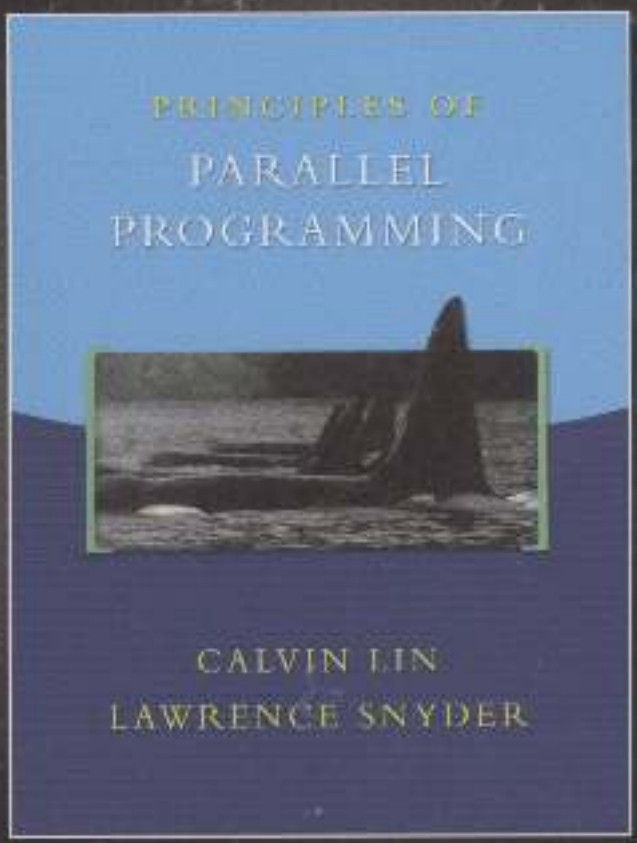
计 算 机 科 学 丛 书

并行程序设计原理

(美) Calvin Lin Lawrence Snyder 著 陆鑫达 林新华 译



Principles of Parallel Programming

机械工业出版社
China Machine Press

并行程序设计原理

多核体系结构的出现使[redacted]工程师和计算机系统设计师变得日益重要。本书着重论述并行[redacted]现象，并分析为何这些现象是成功进行并行程序设计的机遇或[redacted]

本书是高等院校计算机专业高年级本科生或低年级研究生的理想教科书，同时也是专业程序员从事并行程序设计的理想入门书。

本书特色

- 以原理第一的原则重点阐述并行计算的基本原理，而不是指导读者“如何”去管理当前商品化的并行计算机。
- 以原理为背景讨论流行的程序设计语言并论述当代并行计算机编程所使用的工具。
- 使用注释框对书中所提及的内容进行饶有兴趣的扩展。
- 使用定义框对书中关键词和概念进行定义。
- 每章附有习题，便于读者掌握所论述的概念。
- 第10章着重论述可能影响该研究领域未来的当前进展。
- 第11章为读者构造实际的并行程序提供第一手的实践经验。

影印版
ISBN 978-7-111-24734-0
定价：49.00元

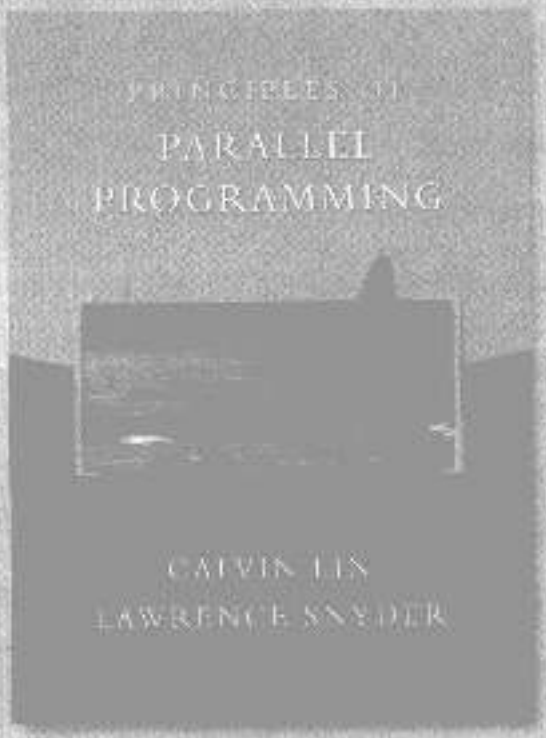
www.PearsonEd.com
投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com
华章网站 http://www.hzbook.com
网上书店：www.china-pub.com

ISBN 978-7-111-27075-1
9 787111 270751
定价：45.00 元

计 算 机 科 学 丛 书

并行程序设计原理

(美) Calvin Lin 著 Lawrence Snyder 著 陆鑫达 林新华 译



Principles of Parallel Programming

机械工业出版社
China Machine Press

本书内容新颖，涉及现代并行硬件和软件技术，包括多核体系结构及其并行程序设计技术。本书侧重论述并行程序设计的原理，并论述了并行程序设计中一些深层次问题，如可扩展性、可移植性以及并行程序设计应遵循的方法学等。

本书是高等院校计算机专业高年级本科生或低年级研究生的理想教科书，同时也是专业程序员从事并行程序设计的理想入门书。

本书版权登记号：图字：01-2009-1336

图书在版编目（CIP）数据

并行程序设计原理 /（美）林（Lin, C. ），（美）斯奈德（Snyder, L. ）著；陆鑫达等译. —北京：机械工业出版社，2009.7

（计算机科学丛书）

书名原文：Principles of Parallel Programming

ISBN 978-7-111-27075-1

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

中国版本图书馆CIP数据核字（2009）第070624号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：刘立卿

三河市明辉印装有限公司印刷

2009年7月第1版第1次印刷

184mm × 260mm · 15.75印张

标准书号：ISBN 978-7-111-27075-1

定价：45.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，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和发展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到积极的推动作用，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分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson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、研究及珍藏。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，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，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。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

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。随着计算机科学与技术专业学科建设的不断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将步入一个新的阶段，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。华章分社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：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育

推荐序

早在1842年，意大利数学家Luigi Menabrea 就在其《查尔斯·巴贝奇的分析引擎》（《Sketch of the Analytical Engine Invented by Charles Babbage》）一文中阐述了并行的理念。他的这篇文章涉及计算机系统结构和程序设计的诸多方面。到了1958年，S.Gill针对并行程序设计进行了探讨，而IBM的John Cocke和Daniel Slotnick则针对并行数值计算进行了探讨。然而直到1962年，第一台多处理计算机才藉由Burroughs公司的D825而面世。此后，为数众多的公司提出并推出了不同系统结构的多处理器系统。

曾几何时，工业界和学术界进行了无数次的尝试，力图将并行程序设计置于主流程序设计的核心地位，但由于种种原因，这些尝试最终都未能取得成功。最主要的原因在于我们既然有能力将处理器的主频每12～18个月就增加一倍，因此除了那些依赖于大规模并行系统来加速其模拟运算的科学社团外，其他人就几乎找不到任何理由来编写并行程序。

然而近来，半导体工业界与微处理器供应商却遭遇到了屏障，即虽然我们依然能增加芯片的密度，但却再也无法增加主频。有鉴于此，多核处理器开始登场，比如Sun的Niagara系列，IBM的Power系列，AMD的Opteron，以及Intel的Xeon。事实上，供应商们未来将会提供时钟速度更低但核数更多的处理器，如Intel的Nehalem以及AMD的Istanbul。此外，其他形式的加速器也纷纷登场，例如GPGPU（如Nvidia的Tesla，AMD的Firestream，Intel的Larabee），Cell处理器，以及FPGA。

这就意味着，为了能充分利用多核处理器，我们必须寻找应用内在的并行性，必须要编写具有并行线程的代码。我们已然预见到串行程序会“逐渐减速”，以此观之，似乎并行程序设计的概念终将成为未来主流程序员所必须掌握的重要概念。

《并行程序设计原理》一书对学生以及初学者将颇有裨益。两位作者Calvin Lin和Lawrence Snyder非常细致地将各种原则与当前的并行硬件和软件场景联系了起来，使得本书既能扎根于计算机科学的基础，又能与时俱进。这本全新的可实践且实用的书，必将使得阅读引人入胜且充满趣味。

本书着力介绍一系列不同类型的程序设计工具：如在共享存储器程序设计中介绍了Pthread、Java Threads以及OpenMP；而在消息传递程序设计中将MPI作为局部视图语言的代表，并提及了UPC等PGAS语言，而将ZPL作为全局视图语言的典范；此外还提及了3个最新的并行程序设计语言，即Sun的Fortress，IBM的X10，以及Cray的Chapel。

本书还探讨了一些高级内容：比如“理想”的并行程序设计语言所应遵循的基本原则；并行软件开发的方法学和方式，如在敏捷软件开发中最常用的增量式开发。

本书无论对于积极进取的HPC业内人士来说，还是通用程序员而言，都是一本极具价值的参考书。总之，这本可靠而翔实的书探讨了整个社团所将面临的有关并行化的挑战与问题。

Simon See博士
Sun高性能计算及云计算技术中心主任兼首席科学家
新加坡南洋理工大学兼职副教授
2009年6月

译者序

随着多核体系结构的出现和快速发展，使得并行计算科学的硬件基础设施发生了很大变化，如果把并行硬件基础设施看成是“经济基础”，则其相应的上层并行软件就可以视为“上层建筑”。由于“经济基础”的变化，作为其中重要的“上层建筑”之一的并行程序设计技术，必须进行相应的变化以适应新的“经济基础”。因此如何在多核体系结构上进行高效的并行程序设计以充分利用多核所提供的硬件并行性，从而大幅度地提升并行计算性能指标就显得非常重要。本书的主要论题之一便是环绕这一问题展开的。

并行程序设计比顺序程序设计要困难得多，一是因为并行程序设计的平台不是唯一的，存在多种不同的并行体系结构，而顺序程序设计只有唯一的冯·诺依曼体系结构；二是因为并行程序有多个进程或线程在同时运行，它们之间往往需要进行通信和同步，这就使并行程序设计变得复杂，特别是在要获得线性加速比时尤为如此；三是因为并行程序设计没有顺序程序设计中如C及Java那样通用和普及的并行程序设计语言，因此只能针对不同的体系结构选择使用不同的并行语言或例程库，例如对于共享地址空间的体系结构就必须选择如OpenMP、Java Threads或POSIX Threads那样的语言，而对于分布地址空间的体系结构就不得不选择如MPI或PVM那样的例程库语言。此外若要开发数据并行性，则就需要选用高性能的如Fortran（HPF）那样的语言。本书另一个主要论题便是环绕这一问题展开的。

本书侧重论述并行程序设计的基本原理，解释各种现象，并分析为何这些现象意味着成功进行并行程序设计的机遇或是阻碍。并行的硬件基础设施和并行的软件设计环境随着时间的变迁会不断发生变化，但原理则永远不会过时。以原理作为第一要素进行论述是本书的特色之一。

本书的另一个特色是，它侧重可扩展性和可移植性，即所设计的并行程序具有在任何数目处理器系统上和在任何并行体系结构平台上运行良好的能力。这一概念在多核时代是非常关键的，这是因为：首先，使得并行计算具有可扩展能力的大多数技术与在多核芯片上生成高效求解的技术是相同的；其次，虽然目前的多核芯片所具有的处理器数目还比较小，通常是2～8个，但今后每个芯片上的核数将会急剧增加，这就使得可扩展并行概念与之直接相关；最后，显然我们应该侧重研究和开发那些在现在和将来都能很好工作的方法。

内容非常实用是本书的又一特色。这是因为本书在介绍并行程序设计系统的同时，还叙述如何在这些系统中应用并行程序的设计原理。作者通过自身丰富的实践经验为读者介绍了在从事并行程序设计时应遵循的方法学。译者认为从事并行程序设计者应注重对并行程序设计方法学的了解、掌握，以及有关素质的培养，唯此才能开发出性能良好以及生命力持久的并行程序，并提高编制并行程序的能力和生产率。

翻译本书的原因有两个：一是本书的内容相当新，涉及现代的并行硬件和软件技术，包括多核体系结构及其并行程序设计技术；二是本书论述了并行程序设计中的一些深层次问题，如可扩展性、可移植性以及并行程序设计应遵循的方法学等。本书的不足之处在于对一些性能问题的定量分析不够充实，此外所介绍的并行机抽象模型也不够全面，只有一个CTA模型。但这些瑕疵并不会影响本书的阅读价值。

本书是计算机科学专业本科高年级学生或一年级硕士生的理想教科书，对专业程序员来讲则是从事并行程序设计的一本理想入门书。本书对软件工程师和计算机系统设计师也是非常值得一读的参考书。

本书的翻译工作由陆鑫达教授负责和组织。陆鑫达教授翻译了目录、前言、第1~4章以及第10~11 章，林新华老师翻译了第5~9章。译稿全文由陆鑫达教授统稿审校。原书只分章不分节，为方便读者阅读，我们统一进行了分节。值此中译本出版之际，译者特向机械工业出版社华章公司的策划和编辑人员表示深切的谢意。

书中的术语翻译，我们尽量采用已公布的计算机科学技术名词（第二版），对于一些未公布的术语（包括一些新出现的术语）我们尽量采用流行的译法。由于时间较为仓促，翻译中的错误或不妥之处在所难免，敬请广大读者不吝指正。

致谢

在本书行将付梓之前，承蒙Sun公司高性能计算及云计算技术中心主任兼首席科学家Simon See博士拨冗为本译著撰写了推荐序，在此仅表深切的谢意！

上海交通大学计算机科学与工程系
陆鑫达　林新华
2009年6月10日

前　言

对于那些因多核芯片出现而激发起学习并行程序热情的读者来说，你找对了地方。本书是为并行计算机无处不在的这个世界而编写的，这种并行计算机涉及广泛，从具有两核芯片的膝上计算机，到超级计算机，再到用于搜索因特网的巨大数据中心机群。

本书侧重可扩展并行，即并行程序具有在任何数目处理器上运行良好的能力。这一概念是非常关键的，原因有二：（1）创建可扩展并行计算所需的大多数技术与在多核芯片上生成高效求解的技术是相同的；（2）目前的多核芯片具有适中数目的处理器，通常是2～8个，在未来几年内每个芯片上的核数肯定会急剧增加，这就使得可扩展并行概念与之直接相关。因而，尽管今天的多核芯片为核间低时延通信提供了机遇，但这种特性很可能只能在短期内得益，因为当芯片上的核数增长时，芯片内不同部分的片内延迟将明显增加。所以，我们不会侧重开发这种短期得益的方法，而是侧重那些能在现在和将来都能很好工作的方法。当然，多核芯片还面临着自身的挑战，特别是它们有限的片外存储器带宽，以及有限的片上总的cache容量。本书也将讨论这些问题。

首先，我们讨论构成实用和高效并行程序的原理。为了获取如程序设计那样复杂的能力，学习原理是至关重要的。当然，对于并行程序设计来讲，原理也许更加重要，因为最新技术水平的变化很快。如果训练过于紧密地捆绑于某个特定的计算机系统或一种语言，将不具有跟上技术发展步伐的支撑力。可是原理（应用于任何并行计算系统的概念以及开发这些特征的观念）对人们深入理解和掌握相关知识将提供深远的影响。

但我们并不会止步于对抽象概念的讨论，还会将那些原理应用到日常的计算中，这将使本书非常实用。我们将介绍几个并行程序设计系统，还将描述在程序设计系统中如何应用这些原理。我们期望读者在读完本书后能编写并行程序。在最后一章我们将专门讨论并行程序设计技术，以及介绍如何开发一个有关并行程序设计的结课课程设计，这个设计可能需要历时一个学期。

读者对象

读者对象可以是任何人，大学生或专业人员，只要他能用C或类似语言成功进行编程，以及自认为是程序员的人。如果具有计算机执行顺序程序的概念，包括取指/执行周期以及高速缓存基础的知识，则就更易理解本书的内容。本书最初的定位是计算机科学专业的高年级学生以及具有计算机科学学士学位的一年级研究生，现在本书仍然适合这一程度。然而，作为本书的宗旨，我们减少了对预备知识的要求，而强调对知识的传授，如果读者已经掌握某些解释所涉及的知识，只需跳过这些知识。

本书结构

因为并行程序设计并不是读者所熟悉的顺序程序设计的直接扩展，所以我们将本书分为四部分：

基础：第1～3章

并行抽象：第4～5章

并行程序设计语言：第6～9章

展望：第10～11章

为使读者能明智地从中选择感兴趣的部分，我们下面说明各部分的目的和内容。

基础 在第1章中，我们通过实现一个计算说明并行程序员必须处理的许多问题是多么困难，而在为顺序计算机编写同样的程序时则非常简单。该例子将我们的注意力集中到一些贯穿于全书与我们相关的论题，但它也强调了解并行计算机是如何操作的重要性。第2章介绍5种不同类型的并行计算机，描述了其体系结构中的少量细节，以及它们扩展到更大规模的能力。在这一章中得出了两个关键的结论：第一，不像顺序计算，并行计算机没有一个标准的体系结构。第二，为了能适应这种体系结构的多样性，我们需要一个抽象的机器模型以指导我们的程序设计。我们给出了一个抽象模型。在留意体系结构的基础上，第3章的内容涉及并发性（包括线程和进程）、时延、带宽、加速比等基本概念，侧重叙述与性能相关的问题。第一部分这些基础内容为第二部分讲解算法和抽象做好了准备。

并行抽象 为支持并行算法的设计和讨论，第4章为编写独立于语言的并行程序引入了一个非正式的伪代码符号。该符号具有跨各种程序设计模型和方法的许多特性，允许我们在讨论算法时不会偏向任何具体的语言或机器。为了引导读者考虑并行算法，第5章介绍了一系列的基本算法技术。在阅读完第二部分后，读者将掌握以并行方式求解问题的方法，从而可使我们进入最后一个问题，即以一个具体的并行程序设计语言来编码算法。

并行程序设计语言 还没有一种并行程序设计语言，能够如C或Java在顺序程序设计中所起的作用那样，为大家所熟知和接受，并作为编码算法的基本手段。为此，第三部分介绍了三种并行程序设计语言：基于线程的语言（第6章），消息传递语言（第7章）和高级语言（第8章）。我们所论及的每种语言的内容，足以使读者能编写小的练习；但编写重大的问题将需要更完整的语言介绍，这可通过在线资源获得。除了介绍一种语言，每一章还包含一个有关语言的简短综述，这些语言在并行程序设计社团中有一批追随者。第9章简单地比较和对比了所有已论述的语言，指明它们的优缺点。能通读这三章最好，但是我们知道许多读者将只会定格在一种方法上，因此这三章是相互独立的。

展望 第四部分是展望未来。第10章涉及一系列新的、有前途的并行技术，它们无疑将影响未来的研究和实践。我们的观点是，这些技术还未为它们的“全盛时期做好准备”，但它们是重要的，且值得我们去熟悉它们，即使在它们还未被完全部署之前。最后，第11章侧重论述并行机程序设计的第一手实践技术。该章的前两节应在研究并行程序设计之前阅读，也许与第4章和第5章的抽象一起学习更好。但该章的主要目的是帮助读者编写作为结课课程设计内容的一个真实程序。

本书使用方法

虽然本书的内容以逻辑顺序排列，但也不必从头到尾阅读本书。的确，作为一个学期的课程，在学完所有论题之前就开始程序设计的练习，这可能是一个明智的选择。请看如下的一个明智的学习计划：

- 第1章和第2章
- 11.1节，3.4节；开始程序设计练习
- 第4章和第5章

- 第6～8章（关于程序设计语言）之一
- 完成第3章和第11章，然后开始学期课程设计
- 依次完成以下各章：关于语言的各章，第9章和第10章

当然，从头到尾阅读本书并无坏处，但上述方法的优点是阅读和程序设计可并行进行。

致谢

要真诚地感谢E Christopher Lewis 和Robert van de Geijn两位对本书初稿的评论。也要对以下评阅人的宝贵反馈意见和建议表示感谢：

David Bader, 佐治亚技术学院

Purushotham Bangalore, 亚拉巴马大学伯明翰分校

John Cavazos, 特拉华大学

Sandhya Dwarkadas, 罗切斯特大学

John Gilbert, 加州大学圣巴巴拉分校

Robert Henry, Cray公司

E Christopher Lewis, VMWare

Kai Li, 普林斯顿大学

Glenn Reinman, UCLA（加州大学洛杉矶分校）

Darko Stefanovic, 新墨西哥大学

我们感谢Karthik Murthy和 Brandon Plost两位在编写和运行并行程序中的帮助，以及帮助发现正文中的瑕疵，我们还要感谢Bobby Blumofe, 他与我们在多线程程序设计课程中的早期合作在本书的许多地方都可看到。我们赞赏和感谢华盛顿大学2006年秋季学期并行程序设计环境讨论班（CSE590o）的学生们对本教科书的贡献，他们是：Ivan Beschastnikh，Alex Colburn，Roxana Geambasu，Sangyun Hahn，Ethan Katz Bassett, Nathan Kuchta, Harsha Madhyastha, Marianne Shaw, Brian Van Essen, Benjamin Ylvisaker。Sonja Keserovic, Kate Moore, Brad Chamberlain，Steven Deitz, Dan Grossman, Jeff Diamond, Don Fussell, Bill Mark和David Mohr。

我们要向编辑Matt Goldstein以及Addison Wesley出版社团队成员Sarah Milmore、Marilyn Lloyd、Barbara Atkinson、Joyce Wells和Chris Kelly表示感谢。谢谢Gillian Hall 对我们愚钝行为的特别容忍。

最后，我们要感谢我们的家人在撰写本书期间对我们的容忍。

Calvin Lin

Lawrence Snyder

2008年2月

目　录

第二部分 并行抽象

第三部分 并行程序设计语言

第四部分 展 望

第一部分 基 础

学习并行程序设计先要打好结实的基础。基础中最重要的就是要能分辨顺序程序设计和并行程序设计两者的差异。在顺序计算中，每次只能完成一个操作，使得判断一个程序的正确性和它的性能特性非常直截了当。在并行计算中，许多操作同时进行，使得对程序的正确性和性能的判断变得复杂，因此需要修改我们的程序设计方法。本篇将阐明这一差异所带来的主要后果。

第1章是并行计算的导论，从求解在一维数组中统计3出现次数的一个简单问题开始。这个任务看似普通，在创建一个有合理性能的程序之前，却需要进行四次尝试。不仅如此，我们发现对加速比的最大期望无法实现。在讲解这一例子时，我们将介绍许多并行的基本概念。

第2章描述并行计算机的基本体系结构特征。就其本身而言，便是一个饶有兴趣的主题，因为像处理器间通信这样富有挑战性的问题存在许多的可能解答，而且体系结构师所使用的技术具有很强的独创性。对各种并行机进行考察得出的结论是它们是截然不同的。由于程序员需要知晓机器低层的一些特性才能编写高质量的程序，因此有必要寻找一个能统一各种不同体系结构的机器模型。我们将引入这样的一个模型作为后面研究的基础。

为了清晰地了解并行计算机是如何工作的，在第3章中从概念性角度剖析了许多有关并行性能的问题。我们介绍的主要概念包括时延、带宽、加速比和效率。在程序设计方面，如其中的相关性，将作为并行线程的干扰源进行重点介绍。一旦掌握了所介绍的这些基本概念，就为学习第二部分中的算法思想做好了准备。

第1章 导 论

并行计算是一种能加速计算的基本技术，因此不断增长的并行硬件可用性隐含着巨大的机遇。但是实现一个并行求解意味着对某些概念和程序设计的挑战，这正是本书试图要解决的。为了正确地权衡机遇和挑战，本章将提出与并行程序有关的问题并介绍基本概念。

1.1 并行的威力和潜能

并行在日常生活中时常碰到。更为重要的是，在过去的几十年中并行以各种方式为计算机性能的稳定提高做出了贡献。现在新的机遇正在出现。下面让我们来详细论述这一点。

1.1.1 并行，一个熟悉的概念

并行是我们熟悉的一个概念。杂耍（juggling）是人类能完成的一个并行任务。房屋建造是一种并行活动，因为几个工人能同时完成不同的工作，如电线配线、水管安置、锅炉管道安装等等。大多数的工业产品如汽车、吹风器、速冻食品，都以流水线方式进行生产，在流水线上正在建造的许多单件产品是同时进行加工或装配的。呼叫中心则是另一种应用并行的结构，其中有许多雇员在同时为顾客服务。

虽然熟悉，但应注意这些并行形式是不同的。例如呼叫中心在本质上与房屋建造有所不同：呼叫通常是独立的，因此能以任意次序提供服务，而且工作人员之间几乎没有交互。而在建房时，某些任务能同时完成，如电线配线和水管安置，而另外一些任务则必须依次进行，例如配线架必须在配线之前进行安装。这种顺序限制了能同时进行的并行量，从而也就限制了一个建设项目完成的进度。顺序性也增加了工人之间的交互程度。制造业的流水线与前两者又有所不同，因为它们有严格的顺序约束，制造产品的各个阶段通常以顺序方式进行；并行性来自于流水线上同时在生产许多单件产品。杂耍则属于事件驱动的并行，当一个事件（一个落下的球）发生时将引起有关操作的执行（抓球、抛球）以响应该事件。这些熟悉的并行形式也将出现在我们要讨论的并行计算中。

1.1.2 计算机程序中的并行

以并行方式执行程序指令的主要动机是为了加快计算。但是现今的大多数程序是无法通过并行来改进性能的，因为它们的编写是假设指令是按序执行的，每次执行一条指令，即是顺序执行的。大多数程序设计语言的语义体现为顺序执行，按这种语义编写的程序通常严重地依赖于这种特征来保证正确性，因而很难有机会实现并行执行。当然，还存在一些机会，例如在计算表达式(a+b)*(c+d)时，假定这些都是简单变量，由于子表达式(a+b)和(c+d)相互独立，因此它们能同时计算。这就是指令级并行（Insruction Level Parallelism，ILP）的例子。

的确，我们继续编写顺序程序的一个理由是因为计算机体系结构师们在开发并行性方面如此成功。他们利用硅工艺的稳定改进，在顺序处理器的设计中采用了多种并行方法，其中包括ILP。首先，体系结构师们为指令和数据提供了分离的路径和高速缓存（cache）。这种分

离允许对指令和数据存储器进行并行访问，两者互不干扰。其次，使指令执行流水化，在执行当前指令的同时，对后继的指令进行取指和译码，而前面的指令则正在将结果写回存储器。更进一步，处理器每次可发出（启动）多条指令，预取指令和数据，以猜测方式并行地完成操作，即使不能保证是否真的需要进行这些操作，他们还使用高速并行电路完成基本的算术操作。简言之，现代处理器是高度并行的系统。

对程序员来讲，所有这些并行对顺序程序都是透明可用的。我们称这种并行为*隐藏并行*(hidden parallelism)。这种并行，加上不断提高的时钟速度，使得新一代的处理器芯片总能以更快的速度执行程序，而同时又保持顺序执行的假象。但是要在维持顺序语义的基础上期望找到应用并行的新机遇是非常有限的。更为严重的是，就功率消耗和性能两者而言，开发ILP的现有技术在很大程度上已经到达了它的收益递减点。因而，就当前的技术而言，顺序程序的执行可能已接近它的最大速度。

要想继续获取显著的性能改进，必须超越现有程序典型的单指令序列的工作方式。我们必须使程序以同时运行多指令流的方式工作。这种工作方式需要使用新的程序设计技术，这就是本书要讨论的主题。

1.1.3 多核计算机，一个机遇

虽然单处理器的性能改进可能已接近极限，但摩尔定律的预言使得晶体管密度仍在不断提高。芯片制造商利用这一机会，在单个芯片上放置多个指令执行引擎以及相应的高速缓存。这种结构很快得到新的名称“核”(core)，因为它代表了一个典型顺序处理器的核心部分。早期的多核芯片有2个、4个或8个核，但是核的数目随着新一代处理器的出现在不断增长。

于2005/2006年问世的第一个多核芯片引起了全社会关于“免费午餐结束”的讨论。讨论的焦点如下：

- 由于如前所述的硅工艺和体系结构设计（隐藏并行）的进步，软件开发者几十年来一直享受稳定的性能改进——“免费午餐”。
- 由于无须关心性能，多年来程序员对他们使用的技术和方法学几乎没有什么改变（面向对象的范例是一个值得一提的例外）。
- 现有的软件通常不能直接利用多核芯片。
- 不能利用多核芯片的程序在目前和未来都无法实现性能改进。
- 大多数程序员不知道如何编写并行程序。

令人不快的结论是程序需要改变，为此程序员也必须随之改变。

虽然上述的结论在某些人看来可能是一个坏消息，但同时还有相应的好消息。特别是，如果将一个计算重写成并行形式，而且如果该并行程序是*可扩展的*（表示它能够使用日益增多的处理器），那么随着硅工艺的进步，随之会有更多的核加到未来的芯片上，则已重写的程序的性能将持续上升。但对不可扩展的并行程序来讲，将不能享受硅工艺进步的好处。因此，获取可扩展并行至关重要。

某些特别是在图形学领域的研究人员，无疑对是否需要并行计算的讨论感到非常奇怪，因为他们多年来一直在使用并行性。图形处理部件（Graphics Processing Unit，GPU），又称为图形卡，已经是加速绘制（rendering）流水线的标准技术。虽然GPU看起来像是一个小巧的协处理器，对通用计算机应用来讲微不足道，但硅工艺的进展已使得*在图形处理部件上进行通用计算*（General Purpose computing on Graphics Processing Unit，GPGPU）这一概念成

为可能。大约18个月的代周期，使得GPU随着每一代的发展正在逐步变得更为通用，并行程序员已将它们应用到许多非图形处理的密集计算场合。如同开发多核芯片一样，开发GPU的潜在能力需要并行程序设计的知识。

1.1.4 使用并行硬件的更多机遇

到目前为止所讨论的使用并行的机遇只涉及少量的处理器。但实际上还存在许多更具雄心的机遇。

超级计算机　国家研究实验室、军事部门以及大公司所感兴趣的求解问题通常需要使用超级计算机，所谓超级计算机是指当今世界上最快的计算机。20年前，超级计算机是用户定制的单处理器系统（一般具有向量处理能力），但是1996年11月在500强（Top 500）最快计算机名单中最后出现的单处理器系统共有3台，排名仅分别为第265位、第374位和第498位。现在的500强名单则为具有数以千计的处理器的并行计算机所占据。在许多方面，超级计算机程序员形成了最大和最富经验的并行程序员社区。

机群　常常可以观察到，不论单计算机有多快，将两个或多个计算机连接起来可以组成一个更快的计算机，因为组合的机器在单位时间内能执行更多的指令。当然，需要编写优质的并行程序以开发额外的计算能力。机群自20世纪90年代开始流行，因为用商品化部件构成的机群比较便宜。机群的低价格不但吸引了众多的小团体（实验室或小公司），而且与其他高端计算形式相比，机群具有很高的性能价格比。2007年6月公布的500强名单中（参见www.top500.org），机群占了74.6%。

服务器　因特网的扩张和远程服务（如搜索）的普及促成了大量的连网计算机。就每秒所执行的总指令数而言，这些服务中心代表着巨大的计算资源。典型的计算（如搜索查询处理）是相互独立的；此外，它们使用分布（相对于并行）的程序设计技术（参见1.1.5节）。无论如何，这些庞大的连网系统正被用来分析其工作负载特征，以及完成其他的数据密集计算；其解决方法也是需要应用并行程序设计技术。

网格计算　更进一步的通用化将使得计算机的集合无须在同一场所，也不需要由同一个机构加以管理；毕竟由因特网连接的计算机代表着巨大的计算资源。类似于电力网格，计算网格寻求一种能提供方便的计算服务的机制，即使低层的计算机是由物理上分散的机器组成，而且这些机器由多个行政部门分别管理。在网格普及之前，还存在许多技术问题，但这是一个活跃的研究领域。

由上可知，除了可以在单芯片的少量处理器上使用并行程序之外，还存在相当多使用并行程序的机会。这些大型计算机系统也激发起我们编写可扩展并行程序的冲动。

1.1.5 并行计算和分布式计算的比较

如前所述，分布式计算和并行计算是不同的。

并行计算的传统目的是提供单处理器无法提供的性能（处理器能力或存储器）；因此，它的目的是使用多处理器求解单个问题。而分布式计算的目的主要是提供方便，这种方便包括可用性、可靠性以及物理的分布（能从许多不同场所访问分布式系统）。

在并行计算中，处理器间的交互一般很频繁，往往具有细粒度和低开销的特征，并且被认为是可靠的。而在分布式计算中，处理器间的交互不频繁，交互特征是粗粒度，并且被认为是不可靠的。并行计算注重短的执行时间，分布式计算则注重长的正常运行时间。

当然，并行计算和分布式计算两者是密切相关的。某些特征与程度（处理器间交互频率）

有关，而我们还未对这种交叉点（crossover point）进行解释。另一些特征则与侧重点有关（速度与可靠性），而且我们知道这两个特性对并行和分布两类系统都很重要。由此可知，这两种不同类型的计算在一个多维空间中代表不同但又相邻的点。对并行计算（或分布式计算，但它不是本书的论述重点）越了解，在并行-分布空间中就越游刃有余。学习并行计算的基础对于无须改进性能的程序员来讲也是非常有用的。

1.1.6 系统级并行

让我们暂且回到先前的论点，即为了享受并行的好处，我们必须不受单指令序列的约束。这一论点仅针对单个应用。然而，当我们从系统级观察桌面机的软件时，会发现许多任务是并行执行的。操作系统控制它们的并发执行，在这里它意味着同时处理几个任务，但每次只执行一个任务，这种技术也称为多任务化（multitasking）。[⊖]一个显而易见的问题是，“为什么不直接将这些分离的任务在额外的处理器上运行？”这是有道理的，因为它们的并发设计能保证在它们之间不论发生何种交互都能安全地加以处理。

并发和并行 尽管这两个术语紧密相关，但历史影响了我们如何使用它们。“并发”一词在操作系统中广泛使用，数据库社区将其描述为逻辑上同时执行，而“并行”一词则通常在体系结构和超级计算社区中流行，用来描述物理上的同时执行。不论是哪一种情况，同时执行的代码都显示出未知的定时特征。例如，操作系统在某时刻可能正执行一个代码段，但是因为执行能在任意时间点切换到另一个代码段，因此所遇到的问题与物理上同时执行时是一样的。类似地，对于物理并行，代码段间的定时关系是不可预测的，强制设定任何定时都是可能的。因此，就程序正确性问题而言，对并发计算和并行计算来讲是一样的。从某种意义上讲，它们是各种不同并行资源的两个端点。本书中，我们将交替使用这两个术语来表示逻辑并发。

对于刚才提及的大规模并行，第一个回答是，没有足够多的任务能使大量处理器处于繁忙状态。而对于小规模的并行问题，如目前典型的多核芯片，单独的任务能在不同的处理器上运行。的确，已经有人提议，可以在这些额外的处理器上连续地执行任务（例如安全性软件，或是操作系统本身，就是这种任务很好的候选）。但是，实际上存在的并行机会比最初看上去要少。这是因为，首先，许多应用即使现在也不在乎硬件，字处理器连续地在后台进行拼写检查，它永远也不会落在打字员后面。其次，操作系统中的大多数多任务都源自正在执行的任务请求一个耗时的外部操作时（如缺页、磁盘I/O或网络I/O）切换到一个新任务。在单处理器系统中，当任务A阻塞于这种请求时，可以执行另一个任务B，从而极大地提高了利用率；但是在一个多处理器系统中，任务B可能已在另一个处理器上执行完毕，从而迫使执行任务A的处理器只是处于闲置状态。

然而，在单独的并行处理器上运行多个任务并不是一个好主意，主要原因在于它通常并不能改善单个应用的性能。因此在确实需要改善性能的情况下，关键是要有多个协同指令流有效地使用多处理器。

1.1.7 并行抽象的便利

最后，除了性能以外，开发并行性还有一个理由：某些计算更容易表示成并行计算。例

⊖ 由于某些I/O设备（如磁盘控制器）通常是与主要执行引擎分离的，因此长期以来，实际上处理器与它的外部设备是并行执行的。

如，用户界面通常最好编写为线程的集合，其中一个线程负责与用户交互：该线程等待用户的输入，并分派其他线程给出适当的响应。这样的结构可以大大简化显示小窗口部件的代码，例如，它无需负责轮询用户是否点击鼠标键，而这种点击可能在任何时间发生。

如我们将见到的那样，用于组织和管理并行计算的抽象将使多指令流的使用方便而安全。在控制流不可预测时，纵然结果的指令序列未同时执行，并行仍有帮助。因此，在我们强调快速求解问题的时候，应意识到并行还有其他的用途。

1.2 考察顺序程序和并行程序

在前面几小节中我们着重论述了硬件具备的潜在优势。我们还断言现有的顺序程序不能利用多核计算机，因此现在必须考虑能实现并行硬件优点的方法。

1.2.1 并行化编译器

我们中的许多人知道编译器能将我们编写的程序翻译成所使用计算机的机器指令，但并不了解编译器是如何完成这一奇妙翻译的，因此一个合情合理的想法是，为什么不直接编写一个能将现有顺序程序翻译成适合并行执行的编译器。毕竟顺序程序已经指定了计算，剩下需要做的只是将相同的操作翻译成为并行形式。为并行机编译顺序程序的思想是最初尝试的方法之一，至今它仍是一个梦想。遗憾的是，尽管已进行了30多年的认真研究，这个梦想仍未实现。

持有这种悲观论调的理由是，可扩展并行算法通常与现有程序中的顺序算法在本质上是不同的。我们将通过一个范例来描绘如何使求解方法从顺序方式逐步演变为并行方式。由于编译器在转换程序过程中保持了程序的正确性，所以，编译器不会改变算法的基本特征。（图1-1说

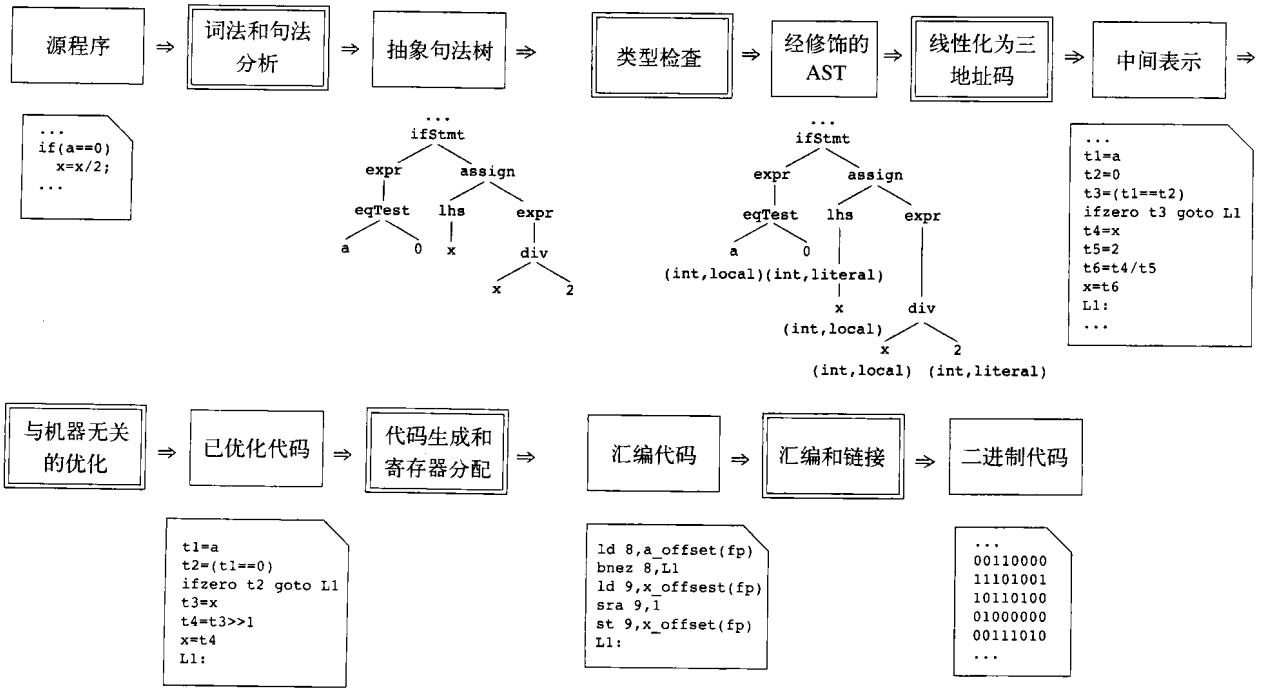


图1-1　通用编译过程。在第1阶段，扫描（词法分析）和解析（句法分析）源程序，生成一个称为抽象句法树的程序表示。在此基础上进行类型检查以确保诸如变量已声明。在第2阶段，程序被转换成3地址代码的线性指令序列。然后，对这种中间表达式进行改进（称为优化）。再将所生成的优化代码转换成特定于机器的汇编代码。此后，再把它转换成二进制代码，并指派虚地址

明了一个通用编译器的工作流程)。编译器改变程序代码的形式；可以删除不需要的指令，例如对变量加零；能添加辅助指令，检查数组索引是否在规定的范围以内；还能移动指令，例如将它们移到循环外，只要这些指令的计算值不受迭代的影响；此外，编译器还能进行其他神奇的转换。但通用的算法仍保持不变。不论源代码是顺序的还是并行的，目标码形式的算法基本上仍保持原样。

尽管由编译器自动完成并行化很不错，但我们必须考虑其他的并行化方法。首先考虑同一任务的顺序和并行算法有什么不同。

1.2.2 范例求解的变化

为了弄清顺序算法和并行算法的不同，我们将比较求一串数总和的不同算法。由于这个例子十分简单，确实有一些编译技术能加以识别，并能生成更并行的解。但我们仍选择它，理由在于它能对顺序解和并行解之间概念性的差异进行简洁的说明。

我们假定该序列有n个数据值，

$$x_0, x_1, x_2, \cdots, x_{n-1}$$

且开始时它们存放在数组x中。

迭代求和 也许，最直观的求解方法是将一个名为sum的变量初始化为0，然后，迭代相加序列中的元素。这种计算方式通常使用一个循环进行编程，用索引值访问序列中的元素，如下所示：

```
1  sum = 0;
2  for(i=0; i<n; i++)
3  {
4    sum+=x[i];
5  }
```

该计算可抽象成一个图，说明组合数字的顺序（参见图1-2）。这种求解方法是直观的，因为大多数人开始学习编程时就采用这种方式。

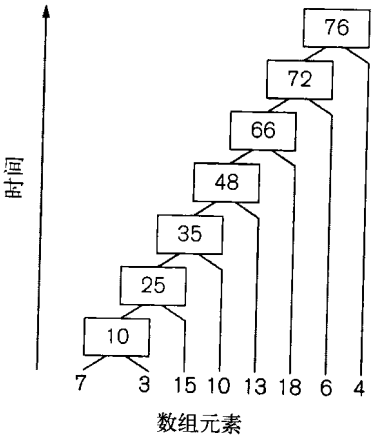


图1-2 顺序求和。将数值序列（7, 3, 15, 10, 13, 18, 6, 4）中的数加到累加变量中时的组合次序

当然，实数的加法是可结合和交换的操作，这就意味着在求和时，不必按从最小的索引值到最大索引值的次序进行。我们可以按其他的次序将它们相加并得到相同的结果，也许这种相加的次序能提供更多的并行性。

非结合性 严格地讲，用固定精度表示的浮点数完成加法时，不具有结合性，因为它们只是近似的实数。因而对于某些数值序列，不同次序的加法将产生不同的结果。我们将忽略这一点，并允许重排计算的次序以改进性能。理由是：a) 在大多数情况下，序列的原始次序就是任意的，b) 在序列的原始次序不是任意的且必须考虑数值精度时，则在整个计算过程中需要对误差进行控制。

成对求和 另一种具有更高并行度的求和方法是将数据值的偶/奇对相加，产生以下的中间和，

$$(x_0 + x_1), (x_2 + x_3), (x_4 + x_5), (x_6 + x_7), \cdots$$

然后将它们再成对相加，

$$((x_0 + x_1) + (x_2 + x_3)), ((x_4 + x_5) + (x_6 + x_7)), \cdots$$

生成进一步的中间和，然后再进一步成对相加，依此类推。这个求解方法可以看成是生成一棵计算树，其中，原始数据值是叶子，中间结点是其下面的结点之和，根是总和（见图1-3）。

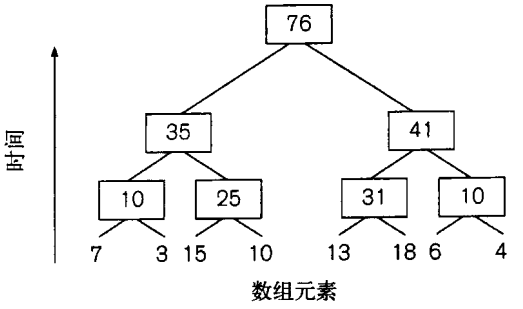


图1-3　成对求和。用递归方法组合数值序列（7，3，15，10，13，18，6，4）的次序，先组合数值对，再组合结果对，依此类推

比较图1-2和图1-3，可以看到，由于这两个求解方法需要相同数目的操作数和中间和，因此当使用一个处理器时，其中任何一个解都没有时间上的优势。然而，如果一个并行计算机至少有$P = n/2$个处理器时，则处于计算树上同一级的所有加法就可以同时进行，从而使求解的时间复杂性与$\log n$成正比。与线性时间的顺序算法相比，成对求和策略在性能上有显著的改进。与顺序求解方法一样，成对求和方法也是一个非常直观的考虑计算的方法。

并行和的表示　迭代求和已经用C语言代码加以说明，但成对求和却没有。如果不考虑所写的代码是针对任意长数组的，那么可以用如下的表达方式突出计算的二叉树结构：

```
1  t[0]=x[0]+x[1];
2  t[1]=x[2]+x[3];
3  t[2]=x[4]+x[5];
4  t[3]=x[6]+x[7];
5  t[4]=t[0]+t[1];
6  t[5]=t[2]+t[3];
7   sum=t[4]+t[5];
```

前4个赋值语句能并行完成；在它们完成之后，后两个赋值语句（5，6）也能并行完成。

1.2.3 并行前缀求和

与求和紧密相关的操作是前缀求和，在许多并行程序设计语言中也称为*扫描*（scan）。与求和操作一样，首先仍有n个值的序列，

$$x_0, x_1, x_2, \cdots, x_{n-1}$$

但希望计算的是如下的序列，

$$y_0, y_1, y_2, \cdots, y_{n-1}$$

其中，每个y_i是输入的前i个元素的和，即有，

$$y_i = \sum_{j \le i} x_j$$

以并行方式求解前缀和不如求累加和那样明显，因为它需要顺序求解所有中间值。初看起来，好像前缀求和既没有优势，也不可能找到更好的解，但事实上前缀求和能以并行方式完成。

通过对成对求和方法的观察，可以发现只要对该方法略加修改就可以计算前缀值。求解的思路是，每个存储有x_i的叶处理器能够计算值y_i，只要它知道在它左边所有元素的和，即它的前缀；在成对求和的过程中，我们知道所有子树的和（参见图1-3），而如果能保留这些信

息，就能确定这些前缀而无需直接对它们求和。为做到这一点，我们从根开始，它的前缀（即在序列元素之前的所有元素和）是0。这也是它的左子树的前缀，而它的左子树的总和则是它的右子树的前缀。归纳地应用这一思路，我们可以得到如下规则：

- 如前所述，在根结点用成对求和方法计算出总和。
- 结束后，假想根从它的父结点（实际不存在）处接收一个0。
- 所有的非叶子结点从它的父结点处接收一个值，将该值转发给它们的左子结点，并将父结点值与它们的左子结点值（这些值在向上成对求和时已经得到）的和发送给右子结点；这些值也即是其子结点的前缀。
- 叶子结点将来自上面的前缀值与它所保存的输入值相加。

沿树向下移动的值是子结点的前缀（参见图1-4，其中向下移动的前缀值用白方框表示）。

这种计算称为并行前缀计算。它在树中进行一次向上和一次向下扫描，但扫描中处于每一层上的所有操作可同时完成。因此，在每个结点上最多只需要进行两次加法，一次向上和一次向下，加上路由的逻辑操作。由上可见，并行前缀求和具有对数的时间复杂性。许多类似的顺序操作在这一方面要差于并行前缀方法。

顺序和并行算法的根本差别在于构造并行算法时需要改变计算的次序。

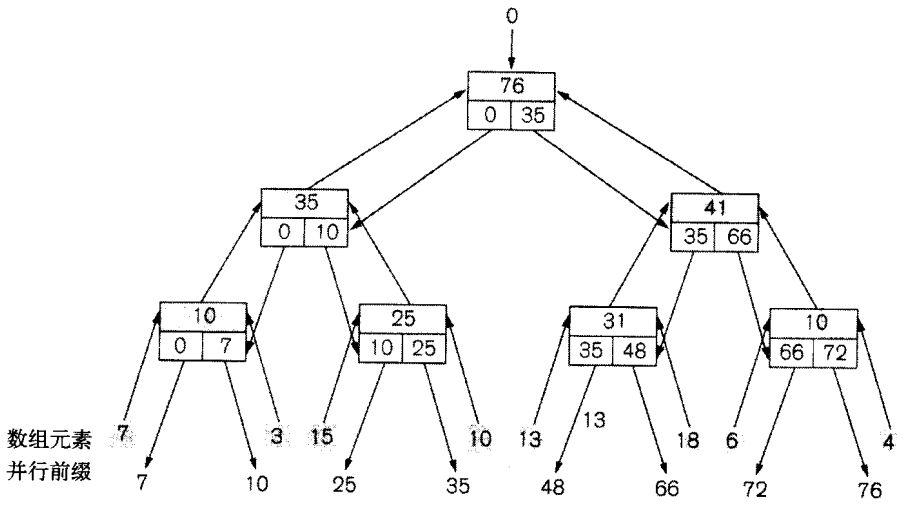


图1-4 前缀和的计算。其中，灰色结点值是沿树向上扫描，用成对求和算法计算得到的值；白色结点值是前缀，沿树向下扫描，用以下简单规则得到：将来自父结点的值送往左子结点，而将来自左结点的和（上传而来）与来自父结点的值两者相加，并将其结果送往右子结点

1.3 使用多指令流实现并行

本节我们将通过开发求解一个简单问题的并行程序来说明并行程序设计的复杂性。通过4次尝试后我们方能得到一个满意的结果。

首先介绍一种描述指令流概念的方法。

1.3.1 线程概念

线程，或执行线程，是一个并行的单位。正如我们将在第3章中讨论的，一个线程具备执

行一个指令流所需的一切（包括一段私有的程序正文，一个调用堆栈，以及一个程序计数器），但它与其他线程共享对存储器的访问。因此多个线程能在全局数据上进行协同计算。

例如，前面讨论的迭代求和循环可以作为一个线程的基础，如果我们将该循环重写成如下形式：

```
1  for(i=start; i<end; i++)          注意：非完整的求解
2  {
3    sum+=x[i];
4  }
```

循环索引i是调用堆栈的局部变量，而变量sum和数组x是共享变量。只要为每一个线程赋予一组不同的start和end值，多个线程就能同时工作，这就是一种并行。

1.3.2 统计3的个数的多线程求解方法

为了了解编写正确、高效和可扩展的线程式程序的困难之处，考虑在一个数组中统计3的个数的求解问题。这个计算在大多数顺序程序语言中都能很简单地表示；那么，当用线程来求解时还需要什么？

并行计算机　为使问题更为具体，我们假设将在有8个处理器的并行计算机上执行我们的并行程序，如图1-5所示。假定两个处理器的标号分别为P0和P1。每个处理器有与其相邻的一个标号为L1 的私有cache。运行程序时，与随机存储器RAM相比，cache是一种快速存储指令和数据的存储器。每一对处理器（P0和P1，P2和P3，等等）共享一个第2级的cache，与L1 cache相比，它有更大的容量和稍慢的速度。最后，8个处理器共享一个第3级的cache，而与L2 cache相比，它有更大的容量，速度更慢，但它仍比RAM的工作速度要快。借助在L2和L3 cache中进行信息交换，可在8个处理器间实现数据共享。

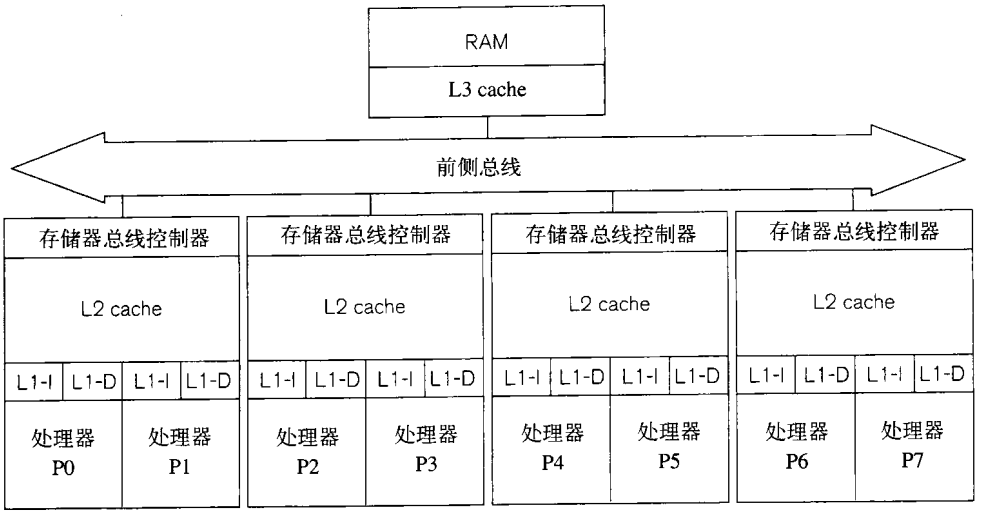


图1-5　运行实验的多核计算机系统的结构。每个处理器有一个私有的L1 cache；与“芯片友”共享一个L2 cache，并与其他的处理器共享L3 cache

第1个求解：尝试1　我们将使用一个线程程序设计模型，其中，每个线程运行在一个专用的处理器上，而线程间的通信通过共享存储器（包括cache）完成。因此，每个线程有自己的进程状态，但所有进程共享存储器和文件状态。统计3的个数的串行代码如下：

```
1  int *array;
2  int length;
3  int count;
4
5  int count3s()
6  {
7    int i;
8    count=0;
9    for(i=0; i<length; i++)
10   {
11     if(array[i]==3)
12     {
13       count++;
14     }
15   }
16   return count;
17 }
```

为实现该代码的一个并行版本，我们可划分该数组，以使每个线程负责1/t数组的统计3的个数的工作，其中t是线程数。图1-6示出了当t = 4和长度 = 16时是如何划分工作的。

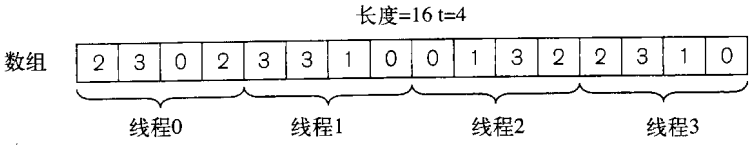


图1-6 向线程分配数据的示意图。按连续的索引进行分配

我们可以用函数thread_create( )来实现这一逻辑，该函数有两个参量：要执行的函数名和标识线程ID的一个整数。它将派生一个线程以执行所指定的函数，并以线程的ID作为一个 参数。图1-7中给出了该求解的结果程序。

```
1  int t;                    /*线程数*/
2  int *array;
3  int length;
4  int count;
5
6  void count3s()
7  {
8    int i;
9    count = 0;
10   /* 创建7个线程 */
11   for(i=0; i<t; i++)
12   {
13     thread_create(count3s_thread, i);
14   }
15
16   return count;
17 }
18
19 void count3s_thread(int id)
20 {
21   /* 该线程应完成的数组计算部分
        should work on */
22      int length_per_thread=length/t;
23      int start=id*length_per_thread;
24
```

图1-7 用线程求解统计3的个数的第1个尝试

```
25    for(i=start; i<start+length_per_thread; i++)
26    {
27      if(array[i]==3)
28      {
29        count++;
30      }
31    }
32  }
```

图1-7 （续）

遗憾的是，这段表面上很简单的代码并不会生成正确的结果，因为在第29行递增（加1）count的语句中存在竞态条件（race condition）。当执行结果依赖于两个以上的事件时，就会出现竞态条件。这里之所以会出现竞态条件，是因为递增count的语句在现代机器上通常实现为以下的一串简单指令：

- 装载count到一个寄存器
- 递增count
- 将count写回存储器

这样，当两个线程执行count3s_thread()代码时，上述的指令可能交叉执行，如图1-8所示。交叉执行的结果使得count的值是1而不是2。当然，可能还存在许多其他的交叉执行，其中某些会产生正确的结果，而另一些可能会产生不正确的结果，其根本的问题在于，对count的递增不是一个原子操作，即它是可中断的。

