来实现。树中每个内部节点（进程）在其局部存储器中维护一个到达标记数组，它的每个子节点对应数组的一个项。一个进程到达栅障时，如果它对应的不是叶节点，先检测其到达标记数组，然后等待直到它所有的子节点到达并设置完各自的相关数组项。然后它在其父节点（远程）的到达标记数组中设置自己的对应项，并在唤醒树中与它的节点相关联的释放标记上忙等待。当根进程到达并且其到达标记数组的所有项都被置位时，就意味着所有的进程都到达了。然后根节点会对唤醒树中其所有的子节点的（远程）释放标记置位；这些进程离开了忙等待循环，然后又各自去设置其所有的子节点的释放标记，依此类推，直到所有的进程都被释放。图 7-43 给出了该栅障的代码，这里假定：到达树使用 4 叉树实现，唤醒树用二叉树来实现。一般而言，基于树的栅障的分支因子的选择要在竞争和以网络事务计算的关键路径长度之间进行权衡。以上介绍的几种类型的栅障对不带一致的高速缓存的可伸缩机器都很有效。

543
~
544

```
struct tree_node {
  struct tree_node *parent;
  int parent_sense = 0;
  int wkup_child_flags[2]; /*在唤醒树中的子节点的标志 */
  int child_ready[4];      /*在到达树中的子节点的标志*/
  int child_exists[4];
}
                           /*从根开始，节点逐层标号为0到P－1*/

struct tree_node tree[P];  /*每个元素（节点）分配在不同的存储器 */
private int sense = 1, myid;
private me = tree[myid];

barrier() {
  while (me.child_ready is not all TRUE) {}; /*忙等待*/
```

图 7-43 只围绕局部变量自旋的合并树栅障。每个树节点分配给惟一的进程并且分配在对应进程的局部存储器中

```
set me.child_ready to me.child_exists; /*为下一次重新初始化*/
if (myid !=0) {            /*设置父节点的子节点就绪标志，等待释放*/
  tree[⌊(myid-1)/4⌋].child_ready[(myid-1) mod 4] = true;
  while (me.parent_sense != sense) {};
}
me.child_pointers[0] = me.child_pointers[1] = sense;
sense = !sense;
```

545

图 7-43（续）

3. 并行前序

在很多并行应用中，在组合多个处理器计算的结果信息和分发由组合产生的结果时，都涉及一个同步点。并行前序操作是对归约和广播的重要的概括，其应用广泛（Blelloch 1993）。给定一个具有结合性的二元算子⊕，我们要计算 $S_i = x_i \oplus x_{i-1} \ldots \oplus x_0$，$i=0, \ldots, P$。一个范例是执行求和，但是其他几个算子也很有用。加法器的设计中用到的先行进位操作实际上是并行前序的一种特殊情况。令人惊奇的是，并行前序操作可与后跟一个广播的归约执行得一样快，因为它只是在二叉树中简单地向上向下扫描。图 7-44 显示了向上扫描的过程：与二元归约一样，每个节点对其子节点传送来的一对值进行操作，然后把结果传送给其父节点。（所传递的值由每个弧线旁边的索引范围指示；在此子序列上应用算子得到值。）此外，每个节点保存其最小子节点（图中即为右节点）传来的值。图 7-45 显示了向下扫描的过程。每个节点等待，直到收到其父节点传来的值。它再把这个值原封不动地传给其右面的子节点。然后把这个值与向上传送时保存的值进行组合，再把结果传给其左面的子节点。

树的右边沿上的节点是特殊节点，因为这些节点不必从其父节点处接收任何信息。这样的并行前序树既可用硬件实现，也可用软件实现。

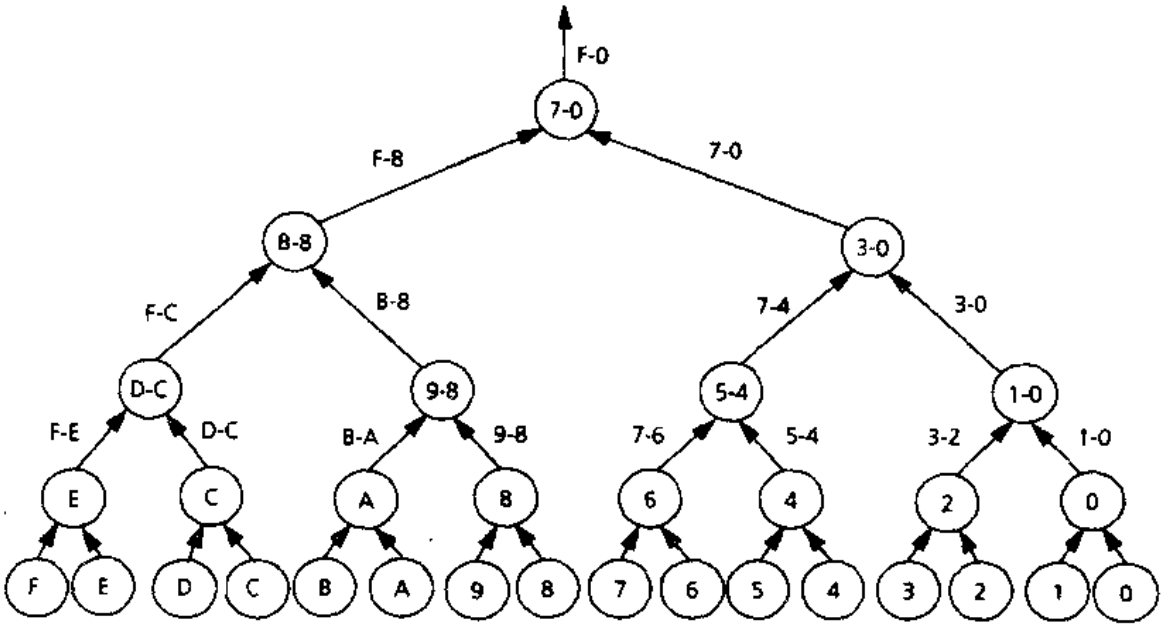


图 7-44 向上的并行前序操作。每个节点从其子节点处收到两个元素，组合后把结果向上传递给它的父节点，并且保存其最小子结点（即右子节点）传来的元素

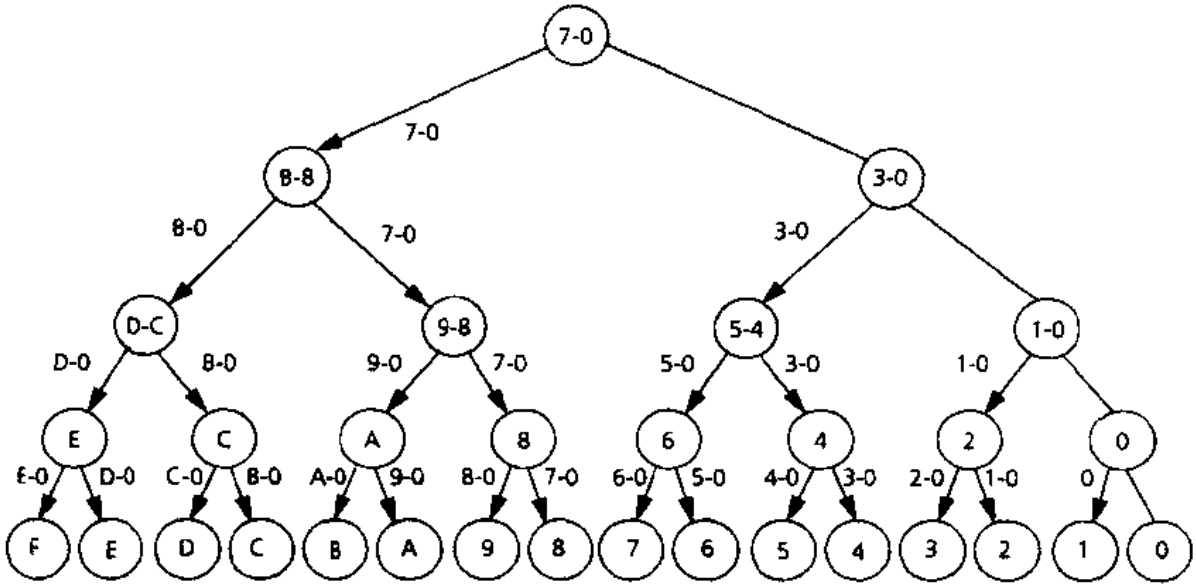


图7-45 向下的并行前序操作。当节点收到其父节点传来的元素后，向下传给其右子节点，并且把将其与自己保存的元素组合，把结果传给其左子节点。最右边分支的节点不需要从其父节点处接收任何信息

4. 全相连的个体间通信

当每个进程都有一组不同数据要传递给其他的每一个进程时，就产生了全相连个体间通信。它的一个范例是矩阵转置操作，比如说每个拥有矩阵的一组行向量的进程需要访问一组列向量的数据。另一个重要的例子是在被阻塞的和形成回路的布线间重新映射一个数据结构。许多这样形式的置换在实践中被广泛地应用。在特殊网络拓扑结构（即无内部竞争网络）上有效地实现全相连个体间通信方面已经做了大量的工作。如果网络是高度可扩展的，网络中的内部通信流是次要的，而网络端点上的竞争成为关键，不管网络的质量如何。一种简单的、广泛应用的方案是：对通信事件序列进行调度，执行 p 个轮次的不相交的成对交换。在第 i 轮，进程 p 向进程 $q = p \oplus i$ 发送它的数据，q 通过二进制数 p 和二进制表示的 i 的异或运算得到。因为异或符合交换律，$p = q \oplus i$，可见该轮次确实实现一个交换。

546 ~ 747

7.10 结论

本章，我们已经了解到大多数现代的大型计算机系统是用通用的节点构成的，节点具有完整局部存储器层次结构，扩充了与可扩展网络接口的通信辅助部件。但是，通信辅助部件的设计选择很多。强烈影响其设计的主要因素是通信辅助部件如何与节点接口：在处理器上？在高速缓存控制器上？还是在存储器总线或 I/O 总线上？另外，通信目标体系结构和编程模型对性能也有极大的影响。大型机上的编程模型是用基本网络事务构造的协议实现的。实现这种协议的挑战之处在于大量的网络传输可能同时处于未决状态，而且没有全局的仲裁。本质上，可以基于任何可用的硬件/软件边界原语实现任何编程模型以及许多正确性和可扩展性问题；但是，其性能特征却大不一样。性能最终影响到机器在程序员眼里如何。

现代大型并行计算机设计主要源于 20 世纪 80 年代中期的技术革命（单片微处理器）在

MPP机开始从传统的向量超级计算机那里抢占高性能计算机的市场时，单片微处理器被称为“杀手微处理器”。不仅如此，这些MPP还开创了自己的革新技术——单片可扩展网络交换器。与微处理器一样，该技术的发展也远远超出了其最初的使用范围，导致了一系列的可扩展系统域网络的出现，包括交换的千兆（或多千兆）以太网。结果，大型并行计算机的设计的发展分成两个分支。基于消息传递概念和显式地读写远程存储器的机器正在被机群所取代，因为机群的成本非常低，工程化容易，又紧随新兴技术。另一个分支是把网络集成到存储器系统中，提供具有自动复制的对全局物理地址空间的高速缓存一致的访问，换句话说，机器在程序员看来与前几章介绍的一样，但却是按本章介绍的那样建立的。当然，提供可扩展带宽和低时延来增强缓存一致性机制依然是一个挑战，这将是第8章的主题。

习题

7.1 用 $\log n$ 个完全并行的步骤去实现 n 个复数的二进制快速傅里叶变换（FFT）程序，每个数据元素读写 $\log n$ 次，总共需要 $5n \log n$ 次浮点操作。如果在舞厅多处理器设计中，所有的处理器都通过网络来访问存储器（如图7-3所示），试计算通信与计算比。若在 p 个处理器上达到 $250p$ MFLOPS，网络需保持多大的带宽？

548

7.2 如果习题7.1中的数据用循环或块分布的方法，分散在NUMA设计的存储器中，有 $\log n/p$ 步访问本地存储器中的数据，假设 n 和 p 都是2的幂，在剩下的 $\log p$ 步中有一半的读和一半的写是本地的。（数据布局的选择决定了哪些操作是本地的，哪些操作是远程的，但是其比率是一样的）。在如图7-2所示的分布式存储器设计中，每个处理器都有一个本地的存储器，试对该设计计算通信与计算比。p 个处理器时为了达到 $250p$ MFLOPS，网络需保持多大的带宽？将其与习题7.1的计算结果比较。

7.3 如果程序员进行了很好的数据布局，对FFT程序可以做到：在网络中传输 $n-n/p$ 个数据元素，需要进行一次全局的转置操作。所有 $\log n$ 步都是在本地存储器中执行。试计算分布存储器型机器的通信与计算比。p 个处理器时为了达到 $250p$ MFLOPS，网络需保持多大的带宽？将其与习题7.2的结果进行比较。

7.4 重新考虑例7.1，n 个节点的配置中跳数为 $\sqrt{n}$。其平均传输时间如何随节点数目的增加变化？$\sqrt[3]{n}$ 时结果又怎样呢？

7.5 对于习题7.4的设计的成本扩展进行形式化。

7.6 考虑例7.1中描述的机器，每次传输占用的链路数为 $\log n$。如果链路不存在竞争，可以同时发生多少个传输？

7.7 重新考虑例7.1，这里网络是一个简单的环。n 个节点的环中两个节点间的平均距离为 $n/2$。其平均传输时间如何随着节点数的增加变化？假定每条链路同时最多只能被一个传输占用，有多少这样的传输可以同时发生？

7.8 对于例7.1中描述的机器，假设一个节点向其他所有的节点进行一次广播用了 n 条链路。试分析同时进行的广播数与节点数之间的扩展关系。

7.9 假设一个16路的SMP系统价格为1万美元外加每个节点2 000美元，每个节点带有一个快速处理器和128 MB的存储器。系统规模从4个处理器加倍到8个处理器时，成本增加了多少？从8个处理器加倍到16个处理器呢？

7.10 试证明7.1.3小节中的命题：只要加速比（p）>成本上升比（p），并行计算就是性

价比合算的。 549

7.11 假设一个有效负载为 n 个字节的总线事务占用总线 $4+\left\lceil\frac{n}{8}\right\rceil$ 个周期，其上限为 64 个字节。画图比较用程序 I/O 和 DMA 发送在存储器中（不是在寄存器中）的可变大小的消息数据各自对应的总线的利用率。对 DMA 而言，包括了将 DMA 地址和长度通知给通信辅助部件的额外工作。分别考虑数据在高速缓存中与不在高速缓存中两种情况。读取状态寄存器需要做什么假设？

7.12 Intel 的 Paragon 的输出缓冲区的大小为 2 KB，可从存储器中以 400MBps 的速率对之进行填充，缓冲区以 175MBps 的速率向网络传输数据。缓冲区在被填充的同时将数据传给网络，但是如果缓冲区满，DMA 设备将暂停。设计消息层时，你决定将长消息分成 DMA 区段，DMA 区段要尽可能长，但又不会长得使 DMA 设备被输出缓冲区堵塞。显然，其大小至少可以是 2 KB，但实际上可以更长。计算在给定的流速率下，把 DMA 突发数据驱动到网络上去的输出缓冲区的尺寸。

7.13 根据图 7-33 的粗略估计，如果使用线性模型拟合通信时间数据，哪一种机器将具有负的 T_0？

7.14 在表 7-1 中描述的机器上运行基准测试程序 BT，请根据表 7-2 提供的消息的频率数据来估测一次迭代中每个处理器用于通信的时间。

7.15 表 7-3 描述了稀疏 LU 基准测试程序的通信特征。

1）其消息尺寸特征与 BT 的有何不同？这说明了应用的什么问题？

2）消息的频率有什么不同？

3）对表 7-1 中描述的机器而言，估计其运行 LU 基准测试程序时每个处理器在一次迭代中的通信上所耗的时间。 550

表 7-3 一定数量范围的处理器上执行 LU 基准程序 A 类问题时一次迭代的通信特征

	4 处理器			16 处理器			32 处理器			64 处理器		
	消息尺寸（KB）	消息	总数据传输量（KB）	消息尺寸（KB）	消息	总数据传输量（KB）	消息尺寸（KB）	消息	总数据传输量（KB）	消息尺寸（KB）	消息	总数据传输量（KB）
	1+	496	605	0.5－1	2 976	2 180	0－0.5	2 976	727	0－0.5	13 888	3 391
	163.5+	8	1 279	81.5+	48	3 382	0.5－1	3 472	2 543	81.5	224	8 914
							40.5+	48	1 910			
							81.5+	56	4 471			
总通信量（KB）			1 884			5 562			9 651			12 305
一次迭代的时间（s）			2.4			0.58			0.34			0.19
平均带宽（MBps）			0.77			10.1			27.7			63.2
每个处理器的平均带宽（MBps）			0.19			0.63			0.87			0.99

551

第8章　基于目录的高速缓存一致性

本章将讨论并行体系结构开发中的一个重要的问题：如何把高速缓存一致性和可扩展的具有分布式存储器的机器组成结构结合在一起。我们已经研究了如何解决具有集中存储器、基于总线的机器的高速缓存一致性问题。我们也看到，为了机器的扩展性，采用了分布式存储器、可扩展的点到点互连网络和一个为支持编程模型而对网络事务进行不同程度解释的通信辅助部件。不管通信辅助部件的复杂程度如何，我们所学习的所有可扩展的机器具有图8-1所示的通用结构。

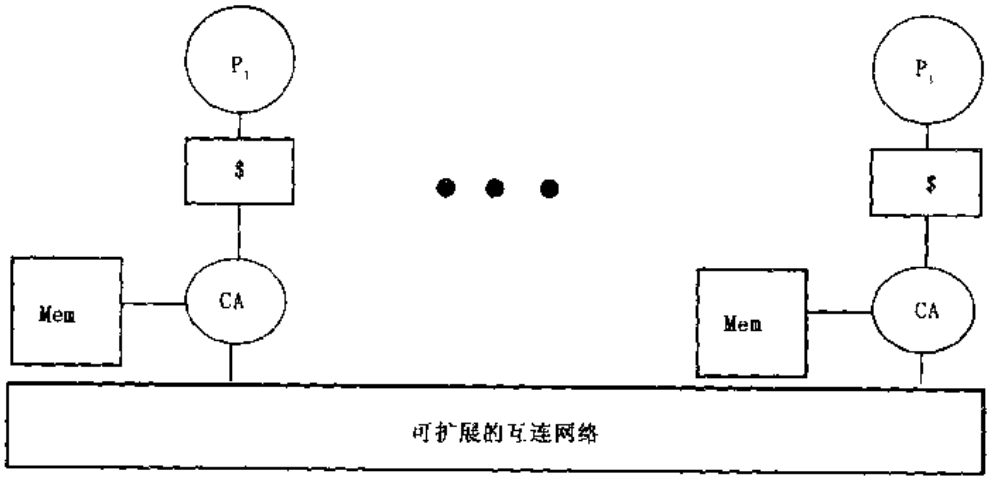


图8-1　通用的可扩展多处理器。本图表示了第7章中讨论的多处理器的通用结构：带有物理分布的存储器的处理节点和可扩展的互连网络。处理节点可以是单处理器（如图所示）或多处理器

在我们设计范畴的最后一点，通信辅助部件以硬件提供了共享的地址空间。但是，尽管高速缓存的用途是复制共享地址空间中被访问的数据，我们还没有讨论如何解决高速缓存的一致性问题。事实上，为了避免一致性问题和简化存储器的同一性，在前面讨论机器的最终设计点时，我们关闭了对物理上远地分布、逻辑上共享的数据的硬件高速缓存，限制了编程模型。

本章将研究如何在不具备总线这样的全局侦听互连机制的分布式存储型机器上，用硬件提供隐含的高速缓存及其一致性的重要问题。正如所看到的，不仅硬件的时延和带宽必须很好地扩展，用于一致性的协议也必须有很好的扩展性，至少达到实践中需要的规模。我们集中讨论对高速缓存一致性的全硬件支持，特别是最常见的叫做基于目录的高速缓存一致性的方法。根据抽象的层次，在硬件/软件接口处直接支持使用一致复制的共享地址空间编程模型，如图8-2所示。其他的一些编程模型，比如消息传递可以用软件实现。下一章将介绍其他几种采取不同硬件/软件折中的方法，例如在存储器中而不是在高速缓存中进行一致性复制，软件控制的同一性和其他的存储器同一性模型。

553

典型的可扩展高速缓存一致性基于目录的概念。因为高速缓存中块的状态不再能通过把请求放到共享总线上，让高速缓存控制器侦听来隐式地决定，所以用目录来显式地维护块的

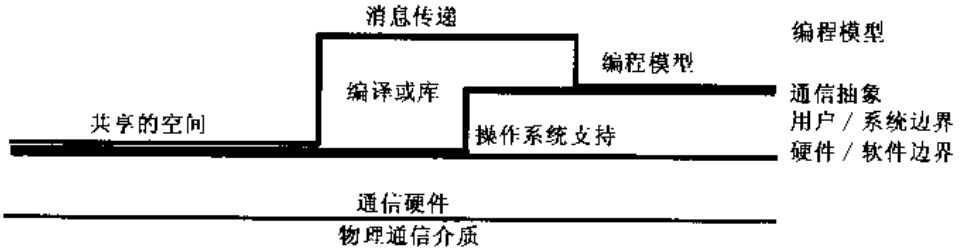


图 8-2　本章所讨论系统的抽象层次。硬件直接对一致的、共享的物理地址空间提供支持，通过软件层支持消息传递

状态，请求能发往并查询目录。考虑一个简单的例子，设想大小等于高速缓存行的每一个主存储器块都与一个高速缓存记录相联系，那个记录同时包含该块的副本和该块在高速缓存中的状态。这个记录就叫做那个块的目录项（见图 8-3）。同基于总线的系统一样，可能很多高速缓存里面有干净、可读的数据块，但如果在一个高速缓存中的块是可写的（可能已经被修改），那么可能只有这个高速缓存中的数据是有效的。当一个节点发生高速缓存扑空时，它先用点对点的网络事务同该块的目录项通信。因为目录项和主存的块一起存放，其地址可以从块的地址得到。节点可以从目录项判断有效的高速缓存副本在哪里，并确定下一步该怎么做。它接着根据需要通过网络事务与那个高速缓存块联系。例如，它可以从另一个节点获得修改过的块，或执行写操作，向其他节点发出作废信号并从那些节点接受应答。高速缓存块状态的改变也通过网络事务送给目录项，使目录保持最新。

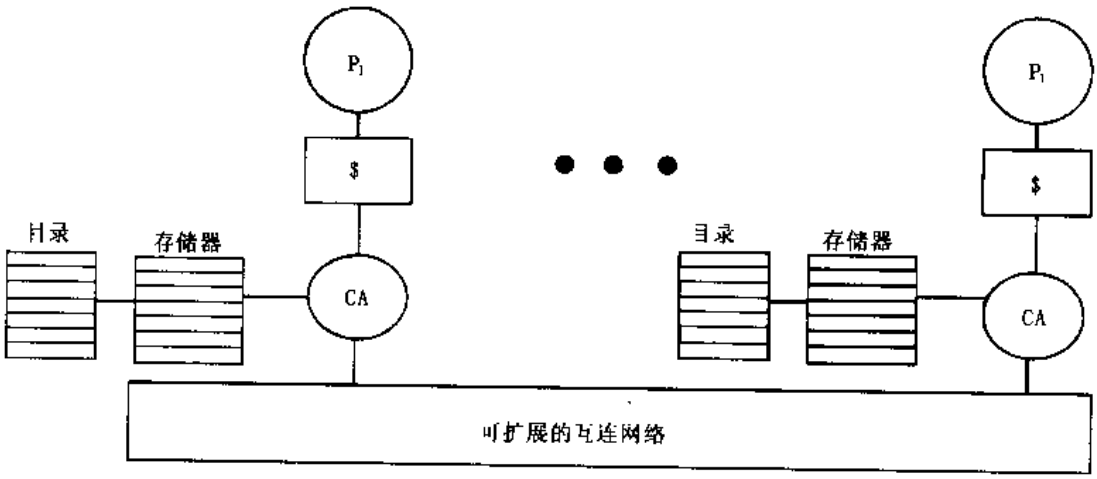


图 8-3　带有目录的可扩展多处理器。每一个和高速缓存块大小相同的主存块都有一个目录项，记录该主存块的高速缓存副本和它的状态

在目录协议中，节点间的请求、应答、作废、更新、确认都是像第 7 章中所提到的网络事务，只不过事务目的节点的端点处理（作废一个块，取出数据并应答）是由通信辅助部件而不是由主处理器完成的。（同第 7 章一样，我们把携带数据“应答”和其他所有信息的响应事务简称为“响应”。）由于目录方法依赖点对点的网络事务，它可用于任何互连网络。目录信息的存储方式和针对该表示形式的正确有效的协议设计都是目录方法的重要问题。

尽管目录法在可扩展的高速缓存一致性方面占主导地位，还有其他一些方法值得关注。已经被尝试过的一种方法是利用广播介质的层次结构如总线或环，对广播和侦听机制的扩展。这种方法在概念上很有吸引力，因为它能用现有的小规模的机制层次式地构造更大的系

554 ~ 555

统。但是它并不适用于网格或立方体这样通用的网络拓扑结构，我们将会看到它在时延和带宽方面存在问题，因此它不是非常流行。比较流行的是一种两层协议方式。机器的每个节点本身都是一个多处理器，节点内部的高速缓存一致性通过一种叫做内部协议的一致性协议来保持，节点之间的一致性则由另一种叫外部协议的不同协议来维护。在外部协议看来，每一个多处理器节点就像单一高速缓存，节点内部的一致性是内部协议的责任。通常，用一个适配器或一个共享的第三层高速缓存在外部协议中代表一个节点。一种通用的结构是，外部协议用目录协议，内部协议用侦听协议（Lovett and Clapp 1996；Lenoski et al. 1993；Clark and Alnes 1996；Weber et al. 1997）。当然也可以用其他的组合如侦听-侦听（Frank，Burkhardt，and Rothnie 1993），目录-目录（Convex Computer Corporation 1993）和侦听-目录等（见图 8-4）。

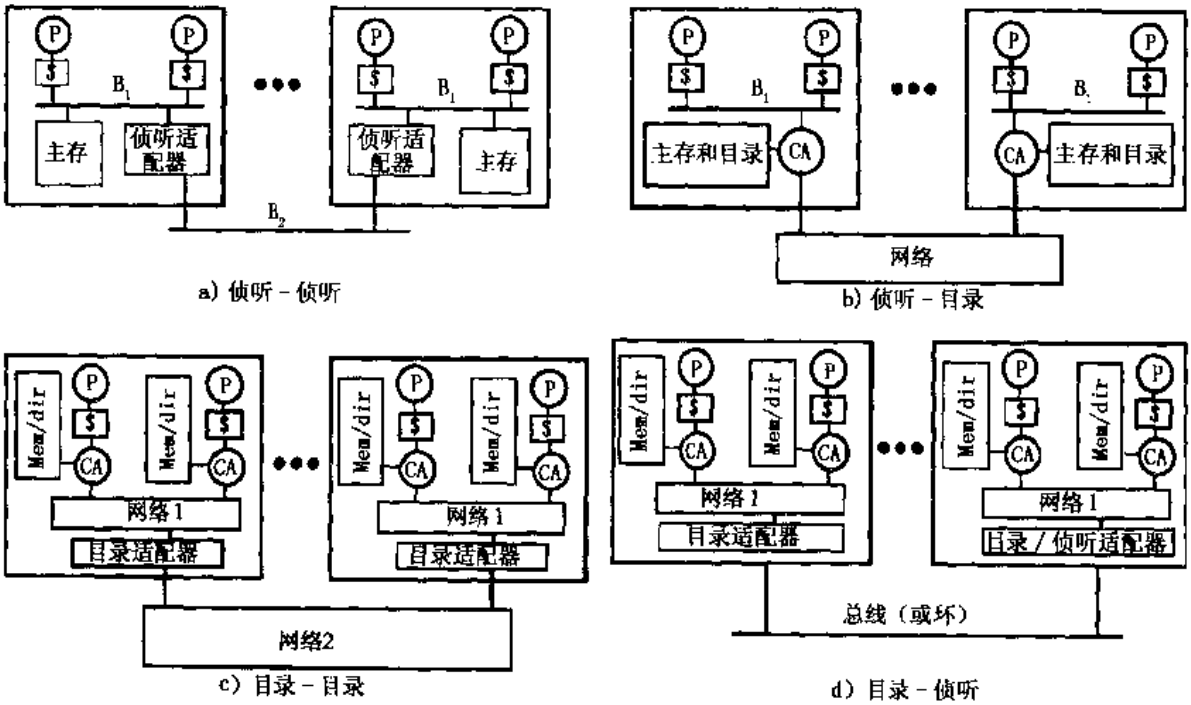


图 8-4 两层的高速缓存一致系统的一些可能的组成结构。外层可见的每个节点本身是一个多处理器。B_1 是第一级总线，B_2 是第二级总线。CA 是通信辅助部件。例如，侦听-目录表示在多处理器节点上用侦听协议维护一致性，用目录协议维护节点间的一致性

用较小规模的机器构成更大的机器的这种两层结构是一种有吸引力的工程选择：它把固定的节点成本在节点内各处理器间分摊，可以利用层次封装的优点，同时节点内部处理器间的通信代价也更低。本章的重点是节点间的目录协议，不管节点是单处理器还是多处理器，也不管它使用什么一致性方法。同时也讨论两层协议间的相互影响。因为目录协议最为成功并很可能最为流行，将主要讨论它，但我们也将简要地考察不太流行的层次式方案。在考察目录的组织结构、支持一致性和同一性的协议以及对通信辅助部件的要求时，我们会发现其他丰富而有意义的设计空间。

本章第一节提出了一个框架，使读者理解在共享地址空间提供一致复制的不同方法，包括侦听、目录和层次式侦听等。8.2 节先介绍了采用简单目录表示的目录协议的基本操作，然后概述了另外一些目录结构和协议。8.3 节给出了对一些高层问题的量化评价和对目录协议在体系结构上的折中。

接下来的几节覆盖了实际设计正确、高效的协议时涉及的问题和技术。8.4 节讨论了在不存在串行化的互连网络情况下，数据的多个副本所带来的新的挑战。其后的两节对两种最流行的基于目录的协议进行了深入的研究，讨论了不同的设计方案，并用两种商品化机的体系结构——Silicon Graphics 公司的 Origin2000 和 Sequent Computer Systems 公司的 NUMA-Q——作为研究实例。8.7 节考察了通信体系结构的关键性能参数对目录协议下并行程序端性能的影响。

8.8 节讨论了基于目录的多处理器的同步，8.9 节讨论了对并行软件的影响。8.10 节覆盖了一些高级主题，包括为实现可扩展一致性的层次式扩展侦听和目录协议的方法。

556 ~ 557

8.1 可扩展的高速缓存一致性

本节简要地介绍了在多处理器的存储器层次结构中提供一致复制的几种主要方法和任何一致性方法都要提供的基本机制。

在一个具有物理分布存储器的机器上，非本地数据可以仅在处理器的高速缓存中复制，或仅在本地主存中复制。如果主存中提供了一致的复制，因为本地存储器中一致的数据才能进入高速缓存，所以就没有必要再去保证高速缓存的一致性。本章假定数据只自动地在高速缓存中复制，不在主存中复制，并像基于总线的机器那样，由硬件以高速缓存块的粒度保证数据的一致性。由于主存在物理上是分布的，具有非均匀的访问代价，这种类型的体系结构被称为*高速缓存一致的非均匀存储器访问*，简称 CC-NUMA 体系结构。更一般地，用分布式存储器和一致复制（在高速缓存或主存）的系统被称为*分布式共享存储器*（DSM）系统。

任何一致性方法，包括第 5、6 两章介绍的侦听一致性，都必须提供一些关键的机制，首先，一个块的副本可以存在于任何一个高速缓存（或本地复制存储器）中，并处于几个状态之一，在不同高速缓存中的状态可能是不同的。协议必须提供这些高速缓存状态和状态转换图以及与状态转换图相联系的动作集合，根据状态转换图，不同高速缓存中的块独立地改变状态。基于目录的协议中每个块也有一个*目录状态*，即目录所知的块的状态。协议可以基于作废，基于更新或者两者混合的，但不管系统是基于侦听还是目录，高速缓存的稳定状态本身通常是相同的（例如 MESI）。选择高速缓存稳定状态时的权衡同第 5 章所讨论的类似，我们在此不再赘述。对任何协议，从概念上讲，一个存储块的高速缓存状态是一个包含了它在系统每一个高速缓存中所处状态的向量。虽然在某一时刻一个数据块的当前状态在不同高速缓存中可能不同，但同一个状态转换图控制不同高速缓存中的副本。不同高速缓存中块状态的变化通过互连网络上的事务，如总线事务或更通用的网络事务来协调。

给定一个高速缓存状态转换级的协议，一个一致的系统必须提供管理协议的机制。首先，需要一个机制来决定何时（即针对什么操作）激活协议。在大多数系统中做法都是一样的，即通过*访问失败*（高速缓存扑空）的检测机制。如果处理器的访问不能被本身的高速缓存所满足，例如，访问的块不在高速缓存，或要写的块在高速缓存中但处于共享状态，就要调用协议。但是，即使不同的高速缓存一致性方法使用相同的状态集、状态转换和访问失败机制，但它们在为访问失败发生时执行的三个重要功能所提供的机制差别很大：

558

1）发现关于单元（高速缓存块）在其他高速缓存中状态的足够信息，确定应采取的动作。

2）如果需要的话（例如，将它们作废），定位块的其他副本。

3）与其他副本进行通信（例如，从它们得到数据或者将它们作废或更新）。

在侦听协议中，这三个功能都是通过广播和侦听机制实现的。处理器把“查找”请求放在总线上，里面有块的地址和其他高速缓存控制器侦听并响应。也可能在分布式机器中使用广播和“侦听”的方法；发生扑空的节点上的通信辅助部件可以对所有节点广播，其他节点的通信辅助部件检查进入的请求并作出适当的响应。但是，广播产生了大量的流量（在 p 个节点的机器上每个扑空上至少是 p 个网络事务），因此扩展性不够好。可扩展的方法包括层次式侦听和基于目录法。

在层次式侦听协议中，互连网络不是单一广播总线（或环），而是一个总线树。树的叶子是基于总线的侦听多处理器。父总线与子总线通过接口相连，接口侦听上下两侧的总线，并且把相关的事务向上或向下传播。主存储器既可以集中在根部，也可以分散在叶子上。在这种情况下，上面所有功能由广播和侦听机制的分层执行：处理器像以前一样把“查找”请求放到总线上，“请求”会根据侦听结果需要往上层或下层传输。我们希望，在大多数情况下，请求都不必被传输得特别远。8.10.2 节会进一步讨论层次式侦听系统。

在前面提到的简单的目录法中，其他高速缓存中块的状态通过网络事务查找目录得到，副本的地址也是从目录中查到的。与副本的通信都是通过任意互连网络中的点对点的网络事务来进行，无须求助于广播。目录信息的实际组织方式会影响到使用网络事务的这个组织结构的协议对上述三个关键功能的实现。

8.2　基于目录方法概述

本节将首先详细地介绍一个简单的目录方案，阐述它是如何利用高速缓存的状态、目录的状态和网络事务工作的。然后讨论把目录扩展到大量节点的组织结构上的问题，提供了可扩展目录组织方案的分类并探讨了与这些结构相关的协议的基本问题。

559

下列定义在我们讨论目录协议时会用到。对于一个给定的高速缓存或存储器块：

- *宿主节点*。包含了块的主存储器所在的节点；
- *脏节点*。其高速缓存中有块的副本并处于被修改过（脏）状态的节点。注意一个块的宿主节点和脏节点可能是一个。
- *拥有者节点*。指当前保持块的有效副本，在需要时能够提供数据的节点；在目录协议中，这可能是宿主节点（当块不在高速缓存中并处于脏状态）或脏节点。
- *排他节点*。指高速缓存中有块的副本并处于排他状态的节点，或者为脏，或者为（干净）的独占（回忆第 5 章，叫做排他的状态意味着这是仅有的有效副本，主存中的块是最新的）。所以，脏节点也是排他节点。
- *本地节点或请求节点*。发出对块的请求的处理器所在的节点。
- 块的宿主对于请求的处理器而言是本地的话，这样的块称为*本地分配块*，或简称*本地块*，所有其他的块都称为*远程分配块*或*远程块*。

下面让我们从基于目录协议的基本操作开始，使用一个非常简单的目录结构。

8.2.1　简单目录方案的操作

当高速缓存扑空（访问控制失败）发生时，本地节点就向宿主节点发出一个请求网络事

务，从那里获得所需块的目录信息。发生读扑空时，目录指出可以从哪个节点获得数据，如图 8-5a 所示。发生写扑空的时候，目录发现块的各个副本，然后向它们发出作废或更新网络事务，如图 8-5b 所示。（回忆一下，对于处于共享状态的块的写也被视为写扑空。）由于作废或更新是通过网络上不同的路径被发往多个节点，我们必须得到所有副本对作废的显式确认，才能确定一个写操作的完成。当排他读请求或更新请求获得对互连网络的访问时，我们不能像在共享总线的情况下那样假定操作已经完成，因为我们无法保证互连网络中事务的次序。

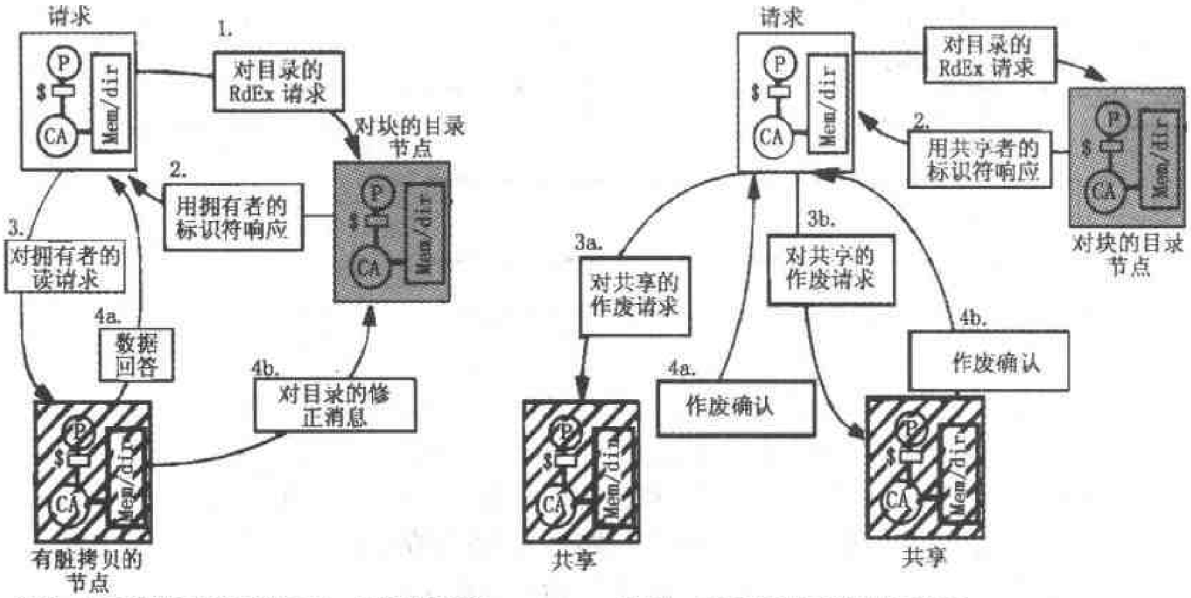


图 8-5　一个简单目录的基本操作。这里显示了两个示例操作。左图是读扑空情况，被读的块当前处于被修改（脏）状态被保持在一个节点上，该节点既不是请求节点，也不是具有目录信息的宿主节点。如果读扑空发生在一个“干净”的主存块上，情况就较为简单：主存只要在应答时将所需的数据发送给请求的节点即可，一个请求-响应事务对就可以满足扑空的要求。右图是对于在另外的两个节点的高速缓存（两个共享者）中处于共享状态的块写扑空的情况。大的方块是节点，带有加框标志的弧线代表网络事务。事务框旁边的数字 1，2 等等，代表事务的串行顺序。相同数字旁边的不同字母表示这些事务可以同时执行，因此重叠

组织目录的一个自然的方法是将块的目录信息和块本身一起保存在存储器中，也就是说放在该块的宿主节点。块的目录信息的一种简单的组织方式是将其组织成一个 p 个存在位的向量和一个或多个状态位（如图 8-6 所示），p 代表节点的个数，p 位分别指示相应节点（单处理器或多处理器）是否有该块的一个高速缓存副本。我们可以简单地假定只有一个状态位，叫做脏位，它是用来指示该数据块是否在某一节点的高速缓存中被修改过。当然，如果脏位被置为 ON。那么只有一个节点（即脏节点）的高速缓存拥有该块的副本，并且只有那个节点的存在位被置为 ON。使用这种结构，读扑空时通过查目录表可以很容易地知道哪个节点持有该块的脏副本，或该块是否在宿主节点的主存中有效。写扑空时能判断哪些节点是必须作废的共享者。

560
561

一个块的目录信息是从主存储器观点对不同高速缓存中该块状态的观察。目录不需要了解每个高速缓存中的确切状态（即MESI），只需要得到能决定采取什么动作的足够信息，

所以目录中的状态就比高速缓存的状态要少。事实上，由于目录和高速缓存之间通过分布式互连相互通信，某段时间就可能出现这种情况：高速缓存的状态已经被改变，但状态改变的通知还没有到达目录，于是目录中的状态便是不正确的。在这段时间内，目录可能基于自己旧的（不再有效的）信息，向高速缓存发出了一个消息。这种状态分布引起的竞争使得目录协议更有意思，在 8.4～8.6 节里，将看到如何用过渡状态或其他手段来解决这些问题。

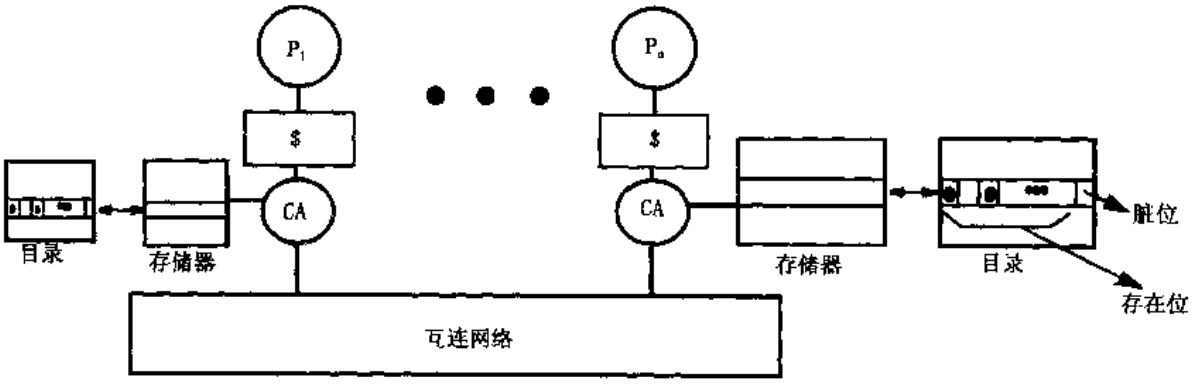


图 8-6 分布存储器型多处理器的目录信息。在简单的组织结构中，一个块的目录项是一个包含 p 个存在位的向量（每个节点对应一位）和一个表示该块是否在任何节点中处于被修改状态的位

为了更详细了解读扑空和写扑空是如何与这种位向量目录结构发生交互作用的，考虑一种有三种稳定高速缓存状态（MSI）、每节点一个处理器、每个处理器一级高速缓存的协议。协议由叫做一致性控制器或目录控制器的辅助部件指挥。当在节点 i 发生读或写扑空时（包括从共享状态的升级），本地的通信辅助部件或控制器查找存储器块的地址，确定其宿主是本地的还是远程的。如果它是远程的，便向块的宿主节点发出一个网络事务。宿主节点的辅助部件负责查询该块的目录项，并使用与图 8-5 中类似的网络事务对扑空做下列处理（本章后面将讨论更为优化的处理）：

562

- 如果脏位为 OFF，那么辅助部件从主存储器取得块，用一个应答网络事务把它提供给请求者，把位向量中第 i 个存在位，即 presence［i］位，置 ON。
- 如果脏位为 ON，那么宿主节点便把与置位的存在位对应的节点（即拥有者节点或脏节点）的身份发给请求节点，然后请求者向那个拥有者节点发送一个请求网络事务。在拥有者节点，高速缓存的状态变为共享，并把块发送给请求节点以及宿主节点的主存，请求节点把块以共享状态存入自己的高速缓存。在主存中，对应的脏位为 OFF，presence［i］位被置为 ON；

处理器 i 的写扑空要访问存储器，按以下进行处理：

- 如果脏位为 OFF，主存储器中的数据副本是干净的。这时必须向所有其 presence［i］是 ON 的节点 i 发送作废请求事务。假设一个图 8-5 中所描述的一个严格的请求-应答场景，宿主节点把块和存在位向量发送给请求节点 i。目录项被清除，只留下对应节点 i 的存在位 presence［i］和脏位。（如果请求是升级而不是排他读的话，向请求节点返回的确认包含位向量而不是数据本身。）请求节点的辅助部件对被需要的节点发送作废请求，然后等待那些节点的作废确认事务，表示对它们的写已经完成。最后，请求节点把块放入自己的高速缓存，状态置为“脏”；
- 如果脏位被置为 ON，那么先使用网络事务从脏节点（它的存在位被置为 ON）恢复

到宿主节点的主存中，脏节点高速缓存对应状态置为无效，然后块被发送到请求处理器，请求处理器把块放入自己的高速缓存，状态置为“脏”。目录项被清除，只留下对应节点 i 的存在位 presence [i] 和脏位为 ON。

当节点 i 替换一个脏块时，将要被替换掉的数据会被写回主存储器，目录随之更新，清除脏位和 presence [i]（同基于总线的处理机类似，回写引起有意思的竞态条件，后面将结合实际协议进行讨论）。最后，如果一个处于共享状态的块被从高速缓存替换掉，是否向目录发出一个消息把对应的存在位复位都可以，所以在下一次对该块写入时不向该节点发送作废请求。这种消息被称为替换提示，是否发送它并不影响协议或执行的正确性。

一种与此类似的目录方案早在 1978 年就被设计出来了（Censier and Feautrier 1978）。它是专为有集中式存储器、少量处理器的系统设计的，并在 Lawrence Livermore 国家实验室的 S-1 多处理器项目中得到了应用（Widdoes and Correll 1980）。而且甚至在此之前，目录方案就以这样或那样的形式被使用了，其最早的应用是在 IBM 的大型机里，这种机器用高带宽的交换机把少量的处理器和集中式的存储器连在一起。由于没有可供侦听的广播介质，在主存储器里便维护了各个处理器高速缓存标签的副本，具有目录的功能。存储器接到请求后先查看所有标签以确定块在其他高速缓存中的状态（Tang 1976；Tucker 1986）。当然，存储器中的标签副本必须保持最新。因为在这些较早的方案中目录都是集中式的，它们被称为集中目录方案。 563

目录的价值在于它记录了哪个节点有某块的副本，消除了广播。显然，在发生读扑空时非常有用，因为对于块的请求或者可以直接在主存储器得到满足，或者可以从目录确切地知道从哪个节点可以得到排他的副本。在发生写扑空时，如果块的共享者（必须对它们发送作废或更新）数目比较少，并且也不随处理节点数量的增加过快地增长，那么目录表方案相对简单广播方案的优势最大。

从我们对并行应用程序的理解，可能已经想到一般来说块的共享者的数目应该比较小。例如，在近邻网格计算中，不管网格的大小有多大，也不管处理器的数目有多少，通常只有两个处理器，最多四个处理器会在分区边界共享一个块。即使在一个应用程序中某一块被所有处理器不断地读写，在写时应该被作废的共享者的数目也取决于处理器读和写时间上的交错关系。一个常用的例子是迁移的数据，即数据先被一个处理器读写，然后被另一个处理器读写，依次类推（比如每个处理器都把自己的值累加到一个全局和上）。尽管所有的处理器都读写同一个单元，但只有前一个写数据块的处理器拥有有效副本并必须被作废，而所有其他副本在前一个写之前已经被作废了。

对程序行为的实验测量表明，在大多数情况下，被写的共享数据的有效副本的数目总是非常小的，并且不随使用的处理器数目的增加而迅速增长，产生大量作废的写的几率也非常小。在 8.3.1 节根据应用特征还会提出并分析这样的并行应用数据（注意，即使如果在大多数写操作时，必须把运行应用的所有处理器作废，但若应用不是在多处理器的所有节点上运行的话，这种目录对于写仍然是有价值的）。这些事实使基于目录方案的可扩展性有良好前途，并能帮我们理解如何经济有效地组织目录。

8.2.2 可扩展性

采用目录协议的主要目的是使高速缓存一致性协议可扩展，使处理器数量超过总线所能

支持的限度。从性能和目录信息的存储开销两方面来理解目录协议的可扩展性是很重要的。具有分布式的存储器和互连网络的系统在正常负荷下，已经在原始时延和带宽方面提供了很好的可扩展性。一个协议在性能可扩展性方面的主要问题是它在时延和带宽方面对系统的需求如何随使用的处理器数量而扩展。带宽需求由每次扑空产生的网络事务量与扑空率的乘积决定；时延由扑空时关键路径上事务的数量决定。进而，目录的组织方式和协议对网络事务的优化程度也会对这些量产生影响。但是，存储开销仅仅由目录信息的组织方式决定。对于那种简单的位向量结构，所需要的存在位的数量随处理节点的数目（每个存储器块 p 位）和主存储器的大小（每存储块一位向量）两者线性增长，导致目录的存储开销巨大。对于有64个处理器、每块64字节的系统，目录的存储开销与其余存储空间的比值为：64位（外加上状态位）除以64字节，约为12.5%，这种情况还不是太坏。如果块大小不变，处理器数目增加到256个，存储开销是50%，处理器增加到1 024个时，比值竟成了200%！目录开销的扩展性不好，尽管在目标机的节点数目较小时，这个协议还是可以接受的。

564

幸运的是，很多组织目录信息的其他方法改进了目录存储的可扩展性。不同的目录组织方法自然导致了不同的高层协议，它们以不同的方法解决8.1节提到的三个协议功能，具有各自的性能特性。本节的其余部分将列举一些目录组织，并将简要地描述使用这些组织方法的简单协议如何处理一个读或写扑空。我们假定当时没有另外的高速缓存扑空，即没有竞争条件，这样目录和高速缓存总是处于稳定的状态。在8.4～8.6节里我们将讨论更深层次的协议问题。

8.2.3 组织目录表的其他方法

由于与高速缓存副本的通信总是通过网络事务进行，各种方法之间的差异一般来说体现在一致性协议的前两个功能：发生扑空时寻找相应的目录信息源和确定相应副本的位置。

寻找一个块的目录信息源的方法主要有两类：*扁平目录方案*（flat directory schem es）和*层次目录方案*（hierarchical directory schemes）。

我们前面介绍的简单目录方案就是扁平方案。之所以称之为“扁平”是因为一个块的目录信息是存放在一个固定的地方，通常在有块地址指示的宿主节点；扑空时，一个请求网络事务直接送给宿主节点以进行目录查询（如果宿主是远程的话），不管宿主节点有多么远。

在层次目录方案中，目录信息的源不能事先得知。存储器仍然分布在各处理器中，但每块的目录信息逻辑上被组织成层次数据结构（一棵树）。树的叶子是各拥有一部分存储器的处理节点，树的内部节点是层次式维护的块的目录信息：一个节点记录它的子树中是否有某一块的副本。高速缓存扑空时，可以通过网络事务一层一层地向上遍历，直到到达一个目录节点，它指出其子树中有适当状态的块复本为止。因此，扑空的处理器只是简单地向其父节点发出一个查找消息，然后向上类推，而不是直接向宿主节点发出一个网络事务。一个块的目录树只是逻辑上的，物理上并不一定要有一棵树，目录树可以嵌在任何通用的互连网络上。每一个块都有自己的逻辑目录树。事实上，系统中的每一个处理节点既是叶子节点（对于它所包含的数据块来讲），又是存有目录信息的内部节点（对其他的数据块来讲）。

565

在层次方案中，关于副本位置的信息也由层次结构自身维护，在目录信息的指引下通过向上和向下遍历找到副本并与之通信。例如，一个节点的目录项既可以指示是否其子树保有某块的有效副本，又能指示是否其子树中分配的块的有效副本在本子树之外存在。在扁平方

案中，副本信息的存放方法有很大的不同。在最高层，扁平方案可以被分为两类：基于存储器的方案和基于高速缓存的方案。基于存储器的方案把关于所有高速缓存副本的目录信息都存放在块的宿主节点上，前面介绍过的基于位向量的方案是基于存储器的：所有有效副本的位置可以从宿主节点得到，并可通过点对点的消息进行通信。在基于高速缓存的方案中，关于高速缓存副本的信息并不全在宿主节点中，而是和副本本身一起分布。宿主节点只有一个指向块的某一高速缓存副本的指针，每个高速缓存副本都有一个指向同一块下一个副本的指针（或标记），形成了一种分布式的链表结构。所需副本位置便可以通过网络事务遍历这个链表得到。

图 8-7 总结了分类体系。层次式目录有一些潜在的优势，例如，当读扑空发生时，如果块的宿主节点在互连网络拓扑中相距比较远，当上下遍历层次的时候发现相距较近的另一节点有块的有效副本，则可以直接从这一节点获得数据而不用去访问宿主节点。此外，从几个节点发出的请求有可能在一个共同的祖先那里被合并为一个，再向上就只发送一个请求。这些优势的发挥取决于逻辑的层次结构同支撑的网络物理拓扑结构的匹配程度。但是，同扁平目录方案发生扑空时只需要少量的点对点网络事务的情况不同，层次式目录方案需要更多的网络事务来上下遍历各层次，这比距离更容易影响系统的性能。（因为端点启动和处理网络事务的代价主导了每跳的代价。）而且，每一次事务都需要查询（或者修改）目的节点上的目录信息，这使得事务的代价更高。因此，层次式目录方案的时延和带宽等特征比扁平方案要差得多，在现代系统中并不常用。所以，本章对层次式目录不做太多讨论，只是在 8.10.2 节和多级侦听方法一起简要提及。本节其余部分将研究基于存储器和基于高速缓存两种类型的扁平目录方案，观察目录的组织方式、存储开销、协议的结构和对性能特点的影响。

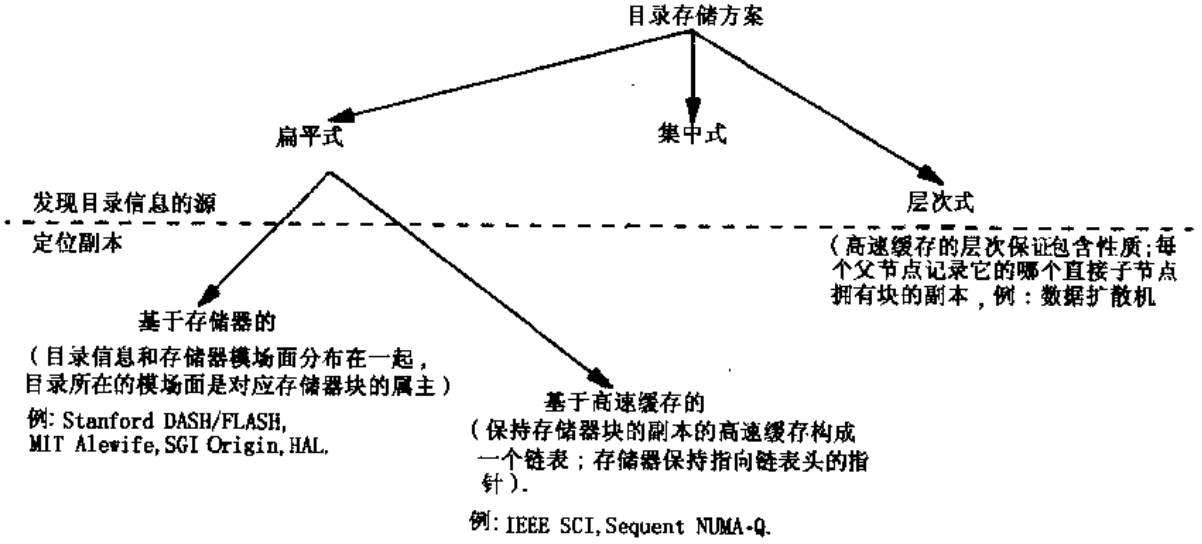


图 8-7　存储目录信息的几种方法。这个两级分类法基于目录信息源和副本本身的定位方法。在层次式目录方案中，这两个功能是由同一种机制来实现的

1．扁平的、基于存储器的目录方案

前面介绍的位向量结构又叫全位向量组织结构，是扁平的、基于存储器的目录方案中存储目录信息最直接了当的方式。所产生的协议的风格前面已经讨论过。考虑一下它执行写的基本性能特征，由于它把关于共享者的所有信息都精确地保存在宿主节点，于是每次作废写

的网络事务的数量仅随实际的共享者的数目增长。所有共享者的标识都可以在宿主节点得到，于是向它们发出的作废请求可以被重叠，甚至并行发送；所以在关键路径上完全串行的网络事务数量并不与共享者的数目成正比，这就减少了时延。

正如前面谈到的那样，全位向量结构的主要缺点就是它的存储开销太大。保持这种结构不变，如果处理器的数目是给定的，我们有两种方法可以降低存储的开销。第一种方法是增加高速缓存块的大小，另一种方法是在节点使用多个处理器，而不仅仅是一个处理器，减少目录协议可见节点的数目，即，使用两级协议。例如，Stanford 的 DASH 机使用的就是全位向量结构，它的节点是有四个处理器的基于总线的多处理器。这两种方法实际上使得全位向量目录结构对规模相当大的处理机都很有吸引力：当每个节点有四个处理器，高速缓存块为128字节的时候，一个有256个处理器的机器的目录存储器开销只有6.25%。随着节点越来越多地采用小规模的多处理器，目录存储开销的问题将不会很严重。

566 ~ 567

不过，以上两种方法都只是在较小程度上降低了存储开销，整个目录的存储开销仍然与 $P \times M$ 成正比，这里 P 表示处理节点的数目，M 表示机器中存储器块的总数（$M = P \times m$，m 是每个局部存储器的块数），在规模非常大的机器中，这种开销是不能容忍的。针对 $P \times M$ 这个表达式中每一个因子采取措施都有可能进一步减小这种开销，例如我们可以通过使目录项所需的位数（又叫目录宽度）不与 P 成正比而减少它，也可以通过几个块共用一个目录项来减少目录项的总数（又叫目录高度）。

我们可以用一种叫做有限指针目录的结构来降低目录宽度。在一个块被写的时候，大多数情况下只有少数节点有该块的副本，因此有限指针方案并不存储所有节点的信息，而是只维护一定数量的指针（比如说 i），每个指针都指向一个当前具有块副本的节点（Agarwal et al. 1988）。每个指针都需要 $\log_2 P$（P 为节点数）的存储空间，但指针数量少。例如一个有1 024个节点的机器，每个指针都需要10位，即使使用100个指针也比全位向量方案的存储开销要少。在实际应用中，5个或再少一些的指针已经基本够用。当然了，协议必须提供某种后备和处理“溢出”的策略，以便应付超过 i 个高速缓存副本的情况，因为它只能对 i 个节点进行精确地记录。一种策略就是当副本数超过 i 时，对所有的节点进行作废广播。此外还有很多其他的策略，即使在“溢出”的情况下也避免用广播。不同的有限目录方案的差异主要就在于它们“溢出”处理策略的不同和采用的指针个数不同。

由于机器中高速缓存的容量要远远小于主存储器，所以在某一时刻只有很少一部分存储器块会在高速缓存中有副本，于是大多数的目录项都不会被用到（Gupta, Weber, and Mowry 1990; O'Krafka and Newton 1990）。我们可以利用这一点，把目录本身组织成高速缓存，这样就可以减少目录的高度。8.10节将更详细地讨论减少目录宽度和高度的技术。

除了这些减少存储开销的优化策略外，不同的扁平的、基于存储器的目录方案寻找副本并与之通信（协议功能[2]和[3]）的基本方法都是一样的。即共享者的标志都由宿主节点维护（至少在没有溢出的情况下），与副本的通信是通过发送点对点的网络事务完成的。

2. 扁平的、基于高速缓存的目录方案

在扁平的、基于高速缓存的目录方案中，块仍然有它的宿主主存；但是，宿主节点的目录项不包含所有共享者的标识，而是仅仅含有指向表中的第一个共享者的指针和几个状态位。这个指针叫做这个块的头指针。含有该块高速缓存副本的其他节点通过一个分布式的、双向的链表结合在一起（使用与节点每个高速缓存行相联系的额外指针），即一个含有块副

568

本的高速缓存包含了指向前一个和后一个含有副本的高速缓存的指针，分别叫做前向指针和后向指针（见图 8-8）。

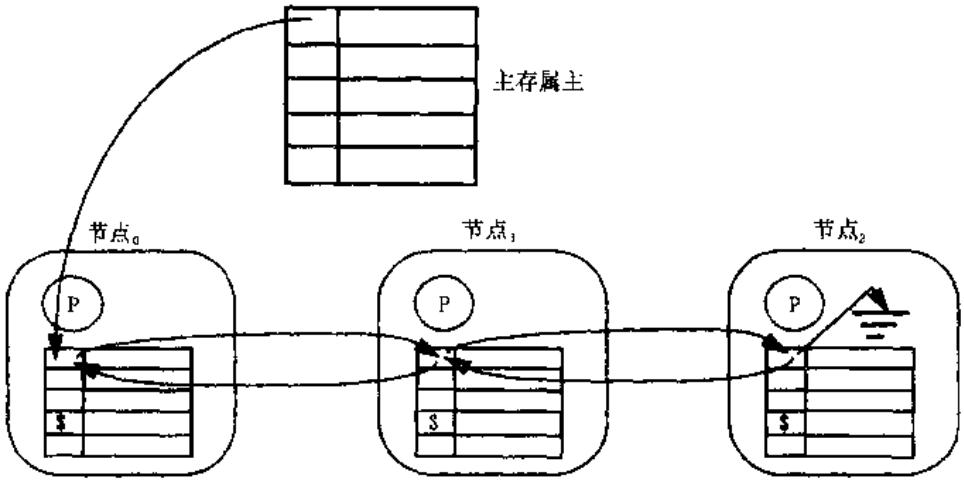


图 8-8 一个双向链表分布式目录结构。高速缓存的一行不仅含有数据和状态信息，而且还有构造分布式链表用的前向和后向指针

发生读扑空时，请求的节点向宿主存储器发出网络事务，以获得该块链表头节点的标识（如果有的话）。如果头指针为空（当前没有共享者），宿主节点直接返回数据，如果头指针不为空，必须把请求者加入到共享者的链表中去。宿主节点向请求者返回头指针。请求者向头节点发送一个消息，请求把自己加入到链表的头部从而成为新的头节点，其效果是使宿主节点的头指针现在指向请求者节点，请求者自己的高速缓存项的前向指针指向老的头节点（老的头节点现在成了链表中的第二个节点），老的头节点的后向指针指向请求者。如块的最新副本在宿主节点，数据就由宿主节点直接提供给请求节点，否则就由头节点提供（头节点作为拥有者总是有最新副本）。

发生写扑空时，写入者仍是先从宿主节点获得头节点的标识（如果有的话），然后像前面那样，把自己插入到链表的头部。（如果写入者已经作为共享者存在于链表中，而现在它在执行一个升级操作，那么先把它从当前位置删去，然后再作为新的头节点插入表中。）接着，通过网络事务逐节点遍历分布链表的其余部分，找到并作废该块的后续副本。如果被写入的块由三个节点 A、B、C 共享，宿主节点只知道 A 节点，那么写入者便向 A 发出作废消息；到达 A 后才能知道另外一个共享者 B 的标识，依次类推。对这些作废的确认都被送回写入者节点。如果写入者需要块中的数据，它可以同发生读扑空时一样从宿主节点或头节点获得。每次作废写引起的消息数量，即带宽需求，与共享者的数目成正比，这与基于存储器的方案相同。不同的是现在处于关键路径上的消息量（也就是说时延）也与共享者数成正比，每一个串行化的消息都是在到达时才激活目的地的通信辅助部件，进一步增加了时延和通信辅助部件的占用度。事实上，即使对一个干净块的读扑空也要涉及三个节点的通信辅助部件，以便把节点加入到链表中。

569

从高速缓存中把数据回写或进行数据块替换也需要节点把自己从对应的链表中删除，这需要节点同链表中相邻的两个节点进行通信。之所以需要这个动作是因为替换旧块的新块需要用高速缓存项的前向指针和后向指针建立自己的链表。我们需要有一定的同步机制，以避

免同时替换掉链表中相邻的节点，另外多个节点的参与也增加了通信辅助部件的占用率。在8.6将更加深入地描述一个基于高速缓存的协议的例子。

虽然在时延和占用度方面有些不尽如人意的地方，但基于高速缓存的方案同基于存储器的方案相比有着一些很重要的优势。第一，目录的开销很小。存储器中的每一个块都附加一个头指针和少数几个状态位，前向和后向指针的数目与机器中高速缓存块的数目成正比，而高速缓存块的数目比存储器块的数目要小的多。第二，链表记录下了各个节点对存储器中该块访问的顺序，这使得协议可以具有更好的公平性并可避免活锁。（后面将会看到，大多数的基于存储器的方案并不记录对存储器的访问顺序）。第三，发送作废时辅助部件所做的工作不是集中在宿主节点，而是分布在各个共享者节点，这样有可能分散对辅助部件的占用率，并降低对特别忙的宿主节点辅助部件的带宽需求。

对分布式链表进行插入或删除操作会导致非常复杂的协议实现。例如，从链表中删除一个节点需要在链表中相邻的处理器之间进行很小心的协调和互斥，以防这些处理器也试图同时替换同一数据块。这些复杂的问题随着基于高速缓存目录协议的标准的形式化和出版而在很大程度上得到了缓解，这个标准就是 IEEE 1596－1992 可扩展一致接口（SCI）标准（Gustavson 1992）。标准里有协议的完整说明和 C 语言源代码。有几个商业性的处理机都采用了这个标准（比如，Sequent NUMA-Q［Lovett and Clapp 1996］、Convex Exemplar［Convex Computer Corporation 1993；Thekkath et al. 1997］、Data General［Clark and Alnes 1996］），另外，采用其他链表表示方法（如使用单链表，而不是 SCI 中的双向链表）的一些变型也已经被开发（Thapar and Delagi 1990）。在 8.6 节将对 SCI 协议本身进行详细考察并讨论它的优缺点。

570

3．对不同目录组织方式的总结

总而言之，有很多方法来安排目录对主存块的高速缓存状态的存储。简单的位向量表示法适用于目录协议中可见的节点不是很多的情况。对于规模比较大的机器，有很多方法可以用来降低存储开销，然而，所选择的组织方式确实影响一致性协议的复杂性以及各种共享模式下目录方案的性能。层次式目录方案在实际的机器中并不多见，而具有扁平的、基于存储器和基于高速缓存（链表）目录方案的机器却已经被建造并使用好多年。

8.3 节将对并行程序的行为及其对基于目录方法隐含的意义，以及一些重要的协议和体系结构上的折中进行量化评估。

8.3　目录协议和折中的评价

本节将同第 5 章一样，用一个模拟器考察应用程序的某些相关特征，这些特征可以反映体系结构上的折中，但是不能通过在实际的机器上的测量得到。至于是采用三状态还是四状态的协议、协议是基于作废还是基于更新这样的问题，在第 5 章已经讨论过，在此不再重述。本节将着重讨论基于作废的协议，因为在可扩展处理机中更新协议还有另外一个缺点：更新不能再通过被所有高速缓存所侦听的单个总线事务完成，而必须对每个目的地产生独立的网络事务。此外，基于更新的协议使得在基于目录的系统中，维持所需的存储器同一性模型更加困难。本节将对目录协议中作废模式的分布进行量化分析，考察在问题规模不变情况下，本地和远程节点间的流量分布如何随处理器数目的增加而变化，并再次考察高速缓存块的尺寸对流量的影响。在所有情况下，我们的实验假定使用基于存储器的扁平目录协议。相对于第 5 章的实验，这里有两个变化。首先，由于基数排序在处理器较多时会表现出大量的

伪共享（我们这里的缺省是 32 个处理器，而不是 16 个），我们用的问题规模为 1 M 个键，而不是 256 K 个。其次，在我们所有的小缓存实验中，我们用的是 8-KB 高速缓存，而不是 64-KB，从而可以考察更小的工作集和高速缓存相适应的效果。

8.3.1 目录方案的数据共享模式

前面讲过写操作时需要发送的作废的数量通常都比较小，这使得目录特别有用，并能在不影响性能的情况下大大减少其存储开销。下面将用一些并行应用程序实例来对此进行量化研究。这将发展成一种按共享模式对数据结构进行分类并理解作废模式是如何扩展的框架，并进而根据这个框架对应用案例分析的行为进行解释。为简单起见，所模拟的协议假设只有三种基本的高速缓存状态（MSI）。 571

1. *应用案例分析的共享模式*

对基于作废的目录方案来说，理解应用程序数据共享模式的如下两个方面是很重要的：1）处理器发出需要作废其他副本的写操作（即在一个 MSI 协议中，对写入者本地高速缓存中一个处于非修改态的数据的写，或称作废写）的频率，称为*作废频率*；2）发生这样的写时需要发出的作废（共享者）的数量分布，称为*作废规模分布*。当平均作废规模小而且作废频率足够大，使得总是使用广播确实影响性能时，目录方案的优势就特别明显。图 8-9 显示了案例分析的并行应用，用第 4 章出现的缺省问题规模，在一个 64 节点的系统（每节点一个处理器）上运行时作废规模的分布。为了捕捉固有的共享模型，这些模拟假设每个处理器具有无限大的高速缓存。若使用有限的高速缓存，发送给目录的替换提示会复位存在位，从而在某些情况下减少了写时发送的作废的数量（尽管流量不会减少，因为必须发送替换提示）。在 MSI 协议中，如果块处于共享状态并且没有其他的共享副本，则对它写时就不用发出作废。但这种情况在高速缓存无限大的 MESI 协议中不可能发生。当高速缓存无限大时，作废频率同计算与通信的比成正比。

显然，作废规模一般都不大，这既表明目录方案在减少流量方面的确是非常有用的，同时也表明基于存储器的扁平目录没有必要为每一个节点都维护一个存在位。非常大规模的作废规模的频率不等于零一般是由同步变量所造成的，当很多处理器都在一个变量上踏步等待，而有一个处理器对该变量进行了写操作，就要向所有其他的处理器发出作废。我们不仅仅对给定的问题规模和处理器数目的结果感兴趣，也很关心它们是如何扩展的。第 4 章中讨论的计算与通信比对作废写的频率扩展性是很好的说明。对应用以及它们的数据结构的用法的理解有助于理解作废规模的分布（并通过实验得到验证），也有助于解释图 8-9 中观察到的基本结果。

2. *一个共享模式的框架*

应用程序的数据访问模式可以用很多方法分类：可预测的和不可预测的、规则的和不规则的、粗粒度的和细粒度的（或地址连续的和地址不连续的）、邻近的和在互连拓扑结构中大范围的等等。为了理解作废模式，相关的分类有：只读、生产者-消费者、迁移、不规则读-写。（一种类似的分类法可以在［Gupta 和 Weber 1992］中找到。）

- *只读*。只读的数据结构一旦被初始化后就不会被再写，不存在作废写操作，所以这类数据对目录不是问题。其实例包括程序代码和 Raytrace 应用程序中的场景数据。 572
- *生产者－消费者*。一个处理器产生（写）一个数据项，然后另一些处理器消费（读）

它，接着又有一个处理器产生它，以次类推。基于标志的同步操作和迭代网格计算中的近邻共享等都是具体的例子。进行写操作的进程可以每次都是同一个，也可以有所不同；例如，在一个分支界限算法中，不同的进程在发现了改进的界限时可以对界限变量写入。这类数据的作废规模取决于每次生产者写入值时已经有了多少个消费者，可以是一个或几个，甚至系统中所有的进程都是消费者。尽管对大多数应用程序来说，作废规模不随处理器数目的增加而迅速增大，作废频率低[㊀]，但这些场合下，还是会有不同的频率和扩展特性。

- 迁移。迁移的数据不断地从一个处理器迁移到另外一个处理器，并被每一个它跳往的处理器写入（经常是读）。一个例子是全局求和，不同的进程都要把自己的部分和加到全局和上面去。每当一个处理器写一个变量时，只有前一个写入者拥有一个副本（因为它在执行自己的写时已经作废了以前的拥有者），所以不管使用了多少处理器，每次写时只需要产生一个作废。
- 不规则读写。这对应于不同进程对数据的不规则或不可预测的数据访问模式。一个简单的例子是分布式任务队列系统。进程在寻找可窃取的任务时要探测（读）任务队列的头指针，在队列头部增加新任务时要写头指针。这种不规则的访问方式常常导致变化很大的作废规模，不过在大多数被观察的应用中，频率通常集中在频谱的一端（见图 8-9 中的 Radiosity 例子）。

573

3. 将上述框架应用于程序案例分析

现在我们简单地看一下图 8-9 中的每个应用，解释依据上述四种共享模式得到的结果，理解作废规模的分布是如何变化的。

在 LU 因子分解程序中，当一个块被写时，只有对它进行写操作的同一个处理器（被分配了该块的进程）在此之前读过它。这意味着没有其他的处理器会拥有该块的高速缓存副本，于是就不需要发送作废。一旦写操作完成，该块就会被其他一些处理器读而不会被再写。我们之所以从图中看到作废规模为 1，是因为矩阵是由单个进程初始化的，由于使用了无限大的高速缓存，它的高速缓存中就有整个矩阵的副本；当另一个处理器第一次写某块时，必须将它作废。还有少数几个作废写操作需要作废所有的进程，那都是由一些全局变量而不是主要的矩阵数据结构引起的。扩展问题的规模或处理器的数量并不改变矩阵作废规模的分布，而只会改变全局变量的作废规模的分布。当然了，作废频率会随着扩展变化，就像计算与通信比一样。

在 Radix 排序内核程序中，两种“生产者-消费者”场合会发出作废。在置换阶段，要写的字或块自从上一次被写之后，只被键字所分配到的进程读过，所以至多需要发送一个作废。目标数组中相同的键字位置可能在排序的不同外层循环迭代中被不同的进程写；但是，每一次迭代中，键字只有一个读出者，所以即使在这种不常发生的情况下，也只需发出两个

㊀ 生产者-消费者的作废规模分布不增长的例子是近邻规则的网格计算中的非角元素和 Radix 中的键字置换。它们产生为 1 的作废规模，且不随处理器的数量或问题的规模增加。所有进程都是消费者（作废规模 $p-1$）的例子之一是全局能量变量，它由所有的进程在物理模拟的一个时间步内读，在时间步结束时由一个进程写入。另一个例子是所有进程绕其自旋的同步变量。尽管这里作废的规模大，幸运的是，在真正的应用中这类写很少发生。最后，几个进程作为消费者的例子包括网格分区的角元素或基于树的同步所使用的标志。产生的作废规模为几个，它可能（也可能不）随处理器的数量增长而扩展（在这两个例子中不增长）。

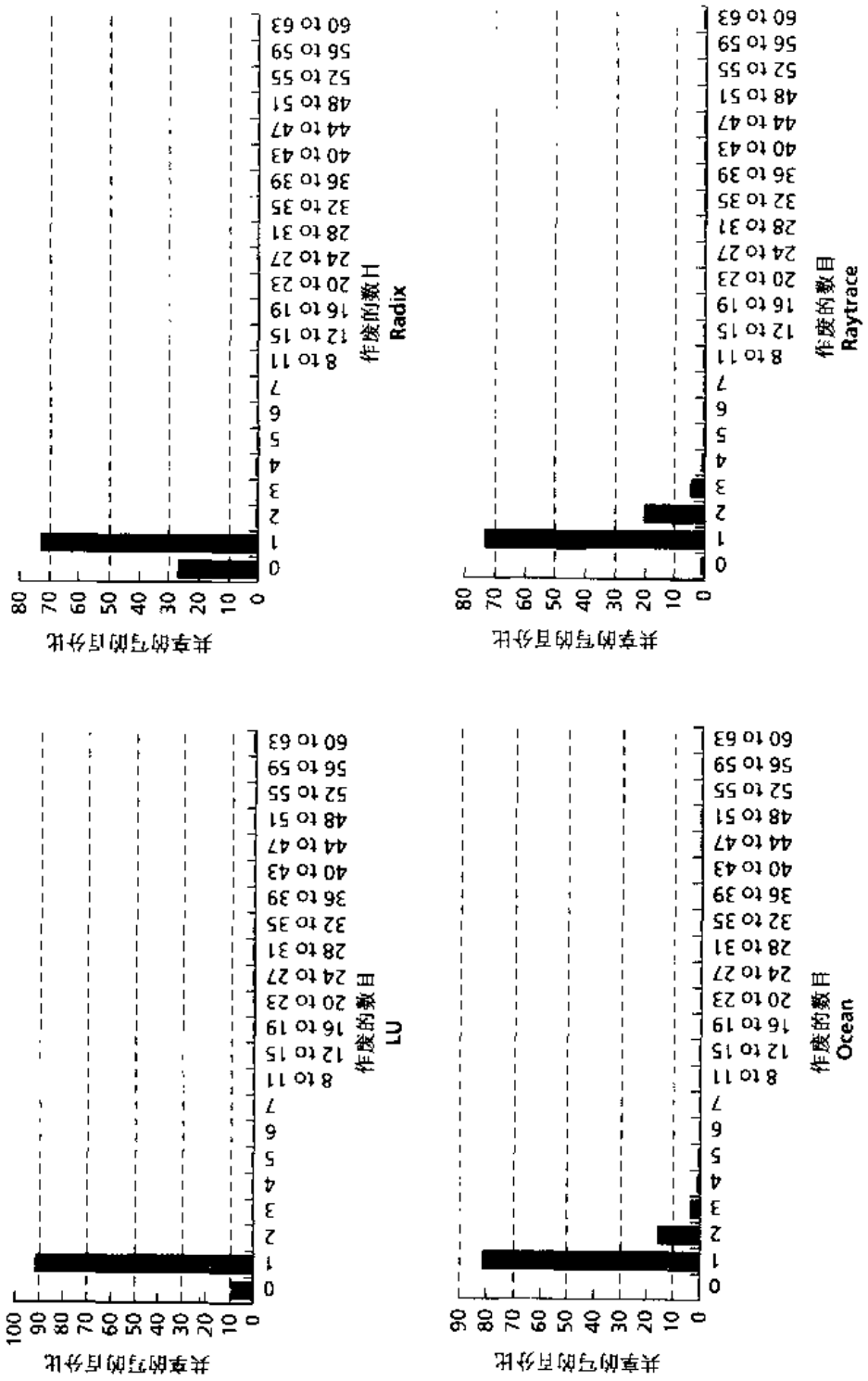
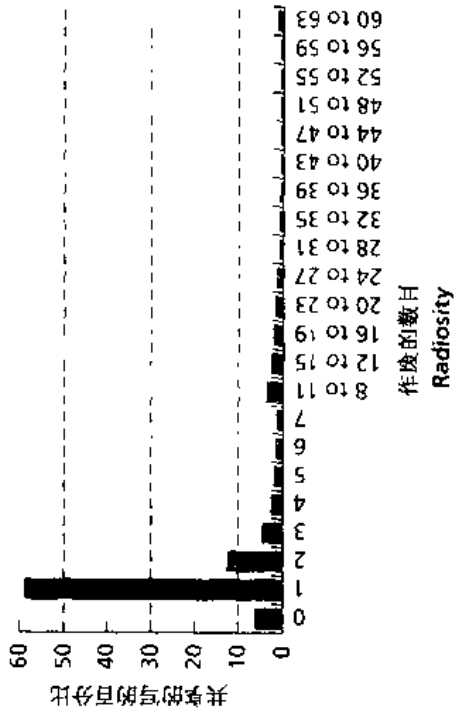


图 8-9　64 个处理器的机器上缺省数据集的作废模式。x 轴显示作废写时的作废规模（活动的共享者的数量），y 轴表示其作废规模取 x 轴值的作废写的百分比。我们模拟一个全位向量目录表示方法，记录每个作废写发生时有多少个存在位（除了写入者外）处于置位状态，由此测量作废规模的分布。数据被平均分配在 64 个的处理器上


共享的写的百分比
0
10
20
30
40
50
60
0
1
2
3
4
5
6
7
8 to 11
12 to 15
16 to 19
20 to 23
24 to 27
28 to 31
32 to 35
36 to 39
40 to 43
44 to 47
48 to 51
52 to 55
56 to 59
60 to 63
作废的数目
Radiosity

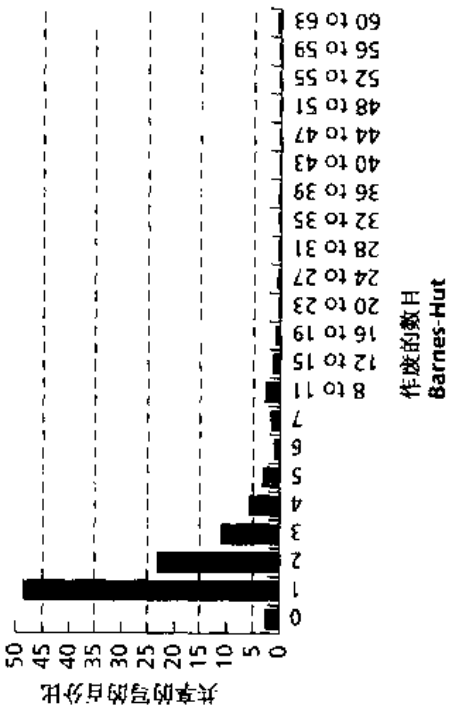

共享的写的百分比
0
5
10
15
20
25
30
35
40
45
50
0
1
2
3
4
5
6
7
8 to 11
12 to 15
16 to 19
20 to 23
24 to 27
28 to 31
32 to 35
36 to 39
40 to 43
44 to 47
48 to 51
52 to 55
56 to 59
60 to 63
作废的数目
Barnes-Hut

图 8-9(续)

作废（一个发给读出者，另一个发给前一个写入者）。如果存在伪共享，所有的共享者都要写同一个块，每次只需要发出一个作废。产生作废的另一种情况是直方图的累加，它是通过一种树形结构的方式进行的，通常每次需要发出少量的作废。很显然，对多个共享者发出作废的情况并不常见。在 Radix 情况下，增大问题的规模不改变两个阶段中的作废规模（尽管可能改变两个阶段中的相对作废频率），而增加处理器的数目会增加作废规模，但也仅限于直方图累加阶段中一些不常用到的部分。占主导地位的作废规模还是 0 和 1。

Ocean 中的规则网格计算产生的近邻“生产者-消费者”通信模式使得在大多数情况下只需要对 0 个或 1 个进程发出作废（在分区边界处）。在多重网格方程求解器的分区边界处，更经常碰到的是作废两个或 3 个共享者的情况。这并不随问题规模或处理器数目的增加而上升。在多重网格层次结构的最高层上，几个处理器的分区的边界元素可能会落到同一个块，导致作废 4 或 5 个共享者的情况。还有一些呈现迁移共享模式（一个作废）的全局累加器变量和少数几个极不常用的单生产者多消费者模式的全局变量（但不是同步变量）。

574 ~ 576

Raytrace 程序中主要的数据结构是场景数据，它是只读的。可读写的数据由图像和任务队列组成。每帧中，图像的每个字仅由一个处理器写一次，如果由同一个处理器负责写下一帧的对应像素（通常是这种情况），就不需要发送作废；如果下一帧由不同的处理器写，就需要发送一个作废。对于任务窃取或存在写-写伪共享时，也是如此。任务队列会导致前面讨论过的不规则的读写访问模式，使得作废规模在一个很大的范围内分布，但大多数作废规模都不大（因此，这种情况下沿 x 轴，只有很少的非零作废）。这里也有一些不常用的单生产者-多消费者的可写全局变量。

在 Barnes-Hut 应用程序中，重要的数据是星体和单元的位置、用来连接树的指针和一些表示能量值的全局变量。位置数据属于“生产者-消费者”类型，一个给定星体的位置在力计算（遍历树）阶段通常被一个或几个处理器读。一个单元（质量中心）的位置会被很多进程读，越靠近根节点读进程数越多，根节点被所有进程读。在引力计算之后的更新和树构造阶段，这类数据所在的处理器对数据的写入导致相当大范围的作废。根节点和较高层单元负责向所有的处理器发送作废，但这样的机率是很小的。树指针的行为与质心单元类似。在构造树的阶段对一个指针的第一次写，要使在前一个力计算阶段中读该指针的那些处理器的高速缓存副本作废，以后的写操作也要使在构造树阶段读指针的处理器中的副本无效，作废规模不规则但大多数处理器数不大。当处理器数目增加时，由于一个给定的数据可能被更多的处理器读，作废规模的分布右移，但移动很慢并且大部分作废规模还是很小。当星体数目增加时则会观察到相反的效果（也很缓慢）。

最后，Radiosity 应用程序对不同类型的数据，包括场景描述数据（片和单元）和任务队列，采用十分不规则的访问模式。这就导致宽范围的作废模式分布，但即使在这种情况下，出现频率最高的不过是 0～2 个作废。许多对场景数据和少数几个计数器的访问都属于“迁移”类型，还有少数几对单生产者-多消费者的全局变量。

经验数据和分类框架表明，在大多数情况下作废规模都比较小。再加上通常把并行处理机用作多道程序的计算服务器，服务于多个串行或小规模并行的应用程序，这也进一步限制了共享者的数量（进程迁移通常导致作废规模为 1）。经验表明，导致大量作废的共享模式在程序执行时出现的机率很小。一个可能的例外是高速竞争的同步变量，我们将看到，通常

会用软件或硬件对它们进行特殊处理。这些结果除了验证了基于目录方案的有效性和它在性能可扩展方面的潜力外，还表明有限指针的目录表示法会很有效，因为发生溢出的可能性很小。

577

8.3.2 本地和远程通信流量

具有分布式存储器的系统一个主要的特性是因高速缓存扑空引起的通信量的多少，或者说协议动作限制在节点内（本地）还是外向到互连网络上（远程）。对于给定的处理器数目和机器的组织结构，本地流量的比例取决于问题的规模。但是，既使在问题规模固定时，（即在 PC 伸缩下），研究流量及其分布如何随处理器数目变化也是一件很有意义的事。图 8-10显示了对于缺省问题规模研究得到的结果，其中将远程流量分成了几类，例如共享（真共享或伪共享）、容量型扑空、冷启动扑空、回写和额外开销。在 8.3.2 节和 8.3.3 节中我们用的都是 MESI 协议，而非 MSI 协议。额外开销包括网络发送每个数据块时必不可少的数据头和与协议有关的不携带任何数据的流量比如作废和应答。协议流量的成分不同于基于总线的机器，在这里每一个点对点的作废都消耗流量，确认也给互连网络带来流量。对不同的应用，我们用每个浮点运算的字节数或每条指令的字节数来表示流量。

我们可以看出，当处理器数目增加时，本地流量和与容量相关的远程流量趋于降低，这是因为处理器的工作集变小了，由本地而不是远程满足的冷扑空也减少了。但是，与共享相关流量却如我们所预料的增加了。在小工作集的应用中，如 Barnes-Hut、LU 和 Radiosity，容量相关流量的比例非常少。在 Barnes-Hut 和 Raytrace 这样非规则的应用程序中，容量相关的流量大都是远程的，而且处理器数目越多越是这样，因为将数据以页为粒度分布使得容量扑空在本地得到满足并不容易。对于 Ocean 这样的程序，使用大高速缓存，容量相关的流量基本上是均匀分布的，而且如果页面放置恰当（通过 4 维数组结构可以很容易做到），流量几乎全是本地的。但如果共享页按循环方式放置，Ocean 中大多数本地容量型扑空变成远程的。

当我们为了捕捉在 Ocean 和 Raytrace 中的工作集不能容纳于高速缓存的场景而采用了较小的高速缓存时，容量型流量变得比以前多得多。因为数据分布得好，Ocean 中的大多数流量仍然是本地的并且远程流量与处理器数目之间的关系趋势不变。不好的页面分布会使网络被流量所充塞，但恰当的分布使远程流量相当的低。然而，Raytrace 中的容量相关的流量大多是远程的，与共享流量占主导的大高速缓存情况相比，Raytrace 中主要是容量相关的流量，以至于改变了远程流量总量曲线的斜率。远程流量仍然随处理器的数量增多而增长，但要缓慢得多，因为工作集尺寸以及容量型扑空率并不像共享扑空率那样依赖于处理器的数量。

当一次扑空在远程被满足时，数据是从宿主节点获得还是用另一个消息从一个脏节点获得，除了跟扑空类型（是共享扑空，还是容量/冲突/冷启动型扑空）有关外，还和高速缓存的大小有关。高速缓存容量较小时，脏数据可能会被替换并回写，另一个处理器的共享扑空就可能在宿主节点而不是在以前的脏节点被满足。对于像 Ocean 这样的程序，数据可以很容易地放到它所在宿主节点的存储器中（即为了局部性适当地分布），通常只有那个节点对它进行写操作，所以即使数据是“脏”的，它也是在宿主节点本身的高速缓存中。这种情况的适用范围取决于应用的数据访问方式、存储器中数据分配的粒度以及程序是否确实恰当地分布了数据等。

578

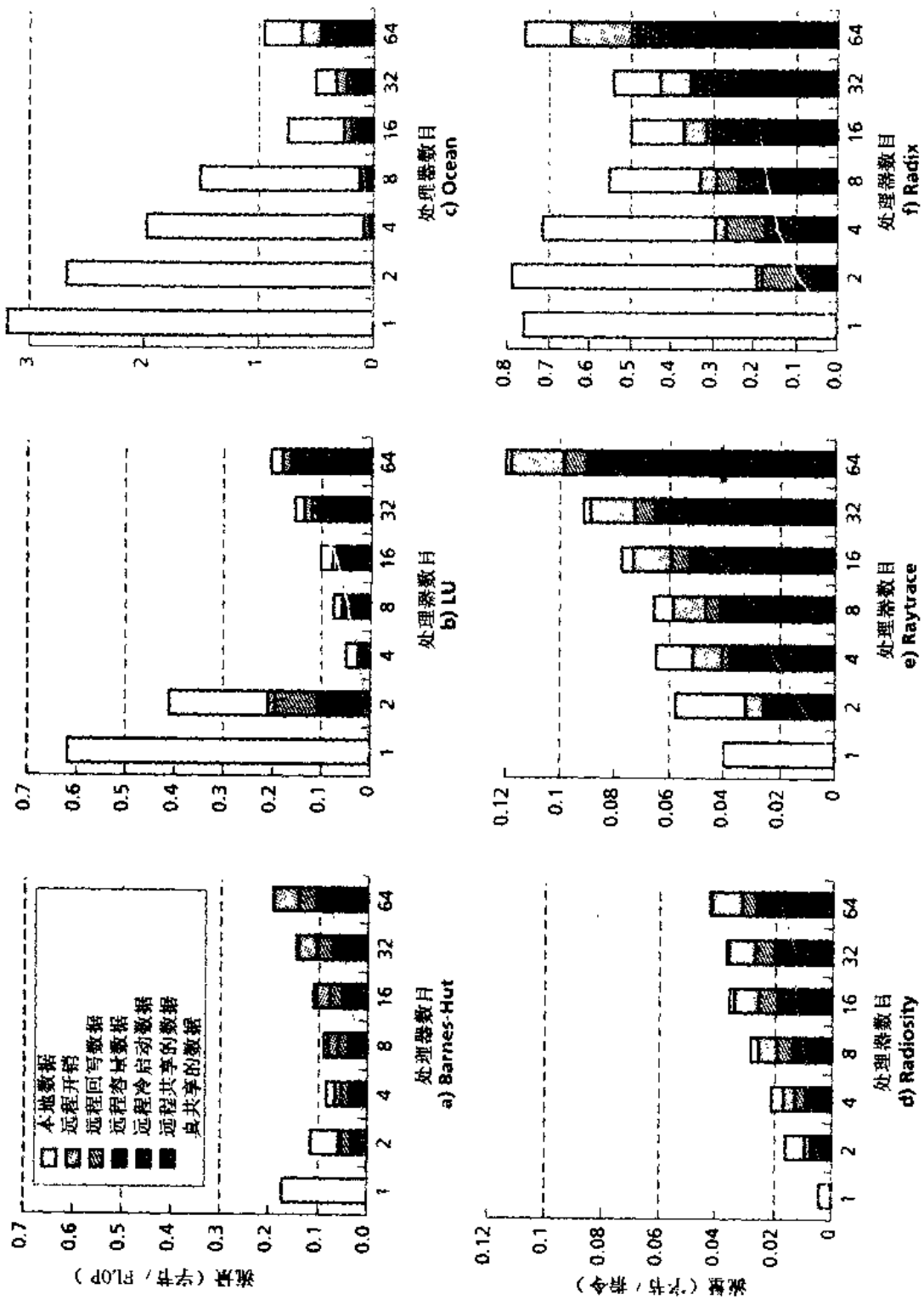


图 8-10　流量与处理器数目的关系。通过网络传输每个块的额外开销是 8 个字节。本地访问没有额外开销。图 a)～f)假定每个处理器有 1 MB 的高速缓存，g)～i)假定每个处理器的高速缓存为 8 KB。高速缓存块尺寸为 64 字节，是 4 路组相联的

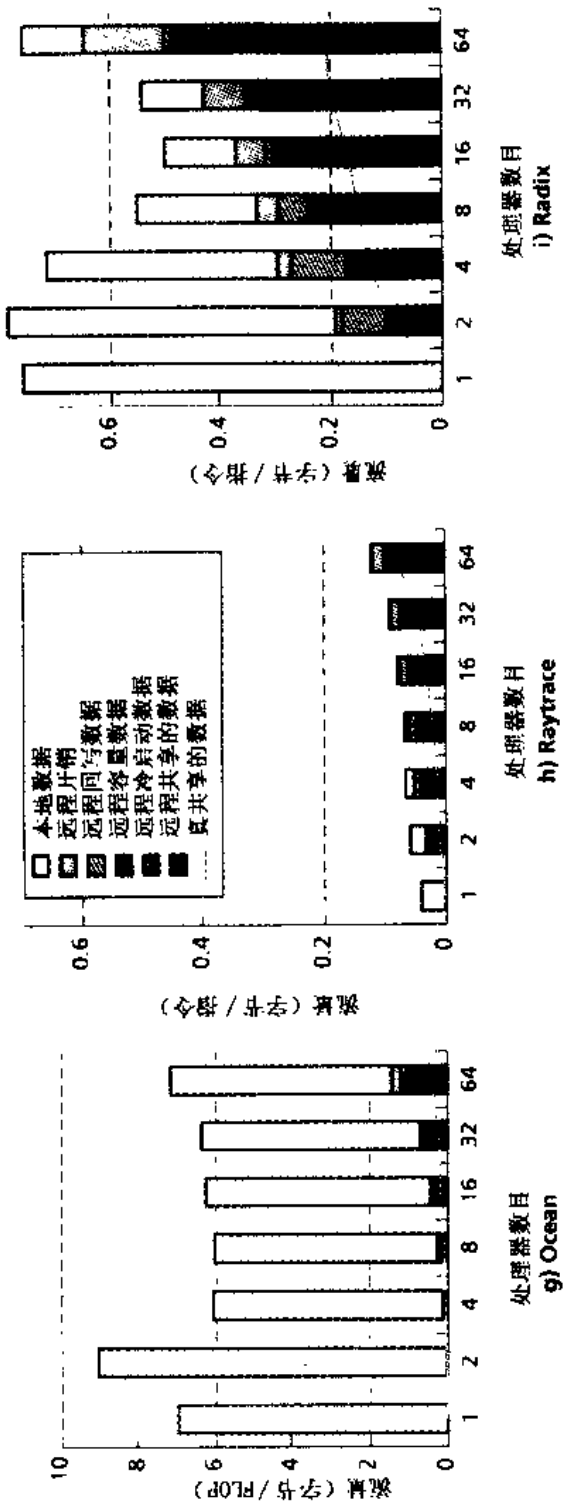

流量（字节/FLOP）
处理器数目
g) Ocean
流量（字节/指令）
本地数据
远程片内
远程同与数据
远程容量数据
远程冷启动数据
远程共享的数据
真共享的数据
处理器数目
h) Raytrace
流量（字节/指令）
处理器数目
i) Radix

图 8-10(续)

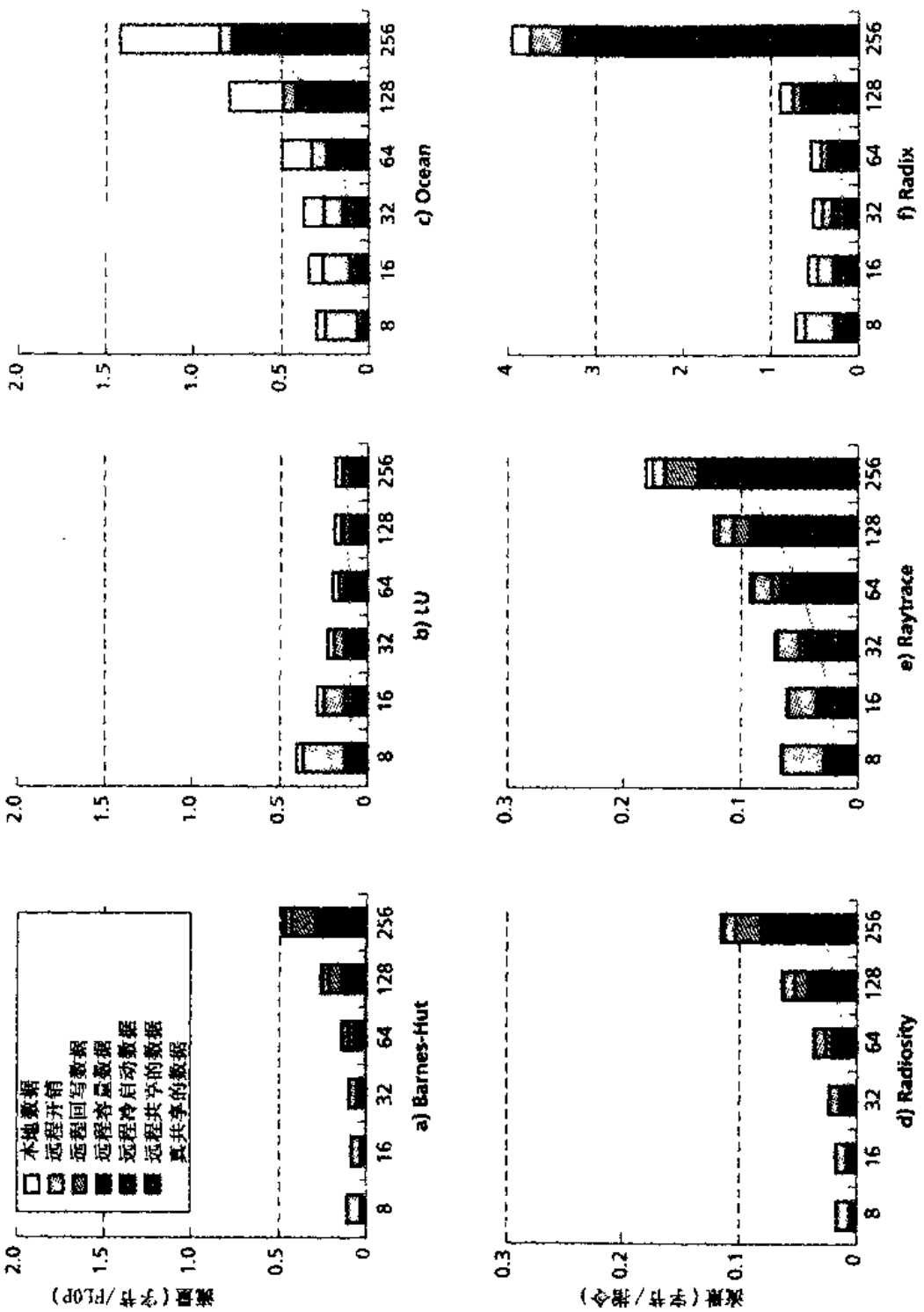


图 8-11　流量与高速缓存块大小的关系。数据为 32 个处理器执行的情况。通过网络传输每个块的额外开销是 8 个字节。本地访问没有额外开销。图 a)～f)假定每个处理器有 1 MB 的高速缓存，g)～i)假定每个处理器的高速缓存为 8 KB。所有的高速缓存都是 4 路组相联的

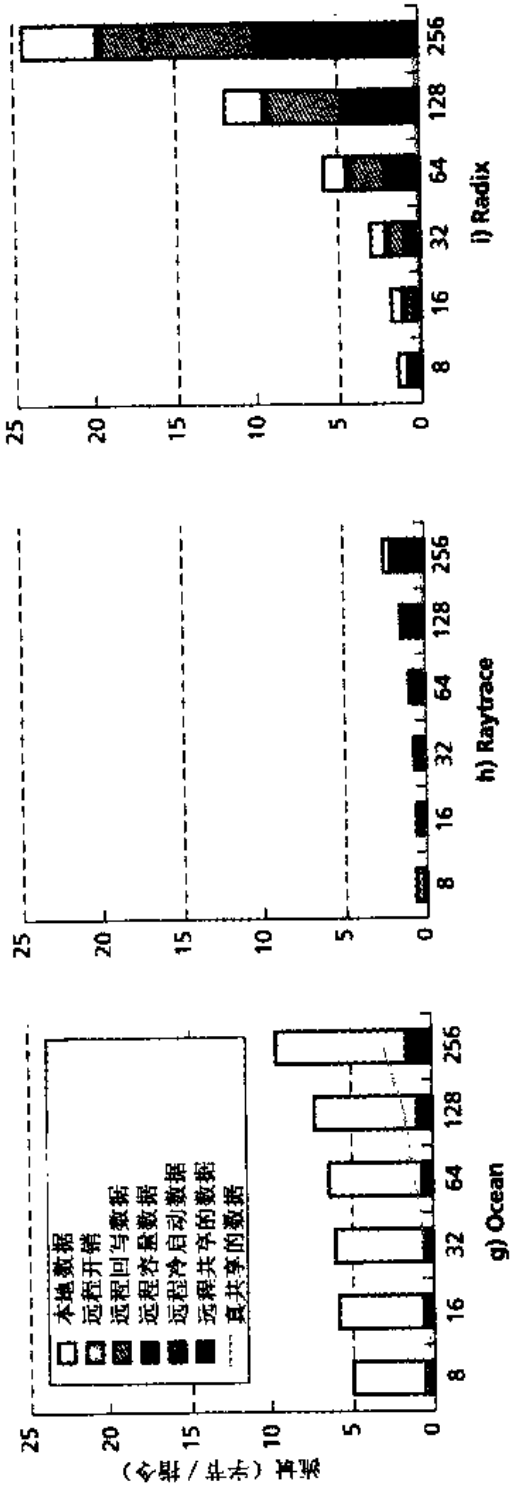

数据（字节/指令）
25
20
15
10
5
0
本地数据
远程开销
远程回写数据
远程容量数据
远程冷启动数据
远程共享的数据
真共享的数据
8
16
32
64
128
256
g) Ocean
h) Raytrace
i) Radix

图 8-11(续)

8.3.3 高速缓存块尺寸的影响

高速缓存块的大小对扑空率和总线流量的影响我们在第 5 章已经讨论过了，至少对那些最多 16 个处理器的机器而言。超过 16 个处理器时扑空率自然增长，第 4 章所讨论的问题规模、处理器数量和块尺寸的相互作用产生的阈值效应是个例外。本节将讨论块尺寸对具有分布式存储器的机器中本地和远程流量成分的影响。

图 8-11 显示了在 32 处理器每个处理器 1 MB 高速缓存的机器上执行应用程序时，流量随块尺寸的变化情况。在 Barnes-Hut 中，总流量在块尺寸较小时随块的变大而缓慢地增加，当块大于 64 字节后，由于伪共享总流量快速增长。但是，总流量比较小，因为通过网络的每个块的额外开销是固定的（作废和确认的代价也是如此），当块尺寸增大到能产生空间局部性（即如果较大的块减少了传输的块的数量）时，额外开销成分会变小。LU 程序有完美的空间局部性，当块增大时数据流量保持不变。额外开销降低，所以随块尺寸的增加，总流量实际上是减少了。在 Raytrace 程序中，远程的容量型流量空间局部性差，所以流量随块的增大而迅速增长。在 Barnes-Hut 和 Raytrace 中，真共享的空间局部性也不好，Ocean 中面向列的分区的边界处也是这样（在面向行的分区边界处，即使远程数据空间局部性也相当好）。最后，Radix 对应的图很清楚地表明，当块尺寸超过一定阈值时（此时大约是 128 字节或 256 字节），伪共享对远程流量的影响明显。使用较小的高速缓存的结果显示，容量型扑空正如所预料的那样起了主导作用。

8.4 目录协议设计上的挑战性问题

设计一个正确高效的目录协议涉及许多比我们已经讨论过的简单的结构更复杂、更精妙的问题，这正像设计一个基于总线的协议要比单纯地选择状态数、为稳定状态画状态转换图要复杂得多一样。我们必须去处理诸如状态转换的非原子性、事务拆分型总线、串行化和排次序、死锁、活锁和饥饿等问题。我们已经理解了目录协议的基本要点，准备好研究这些问题。本节将讨论在正确地实现高性能目录协议时所遇到的新的协议级设计的挑战性问题，认识应付这些挑战的一般性技术。在下两节，我们将结合基于存储器和基于高速缓存的目录协议这两个案例分析，对这些技术进行专门研究。

579 ~ 583

和任何设计一样，可扩展一致性协议设计的挑战性在于，在保证正确性的基础上提供高性能并能包容由此带来的复杂性。我们将依次考察性能和正确性，并将注意力集中于在基于总线和非高速缓存系统中所没有遇到过的问题。因为性能优化常常导致并发性的增加，使正确性问题复杂化，我们就先对它们进行研究。

8.4.1 性能

作为高速缓存一致性协议基础的网络事务与显式消息传递中用到的网络事务有两点不同。第一，它们是由系统，特别是通信辅助部件或控制器，根据协议自动产生的；第二，每个事务本身都很小，携带一个请求、一个确认或是一个数据块加上几个控制位。但是，前几章中提出的网络事务的基本性能模型在这里仍然适用。一个典型的网络事务在它的源端处理器产生一些额外开销（在向外和向内时遍历高速缓存层次结构）；占用端点的通信辅助部件完成某些工作（典型地查找状态，生成请求或介入高速缓存的操作）；由于传输延迟、网络

带宽和竞争而产生网络延迟。一般来说，除了发出请求的处理器外，宿主节点、脏节点、共享节点处的处理器都不直接参与网络事务（虽然其他节点也会由于竞争受到影响）。

用我们在前面介绍过的多处理器系统层次的概念来理解性能是很有用的（见图 8-2）。系统的协议层使用通信抽象提供的网络事务实现编程模型。所以，协议层对单个网络事务的基本通信代价如传输时延、网络带宽、通信辅助部件占用度和处理器额外开销等没有什么调节作用，但它能决定在不同的环境下实现读写这样的存储器操作所需的网络事务的数量和结构。一般来说，有三类技术可以提高性能：1）协议优化，2）机器高层组织结构，3）改善基本通信参数的专用硬件。前两种技术都对通信体系结构假定了一组固定的性能参数，在本节对它们进行讨论。改变基本性能参数的影响将在 8.7 节中考察。

584

1．协议优化

协议级有两个主要的性能指标：第一个是减少每个存储器操作产生的网络事务量，这会相应地减少对网络和通信辅助部件的带宽需求；另一个是减少处理器关键路径上的操作（特别是网络事务）的数量，从而降低非竞争时延。后者可以通过尽可能地重叠存储器操作所需要的事务来实现。在某种程度上，协议设计也可以有助于减少每个事务对端点通信辅助部件的占用，特别当通信辅助部件是可编程的，这既减少了非竞争时延，又减少了端点的争用。流量、时延、占用度等特征不应该随使用的处理节点数量的增加而快速上升，并且应该在热点这样病态条件下表现平稳。

我们已经知道，目录信息的存放方式决定了存储器操作的关键路径上网络事务的数量。例如，基于存储器的协议可以从宿主节点以重叠方式发出作废，而在基于高速缓存的协议中，必须通过网络事务遍历分布式链表才能了解共享者的标识。不过，即使在同一类协议中，也有许多改善性能的方法。

考虑在一个扁平的、基于存储器的目录协议中对一个远程分配块的读扑空，该块在第三个节点（拥有者）中是脏的。图 8-12a 显示了以前讲过的那种严格的请求-响应方案。宿主节点响应请求者的消息中包含了拥有者节点的标识。然后请求者再向拥有者发出请求，拥有者

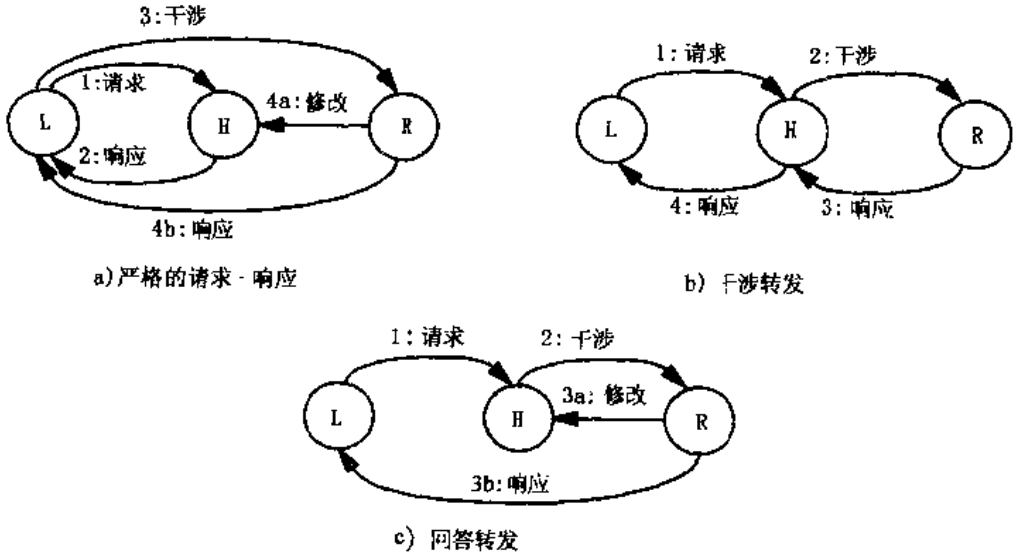


图 8-12　在扁平的、基于存储器的协议中通过转发降低时延。所示例子是一个对处于排他态的块的读请求。L 代表本地节点或请求节点，H 代表块的宿主节点，R 代表拥有块的排他副本的远程拥有者节点

用数据响应（拥有者还向宿主节点发出一个“修正”消息，后者用数据更新存储器并把目录状态置为共享的。）

这种方案读操作时关键路径上有四个网络事务，总共有5个事务。一种减少事务数量的方法叫做*干涉转发*。使用这种方法时，宿主节点并不响应请求者，而是简单地把请求作为一个干涉事务转发给拥有者，要求它从它的高速缓存中取出这个块。干涉和请求相似，只是它是作为对请求的反应而发出，并且它指向高速缓存而不是存储器（在这个意义上，它与作废相似，只是还要从高速缓存取得数据）。拥有者会把数据或确认（如果数据块处在排他态而不是修改态）发回给宿主节点，然后，宿主节点更改其目录状态并用数据应答请求者（见图8-12b）。干涉转发将事务总数降为4个，降低了带宽需求，但所有4个事务仍然都在关键路径上。一个更有进取性的方法是*应答转发*（见图8-12c）。宿主节点也把干涉消息转发到拥有者节点，但是该干涉消息包含请求者的标识，拥有者直接用数据应答请求者。拥有者也向宿主节点发出修正消息，使存储器和目录得到更新，但这个消息并不在读扑空的关键路径上。这种方法保持事务总数为4个，但将关键路径上的事务数减少为3个（请求→干涉→对请求者的应答），所以，它被称为*3－消息扑空*。注意，采用了干涉转发和应答转发，协议不再是严格的请求-响应协议了，因为对宿主节点的请求产生了另外一个对拥有者节点的请求，拥有者接着生成一个响应。后面将会看到，这使死锁的避免更加复杂。

除了时延和流量特征仅仅居中之外，干涉转发还有一个缺点：由于对干涉的响应是发给宿主节点的，所以必须由宿主而不是请求者来记录未决的干涉请求。因为引起干涉的请求可能来自任一节点（假设有 P 个节点），宿主节点就必须能够同时跟踪 $k*P$ 个未决的干涉，这里 k 是每个节点允许的未决请求的数目。而请求者只需跟踪最多 k 个未决的干涉。应答转发不需要宿主节点对未决请求进行跟踪，因而具有更好的性能特性，所以系统都喜欢采用这种方法。在基于高速缓存的方案中也可以使用类似的转发技术来降低时延，不过都失去了严格的“请求-应答”的简单性（见图8-13）。

除转发法外，还有其他一些降低时延的协议优化方法，包括通过推测执行将事务和活动重叠。例如，当请求到达宿主节点时，通信辅助部件可以一边从存储器读数据，一边查看目录，因为大多数情况下数据在宿主节点都处于“干净”的状态。如果目录显示数据的有效副本在其他节点，那么从存储器的访问就浪费了，必须被忽略。最后，协议可能会自动地发现标准的基于作废的协议不能理想地适应常用共享模式并在运行时对自己进行调节，以便更好地适应这些模式（见习题8.9和习题8.10）。

585～586

2．机器的高层组织结构

机器的组织结构也可以同协议相辅相成改进性能。例如在一个节点内部采用大的三级缓存可以减少由于人为通信产生的协议事务量。对于固定的处理器总数量，在两级组织结构中使用多处理器而非单处理器也是很有用的。

两级组织结构在成本和性能方面都有潜在的优势。在成本方面，每节点固定开销可以由节点内各个处理器分摊，并且我们有可能使用现成的SMP系统。在性能方面，优势来自降低涉及目录协议、产生跨节点网络事务的访问数目的共享特性。如果一个处理器把一块数据放入它的高速缓存，同一节点中另一个处理器在访问此块（对于同一个字或不同的字）扑空时，就可以利用高速缓存对高速缓存共享的本地协议，更快地得到满足；如果该块是远程分配的话，效果尤其明显。多个请求还可能被合并：如果一个处理器对某块的请求尚待完成，

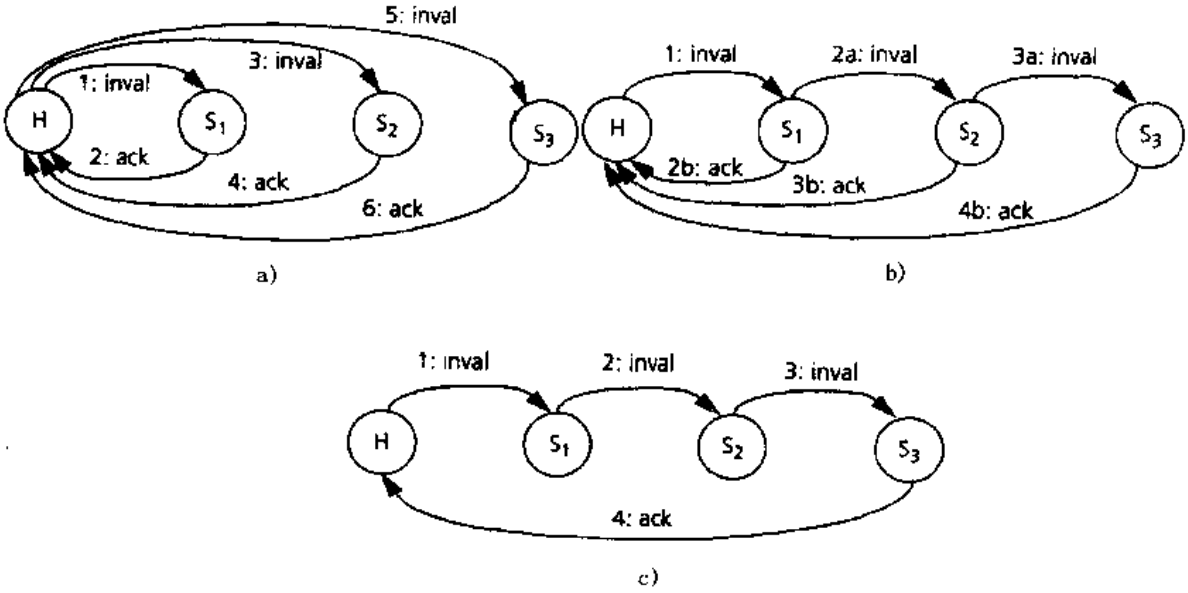


图 8-13 在扁平的、基于高速缓存的协议中降低时延。在这个写操作场景中，宿主节点 H 向共享节点 S_i 发出作废。a）是严格的请求－响应方式，每个节点在其确认（响应）中包含链表中下一个共享者的标识，宿主节点向那个共享者发出作废，在这个作废动作序列中的事务总数为 $2s$，这里 s 为共享者的数量，所有事务均在关键路径上。b）中每一个节点都把作废转发给下一个共享者，同时把确认发给宿主节点，事务总数仍然是 $2s$，但只有 $s+1$ 次在关键路径上。c）中只有链表中最后一个共享者向宿主节点发出一个确认，告诉宿主节点操作序列已经完成。事务总数为 $s+1$。b）和 c 都不是严格的请求－响应方式

587

在同一个SMP节点中的另一个节点对此块的请求就可以被合并，从对第一个处理器的应答中得到所需数据，这就减少了时延、网络事务和潜在的热点竞争。这些优点与全层次结构方法和共享高速缓存的优点类似。事实上，在 SMP 中，各处理器甚至可以在某个层次上共享高速缓存，在这种情况下，第 6 章讨论的共享高速缓存的折中方法都适用。节点数少了，本地的主存储器也增大了。最后，成本和性能特性还可以用一种适当的层次封装技术进一步改善。

当然，两级共享层次结构的优势能在多大程度上被利用取决于下列一些因素：共享的局部性、应用的数据访问模式、在层次结构中进程与处理器的映射以及节点内通信和节点间通信代价的差异等。例如，在一段计算过程中存在很多物理上本地只读共享的应用，比如 Barnes-Hut 星系模拟，如果开始时扑空率很高，就可以从高速缓存到高速缓存的共享中获得显著的好处。像 Ocean 那样呈现近邻共享的应用程序，如果进程和节点有较好的映射关系，那么它们大部分的访问都可以在多处理器节点内部得到满足。然而，尽管某些进程的所有访问都可以在其所在节点内部满足，但另外一些进程却具有沿至少一个边界的访问，必须远程才能满足，这就造成了负载的不均衡，以至于失去层次结构的好处（性能将受到远程通信最多的那个处理器的限制）。在全部对全部的互联通信模型中，固有通信的减少是很有限的。在一个有 p 个处理器的系统中，一个处理器现在需要同 $k-1$ 个本地处理器和 $p-k$ 个远地处理器进行通信，而不是与 $p-1$ 个远地处理器进行通信（k 是每个节点的处理器数），节点间通信最多节省 $(p-k)/(p-1)$。最后值得一提的是，几个进程共享一个主存部件，使得以页粒度在处理器之间恰当地分布数据变得更加容易。对这些权衡和程序特性已有定量的研究（Weber 1993; Erlichson et al. 1995）。在我们的两台案例分析的机器中，Sequent NUMA-Q

机的节点使用的是基于总线的、高速缓存一致的、有 4 个处理器的 SMP 系统；SGI 的 Origin 机则采用了一种有趣的方案：两个处理器为了分摊成本而共享一条总线和一个存储器（及一块板），但它们并没有用总线侦听协议来保持一致性，而是由一个单一的目录协议来保持机器中所有高速缓存的一致。

与使用单处理器节点相比，使用多处理器节点主要的潜在缺点是一个节点内部的通信资源被多个处理器共享。当多个处理器共享一条总线、一个通信辅助部件或者一个网络接口时，它们一方面分摊了成本，另一个方面也不得不相互竞争带宽。如果它们对带宽的需求没有因通信模式的本地性而减少很多的话，这种竞争就会影响到性能。一个解决的办法就是当节点中新增处理器时也相应地增加资源的吞吐量，但这又牺牲了成本上的优势。在一个节点中共享一条总线有一些特别的缺点。首先，如果总线必须接纳几个处理器的话，它就会变长以至于在一块底板或其他的封装单元中放不下，这会使总线速度变慢，从而增加了本地或远程数据的时延。其次，如果总线在节点内支持侦听一致性协议，那么一个必须在远程得到满足的请求就必须等待本地侦听的结果，然后才能发往网络，就会引起不必要的延迟。第三，如果在远程节点上也采用侦听总线，许多确实要去远端的访问将要求本地总线和远程总线上的侦听和数据传输，就会增加时延，降低有效的数据访问带宽。最后，侦听要访问第二级高速缓存的标志，如果侦听不能经常成功实现高速缓存之间的共享的话，就可能导致与处理器的访问之间不必要的竞争。尽管如此，还是有一些基于目录的系统采用了基于侦听一致性协议的多处理器节点（Lenoski et al. 1993; Lovett and Clapp 1996; Clark and Alnes 1996; Weber et al. 1997）。

588

改进协议性能的最后一个方法，即改善通信体系结构的性能参数，将在 8.7 节讨论。

8.4.2 正确性

同基于侦听的系统类似，正确性的考虑可以划分为三类。第一，协议必须能够保证在需要时，相关的数据块能被作废/更新和读取，并发生必要的状态转换。可以假定在所有情况下这种情况总是发生，不对它作进一步的考虑。第二，一致性和同一性模型所定义的串行和次序关系必须被保持。第三，协议及其实现必须避免死锁和活锁，并尽可能避免饥饿的发生。可扩展协议和系统的某些方面使后两类问题的复杂性超出了我们已经了解的基于总线的高速缓存一致的机器和可扩展非一致的机器的情况。存在两个基本的问题。首先，现在一个块可以有多个高速缓存副本，但是不存在一个能看到所有相关事务并对它们进行串行化的部件。其次，由于有大量的处理器，可能同时有许多请求都发向同一个节点，使得第 7 章中谈到过的输入缓冲的问题更加突出。系统的高时延使这些问题更加恶化，促使我们去挖掘以前提到过的那些协议优化。这些优化允许更多的网络事务同时推进，并且不再保持严格的请求－响应的性质，使正确性变得更加复杂。本小节将描述在正确性的各个方面新出现的主要问题和它们的一般解决方案。在案例分析的协议中采用的特殊解决方案将在后面的几小节中详细讨论。

1. 为了一致性而对存储器单元的串行化

回忆一下一致性的写串行化。一个给定的处理器不仅必须能够从对一个给定单元的所有操作（至少是所有写操作和处理器本身的读操作）中构造出一个串行的次序，而且所有处理器必须看到对给定单元的写操作以相同的次序发生。

589

我们需要的串行化机制是一个实体，它能看到不同的处理器对一个给定单元的存储器操作（即不是完全包含在处理节点里的操作）并确定它们的次序。在一个基于总线的系统中，不同处理器的操作是按照它们的请求出现在总线上的次序进行串行化的。在一个不对共享数据做高速缓存的分布式系统中，完成同一性串行化的就是单元所在的主存储器。例如，写操作到达存储器的次序，也即所有处理器看到的次序，读操作所看到的写入的值取决于读到达存储器的时刻。在具有一致的高速缓存的分布式系统中，宿主存储器也是决定对给定单元操作次序的可能的实体，至少在扁平目录方案中是这样，因为所有相关的操作都要先到达这里。如果宿主自身可以满足所有请求，它就只需按先进先出的次序逐个对它们进行处理，从而确定它们的次序。然而当有多个副本时，一个操作对宿主节点可见并不一定意味着它对所有处理器可见。可以很容易地构造一个场景，使处理器所看到的对一个单元的操作次序与请求到达宿主节点的次序不同，或者使不同处理器看到的操作完成的次序不同。

作为一个简单的例子，让我们考虑基于更新的协议和不保证相同端点之间点对点事务次序的网络。如果对一个共享数据的两个写请求以某种次序到达宿主节点，它们引起的更新可能按不同的次序到达各个副本。另外一个例子是，在基于作废的协议中，假定两个节点同时向一个在脏节点中处于被修改状态的块发出排他读请求。在严格的请求－响应协议中，宿主节点会把脏节点的标志发给发出请求者，由后者再向脏节点发出请求。然而即使在一个保持点对点次序的网络中，如果发出请求的节点不是一个，也无法保证请求到达脏节点的次序与它们到达宿主节点的次序相同。那么在这种情况下由谁来对全局一致的串行化负责呢？当对一个块的多个操作同时进行并需要不同节点提供服务的时候又怎么调度呢？

有好几类解决方案可以用来保证对一个单元操作的串行化，它们中大多数都使用称为*忙状态*或*未决状态*的额外目录状态。一个块的目录状态为忙表明前一个到达宿主的针对此块的请求还正在执行，没有完成。当一个请求到达宿主并发现对应的目录状态为忙，就可以通过以下几种机制之一进行串行化。

- *宿主提供缓冲*。请求可以作为未决请求在宿主处缓冲，直到正在进行的对该块的前一个请求完成，不管前一个请求是被转发给脏节点，还是使用了严格的请求－响应协议（当然，与此同时宿主必须处理对其他块的请求）。这个方法保证按请求到达宿主的先进先出的次序对其服务，但它降低了并发性。要求通知宿主写操作何时完成，或更一般地，宿主对于写操作的介入何时结束。最后，它增加了宿主的输入缓冲溢出的危险性，因为缓冲区需要保存宿主所有块的未决请求。溢出时的一个策略是让输入缓冲溢出到主存，因此，只要有足够的主存，它提供了无限的缓冲，避免了潜在的死锁问题。这个方案被用在 MIT 的 Alewife 原型机中（Agarwal et al. 1995）。
- *请求者提供缓冲区*。未决的请求不是缓冲在宿主，而是由请求者自身缓冲，形成分布式的未决请求的链表。这是基于高速缓存目录方案的自然延伸，方案本身就提供了分布式链表支持。这种方法被用于 SCI 协议（Gustavson 1992；IEEE Computer Society 1993）。使用这种方案，一个节点需要记录的未决请求数比较小，并且完全由本地节点自己去管理。
- *否定回答（NACK）并重试*。当目录状态是忙的时候，宿主向进入的请求发出否定回答（即发往请求者的否定回答），而不是将它缓冲。请求者的通信辅助部件稍后会重新发出请求，请求以被目录接收的次序被串行化（被否定回答的请求并不进入串行

590

化次序)。Origin2000 (Laudon and Lenoski 1997) 中就使用了这种方法。

- 转发给脏节点。如果因为前一个请求已经被转发给了脏节点而使目录状态为忙的话，对同一块的后续请求不在宿主缓冲，也不被否定回答，而是也被转发给脏节点，由此决定它们的串行化。因此，串行化的次序是按请求到达宿主节点的次序（当块在宿主中是干净的）或由它们到达脏节点的次序（当块处于“脏”状态）来确定的。如果在一个请求被转发到脏节点之前，脏节点中的对应块脱离了脏状态（例如，因为回写或前一个转发的请求的处理所致），那么该请求就会被脏节点拒绝并被重试。当重试成功时，请求在宿主或脏节点串行化。这种方法被用于 Stanford 的 DASH 协议 (Lenoski et al. 1990; Lenoski et al. 1993)。

不幸的是，当分布式网络中存在一个块的多个副本时，仅仅识别一个串行化实体是不够的。问题在于宿主节点或串行化代理可能知道（或被告知）它参与的请求处理何时结束，但这并不意味着请求相对于其他节点也已经结束了。对该块的下一个请求的某些事务可能在前一个请求的某些剩余事务之前到达其他节点并执行。在 8.5 节和 8.6 节的案例分析协议中，将会看到具体的例子和解决方法。本质上，这些例子说明，除了提供针对块的全局串行化实体外，每一个节点（比如请求者）也应该保持各个块的局部串行次序；例如，不应该把进入的事务提交给仍有未决事务的块。

591

2. 为实现顺序同一性的单元间的串行化

满足顺序同一性 (SC) 的充分条件的两个最有趣的成分是：检测写的结束（保持程序原序所需）和保证写的原子性。在基于总线的机器中，互连网络的固有性质使请求者及早地发现写的结束，写操作一旦获得总线使用权便被提交并对处理器应答，而不等到真正更新或作废其他的高速缓存（见第 6 章）。通过提供一个所有事务经过的集中的路径，并保证新的数据值在该路径之外先进先出的可见次序，基于总线的系统也可以很自然地实现写的原子性。

在一个具有分布式网络但不对共享数据做高速缓存的机器中，检测写的完成需要来自保存该单元存储器的显式的确认（见第 7 章）。事实上，确认可以早一点发出，一旦我们知道写已经到达那个节点并被插入存储器的 FIFO 队列，写就提交了，因为进入队列的所有后续读将不再能看到旧值，我们可以利用提交来代替完成以保持程序原序。写的原子性可以自然地实现：写只有在到达主存储器时才是可见的，在那一点对所有处理器都可见。

当有分布式网络并且块有多个副本时，在作废或更新真正到达所有节点之前假定写的完成是很困难的。不能在写操作到达宿主节点时就向请求者发出确认并假定操作已经有效地完成。因为同一个请求者在接收到对前一个写操作 X 的这样的确认之后，有可能按程序原序发出另一个后续的写操作 Y，但对另一个处理器，Y 可能在 X 之前可见，因此违反了 SC。发生这种情况的原因是 Y 对应的作废或更新事务可能经不同的路径通过网络或者是网络不提供点对点的次序保障。只有从所有副本收到显式的确认时，我们才能认为操作已经完成。当然，拥有副本的节点可以在接收到作废时立即发出确认，而不用等到对高速缓存的操作完成，前提是它能够在本身的高速缓存层次中保证适当的次序（正如第 6 章使用提交而不是完成）。为了满足 SC 的充分条件，处理器在发出一个写请求后必须等待，直到接收到所有的确认，然后才能通过该写操作进行下一个存储器操作。

在存在多个副本和分布式网络的情况下，保证写操作的原子性也同样很困难。为了解这一点，图 8-14 给出了一个依赖写原子性的代码段的语义遭到破坏的例子，例子的代码段选

自第5章（见图5-11）。必须适当地调度网络事务来满足顺序同一性的约束。在基于作废的方案中保持写操作原子性的一种常用方法是，块的当前拥有者（主存模块或在其高速缓存中保持脏副本的处理器）在对生成新值的写操作的所有作废确认返回之前，不允许任何进程访问新值，以此提供原子性。这样，在新值为所有处理器可见之前，没有一个处理器能看到它。维持原子性对于基于更新的协议要困难得多，因为数据被送往各共享者，并立即可以访问。为了保证新值对所有共享者可见之前，没有任何共享者能对它读取，需要两阶段的交互。在第一个阶段，该存储块在所有相关处理器的副本被更新，但禁止这些处理器访问新值。在第二个阶段，通过上述的确认了解到第一个阶段已经完成之后，向这些处理器发出消息，允许它们使用新值。实现上的困难以及性能上的限制，使得更新协议在基于目录的可扩展机器中不像其在基于总线的机器中那样有吸引力。

592

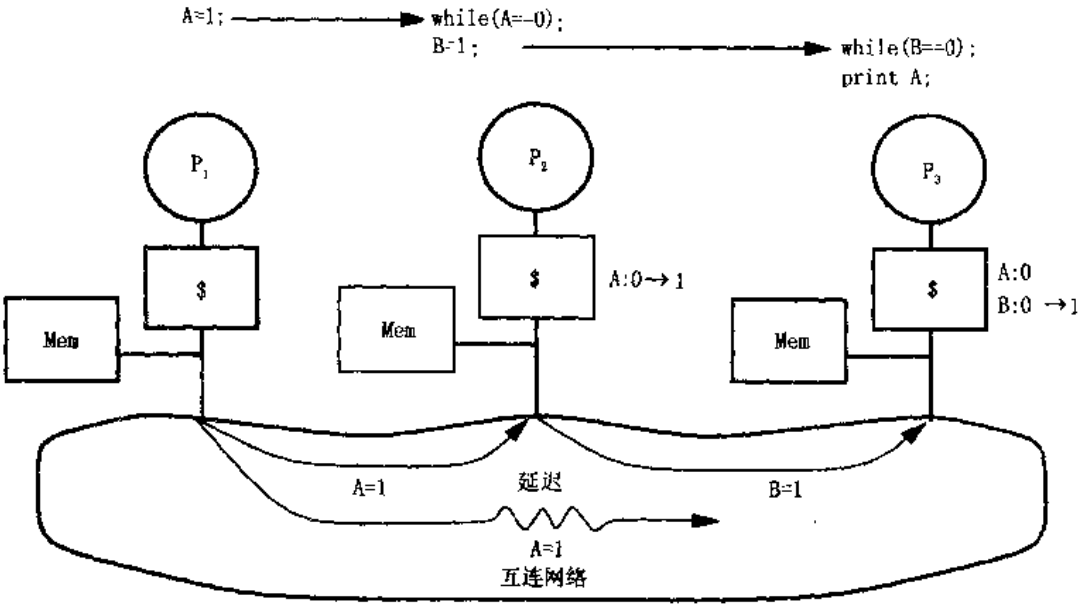


图8-14 在具有高速缓存的可扩展系统中违反写原子性的例子。该图显示了三个处理器和各自执行的代码段。假设网络保持点对点的次序，并且高速缓存中A和B的副本初值为0。为简单起见，忽略查目录和满足读扑空的事务。在SC下，我们希望P_3打印的A的值为1。但是，如果P_2先看到A的新值，跳出自己的while循环，甚至在不知道P_3是否已经能够看到P_1对A的写入值之前，就对B写入。其结果是，P_3可能在看到P_1对A的写入之前，先看到P_2对B的写入，因为P_1的写产生的作废或更新可能经过一段拥塞的网络而被延迟（而其他的事务并不经过拥塞网段）。因此，P_3读到B的新值和A的旧值，产生非预期的结果

593

3．死锁

在第7章中讨论了请求-响应协议潜在死锁的重要来源，和共享地址空间的死锁类似，比如装满了有限的输入缓冲，我们建议了三种解决缓冲死锁的方法：

1）提供一个足够大的缓冲空间，或者在请求端使用分布链表缓冲请求，或者对最大可能的到达事务量提供足够大的输入缓冲空间（通过硬件或主存）。

2）使用否定回答NACK。

3）提供分离的请求和响应网络，既可以是物理上分离的，也可以是各自带缓冲的多路复用，用以防止表现不好行为的请求网络中的活动拥塞正常响应事务的传输。

两个分离的网络对于一个严格的请求－响应协议来说已经足够，也就是说，所有事务被分成请求和响应，这样一个请求只产生一个响应（或什么也不产生），而响应不再产生进一步的事务（即不产生进一步的依赖关系）。但是我们知道为了提高性能，许多实际的一致性协议都使用转发，并不总是严格的请求－响应协议，这破坏了避免死锁的假设前提。一般来说，我们需要的网络（物理的或虚拟的）与完成给定操作所要求的不同类型事务的最长的链一样多，这样总是能保证处于链的端点的事务（即不产生进一步事务的事务）向前推进。但是，使用多个网络代价昂贵，多数并不能充分利用。除了提供足够的缓冲（比如 HAL S1 和 MIT Alewife）或使用 NACK 的方法外，还有两种方法可以避免非严格的请求－响应协议中的死锁。它们都是先假定协议是严格请求－响应协议并且提供两个物理或虚拟的网络，然后，依靠发现死锁可能发生的场景并求助于一种不同的机制来避免这些情况下的死锁。这个机制可以是 NACK 机制，或回到严格的请求－响应协议。

潜在死锁场景的发现可以有多种方法实现。在 Stanford 的 DASH 机中，当输入缓冲和输出缓冲都超过一个阀值，并且在输入缓冲头部的请求会产生如干涉或作废之类进一步的请求时（即该请求会违反严格的请求－响应操作，可能引起死锁），节点就保守地认为将要发生死锁。另一种策略是当输出请求缓冲满，并且在 T 个周期内没有从输出缓冲移走一个事务时，将认为有可能死锁。当发现了潜在的死锁时，DASH 系统采用第一种基于 NACK 的方案来避免死锁：节点从输入队列头逐个移走请求，并且向相应的请求者发出 NACK 消息，直到头部的请求不会产生进一步请求或输出请求队列不再满为止。被拒绝掉的请求者稍后会重发它们的请求。

Origin2000 采用了另外一种死锁避免方法。当发现潜在死锁时，不是向请求者发出 NACK，而是向请求者发出一个响应，让请求者直接向共享者发出干涉或作废请求，也就是说，系统动态地从转发协议回到严格的请求－响应协议，暂时牺牲性能，但切断死锁回路。该方案的优势在于，对于与拥塞相关的问题而言，NACK 是一个统计的方案，但并不是一个健壮的方案：在坏的情况下，请求不得不被重试多次，这导致网络流量和时延增加，推迟了操作的完成。下面将会看到，这种动态退回严格请求－响应协议的方案在处理活锁时也有优势。 594

4. 活锁

在通过为请求提供足够大的缓冲区来避免死锁的协议中，无论缓冲区是集中式的，还是分布链表式的，只要它是先进先出的，就自动地处理活锁和挨饿问题。但其他方案并不考虑活锁和挨饿，通常，处理因竞争条件（多个处理器在同一时刻写同一块）而产生的经典活锁的做法是，让发往宿主的第一个请求通过，但拒绝所有其他的请求。

NACK 是解决上述竞争条件而不产生活锁的有用的机制。但是，当面对输入缓冲的限制，用它来避免死锁时（比如我们前面讲过的 DASH 解决方案），有可能会引起活锁。例如，一个节点发现潜在的死锁，就向一些请求发出 NACK 信号，但所有那些请求有可能又在同一时刻重发。在极端的情况下，这种情况会不断重复出现并导致活锁[⊖]。死锁的另外一种解决方案，即在潜在死锁场景下切换为严格的请求－响应协议，这不会带来活锁问题。它保证

⊖ 尽管 DASH 体系结构的设计使用 NACK，实际的原型实现却通过使用足够大的请求输入缓冲绕过这个问题，因为节点的数量和每个节点可能的未决请求的数量都不大。但是，这对于较大的、更为进取性的、不能提供足够大缓冲空间的机器来说，不是一个健壮的解决方案。

向前推进，消除了宿主节点处的请求－请求依赖关系。

5．挨饿

在设计完善的协议中，挨饿一般是不会发生的，但并不能完全排除这种可能性。解决挨饿最公平的方法是以先进先出的次序对请求进行缓冲，这同时也解决了死锁和活锁的问题。但这可能会导致性能下降。其他不采用这个方法的协议要避免挨饿并不容易。用 NACK 和重试的方法解决死锁和活锁的方案比较容易产生挨饿问题，尤其是当许多处理器反复争用同一资源时更是如此。有些处理器可能总是成功的，而某个或更多个处理器却不那么幸运，总是被拒绝。

595

协议可以不对挨饿进行任何处理，而是依靠系统延迟时间的变化使得那种无限重复的情况不会出现。DASH 机就采用了这种策略，如果那种重复的情况超过时限就会发出一个总线超时错误。另外，我们可以在两次重试之间插入一段随机的延迟来进一步降低发生挨饿的概率。最后我们还可以根据请求被拒绝的次数来赋予它们不同的优先级，Origin2000 的协议就采用了这种技术。

在对基本的目录组织结构、高层的协议、关键性能和正确性问题有了一般性的了解之后，现在可以进入基于存储器和基于高速缓存协议的案例分析了。我们将会看到在实际实现中协议状态和动作的形态，目录协议是如何与底层的处理器交互作用并受其影响的，可扩展高速缓存一致的机器是什么样的，在性能和保证正确性及协议调试验证的复杂性之间，实际的协议是如何权衡的。

8.5 基于存储器的目录协议：SGI 的 Origin 系统

我们的讨论从扁平的、基于存储器的目录协议开始，使用 SGI 的 Origin 体系结构作为研究实例。至少对中等规模的系统来说，这种机器使用了一种全位向量目录表示。Stanford 的 DASH 机研究原型中也使用了同样的目录表示，但是协议稍有不同，它是第一台采用基于目录一致性的分布存储器的机器。在当前和下一个案例分析（Sequent NUMA-Q 机使用的 SCI 协议）中遵循同一个讨论框架。从基本的一致性协议开始，包括目录结构、目录和高速缓存的状态，读、写、回写操作的处理方式以及所使用的性能提升方法。然后我们将简单地讨论针对主要的正确性问题的措施，然后是为了额外功能的重要的协议上的扩展。接着，将考察作为多处理器的机器的其余部分，以及一致性的机制如何适应它。这包括处理器节点、互连网络、输入输出系统、以及在目录协议和底层节点间的有趣交互。这个案例分析最后讨论一些重要的实现问题（指明了它如何工作以及重要的数据和控制通路）、协议的基本性能特征（时延、占用度、带宽）、以及在样本应用所表现出的应用性能。

596

8.5.1 高速缓存一致性协议

Origin 系统是由许多处理节点组成的，这些处理节点由基于交换的互连网络连接在一起（见图 8-15）。每个处理节点包括两个 MIPS R10000 处理器，每个处理器都有一级和二级高速缓存，节点还包括机器主存储器的一部分、一个 I/O 接口和一个称作 Hub 的单片通信辅助部件或一致性控制器，该部件实现一致性协议。Hub 集成在存储器系统中。它可以看见该节点内处理器产生的所有（二级）高速缓存扑空，不管它们是被本地还是远程满足；它从网络接收进来的事务（实际上，Hub 实现了网络接口）；它能从本地处理器高速缓存取得数据。

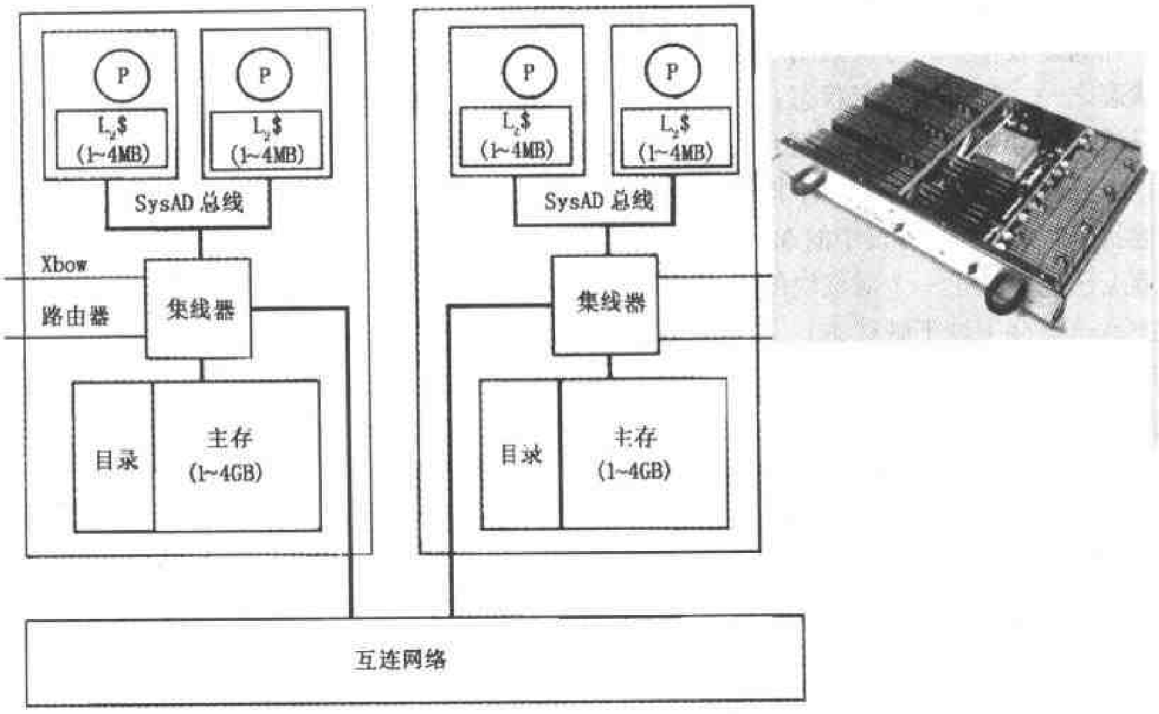


图 8-15　Silicon Graphics 的 Origin2000 多处理器的框图。每个节点包含两个处理器、一个叫做 Hub 的通信辅助部件或控制器，以及带有相关目录的主存储器。照片显示了单节点板。照片引自 Silicon Graphics，Inc

根据 8.4.1 节讨论的性能问题，在协议级，Origin2000 在查找宿主目录的同时，并行地进行应答转发和推测性存储器操作。在机器组织结构级，Origin 选择每个节点两个处理器的设计主要出于成本考虑：节点内的其他部件（Hub、系统总线等等）被处理器所共享，在分摊开销的同时仍希望为每个处理器提供足够的带宽。Origin 的设计者相信，在一个节点内通过侦听总线互连导致的时延和带宽上的弊端大于它的好处，所以决定不在节点内两个处理器间维持侦听一致性。另外，系统地址和数据（SysAD）总线不过是节点内两个处理器之间复用的物理链路。这牺牲了节点内高速缓存到高速缓存之间共享的潜在优势，但也消除了侦听引起的时延、占用和高速缓存标志位的竞争。特别是在一个节点内只有两个处理器的情况下，高速缓存到高速缓存之间成功共享的机会很小，所以由此带来的劣势可能反而占主导地位。由于两个处理器共享 Hub，可以支持对网络的请求（不是对目录协议）的合并，但是这需要额外的实现开销，所以没有这样做。在本节讨论协议时，为简单起见，先考虑每个节点只有一个处理器和相关的高速缓存层次结构、Hub 及存储器的情况。在本节后面，将考虑每节点两个处理器对目录结构和协议的影响。

除了应答转发，Origin 协议的最有趣之处在于它采用了忙状态和 NACK 解决竞争，提供对一个单元操作的串行化，在于它的死锁和活锁的解决方法，在于它处理因回写引起的竞争的方法，还在于它不依赖于在网络事务之间保持任何次序（甚至不依赖于相同端点对之间事务的点对点的次序）。为了说明完整的协议如何在有竞争的情况下工作，以及为了说明在不同情况下使用的性能增强技术，我们将考察 Origin 如何把各种技术综合在一起来处理读和写操作。

597

1．目录结构和协议状态

一个存储器块的目录信息在该块的宿主节点内保存。现在假定它是全位向量，稍候将考

察目录组织结构是如何随着机器的规模变化的。

在高速缓存中，协议采用与第5章中相同的MESI状态。在目录中，一个块可以处于7种状态之一。三个是稳定状态：未拥有，即在系统中没有高速缓存副本；共享，即零个或多个只读高速缓存副本，其位置由存在位向量指明；排他，系统中只有一个读写高速缓存副本，由存在位向量指示。一个排他的目录状态意味着在高速缓存中块是处于脏的或者干净的排他状态（即MESI协议中的M或E状态）。另外三个状态是忙状态。如前所述，这表示宿主节点已经接收了一个对该块的请求，但它本身不能完成操作（例如，该块可能在另一个节点的高速缓存中处于脏状态）；完成该请求的事务仍在系统内进行，所以宿主的目录不能响应对该块的新的请求。三个忙状态对应于三种不同类型的仍然在进行的请求：读、排他读或升级、非高速缓存读（结果不进入处理器高速缓存的读，因此不必保持一致性）。这个协议使用忙状态和NACK（而不是大量的缓冲空间）来避免竞争条件，提供对存储器单元操作的串行化。第7个状态是毒化（poison）状态，用以实现惰性的TLB击落方法，该方法实现页面在存储器中的迁移。（像非高速缓存操作和页面迁移这样的协议扩展在8.5.4节中讨论。）给定这些状态，让我们看一看一致性协议是如何处理来自一个节点的读、写和回写请求的。

598

2. 读请求处理

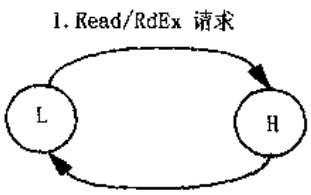
假定处理器发出的读请求在高速缓存层次中扑空。本地Hub检查扑空的地址以决定其宿主节点，并向宿主节点发出一个读请求事务，以便查找目录项。如果宿主就在本地，那么本地Hub自己查找本地的目录。在宿主节点，该块的数据在查找目录项的同时被并行推测访问。目录项的查找比推测性数据访问早一个周期完成，它指出存储器块处于以下几个不同的状态之一，针对不同情况采取不同的动作。

- 共享或未拥有。这意味着宿主节点主存中有数据的最新副本（因此推测性访问成功）。如果状态是共享的，目录存在位向量对应请求者的位被置位；如果状态是未拥有的，目录状态设为独占（实现侦听系统中共享信号所提供的功能）。然后宿主节点以应答事务将数据发回请求者。这些情况满足严格的请求－响应协议。当然，如果宿主节点与请求节点是同一个节点，那么就不会产生网络事务或消息，这是一个本地可满足的扑空。
- 忙。这意味着宿主节点不应该在此刻处理请求，因为对该块的前一个请求的处理正在进行。请求者会收到一个否定回答（NACK）消息，被告知稍后再试。NACK被归类为响应，但像确认一样不携带数据。
- 排他。这是最有趣的一种情形。如果宿主节点不是块的拥有者，那么必须从它的拥有者那里得到有效的数据，并且必须找到拥有者通往请求者和宿主节点的路径。Origin协议采用应答转发，请求被转发给拥有者，拥有者直接应答请求者，并把修正的消息发给宿主节点。如果宿主节点本身就是拥有者，那么宿主简单地应答请求者，把目录状态变为共享，并在存在位向量中置位请求者对应的位。事实上，目录把宿主节点的高速缓存和其他高速缓存一样对待；惟一的不同之处是宿主目录和宿主节点高速缓存之间的“消息”不转换成网络事务。

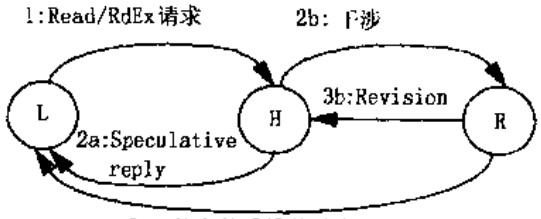
599

现在看一下当读请求到达宿主节点并发现状态为排他时实际发生的细节（图8-16说明了这个情况以及其他几个被讨论的情况）。如通常所做的那样，存储器块的推测性访问与目录查找并行进行。如果发现目录状态是排他，将它置为忙-排他以便应付后续的请求，并且

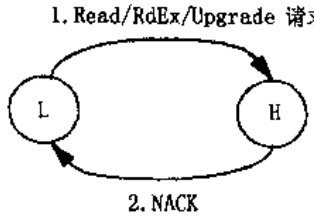
把请求转发给排他节点。我们还不能把状态置为共享，因为存储器中还没有最新的副本，也不想一直让状态停留在排他，因为这样做的话，后续的请求就会像当前的请求一样去追踪块的同一个排他副本，从而要求当前的拥有者节点而不是宿主节点来决定串行化次序。



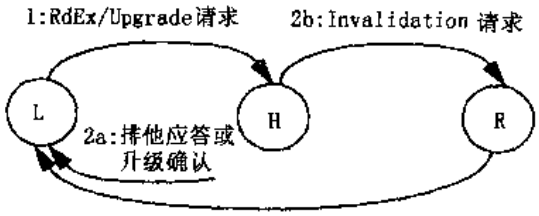
a) 对于目录状态为非拥有的块的读或排他读请求。或者对处于共享态的块的读请求。如果块的目录状态为非拥有，即使在读的情况也发出排他响应. 这样块可以以E而不是S状态装入高速缓存



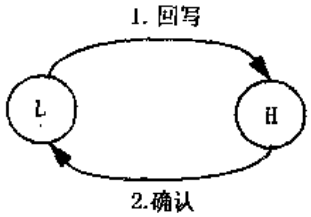
b) 对于目录状态为排他的读或排他读请求。干涉的类型分别是共享或排他，后者引起作废。修正消息是共享回写或拥有权转移



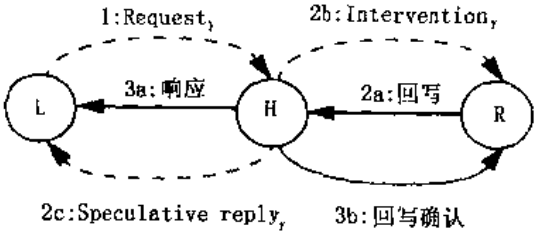
c) 对目录状态为忙的块的读/排他读请求，或者是对目录状态为忙、未拥有、排他的块的升级请求



d) 对目录状态为共享的块的排他读或升级请求



e) 对目录状态为排他的块的回写请求



f) 对目录状态为忙的块的回写请求（带下标 y 的事务和虚线代表使目录为忙的其他请求）

图 8-16　在 Origin 多处理器中响应请求的协议动作。所考虑的情况出现在图下面，指示了请求到达宿主节点时请求的类型和目录项的状态。消息或者事务类型在每条弧线边列出，因为相同的图代表了请求类型和目录状态的不同组合，所以在每一条弧线旁列出不同的消息类型

把目录项置为忙状态后，存在位向量被修改，请求者对应的位被置位，而当前拥有者对应的位被复位。此刻这样做的原因在我们考察回写请求时就会很清楚。现在看到了协议的一个有趣的方面：尽管目录状态是排他的，宿主乐观地假设在该块在拥有者的高速缓存中是（干净的）排他态而不是脏态，它会把在宿主节点推测性访问的存储器块作为推测性应答发给请求者（即这种应答携带的数据可能有用，也可能没用）。与此同时，宿主把干涉请求转发给拥有者。拥有者检查它的高速缓存的状态并执行下列动作之一。如果块处于脏状态，它把带有数据的应答直接发送给请求者，并把包含数据的修正消息发送给宿主。在请求者节点，该响应覆盖宿主以前发来的过时的推测性应答。发送给宿主的带有数据的修正消息被称

做共享回写，因为它把来自拥有者高速缓存的数据写回主存，并告诉主存把块的状态置为共享。如果块处于排他态，拥有者发给请求者的应答和发给宿主的修正消息中不含有数据，因为两者已经有最新的副本（请求者从来自宿主的推测性应答得到数据）。对请求者的响应是一个简单的确认，而修正消息叫做降级消息，因为它要求宿主把块的状态从（忙）排他降为共享。在每一种情况下，宿主接到修正消息后都把状态从忙改为共享。

你可能注意到在这种情况下，使用推测性应答并没有任何性能上的优势，因为请求者为了了解排他节点的真实状态，在能够使用该数据之前只能等待。事实上，一个更简单的替代方案是假设拥有者的高速缓存中块是脏的，因此不发送推测性应答，总是由拥有者发回带有数据的应答，不管它的块是处于脏（干净）的排他状态。为什么 Origin 采用推测性应答呢？这里有两个原因，说明了协议是如何受到现存的处理器的影响以及不同的协议优化如何相互影响。首先，Origin 所使用的 R10000 处理器的高速缓存控制器在收到对一个排他（不是脏）的高速缓存块的干涉请求时，不返回数据，因为假定存储器中有有效的副本。其次，推测性应答允许协议的另一种不同的优化是，它允许处理器高速缓存块被替换时，简单地丢掉一个（干净）排他块而不必通知主存储器，它具有惟一的副本，应该对后续的请求做出应答，因为主存储器在需要时总是发出推测性应答。

3. 处理写请求

600~601

在第 5 章中讲过，一个调用协议的写扑空会产生要求数据和拥有权两者的排他读请求，或者产生只要求拥有权的升级请求，因为请求者的数据是有效的。在每一种情况下，请求将到达宿主节点，查找目录的状态来决定执行何种动作。如果目录的状态不是未拥有（或导致否定回答的忙），在其他高速缓存中的副本必须作废。为了保持次序模型，必须对作废显式地确认。

和读的情况一样，可以采用严格的请求-响应协议、干涉转发或应答转发（见习题 8.4）。Origin 选择了应答转发以降低时延：宿主更新目录状态并直接发出作废；作废消息中还包括了请求者的身份，这样被作废者可以直接向请求者发回确认。排他读和升级的实际处理取决于请求到达时目录的状态，也就是说，看它是未拥有、共享、排他还是忙状态。

- 未拥有。如果是一个升级请求，目录状态应该是共享。未拥有状态意味着这个块已经从请求者的高速缓存中替换，目录已被通报，因为它发出了升级请求（这是可能的，因为 Origin 的协议并没有假设点对点的网络次序）。升级请求不再是一个合适的请求，所以它被否定回答了。写操作将像一个排他读那样被重试。如果请求是排他读，目录状态被改成排他，请求者的存在位。宿主用存储器中的数据应答。
- 共享。在有副本的高速缓存中块必须被作废。宿主的 Hub 先利用存在位向量列出应该被作废的所有共享者。然后，将目录状态设为排他，并置请求者的存在位。这保证下一个对该块的请求能转发给请求者。如果请求是排他读，宿主将向请求者发出一个响应（叫做“具有未决作废的排他应答”），此应答包含共享者的数目，希望从这些共享者得到作废确认。如果请求是一个升级，宿主向请求者发出一个类似的“具有未决作废的升级确认”，但不携带该块的数据。在每一种情况下，宿主下一步给所有的共享者发送作废请求，共享者接着向请求者（不是宿主）发出确认。请求者在“关闭”或结束操作前，等待所有的确认到来。如果在此期间，对该块的一个

新的请求到达宿主，它将看到目录的状态是排他，并且将被作为干涉转发给当前的请求者。当前的请求者不会立刻处理这个干涉，而是将它缓冲，直到收到对自己请求的所有的确认并结束操作（更进一步，在此期间到达宿主的请求将发现块处在忙-排他状态，这前面早已讨论）。

- 排他。如果是一个升级请求，那么，排他目录状态表示另一个写请求在宿主节点胜过它。升级不是合适的请求，于是被拒绝。对一个排他读请求，采取下面的动作。和读一样，宿主将目录状态置为忙，将请求者的存在位置位，向它发送一个推测性应答。向拥有者发出一个包含写请求者身份的作废请求（如果宿主是拥有者，只是对本地高速缓存的作废而没有网络事务）。如果拥有者的块处于脏状态，它向宿主发出一个“拥有权转移”的修正消息（不含数据），向请求者发出一个带有数据的应答。这个应答覆盖请求者从宿主那里收到的推测性应答。如果拥有者的块处于（干净的）排他状态，它依赖于来自宿主的推测性应答，简单地向请求者发一个确认，向宿主发一个“拥有权转移”的修正信息。 602
- 忙。和读的情况一样，请求被否定回答，必须被重试。

4. 处理回写请求和替换

当一个节点替换它高速缓存里的脏块时，它产生一个回写请求。这个请求携带数据并被宿主确认应答。当回写请求到达时，目录不能处在未拥有状态或共享状态，因为回写请求者有脏的副本。（在回写产生和它到达宿主之间，不允许一个读请求把目录状态变为共享，因为这种请求可能会被转发给正在申请回写的那个节点，而目录状态会被设置为忙）下面来看看在排他和忙这两种目录状态下回写请求到达宿主将会发生什么。

- 排他。目录状态从排他变成未拥有（因为仅有的高速缓存副本被从高速缓存中替换掉了），返回一个确认信息。
- 忙。这表示了一个有趣的竞争情况。目录状态只能是忙，因为对该块的一个干涉（由于来自另一个节点 Y 的请求）已经被转发给正在回写的那个节点 X。干涉和回写在互连网络中擦肩而过。现在，我们面临一个有趣的情形。从 Y 来的其他操作已经在执行中，不能被取消。我们不能丢弃这个回写，否则将失去这个块的惟一有效副本。我们也不能拒绝这个回写并在 Y 的操作结束后重试，因为那样的话，Y 的高速缓存中将有一个有效的副本，而一个不同的脏副本会从 X 的高速缓存中回写到存储器去！这个协议本质上通过合并这两个操作，用回写作为对 Y 的请求的响应来解决这个问题（见图8-16f）。发现目录状态是忙的回写请求把状态变为共享（如果状态是忙-共享，即来自 Y 的请求是为了读一个副本）或者排他（如果状态是忙-排他）。回写返回的数据由宿主转发到请求者 Y。这作为给 Y 的应答，而不是像没有回写的情况下，直接从 X 那里收到应答。当由于 Y 的请求，X 收到一个该块的干涉时，它只是忽略它（见习题8.13）。目录也发送一个回写确认给 X。当 Y 收到响应时结束操作，当 X 收到回写确认时，回写操作结束。在讨论操作串行化时，将看到在更复杂的情况下这个处理的一个例外。一般来说，回写把很多微妙的情形引入了基于目录的一致性协议。 603

如果正被替换的高速缓存块处于共享状态，节点可以选择将替换提示消息发回宿主，让

宿主清除目录中的存在位，也可以选择不这样做。替换提示消息避免对该块的下一个无用的作废，并且能减少有限指针目录表示方案中表项的数量，但它们占用通信辅助部件，不能减少流量。实际上，如果该块没有被其他的节点再次写入，替换提示就是一种浪费。Origin 协议不使用有限指针表示，也不使用替换提示。

在 Origin 协议中，用于一致性存储器操作的事务数总共是 9 种请求，6 种作废和干涉，39 种响应。对非一致性操作，例如非高速缓存的存储器操作、I/O 操作和特殊的同步支持，事务的数目是 19 种请求，14 种应答（因为没有一致的高速缓存，所以没有作废或干涉）。

8.5.2 关于正确性问题

到目前为止，我们已经看到了在不同的节点上读扑空或写扑空时发生的情况以及解决一些重要竞争的方法。现在分析 Origin 协议另一个侧面，考察它对 8.4.2 节所讨论的正确性问题的特殊解决方案，以及机器为处理可能发生的错误提供的特性。

1. 为了一致性对单元的串行化

宿主是将来自不同处理器的高速缓存扑空进行次序串行化的实体。我们已经看到，这里所提供的串行化不是在宿主节点缓冲请求，直到前一个请求结束；也不是把请求转发给拥有者节点，即使目录处于忙状态也不这样做；而是在忙状态下由宿主拒绝请求，让它们以后重试。只有在稳定目录状态下请求才被转发。串行化是由宿主接受请求的次序决定的，也就是说，由宿主满足请求或转发它们的次序，而不是由请求到达宿主的次序决定的。

8.4.2 节中对串行化技术的一般性讨论表明：对一个给定单元的串行化比简单的全局串行化要求更高，因为实行串行化的实体没有在所有相关节点上与给定操作有关的事务何时结束的全局性知识。有了对协议的足够深入的理解，我们现在考察这个问题的某些具体例子（Lenoski 1992），看看问题是如何解决的（见例 8.1 和例 8.2）。

604

例 8.1 考虑下面的代码段

P_1	P_2
rd A (i)	wr A
BARRIER	BARRIER
rd A (ii)	

对 A 的写入可以发生在对 A 的首次读之前或之后，但是，它对于首次读必须是串行的。第二次对 A 的读总是返回被 P_2 写入的值，然而，如果我们不小心的话，写的效果很可能被丢失。请说明在像 Origin 这样的协议中这类情况如何发生，并讨论可能的解决方案。

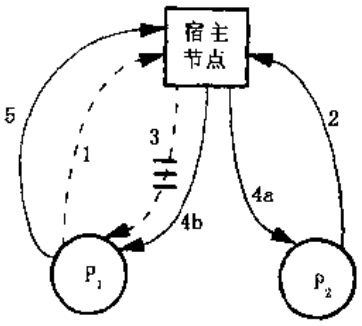
解答： 图 8-17 显示了这个问题如何发生，图中的正文解释了事务、事件序列和问题。有两种可能的解决方案：一种不是很诱人的方案是对读应答本身显式地确认，让宿主接受到这个确认后继续处理下一个请求。这进一步违背了请求－响应协议的本性，引起了缓冲和潜在死锁的问题，并且导致大的延迟。更为可能的方案是 P_1 这样对块有未决请求的节点在它的未决请求完成前，不允许像作废这样的其他请求访问其高速缓存中的对应的块。P_1 可以缓冲进入的作废请求，并且只有在接受并结束了读应答之后，才实施缓冲的作废请求。或者，P_1 甚至可以在收到读应答前实施作废，然后，当它返回并重试读时，认为应答无效

(一个 NACK)。Origin 机用前一种方法，而 DASH 机用后一种方法。在两种机器里，P_1 的第一次读相对于 P_2 的写的次序是不同的，但两种次序都是有效的。需要的缓冲很小，并且不会产生死锁问题。■

例 8.2　除了请求者之外，宿主在一个块的操作结束之前，也不允许真正实施对该块(或对它的目录状态)的新的操作。否则，目录信息会被破坏。请给出一个说明这种需要的例子并讨论解决方案。

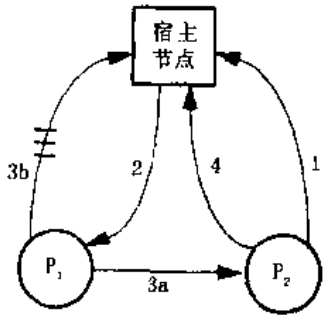
解答：图 8-18 显示了一个更为精妙的例子。发出写请求的节点通过确认检测写的结束(就它是否还要关心这个写操作而言)，然后才能处理对该块的另一个请求。问题是宿主并不会等待涉及它的写操作的结束，即等待修正消息和目录更新，然后才允许其他访问（这里是回写）作用于这个块。Origin 协议用忙状态防止这样的现象发生：如果回写在修正信息之前到达时，目录将会处于忙－排他状态。当目录检测到回写来自使目录状态变成忙－排他的同一个节点，该回写被拒绝并必须被重试（回忆一下关于回写处理的讨论，如果发出导致忙状态的请求的节点不同于发出回写的节点，对回写的处理有所不同；在这种情况下，回写不是被否定回答，而是作为响应被送到请求者去）。■

605



1）P_1 向宿主节点发出读 A 的请求
2）P_2 像宿主发出排他读请求（为写入 A）。宿主（串行化实施者）在结束先收到的来自 P_1 的读之前，不会处理它
3）作为对 1）的响应，宿主向 P_1 发送应答（并设置目录存在位），宿主此时认为读已经结束了（对读应答没有确认）。不幸的是，应答没有马上到达 P_1
4a）作为对 2）的响应，宿主向 P_2 发送对应请求 2 的数据应答
4b）作为对 2）的响应，宿主向 P_1 发送作废，它在事务 3 之前到达 P_1（Origin 没有假定点对点的次序，一般来说，作废是个请求，而事务 3 是个响应，它们在不同的网络中传输）
5）P_1 接受并实施作废，像宿主发送确认
最后，读应答 3）到达 P_1，覆盖作废的块。当 P_1 在路障（Barrier）之后读 A 时，它读到旧值，而不是看到作废的块因而去取新值。对于 P_1 来说，P_2 的写的效果被丢失

图 8-17　说明请求者节点操作的局部串行化需求的例子，这个例子显示了一个写是如何被丢失的，尽管宿主认为它是按次序工作的。与第一个读操作相关的事务用虚线表示，而与写操作相关的事务用实线表示。事务线上的 3 个横杠表示事务在网络中延迟



初始状况：块在 P_1 的高速缓存中处于脏状态
1）P_2 向宿主发出排他读请求
2）宿主把请求转递给 P_1（脏节点）
3）P_1 向 P_2 发送数据应答（3a），向宿主发送“拥有权转移”的修正信息给宿主，把拥有者改为 P_2（3b）
4）接收到它的应答后，P_2 认为写已经结束了，继续执行，但碰到要替换掉刚刚变脏的块，因此要通过事务 4 把该块回写
宿主节点在收到来自 P_1 的拥有权转移修正消息之前收到这个回写（即使网络保持点对点的次序也无济于事），这个块被写入存储器，然后，当修正消息到达宿主节点时，修改目录指示 P_2 拥有脏副本。但这不是事实，我们的协议被破坏了

图 8-18　说明宿主节点上操作的局部串行化需求的例子。这个例子显示了如果宿主不等待涉及它的先前的请求结束（例如，从拥有者节点接收到修正消息）就允许对同一个块的访问，目录信息是如何破坏的

606

这些例子说明了另一个一般性要求的重要性，即除了要有对块做全局串行化的实体存在

之外，节点必须局部地满足正确的串行性。任何节点，不仅仅是实施串行化的实体，只要涉及它本身的话，在对一个块的早先的未决存储器操作（节点已经开始操作）结束之前，不能实施针对该块的新的存储器操作的事务。

2. 保持存储器同一性模型

动态调度的 R10000 处理器允许发出脱离程序原序的独立的存储器操作，同时允许多个操作未决并允许未决操作之间重叠。但是，它保证操作按照程序原序完成，事实上，它保证写操作相对于其他操作以程序原序离开处理器环境并被存储器系统可见；因此，保持了顺序同一性（第 11 章进一步讨论必要的处理器机制）。处理器并不满足第 5 章所说的顺序同一性的充分条件，因为它不等前面的操作完成就发出下一个操作，但是，采用该处理器并提供原子性的系统本身满足顺序同一性模型[⊖]。

因为处理器保证可见性和完成遵循程序原序，存储器层次结构可以在不破坏这个性质的前提下，按它的期望改变对不同单元操作的次序。Origin 协议提供前面讨论过的写的原子性：一个节点不允许任何对作废操作未决的块的新的访问，直到对这些作废的确认已经返回（即写已被提交）。但是，由于 Origin 协议与处理器的交互而产生的一种实现上的考虑对保持 SC 是重要的。回忆一下图 8-16d，对于在目录中处于共享状态的块的写请求（排他读或升级）导致什么情况发生。请求者接收到两种类型的响应：来自宿主节点的排他应答，这在前面已经讨论过了，它的角色是指出在存储器处写已经相对其他对该块的操作串行化了，或许将返回数据；另一种响应是作废确认，指出其他副本已经被作废，写已经结束。然而，微处理器仅仅希望对它的写请求有一个响应，就像在单处理器系统里那样，所以这些不同的响应必须由被请求的 Hub 处理。为了保证顺序同一性，Hub 只有当排他应答和作废确认两者都收到后，才必须把响应传递给处理器，允许它声明写的完成。它不能在收到排他应答时简单地把响应传递给处理器，因为这将允许处理器在对当前单元的所有作废被确认之前，结束对其他单元的后面的访问，从而违反顺序同一性。我们将在 9.1 节看到，当使用比 SC 更放松的存储器同一性模型时，这种违反是有用的。

607

3. 死锁、活锁和挨饿

Origin 使用有限的输入缓冲和一个非严格的请求-响应协议。正如 8.4.2 节所讨论的，为了避免死锁，它用了一个技术：在它侦测到可能引发死锁的高度竞争时，就转回严格的请求－响应协议。因为没有使用 NACK 来减轻竞争，也避免了这种情况下的活锁。典型的活锁是由多个处理器试图同时写一个块引起的，我们用忙状态和 NACK 来避免它（回忆一下在这种情况下，NACK 能避免而不是引起活锁）。到达宿主的第一个请求设置状态为忙并向前推进，其他的请求都被否定回答并必须重试。

总的来说，Origin 协议的思想有两部分：1）成为“非记忆的”，也就是说，每一个节点对进入事件的反应只使用当前的本地状态，不涉及先前事件的历史记录；2）当一个操作请求其他资源时，不允许它占用全局共享资源。后者导致对忙资源请求的否定回答而不是缓冲，这有助于防止死锁。这些决策大大地简化了硬件，在大多数情况下提供了高的性能。然而，因为用拒绝而不是 FIFO 次序，挨饿的问题依然存在。这个问题的解决是给请求赋予优

⊖ 对于在一致性协议控制下的访问来说，这是成立的。处理器也支持一致性协议不可见的非一致的存储器操作，对这些操作，系统不保证任何次序。在这种情况下，插入同步以保持所希望的次序是用户的责任。

先级，一个请求的优先级是该请求被拒绝次数的函数[⊖]。

4．错误处理

即使是正确的协议，运行时还是可能发生硬件和软件的错误。这些错误可能会破坏存储器内容，或者把数据误写到非期望的单元（例如，写操作的地址被破坏）。Origin 系统提供了很多的标准机制来处理部件的硬件错误。所有的高速缓存和存储器都用差错校正码（ECC）保护，而所有的路由器和 I/O 链路都用循环冗余校验码（CRC）和一个硬件链路级的协议来保护，该协议自动地检测错误并重试。此外，当程序引起失效时，系统提供了在机器中该程序运行的部分包容失效的机制。对存储器和 I/O 设备提供了访问保护权限，防止非授权的节点对它们修改。这些访问权允许把操作系统划分成单元和分区，这种结构叫做*细胞结构操作系统*。一个细胞是在启动时配置的一些节点。如果一个应用运行于一个细胞，就不允许它对细胞外的存储器或 I/O 写入。如果应用失败并破坏了存储器或 I/O，它只会影响运行在同一细胞中的其他应用或系统，而不会破坏细胞外的代码。所以，细胞是系统中抑制故障的单位。 608

8.5.3　目录结构的细节

尽管到目前为止为了简单性，我们假定了全位向量目录组织结构，Origin 目录项的实际结构却要复杂一点，有两点理由：第一，为了安排每个节点两个处理器；第二，允许目录结构在 64 位目录项的情况下支持多于 64 个节点的扩展。这里事实上有三种可能的目录位格式或解释：如果一个块在高速缓存中处于排他状态（即，被修改或排他），那么该目录项的其余部分就不是有一位被置位的位向量，而是包含了指向那个特定处理器（不是节点）的显式的指针。这就意味着宿主转发的干涉指向那个特定的处理器。否则，如果目录状态是共享的，目录项就被解释为一个位向量。位向量中的位对应于节点，所以尽管一个节点中的两个处理器高速缓存不能由总线保持一致，这种格式下目录可见的单位是节点或 Hub，而不是处理器。如果一个作废被发送给了 Hub，与干涉不同，它通过连接两个处理器和 Hub 的 SysAD 总线广播给节点中的两个处理器。有两种尺寸的存在位向量：16 位的和 64 位的（在 16 位情况下，目录项存贮在和主存储器一样的 DRAM 中，而在 64 位情况下，多出的位存放在并行查找的扩展目录存储器模块中）。16 位的向量因此支持多达 32 个处理器，而 64 位的向量最多可支持 128 个处理器。

对更大的系统来说，对位的解释变成第三种形式。在一个有 p 个节点的系统中，每一位对应一组固定的 $p/64$ 个节点。当一个组的任何一个或多个节点具有块的副本时，对应位就被置位。如果写发生时一个位处于置位状态，作废就被发送到该位所代表的所有 $p/64$ 节点（然后被广播给每个节点的两个处理器）。例如，在支持最大数目为 1 024 个处理器（512 个节点）的情况下，每一个位对应 8 个节点。这被称作*粗糙向量表示方式*，在 8.10 节当作为高级论题讨论目录表示的溢出策略时将再次遇到它。事实上，在大机器中，系统会在位向量和粗粒度向量表示两种方式之间动态地选择：如果共享一个块的所有节点都位于机器的同

⊖ 优先级机制按如下方式工作。一个块的目录项包含一个与之相关的“当前”优先级，不会使目录状态变为忙的进来的事务总是被服务。其他的请求只有当它们的优先级高于当前目录优先级，才可能被服务。如果一个这样的请求被否定回答（例如，当请求到达时目录处于忙状态），则将目录的当前优先级设置成与被拒绝的请求优先级相同。这保证了在这个请求在经过重试最终被服务以前，目录不会为其他更低优先权的请求服务。为了防止单调增而使目录项的优先级到“顶”，一旦一个优先级高于或等于目录项优先级的请求被服务，就将目录项优先级复位成零。