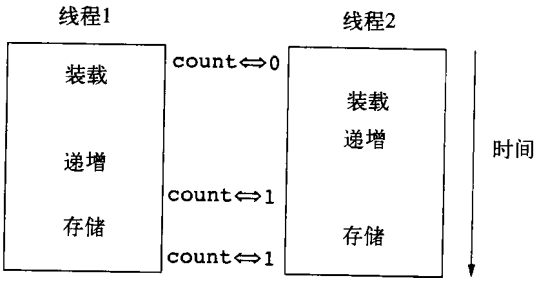


图1-8 对未保护变量count访问时可能出现的一种交叉，表示一种竞态条件

注：符号⇔表示拥有该值。

第2个求解：尝试2 我们可以使用提供互斥功能的mutex（互斥体）解决上面的问题。mutex是一个具有两个状态（上锁和开锁）和两种方法（lock()和unlock()）的对象。这些方法的实现将保证，当一个线程企图上锁一个mutex时，它首先要检查该mutex是否已上锁。如果已上锁，它就等待直到该mutex处于开锁状态后它再将其上锁。通过使用mutex，我们就可以对希望以原子方式执行的对象加以保护，通常称被保护的代码为临界区（critical section），可以保证，在任何时间只有一个线程能对它进行访问。对于统计3的求解问题，我们只需简单地在递增count之前，先将mutex上锁，完成递增之后再对mutex开锁，这样我们就可得到第2个解答（参见图1-9）。

术语 互斥和原子性是一对用来描述不可中断转换的相关术语。

互斥 一段代码以互斥方式执行是指在任何时间最多只有一个线程能执行该段代码。

原子性 术语原子性源自数据库社团。如果一组操作要么全执行，要么全不执行，则它就是原子的。因此，在原子操作中不可能看到部分结果。

```
1  mutex m;
2
3  void count3s_thread(int id)
4  {
5    /* 该线程应完成的数组计算部分 */
6    int length_per_thread=length/t;
7    int start=id*length_per_thread;
8
9    for(i=start; i<start+length_per_thread; i++)
10   {
11     if(array[i]==3)
12     {
13       mutex_lock(m);
14       count++;
15       mutex_unlock(m);
16     }
17   }
18 }
```

图1-9 求解统计3的个数的第2个尝试在count3s_thread()中对count变量提供互斥体保护

经过上述修改后，我们的第2个尝试就是一个正确的并行程序。遗憾的是，我们能从图1-10所示的性能直方图中看到，求解的速度比最初的顺序代码慢很多。当用一个线程时，其执行时间竟是顺序代码的4倍多，可见使用互斥体的开销使性能受到很大影响。更糟的是，当我们使用2个线程时（每个线程运行在自己的处理器上），所得到的性能比只使用单个线程还要坏；这是由于锁竞争而导致的性能下降，因为每个线程要花费额外的时间等待临界区的开锁。

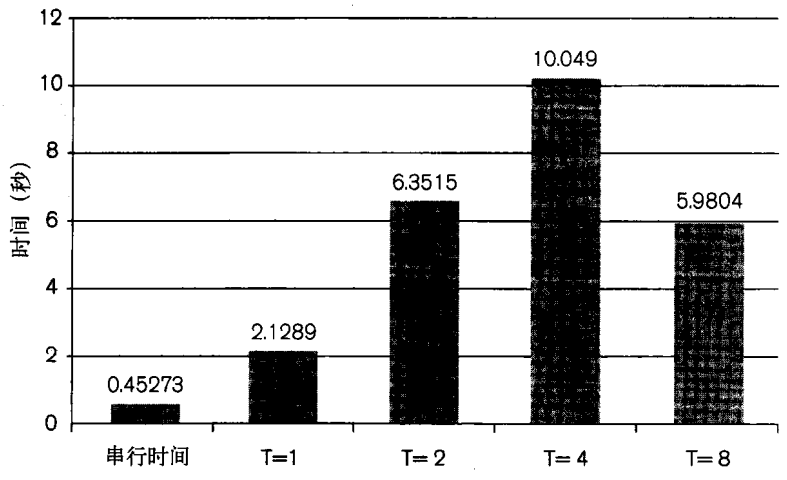


图1-10 第2个统计3的个数的求解方法的性能

第3个求解：尝试3 在了解了锁开销和锁竞争后，我们用第3个版本来实现求解，这次我们采用更大的粒度或共享单位进行运算。我们不让线程在每次递增count时去访问临界段，而是在每个线程的私有变量private_count中累加各自的3统计次数。图1-11中显示了第3种解答的新代码。

以增加少量的额外存储器为代价，现在新程序的运行速度显著加快。如图1-12所示。

```
1  private_count[MaxThreads];
2  mutex m;
3
4  void count3s_thread(int id)
5  {
6    /* 该线程完成它数组计算部分*/
7    int length_per_thread=length/t;
8    int start=id*length_per_thread;
9
10   for(i=start; i<start+length_per_thread; i++)
11   {
12     if(array[i] == 3)
13     {
14       private_count[id]++;
15     }
16   }
17   mutex_lock(m);
18   count+=private_count[id];
19   mutex_unlock(m);
20 }
```

图1-11　在第3种统计3的个数的求解方法中，count3s_thread（）使用private_count数组元素

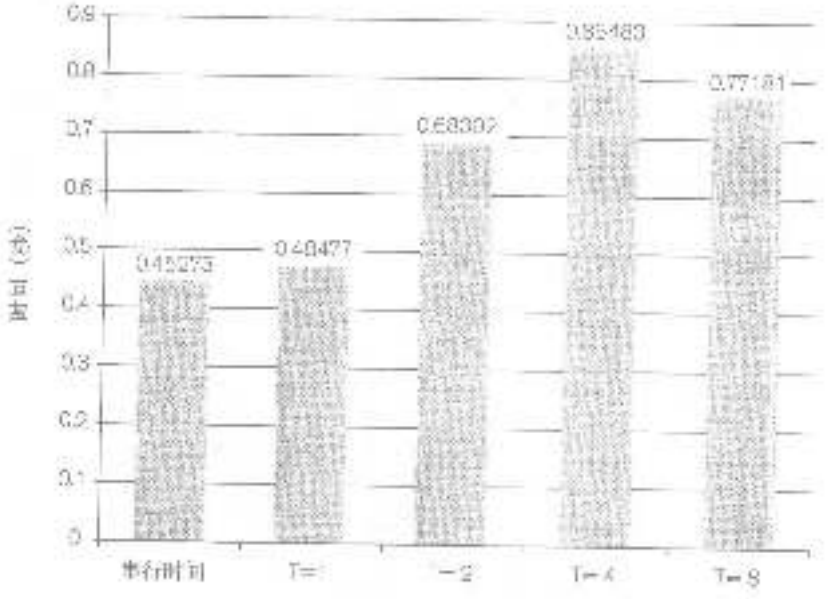


图1-12　第3个统计3的个数的求解方法的性能

从图1-12中可以看到，使用一个线程的执行时间已与顺序代码非常接近，因此我们采用的改变已经消除了大部分锁开销，但是，当使用两个线程时，仍然有显著的性能下降。这一次所出现的性能问题很难用检查源程序的简单方法发现问题。我们需要对低层的硬件进行详细的分析。特别是，我们的硬件使用一种协议以保持一致性的cache，即保证两个处理器所观察到的是相同的存储器映象。如果处理器0对一个指定的存储单元内容进行修改，则硬件将使任何驻留在处理器1的L1 cache中该存储单元的缓存拷贝无效，以防止处理器1意外地访问该数据的旧值。如果两个处理器依次重复修改相同的数据，这种cache一致性协议将会相当昂贵，

因为此时数据会在两个cache中跳来跳去。

第4个求解：尝试4 在我们的代码中，似乎不存在任何共享的修改数据，可是要知道，cache一致性的单位是cache行。在我们的机器中cache行的大小是64字节。因此，虽然在处理器P0和P1上的线程对private_count[0]或private_count[1]进行互斥访问，但硬件将它们放在同一个64字节的cache行中。因为cache一致性的维护粒度是cache行，对cache行中的任何部分修改等同于对整个行的修改，由于共享的cache行在两个cache间的跳跃，使得private_count[0]和private_count[1]被重复地更新（参见图1-13）。逻辑上不同的数据共享一个物理cache行的现象称为伪共享（false sharing）。为了消除假共享，我们可以填充（padding）私有计数器数组使它们处于不同的cache行（参见图1-14）。

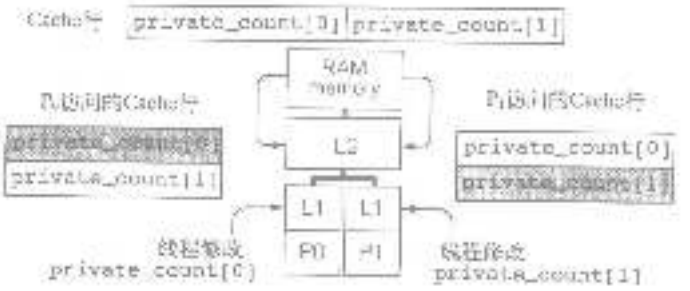


图1-13 假共享。当一个线程（如P0）修改它的private_count（private_count[0]）时，一个cache行（private_count所在行）先从RAM移动到L3 cache，再到L2 cache，最后到L1 cache。当另一线程（如P1）访问它自己的private_count（private_count[1]）时，就会使P0中的L1变为无效，迫使执行将更新写回L2 cache，然后执行再从L2 cache被取到P1的L1 cache中。这样该cache行就在L1 cache和L2 cache间跳跃。虽然P0和P1访问的是不同的存储单元，但它们使用同一cache行

```
1  struct padded_int
2  {
3    int value;
4    char padding[60];
5  } private_count[MaxThreads];
6
7  void count3s_thread(int id)
8  {
9    /* 从线程池完成的数组计算部分*/
10   int length_per_thread=length/t;
11   int start=id*length_per_thread;
12
13   for(i=start; i<start+length_per_thread; i++)
14   {
15     if(array[i] == 3)
16     {
17       private_count[id]++;
18     }
19   }
20   mutex_lock(m);
21   count+=private_count[id].value;
22   mutex_unlock(m);
23 }
```

图1-14 第4个统计3的个数的求解方法中的count3s_thread()，私有count元素得到填充，从而强制其分配到不同的cache行

采用填充措施后，第4种求解方法就消除了使用互斥体的开销和竞争，如图1-15所示，我们已基本取得了成功，我们的并行求解在使用一个线程时执行速度已几乎与顺序执行一样快，在使用两个线程时，并行程序的执行速度接近于顺序程序的两倍，而在4个线程时，则几乎为四倍。

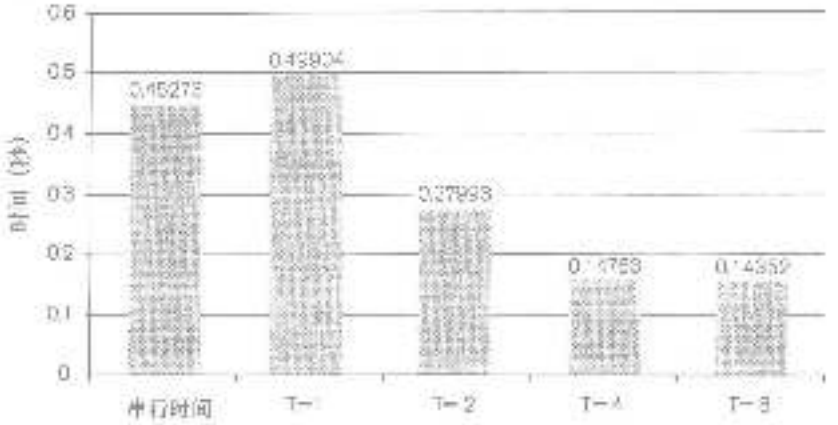


图1-15 统计3问题第4个解的结果表明，单处理器的性能与顺序求解接近，而2个和4个处理器的性能则几乎为顺序的2倍和4倍，但是8个处理器并没有获得额外的性能好处

余下的唯一问题是当使用8个线程时，其性能与4个线程时差不多，这种行为可能有许多理由（与硬件的特性有关），但我们猜测，对于如此大的数组，L2 cache的带宽显得不足。通过在不包含数值3的数组上重复进行实验，我们得到了图1-16，它验证了我们的猜测。我们看到即使在这样的情况下，当未对共享变量进行更新时，用8线程代替4线程，性能没有什么改进。虽然统计3计算所完成的工作几乎与存储器容量毫无关系，但图1-16还是指出了在未来的多核芯片中将存在的存储器有限带宽的问题。如果I/O技术不作改变，则随着晶体管密度的增加，将在一个芯片上集成更多的核，而如果不显著地增长芯片周长以布放更多的I/O脚，则每个芯片的带宽必将会减小。

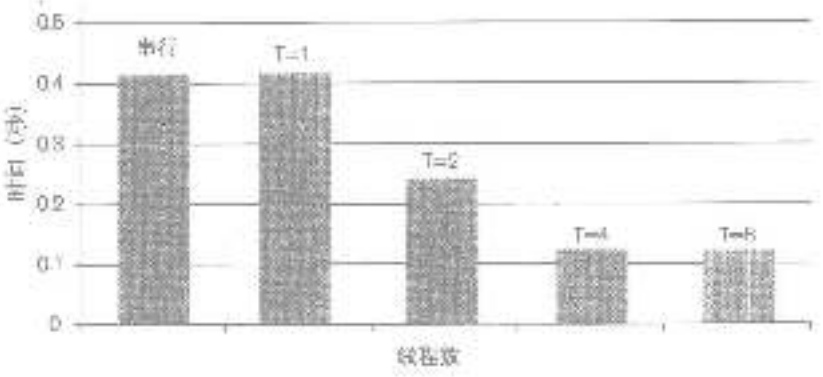


图1-16 在不含数值3的数组上，用第4个求解方法求解统计3问题所获得的性能表明，存储器带宽的限制阻碍了8处理器方案的性能改进

我们的方法学 上述的实验结果是在8个Intel Xeon处理器上获得的，8个处理器是由4个7100系列双核多处理器芯片组成的，主频为2.60 GHz，该系统有三级cache：

L3 cache： 4 MB，合一式，16路组相联

cache行大小为64 B（字节）

L2 cache： 1 MB（每核，总计2 MB），合一式，8路组相联

L1 cache： 16 KB数据和16 KB指令，8路组相联

程序运行的数组数据项大小为50 M（5千万个），随机分布有30%的数值3，实验结果是1000个程序运行所得到的平均值（包括线程派生和销毁），使用微秒定时器。实验使用的操作系统是GNU/Linux 2.6.19-gento-r5，编译器为gcc版本4.1.2 并启用-O2优化。

由上面的例子我们可以看到，创造正确和高效的并行程序比起书写正确和有效的顺序程序要难得多。互斥体的使用说明需要小心控制处理器间的交互。私有计数器的使用说明需要判断并行的粒度，即进程间相互交互的频率。填充措施的使用显示出对机器细节了解的重要性，因为有些时候，小细节隐含着大性能。对这些相互影响的因素进行综合考虑通常使并行性能的优化非常困难。最后，我们还看到两个例子，可以在增加少量存储器容量以获取并行性及性能之间进行折中。

1.4 目标：可扩展性和性能可移植性

统计3程序说明了两个问题：一是并行可以提高性能；二是实现这一点会很复杂。在了解了多核处理器所面临的问题（竞态条件、粒度问题以及假共享）后，我们很容易认为并行程序设计只需考虑正确性和性能这两个问题。事实上，本书的目的更为宽泛。我们的目的是要帮助读者编写好的并行程序，这种并行程序是指具有以下的4个特征：

- 正确
- 好的性能
- 可扩展为使用几千个处理器
- 可移植到各种类型的并行平台上

第一个目的，除了要注意获得并行程序正确性通常比顺序程序要难得多以外，无须多做解释。第二个目的看起来也比较清楚，但我们将在第3章中看到，我们所指的“好性能”的定义存在很多微妙。

对第三个和第四个目的需要做一些详细的解释，因为它们似乎很玄虚且常常被认为不是必要的。例如，对只有几个核的处理器进行编程的人，他对有几千个处理器的超级计算机几乎没有什么兴趣。确实，总是存在这样的一些市场，其终极的目标是性能，因此不惜采用低级的不可移植的求解方案。但对于绝大多数程序员来讲，可扩展性和可移植性是非常重要的，因为并行硬件的前景变化非常迅速。例如，第一个多核芯片只有两个核，但Intel已经在讨论有80个核的芯片。当然，随着芯片中核数的增加，其他的微体系结构特性，如存储器系统，也不得不跟着改变。在快速变化的硬件前景情况下，最好是不要为新出现的硬件而弄得手忙脚乱。要做到这一点，就需要从设计的一开始考虑可扩展性和可移植性，按此原则设计的程序将会有很长的生命周期，这才是对程序设计中投入的巨大智力和经济投资的合理举措。

下面我们将对可扩展性和可移植性进行更为详细的介绍。

1.4.1 可扩展性

为了了解可扩展性问题，可以考虑我们的代码随着处理器数的增加会受到什么影响。统计3程序是参数化的，所以线程数可以变化。这种灵活性允许我们在一个有16个核的芯片上运

行程序而几乎不用修改原来的程序。初看起来，我们好像已得到一个通用解，它可以扩展到几千个线程而只需改变MaxThreads的值[⊖]。但实际上我们并没有做到这一点。确实，扫描数组可以将数组分成独立的段，这样任何进程数都可以并行操作。但是将中间结果组合无法做到这一点，因为所有线程要更新一个全局和。当线程数很大时，我们将再次碰到锁竞争的问题。显然，我们的成对求和方法可以修正这一问题。可扩展性需要可扩展的程序设计的训练。

更一般的，当并行处理器的数目增加时，物理的极限将迫使设计加以改变，从而影响程序完成运算。例如，通信时延（在处理器间传送信息时的延迟）必将随着处理器数的增加而增长，这只是因为光速限制的缘故。在一个单芯片上，存在不同问题，但当芯片上的核数很大时，它们仍会影响通信时延。对于少量的处理器，处理器的邻近能加快某些操作，但当系统的规模增大时，这些操作就不会加快。在可能的情况下，获得这些收益是有意义的，但是程序的成功必须避免依赖于这种收益。高质量的程序应当能充分利用并行计算机中的快速部件，并避免过度使用慢速的部件。

1.4.2 性能可移植性

刚才所讨论的问题——当处理器数目增加时物理极限对并行计算机特征的影响——不只是使某些操作慢下来，实际上问题将会更加严重。

体系结构设计师为了尽力解决物理极限，已经创造了许多并行计算机设计。这些机器相互之间有很大的不同。不像在顺序计算机中的情况，一个新计算机只需要将源代码重新编译就能够很好地执行。在一个并行计算机上能很好地运行的程序，在一个新的并行机上可能不得不重写一个新的程序。

举例来说，并行计算机通常可被分为以下三类之一：共享存储器，以多核处理器为代表；共享地址空间，以各种超级计算机为代表；分离地址存储器（无共享），以机群为代表。这些区别将影响程序中每一次对存储器的访问，因此它对于如何编写程序有很大的影响。试图移植到所有这些平台上的程序必须对这些不同的存储器结构是强壮（鲁棒）的，而保证强壮性的技术将贯穿于本书的叙述中。

按存储器能力的分类只说明了一个方向的变化。并行处理器中还有许多其他的差异。我们可以通过设置一个足够高的抽象层使得这些差异变得不可见，来解决可移植性问题；此后，编译器会映射该高层规范到平台。这种策略将使我们的程序对并行硬件不敏感，但这不是一个好主意。一般来讲，借助编译器确实能完成这种映射，可是它们经常会引入软件层以实现这种抽象；这些附加的软件隐藏了硬件的性能特征，使得程序员很难了解他们所编写的代码的表现。如果我们希望获得高性能，就不能完全与低层的硬件脱钩。因此，我们将在第2章中采用一个不同的策略来解决这一问题。

因此，我们的目的是保证性能的可移植性，通常称为*性能可移植性*（performance portability）。使程序能运行在不同的并行平台上是远远不够的，程序还必须能很好地运行在所有这些平台上。

1.4.3 原理第一

本书不会为编写好的并行程序提供循序渐进的教程，相反，本书着重并行计算的基本原

⊖ 可参见图1-11和图1-14。——译者注

理，解释各种现象，并阐明为什么这些现象可以体现成功并行编程的机遇或阻碍。采用这一方法的理由有两方面。首先，通过侧重原理，我们希望能提供持久的知识，这些知识的生命周期将超过最新硬件和软件技术的细节，如我们已指出的那样，这些技术的变化异常迅速。其次，更为重要的是，并行程序设计社区还没有所有的答案，因此逐步的求解是不可能的。的确，我们的目的之一是鼓励下一代的研究人员了解当前技术的限制，这样他们就能构建未来更好的解决方案。

在介绍这些原理后，我们将讨论一些在当代并行机程序设计中流行的程序设计语言和工具。同样，我们的目的是更多地关注隐藏在这些方法背后的原理，而不是将读者调教成为特殊语言的专家。为此，我们的论述是很有限的，期盼读者查阅参考手册以获取更完整和更详细的信息。

1.5 小结

本章首先观察在日常生活中熟悉的并行概念（为一个目标同时做两件或多件事情）。尽管熟悉，但并行在过去并不是程序设计的一个重要方面，因为顺序计算机的性能几十年来一直在稳步地增长。这种性能的改进是由于技术进步以及计算机系统结构师在顺序处理器设计中采用并行技术（隐藏并行）两者结合的缘故。由于体系结构的机遇已在很大程度上得到开发，技术上的持续进步使得具有多处理器的计算机成为标准。这一趋势对计算机的程序设计产生了巨大的影响。

我们注意到现有的顺序程序一般不能使用并行计算机。主要原因是现有的程序设计语言和标准的程序设计技术与传统冯·诺依曼计算机体系结构的顺序处理密切相关。并行求解，如由几个简单的计算（求和、并行前缀和统计3的个数）所示，已经阐明了并行计算的特征。虽然，它们可能不是我们最初的求解想法，但仍是相当直观的。在程序员本能地设计计算问题的并行求解之前，需要改变计算思维，我们称之为范型改变。

对并行硬件进行了快速、不全面的观察后，我们发现了计算平台的巨大差异，有具有两个处理器的芯片到具有几千个处理器的服务器中心。尽管计算平台在规模和设计上存在巨大差异，但它们的并行特性仅依赖于一个很小的基本原理集。我们认为侧重这些原理的目的在于使得程序员编写的并行程序具有高性能、可扩展性和性能可移植性。

历史回顾

早在20世纪50年代的第一批商业机器中就已经在顺序计算机的设计中采用了并行技术。一个里程碑式的并行机是Illiac IV，于20世纪70年代由伊利诺伊大学厄巴纳-尚佩恩分校团队建造。虽然Illiac IV用低级的类汇编代码成功地进行了编程，但开发将顺序（Fortran）程序转换成并行形式的编译器的任务则是由David Kuck教授和他的同事们完成的。直到20世纪末整个社团的研究者们继续向这一目标努力，出现了大量有关并行化编译器的文献。

习题

1. 解释以下有关线程程序设计术语的含义：
   a. 线程
   b. 竞态条件

c. 互斥体

d. 锁竞争

e. 粒度

f. 假共享

2. 描述如何将成对求和计算改为寻找数组中的最大元素。

3. 修改成对求和程序以log n时间求解统计3计算，假设$P = n / 2$。

4. 修改成对求和程序求解统计3计算，假设$n = 1024$，$P = 8$。

5. 用forall重写迭代求和程序；不要忘了竞态条件。

6. 找一个离您最近的并行计算机——也许膝上计算机或实验室里的计算机——并设法查明它有多少个处理器，每个处理器能访问多大的存储容量，以及可以使用哪些语言和软件进行编程。为该计算机编写一个“hello world”程序。

7. 如前所述，树求和算法总是用$n = 2^m$加以说明，以使树完全平衡。对此算法加以修改以适用于n不是2的幂的情况。

8. 如前所述，树求和算法需要$P = n/2$个处理器，此时能实现完全并行。修改树求和算法以适用于处理器数小于$n / 2$的情况。

9. 任意写一串16个整数的序列。用并行前缀算法创建一个“最大前缀”的新序列；构建该树结构（图1-4）并建立向上和向下的值流动。如果该算法使用带符号数，则流入根的是什么值？

10. 用C语言编写并行前缀算法中在结点上执行的向上流和向下流的代码段。

第2章 认识并行计算机

要编写高效的并行程序，很重要的一点是要认识并行硬件。关于这一主题有大量的资料，因为多年来已经开发了许多并行体系结构。我们不打算全面地讨论这一主题。相反，我们的目的是描述当前一组有代表性的机器，使程序员有一个坚实的并行程序设计基础。

2.1 用可移植性衡量机器特征

如第1章中所讨论的，并行机存在相当大的差异，从具有几个处理器的多核芯片到拥有几千个处理器的机群计算机。为了编写优质的并行程序，对于硬件我们到底应该知道多少？一个极端的答案是，我们应该知晓机器深层次的知识细节，因为有时这样能获得显著的性能改进。可是，由于硬件通常只有较短的生命周期，因此很重要的一点是，我们编写的程序不应与任何具体的机器关联太紧密，否则当新一代的机器出现时将不得不重写我们的程序。因此，可移植性的目标促使我们忽略某些机器细节。

对于“我们应该知道多少”这个问题的回答很可能依赖于用户。在某些情况下，程序员愿意付出巨大的努力去改进性能，例如，游戏开发者、嵌入式系统程序员和硬件供应商。而其他的许多人则期望所编写的代码有更长的生命期，从而促使另一种观点的产生，即应该更侧重可移植性。

因此，本章的目的就是首先阐述存在于并行计算机中的差异性，然后考虑抽象机器细节以支持性能可移植性的方法。我们将从观察6种特殊的并行计算机开始，阐明这些平台如何不同。然后，通过考虑两个并行计算机的抽象模型，PRAM和CTA，从这些机器的细节完成抽象。最后，我们将讨论提供给程序员的3种主要通信机制：共享存储器，单边通信以及消息传递。

2.2 6种并行机介绍

我们现在将更仔细地观察6种并行计算机，从而反映当前并行硬件的差异性。通常，主要强调那些与程序员有关的主题的细节，而且因为我们将对其综合，因此没有必要细致地研究这些细节。

2.2.1 芯片多处理器

基于硅工艺的持续改进，即众所周知的摩尔定律，现在已经允许计算机制造商在一个硅芯片上构造多指令的执行引擎，俗称核（core）。IBM是第一个多核设计的制造商，在2002年宣布了它的PowerPC 970。AMD于2005年5月推出了一个Dual Core Opteron芯片，而Intel则在2006年1月推出了它的Core Duo Pentium。我们先来察看Core Duo，虽然所有这些芯片在本章中都将亮相。

Intel Core Duo 以下是Intel Core Duo设计的特性：

- 一个芯片上有两个32位奔腾处理器

- 每个处理器有自己的32 K L1数据cache和指令cache
- 共享2 MB或4MB的 L2 cache
- 共享存储器控制器，I/O控制器及其他
- 借助共享存储器两个处理器间能进行快速的片内通信

Intel Core Duo设计基于Pentium M的体系结构；它执行单线程代码的速度与单处理器的Pentium M一样，所以对未并行化的程序来讲，性能并没有损失。图2-1显示了Core Duo的逻辑结构。两个处理器有32 KB私有1级指令cache（L1-I）和数据cache（L1-D）。这些cache由共享2MB（或4MB）2级cache（L2）支撑，L2混合存储指令和数据。总线控制器仲裁L2 cache和RAM之间通过前侧总线（Front Side Bus，FSB）的传输。

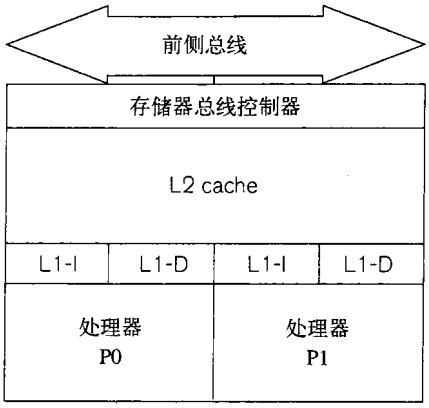


图2-1　Intel Core Duo的逻辑结构。总线控制器作为与RAM相连接的前侧总线的接口

以程序员的观点，Core Duo体系结构的关键特性是，两个处理器所见到的是一个一致的共享存储器映像。当一个处理器访问RAM中的一个单元时，含有该单元的cache行就被传送到L2 cache，并从那里再传送到请求处理器的L1 cache中。如果另一个处理器也访问该单元，由于该单元已在L2 cache中，因此允许将该行立即送往另一个处理器的L1 cache。此后，两个处理器就能快速访问该单元的本地拷贝。

当一个处理器企图改变存储单元中的值时会带来复杂性。如果一个处理器改变它的私有L1拷贝，而另一个处理器继续使用它自己的私有L1拷贝的旧值，那么该值将变得不一致，而计算就会出错；也就是说该值是陈旧（stale）的。使用cache一致性协议就可以避免这种情况。一致性协议限制了两个L1 cache和L2 cache之间的交互，它保证仅在处理器单独使用该cache行的情况下，处理器才可以写入私有的高速缓存行单元；此后，当另一个处理器使用该行时，就需要将已更新的值重新装载到它的L1 cache中。Core Duo的这种特殊协议称为MESI协议，MESI是修改（Modified）、独占（Exclusive）、共享（Shared）和无效（Invalid）的缩写，它们是cache行的四种可能状态。虽然MESI协议较复杂，但很有效。有了这一协议，不同线程就能通过共享存储器方便地进行合作。

虽然cache一致性协议可以防止一个处理器废弃另一个处理器的工作，但在一个芯片上连接两个核还存在有其他复杂性。首先，协议引入了开销。即为了对一个共享的cache行进行修改，修改的处理器必须先获得对该行的独占使用权。对于在一个芯片上共享的简单情况，Core Duo进行了优化，从而可以加快操作速度；不过，代价是昂贵的，因为它潜在地包含了处理器间的交互（参见对称多处理机体系结构）。其次，当两个处理器处理同一问题时，它们对存储器带宽的要求可能是单个处理器需求的两倍；主要的节省来自共享指令。Intel在Core Duo中采用加倍平均带宽的方法来解决这一问题。

对Intel Core Duo与AMD的Dual Core Opteron进行比较是非常有意义的，后者是同代不同的2处理器设计。

AMD Dual Core Opteron　以下是AMD Dual Core Opteron设计的特性：

- 芯片上有两个AMD64处理器

- 每个处理器有一个64 K L1数据cache和指令cache
- 每个处理器有一个独立的1 MB L2 cache
- 提供共享存储器访问的直接连接体系结构（Direct Connect Architectyre）
- 借助系统请求接口（System Request Interface）；两个处理器间能进行快速的片内通信

AMD所采用的2处理器体系结构与Intel的稍有不同。图2-2显示了Dual Core的逻辑结构。处理器执行来自私有、专用64 KB的一级指令cache（L1-I）和数据cache（L1-D）的指令和数据。每个处理器有一个组合的私有1MB L2 cache。系统请求接口（SRI）承担存储器一致的任务，保证两个处理器所见到的是单一的存储器映像。存储器的访问基本上如前面所述，但AMD使用MOESI cache一致性协议，它是MESI协议的扩展，增加了一个“拥有”（Owned）状态，即使RAM拥有的是一个陈旧的拷贝，该状态仍允许处理器间共享cache值。对RAM（或是对其他处理器）的请求使用行业标准HyperTransport技术实现。

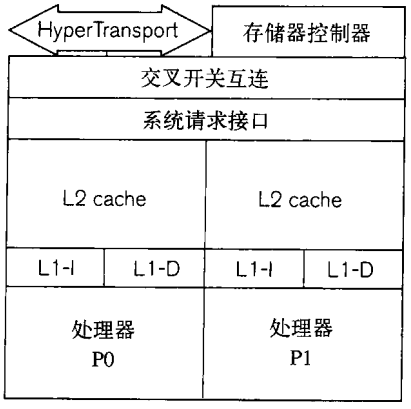


图2-2 AMD Dual Core的逻辑结构。处理器有一个私有L2 cache；由系统请求接口提供存储器的一致性；HyperTransport技术连接到RAM或其他Opteron芯片

芯片多处理器的比较 Dual Core和Core Duo之间的主要差别是L2 cache的位置：在AMD Dual Core中它是处理器私有的，而在Intel Core Duo中它是共享的。这一差别虽然微小，却很重要。在L2“背上”，由SRI管理cache一致性，使得AMD设计不但可使处理器拥有更大的私有存储器，而且也允许一致性的信息能很容易地与其他处理器结合，从而形成一个称为对称多处理器（SMP）的全局体系结构。与之相反，通过在L2“前面”管理cache一致性，Intel的设计在需要时允许一个处理器更多地利用它的L2 cache共享，并且它支持低的片内通信时延。使用单个2-处理器芯片的系统更愿意采用Intel的设计。而组合几个2-处理器芯片的系统则青睐AMD的设计。从程序员的观点看，这两个设计在很大程度上没有什么差别。两种设计实现的都是一个一致性的共享存储器。

2.2.2 对称多处理器体系结构

对称多处理器（SMP，Symmetric multiprocessor）是一种所有处理器访问单一逻辑存储器的并行计算机；有时存储器的一部分在物理上邻近每一个处理器，例如，在同一块板上。为了获得一致性的存储器映像，所有处理器被连接到一个公共点，通常是存储器总线，而每个处理器能在总线上监听（snoop）存储器上的访问活动（见图2-3）。为了说明监听的含义（存在有许多监听协议），设想有几个处理器需要访问存储器块（或cache行）x。如果处理器P0从存储器请求块x，而处理器P1已经缓存有该块，则通过监听总线，P1知道了P0的请求，因此它就将自己的该块副本标记为“共享”，以表明它也有x的一个副本。如果处理器P2对x发出一个“写请求”，则P0和P1均会监听到这一请求，并使它们所拥有的副本无效，以保证P2对该块的唯一拥有，并保证它在对x更新时，没有其他的处理器能使用x；当P2最终完成更新时，存储器也将被更新，此后对块x的请求将获得一个新值。

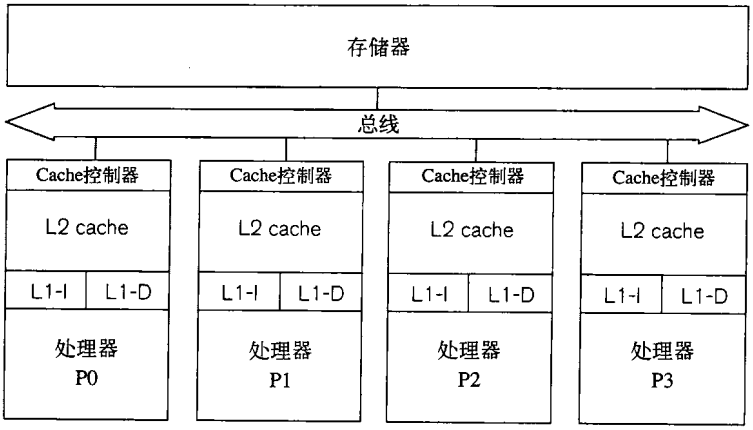


图2-3 对称多处理器（SMP）示意图。每个处理器的cache控制器通过公共的存储器总线建立存储器访问请求。所有的cache控制器监听存储器总线，留意其他处理器所访问的那些地址，并调整它们所缓存的值的标志，以保证一致性cache的正确使用

公共连接点总线是一个潜在的瓶颈，因为每次只能进行一次存储器操作。总线的串行使用限制了以这种方式可连接的处理器数，这就意味着SMP的规模必定是较小的，通常少于20个连接。注意到瓶颈是用单位时间内存储器请求数来衡量的，因此较大容量的L2高速缓存有助于减少总线上的阻塞。另一个含义是用多核取代处理器无助于构造一个更大的机器，因为附加的核将增加对总线的存储器请求，从而使瓶颈更为严重。

SMP以下列两种方式获取高性能：借助小规模和聚簇于总线附近，从而可加快运行速度；借助于使用复杂的高速缓存协议各处理器就能高效地使用总线上的共享资源，减少多个通信操作竞争总线和导致延迟的可能性。AMD Dual Core的系统结构很适合构成SMP。下面要讨论的Sun公司的Sun Fire E25是另一个SMP的例子。

硬件多线程 由于处理器的速度远远高于存储器的速度，硬件多线程已日趋普及。多线程的概念类似于操作系统中的上下文切换：在等待长时延的事件结束前，处理器切换到其他进程。某些多线程处理器在每次cache不命中时切换，而另一些，如Cray MTA1（MTA表示Multithreaded Architecture，多线程体系结构），则每个周期切换一次。硬件多线程需要附加的资源，因此多个处理器间共享硬件资源（带宽、cache、TLB等）的同时多线程（SMT）已变得相当普遍。然而，趋向更简单的多核芯片可能逆转SMT这种发展趋向。

Sun Fire E25K 以下是Sun Fire E25的特性：

- 最多可以有72个处理器，每个处理器能执行两个硬件线程
- 150MHz的Sun Fireplane由地址、响应和数据3个交叉开关互连网组成，此外有18条监听总线
- 所有处理器有相同的共享存储器访问时延
- 共享存储器容量为1.15 TB

Sun Fire E25K采用了一个很有创意的共享存储器设计，如图2-4所示。18块E25K板中的每一块含有4个Ultra SPARC IV Cu处理器；每个处理器能直接访问多至16 GB的存储器，因而整个系统共有1.15TB共享存储器。18块板由地址、响应和数据3个18×18交叉开关互连，由它们处理处理器间的cache行传送。此外还有18条监听总线（图2-4中的虚线）实现存储器的一致

性。在18块板之间，Sun Fire E25体系结构使用一个基于目录（directory based）的cache一致性协议。在基于目录的协议中，对存储器的请求被送往一个集中的目录表，在该表中保存有所有被缓存的存储器块。与监听协议不同，基于目录协议更具可扩展性，但时延更长。因为每个存储器的操作需包括以下3个步骤：对目录的最初请求；操作转发到相应cache；对请求处理器的响应。

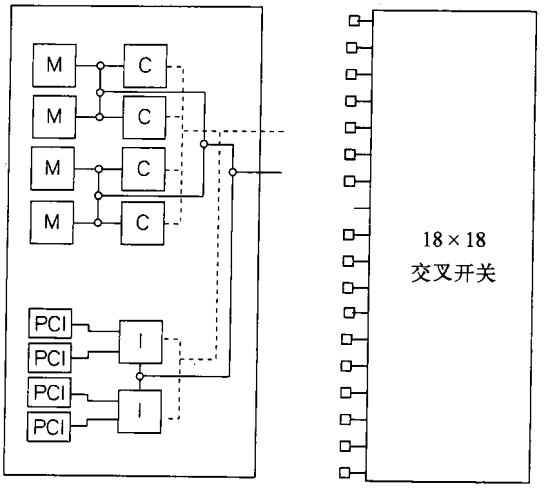


图2-4 Sun Fire E25K。18块板由地址、响应和数据3个交叉开关互连；每块板有4个Ultra SPARC IV Cu处理器；监听总线用虚线表示

在E25K的设计中，交叉开关提供了巨大的总通信能力。图2-5显示了互连4块板的交叉开关，指明在每一对板之间存在一个直接的连接。对n个结点来讲，交叉开关的增长复杂性为n^2，所以交叉开关仅当n值较小时才是实用的，E25K的18×18已接近极限。如所指出的，监听是一个潜在的阻塞源，但被监听的总线仅是对单板的限制。监听逻辑通过交叉开关与系统的其余部分进行通信。由于用一个分离的数据传输交叉开关专门处理cache行的移动，故一致性的处理可以并行进行。

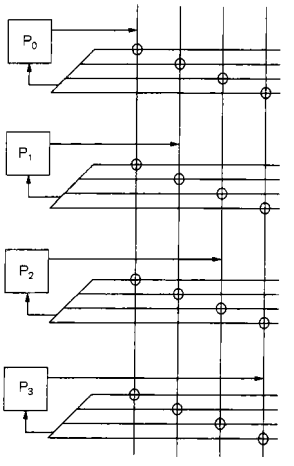


图2-5 连接4个结点的交叉开关。请注意输出和输入通道；除非有图中所示的连接，交叉线不会连接。通过设定相应的一个开圆，一对结点就能直接相连

2.2.3 异构芯片设计

上面的扩展系统规模的方法是将一个标准处理器重复多次，另一种不同的扩展方法是以一个或多个专用计算引擎来扩展一个标准的处理器，这些专用计算引擎称为附属处理器(attached processor)。其思想是由标准处理器完成通用、难以并行化的计算部分，很可能标准处理器已执行得足够快，而由附属处理器完成密集计算部分。以下是这类设计的几种比较熟知的变化：

- 图形处理部件（GPU）
- 现场可编程门阵列（FPGA）
- 为视频游戏设计的细胞（Cell）处理器

我们现在只讨论Cell的体系结构，在第10章中我们将再回到异构体系结构。

Cell 定位于视频游戏市场的Cell处理器是由Sony（索尼）、IBM和Toshiba（东芝）联合开发的。Cell有一个64位PowerPC核和8个支持32位向量操作、称为协同处理单元（SPE，Synergistic processing element）的专用核。Cell处理器最重要的特性是处理器间有很高的带宽，这是单元互连总线（EIB）和连接到片外RAM的12.8GB/s存储器双总线的贡献（参见图2-6）。Cell处理器的主要特性如下：

- 双线程64位PowerPC处理器
- 执行向量指令的8个32位协同处理单元（SPE）
- 每个SPE有256 KB的片内RAM
- 连接SPE的高速单元互连总线（EIB）

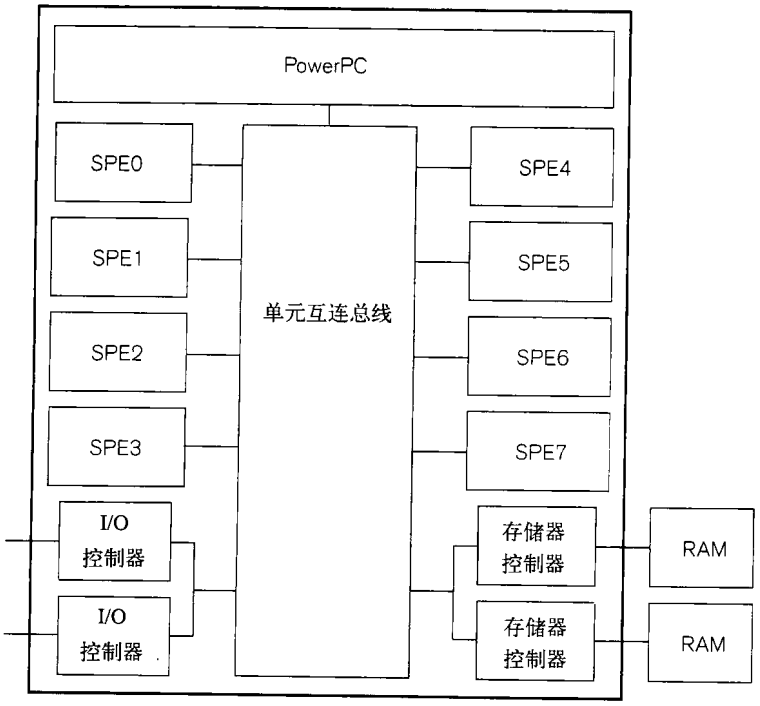


图2-6　Cell处理器的体系结构。该体系结构设计用于移动数据：高速I/O控制器的带宽为76.8 GB/s；连接到RAM的双通道中每一个的带宽为12.8 GB/s；EIB理论上总带宽为204.8 GB/s

不同于我们已讨论过的其他芯片多处理器，Cell不为协同处理单元提供一致性存储器，因此Cell的设计者选择的设计原则是性能和硬件简单性比编程方便更为重要。

为了保持性能比可编程性更重要的设计哲学，SPE有一个128位宽的数据通路，以支持向量指令（vector instruction），向量指令中的一个操作将在几个值上并行完成：16个值上的8位整数操作，8个值上的16位整数操作，4个值上的32位整数操作或（单精度）浮点操作。这将增加程序员的编程难度，程序员必须小心地管理进出各SPE的数据移动，以使向量部件能保持繁忙。如果成功，则Cell处理器将生成令人印象深刻的吞吐量。

向量处理器 第一台成功的并行计算机称为向量处理器（vector processor）。Cray 1是一个著名的例子。向量处理器是对快速顺序（RISC）处理器的扩展，它的向量寄存器容量为64字。向量指令能够从存储器取/存向量（64字块），对两个向量寄存器中的相应元素完成基本的整数和浮点算术运算以及完成归约（即组合一个寄存器中的元素）等操作。虽然程序员可能认为向量操作是并行执行的，实际上向量操作是深度流水化的，从而可获得很高的性能。

2.2.4 机群

机群是由商品部件构成的并行计算机。组成机群的结点通常是一些板，板上包含一个或几个处理器，一个RAM存储器，此外通常还有磁盘存储器。各结点由商品互连网连接，可用的互连形式包括千兆位以太网、Myrinet、Quadrics、Infiniband以及光纤通道。与大多数其他的高端计算形式相比，由于机群使用商品部件，因此机群有很大的性价比优势。机群的一个主要特性是存储器不为各机器所共享；处理器只访问自己板上存储器，当要与其他处理器通信时，需要借助消息传递机制。

由于机群的商品特性，在World Wide Web网上有大量的操作指南，指导用户用商品部件来装配自己的机群。下面是一个典型的装配机群系统的配置方案：

- 8个结点，每个有8路Power4处理器，32 GB RAM，两个磁盘
- 一个控制处理器
- Myrinet 16口交换器，4块板上的8个PCI适配器
- 开源软件

操作指南既说明了如何安装硬件，还会指导用户如何安装软件。

当然，也可由制造商那里购得预先装配好的称为*刀片服务器*（blade server）的机群。一个刀片是一块含有一个或几个处理器芯片、RAM、磁盘、若干个通信口以及几个冷却风扇。刀片服务器性能的变化范围很大，但它们都可作为并行计算机进行编程。例如，HP公司的Cluster Platform 6000刀片：

- 任意（合理）数目的刀片，每个有两个双核的Itanium 2，1.6GHz的处理器，3MB cache
- 每个刀片有16 GB RAM
- 每个刀片有两个磁盘，光纤通道互连
- Myricon公司Myrinet 2000互连网（3.2～3.6 μs NIC时延）[⊖]

2.2.5 超级计算机

传统上超级计算机由国家实验室和大公司所使用，具有许多不同的体系结构，包括机群，

⊖ NIC是指网络接口芯片。——译者注

因此，很难一般性地介绍其设计。一个值得关注的现代设计是BlueGene/L（蓝色基因）机器。

BlueGene/L　IBM继续构造大机器以竞争超级计算机市场，最近的一个产品是BlueGene体系结构。BlueGene/L的一个有趣特征是它的处理器以770MHz中等速度运行，这比以2～3GHz速度运行的典型PC处理器慢了许多。尽管如此，由于它使用了大量的处理器，使得BlueGene/L体系结构的一个实例直接跃居世界最快超级计算机Top500排行榜的首位。

BlueGene/L设备的一种配置如下：

- 65536个双核结点，每个结点是一个440 PowerPC处理器（参见图2-7）
- 每个结点有32 KB的L1 指令cache和数据cache
- 每个结点有一个Double Hummer优化浮点部件，该部件由两个对偶的标准浮点部件组成，每个周期能完成4个操作，即2.8GFLOPS
- 每个结点有一个4 MB 顺序一致性共享片内L3 cache和512 MB的片外共享RAM
- 每个结点有6个双向口连接到3维环绕互连网
- 每个结点有3个双向口连接到一个集合网
- 每个结点有4个口连接到一个障栅/中断网

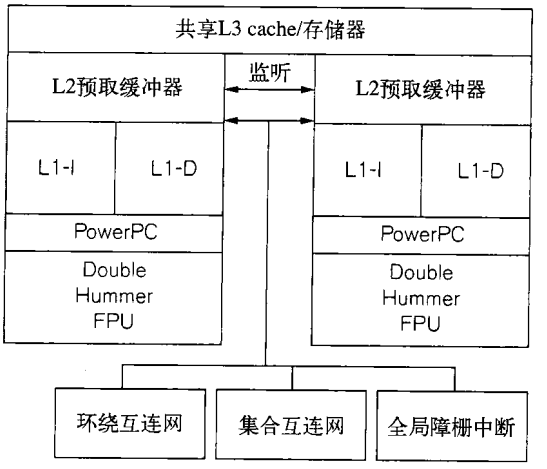


图2-7　BlueGene/L结点的逻辑组成

处理器虽然以中等时钟速率运行，但它具有所有通常高性能处理器的特征，包括指令双发动、乱序执行、大多数指令（包括乘法）在单周期内完成。因此标称的处理器速率为1.4G操作/秒。

结点排列成三维环绕网（torus network），它是带有环绕边的网格（参见图2-8a）。每个结点与它最近的6个相邻结点相连，而整个BlueGene/L扩展成64×32×32的3维阵列。环绕网中不直接相连的结点之间的通信需通过网络中的路由完成。例如，当图2-8中的右前上方结点需要驻留在左后下方结点中的数据时，就需要路由该数据。也许，先是沿水平方向，再沿垂直方向，最后由后向前。由于路由器使用直通（cut through）路由技术，数据不需要在每一个结点处停留。最坏的组合网络时延（在每一维上传输一半路程）需经过32+16+16跳，需时6.4μs。

集合网络（collective network）是第2个连接结点的独立网络。该网络芯片已被扩展成具有算术运算能力，所以通过网络的数据流可以被组合形成全局和（在第1章中已经说明）。在并行计算中，这种全局操作是很普遍的。BlueGene/L支持SUM（总和）、MIN（最小）、MAX（最大），以及按位OR（或），AND（与）和XOR（异或）的结点操作。同样重要的是，集合

网络还支持广播，简化了一对多的通信。采用这种硬件的优点很显著：一个全局的浮点加，只需要两次通过网络，一次是寻找最大指数，而另一次是对尾数求和，总共约需10μs，这只比访问一个任意浮点值的最坏通信时间稍大一些。

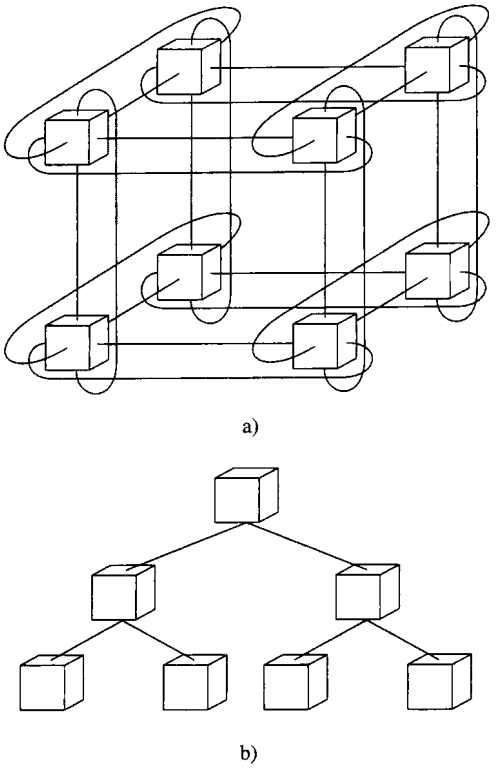


图2-8 BlueGene/L通信网络：a) 3维环绕网，用于标准的处理器间的数据传输；b) 集合网络，用于归约的快速求值

最后，障栅网络（barrier network）提供了第3种类型的全局通信。从概念上讲，障栅网络是连接所有结点的公共线，它能用作中断、障栅同步以及其他简单的全局通信；当然，其实现完全不同。该网络的来回时延是1.5μs。

当程序员编写他们的代码时，他们要访问两个核所共享的512 MB RAM。由于没有其他的共享存储器，一个大型的并行计算问题状态将在处理器的RAM间被分割。处理器通过环绕网来回发送消息进行通信。为了简化程序设计，存储器的结构允许一个处理器专门处理通信，而另一个处理器则专门用于计算。当然，广播是由集合网络来完成的，而同步则使用障栅网络来完成。

Top500超级计算机排行榜 Top500排行榜（www.top500.org）是由曼哈姆大学、田纳西-诺克斯维尔大学以及NSERC/LBNL共同维护的。计算机按测试LINPACK基准测试程序所得到的性能进行排名，LINPACK是一组并行线性代数例程。纵然这种测试是偏向于科学计算，但已有一些明显的趋向表明高端并行计算的多样化。每年，有更多的商业部门参与排名，这些机器由运行更大范围应用的更多公司所使用。例如，在描述这些计算机所运行的应用的表中，在2007年11月报表中报告最多的领域是：金融（72项）、地球物理学（43项）、研究（38项）、服务中心（28项）和半导体（25项）。另一个值得注意的趋势是，机群日益普及，现在在表中占了406项。而在1993年的第一次排行榜中，用户定制的超级计算机则独居鳌头。

如我们在本章稍后将要讨论的，计算机体系结构一个重要性能因数是通信时延λ，它是指一个处理器与任何指定的存储器字节之间的通信时间。用这段时间内可执行的操作数加以衡量。对于BlueGene/L，即是每秒1.4 G次操作乘以6.4μs网络时延，因而其λ = 8960（不包括软件开销）。

2.2.6 对6种并行计算机的评论

这6种机器有多少相似之处？有多少不同之处？一个差别是存储器模型。Core Duo、Dual Core以及Sun Fire E25K实现的是一个*共享地址空间*，是所有处理器可访问的一致性存储器；HP机群和BlueGene/L实现的是一个*分布式地址空间*，每个处理器只能访问整个存储器的一部分。（Cell代表一个附属处理器模型，其中，主处理器可对所有存储器进行全局访问，并为各个SPE建立数据流，在第10章中将对它作进一步的论述。）在分布式地址中，不共享存储器的各个处理器通过消息传递进行相互之间的通信。实现分布式地址空间的并行计算机通常被称为是*分布式存储器机器*。

共享存储器机器对程序而言似乎更为方便和自然，在随之而来的问题是，“为什么要构造这些不同类型的机器？为什么不构建有较低存储器时延和大量处理器的共享存储器机器？”大量机器不采用共享存储器有其深层次的基本原因。其中一个原因是，随着机器规模的增大，光速延迟也将增加。但真正的挑战，与其说是传送信息的时间，不如说是既保持存储器一致性又不显著增加存储器的有效访问时间。有关的技术挑战已超出本书的范畴，但至少通过观察Sun Fire E25K和那些多核芯片的体系结构的巨大差异可以感觉到：Sun公司的工程师不得不组合许多复杂的思想，并采用激进的工程技术，将处理器数提升到72个。每一次的增加都带来更大的挑战。所以，创建一个可扩展的共享存储器体系结构的前景是极其渺茫的，因此，并行体系结构的多样性在可预见的未来将是一个不争的事实。

弗林分类法　1966年Michael J.Flynn（弗林）提出，计算机体系结构可以根据它们所使用的指令流数和数据流数进行分类。

	单指令	多指令
单数据	SISD	MISD
多数据	SIMD	MIMD

SISD对应于我们通常的顺序计算机，MISD是指为了增加可靠性的多个冗余计算。SIMD表示一条指令作用到多个数据值上，如Cell的SPE；而MIMD描述不同指令作用到多个值上，当今的大多数并行计算机都是其典型代表。经常使用的只有SIMD和MIMD这两项。

因为我们的目标是性能可移植性，我们要问“如何编写程序使它能在如此不同的机器上很好地运行？”答案是抽象掉不重要的细节。这是一个屡试不爽和熟悉的方法。

2.3 顺序计算机的抽象

既然并行机器存在差异性而且不希望为每种类型机器编写不同的程序，其关键的一点就是需要有一个能指导程序开发的精确并行计算机模型。与之对照，可以注意到顺序计算由于有这样的一个模型因而长期受益：该模型便是随机访问机（RAM）模型，也称为冯·诺依曼模型，因为作为计算机先驱者他第一个描述了这个模型。（RAM是随机访问存储器和随机访

问机两者的缩写；在本书中我们广泛地对它们进行了论述，为避免混淆，我们始终将后者作为RAM模型。）

RAM模型将一台顺序计算机抽象成一个有指令执行部件和一个有无限容量的存储器的装置（按照定义，计算机还需要其他部件，如输入和输出，RAM模型将忽略这些部件）。存储器存储程序指令和数据，且任何存储单元能在“单位”时间内被访问（读或写），而不论该单元位于何处。单位时间访问是随机访问的特征，这也是取名RAM模型的原因。在RAM模型中，指令执行部件每个周期取出和执行一条指令，除非遇到一条转移指令，它将在下一个周期按序取下一条指令。在一对数据值上操作的指令都是很简单的。这一行为模拟了我们所熟悉的顺序计算机[⊖]。事实上，该模型对我们是如此的熟悉，因此我们很少再去考虑，而且，我们发现很难再去考虑任何其他计算方法。

在用类似想法模拟并行的情况之前，让我们首先回顾顺序程序设计如何从RAM模型获益的。

2.3.1 应用RAM模型

RAM模型的简洁性非常重要，因为它允许程序员根据模型中的指令数来估计整个性能。例如，如果我们要在已排序数组A中查找一项（searchee），可以使用顺序查找或二分查找（参见图2-9）。给定RAM模型，我们知道顺序查找将平均需要对for循环进行$n/2$次迭代，才能查找到该项，此外我们知道每次迭代通常将需要执行少于12条的机器指令。二分查找是一个稍为复杂的算法，但它的期望性能大约是while循环的$\log_2 n$次迭代，而每次while循环将需要执行少于24条的机器指令。对于小的n值，顺序查找较快；而对于大的n值，则二分查找更有效。

a)
```
1  location=-1;
2  for(j=0; j<n; j++)
3  {
4    if(A[j]==searchee)
5    {
6      location=j;
7      break;
8    }
9  }
```

b)
```
1  location=-1;
2  hi=n-1;
3  lo=0;
4  while(lo!=hi)
5  {
6    mid=lo+floor((hi-lo+1)/2);
7    if(A[mid]==searchee)
8      break;
9    if(A[mid]>searchee)
10     hi=mid;
11   else
12     lo=mid+1;
13 }
```