609 一个由 64 节点组成的八分之一中，就使用位向量方式；否则，使用粗糙向量。

8.5.4 协议扩展

除了前面讨论的协议优化外，Origin 的协议还提供了一些扩展，支持与协议交互的特殊操作和活动；包括输入/输出和 DMA 操作、页面迁移和同步。

1. 对输入/输出和 DMA 操作的支持

为了支持 DMA 设备对存储器的读取，协议提供了“非高速缓存的读共享”请求。该请求向 DMA 设备返回数据一致副本的一个快照，但此后协议不再保证该副本的一致性。该请求主要由 I/O 系统和 Hub 提供的块传输引擎使用，因此主要为操作系统所使用。从 DMA 设备对存储器进行写操作时，协议提供了“写作废”请求。写作废只是简单地把一个字的新值写入存储器，将原来的值覆盖了。它还作废了系统中所有存在的该块的高速缓存副本，使目录项回到未拥有状态。从协议的角度来看，它的行为和排他读请求类似，只是它修改了存储器中的块并且将目录项置为未拥有状态。

2. 对自动页迁移的支持

正如我们在第 3 章中所讨论的，对于有物理分布的存储器的机器而言，重要的是将数据适当地在物理存储器内分配，以使得大多数容量扑空、冲突扑空和冷启动扑空在本地满足。在像 Origin 这样的 CC-NUMA 机器中，数据按页的粒度（这里为 16 KB）在存储器中分配。尽管 Origin 的通信体系结构非常的先进，即使在没有竞争的情况下，但访问远程存储器的时延也至少是访问本地存储器时延的 2 ~ 3 倍。页面在存储器中的适当分布可以在运行时动态调整，或者由于并行程序访问模式的变化，或者由于操作系统决定把应用进程从一个处理器迁移到另一个处理器，以便在多道程序应用之间更好地管理资源。所以，系统在运行时发现移动页面的需要，自动把它们迁移到需要的地方是有用的。

Origin 给主存的每一页提供了一个扑空计数器数组，每个节点一个计数器，用它来判定什么时候大多数扑空来自非本地处理器，因此需要将页迁移。扑空计数器存在宿主的目录存储器中。当有对一个页的请求时，就将对应节点的扑空计数器加 1，并和宿主节点的扑空计数器进行比较。如果它超出后者一定的阈值，就可以把该页迁移到那个远程节点去。（对每个页提供了 64 个计数器，如果系统中节点数大于 64，则 8 个节点共享一个计数器。）页迁移的代价一般都很高，通常抵销了进行迁移的好处。其代价之所以这么高的一个主要原因在于要改变所有访问该页的处理器的 TLB 中虚实地址的映射，而不是由于页传送（用 Hub 的块传输引擎传送一个 16 KB 的页大约需要 25 ~ 30μs）。迁移一个页将保持虚地址不变，但是改变了物理地址，因此处理器页表的旧映射现在是无效的。当页表项被改变时，重要的是处理器的 TLB 表项的高速缓存版本应被作废（很像第 6 章讨论过的 TLB 击落）。事实上，由于我们不知道哪个处理器在其 TLB 中缓存了该页的映射，必须向所有的处理器发送一个 TLB 作废消息。处理器被中断，发出作废的处理器必须等到最后一个处理器也做出了响应，才能更新页表项并继续执行。除了页传送本身的代价之外，这个过程一般需要 100 μs 以上的时间。

610

为了减少这个代价，Origin 采用了它的第 7 种目录状态，即毒化状态所支持的分布式惰性 TLB 作废机制。其思想是页移动时不作废 TLB 项，而是当该处理器下一次访问该页时才将其 TLB 项作废。这样做不仅把作废所有 TLB 的时间从管理迁移的处理器的关键路径中移走，而且只是把接着访问该页的处理器的 TLB 项作废。让我们看看这是如何工作的。为了

迁移一个页，块传输引擎用特殊的“非高速缓存的排他读”请求从源页位置读取所有的高速缓存块。这种请求返回数据的最新的一致的副本，并且作废任何存在的高速缓存副本（像一个正规的排他读请求一样），但是它也导致用块的最新版本更新目标主存，并将目录置为毒化状态。迁移本身只花费了块传输的时间。当处理器接着试图使用其过时 TLB 项访问旧的物理页的块时，它会在高速缓存中扑空，并且发现目录中块处于毒化状态。这时，毒化状态导致发出请求的处理器看到一个总线错误。对于该总线错误的特殊的操作系统处理例程将使页表处理器的 TLB 项作废，这样处理器再次访问时可以从页表中获得新的映射。当然，系统在某个时刻会回收旧的物理页以避免存储空间的浪费。一旦块传输完成，操作系统每个时间片作废一个 TLB 项，这样在一个固定的时间后，旧页可以被移到空表中。

3. 对同步的支持

Origin 提供了两种类型的同步支持。首先，R10000 处理器的加锁载入－条件存储（LL-SC）指令可以有效地合成同步操作，这一点前面已经提到过。其次，如果有多个处理器竞争更新一个存储单元，比如全局的计数器或一个栅障，可以提供非高速缓存的 fetch&op 原语。这些 fetch&op 操作在主存中执行，并不在高速缓存中复制块，这样后续的试图更新该单元的节点就不需要从前一个写入者的高速缓存中去取它。当同一个节点反复访问共享的（同步）变量时，使用可高速缓存的 LC-SC 效果更好，当不同的节点以交错或者竞争的方式进行更新，则使用 fetch&op 的效果更好。非高速缓存的读写操作对同步事件的生产者-消费者型的通信也有帮助，因为在通往宿主节点的路上，生产者和消费者的事务甚至可能会重叠。但是，在远程非高速缓存的单元上踏步等待可能产生大量的流量。

611

8.5.5 Origin2000 硬件概述

前面关于协议的讨论使我们了解了扁平的、基于存储器的目录协议是如何用网络事务和状态转换实现的，正像基于总线的协议是用总线事务和状态转换实现的一样。现在让我们把注意力集中到实现这一协议的 Origin2000 的实际硬件上来。本节概述了系统硬件的组成，随后深入考察 Hub 控制器是怎么样实现的（见 8.5.6 小节）。最后，在 8.5.7 节讨论机器的性能。（只对协议感兴趣的读者可以跳过本节的剩余部分而不会失去连贯性。）

除了由一个系统总线连接的两个 MIPS R10000 处理器之外，Origin2000 的每个节点包含机器主存的一部分（每节点 1～4 GB）、一个 Hub（它是通信/一致性控制器和网络接口的合成）以及一个被称为 Xbow 的 I/O 接口。除 Xbow 外的所有组成部分都在一块“16×11”的印刷电路板上。节点中每个处理器都有它自己独立的 L_1 和 L_2 高速缓存，L_2 高速缓存的配置可以是 14 MB，块大小为 128 字节，采用两路组相联。主存的每个块对应一个目录项。存储器采用 4～32 路交叉访问，根据插入的存储模块数来决定（模块内 4 KB 粒度 4 路交叉，模块间 512 MB 粒度 32 路交叉）。系统可以有多达 512 个这样的节点，即 1 024 个处理器。对 195 MHz 的 R10000 处理器而言，每个处理器的峰值性能为 390 MFLOPS 或者 780 MIPS（每个时钟周期 4 条指令），因此最大的机器配置总的峰值性能几乎可达 500 GFLOPS。连接两个处理器的 SysAD 总线的峰值带宽为 780 MBps。Hub 和存储器的连接带宽也是如此。存储器本身的数据带宽约为 670 MBps。Hub 与板外网络路由器芯片和 Xbow I/O 接口的连接带宽分别为 1.56 GBps，采用相同的链路技术。节点板的详图如图 8-19 所示。

Hub 芯片是机器的心脏。它位于节点的系统总线上，连接着处理器、本地存储器、网络

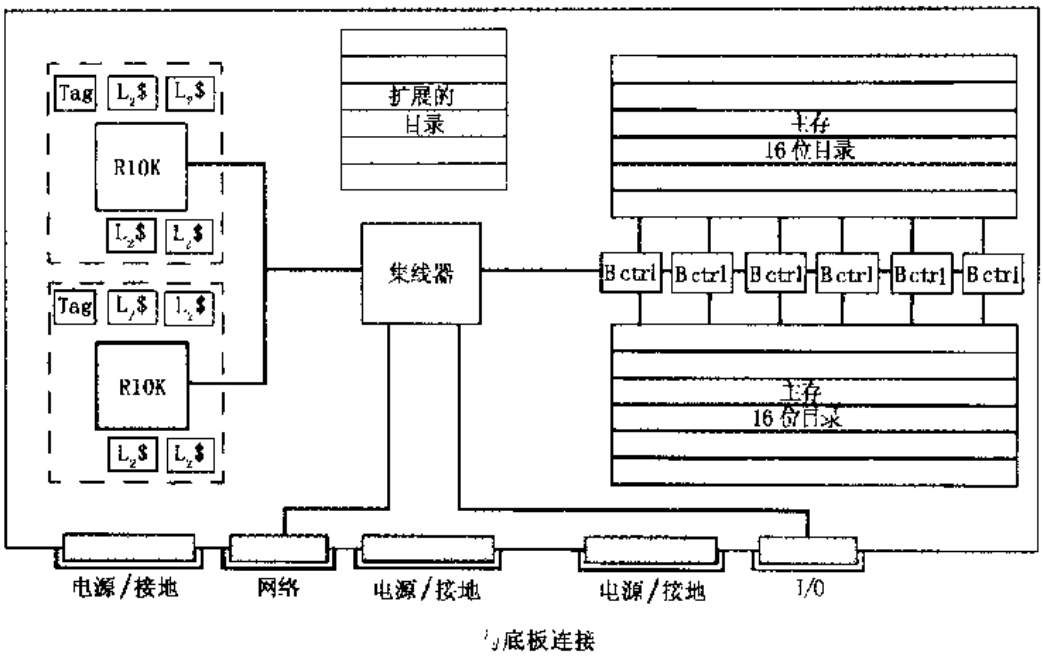


图 8-19 Origin 多处理器的节点板。"L_2 $"表示二级高速缓存芯片，"B Ctrl"表示存储器体控制器

和 Xbow，这些部件都通过 Hub 相互通信。对本地存储器或远程存储器的所有高速缓存扑空都要通过 Hub（它实现一致性协议），所有的非高速缓存操作也要经过 Hub。Hub 的集成度很高，采用 500-k 的门标准单元设计，0.5μm 的 CMOS 工艺。它为节点的两个处理器各设置了一个未决事务缓冲器（每个处理器允许有 4 个未决的请求）、一对支持以系统总线全部带宽进行存储器块复制和填充操作的块传输引擎、网络接口、SysAD 总线、存储器/目录和 I/O 子系统。Hub 还实现前面讨论过的存储器的非高速缓存的 fetch&op 指令和页迁移。

互连网络在机器的处理器数小于 64 个时采用超立方体拓扑结构，但是当处理器个数超过 64 时采用另外一种叫做胖立方体的拓扑结构。（该拓扑结构在第 10 章讨论）每个路由器提供 6 条链路。网络链路具有高带宽（每条链路总带宽 1.56 GBps，双向）、低时延（经过一个路由器点对点时延 41 ns），链路长可达 3 英尺，每条链路支持 4 个虚拟通道。虚拟通道将在第 10 章介绍，现在，我们可以认为机器拥有 4 个独立的网络，每个拥有实际物理链路带宽的四分之一。一个虚拟通道用作请求网络事务，另一个用于响应。其他两个用于缓解拥塞和高优先级事务的传输，此时可按点对点的次序，或者也可以用作 I/O。

Xbow 芯片连接 Hub 和其他的 I/O 接口。它本身被实现为具有 8 个端口的交叉开关。典型地，两个节点（Hub）连接到一个 Xbow 上，经过 Xbow 后连到 6 个外部的 I/O 卡上，如图 8-20 所示。Xbow 和路由器芯片（称为 SPIDER）很相似，但是具有较为简单的缓冲和仲裁，这样就可以在芯片上容纳 8 个接口而不是 6 个接口。仲裁器还支持对特定设备的带宽预留，因此可以支持像视频 I/O 这样的实时要求。像图形卡这样的高性能 I/O 卡直接连接到 Xbow 的端口上，但是，大多数其他的端口还是通过桥或者 I/O 总线连接，这样允许插入多个卡。任意一个处理器都可以访问机器中的任何一个物理 I/O 设备，可以通过对特殊 I/O 地址空间的非高速缓存的访问，或者通过高速缓存一致的 DMA 操作。一个 I/O 设备也可以和系统中任意一个存储器交换数据，而不是局限于通过 Xbow 与其直接相连的节点上的存储器，这样充分利用了共享地址空间的优点。处理器和相应的 Xbow 之间的通信由 Hub 和网络路由器透

612
~
613

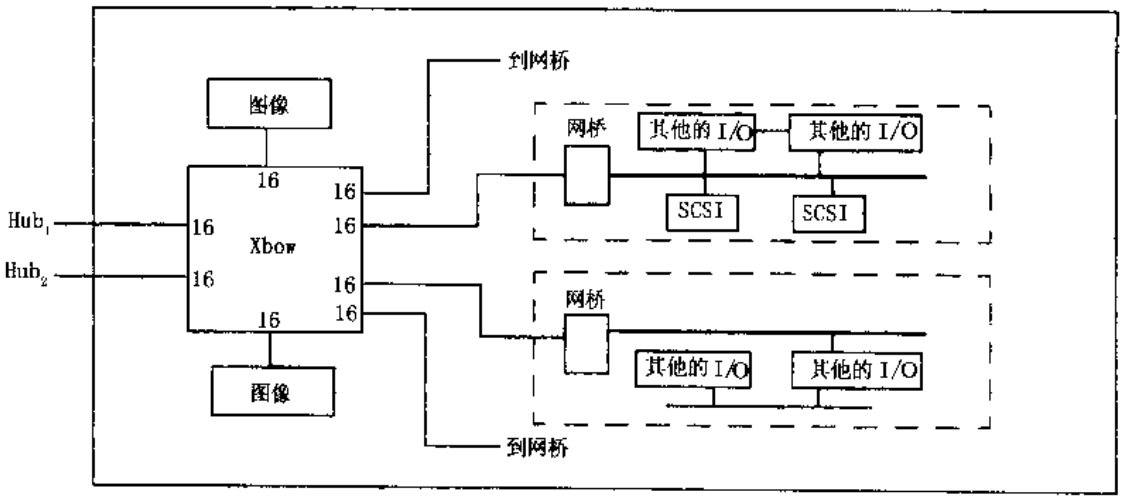


图 8-20　由两个节点共享的典型的 Origin I/O 配置。高性能的图形设备直接连到 Xbow 上，其他的 I/O 设备连到 I/O 总线上，I/O 总线经过桥与 Xbow 相连

明地处理。所以，和存储器一样 I/O 设备也是物理上分布，但全局可访问，因此 I/O 分布的局部性也是性能问题而不是正确性的问题。

8.5.6　Hub 的实现

通信辅助部件，即 Hub，必须具有实现一致性协议的某些基本能力。它必须能够看到所有的高速缓存扑空、同步事件和非高速缓存操作；在继续处理其他发出的请求和进入的事务的同时，跟踪未决的请求；保证接收来自网络的响应；作废高速缓存块；以及为了获得数据对高速缓存的干涉操作。对于来自不同部件流过它的所有不同类型的事务的动作和相互依赖性，它还必须予以协调，并且实现必要通路和控制。因此，设计这样的控制器是一种挑战。本小节简要描述 Orign2000 中所使用的 Hub 控制器的主要成分，并指出在一致性协议的实现中采用的主要特性。了解 Hub 的实际数据和控制通路的进一步的细节，以及用来实际控制消息间交互的机制，对于理解如何实现可扩展的高速缓存一致性协议是有用的，读者可以阅读其他文献（Singh 1997）。

[614]

Hub 划分成 4 个主要的接口，各用于它所连接的每一种类型的外部实体：处理器接口或称 PI、存储器/目录接口或称 MI，网络接口 NI 和 I/O 接口 II（见图 8-21）。接口之间通过芯片上的交叉开关相互通信。每个接口分成几个主要的结构，包括对发往/来自其他接口以及发往/来自外部实体的消息进行缓冲的 FIFO 队列。设计中的一个关键的特性是接口对外部实体屏蔽其他接口和实体的细节，反之亦然。例如，PI 对其他部分隐藏处理器，因此任何其他的接口只需知道 PI 的行为，而不知道处理器和 SysAD 总线本身的行为。让我们简要地讨论一下 PI、MI 和 NI 的结构，以及这些接口提供的屏蔽的例子。

1．处理器接口

在接口中，PI 的控制机制最复杂，因为它必须跟踪未决的协议请求和进入的响应，并且必须对它们进行匹配。PI 一侧与两个 R10000 处理器的存储器（SysAD）总线连接，另一侧和输入输出的 FIFO 队列连接，这些队列将它与另一端的各个 Hub 接口相连接（见图 8-21）。通过提供独立的逻辑和分级缓冲，每个物理的 FIFO 逻辑上被分成独立的请求和响应“虚拟 FIFO”。此外，PI 本身还包含三对一致性控制缓冲器，它们被用于跟踪未决的事务，控制消

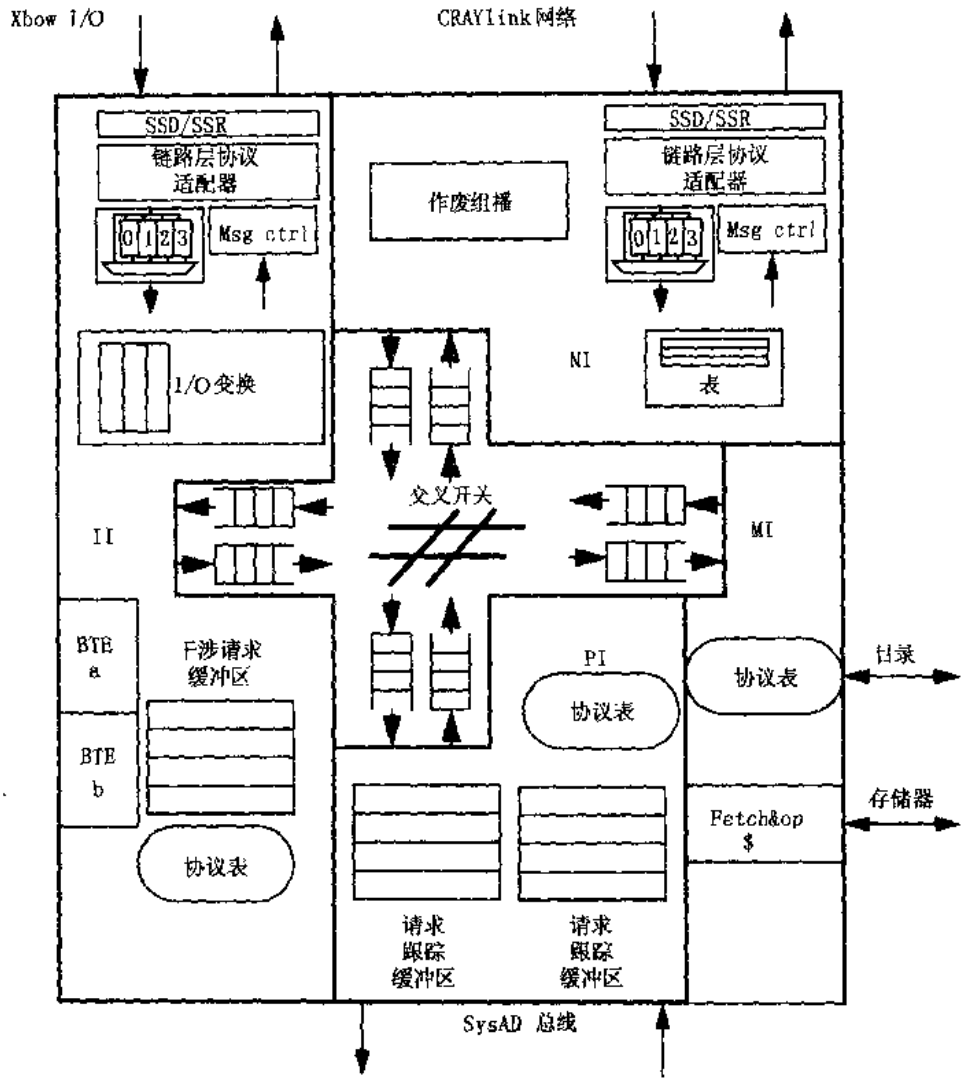


图 8-21 Hub芯片的布局。中央的交叉开关连接4种不同接口的缓冲器。从左下角开始顺时针看，BTE是块传输引擎。左上角是I/O接口或记作II（SSD和SSR对发往和来自I/O接口的信号进行转换）。接下来是网络接口（NI），包括路由表。右下角是存储器/目录接口（MI），最下方是处理器接口（PI）及其请求跟踪缓冲器

息流通过PI，实现协议所指定的消息间的相互作用。但是，这些缓冲器并不保存消息本身。有两个读请求缓冲器（RRB），跟踪来自每个处理器的未决读请求；有两个写请求缓冲器（WRB），跟踪未决的写请求；还有两个干涉请求缓冲器（IRB），跟踪进入的作废和干涉请求。对这三组缓冲器的访问是通过单条总线，因此所有的消息需要竞争访问它们。

记录在一种类型的缓冲器中的消息也可能需要查询其他类型的缓冲器，以便检查是否有来自处理器对同一地址的冲突访问或干涉。例如，向外发出的读请求对WRB执行相关查找，看看是否有对同一地址的未决的回写。如果有一个冲突的WRB项，读请求就不放到PI的发送读请求的FIFO队列中去，而是在RRB项中设置一个位，表明当对应的WRB项释放时，应该重新发出读请求（也就是说，当回写被确认或根据每个协议被进入的作废撤销时）。当来自节点中的处理器或者来自其他接口的结束响应到达时，也要查找缓冲器来关闭其中对应的未决PI事务。因为事务关闭的次序是不确定的，因此一个新的事务必须进入任何可用的

缓冲单元，这样，这些跟踪缓冲器的实现就是全相联的，而不是先进先出（FIFO）的了（保存实际消息的队列是 FIFO 的）。缓冲器的查找决定了 PI 是否应该对处理器或者其他接口发出一个请求。

615 ~ 616

PI 是接口提供屏蔽的很好例子。如果处理器（或高速缓存）对进入的干涉提供数据作为应答，正是 PI 的输出 FIFO 把应答扩展成协议要求的两个响应，一个是给宿主节点的共享的回写修正消息，另一个是给请求者的响应。并不需要修改处理器，让它产生两个应答。另一个例子是用于跟踪和匹配进入和送出的请求和响应的机制。以任一方向通过 PI 的请求被赋予一个请求号，响应也携带这些请求号。但是，处理器本身并不知道请求号；PI 的任务是，当它把一个进入的请求（干涉或作废）传递给处理器时，它保证将处理器的响应与未决的干涉/作废匹配，而处理器无需与请求号打交道。

2. 存储器/目录接口

在 MI 和 Hub 的交叉开关之间也有 FIFO。从 Hub 的交叉开关到 MI 的 FIFO 将消息头与数据分离，以便为当前消息服务时，目录可以检查下一个消息的头；这允许以存储器峰值带宽对写操作流水执行。MI 还含有一个目录接口，一个处理器接口和一个控制器。目录接口包含决定所发生的协议动作的逻辑和表，因此实现了一致性协议。它还包含生成输出消息头的逻辑，而存储器接口包含生成输出消息数据的逻辑。存储器和目录 RAM 都有各自的地址和数据总线。一些消息像到达宿主节点的修正消息可能不访问存储器而只访问目录。

接收到一个读请求，宿主推测性地向存储器发出读操作，同时开始目录操作。在存储器数据读出之前的一个时钟周期获得目录状态，控制器用它（外加消息类型和发出者）来查找目录协议表。这个硬接线实现的表指出控制器应发生的动作和应发送的消息。目录块把后一个信息发送给存储器接口，在那里装配消息头，并将它和从存储器返回的数据一起插入输出 FIFO 中。目录查找本身是一个读-修改-写操作。为此，MI 提供对存储器块部分写的支持，并用一个只有一项的合并缓冲器保存从存储器读出的字节，直到它们被写回。最后，为了加速用于同步的对存储器的 fetch&op 访问，MI 包含了一个 4 项的 LRU fetch&op 高速缓存，用于保存最近的 fetch&op 变量的数据，因此，可以尽量地避免存储器或者目录的访问。这将最佳情况的存储器 fetch&op 串行时间降低为 41 ns，大约 4 个 100 MHz 的 Hub 时钟周期。

617

3. 网络接口

NI 提供 Hub 的交叉开关和该节点的网络路由器的接口。路由器和 Hub 内部使用不同的数据传输格式、不同的协议和不同的速率（Hub 的频率为 100 MHz，而路由器的频率为 400 MHz），因此 NI 的一个主要功能就是在两者之间进行转换。朝向路由器这侧，NI 实现了一个流控机制，以避免网络拥塞（Singh 1997）。NI 和网络之间的 FIFO 也实现分开的虚拟的请求和响应 FIFO，因此实现了独立的虚拟网络。输出 FIFO 包含一个作废目的节点地址的生成器，它利用作废节点的位向量，并生成发给那些节点的一个个消息；输出 FIFO 还包含一个路由表和虚拟通道选择逻辑，路由器根据源节点和目的节点预先决定路由。

8.5.7 性能特征

前面说过，Origin2000 系统的峰值硬件带宽为：SysAD 总线是 780 MBps；本地存储器是 670 MBps；节点到网络的每个方向是 780 MBps。对高速缓存块的一个事务占用宿主节点 Hub 的大约 20 个 Hub 周期（约为 40 个处理器周期），根据后续访问的目录页是否位于同一个目

录 RAM 体或根据后续事务的精确模式，这个占用时间可能在 18 个周期到 30 个周期之间变化。存储器操作的时延取决于很多因素，比如说操作的类型，宿主是本地的还是远程的，数据当前缓存在什么地方，处于什么状态，访问路径上对资源的竞争如何等。时延可以用微基准测试程序来进行度量。首先考察微基准测试程序关于时延和带宽的结果，然后是 6 个并行应用的性能和扩展性。

1. 微基准测试程序的特征

与 SGI Challenge 所使用的 MIPS R4400 处理器不同，Origin 的 MIPS R10000 处理器采用动态调度，不会在读扑空上暂停。这样使得读时延的测量更加困难，因此产生了一个有趣的方法学的问题。例如，我们不能通过简单地执行第 4 章的微基准测试程序，读一个跨距大于高速缓存块尺寸的数组的元素，来测量读扑空的无负载时延。因为扑空发生在不同的单元，后续的扑空会相互重叠，处理器看不到它完整的时延。相反，该微基准测试程序将给出系统在一个处理器发出连续的读扑空时所能提供的吞吐量的度量。吞吐量是重叠后仍剩余的时延的倒数，我们称这种时延为流水时延。

为了测量完整的时延，我们需要保证相继的操作相互依赖。为了做到这一点，可以用一个沿链表追赶指针的微基准测试程序：在前一次的读指针的操作结束之前，处理器得不到下一个读的地址，这样读不能重叠。但是，我们发现这对决定无负载时延有点过于悲观了。其原因是处理器能实现关键字重启，也就是说，一旦所读的字返回到处理器，它就可以使用读返回的值，而不必等高速缓存块的其余部分载入高速缓存。在指针追赶微基准测试程序中，下一个读请求会在前一个块被载入之前发出，并且将和块的其余部分的载入竞争高速缓存的访问。从该微基准测试程序得到的时延，其中包括了这种竞争，可以称之为背靠背时延（正好在前一次读结束时发出下一次读扑空）。要避免连续访问之间的竞争，需要在读扑空之间插入一些计算；该计算应该依赖于正在读出的数据，因此它不可以和读扑空并行地执行，也不应该在两次扑空之间访问高速缓存。其目的就是要让计算的时间与读扑空后将缓冲块的其余部分载入高速缓存所花费的时间重叠，这样下一次读扑空就不必在高速缓存访问上等待。当然，重叠计算的时间必须从微基准测试程序所花费的时间中扣除，这样才得到真实的无负载的读扑空时延，这里假设采用关键字重启的做法。我们能把它称之为真实的无负载的时延。表 8-1 显示了在 Origin2000 上所测得的背靠背时延和真实的无负载的时延。只有一个处理器执行微基准测试程序，但是被访问的数据分布在不同数量的处理器的存储器中。背靠背时延通常要长出 13 个 SysAD 总线周期（133 ns），因为 L_2 高速缓存块的尺寸（128 B）比 L_1 高速缓存块的尺寸（32 B）要长 12 个双字的长度，总线往返还要占一个周期。

618

表 8-1　不同大小的系统上的背靠背时延和真实无负载的时延

扑空被满足的地方	网络路由器遍历	背靠背时延（ns）	真实的无负载的时延（ns）
L_1 高速缓存	0	5.5	5.5
L_2 高速缓存	0	56.9	56.9
本地存储器	0	472	329
4P 远程存储器	1	582	449
8P 远程存储器	2	775	621
16P 远程存储器	3	826	702

注：第一列显示在扩展的存储器层次结构中，扑空是在何处满足的。例如，对 8P 的情况，在一个包含 8 个处理器的系统中，扑空是在离请求者最远的节点满足的。对于给定的 Origin2000 的拓扑结构来说，意味着在这种情况下要经过两个网络路由器。

表 8-2 列出了被访问块的不同的初始状态所对应的背靠背时延（Hristea、Lenoski 和 Keen1997）。回忆一下，当块在目录中处于未拥有或共享状态时，拥有者节点是宿主节点；而当块处于排他状态时，拥有者节点是具有高速缓存副本的节点。宿主和拥有者都是本地节点情况下（即如果块由主存拥有，其他的处理器在同一节点）的真实无负载时延是 338 ns（未拥有状态）、656 ns（干净的排他状态）和 892 ns（被修改状态）。注意，在这个微基准测试程序中不会碰到和来自其他处理器的操作的竞争，而在真实负载下的时延要大一些。 619

表 8-2　块的不同初始状态所对应的背靠背时延（以 ns 为单位）

宿主	拥有者	块的状态		
		未拥有的状态	干净的排他状态	修改的状态
Local	Local	472	707	1 036
Remote	Local	704	930	1 272
Local	Remote	472	930	1 159
Remote	Remote	704	917	1 097

注：第一列指出块的宿主是否是本地的，第二列指出当前拥有者是否为本地的，后三列给出处于各种状态下的块的时延。当然，对未拥有状态应该忽略拥有者节点。

2．应用的加速比

图 8-22 显示了在 32 个处理器的 Origin2000 上的 6 个并行应用的加速比，每个应用采用两种问题规模。我们看到，大部分应用的加速比良好，特别是问题规模足够大时。对问题规模的依赖关系在 Ocean 和 Raytrace 这样的应用中表现的特别明显。加速比不太好的例外是 Radiosity，以及 Radix。对于 Radiosity，即使是较大的那种问题规模对于这种规模和能力的机器来说也太小了。我们可以期望对更大的问题可以看到更好的加速比。对于Radix，问题出

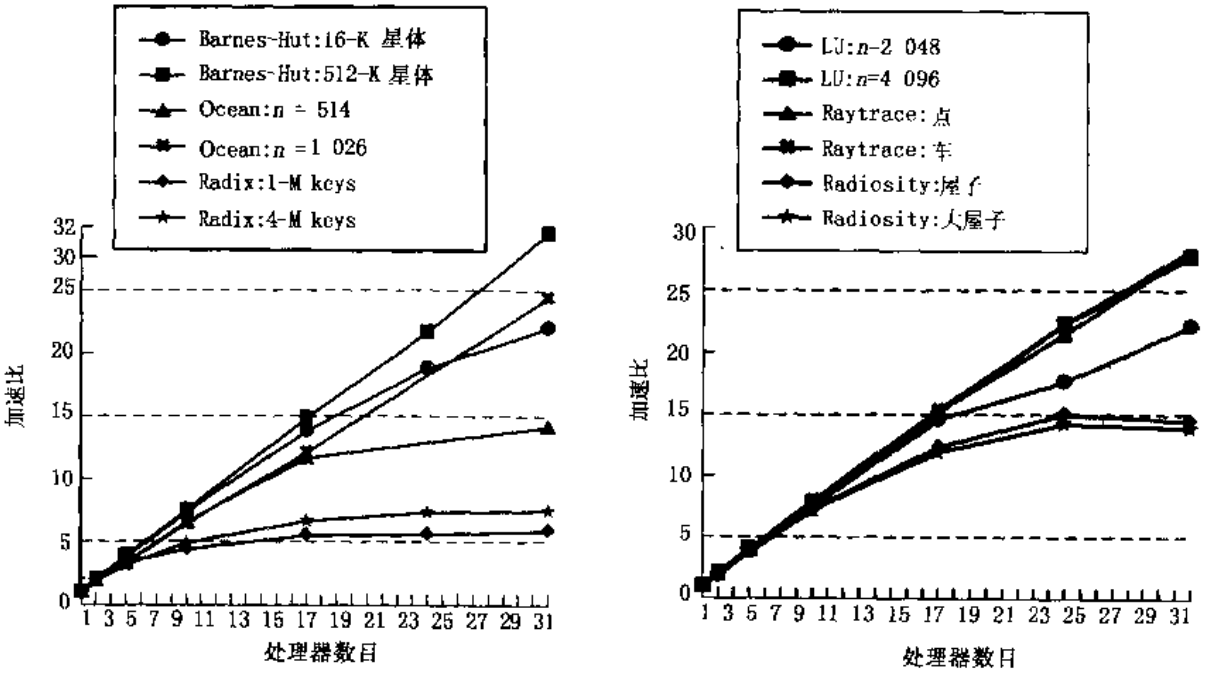


图 8-22　Origin2000 上并行应用的加速比。每一种应用有两种不同的问题规模. Radix 排序程序的扩展性不是很好，Radiosity 应用受到输入的问题规模的限制。其他的应用在采用合理的大问题规模时，加速比都很好

自在置换阶段高度分散的、突发的写入模式。这些写大多数是针对远程分配的单元和目录之间大量的请求、作废和确认，以及对它们产生的应答，导致了在 Hub 和存储器中严重的热点竞争。运行更大问题只是缓解伪共享的情况，因为在这个阶段，除了数据置换之外，没有其他的计算，通信与计算的比本质上与问题的规模无关。事实上，当处理器的键字分区不能容纳于高速缓存时，情况变得更坏，这时还会发生频繁的回写事务。对于像 Radix 这样的应用（比如没有给出的 FFT）来说，它们表现出全部对全部的突发通信，两个处理器共享一个 Hub 和两个 Hub 共享一个路由器的事实也导致这些资源的竞争，尽管它们有着高的峰值带宽（Jiang 和 Singh 1998）。对于这些应用来说，如果机器每个 Hub 和每个路由器上都只有一个处理器的话，其性能会更好。但是，资源的共享确实降低了成本，而且在其他应用情况下性能也还不错。在第 3、4 章中我们以每个处理器为基础，将这种机器的执行时间分成各个组成部分，使我们更清楚地了解到时间都花费在什么地方。

3. 扩展

图 8-23 显示了在 Origin2000 上不同扩展模型下的 Barnes-Hut 星系模拟的加速比。这些结果和第 6 章中在 SGI 的 Challenge 机上得到的结果相当类似，尽管扩展到更多的处理器，但那里的分析基本适用。对于像 Ocean（未示出）这样的应用，重要工作集与每个处理器的数据集成比例，像 Origin2000 这样的机器在比较扩展模型时显示了有意思的效果，这里，我们是从工作集无法容纳于单处理器上的高速缓存中这样的问题规模开始的。在 PC 和 TC 扩展下，每个处理器数据集的尺寸随着处理器的数量增加而减小。所以，尽管通信对计算的比增加了，一旦工作集开始能容纳于高速缓存时我们会观察到超线性的加速比（因为当工作集能容纳于高速缓存时，节点内的性能会变得更好）。在 MC 扩展下，通信对计算的比不变，但是每个处理器的工作集的大小也不变。结果是，尽管对通信体系结构的要求更倾向于 MC 扩展而不是 TC 或 PC 扩展（工作集产生的容量型扑空几乎全是本地的），其加速比却不是很好，因为工作集和高速缓存相匹配时所获得的好处再也看不到了。而且，甚至局部的容量型扑空也占用 Hub 和存储器，导致了竞争。

620
~
621

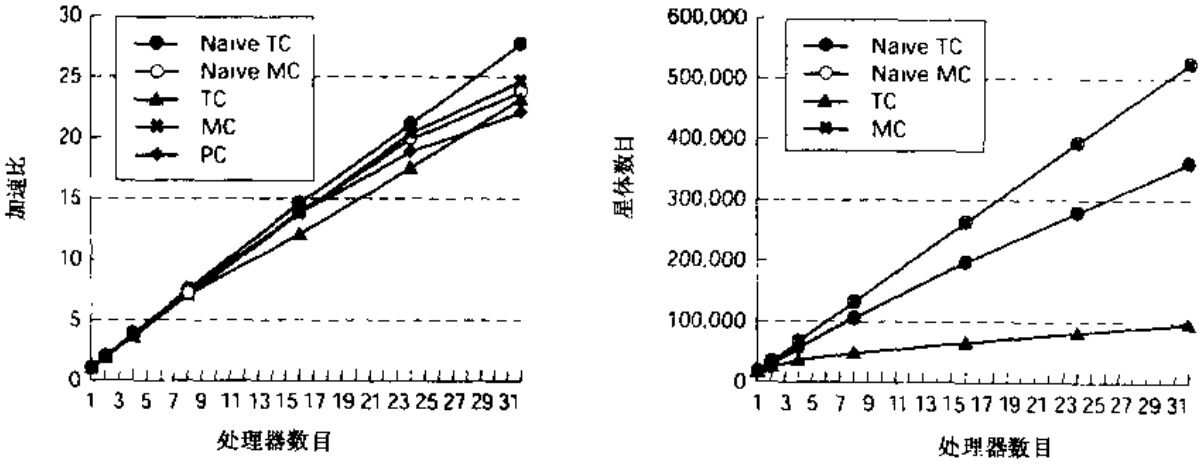


图 8-23　在 Origin2000 上不同扩展模型的 Barnes-Hut 星系模拟所模拟的星体数量和加速比的扩展情况。和第 6 章中基于总线的机器的结果一样，加速比在各种扩展模型下都很好，所能模拟的星体的数量在实际的 TC 扩展下比在 MC 或简单 TC 扩展下增长得慢得多

8.6 基于高速缓存的目录协议：Sequent 的 NUMA-Q

在我们的第二个案例分析中所描述的扁平的、基于高速缓存的目录协议是 IEEE 的标准可扩展一致接口（SCI）协议（Gustavson 1992）。作为这个协议的一个案例，将考察 Sequent Computer System，Inc. 的 NUMA-Q 机，这种机器面向商业工作负载比如数据库、事务处理（Lovett and Clapp 1996）。该机器高度依赖于第三方商品化硬件，采用市售的 Intel 的 SMP 系统作为处理节点，采用市售的 I/O 链路和 Vitesse Semiconductor Corporation 公司的 DataPump 网络接口在节点和网络间传送数据。仅有的定制部件是用于实现 SCI 目录协议的 IQ 链路板。在 Convex Exemplar 系列机中使用了类似的目录协议（采用了很多的定制部件）（Convex Computer Corporation 1993；Thekkath et al. 1997），它和 SCI 的 Origin 一样，瞄准科学计算。 622

NUMA-Q 是一组由环结构的高速链路互连的同构的处理节点（如图 8-24 所示）。每个处理节点是一个便宜的 Intel 的基于总线的 4 处理器，配备 4 个 Intel Pentium Pro 微处理器，这是一个采用大批量生产的 SMP 作为更大系统的构造模块的例子。出自 Data General（Clark and Alnes 1996）和 HAL Computer Systems（Weber et al. 1997）的系统也使用 Pentium Pro 4 处理器作为它们处理节点，前者在 4 处理器节点之间采用类似于 NUMA-Q 的 SCI 协议，后者采用由 Stanford 的 DASH 协议而来的基于存储器的协议。（在 Convex Exemplar 系列中，SCI 协议连接的节点不是基于总线的，而是小型的基于目录的多处理器，内部由一个不同的目录协议保持一致。）在第 1 章描述过 4 处理器的 SMP 节点（见图 1-17），在这里就不进一步讨论了。

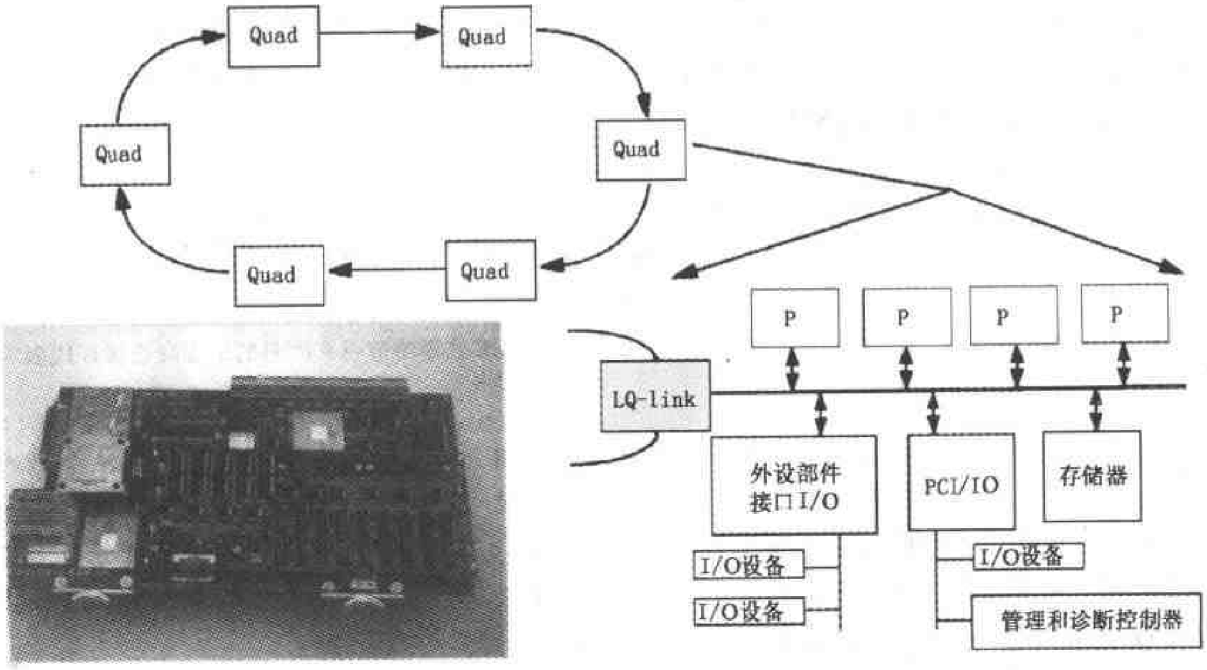


图 8-24 Sequent 的 NUMA-Q 多处理器的框图。该图显示了机器的高层组织结构，跨节点和在节点内。照片显示了一块 IQ 链路板。照片引自 Sequent Computer System，Inc

每个 4 处理器节点的 IQ 链路板插在节点的存储器总线上，取代了 SGI Origin 中的 Hub。除了目录逻辑和存储以及 4 处理器节点总线和网络之间的数据通路之外，它还包含一个大的（可扩展的）32 MB、4 路组相联的远程访问高速缓存，用来缓存从远程存储器取到节点的

块。在下文中，这个远程访问高速缓存被称作远程高速缓存，在跨节点的 SCI 目录协议中代表所在的 4 处理器节点。它是 4 处理器节点中惟一对协议可见的高速缓存；每个处理器的高速缓存通过节点内的总线侦听协议与远程高速缓存保持一致。在很大程度上，目录协议与节点内处理器的数量以及总线协议没多大关系。节点内的远程高速缓存和处理器高速缓存之间保持包容关系；这样，如果从远程高速缓存中替换掉一个块，该块也会在处理器高速缓存中作废；如果一个块在处理器的高速缓存中被置为修改状态，远程高速缓存也要反映这个状态。远程高速缓存中块的尺寸是 64 字节，这因此也是通信和跨节点一致性的粒度。

[623]

8.6.1 高速缓存一致性协议

虽然在 Sequent 的 NUMA-Q 机使用了两个相互作用的协议，这一节的注意力集中在跨远程高速缓存的 SCI 目录协议，忽略四处理器节点的多处理器性质。在 8.6.5 节，将讨论在 4 处理器节点内侦听的 MESI 协议的交互作用。

1. 目录结构

SCI 的目录结构采用扁平的、基于高速缓存的分布式双向链表方案，该结构在 8.2.3 节描述过，如图 8-8 所示。每一个块有一个共享者的链表，指向该链表头的指针存储在对应存储器块的宿主节点的主存里。链表中的一个项对应一个 4 处理器节点的远程高速缓存。远程高速缓存和链表的前向和后向指针都存储在那个节点的 IQ 链路板的同步 DRAM 中。图 8-25 给出了链表的一个简化的表示。第一个元素（节点）叫做链表的头，最后一个元素叫做链表的尾。头节点对它的高速缓存的块有读写许可权，而其他的节点只有读的许可权（称为成对共享的特殊情况是个例外，我们将在 8.6.3 节简要地讨论）。节点中指向链表尾方向的相邻节点的指针叫做前向或下游指针，相反方向的叫做后向或者上游指针。下面看一下跨节点的 SCI 一致性协议是如何使用这种目录表示的。

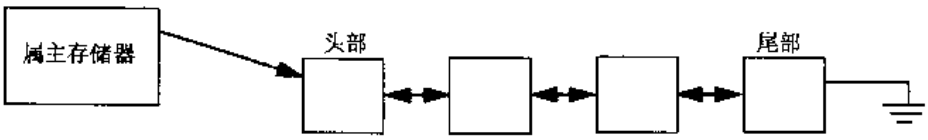


图 8-25 SCI 共享链表、NUMA-Q 机的链表的每个元素是一个多处理器节点，由它的远程高速缓存代表

2. 状态

因为处理器的高速缓存对目录协议是不可见的，并且，与 Origin 机不一样，块从不进入其宿主节点的远程高速缓存，所以，NUMA-Q 机的目录协议确实不记录在宿主节点被高速缓存的副本。保持宿主存储器中的副本与那些被高速缓存的副本一致是总线协议的任务。主存储器中的一个块可以处于三个目录状态之一，这些状态的名字由 SCI 协议如下定义。这些状态与 Origin 协议的目录状态很类似，但不完全相同。

[624]

- 宿主。系统中没有远程节点含有该块的副本（当然，4 倍于宿主节点本身的处理器高速缓存可能含有副本，因为它对 SCI 一致性协议不可见，而是由节点中的总线协议管理的）。这和 Origin 中的未拥有目录状态类似。
- 新鲜。一个或多个远程高速缓存中可能有只读的副本，存储器中的副本是有效的。它类似于 Origin 中的共享状态。
- 过时。另一个远程高速缓存包含一个可写的（排他或脏）副本。本地节点不存在有

效的副本。这类似于 Origin 中的排他状态。

考虑一下远程高速缓存中块的缓存状态。虽然4处理器节点中的处理器高速缓存采用标准的 MESI 稳定状态，管理远程高速缓存的 SCI 方案却有大量可能的高速缓存状态。事实上，使用了7位来表示远程高速缓存中的块状态，该标准描述了29个稳定状态和许多未决（忙）状态或过渡状态。我们可以认为每个稳定状态有两个部分，反映在状态的命名结构中。第一部分描述了块的高速缓存项在共享链表中的位置。可以是惟一 ONLY（单节点链表）、头 HEAD、尾 TAIL 或中间 MID（这意味着在多节点的链表中，它既不是头，也不是尾）。第二部分描述了被缓存块的实际状态。包括脏（dirty）（修改过并可写）、干净（clean）（未修改过，与存储器内容相同，但可写，和 MESI 中的独占状态类似）、新鲜（fresh）（在通知存储器之前，数据可读但不可写）、副本（copy）（未修改，可读）；还有其他一些状态。一个完整的描述可以在 SCI 标准的文档中找到（IEEE Computer Society 1993）。以后，我们会碰到几个这样的状态（如头-脏（HEAD-DIRTY）、尾-干净（TAIL-CLEAN 等）.

SCI 标准定义了三个可以对分布的共享链表执行的基本操作。像读扑空、写扑空、回写和替换这些存储器操作都使用这三个基本操作实现.

1）链表构造：把一个新节点（共享者）加到共享链表的头部。

2）转出：从链表去掉一个节点，需要该节点与其上游和下游相邻节点通信，通知它们谁是它们的新邻居，以便它们更新自己的指针。

3）清除（作废）：链表头的节点可以清除或作废所有其他节点，从而导致单个元素的链表。只有表头节点可以发出清除操作。 [625]

SCI 标准还描述了三级复杂度逐渐提高的 SCI 协议。最小协议甚至不允许读共享，也就是说，同时只能有一个节点保存一个块的高速缓存副本。典型协议是大多数系统期望实现的。它允许读共享（多副本），提供对存储器中处于 FRESH 状态的数据的有效访问，以及有效的 DMA 传输和从错误中恢复的健壮性。最后，完全协议实现标准定义的所有可选功能，包括对仅两个节点间的成对共享的优化和在锁位上排队（QOLB）同步（后面讨论）. NUMA-Q 系统实现了典型协议，这就是我们所讨论的。下面看一下不同类型的存储器操作——读扑空、写扑空和替换（包括回写）——是如何处理的。在各种情况下，首先根据块的地址决定宿主节点。

3. 读请求的处理

假定读请求需要从节点内传出。就 SCI 协议来说，我们可以把这个节点的远程高速缓存看作是请求高速缓存。如果需要，请求高速缓存首先为块分配一个表项，并且把块的高速缓存状态置为未决忙；在这个状态下，它不会处理其他对该块的请求。（SCI 协议经常在请求者处把被高速缓存的块置于忙状态来保持对块的事务的原子性，方便串行化，很像 Origin 协议对目录的忙状态所做的那样；但是，我们将会看到，它不使用 NACK）。然后，它通过向宿主节点发送一个请求开始一个链表构造操作，把自己加到共享链表的表头。当宿主节点收到请求，它的块可能处于前面定义的三个目录状态之一：HOME、FRESH 或 GONE 。

如果目录状态是宿主（HOME），那么不存在高速缓存副本，存储器中的数据是有效的。当收到这样的读请求时，宿主把它的块更新为 FRESH 状态，并把它的头指针指向请求节点。然后宿主向请求节点发送数据，请求节点接到后将它的状态从未决（PENDING）变为惟一新鲜（ONLY_ FRESH），一个节点响应事务的所有动作都是原子的（一个没结束前不会开始

下一个），在任何情况下都遵循严格的请求-响应协议（与 Origin 不一样）。

如果目录状态是新鲜（FRESH），就已经存在一个共享链表，但是宿主中的副本也是有效的。宿主把它的头指针指向请求高速缓存，而不是链表以前的头节点。然后它向请求节点送回一个事务，包含了数据以及指向以前表头的指针。请求节点接收到这个事务后，转移到不同的未决状态，并向以前的表头送出一个事务，要求成为链表的新的表头（链表构造操作）。以前的表头对这个请求的反应是把它的状态从头新鲜（HEAD_FRESH）变为中间有效（MID_VALID）或从惟一新鲜（ONLY_FRESH）变为尾有效（TAIL-VALID），根据情况而定，改变它的后向指针以指向请求者，并且对请求者发送一个确认。当请求者接到确认时，把它的前向指针指向以前的表头，并把状态从未决变为头新鲜（HEAD FRESH）。若当请求到来时，以前的表头处于头新鲜（HEAD_FRESH）状态，事务和动作的顺序如图 8-26 所示。

626

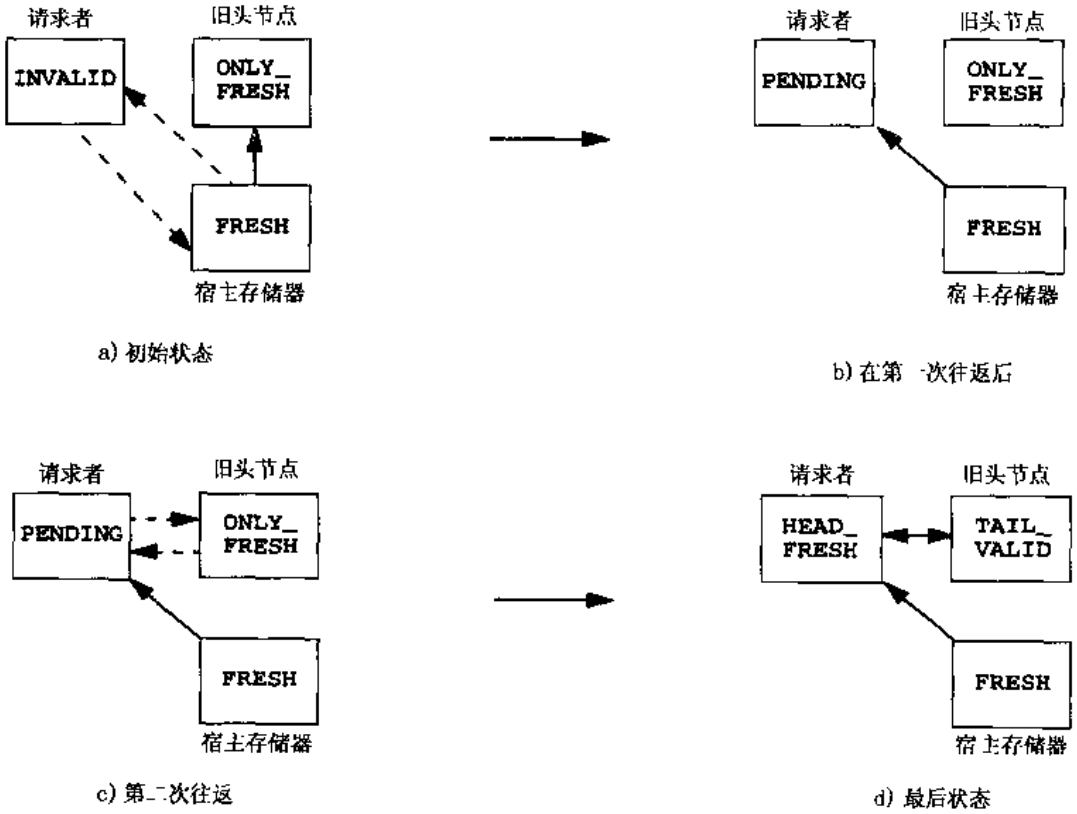


图 8-26 SCI 协议中读扑空的例子。本图显示了当初始情况下一个块在它的宿主节点处于 FRESH 状态时，对它的读扑空产生的消息和状态转换，在共享链表中有一个节点。实线是共享链表中的指针，虚线代表网络事务。空指针没有显示

如果目录状态是过时（GONE）的，在共享链表头部的高速缓存有该块的一个排他（干净或修改过）副本。现在，存储器不用数据应答，只是简单地停留在 GONE 状态，并把指向以前的表头的指针发回请求者。请求者进入一个新的未决状态并向以前的表头发送一个请求，请求得到数据并成为新的头节点（链表构造）。以前的头节点把它的状态从头部脏（HEAD_DIRTY）变为中间有效（MID_VALID）或从惟一脏（ONLY_DIRTY）变为尾有效（TAIL_VALID）（或无论什么合适的状态）；设置它的后向指针指向请求者，并向请求者返回数据。（数据可能必须从以前的头节点的处理器高速缓存中取出）。然后请求者更新它的副本，将它的状态设置为头部脏（HEAD_DIRTY），并把它的前向指针指向新的表头，这些动

作都是单个原子动作。注意，尽管访问是读，共享链表的头节点的状态停留在头部脏（HEAD_DIRTY）。这并不具有我们所熟悉的脏的标准含义，也就是说，头节点可以把数据写入而无需作废任何其他的高速缓存。这意味着它确实可以不和宿主通信就把数据写入高速缓存（甚至在送出作废之前），但是它必须作废共享链表中的其他节点，因为它们处于有效状态。

甚至在目录状态不是 GONE 时，读取一个 HEAD_DIRTY 的块也是可能的。例如，当请求节点希望很快会写那个块时。在这种情况下，如果目录状态是 FRESH，那么存储器给请求者返回数据，并返回指向共享链表的旧表头的一个指针，再把自己置为 GONE 状态。然后，请求者通过向旧表头节点发出请求把自己变成共享链表的头，并使自己进入 HEAD_DIRTY 状态。旧头节点把自己的状态从 HEAD_FRESH 变为 MID_VALID 或从 ONLY_FRESH 到 TAIL_VALID，表中的其他节点仍保持不变。 627

在上面这些情况中，宿主节点总是把请求者引向旧的头节点。旧头节点（叫做 *A*）在收到新的请求者（叫做 *B*）的请求时，可能处于未决状态，因为它本身可能对该块进行了一次未完成的存储器操作。这种情况不是靠在 *A* 中缓冲或否定回答这个请求来处理的，而是将共享链表向后扩展，形成*未决链表*（仍然是分布的）；也就是说，节点 *B* 确实物理地附加到链表的头部，但处于未决状态，等待真正成为表头。如果另一个节点 *C* 现在向宿主发送请求，它会被转给节点 *B*，也加入到未决链表中（现在宿主指向 *C*，这样后续的请求将被引向那里，以此类推）。在任何时候，我们把“真正的头”（这里是 *A*）简称为共享链表的头，把真正的头之前的链表部分叫做*未决的链表*，把最近加入未决链表的节点（这里是 *C*）叫做*未决链表的头*（见图 8-27）。当 *A* 完成它的操作脱离未决状态时，它把真正的头的状态传给 *B*，*B* 在完成其请求时，将“真正的头”状态传给 *C*。还要注意，与 Origin 不同，目录中没有未决或忙状态，只是简单的采取原子操作来改变状态和头指针，并把前一个状态/指针信息返回给请求者。当讨论正确性问题时，我们会讨论这一点。

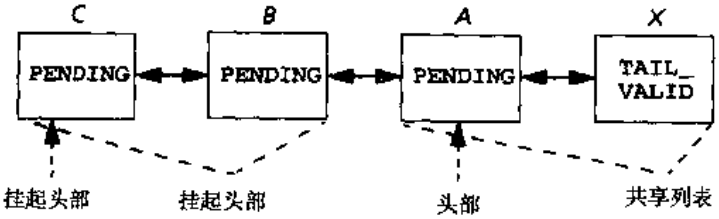


图 8-27 SCI 协议中的未决链表。未决链表是常规共享链表的延续（以相反方向）。真正的头节点（叫做头）和未决链表中的节点都处于未决状态

4. 写请求的处理

我们总是假设共享表的头节点拥有块的最新副本（除非头节点是处于未决状态）。所以，只允许头节点对块写，发出作废。当一个节点发生写扑空时，可能有三种情况。第一种情况，写入者已经在链表的头部，但它不是拥有惟一的被修改副本（可能有其他共享者）。它首先确保自己处于适当的状态，如果有必要，要与宿主通信（在这个过程中确保宿主的块已经处于过时状态或向 GONE 状态转换）。然后它在本地修改数据，并将共享链表的其他节点作废。（下面两段将细化这种情况。）第二种情况是，写入者根本不在共享链表中，写入者必须首先分配需要的空间并得到该块的副本，然后使用链表构造操作把自己加到链表的头部，然后执行上述步骤来完成写。第三种情况是，写入者在共享链表中，但不是在头部。在这种 628

情况下，它必须将自己从链表中移走（转出），然后把自己加到头部（链表构造），最后执行上述的步骤。我们会结合替换更进一步讨论转出操作，而已经看到过链表构造。现在把注意力集中在写入节点已经在头部的情况

如果块在写入者的高速缓存中处于 HEAD_DIRTY 状态，可以直接对它修改（因为目录状态必然已经处于 GONE 状态），然后发出写的节点删除共享链表的其余部分。删除操作是以串行的请求－响应方式完成的：向共享链表的下一个节点发送作废请求，此节点将自己从链表中转出，再向表头送回指向链表中下一个节点的指针。表头再向那个节点发送类似的请求，直到所有的项都被删除为止（即直到对表头的响应包含一个空指针为止，见图 8-28）。写入者或头节点，在删除进行过程中处于未决状态。在这段时间内，试图加入共享链表的新的请求会等待在未决表中。删除共享链表的时延包括几次串行的往返（作废请求、确认和转出事务）加上对每个共享链表项的相关动作，所以重要的是，不要在写入时经常碰到很长的共享链表。通过让每个节点把作废请求传递给下一个节点，向写入者发回确认以前的节点而不是返回指向下一个节点的指针，有可能降低关键路径中网络事务的数量。这不是 SCI 标准的一部分，因为它分布了作废推进的状态，从而使差错恢复的协议层复杂化。但是，实际的系统可能利用这种捷径，尤其在共享链表很长的时候。

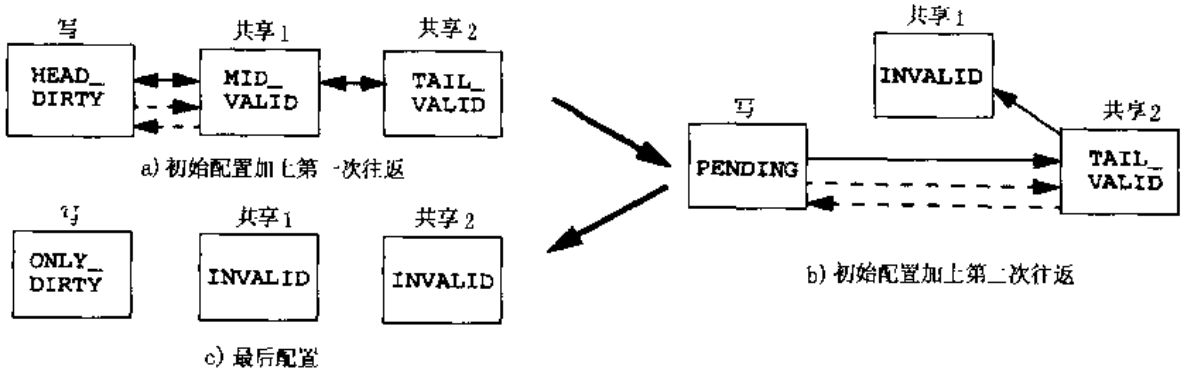


图 8-28　SCI 协议中从一个 HEAD_DIRTY 节点删除共享链表。连接链表节点的实线箭头是链表指针，虚线箭头表示了实现向下一个组态转移的网络事务

如果写入者是共享链表的头，但它的块处于 HEAD_FRESH 状态，那么在它修改该块并删除其他节点项之前，一定要变为 HEAD_DIRTY 状态。写入者进入未决状态，向宿主发出请求，宿主状态从 FRESH 变为 GONG 并应答该消息，然后，写入者进入一个不同的未决状态并删除其余部分。当请求到达宿主时，宿主可能已经不再处于 FRESH 状态了，但它指向在此期间最新进入队列的节点，该新节点已经指向写入者。当宿主查找自己的状态时，发现这种状况，就向写入者发出一个类似 NACK 的响应。当写入者收到这个响应时，它根据自己的本地状态把自己从共享链表中删除（如何做到，将在下一节讨论），通过发给宿主节点适当的新请求，试图使自己重新成为处于 HEAD_DIRTY 或 ONLY_DIRTY 状态的表头。这不是重试，因为写入者重复的不是相同的请求，而是适当改变了的请求，反映了自身和宿主的新状态（类似于第 6 章所讨论的由非原子状态转移引起的竞争情况下，对排他读升级的修改）。头节点写入的最后一个情况是如果写入者的块处于 ONLY_DIRTY 状态，这时，它可以直接修改该块，不产生任何网络事务。

5．回写和替换请求的处理

一个块的共享链表中的节点可能需要删除自己，或者是因为要执行写操作必须成为表头，或者是由于容量或冲突原因，它必须在它的远程高速缓存中替换掉该块，或者是该块正被作废。在替换情况下，即使块是处于共享状态，不需要把数据写回，它在高速缓存中的空间（及指针）也会被另一个块及其指针使用，所以为了保持正确的表示，必须把被替换的块从它的共享链表中删除。这些替换和链表的删除使用了转出操作。

考虑试图从共享链表中间转出节点的通用情况。节点先把自己置为未决状态，然后向它的上游和下游相邻节点分别发送一个请求，要求它们分别更新其前向和后向指针，以跳过要转出的节点。未决状态是需要的，因为没有什么方法可以保证共享链表中两个相邻节点不会试图同时转出自己，如果这样，就会引起指针更新的竞争。即使是置成未决状态，如果两个相邻的节点的确试图同时离开链表，它们会同时将自己设置为未决状态，并相互发送消息。这会引起死锁，因为当处于未决状态时，谁都无法响应。使用了一个简单的优先级系统来避免这样的死锁：作为约定，靠近链表尾部的节点优先转出。当两个相邻节点都已应答，就把要转出的高速缓存项的状态置为无效，结束转出操作。被转出节点的邻居不一定要改变它们的状态，除非被转出的节点是只有两个节点的链表中的第二个；在这种情况下，链表的头节点会将它的状态适当地从 HEAD_DIRTY 或 HEAD_FRESH 改变为 ONLY_DIRTY 或 ONLY_FRESH。

629
630

如果需要转出的项是链表的头，那么该项可能处于脏状态（回写）或新鲜状态（替换）。这两种情况使用同一组事务。头把自己置于未决状态，向它的下游邻居发出一个事务。使后者把自己的后向指针指向宿主存储器，并适当地改变自己的状态（例如，从 TAIL_VALID 或 MID_VALID 到 HEAD_DIRTY，或从 MID_FRESH 到 HEAD_FRESH）。当被替换的（头）节点接到响应时，它向宿主发送一个事务，宿主更新它的指针，指向新的头部，但不改变它的状态。宿主向替换者发送一个响应，现在替换者已经不在共享链表中了，并将其状态置为 INVALID。当然，如果替换者是链表中的惟一节点，它只需要与存储器通信，存储器会把它的状态置为 HOME。

头节点转出的情形提供了在请求到达时请求接收者的状态同请求不兼容的另一个例子。在来自替换者的消息到达宿主之前，宿主可能已经把它的头指针指向了一个不同的节点 X，从 X 节点宿主已经接收到对该块的请求。通常，当一个事务到达时，接收者查看它的本地状态和到来的请求的类型；如果发现不匹配，协议通常采用的策略像在前面看到的对处于 HEAD-FRESH 状态的块写入的例子那样：接收者不执行请求者请求的操作，而是发出一个很类似于 NACK 的响应。请求者将再次检查自己的状态，采取适当的动作。在这种特殊情况下，宿主发现到来的事务类型要求请求者是当前的头节点，但实际不是这样，所以它发出否定回答。替换者不停地向宿主重发请求，又不停地被否定回答。在某个时候，被重定位到替换者的出自 X 节点的请求会到达替换者，要求加入链表。替换者将查看它的（未决）状态并对请求者发出一个响应，告诉它应该去请求下游的邻居（即真正的头节点，因为替换者正在转出链表）。现在，替换者从链表中退出，处于一种不同的未决状态；它正等待进入 INVALID 状态，当下一个来自宿主的 NACK 到来时，它会变为无效状态。因此，SCI 协议中的确包括 NACK，但不是传统意义上当节点或资源忙时要求请求重发的那种 NACK。这里 NACK 只是用来指出不适当的请求，有助于请求者状态的改变。区别在于，在这种情况下，被否定回答的请求永不会以它原来的形式成功，而是引起一种新类型的请求的产生，后者会成功。

631

最后，当因扑空需要把一个块回写时，一个重要的性能问题是应该先满足扑空，还是应

该先把该块回写。在讨论基于总线的协议时，我们看到大多数情况下首先服务于扑空，要回写的块放在一个回写缓冲器中。在 NUMA-Q 机中，简化的策略是先对回写（转出）服务，然后满足扑空。虽然这减慢了扑空的满足，但缓冲的方案要比基于总线的系统复杂的多（在基于总线的系统中，可以简单地侦听回写缓冲区）。另外，我们这里所考虑的替换和回写是对于远程高速缓存的，它足够大（有几十兆字节），所以替换不经常发生。

8.6.2 关于正确性问题

SCI 标准的一个重点是提供一个良好定义的、一致的机制来保持串行化，解决竞争，避免死锁、活锁和饥饿。这个标准在解决饥饿和公平性方面比其他一致性协议要更强。在前面已经提过，使用共享者的分布链表和未决请求能满足大多数的正确性考虑，让我们详细观察一下这是如何做到的。

1. 对给定单元的操作的串行化

在 SCI 协议中，宿主节点是决定对某个块的高速缓存扑空串行化的次序的实体。但是，与 Origin 协议不同，这里的次序是请求第一次到达宿主的次序，用于保证这个次序的机制非常不同。在宿主中没有忙状态。通常（除非前面提到的某些竞争条件下），宿主接受每一个到来的请求，或者自己满足它，或者把它交给共享链表的当前头节点（如果存在未决链表的话则是未决的头）。在把请求交给另一个节点之前，它先更新它的头节点，使其指向当前的请求者。从任何其他节点对该块的请求将看到更新过的状态和指针（即指向当前请求者），虽然对应于当前请求的操作从全局上看还没有结束。这保证宿主不会同时把对一个块的两个冲突的请求交给同一个节点，避免竞争条件。正如我们已经看到的，如果请求在它被提交的头节点不能满足（即如果那个节点处于未决状态），只要需要，请求者会把自己加入到那个块的未决链表中，等待轮到它处理（见图 8-27）。注意，未决链表对块的访问是以先进先出次序进行的，保证它们完成的次序确实和它们首次到达宿主的次序一样。

632

尽管在碰到某些竞争条件时宿主可以拒绝请求，这些请求将永远不会以它们当前的形式成功，所以在串行化时不考虑它们。它们可能会被修改成新的不同的请求，那些请求将会成功，在那种情况下，将按新请求首次到达宿主的次序对它们串行化。

2. 存储器同一性模型

SCI 标准定义了一致性协议和传输层，包括网络接口的设计。但是，它没有说明许多其他的方面，如物理实现的细节和存储器同一性模型。这些都留给了系统的实现者。NUMA-Q 不满足顺序同一性，但是使用叫做处理器同一性（processor consistency）的更为松弛的存储器同一性模型，该模型我们将在 9.1 节讨论。有趣的是，和 Origin 一样，系统选择的同一性模型正是底层微处理器所支持的。

3. 死锁、活锁和挨饿

事实上，用一个分布的未决链表在请求者处保存正在等待的请求，而不是在宿主节点设置硬件的队列，由所有被分配到宿主的块所共享，这意味着没有输入缓冲满的危险，因此在协议一级没有死锁问题。还采用了一个严格的请求-应答协议。因为没有从宿主节点否定回答请求来缓解阻塞和竞争（只在请求必须被修改的特定的竞争条件下才否定回答），请求简单地加入未决链表并总是能够向前推进，所以活锁也不会发生。链表机制也保证以请求初次到达宿主的次序对它们进行处理，所以避免了挨饿。

一个节点能够加入的未决链表的总数等于它能允许的未决请求的数量，用于未决链表的存储空间已经存在于高速缓存的项中，所以在协议级不需要额外的缓冲（不允许替换未决项，引起替换的存储器操作暂停，直到对应的项不再处于未决状态。）。尽管SCI标准没有在较低的传输层涉及排队和缓冲，大多数的实现，包括NAMA-Q，在每个进入和出去的路径上，使用独立的请求和响应队列。

4. 错误处理

SCI标准在典型协议中提供了一些在硬件链路层错误恢复的可选功能。NUMA-Q并没有实现这些功能，而是假定硬件链路是可靠的。它在存储器和网络链路上提供了标准的ECC和CRC检测，发现并从硬件错误中恢复。在协议级对错误的健壮性通常要带来性能的损失。例如，SCI的决策是让写入者依次发出作废；用应答串行化，这简化了错误恢复，因为写入者知道在错误发生时已经完成了多少个作废；但是，它牺牲了性能。尽管NUMA-Q保留了这个特性，其他的系统可能选择不这样做。 633

8.6.3 协议扩展

尽管SCI协议是公平的，并且对错误相当健壮，但许多类型的操作会产生几种串行化的网络事务，因此代价相当高。读扑空要求宿主有两次网络事务，如果存在头节点，它至少有两次网络事务；如果头节点处于未决状态的话，它可能有更多次的网络事务。替换需要一次转出，这需要与它的两个邻居通信。但是，实质上，从可扩展性来看最麻烦的操作是写入时发生的作废，因为作废的代价以一个相当大的常数系数（大于一次往返时间）随共享链表中的节点数线性增长。使用分布的未决链表也会增加时延，一般来说，扑空的时延比基于存储器的协议大。已经对SCI协议提出一些扩展，以便通过硬件和协议的结合来应付广泛共享的数据。例如，SCI标准可以不采用单个大环互连，而是设想通过网桥或交换机把大量较小的环层次式地连接在一起，构造大的系统。在这种层次结构中，可以利用组合的事务。有些扩展需要改变基本协议和硬件结构，而另外一些则与基本SCI协议兼容，只需要网桥的新的实现。扩展的复杂度可能会降低低水平共享的性能。这些标准的扩展还没有最后完成，也超出了我们的讨论范围。读者可以在文献（IEEE Cmomputer Society 1995; Kaxiras and Goodman 1996; Kaxiras 1996）中找到更多的信息。标准已经包括的一个扩展专门处理只有两个节点共享一个高速缓存块的情况，这两个节点反复地对该块写入，从而使拥有权在它们之间不断转移。SCI协议文档（IEEE Computer Society 1993）中描述了这种情形。NUMA-Q包括了另一个扩展，这是一个特殊的协议操作，使处理器获得一个块的副本，即使它正在作废该块的源（非宿主）。

与Origin不同，NUMA-Q并没有对动态页面迁移提供硬件或操作系统的支持。因为有一个非常大的远程高速缓存，处理器高速缓存内对于远程分配的数据的容量型扑空几乎总是可以在本地节点的远程高速缓存中得到满足。但是，在处理器对数据写入而必须得到它的拥有权时，合适的页面分布仍然是有用的。如果没有其他节点拥有一个副本（例如，在方程求解器内核或在Ocean中一个处理器分区的内部），如果宿主是本地的，获得拥有权而且不产生网络通信；但是，如果宿主是远程的，需要与宿主的一个往返来查找目录状态。NUMA-Q的策略是，主存储器中的数据迁移是用户级软件的责任。一个例外是进程的迁移，在这种情况下，操作系统用启发式的算法试探性地将那个进程的工作页也同时迁移，使它们在新的位置 634

成为本地的。设计者认为这是页面迁移的重要场景。类似地，对于超出像 test&set 这样简单的原子交换原语的同步也没有提供什么硬件支持。

8.6.4 NUMA-Q 硬件一览

当前已经使用的 NUMA-Q 系统中，一个 4 处理器节点中每个处理器的第二级高速缓存容量为 512 KB 或 1 MB、4 路组相联，块尺寸为 32 字节。节点总线是 532 MBps 的拆分事务按序总线，具有两级一致性方案所需要的有限的乱序响应的能力。（即使 SMP 节点内的总线提供按序的响应，当一个请求必须发往一个远程节点时，要求它的响应相对于本地节点产生的响应仍然保持次序是不合理的。）一个 4 处理器节点包括多达 4 GB 的全局可访问的主存；两个 32 位宽的 133 MBps 的外围部件接口（PCI）总线通过 PCI 网桥连接到 4 处理器节点总线，IQ 设备、存储器和诊断控制器可以连接到 PCI 总线上；I/O 链路板插在存储器总线上，节点还包括通信辅助部件和网络接口。

除了在总线侧和网络侧都保持用于本地分配的数据的目录信息和用于远程分配但在本地高速缓存的数据的标记之外，IQ 链路板还包含如图 8-29 所示的 4 个主要的功能部件：总线接口控制器、DataPump、SCI 链路接口控制器和 RAM 阵列。Orion 总线接口控制器（Orion bus interface controller，OBIC）提供了和节点共享总线的接口，管理远程高速缓存数据阵列和总线侦听与请求逻辑。它的作用既是一个对非本地数据侦听和访问翻译的伪存储器控制器，又是一个把从网络到达的事务放到总线上去的伪处理器。DataPump 是一块由 Vitesse Semiconductor Corporation 制造的砷化镓芯片，它提供 SCI 标准的链路和数据包一级的传输协议。它提供对环形互连的接口，提取发往本节点的数据包，而让其他的数据包继续传送。SCI 链路接

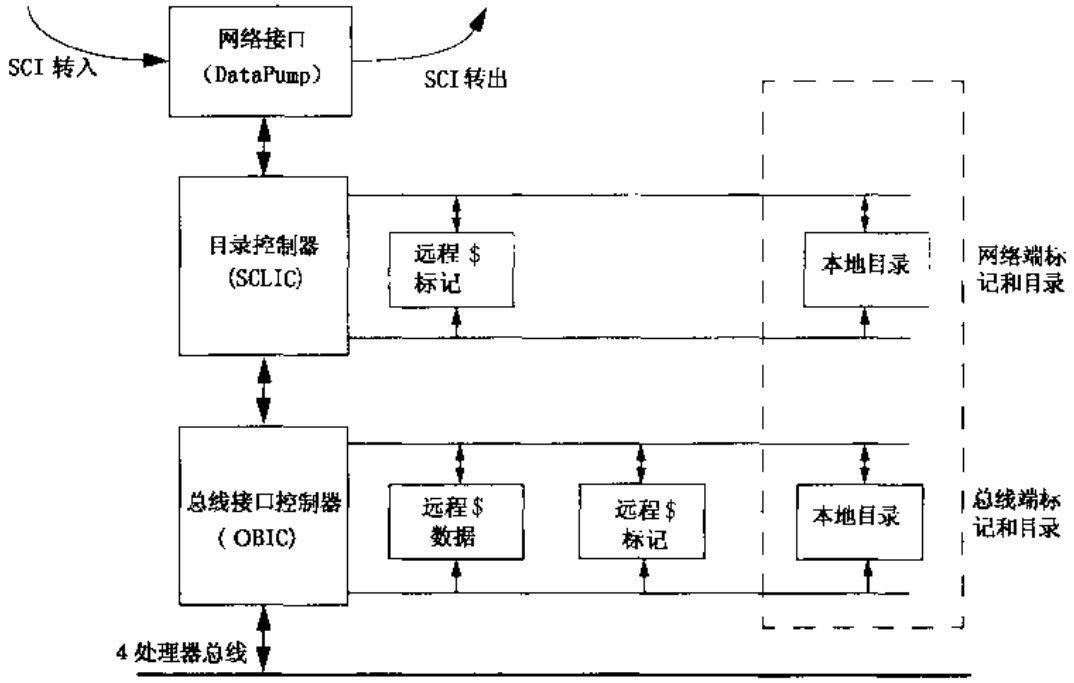


图 8-29 NUMA-Q 的 IQ 链路板的功能框图。同步 DRAM（SDRAM）实现远程高速缓存的数据存储。静态 RAM（SRAM）实现了总线侧的标记和目录，因为对网络侧的标记和目录的访问可以较慢，所以它们用 SDRAM 实现

口控制器（SCLIC）与 DataPump、OBIC、中断控制器以及目录标记接口。它的主要功能是管理 SCI 一致性协议。RAM 阵列实现远程高速缓存的数据和不同标记的存储。在 8.6.6 节讨论 IQ 链路板的实现时，将对这些部件做进一步的描述。

对于 4 处理器节点间的互连，SCI 标准定义了传输层和高速缓存一致性协议。传输层定义了节点对网络接口的功能规范和由点对点链路构成的环这样的网络拓扑。特别地，它定义了一个 1 GBps 的环状互连以及能在它上面产生的事务。NUMA-Q 系统起初采用连接多达 8 个 4 处理器节点的单环拓扑结构，如图 8-24 所示。来自节点的电缆连接到包含在一个盒子里的环路端口，这个盒子叫做 IQ-Plus。较大的系统包含由局域网连接的多个 8 个 4 处理器节点的系统。前面已经提到，由于长环的高时延，SCI 标准设想较大的系统一般将由交换机互连的多个环构成。每个节点只有少量的未决请求，长环的时延会严重限制节点到网络的带宽（见第 11 章）。SCI 的传输层将在第 10 章进一步讨论。

因为机器的目标是用于数据库和事务处理工作负载，I/O 是 NUMA-Q 设计的重点。和 Origin 一样，I/O 可全局寻址，所以任何处理器可以直接读写任何 I/O 设备，而不只限于那些挂接在局部 4 处理器节点上的 I/O 设备。非本地的处理器并不非要向挂接设备的 4 处理器节点发出显式的消息进而让该 4 节点的处理器发出访问。这对于商业应用是十分便利的，因为这些应用经常不是结构化的，因此处理器只需访问它本地的磁盘就可以。I/O 设备连接到两条 PCI 总线上，PCI 总线通过 PCI 网桥连接到 4 处理器节点总线。每条 PCI 总线的时钟频率和其宽度是存储器总线速度的一半，所以带宽大约是存储器总线的 1/4。物理上，处理器有两种方法访问其他节点上的 I/O 设备，其一是经由 SCI 环，可以通过高速缓存一致性协议，也可以通过不经高速缓存的写，就像 Origin 通过 Hub 和网络所做的那样。但是，环网上的带宽是一个珍贵的资源。I/O 传输会占用相当大的带宽，打扰存储器的访问。因此，NUMA-Q 为节点间的 I/O 传输提供了一条经由 PCI 总线的独立通信通路，它是缺省的 I/O 路径。一条 FiberChannel 链路连接到每个节点上的 PCI 总线。这些链路通过点对点连接、仲裁 FiberChannel 环或者 FiberChannel 交换机连接到系统所有的共享磁盘；采用何种方式，取决于处理和 I/O系统的规模（见图 8-30）。

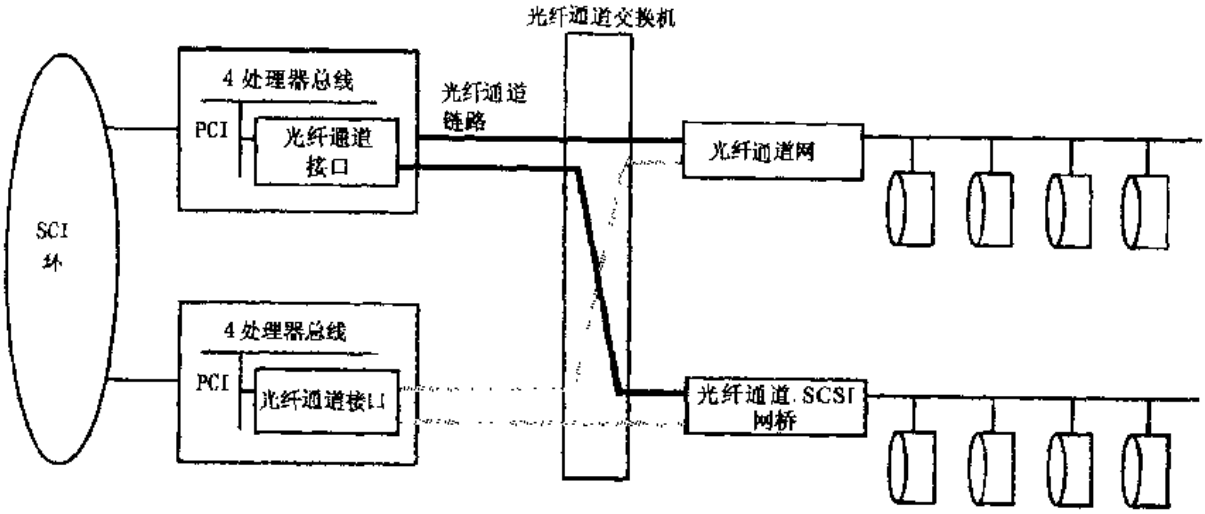


图 8-30 Sequent 的 NUMA-Q 机的 I/O 子系统。I/O 可全局寻址，节点间的 I/O 数据传输可以通过经由 PCI 总线的 FiberChannel 或通过用于存储器操作的 SCI 环

FiberChannel 通过一个将 FiberChannel 数据格式转换成磁盘接受的 SCSI 格式的网桥，以 50 MBps 以上的持续速率访问磁盘。对系统中任何磁盘的 I/O 通常通过经由本地 PCI 总线和 FiberChannel 交换机的路径；但是，如果这条路径由于某种原因失效，操作系统使 I/O 传输经过 SCI 环到达其他节点，再经由那个节点的 PCI 总线和 FiberChannel 链路到达磁盘。FiberChannel 也可以以松散耦合的方式连接多个 NUMA-Q 系统，让多个系统共享磁盘。最后，每个节点的 PCI 总线都连接一个管理和诊断控制器；这些控制器通过一个专用的局域网如 Ethernet 相互连接，并和系统的控制台连接，便于系统维护和诊断。

8.6.5 协议和 SMP 节点的交互

关于 SCI 协议的上述讨论忽略了 4 处理器节点的多处理器性质以及节点内的基于总线的协议。现在，我们已经理解了节点和 IQ 链路的硬件结构，下面考察一下这两个协议间的交互，相互作用的协议对 4 处理器节点和 IQ 链路的要求，以及用市售的 SMP 作为节点产生的一些特殊问题。

635 - 637

读请求说明了某些交互。在处理器第二级高速缓存中的读扑空首先出现在节点总线上。除了被其他处理器高速缓存侦听外，它还被 IQ 链路板上的 OBIC 总线控制器侦听。OBIC 查找远程高速缓存以及本地分配块的目录状态位，判断读是能在本地 4 处理器节点上满足，还是必须要传出该节点。如果是前一种情况，主存或其他高速缓存之一能满足这个读，发生适当的 MESI 状态变化。（在固定数量（4 个）的总线周期之后，按序报告侦听结果；如果控制器不能在此时间内完成侦听，它发出一个暂停信号，申请另外两个总线周期，在此之后，存储器再次检查侦听结果。这个过程继续直到得到所有的侦听结果为止。）节点的总线实现了对请求的按序响应。但是，如果 OBIC 检测到请求必须传出节点，它就必须干预。OBIC 发出一个推迟响应信号，告诉总线可以违反按序响应的性质，继续其他的事务，OBIC 将负责响应这个请求。其实如果总线实现乱序响应，就不必要这样做了。OBIC 把请求传给 SCLIC 以完成目录协议。当响应返回时，从 SCLIC 传回 OBIC，OBIC 把它放到总线上，结束推迟了的事务。注意，当把任何基于总线的系统当作更大的高速缓存一致的机器节点使用时，总线必须是支持拆分事务的，这不仅是为了性能，也是为了简化正确性。否则，在整个远程事务持续期间总线会被占用，甚至不允许完成本地的扑空，也不允许处理器高速缓存处理到达的网络事务。

写操作进出 4 处理器节点的路径与读操作的类似。远程高速缓存中块的状态被 OBIC 监听，指明该块是由本地节点拥有，还是必须由 SCLIC 发送到其宿主节点。如果需要，SCLIC 负责把节点放到共享链表的表头并作废其他节点。当 SCLIC 完成时，它把应答放到 4 处理器总线上（经由 OBIC），结束操作。由于 4 处理器节点本身的限制出现了一种有趣的情况。考虑对一个本地分配的块的读扑空和写扑空，该块在远地高速缓存，状态是被修改。当响应返回并作为推迟的响应出现在总线上时，它应该更新主存。但是，4 处理器节点存储器的实现无法处理推迟的请求和响应，在看到推迟的响应时也不会更新自己。因此，当推迟的响应经由 OBIC 传到总线时，OBIC 也必须保证在它释放总线之前，通过特殊的动作更新存储器。另一个限制来自 OBIC 使用 4 处理器节点总线协议的方式。如果 4 处理器节点内的两个处理器背靠背发出排他读，第一个处理器传到 SCLIC，我们希望第二个处理器会被缓冲，以适当的状态接受来自第一个处理器的响应。但是，具体的实现是拒绝第二个处理器请求，在第一个

638

处理器返回前必须重试。

最后，考虑串行化。因为 SCI 协议级的串行化是在宿主节点进行的，到达宿主的事务不仅相互之间要串行化，而且要相对于宿主 4 处理器节点中处理器的访问串行化。例如，假定一个块在宿主处于 HOME 状态。在 SCI 协议级，这意味着系统中其他远程高速缓存（它必须在其他节点上）没有该块的有效副本。但是，和 Origin 协议中的未拥有状态不一样，这并不意味着宿主节点的处理器高速缓存中没有该块的副本。事实上，即使宿主节点某个处理器的高速缓存中有该块的脏副本，目录也会是 HOME 状态，甚至为了得到正确的值，对于在宿主节点本地分配的块的请求也必须在 4 处理器节点总线上广播，而不能完全由 OBIC 和 SCLIC 处理。类似地，使目录从 HOME 或 FRESH 变为 GONE 状态的请求必须被放到节点总线上，这样处理器高速缓存中的副本才能被作废。因为到来的请求和宿主内数据的本地扑空都出现在节点总线上，所以让总线成为宿主节点上实现串行化的部件就是很自然的。

类似地，串行化的问题在访问远程分配的块的请求 4 处理器节点中也需要解决。4 处理器节点中和远程分配的块相关的动作在本地 SCLIC 而不是在本地总线上串行化。因此，本地处理器对远程高速缓存块的请求和来自 SCI 互连网络对同一块的请求都是在本地 SCLIC 中串行化的。类似地，SCLIC 也负责请求者的未决作废和进入节点的请求之间的本地串行化。和节点协议间的其他交互在考虑 IQ 链路板部件的实现时讨论。

8.6.6 IQ 链路的实现

与 Origin 中单片的 Hub 不同，SCLIC 目录控制器、OBIC 总线接口控制器和 DataPump 是 IQ 链路板上独立的芯片，IQ 链路板上还包含某些用于标记、状态和远程高速缓存数据存储的 SRAM 和 SDRAM（见图 8-29）。

OBIC 可以直接访问远程高速缓存中的数据。使用两组标记来减少 SCLIC 和 OBIC 之间的通信：SCLIC 访问的网络侧标记和 OBIC 访问的总线侧标记。目录状态和本地分配的块也是这样。总线侧的标记和目录状态仅包含总线侦听所需的信息，用 SRAM 实现，所以能以总线的速度查找。网络侧的标记和状态需要更多的信息，可以慢一些，所以用同步 DRAM（SDRAM）实现。总线侧的本地目录 SRAM 为每个 64 字节的块只设两位的目录状态（以区分 HOME、FRESH 和 GONE 状态），而网络侧的目录还包含 6 位的 SCI 头指针。总线侧远程高速缓存标记只有 4 位的状态，并且不包括 SCI 前向和后向链表指针。它记录 14 种状态，某些是过渡状态（例如，记录正在转出的块，以及记录在总线上有未完成的重试的特殊总线代理，它必须获得该块的优先权）。网络侧的远程高速缓存标记是目录协议的一部分，每个块除包含 7 位的协议状态外，还有两个 6 位的链表指针（以及 13 位的高速缓存标记本身）。 639

与 Origin 中的硬接线协议表不同，NUMA-Q 中的 SCLIC 一致性协议控制器是可编程的。这意味着协议可以由软件或固件实现，而不是由硬接线的有限状态机实现。每个来自本地处理器的协议调用操作，以及每个来自网络的事务调用在协议引擎上运行的软件“处理例程”或任务。这些用微码编写的软件处理程序管理协议状态，在节点总线上产生干涉，生成网络事务。SCLIC 引擎有多组寄存器，并行支持 12 个读/写/作废事务和一个中断事务。为了支持 4 处理器节点间的中断，SCLIC 提供了为标准的 4 处理器节点间中断提供路由的网桥和额外几个位，以便在生成中断时包含目标 4 处理器节点号。

一个可编程的协议引擎有几个基本的优点。它允许软件调试协议，通过简单地下载新的

协议代码而修正错误。它提供了在机器建造好之后或发现新的瓶颈时，试验和修改协议的灵活性，并允许机器支持多个协议。它允许为了性能调试，在处理程序中插入代码，监视选定的事件，这对于理解隐式的通信和共享地址空间中人为通信的影响，特别有价值。其缺点在于，每个事务对可编程协议引擎的占用要比对硬接线引擎的占用时间长，因此要付出性能代价。NUMA-Q 的 SCLIC 试图降低这种性能影响。协议处理器有三级流水线，每个周期发送最多两条指令（一条转移，一条其他指令）。用高速缓存暂存最近用过的目录状态和标记信息，而不是每次都访问目录 RAM。最后，它支持目录协议需要的位处理操作以及加快处理例程分支和管理的有用的指令，比如，“当缓冲满时入队”和“当队列空间可用时分支”等指令。Stanford 的 FLASH 多处理器（Kuskin et al. 1994）使用了一个略有不同的可编程协议引擎，它是硬连线 Stanford 的 DASH 机的后继。

每个 Pentium Pro 处理器可以有多达 4 个未完成的请求。节点总线同时可以有 8 个未决请求，并且保证侦听和数据响应的次序（除非使用了前面提到的推迟响应）。OBIC 可以有 4 个外部的对 SCLIC 的未决请求，并能同时缓冲两个进入 4 处理器节点总线的事务。如果来自 4 处理器节点总线的第五个请求需要传出节点，OBIC 会否定回答它，直到缓冲项腾出，但它不会使 4 处理器节点总线暂停处理本地操作。SCLIC 最多可以有 8 个未决请求，能够同时缓冲 4 个进入的请求。SCLIC 的一个简化的说明如图 8-31 所示。最后，DataPump 的请求和响应缓冲的深度是：发往外部网络为两项，从网络向内为 4 项。不管是进入还是送出，所有的请求和响应缓冲在物理实现上是分离的。

640

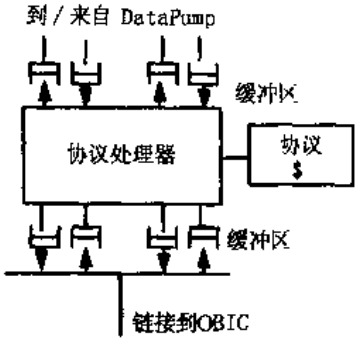


图 8-31 SCLIC 芯片的简化框图。它包括一个可编程协议处理器、一个目录信息高速缓存和对 OBIC（总线）和 DataPump（网络）的接口缓冲

除了以软件提供修改协议处理例程的能力，IQ 链路板上的所有三个部件都提供了性能计数器，允许各种事件和统计的非打扰测量。在 SCLIC 中有三个 40 比特的存储器映射的计数器，在 OBIC 中有四个。每个计数器都能用软件设置，统计任何大量的事件；如协议引擎利用率、存储器和总线利用度、队列占用率、出现的 SCI 命令类型和出现的总线事务类型。在任何时候，计数器都可以用主处理器上的软件读出或被编程，使得在超过预定阈值时产生中断。Pentium Pro 处理器模块本身提供了许多性能计数器，统计一级和二级高速缓存扑空、请求类型的频率、内部资源的占用率等特性。和可编程的处理例程一起，这些计数器在运行工作负荷时提供了机器行为的有用信息。

8.6.7 性能特征

节点总线的峰值带宽是 532 MBps，SCI 环形互连的节点到网络接口每个方向传输速率是 500 MBps。IQ 链路板在这两者之间每个方向的数据传输率是 30 MBps（注意，只有少数出现

在节点总线或 SCI 环上的事务与其他的互连有关)。在主存(或远程高速缓存)满足的本地读扑空的时延在理想条件下平均约为 250 ns。在两个节点的系统中,在远程存储器满足的读时延大约是 2.5 μs,比例大约是 10 比 1。但是,由于有了远程访问高速缓存,附加通信的几率非常低。事务的前 18 个比特通过 DataPump 网络接口的时延是 16 ns,然后,每传输 18 个比特花 2 ns。就网络本身而言,第一个比特从 4 处理器节点的 DataPump 输出进入实现环的 IQ-Plus 盒,然后沿环回到下一个节点的 DataPump,需要 26 ns。 641

NUMA-Q 的设计者用微基准测试程序和数据库及事务处理工作负载对机器进行了一些试验。为了了解机器对微基准测试程序的性能能力、时延随负载的变化以及工作负载的特征,让我们看一下结果。对于 4 个处理器同时以最快的速度产生高速缓存扑空的单 4 处理器节点系统,每个背靠背的读扑空花费 600 ns,对处理器的综合带宽是 290 MBps。在相似的条件下,产生读后跟一个回写的背靠背的写扑空要花费 585 ns,可持续带宽是 195 MBps。对于在每条 PCI I/O 总线上有多个 I/O 控制器的单节点系统,尽可能快地产生 I/O 设备对本地存储器的写入,每个高速缓存块传输需要 360 ns,可持续带宽为 111 MBps。

表 8-3 给出了在多个 4 处理器节点系统中运行各种工作负载得到的时延和特征。前两行对应从微基准测试程序得到的数据,这些微基准测试程序使所有 4 处理器节点同时发出读扑空,并在远程存储器中满足。第三行对应 Transaction Processing Council 的在线事务处理基准测试程序 TPC-B(见附录)。最后一行对应 TPC-D 基准测试程序集的 Query 9,代表决策支持应用。时延通过内嵌在 OBIC 和 SCLIC 中的性能计数器测量,其测量的不是从处理器,而是从总线请求到第一个数据响应。所有的工作负载都运行在 4 个节点(16 个处理器)的系统上,决策支持工作负荷是个例外,它在 8 个节点上运行。对本地分配数据的写扑空包含在最后一列,它引起远程发送的作废,数量很少。

表 8-3 在 8 节点 NUMA-Q 机上运行微基准测试程序和工作负载的特征

重 载	L_2 扑空的时延		SCLIC 利用率	L_2扑空满足以下情况的百分比			
	所有	远程可满足的		本地存储器	其他本地高速缓存	本地"远程高速缓存"	远程节点
远程读扑空	8 020 ns	8 300 ns	95%	1.5%	0%	2%	96.5%
远程写扑空	9 350 ns	9 625 ns	95%	1%	0%	2%	97%
TPC-B-like	630 ns	4 300 ns	54%	80%	2%	11.5%	6.5%
TPC-D (Q9)	580 ns	3 950 ns	40%	85%	5.5%	4%	5.5%

远程数据访问的时延远比无负载时延高得多。通常,SCI 环和协议具有比更为分布的网络和基于存储器的协议更高的时延。但是,至少在事务处理和决策支持工作负荷中,远程访问的许多时间都花在IQ链路板本身,而不是花在总线或环上。图8-32把4到8个节点的系统上两个工作负载的平均远程时延划分成三种成分。在有负载和无负载情况下改善远程访问性能的途径是使 IQ 链路板效率更高。设计者考虑了许多可能性,包括重新设计 SCLIC(或许在可编程 SCLIC 中使用两个而不是一个指令部件),优化 OBIC,希望下一代机器把重载下的远程访问时延降低到 2 μs 左右。远程高速缓存对于使容量扑空本地化很有用。TPC-D (Q9) 工作负荷对 SCLIC 的利用率比 TPC-B 工作负荷小,因为它产生较少的作废。

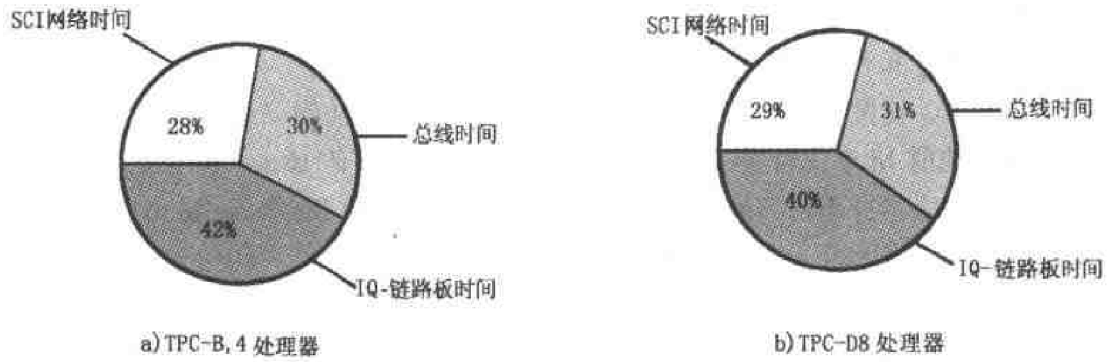


图 8-32 在一个 8 个 4 处理器节点的 NUMA-Q 上两种工作负载下的平均远程扑空时延的成分。在两种情况下，大部分的时间花费在 IQ 链路板上，包括在 SCLIC 和 DataPump 或 SCLIC 和 OBIC 之间的数据传输。图中 OBIC 芯片本身的时间包含在总线时间内

8.6.8 对比案例分析：HAL S1 多处理器

HAL Computer System 的 S1 多处理器是 NUMA-Q 和 Origin2000 一些特性的有趣组合。和 NUMA-Q 一样，S1 采用 Pentium Pro 4 处理器作为处理节点；但是，它在节点间采用 Origin2000 那样的基于存储器的目录协议，而不是基于高速缓存的 SCI 协议。另外，为了降低时延和通信辅助部件的占用，它把一致性机构与节点更紧密地集成，这方面更接近 Origin。它的目录协议控制器（SCLIC）、总线接口控制器（OBIC）和网络接口（DataPump）不是用独立的芯片实现，S1 把整个通信辅助部件和网络接口集成在一块叫做网格一致性单元（MCU）的芯片上，而存储使用单独的芯片。另一方面，高速缓存一致性设计只能扩展到 4 个 4 处理器节点，没有可编程控制器的灵活性，也没有包含降低容量扑空的远程访问高速缓存。

642 ~ 643

因为基于存储器的协议不需要使用每个高速缓存表项的前向和后向指针，所以不需要节点级的远程数据高速缓存来提供这个功能（处理器高速缓存也不提供）；在基于存储器的协议里，远程高速缓存只对降低容量扑空有用，S1 不用它们。目录信息在独立的 SRAM 芯片中保存，但是不保存所有存储器块的目录信息，而是只保存远程高速缓存块的目录信息，把目录本身组织成高速缓存（在 8.10.1 节中讨论），可以大大降低所需要的目录存储容量。MCU 还包含一个支持显式消息传递和在高速缓存一致的共享空间中块传输的 DMA 设备（见第 11 章）。消息传递或显式数据传输可以通过 DMA 引擎实现（对较大的消息有利），也可以通过高速缓存块的传输机制实现（对较小的消息有利）。MCU 是硬接线的而不是可编程的，这降低了协议处理对它的占用，改善了它的性能。MCU 也对性能监测提供了相当多的硬件支持。除了 MCU，惟一的专用芯片是网络路由器，它是一个 6 端口的交叉开关，包含 190 万个晶体管，为提高速度而做了优化，网络时钟是 200 MHz。通过单个路由器的时延是 42 ns，每个方向可用的单链路带宽是 1.6 GBps，两者都类似于 Origin 2000 的网络。S1 的内部互连的实现可以扩展到 32 个节点（128 个处理器）。

在 S1 中，把所有辅助部件功能集成到一个芯片上的主要目标是降低远程访问时延，提高远程带宽。从设计者的模拟测量可知，在本地存储器满足的读扑空的最好无负载时延是 240 ns，对于在远程宿主的干净的块读扑空是 1 065 ns，对于附近的第三方节点中的脏块的读扑空是 1 365 ns。远程到本地时延的比例在 4 到 5 之间（包括竞争）。比 SGI 的 Origin 2000 稍差，但比 NUMA-Q 要好。但是，微基准测试程序时延的比较对于预测工作负载的整体性能不

是很有意义，因为它们忽略了重要的因素，例如远程高速缓存和能显著影响通信频率的灵活性。

HAL S1 在复制单个 4 KB 的页面时实现的带宽是有指导意义的。通过处理器的读写，本地存储器之间实现的带宽是 105 MBps（主要受到 4 处理器节点存储器控制器的限制，它必须处理存储器的读和写）；当采用处理器的读写实现时，本地存储器和远程存储器间每个方向的带宽约为 70 MBps，在通过 MCU 中的 DMA 引擎时，本地存储器和远程存储器每个方向的带宽约为 270 MBps。通过处理器读写的远程传输主要受限于发自处理器的未完成的存储器操作的数量，在 DMA 情况下没有这个问题。DMA 有额外的优势，它在每一个存储器块的访问发出端只需要一次总线事务，而不是像处理器读写时那样两个分裂的事务对（一个为读，一个为写）。至少在不存在传输间的竞争情况下，节点本地 4 处理器节点总线远在互连网络之前就成为带宽瓶颈。

644

现在，对于基于存储器和基于高速缓冲协议两种情况，在一定深度上理解了实现一致共享地址空间编程模型的协议层。让我们简要地考察一下，在决定应用的性能时，协议和通信体系结构基本性能参数之间的相互作用。

8.7 性能参数和协议性能

回忆一下，通信体系结构有 4 个主要的性能参数：主处理器的额外开销、通信辅助部件的占用、网络传输延迟和网络带宽。处理器的额外开销在高速缓存一致的机器上通常很小（不像在消息传递型系统中那样占主导地位），并且完全由底层节点决定。在最好的情形下，不能通过重叠对处理器隐藏的处理器额外开销是发出存储器操作的代价。在最坏的情形下，它是穿过处理器高速缓存层次结构到达通信辅助部件的代价（可能相当大）。所有其他的协议处理动作不在通信辅助部件上（如 Hub 或 IO 链路）。对大多数高性能多处理器网络上的应用而言，网络链路带宽通常是足够的（Holt et al. 1995）。所以，通信体系结构控制下更关键的问题是网络延迟和辅助部件的占用。

正如我们已经看到的，通信辅助部件在协议处理中扮演了许多角色，包括产生请求，查找目录状态，为响应而访问数据，以及发出作废和接收确认。事务处理对辅助部件的占用不仅影响非竞争时延，也引起辅助部件的竞争，增加其他事务的代价。在高速缓存一致的机器上特别是如此，因为存在大量的小事务，如携带数据的事务和请求、作废和确认等其他事务，这意味着占用频繁发生，不能很好地分摊。在不采用高速缓存一致的共享地址空间的机器中情况会好一些，在这些机器上，事务只传输访问的字而不是整个高速缓存块，因为复制和一致性是由程序员管理的（见 3.6 节），但是分摊仍然很小。实际上，辅助部件占用经常主导了节点到网络接口的数据传输带宽，成为端点的吞吐量的关键瓶颈（Holt et al. 1995）。所以保持低的辅助部件占用率是重要的。在协议级，重要的是既要保证辅助部件不被待完成的事务占有，而使它不能处理其他不相关的事务，也要降低每个事务需要辅助部件处理的工作量。例如，如果宿主把一个请求转交给脏节点，宿主的辅助部件不应该被占用直到脏节点返回响应，这样做的话会大大增加辅助部件的占用，应该做是为了转去为下一个事务服务，在以后响应到来时再处理它。在硬件设计级，使辅助部件专门化，并与节点的存储器系统紧密集成是重要的，这样，每个事务对它的有效占用度就低。集成得越紧密，专门化程度越高，设计越不面向市售部件，占用度就越低。

645

1. 网络延迟和辅助部件占用度的影响

图 8-33 显示了辅助部件占用度和网络时延和对性能的影响，假定类似 SGI 的 Origin 2000 的有效的基于存储器的目录协议。在没有竞争的情况下，辅助部件占用的行为和事务路径中的网络传输延迟和任何其他时延成分一样：增加 d 个周期的占用度与保持占用度不变但增加 d 个周期的网络延迟影响一样。因为 x 轴是远程读扑空的总的非竞争往返时延（包括网络延迟和辅助部件占用度的代价），如果增加占用度不引起竞争，那么不同占用度值的所有曲线都相同。事实上，是有竞争的，所以曲线的分离表示了增加辅助部件占用度所引起的竞争的影响。

在图中占用度（o）的最小值代表一个积极的硬连线的辅助部件，它和 Origin2000 使用的类似，与高速缓存或存储器控制器紧密地集成。最一般的方案是在存储器总线上放置一个慢速的通用处理器，扮演通信辅助部件的角色。最富进取的网络延迟代表现代高端处理器内部的互连，最一般的网络延迟接近于使用商品化的系统域网络如异步传输模式（ATM）。我们看到，对于最进取性方案的占用，时延曲线取 $1/l$ 的形状。辅助部件占用度导致的竞争对强调通信带宽的应用的性能有显著的影响（特别是那些产生突发通信的应用），特别对多处理器内使用的低延迟网络而言。所以在曲线的左端，较高占用度的曲线离其他曲线较远。对于合理的占用度，曲线在较大网络延迟处相互接近，因为事务在网络中花的时间更多，使辅助部件不忙碌，从而辅助部件的竞争减少。对于较高的占用度，曲线几乎是平的，至少对较低的网络延迟来说，表示辅助部件已经饱和。对于像排序和 FFT 这样具有突发通信的应用，问题特别严重，因为在通信阶段，通信相对于计算的比率并不怎么随问题的规模而改变，所

646

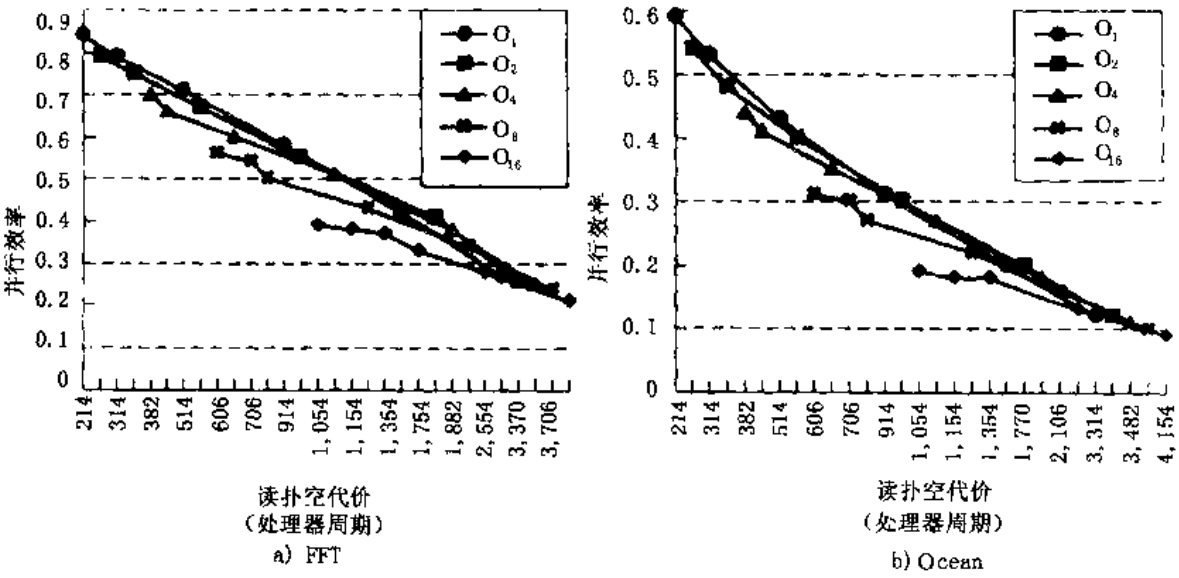


图 8-33　辅助部件占用度和网络延迟对基于存储器的高速缓存一致性协议的影响。y 轴是并行效率，它是对串行执行的加速比被所使用的处理器数去除（1 是理想值）。x 轴是在宿主节点主存满足的读扑空的非竞争往返时延，包括所有代价成分（占用度、传输时延、在缓冲内的时间、网络带宽）。每个曲线对应一个不同的辅助部件占用度（o）值，而沿曲线惟一变化的参数是网络的传输延迟（l）。最低的占用假定是 7 个处理器周期，用 O_1 表示。O_2 对应于两倍的占用度（14 个处理器周期），以此类推。所有的其他的代价，例如传过高速缓存层次和通过缓冲的时间和节点到网络的带宽，都保持不变。该图是模拟 64 个处理器执行产生的。主要的结论是辅助部件占用度引起的竞争对于性能是十分重要的，特别是对低时延的网络而言

以较大的问题规模没有缓解该阶段的竞争。辅助部件占用度问题对于这样的事务不是很严重，即通信事件被大量的计算隔开，对通信带宽的要求不高（例如，Barnes-Hut）。当使用时延包容技术（第 11 章讨论）时，对带宽的要求更高，因此辅助部件占用度的影响就更大，即使是在较高的传输时延情况下也是如此，最高占用度的曲线对于 FFT 和排序几乎完全是平的（Holt et al. 1995）。这个数据表明在以高速缓存块这样的细粒度进行通信和保持一致性的机器中，降低辅助部件的占用度是重要的。辅助部件占用度引起的竞争的影响将会随解题的处理器数量的增加而增加，因为通信对计算的比率会增加。 647

2. 辅助部件占用度对协议折中的效果

辅助部件占用度不仅对给定协议的性能有影响，而且对于协议间的折中也有影响。我们已经看到，基于高速缓存协议的写操作时的时延比基于存储器的协议要大，因为作废共享者的事务是串行的。SCI 的基于高速缓存的协议存储器操作比基于存储器的协议有更多的协议处理工作要做，所以辅助部件的实际占用度要高得多，特别是若辅助部件是可编程的而不是硬接线的。与写操作的较高的时延结合起来，这似乎意味着基于存储器的协议的性能更好些。当辅助部件占用度以及它的性能影响增加时，不同协议的性能差别会变得更大。另一方面，SCI 协议中给定的存储器操作（例如一个写入）的协议处理占用度分布在更多的节点和辅助部件上，所以，依赖于应用的通信模式，一个给定的辅助部件所经历的竞争可能更少。例如，当基于存储器的协议中不规则突发通信引起的热点成为问题时（如 Radix 排序），SCI 可能在某种程度上缓解它。实际的折中依赖真实程序和机器的特征，尽管在总体上，我们可能期望优化实现的基于存储器的协议性能更好。

3. 以硬件改善性能参数

使用更强的专门硬件改善性能特征如延迟、占用度和带宽的方法很多，下面是一些值得注意的技术。首先，SRAM 目录高速缓存可以更靠近辅助部件，以降低目录查找的代价，就像 NUMA-Q 和 Stanford 的 FLASH 多处理器所做的那样（Kuskin et al. 1994）。其次，可以在宿主为每个存储器块维护 SRAM 的一个位，记录该块在本地存储器中是否处于清洁状态。如果是，那么对本地分配块的读扑空就没必要进一步调用辅助部件。第三，如果辅助部件占用率高，可以在协议处理中分级流水，就像 NUMA-Q 和 Stanford FLASH 那样（例如，对请求解码，查找目录，产生响应），或者其占用度和其他动作重叠。辅助部件流水线作业降低了竞争，但是没有降低一次存储器操作的非竞争时延；让辅助部件在做完所有必要的工作之前，生成并送出响应或转发的请求可以得到相反的（或互补的）结果。

8.8 同步

可扩展非高速缓存一致的共享地址空间系统中的软件同步算法在 7.9 节中已经讨论过了，采用的是原子交换指令或者 LL-SC 指令。回顾一下，与基于总线的机器的同步算法相比，这些同步算法致力于充分利用互连中的独立路径的并行性，保证处理器绕局部而不是非局部变量自旋。同样的算法也适用于可扩展的高速缓存一致的机器。但是，有两点不同。首先，绕远程分配的变量自旋的性能影响不很显著，因为处理器会高速缓存该变量，然后在其作废之前一直绕其做本地自旋；让处理器踏步等待不同的变量当然比踏步等待同一个变量要好，因为它可以在写并作废变量时，防止所有的处理器都涌向同一个宿主存储器，从而减少了竞争。同步变量的良好放置能够把作废后发生的扑空从三跳扑空转变成两跳扑空。但是， 648

处理器绕本地分配的变量自旋只在一个不太可能发生的场景下对性能非常重要：如果高速缓存所有层次是统一的和直接映射的，而自旋循环的指令与变量本身冲突，在这个情况下，冲突型扑空能够在本地满足。其次，尽管同步算法的这些性能方面不再是关键的，当它和一致性协议相互作用时实现原子的原语和 LL-SC 却更令人感兴趣。本节考察性能和实现的侧面，首先比较第 5、7 章中介绍的 SGI Origin2000 上锁的不同同步算法的性能，接着讨论超出了第 6 章中讨论过的基于总线机器范围的原子原语的几个新的实现问题。

8.8.1 几种同步算法的性能

这里用来说明同步性能的试验与 5.5 节中对基于总线的 SGI Challenge 所使用的一样，我们再次使用 LL-SC 作为构造原子操作的原语。所用的延迟也是以处理器的时钟周期表示，所以和实际的微秒数不同。第 5、7 章中所描述的加锁算法的结果如图 8-34 所示，这里处理器

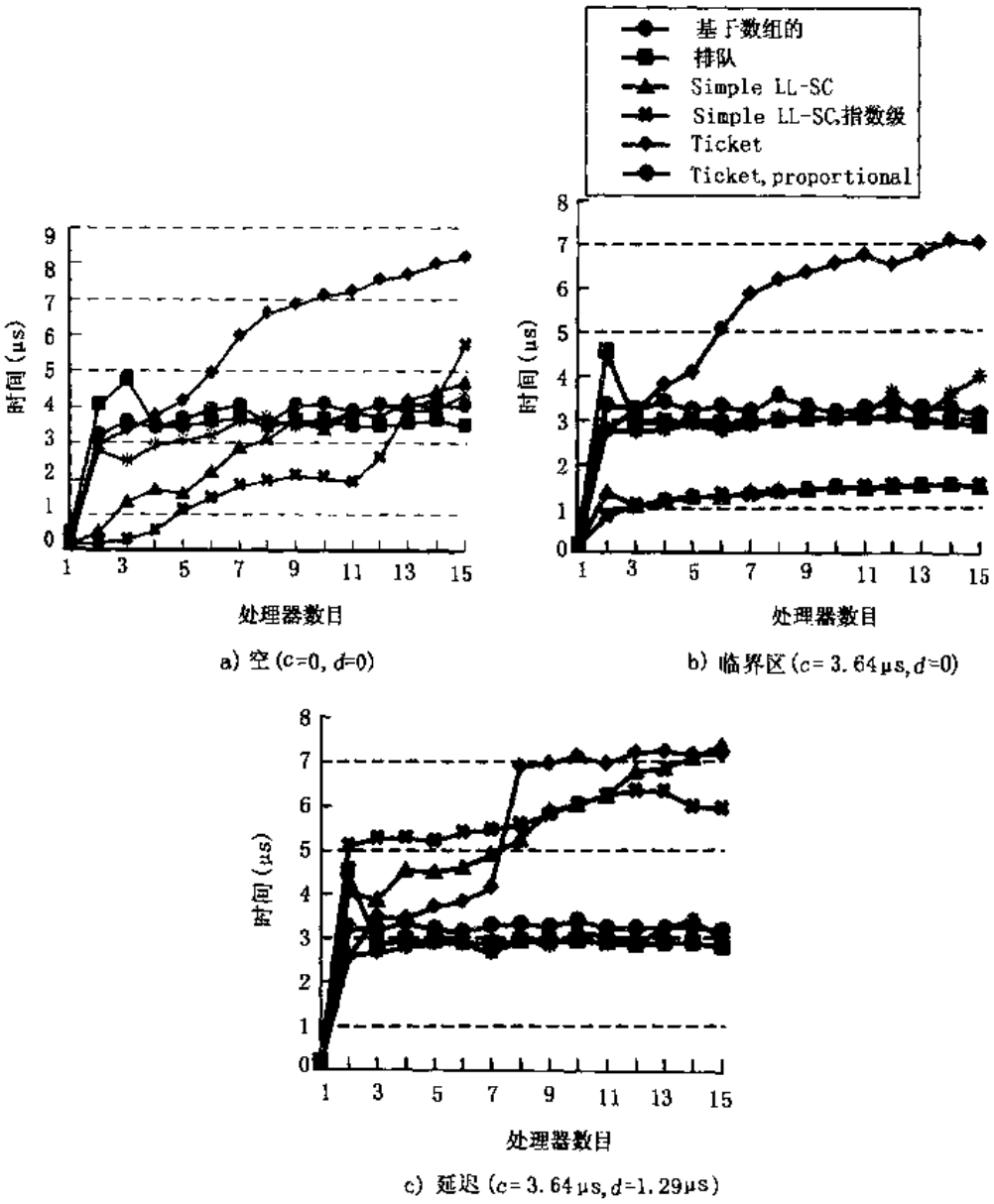


图 8-34 在 SGI 的 Origin2000 上 3 种不同情况下锁的性能

数为 16。同样，这里的时延也有三组不同的值，对应于处理器反复竞争的临界区之内和临界区之后的延迟。

同样，如果不在临界区之间使用延迟，这种简单的锁是不公平的并且产生较高的吞吐量。在空临界区场合下，指数延迟有助于简单的 LL-SC 锁，因为这是需要缓解竞争的情况。标签锁在这种情况下与总线型机器上一样，扩展性还是很差，但是当使用了比例回退后扩展性却很好。基于阵列锁的扩展性也很好。有了一致的高速缓存，通过软件排队锁使锁变量在主存中更好地分布没有太大的用处。如果我们强迫这些简单锁具有公平性，其表现就跟没有比例回退的标签锁差不多。

如果在锁访问之间使用一个非空的临界区和一个延迟就可使所有的锁具有公平性（如图 8-34c 所示）。此时，简单的 LL-SC 锁没有了它的优点，却表现出在扩展性方面的缺点。基于数组的锁、排队锁和带比例回退的票号锁都有很好的可扩展性（至少对这样少量的处理器而言）。排队锁的优化数据分布没有什么作用，但是竞争也没有变得更坏。只有两个处理器时排队锁的性能很差，这是由构造软件队列时的特殊交互所造成的（Mellor-Crummey 和 Scott 1991）。尽管还要在大型机上进行试验，但平坦的曲线表明，总体上，数组锁和票号锁对于可扩展的高速缓存一致的机器性能相当好，也很可靠，至少用 LL-SC 实现时是这样的。对带指数回退的简单 LL-SC 锁而言，如果由于处理器在其本身的高速缓存中非公平地连续访问锁，使得锁在被释放和下一次加锁之间没有延迟，它的性能最好。复杂的排队锁并不必要，但在解锁和加锁之间有延迟时也表现得很好。 649 650

研究人员已经建议了对锁的更强的硬件支持。最突出的例子是硬件版的排队锁，其名称为 QOLB（在锁位排队）。由硬件管理正在等待锁的节点构成的分布链表，锁的释放者把锁交给第一个等待的节点而不影响其他的节点（Kägi、Bueger，and Goodman 1997）。因为 SCI 协议已经有硬件支持的等待节点的分布链表（即未决链标），QOLB 锁很适合于 SCI。硬件支持可以减少锁的传输时间，也可以减少锁流量对数据访问和一致性操作流量的打扰，但是，它并不能改变锁的微基准程序的扩展性，和系统所有的特性一样，它对性能的真正价值要由真实的应用和工作负荷来评价。

在 7.9 节我们讨论了栅障的算法及硬件支持。由于多个节点同时到达栅障导致了读-修改-写式访问共享计数器的竞争，从而引发了许多令人感兴趣的问题：在存储器中访问的计数器变量可缓存还是不可缓存？能否开发一种机制，允许处理器在自己的高速缓存中踏步等待，要么在释放时更新，要么从存储器而不是从释放者的高速缓存中读取释放值？Origin2000 这样的机器提供的对存储器 Fetch&op 操作的硬件支持是否特别有价值？

8.8.2 实现原子性原语

考虑对存储器单元执行的 test&set 这样的原子性交换（读－修改－写）原语的实现。为了确保其原子性，要求另外一个处理器对该存储单元的冲突写要么在读-修改-写操作的读部分之前进行，要么在其写部分结束之后进行。与第 5 章 5.5.3 节关于总线型机器的讨论一样，只要确保读结束之前没有实施对该块的作废，则一旦写操作部分已经相对于其他写操作被串行化，就允许读操作部分结束。如果读-修改-写在处理器上实现（使用可高速缓存原语），这就意味着一旦写获得了所有权就可以结束其读操作，甚至在作废确认返回之前。原子操作也可以在存储器实现，但是如果我们不允许任何处理器对处于脏态的块高速缓存，就 651

更容易实现它。这样所有的写都针对存储器，读－修改－写一旦到达存储器，就相对其他的写被串行化。存储器可以并行地发出对于读部分的响应和对应于写部分的作废。

实现 LL-SC 时为避免活锁的所有考虑都与总线型机器的情况一样，只是更复杂一点。回顾一下，如果条件存储失败，不应该发出作废或更新，否则，两个处理器就会一直相互作废或相互更新，从而失败，这就产生了活锁。为了探测出条件存储的失败，发出请求的处理器需要确定在条件存储之前，其他处理器对该存储块的写操作是否被串行化。在基于总线的系统中，通过针对条件存储检测高速缓存中是否不再有该块的有效副本，或者对已经出现在总线上的块是否有作废或更新，高速缓存控制器可以做到这一点。在具有分布互连网络的系统里，高速缓存控制器无法在本地完成对串行化次序的检测，因此需要不同的机制。在基于作废的协议中，如果一个存储块在高速缓存中还处于有效状态，那么对应于条件存储的排他读请求会被发往宿主节点的目录。在那里检查请求者是否还在共享链表中。如果不在，表明条件存储之前已经有另一个冲突的写，因此不用发出对应于该条件存储的作废，条件存储失败。否则的话，条件存储就成功。在基于更新的协议中，实现起来更困难，因为即使另外一个冲突写已经串行地排在条件存储之前，条件存储的请求者还是仍然在共享链表中。一个解决办法（Gharachorloo 1995）像对更新提供写的原子性一样，再次使用一个两阶段的协议。当条件存储到达目录时，它锁定对应块的目录项，使得其他的请求无法访问它。然后，目录向条件存储的请求者发回一个消息，请求者接到后进行检查，看看对 LL-SC 的锁标志是否已经被清除（清除是由在当前时刻和条件存储请求发出时刻之间到达的更新所实施的）。如果标志已被清除，条件存储已经失败，向目录发回一个失败消息（并将目录项解锁）。如果未清除，只要网络保证点对点的次序，我们就可以断定没有冲突写在目录处影响了该条件存储，该条件存储必然成功。请求者向目录发回一个确认，把目录项解锁，并发出对应条件存储的更新，条件存储成功。

8.9 对并行软件的影响

652

现在让我们从比同步更一般的角度来考虑对并行软件的影响。本章描述的一致的共享地址空间系统和第 5、6 章所描述的系统的区别在于，这里讨论的系统有物理上分布的主存储器，而不是集中式的主存储器。分布式存储器是通过数据局部性改善性能和扩展性的一个机会，但对发掘这种局部性的软件来说又是个负担。就像我们在第 3 章中所看到的，对于具有物理分布的存储器的高速缓存一致的体系结构（或 CC-NUMA 机），比如本章所讨论的那些体系结构，并行程序应该意识到物理上分布的存储器的存在，尤其当它们的重要的工作集不能容纳于高速缓存之中时。如果数据没有在对该数据发生容量型、冲突型或冷启动型扑空的节点的存储器中分配，就会出现人为的通信。这种情况甚至在数据能容纳于高速缓存时也会导致附加的通信，因为写扑空时查目录（包括更新）会产生网络流量和冲突。最后考虑一个多道程序的工作负载，为了负载均衡，应用进程在节点间迁移。迁移一个进程会将本地扑空变成远程扑空，除非系统把被迁移进程的全部数据也搬到新节点的主存储器中。由于以上种种原因，将数据在分布存储器中适当地分配是很重要的。

在本章讨论的 CC-NUMA 机中，主存的管理一般采取相当大的页面粒度。大粒度不利于适当分布共享的数据结构，因为本该分配到两个不同节点的数据可能会落在了同一个分配页面上。操作系统可能会依据硬件计数器得到的信息，透明地把页面迁移到高速缓存扑空发生

最频繁的节点上去；或者程序设计语言的运行时系统会根据用户提供的提示或编译器的分析来迁移页面。（我们知道 Origin2000 提供了高效迁移的协议支持。）现在更为一般的做法是由程序员指导操作系统，把页面放到最靠近特定进程的存储器中去。这和向系统提供“把这个虚拟地址范围内的页面放到进程 X 的本地存储器中去”这样的指示一样简单；或者，它可以额外包含对页边界填充和对齐的数据结构，以便适当地布置它们，它甚至可以要求按不同的方式组织数据结构，以允许按页粒度的放置。需要上述所有三种方法的例子可以在方程求解器内核和 Ocean 中找到，它们使用的是四维数组而不是二维数组。像这样简单而规整的情况也可以由复杂的编译器处理。但另一方面，在 Barnes-Hut 中，适当的数据布局却要求对数据结构以及代码重新组织。不再采用单个线性数组来存放所有的粒子（单元），每个进程在它的本地存储器中为分配给它的粒子安排一个属于自己的数组或链表，在时间步之间，将重新分配的粒子从一个数组或链表搬到另外一个当中去。但是，我们已经看到，数据布局对这个应用没有太大的作用，这是因为小的工作集和低的容量型扑空率，而且它的高代价甚至会影响性能。在使用数据迁移之前理解其代价和潜在的好处是重要的。软件控制的数据复制而不是迁移也存在类似的问题，下一章将讨论在主存中的一致复制和迁移的其他方法。 653

在一致的共享地址空间中程序员最难处理的问题是竞争。不仅隐式的无法预料的数据通信会引起竞争，“不可见”的协议事务也会引起竞争，如所有权请求、作废和确认等，程序员根本不会去考虑这些通信，而且这些通信是点对点的，不能用广播介质来分摊。所有这些类型的事务占用通信辅助部件的协议处理部分，因此保持每个事务对辅助部件非常低的占用度，以包容端点的竞争就变得重要了。不可见的协议消息和竞争要求程序员避免伪共享这样的性能问题，特别是当大量的协议事务都发向同一个节点的时候更为重要。因此，尽管用于固有通信及用于空间局部性和高速缓存块粒度的伪共享的软件技术与总线型机器上的一样，但它们对性能的潜在影响却不一样。例如，我们一般会构造某种类型的数据，如数组，使得每个进程对应一个数组单元。如果数组单元比页小，那么就会有几个单元落到同一页中。如果这些数组单元没有经过填充以避免伪共享，或者如果它们导致了高速缓存中的冲突扑空，所有的扑空和通信都会指向该页的宿主，引起显著的竞争。在具有分布式存储器的机器中，构造一个记录数组而不是多个标量数组（我们在第 5 章为避免伪共享就这样做了），填充记录使其与页对齐，并把页面放到适当的本地存储器中去，都是有好处的。

高性能的并行 FFT 给出了一个有趣的例子，说明竞争是如何导致消息传递型系统和共享地址空间型系统使用不同的调度策略的。从概念上来讲，该计算是分阶段的，局部的计算阶段被通信阶段所分隔，通信阶段涉及矩阵的转置。一个进程从源矩阵读取一列，把它写入目标矩阵指定的行，然后对目标矩阵中分配给它的行执行局部计算。在消息传递型系统中，把数据拼接成大的消息很重要，为了性能有必要对通信也这样做（作为独立于计算的阶段）。但是，在高速缓存一致的共享地址空间有两点不同。首先，传输总是以高速缓存块的粒度进行的。其次，每个细粒度传输都涉及作废和确认（进程写入的每个本地块可能在另一个处理器的高速缓存中处于共享状态，所以必须被作废），导致了对一致性控制器的竞争。所以，在计算正在进行的时候按需要以细粒度执行通信，而不是在独立的转置阶段一起进行，效果也许更好一些；这样做错开了通信，缓解了控制器上的竞争：一个本来要在转置后使用目标矩阵的行进行计算的进程，现在可以在计算时根据需要从远程节点读对应的源矩阵的列中的字、在这个过程中进行转置。哪种方法更好则取决于其体系结构。 654

最后应指出的是，可扩展系统中的同步代价可能很高，因此在程序上必须做出特别的努力，降低高竞争锁或者全局栅障同步的频率。

8.10 高级论题

在对本章进行总结之前，我们介绍两个额外的论题。第一个是关于减少扁平的、基于存储器的目录方案的目录存储开销的实际技术；第二个是层次式一致性的技术，包括侦听的和基于目录的技术。

8.10.1 减少目录存储的开销

在8.2.3节我们讨论了扁平的、基于存储器的目录，提到过可以使用有限数量的指针而不是全位向量来降低目录的大小或宽度，这样做要求在块副本数超过可用指针数的时候有某种溢出机制。根据共享模式的经验数据，在有限指针目录中提供的硬件指针的数量一般很小，溢出机制的效率非常重要。本节首先讨论某些可能的溢出方法。然后考察通过将目录组织成高速缓存，而不是系统中每个存储块对应一个目录项，从而减少目录项数或降低目录"高度"的技术。具有 i 个指针的有限指针方案被命名为 Dir_i，后跟溢出方法的缩写。溢出方法包括广播、非广播、粗粒度向量、软件溢出和动态指针。

1. 降低目录宽度的溢出方法

广播或 Dir_iB 方案（Agarwal et al. 1988）中的溢出策略是当可用指针 i 用完时，在目录项中设置一个广播位。当对该块再次写入时，向系统所有节点发出作废消息，不管它们是否高速缓存了该块。向未高速缓存该块的处理器发出作废消息在语义上没有错误；但是，这可能会浪费网络带宽，如果执行写操作的处理器，在继续之前必须等待确认，时延会上升。广播法的优点在于其简单性。

非广播或 Dir_iNB 方案（Agarwal et al. 1988）从不允许块的有效副本的数量超过 i，从而避免了广播。无论何时，如果共享者的数量达到 i，而另一个节点又请求该块的一个共享副本，协议会把现存的一个共享者的副本作废掉，为新的请求者释放目录项中的那个指针。这个方案的主要的缺点是它没有考虑在一段时间内被许多处理器频繁访问的数据（比如，预计算的值表或者甚至是程序代码），由于这些副本会被不必要地作废，结果导致一连串的扑空。虽然可以对包含代码的块采取特殊措施（例如，可以用软件而不是硬件来管理其同一性），但是如何用该方案处理广泛共享的只读数据还不清楚。

655

粗粒度向量或 Dir_iCV_r 方案（Gupta，Weber，and Mowry 1990）在它的初始表示中也使用 i 个指针，但是溢出时，其表示变为一个粗粒度的位向量，类似于 Origin2000 用于大配置机器的表示方法。在这种表示方法中，目录项的每一个位表示的不是一个节点，而是系统中一个节点组（Dir_iCV_r 中的下标 r 表示组中的节点数），一旦组内的一个节点高速缓存了该块，就把对应位置位（如图8-35所示）。当一个处理器对该块写入，要向对应位被置位的所有组中的所有节点发出一个作废，不管它们是否确实访问过或高速缓存了该块。作为一个例子，考虑一台256个节点的机器，我们在目录项中为其存储了8个指针。由于每个指针的宽度为8位，溢出时粗粒度向量有64位。这样，我们可以实现一个 Dir_8CV_4 方案，每个粗向量位指向一个256/64即4个节点的组。每个目录项另外加一位，指示当前的表示是正常的有限指针

还是粗向量。如图 8-36 所示，一个像 Dir_iCV_r 这样的方案优于 Dir_iB 和 Dir_iNB 方案之处，在于它对不同的共享模式表现得更为健壮。

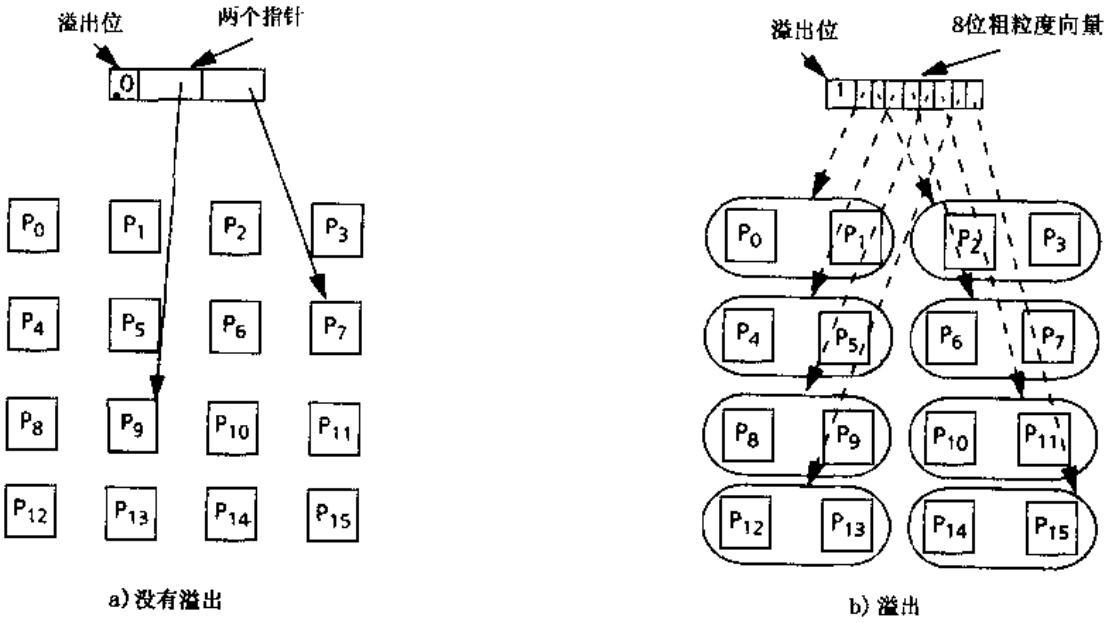


图 8-35　溢出时从有限指针表示变为粗向量表示。出现溢出时，两个 4 位的指针（对于 16 个节点的系统）被看作一个 8 位的粗向量，每位对应两个节点的组。由于也设置了溢出位，所以可以很容易地判断出其表示方式。b）图中的虚线表示粗向量位与节点组之间的对应关系

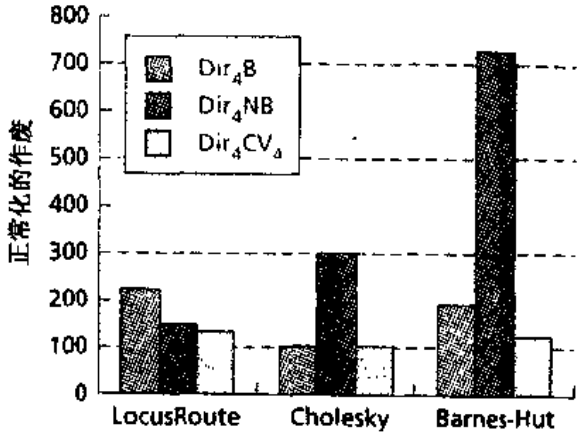


图 8-36　粗向量溢出方法与广播和非广播溢出方法的健壮性比较。图中给出 Dir_4B、Dir_4BN 和 Dir_4CV_4 三种方案所引起的作废流量的比较，流量相对于全位向量方案进行了规格化（全位向量方案表示为 100 个作废），结果取自于（Weber 1993），因此其模拟参数与书中用的不同。处理器数量为 64（每节点一个处理器）。布线路由应用 LocusRoute 具有被频繁写入并被许多节点读出的数据，它的实验数据显示了 Dir_iB 方案的潜在缺点。Cholesky 和 Barnes-Hut 具有被大量处理器读共享的数据（例如，Barnes-Hut 中靠近树根的节点），结果显示 Dir_iNB 方案的潜在缺点。可以发现，Dir_iCV_r 方案是相当健壮的

软件溢出或 Dir_iSW 方案与前面所讲的几种方案不同，发生溢出时它并不丢弃块的精确的高速缓存状态。而是把当前的 i 个指针和指向新的共享者的指针用软件存入该节点本地主存的特殊地方。腾出空间用于新的指针，这样，在必须再次调用软件将指针存入存储器之前，硬件可以处理 i 个新的共享者。溢出导致一个硬件溢出位的置位。该位保证在碰到后续

的写操作时，已经存入存储器的指针将被读出，并且也会向那些节点发送作废消息。在没有非常复杂的（可编程的）通信辅助部件情况下，溢出现场（指针必须被写入存储器和指针必须从存储器中读出并且发送作废消息）由运行在主处理器上的软件处理；因此，当这些事务发生时，处理器必须被中断，或必须产生一个陷阱。这个方案的优点在于即使发生了溢出，也保存共享者的精确信息，与全位向量（或无限指针）方案比较，并没有产生额外的作废流量，溢出处理的复杂性由软件管理。主要的额外开销是中断和软件处理的代价。其缺点表现为三种形式：1）块的宿主节点的处理器把时间用于中断的处理而不是执行用户的计算；2）中断和中断请求处理的额外开销大，会成为潜在的竞争瓶颈，并减慢其他的请求；3）发出请求的处理器暂停的时间较长，因为能引起中断的请求的时延较大，也因为竞争的增加。[⊖]

656 ~ 657

MIT 的 Alewife 研究原型（Agarwal et al. 1995）使用了有限指针目录的软件溢出方案，称之为 LimitLESS 方案（Agarwal et al. 1991）。Alewife 机的设计允许它扩展到 512 个处理器，每个节点上一个处理器。每个目录项宽度为 64 位。它包含 5 个 9 位的指针，记录高速缓存了该块的远程节点，一个专用的位表示本地节点是否也高速缓存了该块（如果高速缓存了则省了 8 位）。溢出指针存放在主存储器中的一个散列表里。Alewife 的主处理器具有对多线程的硬件支持（见第 11 章），支持溢出时自陷的快速处理。在 16 个处理器的系统上，虽然引起 5 个作废并能以硬件处理的请求的时延仅仅是 84 个时钟周期，但是，引起 6 个作废从而需要软件干预的请求却要花费 707 个时钟周期。

动态指针或 Dir_iDP 方案（Simoni and Horowitz 1991）是 Dir_iSW 方案的一个变种。除了有 i 个硬件指针外，该方案中每个目录项含有一个指向本地节点主存的一个特殊部分的硬件指针。这个特殊的存储器带有一个自由链表，需要时可以从中动态地将指针结构分配给处理器。该方案与 Dir_iSW 关键的不同之处在于其所有的链表操作都是由一专用协议处理器的硬件完成，而不是交给本地节点的通用处理器处理。结果是不需要中断，操纵链表的额外开销也很小。因为它也包含一个指向存储器的硬件指针，所以该方案使用的硬件指针数 i 一般很小。Dir_iDP 方案是 Stanford 的 FLASH 多处理器（Kuskin et al. 1994）的缺省目录组织。因为动态指针池是有限的，而且在作废时需要对表遍历，该方案一般还伴随使用替换提示。

所有这些在基于存储器的协议中维护目录信息的方案当中，显然 Dir_iB 和 Dir_iNB 方案对不同的共享模式不是很健壮。但是，对于大型机上的实际应用而言，人们还没有能很好地理解这些方案的实际性能（或者说性价比）权衡。一般的看法是：全位向量对于包含中等数量对目录协议可见的处理节点的机器是合适的。硬件溢出的最佳候选方案应该是粗向量和动态指针：前者的缺点在于溢出时精度不够，后者的缺点则是由于硬件链表处理和自由链表管理，处理开销较大。

658

2. 减少目录的高度

除了降低目录宽度以外，另一个与其正交的降低目录存储器开销的方法是降低目录项的总数，做法是不再让每个存储器块都对应一个目录项（Gupta，Weber，and Mowry 1990；O'Krafka and Newton 1990）；也就是说，从目录存储器开销的表达式 $P \times M$ 中的 M 项入手。

⊖ 实际上在处理自陷之前响应请求者是可能的，因此不会影响请求者所看到的时延。但是，这意味着对该节点的下一个处理器请求被延迟了，处理器会经历一个暂停。

因为这两种降低额外开销的方法是正交的，那么就可以在两者之间进行折中：减少项的数量允许我们在不增加成本的情况下提高项的宽度（使用更多的硬件指针）；反之亦然。

启发我们使用更少的目录项的事实是，高速缓存的总量比机器主存储器的总量小得多。这意味着在给定时刻只有很小一部分存储块在高速缓存中。例如，每个处理节点可以有 1 MB的高速缓存和 64 MB 的主存储器。如果每个存储块有一个目录项，那么在整个机器中，将有 63/64（即 98.5%）的目录项将对应于未在机器任何地方高速缓存的存储块。这样大量的目录项被闲置，没有任何位被置位（特别是当使用了替换提示时）。将目录按高速缓存组织，像对包含程序数据的存储器块分配高速缓存行那样，动态地为目录项分配高速缓存，可以避免存储器的浪费。事实上，如果这个目录高速缓存的项数足够的小，我们可以用快速的 SRAM 而不是较慢的 DRAM 来实现目录，从而降低了目录的访问时间。我们知道，对于许多类型的存储器访问来说，这个访问时间是在决定处理器可见的时延的关键路径中。这种目录结构被称为稀疏目录（sparse directory）。（在 8.6.8 节中介绍的 HAL S1 系统使用这种方法）。

虽然稀疏目录的操作与正规的处理器高速缓存很相似，但也存在一些显著的区别。首先，这个高速缓存不需要一个后备存储器：当一个项被从中替换的时候，如果其中任何的节点位（或者指针）已被置位，我们就简单地向这些节点发送作废或腾空消息。其次，这个高速缓存中，每个存储块只有惟一的一个目录项，因此不存在空间局部性。第三，稀疏目录处理来自所有处理器的访问，而处理器的高速缓存只能被与之相连的处理器访问。最后，稀疏目录看到的访问流是经过过滤的，仅包含不能在处理器高速缓存中满足的访问。为了让稀疏目录不成为瓶颈，重要的是它要足够的大，并具有足够的相关度，这样对正在访问的块不会发生太多的替换。研究稀疏目录规模的某些试验和分析可以在（Weber 1993）中找到。

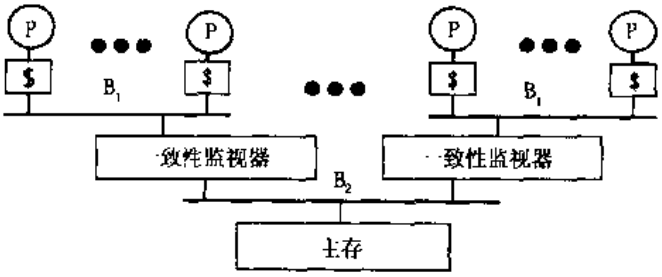
8.10.2 层次式的一致性

本章开头提到了建造可扩展的一致性机器的一种方法，就是层次式地扩展第 5、6 章讨论的基于总线或者环的侦听一致协议。本章还介绍了层次式的目录方案。这一节我们进一步介绍层次一致性方案。虽然一些商用系统（如 KSR1［Frank，Burkhardt，and Rothnie 1993］）和研究原型（如 Toronto 大学的 Hector 系统［Vranesic et al. 1991；Farkas，Vranesic，and Stumm 1992］）中采用了基于环的层次侦听方案，学术界也研究了层次目录，但是这些研究并没有取得多大的进展。尽管如此，层次式地以较小的系统构造大型系统仍然是一种有吸引力的抽象，对于理解基础技术也很有用处。 659

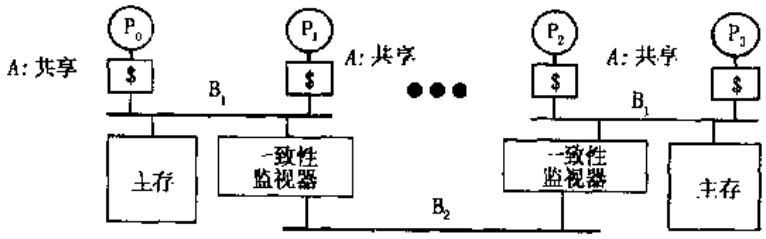
1. 层次式侦听

基于总线和基于环的层次式侦听的问题类似，因此我们主要通过前者来研究。总线层次结构是一棵总线的树。叶节点是基于总线的多处理器，其中包含处理器。构成树的内部节点的总线不包含处理器，但是用于互连和一致性控制：它们允许事务被侦听，在必要时向上层和下层传递。层次式的机器可以将主存储器集中在根节点，也可以将它分布在叶节点的多处理器中（如图 8-37 所示）。尽管集中式存储器可以简化程序设计，但如果能发掘局部性，分布式存储器则有带宽和性能上的优势。（但是注意，如果数据的分布无法使大多数高速缓存扑空在本地得到满足，在最坏的情况下，远程数据实际上会比系统的根节点还远，因此可能导致更糟糕的性能。）此外，使用分布式存储器，层次中的叶节点是一个基于总线的完整的多处理器，它已经是市场上的产品，具有成本优势。下面我们将集中研究具有分布式存储器

的层次结构，集中式存储器的层次结构将留作练习。



a) 集中式的存储器



b) 分布式存储器

图 8-37　基于总线的层次式多处理器，显示两级层次结构。主存储器可以集中在根节点，也可以是物理上分布的，一致性监测器连接父总线和子总线

叶节点（多处理器）内的处理器高速缓存可以通过第 5 章所讨论的任何侦听协议保持一致。在一个简单的二级层次结构中，我们用另外一条总线（B_2）将几个这样的基于总线的系统连接到一起。（向多级层次结构的扩展是简单的）我们需要的是在每一条 B_1 总线上附带一个一致性监测器，它监视（侦听）两条总线上的事务，决定 B_1 总线上的哪一个事务应该转递到 B_2 总线上，B_2 总线上出现的哪一个事物应该转发到它的 B_1 总线上。该装置就好像一个过滤器，只是在两个方向上转发需要的事务，因此降低对总线的带宽要求。

在具有分布式存储器的系统中，节点的一致性监测器必须关心两种类型的数据，其事务可以出现在 B_1 或 B_2 总线上：在远程存储器中分配但是被本地节点的某些处理器高速缓存了的数据，在本地分配但是被远程高速缓存的数据。为了监视前一种数据，可以像 Sequent 的 NUMA-Q 那样为每个节点设置一个远程访问高速缓存或称远程高速缓存。该高速缓存维护与该节点任何处理器高速缓存中缓存的远地数据的包容性（见第 6 章 6.3.1 节），每个块包含一个“脏但陈旧”位，指示该节点的一个处理器高速缓存中该块为脏（即分配在本地存储器中但是没有进入远程高速缓存的数据）。这提供了足够的信息以判断出每个方向上哪些事务相关，并转发它们。

对于本地分配的数据，总线事务可完全由本地存储器或者高速缓存处理，除非该数据被其他（远程）节点的处理器所缓存。对于后一种数据，没有必要将数据本身保存在这个一致性监测器中，因为有效数据要么已经存在于本地，要么在远地处于被修改状态；事实上，我们不想在那儿保存它，因为数据量可能和本地存储器一样大。但是，监测器保存了该数据的状态信息，并且侦听本地 B_1 总线，这样与此数据相关的事务在需要时可以转发到 B_2 总线

上。我们把一致性监测器的这一部分叫做本地状态监测器。最后，一致性监测器也监视 B_2 总线上指向其本地地址的事务，并将它们传递到本地总线 B_1 上，除非本地状态监视器发现它们在远地以被修改状态缓存。对于 B_1 和 B_2 总线事务，远程高速缓存和本地状态监测器都要被查找。

考虑 8.1 节中概述的一致性协议的三个功能：1）在本地节点的一致性监测器（远程高速缓存和本地状态监测器）中隐式地保存关于层次结构中其他节点的状态的足够信息，以决定采取什么行动；2）如果该信息指示需要在本地节点之外寻找其他的副本，向下一级总线（在层次结构中更深的一层）广播请求和搜索，其他相关的监测器会响应；3）作为通过总线上的层次式广播发现副本的动作的一部分，与其他副本的通信可同时进行。

660 ~ 661

让我们进一步考察一下读扑空的路径，假设共享的物理地址空间。本地总线 B_1 上出现一个 BusRd 请求。如果远程访问高速缓存、本地存储器或另一个本地处理器高速缓存中有该块的有效副本，它们就会提供这个数据。否则，远程高速缓存或者本地状态监测器就会知道要把请求传到 B_2 总线上。当这个请求出现在 B_2 上时，其他节点的一致性监测器将侦听到它。如果一个节点的本地状态监测器判定本节点有这个数据的有效副本，它会把请求传到它的 B_1 总线上去，等待响应，然后把响应放回到 B_2 总线上去。如果一个节点的远程高速缓存中含有这个数据，而数据处于共享状态，它会简单地把应答放到 B_2 总线上；如果数据处于脏状态，它会应答并且在其 B_1 总线上广播一个读请求，使处于脏的处理器高速缓存把块的状态降级为共享；如果状态是脏但陈旧，它会简单地在其 B_1 总线上广播一个读请求，然后用读回的结果作为应答。对于最后一种情况，有脏数据的处理器高速缓存会将其状态由脏改为共享，并且把数据放到 B_1 总线上。远程高速缓存将从 B_1 总线上接收数据应答，把它的状态从脏但陈旧改为共享，并且把应答传递到 B_2 总线上。当数据应答出现在 B_2 总线上，请求者的一致性监视器得到它，如果合适，在它的远程高速缓存中将它就位，修改状态，并把它放到它的本地总线 B_1 上。（如果该块必须放置在远程高速缓存中，它可能替换掉某个其他的块，这会触发 B_1 总线上的一个清空/作废请求，以保证包容性的性质。）最后，请求的高速缓存从 B_1 总线得到对它的 BusRd 请求的响应，以共享状态存储它。

对于写入，考虑如图 8-37b 所示的特殊情况。左边的节点中的 P_0 对位置 *A* 发出一个写，*A* 分配在第三个节点（图中没有显示）中。因为 P_0 自己的高速缓存只有处于共享状态的数据，因此在本地的 B_1 上发出一个所有权请求（BusUpgr）。结果，P_1 的高速缓存中的 *A* 的副本被作废。因为远程高速缓存中没有处于脏但陈旧状态的对应块（这很可能是错误的，因为它在 P_1 中的状态为共享），监测器把 BusUpgr 请求传到总线 B_2 上，以作废系统中该块的任何其他副本，同时把远程高速缓存中该块的状态更新为脏但陈旧。在另一个节点上，P_2 和 P_3 的高速缓存中该块的状态为共享。由于包含性质，该块在这两个节点的远程高速缓存中也一定处于共享状态。因此这个远程高速缓存将该 BusUpgr 请求从 B_2 传到它的本地总线 B_1、作废自己的副本。当请求出现在总线 B_1 上时，P_2 和 P_3 的高速缓存中 *A* 的副本被作废。如果 B_2 总线上的一个节点的处理器没有缓存含有 *A* 的块，升级请求 BusUpgr 就不会传到它的 B_1 总线上。现在假设左边节点中的另外一个处理器 P_4 发出对位置 *B* 的存储请求。这个请求在本地节点就可以满足，由 P_0 的高速缓存提供该数据，远程高速缓存保持脏但陈旧状态，没

662 有事务会被传递到 B_2 总线上。

这里对处理器高速缓存和高速缓存控制器的实现要求与第 6 章中讨论的一样。但是对远程访问高速缓存有一些约束。它应该比处理器高速缓存的总和大一些，为了维持包容性而不引起过多的替换，应该有相当大的相关度。它还应该是免锁定（lockup-free）的，就是说，在某些请求仍然是待完成时，能够同时处理来自本地节点处理器的多个请求（详见第 11 章）。最后，无论何时从远程高速缓存中替换掉一个块，根据被替换的块的状态（共享或脏但陈旧），必须将一个作废或腾空请求发送到总线 B_1 上。减少远程高速缓存的访问时间没有增加其命中率那么关键，由于它不在影响处理器时钟频率的关键路径上。所以远程高速缓存一般用 DRAM 而不是 SRAM 来构造。远程高速缓存控制器也必须处理第 6 章所讨论的请求和获得总线中的非原子性问题。

最后，考虑写串行化和判定存储结束。根据第 6 章中讨论过的在单总线上的实现方式，两个请求的串行化显然应该由它们在离根最近的总线上出现的次序来决定。对于在同一个叶节点中完全得到满足的写，它们被其他的处理器（在或不在这个叶节点内）看到的次序是本地总线 B_1 提供的它们的串行化次序。类似地，对于完全在同一个子树中满足的写，它们被其他的处理器（在或不在该子树中）看到的次序是该子树的根总线所决定的串行化次序。如果将挂在一条公共总线上的每条总线看作一个处理器，递归地使用第 5、6 章中对单总线所使用的同样的推理，我们比较容易理解这一点。类似地，顺序同一性需要判定存储的结束，处理器不能假定它的存储操作已经被提交，直到该操作出现在它将要到达的离根最近的总线上。确认（现在可能必须是一个显式的总线事务）一直要到这个时刻才能产生，甚至直到那时，都必须保持该确认在返回请求处理器的途中和其他事务之间的正确次序（见习题 8.26）。一旦从总线上发回了确认，向下传给处理器高速缓存的作废请求本身不需要确认，只要沿路径保持正确的次序就可以了（和第 6 章的多级高速缓存层次结构一样）。

一种最早采用这种层次式侦听总线的分布存储型机器是 Encore Corporation 公司的 Gigamax（Wilson 1987；Woodburdy et al. 1989）。系统包含最多 8 台 Encore Multimax 机（各是一台常规的基于侦听总线的多处理器），由光纤链路连接到另外一条（第 9 条）全局总线上，形成二级层次结构。其框图如图 8-38 所示。每个节点扩充了一块统一互连卡（UIC）和一个统一簇（节点）高速缓存（UCC）卡。UCC 是远程访问高速缓存，而 UIC 是本地状态监测器。由于 Gigamax 的特殊组织结构，它的全局总线监测的实现有些不同。节点通过光纤链路连接到全局总线上，所以当节点的远程访问高速缓存（UCC）缓存远程数据时，它就不直接侦听全局总线。每个节点在全局总线上都有一个辅助 UIC，它监测针对已被缓存在本地节点的远程存储块的全局总线事务。一旦监测到，将相关的请求传递到本地总线上。如果 UCC 也直接挂在全局总线上，就不需要全局总线上的 UIC 了。Gigamax 使用光纤链路，每个节点也不是只使用单个跨在本地和全局总线上的 UIC，其原因是高速总线通常都很短：Encore Multimax 和 Gigamax 中用的纳总线（Nanobus）长度仅为 1 英尺（光在 1 纳秒内走 1 英尺，总线因此而得名）。因为节点至少 1 英尺宽，而全局总线也 1 英尺宽，需要柔性的缆线才能把它们连到一起。使用光纤，可以让链路相当长而不影响其传输性能。

663

将侦听式的高速缓存一致性扩展到环的层次结构，非常类似于将其扩展到具有分布存储器的总线层次结构。图 8-39 显示了其结构框图。局部环和相连的处理器构成节点，节点由一个或多个全局环连接。一致性监测器是环间的接口，其角色和总线层次结构中的一致性监测器一样。

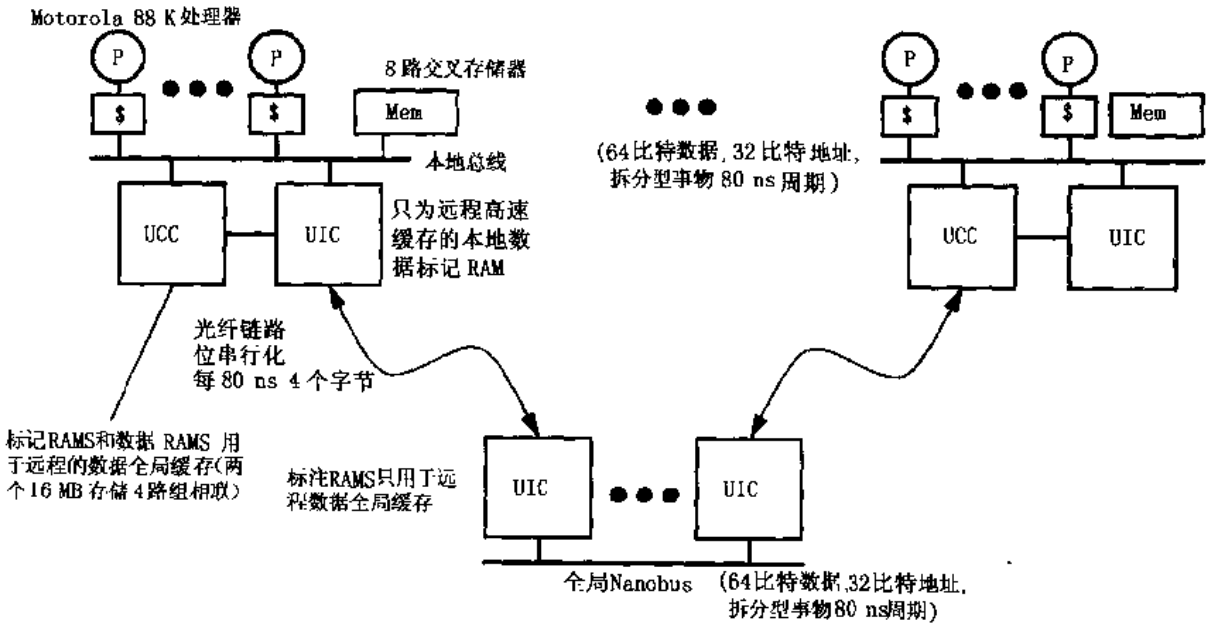


图 8-38　Encore Gigamax 多处理器的块图表。使用两级的总线层次结构，存储器分布在叶节点中

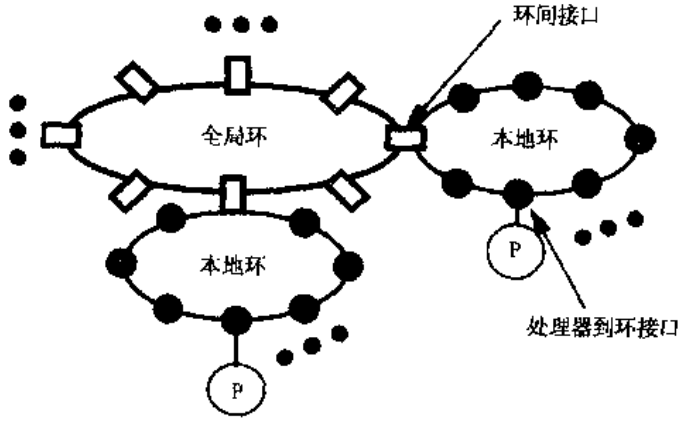


图 8-39　基于层次环的多处理器的结构框图。在图中所示的二级层次结构中，每个局部环在全局环看来是一个节点，环间接口在两者之间传递相关事务

2. 层次式目录方案

层次式目录方案使用点对点的网络事务而不是侦听。但是，正如前面所讨论的，与扁平的目录方案不同，层次式目录中的目录信息的来源不是在一个固定节点上得到的。副本的位置既不是在固定的宿主节点中找到，也不是通过遍历宿主所指向的分布链表中找到的。作废消息也不是直接发给拥有副本的节点。所有这些动作都是通过在节点构成的层次结构（树）向上或向下发送消息来完成的，而且只有在树中的父子节点之间才有直接的通信。 [664]

初看上去，层次目录的组织结构与层次侦听很相似。考虑图 8-40 中的例子。处理节点位于树的叶子，主存储器在处理节点中分布。每个块所在的存储器叫做它的宿主存储器（在叶节点），但是这并不意味着目录信息在那里维护或出自那里。树的内部节点不是处理节点而只是保存目录信息。每个这样的目录节点记录了其子树缓存或者记录的所有存储器块。它为每个数据块设置一个存在位向量，指示它的哪些子树有该块的副本，另外用一位来表示是否副本之一为脏。它也记录被其子树之外的处理节点所缓存的本地存储器块（即分配在其某

个子节点的本地存储器中的块）的信息。与层次侦听一样，用这个信息确定什么时候子树内发出的请求应该被上传到更上一级目录中。因为靠近根部的目录节点中维护的目录信息量非常大，因此目录信息一般组织成高速缓存以减少其尺寸，保持与其子节点的高速缓存或者目录的包容特性。这就要求在树的某一层次的一个目录高速缓存进行替换时，被替换的块也必须从树中其所有的子孙目录中去掉。类似地，关于分配在子树中的一个块的信息被替换时，要求作废或清空存在于子树之外的节点所保存的该块的副本。这些操作的代价可能很高。

665

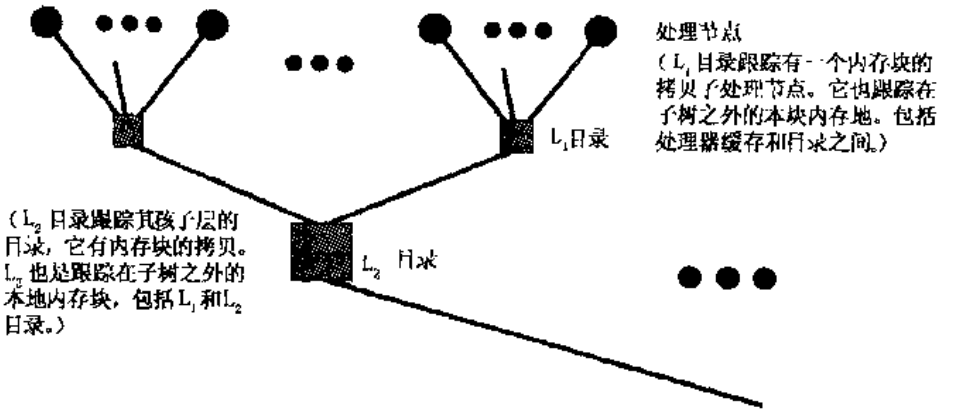


图 8-40　层次式目录的组织结构。处理节点位于逻辑树的叶子，内部节点只包含目录信息，对每个高速缓存的存储块有一棵逻辑树。逻辑树可以嵌入到任意的物理层次结构中

来自节点的读扑空沿着层次结构向上走，要么直至到达一个目录，指出其子树中具有被请求块的一个副本（无论干净或脏），要么直到请求到达作为请求节点和该块的宿主节点的第一个公共祖先的目录，这个目录表明在子树外该块不处于脏状态。然后，请求沿层次结构向下走，到达正确的处理节点，取得数据。应答数据沿着同样的路径返回，沿途更新目录，如果该块为脏，它的副本也要被送到其宿主节点去。

对高速缓存的写扑空也沿着层次结构向上走，直到到达一个目录，其子树包含被请求的存储器块的当前拥有者的目录。如果块是干净的，拥有者就是宿主节点，否则，拥有者是脏的高速缓存。请求向下行进到拥有者，取得数据，请求节点成为块的新的拥有者。如果该块原来处于干净状态，还要通过层次结构向所有缓存了该存储器块的节点发出作废命令。最后，上述存储器操作涉及的所有目录都被更新，反映新的拥有者和被作废的副本。

在层次式侦听方案中，互连网络物理上是层次的，允许侦听。使用点对点的通信，层次式目录不需要依赖于物理上的层次互连。这里讨论的层次结构是逻辑的层次结构或层次的数据结构。它可以在物理的层次式网络（即实际的树状网络，目录高速缓存位于内部节点，叶子是处理节点）上实现，或建立在网格这样的通用非层次式网络之上，层次式目录嵌入通用的网络之中。事实上，每个被高速缓存的块有一个独立的层次目录结构。所以，通用网络中的同一个物理节点对某些块而言可以是叶（处理）节点，而对其他块而言却可以是内部（目录）节点（见图 8-41）。

666

最后，层次式目录的存储额外开销有着吸引人的可扩展性。它是每一级目录高速缓存的代价。越到靠近根部的上层，其目录中的项数越多（为了维护包容性，不产生过多的替换），但是目录的数量越小。其结果是，任何一层的所有目录所需目录存储器总容量差不多都一样。所需要的目录存储容量和主存储器的大小不成比例，而是和处理节点的高速缓存的容量

成比例。总的目录存储器额外开销与主存储器的比例关系是

$$\frac{C \times \log_b P}{M \times B}$$

这里 C 为每个叶处理节点的高速缓存的尺寸，M 是每个节点的主存储器的大小，B 是以比特为单位的存储器块的大小，b 是层次结构的分支因子，P 是处于叶子位置的处理节点的数量（因此 $\log_b P$ 是树的层数）。有关层次目录方案更详细的信息可以在文献（Scoot 1991；Wallach 1992；Hagersten 1992；Joe 1995）中找到。

667

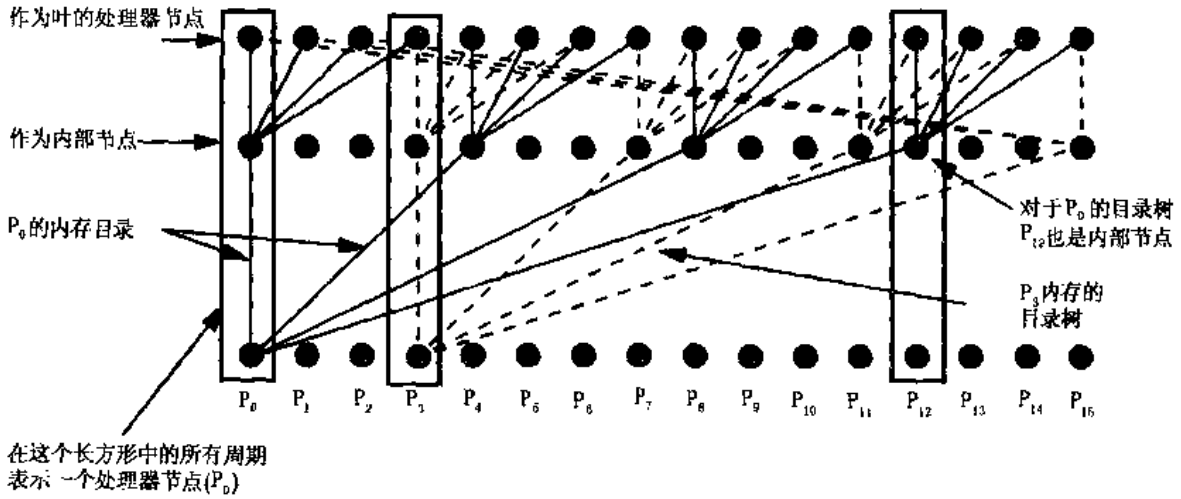


图 8-41 嵌入在任意网络中的多根层次式目录。图中显示了具有 16 个节点的层次结构。对位于处理节点的主存储器中的块而言，节点本身是（逻辑）目录树的根。因此，有 P 个处理节点，就有 P 棵目录树。图中只显示了其中的两棵。除了作为其本地存储器的目录树的根节点外，处理节点还是其他处理节点的目录树的内部节点。存储器块的地址隐含地说明了特定的目录树，并引导在该目录树中父子节点之间的实际遍历

3. 层次式一致性的性能含义

层次式协议，不管是侦听还是目录式的，有着某些潜在的性能优势，这是对前面讨论的二级协议的优点的延伸。其一是，当对于一个块的请求沿层次结构上下移动时，可以将它们合并。如果一个处理节点正在等待一个存储块的到来，另一个请求同一块的处理节点可以从它们的公共祖先目录那里得知该块已经被请求。那么它就可以在这个中间目录处等待，当响应返回时接收它，而不是再发出一个重复的请求。这种组合的事务可以减少流量，从而减少竞争。作废命令的发出和作废确认的收集也可以通过树结构分层地进行。另一个优点是当发生扑空时，如果层次结构中的一个邻近节点上有该块的高速缓存副本，那么就可以从那个邻近节点得到该块（高速缓存到高速缓存的共享），而不必到宿主节点上去取，宿主节点在网络拓扑上可能相距较远。这样可以降低传输时延和宿主节点上的竞争。当然，这第二个优点取决于层次结构的局部性与底层的物理网络的局部性之间是否有良好的对应关系，也取决于应用的共享模式与层次结构是否匹配。

尽管树形网络中的局部性可以降低链路中的传输时延，尤其在非常大型的机器上，但层次方案的总的时延和带宽特征通常不占优势。首先考虑层次式侦听。对于总线，沿途中每个总线上都有一个总线事务和侦听的时延；对于环，在层次结构的每一级遍历，环进一步把时

延提高到非常高的程度。例如，在一台满配置的 Kendall Square Research 的 KSR1 机（Frank, Burkhardt, and Rothnie 1993）上访问远程环的一个单元的无竞争时延超过了 25 μs（Saavedra, Gainies, and Carlton 1993），因此采用了其他的体系结构技术（在第 9 章讨论）来减少远程环的容量型扑空。使用层次侦听的商用系统趋向于采用相当浅的层次结构（最大的 KSR 机采用二级的环层次结构，每个环上最多 32 个节点）。一个节点中有几个处理器的事实意味着节点与它的父节点和子节点间的带宽必须高，以支持它们的综合需求。节点内的处理器不仅竞争总线或链路的带宽，而且竞争侦听带宽和节点对网络接口的占用度、缓冲和请求记录机制。为了缓解靠近层次结构根部链路带宽的限制，在靠近根的地方可以使用多总线或多环；但是，实际的层次系统中带宽的扩展性仍然是相当有限的。

对层次式目录而言，时延问题在于为了满足一个请求，在层次间上下传递的网络事务数比扁平的、基于存储器的方案要大。即使这些事务在网络中可能有更好的局部性，但是每个事务都是完整的网络事务，需要查询或者修改其中间或目的节点的目录。这增加了沿关键路径节点上的额外开销，很可能抵消了减少所经过的网络跳数从而降低网络延迟所带来的好处，尤其是对现代的网络更是如此。虽然在实际中可以使用某种流水线技术，例如，当一个目录节点正被更新时，数据应答可以传递给请求节点，但与没有层次结构的机器相比，其时延还是相当大的（Hagersten 1992; Joe 1995）。具有大分支因子的层次结构能缓解时延问题，但是却增加了竞争。和层次式侦听一样，目录层次结构的根可能成为带宽瓶颈，包括链路带宽和目录查找带宽。可以在靠近根部的地方使用多条链路（特别适合于物理的层次式网络[Leiserson et al. 1996]），目录高速缓存可以在它们之中交错设置。另外，因为每一块都有一个独立的逻辑层次结构，因此可将一个多根的目录层次结构嵌入到非层次的、可扩展的点对点的互连网络中去（Scott 1991; Wallach 1992; Scott and Goodman 1993）。图 8-41 显示了一种可能的组织结构；但是，和层次式目录结构一样，这些技术仍然处于探索研究阶段。

[668]

8.11 结论

支持一致的共享地址空间的可扩展系统在多处理器领域中越来越重要，因为它们综合了一致共享地址空间编程模型的容易编程和分布式存储器和互连网络的可扩展优势。为技术和商用工作负载设计的现代商品化多处理器机器越来越流行用硬件支持高速缓存一致性。这些系统中的大部分都采用基于目录的协议，不管是基于存储器的协议还是基于高速缓存的协议。至少对目前已经建造的中等规模的机器而言，它们的性能都不错，对于很多应用，这些机器的编程比显式的消息传递型机器要容易。

基于目录的高速缓存一致性协议相当的复杂，有很多过渡状态和“边角情形”需要处理。图 8-42 通过 Origin2000 和 NUMA-Q 协议基本上完整的状态转换图来显示这种复杂性。

尽管以硬件提供对高速缓存一致性的支持的设计成本相当高，但是，随着经验的增加、标准的出现以及微处理器本身提供高速缓存一致性支持的事实，这种设计成本会下降。一旦获得微处理器一致性协议，设计者就可以在处理器推出之前开发多处理器协议和通信体系结构，这样处理器的推出和系统设计完成两者之间不会有多大时差。今天的商用的多处理器一般都使用系统出厂时最新的微处理器，减少了多道程序编程人员对追赶处理器技术发展曲线的恐惧感。

硬件实现一致性的共享地址空间系统有一些令人感兴趣的未解决问题，包括：实际应用当中，其性能能否扩展到处理器数量很大的情况（要做到这一点，是否要对当前的协议做显著的修改）？可扩展系统的节点应该是小规模的多处理器还是单处理器？市场上商品化的通信体系结构能在多大程度上成功地支持这种抽象？能否成功地设计通信辅助部件，使其支持面向高速缓存一致性和显式的消息传递两者的适当的机制？具有一致的共享地址空间的系统的某些关键的硬/软件折中将在下一章讨论。 669

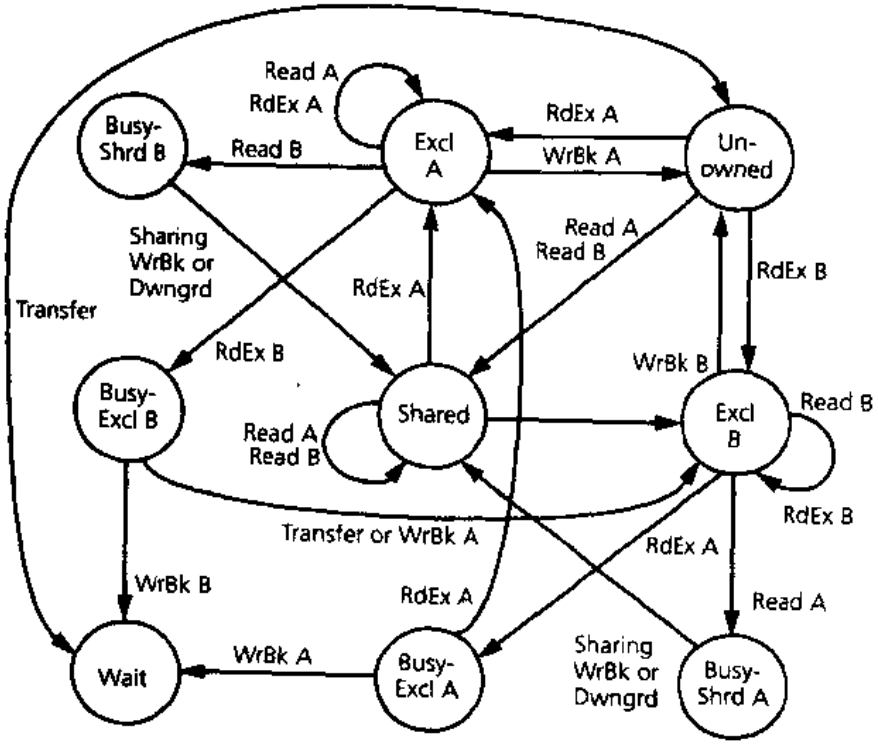
习题

8.1 同我们在第 7 章中讲的机器相比，在高速缓存一致的机器上模拟消息传递，哪些地方效率低？哪些地方效率高？

8.2 1）对于本书用到的并行应用案例分析，你认为哪些在使用多处理器而不是单处理器节点时具有潜在的优势（假设处理器的总数相同）？那些可能有劣势？在什么情况下？

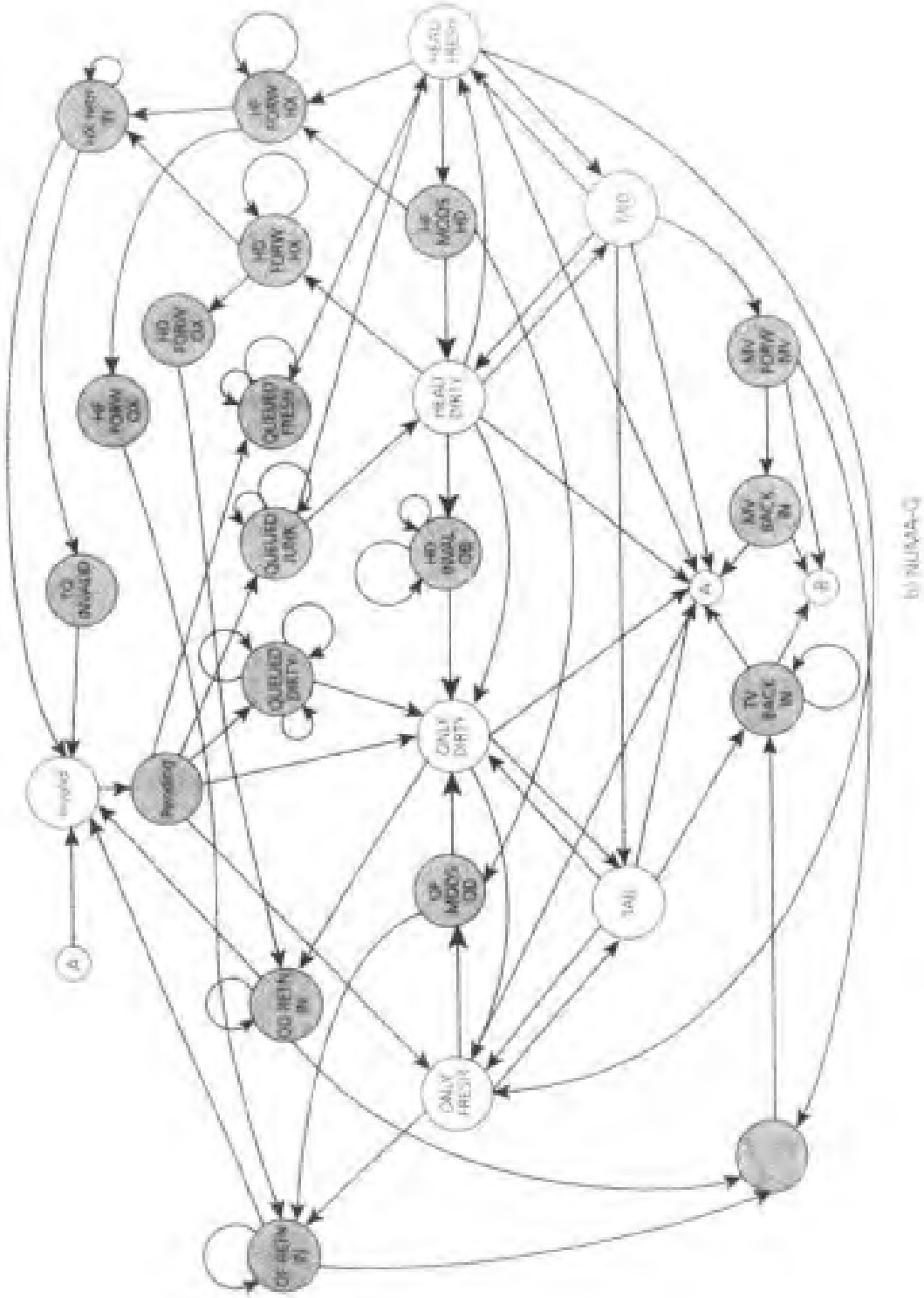
2）当机器的规模扩大时，你对上面问题的回答会发生什么样的变化？也就是说，当机器的规模扩展到几百个处理器时，你认为使用固定大小的多处理器节点的性能好处何在？

3）Illinois MESI 一致性方案对采用多处理器节点的组成结构有什么特殊的好处？



a) Origin2000

图 8-42 本章案例分析中的多处理器的扩展的目录状态图。图 a）是 SGI Origin2000 大大简化的状态图：它只显示了目录的忙状态，但是忽略了 I/O 操作、毒化状态和几种竞争情况。为了说明忙状态的使用，显示了来自两个节点 A 和 B 的访问。例如标记了“Excl A”的状态表示目录认为块在 A 节点处于排他状态，标记了“RdEx B”的弧线表示来自节点 B 的排他读操作。传输操作和等待状态用于处理回写，正如文字所描述的那样。b）图中的 Sequent 的 NUMA-Q 的状态图要完整得多，虽然它也排除了几种边角情况。图中的弧线没有标注，几个状态标记也没有解释；该图的目的不是为了给出完整的协议，而只是为了说明在真实的系统中，完整的状态图可能是相当复杂的



b) MUSSA-2

图 8-42(续)

8.3　给定一个 512 个处理器的系统，每个目录可见的节点有 8 个处理器和 1 GB 的主存储器，高速缓存块的大小为 64 字节，计算下列两种方案的目录存储器开销：

1）全位向量方案；

2）Dir_iB 方案，$i=3$。

8.4　本章提供了一个图，说明了在像 SGI 的 Origin 那样的扁平的、基于存储器的协议中，读操作的严格的请求-响应、干涉转发、应答转发等网络事务（见图 8-12），试对写操作做同样的工作。

8.5　Origin 的协议假设对作废的确认在请求者处收集。另外一种方法是把所有的确认都发回宿主节点（作废请求来自那里），并由宿主节点向请求者发回单个确认。Standford 的 FLASH 多处理器使用这种解决方案。这两种选择之间主要的性能和复杂性的折中是什么？

8.6　请对 Origin 协议的非高速缓存的共享读、非高速缓存的独占读、写作废请求画出网络事务状态图（如图 8-16 所示），并各给出一个使用例子。

8.7　可以不使用 SCI 协议中的双向链表，而使用一个单向链表。这样做有什么优点？请叙述一下如果使用单向链表，对下列操作需要作什么修改：

1）替换共享链表中的一个高速缓冲块。

2）对共享链表中的一个高速缓冲块写入。

定性地讨论其对大规模多处理器性能的影响。

8.8　怎样可以减少 SCI 的协议中引起作废的写的时延？画出其网络事务，主要的折中是什么？

8.9　一个变量呈现迁移共享的时候，读该变量的处理器就是下一个要对它写入的处理器。你能用什么协议优化在这种情况下降低流量和时延？如何动态地发现这种情况？请详细说明一种或两种方案。

670 ~ 672

8.10　另一种可以被动态检测的模式是生产者-消费者模式，在这种模式中一个处理器反复地对一个变量写入（生产），另一个处理器反复地读（消费）这个变量。基于作废的标准 MESI 协议能很好地适应这种情况吗？为什么？你认为什么样的改进或协议会更好一些？在时延和流量方面有何减少？你如何动态地发现和使用这种改进？

8.11　在基于目录的系统中，为什么基于更新的协议比基于作废的协议更难提供写原子性？你如何解决这个问题？在基于总线的系统中存在同样的困难吗？

8.12　考虑如下的程序段在一台高速缓存一致的多处理器上运行，假定所有的变量初值均为 0。

P_1	P_2	P_3	P_4
A = 1	u = A	w = A	A = 2
	v = A	x = A	

只有 A 是共享变量。假定写入者知道高速缓存副本在哪里，并直接向它们发出更新而无需咨询目录节点。假设基于更新的协议，请构造一种违反写原子性的情况。

1）请说明结果中发生的对顺序同一性的破坏。

2）你能否生成一种一致性也遭到破坏的情况？你怎么解决这些问题呢？

3）你能在基于作废的协议中构造同样的问题吗？

4）你能在基于更新的总线型系统中构造同样的问题吗？

8.13　在 Origin 协议中处理回写时，我们提到，如果进行回写的节点接收到一个干涉时，干涉被忽略。给定一个不保证点对点通信次序的网络，在什么情况下决定忽略干涉必须要小心？

我们如何检测该干涉必须被丢弃？如果网络保证点对点通信的次序，还有这个问题吗？

8.14 在 8.5.2 节中讨论的 Origin 的串行化问题在严格的请求-响应协议中还会出现吗？我们提到的指导方针还管用吗？请用实例说明，包括在 8.5.2 节中讨论的例子。

8.15 考虑 NUMA-Q 中写的串行化问题，假定采用两级的层次式一致性协议。如果节点的远程高速缓存中一个块为脏，来自其他节点的写到达该节点时，如何与来自该节点的处理器写实现串行化？为了保证串行化必须产生什么事务？

673

8.16 在 Origin 的实现中，到达存储器/目录接口的请求消息的优先级比到达的响应要高，除非响应有挨饿的可能才不这样做。你认为为什么要给请求更高的优先级？请描述一些检测何时反转优先级的方法。响应挨饿的危险何在？

8.17 1）当进行页面迁移或复制时候为什么需要清空 TLB？

2）对于有软件重载的 TLB 的 CC-NUMA 多处理器，假定有一页需要迁移，那么你会采用下列哪种 TLB 清空方案？为什么？

i）目前只有页项的 TLB。

ii）只有自上次清空后又重新装载过页项的 TLB。

iii）系统中所有 TLB 。

[提示：选择应该基于以下两个标准：进行 TLB 实际清空的代价和为了实现方案跟踪所需信息的难度。]

8.18 对于一个简单的两处理器的 CC-NUMA 系统，来自两个处理器 P_0 和 P_1 的对三个虚页 X、Y、Z 的高速缓存扑空的轨迹显示如下。时间的推移从左到右。“R”是读扑空，“W”是写扑空。P_0 和 P_1 的本地存储器分别是 M_0 和 M_1。本地扑空的代价为一个时间单元，远程扑空的代价为 4 个时间单元。假定读扑空和写扑空的代价相同。

1）假设知道全部轨迹，你在哪个本地存储器里放置页面 X、Y 和 Z？

2）假设三个页面初始都在 M_0。你预先知道全部轨迹。在轨迹开始处，你能对每个页面做一次迁移或一次复制，或什么也不做，这些操作代价为零。你将对各个页面采取什么动作？

页 X:
P_0: RRRR R R RRRRR RRR
P_1: R R R R R RRRR RR
页 Y:
P_0: no accesses
P_1: RR WW RRRR RWRWRW WWWR
页 Z:
P_0: R W RW R R RRWRWRWRW
P_1: WR RW RW W W R

3）如果页面迁移或复制的代价为 10 个时间单元，回答问题 2)。此外，给出每个页面最终的存储器访问代价。

4）如果迁移或复制代价为 60 个单元，回答问题 3)。

5）如果所显示的高速缓存扑空轨迹重复 10 次，回答问题 4)。（你在整个轨迹的开始

仍然只能做一次迁移或复制。）

674

8.19 第 5 章 5.5 节引入的满－空位对细粒度的同步提供了硬件支持，已经被建议在 CC-NUMA 机中使用。满－空位的优点和缺点是什么？为什么现代的系统不采用这种技术呢？

8.20 对于基于作废的协议，锁传输所用的网络事务比必要的要多。另一种办法是不对锁做高速缓存，因此锁变量留在主存储器中，总是在存储器中对它进行访问。

1）试写出采用非高速缓存操作的简单锁和标签锁的伪代码。

2）与高速缓存锁相比，它有什么优缺点？在实际系统中你会部署哪一个？

3）你能描述一种使用高速缓存和非高速缓存两种读写操作的方案，来改善锁的性能吗？你的方案需要什么特殊的操作？

8.21 因为高竞争和低竞争两种情况最好由不同的加锁算法解决，因此有人提议，建立一个同步算法库，在运行时根据对同步变量的访问模式，用硬件支持不同算法间的切换。

1）在你的库中，你会提供什么锁？

2）假定一个基于存储器的目录协议，请设计在运行时在锁之间切换的简单硬件支持和策略。

3）举出一个使用这样的支持特别有效的例子。

4）其潜在的缺点是什么？

8.22 我们在研究体系结构的时候使用了 4 种实际的应用：Ocean、分块的 LU 因子分解、FFT（执行由矩阵转置分隔的行局部计算）和 Barnes-Hut。针对每一种应用回答下面的问题（假设 CC-NUMA 系统）：

1）为了保证和扩展的存储器层次结构之间的良好相互作用，需要对数据结构和数据分布做什么样的修改和增强？

2）和高速缓存大小、分配粒度、一致性和通信之间有什么相互作用？

8.23 考虑在高性能 FFT 这样的计算中使用的并行数据转置的例子。图 8-43 给出了转置的图示。每个进程向每个其他的处理器转置分配给它的行的碎片，包括对它自己的部分。在转置前，进程对分配给它的转置的源矩阵进行读和写；转置之后，读和写分配给它的目的矩阵中的行。分配给一个进程的同时在源和目的矩阵两者中的行在进程的本地存储器中分配空间。有两种转置的方法：进程可以从源矩阵中它自己的行读本地元素，把它们写到目的矩阵的适当的元素中去，不管它们是本地的还是远程的，如图 8-43 所示（叫做发送者启动的转置）；另一种方法是，进程能对目的矩阵的本地行写入，从源矩阵读出适当的元素，不管它们是本地的还是远程的（叫做接收者启动的转置）。

675

1）给定基于作废的目录协议，你认为哪个方法更好？为什么？

2）如果假定基于更新的协议，你认为对 1）的答案如何变化？

3）考虑以下矩阵转置的实现，你打算在一台 8 处理器的机器上运行。每个处理器有一个 8 KB 全相联的一级高速缓存，每行 128 字节。（注意，AT 和 A 不是同一个矩阵。）

```
Transpose(double **A, double **AT)
{
    int i,j,mynum;
    GETPID(mynum);
    for (i=mynum*nrows/p; i<((mynum+1)*(nrows/p)); i++){
```

```
            for (j=0; j<1024; j++) {
                AT[i][j] = A[j][i];
        }
    }
}
```

676

输入数据集是 1024×1024 双精度浮点数的矩阵（即，nrows 为 1024），将它分区，使每个处理器负责生成被转置的矩阵 AT 的连续行的块（即接收者启动的转置）。忽略有所有处理器先访问处理器 0 所引起的竞争，该代码的主要性能问题是什么？你用什么技术去解决它？重新安排代码来尽可能缓解所有的性能问题。写出重新安排的完整循环。

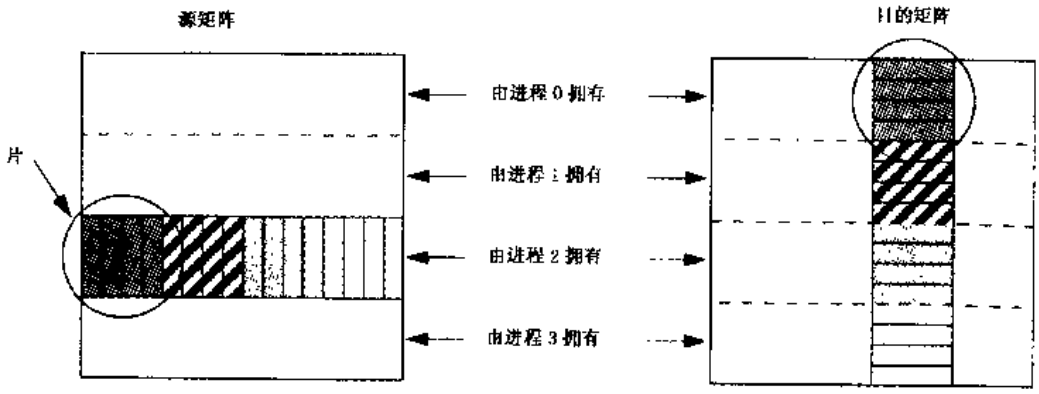


图 8-43 发送者启动的矩阵转置。源和目的矩阵在进程之间划分成连续行构成的组。每个进程把它的 n/p 行分成 p 个片，片尺寸为 $(n/p) \times (n/p)$。以进程 2 为例：分配给它的一个片成为分配给每一个其他的进程的行的集合，在这个情况下，它在本地转置一个片（从左边起第三个）

8.24 考虑一个基于总线的层次式系统在根部有集中式的存储器，而不是本章讨论的分布存储器。满足读和写之间主要的区别是什么？简要的描述一下读和写的路径。

8.25 不追求一个节点中远程访问高速缓存和 L_1 高速缓存之间的包含特性，你能否构造一个带集中式存储器的基于总线的层次式系统？如果可以，会带来什么样的复杂性？

8.26 为了保证两级层次式总线设计的顺序同一性，是否可以在作废请求到达 B 总线时返回一个确认？如果可以，对高速缓存的设计和实现以及事务间保持的次序有什么强制性的约束？如果不可以，请说出理由？如果层次超过两级时可以吗？

8.27 假定在一个基于总线的层次式系统中，两个不同节点上的两个处理器同时对一个块发出了升级的请求。试在系统中跟踪它们的轨迹，讨论所有的状态变化以及它们何时必须发生，防止死锁和防止两个处理器都获得所有权的措施是什么？

8.28 在具有分布存储器的基于总线的层次式系统中的一种优化是高速缓存到高速缓存的共享：如果本地总线上的另外一个处理器的高速缓存可以提供需求的数据，就不必访问全局总线和远程节点了。在基于环的层次式系统中支持这样的优化有什么折中？

8.29 在具有层次式目录的机器中，你会选择什么分支因子？说明主要的折中。你能采用什么技术来缓解性能权衡？尽可能具体地描述。

8.30 不维持目录高速缓存的包含性能实现层次式目录吗？设计一个能做到这点的协议，并讨论其优缺点。

677

第 9 章　硬件/软件功能的折中

本章论述第 8 章所讨论的基于目录的高速缓存一致性系统的潜在局限性以及为克服这些局限所提出的硬件/软件的折中措施。这些系统的主要局限性在于：

- *存储器操作的等待时间长。*顺序同一性（SC）是面向程序员的存储同一性模型，到目前为止，总线侦听式和基于目录两种系统均假定采用此模型。为了满足 SC 的充分条件，处理器必须等待前面的存储操作完成，才能产生下一个存储操作。与基于总线的系统的性能影响相比，这一限制对于可扩展系统的性能影响更大，因为后者的通信延迟更长，而且在其关键路径上会存在更多的网络事务处理。更糟糕的是，这更是对编译器的限制，如果程序员假定顺序同一性的话，编译器根本无法改变对于共享数据的存储操作的次序。
- *用于复制的容量有限。*当数据交换时，它自动地在处理器的高速缓存中复制，而不是在本地主存中复制。当工作集大且包含非局部数据或者当冲突式扑空很多时，这可能导致因容量不足造成的扑空和附加的通信。
- *设计和实现的高成本。*通信辅助部件包含支持高速缓存一致性的特殊硬件并紧密地集成在处理节点之中。协议是复杂的，其正确的硬件实现会花费相当长的设计时间。（所谓成本，在这里我们是指硬件和系统设计时间的费用。但是，回顾第 3 章所述，优良性能的实现同时与编程的成本相关，降低系统成本的措施往往导致编程成本的急剧上升。）

本章将集中讨论这三方面的局限性。尽管人们对解决这些问题的方法还有不同程度的争论，但这些方法的某些部分已经被商品化的并行机的设计者所采纳。此外，我们还常常碰到其他一些局限性（包括第 7 章所讨论的 CRAY T3D 的共享物理地址空间的寻址限制）以及以硬接线的方式在机器中实现单一协议这一事实。然而，这类问题的解决方案经常与那些主要问题的解决方案相结合，它们将被作为高级的专题而讨论。

存储器操作等待时间过长的问题可以用两种硬件的方法解决。第一种方法，实现方案的设计可以不必满足 SC 的充分条件，而仅需满足 SC 模型本身，这一点现代的非阻塞型处理器已经这样做了。即处理器无须等待前一存储器操作结束就可以开始下一操作；但是系统保证这些操作的结束或对外界的效果显现仍维持原有的次序。第 8 章所讨论的 SGI Origin2000 系统使用这种方法。第二种方法，存储器同一性模型本身可以被放松，程序的次序无须严格地保持。放松的同一性模型改变了共享地址空间的语义，对硬件和软件两者均有影响。它要求程序员更加小心才能编写正确的程序，但它允许硬件在更大程度上重叠和改变存储器操作的次序。重要的是，它还允许在将存储器操作提交硬件之前由编译器对它们重新排序，这正是优化编译器通常所做的。9.1 节将讨论放松的存储同一性模型。

复制容量有限的问题可以通过自动地在主存中缓存数据（不是仅仅在处理器的高速缓存中缓存数据），并保持缓存数据的一致来解决。与硬件的高速缓存不同，主存中的复制和一致性保持可以采用各种不同大小的粒度，例如，高速缓存块、存储页面或用户定义的对象并

可以直接用硬件或通过软件来管理。这就提供了非常丰富的协议、硬/软件实现和性能成本折中的空间。一种致力于改善性能的方法是将本地主存当作硬件高速缓存那样管理，以高速缓存块的粒度提供复制和一致性。这种方案被称作惟高速缓存式存储器体系结构，或称 COMA，将在 9.2 节中对其进行讨论。它使软件无须再为容量型扑空以及数据在主存中的初始分布而操心，同时仍然以细粒度提供一致性，从而避免伪共享。然而，它要求在主存中为每个块维护标记和状态，因而需要更多的硬件。

最后，我们来看看解决硬件成本问题的多种途径。方法之一是使通信辅助机构及网络部件与处理节点的关系不要那么紧密，其代价是提高了通信延迟和辅助机构的占用率。另一途径是以软件方式而不是以硬件方式提供自动的复制和一致性，产生图 9-1 中所说明的一系列可能的系统实现方案。这种软件实现的方法在主存内提供复制和一致性并能以各种不同的粒度操作。该类方法允许以市场现成部件实现节点和互连，降低硬件成本，把追求优良性能的更多（现在也是更大）的责任留给程序员。降低硬件成本的途径将在 9.3 节中讨论。

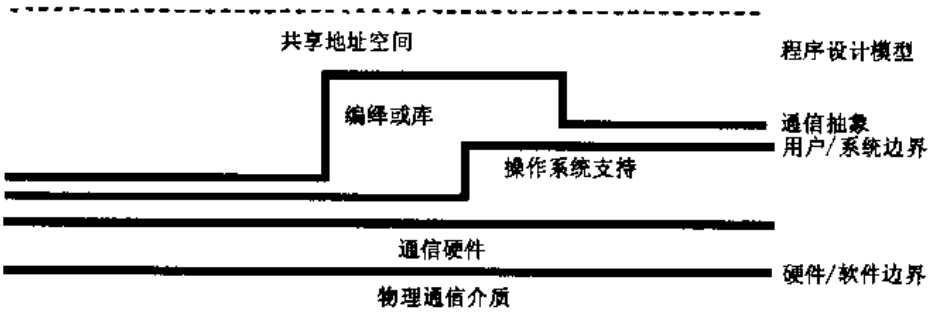


图 9-1　本章所讨论的系统的通信体系结构的层次。本图表示了用于支持一致的共享地址空间的软件介入的程度

以上这三个问题是相互联系的。比如，成本与在主存中管理复制和一致性的方式密切相关：以何种粒度管理？是直接以硬件方式还是通过运行时系统或操作系统管理？成本和粒度也与存储同一性模型相关：对于低成本低性能解决方案和较大的粒度来说，放松的存储同一性模型好处更大，而以软件实现协议更有利于充分发掘语义放松的作用。理解不同方案的空间的一个有用的框架是基于数据在本地复制存储、一致性保持及通信的粒度。9.4 节建立了这样一个总结及关联不同方案的框架。该框架自然地产生了一种在高成本的 COMA 和低成本的全软件途径之间尽力寻求好的折中的方法。9.4 节讨论了这种被称作简单 COMA 的方法。

680

9.5 节深入研究了本章所讨论的系统对并行软件的影响。最后，9.6 节覆盖了某些更先进的专题，包括解决共享物理地址空间和固定的一致性协议所存在的潜在局限性的技术。

9.1　放松的存储同一性模型

让我们回忆第 5 章，共享地址空间的存储同一性模型规定了对于相同或不同的存储单元所能够执行的存储器操作的相互次序的约束，它使程序员能推论出他们的程序的行为和正确性。事实上，支持一个共享地址空间命名模型的任何系统层次都具有一个存储同一性模型，包括程序设计模型或程序员接口、用户/系统接口和硬件/软件接口。与某一层次交互的软件必须了解该层次的存储同一性模型。我们的注意力主要集中于程序员所见的同一性模型，即

位于程序员和由编译器、操作系统及硬件组成的系统其余部分之间的接口，因为这正是程序员[⊖]推论的依据。例如，处理器可能保持提交给它的所有程序的存储器操作的次序，但是，如果编译器已经对操作重新排了序，那么程序员就不能再用硬件对外提供的简单模型来推论。 681

位于程序员接口的同一性模型对于程序设计语言、编译器和硬件也有其意义。对于编译器和硬件而言，它规定了对改变来自进程的访问次序的约束以及不能违反的次序，因而说明了所能使用的性能优化手段。我们将看到，程序设计语言必须提供在必要时引入这种约束的机制。一般而言，允许系统对来自进程的存储器访问所做的重排序越少，我们为程序员提供的程序设计模型越直观，但对性能优化的约束也越大。存储同一性模型的目标是施加的次序约束能使编程复杂性和性能之间达到良好的平衡。模型必须是可移植的，即规范必须能在多种平台上实现，从而同一程序能在所有这些平台上运行并保持相同的语义

到目前为止我们所假定的顺序同一性模型为程序员提供了直观的语义，即各进程内部的程序原序和进程之间重叠的同一性，通过满足其充分条件可以很容易地实现该模型。但是，正因为保持了访问的严格次序，该模型的缺点在于限制了当今单处理器的编译器和多处理器所常用的性能优化手段的使用。由于存储器访问的高代价，如果对来自一个处理器的多个存储器或通信操作的服务进行重排序或重叠，计算机系统能实现较高的性能。保持 SC 的充分条件显然不允许硬件做多少重排序和重叠，能够保持 SC 但不保持充分条件的方法也具有局限性。若在程序员接口保持 SC，即便是针对不同单元的存储器访问，编译器也不能改变其次序，这样就不允许诸如代码移动、公共子表达式消除、软件流水，甚至像例 9.1 所述的寄存器分配这样的关键性的性能优化措施。

例 9.1　说明即使硬件满足 SC，寄存器分配也会导致 SC 的破坏。

解答：考虑图 9-2a 所示的代码段。在寄存器分配之后，由编译器生成并被硬件所见的代码如图 9-2b 所示。在 a）中的 SC 硬件之下，是不允许产生（u，v）＝（0，0）这样的结果，但在 b）的 SC 硬件条件下，这不仅允许，而且会产生。事实上，寄存器分配改变了 P_1 中的 A 的写入和 B 的读出的次序，也改变了 P_2 中的 B 的写入和 A 的读出的次序。一个单处理器的编译器可以很容易地对各个进程执行这些优化：它们对于串行程序这些优化是合法的，因为被改变次序的访问是针对不同位置的。■ 682

P_1	P_2
B = 0	A = 0
A = 1	B = 1
u = B	v = A

a) 在寄存器分配之前

P_1	P_2
r1 = 0	r2 = 0
A = 1	B = 1
u = r1	v = r2
B = r1	A = r2

b) 在寄存器分配之后

图 9-2　说明编译器所做的寄存器分配能破坏 SC 的例子，a）中的代码是程序员赖以推论的原始代码。r1 和 r2 是寄存器，b）中是编译器执行寄存器分配后可能形成的代码

⊖ "程序员"（programmer）一词在这里是指负责生成并行程序的实体。例如，如果一个编程人员（human programmer）写的串行程序由系统软件自动并行化，则系统软件必须与存储一致性打交道；而程序员只是简单地假定单处理器上的串行语义。

在程序员接口提供SC意味着在包括硬件/软件接口的低层接口支持SC。如果要满足SC的充分条件，处理器在发出下一次访问前要等待前次访问的结束，这样存储器访问所经历的大部分时延直接被处理器当作阻塞时间。虽然处理器在一个未完成的存储器访问被服务时能继续执行非存储器指令，但来自这种重叠的收益甚微，因为即使没有指令级并行，平均每三条指令就有一条是存储器访问（Hennessy and Patterson 1996）。我们需要针对这个性能问题采取措施。

我们所能采取的一个措施是既在程序员接口保持顺序同一性，又以某种方式对处理器隐藏长的时延阻塞。这一点能用几种方法做到，它们大致分为两类（Gharachorloo，Gupta，and Hennessy 1991）。第11章将进一步讨论这些技术及它们的性能含义，在这里，仅简要对它们做一些直观的介绍。在第一类方法中，系统仍然保持SC的充分条件，编译器并不改变存储器操作的次序。使用诸如数据预取和多线程这样的时延包容技术使数据传输相互重叠或与计算重叠，对处理器隐藏大部分的时延，但是在前面的读写操作尚未按程序原序完成之前不产生真正的读写操作。

在第二类方法中，系统在程序员接口处保持SC，但不保持SC的充分条件。只要保证在结果中的顺序一致性不被违反，编译器就可以改变操作的排序。人们已经开发了用于此目的的编译器算法（Shasha and Snir 1988；Krishnamurthy and Yelick 1994，1995），但这些算法代价太高，而且它们的分析能力目前还是相当有限的。在硬件层次中，存储器操作可以不按程序规定的次序产生并执行，但是必须保证在其他处理器看来它们仍然是遵循程序的次序。这种方法特别适合于具有动态调度的处理器，例如，SGI的Origin 2000中的R10000处理器，这类处理器能使用指令预取缓冲区来发现可以执行的不相关指令。指令以程序原序送入预取缓冲区，然后变序地从指令预取缓冲区中取出并执行，但是保证指令按程序原序从预取缓冲区中撤除。操作的产生和执行甚至可以根据分支预测变序地越过预取缓冲区中尚未确定的分支点，这叫做推测执行。但是因为分支终究会被确定，并应在推测执行的指令之前被撤除，因此在分支确定之前，推测执行的指令的效果不应在寄存器堆或外部存储器系统中反映。如果分支预测错误，那些预执行的效果将永不可见。一种叫做推测读的技术走得更远一点。这里，读所返回的值甚至在其正确性被确定之前就已被使用了，如果后续的检查确认它们是不正确的，计算就要卷回以便重新发出读操作。注意，不能以这种方式推测写入操作，因为写入一旦被其他处理器所见，卷回并恢复原值将是非常困难的。因此，在前面的访问正确完成之前，写入的值不能为其他处理器或外部存储器系统环境所见。

683

不少现代微处理器已经支持所有这些技术中的一部分，例如MIPS R10000、HP PA-8000和Intel Pentium Pro等。然而，虽然这些技术日趋流行，但所要求的大量硬件资源和高复杂性使它们在隐藏多处理器时延方面的成功还不显著（见第11章），而且并不是所有的处理器都支持这些技术。或许最关键的是这些技术对处理器有用，但它们并不能帮助编译器完成对于优化至关重要的存储器操作的重新排序。

另一种克服由SC所造成的性能上的限制的完全不同的方法是改变存储同一性模型本身，即，不对程序员保证强的次序约束，但仍然保持足够直观有用的语义。通过放松次序约束，这些放松的同一性模型允许编译器在将访问操作发给硬件之前，至少在某种程度上对其重新排序。在硬件层次，这些模型不仅允许来自同一进程的多个存储器访问同时处于未完成状态，甚至允许它们变序结束或对外呈现效果，使大部分时延被重叠而不为处理器所见。支持

放松的模型的直觉是：SC 通常太保守，在大多数情况下，它所保持的次序有许多对于满足程序员的直觉是不需要的。我们可以在文献（Adve 1993；Gharachorloo 1995；Adve and Gharachorloo 1996）中了解放松的存储同一性模型的详细情况。

请考虑图 9-3 所示的简单的例子。左边是 SC 实现所必须维持的次序，右边是直觉上正确的程序语义所必须保持的次序。后者要少得多。例如，在这种情况下，可以改变 P_1 对变量 A 和 B 的写操作而不影响程序观察到的结果，我们所必须保证的是对这两个变量的写应在变量 flag 被置为 1 之前完成。同样，一旦观察到 flag 的值变为 1，P_2 对变量 A 和 B 的读的次序也能被改变。[○] 尽管做了这些排序的改变，仍能获得和 SC 执行一样的结果。另一方面，虽然对 flag 的访问也是简单的变量访问，但允许改变 flag 与任何一个进程中 A 和 B 的相对次序的模型会牺牲直觉语义和 SC 结果。如果系统软件或硬件能自动发现对保持 SC 语义起关键作用的程序原序，并通过改变其他的次序获得更高的性能那真是太好了（Shasha and Snir 1998）。但对于一般的程序而言，这个问题是难以处理的（事实上是无法确定的），而非精确的解决方法通常又太保守而没多大用处。

684

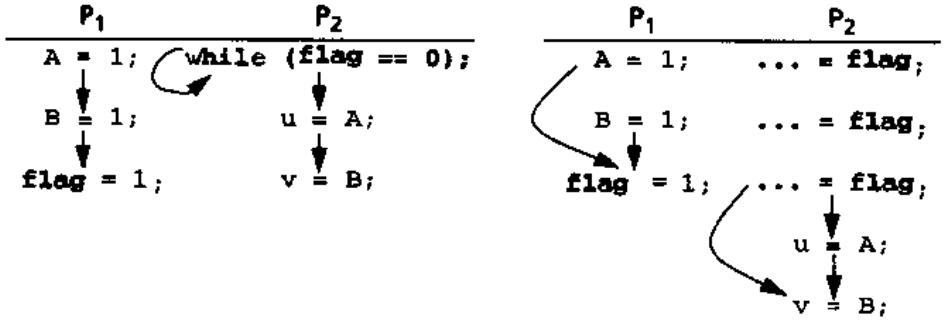


图 9-3 放松的存储同一性模型背后的直觉知识。图中的箭头指示保持的次序。图中 a）显示了顺序同一性模型所保持的次序。图中 b）显示了“正确”或“直觉”语义所需要的次序。黑体字表示对于变量 flag 的访问对于次序是重要的，并且事实上被用来协调事件同步

一个放松同一性模型的完整解决方案包括三个部分：

1）*系统规范*。这是两件事情的明确规范：第一，从可观察的意义上，系统保证能维持的存储器操作间的程序原序，包括是否维持写入的原子性；第二，如果缺省是保证维持所有的程序原序，那么系统对程序员提供何种机制以便在需要时显式地强迫这种次序。现在应该很清楚，编译器和硬件都有其自己的系统规范，但是我们关心的是两者一起或系统整体对程序员提供的规范。对于处理器系统结构而言，它所输出的规范控制了它所允许的变序以及它所提供的保持次序的原语，通常称其为*处理的存储模型*。

685

2）*程序员接口*。系统规范本身是一个同一性模型。程序员可以利用它推断程序的正确性，插入适当的保持次序的机制。但是，对程序员来说，这是非常低层次的接口，不考虑重

○ 实际上，有可能进一步弱化正确操作所要求的条件。例如，并不需要在对 flag 的写入完成之前结束对 A 和 B 的写入，仅需要在处理器 P_2 看到 flag 的值变为 1 之前完成这些操作。事实上，这样的放松模型难以用硬件实现。因此，这种更为放松的模型仅对软件实现有意义，我们将在 9.2 节中对其进行讨论。

新排序和写入原子性，并行程序设计就已经有足够的挑战性！各系统规范所支持的重新排序和次序强制的机制是不同的，因此损害了可移植性。所以，程序员希望的是一种编写“安全”程序的方法。这是一种契约，不管所支持的系统规范允许什么样的缺省次序；如果程序员遵循某种高层次的规则或提供足够的程序标注，例如告诉系统图 9-3 中的 flag 实际上用作同步变量，那么程序运行其上的任何系统将总是能保证顺序同一的执行。程序员的责任在于遵循规则并提供标注，幸运的是，这不牵涉到潜在重排序层次的推论。系统的责任是用规则和标注作为约束，保持顺序同一性的形象。对程序设计语言来说，这意味着它们必须支持所需的标注，提供直观的编程接口。

3）变换机制。它把程序员的标注变换成系统规范提供的接口（特别是次序保持机制），从而使系统能完成其任务。

在下面关于放松的同一性模型的讨论中，首先考察系统，特别是微处理器所输出的不同的低层次的规范。9.1.2 节讨论程序员接口或契约以及程序员如何提供所需的标注。9.1.3 节简要讨论变换机制。9.1.4 节讨论存储同一性模型方面当前的实践。实现复杂性和性能受益的细节将推迟到第 11 章关于时延包容机制时讨论。

9.1.1 系统规范说明

微处理器厂商和研究人员已经建议了几个不同的重新排序的规范，每一个都有其自己的强制次序的机制。这些规范包括*全序存储*（TSO）（Sindhu，Frailong，and Cekleov 1991；Sun Microsystems 1991）、*偏序存储*（PSO）（Sindhu，Frailong，and Cekleov 1991；Sun Microsystems 1991）、来自 Sun Sparc V8 和 V9 的*放松的存储器次序*（RMO）规范（Weaver and Germond 1994）；在文献（Goodman 1989；Gharachorloo 1990）中描述并用于 Intel 的奔腾处理器的*处理器同一性*（PC）；*弱序*（WO）（Dubois，Scheurich，and Briggs 1986；Dubois and Scheurich 1990）；*释放同一性*（RC）（Gharachorloo 1990）；Digital Alpha（Sites 1992）和 IBM/Motorola PowerPC（May et al. 1994）模型。当然，一种处理器的某个特定实现不一定支持其系统规范所允许的所有的重排序。系统规范定义了该体系结构的语义接口，即程序员必须假设哪些重排序可能发生。而实现决定了何种重排序实际发生，可以获得多少性能上的改善。

686

让我们用模型所允许的程序原序的放松作为模型分类的主要因素来讨论某些规范或同一性模型（Gharachorloo 1995）。第一组模型，包括 TSO 和 PC，仅允许读操作按程序原序旁路较早的未完成的写操作，或在其之前结束（即，允许写→读次序被改变）。下一组，包括 PSO，还允许写操作旁路前面的写操作（即写→写重排序）。最后一组，包括 WO、RC、RMO、Alpha 和 PowerPC，更允许读操作或写操作旁路前面的读操作（即允许读和写访问的所有类型的重排序）。一个读-修改-写（read-modify-write）操作被当作读和写两者处理，只有当读和写两者均能相对于其他操作改变次序时，才能改变读-修改-写操作与其他操作的相对次序。在任何情况下，我们假定基本的高速缓存一致性，即写传播和写串行化，单处理器数据和控制依赖性在各个进程中维护。在大多数情况下，所讨论的规范是受处理器的体系结构本身即硬件接口的启发，并为此而定义的。所有这些也适用于编译器，但是，由于复杂的编译器优化需要改变所有类型访问的次序的能力，大多数编译器并不支持如此多的次序模型。事实上，在程序员接口层次，除了最后一组模型外，所有其他组模型用途有限，因为它们无法允许许多重要的编译器优化。

1. 放松写对读的程序原序

这一类模型的主要动机在于允许硬件隐藏写操作的延迟。当写扑空仍然在写缓冲器中而未被其他处理器看见时，处理器能够发出和结束在其高速缓存中命中的读或甚至未在其高速缓存中命中的单个读。我们将在 11 章中看到，隐藏写延迟有重要价值，大多数处理器均利用这一放松模型的优点。

这类模型（如 TSO 和 PC）在多数情况下即使不用任何特殊的操作，也能很好地保持程序员的直觉。例如，事件同步的惯用做法，即在信号变量上的循环测试等待，无须修改就能工作（如图 9-4a 所示）。这是因为 TSO 和 PC 模型保持了写的次序，因此，在系统中按程序原序排在前面的所有写操作结束之前，对信号变量的写入不会被看到。因此，大多数早期的多处理器支持这两种模型之一，包括 Sequent Balance、Encore Multimax、Vax-8800、SparcCenter 1000/2000、SGI 4D/240、SGI Challenge 和 Pentium Pro，操作系统这样的复杂程序在这些机器上的移植相对容易。 687

```
P1            P2
A = 1;        while (Flag==0);
Flag = 1;     print A;
```

a)

```
P1            P2
A = 1;        print B;
B = 1;        print A;
```

b)

```
P1            P2              P3
A = 1;        while (A==0);   while (B==0);
              B = 1;          print A;
```

c)

```
P1                P2
A = 1; (i)        B = 1; (iii)
print B; (ii)     print A; (iv)
```

d)

图 9-4　比较 TSO、PC 和 SC 的代码序列实例。对于代码段 a）和 b），TSO 和 PC 产生相同的结果；对于代码段 c），PC 违反了 SC 语义（TSO 仍提供 SC 语义）；对于代码段 d），TSO 和 PC 均违反了 SC 语义

当然，这些模型的语义不是 SC，所以在某些情况下表现出区别。图 9-4 给出了 4 个代码段的例子，其中的 3 个我们在前面已看到过。假定这些例子中所有变量的初值为 0。代码段 a）是对信号变量 flag 的循环测试等待的例子。在代码段 b）中，SC 保证如果 B 的打印值为 1，则 A 的打印值也一定为 1，因为 P_1 对 A 和 B 的写入次序不能改变。同理，TSO 和 PC 对于这个代码段也有相同的语义。对于代码段 c），只有 TSO 提供 SC 语义，防止 A 的打印值成为 0，但 PC 模型不行。原因是 PC 并不保证写入的原子性。最后，对于代码段 d），在 SC 条件下操作的任何重叠不可能产生 A 和 B 的打印值均为 0 的情况。其原因如下：程序原序意味着在重叠全序条件下的优先关系是（ⅰ）→（ⅱ）和（ⅲ）→（ⅳ）。如果我们能观察到 B = 0，它意味着（ⅱ）→（ⅲ），它进而意味着（ⅰ）→（ⅳ）。但是（ⅰ）→（ⅳ）意味着 A 将被打印为 1。类似地，A = 0 的结果意味着 B = 1。一个被称为 Dekker 算法（流行的全软件的互斥算法）能实现无硬件支持条件下的原子性的读-修改-写操作（Tanenbaum 和 Woodhull 1997），它依赖于本例中 A 和 B 不会被读出为 0 的性质。SC 提供这个性质，可以被看作是直觉的同一性模型。而 TSO 和 PC 都不能保证这一点，因为它们都允许对应于打印（print）的读操作在前面的写的效果显现之前结束。

为了能在需要时保证 SC 语义（例如，需要将一个根据 SC 条件编写的程序移植到 TSO 和 PC 系统），我们需要强制两种额外次序的机制：1）保证读操作的结束不会早于按程序原

序在它前面的写（用于 TSO 和 PC 两者）；2）保证写操作相对于读操作的原子性（仅用于 PC）。对于前者，不同的处理器体系结构提供不同的解决方案。例如 Sun Sparc V9 规范（Weaver and Germond 1994）提供了不同风格的存储器栅障（MEMBAR）或隔栅（fence）指令来保证任何所希望的次序。我们可以在读之前插入一条具有 write-to-read 次序风格的 MEMBAR。这条 MEMBAR 作用是迫使按程序原序排在它后面的任何读指令必须等待 MEMBAR 之前的所有写指令都完成后才能发出。在那些不提供存储器栅障指令的体系结构中，可以通过用原子性的 read-modify-write 操作序列代替原始的读指令实现同样的效果。一条 read-modify-write 指令被当作读和写两种操作处理，所以在本模型中不能改变它们相对于先前的写的次序。当然，read-modify-write 指令所写入的值必须与读出的值相同以保证正确性。在支持 PC 模型的机器中，用一条 read-modify-write 代替一条读指令也保证了在该读指令处的写入原子性。这一点成立的细节比较费解，感兴趣的读者可以参阅其他文献（Adve et al.1993）。

688

2. 放松写对读、写对写的程序原序

如果要允许一个写操作旁路前面的写（对不同地址的写），那么我们就应允许合并写缓冲器，甚至允许在按程序原序前面的写操作完成之前就撤消后续的写缓冲。所以，它允许多个写扑空完全重叠并且以非程序原序对外显现。其动机是进一步降低写延迟对处理器阻塞时间的影响，并通过使新的数据值尽早为其他处理器所见而改善处理器间通信的效率。Sun Sparc 的 PSO 模型（Sindhu，Frailong，and Cekleov 1991；Sun Microsystems 1991）是惟一的这类模型。与 TSO 类似，它保证写原子性。

不幸的是，改变写的次序会在很大程度上违反我们直觉的 SC 语义。甚至用普通变量作为事件同步的标记（图 9-4a）也不再保证正确，因为对 flag 的写可能在对 A 的写之前就为其他处理器所见。所以，必须能显著改善性能，该模型才有吸引力。

比 TSO 增加的惟一的指令是按进程的程序强制写对写（write-to-write）次序的指令，在 Sun Sparc V9 中，这使用具有 write-to-write 风格的 MEMBAR 指令完成。（较早的 Sparc V8 规范提供了称作写栅障或 STBAR 的特殊指令来实现该效果。）例如，为了实现直觉语义，我们可以在 A 和 flag 的写入之间插入这样一条指令。

3. 放松所有的程序原序：弱序和释放同一性

这最后一类规范不缺省地保证任何程序原序（当然，除了进程中的数据和控制依赖性之外）。其优点是多个未完成的读请求能同时存在，它们可以被依程序原序后续的写旁路，其本身也可以乱序结束，因而允许我们隐藏读时延。动态调度的处理器的实现使其能旁路对其他存储器的读扑空而继续执行，因而这类模型与它们特别匹配。这类模型也是允许编译器优化所做的许多关键变序和访问消减的惟一模型。正因为这些编译器优化对于节点性能的重要性以及它们对程序员的透明性，这些模型事实上是多处理器的惟一合理的高性能存储器模型（除非编译器对潜在的非一致性的分析技术有显著的进步）。几个著名的这类模型包括弱序（WO）（Dubois，Scheurich，and Briggs 1986；Dubois and Scheurich 1990）、释放同一性（RC）（Gharachorloo 1990）、Digital Alpha（Sites 1992）、Sparc V9 的放松的存储器次序（RMO）（Weaver and Germond 1994）和 IBM PowerPC（May et al. 1994；Corella，Stone，and Barton 1993）。WO 是基本的模型，RC 是斯坦福大学的 DASH 原型机（Lenoski et al. 1993）所支持的 WO 的扩展，最后三种由商品化体系结构所支持。将分别讨论这些模型，了解它们如何解决在所有这些重排序的条件下提供直觉语义的问题，例如，它们如何处理标记同步的实例。

689

弱序　弱序模型（也称为弱同一性模型）背后的动机相当简单。大多数并行程序在需要时使用同步操作来协调对数据的访问。在同步操作之间，它们并不依赖于访问次序的保持。图 9-5 给出两个例子，左边的 a）使用加锁-解锁（lock-unlock）对说明一个更新链表表头的临界区（Adve and Gharachorloo 1996）。右边的 b）使用标记 flag 来控制对于生产者-消费者交互所涉及的变量的访问（例如 A 和 D 由 P_1 产生，被 P_2 消费）。该例的关键在于把对 flag 变量的访问当作同步操作，因为这正是这些变量的作用。如果我们这样考虑，那么在上述两种情况下，只要同步操作的相对次序及它们相对于数据访问的次序不改变的话，在同步操作或访问之间（即 a）中的临界区域和 b）中 while 循环之后的 4 条语句）发生的任何程序重新排序都不会违反直觉语义。基于这一事实，弱序的缺省是放松所有非同步存储器操作的程序原序，并且仅在系统标明的同步操作处保证程序的次序。通过增加同步操作或把某些存储器操作标为同步可以强制更强的次序。在 9.1.2 节我们将讨论如何把适当的操作标记为同步操作。

$P_1, P_2, \ldots, P_n$
. .
Lock(TaskQ)
newTask→next = Head;
if (Head != NULL)
Head→prev = newTask;
Head = newTask;
UnLock(TaskQ)
. . .

a)

P_1	P_2
TOP: while(flag2==0);	TOP: while(flag1==0)
A = 1;	x = A;
u = B;	y = D;
v = C;	B = 3;
D = B * C;	C = D / B;
flag2 = 0;	flag1 = 0;
flag1 = 1;	flag2 = 1;
goto TOP;	goto TOP;

b)

图 9-5　协调普通共享变量访问的同步操作的使用。同步可以通过使用显式的加锁、解锁和栅障操作或点对点事件的标记变量（flag）来实现

图 9-6 的左侧说明了弱序所允许的存储器操作重排序。具有一组 read/write 的各个块代表来自处理器的一段连续的非同步存储器操作。同步操作单独说明。保证 WO 系统的充分条件

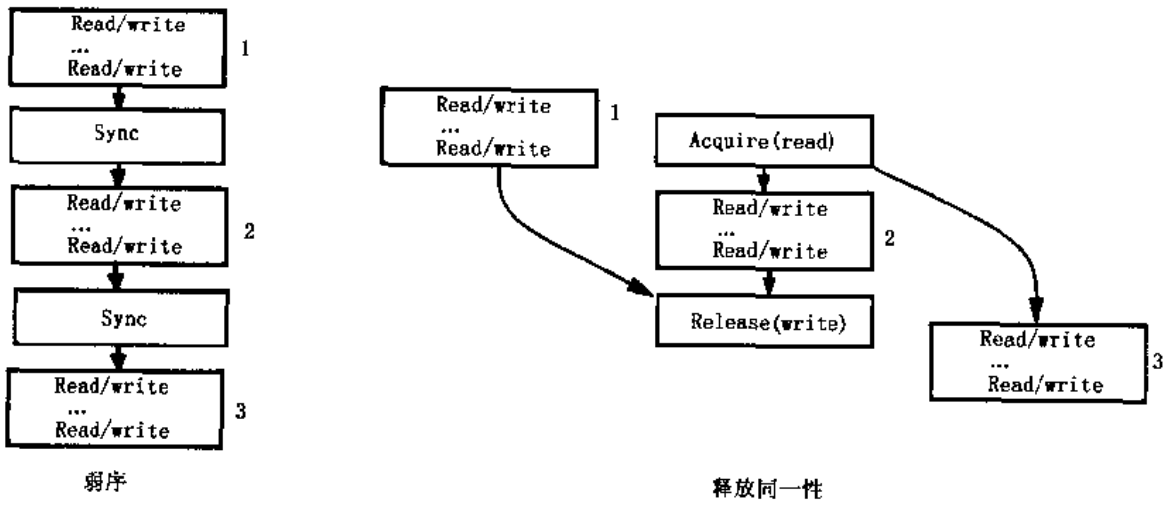


图 9-6　弱序和释放同一性模型的比较。按程序原序，块 1 中的操作先于第一个同步操作，即获取（acquire）操作；块 2 位于两个同步操作之间；块 3 跟随第二个同步操作，即释放（release）操作

如下所述。在发出一个同步操作之前，处理器等待依程序原序所有先导操作（读和写）的结束。同样，跟随同步操作的存储器访问要等待同步操作结束才能发出。在同步操作之间，可以任意改变未标记为同步的读、写、读-修改-写操作的次序。在许多并行程序中，同步操作并不频繁，WO 能为硬件和编译器提供更大的重排序的自由度。

释放同一性 释放同一性认为弱序走得还不够远。它通过区分同步操作的类型并发掘它们的语义扩展弱序模型。特别是，它将同步操作分为获取和释放。获取是一个读操作（它也可以是一个读-修改-写操作），执行获取操作获得一组操作或变量的访问权。获取操作的例子包括图 9-5a 中的 Lock（TaskQ）操作和 b）中 while 语句的条件部分对 flag 变量的访问。释放是一个写操作（或一个读－修改－写操作），它授予另一个处理器对某些操作或变量的访问权，其例子包括图 9-5a 中的 UnLock（TaskQ）操作和 b）中把 flag 变量置为 1 的语句。

690/691

图 9-6 显示，我们可以利用获取和释放操作的分离进一步放松对次序的约束。获取的目的是推迟跟随获取操作之后的存储器访问直至获取结束。它与按程序原序在它前面的访问（块 1 中的访问）毫无关系，所以获取的发出或结束不必等待那些访问的结束，也就是说，获取本身相对于先导的访问的次序可以被改变。与此类似，释放操作的目的是允许访问依程序原序在它前面被修改的数据的值。它与按程序原序跟随它的访问（块 3 中的访问）没有关系，所以那些访问不必推迟到释放操作的结束。但是，在释放为其他处理器可见之前，必须等待块 1 中的访问结束。（因为它们先于释放操作，而且我们无法精确地了解哪个变量与释放相关或被释放所“保护”[⊖]。）同样，我们必须等待获取结束才能执行块 3 中的操作。除了这些约束以外，块 1、块 2、块 3 中的存储器操作可以重叠，改变次序。所以，提供 RC 接口的充分条件是：在标为释放的操作发出之前，处理器必须等待按程序原序所有先导的操作结束；程序原序跟随获取的所有操作在该获取结束前不能发出。当然，这些是充分条件，当我们讨论其他依赖于放松的同一性模型的共享地址空间的方案，寻求好的性能时，将探讨更加大胆的实现。注意，除非存在足够的同步，第 5 章所定义的一致性的写传播短语和写串行化均无法保证，我们将在 9.3.3 节再对其讨论。

Digital Alpha、Sparc V9 RMO 和 IBM PowerPC 的存储器模型 尽管 WO 和 RC 模型的规范是基于强制次序的加标同步操作，但这些操作并不精确对应必须采用的指令。一些商品化的微处理器的存储器模型其缺省并不确保存储器和同步操作的次序，但提供用于强制次序的特殊硬件指令，这些特殊指令是存储器栅障和隔栅。为了用这类微处理器实现 WO 和 RC 模型，WO 和 RC 程序标记为同步的操作（获取或者释放）引导编译器插入适当的特殊指令，或者由程序员直接插入这些指令。

692

例如，Alpha 体系结构（Sites 1992）支持两类隔栅指令：存储器栅障（MB）和存储器写入栅障（WMB）。MB 栅障与 WO 中的同步操作类似，它在发出任何新的访问之前等待所有前面已经发出的存储器访问结束。它与 Sparc 的 MEMBAR 指令的风格不同。WMB 仅仅在写之间强制程序原序（它与 PSO 的 STBAR 类似）。所以，在 WMB 之后发出的读仍然能够旁路一个在 WMB 之前发出的写访问（即在其之前结束），但是，在 WMB 之后发出的写不能这样做。Sparc V9 的放松的存储器次序（RMO）（Weaver and Germond 1994）提供一种 MEMBAR 的

⊖ 可以利用释放允许对先导的获取所控制的操作之外（之前）的操作结果的访问。在硬件层次，很难发现变量和同步访问之间的精确的结合关系。但是，我们将在 9.3 节中看到，放松模型的软件实现确实能发掘这种优化。

隔栅指令，如前所述，该指令带有 4 个风格位。每一位表示在先导和尾随的存-取（load-store）操作之间所强制的次序的一种特殊类型（4 种可能的次序是读对读、读对写、写对读、写对写）。这些位的设置可以组合，提供多种多样的次序选择。最后，IBM PowerPC 的模型（May et al. 1994; Corella, Stone, and Barton 1993）仅提供与 Alpha 的 MB 隔栅指令等价的单一的隔栅指令 SYNC。它的写入不是像处理器同一性（PC）模型中那样是原子性的，这是它与 Alpha 和 RMO 模型的区别。PowerPC 面对的模型是 WO，它是通过在每个同步操作的前后放置 SYNC 指令实现的。在习题 9. 13 中我们将看到如何用这些原语合成不同的模型。

表 9-1（Adve and Gharachorloo 1996）归纳了我们刚刚讨论的这几种著名的规范[⊖]。它们的性能不同，需要不同的强制次序的标记方式。值得注意的是，如果我们定义程序原序是程序员所见的次序，那么只有那些允许在代码段内改变读和写两者的操作次序的模型（WO、RC、Alpha、RMO 和 PowerPC）才能带来许多重要的编译器优化所需的灵活性。当编译器分析技术取得重大进展，能决定特定的同一性模型所允许的可能的重排序时，这种情况或许会改变。推论可允许的变序并插入强制次序的指令的困难性是显而易见的，规范的可移植性问题同样困难。例如，具有足够的存储器栅障，能在 TSO 系统中“正确”工作（产生直觉的或顺序同一的执行）的程序不一定能在 RMO 系统“正确”工作：它需要更多的特殊的操作。所以让我们考察对程序员更方便、更利于移植到不同系统的更高层次的接口，安全地开发各种系统所能提供的性能优势和变序。

693

表 9-1 各种系统规范的特征

模型	写对读变序	写对写变序	读对读/写的变序	对它者先前写的读	对自己先前写的读	保持次序的操作
SC					是	
TSO	是				是	MEMBAR, RMW
PC	是			是	是	MEMBAR, RMW
PSC	是	是			是	STBAR, RMW
WO	是	是	是		是	SYNC
RC	是	是	是	是	是	REL, ACQ, RMW
RMO	是	是	是		是	various MEMBARs
Alpha	是	是	是		是	MB, WMB
PowerPC	是	是	是	是	是	SYNC

注：在某列中的“是”表明对应的次序可以被对应系统模型所违反。“对其他处理器先前写的读”意味着允许处理器在写操作全局结束之前就可以观察到该写操作的结果。

9.1.2 程序员接口

程序员接口受到 WO 和 RC 模型启发，它们假设在同步操作之间完全不必保持程序原序。其概念是程序应保证所有的同步操作，包括使用标记的点到点事件同步都被明确地加以标记或指示，这是程序员的任务。编译器和运行时库根据系统规范，将这些同步操作翻译成

⊖ 表中“对自己先前写的读”这一放松与程序原序和写入原子性两者有关。这允许处理器在它自己先前的写操作尚未与其他对同一位置的写操作排好顺序时（在写结束之前），读出写入的内容。依赖这种放松的一种常用的硬件优化手段是让处理器从它自己的写缓冲器里读变量的值。这一放松几乎能用于所有的模型而不违反它们的语义。只要保持程序原序和原子性的需求，它甚至能用于 SC 模型。

适当的保持次序的操作（存储器栅障或隔栅）。这样，系统（编译器加上硬件）保证顺序同一的执行，尽管它可能在同步操作之间以它所希望的方式改变操作的次序（但不违反进程内对访问位置的依赖性关系），这是系统的任务。这种分工的约定给予编译器充分的灵活性，使它能在同步点之间按其愿望改变操作次序。它也允许处理器在其存储器模型或其实现允许的情况下，进行尽可能多需要的变序而不影响其可移植性：如果能保证 SC 在允许所有重排序的较弱模型下执行，当然能保证 SC 在允许较少重排序的系统上的执行。在程序员接口呈现的同一性模型至少应像硬件接口一样弱（放松），但不一定是相同的。

694

标记了所有同步事件的程序叫做*同步的程序*。人们开发了说明同步程序的形式化模型，即受弱序影响的*无数据竞争的*（data race-free）模型（Adve and Hill 1990a）和*受释放同一性*（Gharachorloo et al. 1990）影响的*适当加标*（prorerly labeled）模型（Gharachorloo et al. 1992）。感兴趣的读者可以从这些参考文献中获得更多的细节信息（模型间的区别是不大的）。程序员必然提出的问题是哪些操作应标记为同步操作？当然，如果使用显式的、系统说明的编程原语如加锁和栅障，这个任务一般已经完成了。这些原语也能容易地被区分为获取或释放，以便使 SC 这样的存储器模型能利用这种区分。例如，加锁是获取，解锁是释放；栅障包含这两者，因为到达栅障是一个释放（表明先导的访问的结束），而离开栅障是一个获取（获得对新一组访问的许可）。真正的问题是怎样才能决定应把哪些作用于普通变量（如 flag 变量）的存储器操作标记为同步操作。通常，程序员容易识别这些操作，因为他们知道什么时候使用这种事件同步。下列定义描述了在其他办法无效时识别同步事件的一种更为通用的办法。

- *冲突操作*。如果来自两个进程的存储器操作访问同一存储器单元，而且至少其中之一是写，则我们说它们是*冲突的*。
- *竞争操作*。这是冲突操作的子集。如果两个冲突存储操作（来自不同进程）以顺序同一的全序（执行）相邻，即它们相互紧邻，其间没有插入任何对共享数据的存储器操作，则它们是*竞争的*。
- *同步的程序*。如果一个并行程序中所有竞争的存储器操作都被标记为同步操作（或许将读操作标记为获取，写操作标记为释放，从而区分获取和释放），则它是同步的。

"竞争"意味着在任何可能的 SC 交错条件下的竞争，这一事实是编程接口的一个重要方面。尽管系统使用一个放松的同一性模型，但推论哪里需要标记这件事本身却可以在假定直觉的、SC 执行模型的条件下进行，这样就使程序员不必直接就重排序进行推论。当然，如果编译器能自动决定冲突或竞争的操作，程序员的任务会简单得多。但是，这个问题与在同一性模型下决定可能的重排序一样困难，因为目前已知的分析技术运行代价高而且保守（Shasha and Snir 1988）；Krishnamurthy and Yelik 1994，1995），任务几乎总是留给了程序员。

695

现在来考察图 9-7，它重复了图 9-3 中的例子。根据前面的定义，对变量 flag 的访问是竞争操作。它真正的含义是，在多处理器系统中，它们可能在各自的处理器中同时执行，彼此没有次序，这样我们就无法保证哪一个执行首先结束。所以，它们也构成了数据竞争。与此相对照，P_1 和 P_2 对变量 A（以及对 B）的访问是冲突操作，但它们被 SC 交错分隔而决定次序，SC 交错是由 P_1 对变量 flag 插入的写和 P_2 对 flag 对应的读形成的。因此对 A 和 B 的访问不是竞争访问，不必将它们标记为同步操作。

让我们以稍微形式化一点方法来说明这个问题。给定一个特定的 SC 执行次序，发生了（来自任何进程）对某个单元的冲突操作的次序是冲突次序。此外，各个进程具有*程序原序*。

图 9-7 显示了我们的代码段的范例执行，其中的弧对应了程序原序和冲突次序。如果在任何 SC 执行（交错）情况下，在两次访问之间总是存在其他访问的链，而链中至少有一段链接是由程序原序弧而不是冲突次序弧所构成，那么这两次访问就是非竞争的。否则，它们就是竞争的。完全形式化的定义可在文献（Gharachorloo 1995）中得到。

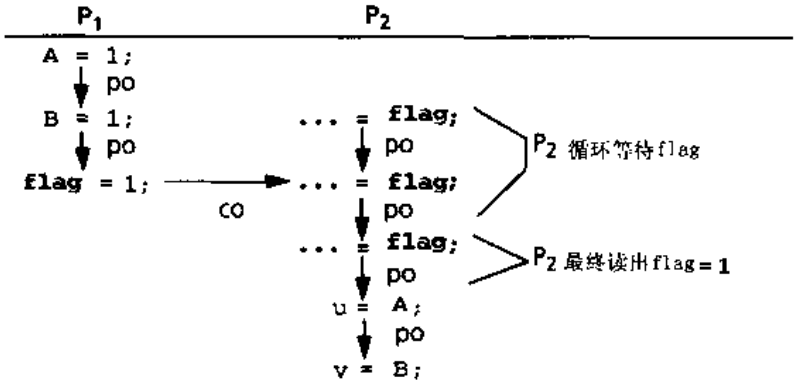


图 9-7　说明程序和冲突次序的代码序列的例子。标记了“po”的弧线显示了程序的次序，标记了“co”的弧线显示了冲突次序。注意，在 P_1 对 A 的写和 P_2 对 A 的读之间是由程序原序和冲突次序组成的访问链。这对于程序的所有 SC 执行都是成立的。这样的包括至少一个程序原序弧的链意味着访问是非竞争的。而对于变量 flag 的访问不是这样的链，因为在访问之间仅有冲突次序弧。黑体字指出对变量 flag 的访问对于排序是重要的，而且事实上也被用来协调事件同步

当然，同步程序的定义允许程序员超出必须保守地标记更多的同步操作而不牺牲正确性。在极端情况下，把所有的存储器操作标为同步操作总是能产生同步的程序。当然，这种极端情况使我们无法利用系统提供的对非同步操作的重排序去获得性能的改善，由于插入的保持次序的指令带来开销，所以在大多数系统中，这种极端情况会导致比直截了当的 SC 实现还要坏的性能。所以，我们的目标是仅仅把竞争操作标记为同步操作。

696

在某些特殊的场合，我们可能希望允许程序包含数据竞争，因而决定不把某些竞争访问标注为同步操作。这样，我们不再保证 SC 语义，但我们出自对应用的了解，知道竞争操作并没有被用作同步操作，而且在某些代码段中，我们不需要强的次序保证。第 2 章中有这样的一个例子，在该例中我们使用异步的方程求解器而不是红－黑次序。在栅障和格扫描之间没有同步，所以在一次扫描中，对一个分割的边界元素的读和写是竞争访问。如果它们未被标注，该程序在允许变序的系统中不再满足 SC 语义。但这没关系，因为求解器重复扫描直至收敛。即使进程在一次扫描中有时读到旧值，有时读到新值，而且不可预料，它们将在下一次扫描（在通过栅障之后）中会读到更新过的值并向收敛前进。如果标注了竞争访问，将会牺牲访问变序和性能。到达收敛的扫描次数可能稍微少一点，但各次扫描的代价却更大了。

关于编程接口的最后一个问题是程序员如何说明用于竞争访问的标记。在很多情况下，这是相当有特点的，并已经存在于程序设计语言之中。某些并行程序设计语言，如高性能 Fortran（High Performance Fortran Forum 1993），其风格允许并行性的表示，提取相关信息不是大问题。比如，在 FORALL 循环（由彼此独立的迭代组成的循环）中，只有循环结尾处的隐含栅障需要被标注为同步。FORALL 说明系统不必担心循环体中存在数据竞争。在更为通用的编程模型中，如果程序员使用诸如 LOCK、UNLOCR 和 BARRIER 这样的同步原语库，那么即使这些原语是使用普通的存储器访问实现的，库的设计者也会对实现它们的代码加以标

注，程序员无须做任何特殊的处理。最后，如果应用程序员希望对存储器操作增加标记，比如像图 9-4 所示的 flag 变量访问或为了保持某种其他的次序，他需要来自编程语言和库的支持。程序设计语言可以对一个变量的说明提供一种属性，表明所有对该变量的访问都是同步访问；或者在语句级有某种记号，指明某个特殊的访问应标记为同步操作。这会通知编译器在通过这些点时限制变序，而编译器则把这类访问翻译成处理器保持次序的适当的机制。

697

9.1.3 翻译机制

对大多数微处理器而言，将标记翻译成保持次序的机制意味着在被标为同步（获取或释放）的操作之前和之后插入适当的存储器栅障或隔栅指令。如果各条取和存指令本身带有指示强制何种次序的风格位，就可以避免额外插入指令。但是，由于同步操作通常并不经常发生，大多数微处理器并不采用在指令集层次做核心改动的方法。

9.1.4 真实的多处理器系统中的同一性模型

随着多处理器系统销售的巨大增长，现代微处理器的设计允许其在这类机器中的无缝集成。结果，微处理器厂商花费了相当大的努力，定义和精确地说明在硬件/软件接口提供的存储器模型。尽管顺序同一性模型仍然是程序员用于推导程序行为的最好的模型，许多厂商为了追求性能，允许更放松的次序。某些厂商，例如 Silicon Graphics 在多指令并行发出、动态调度的处理器（它的 MIPS R10000 处理器）中允许操作的执行乱序地发出，但非乱序地结束或为外界所见，以此继续支持 SC。这允许动态调度的处理器在相当程度上重迭存储器操作，因而不再满足 SC 的充分条件，但是它强迫操作按程序原序结束。Intel 的奔腾系列支持处理同一性模型，读可以在按程序原序先导的写之前结束。Sun Microsystems 的许多微处理器支持 TSO，允许与此相同的变序。还有很多其他厂商采用了允许放松所有次序的模型（例如，Digital Alpha 和 IBM PowerPC），在必要时提供存储器栅障来强制次序。

在硬件接口层，多处理器系统通常遵循所采用的微处理器提供的同一性模型，因为这是最容易的做法。例如，NUMA-Q 硬件输出它的 Pentium Pro 处理器的处理同一性模型。特别是当写入时，拥有权或数据在作废动作开始之前就获得了；允许处理器一旦获得拥有权就马上完成写并继续，而 SCLIC 通信辅助部件负责管理作废和应答的次序。通信辅助部件也可以有限地改变模型，我们在 Origin2000 中可以看到这样的例子。保持处理器的 SC 模型要求通信辅助部件（Hub）在写时只有当收到排他回答和所有的作废应答时才对处理器做出回答（推迟的排他回答），然后，动态调度的处理器可以从它的指令预取缓冲器中撤消写，允许后续的操作的撤消和结束。如果 Hub 在收到排他回答后立即回答处理器（积极的排他回答）而不等待作废应答的接收，那么写可以撤消，后续的操作（包括写）就可以在上述的写真正完成之前结束或为外界所见，这样同一性模型就更为放松。本质上，Hub 是在写的结束时刻上蒙骗了处理器。

698

在上述积极的排他式应答的例子中，用通信辅助部件蒙骗处理器可以提高性能，但同时增加了设计的复杂性。作废应答的处理现在必须由通信辅助部件根据所希望的放松同一性模型异步地完成，甚至在回答已经发送给处理器之后它还要跟踪处理器接口的缓冲器。还有一些其他问题，例如若当前处理器含有一个来自其他处理器的存储块，那么当对那个块的写作废操作已经出现但还未完成时，是否应该允许当前处理器继续访问这个存储块呢（见练习

9.3)？而且，当写作废操作在通信辅助部件中处于待完成状态时，如果当前处理器必须将这个存储块写回存储器（由于替换的原因），会发生什么情况呢？对于后者，或者由辅助部件对写回操作缓冲，把它延迟到所有作废都被应答之后再进行；或者必须扩展协议，使得较晚的对于写回块的访问在其请求者收到应答之前不能被该块的宿主满足。由于这种额外的复杂性，而性能改善又有限，Origin2000 的设计者仍然保持顺序同一性模型和延迟的排他回答。

在编译器方面，存储器同一性模型当前尚未很好定义，使事情对程序员而言变得复杂化。如果编译器随其所愿在访问到达处理器之前改变它们的次序（就像单处理器的编译器所做的那样），那么无论处理器支持顺序同一性还是处理器同一性，对程序员都没多大好处。微处理器的存储器模型定义于硬件接口，它们关心的是提交给处理器的程序原序，而假定编译器会做独立的处理。正如我们已经讨论过的，输出像 TSO、PC 和 PSO 这样的中间模型到程序员接口，没有给编译器留下足够的重新排序的自由度。例如，程序员或许会假定在实际中编译器不会违反同一性模型而改变操作的次序或消除操作，因为编译器对存储器操作的大多数重排序都集中于循环。但这是一个非常危险的假设，有些时候，我们所依赖的次序确实发生在循环之中。为了真正在程序员接口使用这些模型，必须对单处理器的编译器加以改造，使其遵循模型对重排序的限制，但明显牺牲了性能。这些中间模型被硬件接口层支持，但并不特别适用于程序员接口（当然，如果编译器能检测出竞争操作就不同了，在这种情况下，它能对程序员输出最强的 SC 模型，而本身执行讨论过的最放松的模型的重排序）。

像 Alpha、RMO、PowerPC、WO、RC 和同步程序等更为放松的模型甚至可以用于程序员接口，因为这些模型能给予编译器所需的灵活性。在程序员接口传递排序约束的机制不单必须为处理器所注意，编译器也必须遵守，多处理器的编译器正在开始这样做（见 9.5 节）。699
我们在关于同步程序的叙述中知道，在程序员接口的放松模型之下，硬件接口能使用相同或更强的排序模型。但是，人们希望在处理器接口使用放松的模型以便发挥性能潜力。当我们转向讨论用数据的一致副本支持共享地址空间的不同措施时，我们会看到，若想用大于高速缓存块的颗粒支持一致性，放松的同一性模型对性能是至关重要的。我们还会看到，存在比释放同一性还要放松的同一性模型。(你能想出如何做吗?)

9.2 克服容量限制

在像 SGI 的 Origin2000 这样的 CC-NUMA 系统中，在访问数据时，处理器高速缓存直接复制远地分配的数据，并不将数据首先在本地主存中复制。当高速缓存扑空时，通信辅助部件根据物理地址决定是查找本地存储器和目录状态还是直接向远地的宿主节点发出请求。通信的粒度、一致性的粒度以及复制存储器（高速缓存）分配的粒度都是高速缓存的块。正如前面所讨论过的，这样的系统存在的问题是本地复制的容量受硬件高速缓存所限。如果一个远地安装的块被从高速缓存替换，当再次需要它时，必须从远地存储器中去取，这将导致人为的通信。本节中所讨论的系统的目标是用硬件提供高速缓存块粒度的一致性的前提下，克服复制容量的问题。

9.2.1 第三层高速缓存

达到这个目标的一个办法是像 Sequent 的 NUMA-Q 和 Convex 的 Examplar（Convex Computer Corporation 1993；Thekkath et al. 1997）那样使用一个大而慢的远程访问高速缓存。如果机器

节点本身就是小规模的多处理器，为了对节点间协议提供每个节点的单一的高速缓存，这个高速缓存从功能上也总是需要的。这个远程访问高速缓存记录那些在远地分配，但当前位于本地处理器高速缓存之内的块，为性能的需要，可以简单地扩大它的尺寸。那么，它也可以保存已经被从本地处理器高速缓存中替换掉的远地块的副本。在 NUMA-Q 中，这个用 DRAM 实现的远程高速缓存至少有 32 MB，而节点上 4 个最底层处理器高速缓存之和仅仅是 2～4 MB。有一个类似的办法，有时也称为第三层高速缓存（tertiary cache），是把本地主存的固定部分用作远程高速缓存，它需要额外的硬件实现每节点的标记和状态。

这些方法以很小的粒度在主存中复制数据，但它们并不自动地把一个块的宿主迁移（migrate）或改变到最经常发生该块的高速缓存扑空的节点。在块的宿主的主存中总是为该块分配空间。所以，如果应用未能把数据适当地分布到主存中，那么对于第三层高速缓存这种方案，即使只有一个处理器曾经访问过某存储器块（即不需要多个副本或复制品），系统最终会浪费它的可用主存的一半，因为每个块在主存中都有两个副本，一个在宿主，另一个在第三层高速缓存，但是只会用到一个。此外，如果其复制容量并不为性能所需，静态建立的第三层高速缓存是太浪费了。另一种增加复制容量的途径，即惟有高速缓存的存储器体系结构（cache-only memory architecture，COMA），是把整个本地存储器当作硬件控制的高速缓存。这个方案同时实现复制和迁移，并不存在上述问题，后面将对它做更深入的讨论。在所有情形下，复制和一致性是以细粒度管理的，虽然并不一定非要和处理器的高速缓存的块尺寸相同。只有那些已经在本地存储器、远程高速缓存或第三层高速缓存中的数据才被带入处理器高速缓存层次结构，因为数据在较外层次已保持跨节点的一致，无须再用单独的节点间协议保持处理器高速缓存本身跨节点的一致性，所必须保证的仅是处理器高速缓存与本地存储器或远程/第三层高速缓存的一致性。

700

9.2.2 惟有高速缓存的存储器体系结构

在 COMA 机中，整个主存中每个细粒度的存储器块都带有一个硬件的标记。对一个存储器块而言，并不存在一个保证为其分配空间的固定节点。相反，数据是动态地迁移，在那些访问过它或者说“吸引”它的节点的主存中被复制，所以，这些组织称惟高速缓存的主存为吸引存储器（attraction memory）。当一个远地块被访问时，它被复制到吸引存储器并被装入处理器的高速缓存，由硬件保证一致性。块的迁移的实现是通过替换或作废吸引存储器，如果块 x 本来就在节点 A 的主（吸引）存储器中，当节点 B 读它时，B 将获得一个副本（复制）；如果 A 中的存储器后来被 A 访问的另一块所作废或替换，那么该块的仅有的副本就在 B 的存储器中了。所以，不存在第三层高速缓存方案中存在的浪费原始副本的问题。数据的迁移和空间的管理两者都是需求驱动的，因为一个数据块可能存在于任何吸引存储器之中，并且透明地在吸引存储器间移动，数据的位置不再与它的物理地址相关。自动的数据迁移对多道程序负载还有另一个潜在的优点，即操作系统可以在任何时候决定在节点间迁移进程，虽然在这种情况下，用软件迁移页面也是成功的。

1. 硬件/软件的折中

和其他方法一样，COMA 方案引入明确的硬件/软件折中。通过克服纯粹的 CC-NUMA 方案存在的高速缓存容量限制问题，我们的目的是使软件不必再为数据在主存中的分布担心。程序员可以把机器看作具有一个中央的主存，他只需关心固有的通信和伪共享等问题（当

701

然，如果数据分布不适当，冷扑空仍必须由远地满足）。虽然这使软件的编写者的任务大大简化了，COMA 机与纯 CC-NUMA 机相比，需要多得多的硬件支持，因为它们把主存按硬件高速缓存实现。主存要为每个块设立标记并需要比较器。在吸引存储器中复制带来的额外的存储器开销，我们将在本节后面在讨论。最后，吸引存储器的一致性协议比处理器高速缓存的一致性协议复杂。究其原因有二，且两者都与下述事实有关，即数据动态移动到对它进行访问的节点，不存在保存它的固定的“宿主”。首先，当发生吸引存储器扑空时，必须决定数据的位置，因为数据不再与物理地址绑定。其次，不再需要在宿主为块保留空间，所以，在从吸引存储器替换一个块时，必须保证不要丢失系统中该块的最后一个或仅有的副本。第三层高速缓存方案不存在这额外的复杂性。

2、性能权衡

性能有其自身的有趣权衡。虽然，COMA 机减少了人为通信造成的远地访问，但它增加了需要由远地才能满足的访问的时延，包括冷的、真实共享和伪共享的扑空。其原因是对于一个不会被本地吸引存储器满足的高速缓存扑空，也需要先在该吸引存储器中查找一下，了解是否存在该块的一个本地副本。而且，吸引存储器访问的代价要比标准的 DRAM 访问要高，因为吸引存储器通常以组相联的方式实现，标记选择操作处于关键通路上。

就性能而言，COMA 最适合于那些对处理器高速缓存具有高容量扑空率的应用（大工作集情况），这些访问扑空是针对没有在本地分配的数据的，而 COMA 对那些性能由一致性扑空所主宰的应用最不利。当访问模式不可预见或来自不同进程的访问以很小粒度在空间交错从而以软件方式实现页面粒度的数据的放置、替换或迁移在 CC-NUMA 机上变得困难时，COMA 的优点最显著。例如，对于近邻网格计算而言，使用二维数组表示比使用四维数组表示更适合于 COMA 机，因为在后一种情况下，以软件方式通过操作系统以页面粒度适当地分布数据并不困难。事实上，当使用四维数组而且数据适当分布时，较高的通信开销使 COMA 的性能比纯粹的 CC-NUMA 更差。图 9-8 根据应用的特征总结了性能权衡。让我们简要地看一下 COMA 协议的某些设计选择，以及它们是如何解决在扑空时能找到数据和不丢失块的最后副本这些协议问题的。 702

3. 设计选择：扁平法与层次法的比较

可以采用层次式或扁平式的目录方案，甚至层次式的总线侦听来实现 COMA 机（见第 8 章 8.10.2 节）。Data Diffusion Machine 的原型（Hagersten，Landin，and Haridi 1992；Hagersten 1992）使用了基于层次式目录的 COMA，出自 Kendall Square Research 的商品化系统（Frank、Burkhardt and Rothnie 1993）使用了层次式的总线侦听型的 COMA。在这些层次式的 COMA 方案中，在扑空时是通过遍历层次结构来找到数据的，这和第 8 章所讨论的非 COMA 层次协议一样。区别在于，在非 COMA 机中，处理节点中的每一个存储器块都有一个固定的宿主节点，而 COMA 机则没有。当一次访问未命中本地的吸引存储器时，它沿层次结构向上，直到发现某节点指示该块存在于它的子树中且具有适当的状态为止。然后该请求又沿层次结构向下，到达对应的处理节点（它是一个叶节点），根据目录查找数据或者沿着路径采用总线侦听方式找到数据。

在扁平的 COMA 方案中，就为数据保留位置的意义而言，还是没有存储器块的宿主。但是，却有一个能找到目录信息的固定的宿主节点（Stenstrom，Joe，and Gupta 1992；Joe 1995）。这个固定的宿主是通过物理地址（如 CC-NUMA 那样）或者从物理地址得到的全局标识符决定的。目录信息的（静态的）位置与真正的数据的（动态变化的）位置无关。本地吸引存储

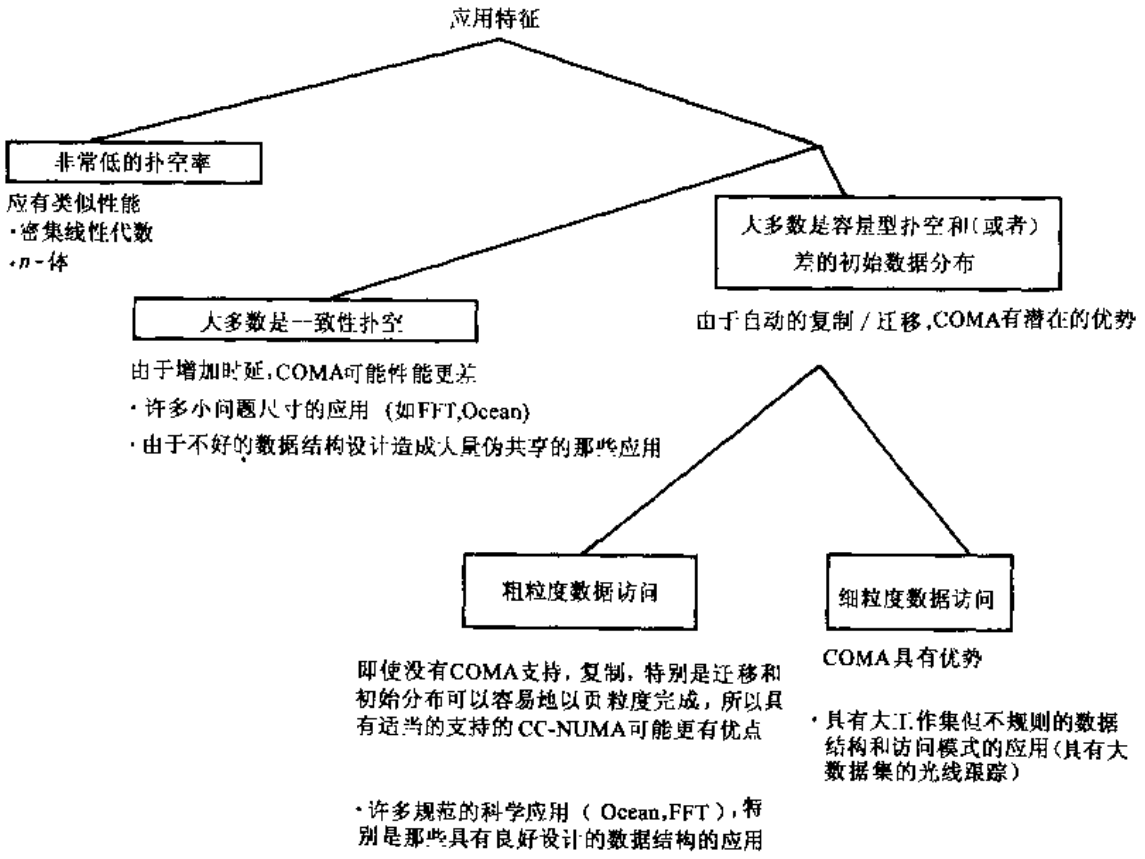


图 9-8 COMA 和 CC-NUMA 体系结构的性能权衡。方框中的是应用的特征，在每个方框下是采用相同技术建造的 COMA 和 CC-NUMA 系统针对该组特征的性能比较，后跟一系列应用领域的例子

器的一个扑空导致查找宿主中的目录信息，该目录以基于存储器或基于高速缓存的方式记录了副本的实际所在地。层次式目录和扁平式目录之间的折中与非 COMA 情况十分类似（见第 8 章 8.10.2 节）。

让我们看一看层次式和扁平式的方案如何解决最后副本的替换问题。如果要从吸引存储器中替换的块处于共享状态，而且如果我们能肯定在系统中存在该块的另一个副本，就可以安全地丢弃被替换的块。但如果一个块处于独占状态或是系统中最后一个副本，必须保证它能在另一个吸引存储器中找到一个位置，而不是把它丢弃。在层次式情况下，对于处于共享状态的块，简单沿层次结构向上，直到发现一个节点，它指明该块的副本存在于它的子树的某处。然后，只要我们已经沿着该路径更新了路径上的状态信息，就可以丢弃那个块。对于一个处于独占状态的被替换块，我们沿层次向上，直到发现某个节点，在该节点的子树的某处存在一个处于无效或共享状态的块，可用要替换的独占块替换它。如果可替换的块处于共享而不是无效状态，那么它的替换需要遵循前面讨论过的替换共享块的过程；如果可替换块处于无效状态，情况较简单，替换就可以。

扁平的 COMA 解决最后副本问题需要较多的机构，因为它没有搜索可用空间的内置机制。一种机制将存储器块的一个拷贝标记为主拷贝，它保证主拷贝在替换时不会被丢弃。在吸引存储器中增加一个新的称为主块的高速缓存状态。在数据最初分配时，每个块都是主拷贝。以后，主拷贝或者是一个独占的副本，或者是共享副本之一。当一个块的非主拷贝共享

副本被替换时，可以安全地丢弃（如果我们愿意的话，可以向宿主的目录项发送一个替换提示）。如果一个主拷贝被替换，必须向宿主发一个替换消息。宿主选择另一个节点，把主拷贝发过去，希望能在那个节点找到空间，然后把那个节点置为主节点。如果这个新节点上那部分吸引存储器所有可用的块都是主块，那么请求就被发回宿主，宿主再试另外一个节点，否则可替换的块之一被替换并被丢弃（[Joe and Hennessy 1994] 讨论了某些优化）。

703 ~ 704

不论我们使用层次式的还是扁平的 COMA 协议，应用的初始数据集不应该占据整个主存或吸引存储器，以保证系统中有足够的可用空间使被替换的最后的（主）拷贝能找到新的存储位置。为了帮助找到可替换的块，吸引存储器应该是高度相关的。缺少足够的、额外的（初始未分配的）可供替换的存储器会导致性能问题，其原因是：首先，它使机器的 COMA 特性在本地满足高速缓存的容量型扑空方面不够有效。其次，它意味着为了给被替换的主拷贝腾出空间，一些仍然有用的块被替换掉。第三，替换最后副本造成的流量可能相当大，引起系统内的大量竞争。至于究竟应为替换留下多少存储器以及需要多强的相关性，可以凭经验决定（Joe and Hennessy 1994）。

4. 总结：一个读操作的途径

考虑一个扁平的 COMA 方案。存储器管理部件首先把一个虚地址转换为一个物理地址。这可能引起缺页，需要建立一个新的映射；这和单处理机一样，虽然在 COMA 情况下，该页的实际数据并不装入主存中。物理地址被用来查找各层高速缓存。如果命中，访问被满足，否则，它必须查找本地吸引存储器。物理地址的某些位被用来找出吸引存储器中的相关的组，而硬件维持的标记存储器被用来检查标记的匹配。如果发现块处于合适的状态，访问被满足。如果状态不合适，必须产生远地的请求，并将该请求发往由物理地址决定的宿主。宿主的目录决定应该把请求向哪里转发，以及数据是处于共享状态还是独占状态，拥有数据的节点以物理地址作为它的吸引存储器的索引，找到数据并将其返回。目录协议保证状态的正确维持。

9.3 降低硬件成本

本章讨论的最后一个主要问题是硬件的成本。降低成本常常意味着把专用硬件的某些功能移往在现存硬件或商品硬件上运行的软件。在这种情况下，涉及的功能有复制和一致性的管理。因为用软件在主存中控制这些功能比在硬件高速缓存中实现要容易得多，正如 COMA 或第三层高速缓存系统所做的那样，低成本的方案倾向于用主存提供复制和一致性。与 COMA 或第三层高速缓存系统不同之处在于，较高的开销和通信辅助部件占用率以及管理复制和一致性的粒度。

705

让我们考虑纯 CC-NUMA 方法的硬件成本。一个节点上通信体系结构部分可以被分成四部分：检测访问控制违规的的部分；用于此项用途的每个块的标记和状态；完成实际的协议处理（包括干预处理器高速缓存）的部分以及网络接口本身。为了以硬件保持以高速缓存块为粒度的数据一致性，访问控制部分需要看到未在高速缓存中命中的对共享数据的每一个取和存操作，以便采取必要的协议动作。所以，辅助部件必须能侦听本地存储器系统（以及向包括高速缓存在内的本地存储器发出请求，以响应来自网络的请求）。

就一致性的有效管理而言，辅助部件其他各个功能部分都大大受益于硬件的专门化和集成。要快速地确定访问失败，要求主存每个块的标记靠近辅助部件的访问控制部分。辅助部

件离高速缓存越近，协议动作激活和辅助部件干预处理器高速缓存的速度越高。协议操作的快速执行要求辅助部件或者用硬件实现，或者是可编程的（像 Sequent 的 NUMA-Q 那样），且专门针对协议最经常执行的类型的操作（例如，位域的抽取和处理）做优化。最后，辅助部件和网络接口之间小块数据的快速传送要求网络接口与通信辅助部件紧密集成。因此，为追求高性能希望把通信辅助部件的四个部分紧密集成，使得它们之间通信的总线越少越好，整个通信辅助部件专用并与节点的存储器系统紧紧集成在一起。

早期的支持高速缓存一致性的机器把硬接线的辅助部件集成在高速缓存控制器内部，并把网络接口紧密地集成进辅助部件之中，从而达到上述目的。然而，现代的处理器一般都在处理器芯片上集成它的第二级高速缓存控制器，因此，处理器一旦完成，就很难再将辅助部件集成到控制器内部。所以 SGI 的 Origin 把它的硬接线的 Hub 集成到存储器控制器中，斯坦福的 FLASH 将它的专用的可编程协议机集成在存储器控制器里，而 Sequent 的 NUMA-Q 和 Hal S1 把专用控制器挂在存储器总线上。通过使用这些专用的、紧密集成的高速缓存一致性的硬件支持，这些方案未在通信体系结构中借助于便宜的商品化部件。所以，它们的昂贵在于设计和实现所需的时间，而不是在于实际使用的硬件的量。

人们试图通过几个不同的途径降低成本。途径之一是以专用硬件执行访问控制，但把其他许多工作降级到以软件或者商品化的硬件完成。其他的途径连访问控制也用软件实现，所以在商品化节点和网络上提供一个一致的抽象共享地址空间，无须专用硬件的支持。访问控制或是通过调整程序的办法以细粒度提供或是在现存的虚拟存储器的支持下以页粒度提供，或是通过使用运行时系统层以用户定义对象的粒度提供，该层对外提供基于对象的编程接口。

706

所有这些方案目前均处于研究阶段，我们将对它们逐一讨论。将更详细地讨论基于页的方案，这是因为它在保留与硬件一致性系统相同的透明编程接口的同时，改变了数据分配、交换、保持一致的粒度，也因为它的协议与已经了解的有很大不同，说明了充分发掘放松的存储同一性模型的放松条件所需的机制。

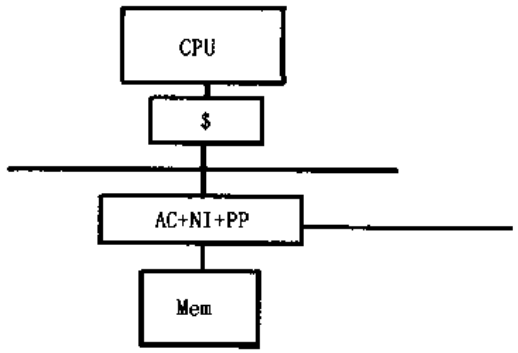
9.3.1 具有去耦辅助部件的硬件访问控制

尽管这个方案以硬件支持细粒度的访问控制，但其他功能的部分或全部（协议处理、标记、网络接口）能与该专用硬件去耦且彼此分离。它们或者采用商品化硬件将其附加到 I/O 总线这样的节点部件上，或者采用的额外硬件不超过通常单处理器节点的情况。例如，块的标记和状态可以放在一个专用快速存储器或常规的 DRAM 中，协议处理可以用一个独立的、便宜的通用处理器实现，或甚至就在主处理器本身用软件实现。网络接口通常具有对细粒度通信的特殊支持，以便降低端点的开销。图 9-9 中是各种功能的几种可能的组合方案。

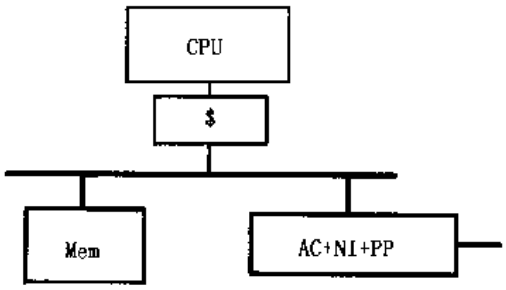
当然，去耦的硬件的问题在于加大了协议调用、协议处理和通信的时延，因为不同部件之间以及它们和节点之间的交互较慢（例如、它可能包含几次总线操作）。更关键的是，去耦的通信辅助部件比专用的集成辅助部件的实际占用率高得多，这对第 8 章 8.7 节所述的很多应用来说，会严重影响性能。

9.3.2 通过代码修改实现的访问控制

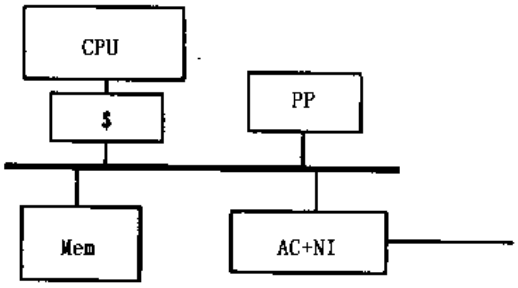
有可能在标准的单处理器节点上不使用额外的硬件支持，而是在主存中用软件实现细粒



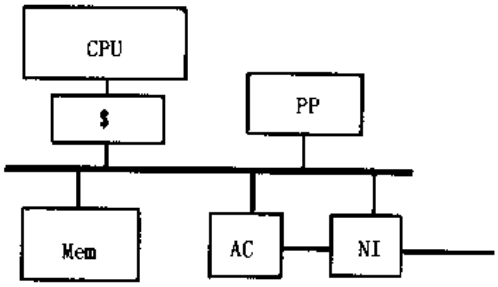
a) 访问控制、网络接口和协议处理部件被一起集成在存储器控制器之中（SGI Origin）



b) 访问控制、网络接口和协议处理部件集成在一起被挂在存储器总线上（例如，威斯康星大学的 Typhoon 建议）



c) 访问控制和网络接口集成在一起被挂在存储器总线上，协议处理部件独立地用商品化部件实现，也挂在总线上，（例如，威斯康星大学的 Typhoon-1 建议）



d) 独立的访问控制部件、商品化部件实现的网络接口和协议处理部件挂在存储器总线上（例如，威斯康星大学的 Typhoon-0 建议）

图 9-9 几种降低高度集成的、专用的辅助部件的成本的方案。AC 是访问控制部件，NI 是网络接口，PP 是协议处理部件（硬接线的有限状态机或可编程的）。a）显示了高度集成的方案，b）、c）、d）是集成度较低的方案。随着部件间的距离增加，部件间通信所需的费时的总线操作的数量上升。在这些设计中，c）和 d）中的商品化 PP 是和主 CPU 一样的完整的处理器，带有自己的高速缓存系统

度的复制和一致性。最复杂的部分是主存中的细粒度的访问控制，因为标准的单处理器并不对此提供支持。为实现这一点，我们可以对软件中的读和写进行修改，增加对主存中保存的块标记和状态数据结构的查看（Schoinas et al.1994; Scales, Gharachorloo, and Thekkath 1996）。根据对高速缓存扑空的预见，只有那些不能命中各层处理器高速缓存的读和写才需要被修改。所需要的协议处理可以在主处理器或在无论什么形式的辅助部件中执行。事实上，这种软件修改使我们能提供任何粒度的访问控制和一致性，甚至不同的数据结构能使用不同的粒度。

软件修改带来运行时的开销，因为它在代码中插入执行检查所需的额外指令。一些已经开发的方法利用几种技巧来降低所需的检查和查找的次数（Scale, Gharachorloo, and Thekkath 1996），使得访问控制和协议调用的开销能与去耦硬件的方案相竞争。主处理器的软件协议处理也有显著的代价，尽管这样的系统所采用的网络接口和互连一般提供对细粒度通信的支持，但通常是基于商品化的部件，因而效率比紧耦合的多处理器要低。

707 / 708

9.3.3 基于页面的访问控制：共享虚拟存储器

无需额外硬件支持提供访问控制的另一个途径是借助于微处理器的存储器管理部件和操

作系统所提供的虚拟存储器支持。存储器管理部件已经以页面的粒度对主存进行访问控制（例如检测缺页），在虚拟地址空间把主存作为全相关的高速缓存管理。通过在缺页中断处理程序中嵌入一致性协议，我们能够以页面的粒度提供复制和一致性，并且把主存作为共享虚拟地址空间的一致的、全相关的高速缓存管理（Li and Hudak 1989）。这样，访问控制就不需要特殊的标记，辅助部件甚至无须了解每一次高速缓存的扑空。只有当对应的页面已经在主存内时，数据才进入本地的高速缓存。与前面两个途径类似，不同节点的处理器高速缓存无需由硬件保持一致，因为当一个页面被作废，TLB 将不允许处理器访问高速缓存中的块了（当然，必须注意保持处理器高速缓存和本地存储器的一致性）。

这个途径叫做基于页面的共享虚拟存储器或简称 SVM。因为成本被整个页的数据分摊，协议处理通常由主处理器自己承担，我们可以更容易地实现没有特殊硬件支持的网络接口上的细粒度的通信。所以，除了标准的单处理器系统所提供的硬件辅助外，几乎不需要更多的硬件。

图 9-10 说明了遵循与纯 CC-NUMA 的作废协议十分类似的共享虚拟存储器一致性的一个非常简单的形式。有几点值得注意。第一，因为不同处理器的存储器管理部件独立管理它们的主存储器，P_1 的本地存储器中的页面的物理地址可能与其在 P_0 的本地存储器中的副本的物理地址完全不同，尽管它们有着相同的（共享的）虚拟地址。存在一个由私有的物理地址空间构成的共享虚拟地址空间。第二，实现协议的缺页中断处理程序在设置页面的合适的访问权限和将控制交回应用进程之前，必须能执行第 8 章所讨论的三个协议功能（找到状态信息的源，找到合适的副本和副本通信）。可以使用基于目录的机制，每个页面必须有一个由

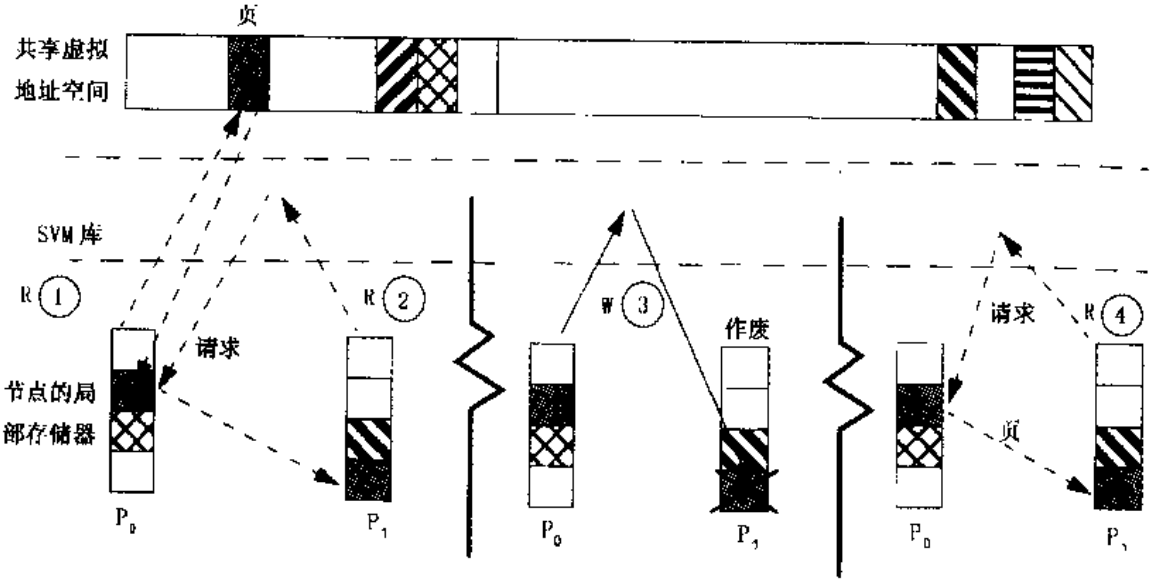


图 9-10　简单的共享虚拟存储器的说明。最初，没有节点保存我们所关心的灰色的共享虚页。事件以 1、2、3、4 的次序发生。读 1 在地址翻译时产生一个缺页事件，将该页的一个副本取到 P_0（假定从磁盘取）。在同一框中显示的读 2 引起缺页并将一个只读副本取到 P_1（从 P_1 取）。这是同一个虚页，但在两个存储器中位于不同的物理地址。写 3 引起缺页（对只读页的写入），在缺页中断处理程序中实现的 SVM 库判断 P_1 持有一个副本并将它作废。P_0 现在获得了对该页的读-写访问权，该页处于被修改过的脏状态。当 P_1 的读 4 试图读取作废页面的某个单元时，产生缺页，通过 SVM 库从 P_0 取得一个新的副本

其虚地址决定的宿主，由宿主维护目录项，我们将看到高性能的 SVM 协议要更复杂一些。

基于页面的共享虚拟存储器的问题在于协议调用和处理的高开销和一致性及通信的大粒度。前者是昂贵的，因为大多数的工作是在通用单处理器上由软件完成的。缺页中断或陷阱的产生、向操作系统的切换以及中断处理程序的调用都是费时的；协议处理本身是软件完成的，发往其他处理器的消息使用底层的消息传递机制，该机制是费时的，特别是商品化的节点和互连更是如此。在 1998 年的有代表性的 SVM 系统上，满足一个远程的缺页的往返的代价大约是几百微秒到 1 毫秒以上，这取决于系统软件的支持程度。与此对比，在硬件支持一致性的系统中，一个读扑空的开销低于 1 微秒。此外，由于协议典型地是由主处理器完成的（避免额外的硬件支持），进入的请求会中断处理器，弄脏高速缓存内容，减慢当前运行的应用线程（该线程或许与进入的请求毫无关系）。

有两个原因使大粒度的通信和一致性成为问题。首先，如果空间局部性不很好，会引起很多通信碎片和无用的数据传输（仅需要一个字，但必须取整个页面）。其次，它容易导致伪共享，这引起昂贵的协议操作和通信的频繁调用。在顺序同一性模型下，一旦检测到写入，立即传播和执行作废动作，因此，真实共享和伪共享可能导致页面在处理器之间不断地往返传送（图 9-11 显示了一个例子）。这种操作的高代价和高频率是不幸的组合，所以，缓解伪共享的效应，降低通信频率是重要的。这导致了与细粒度一致性完全不同的协议和途径，在细粒度情况下，伪共享的效应不显著。让我们更深入地考察这一问题。

709
710

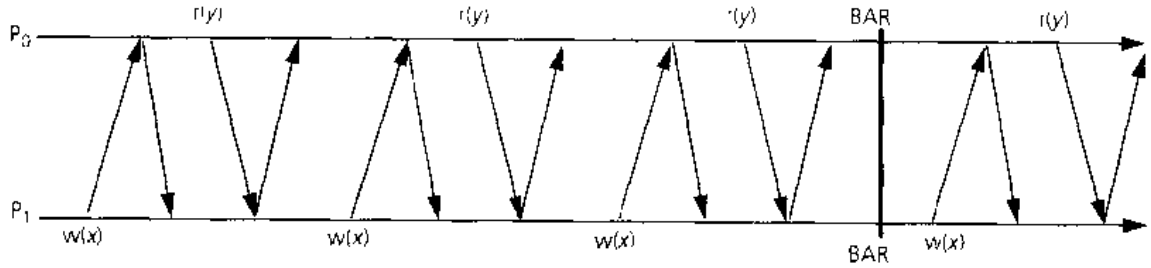


图 9-11　顺序同一性对 SVM 存在的问题。图中时间自左向右推进。一个进程所执行的操作在该进程的水平时间线的上下显示。进程 P_0 在进程 P_1 重复对变量 x 写入时重复读出变量 y，x 和 y 位于同一页面。因为在 SC 模型下，P_1 在作废被传播并被应答之前不能推进，作废必须立即被传播，从而这样的伪共享导致大量的（代价非常高的）通信反复发生

1. 使用放松的存储同一性

使用像释放同一性这样的放松的存储同一性模型能降低通信的频度。它允许把统称为写提示的像作废和更新这样的一致性动作推迟到下一个同步点执行（在此之前，写入不一定非被外界所知）。让我们继续假设基于作废策略的协议。图 9-12 显示了和图 9-11 相同的例子，处理器 P_1 对 x 的写入在到达栅障之前不产生对 P_0 中的页面的副本的作废，这样，伪共享的效应将大大缓解，在栅障之前 P_0 对 y 的所有读都不会引起缺页。当然，P_0 在栅障之后对 y 的访问会产生伪共享引起的缺页，因为页面已经被作废了。对于真实共享，也可以观察到类似的通信的降低，因为协议并不区分两者。根据同一性模型，真实共享的改写在下一个同步点之前不被看到也是可以的。

这里所讨论的情况与硬件一致性机器中或写入时对放松同一性的典型使用有显著的区别。在后一种情况，放松同一性主要用于避免因等待应答（结束）而阻塞处理器，但作废通

常立即传播并实施，因为这对硬件而言是自然而然可做的事。虽然，释放同一性不保证在同步之前显现写的效果，但事实上效果通常是显现的。所以，甚至在不存在同步的区间内，高速缓存块的伪共享的量以及网络事务或消息也未减少多少；其目标主要是对处理器隐藏延迟。对 SVM 而言，系统明确规定：在同步点到达之前，不真正传播作废。当然，对于正确性而言，清楚地标明所有的同步点并使之与系统通信是关键的[○]。

711

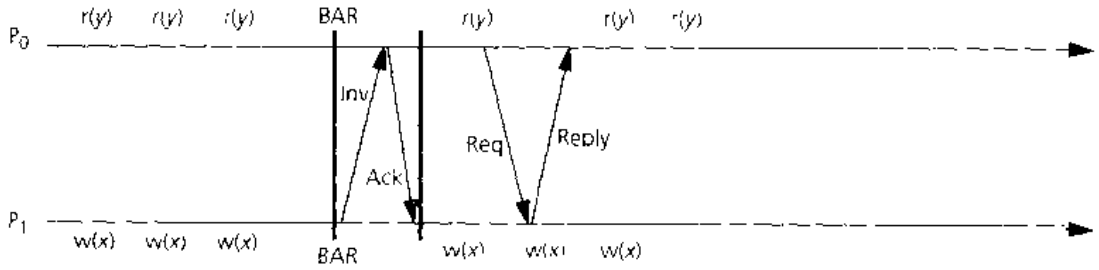


图 9-12　通过使用放松的同一性模型减少因伪共享引起的 SVM 通信。在同步之前的读和写不发生通信，在同步点传播作废，从而使页面一致。只有同步点之后对已经作废页的首次访问会产生缺页请求

那么何时作废（或写提示）必须从写入者传播到其他副本呢？它们又必须何时被实施呢？一个可能是当写入者发出一个释放操作时传播它们。在释放点，该进程将自上一个释放点以来写过的每个页面的作废传播到所有拥有该页副本的处理器。如果在通过该释放点之前等待作废的完成，则满足前面提到的 RC 的充分条件。但是，即使这样的传播也比必须的要早：根据释放同一性，进程在执行获取之前不一定真的需要看到写提示。在释放点向所有副本传播并实施写提示，称为积极释放同一性 Eager Release Consistency，ERC（Carter，Bunnett，and Zwaenepoel 1991，1995），是保守的做法，因为系统不知道其他处理器的获取何时发生，也不知道某个进程将来是否会执行一个获取而需要看到这些写提示。如图 9-13a 所示，它可能把一个通信代价昂贵的作废消息发送到并不需要它的进程去（P_2 无须被 P_0 作废）；它要求独立的作废消息和锁获取消息；它可能早于必要的时刻作废进程，从而导致伪共享（见变量 x 和 y 之间的伪共享，这是由于被 P_1 和 P_2 取了两次，一次在对 y 的读时，一次在对 x 的写时）。当使用基于更新的协议而且对一个页面的反复写入产生反复的更新时，额外消息的问题变得更为突出。这些问题将在习题 9.23～9.25 中进一步讨论。

712

最著名的 SVM 系统使用一种称为惰性释放同一性（Lazy Release Consistency，LRC）的释放同一性。如图 9-13b 所示，LRC 不是在跟随写入的那个释放点对特定进程传播和实施作废，而是在该进程下一个获取点执行这些动作（Keleher，Cox，and Zwaenepoel 1992）。在获取点，进程获得与所有先前的释放操作相关且发生于它前一个获取操作和它当前获取操作之间的写提示，并且对相关的页面实施写提示动作。识别发生于一个指定的获取之前的释放操作是一个有趣的问题。它们可以被定义为：按照同步操作的任何顺序的同一次序，必须在该获取之前出现所有释放操作（这也保留了它们之间的依赖关系）[⊖]。另一种识别的办法是对同步操作施加两类偏序：在各个进程内部的程序原序和由所有进程对同一同步变量的获取和释

713

○ 事实上，也可以采用一种类似的途径减少硬件情况下伪共享的效应。作废可以被缓存在请求部件的硬件中，只有在释放或同步点（取决于遵循释放同一性还是弱序同一性）或当缓冲器满时才送出。也可以把它们缓存在目的地，在目的地的下一个获取点才予以实施。我们称这个方法为延迟同一性（Dubois et al. 1991），因为它延迟了作废的传播。只要处理器看不到作废，它们就可以继续使用它们的副本而无须任何一致性动作，缓解了伪共享的效应。

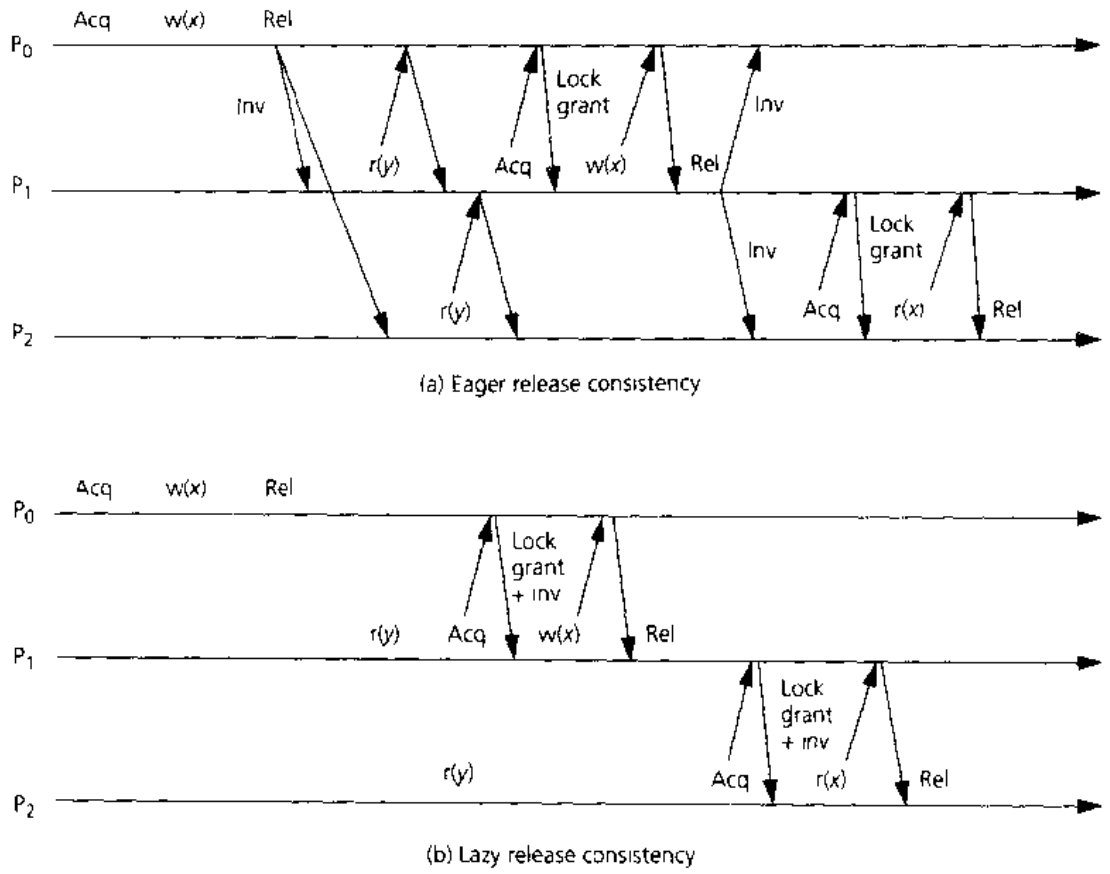


图 9-13 释放同一性的积极和惰性实现方式的比较。积极释放同一性在释放点执行同一性动作（作废），而惰性释放同一性在获取点执行同一性动作。变量 x 和 y 处于同一页面。通信显著减少，特别是对于有较大一致性粒度的 SVM 而言

放所动态决定的依赖次序。对同步变量的访问构成一个遵循依赖次序的成功的获取和释放的链。当一个获取请求来到一个正在执行释放的进程 P，在该释放之前发生的同步操作是那些先于释放点的程序原序和依赖次序交集中的同步操作。我们说这些同步操作在因果意义上，发生于该释放之前。在获取之前的操作是按程序原序先于该获取的操作的并集。图 9-14 澄清了同步操作之间的因果次序这一概念。

通过将一致性动作进一步推迟到获取之时，LRC 缓解了与 ERC 相联系的三个问题；例如，如果产生页面伪共享的存储器操作在获取之前而不是之后发生，它的不良效应将不会被看到（图 9-13b 中对 y 的读不会产生缺页）。另一方面，我们将看到，由于某些应完成的工作和通信从释放点移到了获取点，LRC 的实现比起 ERC 要复杂得多。可能存在某些中间的方法，比如在释放点传播写提示到获取点才实施它们，这样做并不能减少写的流量，但可以降低缺页。然而，LRC 是当前应选择的方法。 714

这些基于页面的软件协议与为硬件一致性系统所开发的同一性模型之间的关系同样令人感兴趣。软件协议不满足第 5 章所讨论的一致性的要求，因为除非存在适当的同步，写并不保证会自动传播。只能通过同步操作才能观察到写这一点使得写的串行化更难保证。不同的

㊀ 这种次序甚至可以是处理器一致性的，因为 RC 允许按程序原序获取（读）旁路前面的释放（写）。

进程可能通过不同的同步链按不同的次序观察到同一个写，因此大多数软件系统不保证写的顺序化。事实上，根据第5章的定义，像释放同一性这样的基于同步的放松的同一性规范不能保证一致性。最后，释放同一性的硬件和软件 ERC 之间的惟一区别在于何时传播写。然而，LRC 仅在获取点才能传播写提示的特点，使得它的实现在写是否传播这一点上与释放同一性不同，进而产生释放同一性条件下不允许的结果。所以 LRC 是与释放同一性不同的同一性模型，因而要求在编程时更加小心（见例 9.2），而 ERC 则仅仅是释放同一性的一个不同的实现而已。但是，如果适当地对程序加以标记，即如 9.1.2 节所讨论的那样标记所有的同步操作，那么就能保证它在 RC 和 LRC 下都能正确运行，一致性和顺序同一性都能被满足。

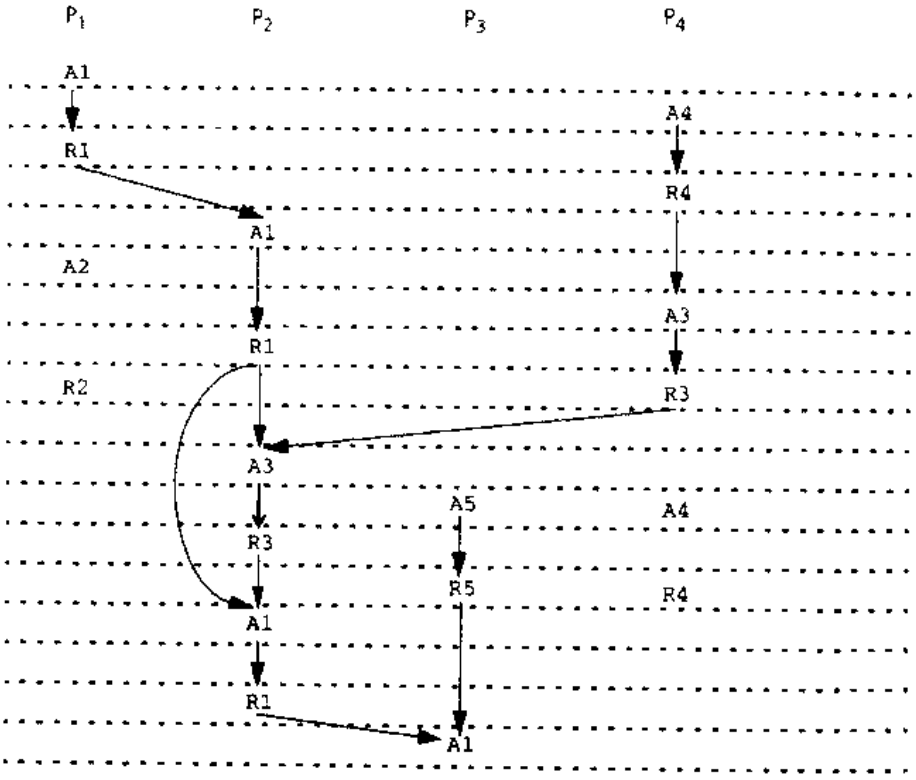


图 9-14　同步操作间的因果次序及这些操作之间的数据访问组．该图说明了在进程 P_3 的获取 A1 和进程 P_2 的释放 R1 之前的同步操作是什么。水平点线是时间增量，向下递增，指明可能的时间上的重叠。粗箭头说明了沿获取-释放链的依赖次序，而灰箭头显示了作为因果次序一部分的程序原序。箭头未涉及 A2 和 R2 及 A4 和 R4 操作对，所以按因果次序它们不在感兴趣的获取之前；因此，在该获取之后的数据访问不一定看到这些操作对之间的数据访问

例 9.2　设计一个例子，使得 LRC 产生与 RC 不同的结果。你如何避免这一问题？

解答：考虑以下代码段，假定指针 ptr 初始为 NULL。

在 RC 和 ERC 下，新的非空指针值保证在 P_1 的 unlock（释放）操作结束前传播给 P_2，这样，P_2 将观察到新的值，并像我们所期望的那样跳出循环。在 LRC 下，P_2 在执行它的 lock（获取操作）之前将不会观察到 P_1 的写；它将因此而进入while循环，而永远看不到 P_1 的写，所以也永远不会退出 while 循环。解决办法是在 while 循环的读之前放置适当的获取同步操作或者把对 ptr 的访问标记为同步，从而生成适当标记的程序。■

715

仅在软件 SVM 层能识别的同步操作处传播一致性信息这一事实具有一个有趣的相关含

```
        P1                                P2
lock L1;
ptr = non_null_ptr_val;
unlock L1;                        while (ptr == null) {};
                                  lock L1;
                                  a = ptr;
                                  unlock L1;
```

义。在采用放松的同一性模型的 SVM 系统上运行现成的应用程序二进制代码是困难的，即使这些二进制代码是针对支持非常放松的同一性模型的系统而编译并且已被适当地标记过。其原因是标记已经被编译成特定的商品化微处理器使用的隔栅指令，而软件 SVM 层可能看不到这些隔栅指令。当然，如果能获得源代码或带有标记的汇编码，就可以把标记转换成 SVM 层能识别的原语；如果只有二进制码，可以使用现有的工具对它编辑，通过插入适当代码而使 SVM 运行时系统能观察到这些标记。

2. 多写入者协议

在前面讨论的例子中，在两个同步点之间，如果只有一个共享者对页面写入（其他共享者可以读这些页面），推迟写提示的方法对缓解伪共享效应非常有效。但是，它不能解决多写入者的问题。试考察图 9-15 中修改过的例子。现在，P_0 和 P_1 两者都在相同两个栅障之间修改同一个页面。如果我们遵循只允许单一写入者的协议的话，那么每个写入者在写之前必须获得页面的所有权，导致同步点间的往返通信，牺牲了（允许多个写入者同时存在的）放松同一性模型的潜在效益。为了真正发掘放松同一性的效益，我们需要多写入者协议。这种协议允许各处理器在同步点间对页面写入，以便在本地修改它自己的副本使副本变得不一致，然后按同一性模型所要求的那样，在下一个同步点使这些副本一致。让我们简要地了解一下几种多写入者机制，它们可用于积极或惰性释放同一性。

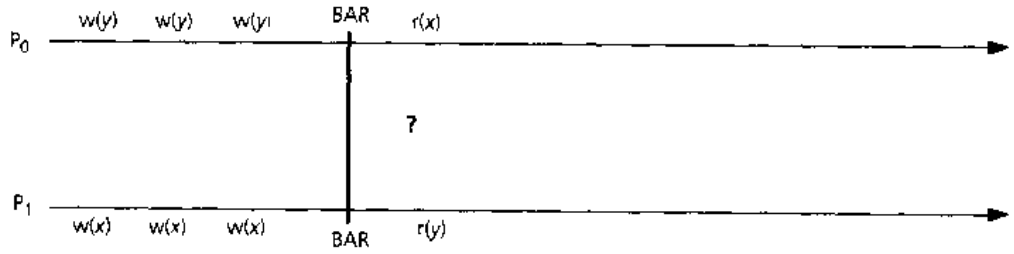


图 9-15　多写入者问题。在栅障点，两个不同的处理器已经分别对相同页面写入过，而它们的修改需要合并

第一种办法是 Rice 大学的 TreadMarks SVM 系统所采用的（Keleher 等 1994）。它的概念相当简单。为了捕获对共享页面的修改，页面被初始化成写保护的。同步点后第一个写产生一个保护违规。此刻，系统用软件生成该页面的一个副本（称为孪生），然后解除实际页面的保护，这样后续的写不会再产生保护违规。以后，在该进程的下一个释放或获取到来时（分别对于 ERC 和 LRC），孪生和当前副本进行比较，生成一个“差异”记录，它不过是两个副本之间不同之处的压缩编码的表示。所以该差异记录捕获了处理器在同步区间对页面所做的修改。当一个处理器产生缺页时，它必须从生成该页的差异记录的其他处理器获得差异并把它与它自己的副本合并。与写提示情况类似，关于何时计算差异，何时将差异记录以及页面

的副本传播给其他处理器存在几种途径（见习题 9.23）。如果在释放点积极地传播差异记录，这些记录和写提示可以立即释放，存储空间可以再利用。在惰性实现情况，差异和写提示在被请求之前可以一直由其生成者保存。在这种情况下，它们必须被一直保持，直到确认再没有其他处理器需要它们为止。因为这些差异和写提示所需的存储空间可能很大，因此需要废料收集（例如，通过强制性地传播差异和写提示）。废料收集的算法相当复杂而昂贵，因为当调用它时，每个页面都可能有未收集的差异散布在许多节点上（Keleher et al. 1994）。

另一种软件的多写入者的方法在仍然实现 LRC 的前提下避免废料收集，它采取一组不同的性能折中（Iftode, Singh, and Li 1996b; Zhou, Iftode, and Li 1996）。其概念是，既不是在需要之前由写入者保持差异，也不是在释放点把差异传播给所有的副本，而是取其折中。与扁平的硬件一致性方案一样，每个页面有一个宿主节点，在释放点把差异传播给宿主。然后执行释放的处理器在将差异传播给宿主后，可以立即释放所占用的存储空间。到达宿主的差异记录和对应的页面合并（差异在那里所占的存储空间也可以释放），所以宿主的页面保持最新。执行获取的处理器像以前那样从先前的释放者得到写提示。但是，当它们针对这些页中的某一页产生后续的缺页时，它并不从所有先导的写入者得到差异记录，而是从该页的宿主取回整个页面。我们称这为*基于宿主的协议*。它除了具有低得多的存储开销和更好的存储可扩展性外，还具有性能上的优点，即在缺页时只需要一个往返的消息来取得数据，而在前面的方案中，必须从所有的先导的（"多个"）写入者获得差异记录。另外，处理器从来不会为以其本身为宿主的页面产生缺页。其缺点是必须取整个页面而不仅仅是差异记录（虽然，我们也可以把差异记录存放在宿主，但不与页面合并，然后从宿主取差异记录来折中存储容量和协议处理开销），而且由于性能原因，尽管页面在主存中有复制，它们在宿主的分布变得重要了。哪个方案的性能更好取决于应用的共享模式如何以页面粒度呈现，也取决于通信体系结构的性能特征。

716
/
717

3. *其他传播写的办法*

差异处理（包括孪生页的生成、差异的计算和差异的应用）会引起显著的开销，需要相当大的额外存储空间，而且会由于替换掉有用的应用数据而污染第一级处理器高速缓存。某些近期的系统在网络接口中对细粒度通信，特别是远地处理器细粒度的写传播提供了硬件支持，这可以被用来加速那些基于宿主的多写入者 SVM 协议，避免使用差异。对于写的硬件传播的概念起源于 PRAM（Lipton and Sandberg 1988）和 PLUS（Bisiani and Ravishankar 1990）系统；现代的例子包括普林斯顿大学的 SHRIMP 多计算机原型的网络接口（Blumrich et al. 1994）和 Digital Equipment Corporation 的存储器通道（Gillett, Collins, and Pimm 1996）。

这些网络接口允许在不同节点的页对之间建立映射，这样对源页面的写由硬件传播到目的页面。写的检测可以通过监听存储器总线（如 SHRIMP 中称为*自动更新机制*）或通过对写操作进行软件修改，产生对不同地址空间的特殊的写（在存储器通道中称为*双写机制*）。然后，网络接口根据映射传播检测到的写，网络接口甚至可以位于 I/O 总线上。侦听的方案要求对高速缓存直写，而后者会有额外的指令开销，需要代码的调整。通过建立页面副本和宿主拷贝的映射关系（当这些副本首次建立时），写可以被传播到宿主，使宿主根据同一性模型而保持最新（见图 9-16）。与前面讨论的完全一样，像写提示的传播和实施这样的同一性动作是在同步点进行的，缺页时从宿主取回完整的页面。利用这些特性，人们开发了*基于宿主的协议*（Iftode et al. 1996; Iftode, Singh, and Li 1996a; Kontothanassis and Scott 1996），事实

上，正是受它们启发产生了全软件的基于宿主的协议。这些细粒度写传播的方法从根本上避免了差异的使用，但是，它们要求硬件的支持，而且由于传播所有的写，而不是仅仅在同步的末尾传播最后的新值，从而增加了数据的流量。 718

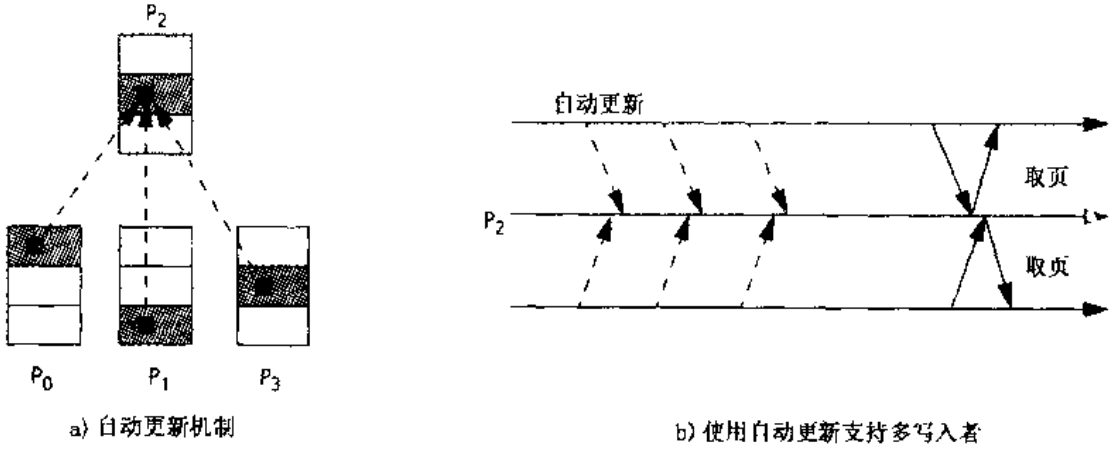


图 9-16 使用自动更新机制来解决多写入者问题，变量 x 和 y 落入同一页，节点 P_2 是该页的宿主。如果 P_0（或 P_1）是宿主，它就不需要传播自动的更新，也不会在该页上发生缺页（只有其他节点会发生）

另一种计算差异的全软件的方法用软件对主存的每一个字或块保存一个脏位（Zekauskas，Sawdon，and Bershad 1994）。一个字的脏位记录了自从脏位最后一次被清除以来，该字是否已被本地节点写入过。脏位在同步点被清除，当到达同步点时，一页中那些发现被置位的脏位等价于该页的差异记录。尽管差异的决定不需要页面与其孪生页的比较，但脏位的置位和复位需要额外的指令和代码调整，这和存储器通道接口的写传播所需的类似。可以用软件分析来降低开销，但开销仍然是相当大的。

至此，我们的讨论集中于不同的惰性程度的功能，但没有提及实现。我们如何保证满足因果偏序的写提示能在正确的时间到达正确的位置？如何降低写提示传输的次数？有许多方法和机制可用于实现不同形式的释放同一性协议（单一或多写入者、基于获取或基于释放以及实际实现的惰性的程度）。机制和各种机制支持的惰性的形式以及它们之间的折中是有趣的，第 9 章 9.6.2 节将对它们进行讨论。

4. 总结：一个读操作的路径

为了归纳一个 SVM 系统的行为，让我们观察一个读的路径。我们考察基于宿主的系统的行为，因为它比较简单。读访问首先在处理器的存储器管理部件经历了从虚地址到物理地 719
址的地址转换。如果发现了本地的页面映射，就查找高速缓存层次，这和一般的单处理器的操作完全一样。如果找不到本地的页面映射，就会发生缺页，然后建立映射，提供物理地址。如果操作系统指示该页当前不存在于任何其他节点，那么从磁盘调入该页，使其具有读写允许权；否则，从其他节点得到具有只读允许的页。此时查找高速缓存。如果在本地存储器和高速缓存层次之间保持包容关系，这是为了使高速缓存与本地存储器保持一致，访问就会扑空，要从主存中页面映射的位置将块调入高速缓存。注意，包容意味着当一个页面被从本地存储器作废，或从读写模式降级为只读模式，或被替换时，要将它从高速缓存腾空（或者改变高速缓存块的状态）。如果对一个只读页发出写访问，那么也会发生缺页，在高速缓存能够满足访问之前，要得到该页的拥有权。

5. 对性能隐含的意义

与顺序同一性实现相比，惰性释放同一性和多写入者协议大大改善了 SVM 系统的性能。但是，与采用硬件以高速缓存块的粒度管理一致性的机器相比，仍存在许多性能问题。伪共享和额外的通信或协议处理开销并不随放松的模型而消失，满足缺页和取页的代价仍然是昂贵的。通信的高代价和由较高的端点处理开销引起的竞争经常严重地扩大了处理器之间通信量和执行时间的不平衡。SVM 系统的另一个问题在于同步是由软件通过显式的软件消息完成的，因而代价非常大。在临界区中经常发生昂贵的页扑空这一事实使问题更加严重，因为这人为地扩大了临界区，从而大大提高了临界区的串行性。其结果是，尽管具有粗粒度数据访问和同步模式（很少的伪共享或通信碎片）的应用在 SVM 系统上运行得很好，但具有较细粒度访问模式（即来自不同进程的对共享虚地址空间的访问以细粒度重叠），特别是较细粒度同步的应用并不能很好地从硬件高速缓存一致性系统迁移到 SVM 系统，除非对它们进行加工（Jiang, Shan, and Singh 1997）。SVM 系统在性能和运行大问题的能力方面的可伸缩性也不确定，因为辅助的数据结构的存储开销随着处理器的数量而上升。

总的来说，我们还不清楚是否能重新构造在硬件一致性机器上运行良好的细粒度应用，使其能有效地在 SVM 系统上运行，也不清楚 SVM 系统是否对宽范围的应用可行。人们正在开展研究，以便理解性能问题和瓶颈（Dwarkadas et al. 1993; Iftode, Singh, and Li 1996a; Kontothanassis et al. 1997; Jiang, Shan, and Singh 1997）以及在仍然用软件维持页粒度的一致性同时，增加某些硬件支持细粒度通信的价值（Kontothanassis and Scott 1996; Iftode, Singh, and Li 1996a; Bilas, Iftode, and Singh 1998）。随着低成本的 SMP 的流行，还有许多研究正在扩展 SVM 协议，以便建造具有二级层次的一致共享地址空间的机器：在 SMP 节点内部是硬件一致性，而在 SMP 节点之间是软件 SVM 一致性（Erlichson et al. 1996; Stets et al. 1997; Samanta et al. 1998）。其目标是尽可能少地调用外层的 SVM 协议，只有当需要跨节点一致性时才调用它，同时仍然保留节点内部的惰性。向高速缓存一致的分布式存储器节点而不是向 SMP 节点的延伸是自然的事情。事实上，像 SVM 这样便宜但性能低的软件共享存储器方法并不是对单处理器节点之间硬件一致性的替代，它可以被看作是一种一致共享地址空间编程模型的扩展方法，即从硬件一致的多处理器节点到这样的节点的群，从而构造大规模的系统。让我们来考察其他一些软件方案。

720

9.3.4 语言和编译器支持的访问控制

我们还可以谋求语言和编译器对一致复制的支持。一个方案是根据数据对象或数据的“区域”来编程，让管理这些对象的运行时系统以对象粒度提供访问控制和一致的复制。通过明确地使用对象，该方法不是把存储器看作扁平的地址空间。下面将看到，这种“共享对象空间”的编程模型启发我们使用更为放松的存储同一性模型。下面还将简要讨论基于编译器的一致性以及用软件提供共享地址空间但不提供自动复制和一致性的方法。

1. 基于对象的一致性

释放同一性模型利用了存储同一性的“时间”（when）维，它告诉我们什么时候需要将一个进程的写入由其他进程执行。正如我们讨论过的，这一点允许我们开发一系列懒惰式的实现，它们尽可能久地推迟写的执行。但是，即使采用释放同一性，如果程序中的同步要求进程 P_1 的写在某一点之前对于进程 P_2 执行或为 P_2 所见，这意味着进程 P_1 对数据的所有写

入都为外界所见，即使 P_2 并不需要看到这些数据（见图 9-17）。更为放松的同一性模型还考虑同一性的“什么”（what）维，就是说，仅仅对执行获取同步的进程根据同步的因果关系而真正需要看到的那些数据传播作废和更新。程序员通过在程序中插入的同步说明释放同一性的时间维。问题是如何说明“什么”维。一种可能是给同步事件（或变量）附加一组页面，这些页面对于该事件必须保证一致。但是，程序员很难完成这个任务。 721

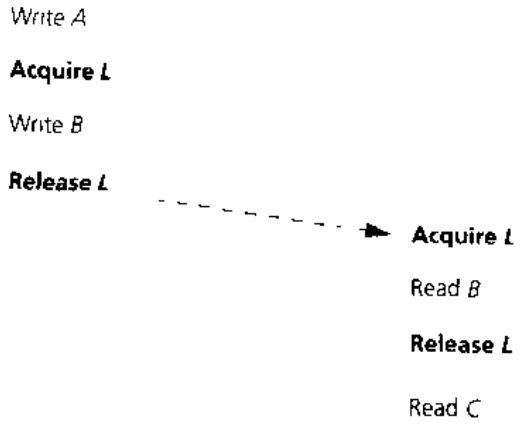


图 9-17 为什么释放同一性是保守的。假定 A 和 B 位于不同的页。因为 P_1 在释放 L 之前对 A 和 B 写入（即使不是在临界区内写 A），对 A 和 B 的作废将被传播到 P_2。而且施加到那里的 A 和 B 的副本上。但是，P_2 仅需要看到对 B 的写，不需要了解对 A 的写。现在假定 P_2 读与 A 处于同一页的另一个变量 C。因为包含 A 的页已经被作废了，P_2 在访问时将产生一个由伪共享所致的页扑空。如果以某种办法把包含 B 的页与锁 L 相联系，那么在 P_2 获取 L 时只会传播 B 的作废，而伪共享引起的扑空就不会发生了

基于域或基于对象的方法提供了更好的解决方案。程序员把数据分割成逻辑对象或区域（区域是任意的、由用户说明的虚地址的范围，它被当作对象处理但不需要面向对象的程序设计）。运行时库以这些区域或对象为粒度维护一致性，而不是把以页面粒度的一致性维护工作全部留给操作系统来完成。这一途径的缺点在于增加了适当说明和使用区域或对象的编程负担，而且在应用和操作系统之间需要复杂的运行时系统。主要优点是 1）逻辑对象（而不是机器提供的固定的一致性粒度）本身有助于降低伪共享和碎片；2）它们提供了逻辑地说明数据的手段，放松了利用“什么”维的同一性。

例如，在入口同一性（entry consistency）模型中（Bershad，Zekauskas，and Sawdon 1993），程序员将一组数据（区域或对象）与程序中每一个像锁或栅障这样的同步变量或每一个同步事件相联系（付出一定代价，这种联系或绑定可以在运行时改变）。在同步点，只保证那些与同步变量相联的对象或区域的一致性。对其他修改过的数据的写提示不必传播和应用。如果未对同步变量说明绑定，就使用缺省的释放同一性模型。但是，绑定不仅是提示性的：如果说明，就必须是完整和正确的，否则就会得到错误的程序结果。清楚正确的绑定的需求给程序员增加了相当大的负担，而且该模型也还没有显示出显著的性能增益来证明它是值得的。程序设计语言 Jade 虽然用不同的方法说明数据，但达到了类似的效果（Rinard，Scales，and Lam 1993）。最后，人们还使用一种称为范围同一性（scope consistency）的模型，试图发掘同步和数据之间基于页的虚拟存储器方式的隐式结合的效果（Iftode，Singh，and Li 1996b）。 722

2. 基于编译器的一致性

研究集中于利用处理器系统额外的硬件支持，以编译器保持共享地址空间高速缓存的一致。这类方法依赖编译器（或程序员）识别并行循环。一种简单的一致性方案是在每个并行循环的结尾插入一个栅障，而在循环之间腾空高速缓存。但是，它不允许高速缓存利用跨循环的数据局部性。即使仅腾空共享的数据，那些在共享地址空间内声明的，但没有实际共享

的数据也不必要地被腾空了。人们提出了更复杂的方案，这些方案要求对选择性作废的支持和用于记录应作废块的相当复杂的硬件支持（Cheong and Viedenbaum 1990）。除了以非标准硬件和编译器支持一致性之外，这些方案的主要问题在于它们依赖编译器对串行程序的自动并行化，对于实际的程序而言，这不是很成功的。

3. 不带一致复制的共享地址空间

这类系统通过语言和编译器支持一个共享的地址空间抽象，但不带自动复制和一致性，这正是 CRAY T3D 和 T3E 的硬件所完成的。一类例子是像高性能 Fortran（High Performance Fortran）这样的数据并行语言（见第 2 章）。编译器或运行时系统利用用户说明的数据分布和拥有者的计算规则，把节点外的存储器访问翻译成显式的消息、形成较大的消息、调整数据以获得更好的空间局部性等等。复制和一致性的任务通常留给用户，这牺牲了编程的容易性；而另一种方法，系统软件力图在主存中自动管理一致复制。人们也正在对基于 C 和 C++ 的语言做与 HPF 类似的努力（Bodin et al. 1993；Larus，Richards，and Viswanathan 1996）。

723

Split-C 语言采取了一种更为灵活的基于语言和基于编译器的途径（Culler et al. 1993）这里，用户明确说明数组是局部的还是全局的（共享的），而对全局的数组，还要说明它们如何在物理存储器中分布。计算可以独立于数据的分布而分配，对全局数组的访问由编译器或运行时系统根据数据分布转换成消息。计算分配和数据分布的分离使得这种语言比用于平衡不规则程序负载的拥有者计算规则要灵活的多，但是它仍然未支持程序员难以管理的复制和一致性维护。当然，可以很容易地将所有这些软件系统移植到硬件维护一致共享地址空间的机器中，后者隐式地提供了共享地址空间、复制和一致性。在这种情况下，可以用运行时系统管理主存中的复制和一致性，以比高速缓存块更大的单元传输数据，但这些功能并不是必须的。

9.4　综合：分类和简单的 COMA

本章讨论的在存储器层次中管理复制和一致性的途径有几个目标：COMA 通过在主存中复制而改善性能，在 SVM 及上一节中讨论的其他系统则降低系统成本。在一个统一的框架中考察复制和一致性的管理产生了能够挑选现存系统的各个侧面的另外一种系统的设计。一种有用的框架在两个密切相关的坐标上区分不同的途径：

1）在最低层的复制存储器中分配数据，保持数据一致及在节点间交换数据的粒度。

2）在通信辅助部件中使用用超出单处理器系统的额外硬件支持的程度。

这两个坐标是相关的，因为某些功能如无额外硬件支持要么不可能实现细粒度（例如数据在主存中的分配），要么达不到高性能。该框架对于在高速缓存（如 CC-NUMA）或在主存中执行复制两种情况都是适用的。

图 9-18 描绘了整体框架和不同类型系统在其中的位置。我们把粒度分为“页”和“块”（高速缓存块）两种，因为它们在不需要诸如对象这类风格编程模型的透明共享地址空间系统中最流行。框架也能包括诸如单个字的细粒度和诸如对象或存储器区域的较粗粒度。我们将会看到，分配、一致性（访问控制）和通信的粒度相互影响。

在图的左侧是 COMA 系统，它通过额外硬件支持主存中以高速缓存块为粒度的分配。仅在高速缓存中复制数据的CC-NUMA系统，称做纯CC-NUMA，也归于这一类。在按高速缓存

724

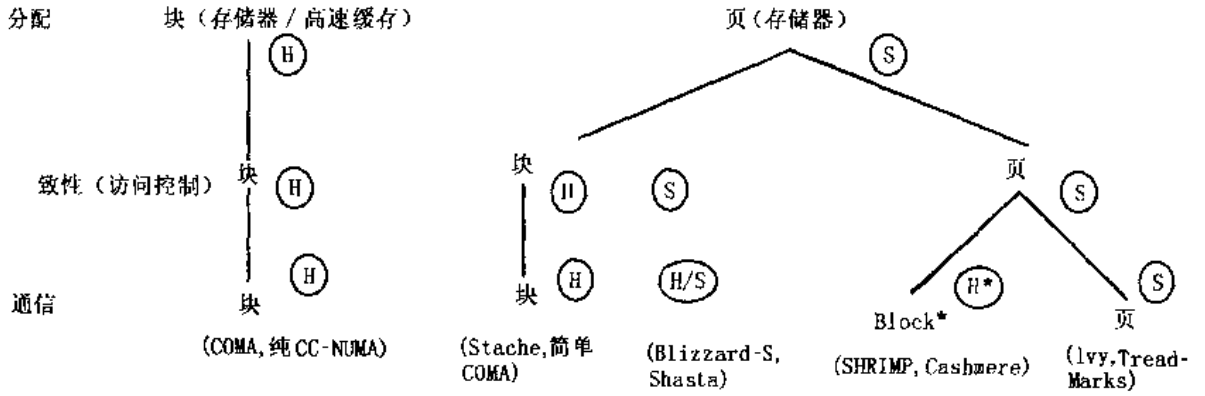


图 9-18　在一致的共享空间系统中分配、一致性和通信的粒度。所说明的粒度是为了当数据被取到一个节点后首次自动复制而规定的。除了纯 CC-NUMA 以高速缓存复制存储外，在所有其他系统中，复制存储的层次是主存。在这个分类的各个叶子之下列出了该类型的某些代表性系统、协议或系统系列，而在各个节点旁边的字母表明该功能是以硬件（H）还是软件（S）支持的。有一处在“块”和“H”旁边加了星号，这表示在该情况下不是所有的通信都以细粒度执行。关于这些系统或系统系列的引用文献包括 COMA（Hagersten, Landin, and Haridi 1992; Stenstrom, Joe, and Gupta 1992; Frank, Burkhardt, and Rothnie 1993），纯 CC-NUMA（Laudon and Lenoski 1997），Stache（Reinhardt, Larus and Wood 1994），简单 COMA（Saulsbury et al. 1995），Blizzard-S（Schoinas et al. 1994），Shasta（Scales, Gharachorloo, and Thekkath 1996），SHRIMP（Blumrich et al. 1994; Iftode et al. 1996），Cashmere（Kontothanassis and Scott 1996），Ivy（Li and Hudak 1989），and TreadMarks（Keleher et al. 1994）

块粒度分配的情况下，至少应以同样粒度保持数据一致并以更细的粒度进行通信[○]。

在图的右侧是在主存中以页面的粒度分配和管理空间的系统，它们不需要额外的硬件完成该功能。这些系统就像 SVM 那样，也可以以页的粒度提供访问控制和一致性，它们可能提供或也可能不提供对细粒度通信的支持。它们可能使用软件调整或硬件支持的方法以更细的块粒度提供访问控制和一致性。一般来说，以细粒度支持一致性的系统也为细粒度的通信提供某种形式的硬件支持。

725

综合：简单的 COMA 和 Stache

尽管 COMA 和 SVM 两者都在主存中复制，分别致力于解决纯 CC-NUMA 存在的部分但不是全部的局限性，但它们处于前面的分类的两个极端。COMA 是硬件密集型的解决方案，它维持细粒度，但是在管理主存方面，产生了一些对于硬件来说是挑战性的问题（如最后一个副本的问题）。SVM 把这些复杂的存储器管理问题留给了系统软件（这简化了硬件，因为通过操作系统虚存到物理存储器的映射，能够像全相关高速缓存那样管理主存），但它的性能却由于大的粒度和一致性的软件开销而大打折扣。图 9-18 的框架产生了一种结合了 SVM 的低成本及硬件简单性和 COMA 的性能优势及易编程性的有趣的方法，即在图的中部所示的简单 COMA（Saulsbury et al. 1995）和 Stache（Reinhardt, Larus, and Wood 1994）方案。这些方案把主存中一致复制的任务分成两部分：存储器管理（地址翻译、分配和替换）和一致性保持（包括复制和通信）。这些方案和 COMA 类似，为达到高性能，通过特殊硬件支持而提供细粒度的一致性，又与SVM相像（因而与COMA不同），它们把存储器管理留给操作系统，

○ 这并不是说细粒度的分配必然意味着细粒度的一致性或通信。例如，即使分配和一致性是细粒度的，为了获得大块数据传输的优点，可能采用粗的通信粒度。但是，我们这里讨论的确实是一般的情况，我们的注意力将集中于这类情况。

以页面粒度完成。让我们先考察简单 COMA。

相对 COMA 来说，简单 COMA 的主要吸引力在于设计的简单。通过虚拟存储器系统执行存储器管理简化了硬件协议，而且还允许使用任意的替换政策对吸引存储器进行全相关的管理。为了提供硬件的细粒度的一致性或访问控制，节点主存的每个页面被划分成任何选定尺寸（比如高速缓存块）的一致性块，并用硬件为每个块保存状态信息。与 COMA 不同之处在于，它不需要标记，因为存在性检测是在页面的级别完成的。与常规情况一样，在访问块之前要检测页允许。当处理器高速缓存扑空时，块的状态的检测与存储器访问并行进行。所以，有两级访问控制：操作系统控制的页面访问控制和在页面访问成功条件下硬件控制的块访问控制。

让我们考虑简单 COMA 相对于 COMA 的性能折中。简单 COMA 降低了被局部吸引存储器满足的访问的时延（希望这是高速缓存扑空时频繁的情况）。它不需要 COMA 所使用的硬件标记比较和选择，事实上也不需要纯 CC-NUMA 机为了决定是否查找本地存储器而要求的本地/远程地址的检查。另一方面，每个高速缓存的扑空都导致对本地吸引存储器的查找，与 COMA 类似，非本地的访问的路径较长。因为在节点的操作系统的独立控制下，一个共享页可能位于不同存储器的不同物理地址；与 COMA 情况不同，不能在扑空时简单地向网络发出块的物理地址并在另一端使用它。这意味着必须支持一个共享的虚拟地址空间，而不是物理地址空间。但是，处理器发出的虚拟地址在检测出吸引存储器扑空时已不存在了。而在那个时刻存在的物理地址必须被反向转换成虚地址或其他全局一致的标识符；该标识符经由网络发送，在另一端的节点转换成物理地址（可能是不同的物理地址）。该进程引起了附加的时延，9.6 节将对其进一步讨论。

726

与 COMA 相比简单 COMA 的另一个缺点是，虽然通信和一致性是细粒度的，分配却是页粒度的。当应用的访问模式不能很好与页面粒度匹配时，这会在主存中产生碎片。如果处理器仅仅访问远程页面的一个字，只有一致性相关的块会被传送，但是，必须在本地存储器中为整个页面分配空间。同样，如果对一个未曾分配的页仅取回其一个块，它可能必须替换掉整个有用的页面（幸运的是，替换是全相关的并处于软件控制之下，因而可能做出复杂的选择）。与此相对照，COMA 系统一般仅为一致性相关的块在吸引存储器中分配空间。所以，简单 COMA 对空间局部性的敏感性比 COMA 更强。

为 Typhoon 系统提出的 Stache 的设计（Reinhardt, Larus, and Wood 1994）采用了一种与简单 COMA 类似的途径，并且在 Typhoon-0 研究原型中得到实现（Reinhardt, Pfile, and Wood 1996）。与简单 COMA 不同，Stache 并不把所有存储器按高速缓存管理，而是在主存中使用前面讨论的第三级高速缓存的方法进行复制；但是，和简单 COMA 一样，Stache 采用软件的页面级管理分配和硬件的细粒度一致性管理。Typhoon 系统的通信辅助部件是可编程的，物理地址被反向转换成虚地址而不是全局标识符，因而能在用户级软件中编写协议的处理程序（见 9.6 节）。人们还提出了一些综合了 CC-NUMA 和简单 COMA 优点的设计（Falsafi and Wood 1997）。

总结：一个读访问的路径

让我们考虑简单 COMA 中一个读访问的路径。处理器的存储器管理部件首先把虚地址转换为物理地址。如果发生缺页，必须为新页分配空间，虽然该页的数据尚未装入。虚存系统决定替换哪个页并建立新的映射。为保持相容性，高速缓存中被替换页的数据必须被腾空或

作废。新映射页的所有块的状态置为无效。然后，使用物理地址查找高速缓存各层次。如果命中（在缺页发生情况是不可能命中的），访问被满足；如果扑空，查找本地吸引存储器，现在地址一定对应到那个页了。如果所需的块处于有效状态，访问结束；如果非有效，再把物理地址转换回全局标识符，该标识符起到虚地址的作用，它在目录式一致性协议的指导下被发送到网络上。远地的节点把这个全局标识符转换成本地的物理地址，利用该物理地址找出其存储器层次中的块并把该块发回请求的节点。然后，把该块装入本地吸引存储器和高速缓存，把数据传递给处理器。 727

图 9-19 总结了使用硬件支持保持细粒度一致性的方案的节点结构。

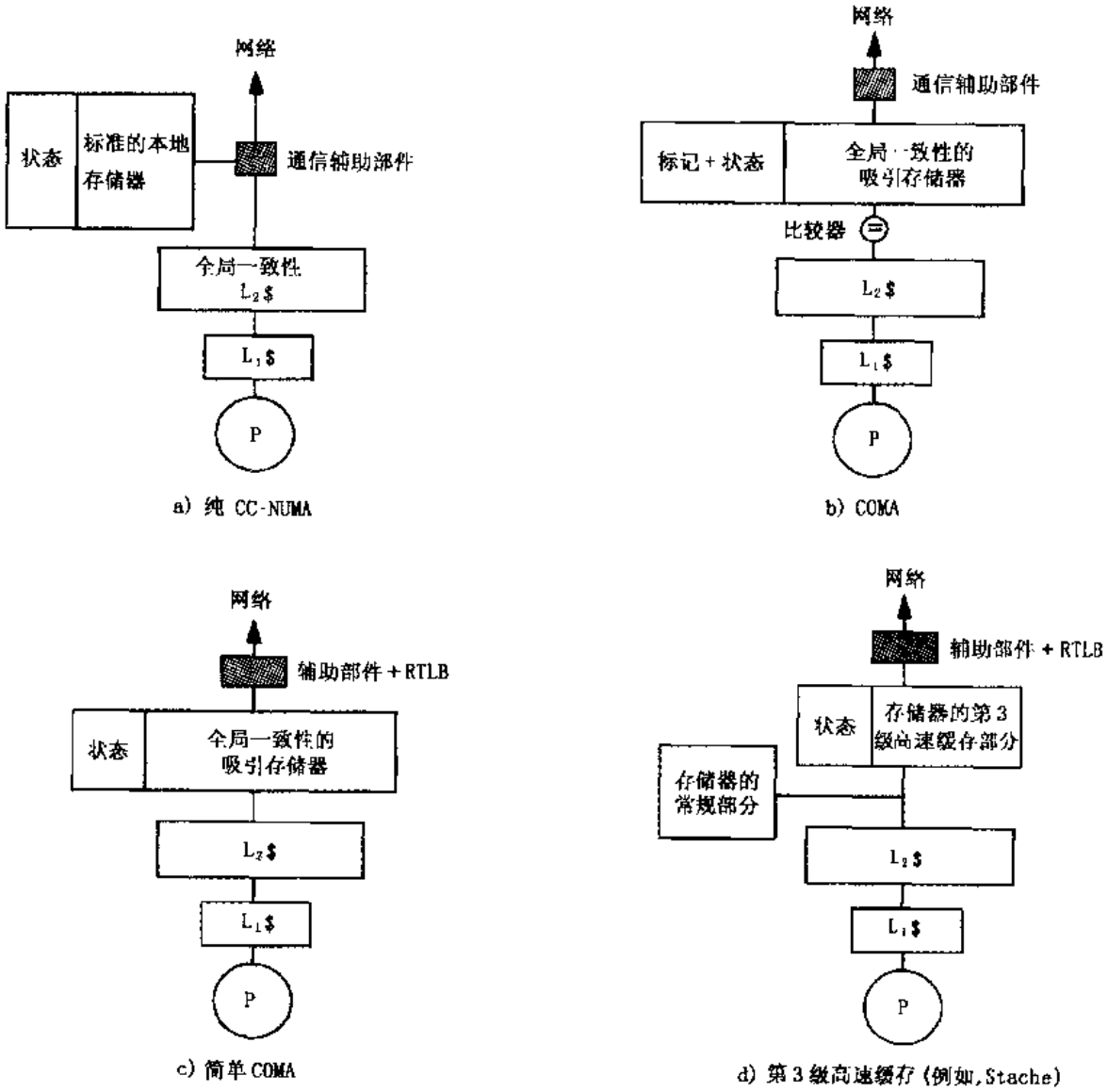


图 9-19 一致共享地址空间的四种方案的典型节点逻辑结构。以硬件维护细粒度的一致性是这些方案的公共特性。图中跨越节点的垂直路径描绘了必须被远地满足的读扑空的路径（通过主存指的是必须查找主存，虽然在使用推测查找情况下，主存查找可以与远程请求的发出并行进行）。RTLB 代表反向翻译快查缓存，它是提供物理地址到虚地址（或到全局标识符）反向转换的机构 728

9.5 对并行软件的影响

现在来考察一下除了在前面几章已经讨论过的并行编程问题外，本章所讨论的所有方案对于并行软件来说还意味着什么。

放松的存储同一性模型要求并行程序把希望的冲突访问标记为同步操作。通常，插入点是相当有特点的，比如，寻找 while 循环推进之前进程踏步等待的变量；但有些时候，即使没有踏步等待，也必须保持次序，正如图 9-4 中的某些例子那样。程序设计语言可以提供某种支持手段，把变量或访问标记为同步，然后由编译器将它们翻译成适当的保持次序的指令。但当前的程序设计语言并不提供对这种加标记的一体化的支持，而是依赖程序员插入特殊的指令或对同步库函数的调用。编译器本身也可以利用标记来限制它自己对共享存储器的乱序访问。一种机制是把某些（或全部）共享变量说明为“易失（Volatile）”类型，这意味着这些变量将不会在寄存器中分配，并且不会改变它们相对于周围的访问的次序。我们还记得共享变量的寄存器分配可能导致违犯一致性，所以为了保证像 flag 这样的关键共享变量本身的一致性，甚至在顺序同一性情况下，也通常将它们说明为易失的。有些新的编译器也能识别显式的同步调用，因而不改变跨越这类调用的存储器操作的次序（或许区别获取和释放调用），或者遵循程序中插入的保持次序的特殊指令所规定的次序

COMA 机所提供的自动的细粒度复制和迁移允许程序员忽略主存的分布式的性质。当数据访问是细粒度的并且主要产生容量型或冲突型的扑空时，这是非常有用的。并行应用的经验表明，由于现代系统的工作集的性质和高速缓存的尺寸等原因，迁移特性可能比复制和一致性用途更广，这就是说，由于在多个主（吸引）存储器中存在一个块的多个副本而获益的情况并不经常发生，通常更有效的方法是把数据的单个拷贝取到合适的吸引存储器里以供一段时间的计算所用。当然，那些仅有小尺寸高速缓存的系统或具有大的非结构化工作集的应用也能从主存复制受益。当我们不能以页粒度合适地分布程序的数据结构时，细粒度自动迁移特别有用，而 Origin2000 提供的页面迁移技术，甚至显式的页面迁移或替换就可能不那么有用；例如，当应该分配到两个不同节点的数据落到同一页的情况时（见第 8 章 8.9 节）。与进程迁移相联系的数据迁移也很有用，虽然页粒度的数据迁移也能很好地应付这种情况。

[729] 一般来说，虽然 COMA 系统的性能受高通信时延的影响要甚于 CC-NUMA 系统，但它们只需较小的编程努力就能使范围更宽的工作负载工作得不错。有趣的是，在扁平 COMA 系统中，尽管其具有 COMA 的性质，但显式的迁移和适当的数据（宿主）替换的作用仍相当有限。其原因是，写入时的拥有权请求仍然要访问目录，而目录可能在远地，而且不随数据迁移，所以即使只有单个处理器对页面写入，也会产生额外的流量和竞争。

以市售产品构成的系统更依赖于软件，不仅为了降低通信的量，也为了仔细地协调数据访问和通信，因为通信的代价或粒度要大得多。让我们考虑以页粒度执行通信和一致性的共享虚拟存储器的情况。实际的数据访问和共享模式与这种粒度相互影响，产生一种页粒度的诱导共享模式，该模式与系统相关（Iftode, Singh, and Li 1996a）。通信对计算的高比率并不是影响性能的主要因素，而是当这种诱导模式包含写共享（因而存在对同一页的多个写入者）或者包含通信碎片时对性能有坏的影响。因此，尽力安排程序结构，使来自不同进程的访问不在地址空间中以细粒度交错就变得重要了。同步的高代价以及由于在临界区内的缺页造成的临界区的扩张，使得在 SVM 系统的编程中减少同步更是特别重要。最后，通信和同

步的高代价使我们难以成功地采用任务窃取来实现 SVM 系统的动态负载平衡。窃取所需要的远程访问和同步可能是如此昂贵，以至于当窃取成功时已没有多少工作可被窃取了。所以，就基于任务窃取的计算而言，良好均衡的初始任务分配对于 SVM 系统要比对于硬件一致性系统重要得多（Jiang, Shan, and Singh 1997）。一般来说，不同的编程模式和算法优化的重要性取决于设计的系统的通信代价和粒度。

9.6 高级论题

在本章结束之前，让我们讨论另外两个论题：传统 CC-NUMA 方法的其他局限性和以软件利用放松的存储同一性模型的机制和技术，例如，共享虚拟存储器协议。

9.6.1 灵活性和 CC-NUMA 系统中的地址约束

传统的纯 CC-NUMA 系统的另外两个局限是：仅有单个一致性协议以硬接线的方式固化在机器中，以及对共享物理地址空间可寻址性的潜在限制。让我们对它们分别加以讨论。

1. 提供灵活性

不可能有适应一切的方案。我们总是可以发现这样一些工作负载，它们要求有不同于固化在机器中的协议才能工作得更好。例如，尽管我们已经看到对于高速缓存一致的系统，基于作废的协议一般优于基于更新的协议，但是对于单纯的生产者－消费者共享模式而言，基于更新的协议更有优势，对于单字形式的生产者－消费者交互，在基于作废的协议的情况，生产者会产生一个被应答的作废，然后消费者将产生一个生产者将满足它的读扑空，这导致四次网络传送事务；而基于更新的协议在这种情况下仅需要一次。另一个例子是，如果大量能预先决定的数据要从一个节点传到另一个节点，那么用单个大的显式的消息传送要比针对取和存扑空一次传送一个高速缓存块更有利。单一的协议甚至不能最佳地适用于单一应用的所有阶段或所有的数据结构。如果在不同场合使用不同协议的性能优点是显著的，在通信体系结构中支持多个协议可能是有用的。当不匹配的协议造成的性能损失很大时，例如那些有着低效率的通信体系结构的基于市售产品的系统，这一点就格外需要。

730

使通信辅助部件的协议处理部分是可编程的而不是硬接线的，即用软件而不是硬件实现协议，可以允许协议被改变（或者说混合和匹配）。在一致性协议的软件实现中，这显然是自然的事，因为在这里，协议包容在可编程的软件处理程序之中。但对于硬件支持的、细粒度一致性，可以发现不同协议对可编程的辅助部件的要求通常是十分类似的。就控制而言，它们都需要根据事务类型向协议处理程序的快速转移，也需要对标记的位域高效处理的支持。就数据而言，它们需要控制器和网络接口之间高带宽、低开销的数据流水传送。第 8 章所讨论的 Sequent NUMA-Q 提供了一个流水线式的专用可编程一致性控制器，正如 Stanford FLASH 的设计所做的那样（Kuskin et al. 1994; Heinrich et al. 1994）。注意，可编程的控制器并没有改变对一致性特殊硬件支持的需求；这些问题仍然和固定协议时一样。

在这些细粒度一致性机器的控制器上运行的协议代码工作于优先模式，这样它能通信并使用它直接从总线上看到的物理地址。某些研究者还力图证明，如果允许用户在用户级编写它们自己的协议处理程序，它们能够修饰协议，使其比预先决定的系统协议库更好地适应不同应用的需求（Falsafi et al. 1994）。尽管这可能是有优势的，特别是对那些通信结构效率较低的机器而言，但它引入了复杂性。例如，因为在检测到高速缓存扑空之前，处理器的存储

器管理部件已经完成了地址翻译，通信辅助部件只能看到总线上的物理地址。但是，为了维持保护不允许用户级协议软件访问这些物理地址。这样，如果另一端的辅助部件的协议软件要运行在用户级，在软件使用物理地址之前，必须将它们转换回虚拟地址。这种反向翻译（简单 COMA 为了不同的原因也需要它）要求进一步的硬件支持，使时延增加，而且实现复杂。此外，因为协议具有与正确性和死锁相关的难以捉摸的复杂性，因此是难以调试的。我们也不清楚允许用户自己编写可能使共享机器死锁的协议的必要性有多大。

731

2. 克服物理地址空间的限制

在共享物理地址空间，产生请求的辅助部件通过网络发送单元（或高速缓存块）的物理地址，该地址由另一端的辅助部件解释。其优点是另一端的辅助部件在访问物理存储器之前无须对地址做反向翻译，但发生了新的问题，即作为整个共享物理地址空间的全局地址，处理器生成的物理地址的位数可能不够，我们在第 7 章的 CRAY T3D 看到了这个问题（Alpha 21064 处理器仅发出 32 位的物理地址，不足以寻址一个具有 2048 个处理器的机器的 128 GB 物理存储器，这需要 37 位的地址）。在 T3D 中，使用了通过 annex 寄存器的分段来扩展地址空间，从而在存储器访问的临界路径引入延迟。另一种方法不提供共享的物理地址空间，而是经由网络送出虚拟地址，它由另一端转换回（可能不同的）物理地址。正如我们已经看到的，SVM 和简单 COMA 系统使用这个方法实现共享虚拟地址空间，而不是共享物理地址空间，用户级可编程协议也是这样。

这个方法的优点之一是为了索引整个共享（虚拟）地址空间，只要求虚拟地址足够长，物理地址的长度只要足以寻址给定处理器的主存就行。如前所述，该方法的第二个优点是各个节点只管理自己的地址空间，完成虚拟到物理的地址转换，因此可以采用更为灵活的主存分配和替换策略。但是，这个方法要求通信的两端都要进行地址转换。

9.6.2 以软件实现放松的存储同一性

关于利用释放同一性的共享虚拟存储器方案的讨论揭示：这样的方案可以支持单个或多个写入者，能够在释放或获取时传播一致性信息。可以使用多种技术来实现这些方案，逐渐地增加复杂性，产生新的方案，使写提示和数据的传播的惰性更大。这些技术太复杂，如果用硬件实现需要太多的结构，而所缓解的问题的严重性却要低得多。然而它们却很适合于用软件实现。虽然 SVM 当前还不是一种主流的商品化技术，但它们的价值在于帮助我们理解如何保持释放一致性模型所希望的各种不同程度的惰性。本节将考察各种技术及它们之间的折中。

732

SVM 方案对 8.1 节提出的一致性协议的三个基本功能有不同的做法。首先，硬件一致性方案是在数据访问失败时调用协议，而 SVM 方案则在访问失败时（为了找到所需的数据）和同步点（为了在写提示中传送一致性信息）都要调用协议。前两种重要的功能，即找到一致性信息的源和决定与哪个处理器通信，取决于是使用基于释放还是基于获取的同一性。对前一种情况，我们需要在释放点向所有有效的副本发送写提示，所以我们需要一种记住副本的机制。对后一种情况，获取者仅仅与同步变量的最后一个释放者通信，并从那里仅取回所有自己所需要的写提示，因此无须显式地记住所有的副本。最后一个功能是与所需要的节点（或副本）通信，典型的做法是点对点消息。

因为一致性信息不是在每个写失败时传播，而是仅仅在同步事件点传播，这就提出了一个新的问题：我们如何决定必须为哪个页发送写提示？在基于释放的方案中，因为是在每个

释放点对当时所有有效的副本发送写提示，节点仅需要对它上一次释放以来执行的写发送写提示。在因果意义上所有先导的写提示（见 9.3.3 节）此刻已经被发送到所有相关的副本，或者是在对应的先导释放点直接发送，或者经由其他进程间接发送。以基于获取的方法，我们必须保证在获取点能看到已经由许多不同节点生成的，在因果关系上需要的写提示，尽管获取者仅仅到先导的释放者那里获得这些写提示。释放者不能简单地向获取者发送自从它上一次释放以来自己所产生的写提示或它产生过的所有写提示，而是必须还要发送在它自己前一个获取点从其他节点收到的因果相关的写提示。基于释放和基于获取这两种情况都采用了几种机制来减少传送和申请的写提示的数量，包括版本号和时间戳，后面将对它们有所了解。我们还将看到，在何时传播写提示（和数据）及何时申请写提示方面协议有所区别，实现基于获取或释放的不同程度的惰性。

为了更清楚地理解问题，首先考察如何用基于释放和基于获取的方法实现单写入者释放一致性。然后我们将对多写入者的协议做同样的考察。

1. 单写入者基于释放的同一性

维护同一性的最简单的办法是在每个释放点向页面的所有共享者发送写提示，而这些页面是执行释放的处理器自其上一次释放以来已经改写过的。在单写入者协议中，由页面的当前拥有者（拥有写许可权者）维护当前共享表并在拥有权改变时（当另一个节点写这个页时）转交这个表，以此来记住副本。在释放点，一个节点为所有它写过的页面向在它的共享表中记录的节点发送写提示（见习题 9.31）。

733

这个方案存在两个性能问题。首先，因为拥有权的转移并不引起副本的作废（只读副本可以与一个可写副本共存，这与硬件一致性方案不同），前一个拥有者和新的拥有者的共享表中完全可能有同一个节点。当它们两者都到达释放点（针对相同的或不同的同步变量），它们将会对某些相同的页发出作废，这样一个页可能收到多个（不必要的）作废。可以通过指定单一位置记录一个页的哪些副本已经被作废来解决这个问题，例如，可以利用基于存储器的目录记录共享表，而不是由动态变化的拥有者来维护它们。在写提示送出以前要查找目录，已被作废的副本记录在目录中，这样就不会对同一个副本发送多个作废了。

第二个问题是一个释放可能会作废掉一个比被写入过的更新的页副本，而这是不需要的，正如图 9-20 所说明的那样（它不会作废掉当前拥有者的最新的副本，所以这不是正确

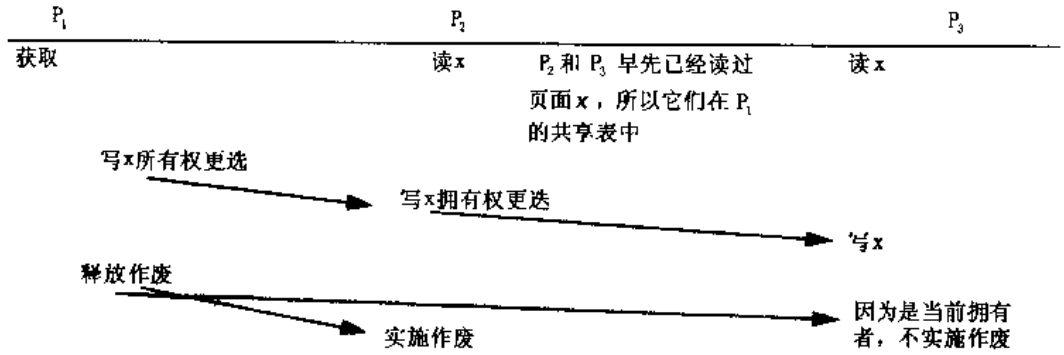


图 9-20　在单写入者协议中作废较新副本的情况。处理器 P_2 从 P_1 得到所有权，然后 P_3 从 P_2 得到拥有权。但是 P_1 的释放晚于所有这些动作才发生。在释放点，P_1 向 P_2 和 P_3 发出作废。P_2 实施这个作废，尽管它的副本比 P_1 的还新（因为它不知道这一点），而 P_3 是当前的拥有者，所以不实施作废动作

性问题)。该问题的解决是给页的每一个副本附加一个版本号。一个节点一旦从另一个节点获得一个页的拥有权后，就对该页的版本号加1。在没有目录的情况下，处理器在释放点将向所有的共享者发送写提示（因为它不知道其他共享者的版本号）以及它自己保持的该页的版本号，但是，只有具有更小的版本号的接收者才会真正作废那个页。在有目录的情况下，在页的目录项中保存副本的版本号，只向那些版本号比释放者的更小的副本发送写提示，这样不仅能减少作废实施和缺页的次数，也能降低写提示的流量。拥有权和写提示请求都进入目录很容易管理。所以目录与版本号结合使用解决上述的两个问题。然而，这仍然是基于释放的方案，因此可能不必要地过早地发送和实施作废，引起不必要的缺页。

2. 单写入者基于获取的同一性

利用获取之前不需要一致性动作这一事实的一个简单的办法是：释放者仍然对所有的副本发送写提示，但只有当来自任何其他进程的下一个获取请求到达时才发送，而不是在释放时就送出。这延迟了写提示的发送。但是这样一来外来的获取请求必须等待释放者送出写提示并且接收到应答，因此，获取操作的临界路径之中就多了这些操作。这种基于释放的基本方案所能做到的最好的事是在释放点就积极地送出写提示，而在响应下一个外来的获取请求之前才等待应答。这样做允许写提示和应答的传播与释放和下一次获取间的计算重叠。不管何时传播写提示（这影响流量），接收的进程可以选择在接到写提示后立即对页面实施，或选择在它执行下一次获取时才实施，从而实现不同程度的惰性。

现在考虑惰性的基于拉的仅在获取点传播一致性信息的办法，而且仅仅从释放者向对应的获取者传播。获取者向一个同步变量的最后一个释放者（即当前保持者）发出一个请求，获取者能从该变量的指定管理节点获知谁是对应的释放者。这是获取者获得信息的惟一地方，它必须能看到在它之前以因果次序发生的所有写操作。如果没有附加的支持，释放者必须向获取者发送到目前为止（至少是从它上一次向这个获取者发送写提示以来，如果它记录这样的信息）它自己已经生成的或者已经从其他地方接收到的所有写提示。释放者不能仅仅发送那些自前一次释放以来它所产生的写提示，因为它根本不知道该获取者已经从其他进程先前的获取处得到了多少其他必要的写提示。获取者也必须保存这些写提示，再传递给下一个获取者。

保存写提示的全部历史记录显然不是一个好主意。如果在发送写提示的同时传送前面讨论过的在拥有权更迭时加1的版本号，能帮助我们减少作废实施（进而访问缺页）的次数，但是他们无助于写提示发送次数的降低。获取者不可能通过向释放者传送版本号来降低流量，因为它不知道释放者想对哪些页发送写提示。带有版本号的目录也无助于流量的降低，因为释放者必须向目录发送写提示的历史。事实上，获取者想要得到的是那些对应于因果关系上先于它的所有释放的写提示，而这些写提示它还没有通过自己先前的获取得到过。保持页面层次的信息达不到这个目的，因为获取者和释放者都不知道对方已经看到了哪些页的写提示，也都不愿意发送它已经知道的信息。解决的办法是建立一个逐进程或逐节点虚拟时间的系统，在该系统中，时间步由同步时间划分。概念上，每个节点负责记住直至它已看到每个其他节点的写提示为止的虚拟区间。获取者向前面的释放者送出这个时间向量；释放者将此向量与它自己的比较，然后向获取者发送释放者已经看到而获取者还没看到的时间区间内发生的写提示。因为因果性所应满足的偏序是基于同步事件的，所以把时间增量与同步事件