图2-9 两种查找计算：a) 线性查找，b) 二分查找

2.3.2 评估RAM模型

在实际的硬件中实现RAM模型非常重要：该模型描述一个程序是如何运行的，要使模型有用，则硬件必须如描述的那样运行。如果不能做到这一点，那么就需要一个新的模型。模型和结果之间如果不吻合，则必须重新评估我们的算法及相应的程序。的确，这个惟一长期生存的模型已使得算法设计多年来不断向前发展而无须考虑每种独特计算机的繁杂细节。考虑到硬件35年来的指数性能改善，以及35年来的硬件复杂性的增加，无疑这是巨大的成就。

当然，RAM模型是不现实的。例如，对当今的处理器来讲，取数的单周期代价无疑是一

⊖ 虽然现在的计算机似乎仍符合上述描述，但已在许多方面与理想化的机器有所不同。

个神话，就如同幻想存储器容量是无限的一样，然而RAM模型对于大多数的应用仍有效，抽象的代价抓住了那些顺序计算机最重要的性质。针对机器细节对算法进行定制实现能显著地改进算法性能时，RAM就获得了成功，因为它有效地指导了通用情况下的算法设计。

当然，模型不会适用于所有硬件，特别是向量处理器，由于它能在单周期中取出很长的数据向量，因而它不适用于RAM模型，所以应该记得用RAM模型编写的传统程序在向量机上无法获得成功。因此直到程序员学会开发一个新的程序设计向量模型后，诸如Cray 1的向量处理器才会充分发挥了它的全部潜力。

2.4 PRAM：一种并行计算机模型

要将顺序程序设计的成功经验转换到并行情况，需要一个理想的并行计算机与RAM模型相对应。类似于RAM模型，该理想计算机应尽可能小而且通用。显而易见的RAM模型扩展是PRAM，即并行随机访问机器模型。纵然不是很明显，这样的结论并不一定成立，我们很快就会看到这一点。

PRAM由连接到一个有无限容量共享存储器（包含程序和数据）的不指明数目的指令执行部件（与RAM模型中类似）组成。每个指令执行部件能按它们自己的程序线程进行执行，为方便同步，它们以锁步方式执行指令。所有的执行部件访问全局存储器，且它们所看到的是一个单一的存储器状态的变化序列，称为*单一存储器映像*（single memory image）。也就是说，如果在一条指令改变x[0]的同时另一条指令改变x[1]，执行下一条指令时，这两个单元的值均已更新。与RAM模型相似，PRAM模型中的存储器的访问也是以“单位时间”完成的。这样，当多条指令同时要访问同一存储单元时，PRAM模型就会出现一种复杂情况。对于读操作，通常允许同时访问。但对于写操作，企图用不同的值写同一个存储单元将存有问题，不同版本的PRAM模型指定不同的协议。协议的范围很广，从完全禁止同时写，到允许同时写，但“对同一存储单元的同时写必须全都写相同值”，再到“允许对同一存储单元进行不同的写，但只有其中一个写（任意选择）是成功的。”对这些不同的协议，有许多文献进行了论述。

不同于RAM模型，对于程序员来讲PRAM不是一个好的工作模型。它的失败在于对存储器行为的不实描述，这一点是很奇怪的，因为简化的存储器行为是RAM模型的优点。PRAM的问题在于，对于可扩展的机器，要在单位时间内实现单一存储器映像实际上是不可能的。即是说，要求所有指令执行部件都“见到”一个一致性的存储器映像，并能以指令执行的单位时间速率去访问存储器，仅当执行部件比较小的时候才有可能。前面所讨论的多核和SMP体系结构可以作到这一点。但当执行部件增加时，为了保持存储器映像一致的延迟就会大大增加。因此，PRAM模型的预测在实际中是无法观察到的。

模型预测与实际不符可能会导致严重的后果，它很可能使算法设计者误入歧途，因为在实际硬件上被认可的那些算法将无法完成，此外模型所预测的其他算法也将是很差的。问题出在所有存储器的访问只需使用单位时间，PRAM完全忽略了并行计算机中一个重要的性能因素，即通信开销。由于这一难题，我们采用了一个更为实际的并行计算机模型。

2.5 CTA：一种实际的并行计算机模型

为了克服PRAM的缺点，我们需要一个考虑通信开销的模型。下面我们描述一个模型，

由于历史的原因，该模型被称为候选型体系结构（Candidate Type Architecture），简称CTA。CTA模型能显式地区分两类存储器访问，分别被称为低廉的本地访问和昂贵的非本地访问。

PRAM的问题 在对并行算法的性能极限进行理论分析时，PRAM是一个颇为有用的模型，但是它的存储器访问时间的单位代价却无助于实际的并行程序设计。特别是，该模型本应理想地指导程序员为求解问题选择最好的算法，可实际上会误导他们到一个错误的解。例如，对于寻找一个数组中最大值元素的问题，最好的实际算法是锦标赛算法，它是第1章中所讨论的成对求和算法的变异；当$n = P$处理器时，其性能应正比于$\log n$。寻找最大值元素最好的PRAM算法是具有独创性的称为Valiant算法。该算法的工作步骤如下；第1步将n个值按3个分组，并分配处理器进行所有可能的比较，在一步时间内找出每一组的最大值。由于在一个3元素组{a, b, c}中找出最大值需要a:b, a:c,b:c共3次比较，因此$P/3$组将需要$3*P/3$个处理器，即正好P个处理器。这一步将问题的规模减小为原来的1/3。以后的几步，组将会更大（第2步按7个分组），但组数会更少，使得所有的比较用P个处理器就可完成。整个求解将在$\log \log n$步内完成。由于完成每一步所需的指令数是固定的，因此整个运行时间将正比于$\log \log n$。虽然该算法非常巧妙，但它并不实用，因为在实际的硬件上运行时，它无法达到所预测的运行时间（原因在于PRAM模型假设可以在单位时间内完成存储器的访问）。的确，如果我们乐观地估计一次通信的代价正比于$\log P$（P为并行计算机中的处理器数），那么完成一步所需时间将正比于$\log P$，这就意味着该PRAM算法将需时$\log n$ ($\log \log n$)，当$n = P$时，它的性能将比锦标赛算法更差。由此我们可以看到，PRAM模型并不能指导程序员去获得最好的实际解。

2.5.1 CTA模型

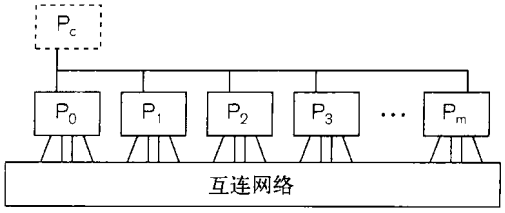
图2-10中显示了CTA并行计算机模型的示意图。它由P个标准的顺序计算机（称为处理器或处理单元）所组成，这些处理器由一个互连网络（又称通信网络）相连接。RAM模型所描述的处理器则由一个执行引擎和一个随机访问存储器组成，在存储器中存放有程序和数据。第P+1个处理器（以虚线表示）是控制器，它的功能是帮助完成如初始化、同步和eureka[⊖]（找到）等操作[⊜]。许多并行计算机并没有这样单独的一个控制器，在这种场合处理器P_0将完成上述的那些操作。

CTA互连网络的拓扑没有说明。图2-11中显示了几种互连网络常用的拓扑。适合于并行计算机的最佳拓扑是需由体系结构师根据各种技术考虑作出的设计决策。程序员对拓扑并不感兴趣，因此在CTA中未显示其细节。

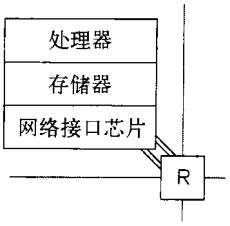
网络接口芯片（Network Interface Chip，NIC）是处理器/网络连接的媒介（参见图2-11b）。图2-10a中的框图显示处理器用4根线与网络相连，这便是结点度（node degree），但是实际的连接数取决于拓扑的特性和网络接口的设计；最少只有一个（双向）连接，而最多则不会超过6个。来往于网络上的数据存放在存储器中，通常借助直接存储器访问（DMA）机制进行读写。

⊖ eureka是一个处理器对其他处理器的中断，使用在如搜索那样的操作中。

⊜ 希腊语eureka原意为“我找到了”，在计算机中eureka用来表示一种控制同步，例如在并行搜索中，当有一个分支找到搜索目标时，就立即发出eureka同步信号，以终止其他分支的继续搜索。——译者注

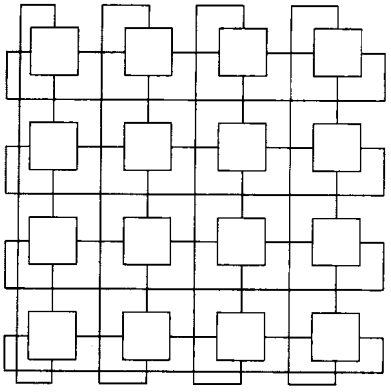


a)

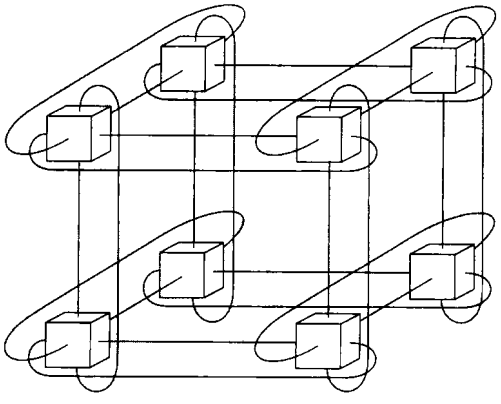


b)

图2-10　CTA并行计算机模型：a) CTA由互连网络连接的P个顺序计算机所组成；以虚线标志的计算机是控制器，完成诸如启动处理等事务性功能。b) 处理单元的细节。更详细的介绍见正文

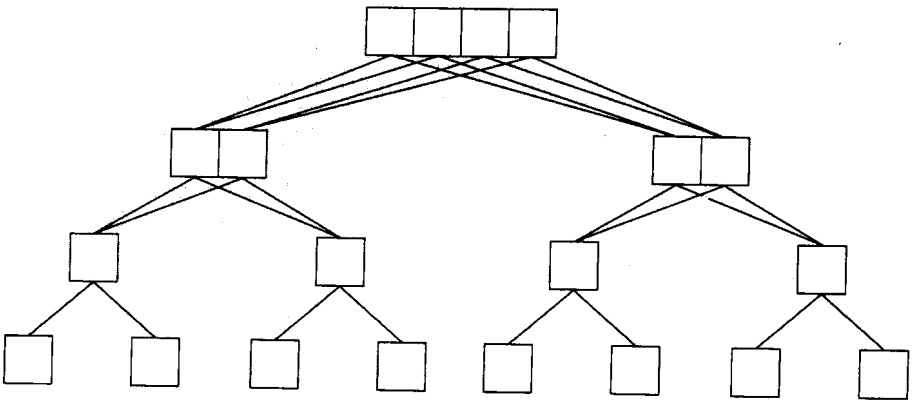


a)

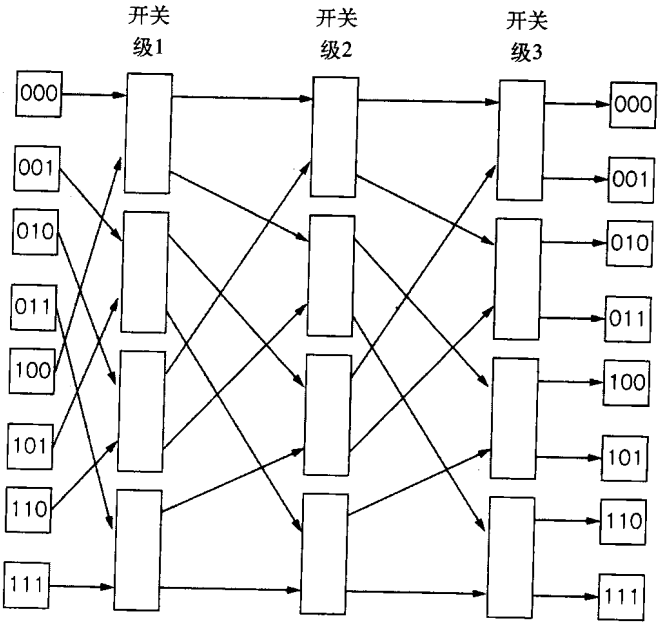


b)

图2-11　互连网络常用的拓扑：a) 二维环绕网，b) 二元3立方体（参见习题8），c) 胖树，d) Ω网



c)



d)

图2-11 （续）

虽然处理器间能进行同步和在遇到障栅时集合停顿，通常它们是独立执行的，运行它们自己的本地程序。如果在每个处理器中的程序是相同的，则这种计算就常称为*单程序多数据*（Single Program Multiple Data，SPMD）计算（请回忆弗林分类），尽管SPMD指的是软件，而弗林的术语指的是硬件。该名称的使用有一定局限，因为即使代码在所有处理器中都相同，但事实上可能执行相同代码的不同部分（它们有一个代码的副本以及自己的程序计数器），因而它们是完全自治的。

处理器可以对自己的*本地存储器*（受cache支持）进行数据访问，其过程与标准顺序计算机的相类似。此外，处理器可以访问非本地存储器，即其他处理器单元的存储器。（CTA模型没有全局存储器）。有三种广泛使用的机制可用于访问非本地存储器：共享存储器、单边通信（我们将其简写为1-sided）以及消息传递。我们将在下面存储器访问机制这一节中对这三种机

制进行描述，这三种通信机制将对程序员和硬件赋予不同的负担，但就CTA机器模型观点而言，它们是可以互换的。

最后，应注意，在本章开始时所讨论的那些计算机都是CTA的实例。BlueGene/L有65536个带有本地存储器的处理器单元（双处理器），一个3维环绕互连网（与网络有6个连接）以及一个硬件控制器（障栅/互连线路）。另一个极端，AMD的Dual Core有2个处理器，一个直接连接（或交叉开关，取决于系统如何逻辑划分从而与CTA挂钩），没有控制器。其他的并行计算机可类似地与CTA挂钩。

2.5.2 通信时延

并行计算机的一个关键点是对本地和非本地的访问需要用不同的时间来完成。对存储器访问所需的延迟称为*存储器时延*（latency）。存储器时延不能用秒来表示，因为模型是对许多不同体系结构的概括，而每种体系结构采用不同的技术和不同的设计单元。所以时延是用相对于处理器本地存储器的时延来加以说明的。这一约定隐含着本地存储器时延大致遵循处理器的速率，而我们乐观地假设本地存储器将以每指令一个字的速率进行访问。当然，本地存储器的访问还受cache行为、处理器及算法设计很多方面的影响，这就使本地存储器的访问充满了变数。好在我们并不需要有关时延的精确值。

在CTA模型中，非本地的存储器时延是用希腊字母λ表示的。非本地存储器访问的代价是很昂贵的，它的λ值比本地存储器访问的λ值要高出2～5个数量级。类似本地存储器访问，非本地访问的代价受许多因素的影响，包括技术、通信协议、拓扑、结点度、网络堵塞以及通信处理器间的距离等。但是由于因数太多，精确地知晓它们是不必要的。

为方便说明起见，表2-1给出了本章前面所介绍过的每种机器类型的估计λ值。

最后，类似于RAM模型，CTA模型忽略外部的I/O。显然，I/O对性能至关重要，但是估计有关I/O的代价比起估计基本体系结构的通信代价更加困难。

表2-1　常用体系结构的λ值估计；速度通常不包括拥塞或其他通信延迟

体系结构系列	计算机	λ
芯片多处理器①	AMD Opteron	100
共享存储器多处理器	Sun Fire E25K	400～600
协处理器	Cell	不详
机群	HP BL6000 w/GbE	4160～5120
超级计算机	BlueGene/L	8960

① CMP的λ值衡量芯片上L1数据cache间的传送。

2.5.3 CTA的性质

以下是有关抽象机特性的总结：

- 有P个处理器，它们是执行本地指令的标准顺序计算机
- 本地存储器访问时间即是通常顺序处理器的存储器访问时间
- 非本地存储器的访问时间$\lambda>>1$，可能比本地存储器的访问时间高出2～5个数量级
- 低结点度意味着一个处理器不能同时进行多个网络的传输（通常为1个或2个）
- 一个全局的控制器以帮助完成如初始化、同步等基本操作

在对存储器访问机制进行更深入的讨论之后，我们将对CTA模型作进一步的观察。

就并行计算机的程序设计而言，可将上述CTA模型的这些特性封装成一个简单的规则：

本地性规则 快速程序趋向于最大化本地存储器的访问次数和最小化非本地存储器的访问次数。

每一个并行程序员在思考算法的设计时，必须牢记这一最重要的指导原则。

应用本地性规则 为了了解如何应用本地性规则，假想一个计算，它有多个线程运行在多个处理器上，算法的每次迭代需要一个新的随机数r。一个明显的解是，使得其中一个处理器存储该种子，并在每个周期生成r，而其他的处理器访问该随机数值。符合本地性原理的一个更好的方法是每个处理器将种子存储在本地，并在每次迭代时冗余地生成r。虽然第2个求解方案需要执行更多的指令，但它们是并行执行的，所以它们不会比一个处理器单独生成r需要更多的执行时间。更重要的是，第2个求解方案避免了非本地的访问，而且由于通常计算一个随机数比完成一次非本地存储器的访问要快，从而就加快了整个计算。

CTA体系结构提及P个处理器，意味着机器是可扩展的。程序员所写的代码独立于精确的处理器数，一般确切的处理器数将在运行时提供。确实，λ会随P的增加而增加，虽然不会增加得那么快；在一个良好工程化的计算机中，处理器的加倍不会加倍λ。

概括地讲，CTA是一个通用并行计算机的模型，它抽象了过去几十年来建造的所有可扩展（MIMD）并行计算机的关键特性。虽然主题有所变化，但CTA所展示的性质应是任何并行计算机所期盼的。

2.6 存储器访问机制

CTA模型没有说明存储器访问机制是共享存储器、单边通信或是消息传递。这三种通信机制都得到普遍使用，下面我们对它们分别加以描述。

2.6.1 共享存储器

共享存储器机制是对顺序计算机扁平存储器的自然扩展。普遍认为共享存储器比其他机制更易使用，但它也被批评难于编程（易产生竞态条件以及难于发现），此外，它可能鼓励低效程序的产生，因为进行非本地的访问过于容易。共享存储器为多线程提供了一个单一的一致存储器映像，但通常它需要某种程度的硬件支持方能很好实现。

在方便地允许任何线程访问任何存储器单元的同时，导致了两个或多个线程企图同时改变同一存储单元所带来的风险。这种竞态条件在第1章中就引起了我们的注意，不过那个例子很容易解决。一般而言，竞态条件是引入难于发现隐错的巨大隐患，从而促使程序员小心翼翼地使用某种同步机制来保护所有共享存储器的访问。在第6章中将会介绍更多的有关信息。

2.6.2 单边通信

单边通信（一边通信），在Cray机上称为shmem（共享存储器），是放宽的共享存储器概念：它支持单一共享地址空间，即所有线程能访问所有的存储单元，但它并不试图保持存储器一致。这种改变简化了硬件，因为它不再需要实现复杂的cache一致性协议，但它将更大的负担加到了程序员身上，因为此时不同的线程可能对同一变量观察到的是不同的值。

对于单边通信来讲，所有地址除了那些被显式地指定为私有的以外，将能为所有处理器

所访问。对本地存储器的访问使用标准的取/存（load/store）机制，但访问非本地存储器时需用get()或put()。get()操作以一个存储单元作为参量，并从非本地处理器的存储器中取值。put()操作则将存储单元和值作为两个参量，将该值存放到非本地的存储器单元中。get()和put()操作的完成无须通知被访存储器的处理器。因此像共享存储器一样，单边通信需要用某种同步协议来保护关键的程序变量，以保证处理器不会误用陈旧的数据。

“单边”一词源自这样的特性，即通信操作可以仅由传送的一边发动。

2.6.3 消息传递

与其他两种通信机制相比，消息传递机制是最初等的，而且对硬件支持的需要也是最少的。由于不支持共享地址空间，处理器只能访问本地存储器。要访问非本地数据，必须使用消息，其中最基本操作是发送和接收操作，通常用send()和recv()表示。send()操作传输一个消息到某个指定的处理器。该消息是一些数据，它们连续存放在发送端的存储器中。recv()操作则接收来自某个其他处理器的消息，并指定本地缓存器的地址以接受该消息。

因此，消息传递是一个双边机制，表示源和目的处理器必须以合作方式传递消息。因为消息传递是由数据值的拥有者发动的，对于某些计算范例，就可能需要比较麻烦的协议。例如，一个处理器不能简单地访问一个远程的工作队列。相反，它必须向工作队列的管理者请求工作。但是为了能收到这样的一个消息，管理者必须期待此请求，通常它需要执行一个查询循环以检查是否有任何到来的消息。

另一个复杂的问题是程序员必须考虑分布式的数据结构以及使用两个完全不同的移动数据的机制：存储器访问用来访问本地存储器，而消息传递用来访问非本地存储器。而在另一方面，消息传递程序以它们代码中具有明确定义的部分相互进行交互，因此被认为比共享存储器程序更易排错。这一点也适用于单边通信，因为它也是用显式方法识别通信操作。

2.6.4 存储器一致性模型

存储器一致性概念的问题已经提上议事日程，因为大多数现代的微处理器利用时延隐藏技术来改进性能，但这种技术也影响了并行程序的语义。存储器一致性模型与实现共享地址空间的并行计算机有关，而不论共享地址空间是通过共享存储器实现还是通过单边通信实现。

最直观的模型是*顺序一致性*（sequential consistency），在这个模型中，按任何执行顺序执行的结果总是相同的，如果：（1）所有处理器的操作按某种顺序次序执行，（2）每个处理器中的操作按其程序所指明的次序进行。不幸的是，由于约束了如缓冲和流水那样的时延隐藏技术的使用，顺序一致性限制了多处理器的性能。

与顺序一致性相比，*松弛一致性*（relaxed consistency）模型实现的是一种弱次序约束。为了了解这类模型的动机，请回忆在过去的年代里，以处理器周期衡量的存储器时延一直是在稳定地增长。（从根本上讲，构建又大又快的存储器是很困难的，所以存储器时延的改进无法跟上CPU时钟速度的改进步伐）。为了减少写操作的时延，现代的微处理器使用*写缓冲器*，它与处理器驻留在同一个芯片上；当处理器发出一个写命令时，数据就存放到写缓冲器，而处理器则继续执行而无须等待数据被确切地写入主存。此时，DRAM就可以为后继的访问其他地址的读操作服务，而不必等待被缓冲的写操作的完成，这样就减少了它们的时延，但这违反了顺序一致性。已经提出了各种松弛一致性模型，都是企图在形式化存储器系统的行为的同时，允许硬件能自由地减少存储器操作的时延。不幸的是，许多不同松弛一致性模型的

差异非常微妙，且它们复杂的语义使得并行程序设计相当困难。

为了理解松弛一致性的困难，考虑有关两个处理器之间互斥的Dekker经典算法的一个简化版本，其代码为如下，假设开始时flag1和flag2均为0：

```
处理器 1:                    处理器 2:
flag1 = 1;                   flag2 = 1;
if (flag2 = = 0)             if (flag1 = = 0)
{                            {
/*临界区*/                   /*临界区*/
}                            }
flag1 = 0;                   flag2 =0;
```

在顺序一致性模型中，flag1或flag2总有一个会先被置位为1，而所有的后继读将认可这一事实，所以我们能保证在临界区内部在任何时间最多只有一个flag为0（flag1或flag2），这样就能在临界区内保证互斥的实现。

在松弛一致性模型中，处理器1可能将flag1置成1，而该值可能仍在写缓冲器中，处理器2可能读flag1但得到的却是旧的值（即0）。这就破坏了只有一个flag为0的法则，以至于无法保护临界区。

为支持所选择的顺序一致性，现代的微处理器实现了专门的原子操作，它能保证不使用写缓冲器。例如，下面的代码使用一条test_and_swap指令，它能自动地测试一个存储单元的旧值，并将其置成为某个新值。

```
/* lock = 0时临界区为空 */
do
{
   old = test_and_swap (&lock,1) ;
} while (old != 0);
/* 进入临界区 */
test_and_swap (&lock,0) ;
/* 退出临界区：清锁以使其他进程能够进入 */
```

因为test_and_swap操作是一个原子操作，且不使用写缓冲器，因此只有一个处理器能将该值置为1，而同时读出旧值0，从而保证了对临界区的互斥。不像Dekker算法，上面的代码在多于两个处理器的情况下仍能工作。

直到如今，没有人能清晰地明确一个像顺序一致性那样实用的存储器一致性模型。再回到前面，从讨论中我们得到的一个比较重要的教训是，并行程序设计常常会削弱抽象，阻止我们隐藏某些低层的细节。

2.6.5 程序设计模型

本小节已对硬件的通信机制作了讨论，隐含的假设是程序员将能直接使用所提供的这些工具。当然我们也应注意，我们总会构造与底层硬件接口不相匹配的软件接口。一个明显的例子是消息传递接口（在第7章中将对其进行进一步的介绍），几乎无处不实现，甚至在实现共享存储器的硬件上也不例外。也有人提出虚拟共享存储器，尽管底层硬件不支持共享存储器，但仍能用软件来实现共享存储器，但是这类系统通常性能较差。第8章将提出一个更高级的程序设计语言，可以构建在采用共享存储器、单边通信、抑或是消息传递机制进行通信的机器上。

2.7 进一步研究通信

由CTA模型指定的非本地存储器的巨大时延λ代表了一个极端的开销。如果能够避免，我们就可使程序运行得更快。减少λ的影响几乎成了我们进行程序设计的核心。本小节将回答这样的问题，“为了减少通信时延我们能做些什么？”一个肯定的回答当然是简化程序设计。

要让P个处理器相互直接通信，例如，处理器p_i要对处理器p_j的存储器用直接存储器访问（DMA）方式进行访问，就需要有线路连接p_i和p_j。对图2-11中的拓扑进行快速回顾后，就可发现，并不是所有处理器对之间都有直接连接。从技术上讲，CTA模型中没有处理器直接连接到其他处理器；每个处理器与其他任何处理器至少需要有一跳间隔，因为它必须进入网络。可是，如果在所有情况下处理器对能以一跳完成通信，我们就认为这就是“直接”连接，即不需要通过网络导航。就图2-11中的拓扑而言，信息必须通过网络进行交换，从而导致交换延迟、冲突、堵塞等等。这些现象延迟了数据的移动。

从数学观点而言，在P个处理器的所有对之间进行直接连接本质上有两种方法：总线和交叉开关。

- 在*总线*的设计中（图2-3），所有处理器连接到一个公共的线集。当处理器p_i要与处理器p_j进行通信时，它需要在总线上传输数据，这样就不可能有其他处理器对同时进行通信，因为它们的信号会干扰p_i和p_j的通信。这样，尽管任何两个处理器间有直接的连接，总线只允许每次有一个通信操作进行，这就意味着通信操作是*串行化的*。
- *交叉开关*的设计（图2-4）通过连接每个处理器到每一个其他的处理器来克服串行化通信的问题，它允许任何不同处理器对的集同时进行通信。从计算的角度来看，该设计是很理想的，因为它允许直接、无冲突的数据通信，但该方法过于昂贵。实现交叉开关所需的连线数以n^2增长，除了非常小的计算机，例如$n = 32$或更小，这种方案是不现实的。

就两种可用的基本设计而言，直接连接只适用于处理器数很小的情况，惟有如此才能减少通信操作竞争的可能性（采用总线时）或是减少器件的成本（采用交叉开关时）。

由于直接连接的困难，体系结构师们已发明了许多有不同拓扑和通信协议的通信网络，以使并行计算机能实现规模扩展。有关这一主题有许多的参考文献，而图2-11只给出了几个代表性的互连拓扑。所有这些互连网络所实现的连接性均低于交叉开关，且使用较少的资源，从而导致更长的延迟和较低的成本。较大的延迟迫使我们采用大的λ值。此外，λ值随计算机规模（相应的，通信网络规模）的扩展而增大，且通常不是平滑的。

2.8 CTA模型的应用

回忆我们在第1章中求解统计3问题时假设采用一个共享存储器接口。开始用一个直截了当的方法（尝试1），发现它存在竞态条件。然后，我们作了改正（尝试2），又发现由于公共变量count的使用导致很差的性能，因为它引入了过度的锁开销。然后我们再次加以修改（尝试3），发现由于存在假共享性能仍不理想。最后的程序（尝试4）达到了合理的性能，不过我们将在第4章中对该代码做再一次的改进。

CTA模型会对求解统计3问题提供好的指导吗？是的。CTA模型由于独立于共享存储器通信机制或高速缓存，不会引入尝试2或尝试4中所改正的问题。它也会指导我们避免犯尝试3所

改正的错误，即单个全局变量count和对其更新而引起的锁竞争。该模型已经告诉我们使用单个全局变量意味着许多的访问将会是非本地的，从而导致仅为了更新count的λ开销；从而也知道一个更好的方案是创建局部的count，然后再将它们加以组合。经该模型的指导，我们应该能在第一场合就写出一个较好的程序。

应注意的是，模型能预测可能存在的问题（依赖于单个全局变量）和修改方法（将局部变量分配给每个线程），但不能预测产生问题的确切原因。模型担心的是访问单个全局变量的高代价，而问题的实质在于许多线程访问单个全局变量的高代价。不同的解释并不是问题，只要模型能识别这些不好的案例并指导我们采取正确的补救办法，CTA模型作到了这一点。但CTA不是一个实际的机器，它是对大量机器系列通用化的抽象模型，所以它不可能匹配每一种机器。但是就编写高质量的程序所需的信息而言，CTA模型提供了有关并行计算机操作的全面的指导，在忽略细节的同时，抓住了并行计算的基本特性。因而，某些实现存在访问全局变量的存储器时延问题，而某些实现则没有，但是它们会有其他的问题，如竞争或是其他更奇怪的问题。不同的实现将以不同方式表现并行计算机的基本行为。

2.9 小结

并行计算机之间的差异很大，如6台计算机的情况所示。通晓所有并行计算机的细节并编写能在任何平台上运行的可移植的程序，这几乎是不可能的。为解决这一问题，我们采用了CTA模型（一台抽象的并行计算机）作为我们进行程序设计的基础。借助对抽象机设计程序（以同样方法对RAM模型设计顺序算法），我们所设计的程序就能运行在所有按CTA模型化的机器上，它实际上代表了所有多处理器计算机。

历史回顾

有关本章的主题有许多文献。从弗林分类法开始直到当今处理器设计，已经引起了广泛的研究。在理论社区界，PRAM模型是广泛分析的基础旨在识别并行的极限，此外，它也是许许多多并行算法的基础，包括1975年的计算最大值的Valiant算法。CTA模型作为一种统一并行计算机的方法于1986年提出。有关处理器间通信的三种主要方法中，共享存储器方法和消息传递方法已衍生出了大量文献。

习题

1. 1024个数用4个线程进行相加，并假设将一个单独变量sum分配到一个处理器的存储器中（第1章中的尝试2），则由CTA模型所预测的通信代价（以λ表示）将为多少？假定数据在4个处理器间均匀分布。
2. 重做习题1，但修改该算法，使得每个线程保持sum[⊖]的一个本地拷贝（尝试4）。
3. CTA 是如何对Sunfire E25K建模的？
4. CTA是如何对机群计算机建模的，特别是HP Cluster Blade Platform 6000的？
5. CTA的互连网络说明多个处理器之间可以同时通信，隐含着数据的并行传输，但总线操作是串行化的。那么为什么用CTA对SMP建模会是合理的？

⊖ 原文误为count。——译者注

6. 假定BlueGene/L的每个处理器中有一个数，即每个结点有两个数；估算完成这些数的求和并将该总和值广播给每个处理器所需要的时间。
7. 用WWW查找其他“刀片机群”的配置，并估算它的λ值。
8. 单个处理器是0−立方，两个处理器是1−立方。给定两个n−立方，连接它们相应的单元可生成一个(n+1)−立方。在一个n−立方中，连接任意两个结点所需的通路长度为多少？
9. 在BlueGene/L机器的3维环绕网中，如果一“跳”是指将一个包文从一个结点送到它的6个邻结点中的一个，则将一个包文从结点（1, 2, 3）送到结点（4, 5, 6）需要多少跳？
10. 如果BlueGene/L没有全局+−reduction（加归约）的组合线路，则处理器就需要实现如图1-3的树排列，问该树的深度（即层数）将为多少？

第3章 性能分析

性能也许是并行计算最重要的目标，而我们在第1章中已经看到要获得高性能是多么困难。本章我们将介绍有助于我们分析并行性能的一些概念。我们从区分并行性和性能之间的不同和定义基本术语开始。在识别各种限制并行性能的因素之后，将解释相关性、局部性以及粒度的见解。然后我们将更深入地讨论为生成高效的程序必须考虑的许多折中方法。最后，我们将论述衡量性能的各种方法，它并不如我们所想像的那样容易，此外，我们将使用这些技术展示可扩展性很难获得的原因。

3.1 动机和基本概念

如第1章中加一个向量数例子所指出的那样，不同的程序可以实现不同的并行数量，尽管它们需要相同的工作量；在上述情况下，即是每个程序需完成相同的加法次数。自然的求和循环导致一个顺序的操作规程，如果按此规程执行就需要$O(n)$时间，因为它没有考虑其他进程参与求解。树的求和方法允许同时完成子计算，当有足够处理能力时就可获得$O(\log_2 n)$的执行时间。这是最好的可能解吗？是哪些约束限制了最好性能的获得？还存在其他机遇未被开发吗？我们将在本章中探讨这些问题。

3.1.1 并行和性能

理想情况下，在一个处理器上用T时间完成求解的问题，在P个处理器上应能用T/P时间完成。当然，存在很多理由使这种理想很难实现。首先，需要明确至少存在P倍并行性。其次，如我们在第1章的统计3的个数求解中所见到的，并行计算通常会引入顺序计算中不存在的那些开销。再次，即使有良好设计的程序，满足T/P目标的挑战随着P的增加也将变得越加困难，这是因为，例如，与开销的代价相比，并行的边际获益减小了。此外还有更为复杂的情况，在某些情况下，P个处理器可产生比所预测的T/P评估值还要短的执行时间！因此，并行与性能是相关的，但它们并不等同。

本节的其余部分将定义我们将要详细探讨的那些术语。

3.1.2 线程和进程

描述并行有两个常用的抽象：线程和进程。

*线程*是指控制线程，逻辑上它由程序代码、一个程序计数器、一个调用堆栈以及适量的线程专用数据（包括一组通用寄存器）所组成。线程共享对存储器的访问，所以线程间能通过读或写对它们都可见的存储器互相通信。（线程也共享对文件系统的访问。）用线程进行程序设计被称为*基于线程的并行程序设计或共享存储器并行程序设计*。

*进程*是拥有私有地址空间的线程。进程间的相互通信需借助传递消息（也有借助文件系统的情况，但非常少）。用进程进行程序设计常被称为*消息传递的并行程序设计或非共享存储器并行程序设计*。

因为进程比线程有更多的相应状态，因此创建或销毁进程的开销要比创建或销毁线程的大得多。因而，进程趋向于长期生存，而线程则随着计算的进行不时地动态派生或销毁。

3.1.3 时延和吞吐率

在开始讨论性能问题之前，我们必须对性能的含义有一致的观点。虽然我们常讲加速计算，但还可能存在两个“比速度更好的”指标：时延和吞吐率。它们分别代表不同的问题和不同的解决方案。

时延：时延是指完成一个给定工作单位所需的总时间。

吞吐率：吞吐率是指每单位时间能完成的总工作量。

开发并行性通常能改进吞吐率。例如，流水化的微处理器通过将一条指令的执行分成几个阶段并流水化多条指令的执行，就可在每秒内执行更多的指令，如图3-1所示。在这个例子中，完成每条指令的时间并没有减少（事实上通常还有所增加），但在每秒中却能完成大量的指令。如果具有同时执行5条指令的能力，则吞吐率就可潜在地增加5倍，因而也就减少了程序的执行时间。

有时我们开发并行性是为了隐藏时延。在20世纪60年代推出的Multics操作系统中就已经应用了这一概念。代之以等待一个长的时延操作，处理器切换上下文环境，并转去执行另一个进程。这一原理（与其闲置，不如使计算的其余部分前行）经常被使用。时延隐藏技术实际上并没有减少时延，它只是隐藏了时延的代价，即由于等待而丧失的执行时间。还需注意的是，这种技术依赖于有足够多的可用并行性（独立的任务），以使处理器在等待长时延事件时仍能处于繁忙状态。所需的并行性数通常随我们试图隐藏时延的增长而增加。

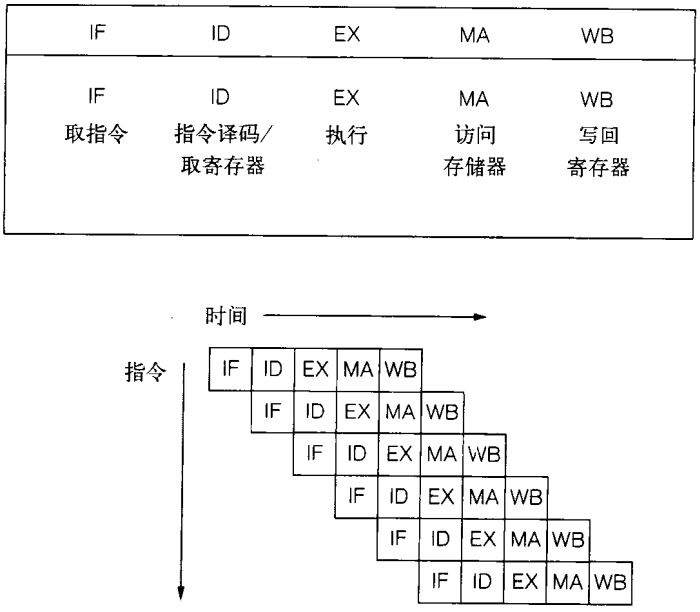


图3-1　简化的处理器流水线。将指令的执行分为5个相等部分，理想情况下可同时执行5条指令，与执行单条指令相比执行时间可改进5倍

毫不奇怪，为了实际地减少时延，还有许多利用并行的硬件技术，如使用cache和存储器预取。

3.2 性能损失的原因

尽管我们满怀希望地相信使用P个处理器能使计算加速P倍，但以下4个基本理由表明这种希望可能无法达到。这些理由有时重叠，它们分别是：

1. 顺序计算不需要付出的开销
2. 不可并行化的计算部分
3. 闲置的处理器
4. 对资源的竞争

所有其他的开销来源均是这4个理由的特殊情况。

3.2.1 开销

任何出现在并行求解中但不会出现在串行求解中的代价被认为是开销。建立线程和进程以并发执行，以及撤消线程和进程均存在开销，图3-2给出了这种开销的示意图。

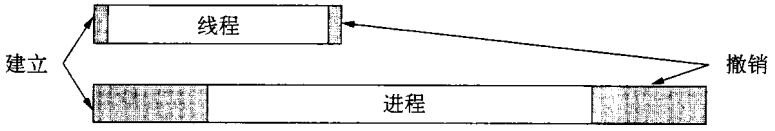


图3-2 线程和进程的建立和撤消开销的示意图

由于存储器的分配及其初始化非常昂贵，因此进程的建立开销远大于线程。在第一个进程建立后，所有后继的线程和进程的建立所形成的开销在顺序计算中不会出现。这些代价表示并行化的开销。

通常认可的并行开销来源有以下4种。

通信 线程和进程间的通信是开销的主要部分。由于在顺序计算中，处理器并不需要与其他处理器进行通信，因此在并行计算中所有通信都是一种开销。例如，在统计3的个数求解中，假共享是处理器间通信的一种开销（尽管这不是所希望的通信），因为cache行将在不同处理器的cache中来回跳跃。在统计3的个数的计算例子中，某些通信开销（如与共享计数器的通信）是不可避免的，而其他的通信，如由假共享引起的，可以通过提供其他资源（如额外的存储器）加以避免。

通信的具体代价与硬件的细节密切相关。表3-1中列出了不同通信机制的硬件通信代价的各种组成部分。

表3-1 由通信机制引起的通信开销源

机制	通信代价的成分
共享存储器	传输延迟，一致性操作，互斥，竞争
单边通信	传输延迟，互斥，竞争
消息传递	传输延迟，数据编组，消息格式化，数据解组，竞争

同步 当一个线程或进程必须等待另一个线程或进程中出现的事件时就存在同步开销。例如，一个线程可能等待某个进程计算一个值或释放资源。比如，在统计3的个数例子中，线程化的代码需要获取或释放锁就是同步开销，因为在顺序代码中不需要完成这种操作。我们从该例中还可看到如果锁的获取次数很大的话，那么这种开销就会非常可观。在许多消息传递

方式中同步是隐式的，而在用线程进行程序设计时，则同步通常是显式的。

计算　并行计算几乎总是要完成一些额外的计算，这些计算在顺序求解时是不需要的。例如需要计算出每个线程要负责完成计算中的哪一部分。在统计3的个数例子中，这种开销是很小的。另一个例子是，出现在顺序程序中的计算同样也要出现在每个线程中；初始化便是一个例子，因为t个线程将执行该代码t次，而不是一次。这些冗余的计算代表不可并行代码的特殊情况，关于这一点我们稍后将进行更详细的讨论。

存储器　并行计算常常导致存储器的开销。虽然这种开销并不总是会影响性能（在统计3的个数例子中，作为存储器开销的数据额外填充还可改善性能），但对并行计算来讲可能很重要，因为计算的规模受制于存储器的容量。

3.2.2 不可并行代码

当然，如果一个计算在本质上是顺序的（即表明它不可能并行化），那么使用再多的处理器也无法改进性能。有时这种代码以冗余计算形式表示在一个并行计算中。例如，如果顺序和并行计算两者都循环k次，那么循环的开销，如递增归纳变量（induction variable）和测试终止条件，并不会因并行而增多。在其他情况中，计算由单个线程或进程完成，而其他的线程或进程则处于闲置状态。一个可能的例子是磁盘I/O，由于某些处理器不附属于网络，因而在执行I/O操作时，这些处理器就不得不处于闲置状态。

阿姆达尔定律　不可并行计算的存在是很重要的，因为它将限制并行化的潜在好处。阿姆达尔定律指明如果一个计算的$1/S$本质上是顺序的，那么最大的性能改进将受限于因数S。其论证如下，一个并行计算的执行时间T_P将是顺序部分计算时间和可并行化部分计算时间两者的和。如果该计算顺序地执行需要花费的时间是T_S，则当有P个处理器时，T_P可表示为

$$T_P = 1/S \cdot T_S + (1 - 1/S) \cdot T_S / P$$

假想P值非常大，使得可并行化部分的执行时间可以忽略不计，则最大可改进的性能将是因数S。也就是说，顺序执行代码在计算中所占的比例决定了使用并行手段所能改进性能的潜力。

阿姆达尔定律（Amdahl's Law）　阿姆达尔定律是由IBM公司的计算机体系结构师吉恩·阿姆达尔在1967年发表的论文中提出的。这一定律与供求定律（Low of Supply and Demand）具有同样的意义：如前面的方程式所示，它描述了程序执行时间中两部分的关系。两个定律均是解释重要现象行为的有力工具，且两个定律都将影响该行为的其他参量视为常数。特别地，阿姆达尔定律适用于单程序场合。

实际的情况可能比阿姆达尔定律暗示的更坏。一个明摆着的问题是计算的可并行化部分可能无法改进到极致（即只要增加仍可继续改进性能的处理器数总有一个上限），因此并行执行时间不大可能消失。此外，如在上一节所提到的，一个并行实现经常会比顺序求解需要执行更多的总指令数，导致低估$(1 - 1/S) \cdot T_S$值。

包括阿姆达尔在内的许多人对该定律的解释表明使用大量的处理器求解问题只能获得有限的成功，但这似乎与大量的并行计算机能显著改进计算性能的报道相冲突。到底哪一个是正确的呢？

阿姆达尔定律描述的一个关键事实是它只适用于计算的一种场合，即施行并行化后计算中的顺序部分将占据执行时间的主要部分。阿姆达尔定律是在固定应用规模的前提下考虑并行性增长的影响。但大多数并行计算则是固定并行性而扩展应用的规模。在这种情况下，随

着所考虑规模的增加，顺序代码所占的比例就越来越小。所以，将问题规模翻倍后，顺序部分的增长几乎可以忽略，从而使得求解问题有更多的部分可以并行执行。

概括地讲，阿姆达尔定律并不否定并行计算的价值。相反，它提醒我们要想达到并行性能就必须考虑整个程序。

3.2.3 竞争

为争夺共享资源而引起的竞争将使系统性能衰减。我们将竞争视为是开销的一种特殊情况，但竞争值得我们特别的注意，因为它的影响经常会降低系统的性能，甚至比单处理器的还要差。例如，我们在第1章中已经见到如何由于锁的竞争而导致存储器的过度负载，最终降低了系统性能。特别是，如果锁实现为*自旋锁*，此时一个等待的线程将重复检查锁的可用性，这种等待锁的操作将增加总线的通信量。这种竞争特别有害，因为它影响所有企图访问此共享总线的线程，尽管有些线程并不竞争这同一个锁。第1章还说明了第二种形式的竞争，即由假共享引起数据值在不同的cache中来回跳跃，导致性能的衰减。在那种情况下，两个本地cache竞争相同的cache行，但所有处理器所感受到的效果是增加了总线的通信量。

3.2.4 空闲时间

理想情况时，所有处理器在所有时间都忙于工作，但实际可能并非如此。一个进程或线程由于缺少工作或因为正在等待某种外部事件的发生，如来自其他进程的数据，将无法继续执行。因此，闲置时间通常是同步和通信所造成的。正如下一节的相关性所展示的，闲置时间将以多种方式显示出来。

负载不平衡 闲置时间的一个公共来源是处理器负载的不均衡分布，这就是常说的*负载不平衡*。极端的例子出现在一个顺序计算只运行在并行计算机的一个处理器上，而其他的处理器处于闲置状态。在其他的情况下，可能工作是并行执行的，但工作负载的分配不平衡。

例如，在第1章的基于树的并行求和中，第1组的成对求和计算需要$n/2$个处理器，第2组的成对求和需要$n/4$个处理器，等等。如此，在第1步的计算后，在求和完成时，有一半的处理器处于闲置状态。在第4章中，我们将看到一个更聪明的方法以减少这种不平衡的影响。

存储器受限计算 一个处理器如果正在等待一个存储器操作时，如读DRAM时，也可能发生停顿。存储器系统的性能可用以下两个参数衡量——带宽和时延。由于历史的原因，DRAM的时延改进的速度比处理器速度的改进速度要慢，所以就CPU周期而言，与DRAM的距离不断拉大。当数据能存放在cache中时，就可以避免这些时延，但是当计算显示出差的局部性或当要访问大量数据时，就必须经常访问工作速度慢的DRAM。DRAM带宽之所以重要有两个原因。首先，许多时延隐藏技术，如预取和硬件多线程，在企图减少存储器时延时引入了额外的存储器通信量。其次，即使计算的工作集能驻留在cache，但当它们装载这些cache时也必定会消费存储器的带宽。

多核芯片的带宽约束 *当核的数目增长正比于指数增长的晶体管数时，存储器带宽的约束就成为多核芯片中的特别问题，存储器带宽和I/O带宽受芯片脚数的限制，而这通常又受限于芯片的边限。新的工艺技术可能致力于解决这一问题，但这种解决办法还未达到商用程度。*

3.3 并行结构

在了解了各种可能降低并行性能的因素之后，现在转而讨论三个紧密相关的概念，这些概念能帮助我们避免这些问题。相关性的概念将提供一种推断低效来源的方法；粒度概念将帮助我们匹配计算和底层的硬件资源；而局部性的目的是在于指导我们考虑自然地具有合适粒度和减少相关性的解决方案。

3.3.1 相关性

*相关性*是指两个计算间的排序关系。在不同场合相关性以不同形式表现出来。例如，在两个进程间，当一个进程等待来自另一进程的消息到达时，就出现了相关性。相关性也能以读写操作加以定义，对于线程计算来讲就相应于存储器的装载和存储。下面就一个程序需要对某个特定的存储单元在写（存储）之后进行读（装载）这一情况进行讨论。

为了产生正确的结果，在写操作和读操作之间存在相关性，如图3-3所示。如果两个操作顺序在时间上发生错位使得线程2的读先于线程1的写，则就会违反相关性，程序的语义就可能发生改变（参见图1-8中针对此情况的例子）。遵从所有相关性的任何执行顺序的排列都将产生如最初程序所指定的相同结果。因此，相关性的概念允许我们描述和区别哪些执行顺序必须保证而哪些不必，从而保证程序执行的正确性。

相关性提供了一个通用方法以描述对并行的限制，因此它不但有助于推断正确性，而且提供了一种方法以推断性能损失的可能原因。例如，跨线程或进程边界的数据相关性就需要在两个线程或进程间进行同步或通信。在了解数据相关性的存在后，就能先行了解并行的后果，即使我们并不知道是计算中的哪一部分导致了这种排序关系。为了更清楚地了解这一点，下面考虑相关性中一种称为数据相关性的特殊形式。

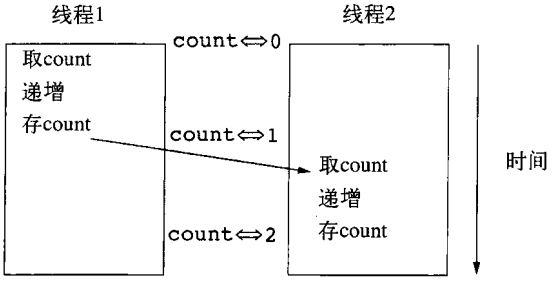


图3-3　一个线程的存储器写（存储）操作和另一个线程的存储器读（装载）操作之间的相关性

数据相关性　数据相关性是指对一对存储器操作的排序，为了保持正确性必须保持这种排序关系。有三种类型的数据相关性：

- *流相关*：写后读
- *反相关*：读后写
- *输出相关*：写后写

刚才叙述的流相关也称为*真相关*，因为它代表了存储器操作的基本排序。与此相反，反相关和输出相关被认为是*伪相关*，因为它们是由存储器的重用而引起的而不是操作的基本排序；虽然将它们称为“伪”相关，但它们仍与我们有关，因为经常希望重用存储器。为了完整起见，有些时候需考虑第四种形式的相关性，即*输入相关性*（读后读），它不会强加任何排

序的约束，但有时会有助于对暂时本地性的推断。

为了了解真伪相关性的差别，考虑以下的代码段：

```
1  sum=a+1;
2  first_term=sum*scale1;
3  sum=b+1;
4  second_term=sum*scale2;
```

在行1和行2之间存在流相关（通过sum），类似地在行3和行4之间有流相关。此外，在行2和行3之间对sum有反相关。该反相关阻止第1对语句和第2对语句的并发执行。但可以看到如果将第1对语句中的sum重命名为first_sum，而重命名第2对语句中的sum为second_sum，就可使这两对语句并发执行，其代码如下：

```
1  first_sum=a+1;
2  first_term=first_sum*scale1;
3  second_sum=b+1;
4  second_term=second_sum*scale2;
```

这样，通过增加存储器的使用，我们就可增加程序的并发性。

与此相反，无法通过重命名变量来消除流相关。初看起来，可以像如下那样直接替换行2和行4中的sum来消除流相关：

```
1  first_term=(a+1)*scale1;
2  second_term=(b+1)*scale2;
```

但实际上却无法消除，因为在以上的两项中不论如何书写表示式，加法必须先于乘法执行。从sum的写入（可能写入一个内部寄存器）到作为操作数的sum读出（可能从一个内部寄存器读出）的这个数据流仍保持不变。

3.3.2 相关性限制并行性

为了了解相关性是如何限制并行性的，请回忆第1章中对n个数求和的例子。它的顺序代码段为如下形式：

```
sum=0;
for(i=0; i<n; i++)
{
  sum+=x[i];
}
```

我们断言以下的代码具有更多的并行性，即使两个代码段都用顺序语言表示：

```
((x[0]+ x[1])+ (x[2]+ x[3]))+ ((x[4]+ x[5])+ (x[6]+ x[7]))
```

图3-4图示了这两个代码段；不含叶子的边代表一个流相关，因为下面的函数将写存储器，而上面的函数将从同一存储单元读出。两段代码的之间的差异现在就很清楚了。在图3-4a中，顺序求解定义了一系列的流相关，必须遵从它们的排序关系。相反，图3-4b指明的是较短的流相关链，意味着较少的排序约束以及允许更多的并发性。实际上，当我们用C语言说明数的相加时，除了说明哪些数相加之外，还隐含说明了相加顺序的约束。（我们需要识别该操作（加）是可结合的，从而知道这两个求解会生成相同的结果。）

关键在于当创建一个并行程序时。我们必须十分小心避免引入相关性，虽然这种相关性对计算无关紧要，但这些相关性将会不必要地限制并行性。（知晓f()是加法这一点，就可以使编译技术将该代码转换成更为并行的形式，但是这种技术的应用范围有限；因此最好是在一开始就努力避免相关性。）

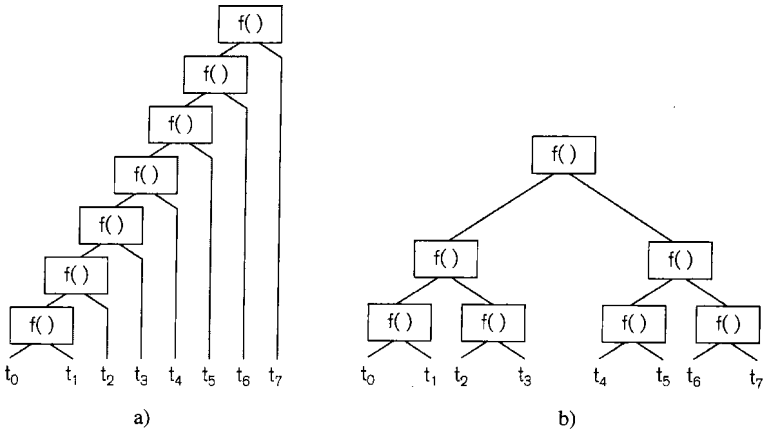


图3-4　顺序和基于树的加法算法示意图；不与叶子连接的边表示流相关

易并行计算　某些计算很容易并行化，因为它们是由大量相互之间显然不存在相关性的线程所组成。我们称这些计算为易并行的。例子之一是曼德勃罗特集，它是一种分形计算，在这种计算中，复数平面上的每一个点可以相互独立地计算。光线跟踪，它是一种图形绘制技术，模拟光线的流动以生成现实的图像，也是易并行计算，因为它跟踪多条独立的光线以绘制一个图像。(传统的递归光线跟踪算法是易并行的，但这种算法在带宽受限的多核芯片上并不能很好运行，因为它不能有效地利用存储器层次。)

3.3.3 粒度

管理由相关性引起约束的一个关键概念是粒度，通常使用*粗*和*细*来描述粒度，虽然也可用*大*和*小*来描述。

并行的粒度　并行的粒度由线程或进程间的交互频率所决定，即跨越线程或进程边界的相关性的频率。这里的频率是用交互之间的指令数衡量的。因此，粗粒度是指线程和进程依赖于其他线程或进程的数据或事件的频度是较低的，而细粒度计算则是那些交互频繁的计算。粒度的概念很重要，因为每次交互必将引入通信或同步，以及它们相应的开销。

实现低时延通信的硬件平台，如多核芯片，支持较细粒度的计算。而实现高时延通信的硬件平台，采用较大的粒度就较好，因为交互的开销较高。由于消息传递的软件开销通常很大，因此消息传递程序通常最适宜粗粒度计算，即使在低时延的通信基础上也为如此。如我们在本章后面将要讨论的那样，存在许多创建粗粒度计算的技术，这些技术一般以损失存储器或计算为代价，以减少交互的频率或隐藏这些交互的时延。

粒度概念的应用　一个重要的经验是在所有情况下最好不要固定粒度。相反，应该使计算的粒度与低层硬件可用资源以及求解问题具体需求相匹配。例如，第1章中所叙述的最初的前缀求和是一个只含有很小工作量以及与相邻线程进行细粒度交互的细粒度计算。

在极端的情况，最粗粒度的计算含有巨量的计算而无交互。BOINC（Berkeley Open Infrastructure for Network Computing，伯克利网络计算开放设施）就支持这种计算，它允许将子问题分布到个人计算机上并完全在本地求解；唯一的通信是在最后将结果报告给问题的分配者。在这种框架中，并行计算机就是用因特网连接的PC机集合。采用这种超粗粒度的策略是非常关键的，因为因特网的通信有很大的时延。