734 ~ 735

结合使我们能以明白的方式表示这些偏序。

更精确地说，每个进程的执行被划分为许多区间。当进程成功地执行了一个释放或者一个获取时开始一个新的区间（该进程的局部区间计数器加 1）。不同进程的区间是按照第 9 章 9.3.3 节所讨论的因果优先关系偏序排列的：1）单个进程的区间按程序原序全序排列；2）如果在相同变量的释放-获取操作链上，某区间上 P 的获取先于 Q 的获取，则进程 P 的区间早于进程 Q 的区间。即，区间也依据依赖关系排序，但这或许不是静态能决定的，因为区间编号是由进程本地维护和加 1 的，2）中所描述的偏序并不意味着执行获取的进程的区间号一定要比执行释放的进程的区间号大。它真正指出的或我们必须保证的是如果释放者从进程 X 看到区间 8 的写提示的话，那么该同步变量的下一个（动态的）获取者在被允许结束其获取之前，也必须至少从进程 X 看到了区间 8。为了记录一个进程已经从其他进程看到了哪些区间，从而保持偏序，每个进程为它的每一个区间保持一个向量时间戳（Keleher et al. 1994）。令 V_i^P 为进程 P 的区间 i 的向量时间戳，向量 V_i^P 中的元素个数等于进程的数量。进程 P 自己在 V_i^P 中的项等于 i。对于任何其他的进程 Q 来说，其对应的项指明了按偏序次序，先于进程 P 的区间 i 的进程 Q 的最新的区间。所以，向量时间戳表示了一个进程在进入区间 i 时，其他进程相对于它的最近的区间，而且它已经从这些区间接收到并且实施了写提示（通过前一个获取）。

在一个获取点，进程 P 需要从最后一个释放者 R 处获得（来自任何其他进程）写提示，这些写提示属于 R 在它的释放之前已经看到而获取者还没有在它前一个获取点看到过的区间。这足以保证因果性：对于 P 能够从其他进程看到的任何其他区间，P 必然从按程序原序的前一个获取点就已经看到它们了。所以 P 对 R 送出它当前的向量时间戳（即它的 $i-1$ 区间的时间戳），告诉 R 在这个获取之前，其他进程相对于它的最新区间有哪些。R 将来自 P 的入向量时间戳与自己的向量时间戳逐项比较，然后把所有 R 已经看到（包括在 R 的当前时间戳中）但 P 尚未看到（不在 P 的时间戳中）的区间的写提示搭载在对 P 的应答中送出（这种做法是保守的，因为 R 的当前时间戳中所能看到的区间可能比 R 在相关的释放时刻所看到的区间要多）。因为 P 现在收到了这些写提示，它为以当前的获取开始的区间 i 设置新的向量时间戳，即 R 的向量时间戳和它自己前一个向量时间戳中对应项的最大值。P 现在已按偏序更新。这意味着一个进程必须保持它自己产生或从其他进程收到的写提示，知道没有后续的获取者再需要它们为止。它与基于释放的协议不同之处在于，在基于释放的协议中，释放者一旦向所有的副本送出了写提示后，就可以丢弃它们。而这种方法会导致相当大的存储开销，而且可能需要废料收集技术来保持存储空间。

736

3. 多写入者基于释放的同一性

对于多写入者来说，新的关键问题是来自多个写入者的数据（例如，差异记录）的合并及管理。我们所用的写提示和同一性的协议类型取决于如何管理数据的传播，即差异是由多个写入者维护，还是在释放时传播到固定的宿主去。就基于释放的方案而言，不管哪一种情况，进程只需要针对自从它上一次释放以来执行过的写而向所有的副本发出写提示，然后在下一个外来的获取点等待应答，这和单写入者的情况一样。然而，因为在任何时刻，一个页面都不存在单一的拥有者（写入者），所以我们无法依赖这单一的拥有者来得到共享表的最新的副本。我们必须或者广播写提示，或者使用像目录这样的机制来记住副本。

下一个问题是，在作废引起缺页时，进程如何找到所需的数据（差异）？如果使用基于

宿主的协议来管理多写入者，这个问题很容易处理：能够在宿主找到页或者差异（一个释放在结束之前必须等待差异到达其宿主，这样在相关的获取之后所产生的缺页就一定能看到对应的写）。如果是由写入者管理差异（即以分布的形式），缺页的进程不仅必须知道从哪里得到差异，还要知道以什么次序实施它们。这是因为对同一数据的差异可能在同一进程的不同区间产生，也可能产生于同一个释放和获取因果链上不同进程的区间；当它们到达处理器时，必须按照因果关系所要求的偏序实施它们。差异的位置可以从进入的写提示决定，然而如果没有向量时间戳的话，实施的次序却难以决定。但是，正如我们已经知道的，向量时间戳只有在基于获取的协议中才得到充分的应用。这就是为什么当数据不存在宿主时，简单目录和（积极的）基于释放的多写入者一致性方案是使用更新而不是作废作为写提示：在释放时将差异本身发往共享者。所以，同一性所使用的协议和机制是从基于释放的单写入者情况变化而来。因为释放在结束前要等待应答，所以尽管没有使用向量时间戳，却不存在实施差异的次序问题。它能保证差异精确地按照所希望的次序到达处理器。Munin 系统使用了这类基于更新的、积极的、多写入者协议（Carter，Bennett，and Zwaenepoel 1991）。

737

让我们考虑使用更复杂的同一性的跟踪机制。对多写入者方案而言，版本号没什么用处。我们不能在拥有权交替时更新版本号（因为不存在拥有权交替），而只能在释放点这样做。在这种情况下，版本号不能帮助我们节省写提示，因为释放者刚刚在释放点本身获得页的最新版本号，不可能有另一个节点具有该页的更新的版本号。此外，因为释放者只需要发送从它上一次释放以来它所产生的写提示（或差异），基于更新的积极协议不再需要向量时间戳。（如前所述，在一个基于作废的协议中，如果差异不是在释放点送出而是由释放者保持，那么就需要时间戳来保证按正确的次序运用这些差异。）

4. 多写入者基于获取的同一性

多写入者基于获取的协议的问题和机制与单写入者基于获取的方案非常类似，主要的区别在于如何管理数据。在没有特殊支持的情况下，获取者必须从最后一个释放者得到自从它们上一次交互以来该释放者产生或收到的所有写提示，这样必须维护和交流大量的历史信息。在单写入者的情况下，版本号能帮助降低作废实施的次数，但不能减少传输的次数。（对基于宿主的方案而言，每当差异到达宿主时，版本号加 1，但是释放者在满足下一个外来的获取之前，必须等待，接收这个版本号；在无宿主的情况，必须为每个页指定一个独立的版本管理者，这使问题复杂化。）最好的办法是使用前面提到的向量时间戳。向量时间戳管理基于宿主方案的写提示（一致性信息）的传输；对非基于宿主的方案，它们也管理差异（即数据）的获得和实施的正确次序（即由与写提示一起到来的时间戳所指定的次序）。

为了实现基于获取的 LRC（惰性释放同一性），处理器必须保持很多辅助的数据结构，包括：

- 每个页面在本地存储器有一个以进程索引的数组，它的每一项是从该进程收到的针对该页的写提示或差异的链表。
- 一个独立的由进程索引的数组，其每一项是指向一个区间记录链表的指针。一个进程的项代表了该进程的一些区间，对这些区间，当前进程已经收到了写提示。每个区间记录指向对应的写提示的链表，而每个写提示指向它的区间记录。
- 产生差异的自由池空间。

因为这些数据结构，特别是差异，必须保持一段时间，时间段的长短由运行时建立的偏

序优先次序决定，因此可能限制所能运行的问题的尺寸和该方法的可扩展性。基于宿主的方法有助于降低差异的存储容量，因为在通过释放之后，释放者不必保存差异，上面列出的第一个数组不必保存接收到的差异的链表。我们可以从（Keleher et al. 1994; Zhou, Iftode, and Li 1996）获得这些机制的细节。 738

9.7 结论

本章所讨论的支持共享地址空间一致的复制的不同途径产生了许多有趣的硬件/软件折中的集合。放松存储器模型为了正确地标记程序，增加了应用软件的负担，但它允许编译器实施优化，也允许硬件更多地发掘低层次的并发性。COMA 和相关的系统要求更高的硬件复杂性，但通过降低数据替换的重要性简化了程序员的任务。它们的性能效果很大程度上取决于应用的性质（即主宰节点间通信的是共享扑空还是容量扑空），也取决于远程访问延迟和辅助部件占用增加到什么程度。最后，基于市售产品的方法降低了系统的成本，为用户提供了一个更好的增量式采购的模式，但是为了达到好的性能，通常要付出大的多的编程努力。

对这些不同的途径仍然是有争论的，折中并未展开。尽管放松的存储器模型对编译器而言是非常有用的，但在第 11 章我们将看到，某些现代的处理器选择在硬件/软件接口上实现顺序同一性（或 Intel Pentium Pro 的处理器同一性），以日趋复杂的其他技术获得重叠，隐藏延迟。程序员和系统之间的约定也没有很好地被集成到当前的程序设计语言和编译器之中。就主存中的复制而言，全硬件支持的 COMA 在 KSR1 中得到了实现（Frank, Burkhardt, and Rothnie 1993），但由于成本太高，在现代的系统中它并不流行。然而，商品化的系统开始接受类似于简单 COMA 的方案。

页粒度的 SVM 这样的全软件方案和细粒度的软件访问控制在相对较小规模的系统中对于某些类型的应用显示了良好的性能。因为它们很容易建造和部署，可能被用于某些环境下的工作站机群和 SMP。但是，对于大范围的应用来说，这些系统与全硬件系统相比在可编程性及性能方面还有相当大的差距，这方面的弱点比消息传递更甚，它们的可扩展性也还没显示出来。基于市售产品的方案仍处于研究阶段，它们是否能对于足够大的应用集合成为硬件高速缓存一致的机器的竞争者还有待证明。硬件一致性辅助部件的商品化以及它们在存储器中集成的方法更使这些全软件的方案勉强够格，使它们主要用于不希望购买并行机，而把现有的机器在空闲时当作共享地址空间多处理器使用的场合（或用于开发并行应用的早期版本）。至少在短期内，经济上的原因可能会给全软件解决方案提供某种机会。因为厂商倾向于建造只包含几十个至一百个左右节点的紧耦合系统，用软件一致性层连接这类硬件一致性系统是构造支持一致共享地址空间编程模型的巨型机器的最可行方法。尽管在具有物理分布的存储器的可扩展系统上支持这一编程模型被认为是构造系统的理想的方法，但只有时间才能证明这些不同途径效果如何。 739

习题

9.1 为什么在基于目录的机器中，基于更新的一致性方案相对来说与顺序同一性模型不兼容？

9.2 第 7 章讨论的 Intel Paragon 机每个节点有两个处理单元，其中之一总是以内核模式运行来支持通信。能否把这个处理器当作 Stanford FLASH 或 Sequent NUMA-Q 的可编程通信

辅助部件那样有效使用，以支持硬件的块粒度的高速缓存一致性吗？

9.3 在第8章讨论的Origin协议中，当请求者仍有悬而未决的作废，即对作废的应答尚未到达之前，其他的读请求和写请求可以到达该请求者。

1）对于延迟排他应答，这是否可能？有无问题？对于积极的排他应答呢？如果有问题，解决问题的最简单的办法是什么？

2）假设你确实想允许具有未决作废的请求者处理外来的读和写请求。首先考虑写请求，请构造一个会出问题的例子。（提示：考虑下列情况：P_1 对单元 A 写入，P_2 先对单元 B 然后对 flag 写，P_3 在对 flag 踏步等待之后读单元 B 的值。）你如何既允许对外来的写进行处理同时又保持正确性？作废应答由宿主收集（如 Stanford 的 FLASH 所做的那样）或由请求者收集（如 Origin 协议中那样）哪一种更易于实现上述功能？

3）对外来的读回答2）中的问题。

4）如果你使用基于更新的协议呢？当先前对一个块的写引起的更新尚未完成时，要处理外来的对该块的请求会产生什么复杂性？你如何解决它们？

5）总体上说，你是会选择在作废或更新未决时允许还是拒绝处理外来的请求的做法？

9.4 假定一个块的作废尚未完成时要对它写回，这是否安全？有无问题？如果有问题，你如何解决这个问题？

740

9.5 积极的独占应答对于SC模型是否有用？如果处理器与MIPS 10000不同，本身就允许存储器操作的乱序结束，积极的排他应答是否还有任何用处？

9.6 如果使用积极的而不是延迟的排他应答，在宿主或在请求者处收集应答两者之间的折中是否会改变？

9.7 如果处理器的存储器模型是SC而编译器根据WO改变访问的次序，在程序员接口处的同一性模型是什么？如果编译器是SC并且不改变存储器操作的次序，但处理器实现RMO，程序员接口处的同一性模型又是什么呢？

9.8 除了像本章例9.1所讨论的那样改变存储器操作（读和写）的次序之外，编译器的寄存器分配也能完全消除存储器访问。考虑下列代码段的例子。说明寄存器分配如何违犯了SC。单处理器编译器能这样做吗？你如何在你常用的编译器中避免这一点？

P_1	P_2
A = 1	while (flag == 0);
flag = 1	u = A

9.9 考虑本章中讨论的所有规范，将它们按弱化程度排列，即在模型之间画弧线，从模型 A 到模型 B 的弧线表示 A 比 B 强（即任何在 A 下正确的执行在 B 下也是正确的，但反过来就不一定）。

9.10 PC和TSO哪一个更适合于基于更新的目录协议？为什么？

9.11 你能不像入口同一性那样明确地把数据与同步结合，而描述一种比释放同一性更为放松的系统规范吗？它是否要求超出RC的额外的编程上的注意？

9.12 你能为WO描述一组更为放松的充分条件吗？RC呢？

9.13 使用DEC Alpha提供的隔栅操作和Sparc的RMO同一性规范，说明你需要如何插入这

些操作来保证程序遵循下面每一种模型：RC、WO、PSO、TSO 和 SC。你认为以这种方式，哪一种能有效地实现？哪一种不能，为什么？

9.14 一个写隔栅（write-fence）操作（类似 Alpha 体系结构中的写存储器栅障）在处理器先前所有写操作结束之前，阻塞后续的写操作。一个全隔栅（full fence）在所有先前的存储器操作结束之前阻塞处理器。

1）在下列代码中插入最少的隔栅指令，使其成为顺序同一的，假定如不这样做，系统不保持任何程序原序。当写隔栅能满足要求时不要使用全隔栅。 741

```
ACQUIRE LOCK1
LOAD A
STORE A
RELEASE LOCK1
LOAD B
STORE B
ACQUIRE LOCK1
LOAD C
STORE C
RELEASE LOCK1
```

2）重复 1）的工作以保证释放同一性。

9.15 给定下列代码段，SC 不允许（u，v，w，x）值的哪些组合？在这些情况下，IBM 370、TSO、PSO、PC、RC 和 WO 这些模型能否不插入次序指令或标记而保持 SC 语义吗？（除了在按程序原序先于读的写完成之前不允许读返回该写操作写入的值之外，IBM 370 同一性模型与 TSO 很相像。）如果不可以，插入必要的隔栅指令使其符合 SC。假设在到达该代码段之前，所有变量的值为 0。

1)

P_1	P_2
A = 1	B = 1
u = A	v = B
w = B	x = A

2)

P_1	P_2
A = 1	B = 1
C = 1	C = 2
u = C	v = C
w = B	x = A

9.16 考虑使用释放同一性的以基于存储器的目录协议连接基于侦听的 SMP 的两级一致性协议。在对于一个存储器块的写的作废应答尚未到达时，能够安全地把数据提交给另一处理器吗？设另一处理器位于 1）同一 SMP 节点，或 2）不同的 SMP 节点。验证你的答案并说明任何假设条件。

9.17 一个没有适当标记的程序能在一个支持释放同一性的系统中正确运行吗？如果能，如何做？如果不能，为什么？ 742

9.18 为什么 Sun Sparc V9 规范中存储器栅障指令有 4 个风格位？为什么不能只使用两位，用其中之一等待所有前面的写结束，而用另一位等待前面的读结束呢？

9.19 为了与硬件交换标记信息（即存储器操作被标记为获取或释放），有两种选择：其一是将标记与存储单元的地址结合；另一种是将标记与代码中特殊的操作相结合。它们之间的折中是什么？

9.20 P_1 和 P_2 这两个处理器在顺序同一性（SC）和释放同一性（RC）模型下执行以下代码段。

P_1	P_2
LOCK (L1)	LOCK (L1)
A = 1	x = A
B = 2	y = B
UNLOCK (L1)	x1 = A
	UNLOCK (L1)
	x2 = B

假设某一种体系结构，读和写扑空都需要 100 个周期完成。但是，你能假设允许在同一性模型下重叠的访问确实完全重叠。从其他处理器获取一个空闲的锁或释放一个锁花费 100 个周期，来自同一处理器的加锁和解锁操作不能重叠。再假设所有的变量和锁初始没在高速缓存中，所有的锁初始都未锁定，所有存储器单元初始化为 0，所有存储器单元地址都不同并映射到高速缓存的不同索引（即不同的高速缓存行）。

1）在 SC 下 x 和 y 可能的输出是什么？在 RC 下呢？

2）假定 P_1 先获得锁。在 P_1 的锁定操作开始后多久 P_2 能完成它所有的操作，同时满足第 5 章所述 SC 的充分条件？如果满足本章所述的 RC 的充分条件又会如何呢？

9.21 给定下列代码段，我们希望计算在各种存储器同一性模型下它的执行时间。假设具有任意深的写缓存的处理器体系结构。所有的指令均占一个周期，忽略存储器系统的影响。读和写扑空都需要 100 个周期来完成（即全局执行）。锁可进高速缓存，取是非阻塞的。假设所有变量和锁初始都不在高速缓存内，所有的锁都未锁定。进一步假设一个行一旦被装入高速缓存，它在代码执行期间不会被作废。另外，访问的所有存储器单元都是可区别的，它们映射到不同的高速缓存行。

743

1）如果保持顺序同一性的充分条件，执行这段代码将花费多少周期？

2）对弱序重复 1）的工作。

3）对释放同一性重复 1）的工作。

```
LOAD A
STORE B
LOCK (L1)
STORE C
LOAD D
UNLOCK (L1)
LOAD E
STORE F
```

9.22 在积极进取动态调度的单指令发出处理器上执行下列代码。处理器能够有多个未决操作，高速缓存允许多个未决的扑空，写缓存能隐藏存储的延迟（当然，这些特性只有在存储器同一性模型允许条件下才能使用）

假设锁不缓存，能在发出请求 50 个周期后或释放结束后 20 个周期获得（取两者较晚的时刻）。释放的完成需要 50 个周期。读命中花费一个周期完成，写花费一个周

```
处理器1:                          处理器2:
sendSpecial(int value) {        receiveSpecial() {
  A = 1                           LOCK(L);
  LOCK(L);                        if (READY) {
  C = D*3;                          D = C+1;
  E = F*10                          F = E*G;
  G = value;                      }
  READY = 1;                      UNLOCK(L);
  UNLOCK(L);                    }
}
```

期放入写缓冲区。读共享变量的读扑空需要 50 个周期来完成，写的完成需要 50 个周期。处理器的写缓冲区足够大，因此从来不会填满。仅仅计算对共享变量（那些大写的变量）和锁的读和写的时延，初始时所有共享变量不在高速缓存中，且值为 0。假设处理器 1 首先获得锁。

1）在 SC 下，从处理器 1 进入 sendSpecial（）例程到处理器 2 离开 receiveSpecial（）有多少周期？请验证你的答案。还要注意各个同步事件的产生和结束时刻。 744

2）在释放同一性下从 receivespecial（）返回将花费多少个周期？

9.23 在积极释放同一性协议中，差异在释放时刻生成并传播，这与写提示类似；而在惰性释放同一性中，它们在释放时刻生成，但在获取时刻（即当获取同步请求来到处理器时）才被传播。一般来说，数据可以像写提示一样以不同的惰性程度传播。

1）描述在全软件惰性释放同一性下生成、传播和实施差异的其他可能的时刻（考虑释放时刻、获取时刻或访问缺页时刻）。你能设计出的惰性最大的方案是什么？

2）在实现各个惰性更大的方案时的复杂性如何？你选择实现哪个方案？为什么？

9.24 硬件一致性系统也可以把作废的传播推迟到释放点或甚至下一个获取点（如惰性释放同一性那样）。为什么不在硬件一致性系统中使用 LRC？把作废推迟到释放点（而不是获取点）有何优点吗？

9.25 假定在全软件的 SVM 系统中你有一个执行差异的生成和实施的协处理器，所以不需要在主处理器上执行这些动作。考虑在习题 9.23 中你设计的积极释放同一性和惰性协议的变型，试评价协议处理动作与主处理器的计算重叠的程度。画出时间坐标来说明主处理器和协处理器能做什么。你期望性能会有很大改善吗？你认为在协处理器上执行所有的协议处理和管理的主要优点和主要复杂性是什么？

9.26 为什么 TreadMarks 风格的惰性释放同一性下的废料收集要比积极释放同一性下的更为重要和复杂？在基于宿主的惰性释放同一性下呢？设计一个进行周期性废料收集的方案（讨论时刻和方法）并讨论复杂性。

9.27 在像 Blizzard-S 和 Shasta 这样以修改软件中的读和写操作来提供细粒度访问控制的系统中，主要的性能目标是降低修改部分的开销。试说明你可能使用的某些技术。你认为在何种程度上，这些技术能由编译器或由能产生可执行修改代码的工具自动实现？

9.28 在软件的共享存储器系统中（不论是细粒度的还是粗粒度的），当消息（例如页请求或锁请求）到达一个节点时，在不存在可编程的通信辅助部件情况下，有两种主要的处理方法。其一是中断主处理器，另一种是让主处理器轮询消息。

745

1）这两种方法之间主要的折中是什么？

2）你如何管理主处理器的轮询？特别是何时去查询？

3）你认为对于基于页面的虚拟存储器哪一种方法性能更好？为什么？对于细粒度的软件共享存储器呢？应用的哪些特征对你的决定的影响最大？

4）如果每个节点是一个 SMP 而不是单处理器，会有什么新问题？折中会发生什么变化？

5）你将考虑如何安排 SMP 节点的消息处理？即，进入的消息在何处处理及如何管理消息提示？

9.29 列出所有你能想出的基于差异（非基于宿主）的和基于自动更新（基于宿主）的 LRC 之间的折中。你认为哪一个性能更好？基于差异的宿主型 LRC 和基于自动更新的宿主型 LRC 相比又如何呢？

9.30 是否能保证适当标记的程序在 LRC 下的正确运行？在 ERC 下呢？在 RC 下呢？在 ERC 下能正确运行的程序一定能在 LRC 下正确运行吗？它一定能保证是适当地标记了吗？（即未适当标记的程序能在 ERC 下正确运行吗？）在 LRC 下能正确运行的程序一定能在 ERC 下正确运行吗？

9.31 考虑单写入者的基于释放的协议。在释放点，节点是否需要从当前拥有者那里获得自从上一次释放以来它修改过的每一个页的最新共享表吗？或仅仅根据自己版本的共享表，向其他节点送出各个修改过的页面的写提示？解释为什么。

9.32 考虑没有目录情况下的页版本号。它能避免在单写入者的基于释放的协议中向同一副本发送多个作废的问题吗？解释为什么，或给出一个反例。

9.33 跟踪 1）纯 CC-NUMA，2）扁平 COMA，3）带自动更新的 SVM，4）不带自动更新的 SVM，5）简单 COMA 中写访问的路径（提示：见本章中读访问是如何做的）。

9.34 你正在用下述四种应用进行体系结构研究：Ocean、LU、在本地行计算之间采用矩阵转置的 FFT（见习题 8.23）和 Barnes-Hut。对每一个应用，回答下列问题。假设是页粒度的 SVM 系统（在第 8 章针对 CC-NUMA 问了这些问题）。

1）你将对数据结构或数据分布做何变动或改进来保证扩展的存储器层次间的良好的交互？

2）就方法学而言，对于以高速缓存尺寸或以分配、一致性和通信粒度的交互，哪些是需要特别小心地表示或不去表示？哪些新交互在 CC-NUMA 中不那么重要而在 SVM 系统中变得重要了？哪些交互相对于其他变得不太重要了？

746

3）以高速缓存粒度的交互在 SVM 中还像它们在 CC-NUMA 中那么重要么？如果它们有不同的重要性，说明为什么。

9.35 考虑如习题 8.23 所述的具有矩阵转置的 FFT 运算。假定你在基于页面的 SVM 系统上运行这个程序，使用全软件的、基于宿主的多写入者协议：

1）你是愿意使用让处理器读本地分配的数据，写远地分配的数据这样的方法来实现转置，还是使用处理器读远地数据写本地数据的方法来实现转置呢？

2）现在假定你为了进一步加速基于宿主协议，以硬件支持自动的更新传播。这会改变你的折中吗？

3）协议的什么部分限制了性能？你能想出什么样的协议优化，从而大大提高这种或其他方案的性能。

747

第 10 章　互连网络设计

从本书我们已经了解到可扩展高性能互连网络处于并行计算机体系结构的核心。广义并行机有三个基本组成部分：处理器－存储器节点、节点至网络的接口和连接各部件的网络。通过前面几章已经基本了解了对并行机互连网络的要求，本章将深入考察并行机的高性能互连网络的设计。这些网络在基本概念和技术上与局域网（LAN）和广域网（WAN）类似，许多读者可能已经很熟悉了，但由于互连网络工作于完全不同的时间尺度，它的设计原则和折中很不一样。

并行计算机网络涉及如此多的不同侧面，因此是一个丰富多彩的有趣主题，但其丰富多彩也使这个主题总体上说难以理解。例如，并行计算机网络一般以有规律的模式连接。这些网络的拓扑结构有着优美的数学性质，所以拓扑结构和重要并行算法的基础通信模式之间有着深刻的关系。然而，伪随机的线路连接模式有着另一组不同的良好的数学性质，它倾向于获得更均匀的性能，不会有特别好或特别坏的通信模式。考察这个抽象层次，我们可以看到范围宽广的有意思的折中，大量研究论文的注意力完全集中在网络设计的这个侧面。另一方面，在两个独立的异步设备之间经由电或光的链路传递信息产生了复杂的工程性问题。这些问题导致了标准化的工作。从第三个观点看，多个竞争通信资源的信息流之间的交互作用对性能有微妙的效应，这些效应受到很多因素的影响。网络的性能模型是另一个理论和实践研究的巨大领域。真实网络的设计涉及每一个层次的问题。本章的目标是提供对并行计算机网络的诸多侧面的整体理解，这样读者可以在由应用需求驱动的并行机设计这一更大的问题中观察到更为广阔的网络设计空间。

与其他所有的设计一样，网络设计涉及对权衡的理解和折中的选择，从而使解决方案在全局意义上接近最优，而不是追求对某一特定的感兴趣的部分的优化。许多相互作用的侧面对性能的影响可能相当微妙。此外，在合适的网络代价模型方面并不存在清晰的一致意见，因为折中可能在非常不同的技术之间作出：例如，在链路的带宽和交换机的复杂性之间可能作出折中。建立用于评价网络设计的良好定义的工作负载也是非常困难的，因为程序需求在提交给网络之前会受到系统其他各个层次的影响。这种情况通常产生不同的“设计阵营”而不是热烈的辩论，从而往往忽视了基本假设中的区别。本章在展开计算机网络的概念的过程中指出了代价模型和工作负载的选择会如何导致各种各样重要的设计点，它们反映了关键的技术假设。

前几章已经阐明了网络设计的驱动因素。如果处理器要保持一定的计算速度，程序中通信与计算的比对网络所必须提供的数据带宽提出了要求。但是，不同程序的负载变化很大，信息的流动在物理上可以是局部的或分散的，在时间上可能是阵发的或是相当均匀的。此外，程序的等待时间受到网络延迟的严重影响，等待花费的时间影响带宽的需求。我们已经看到不同编程模型的实现使用不同的通信粒度（这会影响网络所看到的数据传输的尺寸），为了实现更高层的编程模型，它们在网络事务层使用不同的协议。

本章首先给出作为所有网络基础的一组基本的定义和概念。10.2 节建立了通信延迟和

带宽的简单模型以便揭示网络设计风格上的核心差别。10.3 节具体说明了构成网络的关键成分。10.4 节说明了在公共框架下互连拓扑结构的丰富空间，10.5 节在基本的工作负荷假设下建立了设计取舍与代价、延迟和带宽之间的联系。10.6 节阐明了在网络拓扑中各种避免死锁的消息路由的方法，并说明了路由对通信性能的进一步影响。为了帮助读者更精确地理解对各种选择的工程性折中以及支撑更抽象的网络概念的机制，10.7 节深入讨论了构成网络的基本构造模块即交换机的硬件组成。10.8 节探索了网络内流控的不同途径。以上述论述为基础，10.9 节通过一系列案例分析综合了全部论点，考察了并行计算机网络技术向其他网络体系的转移，包括正在出现的系统域网络（SAN）。

10.1 基本定义

750 并行机中互连网络的任务是从任何源节点向任何目的节点传输信息，支持用以实现编程模型的网络事务处理。它必须以尽可能小的延迟完成这个任务，并允许大量这样的传输并发地发生。此外，相对于机器其他部分的成本来说，它必须是便宜的。

图 10-1 是我们的广义大规模并行体系结构的展开图示，它说明了一台并行机中的互连网络的结构。源节点的通信辅助部件通过把信息推向网络接口（NI）而启动网络事务处理。根据所支持的通信抽象，这些事务在目的节点由通信辅助部件、处理器或存储器控制器处理。

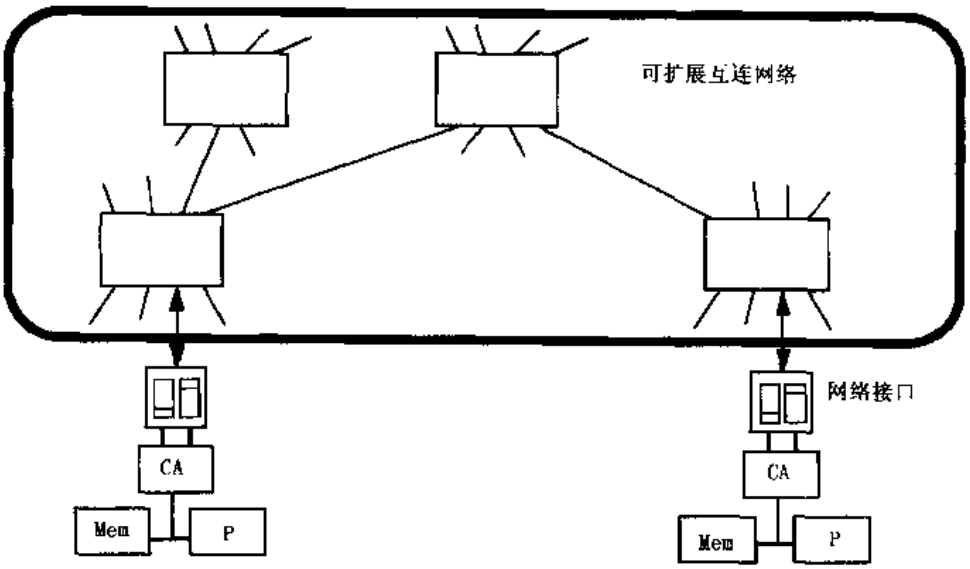


图 10-1 广义的并行机互连网络。通信辅助部件 CA 代表处理器或存储器，通过网络接口启动网络事务处理，使信息经过一系列链路和交换机的传输，到达远程的节点，在那里发生网络事务处理

网络由链路和交换机组成，它提供了从源节点到目的节点导引信息的手段。链路本质上是传送模拟信号的一束导线或光纤。为了使信息沿链路流动，一端的发送器把数字信息转换成模拟信号，模拟信号经驱动在链路上传送，由另一端的接收器再转换成数字的符号。在数字符号流和模拟信号流之间进行转换的物理协议构成网络设计的最低层。发送器、链路和接收器构成了与链路相连的交换机（或 NI）之间数字信息流动的通道。链路层协议把穿越通道的符号流分割成称做数据包（packet）或消息（message）的较大的逻辑单元，交换机对这
751 些单元进行解释，以便将每一个从输入通道到达的单元转发到合适的输出通道。处理节点经

首先考虑交换策略。对于电路交换，我们可以期望延迟与 h 成比例，包括建立线路，配置路线上的每一个交换机，通知源节点路径已经建立。从这以后，数据沿着线路传送的时间是 n/b 加上与 h 成比例的额外的小的延迟。所以，当一个 n 个字节的消息在一个电路交换网络中传送距离为 h，以网络周期时间 τ 为单位的无负载时延是

$$T_{cs}(n, h) = \frac{n}{b} + h\Delta \tag{10-3}$$

式（10-3）中，电路建立延迟和路由延迟是相加关系，与消息的大小无关。所以，当消息的长度增加时，路由距离（即拓扑结构）成为无负载通信时延中不显著的部分。电路交换传统上用于电信网，因为通话的建立与通话的持续时间相比是很短的，它也少量用于并行计算机网络，包括 Meiko CS-2 和 BBN Butterfly。一个重要的区别是，并行计算机中电路的建立是通过引导消息通过网络并占据路径使其为一条电路，更为传统的方式是在路径的一端计算路由，对交换机进行配置，然后在电路上传输信息。

我们也可能既保持包交换，又降低单纯存储转发路由的无负载时延。式（10-2）中的关键在于延迟是路由距离和整个消息的占用度的乘积。但是，可以把一个长的消息分片成几个小的数据包，以流水的形式流过网络。在这种情况下，无负载时延成为 756 ~ 757

$$T_{sf}(n, h, n_p) = \frac{n - n_p}{b} + h\left(\frac{n_p}{b} + \Delta\right) \tag{10-4}$$

这里 n_p 是分片的大小。实际的路由延迟是与数据包的尺寸而不是消息的尺寸成正比。这是因特网这样的传统的数据通信网所采用的基本方式（用软件实现）。

在并行计算机网络中，路由和通信流水执行的概念被被进一步发挥。大多数并行机使用直通路由的数据包交换，这里交换机在检查了数据包头的前几个 phit 后就做出路由的决策，让数据包的其余部分从输入到输出通道间直通，如图 10-3 所示，这样即使单个数据包的传输也能流水执行。（如果回到我们的车辆的比喻的话，直通路由就像火车在轨道上碰到道岔所发生的情形类似，火车头被引导到正确的轨道，后面的车厢跟着走。与此对照，存储转发路由与车站上发生的情形类似，火车的所有车厢到达并停留，直到车头牵引它们继续向下一站进发。）对于直通路由，无负载时延具有如下与电路交换类似的形式，虽然由于两者的处理机制有相当大的差异，路由系数 Δ 可能不同。

$$T_{ct}(n, h) = \frac{n}{b} + h\Delta \tag{10-5}$$

注意在直通路由下，单个消息可能占据从源到目的地的整个路径，与电路交换很类似。消息的头在向其目的地移动时建立起路径，当尾通过后路径被清除。

前面对通信时延的讨论谈的是在路途中不遭遇其他通信流量的情况下消息从源到目的地的流动。在这种无负载的情况下，可以简单地把网络看作一个流水线，具有启动开销、流水深度和每级的时间等参数。不同的交换和路由策略会改变流水线的有效结构，而拓扑结构、链路带宽和分片决定了流水深度和每级的时间。当然，网络的魅力在于它们不是简单的流水线，而是很多流水线的交织物。使用网络而不是总线的动机就是允许多个数据传输同时发生。这意味着一个消息流可能与其他消息流发生冲突，竞争资源。从根本上说，网络必须提

由一系列链路和交换机进行通信。节点层的协议把对远程通信辅助部件的命令嵌入在节点间交换的数据包或消息之中，完成网络事务处理。

我们可以将并行机的互连网络形式化地描述成一个图，这里顶点 V 是由通信通道 $C \subseteq V \times V$连接的处理主机或交换机单元。通道是主机或交换机单元之间的一条物理链路，包括保存传送数据的缓冲器。通道宽度为 w，信号速率为 $f = 1/\tau$（τ 是周期时间），它们一起决定了通道带宽 $b = wf$。在一个周期中链路上传输的数据的量叫做一个物理单元，或 *phit*[⊖]。交换机在固定数量的输入和输出通道之间建立连接；这个数量叫做交换机的度。典型情况下，主机与单个交换机相连，但也可以与多个独立的通道连接。消息沿一条路径或路由，通过网络从源主机传输到目的主机，路径由一系列通道和交换机组成。

由街道和交叉路口组成的道路系统是一个有用的比喻。每条街有它的速度限制，具有几条车道，这决定了它的峰值带宽。街道可以是单向或双向的。交叉路口允许旅行者在数量固定的街道之间进行切换。每一次旅行中，一组人员从源地点出发沿一条路线旅行到目的地。他们可以在多条可能的路线、多种交通工具和应付途中可能遇到的交通拥挤的多种不同方法中做出选择。在一个城市中，大量的这样的旅行会同时发生，它们各自的路线可能交叉，也可能共享路线上的某些路段。

网络的特征由其拓扑结构、路由算法、交换策略和流控机制所刻画。

- 拓扑结构是网络图的物理互连结构，它可以是规则的，如二维的网格（许多中心都市的典型结构），也可以是非规则的。大多数并行机采用高度规则的网络。通常我们区别直接或间接网络；直接网络使主机与每一个交换机连接，而间接网络中主机仅与特定的交换机子集相连，这一子集形成了网络的边沿。许多机器使用混合的策略，所以更重要的是要区别两类节点：主机可以产生和吸收流量，而交换机仅仅传递经过的流量。
- 路由算法决定了消息在网络图中沿哪一条路径移动。路由算法把可能的路径集合限制为较小的合法路径集。存在许多不同的路由算法，提供不同的保证和各异的性能折中。例如，继续我们前面城市交通的比喻，一个城市为了消除网格中的死锁，可能会立法，规定汽车在到达目的地的惟一一次南北方向的转弯之前必须保持东西方向行驶，而不能之字形地穿过城市。我们将看到这一措施的确消除了死锁，但是也限制了司机避免途中交通拥挤的能力。在并行机中，我们仅仅关心主机到主机的路径。 752
- 交换策略决定了消息中的数据如何穿越它的路径。有两种基本的交换策略。在电路交换中，在源和目的之间建立一条路径并一直保持到消息通过电路为止（这个策略有点像预约一条游行路线，这有利于大量人员的通过，但是它需要预先计划，而且对想要穿过或共享预约路线的路段的任何交通来说，都可能是不愉快的，因为即使在看不到游行队伍时也不能通行。这也是电话系统使用的策略，它为每次通话建立一条线路，可能要经过许多交换机。）另一种策略是数据包交换，它将消息分割成一系列数据包。数据包包含路由、序列信息和数据。数据包被逐个从源引导到目的地。

⊖ 因为很多网络异步地操作而不是由单一全局时钟控制，网络“周期”这一概念并不像周期在处理器中用得那样广泛。我们可以等价地把网络周期定义为传输最小的信息物理单元，即 phit，所需的时间。对于并行体系结构而言，可以方便地按常用术语来理解处理器周期时间和网络周期时间。的确，这两个技术体系越来越相像了。

（一个显而易见的比喻是把人们分成小组，分乘小轿车旅行。）数据包交换允许我们更好地利用网络资源，因为链路和缓冲器只有在数据包通过时才被占用。

- 流控机制决定消息或消息的一部分，何时在它的路径上运动。当两个或更多的消息试图同时使用同一网络资源（例如，一个通道）时，特别需要流控。可以让其中某个数据流原地暂停，或令其分流进入缓冲器，或迂回到另一条路径，或简单地丢弃它。每一种选择都对交换机的设计提出特殊的要求，并影响到通信子系统的其他方面。（在交通系统的比喻中，丢弃交通流量显然是不可取的。）能在链路上传输、接受或拒绝的最小信息单位叫做流控单位，或称 *flit*。它可以和 phit 一样小，也可能和数据包或消息一样大。

为了说明 phit 和 flit 之间的区别，让我们考察 7.1.4 节中的 nCUBE 实例。链路的宽度是 1 比特，所以 phit 是 1 比特。但是，交换机接纳以 36 比特（32 比特的数据加上 4 比特的奇偶校验位）的块为单元的消息。只有当缓冲器能容纳 36 比特时，它才允许下一个 36 比特到来，所以 flit 的尺寸是 36 比特。在很多更为现代的机器如 T3D 中，phit 和 flit 是相同的。

拓扑的一个重要的性质是网络的直径，它是任何两个节点之间最短路径当中最长的一条路径的长度。一对节点之间的路由距离等于途中穿过的链路的数量，这至少与节点间的最短路径一样长，也可能更长。平均距离是所有节点对之间路由距离的平均值，它也是随机的节点对之间的期望距离。在一个直接网络中，必须在每一对交换机之间提供路径，而在间接网络中，只需要在主机之间提供路径。如果从网络中去掉一组链路或交换机，某些主机之间不再有路径连接的话，该网络是可分割的。

753

本章大多数的讨论都涉及到数据包，因为大多数现代的并行机网络使用数据包交换。当涉及电路交换的特别重要的性质时，我们会特别指出。数据包是自分界的数字符号的序列，逻辑上由图 10-2 所说明的三部分组成：数据包头、有效负载和数据包尾。数据包头位于数据包的前部，通常包含路由和控制信息；当数据包到来时，交换机和网络接口能根据数据包头决定对数据包做什么事情。数据包的有效负载部分包含网络上传输的数据。数据包尾是数据包的结尾，通常包含当消息进入链路时生成的差错校验码。数据包头也可以包含一个独立的差错校验码。

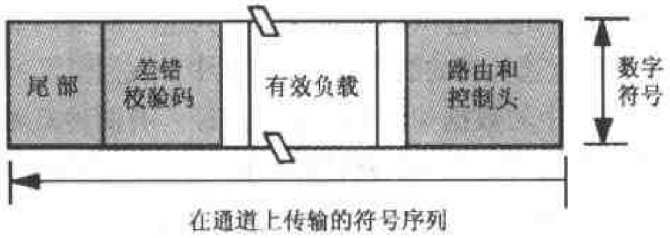


图 10-2 数据包的典型格式。数据包是沿着路径由交换机控制的信息的逻辑单位。它由三部分组成：数据包头、有效数据负载和数据包尾。当数据包沿路径前进时，数据包头和数据包尾由交换机解释，但有效负载不会。节点层次的协议由有效负载携带

在网络环境下建立抽象的两个基本机制是封装和分片。封装是指在给定层次的消息格式中携带不加解释形式的更高层协议信息。分片则是将较高层协议信息分割成给定层次的消息序列。虽然任何网络都有这些基本的机制，但并行计算机网络的抽象的层次比起 Internet 这样的网络就要浅得多，而相互高效地配合是其设计原则。为了使这些概念更加具体，让我们

来考察一下数据包，数据包的头和尾构成一种由交换机解释的、封装了有效负载的封套。与节点协议有关的信息包含在有效负载之中。例如，读请求通常可以由单个数据包传递给远地的存储器控制器，而回答的高速缓存行又是另一个数据包。存储器控制器并不关心数据包所经过的实际路由或数据包头和尾的格式。同样，网络也不关心在数据包有效负载里的远程读请求的格式。大块的数据传输一般不能由单个数据包完成，而是要把它分成几个数据包。每一个数据包都需要包含说明把数据送到哪里以及它携带整个数据序列中的哪一片这样的信息。向下一个层次，每一个数据包在链路层又被分割成一系列符号，依次经由链路发送，所以不需要序列信息。较高层的信息通过一个或多个由较低层协议解释的封套所携带的情况在网络设计的每一个层次都会发生。 754

10.2　基本的通信性能

网络设计四个主要方面中的任何一个方面都有很多东西要理解，但是在进入这些方面的细节之前，对它们之间如何相互作用，决定整个通信子系统的性能和功能的状况有个一般性的了解是有益的。基于第 7 章对网络的简要讨论，让我们从时延和带宽两个侧面来观察性能。

10.2.1　时延

为了建立有助于理解网络的基本性能模型，我们可以扩展从第 1 章起就一直沿用的通信时间的模型。从源到目的地传送 n 个字节的信息所需的时间由如下 4 个部分组成：

$$\text{时间}\ (n)_{S-D} = \text{额外开销} + \text{路由延迟} + \text{通道占用度} + \text{竞争延迟} \tag{10-1}$$

额外开销（overhead）和实际传输的端点如何将消息引入和引出网络有关，前面章节中论及网络接口时已经对它进行了广泛的讨论。我们知道有些机器的设计依据是所传送的块的大小等于高速缓存行的尺寸，而另一些机器则针对较大消息的 DMA 传输进行了优化。关于其他几个部分，路由延迟（routing delay）和通道占用度（channel occupancy）已在前面的章节中被有效地概括为网络对于典型尺寸的消息的无负载时延（unloaded latency），而竞争却在很大程度上被忽略了。这几个部分是本章的重点。

通道占用度方便地提供了通信时延的下界，它与消息的走向及网络中发生的其他事情无关。当我们更深入地考察网络设计时，我们将看到每条链路的占用度受到通道宽度、信号速率和控制信息量的影响，而后者又受到拓扑结构和路由算法的影响。前面的章节涉及的是从网络外部观察到的通道占用度，即经过路由中的瓶颈通道传送消息的时间；而从网络内部看，通道占用度与沿路由的每一步相关。通信辅助部件的占用时间周期从接受来自处理器或存储器控制器的通信请求起直到将数据包送入网络为止。数据包在途中经过的每一个通道都被数据包占用一段时间，目的地的通信辅助部件也一样。 755

例如，我们需要了解的一个问题是数据包编码的效率。数据包封装提高了占用度，因为源节点附加了数据包头和尾符号，而目的节点要把它们剥离。所以，对于大小为 n 的有效负载，通道的占用度是

$$\frac{n + n_E}{b}$$

这里 n_E 是封装的尺寸，b 是通道的原始带宽。这个问题可以从“外观”上通过说明链路的

有效带宽来论述的，至少对于固定大小的数据包来说，有效带宽与原始带宽的比率是

$$\frac{n}{n+n_F}$$

但是，在网络内部，数据包的效率仍然是一个设计要考虑的问题。对小的数据包来说，其影响更为突出，但它也取决于路由是如何完成的。

从网络外部看到的路由延迟是从源到目的地传送一个特定符号，比如说消息的第一个比特，所需的时间。从网络内部看，沿路径的每一步都发生路由延迟，它们积累起来成为外部观察到的延迟。路由延迟是路径中通道数量，称为路由距离 h，和每个交换机选择正确输出端口所产生的延迟 Δ 的函数。（就产生路由延迟来说，我们可以方便地把节点至网络的接口等同于交换机。）路由距离取决于网络拓扑、路由算法以及特定的源和目的节点对。整体延迟受到交换和路由策略的很大影响。

如图 10-3a 所示，对于数据包交换的存储转发式路由，交换机接收整个数据包，然后才将它转发到下一条链路上去。大多数广域网和几个早期的并行计算机使用这个策略。对于一个包括封装的 n 个字节的数据包，使用存储转发路由，网络无负载时延是

$$T_{sf}(n,\ h) = h\left(\frac{n}{b}+\Delta\right) \tag{10-2}$$

这里 Δ 是每一跳的额外路由延迟。

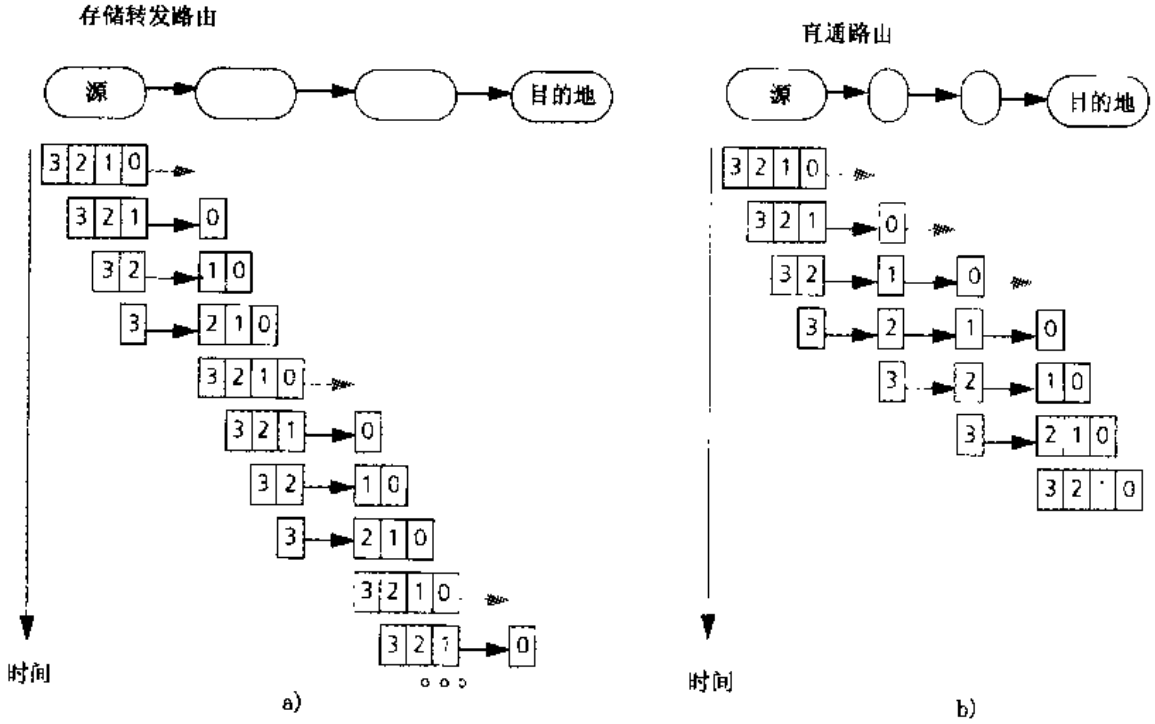


图 10-3　包交换网中存储转发路由与直通路由的比较。一个 4 flit 的数据包在存储转发和直通路由下从源到目的地穿过三跳。直通根据第一个 flit 做出路由决定（灰箭头），并使数据包流水地通过各个交换机，从而实现较低的时延。存储转发要积累整个数据包，然后才将它送往通向目的地的路由

式（10-2）告诉我们网络拓扑在决定网络时延中极为重要，因为拓扑结构从根本上决定了路由距离 h。事实上，问题要比这更复杂。

供处理竞争的机制。虽然竞争状态下的行为取决于网络设计的几个侧面：拓扑结构、交换策略和路由算法，但是基本的一点是：在任何给定的时刻，一个通道只能被一个消息占用。如果两个消息试图同时使用相同的通道，其中之一必须被推迟。典型的做法是，由每个交换机为输出通道提供某种仲裁的手段。所以，交换机从竞争各个输出的输入数据包中选出一个，而其他的则以某种方式推迟。

758

一个贯穿始终的网络设计问题是如何处理竞争。这个问题在本章中反复出现，所以让我们首先看一看它对时延意味着什么。显然，竞争增加了节点所经历的通信时延。它究竟如何增加时延取决于处理网络内竞争所用的机制，而这种机制又根据基本的网络设计策略而不相同。例如，对于数据包交换，各个交换机都可能经历竞争。使用存储转发路由，如果在交换机中缓存的多个数据包需要使用相同的输出通道，数据包之一会被选中，其他的数据包在缓存中阻塞，直至它们被选中为止。因此，竞争在基本的路由延迟之上增加了排队延迟。对于电路交换，竞争的效应在试图建立电路时出现，典型的做法是，一个路由探针向目的地伸展，当遇到一个已被预定的通道就缩回来。在一段时延之后，网络接口再次尝试建立电路。所以，在竞争情况下，获得对网络访问的启动开销增加了，但是一旦电路建立起来了，整个消息将以全速传输。

对于直通数据包交换，有两种可用的数据包阻塞的选择。虚拟直通（virtual cut-through）方式把阻塞的输入数据包送入缓存，这样在竞争条件下其行为降级为存储转发路由。虫孔方法只在交换机中缓存几个 flit，而将消息的尾部原地留在路径上。消息的阻塞部分很像在网络中保持的一条电路。

交换机中只有有限的供数据包使用的缓存，所以在持续竞争的情况下，交换机中的缓存可能被填满。如果在交换机中已没有缓存空间来保存输入的数据包怎么办？在传统的数据通信网络中，链路很长，在通道的两端点之间很少有反馈发生，所以典型的做法是丢弃数据包。因此，在竞争条件下，网络变得很不可靠，要在节点中使用复杂的协议（例如 TCP/IP 慢启动）使请求的通信负载与网络能提供的能力相匹配，不要引起高丢弃率。大多数 ATM 交换机也使用缓存溢出丢弃的策略，即使它们是被用来建造紧密集成的机群系统。与广域网的情况一样，源节点收不到任何关于它的数据包丢失的指示，所以它必须依赖于某种超时机制来推测问题的发生。

在并行计算机网络中，处理发往已满缓存的数据包的典型做法是原地阻塞，而不是丢弃；这要求在链路上输出端口和输入端口间有握手交互，也就是说，要有链路级的流控。在持续拥塞的情况下，从网络中发生资源竞争的点起，朝着向该点推送流量的源会发生流量"堆积"。最终源节点会感受到来自网络的堆积压力（当网络拒绝接受数据包时），这使得进入网络的数据流的速率放慢到能通过瓶颈[⊖]。在网络中增加缓存的量能允许竞争坚持更长时间而不在源节点引起堆积压力；但是当竞争确实发生时，这也提高了网络中潜在的排队延迟。

759

对于采用消息阻塞的网络有一个问题值得重视，即堆积可能会影响那些并不是流向高度

⊖ 这个情形和多个车道的车流汇聚到一个狭窄的隧道或桥梁的情况完全一样。当交通流量低于隧道的流通能力时，几乎不会发生延迟，但是当进入的车流超过隧道的带宽时，交通就会阻塞。当交通拥塞用尽了道路的容纳能力时，向前移动的平均流量会足够慢，使得平均的带宽等于隧道的带宽。

竞争的输出点的流量。假定某个节点由于保持着广泛使用的重要全局变量而成为网络流量经常流向的目的地，这被称为“热点”。如果以那个输出为目的地的流量的总量超过了输出带宽的话，该流量将在网络内积压。如果这种情形持续下去，积压会经由指向该目的地的通道树反向传播，这被称为树饱和（Pfister and Norton 1985）。跨越该树的任何流量都会被延迟。在虫孔路由网络中，阻塞消息的另一种有趣的方法是丢弃消息，但将拥塞的情况通知源节点。例如，在 BBN Butterfly 机中，源节点在消息头到达目的地（即形成电路）之前扣压消息的尾，如果在途中发生了冲突，虫整个缩回其源节点（Reitberg and Thomas 1986）。这大大减轻了树饱和的影响。

我们能从以上简要的讨论中看到，网络设计的所有方面——链路带宽、拓扑结构、交换策略、路由算法和流控——综合决定了每个消息的时延。我们也应该清楚地了解到在时延和带宽之间存在关系。如果程序所要求的通信带宽与可用的网络带宽相比较低，很少会有冲突发生，缓存趋向于空，时延也低，直通路由特别如此。当要求的带宽增加，由于竞争时延将会上升。

重要的但经常被忽略的一点是：并行计算机网络是一个有效的闭合系统，网络对流量的源有反馈。对网络施加的负载取决于处理节点请求通信的速率，该速率又取决于网络以多快的速度提供通信。当程序涉及某种依赖关系时，程序性能受时延的影响最大。依赖关系表现为：程序必须等待，直到读操作完成或接收到一个消息为止；当它等待时，提交到网络的负载被丢弃，时延降低。这种情形与国家范围内的文件传输完全不同，因为那是不相关的流量在竞争。在并行机中，程序主要与其自身竞争通信资源。如果以多道程序的形式使用计算机，并行程序之间也会互相竞争；但是，当网络的服务速率由于竞争而降低时，每道程序的请求速率将会降低。因为对于并行程序的性能来说低时延的通信是关键的，本章的重点将放在直通数据包交换网络上。 760

10.2.2 带宽

网络带宽对并行程序性能是决定性的因素，部分原因是较高的带宽降低占用率，也因为较高的带宽降低了竞争的可能性，还因为程序某些阶段可能推出大量数据而不必等待各个数据的传输结束。由于网络的行为像流水线，所以即使在大时延情况下它也可能提供高带宽。

以两个观点观察带宽是有用的：通过网络对所有节点可用的“全局”聚合的带宽和对一个节点可用的“局部”个体带宽。如果一个程序的总通信量是 M 个字节而网络的聚合通信带宽是每秒 B 个字节，那么显然通信时间至少是 M/B s。但另一方面，如果所有的通信是指向或来自单个节点，该估计显然是过于乐观了，通信时间应由通过该节点的带宽所决定。

首先考察单个节点可用的带宽并了解它如何受到网络设计的影响。前面已经看到了有效局部带宽由于数据包编码密度而比原始链路带宽降低，成为

$$b\frac{n}{n+n_E}$$

进而，如果交换机在做出其路由决定时将数据包阻塞了 Δ 个周期的路由延迟时间，那么有效局部带宽进一步降低为

$$b\left(\frac{n}{n+n_E+w\Delta}\right)$$

因为 $w\Delta$ 是传输在链路阻塞时丢失数据的机会。所以，数据包格式和路由算法这样的网络设计问题将影响单个节点可以看到的带宽。如果多个节点同时通信并产生竞争，那么可以看到的局部带宽将进一步降低（时延会上升）。如果多个节点向同一节点发送消息，任何网络中都会在端点发生竞争，但竞争也可能在网络内部发生。网络拓扑结构和路由算法的选择对网络内部的竞争的可能性会有影响。

如果许多节点同时通信，我们不仅要了解每个个体节点可用的带宽，也要注意网络能够支持的全局带宽。首先，我们应该明确网络聚合通信带宽的概念。最常用的聚合带宽的概念是对分带宽，它是网络中存在通道的最小集合；如果去掉这些通道，就会将网络分成两个相等的非连接的节点集合，这个最小集合中的通道的带宽的总和就是对分带宽。这是一个有价值的概念，因为如果通信模式是完全均匀的，两个方向都有一半的消息会跨越对分边界。后面将看到对不同的网络拓扑结构平均每节点的对分带宽变化很大。但是，对分带宽作为聚合网络带宽的尺度来说，并不能完全令人满意，因为通信不一定非得是均匀分布在整个机器中。如果通信是局部化的而不是均匀的，对分带宽将给出通信时间的悲观的估计。适应局部化通信模式的另一个全局带宽的概念是从节点到网络的链路带宽的总和。这个全局带宽概念担心的是网络的内部结构可能不支持它。显然，网络的平均聚合带宽依赖于通信的模式，特别是它依赖于数据包移动多远，所以应该更细致地考察这个关系。

761

网络中所有通道（或链路）的总带宽是通道的数量 C 与每个通道的带宽的乘积，即每秒 Cb 个字节，每周期 Cw 个比特或每周期 C 个 phit。如果 N 个主机中的每一个主机每 M 个周期发出一个平均路由距离为 h 的数据包，那么每个数据包平均占用 h 个通道，占用时间为 $l = n/w$ 个周期，网络的总负载是每周期 Nhl/M 个 phit。链路的平均利用率至少是

$$\rho = M\frac{C}{Nhl} \tag{10-6}$$

显然这必然小于 1。认识这一点的一种办法是每个节点的链路数 C/N 反映了每个节点可用的平均带宽（每节点每周期的 phit 数）。该带宽的消耗与路由距离和消息尺寸直接成比例。每节点的链路数是拓扑结构的静态性质。平均路由距离由拓扑结构、路由算法、程序通信模式和程序对机器的映射决定。好的通信局部性可能产生小的 h，而随机的通信将产生平均的路由距离，实在太坏的通信模式则会导致消息穿越整个直径。消息的大小是程序行为和通信抽象所决定的。一般地，式（10-6）中的聚合通信的需求表明当机器规模扩大，每个节点的通道数量必须随着时延期望值上升而增加。

在实践中，几个因素限制了通道利用率 ρ，使它比 1 小很多。各个链路的负载可能不是完全平衡的。即使它是平衡的，路由算法可能使程序所使用的特定的通信模式无法利用所有的链路。即使所有的链路都可以利用并且在程序持续的整个期间负载是平衡的，也会发生负载的随机变化和对低层资源的竞争。所有这些因素影响了网络的饱和点，网络饱和点代表了网络能有效地交付的全部通道带宽。正如图 10-4 所说明的，如果处理器对网络提交的带宽需求（称作建议带宽）不大、时延保持低，交付带宽随建议带宽的增加而上升。但是，在某一点，要求更多的带宽仅仅加剧了资源的竞争，使时延急剧上升。网络本质上只能传送它能应付的流量，额外的请求只能在缓存中积聚起来。增加建议带宽并不能增加交付带宽。我们所设计的并行计算机应尽量避免使网络饱和，这可以通过提供足够的通信带宽或者通过限制

处理器所提交的需求来解决。

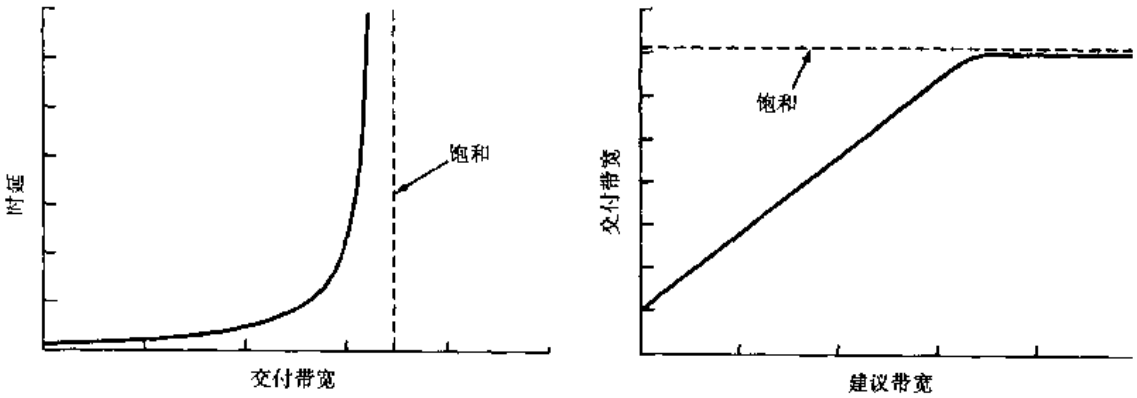


图 10-4 典型的网络饱和行为。当请求的带宽大大低于网络所能交付的带宽时，网络能提供低时延的传输。在这种形态下，交付带宽随请求线性增长。但是，在某一点，网络饱和，额外的负载导致时延急剧上升，但不能产生额外的交付带宽

关于图 10-4 所说明的当网络负载接近饱和时，延迟急剧上升的现象有一点应注意，图中所说明的行为是假设向系统施加的负载与响应时间无关的所有排队系统（和网络）的典型特征。源一直以高于系统能服务的速度向系统推送消息，以致于在某处形成任意长度的队列，时延随该队列的长度而上升。换句话说，这种简单分析假设了一个开放系统，而在真实世界中，并行机是闭合的系统。网络中只有有限的缓存能力，通常在网络接口只有有限的通信缓存。所以，如果这些“队列”已被填满，源就会放慢送出消息的速度，降低它们对服务速率的需求，因为在从队列中移走一个数据包之前没有地方存放下一个数据包。流控机制影响源和链路之间的耦合。此外，并行程序中所固有的相关性实际建立了某种程度的端到端的流控，因为处理器必须先要收到远程的信息才能继续依赖于这些信息的工作，再产生额外的通信流量。不过，重要的是要认识到，即使是在最佳的场合，也不能期望像网络链路这样的共享资源能达到 100% 的利用率。

这个并行机网络的简略的性能模型说明真实网络的时延和带宽取决于网络设计的所有方面，我们将在本章的剩余部分详细地考察这些方面。网络性能建模本身是一个有着大量文献的丰富领域，感兴趣的读者应该把参阅下列参考文献作为他们的出发点：Agarwal（1991），Dally（1990b），Karol et al.（1987），Kermani and Kleirock（1979），Kruskal and Snir（1983）以及 Peterson and Davie（1996）。此外，懂得性能并不是网络设计中惟一的驱动性因素也是重要的。成本和容错是另外两个关键性的指标。例如，在几种大型的机器中，网络的连线复杂度成为关键的问题。因为这些问题在相当大程度上依赖于设计的特殊细节以及所采用的拓扑结构，将在考察不同的设计方案时再讨论它们。 762 763

10.3 组织结构

本节概述并行计算机网络的基本组织结构。一个有用的方法是依据我们所熟悉的处理器组织结构和可采用的工程约束来考虑这个问题。我们一般认为处理器由数据通路、控制逻辑和存储器接口组成，或许还包括存储器层次结构中的片内部分。数据通路还可以进一步分为 ALU、寄存器组、流水线锁存器，等等。控制逻辑是建立在对数据通路上发生的数据传输的

考察基础之上的。数据通路内部的局部连接都很短，且可以随着 VLSI 技术的改进而很好地扩展，而控制线和总线则相当长，随着芯片密度的增加，它们跟不上门电路的速度。一个非常类似的分解和组装的概念适用于网络。可扩展的互连网络由三个基本成分组成：链路、交换机和网络接口。对这些组成元素、它们的性能特征和它们固有的成本的基本理解是评价不同网络设计的基础。这些组成元素所执行的操作相当有限，基本上就是把数据包传送到它们想去的目的地。

10.3.1 链路

一条链路是由一条或多条电线或光纤组成的缆线，两端各有一个与交换机或网络接口端口相连的连接器。它允许模拟信号从一端开始传送，在另一端接收，通过采样获得原本的数字信号流。在实践中，链路的电气和具体工程的实现多种多样，但是它们基本的逻辑性质可以用三个彼此独立的维度来刻划：长度、宽度和定时。

1）短链路是同时只能传递单个逻辑值的链路；长链路可以被看作允许多个逻辑值以光速几分之一的速度（每纳秒 1～2 英尺，由特定的介质决定）同时沿一条链路传播的传输线路。

2）窄链路是将数据、控制和定时信息在各条线路上多路复用传输的链路，例如单条串行链路；宽链路是能同时传输数据和控制信息的链路。在这两种情况下，网络链路一般要比处理器内部的数据通路窄，比如说 4～16 比特宽。

764

3）时钟可以是同步或异步的。在同步情况下，源和目的地以同一个全局时钟操作，所以接收端根据公共时钟对数据采样；在异步情况下，源节点以某种方式把自己的时钟编码到被传输的模拟信号之中，目的地节点从信号中恢复源时钟，并将该信息转换到自己的时钟域。

一条短电气链路的行为和数字部件之间的传统连接相类似。信号速率本质上是由对导线充电，直到其两端表示同一逻辑值的时间决定的。如果使用足够大的功率来驱动链路的话，该时间仅随链路长度对数增长[⊖]。此外，还必须对导线进行适当的终端匹配，以避免反射。这就是为什么链路是点对点的、而不是像总线那样是多点式的原因。

CRAY T3D 提供了一个宽、短、同步的链路设计的很好的例子。每一条双向链路在两个方向各包含 24 比特：16 比特数据，4 比特控制，另外 4 比特反向提供链路的流控，从而交换机不会力图向满的缓存发送流控单元（flit）。整台机器在同一个 150 MHz 时钟下工作。一个流控单元等于一个 16 比特的物理单元（phit）。控制位中有两位标识物理单元的类型（00 是无信息，01 是路由标签，10 是数据包，11 是数据包的结尾）。

在一条长导线或光纤上，信号沿链路从源向目的地传播。对于长链路而言，延迟显然与导线的长度成线形关系。信号速率是由接收端能对信号正确采样的时间决定的，因此导线长度是由信号沿链路的衰减所限制的。如果链路存在一根以上的导线，信号速率和导线长度受到多根导线上信号扭斜的限制。

我们可以把正确采样的模拟信号看作一段时间内从源向目的地传送的数字符号（phit）的流。每条导线上的逻辑值是由电平的高低或电平的跳变所表示的。一般对数字符号进行编码，使得常见的故障（如粘附故障和连接开路）易于发现，也容易维持时钟。我们必须在符

⊖ 导线的 RC 延迟随长度的平方而增长，所以对于固定的信号驱动功率，网络的周期时间主要受导线长度的影响。但是，如果使用驱动树增加驱动器的驱动强度，在较长的导线上驱动负荷的时间仅对数地增长。（如果 τ_{inv} 是基本门电路的传输延迟，则长度为 l 的短导线的净传播延迟的增长为 $t_s = K\tau_{inv}\log l$）

号流中识别出一个个的数据包。所以，分帧是链路的信号传输惯例的一部分，它标明数据包的头和尾物理单元。在一条宽链路中，可以用独立的控制线来标识头和尾物理单元。例如，当数据包头的第一个物理单元到来时某一条线的电平变为高，在数据包尾的最后一个物理单元到来之前，它一直保持为高。在 T3D 中，路由标签物理单元和数据包结束物理单元提供了数据包的分帧功能。在异步串行链路中，还必须从输入的模拟信号中提取时钟，这通常是以一系列二进制值中的特殊同步簇实现的（Peterson and Davie 1996）。 765

CRAY T3E 的网络提供了一种与 T3D 的方便的对照。它使用长、宽、异步的链路设计。它的链路每个方向为 14 比特宽，工作于 375 MHz 的频率。每个“比特”是以用低电平的差分信号（LVDS）输送的，一对导线上额定的摆幅为 600 mV；即接收器检测两条导线的电平差异而不是相对地电平的电位的高低。时钟与数据一起传输。最大传输距离大约是 1 m，但即使在这个长度，多个位可以在导线上同时传输。一个流控单元包含 5 个物理单元，所以交换机以 75 MHz 频率对 70 比特操作，这 70 比特中有一个 64 比特的字加上一些控制信息。流控信息由反方向链路上的数据包和空闲符号携带。流控单元序列被组装成单字和 8 字的读和写请求数据包、消息数据包和其他特殊数据包。链路的最大数据带宽是 500 MBps。

一般来说，帧内数据包的编码是由挂在链路上的节点解释的。典型的做法是，交换机为了完成路由和差错校验要解释数据包的封装，对有效负载不加处理传送到其目的地主机，在那里再对进一步的分层或内部的封装进行解释并剥离。然而，目的地节点可能需要通知源节点它是否能保持数据。这需要某种有别于实际通信的节点对节点的信息。对于宽链路而言，可以设置两个方向的控制线来提供这种信息。而窄链路几乎总是双向的，所以可以像 T3E 所做的那样，在反向的流之中插入特殊的流控信号[○]。

可扩展的一致接口（SCI）定义了一条长而宽的铜线链路和一条长而窄的光纤链路。链路是单向的，节点总是组织成环形。铜线链路由 18 对导线组成，在 250 MHz 时钟的上下沿都使用差分信号。它携带 16 比特数据、时钟和一个标签位。光纤链路是串行的，工作于 1.25 Gbps。数据包由一系列 16 个比特的物理单元组成，它的头包含一个目标节点号物理单元和一个命令物理单元。它的尾由一个 32 位的 CRC（循环冗余校验码）字组成。标签位通过区分空闲符号和数据包物理单元而实现数据包的分帧。在连续的数据包之间至少要有一个空闲的物理单元。

很多网络评价工作把链路按固定的成本处理。常识告诉我们，该成本随链路的长度和宽度的增长而增长。这个问题实际上是该领域争论的焦点之一，因为不同网络的相对质量取决于在评价中使用的成本模型。成本中的一大部分是连接器和连接它们所涉及的劳动力的成本，所以固定的成本是基本的。连接器的成本随宽度而上升，而导线的成本随宽度和长度增加。在很多情况下，关键的约束条件是链路集束的横截面积，比如在对分位置，成本随宽度而增加。 766

10.3.2 交换机

如图10-5所示，一个交换机由一组输入端口、一组输出端口、一个将每个输入与每个输

○ 这种链路内在的流控的观点与更传统的网络应用中的流控观点有相当不同，那里流控是由特殊数据包在链路层协议之上实现的。

出连接起来的内部“交叉开关”、内部缓冲器和在每个时间点影响输入/输出连接的控制逻辑组成。通常，输入端口的数量等于输出端口的数量，这称为交换机的度[⊖]。每个输出端口包含一个驱动链路的发送器。每个输入端口包括一个匹配接收器。大多数设计中的输入端口有一个同步器，它将输入的数据与交换机的本地时钟域对齐，这本质上是一个先进先出队列(FIFO)，所以自然要对每个输入端口提供某种程度的缓冲。输出端口也可能有缓冲，或者整个交换机共享一个缓冲。控制逻辑的复杂性取决于路由和调度算法，后面将讨论它们。至少，控制逻辑的复杂性必须能决定各个输入数据包所要求的输出端口，并且能在要求连接同一输出端口的多个输入端口之间做出仲裁。

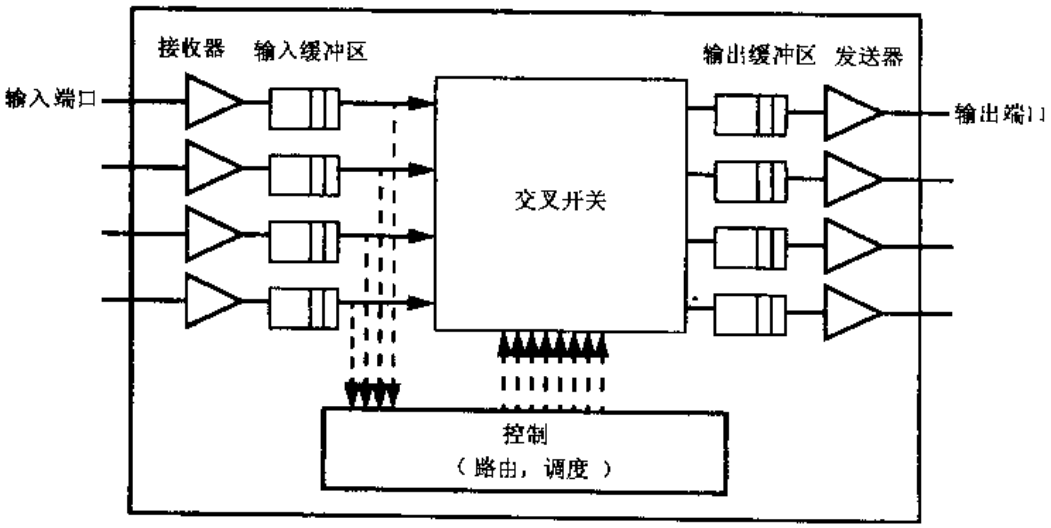


图 10-5　交换机的基本组织。一组输入端口通过交叉开关与一组输出端口相连接。控制逻辑在每个时间点影响输入/输出的连接

767

很多网络评价工作把交换机的度当作它的成本考虑。这显然是一个主要的因素，但也存在争论。交换机的某些部分的成本与它的度成线性关系，如发送器、接收器和端口缓冲。但是，内部连接的成本随它的度的平方而增长。内部缓冲的数量和路由逻辑的复杂性也超出线性增长。对现代的 VLSI 交换机而言，主要的约束条件是芯片的引脚，它与端口数量和每个端口的宽度的乘积成正比。

10.3.3　网络接口

网络接口（NI）包含一个或多个在通信辅助部件指导下发出数据包和从网络接收数据包的输入/输出端口，我们在前面的章节已经知道网络接口与处理节点相连。网络接口或者主机节点的行为与交换机节点有很大不同，它们可以通过特殊的链路连接。NI 组织数据包的格式，构造路由和控制信息。与交换机比较，它可以有较多的输入和输出缓存。它可以执行端到端的差错校验和流控。显然，它的成本受其存储容量、处理复杂度和端口数量的影响。

⊖ 和大多数规则一样，存在一些例外。比如，BBN Monarch 的设计使用了两种不同的交换机，它们有着不等数量的输入和输出端口（Rettberg et al，1990）。基于数据包头中的路由信息将数据包引导到其输出端口的开关所具有的输出可能多于输入。另一种叫做集中器的装置将数据包引导到任何输出端口，它的输出端口数量少于输入端口的数量，而且所有输出端口都通向同一节点。

10.4 互连拓扑结构

至此我们已经理解了决定网络的性能和代价的基本因素，能够考察与这些因素相关的设计空间的各个主要的维度了。本节的内容覆盖了一组重要的互连拓扑结构。每种拓扑实际上是一类随主机节点数量 N 扩展的网络，因此我们想知道作为 N 的函数的每一类关键特征。在实践中，像距离这样的拓扑特性并不是完全独立于像长度和宽度这样的物理特性的，因为某些拓扑结构在组合成物理体时，本质上需要较长的导线，所以对这两个方面都有所理解是重要的。

10.4.1 全连接网络

全连接网络本质上是把所有的输入与输出连接起来的单个交换机。其直径是 1 个链路，度数为 N。交换机的失效将使整个网络不复存在；但是，一条链路的失效只会丢失一个节点。这样的网络的一个例子是简单的总线，我们可以利用它来作为描述这种网络基本特征的参考点。它的一个良好的性质是其成本随规模的扩展按 $O(N)$ 上升。不幸的是，总线上同时只发生一次数据传输，所以整个带宽是 $O(1)$，对分带宽也是如此。事实上，带宽的扩展比 $O(1)$ 还差，因为随着端口的增加，RC 延迟使得总线的时钟频率下降。（以太网实际上是一个位串连的分布总线；它的工作频率足够低，从而允许大量的物理连接。）另一种全连接的网络是交叉开关。它提供 $O(N)$ 的带宽，但是互连的成本和交叉点的数量成正比，或者说 $O(N^2)$。在这两种情况下，全连接网络实际上都不是可扩展的。这并不等于说它们不重要。独立的交换机内部通常是全连接的，为更大的网络提供了基本的构造模块。网络技术进步的一个关键指标是效能成本合算的交换机的度。随着 VLSI 芯片密度的上升，能够被一个有着效能成本合算的交换机全连接的节点的数量正在上升。

768

10.4.2 线性阵列和环

最简单的网络是由双向链路连接连续编号的节点 0，…，$N-1$ 构成的线性阵列。其直径为 $N-1$，平均距离大约是 $2/3N$，去掉一条链路就分割了网络，因此它的对分宽度是 1 条链路。这样的网络中的路由是简单的，因为在任何一对节点之间只有一条路径。为了描述从节点 A 到节点 B 的路由，我们定义 $R=B-A$ 为从 A 出发到 B 的相对地址，这个带符号的长度为 $\log N$ 比特的数字是从 A 到 B 所跨越的链路数目，以离开节点 0 为正方向。因为在一对节点之间存在惟一的路径，该种网络显然不提供容错能力。该类网络由 $N-1$ 条链路组成，能很容易地在 $O(N)$ 的空间内以短导线进行布线。任何连续的节点段都是具有与整个网络相同拓扑结构的子网。

通过将阵列的两端简单连接就形成了环（ring）或者花环（torus）。采用单向链路，直径是 $N-1$，平均距离是 $N/2$，对分剖面宽度是 1 条链路，在任何一对节点间有一条路径。从 A 出发到 B 的相对地址是 $(B-A) \bmod N$。若采用双向链路，直径是 $N/2$，平均距离是 $N/3$，节点的度是 2，对分剖面是 2。在一对节点间存在两条路径（两个相对地址），因此网络能够在一条链路故障的情况下性能降级工作。如图 10-6 所示，通过将环对折，该网络能很容易地在 $O(N)$ 的空间内只使用短导线进行布线。可以将该网络分割为较小的子网，但是，子网成为线性阵列而不是环。

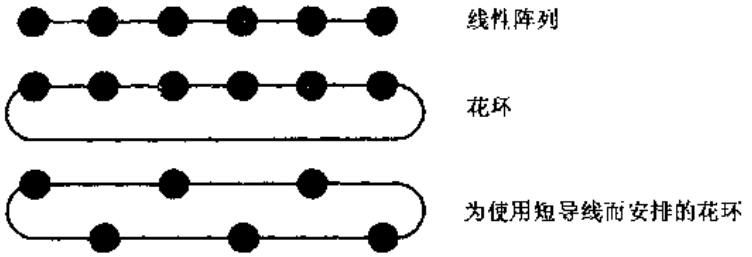


图 10-6　线性和环拓扑结构。线性阵列和花环易于用均匀的短导线来布线。距离和成本按 $O(N)$ 增长，而聚合的带宽仅仅为 $O(1)$

虽然这些一维的网络从实际意义上说不是可扩展的，但它们从概念上及在实践中是重要的构造模块。环的简单的路由和低硬件复杂性使得它们在局域互连中非常流行，包括 FDDI、光通道仲裁环（FiberChannel Arbitrated Loop）和可扩展的一致接口（Scalable Coherent Interface，SCI）。因为可以用非常短的导线对它们布线，因此有可能使链路变得非常宽。例如，KSR1 使用 32 节点的环作为构造模块，链路宽度为 128 比特。SCI 使用 16 比特宽的链路获得其带宽。

10.4.3　多维网格和多维花环

可以很自然地将环和阵列推广到更高的维数，包括 2D 的网格和 3D 的立方体，它们可以具有或没有首尾的连接。一个 d 维的阵列由 $N = k_{d-1} \times \cdots \times k_0$ 个节点组成，每个节点由它的 d 元坐标向量（i_{d-1}，…，i_0）来标识，这里 $0 \leqslant i_j \leqslant k_{j-1}$，$0 \leqslant j \leqslant d-1$。图 10-7 说明了二维和三维的一般形式。为简单起见，假定各维的长度相等，所以 $N = k^d$（$k = \sqrt[d]{N}$，$d = log_k N$）[⊖]。我们称之为 d 维 k 元网格。（在实践中，工程上的约束经常导致非均匀的维度，但我们可以很容易扩展该理论来处理那些情况。）每个节点是一个交换机，它通过以 k 为基的 d 维坐标的向量寻址，它所连接的节点的地址仅在一个坐标上相差 1。节点的度在 d 和 $2d$ 之间变化，包括 d 和 $2d$。网格中部的节点具有完全的度数，而位于角上的节点具有最小的度数。

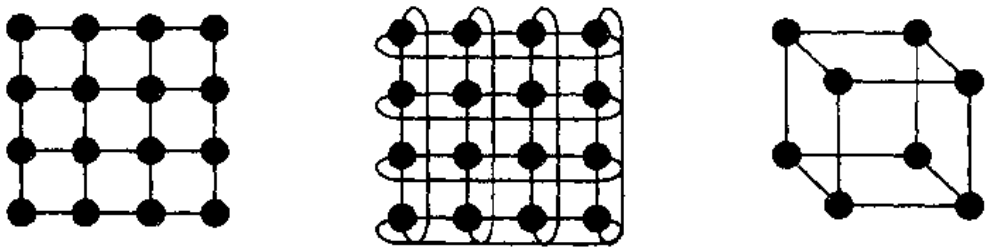


图 10-7　网格、多维环和立方体拓扑结构。网格、多维环和立方体是 k 元 d-立方体网络的特殊形式，k 元 d-立方体网络是 d 个维度每一个都由 k 个节点组成。可以用短导线在物理空间中很好地布置低维网络

对于一个 d 维 k 元的花环来说，边沿节点向各个面卷回，所以每个节点的度数都是 $2d$（在双向的情况下），并与各维中坐标相差 1（mod k）的节点相连接。我们将把阵列和多维花环统称为网格。d 维 k 元单向花环是一类非常重要的网络，通常叫做 k 元 d-立方体，它在现代并行计算机中被广泛使用。这些网络一般配置为直接网络，所以若要求每个交换机与主机

⊖ 原文是 $r = \log_d N$。——译者注

之间的双向连接，交换机需要增加一个额外的度。一般它们的度数不高，为 2 或 3，所以网络通过增加某些维或所有维的元数 k 来扩展。

769 ~ 770

为了定义在 d 维阵列中从节点 A 到节点 B 的路由，令 $R=(b_{d-1}-a_{d-1},\cdots,b_0-a_0)$ 为从 A 到 B 的相对地址。一条路径必须在各维 i 中跨越 $r_i=b_i-a_i$ 条链路，这里符号说明了适当的方向。最简单的方法是依次穿越各个维，所以对于 $i=0\cdots d-1$，在第 i 维穿行 r 跳，与此对应的是驾车在都市道路网格的两地间旅行，首先沿东西方向开，转一次弯，沿南北方向开到目的地。当然，我们也可以先沿南北再沿东西，或者走之字形的路径到达同一目的地。一般地，我们把源点和目的地点看作子阵列的角点，从源到目的地可以沿角点间的任何路径，只要每一跳都能对指向目的地的相对地址进行约减。

网络的直径是 $d(k-1)$。平均距离是各维的平均距离的和，大约是

$$d\frac{2}{3}k$$

如果 k 是偶数，d 维 k 元阵列的对分面有 k^{d-1} 条双向链路。这是简单地用一个与阵列某一维垂直的（超）平面切入阵列的中央而获得的。（如果 k 是奇数，对分面会稍大一些。）对于单向的花环，所有的节点的度数为 d（加上连接主机的度数），在各个方向上都有 k^{d-1} 条链路穿过网络中部。

显然，使用短线可以在平面的 $O(N)$ 大的空间内对二维网格布线，在自由空间的 $O(N)$ 体积中对三维网格布线。在实践中，工程因素会起作用，建造巨大的二维结构是不实际的，在如何利用三维空间方面会出现结构上的问题，例 10.1 说明了这些情况。

例 10.1　使用像 Intel 的 Paragon 机那样的直接二维网格拓扑，这里一个机柜内装 64 个处理器，构成宽为 4，高为 16 的节点阵列（每个节点包含一个消息处理器和 1～4 个计算处理器），如果只用短线，要建造一台大机器，你如何安排机柜？

解答：虽然有许多可能的方案，Intel 所用的方案在第 1 章的图 1-24 中已经说明。机柜竖立在地板上，大配置的方法是把这些机柜并排连接，形成一个 $16\times k$ 的阵列。最大的配置是 Sandia 国家实验室的 1 824 个节点的机器，构成一个 16×114 的阵列。对分带宽是由跨越机柜的 16 条链路决定的。■

其他的机器采取另外的策略来处理实际装配的限制。MIT 的 J-machine 是一个三维的花环，每一块主板包含了前两维中的一个 8×16 的花环。将这些主板一块块相邻堆叠，板间的连接提供了第三维的链路。具有 4 536 个计算节点的 Intel 的 ASCI Red 机有 85 个机柜。它允许长导线在各维中跨越机柜。对于高维的网格，一般是使用长线将几个逻辑维嵌入在各个物理的维中。图 10-8 显示了一个 $6\times 3\times 2$ 的阵列和嵌入在平面上的四维 3 元阵列。显然，对于给定的物理维度，导线的平均长度和导线的数量随逻辑维的增加而增加。

771

10.4.4　树

网格的直径和平均距离随 N 的 d 次方根增长。有许多其他的拓扑结构，它们的路由距离仅按对数上升。最简单的一种拓扑结构是**树**。一个二叉树的度数为 3。一般树作为间接的网络使用，以主机作为叶子，所以对 N 个叶子的树，直径是 $2\log N$。（这样的拓扑结构也能作为具有 $N=k\log k$ 个节点的直接网络使用。）在间接网络情况下，我们可以把各个节点的

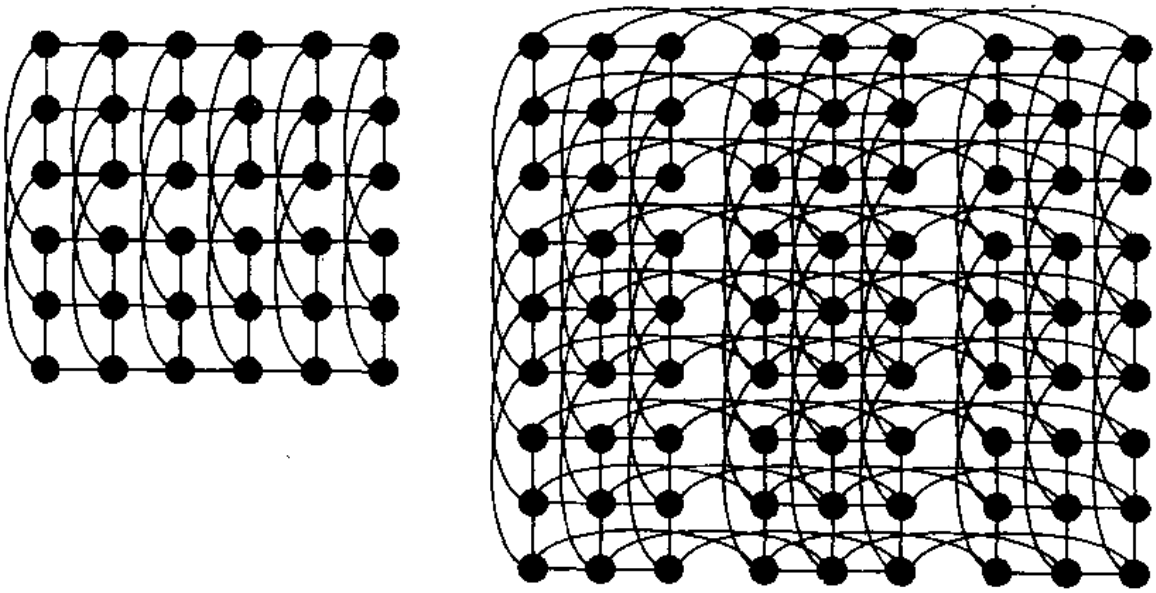


图 10-8　在二维物理空间嵌入多个逻辑维。一个较高维的 k 元 d-立方体可以二维布线，这是通过将二维的片重复布置，然后跨越剩余的维连接这些片而实现的。从图中我们很容易看出这些较高维的网络连线复杂度

二进制地址当作 $d=\log N$ 位的向量处理，它说明了从树的根到该节点的一条路径——最高位指示该节点是在根节点的左子节点还是右子节点之下，沿树的层次向下依次类推。树的层次直接对应网络的“维”。从节点 A 到节点 B 的一种路由方法是从 A 一直向上溯到根节点，然后沿着由 B 的地址说明的路径向下到 B。当然，我们实际上仅需要向上走到两个节点的第一个公共的父节点即可以向下走了。令 $R=B\oplus A$，即节点地址的按位异或形成 A 和 B 的相对地址，i 为 R 中为 1 的最高位的位置。从节点 A 到节点 B 的路径是向上 $i+1$ 跳，接着向下 $i+1$ 跳，在每个分支点所取的方向由 B 地址的低 $i+1$ 位说明。

[772]

以形式化方式描述，一个完全的间接二叉树是一个具有 $2N-1$ 个节点的网络，其节点组织成 $d+1=\log_2 N+1$ 层。主机节点占据第 0 层并由一个 d 位的地址 $A=a_{d-1}, \cdots, a_0$ 标识。一个交换机节点是由它的层次 i 和它的 $d-i$ 位地址 $A^{(i)}=a_{d-1}, \cdots, a_i$ 标识。交换机节点 $[i, A^{(i)}]$ 与父节点 $[i+1, A^{(i+1)}]$ 和两个子节点 $[i-1, A^{(i)} \| 0]$ 和 $[i-1, A^{(i)} \| 1]$ 相连接，这里垂直的双线“‖”代表按位的拼接。在任何一对节点间存在惟一的路径，即向上走到最近的公共祖先结点，因此，不存在容错能力。平均距离几乎和直径一样大，树可划分为子树。树的一个优点在于它能很容易地支持一个节点对多个节点的广播和多播操作。

显然，提高树的分支因子可以降低路由距离。在一个 k 元树中，每个节点有 k 个子节点，树的高度是 $d=\log_k N$；主机的地址由以 k 为基，长度为 d 的坐标向量来说明，它描述了从根节点向下的路径。

树的一个潜在问题是它们需要长线。不管怎样，当我们在平面上画一棵树时，靠近根方向的线的长度通常随层数按指数增长，图 10-9 的上半部分说明了这一情况，这导致在 $O(N\log N)$ 的布线中要使用 $O(N)$ 根长线。这实际上是你如何观察它的问题。使用递归的“H-树”模式，可以把同一个 16 节点的树紧凑地布置在二维空间，它允许在 $O(N)$ 的布线

中仅仅使用 $O\sqrt{N}$根长线（Bhatt and Leiserson 1982）。我们可以想象在芯片内的多个节点间，在主板上的节点（或子树）间，以及在机柜之间使用 H-树模式，但是主板之间必须使用线性布线。

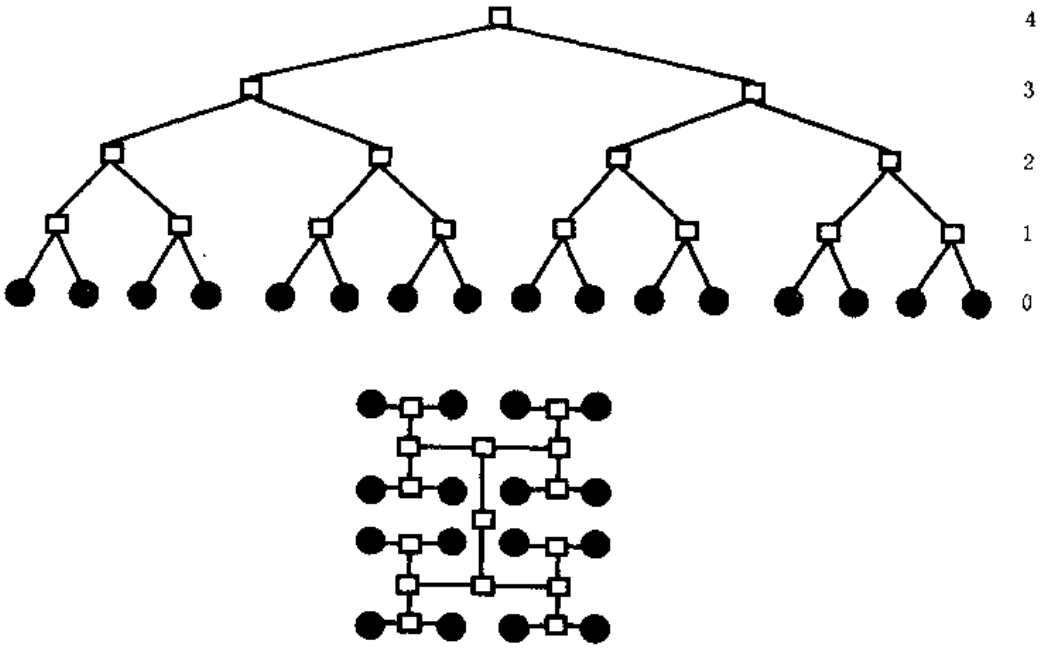


图 10-9　二叉树。树是具有对数深度的简单网络，使用 H-树可以在二维空间有效地布置树。树的路由简单，它们包含作为子网络的树。但是，对分带宽仅仅是 $O(1)$

树的更严重的问题在于其对分面。去除靠近根的单条链路会将网络对分。计算机科学家的树的概念很滑稽，真正的树是越靠近树干变得越粗。一个更好的比喻是人类循环系统，这里心脏构成根，细胞构成叶子。血液细胞向上经过静脉到达根，向下经过动脉到达细胞。穿过各个层次的带宽本质上是不变的，从而使血流均匀。树形网络的一个叫做胖树的有趣变型利用了这个概念，这里朝向父节点的向上链路的带宽是朝向子节点的链路带宽的两倍。当然，数据包和血液细胞的行为并不太相像，所以如何真正实现这种连线还要弄清楚一些问题。这些将很容易从蝶网（butterfly）得到结果。

10.4.5　蝶网

如果具有“许多根”的话，可以避免树结构在根位置的收缩。一种叫做蝶网的重要的对数型网络提供了这一特性。（蝶形拓扑出现在很多文献中。它是逐元素层次的 FFT、Batcher 奇偶合并排序和其他重要的并行算法的固有的通信模式。它与文献中包括 Omega 和 SW-Banyan 网络在内的一些拓扑结构是同构的，并且与混洗-交换网络和超立方体密切相关，我们将在后面讨论它们。）如图 10-10 的上部所示，对给定的 2×2 的交换机，线对中各引出一根线简单地交叉就获得了蝶网的基本构造模块。这成为对于相对地址的一位进行校正的工具，直传保持该位不变，交叉则使该位翻转。按图 10-10 下半部的 16 个节点蝶网所说明的方法，有规律地改变交叉的边沿引出线，可以将这些 2×2 的蝶形结构在一个 $N=2^d$ 个节点的网络中安排成 $\log_2 N$ 层的交换机。这种组态说明了一个具有向上单向链路的间接网络，主机把数据

包传入第0层，从第 d 层接收数据包。每一层都校正相对地址的一位。在第 d 层的每个节点都成为一棵树的根，该树以所有的主机为叶子，而从每个主机出发又是到达第 d 层的每个节点的一棵路径树。

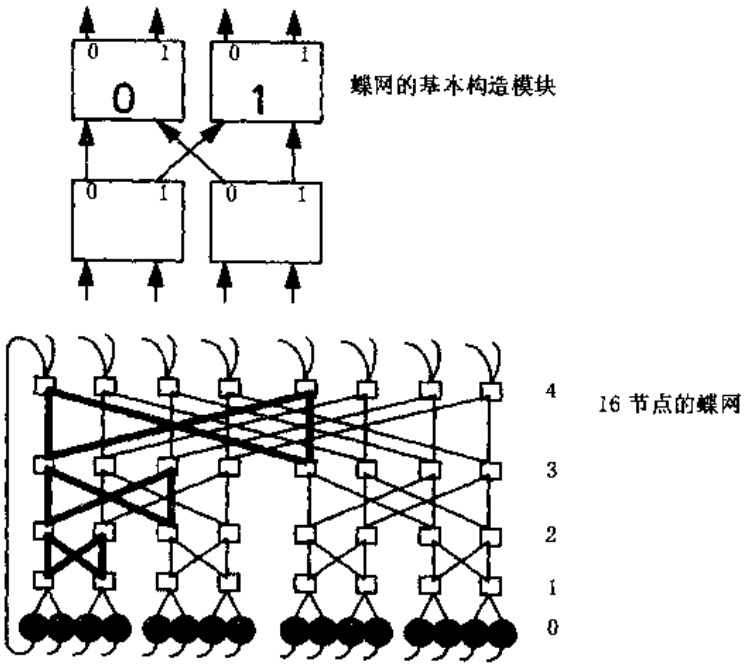


图 10-10　蝶网。蝶网是一个具有对数深度的网络，它由能够校正一位相对地址的 2×2 模块组成。我们可以把它看作一棵有多个根的树

一个 d 维的间接蝶网有 $N=2^d$ 个主机节点和 $d2^{d-1}$ 个度数为2的交换机节点，这些交换机组织成 d 层，每层具有 $N/2$ 个节点。一个位于层 i 的交换机节点［i，A］的输出与节点［$i+1$，A］和［$i+1$，$A\oplus2^i$］连接。为了找到从 A 到 B 的路径，计算相对地址 $R=A\oplus B$，在第 i 层，如果 r_i 是0，使用"直传"引线，否则，使用交叉引线。该网络直径是 $\log N$。事实上，所有的路径的长度都是 $\log N$。对分带宽是 $N/2$。（如果每个边沿交换机只接一台主机，公式略有区别，对分带宽为 N，但每层的交换机数加倍，层数多1。）

773
-
774

使用度数为 k 的交换机可以获得一个 d 维 k 元蝶网，k 应是2的幂。节点的地址是以 k 为基的 d 维坐标的向量，所以每一层校正相对地址中的 $\log_k N$ 位。在这种情况下，层次数为 $\log_k N$。事实上，这是把相邻的层次融合成为一个较高基数的蝶网。

从每个主机的输出到每个主机的输入只有一条路径，因此这种基本的拓扑不提供内在的容错。但是，与断开一条链路就使网络分割的1维网格和树不同，蝶网有容错的潜力。例如，从 A 到 B 的路径可能断开，但有另一条路径从 A 通到 C，从 C 通到 B。人们已经提出了许多通过增加少量额外的链路而使蝶网容错的建议。一个简单的方法是给蝶网增加额外的一层从而使每个源到每个目的地有两条链路。BBN T2000采用了这一途径。

蝶网似乎是一种比网格和树可扩展性更好的网络，因为每个数据包要跨越 $\log N$ 条链路，而网络中共有 $N\log N$ 条链路；因此平均来讲，所有的节点有可能同时向任何地方发送消息。与此对比，二维的花环或树每个节点只有两条链路，所以节点只能偶尔向远距离发送消息，邻近的节点非常少。关于对分带宽也有类似的争论。对于在 N 个节点间的数据的随机置换，

每个方向上穿过对分面的消息的期望数为 $N/2$。蝶网有 $N/2$ 条链路跨越对分面，d 维的网格穿越对分面的只有 [775]

$$N^{\frac{d-1}{d}}$$

条链路，而树只有一条链路穿越对分面。所以，当机器规模扩大时，蝶网的一个给定节点能每隔一个消息向机器另一侧的一个节点发送一个消息，二维网格中的一个节点只能每

$$\sqrt[d]{\frac{N}{2}}$$

个消息向另一侧的节点发送一个消息，而树只能每 N 个消息向另一侧的节点发送一个消息。

但是这一分析有两个潜在的问题。第一个是成本问题，对于树和 d 维的网格，网络的成本占机器成本的比例固定，因为对应每个主机节点有一个交换机和 d 条链路。对于蝶网，每个节点的网络成本随节点的数量增加而上升，因为对应每个主机有 $\log N$ 个交换机。所以，两者的可扩展性都不是完美的。实际中的问题是，相对于处理器而言交换机的成本如何？我们打算把机器整个成本的多大比例投入网络才能提供一定的通信性能？如果交换机和链路的成本是节点成本的 10%，那么对于 1 024 个处理器的机器，蝶网的成本仅仅是全部成本的一半。另一方面，如果交换机的成本与节点成本相同，我们就不太可能考虑二维以上的网络。相反，如果我们降低网络的维数，我们就可能为每个交换机投入更多。

第二个问题是，即使蝶网具有足够多的链路，能支持随机置换要求的带宽 × 距离的乘积，但蝶网的拓扑结构不允许 N 个节点之间 N 个消息的任意置换的无冲突路由。从一个输入到一个输出的一条路径阻塞了许多其他输入/输出对的路径，因为存在共享的交换机引出线。事实上，即使允许两次通过蝶网，也还存在着路由冲突的置换。但是，如果把两个蝶网背靠背地放置，从而消息正向通过一个，反向通过另一个，那么对于任何置换，总是存在一种允许无冲突置换路由的中间位置的选择。这种背靠背的蝶网叫做 Benes 网络（Benes 1965；Leighton 1992），由于其优美的理论性质，人们对它曾经开展了广泛的研究。但我们常常发现它的实际重要性并不大，因为计算中间位置的成本太高，而且必须预先知道置换。另一方面，另一个有趣的理论结果揭示在蝶网上可以（以高概率）找到冲突非常少的任何置换的路由；做法是把每个消息首先发往一个随机的中间节点，然后再把消息路由到希望的目的节点（Leighton 1992）。在下面介绍的胖树中，这两种结果以非常精巧的实际方式走到了一起。

图 10-11 显示，将一个 d 维 k 元的 Benes 网络从高阶维度向本身折回就形成了一个 d 维 k 元胖树。在第 i 层的 N/2 个交换机的集合被看作 N^{d-i} 个“胖节点”，每个胖节点包含 2^{i-1} 个交换机。正向蝶网的边沿向上通向根，反向蝶网向下通往叶子。为了从 A 到达 B，在 A 和 B 最近的公共祖先胖节点中随机选中一个节点 C，沿惟一的树路径从 A 到达 C，再沿惟一的树路径从 C 向下到达 B。令 i 为区别 A 和 B 的最高的维，那么有 2^i 个根节点可供选择，所以路由距离越长，越能把流量分布开。这种拓扑结构显然有大量容错路径，它具有蝶网的对分带宽，树的分割性质，允许所有的置换路由而只发生非常少的竞争。它在 Connection Machine CM-5 和 Meiko CS-2 中得到应用。在 CM-5 中，向上路径的随机选择是由交换机动态完成的；在 CS-2 中，由源节点选择其祖先节点。节点的度数固定，与网络的规模无关是蝶网和胖树的一个特别重要的实用性质。这个性质允许我们用同样的交换机构造任意大小的网 [776]

络。如图 10-11 所示，胖树或任何与蝶网类似的网络的较高层次的实际连线复杂度成为问题的关键，因为需要大量的长线连接不同的位置。

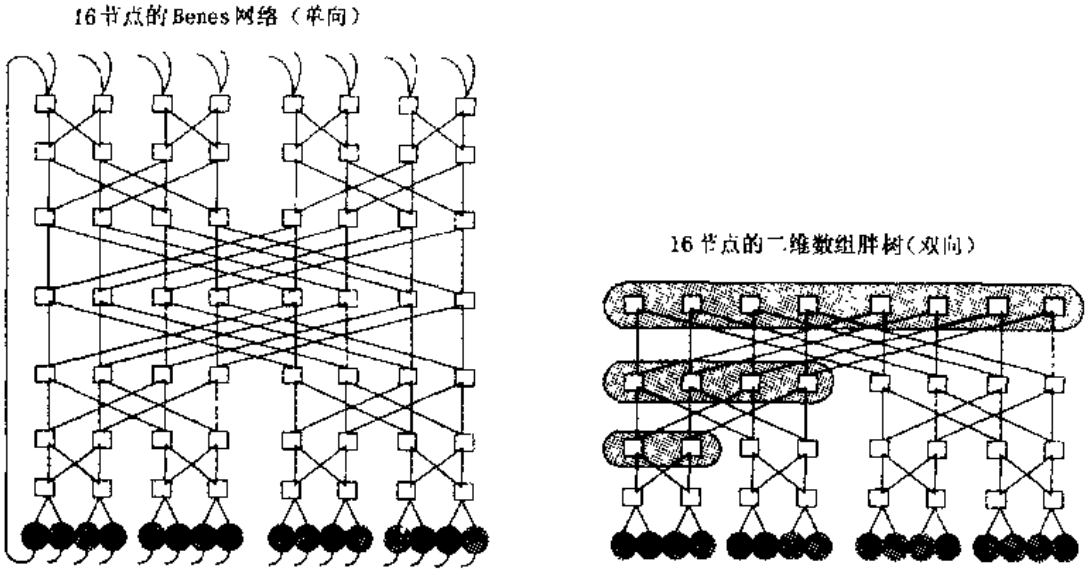


图 10-11 Benes 网络和胖树。一个 Benes 网络本质上是通过把两个蝶网背靠背连接而构成的。它的一个有趣的性质是：如果可以脱机计算路由的话，它能以无冲突的方式实现任何置换路由。两个正向的蝶网合起来并不具有这个性质。把 Benes 网络的第二个一半折回它本身并融合两个方向就得到了一个胖树，这样它能在任何层转向、交换机的组合成为胖节点

777

10.4.6 超立方体

蝶网中的直传引线还有优化的潜力，因为它们将一个数据包直传到同一列的下一层。让我们考虑如果把一列中的所有交换机压缩成一个度数为 $\log N$ 的交换机会发生什么情况。这会带给我们闭合的回路，成为一个 d 维 2 元花环！实际上，我们需要把一列中的交换机分成两半，并将它们与相邻两个的节点结合。这叫做超立方体或二进制 n-立方体。其 $N=2^d$ 个节点中的每一个都与另外 d 个节点相连接，它的地址与这 d 个节点的地址只有一位不同。相对地址 $R(A, B)=A \oplus B$ 规定了从 A 到 B 必须穿越的维度。显然，路径的长度等于相对地址中的 1 的个数。可以按任何次序校正维度（对应跨越了立方体对角的不同的路径），蝶网的路由正好与维序路由对应，这在超立方体的文献中称为 e-立方体路由。胖树路由则是在由相对地址最高位 1 所定义的子立方体中随机选取一个节点，把数据包向“上”发送到该随机节点，然后掉头向“下”到达目的地。注意胖树在两个方向上使用不同的链路集合，在超立方体中，为了获得相同的性质，需要在节点间使用一对双向链路。

超立方体是一种重要的拓扑结构，因此引起理论文献的广泛的注意。例如，通过选择节点的适当标记，可以把低维网格一一对应地嵌入到超立方体中。让我们回忆一下数字设计的方法，格雷码序列按某种规则排列从 0 到 2^{d-1} 的数字，使得相邻的数字只有一位不同。这说明了如何把一维的网格嵌入到 d-立方体，而且可以扩展到任意的维数（见习题 10.7）。显然，蝶网、混洗-交换网等也能容易地嵌入。（有趣的是，一个 d-立方体无法正好嵌入到一个 $d-1$ 层的树，因为在 d-立方体中多一个节点。）

其实，超立方体只被用于很多早期的大规模并行计算机，包括 Cal Tech 的研究原型（Seitz 1985）、Intel 的前三代 iPSC 机（Ratner 1985）和三代 nCUBE 机。较晚的大规模机器，包括 Intel 的 Delta、Intel 的 Paragon 和 CRAY T3D，都使用低维的网格。这种转移的原因之一是，在实践中，超立方体拓扑强迫设计者使用具有能支持最大可能配置度数的交换机。在较小的配置中，端口被浪费了。K 元 d-立方体的方法提供了实用的可扩展性，它允许使用指定的部件集，也就是说，使用度数固定的交换机来构造任意大小的配置。然而，这带来了下述问题：什么是需要的度数？

并行机网络设计的一般趋势是使用能用于任何拓扑结构连接的交换机。例如，我们在 IBM 的 SP-2、SGI 的 Origin、Myricom 网络和大多数 ATM 交换机中看到了这一点。设计者可以选择某一特定的规则的拓扑结构，或把不同尺寸的配置按不同方法连接在一起。在任何时候，像引脚引出和芯片面积这样的技术因素限制了最大可能的度数。 778

10.5 对网络拓扑设计折中的评价

k 元 d-立方体提供了一个评价直接网络不同设计方案的方便的框架。我们可以以两种方式提出设计问题。给定选择的维数，交换机的设计确定，我们可以探求机器规模如何扩展。或者，对于感兴趣的机器规模，即 $N = k^d$，可以问一下在明确的成本约束条件下，最好的维数是什么。2D 的花环处于一个极端，超立方体处于另一个极端，在这之间是网络的谱系。和体系结构的大多数方面一样，评价设计取舍的关键是定义成本模型和性能模型，然后根据它们优化设计。网络拓扑在并行体系结构的历史上是有争论的热点。在很大程度上，是因为在不同的成本模型下不同的选择有其意义，而技术一直在进步。一旦决定了交换机的维（或度数），可供选择的网络类型空间受到相对的约束，所以问题在于值得努力去获得的度有多大。

让我们把对 k 元 d-立方体这类网络的参数收集到一起。不管其度数是多少，交换机的总数是 N；但是，交换机的度数是 d，所以链路的总数是 $C = Nd$，每个节点有 $2wd$ 个引脚。平均路由距离是

$$d\left(\frac{k-1}{2}\right)$$

直径是 $d(k-1)$，从各个方向穿越对分面的链路条数是 $k^{d-1} = N/k$（k 为偶数）。所以有 $2Nw/k$ 根导线穿过网络的中部。

如果我们主要关心路由距离，那么我们倾向于使维数最大来建造一个超立方体。这可能是存储－转发路由的情况，假设交换机的度和链路的数量不是主要的成本因素。此外，我们还可以拥有它精巧的机械性质。因此，这是大多数第一代大规模并行机选择的拓扑结构。但是，对于直通路由和更实际的硬件成本模型，这个选择的理由就不那么明显了。如果链路的数量或交换机的度是占主导地位的成本，我们倾向于使维数最小而建造一个网格。为了使评价有意义，我们希望比较成本大致相同的不同设计的性能。关于系统的哪一方面成本最高的不同假设会导致非常不同的结论。

假设的通信模式也影响设计决策。如果观察一下各种网络的最坏的流量模式，我们倾向于高维网络，其所有路径都是短的。如果我们观察各个节点只与一个或两个相邻节点通信这

779 样的模式，我们将选择低维网络，因为只有少数的维度被实际使用。

10.5.1 无负载时延

图 10-12 显示了使用直通路由，当机器的规模扩展时，二维、三维和四维立方体以及二进制 d-立方体（$k=2$）的平均无负载时延的增加。它假定每级为单位路由延迟（$\Delta=1$），消息大小为 40 和 140 字节，$w=1$ 字节。底部的线是由通道占用度产生的时延部分。正如我们所预料的，对较小的消息（或每级较大的路由延迟），低维网络的扩展性较差，因为消息平均要经历较多的路由步。但是，当我们比较图中的曲线时，实际默认交换机的度的区别不是系统成本的主要成分。此外，1 个周期的路由是非常冒进的假设，对高性能交换机而言，更为典型的值是 4～8 个网络周期（见表 10-1）。另一方面，较大的消息也是常见的。

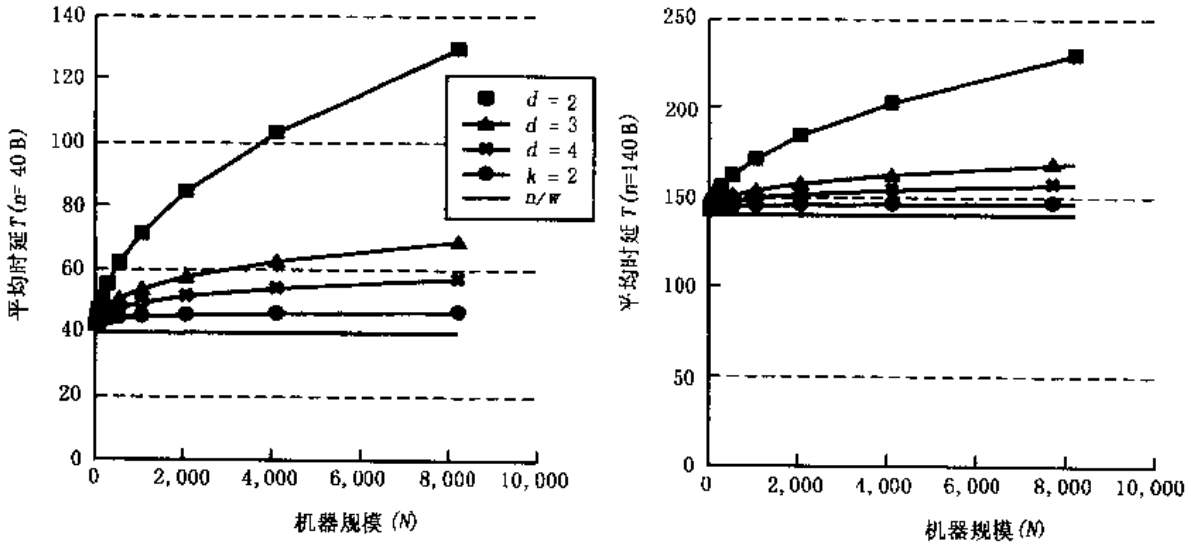


图 10-12 具有固定链路宽度不同维度的网络的无负载时延的尺度。线 n/w 说明消息传输的通道占用度分量，即消息的比特通过单个通道的时间，它与网络的拓扑无关。曲线显示了由路由引起的额外时延

表 10-1 各种并行机网络的链路宽度和路由延迟

机器	拓扑	周期时间（ns）	通道宽度（位）	路由延迟（周期）	Flit（数据位）
nCUBE/2	超立方体	25	1	40	32
TMC CM-5	胖树	25	4	10	4
IBM SP-2	Banyan	25	8	5	16
Intel Paragon	2D 网格	11.5	16	2	16
Meiko CS-2	胖树	20	8	7	8
CRAY T3D	3D 花环	6.67	16	2	16
DASH	花环	30	16	2	16
J-Machine	3D 网格	31	8	2	8
Monsoon	蝶网	20	16	2	16
SGI Origin	超立方体	2.5	20	16	160
Myricom	任意	6.25	16	50	16

为了把我们的注意力集中到网络的维数这一设计问题上来，我们把成本模型和反映我们设计出发点的节点数量固定，考察以固定成本和不同的 d 情况下网络的性能特征。图 10-13 显示了在四种规模的机器的条件下，短消息的无负载时延作为维数的函数的情况。对大的机器，路由延迟在低维网络中起主导作用；对较高的维度，时延趋近于通道时间。这种“相同节点数量”的成本模型被广泛使用，以支持低维网络扩展性不好这一观点。

780 ~ 781

在图 10-13 的成本模型下，高维网络的优越性能并不令人惊奇，因为在这个模型下，交换机的度、通道数量和通道长度的增加无须付出代价。高维网络的连线和引脚的数量比低维网络多得多，其交换机也比低维网络的交换机要大。对于图 10-13 中曲线最右端所代表的网络设计是不实际的。网络的成本占了大规模并行机成本的很大部分，所以在适当的技术假设下比较成本相等的设计才是有意义的。随着芯片的尺寸和密度改善，交换机内部组成不再是主要的成本部分，而引脚和连线仍然是关键因素。在极端情况下，导线的物理体积成为限制可能实现的互连的根本因素。所以让我们在物理连线复杂度相等的假设下比较这些网络。一种有意义的比较是保持每个节点的连线总数为常数，也就是说，固定引脚的数量为 $2dw$。让我们以通道宽度 $w=32$ 的二维立方体为基准，这样每个节点就有 128 条连线。如果维度提高，通道增加，每个通道的宽度必须降低。通道宽度可记为 $w_d=\lfloor 64/d \rfloor$。所以对于一个 8 维立方体，链路就只有 8 位宽了。我们假设消息长度为 40 和 140 字节，每跳的路由延迟均匀，为两个周期，周期时间均匀，图 10-14 显示了在同等引脚规模情况下的无负载时延。该图揭示了一种完全不同的现象。随着维数的增加，通道变窄，通道时间增大；这抵消了由较小的路由距离所带来的路由延迟的降低。非常大的配置在低维情况下仍然会有大的路由延迟，不管通道的宽度如何，但是所有的配置在维数较低点都达到最佳的无负载时延。

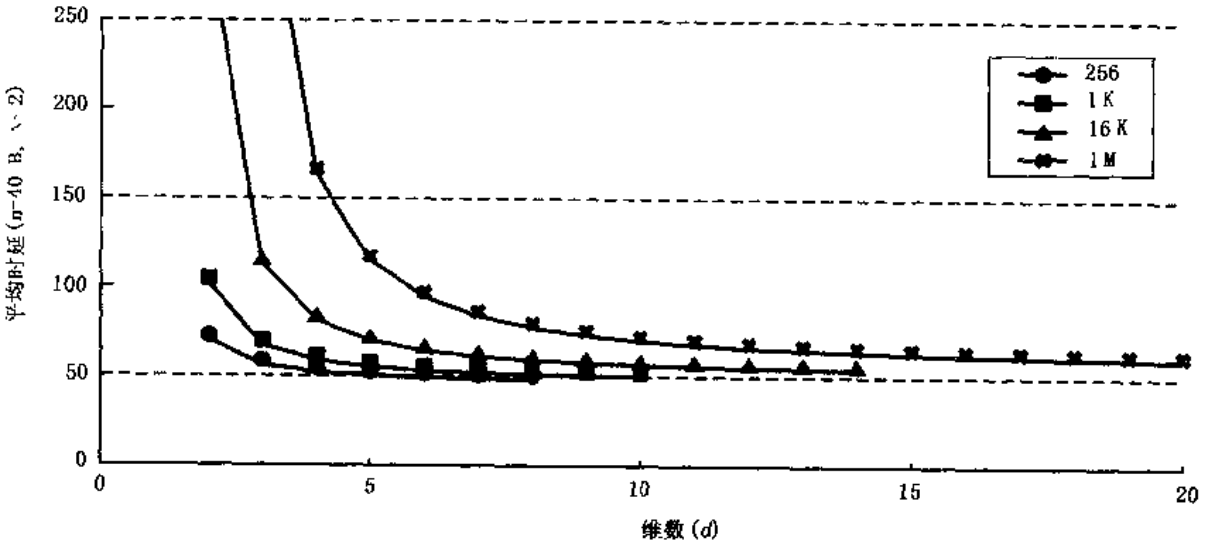


图 10-13 节点数量相同（$n=40$ B，$\Delta=2$）的 k 元 d-立方体情况下，作为度数函数的无负载时延。当链路宽度和路由延迟固定，由于较长的路由距离，大规模网络的无负载时延在低维处急剧上升

如果引脚不是设计的限制因素，那么穿过机器中央的连线数量就成为连线复杂度的关键。如果把机器看作在平面上展开，对分面的物理宽度仅随面积的平方根而增长；在三维空间，它随体积的三分之二次方增长。即使网络具有高的逻辑维数，它也必须嵌入在少量的物理维度中，因此设计者必须为穿过中间层面的导线所占的横截面积而费心。

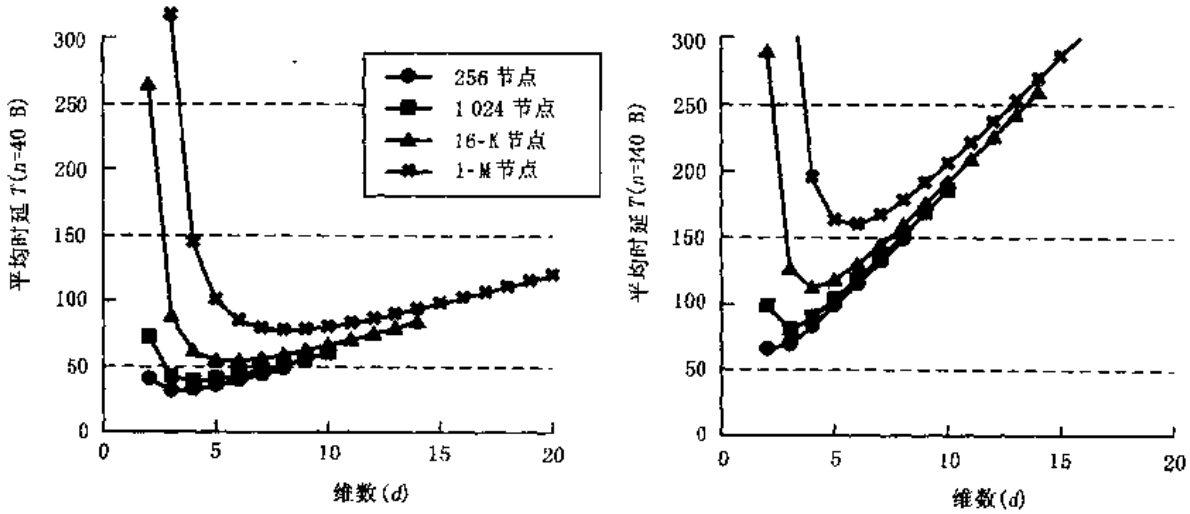


图 10-14　具有相等引脚数的 k 元 d-立方体的无负载时延（$n=40B$ 和 $n=140B$，$\Delta=2$）。在相等引脚数情况下，较高的维度意味着较窄的通道，所以最优的设计点是平衡路由延迟（随维数降低增加）和通道时间（随维数上升增加）

我们可以通过比较有相同数量导线穿过对分面的设计来关注成本的这个侧面。在一个极端，超立方体有 N 条这样的链路。让我们假设这些链路都具有单位尺寸。一个 2D 的花环只有 $2\sqrt{N}$ 条链路穿越对分面，所以每条链路的宽度可以是超立方体所用链路宽度的 $\sqrt{N}/2$ 倍。在对分面相等的标准下，具有位串行链路的 1 024 节点的超立方体应该是和使用 32 位宽链路的相同节点规模的花环相比较。一般来说，与 N 个节点的超立方体具有相同对分面宽度的 d 维网格的链路宽度是 $w_d=\sqrt[d]{N}/2=k/2$。假设直通路由，在一个无负载网络上一个 n 字节的数据包到达一个随机目的地的平均时延是

$$T(n,N,d)=\frac{n}{w_d}+\Delta\cdot d\left(\frac{k-1}{2}\right)$$

[782]

$$=\frac{n}{k/2}+\Delta\cdot d\left(\frac{k-1}{2}\right)=\frac{n}{\sqrt[d]{N}/2}+\Delta\cdot d\left(\frac{\sqrt[d]{N}-1}{2}\right)\tag{10-7}$$

所以，提高维数会降低路由延迟但增加通道时间，这和引脚数相等的情况一样。（读者可以验证当这两项基本上相等时达到最小的时延。）

图 10-15 显示了在假定 $\Delta=2$ 情况下，长度为 40 字节的消息的平均时延，它是一系列不同规模的机器的维数的函数。当维数从 $d=2$ 开始增加时，路由延迟急剧下降，而通道时间随着链路变细持续上升（对于 $N=1$ M 个节点的情况没有显示 $d=2$ 这个点，因为它是很可笑的，链路会有 512 位宽，而平均级跳数高达 1 023。）对于具有几千个以内节点的机器而言，$\sqrt[3]{N}/2$ 和 $\log N$ 非常接近，所以增加通道对通道宽度的影响就成为起主导作用的因素。如果考虑大尺寸的消息，路由的成分就更不显著。对于大机器来说，在这样的衡量规则下低维网格就变得不实际了，因为链路将变得非常宽。

到目前为至，我们关心的是连线所占的横截面积，但是还没有考虑连线的长度。如果在一个平面上嵌入 d-立方体，也就是说，在每个物理维度上嵌入 $d/2$ 个维度，并使节点的中心距不变的话，那么每增加一个维度都会使最长的连线加长 $\sqrt{k}$ 倍。所以，d-立方体中最长

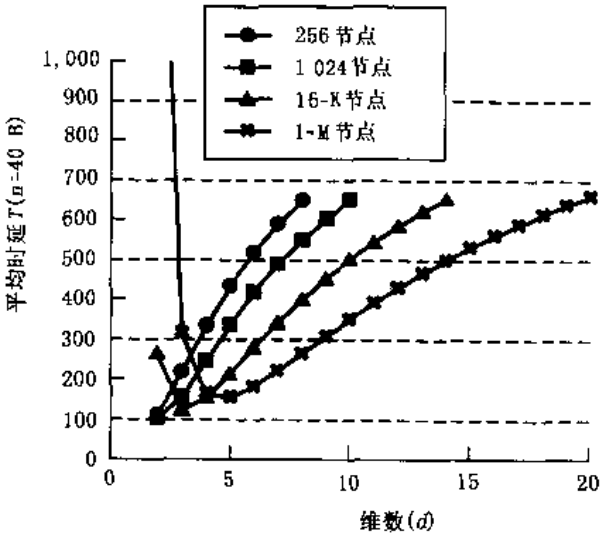


图 10-15　具有相等对分宽度的 k 元 d-立方体的无负载时延（$n=40$ B，$\Delta=2$）。在相等对分面宽度的衡量规则下，路由延迟和通道时间之间的平衡使低维的网络更为优越

的连线的长度是二维立方体中的 $k^{n/2-1}$ 倍。计算增加的连线长度进一步加强了采用较低维度的论点。这种计算可以以三种方式进行。如果我们假定在一条导线上多个位形成流水，那么长度增加实际上提高了路由延迟。如果导线上信息不是以流水的方式传送，那么网络的周期时间随导线驱动时间的加长而增加，导线驱动时间与导线的长度成对数关系。 783

将 d 维网络嵌入较低物理维度产生了能增强低维效益的辅助效应。如果把一个高维网络系统地嵌入到一个平面上，连线的密度在靠近对分面处最高，在靠近周边处最低。一个二维的网格各处的连线密度均匀，所以它更好地利用了它所占据的面积。

我们也应该从带宽的角度来观察网络设计中的折中。在相同连线复杂度的尺度下影响延迟的关键因素是低维网络通道带宽的增加。如果大多数的流量来自或发往一个或少数节点，较宽的通道是有益的。如果流量是本地化的，因而每个节点只与几个邻近节点通信，只有几个维度被利用，较高的链路带宽仍然起主导作用。如果很多节点穿越整个机器进行通信，那么我们需要根据观察到的时延对竞争效应建立模型，并观察网络在哪里饱和。

在结束对无负载情况下时延的折中措施的考察之前，我们应该注意到这种评价对于通过一条导线和穿过一个交换机的相对时间相当敏感。图 10-16 说明，如果每个交换机的路由延迟 20 倍于导线的传输延迟，情况就会大不一样。这就是 SGI Origin 为什么使用较高维度网络的原因。 784

10.5.2　负载情况下的时延

为了分析负载情况下网络的行为，需要捕捉拥塞对所有其他正在流过网络的通信流量的效应。这些效应可能是微妙而难以觉察的。让我们回到交通系统的比喻，例如，当你下一次驶入有交通负载的高速公路时，请注意在两条高速路汇合然后又分开的地方，交通的拥塞要比经过一系列斜线合并进入单条高速路时更糟糕。在交汇点，司机之间的交互要多得多，当达到某种交通负荷的水平时，整个交通就停下来了。网络的行为与此类似，但它的交汇点要多得多。为了对各种拓扑结构评价这种效应，必须确定通信模式、路由算法、流控策略以及

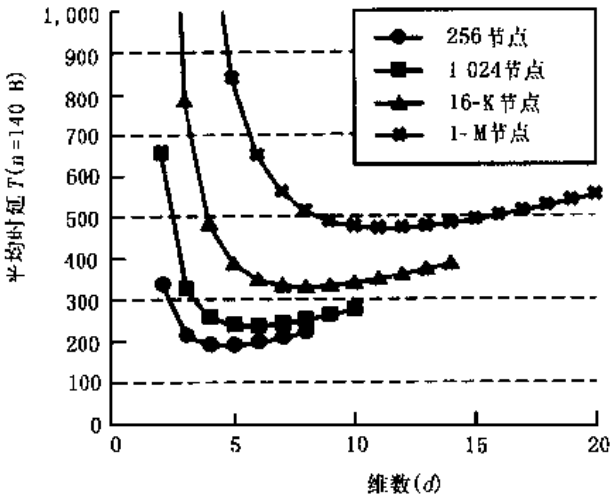


图 10-16 具有相等数量的引脚和较大路由延迟的 k 元 d-立方体的无负载时延（$n=140$ B，$\Delta=20$）。在实践中我们常发现，通过交换机的时间比通过导线的时间大得多，此时具有较高度的交换机更有吸引力

交换机内部设计的许多细节。我们可以为系统开发一个排队模型，或者针对所建议的那些设计建立仿真器。和计算机设计的其他大多数方面一样，其技巧在于，所发展的模型应足够简单，以便在适当的设计层次上提供直观的信息，但又应足够精确，以便在改进设计的过程中提供有用的指导。

对于使用维序的直通路由和无限内部缓存的 k 元 d-立方体，我们使用 Agarwal（1991）所发展的封闭式的竞争延迟模型，因此不会出现流控和死锁的问题。该模型的预测与对于满足相同假设的网络仿真的结果有很好的对应关系。这个模型是基于 Kruskal 和 Snir（1983）早期对间接（Banyan）网络性能建模的工作。省略推导过程，其主要的结果是能为 k 元 d-立方体上大小为 n 的消息的随机通信建立时延模型，这里通道宽度为 w，负载水平对应于聚合通道利用率为 ρ，模型表示为：

785

$$T(n,k,d,w,\rho)=\frac{n}{w}+h_{ave}(\Delta+W(n,k,d,w,\rho))$$

这里

$$W(n,k,d,w,\rho)=\frac{n}{w}\cdot\frac{\rho}{1-\rho}\cdot\frac{h_{ave}-1}{h_{ave}^2}\cdot\left(1+\frac{1}{d}\right) \tag{10-8}$$

并且这里

$$h_{ave}=d\left(\frac{k-1}{2}\right)$$

使用这个模型，我们能比较各种尺寸的低维或高维网络在有负载情况下的时延，这正如我们对无负载时延所做的一样。图 10-17 显示了一个 1 024 节点的 32 元二维立方体和一个 1 000节点的 10 元三维立方体上的预测的时延，它是所请求的聚合通道利用率的函数，假设通道宽度相等，消息的尺寸较小，为 4、8、16 和 40 个物理单元。我们能从曲线的右端看到，这两种网络在大致相同的通道利用率处饱和；但是，饱和点随消息尺寸而快速下降。曲线的左端表示无负载时延。在相同的通道时间条件下，度数较高的交换机有着较低的基本路由延迟，这是因为跳数较少，通道宽度相等。当负载增加时，这种区别变得不那么显著了。

请注意与无负载时延相比，竞争延迟有多大。显然，为了给用户程序提供低时延的通信，重要的问题是机器的设计应保证网络不会轻易地进入饱和，这可以通过提供过剩的网络带宽或调节处理器的负载完成。

图 10-17 中的数据提出了网络设计的一个基本的折中问题。网络上的数据包应该被设计成多大？数据清楚地说明网络对小数据包的传送效率比对大数据包要高。但是，较小的数据包的利用率较差，这是因为每个数据包都要包含路由和控制信息，对相同量的数据传输而言，会产生更多的网络接口事件。对于任何给定的技术和详细的设计，存在一个最优点。

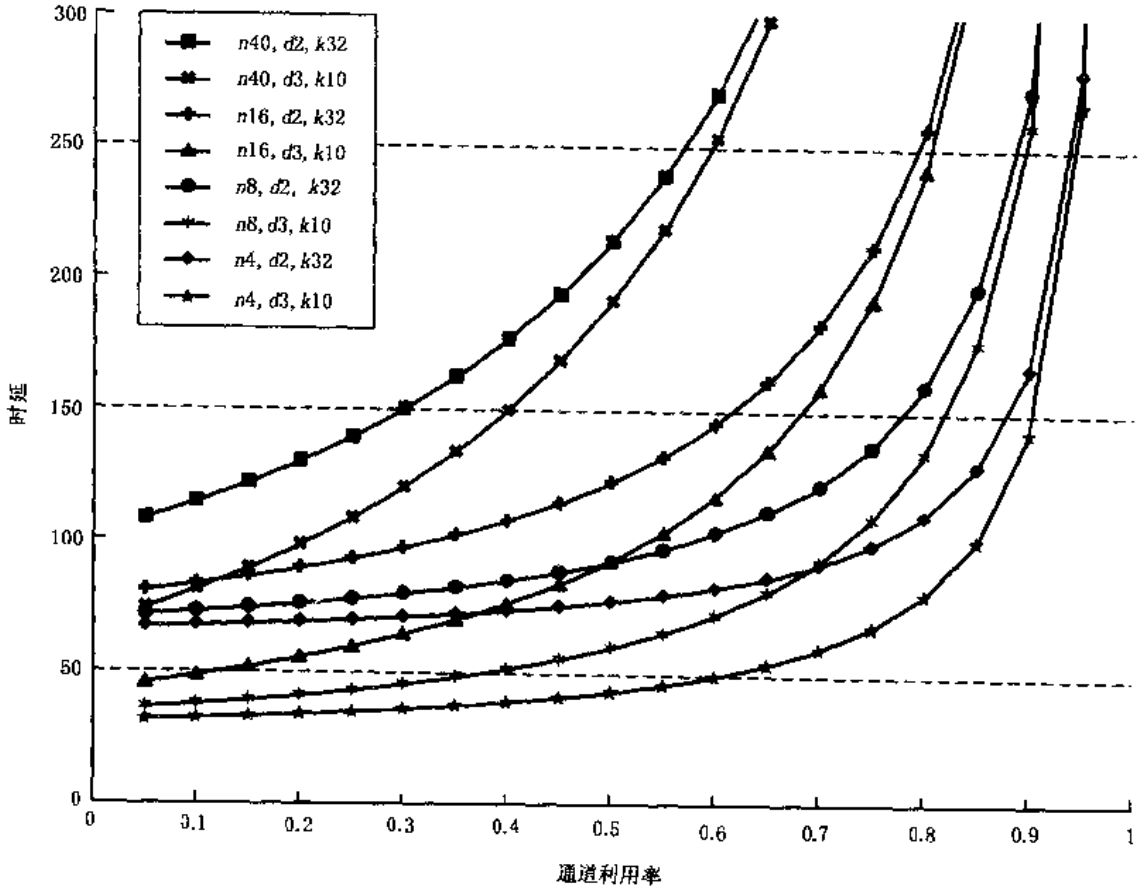


图 10-17 路由延迟为 2 的 32 元二维立方体和 10 元三维立方体的竞争时延与负载的比较。以较低的通道利用率，较高维的网络在通道宽度相同情况下的时延低得多，但是随着利用率的上升，它们向同一饱和点收敛

我们在根据图 10-17 得出关于网络维数的选择的结论时必须小心。图中的曲线显示了各种网络利用它可用的通道集合的效率。该图的结论似乎是两者以大致相同的效率使用它们的通道。但是，较高维的网络的每个节点可用的带宽要大得多；其每个节点拥有的通道为较低维网络节点的 1.5 倍，而每个消息使用较少的通道。在一个 k 元 d-立方体中，随机通信情况下每个周期可用的物理单元（phit）数是

$$\frac{Nd}{d\frac{(k-1)}{2}}$$

[786] 或者每周期每节点 $2l(k-1)$ 个物理单元（每周期 $2w/k-1$ 位）。假定通道带宽相等，我们的例子中的三维立方体有 4 倍之多的可用带宽，而通道利用率相同。所以，正如图 10-18 所示，如果我们观察与可提供带宽相对照的时延，情景就会大不一样。二维立方体从一个较高的基础时延开始，在三维立方体感觉到负载之前就饱和了。

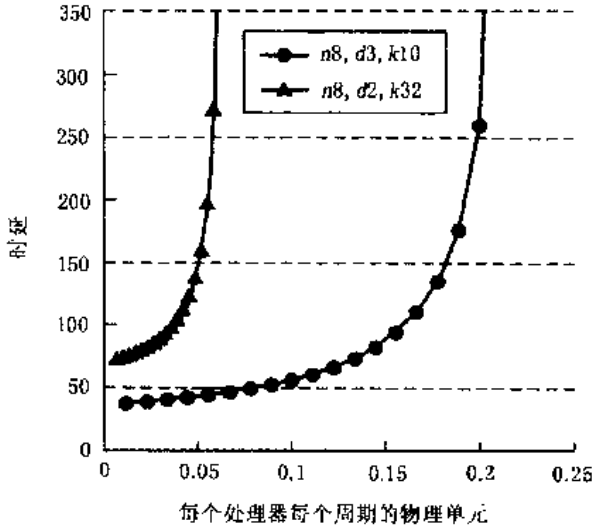


图 10-18　竞争情况下时延与每周期物理单元数的对照。在每通道具有相等的平均流量条件下，比较路由延迟为 2 的 32 元二维立方体和 10 元三维立方体，结果显示较高维数的网络在饱和之前能处理较大的负载

这一比较使我们回到什么是合适的相等成本比较的问题。作为一个习题，你可以研究这些曲线做出相等引出管脚和相等对分面的比较。加宽通道降低了通道时间从而使基础时延下移，提高了总体可用带宽，因为对每个数据包服务得更快而降低了在各交换机处的等待时间。所以，结果对于成本模型是相当敏感的。

[787]

当通道宽度相对维数变化时产生了一些有趣的现象。例如，使用相等对分面规则，每个节点的容量是

$$C(N,d) = W_d \frac{2}{k-1} \approx 1$$

随机业务量的聚合容量本质上与维数无关！各个主机在每个网络周期内平均产生不到 1 位。这一观察产生了低维网络的新的前景。一般来说，几个节点中的每一个都必须沿一个方向将消息引导相当长的距离。所以，各个节点必然不是频繁地送出数据包。在固定对分宽度的假设下，低维情况下的通道成为几个节点共享的资源，而高维网络为各个不同的维的流量划分带宽资源。在低维情况下，当一个节点使用一个通道时，它只在较短的时间内使用它。在大多数系统中，轮询比划分能产生更好的利用率。

在现代的机器中，节点占据的面积比连线的剖面所占的面积大得多，而机器的规模通常限制在几千个节点之内。在这样的体制下，机器的对分面通常由电缆集束构成。这是个工程上的挑战性问题，但并不是对机器设计的根本性的限制。正如表 10-1 所说明的那样，一个趋势是在具有较短连线的拓扑结构中使用更宽的链路和更快的信号，但是其效果并不像相等

对分面扩展规则所建议的那么显著。 788

10.6 路由

让我们回忆一下，网络的路由算法决定了使用源到目的地所有可能的路径中哪一条作为路由，也决定了各个特定数据包如何遵循该路由。比如，我们已经了解到在一个 k 元 d-立方体中，一组最短的路由完全是由源和目的地的相对地址所描述的，它说明了在各个维度上需要穿越的链路的数量。维序的路由限制了合法路径的集合，从而从每个源节点到每个目的节点只有一条路径，即首先沿低序维行进适当的距离，然后再沿下一个维行进，依次类推。本节将描述在现代机器中使用的不同级别的路由算法以及好的路由算法的关键性质，例如生成一组免死锁的路由，维持低时延，均匀分布负载和容错等。

10.6.1 路由机制

让我们从基本概念谈起。回忆一下交换机的基本操作是监视在其输入到达的数据包，对每个输入数据包，选择一个将其送出的输出端口。所以，路由算法是函数 R：$N \times N \rightarrow C$，它在各个节点将目的节点 n_d 映射到路由上的下一个通道。高速交换机基本上使用三种机制从数据包头的信息来决定输出通道：算术运算的、基于源的端口选择和查表。在并行计算机的网络中，交换机应该能在几个周期内对它的所有的输入做出路由的决定，所以这种机制应该是简单快速的。

对于大多数规则的拓扑结构而言，简单运算操作足以选择输出端口。例如，在 2D 网格中，可以由每个数据包在其数据包头内携带以带符号数表达的在各个维内传送的距离 $[\Delta x, \Delta y]$。在交换机 ij 中的路由操作由下列表示给出：

方向	条件
西（$-x$）	$\Delta x < 0$
东（$+x$）	$\Delta x > 0$
南（$-y$）	$\Delta x = 0$，$\Delta y < 0$
北（$+y$）	$\Delta x = 0$，$\Delta y > 0$
本处理器	$\Delta x = 0$，$\Delta y = 0$

为了完成这类路由，交换机需要测试数据包头中的地址并对一个路由域减 1 或加 1。网格中决定路由的典型做法是先沿 Δx 方向运动，然后再沿 Δy 方向运动。更为一般的情况是，如在 k 元 d-立方体中，通过从最低编号到最高编号的各个维依次运动决定路由，这叫*维序路由*。对于一个二元立方体来说，交换机计算出目的地和本地节点地址第一个有区别的位的位置（如果数据包携带目的地相对地址，则是第一个非零位），然后穿过对应维的链路，这叫 *e-立方体路由*。在并行机中，Intel 和 nCUBE 的超立方体、Paragon 机、加州理工大学的花环路由芯片（Seitz and Su 1993）和 J-machine 都使用这种机制。 789

一种更为通用的方法是*基于源的路由*，在这种方法中，源节点建立起一个数据包的头，其中包含了沿路由各个交换机输出端口号 P_0，P_1，…，P_{h-1}。各个交换机只需简单地从消息头中分离端口号，然后把消息从所指定的通道送出。这种方法允许使用非常简单的交换机，只有很少的控制状态，甚至不需要支持任意拓扑结构复杂的路由功能的运算部件。所有的智能都存在于主机节点之中。它的缺点是数据包头可能会较大，而且尺寸不固定。如果允

许交换机的度为 d，路由长度为 h，就需要数据包头携带 $h\log d$ 个路由位。MIT 的 Parc 和 Arctic 路由器、Meiko 的 CS-2 和 Myrinet 使用这个方法。

第三种方法是一种通用的方法，允许使用小且长度固定的数据包头。它采用表驱动路由，每个交换机包含一个路由表 R，而数据包头包括一个路由域 i，用路由域作为索引查表来决定输出端口，即 $o = R[i]$。例如，HPPI 和 ATM 交换机中使用该方法。一般来说，表项也给出了沿路径下一跳的路由域——o，$i' = R[i]$，这允许构造表的更大的灵活性。该方法的缺点是交换机必须包含相当大数量的路由状态，而且需要用额外的专门用于交换机的消息或其他的机制来建立路由表的内容。模拟简单的路由算法就需要相当大的路由表。这种方法更适合于 LAN 和 WAN，因为同时只使用节点集合间少量的可能路由，大多数路由是长时间持续的连接。与此相对比，在并行计算机中，通信通常涉及所有的节点。

传统的网络路由器包括一个全功能的处理器，它能检查进入的消息，执行任意的计算来选择输出端口，并且为该输出端口建立包含了消息数据的数据包。这一类的方案通常在连接完全不同的网络的路由器（例如在以太网、FDDI、ATM 网之间提供路由的路由器）或至少是在连接不同的数据链路层之间的桥接器中使用；从通信的时间尺度上看，在高性能并行机中采用这类方案是没有什么意义的。

10.6.2 确定性路由

如果消息所取的路由完全是由它的源和目的地决定的而与网络中其他的流量无关，这种路由算法叫做确定（非自适应）性算法。例如，维序路由是确定性的，数据包不管其路径的链路是否阻塞都要沿该路径走下去。维序和 e-立方体路由是确定性路由算法的例子。自适应路由算法允许路径上的其他流量影响数据包的路由。例如，在网格中，如果沿维序路径的链路阻塞或出错，数据包会沿锯齿形的路由流向其目的地。在胖树中，朝向公共祖先的向上路径可以避开阻塞的链路，而不是一定要遵循那条在将消息送入网络时决定的特定路径。如果路由算法仅选择通向目的地的最短路径，它就是最小化的（minimal），否则它就是非最小化的（nonminimal）。显然，自适应（以及容错）要求源和目的地之间有多条路由，这同时也提供了在多条链路上分散负载的机会。这些优点也可以为基于源和表驱动的路由所利用，但路径的选择必须在数据包注入网络时做出。自适应路由将路径的选择推迟，直到数据包真正在网络中流动时才选择，这显然会使交换机更为复杂，但它可能获得更好的链路利用率。

790

在研究自适应路由之前，我们将首先集中精力研究确定性算法，理解几种最流行的提供路由的算法以及证明它们免死锁的技术。

10.6.3 免死锁

在我们关于网络时延和带宽的讨论中，我们已经隐含地假设消息向前运动，这样谈论性能才是有意义的。本节将说明如何证明一个网络是免死锁的。让我们回忆一下，死锁是在一个数据包等待一个不能发生的事件时发生；例如，当消息系统的队列都是满的而且都在等待其他系统让出可用的资源时，任何消息都不能向其目的地前进。这种情况应与无限推迟（indefinite postponement）相区别，后者是在数据包等待一个能发生但从不发生的事件时产生的，也应与活锁（livelock）相区别，活锁是在数据包的路由不再指向其目的地时发生的。无限推迟主要与公平性问题有关，而活锁只会在自适应非最小化路由的情况才会发生。避免死

锁是良好设计的网络的基本性质，它必须从设计的最初就予以注意。

死锁能发生于各种不同的场合。当两个节点试图向对方发送，而且每个节点在对方能接收之前就开始了发送的话，就发生了“碰头”死锁。显然，如果它们两者都试图在转入接收之前结束发送，没有人能向前进。在使用同步发送接收的用户消息处理层以及在节点到网络的接口层，我们都能看到这种情况。在网络内部，在半双工通道的场合，或者当交换机控制器没有能力在双工的通道上同时发送和接收时，也会发生这种死锁。必须把通道考虑为逐步获得的共享的资源，首先在发送端，然后在接收端。在任一种情况下，都应该保证当节点不再能发送时仍能继续接收。一个可靠的网络只有当其节点即使无法送出数据包仍能从网络移去数据包，才是免死锁的。（另一方面，死锁的解除也可以通过最终检测到超时，从而去除一个或多个数据包，通过有效地抢占共享资源的方法来实现。但这的确会带来无限推迟的可能性，我们将在以后讨论。）这种碰头的场合并不涉及路由，问题是由交换机设计导致的限制所产生的。 791

一种更有趣的死锁情况发生于多个消息竞争网络内的资源的场合，如图 10-19 所示的路由死锁。这里，有几个消息正在通过网络，每个消息包含几个流控单元（flit）。我们应该认为网络中每个通道都与一定量的缓冲相结合，它们可以是在通道目的地的输入缓冲、或在通道源的输出缓冲，或者两处皆有。在我们的例子中，每个消息都试图左转，而与 4 条通道相联系的所有数据包缓冲都已经满了。在获得一个能够进入的新的数据包缓冲之前没有一个消息会释放任何数据包缓冲。我们可以将交换机分割成额外的交换机和通道来细化这个例子，但显然网络中的通道资源是逐步分配的，这是因消息流过而分布式地进行的，资源是非抢占式的，至少没有数据包丢失；因此，总是可能会产生死锁。

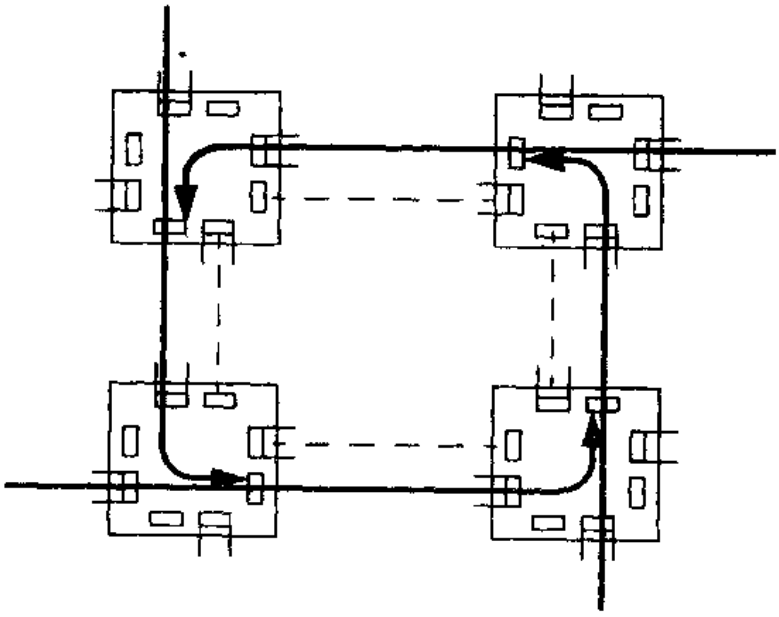


图 10-19 网络路由死锁的例子。4 个交换机每个都有 4 个输入端口和 4 个输出端口。4 个数据包每一个都获得了一个输入端口、一个输出缓冲和一个输出端口[⊖]，当这些数据包左转弯时都试图获得下一个输出缓冲。数据包在它们能向前移动之前都不会释放它们的缓冲，因此谁也无法前进

路由死锁可能发生于存储-转发或直通路由的情况，虽然对于直通来说，死锁的机会更大，因为每个数据包都会跨几个流控单元缓冲。只有数据包的头流控单元携带路由信息，因

⊖ 原书为 input port，有误。——译者注

此一旦消息的头单元进入通道，该消息所有剩余的流控单元必须走同一通道。所以，单个数据包就可能占据跨几个交换机的通道资源。这些例子的基本点是资源逻辑上与通道相关，消息在通过网络时引入了通道资源间的依赖关系。

[792]

证明一个网络是无阻塞的基本技术是要搞清楚当消息通过网络时通道之间的依赖关系，并且说明所产生的通道依赖图中没有闭合回路；这意味着通信模式不会产生死锁。完成该任务的最常用的办法是对通道资源编号，使每一条合法的路径都遵循单调升（或减）的顺序，因此，这不会产生依赖关系的闭合回路。对于蝶网，这很简单，因为网络本身是无回路的。对于树和胖树，只要向上和向下的通道是独立无关的，做到这一点也是很简单的。对于通道图中具有回路的网络来说，情况就比较微妙。

为了说明显示路由算法免死锁的基本技术，让我们说明在 k-元二维阵列上的 Δx，Δy 路由是免死锁的。为证明这一点，我们可以把每个双向通道都看作是一对独立编号的单向通道。将每个 x 正向通道 $<i, y> \rightarrow <i+1, y>$ 编号为 i，类似地，将 x 反向通道自 x 方向编号最大的正边沿从 0 开始简单地编号。将 y 正向通道 $<x, j> \rightarrow <x, j+1>$ 编号为 $N+j$，而将 y 反向通道自 y 的编号最大的正边沿从 N 起[○]。简单地编号。图 10-20 说明了这种编号方案。任何包含了一系列 x 方向的连续边，转弯 90 度，又包含了 y 方向一系列连续边的路径是严格递增的。通道依赖图中对网络中每条单向链路有一个节点，如果一个数据包能够通过通道 A 然后通道 B 的话，从节点 A 到节点 B 存在一条边。通道依赖图中所有的边都从低编号节点指向较高编号的节点，这样在通道依赖图中就没有闭合回路（尽管在网络中有很多回路）。

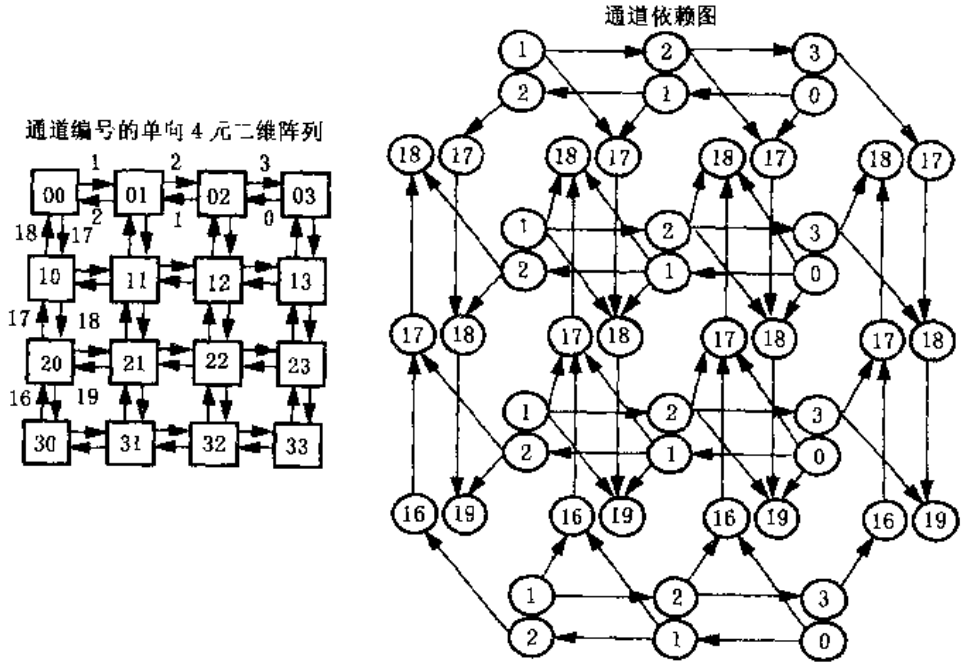


图 10-20　网络图中的通道编号和对应的通道依赖图。为了说明路由算法是免死锁的，只要证明通道依赖图中不存在回路

这个证明可以很容易地推广到任意维的情况，因为二进制的 d-立方体的每个维包含一对单向通道，可以说明 e-立方体路由对超立方体来说是免死锁的。但是，在一般情况下，该证明对 k 元 d-立方体并不适用，因为通道编号在头尾卷回边处会递减。确实，不难说明

○ 原文无写从 N 起。——译者注

对于 $k>4$，维序路由在单向花环（$d=1$）上会产生依赖回路。

注意，即使每条通道只有单个流控单元缓冲，免死锁路由的证明也能适用。还要注意对于采用存储转发路由的 k 元 d-立方体来说，即使有多个数据包缓冲，仍然存在潜在的死锁，因为一个消息可能占满沿其路径的所有数据包缓冲。但是，如果对通道的使用加以限制的话，就可能打破死锁。例如，考虑每通道有多个数据包缓冲、采用存储转发路由的单向花环的情况。假定与每个通道相联系的一个数据包缓冲给那些其目的节点的编号大于其源的编号的消息而保留，也就是说，为那些不使用卷回通道的数据包而保留，这意味着那些向正方向运动的消息总是可能前进。虽然卷回的消息会被推迟，网络不会死锁。这是使存储转发包交换网络免死锁的一组技术中的典型解决办法。有一个概念叫做*结构化的缓冲池*（在其中，某些缓冲有特定功能），路由算法限制缓冲对于数据包的分配来打断死锁回路。这个解决方案对于虫孔路由来说是不充分的，因为它默认不同消息中的数据包在向前运动时能被重叠。 793

请注意，免死锁路由并不意味着系统是免死锁的。只要流量能流入网络接口卡，即使网络接口卡无法发送，网络也是免死锁的。如果使用两阶段协议，我们需要保证避免取操作的死锁。这意味着或者提供两个逻辑上独立的网络，或者保证像第 7 章中所讨论的那样，两个阶段不通过网络接口卡发生耦合。当然，程序仍然可能死锁，如对锁的循环等待或者使用同步消息传递导致的碰头冲突。我们已经从顶层向下研究，说明了如何使得这些层次在下一个较低层是免死锁的时候免死锁。

给定一个网络拓扑和每个通道的一组资源，有两种办法构造免死锁的路由算法：限制数据包能走的路径或者限制资源的分配。这一观察产生了许多有意思的问题。对于采用虫孔路由的任意拓扑结构是否存在一种生成免死锁路径的通用的技术？这种路由可以是自适应的吗？是否至少需要一定量的通道资源？ 794

10.6.4 虚通道

使采用虫孔路由的网络免死锁的基本技术是对每个物理通道提供多个缓冲并将这些缓冲分割成一组虚通道。回到我们的网络基本成本模型，这样做并不增加网络中链路的数量，也不增加交换机的数量。事实上，它甚至并不增加每个交换机内部的交叉开关阵列的尺寸，因为同时只会有一个流控单元通过交换机到达各输出通道。如图 10-21 所示，它的确需要在交

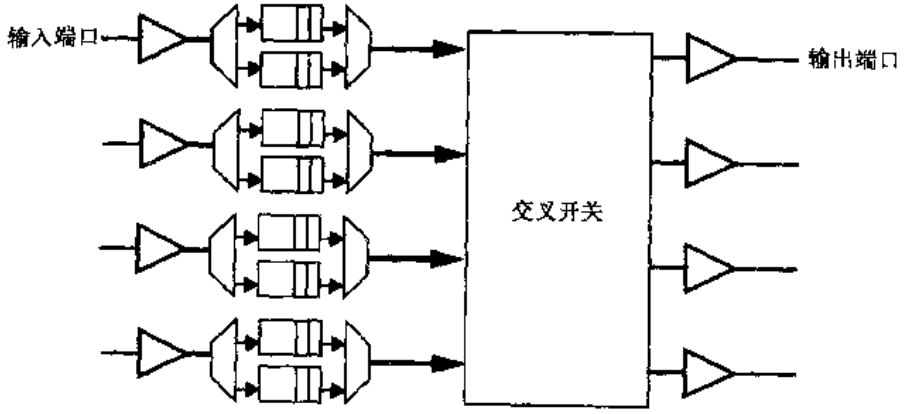


图 10-21 基本交换机中的多虚通道。每个物理的通道由多个虚通道共享。交换机的输入端口将进入的虚通道分离成独立的缓冲。但是，这些缓冲以多路复用方式通过交换机以避免扩张交叉开关阵列

换机内部增加选择器和多路器以便允许每个物理通道的多个虚通道之间共享链路和交叉开关。

我们可以通过使用虚通道，系统地切断通道依赖图中的回路来避免死锁。例如，考虑图10-19中的4路路由死锁回路，假定每个物理通道有两个虚通道，如果节点编号比消息的目的节点编号大，消息被送到高通道，如果节点编号比消息的目的节点编号小，消息被送到低通道。如图10-22所说明的那样，依赖回路被打断。将此方法用于 k 元 d-立方体，可以认为通道的编号是以 $d+1+k$ 为基的数，形式为 ivx，这里 i 是维度，x 是通道在 i 维中的源节点的坐标，而 v 是虚通道号。在每个维度上，如果目的节点坐标值比源节点的坐标值小（即如果消息必须使用卷回边的话），就在那个维中使用 $v=1$ 虚通道。否则，使用 $v=0$ 通道。你可以验证使用这种虚通道的分配方法，维序路由是免死锁的。对其他流行的拓扑可以使用类似的技术（Dally and Seitz 1987）。注意，有了虚通道，我们应该把路由算法看作函数 R：$C \times N \rightarrow C$，因为输出选择的虚通道也依赖于输入的通道。

795

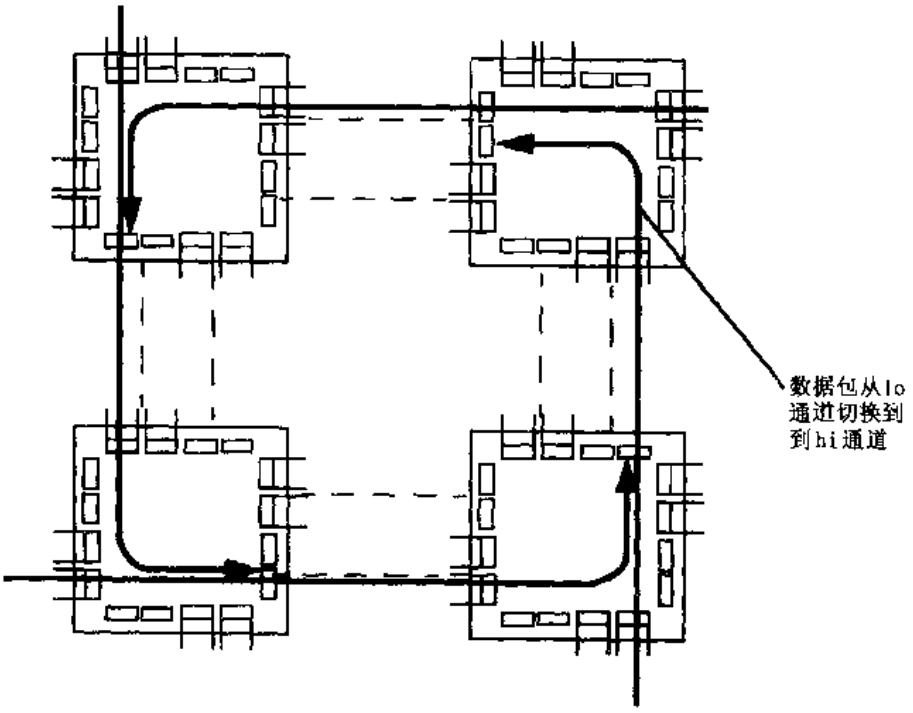


图10-22　用虚通道断开死锁回路。每个物理通道被分成两个虚通道，称它们为lo和hi。一般都使用与输入端口对应的虚通道作为输出，但向北向西的折转是个例外，它导致从lo到hi的虚通道切换

10.6.5　上行* - 下行* 路由

在任意拓扑结构上的免死锁虫孔路由是否需要虚通道？答案是否定的。如果我们假设所有的通道都是双向的，则存在一个为任意拓扑推导出免死锁路由的简单算法。不足为怪，该算法限制合法路由的集合。这种通用的策略与树中的路由类似，路径离开源沿树上行，然后下行到达目的地。我们假定网络由一组交换机组成，其中某些交换机上面连接有一台或多台主机。给定网络图，我们希望对交换机编号，使得当我们远离主机时编号增大。一种办法是构造一个图的生成树，以主机为叶子，编号朝向根而增大。显然，对任意的源主机，可以通过一条上行* - 下行* 的路径到达任意的目的地，该路径包含零个或多个上行的通道序列（朝向较高编号的节点）、一个折返和零个或多个下行的通道。此外，遵循这样的路径的路由

796

是免死锁的。网络图可能有回路，但在上行* - 下行*路由规则下的通道依赖图则没有回路。上行通道构成一个有向无回路图（DAG），下行通道构成另一个 DAG。上行通道只与较低编号的上行通道有关，而下行通道也仅依赖于上行通道和较高编号的下行通道。

这种风格的路由曾为 Autonet 开发（Anderson et al. 1992），其目的主要是自配置。每个交换机都包含一个处理器，该处理器可以运行分布算法来决定网络的拓扑，发现一棵惟一的生成树。每个主机按受限最短路径问题计算上行* - 下行*路径。然后建立交换机中的路由表。Atomic（Felderman et al. 1994）和 Myrinet（Boden et al. 1995）使用一种与广度优先搜索类似的算法，其交换机是被动的，由主机通过探察网络决定拓扑。每个主机运行一个算法，该算法将网络分层，主机节点在第 0 层，在各层的交换机对应于它距离一台主机的最大距离。从最高编号的交换机开始的广度优先的搜索给出这种编号，该算法决定了从主机到其他节点的基于源的路由的集合。网络自动映射的一个重要挑战性问题是，决定什么时候两条通过网络的不同路由通向同一个交换机（Mainwaring et al. 1997），特别是在使用只传送消息而不做任何特殊处理的简单交换机时更是如此。一个解决方法是通过从以前已知的交换机反向路由，试图回到源；另一种办法是检测从两个假设不同的交换机到同一主机是否存在相同的路径。

10.6.6 折转模型路由

我们已经看到可以通过限制网络内路由集合或对每个通道提供结构化模式的缓冲来构造免死锁的路由算法。我们应该在什么程度上限制路由呢？是否存在最小的限制集合或路由限制与缓冲的最小组合呢？在这个方向上最重要的发展是折转模型路由（Glass and Ni 1992）。例如，考虑一个 2D 的阵列。如图 10-23 所示，有 8 种可能的折转，构成两个简单的回路。（该图说明的是在涉及多个消息的网络中出现的回路，在通道依赖图中有一个对应的回路。）维序路由不允许使用 8 种折转中的 4 个，当沿 $-x$ 或 $+x$ 前进时，做 $-y$ 或 $+y$ 的折转是合法的，但是一旦数据包沿 $-y$ 或 $+y$ 前进，它就不能做进一步的转弯了。不合法的折转由图中的灰线表示。直觉地，只要在每个回路中去掉一个折转就可以防止回路出现。

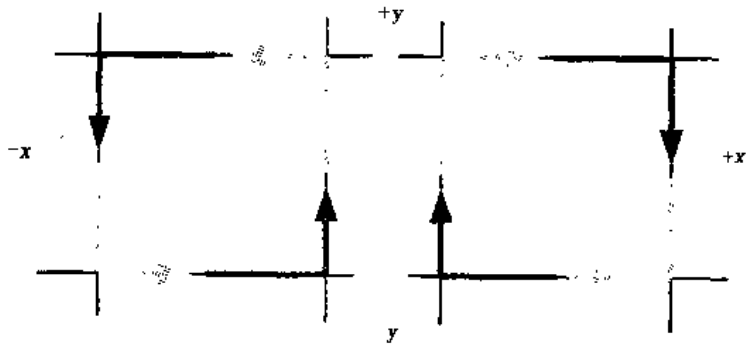


图 10-23 Δx，Δy 路由的折转限制。2D 阵列上的维序路由禁止使用 8 种可能的折转中的 4 种，从而断开这两种简单依赖回路。通过仅禁止两种折转就能获得免死锁的路由算法

在 2D 阵列中禁止两次折转的 16 种方法中，有 12 种避免了死锁。这些方法包含了图 10-24中的 3 种不同的算法及它们的旋转。首先向西算法之所以这样命名是因为不允许向 $-x$ 方向转弯；所以，如果数据包需要向这个方向前进的话，它必须在做出任何折转之前就这样做。同样，在最后向北的算法中，不能从 $+y$ 方向再转弯了，所以路由在朝向该方向之前必 797

须完成它所有其他的调整。最后，首先负向算法禁止从正方向转向负方向，这样路由在转向任何正方向之前必须走到它所要求的最负点。

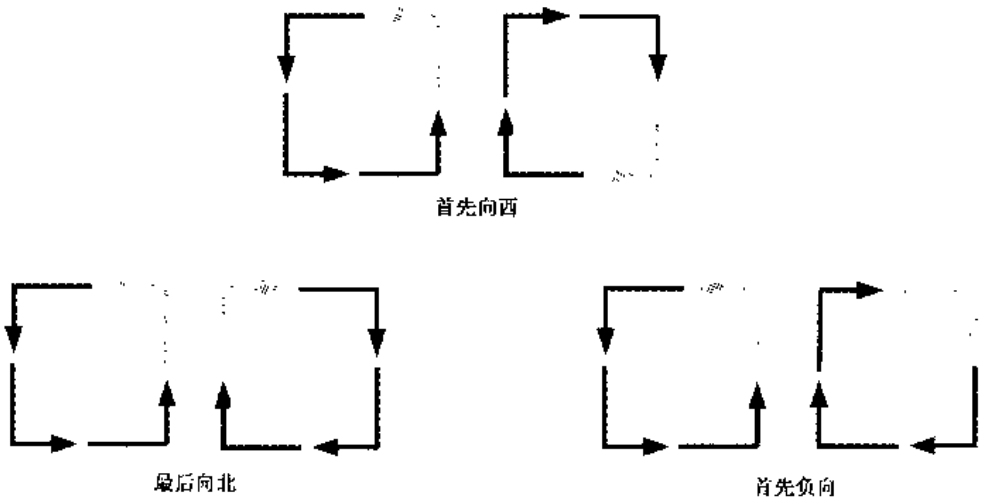


图 10-24　在 2D 中的最小折转模型路由。为了获得一个免死锁算法，仅需要禁止 8 种可能的折转中的两个。这里对 3 个这样的算法显示了合法的折转

798

这些折转模型算法的每一个都允许复杂的、甚至非最小化的路径。例如，图 10-25 显示了可以在首先向西路由下采用的某些路径。拉长的长方形表示阻塞或断开的链路，它们使这样一组路由能够使用。显然，最小折转模型在路由选择上有很大的灵活性。在一对节点之间有很多合法的路径。

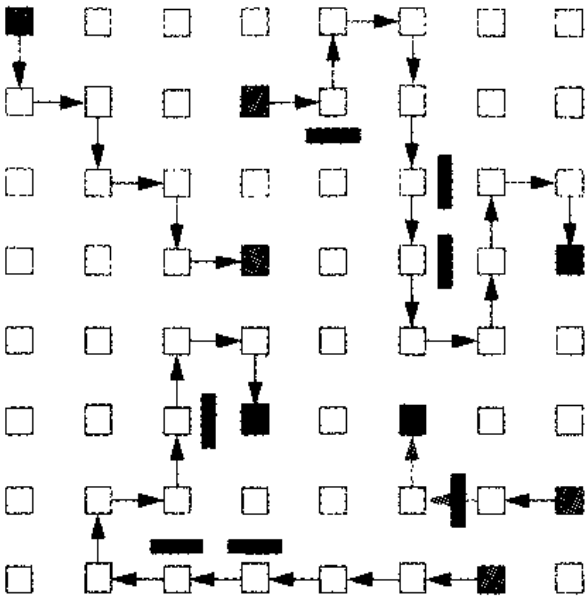


图 10-25　在一个 8×8 的阵列中合法的首先向西路由的例子。折转模型路由获得相当大的路由自适应性，因此提供了以免死锁方式的绕开故障的路由能力

折转模型方法可以和虚通道合并使用，而且可以应用于任何拓扑结构。（在某些网络中，例如单向的 d-立方体，仍然需要虚通道。）基本的方法如下：1）根据通道引导数据包的方向将它们分成组（不包括卷回的边）；2）识别由方向“折转”形成的潜在的回路；3）在每

个抽象的回路中禁止一个折转，小心地断开所有的复杂的回路。最后，在不引入回路的前提下可以结合使用卷回边。如果存在虚通道，则把每一组虚通道作为一个不同的虚方向处理。

上行* - 下行*本质上是一种假定双向通道的折转模型算法，它仅使用两个方向。的确，回顾上行* - 下行*算法，许多最短路径可能遵从上行* - 下行*的限制，而许多非最小化的上行* - 下行*路由肯定存在于大多数网络之中。虚通道允许进一步放松路由的限制。

10.6.7 自适应路由

放松路由限制的最根本的优点在于它允许节点对之间有多条合法路径。这对容错而言是基本的条件。假如路由算法仅允许一条路径，单条链路失效将使网络断开。对于多路径路由，则有可能绕开故障。此外，它还允许在可用通道上更广泛地分布流量，从而改善网络的利用率。当一辆车停在街道中央时，别的车能选择绕开该街区当然是最好的了。 799

简单的确定性路由算法可能在网络中引起严重的竞争，即使当通信负载在独立的目的地之间均匀分布时也不能避免。例如，图 10-26 显示了一个简单的情况，2D 网格中 4 个数据包正从不同的源向不同的目的地前进，在维序路由下，它们都被强迫沿同一链路前进。通信在通过瓶颈时完全是串行的，而其他最短路径上的链路却闲置未用。多路径路由算法可以使用其他的通道，正如图 10-26 的右部所显示的那样。对于任何网络拓扑，总存在坏的置换（Gottlieb and Kruskal 1984），但简单确定性路由使得路径更容易碰到这些坏的置换。图 10-26 中的特殊例子有着重要的实际意义。一种公共全局通信模式是转置。在一个维序路由的 2D 网格上，一行中所有的数据包在填入列之前必须通过单个交换机。

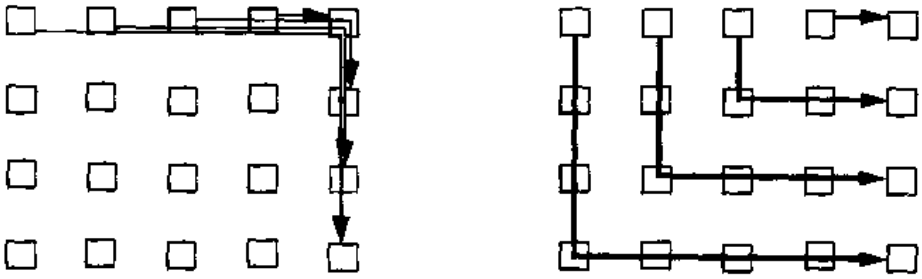


图 10-26 确定性维序路由下的路由路径冲突。从不同源到不同目的地的几个消息在维序路由下竞争资源，而自适应路由方案能使用不相交的路径

多路径路由可以作为任何基本交换机制的延伸而结合使用。对于基于源的路由，源简单地在多条合法路由中挑选，并根据选择建立数据包的头。对交换机不需要改变。对于表驱动路由，可以通过为多条路径建立表项来完成。对于运算路由，需要给数据包头附加额外的控制信息，并由交换机解释这些信息。

自适应路由是多路径路由的一种形式，这里路由的选择是由交换机根据路由中碰到的流量动态地决定的。形式化地表示，自适应路由函数是形如 R_A：$C \times N \times \Sigma \rightarrow C$ 的一个映射，这里 Σ 表示交换机状态。特别是如果所希望的输出端口之一被阻塞或失效，交换机可以选择一个替代通道送出数据包。最小自适应路由仅仅沿到达目的地的最短路径引导数据包，也就是说，每一跳都必须缩短到达目的地的距离。允许使用所有最短路径的自适应算法是完全自适应的，否则就是部分自适应的。非最小化自适应路由的一个有趣的极端情况是所谓“热土豆”路由。在这个方法中，交换机从来不缓冲数据包。如果一个以上的数据包指向同一个

输出通道，交换机将一个送往其目的地，而将其他的“误导”到其他通道。

虽然自适应路由在研究文献中（Ngai and Seitz 1989；Linder and Harden 1991），特别是通过 Chaos 路由器（Kostantantindou and Snyder 1991），已经得到深入的研究，但它在现代的并行机中并未得到广泛的应用。CRAY T3E 提供了立方体结构中的最小自适应路由。nCUBE/3 机提供了超立方体结构的最小自适应路由。Tera 机（Alverson et al. 1990）建议的网络使用热土豆路由，在 3ns 的周期内传递 128 位的数据包。

800

虽然自适应路由有明显的优点，但也存在其缺点。显然，它增加了交换机的复杂性，这只会使交换机变得更慢。带宽的降低会抵消更复杂的路由所带来的增益，一个处于其线性工作区的简单确定性网络的性能可能胜过一个处于饱和状态的智能的自适应网络。在像 d-维阵列这样的非均匀网络中，在均匀随机流量负载下自适应性会损害性能。负载的随机变化对任何网络都引入暂时的阻塞。对于位于阵列边界的交换机，这将导致将数据包推向中心。结果，在阵列中心形成确定性路由所没有的竞争。自适应路由也会对某些类型的非均匀流量产生问题，我们将在 10.8.3 节考察这些问题。

当网络进入饱和时，非最小化自适应路由的性能可能很差，因为数据包流过额外的链路从而消耗更大的带宽。如图 10-17 所说明的那样，当负荷上升时，网络的吞吐量会下降，而不是在饱和点保持平坦。