另一个极端则是运行在多核芯片上线程，由于处于同一芯片上的各处理器间的通信具有很低的时延，因此在交互间使用具有几百条指令的细粒度线程是可行的。

3.3.4 局部性

与粒度密切相关的概念是局部性。计算能显示出时间局部性（存储器访问在时间上成簇）和空间局部性（存储器访问在地址上成簇）两者。请回忆局部性是计算中的重要现象，它是cache能工作的原因。当然，并行计算机的处理器也使用cache，所以所有时间和空间局部性的得益均是可能的，只要访问保持局部性以使得被访问的对象均处在cache中。的确，运行在数据块而不是单个数据项上的算法总是显示出空间局部性，因此这种算法总是受青睐的。

在并行的环境中，局部性还具有使线程或进程间相关性减到最小的附加好处，从而能减少开销和竞争。如在前面所描述的那样，非本地的访问意味着某种形式的通信，它是纯开销，所以会限制并行性能。此外，为了进行非本地的访问，线程或进程常会竞争互连网络或是存储器系统资源。

为使局部性的优点更加具体，考虑统计3的个数问题的最后一个解（尝试4）。由于工作在连续的存储器块上，每个线程开发了空间局部性；由于在本地变量上累加中间和，每个线程开发了时间局部性；由于每个线程只更新共享和一次，使通信减少到最低程度，从而改进了局部性并减少了开销和通信。请注意，使用本地的累加变量是用少量的额外存储器消除假相关的又一个例子。

3.4 性能协调

前面几节已经解释了相关性是如何限制并行的，以及为什么考虑局部性和合适的粒度是重要的。我们现在解释为什么产生高效的并行程序可能很困难。首先与顺序性能分析情况作对比，然后讨论多种可行的协调。

一般来讲，推断顺序计算的性能是非常直截了当的。对于大多数程序，只要统计程序所执行的指令数就可以了。在某些情况下，我们意识到存储器系统性能是瓶颈，所以只需寻找减少对存储器使用或是设法改进存储器系统性能的方法。在这种框架中，通常要求程序员避免过早地使用90/10规则进行优化，90/10规则是指90%的程序执行时间将花在10%的程序代码上。因此，一个谨慎的策略是不加任何修饰地编写程序，如果它的性能需要改进，则将支配执行时间的10%代码识别出来。然后，对这10%的代码进行重写，也许重写时要使用能提供更大控制的其他语言，如C或汇编语言。

遗憾的是，并行程序的情况要复杂得多。正是阿姆达尔定律告诉我们，不能并行的10%的计算将限制我们可能从并行化获得的最大性能改进最多为10倍，而这在许多场合是不能满足要求的。此外，决定性能的因素不只是指令数，还包括通信时间、等待时间、相关性等等。动态的影响（如竞争）则与时间有关，且随不同求解问题和不同机器而会有所不同。因此，控制代价和识别瓶颈就会比顺序情况更加复杂。

我们已经了解了通信代价、闲置时间、等待时间以及其他能影响并行计算性能的参量。复杂的原因在于当企图降低某一因素的代价时可能会增加其他因素的代价。因此了解这些因素的协调（trade-off）是很重要的，因为不同的算法和不同的计算平台将青睐不同的协调方案。下面将讨论这些问题。

3.4.1 通信和计算

通信代价常常是开销的重要来源。通过完成附加的计算常常可减少通信。

重叠通信和计算　减少通信代价的一种方法是重叠通信和计算。关键在于要识别出独立于通信的计算。通过并发地执行计算和通信，通信的时延就可部分或完全地被隐藏。在第6章和第7章中将通过实例展示这一技术。从性能角度看，实现重叠通信和计算通常是无需代价的。从程序员的角度看，通信和计算的重叠可能会复杂化程序的结构，而且有时会采用非一般的方式。

冗余计算　减少通信代价的另一种方法是执行冗余计算，即在本地重新计算一个值，而不是等待该值从别处传输过来。例如，在第2章中我们观察到由所有进程本地产生的随机数r优于由一个线程产生该值，然后使该值与其他线程共享。不像重叠通信和计算，冗余计算将带来代价，因为所有进程必须执行随机数生成代码。换句话说，我们通过增加总的指令执行数来消除通信的代价。只要冗余计算的代价小于通信代价（对于像产生随机数那样的简单计算一般均成立），进行冗余计算就是值得的。

请注意，冗余计算也消除了生成值的进程和需要值的进程之间的相关性。因为这种相关性常常导致同步的开销，因此即使附加的计算的代价和通信的代价完全相当，就消除相关性而言也是值得的。在随机数生成的情况中，冗余计算消除了一个客户进程不得不等待服务器进程产生所需值的可能性。对于这种情况的考虑使协调的评估更为复杂。

3.4.2 存储器和并行性

通常增加使用存储器的代价可以增加并行性。在某些情况下，少量的额外存储器能显著改进性能。（当然，我们必须非常小心，因为存在一些情况，其中并行计算的主要益处是有较大的可用存储器容量，所以我们不希望丢弃这种益处。）

私有化　通过使用附加的存储器消除伪相关就可增加并行性。例如，在统计3的个数程序中，private_count变量的使用就可消除线程在每次遇到3时所需进行的交互，其影响是使count的变量数从1增加到线程数t。这只花费了很小的存储器代价，但却因减少相关性而获得了很大的节省。

填充　也可通过填充（padding）增加并行性，填充是指分配额外的存储器迫使变量留驻在它们自己的cache行上。在统计3的个数例子中，我们已看到如何使用填充来消除假共享，当对独立的不相关的变量访问变成相关时，由于这些独立变量被分配在同一cache行上就导致这种假共享的出现。我们可以把假共享看成是伪相关的一种特殊形式，由于消除假共享能减少线程间的交互频率，从而就会使性能获得显著的改进。

3.4.3 开销与并行

并行和开销通常是不协调的。一个极端是，只使用一个线程使所有并行开销（如锁竞争）消失。增加线程数时，并行性就会跟着增加，但开销和竞争也会同时增加。如果在增加线程数时固定求解问题的规模，那么每个线程在同步之间所完成的工作就会减少，使得同步占据了整个计算的大部分。每个线程只负责处理较小的问题规模也隐含着只有较少的计算可与通信重叠，这就会增加数据的等待时间。

正是并行的开销通常影响了处理器数P无限制增长所带的好处。的确，即使计算可以在概

念上分配一个处理器专门对一个数据点进行计算，但通常在$P = n$之前就因为开销过大而被迫放弃。因此，对于给定的输入规模，我们发现大多数程序有一个上限，过了上限之后每一个附加处理器的增益值趋向0，即是说，增加更多的处理器将不会有任何益处。下面我们评价三个开销和并行性之间的协调。

并行化开销 在统计3的个数例子中，各个线程最初使用一个全局变量来累加每个线程的priv_count；如果线程数很大，则最后的累加将成为瓶颈。一个简单的解决方法是使用树求和方法，此时线程成对组合它们的priv_count值。其实质是，线程将累加中间值的任务分成为几个独立的并行活动，从而并行化某些开销并减少了它的成本。

负载平衡和开销 增加并行性也能改进负载平衡，因为均匀地分布大量细粒度工作单位要比均匀地分布少量粗粒度工作单位更加容易。过度分解问题的思想特别适合于工作量不规则或动态变化的那种计算。此时，很难精确地决定向每个处理器分配多少工作，因此对问题进行过度分解就有更大的可能性使所有处理器处于繁忙状态。当然，过度分解的缺点是将增加开销，且通常是增加通信或同步。

粒度协调 上面许多的协调是与并行粒度相关的。减少相关性数的一种方法是增加交互的粒度。批处理是一种程序设计技术，在这种技术中工作以组为单位完成。例如，与其每次传送数组的单个元素，不如更有效地每次传送数组中的整行或整列；与其每次从任务队列中抓取一个任务，不如每次抓取几个任务以减少竞争。批处理增加了交互的粒度，从而减少了它们的交互频率。

很自然，粒度带来的好处能增加到何种程度会有一个限制。最佳的粒度常常依赖于算法的特征和硬件的特征。当填充priv_count数组以避免假共享时，我们已经看到匹配数据结构的粒度和底层硬件的优点。在那种情况下，解决方案没有直接采用减少并行性的方法，但增加粒度的通常结果是减少了并行性。一般而言，当系统具有高的通信和同步时延时值得采用粗粒度计算，因为它在充分开发可用并行性的同时尽可能多地避免了交互。

函数程序的并行化 由于纯函数语言提供了访问（参考）的透明性，即一个表达式的值不会随时间发生变化，一个表达式的子表达式能以任意顺序求值而不影响其结果，因此函数式程序能提供丰富的并行性。的确，函数程序并行化领域有很长的研究历史。但在本章中我们已表明我们更多地是关心如何获得良好并行性能而不是简单地识别并行性：我们必须小心地管理线程间数据的移动和交互，我们还必须小心地选择一个合适的并行粒度。所以，虽然函数式语言能使识别大量的细粒度的并行性变得非常简单，但是函数式语言没有提供控制局部性、粒度以及跨线程的相关性的机制。就局部性而言，函数式语言抽象掉存储单元的概念，从而使得程序员或编译器很难推断局部性。就粒度而言，函数式语言虽然能够为每一个函数子表达式派生一个新线程，可是在新线程完成很少工作后就必须与另一线程进行同步；在这些情况中，额外的线程开销可能超过了所有从并行所获得的性能。不幸的是，很难知道组合哪些表达式来创建更粗粒度的线程。最后，如果没有能力推断局部性或粒度，则推断跨线程的相关性也将非常困难。

3.5 性能度量

如我们所看到的，当论及并行计算时，性能的概念就变得复杂，因为此时必须多方面加

以考虑，包括存储器使用、处理时间以及开销。毫不奇怪，正确的使用性能指标包含着许多微妙之处。我们现在讨论几个性能指标重点在于了解它们的优缺点。

3.5.1 执行时间

也许最为直观的指标是*执行时间*，也称为时延。它的简单定义是指从第一个处理器开始执行程序到最后一个处理器完成执行所花费的时间。我们将在第11章中再次讨论这个定义，就会发现它比我们想象的更要复杂。

另一个常用的指标是FLOPS，即*每秒浮点操作数*（floating-point operations per second）的缩写。在科学计算中，通常用FLOPS作为指标数字，并以单精度或双精度报告。使用FLOPS指标的一个明显不足是它忽略了如整数计算那样的其他成本，而整数计算可能也是计算时间的主要成分。也许更重要的是FLOPS速率常常会受特别低级程序修改的影响，这种修改允许程序开发硬件的专门特性，例如组合的乘/加。这种“改进”对其他计算或相同计算的其他计算机来讲几乎没有什么通用性。

上面所论述的两个指标，它们的一个共同局限是将所有性能浓缩成一个数字，而对计算的并行行为则没有任何指示。而我们经常希望能了解改变可用并行性时程序的性能是否会成比例地发生变化。

3.5.2 加速比

加速比定义为顺序程序执行时间除以计算同一结果的并行程序的执行时间。即*加速比* = T_S / T_P，其中，T_S是顺序时间，而T_P是运行在P个处理器上的并行时间。加速比经常作为y轴而作图，此时用处理器数作为x轴，如图3-5所示。

加速比展示了许多并行程序的典型特征，增加处理器数时，加速比曲线会趋平。这一特征是因为在增加处理器数的同时保持问题规模不变的缘故，它导致每个处理器的工作量的减少；由于每个处理器只有少量的工作量，使得诸如开销等代价变得更为突出，导致总的执行时间无法线性地改进。

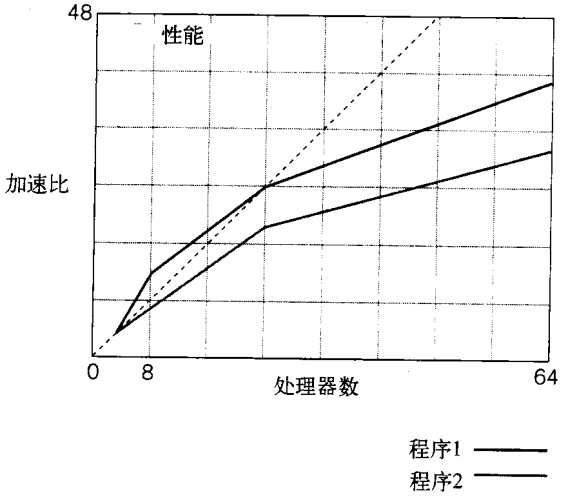


图3-5　展示两个程序性能的典型加速比图；虚线表示线性加速比

3.5.3 超线性加速比

有时会出现一种奇怪的现象，即并行程序能以比顺序程序快P倍的速度运行，称为*超线性加速比*（superlinear speedup）。例如，图3-5中的程序1，当使用8个处理器时就显示出超线性加速比。

超线性加速比似乎违反直觉，因为作为加速比比较基础的一个顺序程序，能很容易地仿真并行程序的P个进程的执行，且执行时间不会超过并行执行时间的P倍。那怎么可能导致超线性加速比？基本的解释是并行程序完成了较少的工作。最通常的情况是，并行执行时所要访问的数据都驻留在每个处理器的cache中，而顺序执行时必须访问存储器系统中较慢的部分，因为此时数据无法全都驻留在单个cache中。当然，超线性加速比现象毕竟是很少的，因为存储器系统更有效的使用必定会克制所有的并行开销。

第二种超线性加速比情况出现在当所要查找的元素已发现因而终止搜索时。当搜索以并行方式进行时，能以不同的次序有效地进行，意味着总的被搜索数据量将少于顺序搜索的情况。因此，并行执行将完成较少的工作。

3.5.4 效率

效率是对加速比的规格化衡量，由它表明每个处理器的有效使用情况：*效率 = 加速比/P*。理想效率1指明是线性加速比，且表明所有处理器极尽全力在工作。由于存在各种性能损失源，通常效率总是小于1，且随处理器数的增加而减小。在超线性加速比的情况下会出现效率大于1。

3.5.5 加速比问题

由于加速比是两个执行时间的比，所以它是一个无量纲的指标，这就显得加速比似乎未将诸如处理器速度等技术细节考虑在内。与此相反，这些细节会微妙地影响加速比，所以我们必须小心地解读加速比的数字。以下是几个需要考虑的因素。

硬件 首先，我们应意识到对不同代的机器进行加速比的比较是很困难的，即使它们有相同的系统结构。问题在于一个并行机器的不同部分一般不是等量改进的，从而会改变它们的相对重要性。例如，处理器性能的增加是与时俱进的，但通信时延的改进并不以相同速度进行。因此，在一台新的计算机上，通信时间的减少会比计算时间的减少要小。其结果是，加速比的值通常会随时间而减小，因为一个计算的通信成分相对于处理成分变得更为昂贵。

顺序时间 下一个问题与加速比公式中的分子T_S有关，它应该是给定处理器和问题规模的最快顺序求解时间。如果人为地提高T_S，就会增大加速比。增加T_S的一个微妙方法是对顺序和并行程序关闭标量编译器的优化。初看起来这一举动是公平的，因为关闭编译器优化的举措对两个程序都进行了实施。然而，这一改变将显著地减慢了处理器，改善了（相对而言）通信时延。所以在报导加速比时应说明所用的顺序程序以及编译器优化设置的细节。

相对加速比 不适当地增加加速比的另一个常见途径是将*并行程序单处理器运行性能*作为T_S。在T_1基础上计算的加速比称为*相对加速比*，在报告时应作说明。真实的加速比很可能意味着顺序算法与并行算法有所不同。而相对加速比只是简单地比较同一算法的不同运行，它将适合于并发执行但非并行的优化算法作为比较出发点；由于存在并行开销，这很可能使运行速度变慢，导致较高的加速比。应注意的是，在单处理器上精心编写的并行程序有时会比任何已知的顺序程序运行的要快，也使它成为最好的顺序程序。在这种情况下我们获得的是真

实的加速比而不是相对加速比，而这种情况应明确地加以确认。

不可能总是回避相对加速度。例如，对于大型计算，往往无法衡量一个给定问题规模的顺序程序，因为此时数据结构可能无法全部放置在存储器中。在这种情况下，所能报告的就只有相对加速比。基本的比较点将是小数目处理器上的并行计算，而y轴加速比的绘制应以该量值按比例进行缩放。例如，如果最小可能运行的处理器数$P = 4$，则用$P = 64$时的运行时间相除，就可在$y = 16$处显示正确的加速比。

冷启动　冷启动是另一个不经意影响T_S的因素。偶然获得大的T_S值的一种简单方法是先运行顺序程序一次，并定时所有分页情况和强制（必须）的cache不命中。一个良好的习惯是将一个计算运行若干次，但只测量后面几次的运行。这就可使cache“热身”，以使强制的cache不命中次数不包含在性能测量中，而不至于复杂化我们对程序加速度的理解。（当然，如果程序存在冲突不命中，则就应计算在内。）虽然容易被忽略，不过冷启动很容易进行修正。

外围负荷　更令人不安的是那些相当可观的处理器外活动，例如磁盘I/O。一次性I/O突发传输，例如读入求解问题数据，不会有什么问题，因为定时测量能将其绕过；问题是处理器之外的连续操作。不但因为它们的工作速度相对于处理器来讲很慢，而且它们一般会复杂化对计算的加速比分析。例如，如果顺序和并行求解不得不完成源自同一处的相同处理器之外的操作，则这些操作所需的极大的时间可能完全掩盖了并行性，因为它们将决定测量时间。

在这种情况下，就完全没有必要并行化程序。如果处理器能独立地完成处理器外操作，那么该并行本身就支配着加速比的计算，很可能相当完美。任何包含处理器外负荷的计算必须小心地控制其影响。

3.5.6 可扩展加速比和固定加速比

选择求解问题的规模是一件困难的任务。通常最直观的加速比曲线是针对不同处理器数时始终保持求解问题规模的固定。然而，当处理器数跨越的范围很大时，固定规模的加速比就有了问题，因为当问题规模小到足以放入一个处理器的存储器中时，面对使用几百或几千个处理器时就会很容易显得过小而不合理。主要问题在于，并行计算的效率通常依赖于问题的规模，因此，固定问题规模很可能偏向于某个特定的处理器数。

一个明显的答案是随处理器数的增多扩大求解问题的规模，所以运行在4个处理器系统上的问题规模应是两倍于运行在2个处理器系统上的问题规模。即使是这种情况，也不总能清楚“两倍于”的真正含义，因为大多数计算的渐近复杂性不是按线性增长的。此外，存储器和通信需求并不总是按计算需求的相同速率增长，所以我们对问题规模的任何改变都将可能导致存储器子系统、计算能力或通信基础设施相对重要性的变化。

3.6 可扩展性能

使用更多的处理器就能改善性能这种想法是天真的。这种错误观点随着并行计算机中的处理器数增长的报道可能已被进一步的扭曲。并行计算机趋向于使用更简单和功能不很强的处理器，这意味着处理器数比处理器本身的功能强弱更为重要。确实，已经有预测，未来的多核计算机将使用比目前简单得多的CPU。这一小节在讨论有关可扩展性能的几个问题时，将考虑这些硬件发展趋向的动机。

3.6.1 难于达到的可扩展性能

首先说明可扩展性能难于达到，因为当增加P时，要达到高的并行效率就越加困难。为便于说明，假想一个算法，如一个按字母次序排序的并行算法，在该算法中，用一个进程负责排序所有以相同字母开始的词，其排序过程如下：

a. 从最初的数据结构中移出以给定字母开头的所有词，

b. 将它们局部排序，

c. 在全体排序时将它们放回到合适位置。

如果可用的处理器数小于26个，则每个处理器要处理多个字母。（我们将在第4章看到这种计算的更多细节。）a和c两部分是并行开销，因为b足以顺序地求解此问题。乐观（不现实）地假设当增加处理器数时开销的总量将保持不变，并假设在一个处理器上执行时的开销为20%。

假定局部按字母顺序排序（b）所需的顺序计算时间为T_S。因为20%开销是不可并行化的，因此在两个处理器上的并行求解将需时T_2：

$$T_2 = \frac{T_S}{2} + 0.2T_S$$

$$= 7\frac{T_S}{10}$$

使用两个处理器的效率为：

$$E_2 = \frac{\frac{T_S}{T_2}}{2} = \frac{\frac{10}{7}}{2} = \frac{5}{7} = 0.71$$

使用10个处理器时，执行时间为：

$$T_{10} = \frac{T_S}{10} + 0.2T_S = \frac{3T_S}{10}$$

因此，10个处理器的效率为：

$$E_{10} = \frac{\frac{10}{3}}{10} = 0.33$$

对于100个处理器，则有：

$$T_{100} = \frac{T_S}{100} + 0.2T_S = \frac{21T_S}{100}$$

$$E_{100} = \frac{\frac{100}{21}}{100} = \frac{1}{21} = 0.047$$

这样，对于100个处理器情况，每个处理器只有4.7%时间在进行有用的工作！这一极低的效率表明增加更多处理器的好处随处理器数的增加而减少。这就突出了采用大规模并行系统时使开销最小化的重要性。

3.6.2 硬件问题

以上的例子说明为什么我们能在增加处理器数时使用功能不很强的处理器：随着处理器数的增加，改进每个处理器的CPU速度可获得的好处是很小的。换句话说，使用较慢的核会使不可并行化的部分在整个计算中所占有的比例更小，按照阿姆达尔定律的提示，使用较慢

的核我们反而能获得更好的加速比和效率。因此，P = 64K 的BlueGene/L使用相对较慢的处理器（与第2章中所描述的其他机器相比）是有道理的。

3.6.3 软件问题

如前面所提及的，减少开销的一个常用方法是成批完成任务，这种方式通过增加每个进程的本地工作量从而减少与其他进程交互所花费的时间来增大粒度。批处理对算法的好处犹如具有好的“表面积对体积的比”。例如，一个算法在$m \times m$数据数组上进行本地计算，并与其他线程或进程沿二维数组的边进行通信，它的通信代价将随m线性地增长，而计算代价则将按m的2次方增长。此外，随着m值的增长，就会有更多的机会实施时延隐藏技术，如重叠通信与计算。

减少开销的需要提示我们对于具有许多处理器的并行计算机系统，很重要的一点是需要针对机器的专门特性进行优化。例如，仔细考虑开销与并行性这一小节中已概要介绍的并行累加私有和的概念。当处理器的数目增加时，在通信代价和计算代价之间需要进行协调：具有低扇出度的组合树会比较高，将导致较大的通信时延，而具有高扇出度的组合树将在树的每个结点上需要完成更多的计算量。仅当处理器数相当大时，这种协调才会变得比较重要。

较大问题规模会产生较好并行性能的概念隐含在半性能指标中，记为$n_{1/2}$，对于给定的程序和机器，它衡量为获得二分之一的效率时所需的输入大小。

3.6.4 问题规模的扩展

分析效率时假设当我们增加处理器数时问题的规模将保持不变。然而，当使用更多功能更强的处理器时，常常希望增加问题的规模。因此，在忽略存储器的约束并假设有理想加速度的情况下，考虑并行性是如何影响问题规模的。

对于一个运行时间为$O(n^2)$的顺序算法，我们有

$$T = cn^x$$

如果我们假定使用P个处理器时可使问题规模以m因数增加，对于相同的执行时间T，则有

$$T = \frac{c(mn)^x}{P} = cn^x$$

求解m，可得：

$$(mn)^x = Pn^x$$
$$m^x n^x = Pn^x$$
$$m = P^{(1/x)}$$

因此，对于渐近复杂性为$O(n^4)$的求解问题，如果将该问题的规模扩大100后，就需要100 000 000（一亿）个处理器！与之相比较，对于渐近复杂性为$O(n^2)$的求解问题，如果将该问题的规模扩大100后，我们仍需要10 000个处理器；但若渐近复杂性是线性的，则仅需要100个处理器。由此获得的教训是，基本算法应尽可能是可扩展的，而简单地增加更多的处理器数不会改变这一事实，反使问题会变得更麻烦。这一论点被称作适当潜力推论（corollary of modest potential）。

面对众核的挑战　目前的芯片多处理器含有的核数较少，但架构师们已在讨论有几十或几百个处理器的众核（many-core）机器。综合本节和在第2章中所学到的知识，可以看到众核的概念面临许多问题。首先，当核数增长时，核间的通信时延也将增长。由于时延的增长，我们就希望实现更大粒度的并行。因此，在众核芯片上，理想地希望执

行许多独立的进程，但是由于有限的RAM和芯片之间的总带宽，将使这一希望很难实现。另一方面，如果核在数目较少的并行任务上合作，可以看到效率将会成为问题，除非我们能够扩展问题的规模。不论是那一种情况，有效使用大量核的最好方法是使存储器具有很大的总带宽，但这受限于多核芯片的物理尺寸。

3.7 小结

在本章中已经看到，单纯增加并行性不会直接导致性能的增加，这里的增加性能是指减少时延或增加吞吐率。我们还看到并行性能的概念是很复杂的，因为存在许多性能的相互影响方面：存储器消费量，处理器利用率，同步，通信代价。因此，要改善性能我们通常需要在这些方面进行不同的协调，以达到我们增加局部性、减少跨线程相关性以及控制粒度的综合目标。最后，对于给定的并行性能复杂性，应意识到在评估性能和选择性能指标时存在许多细微的差别。

历史回顾

本章的主题涉及贯穿整个并行计算历史中所研究的主要部分，所介绍的词汇和概念实际上在每一篇文章中都会出现。阿姆达尔定律在阿姆达尔1967年的文章中提出；关于这一概念后来又有很多文献。$n_{1/2}$性能指标是由Roger W.Hockney于1977年在评估向量计算机性能时首先引入的；有关的精彩分析可参见Hockney和Jessoup [1988]的文章。适当潜力推论是由Snyder[1986]描述。

习题

1. 在事务存储器系统中，乐观地假设线程不会访问可能引起与其他线程冲突的共享数据。如果没有检测出访问违例事务就成功提交，否则事务就回滚。识别事务存储器系统中的各种性能损失源，按开销、竞争或闲置时间对它们进行分类。
2. 竞争应视为是开销的一种特殊情况吗？说明理由。
3. 闲置时间应视为是开销的一种特殊情况吗？在单线程程序中会有闲置时间吗？说明理由。
4. 阿姆达尔定律认为一个计算中的顺序代码的比例决定了它的并行加速比的潜力。那么超线性加速比可能性与该结论冲突吗？说明理由。
5. 请描述一种并行计算，要求它的加速比将不会随问题规模的增长而增加。
6. 假定一个并行计算有5%的开销，试计算使用128个处理器时的效率，假设当处理器数增加时开销保持不变。
7. 大多数并行密集矩阵乘法程序所使用的算法需完成n^3算术操作。使用求解问题规模扩展这一节中的推理，则为了使矩阵乘法的性能改进10倍需使用多少处理器？
8. 图3-3中说明了哪种相关性？
9. 在开销与并行性这一节中曾断言："…大多数程序有一个上限[针对每一种数据规模]，此后附加处理器的得益值将趋向于0…。"描述在此上限时这种计算的加速比曲线。
10. 在开销与并行性这一节中介绍了使用树来累加priv_counts以改进统计3的个数程序的性能。设计一个新程序进行统计3的个数和寻找最小和最大值，仍使用树累加方法。说明如何在新程序中使用批处理的概念以避免完成三个连续的基于树的累加。

第二部分 并行算法

由于第一部分已为我们打下良好的基础，现在来探讨创建高效并行算法的方法，这种算法的并行度能随着可用处理器数的增多而扩展。我们将展示一些成功的算法，并概要介绍针对不同计算场合提升并发性的原理。

为了确切地描述算法技术，我们将介绍一种编写并行程序的伪代码。伪代码是描述算法的简化和非正式记法，在算法的教科书中经常使用。伪代码在展示、讨论以及分析算法时不会偏向某一特定的程序设计语言，而且不会为不重要的细节而分心。用伪代码表示的算法对我们人类而言在概念上是一个完整的程序，虽然对计算机并非如此。然而，一旦理解了一个算法，用成熟的程序设计语言加以实现就非常直截了当，此时计算机就能理解该算法。在第二部分中，我们将努力使读者对并行化计算策略建立直观认识。并行程序的开发经常需要判断何时应开发并行性以及何时不予开发，因为此时如果硬要开发则可能导致过高的代价，而与可能获得的性能增益相比是得不偿失的。我们通过介绍有关数据分配、工作分配、数据结构设计以及算法的技术建立这种认识，而这些技术已经过充分的检验。这种知识是从事实际程序设计的理想资源，我们将在后面几章加以论述。

第4章 并行程序设计起步

要成为得力的并行程序员，我们必须学习表达诸如算法和数据结构等基本并行概念。我们还必须学习如何分析程序以确定它们的运行时间和存储器使用，我们在第11章中将对这一主题进行讨论。就此刻而言，也许最重要是要获得一种思考能力以使得所设计的算法能很好地匹配可用的语言和计算机。我们将从介绍两种并行计算的类型开始。然后将介绍描述并行算法的符号，并使用这种符号去研究构造并行计算的三种方法：无限并行性、固定并行性以及可扩展并行性。然后我们用一个按字母顺序排序的例子比较上述三种方法的有效性。

4.1 数据和任务并行

并行计算通常可被分为两种主要类型：数据并行和任务并行。这两个术语将指导我们考虑并行算法的设计。

4.1.1 定义

已经提出了许多有关数据和任务并行的定义，但我们采用以下的定义，因为它们能区别两种考虑并行化的方法，即我们是要并行化数据还是代码？

- *数据并行计算*是指同时对不同数据项完成相同操作的并行；并行量将随数据规模而增长。
- *任务并行计算*是指同时完成不同计算或任务的并行。由于任务数是固定的，因此它的并行性是不可扩展的。

与大多数分类学的结果类似，对计算的分类不总是清晰的。许多计算是混合式的，比如由一个固定的任务数复合而成，而其中某些任务可能是数据并行的。即使当计算不能很明确地归于两类中的哪一类，术语可以为我们提供了解其组成部分的手段。

4.1.2 数据和任务并行的说明

为了强调这两种并行类型的不同，让我们考虑准备一个宴会的工作。

数据并行方法将每一道膳食看成是一个并行单位，因此它使用P个厨师制作N道膳食，这样每个厨师要制作N/P道膳食。当N值增加时我们可以增加P，如果我们有足够的资源，如炉子、电冰箱、切板等。

任务并行方法认为膳食准备由许多任务组成，如准备开胃食品、色拉、主食和甜食。按这种方法，我们只需征召四个不同厨师每人负责其中一项任务。当然，通过细化任务还可开发附加的并行性。例如，色拉的准备可以进一步分为清洗、切割以及拼盘的子任务。显然，在这些子任务间存在相关性，因为在切割蔬菜前必须先清洗，而在拼盘色拉之前需先进行切割。其结果是将形成更大的并行量，因为一旦色拉的准备完成，洗、切和拼的操作就将同时进行。

色拉准备的例子说明了任务并行中的一种称为流水线的重要类型。在流水线中，一系列任务以顺序方式求解；在任一时间点，每个任务运行在问题的不同场合，而当一个任务完成

时，就将已完成的实例连续地传递到下一任务。在第3章的时延和吞吐率这一节中，我们看到处理器如何使用流水线执行来改善指令的吞吐率，而这一概念通常要应用到一系列操作上，且这些操作必须处理许多实例。

也存在组合数据和任务并行两者的混合求解。就我们的例子而言，可以先将宴会的准备分成若干任务，然后将数据并行实施到每一个任务上。例如，多个厨师可以为准备色拉切割蔬菜，从而在切割任务中形成数据并行。

4.2 Peril-L记号

顺序算法通常用某种形式的伪代码表示，因为它允许表示核心概念时无须说明每一个细节。对于并行算法我们有类似的期望，所以我们现在定义一种称为Peril-L的混合程序设计语言，它能用来开发和分析并行算法[⊖]。该语言要能达到以下的目标：

- 应尽可能的小，以方便学习
- 应足够通用以使它不会偏向任何一种语言、并行计算机或算法方法
- 应允许我们分析性能

其结果是，只要假想计算是在CTA模型上执行的，Peril-L对在第2章中出现的许多问题显示出中立。因此，Peril-L了解本地和全局存储器访问时间的差别λ。因为并行程序设计比顺序程序设计有更多的问题，我们将需要引入一些通常在顺序伪代码语言中并不使用的附加概念。

4.2.1 扩展C语言

为了减少需要学习的概念，Peril-L将扩展C程序设计语言的基本计算功能，我们选择C的理由是因为它比较稳定且为我们所熟悉。此外它也比较原始，允许进行位处理操作，在我们的算法中偶尔需要使用这种位操作，另一方面，C语言也足够高层从而在表达简单例子时非常简洁。

4.2.2 并行线程

Peril-L的计算世界开始于一个顺序线程。使用forall语句引入多线程，关于这一点我们已在第1章中提及。Forall语句具有如下形式，其中<*integer variable*>是有关一个整数变量的隐式声明：

```
forall(<integer variable> in(<index range specification>))
{
  <body>
}
```

该语句有如下的语义：对索引范围规范（index range specification）求值，形成一个索引集S。索引S是$|S|$个逻辑线程的名称。S中的每一个线程s执行<*body*>中的代码拷贝，使得在该线程中的<*integer variable*>值为s。当最后到达闭大括弧时各线程便终止。除了由forall语句指明并行外，总是单线程控制，所以在前面的语句未完成之前，不会开始下一个语句的执行。特别是，除非所有线程已经终止，否则就认为forall语句还未完成。注意，forall语句可以嵌套，这只要在另一个forall语句的<*body*>中放置一个forall语句即可。

作为forall语句的一个例子，以下的代码将按某种不确定的次序打印12行：

⊖ 选择该名称是为了与“parallel”同音，并没有任何危险的含义。

```
forall(index in(1..12))
{
  printf("Hello, World, from thread %i\n", index);
}
```

在这个例子中，来自不同线程的输出是无法预知的，因为所有线程虽然不是以锁步方式，但在逻辑上它们是同时执行的。的确，线程的执行可能是交叉的，或是某些线程已执行完毕而其他的线程可能还未开始。甚至这些语句按序的顺序执行也是可能的。其结果是，如果不采用某种方式的同步，就无法预测输出的结果会是怎样的。

4.2.3 同步和协同

如我们在前面Hello World例子中所看到的，在线程中排序操作有时很有用。在Peril-L中，由排它块（exclusive block）提供互斥。它的语法如下：

```
exclusive {<body>}
```

其语义保证每次只有一个线程能执行<*body*>。如果一个线程正在执行<*body*>，其他试图执行<*body*>的线程就必须等待。当一个线程完成执行<*body*>后，等待的进程以一个未指定的次序继续执行。我们可以用这一构造来改进Hello World例子，如下：

```
forall(index in(1..12))
{
  exclusive
  {
    printf("Hello, World, from thread %i\n", index);
  }
}
```

使用exclusive 块后，上述代码将完整地打印12行，但仍以未指定的次序出现。另一种方法可达到同样的效果，即将exclusive块放在printf()例程的实现中。

第2个工具是障栅同步，

```
barrier
```

它仅在forall内部起作用。barrier语句强使线程停止直至所有线程到达barrier，在此点后它们可以继续前行。例如，下面的代码使用barrier以保证所有的“tweedle dee”出现在任何的“tweedle dum”之前。[⊖]

```
forall(index in(1..12))
{
  printf("tweedle dee\n");
  barrier;                    等待最后一个tweedle dee
  printf("tweedle dum\n");
}
```

Peril-L还提供一种完成细粒度同步的方法，但在讨论这一机制之前，让我们先说明Peril-L的存储器模型。

4.2.4 存储器模型

为了避免偏向共享存储器或非共享存储器，Peril-L语言提供两个地址空间：一个全局地址空间和一个局部地址空间。在全局空间中的变量可为所有线程看到，而在局部空间中的变

⊖ tweedle dee和tweedle dum是指难于区分的两个人或两件事，俗称半斤八两。——译者注

量只能为一个线程所看到。在forall语句中声明的变量将为每个线程创建一个本地拷贝，而在forall语句外说明的变量是一个全局变量，可为所有由forall语句创建的线程所看到。

为了与CTA模型保持一致，假设全局变量具有时延λ，而局部变量可在单位时间完成访问。相应地，Peril-L使用以下的约定从视觉上区分全局变量和局部变量：

Peril-L命名约定 *全局变量使用下划线；局部变量不使用。*

为说明起见，考虑以下的一个程序，计算数组data中n个数值的绝对值：

```
int data[n];
forall(index in(0..n-1))
{
  if(data[index]<0)
  {
    data[index]=-data[index];
  }
}
```

请注意，Peril-L存储器模型是按逻辑线程定义的，因此如果有多个线程被分配到同一处理器，则它们的局部变量仍保持不同。

全局存储器的读和写 在Peril-L存储器模型中，多个线程能同时读相同的全局单元，完成所谓的*并发读*。然而，每次只能有一个线程改变全局单元的内容。如果两个线程写同一个单元，则存储在该单元的值定义为最后写入的值，但是由于其定时无法知道，因此最后的结果是未知的。当变量的值为未知时，分析程序的正确性通常就很困难，所以我们建议在没有诸如exclusive语句的保护机制时，多个线程不要同时写同一个全局变量。

在上面的例子中，线程同时读和写全局数据数组data，但是因为每个线程只访问对应于它的索引值的项，因此永远不会发生并发读和并发写。

注意，取决于所使用的硬件，使用全局存储器同时访问的性能影响可能超过标准的λ代价，因为在某些硬件中访问需顺序地进行。

连接全局和本地（局部）存储器 因为CTA模型没有全局存储器，所以全局的数据结构就分布在所有处理器的本地存储器中；全局寻址是借助将全局地址转换成指定处理器的恰当本地地址而实现的。

我们通常希望所设计的算法能运行在部分全局数据结构上，而这部分已被局部地分配给一个指定的线程，为此，需要一种方法以一个本地名引用已全局命名的数据。在Peril-L中，localize()函数将被用作此目的，我们可用它初始化本地数据结构。例如，

```
int alldata[n];                                    声明全局数据结构
forall (threadID in (0...P-1))                     派生线程
{
  int  size = n/ P;                                计算局部分配的大小
  int locData[size]= localize (allData []);        映射全局到该线程的局部
  ...
}
```

localize()函数返回一个对部分全局数据结构的访问，该部分已分配给执行该线程的处理器；此后，所有使用局部名对本地数据结构的修改等同于无λ代价的全局数据结构修改。

localize()函数很有用，因为它允许我们在创建算法时无须知道全局数据如何在处理器中分配的细节。同时，它还允许我们编写运行在本地数据上的算法。

关于全局存储器的本地化有三个重要的问题：

1. 含有本地化数据的数组使用本地索引；所以不管它们的全局索引值是什么，本地数组的第1项的索引值为0。

2. 如果有多个线程被分配到一个处理器，则localize()函数对每个线程分配全局数据结构；即每个线程只运行在分配给它的那部分全局数据结构上。

3. 不存在本地拷贝——全局和本地访问都指向同一存储单元。

根据第3点，通常比较明智的做法是不要对数据结构混合进行全局和本地的访问，除非已设置了某种类型的同步。例如，线程可以建立保护的全局访问，并通过调用localize()函数进行本地访问。这种预防有助于一致性存储器映像的实现。

拥有者计算规则　localize()函数能形成一种称为拥有者计算的数据并行程序设计风格。其思想是向进程分配的计算它所需要的全局数据结构部分正好是分配给它的那部分，即正是它所“拥有”的。这一规则为并行语言编译器所广泛使用，它将最大化本地数据的访问数。

另外两个例程也很有用。mySize(global, i)函数返回全局数组局部化部分中第i维的大小。因此，虽然上面的例子假设P正好能除尽n，计算size的一个更好方法应是：

```
size = mySize (allData[ ], 0)；局部分配的第1维的大小
```

最后，localToGlobal (locData, i, j)将返回相应于局部分配locData第j维中第i个索引的全局索引。

全局视图和局部视图　对localize()函数的使用并不意味程序设计语言应该采用全局还是局部索引存在价值判断。在第9章中，我们将定义全局视图语言和局部视图语言，且我们认为全局视图语言能提供局部视图语言不能提供的程序设计方便。不过由于Peril-L的目的是允许程序员为任何程序设计模型表示算法，需要提供对数据全局和局部视图的两种访问形式。此外，当Peril-L代码描述归约使用时，可能需要数据的全局视图，而在使用Peril-L描述归约操作实现时，就可能需要数据的局部视图。

4.2.5　同步存储器

在Peril-L中我们还可以看到称为full/empty variable（满/空变量）的另一种存储器，这是全局变量，支持细粒度同步。full/empty变量（或更简洁地，FE变量）的工作恰如其名。当它声明为empty（空）时，即不含任何内容。任何空的FE变量可以被赋值，此时就变为满。我们称一个满的FE变量为full（满）。任何满的变量可以被访问（读），此时就被清空，即返回到空状态。还有其他两种可能性，即试图对满的FE变量填入或试图访问一个空的FE变量，都将导致计算停顿直至变量相应地被清空或填满。表4-1总结了这些状态。考虑FE变量的一种方法是它们视信息如物品：为了将物品放到某处，必须先清空地方，当物品被取走时，它就会消失。

表4-1　full/empty变量的语义

FE变量的状态	变量访问（读）	变量赋值（写）
Empty（空）	停顿	填入值，变为满
Full（满）	取值，变为空	停顿

为了从语法上区别full/empty变量，我们在该变量名后加一个’。例如，可以如下声明一个

FE变量：

```
int t'= 0;   声明t'并填入0
```

因为它们可为所有线程看到，因此full/empty变量是全局的，所以它们将带来与标准全局存储器访问相同的代价。此外还有一些因同步机制所导致的附加开销，即使在线程不停顿时也为如此。

4.2.6 归约和扫描

请回忆我们在第1章中所介绍的全局求和（+-reduce）和并行前缀求和（+-scan）。为了通用化这些集合操作，在语言中我们允许归约和扫描作用于任何允许结合和交换的基本操作，包括+、*、&&、‖、max和min。在操作符和操作数之间，归约操作用斜杠(/)表示，而扫描操作用反斜杠(\)表示，如下：

```
+/count        Reduce（归约），即累加count中的所有元素
min\items      Scan（扫描），即寻找最小的items前缀
```

我们允许操作数值是一个完整的结构（struct），只要操作可作用于结构的各个组成元素。

归约和扫描的奇特记法 Peril-L使用斜杠和反斜杠来分割操作符和操作数，对大多数程序员来讲可能感觉不习惯。许多语言选择如plus-reduce（count）或min-scan（items）的记法。但是这种记法对于程序的例程函数调用来讲是不可区分的，模糊了高度抽象的重要性并降低了它们的作用。我们特意采用这两个斜杠——它们源自于APL语言——是为了强调归约和扫描的含义。

全局和局部操作数 归约和扫描能作用于全局值生成本地可用的结果，如下面所示：

```
least = min/dataArray;
```

它将生成一个标量，存储到每个线程的本地变量least中。但归约和扫描也能组合多个线程的值。操作数，即要组合的值，也可以是局部变量，且结果能被赋值到一个局部或一个全局变量。因此，在下面的语句中

```
total = +/count;
```

将组合每个线程中的局部变量count，且将生成的结果存入局部变量total。即每个线程接收该结果的一个拷贝。

将扫描作用到跨多个线程的局部变量时，如在下面的语句中

```
beforeMe = +\count;
```

count变量将被累加（以线程的索引值次序）；所以第i个线程的beforeMe变量将被赋值为前i个count值的和。

归约和扫描的同步 请注意，由于归约和扫描可以跨多个线程操作，当它们只对局部变量进行访问和赋值时就蕴含着同步。特别地，在以下语句中

```
largest = max/localTotal;
```

每个线程的localTotal值将组合在一起并被赋值给largest。所有线程必须到达此语句以完成求和，且没有线程能继续前行直至赋值结束。所以，将归约或扫描局部值的结果赋值给局部变量具有障栅同步的效果。如果该结果存入到一个全局变量，则一旦一个线程贡献其值和完成树操作的作用后，它就可以继续向前执行。

4.2.7 归约的抽象

最后，作为一个重要的程序设计指导方针，我们建议，当需要从多个线程处组合值时，应该使用归约操作而不要显式地编写程序进行计算。在Peril-L中很容易编写一个程序直接求全局累加，如，

```
exclusive {total += prov_count; }                    (1)
```

使用归约就更方便，如，

```
total = +/prov_count;                                (2)
```

但问题是不便于编程。更确切地说，其重要性是在于要说明所完成的全局累加是一个可扩展的代码。

为了分析这一问题，假设P = 10000。使用exclusive块（1）完成显式求和，就必须使进程顺序化；在执行中就没有并行，因此性能将显著受限。相反，归约（2）可用前面几章所述的树实现，或是可以使用如BlueGene/L（见第2章）机器上所提供的专门硬件。树的实现将实现并行化，并避免顺序化。用最简单的术语讲，它将操作从$O(P)$转换成$O(\log P)$。一个编译器能识别语句（1）是+-reduce的一个实例吗？也许可以，那么为什么不冒险试一下？明确地陈述这一点将有助于编译器和有关人员，同时又可改进最坏的性能限制。

4.3 统计3的个数程序实例

第1章提出了统计3的个数计算的几个版本。我们能用Peril-L对它们加以说明。例如，图4-1中显示了相应于尝试3的Peril-L程序。请注意，为了与原始的尝试3逻辑相匹配我们违反了总是使用归约完成全局求和的规则（仅此一次）。在后面的所有情况中我们将使用归约。

4.4 并行性的表示

应该如何对并行计算进行表达？在给出我们的回答之前，先考虑将导致不满意结果的两个显而易见的方法。

4.4.1 固定并行性

因为我们通常知道（至少开始时知道）将使用哪种计算机来运行程序（例如k个处理器的多核芯片），因此很自然用实际的并行量来描述求解方法，在这种情况下即是采用k路并行算法。虽然在程序设计时启发式方法也许能有助于我们考虑，但是固定一个特殊的并行度就使得所生成的程序不具有可扩展性。一个具有2k个处理器的改进多核芯片将无法改进性能。

简单的统计3的个数程序有一个固定并行性版本，如4个处理器。在图4-2中所示的代码基本上是图4-1中的解在t = 4时的情况。不论如何，好的程序设计实践会鼓励我们将该求解通用化成有t个线程，但是对于复杂的计算，一种更为基本的方法是在受限的版本中将处理器数嵌入到计算中。

需要注意的一个相关问题是，我们通常所感兴趣的计算机规模扩展常遵循某种数学特性的规则（如P是2的幂次方，P是一个完全平方，等等），并将这一假设嵌入到程序的并行性中。程序可以应用增加的并行性，所以采用这一假设没有什么坏处。但是，许多体系结构并不按此规则扩展，所以当将程序移植到那些机器上时将可能浪费显著的计算能力。一般而言，比较明智的做法是应尽量避免编写任何具有特定并发度的程序。

```
1  int array[length];                                全局数据
2  int t;                                            期望的线程数
3  int total=0;                                      计算结果，全部总和
4  int lengthPer=ceil(length/t);
5  forall(index in(0..t-1))
6  {
7    int priv_count=0;                               局部累加
8    int i, myBase=index*lengthPer;
9    for(i=myBase; i<min(myBase+lengthPer, length); i++)
10   {
11     if(array[i]==3)                               由于数组已被分割，无并发读
12     {
13       priv_count++;
14     }
15   }
16   exclusive { total+=priv_count; }                计算全部总和
17 }
```

图4-1 用Peril-L记法编写的统计3的个数计算（尝试3）

```
1  int array[length]; total;                         全局数据
2  int seg=ceil(length/4);
3  forall(j in(0..3))
4  {
5    int priv_count=0;                               局部累加
6    for(i=u*seg; i<min(length, j*(seg+1)); i++)
7    {
8      if(array[i]==3)                               检查局部段
9      {
10       priv_count++;
11     }
12   }
13   total =+/priv_count;                            计算全部总和
14 }
```

图4-2 统计3的个数的固定并行性求解（$t=4$）

4.4.2 无限并行性

对上述方法的一个自然改变是假定底层的硬件能提供无限的并行性，从而可以开发所有可能的并行性；当需要时，算法中的并发性能被聚集到顺序代码中，从而与硬件中的可用并行性相匹配。

按照这种观点，我们可按开发最大并行性方法来求解统计3的个数问题：

```
1 int count = 0;                                    由P0完成
2 forall(i in (0..n-1)
3 {                                                 由Pi完成
4   count = +/(array[i] = = 3? 1: 0);
5 }
```

这段代码很漂亮，测试各项并使用+-reduce累加与3匹配的元素数，但它有一些欺骗性。假定+-reduce使用树算法加以实现，则计算将需时$O(\lambda \log n)$，这是因为建立线程的开销是常量，相等测试的计算是常量，而组合的代价是对数的缘故。（我们在时间的分析中引入了λ，因为虽然我们认为它应是一个常数，但它会随P的增加而增长。）事实上，计算所需时间为$O(\lambda$

log$n$ + n/P）因为当P<<n时，一个平衡的分配将为每个进程分配n/P个元素，且它们必须顺序地进行测试。完成局部测试的计算（图4-2所示的固定并行性求解代码中的5-14行）在这里是不可见的，但当P = 4时将是需要的。上述这些考虑是相对的，因为实际上P永远不会等于n，至少对数据并行计算来讲是这样。

但是该问题实际上比简单地隐藏相关代价还要坏（毕竟我们能学习自己进行这种分析）。主要的问题是基于无限并行性的串行化程序非常昂贵。当P<n时，编译器有两个选择：一是模拟不存在的进程工作于代码上，这正是程序的语义实际含义；二是以某种方式生成可扩展代码以实现程序员的意图，这实际上就是创建图4-1中的7-16行代码。前一选择是低效的，而后一种选择则对编译器来讲非常难（虽然在我们的非常普通的统计3的个数例子中可能不是这样）。这种通用但又昂贵的求解在于它要模拟不存在的进程。

认识以下这一点是重要的，即并行程序设计的难点通常不是如何识别并行性，而是在于如何构造并行性以达到管理和减少线程间交互的目的，因为这些交互通常将导致性能的损失。

4.4.3 可扩展并行性

第3种方法是根据第2章的局部性原理，它表示并行求解的过程如下：首先，确定求解问题的组成部分（如数据结构、工作负载等等），是如何随计算规模n的增长而增加的；其次，我们设计一个重要（substantial）子问题集S，将大小为s的自然求解单位分配给每个子问题；最后，尽可能独立地求解这些子问题。

- 强调“重要”子问题旨在保证在一个线程中有足够多的局部工作，从而可平均分担诸如通信的并行开销，并说明分布求解问题数据的合理性，这正是我们的策略。与之相反，统计3的个数问题的无限并行性求解中的线程只有极少的工作量。
- 强调“自然”子问题所依据的事实是计算并不总是如统计3的个数问题那样可以平滑划分；例如，数组计算可能最好按整行或整列定义。因为子问题的大小决定了并行量，S = n /s，在执行时，将调整s，以使得S = P。
- 强调“独立”子问题所依据的事实是减少子问题间的交互可减少闲置时间和通信等。由于无限并行方法对全局count进行递增，因而也违反了这一依据。

这些定性的条件是相互关联的。例如，一个子问题小到何种程度而仍能有效地分担开销是有限制的，它意味着，对于一个给定的n，存在s的一个下限和P的一个上限。

由于问题并不总能平滑划分，因此在某些情况下，对于给定的n，创建一个有合适大小的s，使得n/s = S = P是很困难的。很显然，如果$S \geqslant P$，则多个子问题将被分配到一个进程上，这是通常的情况。如果S<P，则对于给定的求解问题只能使用较少的进程。

图4-3所示的统计3的个数程序是一个可扩展的版本。请注意它通过数据本地化和应用拥有者计算的规则来遵循局部性原理。由于该程序在线程之间没有交互直至计算结束处的归约，因此是非常高效的。

```
1  int array[length];                          全局数据
2  int t;                                      期望的线程数
3  int total;                                  计算结果，全部总和
4  forall(j in(0..t-1))
5  {
6    int size=mySize(array,0);                 计算全局数据的局部大小
7    int myData[size]=localize(array[]);       将全局数据中自己部分与局部
                                                变量关联在一起
8    int i, priv_count=0;                      局部累加
9    for(i=0; i<size; i++)
10   {
11     if(myData[i]==3)
12     {
13       priv_count++;
14     }
15   }
16   total =+/priv_count;                      计算全部总和
17 }
```

图4-3 统计3的个数问题的可扩展求解。请注意，数组段已被本地化

4.5 按字母顺序排序实例

统计3的个数的例子允许我们说明固定、无限和可扩展并行的三种求解概念。但是统计3的个数问题过于简单，因此各种方法的差别似乎很小。为了强调三种方法的差别并说明Peril-L的运用，让我们考虑对记录序列L中的某一字段x按字母顺序进行排序的求解。

按字母顺序排列是一种应用在少量数据（目录的文件名）和大量数据（数据库）中的常用排序计算。在我们的例子中，假设类型为rec（C语言结构）的记录以线性数组形式存放在并行计算机的全局存储器中；如果数据在磁盘上，仍可使用类似的技术，但在进行时间分析时，从磁盘到处理器的传送将占据主要部分，它比λ的开销要大得多。

表4-2中列出了按字母顺序排序程序中所使用的几个辅助函数。

现在我们描述由三种方法所生成的解答，然后对三种生成的算法进行比较。

表4-2 辅助函数

辅助函数	描 述
strcmp(str1, str2)	比较以空结尾的字符串str1和str2，返回一个小于、等于或大于0的值；可在string.h中找到
charAt(str, pos)	返回pos位置（初始为0）的字符；与JavaScript做法相同
LetRank(chr)	返回其参数字母的序号（初始为0）
AlphabetizeInPlace()	按字母顺序重新排序记录的参数数组

4.5.1 无限并行性

在无限并行性方法中，可以注意到我们可同时完成的最大比较数是将记录的一半与另一半进行比较。这种求解方法称为奇/偶交替排序（Odd/Even Interchange Sort）（参见图4-4）。在一连串的步骤中，并行地检查连续的奇/偶对值，如果它们不按次序就交换；接着进行类似的偶/奇对操作。这种交替对的测试（奇索引然后偶索引）顺序地执行直至不再有交换发生。

该程序使用两个全局变量L和continue；所有其他变量i、done、temp是线程的局部变量。

在每一个forall循环中，L的每一个元素只为一个线程所访问，因此在访问L时不存在竞态条件。当按字母顺序排序完成后用全局布尔变量continue来停止计算。特别是在第2个forall循环的结束处，使用一个&&-reduce组合所有的局部done变量，由它计算continue，来确定是否还有工作未做完。

虽然有许多的并发性，但也存在许多的副本。例如，如果一个表中的最小的元素处在最后位置，在排序时它就会移动经过文件中的每一个位置。

```
 1  bool continue=true;
 2  rec L[n];                                          全局数据
 3  while(continue) do
 4  {
 5    forall(i in(1:n-2:2))                            步长为2
 6    {
 7      rec temp;
 8      if(strcmp(L[i].x,L[i+1].x)>0)                  奇偶对错序?
 9      {
10        temp=L[i];                                   是，修正
11        L[i]=L[i+1];
12        L[i+1]=temp;
13      }
14    }
15    forall(i in(0:n-2:2))                            步长为2
16    {
17      rec temp;
18      bool done = true;                              为终止测试建立设置
19      if(strcmp(L[i].x,L[i+1].x)>0)                  奇偶对错序?
20      {
21        temp=L[i];                                   是，交换
22        L[i]=L[i+1];
23        L[i+1]=temp;
24        done=false;                                  还未完成
25      }
26      continue=!(&&/done);                           还有任何改变吗?
27    }
28  }
```

图4-4　用奇/偶交替方法按字母顺序根据字段x对一个记录表L排序

4.5.2 固定并行性

由于在英文字母表中只有26个字母，我们可以想象用26个线程进行并行求解，此时每个线程负责一个字母；我们称这种求解方法为*按字母顺序批处理*（alphabetic batch）。在将全局数据局部化后（即仅在指派给它的部分L数组上设置工作），每个线程统计以自己相应字母开始的记录数。对这些局部统计用+-reduce进行归约，就可获得全局的以每个字母开始的记录数，即批量。每个线程知道它的责任是对myLet大小进行批处理，分配足够的局部存储器，从全局数组中获取它的记录，并对它们进行局部排序。最后，对myLet进行+-scan操作，此时每个线程就知道在最后的安排中有多少个记录在它的字母之前，这样就能把它的成批记录返回原来的全局数组。显然，这个求解方法无法扩展超过26个线程，但每项排序所花费的只是2λ通信代价（从最初位置移到局部存储器，以及此后返回到它的最后位置）。图4-5给出了该求解的细节。

```
 1  rec L[n];                                          全局数据
 2  forall(index in(0..25))                            每个字母一个线程
 3  {
 4    int myAllo=mySize(L, 0);                         局部项数
 5    rec LocL[]=localize(L[]);                        使数据可在局部访问
 6    int counts[26]=0;                                统计每个字母数
 7    int i, j, startPt, myLet;
 8    for(i=0; i<myAllo; i++)                          首先，需要统计每个字母相应的词数
 9    {
10      counts[letRank(charAt(LocL[i].x,0))]++;
11    }
12    counts[index]=+/counts[index];                   计算出每个字母相应词数
13    myLet=counts[index];                             以本字母起首的记录数
14    rec Temp[myLet];                                 为记录分配局部存储器
15    j=0;                                             数组索引
16    for(i=0; i<n; i++)                               局部移动记录进行局部按字母顺序排序
17    {
18      if(index==letRank(charAt(L[i].x,0)))
19      {
20        Temp[j++]= L[i];                             局部保存记录
21      }
22    }
23    alphabetizeInPlace(Temp[]);                      对应这个字母进行局部字母顺序排序
24    startPt=+\myLet;                                 扫描统计应排序在本字母之前的记录数；进行
                                                       扫描同步，一旦完成排序就可重写L
25    j=startPt-myLet;                                 在全局数组中寻找该字母的起始索引值
26    for(i=0; i<count; i++)                           将记录返回到最初的全局存储器
27    {
28      L[j++]=Temp[i];
29    }
30  }
```

图4-5 固定26路并行求解按字母顺序排序。函数letRank(χ)将返回拉丁字母χ的字号（初始为0）

归约和扫描在这个求解中起着重要的作用。在开始时，每个线程确定以该线程负责的字母开始的、应局部存储的记录数；然后对counts[ ]进行+-reduce操作，这就确定了该线程应对全局负多少责任（见第12行）。此后线程获取它的记录并对这些记录进行局部排序。最后，+-scan操作（见第24行）将告诉每个线程在它的最后输出中将处在它之前的记录数，这就允许每个线程将它所负责排序的元素返回到原先的数组中。

请注意，已按字母顺序排序好的批处理记录不能马上返回到全局数组，仅在所有线程已从全局数组中移出了它们的批记录后才可返回。图4-5中的解没有对此进行显式的同步；更恰当地说，是完成+-scan操作所需的同步保证了所有的数据已被移出。

固定并行性的求解要优于无限并行性的求解，但需注意，它不能扩展。

4.5.3 可扩展并行性

第3个即最后一个求解，使用可扩展并行性方法生成一个Batcher双调谐排序算法，它既是可扩展的，又是我们所熟悉的顺序归并算法的并行版本。Batcher为硬连线的排序网络开发了这一算法，当然我们感兴趣的是它的并行版本。

初步想法 总的设想是每个线程（线程总数必须是2的幂次方）含有某些局部记录数，最初它们依据线程的索引值和计算的进展以递增或递减次序进行排序。然后，各个线程对用一个妥善的协议归并它们的序列，并生成一个更长的已排序的序列直至整个数组按字母顺序排序。在以下的讨论中将涉及图4-6，图中展示的是用$t = 8$个线程对$n = 24$个值的序列进行排序。

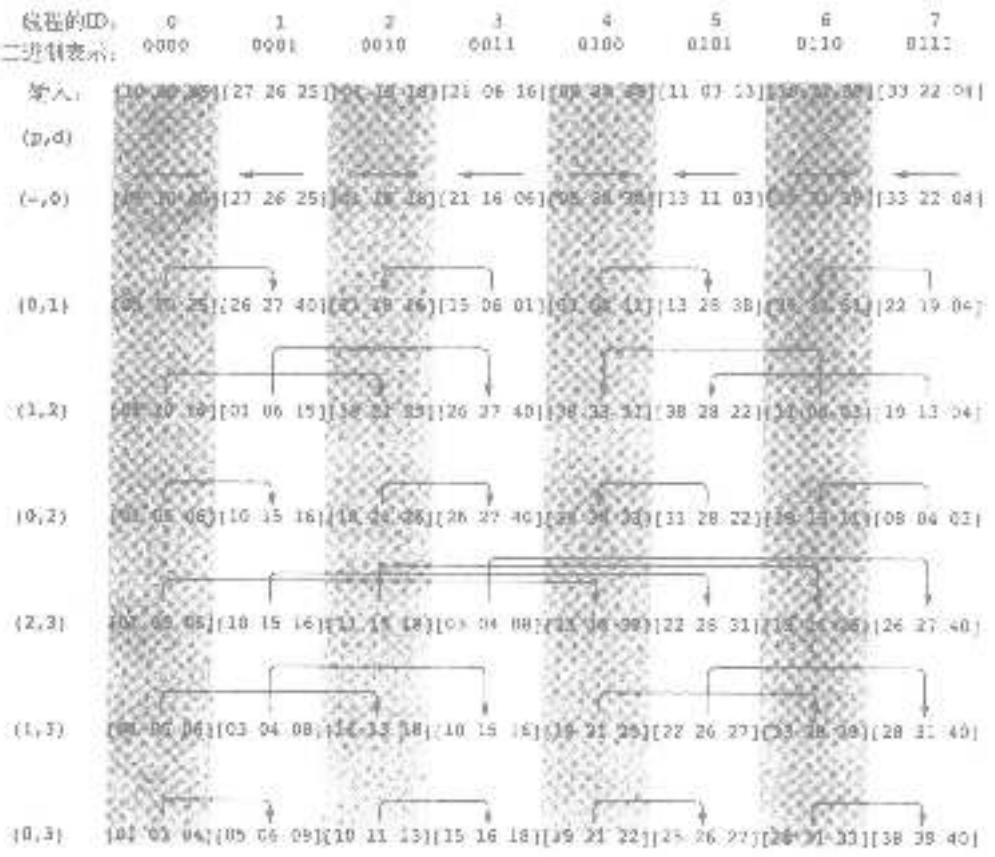


图4-6 用$P=8$个线程运行Batcher的排序算法排序$n=24$个数（一个线程的索引值以二进制表示为$b_2\,b_1\,b_0$）。由箭头指示增加次序的方向，即右箭头表示递增，步Step（-，0）是最初的局部排序；此后进程完成一个归并，被归并的线程仅在b_d位上不同，归并后产生一个递增（$b_d=0$）或递减（$b_d=1$）序列

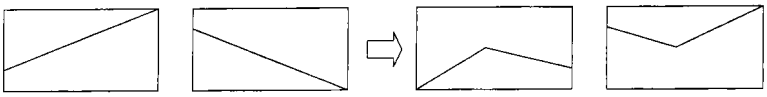
图中显示各个线程对的归并，按照由两个参数(p,d)所说明的协议进行，p指定哪些线程对将进行值的归并，而d指定排序的方向。

参数p受二进制表示的线程索引值支配。只有在二进位p位上不相同的那对线程才进行归并。（线程索引值的二进制表示$b_2\cdots b_0$示于图4-6的顶端。）例如，在第1阶段（0,1），线程0的二进制编码为0000，而线程1的二进制编码为0001，由于这两个线程在b_0位不同，因此它们将完成一个归并，类似地其他的相邻对也将进行归并。在下一阶段（1,2），线程0将与线程2归并。这是因为线程2的二进制表示为0010，它与线程0的00000表示在b_1位不同，依次类推。

排序的方向由参数d控制，即线程索引值的第d位b_d，当$b_d=0$排序就按递增方向，而若$b_d=1$排序就按递减方向。请注意，p永远不会与d相等；在一个阶段开始时，p的值与$d-1$相同，并在以后的子阶段不断递减，故两个在b_d位上不同的归并的线程需要与向b_d位给定的排序方向相一致。

一个阶段的第一个归并自归并两个已排序的序列开始，一个已排序的序列是单调递增或单调递减的。（从技术上讲，Batcher的排序算法同样适用于有重复值的排序，所以更恰当的术语应分别是非递减或非递增，为便于阅读，我们作了简化。）归并的结果将产生两个双调序列，即一个递增和一个递减的子序列，使得它们形如一个V或一个Λ。一个归并将比较相应的元素，将较小的值（那些靠近字母表前面的）移动到另一半中，我们称其为下双调谐部分

(lower bitonic half)。就形状而言，归并可用如下的图形加以描述：



很显然，此后如果归并这两个半双调谐（由其余的子阶段通过递归加以实现），就可完成对该组合序列的排序。虽然该算法有满意的递归性，我们的Peril-L解将着眼于快速直接的自底向上的方法。

Peril-L的解 开始时，借助localize()函数为每个线程分配一个记录序列。然后，每个线程使用key()函数将记录压缩成一个新的数据结构K，它仅包含每个记录的关键字段x和它的全局索引值：

```
Struct key(char[32] x, int home);
```

这一压缩避免了不需要的数据传送，加快了求解。(在固定和无限并行算法中也可使用压缩，但并没有什么价值)。每个线程在开始时对已压缩的文件进行局部按字母顺序排序（根据b_0位，进行递增或递减排序），此后以$p = 0$和$d = 1$进入算法的结构阶段。

Batcher并行排序的内循环是两个双调谐序列的归并，称为*双调谐归并*（bitonic merge），在该归并中，比较每个序列的相应位置并交换它们的位置如果较大的项处在低半部分；归并的结果产生另一个双调谐序列。双调谐序列的双调谐归并产生的还是双调谐序列，这是证明该算法确实能产生一个排序序列的基础[⊖]。为了实现递增和递减排序，存在有两个操作mergeUp()和mergeDown()。两者的差别在于是较小的线程索引值（向上）还是较大的线程索引值（向下）处在低半部分，因为术语递增或递减是相对于增加线程索引值而言的。在表4-3中我们可以看到一个线程$b_r \cdots b_0$的操作有四种可能。处理这四种的代码构成了该算法的内循环(参见图4-7中的第32～71行)。

表4-3 归并操作

操作	与其他线程对索引值的关系	归并时保留哪部分
向上归并	较小索引值，$b_r \cdots b_{p+1}0\ b_{p-1} \cdots b_0$	保留低半部分
	较大索引值，$b_r \cdots b_{p+1}1\ b_{p-1} \cdots\ b_0$	保留高半部分
向下归并	较小索引值，$b_r \cdots\ b_{p+1}0\ b_{p-1} \cdots\ b_0$	保留高半部分
	较大索引值，$b_r \cdots\ b_{p+1}1\ b_{p-1} \cdots\ b_0$	保留低半部分

算法的数据移动工作如下面所述。两个线程在一个子阶段开始时交换它们的数据，然后依据表4-3中所说明的规则，在内循环中局部地存储归并的低半或高半部分。虽然向上和向下归并都对相同的集进行比较，但该求解增加了计算的粒度，从而允许序列在单次传输中实现流水化。为传送数据服务的缓冲器，用一个全局变量BufK为每个索引值加以声明，数据将被存储在该变量中；这种本地化的目的在于加快访问速度。

需要使用同步机制来保证仅当缓冲器为自由时（即空闲的）才可存入数据。一个解决方法是声明BufK含有满/空变量，但只需单个满/空变量就足以涉及整个缓冲器。我们使用free'和ready'两个满/空变量来控制一对线程间的交互。当free'为满时（初始化时为如此），就可由配对的另一个线程对缓冲器进行填入；当ready'为满时，数据已被送往接收线程，从而可开始归并操作。

⊖ 参见Knuth的“计算机程序设计艺术”一书的第3卷。

```
1  int t;                                          线程数必须为2^m
2  int m=log2(t);                                  线程数的指数
3  rec L[n];                                       需按字母顺序排序的记录
4  int size=n/t;                                   局部部分；假设可分
5  key BufK[t][size];                              让关键字段通过的缓冲器
6  bool free'[t] = false; ready'[t];               管理缓冲器的满空变量
7  forall(index in(0..t-1))                        线程部分开始
8  {
9    int i, d, p; bool stall;
10   rec LocL[size]=localize(L[]);                 映射全局到局部
11   rec inputCopy[size];                          对值进行局部副本以简化同步
12   key Kn[size]=localize(BufK[]);                将全局缓冲器映射到局部以加快访问
13   key K[size];                                  工作序列数组
14   for(i=0; i<size; i++)                         压缩成仅含关键字段
15   {
16     K[i].x=LocL[i].x;                           保留字母串
17     K[i].home=localToGlobal(LocL,i,0);          记住全局索引值
18   }
19
20   alphabetizeInPlace(K[],bit(index,0));         局部排序，向上或向下排序方向由第0位决定
21   for(d=1; d<=m; d++)                           循环，m个阶段
22   {
23     for(p=d-1; p>=0; p--)                       为每个子阶段定义p
24     {
25       stall=free'[neigh(index,p)];              停顿直至能给出数据
26       for(i=0; i<size; i++)                     将数据发送给邻居（在该步内）
27       {
28         BufK[neigh(index,p)][i]=K[i];           发送给邻居
29       }
30       ready'[neigh(index,p)]=true;              释放邻居以进行计算
31       stall=ready'[index];                      停顿直至自己数据已可用
32       if(bit(index,d)==0)                       按哪个方向排序?
33       {
34         for(i=0; i<size; i++)                   向上归并：将早先数据移至有较低索引
35         {                                       值的线程
36           if(bit(index,p)==0)                   下线程对
37           {
38             if(strcmp(Kn[index][i].x, K[i].x)>0)
39             {
40               K[i]=Kn[i];
41             }
42           }
43           else                                  上线程对
44           {
45             if(strcmp(Kn[index][i].x, K[i].x)<0)
46             {
47                  K[i]=Kn[i];
48             }
49           }
50         }
51       }
52       else
53       {
54         for(i=0; i<size; i++)                   向下归并：将早先的数据移至有较高
55         {                                       索引值的线程
56           if(bit(index,p)==1)                   下线程对
57           {
58             if(strcmp(Kn[index][i].x, K[i].x)>0)
59             {
```

图4-7　使用Batcher排序算法按字母顺序排序L中记录的Peril-L程序

```
60                K[i] = Kn[i];
61              }
62            }
63            else                                    上线程对
64            {
65              if(strcmp(Kn[index][i].x, K[i].x)<0)
66              {
67                K[i]=Kn[i];
68              }
69            }
70          }
71        }
72        alphabetizeInPlace(K[],bit(index,p));    局部排序，向上/向下方向由第p位决定
73        free'[index]=true;                       完成写缓冲区，启用
74      }                                          子阶段循环结束
75    }                                            阶段循环结束
76    for(i=0; i<size; i++)                        获取属于该线程的记录
77    {
78      inputCopy[i]=L[K[i].home];                 获取记录并局部保存
79    }
80    barrier;                                     等待直至每一个均已完成
81    for(i=0; i<size; i++)1                       使输出变为可用
82    {
83      LocL[i]=inputCopy[i];                      使记录变为全局可用
84    }
85  }
```

图4-7 （续）

所采用同步的策略为如下：一个线程将一直处于停顿状态直至它的相邻线程的控制变量free'变为满；此时它将数据放入它的相邻线程的缓冲器中（BufK）并通过填满ready'告诉相邻线程继续向前执行；此后该线程就处于停顿状态直至它自己的缓冲器被它的相邻线程所装载，并被告知继续向前执行；在结束时，它将free'变量再次填满，表示它已为下次的子阶段作好了准备。

概括来讲，(*p*, *d*)策略控制该算法的整个活动，它主要由合并操作所组成，在一对线程之间用两个同步变量来管理它们的交互，或是为邻居设置（ready'）或是为它们自己设置（free'）。

`4.6 三种求解方法的比较

三种求解方法相当不同，因此不可能将一种方法视为是另一种的变化。固定并行和无限并行的求解与可扩展求解相比也许更容易创建。而可扩展求解在利用Peril-L的特性方面要完整得多，这就意味着它的并行结构更为复杂。那么，究竟哪一种方法最为有效？

“有效”有许多含义。我们可以观察到，按照任何的定义奇/偶求解总是低效的，因为它将数据从一个位置移动到它的邻近位置，用呆板的方法排序。按字母顺序的固定解具有最少的数据移动（任何记录最多2次），但它过度地使用全局通信来计算字母的位置。这就是说，为了识别记录“属于”哪一个指定的字母，每个线程要读所有的数据，这就需要将x字段从记录存储的场所到每一个进程移动两次。因此，虽然最小化了数据移动，但对数据的访问并非如此。可扩展求解需要大量移动数据的描述符，但不像无限求解，它以流的形式移动，因而可以流水化。可扩展求解也不像固定求解，它移动它们到固定可预测的目的地。实际的记录只移动了一次，即最后将它们按序排放时。另外，除了将数据传递给配对的另一个线程时，可

扩展求解的所有计算都是局部的。

但是正如所述，这些求解的主要差别，是它们所涉及进程数的通用性。当$n>P$时，无限求解必须通过多线程模拟不存在的进程。固定求解不能使用多于26个的进程，虽然在$P<26$时很容易使用简单的多线程技术使多个字母挤在一个进程中。可扩展求解适合于任何进程数直至$P = n$，虽然P必须是2的幂次方，如若不是，就将浪费进程。为可扩展求解而做的程序设计努力是值得的，因为它不但是高效的，而且适用于很广的应用环境。

4.7 小结

在本章中，我们介绍了描述并行算法的记法Peril-L。使用该记法，我们描述了三种不同的表示并行算法的方法。

我们认为固定并行过于限制，不能适应处理器数增加的情况。我们给出了无限并行方法生成的解，它只能直接应用在非实际的$n = P$的情况，且在$n>P$时将导致显著的复杂性。最后，我们认为可扩展并行求解是最好的，因为它能使用许多处理器而同时又限制了通信和线程的交互。在第5章中我们将通过引入支持可扩展并行的抽象继续进行这一讨论。

历史回顾

为描述算法开发的混杂语言，在计算机科学中有很长的历史，这可追溯到Knuth在他的《基础算法》一书中所发明的MIX[1997]。在多个独立处理器的局部存储器上应用覆盖全局地址的概念，自Cray T3E问世以来已在计算机体系结构中经常被使用；对覆盖概念的各种软件变化的描述最早出现在分区的全局地址空间（PGAS）语言中。在Iverson的《A Programming Language》[1962]中首次介绍了归约和扫描（本书中所使用的形式）。HEP计算机首次使用具有满/空标志位的存储器，Smith对此作了描述[1978]。Knuth的“计算机程序设计艺术”一书的第3卷“排序和搜索”[1998]对Batcher的排序能产生一个有序顺序给出了证明。

习题

1. 快速排序能提供数据并行或任务并行吗？请解释理由。
2. 下棋程序能提供数据并行或任务并行吗？请解释理由。
3. 重新设计奇/偶交替排序程序使它能以少于$P = n$的处理器数工作。即每个处理器负责$s = n / P$项。考虑以下两种情况：
   a. s个项是L的连续元素。
   b. 由$[0\cdots n-1]$中处理器index负责的s个元素处在index, index+p, index+2P, …位置上。
4. 用习题3（b）的求解方法，并假设每次交换（奇/偶或偶/奇）所需的通信可在λ时间中完成，评估在1024个处理器上对1,000,000个记录进行排序所需的执行时间（以λ表示）。
5. 使用习题4中的问题规模，评估按字母顺序容器求解所需的执行时间。
6. 计算在2^5个处理器上对2^{20}个输入进行Batcher排序需要多少个子阶段。
7. 针对习题6中的条件，并假设在L中的每个记录的长度为256个字节，计算每个子阶段通过构建k可节省多少数据传送。
8. 给出在Batcher排序的前两个阶段中，free'[0]、free'[1]、ready'[0]和ready'[1]的值序列。
9. 解释Batcher排序中inputCopy的作用。

10. 红/蓝计算仿真两个交互流：在一个$n \times n$个格的板上，初始化后每个格子将具有红、白、蓝三种颜色之一，其中白色表示为空，红色表示向右移动，兰色表示向下移动。当颜色到达边线时就环绕到对边。在一个交互的前半步，任何红色可向右移动一格，如果它的右边格未被占用；在一个交互的后半步，任何蓝色可向下移动一格，如果它的下边格未被占用；允许在前半步红色空出的格在后半步由蓝色移入。当用$t \times t$块瓷砖（t能整除n）覆盖该板时观察该板，如果任何瓷砖的带色区域多出一种颜色的c%时就终止计算。用Peril-L为红/蓝计算编写一个求解程序。

第5章　可扩展算法技术

了解了描述并行算法所需的记法之后，现在开始讨论生成可扩展并行程序的方法。在这里，我们关注于当进程数P增加时，仍能获得良好的并行效率。通常而言，效率可以通过增大问题的规模来增加。因此本章关注于数据并行计算，因为任务并行通常无法随进程数P的增加而显著地扩展。当然，由于许多任务并行的计算能在单个任务中使用数据并行，因此本章中所讨论的这些思路是可以广泛采用的。

本章首先简要讨论一个理想的数据并行计算。这是一个能以因特网规模执行的计算，因为它的结构拥有很多大的独立计算块。然后我们应用这种结构来产生能在多个进程间组合值的可扩展技术。我们还将描述基于相同结构的归约和扫描（这两种可扩展抽象有许多用途）的通用实现。本章最后，将可扩展并行的目标与在进程间分配数据的问题联系起来，首先考虑静态分配，然后考虑动态分配。

5.1　独立计算块

理想的并行计算是由很多大的独立计算块所组成，且这些块之间没有交互。这种计算虽然很少见，但的确存在，而且能在因特网上利用空闲的PC计算资源加以解决。美国加州大学Berkeley分校的BOINC项目就整理了这类项目的一份清单。通常，独立计算任务被下载到参与者空闲的PC上进行计算，然后将结果返回给服务器，由服务器聚合这些结果。这类问题可能很理想，但并不典型。相反，几乎所有的并行计算都需要线程交互，而交互的数量会影响到我们获得良好性能的难易程度。

另一些并行计算虽然更复杂，但只要它们能使用独立计算块的策略，就仍能从中获益。我们在第4章中曾提及的统计3的个数解决方案就使用了此种方式。如图4-1中所示，各个线程负责lengthPer项的块。该解决方案是应用了以下重要原则的一个例子：

当并行程序关注于计算块时就更具可扩展性（通常而言块越大越好），这将最小化线程间的相关性。

使用大计算块是很重要的，因为如果我们假设有一个固定的问题规模，则当进程数增加时，每个进程将在更小的块上操作。如同我们曾在第3章中提到的，通常有效块的大小有一个下限，但在达到该下限点之前，持续的扩展是有可能的。这种下限点必须在程序设计过程后期的性能调谐阶段，或是当将程序移植到新硬件上时确定（参见第11章）。

5.2　Schwartz算法

从美国纽约大学计算机科学家Jacob“Jack”Schwartz为树操作提出的算法中，可以看得到我们指导性的原则，例如+-reduce（加归约）。他主要的观察是，树结构应该用来连接进程，而不是所有的项。比如对于固定的进程数P和值的数量n，其中$P<n$，有两种方式可用来计算+-reduce：（1）引入逻辑线程实现一棵组合树，它精确地编码实现了图1-3中具有$n/2$逻辑并发性的解决方案；或者（2）使每个进程在本地对n/P个项进行求和，然后在与所有进程相连且

有P个叶结点的树中，对P个中间结果求和。Schwartz认为第2种方案更佳，因为在一个紧凑的循环中求和比在多个进程中要快。从本质上来讲，Schwartz算法的解决方案使所有进程都直接参与问题的求解，但实际上这种做法并没有使性能有任何的改进。Schwartz算法印证了第4章中得出的观点：即无限制的并行方法比可扩展的并行方法要差。虽然第1种解决方案提供了更好的逻辑并发性，但实际的性能却永远也不会超过P倍，而且额外的逻辑并发性通常会导致额外的工作，而这种工作在第2种（本地）方案中是不需要的。(Schwartz还提供了一个更详细的分析，但这个观点已经很清晰。)

图5-1给出了Schwartz算法的示意图。我们看到所有进程先进行本地操作，然后使用树结构组合它们的结果。在树的组合阶段，并行性是不平衡的。因为进程0会参与树的每一层组合，而其他进程仅在发送它们的值到相邻进程时才会参与。

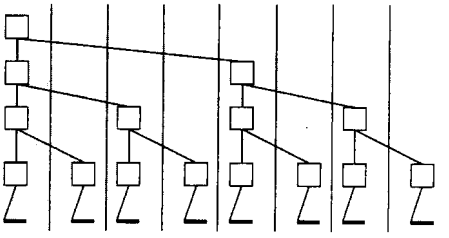


图5-1 进程归纳树。各进程先本地计算一系列的值（用粗线表示），然后成对地组合结果，归纳成树；注意进程0在树的各个层次均有参与

Schwartz算法的组合树能利用我们的Peril-L混合语言很好地描述。如图5-2中所示，该程序依赖于满/空变量的使用。请回忆在表4-1中所描述的这种变量的语义。它们一开始是空的，

```
 1  int nodeval'[P];                          全局满/空变量，用来保存来自右
 2                                            子结点的值
 3  forall(index in(0..P-1))
 4  {
 5    int tally;
 6    stride=1;
 7    ...                                     在此处计算tally（计数器）
 8
 9    while(stride<P)                         开始树的逻辑
10    {
11      if(index%(2*stride)==0)
12      {
13        tally=tally+nodeval'[index+stride];
14        stride=2*stride;
15      }
16      else
17      {
18        nodeval'[index]=tally;              开始时发送到树结点
19        break;                              如果不再是一个父结点，则退出
20      }
21    }
22
23  }
```

图5-2 归纳图5-1中树的Schwartz算法。第8行将局部计算的值加载到树中；第14行在两个操作数都可用时，执行求和。线程会在无事可做时退出

对空单元的访问将导致停顿；一个赋值将填满一个空单元，对一个填满单元的访问将取走该

项；最后，对已填满的单元进行赋值将导致停顿，直至占用该单元的项被取走。由满/空变量提供的同步将允许代码以一种自适应的巧妙方式运行。当一个线程发现它的高度为偶数时，它将组合它的2个子结点。左边子结点与线程有相同的索引值（index），而右边子结点的索引值与其差Stride个单元（index+stride）。当这些操作数被访问时，它们的存储器单元将依照满/空变量的语义清空，这就允许将结果值存回到父结点中。每次while循环的迭代之后，就会有一半线程结束。

与Schwartz的看法一致，我们认为归约、扫描和其他基于树的算法是局部操作和有P个叶结点的全局组合树这两者的组合。在程序设计中使用这种操作时，如

```
total = +/ data;
```

我们会期望编译器使用Schwartz的局部/全局方法生成代码。

具有更高扇出度的树　在我们的例子中，已经使用了二叉树，但二元性不应成为唯一的真理。具有更高扇出度的树提供了一种折中：更高的扇出度将导致一棵更浅的树，这能减少从根结点到叶结点的时延，但同时也会减少可用并行性的数量，且当多个子结点都试图与同一父结点进行通信时，将增加竞争。

5.3　归约和扫描抽象

我们已经讨论了树通信结构的价值。在第1章中，先使用树结构从数组值的相加中去除了顺序性，之后我们使用Schwartz风格的树在统计3的个数程序的一个版本中消除了竞争，否则将会有许多进程竞争共享单元。树结构也可用来描述归约和扫描操作，这两个操作可用诸如+、*、and、or、min、max等操作加以定义。在本节中，我们将力图表明这些抽象远比这种有限操作集所建议的功能要强大得多，此外，我们所开发的通用归约和扫描函数，在定制后可完成许许多多的操作。

为什么归约和扫描是重要的抽象？下面来逐一观察：

- **归约**，组合一组值而产生单个值，这几乎总会要用到，因为在并行计算中，在某些时刻，总会要比较或组合由不同线程产生的结果。这或是用来汇总计算，或是用来控制它的执行。
- **扫描**，并行前缀计算。它包含了执行分离的串行操作的逻辑，以及产生一系列中间结果计算的逻辑。循环的迭代通常看上去是串行的，因为当它们迭代时，它们会按序累加信息。但如果更仔细观察，就会发现它们经常可使用扫描来解决，这就可以使用更多的并行性。

我们推荐将归约和扫描的使用扩展到尽可能大的程度。即使当程序设计语言中没有这两个操作时，它们仍应当被抽象成函数，而不是硬编码到程序中。这是因为，首先，它们是高层的，它们的使用传递了有关程序逻辑的宝贵信息。其次，使用这种抽象允许为目标机进行定制的实现。一些并行机甚至为执行全局求和提供了硬件支持，例如在第2章中曾提及的IBM公司的BlueGene/L，它就有一个如图2-10b中所示的组合网络。最后，它们通常有许多机会可进行优化。例如，为了计算包含n个欧几里得点的数组Pt的边界框（bounding box），其中数组中的每个元素都包含了x坐标和y坐标的字段，可以编写如下：

```
topEdge = max/Pt.y;        寻找Pt数组中y字段中的最大值
botEdge = min/Pt.y;        寻找Pt数组中y字段中的最小值
rightEdge = max/Pt.x;      寻找Pt数组中x字段中的最大值
leftEdge = min /Pt.x;      寻找Pt数组中x字段中的最小值
```

这段代码可以被组合成（通过编译器生成或由程序员手动）跨数据的单次传送。这种优

化在查看具体的实现细节时并不显而易见。

为了最大化地使用这些很有用的思想，支持通用的归约和扫描是至关重要的，而这正是下一小节的主题。

两种扫描 必须清楚，扫描能依据在计算i个元素总和时，是否包括第i个元素本身加以区分，包括的称为内含（inclusive）扫描，不包含的称为排除（exclusive）扫描。例如，对于序列$A = \{2, 4, 6\}$，内含扫描产生的结果：$+\backslash_{in}\ A = \{2, 6, 12\}$，而排除扫描产生的结果：$+\backslash_{ex}\ A = \{0, 2, 6\}$，对于排除扫描，第1项通常是函数的本身，或是某个占位符（place-holder）。由于存在如下恒等式：

$A<op>(<op>\backslash_{ex}\ A)=<op>\backslash_{in}A$

这使得各个实现之间略有不同，Peril-L就使用了更为通用的内含扫描。

5.3.1 通用归约和扫描举例

为了了解归约和扫描的强大功能，考虑以下问题及对它们解决方案的描述。此处，我们以串行的方式描述每个解决方案，但实际上每个方案都可以使用定制的归约操作并行地解决：

- **次小的数组元素** 在数组中识别出次小的元素可能很有用。例如在一个包含0的数组中识别出最小正整数，则解决方案为保存2个变量，smallest（最小）和next-smallest（次小），每个都初始化为正无穷。用它们的值与数组中的每个值进行比较：如果数组的值比smallest的值要小，则smallest和next-smallest都要因此而更新；否则，如果数组的值只比next-smallest的值小，则只更新next-smallest的值。
- **直方图**（K路） 给定一个值数组a，计算间隔k的直方图，则解决方案将假设min归约和max归约可用来计算此范围内的最小值和最大值。一旦知道这些值后，我们会初始化有k个元素的数组hist为0。然后我们在数组a中进行迭代，计算每个元素归属的区间，并递增hist数组中相应的元素。
- **最长的全1子串的长度** 给定一个值数组v，计算最长的全为1的子串。该问题可以通过使用current和longest这两个变量求解。两者初始时皆赋值为0，current变量将保存当前全为1的长度，而longest变量将保存迄今为止最长的全1的长度。然后我们在数组v中进行迭代：如果v的值为1，我们将递增current的值，如果v的值不为1，我们会将longest的值设为max(current, longest)，然后重置current的值为0。当迭代完成时，答案将会是：max(current, longest)。
- **首次出现x的索引** 给定一个符号数组s，返回首次出现x的位置，则解决方案是初始化一个2元素的数组temp为x和正无穷，然后在这个数据结构上迭代寻找x，找到之后，保留已保存的索引与所找到的索引这两者中的较小者。

虽然上述这些计算都能足够容易地用Peril-L或其他语言实现，但它们最终都需要组合中间结果，可能还需要一个常规的归约。为整个计算调用一个更抽象的解决方案可能更为明智。

注意到上述所有例子中，这些解决方案都使用了一定容量的额外存储器来保存累加的中间结果。例如，在次小的数组元素的计算中，我们保存了最小的和次小的值。在计算直方图时，我们保存了数组hist的k值；在识别最长全为1的子串中，我们保存了current长度和longest长度。当我们开发自己的通用归约解决方案时，会考虑一个通用的做法，将这些中间值放置于一个称为tally（计数器）的小数组中。

与归约抽象相同，扫描抽象也具有较强的功能，因为它能并行计算那些看起来只能串行计算的问题。考虑以下例子，与之前一样，我们以串行的方式阐述问题，但以并行的方式进行求解。

- **参赛队的排名**　给定一个数组results，它保存了k个队伍间所进行的n场系列比赛的结果，所产生的结果是在各轮比赛中获胜场次最多的队伍ID（如果有多个队伍获得了相同的获胜场次而导致平局，则输出的结果为0）。假设数组results的第i项保存了赢得第i场比赛的队伍ID，则解决方案是先维护一个有k个元素的数组times-won，且初始化为0，然后在数组results中进行迭代，在数组times-won中增量获胜者的项，并输出超过该点的队伍数。
- **保存最长的全1子串**　给定一个二进制数组，除了最长全为1的子串之外，删除其余所有的1。与用归约找到最长的全1子串的长度类似，该解决方案不仅保存了每个全为1子串的索引位置，而且也保留了迄今为止所能看到的最长全1子串的长度。在获得全局的答案之后，该解决方案迭代地在数组上逆序寻找全1子串，并将不属于最长全1子串的其余任意全1子串置为0。
- **最后1次出现的索引**　给定一个其元素值选自$(1\cdots k)$的数组，输出与索引i处所存储的值相同的最近1次的索引；如果是第1次出现则输出0。

还有许多其他例子，但这些例子已经充分说明了这一思想。与归约操作相同，这里的每个扫描操作都需要有一定容量的、称为tally的额外存储空间来保存中间结果。

5.3.2　基本结构

有一种基本结构对所有的归约和扫描操作都能通用。由于我们在寻找一种可扩展的解决方案，我们就假设归约或扫描所操作的数据结构已被分割成片，分配到未知数量的进程上。由于我们采用全局方式组合数据，因此将使用类Schwartz算法，即先进行本地计算，然后使用树来完成计算。我们还假设每个进程中已存在了一个称为tally的本地变量，它相当于在之前例子中所提到的额外存储器，可用来存储归约和扫描操作的中间结果。基于这些假设，所有的归约和扫描操作都可以通过定义下列函数的变体来实现：

- init()函数初始化为本地计算所准备的计数器tally。
- accum()函数对一个操作数元素执行本地累加，并将结果存到tally。
- combine()函数组合从它两棵子树中获得的tally中间结果，并将组合后的结果传递给其父结点。
- x-gen()函数获得全局结果，并生成最终答案。对于归约和扫描，会有各自不同形式的x-gen()。

作为示例，我们来定义能实现标准+-reduce（加归约）的函数，即+/ A。

- init()函数创建了一个整数的临时空间tally，并初始化为0。
- accum(tally,val)函数将val的值（即对应于某个i的A[i]的值）与tally中的值相加，并将结果赋值给tally。
- combine(left,right)函数将左右子结点的tally值相加，并将结果传递给父结点。
- reduce-gen(root)函数对于该简单操作而言是一个空操作，它直接返回它的参量作为全局结果。

在本地累加操作的末尾，每个进程都发送它的值到父结点。图5-3表现了这个逻辑，图5-4给出了实现该逻辑的Peril-L代码。

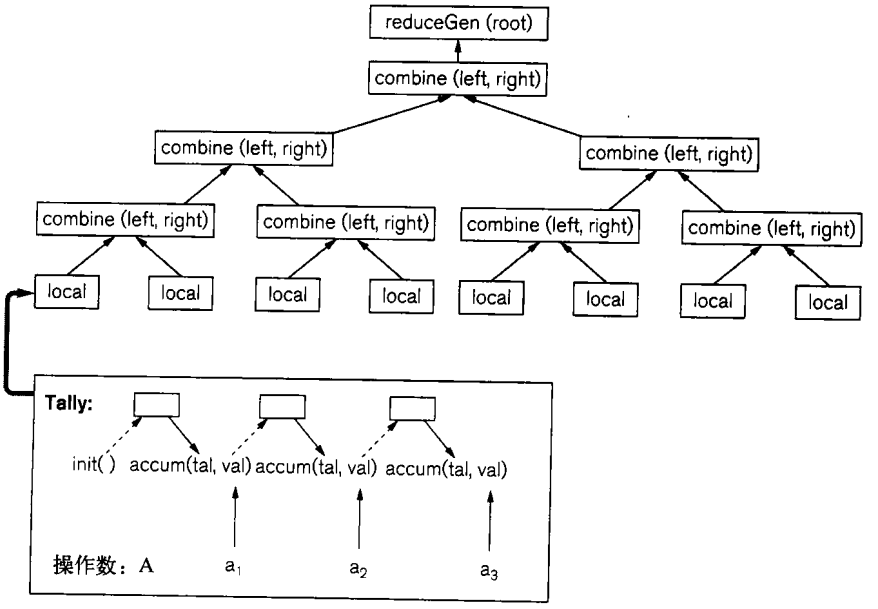


图5-3 实现+/ A的通用归约逻辑示意图。底部的大方框概述了本地计算，即以init()开始，并将accum()应用到每个A值；当本地累加完成后，tally被传递给父结点；树结点将combine()应用到来自子结点的tally值；最终根结点使用reduceGen()抽取出结果

```
 1  int nodeval'[P];                                    全局的满/空变量
 2  int result;
 3  forall(index in(0..P-1))
 4  {
 5    int myData[size]=localize(dataarray[]);           全局数据值的局部部分
 6    int tally;
 7    int stride=1;
 8    tally=init ()                                     初始化tally
 9    for(i=0; i<size; i++)
10    {
11      tally=accum (tally, myData[i]);                 局部累加
12    }
13    nodeval'[index]=tally;                            开始时发送到父结点
14    while(stride < P)                                 开始树的逻辑
15    {
16      if(index%(2*stride)==0)
17      {                                               全局组合值
18        nodeval'[index]=combine(nodeval'[index],
19                                nodeval'[index+stride]);
20        stride=2*stride;
21      }
22      else
23      {
24        break;
25      }
26    }
27    if(index==0)
28    {
29      result=reduceGen (nodeval'[0]);                 生成归约值
30    }
31  }
```

图5-4 通用归约逻辑的Peril-L代码。注意那4个构成函数所在的位置。树的组合依赖于满/空存储器的使用，它驱动了树的累加。线程在完成它们在组合树中的角色后将会终止

5.3.3 通用归约结构

对于简单操作+/ A而言，上述4部分的说明过于通用，但该结构即使在更复杂（同时功能也更强大）的实例中也是最基本的。我们的目标是就要以此种更抽象的方式考虑归约。为了说明这个问题，我们给出了代码以表明secondMin/ A（次小的值）是如何进行计算的。图5-5给出了tally项的声明，以及实现该归约的4个函数。

显而易见，图5-5中的4个函数只进行了少量修改，就替代了图5-4中对应的4个函数，产生了高度并行、可扩展的secondMin/ A实现。Schwartz模板提供了这个结构，我们只需要在处理过程的不同阶段确定要执行哪些操作即可。

```
 1  struct tally
 2  {
 3    float smallest1;                          最小元素
 4    float smallest2;                          次小元素
 5  };
 6
 7  tally init()                                初始化tally
 8  {
 9    tally t;
10    t.smallest1=MAX_FLOAT;
11    t.smallest2=MAX_FLOAT;
12    return t;
13  }
14
15  tally accum(tally t, float elem             局部累加
16  {
17    if(t.smallest1=elem)                      这是新的最小元素吗?
18    {
19      t.smallest2=t.smallest1;
20      t.smallest1=elem;
21    }
22    else
23    {
24      if(t.smallest2>elem)                    这是新的次小元素吗?
25      {
26        t.smallest2=elem;
27      }
28  }
29      return t;
30  }
31
32  tally combine(tally left, tally right)      通过累加右边的值，组
33  {                                           合到“左边”
34    tally t;
35    t=accum(left, right.smallest1);
36    t=accum(t, right.smallest2);
37    return t;
38  }
39
40  float reduceGen(tally t)
41  {
42    return t.smallest2;
43  }
```

图5-5　实现secondMin归约的四个通用归约函数。tally是个双元素的结构

5.3.4 通用扫描组件举例

通用扫描应用了与通用归约相同的概念。其主要区别在于组合结束后，中间结果必须要沿组合树向下回传。也即是说，为了完成本地值上的前缀计算，每个进程都需要从组合树中获得中间值。(参见第1章中关于并行前缀的讨论，尤其要注意图1-4)。

通用扫描一开始很像通用归约，而且这两个算法的init()、accum()和combine()函数在概念上是完全相同的。但是为了将中间结果传递给所有进程，tally值使用以下的约束方式沿着树向下传递：

每个进程从父结点处所接收到的值，即是父结点最左边的叶结点所遗留的tally值。

由于我们在块上进行计算，所以从父结点处所接收到的tally值，即是最左边叶结点块上第1项内所遗留的组合项值。由于根结点没有父结点，所以init()必须要稍作修改，创建一个值作为输入，以作为根结点逻辑上的父结点。每个结点从其父结点处接收值，并将该值传递给其左边子结点。对于它右边的子结点，它组合来自左边子结点向上扫描时收到的值和来自其父结点的值，然后将组合结果发送给其右边子结点。

当叶结点收到tally值时，它必须和存储在操作数数组中的值进行组合，以计算前缀的总和，即结果。在图5-6中，这些操作数以图示的方式出现在底部的方框中。因此scanGen()过程生成了最终结果。

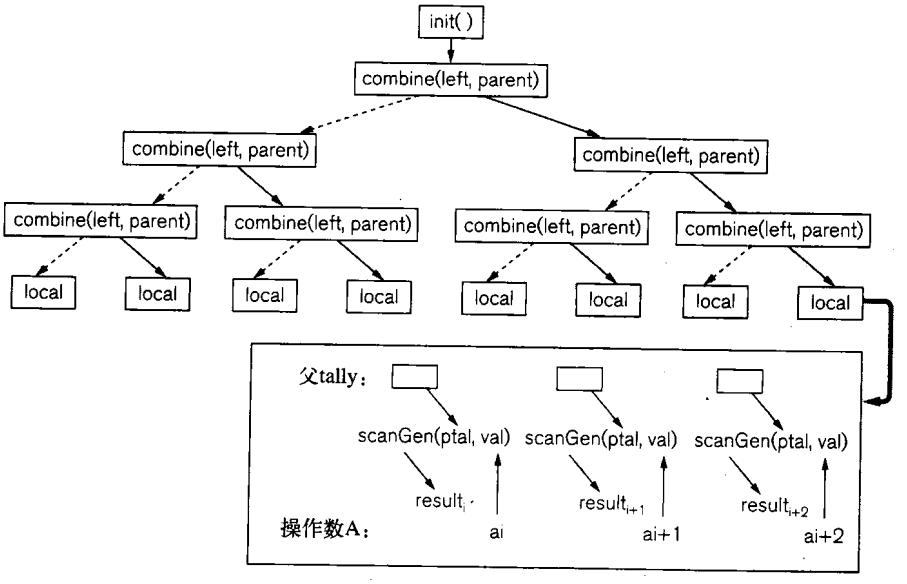


图5-6 扫描操作示意图。算法的第一部分是简单的通用归约，如图5-3中所示。一旦找到全局tally之后，前缀会根据以下规则沿着树向下传递：父结点的值被发送给左边子结点（用虚线箭头表示），combine(left,parent)被发送给右边子结点；当前缀达到叶结点后，本地操作应用scanGen()函数，并在操作数元素中存储结果

图5-7显示了通用扫描的逻辑。对比图5-4中的逻辑，我们注意到以下几点：

- 分配全局存储（ltally），用来保存在向上扫描过程中左边子结点的值。(参见第19行)。
- 归约和扫描一直到第28行都是完全相同的，除了保存左边子结点的值。
- 调用init()函数（第33行），用来为根结点提供父结点的值。

```
 1  int nodeval'[P];                                          全局的满/空变量
 2  int ltally[P];                                            保存组合操作的左操作数
 3  forall(index in(0..P-1))
 4  {
 5    int myData[size]=localize(operandArray[]);              局部数据值
 6    int tally;
 7    int ptally;                                             从父结点获得的tally
 8    int stride=1;
 9    tally=init ();                                          初始化
10    for(i=0; i<size; i++)
11    {
12      tally=accum (tally, myData[i]);                       累加
13    }
14    nodeval'[index]=tally;                                  最初发送到父结点
15    while(stride<P)                                         开始树的逻辑
16    {
17      if(index%(2*stride)==0)
18      {                                                     组合
19        ltally[index+stride]=nodeval'[index];
20        nodeval'[index]=combine (ltally[index+stride],
21                                 nodeval'[index+stride]);
22        stride=2*stride;
23      }
24      else
25      {
26        break;
27      }
28    }
29    stride=P/2;
30    if(index==0)
31    {
32      ptally=nodeval'[0];                                   清空存在的上扫值
33      nodeval'[0]=init ();                                  发送init()作为父结点输入
34    }
35    while(stride>1)                                         开始树向下的逻辑
36    {
37      ptally=nodeval'[index];                               获得父结点
38      nodeval'[index]=ptally;                               将其发送到左子结点
39      nodeval'[index+stride]=                               发送父结点+左子结点到右子结点
          combine (ptally, ltally[index+stride]);
40      stride=stride/2;                                      降到下一个层次
41    }
42    for(i=0; i<size; i++)
43    {
44      myResult[i]=scanGen (ptally, myData[i]);              生成扫描
45    }
46  }
```

图5-7　通用扫描程序。第35行开始向下广播tally值，将中间结果分发到所有线程，以此计算最终结果（第44行）

- 所有的线程进入向下扫描的while循环（第35-41行），但只有那些已接收父结点值的线程能参与。它们保存了父结点的tally（第37行），然后将它发送给左边子结点（第38行），并与已保存的左边子结点的tally值进行组合，然后发送结果给右边子结点（第39行）。
- 向下扫描过程分发ptally给每个线程，在scanGen()中会用它计算最终的结果（第44行）。

注意在该扫描中使用了两种不同形式的init()和combine()。

5.3.5　应用通用扫描

为了说明通用扫描的操作，想象有一个整数数组A，由序列1..*k*组成。扫描

```
lastOccurrence\ A
```

将为最近一次出现的A[i]返回索引位置i；若A[i]是第一次出现，则返回0。我们使用一个有k个元素的数组作为tally（计数器），初始化为0。如果A[i]为j，则accum()函数将i存储到tally[j]中作为最近一次出现。combine()函数选取两个数组中各元素中的最大值，然后扫描生成器会重新处理数据块，使用ptally作为它的初始值。图5-8显示了产生该结果的4个函数。（注意最后的tally数组将是数组的直方图）。

```
tally init(tally t)                                    设置全局分配的数组
{
  for(i=0; i<k; i++)
  {
    t[i]=0;
  }
  return t;
}
tally accum(int elem, tally t, int i)                  局部累加，需要数组索引
{
  t[elem-1]=i;
  return t;
}
tally combine(tally left, tally right)                 组合到“左边”
{
  for(i=0; i<k; i++)
  {
    left[i]=max(left[i],right[i]);
  }
    return left;
}
int scanGen(int elem, tally t, int i)                  完成扫描
{
  t[elem-1]=i;
  return t[elem-1];                                    与父tally累加
}                                                      保存运行计数器
```

图5-8 定制的扫描函数，用来返回位于第i个操作数位置的元素最后一次出现时的索引；tally被全局分配为有k个元素的数组

5.3.6 通用向量操作

我们已经使归约和扫描操作相当通用化了。虽然所使用的是一些容易编程和解释的小例子，但是必须清楚有许多计算可以纳入这种类型。实际上，有一种方式是将许多数据并行计算看成是一系列的本地操作，当需要更多的全局信息时，就交替地执行扫描和归约。在这种方式中，起作用的归约和扫描都丧失了它们各自的特性，此时计算可被认作是一个通用的向量操作。在这种方式中，本地和全局计算都进行了高效计算。这是一个值得记住的程序设计方法。

5.4 静态为进程分配工作

在我们关于归约和扫描的讨论中，隐含着这样的概念，即我们的数据结构项将会成组地分配到不同的进程。现在来具体研究分配数据和工作到进程的问题。本小节将讨论静态分配工作到进程的方法，下一小节将考虑动态分配工作到进程的必要性。

为了与第4章中所提到的可扩展并行方式保持一致，我们所构思的分配将是针对某个逻辑

进程集的，这样所产生的程序就不会绑定到任意特定数量的处理器上。例如，对于2维数组A，可以以大小为u × v的块分配到各个线程。以后可根据需要，选择合适的、不同的u和v的值对该设置进行调整。

首先考虑静态分配数组的实例。我们的基本方式是静态分配数据到线程，然后赋予每个线程计算它所“拥有”数据的责任。在共享存储器系统中，分配数据到线程可能是隐式的，即数据结构是全局分配的，每个线程却只计算整个数据结构的一部分，这一部分就自然而然地迁移到其cache层次结构部分。在分布式存储器系统中，每个进程将分配到自己部分的数据。但无论是哪种存储器模型，都会要用到本小节中的概念。

5.4.1　块分配

如果我们的目标是利用局部性，就应当遵循如下的原则：即将一个数据结构中大部分的连续计算部分一起分配到同一进程。这就是我们所熟悉的在cache中对空间和时间局部性的利用。因此一维数组通常会以连续索引块的形式分配到进程。对于二维数组而言，二维块（即在2个维上都连续的索引）形式的分配通常会产生高效的解决方案。此外，二维的块分配显得比整行分配更为合理，因为块分配通常能减少通信，例如，对于那些依赖于邻居结点值的计算，如所谓的模板计算（stencil computation），

```
B[i,j] = (A[i-1,j]+A[i,j+1]+ A[i+1,j] - A[i,j-1])/4.0;
```

块分配就比行分配要好，这可以从图5-9中看出。类正方块的数组值具有如下的性质，即

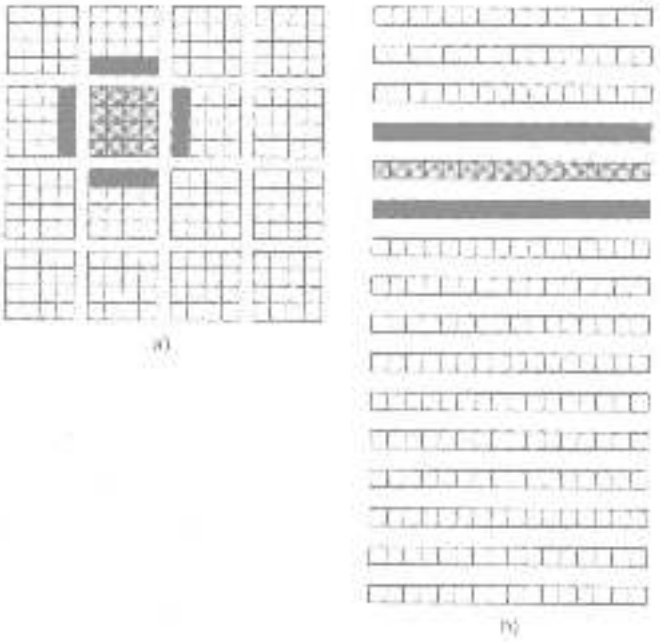


图5-9　向16个进程分配一个16 × 16数组的两种方案：a) 二维分块，b) 行。对于每个元素依赖于左边和4个最近邻居的计算，用灰色值表示的进程必须与右的邻居进行通信以获得黑色值。行分配所需传输的值为块分配的两倍，而且由于表面积对体积的优势，块分配的好处会随着本地元素的增长而更加显著。一个复杂的问题是，事实上行分配每个线程能发送更少的消息。第7章中解释了为什么通常而言，是所要发送消息的数量重要，而不是所要发送消息的大小，因为发送消息的软件开销通常相当大。而实际数据的发送通常被高效地流水化了。但是注意，行分配的列数是固定的，而二维块分配的列数会随着数组大小而扩展

它位于边界的元素，必定会被进行模板计算的其他进程所访问。当块大小增加时，边界元素数的增加起步缓慢的多，从而减少了通信开销。我们说，块具有表面积对体积的好处（surface area to volume advantage）。因为当大小（体积）增加时，边界元素数（表面积）增加得较慢，这就意味着在线程或进程间只需发送较少的数据。图5-9中的4×4小例子需要从邻居处接收$4\times4=16$个元素，而如果是按行分配，就需要从邻居处接收$2\times16=32$个元素。在更大规模的问题中，例如将1024×1024的数组分配到$P=256$个处理器，若按2维块布局进行分配，则每个进程将分配到64×64个元素。而如果按1维布局进行分配，则会将4×1024个元素作为整行分配给各个进程。块分配只需要从邻居处接收$4\times64=256$个元素，而行分配就会需要从邻居处接收$2\times1024=2048$个元素。

对于更高维的d维数组，有时会按d维块进行分配，以使其具有表面积对体积的好处。但实际上，只全局分配两个维，而使其他维的分配局限于局部才是最常见的。后面这种选择通常或是因为没有足够多的进程数进行d维分配，或是因为各维间极端的尺寸比。

5.4.2 重叠区域

我们在大计算块上操作的原则导致了重叠区域概念的产生，这是软件cache的一种类型，例如，在许多基于数组的计算中，每个元素值都是通过访问一个固定邻居组的数组值计算得来的，这产生了所谓的模板计算，因为它们的存储器访问模式（以及所引入的通信模式）是一个能作用到数组中每个点的模板。这些计算需要访问由邻居进程拥有的值。例如，4个最近邻居点的模板

```
B[i,j] = (A[i-1,j]+A[i,j-1]+ A[i+1,j]+ A[i,j+1])/4.0;
```

会访问存储于每个进程的4个邻居中的值。对于块分配而言，图5-9(a)显示了对于所分配的顶部行，它需要访问位于它之上的进程中的$A[i-1,j]$。对于块中其他边界元素的访问也可依此类推。这些邻居值最好按如下方式加以访问：

- 为一个分布式数组的本地部分分配存储时，分配一块额外的空间来保存将会被访问的非本地值，这种称为重叠区域的额外存储，被分配成与数组的本地部分连续，就如同该进程也拥有那些非本地数组部分。
- 从其他进程中获得所需的非本地值，并将它们存储到重叠区域。
- 现在可在完全是本地的数据值上执行计算，可以对这些已缓存的非本地值进行访问，直至拥有进程修改了它们的值，那时就需要获取新的值。

参见图5-10。

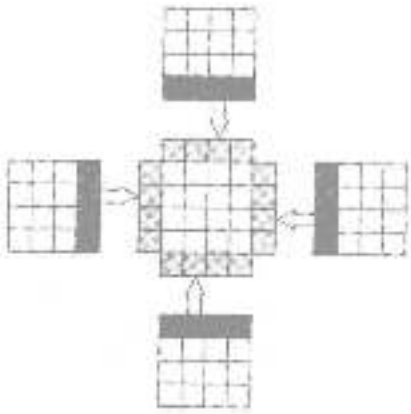


图5-10 数组块的重叠区域（灰色部分）显示相邻进程上的非本地值必须被移来以充重叠区域；一旦重叠区域被填充完，则模板计算就是完全本地的（未使用“缺失的角落”，但它们会被分配以简化数组索引的计算。）

有几个好处使得我们推荐这个方法。第一，一旦重叠区域被填满之后，计算中所有的访问都是本地的，这将导致大的独立计算块。第二，计算对于所有数组的访问都使用了相同的索引计算，因此它能在

单个循环嵌套中执行，而不会产生特殊的边界例外。第三，借助于批处理通信，我们能将多个跨进程的相关有效地组合成b相关，即计算只需访问来自b个邻居进程的数据。最后，借助于批处理通信，我们通常能减少通信代价。对于分布式存储器程序设计模型，数据的传输需要$t_0 + dt_c$秒来传输d个字节，此处t_0是（初始化）开销，t_c是传输每个字节的时间，批处理通信通过使用更大的消息分摊了通信开销。对于共享存储器机器，这个方法能通过移动整个cache行来减少通信代价，即无需再为单个数组元素请求建立互斥的存储器，使用缓存区域也有利于减少假共享。

5.4.3　循环分配和块循环分配

在那些工作量与数据量不成比例的算法中，块分配可能会存在糟糕的负载平衡。例如，就LU分解而言，这是一种线性代数计算，它将一个矩阵分解成2个矩阵的乘积，以求解线性方程组。这种计算在矩阵上进行迭代，且每次迭代都会求得一行和一列的结果。因此工作量在每次迭代后都会减少。图5-11显示了余下的工作量分布是不平衡的。图5-11a显示在白色列和黑色行中的结果均已分别求得。图5-11b显示了16个进程在逻辑上布局为一个网格，而图5-11c则显示了矩阵是如何使用块分配将数据分配到进程的。我们看到当这个算法进行时，那些拥有矩阵中黑色和白色部分的进程，它们的工作量将逐渐减少。例如，在第一个25%的行加到结果数组之后，位于数组左边和顶部的7个进程（44%进程）将无事可做。因此，在处理完所有行的四分之一之后，几乎有一半的进程就将处于空闲状态。事实上，最后25%的行工作是由进程P_{15}串行处理的。