近年来，人们提出了不少低成本的部分和完全自适应路由的建议，它们采用有限数量的虚通道和限制折转集合相结合的办法（Chien and Kim 1992；Schwiebert and Jayasimha 1995）。这些方案显示，以非常有限的自适应度就能获得自适应路由的大多数优点，包括容错和通道利用率。

10.7 交换机的设计

最终，网络的设计就归结为交换机的设计以及如何用线路将这些交换机连接在一起。交换机的度、交换机的内部路由机制和它的内部缓冲决定了它能够支持什么样的拓扑结构和能实现什么样的路由算法。因为我们已经理解了较高层网络设计的问题，我们可以回到交换机设计的细节上来。像一个计算机系统中其他硬件成分一样，一个网络交换机包括数据通路、控制和存储。在本章开始的图 10-5 中已经描述了这种基本结构，在并行计算早期，交换机是由许多低集成度的元件组成的，占据一个主板或者一个机架。从 20 世纪 80 年代中期开始，大多数并行计算机网络是围绕单芯片 VLSI 交换机设计的，即采用了与微处理器完全相同的技术（十年后，这一技术开始使用在 LAN 中），因此，交换机的设计与第 1 章讨论的技术趋势密不可分：减少特征尺寸，增大面积，增加引脚数目。我们应该从 VLSI 的角度看待现代交换机的设计。

801

10.7.1 端口

引脚的总数基本上等于输入端口和输出端口之和与通道带宽之积，因为芯片的周长比其面积而言增长得要慢，所以交换机肯定要受到引脚数目的限制，这就促使设计者去考虑窄而高频的通道。非常高速的串行链路是特别有吸引力的，因为它们使用最少的引脚并且消除了通道上位线之间的扭斜问题。但是在串行链路中，时钟和所有的控制信号都必须在串行位流的帧内编码，在并行链路中，线路之一是作为其他线路上数据的基准时钟。使用一条单独的

线路，提供准备好/确认握手信号来实现流控。

10.7.2 内部数据通路

数据通路为每个输入端口（如输入锁存器、缓冲区或 FIFO）与每个输出端口之间提供连接。虽然它可以有多种不同的实现方法，但通常被称作内部交叉开关。一个非阻塞的交叉开关是指在任何一种置换下，它可以同时将每个输入端口连到一个不同的输出上。逻辑上讲，对于一个 $n \times n$ 的交换机，非阻塞交叉开关只不过是一个 n 路的连接各个目的端口的多路复用器。如图 10-27a 所示，多路复用器根据其下层的技术可以有很多不同的实现方法，比如，在 VLSI 中，它通常是靠在一根总线加上 n 个三态驱动器来实现的，如图 10-27b 所示；在这种情况下，控制通路为每个输出提供 n 个使能点。一种越来越常用的技术是使用内存作为交叉开关，通过对每个输入端口写入和对每个输出端口读出来实现，如图 10-27c 所示。

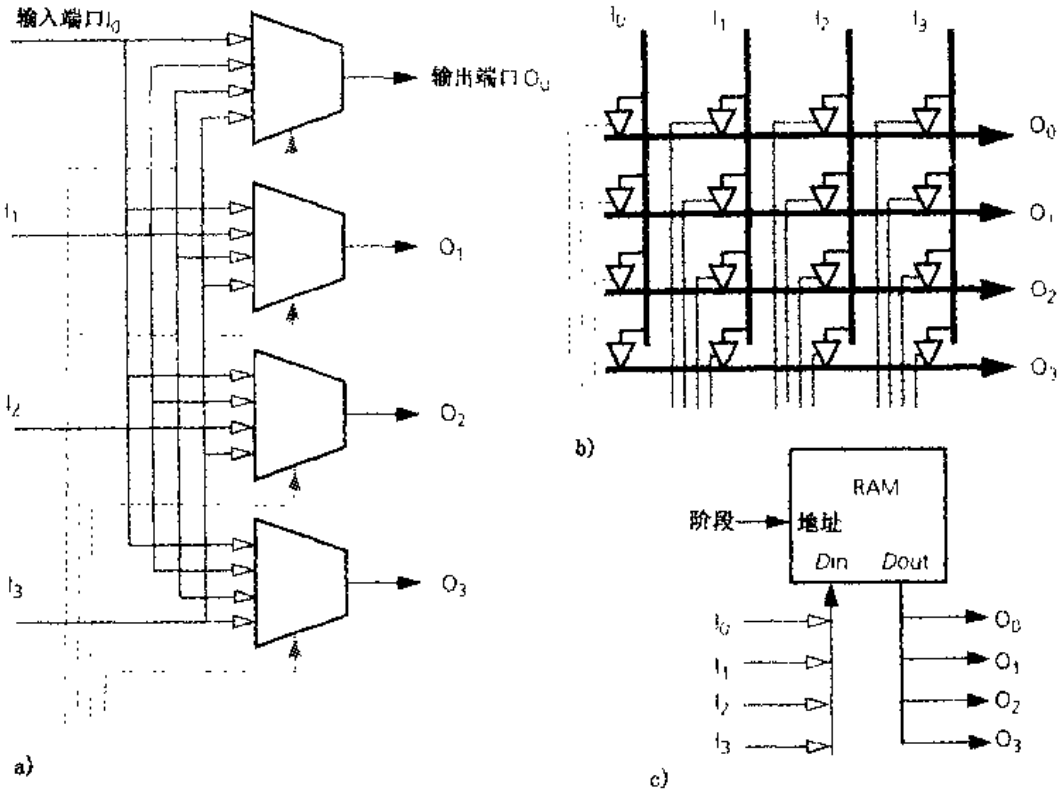


图 10-27 交叉开关实现。一个交换机的内部交叉开关的实现可以是：a）多路复用器的集合，b）三态驱动器的网格，c）一个在端口间时分复用的争用型的静态 RAM

很显然，交叉开关的硬件复杂度是由连线决定的，在每个方向上有 nw 条数据线，需要 $(nw)^2$ 的面积，还需要 n^2 条控制线，这就显著地增加了复杂度。我们如何做到交换机跟随 VLSI 技术而改进呢？假设交叉开关的面积是常数，而特征尺寸减小，理想化的 VLSI 缩放定律指出，如果特征尺寸减小为 $1/s$（包括门厚度），电压也减小到 $1/s$，则晶体管的速度则会改善 s 倍，连接相邻晶体管的线路延迟将改善 s 倍，每个单元面积的晶体管总数在相同的能量密度时增加 s^2 倍。对于交换机而言，这就意味着连线会更细、更密，所以交换机的度可以提高 s 倍。注意，交换机度的提高只是逻辑密度提高的平方根，不好的一点是这些连线经

802

过交叉开关的整个长度，所以连线的长度是一个常数；线越细，它们的电阻就越大；电容减小，净效应是传输延迟不变（Bakoglu 1990）。换句话说，理想的缩放规律使我们在相同面积上有了交换机度的改进，但是没有速度的提高，如果电压级别保持不变时，速度的确会提高 s 倍，但是这时功耗增加 s^2 倍。芯片面积的增长将允许更大的度数，但连线长度和传输延迟都会增长。

一些对于度的混淆理解源于术语“交叉开关”中，在传统的关于交换的文献中，有时将有单一的控制器的多级互连网络称作交叉开关，即使通过交换机的数据通路被组织成为小交叉开关的互连，在很多情况下，它们之间的连接很像 Banyan 网络那样的蝶形拓扑结构。如果对一个 Banyan 网络的输入是排序的，则该网络是非阻塞的，所以在一个 Banyan 网络之前，构造一些非阻塞的交叉开关构成的 Batcher 排序网（Peterson and Davie 1996）。有一种方法在很多方面与 Benes 网类似，它使用了蝶形网的变形，即 delta 网络，且串连使用了两个这样的网络。第一个是用来将数据包的位置与输入端口的对应关系随机化，第二个是生成到输出端口的路由。比如，它用于一些商用的 ATM 交换机（Turner 1988）。在 VLSI 交换机中，实际地建立一个非阻塞的交叉开关通常更有效，因为它简单、快速，而且有规则。关键问题还是引脚数目的限制。

803

显然 VLSI 交换机在现存的支撑技术下会继续进步，尽管它进步的速度很可能会低于存储和逻辑部件的改进速度。如果我们放弃非阻塞的特性，并限制能同时连接到输出上的输入端口的数目，交叉开关的硬件复杂度将会降低；在极限情况下，它将缩减为一条有 n 个驱动器和 n 个输出选择的总线。然而，在实践中最严重的问题却是连线的长度和引脚的数量，所以减少单个交换机内部带宽几乎没有带来任何节约，反而严重地损失了网络的性能。

10.7.3 通道缓冲

在交换机中缓冲存储器的组织在很大程度上影响着交换机的性能，传统的路由器和交换机在交换机构之外一般都有大的 SRAM 或者 DRAM 缓冲，而对于 VLSI 交换机，缓冲设在交换机内部，占用与数据通路与控制部分相同大小的硅片面积。有 4 种基本的选择：无缓冲（只有输入输出锁存）、输入有缓冲、输出有缓冲和一个集中共享的缓冲池。在输入和输出通道上提供几个流控单元大小的缓冲能使链路两端的交换机脱离关系，这显著地改进了性能。随着芯片尺寸和密度的增加，将会有更多的缓冲，网络的设计者也将有更多的选择；但是缓冲仍然是一种珍贵的资源，因此合理的组织结构是重要的。像网络设计中的很多其他的方面一样，问题不仅仅是如何有效地利用缓冲资源，而是缓冲将如何影响网络中的其他成分的利用。

直觉地，我们也许会估计到共享交换机的存储资源要比把存储器分配到各个端口更难实现，但它却能够更好地使用这些资源，所有的通信端口需要同时访问共享池，这就需要一个带宽非常高的存储器，更令人惊奇的是：按需求共享缓冲区在某些情况下会损害网络的利用率，因为单个拥挤的输出端口就会占用缓冲池的大部分，因而防碍了其他的网络流量通过交换机。

1. 输入缓冲

一个有吸引力的方法是在每个输入端口提供独立的 FIFO 缓冲区，如图 10-28 所示。在每个时钟周期内，每个缓冲区要能够接收一个物理单元且将一个物理单元传送到输出上，这

804

样交换机的内部带宽很容易与进入的数据流相匹配。交换机的操作是比较简单的，它监视每个输入 FIFO 的头，分别计算其希望的输出端口，相应地调度数据包通过交叉开关。典型的情况是，每个输入端口都附带路由逻辑来决定所希望的输出。对于基于源的路由来讲这很简单，它需要为每个输入增加一个计算路由的运算单元，通常，每个输入端口还要有一张支持表驱动路由的路由表。对于直通路由，决策逻辑不是在每个时钟周期而是对每个数据包做一次独立选择，因此，路由逻辑基本上是一个有限状态机，它在数据包边界处做出一个新的路由决定之前，把当前数据包的所有流控单元都送到同一个输出通道（Seitz and Su 1993）。

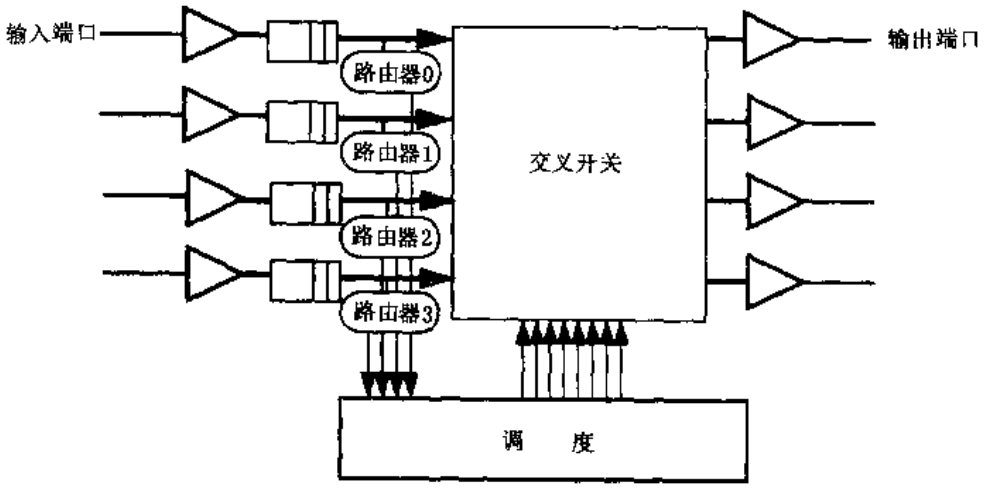


图 10-28 带输入缓冲的交换机。每个输入端口提供一个 FIFO，但是控制器只能在输入 FIFO 头上检查和处理数据包

这种简单输入缓冲方法存在的一个问题是发生“排头”阻塞。假设有两个端口的数据包的目的地是同一个输出端口，它们之中的一个会被调度到该输出端口而另一个则被阻塞。跟在这个被阻塞的数据包后面的数据包的目的地可能是一个并未使用的输出端口（一定存在未被使用的输出端口），但是它也不能向前传输。这种排头阻塞问题类似于我们日常的交通阻塞。就好像通往交叉路口只有一条车道，如果最前面的汽车在试图拐弯时被阻塞了，尽管前面的街道是空的，其他的车子也没有办法绕过它继续前进。

我们能容易地估计排头阻塞对于通道利用率产生的影响。如果我们有两个输入端口，为每一个输入随机地选择一个输出，第一个选择成功后，第二个有 50 比 50 的几率能选择到一个空闲的输出。所以，每个时钟周期通过交换机的数据包的期望数为 1.5 个，从而每个输出端口的利用率是 75%。推广到一般情况，如果 k 个随机输入到达一个 n 端口交换机所覆盖的输出端口的期望数表示为 $E(n, k)$，则 [805]

$$E(n,k+1)=E(n,k)+\frac{n-E(n,k)}{n}$$

对于不同尺寸的交换机计算这种迭代直到 $k=n$，揭示了对于一个满负载交换机的一个时钟周期，输出通道的使用率的期望值会迅速下降到 65% 左右。排队论理论分析表明在具有输入队列的稳定的状态下利用率的期望值为 59%（Karol，Hluchyj，and Morgan 1987）。

头阻塞的影响可能比这种简单概率分析的结果要更显著。在一个交换机中，总会有针对一个输出端口的突发流量后跟针对另一个输出端口的突发流量的现象出现。即使流量被平均

地分布，如果窗口足够大的话，每次突发效应都会导致所有输入上的阻塞（Li 1988）；即使在交换机中不存在输出的竞争，在输入缓冲区首部的数据包也可能会被分到一个由于网络其他地方拥塞而阻塞的输出上，这样，这个数据包之后的数据包也不能向前移动。在一个采用虫孔路由的网络中，整个蠕虫会被原地阻塞，消耗链路的带宽而到达不了任何地方。一个更加灵活的缓冲资源的组织结构或许能允许数据包滑动到被阻塞的数据包之前。

2. 输出缓冲

我们需要对交换机做的基本改进是提供一种方法，把各个输入端口的多个数据包作为发往输出端口的候选者。一种很自然的做法是扩展输入的 FIFO，从而为每个输出端口提供一个独立的缓冲区，这样数据包在到达时就可以自动地根据目的地址分类，如图 10-29 所示。（这是一种在本章 10.5 节中常规延迟分析所假设的那种交换机，因为交换机不会引入内部额外的竞争效应，所以这种分析被简化了。）当输入的业务流稳定，输出就会基本上 100% 被驱动；但是，这种设计的优点不付出代价是得不到的。需要额外的缓冲存储器和内部的互连[○]。加上分类的级和更宽的多路复用器，这可能加长交换机的周期时间或增加它的路由延迟。

在图 10-29 中的缓冲区是与输入还是输出端口相联系是一个观察角度的问题。如果把它看作是输出端口的缓冲区，则主要的特征是每个输出端口在一个时钟周期内有足够大的内部带宽来接收从每个输入端口来的数据包，这可以通过一个惟一的输出 FIFO 来实现，但是它工

806

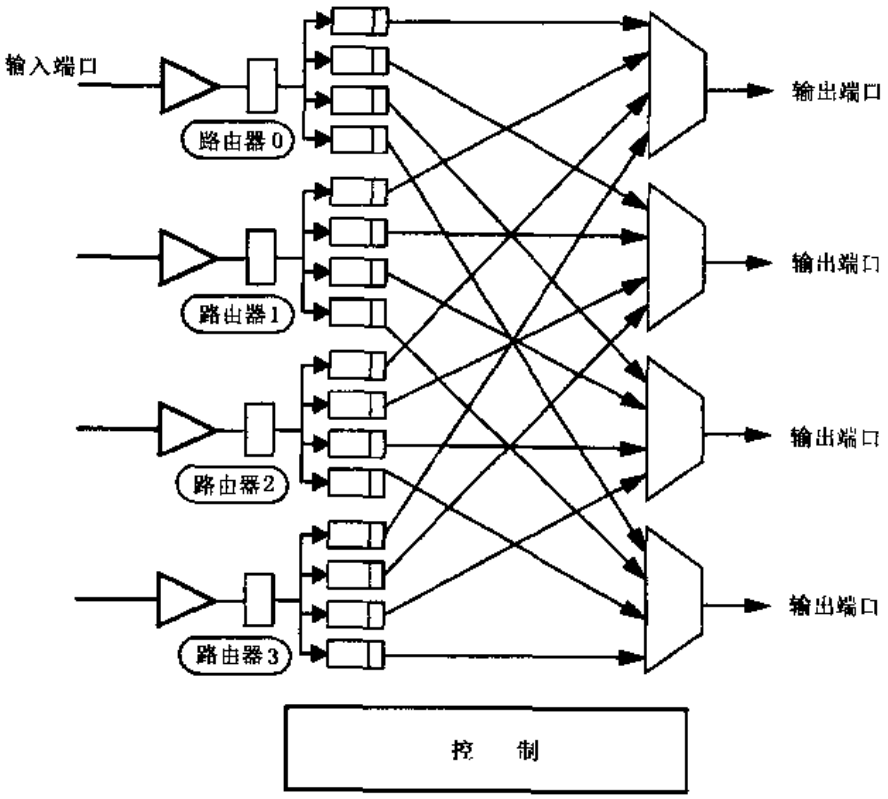


图 10-29 避免排头阻塞的交换机设计。在每个输入端口，数据包按输出端口分类，这样控制器可以在任何输入有数据包要达到某一输出端口时，向这个输出端口调度一个数据包

○ 提供带输出缓冲的的交换机也是可以的，从而避免存储和互连的代价（Joerg 1994），各个输入的缓冲区的集合形成一个缓冲池，每个输出有一系列指针，指向以该输出为目的地的数据包。该设计的定时要求是每个周期能将 n 个指针而不是 n 个数据包压入输出端口缓冲。

作的内部时钟频率应该是输入端口时钟频率的 n 倍。

3. 共享池

通过共享池，每个输入端口把数据存放在一个中央存储器中，每个输出缓冲区都可以从那里读取。因为输入端口可以不考虑输出端口而向共享池写数据，假如空间足够大，就可以避免排头阻塞。难点在于匹配 n 个输入端口和 n 个输出端口的带宽。一个通常的办法是使通向共享池的内部数据通路的宽度为链路宽度的 $2n$ 倍，每个输入端口在向共享池写数据之前缓冲 $2n$ 个物理单元，每个输出端口一次获得 $2n$ 个物理单元。经常使用高速缓存所采用的 SRAM 技术来构造这些共享池。

4. 虚通道缓冲

虚通道建议了另外一种组织交换机的内部缓冲区的方法。回忆一下，一个虚通道的集合提供了在一条单一物理链路上多个独立的数据包的传输。如图 10-21 所示，为了支持虚信道，链路上的数据流到达输入端口时被分配到不同的通道缓冲区中。然后，在交叉开关之前或之后，又被多路复用合成到一起，输出到输出端口。如果一个虚通道缓冲区阻塞了，很自然地会考虑将其他的虚通道送到输出。交换机有机会在每个输入的多个数据包中进行选择并送到输出端口；但是，这种情况下的选择是在不同的虚通道之间而不是在不同的输出端口之间。有可能会发生这种情况，即所有的虚通道被路由到同一个输出端口，但是输出端口复盖的期望值要好得多。在概率分析中，我们会问：在 n 个端口的 vn 个请求中选择时，所复盖的不同的输出端口的期望数是多少？这里 v 是虚通道的数目。

807

模拟研究表明，使用中等大小的缓冲（每个通道 16 个流控单元）和虫孔路由的大的二元蝶状网络（256 ~ 1 024 个节点），在随机流量下，通道使用率为 25% 时达到饱和。如果同样是每个通道 16 个流控单元的缓冲，但分布到更多的虚通道上，饱和带宽将显著增加。在两个虚通道（每通道 8 个流控单元缓冲区）时超过 40%，在 16 个带单个流控单元缓冲的通道时，接近 80%（Dally 1990a）。尽管这个研究保持每个通道的总缓冲区的大小固定，它却不能真正保持成本不变。这是因为在考虑把从任何一个通道来的数据包送到输出端口时，必须要为每个虚通道而不是为每个物理通道计算路由决策。

至此，你正在向前跳越了一步，发现其他一些可以考虑的代价-性能的折中。比如，如果交叉开关对每个虚通道都有一个输入，那么从一个端口来的多个数据包可以一次通过，这就增加了每个数据包前进的概率，从而提高了通道的使用率。交叉开关仅仅在一个维度上增加了尺寸，但取消了多路复用器，因为每个输出端口逻辑上是一个 vn 路的多路复用器。交换机的设计为创新留下了很大的空间。

10.7.4　输出调度

我们已经看到了为每个输入数据包决定所希望的输出端口的路由机制、提供从输入端口到输出端口的连接的数据通路、以及允许每个输入端口的多个数据包作为送往输出端口的候选者的缓冲策略。在交换机设计中遗漏的一个关键部分是*调度算法*，它在每个时钟周期里选择向前发送的数据包。选择确定后，交换机控制逻辑的其余部分激活在交叉开关或多路复用器和缓冲或锁存器中的控制点，产生从各个选定的输入到相关输出的寄存器传送。和交换机设计的其他方面一样，也有从简单到复杂的一系列的解决方案。

一个简单的方法是将调度问题看作是 n 个独立的仲裁问题，每个对应于一个输出端口。

如图 10-30 所示，每个候选的输入缓冲和每个输出端口之间，都有一条通往输出端口请求线和一条来自输出端口许可线。（该图显示了驱动 3 个输出端口的 4 个候选输入缓冲，表明路由逻辑和仲裁输入是基于每个输入缓冲而不是基于每个输入端口的。）路由逻辑计算所希望的输出端口且激活对所选择的输出端口的请求线。输出端口调度逻辑在请求之间仲裁，选中一个，并激活对应的许可信号。针对在图 10-27b 中采用三态驱动器的交叉开关的特定情况，输出端口 j 通过发出控制使能信号 e_{ji} 来使能输入缓冲 i。一条许可线的激活导致输入缓冲逻辑发送它的 FIFO。

808

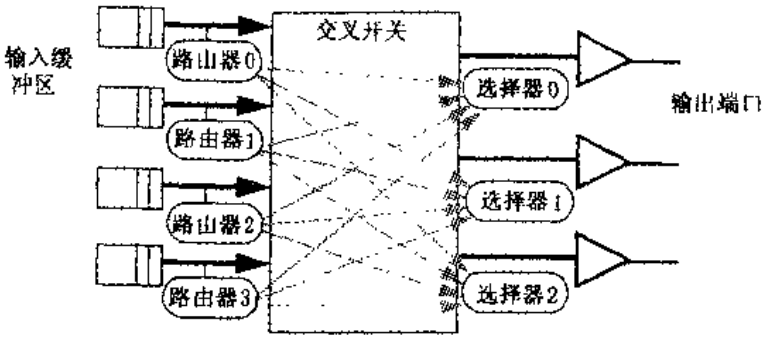


图 10-30　输出调度的控制结构。和每个输入缓冲区相联系的是决定输出端口的路由逻辑、通向各个输出的请求线、来自各个输出的许可线。每个输出有一个选择逻辑，它对发来的请求仲裁并且激活一个许可信号，从而引起一个流控单元从输入缓冲发到输出端口

设计中的另一个的问题是用于调度流控单元到输出的仲裁算法，可选的有静态优先级、随机、循环优先级、最老者优先调度，每种算法都有不同的性能和实现复杂度。显然，静态优先级是最简单的，它仅仅是一个优先级编码器。但是在一个大型网络中，它可能引起无限推迟。一般说来，对于输入提供公平服务的调度算法的性能要好些。循环优先级算法需要在每个时钟周期内增加一个额外的状态来改变优先级的顺序。最老者优先的平均时延与随机分配的一样，但是却显著地降低了时延的方差（Dally 1990a）。实现最老者优先调度的一种方法是在每个输出端口上使用一个大小等于输入端口数的控制 FIFO。当一个输入缓冲请求输出时，请求被排在队列中，在 FIFO 头部的最老请求才会被许可。

图 10-30 对于考虑实现不同的路由算法和拓扑结构是十分有用的。比如，在一个直接的 d-立方体中，有 $d+1$ 个输入（将它们编号为 i_0，…，i_d）和 $d+1$ 个输出（编号为 o_1，…，o_{d+1}），主机连接输入 i_0 和输出 o_{d+1}，直传路径 $i_j \rightarrow O_j$ 对应在相同维上的路由，其他路径则要改变维。对于维序路由，数据包只能在经过交换机时提高维序，因此并不是需要完全的请求/许可逻辑，输入 j 只需要向输出 j，…，$d+1$ 发出请求，输出 j 只需要许可输入 0，…，j。明确的静态优先级机制以升序或降序来分配优先权。

809

自适应路由在实现上有什么需要呢？第一，每个输入的路由逻辑必须根据算法的特定规则来计算多个候选输出，比如，折转限制和平面限制。对于部分自适应路由，可能只有几个候选者，每个输出接收到几个请求且只能许可一个（或者如果输出阻塞，它不会许可任何一个）。微妙的是一个输入可能被几个输出选中，它需要选择其中之一，但是没有被选择的输出会怎么样呢？它是应该反复仲裁选择另外一个输入端口，还是应该进入空闲状态呢？这个问题可能被形式化描述成一个在线双侧匹配问题（Karp，Vazirani，and Vazirani 1990）。请求定义了一个双侧图，输入为一侧，输出为另一侧，许可（每个输出一个）定义了一个在请求

图中输入/输出对的匹配，最大匹配是允许在一个时钟周期中最大数量的输入被向前发送，它应该给出最大的通道利用率。从这个观点看这个问题，交换机调度逻辑应该近似于一个快速并行匹配算法（Anderson et al. 1992）。基本的思想是用一个简单的贪婪算法形成一个试探性的匹配，比如在每个输出上随机选择请求，接着在每个输入上选择许可；然后，对于每一个没有选上的输出，尽量对试探性匹配作一些改进，实际上，在几次迭代之后就无法再做改进。这显然是复杂性和速度相对照的又一个例子，如果调度算法增加交换机的时钟周期或路由延迟，它可能在应付一些额外的阻塞时情况要好些，任务完成得要快些。

这种最大匹配问题适用于交叉开关的各个输入上多个虚通道多路复用通过的情况，甚至对确定性路由也是如此。（的确，该算法的提出是为了 AN2 ATM 交换机能应付对来自几条"虚电路"[○]的码元进行调度的情况。），每个输入端口有多个缓冲区，它可以调度到其交叉开关的输入，它们可能指向不同的输出。输出的选择决定了向前发送哪个虚拟通道。如果加宽交叉开关，而不是对输入多路复用，匹配问题将会消失，每个输出将能做出简单的独立仲裁。

10.7.5 堆叠式维度交换机

如果只有两个输入和两个输出，交换机设计的很多方面将可以简化，包括控制、仲裁和数据通路。包括花环路由芯片（Seitz and Su 1993）、J-machine 和 CRAY T3D 在内的几个设计使用了简单的 2×2 构造块，并且把它们堆叠起来构成更高维的交换机，如图 10-31 所示。如果我们考虑一个 d-立方体，在给定维上的连续流量直接通过那个维的交换机，如果它需要转到另一个维上时，它就会被垂直地引导通过交换机。注意，除了最低维之外，这对所有维增加了一跳。同样的技术在用于超立方体时产生了一种称作*立方体连接回路*的拓扑，超立方体的每一个 $n\times n$ 节点将被 n 个 2×2 节点的环所代替。 [810]

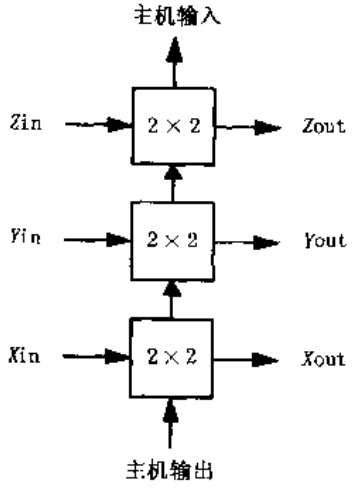


图 10-31 堆叠维度交换机。在一个维上的连续流量直接通过一个 2×2 交换机，而当它要转向另一个维时，它就被沿堆叠向上垂直引导

10.8 流控

本节，我们将仔细地考虑当网络中的多个数据流试图同时使用相同的共享网络资源时会发生的情况。必须采取一些措施来控制这些流。如果不想让任何数据丢失，必须在处理

○ 不应该把虚电路与虚通道相混淆。前者是在从源到目的地的整个路径上附带资源路由的技术，而后者是安排每条链路所带的缓冲的结构的策略。

其他流时阻塞一些流。流控的问题出现在所有的网络中并且在多个层次中，但是在并行计算机网络中的流控与局域网和广域网的流控本质上是完全不同的。在并行计算机中，网络流量需要像流经总线的流量那样被可靠地传递，而且在非常小的时间尺度上有大量并发的数据流发生，没有其他的网络体系有这样严格的要求。我们将简要地看一看这些不同点，然后详细地考察一下并行机中链路层和端对端的的流控问题。

10.8.1 并行计算机网络与局域网、广域网的对照

为了建立对于并行机网络上这种独特的流控的感性的认识，我们先稍微离题，看看在我们每天都为文件传输之类的工作而与之打交道的网络中流控扮演的角色。我们将考察3个例子：以太网风格的基于冲突检测的仲裁机制，FDDI风格的全局仲裁机制，广域网的无仲裁机制。

[811]

在以太网中，整个网络实际上是一根共享线路，就像总线那样，只是要长些。（总带宽等于链路带宽）但是，它和总线不同的地方是没有明显的仲裁单元。一台主机在尝试发送数据包时，首先检测到网络闲，然后（乐观地）把数据包发送到线路上。所有的节点包括尝试发送数据包的主机都在监视线路，如果线路上只有一个数据包，每台主机都会“看到”它，被标记成目的节点的主机会接收到这个数据包。如果线路上有冲突，包括多个发送者在内的每台主机都会检测到混淆的信号，一台主机驱动一个数据包的最短时间（即最小的通道时间）是50 μs，这个时间段允许所有的主机来检测到冲突。

流控处理的是如何重新发送数据包。发生冲突时，每个发送者回退，等待一段随机的时间，然后再尝试发送。随着每一次重复的冲突，随机时间延迟决定的重发时间间隔会逐渐增加。冲突检测在网络接口硬件中执行，而重发是由以太网驱动程序的最低层来处理的。如果在很多次尝试后仍然不成功的话，以太网驱动程序就会放弃，丢掉数据包。但是，消息操作是由高层的通信软件引起的，它们有自己的传输约定。比如，TCP/IP层通过超时会检测到发送失败并且使用它自己的自适应的重传机制，就像我们将要提到的使用广域连接那样的机制。UDP层会忽略掉传输的失败，把它留给用户应用程序来检测这个事件从而重发数据。以太网的基本概念是基于线路速度比主机通信能力快得多的假设（这种假设在20世纪70年代中期以太网被开发的时候是有道理的）。对于基于冲突的介质访问控制的特性已经作了很多研究工作，但是基本特性是，随着网络达到饱和，交付的带宽会急剧下降。

基于环的局域网络，比如令牌环网和FDDI，使用了一种分布形式的全局仲裁机制来控制共享介质的访问。当存在空的时隙时，一个特殊的仲裁令牌在环上循环移动。想发送数据包的主机等待令牌的到来，获得令牌，然后把数据包发送到环网上。发送出数据包之后，仲裁令牌被交还到环网上。实际上，作为获得环网访问的一部分，主机对于每个数据包都要执行流控。即使环网空闲，主机也必须平均等待令牌经过半个环网的时间。（这就是为什么对于FDDI而言，无负载时延比以太网要高的原因。）但是，在高负荷时，可以使用链路的全部容量。再重复一遍，这种全局仲裁体系的基本假设是网络操作的时间要远远小于在主机上的通信操作时间。

在广域网中，每个TCP连接（以及每个UDP数据包）都会通过一条从源到目的地跨越不同速度的介质的路径，该路径经过一系列的交换机、网桥和路由器。由于Internet是一个图结构，而不是一个简单的线性结构，所以在任何一点上，流入的数据流集合的总带宽可能比流出的总带宽大，其结果是网上的流量可能堆积。广域网的路由器提供了一定数量的缓冲

区来平滑随机流量的变化，但是如果竞争情况持续，这些缓冲区将最终被填满。这时，大多数路由器会简单地丢弃数据包。广域网链路可以是许多英里长的光纤，所以当数据包被发送到链路上时是很难预知到这个数据包在到达另一端时，是否会能有一个缓冲区来存放它。而且，通过一个交换机的数据流通常是不相关的，一般来讲，它们不会是来自一个并行程序的数据流的集合，这种数据流具有一个内在的端到端的流控制约，即在继续执行前要等待接收它需要的数据。TCP 层提供端到端的流控，而且动态地适应该连接所占用的路由的一些明显的特征。它假设中间节点上的竞争导致了数据包的丢失（通过超时而被检测），所以一旦它经历一次数据包丢失，就会急剧减小发送速率（通过减小它的突发窗口），当数据成功地传输后（检测到从目的节点到源节点的确认），它会慢慢增加发送速率（即增加窗口的尺寸），直到再一次经历数据丢失。所以，源节点通过检测超时和获得确认信息来控制每个数据流。 812

当然，广域网的例子是以几分之一秒级的时间尺度工作的，而并行机网络上则是以纳秒级的时间尺度工作的。所以，不能直接简单地把这种流控技术运用于并行机网络上。有意思的是，TCP 的流控机制在基于冲突的仲裁如以太网的情况下确实很有成效（部分原因可能是上层软件的额外开销时间留出了网络清空的时间）。但是，随着高速交换局域网和广域网（特别是 ATM）的出现，流控问题已经呈现出很多并行机网络的特征。大多数商用 ATM 交换机为每一条链路提供一个相当大的缓冲区（通常每条链路上有 64 至 128 个码元的缓冲），但是当超过这个数量时，它会丢掉码元。每个码元有 53 个字节，所以对于 155 Mbps 的 OC-3 速率，一个数据码元的传输时间是 2.7μs。与传统的 LAN/WAN 的端对端时间相比，缓冲区会很快被填满，当与 ATM 设置中更具有进取性的协议（如 UDP）竞争时，TCP 机制就不大有效，这提示了 ATM 标准化要包括数据链路级流控和速率控制的手段。

10.8.2 链路级的流控

基本上所有并行机的互连网络都提供链路级的流控。图 10-32 说明了基本的问题。数据从一个节点上的输出端口经过一条链路传输到另一个自治地运行的节点的输入端口上。存储部件可能是一个简单的锁存器、一个 FIFO 或者一个缓冲存储器。链路可能或长或短，或宽或窄，或同步或异步。关键的一点是，由于目的节点的运行情况，目的节点的输入端口可能没有空闲的缓冲来接收数据，所以数据必须在源节点中保持，直到目的节点可以接收为止。这可能会引起源节点缓冲区被填满，进而抑制它的源节点的发送。

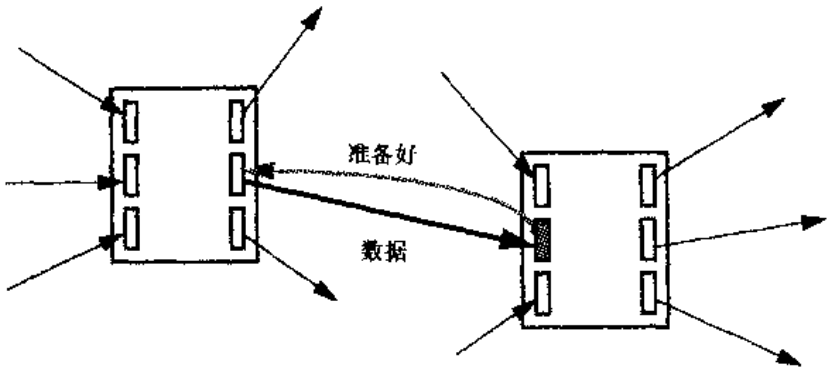


图 10-32 链路级的控制。考虑目的节点的一种情况，目的节点的输入端口可能没有缓冲区来接收传输过来的数据，源节点必须保存这些数据，直到目的节点可以接收数据为止

链路级流控的实现根据链路设计的不同而各异，但是主要思想是一样的。目的节点向源节点提供反馈来指示它是否能够接收链路上额外的数据，源结节保持数据直到目的节点指示它可以接收数据。在我们研究这种反馈是如何使用到交换机操作中之前，先看看在不同的链路中流控是如何实现的。

813

短而宽链路上的传输本质上类似于机器中的寄存器传送，只是扩展了一对控制信号。我们可以想象把源寄存器和目的寄存器扩展了一个“满-空”位，如图 10-33 所示。如果源满而目的地空，传输就开始，其结果是，目的寄存器满，源寄存器空（除非源会再从它自己的源补充到数据）。在同步操作中（如在 CRAY T3D、IBM SP-2、TMC CM-5、MIT J-machine 中），流控决定是否在一个时钟周期中发生一次数据传输。在边缘触发或者多相电平敏感的设计中很容易看到它是怎么实现的。如果交换机的操作是异步的，其行为很像是在自定时设计中的寄存器传送。源节点在缓冲区满并准备好传输数据时激活一个请求（req）信号；目的节点在它的输入端口可用时，使用该信号接收数据，并且在接收到数据后，激活一个确认（ack）。对于短而窄的链路，除了每一对请求/确认握手信号要传输一串物理单元以外，其余处理是类似的。

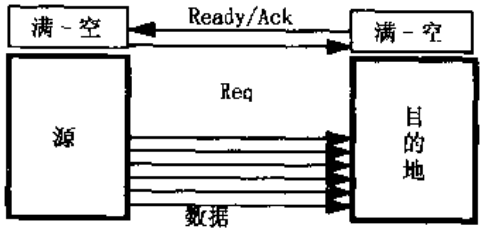


图 10-33 简单链路级的握手。源节点准备发送一个流控单元时发出它的请求；目的节点在准备好接收下一个流控单元时会发送一个流控单元接收确认。之后，源节点会重复上述方法来传送流控单元

请求/确认握手信号可以看成是在源节点和目的节点之间传送一个惟一的令牌或者信用。当目的节点释放输入缓冲区后，它就把令牌送到源节点（即增加其信用值），源节点在发送下一个流控单元时要使用这个信用，而且在重新获得信用之前必须等待。对于长链路，这种信用机制可以被扩展，使得与链路传播延迟相关的整个流水线都能被填满。假设链路足够长，同时可以有几个流控单元同时传输，如图 10-34 那样，反方向传播的确认可能需要几个时钟周期，所以多个确认（信用）也可以处于正在传送的状态。这种明显的基于信用的流控是让源节点来记录目的节点的可用输入缓冲的数目。计数器值初始化时等于缓冲区的大小，发送一个流控单元，计数器减一；如果计数器达到零值时，输出阻塞。当目的节点从输入缓冲区移走一个流控单元时，它向源节点返回一个信用，使源节点对计数器加一。如此，输入缓冲区永远不会溢出，总是有空间来把链路上的数据放到缓冲区中。这种方法最适用于宽链路，因为宽链路会留有专用的控制线用于反向的确认信号的传输，对于把确认信号在反方向的信道上多路复用传送的窄链路，可以通过成块地传送信用，避免对每个流控单元发送确认。但是，问题仍然存在，这种方法在信用令牌丢失时不是非常健壮。

814

理想的情况是：数据流能向前平稳地传输时不需要流控。流控机制应该是一个微调输入速率使其与输出速率逐渐匹配的管理者。链路上的传播延迟给了系统这样的动力。另一种链路级信用的方法是把目的节点的输入缓冲区看成是一个分级的容器，这个容器有低水位线和高水位线，如图 10-35 所示。当容器的充满程度低于低水位线时，向源节点发送 GO 信号，当容器的充满程度高于高水位线时，会产生 STOP 信号。低于低水位线的容量要足够容忍一

个完整的往返延迟时间（这种往返延迟时间是指：GO 信号传播到源节点，被处理，直到随后的数据流的第一个流控单元传到目的节点的时间总和）。除此之外，还必须有足够的高于高水位线的净空高度来接收一个往返时间内在途中的所有流控单元。这种方法的优点是：在低水位线[⊖]以下时，可以发冗余的 GO 信号；而在高水位线[⊖]以上时，可以发多个 STOP 信号而不会产生任何负面影响，所以，简单地在两个特定的区域中周期性地发送相应的信号即可。在输入缓冲区的高、低水位线之间增加存储容量可以减少流控信号使用的链路带宽的部分。比如在 Cal Tech 路由器（Seitz and Su 1993）和随后的 Myrinet 的商用产品（Boden et al. 1995）中就使用了这种方法，在调制解调器中也使用了类似的技术。 815

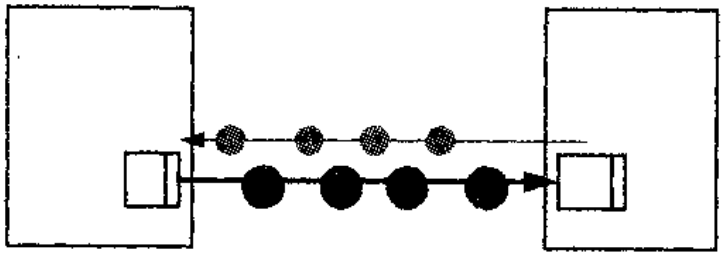


图 10-34 长链路上的正在传输的流控单元和确认，在长链路上，为了使链路填满，流控机制应比较宽松。在确认返回前可以先把几个流控单元发送到线路上

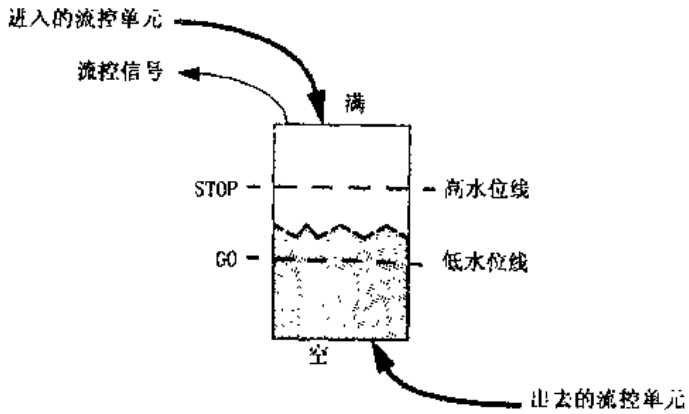


图 10-35 松弛的缓冲操作。当充满程度低于低水位线时，向源节点发送 GO 信号；当它超过高水位线时，产生 STOP 信号

值得注意的是，主机和交换机间链路同交换机和交换机间链路一样都使用链路级的流控。实际上，它也经常被扩展到处理器和网络接口之间的界面。但是，不同的接口使用的技术是不同的，比如，在 Intel Paragon 中，175 MBps 网络链路都是具有非常小的流控单元缓冲的短链路，而网络接口（NI）却有大的输入和输出 FIFO。通信辅助部件包括一对允许在主存和网络的 FIFO之间有 300 MBps 的突发传输的 DMA 引擎。一个基本点是输出（输入）缓冲区在突发传输中间不能达到满(空)，因为不能传输而占用总线事务可能会导致性能下降和一些潜在的死锁，所以突发传输必须与缓冲区的中间区域相匹配，而高低水位线必须与NI与DMA

⊖ 原文为高水位线。——译者注
⊖ 原文为低水位线。——译者注

控制器之间的控制信号的往返时间相匹配。

10.8.3 端到端的流控

如果拥塞持续，链路级的流控行使了一定数量的端到端的流控。因为，缓冲区会被填满，直到源主机节点的数据流将被控制，这称作反向压力作用。比如，如果 k 个节点向同一个目的节点发送数据，它们的平均带宽最终会降低到输出带宽的 $1/k$。如果交换调度是公平的，通过网络的所有路由是对称的，反向压力作用将完全能做到这一点。问题在于当源节点感知到反向压力，并调整它们的输出流之前，在树中所有从热点到源节点的所有缓冲区都已经被填满了。

816

1. 热点

热点问题在由成百上千个处理器构造机器的技术出现时，引起了相当的注意（Pfister and Norton 1985）。如果 1 000 个处理器向任何一个目的节点传送平均超过它们的网络流量 0.1%的数据流，目的节点就已经饱和了。如果这种情况持续下去，比如说如果要经常地访问一个重要的共享变量，在该目的节点之前就会形成一个饱和树结构，一直通往所有的源节点。这时，系统中所有其余流量也会被严重的阻塞。这个问题在蝶状网中特别有害，因为到每个目的节点只有一条路，从一个目的节点有大量的共享路径，自适应路由会使热点问题变得更糟，因为指向热点的网络流量必定会碰到竞争，并且被转移到其他替代的路径上，最终，整个网络将被阻塞。即使大的网络缓冲区也不能解决这个问题，它们也仅仅是延迟现象发生而已。热点消除的时间与网络上缓冲的热点流量的总量成正比，所以自适应算法和大缓冲区增加了在负荷去除后热点消除的时间。

人们已经开发了各种不同的机制来缓解热点发生，比如像第 7 章讨论的那样，让所有需要对共享变量加一的节点执行一个并行扫描操作，或者在网络中实现组合的 fetch&add 操作（Pfister et al. 1985; Gottlieb, Lubachevsky, and Rudolph 1983）。但是，这些方法仅仅能够应付产生问题的流量在逻辑上相关的情况。更根本性的问题在于链路级的流控就像在高速公路上停车。一旦交通拥塞形成，你就被钉住了。更好的解决办法是在这个时候不上高速公路。网络中的数据包越少，正常的机制越能更好地使通信流量到达目的节点。这也就是为什么 BBN 蝶状网在有冲突时，拆除由消息建立的电路的原因之一。

2. 全局通信操作

人们已经通过完全平衡通信模式观察到简单的反向压力的问题。完全平衡通信模式的例子有：每个节点向其余各个节点发送 k 个数据包。这种情况在很多时候会发生，包括转置一个全局矩阵，在阻塞的和回路的布线之间的转换，或在 FFT 的抽选步骤（Brewer and Kuszmaul 1994; Dusseau et al. 1996）。即使拓扑结构足够的健壮以至于可以避免这些操作的严重的内部瓶颈问题，即它是一个真正的胖树结构而不是低维度的网格（Leignton 1992），一个瞬时的积压也会有一个级连的效应。当一个目的节点来不及接收网络上的数据包时，开始形成积压，如果给了它优先权来接收网络上的数据包，这个节点将来不及向其他节点发送，因而使那些节点发送的会比接收的多，使积压扩展。

817

简单的在全局通信路径上的端对端协议已经在实际中缓解了这个问题。比如，一个节点在发送一定数量的数据后可以一直等待直到它收到同样数量的数据；或者，它可以等待它发送的这些数据块的确认信息。这些防范措施使处理器之间更加密切同步，而且在通信流之间

插入了小的间隙，这样减弱了处理器与处理器之间通过网络交互的耦合。（有趣的是，这种技术也被用于高度繁忙的桥梁上的计时灯，间隔一个短的时间，汽车周期性地进入桥梁。这种方法降低了随机的瞬时阻塞，并避免了级连的阻塞。）

3. 接纳控制

对于浅的直通网络，时延在达到饱和之前是很小的，实际上，在大多数现代的并行机网络上，单个小消息（或者几个消息）会占据从源到目的的整个路径，如果远程的网络接口没有准备好接收消息，最好是把消息保存在源节点的网络接口中而不要把它放在网络中阻塞网络流量。实现这种思想的一个方法是执行网络接口之间基于信用的流控。考察在一些网络中这种技术的使用的研究（Callahan and Goldstein 1995）表明：每对网络接口之间允许一个正在传输的消息能产生大的吞吐量并维持低的时延。

10.9 案例分析

从实际设计和工程的角度来看，网络是一个迷人的研究领域，因为它们只是做一个简单的操作，即把信息从一处传到另一处，但是却有着不同设计的巨大空间。尽管网络最明显的特征是它的拓扑结构、链路带宽、交换策略和路由算法，在完整地描述设计时，还要说明更多的特征。这些特征包括周期时间、链路宽度、交换机缓冲区的容量和分配策略、路由机制、交换机输出选择算法和流控机制。虽然设计的每一个成分都可以被单独地理解和优化，但是它们之间的交互作用决定了在特定的节点结构环境中，任何一个特定流量模式下的网络性能和平台所运行的程序中内嵌的相互依赖关系。

本节将总结在重要的商用型和研究型的并行体系结构中一些具体的网络设计要点。使用本章建立的框架，它系统地勾画了一些关键的设计参数。

10.9.1 CRAY T3D 网络

CRAY T3D 网络是由一个三维双向的花环组成，包含多达 1 024 个交换机节点，每个开关节点和一对处理器直接相连[1]，每个通道的数据速率是 300 MBps。每个节点在一个主板上，该主板上有两个处理器和存储器。每个机箱里最多有 128 个节点（256 个处理器），更大的配置是通过用电缆线将机箱连在一起而构成。使用维序、直通、数据包交换路由。高层次的系统设计在很大程度上影响着该网络的设计。在一组单一的物理链路上支持逻辑上独立的请求和响应网络，各自都有两个虚通道，以避免死锁。数据包是变长的，但必须是 16 位的整数倍，如图 10-36 所示，网络传输处理包括不同的读、写以及启动远程块传送引擎（BLT）能力，BLT 基本上是一个 DMA 设备。第一个物理单元总是包括路由，接着是目的地址，数据包类型或命令码。有效负载的其余部分由数据包类型决定，包括相关地址和数据。除了路由物理单元外，数据包的所有部分都由奇偶校验保护。如果路由损坏，数据包就会被错误地传送，在目的节点可以检测到错误，因为目的地址与数据包中的值不匹配[2]。

818

[1] 与立方体的边界相连的专用 I/O 网关节点包括两个处理器，每一个连接到一个与 x 和 z 维度相连的二维交换机节点上。

[2] 这种差错校验的方法揭示了网络的微妙侧面。如果路由被破坏，存在非常小的概率，数据包会正好在错误的时刻拐错方向，从而与另一个数据包碰撞，导致网络的死锁。在这种很难出现的情况下，端节点的差错检测将不会介入。

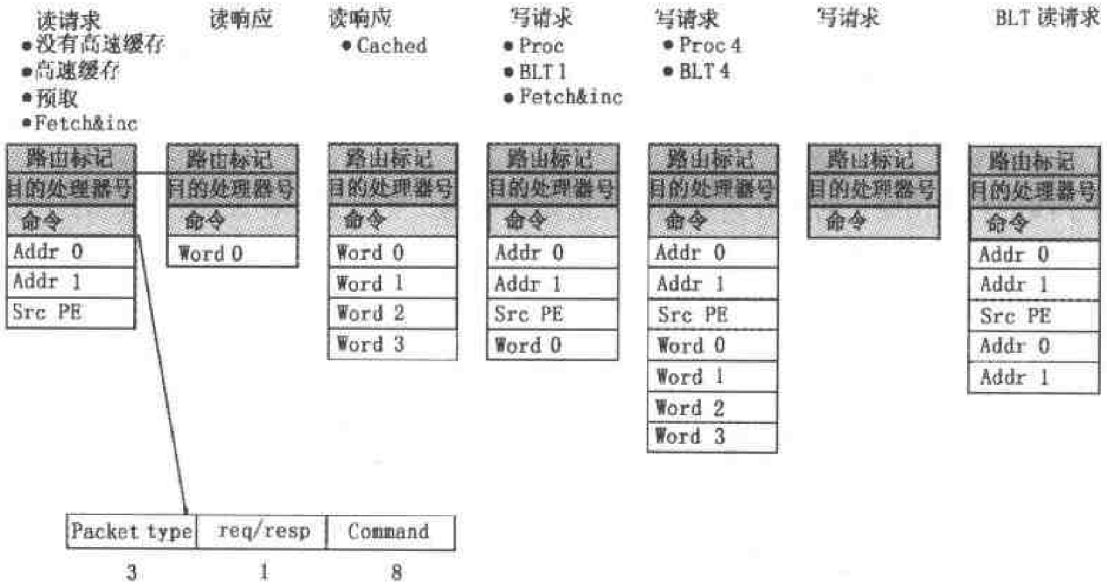


图 10-36 CRAY T3D 包格式。所有的数据包由一系列 16 位的物理单元构成，前三个物理单元是路由和标记（Route tag）、目的处理器号（Dest PE）和命令（Command），数据包的其余部分的解析和处理由标记和命令来决定

T3D 链路是短而宽的同步链路。每一个单向的链路的宽度是 16 个数据位，8 个控制位，时钟频率是 150 MHz，流控单元和物理单元都是 16 位，用控制位中的两位来表明物理单元类型（00：没有信息，01：路由标记，10：数据包，11：最后一个数据包）。路由标记物理单元和最后一个数据包物理单元提供了数据包的帧封装。还有两位控制位标明虚通道（req-hi，req-lo，resp-hi，resp-lo），其余 4 条控制线是反方向的确认，每个虚通道一个。因此，以一个个时钟周期为基础，每个虚通道和物理单元的流控可以在虚通道间交错重叠。

819

交换机是用 6 个 10-K 门阵列实现的，构成 3 个独立维的路由器。在每一个交换机中有少量的缓冲区（三维中每个维的 4 个虚通道分别有 8 个 16 位的包缓存），当阻塞时，数据包被压到缓冲区里。交换机中有足够的缓冲来存储小数据包。输入端口根据一个简单的运算操作决定希望的输出端口。路径长度被减一，如果结果非零，数据包继续按当前维当前方向传递；否则，它就被送到下一维。每一个输出端口对请求该输出端口的输入端口使用轮转优先权，对每一个输入端口而言，也存在对请求该输出端口的虚通道的轮转优先权。

网络接口包括了 8 个数据包缓冲区，每个虚通道两个。整个数据包在被传输到网络之前缓冲在源节点的网络接口中，在被传到目的处理器或存储器系统之前，缓存在目的节点的网络接口中，这种存储-转发的延迟有效的去除了网络和节点操作的耦合。除了主数据通信网络之外，还提供了逻辑与（栅障）和逻辑或（存在）的树状网络。

双向链路的出现提供了在每一维的两种可能的选择，在源网络接口处执行一个表查询来选择确定性路由，包括分别在三个维度上的方向和距离。每个程序都占有机器的一个划分，包括一个逻辑上连续的任意形状的子阵列（在操作系统配置下）。在通信辅助部件中的移位和屏蔽逻辑把与划分相关的虚拟节点地址映射到一个全机范围中的逻辑 $<X、Y、Z>$ 坐标地址。机器的一些节点可以被配置为空闲节点，它们可以代替一个失效的节点。$<X，Y、Z>$ 用作路由查询的一个索引，所以网络接口路由表提供了最后一层转换，通过从源节点的 $<\pm\Delta x，\pm\Delta y，\pm\Delta z>$ 路由来确定物理节点。这个路由查询也指明了在 4 个虚通道中使用

了哪一个。为了在请求或者响应（虚拟）网络中避免死锁，高通道用于穿越回卷链路的数据包，而低通道用于其他数据包。

10.9.2 IBM SP-1、SP-2 网络

在 IBM SP-1 和 SP-2 并行机（Abali and Aykanat 1994；Stunkel et al. 1994）中使用的网络比 CRAY T3D 的网络更为通用，但性能较低，不支持两阶段的请求-应答操作。它是包交换网络，采用直通式基于源路由，没有虚通道。交换机有 8 个双向 40 MBps 端口，能够支持许多不同的拓扑结构，但是，在 SP 机中，交换机集合是在板上作为 4 元二维蝶形网布置的，有 16 个内部连接和与交换板同一个机架上的内部主机相连接，到其他机架有 16 个外部连接，如图 10-37 所示。不同机器的机架级拓扑各异，但典型结构是蝶形结构的变种。第 1 章的图 1-23 显示了在 Maui 高性能计算中心的 IBM SP-2 的配置。单个机柜在底层提供连接 16 个内部节点和 16 个外部链路的第一层路由，额外的机柜在机柜集合之间提供互连。这些额外层次的连线铺设在机房地板下面。因为物理拓扑没有用硬件固定，网络接口通过查表为每一个送出的消息插入路由。

820

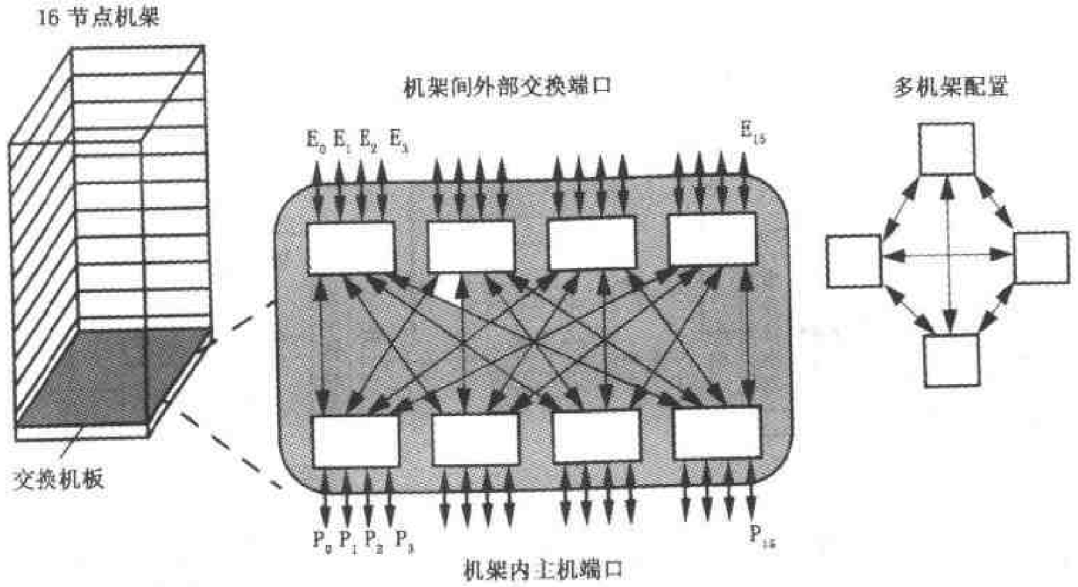


图 10-37 SP 交换机的封装。机架上的交换机集合提供 16 个内部端口和 16 个外部端口之间的连接

数据包最多由 255 个字节的序列组成，第一个字节是数据包的长度，接着是一个或多个字节的路由字节，然后是数据字节。每个路由字节包含两个 3 位的输出说明、一个选择位。链路是同步的，宽且长。单个 40 MHz 时钟频率被送到所有的交换机，每个链路的延迟调整为时钟周期的整数倍（板间信号是 100 K ECL 差分线对信号，板内的时钟树也是 100 K ECL。）链路由 10 根线组成：8 根数据位，一根帧“标记”控制位，一根反向流控位。因此，物理单元是一个字节，标记位表明长度和路由物理单元。流控单元是两个字节，用两个时钟周期来指示在接收缓冲区中有两个字节缓冲可用。任何时候，数据/标记流控单元的流沿着该链路沿一个方向传播时，一个信用令牌流沿相反方向传播。

交换机在每个输入端口提供了 31 个字节的 FIFO 缓冲，允许链路为 16 个流控单元长。除此之外，在每个输出端口上有 7 个字节的 FIFO 缓冲。还有一个具有 128 个 8 字节块的共

821

享中央队列。如图 10-38 所示，交换机包括一个不缓存的按字节串行的交叉开关和一个 128×64位的双端口 RAM 作为输入与输出端口间的互连。当数据包的两个字节到达输入端口后，输入端口的控制逻辑可以请求希望的输出端口。如果这个输出端口可以使用，数据包经过交叉开关，直通到输出端口，每个交换机的最小路由延迟是 5 个时钟周期。如果该输出端口此时不能使用，数据包将进入输入的 FIFO 中。如果输出端口仍旧阻塞，数据包就会按 8 字节的块装入中央队列。因为中央队列在每个时钟周期只能接收一个 8 字节的输入和一个 8 字节的输出，其带宽与交换机 8 字节串行的输入及输出端口相匹配。在内部，中央队列被组织成 8 个 FIFO 链表，每个输出端口一个，为链路使用了一个辅助的 128 位×7 位的 RAM。为每一个输出端口保留有一个 8 字节的块。因此，当负载小时，交换机工作在字节串行模式；而当竞争发生时，它利用中央队列来时分复用这些 8 字节的块，此时以输入作为串行变并行，以输出作为并行变串行的部件。

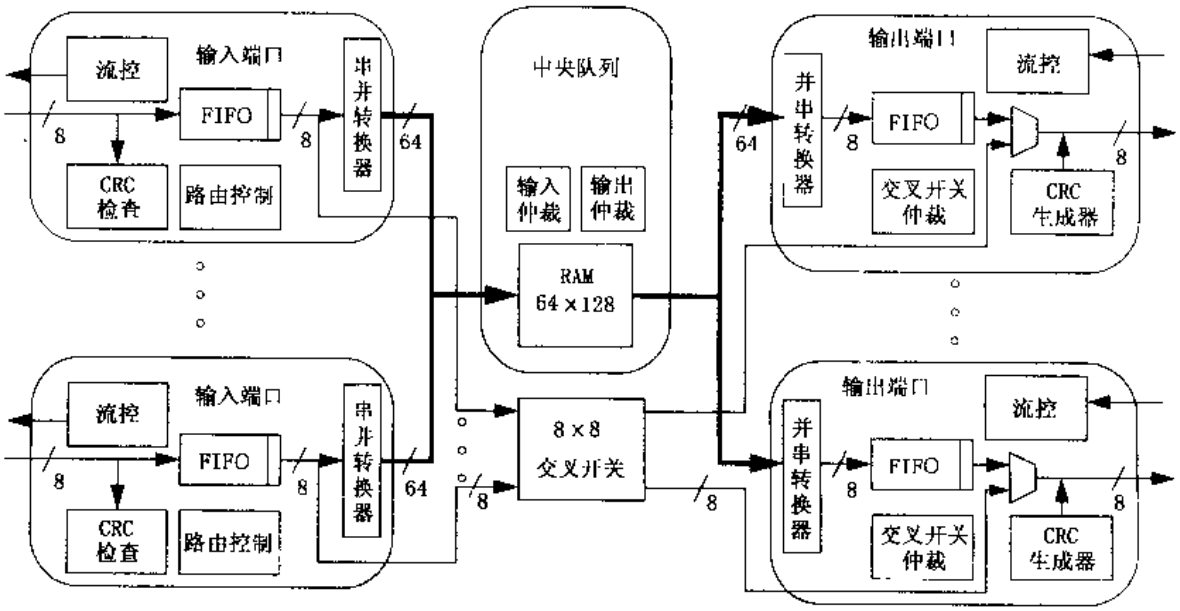


图 10-38　IBM SP（Vulcan）交换机设计。IBM SP 的交换机为无缓冲数据包能直接从输入端口传送到输出端口使用了交叉开关，但是所有的缓冲数据包被分流到一个公共存储器池中。由仲裁逻辑调度流控单元对中央队列 RAM 的进入和流出

每个输出端口基于 LRU 算法对到来的请求进行仲裁，中央队列中的块的优先权比输入 FIFO 中字节的优先权要高。中央队列按 LRU 顺序为输出端口服务。对那些以未阻塞输出端口为目的的输入，中央队列给予优先权。

SP 网络有三个特殊的方面。第一，由于其操作是全局同步的，不是每个数据包的封装都带有一个 CRC 信息，而是将时间划分成 64 个周期大小的“帧”。每一个帧的最后两个物理单元携带 CRC。输入端口检查 CRC，而输出端口产生 CRC（在去掉已使用的路由物理单元后计算得到的）。第二，交换机由单个芯片组成，每一个交换芯片被另一个完全相同的交换芯片遮蔽，引脚是双向 I/O 引脚，所以其中一个芯片仅仅是检查一下另一个芯片的操作（这可能会检测到交换机错误，但不是链路错误）。最后，这种交换机支持一种用于各种诊断用途的电路交换“服务模式”。在改变模式之前，网络腾空它上面的所有数据包。

10.9.3 可扩展一致性接口

可扩展一致性接口（Scalable Coherent Interface）提供了高性能互连的一个良好的案例，因为它是经过标准化的过程之后出现的，而不是作为专用的设计和学术的建议出现的。它的标准化过程经历了相当长的一段时间，当其实现工作开始时获得了青睐。一些厂商已经采用了它，尽管在许多情况下只遵循了它的部分规范。完整的 SCI 规范用于 HP/Convex 的 Exemplar 和 Sequent 的 NUMA-Q 机的互连。CRAY SCX 的 I/O 网络在很大程度上也是建立在 SCI 基础上的。

SCI 设计的一个关键元素是它建立在称之为小环（ringlet）的单向环的概念上，而不是建立在双向链路上。小环由交换机相连从而形成大的网络。SCI 的规范定义了三层：物理层、数据包层、事务层。物理层被规定成两个 1 GBpS 的形式。数据包由 16 位单元的序列组成，很像 CRAY T3D 的数据包。如图 10-39 示，所有的数据包包括一个目标 ID（TargetID）、命令（Command）、源 ID（SourceID），随后是零个或者更多的由命令规定的单元，最后是一个 32 位的 CRC。一个特殊的方面是源和目标 ID 是任意的 16 位的节点地址。数据包不含路由。从这个意义上说，该设计像一条总线：源把目标地址放在互连机构上，由互连机构决定如何把信息放到正确的地方。在环中，这个任务实现起来很简单，因为数据包沿环流动而目标可以抽取数据包。一般情况下，交换机使用表驱动路由把数据包从一个环传递到另一个环上。

822 ~ 823

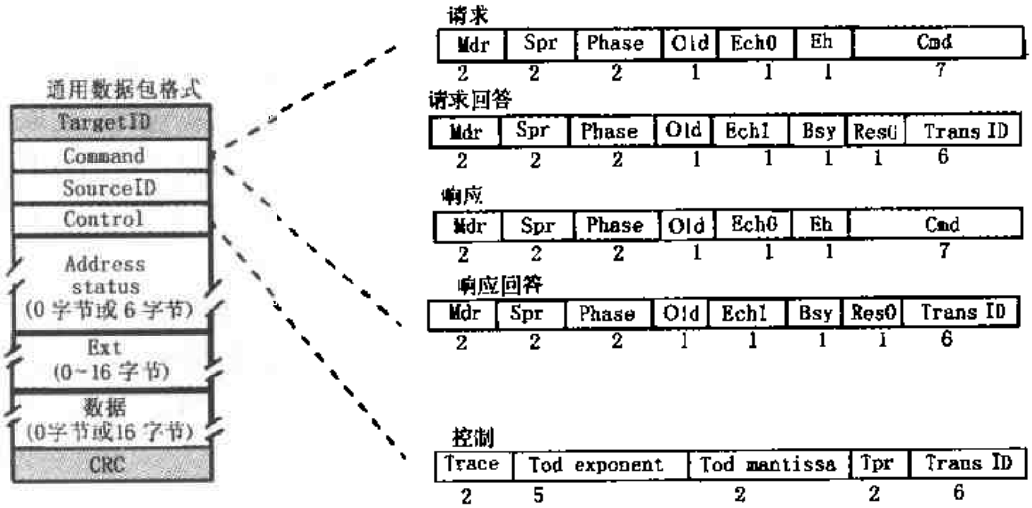


图 10-39 SCI 数据包格式。SCI 操作包括一对事务：一个请求和一个响应。由于其基于环的支撑基础，每一个事务包括从源节点向目标节点运输一个数据包，从目标节点向源节点返回一个回答（这个回答沿环中数据包未走完的路径回答源节点）。所有的数据包由一系列的 16 位的单元组成，前三个单元指示目的节点，命令和源节点，最后一个单元是 CRC。请求包括一个 6 字节地址、选项扩展和选项数据。响应有相同的格式，但地址部分携带的是状态信息。两种回答包都只包含最少的 4 个单元。命令单元通过 phase 和 ech 域表明数据包的类型。它还描述了要执行的（请求 cmd）或匹配的（trans id）操作。其余域和控制单元涉及低层的问题，比如，如何让数据包排队以及在数据包层次处理重传的问题

一次 SCI 事务，比如读或写，包括两个网络事务（请求和相应），在每个小环上每个操作都分成两个阶段，让我们一步一步地来看。源节点向目标节点发出一个请求数据包，这个请求数据包在源小环上环绕移动。如果目标节点和源节点在同一个环上，目标节点将会从环

上抽取这个请求数据包，用回答包替换它放回到环网上，这个数据包会继续绕环传送回到源节点，在那里源节点把它取消。回答数据包的目的是通知源节点数据原始包或者已被目的节点接收到，或者被拒绝接收；对后一种情况，回答数据包中包含 NACK。被拒绝接收的原因或者是因为缓冲区已满，或者是因为数据包已经被破坏。在源节点对正在传送的数据包维持一个计时器，这样它就可以检测到回答数据包是否丢失或损坏。如果目标节点与源节点不在同一个小环上，环中的交换节点充当目标节点代理的角色，交换节点接收数据包并在它成功地缓冲该数据包后发送回答数据包。因此，回答数据包只是告诉源节点数据包成功地离开了该小环，交换节点会沿着通往目标节点的路由向另外一个环启动这个数据包的发送。在接收到请求后，目标节点将开始一次响应事务。在返回源节点的路径上，这也将在每个小环上有一个数据包阶段和一个回答阶段。请求回答数据包把分配给这次事务的事务 ID（Trans ID）通知源节点，它被用来与最终的响应匹配，这很像一条事务拆分型总线。

在 SCI 中，各层次没有明确的封装，它们在数据包格式上有些模糊，用几个域控制队列管理和重传机制。控制 tpr（事务优先权）、命令 mpr（最大环优先权）和命令 spr（发送优先权）一起决定了数据包的 4 种优先级别之一。事务优先权最初由请求者设置，但是实际的发送优先权则是由沿路由的节点根据它们队列中其他被阻塞的事务而建立的。阶段（Phase）和忙（Bsy）域作为源节点和目标节点之间流控协商的一部分来使用。

在 Sequent 的 NUMA-Q 机中，18 位宽的 SCI 环由数据泵（DataPump）以 1 GBps 的结点到网络的带宽在四芯线中直接驱动，节点到网络的带宽为 1 GBps。传输层严格地遵循请求－应答协议，当数据泵把一个数据包放到环网上，它在对该数据包的回答返回之前，一直在自己的输出缓冲中保留着该数据包的副本。当数据泵发现环网上有进入的数据包时，它取走该数据包，并发出一个肯定回答包。如果数据泵检测环网上有一个目标地址是自己的数据包，但它却没有空间取走该数据包时，它会发送一个否定回答包，这样发送者将会尝试重新传送该数据包（发送者此时会在输出缓冲保存着这个数据包的副本）。

824

因为在环网上通信的时延随着网上节点数目的增加而线性增长，我们期望通过以任意的网络拓扑互连较小的环的方法来构造大型高性能的 SCI 系统。比如，性能研究表明单个环能够有效地支持 4 到 8 个高性能处理节点（Scott，Vernon，and Goodman 1992）。

10.9.4 SGI 的 Origin 网

SGI 的 Origin 网是以一个被称为 SPIDER 的灵活的交换机为基础的，该交换机支持 6 对单向链路，每对链路在两个方向上提供 1.56 GBps 以上的总带宽。每个交换机连接两个节点（4 个处理器），所以总共有 4 对链路连到其他交换机上。这种构造模块用一种与超立方体相关的拓扑结构系列组织到一起，如图 10-40 所示。链路是柔性的（长而宽）最多可达 3 m 长的缆线。消息以流水方式通过网络，从引脚到引脚通过路由器本身的时延是 41 ns。路由是表驱动的，所以作为网络初始化的一部分，在每个交换机中建立路由表。这就允许路由是可编程的，所以它可能支持各种不同的配置，可以形成部分使用的网络（即不是使用所有的端口），以及绕过失效部件的路由。路由器是分别供电的，并且通过硬件的错误重传机制来对每一条链路提供错误保护，所以也提高了可用性。交换机为请求和应答提供了分离的虚通道，并且以数据包寿命方式支持 256 个级别的消息优先权。

825

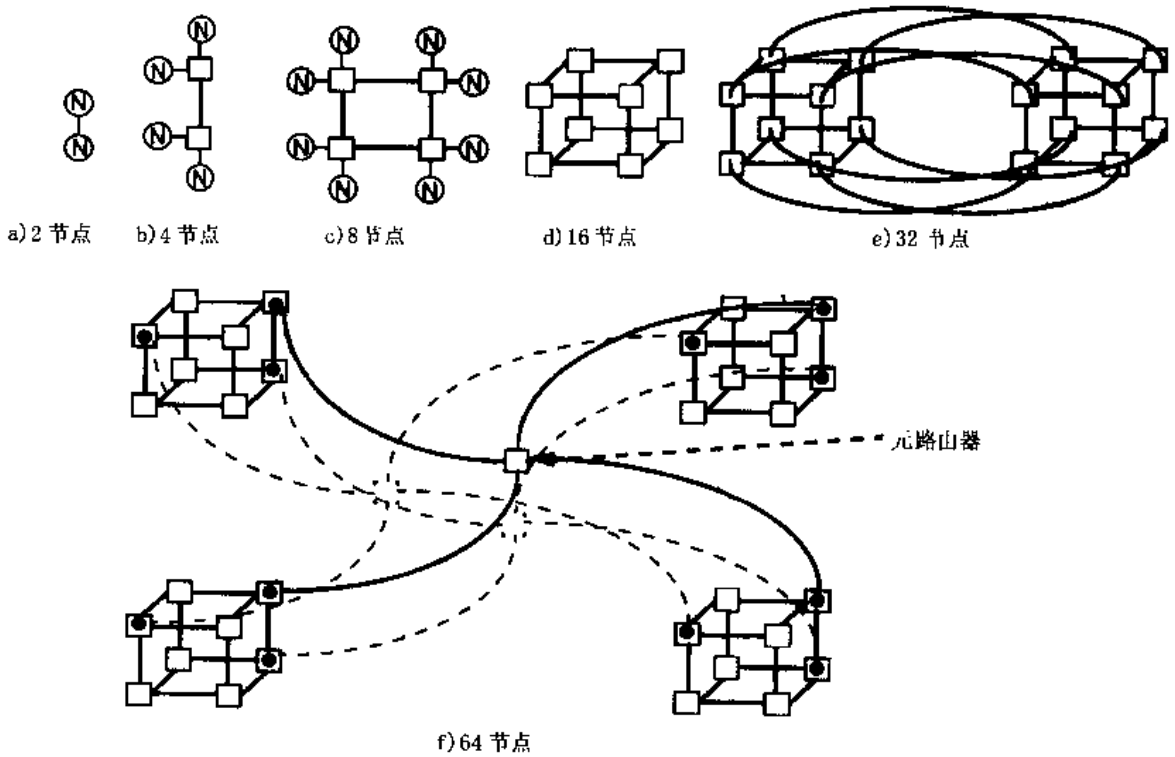


图 10-40 在 SGI Origin 多处理器系统中的网络拓扑和路由器之间的连接。超立方体结构用于最多 32 个节点的情况，如 a) ~ e)，超过这个范围之后则使用了一个胖树变形结构。d) ~ f) 的配置为简化起见只显示了路由器，省略了每个路由器上连接的两个节点。配置 f) 也只显示了跨过超立方体的一部分胖立方体连接，连接 16 个节点的子立方体的路由器被称为元路由器。对于一个 512 节点（1 024 个处理器）的配置而言，每个元路由器本身将被一个五维的路由器超立方体结构所代替

10.9.5 Myricom 网络

作为最后一个研究的例子，我们简略地考察一下在一些机群系统中使用的 Myricom 网络（Boden et al. 1995）。在第 7 章 7.7 节描述了网络接口卡中的通信辅助部件，这里我们关心的是用于构造一个可扩展互连网的交换机。或许其设计中最有趣的侧面是它的简单性。基本的构造块是一个具有 8 个双向端口的交换机，每个端口为 160 MBps。物理链路是 9 位宽的长线，最长可达 25 m，最多可同时传送 23 个物理单元。编码方式允许控制流、帧符号及“间隔”（或空闲）符号与数据符号交错传输。信号不间断地在跨越链路两端的交换机之间传输，所以交换机可以判断它的哪些端口处于连接状态。一个数据包不过是一串路由字节，后跟一系列有效负载字节和一个 CRC 字节。数据包的结尾由间隔符号的出现而被分界，而一个数据包的开始则由一个非间隔符号来指示。 826

交换机的操作非常简单：它将接收到的数据包的第一个字节去掉，然后把路由字节的内容加上输入端口号计算出输出端口号，向那个端口传送数据包的其余部分。交换机使用每个输入和输出端口上少量的缓冲来实现虫孔（wormhole）路由。通过一个交换机的路由延迟小于 500 ns。交换机可以连接成一个任意的拓扑结构。构造网络物理互连接上的有效路由是主机通信软件的责任。如果一个数据包试图从一个无效或未连接的端口上走出来，它就会被丢弃。当所有的路由字节都被用完之后，数据包就应该到达一个网络接口卡（NIC）了。消

息的第一个字节中有一位指示它不是一个路由字节，其余位指明数据包的类型。所有更高层次的数据包的格式形成和处理都由网络接口卡和主机来完成，互连网络仅仅传送这些位而已。

10.10 结论

并行计算机网络呈现了一种综合了几个设计层次的丰富而多样的设计空间。物理链路层问题代表了在计算机设计中最有意思的电气工程方面的问题。为了跟上不断增长的处理器速率，人们进行了不懈的努力，使链路的速率也一直在改善。目前，我们已经看到在铜绞线上的多吉比特的传输速率，在不远的将来，并行光纤技术为每条链路提供多吉比特的传输速率也将成为可能。在这样的速率下，主要问题之一是错误的处理。如果物理介质误码率是每位10^{-19}，数据以每秒10^9字节的速率传送，那么大概每十分钟链路上就会出现一次错误。因为机器中有几千条链路，所以每秒钟都会出现错误。这些错误都必须能够被迅速地检测和纠正。

交换机到交换机层的设计也提供了一系列丰富的折中，包括如何将问题的一些侧面下推到物理层和上推到数据包层。比如，流控可以内置在链路上数字符号的交换之中，或由与链路上的数据包相关的上一个层次来解决，或者由与端到端发送的消息相关的更高一层来实现。交换机本身的设计就有巨大的选择空间，它们也受到下层的工程的约束和上层的设计需求的限制。

即使是基本的拓扑结构、交换策略、路由算法都体现了下层的工程需求和上层的设计需求之间的折中。比如，我们已经看到了一些基本的问题，像交换机的度，是如何严重地依赖于与目标技术相关的成本模型以及目标工作负载通信模式的特征。最后，和体系结构的很多其他方面一样，关于应该把节点到节点通信抽象的多少高层语义嵌入到网络本身的硬件设计之中还有相当大的争论空间。并行计算机网络设计的整个领域注定是未来许多年令人兴奋的话题。特别是在并行计算机网络和不断进步的可扩展局域网和系统域网之间有着概念和技术上的密切联系和相互影响。

827

习题

10.1 一个数据包的格式是10个字节的路由及控制信息，6个字节的CRC校验和其他尾部信息，有效负载包含一个64字节高速缓存块，同时有8个字节的命令及地址信息。如果原始链路带宽是500 MBps，用这种格式高速缓存块传输的有效数据带宽是多少？如果高速缓存块为32字节、128字节、4 KB页的传输，其有效数据带宽又分别是多少？

10.2 假设链路是1字节宽，在频率为300 MHz网络上，对P个节点该网络的节点间平均路由距离为$\log_4 P$，假设每一跳由4个时钟周期的延迟来决定路由，P的范围是16～1 024个节点，试比较在存储-转发方式路由和直通路由下，80字节的数据包的无负载时延。

10.3 针对分片成1 KB数据包的32 KB传输的情况做习题10.2中的比较。

10.4 作为一种常见的页尺寸的传输，基于下面的假设，找出一个为8 KB消息分片的优化策略。设每个分片有一个固定的o个时钟周期的额外开销，通过源和目的地的网络接口有一个存储转发的延迟，数据以直通经过网络时有R个时钟周期的路由延迟，数

据传输速率为每个时钟周期 B 个字节。

10.5　假设一个 $n \times n$ 的双精度数的矩阵按行被安排在 P 个节点上，为了对列计算，我们希望将它转置，按列分布，在这个操作中有多少数据穿过对分面？

10.6　考虑一个二维花环构成的直接网络，具有 N 个节点，链路带宽每秒 b 个字节。请计算对分面带宽和平均路由距离。比较采用式（10-6）估计的聚合通信带宽和基于对分面的估计带宽。此外，假设每个节点仅与相隔两跳的节点通信，那么可用的带宽是多少？如果每个节点仅与同一行中的节点通信又将如何？

10.7　试说明如何把一个 N 节点的花环嵌入在一个 N 节点的超立方体中，使这个花环中相邻的节点（即距离为 1 的节点）也是超立方体中相邻的节点。（提示：考虑当这些地址按葛莱码顺序排列时的情况。）将这种嵌入方法推广到更高维的网格的情况。 828

10.8　针对超立方体网络分析式（10-6），在最后一步中，考虑行是由网格到超立方体的葛莱码映射定义的，并按此处理。

10.9　计算一个 N 个节点的线性阵列的平均距离（假设 N 是偶数），计算一个有 N 节点的二维网格和二维花环的平均距离。

10.10　对于所感兴趣的工作负载和特定的网络拓扑，通过决定每个数据包经过的通道的平均数 h_{ave}，能得到对通信时间的比较精确的估计。有效的聚合带宽最大为

$$\frac{\| C \|}{h_{ave}} B$$

假设程序调度把处理器作为 $\sqrt{p} \times \sqrt{p}$ 的逻辑网格处理，每个节点与 4 个方向和对角线方向的 8 个相邻节点交换 n 个字节的数据。请给出网格和超立方体上的通信时间的估计值。

10.11　说明如何把一个 N 节点树的一边伸展穿过超立方体的两条边，将这个 N 节点树嵌入到一个 N 节点的超立方体中去。

10.12　运用电子表格建立图 10-12 的比较，其设计出发点是 10 个周期的路由延迟和 16 位宽的链路。将这个比较扩展到 1 024 字节的消息的情况。你能得出什么结论？

10.13　试验证在图 10-14 中当路由延迟等于通道时间时，在相等引脚的尺度下能达到最小的时延。

10.14　在基于式（10-7）的相等对分面尺度下，试推导出能实现最小时延的维度公式。

10.15　在相等对分面尺度规则下，具有 100 万个节点的二维网格的链路应有多宽？

10.16　对于相等引脚和相等对分面两种情况，像图 10-17 中所示的那样，对尺寸大约相同的二维和三维立方体的有负载行为进行比较。

10.17　设二维网格采用 Δx，Δy 路由，试说明用于在交换机中计算路由函数的布尔逻辑。

10.18　如果采用 Δx，Δy 路由，决定数据包头中的距离计数值对数据包尾中的 CRC 有什么效应？如何在交换机的设计中处理这个问题？

10.19　试说明在超立方体维序路由的交换机中计算路由函数的布尔逻辑。

10.20　试证明在超立方体中的 e-立方体路由是免死锁的。

10.21　在二维网格上建立基于表的 Δx，Δy 路由的等价方法。

10.22　说明如果允许图 10-24 所禁止的折转对，那么将导致复杂的回路。（提示：它们看起

来像 8 字形。)

829 10.23 说明如果具有两个虚通道，可以使双向网络中的任意路由免死锁。

10.24 挑选本章中的任一个流控方案，计算流控符号所占用的带宽部分。

10.25 对于堆叠维度交换机，修正图 10-14 的时延估计值。

10.26 表 10-1 提供了几个重要的设计的拓扑结构和它们的基本通信性能特征的估计。试对每一种设计计算在 16 节点和 1 024 节点配置下，长度为 40 字节和 160 字节的消息的网络时延。时延中路由延迟和通道占用度各占多大比例？

830

第 11 章　时延的包容

从第 1 章我们看到，微处理器的速度每 10 年增长十倍以上，而常用存储器（DRAM）的访存时间却只缩短到原来的一半。因此，相对于处理器来说，访存时延在以每 10 年 5 倍的速度增长！多处理器系统可大大加速程序的执行，但在基于总线的系统中，其侦听却使得时延增加。在分布式存储系统中，访问一个节点的局部存储器的时间必须包括网络时延、网络接口和终点处理时间。高速缓存有助于减少长时延访存的次数，但它们也并不是万能的：它不能减少固有的通信，而且由于其他一些原因所导致的程序扑空率也相当明显。当节点增多时，相对于计算而言通信会增多，普通通信的网络转接增加，网络竞争也可能增加。

前几章所开发出来的若干协议的目标是减小长时延事件的频率和对通信介质带宽的要求，同时提供方便的编程模型。硬件设计的基本目标是，在减小数据访问时延的同时保持高度的可扩展带宽。通常可通过增加硬件改进带宽（如，用更宽的链路或更丰富灵活的拓扑），但时延是更基本的限制。

到目前为止，我们已经看到三种在多处理器系统中减少数据访问时延的方法——前两种的实施是系统的责任，第三种要由应用来负责。

- *减小扩展存储结构中每层的访问时间*。这要求十分注意具体的细节，以使得访问路径中的每一步都有很高的效率。处理器与缓存间的接口可以非常紧密，缓存控制器在扑空时需要快速反应以减小进入下一层的开销。网络接口可与节点紧密耦合并通过设计能够快速规格化、传送以及处理网络事务。网络本身也可以通过设计来减小路由时延、传送时间及拥塞时延。通过仔细设计，可以做到不过多地超过工艺的内在时延。不过，正是这些累计的开销才使得数据访问得以顺利进行。
- *重组系统以减小长时延访问的频率*。这是自动复制的基本工作，例如在缓存技术里利用程序访问的时间和空间的局部性使最重要的数据靠近其被使用的处理器。我们可以通过剪裁机器结构使得复制更为有效，例如每个节点保持一定数量的存储。
- *重组应用以减小长时延访问的频率*。其中包括分解并分配计算给某些处理器以减小内在的通信，重建访问模式以增加时间和空间的局部性。

除数据访问和通信外，还有其他潜在的长时延事件，如同步等，也可以用类似的办法来处理。这些为减小时延而在系统和应用方面的努力会有效果，但往往还不够。本章将讨论克服剩余时延的另一种方法：*包容剩余时延*，即通过与计算或者其他长时延事件的重叠来隐藏来自处理器关键路径上的时延。基本思路是：在长时延事件处理过程中，允许处理器做其他有用的工作，如数据访问及通信等。事实上，时延包容的关键是利用并行性，重叠的行为必须是相互无关的。其实，时延包容的思想很简单，在日常生活中经常用到；例如，当在等待一件事（如等待洗衣机洗完里面的衣服）的同时做其他事情（如，其他与洗衣机并行的事）。

时延包容技术与本书讨论的所有问题都有关，而且牵连到硬件和软件，因此时延包容在“一揽子考虑”中会发挥作用。我们将看到，时延包容技术的成功不仅依赖于应用的具体特

点，同时依赖于机器硬件机制的效率。

从单处理器中的多任务编程的角度看，我们对时延包容应该很熟悉。例如，在一次磁盘访问时，该事件的确是长时延的，但处理器并不停下来等待访问结束。相反，操作系统会阻塞磁盘访问的进程，而去执行其他进程（最为典型的是来自其他应用），这样就会用有用的工作重叠磁盘访问。正常情况下，被阻塞的进程以后会恢复，当然理想的是磁盘访问完成以后恢复。以上就是时延包容的概念：虽然磁盘访问本身一点也不能提早完成，但是其他潜在的资源（如处理器）并没有停止，而去履行其他任务。通过操作系统实现从一个进程到另一个进程的切换会额外执行许多指令，但我们认为这是值得的，因为磁盘访问时间足够长。在以上的多道编程例子中，我们并没有减小单个进程的执行时间（事实上，会有所增加）；然而却提高了系统的吞吐率和利用率。完成的工作总量越多，意味着单位时间内完成的进程越多。

本章讲的时延包容与前面的例子区别在于以下两方面：首先，主要集中于用同一应用中的工作重叠时延，即目标是利用时延包容减小给定应用的执行时间。第二，我们所包容的是存储和通信系统中的时延，而非磁盘。这些时延非常小以至于造成该时延的事件对操作系统是不可见的；因此不能通过费时的操作系统控制的应用程序或进程的切换来实现包容。

832

在进行进一步的讨论之前，我们先定义一些本章用到的术语。存储访问或通信时延包括从处理器发出到完成的各个部分的消耗时间。对于通信来说，时延包括处理器开销、通信辅助硬件的占用度、传输时延、与带宽有关的开销以及重用。单向通信和往返通信通常都可通过上下文来判断，其开销除数据传输外还包括像作废或确认等协议处理。同步时延包括从处理器发送同步操作（如加锁或屏障）开始直到经过该操作的一段时间，其中包括访问同步变量和事件等待时间。指令时延指的是从指令发射到在处理器流水线内完成的时间，其中假设没有内存、通信以及同步时延。除了部分长周期指令（如浮点除法）或流水线起泡外，大部分的指令时延都被处理器流水隐藏，但人们已研究了多种技术以隐藏前者带来的时延。本章的主要目的是讨论隐藏通信时延（包括显式和隐式），但是部分技术也适用于局部存储、同步和指令时延，因此也可用于单处理器。

由一个单个用户引发的从一个节点到另一个节点的通信，不管其大小如何，都称为一个消息。例如，在显式信息传递抽象结构中，当由一个缓存扑空触发的每个网络事物不能在共享的地址空间得到局部满足时（若在局部得到满足，其时延称为*局部存储时延*或*简单存储时延*；若在远程得到满足，其时延称为*通信时延*），一个 Send 命令则发送一个信息。最后，有关通信的一个重要方面是，谁是通信的发起者：数据的发送者（源）还是数据的接收者（目的）？如果通信操作是在没有接受者的请求时由数据的生产者或拥有者发起的，我们称之为*发送发起者*，例如在消息传递中的发送（send）操作。如果通信操作是由接收者进程引起的，我们称之为*接收发起者*，例如在共享存储中对非本地数据的读扑空。发送者发起的及接收者发起的通信将结合特定的编程模型详细讨论。

本章将按照以下顺序讨论时延包容技术。11.1 节将对存储及通信中的时延问题进行讨论，并引入时延包容的 4 种方法：成块数据转送、预通信、在同一线程中处理一个待完成的通信事件、多线程技术或用其他线程中的独立的工作来重叠。同时，本章还讨论适合任何时延包容技术的系统和应用要求，以及在实际系统中开发时延包容的潜在效益和基本的局限性。

833

本章的其他部分将检验以上 4 种方法如何在两个主要通信抽象中的应用。11.2 节讨论显式信息传递的方法，接着是有关共享存储空间问题。后者的讨论更为详细，因为对于通过单个的 load/store 进行的通信，与可变大小的传输相比，前者具有更大的时延瓶颈。另外，时延包容还与完成共享空间体系结构的支持工作显示出有趣的交互性，而且本节中的很多技术对于单处理器也是适应的。11.3 节对共享地址空间中的时延包容进行了综述，后续的 4 节分别集中讨论以下 4 个方法：实现要求、性能收益、多种技术的折中和协作、软硬件方面的应用。以上技术中的一个统一的要求是缓存应当是非阻塞的或免锁定的，于是，11.8 节讨论了免锁定的缓存的实现。

11.1　时延包容技术概述

为讨论包容通信时延的技术，我们先看一个生产者-消费者的例子，这个例子将在本章中经常引用。

一个进程 P_A 计算并写数组 A 的 n 个元素，另一个进程 P_B 读数组。两个进程在读/写数组的循环中各自做无关的计算，其中数组 A 存于 P_A 所在的处理器的内存中。若没有时延包容，则发起通信的进程只是简单地每次完成一个字（在每个通信抽象中显式或隐式地进行），而且只有等到一个字长信息完成后才能执行其他的任务。这种过程称为*基准通信结构*。图 11-1a 是显式消息传递中的计算，图 11-1b 是隐式的共享存储空间的读写过程。前者的特点是进程 P_A 通过发送操作产生实际的数据通信，而后者的特点是进程 P_B 通过读 A［i］来实现[⊖]。我们假定读操作使得处理器停止直到完成，而且使用同步发送（参照第 2 章 2.3.6 节）。通信发起进程的结果时线如图 11-2 所示。该图表明一个进程花费大部分的时间等待通信。 834

```
P_A                          P_B
for i←0 to n-1 do
  compute A[i];
  write A[i];
  send A[i] to P_B;
  compute f(C[i]);
    /*unrelated*/            for i←1 to n-1 do
end for                        receive (myA[i]);
                               use myA[i];
                               compute g(B[i]);
                                 /*unrelated*/
                             end for
```

a) 消息传递

```
P_A                          P_B
for i←0 to n-1 do
  compute A[i];
  write A[i];
  compute f(C[i]);
end for
flag ← 1;                    while flag = 0();
                               for i←1 to n-1 do
                                 read A[i];
                                 use A[i];
                                 compute g(B[i]);
                             end for
```

b) 共享地址空间

图 11-1　例子计算的伪码。a) 是显式消息传递的伪码，b) 是共享地址空间中隐式读写通信的伪码，两种情况都不带时延包容。加框代码指出产生数据传送的操作

11.1.1　时延包容与通信流水线

时延包容方法的好坏可通过机器利用率的高低得到很好的理解。站在处理器的立场上

⊖ 事实上，这些例子在它们的基础通信结构上并不完全是对称的。这是由于在消息传递版本中，数据的通信在每个数组项产生后都会发生；而在共享地址空间版本，通信的发生是在整个数组产生后。然而，要实现和共享地址空间完全类似的细粒度同步，就要求在每一数组项上的同步，所导致的通信会将我们的讨论大大复杂化。这种不对称性不影响时延包容的讨论。

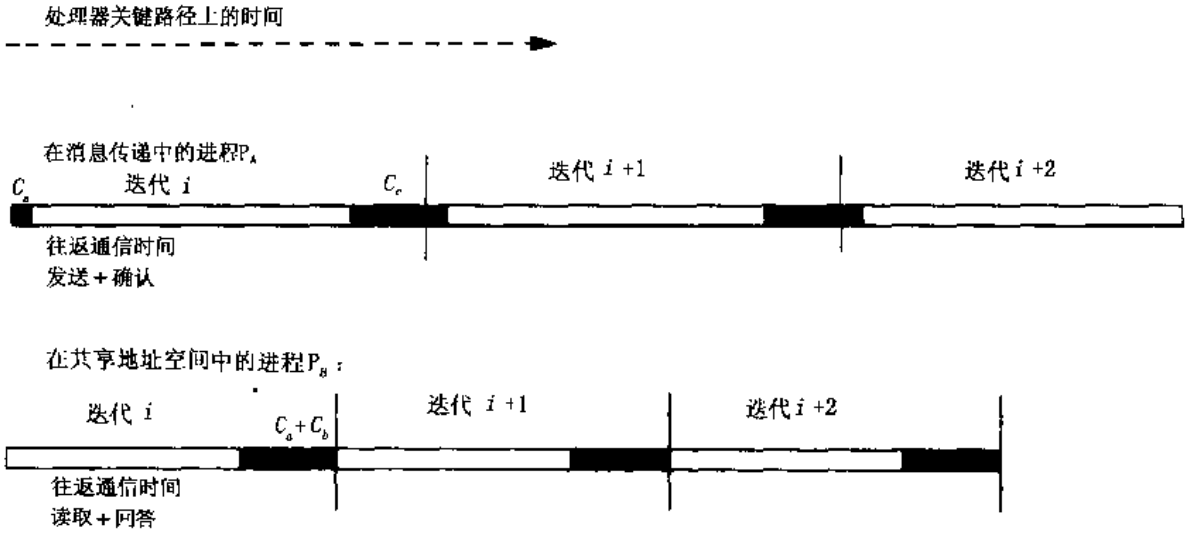


图 11-2 在没有时延包容情况下，发起通信的进程的时间表。时间表上的黑色段为本地处理时间（实际上包含访问本地数据等待的时间），白色段是在通信上等待的时间。C_a，C_b，C_c 分别是计算数组项 A [i]，完成不相关的计算 f（B [i]）和g（c [i]）的时间

看，从一个节点到另一个节点的通信可被视作一个流水线：其流水段包括源和目的的网络接口，以及沿途的网络链路和交换机（见图 11-3）。当然还可以包括通信辅助部件、局部存储/缓冲系统，甚至主处理器，这要依赖于通信的管理体系结构。值得注意的是，虽然通信的其他部分可被潜在地隐藏，但由处理器自身处理的指令终点开销（并不访问内存）是无法隐藏的。因此，当一个系统具有高的终点开销时，其很难实施时间时延包容。除非特殊说明，本章论述忽略处理器开销；我们假定大部分的消息终点处理由通信辅助部件完成，而且主要集中于终点的辅助处理的占用时间，因为这部分时间有可能被隐藏。另外还假定，每个消息一次发起和接收的开销是固定的（即，并不与消息的大小成正比）。

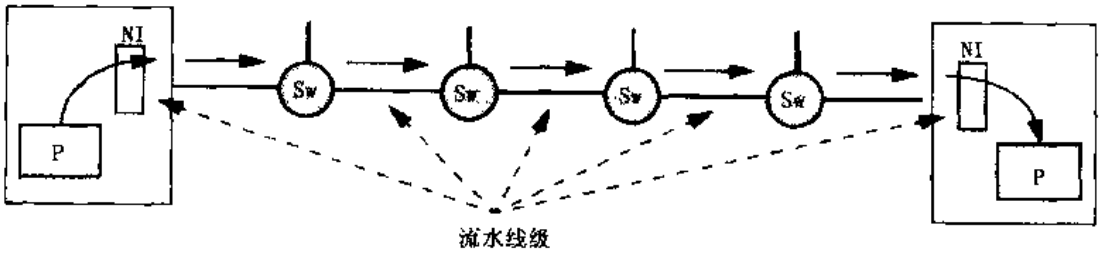


图 11-3 将网络看成是一个从一个处理器到另一个处理器的通信流水线。流水线的阶段包括网络接口（NI），在两个处理器（P）之间链路上相继交换机（Sw）之间的跳步

如果不采取措施将时延包容起来，可能会导致流水线和其他资源的利用率降低，图 11-4 显示了前面描述的基准通信结构中的利用率情况：对于给定时刻，或者处理器忙，或者通信结构忙；对于后者，一次只有一个通信结构的流水段忙[⊖]。时延包容的目的就是尽可能重叠使用这些资源。从处理器一端看，可开发的重叠有三种类型。第一种是两节点间的通信流水线（communication pipeline）内：正像指令流水线重叠使用处理器的不同资源（取指单元，寄

⊖ 为简单起见，我们忽略这样一个事实：网络链路的宽度通常是小于一个存储字的，从而一个字可能要占据流水线中网络部分的多个阶段。

存器堆，执行单元等）一样，我们可利用网络资源一次传输多个字节。这些字可以来自同一个消息（超过一个字的信息），也可以来自不同的消息。第二种是开发不同点对点通信流水线之间以及网络不同部分之间的重叠，这种方法一次允许多个节点的同时进行待完成的通信。从处理器的角度看，这两种情况都是不同字通信之间的重叠，因此称之为通信和通信的重叠。第三种是计算与通信间的重叠。即，当一个处理器操作完成发起通信后继续执行有用的局部工作。

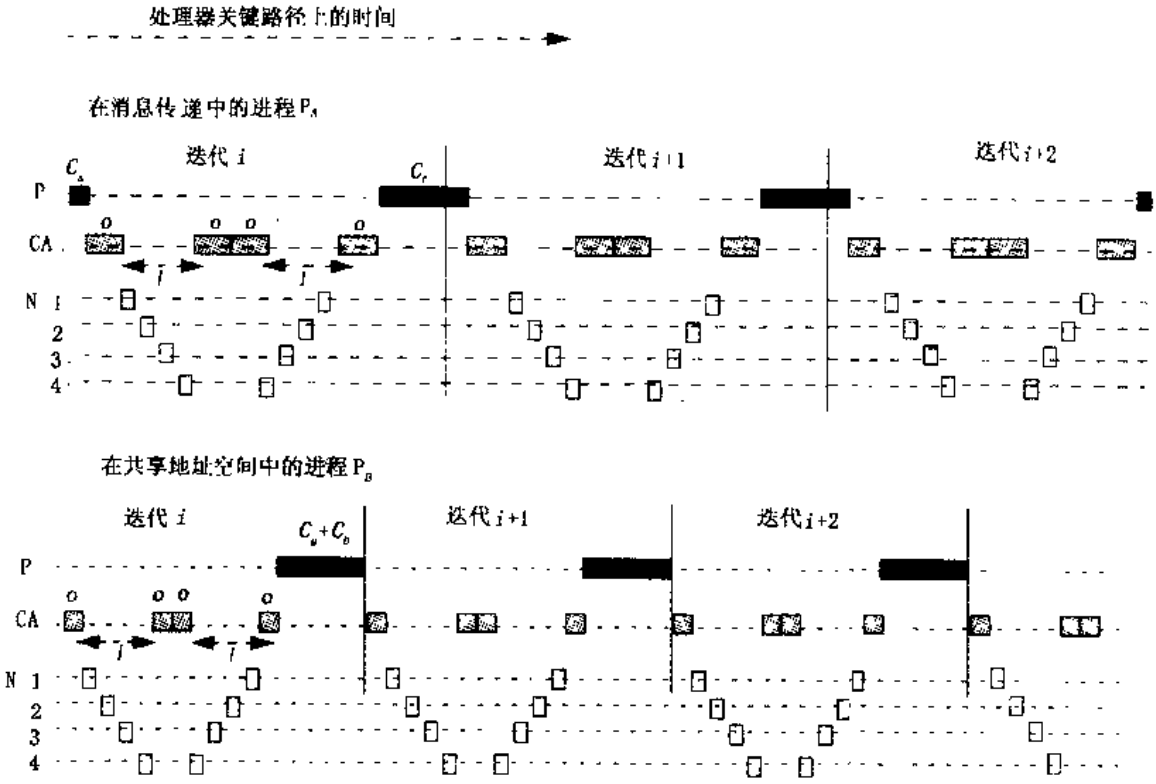


图 11-4　无时延隐藏情况下，消息传递和共享地址空间程序的时间特性。一个进程的时间被分为几个部分，花在处理器（包括本地存储系统）上的（P），在通信辅助部件（CA）上的以及在网络（N）上的。在网络部分的数字表示在消息通路上不同的跳步或连接。在消息传递情形下端点开销 o 要比共享地址空间情形要大，我们假设后者是由硬件支持的。l 是网络时延，是不涉及到处理节点所消耗的时间

11.1.2　采用技术

要想开发硬件资源间的重叠以达到时延包容，有 4 种关键途径。第一种称为块数据传送，主要是增大每次通信的消息，这样可通过网络实现流水。其他三种是使一个消息与计算或与其他消息重叠，因而是块数据传送的补充。这三种是：预通信、跨越同一线程中的通信和多线程。每种方法都可以应用于共享地址空间和消息传递抽象中，尽管两者要求的特殊技术和支持是不同的。以下对两者进行具体介绍。

835
837

1．块数据传送

增大消息长度可带来多种好处。首先，它可开发两个节点间的通信流水线，用通信重叠通信。处理器只能看到消息第一个字的时延，后续字大约每个网络周期到达一次，其速度只受流水线频率和带宽的限制。第二，这种方法将每个消息的终点开销分摊到发送的大量数据

中。第三，依赖于数据包的结构，该方法也可分摊消息的路由和头信息。最后，一个大的消息可只需要一个确认而不是每个字一个确认，这不仅可以减小时延而且可以减小传输次数和冲突。这些优点类似于在单处理器存储系统中增大缓存块所得到收益，只是规模不同而已。图 11-5 显示了在显式消息传递下使用单个大消息的效果，其中仍然使用同步的消息且无计算重叠。为了说明方便，网络流水采用一个阶段。

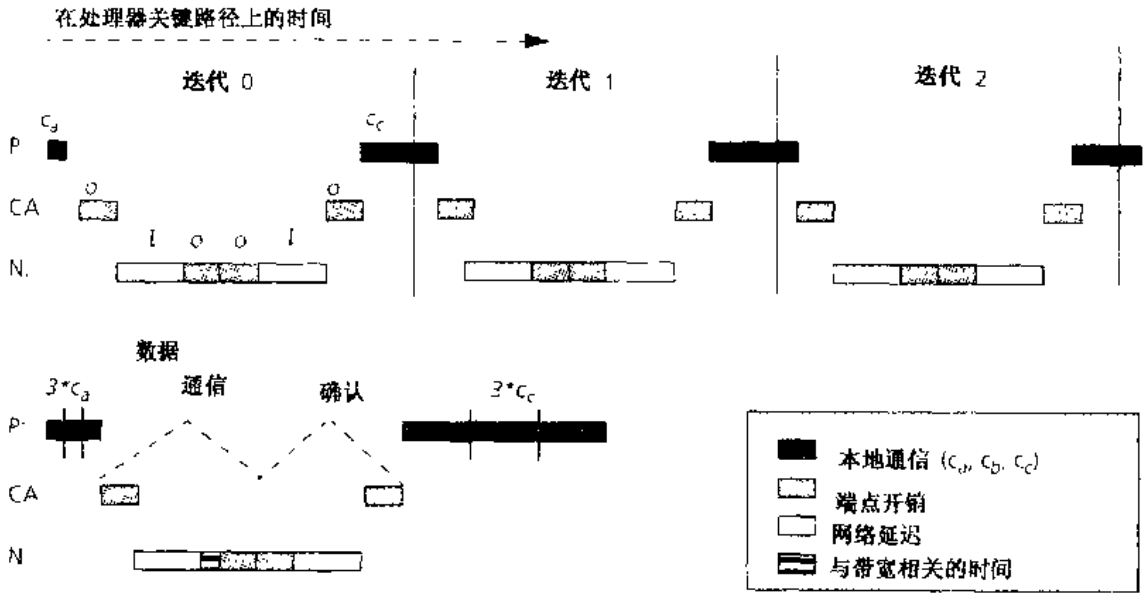


图 11-5 对于消息传递例子程序，在时间上将消息放大的效果。图中假设执行了三次循环迭代。发送进程（P_1）首先计算要送出的三个值 A[0..2]（计算 C_a），然后将它们用一个消息送出，然后完成所有三个无关的计算 f（c[0..2]），即计算 C_c。数据通信的开销被均摊了，并且只需要一个确认。由于传送的数据量增加了，和带宽相关的通信代价会有些增加，但和节省下来的时延和开销相比，这是微不足道的。对于确认，不需要额外的带宽开销，这个开销也是很小的

虽然增大消息有助于保持两点之间流水线的繁忙，但当一个消息在进程中时，它本身并不能保持处理器和通信辅助部分的繁忙，也无助于保持通过网络的其他路径的繁忙。其他的方法也可做到。这些都是块数据传送的补充，无论消息是长还是短，都是适用的。以下一一介绍。

2. 预通信

程序中在通信操作出现之前产生通信，可使在真正需要数据时已经部分或全部完成通信；以上目标可通过软件插入代码方法实现，也可利用硬件检测时机并较早发出通信操作[○]。实际使用数据的典型操作仍然保留在程序中的原位置上。当然，预通信事务本身不应当使处理器停下来等它完成，否则不会实现重叠。预通信的很多形式都需要长时延事件能够被预知，以便硬件或软件可预测并提前发送。

838

发送者发起通信是当发送者产生数据不久自然产生的，这样可使得数据在接收者实际使用之前到达，这就形成了一种对于接收者无开销的预通信形式。另一方面，对于发送者来说，为了隐藏发送者所看到的时延所做的任何通信的提前可能都是很困难的。实际中的预通

○ 尽管这里讲的是通信，但相应的技术，包括这一种，也能够用来隐藏本地存储访问时延。这是因为我们可以将本地的访问看成是和本地存储系统的通信。

信（提前产生通信操作）通常发生在接收者发起的情况下，此时的通信是当数据需要时自然发起的，其数据可能已经产生很长时间。

3. 进行同一线程中已传递的通信

通信操作可在程序中发生时自然产生，但是处理器可允许继续进行通过该通信点，并在同一进程或线程中寻找其他独立的后续计算或通信。这样，预通信使得通信与同一线程中位于源程序中产生通信点之前的其他指令相重叠，该技术可使得通信与同一线程中后面的指令重叠。这些指令本身可产生通信和存储访问，或也与这些行为发生重叠。正像我们的想象那样，由于在发送方紧跟在通信操作之后的指令并不依赖于该通信操作，并可易于重叠；因此，这种时延包容方法通常容易在发送者发起通信中得到有效的使用。在接收者发起的通信中，一个发送者在需要数据前自然地访问数据，因此在通信和使用之间无法找到足够的无关工作。当然，通过在指令流中后推来推迟数据的使用，也可等价于在使用方找到无关的工作，而且编译和处理器硬件可从中开发重叠。两种情况中，软件或硬件必须在依赖通信的指令执行前检查数据传输的完成。

839

4. 多线程

这种方法基本与前者相仿，只是其无关工作是通过切换到映射于同一处理器上的其他线程来得到的。这样就使得接收者发起的通信时延比前者更容易隐藏，无论发起方还是接受方都易于实现。事实上，由于这里的重叠是通过其他线程得到的，很少关心所时延包容的类型和结构，从该意义上讲多线程是最通用的技术。然而，多线程技术意味着在遇到长时延事件时，在同一处理器中需有多个可同时执行的线程以供切换。这些线程可来自于同一并行的程序，也可来自于完全不同的任务（如同前面的多程序的例子）。多线程程序与一般的并行程序并没有什么两样，只是将程序分解并分配给 P 个进程，其中 P 比实际的物理处理器数 p 大，P/p 映射到同一个物理处理器。因此，多线程要求时延包容需要的附加并行性以线程的形式显式出现。

11.1.3 基本要求、优点与局限性

在这些技术应用于特殊的程序模型和系统之前，懂得时延包容的基本要求、优点和局限性是非常有用的。这些基本的分析确定了我们指望性能改进的边界，其只是基于资源使用的重叠和这些资源的占用。

840

1. 要求

时延包容需要额外并行性、带宽，在多数情况下还需要更高级的硬件和协议。

- 额外并行性，或松弛性。由于重叠的行为（计算与通信）必须是相互独立的，应用程序中的并行性一定大于所使用的处理器数。与多线程技术一样，额外的并行性可显示为多个线程或存于一个线程内部。甚至在并行情况下，同一进程发出的两个字的通信也蕴涵着它们之间不是串行的关系，因此有额外并行性。
- 增强的带宽。尽管时延包容可减少执行时间，但并不能减少通信完成的数量。以更少的时间完成通信意味着单位时间内的更高通信速度，同时也意味着对通信结构的更高的带宽要求。事实上，如果要求的带宽高于机器提供的带宽时，其带来的资源争用会减慢其他的无关事务，这样时延包容可能会有损而不是有助于性能。
- 更复杂的硬件和协议。除了增大消息的情况外，处理器必须允许跨越长时延操作

（在操作完成之前）而继续进行；而且要想实现通信与通信之间的重叠，而不仅是与计算的重叠，处理器必须允许多个待完成的长时延操作。

这些要求预示着所有时延包容技术都有重要的开销。因此，我们应当在使用时延包容技术之前，首先使用算法技术减小长时延事件出现的频率。长时延事件越少，时延包容所需要的技术激进性就越小。

2. 潜在的优点

我们通过简单的分析就能得到所期待的时延包容的收益限度，因而这也就确定了实际中的期望值。我们集中讨论容许通信时延，并假定局部的存储访问时延不被隐藏。假设当没有时延包容时，处理器看到的执行时间剖视如下：T_c 个周期用于局部计算，T_{ov} 周期用于处理器的处理消息开销，T_{occ}（占用度）个周期用于通信辅助，T_l 个周期用于等待网络消息传输。如果假设处理器的其他资源可很好的重叠使用，于是潜在的加速比可以简单利用 Amdahl 定律来确定。处理器须占用 $T_c + T_{ov}$ 个周期；处理器可隐藏的最大时延为 $T_l + T_{occ}$ 周期，于是时延包容的加速比为：

[841]

$$\frac{T_c + T_{ov} + T_{occ} + T_l}{T_c + T_{ov}}$$

或

$$\left(1 + \frac{T_{occ} + T_l}{T_c + T_{ov}}\right)$$

由于假设理想的资源重叠，而且没有额外的时延包容开销，该极限即为一个上限。然而，它却给我们以有用的期望。例如，如果进程原来花费在通信系统上停滞的时间至少和局部计算时间或处理器开销时间一样多，则从包容通信时延得到的加速比最大为 2。如果原来的通信停顿时间只与处理器的行为重叠，而不与其他的通信重叠，其最大加速比为 2，不管有多少通信时延被隐藏。如图 11-6 所示。

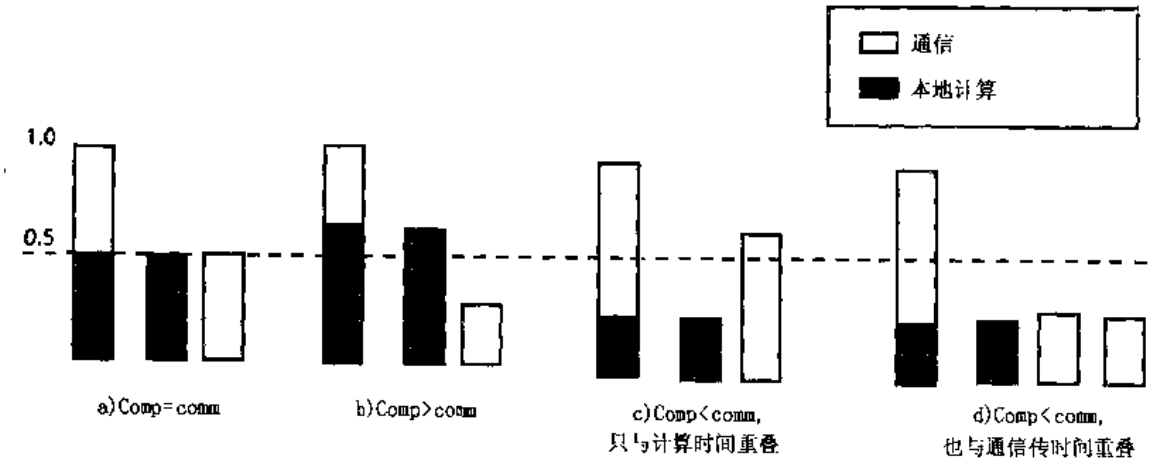


图 11-6　时延包容最多能得到多大的好处。每个图表示一个不同的情形。灰色表示计算时间，白色表示通信时间。在每一组中，左边的条表示没有时延隐藏的时间，规格化到 1.0 个单元，右边的条表示延时隐藏的情形。在 a）中，计算时间（Comp）等于通信时间（Comm），加速比上限是 2。在 b）中，计算时间超过通信时间时，受过多计算时间的影响，上限小于 2。c）的情况和 b）类似，通信时间超过计算时间，且只能由计算来重叠。要获得比 2 大的加速比，就要有 d）的情形，即通信时间总体上比计算时间长，但通信本身还可以重叠

多少时延在实际中被隐藏依赖于包括应用和体系结构在内的很多因素。与应用相关的特征包括通信结构以及多少其他的工作与通信重叠。体系结构问题包括：与辅助通信相比有多少终点处理由处理器来做；通信是否能与计算或其他通信重叠；某个处理器一次能同时处理多少个待完成的消息；终点开销处理与网络中消息的数据传送能重叠到什么程度；辅助通信与网络流水段的占用资源如何。

图 11-7 显示了几种不同类型的消息结构和重叠的时间效果，此图只是说明性的。例如，其中假设每个消息的开销与传输时延相比非常大，而且任何时候都不考虑争用。在这些假设下，较大的消息比很多小的重叠的消息更具有吸引力，因为大消息更能分摊开销。而且，对于小消息来说，流水线的频率受每个消息终点处理（其决定消息的间隔）的限制，而不是网络链路的速度。其他的假设也会导致不同的结果。习题 11.2 更加定量地分析了一个通信只与其他通信重叠的例子。

显然，某些通信时延比其他的部分更容易隐藏。例如，一个处理器的指令时延不可被该处理器隐藏，而节点外的时延（或者在网络上，或者在与之通信的其他节点上）比通信辅助或节点内的其他开销更容易被其他消息重叠隐藏。以下我们讨论影响达到时延包容上限的关键局限性。

3. 局限性

其主要的局限性可分为三类：应用局限性、通信体系结构局限性、处理器局限性。

应用局限性　可用来和时延重叠的无关计算的时间大小可能受到限制，因此并非所有的时延都能被隐藏，处理器仍有一部分时间处于停滞。即使程序中存在足够的工作和并行性，但程序的结构可能使得系统或程序员很难识别出这些并发操作，以至于难以协调重叠。这一点将在讨论特殊时延包容机制时看到。

通信体系结构局限性　通信体系结构可能使同一节点一次发出的待完成的消息或字的数量受到限制，而且通信体系结构的性能参数也可能限制隐藏的时延大小（Culler 1994）。

当只有一个消息处于通信时，独立的计算可被辅助处理和网络传输重叠。然而，对于同一消息来说，辅助处理可能被网络传输重叠，也可能不被重叠；网络流水线本身只有当消息较大时才能保持繁忙。当多个消息可同时通信时，处理器时间、辅助占用度、网络传输都可被重叠。由于多个消息可并发进行，即使消息很小，网络流水线也可得到很高的利用率。因此，对于一定的消息大小，当一个节点允许多个通信同时进行时，可以容许更多的时延。若 L 为可被隐藏的每个消息时延，r 为和每个消息重叠的有效独立计算周期，要得到最大的时延隐藏需要 $\lceil L/r \rceil$ 个待完成的消息同时进行。当能被隐藏的时延已被隐藏时，增多待完成的消息数则无济于事，同样，一个节点同时处理的字数量也是很重要的。若一个处理器一次可同时处理 k 个字（来自一个或多个消息），则处理器所看到的时延将减小 k 倍。更精确地讲，对于具有同一目的地的单向消息来说，其时延将从 $k * l$ 周期减小到 $l + k/B$ 周期，其中 l 为一个比特所需的网络传输时间，B 是以字/周期为单位的通信流水线的带宽或速度（网络也可能是辅助部件）。

842
843

假定允许足够的未完成消息或字，通信体系结构的性能参数也会成为限制条件。以下是对某些参数的检验，假定消息的大小预定，要隐藏的通信的每个消息的时延为 L 周期。为简化，我们只考虑单向的无确认的数据消息。

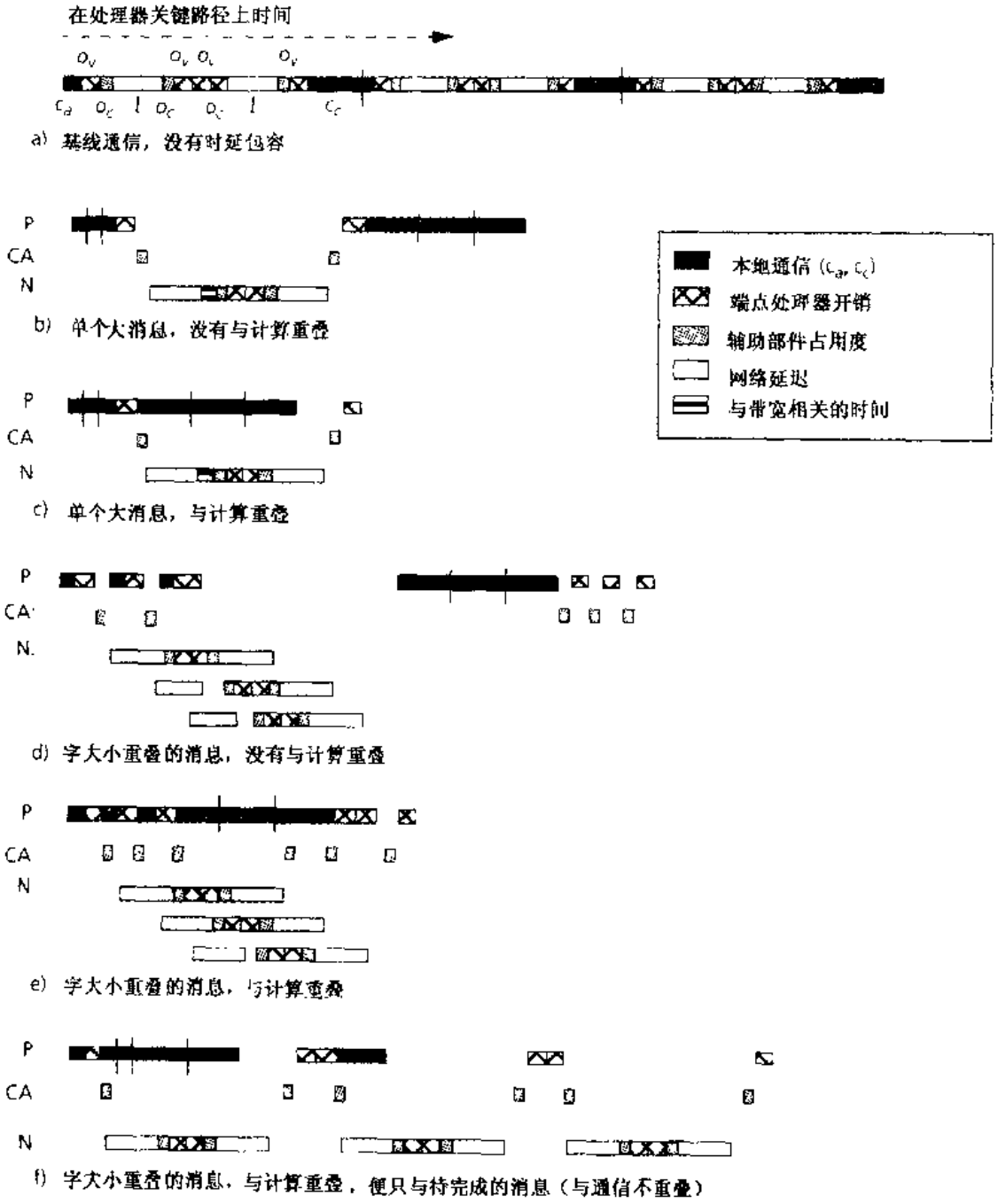


图 11-7 不同形式的时延包容的时间表。P 指出花在主处理器上的时间，CA 是在本地通信辅助部件上的时间，N 是非本地时间（网络中或者是在另外的节点上）。所表示的时间表是从图 11-1 消息传递例子中进程 P_1 的角度所看到的

- 开销。发生在处理器上的消息处理开销是不能被隐藏的。
- 辅助部件占用度。辅助部件占用度可被计算重叠。辅助部件占用度是否能被其他通信（在同一消息中或在不同消息中）重叠依赖于其结构是否为内部流水，以及一个消息是否能被数据在网络中的传输重叠。

一个 o 周期的端点消息处理时间（开销或占用度）是我们能够向网络发送消息的频

率上限（至多每 o 周期发送一次）。若假定处理器平均的接受信息与发送信息相同，而且接受和发送的时间也一样，则每个消息发送的端点处理时间为 $2o$ 周期；因此，一个处理器在 L 周期内允许为完成消息的最大数量为 $L/2o$。如果 $2o$ 大于可重叠消息的 r 周期的计算量，那么，将无法允许 L/r 个未完成消息，进而无法隐藏全部的时延。尤其当消息很小时，每个消息的开销和占用度对性能的影响更加严重。

- 点到点的带宽。即使开销不是问题（例如，当消息很大时），向网络发送消息的速率也会受从源到目的端中整个流水线中最慢链接的限制，即网络阶段本身、节点到网络的接口，每个字的辅助占用度和处理器开销。正像 o 周期的终点开销将 L 周期内的消息执行数限制在 L/o 一样，网络中的每个字流水段时间 s 也会使得网络的未完成字为 L/s。

844
~
845

- 网络容量。一个处理器能同时处理的消息数是受网络的总带宽和数据运输容量限制的，不同的处理器也以该有限容量作为竞争的指标。如果每个链接带宽都是一个字节，在给定时间内一个 M 字的消息若在网络中进行 h 转发，那么就相当于要求最多 $M*h$ 个链路。若每个处理器可同时进行 k 个这样的消息处理，则每个处理器就要求在给定时间内具有 $M*h*k$ 个链路。然而，若网络中总共具有 D 个链路，每个处理器都以这样的方式传输，则处理器在给定时间内平均只占用 D/p 个链路。因此，每个处理器 k 的并行消息数受下式限制：

$$M \times h \times k < D/p，或\ k < \frac{D}{p \times M \times h}$$

处理器局限性 在具有缓存一致性的共享存储空间中，基于长缓存块的空间局部性已经允许一次未完成的通信超过一个字。要想隐藏更多的时延，一个处理器必须能够允许多个缓存扑空同时发生。我们将看到其开销是很大的，因此处理器及高速缓存对同时发生的扑空次数具有相对小的限制。虽然显式通信可能限制系统中一次通信的数量，但它要求较少的硬件跟踪且在该方面具有更多的灵活性。

表 11-1 单字消息微基准测试程序中一次允许未完成的消息数受 L/o 比率与消息发送到网络的速率约束情况

机器	单项网络事务		远程序		机器	单项网络事务		远程序	
	$L/2o$	Msgs/ms	L/o	Msgs/ms		$L/2o$	Msgs/ms	L/o	Msgs/ms
TMC CM-5	2.12	250	1.75	161	NOW-Ultra	1.79	172	1.77	127
Intel Paragon	2.72	131	5.63	133	CRAY T3D	1.00	345	1.13	2 500
Meiko CS-2	3.28	74	8.35	63.3	SGI Origin				5 000

注：其中 L 表示包括开销在内的消息总时延，并考虑以下两种操作类型：一类是共享地址空间中单字往返远程读操作，另一类是包括发送和接收操作在内的单项网络事务。对于远程读，o 是启动处理器时的开销，这部分是不能被隐藏的。对于单项的消息传递网络事务，我们假定消息的传送是对称的，因此发送和接收的开销总和为 $2o$。对于全部用硬件支持共享地址空间的机器，这里的间隙，即消息注入的速率的制约因素不是处理器的开销，而是辅助部件的占用度，通常是较小的。在这些情况下，主要的限制是处理器或辅助部件允许的待完成扑空的个数，但在表中我们没有考虑。

从以上的讨论中我们清楚地看到，有效的通信结构（高带宽、低的端点开销或占用度）对于时延包容的效率是十分重要的。前几章中我们已经根据以下特点分析了一些真正机器的

特性：如网络带宽、节点到网络的带宽、终点开销。从数据开销、通信时延、消息间隙，我们可以计算出对于理解时延隐藏很有用的两个数据。第一个是 L 周期内允许一次待完成的消息数 L/o（假定与此同时其他处理器不向该处理器发送数据），该值显示了通信性能的局限性。第二个是消息间隙的倒数，表示消息流水进入网络的速率，其受 o 的影响。如果没有足够的计算来重叠 L/o 个消息通信时延，隐藏其他时延的惟一方法是增大消息大小。从图 7-31、图 7-32 和第 8 章中从 Origin2000 得到的微基准测试程序数据（表 11-1），可以计算出共享地址空间中的远程读和对使用显式消息传递库的显式地消息传递读的次数，L/o。

对于消息传递系统（CM-5、Paragon、CS-2、NOW-Ultra），很清楚，它们一次允许进行的消息数量很少；其小消息的时延隐藏显然受到端点处理开销的限制。因此增大消息的大小，会比重叠多个小消息时延更有效。对于硬件支持的共享存储空间机器（T3d 和 Origon），其隐藏小消息时延能力则很少受通信体系结构性能参数的影响。其主要的限制是处理器或缓存系统支持的待完成请求的数量，即，一个给定的存储操作可以产生多种协议事务（消息），这样做加重了辅助部件的占用度。

懂得这些基础，我们就可以讨论两种主要通信抽象中的各种技术。由于显式消息传递通信中的时延包容都是很简单的，下一节依据同一背景对 4 种常用技术进行讨论。

11.2 显式消息传递中的时延包容

为了搞清楚哪种时延包容最有效以及如何应用，我们将通信结构进一步抽象；尤其是要分析清楚发送者发起与接收者发起通信的分布，固定与可变消息的使用特点。

846 ~ 847

11.2.1 通信结构

在显式消息传递中的数据传输属于典型的发送者发起通信；接收操作本身并不能引起穿过网络的通信，而只是将活动缓冲区中的数据拷贝到应用地址空间。接收者发起通信是首先向数据的源进程发送请求消息，然后数据节点发回数据[○]。图 11-1 所示的例子中的基准通信结构是采用同步发送和接受的发送者发起的通信。一个同步发送操作的通信时延时间包括：将所有数据传输到目的节点的时间，接收处理时间，以及确认时间。一个接收操作的时延是指它的处理开销，包括把数据拷贝到应用区，当数据还没到时还应包括等待时间。我们应在两端隐藏这些时延。为了更好地隐藏开销，我们进一步假定每个端点开销发生在通信辅助部件上而不是在主处理器上，以此分析以上 4 类时延包容技术是如何运用到典型的发送者发起的消息传递通信中的。

11.2.2 块数据传送

增大消息的大小对于分摊开销以及保证通信流水线的速率不受端点消息处理开销的限制是很重要的。这些优点甚至在同步通信中也可以获得。然而，如想利用计算或其他消息来重叠通信，我们必须使用其他时延包容方法。虽然这些方法可用于大（包括块传送）、小消息，而且作为块传送的补充，我们仍然用基准通信结构中使用的小消息来说明。

○ 在其他一些建立在消息传递机器上的通信抽象中（如在第 7 章所讨论的），例如远程过程调用或主动消息，接收方送出一个请求消息和一个处理这个请求的程序柄，数据源将数据送回而不需要涉及那里的进程。不过，我们这里将主要讨论经典的消息传递，它在程序设计模型层次具有支配的地位。

11.2.3 预通信

图 11-8 显示了图 11-1 中消息传递程序的预通信方案。发送方 P_A 的循环分成两部分。所有的发送都移到计算函数 f（B[]）之前，这些计算就推迟到一个单独的循环中。这样，发送成为异步操作，进程不必等到这些操作完成便可继续进行。

P_A	P_B
for i←0 *to* n-1 *do*	a_receive (myA[0] from P_A);
compute A[i];	*for* i←0 *to* n-2 *do*
write A[i];	a_receive (myA[i+1] from P_A);
a_send (A[i] to proc.P_B);	while (!recv_probe(myA[i]) { };
end for	use myA[i];
for i←0 *to* n-1 *do*	compute g(B[i]);
compute f(C[i]);	*end for*
	while (!received(myA[n-1]) {};
	use myA[n-1];
	compute g(B[n-1])

图 11-8 在消息传递例子中通过预通信来隐藏时延。在这个伪码中，[a_send] 和 [a_receive] 分别是异步发送和接收操作

这样做是否是个好主意呢？其优点是消息尽可能早地发给接收者；缺点是在发送方消息之间很少的工作被重叠，但这样会加重接收方缓冲区的压力，因为消息会在需要之前到来。其网络运行的结果是否得到收益依赖于每个消息的开销、处理器允许一次进行的消息数、接收晚于发送的时间。例如，若只有一两个消息是待完成的，那么我们可以较好地在异步发送之间分散计算 f（B[]）的，例如建立计算和通信的软件流水线就可避免等待这两个消息完成而停止工作。当通信辅助的开销较大时，也应采取同样措施以便当每个消息具有较大的辅助开销时使得处理器繁忙。理想情况是建立一个平衡软件流水线将发送操作上拉，以便在消息发送的间隙做适当的计算。

现在考虑图 11-8 中的接收方 P_B。要想通过预通信隐藏接收方的时延，将这些操作在代码中上移，并使用异步接收。a_receive 调用简单地将接收描述送到消息层，然后处理器继续进行。当数据到来时，辅助通信得到通知将数据移到应用数据结构中，然后透明地传送给处理器。在安全使用这些数据之前，应用必须检查数据是否真的到来（使用 recv_prob 调用）。当发 a_receive 调用（预发送）时，若消息已经到达，则我们所希望通过预发送隐藏的是通信辅助的开销；否则应该隐藏传输时延和接收处理开销。

正像第 7 章所讨论的，接收开销通常比发送开销大。因此，当发生（或预发生）许多异步接收的时候，如果将它们之间插进一些计算，而不是一个紧接着一个，通常效果要好得多。否则，通信辅助部件跟不上处理器发出它们的速度，从而一旦它们之间的缓冲已满，处理器就会停滞。解决的方法之一是像图 11-8 那样建立一个计算和通信的软件流水，每一次迭代发送一个接收调用 a_receive 以准备下一次迭代的数据，然后进行当前迭代中的工作。希望的情况是，当下一个迭代来到时，当前迭代发出的接收的消息应已到达，而且已处理完毕，数据准备好。如果接收开销大于每次迭代的计算时间，a_receive 调用可提前多个迭

848
849

代发出而不只是提前一个迭代。

软件流水线由三个部分组成。除包括刚才提到的稳定状态循环部分外，为了启动和结束流水线还要做一些其他的工作。该例子中，为了执行稳定循环中的第一次迭代，开始时需发一个接收数据调用，结束时应处理好原循环中的最后一次迭代（该部分工作不属于稳定循环，因为我们不想对一个不存在的下次迭代发异步接收）。对于接收者发起的情况，也可以建立类似的软件流水线。例如，若向执行 P_4 的节点发送 a_receive 或 get 操作请求时，该节点的事件管理者会对该请求做出反应，并不需要程序中的显式发送操作就可提供所需数据。由于接收或获取操作通过网络真正导致了取数据，预通信策略又称为数据预取。鉴于实际的接收发起者通信较多发生在共享地址空间中，因此有关预取的更详细的情况将在 11.6 节讨论。

11.2.4 跨越同一线程中的通信

现在假定并不想在代码中前提通信操作，即并不改变这些操作在代码中的位置。该情况下的隐藏时延的一个简单方法是，将通信消息异步化，该线程可越过通信点继续进行其他的计算或异步通信消息操作。我们可以按照以上的方法继续运行，直到遇到依赖通信消息完成的操作，或者到了处理器允许的同时进行的未完成消息的极限数。当然，在使用预取策略时，异步接收操作意味着我们必须增加探测或同步指令以保证在使用消息时数据是有效的（而且在发送端，在重新使用消息所在的存储区时，其源数据已经被拷贝或已发出）。

11.2.5 多线程技术

多线程技术中，当一个进程发出一个发送或接收操作后，它可以将自己挂起，并允许应用中的其他就绪的进程或线程执行。当一个线程发出接收或发送操作后，它也会挂起自己，而其他线程会切换进来。我们希望当切换进来的线程执行后，第一个线程在被调度时其发出的通信操作会执行完。线程的切换和管理可由消息传递库来负责，而不是由应用程序。其任务是通过操作系统调用改变程序指针，执行其他保护的线程管理功能。例如，发送原语在实现时可使得发送初始化后自动引发线程切换（当然，在处理节点上若没有其他的就绪线程，则切换的还是原线程）。多线程技术甚至还可用于同步消息传递编程模型。该技术是 Transputer 上 Occam 语言的基础。

850

切换一个线程需要保存重新启动所必需的处理器状态，包括处理器寄存器、程序指针、堆栈指针、各种处理器状态字。当线程再切换回来时，其保存的状态应能正确恢复。用软件实现状态的保存和恢复是非常昂贵的，会严重降低多线程所带来的好处。一些消息传递体系结构已为多线程技术提供硬件支持，例如在硬件上提供多组寄存器与程序指针。值得注意的是，运行在同一主处理器上与应用程序独立的支持异步消息的处理程序的系统中，应用程序与处理程序间的多线程技术是非常重要的，即使应用程序本身并没有用多线程技术。这种形式的多线程技术在某些研究性的体系结构中已被硬件支持（例如，消息驱动处理器，J-machine [Dally et al. 1992; Noakes, Wallach, and Dally 1993]）。我们在 11.7 节分析共享地址空间中的多线程技术时，将详细讨论硬件支持问题。

11.3 共享地址空间中的时延包容

本章其余部分集中讨论基于共享地址空间硬件支持的时延包容。该问题的讨论要比消息

传递详细，这是由于：第一，已存在对通信的硬件支持使得其技术与体系结构和软硬件接口更为密切。第二，长时延事件的隐式特性（如通信）使得时延包容被系统解决的可能性比用户程序的大，第三，从通信的粒度以及基本通信机制的效率来看，时延包容的有效性依赖于硬件技术的支持。总之，由于很多时延来源于读、写或者指令（但不像 send，receive 等显式通信指令），因此大多数技术可适用于单处理器。事实上，由于我们并不能提前知道哪些读写操作导致通信，时延隐藏的处理方式更像多处理器共享地址空间中通信的局部访问。不同点是时延大小以及与缓存一致性协议的交互（在缓存一致性系统中）。

关于共享地址空间时延包容的多数讨论都适用于缓存一致性系统，也适用于那些没有缓存共享数据的系统。多数情况下，假设共享地址空间（以及缓存一致性）是硬件支持的，而且其通信和连贯的默认粒度为单个的字或缓存块大小。后面章节中实验结果来自于参考文献，这些结果使用了前几章中的几个应用程序，但它们却是不同的版本，具有不同的通信计算比等其他的行为特点，而且不总是按照第 4 章描述的方法。在各实验中，所用的系统参数也随不同的讨论而变，因此其结果不能用来比较不同的技术，而只是为了方便解释。像消息传递一样，我们首先简要介绍本节用的抽象通信结构。 851

通信结构

共享地址空间的基准通信是通过读和写实现的，方便地称为读写通信。典型的接收者发起的通信是通过存储操作实现的，该存储操作会导致另一个处理器中存储器或缓存的数据被访问。因此，它是单处理器编程模型中存储访问的自然扩展：当需要时就访问存储字。

如果共享数据没有缓存，发送者发起的通信可以通过写远程内存的数据得以实现[⊖]。对于缓存一致性的系统，其写的效果更为复杂。写操作导致发送者发起的通信还是接收者发起的通信依赖于缓存一致性协议。例如，假设处理器 P_A 向分配在处理器 P_B 的内存写一个字。在基于无效的缓存一致性协议，当采用回写策略时，读操作只是产生独占的读或更新请求以及可能的无效操作，同时数据会传送给自己。事实上，这种协议并不会将最近的数据传到 P_B。然而请求和无效操作也包括网络事务，隐藏其中的时延也很重要，以后 P_B 读写数据时就会导致新值从 P_A 传送到 P_B 的实际通信。这种情况实际上是接收者发起的通信。或许，异步通信也会使 P_A 缓存中的数据传回到 P_B 内存并替换原来的数据。另一方面，在有的更新协议中，如果 P_B 的缓存保存着数据的拷贝，则写操作本身会使数据从 P_A 传送到 P_B 进行通信。

无论是发送者发起的通信还是接收者发起的通信，在共享地址空间中通过硬件支持的通信自然就是细粒度的，因此也使得时延包容特别重要。不同的时延包容方法适合于不同的时延类型与量级，而且得到不同商业产品的接受。下面的章节将详细地讨论这些方法。 852

11.4　共享地址空间中的数据成块传送

在共享地址空间中，显式地通过用户程序或者透明地通过系统都可以实现将数据合并成大数据块并启动块传输。例如，现代计算机中流行使用的长缓存块是对缓存块大小的消息透明块传送的一种方法。放松存储一致性模型还可进一步允许我们对字或缓存块进行缓冲，仅

⊖ 一种有趣的情形是，一个处理器向分配给另一个处理器内存中的某个字做写操作，第三个处理器再读这个字。在这种情形下，将数据从生产者到消费者引起了两种数据通信事件——一是“发送方启动的”，一是“接收方启动的”。

仅在同步点在合并消息中将它们发送，这在第 9 章提到的软件共享地址空间系统中特别有用。然而，这里将集中讨论块传输的显式发起。

11.4.1　技术和机制

显式块传送通过显式地发出一个 put 命令发起，与 send 命令相似，但由用户程序中的发送者提供源和目的地址，图 11-9 是简单的描述。其中，put 命令由通信辅助部件解释，然后将数据按照流水线的方式从源节点传输到目的节点。在目的端，通信辅助部件将数据从网络传输到特殊的存储区，其处理路径如图 11-10 所示。发送/接受消息传递机制的主要区别在于发送进程直接指定程序数据结构（虚拟地址）的能力。既然存储区是共享的地址空间，以上数据结构即为目的节点存放数据的存储区。由于将到的消息会指定数据应放的程序地址空间，接收操作在编程模型中是不需要的。如果目的节点的辅助通信是可用的并将数据直接从网络接口保存到内存中的数据结构，在目的节点中的内存系统中的缓冲和拷贝也是不必要的。然而，我们必须采用某种形式的同步（如对标志轮询或阻塞）以保证目的端进程在使用数据之前所需的数据已经来到，同时还要保证数据到来时目的端的存储区是可用的。我们也可能使用接收者发起的块传输形式，此时传输数据的请求是由接收者发起的，而由源的辅助通信来处理。

853

P_A	P_B
for i←0 to n-1 do	
A[i]←...;	
end for	
put (A[0..n-1] to tmp[0..n-1]);	
flag ← 1;	while (!flag) {}; /*spin-wait*/
for i←0 to n-1 do	for i←1 to n-1 do
compute f(C[i]);	use tmp[i];
end for	compute g(B[i]);
	end for

图 11-9　对于图 11-1 中的例子，在共享地址空间用块传送的情形。数组 A 分配在处理器 P_A 的本地存储器，数组 tmp 分在 P_B 的本地存储器

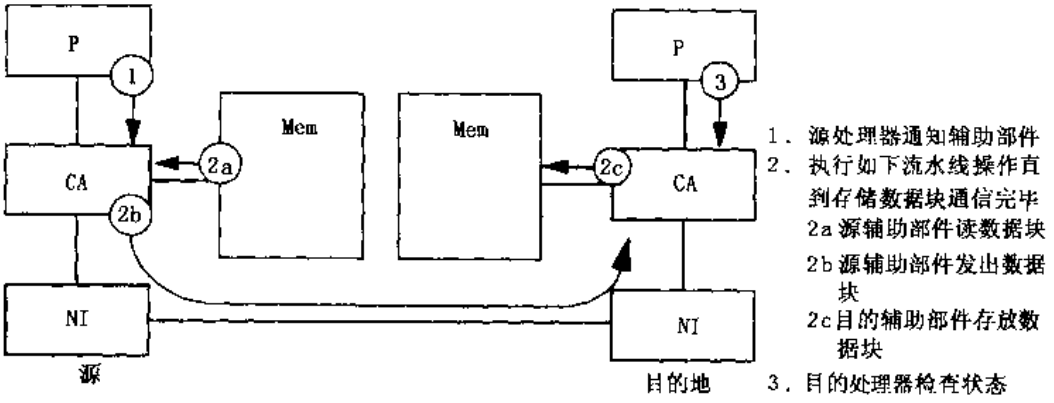


图 11-10　在共享地址空间的机器从源节点到目的节点块传送的通路。传送以缓存块为单位，因为它是机器优化所针对的通信的粒度

节点上执行块传送的通信辅助部件可以使用与处理一致性协议事务的同一硬件，也可以是专用于块传送的独立 DMA 引擎。通信辅助部件在功能设计上可有所不同，例如可以只允许连续的数据块传送、等步长、或者更通用的分发-收集（scatter-gather）操作。块传送可借助于对缓存块传送的有效支持，也可以是完全独立的机制。在一致性共享存储空间中块传送可能需要与一致协议相互作用，因此假定块传送是建立在整个缓存块的流水线传送的基础上；即块传送引擎是源端辅助硬件的一部分，该辅助器从内存中读出缓存块并以流水线的方式传送给网络。图 11-10 显示了在缓存一致性共享地址空间中可能的一种块传送实现步骤。

11.4.2 策略问题和折中方案

块传送中有两个有趣的策略问题：与基本的共享地址空间的交互以及缓存一致性协议如何处理，在目的节点上块传送的数据存放在哪儿。

854

1. 与高速缓存一致性共享地址空间的交互

第一个有趣的交互是与共享地址空间本身有关，而与是否包括自动一致缓存无关。在块传送中一个处理器“放”的数据可能不在其局部存储器中，而在另一个处理节点的存储器中或甚至分布在其他节点的存储器中。这里的选择不允许出现这样的传送，发起节点从其他存储中检索数据并以流水线的方式转发，或者让发起节点向拥有节点的通信辅助部件发消息并要求它们执行相关的传送。

由于目前同一个数据结构可以以两种不同的协议（块传送与缓存一致）通信，第二种交互是与具体的缓存一致性共享地址空间相关的。不管数据分配到哪个主存，它可能缓存在发送者的缓存中成脏态，在接受者的缓存中成共享态或在其他处理器（包括接收者）的缓存中成脏态。前两种情况造成了所谓的局部一致性问题；也就是说，保证发送的数据成为发送方的最新值，传输后保证接收方的数据成为一致状态。第三种情况形成了称为全局一致性问题；也就是说，保证系统中任何地址的值在传送后都成为最新值（按照一致性模型），而且与传输有关的数据在整个系统中都是一致的。再次重申，这里的选项并不保证以上三种中的任何情况；只是提供局部一致而非全局一致，或者完全的全局一致。每一个后续情况都使问题的工作更简单，但给通信辅助部件带来更多的要求。要提供一致，通信辅助部件必须检查每一个被传送块的状态，对从相应的缓存中检索数据并作废缓存中数据。传输的数据可能与缓存块不是对齐的，这就使得块级的一致性变得更为复杂，这与有发送节点不包含被传送数据的目录信息的可能性是一样的。显式消息传递编程模型在该方面的做法较简单：由于一个进程所能访问（或传输）的任何数据都在其私有的地址空间，并不能被任何其他的处理器缓存，因此它只要求局部一致而非全局一致。

在块传送中即使保持局部的一致性，通信辅助部件对每个缓存块都要做些工作，并成为传输流水线的完整部分。因此，这种通信辅助部件又可能成为传输带宽的瓶颈，甚至影响那些并不要求交互的缓存块。全局一致可能要求通信辅助部件在发送每个块之前向周围发送网络事务以戏剧化地减小带宽。对于一个块传输系统来说，“使用收费”的特性是很重要的：即一致性传输不应损害无需一致情况下的性能，尤其是后者是大概率事件时。例如，当块传送应用于仅支持显式消息传递的程序中，而不加速其他读写的共享地址空间程序中的粗粒度通信时，仅仅提供局部一致而非全局一致是有道理的。然而，像例 11.1 所描述的，要想基于一致读写共享地址空间提供真正完善的块传送，而且还不对编程模型进行限制，全局一致性是本质的。

855

例 11.1 给出需要全局一致的简单例子，假设情况为缓存一致性共享地址空间块传送。

解答： 考虑伪共享的情况。一个处理器 P_1 要将可能分配在 P_1 的本地内存数据发给 P_2 处理器，但另一个处理器 P_3 最近已经在向这些缓存块写入了其他字。则这些缓存块在 P_1 中是无效状态，为了将数据发给 P_2，P_1 必须将数据从 P_3 收回。假伪共享的例子如下：不难看出，在真正的共享地址空间的程序中，从 P_1 发送到 P_2 的字本身也许已经被另一个处理器更新。■

缓存一致机器中更详细的有关块传送情况可参考文献（Kubiatowicz and Agarwal 1993; Heinlein et al.1994; Woo, Singh, and Hennessy 1994; Heinlein et al.1997）。

2. 何处存放传送数据

其他的令人感兴趣的策略问题是块传送的数据放在目的端的主存中、缓存中还是两者都放。由于目的端会读数据，数据放入缓存是有用的。然而，这也有不利的地方。首先，会干预处理器缓存，这在现代系统中开销较大，而且同时会阻碍处理器访问缓存。第二，除非很快使用传送的数据，否则会被替换出缓存，因此为了减少局部缓存扑空应当存入主存。第三，也是最危险的，传送的数据会替换出缓存里的其他正处于当前处理器的活动工作集中的数据。由于这些原因，将大数据块传入小的第一层缓存是不可取的，而传入较大的第二层缓存中更为有用。

11.4.3 性能收益

在共享地址空间中使用大的显式传送比缓存块大小的隐式通信具有多种优点。然而，前面讨论的消息传递的一些优点会被打折扣，而且也有一些缺点。我们将定量地讨论折中方案并对一些性能数据进行分析。

856

1. 潜在的优点和缺点

以下是块传送的主要性能优点（前两种已在消息传递中讨论过，因此这里只指出其区别）。

- 分摊每个消息的开销。此优点也许并不十分重要，因为硬件支持的共享地址空间中的块传送的终点开销已经很小。事实上，显式的可变大小传输比小的固定长度传送更趋向于较大的终点开销，因为备用的缓冲硬件很容易实现，而前者需要软件管理和拷贝。结果，块传送每次通信开销可能大于读-写通信。很多系统的块传送机制很像 DMA 设备，且对物理地址进行操作，因此它们运行于内核模式而且请求系统调用，大大增加了开销。这种开销增加成为几种商用的硬件一致性机器主要的阻碍，目前这些设备主要用来处理系统操作（如页面迁移等）而不是应用程序。然而，在效率不高的通信结构中，例如一些由商品化部件构成的系统，其每次通信开销（甚至缓存块传送）是很大的，基于块传送的开销分摊则显得非常重要。
- 大片数据的流水线传送。
- 浪费的带宽较少。通常，消息越大，则消息头和路由信息相对于有效载荷的开销就越小。在现存的缓存行传送机制中每个缓存块都要发送一个头消息。因此，若建立在这种机制上，块传送的上述优点就会消失。运用适当，则块传送也能减小协议（如，无效、确认等）的消息数。
- 传送数据在目的节点主存中的复制。由于块传送通常写入主存，以后在目的节点就会造成局部容量扑空。这就会减小目的节点不得不执行的远程容量扑空，就像 COMA 机器一样。然而，不采用 COMA 体系结构，那么用户就要管理主存中复制数据的

一致和替换。

- 同步和数据传输的绑定。同步标志可以与传输的消息结合在一起，不必将数据传输与同步分离。虽然缺少显式块接收操作，从功能上讲意味着在端节点同步管理仍然要与数据传输分开，这与异步消息传递一样，但这会减小需要的消息数。

857

块传送的潜在的性能缺点：

- 每次传送开销高。
- 冲突增加。长消息会在终点以及网络中造成较高的竞争率，因为它在每次发送时占用更多的资源；其提供的时延包容要求通信体系结构具有高的带宽。另一方面，前面已讨论过，与缓存一致性协议相比，协议消息要求的带宽相对减少。
- 额外工作。为了进行大块传输（若能有效的完成），程序必须做额外的组织工作。这种额外的工作有时会比块传输带来的效益代价还高（参看第 3 章 3.6 节 Barnes-Hut 的应用）。

例 11.2 说明了使用块传输而不用读-写的性能改进，尤其是在分摊开销和流水线数据传输方面。

例 11.2 假设我们将 4 KB 的数据从源节点传送到目的节点，其中使用缓存一致性共享地址空间机器，一个缓存块大小为 64 字节。假设内存中的数据是连续的，以便空间局部性可得到很好地开发。（当空间局部性不是很好时，习题 11.6 讨论的问题会发生）假定从内存中读出一个缓存块，源通信辅助部件花费 40 个处理器周期，需要 50 个处理器周期通过网络接口发出，接收者的补充操作也花费同样的处理器周期。假设局部读扑空时延为 60 个周期，远程读扑空时延为 180 个周期，启动一个缓存块传送时间为 200 周期。采用块传送方式比缓存扑空方式有多少性能提高？假设缓存操作时处理器阻塞存储操作直到完成。

解答：从处理角度看，通过读扑空得到数据的开销为：180 × （4 096/64） = 11 520 周期。使用块传送时，传输流水线的最大速率为 max（40，50，50，40）即 50 周期/块。这就将数据传到了目的节点的局部存储区，处理器通过局部读扑空读入，每次扑空开销为 60 个周期。于是，目的处理器通过块传送得到数据的开销为 200 + （4 096/64） × （50 + 60），即 7 240周期。因此，使用块传送的加速比为 11 520/7 240 = 1.6。■

实现块传送的另一种方法是使用向量操作，例如对远程内存的向量读。在这种情况下，一个指令就可将数据读入向量寄存器，并不需要多条的装入/保存指令，同时可节约指令带宽和局部缓存扑空。通常的向量寄存器是由软件管理的，其缺点是寄存器别名或名字绑定，但优点是减少缓存冲突。然而，现在的许多高性能系统并不含有向量操作，因此我们将集中讨论需要单个局部读写操作访问和使用数据的块传送。

858

2. 实际程序中的性能效益和限制

块传送能否有效地应用于实际程序依赖于程序和系统的特性。现考虑一个服从块传送的简单例子，网格上的近邻方程求解器，以分析性能受到的影响。在此忽略前面讨论的一致性的复杂性，并假设传输是从源节点的主存到目的节点的主存，而且在任何地方数据不被缓存。我们假设一种二维网格的四维数组表示，以及网格在处理器局存中的一种划分。

如图 11-11 所示，这里处于分区的元素并不是通过单独的缓存块通信，而是一个进程可将包含所有相应元素的消息一次传送给其相邻者。通信与计算的比率与 $n/\sqrt{P}$ 成正比[⊖]。由

⊖ 原书为 $\sqrt{p}/n$。——译者注

于块传送的目的是提高通信的性能，该比率本身表示了随处理器的增加块传送的相对有效性。然而，每个传输块大小为 $n/\sqrt{p}$ 个元素，因此随 p 的增加而减小。传输块的减小意味着块初始化开销相对重要。这两种因素的折中产生了一个理想点，该点是在给定网格大小（如图 11-12 所示）时，块传送最有效的处理器数。当网格大小增大时，该点移向更大的处理器数。

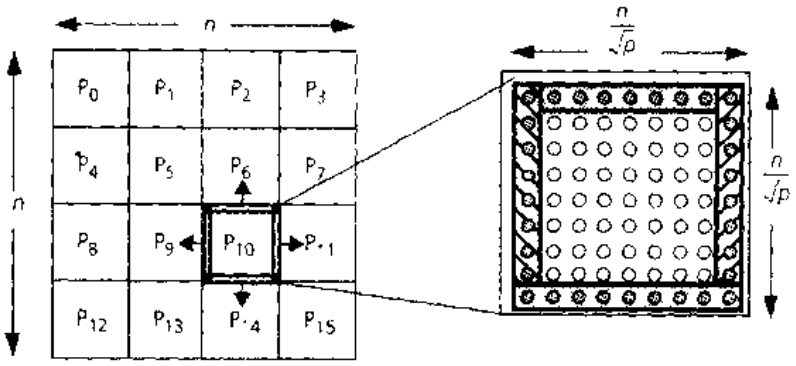


图 11-11　在近邻方程求解器中块传送技术的使用。一个进程可以将它的划分中的一整行或者列以一个消息发送给在单个消息中拥有邻居划分的进程。每个消息的大小是 $n\sqrt{p}$ 个元素

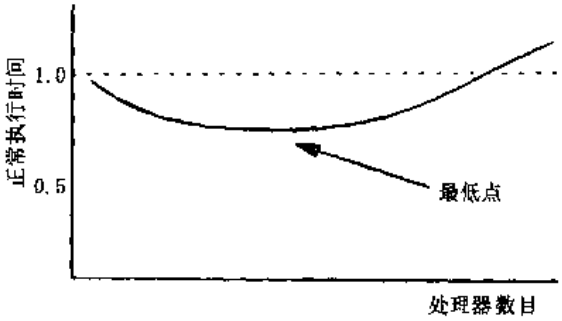


图 11-12　块传送所带来的相对性能改善。图中给出了使用块传送正常化使用装载和存放的执行时间。最低点发生在通信量足够大，并且开销不起支配作用的时候

图 11-13 表示了在 FFT 应用中该理想点的效果，其环境为 Stanford FLASH 多处理器系统的模拟体系结构，与 SGI Origin2000 十分接近（虽然它的块传送能力和性能更强）。我们看到当缓存块增加时块传送对于一般的缓存一致通信的相对收益逐渐消失，这是因为该程序中的极好的空间局部性使得长缓存块本身的行为与小块传输类似。对于最大为 128 字节的缓存块（SGI Origin2000 中的实际块大小），使用块传送的收益很小，即使用非常有效的块传送器。图 11-14 是在 Ocean 上的结果，这是图 11-11 表示的基于最近邻接通信的更完全、更规则的应用。在面向行划分块传送的连续数据的边界，这具有很好的空间局部性；在按列划分块的边界时，空间局部性就较差；如何开发好块传送还存在困难。当用基于整个缓存块的流水实现块传送时，除非处于列边界的字首先拷贝到一块连续的数据结构中，否则边界的每个字都会通过单独的缓存块传送。总的来看，Ocean 中的通信和计算比要比 FFT 小。虽然随着处理器数减小该比率在多网格求解器的网格层的高层次上会增加，但该层次上的块传输仍然很小，并不能分摊开销。其结果表示块传送对这样的应用没有多少帮助，而且块传送的相对收益对缓存快大小的依赖性并不很大。其他应用的定量分析数据参看（Woo，Singh，and Hennessy 1994）。

块传送可能在一些并行管理方面是有用的。例如，在任务窃取环境中，如果被窃取任务的描述较大，那么其传输可用块传送；有关的数据情况也是一样的，而且有关任务队列需要

的同步也可与这些传输捆绑在一起。

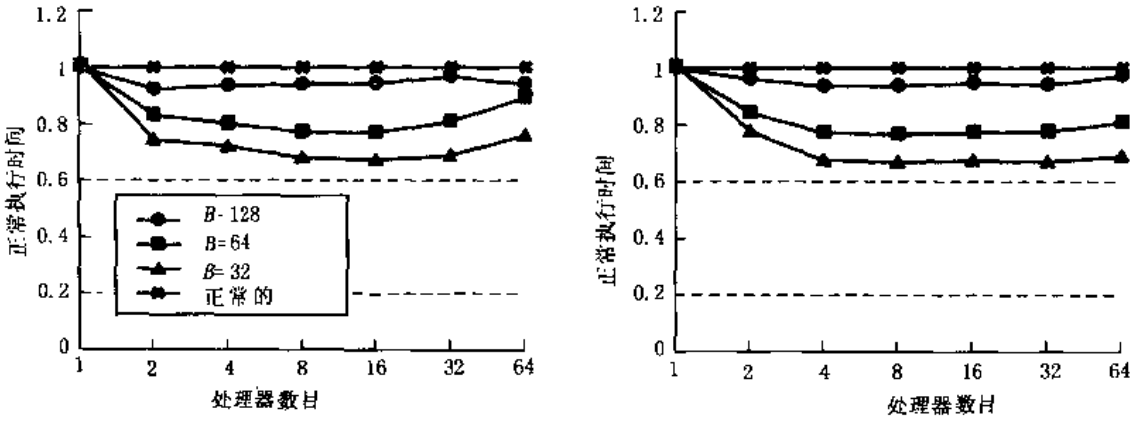


图 11-13 在快速傅立叶变换程序中成块数据传送带来的收益。两个图分别针对两种不同的问题规模。所用的体系结构模型和参数类似于 Stanford FLASH 多处理器。(Kuskin et al.1994) 也和 SGI Origin 2000 相似。每条曲线针对二级缓存中不同的缓存块大小 (B)，y 轴表示规格化后的执行时间，规格化基准是同样缓存块大小但没用块传送方式的执行时间。这样，不同曲线的规格化基础是不同的(自规格化)，相同 x 轴数值对不同缓存块大小可能对应相同的执行时间，但它们在图中不一定规格化到同样的 y 值[⊖]。一个点的 y 值越大，意味着块传送相对于普通读写通信的性能改善越小

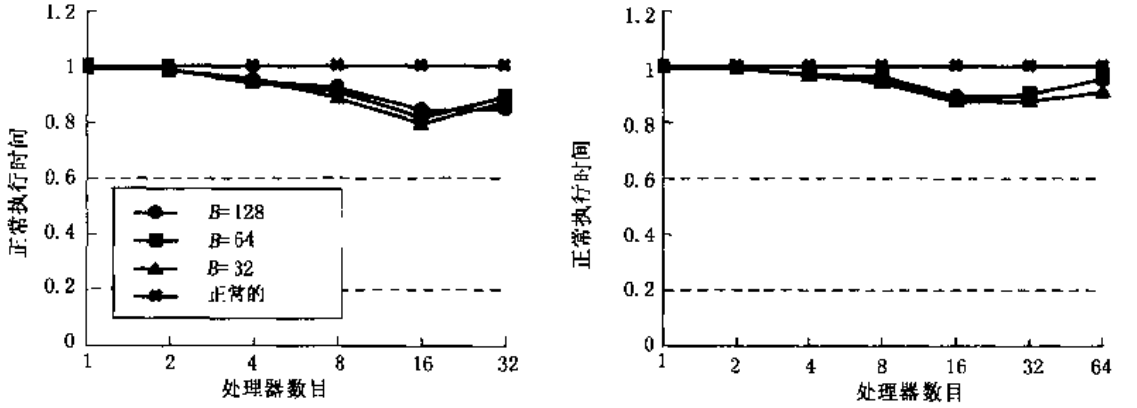


图 11-14 在 Ocean 应用程序中成块数据传送带来的收益。其平台和数据的含义与图 11-13 的完全相同。B 是机器的二级缓存块大小。得到的收益以及对缓存块大小的依赖性不像快速傅立叶变换程序那么大

块传送能获益的明显的一种情况是：启动远程读写访问的开销非常高；例如，当共享地址空间用软件实现时即是如此。要知道通信体系结构其他参数（如网络时延、点对点带宽等）的效果，就得对块传送进行更多的分析。在此我们仍然假定读扑空使处理器停滞，且没有时延隐藏。通常来说，通过远程读扑空传送一个大数据块的时间是：*读扑空数 × 远程读扑空时间*，通过块传送使数据到达目的处理器的时间是：*启动开销 + 缓存块传输次数 × 每个缓存块的流水段时间 + 读扑空数 × 局部读扑空时间*。最简单的情况是可以忽略启动开销的非常大的连续数据片，并假定具有很好的空间局部性。在这种情况下，块传送的加速比限制在下面比率：

859 ~ 861

$$\frac{\text{远程读扑空时间}}{\text{块传送流水段时间} + \text{局部读扑空时间}} \tag{11-1}$$

其中块传送流水段时间是图 11-10 所示的每个流水段中的最大值。

⊖ 这一句原文过于简略，并有错；这里的译文是细化了的。——译者注

一个长的网络时延意味着远程读扑空时间的增加。若点对点的网络带宽并不随着网络时延增加而减小，则其他的条件不受影响，同时较长的时延可支持块传送。另外，网络带宽也可以随着时延增加而成反比例减小；例如，网络时延是通过减小网络时钟速度引起的。此时，随着时延增加，在某一点数据进入网络的速度就比通信辅助部件读写内存的速度小。直到该点，存储访问时间决定着块传输流水线分段的时间，于是网络时延的增加并没有改变块传输的性能，因此这是有助于块传输的。超过该点之后，两者比率的分子和分母随着网络时延以相同的速度增加，因此块传送的有关优点仍保持不变。

最后，假设时延和开销固定，而带宽变化（如给网络链路增加连线）。我们也许认为基于远程读停滞的通信是受时延约束的，而块传送是受带宽约束的；于是增加带宽应当有助于块传送。当网络带宽限制块传送流水线分段时间时，这种说法是对的。然而，当增加网络带宽超过该点时，就意味着存储访问时间将成为瓶颈，因此继续增加带宽并不能改进块传输的性能。减小带宽则具有相反的效果。因此，当其他变量保持不变时，块传送在下列情况下会更有效：增加每个消息开销，增加网络时延，增加带宽，直到某一点。

总之，通过读-写缓存一致通信实现的块传送，其性能改进依赖于下列因素：

- 在块传送中花费在通信上的执行时间部分。
- 为了块传送组织通信而必须的额外工作。
- 问题大小和处理器数，这会影响通信和计算比与传送的大小。
- 系统的开销、时延与带宽。
- 程序的空间局部性以及与传输的粒度如何进行交互。

862

如果我们能够使得所有消息变得足够大，通信的时延部分可能不是问题；则带宽成为重要的限制。但是，如果不能，我们仍然还要通过重叠计算或其他的访问来隐藏访问数据的时延。其他三种时延隐藏方法即是这样做的；然而，要想对缓存块大小的传输取得成功，还需要在微处理器级进行支持。除预通信之外，下面我们将讨论在同一个线程计算中移过长时延访问的技术。

11.5 跨越长时延事件

若访存操作是非阻塞的，则处理器可越过访存操作而执行其他的指令。对于写操作，这种跨越可直接进行：将写操作放入写缓冲，处理器可继续进行。写缓冲可负责将写操作发送给存储系统，而且跟踪其操作的完成。很多处理器可支持非阻塞的读操作：读操作进行时，处理器可执行其他的指令。若没有另外的支持，当处理器遇到依赖存储的结果时就会停滞。而问题是在多数的程序中相关指令往往距离读操作很近。如果读在缓存中扑空，该情况下只有很少的时延被隐藏。有效地隐藏时延，需要越过读指令提前察看并在指令流中发现多条与读指令无关的指令。这需要在编译指令调度、硬件或双方的支持。隐藏写时延通常不需要指令预测：由于一般不会很快遇到相关的指令，因此在遇到相关指令时可使处理器停滞。

在完成之前跨越该操作执行可从缓冲和流水两方面得到好处。存储操作缓冲可允许处理器继续进行其他的行为，可推迟请求操作（如作废操作）的传播与应用，也可合并对同一个字的操作以及对缓冲缓存块的操作（第9章讨论的基于页的、完全软件的、共享虚拟存储协议是缓冲、推迟传播直到同步点的极端例子）。使多个存储操作流水进入存储系统可允许其时延重叠。这些优点在单处理器以及多处理器中都存在。

在多处理器中，在完成或提交之前跨越存储操作会违背顺序同一性的充分条件，这在第5章已讨论过。它是否真的在实际中造成 SC 冲突还依赖于其涉及的操作对程序执行顺序的改变是否是可见的。若要求来自于同一线程的存储操作不应当（包括在其他处理器中）出现改变程序顺序的执行，SC 就会限制（这是无法消除的）所能开发的缓冲和可利用的重叠的数量。放松同一性模型允许较大程度的重叠，包括允许一些类型的存储操作可乱序执行。因此可能重叠的程度取决于机器机制和同一模型：若不提供支持重排（例如写缓冲、编译或动态调度）机制，松散模型也不会起作用，而且一些能实现重叠的更强类型受到同一性模型的限制。

863

我们对跨越操作执行的讨论是围绕越来越先进的机制组织的，这些机制需要重叠读写操作的时延。其中的每种机制是与一类特别的同一模型自然相连的；该类模型允许其操作乱序完成，但是通过允许重叠而不是乱序完成，这些机制可应用于更严格的同一性模型并在方式上更加限制。对于每种情况，我们将分析其性能增加和实现复杂性（假定需要完全开发其重叠的最强的同一性模型），以及顺序同一所能开发重叠机制的程度。虽然很多问题很自然地适用于非缓存共享数据或软件实现同一，我们还是集中于硬件缓存一致性系统。为了简化讨论，首先分析只隐藏写时延的机制，因为读时延隐藏需要更详细的支持（虽然很多现代微处理器都支持写时延隐藏）。开始我们假设读阻塞，因而处理器不能跨越它们；以后也会分析隐藏读时延。

11.5.1　跨越写操作

我们首先开始讨论的是一个简单的、静态调度的支持阻塞读的处理器。要想跨越写扑空，我们需要处理器的惟一支持是写缓冲。我们将发生在一级缓存中的扑空的写操作简单地放在缓冲里，处理器可继续执行同一线程中的其他工作。处理器（或一级缓存）只有当写缓冲满时才引起写停滞。写缓冲也可以放于一级缓存的前面，此时所有的写操作都放在该缓冲里。

写缓冲负责控制着对其他扩展的存储层次结构（也包括其他处理器）写操作的可见性，而且也负责完成该处理器中的写操作。这也缓解了处理器内的执行单元，不必担心对扩展的存储层次结构写操作的顺序。现考虑与读操作的重叠。为了单处理器的正确操作，读可对写缓冲进行旁路，而且只要对同一地址的写操作在缓冲中没有挂起，读就可以发送给存储系统。如果在缓冲中挂起，来自于写缓冲的值可以转发给读操作（甚至在写操作完成以前），否则等到写缓冲刷新后再发送读操作。转发可允许程序中的读与其前面的写乱序完成，因此在多处理器中与顺序同一冲突；而刷新操作则不会。除非读操作不能在其旁路的写之前完成（连同数值），否则旁路写缓冲的读也会与顺序同一发生冲突。因此，SC 可从写缓冲中得到好处，但其收益是有限的。从第 9 章我们知道，处理器同一(PC)和所有按序保存是只允许读先于前面的写完成的同一模型。

864

现在讨论写之间的重叠问题。多个写可放于写缓冲里，该缓冲也确定了其可见的顺序或完成的顺序。如果写允许无序完成，则缓冲就可开发出写操作中的大量重叠并具有更大的灵活性。首先，若写的地址或缓存块在当前的缓冲中，则该写操作可与该缓存块合并；而且只能由该缓冲发出单个的所有权请求，这样也减小了拥挤情况。尤其是当该合并不是最后的缓冲项时，这就使写操作的程序乱序成为可见。第二，当缓冲的写发向存储系统时，它们可在缓冲中引退，这也使得后面的写操作在其完成之前跨越该操作。这也允许写所有权请求流水通过扩展

存储系统，但会使得该写操作的程序乱序对于其他处理器可见，并可能危及写的原子性。

部分保存顺序（PSO）或更放松的同一性模型允许写操作程序乱序完成。更严格的模型，例如 SC、TSO 和 PC 从本质上限制写合并到写缓冲的最后一项（即使是在严格限制的环境中），因为向缓存块中不同字的不同顺序写对其他处理器来说不应当是可见的。只有当写操作的可见和完成顺序在扩展存储系统中可保留时，在较严格的模型中才有可能提前引退写操作以使其他的写操作越过。这在基于总线的机器中相对容易些，但是在分布式存储机器中则十分困难，因为该机器具有不同的局部内存和独立的网络路径。在以下的系统中，保证程序中的写顺序（像 SC、TSO 和 PC 所需要的）从本质上要求写操作在所有涉及的处理器提交之前不能从 FIFO 缓冲的头部引退。

总之，一个像 SC 这样的严格模型能利用写缓冲实现用读操作或其他写操作重叠时延，但是要受到一定程度的限制。要想隐藏更多的时延，就需要更放松的同一性模型。在松散模型中，何时写操作发向扩展存储层次系统并对于其他处理器可见，也依赖于对实现和性能的考虑。例如，只要写操作能到达写缓冲就可发出作废操作，或者在缓冲中延迟这些操作直到下一个同步点。后者的选择允许大量的写合并，以及减小由于伪共享导致的作废和扑空。然而，这也隐含着作废相关的传输量将突加给同步点，而不是流水计算。

性能影响

假定阻塞读但并不保持顺序同一（Gharachorloo, gupta, and Hennessy 1991a; Gharachorloo 1995），在原始的 SPLASH 应用组上的并行应用模拟研究已显示出允许处理器跨越写操作的优点（Singh, Weber, and Gupta1992）。其导致的技术分为几个独立部分：允许违反程序中的读-写顺序的，这些技术可满足 TSO 或 PC；同时允许违背程序中的写-写顺序的，这些技术可满足 PSO。其中后者在本质上是最优秀的，当读阻塞时，像松散存储顺序（RMO）、弱序（WO）、或释放同一（RC）这样的更松散的模型都可以实现。

865

图 11-15 显示了单发射静态调度处理器执行时间的减小（分为两部分），主要针对两类并行程序当写-读和写-写顺序允许违背时的结果。这些程序是 Ocean 与 Barnes-Hut 应用的老版本，在 16 个处理器模拟的基于目录的缓存一致性系统上运行问题较小的程序，其指令调度比目前的微处理器都很落后。比较的基准是顺序同一的直接实现，并满足充分条件：当处理器发送一个读或写时，处理器停滞，直到该访问完成。（仅仅使用写缓冲——通过阻止读直到写缓冲空——保持顺序同一，与在所有写时停滞进程相比，显示出非常小的性能改进。）

图 11-15 中的第二个竖条表示，使用较深的写缓冲以及这些系统假设，仅仅允许写到读重排一般就足可以隐藏大多数的处理器写时延。由于存在很频繁的写扑空，在 Ocean 中并不是很成功。整个研究还显示出了一些有趣的附加效果。在某种情况下，写停滞时间比基本 SC 有轻微的增加。这是因为隐藏写时延对带宽的额外要求与读扑空发生了冲突，增加了开销。虽然这对于动态调度的处理器隐藏读时延具有更要的意义，但这对于简单处理器来说具有相对小的影响。另一个有益的效果是有时也减小同步等待时间。随着存储停滞时间的减小，不同处理器间存储停滞时间的不平衡也会减小，因而减小了负载不均衡。同时，如果一个关键部分中的写时延被隐藏，那么该关键部分就会更快地完成。保护锁可更快地传送到另一个处理器，这样也减小了对锁的等待时间。

图 11-15 中的第三个竖条表示了写到写顺序也可违背时的结果。同一个 16 项写缓冲中的

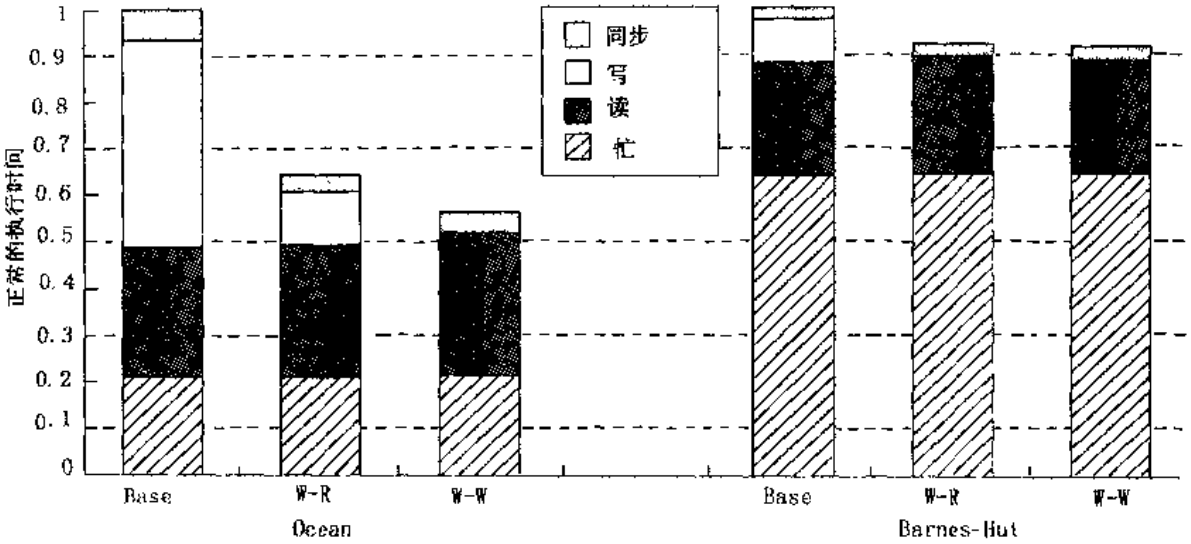


图 11-15 利用放松同一性模型跨越写操作的性能收益。这些结果是假定在静态调度的处理器模型中得出的。对于每种应用，Base 代表没有时延隐藏的情况，W-R 代表读可旁路写的情况（如，写-读顺序可违背的情况），W-W 代表写和读都可以旁路写的情况。其中 Base 情况的执行时间规格化成 100 个时间单位。被模拟的系统是使用扁平的缓存一致性的基于存储器的目录协议，很像斯坦福的 DASH 多机系统（Lenoski et al.1993）。该模拟器假定具有写穿透一级缓存和回写二级缓存，二级缓存之间设置较深的 16 项的写缓冲。该处理器是单发射且静态调度的，在编译时也没有面向时延隐藏的特殊调度。其写缓冲是较激进的，读操作既可以旁路也可以前递。为了保留写-写顺序（此时不允许重排写操作），写操作之间不允许合并，而且缓冲区头的写操作只有完成时才能引退。对于一个读扑空数据访存参数假定：一级缓存 L_1 命中时间为 1 个周期，L_2 命中时间为 15 个周期；如果扑空并访问局部存储期的时间为 29 个周期，两个消息远程扑空时间为 101 个周期，三个消息远程扑空时间为 132 个周期。以上情况中的写时延是非常小的。该系统假定了比现代系统的存储系统更慢的处理器