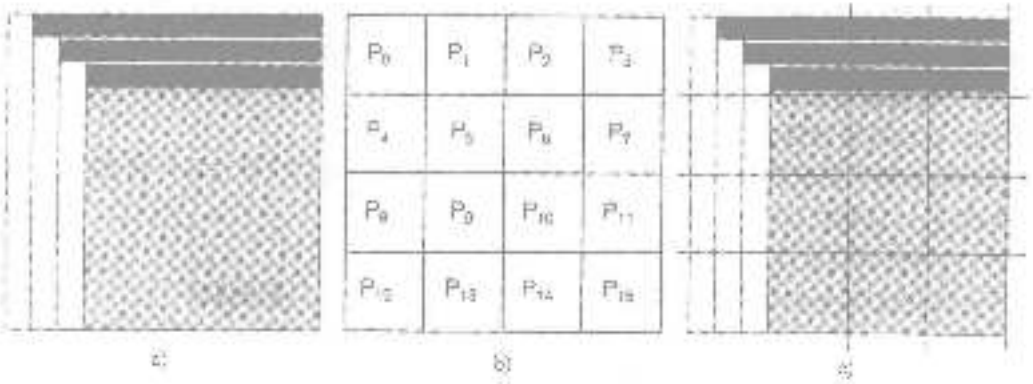


图5-11　示意图：a) LU分解算法，b) 16个进程以逻辑网格布局，c) 分布数组元素到进程，例如，进程P_5被分配了左上角完整的三行三列的数组部分

一个可以解决这种负载不平衡的方法是使用循环分配，即将各个元素以扑克牌游戏中常用的轮转发牌方式（round-robin）分到到不同线程（参见图5-12）。循环分配能平衡负载，且为它试图在多个进程间分配热点。但是，循环分配会由于增加了跨进程相关的数量，从而增加通信代价；循环分配实际上是“大计算块”方法的对立面。

为了在局部性和负载平衡之间进行折衷，使用块－循环（block cyclic）分配是一个常用的做法，即以块为单位进行循环分配。图5-13显示了矩阵中的连续块以块－循环分配方式循环地分配到不同进程的情况。图中显示出了块－循环分配的一些特征。第一，块的大小不必整除矩阵的大小；仅当需要时才对块进行简单的分割。第二，每个进程从整个矩阵中接收块，

因此当计算进行时，已完成部分和未完成部分均将驻留于各个进程上。图5-14显示了图5-13的分配在LU计算执行到中途时的情况。请注意余下的工作在进程间平衡得相当好。

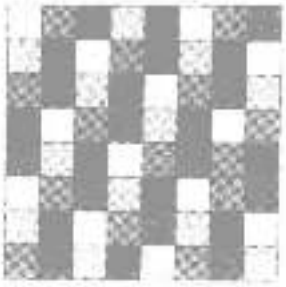


图5-12 将8×8数组循环分配到5个进程的图示

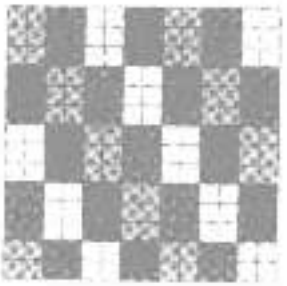


图5-13 以2×2的块将14×14元数组块循环分配到4个进程（以不同颜色标出）

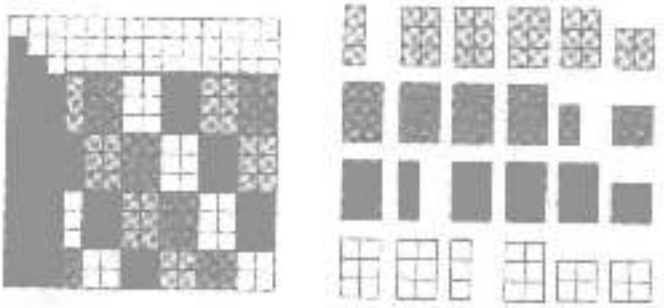


图5-14 至5-13的块循环分配在计算中途时的情况。位于右边的块简要表示了各个进程上的活跃值

图5-13中所示的特殊分配使用了3×2的块，我们看到每个元素都与分配到不同进程上的元素相邻，因此如果计算的某些其他部分必须要访问那些最近的邻居时，这种小块的规模很有可能会导致相当可观的开销，第一，生块之间会有相当大的通信。第二，每个3×2块的重叠区域需要比块本身更大的存储器，第三，很小的块无法利用局部性。使用较大的块，比如64×64，能减少通信总量，并减少每块所分摊的开销。但是如果块变得太大，负载不平衡的可能性就会再次增加。一般而言，平衡块－循环分配之间各个相互矛盾的目标是一件需要慎重考虑的工作。

例：使用循环分配的负载平衡 基于曼德勒罗特集（Mandelorot set）的朱莉亚集（Julia set），由以下迭代函数定义：

$$z_{n+1} = z_n^2 + c$$

其中z_n是函数第n次迭代的值，c是一个复数系数，它决定了朱莉亚集在复数平面上的形状，函数的第1次迭代被定义为$z_0 = a_0$，其中a_0是某个常数值。为了针对给定的系数c生成一个集，复数平面上（基于某一指定的精度）的每个点将一直迭代，直至结果的绝对值超过2.0，或迭代的次数达到某一上限值。复数平面上每个点所经历的迭代次数决定了该点的颜色。

我们将创建朱莉亚集作为一个需要负载平衡的并行计算实例（参见图5-15）。虽然计算是易并行的（因为每个点都能独立计算），但收敛所需的迭代数则是高度可变的。如果我们分配工作到进程在空间上是基于图中每个点的位置，则某些进程将远先于其他进程完成迭代。实际上，基于邻近原则的绝大多数分配策略都将导致糟糕的分配。

一个显而易见的解决方案是以循环分配的方式将像素分配到进程。因此像素(0, 0)将被分配到进程P_0，而像素(0, 1)将被分配到进程P_1。一般而言，如果图像是$x \times y$，则像素(r, s)将被分配到进程$P_{(y*s+r) \bmod p}$。如果P不能整除y，则分配将确保任务在图像上是规则采样的，在易与难的像素间给出了大致平衡的分配。当P能整除y时，则分配将导致像素按垂直线分布，这或许仍然是可接受的。但万一无法接受，我们还能基于一个小的$k \times k$像素区域分配工作，以避免该问题。此处k对行的大小是个相对素数。

5.4.4　不规则分配

当然，还有许多算法使用了非数组的其他数据结构，如非结构化网格。这些形状不规则的网格通常由三角形组成，常会在有限元计算中用到，可用来仿真诸如机翼的流体动力学（参见图5-16）。虽然使用了更为复杂的数据结构，但仍可应用相同的原理。假设交互发生在网格的边界，通过识别出具有较大表面积对体积比的网格的大部分，可最小化进程间的通信。这种划分技术可分为两类：基于几何的划分和基于图像理论的划分。划分技术的相关文献通常可通过搜索诸如"网格划分"之类的术语访问到。

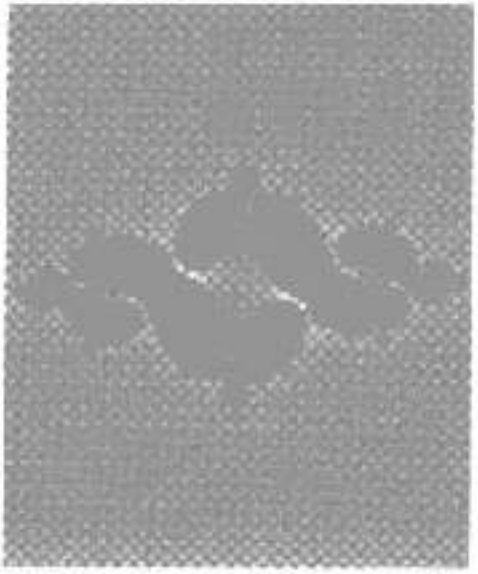


图5-15　朱莉亚集（Julia Set），从网站http://alephO.clarku.edu/~djoyce上生成

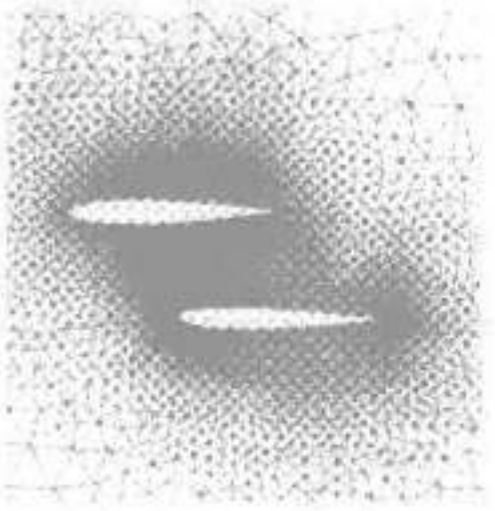


图5-16　非结构化网格示例。图中显示了两个机翼的压力分布，该图来源：http://fun3d.larc.nasa.gov/example-24.html

由于数据访问是非规则的（经常直到运行时才会清楚），因此很可能这是一种非常低效的获取非本地值的方法：即发现一个访问是非本地的，获得它，发现下一个访问是非本地的，获得它，以此类推。为了避免这种细粒度、串行的解决方案，并希望能用到好的并行概念，人们研发了一种称为检查者/执行者（inspector/executor）的技术。在执行一次迭代或其他一大段代码之前，对不规则数据结构的数据访问进行"检查"，即分析并识别出哪些是非本地的，以及它们所分配于的进程，所有对非本地进程的访问将先分批，再成批访问。之后，当数据变为本地后，执行者才执行计算。注意这与数组分配中使用重叠区域以缓存非本地数据的思路是相同的。

检查者/执行者是动态分配技术的一个例子。我们现在就来观察其他的动态分配技术。

5.5 动态为进程分配工作

在许多应用场合，采用固定的工作分配是不可能的，因为新的工作会在计算过程中动态地创建。这些例子包括：处理客户端请求的服务器；一种能创建新的工作以使用额外计算能力（因为解决方案需要有最大的计算能力）的*自适应算法*；许多图算法。在另一些应用场合，计算能力的总量与数据的总量是不成比例的，因此静态分配将导致糟糕的负载平衡。在上述这些情况中，动态工作分配能平衡负载，因为空闲进程会被分配到额外的工作。本小节将描述工作队列，这是动态分配工作到进程的一种策略。我们的讨论将关注于各个进程与工作队列之间的交互，但关键是要记得工作队列中的任务可以直接与队列中另外一个任务进行交互，在开发一个并行算法时同样需要考虑到这种交互。

5.5.1 工作队列

工作队列是用来向线程或进程动态分配工作的一种数据结构，它在许多串行算法中都有用到，尤其当工作会在计算过程中动态生成时，因而将其推广到多个进程也是顺理成章的。最简单的工作队列是先进先出（FIFO）的任务描述符表，其中新创建的任务会添加到队列的一端，而欲处理的任务则从队列的另一端移走。例如，*常见的生产者/消费者*（producer/consumer）范例就使用了FIFO表，用以在生产者进程与消费者进程之间通信作业。

作为一个能以工作队列形式表达的普通例子，让我们考虑$3n+1$猜想（或称为Collatz猜想）问题：它建议对下述提问给予肯定的回答，“对于任意的正整数a_0，如下定义的计算步骤是否会收敛于1？”（参见http://mathworld.wolfram.com/CollatzProblem.html）。

$$a_i = \begin{cases} 3a_{i-1}+1 & a_{i-1}\text{为奇数} \\ a_{i-1}/2 & a_{i-1}\text{为偶数} \end{cases}$$

例如，对于$a_0 = 15, 16, 17$，则相应的a_i序列便为：

15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1

16,8,4,2,1

17,52,26,13,40,20,10,5,16,8,4,2,1

借助于计算机的搜索，我们已知该猜想对于所有小于$3 \cdot 2^{53}$的整数都是正确的，但我们感兴趣的问题是“定义为$\max(a_i) / a_0$的最大扩张因子是多少？”，即相对于初始值，最大的中间结果会有多大？对于15，扩张因子是160/15=10.67；对于16，它是1；而对于17，它是52/17=3.06。（现在还不清楚该扩张因子的值有没有界限，或许真正意义上，这个问题更为重要。）我们将以实现最大扩张因子的搜索作为例子，因为这是一个能多方面展示工作队列的简单例子。

我们的解决方案假定工作队列包含了要测试的下一个整数。对于P个进程，我们初始化队列为前P个正整数：

```
int i;
for(i=0;i<P;i++)
{
g[i]=i+1;
}
```

此处的整数是任务描述符。作为一个通用原则，即当任务描述符能自包含时，让它们尽

量小是明智之举。在本例子中，使用整数已然足够了，因为它们就是要测试的数字。

每个线程都将扮演生产者和消费者的双重角色。作为消费者，它从队列中移除第一项；作为生产者，它将在它刚移除的值上加上P，然后将新值附加到队列末尾。加上P的基本原理是由于有P个线程会同时检查整数，因此前进P就能跳过那些将被其他线程处理的值。线程的逻辑如图5-17所示。

```
1  float ef=1.0;                          最佳扩张因子
2  int bestA=1;                           记录哪个a_i有最佳扩张
3  int head=0;                            Q中下一个要处理的项
4  forall(index in(0 .. P-1))             定义P个线程来执行测试
5  {
6    int a=1;                             测试数量
7    float myEF=1.0;                      本地扩张因子
8    int big, atest=1;                    用来计算扩张因子的本地变量
9
10   while(a<runSize)                     限量20亿，或扩张到单精度
11   {
12     exclusive                          从Q中获得a，然后留下a+P
13     {
14       a=q[head];
15       q[head]=a+P;
16       head=(head+1)%P;
17     }                                  单线程区域的结尾
18     atest=a;                           为测试进行设置
19     big=a;                             开始值可以是我们见到的最大值
20     while(atest!=1)
21     {
22       if(even(atest))
23       {
24         atest=atest/2;
25       }
26       else
27       {
28         atest=3*atest+1;
29         big=max(big, atest);
30       }
31     }
32     myEF=big/a;                        为这个a计算扩张
33     exclusive                          从Q中获得a，然后留下a+P
34     {
35       if(myEF>ef)                      记录任何的进展（在第一次之后）
36       {
37         ef=myEF;
38         bestA=a;
39       }
40     }
41   }
42 }
```

图5-17　计算Collatz猜想扩张因子的代码

在声明变量和创建线程之后，计算进入while循环以处理一组整数。该过程的第一步是进入一个单线程区域，以允许线程无竞态地处理全局队列。当下一个数从队列中移除时，队列将用新数进行更新，同时队列头前进一步。

在计算的主体部分，线程设置了两个值（big, atest），然后又进入了一个定义了a序列的while循环。当每个$3\times a+1$发生变化时，新的a将被测试，以查看是否比之前所看到的要大。

如果是，则它会被保存下来。当a==1时，该循环结束。在结尾部分，会再次进入临界区，以检查最后一个扩张因子是不是新的高值。如果是，则将它和产生它的a_i一起记录下来。

现在来考虑工作队列的行为。首先，注意虽然P个子序列相当于是被同时检查的，但它们并不会保持锁步，因为每个检查所需的工作量不尽相同。例如，对于4个进程，队列在开始时可能通过如下状态进行转换，其中最近的赋值以粗体字表示：

工作队列	活跃进程
1, 2, 3, 4	–
2, 3, 4, 5	P_0[**1**]
3, 4, 5, 6	P_0[**2**]
4, 5, 6, 7	P_0[2], P_1[**3**]
5, 6, 7, 8	P_0[2], P_1[3], P_2[**4**]
6, 7, 8, 9	P_0[5], P_1[3], P_2[4]
7, 8, 9, 10	P_0[5], P_1[3], P_2[4], P_3[**6**]

我们看到，由于对1的处理很容易，因此P_0甚至有可能在其他任意进程开始之前，就返回队列去获取下一个值2。同样，对2的处理也很容易，因此P_0会再次迅速返回。同时也应注意工作者不必处理一个有类似间隔的子序列。实际上，如果定时合适，则所有进程都能同时在同一子序列上工作。简而言之，虽然我们的工作队列是编组的，但是其自然动态性使得它允许包容许多不同的定时特性。

5.5.2 工作队列的变体

有许多工作队列的变体。从算法上来说，队列中的元素不必一定非要以先进先出（FIFO）的次序进行处理，可以按后进先出（LIFO）、随机或优先级的次序，而且每一种都有其特定的用途。

工作队列的具体使用也能基于不同的工作粒度而发生变化。在我们的例子中，我们假设工作单元是单个工作描述符，正如同在Collatz猜想问题中那样，它应当尽可能的小。但是，如同我们曾在第3章中提到的，工作粒度体现了折中。细粒度会增加队列操作的开销，会增加竞争的可能性，而粗粒度会增加遭受负载不平衡的机会。

一个试图在开销与负载不平衡之间获取平衡的策略是使用*可变粒度大小*，它与队列中元素数成比例。当队列为满时，线程或进程会抓取大块的工作，假设队列中有足够多的工作能保证所有进程都在忙碌。当队列中元素数减少时，粒度大小也会随之变小。

迄今为止，我们所考虑的仅是单个工作队列，它当然是不可扩展的。一个可扩展的策略是使用多个队列，使每个线程或进程被分配到不同的队列或队列集。此种方法将减少竞争，但如果线程在找到一个非空队列之前需要先检查多个队列，则它将增加时延。多个队列的使用也会引入负载平衡的问题。当一个队列为空而其他队列非空时，将会发生什么？一个解决方案是让线程从其他队列中获取尚未分配的工作，这个行为通常被称为*工作窃取*。

5.5.3 案例研究：并发存储器分配

考虑为对称多处理器实现一个存储器分配器（allocator）的问题。存储器分配器在接收到一个分配存储器的请求后，会返回一个所请求大小的存储块地址。好的分配器会随着处理器的数量而扩缩，会避免假共享，会有效使用虚拟存储器。另外一些目标诸如较少的碎片（空

闲的存储器不应被分割成许多的小片段），当然也是很重要的，但为了简单起见，我们将在本次讨论中忽略这些问题。

在解释一个优秀的解决方案之前，我们先来解释一些之前的解决方案以及它们所存在的问题：

- 最简单的解决方案是使用单个堆来满足所有的请求。但当有多个处理器访问该堆时，随着进程数的增加，对该堆及其互斥体的竞争将会成为瓶颈。
- 常见的第二种解决方案是为每个处理器配备一个私有堆。因为没有共享，所以这些堆也不需要锁。但这种方案会导致无限制的存储器分配（甚至当使用了有限数量的存储器时亦是如此），这将最终破坏程序。为了证实这一点，考虑两个处理器p和c之间存在生产者和消费者的关系。处理器p将分配那些被c消耗的存储器。当c用完某块存储器之后，它会将其释放，并将其返回给c的私有堆。由于p从不会访问那些空闲存储器，因此它必须继续分配存储器以满足每个对p的请求，而这将最终导致无限制的存储器分配。
- 第三种解决方案通过使用具有所有权的私有堆以避免存储器分配的无限制增长，这意味着每片存储器都是由所分配的处理器拥有的，因此当它被释放后，它将返回到拥有它的处理器的堆中。因此在我们生产者和消费者的场景中，c将返回存储器到p的堆中。由于p能重用之前它分配的存储器，因此我们不会再看到存储器分配的无限制增长。该解决方案的问题在于没有共享空闲存储器，因此当一个处理器分配存储器时，其他处理器都无法从中获益。所以该方案将导致存储器分配的P倍增长。考虑有P个处理器在生产者－消费者的关系中形成链的例子，此时处理器$i \bmod P$将分配一块存储器块，而处理器$(i+1) \bmod P$将释放该块。在此种情况下，单个块会由处理器0产生，而且从概念上讲，可以在各个处理器之间流转，直至它回到处理器0，因为当它一旦被释放就立即会被另一个处理器分配。因此在理想的系统中，有可能在整个来回过程中只分配和重用了单个块，但实际上具有所有权的私有堆会为每个处理器分配一块存储器块，共P块。因子P的增长是非常显著的。例如，在一个32位地址空间的16个处理器的机器上，每个处理器在用完虚拟存储器之前，各自只能分配到容量不大的128MB存储器。
- 某些存储器分配器会在从同一cache行分配存储器给不同处理器时，引入假共享。我们当然知道可以通过扩展存储器的分配，即让它们的大小是多倍cache行的大小，以此来避免假共享。但分配器却不能执行这种扩展，因为这会显著增加所分配的存储器总量。

Hoard存储器分配器通过应用以下两个原则，解决了上述这些问题：

1. **限制本地存储器的使用**。如果一个私有堆变得太大，它会移一些块到全局堆中，而全局堆是能被其他堆所访问的。尤其是，当一个堆为空时，在分配新块之前，它会先尝试使用全局堆中的存储器来满足分配请求。

2. **以大块的方式管理存储器**。大块的使用将减少假共享，并减少对全局堆的竞争。

虽然忽略了许多细节，但我们仍能如下总结Hoard存储器分配器的优势。多个私有堆的使用提供了能随处理器数扩缩的并发性；全局堆的使用提供了负载平衡的机制，使空闲的页面不会堆积在私有堆中；大块的使用遵循了我们增大粒度以减少开销和竞争的通用性原则。

Hoard存储器分配器在许多现有操作系统中都常用到，包括Solaris，Linux和Windows，也有许多使用Hoard存储器分配器之后（借助于它能更有效使用存储器的特点）显著提升程序性能的实例。此外，Hoard存储器分配器已被证实可用来约束存储器崩溃以及保持较低的同步开销。

5.6 树

树是重要的数据结构，因为它体现了层次的结构，例如，kd 树能用于图像绘制和重力仿真中，它将空间划分为层次的单元。树结构使得算法能在与树高成正比的时间内找到在该空间中的项。

对于并行计算提出了多个挑战。第一，树通常使用指针来构建。在不提供共享存储器的语言中，指针只对一个进程是本地的。第二，我们通常出于树的动态灵活性而使用它，但动态行为通常也意味着存在大量限制性能的通信。第三，一些非结构化的树使得推断通信和负载平衡变得极为困难。但无论有多大挑战，树确实非常有用，因此无法将其忽略。

5.6.1 按子树分配

最大化大块独立计算数的方针也适用于树，正如同我们希望将整棵子树分配到单个进程。例如，我们能复制称为“帽子”的树顶，然后将它的一个副本和一棵子树分配到P个不同的进程之一，如图5-18中所示。如果子树间没有通信，我们就得到了一个可扩展的解决方案。

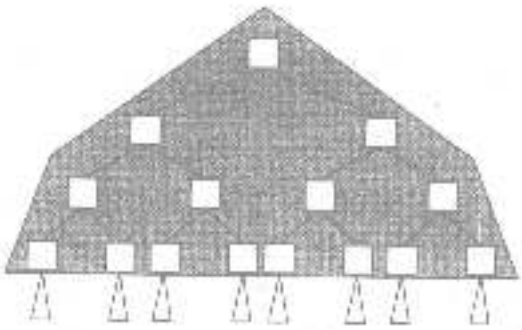


图5-18 为P = 8个进程的二叉树分配帽子。每个进程分配到一个叶子树，以及一个帽子的副本（用阴影表示）

帽子分配是高效的，因为帽子接近于根结点，对于树的其余部分而言相对较小。而且，由于根结点和它的直接后代在所有进程上都可用，那些需要通过根结点传递的交互就能使用本地数据来识别正确的目标子树。通过目标子树的导航通常作为一个任务，分配给其所有者。作为额外的优点，在重力仿真的高级算法中，问题的大块区域聚合成“元点（meta-point）”，根结点周围的点就有可能成为所有的元点。因此一旦创建后就是只读的，这消除了通信、竞态条件以及锁等一系列问题。当然，当计算进行时，帽子中的变化必须在进程间保持一致，这意味着所有进程都必须看到相同的状态。反之，如果树顶被频繁修改，就应当将其以各种方式分布在各进程中。

当整棵树在计算开始时枚举时，或是当结点以层次顺序的方式生成时，上述这种分配方法会很有效。例如，图5-19a显示了如何将一棵二叉树分配到8个进程，而图5-19b显示了如何将同一棵树分配到6个进程。

游戏搜索举例 以子树方式进行分配，对那些能递归分解为子问题的问题会比较有效。例如，假设我们在P=4个进程上搜索Tic-Tac-Toe(圈和叉)游戏树。考虑到对称性，只有3个初始位置，我们扩展其中之一以生成第4个搜索任务（参见图5-20），也即是各个进程将从所指示的板位置开始搜索游戏树的后代。

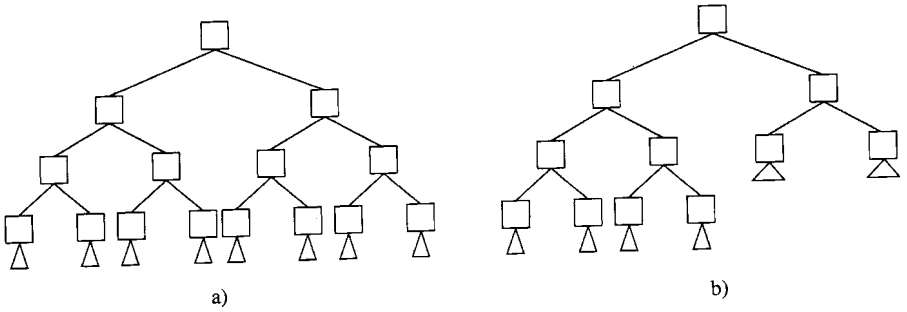


图5-19　逻辑树的表示：a) $P = 8$时的二叉树；b) $P = 6$时的二叉树

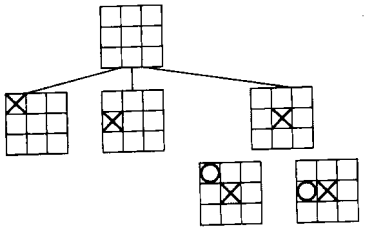


图5-20　枚举Tic-Tac-Toe游戏树；每个进程被分配来搜索以四个初始移动序列之一开始的游戏

5.6.2　动态分配

某些树以不可预测的方式动态进行分配，而另一些树则具有不平衡的深度。例如在游戏搜索中经常会用到带alpha-beta修剪的搜索，它将基于之前看到的结果，修剪掉一部分层次搜索空间，这将导致带有叶结点的搜索树有不同的深度。在这种情况下，可使用工作队列，队列中的项表示未分配的树结点。相同的折衷原则曾在动态分配进程工作一节中讨论过，也可以在此处应用。当然，在alpha-beta修剪这个例子中，需要有某种通信机制用来在进程之间通信之前的结果。这些结果可使用共享存储器或某种组合树进行通信。

5.7　小结

本章中，我们讨论了开发可扩展并行计算所需的算法技术和分配技术。首先讨论了通用归约和扫描抽象的强大功能，然后讨论了为进程分配工作的不同方法。我们看到了一个贯穿于全章、隐含的简单指导性原则（识别出大的独立计算块），同时也看到了有时负载平衡问题会如何干扰该目标。

历史回顾

依靠社区求解的问题已经流行了十多年（SETI@Home可能是其中最著名的一个，但是BOINC项目还列出了许多其他类似的问题）。Lander和Fischer[1980]提出了并行前缀算法的关键性思想，而Deitz[2005]则提出了用来分解参数化扫描的4个函数；Blelloch[1996]也在极力倡导使用扫描和通用向量操作。非规则网格的划分算法中最富盛名的或许要属由Pothen,Simon和Liou[1990]等人提出的递归频谱二分法。Hoard存储器分配器由Berger等人[2000]提出。

习题

1. 重做图5-2中的Schwartz模板。原图中使用了二叉树，重做时要使树的扇出度d>2。假设P和d都是2的幂。
2. 重做图5-2中的Schwartz模板。这次假设P不是2的幂，其余要求与习题1相同。
3. 使用Schwartz方法，编写一个并行程序，用以确定在对应于MIPS机器指令的一串二进制字（word）的序列中，所包含的之后跟有空操作指令（定义为全0的一个字）的分支（branch）指令（定义为一个字中的bit_{28}==1）数。
4. 修改图5-17中Collatz猜想的代码，批处理1000个数。
5. 编写一个Peril-L程序执行图5-10中所示的操作，即在数据数组A中一个给定的区域上，完成4个点的模板计算。设置包括重叠区域在内的本地存储器，并假设数据最初是被分配到本地数组的内部。设置全局数据结构，以允许“相邻”线程交换边界元素。要包括交换的逻辑。提示：满/空变量能管理相邻线程之间的交换。
6. 编写一个Peril-L程序计算longest_run_of_1s/ A。
7. 编写一个Peril-L程序计算team_standings\ A。
8. 假设如边界框的例子一样，需要依次计算4个归约。编写一个Peril-L程序，它能将4个归约合并成1个。
9. 使用B+树的标准定义（依照串行算法书中所给出的），用共享帽子实现该数据结构。
10. 在使用共享帽子实现B+树数据结构之后，编写代码使得P个进程中的每一个从工作队列中取出查询，如果(a)该查询是针对本进程所管理的树的一部分，则回答所有成员的询问，否则(b)将该查询放到指定的管理相应树部分（其中含有所要查询的相应信息）的那个进程的队列中。

第三部分 并行程序设计语言

在前几章基础性的介绍中，我们抽象地建立了直观认识，介绍概念的形式与特定的计算机或程序设计语言基本无关。现在我们准备讨论一些实际的并行程序设计语言，编写程序并运行。我们将介绍多种不同方法，包括那些在实践过程中最常用到的方法。特别地，我们将考虑两种基本方法：线程和消息传递，以及多种高级语言。

线程的概念源自操作系统社区。现今随着多核芯片的出现，人们对基于线程的程序设计的兴趣日益浓厚。我们将介绍可能是目前应用最广泛的系统POSIX Threads，并通过它来阐明基于线程的程序设计中的一些主要问题。我们还将简单讨论OpenMP以及Java支持线程的能力，这两种是基于线程的程序设计的更高级方法。

对于大型并行计算机，消息传递是使用最广泛的程序设计方法。它诞生自科学计算社区。消息传递最主要的抽象是它对分布式存储器并行计算机的适用性。有两种广泛使用的消息传递库：并行虚拟机（Parallel Virtual Machine，即PVM）和消息传递接口（Message Passing Interface，即MPI）。它们在实现细节上或有所不同，但就能力而言，两者是相当的，因此我们将只介绍MPI。另外，我们还将扼要介绍适合在分布式存储器机器上进行程序设计的三种相关更高等级语言：Co-Array Fortran（CAF），Unified Parallel C（UPC）和Titanium。

迄今为止，尚没有一种广泛使用的高级并行程序设计语言，但相信总有一天会有的。这个信念驱使着我们继续深入研究，因此我们将介绍ZPL语言。我们主要感兴趣的是ZPL语言的能力，即它能给予程序员对其代码的并行行为有个合理、细致的理解，又提供了高级抽象的概念。这个正是我们希望在未来高级并行语言中看到的特性之一。这一章最后，我们还将介绍另一个更加高级的并行语言NESL。

最后，在了解了9个不同的程序设计系统后，我们专门辟出了较短的第9章，用以比较和对比这些不同的系统，同时也作为今后进一步研究以及开发并行语言的起点。

第6章 线程程序设计

本章将讨论为共享地址空间的计算机编写程序的方法。这类重要的计算机特别值得注意，因为很快它们就将随处可见。我们将首先详细讨论POSIX Threads（Pthreads）接口，它就是我们在第1章计3程序中所使用的关键性接口。我们的目的是要了解Pthreads程序员将要面对的重要概念和问题。在研究了关注于如何获得优秀性能的三个案例之后，我们将结束对Pthreads的讨论。此后我们将简单讨论另外两个线程程序设计方法，Java Threads和OpenMP，它们都试图隐藏Pthreads的一些复杂性。

6.1 POSIX Threads

我们介绍POSIX Threads有两个目的。首先是为了提供关于POSIX Threads标准足够具体的细节，这样读者就能开始编写Pthreads程序；其次是虽然模型中的设施（facility）单独而言都很简单，但整个程序设计的模型却会有许多重要的性能问题和正确性问题，其中有些还是相当微妙的，所以要在这里解释。为了达到这两个目的，本章虽然不能介绍Pthreads接口的每一个细节，但仍会以一定深度介绍Pthreads标准。我们将首先介绍创建线程和控制线程间交互的基本机制，然后讨论安全性问题和性能问题，最后关注于POSIX Threads使用过程中可能会遇到的一些实际问题。

在线Pthreads教程 想了解Pthreads的更多细节，包括例子和文档，请参考美国Lawrence Livermore实验室的网页：http://www.llnl.gov/computing/tutorials/pthreads/

6.1.1 线程的创建和销毁

考虑下列代码：

```
 1  #include <pthread.h>
 2  int err;
 3
 4  void main()
 5  {
 6    pthread_t tid[MAX]; /* 线程ID的数组，每个*/
 7                        /* 创建的线程对应数组的一个元素*/
 8
 9    for(i=0; i<t; i++)
10    {
11      err=pthread_create(&tid[i], NULL, count3s_thread, i);
12    }
13
14    for(i=0; i<t; i++)
15    {
16      err=pthread_join (tid[i], (void **)&status[i])
17    }
18  }
```

这段代码展示了main()函数，它在第1个循环中创建并初始化了t个线程，然后在第2个循环中等待这t个线程结束。我们通常称创建的线程为父线程，被创建的线程为子线程。

这段代码与第1章中的伪代码在细节上略有不同。第1行包括了Pthreads的头文件，而头文件中声明了各种Pthreads例程和数据类型。每个创建的线程都需要有自己的线程ID，因此在第6行声明了这些线程ID。为了创建线程，我们调用了pthread_create()例程，它有4个参数（参见代码规范6.1）：

1. 指向线程ID的指针。当该线程成功创建并返回时，它将指向一个有效的线程ID。
2. 线程属性（attribute）。稍后讨论；在这个例子中，默认属性为NULL值。
3. 指向起始函数的指针。起始函数是线程创建后立即执行的函数。
4. 一个传递给起始函数的参量（argument）。在这个例子中，它表示位于0与$t-1$之间的一个唯一整数，该整数与各线程相关。

第16行的循环调用了pthread_join()以等待每个子线程的终止。如果无需等待子线程结束，main()函数可以通过调用pthread_exit()结束并退出，而子线程仍能继续执行。否则当main()函数结束时，由于整个进程被终止，会导致子线程自动终止。参见代码规范6.2。

代码规范6.1 pthread_create()。POSIX Threads中用来创建线程的函数

pthread_create()

```
int pthread_create(                          // 创建新线程
  pthread_t *tid,                            // 线程ID
  const pthread_attr_t *attr,                // 线程属性
  void *(*start_routine)(void *),            // 指向所要执行函数的指针
  void *arg                                  // 要传递给该函数的参量
);
```

参量：

- 创建成功的线程ID。
- 线程属性，稍后会解释。默认属性为NULL值。
- 新线程创建后立即执行的函数。
- 要传递给start_rountine()的参量。

返回值：

如果成功，则为0；否则为<errno.h>中的某个错误码。

注释：

使用一个结构可以向起始函数传递多个参量。

代码规范6.2 pthread_join()。POSIX Threads中用来结合线程的函数

pthread_join()

```
int pthread_join(                            // 等待线程的终止
  pthread_t tid,                             // 所要等待线程的ID
  void **status                              // 退出状态
);
```

参量：

- 所要等待线程的ID。
- 如果status不为NULL，退出线程的完成状态将被复制到*status中；否则，完成状态将不会复制。

返回值：

如果成功，则为0；否则为<errno.h>中的某个错误码。

注释：

线程一旦结合后，它将不再存在，其线程ID也不再有效，也不能再与其他线程结合。

线程ID　每个线程都有一个pthread_t类型的唯一ID。与所有Pthreads数据类型一样，线程ID必须作为*不透明类型*对待，即决不能直接访问该数据结构中的各个字段。子线程并不知道自己的ID，但pthread_self()例程能允许线程获得自己的ID，而pthread_equal()例程能用来比较2个线程的ID。参见代码规范6.3和6.4。

代码规范6.3　pthread_self()。POSIX Threads中用来获得线程ID的函数

pthread_self()

```
pthread_t pthread_self();                    // 获得线程自己的ID
```

返回值：

调用了该函数的线程的ID。

代码规范6.4　pthread_equal()。POSIX Threads中用来比较2个线程ID等同性的函数

pthread_equal()

```
int pthread_equal(                           // 测试等同性
  pthread_t t1,                              // 第1个操作数线程ID
  pthread_t t2                               // 第2个操作数线程ID
);
```

参量：

2个线程ID。

返回值：

- 如果这2个线程ID相同，则返回非0值（遵循C语言规范）。
- 如果这2个线程ID不同，则返回0。

销毁线程　有三种方法可以销毁线程：

1. 线程从起始例程返回。
2. 线程调用pthread_exit()；参见代码规范6.5。
3. 线程被另一线程取消。

在各种情况中，当线程被销毁后，其资源就不再可用。

代码规范6.5　pthread_exit()。POSIX Threads中用来终止线程的函数

void pthread_exit()

```
void pthread_exit(                           // 终止线程
  void *status                               // 完成状态
);
```

参量：

已退出的线程的完成状态。该指针值可供其他线程使用。

返回值：

无。

注释：

当线程通过从起始例程简单返回的方式退出时，线程的完成状态将被设为起始例程的返回值。

线程属性 每个线程有自己的特性（property），称为属性（attribute），存储在pthread_attr_t类型的结构中。例如线程可以是分离的（detached），或是可结合的（joinable）。分离的线程不能与其他线程结合，因此它们在某些POSIX Threads实现上有较少的开销。对于并行计算，我们很少使用分离的线程。线程也可以是约束的（bound），或是不受约束的（unbound）。约束的线程由操作系统进行调度，而不受约束的线程则由Pthreads库进行调度（参见稍后关于线程调度小节）。对于并行计算，我们通常使用约束的线程，这样各线程均能提供物理并发性。POSIX Threads提供了多个例程用来初始化线程属性、设置属性和销毁属性，参见代码规范6.6。

代码规范6.6 在POSIX Threads接口中如何设置线程属性的例子

线程属性

```
pthread_attr_t attr;                          // 声明线程属性
pthread_t tid;
pthread_attr_init(&attr);                     // 初始化线程属性
pthread_attr_setdetachstate(&attr,            // 设置线程属性
        PTHREAD_CREATE_UNDETACHED);
pthread_create(&tid, &attr, start_func, NULL);  // 使用属性创建线程
pthread_join(tid, NULL);
pthread_attr_destroy(&attr);                  // 销毁线程属性
```

注释：

还有许多其他线程属性，具体细节可参见POSIX Threads手册。

潜在的陷阱 以下的例子说明了由于父子线程之间存在交互，从而导致了潜在的陷阱。父线程简单地创建一个子线程，并等待该子线程的退出。子线程做一些有用的工作，然后退出，并返回错误码。你能看出在这段代码中存在什么错误吗？

```
 1  #include <pthread.h>
 2
 3  void main()
 4  {
 5    pthread_t tid;
 6    int *status;
 7
 8    pthread_create(&tid, NULL, start, NULL);
 9    pthread_join(tid, (void *)&status);
10  }
11
12  void start()
13  {
14    int errorcode;
15    /*做一些有用的事*/
16
17    if(. . . )
18    {
19      errorcode=something;
20    }
21    pthread_exit(&errorcode);
22  }
```

问题就发生在第21行调用pthread_exit()，子线程试图返回错误码给父线程时。由于errocode已声明为start()函数的局部变量，errocode的存储器就被分配在子线程的栈上。当子线程退出后，它的调用栈取消，因此父线程就有了一个指向errocode的虚悬指针（dangling pointer）。将来的某一时刻，当一个新过程调用时，它将改写errocode所在栈的位置，使得

errocode的值发生改变。

6.1.2 互斥

为了使线程进行结构化的交互，我们需要有协调交互的方法。特别当两个线程共享的访问存储器时，通常比较有用的是使用称为*互斥体*（mutex）的锁，用以提供对变量的互斥访问，也称为*互斥*（Mutual Exculsive）。如果没有互斥，竞态条件就将导致无法预测的结果。如同我们在第1章中所看到的，当有多个线程执行以下代码时，在所有线程间共享的count变量将不会原子地更新。

```
for(i=start; i<start+length_per_thread; i++)
{
  if(array[i]==3)
  {
    count++;
  }
}
```

解决的方法自然是使用互斥体来保护count的更新，如下所示：

```
1  pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
   . . .
2    if(array[i]==3)
3    {
4      pthread_mutex_lock(&lock);
5      count++;
6      pthread_mutex_unlock(&lock);
7    }
```

第1行说明互斥体能被静态的声明（参见代码规范6.7）。与线程相似，互斥体也有属性。通过将互斥体初始化为PTHREAD_MUTEX_INITIALIZER，互斥体就拥有了默认属性。为了使用互斥体，它的地址分别传递给第4行的加锁例程和第6行的解锁例程。当然，恰当的做法是将所有临界区（即代码每次只能被一个线程原子地执行）用括号括起来，这可以通过用一个互斥体在进入处加锁，用另一个互斥体在离开处解锁的方法加以实现。任何时候，只能有一个线程获得互斥体。如果某线程试图获得一个已经为另一线程获得的互斥体，它将被阻塞。当互斥体解锁或释放时，一个阻塞在那里并等待获得锁的线程将会被解除阻塞，并获得互斥体。POSIX Threads标准并没有定义公平性的概念，因此获得锁的顺序不能保证与试图获得锁的顺序一致。

代码规范6.7　POSIX Threads中用来获得和释放互斥体的函数

获得和释放互斥体

```
int pthread_mutex_lock(                    // 加锁互斥体
  pthread_mutex_t *mutex);
int pthread_mutex_unlock(                  // 解锁互斥体
  pthread_mutex_t *mutex);
int pthread_mutex_trylock(                 // 非阻塞的锁
  pthread_mutex_t *mutex);
```

参量：

每个函数都使用了一个互斥体变量的地址。

返回值：

如果成功，则为0；否则为<errno.h>中的某个错误码。

注释：

pthread_mutex_trylock()例程试图获得互斥体，但不会被阻塞。如果互斥体已经上锁，则该例程将返回POSIX Threads常量EBUSY。

互斥体的创建和销毁 在之前的例子中，我们知道只需要一个互斥体，因此能静态地分配。在一些不能预知所需互斥体数的情况下，可以动态地分配和释放互斥体。代码规范6.8和6.9就显示了这种互斥体如何动态分配，并使用默认属性初始化，最后销毁。

代码规范6.8 POSIX Threads中用来动态创建和销毁互斥体的函数

互斥体的创建和销毁

```
int pthread_mutex_init(                    // 初始化互斥体
  pthread_mutex_t *mutex,
  pthread_mutexattr_t *attr);
int pthread_mutex_destroy(                 // 销毁互斥体
  pthread_mutex_t *mutex);
int pthread_mutexattr_init(                // 初始化互斥体属性
  pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(             // 销毁互斥体属性
  pthread_mutexattr_t *attr);
```

参量：

- pthread_mutex_init()例程需要两个参量：指向互斥体的一个指针以及指向互斥体属性的一个指针。假设互斥体属性已经初始化了。
- pthread_mutexattr_init()和pthread_mutexattr_destroy()例程都将指向互斥体属性的指针作为参量。

注释：

如果pthread_mutex_init()的第2个参量为NULL，则使用默认属性。

代码规范6.9 在POSIX Threads接口中如何动态分配互斥体的例子

动态分配互斥体

```
pthread_mutex_t *lock;                     // 声明指向锁的一个指针
lock=(pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL);
   /*
    * 使用该锁的代码
    */
pthread_mutex_destroy(lock);
free(lock);
```

可串行性 显然我们能使用互斥体来实现原子性。得到互斥体m的线程将执行临界区中的代码，直到它释放互斥体。因此在我们的例子中，计数器每次只能由一个线程更新。原子性很重要，是因为它能保证*可串行性*（serializability）。如果执行能确保对应于多个线程的*某种*串行执行，这些线程的并发执行则是*可串行化的*。

正确性问题 解锁一个尚未上锁的互斥体是错误的，同理，加锁一个已经被其他线程上锁的互斥体也是错误的。后者会导致*死锁*，即线程不能向前执行，因为它会一直等待一个永远都不会发生的事件。我们将在稍后的几个小节中讨论死锁和避免死锁的技术。

6.1.3 同步

互斥体已经足够为临界区提供原子性了，但在许多情况中，我们希望一个线程能与其他线程同步它们的行为。例如，考虑经典的约束缓冲区问题，即一个或多个线程将项（item）放置到循环缓冲区中，而其他一些线程将这些项从同一个缓冲区中取出。如图6-1所示，当消费者跟不上生产者的速度，导致缓冲区变满时，我们希望生产者能停止生成数据并等待。同样，当缓冲区为空时，我们希望消费者能等待。

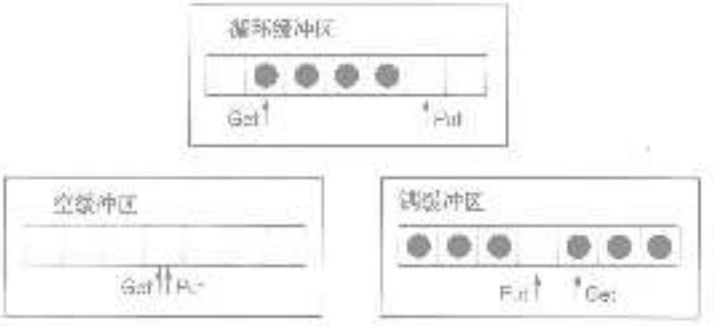


图6-1　使用生产者和消费者的约束缓冲区。Put游标和Get游标分别指示了生产者插入下一项的位置和消费者删除下一项的位置。当缓冲区为空时，消费者必须等待。当缓冲区为满时，生产者必须等待

此种同步可由条件变量（condition variable）支持，它们具有比pthread_join()更加通用的同步形式。参见代码规范6.10和6.11。条件变量允许线程等待，直到一些条件为真。在那一刻，等待线程之一将被随机选中以停止等待。我们可以将这些条件变量想像成一扇门（参见图6-2）。线程等在门口，直到某个条件为真。其他线程打开此门表示该条件已为真。此时，等待者之一将被允许进入门，继续执行。如果一个线程打开门而没有其他线程在等待，则该信号将不起作用。

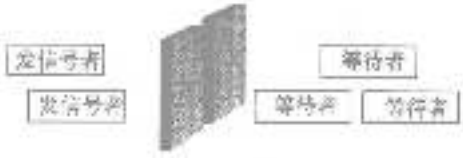


图6-2　条件变量的作用就如门。线程通过调用pthread_cond_wait()在门外等待，通过调用pthread_cond_signal()打开门。当门打开时，等待者之一将被允许通过。当门打开而没有等待者时，该信号将不起作用

代码规范6.10　pthread_cond_wait()。POSIX Threads中用来等待条件变量的例程

```
pthread_cond_wait()

int pthread_cond_wait(
  pthread_cond_t *cond,                    // 要等待的条件
  pthread_mutex_t *mutex);                 // 用来保护该互斥体
int pthread_cond_timedwait(
  pthread_cond_t *cond,
  pthread_mutex_t *mutex,
  const struct timespec *abstime);         // 超时值
```

参量：

- 要等待的条件变量。

• 用来保护对条件变量访问的互斥体。互斥体将在某个线程阻塞前被释放，而且这两个操作是原子性的。稍后当这个线程不阻塞时，互斥体将重新被这个线程获得。

返回值：

如果成功，则为0；否则为<errno.h>中的某个错误码。

代码规范6.11 pthread_cond_signal()。POSIX Threads中用来发条件变量信号的例程

pthread_cond_signal()

```
int pthread_cond_signal(
  pthread_cond_t *cond);                    // 发信号的条件
int pthread_cond_broadcast(
  pthread_cond_t *cond);                    // 发信号的条件
```

参量：

要发信号的条件变量。

返回值：

如果成功，则为0；否则为<errno.h>中的某个错误码。

注释：

• 如果没有线程等待cond，则这些例程不起任何效果。特别是，当稍后要调用pthread_cond_wait()时，不存在对该信号的记忆。

• pthread_cond_signal()例程可能唤醒了不止一个线程，但只有其中之一能最终获得该保护性互斥体。

• pthread_cond_broadcast()例程唤醒了所有例程，但只有一个已唤醒的例程能最终获得该保护性互斥体。

可以使用两个条件变量nonempty和nonfull来解决约束缓冲区问题，如图6-3所示。当然由于有多个线程要更新这些条件变量，我们需要使用互斥体来保护对它们的访问。因此第1行声明了一个互斥体，余下的声明定义了一个缓冲区和它的两个游标，put和get，当它们超过缓冲区边界时将进行环绕，从而构成一个循环的缓冲区。正如我们在图6-1中所见到的，put游标指向下一个空位，而get游标指向下一个要移除的元素。当这两个游标指向同一位置时，缓冲区为空；当这两个游标间有SIZE-1的距离时，缓冲区为满。实际上所谓满的缓冲区还留出了一个空位，这是因为put和get使用模算术来防止溢出，如果允许缓冲区完全填满，我们将无法确定缓冲区到底为满还是为空。

给定这些数据结构后，生产者线程执行insert()例程，它将先获得访问条件变量的互斥体。（图中略去了生产者和消费者线程的代码，假设它们分别反复地调用insert()和remove()例程）。如果缓冲区为满，生产者将等待nonfull条件。稍后当缓冲区非满时，它将被唤醒。如果生产者线程阻塞，它所持有的互斥体必须被释放以避免死锁。因为释放互斥体和阻塞等待线程这两个事件必须原子地出现，因此它们必须通过pthread_cond_wait()执行，所以互斥体将作为一个参数传递给pthread_cond_wait()。当生产者从第14行的等待中返回时，它将继续执行，而保护性互斥体将重新被代表生产者的系统获得。

稍后我们将解释为什么需要第11行的while循环，但现在先假设当生产者执行第16行时，缓冲区是非满的，因此加入一个新项并将put游标推前一格都是安全的。此时，缓冲区不可能为空，因为生产者刚刚插入了一个元素，因此生产者发信号表示缓冲区为非空，并唤醒一个

或多个之前等待空缓冲区条件的消费者。如果没有等待的消费者，则该信号将丢失。最后，消费者释放互斥体并退出例程。同样，消费者线程以一个非常类似的方式执行remove()例程。

```
1   pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
2   pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
3   pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
4   Item buffer[SIZE];
5   int put=0;                          // 指向下一个插入项的缓冲区索引
6   int get=0;                          // 指向下一个移除项的缓冲区索引
7
8   void insert(Item x)                       // 生产者线程
9   {
10    pthread_mutex_lock(&lock);
11    while ((put+1)% SIZE==get               // 当缓冲区为满时
12
13    {
14      pthread_cond_wait(&nonfull, &lock);
15    }
16    buffer[put]=x;
17    put=(put+1)%SIZE;
18    pthread_cond_signal(&nonempty);
19    pthread_mutex_unlock(&lock);
20  }
21
22  Item remove()                             // 消费者线程
23  {
24    Item x;
25    pthread_mutex_lock(&lock);
26    while(put==get)                         // 当缓冲区为空时
27    {
28      pthread_cond_wait(&nonempty, &lock);
29    }
30    x=buffer[get];
31    get=(get+1)%SIZE;
32    pthread_cond_signal(&nonfull);
33    pthread_mutex_unlock(&lock);
34    return x;
35  }
```

图6-3　使用条件变量nonempty和nonfull的约束缓冲区示例

保护条件变量　现在让我们回到约束缓冲器程序第11行的while循环。如果系统中有多个生产者线程，则这个循环将是关键所在，因为pthread_cond_signal()能唤醒多个等待线程[⊖]，但在任意时刻只有一个能获得保护性互斥体。参见代码规范6.10和6.11。因此在发信号时，缓冲区是非满的，但当某个线程获得了互斥体后，缓冲区可能重新变满。在这种情况下，线程应再次调用pthread_cond_wait()。当生产者线程执行第16行代码时，缓冲区必定为非满，因此插入一个新项和将put游标推前一格都将是安全的。

在第18行和第32行，我们看到对pthread_cond_signal()的调用也由锁保护。图6-4中显示的例子解释了为什么这种保护是必要的。等待线程（在这里是消费者），在获得保护性互斥体的同时，发现缓冲区为空，因此它执行了pthread_cond_wait()。如果发信号线程（此处是生产者），没有使用互斥体保护对pthread_cond_signal()的调用，它可能会在等待线程发现缓冲区为空后，马上插入一项到缓冲区中。如果在等待线程执行pthread_cond_wait()调用之前，生产者就发送了缓冲区非空的信号，则该信号将被忽略，导致消费者线程无法意识到缓冲区实际上已非空。

⊖ 这些语义会依赖于实现细节。在某些情况中，确保只有一个等待者会被信号解除阻塞所花费的代价是昂贵的。

在这种情况下，生产者只插入了一个项，而等待线程将会不必要地永远等待下去。

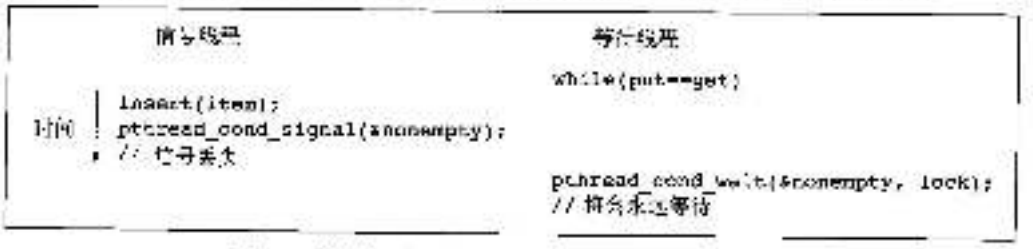


图6-4 举例解释为什么信号线程需要互斥体保护

显然这个问题涉及了缓冲区操作的竞态条件。一种最简易的解决方案是使用相同的互斥体保护对pthread_cond_wait()和pthread_cond_signal()的调用，就如同之前我们在约束缓冲区解决方案所使用的代码中展示的一样。由于insert()和remove()这两个例程都由同一互斥体保护，我们会有与非空缓冲区相关的三个临界区，如图6-5中所示。当等待进程认为缓冲区为空时，绝不会发生信号丢失的情况。

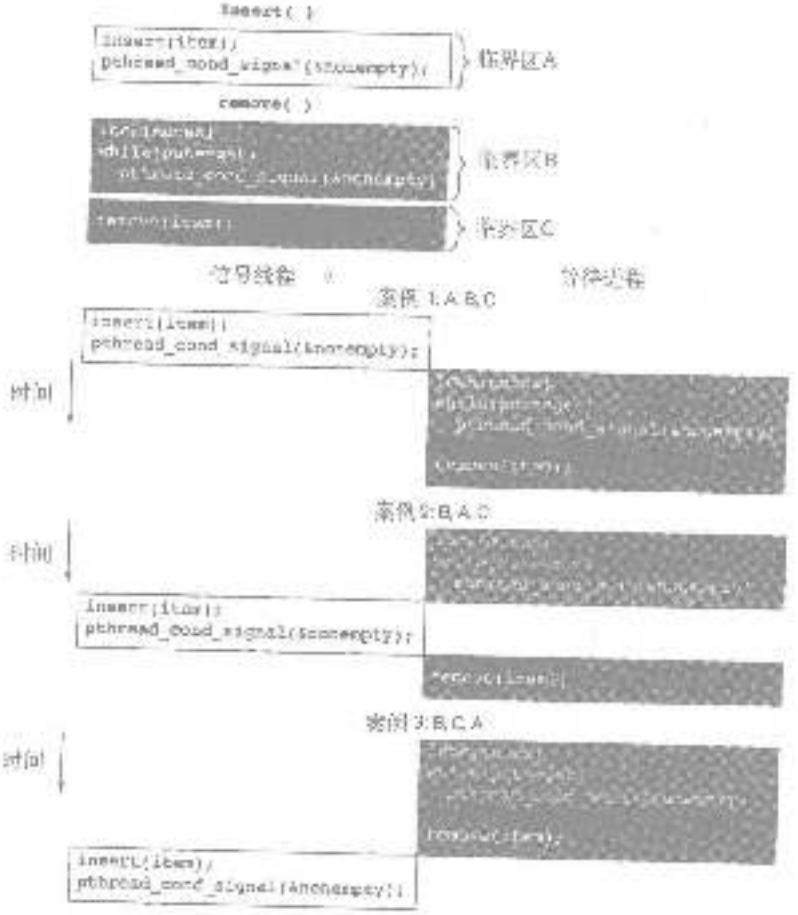


图6-5 信号代码合适的下面可防止竞态条件，通过使用一个互斥体保护三个与非空缓冲区相关的临界区，可以保证A、B和C将会原子地执行，从而避免了图6-4中的问题：由于B必须先于C，因此只有三种方法能使A、B和C交叉，绝不会出现线程执行remove()例程认为缓冲区为空时，insert()例程的信号丢失的情况

我们认为对pthread_cond_signal()的调用一定要由保护等待代码的同一互斥体保护。但是注意到竞态条件不是出自条件变量的信号，而是源自对共享缓冲区的访问。因此，我们可以改为简单地保护任何操作共享缓冲区的代码，这意味着insert()代码，在插入一项到缓冲区之后，但在调用pthread_cond_signal()之前，能立即释放互斥体。这段新的代码不仅合法，而且产生了更好的性能，因为它减小了临界区的大小，从而允许了更大的并发性。

创建和销毁条件变量　条件变量与线程、互斥体一样，能通过静态或动态的方式创建和销毁。在约束缓冲区的例子中，静态条件变量能通过初始化为PTHREAD_ COND_ INITIALIZER而获得默认属性。条件变量也能动态分配，如代码规范6.12中所示。

代码规范6.12　POSIX Threads中用来动态创建和销毁条件变量的例程

动态分配条件变量

```
int pthread_cond_init(
  pthread_cond_t *cond,                    // 条件变量
  const pthread_condattr_t *attr);         // 条件属性
int pthread_cond_destroy(
  pthread_cond_t *cond);                   // 要销毁的条件
```