任何块都可以进行写合并，只要写操作到达了缓冲头就可以引退（即使还没有提交）。这样，通过存储和互连的写流水线更多时候比写缓冲具有更高的优先级（这可允许合并和延迟的作废操作）。当然，允许存储系统进行多个写也要求缓存同时处理多个待完成的扑空。

写-写重叠甚至在 Ocean 中也可隐藏写-读重叠隐藏后的剩余写时延。由于写缓冲中的引退速度较快，因此写缓冲并不会轻易填满而停滞处理器。由于在释放之前一系列的写操作会很快地完成，同步等待时间在某些情况下也进一步减小。然而在多数应用中，写-写重叠很少发挥其优点，因为多数的写时延可通过允许读在写之前旁路被隐藏。限制写-写重叠效果的另一个因素是处理器假定是读阻塞的，因此写-写重叠不能跨越操作。两种模型间的区别（如弱序和释放同一）以及多模型（如 TSO：保持写原子操作；PC：不保留）间的细微差别看起来并不能实质上影响性能。

由于性能很大程度依赖于实现、问题和缓存大小，因此评测一些并不激进的写缓冲、二级缓存结构和缓存大小（该缓存不能完全容纳所有的重要工作集）是很有用的。正像我们所希望的，一个免锁定的二级缓存对获得性能改进是很重要的。可旁路的写缓冲对于允许写到读重排的系统是非常重要的，尤其当二级缓存是自由查找时；但对于同时也允许写到写重排的系统并不很重要。原因是前者的从写缓冲中写引退是在完成之后，因此极有可能当读操作在第一层缓存中扑空时，一个写仍然在写缓冲里，这可阻塞读直到写缓冲空，进而造成性能损害。对于后一种情况，写引退非常快，因此在写缓冲里发现一个写来旁路的可能性很小。

867

同样的原因，当允许写到写重排时写缓冲的大小并不是关键的。若没有自由查找缓存，无论写到写重排是否允许，读旁路缓冲里写的能力很少有优点，因为停滞二级缓存将成为瓶颈。系统的所有部分都应适当地设计以获得重叠效益。

采用较小的 I_1 和 I_2 缓存的研究结果如图 11-16 所示，其中是改变缓存块大小时的结果。首先考虑缓存的大小。采用小缓存时，写时延仍然可通过允许写到读和写到写重排有效地隐藏（像第 4 章讨论的，对于 Barnes-Hut 最小的缓存并不能代表理想情况）。有趣的是，隐藏写时延对总体性能的影响要比大的缓存大的多，即使隐藏的写时延总和是很小的。这是因为大缓存对于减小读时延比减小写时延更有效，因此后者的时延隐藏显得更重要。所有这些情况都假设缓存块的大小为 16 个字节，在今天是很不理想的。大的缓存块希望减小这些应用的扑空率，从而在某种程度上减小这些重排对执行时间的影响，正像在图 11-16 b 中所看到的 Ocean 那样。

11.5.2 跨越读操作

要想有效地隐藏读时延，既需要非阻塞读，又需要跨过相关指令的预测机制。由于转移指令会在相关指令不远后出现，编译和硬件处理都是复杂的。通过代码预测将来的路径而发现无关指令，要求有效地转移预测和推测式执行跨过预测的转移指令。推测式执行指令反过来要求在错误预测时取消推测指令执行的影响。

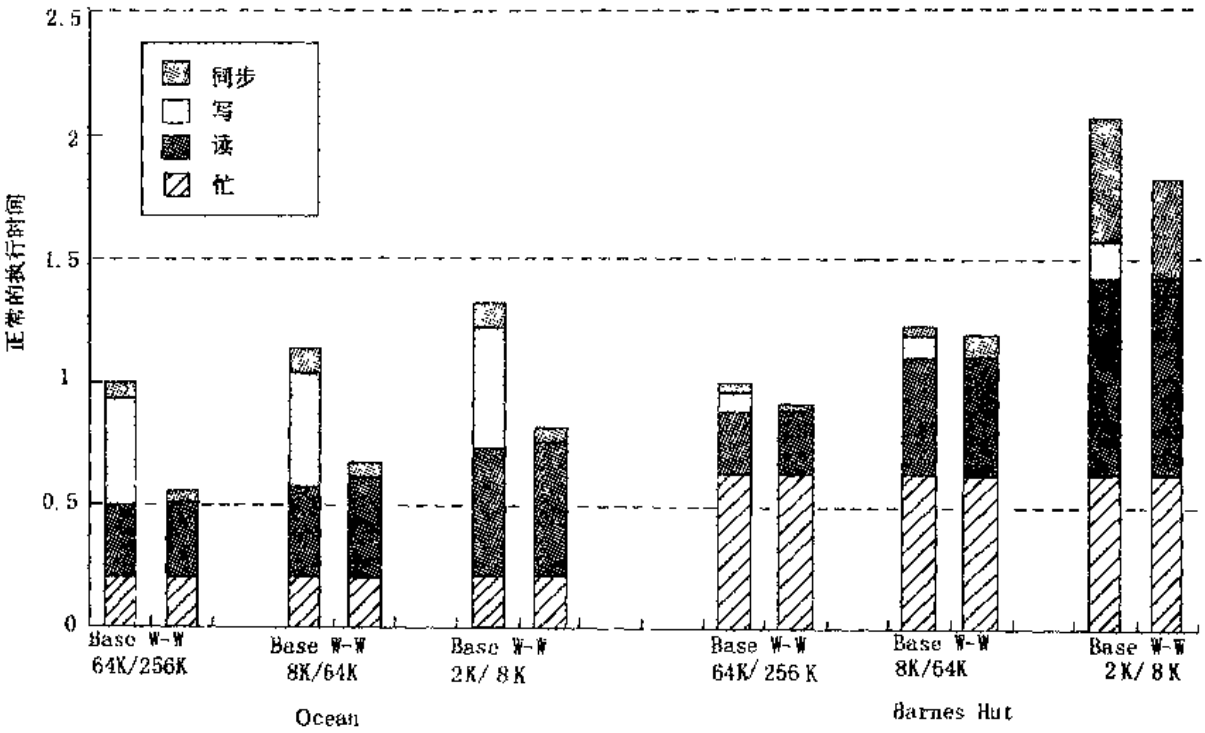
今天微处理器工业的趋向是在硬件上提供所有这些特征的越来越复杂的处理器。例如，它们包括在如 Intelpentium Pro（Intel 公司 1996）系列、Silicon Graphics R10000（MIPS 技术 1996）系列、Sun UltraSparc（Lee，Kwok，and Briggs 1991）系列、惠普 PA8000（Hunt 1996）系列中，因为存储系统和功能部件中的时延隐藏在单处理器中也是十分重要的。这样的设计和机制是昂贵的，但是既然微处理器中都出现了，多处理器中也会同样采用。一旦写时延也能够隐藏，单独的写缓冲就不需要了。

868

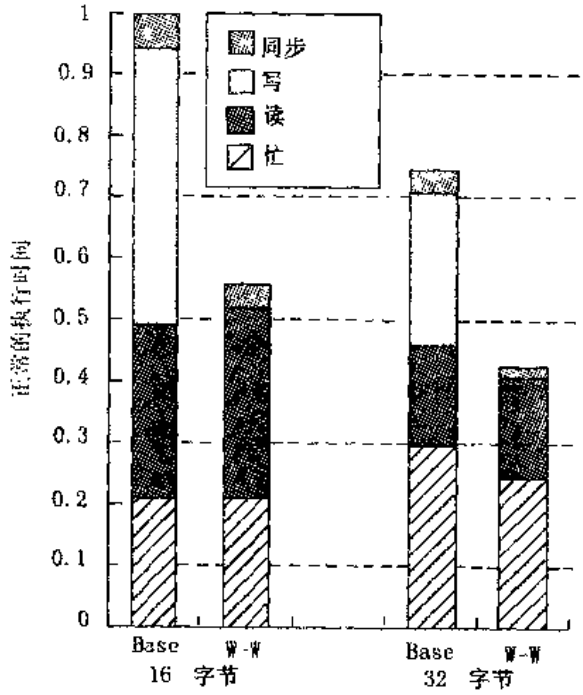
1. 动态调度与推测式执行

为了更好地理解如何利用动态调度技术和推测式执行隐藏存储时延，尤其是这些技术如何与存储同一性模型相交互，我们首先回忆在单处理器中这些方法是如何工作的。更详细的资料可在一些书中查到，如（Hennessy and Patterson1996）。动态调度是指指令的取指和译码是按照编译的顺序进行的，但它们在功能部件的执行顺序是按照在执行过程中操作数的有效性来执行的。一种协调乱序执行方法是通过保留栈和 Tomasulo 算法来实现（Hennessy and Patterson1996）。动态调度并不意味着存储操作成为可见的或者可不按程序顺序完成，下面可以看到。推测式执行是指允许处理器考虑和调度将来对程序执行不一定有用的执行指令，如越过一个将要转移的指令。功能单元假设这些指令有用而执行它们，但直到推测的有效性确定后（如转移的结果完成），其结果才提交给寄存器或被其他系统可见。

推测式执行的关键机制是指令预测或重排缓冲区。指令译码后，无论是内嵌的或推测式地越过预测的转移，它们都被放入重排缓冲区。该重排缓冲区保持指令的程序顺序。其中，那些不依赖于未完成操作的指令将被选择执行。排在缓冲前面的未完成或未执行指令很有可能是无关的长时延指令（如，读），于是在这种情况下处理器跨过未完成的存储指令或其他指令。一旦操作数被其他指令产生，重排缓冲中的指令就变为就绪准备执行，并不需要等到其操作数出现在寄存器堆中。指令将其结果保存在重排缓冲区里而不是提交给寄存器堆，只



a) 缓存尺寸的有效性



b) 用于Ocean 的缓存块尺寸的有效性

图 11-16 缓存大小和缓存块大小对跨越写操作的影响。在 a) 中，改变缓存大小时，我们假定缺省的缓存块是（小）16 个字节大小。X 轴显示的是 L_1 和 L_2 的缓存大小。两者被斜线分开。b) 显示的是缓存块大小变化，并假定 L_1 为 64 KB，L_2 为 256 KB。其中 Y 轴是规格化的执行时间，以便使得每个图中最左边的矩形条为一个单位

要到达缓冲区的顶部时就引退（即按照程序的顺序）。只有在该引退点，读或其他指令的结果才被放入寄存器堆，写的数据才能被存储系统可见。因此，即使是最激进的乱序执行，存储操作也能按程序的顺序完成。

按照程序的顺序引退指令可使推测简化：如果转移在重排缓冲中引退前发现预测错误，则该转移后的指令（按程序顺序）还没有从重排缓冲中引退及提交结果。此时，读还没有更新寄存器，写的结果还没有被存储系统看到。当检测到错误预测时，转移后的所有重排缓冲中的指令置为无效，其相关的保留栈也无效，译码也从正确的目标地址重新开始。按序引退也使得精确例外容易实现。然而，按序引退意味着若读扑空在其数据回来之前到达重排缓冲的顶部，则重排缓冲区（非处理器）就会停滞而且后面的指令不能引退。这种 FIFO 特性以及停滞意味着时延隐藏的程度可能受重排缓冲区大小的限制。

869 ~ 870

2. 在顺序和释放同一性中隐藏时延

一个操作从重排缓冲引退与完成的不同是很重要的。当读操作的值确定后就完成了，之前也可能已到达了重排缓冲的顶部而且能引退（修改处理器的寄存器）。另一方面，一个写操作即使已到达了重排缓冲的顶部而且进入存储系统，该操作也未完成；只有当写已被所有其他处理器看到后才完成。知道这些区别可以有助于我们理解无序和推测式处理器在 SC 和像释放同一性等更松散的模型中是如何隐藏时延的。

在 RC 中，一个写操作可能在完成之前从缓冲引退，以允许后面的操作更快地引退。在 SC 中，一个写操作只有在完成之后引退（至少提交给全部的相关处理器），因此一旦到达重排缓冲的顶部并发给存储系统，它会阻塞较长的时间。在 RC 下，一个读操作发给存储系统后，在插入到重排缓冲中后的任何时候都可能完成，除非其前面的请求操作还没有完成。在 SC 下，虽然读可以发给存储系统而且可在到达缓冲的顶部前完成，但是在缓冲区其前面的所有存储操作完成之前它不被发给存储系统（无需引退）。因此，SC 比 RC 开发的重叠要少，但是区别并不像按序执行那样大。

3. 加强时延包容的其他技术

其他技术——包括硬件预取、推测读和写缓冲——可与动态调度、推测式处理器结合在 SC 与 RC（Gharachorloo、Gupta，和 Hennessy 1991 b）中进一步隐藏时延。硬件可发送重排缓冲里的预取存储操作，但在同一模型中还不能实际发送给存储系统。例如，处理器可能预取读操作，对于 SC 在缓冲里其前指令可能为未完成的存储操作，或者在 RC 下为未完成的请求操作，于是将它们重叠。对于写操作，预取允许数据及所有权在写实际到达缓冲的顶部前达到缓存，而且可以发射给存储系统。本质上，独占的写处理较早发送。这些预取是非捆绑的，意思是数据到达缓存，但并不进入寄存器或重排缓冲，而且对于一致协议仍然是可见的，因此预取并没有违背同一模型。如果该块在读提交之前是无效的，其损害是少量的额外存储流量。（更一般情况下的预取将在 11.6 节讨论。）

871

当被预取的地址未知时硬件预取并没有效果，而且确定地址本身要求一个长时延操作的完成。例如，如果一个对数组项 $A_{[I]}$ 的读扑空之前发生了对索引 I 的读扑空，于是这两个操作不能重叠；因为处理器直到读 I 操作完成之后，才知道 $A_{[I]}$ 的地址。I 可以预取，但是得到 $A_{[I]}$ 的地址要求 I 读操作完成。为了增加重叠，处理器可使用推测式读操作，推测式读的意思是在读完成之前同一性模型允许读推测地完成，亦即在到达重排缓冲顶部之前进行读推测，而且在顺序同一的环境下进行。其地址可用作以后存储操作的地址，但是其使用对于转

移后的指令应进行保护。实质上，处理器推测预取值在读推测与实际读按照存储同一模型执行之间不会改变；这样，推测的读及所有之后操作的效果应当是可恢复的。

在本例中，I 不仅被预取，而且在它到达重排缓冲顶端之前也进行读推测，因此对 $A_{[I]}$ 的读也可以提前预取。我们需要保证 I 的值是正确的，这才能读到正确的 $A_{[I]}$ 值。由于该原因，推测的读并不装入到实际的寄存器中而是装入到推测读缓冲里，直到读在重排缓冲中引退。该推测读缓冲监测缓存而且报告对那些块的行为。如果是作废、更新或在其地址在推测读缓冲里的一个块的缓存发生替换，则推测读以及其后的重排缓冲中的所有指令都应该取消而重新执行，像错误转移预测一样。这样的硬件预取以及推测读支持已出现在处理器中，像 Pentium Pro（Intel 公司 1996），MIPS R10000（MIPS 技术 1996）以及 HP PA-8000（Hunt 1996）。值得注意的是在 SC 下，在前面的存储操作完成之前发送的存储指令都是推测读，而且进入推测读缓冲，然而在 RC 下仅当其发送是在前面的分配完成之前完成时，读是推测的（必须监测缓存行为）。

用来增加重叠的最后一种优化（即使是在动态调度的处理器中）是一个独立的写缓冲。写并不在重排缓冲的顶端一直等到写完成并占据重排缓冲，而是当它到达重排缓冲的顶端时移入写缓冲。写缓冲可允许它们对扩展存储结构是可见的，按照同一模型跟踪它们的完成对于松散模型（如 RC 和 PC）来说，写缓冲的用途是很明显的，其中读可以对写旁路。这样，读就可以到达重排缓冲的顶端并更快地引退。在 SC 下，我们可以将写操作放入写缓冲里，但是读操作当到达重排缓冲的顶端时就会停滞一直到写完成。因此，写能看到的许多时延可能不会被将要到达重排缓冲顶端的下一个读看到。

872

4．性能影响

模拟研究已经检验了在不同的同一性模型下硬件预取、推测读和写缓冲技术能够隐藏时延的程度。对 SC 模型的一项研究发现，读时延的关键部分事实上可用具有推测执行的动态调度处理器隐藏，被隐藏时延数量随着重排缓冲的大小增大而增加（大小为 64 ~ 128 项）（Gharachorloo，Gupta，and Hennessy 1992）。一项更详细的研究在先进的多发射动态调度处理器（Pai et al. 1996）上对 SC 和 RC 进行了比较，同时也检验了每种模型分别在硬件预取和推测读时的效益。当一级缓存采用写穿透而不是回写时，即使是在一级缓存中命中，写也需要较长的时间；该研究检验了一级缓存的两种类型，其中假定采用回写的二级缓存。这种研究只是初步的，因为其用了很小的问题和按规模减小的缓存。这些模拟并没有采用高级编译技术来调度指令，这些调度可通过动态调度和推测执行来获得性能。（例如，紧跟在存储操作后面放置更多的无关指令，这只需要一个小的重排缓冲就足够了）然而，这些结果显示了机制与模型间交互的重要性。

最令人感兴趣的问题是使用先进的动态调度处理器在软硬件接口方面 RC 是否比 SC 能得到更多的性能。如果不能，则放松同一性模型的编程负载还不能用这些处理器裁定。（放松模型在支持编译优化的编程接口方面是很重要的，但如果只需编译进行重排操作则编程负担就会减轻）图 11-17 和图 11-18 显示的对两个程序的第二项研究成果表明，即使与读阻塞处理器的情况相比的差距已非常接近，但是 RC 仍然是有效益的。该图显示了没有任何高级优化时（硬件预取、推测读和写缓冲）的 SC 结果，然后在每个的后面显示了优化的结果其中还显示了处理器同一（PC）和 RC 的结果。PC 与 RC 的情况都假设有写缓冲，图 11-17 和图 11-18 中首先显示了没有该两种优化的情况，然后显示了增加优化的结果。

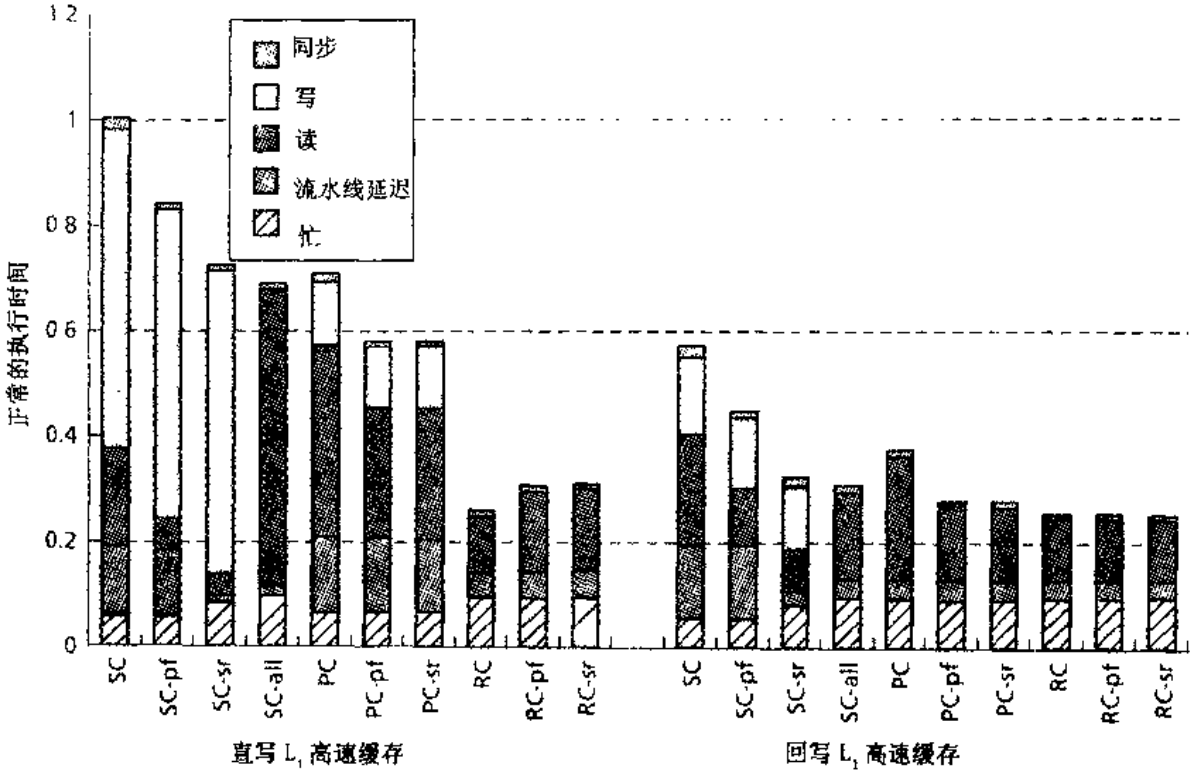


图 11-17 假定具有推测式执行的动态调度处理器中，具有不同的一致性模型中 FFT 内核的性能。左边的 10 个柱条是假定写穿透一级缓存得到的，右边的 10 个柱条是在回写的一级缓存中得到的。二级缓存都是回写的。SC、PC 和 RC 分别是顺序的、处理器的、释放的一致性。对于 SC 有 4 个柱条。第一个柱条不含硬件预取、推测读以及写缓冲；第二个柱条（pf）使用了硬件预取，第三个（sr）也使用了推测读，第四个柱条（all）包含了所有的三种优化。对于 PC 和 RC，总是假设使用写缓冲，因而只有三组柱条（pf 此时意思是硬件预取和写缓冲，sr 包括所有的三种优化。处理器假定是与 MIPSR 10000 类似的（MIPS 技术 1996）。处理器的时钟频率为 300 MHz，而且每周期发射 4 条指令。该处理器使用了 64 项的重排缓冲区，具有 8 项的可归并写缓冲，4 KB 的直接相连的一级缓存，以及 64KB 的 4 路组相联二级缓存。由于使用了少量的数据组，我们选择了较小的缓存，但足可以充分体现时延隐藏的效果。其他更多的参数细节参照（Pai et al. 1996）

当硬件预取和推测读都不使用时，即使使用动态调度处理器，RC 比 SC 也具有重要的优势。这主要是因为 RC 比 SC 能更成功地隐藏写时延，这和简单的处理器一样。这可以允许写更快地引退，使后面的访问在前回的写完成前发送给存储系统并完成。甚至在写穿透缓存中 RC 导致的改进也较大，因为在 SC 中处于重排缓冲顶部的写发送给存储系统，但是必须等待到写操作在二级缓存执行，即使此时在一级缓存中命中。而即使在 SC 下读时延也可得到某种成功，RC 却允许更早发送、完成及读引退。

SC 下硬件预取与推测读的效果比 RC 下大很多，因为在 RC 下允许存储操作以任何方式无序发送和完成。然而，即使有这些优化，SC 与 RC 相比仍然存在一些差距，尤其是在写时延隐藏方面。由于前面讲的原因，写缓冲在 SC 下并不是非常有用。

873 ~ 874

图 11-17 和图 11-18 也证实了，与写返回缓存相比，在 SC 下写穿透的一级缓存比回写高速缓存中写时延的问题显得更重要，然而在 RC 下两种缓存的时延隐藏是一样的。SC 下写穿透高速缓存和回写高速缓存中写时延的区别，在 FFT 中显得比 Radix 要大，因为 FFT 中在

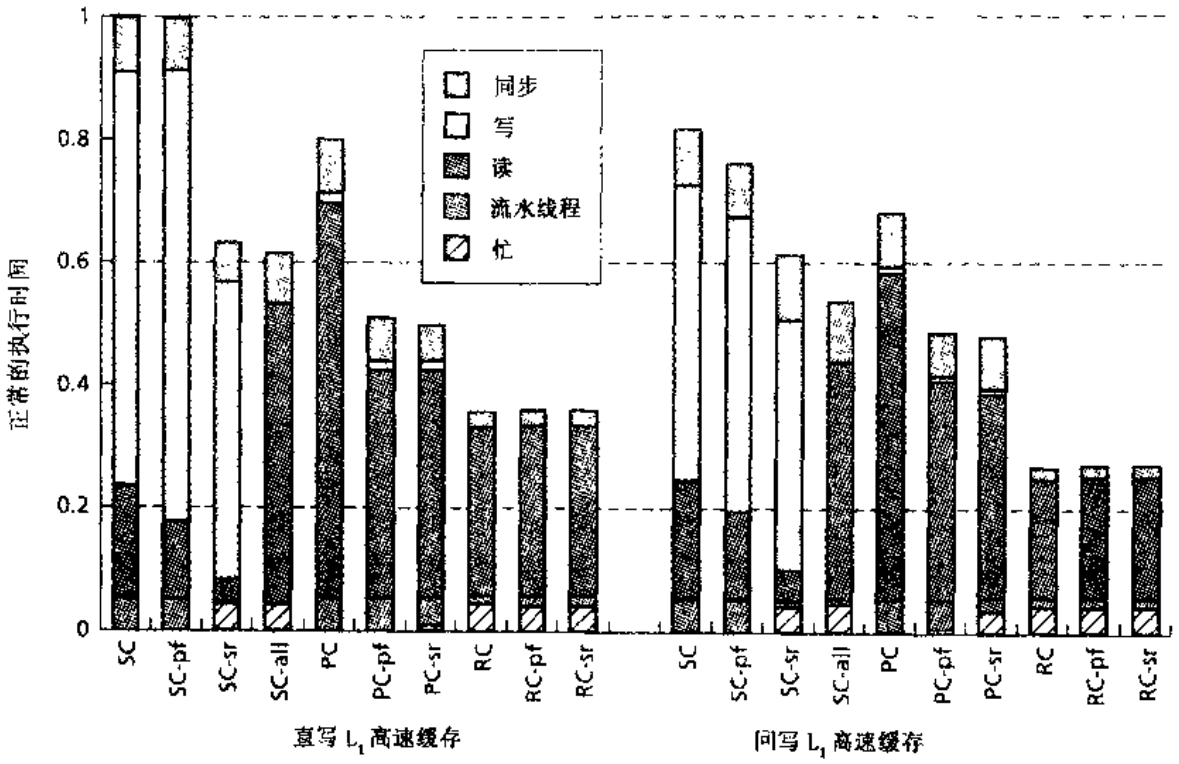


图 11-18　假定具有推测式执行的动态调度处理器中，不同的一致性模型中 Radix 的性能。该系统的假设以及图的组织与图 11-17 相同

当前的问题规模下的写都在局部数据中，而且在回写缓存中命中，而在 Radix 中写的都是非局部数据，而且在两种情况下扑空。总之，在硬件预测帮助下 PC 介于 SC 与 RC 之间，但是推测读带来的帮助不如 SC。

最后，从图 11-17 和图 11-18 中来看，似乎出现下面的异常：不同的设计方案中处理器的繁忙时间是不同的，即使是相同的应用、问题大小与存储系统配置。其原因是对超标量处理器方法的兴趣点不同。由于在同一周期发送多条指令，那么如何决定一个周期是忙还是处于一个特殊类型的停滞呢？这里没有一个非常完美的答案。多数文献中的研究以及以上结果中的确定方法是：只有当所有的指令时间片都发出时，该周期才属于忙时间；否则，该周期就属于第一条（从重排缓冲的头部开始）应当发送而没有发送的指令停滞的类型。由于这种模式依赖于一致性模型和具体实现，因此对于不同的设计方案，其忙时间是不同的。

875

出于感兴趣，我们较详细地检验了硬件预取和推测读的相互影响，尤其是与应用特性的交互。当大量扑空的操作（若不采用重排缓冲预取）在代码中地址相近时，重排缓冲中操作的预取是最成功的，这样这些操作就会一起出现在重排缓冲区内。这样的情况发生在 FFT 的矩阵转换阶段，因此图 11-17 中的预取效果显著。其他程序显示，编译适当调度操作是有帮助的。图 11-18 显示了基数排序应用中使用首次激发推测读（预取的地址直到一次读完成后才能知道）的情况。其中，有关扑空是针对数组项的，该数组项按照直方图索引，但其值也是需要读操作。令人感兴趣的是，两程序中处理器的繁忙时间被推测读减小了。这是因为推测使用读值的能力使得很多本来相关的指令成为可执行的，因此增加了超标量处理器的利用率。另外，即使没有间接的数组读，推测读也有助于减小 FFT 中的读时延。这是因为二级缓

存中的冲突扑空减小了，原因是对于等待完成的缓存块访问的大量合并；许多本来导致二级缓存冲突扑空的访问用推测读来重叠，这些访问因此就可以由跟踪缓存的机制来合并。这说明重要的观察结果有时并不是来自研究的特征。虽然推测读以及通常的推测执行可能由于错误推测而受到损害及导致返转，但是这在研究的程序中很少出现（在代码中采用易于预测而且简明的代码）。这些结果与那些控制流和访问模式很难预测的程序可能是有区别的

11.5.3 小结

在多处理器中通过跨越读和写来包容时延的程度依赖于具体实现以及存储同一性模型的先进性。在缓存一致多处理器中（甚至简单的读阻塞处理器），当采用松散存储同一模型时，包容写时延相对容易。现代的动态调度处理器既能够隐藏读时延又能够隐藏写时延，但只是部分地隐藏。需要隐藏读时延的指令预测窗口（重排缓冲）大小是很重要的，而且随隐藏时延的增大而增大。幸运的是，能隐藏的时延随窗口的增加而增大，而在任何有效时延发生之前并不需要一个很大的临界值。事实上，采用现代处理器中广泛使用的机制，即使在顺序同一，至少中档规模系统中，读时延都能够被合理地隐藏。指令的编译调度有助于处理器更好地隐藏时延。

876

通常，保守的设计选择——如读阻塞或缓存阻塞——较容易保持顺序，但是以性能为代价。例如，在动态调度处理器中，从重排缓冲里延迟写引退直到所有前面的指令都执行完以避免在写上转回（例如，精确的例外），这样可容易地保持顺序同一性，但是也使得写时延在 SC 下更难隐藏。读时延，尤其是写时延在采用放松同一性模型下都可较好地隐藏。在硬件同一性系统中放松同一模型的实现要求，与在现代单处理器系统中所提供的隐藏读和写或甚至与顺序同一所需要的（参看习题 11.9）相比，并没有什么非常过分的需求。保持一给定的同一模型的多数支持是在靠近处理器的缓冲区或缓存中；存储体系的其他层中的处理可以按照要求重新排序。

通过传递操作来隐藏时延的方法有两个重要的缺点。第一个是它很严格地要求非常有效的放松一致模型，尤其是对于简单的静态调度处理器以及动态调度处理器。这将给程序员增加负担以标志同步操作或插入内存栅障，虽然放松的同一模型在很大程度上可允许编译进行许多优化。第二个缺点是，在非动态调度处理器有效隐藏读时延的有效性，以及隐藏多处理器（甚至是动态调度的）中读时延的资源要求，尤其是当时延较大时。在这些情况下，其他方法，如预通信和多线程对于隐藏读或其他形式的时延会更成功，也可能在同一线程中与跨越存储操作相结合。

11.6 共享地址空间中的预通信

预通信支持，尤其是预取已经广泛应用于商业处理器，将来其重要性可能还要增加。为了了解预通信技术，我们首先考虑没有共享数据缓存的共享地址空间，其中所有的数据访问都进入相关的主存。介绍基本概念之后，我们将对缓存一致性共享地址空间中的预取进行考察，包括性能收益和实现问题。

11.6.1 没有共享数据高速缓存的共享地址空间

在没有共享数据缓存的共享地址空间中，接收者发起通信由非本地分配的数据读激发，

而发送者发起通信是由非本地数据写启动。在图 11-1 所示例子代码的基准通信结构中，如果假设数组 A 分配在处理器 P_B 的本地存储器而非 P_A 的本地存储器，则其通信是由发送者发起的。像消息传递中的发送情况一样，我们所能做的最多的预通信是分成两个循环（参看图 11-19 的第一列），在对 f（B[]）的任何计算之前执行所有的对数组 A 的写操作。采用非阻塞写可允许部分写时延与 f（B[]）的计算性重叠；当然在写非阻塞时这样的情况会发生，但问题是在哪里写。效果更为显著的是使 P_A 上的写早些完成，这样可允许 P_B 的读在 while 循环中更快地出现。

877

P_A	P_B
for i←0 *to* n-1 *do*	while flag = 0 {};
compute A[i];	prefetch(A[0]);
write A[i];	*for* i←0 *to* n-2 *do*
end for	prefetch (A[i+1]);
flag ← 1;	read A[i] from prefetch_buffer;
for i←0 *to* n-1 *do*	use A[i];
compute f(B[i]);	compute g(C[i]);
	end for
	read A[n-1] from prefetch_buffer;
	use A[n-1];
	compute g(C[n-1])

图 11-19 共享地址空间预取的例子。该例程假定共享数据不被缓冲，因此预取的数据来自于预取缓冲而非缓存

如果数组 A 分配在 P_A 的本地内存，则其通信是接收者发起的。此时生产者 P_A 的写操作是本地的，而消费者 P_B 的读则是远程的。该情况下的预通信是在实际用之前先预取（prefetching）数组 A 的元素，像在消息传递中需要数据之前发送 a_receives 一样。不同的是其预取不是在本地（如接收），而是导致数据通过网络传输：一个预取请求通过网络发送给远程节点（P_A），该节点的通信辅助部件响应请求并将数据传回。同时，P_B 继续执行其他的工作。有很多方法实现预取，我们以后将看到。一个是发出一个特殊的预取（prefetch）指令，并像消息传递中一样建立一个软件流水线。共享地址空间中的预取代码如图 11-19 所示。软件流水有一个为第一次迭代发射预取操作的开头部分，$n-1$ 次迭代的稳态期间，在该期间内进行当前迭代的计算时为下一次迭代发预取操作、迭代，以及由最后一次迭代组成的结尾部分。其中我们看到，预取指令并不能取代源程序中的实际读操作（装入指令）。另外，如果要完成通过重叠隐藏时延的目标，预取指令本身应是非阻塞的（不应当停滞处理器）。

由于在这种情况下共享数据不被缓冲，预取的数据则放在称为预取缓冲的特殊硬件结构中。当下一个迭代中一个字被实际地装入寄存器时，则从预取缓冲的头而不是从存储器读。如要隐藏的时延大于单个迭代的计算时，我们应当预取多个迭代，而且要求一次能保存多个字。CRAY T3D 多处理器就提供这样的硬件支持，数据在缓冲区中成为有效的顺序与预取发射的顺序一致，以使处理器可按照同样的顺序读。CAEY T3E 使用一组外部的寄存器作为预取缓冲。

即使数据不能足够地提前预取（在数据实际访问时没有到达），预取也是很有好处的。

如实际的访问发现有一个待完成的预取与其访问相联系，它可在其余的时间等待一直到数据取回，这样隐藏了其他的时延。另外，依赖于一次允许待完成的预取个数，预取可以一个接一个地发射以重叠它们的时延。这样就提供了流水数据移动，虽然其速度可能受到发射预取开销的限制。

878

11.6.2 缓存一致的共享地址空间

在缓存一致的共享地址空间中采用预通信更令人感兴趣，这是由于：共享的非局部数据可以通过预通信直接进入处理器的缓存而不需特殊的缓冲，预通信可与缓存一致协议相交互。因此我们以此为背景更详细地讨论这些技术。

现考虑基于无效的缓存一致协议。读扑空要求从它所在的地方取回数据。产生排他读的写取回数据以及所有权（通过通知其宿主，或许作废其他的缓存，接收确认），产生更新的写操作只取回所有权。所有这些“取”都有时延，都是预取的可选对象。我们可以预取数据、所有权或两者兼要。

基于更新的缓存一致协议产生发送者发起的通信，从目的节点的立场看它提供了某种形式的预通信。虽然更新协议不是非常流行，但有选择地对拷贝进行更新可用于发送者发起的预通信，即使在基于无效协议的环境下。一种可能是有选择地插入产生更新的软件指令；另一种是使用混合的更新无效方法。本节的后面将讨论这些技术。

879

1. 预取的概念

两种主要的分类适用于多处理器和单处理器：硬件控制的预取和软件控制的预取。在硬件控制的预取中，没有特殊的指令插入程序代码，而使用特殊的硬件通过观察预测将来的访问，而且按照这种预测进行预取。在软件控制的预取中，预取什么以及何时预取是由编程者或编译来决定的（希望是由编译决定），主要通过分析代码并在代码中适当插入预取指令。本节的后面将讨论硬件控制的及软件控制的平衡。预取也能在块预取（接收者发起的大数据块预取）和块放置（发送者发起）中与块传送结合。

在多处理器中，在硬件和软件控制的预取中要确定提前多少预取，一个关键问题是预取是绑定的还是非绑定的。一个绑定预取是指预取数据的值在预取时被绑定；意思是当进程以后通过常规的读操作（装入指令）读该变量时，将看到该变量被预取时的值；即使该值在预取和实际使用期间已被其他处理器修改而且对于读者的缓存已成为可见。我们讨论的在非缓存一致性情况下的预取（预取进预取缓冲，如图 11-19 所示）是典型的绑定预取，直接预取到处理器中的寄存器也是一样的。一个非绑定预取是指预取指令带来的值在实际操作使用前一直保持修改，像 11.5.2 节讨论的那样。例如，在采用非绑定预取的缓存一致系统中，预取的数据是进入缓存而不是寄存器或预取缓冲（两者一般都不在一致性协议控制下使用），而在预取和实际使用之间被其他处理器的修改将按照协议更新或作废该缓存块。这意味着预取操作可在任何时候发出，而不影响并行程序的语义或可能产生的结果。相反，绑定预取却影响程序语义，因此我们一定要在提前发出时特别小心。例如，如果进程在关键部分增加了共享的计数器，则在关键部分之前（之外）发出对计数器的绑定预取就会不安全，这是由于在预取和获得锁之间另一个进程可能得到了该锁并修改了该数据；然而在该关键部分发送非绑定预取是安全的。由于非绑定预取可尽早产生，因而也具有性能优点。

其他重要的问题是有关确定预取什么数据（分析）以及何时初始化预取（调度）。只有

当访问的地址能提前确定时，我们才认为该预取是可能的。例如，若一个地址恰在该字被访问之前确定，则他就不可能被预取。在用指针实现的非规则以及动态数据结构应用中（如链表和树）这样考虑是非常重要的。

880

预取技术的*覆盖率*是指没有预取时扑空，而在实际访问之前可提前预取的缓存扑空。然而，得到高的*覆盖率*并不是惟一的目标。我们不应当预取处理器不需要的数据，也不能预取已在缓存里的数据。这些预取会消耗开销以及缓存的访问带宽，影响正常的访问而做无用的事。并不是预取的越多就越好。这些不需要的预取称为*非需预取*。要避免非需预取，我们就要分析应用程序中的数据访问的数据局部性以及与缓存大小和组织的相互关系（以便只为了那些目前不在缓存中的数据发送预取）。

最后，适时和幸运在预取中起着重要作用。一个预取也许是可能的而且并非不需要的，但是可能初始化太晚而不能隐藏实际访问的时延。或者初始化太早，在实际访问之前到达高速缓存，很快或者已被替换或无效。因此预取应是有效的：早以足够隐藏时延，晚以使得替换或无效的概率最小。

预取分析的目标是尽量增大覆盖率减小非需预取，而预取调度的目标是效果最大化。以下我们将在一定程度上详细地考虑硬件控制的预取和软件控制的预取，分析它们在这些重要问题上的成功程度。

2. *硬件控制的预取*

硬件控制预取的目标是在硬件上提供执行时访问模式的监测。硬件控制的预取假定在不久的将来访问遵循过去的模式。在这种假定下，包含某数据的缓存块可被预取而进入处理器的缓存，以便在以后的访问可在缓存中命中。以下的讨论假定的是非绑定预取。

其分析和调度都由硬件负责，并没有特殊软件的支持，两者都是在程序执行的过程中动态执行的。分析和调度是十分紧密地耦合的，因为一旦某缓存块认定为预取块，则该块的预取就会初始化：由硬件独立地决定该发送是十分困难的。

预取的硬件应简单而廉价，而且不应当在处理器周期时间的关键路径上。

许多简单的硬件预取方案已经出现。在最基本的层次上讲，长缓存块本身就是硬件预取的一种形式，可很好地开发程序中的空间局部性。对于限制不需要预取的情况，没有分析依据；覆盖率依赖于程序的空间局部性，有效性依赖于访问缓存块中的第一个字与访问其他字所占用的时间。例如，如果一个进程简单地以单位步长遍历一个大矩阵，则长缓存块的预取覆盖率会很好（对于 4 个字的缓存块为 75%），而且没有非需预取——但是效果不会非常大，因为预取发送太晚。对长缓存块的思想进一步扩展则是*整块预测方案*（one-block lookahead，OBL），其中对于第 i 块的访问将激发对第 $i+1$ 块的预取。不同形式的分析和调度可应用于该技术；例如当对第 i 块的访问被检测到时，可预取第 $i+1$ 块，或只有当第 i 块检测到一个缓存扑空，或当第 i 块预取后第一次访问。这种扩展还包括提前预取多个缓存块（如，块 $i+1$，$i+2$ 和 $i+3$）而不只是一个块（Dahlgren、Dubois、and Stenstrom 1995），实际是采用了以下单处理器中*流缓冲*思想：当遇到扑空时，几个后续块预取入一个独立的缓冲，而不是缓存（Jouppi 1990）。当访问多为单步长时，这种技术是很有用的。

881

对于非单位跨距或较大跨距数据访问的情形，检测和预取访问的一种简单方法是：对给定指令（即给定的程序计数器的值），在一个历史表中保留先前访问数据项的地址，该历史表由程序计数器索引。当相同的指令再次发出的时候（例如在循环的下一次迭代中）。如果

在历史表中发现了相同的程序计数值。就能算出跨距等于当前数据地址和历史表中指令对应数据地址的差。预取就可以针对当前地址加上跨距得到的地址发出（Fu and Patel 1991；Fu，Patel，and Janssens 1992）。其历史表更像一个转移历史表，实质上是检测一个指令访问数据的规则步长，而且依此预测将来该指令访问的数据紧跟在该步长的后面。当步长为常数时这种方案可能工作的很好，然而以下我们将讨论的硬件以及软件的其他预取方案也可能在这种情况下很好地工作。

到目前为止，发现的这些方法只是检测简单的规则模式，但当访问不遵循这种模式时，它们并不能防止非需预取。例如，当一个指令连续三次访问不能保持相同的步长时，以上的方案将预取无用的数据。这种非需预取造成的拥挤会导致有用的常规访问的竞争，从而损害性能。

在更高级的硬件预取方案中，历史表不仅保存指令最后一次访问的数据地址，而且还保存该指令访问的前两个访问地址间的步长（Baer and Chen 1991；Chen and Bear 1992）若该指令访问的当前数据地址与前一个地址之间具有相同的步长，则认为是规则步长模式，并可发送预取。否则，认为是模式中的一个断点，不能发出预取操作，这样可减小非需预取。除了地址和步长之外，表项也可包含一些状态位，这些状态位跟踪该指令的最近访问是否是规则的步长模式，曾经处于非规则模式，或正转换成规则模式，或由规则模式转来。基于这些步长匹配，一组简单的规则可用来确定是否要发送预取。若结果是潜在预取，则缓存查找该块是否已存在，若不存在则发送预取。

882

这种模式可改进分析，但还不能在预取调度方面起重要作用。尤其是，在循环中只能提前预取一次迭代。若一次循环迭代中所做的工作小于需要隐藏的时延，这将不足以隐藏时延。调度的目标是得到恰好的预取，即，在需要数据的指令前 l 个周期发送预取，其中 l 为要隐藏的时延，以便于在指令需要数据时预取刚刚到达，而这样的预取才是有效的。这意味着预取应该在实际需要数据的指令执行前 $\lceil l/b \rceil$ 个循环迭代发送，其中 b 是预测的一个循环体迭代的执行时间。

用硬件实现这种调度的一个方法是使用前瞻程序计数器（LA-PC），该计数器可超前于当前实际的 PC l 个周期。LA-PC 用来访问历史数据表并产生预取的而非当前（但和当前 PC 有关）的实际 PC。LA-PC 总是比实际的 PC 超前启动一条指令，但每个周期都在递增，即使是处理器由于缓存扑空而停顿；这样 LA-PC 总是超前实际的 PC。就像实际的 PC 一样，LA-PC 同时也查看转移的历史表，因此转移预测机制也能修改 LA-PC 并保持正确的方向。当检测到与 LA-PC 相关的转移预测错误时，LA-PC 则重置为 PC + 1。其中一个界限寄存器用来控制 LA-PC 超前 PC 的最大距离。当超出该界限或者未完成的访问缓冲区满时（此时不能发出任何预取），LA-PC 就会停顿。

LA-PC 和 PC 每个周期都检查预取的历史表（当然，它们可能访问不同的表项）。对 PC 的查询将按照规则更新命中项的“前一个地址”和状态域，但并不产生预取。每个表项中增加了一个“次数”域，用来跟踪 LA-PC 超前于 PC 的迭代数（遇到该指令的次数）。LA-PC 的查询使得命中项中的该域递增，并按照规则产生一个潜在的预取地址。所产生的地址是该域的值乘以对应项中的步长，加上前一个地址域。当 PC 在预取历史表中查询到该条指令时，“次数”域递减。更详细的资料参考（Chen and Baer 1992）。

这些硬件方案中的预取只能作为一种暗示，因此在扩展的存储系统中真正的缓存扑空的

优先级要高于这些预取。如果预取遇到了缓存扑空（页扑空或者其他的例外），则该预取被简单地注销而并不处理该例外。人们已经提出更加具体的硬件控制的预取机制，如基于不规则补偿的访问预取（Zhang and Torrellas 1995）等。然而，即使是最简单的预取技术也没有找到解决多处理器中的单处理器预取方法。相反的趋势却是为软件控制的方案提供可使用的预取指令。在讨论相对于硬件控制方案的优缺点之前，我们先观察以下软件控制的预取。 883

3. 软件控制的预取

在软件控制的预取中，分析预取的内容以及预取发射的调度时刻一般由软件静态完成。编译（或编程者）在认为适当的点将特殊的预取指令插入代码。像从图 11-19 看到的那样，这也可能在某种程度上重组循环。所需要的硬件支持是提供这些非阻塞指令，允许多个未完成的访问，以及一些对挂起访问的跟踪机制。后两种机制实际上是在基于读写的系统中各种形式的时延包容所要求的。

我们首先从一个处理器在访问流中尽力隐藏时延的观点考虑软件控制的预取，而非由于与其他处理器交互而导致的复杂化。该问题与单处理器中的预取是等价的，只是隐藏的时延范围不同。然后讨论由多处理器导致的复杂情形。

4. 单处理器中的预取

考虑一个简单的循环，像图 11-1 描述的例子。一个自然的方法是在循环中的数组访问中总是提前一个迭代发出一个预取指令。这将导致一个如图 11-19 所示的软件流水线，但有两点不同：数据送到缓存而不是预取缓冲，以后的数据装入是来自数据的地址而不是预取缓冲（例如，read（A［i］）与 use（A［i］））。这能很容易地通过提前多个迭代发出预取，扩展成接近恰好及时预取（just-in-time），这在前面已讨论过（见习题 11.15）。

为了最小化不必要的预取，分析与预测程序中的时间和空间局部性以及将来访问的地址是非常重要的。例如，在块矩阵因子分解中很多次重复使用缓存中当前块的数据，因此预取块中的所有访问是没有意义的。在这种软件情况下，非需预取除了缓存查找带宽与可能的无用通信量外还有另外一个缺点：在代码中导致了无用的预取指令，这也增加了执行开销。预取指令通常处于条件表示中，而且对于复杂的数据结构经常需要额外的指令计算预取的地址，两者都要增加指令开销。

如何用软件很容易地确定预取哪些访问呢？尤其是，编译是否能做到，或者只能留给编程者？答案依赖于程序访问的模式。当以某种规则方式遍历数组时，其可预测度就非常大，例如，一个简单的预测是在循环中何时数组的元素被访问，而且数组的索引是循环嵌套的仿射函数（例如，循环索引的线性组合）。下面的代码是一个例子： 884

```
for i ← 1 to n
  for j ← 1 to n
    sum = sum + A[3i+5j+7];
  end for
end for
```

给定希望隐藏的时延数量，我们能在相关循环中提前多个迭代发出预取。数组访问的数量以及遍历中的空间局部性在该例中是容易预测的，这使得容易进行减小非需访问的局部性分析。分析数据局部性的主要复杂度在于预测由映射冲突导致的缓存扑空。

更难以分析的一类访问是间接数组访问；例如，

```
for i ← 1 to n
  sum = sum + A[index[i]];
end for
```

然而若能容易的预测 i 的值，就能预测将访问的索引数组元素；但若不能预测 index [i] 的值，因此也就不能预测将访问的 A 的元素。为了预测对数组 A 的访问，我们必须足够早地预取 index [i]，再利用预取的值确定数组 A 的预取元素。后者要求将新的指令插入代码。对于调度，若我们通常提前预取的迭代数为 k，我们必须提前 $2k$ 个迭代预取 index [i]，以便在需要 A [index [i]] 时可提前 k 个迭代得到值，在这点可以使用 index [i] 值来预取 A [index [i]]。在这些情况下分析空间和时间的局部性比预测地址还困难。由于我们不能提前知道访问 A 的空间关系，甚至不知道将访问 A 的多少个地址（索引数组中的不同的项可能有相同的值）。我们的选择只能是预取所有对数组 A [index [i]] 的访问，或根本没有预取，或在执行时得到访问模式轮廓信息以做决定，或利用高层编程知识。

目前的编译技术在前述的限制下已经可以很好地处理前面的循环中的数组访问类型。局部性分析（Wolf and Lam 1991）首先用来在给定的缓存中预测数组访问何时会扑空（通常在一级缓存中）。这导致了一次预取预测，该预测可认为是为给定的迭代发出预取的条件式。于是利用基于时延的调度来确定提前多少个迭代发出预取。由于编译不能确定在存储系统的哪一层发生扑空，只能保守地假定最坏的时延情况。

885

预测冲突扑空是非常困难的。基于全相连缓存的局部性分析可能认为某块应当仍然在高速缓存里而不应当发出预取；但是该块可能由于冲突而被替换，此时预取就会带来好处。一个可能的途径是假定将一个有 C 个字节的小的相联缓存当作一个按照一定系数缩小的全相连缓存，但这并不是安全的。进程切换使得多用户编程难以对扑空进行预测，这是由于一个进程可能污染另一个进程的缓存状态，即使局部性分析并不假定缓存块长时间停留在缓存中。即使存在这些问题，有限的实验已经显示出：当大多数缓存扑空是对循环中仿射或者间接数组的访问时，编译产生的预取可取得潜在的成功（Mowry 1994）。这些程序包括规则网格或密集的矩阵（也包括稀疏矩阵）计算（这些计算中包括间接的数组访问，但为了效率经常采用压缩的密集形式）。这些实验都是通过模拟获得的，因为实际的机器刚开始提供软件控制预取所需要的基本硬件支持。

对于编译来说预取中的真正困难是包含指针或链接的数据结构（例如链表和树）。不像数组索引，在遍历这些数据结构时不能参考指针的值；由于链表和树中的地址只有访问该元素时才能知道，因此并不容易预取。在这样的数据结构中预测其局部性也是很困难的。目前，这样的预取必须由程序员完成，这可开发程序及数据结构中的高级语义知识，如图 11-3 所示。基于指针和链接数据结构的编译预取已成为研究的主题，而且别名和指针会有助于该研究（Luk and Mowry 1996）。通常，编译在其他领域（例如程序间的分析）的局限性也会限制其预取分析的有效性。

例 11.3 考虑第 3 章描述的 Barnes-Hut 应用中计算粒子引力的遍历问题。其中，每个粒子对应于一个进程并对其重复遍历，而且连续的粒子使用许多树中的数据，在不同的粒子处理中，这些数据可能停留在缓存中。在这种树遍历代码中如何进行预取呢？讨论多种可能性及其折中方案。

解答： 对 8 叉树采用深度优先遍历。然而，预先假定若需要打开一个树单元，则其 8 个

子单元将被考查。于是，只要确定一个单元被打开，就可以为该单元的所有儿子插入预取（或者我们只要遇到一个单元就可以推测式发送预取，而不再访问儿子的指针）。由于我们希望工作集至少适合于二级缓存中（这是编译很难确定的），因此我们只有在第一次访问时才应当预取单元（即第一次访问的粒子），而不是为了后续的粒子。缓存冲突的可能发生导致不可预测的扑空，对此我们只能采取一点静态的措施。由于预取需要我们做一些工作（确定是否是对单元的第一次访问，是否访问子单元的指针等等），因而预取的开销可能是几条指令。若预取的开销经常多于成功的预取，这会抵消预取的效益。这种策略的另一个问题是当存储时延较长时我们就不能预取得足够早。由于我们一次预取完所有的子单元，多数情况下为第一个（或两个）子单元所做的深度优先工作应当足够隐藏其他节点的时延，但该情况并非如此。其改进的惟一方式是当遇到单元时沿着树进行多层的推测预取，而不参考推测预取的指针以确定预取地址；以期望取到应预取的单元，当我们到达该单元时其数据已在缓存中。由于预取是非绑定的，因而不会违背正确性。在其他使用非结构化的链表应用中，对于编译甚至程序员来说，预取成功就变得更加困难。■ 886

5. 和多处理器一致性协议之间的相互作用

并行程序中预取必须要考虑的另外两个问题是预取通信扑空和所有者预取。两者源于以下的事实：当一个处理器访问一个数据时，其他的处理器也许会正在访问或者修改该数据。

在基于作废的缓存一致性协议多处理器中，数据从一个处理器的缓存中移出（因此发生扑空）不仅因为替换而且因为共享。我们在预取时不应当太早，以至于该数据在使用之前已无效。幸运的是，在非绑定预取中这些只是性能问题而不是正确性问题。

对于编译来说预测将来的作废和做需要的分析是很困难的，因为共享地址空间的显式并行应用程序中的通信很难推断。其中对于编译有利的一个情况是：编译本身并行化程序。但即使如此，动态任务分配和数据伪共享也危及着分析的成功。

程序员具有进程间通信的语义信息，因此在具有作废操作的系统中程序员较容易插入和调度预取。在并行程序中编译具有的一种信息是由显式同步语句传达的。由于同步通常预示着数据共享（例如，在“适当标记的”程序中，一个进程修改数据以及另一个进程使用数据是被标号的同步操作分开的），编译可以假定通信发生，而且当看到同步信号时认为缓存中的数据已经无效。当然，这有点儿保守，可能会导致不必要的预取，尤其是当同步非常频繁而且在同步操作之间很少有数据作废时。当然若同步时能传达数据可能被修改（或者有效确定）的一些信息会更好，然而事情往往并非如此（Wood et al.1993）。

作为另一种加强，由于处理器经常在准备写之前首先以排他的所有权取得一个缓存块，因此在写之前以排他方式预取也是有意义的。若明智的使用，这可带来两方面的好处。首先，这可减小后面写操作的时延，因为此时写不必注销其他的块以及等待得到排他所有权（这已经被预取完成）。这是否会对性能带来影响还依赖于是否写时延已经被其他的方法隐藏，例如放松同一性模型。第二种好处是基于下面的事件：一个进程常常先读一个变量，而后写该变量。这种情况下一个单个的所有权预取（即使在读以前）会隐藏读和写时延。而且使得业务量减半，如图 11-20 所示，因此减少了冲突和带宽要求，进而改进了其他访问的性能。文献（Mowry 1994）定量地讨论了排他模式预取的好处。 887

6. 硬件控制的预取与软件控制的预取

分析了硬件控制的以及软件控制的预取的工作过程后，下面考虑一下它们的相对优点。

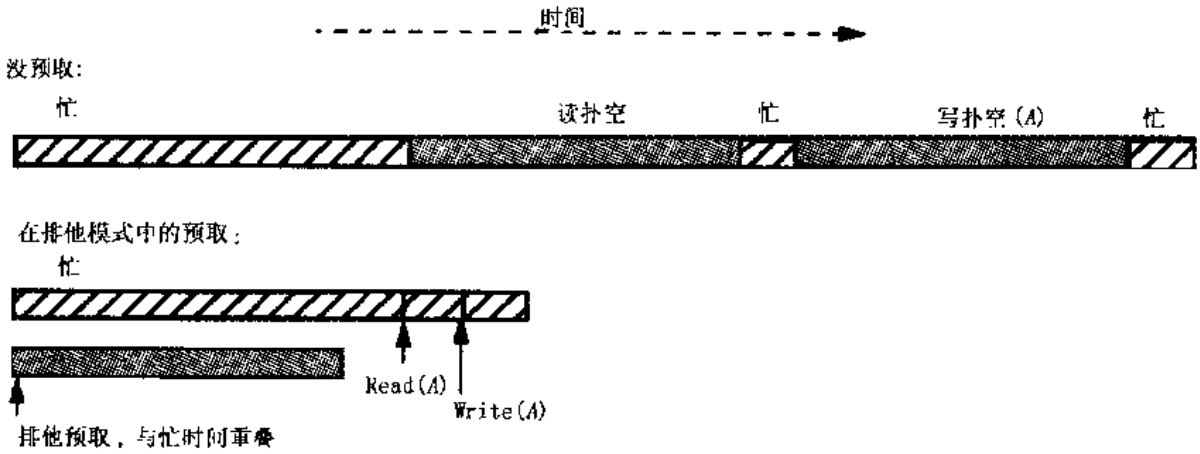


图 11-20 所有权预取所获得的收益。假定开始时 A 的最新拷贝在局部缓存中不存在，但在其他的缓存中存在，然后对其分别进行读写操作。正常的缓存一致性对于读操作将在共享状态时预取相应的块，然后通过通信得到所有权后再进行写操作。具有预取功能时，如果能够识别这种读写模式，我们就可以在读之前发出一个排他的预取，不必要为写再进行任何通信。当写操作发生时，该块已经处于排他状态。读之前在共享模式时进行预取就可隐藏读时延，但并不能隐藏写时延，因为写时仍然会扑空

硬件控制预取的最大优点是：不要求来自于程序员和编译的任何软件支持；不要求重编译代码（当源代码不存在时，这是非常重要的）；不要求指令开销和代码扩展。另一方面，其最显然的缺点是要求硬件支持，其预取算法用硬连线设置在机器中。然而，还必须与覆盖、最小化无需预取、最大化效果等做许多权衡。以下我们集中在编译产生而非编程者产生的软件预取进行总的权衡。

888

- 覆盖率。在分析预取什么的问题上，硬件和软件所采取的策略非常不同。软件策略可检测代码中的所有访问，但只能具有静态信息；然而硬件可观察访问模式的窗口，并基于当前的模式预测将来的访问。软件策略具有覆盖复杂访问模式的潜力，但其分析具有局限性；而硬件则局限于保持复杂历史纪录的开销以及必要技术的精确性（如转移预测）。与硬件不同，编译（甚至程序员）不能对动态信息做出反应，例如由于非预测的缓存冲突造成的替换。在预取更多类型的访问模式上，两种方法的覆盖率正在增加（Zang and Torrellas 1995；Luk and Mowry 1996）；然而硬件方面的开销很高。用执行时的信息反馈改进软件预取的有效覆盖是可能的，但在该方向上并没有更多进展。
- 减少非需预取。硬件预取为增加的覆盖所驱动，但不致力于局部性分析以减小非需预取。它因此可能浪费缓存访问带宽以及互连带宽，甚至替换掉有用的数据。尤其是在基于总线的机器中，在预取上浪费过多的互连带宽至少会潜在地使总线饱和，进而降低而不是增加性能（Tullsen and Eggers 1993）。
- 最大化效果。在软件预取中，调度是基于预测的。然而，预取花费多长时间是很难预测的，例如，存储结构中的哪一层会满足数据，以及会遇到多少冲突。理论上，硬件可在执行时采取调度，它可按照需要对 PC 进行提前预测。然而，由于转移预测而导致隐藏长时延会变得非常困难，每次的错误预测都会导致预测的 PC 重新设置，直到再对 PC 进行足够的提前预测才会提高预测效果。因而，软件和硬件策略在有效性或“恰好及时”（just-in-time）预测上都有潜在的问题。

在动态调度的微处理器中，硬件预取用来为等待在重排缓冲中而还没有发送的指令预取数据。然而，在这种情况下硬件并不检测模式以及分析预取什么。这种严格的硬件预取模式在多处理器中变得很普遍，然而目前的研究显示，对更通用的基于非单位步长访问的硬件分析与预取所需的在片支持是没有价值的。另一方面，微处理器正逐渐提供软件使用的指令预取（甚至在单处理器系统中）。有关预取的编译技术也在取得进展。通常，软件预取将数据传入第一层缓存而不是预取缓冲里。这种以及其他的预取策略问题在习题 11.19 中讨论。 889

7. 发送者发起的预通信

除了基于更新的协议（支持显式方式）之外，软件控制的“更新”、“传递”或“生产者预取”指令已经出现。一个例子是出现在 Kendall Square 研究中的 KSR1 处理器的“后存储”指令，该指令是将整个缓存块的内容存入目前包含该块（大概是旧的）拷贝的缓存中。插入这些更新指令的一个合理位置是在释放同步操作前对共享数据的最后一个写操作，因为该数据是可能被消费者所需要的。更新的目的节点是目录项的共享者（与更新协议一样），并遵循假定：过去的共享模式可用来很好地预测将来行为。（另外，目的节点也可用软件指令来指定，或者数据可仅仅放进本地的主存缓存，而不是其他缓存，即直写而非更新可隐藏来自目的节点的一部分而不是全部的时延。）这些基于软件的更新技术具有与硬件更新协议相同的问题，只是程度较小：因为并不是每个写都产生总线事务。与更新协议一样，竞争的多混合策略也是可能的（Ohara 1996；Grahn and Stenstrom 1996）。

与预取相比，软件控制的发送者发起通信具有以下优点：通信正好发生在数据产生时。另外与基于无效的策略相比，减少了重复生产者－消费者模式的通信量。然而，它也有多种缺点。第一，数据通信太早，使用前有可能已被替换出缓存，尤其是数据放在本地主缓存时。另外，这种策略的预通信只适于进行通信（一致）扑空，而不是容量或冲突扑空。再者，尽管消费者知道它将取什么数据而且能为该数据发送预取，但是如果过去的共享模式不能很好地预测未来，生产者就会传递不需要的数据，或者多次传递重复的数据。进而，预取检测缓存，当在缓存中发现所需数据时则结束，这样就减少了不需要的网络传输；软件更新或传递并不进行这样的检测，可能增加传输量和竞争，即使由于不需要多个协议事务就可将数据存放在正确的位置可以减少传输。最后一点，接收者不再控制接收到多少个预通信信号，这可能导致缓冲溢出。到目前模拟结果的一点发现，对于多数应用来说预取策略比传递或更新策略好（Ohara 1996），即如果两者都使用时可以互补（Abdel-Shafi et al.1997）。

预取和软件更新策略都可以对其能力进行扩展，以传送大的数据块（例如，多个缓存块、整个目标或随机定义的地址区域）而非一个单个的缓存块。这称为块预取与块存放机制（块存放与 11.5 节讨论的块传输是有区别的，其中数据存放在缓存中而不是主存）。这里的问题与那些预取和软件更新指令中的是类似的，除了大小不同以外。例如，预取或传递一个大的数据块给主缓存可能并不是一个好主意。 890

11.6.3 性能收益

多数的预取性能结果都是通过目前为止的模拟得到的。为了说明其潜力，我们考察程序插入软件预取的结果，这些实验来源于书（Woo、Singh，and Hennessy 1994）中的一些应用例程。我们使用程序插入的预取是因为它们比现行的最好的编译算法更具有挑战。我们也将分析目前水平的编译算法结果。

1. 单发射，静态调度处理器中的收益

我们首先观察 11.4.3 节中的程序和平台预取是怎样工作的。为了便于对块传送进行比较，该实验仅仅集中于远程访问的预取（导致通信的缓存扑空）。图 11-21a 显示，对于可预测访问模式的程序或者像 FFT 这样具有很好的空间局部性的程序来说，预取远程数据对性能是有潜在帮助的。与块传送的情况一样，大缓存块的收益比小缓存块的收益小，因为大缓存块本身已经得到了相当意义的预取。图 11-21b 直接对块预取情况的性能进行了比较，结果显示该程序下的结果非常类似。只要假定一次可允许足够的待完成的块传送数，预取能够发挥块传送的许多优点，甚至一些激进的的块传送。图 11-22 显示了对于 Ocean 应用的相同结

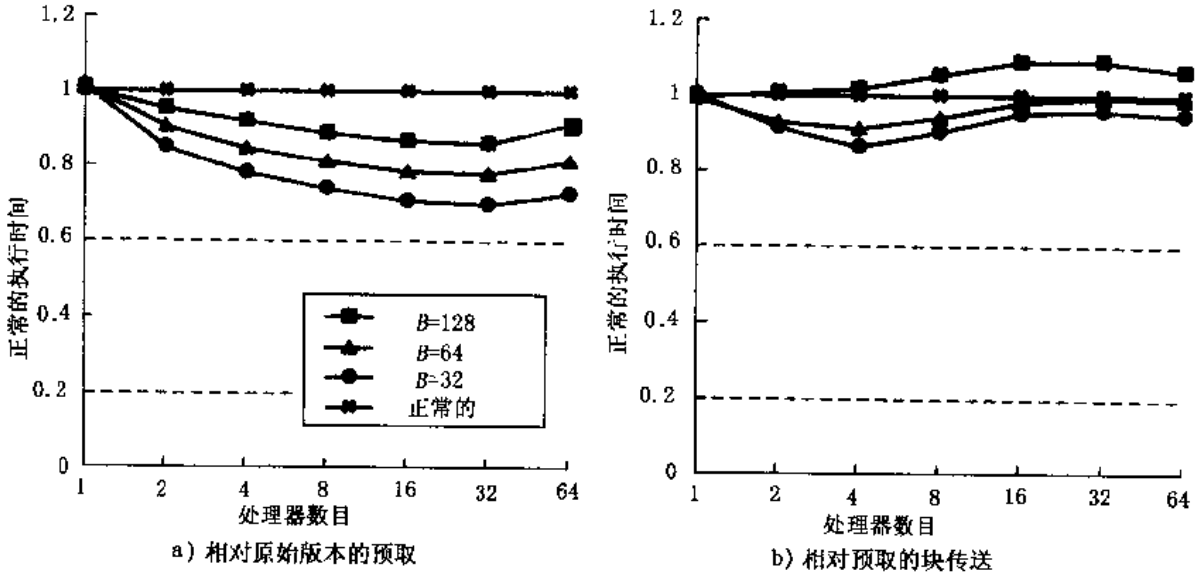


图 11-21　快速傅立叶变换中预取远程数据得到的性能收益。a）显示了预取版本相对于原始版本的性能。该图恰可以用图 11-13 来解释：其中每条曲线显示了在一定处理器数相同缓存块大小（B）的情况下预取版本与非预取版本的执行时间。b）显示了块传送（但没有预取，前面已经讨论过）版本的相对于预取版本（但没有块传送）而不是原始版本的性能。这是我们可以比较块传送预取远程数据的性能。该预取实验总共可允许一个处理器同时具有 16 个待完成的存储操作（包括预取）

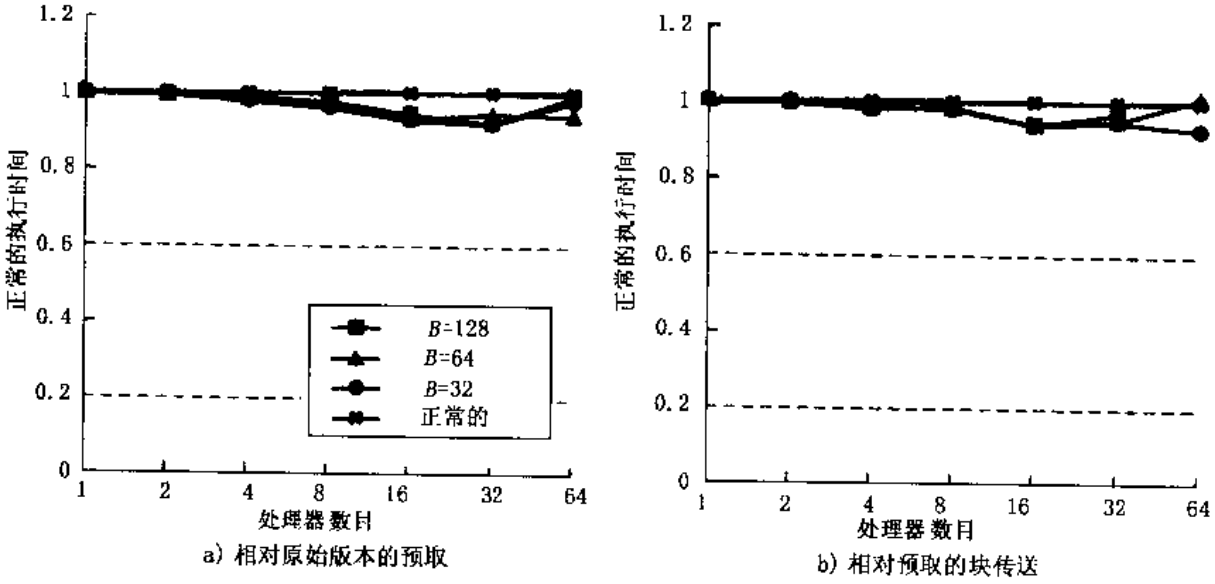


图 11-22　Ocean 例程中预取远程数据的性能收益

果。与块传送一样，预取在这里并没有多少帮助，因为通信的时间很少，而且因为并不是所有的预取数据都是有效的（面向列的划分使得通信的数据具有很差的空间局部性）。

预取在局部访问中通常更能获得成功。例如，在 Ocean 应用中近邻网格计算的迭代中，对于两次扫描间的栅障，从相邻分区足够早地发送边界元素的预取，使新值正好在需要时到来是非常困难的。然而，一个进程可很容易地为分区中的网格点（不被其他的进程接触）足够早地发送预取。对目前编译算法的研究结果显示，当访问模式的可预测性极强时（Mowry 1994），编译可在密集数据的规则计算中成功运用预取。图 11-23 中的两个应用的结果包括在 16 个处理器上运行的局部和远程的访问中。一般来说，这些情况中的仅有问题是足够提前预取的能力（例如，循环嵌套开始或刚过同步点之后什么时候扑空发生），以及分析和预测冲突扑空的能力。

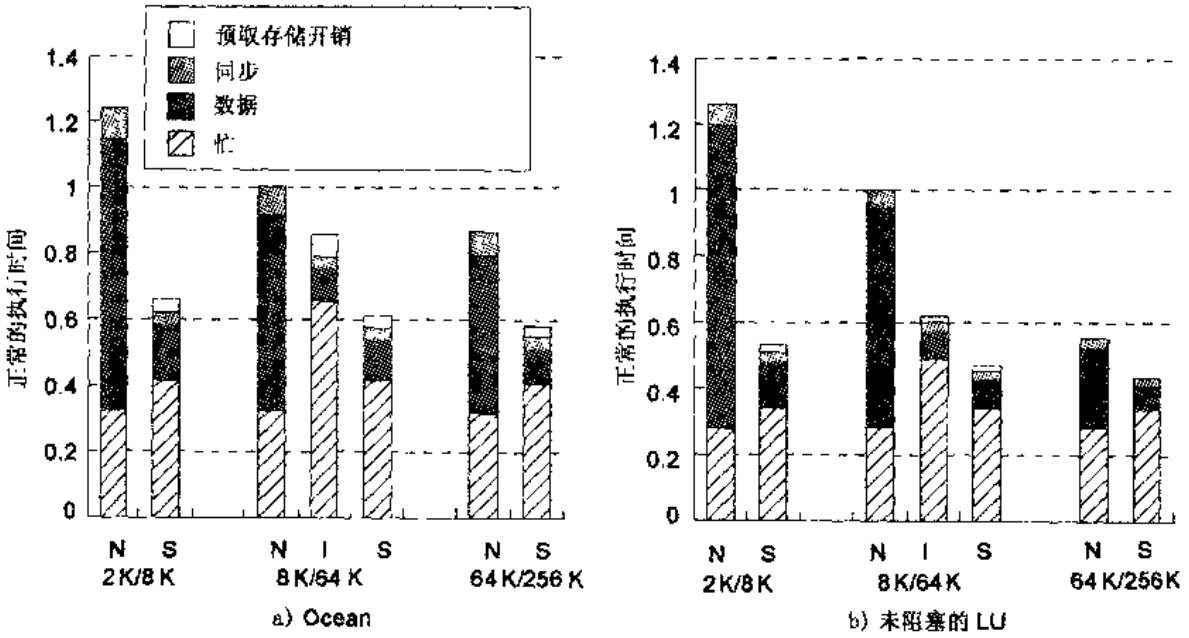


图 11-23　编译产生预取的性能收益。结果显示了两个并行程序在 16 个处理器模拟机上的运行情况。第一个是 Ocean 模拟程序的旧版本，其中将完全的行分成块，因此每块都具有较高的通信计算比率，因此没有按面向列定界的较差的局部性问题。第二个是非阻塞的较低性能密度的 LU 因子分解。其中本地和远程的访问都进行预取，不像图 11-21 与图 11-22 那样。这有 3 组柱条对应于不同的 L_1/L_2 大小结合。每种组合的柱条代表非预取（N）和选择预取（S）的执行时间。对于中间组合（8 K L_1 和 64 K L_2），其结果反映的是预取无区别地（I）发射的情况，但没有局部性分析。所有的执行时间都规格化为 8 K/64 K 缓存大小组合的无预取执行时间。其中处理器、存储器以及通信结构参数的选择与较旧的斯坦福 DASH 多处理器类似，见（Mowry 1994）。相对 DASH 来说，现代系统中的时延大的多。从中我们看到预取有助于性能，缓存大小的选择对预取具有不同的影响。具有预取（尤其是无区别地预取）时的“忙”时间增加是由于：预取指令开销包含在忙时间中。值得注意的是对于阻塞的 LU 因子分解性能收益很小，因为对隐藏数据等待的时间更少；由于阻塞的 LU 因子分解比非阻塞的更为普遍，这也使得方法研究更为重要

通过编译产生预取虽然在使用间接寻址的稀疏数组或矩阵计算中已经取得一些成功，但对更不规则的或基于指针的应用之中还未取得更大的进展。例如，由于例 11.3 讨论的原因，编译算法在 Barnes-Hut 应用的树遍历中并未取得成功。正像前面讨论的，程序员在这些情况下能做得更好，而且执行时搜集的数据有助于标识出产生最多扑空的访问。

891 ~ 894

对于预测完全成功的情况，局部性分析已经建立并在不丢失更多有效覆盖的情况下充分

减小发送的预取个数；因而比非局部性分析时不加选择地预取所有的预测访问（如图 11-24 所示）要更好。通过写停滞而实现的顺序同一性的处理器结构中，建立排他所有权预取可充分地隐藏写时延；但是当写时延已经通过放松存储同一模型隐藏时，这种预取并非很重要。在任何情况下，这种预取都可以大大减小传输量。

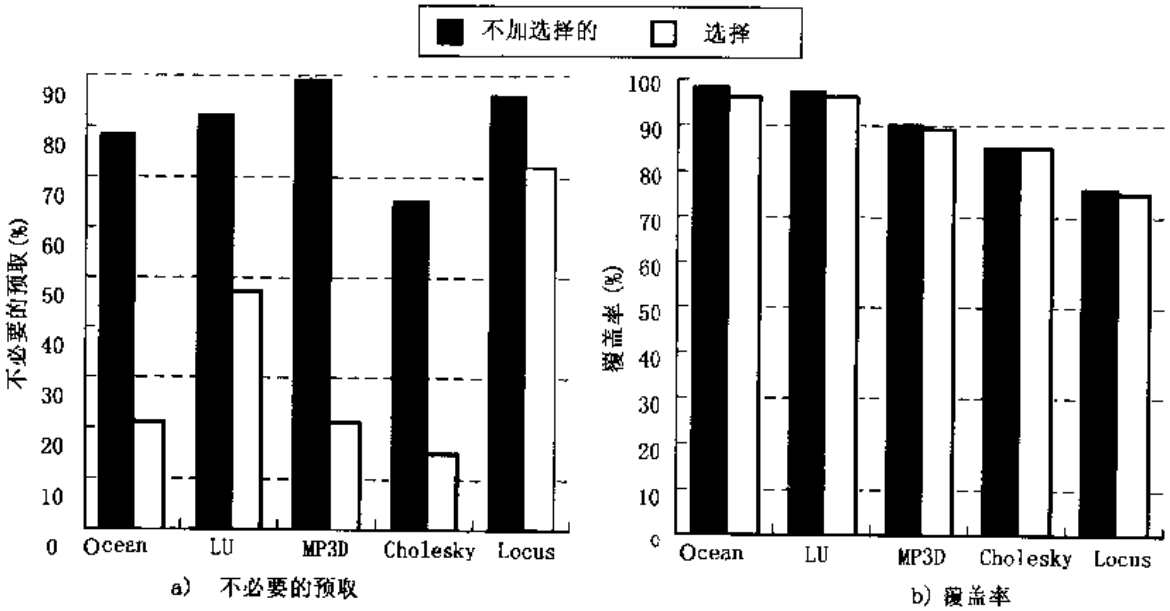


图 11-24　通过进行局部性分析选择预取的收益。不需要预测的部分充分减小了，但整体优势并没有降低。MP3D 是模拟纯净流体力学的应用，Cholesky 是一个稀疏矩阵因子分解的内核程序，LocusRoute（简写为“Locus”）是 VLSI CAD 中的线路由应用。MP3D 和 Cholesky 使用间接数组访问，而 LocusRoute 使用指针实现链表链接，因而很难得到令人满意的性能

最后，定量的评价显示只要缓存合理增大，预取对缓存的干涉效果是可忽略的（Mowry 1994；Chen and Baer 1994）。这些也说明，与硬件预取相比，只要加以挑选，软件预取实际上会导致更少的不必要的传输以及更少的缓存冲突扑空。然而，额外指令以及相关的地址计算开销有时也是很重要的，尤其是那些不规则访问模式的应用。

2. 多发射动态调度处理器中的收益

软件控制预取的效果已经通过模拟在多发射动态调度的处理器上进行了测量（Luk and Mowry 1996；Bennett and Flynn 1996a，1996b），而且也与简单的静态调度处理器进行了比较（Ranganathan et al.1997）。不管动态调度处理器提供的时延包容（包括重排缓冲中的硬件预取操作），软件预取在进一步减小执行时间上显示出有效性。动态调度处理器在数据等待时间上减小的百分率比静态调度的处理器在某种程度上要小。然而，由于数据等待时间在执行时间中占据更大的比率（与减小访存等待时间相比动态调度的超标量处理器可更有效地减小指令处理时间），由于预取而在所有执行时间上减小的百分比，在两种情况下常常是可比的。

在动态调度的超标量处理器中，预取在减少数据等待时间方面的效果并非很好，其原因来自两方面。增长的指令处理速率意味着更少的计算时间来与预取相重叠，而且预取经常结束得太迟。另外，动态调度的处理器会导致存储操作遇到更多的资源竞争，甚至在到达一级缓存之前（例如，挂起的请求表、功能部件、跟踪缓冲等）。这是因为它们允许多个存储操作同时待完成，而且在读扑空时并不停滞。预取会进一步加重这些资源竞争，因而会增加非

预取访问的时延。由于这种时延发生在一级缓存之前，而不能用预取有效地隐藏。这类资源冲突会导致下列问题——简单地提早发出预取不会总能解决以后的预取问题：不仅前面的预取常常浪费，而且这些预取将会很长时间占用处理器资源，因为多个待完成的预取可同时存在。文献（Ranganathan et al.1997）的研究表明，通过改变预取发送的提前时间并不能大大地改进性能。与单发射静态调度的处理器相比，从预取的观点看，多发射动态调度的超标量处理器的一个优点是指令预取的开销往往较小，因为在超标量处理器中预取指令经常占用空的时间片，这样可与其他指令相重叠。

3. 与放松存储同一性的比较

将预取与放松同一模型的比较研究发现，在读阻塞的静态调度处理器中这两种技术是非常互补的（Gupta et al. 1991）。放松模型的目的是减小写停滞时间，而不过多地处理读停滞时间，而预取有助于减小读停滞时间。即使都采用预取机制，顺序和松散同一之间在性能上的基本区别也是存在的；原因是预取不能像放松同一性那样有效地隐藏写时延。在非阻塞读的动态调度处理器中，放松模型同时有助于减小读停滞时间和写停滞时间。预取也同时有助于两者，然而以下考察是有趣的：是采用顺序同一性且只有预取有助于性能还是只有放松同一模型有助于性能。即使当顺序同一性所有能改进性能的优化都应用（像硬件预取、推测读、写缓冲）时，结果发现在动态调度的处理器中使用无软件预取的放松一致模型比使用软件预取的顺序同一模型会更有优点（Ranganathan et al.1997）。其原因是虽然预取与放松同一性相比可在某种程度上更有助于减少读停滞时延，但几乎与放松同一性一样，对隐藏写时延是不起作用的。

895

11.6.4 小结

作为对缓存一致性共享地址空间中的预通信讨论的总结，目前最通用的方法是在微处理器中提供被软件预取使用的预取指令，该指令可由编译或程序员插入。与之相同的机制可用于单处理器或多处理器系统中。研究发现在数据访问模式相对规则的可预测的应用中，预取可成功地隐藏时延，而且该情况下的编译算法已开发成功。在硬件一致中，预取与块数据传送相比，即使在后者工作得很好的情况下，也能很成功；纵然预取涉及处理器的每一个缓存块访问。（在每个通信的终点开销很大的系统中——例如软件实现的共享地址空间——块传送可能相对来说更成功）然而，预取非规则的计算（尤其那些大量使用指针的计算）还有很长的路要走。由于程序员具有计算中访问模式的信息知识，而这些知识能够得到较早或更好的调度，因而程序员插入的预取仍然胜过编译插入的预取。硬件预取只以非常局限的形式得以通用，例如在动态调度处理器中的重排缓冲里以预取操作的形式。即使硬件预取享有不需应用程序重编译的重要优点，但它并不能在通用预取中用作分析和调度，而且在多处理器中的前景也并不清楚。支持发送者发起的预通信指令也不像预取那样通用。预取的一些实现问题在习题 11.18 中讨论。

11.7 共享地址空间中的多线程技术

硬件支持的多线程也许是隐藏时延最万能的技术。与其他方法相比，在概念上它具有下列优点：

896

- 它不要求特殊的软件分析或支持（除了在并行程序中具有比处理器数更多的显式线

程或进程)。

- 因为动态调用，它可以（像可预测的事件一样）非预测地处理事件，例如缓存冲突和通信扑空。
- 尽管前面的技术目的是隐藏存储访问时延，但它却可以隐藏任何长时延事件，只要事件在执行时被检测到。这样的事件包括同步与指令时延。
- 像预取一样，多线程并不改变存储同一模型，因为它不重排在一个线程中的实际的存储操作。

尽管具有这些潜在的优点，当前多线程在商业系统中还没有成为通用的时延包容技术。这有两个原因：第一，它要求微处理器体系结构发生重大的改变。第二，它在单处理器和桌面系统中的使用价值至今还没有足够的证明，这是主导市场的最大因素。下面我们将会看到这样说的原因。然而，随着时延相对于处理器速度逐渐增大，很多的处理器提供了可扩展为多线程的高级技术，而且与指令级并行相结合开发出新的多线程级技术，这种趋势在将来可能会改变。

我们先从消息传递背景下多线程的简单形式开始，其中来自一个线程中的指令一直执行遇到一个长时延事件为止，此时该线程切换出去，而另一个线程切换进来。一个线程的状态称为一个上下文，于是多线程也称为多上下文处理。这种在上下文切换时要保存和恢复的状态包括处理器寄存器、程序指针、堆栈指针、以及处理器状态字的每个进程要保留的部分(例如，条件码等)。上下文切换的开销可能还包括流水线中指令的冲空，以下我们将看到。如果我们将容许的时延足够大，可在上下文切换时将切换出的上下文用软件包存在内存中，当切换回来时再装入上下文。这就是多线程在消息传递时机器中是如何工作的，因此一个标准的单线程微处理器可在该情形下工作。在硬件支持的共享地址空间中，甚至在单处理器中，我们所要容许的时延并没有那么大。用软件保存和恢复状态的开销太大而使切换变得没有价值，此时我们可能要求硬件支持。我们将稍微定量地检验这种开销和时延的关系。

现在考虑处理器的利用率，即处理器花费在执行有用指令的时间，不包括停滞及其他的开销。一个线程执行指令直到遇见一个长时延事件的时间称为忙时间，花费在线程之间切换的时间称为切换时间。若没有就绪的线程，处理器就会停滞一直到一个线程就绪或者长时延事件完成。不管任何原因，花费在停滞事件上的所有时间称为空闲时间。处理器的利用率可表示为：

897

$$利用率 = \frac{忙时间}{忙时间 + 切换时间 + 空闲时间} \tag{11-2}$$

显然保持小的切换时间是很重要的。即使我们能够通过多线程包容所有的时延而能完全消除空闲时间，利用率和性能也会受到切换时间的限制。

11.7.1 技术和机制

对在给定的周期中从单个线程发射指令的微处理器，按照何时进行线程切换可将硬件支持的多线程分为两类。迄今这种方法在消息传递、在多编程中用来容许磁盘时延，本节中让线程一直执行直到遇到一个长时延事件（例如，一个缓存扑空、一个同步事件或一个长周期指令，如除法）而且换到另一个就绪的线程。这种叫做阻塞的方法，因为一次上下文切换只

有当一个线程因为某种原因阻塞或者停滞时才发生。在共享地址空间系统中，该方法用在 MIT 的 Alewife 研究原型中（Agarwal et al.1995）。其他的硬件支持的主要方法是（不管是否遇到长时延事件）简单的每个处理器周期切换线程，这可在单周期粒度上在多个就绪线程池之间有效地交替使用处理器的资源。当一个线程遇到一个长时延事件时，该线程标志为未就绪而停止执行，直到长时延事件完成而又加入就绪池。这称为交替方法。下面我们较详细地考察该两种方法，观察其量化特征与折中以及它们的量化评价和细节。当在单线程处理器中讨论完两种方法后，将考察多线程与指令级并行性的结合（超标量结构），该方法有望克服传统方式的局限性（参看 11.7.5 节）。

1．阻塞的多线程

阻塞的多线程的硬件支持通常包括多个硬件寄存器堆和多个线程使用的程序计数器。一个活动的线程或上下文是指当前分配给硬拷贝之一的线程。活动线程的数量可能比就绪线程（没有停滞但准备好执行）的数目小，而且也受资源硬拷贝数的限制。我们首先从高层上观察一下容许的时延、线程切换开销以及活动线程之间的关系，其中使用的分析方法与前面分析处理器利用率的方法是一样的（Culler 1994）。 898

假定一个处理器提供 N 个活动的线程（N 路多线程），每个线程重复下列操作：非停滞地执行有用的指令 R 个周期（R 是停滞间的忙时间，称为运行长度）；遇到一个长时延时间而切换到另一个线程。现假设每次要包容的时延长度为 L 个周期，一次线程或上下文切换开销为 C 个周期。给定一组值 R、L、C，则处理器利用率与线程数 N 的关系如图 11-25 所示。其中有两种操作方式：利用率随线程数成线性增长，直到一个临界点，在该点变为饱和。下面分析其原因。

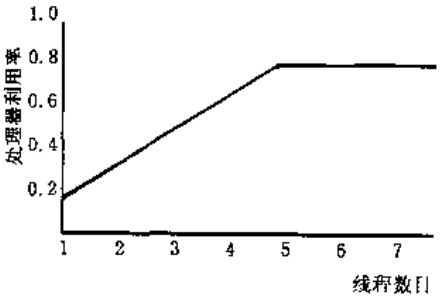


图 11-25　阻塞多线程中处理器相对于线程数的利用率。该图显示了两种操作区段：线性区段和饱和区段。其中假定 $R=40$，$L=200$，$C=10$ 个周期

开始，在一个线程的停滞期间（L）增加线程的数目可使其他的线程做更多有用的工作，而且其时延可继续隐藏。一旦 N 变得足够大，其他的线程都经过一个周期返回（每个都是 R 个周期运行时间，C 周期的切换开销），此时又回到第一个线程，我们可能包容所有的 L 时延。除此之外，设置多个线程并无其他好处，因为所有的时延都已被隐藏。饱和时的 N 值可通过以下式求出：$(N-1)R+NC=L$，或

$$N_{sat}=\frac{R+L}{R+C}$$

除了该点之外，处理器总是或忙于运行一个线程或发生切换开销，因此其利用率为：

$$u_{sat}=\frac{R}{R+C}=\frac{1}{1+\frac{C}{R}} \tag{11-3}$$

如果 N 相对于 L 来说并非足够大，则其他的 $N-1$ 个线程就会在 L 时延经过之前完成。因此处理器在 $R+L$ 周期内做的有用工作为（$N-1$）$\times R$ 或 NR 个周期，其他时间或者是停滞或者是切换，在该线性时间段内的利用率为：

899

$$u_{lin}=\frac{NR}{R+L}=N\times\frac{1}{1+\frac{L}{R}} \tag{11-4}$$

这种分析显然很简单，因为它假定了等长的执行时间 R 并忽略掉了扑空或其他长时延事件的突发性。平均情况分析可能导致我们假设在时延包容非常关键时，处理突发事件需要的线程少于实际的需要。（更精确的模型参看［Culler 1994］）然而这样的分析对于一些关键点来说是足够的。对于任意的线程数 U_{sat}，由于能得到的最好的利用率都将随着切换开销 C 的增加而减小，因而减小切换开销是非常重要的。切换开销的大小也影响着隐藏时延的类型；例如，除非切换开销非常小，否则流水线时延或二级缓存的扑空时延是很难隐藏的。

若我们在硬件上支持多个活动的线程（包括独立的寄存器堆、PC 等等），切换开销就可以变得很小。这样在发生上下文切换时不用软件就可以简单地从一个硬件状态切换到另一个状态，这也是大多数硬件多线程处理器所采用的方法。典型地，一个大的寄存器堆可以静态地分成多个寄存器帧，以此作为对活动线程的支持（称为*段式寄存器堆*），或者像缓存一样对寄存器堆进行动态管理以保存活动线程的寄存器。

虽然硬件支持的复制上下文状态可将活动线程切换减小到一个周期（实现时是改变指针而不是拷贝大的数据结构），但另一种上下文切换开销我们目前还没有讨论。这就是来自指令执行流水线的开销。

当一个长时延事件发生时，我们要将当前的线程切换出去，并假设长时延事件是缓存扑空。缓存扑空只能发生在指令流水线的取数据阶段，在流水线的后面。典型地，假设命中/扑空只有回写阶段检测，而该阶段在流水线的最后。这意味着当我们知道缓存扑空而且应当将当前的线程切换出时，来自该线程的其他指令（潜在的 k 条指令，与流水线的深度相当）已被取出并在流水线中了（如图 11-26 所示）。此时我们将面临三种可能性：1）允许该序列指令执行完，并同时允许从新的线程中取指令；2）在新线程开始取指之前允许这些指令完成；3）排空指令流水线，然后允许从新的线程取指。

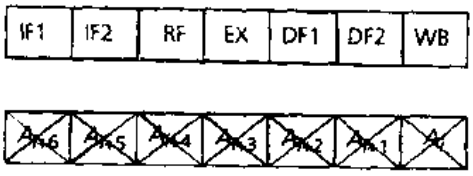


图 11-26　流水线中延后检测扑空的影响。线程 A 是处理器上运行的当前线程。指令 A_i 中的缓存扑空在流水线的第二个取数阶段（*DF2*）之后才检测到（即，在 A_i 指令流经流水线中的回写阶段［*WB*］）。此时，同一线程 A 后面的 6 条指令（从 A_i+1 到 A_i+6）已经进入 7 个阶段流水线的不同阶段（两个周期的取指［IF］，一个周期的寄存器取数［RF］，一个执行［EX］周期，然后是两个周期的取数和回写）。如果当检测到扑空（下面一行打叉的位置）时将所有的指令作废，则至少会浪费 7 个周期的工作

第一种情况实现较为复杂，其原因有二。第一，由于来自于不同线程的指令将同时存在于流水线中，这就要求改变标准的流水线寄存器与相关解决机制（互锁机制）。这样，在流

经流水线的指令必须标志相应的上下文，而且（或者）要设置多个流水线寄存器来区分来自不同上下文的结果。这除了增加面积外，每个流水段的多个流水线寄存器意味着，寄存器必须经过多路复用进入锁存以供下一个流水段用，这可能增加处理器的执行周期时间。因为阻塞模式的部分动机是其设计简单以及尽量不花费设计与修改就可利用商业处理器的能力，因此这可能不是非常适当的选择。这种选择的第二个问题是导致缓存扑空的指令在流水线中，而且由于流水线中来自同一个线程的指令依赖扑空返回的数据而使流水线停滞。 900

第二种选择避免来自多个线程的指令同时存在于流水线，但由于指令相关仍然逃避不了流水线停滞的问题。第三种选择避免了上述的两种问题，而且由于标准的单处理器流水线就具有冲空流水线的能力，其实现起来较简单。尽管阻塞方式会与浪费在切换上的流水线深度的周期个数相当，但阻塞多线程还是受欢迎的选择。

在阻塞方法中如何触发上下文来隐藏不同种类的时延呢？当缓存扑空时，可由硬件支持的扑空检测触发切换。对于同步时延，我们可以简单地在导致长时延的同步事件（或者是所有的同步事件）后插入显式的上下文切换指令。由于同步事件可能由同一处理器上的其他线程来处理，为了防止等待超时而死锁需要显式的切换指令。长时间的流水线停滞也需要在像除法等长时延指令之后插入显式切换指令。最后，流水线的短停滞（如数据失效等）很难通过阻塞方法来隐藏。

总之，阻塞方法相对来说实现开销低（以后将详细讨论），而且具有好的单线程性能（当处理器中只有一个线程在运行时不发生上下文切换，这种模式与标准的单处理器相似）。其缺点是切换开销大：大约等于流水线的深度，即使寄存器或者处理器的其他状态不需要保存或存入存储器。这种开销限制了能隐藏的时延类型以及效率。例 11.4，来自（Laudon, Gupta, and Horowitz 1994），考察了其性能影响。 901

例 11.4 假定有 4 个线程，*A*，*B*，*C* 和 *D*，在一个处理器中执行。这些线程具有以下的行为：

A 发射两条指令，第 2 条指令导致缓存扑空，然后再发射 4 条以上的指令。

B 发射一条指令，然后跟有一个两个周期的流水线相关，接着是两条以上指令，最后的指令导致缓存扑空，紧跟着是两条以上指令。

C 发射 4 条指令，第 4 条指令引起缓存扑空，紧跟着发送 3 条以上的指令。

D 发射 6 条指令，第 6 条指令引起缓存扑空，紧跟着一条以上的指令。

试说明在阻塞多线程执行中流水线的时间片被线程使用和浪费的情况。假定流水段数为 4，因此上下文切换开销为 4 个周期，一次缓存扑空为 10 个周期（为了说明容易在此假定的数较小）。

解答： 图 11-27 显示了解决方案，假设按照循环（round-robin）算法选择线程，并从线程 *A* 开始。当多数的存储时延都被隐藏时，这是以上下文切换开销为代价的。假设在该指令序列的开始流水线是平稳的，我们可以从第一条指令到达 WB 阶段（即，图 11-27 的底部显示多线程执行的第一个周期）开始对周期计数。在多线程执行的 51 个周期中，21 个处于忙周期，两个为流水线停滞周期，没有存储停滞的空闲周期，28 个为上下文切换周期，因而处理器的利用率为（21/51）× 100/100，即 41%，尽管我们假定的缓存扑空损失很低。■

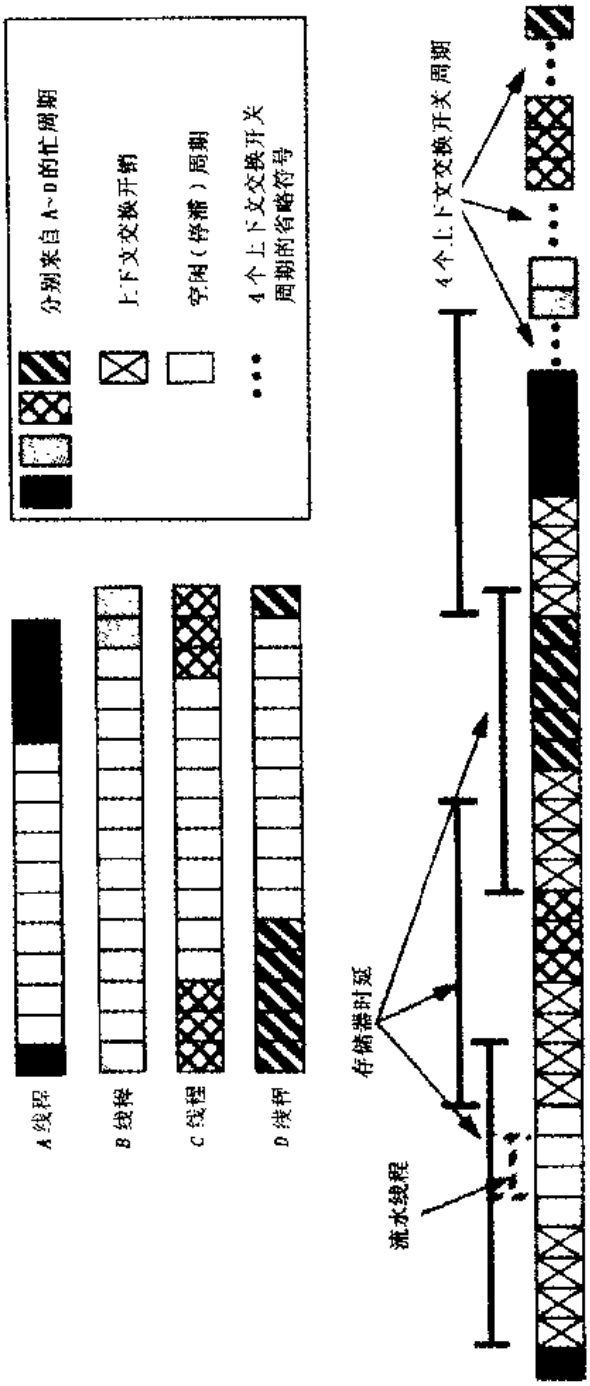


图 11-27 阻塞多线程方法中的时延包容。每个时隙显示的指令是该周期流水线的最后阶段(WB)。图的顶部显示了4个活动的线程在处理器中分别只有一个执行时的情况。例如，第一个线程发射一条指令(该指令的WB周期显示的是线程A的第一个周期)，然后发射第二条指令并遇到缓存扑空(因此，线程A的第二个周期即是该指令的WB阶段，是空的，与后面的几个空位置相同)，然后再发射其他指令。图的底部显示了遇到缓存扑空时处理器在线程间切换的情形。如果线程A的第二条指令没有发生扑空，则将如图底部的第二个周期处于WB状态。然而，由于遇到了扑空，为了完成线程切换，该周期会浪费，而且已经进入流水线的线程A的其他三条指令也会排空。值得注意的是，两个周期的流水线时延不会产生线程切换，因为4个周期的切换开销远远大于该时延

2．交替多线程

在交替的方法中，每个处理器周期从不同的就绪的活动线程（装入一个硬件上下文）中选择一个指令，同时线程切换是每个周期进行一次，而不是仅当遇到长时延事件时才发生。当一个线程发生长时延事件时，该线程就被挂起并从就绪线程池中移出，直到长时延事件完成该线程才标记为就绪的（该线程在硬件状态资源中一直保持活动的状态）。此时使用分段的或复制的寄存器堆来避免保存和恢复寄存器。交替方式的重要优点是没有上下文切换开销。由于上下文切换是每周期进行的，因而不需要检测以激发上下文切换；同时有足够的线程，来自同一线程的指令不会同时处于流水线中，因此没必要挤掉指令。因此，当有足够的并发线程时，所有的时延都被隐藏且没有切换开销，这样处理器每个周期都做有用的工作图 11-28 显示了一个理想情况的例子，其中我们假定了 6 个活动的线程。交替方式的典型缺点是高的硬件开销和复杂度，当然具体的缺点和其他潜在的缺点还与所用的交替方式有关。

902
~
903

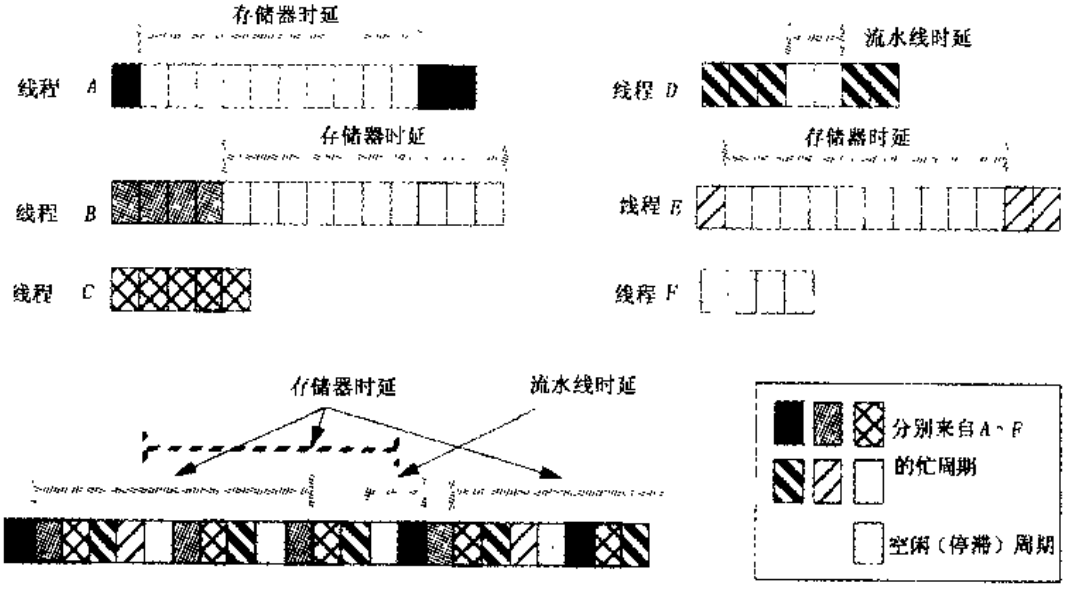


图 11-28　交替多线程中理想设置下的时延包容。图的顶部显示了当每个作为惟一执行的线程时，6 个活动线程在处理器中的执行情况。图的底部显示了处理器在就绪线程间采用循环（round-robin）算法周期切换的情况，其中省略了那些最后一条指令引起处理器停顿但仍然还没有解决的线程。例如，对于线程 A（实心部分），只有其长时延的访存操作得到满足后才能重新选择，再次进入 round-robin 的循环模式

交替模式经历了一个相当长期的演变。较早的交替方式对同时进入流水线的来自给定线程的指令数量和类型具有严格的要求，这减少了对硬件互锁以及冲空指令的需要。这虽然简化了处理器设计，但对单线程的性能产生了严重影响。最近出现的交替模式已大大减少了这些限制，以下将看到这一点。实际应用中的交替模式的另一个显著特点是在时延包容前是否用缓存减小时延。至今使用交替方式的机器根本不用缓存，而只是靠多线程时延包容（Smith 1981；Alverson et al.1990）。更近的研究建议提倡交替模式以及基于缓存的全面流水线互锁（Laudon，Gupta，and Horowitz 1994）（阻塞的多线程系统由于要尽量保持单线程的有效执行而使用缓存）。下面我们介绍交替模式发展的几个阶段。

904

3．基本的交替模式

第一个交替的多线程模式应用在 Denelcor HEP 异种部件处理器多处理器中，该机器在

1978至1985年期间开发（Smith 1981）。每个处理器具有128个活动上下文、64个用户级、64个特权级（尽管实际中只有50个用户可见）。由于机器没有缓存，而且每个存储访问都有存储时延，因而即使存储时延很小（无冲突时20～40个周期）也需要多个活动上下文。（多模块的存储器都设置在多级互连网的另一端，同时处理器对其“局部”模块还有另外的直接连线）其中128个活动上下文是寄存器堆以及其他关键状态的128个复制。HEP中的流水线的深度为8，支持对非存储指令的互锁，但不允许一个以上的存储、转移、除法操作同时存在于流水线。这就意味着每个处理器允许多个活动线程有效地使用流水线，甚至没有存储以及其他的停滞。增加缓存以及进一步隐藏存储时延则需要更多的线程。这也意味着程序中的显式并行度必须远远大于处理器数，从而限制了应用程序的运行效果。

4．更好地使用流水线

只允许来自于一个线程的单个存储操作的缺点是，单线程的性能差和需要大量的线程。因此HEP的派生系统也受同样的限制。这样的系统包括Horizon（Kuehn and Smith 1988）和Tera（Alverson et al.1990）多处理器。它们仍然没有设置缓存，完全依靠多存储访问的时延包容。

第一个设计是Horizon，实际中从来没有制造过。不像HEP，它却允许来源于同一个线程的多个存储操作同时存在于流水线上。然而，Horizon却没有提供流水线互锁，即使非存储操作，同时将相关分析留给了编译。其思路非常简单。基于编译分析，每个指令有3位的“预测”标志域，该域指示出本指令后的多少个周期才与该指令无关。假设某指令的预测域为5，则意味着该指令的后第5条指令（存储指令或其他）与本指令无关，因而即使没有硬件互锁解决相关也可与当前指令同处于流水线中。因此，当一条指令遇到长时延事件时，此线程等到5个指令后才变成未就绪，而不是立刻改变。预测域的值最大为7：机器可禁止来自于同一线程的7条以上指令，直到当前指令离开流水线。所提供预测最小位数受以下因素的影响：指令字所贡献的位数，编译使用更深度预测的能力，尤其是对寄存器的压力（每个指令的3条结果×预测指令数）；更深度的预测以及更大的寄存器压力可能会给指令调度带来反作用。

905

在Horizon中每个“指令”或周期可包括三个操作：其中一个可能是存储操作。这就意味着要得到最大的预测深度，需要找到21条无关的操作，因此依赖于编译的高级指令调度技术是很重要的。对于特殊的应用程序，很可能每条指令或每两条指令发送一个存储操作。由于最大预测深度大于存储操作间的平均距离，因而允许多个存储操作同时进入流水线是很重要的。然而当没有缓存时，每个存储操作都是长时延事件，此时就需要更多的就绪线程来隐藏时延。尤其是单线程的性能（非多线程化程序的性能）在没有缓存时小数量的预测并没有多大的帮助，而且受到很大的限制。

由Tera计算机公司研制的Tera体系结构是没有缓存的交替多线程系列中最新款式的。Tera处理线程依赖的方式与Horizon和HEP不同，而是使用了混合的方式：提供了非存储操作的流水线互锁（像HEP）以及类似Horizon的存储指令的预测。

Tera机器将操作分为存储操作、算术（或逻辑）操作以及控制操作（例如转移）。非通用的定制处理器每个指令可发送3个操作（很像Horizon）：或者每种类型中的一个，或者两个算术操作一个存储操作。算术与控制操作进入一个流水线，此时由硬件互锁允许来自一个线程的多个操作。其流水线的深度很大，即使指令之间并没有相关，同一个线程的连续的两条

指令之间也要具有一个可调整的最小发送时延（大约 16 个周期，流水线深度大于该值）。因此，虽然允许来自同一线程的多条指令可同时进入流水线，但即使隐藏指令时延也需要多个交替的线程（大约 16 个）。即使没有存储操作，单线程至少需要 16 个周期才能完成一条指令。

存储操作则显示出一个较大的问题。虽然 Tera 机器使用了较强的存储和网络技术，但在没有竞争的情况下，存储操作的平均时延也需要 70 个周期（一个处理器周期为 2.8 ns）。因此 Tera 使用编译技术为存储操作产生预测域，其语义与 Horizon 稍有不同。每个包括存储操作的指令（称为存储指令）具有 3 个比特的预测域，该域标示出紧跟在后面的多少条指令（存储器或其他）与本存储操作无关。其后续指令不必彼此无关，只是与被标志的指令无关即可。该线程可在导致未就绪前可越过该存储指令发送那些后续指令。这些预测的变化使得在预测度较大时编译较容易地调度指令，而且也减轻了寄存器的压力。具体说明见例 11.5 906

例 11.5 假设 Tera 中连续的两条指令间的最小发送时延是 16，要隐藏的平均存储时延是 70 周期。在完全隐藏时延的最好情况下需要的最少线程数？在该情况下每个存储器需要多大的预测？

解答： 最小发送时延 16 意味着，在不考虑存储器的情况下需要 16 个线程来充满流水线。由于每个线程中几乎每条指令一个存储操作，若每个线程在遇到存储操作而未就绪前可发送 4 条无关的指令，16 个线程就可以隐藏 70 个周期的存储时延，即预测深度为 4 就可以。由于实际的时延经常会高于 70 个周期，因而提供需要较深的预测（最多为 7，3 位）。当然更长的时延需要更大的预测值和高级编译（或更多的线程）。因而，在非存储流水线中虽然会减小最小发送时延，但我们还是希望减小线程数。■

从前述可以认为支持几十个活动线程就足够了，因而 Tera（与 HEP 类似）在硬件上支持了 128 个活动的线程。在 Tera 中，将所有的处理器状态（程序计数器、处理器状态字与寄存器）复制了 128 次，结果导致了 4 096 个 64 位通用寄存器（每个线程 32 个），1 024 个转移目标寄存器（每个线程 8 个）。对大量线程的支持出于以下多方面的原因，同时反映出机器对时延隐藏（而不是减少时延）的依靠。第一，有些指令可能根本没有大的预测值，特别是读指令；当一个指令含有 3 个操作时，若预测值为 4 个指令，则意味着在读指令与依赖其的指令之间必须有 12 个无关的操作。第二，当多数的存储访问进入网络后，可能造成争用，此时要隐藏的时延可能大于 70 个周期。第三，对大量线程支持的目标不仅隐藏指令和存储时延，而且也容许同步等待时间。这些同步时间是由负载不平衡以及关键资源的冲突造成的，而且其时延往往比数据访问时延大得多。

与其他时延隐藏技术相比，Tera 系统以及以前系统的设计者在设计多处理器时采纳了更为激进的观点。不像我们目前使用的方法——首先减少时延，再隐藏其余的——这种方法根本不特别注重减少时延；这种处理器只是集中于通过线程交替隐藏细粒度的访问时延。但值得讨论的是一般商业处理器依赖于缓存以及很少量的硬件上下文，因而并不适合于普通的多机处理。因为现代会聚体系结构中的长时延以及存储的物理分布，选用相对非时延包容的商用微处理器作为组建模块意味着必须非常注重缓存的数据局部性以及主存级的数据分布。这 907
使得以性能为目的的编程更为复杂，因为除非最简单的情况，编译还不能成功地自动管理局部性；因此这种方法的潜能还不是很清楚。Tera 方法认为使得多处理技术真正通用的惟一途径是置局部性管理于软件之外，并建立在处理器对时延包容高度支持的体系结构之上。若存在足够的额外线程而且该技术取得成功，则在程序员的眼里机器才确实是一个 PRAM（即，

忽略数据访问开销），这样程序员就可以将精力集中在并发性而不是时延管理上。当然，这种方法会牺牲商业处理器和缓存的作用，并直接面对设计非标准高性能处理器及相关系统软件的巨大耗费。这也可能导致单线程性能降低，这意味着甚至单处理器的应用必须充分多线程化（或系统充分并行化）以获得高的性能。

5. 对单线程流水线的充分支持以及使用缓存

到目前为止讨论的交替方法与阻塞多线程方法有很大的不同，两者都具有局限性。Tera的交替方式对基本的HEP方式进行了改进，但高的利用率仍然依赖多个并发线程。不用缓存意味着每个存储操作都是长时延操作。这除了会增加需要的线程以及隐藏时延的难度外，每次存储访问都要消耗存储以及通信带宽，因而机器也要提供巨大的带宽。

另一方面，阻塞的多线程方式对商业的处理器修改较少。它使用缓存而且在缓存命中时并不发生线程切换，因而保持了高的单线程性能，而且需要的线程数较少。然而它的切换开销大，不能隐藏短时延。同时，高的切换开销也使得该方法不能包容单处理器中并不是很大的时延，因而，很难确定哪种方式适合单处理器，哪种方式适合高端市场。

在交替方式中同时支持缓存以及完全的单线程流水线也是可能的，这就要求使用小数量的线程隐藏存储器时延、比阻塞方式低的切换开销、以及对单处理器的更好支持。该方面的详细研究参看（Laudon，Gupta，and Horowitz 1994）。从HEP和Tera方法中我们可以看出，交替的方法采用以下思想：保持一组活动线程，每个活动线程具有自己的寄存器和状态字，每个周期允许处理器从一个就绪线程中选择一条指令。这种选择非常简单，像用round-robin方法从就绪线程中选择。一个线程遇到长时延事件时导致本身未就绪，直到该长时延事件完成为止，这与前面讨论的一样。关键的不同点是其流水线是标准的微处理器流水线，而且具有全面的旁路和转发支持，可使得来自同一个线程的指令在连续的周期发送（与阻塞方式相同）；不像Tera那样具有最小的发送时延。在最理想的情况下，一个k-深度的流水线可包含同一线程的k个指令。除此之外，用缓存减小时延意味着很多存储操作并非长时延事件；因此一个线程在大部分时间是就绪的，而且隐藏时延的线程数是很小的。例如，若一个线程每30个周期发生一次缓存扑空，每次扑空开销为120个周期，于是保持处理器的完全利用只需要5个线程（其中一个为扑空线程）。

908

这种交替方式中的开销来源与阻塞方式是一样的。一个缓存扑空导致该线程未就绪，但该扑空在流水线的后期才被检测到；若只有少数的交错线程，此时发生扑空的线程可能已经把其他的指令取到流水线中。不像Tera机器那样，编译可通过预测域保证流水线中的这些后续指令与该扑空指令无关；在这里我们必须自己处理这些指令。与阻塞方式同样的原因，建议选择的方法是排空这些指令，即将这些指令标记为不对处理器状态进行修改的指令。与阻塞方式的关键不同点是，由于其他线程的指令是一个周期接一个周期进入流水线的，不是所有的指令需要排空的——只有那些来自扑空线程的指令才需要。因而，线程未就绪的开销比阻塞方式小得多，后者流水线中所有的指令都需要排空。事实上，如果就绪和活动的线程足够（这要求大量的机器状态复制，并非该方式提倡的），扑空线程的指令根本不能进入流水线，则就不需要排空流水线（与HEP/Tera的方式形同）。与阻塞方式的比较如图11-29所示。

909

使线程未就绪的效果是短时延（如局部存储器访问或长指令时延）比阻塞的情况更容易隐藏，这也使得交替方式更适合单处理器中的多线程技术。对于一些在阻塞情况下不能被隐藏的非常短的时延，如流水线相关等，常常不需要使得线程未就绪就可以通过线程周期切换

的方式自然地隐藏。在 4 个线程 4 深度流水线中使用阻塞方式的例子（如图 11-27 所示）的不同效果如图 11-30 所示。在该简单的例子中，假定在指令序列的开始流水线处于稳定状

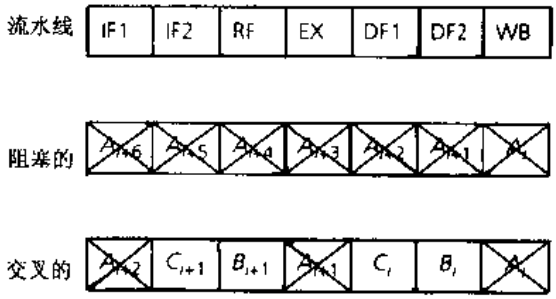


图 11-29　完全支持单线程的交替策略下线程变为未就绪的开销与阻塞策略下上下文切换开销的比较。图中首先显示了假定的 7 段流水线，然后是阻塞策略中延后扑空检测的影响：其中所有指令都来自于线程 A 并且流水线中的指令都被碾压；再后是交替线程策略的情况。在交替线程策略中，来自了三个不同线程的指令进入流水线，只有来自于线程 A 的指令才被碾压。此种情形下，扑空的切换开销是 3 个周期而非 7 个周期

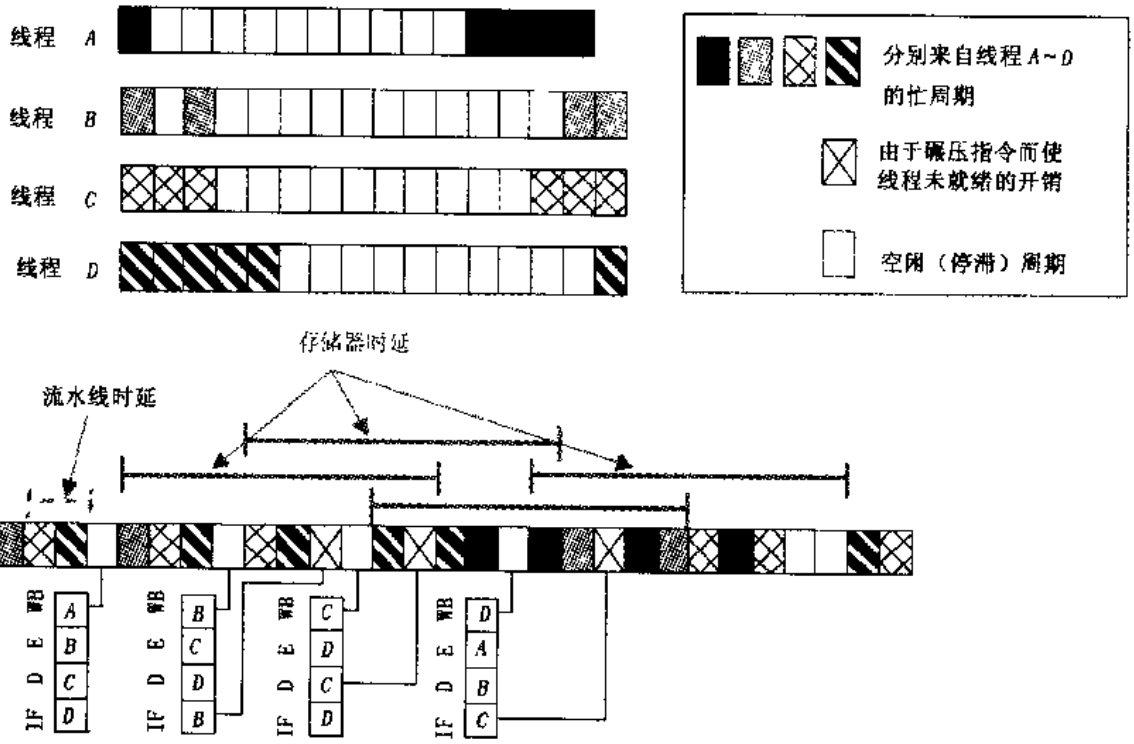


图 11-30　交替策略中的时延包容。现假定流水线为 4 个阶段，分别为取指（IF）、译码（D）、执行（E）、回写（WB）。图的顶部显示了当该线程是处理器中惟一执行者时，4 个活动线程的执行情况。底部显示了处理器在线程间的切换情况。与图 11-27 一样，其时隙中的指令是该周期从流水线引退（即将引退）的指令。显示的前 4 个周期中，来自于每个线程的指令分别引退。到第 5 个周期，来自线程 A 的指令即将引退，但此时检测到扑空需要未就绪挂起，于是该位置成为空闲。该时刻流水线的其他 3 条指令（空闲时隙的后面）来自于其他不同的线程，这样除了该扑空指令本身的一个周期外没有其他的切换开销。当 B 的下一条指令到达 WB 阶段（第 9 个周期）时，此时检测到扑空并需要转为未就绪。此时，由于线程 A 已经是未就绪的，分别来自于线程 C 和 D 指令已经进入流水线，同时还有 B 的不只一条指令（如图所示，该指令已经进入流水线的取指阶段，第 12 个周期将要引退）。这样，线程 B 扑空的指令浪费了一个周期，而且线程 B 的一条指令也被废除掉。类似地，线程 C 应该在第 13 周期引退的指令发生扑空时，也会导致来自于 C 的一条指令废除，等等

态，则处理器的利用率为21/30或70%（图11-27中的利用率为21/51或41%）。为了说明方便，本例使用的是一些非实际的参数，而事实上在几乎每周期发送一条存储指令的现代超标量处理器中，在阻塞方式中缓存扑空导致的切换开销可能非常高。与阻塞方式相比，该方式的缺点是实现起来较复杂。

阻塞方式以及最后一种交替方式（因而也称为交替方式）从简单的具有流水线互锁以及缓存的商用处理器出发，通过修改而实现多线程。正像前面所述，即使用在超标量处理器中，在给定的周期其发送的指令也是来自同一个线程。适合超标量需要更高级的多线程技术，但为了简单起见，我们首先检验其性能与实现问题，这可直接对两者进行比较。

11.7.2 性能收益

模拟研究显示阻塞和交替的方式（使用全流水线互锁及缓存）都能有效地隐藏读与写时延（Laudon，Gupta，and Horowitz 1994；Kurihara，Chaiken，and Agarwal 1991）。同时也发现所需要的活动上下文数很少，大约在4～8个之间，该数量可能随时延的增大（相对处理器周期的时间）而改变。

我们先检验对一些并行程序的模拟结果（Laudon 1994）。其结构模型仍然是具有16个处理器的缓存一致性多处理器系统，使用平面的基于主存的目录协议。处理器模型为：单发送，MIPS R4000的定点流水线，Alpha 21064的浮点流水线。缓存的结构为：小的（64 KB）单级缓存，不同类型访问的时延采用Stanford DASH多处理器原型（Lenoski et al.1992）。总之，建立阻塞与交替方式的目的是尽量改进处理器的性能。在供研究的7个应用程序中，其中3个的多线程加速比为2.0～3.5之间，其他3个的加速比为1.2～1.6之间，而最后一个应用的加速比可以忽略，因为其几乎没有并行性（如图11-31所示）。模拟发现交替模式总是优于阻塞模式，这与前面的讨论相吻合，所有的应用的几何平均加速比为2.75比1.9。

研究发现当发生很多短的流水线停滞（如浮点加、浮点减、乘法指令等）时，交替方式显示出最大的优点；这是因为这些时延不能被阻塞方式隐藏，而能被交替方式无开销地隐藏。长的流水线时延，如几十个周期的除法操作，都能被两种方法很好的隐藏（虽然交替方式由于很小的切换开销具有较好的隐藏效果）。即使扩展存储的结构和性能参数改变（例如，长的时延，多级缓存），交替方式的优点依然保持。甚至在现代的每个周期的多发射操作处理器中其优点可能会更大，因为缓存的扑空率增大导致上下文切换增加。两种多线程方式的潜在缺点是多个执行的线程共享同一个缓存、TLB、转移预测单元，这会增加其中的负干涉（例如，在低相联度缓存中线程间的映射冲突）；然而，研究中显示这些负效应是很小的。

910～913

图11-32显示了对于两个应用的更为详细的处理器执行时间（取所有处理器的平均时间），并显示了有趣的效果。对于单个上下文来说，Barnes-Hut由于使用小的单级的直接映像的相邻缓存而导致了严重的存储停滞时间（该应用具有4 K个体的小复杂度）。每个处理器使用更多个上下文可有效地隐藏这种数据访问时延，在图中切换开销较小的交替方式显示得更为明确。在Barnes-Hut（Water）应用中时延的其他主要方式是长时延的浮点指令，尤其是除法。交替方式能比阻塞方式更有效地隐藏这种时延。但是，当上下文的数量超过4时，两者隐藏能力增加会逐渐减小。这是因为模拟的除法单元不是流水线的，当来自多个线程的除法增多时该单元成为了资源瓶颈。PTHOR是使用多个上下文而不能显著提高性能的一个应

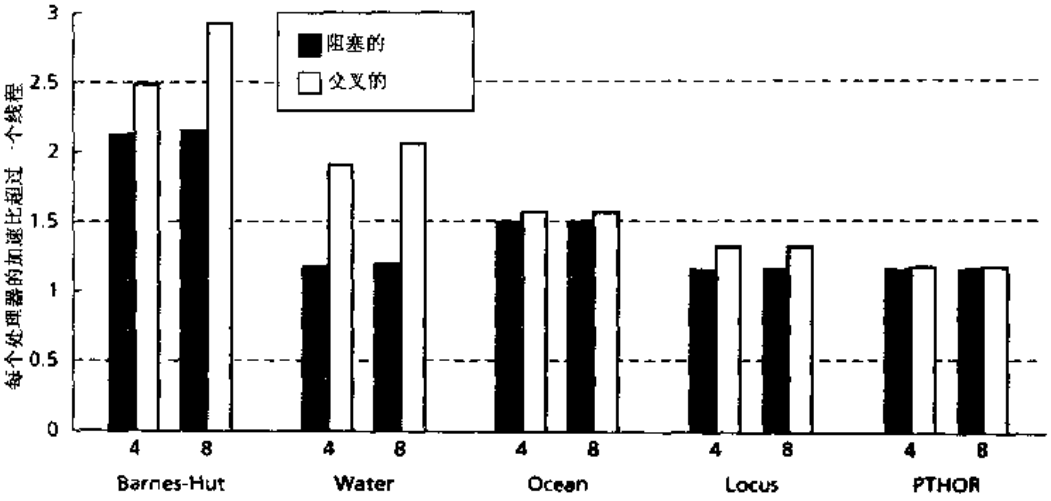


图 11-31 阻塞和交替多线程的加速比。图中的柱条显示了在单执行线程的处理器中 7 个应用程序执行时随上下文数目变化（4 和 8）的加速比。所有的结果都是在 16 个处理器上执行得到的。Locus 应用在图 11-24 中已经介绍过。Water 是模拟液态下水分子分子运动的模拟程序。PTHOR 是逻辑电路的并行事件驱动的模拟器，其并发剖视图见图 2-5 所示。多处理器系统模拟模型假定每个处理器配置单级的 64-KB 的直接映射的回写缓存。存储系统时延假定缓存命中时为 1 个周期，当缓存扑空若在局部存储器中得到数据时，该时延统一为 24 ~ 25 个周期，扑空且在远程户主访问时延为 73 ~ 135 个周期，扑空时在非宿主目录节点访问时延为 96 ~ 156 个周期。按照现代的观点来看，这些时延显然是不够的。类似 MIPS R4000 的整数单元具有 7 个周期的流水线，浮点单元具有 9 个阶段的流水线（5 个执行阶段）。除法指令的时延为 61 个周期，与其他单元不同的是浮点单元没有流水线。在遇到浮点指令时采用两种策略切换线程（或者说使线程成为未就绪）。阻塞策略对同步事件或除法指令采用显示的切换指令，此时耗费 3 个周期（比 7 个周期的切换要少，因为切换行为在译码阶段之后就已经知道，而不必等到回写阶段）。交替策略在该情况下使用了后退指令（在 11.7.4 节讨论），其结果是根据要废除的指令个数不同，其消耗的周期数为 1 ~ 3 个周期

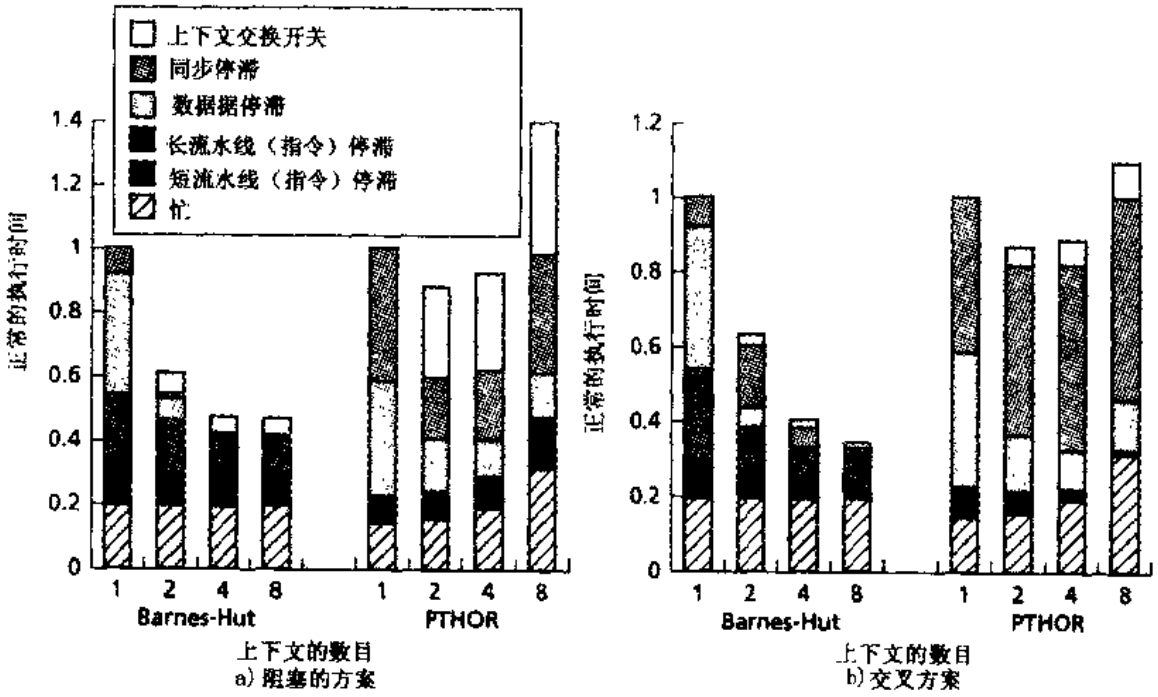


图 11-32 多线程下两个应用的执行时间分析。忙时间是指花费在执行应用程序指令的时间；流水线停顿时间是指由于流水线或指令相关而耗费的时间；数据停顿时间和同步停顿时间分别是指花费在存储系统以及同步事件的时间。最后，上下文切换时间是指上下文切换的开销

用，有时增多上下文反而有害。当上下文数为2时，存储时延能有效地隐藏，然而其主要的瓶颈是同步时延。该应用不具有额外的并行性以有效地开发多个上下文：即使使用多个线程，其大量的时间也是耗费在同步点上。注意，当上下文数量增多时，PTHOR的忙时间增多。这是因为该应用使用了一组分布的任务队列，随着线程增多，管理这些队列的费用会增大 这些额外的指令也会导致缓存扑空。

与预取一样，多线程会隐藏同类的数据访问时延，而且在不同的情况下各有特点（例如多线程可有效隐藏同步和指令时延，预取则不能直接隐藏）。两者混合应用导致的性能收益并不是很好理解的。例如，多线程的应用会导致缓存中多线程间的建设性和破坏性的干涉；而这是不可预测的，给预测分析带来了很大困难。与预取一样，多线程在阻塞读处理器中可与放松同一模型形成很好的互补；在更先进的处理器中的交互作用还不是很清楚。

以下两节将详细讨论阻塞方式与交替方式的实现问题，主要包括与商用处理器相比增加的部分。当然读者若对内容较熟悉，也可以直接阅读11.7.5节的超标量处理器中的多线程技术部分。

11.7.3 阻塞方式的实现问题

阻塞方式与交替方式都具有三种要求：*状态复制*、*程序指针（PC）单元的加强*、*控制加强*。状态复制基本上包括复制寄存器、程序计数器、以及每个活动上下文相关的程序状态字，这与前面讨论的相同。在多线程控制中多处理器需要的PC具有重大的改变。为了加强控制需增加一些逻辑和寄存器，以支持上下文间的切换管理、上下文就绪和未就绪等。我们将对这些需求一一进行讨论。

1. *状态复制*

我们首先对寄存器堆和处理器状态字分别进行讨论。为每个上下文分配一个寄存器堆或者一个静态的大段寄存器可以加快寄存器的访问，当然这会使得难以有效地使用硅的面积（如图11-33所示）。例如，在阻塞方式中由于一个上下文一直执行到遇到一个长时延时间为止，因而当其他的上下文闲置时只有一个活动的寄存器在使用。至少多个寄存器堆之间可以共享读写端口，因为这些端口会潜在地占据寄存器堆中硅的面积。除此之外，不同的线程实际使用的寄存器数量可能不同，而随着执行其所需要的寄存器数会动态变化。因而，上下文间动态地共享大的寄存器堆可能比静态地划分寄存器堆有更高的使用率。这就导致了类似缓存的寄存器结构：以上下文数和寄存器偏移为索引；当然也具有潜在的缺点：寄存器堆增大会增加访问时间。为了更有效地改进寄存器堆，已有多种途径（Nuth and Dally1995；Laudon

914

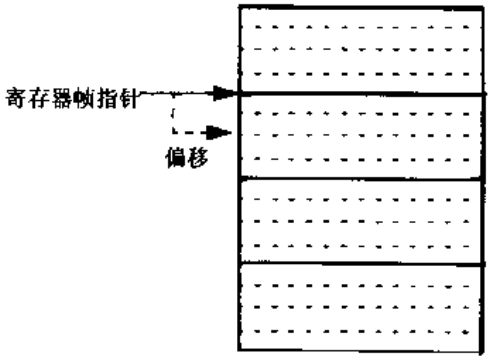


图 11-33 多线程处理器中分段的寄存器堆。该寄存器堆分为4个帧，假定在给定的时间可允许4个活动的上下文。每个活动上下文的寄存器值通过切换保留在该上下文的寄存器帧中，每个上下文的帧是通过编译管理的，就像是自己的完整的寄存器堆一样。帧中的寄存器访问是通过当前硬件寄存器帧指针实现的，该指针指定了在帧中的偏移，此编译不需要知道一个上下文使用哪个帧（这是在执行时确定的）。切换到不同的活动上下文只需要改变当前帧指针

1994；Omondi 1994；Smith 1985），但所有情况下复制的可能性依然存在。MIT 的 Alwife 机器在 Sun Sparc 处理器中通过修改其寄存器窗口机制来提供多个复制的寄存器堆。

现代的“处理器状态字”实际上包括多个寄存器；只有一部分（如浮点状态/控制等）保持具体进程的状态而非全局的机器状态，因而只有这些是需要复制的。另外，多线程又引入了一个新的全局状态字，称为上下文状态字（CSW）。该字包含一指明当前正在执行的线程标志，一位指明该上下文是否是可以被切换的（以下我们将看到当发生异常时该位是非使能的），一个比特向量指明哪个当前活动的上下文是准备执行的。最后，TLB 控制寄存器需要修改以支持多上下文中的不同地址空间的标志，并且允许一个单个的 TLB 项用来被多个上下文共享一个页。

2. 程序计数器单元

不同的活动上下文必须在硬件上保持其自己的 PC。有效支持异常的处理器也提供了最少的硬件复制，因为在很多方面异常很像上下文切换。除了 PC 链外（PC 链用来保持流水线不同流水段中指令的 PC 值），这种处理器还提供一个称为异常程序指针（EPC）的寄存器。EPC 是由 PC 链来填充的，总是保存最后一个来自流水线引退指令的地址。当发生异常时，由于错误的指令而停止装入 EPC，流水线中所有未完成的指令全部排空，同时将异常处理程序的地址放入 PC。当异常处理返回时，EPC 装入 PC 以便错误的指令能够重新执行。这正是在多上下文机制中所需要的功能。我们只需要复制 EPC 寄存器，使得每个活动线程一个寄存器。每个上下文的 EPC 用来处理异常或者上下文切换。当一个上下文正在操作时，该上下文将其 EPC 装进 PC 链，而其他的上下文继续保持原值。当异常发生时，当前上下文 EPC 的行为与单线程的情况是完全一样的。在上下文切换时，遇到错误指令（长时延指令）停止装入 EPC，同时将流水线中未完成的指令排空（与异常一样）；装入选择的下一个上下文的 EPC，执行其第一条未完成的指令，如此执行。将 EPC 用作两种用途的缺点是异常程序不能进行上下文切换（否则以前保存 EPC 的地址就会丢失），因而异常时上下文切换被禁止，当异常返回时切换使能位。然而，在该情况下 PC 的保存与恢复还是通过软件来完成的。 915

3. 控制

在阻塞方式实现中控制逻辑的关键功能是检测何时切换发生、选择下一个上下文、协调并执行切换。以下我们一一介绍。

在阻塞方式中，一个上下文切换可由以下三种事件激发：缓存扑空；显式上下文切换指令、同步事件和长时延指令；超时操作。超时事件是用来保证单个的上下文不会执行太久，或查询等待一个被同一处理器上其他线程设置的标志位。在缓存扑空时切换基于三个信号：缓存扑空标志，上下文使能位，就绪上下文标志位。实现显式上下文切换指令的一个简单的方法是假设后续的指令发生缓存扑空（即，使缓存扑空信号有效或发出另一个信号激发上下文切换逻辑）；这将导致上下文切换并在下一条指令后重新执行。最后，超时信号可通过一个门限计数器来实现。

虽然选择下一个线程的策略很多，但是实际中只是简单地以 round-robin 方式——并不关心上下文间的关系以及上下文执行的历史——也工作得不错。此时的信号要求是当前上下文标志、就绪且活动的线程向量，以及是否需要切换的检测信号。 916

最后，阻塞方式中的上下文切换还需要以下的行为。这些行为要求在处理器流水线中的不同阶段完成，因而控制逻辑必须在一个时间窗口上使能各种信号。

- 保存当前线程中第一条未完成的指令地址（例如，其上下文的 EPC）。
- 排空流水线中所有的未完成指令。
- 从选择的（或保存的）上下文的 PC（来自于 EPC）开始执行。
- 在 TLB 边界寄存器中装入正确的地址空间标志。
- 装入相关的新上下文的控制/状态寄存器（来自于以前保存的），包括浮点控制/状态寄存器以及上下文标志。
- 在多寄存器堆中切换寄存器堆的指针，指向新的上下文。

总之，阻塞的上下文交换方式中实现的主要开销是对寄存器堆的复制以及管理，这不仅涉及面积，而且还涉及寄存器堆的访问时间。如果后者在处理器周期时间的关键路径上，则为了保持高的时钟频率需要增加流水线的深度；但这又会加大转移错误预测的损失。所有这些因素都是应该进行性能评价的。除此以外，其他的硬件开销很小。

11.7.4 交替方式的实现问题

阻塞方式实现容易的一个关键原因是大多数时间其行为很像一个单线程处理器，只是上下文切换时才会导致其他复杂性以及处理器状态的改变。交替方式由于在线程间实现单周期切换而需要更多的一点支持。处理器的状态必须实现单周期切换，而且指令发射单元必须在连续的周期发射多个活动上下文的指令。同时需要一种机制实现：使得上下文活动或非活动，每周期要实现将活动/非活动的状态送往指令单元。下面再分别讨论一下状态复制、PC 单元以及需要的控制。

1. 状态复制

与阻塞方式一样，寄存器堆必须能动态地复制或管理，但由于连续的周期内可能访问不同的上下文寄存器，因而快速访问寄存器的压力就会增大。我们不能依赖于多阻塞方式访问模式的逐渐改变而实现。（因此，Tera 处理器使用存储体式或交替的寄存器堆，一个线程也会因为寄存器体忙而导致未就绪）。其中必须被复制的进程状态字部分与阻塞方式类似，即使处理器一定要在每个周期切换状态字。

917

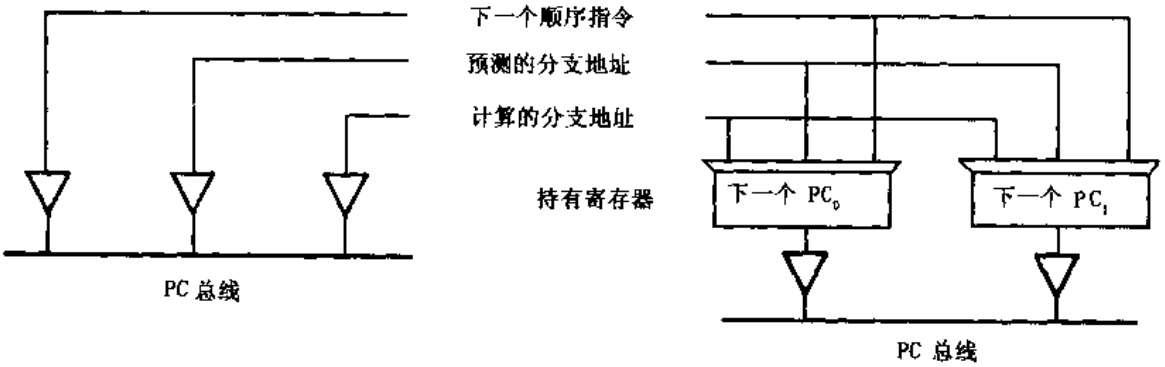
2. 程序指针单元

区别最大的部分是 PC 单元。来自不同线程的指令同时存在于流水线，处理器必须能够每个周期发送不同线程的指令，以避免因未就绪的线程而停滞。此时流水线也会受到影响，因为在实现正确的旁路和转发时，PC 链必须保持不同流水段指令对应的上下文标志。对于 PC 单元本身来说，需要一种新的机制以实现下列功能：处理上下文的实用性，排空指令，跟踪下一条要发送的指令，处理转移及异常等。以下我们将对其进行部分的讨论，更全面的内容可参考文献（Laudon 1994）。

下面考虑上下文的有效性。当遇到缓存扑空或显式后退指令（使得上下文在指定的周期内不可用）时，上下文会变成不可用。例如在同步事件处理时发送后退指令（如果指定的后退时间期满时同步事件还没有发生，而且该线程又成为可用时，则该线程的周期会浪费一直到等待的事件得到满足，或者发送另一条后退指令）。通过清“上下文可用”信号可使得该上下文停滞继续发送指令。为了排空流水线中的该线程的指令，我们必须向所有的流水段发出上下文标志信号，因为我们不知道哪个流水段包含该上下文的指令。在缓存扑空的情况下，导致扑空的指令地址装入 EPC。一旦扑空得到处理而该上下文又可用时，若上下文被选

择，则 PC 总线从 EPC 装入。显式后退指令与缓存扑空类似，只是不从该后退指令开始恢复，而是从后一条指令开始恢复。在 EPC 中包含一个下一位，该位表示从错误指令恢复还是从下一条指令恢复。

即使在一个标准的、单上下文单处理器中，三种来源可决定一个线程将要发送的下一个指令：下一个顺序指令、来自转移目标缓冲（BTB）中的转移预测，预测错误时的计算结果。当一次流水线中只有一个上下文时，一旦确定则“next PC”（NPC）寄存器就可以作为下一个指令地址来驱动 PC 总线。然而在交替的处理器中，在某上下文的下个指令地址确定并准备放入PC总线的周期期间不允许进行上下文调度。另外，由于上下文的NPC可能由不同指令的不同流水端来确定——例如，错误转移预测要比正确的预测或者非转移指令的确定迟得多——在同一时刻不同上下文可产生NPC值。因此，不同上下文的NPC值需保存在寄存器中，直到相应的上下文执行下一条指令时才将其驱动到 PC 总线上（如图 11-34 所示）。 918



a) 阻塞的方法　　b) 交叉的有两个上下文

图 11-34 具有两个上下文时在阻塞和交替多线程方法中驱动 PC 总线。如果使用更多的上下文，交替策略会需要更多的复制，而阻塞策略则不会

转移处理也需要另外的机制。当预测错误而要排空流水线时，上下文的标志要广播到所有的流水段，但这与无效上下文的功能是相同的。当实际的转移目的地址计算出时，推测发送的指令可能在所有的流水段，甚至可能还没有发送到流水线中（由于其他上下文的交替是随机的）。为了发现已预测的地址以进行正确的预测，当转移指令在流水段中推进时携带预测地址是很必要的。例如，一个预测的 PC 寄存器链与 PC 链在执行时并行推进，当转移操作到达适当的流水段时可装入和检查该预测链。

最后，我们考虑在一个上下文中发生异常的情况。一种选择是使得该上下文为就绪而让位于异常处理，并使得异常处理与其他的上下文交替执行（Tera 采用了类似的方法）。在这种情况下，另一用户线程也可能发生异常，因而异常处理应支持多个异常发生的情况。另一种选择是，当任何上下文中有异常发生时，使得所有的上下文为就绪，排空流水线中的所有指令；当异常返回时再使能所有上下文。但当异常频繁时，这会降低性能。这也意味着当异常发生时，所有活动上下文的（异常 PC EPC）应装入其对应线程的第一个未完成的指令。这比阻塞方式更加复杂，在阻塞方式时只需要保存当前正执行（发生异常）线程的 EPC。 919

3. 控制

除 PC 单元之外的与控制相关的两个有兴趣的问题是：如何追踪上下文的有效性信息以及提供给 PC 单元；每个处理器周期如何选择并切换到下一个上下文。“上下文有效”信号在缓存扑空时被修改，或被扑空返回时的回退及停滞操作修改。缓存扑空导致的有效状态可通过每个上下文的挂起扑空寄存器来追踪，该寄存器在扑空时装入，在扑空返回时检查并使能适当的上下文。对于显式回退指令，我们可为每个上下文保持一个计数器，遇到回退指令时初始化成回退值（此时上下文的有效信号也清除）。该计数器每个周期减 1，当减到 0 时该上下文的有效信号重新置位。

回退指令也可以用来隐藏指令时延，但是在交替多线程方式中很难选择一个好的回退周期数。由于编译可透明地重新安排指令，这种情况就变得更加复杂。随着处理器的更新换代，这种回退值在不断改变。幸运的是，短指令时延常常被其他上下文的交替自然地隐藏，而不用回退指令，如图 11-30 所示。对长指令时延的更好的解决还依赖于更多的上下文，如计分板技术。

至于选择下一个上下文，一个合理的方法是根据上下文的有效程度按照 round-robin 算法选择。

11.7.5 在多发布处理器中集成多线程

到目前为止，我们讨论的多线程与一个周期发射的操作数是正交的。而 Tera 系统中每个周期发送三条指令，其单线程中的多指令的包装（成一个宽指令）是通过编译来实现的，硬件所做的工作是每周期从单个线程中选出一个三操作的指令。单个的线程常常没有足够的指令级并行性以填满每周期的时间片，现代处理器中已经如此；然而随着每周期发送更多的指令，这种情况会变得更严重。当存在多个线程时，一个自然的想法是每个周期发送来自不同线程的指令，这样可更有效地填充发送时间片。这种方法称为*并发多线程*（simultaneous multithreading）而且已有很多的研究（Hirata et al. 1992；Tullsen，Eggers，and Levy 1995）。并发多线程与交替多线程类似，只是每个周期来自不同有效线程中的操作竞争使用发射时间片以及功能部件。

另一方面，传统的多发射处理器对效率的影响来自两个方面。第一，由于在单线程中发现指令级并行的能力有限，因而并不是所有的时间片都能填满。第二，由于长时延指令而使得很多周期没有事情可调度。简单的多线程只致力于第一个问题，而并发多线程则致力于两者的解决（如图 11-35 所示）。

920

对于编译来说，在同一周期从不同的线程中选择操作来调度可能是很困难的，但在动态调度的微处理器中已支持很多这样的机制。取指单元必须扩展成能在一个周期从多个上下文中取操作，并放入重排缓冲区中；不管来自于哪个上下文，发送逻辑可从该缓冲区中选择指令发送。对多发射动态调度单线程处理器的研究表明，空闲周期和空闲时间片在长时延指令、缓存扑空、TLB 扑空以及装入时间片（其中前两个尤其重要）中具有很好的分布。时间浪费和时延原因的多样性显示细力度多线程是一个不错的方案。

除了在交替多处理器中讨论的发送外，在实现并发多线程处理器时还会遇到几种新的问题（Tullsen et al.1996）。第一，从不同的线程中取指的灵活性究竟有多大？灵活性越大（对比于在一个周期内从单个上下文中取，或在一个周期从单个线程中最多取两个操作），取值

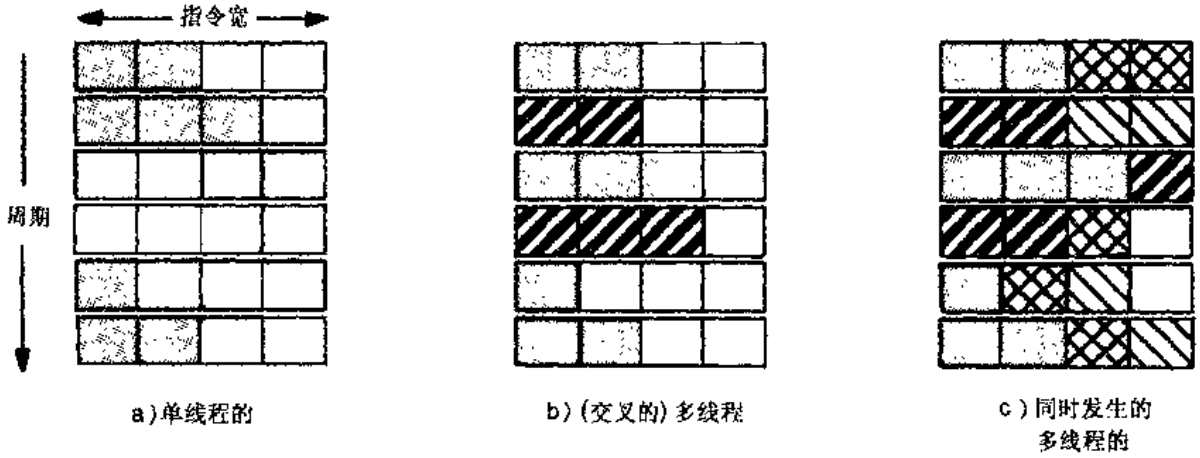


图 11-35　同时并行的多线程。图中说明了在 4 发射的处理器中简单的交替多线程和同时并行多线程潜在的性能改进情况。阴影部分以及方框中的不同图案用来区分不同的操作与线程，空白方框表示空闲指令时隙

逻辑与指令缓存设计就越复杂。然而，更多的灵活性会减小空闲取指时间片的频率。第二，如何选择下一个要取指的上下文？我们可以按照固定的特权顺序选择上下文（例如，可以先选择上下文 0 填充指令，再选择上下文 1 等等），也可以按照上下文的执行信息来选择（例如，给定当前在取指单元或重排序缓冲中指令最少的上下文最高的优先级，或者给定当前未处理完的缓存扑空最少的上下文最高的优先级）。最后，每个周期如何从重排缓冲中选取就绪的操作？在动态调度处理器中的标准实现是选择最老的就绪操作，其他的选择，如基于操作属于的线程以及线程的行为等，可能更适合这种情况。 921

有关多处理器上下文中的并发多线程的性能数据非常少。对于单处理器的性能收益以及刚才讨论的折中方案的对性能的影响研究发现该技术很有希望，而且与动态调度的单线程处理器相比，并发多线程中对推测执行的支持并不重要，这是因为存在更多的有效线程，因而可能有非推测的指令可选择（Tullsen et al. 1996）。

总之，随着数据访问和同步时延相对于处理器的速度在逐渐增大，在多处理器中应用程序的数据访问的模式越来越复杂而且不可预测（多处理在不断成熟并扩展），多线程在隐藏时延方面越来越成功。多线程是否在实际中与微处理器结合依赖于许多因素，例如是否使用其他的时延包容技术（如预取，动态调度，放松一致模型等），以及多线程技术如何与之相配合。鉴于多线程要求额外的显式线程、强烈的复杂度以及状态的复制，一个有意义的选择是将多个简单的处理器集成在一个芯片上，不同的线程在不同的处理器上执行。此时定性的平衡是比较清楚的，但无论作为桌面系统还是多处理器的单节点，在耗费和性能方面如何与并发多线程相比较还不是十分清楚。

表 11-2 总结并比较了在共享地址空间中隐藏时延的 4 种主要技术的一些关键特征，这在 11.4～11.7 节中已经讨论过。这些技术可以并且常常混合使用。例如，读阻塞的处理器可利用放松一致性模型隐藏写时延，可用预取或多线程隐藏读时延。而且我们已经看到动态调度处理器可从预测、放松一致模型以及多线程中分别获益。在动态调度处理器中这些技术如何相互作用、甚至在读阻塞处理器中预取与多线程如何互补，随着处理器速度与数据访问时延的差距增大，这些技术如何成功隐藏时延，这些问题可能在将来会很清楚。

表 11-2 共享地址空间中 4 种时延包容技术的关键性质

性 质	放松模型	预 取	多线程	成块数据传送
时延包容的类型	写（阻塞读处理器）读和写（动态调度，处理器）	写、读	写、读同步指令	写、读
软件需求	标注同步操作	可预测性	显式额外的并发性	识别并协调成块传送
额外硬件支持	几乎不要	几乎不要	很大	在处理器中没有但在存储系统中要有成块传送器
在商用微处理器中是否提供支持	有	有	目前没有	不需要

11.8 免锁定的缓存设计

922

通过本章我们已经看到，除了处理器需要的支持（另加的带宽以及存储和通信系统较低的占用率）之外，共享地址空间中的几种时延包容技术的有效性要求缓存一次允许多个扑空挂起。在引出本章的结论之前，我们首先看一看这样的免锁定缓存的设计。

在允许多个待完成扑空的缓存子系统中有几个关键的设计问题：

- 多少以及哪种扑空允许同时处于待完成状态？与处理器一样，同时允许多个写比多个读更容易。复杂度上的两个不同点是：1）单个读多个写，2）多个读写。
- 如何跟踪多个未完成的扑空？对于读，我们需要跟踪的是：字请求的地址；读请求的种类（例如，读、独占读、预取、单字读、双字读等）；返回数据的位置（返回处理器的哪个寄存器，当多个处理器共享缓存时返回到哪个处理器）；未完成请求的当前状态。对于写来说，不需要跟踪返回数据，但要写的新数据必须与存储结构中下一层（如果有）的返回数据块相结合。其中一个关键问题是是否存储缓存块内部的大部分信息，或者为未完成的扑空设置一组事务缓冲。当然，在履行这些要求的过程中，我们需要保证设计不存在死锁或活锁。
- 如何处理对同一内存块多个未完成访问的冲突？哪种类型的冲突扑空是不允许的（例，停滞处理器）？例如，是否当读扑空未完成时允许对同一块的写？

923

- 当多个未完成的请求映射到同一个缓存行造成冲突时，即使它们指向不同的存储块，如何处理？

为了说明这些选项，我们先考察两种不同的设计。这些设计的主要区别在于如何保存跟踪未完成扑空的信息。第一种设计使用一组分离的事务缓冲区来跟踪请求。第二种设计尽可能使用缓存块本身跟踪未完成的扑空。

第一种设计是用在巨型机 Control Data Corporation's Cyber 835 中的简化版本，它是在 1979 年提出的（Kroft 1981）。该设计在缓存中增加了几个扑空状态保持寄存器（MSHR），以及一些相关逻辑。每个 MSHR 处理一个以上的对单个存储块的待完成扑空。这种设计在对同一块的并发处理上允许相当大的灵活性，因而每个 MSHR 存储了大量的状态信息，如表 11-3 所示。

在常规缓存中对MSHR的访问是并行的。若在缓存中访问命中，则执行一般的命中行为；

表 11-3 MSHR 状态顶及其作用

状 态	描 述	目 的
高速缓存块指针	分给本请求的高速缓存块指针	在组相联高速缓存中指定一组中的行
请求地址	挂起请求的地址	允许请求合作
单位标识标签（每字一个）	识别处理器中的请求单元	给出返回本字的位置
发向 CPU 的状态（每字一个）	单元标识标签的有效位	指出本单元 ID 标签是否有效
部分写代码（每字一个）	跟踪对一个字进行了部分写操作的位向量	从存储器返回哪个字来覆盖
有效指示	有效 MSHR 的内容	内容有效吗

当缓存访问扑空时，其采取的行动依赖于 MSHR 的内容：

- 如果对于该块没有分配 MSHR，则分配一个新的 MSHR 并初始化（若没有可用的 MSHR，或者改组所有的缓存行都具有未决的请求，则处理器停滞）。若当前扑空的缓存行包含脏数据，则启动回写。然后，若处理器请求是写操作，则数据写进该缓存块的正确偏移地址，同时设置 MSHR 中的相应的部分写编码位。另外也要发请求从主存子系统（例如，BusRd，BusRd X）中取出该块，并放入缓存。
- 若该缓存块的 MSHR 已经分配，则该新的请求与以前对同一块的请求合并。例如，一个新的写请求可通过向已分配的缓存块写数据，并设置 MSHR 中的部分写位来合并。对一个字的读请求，若该字已经被以前的写操作写入时，可通过对缓存数据的简单读而完成。当读请求的字不曾被访问时，则简单地设置适当的单元标志；当该字已经被请求过时，则或者必须分配一个新的 MSHR（因为每个字只有一个单元标识），或者必须停顿处理器。由于写不需要单元标志，当对一个已经处于挂起状态的字进行写请求时，其处理是较容易的：从存储器返回的数据可直接转发给处理器。当然，对一个块的第一次写操作会产生独占所有权的请求。

最后，当某块的数据返回缓存时，MSHR 中的缓存块指针应表明将数据放在哪里。其中的部分写编码可以用来避免对最近写过的缓存数据重写，发给中央处理器（send-to-CPU）位以及单元标志用来对等待功能单元的直接回应。

924

这种设计需要对 MSHR 进行相联查找，但可允许缓存支持不同种类的存储访问同时进行。幸运的是，由于 MSHR 对给定的块履行着合并以及分开的复杂任务，对于扩展存储结构中的以下部分来说就好像来自处理器的对不同块的请求。前面章节讨论过的一致性协议在设计上已经能非死锁地处理这些对不同块的多个待完成的请求。

该设计的另一个选择是将与未完成的写请求有关的状态保存在缓存行本身，而不用独立的 MSHR。在写返回缓存中除了标准的 MESI 状态外，我们又增加了 3 个临时的或挂起的状态：无效挂起（IP），共享挂起（SP），独占挂起（EP），这些状态表明当目前的写扑空（未完成）发出后块的状态。每个状态中，缓存标记是有效的，缓存块正在等待来自存储系统的数据。每个缓存块具有一个子块写比特（SWB）的向量，每个字对应于一位。在 SP 和 EP 状

态中，打开的位表明该字由于存储请求而被处理器写过，这些被写过的数据不应当重写。然而，对于关掉的比特对应的那些字，在 EP 状态中表示无效，而在 SP 状态中则表示有效（没过时）。最后，需要一组独立的挂起读寄存器以保持挂起读请求的地址和类型。

保持每个块的额外状态信息的重要优点在于：在跟踪写挂起请求时不需要另加的存储。当一个写操作发现对应的块不处于修改状态时，则该块简单地进入适当的挂起状态，发起相应的事务，并设置 SWB 位以表明当前写改变了哪些字，这可使得后续的归并可正确地产生。当写时发现其对应的块正处于挂起状态时，只要求将该字写入该行，并设置相应的 SWB 位为打开状态。读操作可能使用读挂起寄存器。当读操作发现目的数据字在块中处于有效状态时（包括一个 SWB 位打开的挂起状态），则简单地返回数据即可；否则，将该读操作放入被跟踪的读挂起寄存器中。

925

若被读写访问的块不在缓存中（没有与之匹配的标记）时，则可能产生回写。该块设置成无效的挂起状态，所有该块的 SWB 读关掉（除非当写操作时被写的那些字），在总线上替换相应的事务。若没有匹配标记而且该块已经处于挂起状态，于是处理器停滞。最后，当一个正进行的请求返回时，则更新相应的缓存块（除了其 SWB 位打开的那些字外）；此时该缓存块离开挂起状态。所有的读挂起寄存器都检查该块是否是其正等待的块；若是，则向其请求返回这些数据并写进那些挂起的寄存器。实际状态变化、行为以及竞争条件的细节参照文献（Laudon 1994）。使得竞争条件相对容易处理的关键是要看到，即使在所有权得到之前应完成对缓存块的写操作，这些字对其他处理器的请求再得到所有权之前也是不可见的。

总之，这两种免锁定缓存设计并没有概念性的区别。后者将写操作的状态保存在缓存块中而减少了挂起寄存器的数量以及相连查找的复杂度；然而，它比 MSHR 具有更强的存储敏感性，即使其中只有少数同时具有待完成的请求，但所有行的状态都保存在缓存中。两种情况下与其他协议的正确操作是类似的，而且要适度。

11.9 结论

随着处理器与存储器以及通信在速度上的差距不断增加，时延包容在将来的多处理器（包括单处理器）中越来越关键。目前已经开发了多种时延包容技术，每种也具有各自的优缺点。这些技术都要求应用中的并行性多于使用的处理器数，而且都对通信体系结构带宽提出增强的要求。这种越来越大的压力使得通信体系结构其他方面性能（如处理器开销、辅助占用开销、网络带宽）的有效性和负载平衡更加重要。例如，由于发生主处理器上的开销不能由该处理器隐藏，如果该开销是由数据访问时延的关键部分造成的，那么除了增大消息外的其他时延包容技术是很难奏效的。

对于缓存一致性的处理器，时延包容技术是由处理器和缓存存储系统硬件来支持的，在设计上具有广泛的选择。大多数硬件支持的时延包容技术也是适应单处理器的；事实上这些技术在商业上的成功依赖于它们在隐藏小时延的高端单处理器的生存力。像动态调度、放松同一性模型以及预取等技术在今天的微处理器体系结构中是经常遇到的。最通用的时延隐藏技术——多线程——在商业上还不是很流行，很大程度上是因为它在单处理器上还没有得到检验。近来将多线程集成于超标量处理器中进行动态调度的方向看来很有希望，但与单个芯片上集成多个简单处理器还难以比较。大家感兴趣的一个常见问题是：在单处理器中的时延隐藏在多处理器中的成功程度究竟如何。

926

不管问题有多少以及硬件支持的选择如何，目前很多时延隐藏问题也是软件问题。编译自动预测到何种程度就可使用户不再担心呢？如果不可能自动化到理想程度，那么用户将向编译传递什么信息呢？如果块传送在缓存一致性机器中确实有用，那么用户如何对这种既有隐式的读写通信又有显式的块传送的混合模式来编程呢？放松一致模型本身也存在指定重排的适当信息的软件问题（例如，按照需要标记出冲突操作等）。最后，程序是否被分解并分配已具有多线程所需要的足够的显式并行性（额外线程）？使支持时延包容的软件自动化和简化方面还远远没有完成。事实上，时延包容技术在将来如何发挥作用，以及它们需要什么样的软件支持，依然是并行体系结构中一个有意义的开放问题。

习题

11.1　从通常的意义上讲，为什么时延减小比时延包容的想法好？

11.2　假定一个处理器在传输 m 个等大小消息时，通信量为 k 个字，处理消息时通信辅助占用的开销为 o，处理器上没有开销。当只有通信（而非计算）与通信重叠时，求处理器所见的最好情况的时延？首先假定通信没有确认信息（即，瞬间传播且不发生任何开销），然后具有确认信息。画出时空图，并说明其中重要的假定。

11.3　你已经学了不少在共享存储多处理器中进行时延包容和隐藏的技术。这些技术包括成块、预取、多上下文处理器以及放松同一性模型，针对下面的每一种情况，对每种技术，讨论它们是否能有效地减少隐藏时延。假设我们讨论的处理器具有成块读的能力并列举你所作的任何其他假设。

1）一个复杂的图算法，它有大量并行性的并用链接指针结构表示。

2）一个并行排序算法，其中通信由生产者发起，并通过大时延写操作实现。不可以做接收方发起的通信。

3）一种迭代方程求解器，其中内层循环由矩阵×矩阵构成，假设两个矩阵都很大，不能放在高速缓存中。

927

11.4　你将负责在一种新的并行超级计算机上实现消息传递。该机器的体系结构还没有解决，由于目前体系结构要运行前代机器体系结构上运行的消息传递应用程序，你的上司说是否提供硬件缓存一致性支持依赖于在两种系统上运行的消息传递的性能。在没有缓存一致性（“NCC”系统）的系统中，你们组的工程师告诉你消息传递在实现时应支持连续传送 1 KB。为了避免在接收方的缓冲出现问题，在开始下一次消息传播前，应该对每个单个的 1 KB 传送发送一个确认（因此每次只有一个块在进行中）。每 1 KB 传送需要 200 周期的建立时间，之后开始进入网络。该开销主要包括确定读内存缓冲区的位置以及建立 DMA 引擎的时间，该引擎执行传输操作。假定 1 KB 的块到达目的节点后，需要 20 个周期产生应答，而且在发送方需要 50 个周期接受确认后才启动下一个 1 KB 的传送。

在具有缓存一致性的系统（“CC”系统）中，消息是按照一系列 128 字节的缓存行发送的。然而在这种情况下，只需在每 4 KB 页的末尾发送确认。这里，每次传送需要 50 个周期的建立时间，期间可以提取出一个缓存行，如果需要也可保持缓存的同一性。该行然后发送给网络，只有该行完全发送到网络时，才能处理下一行。

下面是系统参数：时钟频率 = 10 ns（100 MHz），网络时延 = 30 个周期，网络带宽

= 400 MB/s。其他假定任意。

1）在 NCC 系统中对于 4 KB 的消息来说，其时延（直到目的节点收到最后一个字节）和得到的带宽是多少？

2）在 CC 系统中相应的时延和带宽是多少？

3）一个团队的设计者向你说明你可以很容易地改动该 CC 系统，使得前一个消息在发送到网络的同时，下一行就可以处理。在这样的修改后的 CC 系统中，计算 4 KB 的消息时延。

11.5 考虑一个并行数据矩阵转置的例子，如高性能快速傅立叶变换中的计算。图 11-36 显示了转置的插图。每个处理器转置其分配行的一块并传送给其他处理器，其中也包括它自己。通过读写执行转置已在第 8 章的习题中讨论过。由于一个处理器将数据发送给哪些处理器是完全可预测的，因此处理器可一次在一个单个消息里发送一整体块，而不是通过单个的读写缓存扑空来传送每个块。

928

1）你希望块传输性能相对于读写性能的曲线是什么样的？

2）该块传送机制最得益于什么特殊特点？

3）编写该块传输形式的伪代码。

4）假设你想在 Raytrace 应用中使用该块数据传送。使用它的目的是什么，如何使用？你认为会得到重大的收益么？

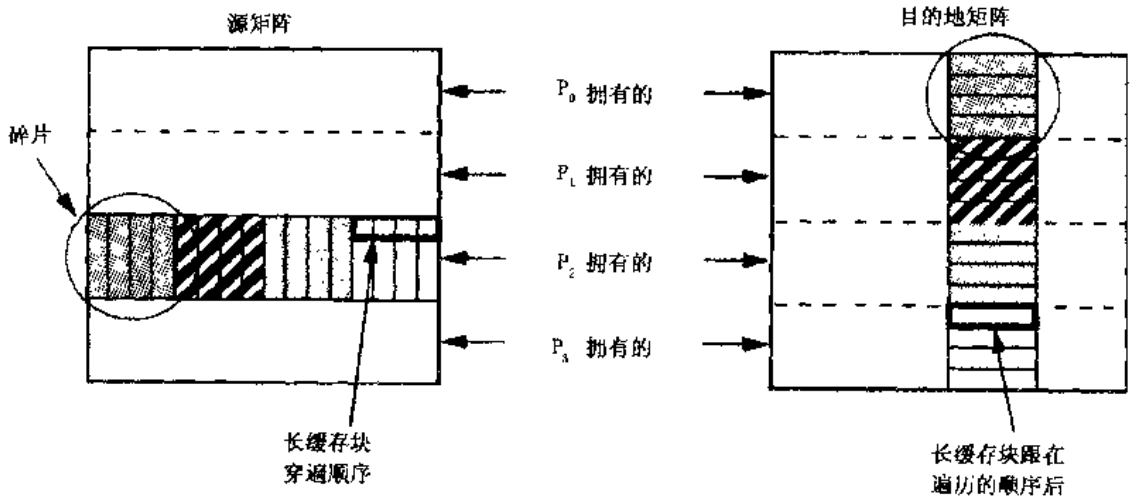


图 11-36 发送者启动的矩阵转置。源和目的 n×n 矩阵按照毗邻行分组，并分给若干进程。每个进程将自己的 n/p 行划分成大小为 $n/p \times n/p$ 的 p 个碎块。作为代表，我们考虑进程 P_2：它向其他每个进程发送一个碎块，并且转置局部的一个碎块（从左面数第三个）。每个碎块可以作为一个单块发送消息，而不是通过单个的远程写或远程读（如果是接收者发启的）

11.6 在缓存一致性共享存储空间的块数据传送中，一个令人感兴趣的问题是必须考虑长缓存块和空间局部性的影响。假定数据块传送移动可以平衡缓存一致性机制。考虑规则网格（毗邻通信）中的简单方程求解器。假设 $n \times n$ 的网格按照 p 个处理器分成方形的子块。

1）与本章的 FFT 结果相比，当每个行或列边界通过块传送直接发送时，你希望该应用会有怎样不同的曲线，为什么？

2）你如何构建块传送结构只发送有用数据？与前一个相比，你希望此情况下的性能如何？

3）什么样的体系结构参数最能影响 2）部分的平衡问题？ 929

4）如果你确实想使用块缓存传送，对于并行程序你希望做什么样的深入改变呢？

11.7 如果在用阻塞读跨越写操作的性能研究中二级缓存是阻塞的，那么允许写到写重排相对于不允许重排序有什么好处呢？如果允许重排序，是在什么样的条件下呢？

11.8 1）写缓冲能允许几种优化策略，如缓冲本身，将缓冲里对同一缓存行的写归并，在缓冲里发现匹配时直接将缓冲里的值前递给读操作。在 SC 下为了保持程序顺序，对于这些优化应当进行什么样的约束呢？在处理器一致模型中归并优化有什么危险呢？

2）为了保持如 SC 中的所有的程序顺序，我们可以按照下面的方法进行优化。在基线实现中，写操作时处理器立刻停顿，直到写操作完成。另一种选择是将写操作放入缓冲区而处理器不停顿。为了维护写操作期间的程序顺序，写缓冲引退一个写操作（即，在存储结构中继续下传并可能对其他处理器可见）仅在写操作完成之后。写到读的顺序是通过在读扑空时刷洗写缓冲保持的。

ⅰ）这种优化提供什么样的重叠？

ⅱ）你是否希望这会产生大幅度的性能提高？能否说出你的理由？

11.9 考虑在缓存一致性共享地址空间中跨越存储操作的实现要求。为了跨越写操作，我们需要一个写缓冲和一个非阻塞的写。为了有效地跨越读操作，我们需要非阻塞读以及指令前瞻和推测执行。在存储系统级，我们需要允许多个未完成扑空操作的免锁定缓存。这些靠近处理器的结构关心同一性模型的维护，扩展存储结构的其他部分可以自己重排操作。现在考虑下面需要保持同一性的机制，给定的支持如下：

1）我们需要的大多数机制必须确定一个（多个）操作的完成。在读阻塞的处理器中，为了保持释放同一性，与顺序同一性相比我们需要什么样的新机制呢？你打算采用什么样的附加结构（如果有）实现呢？

2）假设在一个写缓冲用不同的方法支持写重排的处理器中遇到了一个写到写存储屏障（MEMBAR）。该处理器必须停顿么？或者能跨越过该存储屏障么？解释如何提供存储屏障指示的写到写次序，什么时候写缓冲和处理器必须停顿？

3）任何同一性模型中的一个关键机制是计数将到的产生作废的写的确认。机器需要做的可以在接近处理器一方，或者在接近存储控制器一方。其主要的折中是什么，在决定做什么时有什么样的经验信息可以帮助？ 930

11.10 1）在缓存一致系统中你能将绑定的预取提前到什么时候发送呢？

2）我们已经讨论了多处理器中非绑定预取的优点。在单处理器中非绑定预取比绑定预取具有哪些优点呢？这里的预取指取进寄存器堆。

11.11 有时要对是否发出预取进行估计。这就在包含预取的循环中引进了条件表达式。利用伪代码构造一个简单的例子，并阐述其中的性能问题。你如何弥补该问题呢？

11.12 阐述以下情况：生产者发起的传输操作比预取或更新协议更适当。当你设计一个机器时是否实现传输指令呢？

11.13 考虑下面的循环：

```
for i ← 1 to 200
  sum = sum + A[index[i]]
end for
```

使用非绑定的软件控制的预取插入写一种代码。包括开始、软件流水线稳态部分和结尾。假定存储时延为 5 个循环迭代时间（数据已经返回并预取在基本缓存里）。

1）对于类似本例子中的情形，什么时候进行间接访问预取，需要时可使用额外指令产生地址。一种可能是将预取时计算的地址保存在寄存器里，以后装入时再使用。这样做有什么问题或有什么缺点么？

2）在向下预取多级的间接访问中发生例外时会怎样？这会带来怎样的复杂性，如何来寻址？

11.14 描述一些能够预取非规则访问的的硬件机制（高层次上），如记录、列表等。

11.15 在单处理器中为了隐藏时延，如何重写下面的循环：

```
for i=0 to 128 {
  for j=1 to 32
    A[i,j] = B[i] * C[j]
}
```

为了减小开销，尽量预取哪些你希望在缓存扑空的访问。假定读预取表示为 PREFETCH（&variable），而且预取 variable 所在的共享状态的缓存行。一个读排他预取操作，表示为 RE_PREFETCH（&variable），预取排他状态的行。该机器的缓存扑空时延为 32 个周期。显式地预取每个需要的变量（不考虑缓存块的大小）。假定缓存很大（不必考虑冲突扑空），但是并不足够大让你在开始时就预取任何事情。换一句话说，我们在寻找恰巧能及时预取。矩阵 A[⊖] 在存储器中的存放方式是 A[i, j] 和 A[i, j+1] 相邻。假定循环中的计算需要 8 个周期完成。

931

11.16 列举两个例子。其中，当预取编译看到一个同步操作非常保守时，假定什么都在缓存中的编译预取决策是无效的。此种情况下程序员如何能做得更好。本书中的一些应用情况研究会有所帮助。

11.17 预取的一个选择是使用非阻塞装入操作，而且在计算需要数据前发送这些操作。在这种情形下预取和非阻塞装入如何折中？

11.18 软件控制的预取的实现问题可分为两类：指令集的加强和对未完成预取的追踪。

1）预取指令和通常的指令有什么不同？

2）预取指令的形式有多种选择。例如，一些体系结构允许装入指令具有多种风格，一种可为预取保留。或者一些体系结构中保留一个特殊的寄存器的值总为 0（如 MIPS 和 SPARC 体系结构），以该寄存器为目的操作数的指令解释为预取指令，因为该指令不会改变任何寄存器的值。第三个选择是具有独立的预取指令，具有与装入指令不同的操作码。

i）你认为哪个是最好的选择，为什么？

⊖ 原书为 A[i, j]，有误。——译者注

ⅱ）你会使用什么样的预取指令寻址模式，为什么？

3）是否有必要在处理器内部保存来完成的预取状态？这样会增加性能么？理由是什么？你是否会将此种支持与未完成写合并？或者使用独立的结构呢？讨论其中的折中问题。

11.19　考虑软件控制预取的一些策略问题。

1）假设我们对认为在主（一级）缓存中扑空的访问发出预取。出现的一个问题是，我们在存储系统的哪一层缓存（除了第一层）探测预取是否满足？由于编译算法通常在调度预取时保守地假定要隐藏的时延是机器中最大的时延（假定是没有争议的），一个可能性是甚至不探测缓存的中间层，而总是假定从主存或另一个处理器的缓存中得到数据（若该块是脏的）。你认为这种方法有什么问题，哪个是最重要的？

932

2）由于预取是暗示性的，硬件可以取消而不影响正确性。你会取消下列的预取吗？ⅰ）当发生 TLB 扑空时；ⅱ）当跟踪未完成存储操作（包括预取）的缓冲区满时。哪种是最关键的？

11.20　考虑下面预取到主缓存和仅预取到二级缓存的折中问题。

1）在选择时的定性折中和问题是什么？你如何做？

2）当预取到一级缓存很有好处时，仅仅使用下面的参数构建一个分析表达式。p_i 是预取数据到主缓存中足够早时的预取数量，p_a 是在使用前预取的数据从缓存移动的情况数，p_c 是预取替换有用数据的冲突数，p_f 是预取填充数（例如，预取将数据放入缓存的次数），l_s 是对二级缓存的访问时延（对主缓存的访问除外），l_f 是预取填充停顿处理器的平均周期数。确定好成熟的通用表达式后，你的目标是确定值得预取到一级缓存时的关于 l_f 的条件。为此，你可以采用以下简单的假设：$p_s = p_i - p_a$，而且 $p_c = p_a$。你的分析怎样，或者遗漏的什么，哪些使得在实际中很难依赖该表达式？

11.21　考虑一个“阻塞”的上下文切换处理器（例如，只有当长时延事件发生时才切换）。假定具有任意多的线程和上下文，在下面的回答中要清楚地说明一些假定。某个应用的线程已经分析过，其执行剖视情况如下：

- 40%的周期花费在指令执行期（忙周期）
- 30%的周期花费在 L_1 高速缓存扑空停顿但 L_2 高速缓存命中（10 个周期扑空开销
- 30%的周期花费在 L_2 高速缓存扑空停顿（30 个周期扑空开销）

1）如果上下文切换开销是 5 个周期，忙时间是多少？

2）如忙时间必须保证大于等于 50%，则最大上下文切换时延是多少？

11.22　在阻塞的具有缓存的多上下文的处理器中，上下文切换仅仅发生在缓存访问扑空时。此时阻塞的上下文进入“停顿”状态，该状态保持到要求的数据进入缓存。这时，该上下文进入“就绪”状态，当活动的上下文阻塞时就允许其调度执行。当一个活动的上下文刚开始执行时，就发送使其阻塞的访问。在刚描述的模式中，多上下文之间的交互是否存在潜在的死锁？若能，举例说明所有上下文都不能前进的例子。你如何防止该种情况发生？若不能，说明理由。

933

11.23 若考虑缓存扑空，图 11-25 所示的处理器利用率相对于多线程深度的理想曲线会发生什么呢？在理想曲线图中画出该图的更为实际的曲线。

11.24 给定下面的输入信号，写出确定在阻塞模式中是否发生上下文切换的逻辑等式：缓存扑空或者 CM，扑空切换使能或 MSE（高速缓存扑空的使能切换），允许处理器使能切换的信号（CE），就绪上下文个数 OneCount（Cvalid），显式上下文切换指令（ES），超时（TO）。写出描述确定何时处理器应当停顿而非切换的等式。

11.25 在讨论阻塞多线程中实现 PC 单元时，我们认为例外 PC 以及上下文切换的使用意味着上下文切换在例外时非使能。这是否说明内核根本不能使用硬件提供的多线程？若是这样，为什么？若不是，则如何管理内核来使用多个硬件的上下文呢？

11.26 为什么异常处理在交叉方案情况下要比阻塞方案复杂得多？你会如何处理所引起的问题？

11.27 你如何理解 Tera 处理器可能前瞻跨越过转移指令？该处理器提供 JUMP_OFTEN 和 JUMP_SELDOM 转移操作。你为什么这样想？

11.28 考虑一个简单的类似 HEP 无缓存的多线程机器。假定平均的存储时延为 100 个时钟周期。每个上下文具有阻塞的装入操作，而且机器强制为顺序同一性。

1）假定一个典型工作负载的 20% 指令为装入指令，10% 是保存指令，为了隐藏存储操作需要多少个活动的上下文？

2）若机器支持释放同一性模型要求有多少个上下文（仍然具有阻塞装入）？说明所有的假定。

3）我们假定为阻塞的而非周期交替的 HEP 处理器，1）和 2）又需要多少个上下文呢？假定装入和保存的缓存命中率均为 90%。

4）对于 3），假定上下文切换开销为 10 个周期，处理器的峰值利用率是多少？

11.29 对应用的研究显示，将释放式同一性和预取结合起来，同单独用这些技术相比，总是能得到更好的性能。当多个上下文和预取技术结合就不一定如此；结合的性能有时会更差一些。用一个例子来解释后面这种现象。

934

第 12 章　将来的发展方向

在写本书的过程中，我们感到并行计算机体系结构的一个突出特点是迅猛的变化步伐。当使用新的设计时，以前的那些关键的重要设计则成为“过去”，而且主要的开放问题也会在新设计中得到回答。由于早已建立的公司大踏步跨入并行计算和强竞争者的联合力量中，而刚启动的公司则会离开该市场。第一台万亿浮点（teraflops）性能机器已建立，该工作组已经懂得如何进行加速进程以获得千万亿次浮点（petaflops）性能。电影厂商在一个很大的机群系统上生产了第一部全计算机动画制作的电影，并行的下棋程序第一次击败了著名棋手。同时，随着 Intel Pentium Pro 和非粘结高速缓存一致的存储总线的推出，出现了大量的多处理器系统。为了更好地利用存储结构，人们已经使用并行算法来改进机器的性能。当我们考虑在单个芯片内集成 10 亿个晶体管时，网络技术、存储技术、甚至处理器设计都会综合使用。

从并行计算机体系结构的未来看，一个确定的预测将会不断的变化。难以预测的改变会使并行计算机体系结构成为一个激动人心的领域并导致相关的研究。我们需要重新回顾一些基本的问题，如并行计算机的适当的组成模块如何？处理器的设计、辅助通信以及如何与处理器、存储器、互连网络集成的本质要求是什么？是否以上这些仍然使用商用桌面系统的部件，或者随着并行计算的成熟并大量进入应用后是否会产生一个新的形态？这些变化会给工业带来大量的机遇，同时也会带来挑战。

虽然很难准确地预测该领域的走向，但是本章会大致描述并行计算机体系结构发展的关键领域以及相关技术。无论市场如何发展以及出现何种技术突破，本书讨论的基本问题都是适合的。并行编程模型的实现仍然依赖于命名、定序以及同步。设计者仍然会在开销、时延、带宽以及成本之间平衡。解决这些问题的核心技术应该是有效的；然而，随着性能、成本、容量以及规模中的关键系数改变，这些技术的使用方式也会不同。基本应用工作负载的需求随着新算法的发明而变化，但是其基本的分析技术是不变的。 [935]

对于本书中给定的方法，只有建立在硬件和软件的将来潜在的发展方向上的讨论才是有意义的。对于其中的每种方法，我们都要寻求其可能的发展方向，因此，都假设了一个演变发展的基本条件；这些方法也会由于基本限制或者某突破改变其方向而导致突然停止。12.1 节考察了技术及体系结构的发展趋势；12.2 节讨论软件的需求变化如何影响系统设计的发展，以及应用的基础如何可能拓展和改变。

12.1　技术与体系结构

改变并行计算机体系结构未来的技术力量可分为三类：演变力量（过去和当前的发展趋势所揭示的）、阻碍将来的进程沿原趋势发展的基本限制、中断当前发展并形成新趋势的突破。当然，其实际的情况只能由时间来回答。本节将具体分析以上三种情况以及可能发生的体系结构变化。

为了更清晰讨论，首先考虑两个问题。在高端方面，如何实现下一次千倍的性能提高？

在中规模方面，并行系统的成本效益如何考虑？按照并行性，1998 年的计算机系统形成的并行性金字塔如图 12-1 所示。单处理器 PC、工作站以及服务器的总发货量为千万到亿的数量级。2～4 个的并行处理器的计算机市场（二级市场）是 10 万到数百万的规模。这些一般都是专用服务器，但都在朝桌面机发展。该市场从 20 世纪 80 年代到 20 世纪 90 年代早期以中等速度发展，并且随着重要的 PC 厂商制作费用较低的 SMP 机器、工作站和服务器的造价也降低以扩大销售量，该市场迅速崛起。下一级市场则被 5～30 个的处理器机器所占据。这些主要是专用的高端服务器，销售量在几十万台的规模，并稳步增长；该部分主宰高端服务器市场，包括过去被称为大型机的企业市场。在数十到上百的处理器规模上，其销售量为千量级。这些主要是支持大规模数据库、大计算量的科学应用或者大的工程开发（如石油开发，结构模型或流体动力）等的专用机器。上千个处理器系统销售量比近百个处理器系统大大减小，量级在数十左右。1990 年以来，非常高级的机器在规模上包括 1 000～2 000 个处理器。在 1996～1997 年，机器包括的处理器数量上升到 10 000 个。最明显的高端机器是专门用于高级的科学计算，包括美国能源部“ASCI”的万亿浮点机器以及 Hitachi SR2201（由日本的技术和企业部门资助）。

936

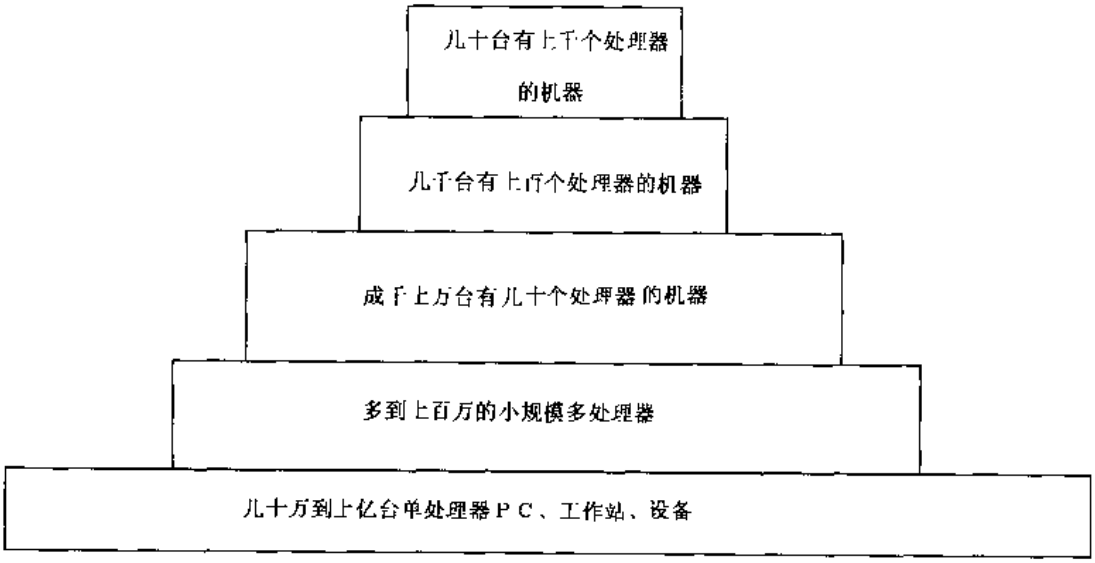


图 12-1　并行计算机的市场金字塔。最强大的机器（并行计算机）在市场金字塔的顶端，它针对要求最高的应用需求并且必须用最先进的技术

12.1.1　演变趋势

如果保持当前的技术发展趋势，则并行计算机有望经历由经济和市场力量起关键作用的发展途径。由此展开，我们预测领域的发展如何与该途径分离的可能情况。目前处理器的性能每十年增加 100 倍（或者 200 倍，如果基于 LINPACK 或 SpecFP）。DRAM 的容量每十年也增加 100 倍（3 年增加 4 倍）。因此，从目前的趋势来看，并行机器中节点计算性能与存储容量（MFLOPS/MB）的基本平衡可粗略保持稳定。尽管根据应用目标以及耗费要求该比值在当前的机器中各有不同，但是从发展前景看，容量和性能的大大增加仍然会成为将来的选择对象。按照简单的商品增长曲线，到 2010 年我们希望得到千万亿次浮点规模的性能，如果并

行性规模增加，这样的性能可提前两年得到，但是实现这样的系统要花费 1 亿美元。鉴于以下要讨论的原因，我们还不清楚这样的机器的通信性能如何。要想比通用的系统提早得到该种规模的性能将需要投资和一定规模的工程，而且这些有可能还不会实现，虽然特殊的设计会为一部分应用提前提供支持的机会。 [937]

要搞清楚将要采用的体系结构方向，基于性能的 VLSI 工艺趋势以及部件的容量趋势是很重要的。微处理器的时钟频率以每十年 10 ~ 15 倍的速度增长，而每个微处理器的晶体管以每十年 30 倍的速度增长。另一方面，DRAM 的周期时间改善得却非常慢，每十年大约以两倍的速度改进。因此说，处理器速度与存储器速度间的差距可能在继续加大。为了保持处理器的速度增长趋势，存储时间与处理器周期的比率加大会要求处理器采用更好的时延避免和容许时延技术。另外，由于周期时间和并行性的组合，处理器指令速率的增加将对存储器提出带宽要求○。

这两种因素（时延避免和高的存储带宽）以及片上存储容量的增加将导致存储层次结构更深更复杂。这两种因素会导致指令级并行性动态调度的增加。时延包容从本质上包括允许大量的指令并发执行（包括多个存储操作）。支持多个存储操作同时工作的存储系统可允许使用流水和交替方式来增加带宽。因此，VLSI 工艺的发展趋势可能会使得处理器设计与存储系统更加分离且更灵活，以适应存储系统的行为。这对于并行计算机体系结构来说是一个很好的预兆，因为处理器对于不频繁的长时延操作来说会越来越强壮。

遗憾的是，高速缓存和动态指令调度都不能减小实际的跨越处理器芯片边缘的操作时延。从演变过程看，每一层的存储结构的增加都会增加访存的开销（例如，我们知道 CRAY T3D 和 T3E 的设计者设计工作站时减少了一层高速缓存以减小存储时延，而且 T3E 中的一个两层的片上高速缓存还增加了通信时延）。这种由于层次深度而增加时延的现象是很自然的，因为设计者依赖的命中是经常发生的，提高命中率和减小命中时间比减小扑空损失会更有效。由于通信从自身特点上跨出了节点存储结构的最低层，因而越来越深的层次趋势会给并行结构带来问题。扑空损失是通信开销的一部分，不管通信抽象是共享地址访问还是消息传递。先进的体系结构和聪明的编程能够将不必要的通信减小到最小，但具体的算法还会存在不同程度的内在通信。以下还是一个开放的问题：在通过层次深度努力提高处理器性能的同时，获得有效通信需要的最低扑空损失。然而，在该方向上已有一些有用的提示。由于很多科学应用交换大量的数据组，因而具有较差的高速缓存行为。从 IBM Power2、SGI PowerChallenge 到 Sun UltraSparc 结构，人们已经注意改进高速缓存外的带宽，至少已经注意顺序访问。这些工作对数据库应用来说证明是很有效的。因此我们希望节点的存储结构能够支持高带宽，甚至演变的过程也是如此。 [938]

明显的一个提示是多线程会在将来处理器中得到应用，以隐藏局部存储器的时延。通过在单处理器中引入线程级并行，该方法进一步减小了从一个处理器到多个处理器的传输开销，因此使得小规模的 SMP 更具有广泛的吸引力。同时，这也在产生大量的时延包容上建立了长期的结构发展方向。

虽然基于改进光刻和制造工艺的 CMOS 技术发展并不会很顺利，但链路和交换带宽正在增加。链路的趋势是通过离散的工艺改变而发展。例如，铜链路已经通过一系列的驱动电路

○ 考虑到处理器和存储器之间的速度差别的扩大，比较处理器的速度和存储器访问时间时经常采用访问时间的倒数。用这种方法比较吞吐率和时延使差别看起来要大些。

引入：一次携带一个二进制位的非终止线被终止线所替代，从而多个位可在线上流水传送；这些也可能被主动平衡技术（Horowitz 1997）所代替。同时，由于连接器技术的改进链路已经越来越宽；而且电缆制造技术已经进步，允许更细的间距、更好的匹配链接，对信号扭斜提供更好的控制。过去几年，人们都认为光纤将很快会成为高速链路的选择。然而，收发器和连接器的消耗已经阻碍了其进步。随着有效的 LED 阵列（在 GaAs 技术中出现）在 CMOS 中变得富有成效，这种情况将来也许会改变。当然，消耗减小的真正驱动是数量。千兆位以太网的到来（使用光纤通道物理链路）可能最终通过提高光纤收发器的量而使成本急剧下降的最后因素。另外，高质量的并行光纤已经得到论证。因此，具有小物理截面的灵活的高性能光纤可以在不远的将来成为相当不错的链路技术。

带宽（甚至也是单处理器中要求的）以及为此而需要的并发存储处理数量正在延伸共享总线的限制。许多系统设计已经通过要求所有的部件传输整个高速缓存行来使得总线效率更高。I/O 设备的适配器按照一个高速缓存块来构造，以支持高速缓存一致性协议。这样，从本质上讲，所有的系统都是 SMP，尽管只连接一个处理器。总线逐渐地会被交换机所替代，侦听协议逐渐会被目录替代。例如，HP/Convex Exemplar 在 PA-8000 处理器中使用了交叉开关而不是总线，Sun UltraSparc UPA 在 Enterprise 6000 中 2～4 个处理器的节点上具有一个交换网络，尽管这些节点之间是用分组交换的总线连接的。基于 IBM PowerPC 的 G30 的数据通路使用了交换技术，但是仍然在寻址和侦听上使用了共享总线。SGI Origin 和 Sun Enterprise 10000 已经完全使用了交换技术。即使是使用总线，也是用了分组交换技术（阶段分离）。因而，即使从发展前景看，我们也希望能看到高性能互连网络集成于大量的设计中。这种趋势使得从大产量、中规模并行系统到大规模、中产量的并行系统的转换更具吸引力，因为这只需要很少的新技术。

939

更高速的网络也是当前 I/O 子系统的主要问题。在支持对 I/O 的改进方面已经引起了大量的关注，例如 PCI 总线已经取代了传统的商业 I/O 总线。我们非常希望支持快速的局域网，如千兆位的以太网、OC-12 ATM（62Mbps）、SCI、光纤通道以及 P1394.2。一个标准的 PCI 总线可提供大约 1 Gbps 的带宽。对于扩展的 64 位 PCI 总线，具有 66 MHz 的操作速度，将来有望获得广泛的推广；这种总线在商业机器中可提供数千兆位的性能。一些厂商正在寻找提供基于高性能网络的直接存储总线访问或分布式共享存储扩展。

从这些趋势可以断定，小规模的 SMP 将继续保持吸引力，而且机群结构以及紧密包装的商用节点集将成为大规模系统的可选项。这些设计很可能随着高性能互联网更加成熟而继续改进。我们已经看到了一种将高速缓存接口协议与高速缓存一致性协议更好集成的趋向，这样可使控制寄存器得到缓冲，DMA 可在用户级数据结构上直接执行（Mukherjee and Hill 1997）。由于多种原因，大规模设计很可能使用 SMP 节点，因而 SMP 群很可能成为并行计算的重要工具。最近随着基于 CC-NUMA 的设计出现，如 HP/Convex SPP、SGI Origin，尤其是基于 Pentium Pro 的机器，大规模的高速缓存一致性设计越来越具有吸引力。其核心问题是是否会出现一个基于 SMP 的可组装的节点，这可使得大的 SMP 群的组装本质上就像单个节点增加存储或 I/O 一样容易。

12.1.2 遇到的阻碍

如上所述，如果保持当前的趋势，并行计算机体系结构的发展前景就很光明。为什么这

或许不可能会出现呢？也许我们遇到了阻碍？这里有三种基本的可能性：时延阻碍、开销阻碍、成本或者功耗阻碍。

时延阻碍本质上是光的传输速度或者电子信号的传输速度问题。不久就会看到处理器的操作速度超过 1GHz，或者说时钟周期会小于 1ns，信号以每纳秒 1 英尺的速度传输。从发展的角度看，节点的物理尺寸不会有多少减小，它会变得很快且具有更多的存储，但是仍然还会具有由连接器连接的多个芯片和 PC 板的迹象。事实上，从 1987～1997，一个基本处理器和存储模块所占的面积并没有减小多少。有一个改进是从每板有两个节点到每板大约 4 个结点，此时第一级高速缓存移到芯片内，DRAM 芯片向 SIMM 转变，但是多数的设计在芯片外部保持一个高速缓存层。即使没有片外高速缓存（如 CRAY T3D 和 T3E），每代处理器也会消耗更多的功耗，而且需要更多的表面面积散热。因此，1 000 个处理器的机器将会具有数米（而不是英寸）的点距。 940

虽然时延阻碍的确存在，然而有几种原因来解释为什么它可能不会在可预见的将来成为并行计算机体系结构发展的障碍。一个原因是容许时延技术在处理器级数十个周期上是非常有效的。有些研究已经建议由于存储访问时延，高速缓存在单处理器中失去了有效性（Burger，Goodman，and Kagi 1996）。然而，这些研究都假设存储操作不是流水线的形式，而且处理器为 20 世纪 90 年代中期的设计。其他的研究认为，若存储操作是流水线的形式，而且处理器允许从指令窗口发射多条指令，此时转移指令的精确率将比时延更可能成为性能的限制（Jouppi and Ranganathan 1997）。当预测相当成功时，这样的设计在很多应用中能够容许 100 个周期左右的存储器的访问时间。多线程技术为指令级并行性提供了另外一种选择：可隐藏时延，甚至可隐藏非成功预测的转移开销。但是这样的时延容许技术从本质上具有带宽要求，带宽会有开销。这种开销一方面来自更高的信号速率、更多的线、更多的管脚、更多的材料，或者以上多者的组合。另外，部件所能支持的流水度是由其占用率限制的。隐藏时延特别要注意访问或通信路径上每个阶段的占用率。当占用率不能再减小时，就要使用交替技术减小有效的占用率。

从发展的道路来看，光速对时延的效果可能会受带宽的支配。目前，单个高速缓存块传送一般是几百位，由于链路相对狭窄，使用直通路由一次网络事务就可完全通过机器。随着链路加宽，一次事务的网络长度（例如，phits[⊖] 的数量）就会减小。但除非处理器同时处理多个事务进而全部占用物理时延，否则就会仍然保留很大的发展空间。而且，高速缓存块大小的增加只是分摊 DRAM 的访问开销，因而一次网络事务的长度（也是为了隐藏时延而挂起的事务数）可能随机器的发展接近一个常数。从发展的趋势看，处理器处理小于高速缓存块大小的对象时效率不是很高，因而显式消息的大小可能遵循类似的趋势。

目前很多通信时延是在网络接口上（尤其是，存储转发时延发生在源和目的节点），而不是网络本身。网络接口时延已经占据网络时延的较大部分，该时延也可能随着设计的成熟而减小。例如，考虑如何快速穿过一端或两端的网络接口。在源端，将目的文件翻译成路由以及并发地将处理器的有效数据假脱机发到网线上并不困难。然而，处理器向 NI 发数据的速度不会像 NI 向网络发数据那样快；因而，作为链路协议的一部分，保持信息还是有必要的（向网线发送空闲 phits）。这种机制已经内置在大多数的交换设备中。像 Intel Paragon、 941

⊖ 即交易平均长度和通道宽度之比。——译者注

Meiko CS-2 和 CRAY T3D 这些机器都提供穿过 NI 以及回到存储系统的流控，这样可以在没有存储转发时延的情况下实现大块传送。另外也可以开发一种通信辅助部件完成以下操作：一旦小信息（如，一个高速缓存行）开始进入网线就可以无时延传输。

在目的端消除存储转发时延具挑战性，这是因为，通常数据在完全接收和检查之前是无法确定其完好性的。如果直接暂存在内存，就会形成堆积。但我们观察到地址的正确性比内容的正确性更重要，因为我们不想将数据暂存在内存中的错误位置。信息头可以加一个分离的校验和。目的节点上信息头在信息暂存入内存前被检验。对于较大的数据传输来说，一般包含一个完成事件，这使得在数据被标志为“到达”之前缓存在内存并校验。注意，这意味着通信抽象不应当允许应用在大块数据传送中间通过抽取数据的值来推测传送是否结束。对于小的数据传送，可采用一些小的技巧推测地将数据移入高速缓存。本质上，高速缓存分配一行然后传送数据，但是如果没有正确地进行校验，高速缓存行的有效位就永远不会设置。因而，在设计通信辅助部件和存储系统时应特别注意通信事件，但是提出一种比目前更为流畅的网络事务技术来减小时延是可能的。

并行计算机没有受到根本的时延的阻碍的主要原因是整个的通信时延仍然被额外开销所主宰。时延将仍然存在，但是只占实际通信时间不大的份额。其原因在于当前工业界的设计过程。当处理器具有一层或多层高速缓存、一层片外高速缓存时，自然就会产生存储系统；在设计存储结构中的某层的控制器时，设计者就会遇到一个问题：靠近处理器的一方速度快，靠近存储器的一方速度慢。设计的目标是使得对于交付给处理器方的高速缓存的典型地址流 S，以下的表达式最小：

942

$$\text{平均访存时间}(S) = \text{命中时间} \times \text{命中率}_S + (1 - \text{命中率}_S) \times \text{扑空时间} \quad (12\text{-}1)$$

由于任何部分的很大改进都会损害其他部分，以上设计目标就出现了内在的权衡问题。因而，在设计空间的每个方向上，理想设计是极端间的折中。通常命中时间是由其最快部件的目标频率决定的，这也给其他设计部分建立了优化极限——设计者要在该极限内保持命中时间。我们可以在高速缓存结构上采取改进措施（例如，增加互连度等）以提高命中率，但只需能够达到命中时间的期望值就可以了（Przbylski，Horowitz，and Hennessy 1988）。并行体系结构的关键部分涉及到扑空时间。设计者要想减小扑空时间究竟有多困难呢？其中一个原则是使得两部分大致相同。这就保证其优化程度在两倍的范围内，而且较实用。关键问题是，由于在单处理器中扑空率很小，因而扑空时间可以是命中时间的数倍。对于第一层高速缓存其命中率要超过 95%，因而其扑空时间可以是命中时间的 20 倍。在更低的高速缓存层，扑空时间和命中时间也差一个数量级。扑空时间的一个重要部分是到较低存储级的传输时间，对其较小的改进只能对单处理器的性能产生不大的影响。高速缓存的设计中可以在多种用途上利用此自由度。例如，增加高速缓存行的大小可以改进命中率，其代价是增加扑空时间。

另外，存储结构中的每层都要增加数据传送的开销，因为数据必须经过所有层的接口。为了模块化设计，接口应使得两边的操作是松耦合的。在片高速缓存间的握手会有一些开销，片内高速缓存与片外高速缓存间的接口开销会大一些，然而经过存储总线的复杂协议会具有更大的开销。另外，对于通信来说，也有一些与接口本身相关的协议。以上这些效应的累计就形成了实际的通信时延总是光速的很多倍。设计者对通信的自然反映总是增加传输的最小数据量，例如，增加最小信息块的高速缓存行大小。这就将关键时间从时延移到占用时

间（occupancy）。如果每次传送的数据量足够大以至可以分摊开销，则此时的光速时延将又成为不大的附加部分。

当设计分成多层的存储结构，而且强调与处理器一侧访问流相关一层的最大化，设计者的这种自然倾向就会导致处理器到通信辅助部件的每一层都要增加开销。为了接近光速传输时延的极限，在处理器设计、高速缓存设计与存储器设计中应该建立一种不同的设计方法。该方法的一种体系结构趋势是使用扩展的无序执行或者多线程隐藏时延，这甚至可用于单处理器中。这些技术改变了高速缓存设计者的目标：并不是最小化式（12-1）的和，而是从本质上最小化每一部分。 943

当扑空发生时，处理器并不等待服务，而是继续执行而且发出更多的请求，这些请求很多都会命中。同时，高速缓存继续忙于扑空处理。我们希望当另一个扑空产生或者处理器执行完其他并发任务时，此次的扑空已经完成。即使扑空需要一定的时间处理，其检测和发出处理服务也不应花费过多的处理器周期。事实上，扑空处理本质上要求在命中时间预算内进行。

而且，为了使处理器繁忙，可能需要允许保持多个在线请求，这在第 11 章中有全面的解释。对于每个扑空来说，其扑空时间也许都不满足式（12-1），或者因为时延或者因为开销。扑空与通信事件的趋势是集群化，因而需要服务的操作间隔比平均水平越来越小。

Little 定律建议了另一种潜在的障碍——成本障碍。该定律认为，若需要隐藏的总的时延是 L，长时延请求的发生率为 ρ，则当该时延被隐藏时每个处理器需要的在线请求数是 ρL；当考虑集群通信时，该数量会更大。当这些通信事件都在进行时，P 个处理器的需要的网络带宽为 $P\rho L$（P），L（P）反映了随机器增大而产生的时延增加。这就需要建立网络成本的下限。为了提供该网络带宽，网络的总带宽需要更高（第 10 章讨论的）；因为通信时会有突发、冲突等等。因而，为了保持发展的趋势，系统设计的很多方面都需要考虑容许时延，而且在网络技术上需要考虑带宽和成本的改进。

12.1.3 潜在的突破

我们已经看到了并行体系结构的美好的发展前景，当然在前进的道路上也会出现阻碍其发展的一些“乌云”。是否也会有“银色的闪电”呢？是否存在形成新的并行计算机结构设计的可能性呢？回答当然是肯定的，但是这些具体的方向还不确定。尽管技术上可能发生戏剧性的变化，如量子器件、自由空间光互连、分子计算或者纳米级机械设备等正在研究中，但是在常规 CMOS VLSI 设备上尚存潜在的发展空间（Patterson 1995）。集成度连续发展这一简单的事实就很可能给并行计算机设计带来一场革命。

从学术的角度看，很容易低估集成过程中一些封装限度的重要性，但历史说明这些因素的确令人关注。集成度限度的通常效果说明如图 12-2 所示，该图显示出两个量化趋势。直线反映出系统集成度的水平随时间稳定增长，与之交叉的曲线描述的是系统设计中的革新数量。不管技术的发展，一个设计方案应该在一段相当长的期间内是稳定的，但是当集成度超过了某一关键界限时，很多新的设计选择成为可能，新的设计也会随之出现。该图显示了计算机体系结构历史上的两个时代。 944

回忆从 20 世纪 70 年代到 20 世纪 80 年代期间，计算机系统设计经历了一个稳定的发展道路，并且显示出清晰的阶段：小型计算机，在工程和学术市场被 DEC Vax 主宰；大型机，

在商业市场中由 IBM 主宰；向量超级计算机，在科技市场被 CRAY 研究院主宰。由于 MSI 和 LSI 部件，尤其是半导体存储器允许在相对小的工程量下可以设计复杂的电路，这在早期打破了技术屏障，作为结果小型计算机发生了突破性变化。特别是，此时的集成水平已经可以允许使用微程序技术支持大量的虚拟地址空间和复杂的指令集。大型机很早就开始发展。向量超级计算机的成功反映了转换的结束。其完美的 ECL 电路设计，在简洁的装载-存储体系结构中与半导体存储器耦合清掉了早期很特殊的并行计算机设计。这三个主要阶段经历了可预言的演变方式，每个阶段具有各自的市场。与此同时，微处理器从 4～8 位，进而发展到 16 位，这就导致了个人计算机和图形工作站的产生。

945

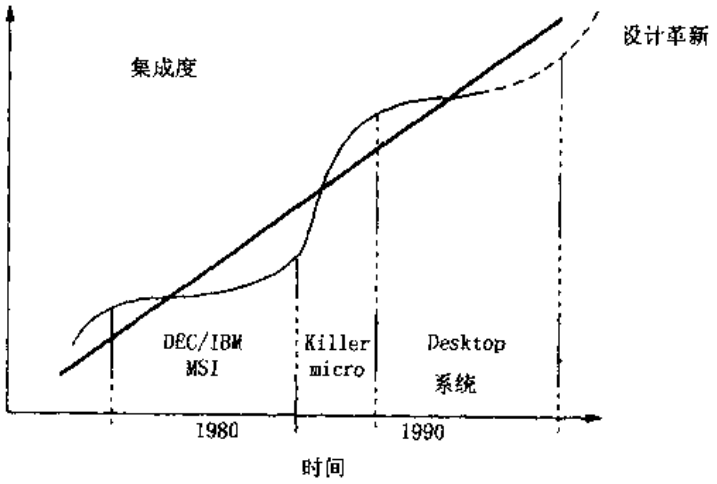


图 12-2 变化的潮流。计算的历史是在稳步发展和快速创新的交替中前进的。关键技术通常不是突然发生的；它有出现和演化的过程。"革命"的发生只是当技术越过了一个关键的台阶。其中一个例子是在 20 世纪 80 年代中期 32 位微处理器的到来，它突破了集成度较低的小型计算机和大型机的稳态，促成了计算机设计的一次复兴，包括片上的低层并行性和多处理器的高层并行性。20 世纪 90 年代后期，这个转变完全成熟了，基于微处理器的台式机和服务器技术统治了市场的各个方面。在并行计算机设计上有一个很明显的融合。然而，集成度在继续提高，不久单片机将会像 20 世纪 80 年代的单板机那样自然。问题是这将引起一场什么样的设计复兴

在 20 世纪 80 年代中期，微处理器到达了一个关键技术门限：一个全 32 位微处理器可全部放于一个芯片中。一瞬间，整个画面完全改变：一个具有一定性能和容量的完整计算机放在了一个电路板上，多个这样的电路板组成一个系统。基于总线高速缓存一致性的 SMP 出现于很多小公司，包括 Synapse、Encore、Flex 和 Sequent。较大的消息传递系统出现于 Intel、nCUBE、Ametek、Inmos 和 Thinking Machines 公司。同时，也出现了数个小型计算机厂商，包括 Multiflow、FPS、Culler Scientific、Convex、Scientific Computing System 和 Cydrome。其中几个公司随着新的稳定状态建立遭到失败。工作站以及后来的个人计算机吸收了小型计算机的技术计算市场。SMP 替代了大规模数据中心、事务处理、工程分析，从而替代了超小型计算机。向量超级计算机让位于大规模并行基于微处理器的系统。自那时起，设计的发展开始稳定。对高速缓存一致性技术的深入理解可以允许大规模地对共享地址支持。可扩展的低时延网络从 MPP 到常规的局域网或者计算机房环境的转化，已经允许以很低的代价随意地建立起 PC、工作站或 SMP 的机群系统来提供潜在的性能，尤其是可以作为一个个人超级计算机。几个特殊的大型计算机也可以组成不同大小的共享存储机器的机群系统。本书很显然是集中在发展上，并行计算机设计者面临的基本问题是如何集成商业的部件，并不是使用何

种部件。

同时，微处理器和存储器的集成度正在接近一种新的关键门限：一个完全的计算机系统可集成在一个芯片内，而不是一个主板上。到 20 世纪末，微处理器的集成度正在达到 1 亿个晶体管。到 21 世纪初，千兆位的 DRAM 将会出现，像 20 世纪 80 年代中期的 32 位处理器、20 世纪 70 年代中期的半导体存储器、20 世纪 60 年代中期的集成电路那样，这一新的门限将可能带来新的设计复兴。从本质上看，处理器芯片和存储芯片的巨大差别将减小，而且多数的芯片都会含有逻辑电路和存储器。

很容易列举数条原因来说明为什么会发生处理器和存储器级的集成，而且会导致计算机设计的重大变化；尤其是并行计算机设计。对于这种新的设计空间，已经投入多个研究计划，分别具有不同的缩写（PIM、IRAM、C-RAM 等）。为了避免与这些缩写名词混淆，我们称处理器和存储器的概念为另一个缩写词——PAM。只有历史才能告诉我们哪种老的结构思想会获得新生，哪种完全新的思想会来到。先看一看主导新兴设计的一些技术因素。 946

一个明显的因素是微处理器芯片中大部分是存储器。其中使用 SRAM 作为高速缓存，但仍然是存储器。表 12-1 显示了用作高速缓存及存储器接口（包括存储缓冲等）的晶体管和面积所占的百分比，其中包括两个厂商的 4 个最近的处理器（Patterson et al.1997）。实际的处理器是微处理器芯片上一个小而且是进一步缩小的部件，即使处理器变得越来越复杂。图 12-3 非常明显地揭示了这种趋势，其中显示了过去十年几个微处理器中高速缓存占用的晶体管数（Burger 1997）。

表 12-1 微处理器芯片上用于存储器的百分比

年份	微处理器	片载缓存大小	晶体管总数	用于存储器的百分比	芯片面积（mm^2）	用于存储器的百分比
1993	Intel Pentium	I：8 KB D：8 KB	3.1M	32%	~300	32%
1995	Intel Pentium Pro	I：8KB D：8 KB L_2：512 KB	P：5.5M + L_2：31M	P：28% + L_2：100% （Total：88%）	P：242 + L_2：282	P：23% + L_2：100% （Total：64%）
1994	Digital Alpha 21164	I：8 KB D：8 KB L_2：96 KB	9.3 M	77%	298	37%
1996	Digital Strong-Arm SA-110	I：16 KB D：16 KB	2.1 M	95%	50	61%

注：I：表示指令，D：表示数据，P：表示处理器晶片，L_2：二级高速缓存

很大部分的芯片面积甚至更大部分的晶体管是用作数据存储，而且组成在片的多层高速缓存结构。这些在片上的存储的投入是需要的，这是由于访问片外存储时间的缘故，该时间亦即芯片接口、片外高速缓存、存储总线、存储控制器和实际 DRAM 的时延。对于很多的应用，提高性能的一个重要方面是增加在片上存储的数量。

这种技术趋势的一个机遇是可将多个处理器放于一个芯片内。由于处理器只占芯片实际体积的一小部分，因而只增加一小部分就会大大提高潜在的峰值。由于复杂性所带来的性能

回报逐渐减小，这种方法的论据也越来越充分；例如，花费在寄存器端口、指令预取窗口、相关危险检测以及旁路，都会按照每周期发送的指令数超线性增长，当超过4路的超标量时性能改进却非常小。因而，对于相同的面积，可选择非激进的多处理器设计（Olukotun et al.1996）。这也启示我们对20世纪80年代中期已经发展很长时间的SMP技术再重新考察。

947

多数早期的机器共享第一层片外高速缓存，于是就会有空间给分离的高速缓存，L_1 高速缓存移入芯片；而有时共享 L_2，有时并不共享 L_2。对于板级和芯片级的多处理器系统，许多基本的平衡问题仍然是一样的：共享离处理器近的高速缓存可允许细粒度的数据共享，而且消除了其他层高速缓存的一致性支持问题；但是由于经互联网与共享高速缓存相连，势必增大访问时间。任何层的共享都可能发生相互干涉，其正负影响依赖于应用程序使用数据的模式。然而，板级设计很大程度上取决于可用部件的特殊性质。对于多个处理器在同一芯片的情况，所有基于一个同构介质的设计选择都可以考虑。另外，由于不同的消耗和性能特点，具体的折中也具有不同的方法。假设出现了许多廉价的SMP，尤其是带有粘结性高速缓存一致性支持，单芯片上的多处理器则是多处理器发展的自然选择。

948

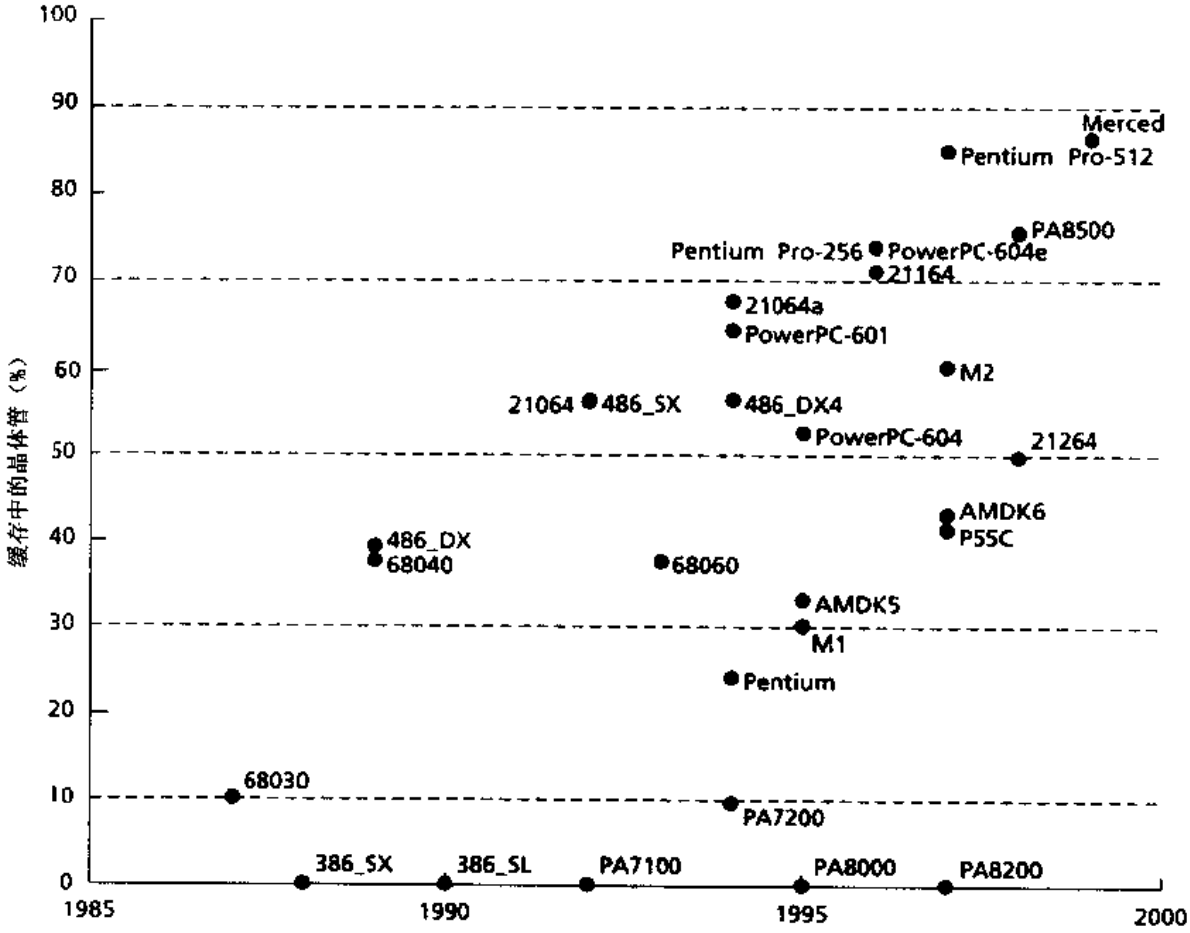


图 12-3 在微处理器芯片上用于高速缓存的晶体管的份额。自从20世纪80年代中期高速缓存移到了片上，在商用微处理器上用于高速缓存的晶体管所占比例越来越高。尽管处理器很复杂并且开发了很强的指令级并行性，只有通过提供大量的本地存储并且开发局部性，才能满足处理器对数据的带宽要求

一个更为激进的建议是：最接近处理器的是DRAM而不是SRAM存储器。传统上，SRAM与处理器使用相同的工艺，而DRAM则使用完全不同的工艺。从传统上讲，微处理器

和 DRAM 的工程要求也是非常不同的。微处理器的制造趋向于高时钟频率，而数据通路和控制器之间采用多层金属的丰富互连性结构；而 DRAM 结构强调密度高且消耗小，其封装也是非常不同的。由于要求带宽，微处理器使用较多的管脚，其封装较为昂贵，而且其材料产生较大的热量。DRAM 封装具有较少的管脚、低的开销，而且很适合低功耗的 DRAM 电路。然而，这些不同也在逐渐消失。DRAM 的制造过程也正逐渐适合处理器实现，具有两层或 3 层金属以及较好的逻辑速度（Saulsbury，Pong，and Nowatzyk 1996）。

把逻辑集成于 DRAM 的动力一方面来自需要，另一方面是由于机会。容量的巨大增加要求 DRAM 的内部结构和外部接口应作相应的改变。早期的 DRAM 设计由一个方形的位阵列组成；其地址由行列两部分组成，一次可以读出一行，然后再选择列。随着容量的增加，需要将多个小阵列放在一个片内，而且需要在多个阵列和管脚之间增加互连。另外，由于管脚有限而且还需要增加带宽，这就需要一部分 DRAM 以较高的频率工作。很多现代的 DRAM 设计，包括同步 DRAM、增强 DRAM 以及 RAMBUS，更有效地利用了 DRAM 片内的行缓冲，而且在行缓冲和管脚之间提供高带宽传输。这就要求 DRAM 也能够处理逻辑。同时，也有很多机会将新的逻辑函数与 DRAM 结合，尤其是支持图形的视频 RAM。例如，3D-RAM 在提供帧缓冲的视频 RAM 中直接支持 z－缓冲逻辑操作。

将处理器与 DRAM 集成的吸引力是一种开始的现象。虽然处理器设计受芯片面积的限制，但也没有理由不使用高速处理器的制造工艺；而且虽然存储芯片较小，但一些系统中用的如此之多，以至于无法裁定与处理器结合而造成的增加的开销。然而，DRAM 的容量比晶体管数（也可以说处理器的使用面积）以更快的速度增加。在千兆 DRAM（也许是下一代）时代，处理器的开销增长不会很大，大概为 20%。从处理器设计者的观点来看，DRAM 比 SRAM 的优点是在密度上超过一个数量级。然而，访问时间长且受限制，同时要求刷新（SaulsburyPong and Nowatzyk 1996）。 949

或许对门限的敏感会进一步增加 PAM 的吸引力。DRAM 容量的增加已经超出了应用程序在存储上的要求。由于更大的存储容量可以执行较大的应用程序，因而存储的容量增加已在高端机器上获益：这可以减小通信计算的比率，而且使并行计算更有效。在低端，其效果是减小了每个系统的存储片数量。当只需要少数的存储片时，具有较少管脚的传统存储接口不能很好地工作，因而本质上需要新的具有高速逻辑的存储接口。当典型的系统具有一块存储片时，这时就可以将两者集成到一个芯片，省去了两者之间的所有相关问题。然而，这就会带来一个问题：在大系统中的存储结构如何组织。

技术演变中关键门限产生的影响是新技术因素和市场变化。其中之一是高速的 CMOS 串行链路。超过 1 Gbps 的串行链路的标准单元已经实现，而且更高的速率已经在实验室中得到论证。在以前，这样高的速率只能通过非常昂贵的 ECL 电路和 GaA 技术实现。高速的链路使用少数的管脚提供集成 PAM 芯片到大系统中的效能成本合算，而且可以形成并行机互连网的基础。第二个因素是可配置逻辑的发展以及广泛使用。这样可能将处理器、存储器和非配置逻辑集成到一个芯片内，这可以配置适合不同的应用。最后一个因素是基于网络应用市场快速增长的低功耗处理器，有时称为 WebPC、Java 工作站、掌上电脑或其他高级电子设备。对于这些应用，中等芯片的 PAM 提供大量的处理和存储容量。这些极大的市场可能使 PAM 比桌面系统更能成为商业组建的“积木”。

这些技术机会面临的问题是计算机节点如何改变其组织结构。图 12-4 显示了基本的出

发点，其中显示即使内部具有宽通道以及增加时延，因为管脚和连线非常昂贵，在处理器和存储器位阵列之间的子系统提供的接口是狭窄的。在 DRAM 片内，其数据通路是非常宽的。比特阵列本身是封装非常紧的沟道式电容器，因而很难有所作为。然而，比特阵列和外部接口之间的缓冲是非常宽、不紧密，而且本质上是 SRAM 和逻辑。当读 DRAM 时，一部分地址用来选择行，该行被读入数据缓冲区；同时另外一部分地址用来选择数据缓冲区中的一部分比特。由于读操作对比特具有破坏性，因而该缓冲区还要写回。在写操作时，先读一行，在缓冲区里修改某些位，最后写回该行。

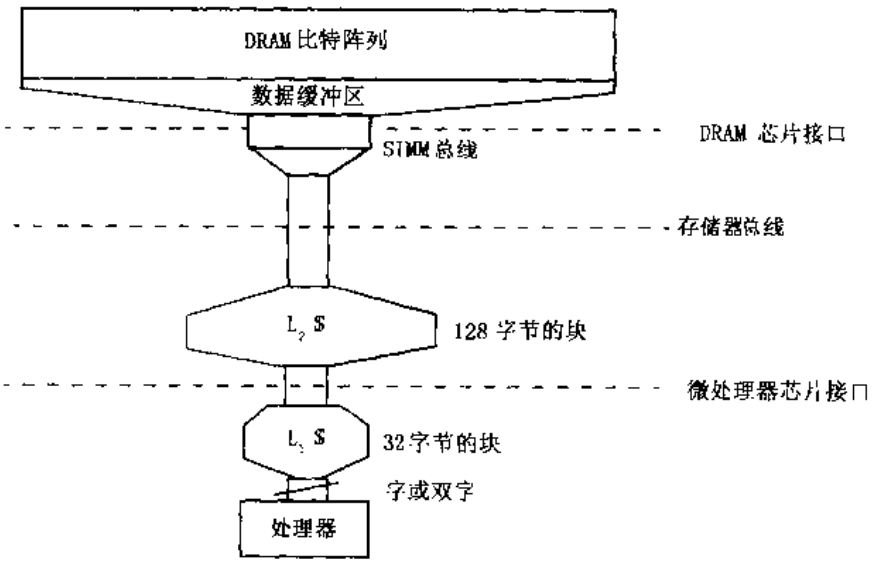


图 12-4 跨越一个计算机系统的各种带宽。处理器数据通路为几个字宽，通常和它的 L_1 缓存的接口为两个字。L_1 缓存的块是 32 或 64 字节宽，它们的制约条件是处理器每次一个字的处理方式和微处理器芯片的界面。L_2 缓存更宽些，它们的制约条件是处理器芯片的接口和存储器总线接口。形成一个存储模块的 SIMMS 可能有一个接口，比存储总线要宽但要慢。在内部，这是一个很宽的数据缓冲区，直接在实际的位阵列之间交换数据

950 目前的研究主要面向三个基本的重组可能性，每种都有一定的历史，而且能被理解、探究以及根据本书提供的基本设计原理进行评价，这里只是简单的概述，但希望读者能查阅最近的文献和 Web 网页。

第一种选择是在与常规 DRAM 片数据缓冲区逻辑上相关的加入简单的专用的处理单元，如图 12-5 所示。这种方法称为存储器中的处理器（processor in memory）（PIM）（Gokhale, Homes, and Iobst 1995）和计算 RAM（Kogge 1994; Elliot, Snelgrove, and Stumm 1992）。其基本上是一个数据并行操作的受限类型的 SIMD 处理。一般来说，这些将是小的比特串处理器并提供基本的逻辑操作；但它们也可以一次处理多个比特甚至一个字。正像我们在第 1 章所看到的，这种方法已经在并行计算机结构的历史上出现过数次，一般是在一种技术转换的开始而通用处理器并不很适合时出现，因此说特殊的操作也同样具有通用的性能优点。而每当该方法证明适合有限的操作时（经常是图像处理、信号处理或密集的线性代数），就会根据技术的发展让位于更通用的解决办法。

举例来说，在 20 世纪 60 年代早期就出现过数种 SIMD 机器并试图允许只通过复制功能
951 部件和共享单个指令序列来得到高性能机器，包括 Staran PEPE 和 Illiac。早期的计划在伊利

诺斯大学开发 Illiac IV 时达到了鼎盛，该机器开发了许多年，只是比 CRAY-1 提前几个月运行。在 20 世纪 80 年代早期，这种方法又在 ICL DAP 中出现，该机器提供一个压缩的处理器阵列，在 20 世纪 80 年代中期获得很大进展，此时处理器芯片已经大到足以支持 32 位的串行处理器而不是一个全 32 位处理器。一些机器如 Goodyear MPP、Thinking Machines CM-1、MasPar 都是在这种机遇下出现的。从后两种机器比前几种更成功的例子来看，其关键是在多个单元间提供通用的互连网，并非只是一个低维的网格（该结构显然重建起来很便宜）。Thinking Machine 也能够抓住单片浮点单元的到来而改变了 CM-2 设计，进而提供 2k 个 32 位 PE 而不是 64k 个 1 位 PE。然而，这些设计本质上是 Amdahl 定律的基本挑战，因为高性能模式只能用在应用中适合特殊操作的情况。几年内，导致了具有几千个通用微处理器的 MPP 设计，这种设计可执行 SIMD 操作以及更通用的操作，这可在更多的时间内利用并行性。PIM 方法在 CRAY 3/SSS 使用（在该公司申报第 11 法案之前）以为国家安全局（美国）提供特殊的支持。对于更多传统的技术、文献（Aimoto et al.1996; Shimuzu et al.1996）已有说明。

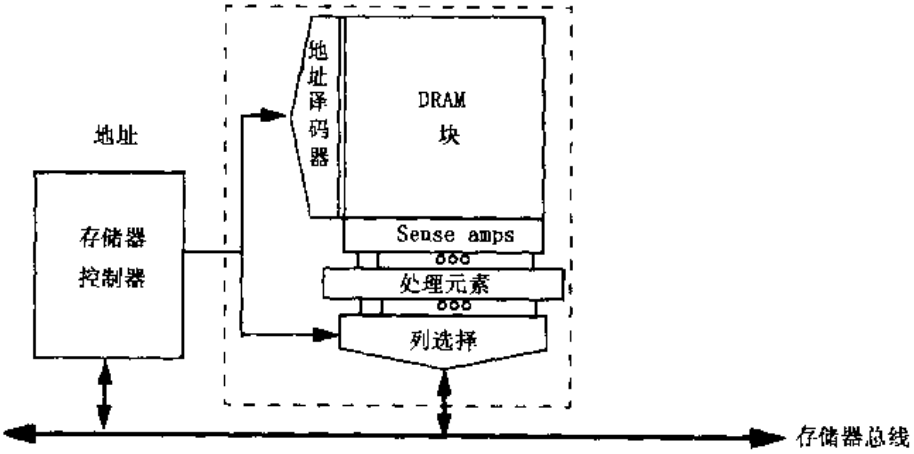


图 12-5 存储中的处理器的组织方案。简单的功能单元（处理单元）和典型 DRAM 芯片的数据缓冲安排在一起

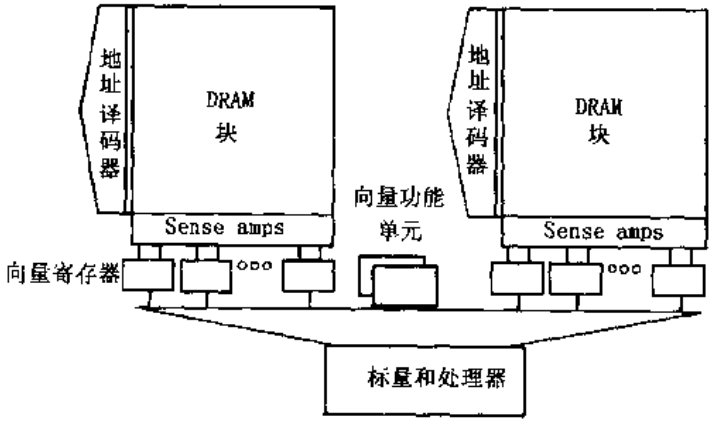


图 12-6 向量 DRAM 的组织：将 DRAM 数据缓冲区增强，以提供向量寄存器。向量寄存器可以和相应的功能单元一起构成流水线。片载存储系统和向量支持连接到一个标量处理器上，该处理器还可能有它自己的高速缓存

第二种重建选择是加强对 DRAM 体中数据缓冲区的支持，以作为向量寄存器用，如图 12-6 所示（Patterson et al. 1997）。当使用宽的比特阵列时，在 DRAM 行与向量寄存器之间就

952 可实现宽带传输，但是对向量寄存器的算术运算是通过数据流经少数传统的功能部件实现的。这种方法也具有大量的历史依据，包括非常成功的 CRAY 机器以及其他未成功的尝试。CRAY-1 的成功部分与一年前未成功的 CDC Star-100 向量处理器形成鲜明对照。Star-100 在内部只对短数据段进行操作，而且其数据是从存储器到临时寄存器。然而，它只能对连续（跨距为 1）的向量操作，而且其主存具有长的时延。CRAY-1 的成功不只是向量寄存器的应用，而且也是与快速半导体存储器的结合；倘若在向量寄存器和存储器之间具有通常的互连网，就可以执行非单位跨距和后来的集聚/散列操作，还有一些标量和向量操作的有效耦合。换句话说，就是它提供了低时延和高带宽的访问模式和有效同步。后来新技术在这种设计类型上的努力并没有完全在这方面吸取教训，包括 FPS-DMAX（具有向量在存储器中的线性组合）；Stellar、Ardent、和 Stardent 向量工作站；对 SparcStation 的向量扩展 Star-100；CM-5 存储控制中的向量单元；Meiko CS-2 中的向量功能单元。我们总是着眼于峰值性能以及特殊情况的低开销，但是如果启动时间很长、寻址能力差或者与标量访问的交互性弱，则使用扩展部分的时间会急剧下降，而且并不是通用的解决方法

第三种选择是寻求通用的设计，而不仅仅是移去与不同处理器封装、片外高速缓存和存储系统有关的抽象层。与 DRAM 有关的数据缓冲基本上是作为最后一层高速缓存。由于高级 DRAM 设计从本质上使用数据缓冲作为高速缓存已经有数年，因而这种缓冲技术非常具
953 有吸引力。然而，在过去 DRAM 由于受狭窄的片外高速缓存接口以及存储总线的限制而被束缚。在更加集成的设计中，DRAM 数据缓冲区高速缓存可与一个或多个在片上处理器具有直接的接口。这种方法对基本的高速缓存设计折中在某种程度上有了改变，但传统分析技术仍然适用。长的存储行开销由于数据缓冲和比特阵列间的并行执行而减小。当前与 DRAM 靠近的逻辑开销对相联度产生更多的成本。因而很多方法使用直接映射的 DRAM 缓冲区高速缓存并采用“牺牲者”缓冲区或者类似的技术来减小冲突扑空率（Saulsbury，Pong 和 Nowatzyk 1996）。当这些集成设计被用作并行机器的组块时，如果出现空间局部性，就会观察到预期的结果：随着数据预取的改进，长的高速缓存块会导致伪共享增加（Nayfeh，Hammond，and Olukotun 1996）。

当 PAM 方法从学术的、基于模拟的研究转到实际的商业开发时，我们希望看到比设计过程的改变更深远的影响。高速缓存设计者的工作不再是对一边是快速的接口，另一边是慢速接口的通路中进行某一步的优化。其界限将被打破，其设计优化成为端到端的问题。我们已经看到集成通信辅助设备的所有可能性都正在出现：在处理器级集成、集成进高速缓存控制器、集成进存储控制器；而在 PAM 中，可采用混合解决方法而不是只局限于系统中的某一部件。

然而，随着集成处理器和存储器的详细设计出现，这些部件可能为大规模并行机提供新的通用的组块。显然，这些组件可以连结成分布式存储的机器，而且通信辅助部件也能在剩余的设计中很好地集成，因为这些组件都在一个芯片上。另外，互连管脚可能是惟一的外部接口，因而通信效率就显得更加重要。显然我们可以在比以前更大的规模上组建并行机器。自最早的计算机出现以来，一个实际中的限制是大规模系统的总构件数在 10 000 以内。更大规模的系统已经建立，但是显得难以维持。在早期，系统的规模为 10 000 个真空管，然后是 10 000 个门，10 000 个芯片。最近，一些大机器已经具有大约 1 000 个处理器，而且每个处理器大约 10 个组件，或者是芯片或者是存储器 SIMMS。第一个万亿浮点的机器拥有大约

10 000 个处理器，每个处理器具有多个芯片，因此我们正在注视着这种模式是否会改变。单个芯片上的完全系统最好是将来的商业组件，这种组件在多种智能应用中的使用量可能大大超过桌面系统，其成本非常小。无论如何，能希望随着单个芯片上的处理器个数增加而得到更大规模的并行性。

一个非常有趣的问题是当一个程序需要访问的数据超出 PAM 时，应该如何解决。一个方法是提供传统的存储接口扩展。另一个更有趣的选择是简单地提供到其他 PAM 片的高速缓存一致性访问。这种技术人们已经很熟悉，因为提供这种功能需要较小的硬件支持。如果 PAM 的处理器构件确实很小，那么即使系统中只使用一个进程，在靠近存储器的地方增加一个处理器的开销会很小。由于并行软件应用越来越普遍，因而当应用需要时就会提供额外的处理能力。 954

12.2 应用程序和系统软件

很明显，在将来的并行计算机体系结构中，并行软件及软硬件的相互作用会变得越来越重要。可以从以下五个方面来看并行软件：应用程序、编译器、编程语言、操作系统及工具。并行软件的改变和其他所有的改变一样，有三个阶段：发展、碰壁、突破。

12.2.1 演变趋势

尽管数据管理和信息处理似乎成为多处理器结构中的主流应用，但科学工程方面的应用往往需要更高性能的计算。早期的科学计算主要集中在对一些具有相当规则计算与通信特性的物理现象的建模，这样，对问题的简单划分就能很好地利用多处理器带来的性能，就像早期的工作者可以有效地利用向量体系结构一样。随着对主流应用和并行计算的理解的不断深入，早期的工作人员开始构造更复杂的、动态的以及适应性的模型，这些模型适用于大多数的物理现象，这使得应用程序具有不规则的和不可预知的特点。人们希望这方面的研究能够持续下去，这样随之而来的复杂的情形可以更有效地利用并行性。

由于多处理技术越来越普遍地被研究和使用，它的应用领域也随之不断发展，这些应用很多已经与原来大型机处理应用相关。大量在金融和逻辑学遇到的优化问题——例如决定商业航班的人员日程——需要付出巨大的代价才能解决，而这些问题常常对企业利益至关重要，并且这些都是并行性计算应该解决的问题。人工智能领域发展起来的解决问题的方法，包括搜索技术和专家系统，在许多领域都有其实用价值，这些方法也极大得益于计算能力和存储技术的提高。在信息管理方面，一个很重要的方向是如何使用数据挖掘和决策支持技术（对后者，一些复杂的查询用于为决定一些方向提供重要的决策基础），更好地从大量数据中提取有用信息。这些应用程序往往对计算有强烈的要求，就像对数据库有强烈的要求一样，并且对有关计算和数据存储/检索相结合以建立计算及信息服务利益。在诸如科学计算研究领域，类似的问题越来越多地出现。例如，操作和分析极为大量的生物链信息，这些信息现在通过基因及排序科学能够快速得到；还有，对发现的数据进行计算以对三维结构进行估计。Internet 和 WWW 的出现和发展带来的大规模分布式计算的迅速发展和现代社会中存在的大量冗余信息使得计算与信息管理的结合更加不可避免地成为未来快速发展的一个方向。多处理器已经成为 Internet 搜索引擎和 WWW 查询的服务器。将来的查询要求信息服务器能满足各个方面的要求，从一次只有少量的、复杂的数据挖掘和决策支持类的查询到数 955

日巨大的并行的查询，在使用家庭计算机或手提信息设备作为客户端的时候会出现这种情况。

随着网络通信特性的改善和不同耦合度并行性相耦合的需求日趋强烈，可以看到并行和分布式计算更强烈的结合。分布式计算中的技术将被应用于并行计算以建立大量的信息服务器，这样可以获得更好的性能，同时，并行计算技术在另外一些方向中会使用分布式系统作为平台来并行地解决一个独立的问题。多处理器仍然会充当服务器的角色——数据库和事务服务器、计算服务器以及存储服务器——尽管这些服务器操纵的数据类型正变得越来越丰富，差异也越来越大。从仅包含文本信息的记录，直到现在图像、现实物体的三维模型、视频和音频等各种信息都可以被存储、索引、检索和分配。在查询中对数据类型的匹配往往是近似的，提供这些数据时经常采用高级的压缩技术以节省带宽，这两者都需要大量的计算。

最后，图像、媒体及从传感器得到实时数据日益重要——应用于军事、社会、娱乐等方面——将导致多处理器对数据的实时计算愈发重要。与原来的从存储系统中读出数据、对数据进行操作然后写回存储器不同，这要求对数据在从输入到输出数据端口的机器整个路径上都进行操作。如何使处理器、存储器和网络能像高速缓存一样很好的协同工作，必须要在这个发展阶段再次考虑。随着应用空间的增长，我们将逐渐知道哪类应用能真正使用大规模的并行计算，以及对其他的应用并行性的规模应该是多大。还将知道是否按比例增长的通用体系结构适合最高端的计算或是否这些应用的特点确实相差很大以致需要差别很大的资源，并且这些资源的集成是有好的性能价格比算的。

由于并行计算被应用到越来越多的领域，我们开始注意到可移植的封装好的并行应用可以用在许多应用领域中。一个很好的例子就是称为 Dyna3D 的应用程序，这个程序使用一种称为有限元素的方法技术，求解计算高度不规则问题中的偏微分方程。Dyna3D 最初是在国家研究实验室为武器建模系统而开发的，系统的平台是价值几百万美元的进口超级计算机。冷战结束后，这项技术进入商界，运行于流行的并行机上，成为汽车工业界以及其他一些领域中碰撞建模的支柱。当本书还在写作过程中时，这项标准正广泛应用于性价比良好的并行服务器对日常器械进行模拟，例如，当蜂窝电话掉线时会有什么事情发生。

[956]

导致并行应用程序广泛应用的一个主要因素是在大范围计算机上实现编程模型的可移植库。随着消息传递接口（MPI）的出现，它已成为消息传递机制的一个实现实体，并应用于 Dyna3D：针对不同问题的 Dyna3D 应用程序通常在差别非常大的平台上运行，从用于消息传递的计算机（例如 Intel Paragon）、用硬件支持共享地址空间的计算机（例如 CRAY T3D）到网络工作站。共享地址空间的编程模型在不同的通信体系结构性能特征上的可移植性还没有实现，不过这很快就要完成了。

由于很多应用程序既用共享地址空间模型实现，又用消息传递模型实现，我们发现能将并行程序的算法很好地分成两类，一类更富于技巧性、更依赖体系结构，另一类相对来说更少依赖于编程模型和体系结构。例如星系模拟和其他分层 n 体计算（Barnes-Hut）。较早的消息传递并行程序用正交递归对分法将计算域分给多个处理器，并且不维护一个全局的树数据结构。另一方面，基于共享地址空间的程序则使用全局树和不同的分解法将计算域做很不同的划分。随着时间的流逝和消息传递通信结构的发展，消息传递逐渐采用了与共享地址空间类似的分解算法，包括使用哈希算法建立、保持逻辑全局树。在光线跟踪领域中也出现了类似的变化，在数据结构上没有任何逻辑共享的并行化已被逻辑共享的数据结构所取代，这已

经在消息传递模型中用哈希方法实现。这种趋势的持续发展将会使应用程序在优先支持两种模型之一的系统中有很好的可移植性。

与并行应用程序一样，并行编程语言也在快速的发展。最通用的并行计算语言将持续以最通用的顺序执行编程语言（C、C++、Fortran）为基础，加之对并行性的扩展。实际应用的需求正推动着并行性的不断扩展。将从一类特殊的应用中提取出的特性加到语言中，在以后的应用中会发现这些特性其实也广泛适于其他类的应用。在编程语言中，这种方法其实并不罕见。在一个相似的系统中，常用数据结构和并行计算算法的可移植库正在被开发，模板也正在被建造，用户可以针对不同的特殊应用对模板进行定制。 957

多种风格的并行语言已经被开发出来了。消息传递编程中不太被公认的是 MPI，已经被多个平台所广泛采用，与其相似的是一种包含增强原语的串行语言，这已在第 2 章中的共享地址空间中讨论过。许多其他的语言系统都尽量为用户提供一个基于共享地址空间的自然编程模型，并且通过软件层将机器间的通信特征屏蔽掉。一个主要的发展方向是显示并行面向对象语言，它强调以与底层的数据支持相结合的方式，提供描述并行和同步的适当机制。另一个是含有划分制导语句的数据并行语言，例如正被扩展到处理不规则应用的高性能 Fortran。第三个方向是隐式并行语言。程序员不用说明赋值、协调和映像，而只需说明任务的分解和每个任务需存取的数据。基于这些说明，这种语言的运行系统能确定任务间的依赖关系和赋值，并在并行执行平台上合理地组织任务。程序员的负担被减轻了，至少被局部化了，但是系统的负担却增加了。随着数据存取对性能影响的增加，这些语言中已有不少已经提出分解负担的语言机制以及运行时的策略，以达到数据的局部性，并以一种合理的方法减少程序员和系统间的交互。

最后，应用程序编写复杂性和并行程序建模复杂性的逐渐增加最终导致了支持从小部分构建较大应用程序的语言的发展。我们可以预测以下的所有方向都能很好的持续发展：并发的正确提取、数据提取、同步以及数据的管理；库、模板；显式和隐式的并行编程语言，它们的目标是在编程的简易性、性能以及多个平台（功能与性能）间的可移植性中达到很好的权衡。

能使程序自动并行的编译器在过去的 10 年或者 20 年中已经有了很好的发展。伴随着对数据存取间依赖性分析的重大进展，编译器已经能使基于数组的 Fortran 程序自动并行化，并在小规模的多处理器上取得了很好的性能。对于具有特定性能特征的体系结构，处理高速缓存和主存中数据局部性的编译器算法以及通信结构的优化也已经有了很大的发展（例如，增大消息传递计算机的通信消息）。然而，编译器还不能很有效地并行较复杂的程序，特别是大量使用指针的程序。由于指针指向的地址在编译时还不可知，因此很难确定通过一个指针完成的内存操作与另一个内存操作是否具有相同的地址。在这种情况下，交互式并行化编译器越来越多地被使用。在这种机制下，编译器尽其所能发现程序中的并行性，然后在它不能确定的地方给用户一些智能化的反馈信息并向用户提出若干问题由用户进行选择，如分解、赋值及协调等操作。如果能得到具有指导意义的问题和信息，一个对程序熟悉的用户可以发现有关应用的高层知识，而这些知识是编译器不具备的。另一个类似“血管”的方法是编译时和运行时的并行工具结合在一起，这时一些在运行时刻收集到的信息可以帮助编译器（或者是帮助用户）成功地将应用并行化。编译器在这些领域还会继续发展，同样在为解决放松存储的同一性模型提供支持这一更简单的领域上也会继续发展。 958

操作系统正在经历从单处理器到多处理器，从用作面向批处理的计算引擎的多处理器到用作多道计算和存储的服务器的转变。前一种转变包括增加操作系统的可扩展性，减少操作系统的串行化，并使操作系统的进程安排和资源分配策略考虑空间和时间的局部性（例如，不过多移动机器上的进程；调度进程使它们更靠近访问的数据；尽量每次把同一进程安排在同一个处理器上）。后一种转变的一个主要挑战是在公平性与性能之间找到一个平衡点来进行资源分配和进程调度，就像在单处理器操作系统中一样。在调度多道程序进程时数据的局部性以及决定资源时的并行性能已经引起人们的关注。例如，多道程序系统中应用程序的相关并行性能可以用来测定处理器和其他资源以其他程序为代价对它的访问次数。并行处理器的操作系统愈来愈多地融合过去多道处理的巨型机的一些重要服务特性。这个领域的一个挑战是在提供分布式系统的可靠性、容错性的同时，让用户觉得是在使用一个单操作系统（这叫做单一系统映像）。另一个挑战是错误包容技术，使得仅仅出错的程序或者出错程序访问的资源受到该错误的影响，以提供人们想从巨型机得到的可靠性和实用性。如果基于微处理器的可扩展多处理器系统想真正替代大型机在大型组织中的“支柱性”服务器的地位，同时执行多道应用程序的话，这种发展是必要的。

随着应用程序和系统交互操作的复杂性的增加和存储，以及通信系统对系统性能的重要性的提高，使用好的性能问题诊断工具十分重要。这一点对共享地址空间的编程模型尤其重要，因为它的通信是隐性的，人为的通信很容易对性能产生其他影响。对程序员来说，竞争对性能的影响是特别普遍和难于诊断的，主要是因为这种影响出现在程序（或机器）中的位置与引起竞争的位置相差甚远。性能评测工具的发展可以提供反馈，在程序中占据最大执行时间的模块在哪里，什么样的数据结构可能引起数据访问花费等等。我们高兴的看到在提高性能监测工具的可见性方面的技术进步，机器设计者将有助于这种进步；他们越来越多的倾向于在机器的关键点增加一些寄存器或计数器，以助于进行性能监测。我们还看到给用户的反馈信息的质量方面的进步，从程序代码中哪些地方因数据访问延迟而占用大量的时间（今天已经可用），到哪种数据结构在这个延迟中所占的比例最大，到这个问题的原因所在（通信占用、容量扑空、与其他数据结构的冲突扑空或者竞争）。信息越详细，越依据程序员所处理的概念（而不是机器的概念），程序员越容易处理这些信息，改进性能。沿着这样的路线发展，有希望在软硬件对性能诊断的支持中找到一个很好的平衡。

959

这种发展的最后一个侧面是前面所讲过的不同系统软件的整合。编程语言的设计将使编译器在发现和处理并行性方面的工作更简单，并将传送给操作系统更多的信息，以帮助操作系统进行进程调度和资源定位。

12.2.2 遇到的困难

在软件方面，有一个我们多年以来一直碰到的难题，就是可编程性。编程模型变得可以更好地移植，体系结构越来越收敛，许多年来一直有很好的进步，但是并行编程仍然比串行编程困难得多。为了编好一个有高性能的程序，在寻找并行性，编程实现和调配方面需要花费大量的时间。调试一个并行程序简直是一门艺术，或者至少是在用很原始的科学方法。并行程序调试的困难在于具有各自的执行序列的多个进程相互作用，以及对时间的敏感性。一个错误是否在运行中出现，取决于一个进程中的某个事件是否因另一个进程中事件的出现而出现。这样，操纵代码的执行以使特定的事件出现，可能打断时序，从而使错误无法显现。

尽管技术进步已经极大地提高了并行计算的适应性，但是，要解决这一难题需要一个根本性的突破，以使得并行计算真正实现技术和体系结构的进步所给予它的潜在能力。现在还不清楚这一突破是在于编程语言本身，还是在程序设计方法学，或者它正处在一个逐步改进的过程的关键台阶。 960

12.2.3 潜在的突破

除了并行程序设计语言或方法学以及并行调试的突破，我们还希望在对并行程序的推导理论模型上获得突破。虽然许多模型能很好地表现系统的本质性能特性（Valiant 1990；Culler et al.1996），并且一些模型提供了一个方法学来使用这些参数，但是更重要的方面是对复杂并行应用程序的建模以及它们与系统参数的相互作用。目前还没有一个定义得很好的方法学，该方法学可以让程序员或算法设计人员用于决定，一个算法在一个并行系统或各竞争部分或协调方法上哪一个工作得更好。另一个突破来自体系结构，如果可以在良好性价比的情况下设计一个机器，这台机器使程序员能够不再太担心数据局部性和通信特性；这就是说，要真正地设计一台能够让程序员看上去像 PRAM 那样的机器。这样的一个例子是，是否由程序引起的所有延迟体系结构都能容忍。但是，这似乎需要巨大的带宽，而这些带宽需要巨大的花费，并且还不清楚为了并行执行和容忍时延而赋予一个应用并发特性有多大好处。

当然，最后的突破在于并行编译器的完全成功。编译器可以将大量的串行化程序转化成可以在一个给定的平台上有效执行的并行操作，以达到接近于由应用的固有特性决定的最佳性能（即，接近我们手工编程可以达到的最好性能）。除了前面讨论过的问题外，并行化编译器努力寻找和利用程序中低级别的、局部的信息，而不像现在只是能很好地处理高级别的、全局分析变换。编译器还缺少应用程序的语义信息；例如，如果一个特定的针对一个问题某阶段的串行算法不能很好地并行，编译器也没办法选择另一个算法。对编译器而言，考虑数据局部性、附加通信及管理多处理器的扩展内存结构是很困难的。但是，要使并行计算真正成为主流，有效的程序员辅助编译器并行化，即使程序员最低程度的介入，或许也是最重要的软件突破。

无论未来如何发展，可以肯定的是，持续不断的技术演化会越过一些重要的台阶；也会面临很大的障碍，但找到绕开它们的道路是可能的，而且可能产生一些意想不到的突破。在硬件和软件相互作用的进化周期中，计算机技术和应用都会发生激动人心的变化，并行计算技术将活跃在这种变化的前沿。 961

附录　并行基准测试程序集

尽管很难提出“有代表性的”并行应用程序，而且并行软件还不成熟（语言和编程环境），我们仍然必须走用工作负载驱动来定量地评价机器和体系结构的道路。因此，一系列并行“基准测试程序”被开发和推广起来。术语基准测试程序被加上引号，是因为很难用单一的一套程序对并行计算做出明确的性能评估。用于多处理器的并行基准测试程序随应用领域和运行时特征等方面的不同而不同；例如，它们是否只包含简单的“玩具程序”、内核或者是真正的应用程序；它们所针对的通信抽象；以及它们对基准测试程序和体系结构评估的思想。下面是一些最常用的公开的并行处理基准测试程序。表 A-1 给出如何获得这些基准测试程序集。

A.1　ScaLapack

ScaLapack 测试程序（Dongarra and Walker 1995；Choi et al.1992）是 LAPACK 线性代数核心的并行化版本，通信手段为消息传递。LAPACK 测试程序包括线性方程组的求解、特征值问题、奇异值问题、矩阵乘法、矩阵因式分解、特征值的求解方法。关于 ScaLapack 测试程序和程序本身的信息可以从 Oak Ridge National Laboratories 维护的 Netlib 站点获得（见表 A-1）。

表 A-1　公开能获得的基准测试程序和应用程序

基准测试程序集	通信抽象	应用领域	如何获得代码或相关信息
ScaLapack	消息传递	科学	http：//www.netlib.org
TPC	消息传递或共享地址空间（不提供并行）	数据库/事务处理	http：//www.tpc.org
SPLASH-2	共享地址空间（CC）	科学/工程/图形	http：//www-flash.Stanford.edu/apps/SPLASH
SPLASH-3	共享地址空间（CC）	不定	http：//www.cs.princeton.edu/prism/splash-3
NAS	消息传递或共享地址空间（纸上谈兵）	科学	http：//www.nas.nasa.gov
NPB2	消息传递	科学	http：//science.nas.nasa.gov/Software/NPB
PARKBENCH	消息传递	科学	netlib@ornl.gov

A.2　TPC

成立于 1988 年的事务处理性能委员会（TPC）建立了一套公用的基准测试程序，TPC-A，TPC-B、TPC-C 和 TPC-D。这些基准测试程序代表了不同类型的输入和对事务处理与数据库系统程序的不同要求（事务处理委员会 1998）。鉴于数据库和事务处理的工作负载在并行机实际应用中十分重要，这些程序的源代码对于它们的开发者来说就有了竞争价值，因而这些

源代码是很难获得的。

TPC 对报告结果有严格的规定。报告结果使用两个公制单位：1）每秒钟处理的事务中的吞吐量，要求对 90%以上事务的响应时间限制在一个特定的阈值内；2）每秒钟每笔事务的价格性能比，价格指的是整个系统的价格及其 5 年内的维修费用的总和。这些基准测试程序的规模随着系统能力的增强而增大，是为了能够切实地评估系统的性能。

第一个 TPC 基准测试程序是 TPC-A，只有一个单一的、简单的、更新密集型的事务。它提供一个简单、可重复的工作负载单位来测试在线事务处理（OLTP）系统的主要特征，比如银行户头记录或机票定座系统。所选择的银行事务包括从终端读 100 个字节、修改账户、营业部门和出纳记录、写下历史记录、最后向终端写回 200 个字节。TPC-A 已不再使用。

TPC-B 是比较集中些的数据库基准测试程序（不是在线事务处理）。它用于测试对于修改密集型的数据库事务很必要的系统配件。因此它的主要时间耗在大量的磁盘 I/O 上，而适当的系统和应用程序的执行时间一般，它还要求事务的完整性。不同于在线事务处理，TPC-B 不需要终端或网络连接。上面谈到的两个基准测试程序（TPC-A 和 TPC-B）于 1995 年宣布被淘汰，现在它们已完全被 TPC-C 和 TPC-D 取代。

TPC-C 于 1992 年通过。它比 TPC-A 更实用，且也继承了 TPC-A 的许多特点。TPC-C 是个多用户的基准测试程序，它需要一个远程终端仿真器来模仿用户群体和他们的终端。它模拟不同地域分布的销售区内批发商的活动：客户定货、交费、查询以及送货和存货的检查。数据库的大小随系统的吞吐量扩放。TPC-C 的数据库结构比 TPC-A 更复杂，比如，多种不同复杂度的事务类型、在线和推迟的执行模式、高层次的数据访问和更新的冲突、模拟访问热点的模式、通过主键和非主键的访问、对全屏终端 I/O 和格式的实际要求，对数据划分的透明性和事务处理回滚的要求。 964

TPC-D 是一个决策支持的基准测试程序。根据事务处理委员会的定义，决策支持描述了一个系统的能力，这种能力可以支持在复杂查询要求下商务决策的形成。复杂的查询涉及数据库中的很大范围，而不只是单独的记录（如在 OLTP 中），它们包括多表格连结、大量的排序操作、分组和合并、还有顺序扫描。在线事务处理（TPC-C）只涉及少量的、主要是更新性质的事务，但决策支持的查询则是对数据库大量的、独立耗时的只读操作。决策支持数据库不经常更新，其更新或者是由阶段性的批处理完成，或者由后台一些机动性小活动产生。这些更新活动也包括在 TPC-D 中。和 TPC-C 中的常规事务操作不同，TPC-D 模拟随意的查询，如销售趋势的决策等，它通常只有几个并发用户。

事务处理委员打算增加更多的基准测试程序，包括商业服务器的基准测试程序。它对同步在线事务处理、批量事务处理、只读在线事务处理有更短的响应时间限制。客户机-服务器的基准测试程序也在考虑之中。所有这些信息可以从事务处理性能委员会获得（见表 A-1）。

A.3 SPLASH

斯坦福共享内存并行应用（SPLASH）基准测试程序（Singh，Weber，and Gupta 1992）最初在斯坦福大学开发的。它用于支持一致性复制的共享地址空间的体系结构的评估。后来被 SPLASH-2 测试程序取代（Woo et al.1995）。SPLASH-2 增强和增加了一些应用功能，极大地拓宽了其覆盖的范围和特点。SPLASH-2 系列现在具有 7 个完整的应用功能和 5 个计算内核。

部分应用功能和计算核心随数据结构和使用的不同，有不同的版本和优化程度（参考4.2.2节有关优化层的讨论）。这些程序都有不同的计算领域，大部分在科学和工程以及计算机图形等领域。它们都用C语言编写并使用Argonne国家实验室（Boyle et al.1987）的Parmacs宏包做并行设计。它们的特点和使用方法逻辑指导可以参阅《*Woo et al.1995年*》的工作。这本书中所有共享地址空间机器上以作业量为评价标准的并行程序都源于SPLASH-2基准测试程序。该程序及其文档资料可以通过表A-1中的信息获得。SPLASH-2的设计者打算在不同应用领域增加共享地址空间程序，形成新版本SPLASH-3。

965

A.4 NAS并行基准测试标准

NAS基准测试标准（Bailey et al.1991，1995）由美国国家航空航天局（NASA）的数字动态航空模拟组开发。它们是8种计算的集合——5个内核和3个伪应用（不完全，但是比各种数据结构、数据转换和航空物理应用中的计算的核心更具代表性）。每种计算都分别用于航空物理应用中的高度并行计算的一些重要方面。内核包括很复杂的计算，如：多栅方程求解器、共扼方程求解器、三维FFT方程求解器，整数排序程序。每个基准测试标准都有一小一大两个不同的数据集合。这些基准测试程序用于评价机器的性能，并且给出仔细的报告和有效的评价原则。

最初的NAS基准测试程序不同于前面提及的其他基准测试程序。那些程序是用高级语言写成（如Fortran或C并且带有并行性结构）的。但是NAS基准测试程序并没有提供并行操作，而是纸上谈兵式的基准测试程序。它们对问题进行详细的描述（例如方程组及其限制）和使用高级求解方法（例如多项式或共扼方程求解）。它们并不给出并行程序，而是让用户使用手边机器上的最好的并行操作。用户可以选择任何语言来实现并行（但这种语言必须是C或Fortran的扩展）、数据结构、通信抽象和机制、处理器映射、内存分配和使用以及低级优化（对汇编语言使用的一些限制）。这样做的目的是：由于并行体系结构多样化的性能特点及其编程模型的不同，同时还没有一个在所有体系结构上都有效的程序语言或通讯抽象；因此很适合一台机器的一个并行程序操作，并不一定适合另一台机器。如果我们要用一个给定的计算或测试程序来比较两台机器的性能，我们就应该用最适合每台机器的操作。这一手段给使用该程序的用户造成很大压力，但是却非常适合两台截然不同的机器的比较。从另一方面来看，有了源代码用户就更容易在完整定义的相似体系结构中作出权衡。因此，在保持其基本精神的同时，NAS还是给出了基准测试程序的实现，它们可以做为用户编写自己的基准测试程序的出发点。

966

因为NAS基准测试程序，特别是它的内核相对容易实现，而且也代表了科学计算领域的一个有意义的范围，它们受到多处理器提倡者的广泛支持。最近形成了一种称做NPB2的项目，它通过消息传递界面（MPI）标准提供的可移植性，使用固定的用MPI写成的应用程序，而不是以前描述性的基准测试程序了，就像传统的基准测试程序一样。该程序及其文档资料可以从NAS获得，参见表A-1。

A.5 PARKBENCH

并行内核基准测试程序（PARKBENCH）的工作（PARKBENCH委员会1994）是这样一个大规模的项目，它要开发一系列用于并行机基准测试的微测试程序、内核和应用程序。

PARKBENCH 最初主要是用 Fortran77 写的消息传递程序。高性能 Fortran（HPF）语言（高性能 Fortran 论坛 1993）版本的设计也是为了在消息传递和共享地址空间的平台上提供可移植性功能。这些程序来自于科学计算

PARKBENCH 提供了我们讨论过的不同类型的基准测试程序：低级基准测试程序或微基准测试程序、内核、简化应用程序和编译测试程序（后两种还没有完全完成）。低级微基准测试程序测试每个节点的性能。这些单处理器的微基准测试程序是为了评估体系结构和编译系统各方面的性能，并且获取一些可了解内核和简化应用程序性能的参数。单处理器的微程序包括定时呼叫、算术运算、内存带宽和时延测试例程，也有单处理器的微基准测试程序测试通信时延和带宽——点对点和全部对全部——就像全局同步栅障一样。一些多处理器的微基准测试标准来自于早期的 Genesis 基准测试程序（Hey 1991）。

内核测试程序可分解成矩阵内核（乘法、因式分解、转置、三角化）、傅立叶变换内核（大规模的一维 FFT 和三维 FFT）、偏微分方程核心（三维回归解和 NAS 系列的多元核心），其他像共扼梯度、整数排序和 NAS 系列中复杂的并行内核，以及描述型的 I/O 基准测试程序。

简化应用程序（完备且但简化了的应用程序）应用于天气预报模型、计算流体动力学、经济模型、文件优化、分子动力学、等离子物理学、量子化学、量子动力学、水库模型等诸多领域。最后，编译测试程序是为了开发高性能 Fortran 编译器来测试编译的优化程度，而不是特别用于评估体系结构。关于 PARKBENCH 基准测试标准的信息可从表 A-1 获得。 967

A.6 其他正在进行的工作

SPEC/HPCG：SPEC 单处理器基准测试标准（SPEC 1995）系列的开发者（标准性能评估公司）与传统向量超级计算机（Berry et al.1989）的基准测试标准 Perfect（效用转换性能评估）的开发者联合起来正在开发 SPEC/HPG 测试标准系列。这一标准是为测试有着最先进计算技术的系统的性能，也就是多处理器系统。这些基准测试程序也集中在测试科学计算的性能上。

还有许多其他建立测试标准的计划。正如我们所见到的，许多基准测试程序到目前为止还是用于消息传递的计算，虽然其中一些测试程序开始提供高性能 Fortran，使其能够运行于任何有合适编译器支持的主流通信抽象之上。实现更多的共享地址空间测试标准也指日可待，比如将来的 SPLASH 和 SPLASH-2。当前许多基准测试程序也还是针对科学计算的（PARKBENCH 和 NAS 是其中规模最大的），同时也正在逐渐开发一些面向并行计算机其他工作领域的基准测试程序（比如商业和一般的服务器工作）。几乎所有的基准测试程序都设计成一次在机器上运行一个应用程序。现在还没有好的基准测试标准可以使多个基准测试程序同时运行在一个操作系统之上，也没有针对并行计算机的 I/O 密集型测试标准。总而言之，开发既具有并行计算的代表性又容易在不同机器上移植的基准标准测试程序是一个很难但又很重要的问题（因为我们在体系结构权衡方面得到的结论依赖于所使用的测试标准），上面所提到的都是朝着这个方向的努力。 968

参考文献

Abali, B., and C. Aykanat. 1994. Routing Algorithms for IBM SP1. *Lecture Notes in Computer Science, Vol. 853*. New York: Springer-Verlag, 161–175.

Abdel-Shafi, H. A., J. Hall, S. V. Adve, and V. S. Adve. 1997. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. *Proc. Third Symposium on High Performance Computer Architecture* (February).

Adve, S. V. 1993. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. Ph.D. diss., University of Wisconsin-Madison. Available as Tech. Report #1198, University of Wisconsin-Madison, Computer Science (December).

Adve, S. V., and K. Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29(12):66–76.

Adve, S. V., K. Gharachorloo, A. Gupta, J. L. Hennessy, and M. Hill. 1993. Sufficient Systems *Requirements for Supporting the PLpc Memory Model*. Tech. Report #1200, University of Wisconsin-Madison, Computer Science (December). Also available as Tech. Report #CSL-TR-93-595, Stanford University.

Adve, S. V., and M. Hill. 1990a. Weak Ordering: A New Definition. 1990. *Proc. 17th Int'l Symposium on Computer Architecture* (May):2–14.

———. 1990b. Implementing Sequential Consistency in Cache-Based Systems. *Proc. 1990 Int'l Conference on Parallel Processing* (August):47–50.

———. 1993. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems* 4(6):613–624.

Agarwal, A. 1991. Limit on Interconnection Performance. *IEEE Transactions on Parallel and Distributed Systems* 2(4):398–412.

Agarwal, A., R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. 1995. The MIT Alewife Machine: Architecture and Performance. *Proc. 22nd Int'l Symposium on Computer Architecture* (May/June):2–13.

Agarwal, A., and A. Gupta. 1988. Memory-Reference Characteristics of Multiprocessor Applications Under MACH. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May):215–225.

Agarwal, A., B.-H. Lim, D. Kranz, and J. Kubiatowicz 1990. (April): A Processor Architecture for Multiprocessing. *Proc. 17th Annual Int'l Symposium on Computer Architecture* (June):104–114.

———. 1991. LimitLESS Directories: A Scalable Cache Coherence Scheme. *Proc. Fourth Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (April):224–234.

Agarwal, A., R. Simoni, J. Hennessy, and M. Horowitz. 1988. An Evaluation of Directory Schemes for Cache Coherence. *Proc. 15th Int'l Symposium on Computer Architecture* (June):280–289.

Aiken, A., and A. Nicolau. 1988. Optimal Loop Parallelization. *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (June):308–317. Also published in SIGPLAN Notices 23(7).

Aimoto, Y., T. Kimura, Y. Yabe, H. Heiuchi, et al. 1996. A 7.68GIPS 3.84GB/s 1W Parallel Image-Processing RAM Integrating a 16Mb DRAM and 128 Processors. *International Solid-State Circuits Conference*, San Francisco (February):372–373.

Alexander, T. B., K. G. Robertson, D. T. Lindsay, D. L. Rogers, J. R. Obermeyer, J. R. Keller, K. Y. Oka, and M. M. Jones II. 1994. Corporate Business Servers: An Alternative to Mainframes for Business Computing (HP K-Class). *Hewlett-Packard Journal* (June):8–33.

Almasi, G. S., and A. Gottlieb. 1989. *Highly Parallel Computing*. Redwood City, CA: Benjamin/Cummings.

Alverson, R., D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. 1990. The Tera Computer System. *Proc. 1990 Int'l Conference on Supercomputing* (June): 1–6.

Amdahl, G. M. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS 1967 Spring Joint Computer Conference* 40:483–485.

Anderson, J. P., S. A. Hoffman, J. Shifman, and R. Williams. 1962. D825-A Multiple-Computer System for Command and Control. *AFIP Proc. FJCC* 22:86–96.

Anderson, J., and M. Lam. 1993. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. *Proc. SIGPLAN'93 ConferenceConference on Programming Language Design and Implementation* (June).

Anderson, T. E., D. E. Culler, D. Patterson. 1995. A Case for NOW (Networks of Workstations). *IEEE Micro* 15(1):54–6

Anderson, T. E., S. S. Owicki, J. P. Saxe, and C. P. Thacker. 1992. High Speed Switch Scheduling for Local Area Networks. *Proc. ASPLOS V* (October):98–110

Archibald, J., and J.-L. Baer. 1986. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems* 4(4):273–298.

Arnould, E. A., F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste. 1989. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. *Proc. ASPLOS III* (April):205–216.

Arpaci, R. H., D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick. 1995. Empirical Evaluation of the Cray-T3D: A Compiler Perspective. *Proc. 22nd Int'l Symposium on Computer Architecture* (June):320–331.

Arvind, and D. E. Culler. 1986. Dataflow Architectures. *Annual Reviews in Computer Science* 1:225–253. Palo Alto, CA: Annual Reviews. Reprinted in *Dataflow and Reduction Architectures*. Edited by S. S. Thakkar. Los Alamitos, CA: IEEE Computer Society Press, 1987.

Athas, W. C., and C. L. Seitz. 1988. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer* 21(8):9–24.

August, M. C., G. M. Brost, C. C. Hsiung, and A. J. Schiffleger. 1989. Cray X-MP: The Birth of a Supercomputer. *Computer* 22(1):45–52.

Baer, J.-L., and T.-F. Chen. 1991. An Efficient On-Chip Preloading Scheme to Reduce Data Access Penalty. *Proc. Supercomputing '91* (November):176–186.

Baer, J.-L., and W.-H. Wang. 1988. On the Inclusion Properties for Multi-Level Cache Hierarchies. *Proc. 15th Annual Int'l Symposium on Computer Architecture* (May):73–80.

Bailey, D. H. 1990. FFTs in External or Hierarchical Memory. *Journal of Supercomputing* 4(1):23–35. Also published in *Proc. Supercomputing '89* (November):234–242.

———. 1991. Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers. *Supercomputing Review* (August):54–55.

———. 1993. Misleading Performance Reporting in the Supercomputing Field. *Scientific Programming* 1(2):141–151. Also published in *Proc. Supercomputing '93*

Bailey, D. H., E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. 1991. The NAS Parallel Benchmarks. *Intl. Journal of Supercomputer Applications* 5(3):66–73. Also published as Tech. Report RNR-94-007, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center (March 1994).

Bailey, D. H., E. Barszcz, L. Dagum, and H. D. Simon. 1994. NAS Parallel Benchmark Results 3–94. *Proc. Scalable High-Performance Computing Conference*, Knoxville, TN (May):111–120.

Bailey, D., T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. 1995. *The NAS Parallel Benchmarks 2. 0.* Report NAS-95-020, Numerical Aerodynamic Simulation Facility. NASA Ames Research Center (December).

Baker, W. E., R. W. Horst, D. P. Sonnier, and W. J. Watson. 1995. A Flexible ServerNet-Based Fault-Tolerant Architecture. Proc. *25th Int'l Symposium on Fault-Tolerant Computing* (June). Los Alamitos, CA: IEEE Computer Society Press, 2–11.

Bakoglu, H. B. 1990. *Circuits, Interconnection, and Packaging for VLSI.* Reading. MA: Addison-Wesley.

Ball, J. R., R. C. Bollinger, T. A. Jeeves, R. C. McReynolds, D. H. Shaffer. 1962. On the Use of the Solomon Parallel-Processing Computer. *Proc. AFIPS Fall Joint Computer Conference* 22:137–146.

Banks, D., and M. Prudence. 1993. A High Performance Network Architecture for a PA-RISC Workstation. *IEEE Journal on Selected Areas in Communication* 11(2):191–202.

Barnes, J. E., and P. Hut. 1989. Error Analysis of a Tree Code. *Astrophysics Journal Supplement* 70(June):389–417.

Barosso, L., and M. Dubois. 1993. The Performance of Cache-Coherent Ring-Based Multiprocessors. *Proc. 20th Annual Int'l Symposium on Computer Architectures (ISCA)* (May):268–277.

———. 1995. Performance Evaluation of the Slotted Ring Multiprocessors. *IEEE Transactions on Computers* 44(7):878–890.

Barroso, L. A., S. Iman, J. Jeong, K. Oner, K. Ramamurthy, and M. Dubois. 1995. RPM: A Rapid Prototyping Engine for Multiprocessor Systems. *IEEE Computer* 28(2):26–34

Barszcz, E., Fatoohi, R., Venkatakrishnan, V., and Weeratunga, S. 1993. *Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors.* Tech. Report NAS RNR-93-007, NASA Ames Research Center, Moffett Field, CA (April).

Barton, E., J. Crownie, and M. McLaren. 1994. Message Passing on the Meiko CS-2. *Parallel Computing* 20(4):497–507.

Baskett, F., T. Jermoluk, and D. Solomon. 1988. The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 Lighted Polygons per Second. *Proc. 33rd IEEE Computer Society Int'l Conference—COMPCON '88* (February):468–471

Batcher, K. E. 1974. Staran Parallel Processor System Hardware. *Proc. AFIPS National Computer Conference*, 405–410.

———. 1980. Design of a Massively Parallel Processor. *IEEE Transactions on Computers* C-29(9):836–840.

Bell, C. G. 1985. Multis: A New Class of Multiprocessor Computers. *Science* 228:462–467.

Benes, V. 1965. *Mathematical Theory of Connecting Networks and Telephone Traffic.* San Diego, CA: Academic Press.

Bennett, J. E., and M. J. Flynn. 1996a. *Latency Tolerance for Dynamic Processors.* Tech. Report #CSL-TR-96-687, Computer Systems Laboratory, Stanford University.

———. 1996b. *Reducing Cache Miss Rates Using Prediction Caches.* Tech. Report #CSL-TR-96-707, Computer Systems Laboratory, Stanford University.

Berry, M., D. Chen, P. Koss, et al. 1989. The PERFECT Club Benchmarks: Effective Performance Evaluation of Computers. *Int'l Journal of Supercomputer Applications* 3(3):5–40.

Bershad, B. N., M. J. Zekauskas, and W. A. Sawdon. 1993. The Midway Distributed Shared Memory System. *Proc. COMPCON '93* (February).

Bhatt, S. M. and C. E. Leiserson. 1983. How to Assemble Tree Machines. *ACM Symposium on Theory of Computing (STOC '82).* New York: ACM Press.

Biagioni, E., E. Cooper. and R. Sansom. 1993. Designing a Practical ATM LAN. *IEEE Network* (March).

Bilas, A., L. Iftode, and J. P. Singh. 1998. Evaluation of Hardware Support for Next-Generation Shared Virtual Memory Clusters. *Proc. Int'l Conference on Supercomputing* (July)

Bisiani, R., and M. Ravishankar. 1990. PLUS: A Distributed Shared-Memory System. *Proc. 17th Int'l Symposium on Computer Architecture* (May):115–124.

Black, D., R. Rashid, D. Golub, C. Hill, R. Baron. 1989. Translation Lookaside Buffer Consistency: A Software Approach. *Proc. Third Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Boston (April):113–122.

Blackford, L. S., J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1997. *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM).

Blelloch, G. 1993. Prefix Sums and Their Applications. In *Synthesis of Parallel Algorithms*. Edited by J. Reif. San Francisco: Morgan Kaufmann, 35–60.

Blelloch, G. E., C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. A. Zagha. 1991. Comparison of Sorting Algorithms for the Connection Machine CM-2. *Proc. Symposium on Parallel Algorithms and Architectures* (July):3–16.

Blumrich, M. A., C. Dubnicki, E. W. Felten, K. Li, M.R. Mesarina. 1994. Two Virtual Memory Mapped Network Interface Designs. *Proc. Hot Interconnects II Symposium* (August).

Blumrich, M., K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. 1994. A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer. *Proc. 21st Int'l Symposium on Computer Architecture* (April):142–153.

Boden, N., D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. 1995. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro* 15(1):29–38.

Bodin, F., P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. 1993. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. *Proc. Supercomputing '93* (November):588–597. Also in *Scientific Programming* 2(3).

Bolt Beranek and Newman Advanced Computers. 1989. *TC2000 Technical Product Summary.* Cambridge, MA: Bolt Beranek and Newman.

Bomans, L., and D. Roose. 1989. Benchmarking the iPSC/2 Hypercube Multiprocessor. *Concurrency: Practice and Experience*, 1(1):3–18.

Borkar, S., R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. 1990. Supporting Systolic and Memory Communication in iWarp. *Proc. 17th Annual Int'l Symposium on Computer Architecture*, Seattle, WA (May):70–81. Revised version appears as Tech. Report #CMU-CS-90–197, Carnegie Mellon University.

Bouknight, W. J., S. A Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. 1972. The Illiac IV System. *Proc. IEEE* 60(4):369–388.

Boyle, J., R. Butler, T. Disz, B. Glickfield, E. Lusk, W. R. Overbeek, J. Patterson, and R. Stevens, 1987. *Portable Programs for Parallel Processors*. New York: Holt, Rinehart and Winston.

Brewer, E. A., F. T. Chong, F. T. Leighton. 1994. Scalable Expanders: Exploiting Hierarchical Random Wiring. *Proc. 1994 Symposium on the Theory of Computing*, Montreal, Canada (May):144–152.

Brewer, E. A., F. T. Chong, L. T. Liu, J. Kubiatowicz, S. D. Sharma. 1995. Remote Queues, Exposing Network Queues for Atomicity and Optimization. *Proc. Seventh Annual Symposium on Parallel Algorithms and Architectures* (July):42–53.

Brewer, E. A., and B. C. Kuszmaul. 1994. How to Get Good Performance from the CM-5 Data Network. *Proc. 1994 Int'l Parallel Processing Symposium*, Cancun, Mexico (April):858–867.

Bruno, J., P. R. Cappello. 1988. Implementing the Beam and Warming Method on the Hypercube. *Proc. Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan 19–20.

Burger, D. 1997. *System-Level Implications of Processor-Memory Integration*. Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. Presented at the Int'l Symposium on Computer Architecture (ISCA) '97 (June).

Burger, D., J. Goodman, and A. Kagi. 1996. Memory Bandwidth Limitations in Future Microprocessors. *Proc. 23rd Annual Symposium on Computer Architecture* (May):78–89.

Burkhardt, H., et al. 1992. *Overview of the KSR-1 Computer System*. Tech. Report KSR-TR-9202001, Kendall Square Research, Boston (February).

Butler, M., T-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. 1991. Single Instruction Stream Parallelism Is Greater Than Two. *Proc. Annual Int'l Symposium on Computer Architecture (ISCA)*, 276–86

Callahan, T., and S. C. Goldstein. 1995. NIFDY: A Low Overhead, High Throughput Network Interface. *Proc. 22nd Annual Symposium on Computer Architecture* (June):230–241.

Cardoza, W., F. Glover, and W. Snaman Jr. 1996. Design of a TruCluster Multicomputer System for the Digital UNIX Environment. *Digital Technical Journal* 8(1):5–17

Carter, J. B., J. K. Bennett, and W. Zwaenepoel. 1991. Implementation and Performance of Munin. *Proc. 13th Symposium on Operating Systems Principles* (October):152–164.

———. 1995. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions of Computer Systems* 13(3):205–244.

Catanzaro, B. 1997. *Multiprocessor System Architectures: A Technical Survery of Multiprocessor/Multithreaded Systems Using SPARC, Multi-level Bus Architectures and Solaris (SunOS)*. Mountain View, CA: Sun Microsystems.

Cekleov, M., D. Yen, P. Sindhu, J.-M. Frailong, et al. 1993. SPARCcenter 2000: Multiprocessing for the 90s, Digest of Papers. *Proc. COMPCON Spring '93*. Los Alamitos, CA: IEEE Computer Society Press, 345–353.

Censier, L., and P. Feautrier. 1978. A New Solution to Cache Coherence Problems in Multiprocessor Systems. *IEEE Transaction on Computer Systems* C-27(12):1112–1118.

Chan, K., et al. 1993. Multiprocessor Features of the HP Corporate Business Servers. *Proc. COMPCON* (Spring):330–337.

Chandy, K. M., and J. Misra. 1988. *Parallel Program Design: A Foundation*. Reading, MA: Addison Wesley.

Chang, P. P., S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. 1991. IMPACT: An Architectural Framework for Multiple-Instruction Issue Processors. *Proc. 18th Int'l Symposium on Computer Architecture (ISCA)* 19(3):266–275.

Chen, T.-F., and J.-L. Baer. 1992. Reducing Memory Latency via Non-Blocking and Prefetching Caches. *Proc. Fifth Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):51–61.

———. 1994. A Performance Study of Software and Hardware Data Prefetching Schemes. *Proc. 21st Annual Symposium on Computer Architecture* (April):223–232.

Cheong, H., and A. Viedenbaum. 1990. Compiler-directed Cache Management in Multiprocessors. *IEEE Computer* 23(6):39–47.

Chien, A. A., and J. H. Kim. 1992 Planar-Adaptive Routing: Low-Cost Adaptive Networks for Multiprocessors. *Proc. 19th Annual International Symposium on Computer Architecture (ISCA)*, Gold Coast, Australia (May):268–277.

Choi, J., J. J. Dongarra, R. Pozo, and D. W. Walker. 1992. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. *Proc. Fourth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA. Los Alamitos, CA: IEEE Computer Society Press, 120–127.

Chun, B. N., A. M. Mainwaring, and D. E. Culler. 1998. Virtual Network Transport Protocols for Myrinet. *IEEE Micro* (January):53–63

Clark, R., and K. Alnes. 1996. An SCI Chipset and Adapter. *Symposium Record, Hot Interconnects IV* (August):221–235

Cohen, D., G. Finn, R. Felderman, and A. DeSchon. 1993. ATOMIC: A Low-Cost, Very High-Speed, Local Communication Architecture. *Proc. 1993 Int. Conference on Parallel Processing.*

Convex Computer Corporation. 1993. *Exemplar Architecture*. Richardson, TX: Convex Computer Corp.

Corella, F., J. Stone, C. Barton. 1993. *A Formal Specification of the PowerPC Shared Memory Architecture*. Tech. Report Computer Science RC 18638 (81566), IBM Research Division, T.J. Watson Research Center (January).

Cornell, J. A. 1972. Parallel Processing of Ballistic Missile Defense Radar Data with PEPE. *COMPCON* 72, 69–72.

Cox, A., and R. Fowler. 1993. Adaptive Cache Coherency for Detecting Migratory Shared Data. *Proc. 20th Int'l Symposium on Computer Architecture* (May):98–108.

Crowther, W., J. Goodhue, R. Gurwitz, R. Rettberg, and R. Thomas. 1985. The Butterfly Parallel Processor. *IEEE Computer Architecture Technical Newsletter*, 18–46.

Culler, D. E. 1994. Multithreading: Fundamental Limits, Potential Gains, and Alternatives. In *Multithreaded Computer Architecture: A Summary of the State of the Art*. Edited by R. Iannucci. Dordrecht, Germany; Norwell, MA: Kluwer Academic Publishers, 97–138.

Culler, D. E., A. C. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. 1993. Parallel Programming in Split-C. *Proc Supercomputing* '93 (November):262–273.

Culler, D. E., A. C. Dusseau, R. P. Martin, and K. E. Schauser. 1993. Fast Parallel Sorting under LogP: From Theory to Practice. In *Portability and Performance for Parallel Processing*, Chapter 4. New York: John Wiley & Sons, 71–98.

Culler, D. E., R. M. Karp, D. A., Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. 1993. LogP: Toward A Realistic Model of Parallel Computation. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (May):1–12

———. 1996. LogP: A Practical Model of Parallel Computation. *CACM* 39(11):78–85.

Culler, D. E., A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. 1991. Fine-Grain Parallelism with Minimal Hardware Support. *Proc. Fourth Int'l Symposium on Arch. Support for Programming Languages and Systems (ASPLOS)* (April):164–175.

Culler, D. E., K. E. Schauser, and T. von Eicken. 1993. Two Fundamental Limits on Dataflow Multithreading. *Proc. IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, FL.

Dahlgren, F. 1995. Boosting the Performance of Hybrid Snooping Cache Protocols. *Proc. 22nd Int'l Symposium on Computer Architecture* (June):60–69.

Dahlgren, F., M. Dubois, and P. Stenstrom. 1994. Combined Performance Gains of Simple Cache Protocol Extensions. *Proc. 21st Int'l Symposium on Computer Architecture* (April):187–197.

———. 1995. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 6(7).

Dally, W. J. 1990a. Virtual-Channel Flow Control. *Proc. 17th Annual Int'l Symposium on Computer Architecture (ISCA)*, Seattle, WA, (May):60–68.

Dally, W. J. 1990b. Performance Analysis of *k*-ary *n*-cube Interconnection Networks. *IEEE-TOC* 39(6):775–85.

Dally, W. J., A. Chien, S. Fiske, W. Horwat, J. Keen, J. Larivee, R. Lethin, P. Nuth, S. Willis. 1989. The J-Machine: A Fine-Grained Concurrent Computer. *Proc. IFIP 11th World Computer Congress, Information Processing '89*, 1147–1153.

Dally, W. J., J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, and P. R. Nuth. 1992. The Message Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro* (April):23–39.

Dally, W. J., J. S. Keen, M. D. Noakes. 1993. The J-Machine Architecture and Evaluation. *Digest of Papers. COMPCON Spring '93*, San Francisco, CA (February):183–188.

Dally, W. J., and C. Seitz. 1987. Deadlock-Free Message Routing in Multiprocessor Interconnections Networks. *IEEE-TOC* C-36(5):547–553.

Denning, P. J. 1968. The Working Set Model for Program Behavior. *Communications of the ACM* 11(5):323–333.

Dennis, J. B. 1980. Dataflow Supercomputers. *IEEE Computer* 13(11):93–100

Digital Equipment Corporation. 1992. *Alpha Architecture Handbook*. Maynard, MA: Digital Equipment Corp.

Dijkstra, E. W. 1965. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM* 8(9):569.

Dijkstra, E. W., and C. S. Sholten. 1968. Termination Detection for Diffusing Computations. *Information Processing Letters* 1:1–4.

Dongarra, J. J. 1990. *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment*. Tech. Report CS-89-85, University of Tennessee, Computer Science Dept. (March).

———. 1994. *Performance of Various Computers Using Standard Linear Equation Software*. Tech Report CS-89-85, University of Tennessee, Computer Science Dept. (November); current report available from *netlib@ornl.gov*

Dongarra, J. J., J. Martin, and J. Worlton. 1987. Computer Benchmarking: Paths and Pitfalls *IEEE Spectrum* (July):38.

Dongarra, J. J., and D. W. Walker. 1995. Software Libraries for Linear Algebra Computations on High performance Computers. *SIAM Review* 37:151–180.

Dongarra, J. J., and W. Gentzsch, eds. 1993. *Computer Benchmarks*. Amsterdam: Elsevier Science B. V., North-Holland.

Dubnicki, C., L. Iftode, E. W. Felten, K. Li. 1996. Software Support for Virtual Memory-Mapped Communication. *Tenth Int'l Parallel Processing Symposium* (April).

Dubnicki, C., and T. LeBlanc. 1992. Adjustable Block Size Coherent Caches. *Proc. 19th Annual Int'l Symposium on Computer Architecture* (May):170–180.

Dubois, M., and C. Scheurich. 1990. Memory Access Dependencies in Shared-Memory Multiprocessors. *IEEE Transactions on Software Engineering*16(6):660–673.

Dubois, M., C. Scheurich, and F. Briggs. 1986. Memory Access Buffering in Multiprocessors. *Proc. 13th Int'l Symposium on Computer Architecture* (June):434–442.

Dubois, M., J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. 1993. The Detection and Elimination of Useless Misses in Multiprocessors. *Proc. 20th Int'l Symposium on Computer Architecture* (May):88–97.

Dubois, M., J.-C. Wang, L. A. Barroso, K. Chen and Y.-S. Chen. 1991. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. *Proc. Supercomputing '91* (November):197–206.

Dunigan, T. H. 1988. *Performance of a Second Generation Hypercube* Tech. Report ORNL/TM-10881, Oak Ridge National Lab. (November)

Dunning, D., G. Regnier, G. McAlpine, D. Camaron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. 1998. The Virtual Interface Architecture. *IEEE Micro* 18(2).

Dusseau, A. C., D. E. Culler, K. E. Schauser, and R. P. Martin. 1996. Fast Parallel Sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems* 7(8):791–805.

Dwarkadas, S., P. Keleher, A. L. Cox, and W. Zwaenepoel. 1993. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. *Proc. 20th Int'l Symposium on Computer Architecture* (May):144–155.

Eggers, S., and R. Katz. 1988. A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation. *Proc. 15th Annual Int'l Symposium on Computer Architecture* (May):373–382.

———. 1989a. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. *Proc. Third Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (May):257–270.

———. 1989b. Evaluating the Performance of Four Snooping Cache Coherency Protocols. *Proc. 16th Annual Int'l Symposium on Computer Architecture* (May):2–15.

Eigenmann, R., and S. Hassanzadeh. 1996. Benchmarking with Real Industrial Applications: The SPEC High Performance Group. *IEEE Computational Science and Engineering* (spring).

Elliott, D. G., W. M. Snelgrove, and M. Stumm. 1992. Computational RAM: A Memory-SIMD Hybrid and Its Application to DSP. *Custom Integrated Circuits Conference*, Boston, MA (May):30.6.1–30.6.4.

Elliott, D. G., M. Stumm, and W. M. Snelgrove. 1997. *Computational RAM: The Case for SIMD Computing in Memory.* Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. Presented at Annual International Symposium on Computer Architecture (ISCA) '97 (June).

Erlichson, A., B. Nayfeh, J. P. Singh and Oyekunle Olukotun. 1995. The Benefits of Clustering in Cache-Coherent Multiprocessors: An Application-Driven Investigation. *Proc. Supercomputing '95* (November).

Erlichson, A., N. Nuckolls, G. Chesson, and J. L. Hennessy. 1996. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. *Proc. Seventh Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):210–220.

Falsafi, B., A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. 1994. Application-Specific Protocols for User-Level Shared Memory. *Proc. Supercomputing '94* (November):380–389.

Falsafi, B. and D. A. Wood. 1997. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. *Proc. 24th Int'l Symposium on Computer Architecture* (June):229–240.

Farkas, K., Z. Vranesic, and M. Stumm. 1992. Cache Consistency in Hierarchical Ring-Based Multiprocessors. *Proc. Supercomputing '92* (November).

Feigel, C. P. 1994. TI Introduces Four-Processor DSP Chip. *Microprocessor Report* (March):28.

Felderman, R., et al. 1994. Atomic: A High Speed Local Communication Architecture. *Journal of High Speed Networks* 3(1):1–29.

Fenwick, D. M., D. J. Foley, W. B. Gist, S. R. VanDoren, and D. Wissell. 1995. The AlphaServer 8000 Series: High-End Server Platform Development. *Digital Technical Journal* 7(1):43–65.

Flanagan, J. L. 1994. Technologies for Multimedia Communications. *IEEE Proceedings* 82(4):590–603.

Flynn, M. J. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computing* C-21(September):948–960.

Fortune, S., and J. Wyllie. 1978. Parallelism in Random Access Machines. *Proc. 10th ACM Symposium on Theory of Computing* (May).

Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. 1988. *Solving Problems on Concurrent Processors, vol 1.* Englewood Cliffs, NJ: Prentice Hall.

Frailong, J.-L., et al. 1993. The Next Generation SPARC Multiprocessing System Architecture. *Proc. COMPCON* (spring):475–480.

Frank, S., H. Burkhardt III, and J. Rothnie. 1993. The KSR1: Bridging the Gap between Shared Memory and MPPs. *Proc. COMPCON, Digest of Papers* (spring):285–294.

Fu, J. W. C., and J. H. Patel. 1991. Data Prefetching in Multiprocessor Vector Cache Memories. *Proc. 18th Annual Symposium on Computer Architecture* (May):54–63.

Fu, J. W. C., J. H. Patel., and B. L. Janssens. 1992. Stride Directed Prefetching in Scalar Processors. *Proc. 25th Annual Int'l Symposium on Microarchitecture* (December):102–110.

Fuchs, H., G. Abram, and E. Grant. 1983. Near Real-Time Shaded Display of Rigid Objects. *Proc. SIGGRAPH.*

Galles, M., and E. Williams. 1993. Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor. *Proc. 27th Hawaii Int'l Conference on System Sciences Vol. I: Architecture* (January). Also in *SGI Challenge.* Edited by T. N. Mudge and B. D. Shriver. Los Alamitos, CA: IEEE Computer Society Press, 1994, 134–143.

Geist, A., A. Beguelin, and J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. 1994. *PVM 3.0 Users' Guide and Reference Manual.* Tech. Report ORNL/TM-12187, Oak Ridge, TN: Oak Ridge National Laboratory (February). *http://www.eece.ksu.edu/pvm3/ug.ps.*

Geist, A., A. Beguelin, J. Dongarra, R. Manchek, W. Jiang, and V. Sunderam. 1994. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* Cambridge, MA: MIT Press.

Geist, G. A., and V. S. Sunderam. 1992. Network Based Concurrent Computing on the PVM System. *Journal of Concurrency: Practice and Experience* 4(4):293–311.

Gharachorloo, K. 1995. *Memory Consistency Models for Shared-Memory Multiprocessors.* Ph.D. diss., Computer Systems Laboratory, Stanford University (December). Also published as Tech. Report #CSL-TR-95-685.

Gharachorloo, K., S. Adve, A. Gupta, M. Hill, and J. L. Hennessy. 1992. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing* 15(4):399–407.

Gharachorloo, K., A. Gupta, and J. L. Hennessy. 1991a. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. *Proc. 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (April):245–257.

———. 1991b.Two Techniques to Enhance the Performance of Memory Consistency Models. *Proc. Int'l Conference on Parallel Processing* (August): 1355–1364.

———. 1992. Hiding Memory Latency Using Dynamic Scheduling in Shared Memory Multiprocessors. *Proc. 19th Int'l Symposium on Computer Architecture* (May):22–33.

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Proc. 17th Int'l Symposium on Computer Architecture* (May):15–26.

Gillett, R. 1996. Memory Channel Network for PCI. *IEEE Micro* 16(1):12–18.

Gillett, R., M. Collins, and D. Pimm. 1996. Overview of Network Memory Channel for PCI. *Proc. IEEE Spring COMPCON '96* (February).

Gillett, R., and R. Kaufmann. 1997. Using Memory Channel Network. *IEEE Micro* 17(1):19–25.

Glass, C. J., and L. M. Ni. 1992. The Turn Model for Adaptive Routing. *Proc. Annual International Symposium on Computer Architecture (ISCA)* (May): 278–287.

Godiwala, N. D., and B. A. Maskas. 1995. The Second-Generation Processor Module for AlphaServer 2100 Systems. *Digital Technical Journal* 7(1).

Gokhale, M., B. Holmes, and K. Iobst. 1995. Processing in Memory: The Terasys Massively Parallel PIM Array. *IEEE Computer* 28(3):23–31.

Goldschmidt, S. R. 1993. *Simulation of Multiprocessors: Speed and Accuracy.* Ph.D. diss., Stanford University (June).

Golub, G., and C. Van Loan. 1997. *Matrix Computations 3e.* Baltimore, MD: Johns Hopkins University Press.

Goodman, J. R. 1983. Using Cache Memory to Reduce Processor-Memory Traffic. *Proc. 10th Annual Int'l Symposium on Computer Architecture* (June):124–131.

———. 1987. Coherency for Multiprocessor Virtual Address Caches. *Proc. Second Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA (October):72–81.

———. 1989. *Cache Consistency and Sequential Consistency.* Tech. Report #1006, University of Wisconsin-Madison, Computer Science Dept. (February).

Goodman, J. R., M. K. Vernon, P J. Woest. 1989. Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor. *Proc. Third Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (April):64–75.

Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. 1983. The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers* C-32(2):175–189.

Gottlieb, A., and C. P. Kruskal. 1984. Complexity Results for Permuting Data and Other Computations on Parallel Processors. *Journal of the ACM* 31(April):193–209.

Gottlieb, A., B. Luhachevsky, and L. Rudolph. 1983. Basic Techniques for the Efficient Coordination of Large Numbers of Cooperating Sequental Processes. *ACM Transactions on Programming Languages and Systems* 5(2).

Grafe, V. G., and J. E. Hoch. 1990. The Epsilon-2 Hybrid Dataflow Architecture. *Proc. COMPCON Spring '90*, San Francisco, CA (March):88–93.

Grahn, H., and P. Stenstrom. 1996. Evaluation of a Competitive-Update Cache Coherence Protocol with Migratory Data Detection. *Journal of Parallel and Distributed Computing* 39(2):168–180.

Grahn, H., P. Stenstrom, and M. Dubois. 1995. Implementation and Evaluation of Update-Based Protocols under Relaxed Memory Consistency Models. *Future Generation Computer Systems* 11(3):247–271.

Granuke, G., and S. Thakkar. 1990. Synchronization Algorithms for Shared Memory Multiprocessors. *IEEE Computer* 23(6):60–69.

Gray, J. 1991. *The Benchmark Handbook for Database and Transaction Processing Systems*. San Francisco: Morgan Kaufmann.

Green, S. A., and D. J. Paddon. 1990. A Highly Flexible Multiprocessor Solution for Ray Tracing. *The Visual Computer* 6:62–73.

Greenberg, R. I. and C. E. Leiserson. 1989. Randomized Routing on Fat-Trees. *Advances in Computing Research* 5:345–374

Greenwald, M., and D. R. Cheriton. 1996. The Synergy between Non-Blocking Synchronization and Operating System Structure. *Proc. Second Symposium on Operating System Design and Implementation, USENIX*, Seattle (October):123–136.

Gropp, W., E. Lusk, and A. Skjellum. 1994. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press.

Groscup, W. 1992. The Intel Paragon XP/S Supercomputer. *Proc. Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology* (November):262–273.

Gunther, K. D. 1981. Prevention of Deadlocks in Packet-Switched Data Transport Systems. *IEEE Transactions on Communication* C-29(4):512–24.

Gupta, A., J. L. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. 1991. Comparative Evaluation of Latency Reducing and Tolerating Techniques. *Proc. 18th Int'l Symposium on Computer Architecture* (May):254–263.

Gupta, A., and W.-D. Weber. 1992. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers* 41(7):794–810.

Gupta, A., W.-D. Weber, and T. Mowry. 1990. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache-Coherence Schemes. *Proc. Int'l Conference on Parallel Processing* I(August):312–321.

Gurd, J. R., C. C. Kerkham, and I. Watson. 1985. The Manchester Prototype Dataflow Computer. *Communications of the ACM* 28(1):34–52.

Gustafson, J. L. 1988. Reevaluating Amdahl's Law. *Communications of the ACM* 31(5):532–533.

Gustafson, J. L., and Q. O. Snell. 1994. *HINT: A New Way to Measure Computer Performance*. Tech. Report, Ames Laboratory, U.S. Dept. of Energy, Ames, IA.

Gustavson, D. 1992. The Scalable Coherence Interface and Related Standards Projects. *IEEE Micro* 12(1):10–22.

Gwennap, L. 1994a. Microprocessors Head Toward MP on a Chip. *Microprocessor Report* (May).

———. 1994b. PA-7200 Enables Inexpensive MP Systems. *Microprocessor Report* (March).

Hagersten, E. 1992. *Toward Scalable Cache Only Memory Architectures*. Ph.D. diss., Swedish Institute of Computer Science (October).

Hagersten, E., A. Landin, and S. Haridi. 1992. DDM—A Cache Only Memory Architecture. *IEEE Computer* 25(9):44–54.

Hanrahan, P., D. Salzman, and L. A. Aupperle. 1991. A Rapid Hierarchical Radiosity Algorithm. *Proc. SIGGRAPH* (July).

Hayashi, K., T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihata, and T. Shindo. 1994. AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler. *ACM SIGPLAN Notices* 29(11):196.

Heinlein, J., R. P. Bosch, Jr., K. Gharachorloo, M. Rosenblum, and A. Gupta. 1997. Coherent Block Data Transfer in the FLASH Multiprocessor. *Proc. 11th Int'l Parallel Processing Symposium* (April).

Heinlein, J., K. Gharachorloo, S. Dresser, and A. Gupta. 1994. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. *Proc. 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):38–50.

Heinrich, M., J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. 1994. The Performance Impact of Flexibility on the Stanford FLASH Multiprocessor. *Proc. 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):274–285.

Hennessy, J. L., and N. Jouppi. 1991. Computer Technology and Architecture: An Evolving Interaction. *IEEE Computer* 24(9):18–29.

Hennessy, J. L., and D. A. Patterson. 1996. *Computer Architecture: A Quantitative Approach*. 2nd ed. San Francisco: Morgan Kaufmann.

Herlihy, M. P. 1988. Impossibility and Universality Results for Wait-Free Synchronization. *Seventh ACM SIGACTS-SIGOPS Symposium on Principles of Distributed Computing* (August):276–290.

———. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems* 13(1):124–149.

———. 1993. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems* 15(5):745–770.

Herlihy, M. P., and J. E. B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. *Proc. 20th Annual Symposium on Computer Architecture*, San Diego, CA (May):289–301.

Herlihy, M. P., and J. Wing. 1987. Axioms for Concurrent Objects. *Proc. 14th ACM Symposium on Principles of Programming Languages* (January):13–26.

Hernquist, L. 1987. Performance Characteristics of Tree Codes. *Astrophysics Journal Supplement* 64(August):715–734.

Hey, A. J. G. 1991. The Genesis Distributed Memory Benchmarks. *Parallel Computing* 17:1111–1130.

High Performance Fortran Forum. 1993. High Performance Fortran Language Specification. *Scientific Programming* 2(1):1–270.

Hill, M. D., S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. A. Hodges, R. H. Katz, J. Ousterhut, and D. A. Patterson. 1986. Design Decisions in SPUR. *IEEE Computer* 19(10):8–22. Also in *Computers for Artificial Intelligence Processing*. Edited by B. W. Wah and C. V. Ramamoorthy. New York: John Wiley and Sons, 273–299.

Hill, M. D., and A. J. Smith. 1989. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers* C-38(12):1612–1630.

Hillis, W. D. 1985. *The Connection Machine*. Cambridge, MA: MIT Press.

Hillis, W. D., and G. L. Steele. 1986. Data Parallel Algorithms. *Communications of the ACM* 29(12):1170–1183.

Hillis, W. D., and L. W. Tucker. 1993. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM* 36(11):31–40.

Hirata, H., K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. 1992. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. *Proc. 19th Int'l Symposium on Computer Architecture* (May):136–145.

Hoare, C. A. R. 1978. Communicating Sequential Processes. *Communications of the ACM* 21(8):666–667.

Hockney, R. W., and C. R. Jesshope. 1988. *Parallel Computers* 2. London: Adam Hilger.

Holt, C., M. Heinrich, J. P. Singh, E. Rothberg and J. L. Hennessy. 1995. *The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors*. Tech. Report #CSL-TR-95-660, Computer Systems Laboratory, Stanford University (January).

Homewood M., and M. McLaren. 1993. Meiko CS-2 Interconnect Elan—Elite Design. *Hot Interconnects* (August).

Horiw, T., K. Hayashi, T. Shimizu, and H. Ishihata. 1993. Improving the AP1000 Parallel Computer Performance with Message Passing. *Proc. 20th Annual Int'l Symposium on Computer Architecture* (May):314–325

Horowitz, M. 1997. Limits of Electrical Signalling. *Hot Interconnects Keynote* (August)

Horst, R. 1995. TNet: A Reliable System Area Network. *IEEE Micro* 15(1):37–45.

Horst, R. W., and T. C. K. Chou. 1985. An Architecture for High Volume Transaction Processing. *Proc. 12th Annual Int'l Symposium on Computer Architecture* (June):240–245, Boston MA. (Tandem NonStop II)

Horst, R. W., R. L. Harris, and R. L. Jardine. 1990. Multiple Instruction Issue in the NonStop Cyclone Processor. *Proc. Annual International Symposium on Computer Architecture (ISCA)*, 216–226.

Hristea, C., D. Lenoski, and J. Keen. 1997. Measuring Memory Hierarchy Performance of Cache Coherent Multiprocessors Using Micro Benchmarks. *Proc. SC97* (November; all-Web conference proceeding).

Hunt, D. 1996. *Advanced Features of the 64-Bit PA-8000*. Palo Alto, CA: Hewlett Packard Corp.

IEEE Computer Society. 1993. *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE Standard 1596–1992. Washington, DC: IEEE Computer Society.

———. 1995. *IEEE Standard for Cache Optimization for Large Numbers of Processors Using the Scalable Coherent Interface (SCI)* Draft 0.35 (September). Washington, DC: IEEE Computer Society.

Iftode, L., C. Dubnicki, E. W. Felten, and K. Li. 1996. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. *Proc. Second Symposium on High Performance Computer Architecture* (February):14–25.

Iftode, L., J. P. Singh, and K. Li. 1996a. Understanding Application Performance on Shared Virtual Memory Systems. *Proc. 23rd Int'l Symposium on Computer Architecture* (April):122–133.

———. 1996b. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Proc. Symposium on Parallel Algorithms and Architectures* (June).

Intel Corporation. 1994. *1750, 1860, 1960 Processors and Related Products*. Santa Clara, CA: Intel Corp.

———. 1996. *Pentium® Pro Family Developer's Manual*. Santa Clara, CA: Intel Corp

Jeremiassen, T. E., and S. J. Eggers. Eliminating False Sharing. *Proc. 1991 Int'l Conference on Parallel Processing* (August):377–381.

Jiang, D., H. Shan, and J. P. Singh. 1997. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors. *Proc. Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (June):217–229.

Jiang, D., and J. P. Singh. 1998. A Methodology and an Evaluation of the SGI Origin2000. *Proc. SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (June).

Joe, T. 1995. *COMA-F: A Non-Hierarchical Cache Only Memory Architecture*. Ph.D. diss., Computer Systems Laboratory, Stanford University (March).

Joe, T., and J. L. Hennessy. 1994. Evaluating the Memory Overhead Required for COMA Architectures. *Proc. 21st Int'l Symposium on Computer Architecture* (April):82–93.

Joerg, C. F. 1994. *Design and Implementation of a Packet Switched Routing Chip*. Tech. Report MIT/LCS/TR-482, MIT Laboratory for Computer Science (August).

Joerg, C. F., and A. Boughton. 1991. The Monsoon Interconnection Network. *Proc. ICCD* (October).

Johnson, M. 1991 *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice Hall.

Jordan, H. F. 1985. HEP Architecture, Programming, and Performance. In *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*. Edited by J S. Kowalik. Cambridge, MA: MIT Press, 8.

Jouppi, N P. 1990 Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *Proc. 17th Annual Symposium on Computer Architecture* (June):364–373.

Jouppi, N. P., and P. Ranganathan. 1997. *The Relative Importance of Memory Latency, Bandwidth, and Branch Limits to Performance*. Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. Presented at the Annual Int'l Symposium on Computer Architecture (ISCA) '97 (June).

Jouppi, N P., and D. Wall. 1989. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. *ASPLOS III*, 272–282.

Kagi, A., D. Burger, and J. R. Goodman. 1997. Efficient Synchronization: Let Them Eat QOLB. *Proc 24th Int'l Symposium on Computer Architecture (ISCA)* (June):170–180.

Karlin, A. R., M. S. Manasse, L. Rudolph and D. D. Sleator. 1986. Competitive Snoopy Caching *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science*.

Karol, M., M. Hluchyj, and S. Morgan. 1987. Input versus Output Queueing on a Space Division Packet Switch. *IEEE Transactions on Communications* 35(12):1347–1356.

Karp, R., U. Vazirani, and V. Vazirani. 1990. An Optimal Algorithm for On-Line Bipartite Matching. *Proc. 22nd ACM Symposium on the Theory of Computing* (May):352–358.

Kaxiras, S. 1996. Kiloprocessor Extensions to SCI. *Proc. 10th Int'l Parallel Processing Symposium*

Kaxiras, S., and J. Goodman. The GLOW Cache Coherence Protocol Extensions for Widely Shared Data. *Proc. Int'l Conference on Supercomputing* (May):35–43.

Keeton, K K., T. E. Anderson, and D. A. Patterson. 1995. LogP Quantified: The Case for Low-Overhead Local Area Networks. *Hot Interconnects III: Symposium on High Performance Interconnects* (August).

Keleher, P., A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proc. Winter USENIX Conference* (January):15–132.

Keleher, P., A. L. Cox, and W. Zwaenepoel. 1992. Lazy Consistency for Software Distributed Shared Memory. *Proc. 19th Int'l Symposium on Computer Architecture* (May):13–21.

Kermani, P., and L. Kleinrock. 1979. Virtual Cut-Through: A New Computer Communication Switching Technique. *Computer Networks* 3(September):267–286.

Kessler, R. E., and J. L. Schwarzmeier. 1993. Cray T3D: A New Dimension for Cray Research. *Proc Papers, COMPCON Spring'93*, San Francisco (February):176–182.

Knuth, D. E. 1966. Additional Comments on a Problem in Concurrent Programming Control. *Communications of the ACM* 9(5):321–322.

Koebel, C., D. Loveman, R. Schreiber, G. Steele, and M. Zosel. 1994. *The High Performance Fortran Handbook*. Cambridge, MA: MIT Press.

Koeninger, R. K., M. Furtney, and M. Walker. 1994. A Shared Memory MPP from Cray Research. *Digital Technical Journal* 6(2):8–21.

Kogge, P. M. 1994. EXECUBE—A New Architecture for Scalable MPPs. *1994 Int'l Conference on Parallel Processing* (August):177–184.

Kontothanassis, L. I., G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S Dwarkadas, and M. Scott. 1997. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. *Proc. 24th Int'l Symposium on Computer Architecture* (June).

Kontothanassis, L. I., and M. L. Scott. 1996. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. *Proc. Second Symposium on High Performance Computer Architecture* (February):166–177.

Kostantantindou, S., and L. Snyder. 1991. Chaos Router: Architecture and Performance. *Proc. 18th Annual Symposium on Computer Architecture* (May):212–221.

Krishnamurthy, A., K. E. Schauser, C. J. Scheiman, R. Y. Wang, D. E. Culler, and K. Yelick. 1996. Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines. *ACM SIGPLAN Notices* 31(9):37–48.

Krishnamurthy, A., and K. A. Yelick. 1994. Optimizing Parallel SPMD Programs. *Seventh Annual Workshop on Languages and Compilers for Parallel Computing* Ithaca, NY (August).

———. 1995. Optimizing Parallel Programs with Explicit Sychronization. *Programming Language Design and Implementation*, 196–204.

——— 1996. Analyses and Optimizations for Shared Address Space Programs. *JPDC* 38(2):130–144.

Kroft, D. 1981. Lockup-Free Instruction Fetch/Prefetch Cache Organization. *Proc. Eighth Int'l Symposium on Computer Architecture* (May):81–87.

Kronenberg, N. P., H Levy, and W. D. Strecker. 1986. Vax Clusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems* 4(2):130–146.

Kruskal, C. P., and M. Snir 1983. The Performance of Multistage Interconnection Networks for Multiprocessors. *IEEE Transactions on Computers* C-32(12):1091–1098.

Kubiatowicz, J., and A. Agarwal. 1993. The Anatomy of a Message in the Alewife Multiprocessor. *Proc. Int'l Conference on Supercomputing* (July):195–206.

Kuehn, J. T, and B J. Smith 1988. The Horizon Supercomputing System: Architecture and Software. *Proc. Supercomputing '88* (November):28–34.

Kumar, M. 1992. Unique Design Concepts in GF11 and Their Impact on Performance. *IBM Journal of Research and Development* 36(6):990–1000.

Kumar, V., A. Grama, A. Gupta, and G. Karypis. 1994 *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City, CA: Benjamin/Cummings Publishing Company

Kumar, V., and A. Gupta. 1991. Analysis of Scalability of Parallel Algorithms and Architectures: A Survey. *Proc. Int'l Conference on Supercomputing* (June):396–405.

Kung, H. T., R. Sansom, S. Schlick, P. A. Steenkiste, M. Arnould, F. J. Bitz, F. Christianson, E. C. Cooper, O. Menzilcioglu, D. Ombres, and B. Zill. 1989. Network-Based Multicomputers. An Emerging Parallel Architecture. *Proc. Supercomputing '91 Conference* (November):664–673

Kurihara, K., D. Chaiken, and A. Agarwal. 1991. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. *Proc. Int'l Symposium on Shared Memory Multiprocessing* (April):91–101.

Kuskin, J., D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. 1994. The Stanford FLASH Multiprocessor. *Proc. 21st Int'l Symposium on Computer Architecture* (April): 302–313.

Lam, M. S., and R. P. Wilson. 1992. Limits on Control Flow on Parallelism. *Proc. 19th Annual Int'l Symposium on Computer Architecture* (May):46–57.

Lamport, L. 1979. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers* C-28(9):690–691.

Larus, J. R., B. Richards, and G. Viswanathan. 1996. Parallel Programming in C**: A Large-Grain Data-Parallel Programming Language. In *Parallel Programming Using C++*. Edited by G. V. Wilson and P. Lu. Cambridge, MA: MIT Press.

Laudon, J. P. 1994. *Architectural and Implementation Tradeoffs in Multiple-Context Processors*. Ph.D. diss., Stanford University, Stanford, California. Also published as Tech. Report #CSL-TR-94-634, Computer Systems Laboratory, Stanford University (May).

Laudon, J., A. Gupta, and M. Horowitz. 1994. *Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors*. In *Multithreaded Computer Architecture: A Summary of the State of the Art*. Edited by R. A. Iannucci. Dordrecht, Germany; Norwell, MA: Kluwer Academic Publishers, 167–200.

Laudon, J. P., and D. Lenoski. 1997. The SGI Origin: A ccNUMA Highly Scalable Server. *Proc. 24th Int'l Symposium on Computer Architecture*.

Lawton, J. V., J. J. Brosnan, M. P. Doyle, S.D. O'Riodain, and T. G. Reddin. 1996. Building a High-Performance Message-Passing System for MEMORY CHANNEL Clusters. *Digital Technical Journal* 8(2):96–116

Lee, C. G. 1989. Multi-Step Gradual Rounding. *IEEE Transactions on Computers* 38(4):595–600.

Lee, R. L., A. Y. Kwok, and F. A. Briggs. 1991. The Floating Point Performance of a Superscalar SPARC Processor. *Proc. 4th Symposium on Architectural Support for Programming Languages and Operating Systems* (April):28–37.

Leighton, F. T. 1992. *Introduction to Parallel Algorithms and Architectures*. San Francisco: Morgan Kaufmann.

Leiserson, C. E. 1985. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers* C-34(10):892–901.

Leiserson, C. E., Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, and R. Zak. 1996. The Network Architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing* 33(2):145–158. Also in *Proc. Fourth Symposium on Parallel Algorithms and Architectures* '92 (June):272–285.

Lenoski, D. 1992. *The Stanford DASH Multiprocessor*. Ph.D. diss., Computer Systems Laboratory, Stanford University.

Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. 1990. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. *Proc. 17th Int'l Symposium on Computer Architecture* (May):148–159.

Lenoski, D., J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. L. Hennessy. 1992. The DASH Prototype: Implementation and Performance. *Proc. 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia (May):92–103.

———. 1993. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems* 4(1):41–61.

Li, K., and P. Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems* 7(4):321–359.

Li, S.-Y. 1988. Theory of Periodic Contention and Its Application to Packet Switching. *Proc. INFOCOM '88* (March):320–325.

Lim, B.-H., and A. Agarwal. 1994. Reactive Syncronization Algorithms for Multiprocessors. *Proc. Sixth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 25–35

Linder, D., and J. Harden. 1991. An Adaptive Fault Tolerant Wormhole Strategy for k-ary n-cubes. *IEEE Transactions on Computer* C-40(1):2–12.

Lipton, R., and J. Sandberg. 1988. *PRAM: A Scalable Shared Memory.* Tech. Report #CS-TR-180–88, Computer Science Dept., Princeton University (September).

Litzkow, M., M. Livny, and M. W. Mutka. 1988. Condor—A Hunter of Idle Workstations. *Proc. Eighth Int'l Conference of Distributed Computing Systems* (June):104–111.

Lo, J. L., S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. 1997. Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems* (August).

Lonergan, W., and P. King. 1961. Design of the B 5000 System. *Datamation* 7(5):28–32

Lovett, T., and R. Clapp. 1996. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. *Proc. 23rd Int'l Symposium on Computer Architecture* (May):308–317.

Luk, C.-K., and T. C. Mowry. 1996. Compiler-Based Prefetching for Recursive Data Structures. *Proc. Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (October):222–233.

Lukowsky, J., and S. Polit. 1997 (date accessed). IP Packet Switching on the GIGAswitch/FDDI System. *http://www.networks.digital.com:80/dr/techart/gsfip-mn.html.*

Mainwaring, A., B. Chun, S. Schleimer, and D. Wilkerson. 1997. System Area Network Mapping. *Proc. Ninth Annual ACM Symposium on Parallel Algorithms and Architecture*, Newport, RI (June):116–126.

Mainwaring, A., and D. E. Culler. 1996. *Active Message Applications Programming Interface and Communication Subsystem Organization.* Tech. Report CSD-96-918, University of California at Berkeley.

Martin, R. 1994. HPAM: An Active Message Layer of a Network of Workstations. Presented at *Hot Interconnects II* (August).

Massalin, H., and C. Pu. 1991. *A Lock-Free Multiprocessor OS Kernel.* Tech. Report CUCS-005-01, Columbia University, Computer Science Dept. (October).

Matelan, N. 1985. The FLEX/32 Multicomputer. *Proc. 12th Annual Int'l Symposium on Computer Architecture*, Boston, MA. (Flex) (June):209–213

May, C., E. Silha, R. Simpson, and H. Warren, eds. 1994. *The PowerPC Architecture: A Specification for a New Family of RISC Processors.* San Francisco: Morgan Kaufmann.

McCreight, E. 1984. *The Dragon Computer System: An Early Overview.* Tech. Report, Xerox Corp. (September).

Mellor-Crummey, J. and M. Scott. 1991. Algorithms for Scalable Synchronization on Shared Memory Mutiprocessors. *ACM Transactions on Computer Systems* 9(1):21–65.

Melvin, S., and Y. Patt. 1991. Exploiting Fine-Grained Parallelism through a Combination of Hardware and Software Techniques. *Proc. Annual Int'l Symposium on Computer Architecture (ISCA)*, 287–296.

Michael, M., and M. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. *Proc. 15th Annual ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA (May): 267–276.

Minnich, R., D. Burns, and F. Hady. 1995. The Memory-Integrated Network Interface. *IEEE Micro* 15(1):11–20.

MIPS Technologies. 1991. *MIPS R4000 User's Manual*. Mountain View, CA: MIPS Technologies.

———. 1996. *R10000 Microprocessor User's Manual, Version 1.1* (January). Mountain View, CA: MIPS Technologies.

Miyoshi, H.; M. Fukuda, T. Iwamiya, T. Nakamura, M. Tuchiya, M. Yoshida, K. Yamamoto, Y. Yamamoto, S. Ogawa, Y. Matsuo, T. Yamane, M. Takamura, M. Ikeda, S. Okada, Y. Sakamoto, T. Kitamura, H. Hatama, M. Kishimoto, M. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. Sansom, S. Schlick, P. A. Steenkiste, and B. Zill. 1994. Development and Achievement of NAL Numerical Wind Tunnel (NWT) for CFD Computations. *Proc. Supercomputing '94*, Washington, DC (November):685–692.

Mowry, T. C. 1994. *Tolerating Latency through Software-Controlled Data Prefetching*. Ph.D. diss., Computer Systems Laboratory, Stanford University. Also published as Tech. Report #CSL-TR-94-628, Computer Systems Laboratory, Stanford University (June).

MPI Forum. 1993. *Document for a Standard Message-Passing Interface*. Tech. Report CS-93-214, University of Tennessee, Knoxville, Computer Science Dept. (November).

———. 1994. MPI: A Message Passing Interface. *Int'l Journal of Supercomputing Applications* 8(3/4). Special Issue on MPI. (updated 5/95). Also published in *Proc. Supercomputing '93 Conference* (May). Los Alamitos, CA: IEEE Computer Society Press, 878–883. Updated spec at *http://www.mcs.anl.gov/mpi/*.

Mukherjee, S., and M. Hill. 1997. A Case for Making Network Interfaces Less Peripheral. *Hot Interconnects* (August).

NAS Parallel Benchmarks. 1998 (date accessed). *http://science.nas.nasa.gov/Software/NPB/*.

Nayfeh, B. A., L. Hammond, K. Olukoton. 1996. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. *Proc. 23rd Annual Int'l Symposium on Computer Architecture* (May). New York: ACM Press, 67–77.

Nestle, E., and A. Inselberg. 1985. The Synapse N+1 System: Architectural Characteristics and Performance Data of a Tightly-Coupled Multiprocessor System. *Proc. 12th Annual Int'l Symposium on Computer Architecture*, Boston, MA (Synapse) (June):233–239.

Ngai, J., and C. Seitz. 1989. A Framework for Adaptive Routing in Multicomputer Networks. *Proc. 1989 Symposium on Parallel Algorithms and Architectures* (June):2–10.

Nickolls, J. R. 1990. The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer. *COMPCON Spring '90, Digest of Papers*, San Francisco, CA (February/March):25–28.

Nikhil, R. S., and Arvind. 1989. Can Dataflow Subsume von Neumann Computing? *Proc. 16th Annual Int'l Symposium on Computer Architecture* (May):262–72.

Nikhil, R., G. Papadopoulos, and Arvind. 1993. *T: A Multithreaded Massively Parallel Architecture. *Proc. Annual Int'l Symposium on Computer Architecture (ISCA) '93* (May):156–167.

Noakes, M. D. D. A. Wallach, and W. J. Dally. 1993. The J-Machine Multicomputer: An Architectural Evaluation. *Proc. 20th Int'l Symposium on Computer Architecture* (May):224–235.

Nuth, P., and W. J. Dally. 1992. The J-Machine Network. *Proc. Int'l Conference on Computer Design: VLSI in Computers and Processors* (October).

——— 1995. The Named-State Register File: Implementation and Performance. *Proc. First Int'l Symposium on High-Performance Computer Architecture* (January):4–13.

O'Krafka, B., and A. Newton. 1990. An Empirical Evaluation of Two Memory-Efficient Directory Methods. *Proc. 17th Int'l Symposium on Computer Architecture* (May):138–147.

Office of Science and Technology Policy. 1993. *Grand Challenges 1993: High Performance Computing and Communications, A Report by the Committee on Physical, Mathematical, and Engineering Sciences*. Washington, DC: Office of Science and Technology Policy.

Ohara, M. 1996. *Producer-Oriented versus Consumer-Oriented Prefetching: A Comparison and Analysis of Parallel Application Programs*. Ph.D. diss., Computer Systems Laboratory, Stanford University. Available as Tech. Report #CSL-TR-96-695, Stanford University (June).

Olukotun, K., B. A. Nayfeh, L. Hammond, K.Wilson and K. Chang. 1996. The Case for a Single-Chip Multiprocessor. *Proc. ASPLOS* (October):2–11.

Omondi, A. R. 1994. Ideas for the Design of Multithreaded Pipelines. In *Multithreaded Computer Architecture: A Summary of the State of the Art*. Edited by R. Iannucci. Dordrecht, Germany; Norwell, MA: Kluwer Academic Publishers, 1994. See also A. R. Omondi, Design of a High Performance Instruction Pipeline. *Computer Systems Science and Engineering* 6(1):13–29 (1991).

Pacheco, P. 1996. *Parallel Programming with MPI*. San Francisco: Morgan Kaufmann.

Padegs, A. 1981. System/360 and Beyond. *IBM Journal of Research and Development* 25(5):377–390.

Pai, V. S., P. Ranganathan, S. V. Adve, and T. Harton. 1996. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. *Proc. Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (October):12–23.

Papadimitriou, C. H. 1979. The Serializability of Concurrent Database Updates. *Journal of the ACM* 26(4):631–653.

Papadopoulos, G. M., and D. E. Culler. 1990. Monsoon: An Explicit Token-Store Architecture. *Proc. 17th Annual Int'l Symposium on Computer Architecture*, Seattle, WA (May):82–91.

Papamarcos, M., and J. Patel. 1984. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. *Proc. 11th Annual Int'l Symposium on Computer Architecture* (June):348–354.

PARKBENCH Committee. 1994. Public International Benchmarks for Parallel Computers. *Scientific Programming* 3(2). Also published as Tech. Report CS93-213, University of Tennessee, Knoxville, Dept. of Computer Science (November).

Patterson, D. A. 1995. Microprocessors in 2020. *Scientific American* (September).

Patterson, D. A., T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K.Yelick. 1997. A Case for Intelligent RAM. *IEEE Micro* 17(2):34–44.

Peterson, L., and B. Davie. 1996. *Computer Networks*. San Francisco: Morgan Kaufmann.

Pfeiffer, W., S. Hotovy, N. Nystrom, D. Rudy, T. Sterling, M. Straka. 1995 (date accessed). JNNIE: The Joint NSF-NASA Initiative on Evaluation. *http://www.tc.cornell.edu/JNNIE/finrep/jnnie.html*.

Pfister, G. F. 1995. *In Search of Clusters—The Coming Battle for Lowly Parallel Computing*. Englewood Cliffs, NJ: Prentice Hall.

Pfister, G. F., W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliff, E. A. Melton, V. A Norton, and J. Weiss. 1985. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. *Proc. Int'l Conference on Parallel Processing* (August):264–771.

Pfister, G. F., and V. A. Norton. 1985. Hot Spot Contention and Combining Multistage Interconnection Networks. *IEEE Transactions on Computers* C-34(10).

Pierce, P. 1988. The NX/2 Operating System. *Proc. Third Conference on Hypercube Concurrent Computers and Applications* (January):384-390.

Pierce, P., and G. Regnier. 1994. The Paragon Implementation of the NX Message Passing Interface. *Proc. Scalable High-Performance Computing Conference* (May):184–90.

Porter, R. E. 1960. *Datamation* 6(1):8–14.

Przybylski, S., M. Horowitz, J. L. Hennessy. 1988. Performance Tradeoffs in Cache Design. *Proc. 15th Annual Symposium on Computer Architecture* (May):290–298.

Ranganathan, P., V. S. Pai, H. Abdel-Shafi, and S. V. Adve. 1997. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. *Proc. 24th Int'l Symposium on Computer Architecture* (June).

Ratner, J. 1985. Concurrent Processing: A New Direction in Scientific Computing. *Proc. 1985 National Computing Conference*, 835.

Reddaway, S. F. 1973. DAP—A Distributed Array Processor. *First Annual Int'l Symposium on Computer Architecture* (December):61–65.

Reinhardt, S. K., M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. 1993. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May):48–60.

Reinhardt, S. K., J. R. Larus, and D. A. Wood. 1994. Tempest and Typhoon: User-Level Shared Memory. *Proc. 21st Int'l Symposium on Computer Architecture* (April):325–337.

Reinhardt, S. K., R. W. Pfile, and D. A. Wood. 1996. Decoupled Hardware Support for Distributed Shared Memory. *Proc. 23rd Int'l Symposium on Computer Architecture* (May):34–43.

Rettberg, R., W. Crowther, P. Carvey, and R. Tomlinson. 1990. The Monarch Parallel Processor Hardware Design. *IEEE Computer* (April):18–30.

Rettberg, R., and R. Thomas. 1986. Contention is No Obstacle to Shared-Memory Multiprocessing. *Communications of the ACM* 29(12):1202–1212.

Rinard, M. C., D. J. Scales, and M. S. Lam. 1993. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer* 26(6).

Rodgers, D. 1985. Improvements on Multiprocessor System Design. *Proc. 12th Annual Int'l Symposium on Computer Architecture*, Boston, MA (Sequent B8000) (June):225–231.

Rosenblum, M., S. A. Herrod, E. Witchel, and A. Gupta. 1995. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology* 3(4).

Rosenburg, B. 1989. Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors. *Proc. Symposium on Operating Systems Principles* (December).

Rothberg, E., J. P. Singh, and A. Gupta. 1993. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. *Proc. 20th Int'l Symposium on Computer Architecture* (May):14–25.

Russel, R. M. 1978. The CRAY-1 Computer System. *Communications of the ACM* 21(1):63–72.

Saavedra-Barrera, R. H., D. E. Culler, T. von Eicken. 1990. Analysis of Multithreaded Architectures for Parallel Computing. *Second Annual ACM Symposium on Parallel Algorithms and Architectures* (July):169–178.

Saavedra, R. H., R. S. Gaines, and M. J. Carlton. 1993. Micro Benchmark Analysis of the KSR1. *Proc. Supercomputing '93*, Portland, OR (November):202–213.

Saavedra, R. H., and A. J. Smith. 1996. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Transactions on Computer Systems* 14(4):344–384.

Sakai, S., Y. Kodama, and Y. Yamaguchi. 1991. Prototype Implementation of a Highly Parallel Dataflow Machine EM4. *Proc. Fifth Int'l Parallel Processing Symposium*, Anaheim, CA (April/May):278–286.

Salmon, J. 1990. *Parallel Hierarchical N-body Methods*. Ph.D. diss., California Institute of Technology.

Salmon, J. K., M. S. Warren, and G. S. Winckelmans. 1994. Fast Parallel Treecodes for Gravitational and Fluid Dynamical N-body Problems. *Intl. Journal of Supercomputer Applications* 8:129–142.

Samanta, R., A. Bilas, L. Iftode, and J. P. Singh. 1998. Home-Based SVM Protocols for SMP Clusters: Design, Simulations, Implementation, and Performance. *Proc. 23rd Annual Int'l Symposium on Computer Architecture* (February).

Saulsbury, A., F. Pong, and A. Nowatzyk. 1996. Missing the Memory Wall: The Case for Processor/Memory Integration. *Proc. 23rd Annual Int'l Symposium on Computer Architecture* (May):90–101.

Saulsbury, A., T. Wilkinson, J. Carter, and A. Landin. 1995. An Argument For Simple COMA. *Proc. First IEEE Symposium on High Performance Computer Architecture* (January):276–285.

Savage, J. 1985. Parallel Processing as a Language Design Problem. *Proc. 12th Annual Int'l Symposium on Computer Architecture*, Boston, MA (Myrias 4000) (June):221–224.

Scales, D. J., K. Gharachorloo, and C. A. Thekkath. 1996. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. *Proc. Seventh Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):174–185.

Scales, D. J., and M. S. Lam. 1994. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. *Proc. First Symposium on Operating System Design and Implementation* (November):101–114.

Schanin, D.J. 1986. The Design and Development of a Very High Speed System Bus—The Encore Multimax Nanobus. In *Proc. Fall Joint Computer Conference (Encore)*, Dallas, TX (November). Edited by H. S. Stone. Los Alamitos: IEEE Computer Society Press, 410–418.

Schauser, K. E., and C. J. Scheiman. 1995. Experience with Active Messages on the Meiko CS-2. *Proc. Ninth Int'l Symposium on Parallel Processing* (IPPS'95) (April):140–149.

Scheurich, C. and M. Dubois. 1987. Correct Memory Operation of Cache-Based Multiprocessors. *Proc. 14th Int'l Symposium on Computer Architecture* (June):234–243.

Schoinas, I., B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. 1994. Fine-Grain Access Control for Distributed Shared Memory. *Proc. 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):297–306.

Schroeder, M. D., A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. 1991. Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links. *IEEE Journal on Selected Areas in Communications* 9(8):1318–1335

Schwiebert, L., and D. N. Jayasimha. 1995. A Universal Proof Technique for Deadlock-Free Routing in Interconnection Networks. *Symposium on Parallel Algorithms and Architecture* (July):175–184.

Scott, S. 1991. A Cache-Coherence Mechanism for Scalable Shared-Memory Multiprocessors. *Proc. Int'l Symposium on Shared Memory Multiprocessing* (April):49–59.

———. 1996. Synchronization and Communication in the T3E Multiprocessor. *Proc. Seventh Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):26–36, Cambridge, MA,.

Scott, S., and J. R. Goodman. 1993. Performance of Pruning Cache Directories for Large-Scale Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 4(5):520–534.

———. 1994. The Impact of Pipelined Channels on k-ary n-Cube Networks. *IEEE Transactions on Parallel and Distributed Systems* 5(1):2–16.

Scott, S., M. Vernon, and J. R. Goodman. 1992. Performance of the SCI Ring. *Proc. 19th Int'l Symposium on Computer Architecture* (May):403–414.

Seitz, C. L. 1984. Concurrent VLSI Architectures. *IEEE Transactions on Computers* 33(12):1247–1265.

———. 1985. The Cosmic Cube. *Communications of the ACM* 28(1):22–33.

Seitz, C. L., and W.-K. Su. 1993. A Family of Routing and Communication Chips Based on Mosaic. *Proc. of Univ. of Washington Symposium on Integrated Systems*. Cambridge, MA: MIT Press, 320–337.

Shah, G., J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. 1998. Performance and Experience with LAPI—A New High-Performance Communication Library for the IBM RS/6000 SP. *Twelfth Int'l Parallel Processing Symposium* (March):260–266

Shasha, D., and M. Snir. 1988. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Operating Systems* 10(2):282–312.

Shimada, T., K. Hiraki, and K. Nishida. 1984. An Architecture of a Data Flow Machine and Its Evaluation. *Proc. COMPCON '84*, 486–90.

Simoni, R., and M. Horowitz. 1991. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. *Proc. Int'l Symposium on Shared Memory Multiprocessing* (April): 72–81.

Sindhu, P., J.-M. Frailong, and M. Cekleov. 1991. *Formal Specification of Memory Models*. Tech. Report (PARC) CSL-91-11. Xerox Corp., Palo Alto Research Center, Palo Alto, CA.

Sindhu, P., et al. 1993. XDBus: A High-Performance, Consistent, Packet Switched VLSI Bus. *Proc. COMPCON* (Spring): 338–344.

Singh, J. P. 1993. *Parallel Hierarchical N-body Methods and Their Implications for Multiprocessors*. Ph.D. diss., Tech. Report #CSL-TR-93-565, Stanford University (March).

———. 1998. *Some Aspects of Controlling Scheduling in Hardware Control Prefetching*. To be published as Tech. Report, Princeton University, Computer Science Dept.

Singh, J. P., A. Gupta, and M. Levoy. 1994. Parallel Visualization Algorithms: Performance and Architectural Implications. *IEEE Computer* 27(6).

Singh, J. P., J. L. Hennessy, and A. Gupta. 1993. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer* 26(7):42–50.

———. 1995. Implications of Parallel Hierarchical N-body Applications for Multiprocessors. *ACM Transactions on Computer Systems* (May).

Singh, J. P., C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy. 1995. Load Balancing and Data Locality in Hierarchial N-body Methods: Barnes-Hut, Fast Multipole and Radiosity. *Journal of Parallel and Distributed Computing* (June).

Singh, J. P., T. Joe, A. Gupta, and J. L. Hennessy. 1993. An Empirical Comparison of the KSR-1 and DASH Multiprocessors. Proc. *Supercomputing '93* (November).

Singh, J. P., E. Rothberg, and A. Gupta. 1994. Modeling Communication in Parallel Algorithms: A Fruitful Interaction between Theory and Systems? *Proc. 10th Annual ACM Symposium on Parallel Algorithms and Architectures*.

Singh, J. P., W.-D. Weber, and A. Gupta. 1992. SPLASH: The Stanford ParallelL Applications for SHared Memory. *Computer Architecture News* 20(1):5–44.

Sites, R. L., ed. 1992. *Alpha Architecture Reference Manual*. Hudson, MA: Digital Press, Digital Equipment Corp.

Slater, M. 1994. Intel Unveils Multiprocessor System Specification. *Microprocessor Report* (May):12–14.

Slotnick, D. L. 1967. Unconventional Systems. *Proc. AFIPS Spring Joint Computer Conference* 30: 477–481.

Slotnick, D. L., W. C. Borck, and R. C. McReynolds. 1962. The Solomon Computer. *Proc. AFIPS Fall Joint Computer Conference* 22: 97–107.

Smith, A. J. 1982. Cache Memories. *ACM Computing Surveys* 14(3):473–530.

Smith, B. J. 1981. Architecture and Applications of the HEP Multiprocessor Computer System. *Proc. SPIE: Real-Time Signal Processing IV* 298(August):241–248.

———. 1985. The Architecture of HEP. In *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*. Edited by J.S. Kowalik. Cambridge, MA: MIT Press, 41–55.

Smith, M. D., M. Johnson, and M. A. Horowitz. 1989. Limits on Multiple Instruction Issue. *Proc. Third Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 290–302, Apr.

Snir, M., S. Otto, S. H. Lederman, D. Walker, and J. Dongarra. 1995. *MPI: The Complete Reference*. Cambridge, MA: MIT Press.

Sohi, G., S. Breach, and T. N. Vijaykumar. 1995. Multiscalar Processors. *Proc. 22nd Annual Int'l Symposium on Computer Architecture* (June):414–425.

SPEC (Standard Performance Evaluation Corporation). 1995 (date accessed). *http://www.specbench.org/*. (SPEC Benchmark Suite Release 1.0., 1989)

Spertus, E., S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, W. J. Dally. 1993. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. *Proc. 20th Annual Symposium on Computer Architecture* (May):302–313

Stenstrom, P., T. Joe, and A. Gupta. 1992. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. *Proc. 19th Int'l Symposium on Computer Architecture* (May):80–91.

Stets, R., S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. 1997. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. *Proc. 16th ACM Symposium on Operating Systems Principles* (October).

Stone, H. S. 1970. A Logic-in-Memory Computer. *IEEE Transactions on Computers* C-19(1):73–78.

Stunkel, C. B., D. G. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao. 1994. The SP-1 High Performance Switch. *Proc. Scalable High Performance Computing Conference* (May):150–157 Knoxville, TN.

Stunkel, C. B., et al. 1998 (date accessed). *The SP2 Communication Subsystem. http://ibm.tc.cornell.edu/ibm/pps/doc/css/css.ps.*

SUN Microsystems. 1991. *The SPARC Architecture Manual*. #800-199-12, Version 8 (January). Mountain View, CA: SUN Microsystems.

Sunderam, V. S. 1990. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience* 2(4):315–339.

Sunderam, V. S., J. Dongarra, A. Geist, and R Manchek. 1994. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing* 20(4):531–547.

Swan, R. J., A. Bechtolsheim, K.-W. Lai, and J. K. Ousterhout. 1977. The Implementation of the CM* Multi-Microprocessor. *Proc. AFIPS Conference/National Computer Conference* (46):645–655.

Swan, R. J., S. H. Fuller, and D. P. Siewiorek. 1977. CM*—A Modular, Multi-Microprocessor. *Proc. AFIPS Conference/National Computer Conference* (46):637–44.

Sweazey, P., and A. J. Smith. 1986. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. *Proc. 13th Int'l Symposium on Computer Architecture* (May):414–423.

Tamir, Y., and G. L. Frazier. 1988. High-Performance Multi-Queue Buffers for VLSI Communication Switches. *Proc. 15th Annual Int'l Symposium on Computer Architecture*, 343–354.

Tanenbaum, A. S., and A. S. Woodhull. 1997. *Operating System Design and Implementation 2nd ed.* Englewood Cliffs, NJ: Prentice Hall.

Tang, C. 1976. Cache Design in a Tightly Coupled Multiprocessor System. *Proc. AFIPS Conference* (June):749–753.

Teller, P. 1990. Translation-Lookaside Buffer Consistency. *IEEE Computer* 23(6):26–36.

Thacker, C., L. Stewart, and E. Satterthwaite, Jr. 1988. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers* 37(8):909–920.

Thapar, M., and B. Delagi. 1990. Stanford Distributed-Directory Protocol. *IEEE Computer* 23(6):78–80.

Thekkath, R., A. P. Singh, J. P. Singh, J. Hennessy, and S. John. 1997. An Application-Driven Evaluation of the Convex Exemplar SP-1200. *Proc. Int'l Parallel Processing Symposium* (June).

Thompson, M., J. Barton, T. Jermoluk, and J. Wagner. 1988. Translation Lookaside Buffer Synchronization in a Multiprocessor System. *Proc. USENIX Technical Conference* (February).

Thornton, J. E. 1964. Parallel Operation in the Control Data 6600. *AFIPS Proc. Fall Joint Computer Conference*, Part 2 26:33–40. Reprinted in Siework, Bell, and Newell. 1982. *Computer Structures: Principles and Examples*. New York: McGraw-Hill.

Tomasulo, R. M. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11(1):25–33.

Torrellas, J., M. S. Lam, and J. L. Hennessy. 1994. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers* 43(6):651–663.

Transaction Processing Council. 1998. *http://www.tpc.org*

Traw, C., and J. Smith. 1991. A High-Performance Host Interface for ATM Networks. *Proc. ACM SIGCOMM Conference* (September):317–325.

Traylor, R., and D. Dunning. 1992. Routing Chip Set for Intel Paragon Parallel Supercomputer. *Proc. Hot Chips '92 Symposium* (August).

Tucker, L. W., and A. Mainwaring. 1994. CMMD: Active Messages on the CM-5. *Parallel Computing* 20(4):481–496.

Tucker, L. W., and G. G. Robertson. 1988. Architecture and Applications of the Connection Machine. *IEEE Computer* 21(8):26–38

Tucker, S. 1986. The IBM 3090 System: An Overview. *IBM Systems Journal* 25(1):4–19.

Tullsen, D. M., and S. J. Eggers. 1993. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. *Proc. 20th Annual Symposium on Computer Architecture* (May):278–288.

Tullsen, D. M., S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. 1996. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. *Proc. 23rd Int'l Symposium on Computer Architecture* (May):191–202.

Tullsen, D. M., S. J. Eggers, and H. M. Levy. 1995. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proc. 20th Annual Symposium on Computer Architecture* (June):278–288.

Turner, J. S. 1988. Design of a Broadcast Packet Switching Network. *IEEE Transactions on Communication* 36(6):734–743.

Valiant, L. G. 1990. A Bridging Model for Parallel Computation. *Communications of the ACM* 33(8): 103–111.

Valois, J. 1995. Lock-Free Linked Lists Using Compare-and-Swap. *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Canada (August):214–222.

Vick, C. R., and J. A. Cornell. 1978. PEPE Architecture—Present and Future. *Proc. AFIPS Conference* 47:981–1002.

von Eicken, T., A. Basu, and V. Buch. 1995. Low-Latency Communication Over ATM Using Active Messages. *IEEE Micro* 15(1):46–53.

von Eicken, T., D. E. Culler, S. C. Goldstein, and K. E. Schauser. 1992. Active Messages. A Mechanism for Integrated Communication and Computation. *Proc. 19th Annual Int'l Symposium on Computer Architecture*, Gold Coast, Australia (May):256–266

Vranesic, Z., M. Stumm, D. Lewis, and R. White. 1991. Hector: A Hierarchically Structured Shared Memory Multiprocessor. *IEEE Computer* 24(1):72–78.

Wall, D. W. 1991. Limits of Instruction-Level Parallelism. *ASPLOS IV* (April):176–188.

Wallach, D. A. 1992. *PHD: A Hierarchical Cache Coherence Protocol*. S.M. thesis, Massachusetts Institute of Technology. Also available as Tech. Report #1389, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Boston, MA (August).

Wang, W.-H., J.-L. Baer and H. M. Levy. 1989. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. *Proc. 16th Annual Int'l Symposium on Computer Architecture* (June):140–148.

Warren, M. S., and J. K. Salmon. 1993. A Parallel Hashed Oct-Tree N-body Algorithm. *Proc. Supercomputing '93*. Washington, DC: IEEE Computer Society, 12–21.

Weaver, D., and T. Germond, eds. 1994. *The SPARC Architecture Manual*. SPARC International. Version 9. Englewood Cliffs, NJ: Prentice Hall.

Weber, W.-D. 1993. *Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors*. Ph.D. diss., Computer Systems Laboratory, Stanford University (January). Also available as Tech. Report #CSL-TR-93-557, Stanford University.

Weber, W.-D., S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke. 1997. The Mercury Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers. *Proc. 24th Int'l Symposium on Computer Architecture* (June):98–107.

Weiss, S. and J. Smith. 1994. *Power and PowerPC*. San Francisco: Morgan Kaufmann.

Widdoes, L., Jr., and S. Correll. 1980. The S-1 Project: Developing High Performance Computers. *Proc. COMPCON* (Spring):282–291.

Wilson, A., Jr. 1987. Hierarchical Cache / Bus Architecture for Shared Memory Multiprocessors. *Proc. 14th Int'l Symposium on Computer Architecture* (June):244–252.

Wolf, M. E., and M. S. Lam. 1991. A Data Locality Optimizing Algorithm. *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (June):30–44.

Wolfe, M. 1989. *Optimizing Supercompilers for Supercomputers*. Cambridge, MA. MIT Press.

Woo, S. C., J. P. Singh, and J. L. Hennessy. 1994. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. *Proc. 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):219–229, San Jose, CA.

Woo, S. C., M. Ohara, E. J. Torrie, J. P. Singh, and A. Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. 22nd Annual Int'l Symposium on Computer Architecture* (June):24–36

Wood, D. A., S. Chandra, B. Falsafi, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, S. S. Mukherjee, S. Palacharla, and S. K. Reinhardt. 1993. Mechanisms for Cooperative Shared Memory. *Proc. 20th Annual Symposium on Computer Architecture* (May):156–167.

Wood, D. A., S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz, and D. A. Patterson. 1986. An In-Cache Address Translation Mechanism. *Proc. 13th Annual Symposium on Computer Architecture* (June):358–365.

Wood, D. A., and M. D. Hill. 1995. Cost-Effective Parallel Computing. *IEEE Computer* 28(2):69–72.

Woodbury, P., A. Wilson, B. Shein, I. Gertner, P.Y. Chen, J. Bartlett, and Z. Aral. 1989. Shared Memory Multiprocessors: The Right Approach to Parallel Processing. *Proc. COMPCON* (Spring):72–80.

Wulf, W., R. Levin, and C. Person. 1975. Overview of the Hydra Operating System Development. *Proc. 5th Symposium on Operating Systems Principles* (November):122–131.

Yamashita, N., T. Kimura, Y. Fujita, Y. Aimoto, T. Manaba, S. Okazaki, K. Nakamura, and M. Yamashina. 1994. A 3.84GIPS Integrated Memory Array Processor LSI with 64 Processing Elements and 2Mb SRAM. *Int'l Solid-State Circuits Conference*, San Francisco (February):260–261.

Zekauskas, M. J., W. A. Sawdon, and B. N. Bershad. 1994. Software Write Detection for a Distributed Shared Memory. *Proc. Operating Systems Design and Implementation Symposium* (November):87–100

Zhang, Z., and J. Torrellas. 1995. Speeding Up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. *Proc. 22nd Annual Symposium on Computer Architecture* (May):188–199.

Zhou, Y., L. Iftode, and K. Li. 1996. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. *Proc. Operating Systems Design and Implementation Symposium* (October).