参量：

如果attr是NULL，则使用默认属性。

返回值：

如果成功，则为0；否则为<errno.h>中的某个错误码。

等待多个条件变量　在某些例子中，一段代码在多个条件未被同时满足之前不能执行。在这种情况下，等待线程必须同时测试所有的条件，如下所示：

```
1  EatJuicyFruit()
2  {
3    pthread_mutex_lock(&lock);
4    while(apples==0||oranges==0)
5    {
6      pthread_cond_wait(&more_apples, &lock);
7      pthread_cond_wait(&more_oranges, &lock);
8    }
9    /*临界区：既吃苹果又吃桔子*/
10   pthread_mutex_unlock(&lock);
11 }
```

相反，以下这段代码依次等待每个条件，最终会因为没有确保两个条件同时为真而出错。即在返回第一个pthread_cond_wait()调用之后，但在返回第二个pthread_cond_wait()调用之前，其他线程有可能移走了苹果，从而导致第一个条件为假。

```
1  EatJuicyFruit()              注意：这段代码是有错误的!
2  {
3    pthread_mutex_lock(&lock);
4    while(apples==0)
5    {
6      pthread_cond_wait(&more_apples, &lock);
7    }
8    while(oranges==0)
9    {
10     pthread_cond_wait(&more_oranges, &lock);
11   }
12
13   /*临界区：既吃苹果又吃桔子*/
```

```
14    pthread_mutex_unlock(&lock);
15  }
```

线程指定数据 线程擅长维护私有数据而非共享数据。例如我们在之前的一些例子中看到，将一个线程的索引传递给起始例程，就可以使线程知道该在数据的哪个部分上工作。索引可用来为每个线程分配一个数组中不同的元素，如下所示：

```
1  ...
2
3  for(i=0; i<t; i++)
4  {
5    err=pthread_create(&tid[i], NULL, start_function, i);
6  }
7
8  void* start_function(void*index)
9  {
10   private_count[index]=0;
11 ...
```

但这样做有一个问题，如果这段代码在某个函数foo()中访问索引，而这个函数又埋藏于其他代码中。在此种情况下，foo()应如何获得索引值？一个解决方案是将索引作为参数传递到每个调用foo()的过程中，包括通过其他过程间接调用foo()的过程。这种解决方案相当繁琐，特别对于那些需要参数但又不直接使用的过程而言。

与此相反，实际上我们真正需要的是在全局范围内对所有代码可用、但对每个线程又有所不同的值。POSIX Threads以*线程指定数据*（thread-specific data）的形式支持了这种概念。它使用了一组*键*，既能在一个进程中被所有线程共享，又能为每个线程映射不同的指针值。（参见图6-6）。

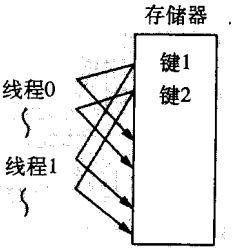


图6-6 POSIX Threads中线程指定数据的举例。线程指定数据通过键访问，它在不同线程中会映射到不同的存储器地址

作为一个特殊例子，POSIX Threads例程的错误码值以线程指定数据方式返回，但这些数据并不使用代码规范6.13至6.17中定义的接口。相反，每个线程都有自己的errno值。

代码规范6.13 这个例子说明了线程指定数据是如何使用的。一旦使用这段代码初始化，任意过程均能访问my_index的值

线程指定数据

```
pthread_key_t *my_index;
#define index(pthread_getspecific(my_index))
main()
{
  ...
  pthread_key_create(&my_index, 0);
  ...
}
void start_routine(int id)
{
  pthread_setspecific(my_index, id);
  ...
}
```

注释：

要避免在一个紧凑的内循环中访问索引，因为每次访问都需要一个过程调用。

代码规范6.14　pthread_key_create()。POSIX Threads中用来为线程指定数据创建键的例程

pthread_key_create

```
int pthread_key_create(
  pthread_key_t*key,                    // 要创建的键
  void(*destructor)(void*));            // 析构函数
```

参量：

- 指向要创建的键的指针。
- 析构函数，NULL表示无析构函数。

返回值：

如果成功，则为0；否则为<errno.h>中的某个错误码。

注释：

要避免在一个紧凑的内循环中访问索引，因为每次访问都需要一个过程调用。

代码规范6.15　pthread_key_delete()。POSIX Threads中用来删除键的例程

pthread_key_delete

```
int pthread_key_delete(
  pthread_key_t*key);                   // 要删除的键
```

参量：

指向要删除的键的指针。

返回值：

如果成功，则为0；否则为<errno.h>中的某个错误码。

注释：

不会调用析构函数。

代码规范6.16　pthread_setspecific()。POSIX Threads中用来设置线程指定数据的值的例程

pthread_setspecific

```
int pthread_setspecific(
  pthread_key_t*key,                    // 要设置的键
  void *value);                         // 要设置的值
```

参量：

- 指向要设置的键的指针。
- 要设置的值。

返回值：

如果成功，则为0；否则为<errno.h>中的某个错误码。

注释：

在键创建之前或删除之后调用pthread_setspecific()都会出错。

代码规范6.17　pthread_getspecific()。POSIX Threads中用来获得某个线程指定数据的值的例程

pthread_getspecific

```
int pthread_getspecific(
  pthread_key_t*key);                   // 映射到值的键
```

参量：

要获得值的键。

返回值：

调用线程所对应的键的值。

注释：

如果线程在键创建之前或删除之后调用pthread_getspecific()，则该例程的行为就是未定义的。

6.1.4 安全性问题

许多类型的错误源自不正确的使用锁和条件变量。例如，我们已经提及的双重加锁问题，即线程试图获得一个它已经拥有的锁。又如，如果线程访问没有上锁的共享变量，或线程获得锁但没有释放它，也会发生问题。一个特别重要的问题是如何避免死锁。本小节将讨论各种不同的避免死锁及其他潜在隐错的方法。

死锁 死锁有四个必要条件。

1. **互斥**：一个资源最多只能分配给一个线程。

2. **持有和等待**：两个线程都持有一些资源，并请求其他资源。

3. **非抢占式**：分配给某个线程的资源只能由该线程自己释放。

4. **循环等待**：每个线程都在等待已经分配给其他线程的资源，且这些线程之间存在着回路。(参见图6-7)。

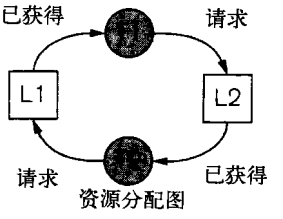


图6-7 死锁举例。线程T1和T2分别持有锁L1和L2，两个线程均试图获得另一个锁，而这是不可能获准的

当然，对于基于线程的程序设计而言，互斥体是一种能导致死锁的资源。有两种处理死锁的常用方法：(1) 防止死锁；(2) 允许死锁发生，但一旦侦测到它们的存在，便破坏死锁。我们主要关注于死锁的避免，因为POSIX Threads没有提供破坏死锁的机制。

锁的层次体系 一个防止死锁的简单方法是在资源分配图中防止回路的出现。我们对锁进行排序，并要求所有线程必须以相同的顺序获得锁，以此防止出现回路。这种方法称为*锁的层次体系*（lock hierarchy）。

锁的层次体系要求程序员预先知道线程需要获得哪个锁。假设在获得锁L1，L3和L7之后，线程发现它还需要获得L2，而这会违反锁的层次体系。一种解决方案是线程先释放锁L3和L7，然后再依次获得锁L2，L3和L7。当然这种对锁层次体系的严格一致性会导致高昂的成本。另一种更好的解决方案是使用pthread_mutex_trylock()尝试获得锁L2（参见代码规范6.7），其结果或是获得了锁，或是马上返回而不被阻塞。如果线程不能获得锁L2，则它必须使用第一种方法。

监视器 使用锁和条件变量都容易产生错误，因为它们都依赖于程序员自身的素质。另一种解决方案是提供语言支持，这将允许编译器强制实现互斥和正确同步。*监视器*（monitor）就是这样的一种语言结构，如图6-8所示。监视器封装了代码和数据，并确保了互斥。尤其是，监视器有一组良定义的入口点，其数据只能由位于监视器内部的代码访问，并且任意时刻都只有一个线程能执行监视器的代码。我们可以用面向对象的语言实现监视器，比如C++，图6-9显示了一个与图6-3略有不同的解决方案。

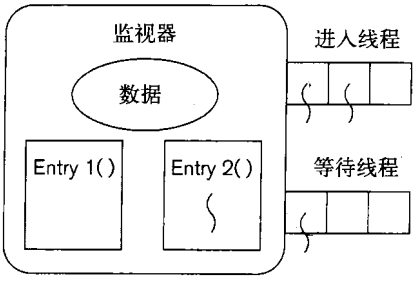


图6-8　监视器提供了一种同步抽象，任意时刻都只有一个线程能访问监视器的数据。其余线程均被阻塞，或等待着进入监视器，或等待来自监视器内的事件

```
1  class BoundedBuffer
2  {                                          //  实现监视器
3    private:
4      pthread_mutex_t lock;                  //  同步变量
5      pthread_cond_t nonempty, nonfull;
6      Item *buffer;                          //  共享数据
7      int in, out;                           //  游标
8      CheckInvariant();
9
10   public:
11     BoundedBuffer(int size);               //  构造函数
12     ~BoundedBuffer();                      //  析构函数
13     void put(Item x);
14     Item get();
15 }
16
17 // 构造函数和析构函数
18 BoundedBuffer::Bounded(int size)
19 {
20   // 初始化同步变量
21   pthread_mutex_init(&lock, NULL);
22   pthread_cond_init(&nonempty, NULL);
23   pthread_cond_init(&nonfull, NULL);
24
25   // 初始化缓冲区
26   buffer=new Item[size];
27   in=out=0;
28 }
29
30 BoundedBuffer::~BoundedBuffer()
31 {
32   pthread_mutex_destroy(&lock);
33   pthread_cond_destroy(&nonempty);
34   pthread_cond_destroy(&nonfull);
35   delete buffer;
36 }
37
38 // 成员函数
39 BoundedBuffer::Put(Item x)
40 {
41   pthread_mutex_lock(&lock);
42   while(in-out==size)                      //  当缓冲区为满时
43   {
44     pthread_cond_wait(&nonfull, &lock);
```

图6-9　用C++实现的监视器

```
45      }
46      buffer[in%size]=x;
47      in++;
48      pthread_cond_signal(&nonempty);
49      pthread_mutex_unlock(&lock);
50    }
51
52    Item BoundedBuffer::Get()
53    {
54      pthread_mutex_lock(&lock);
55      while(in==out)                          // 当缓冲区为空时
56      {
57        pthread_cond_wait(&nonempty, &lock);
58      }
59      x=buffer[out%size];
60      out++;
61      pthread_cond_signal(&nonfull);
62      pthread_mutex_unlock(&lock);
63      return x;
64    }
```

图6-9 （续）

监视器不仅能强制互斥，而且能实现抽象，这就简化了我们推断并发性的方法。尤其是，数量有限的入口点方便了对不变式的保留。不变式（invariant）是指在进入时假设为真，在退出时必须恢复的属性。如图6-10所示，当监视器的锁被持有时，这些不变式可能会被破坏，但它们必须在锁释放前恢复。

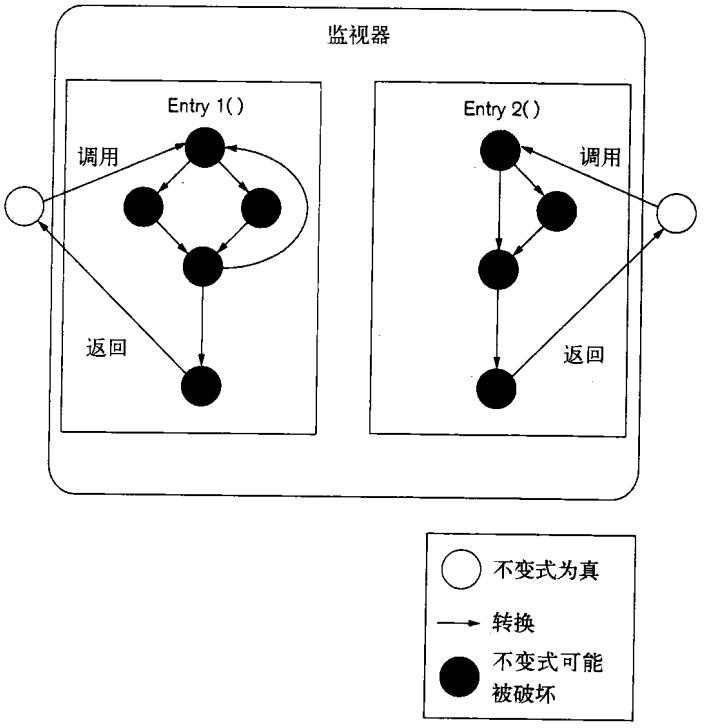


图6-10 监视器和不变式。实心圈表示不变式可能被破坏的程序状态。空心圈表示不变式假设为真的程序状态

例如，在我们的约束缓冲区的例子中，我们有两个不变式：

1. In和Out游标之间的距离必须小于缓冲区的大小。

2. In游标不能在Out游标的左边。（图6-1中，Put箭头不能在Get箭头的左边。）

一旦我们确定了不变式，我们就能编写一个例程用以检查所有的不变式，而且每次进入和退出监视器时都会调用该例程。这种不变式可用作重要的调试工具。例如在图6-11中，程序检查了这些不变式，以此帮助调试该监视器的实现。注意在第20行和第22行都插入了检查。需要第20行的检查是因为第21行调用的pthread_cond_wait()可能隐式地释放了锁，因此这是一个潜在的监视器退出；而需要第22行的检查是因为从pthread_cond_wait()的返回隐式地重新获得了该锁，因此这是一个监视器进入。

```
 1  BoundedBuffer::CheckInvariant()
 2  {
 3    if(in - out>size)                          // 检查不变式(1)
 4    {
 5      return(0);
 6    }
 7    if(in<out)                                 // 检查不变式(2)
 8    {
 9      return(0);
10    }
11    return(1);
12  }
13
14  Item BoundedBuffer::Get()
15  {
16    pthread_mutex_lock(&lock);
17    assert(CheckInvariant());                  // 在每个入口点检查
18    while(in==out)                             // 当缓冲区为空时
19    {
20      assert(CheckInvariant());                // 在每个退出点检查
21      pthread_cond_wait(&nonempty, &lock);
22      assert(CheckInvariant());
23    }
24    x=buffer[out%size];
25    out++;
26    pthread_cond_signal(&nonfull);
27    assert(CheckInvariant());
28    pthread_mutex_unlock(&lock);
29    return x;
30  }
```

图6-11　用来检查图6-9约束缓冲区程序中不变式的程序

再入监视器　虽然监视器有助于强制实现上锁的规定，但它无法解决所有的并发问题。例如，如果监视器内的过程试图通过调用进入过程重新进入监视器，就会发生死锁。为了避免这种问题，过程首先必须恢复所有的不变式，并释放监视器的锁，然后再尝试重新进入监视器。当然，这种结构就意味着原子性的丧失。如果监视器内的过程试图通过调用其他外部过程间接地重新进入监视器，同样的问题也会发生。因此监视器过程调用外部例程时必须慎之又慎。

那些需要太长时间或等待外部事件发生的监视器函数会妨碍到其他线程进入监视器。为了避免这种问题，需要长时间运行的函数通常被重写为等待某个条件，以此释放锁，增加并行性。与重新进入的例程相同，此类函数需要在释放锁之前恢复不变式。

6.1.5　性能问题

在第3章中，我们看到线程间的相关性会限制并行性。由于锁会在线程间动态地强加相关

性，因此锁的粒度大小对并行性有很大影响。其中一个极端是，最粗粒度的锁方案是为所有共享变量使用单个锁。这种方案很简单，但当存在共享时，将严重制约并发性。而另一个极端是，我们为数据结构中每个细粒度的子结构都配备锁。例如，在统计3的个数例子中，我们可以用不同的锁来保护聚合树中的每个结点。这种方案虽然增加了并发性，但同时也增加了获得和释放锁的时延，甚至在没有竞争数据结构时亦会如此。而且由于每个线程必须要获得多个锁才能在多个数据结构上进行操作，出现死锁的几率就会增大。作为折中，我们可以为每棵聚合树配备一个锁。一般而言，需要在并行性与开销之间折中粒度的粗细。

读者和写者举例：粒度问题 如同锁有不同粒度，使用条件变量也有不同粒度。考虑有一个资源能被多个读者所共享，或只被一个写者排他地访问（参见图6-12）。为了协调对这种

```
 1  int readers;                                  // 负值=>活跃的写者
 2  pthread_mutex_t lock;
 3  pthread_cond_t rBusy, wBusy;                  // 为读者和写者分别使用独立的条件变量
 4
 5  AcquireExclusive()
 6  {
 7    pthread_mutex_lock(&lock);
 8    while(readers !=0)
 9    {
10      pthread_cond_wait(&wBusy, &lock);
11    }
12    readers=-1;
13    pthread_mutex_unlock(&lock);
14  }
15
16  AcquireShared()
17  {
18    pthread_mutex_lock(&lock);
19
20    while(readers<0)
21    {
22      pthread_cond_wait(&rBusy, &lock);
23    }
24    readers++;
25    pthread_mutex_unlock(&lock);
26  }
27
28  ReleaseExclusive()
29  {
30    pthread_mutex_lock(&lock);
31    readers=0;
32    pthread_cond_broadcast(&rBusy);             // 只唤醒读者
33    pthread_mutex_unlock(&lock);
34  }
35
36  ReleaseShared(
37  {
38    int doSignal;
39
40    pthread_mutex_lock(&lock);
41    readers--;
42    doSignal=(readers==0)
43    pthread_mutex_unlock(&lock);
44    if(doSignal)                                // 信号的执行在临界区之外
45    {
46      pthread_cond_signal(&wBusy);              // 唤醒一个写者
47    }
48  }
```

图6-12 支持多读者单写者的例程

资源的访问，我们实现了读者和写者都能调用的四个例程－AcquireExclusive()，RelaeaseExculsive()，AcquireShared()，ReleaseShared()。每个例程均由一个互斥体保护，总共使用了两个条件变量。为了在排他模式中获得资源，线程将等待wBusy条件变量，以此确保没有读者还能访问该资源。当最后一个读者以共享模式完成访问资源时，它就发送信号给wBusy条件变量以允许一个写者进行操作。类似地，当一个写者以排他模式完成访问资源时，它就发送信号给rBusy条件变量以允许任意数量的读者访问该资源。在访问共享资源之前，这些线程一直在等待rBusy条件变量。

关于这段代码有两点值得注意：

第一，这段代码使用了双条件变量，这很自然地让人疑心是否单条件变量就足够。实际上，单条件变量是可行的，如图6-13所示，而且代码在功能上也是正确的。但不幸的是，使

```
1  int readers;                          // 负值=>活跃的写者
2  pthread_mutex_t lock;
3  pthread_cond_t busy;                  // 使用一个条件变量来表示数据是否忙
4
5  AcquireExclusive()
6  {
7    pthread_mutex_lock(&lock);          // 这段伪码存在伪唤醒问题!!!
8    while(readers!=0)
9    {
10     pthread_cond_wait(&busy, &lock);
11   }
12   readers=-1;
13   pthread_mutex_unlock(&lock);
14 }
15
16 AcquireShared()
17 {
18   pthread_mutex_lock(&lock);
19   while(readers<0)
20   {
21     pthread_cond_wait(&busy, &lock);
22   }
23   readers++;
24   pthread_mutex_unlock(&lock);
25 }
26
27 ReleaseExclusive()
28 {
29   pthread_mutex_lock(&lock);
30   readers=0;
31   pthread_cond_broadcast(&busy);
32   pthread_mutex_unlock(&lock);
33 }
34
35 ReleaseShared(
36 {
37   pthread_mutex_lock(&lock);
38   readers--;
39   if(readers==0)
40   {
41     pthread_cond_signal(&busy);
42   }
43   pthread_mutex_unlock(&lock);
44 }
```

图6-13　基于单条件变量，但存在伪唤醒问题的支持多读者单写者的例程

用单条件变量后，代码会出现伪唤醒（spurious wakeup）的情况，即写者会在被唤醒之后马上重新进入睡眠。尤其是，当调用ReleaseExclusive()时，读者和写者都被发送了信号，因此只要有一个读者仍在等待条件变量，写者就会遭受到伪唤醒的困扰。而我们之前的解决方案用双条件变量来避免伪唤醒，即只要对两种类型的访问都有需求，它就将强制轮换排他访问和共享访问。

第二，ReleaseShared()例程在临界区外发送信号给wBusy条件变量，能避免锁的伪冲突（spurious lock conflict）问题，即一个线程被信号唤醒，执行了一些指令，然而当它试图获得锁时马上就会被阻塞。另一种情况，若ReleaseShared()例程在临界区内发送信号，如下段代码所示，则任何醒着的写者都会因试图获得锁几乎立即阻塞。

```
35  ReleaseShared(
36  {
37    pthread_mutex_lock(&lock);
38    readers--;
39    if(readers==0)
40    {
41      pthread_cond_signal(&wBusy); // 在临界区内唤醒写者
42    }                              // the critical section
43    pthread_mutex_unlock(&lock);
44  }
```

将信号发送移到临界区之外的决定是一个折中。它减少了锁的伪冲突，但ReleaseShared()例程在唤醒等待的写者之前，会允许新的读者进入临界区，而这将导致读者使写者饥饿，尽管这个几率比使用单条件变量时要小的多。

线程的调度 到目前为止，我们在讨论线程时并没有考虑它们与操作系统的交互，但它们的行为却依赖于它们是如何被调度的。为了理解这个问题，我们需要先定义内核线程（kernel thread），这是内核或操作系统中调度的最小单元。我们在本章中所讨论的线程可以以不同的方式映射到内核线程，以下是两种主要的方式：

1. 我们可以将多个线程映射到单个内核线程。这就意味着操作系统并不知道有这些线程存在。它们有时也被称为用户级线程，会比较高效，因为它们的创建和销毁都无需操作系统干预。但会有一个问题，即如果其中某个线程阻塞，或许是在等待I/O操作，就会导致整个内核线程阻塞，映射到该内核线程的其他线程都将无法执行。这种线程在Pthreads中被称为非约束线程（unbound thread）。

2. 另一种方式是将每个线程映射到各自的内核线程。在这种情况中，操作系统知道所有的线程。因此当某个线程阻塞时，另一个线程会被调度到它的位置开始执行。这种线程在Pthreads中被称为约束线程（bound thread）。如果使用线程是为了增加性能，通常会使用约束线程。

在第一种方式中，各线程在单进程内竞争资源，而在第二种方式中，各线程竞争操作系统释放的少量系统资源。在POSIX Threads中，可通过设置一个称为范围竞争属性的线程属性来指定调度方式（参见代码规范6.18）。

代码规范6.18 POSIX Threads中用来设置线程调度属性的例程

线程调度属性

```
int pthread_attr_setscope(
  pthread_attr_t *attr,
  int contentionscope);
```

参量：

- 指向要设置的属性的指针。
- 非约束线程设置为PTHREAD_SCOPE_PROCESS，约束线程设置为PTHREAD_SCOPE_SYSTEM。

返回值：

如果成功，则为0；否则为<errno.h>中的某个错误码。

注释：

线程调度通常是操作系统特定的，有些操作系统不能同时支持这两种调度方式。

优先级倒置　还有许多的调度细节尚未在本书中提及。例如，线程能使用不同的调度策略，而且线程可以有不同的优先级。在本小节中，我们将关注一个常见的调度问题，它将引起所谓的*优先级倒置*（priority inversion）。

当低优先级线程持有了高优先级线程想要获得的锁时，就会发生优先级倒置。由于高优先级线程必须在低优先级线程持有互斥体时阻塞，所以实际上高优先级线程比低优先级线程的优先级要低。如果只有2个线程，则这种情况只是暂时的，假设低优先级线程终会释放锁。而且如果临界区较小，则该问题的影响就会被最小化。但当有第3个线程被创建时，而且它的优先级处在低优先级与高优先级之间，则这个问题将变得很棘手。这个中优先级线程会抢占低优先级线程，阻止低优先级线程执行并释放锁。因此只要中优先级线程在执行，高优先级线程只能一直阻塞。

这个问题有两个常用的解决方案。一个方案是使用*优先级天花板*（priority ceiling），即设定最高可能的线程优先级。任何获得了锁的线程都将以最高可能的优先级执行，这就能防止某个中优先级线程的抢占。另一个方案是使用*优先级继承*（priority inheritance）。一个线程获得了互斥体后，它将暂时继承被阻塞以等待互斥体的最高优先级线程的优先级，当然除非这样做反而会降低它原有的优先级。当该线程释放互斥体后，它的优先级就恢复到原有的优先级。因此任何获得互斥体的线程都将继承被阻塞的最高优先级线程的优先级，因为它在代表被阻塞线程有效地继续执行。

成组调度（Gang Scheduling）　由于在系统中进程数通常比可用处理器数要多许多，因此现代操作系统必须通过对处理器时间分片来共享物理资源。也即是说，操作系统必须备有调度进程到处理器的策略。当调度两个需要交互的进程到不同时间片中执行时，不佳的调度对性能是有害的。尤其是，如果进程p与另一个进程需要进行交互，但没有被调度到相同时间片中执行，则进程p将白白浪费它的整个时间片来等待一个永远不会发生的事件。解决该问题的方法是采用称为成组调度或协同调度（co-scheduling）的策略，其关键在于P个需要交互的进程必须被同时调度到P个处理器上。

在大型并行计算机上，成组调度是大多数应用程序员不需要关心的系统问题。但相关问题会更普遍的存在于线程调度中：即如果并行程序使用$P+k$个需要交互的线程在P个处理器上执行，若$k>0$，则性能通常会受影响。我们的结论是，编写代码时最好是针对数量固定但不指定的处理器，如第4章中可扩展并行方法所建议。

6.1.6 案例研究1：连续过度松弛

第5章中我们讨论了几种将作业分配到线程的不同方法，而本章中看到了如何使用线程表

现并行性，考虑创建具体的并行计算时，会应用到这两个概念。我们将重点关注于那些影响性能的问题，而将具体的程序设计细节作为习题留给读者完成。

我们的问题是计算一个二维的连续过渡松弛（Successive Over-Relaxation，SOR）的程序，它能用来求解3维形式的偏微分方程组，例如可用来计算液体流的Navier-Stokes方程。我们的计算从有n个值的二维数组开始。在每次迭代中，计算将数组的每个值替换为周围4个最近邻居的平均值。那些在数组边界上的值将使用预先确定的常数值，称为边界值，作为它们所缺失的邻居值。我们假定左边界的边界值都为1，其余的边界值都为0，而二维数组内的每个值均初始化为0。

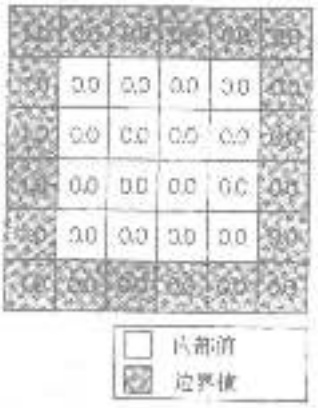


图6-14 一维松弛（在每次迭代中）将所有内部值替换为其周围4个最近邻居的平均值

处理边界值最简单的方法是将它们分配到二维数组中，并用它们的常数值初始化它们，得到如图6-14中所示的情况。

为了形式化描述我们的并行解决方案，需要检查计算中出现的数据相关性。假定一个数组包含了当前的状态值（old），而另一个数组将被填充下次迭代的状态值（new）。我们将在每次迭代的最后，交换这两个数组的指针完成更新。在每次迭代中，old数组的每个元素被读取了4次，因为它是其他4个数组元素的邻居。因此old数组只会引入输入的相关性。在每次迭代中，new数组的每个元素都会被修改。因此就该数组元素而言，在单次迭代中并不存在真正的相关性，因为读取和写入所针对的是不同数据；但在跨迭代的计算中存在着真正的相关性，因为在所有new数组的值更新之前，我们不能交换old数组和new数组的指针。

在了解该案例及其数据相关性之后，我们现在来问一个简单的问题，"应当静态还是动态地分配工作？"我们总是能使用工作队列动态分配工作，但这种方法会有两个缺点。首先，必须负担访问工作队列的开销。其次，可能会看到减少的空间局部性，这是当为每个处理器有自己的L1 cache，在不同迭代中可能会分配到数组的不同部分。由于工作相当的规则（即数据结构是一个二维数组，每个数组元素所需的工作量基本上是常数，且不会随时间发生变化），且由于工作能被完美的划分，因此在该案例中，采用静态分配更为合理。

由于工作是规则且静态的，因此使用尽可能少的线程是有意义的，因为这样能最小化线程开销。在程序初始时，我们将创建t个线程，并大致上分配1/t的工作给每个线程。在计算完成之前，将不会销毁这些线程。还将假设可以设置或匹配可用的处理器数，从而最大化计算的并行性。

在确定以静态分配的方式将工作分配给线程后，我们通过在每次迭代结束前放置一个障碍同步，来确保能保护跨迭代的相关性。由于这些障碍能保护所有的程序相关性，因此也就无需再使用任何其他的锁或条件变量了。

那么，如何确切地把工作分配到每个线程？正如第5章中所述，可以使用块分解来分配大块的工作，或是使用循环分解来交叉分配工作。由于所求解的问题每次迭代都会在数组的所有元素上迭代，而负载平衡取决于分配到各个线程的工作量，因此循环分解的负载平衡优势是最小的。相反，块分解有改进局部性的优点，因为对于给定数量的数组元素，循环分解比块分解需要更多的邻居，因而也就需要访问更多的数据。

最后，应当使用1维还是二维的块分解？为了回答这个问题，必须认识到虽然在每次迭代中并没有真正的相关性，但仍会存在假共享的可能性。而使用垂直的1维块分解策略能最小化该问题，图6-15中展示了该策略在两个线程可能存在假共享时是如何最小化假共享数据点的。实际上，若使用所建议的块分解方法，则假共享能通过扩展左边或右边的边界值来消除。该程序如图6-16所示。

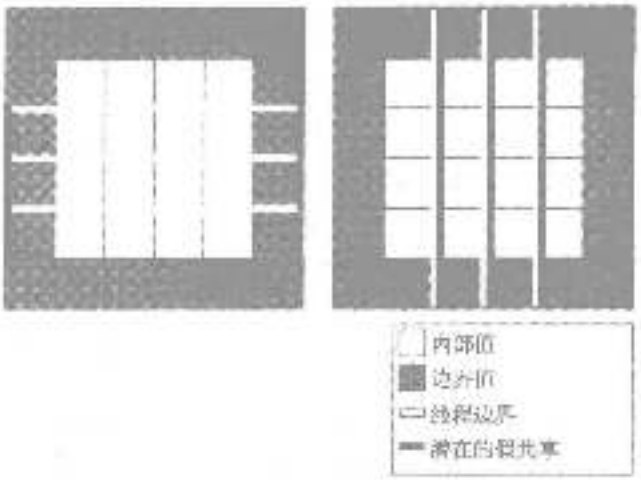


图6-15　(左)如果我们执行优先的存储器分配，则垂直的一维块分解能最小化潜在的假共享。(右)水平的一维块分解增加了潜在的假共享。二维块分解(未在图中显示)将会进一步地增潜在的假共享，而循环分解甚至会更糟糕

```
 1  #include <pthread.h>
 2  #include <limits.h>
 3  #define MAXTHREADS 16          /* 线程的最大数量*/
 4  void* thread_main(void *);
 5  void InitializeData();
 6  void barrier();
 7
 8  pthread_mutex_t update_lock;
 9  pthread_mutex_t barrier_lock; /* 障碍的互斥体 */
10  pthread_cond_t all_here;       /* 障碍的条件变量 */
11  int count=0;                   /* 障碍的计数器 */
12
13  int n, t, threshold, rowsPerThread;
14  double myDelta;
15  double **val, **new;
16  double delta=0.0;
17
18  /*
19    命令行参量：网格大小、线程数、阈值
20  */
21
22  int main(int argc, char *argv[])
23  {
24   /*线程id和属性 */
25    pthread_t tid[MAXTHREADS];
26    pthread_attr_t attr;
27    int i, j;
```

图6-16　用POSIX Threads编写的二维过度连续松弛程序

```
28
29      /* 设置全局的线程属性 */
30      pthread_attr_init(&attr);
31      pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
32
33      /* 初始化互斥体和条件变量 */
34      pthread_mutex_init(&update_lock, NULL);
35      pthread_mutex_init(&barrier_lock, NULL);
36      pthread_cond_init(&all_here, NULL);
37
38      /* 读入命令行参量 */
39      n=atoi(argv[1]);
40      t=atoi(argv[2]);
41      threshold=atof(argv[3]);
42      rowsPerThread=n/t;
43      InitializeData();
44
45      for(i=0; i<t; i++)
46      {
47        pthread_create(&tid[i], &attr, thread_main, (void *) i);
48      }
49      for(i=0; i<t; i++)
50      {
51        pthread_join(tid[i], NULL);
52      }
53
54      printf("maximum difference:  %e\n", delta);
55    }
56
57    void* thread_main(void *arg)
58    {
59      int id=(int) arg;
60      double average;
61      double **myVal, **myNew;
62      double **temp;
63      int i, j;
64      int start;
65
66    /* 确定该线程拥有的第一行 */
67      start=id*rowsPerThread+1;
68      myVal=val;
69      myNew=new;
70
71      do
72      {
73        myDelta=0.0;
74        if(id==0)
75        {
76          delta=0.0;          /* 重置delta的共享值 */
77        }
78        barrier();
79
80      /* 更新每一个点 */
81        for(i=start; i<start+rowsPerThread; i++)
82        {
83          for(j=1; j<n+1; j++)
84          {
85            average=(myVal[i-1][j] + myVal[i][j+1] +
86                     myVal[i+1][j] + myVal[i][j-1])/4;
```

图6-16 （续）

```
87            myDelta=Max(myDelta, Abs(average-myVal[i][j]));
88            myNew[i][j]=average;
89          }
90        }
91        temp=myNew;                    /* 为下次迭代做准备 */
92        myNew=myVal;
93        myVal=temp;
94
95        pthread_mutex_lock(&update_lock);
96        if(myDelta>delta)
97        {
98          delta=myDelta;               /* 更新delta */
99        }
100       pthread_mutex_unlock(&update_lock);
101
102       barrier();
103     } while(delta < threshold);
104   }
105
106   void InitializeData()
107   {
108     int i, j;
109
110     new=(double **) malloc((n+2)*sizeof(float *));
111     val=(double **) malloc((n+2)*sizeof(float *));
112     for(i=0; i<n+2; i++)
113     {
114       new[i]=(double *) malloc((n+2)*sizeof(float));
115       val[i]=(double *) malloc((n+2)*sizeof(float));
116     }
117
118     /* 初始化为0.0, 除了沿左边界的设置为1.0 */
119     for(i=0; i<n+2; i++)
120     {
121       val[i][0]=1.0;
122       new[i][0]=1.0;
123     }
124     for(i=0; i<n+2; i++)
125     {
126       for(j=1; j<n+2; j++)
127       {
128         val[i][j]=0.0;
129         new[i][j]=0.0;
130       }
131     }
132   }
133
134   void barrier()
135   {
136     pthread_mutex_lock(&barrier_lock);
137     count++;
138     if(count==t)
139     {
140       count=0;
141       pthread_cond_broadcast(&all_here);
142     }
143     else
144     {
145       pthread_cond_wait(&all_here, &barrier_lock);
146     }
147     pthread_mutex_unlock(&barrier_lock);
148   }
```

图6-16　(续)

6.1.7 案例研究2：重叠同步与计算

我们曾在第4章中提到，将长时延的操作与独立计算重叠起来通常会很有用。例如，在图6-17中,线程0在线程1之前到达障栅，因此对于线程0而言，做些有用的工作总比单纯的闲等要好的多。为了利用此种机会，我们通常需要创建分阶段（split-phase）操作，它将一个操作分为2个阶段：开始阶段和完成阶段，如图6-18中所示。

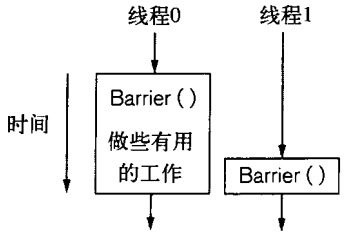


图6-17 在等待某个长时延的操作完成时，做些有用的工作通常是有好处的

```
// 启动同步工作
barrier.arrived();

// 做些有用的工作

// 完成同步
barrier.wait();
```

图6-18 分阶段障栅允许线程在等待其他线程到达障栅时做些有用的工作

让我们通过一个具体实例来了解分阶段操作的作用。考虑1维连续过度松弛的程序，其中我们的数组有n+2个值：n个内部值和2个边界值。与我们之前研究的案例一样，在每次迭代中，计算都会将所有内部值替换成它们左右2个最近邻居的平均值，如图6-19中所示。计算1维连续过度松弛的代码使用了单阶段障栅，如图6-20所示，此处我们假设有t个线程，每个负责计算n / t个值的过度松弛。使用了分阶段障栅后，主例程将做如图6-21中所示的修改。

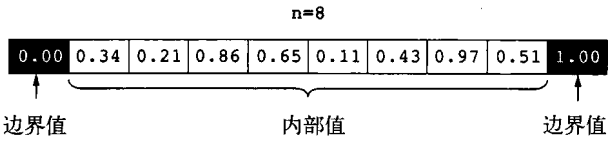


图6-19 一维过度松弛（每次迭代）将所有内部值替换成其左右2个最近邻居的平均值

```
 1  double *val, *new;                    // 持有n个值
 2  int n;                                // 内部值的数量
 3  int t;                                // 线程数
 4  int iterations;                       // 所要执行的迭代数
 5
 6  void* thread_main(void *arg)
 7  {
 8    int index=(int) arg;
 9    double *myVal=val;
10    double *myNew=new;
11    int n_per_thread=n/t;
12    int start=index*n_per_thread+1;
13
14    for(int i=0; i<iterations, i++)
15    {
16      // 更新值
17      for(int j=start; j<start+n_per_thread; j++)
18      {
19        myNew[j]=(myVal[j-1]+myVal[j+1])/2.0;
20      }
21      swap(myNew, myVal);
22      barrier();                        // 同步
23    }
24  }
```

图6-20 使用单阶段障栅的1维连续过度松弛程序

```
 1  double *val, *new;                    // 持有n个值
 2  int n;                                // 内部值的数量
 3  int t;                                // 线程数
 4  int iterations;                       // 所要执行的迭代数
 5
 6  void* thread_main(void *arg)
 7  {
 8    int index=(int) arg;
 9    double *myVal=val;
10    double *myNew=new;
11    int n_per_thread=n/t;
12    int start=index*n_per_thread+1;
13
14    for(int i=0; i<iterations, i++)
15    {
16      // 更新局部边界值
17      int j=start;
18      myNew[j]=(myVal[j-1]+myVal[j+1])/2.0;
19      j=start+n_pre_thread -1;
20      myNew[j]=(myVal[j-1]+myVal[j+1])/2.0;
21
22      // 开始障栅
23      barrier.arrived();
24
25      // 更新局部内部值
26      for(j=start+1; j<start+n_per_thread-1; j++)
27      {
28        myNew[j]=(myVal[j-1]+myVal[j+1])/2.0;
29      }
30      swap(myNew, myVal);
31
32      // 结束障栅
33      barrier.wait();
34    }
35  }
```

图6-21　使用分阶段障栅的1维连续过度松弛程序

实现分阶段障栅的代码似乎已经足够直截了当。如图6-22中所示，我们实现了一个Barrier类来保存已到达该障栅的线程数量的计数器。为了启动同步，每个线程调用arrived()例程，它会递增计数器。最后一个到达障栅的线程（参见第32行）会发送信号给所有的等待者，将它们唤醒并继续执行。最后一个到达的线程还会把计数器置为0，以便为下次使用障栅做好准备。为了结束该同步，wait()例程将检查计数器是否为非0。如果为非0，它将等待最后一个线程的到达。当然，锁可以用来提供互斥，而条件变量可以用来实现同步。

但遗憾的是，图6-22中提供的代码无法正确工作！尤其，如果我们考虑2个线程和2次迭代的执行，如图6-23中所示。一开始，计数器为0。线程0的到达将递增计数器的值到1，线程1的到达应递增计数器的值为2。由于线程1是最后一个达到障栅的线程，它会重置计数器为0，然后唤醒所有等待中的线程，当然此处没有等待的线程。问题就出在当线程1先于线程0到达，并在线程0调用第1次迭代的wait()之前，执行其下一次迭代，即再次调用arrive()和wait()。在这个例子中，线程1将递增计数器到1，而当线程0到达它的等待处时，它将被阻塞。此时，线程0被阻塞以等待第1次迭代中的障栅完成，而线程1被阻塞以等待第2次迭代中的障栅完成，这最终导致了死锁。当然，其实第1个障栅已经完成，但线程0却并不知道这个重要情况。

```
1  class Barrier
2  {
3    int nThreads;                    // 线程数
4    int count;                       // 参与的线程数
5    pthread_mutex_t lock;
6    pthread_cond_t all_here;
7    public:
8      Barrier(int t);
9      ~Barrier(void);
10     void arrived(void);            // 启动障栅
11     int done(void);                // 检查是否完成
12     void wait(void);               // 等待完成
13 }
14
15 int Barrier::done(void)
16 {
17   int rval;
18   pthread_mutex_lock(&lock);
19
20   rval=!count;                     // 如果计数为0，则完成
21
22   pthread_mutex_unlock(&unlock);
23   return rval;
24 }
25
26 void Barrier::arrived(void)
27 {
28   pthread_mutex_lock(&lock);
29   count++                          // 另一个线程到达
30
31   // 如果最后一个线程到达，则唤醒所有的等待者
32   if(count==nThreads)
33   {
34     count=0;
35     pthread_cond_broadcast(&all_here);
36   }
37
38   pthread_mutex_unlock(&lock);
39 }
40
41 void Barrier::wait(void)
42 {
43   pthread_mutex_lock(&lock);
44
45   // 如果尚未完成，则等待
46   if(count !=0)
47   {
48     pthread_cond_wait(&all_here, &lock);
49   }
50
51   pthread_mutex_lock(&lock);
52 }
```

图6-22 最初的分阶段障栅实现，会维护已到达线程数量的计数器

当然，我们似乎相当的不走运，使线程0比线程1执行的要慢。但由于我们的障栅需要在所有情况下都能工作，所以必须处理这个竞态条件。

会发生图6-23中的问题是因为线程0所注视的计数器状态是由障栅的错误调用造成的。因此解决方案就是追踪障栅的当前阶段。具体地，arrived()方法将返回一个阶段编号，然后传递

给done()和wait()方法。正确的代码如图6-24中所示。由于障栅例程的接口发生了改变，因此我们需要修改之前的松弛代码，如图6-25中所示。

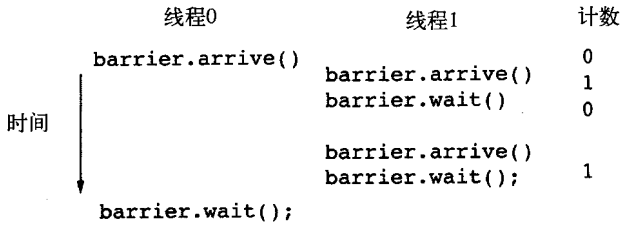


图6-23　分阶段障栅最初实现中的死锁；线程0和线程1各自在等待障栅的不同实例

```
 1  class Barrier
 2  {
 3    int nThreads;                    // 线程数
 4    int count;                       // 参与的线程数
 5    int phase;                       // 该障栅的阶段编号
 6    pthread_mutex_t lock;
 7    pthread_cond_t all_here;
 8    public:
 9      Barrier(int t);
10      ~Barrier(void);
11      void arrived(void);            // 启动障栅
12      int done(int p);               // 检查阶段P是否完成
13      void wait(int p);              // 等待阶段P完成
14  }
15
16  int Barrier::done(int p)
17  {
18    int rval;
19    pthread_mutex_lock(&lock);
20
21    rval=(phase!=p);                 // 如果阶段编号改变，则完成
22
23    pthread_mutex_unlock(&lock);
24    return rval;
25  }
26
27  int Barrier::arrived(void)
28  {
29    int p;
30    pthread_mutex_lock(&lock);
31
32    p=phase;                         // 获得阶段编号
33    count++                          // 另一个线程到达
34
35    // 如果最后一个线程到达，则唤醒所有的等待者，进入下一个阶段
36    if(count==nThreads)
37    {
38      count=0;
39      pthread_cond_broadcast(&all_here);
40      phase=1 - phase;
41    }
42
43    pthread_mutex_unlock(&lock);
44    return p;
```

图6-24　正确的障栅实现，它维护正确阶段

```
45  }
46
47  void Barrier::wait(int p)
48  {
49    pthread_mutex_lock(&lock);
50
51    // 如果尚未完成，则等待
52    while(p==phase)
53    {
54      pthread_cond_wait(&all_here, &lock);
55    }
56
57    pthread_mutex_unlock(&lock);
58  }
```

图6-24 （续）

```
1  double *val, *new;                    // 持有n个值
2  int n;                                // 内部值的数量
3  int t;                                // 线程数
4  int iterations                        // 所需执行的迭代数
5
6  thread_main(int index)
7  {
8    int n_per_thread=n/t;
9    int start=index*n_per_thread+1;
10   int phase;
11
12   for(int i=0; i<iterations; i++)
13   {
14     // 更新局部边界值
15     int j=start;
16     new[j]=(val[j-1]+val[j+1])/2.0;
17     j=start+n_per_thread -1;
18     new[j]=(val[j-1]+val[j+1])/2.0;
19
20     // 启动障栅
21     phase=barrier.arrived();
22
23     // 更新局部内部值
24     for(j=start+1; j<start+n_per_thread-1; j++)
25     {
26       new[j]=(val[j-1]+val[j+1])/2.0;    // 计算平均值
27     }
28     swap(new, val);
29
30     // 完成障栅
31     barrier.wait(phase);
32   }
33 }
```

图6-25 图6-19中一维过度松弛使用分阶段障栅代码后的程序

使用了新的障栅实现之后，图6-23中的情况不会再产生死锁。如图6-26中所示，线程0对wait(0)的显式调用是为了等待第1次障栅调用的完成，因此当它执行第50行的wait()例程时，它将退出while循环，而且永远不会再调用pthread_cond_wait()。这样就避免了死锁。

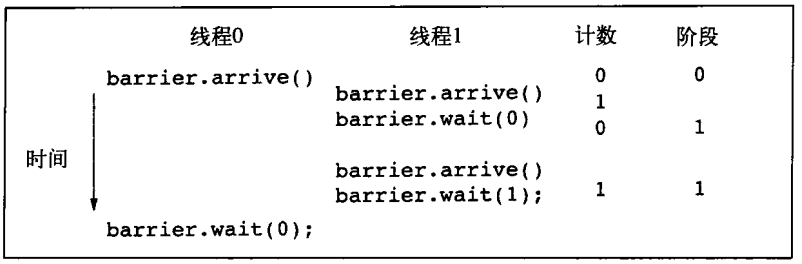


图6-26　使用新的分阶段障栅后，就不会发生死锁

6.1.8　案例研究3：多核芯片上的流计算

为了理解针对多核芯片与针对更加传统的对称对处理器（SMP）的解决方案之间有何不同，我们考虑设计一个并行程序来压缩一部电影，而且关注于确定适当的并行粒度。假设处理是计算密集型的，而且需要高带宽。此外还假设电影由一串帧简单地组成，其中每帧都是一个二维图像。

下面罗列了所有可能的解决方案：

为每个进程分配自有的帧。通过使每个进程在单独的帧上操作，结束后将压缩的帧写回到存储器中，我们能以较大的粒度并行化这个问题。该解决方案很简单，不需要在进程间进行通信。但在所有可能的解决方案之中，该方案需要最大的存储器带宽，特别是芯片上的cache，而且它不并行化实际的压缩过程，而这往往相当需要CPU密集计算。

为每个进程分配单帧的部分。例如，每个进程压缩1/P帧。该解决方案使用所有的进程集体压缩每个帧，因此它只需要较低的存储器带宽。但是依据压缩算法，该方案可能需要在进程间进行通信。

让多个进程在单帧的各部分上协作。我们能够使用更细粒度的方法，即让所有的进程在单帧的某一部分上协作。例如，可以载入1/k帧到片上cache，然后让所有P个进程协同压缩该帧的一部分。当然，这个方法是否可行主要取决于压缩是否能使用多个进程。例如如果算法包含了比如4个阶段，它就有可能采用任务并行的方法，使得每个进程执行4级流水线中的1级。

最佳的选择不仅取决于计算带宽和计算需求这两者的具体细节，也取决于目标硬件的细节。如果压缩是计算约束的，则更细粒度的方法会较有吸引力，尤其是如果多核芯片有较低的片上通信时延。另一方面，如果压缩是存储器约束的，则多核芯片可能没有足够的带宽来支持第一种方案，但它在SMP上却可能很容易实现，因为每个处理器都有到主存储器的自己的接口，以及自己的片上cache。第三种策略能同时最小化总带宽和峰值带宽。

6.2　Java Threads

Java或许是所有能显式支持并行的程序设计语言中使用最为广泛的。本节将着重介绍Java Threads的主要特征。这里假设读者已经仔细阅读了之前POSIX Threads一节，并且对Java和基本的面向对象概念有一定了解。

Java对线程提供了非常有趣的支持。在它的核心，提供了与POSIX Threads相同的概念特性，但由于它面向对象的设定，也能支持更加易用的更高级抽象。与此同时，还能支持可用

来构建新型并发数据结构的低级特性。

在Java中线程是Thread类的实例，有运行和同步线程的方法（其中一些最重要的方法请参见代码规范6.19）。因此Java程序员可以通过使用专门形式的线程扩展Thread类来使用线程，但这种方式是有局限性的，因为Java只支持单继承，所以一个类扩展了Thread类之后就不能再扩展其他类了。

代码规范6.19 Java Thread类的方法

Java Thread类

```
public synchronized void start()
```

- 启动该线程，并在调用run()方法之后马上返回。
- 如果线程已经启动，则抛出IllegalThreadStateException异常。

```
public void run()
```

- 该线程的主体部分，在线程启动后调用。

```
public final synchronized void join(long milis) throws InterruptedException
```

- 等待该线程消亡。所指定的超时可以精确到微秒，如果超时设为0，则表示线程将永远等待下去。

```
public static void yield()
```

- 导致目前正在执行的线程对象放弃处理器，使得其他一些可运行的线程得到调度。

```
pulic final int getPriority()
```

- 返回线程的优先级。

```
public final void setPriority(int newPriority)
```

- 设置线程的优先级。

注释：

如果要查看Thread类公共方法的完整列表，参见java.lang.Thread。

幸运的是，通常并不想改动线程的核心功能，相反，我们只是想实现一个新的例程thread_main。在这种情况下，我们更倾向于定义一个实现Runnable接口的新类，并在主方法中实例化一个或多个线程。这种方法更灵活，因为它允许新类扩展其他类；这种方法也更清晰，因为它不会改变Thread类的基本功能。

6.2.1 同步方法

除了基本的Thread对象，Java还支持一个简单方法，可以通过使用称为*同步方法*的结构来定义监视器。例如，可以如下定义一个同步计数器：

```
public class SynchronizedCounter
{
  public synchronized void update(int x)
  {
    count +=x;
  }
  public synchronized void reset()
  {
    count=0;
  }
}
```

通过这种定义，SynchronizedCounter类的一个实例成为了监视器，而且任意时刻，只有一个线程能执行该实例任意的同步方法，其余线程必须等待该线程退出监视器。这些监视器用起来非常方便，因为它们没有向外暴露保护监视器的互斥体。相反，隐藏的互斥体会被任意一个进入监视器的线程隐式地获得，然后在线程退出监视器时隐式地释放。

注意锁由线程所拥有，因此一个线程能再次获得它已经拥有的锁，从而允许同步方法递归使用。

6.2.2 同步语句

Java也提供了同步语句，它能支持比监视器粒度更细的同步。与POSIX Threads不同，Java没有特殊的互斥体数据结构，相反，任意Java对象都能用来指定同步。因此我们可以如下定义一个临界区：

```
Object lock;
synchronized(lock)
{
  /*临界区*/
}
```

同步语句比同步方法的粒度要细，这可从两方面说明：第一，临界区可以更小，可由某个方法中的单个块组成。第二，两段不交互的代码能在独立对象上同步，从而得到更大的并行性。

同步方法和同步语句除了实现互斥之外，还有另一个作用。为了理解这一点，我们需要知道，出于性能的原因，线程在本地保存了变量的缓存副本，而这些缓存副本中的值可能与主存中对其他线程可见的值有所不同[⊖]。同步语句和同步方法会刷新所有线程缓存的值到主存中，使它们在那一刻对所有其他线程可见。当然，这种变量刷新会产生性能开销。

6.2.3 统计3的个数程序实例

了解线程和同步语句的基本概念后，现在开始使用Java Threads实现统计3的个数问题，如图6-27中所示。可以看到使用了Runnable接口，在第78行使用了同步语句原子地更新count变量，而且该同步语句使用了名为lock的通用对象提供同步。

```
 1  import java.util.*;
 2  import java.util.concurrent.*;
 3
 4  public class CountThrees implements Runnable
 5  {
 6    private static final int ARRAY_LENGTH=1000000;
 7    private static final int MAX_THREADS=10;
 8    private static final int MAX_RANGE=100;
 9    private static final Random random=new Random();
10    private static int count=0;
11    private static Object lock=new Object();
12    private static int[] array;
13    private static Thread[] t;
14
15    public static void main(String[] args)
16    {
17      array=new int[ARRAY_LENGTH];
18
```

图6-27　用Java实现统计3的个数问题的解决方案

⊖ 这种存储器不一致性只是Java存储器模型的多种性质之一，它会导致不可预测的结果。更多细节请参见Java规范请求（JSR）133。

```
19        // 初始化数组中的元素
20        for(int i=0; i<array.length; i++)
21        {
22          array[i]=random.nextInt(MAX_RANGE);
23        }
24
25        //创建线程
26        CountThrees[] counters=new CountThrees[MAX_THREADS];
27        int lengthPerThread=ARRAY_LENGTH/MAX_THREADS;
28
29        for(int i=0; i<counters.length; i++)
30        {
31          counters[i]=new CountThrees(i*lengthPerThread,
32                                      lengthPerThread);
33        }
34
35        //运行线程
36        for(int i=0; i<counters.length; i++)
37        {
38          t[i]=new Thread(counters[i]);
39          t[i].start();
40        }
41        for(int i=0; i<counters.length; i++)
42        {
43          try
44          {
45            t[i].join();
46          }
47          catch(InterruptedException e)
48          { /*不做任何事*/   }
49        }
50
51        //打印3的数量
52        System.out.println("Number of threes: " + count);
53      }
54
55      private int startIndex;
56      private int elements;
57      private int myCount=0;
58
59      public CountThrees(int start, int elem)
60      {
61        startIndex=start;
62        elements=elem;
63      }
64
65      //重载Thread类中run方法
66      public void run()
67      {
68        //计算3的数量
69        for(int i=0; i<elements; i++)
70        {
71          if(array[startIndex+i]==3)
72          {
73            myCount++;
74          }
75        }
76        synchronized(lock)
77        {
78          count+=myCount;
79        }
80      }
81    }
```

图6-27 （续）

6.2.4 易变存储器

Java通过使用volatile字段修饰符，为同步变量提供了更轻量级的选择。volatile字段修饰符会告知语言实现，某个特定变量在寄存器中不能有单独的缓存值，因为该值必须与其他线程所看到的值保持一致。因此任何时候，易变变量都能在所有线程间保持一致。但它不同于同步方法或同步声明，使用它不需要刷新线程中其他变量的值，因此使用易变变量很高效。

6.2.5 原子对象

Java 5.0支持原子对象，在不需要使用锁的情况下，可以对单独的对象实现原子操作（参见代码规范6.20）。例如，java.util.concurrent.atomic包实现了AtomicBoolean类、AtomicInteger、AtomicLong和AtomicReference等类。这些类能支持原子地读取和更新变量的操作，因此原子对象可作为创建更高级非阻塞数据结构所需的构件。

代码规范6.20 Java原子对象的例子

AtomicInteger操作举例

```
boolean compareAndSet(expectedValue, updatedVaule);
```

- 如果当前值与expectedValue相同，则原子地设置当前值为updatedVaule。

```
int getAndIncrement();
```

- 原子地读取当前值，并将当前值递增1。

6.2.6 锁对象

Java 5.0在java.util.concurrent.locks包中提供了显式锁对象。隐式锁的一个问题是无法将锁回收，而显式锁对象就提供了这种能力，能用来防止死锁。尤其是tryLock()方法提供了超时机制。

6.2.7 执行器

Java 5.0还有执行器，它将线程管理与线程真正所完成的工作区分开来。和Pthreads接口不同，执行器接口允许线程返回值给它们的父进程，同时执行器接口支持更灵活的调度工作，例如允许线程以给定频率周期性地调用。执行器支持线程池的概念，允许程序重用线程，这远比不断创建和销毁线程开销要低。