索　引

索引中的页码为英文原书的页码，与书中边栏的页码一致。

A

absolute performance（绝对性能），202，228～229
measuring（测量，度量），203
wall-clock time（挂钟时间），228～229
abstraction layers（抽象层次），52
for bus-based SMPs（对基于总线的SMP），270（fig.）
parallel architecture（并行体系结构），27（fig.）
access（访问）
data（数据），137～142，254
faults（故障，失效），558
memory（存储器），310，942
order with explicit synchronization（按显式同步的次序），285（fig.）
order without synchronization（无同步的次序），285（fig.）
user-level（用户级），491～496
access control（访问控制），706
through code instrumentation（通过代码插桩），707～708
with decoupled assist（使用分离的辅助部件），707
fine-grained（细粒度），707
through language/compiler support（通过语言/编译器支持的），721～724
page-based（基于页面的），709～721
access time（访问时间）
CRAY T3D（CRAY T3D），510
DRAM（DRAM），831
reducing（减少），831
remote cache（远程高速缓存），663
acquire-based consistency（基于"获取"的同一性），738～739
acquire method（获取方法），335
acquire operation（获取操作），692
incomplete in RC（释放同一性中的"未完成"），871
single writer with consistency at（单写入者的同一性），734～737
Active Messages（主动消息），481～482，490，515
bulk transfers（大批传输），482
echo test（回应测试），522
incoming message notification（抵达消息通告），482
one-way time（单程时间），523
primitives（原语），481
request/response transactions（请求/响应事务），481
active threads（主动线程），898
number of（数目），898
support（支持），907。参见 multithreading; threads
adaptive routing（自适应路由），790，799～801
fully adaptive（完全自适应），800，801
hot spots and（热点），817
nonminimal（非最小的），801
partially adaptive（部分自适应），800，801。参见 routing
address home board（地址宿主板），416
address space（地址空间）
global physical（全局物理的），55
identifiers（ASIDs）（标识符（ASID）），440，441
independent local physical（独立本地物理的），55
private（私用），112
reflective memory（反射存储器），519（fig.）。参见 shared address space
admission control（接纳控制），818
Alewife project（Alewife 项目），70，915
algorithmic optimization（算法优化），219
algorithmic speedup（算法加速比），220
measuring（测量，度量），254
for parallel applications（对并行应用），256（fig.）　参见 speedup
algorithms（算法）
Barnes-Hut（Barnes-Hut），79
barrier（栅障），356～357，542～547
Dekker's（Dekker 的），688
Gaussian elimination（高斯消去法），194
lock（锁），337，342～343，346～348，538～541
routing（路由），752～753
scheduling（调度），808
sequential（串行的，顺序的），161，166～169，175，179～180，389
software queuing lock（软件排队锁），541（fig.）
synchronization, performance（同步，性能），649～651
waiting（等待），335
west-first（西向优先），797～798，798～799

Alliant FX-8 machine（Alliant FX-8 机器），435
all-to-all personalized communication（全体至全体个性化通信），547
example（示例），547
savings（节省），588
AMBER（Assisted Model Building through Energy Refinement）（AMBER），8
Amdahl's Law（Amdahl 定律），84，841
application of（……的应用），88
rewording（重述），87
application miss rate（应用扑空率），319～324
breakdown for Barnes-Hut，LU，Radiosity（相对于 Barnes-Hut、LU、Radiosity 的分解），320（fig.）
breakdown for Multiprog（相对于多道程序的分解），323（fig.）
breakdown for Ocean，Radix，Raytrace（相对于 Ocean，Radix，Raytrace 的分解），321（fig.）
breakdown for 64-KB caches（相对于 64K 字节高速缓存的分解），325（fig.）
reducing（降低），325。参见 cache miss rate
applications（应用程序）
performance（性能），429～433
performance improvement with parallelism（利用并行性的性能改进），6
scaling（伸缩，扩放），431～433
speedups（加速比），430～431。参见 specific applications
application trends（应用趋势），6～12
commercial（商业的），9～12
scientific/engineering（科学的/工程的），6～9
architectural trends（系统结构的趋势），14～21
microprocessor design（微处理器设计），15～19
system design（系统设计），19～21
array-based locks（基于数组的锁），347～348
acquire method（获取方法），347
drawback（缺点），348
latency（时延），347
performance（性能），350，651
scalable problems（可扩展的问题），539。参见 locks
arrays（数组，阵列）
aligning（对齐，对准），365～366
alternative organizations（其他组织结构），365（fig.）
4D vs.2D performance impact（四维与二维对性能的影响），363（fig.）
organization determination（组织决策），364～365
padding（填充），364
smaller than 32KB（小于 32K 字节），424
artifactual communication（附加通信），137，139～142，653
avoiding（避免），257
in extended memory hierarchy（在扩展的存储器层次结构中），139～140
problem size and（问题规模），223
Raytracing application（光线跟踪应用），176
reducing（降低），142～150
replication and（复制），140～142
shared address space（共享地址空间），142 146
sources（源），139～140
ASCI Red machine（ASCI Red 机），502 503 771
assignment（分配），83 88～89
Barnes-Hut application（Barnes-Hut 应用），169～170
block（块），98
changes（改变），107～108
compile-time（编译时），88
cyclic（循环的），98 108
Data Mining application（数据挖掘应用），83 180～181
dynamic（动态的），88，126～129
equation solver kernel（方程求解器程序内核），98～99
load-balanced（负载平衡的），127
Ocean application（Ocean 应用），161～163
perfomance goals（性能目标），88
Raytrace application（Raytrace 应用），175～176
static（静态的），88，99，126～129。参见 parallelization process
assist occupancy（辅助部件占用度）
contention impact on（争用对……的影响），647
effects on protocol trade-offs（对协议权衡的作用），648
impact on cache coherence protocols（对高速缓存一致性协议的影响），647（fig.）
latency tolerance and（时延容忍），845
asynchronous links（异步链路），765
asynchronous message passing（异步消息传递）
protocol（协议），479（fig.）
storage（存储），479。参见 message passing
ATM（asynchronous transfer mode）（ATM（异步传输模式）），646
flow control（流控），813
standard（标准），514
switches（交换机），514
atomic bus（原子性总线），281
lock-down（锁定），391
single-level caches with（单级高速缓存），380～393
atomic primitive implementation（原子性原语实现），391～393，651～652

atomic read-modify-write operation（原子性读－改－写操作），334
attraction memories（吸引存储器），701
automatic update mechanism（自动更新机制），718，719（fig.）
availability clusters（可用性机群），514

B

backoff（回退）
dynamic（动态），595
exponential（指数的），649，651
between failed operations（失败操作之间的……），393
instructions（指令），920
test & set locks with（带回退的 test-set 锁），342
values（数值），920
back pressure（反向压力），483，816
back-to-back latency（背靠背时延），619
backward pointer（向后指针），569，624
bandwidth（带宽），59
aggregate（聚合），761，762
bisection（对分，剖分），761
block data transfer and（块数据传输），241，242，857
broadcast（广播），459
bus-based organization and（基于总线的组织结构），34
channel（信道），752
communication processor（CP）（通信处理器（CP）），502
across computer system（跨越计算机系统），951（fig.）
data transfer operation（数据传输操作），60
effective（有效的），756
HAL SI（HAL SI），644
hierarchical directory scheme（层次目录方案），566
individual（个别的，个体的），761
latency vs.（时延与……的比较），59，787
link（链路），760 939
link limitations（链路限制），668
matching（匹配），807
message-passing（消息传递），529
message size vs（消息尺寸与……比较），531（fig.）
microprocessor bus（微处理器总线），19，21
network（网络），761～764，846
node-to-network（结点到网络），846
NUMA-Q（NUMA-Q 机），641
number of ports and（端口数目），462
offered（提供的），762
on rings（环上的），443，444
out-of-cache memory（流出高速缓存的），939
point-to-point（点对点），151，845～846
per-processor requirements（每个处理器的需求），312（fig.）
protocol design and（协议设计），307～311
scaling（伸缩，扩放），445～446，457～459
SGI Origin2000，peak hardware（SGI Origin2000，峰值硬件），618
switch（交换开关），939
total（总的），762
uniprocessor design（单处理器设计），939
vector memory transfer（向量存储器传输），46
Banyan network（Banyan 网络），803～804
Barnes-Hut algorithm（Barnes-Hut 算法），79
Barnes-Hut application（Barnes-Hut 应用），76～77，78～79，166～174
assignment（分配），169～170
Barnes-Hut algorithm and（Barnes-Hut 算法），79
barrier（栅障），106
cache-to-cache sharing（高速缓存到高速缓存的共享），588
computation flow（fig.）（计算流（图示）），168
costzones（成本区域），170，171（fig.）
decomposition（分解），169～170
error and（错误），265
execution time（执行时间），174（fig.），214
force calculation（作用力计算），166，169
invalidation pattern（作废模式），575（fig.）
invalidations（作废），577
latency in（时延），913
mapping（映射），173
miss rates（扑空率），320（fig.）
naming（命名），184
n-body problem（n 体问题），79，80（fig.）
ORB（ORB），170，171（fig.）
orchestration（协调），170～173
particles，number of（粒子，数目），265
partitioning（分区），169，171（fig.）
scaling（伸缩，扩放），432
scaling of speedups on SGI Origin2000（相对于 SGI Origin2000 加速比的比例），622（fig.）
sequential algorithm（串行算法），166～169
spatial locality（空间局部性），170～172，322
speedup（加速比），431
speedup on SGI Origin2000（SGI Origin2000 的加速比），621（fig.）
star force computation（星体引力计算），79
summary（总结），173～174

synchronization（同步），172～173
tasks（任务），82
temporal locality（时间局部性），172
three-dimensional space calculation（三维空间计算），167
traffic vs.local cache（流量与本地高速缓存的关系），582（fig.）
traffic vs.number of processors（流量与处理器数目的关系），580（fig.）
tree traversals in（树遍历），893
two-dimensional particle distribution（二维的粒子分布），168（fig.）
working sets（工作集），172。参见 case studies
barrier algorithms（栅障算法），356～357，542～547
all-to-all personalized communication（全部到全部个性化通信），547
global synchronization（全局同步），356～357
parallel prefix（并行前缀），546～547
scalable multiprocessor synchronization（可扩展的多处理器同步），542～547
software combining trees（软件合并树），542～543
tree barriers with local spinning（带有本地踏步等待的树栅障），543～545
barriers（栅障）
centralized（集中的……），353～356
fairness（公正性），356
hardware（硬件），358
latency（时延），356
memory（MB）（存储器（MB）），693
performance goals（性能目标），356
performance on SGI Challenge（SGI Challenge 的性能），357（fig.）
scalability（可扩展性），356
static binary tree（静态二叉树），544
storage cost（存储成本），356
traffic（流量），356
write memory（WMB）（写存储器（WMB）），693
barrier synchronization（栅障同步），252，353～358
barrier algorithms（栅障算法），356～357
centralized barrier with sense removal（带有感应逆转的集中式栅障），354～356
centralized software barrier（集中式软件栅障），353～354
hardware barrier（硬件栅障），358
hardware primitives（硬件原语），357～358
performance（性能），356。参见 synchronization
baseline communication structure（基准通信结构），834
BBN Butterfly maching（BBN Butterfly 机），760
benchmarks BT（基准测试程序 BT），532～537
kernel（内核），967
LAPACK（LAPACK），963
LINPACK（LINPACK），13 209
low-level（底层），967
LU（LU），532，533（fig.），537～538
microbenchmarks（微基准测试程序），215～216，967
NAS（NAS），966～967
ongoing efforts（正在进行的努力），968
PARKBENCH（PARKBENCH），967～968
Perfect Club（perfect Club 基准测试集），968
Scalapack（Scalapack 基准测试集），963
SPEC（SPEC 基准测试集），13，199
SPLASH（SPLASH），307，965～966
TPC（TPC），642，643，963～965
workloads（工作负载），201
Benes network（Benes 网络），776，777（fig.）
binay trees（二叉树），772
illustrated（解释），773（fig.）
space partitioning（BSP）（空间划分），250
binding prefetch（绑定预取），880
bisection bandwidth（对分宽带），761
bisection scaling rule（对分扩放规则），788
bit-level parallelism（位级并行性），15
bit-serial design（位串行设计），46
Blizzard-S，745
block data transfer（块数据传输），187-188，837
advantages（优点），188 838 857
amortized per-message overhead（每条消息分摊的开销），857
bandwidth and（带宽），241～242
bandwidth waste（带宽浪费），857
communication assist（通信辅助部件），854
contention（竞争），858
disadvantages（弱点），858
effect（效果），839（fig.）
explicit（显式的），853
facility（设施），238
in FFT program（在 FFT 算法中），861
interaction with cache-coherent shared address space（和高速缓存一致的共享地址空间的交互作用），855～856
mechanisms（机制），853～854
message passing（消息传递），848
in near-neighbor equation solver（在近邻式方程求解器程序里），859（fig.）
in Ocean application（在 Ocean 应用中），861（fig.）
overhead（开销），242

overhead per transfer（每次传输的开销），858
path（通路），854（fig.）
performance benefits（性能益处），856～863
pipelined transfer（流水传输），857
pipeline stage time（流水线的级时间），862
policy issues（策略问题），854～856
read-write cache-coherent communication vs.（高速缓存一致的读写通信与……比较），862～863
relative performance improvement with（相对性能改善），860（fig.）
at row-oriented partition boundaries（在面向行的分区边界），860
shared address space（共享地址空间），853～863
synchronization bundling（打包同步），857
system buffering/copying（系统缓冲/拷贝），853～854
techniques（技术），853～854
trade-offs（权衡，折中），854～856，859
transferred data placement and（传输数据的放置），856
transferred data replication（传输数据的复制），857。参见 latency tolerance
blocked multithreading（阻塞的多线程），898～902
context switch（现场切换），916～917
control（控制），916～917
disadvantages（劣势），912
EPC（EPC），915～916
implementation issues（和实现有关的问题），914～917
interleaved scheme vs.（交错方案与……比较），911
latency tolerance（时延容忍），903（fig.）
PC bus（PC 总线），919（fig.）
processor utilization vs. number of threads（处理器利用率与线程数的关系），899（fig.）
requirements（需求），914
speedup（加速比），912（fig.）
state replication（状态复制），914～915。参见 multithreading
blocking（阻塞），144，195
asynchronous SEND operation（异步发送操作），115
benefits（好处），245
busy-waiting trade-offs（忙等待的权衡），335
caches（高速缓存），414　877
data reuse and（数据重用），246
to exploit temporal locality（开发时间局部性），144（fig）
head-of-line（行头），805～806
reads（读），864　877
block prefetch（块预取），880
blocks（块）
B-by-B（一块一块地），246
busy state（忙状态），590
contiguous data（连续数据），247
directory information（目录信息），562
directory state（目录状态），558
directory tree（目录树），566
head pointer（头指针），569
interleaving（交叉、交错），246
local（局部），560
remote（远程），560
size of（尺寸），246
block transfer engine（BLT）（块传送引擎），819
board-level integration（板级集成），465～466
bounded buffer problem（有限缓冲区问题），117
branch target buffer（BTB）（分枝目标缓冲区），918
broadcast（广播）
bandwidth（带宽），459
bit（位），655
media，hierarchy of（介质，层次结构），555
scheme（方案），655
snooping mechanism（侦听机制），559
version（版本），119
BSP model（BSP 模型），191
BT benchmark（BT 基准测试程序），532～537
application performance（应用性能），534（fig.）
communication characteristics（通信特征），535（fig.）
message profile over time（随时间变化的消息概况），536（fig.）
speedups（加速比），533
temporal communication behavior（通信的时态行为），536。参见 benchmarks
buffer deadlock（缓冲区死锁），412
causes（原因），412
solutions（解决方案），412，594。参见 deadlock
buffering（缓冲）
at home（在宿主处），590～591
input（输入），804～806
memory operation（存储器操作），863
output（输出），806
at requestors（在请求者方），591
switch（交换开关），759　768
virtual channel（虚拟通道），807～808
buffers（缓冲区）
branch target（BTB）（分枝目标），918
channel（信道），804～808
FIFO（先入先出），409　616　804

flit（流控单元），816
input（输入），472 482～483
intervention request（IRBs）（干预请求（IRBs）），615
lookahead（先行），870
output（输出），472
prefetch（预取），878
read request（RRBs）（读请求（RRBs）），615
reorder（重排序），414，870，876
stream（流），882
TLB（翻译后援缓冲器），67，223
write（写），413，865，867，874
write-back（回写），385
write request（WRBs）（写请求），615
bus-based systems（基于总线的系统）
hierarchical（层次式的），677
organization（组织结构），32～34
scaling data in（数据规模扩放），445
snoop bandwidth in（侦听带宽），445～446
buses（总线），555
PCI（PCI），635，636，637
sharing（共享），588，589
snooping（侦听），277～283，589
split-transaction（事务拆分），398～415
SysAD（SysAD），609，612，613
bus order（总线操作序），281～282
program order and（程序操作序），282
writes/reads serialized in（串行化的写/读），291
bus traffic（总线流量）
block size impact on（块尺寸的影响），324～327
cache block size for Multiprog（Multiprog中高速缓存块的尺寸），327（fig.）
cache block size with 1-MB caches（1MB高速缓存的块尺寸），326（fig.）
cache block size with 64-KB caches（64KB高速缓存的块尺寸），328
test-and-test&set locks and（test-and-test&set锁），343
bus transactions（总线事务）
arbitration medium（仲裁介质），470
completion of（完成），470
destination（目的地），470
information format（信息格式），470
input registers（输入寄存器），470
ordering properties（定序性质），473
output registers（输出寄存器），470
bus upgrades（总线升级），295
busy-overhead（忙碌开销），158
busy states（忙碌状态），590
blocks in（块的），590
SGI Origin2000（SGI Origin2000），597，600
busy time（忙碌时间），897
busy-useful（有用忙），157，210
busy-waiting（忙等待），106～107，335～336
cost（成本），336
trade-offs（折中），335
butterflies（蝶式），774～777
back-to-back（背靠背），776
d-dimensional indirect（d维间接），774
d-dimensional k-ary（d维k元），775
fault tolerance（容错），775
forward-going（前向推进），777
illustrated（举例），775（fig.）
links（链路），775～776
routing（路由），778
2 x 2，774
with wormhole routing（具有蛀洞路由的），808。参见network topologies

C

cache associativity（高速缓存的关联度），241
cache-based directory scheme（基于高速缓存的目录方案），566，568～570
latency reduction（时延减少），587（fig.）
NUMA-Q（NUMA-Q），622～645。参见directory protocols；directory schemes
cache blocks（高速缓存块）
address tag（地址标记），329
dirty（脏），406，409
exclusive copy of（独占副本），292
false sharing of（假共享），711
granularity（粒度），706
invalid，writing into（无效的，写入），294，296
large，alleviating drawbacks of（大的，减少弱点），328～329
lifetime of（寿命），316
modified state（已修改状态），292
multiword（多字），319
old invalidating（旧的作废），389～390
overhead（开销），242
owner（拥有者），292
promoted（提升的），294
replacement（替换），296
shared，writing into（共享的，写入），295，296

sharing state（共享状态），402
states（状态），279
state transition diagram（状态转换图），279～280
state transitions（状态转换），279，367
write-back protocols and（回写协议），296
cache block size（高速缓存块尺寸），241
bus traffic and（总线流量），324～327
effects on proceeding past writes（越过写操作的效果），869（fig.）
miss rate and（扑空率），318～319
trade-offs（折中），313～329
cache coherence（高速缓存一致性），272，273～283
with bus snooping（总线侦听），277～283
directory-based（基于目录的），553～677
example problem（实例问题），274（fig.）
extending（扩展），433～436
as hardware design issue（硬件设计问题），275
hardware support for（硬件支持），669
home memory and（宿主存储器），590
I/O buses and（输入输出总线），515
memory bus（存储器总线），508
NUMA-Q（NUMA-Q），624～632
problem（问题），273～277
processor overhead and（处理器开销），645
protocol（协议），305
scalable（可扩展的），558～559
SGI Origin 2000（SGI Origin 2000），597～604
snoop-based, on rings（基于侦听的），441～444
snooping（侦听），272
cache-coherent multiprocessors（高速缓存一致的多处理机）
block data transfer vs.（与成块数据传输比较），862～863
bus-based（基于总线的），441
coherence misses（一致扑空），314～315
hardwired assist（硬接线的辅助部件），706
latency tolerance（时延包容），926～927
snoop-based（基于侦听的），589
cache-coherent, nonuniform memory access（CC-NUMA）（高速缓存一致的，非均匀的存储器访问（CC-NUMA））
architectures（体系结构），558，725
flexibility（灵活性），730～732
hardware cost（硬件成本），705～706
main memory management（主存管理），653
performance trade-offs（性能折中），703（fig.）
physical address space limitations（物理地址空间限制），732
two-processor system（双处理器系统），674
cache-coherent shared address space（高速缓存一致的共享地址空间），879～891
hardware-controlled prefetching（硬件控制的预取），881～883
hardware-controlled vs. software-controlled prefetching（硬件控制与软件控制预取的比较），888～890
interactions with multiprocessor coherence protocol（和多处理器一致性协议的交互作用），887～888
prefetching concepts（预取原理），880～881
prefetching with single processor（单处理器的预取），884～887
sender-initiated preconmunication（发送方发起的预通信），890～891
software-controlled prefetching（软件控制的预取），884、参见 shared address space
cache conflicts（缓存冲突），224
cache controllers（高速缓存控制器），62，389
in bus-based system（基于总线的系统），652
design（设计），381～382
hardwired assist（硬接线的辅助部件），706
implementation requirements（实现要求），663
lowest-level（最低层），397
request table（请求表），402
snooping（侦听），277
uniprocessor（单处理机），381～382
cache misses（高速缓存扑空），17，140～141
application structure and（应用结构），319～324
capacity（容量），141，223，313
classification of（分类），316～318
coherence（一致性），314～315
cold-start（冷启动），140
compulsory（强制的），313
conflict（冲突），141，164，314，362～363
"context available" signal and（"现场可用"信号），920
cost of（……的成本），324
data transfer（数据传送），58
decomposition of（分解），318
detection mechanism（检测机制），558
false-sharing（假共享），315～316
latency of（时延），17，324
path of（路径），404～406
process migration and（进程迁移），324
read（读），662，744
reducing（减少），141
time（时间），943，944
TLB（转换检测缓冲器），223，429

true-sharing（真共享），315，316
upgrades（升级），318
write（写），666
cache miss rate（高速缓存扑空率），943
Barnes-Hut application（Barnes-Hut 应用），320（fig.）
block size and（块尺寸），318～319
hybrid protocols（混合协议），332（fig.）
invalidation-based protocols（基于作废的协议），332（fig.）
LU application（LU 应用），320（fig.）
Multiprog application（Multiprog 应用），323（fig.）
Ocean application（Ocean 应用），321（fig.）
Radiosity application（Radiosity 应用），320（fig.）
Radix application（Radix 应用），321（fig.）
Raytrace application（Raytrace 应用），321（fig.）
update-based protocols（基于更新的协议），332（fig.）
cache-only memory architecture（COMA）（唯高速缓存的存储器体系结构（COMA）），680，701～705
access latency（访问时延），702
attraction memories（吸引存储器），701
communication latencies（通讯时延），729
design options（设计选项），703～705
explicit migration（显式迁移），730
fine-grained replication（细粒度复制），729
flat directory scheme（扁平目录方案），703～704
hardware-intensity（硬件密度），726
hardware/software trade-offs（硬件/软件折中），701～702
hardware support（硬件支持），702，739
hierarchical directory scheme（层次式目录方案），703～705
high-capacity miss rates and（高的容量性扑空率），702
memory blocks（存储器块），701
performance trade-offs（性能折中），702～703
read operation path（读操作的途径），705
Simple（简单的），681，726～728
workloads（工作负载），729～730
caches（高速缓存），17，46
allocation granularity（分配粒度），277
blocking（成块），414，877
block size（块尺寸），226
commercial use of（商业用途），67
CRAY T3E（CRAY T3E），36
direct-mapped（直接映射），241，247，362（fig.）
filter bandwidth（过滤器带宽），68
hardware（硬件），185
infinite（无限的），572
L_i（L_i），394～395，396
local（本地的），139
lockup-free（免锁的），414，922～926，930
LRU（最近最少使用算法），259
mapping conflicts（映射冲突），362（fig.）
microprocessor transistors devoted to，（用于……的微处理器晶体管），948（fig.）
multilevel（多级），393～398
organization of（组织结构），313
per-processor（每处理器的），272
remote（远程），623～624，660，662
replacement（替换），185
scaling（伸缩，扩放），236～237
shared（共享的），434～437
single-level（单级的），281，380～393
snooping（侦听），383，386（fig.），398（fig.）
software，fixed-size（软件方式的，固定大小的），186
SRAM（SRAM），304
tertiary（三元的），700～701
uniprocessor（单处理器），381
virtually indexed（虚拟索引的），437～439
write-back（回写），274，283，291
write-no-allocate（写～无分配），281（fig.）
write-through（直写），278，279，281（fig.），291。参见 cache controllers；cache misses；cache miss rate
cache sizes（高速缓存尺寸），142（fig.），236
choosing（选取），239～240
miss rate vs.（与扑空率关系），224（fig.）
for scaled-down problem（问题规模缩小），237（fig.）
cache tags（高速缓存标记）
design of（设计），381～382
dual（双重的），397
looking up（查找），401
nonmatching（不匹配），396
cache-to-cache handshake（高速缓存到高速缓存的握手），383
cache-to-cache sharing（高速缓存到高速缓存的共享），300，589
Barnes-Hut application（Barnes-Hut 应用），589
likelihood of（……的可能性），597
capacity limitations（容量限制），700～705
COMA（COMA），701～705
tertiary cache（三级高速缓存），700～701
capacity misses（容量型扑空），141，223，314
block size and（块尺寸），318
COMA machines and（COMA 机），702
occurrence of（……的发生），313
Ocean application（Ocean 应用），324

Raytrace application（Raytrace应用），324
reducing（减少），314
spatial locality in（空间局部性），324。参见 cache misses
case studies（案例研究）
Barnes-Hut application（Barnes-Hut应用），76～77，78～79
CM-5（CM-5），493～494
CRAY T3D（CRAY T3D），508～511，818
CRAY T3E（CRAY T3E），512～513
Data Mining application（数据挖掘应用），77，80～81
IBM SP-1/SP-2（IBM SP-1/SP-2），820～822
Intel Paragon（Intel的Paragon机），499～503
LU（LU），244～248
Meiko CS-2（Meiko CS-2），503～506
Multiprog（多道程序），244，252
Myricom network（Myricom网络），826～827
Myrinet SBUS Lanai（Myrinet SBUS Lanai），516～518
nCUBE/2（nCUBE/2），488～490
network design（网络设计），818～827
NUMA-Q（NUMA-Q），622～645
Ocean application（Ocean应用），76，77～78
parallel application（并行应用），76～81，160～182
PCI Memory Channel（PCI存储器通道），518～521
Radiosity application（Radiosity应用），244 249～252
Radix application（Radix应用），244，248～249，267
Raytrace application（Raytrace应用），77，79～80
SCI（SCI），822～825
SGI Challenge（SGI Challenge），415，417～424
SGI Origin2000（SGI Origin2000），596～622，825～826
Sun Enterprise 6000（Sun Enterprise 6000），415～416，424～429
workload（工作负载），244～253
CDC（CDC）
CDC（CDC 7600），67
STAR-100（STAR-100），67，953
centralized barriers（集中式栅障），353～356
with sense reversal（原书误为 removal）（带有感应逆转的），354～356
sense reversal method（感应逆转方法），355
shared counter（共享计数器），353。参见 barriers
centralized directory schemes（集中型目录方案），564
centralized queue（集中式队列），128
channel buffers（信道缓冲区），804～808
input（输入），804～806
output（输出），806
shared pool（共享池），807
virtual channel（虚拟信道），807～808。参见 buffers
channels（信道），751，752
bandwidth（带宽），752
dependence graph（依赖图），793，794（fig.）
occupancy（占用），755
ordering（排序），794（fig.）
packet occupation（数据包占用），756
resource allocation（资源分配），792
as shared resource（作为共享资源），791
time（时间），460
virtual（虚拟的），613，795～796
width（宽度），784，787
chip-level integration（芯片级集成），463～464
circuit switching（电路交换），756～757
CISC microprocessors（CISC微处理器），19
clocking（定时）
asynchronous（异步），765
synchronous（同步），765
closed systems（封闭系统），760
clusters（集群，簇），12
availability（可用性），514
hardware primitives（硬件原语），515
as parallel machines（作为并行机），514
potential of（……的潜能），514
CM*（CM*），68
CM-5（CM-5），70，465～466，953
C.mmp（C.mmp），68
CM-1（CM-1），69，952
CM-2（CM-2），69，952
case study（案例研究），493～494
communication assis（通信辅助部件），493
communication latency（通信时延），525
"control" network（"控制"网络），542
data networks（数据网络），494
latency（时延），493
machine organization（机器组织结构），466（fig.）
message interleaving（消息交叉），495
network（网络），465
receive overhead（接收开销），525
send overhead（发送开销），523～524
shared address read（共享地址读），527
user-level network support（用户级网络支持），493～494
coarse-grained tasks（粗粒度任务），83
coarse vector overflow method（粗糙向量溢出方法），656，657（fig.）
coarse vector representation（粗糙向量表示），609，656（fig.）
code instrumentation（代码修正），707～708

protocol cost（协议成本），708
run-time cost（运行时成本），707
coherence（一致性），272
cache（高速缓存），272，273～283，433～446，553～677
compiler-based（基于编译器的），723
global（全局的），855～856
granularity of（……的粒度），146，724，725（fig.）
hierarchical（层次的），659～669
local（本地的），855
management（管理），724
memory system（存储器系统），276
MSI protocol and（MSI协议），297
multilevel cache hierarchies and（多级高速缓存的层次），393
object-based（基于目标的），721～723
properties（性质），275，277
serialization to location for（对单元访问的串行化），589～591，604～607，632～633
software SVM（软件 SVM），721
TLB（转换检测缓冲器），439～441
for virtually indexed caches（虚拟索引的高速缓存），437～439
coherence controllers（一致性控制器），562
contention at（竞争），654
NUMA-Q（NUMA-Q），731
Stanford FLASH（Stanford FLASH），731
coherence misses（一致性扑空），314～315
false-sharing（假共享），315
improving locality on（改善……的局部性），328
occurrence of（……的发生），314～315
true-sharing（真共享），315。参见 cache misses
cold misses（冷启动扑空），140，313
block size and（块尺寸），318
occurrence of（……的发生），313
reducing（减少），314
spatial locality in（……的空间局部性），324。参见 cache misses
commercial computing（商业计算），9～12
communication（通信），25
all-to-all personalized（全部对全部个性化的），547
architecture implications and（体系结构的影响），135
artifactual（人为的，附加的），137，139～142，653
available，operations（可用的，操作），26～27
baseline，structure（基准，结构），834
collective（集体的），55
with communication（通信），837
computation with（对……的计算），837
cost（成本），62～63，122，151，156
endpoints（端点），518
expense of（……的花费），138
explicit（显式的），116，187，189
framework for understanding（理解框架），26
frequency of（……的频率），62
granularity（粒度），186～187，724，725（fig.）
hardware support for（……的硬件支持），515
impact of（……的影响），132
implicit（隐式的），189
inherent（固有的），140
interprocess（进程间），58，131
Meiko CS-2（Meiko CS-2），505（fig.）
message-passing（消息传递），38，111
microbenchmarks（微基准测试程序），216
misses（扑空），141
in multimemory system（在多存储器系统中），137～142
network as pipeline for（网络作为……的流水线），836（fig.）
overhead（开销），186～187
overhead，reducing（开销减少），487
overlapping（重叠），63，155～156
performance（性能），755～764
pipeline（流水线），836
between processes（在进程之间），59
processes timelines（进程时线），835（fig.）
queues（队列），498
read-write（读写），852
receiver-initiated（接收方发起的），833
reducing（减少），123，131～135
redundant，data（冗余数据），140
replication and（复制），58～59
sender-initiated（发送方发起的），833，879
shared address space（共享地址空间），473，474（fig.）
structure（结构），137，848，852
structuring，to reduce cost（为减少成本的结构），150～156
SVM（SVM），712（fig.）
true（真），58
communication abstraction（通信抽象），26，52，56
"contract"（"契约"），53
implementing（实现），488
interface（接口），53
in large-scale machines（在大规模机器中），469
message passing（消息传递），39（fig.），488
role of（……的角色），53

communication architecture（通信体系结构），25～28
design space（设计空间），485～486
facets（侧面），25
communication assists（通信辅助部件），50～51，156
blind physical DMA（盲物理 DMA），487（fig.）
block data transfer（块数据传输），854
CM-5（CM-5），493
CRAY T3D（CRAY T3D），511
design（设计），51
HAL S1 multiprocessor（HAL S1 多处理器），644
Memory Channel（存储器通道），518
Myricom network（Myricom 网络），826
network transaction formatting（网络事务格式化），485
in protocol processing（在协议处理中），645
shared memory design and（共享存储器设计），51
shared physical address space（共享物理地址空间），506，507
for user-level handlers（用户级的处理例程），495（fig.）
for user-level network ports（用户级的网络端口），492（fig.）
communication latency（通信时延），522～523，833
comparison（比较），525～526
components，hiding（构成部份，隐藏），842
message breakdown（消息分解），523（fig.）
speedup（加速比），842。参见 latency；latency tolerance
communication processor（CP）（通信处理器），455，496
bandwidth（带宽），502
computer processor and（计算机处理器），498
concurrency intensive function（并发性密集的函数），499
cooperation（合作），496
dedicated（专用的），497
Meiko CS-2（Meiko CS-2），503
message delivery（消息递交），499
polling（轮询），497
synchronization operations（同步操作），496
communication-to-computation ratio（通信与计算比），132
computing（计算），132
control of（……的控制），132
data set size and（数据集大小和……），258
growth rates of（……的增长率），258～259
MC scaling（MC 伸缩），433
measuring（测量），257
n/p ratio and（n/p 比率），226
processor count vs.（与处理器数的关系），258
scaling models（扩放模型），212
TC scaling（TC 扩放），433
in three dimensions（三维），132
in two dimensions（二维），132
compare&swap instruction（比较并交换指令），334，340，649
atomic（原子的），540
implementing（实现），392
queuing lock（排队锁），540
competing operations（竞争操作），695
compiler-based coherence（基于编译器的一致性），723
compilers（编译器），959
advanced optimizations（高级优化），289～290
architecture evolution and（体系结构演变），75
parallelizing（并行），961
technology（技术），959
uniprocessor（单处理器），289
complete applications（完整的应用），217～218
compulsory traffic（强制性的流量），140
Computational RAM（计算型 RAM），951
computer architecture（计算机体系结构）
change and（变化），4
generations（发展阶段），15
history of（……的历史），945（fig.）
computer systems（计算机系统）
bandwidths across（跨……的带宽），951（fig.）
500 fastest（500 个最快的），24（fig.）
concurrency（并发性），
degree of（……的程度），98
exposing（暴露），85
extra（额外的），155
finding with program structure（通过程序结构发现），93～94
Gaussian elimination（高斯消去法），194～195
in Gauss-Seidel equation solver computation（在高斯－赛德尔方程求解器计算中），95（fig.）
identifying（识别），124～126
limited（有限的），84，86
loop nests and（循环嵌套），93
managing（管理），126～129
reducing（减少），99，101
scaling models（扩放模型），212
workload（工作负载），254～257
concurrency profiles（并发性态），85
area under curve（曲线下的面积），86
distributed time，discrete-event logic simulator（分布的时间，离散事件逻辑模拟器），87
x-axis points（x－轴点），87
conflicting operations（冲突操作），695

conflict misses（冲突型扑空），141，314，876
across grids（跨网格的），164
increase of（……的增长），319
occurrence of（……的发生），314
predicting（预测），885
reducing（减少），314，参见 cache misses
conflict order（冲突次序），696
constraints（约束）
resource-oriented（面向资源的），207
user-oriented（面向用户的），207
contention（竞争）
assist occupancy and（辅助部件占用度），647
block data transfer and（块数据传输），858
cause of（……的原因），154，654
at coherence controllers（在一致性控制器处），654
endpoint（端点），154
at endpoints（在端点处），761
latency and（时延），759，786，787（fig.）
lock algorithms and（锁算法），675
network（网络），154
orchestration strategies and（协调策略），654
prefetching and（预取），895
preserving（保留），235
problem（问题），153～154
reducing（减少），153～154
resource（资源），62
context status word（CSW）（上下文状态字），915
context switches（上下文切换），897，901
in blocked scheme（在成块方案中），916～917
overhead（开销），901。参见 switches
Convex Exemplar（Convex Exemplar），570，700
crossbar（交叉开关），940
SCI specification（SCI 规范），822
correctness（正确性），377
classes（类），589
deadlock（死锁），594～595
livelock（活锁），595
NUMA-Q（NUMA-Q），632～634
requirements（需求），378～380
serialization across locations for sequential consistency（为了顺序同一性的跨单元串行化），592～593
serialization to location of coherence（对一致性单元的串行化），589～591
SGI Origin2000（SGI Origin2000），604～609
starvation（挨饿），595～596
cost（成本），59
amortizing（分摊），588
busy-waiting（忙等待），336
cache miss（高速缓存扑空），324
communication（通信），62～63，122，151，156
data access（数据存取），138
delay（延迟），152
design（设计），188，679
fixed（固定的），461
hardware（硬件），188，705～724
implementation（实现），679
message（消息），151
network（网络），782
parallelism（并行性），537
performance trade-off（性能折中），2，4，15
pipelining（流水操作），900
reducing over integrated/specialized assist（通过集成的或专门的辅助部件来减少），708
reducing through communicationstructure（通过通信结构来减少），150～156
scaling（伸缩，扩放），461～462
start-up（启动），60
switch（交换开关），768，898，900
unready thread（未就绪的线程），909～910
costup（成本增加），462
costzones（成本区域），170，171（fig.）
CRAY（CRAY）
C90 vector processor（C90 向量处理机），8
CRAY-1（CRAY-1），45，67，953
CS6400（CS6400），410，445
SCX I/O network（SCX I/O 网络），822
3/SSS（3/SSS），952
vector supercomputers（向量超级计算机），8，22
Xmp（Xmp），337
CRAY T3D（CRAY T3D），8，70，186，508～511
access time（访问时间），510
Alpha architecture（Alpha 体系结构），509
Alpha 21064（Alpha 21064），509～510，511
annex registers（相联寄存器），510
BT benchmark（BT 基准程序），533
bulk transfer engine（成批传送引擎），511
communication assist（通信辅助部件），511
design（设计），508～509
DTB Annex（DTB Annex），510
flow control（流控），820，942
hardware support for barriers（硬件对栅障的支持），542
links（链路），819，820

LU benchmark（LU 基准测试程序），532
machine organization（机器组织结构），509（fig.）
network，case study（网络，案例研究），818～820
nonblocking stores（无阻塞的存贮），511
nonblocking writes（无阻塞的写），513
packet buffers（数据包缓冲区），820
packet formats（数据包格式），819（fig.）
prefetch buffer（预取缓冲器），878
prefetch instruction（预取指令），511
routing distance（路由距离），820
scaling（扩放，伸缩），509
shared address read（共享地址读），528
synchronous link design（同步链路设计），765
three-dimensional bidirectional torus（三维双向花环），818
user-level messaging support（用户级消息支持），526
virtual-to-physical translation（虚实转换），510
write buffer（写缓冲器），511
CRAY T3E（CRAY T3E），36（fig.），186，512～513
Alpha 21164 processor（Alpha 21164 处理器），512
asynchronous link design（异步链路设计），766
block transfer engine utility（块传送引擎设施），512
design（设计），512
E-registers（E～寄存器组），512
improvements（改善），512～513
nonblocking writes（无阻塞写），513
prefetch buffer（预取缓冲器），878
prefetch queue utility（预取队列设施），512
critical section（临界区），105
crossbar（交叉开关），803～804
Convex Exemplar（Convex Exemplar），940
implementations（实现），803（fig.）
memory as（存储器），802
size increase（尺寸增长），808
tristate drivers and（三态驱动器），809
crossbar switch（交叉开关交换机），30，31（fig.），34
cube-connected cycles（立方体连接环），811
cut-through routing（直通路由），757（fig.），782
deadlock（死锁），792
distance（距离），779。参见 routing
Cyber 835 mainframe（Cyber 835 主机），924～925
cycles per instruction（CPI）（每指令周期数），138
cyclic assignment（循环分配），98，108
cyclic redundancy checks（CRCs）（循环冗余校验），608，633
SP networks and（SP 网络），822
trailer word（尾字），766

D

dancehall multiprocessor organization（舞池型多处理器组织结构），459（fig.）
data access（数据存取）
cost（成本），138
in multimemory system（在多存储器系统中），137～142
workload（工作负载），254
dataflow architecture（数据流体系结构），47～49
division（分割），48
dynamic scheduling（动态调度），51
execution pipeline（执行流水线），48（fig.）
graph（图），48（fig.）
operation naming（操作命名），48～49
tagged-token（加标记的令牌），48
tags（标签），47～48
tokens（令牌），47。参见 parallel architectures
Data General（Data General），570
data-local（局部数据），158
data mining（数据挖掘），80～81
for associations（关联），80～81
data queries vs.（数据查询），80
Data Mining application（数据挖掘应用），77，80～81，178～182
assignment（分配），83，180～181
barrier（栅障），106
decomposition（分解），180～181
disk access（磁盘存取），182
equivalence classes（等价类别），180，181，182
itemsets（项目集），180，278～279
lexicographic sorting（字典式的排序），182
load balance（负载平衡），180～181
mapping（映射），182
orchestration（协调），181～182
sequential algorithm（串行算法），179～180
spatial locality（空间局部性），182
summary（总结），182
synchronization（同步），182
tasks（任务，作业），82～83
temporal locality（时间局部性），182
data parallelism（数据并行），124，125
data parallel processing（数据并行处理），26，44～47
equation solver pseudocode（方程求解器伪代码），100（fig.）
languages（语言），47
multiple cooperating microprocessors（多协作微处理器），46

orchestration under（在……之下协调），99～101
"virtual processor" operations（"虚拟处理器"操作），49
data-race-free models（无数据竞争的模型），695
data races（数据竞争），696
data-remote（远程数据），158
data reuse（数据重复使用），246
data set size（数据集大小），206
data structuring optimization（数据结构优化），219
data traffic components（数据流量成分），360
data transfers（数据传输），58
bandwidth（带宽），60
granularity（粒度），145
occurrence of（……的发生），58，59
time for（……的时间），60～61
within machines（机器内部），59。参见 block data transfer
deadlock（死锁），378～379，791
avoidance assumption，breaking（回避假设，突破），594
buffer（缓冲区），412，594
in computer system（在计算机系统中），379（fig.）
cut-through routing（直通路由），792
DASH system and（DASH 系统），594
detection methods（检测方法），594
examples（例子），792（fig.）
fetch（取），390，412，483～485
freedom（自由度），791～795
"head on"（朝向），791
NACK solutions（NACK 解决方案），596
network transactions and（网络事务），473
NUMA-Q（NUMA-Q），633
potential（潜能），378，390
prevention（预防），379，412
SGI Origin2000（SGI Origin2000），595，608
store-and-forward routing（存储转发路由），792
at traffic intersection（在通信流的交聚点），379（fig.）。参见 livelock; starvation
deadlock-free networks（免死锁网络），483
DEC Alpha（DEC Alpha），12，693，699
DEC Memory Channel（DEC 存储器信道），520
hardware organization（硬件组织结构），521（fig.）
transmit regions（发送区域），520
decomposition（分解），83，84～88
Barnes-Hut application（Barnes-Hut 应用），169～170
of cache misses（高速缓存扑空），318
changes to（转变到），107～108
Data Mining application（数据挖掘应用），180～181
domain（领域），132，134（fig.）
equation solver kernel（方程求解器内核），93～98
goal of（……的目标），84
Ocean application（Ocean 应用），161～163
Raytrace application（Raytrace 应用），175～176
row-based（基于行的），98
scatter（散布），176，246。参见 parallelization process
DECOMP statement（DECOMP 语句），100～101
decoupled hardware approach（去耦合的硬件方法），707
dedicated message processing（专用消息处理），496～506
machine organization with embedded processor（具有嵌入处理器的机器组织），498（fig.）
machine organization with symmetric processor（具有对称处理器的机器组织），497（fig.）
Dekker's algorithm（Dekker 算法），284，688
delay（延迟）
cost（成本），152
network（网络），61，62，646～647
network transit（网络传输），152
propagation（传播），815
reducing（减少），152～153
routing（路由），460，758，761，781（fig.）
switching（交换），756
delayed-exclusive replies（延迟的排他应答），698
delta network（delta 网络），804
dependence order（依赖序），54
dependences（相关性），124
data（数据），94
in Gauss-Seidel equation solver computation（在高斯－赛德尔方程求解器计算中），95（fig.）
Ocean application，among grid computations（Ocean 应用，在网格计算中），162
deterministic routing（确定性路由），790～791
algorithms（算法），791
multiple routes（多路由），791
routing path conflicts（路由路径冲突），801（fig.）。参见 routing
diffs（diffs），717～718，737
application（应用），718
computation（计算），718
correct order of（……的正确顺序），738
ERC（ERC），745
LRC（LRC（纵向冗余校验）），717，746
processing（数据处理），718
storage，reducing（存贮，减少），739
digits（数位），248
dimension order routing（维序路由），789，800

directed acyclic graph（DAG）（有向无环图），797
direct-mapped caches（直接映射高速缓存），241，247，362（fig.）
direct memory access（DMA）（直接存储器访问），455
controllers（控制器），490，495
descriptor（描述符），493
devices（设备），486
engine（引擎），486，499～502，517
full-cache-block write（对高速缓存的满块写），422
HIO interface chips and（HIO接口芯片），422
IBM SP-2 engine（IBM SP-2引擎），41
input channels（输入通道），488，489
I/O transfers by（I/O传输），274
nonblocking sends and（无阻塞发送），40
output channels（输出通道），488
partial-block write（部分块写），422，423
performance advantages（性能优势），493
physical（物理的），486～491
read requests（读请求），422
read responses（读响应），422
SGI Origin2000 support（SGI Origin2000支持），610
transfers（传输），40，43，481，487，391，493
direct networks（直接网络），489
directories（目录）
basic operation of（……的基本操作），561（fig.）
block information（块信息），562
distributed，doubly linked list（分布式的，双向链表），569（fig.）
height（高度），568
height reduction（高度减少），659
hierarchical（层次式的），660，664～667
limited pointer（有限指针），568
on read miss（读失效），560
organization of（……的组织结构），565～571
sparse（稀疏的），659
state diagrams（状态图），670（fig.），671（fig.）
storage overhead，reducing（存储开销，减少），655～659
value of（……的值），564
width（宽度），568
width reduction（宽度减少），655～658
directory-based cache coherence（基于目录的高速缓存一致性），553～677
approaches（途径，路线），559～571
complexity（复杂性），669
hierarchical coherence（层次式的一致性），659～669
parallel software implications（并行软件的影响），652～655
performance parameters（性能参数），645～648
storage overhead reduction（存储开销减少），655～659
synchronization（同步），648～652
two-level organizations（二层组织），557（fig.）。参见 cache coherence
directory-based multiprocessors（基于目录的多处理器），553～677
directory controllers（目录控制器），562、639
directory lookup（目录查找），586、599
directory protocols（目录协议），556
assessing（评估），571～579
assist occupancy impact（辅助部件占用度的影响），646～647
cache block size effects（高速缓存块大小影响），579
correctness（正确性），589～596
definitions（定义），560
design challenges（设计挑战），579～596
directory state（目录状态），558
dirty node（脏节点），560，562
exclusive node（独占节点），560
home node（宿主节点），560
local node（本地节点），560
local vs. remote traffic（本地与远程流量比较），578～579
machine organization and（机器组织结构），587～589
network delay impact（网络延迟的影响），646～647
NUMA-Q（NUMA-Q），624
operation（操作），560～564
optimizations（优化），585～587
outer protocol as（外部协议），556
overview（概述），559～571
owner node（拥有者节点），560
performance（性能），584～589、645～648
scaling（扩放，伸缩），564～565
SGI Origin（SGI Origin），588
directory schemes（目录方案）
alternatives（可选的，另外的，其他的），567（fig.）
cache-based（基于高速缓存的），566，568～570
centralized（集中的），564
data sharing patterns（数据共享模式），571～577
flat（扁平的），565，567～571
hierarchical（层次的），565～567
memory-based（基于存储器的），566，567～568
operation（操作），560～564
summary（总结），571
directory states（目录状态）
busy（忙），603～604

exclusive（独占），603
directory trees（目录树），566
dirty bit（脏位），560，563
dirty node（脏节点），560，562，591
distributed，doubly linked list（分布式的，双向链表），569
distributed-memory organization（分布式存储器组织）
consistency model（同一模型），508
generic（一般的），458（fig.）
distributed queues（分布式队列），128
distributed shared memory（DSM）systems（分布式共享存储器系统），558
domain decomposition（区域分解），132
load balanced（负载平衡），133～134
for simple nearest-neighbor computation（对于简单的近邻计算），134。参见 decomposition
downgrade（降级），600
Dragon write-back update protocol（Dragon 回写更新协议），301～305，374
bus transactions（总线事务），302
exclusive-clean（E）state（独占的干净状态），302
lower-level design choices（低层设计选择），304～305
modified（M）state（已修改状态），302
for processor actions（为处理器操作），305（fig.）
read miss（读扑空），303
replacement（替换），304
shared-clean（Sc）state（共享的干净状态），302
shared-modified（Sm）state（共享的已修改状态），302
SRAM caches（SRAM 高速缓存），304
state transition diagram（状态转换图），303（fig.）
state transitions（状态转换），303～304
write（写），303～304。参见 update-based protocols
DRAM chips（动态随机存取存储器芯片），14
access time（访问时间），831
bit array（位阵列），950
buffer caches（缓冲区高速缓存），954
capacity（容量），14，949，950
cycle times（周期时间），938
designs（设计），949
engineering requirements for（工程要求），949
enhanced（提高的），949
gigabit（千兆位），946
integrating logic into（集成逻辑），949
interfaces（接口），950
packages（数据包），949
remote cache（远程高速缓存），663，700
vector，organization（向量，组织结构），953（fig.）
D-tags（D～标记），427，429
dual-ported RAM（双端口 RAM），382
dynamic assignment（动态分配），88
load balancing and（负载平衡），128
static assignment vs.（静态分配），126～129
dynamic backoff（动态回退），595
dynamic partitioning（动态分区），126，127（fig.）
dynamic scheduling（动态调度），870
latency and（时延），938
processors（处理器），895，922
Radix（基数），875（fig.）
dynamic tasking（动态任务），126～128
advantages（优势），127～128
example（实例），128
task pool（任务池），127，129（fig.）
techniques（技术），128～129
Dyna3D（Dyna3D），956～957

E

eager-exclusive replies（积极的独占应答），698～699
eager release consistency（ERC）（积极的释放同一性），712
diffs（差异），745
lazy vs.（与惰性方案相比），713
non-null pointer value（非空指针值），715～716
software（软件），715
echo（回应），824
negative（负面的），824
positive（正面的），824
test（测试），522
e-cube routing（e～立方体路由），778，790
effective bandwidth（有效带宽），756
efficiency-constrained scaling（效率约束的扩放），231
EM-4（EM-4），494，495
encapsulation（封装），754
Encore Gigamax（Encore 的 Gigamax 机）
block diagram（框图），664（fig.）
system configuration（系统配置），663。参见 hierarchical snooping
Encore Multimax（Encore Multimax），435
endpoint contention（端点竞争），154
end-to-end flow control（端到端流控），494，763，813，816～818
admission control（接纳控制），818
global communication operations（全体通信操作），817～818
hot spots（热点），817。参见 flow control
entry consistency model（入口同一性模型），722

equation solver kernel（方程求解器内核），92～116
accumulation and convergence determination in（累积与汇聚决策），114（fig.）
assignment（分配），98～99
data parallel, pseudocode（数据并行，伪代码），100（fig.）
decomposition（分解），93～98
with decomposition into grid points（分解到格点），98（fig.）
finite differencing method（有限差分法），92
ghost rows（阴影行），109，111
nearest-neighbor update（最近邻域更新），93（fig.）
orchestration under data parallel model（数据并行模型下的协调），99～101
orchestration under message-passing model（消息传递模型下的协调），108～116
orchestration under shared address space model（共享地址空间模型下的协调），101～108
picking cache sizes with（选择高速缓存的大小），240（fig.）
pseudocode（伪代码），94（fig.）
red-black ordering（红黑次序），97（fig.）
row-based, cyclic assignment of（基于行的，循环分配），108（fig.）
scaling model impact on（扩放模型的影响），211～213
shared address space model pseudocode（共享地址空间模型伪代码），104～105（fig.）
spatial locality in（空间局部性），149（fig.）
temporal locality in（时间局部性），143～144
error correction codes（ECCs）（差错校正码），608，633
error handling（差错处理）
NUMA-Q（NUMA-Q），633～634
SGI Origin2000（SGI Origin2000），608～609
Ethernets（以太网）
flow control（流控），812
switched gigabit（千兆交换），548，940
event-driven simulation（事件驱动模拟），233，234（fig.）
event synchronization（事件同步），57，103，283，887
acquire method（获取方法），335
components（成分），335～336
flags for（标记），106，107（fig.）
global（全局的），106，113，353～358
group（成组），107
identifying（识别），695
between pairs/groups of processes（在进程对或进程组之间），106
point-to-point（点到点），107（fig.），117，352～353
release method（释放方法），335
requirements of, through flags（需求，通过标记），285（fig.）
waiting algorithm（等待算法），335。参见 synchronization
exception program counter（EPC）（异常程序计数器），915～916，918～919
exclusive node（独占节点），560
exclusive pending（EP）state（独占挂起状态），925
exclusive reply（独占的应答），607
execution time（执行时间）
Barnes-Hut application（Barnes-Hut 应用），174（fig.），214
busy-overhead（忙碌开销），158
busy-useful（忙有用），157
components（成分），157（fig.）
data-local（局部数据），158
data-remote（远程数据），158
mapping（映射），159（fig.）
MC scaling（MC 扩放），211
multithreading, breakdowns（多线程，分解），913（fig.）
Ocean application（Ocean 应用），166（fig.）
perfect-memory（完美的存储器），210
as performance metric（作为性能指标），203
Raytrace application（Raytrace 应用），179
synchronization（同步），158
explicit communication（显式的通信），116
advantages of（优势），189
flexibility（灵活性），187。参见 communication
explicit synchronization（显式同步），285（fig.）
exponential backoff（指数回退），649，651
extended memory hierarchy（扩展的存储器层次结构），138～140
artifactual communication in（附加通信），139-140
improving architecture of（改善体系结构），139
in multiprocessors（在多处理器中），138～139，271（fig.）

F

false sharing（假共享），146，226，322，360
of cache blocks（高速缓存块），711
cause of（原因），329，360
cross-particle（跨粒子的），365
effects of（效应），710
page fault from（缺页），711
protocol operations and（协议操作），710
reducing（减少），328，329，361（fig.）
SVM communication from（SVM 通信形式），712（fig.）
write-write（写－写入），410
false-sharing misses（假共享扑空），315～316

block size and（块尺寸），318
increase in（增长），319
likelihood of（可能性），316
reducing（减少），316
Fast Fourier Transform（FFT）（快速傅立叶变换），196
block data transfer benefits in（块数据传送的好处），861（fig.）
implementation（实现），549
kernel performance with consistency models（同一性模型下的内核性能），874（fig.）
radix-2（基数－2），548
sweet spot（扫描点），859
fat cube（胖立方体），613
fat-trees（胖树），774
d-dimensional k-ary（d－维 k 元），776
illustrated（说明），777（fig.）
routing（路由），778
fetch&add instruction（取数加指令），340，372
fetch&decrement instruction（取数减量指令），340
fetch&increment instruction（取数增量指令），340，346～347
fetch&op instruction（取数且操作指令），611，613
at-memory（在存储器），651
four-entry LRU（四表项 LRU），617
fetch&setup instruction（取数建立指令），340
fetch&store instruction（取数存贮指令），540
fetch deadlock（取死锁），390，412，483～485
problem cause（问题原因），484
solution（解决方案），484。参见 deadlock
Fiber Channel（光纤信道），637
Arbitrated Loop（仲裁环），769
switch（交换机），637
FIFO（先入先出）
buffers（缓冲区），409，616，804
input/output（输入/输出），493
physical（物理的），615
request-response ordering（请求－响应次序），409
transmit/receive（发送/接收），500
virtual（虚拟的），615
fine-grained tasks（细粒度任务），83
finite differencing method（有限差分法），92
finite element method（有限元法），957
Firefly update protocol（Firefly 更新协议），374
fixed-size machine evaluation（固定尺寸机器评价），221～226
inherent behavioral characteristics（固有行为特征），221～222
problem size range determination（问题规模范围判定），221
spatial locality（空间局部性），224～226
temporal locality and working sets（时间局部性与工作集），222～224
flat directory schemes（扁平目录方案），565，567～571
cache-based（基于高速缓存的），566，568～570
COMA（COMA），704
full bit vector organization（全比特向量组织），567～568
latency reduction（时延减少），586（fig.），587（fig.）
memory-based（基于存储器的），566，567～568
overflow strategy（溢出策略），568。参见 directory schemes
floating-point operations（浮点操作），101，209
FLOPs（FLOPs），230，257
flow control（流控），399，404，811～818
ATM（异步传输模式），813
CRAY T3D network（CRAY T3D 网络），820
end-to-end（端到端），494，763，813，816～818
in Ethernet network（在以太网络中），812
link-by-link（逐链路地），489
link-level（链路级），813～816
main memory（主存储器），404
in ring-based LANs（在基于环的局域网中），812
SGI Challenge（SGI 的挑战），424
split-transaction bus（事务拆分型总线），404
TCP（TCP），813
in wide area networks（在广域网中），812～813
Flynn's taxonomy（Flynn 分类法），44
fork-in approach（分叉-汇合方法），136
forwarding（转发），594
to dirty node（到脏节点），591
intervention（干预），585，586
reply（应答），585，586，597
forward pointer（前向指针），569
four-state invalidation protocol（四状态作废协议），299～301
four-state update protocol（四状态更新协议），301～305
fragmentation（碎片），360，754
full bit vector organization（全比特向量组织），567～568
full-empty bit（满-空位），352～353
flexibility and（灵活性），353
set to empty（置为空），352
full-word operation（全字操作），15
function parallelism（操作并行），124
availability（可用性），125
exploiting（开发），125
types of（类型），124～125

G

galaxy evolution case study.（星系进化案例研究）。参见 Bar-

nes-Hut application
Gaussian elimination（高斯消去法），118
algorithms（算法），194
concurrency（并发），194～195
parallelizing（并行化），119
sequential, pseudocode（串行的，伪代码），119（fig.）
Gauss-Seidel method（高斯-赛德尔方法），92
concurrency（并发），95（fig.）
dependencies（相关），95（fig.）
generic parallel architecture（通用并行体系结构），50～52
communication assist（通信辅助部件），50～51
convergence toward（向……收敛），50
organization（组织结构），51（fig.）。参见 parallel architectures
ghost rows（附影行），109，111
Gigaplane system bus（Gigaplane 系统总线），424～427
centerplane design（中央平面设计），424
collision-based speculative arbitration（基于冲突的投机仲裁）425
invalidations and（作废），427
outstanding transaction support（未决事务支持），425
signals（信号），425
signal timing（信号定时），426（fig.）。参见 Sun Enterprise 6000
global coherence（全局一致性），855～856
global synchronization（全局同步），95，96
barrier algorithms（栅障算法），356～357
centralized barrier with sense reversal（原著误为 removal）（具有感应逆转的集中型栅障），354～356
centralized software barrier（集中型软件栅障），353～354
event（事件），106，113，353～358
between grid sweeps（在网格遍历之间），96
hardware barriers（硬件栅障），358
hardware primitives（硬件原语），357～358
performance（性能），356
point（点），156。参见 synchronization
global trees（全局树），957
Goodyear MPP（Goodyear MPP），952
GO symbol（GO 符号），815
Grand Challenge（重大挑战）
application requirements（应用需求），7（fig.）
example（例子），8
granularity（粒度），183
of allocation（分配），146，237，724，725（fig.）
cache block（高速缓存块），706
coarse（粗糙的），724
of coherence（一致性），146，724，725（fig.）
communication（通信），186～187，724，725（fig.）
data transfer（数据传输），145
exploiting（发掘），145
fine（细的），724，725
memory consistency model and（存储器同一性模型），681
page（页），725
Raytrace application（Raytrace 应用），177
task（任务），129～131
grid computations（格计算）
iterative nearest-neighbor（近邻迭代），264
Ocean application（Ocean 应用），162（fig.），163，164
group event synchronization（组事件同步），107
guided self-scheduling（引导式自调度），128

H

half-power point（半功率点），60
HAL S1 multiprocessor（HAL S1 多处理器），643～645
bandwidths（带宽），644
communication assist（通信辅助部件），644
DMA engine（DMA 引擎），644
MCU（MCU），644
hardware-controlled prefetching（硬件控制的预取），880，881～883
advantages（优势），888，896
coverage（覆盖），889
disadvantages（劣势），889
effectiveness（有效性），889
goal（目标），881
history table（历史表），882
LA-PC（LA-PC），883
OBL schemes（OBL 方案），882
scheduling（调度），883
schemes（方案），881
software-controlled vs.（与软件控制方案比较），888～890
unnecessary prefetch reduction（减少不必要的预取），889。参见 prefetching
hardware cost（硬件成本），188
access control through code instrumentation and（通过代码修改的访问控制），707～708
access control through language/compiler support and（通过语言/编译器支持存取控制），721～724
access control with decoupled assist and（带有独立辅助部件的访问控制），707
CC-NUMA（CC-NUMA），705～706
page-based access control and（基于页的存取控制），

709 ~ 721
reducing（减少），705 ~ 724
hardware locks（硬件锁），337 ~ 338
around instruction sequence（围绕指令序列的），339，参见 locks
hardware/software trade-offs（硬件/软件折中），679 ~ 747
capacity for replication（复制容量），679 ~ 680
COMA（COMA），701 ~ 702
design/implementation cost（设计/实现成本），679 ~ 680
parallel software implications（并行软件的影响），729 ~ 730
waiting time at memory operation（存储器操作的等待时间），679 ~ 680
header（头），754 ~ 792
head node（头节点），624
rolling out scenario（转出方案），631
of sharing list（共享表），628
write（写），630
head-of-line blocking（行头阻塞），805
avoiding with switch design（通过交换开关设计来避免），807（fig.）
impact of（影响），806
head pointer（头指针），569
HEP（HEP），905，906，907
hierarchical coherence（层次式一致性），659 ~ 669
hierarchical directories（层次式目录）
branching factors（分枝系数），669
latency problem（时延问题），668
multirooted（多根的），667（fig.）
organization（组织结构），666（fig.）
storage overhead（存储开销），667。参见 directories
hierarchical directory scheme（层次目录方案），565 ~ 567
advantages（优势），566
bandwidth characteristics（带宽特征），566
COMA（COMA），703 ~ 704
latency characteristics，（时延特征），566
location information（位置信息），566
processing nodes（处理节点），565，566。参见 directory schemes
hierarchical parallelism（层次式并行），193
hierarchical ring-based multiprocessor（层次式基于环的多处理器）665（fig.）
hierarchical snooping（层次式侦听），559，660 ~ 664
bus hierarchy（总线层次），660
bus transactions（总线事务），661
Encore Gigamax（Encore 的 Gigamax 机），663 ~ 664
interconnection network（互连网络），666
processor caches（处理器高速缓存），660
remote caches（远程高速缓存），660
write serialization（写串行化），663，参见 snooping
Hierarchical Uniform Grid（HUG）（层次式均匀格），174
High Performance Fortran（HPF）（高性能 Fortran），723，967
hit rate（命中率），943
home-based protocols（基于宿主的协议），717
LRC（LRC），746
performance advantage（性能优势），717 ~ 718
home node（宿主节点），560
need for local serialization at（局部串行化的需要），606（fig.）
page（页），717
Horizon design（水平设计），905
cycle（周期），906
lookahead（先行），906
hot spots（热点）
adaptive routing and（自适应路由），817
end-to-end flow control（端到端流控），817
network（网络），760
HP PA-8000 processor（惠普 PA-8000 处理器），684，868，940
hybrid protocols（混合协议），330 ~ 332
competitive scheme（竞争方案），331
miss rates for（扑空率），322（fig.）
mixed approaches（混合方法），331
upgrade/update rates for（升级/更新率），333（fig.）。参见 protocols
hypercubes（超立方体），38，778
links（链路），782
ports（端口），778
uses（使用），778。参见 network topologies

I

IBM PowerPC（IBM PowerPC），693，699
G30（G30），940
systems（系统），12
IBM SP-1/SP-2（IBM SP-1/SP-2），40，41（fig.），466
BT benchmark（BT 基准测试程序），533
LU benchmark（LU 基准测试程序），533
network case study（网络案例研究），820 ~ 822
output port（输出端口），822
packets（数据包），821
switch design（交换开关设计），823
switch packaging（交换开关封装），821（fig.）
workstation as node support（工作站作为节点支持），467
idle time（空闲时间），898

Illiac IV (Illiac IV), 67
implicit communication (隐式通信), 189
inclusion (包含), 393 ~ 394
fall out (脱离), 394
maintaining (维护, 维持), 394 ~ 396
requirements (需求), 394
IncPIO requests (IncPIO 请求), 424
indefinite postponement (无限推迟), 791
infinite caches (无限高速缓存), 572
inherent behavioral characteristics (固有行为特征), 221 ~ 222
inherent communication (固有通信), 140
inner protocol (内部协议), 556
input buffers (输入缓冲), 472
associated with incoming channels (与输入通道相联), 490
full (全), 484
management of (……的管理), 482
overflow (上溢), 482 ~ 483
scalability and (可扩展性), 484
space availability (可用空间), 484
state of (……的状态), 483。参见 buffers
input-output microbenchmarks (输入输出微基准测试程序), 215
instruction-level parallelism (指令级并行), 15 ~ 17
distribution of (……的分布), 18
increasing amount of (数量增加), 17
superscalar execution (超标量执行), 17
integration (集成)
board-level (板级), 465 ~ 466
chip-level (芯片级), 463 ~ 464
system-level (系统级), 466 ~ 467
Intel Paragon (Intel 的 Paragon 机), 8, 41, 42 (fig.), 68
in ASCI Red machine (在 ASCI Red 机中), 502, 503
case study (案例研究), 499 ~ 503
communication latency (通信时延), 525
CP elimination (取消通信处理器), 525
DMA engines (DMA 引擎), 499 ~ 502
DMA transfers (DMA 传输), 500, 501
flow control (流控), 942
i860XP (i860XP) 500, 501
machine organization (机器组织结构), 500 (fig.)
NI chip (网络接口芯片), 499, 500
node operating systems (节点操作系统), 526
nodes (节点), 499
output buffer (输出缓冲), 550
processing elements per node (每个节点的处理单元), 740
send overhead (发送开销), 524
shared address read (共享地址读取), 527 ~ 528
Intel Pentium Pro (Inter 奔腾 Pro), 868, 935
cache coherence memory bus (高速缓存一致性内存总线), 935
four-processor "quad-pack", (四处理器模块), 32 (fig.), 33 (fig.)
processor consistency model (处理器同一性模型), 698
processor module (处理器模块), 641
interconnection network (互连网络)
design (设计), 749 ~ 830
generic parallel machine (类属并行机), 751 (fig.)
scalable (可扩放, 可伸缩), 764, 参见 networks
interleaved multithreading (交错多线程), 902 ~ 910
advantages (优势), 902, 911
basic (基础), 905
blocked scheme vs. (分块方案与……相比), 911
context availability (上下文可用性), 918
control (控制), 920
disadvantages (劣势), 912
evolution (演变), 904
implementation issues (实现问题), 917 ~ 920
latency tolerance (时延包容), 904 (fig.), 911 (fig.)
long-latency events and (长时延事件以及), 908
overhead (开销), 909
PC bus (PC 总线), 919 (fig.)
PC unit (PC 单元), 918 ~ 919
pipeline use (流水线使用), 905 ~ 908
requirements (需求), 914
short-latency events and (短时延事件以及), 909
single-thread pipelined support (单线程流水支持), 908 ~ 910
speedup (加速比), 912 (fig.)
state replication (状态复制), 917 ~ 918
Tera (万亿), 906 ~ 908。参见 multithreading
intervention (干预), 585
forwarding (前向), 585, 586
request buffers (IRBs) (请求缓冲 (IRBs), 615
invalidation acknowledgments (作废确认), 607, 699
invalidation-based protocols (基于作废的协议), 278
lock transfers (锁传输), 675
MESI (four-state) (MESI (四状态)), 299 ~ 301
misses and (扑空), 887
miss rates (扑空率), 332 (fig.)
MOESI (five-state) (MOESI (五状态)), 300
MSI (three-state) (MSI (三状态)), 293 ~ 299
update-based protocols combined with (基于更新的协议与

……结合)，330～332
update-based protocols vs. (基于更新的协议与……相比较)，329～330
upgrade/update rates for (升级/更新率)，333 (fig.)
write atomicity (写操作的原子性)，592～593
write-back (回写)，293～299，391
write-through (直写)，280，283 (fig.)
invalidation frequency (作废频率)，572
invalidation patterns with default data sets (缺省数据集的作废模式)，574～575 (fig.)
irregular read-write (不规则读－写)，573
migratory (迁移)，573
producer-consumer (生产者－消费者)，573
read-only (只读)，572
invalidation size distribution (作废尺寸的分布)，572
invalid pending (IP) state (无效的挂起 (IP) 状态)，925
I/O buses (输入/输出总线)，515
IQ-Link board (IQ-链路板)，622，623，643
block diagram (块图)，636 (fig.)
efficiency (效率)，643
implementation (实现)，639～641
IRIX (IRIX (一种类 UNIX 操作系统))，415
Itemsets (项目集合)，81
iWARP (iWARP)，69，494，495

J

Jade programming language (Jade 编程语言)，723
J-machine (J-machine)，494～495，810
execution contexts (执行现场)，495～496
3D torus (3 维花环)，771

K

Kernels (内核)，92，216
benchmarks (基准测试程序)，967
data structure sharing (数据结构共享)，218
equation solver (方程求解器程序)，92～116
examples (例子)，216
interaction among (在……之间互相作用)，217
message-passing abstraction support (消息传递抽象支持)，488
performance-relevant characteristics (性能相关的特征)，217
software (软件)，488
user linkages (用户链接)，488
keys (关键字)，248

L

LAN (局域网)
controllers，(控制器)，490～491
flow contro (流控)，812
interfaces (接口)，490～491
ring (环)，514
scalable high-performance (可伸缩高性能)，503
shared bus (共享总线)，514
LAPACK suite (LAPACK 组)，963
latency (时延)，17，59，151
array-based lock (基于数组的锁)，347
avoidance (避免)，938
back-to-back (背靠背)，619，620 (fig.)
bandwidth vs. (带宽)，59，787
Barnes-Hut (Barnes-Hut)，913
barrier (栅障)，356
of cache misses (高速缓存扑空)，17，324
CM-5 (CM-5)，493
COMA，(仅高速缓存存储访问)，729
communication (通信)，522～523，525～526，833
contention and (竞争)，759，786，787 (fig.)
dynamic scheduling and (动态调度)，938
flat cache-based directory protocol，(基于高速缓存的扁平目录协议)，587 (fig.)
flat memory-based directory protocol (基于内存的扁平目录协议)，586 (fig.)
hiding (隐藏)，155，842，849 (fig.)，870，871，877，914
hierarchical directory (层次式目录)，566，668
LL-SC lock (LL-SC 锁)，346
lock (锁)，343
lock acquisition (锁获取)，337
memory (存储器)，833
network (网络)，755～761
network transaction (网络事务)，475
NUMA-Q (NUMA-Q)，641～642
performance degradation from (性能退化)，17
phits vs. (物理单元与……相比较)，788 (fig.)
pipelined (流水线式)，618，911
reducing (减少)，831～832
scaling (伸缩，扩放)，460～461
shared memory approach and (共享存储器解决方案)，37
sharing list purge (共享表清除)，629
snoop (侦听)，383
test-and-test&set lock (测试-测试并设置锁)，343
trade-offs (权衡)，784
true unloaded (真卸载)，619
unloaded (卸载)，755，756，780～785
latency tolerance (时延包容)，832～951

application limitations（应用限制），843
approaches（解决方案），837～840
assist occupancy and（辅助部件占用度），845
benefit bounds（受益界限），843（fig.）
benefits（益处），841～843
block data transfer（块数据传送），837，838，848
in blocked multithreading（阻塞的多线程），903（fig.）
cache-coherent multiprocessor（高速缓存一致的多处理器），926～927
communication architecture limitations（通信体系结构限制），843～846
communication pipeline and（通信流水线），836～837
goals（目标），836
in interleaved multithreading（交错多线程），804（fig.）
interleaved scheme（交错方案），911（fig.）
limitations（限制），843～847
in message passing（消息传递），847～851
multithreading（多线程），840，850～851
network capacity and（网络容量），846
no（无），834
overhead and（开销），845
overview（概貌），834～847
point-to-point bandwidth and（点对点带宽），845～846
precommunication（预通信），838～839，848～850
proceeding past communication in same thread（在同一线程中推进越过通信），839～840，850
processor limitations（处理器限制），846～847
requirements（需求），841
in shared address space（在共享地址空间中），851～852
techniques（技术），647，926～927
timelines for different forms（不同形式的时间量），844（fig.）
write（写），876
latency under load（有载时延），785～788
comparison（比较），786
with routing delay（带有路由延迟），787（fig.）
latency wall（时延障碍），940～942
network interface（网络接口），941～942
parallel architecture evolution and（并行结构演变），941
speed-of-light（光速），943
lazy release consistency（LRC）（惰性释放同一性（LRC）），713
acquire-based（基于获取的），738
complexity（复杂性），715
diffs（差别），717，736
eager vs.（积极型与……相比较），713（fig.）
home-based（基于宿主的），746
implementations（实现），715
write notices（写通知），717，参见 release consistency（RC）
least recently used（LRU）（最近最少使用（LRU））
caches（高速缓存），259
replacement（替换），395
limited pointer directories（有限指针目录），568
LimitLESS scheme（无限制方案），658
linear arrays（线性数组），769，770（fig.）
linear scaling property（线性扩放特性），209
link-by-link flow control（逐链路流控），489
link-level flow control（链路级流控），813～816
handshake（握手），815（fig.）
as host-switch links（作为主机交换开关链路），816
illustrated（被说明的），814（fig.）
implementation（实现），813
problem with（问题），817
short-narrow link（短窄链路），814
short-wide link（短宽链路），814。参见 flow control
link-level protocol（链路级协议），751
links（链路），751，764～767
advancement of（进步），939
asynchronous（异步的），765
bandwidth（带宽），760，939
butterfly（蝶型的），775～776
CRAY T3D network（CRAY T3D）网络，819，820
framing（成帧），765
host-switch（主机交换开关），816
hypercube（超立方体），782
long（长），764，815（fig.）
narrow（窄），764～765，815
propagation delays（传播延迟），815
SCI（SCI），766
SGI Origin network（SGI Origin 网络），825
short（短），764
synchronous（同步的），765
torus（花环），775
tree（树），775
wide（宽），764～765
width（宽度），781（fig.）
LINPACK benchmark（LINPACK 基准测试程序）13，503
matrix factorization（矩阵因式分解），209
microprocessor-based systems on（基于微处理的系统），22
supercomputer/MPP performance on（超级计算机/MPP 性能），24（fig.）
uses（使用），23。参见 benchmarks

Little's Law（Little's定律），944
livelock（活锁），379，390，791
avoiding（避免），390
directory protocols and（目录协议），595
LL-SC implementation and（LL-SC实现），392
NACK solutions（NACK解决方案），596
NUMA-Q（NUMA-Q），633
potential（潜在的），379，390，392
problem solution（问题解决方案），380
SGI Origin2000（SGI Origin2000），608。参见 deadlock；starvation
load balancing（负载平衡），88，123～131
dynamic partitioning for（动态分区），127（fig.）
goal of（目标），124
process（进程），124
as software responsibility（作为软件的责任），131
static assignment and（静态分配和），126
workload（工作负载），254～257
load-locked（LL）instruction（负载锁定（LL）指令），344，345
load-locked，store-conditional（LL-SC）locks（负载锁定，条件存储（LL-SC）锁），344～346
exponential backoff and（指数式延迟回退），649，651
guarantees（保证），345
guidelines（指南），345
implementing（实现），392，652
for implementing atomic operations（实现原子操作），344
latency（时延），346
livelock and（活锁），392
performance（性能），348～349，350，392
R10000 processor instructions（R10000处理器指令），611
spin-lock built with（用……构造的踏步锁），346
storage（存储），346。参见 locks
local blocks（本地块），560
local caches（本地高速缓存），139
local coherence（本地一致性），855
locality（局部性）
characteristics，changing（特征，变化），141
exploiting（开发），57，139，142，143～150
parallelism and（并行性），14
spatial（空间的），142，145～150
temporal（时间的），142，143～145
in tree network（在树状网络中），668
local memory microbenchmarks（本地存储器微基准测试程序），215
local node（本地节点），560
local spinning（本地踏步），543～545
local state monitor（本地状态监视），661
lock algorithm（锁算法），337，538～541
advanced（高级的），346～348
enhancements（增强），342～343
high-contention/low-contention and（高竞争/低竞争），675
lock-free synchronization（无锁同步），351
locks（锁）
accessing（访问），336
acquiring（获取），106
acquisition latency（获取时延），337，343
array-based（基于数组的），347～348，539，651
expense of（……的开销），106
fairness（公平性），343
hardware（硬件），337～338
LL-SC（LL-SC），344～346
performance（性能），340～342，348～350
performance goals（性能目标），343～344
QOLB（QOLB），651
queuing（排队），651
scalability（可伸缩性），343
on SGI Origin2000（在SGI Origin2000机上），650（fig.）
software（软件），338～340
software queuing（软件队列），539
storage cost（存储成本），343
test&set（Test and Set），340，341（fig.），342
test-and-test&set（测试-测试并设置），342～343
ticket（入场卷），346～347
traffic（流量），343
lockup-free caches（无锁高速缓存），414
Cyber 835 mainframe（Cyber835大型机），924
design（设计），922～926
design questions（设计问题），923～924
at memory system level（在存储系统级），930
options（选择），924
LogP model（LogP模型），191
L_1 caches（L_1高速缓存）
set-associative（组相联），394～395
write-back（回写），396
write-through（直写），396
long links（长链路），764
lookahead buffer（先行缓冲），870
lookahead program counter（LA-PC）（先行程序计数器），883
lookup-free cache design（无锁高速缓存设计），922～926
loop nests（循环嵌套），93，95
loops（循环）

analysis of（分析），93
bounds（界），113
parallelizing（并行化），124
parallel, self-scheduling of（并行、自调度），128
sequential（串行化），93
low-voltage differential signal（LVDS）（低电压差分信号），766
LU benchmark（LU 基准测试程序），532
application performance（应用程序性能），533（fig.）
communication characteristics（通信特征），551（fig.）
speedups（加速比），532
working set curves（工作集曲线），537（fig.）。参见 benchmarks
LU factorization（LU 因式分解），244～248
computational complexity of（计算复杂性），244
Gaussian elimination and（高斯消去法），244
invalidation pattern（作废模式），574（fig.）
invalidations（作废），576
load imbalance（负载不平衡），256
locks and（锁），248
miss rates（扑空率），320（fig.）
parallel blocked（并行分块的），247（fig.）
sequential blocked（串行分块的），245（fig.）
spatial locality（空间局部性），320，322
speedup（加速比），431
speedup on SGI Origin2000（SGI Origin2000 上的加速比），621（fig.）
traffic vs. local cache（流量与本地高速缓存比较），582（fig.）
traffic vs. number of processors（流量与处理器数量比较），580（fig.）
writing block in（写入块），576。参见 workload case studies

M

machine size（机器规模），206
main memory（主存储器）
flow control（流控），404
out-of-order responses and（乱序响应），409～410
SGI Challenge（SGI 挑战），415，420
subsystem（子系统），405
Manchester Dataflow Machine（曼彻斯特数据流机），494
mapping（映射）83，89～90
Barnes-Hut application（Barnes-Hut 应用程序），173
cumulative performance（累积性能），531
Data Mining application（数据挖掘应用程序），182
execution time（执行时间），159（fig.）
Ocean application（Ocean 应用程序），165
parallel algorithms（并行算法），153
Raytrace application（Raytrace 应用程序），178。参见 parallelization process
MasPar（MasPar），952
massively parallel processors（MPPs）（大规模并行处理器），23
dedicated proprietary network（专用网络），454
packaging（封装），454
performance（性能），24（fig.）
matrix transposition（矩阵转置），876
process（进程），675～677
sender-initiated（发送者启动的），676（fig.），929（fig.）
MC（memory-constrained）scaling（MC（存储器约束）的伸缩），207～208，431，433
communication-to-computation ratio（通信与计算之比），212，433
concurrency（并发性），212
execution time（执行时间），211
memory requirements（内存需求），212
naive（简单的），433
realistic（真实的），433
scaled speedup（可扩展的加速比），210
SGI Origin 2000（SGI Origin2000），621～622
spatial locality（空间局部性），213
speedup（加速比），211，213
synchronization（同步），213
temporal locality（时间局部性），213
work（工作），211
working set（工作集），227。参见 scaling; scaling models
media arbitration（介质仲裁），472
Meiko CS-2（Meiko CS-2），503～506，953
asymmetric CP（非对称 CP），503
case study（案例研究），503～506
communication assist（通信辅助部件），505（fig.）
communication latency（通信时延），525～526
conceptual structure（概念性结构），504（fig.）
design shortcoming（设计缺陷），506
DMA processor（DMA 处理器），505，506
elan（elan（一种处理器的名字）），504
flow control（流控），942
machine organization（机器组织结构），505（fig.）
MBUS（MBUS），503
network transactions（网络事务），503，505
node architecture（节点体系结构），503
node operating systems（节点操作系统），526
page table（页表），506
send overhead（发送开销），524～525

shared address read（共享地址读取），528
memory（存储器）
attraction（吸引），701
main（主），404，405
mainframe（主机，大型机），30
reflective（反射性），515，518
shared memory parallel program（共享内存并行程序），30
shared virtual（共享虚拟），709～724
memory accesses（存储器访问）
average（平均），942
frequency of（频率），310
memory barrier（MEMBAR）（存储器栅障（MEMBAR）），688～689，692～693
memory-based directory schemes（基于存储器的目录方案），566，567～568
latency reduction in（时延降低），586（fig.）
SGI Origin（SGI Origin），596～622。参见 directory schemes
memory consistency（存储同一性），283～291
at acquire（在获取点），734～737，738～739
at release（在释放点），733～734，737～738
sequential（串行的），286～291
memory consistency models（存储同一性模型），285
cost and（成本），681
granularity and（粒度），681
microprocessor（微处理器），699
NUMA-Q（NUMA-Q），633
in real multiprocessor systems（在实时微处理器系统中），698～700
relaxed（放松的），681～700
SGI Origin2000（SGI Origin2000），607～608
shared address space（共享地址空间），681
memory/directory interface（MI）（存储器/目录接口），617
memory latency（存储器时延），833
hiding（隐藏），870
two contexts and（双上下文），913。参见 latency；latency tolerance
memory management unit（MMU）（存储管理单元），507
memory operations（存储器操作），275～276
for execution with write-through invalidation protocol（使用直写作废协议的执行），283（fig.）
incomplete，in SC（未完成，在 SC 情况），871
parallel case and（并行情况），276
reordering（重排序），290
with respect to processors（相对处理器而言），275～276
sequential consistency（顺序同一性），286
within instructions（在指令中），275
memory system design（存储系统设计），305
multiprocessor（多处理器），276
SGI Challenge（SGI Challenge），424
Sun Enterprise 6000（Sun Enterprise 6000），429
memory usage（存储器使用），206
meshes（网格），38，770
with dimension order routing（维序路由），800
two-dimensional（二维的），771
uniform wire density（一致的连线密度），784
MESI write-back invalidation protocol（MESI 回写作废协议），299～301
bandwidth requirement（带宽需求），307～311
bus actions（总线动作），311
bus transactions（总线事务），299
cache-to-cache sharing（高速缓存对高速缓存的共享），300
exclusive clean state（独占的干净状态），299
lower-level design choices（低层设计选择），300～301
owned state（拥有状态），300
SGI Challenge（SGI Challenge），422
shared signal（S）（共享信号），300
stable states（稳定状态），387
states（状态），299
state transition diagram（状态转换图），301（fig.），388（fig.）
state transitions（状态转换），299～300
variants（变量），299。参见 MSI write-back，invalidation protocol
message passing（消息传递），26，37～42
abstraction（抽象），39（fig.），488
asymptotic bandwidths（渐进带宽），529（fig.）
asynchronous（异步的），479（fig.）
bandwidth vs. message size（带宽与消息大小的关系），531（fig.）
collective communication operations（集合通信操作），55
communication（通信），111
convergence（收敛），42～43
equation solver pseudocode（方程求解器伪代码），110～111（fig.）
implementation（实现），43
libraries（库），43，52
local references（局部引用），187
machines（机器），38，39（fig.）
mutual exclusion（互斥），538
naming（命名），56
operations（操作），43，56

operation time vs. message size（操作时间与消息大小的关系），530（fig.）
orchestration under（在…… 下协调），108～116
ordering（次序），56
parallel software operations（并行软件操作），528～531
point-to-point events（点对点事件），538
primitives（原语），109
replication（复制），185
research efforts（研究努力），68
send/receive pair（发送/接收对），476，481
start-up costs（启动开销），60，529（fig.）
store-and-forward（存储转发），40
synchronization（同步），111，113
synchronous（同步的），39，478（fig.）
three-phase protocol（三阶段协议），480
user communication（用户通信），38
user-level（用户级），476，477，490
uses（使用），38。参见 shared address space
message-passing architectures （消息传递体系结构），37
node packaging（节点封装），37
processors in（处理器），42
Message Passing Interface（MPI） （消息传递接口（MPI）），112，477，957，967
message passing latency tolerance（消息传递时延包容），847～851
block data transfer（块数据传送），848
multithreading（多线程），850～851
precommunication（预通信），848～850
precommunication latency hiding（通过预通信隐藏时延），849（fig.）
proceeding past communication in same thread（在同一线程中处理过去的通信），850
structure of communication（通信结构），848。参见 latency tolerance
messages（消息），751
flow control（流控），753
making larger（变大）838，839（fig.）
overhead（开销），755
size of（尺寸），762
MFLOPS（millions of floating-point operations per second）（每秒百万次浮点操作），230，231
microbenchmarks（微基准测试程序），215～216，967
communication（通信），216
echo test（回应测试），522
experiment results（实验结果），217（fig.）
implementation（实现），216
input-output（输入输出），215
local memory（本地存储器），215
processing（处理），215
role of（……的角色），216
SGI Origin2000 characterization with（SGI Origin2000 的特征），618～620
synchronization（同步），216。参见 benchmarks
microprocessors（微处理器），63～64
architecture design trends（体系结构设计趋势），15～19
bit-level parallelism（位级并行），15
bus bandwidth（总线带宽），19，21
bus-based shared memory（基于总线的共享存储器），20
CISC，19
clock frequency improvement（时钟频率提高），13
engineering requirements for（工程需求），949
era（时代），6
fraction devoted to memory（用于存储器的一部分），947（fig.）
growth curve（增长曲线），15
logic density improvement（逻辑密度改进），13
markets（市场），19
memory consistency models（存储同一性模型），699
parameters（参数），71（fig.）
Pentium（Pentium），20
performance（性能），4，13
RISC（RISC），19
transistors（晶体管），946，948（fig.）
user-level interrupts and（用户级中断），491
MIPS（millions of instructions per second），（每秒百万条指令），230，231，257
MIPS R4000 processor（MIPS R4000 处理器），910
MIPS R10000 processor（MIPS R10000 处理器），597，684，868
misses（扑空），参见 cache misses
miss state holding registers（MSHRs）（扑空状态保存寄存器），924～925
access（访问），924
contents（内容），924
state entries（状态入口），925（fig.）
MOESI protocol（MOESI 协议），300
Monsoon（一种机器的名称），494，495，496
MSI write-back invalidation protocol（MSI 回写作废协议），293～299
bus read exclusive（总线读独占），294
coherence satisfaction（一致性满足），297
illustrated（说明），295（fig.）

invalid state（无效状态），293
lower-level design choices（较低层设计选择），298～299
modified state（已修改状态），293
for processor transactions（处理器事务），297（fig.）
sequential consistency satisfaction（顺序同一性满足），297～298
shared state（共享状态），293
state transitions（状态转换），294～296
transactions and（事务），293～294
write to unmodified blocks（对未修改块的写），296。参见invalidation-based protocols
multicast（组播），120
multidimensional meshes（多维网格），770
multilevel cache hierarchies（多级高速缓存层次结构），393～398
coherence and（一致性），393
dual tags and（双标志），397
handling（处理），393
inclusion property（包含性质），393～394
internal queues（内部队列），411（fig.）
intrahierarchy protocol（层内协议），397
propagating transactions in（传播事务），396～398
split-transaction bus with（事务拆分型总线），410～413
two-level（二级），394（fig），398（fig.）
multimemory system（多存储器系统），137～142
multipath routing（多路径路由），800
multiple-context processing（多上下文处理），897
multiple-instruction-multiple-data（MIMD）（多指令-多数据（MIMD）），44
multiple outstanding references（多重未完成引用），413～414
exploiting（发掘），414
semantic implications of（语义蕴含），414
write buffer example（写缓冲实例），413
multiple writer protocols（多写入者协议），716～718
with acquire-based consistency（基于获取的同一性），738～739
automatic update mechanism（原子的更新机制），719（fig.）
home-based（基于宿主的），717
problem（问题），717（fig.）
with release-based consistency（基于释放的同一性），737～738
software method（软件方法），717
TreadMark SVM（TreadMark SVM），716
multiprocessors（多处理器），12
architectures（体系结构），23
board-level（板级），948
cache-coherent（高速缓存一致性），314，926～927
chip-level（芯片级），948
dancehall organization（舞池式组织结构），459（fig.）
directory-based（基于目录的），553～677
distributed-memory organization（分布式存储器结构），458（fig.）
extended memory hierarchies（扩展的存储器层次结构），138～139，271（fig.）
FLASH（闪存）640，648
HAL S1（HAL S1），643～645
hierarchical bus-based（基于层次式总线的），661（fig.）
hierarchical ring-based（基于环的），665（fig.）
as memory hierarchy（存储器层次结构），140
performance measures（性能度量），202～204
ring-connected（环型连接），442，443（fig.）
scalable（可伸缩的），453～551
shared memory（共享存储器），23，28，29，269～376
simulations with（用……模拟），200，233～234
snooping cache-coherent（侦听式高速缓存一致性），278（fig.）
symmetric（对称），32～34，269～271
use goal for（应用目标），121
Multiprog（多道程序），244，252
miss rates（扑空率），323（fig.）
spatial locality（空间局部性），323～324
statistic gathering（统计收集），254
traffic as function of cache block size for（作为高速缓存块大小的函数的流量），327（fig.）
workload（工作负载），313，323
multiprogrammed workloads（多道程序工作负载），218
multistage interconnect（多级互连），30～31，34
multithreading（多线程的），155
blocked（阻塞的），898～902，914～917
execution time breakdowns（执行时间分解），913（fig.）
future of（……的未来），939
hardware-supported（硬件支持的），896～897
integrating with multiple-issue processors（与多指令发射处理器的集成），920～922
interleaving（交错），902～908，917～920
latency hiding（时延隐藏），914
message passing（消息传递），850～851
parallelism（并行性），840
performance benefits（性能益处），910～914
relaxed memory consistency and（放松的内存同一性），914
shared address space（共享地址空间），896～922
simultaneous（同时的），920～922

support（支持），851
techniques（技术），898～910。参见 latency tolerance; threads
Munin system（Munin 系统），738
mutual exclusion（互斥），57，103，113，124，188，337～351
algorithm（算法），688
implementation of（实现），335
message-passing model（消息传递模型），538
operation implementation（操作实现），337
shared address space（共享地址空间），538 参见 synchronization
Myricom network（Myricom 网络），826～827
communication assist（通信辅助部件），826
packets（数据包），827
switches（交换机、交换开关），827
Myrinet（Myrinet），514～515
case study（案例研究），516～518
communication endpoints（通信端点），518
DMA engines（DMA 引擎），517
Lanai processor（Lanai 处理器），516
NIC（网络接口卡（NIC）），516～518
NOW organization（工作站网络（NOW）组织结构），517（fig.）

N

naming（命名），55～56
Barnes-Hut application（Barnes-Hut 应用程序），184
message-passing model（消息传递模型），56
Ocean application（Ocean 应用程序），184
Raytrace application（Raytrace 应用程序），184
shared address model（共享地址模型），55，184
narrow links（窄带链路），764～765
NAS benchmarks（NAS 基准测试程序），966～967
BT（BT），532～537
kernels（内核），967
LU（LU），532，533，537～538
paper-and-pencil（纸和笔），966
parallelism costs（并行代价），537。参见 benchmarks
n-body problems（n－体问题），76，80（fig.）
nCUBE（nCUBE），68，463
nCUBE/2（nCUBE/2），463～464
case study（案例研究），488～490
DMA controllers of（DMA 控制器），490
input channels（输入信道），489
machine organization（机器组织结构），464（fig.）
network interface organization（网络接口组织结构），489（fig.）
node chip（节点芯片），463
nearest-neighbor sharing（最近相邻者共享），588
negative acknowledgment（NACK）（否认（NACK）），591
input buffer space availability（输入缓冲空间可用性），484
lines（线），400
NUMA-Q and（NUMA-Q），632
SGI Challenge（SGI Challenge），404
SGI Origin2000（SGI Origin2000），597
network design（网络设计），749～830
case studies（案例研究），818～827
trade-offs（权衡），749～750，779～788
network interface card（NIC）（网络接口卡（NIC））
IBM SP-2（IBM SP-2），41
Memory Channel（存储器通道），519～520
Myrinet（Myrinet），516～518
network interface（NI）（网络接口（NI）），493，525，751，768
input/output ports（输入/输出端口），768
Intel Paragon（Intel 的 Paragon 机），499，500
packet formatting（数据包格式化），768
SGI Origin2000（SGI Origin2000），618
network（s）（网络）
back pressure and（反向压力），483
bandwidth（带宽），761～764
Benes（Benes），776，777（fig.）
buffering（缓冲），760
capacity（容量），846
closed system（封闭系统），760
communication performance（通信性能），755～764
contention（争用），154
cost（成本，开销），782
deadlock-free（免死锁），483
definitions（定义），750～755
delay（延迟），61，62
diameter（直径），753
direct（直接），489
flow control mechanism（流控机制），753
fully connected（全连接的），768～769
"hot spots"（"热点"），760
latency（时延），755～761
open system（开放系统），763
organizational structure（组织结构），764～768
performance modeling（性能建模），763～764
routing algorithm（路由算法），752～753
routing distance（路由距离），753
saturation point（饱和点），762

switches（交换开关），457
switching strategy（交换策略），753
wiring complexity（连线复杂性），764。参见 network topologies
networks of workstations（NOWs）（工作站网络（NOWs）），513～521
communication latency（通信时延），526
convergence（收敛），513
hardware primitives（硬件原语），515
Myrinet SBUS Lanai case study（Myrinet SBUS Lanai 案例研究），516～518
node operating systems（节点操作系统），526
organization using Myrinet（采用 Myrinet 的组织结构），517（fig.）
PCI Memory Channel case study（PCI 存储器通道案例研究），518～521
receive overhead（接收开销），525
send overhead（发送开销），525
speedup（加速比），532
network topologies（网络拓扑），752，768～778
butterfly（蝶型的），774～777
choice of（选择），761
fully connected（全连接的），768～769
hypercube（超立方体），778
latency and（时延），756
linear array（线性数组），769
multidimensional mesh（多维网格），769～772
ring（环），769
trade-offs（权衡），779～788
tree（树），772～774
network transactions（网络事务），455，468
action（动作），472
bus transactions vs.（总线事务与……比较），469
completion detection（结束检测），472
deadlock avoidance（死锁避免），473
delivery guarantees（递交保证），473
destination name and routing（目的地名字和路由），472
effects（效应），469
flow with symmetric communication（对称通信流），499（fig.）
format（格式），471，485
input buffering（输入缓冲），472
interpreting（解释），490
latencies（时延），475
media arbitration（介质仲裁），472
Meiko CS-2（Meiko CS-2），503，505
Monsoon design（Monsoon 设计），496
output buffering（输出缓冲），472
overhead（开销），584
parallel software and（并行软件），522～526
performance（性能），522～526
point-to-point（点对点），555
primitive（原语），469（fig.），470～473
processing in large-scale architecture（大规模体系结构中的处理），485（fig.）
protection（保护），471
transaction ordering（事务次序），473
uncoupled source/destination（去耦的源/目的地），471
user/system flag（用户/系统标记），491
next PC（NPC）（下一PC（NPC）），918～919
node-level protocols（节点级协议），752
nodes（节点），457
dirty（脏），560，562
exclusive（独占的），560
head（头），624
home（宿主），560
Intel Paragon（Intel 的 Paragon 机），499
linear array of（线性数组），769
local（本地的），560
NUMA-Q（NUMA-Q），623
outer protocol（外部协议），556
owner（拥有者），560
pending lists and（未决表），633
ring of（……的环），769
routing distance between（……之间的路由距离），460
SGI Origin2000（SGI Origin2000），597
SMP（对称多处理器（SMP）），467，503，721
tail（尾），624
workstations as（作为……的工作站），467
node-to-network interfaces（节点－网络接口），156，454
physical DMA（物理 DMA），486～488
user-level access（用户级访问），491～493
nonadaptive routing（非自适应路由），参见 deterministic routing
nonbinding prefetch（非绑定的预取），880
nonblocking asynchronous SEND operation（非阻塞异步 SEND 操作），115
nonblocking synchronization（非阻塞同步），351
nonuniform memory access（NUMA）approach（非一致的存储器访问（NUMA）方法），34，549
scalable shared memory multiprocessor（可伸缩的共享存储器多处理器），35（fig.）

shared memory context（共享内存器上下文），70
use of（对……的运用），36
NUMA-Q（NUMA-Q），556，622～645，700
average remote miss latency components（平均的远程扑空时延成份），643（fig.）
bandwidth（带宽），641
case study（案例研究），622～645
coherence controller（一致性控制器），731
comparison case study（比较性案例研究），643～645
correctness issues（正确性问题），632～634
DataPump（DataPump），641，642
deadlock（死锁），633
directory protocol（目录协议），624
directory state diagrams（目录状态图），670（fig.）
directory structure（目录结构），624
error handling（出错处理），633～634
hardware overview（硬件概述），635～637
interquad I/O transfers（四处理器之间的 I/O 传送），637
I/O subsystem（I/O 子系统），637（fig.）
IQ-Link board（IQ-链路板），622，623，643
IQ-Link board block diagram（IQ－链路板框图），636（fig.）
IQ-Link implementation（IQ-链路实现），639～641
latency（时延），641～642
livelock（活锁），633
memory consistency model（存储同一性模型），633
microbenchmark characteristics（微基准测试程序特征），642（fig.）
multiprocessor block diagram（多处理器框图），623（fig.）
NACKs and（NACKs），632
performance characteristics（性能特征），641～643
processing nodes（处理节点），623
processor consistency model（处理器同一性模型），698
protocol extensions（协议扩展），634～635
protocol interactions with SMP node（与 SMP 节点的协议交互），637～639
read request handling（读请求处理），626～628
remote cache（远程高速缓存），623～624，700
SCI ring（SCI 环），824
SCI specification（SCI 规范），822
SCLIC（SCLIC），635，638，639～640
serialization of operations to given location（对特定单元操作的串行化），632～633
serialization of writes（写串行化），673～674
sharing list（共享表），624，625（fig.），628
SMPs（SMPs），623
SMPs as nodes（以 SMP 作为节点），588
starvation（饮饿），633
states（状态），624～626
third-party commodity hardware（第三方商用硬件），622
TPC-B benchmark（TPC-B 基准测试程序），642，643
TPC-D benchmark（TPC-D 基准测试程序），642，643
workload characteristics（工作负载特征），642（fig.）
write-back request handling（回写请求处理），630～632
write request handling（写请求处理），628～630。参见 SGI Origin2000

O

object-based coherence（基于对象的一致性），721～723
advantages/disadvantages（优势/劣势），722
synchronization events（同步事件），723。参见 coherence
occupancy（占用度），61
Ocean application（Ocean 应用程序），76，77～78，161～166
assignment（分配），83，161～163
block transfer benefits in（块传输的好处），861（fig.）
capacity misses（容量型扑空），324
concurrency（并发性），78
decomposition（分解），161～163
dependences among grid computations（格计算中的相关性），162（fig.）
equation solver kernel（方程求解器内核），92～116
execution time breakdown（执行时间分解），166（fig.）
grid arrays（格数组），90
grid computations（格计算），162（fig.），163，164
horizontal cross sections（水平断面），78（fig.）
invalidation pattern（作废模式），574（fig.）
invalidations（作废），576
iterative nearest-neighbor grid computations（迭代的最近邻格计算），891
mapping（映射），165
miss rates（扑空率），321（fig.）
models（模型），77
naming（命名），184
nearest-neighbor sharing（最近相邻共享），588
on Origin2000 machine（在 Origin2000 机器上），149
orchestration（协调），163～165
owner computes arrangement（拥有者计算方案），90
prefetching remote data in（预取远程数据），892（fig.）
process streams（进程流），261
scaling（扩放，伸缩），432
sequential algorithm（串行算法），161
spatial locality（空间局部性），163～164，320，321

speedup（加速比），431
speedup on SGI Origin2000（在 SGI Origin2000 上的加速比），621（fig.）
summary（总结），165
synchronization（同步），165
tasks（任务），82
temporal locality（空间局部性），164
time-step phases（时间步阶段），162（fig.）
traffic vs. local cache（流量与本地高速缓存的关系），582（fig.），583（fig.）
traffic vs. number of processors（流量与处理器数目的关系），580（fig.），581（fig.）
two-dimensional grids（二维格），77
working sets（工作集），164。参见 case studies
offered bandwidth（提供的带宽），762
one-block lookahead（OBL）schemes（一块先行（OBL）策略），882
on-line bipartite matching（在线双向匹配），810
on-line transaction processing（OLTP）benchmarks（在线事务处理（OLTP）基准测试程序），10
open systems（开放系统），763
operations
atomic，implementing（操作原子性，实现），391～393，652
competing（竞争），695
conflicting（冲突），695
critical section（临界区），105
floating-point（浮点），101，209
global communication（全局通信），817～818
interleaving，prohibiting（交错，禁止），105
lock-based（基于锁的），351
message passing（消息传递），43，56
out-of-order processing of（乱序处理），290
parallel prefix（并行前缀），546～547
programming model（编程模型），56
RECEIVE（接收），112，114～115
release（释放），692，733～734
SEND（发送），112，114～115
serial order of（串行次序），276
shared address model（共享地址模型），56
optimization（优化），219～220
algorithmic（算法的），219
data layout，distribution，alignment（数据布局），220
data structuring（数据结构化），219
orchestration（协调），220
orchestration（协调），83，89
Barnes-Hut application（Barnes-Hut 应用程序），170～173
choices（选择），89
Data Mining application（数据挖掘应用程序），181～182
under data parallel model（在数据并行模型下），99～101
under message-passing model（在消息传递模型下），108～116
Ocean application（Ocean 应用程序），163～165
optimization（优化），220
for performance（性能），142～156
performance goals（性能目标），89
Raytrace application（Raytrace 应用程序），176～177
under shared address space model（在共享地址空间模型下），101～108。参见 parallelization process
ordering（定序），56～57
bus transaction properties（总线事务属性），473
channel（信道），794（fig.）
message-passing model（消息传递模型），56
network transaction（网络事务），473
partial store（PSO）（部分存储（PSO）），686，689，865
red-black（红－黑），96，97（fig.）
relaxed memory（RMO）（放松的存储器（RMO）），686，690，693，699，866
request-response（请求－应答），409
shared address model（共享地址模型），57
static（静态的），445
synchronization and（同步），57
total store（TSO）（全部存储（TSO）），686，865，866，867
weak（WO）（弱序（WO）），686～687，690～691，699，866，867
Orion bus interface controller（OBIC）（Orion 总线接口控制器（OBIC）），635，638，639
orthogonal recursive bisection（ORB）（正交递归对分（ORB）），170
Barnes-Hut（Barnes-Hut），171（fig.）
hypercube technology mapping（超立方体技术映射），173
outer protocol（外部协议），556
output array（输出数组），248
output scheduling（输出调度），808～810
algorithm（算法），808
control structure（控制结构），809（fig.）
overflow（溢出）
broadcast scheme（广播方案），655
coarse vector scheme（粗向量方案），656～657
dynamic pointers scheme（动态指针方案），658
methods for reducing directory width（减少目录宽度的方法），655～658

no broadcast scheme（无广播方案），655～656
pointers（指针），658
software scheme（软件方案），657～658
overhead（开销），61
block transfer（块传输），242
cache block（高速缓存块），242
communication（通信），186～187，487
context switch（现场切换），901
directory memory（目录存储器），667
endpoint（端点），183，846
hierarchical directory storage（层次式目录存储），667
interleaved scheme（交错方案），909
latency tolerance and（时延包容），845
message（消息），755
network transaction（网络事务），584
per-message, amortized（每条消息，分摊），857
receive（接收），522，523，525
reducing（减少），151～152，659
send（发送），522，523～525
single-thread support scheme（单线程支持方案），909
thread-switching（线程交换），898～899
unready thread cost（未就绪线程开销），909（fig.）
overlap（重叠），63
overlapping transactions（重叠事务），586
owner computes arrangement（拥有者计算方案），90
owner node（拥有者节点），560

P

packet buffers（数据包缓冲），820
packets（数据包），751
CRAY T3D formats（CRAY T3D 格式），819（fig.）
crossing channels（交叉信道），756
envelope（封装），756
format（格式），754（fig.）
formatting（格式化），768
fragmentation（碎片），755
full buffers and（全缓冲），759
header（头），754，792
IBM SP network（IBM SP 网络），821
Myricom network（Myricom 网络），827
parts（部件），754
payload（有效负载），754
SCI. formats（SCI，格式），825（fig.）
trailer（尾），754，766
packet switching（包交换），758，759
page-based access control（基于页的访问控制），709～721
page fault handlers（缺页处理例程），709
page granularity（页粒度），725
page migration（页面迁移）
automatic（自动的），729
explicit（显式的），729
SGI Origin2000 and（SGI Origin2000），610～611，729
page table entries（PTEs）（页表项（PTEs））
locking（锁定），440
modifying（修改），439
pairwise sharing（成对共享），624
PAL code（可编程阵列逻辑代码），526
parallel architectures（并行体系结构）
abstraction layers（抽象层），27（fig.），52
commercial computing and（商用计算），9～10
convergence of（会聚），25～52
cost-performance trade-offs and（性能成本权衡），4
dataflow（数据流），47～49
data parallel processing（数据并行处理），44～47
efficiency（效率），230
engineering component（工程元素），64
evolutionary scenario（演变情况），937～940
evolution of（演变），2
form and function elements（形式和功能元素），1
future of（……的未来），935～961
generic（一般的），50～52
layers of（……的层），469（fig.）
learning process（学习过程），2～3
message passing（消息传递），37～44
potential breakthroughs（潜在的突破），944～955
reasons for（……的原因），4～25
role in information processing（在信息处理过程中的角色），2
scaling in（……方面的伸缩），467～468
scientific computing and（科学计算），8
shared address space（共享地址空间），28～37
systolic（脉动的），49～50
technology and（技术），936～955
parallel computers（并行计算机），1
cost-performance trade-offs（性能成本权衡），2
hardware/software responsibilities in（硬件/软件责任），2
market pyramid（市场金字塔），937（fig.）
operating systems（操作系统），959
performance characteristics（性能特征），202
programming model realization（编程模型实现），468
parallelism（并行性），14
attractiveness of（吸引力），5

bit-level（位级），15
computer architecture and（计算机体系结构），1
costs（成本），537
data（数据），124，125
exploiting（发掘，利用），57
freedom of（……的自由），75～76
function（功能），124
hierarchical（层次结构的），193
importance of（……的重要性），12
instruction-level（指令级），15～17
learning curve（学习曲线），9
levels of（……级别），193，194（fig.）
locality and（局部性），14
multithreading（多线程），840
1960s innovations（60年代的革新），67
performance improvement and（性能改进），6
pipeline（流水线），119，120，124，125
process-level（进程级），19
thread-level（线程级），17～19
parallelization process（并行化过程），81～92
data（数据），90～91
example program（范例程序），92～116
goals of（……的目标），82，91～92
gradual（逐渐的），366
steps in（步骤），82～90
parallel languages（并行语言），958
parallel prefix operation（并行前缀操作），546～547
downward sweep（向下扫描），547（fig.）
performance（性能），546
upward sweep（向上扫描），546（fig.）
parallel programming（并行程序设计）
languages, immaturity of（语言，不成熟性），200
models（模型），26
transition to（向……的转移），12
parallel programs（并行程序），75～120
algorithm designers and（算法设计者），75
architects and（体系结构设计者），75
case studies（案例研究），76～81
debugging（调试），960
deterministic（确定的），96
example parallelization of（并行化的实例），92～116
immaturity of（不成熟性），200
load imbalanced（负载不平衡），88
programmers and（程序员），75
speedup of（加速比），122
statistics about（关于……的统计），255
Parallel Random Access Memory（PRAM）model（并行随机访问存储器（PRAM）模型），136～137，191，718，908
algorithm development for（算法开发），137
speedups on（加速比），254
usefulness of（有用性），191
parallel software（并行软件），522～538
application-level performance（应用程序级性能），531～538
categories（类别），955
directory-based cache coherence implications（基于目录的高速缓存一致性实现），652～655
evolutionary scenario（演变情况），955～960
hardware/software trade-offs and（硬件/软件权衡），729～730
message-passing operations（消息传递操作），528～531
network transaction performance（网络事务性能），522～526
potential breakthroughs（潜在突破），961
shared address space operations（共享地址空间操作），527～528
walls（墙），960～961
parallel vector processors（PVPs）（并行向量处理器），23，24（fig.）
parallel virtual machine（PVM）（并行虚拟机（PVM）），52
parameter space（参数空间），238～243
associativity（相联性），241
cache block size（高速缓存块尺寸），241
cache/replication size（高速缓存/复制尺寸），239～240
goals（目标），238
number of processors（处理器数目），238～239
performance parameters of communication architecture（通信体系结构的性能参数），242
problem size（问题大小），238～239
PARKBENCH（PARKBENCH），967～968
partial store ordering（PSO）（部分存储序（PSO）），686，689，865
partitioning（划分，分区），83，123～137
Barnes-Hut application（Barnes-Hut 应用程序），169，171（fig.）
dynamic（动态的），126，127
goal of（……的目标），132，135～136
image plan（图像平面），177（fig.）
into contiguous subdomain（进入连续的子域），135
principle（原理），359
profiling-based semistatic（基于性态的半静态），169
repartitioning（重划分），134
specifying（指定），101
static block（静态块），176

techniques（技术），136
trade-offs（权衡），135
patches（补丁），249～250
interaction list（交互表），250，253（fig.）
light transfer between（……之间的轻量传输），250
payload（有效负载），754
PCI Memory Channel（PCI存储器信道），518～521
communication assist（通信辅助部件），518
DMA engine（DMA引擎），520
interface（接口），719
NIC（网络接口卡（NIC）），519～520
page control table（PCT）（页面控制表（PCT）），520
shared physical address space（共享物理地址空间），518
write doubling mechanism（写加倍机制），718
PC（problem-constrained）scaling（问题约束（PC）的伸缩），207
communication-to-computation ratio（通信与计算之比），212
concurrency（并发性），212
memory requirements（存储器需求），212
problem size（问题规模），208
spatial locality（空间局部性），213
speedup（加速比），208，213
synchronization（同步），213
temporal locality（时间局部性），213
working set（工作集），227。参见 scaling；scaling models
pending lists（未决表）
distributed（分布的），634
SCI（SCI），629（fig.）
total number for nodes（节点总数），633
pending states（未决状态），590
Pentium。参见 Intel Pentium Pro
Perfect Club benchmarks（Perfect Club基准测试程序），968
perfect-memory execution time（完美的存储器执行时间），210
performance（性能），59～63，121～197
absolute（绝对的），202，203，228～229
application parameter impact on（应用程序参数对……的影响），202
array-based lock（基于数组的锁），350，651
block data transfer benefits（块数据传输的好处），856～863
COMA，trade-offs（COMA，权衡），702～703
communication（通信），755～764
components（成分），59
cost trade-off and（成本权衡），4
data transfer time（数据传送时间），60～61
directory protocol（目录协议），584～589，645～648
example（例子），59～60
factors（因素），156～159
floating-point（浮点），13
hierarchical coherence and（层次式一致性），668～669
isolation with microbenchmarks（用微基准测试程序来分离），215～216
lock（锁），340～342，348～350
microprocessor（微处理器），4，5（fig.），13
modeling（建模），189，763～764
MPPs（大规模并行处理器（MPPs）），24
multiprocessor，measures（多处理器，度量），202～204
multithreading（多线程），910～914
NUMA-Q（NUMA-Q），641～643
occupancy and（占用度），61
orchestration for（协调），142～156
overhead and（开销），61
partitioning for（为……分区），123～137
percentage improvement in（在……方面提高的百分比），231
processor，growth（处理器，增长），4，5（fig.）
programming（程序设计），121～197
relaxed consistency models and（放松的同一性模型），700
SGI Origin2000（SGI Origin2000），618～622
supercomputer（超级计算机），24
SVM（SVM），720～721
synchronization algorithm（同步算法），649～651
trade-offs（权衡），122，123
uniprocessor，supercomputer（单处理器），22（fig.）
workstation（工作站），71（fig.）
performance metrics（性能指标），228～231
absolute performance（绝对性能），228～229
choosing（选择），228～231
execution time（执行时间），157～159，179，203
percentage improvement（提高的百分比），231
problem size（问题规模），230～231
processing rate（处理率），230
rate-based（基于比率的），262
speedup（加速比）。参见 speedup
utilization（利用），230，262
perimeter-to-area ratio（周长－面积比例），132，133（fig.）
peripheral component interface（PCI）buses（外围部件接口总线），635，636，637
per-processor heaps（每处理器的堆），363
phits（物理单元），752
flits vs.（与流控单元的关系），753
latency vs.（与时延的关系），788（fig.）
physical DMA（物理DMA），486～491

blind, communication assist for（盲的，通信辅助部件），487（fig.）
communication abstraction implementation（通信抽象实现），488
LAN interfaces（局域网接口），490～491
nCUBE/2（nCUBE/2），488～490
node-to-network interface（节点到网络的接口），486～488。参见 direct memory access（DMA）
physical protocol（物理协议），751
physical scaling（物理上的扩放），462～468
board-level integration（板级集成），465～466
chip-level integration（芯片级集成），463～464
system-level integration（系统级集成），466～467。参见 scaling
pipelined latency（流水时延），618，911
pipelined parallelism（流水并行性），119，120，124～125
availability（可用性），125
example（例子），125
pipeline stalls（流水线暂停），901
pipelining（流水）
cost（成本），900
late miss detection in（晚期扑空检测），901（fig.）
multiple memory operations（多个存储器操作），863
strategy（策略），400～401
of writes（写），866
pivot element（主元素），118
pivot row（主行），118，119
"platform pyramid"（"平台金字塔"），6
PLUS system（PLUS 系统），718
P-node trees（P－节点树），544
point-to-point bandwidth（点对点带宽），151，845～846
point-to-point events（点对点事件），538
point-to-point network transactions（点对点网络事务），555
point-to-point synchronization（点对点同步），96，172
data dependences and（数据依赖），130
event（事件），107（fig.）117，352～353
full-empty bits and（满-空位），352～353
hardware support（硬件支持），352～353
interrupts（中断），353
software algorithms（软件算法），352
polling（轮询），482
PowerChannel-2 I/O subsystem（PowerChannel-2 输入/输出子系统），422～424
HIO interface chips（*HIO 接口芯片*），422
organization（组织结构），423（fig.）。参见 SGI Challenge
Powerpath-2 system bus（Powerpath-2 系统总线），417～420
acknowledge cycle（应答周期），419
address/data buses（地址/数据总线），418
data resource ID line（数据资源 ID 线），419
distributed arbitration scheme（分布式仲裁方案），418～419
four-cycle sequence（四周期序列），420
signals（信号），417
state transition diagram（状态转换图），418（fig.）
timing diagram（时序图），419（fig.）
urgent requests（紧急请求），418。参见 SGI Challenge
Precommunication（预通信），155，838～839
latency hiding through（通过……的时延隐藏），849（fig.）
long-latency events and（长时延事件），839
message passing（消息传递），848～850
with multiple-issue, dramatically scheduled processors（多指令发射、动态调度的处理器），895
performance benefits（性能收益），891～896
prefetching（预取），850，878
in receiver-initiated communication（接收者发起的通信），839
relaxed consistency comparison（放松的同一性比较），895～896
sender-initiated（发送者发起的），890～891
shared address space（共享地址空间），877～896
with single-issue, statically scheduled processors（单指令发射、静态调度的处理器），891～894
summary（总结），896
without caching of shared data（不高速缓存的共享数据），877～879，参见 latency tolerance
predicted PC（预测的 PC），919
prefetch buffers（预取缓冲），878
prefetching（预取），434，850
analysis（分析），880，881
binding（绑定），880
block（块），880
compiler-generated, performance benefits（编译器产生的性能收益），893，（fig.）
concepts（概念），880～881
contention and（竞争），895
coverage（复盖），881
effectiveness（有效性），895
effects for SC（SC 的效果），874
hardware-controlled（硬件控制的），880，881～883
hardware-controlled vs. software controlled（硬件控制与软件控制的比较），888～890
implementation issues（与实现有关的问题），896
ineffectiveness（无效性），872

nonbinding（非绑定），880
on local accesses（本地访问），891
with ownerships（拥有权），888（fig.）
precommunication（预通信），850，878
relaxed memory consistency comparison（放松的存储器同一性比较），895～896
remote data（远程数据），891
remote data，performance benefits（远程数据，性能收益），892（fig.）
reorder buffer operations（重排序缓冲器操作），876
scheduling（调度），880，883
selective，benefits of（选择，……的益处），894（fig.）
in shared address space（在共享地址空间中），879（fig.）
with single processor（对于单处理器），884～887
software-controlled（软件控制的），880，884
timing（定时），881
unnecessary（不必要的），881，884
presence bits（存在位），496，560
Princeton SHRIMP designs（普林斯顿 SHRIMP 设计），521
private address space（私有地址空间），112
problem sizes（问题规模），206
artifactual communication and（附加通信），223
choosing（选择），238～239
choosing based on working sets（基于工作集的选择），224（fig.）
effect of（……的效果），227（fig.）
impact on spatial locality behavior（对空间局部性行为的影响），225（fig.）
metric（指标），230～231
PC scaling（问题约束的扩放），208
range determination（范围判定），221
TC scaling（时间约束的扩放），208
proceeding past communication in same thread（在同一线程越过通信），839～840
buffering and（缓冲），863
message passing（消息传递），850
pipelining and（流水），863
reads（读），868～876
in receiver-initiated communication（在接收者发起的通信中），840
shared address space（共享地址空间），863～877
summary（总结），876～877
writes（写），864～868，参见 latency tolerance
proceeding past reads（越过读操作），868～876
dynamic scheduling（动态调度），870
enhancing（提高），871～872
performance impact（性能影响），873～876
release consistency（释放同一性），871
sequential consistency（顺序同一性），871
speculative execution（投机性执行），870～871
speculative reads（投机性读），872
proceeding past writes（越过写操作），864～868
cache size/cache block size effects on（高速缓存尺寸/高速缓存块尺寸对……的影响），869（fig.）
performance benefits（性能收益），867（fig.）
performance impact（性能影响），865～868
write buffer and（写缓冲），867～868
processes（进程），29，83
communication between（之间的通信），59
descheduling（反调度），233
execution intervals（执行间隔），736
pinning（探询），89
relationships of（……的关系），84（fig.）
serialization of（串行化），124
shared memory machine（共享存储器的机器），49
processing elements（PEs）（处理单元），44～45
condition flag（条件标记），45
systolic architecture（脉动体系结构），49
"virtualizing"（"虚拟化"），46
processing microbenchmarks（处理微基准测试程序），215
processing rate（处理速率），230
process-level parallelism（进程级并行），19
processor-and-memory（PAM）（处理器和存储器（PAM）），946，950，954，955
processor-cache handshake（处理器－高速缓存握手），388
processor consistency（PC）（处理器同一性（PC）），686，866，867
comparison（比较），688（fig.）
model（模型），698
write ordering preservation（写次序保持），687，参见 system specification
processor-in-memory（PIM）（存储器内置处理器（PIM）），951，952（fig.）
processor interface（PI）（处理器接口（PI）），615～617
physical FIFO（物理先进先出缓冲（FIFO）），615
request numbers（请求号），617
shielding（屏蔽），617，参见 SGI Origin2000
processors（处理器）
arrays，development of（数组，……的发展），45
building blocks（构造块），4
bus，assist，interface sharing（总线，辅助部件，接口共享），588

in bus-based shared memory microprocessors（基于总线的共享存储微处理器），20
dynamically scheduled（动态调度的），895，922
HP PA-8000（HP PA-8000），684
initiator（启动程序），440
massively parallel（MPPs）（大规模并行（MPPs）），23
message-passing machine（消息传递型机器），42
MIPS R10000（MIPS R10000），597，684，868
number，choosing（号码，选择），238～239
number，effect of（号码，……的效果），227（fig.）
parallel vector（PVPs）（并行向量（PVPs）），23，24（fig.）
performance growth（性能增长），4，5（fig.）
relationships of（……的关系），84（fig.）
statically scheduled（静态调度的），891～894
status word（状态字），915
superscalar（超标量），895
trust between（……之间的信任），453
utilization（利用），230
vector（向量），21～22
program counters（PCs）（程序计数器），915～916，918～919
blocked multithreading（阻塞的多线程），915～916
bus（总线），919（fig.）
chain（链），915，918
exception（EPCs）（异常），915～916，918～919
interleaved multithreading（交错多线程），918～919
managing（管理），916
next（NPC）（下一程序计数器值），918～919
predicted（预测），919
programmer's interface（程序员接口），686
access labels（存取标号），697
consistency model（同一性模型），694
relaxed models at（放松模型），699
synchronization event identification（同步事件识别），695，参见 relaxed memory consistency models
programming languages（程序语言），682
High Performance Fortran（高性能 Fortran 语言），723
Jade（Jade），723
labeling support（标记支持），729
parallel，immaturity of（并行，不成熟），200
Split-C（Split-C），724
variable declaration attribute（变量声明的属性），697
programming models（编程模型，程序设计模型），26
abstraction（抽象），28
Active Messages（主动消息），481～482
challenges（挑战），482～485
data parallel（数据并行），44～47
differences in（差异），183
implementation of（实现），183
implications for（蕴涵），182～190
message passing（消息传递），37～44，476～481
naming（命名），55～56
operations（操作），56
ordering（排序），56～57
requirements（需求），53～57
shared address space（共享地址空间），37，473～476
shared memory processor（共享存储型处理器），29
programming performance（编程性能），121～197，960
program orders（程序操作序），687～694
relaxing all（全部放松），689～694
write-to-read，relaxing（写到读，放松），687～689
write-to-write，relaxing（写到写，放松），689
properly labeled model（适当标号的模型），695
protocol design trade-offs（协议设计的折中），305～334
bandwidth requirement（带宽需求），307～311
in cache block size（按照缓存块尺寸），313～319
decisions（决策），306
impact of（……的影响），311～313
methodology（方法），306～307
per-processor bandwidth（每个处理器的带宽）
requirements（需求），312（fig.）
update-based vs. invalidationbased and（基于更新方式与基于作废方式的比较），329～334
protocols（协议）
home-based（基于宿主的），717
hybrid（混合的），330～332
inner（内部的），556
invalidation-based（fig.）（基于作废的（图）），278，280，283（fig.）
multiple writer（多写入者），716～718
outer（外部的），556
processing（处理），708
space of，for write-back caches，（空间，用于回写的高速缓存），283
update-based（基于更新的），278，292

Q

queue-on-lock-bit（在锁位上的队列），（QOLB）
synchronization（同步），626，651
queues（队列）
centralized（集中式的），128
communication（通信），498
distributed（分布式的），128

inside multilevel cache hierarchy（在多层高速缓存层次结构内部），411（fig.）
structures（结构），413
task（任务），128，129（fig.）
queuing locks（排队锁），651

R

Radiosity（Radiosity 应用），244，249～252
barrier synchronization（栅障同步），252
BSP tree（BSP 树），250
interaction list（交互链表），250，253（fig.）
invalidation pattern（作废模式），575（fig.）
invalidations（作废），577
load imbalance（负载不平衡），256
miss rates（扑空率），320（fig.）
patches（补丁），249～250
quadtrees（四叉树），251（fig.）
spatial locality（空间局部性），322
speedup（加速比），431
speedup on SGI Origin2000（SGI Origin2000 的加速比），621（fig.）
traffic vs. local cache（流量与本地高速缓存的关系），582（fig.）
traffic vs. number of processors（流量与处理器个数的关系），580（fig.）。参见 workload case studies
Radix（Radix 应用），244，248～249，267
bandwidth requirements（带宽需求），327
digits（数位），248
dynamic scheduling（动态调度），875（fig.）
false-sharing effect in（假共享效应），323
invalidation pattern（作废模式），574（fig.）
invalidations（作废），576
kernel sorting（内核排序），430
keys（键字），248
local keys（本地键字），248
miss rates（扑空率），321（fig.）
permutation step（置换步骤），249（fig.）
process streams（处理流），261
sharing behavior（共享行为），323
sorting（排序），249（fig.）
sorting kernel（排序内核），576
speculative execution（投机执行），875（fig.）
speedup（加速比），256，431
speedup on SGI Origin2000（SGI Origin200 的加速比），621（fig.）
traffic vs. local cache（流量与本地高速缓存的关系），582（fig.），583（fig.）
traffic vs. number of processors（流量与处理器个数的关系），580（fig.），581（fig.）。参见 workload case studies
RAM（随机访问存储器）
address map（地址映像），423
arrays（阵列），635
Computational（可计算的），951
dual-ported（双端口的），382
ray-oriented approach（面向射线的方法），175
Raytrace application（Raytrace 应用），77，79～80，174～178
artifactual communication（人为通信），176
assignment（分配），175～176
capacity misses（容量型扑空），324
decomposition（分解），175～176
dynamic tasking（动态任务分配），127
execution time breakdown for（执行时间分解），179
granularity（粒度），177
HUG（层次均匀格），174
image plane partitioning（图象平面分割），177（fig.）
invalidation pattern（作废模式），574（fig.）
invalidations（作废），577
mapping（映射），178
miss rates（扑空率），321（fig.）
naming（命名），184
orchestration（协调），176～177
ray-oriented approach（面向射线的方法），175
rays（射线），174，175，176
scene-oriented approach（面向景物的方法），175
sequential algorithm（串行算法），175
serialization（串行化），256
spatial locality（空间局部性），176～177
speedup（加速比），431
speedup on SGI Origin2000（SGI Origin2000 的加速比），621（fig.）
subspace（子空间），175
summary（总结），178
synchronization（同步），177
tasks（任务），82
temporal locality（时间局部性），177
traffic vs. local cache（流量与本地高速缓存的关系），582（fig.），583（fig.）
traffic vs. number of processors（流量与处理器个数的关系），580（fig.），581（fig.）
working sets（工作集），177
ray tracing（光线跟踪），79～80
read exclusive（读互斥），292

bus transactions（总线事务），292
cache coherence and（高速缓存一致性），294
writing cache and（写高速缓存），297
read misses（读扑空），662，744
blocking（阻塞），864，877
proceeding past（越过），868～876
read-only sharing（只读共享），572，588
read requests（读请求）
buffers（RRBs）（缓冲），615
NUMA-Q（NUMA-Q），626～628
SGI Origin2000（SGI Origin2000），599～601
read-to-own（读拥有）。参见 read exclusive
read-write communication（读写通信），852
ready threads（就绪线程），898
realistic scaling（真实的伸缩），266
RECEIVE operation（RECEIVE 操作），112
blocking asynchronous（阻塞式异步的），115
nonblocking asynchronous（非阻塞式异步的），115
semantics（语义），114～115
synchronous form（同步形式的），112，114
receive overhead（接收开销），522
comparison（比较），525
message breakdown（消息分解），523（fig.）
red-black ordering（红－黑序），96
advantages（优点），96
of equation solver（方程求解器），97（fig.）
REDUCE statement（REDUCE 语句）100（fig.），101
reducing（减少）
application miss rate（应用扑空率），325
artifactual communication（附加通信），142～150
capacity misses（容量型扑空），314
cold misses（冷启动扑空），314
communication（通信），123，131～135
concurrency（并发），99，101
conflict misses（冲突型扑空），314
contention（竞争），153～154
delay（延迟），152～153
extra work（额外工作），123，135～137
false sharing（伪共享），316，328，329，361（fig.）
overhead（开销），151～152
serialization（串行化），130～131
spatial interleaving（空间交错），360～362
true-sharing misses（真共享扑空），316
reference generators（访问生成器），233
reflective memory（反射式存储器），515
address space organization（地址空间的组织），519（fig.）
receive region（接收区域），518
transmit region（发送区域），518
register insertion rings（寄存器插入环），443，444
relaxed memory consistency models（放松的内存同一性模型），681～700
intuition behind（直觉），684，685（fig.）
multithreading and（多线程），914
performance and（性能），700
prefetching comparison（预取比较），895～896
programmer's interface（程序员接口），686，694～697
in real multiprocessor systems（在真实的多处理器系统中），698～700
single writer with consistency at acquire（在获取时具有同一性的单写入者），734～737
single writer with consistency at release（在释放时具有同一性的单写入者），733～734
software implementation（软件实现），732～739
solution components（解的成分），685～686
system specification（系统规范），685～694
translation mechanism（翻译机制），686，698
using（使用），711～716。参见 memory consistency models
relaxed memory ordering（RMO）（放松的存储器次序（RMO）），686，690，693，699，866
release-based consistency（基于释放的同一性），737～738
release consistency（RC）（释放同一性（RC）），687，690，691～692，699，866，867
acquire（获取），692
conservative（保守的），722（fig.）
eager（积极的），712，713（fig.）
hardware implementations（硬件实现），715
incomplete acquire operation in（未完成的获取操作），871
interface（接口），692
latency hiding under（隐藏的时延），871
lazy（惰性的），713（fig.），715，717，738，746
non-null pointer value（非空指针值），715～716
at programmer's interface（在程序员的接口），699
release（释放），692
weak ordering vs（与弱序相比较），691（fig.）。参见 system specification
release method（释放方法），335
release operation（释放操作），692，733～734
remote blocks（远程块），560
remote caches（远程高速缓存），623～624，660，662
access time，minimizing（访问时间，最小化），663
DRAM（DRAM），663，700
renaming（重命名），18

reorder buffers（重排序缓冲区），414，870，876
repartitioning（重划分），134
replication（复制），184～186
artifactual communication and（附加通信），140～142
communication and（通信），58～59
fine-grained（细粒度的），729
finite capacity（有限容量），140
limited capacity（有限制容量），679，680
local（本地的），140
management（管理），185～186，724
message-passing model（消息传递模型），185
shared address space（共享地址空间），185
shared address space without（不带复制的共享地址空间），723～724
size（尺寸），239～240
state（状态），914～915，917～918
reply forwarding（应答转发），585
outstanding requests and（未完成的请求），586
SGI Origin2000（SGI Origin2000），597，602
requests（请求），398
conflicting（冲突），398～399，402～403
IncPIO（IncPIO），424
outstanding（未完成的），399
urgent（紧急的），418
writes and（写），403。参见 split transaction bus
request tables（请求表），402
fully associative（全相联），409
requests entered into（进入的请求），405
resource-oriented properties（面向资源的属性），207
resources（资源）
contention for（争用），62
hot spot（热点），154
occupancy（占用），153
resource（资源），154
responders（响应者），382
response transaction（响应事务），398
ring access control mechanism（环访问控制机制），442～443
ring-connected multiprocessors（环形连接的多处理器），442，443
ringlets（小环），822
rings（环），441～444，555，769
advantages（优点），441～442
bandwidth on（带宽），443，444
broadcast media（广播介质），442
disadvantages（缺点），442
illustrated（说明的），770（fig.）
interface（接口），442
misses on（扑空），444
register insertion（寄存器插入），443，444
sequential consistency on（顺序同一性），441，444
serialization on（串行化），441，444
slotted（分槽的），443
snooping cache coherence on（侦听式高速缓存一致性），441
token-passing（令牌传递），442～443
uses（使用），769。参见 network topologies
RISC microprocessors（RISC 微处理器），19，53，68
routes（路径），752
routing（路由），789～801
adaptive（自适应），790，799～801
bits（位），790
butterfly（蝶形的），778
cut-through（直通），757（fig.），782
deadlock（死锁），791～792
deadlock-free（免死锁的），793
delay（延迟），460，758，761，781（fig.）
deterministic（确定性的），790～791
dimension order（维序），789，800
e-cube（e-立方体），778，790
fat-tree（胖树），778
mechanisms（机制），789～790
multipath（多路径），800
operations at switches（交换开关上的操作），789
source-based（基于源的），790，805
store-and-forward（存储转发），756，757（fig.），758
table-driven（表驱动的），790
turn-model（折转模型），797～799
up*-down*（向上-向下），796～797，799
wormhole（蛀洞），759，794
routing algorithms（路由算法），752～753
adaptive（自适应的），790
classes of（类别），789
deadlock-free（免死锁的），793，794
deterministic（确定性的），791
minimal（最小的），791
multipath（多路径），800
nonminimal（非最小的），791
properties of（性质），789
west-first（西向优先），797～798，799（fig.）
routing distance（路由距离），753，756，779
average（平均），753
CRAY T3D network（CRAY T3D 网络），820

cut-through（直通），779
store-and-forward（存储转发），779
run length（运行长度），899

S

saturation point（饱和点），762，763（fig.）
scalability（可扩展性），455，456～468
input buffering and（输入缓冲），484
limitations overcome by（通过……克服局限性），456～457
scalable cache coherence（可扩展的高速缓存一致性），558～559
Scalable Coherent Interface（SCI）（可扩展一致性接口（SCI））
case study（案例研究），822～825
interconnection across quads（四处理器的互连），635
links（链路），766
packet formats（数据包格式），825（fig.）
pending lists（未决表），629（fig.）
purging sharing list in（在……中清除共享表），630（fig.）
read miss（读扑空），627（fig.）
ringlets（小环），822
rings（环），636，643
serialization（串行化），639
sharing list（共享表），625（fig.）
standard（标准），570，635
transactions（事务），824
transport layer（传输层），636
scalable machines（可扩展的机器）
abstract view（抽象的观点），458（fig.）
configuration support（配置支持），461
dancehall organization（舞池式组织结构），459（fig.）
distributed-memory organization（分布式存储器组织结构）458（fig.）
sequential consistency in（顺序同一性），475
scalable multiprocessors（可扩展的多处理器），453～551
array-based lock problems（基于数组的锁问题），539
dedicated message processing（专用消息处理），496～506
with directories（带目录的），555（fig.）
parallel software implications（并行软件的影响），522～538
synchronization（同步），538～547，655
user-level access（用户级的访问），491～496
ScaLapack suite（ScaLapack 程序组），963
scaled speedup（可扩放的加速比），210
scaling（伸缩，扩放），202～214
bandwidth（带宽），457～459
bisection，rule（对分，规则），788
caches（高速缓存），236～237
cost（成本），461～462
data in bus-based systems（总线型系统的数据），445～446
directory protocol（目录协议），564～565
efficiency-constrained（效率约束的），231
error and（差错），265
in generic parallel architecture（一般的并行体系结构），467～468
importance of（重要性），204～206
key issues in（关键问题），206～207
latency（时延），460～461
linear，property（线性的，特性），209
machines（机器），206
physical（物理的），462～467
questions（问题），207
realistic（真实的），266
relationships，preserving（关系，保持），235
SGI Origin2000（SGI Origin2000），621～622
snoop bandwidth（侦听带宽），445～446
unloaded latency（无载时延），780（fig.）
VLSI（VLSI），802
workload parameters（工作负载参数），213～214
scaling down problem（规模缩小的问题），234～237
cache size for（高速缓存的大小），237（fig.）
confidence and（置信度），237
scaling models（伸缩模型，扩放模型），205，207～213
impact on equation solver kernel（对方程求解器内核的影响），211～213
memory-constrained（MC）（内存约束），207～208，210～211，431，433，621～622
problem-constrained（PC）（问题约束的（PC）），207～208，212～213，227
time-constrained（TC）（时间约束的（TC）），207，208～210，431，433
scatter decomposition（散列分解），176
scene-oriented approach（面向场景的方法），175
scheduling（调度）
algorithm（算法），808
based on latency（基于时延），885
dynamic（动态的），870
output（输出），808～810
prefetch（预取），880，883
scientific/engineering computing（科学/工程计算），6～9，218～219
SCI link interface controller（SCLIC）（SCI 链路接口控制器（SCLIC）），635，638，639
chip block diagram（芯片框图），641（fig.）

coherence controller（一致性控制器），640
directory controller（目录控制器），639
engine（引擎），640
memory-mapped counters（存储器映射的计数器），641
scope consistency（域同一性），723
segmented register file（分段寄存器文件），900
for multithreaded processor（多线程处理器），915（fig.）
statically（静态地），914
self-scheduling（自调度的），128
sender-initiated communication（发送者启动的通信），833，879
precommunication（预通信），890～891
software-controlled（软件控制的），890
sender -initiated matrix transposition（发送者启动的矩阵转置），676（fig.），929（fig.）
SEND operation（SEND操作），112
blocking asynchronous（阻塞的异步的），115
nonblocking asynchronous（非阻塞的异步的），115
semantics（语义），114～115
synchronous form（同步形式），112，114
send overhead（发送开销），522
comparison（比较），523～524
message breakdown（消息分解），523（fig.）
sense reversal（感应逆转），355
Sequent Computer Systems（Sequent 计算机系统）。参见NUMA-Q
sequential algorithms（串行算法）
Barnes-Hut application（Barnes-Hut 应用），166～169
Data Mining application（数据挖掘应用），179～180
Ocean application（Ocean 应用），161
providing（提供），389
Raytrace application（Raytrace 应用），175
sequential consistency（SC）（顺序同一性），286～291，865
comparison（比较），688（fig.）
execution model（执行模型），695
implementing（实现），287
incomplete memory operation in（未完成的存储器操作），871
interleaving（交错，交叉，交织），288，695
invalidations and（作废），710
latency hiding under（时延隐藏），871
memory operations（存储器操作），286
model（模型），682
MSI invalidation protocol and（MSI 作废协议），297～298
performance limitations（性能局限性），684
prefetching effects for（预取效果），874
preservation conditions（保留条件），289～291
at programmer's interface（在程序员的接口），683
register allocation by compiler and（通过编译器分配寄存器），683（fig.）
in rings（在环中），441，444
in scalable machines（在可扩展的机器中），475
semantics（语义），685，688，697
sequentially consistent execution（顺序同一性执行），288
sequentially consistent system（顺序同一性系统），288
serialization and（串行化），289，592～593
split-transaction bus（事务拆分型总线），406～409
for SVM（for SVM），711（fig.）
in two-level hierarchical bus design（二级层次总线设计），677
write atomicity（写原子性），288，289（fig.）
write buffers（写缓冲区），865，874
sequential loops（顺序循环），93
sequential program order（顺序的程序原序），53，54
serialization（串行化），388～390
across locations for sequential consistency（跨越单元以保证顺序同一性），592～593
buffer at home（宿主节点的高速缓存），590～591
buffer at requestor（请求方的缓冲），591
for bus-based machines（基于总线的机器），291
forward to dirty node（转发给脏节点），591
globally consistent（全局同一的），590
to location for coherence（单元一致性），589～591，604～607，632～633
minimizing（最小化），124
NACK and retry（否定回答并重试），591
of operations at different locations（在不同位置的操作），406
Raytrace application（Raytrace 应用），256
reducing（减少），130～131
in rings（在环中），441，444
SCI protocol（SCI 协议），639
sequential consistency and（顺序同一性），289
split-transaction bus（事务拆分型总线），406～409
write（写），277，288，389，589，663
SGI Challenge（SGI 的 Challenge 机），311
application performance（应用性能），429～433
application speedups（应用加速比），430～431
barrier performance on（栅障性能），357（fig.）
bus architecture（总线体系结构），400
bus interface chips（总线接口芯片），420
cache-to-cache transfers（高速缓存到高速缓存的传输），

384
case study（案例研究），415，417～424
chip types（芯片类型），420
design（设计），415
flow control（流控），424
I/O subsystem（输入/输出子系统），422～424
lock performance on（锁性能），348～350
main memory（主存储器），415，420
memory subsystem（存储器子系统），420～422
memory system performance（存储器系统性能），424
MESI protocol（MESI 协议），422
NACK lines（NACK 线）404
operating system（操作系统），415
outstanding transactions（未完成的事务），449
Powerpath-2 system bus（Powerpath-2 系统总线），417～420
processor board chip partitioning（处理器板芯片的分区），421（fig.）
processor subsystem（处理器子系统），420～422
read microbenchmark results（读微基准测试程序结果），425（fig.）
scaling results（扩放结果），432
system organization（系统组织结构），416（fig.），参见 Sun Enterprise 6000
SGI Origin2000（SGI Origin2000），36～37，150，160，556，596～622
access protection rights（访问保护权），608～609
application speedups（应用的加速比），620～621
arbitrator（仲裁器），614
automatic page migration support（自动页迁移支持），610～611
back-to-back latency（背靠背时延），619，620（fig.）
bandwidths（带宽），618
Barnes-Hut application on（Barned-Hut 应用），174
busy states（忙状态），597，600
cache coherence protocol（缓存一致性协议），597～604
case study（案例研究），596～622
characterization with microbenchmarks（通过微基准测试程序进行特征描述），618～620
correctness issues（正确性问题），604～609
deadlock（死锁），608
deadlock detection（死锁检测），595
directory lookup（目录查询），599
directory protocol（目录协议），588
directory state diagrams（目录状态图），670（fig.）
directory structure（目录结构），598～599，609
DMA operation support（DMA 操作支持），610
error handling（差错处理），608～609
fat cube（胖立方体），613
hardware（硬件），612～614
input/output operation support（输入/输出操作支持），610
interconnection network（互连网络），613
I/O configuration（输入/输出配置），614（fig.）
lazy TLB invalidation mechanism（惰性 TLB 作废机制），611
links（链路），825
livelock（活锁），608
local serialization at home node（宿主节点的局部串行化），606（fig.）
local serialization at requestor（局部串行化需求），606（fig.）
lock performance on（锁性能），650
memory consistency model（存储同一性模型），607～608
MIPS R10000 processors（MIPS R10000 处理器），597，684，868
multiprocessor block diagram（多处理器框图），598（fig.）
NACKs（NACKs），597
network case study（网络案例研究），825～826
network topology（网络拓扑），826（fig.）
node board（节点板），613（fig.）
Ocean application on（Ocean 应用），150
performance characteristics（性能特征），618～622
pipelined latency（流水线延迟），618
processing nodes（处理节点），597
protocol actions in response to requests（响应请求的协议动作），601（fig.）
protocol extensions（协议扩展），610～612
protocol states（协议状态），598～599
Raytrace application on（Raytrace 应用），179
read-exclusive handling（读互斥处理），602
read request handling（读请求处理），599～601
reply forwarding（应答转发），597，602
router connections（路由器连接），826（fig.）
scaling（伸缩，扩放），621～622
serialization to location for coherence（为一致性在单元上的串行化），604～607
speedups on（加速比），205（fig.）
SPIDER switch（SPIDER 开关），825
starvation（挨饿），608
synchronization support（同步支持），611～612
SysAD bus（SysAD 总线），609，612，613
true unloaded latency（真实无载时延），619
virtual channels（虚信道），613
write atomicity（写原子性），607

write-back request handling（回写请求处理），603～604，673
write request handling（写请求处理），601～603
Xbow（Xbow），612、613、614，参见 NUMA-Q
SGI Origin2000 Hub（SGI Origin2000 立方体），706
chip（芯片），612
chip layout（芯片布线），616（fig.）
connections（连接），612
controller components（控制器部件），615
crossbar（交叉开关），617，618
implementation（实现），614～618
memory/directory interface（MI）（存储器/目录接口（MI）），617
network interface（NI）（网络接口（NI）），618
processor interface（PI）（处理器接口），615～617。参见 SGI Origin2000
shared address（共享地址），29
Fortran 90/High Performance Fortran（Fortran 90/高性能 Fortran），52
programming（编程），26
shared address space（共享地址空间），28～37，116
abstraction support（抽象支持），475
arrays representing grid in（代表网格的阵列），147（fig.）
artifactual communication in（附加通信），142，146
cache-coherent（高速缓存一致性），879～891
without caching of shared data（不做高速缓存的共享数据），877～879
coherent replication in（一致的复制），556
communication abstraction（通信抽象），473，474（fig.）
convergence（会聚），42～44
data transfer（数据传输），37
equation solver pseudocode（方程求解器伪指令），104～105（fig.）
key primitives（关键原语），102（fig.）
memory consistency model（存储同一性模型），681
mutual exclusion（互斥），538
naming（命名），55，184
node structures（节点结构），728（fig.）
operations（操作），56
orchestration under（在……下的协调），101～108
ordering（定序），57
parallel software operations（并行软件操作），527～528
physical（物理的），52，506～513
point-to-point events（点到点的事件），538
prefetching in（预取），879（fig.）
programming model（编程模型），37
read performance comparison（读性能比较），527（fig.）
read/write to shared data（读/写共享数据），183
replication（复制），185
spatial locality in（空间局部性），146～148
threads of control（控制线程），54
virtual（虚拟的），43
without coherent replication（无一致性的复制），723～724
working set（工作集），186、参见 message passing; programming models
shared address space latency tolerance（共享地址空间时延包容），851～922
block data transfer（块数据传输），853～863
cache-coherent（高速缓存一致性），879～891
communication structure（通信结构），852
multithreading（多线程），896～922
precommunication（预通信），877～896
proceeding past communication in same threads（越过同一个线程中的通信），863～867
system application（系统应用），851～852
technique properties（技术属性），923（fig.）。参见 latency tolerance
shared bus（共享总线），453
shared caches（共享的高速缓存），434～437
associativity（相联度），436
bandwidth requirement satisfaction（带宽需求满足），436
benefits of（好处），434～435
cache hardware utilization（高速缓存的硬件利用率），435
disadvantages of（缺点），436～437
examples（例子），435
first-level（第一层），435，436～437
hit latency（命中时延），436
multiprocessor architecture（多处理器体系结构），435（fig.）
size of（大小），436
spatial locality and（空间局部性），434
working sets and（工作集），434，参见 caches
shared memory multiprocessors（共享存储器的多处理器），23，28，269～376
bus interconnection（总线互连），32
communication assist and communication hardware（通信辅助部件和通信硬件），51～29
crossbar switch（交叉开关交换机），30，31，34
extending systems into（扩展系统），31（fig.）
interconnection schemes（互连方案），31（fig.）
latency and（时延），37
memory capacity, increasing（存储器容量，增加），29

memory model（存储器模型），30（fig.）
multistage interconnect（多级互连），30～31，34
processes（进程），49
programming model（编程模型），29
scalable（可扩展的），34
small-scale（小规模的），66，453
software implications（软件的影响），359～366
synchronization（同步），334～358
shared pending（SP）state（共享的未决（SP）状态），925
shared physical address space（共享的物理地址空间），52，506～513
cachability（可高速缓存性）508
communication assist（通信辅助部件），506，507
CRAY T3D case study（CRAY T3D案例分析），508～511
CRAY T3E case study（CRAY T3E案例研究），512～513
early designs（早期的设计），506
limitations，overcoming（局限性，克服），732
machine organization（机器组织结构），507（fig.）
Memory Channel（存储器通道），518
MMU（存储器管理部件），507
summary（总结），513。参见 shared address space
shared pools（共享池），807
shared virtual memory（SVM）（共享的虚拟存储器（SVM）），709～724
coherence protocol（一致性协议），733
communication（通信），712（fig.）
high-performance，protocols（高性能，协议），709
illustration（说明），710（fig.）
memory management problems（存储器管理问题），726
page-based（基于页的），709
page-grained（页粒度的），739
path of read operation（读操作的路径），719～720
performance implications（性能影响），720～721
sequential consistency for（顺序同一性），711（fig.）
software，coherence（软件，一致性），721
software，layer（软件，层），716
TreadMarks system（TreadMarks系统），716
write notices（写通告），711
sharing list（共享表）
elements（元素），625（fig.）
head node（头节点），628
long（长），629
primitive operations（原语操作），625
purging（清除），630（fig.）
purging latency（清除时延），629
writer in（写入），629
sharing write back（共享的回写），600
Shasta（Shasta），745
short links（短链路），764
SHRIMP（SHRIMP），718
Sigma-1（Sigma-1），494
Simple COMA（简单 COMA），681，726～728
design（设计），726
drawbacks（缺点），727
path of read reference（读访问的路径），727～728
performance trade-offs（性能折中），726～727
presence check（存在性检查），726。参见 cache-only memory architecture（COMA）
simulated time（模拟的时间），233
simulations（模拟）
event-driven（事件驱动），233，234（fig.）
expense of（代价），232
limitations of（局限性），200，232
machine parameters for（机器参数），234～237
multiprocessor（多处理器），200，233～234
speeding up（加速化），234
trace-driven（踪迹驱动的），233
simulators（模拟器），199
accurate，building（精确的，构造），232
memory system（存储器系统），233
processor（处理器），233
simultaneous multithreading（并发多线程），920～922
illustrated（说明的），921（fig.）
issues（问题），921～922
for uniprocessors（单处理器），922。参见 multithreading
single-instruction-multiple-data（SIMD）（单指令多数据（SIMD）），44，103
data parallel programming model，（数据并行程序设计模型），44
evolution（演变，进化），46～47
organization（组织），45（fig.）
renaissance（复兴），46
single-instruction-single-data（SISD）（单指令单数据（SISD）），44
single-level caches（单级高速缓存），281
with atomic bus（原子总线），380～393
atomic operation implementation（原子操作的实现），391～393
base organization（基本组织结构），385
cache controller design（高速缓存控制器设计），381～382
deadlock（死锁），390
livelock（活锁），390

nonatomic state transitions（非原子性的状态转换），385～388
serialization（串行化），388～390
snoop results（侦听结果），382～384
starvation（挨饿），390～391
tag design（标记设计），381～382
write backs（回写），384～385。参见 caches
single-program-multiple-data（SPMD）(单程序多数据(SPMD))，46～47，103
single-system image（单一系统映像），513
single writer protocol（单写入者协议）
with consistency at acquire（获取时的同一性），734～737
with consistency at release（释放时的同一性），733～734
recent copy invalidation（最近副本作废），735（fig.）
slackness（松弛），155
slotted rings（分槽环），443
snoop-based multiprocessors（基于侦听的多处理器）
cache-coherent（高速缓存一致性），278（fig.）
design（设计），377～452
snooping（侦听）
buses（总线），277～283，589
cache controllers（高速缓存控制器），277
hierarchical（层次的），559，660～664
latency（时延），383
results，matching（结果，匹配），402～403
results，reporting（结果，报告），382～384
snooping caches（侦听高速缓存）
base machine design（基本机器设计），386（fig.）
organization of（组织结构），383
two-level，organization（两级，组织结构），398（fig.）。参见 caches
snooping protocols（侦听协议），280
cache-coherent（高速缓存一致性），272
components（部件），280
design space for（设计空间），291～305
supporting（支持），381
software caches（软件高速缓存），186
software combining trees（软件合并树），542～543
flat arrive structure vs.（与扁平的到达结构相比），543（fig.）
for release（为了释放），543
software-controlled prefetching（软件控制的预取技术），880
coverage（覆盖性），889
effectiveness（有效性），889，895
hardware-controlled vs.（与硬件控制的比较），888～890
scheduling（调度），884
with single processor（单一处理器），884～887
unnecessary prefetch reduction（不必要的减少预取），889
参见 prefetching
software instrumentation（软件修改），707～708
protocol processing（协议处理），708
run-time cost（运行时间代价），707
of write operations（写操作），718
software locks（软件锁），338～340
implementation（实现），339
problem（问题），338～339
state storage（状态存储），338。参见 locks
software overflow scheme（软件溢出方案），657～658
for limited pointer directories（有限指针目录），658
overhead（开销），657～658
software queuing locks（软件排队锁），539
algorithm for（算法），541（fig.）
compare&swap operation（比较并交换操作），540
space per lock（每个锁的空间），541
states（状态），540（fig.）。参见 locks；software locks
Solaris UNIX（Solaris UNIX），416
source-based routing（基于源的路由），790，805
space-sharing（空间共享），89～99
sparse directory（稀疏目录），659
spatial interleaving（空间交错），360～362
spatial locality（空间局部性），142，319～320
analyzing（分析），885
Barnes-Hut application（Barnes-Hut 应用），170～172，322
in capacity misses（容量型扑空），324
centralized memory and（集中式存储器），360
in cold misses（冷启动扑空），324
Data Mining application（数据挖掘应用），182
in equation solver kernel（方程求解器内核），149（fig.）
exploiting（发掘），145～150
increasing by copying data（通过复制数据增加），363～364
interaction example（相互作用的例子），225
linked lists and（链表），324
LU（LU），320，322
mismatches（不匹配），146
Multiprog（多道程序），323～324
Ocean application（Ocean 应用），163～164，320，321
poor（不良），145
problem size impact on（问题规模的影响），225（fig.）
of processes' access patterns（进程的访问模式），148
Radiosity application（Radiosity 应用），322
Raytrace application（Raytrace 应用），176～177
scaling models（伸缩模型，扩放模型），213

in shared address space（共享地址空间），146～148
shared caches and（共享高速缓存），434
using（使用），224～226。参见 locality
SPEC benchmark（SPEC 基准测试程序），13，199，968
SPEC95（SPEC95），199
SPEC92（SPEC92），199。参见 benchmarks
speculative execution（投机性执行），684，870～871，876
instructions（指令），870～871
lookahead buffer（先行缓冲），870
Radix（基数），875（fig.）
speculative reads（投机性读），684，872，876
speculative reply（投机性应答），600
speedup（加速比），122，203
algorithmic（算法），220，254，256（fig.）
application（应用），430～431
Barnes-Hut（Barnes-Hut），431
blocked multithreading（阻塞的多线程），912（fig.）
CRAY T3D（CRAY T3D），532
IBM SP-2（IBM SP-2），532
interleaved multithreading（交错的多线程），912（fig.）
limit（限制），135，136
LU（LU），431
MC scaling（存储器约束的扩放），211，213
measuring（测量），163，203，229
Ocean（Ocean 应用），431
on PRAM architectural model（PRAM 体系结构模型），254
on SGI Origin2000（SGI Origin2000），205（fig.），620～621
PC scaling（问题约束的扩放），208，213
perfect（完美的），266
Radiosity（Radiosity 应用），431
Radix（Radix 应用），431
Raytrace（Raytrace 应用），431
scaled（伸缩的），210
self-relative（自相关的），163
superlinear（超线性的），204，205
TC scaling（时间约束的扩放），209，213
UltraSparc cluster（NOW）（UltraSparc 集群（NOW），532
over uniprocessor（优于单处理器），204
SPIDER switch（SPIDER 交换开关），825
spin-waiting（踏步等待），106～107
SPLASH-2 suite（SPLASH-2 程序组），307，965～966
Split-C language（Split-C 语言），724
split-transaction bus（事务拆分型总线），398～415
advantages/disadvantages（优点/缺点），398
alternative design choices（其他设计选择），409～410
cache miss path（高速缓存扑空路径），404～406
example design（示例设计），400
flow control（流控），404
incoming transactions（进入的事务），408
issues（问题），398～399
lock-down（锁定），391
with multilevel caches（多层高速缓存），410～413
multiple outstanding miss support（多个未决扑空支持），413～415
negative acknowledgment（NACK）（否定应答（NACK）），400
read transaction（读事务），401（fig.）
request-response matching（请求-响应匹配），398，400～402
request transaction（请求事务），398
response transaction（响应事务），398
sequential consistency（顺序同一性），406～409
serialization（串行化），406～409
snoop results（侦听结果），402～403
SparcCenter 2000（SparcCenter 20000），410
SRAM（静态随机存储器（SRAM）），949，950
stable states（稳定状态），387
Stache（Stache），726～728
allocation management（分配管理），727
memory management（存储器管理），727
stacked dimension switches（栈式维度交换开关），810～811
design（设计），810
illustrated（说明的），811（fig.）。参见 switches
Stanford DASH（Standford 的 DASH 机），640，910
deadlock detection（死锁检测），594
directory-based coherence（基于目录的一致性），596
multiprocessor（多处理器），301
project（项目），69
Stanford FLASH（Standford 的 FLASH 机），640，648，658
coherence controller（一致性控制器），731
programmable protocol engine（可编程协议引擎），706
starvation（挨饿），380，390～391
directory protocols（目录协议），595～596
elimination of（消除），380，390
NUMA-Q（NUMA-Q），633
potential（潜在的），380
SGI Origin2000（SGI Origin2000），608
solution（解），595～596。参见 deadlock；starvation
state bits（状态位），560
states（状态）
busy（忙），603～604
cache block（高速缓存块），279

directory（目录），603～604
dirty（脏），600
exclusive（独占的），603
exclusive-clean（E）（独占-干净），302，600
exclusive pending（EP）（独占-挂起），925
expanding number of（扩展数量），387
invalid pending（IP）（无效的挂起），925
master（主方），704
modified（M）（已修改的），293，302
NUMA-Q（NUMA-Q），624～626
shared（共享），293
shared-clean（Sc）（共享-干净），302
shared-modified（Sm）（共享-被修改），302
share pending（SP）（共享挂起），925
stable（稳定的），387
transient（过渡的），388（fig.）
state transition diagrams（状态转换图）
cache block（高速缓存块），279～280
Dragon protocol（Dragon 协议），303（fig.）
MESI protocol（MESI 协议），301（fig.），388（fig.）
Powerpath-2 bus（Powerpath-2 总线），418（fig.）
state transitions（状态转换），308～309
applications with smaller caches（较小高速缓存的应用），314
bus actions corresponding to（对应的总线动作），311
cache block（高速缓存块），367
Dragon update protocol（Dragon 更新协议），303～304
frequency data（频度数据），308～309
MESI invalidation protocol（MESI 作废协议），299～300
MSI invalidation protocol（MSI 作废协议），294～296
nonatomic（非原子的），385～388
static assignment（静态分配），88，99
dynamic assignment vs.（与动态分配相比较），126～129
load balance and（负载平衡），126
task granularity with（任务粒度），130
task management and（任务管理），126
task size and（任务规模），130。参见 assignment
static ordering（静态次序），445
steady-state loop（稳态循环），850
STOP symbol（停止符号），815
storage technology revolution（存储技术革命），67
store-and-forward（存储转发），40，460
store-and-forward routing（存储转发路由），756，758
cut-through routing vs.（与直通路由比较），757（fig.）
deadlock（死锁），792
delay（延迟），942
distance（距离），779
store-conditional（条件存储），344，652
stream buffers（流缓冲区），882
subblock write bits（SWBs）（子块写位（SWBs）），925～926
subgrids（子格），133，148
Sun Enterprise 1000（Sun Enterprise 1000），445～446
Sun Enterprise Server（Sun Enterprise 服务器），35（fig.），384，404
Sun Enterprise 6000（Sun Enterprise 6000），415～416，424～429
block diagram（框图），417（fig.）
design（设计），415
D-tags（D-标签），427，429
FiberChannel modules（光纤信道模型），429
Gigaplane system bus（Gigaplane 系统总线），424～427
I/O subsystem（输入/输出子系统），429
memory（存储器），416
memory subsystem（存储器子系统），427～429
memory system performance（存储器系统性能），429
operating system（操作系统），416
processing and I/O board（处理和 I/O 板）
organization，（组织结构），428（fig.）
processor subsystem（处理器子系统），427～429
read microbenchmark results（读取微基准测试程序结果），430（fig.）
SysIO ASICs（SysIO ASICs），429。参见 SGI Challenge
Sun Spare（Sun Spare）
MEMBAR instructions（MEMBAR 指令），693
PSO model（PSO 模型），689
V8（V8），689
V9 RMO（V9 RMO），689，690，693
Sun SpareCenter 2000（Sun SpareCenter 2000），331
multiple bus approach（多总线方式），445
split-transaction buses（事务拆分型总线），410
Sun UltraSpare（SUN 公司的 UltraSpare），533，868，940
supercomputers（超级计算机），21～23
CRAY vector（CRAY 向量），8，22
performance（性能），24（fig.）
uniprocessor performance of（单处理器性能），22（fig.）
vector processors（向量处理器），21～22，946
superlinear speedup（超线性加速），204，205
superscalar execution（超标量执行），17
superscalar processors（超标量处理器），895
surface-area-to-volume ratio（表面积对体积的比例），132
swap instruction（交换指令），339
switches（交换开关），457，767-768

bandwidth（带宽），939
buffering（缓冲），759
context（上下文），897，901
cost（成本），768，900
degree（度），457，752，767，768，801
design of（设计），801～811
input buffered（输入缓冲的），805（fig.）
input ports（输入端口），767，802
internal buffering（内部缓冲），768
internal datapath（内部数据路径），802～804
"intranode"，（"结点内"），457～458
Myricom network（Myricom 网络），827
network（网络），457
organization（组织结构），767（fig.）
output ports（输出端口），767，802
output scheduling（输出调度），808～810
routing operations at（路由操作），789
scale limitation（扩放限制），457
SPIDER（SPIDER），825
stacked dimension（栈式维度），810～811
VLSI（超大规模集成电路），768，802，804
switching（交换）
circuit（电路），756～757
cost（成本），898
delay（延迟），756
packet（数据包），757
strategy（策略），753
time（时间），897
symmetric multiprocessors（SMPs）（对称多处理器），32～34，269～271
building blocks（构造块），515
bus-based（基于总线的），271，457
design challenge（设计挑战），366～367
inexpensive（廉价的），949
layers of abstraction（抽象层），270（fig.）
nodes（节点），467，503，721
symmetric successive overrelaxation（SSOR）（对称连续超松弛），532
Synapse multiprocessor（Synapse 多处理器），298
Synchronization（同步），334～359
algorithms for barriers（栅障算法），542～547
algorithms for locks（锁算法），538～541
Barnes-Hut application（Barnes-Hut 应用），172～173
barrier（栅障），252
block data transfer and（块数据传输），857
Data Mining application（数据挖掘应用），182
directory-based cache coherence（基于目录的高速缓存一致性），648～652
directory-based multiprocessors（基于目录的多处理器），556
event（事件），57，103，106，283
execution time component（执行时间的成分），158
explicit（显式的），285（fig.）
fine-grained（细粒度的），130
frequency of（频率），95
global（全局的），95，96，106
interprocess（进程间），118
library design（库设计），336
lock-free（免锁的），351
MC scaling（MC 伸缩），213
message-passing program（消息传递程序），111，113
microbenchmarks（微基准测试程序），216
multiple-producer，singleconsumer group（多生产者、单消费者组），173
mutual exclusion（互斥），57，103，113，124，188，337～351
nonblocking（非阻塞），351
Ocean application（Ocean 应用），165
operation order（操作次序），714（fig.）
PC scaling（PC 伸缩），213
point-to-point（点到点），96，117，172
QOLB（QOLB），626，651
Raytrace application（Raytrace 应用），177
scalable multiprocessor（可扩展的多处理器），538～547，655
SGI Origin2000 support（SGI Origin2000 支持），611～612
for shared data variable access（共享数据变量访问），691（fig.）
shared memory multiprocessor（共享存储器多处理器），334～359
software algorithms（软件算法），336
summary（总结），358
support（支持），57
TC scaling（TC 伸缩），213
wait-free（无等待），350
wait time（等待时间），124，866
workload（工作负载），254
synchronized programs（同步程序），695
at programmer's interface（程序员接口），699
yielding（产生），697
synchronous links（同步链路），765
synchronous message passing（同步消息传递），39

matching rule（匹配规则），478
protocol（协议），478（fig.）。参见 message passing
system area networks（SANs）（系统域网络（SANs）），467，750
Myrinet（Myrinet）516～518
scalable high-performance（可扩展的高性能），503
system design trends（系统设计趋势），19～21
system-level integration（系统级集成），466～467
system specification（系统规范），685～686，690，693
Alpha（Alpha）693
characteristics（特征），694（fig.）
PC（PC），686，687，688（fig.）
PowerPC（PowerPC），693
PSO（PSO），686，689
RC（RC）687，690，691～692
TSO（TSO），686，687，688（fig.），689
WO（WO），686～687，690～691。参见 relaxed memory consistency models
systolic architectures（脉动体系结构），49～50
computation of inner product（内积计算），50（fig.）
PEs（PEs），49
solutions on generic machines（在一般机器上的解法），50。参见 parallel architectures

T

* T machine，（T machine），495
table-driven routing（表驱动的路由），790
tail node（尾节点），624
Tandem Himalaya system（Tandem 的 Himalaya 系统），12
task granularity（任务粒度），129～131
determining（确定的），129～131
with dynamic task queuing（动态任务队列），129～130
fineness（细），130
with static assignment（静态分配），130
task pools（任务池），127，128
task queues（任务队列），128，129
distributed（分布式的），192
implementing（实现），130～131
locking（加锁），131
remote, accessing（远程的，访问），131
tasks（任务），82～83
assignment of（分配），83，116
coarse-grained（粗粒度的），83
examples（示例），82～83
fine-grained（细粒度的），83
relationships of（关系），84（fig.）
task stealing（任务窃取），128
dynamic（动态的），134
implementing（实现），128
termination detection and（终止检测），195
TC（time-constrained）scaling（TC（时间约束的）扩放），207，431，433
communication-to-computation ratio（通信与计算之比），212，433
concurrency（并发），212
memory requirements（存储器需求），212
naive（简单的），433
problem size（问题规模），208
realistic（真实的），433
spatial locality（空间局部性），213
speedup（加速比），209，213
synchronization（同步），213
temporal locality（时间局部性），213
viability（生存力），211。参见 scaling; scaling models
technology trends（技术趋势），12～14
temporal locality（时间局部性），142，143～145
analyzing（分析），885
Barnes-Hut application（Barnes-Hut 应用），172
Data Mining application（数据挖掘应用），182
in equation solver kernel（方程求解器内核），143～144
exploiting（发掘），143～145，359
goal（目标），359
implications（隐含的意义），145
Ocean application（Ocean 应用），164
Raytrace application（Raytrace 应用），177
scaling models（扩放模型），213
techniques（技术），145。参见 locality
Tera architecture（Tera 体系结构），906～908
active thread support（主动线程支持），907
general purpose multiprocessing（通用多处理），908
minimum issue delay（最小发射延迟），909。参见 interleaved multithreading
termination detection（终止检测），128，195
tertiary caches（一级高速缓存），700～701
test&set instruction（测试并设置指令），339，341，344，391
implementing（实现），391
success determination（成功的判定），339
test&set locks（测试并设置锁）
with backoff（回退），342
performance（性能），340，341（fig.）
problem with（问题），341
test-and-test&set locks（测试-测试并设置锁），342～343

failure（错误），344
latency（时延），343
TFLOPS（one trillion floating-point operations per second）（TFLOPS（每秒万亿次浮点操作）），502
Thinking Machines.（Thinking Machines.）。参见 CM-1；CM-2；CM-5
thread-level parallelism（线程级并行），17～19
threads（线程），29，53，83
active（主动的），898，907
busy time（忙时间），897
context（上下文），897
idle time（空闲时间），898
increasing number of（数量增加），899
lightweight（轻量级），136
multiple concurrent（多并发），19
ready（就绪），898
in RISC machines（在 RISC 机中），53
shared address space（共享地址空间）
programming model（程序设计模型），54
switching arrangement（切换安排），850，851
switching time（切换时间），897
unready（未就绪），909～910。参见 multithreading
3D cubes（3D 立方体），769，770（fig.）
three-message miss（三消息的扑空），586
three-state invalidation protocol（三状态的作废协议），293～299
ticket locks（票号锁），346～347
acquire method（获取方法），346～347
performance（性能），350
read traffic problem（读流量问题），347。参见 locks
tiles（图像分片），176
token-passing rings（令牌传递环）442～443
topologies（拓扑） 参见 network topologies
topology-oriented program design（面向拓扑的程序设计），153
torus（torus）
illustrated（说明的），770（fig.）
links per node（每个节点的链路），775
routing chip（路由芯片），810
3D（3D），771。参见 meshes
total store ordering（TSO）（全序存储序（TSO）），686，865，866，867
comparison（比较），688（fig.）
write atomicity（写原子性），689
write ordering preservation（写次序保持），687 参见 system specification
TPC。参见 Transaction Processing Council（TPC）
trace-driven simulation（踪迹驱动的模拟），233
trade-offs（折中），122，123
architectural（体系结构上的），531
block data transfer（块数据传输），854～856，859
busy-waiting and blocking（忙等待和阻塞），335
cost-performance（价格性能比），2，4，15
emergence of（出现），148
evaluating（评估），199，231～243
extra work，load balance（额外工作，负载平衡）
communication（通信），136
hardware/software（硬件/软件），679～747
identifying（识别），199
latency（时延），784
network design（网络设计），749～750
partitioning（划分），135
protocol design（协议设计），305～334
realistic applications for（真实的应用），123
switch-to-switch layer（开关到开关层），827
trailer（报尾，尾部），754，766
Transaction Processing Council（TPC）（事务处理委员会），10
data（数据），10，11
March 1996 report（1996 年 3 月的报告），12
result reporting（结果报告），964
TPC-A（TPC-A），964
TPC-B（TPC-B）642，643，964
TPC-C（TPC-C）10，964～965
TPC-D（TPC-D）642，643，965。参见 benchmarks
transient states（过渡状态），388（fig.）
transistors，per processor chip（每处理器芯片的晶体管数目），16
translation lookaside buffer（TLB）（转换检测缓冲器（TLB）），67
ASIDs（地址空间标识（ASIDs）），440
coherence（一致性），439～441
control registers（控制寄存器），915
entries（入口），440，441
flush notices（刷新通知），451
handler（处理例程），429
hardware-loaded（硬件装载的），440
lazy，invalidation mechanism（惰性的，作废机制），611
misses（扑空），223，429
PTEs（PTEs），439
shootdown（击落），440，451，452（fig.）
software-loaded（软件装载的），440
translation mechanism（翻译机制），686，698
TreadMarks SVM system（TreadMarks SVM 系统），716

tree barriers（树栅障）
arrival（到达），543（fig.）
combining, with sense reversal（带有感应逆转的合并），543，544（fig.）
with local spinning（本地踏步），543～545
release（释放），543
static binary（静态二叉树），544
traffic distribution（流量分布），543
trees（树），772～774
binary（二叉树），772，773（fig.）
bisection（对分），774
branch factor（分支因子），773
fat（胖的），774，776
global（全局的），957
links per node（每个节点的链路），775
locality（局部性），668
saturation（饱和），760
16-node（16－节点），774
wire problem（连线问题），773
trends（趋势）
application（应用），6～12
architectural（体系结构的），14～21
microprocessor design（多处理器设计），15～19
system design（系统设计），19～21
technology（技术），12～14
VLSI technology（VLSI技术），938
TruCluster Memory Channel software（TruCluster存储器信道软件），520
true-sharing misses（真共享扑空），315
block size and（块尺寸），318
reducing（减少），316
turn-model routing（折转模型路由），797～799
minimal（最小化），798（fig.）
restrictions（约束），798（fig.）
virtual channels（虚信道），799。参见 routing
twins（孪生子），716
2D grids（2D格），769，770（fig.）
two-level sharing hierarchy（二级共享层次结构），587～589
advantages（优点），587
locality and（局部性），588

U

uniform cluster cache（UCC）card（统一的机群高速缓存），663
uniform interconnection card（UIC）（统一的互连网络卡（UIC）），663
uniprocessors（单处理器）
bandwidth（带宽），939
cache controller（高速缓存控制器），381～382
cache design in（高速缓存设计），314，381
compilers（编译器），289
memory system（存储器系统），275
PC shipments（PC装运），936
simultaneous multithreading for（同步多线程），922
speedup over（加速比），204
state diagram（状态图），280
supercomputer performance（超级计算机性能），22（fig.）
write-back caches on（回写高速缓冲），292
unloaded latency（无载时延），755，780～785
with equal pin count（相等的引脚数），785（fig.）
as function of degree（作为度的函数），781（fig.）
for k-ary d-cubes（k元d－立方体），783（fig.）
for n-byte packet（对于n个字节的数据包），756
scaling（伸缩），780（fig.）
for short messages（对于短消息），780。参见 latency
unnecessary prefetches（不必要的预取），881
minimizing（最小化），884
reducing（减少），889。参见 prefetching
up*-down* routing（向上－向下路由），796～797，799
update-based protocols（基于更新的协议），278，292
bounding losses of（限制损失），331
Dragon（four-state）（Dragon（四状态）），301～305，374
Firefly（Firefly），374
invalidation-based protocols（基于作废的协议）
combined with（组合），330～332
invalidation-based protocols vs.（与基于作废的协议比较），329～330
miss rates（扑空率），332（fig.）
upgrade/update rates for（升级/更新率），333（fig.）
write atomicity（写原子性），673
write operation and（写操作），292。参见 protocols
upgrades（升级），318
user flags（用户标记），491
user-level access（用户级访问），491～496
case study（案例研究），493～494
CM-5（CM-5），493～494
node-to-network interface（节点到网络的接口），491～493
user-level handlers（用户级处理例程），494～496
communication assist（通信辅助部件），495（fig.）
message processing（消息处理），493
user-level network port（用户级网络端口），491
architecture（体系结构），492（fig.）

communication assist for（通信辅助部件），492（fig.）
user-oriented properties（面向用户的特性），207
utilization metric（应用度量指标），230，262

V

vector processors（向量处理器），21～22，67
vector time stamp（向量时间戳），736
virtual channels（虚信道），613，795～796
in basic switch（在基本交换开关中），795（fig.）
breaking deadlock cycles with（切断死锁回路），796（fig.）
buffering（缓冲），807～808
routing algorithm and（路由算法），796
support（支持），807
turn-model routing with（折转模型路由），799
uses for（用于），795。参见 channels
virtual circuits（虚电路），514
virtually indexed caches（虚拟索引的高速缓存），437～439
organization（组织结构），439（fig.）
processor and（处理器），438
virtual index（虚拟索引），438
visualization case study.（可视化案例研究），参见 Raytrace application
VLSI（VLSI）
CMOS devices（CMOS 设备），944
feature size（特征尺寸），12
generation（代），15，63
scaling（伸缩，扩放），802
switches（交换开关），768，802，804
technology trends（技术趋势），938
von Neumann model（冯·诺依曼模型），189

W

wait-free synchronization（无等待同步），351
waiting algorithm（等待算法），335，336
wall-clock time（墙钟时间），228～229
WAN flow control（广域网流控），812～813
weak ordering（WO）（弱序（WO）），686～687，690～691，699，866，867
motivation（动机），690
at programmer's interface（程序员接口），699
release consistency vs.（与释放同一性比较），691（fig.）。参见 system specification
west-first algorithm（西向优先算法），797～798，799（fig.）
wide links（宽链路），764～765
working sets（工作集）
Barnes-Hut application（Barnes-Hut 应用），172
curves（曲线），240，260（fig.）
effect of（效果），227（fig.）
execution characteristics and（执行特征），222
growth rates（增长率），261
LL benchmark curves（LL 基准测试程序曲线），537（fig.）
MC scaling（MC 伸缩），227
nonlocal data（非局部数据），239
Ocean application（Ocean 应用），164
PC scaling（PC 伸缩），227
problem sizes based on（问题规模），224（fig.）
Raytrace application（Raytrace 应用），177
shared address space（共享地址空间），186
shared caches and（共享高速缓存），434
sizes（大小），259～261
workload case studies（工作负载案例研究），244～253
LU（LU），244～248
Multiprog（多道程序），244，252
Radiosity（Radiosity），244，249～252
Radix（Radix），244，248～249，267
workload-driven evaluation（工作负载驱动的评价），199～267
of architectural idea（体系结构概念），231～243
difficulty of（难点），200
of fixed-size machine（固定规模的机器），221～226
performance metrics，choosing（性能指标，选择），228～231
protocol trade-offs（协议权衡），332～334
for real machine（对于真实的机器），215～231
of trade-off（权衡），231～243
varying machine size and（改变机器规模），226～228
workload parameters（工作负载参数）
relationships among（之间的关系），214
scaling（伸缩，扩放），201，213～214
workloads（工作负载）
benchmark（基准测试程序），201
characteristics of（特征），253～261
choosing（选择），216～220，238
communication-to-computation ratio（通讯与计算之比），257～259
concurrency（并发），220，254～257
coverage of behavioral properties（行为特性的覆盖），219～220
data access（数据访问），254
load balance（负载平衡），254～257
with low/high communication-to-computation ratios（低的/高的通讯与计算之比），219
multiprogrammed（多道程序的），218

representation of application（应用的表示）
domains（域），218～219
synchronization（同步），254
working set sizes（工作集尺寸），259～261
work metric（工作指标），209
workstations（工作站）
networks of（NOWs）（工作站网络（NOWs）），513～521
performance（性能），71（fig.）
wormhole routing（蛀洞路由），759，794，808
write atomicity（写原子性），288，389
appearance of（表现），593
distributed interconnect and（分布式互连），592
example（例子），289（fig.）
importance（重要性），288
in invalidation-based scheme（在基于作废的方案中），592～593
preservation（保持），291
SGI Origin2000（SGI Origin2000），607
TSO（TSO），689
with update protocols（对于更新协议），673
violation in scalable system（在可扩展系统中的违规），593（fig.）。参见 sequential consistency
write-back buffers（回写缓冲区），385
write-back caches（回写高速缓存），274，291
buffer deadlock problem in（缓冲区死锁问题），412
invalidation-based protocol（基于作废的协议），293～299，391
L_1（L_1），396
on uniprocessors（在单处理器中），292
space of protocols for（协议空间），283
update-based protocol（基于更新的协议），301～305。参见 caches
write backs（回写），384～385
NUMA-Q handling of（NUMA-Q 处理），630～632
SGI Origin2000 handling of（SGI Origin2000 处理），603～604，673
sharing（共享），600
write buffers（写缓冲区），413
bypassable（可旁路的），867
merge restriction into（合并的限制），865
SC utilization of（SC 的利用），865，874 参见 buffers
write-fence operation（写－隔栅操作），741
write memory barrier（WMB）（写存储器栅障 WMB）），693
write misses（写扑空）
hierarchy flow（层次流），666
performance impact（性能影响），865～868
proceeding past（越过），864～868
write-no-allocate caches（写不分配的高速缓存），281（fig.）
write notices（写提示），711，734
acquirer retention of（获取者保持），735
at every release（在每个释放点），733
in lazy implementation（在惰性实现中），717
propagation of（传播），734
in release-based protocol（基于释放的协议），737
write propagation（写传播），277
write request buffers（WRBs）（写请求缓冲区（WRBs）），615
write requests（写请求）
NUMA-Q（NUMA-Q），628～630
SGI Origin2000（SGI Origin2000），601～603
write serialization（写串行化），277，289，663
extending（扩展），288
providing（提供），389
write sharing（写共享），359
write-through caches（直写高速缓存），278，279，291
invalidation-based protocol for（基于作废的协议），280，283（fig.）
L_1（L_1），396
multiprocessor snoopy coherence（多处理器侦听一致性），281（fig.）
write-to-read program order（写－读程序次序），687～689
write-to-write program orders（写－写的程序次序），689

X～Z

Xbow（xbow），612，613，614