6.2.8 并发集合

最后，Java 5.0支持了一组并发数据结构。这些数据结构都能被多个线程并发访问，包括：

- BlockingQueues
- ConcurrentMap
- ConcurrentSkipListMap

其他数据结构可以参考在线资源。

通过上述扩展性的支持，Java成为了可替代Pthreads的便利方案。

6.3 OpenMP

OpenMP接口是在上世纪90年代后期开发的，它为程序员提供了一个方便的方法，可以在

共享地址空间的机器上获得并行性。其基本的思想是以辅助的方式增强串行程序，指出能并行执行的代码。OpenMP模型使用起来明显比Pthreads要简单的多，但对可描述的并行交互类型的约束也要严格的多。OpenMP标准能支持C语言，C++和Fortran，在本小节中，我们对OpenMP的简单介绍将使用C语言例子。更多完整的信息可以在http://www.openmp.org上获得。

6.3.1 统计3的个数程序实例

OpenMP是通过pragma引入到C或C++程序中，通过注释（comment）引入到Fortran程序中。例如我们在第1章曾提及的统计3的个数的程序中增加三个pragma和一个声明，如图6-28所示。可以看到pragma形式如下所示：

```
#pragma omp <specifications>
```

兼容OpenMP的C语言编译器会识别出该结构，并用<specifications>生成正确的多线程代码；而那些不兼容OpenMP的编译器会简单地忽略pragma，进行标准的串行执行。在图6-28中，可以看到如果将pragma当成空格，则结果是该程序与第1章中第1个统计3的个数的程序基本相同，只不过有一些额外计算，但任何合理的编译器都能消除。因此在程序中增加pragma的做法在使用兼容编译器时会获益，否则也不会产生明显的开销。

图6-28中第3行的声明引入了私有变量count_p，它的角色是提供线程指定的位置，使得for循环中的每个线程都能累加自己的结果。第一个pragma（第5行）指定了程序主体部分能并行执行，而且指定了在主体部分中所用到的变量是共享的，或是私有的。若未指定，则所有变量默认都是共享的。所创建的线程数由系统掌握，很少由程序员控制。

```
 1  int count3s()
 2  {
 3    int i, count_p;
 4    count=0;
 5    #pragma omp parallel shared(array, count, length)\
 6      private(count_p)
 7    {
 8      count_p=0;
 9      #pragma omp for schedule(static) private(i)
10      for(i=0; i<length; i++)
11      {
12        if(array[i]==3)
13        {
14          count_p++;
15        }
16      }
17      #pragma omp critical
18      {
19        count+=count_p;
20      }
21    }
22    return count;
23  }
```

图6-28 将OpenMP应用到第1章串行的统计3的个数程序中。第1个pragma（第5行）识别出第7-21行将被并行化；第2个pragma（第9行）声明for循环将被多线程化；最后一个pragma（第17行）在临界区中组合在多线程的for循环中所创建的私有计数器

第2个pragma（第9行）说明了for循环中的迭代能以任意顺序执行，这意味着该迭代可以

并行执行。因此OpenMP将为每个线程分配

总迭代次数/线程数（totalIterations/numThreads）

次的迭代，虽然程序员也能控制批处理的大小，这会在稍后解释。当然如果pragma由于存在相关性而不正确，则并行执行的结果将是不可预测的。即使pragma在语义上是正确的，也不是所有for循环都能在OpenMP中被合法并行化。如代码规范6.21中所述，置于循环规范中的约束，允许编译器预测所要执行的迭代次数，并将它们分配到线程。

代码规范6.21　OpenMP的parallel for语句

```
Parallel for
#pragema omp parallel for
for(<var>=<exprl>;<var><relop><expr2>; <var>=<expr3>){<body>}
```

条件：

- <var>必须是一个带符号的整型变量，且在各个实例中都相同。
- <relop>必须是<，<=，=>，>其中之一。
- <expr2>，<expr3>必须是循环不变式的整型表达式。
- 如果<relop>是<或<=，则<expr3>必须每次迭代都递增；如果<relop>是=>或>，则<expr3>必须每次迭代都递减。
- <body>必须是基本块，即没有其他的进口或出口。

注释：

- pragma行可选的规范包括private和nowait。
- parallel for所创建的一组线程会在结束时结合，这意味着隐式的障栅同步。

代码规范6.22　OpenMP的归约操作。<op>可从随后的附表中选取

```
reducation
reduction(<op>, list>)
```

条件：

- <op>为附表中的某个操作符，其标识符是归约操作第一步中左边操作数的值。
- <list>是一组用来归约到累加值的变量，例如之前统计3的个数例子中的count。

注释：

Fortran有更多的<op>，包括<min><max>。

<op>	标识符
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

图6-28统计3的个数问题的解决方案中，每个线程执行一批迭代，并累加所找到的3的数量到每个线程的私有变量count_p中。线程完成for循环后，开始执行由第3个pragma说明的临界区（第17行）。编译器使用锁保护临界区，确保任意时刻最多只能有一个线程进入临界区，

使得各私有变量count_p能正确地组合到全局变量count。当所有线程退出临界区，到达第1个pragma并行块末尾时，它们会结合，并导致由单个线程控制。最后返回结果给调用程序。

6.3.2 parallel for的语义局限

虽然paralle for对于在串行程序中引入并行是一个方便的工具，但程序员必须要谨慎，所指定的循环迭代在实际上必须是独立的。例如，我们可以编写如下的统计3迭代：

```
#pragma omp parallel for private(i)          注意：这是错误的代码!
for(i=0; i<length; i++)
{
  if(array[i]==3)
  {
    count++;
  }
}
```

但由于count变量是共享的，因此在不同迭代间使用count会存在流相关，这意味着在不同线程间使用count也会存在流相关。当所有线程同时修改count时，就会产生一个毫无意义的结果。图6-28中的代码使用私有变量count_p，并在循环结束后将这些私有变量累加到count，以此解决该问题。除了私有化变量外，在OpenMP中，相关性也可以使用atomic，critical和barrier等pragma加以处理（参见代码规范6.21）。

6.3.3 归约

OpenMP的归约pragma能简化需要全局组合变量的计算。程序员使用reduction关键词，并给出组合操作符以及需要累加的变量，如代码规范6.22中所示。因此最佳的OpenMP的统计3的个数解决方案可能如下所示：

```
count=0;
#pragma omp parallel for reduction(+,count)
for(i=0; i<length; i++)
{
  count +=(array[i]==3)?1:0;
}
```

编译器会在几个线程间划分循环迭代，即先在局部累加计数器，然后自动组合这些结果。

代码规范6.23 OpenMP中原子性规范

atomic

```
#pragma omp atomic
  <var> <op> <expr> | <expr>++ | <expr>-- | ++<expr> | --<expr>
```

效果：

该pragma之后的语句执行不可间断。

条件：

- <var>是一个程序变量。
- <op>是以下操作之一：+=、-=、/=、<<=、>>=、&=、|=、^=。
- <expr>可以是任意的合法表达式。

注释：

在循环中使用原子操作会导致严重的性能问题。

6.3.4 线程的行为和交互

在OpenMP中，通常会创建线程以适应parallel for。在parallel for结束时，各线程会结合到一起，作为单个线程继续执行。这种结合以及parallel for结尾处的隐式障栅，可以通过在parallel for pragma中放置nowait的pragma加以避免：

```
#pragma omp parallel for nowait
```

如此，在循环结尾处，线程会简单地接着执行parallel for之后的指令。注意，这种方式的主要优势在于紧随其后的语句也是parallel for时。

为了确保线程修改某个公有变量时，不会被其他线程所干扰，可以使用atomic进行标识：

```
#pragma omp atomic
  score +=3;
```

atomic标识能保证存储器更新不会被干扰，即机器指令序列（1）从存储器中加载变量，（2）递增变量的值，（3）返回变量到存储器，将总是作为单个单元执行。（其他类型的变量更新也是有可能的，如代码规范6.23中所示）。当然，如果在循环中反复执行此种指令，则会影响到性能，因为线程必须串行地访问变量。

建立临界区比更新简单变量要复杂得多，可使用如下的排他程序。

```
#pragma omp exclusive(oneAtATime)
{
  ...任意时刻只有一个线程在此处...
}
```

它允许整个块没有冲突的执行，因为任意时刻，只允许一个线程执行临界区。圆括号中的命名（oneAtATime）是可选的临界区命名。如果一个程序的多处位置都使用了相同的名，则当某个线程执行其中一个临界区时，将排斥所有其他位置的所有线程。如果未命名，则相当于所有未命名的临界区都使用同一个命名。

6.3.5 段

OpenMP使用sections的pragma表示任务并行。例如，如果有三个独立的任务，Task_A()，Task_B()和Task_C()，它们可以并行执行且相互之间没有相关性，则其并行执行能以如下方式指定：

```
#pragma omp sections
{
  #pragma omp section
  {
    Task_A();
  }
  #pragma omp section
  {
    Task_B();
  }
  # pragma omp section
  {
    Task_C();
  }
}
```

sections之后的代码块列出了并行任务，可以是任意的C代码块，包括函数调用，且不限任务数。这其中的语义是每段都由一个线程控制，且直至执行到完成都与其他线程无关。执行的顺序并未指定。注意每个pragma之后的块都必须从自己的行上开始。

6.3.6 OpenMP总结

这一节OpenMP的简单介绍说明了该方法简单易用。其方便性源自于限制了程序员对并行的控制，限定了所提供信息的数量。无论如何，当“少许并行”就足够时，它会比较方便。更多信息请参考OpenMP的网站，也可以参见第9章中的评论，其中将OpenMP与其他程序设计技术进行了比较。

共享虚拟存储器 为什么基于线程的程序无法在那些不支持共享存储器的机器上执行？为什么不能使用软件在这种机器之上提供虚拟的共享地址空间？这些问题在上世纪80年代和90年代之间曾进行了深入的研究。其根本问题在于共享虚拟存储器系统需要处理所有的数据移动，当不了解应用的共享行为时，会很难做到有效地处理。尤其是，在共享粒度上存在折中：粗粒度能减少处理器间通信的开销，但会带来假共享；而细粒度会减少假共享，但会增大数据移动的开销。通常，我们会理想化地期望共享虚拟存储器系统的共享粒度能与应用共享的逻辑粒度相匹配。当然，即使这底层的共享虚拟存储器系统变得非常高效，基于线程的程序设计是否就是最佳的程序设计模型还是个未知数。

6.4 小结

POSIX Threads是非常强大的并行程序设计工具，它将巨大的潜能交到了程序员手中。从好的一面来说，基于线程的程序所使用的技巧几乎没有任何限制，这给予了程序员得出高效且有效的解决方案的能力。从坏的一面来说，这种能力同时伴随着相当大的风险。例如这类程序需要有精细的管理，以此确保正确性、无竞态条件、无死锁以及无难以捉摸的性能瓶颈。

Java通过提供对监视器和更高级并发数据结构的支持，提供了一种更简单的线程接口。但由于这些接口不够通用，因此Java还提供了低级的接口以实现POSIX Threads中的诸多功能。OpenMP展示了如何进行约束，以提供一个特别简单但又不太通用的程序设计模型。当我们展望未来，一个尚未有答案的问题是我们能否确定一个高级抽象集合，它位于OpenMP与POSIX Threads之间，既满足程序设计的方便性和灵活性，又能兼顾特殊的性能需要。这是一个值得深思而又吸引人的问题。

历史回顾

死锁的四个必要条件首先由Coffman等人提出[1971]。Hoare[1974]和Brinch Hansen[1975]在上世纪70年代中期分别提出了稍有不同的监视器。许多POSIX Threads的实现基于IEEE POSIX 1003.1c-1995标准，也有许多现成的书籍和教程介绍其高级技巧。OpenMP是上世纪90年代后期由供应商和学术界组成的社区所创建的。

习题

1. 编写一个简单的C程序，确定多核芯片是否能保持一致的cache。（请回忆cache可以是多层次的。）你编写的程序每次执行都确实能提供正确答案吗？如果允许用汇编语言编写（这样就可以使用Pthreads调用），能改进你的程序吗？
2. 在约束缓冲器例子中，我们使用单个互斥体同时保护nonempty和nonfull这两个条件变量。能否为每个条件变量各自配备一个互斥体？折中方案又是怎样的？

3. pthread_cond_wait()例程将保护性互斥体的地址作为参数，因此该例程能原子地阻塞等待线程，并原子地释放被等待线程持有的锁。解释为什么这两个操作必须原子地执行？
4. 针对图6-1中约束缓冲区的例子，举出一个关于缓冲区的不变式。它必须在监视器之外成立，但有时在监视器之内不满足。提示：答案可能是某个C语言代码无法检查的不变式。
5. 修改约束缓冲区的代码，以允许缓冲区的容量使用每个位置。注意修改时不能引入额外的变量。
6. 用Pthreads实现一个2维连续过度松弛程序，它使用分阶段的障栅。研究不同的数据划分，并观察这之间是否有明显的性能差异。
7. 测试图6-27中给出的OpenMP统计3的个数的程序，度量其性能，并根据你计算机上的处理器数确定其效率。
8. 编写OpenMP程序解决红/蓝仿真问题（参见第4章中的习题10）。
9. 使用习题8中你给出的解决方案，试着通过提高访问的局部性提高性能。
10. 编写Java Threads程序解决红/蓝仿真问题（参见第4章中的习题10）。

第7章 MPI和其他局部视图语言

在第6章中，我们关注了面向共享地址空间计算机的语言，本章和下一章将讨论能运行在任意并行计算机上的语言。我们将这些语言分成两大类：*局部视图语言*（本章的主题）和*全局视图语言*（第8章的主题）。我们会在第9章中定义这两个概念，现在只需知道全局视图语言提供了一些局部视图语言未曾提供的概念性的便利。

与第6章中一样，我们所说的“语言”包含库在内。本章的主题是消息传递接口（Message Passing Interface，MPI），这个库在局部视图语言中已经成为了标准。之后将简单介绍3种近期开发的语言：Co-Array Fortran、Unified Parallel C和Titanium，它们展示了局部视图语言如何将抽象层次提升到MPI之上。

7.1 MPI：消息传递接口

MPI程序设计模型极其简单。它提供了一个分布式存储器的程序设计模型。一组进程在该模型中通过发送消息进行通信。进程管理在MPI外部进行。MPI的执行一开始会有一些静态数量的进程，通常每个进程会被分配到不同处理器。由于每个进程都有各自的地址空间，程序员必须明确地表示他们的程序，以使各实例能在分布式数据结构的各独立部分上运行。

尽管概念很简单，但MPI却提供了超过一百个的例程。在本节中，我们将关注于构造这些例程的主要思想。首先，我们使用统计3的个数程序实例来介绍MPI程序的基本结构。此处我们使用了MPI的C语言绑定，而MPI标准还提供了Fotran和C++的绑定。之后将更为详细地解释MPI的核心概念，并介绍一组最常用的MPI函数。

在线MPI教程 *想了解MPI更多的细节，包括例子和文档，请参考美国*Lawrence Livermore*实验室的网站*：http://www.llnl.gov/computing/tutorials/mpi/

7.1.1 统计3的个数程序实例

图7-1显示了一个解决统计3的个数问题的MPI程序，图中第12～14行和第66行给出了对于所有MPI程序都通用的4个基本MPI例程。

MPI_Init()（第12行）用来初始化MPI的数据结构，它必须在每个进程调用任何其他MPI例程前被调用。MPI_Finalize()（第66行）用来清除MPI的数据结构，它必须是进程调用的最后一个MPI例程。

MPI_Comm_size()（第13行）用来确定在某个特定MPI*通信子*（communicator）中的进程数。我们会在下一节中更多地讨论通信子的概念，现在只需知道默认的通信子MPI_COMM_WORLD包含了所有进程。

MPI_Comm_rank()（第14行）用来返回在通信子中正在执行的进程的*序号*（rank）。该序号在通信子中用作区分各个进程的唯一标识，它允许进程指定它们所希望通信的进程。该序号是一个位于0与$P-1$之间的整数，含$P-1$，其中P是通信子中的进程数。（参见代码规范7.1～7.4）。

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

int main(argc, argv)
int argc;
char **argv;
{
  int myID, value, numProcs;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myID);

  length_per_process=length/numProcs;
  myArray=(int *) malloc(length_per_process*sizeof(int));

  /*读入数据，并将其在不同进程间分配*/
  if(myID==RootProcess)
  {
    if((fp=fopen(*argv, "r"))==NULL )
    {
      printf("fopen failed on %s\n", filename);
      exit(0);
    }
    fscanf(fp,"%d", &length);      /*读入输入的大小*/

    for(p=0; p<numProcs-1; p++)   /*代表各个其他进程读入数据*/
    {
      for(i=0; i<length_per_process; i++)
      {
        fscanf(fp,"%d", myArray+i);
      }
      MPI_Send(myArray, length_per_process, MPI_INT, p+1,
               tag, MPI_COMM_WORLD);
    }

    for(i=0; i<length_per_process; i++)  /*现在读入自己的数据*/
    {
      fscanf(fp,"%d", myArray+i);
    }
  }
  else
  {
    MPI_Recv(myArray, length_per_process, MPI_INT, RootProcess,
             tag, MPI_COMM_WORLD, &status);
  }

  /*做实际的工作*/
  for(i=0; i<length_per_process; i++)
  {
    if(myArray[i]==3)
    {
      myCount++;                          /*更新局部count值*/
    }
  }

  MPI_Reduce(&myCount,&globalCount, 1, MPI_INT, MPI_SUM,
             RootProcess, MPI_COMM_WORLD);

  if(myID==RootProcess)
  {
    printf("Number of 3's: %d\n", globalCount);
  }
  MPI_Finalize()
  return 0;
}
```

图7-1　统计3的个数问题的MPI解决方案

代码规范7.1 MPI_Init()

MPI_Init()

```
int MPI_Init(                          // 初始化MPI
  int *argc,                           // 命令行参量数
  char **argv,                         // 命令行参量
);
```

参量：

- 命令行参量数。
- 命令行参量。

注释：

在每个MPI进程中，该例程必须在任何其他MPI例程调用前调用。在一个进程中调用该例程超过一次就会发生错误，除非随后有一个MPI_Finalize()被调用。

返回值：

某个MPI错误码。

代码规范7.2 MPI_Finalize()

MPI_Finalize()

```
int MPI_Finalize(
);
```

注释：

在每个MPI进程中，该例程必须是最后一个被调用的MPI例程，它只能在所有其他MPI例程完成之后被调用。尤其是，任何未完成的通信操作都必须在调用该例程之前完成。

返回值：

某个MPI错误码。

代码规范7.3 MPI_Comm_Size()

MPI_Comm_Size()

```
int MPI_Comm_Size(                     // 获得指定通信子中的任务数
  MPI_Comm comm,                       // 指定的通信子
  int *size,                           // 任务数
);
```

参量：

- 指定的通信子。
- 指向大小的指针，其目标将包含指定通信子中的任务数。

注释：

该例程将获得通信子中的进程数。

返回值：

某个MPI错误码。

代码规范7.4 MPI_Comm_Rank()

MPI_Comm_Rank()

```
int MPI_Comm_Rank(                     // 获得某个进程在通信子中的序号
```

```
  MPI_Comm comm,                    // 通信子
  int *rank,                        // 序号
);
```

参量：

- 指定的通信子。
- 指向序号的指针，其目标将包含某个进程在指定通信子中的序号。

注释：

该例程将获得某个进程在通信子中的序号。

返回值：

某个MPI错误码。

为了理解这段代码的主体部分，假设文件globals.h包含了以下代码行：

```
#define RootProcess 0
int length;
int length_per_process;
int myStart;
int myCount=0;
int globalCount;
MPI_Status status;
int tag=1;
```

第19～48行显示了点对点通信（即从一个进程发送数据到另一个进程），如何在不同进程间分发数据（稍后我们将看到如何使这种分发更为简捷有效地执行）。这段代码从一个称为根进程（root process）的单个进程开始，从文件中读入数组内容，然后分发这些数据到其他进程。在第22～27行，根进程根据从命令行参量传入的名字，打开该名字所指定的文件，然后读入文件大小。第29～37行，根进程将文件内容读入到numProcs个字节块（chunk）中，然后发送前numProcs－1个字节块到其他进程，同时保留最后一个字节块为己所用。

具体地，第35行使用MPI_Send()例程将数据发送到其他进程。在MPI中，这种通信在发送者和接收者双方做了冗余指定。因此第44～48行显示了numProcs－1个非根进程调用MPI_Recv()接收数据。我们看到需要许多细节才能指定这样的一个消息，比如MPI_Send()有6个参数，而MPI_Recv()有7个。这些参数指定了涉及该通信操作的其他进程、消息的类型和长度以及标记（tag，即消息标记）。代码规范7.5和7.6给出了全部细节。

代码规范7.5　MPI_Send()

```
MPI_Send()
int MPI_Send(                           //阻塞式发送例程
  void          *buffer,                //欲发送数据的地址
  int           count,                  //欲发送数据元素的数量
  MPI_Datatype  type,                   //欲发送数据元素的类型
  int           dest,                   //目的地进程ID
  int           tag,                    //识别该消息的标记
  MPI_Comm      comm                    //MPI通信子
);
```

参量：

- 欲发送数据的地址。
- 欲发送数据元素的数量。

- 欲发送数据元素的类型。
- 接收该消息的进程ID。
- 消息标记，它能将该消息与发送到同一进程的其他消息区分开来。
- 所要使用的MPI通信子。

注释：

该例程用来发送消息到另一个进程。它具有阻塞式语义：即直到消息被发送后，例程才会返回。MPI_Isend()是该发送操作的非阻塞式版本。它使用了MPI_Request类型的第7个参数，用来在等待完成时将该发送与其他MPI_Isend()调用区分开来。

返回值：

某个MPI错误码。

代码规范7.6 MPI_Recv()

```
MPI_Recv()
int MPI_Recv(
  void              *buffer,      //阻塞式接收例程
  int               count,        //欲接收数据的地址
  MPI_Datatype      type,         //欲接收数据元素的数量
  int               source,       //欲接收数据元素的类型
  int               tag,          //发送进程的ID
  MPI_Comm          comm,         //识别该消息的标记
  MPI_Status        *status       //MPI通信子
);                                //该接收操作的状态
```

参量：

- 前6个参量分别对应于MPI_Send()的参量。
- 如欲从其他任意进程接收消息，则在source处使用MPI_ANY_SOURCE。
- 如欲匹配任意标记，则在第5个参数处使用MPI_ANY_TAG。

注释：

该例程从其他进程处接收数据。它有阻塞式语义：即直到接收到消息后，例程才会返回。MPI_Irecv()是该接收操作的非阻塞式版本。它使用了MPI_Request类型的第7个参数，用来在等待完成时将该接收与其他MPI_Irecv()调用区分开来。

返回值：

某个MPI错误码。

真正的工作是在第50～57行完成的，此处每个进程统计自己那部分数组的3的数量。最终，通过调用MPI_Reduce()将各个count的本地值相加，归约成单个值。MPI_Reduce()是涉及了通信子中多个成员的集合通信操作的一个实例（参见代码规范7.7）。在本例中，每个进程都提供了单个整数，所有这些值相加后，会返回到根进程中由第2个参数所指定的地址。请注意，此处将私有变量的概念引入到消息传递中是如此的自然！

集合通信操作比点对点通信操作要更加容易使用。例如数组数据的分发，在第35行和46行使用了点对点通信，而如果使用MPI_Scatter()将会更为简洁，它是一个能将数据从一个进程分发到所有其他进程的集合操作（参见代码规范7.8）。如图7-2所示，整个输入数组被读入到单个数组中，然后使用MPI_Scatter()例程将该数组散射到通信子MPI_COMM_WORLD中的

所有进程。注意MPI_Scatter()将被所有参与的进程调用。MPI_Gather()是一个与之相对的例程，它从多个进程处收集数据，然后放置到一个进程中（参见代码规范7.9）。

```
16  length_per_process=length/size;
17  myArray=(int *) malloc(length_per_process*sizeof(int));
18
19  array=(int *) malloc(length*sizeof(int));
20
21   /* 读入数据，在不同进程间分配*/
22  if(myID==RootProcess)
23  {
24    if((fp=fopen(*argv, "r"))==NULL)
25    {
26      printf("fopen failed on %s\n", filename);
27      exit(0);
28    }
29    fscanf(fp,"%d", &length);        /* 读入输入的大小*/
30
31    for(i=0; i<length-1; i++)        /* 读入整个输入文件*/
32    {
33      fscanf(fp,"%d", myArray+i);
34    }
35  }
36
37  MPI_Scatter(Array, length_per_process, MPI_INT,
38              myArray, length_per_process, MPI_INT,
39              RootProcess, MPI_COMM_WORLD);
```

图7-2　使用散射操作分发数据的替换代码（对应于图7-1中第16～48行）

代码规范7.7　MPI_Reduce()

```
MPI_Reduce()
int MPI_Reduce(                          //归约例程
  void          *sendbuffer,             //欲发送数据的地址
  void          *recvbuffer,             //欲接收数据的地址
  int           count,                   //欲接收数据元素的数量
  MPI_Datatype  datatype,                //每个元素的类型
  MPI_OP        op,                      //MPI操作符
  int           root,                    //将保存结果的进程
  MPI_Comm      comm                     //MPI通信子
);
```

注释：

该例程实现了归约操作。它的一个特殊形式MPI_AllReduce()，会把所有进程当作根结点一般对待，这意味着归约后的值将被传递到所有进程中由第2个参数所指定的地址。MPI_AllReduce()相当于调用MPI_Reduce()之后，再接着调用MPI_Bcast()，后者会将值广播给通信子中的所有进程。

返回值：

某个MPI错误码。

代码规范7.8　MPI_Scatter()

```
MPI_Scatter()
int MPI_Scatter(                          //散射例程
  void          *sendbuffer,              //欲发送数据的地址
  int           sendcount,                //欲发送数据元素的数量
  MPI_Datatype  sendtype,                 //欲发送数据元素的类型
  void          *destbuffer,              //欲接收数据的地址
  int           destcount,                //欲接收数据元素的数量
  MPI_Datatype  desttype,                 //欲接收数据元素的类型
  int           root,                     //根进程的序号
  MPI_Comm      comm                       //MPI通信子
);
```

参量：

- 前3个参量指定了要发送给各进程的数据元素的地址、数量和类型。这些参量只对根进程有意义。
- 随后的3个参量指定了各进程所要接收的数据元素的地址、数量和类型。发送数据与接收数据的数量和类型之间可能会有所不同，这是由于数据类型转换的缘故。
- 第7个参量指定了根进程，即数据源。
- 第8个参量指定了所使用的MPI通信子。

注释：

该例程从根进程分发数据到所有进程，包括根进程本身。一个更为复杂的版本是MPI_Scatterv()，它允许根进程发送不同数量的数据到不同进程。具体细节见MPI标准。

返回值：

某个MPI错误码。

代码规范7.9 MPI_Gather()

```
MPI_Gather()
int MPI_Gather(                           //聚集例程
  void          *sendbuffer,              //欲发送数据的地址
  int           sendcount,                //欲发送数据元素的数量
  MPI_Datatype  sendtype,                 //欲发送数据元素的类型
  void          *recvbuffer,              //欲接收数据的地址
  int           recvcount,                //欲接收数据元素的数量
  MPI_Datatype  recvtype,                 //欲接收数据元素的类型
  int           root,                     //根进程的序号
  MPI_Comm      comm                      //MPI通信子
);
```

参量：

- 前3个参量指定了各进程所要发送的数据元素的地址、数量和类型。
- 随后的3个参量指定了接收进程（即根进程）所要接收的数据元素的地址、数量和类型。发送数据与接收数据的数量和类型之间可能会有所不同，这是由于数据类型转换的缘故。
- 第7个参量指定了要接收数据的根进程。
- 第8个参量指定了所使用的MPI通信子。

注释：

该例程从通信子中所有进程处收集数据，然后放置到根进程中。一个更为复杂的版本是MPI_ Gatherv()，它允许根进程从不同进程处接收[⊖]不同数量的数据。具体细节见MPI标准。

返回值：

某个MPI错误码。

7.1.2 组和通信子

在MPI中，*通信子*（communicator）是一个范围机制，它定义了一个可以相互间通信的进程集合。例如，为了将库例程的消息与应用层例程的消息区分开来，我们为库例程定义了单独的通信子。

组（group）是一组有序进程的集合，它可用来定义集合通信操作。组中的每个进程都分配了唯一序号，即ID，它们的值位于0到$P-1$之间，含$P-1$，其中P是组中的进程数。一个进程可以隶属于多个组。例如，考虑在一个程序中，我们可以将所有进程看作是进程的一个2维数组。在此种计算中，一个进程既可以隶属于该2维数组进程所定义的行进程组，也可以隶属于同一2维数组进程所定义的列进程组。通过定义这两个组，一个进程能在其进程行或进程列之间散射数据，或在这些组中执行其他的集合通信操作。

通信子和组都能动态创建和销毁，我们将在讨论集合通信细节时给出使用它们的一个实例。有许多用来操作组和通信子的例程，具体细节请参见MPI手册。

7.1.3 点对点通信

在MPI中，点对点通信通过发送进程和接收进程这两者冗余地指定。消息的匹配基于操作所说明的源/目的进程，以及所指定的消息*标记*（tag）。标记是一个用户自定义的非负整数，它能用来从逻辑上区分同一对进程间的不同消息。在某些实例中，进程可能无法预知它要与哪些进程进行通信，此时它可以指定MPI_ANY作为其源进程或目的进程。

MPI能确保同一源与目的地进程之间的消息是顺序传递的，但当通信所涉及的进程数超过2时，就无法保证消息的顺序性。

MPI提供了许多点对点通信的变体。原因在于必然会发生的重要同步和数据拷贝，如图7-3所示，对于每个消息，数据必须在四个地址空间之间拷贝。注意这个图只关注了最基本的数据传输操作，它忽略了特定实现的细节，那可能需要额外的缓冲和联络(handshaking)。

为了允许程序员隐藏部分时延，MPI提供了给出部分细节的不同版本的发送和接收操作。例如，非阻塞式版本允许一个进程当它在等待消息传递时执行其他一些独立的工作。这种通信与计算的重叠很类似于我们在第6章中曾看到的分阶段障栅，因此它能以增加程序复杂性为代价提高性能。

我们现在来讨论点对点通信操作的多种变体。

⊖ 原文此处误为发送。——译者注

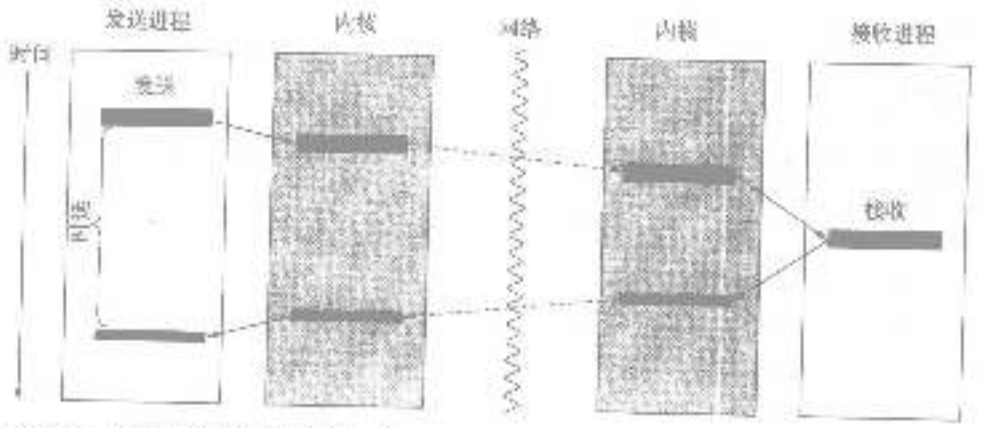


图7-1 每个消息在四个地址空间之间移动时，必须通过拷贝的方式，为每次拷贝都会对整个时延有所增加

标准发送和接收。标准发送和接收是阻塞式的，即在传输没有本地完成前，它们是不会返回的。如果消息传输在本地进程中所需执行的传输部分已经完成，就称为是本地完成。如果消息的整个传输都已完成，则称为是全局完成。因此，一旦MPI_Send()返回，发送进程改写缓冲区就是安全的。同理，一旦MPI_Recv()返回，接收进程使用缓冲区中的值就是安全的。

非阻塞式通信。为了隐藏通信时延，通常会重叠通信和计算，因此MPI提供了非阻塞式通信。MPI_Isend()和MPI_Irecv()就是标准发送和接收的非阻塞式版本。它们在操作本地完成之前，就立即返回。（此处I是immediate（立即）模式的缩写）。

使用MPI_Isend()，发送进程并不知道缓冲区中的数据何时会被真正地传输到接收进程，因此除非有某种显示表明消息传输已经完成，否则改写缓冲区将是不安全的。为了等待非阻塞式操作完成，进程可以调用MPI_Wait()例程，它能阻塞进程，直至指定操作已全局完成。为了测试一个非阻塞式操作是否已完成，MPI_Test()例程能立即返回，并依据消息传输完成的状态，通过将第2个参数值设置为真或伪加以表示。

类似地，对于MPI_Irecv()，直到操作全局完成，缓冲区中的数据都是无效的。因此MPI_Wait()和MPI_Test()对于接收进程而言也是很有用的操作。

其他通信模式。除了标准发送操作，MPI还提供了相关语义的其他三种模式。每种模式的发送可以是阻塞式的或是非阻塞式的，同时任意的发送例程都可以与任意的接收例程相匹配。

- 同步发送。同步发送（MPI_Ssend()和MPI_Issend()）在语言中提供了类似于Ada语言中集结点（rendezvous）的语义：即直到接收进程开始接收消息，发送进程才会返回。
- 缓冲发送。缓冲发送（MPI_Bsend()和MPI_Ibsend()）允许程序员为消息申请缓冲区空间，这能使程序避免由于系统缓冲区空间不足而引起的任何问题。与MPI实现相关，该模式对于需要用大容量存储器来缓冲消息，或是任意时间均有特别大量消息在传输的程序特别适用。对于这些例程，MPI_Buffer_attach()和MPI_Buffer_detach()例程可用来指定已分配的存储器。
- 就绪发送。为了在众多并行计算机上提高性能，MPI_Rsend()和MPI_Irsend()例程允许将消息直接放置到存储器单元，以避免联络和缓冲的开销。为了使用这种例程，程序员必须保证接收操作在消息到达之前就已经启动。如果违反了此种定时假设，则在执行接收操作时将标志一个错误。当然，由于这种额外的假设，这种模式极易出错。

虽然这些发送和接收的更复杂版本能提高性能，但它们会妨碍性能可移植性，当机器特性

发生变化时，对不同版本的选择也会发生变化。况且使用这些例程会显著复杂化程序的文本。

7.1.4 集合通信

集合通信操作是涉及到多个进程的高层通信操作。除了我们之前在统计3的个数程序实例中看到的散射（scatter）和归约（reduce）例程，还有扫描（scan）例程，广播（broadcast）例程，障栅（barrier）例程，以及将分布式的数据收集到单个进程的聚集（gather）例程。这些例程的接口可参见代码规范7.7～7.12。

我们之前所举的例子将集合通信操作应用到所有进程。我们现在介绍一个例子来说明如何设置多个组，而每个组对应于2维数组进程的每一行。设置好这些组后，我们将展示如何在各处理器行间广播不同的值。

图7-4显示了设置这些组的代码。假设全局整数numCols能保存进程的列数。为简短起见，我们采用了C语言的一些特性从而能静态初始化一个动态分配的数组。

```
 1  int numCols;                    /* 已在别处初始化 */
 2
 3  void broadcast_example()
 4  {
 5    int **ranks;                  /* 属于每个组的序号 */
 6    int myRank;
 7    int rowNumber;                /* 该进程的行号 */
 8    int random;                   /* 欲广播的值 */
 9    rowNumber=myRank/numCols;
10    MPI_Group globalGroup, newGroup;
11    MPI_Comm rowComm[numCols];
12
13    /* 初始化rank[][]数组 */
14    ranks[0]={0,1,2,3};   /* 非合法的C语言 */
15    ranks[1]={4,5,6,7};
16    ranks[2]={8,9,10,11};
17    ranks[3]={12,13,14,15};
18
19    /* 抽取出原先组的句柄 */
20    MPI_Comm_group(MPI_COMM_WORLD, &globalGroup);
21
22    /* 定义新组 */
23    MPI_Group_incl(globalGroup, P/numCols, ranks[rowNumber], &newGroup);
24
25    /* 创建新通信子 */
26    MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);
27
28    random=rand();
29
30    /* 在行间广播随机数'random' */
31    MPI_Bcast(&random, 1, MPI_,INT rowNumber*numCols, newComm);
32  }
```

图7-4　组内集合通信示例

在第20行，MPI_Comm_group()例程（参见代码规范7.14）返回一个与现有MPI_COMM_WORLD通信子相关的组句柄。然后我们使用MPI_Group_incl()例程定义一个新组：第1个参数从globalGroup中选择了组的集合，用ranks[][]数组来指定，其中新组的大小是P/numCols，名称为newGroup（参见代码规范7.15）。

在第26行，MPI_Comm_create()例程为新组创建了新通信子，它能被传递给第31行的MPI_Bcast()（参见代码规范7.16）。在此种情况下，广播将从一行中最左边的进程向同一行中的所有进程发送随机生成的一个整数。

这个例子手动创建了2维数组的进程组。其实MPI已经提供了虚拟拓扑的概念，并允许定义和重用这种拓扑。尤其是，MPI定义了笛卡尔（Cartesian）网格和图像拓扑。对这个高深主题感兴趣的读者可以参考MPI手册。

代码规范7.10 MPI_Scan()

MPI_Scan()

```
int MPI_Scan(                          //扫描例程
  void          *sendbuffer,           //欲发送数据的地址
  void          *recvbuffer,           //欲接收数据的地址
  int           count,                 //欲接收数据元素的数量
  MPI_Datatype  datatype,              //欲接收数据元素的类型
  MPI_OP        op,                    //MPI操作符
  MPI_Comm      comm                   //MPI通信子
);
```

注释：

该例程具有与归约操作几乎相同的接口，只是不需要根进程而已。

返回值：

某个MPI错误码。

代码规范7.11 MPI_Bcast()。用来从一个根进程广播数据到通信子中所有其他进程的MPI例程

MPI_Bcast()

```
int MPI_Bcast(                         //广播例程
  void          *buffer,               //欲发送数据的地址
  int           count,                 //欲发送数据元素的数量
  MPI_Datatype  datatype,              //欲发送数据元素的类型
  int           root,                  //根进程的序号
  MPI_Comm      comm                   //MPI通信子
);
```

参量：

- 前3个参量指定了欲发送给各进程的数据元素的地址、数量和类型。
- 第4个参量指定了根进程或发送进程的序号。
- 第5个参量指定了所使用的通信子。

注释：

该例程用来从根进程广播数据到通信子中所有其他的进程。与MPI_Scatter()和MPI_Gather()不同，根进程与接收进程之间的元素数量和类型必须保持一致。

返回值：

某个MPI错误码。

代码规范7.12　MPI_Barrier()

```
MPI_Barrier()
int MPI_Barrier(                                    //障栅例程
  MPI_Comm          comm                            //MPI通信子
);
```

参量：

- 该参量指定了所使用的通信子。

注释：

该例程将阻塞进程，直到通信子中所有进程都到达该点。

返回值：

某个MPI错误码。

代码规范7.13　MPI_Wtime()

```
MPI_Wtime()
double MPI_Wtime(    //定时例程
);
```

注释：

该例程返回用秒表示的当前时间。

返回值：

某个MPI错误码。

代码规范7.14　MPI_Comm_group()

```
MPI_Comm_group()
int MPI_Comm_group(
  MPI_Comm          comm                  //MPI通信子
  MPI_Group         *group                //与通信子相关的组
);
```

注释：

返回与通信子相关的组的句柄。

返回值：

某个MPI错误码。

代码规范7.15　MPI_Group_incl()

```
MPI_Group_incl()
int MPI_Group_incl(
MPI_Group           group                 //现有的组
  int               size,                 //新组的大小
  int               *rank                 //欲包括的进程的序号
  MPI_Group         *newGroup             //欲创建的新组
);
```

注释：

通过从现有的组中选出进程，以创建新组。

返回值：

某个MPI错误码。

代码规范7.16 MPI_Comm_create()

```
MPI_Comm_create()
int MPI_Comm_create(
  MPI_Comm          origComm          //现有的通信子
  MPI_Group         newgroup          //新组
  MPI_Comm          *newComm          //新建的通信子
)
```

注释：

为指定的组创建一个新通信子。

返回值：

某个MPI错误码。

7.1.5 举例：连续过度松弛

为了获得使用MPI实现并行程序的经验，考虑创建2维连续过度松弛（Successive over-relaxation，SOR）程序。

问题说明。SOR经常用来求解微分方程组，例如用来计算液体流的Navier Stokes方程。我们的计算从有n个值的二维数组开始。在每次迭代中，计算将数组的每个值替换为周围4个最近邻居的平均值。那些在数组边界上的值将使用预先确定的常数值，称为边界值，作为它们所缺失的邻居值。我们假定左边界的边界值都为1，其余的边界值都为0，而二维数组内的每个值均初始化为0。

处理边界值最简单的方法是将它们分配到二维数组中，并用它们的常数值初始化，得到如图7-5中所示的情况。

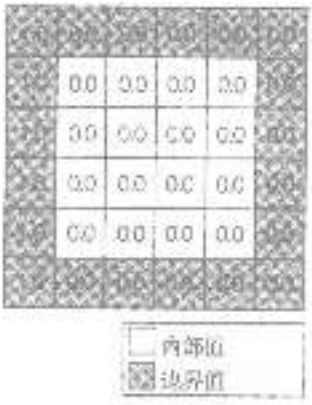


图7-5 二维松弛（每次迭代）将所有内部值替换为其周围4个最近邻居的平均值

MPI解决方案。所需求解的问题是在二维数组上定义的，由于每次迭代中每个数组元素所需的计算量是固定的，因此该计算是规则且平衡的，所以静态分配这项工作是有可能的，即每个进程将分配到大约$1/P$个数组元素。由于每个数组元素的值依赖于它周围4个最近的邻居，因此块分配比循环分配或块－循环分配的局部性都要好。尤其是，我们选择了二维块分配，因为与1维块分配相比，它能最小化通信总量。

图7-6显示了二维SOR解决方案的主循环部分。这段代码假设我们使用了Rows × Cols个进

程，且每个进程拥有Height × Width个数据，这当中包括了用来保持重叠区域的额外行和列。使用这些重叠区域将允许计算的主循环部分正常处理，而无需考虑非本地值的特殊情况。此处我们只是简单假设它们的值早已被通信操作正确地设置了。在每次迭代中，代码从val数组处读取，然后写回到new数组中。在每次迭代的最后，交换这两个数组的指针，以便为下次迭代做好准备。注意，发送数据到东边邻居和西边邻居时，数组列会先复制到一段连续的缓冲区中，然后作为单个消息发送。

```
  1  #define Top      0
  2  #define Left     0
  3  #define Right    (Cols-1)
  4  #define Bottom   (Rows-1)
  5
  6  #define NorthPE(i)       ((i)-Cols)
  7  #define SouthPE(i)       ((i)+Cols)
  8  #define EastPE(i)        ((i)+1)
  9  #define WestPE(i)        ((i)-1)
...
101  do
102  { /*
103    * 发送数据到周围4个邻居
104    */
105    if(row !=Top)                                    /* 发送到北边邻居 */
106    {
107      MPI_Send(&val[1][1], Width-2, MPI_FLOAT,
108               NorthPE(myID), tag, MPI_COMM_WORLD);
109    }
110
111    if(col !=Right)                                  /* 发送到东边邻居 */
112    {
113      for(i=1; i<Height-1; i++)
114      {
115        buffer[i-1]=val[i][Width-2];
116      }
117      MPI_Send(buffer, Height-2, MPI_FLOAT,
118               EastPE(myID), tag, MPI_COMM_WORLD);
119    }
120
121    if(row !=Bottom)                                 /* 发送到南边邻居 */
122    {
123      MPI_Send(&val[Height-2][1], Width-2, MPI_FLOAT,
124               SouthPE(myID), tag, MPI_COMM_WORLD);
125    }
126
127    if(col !=Left)                                   /* 发送到西边邻居 */
128    {
129      for(i=1; i<Height-1; i++)
130      {
131        buffer[i-1]=val[i][1];
132      }
133      MPI_Send(buffer, Height-2, MPI_FLOAT,
134               WestPE(myID), tag, MPI_COMM_WORLD);
135    }
136
137    /*
138     * 接收消息
139     */
140    if(row !=Top)                                    /* 从北边邻居处接收 */
```

图7-6　二维SOR计算中主循环部分的MPI代码

```
141     {
142       MPI_Recv(&val[0][1], Width-2, MPI_FLOAT,
143                NorthPE(myID), tag, MPI_COMM_WORLD, &status);
144     }
145
146     if(col !=Right)                          /* 从东边邻居处接收 */
147     {
148       MPI_Recv(&buffer, Height-2, MPI_FLOAT,
149                EastPE(myID), tag, MPI_COMM_WORLD, &status);
150       for(i=1; i<Height-1; i++)
151       {
152         val[i][Width-1]=buffer[i-1];
153       }
154     }
155
156     if(row !=Bottom)                         /* 从南边邻居处接收 */
157     {
158       MPI_Recv(&val[Height-1][1], Width-2, MPI_FLOAT,
159                SouthPE(myID), tag, MPI_COMM_WORLD, &status);
160     }
161
162     if(col !=Left)                           /* 从西边邻居处接收 */
163     {
164       MPI_Recv(&buffer, Height-2, MPI_FLOAT,
165                WestPE(myID), tag, MPI_COMM_WORLD, &status);
166       for(i=1; i<Height-1; i++)
167       {
168         val[i][0]=buffer[i-1];
169       }
170     }
171
172     delta=0.0; /* 计算所有点的平均值和delta */
173     for(i=1; i<Height-1; i++)
174     {
175       for(j=1; j<Width-1; j++)
176       {
177         average=(val[i-1][j]+val[i][j+1]+
178                  val[i+1][j]+val[i][j-1])/4;
179         delta=Max(delta, Abs(average-val[i][j]));
180         new[i][j]=average;
181       }
182     }
183
184     /* 找出最大的差值 */
185     MPI_Reduce(&delta, &globalDelta, 1, MPI_FLOAT, MPI_MAX,
186                RootProcess, MPI_COMM_WORLD);
187     Swap(val, new);
188   } while(globalDelta >=  THRESHOLD);
```

图7-6 （续）

7.1.6 性能问题

且不管底层硬件通信基础设施的时延，MPI消息本身在每个消息发送时就产生了很大的开销。因此我们通常希望减少消息发送的数量，通过（1）选择那些极少产生跨进程相关性的算法；（2）将多个消息合并成单个大消息。

二维SOR解决方案（图7-6）第113-116行中可以看到使用第2种方法的例子。我们将本地

数组沿右边一列的值拷贝到一个称为buffer的数组中。希望发送任意长度链表（linked list）的元素时，情况将会变得更加复杂。这里有一些细节需要解决，即如何在只有发送者知道长度的情况下，让发送者和接收者对消息长度达成共识？一种解决方案是使用一个协议，让发送者先发送长度给接收者，但这样就加倍了消息数。另一种解决方案是将消息分解为一些固定大小的大字节块，并在消息中加入提示，是否会有更多的数据需要发送。第二种方案的缺点在于短消息将被扩展到比其更大的固定大小，因而会浪费缓冲区的空间。

消息巨大的开销以及MPI进程固定的集合，都表明了MPI程序内在的静态性。但正如图7-7所示，MPI是能够执行动态工作分配的。图中显示了一组4个进程，在周期地交换有关各进程负载状态的信息，并使用该信息在本地计算出下次迭代所期望的工作分配，传送相应的工作（也许是接收工作），然后继续计算。当然关键在于，只有当这种工作分配的计算只占所有计算的一小部分，且它所改进的负载平衡得益大于这个昂贵的工作再分配协议所付出的代价时，这种方案才是可行的。

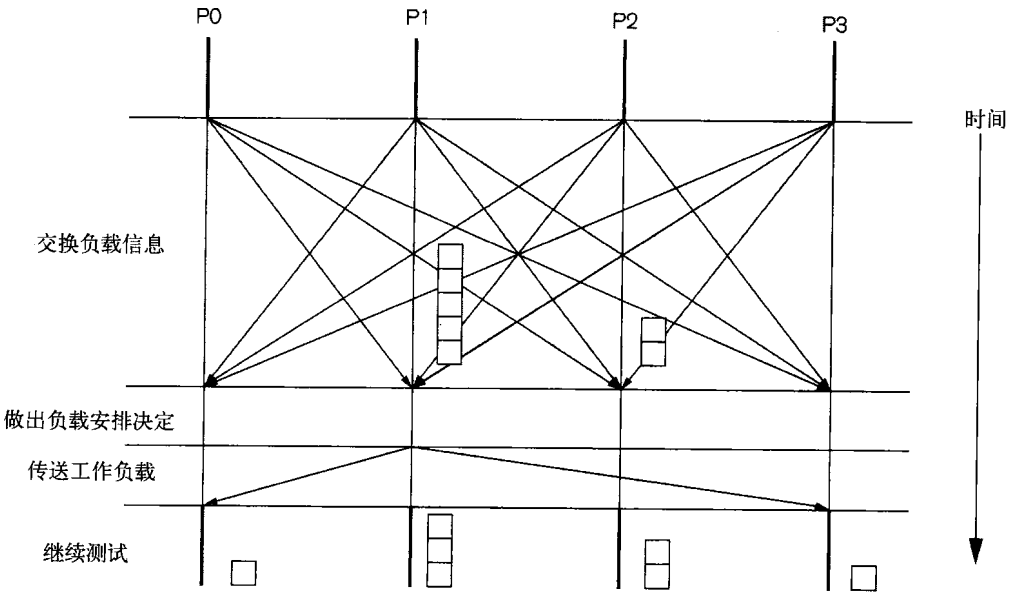


图7-7　MPI中动态工作再分配的图示

重叠通信和计算。由于每个消息巨大的启动开销，因此重叠通信和计算通常是很有用的。作为一个具体实例说明如何通过使用非阻塞式通信来实现这种优化，图7-8显示了我们应怎样修改图7-6的2维SOR程序以隐藏通信的时延。在代码中看到了一些主要的差异。首先，我们使用MPI_Isend()和MPI_Irecv()替换了原先的阻塞式版本。其次，当这些通信在执行时，我们对所有完全能在本地计算的数组中的值进行连续过度松弛计算，即所有无需访问任何非本地数组值的计算。因此第173～182行中嵌套的for循环并不迭代本地数组的边界，这是由于边界值依赖于邻居进程上的值。同时还在第183行增加了MPI_Waitall()（参见代码规范7.17）的调用，它将一直阻塞，直至所有8个非阻塞式调用完成。那时，我们将能完成余下边界点的计算（第185～215行）。

正如我们所看到的，重叠通信和计算的主要代价在于将平均值的局部计算一分为二，一部分能无关于通信独立执行，而另一部分则要依赖于通信的结果。这使得计算的整个逻辑变得晦涩难懂，同时也明显增加了代码的长度。

```
101  do
102  {  /*
103     * 发送数据到周围4个邻居
104     */
105    if(row!=Top)                          /* 发送到北边邻居 */
106    {
107      MPI_Isend(&val[1][1], Width-2, MPI_FLOAT,
108                NorthPE(myID), tag, MPI_COMM_WORLD, &requests[0]);
109    }
110
111    if(col!=Right)                        /* 发送到东边邻居 */
112    {
113      for(i=1; i<Height-1; i++)
114      {
115        buffer[i-1]=val[i][Width-2];
116      }
117      MPI_Isend(buffer, Height-2, MPI_FLOAT,
118                EastPE(myID), tag, MPI_COMM_WORLD, &requests[1]);
119    }
120
121    if(row!=Bottom)                       /* 发送到南边邻居 */
122    {
123      MPI_Isend(&val[Height-2][1], Width-2, MPI_FLOAT,
124                SouthPE(myID), tag, MPI_COMM_WORLD, &requests[2]);
125    }
126
127    if(col!=Left)                         /* 发送到西边邻居 */
128    {
129      for(i=1; i<Height-1; i++)
130      {
131        buffer[i-1]=val[i][1];
132      }
133      MPI_Isend(buffer, Height-2, MPI_FLOAT,
134                WestPE(myID), tag, MPI_COMM_WORLD, &requests[3]);
135    }
136
137    /*
138     * 接收消息
139     */
140    if(row!=Top)                          /* 从北边邻居处接收 */
141    {
142      MPI_Irecv(&val[0][1], Width-2, MPI_FLOAT,
143                NorthPE(myID), tag, MPI_COMM_WORLD, &requests[4]);
144    }
145
146    if(col!=Right)                        /* 从东边领居处接收 */
147    {
148      MPI_Irecv(&buffer, Height-2, MPI_FLOAT,
149                EastPE(myID), tag, MPI_COMM_WORLD, &requests[5]);
150      for(i=1; i<Height-1; i++)
151      {
152        val[i][Width-1]=buffer[i-1];
153      }
154    }
155
156    if(row!=Bottom)                       /* 从南边邻居处接收 */
157    {
158      MPI_Irecv(&val[Height-1][1], Width-2, MPI_FLOAT,
159                SouthPE(myID), tag, MPI_COMM_WORLD, &requests[6]);
```

图7-8 使用非阻塞式发送和接收的2维SOR的MPI程序

```
    }

    if(col!=Left)                    /* 从西边邻居处接收 */
    {
      MPI_Irecv(&buffer, Height-2, MPI_FLOAT,
                WestPE(myID), tag, MPI_COMM_WORLD, &requests[7]);
      for(i=1; i<Height-1; i++)
      {
        val[i][0]=buffer[i-1];
      }
    }

    delta=0.0; /* 计算所有点的平均值和delta */
    for(i=2; i<Height-2; i++)
    {
      for(j=2; j<Width-2; j++)
      {
        average=(val[i-1][j]+val[i][j+1]+
                 val[i+1][j]+val[i][j-1])/4;
        delta=Max(delta, Abs(average - val[i][j]));
        new[i][j]=average;
      }
    }
    MPI_Waitall(8, requests, status);

    /* 更新顶部和底部的边界包括各个角 */
    for(j=1; j<Width-1; j++)
    {
      i=1;
      average=(val[i-1][j]+val[i][j+1]+
               val[i+1][j]+val[i][j-1])/4;
      delta=Max(delta, Abs(average-val[i][j]));
      new[i][j]=average;

      i=Height-2;
      average=(val[i-1][j]+val[i][j+1]+
               val[i+1][j]+val[i][j-1])/4;
      delta=Max(delta, Abs(average-val[i][j]));
      new[i][j]=average;
    }

    /* 更新左边和右边的边界不包括各个角 */
    for(i=2; i<Height-2; i++)
    {
      j=1;
      average=(val[i-1][j]+val[i][j+1]+
                 val[i+1][j]+val[i][j-1])/4;
      delta=Max(delta, Abs(average - val[i][j]));
      new[i][j]=average;

      j=Width-2;
      average=(val[i-1][j]+val[i][j+1]+
                 val[i+1][j]+val[i][j-1])/4;
      delta=Max(delta, Abs(average-val[i][j]));
      new[i][j]=average;
    }
    /* 找出最大的差值 */
    MPI_Reduce(&delta, &globalDelta, 1, MPI_FLOAT, MPI_ MAX,
               RootProcess, MPI_COMM_WORLD);
    Swap(val, new);
  } while(globalDelta >= THRESHOLD);
```

图7-8 （续）

代码规范7.17 MPI_Waitall()

```
MPI_Waitall()
int MPI_Waitall(                          //等待例程
  int           count,                    //欲等待的全部进程数
  MPI_Request   *requests,                //欲等待的请求数组
  MPI_Status    *status,                  //状态值的数组
);
```

注释：

- 在所有计数（count）操作完成之前，该例程将一直阻塞。第2个和第3个参量必须有各自的计数器入口，status数组中每个入口都会收到来自request数组中对应入口的返回状态。
- 有许多等待调用的变体。例如MPI_Wait()将等待某一个请求，MPI_Waitany()将等待任意一个未完成请求的完成，MPI_Waitsome()将等待未完成请求列表中任意一些请求的完成。

返回值：

某个MPI错误码。

派生数据类型。将数据编组（marshaling）到连续缓冲区不一定都像2维SOR程序中那般直截了当。例如，数据可能由异构类型组成。对于这些情况，MPI提供了派生数据类型（derived data type）的概念，以允许程序员从非连续的缓冲区、或是从包含异构类型的缓冲区中传输数据。

方法是动态注册一个新数据类型，它需要我们告知MPI运行时（runtime）系统派生类型中各个组件的类型、大小和偏移。我们能使用MPI_Type_Struct()定义这个新数据类型，然后使用MPI_Type_commit()向系统注册这个类型。由于MPI_Type_commit()将创建新的运行时结构，因此调用MPI_Type_free()注销派生数据类型以避免内存泄漏（memory leak）将是一个良好的程序设计习惯。

例如，假设我们希望广播Person，这是一个由某人姓名和年龄组成的数据结构。图7-9显示了Person结构的实例如何广播到系统中其他的进程。具体地，在我们的BuildPersonType()函数中，第14-15行定义了该派生数据类型每个字段的类型，而第16～17行定义了每个字段中的变量数。

随后获得了每个字段的地址，从而能计算它们的偏移。最后调用MPI_Type_Struct()告知MPI运行时系统有这个新数据类型，并调用MPI_Type_commit()注册该数据类型。

给出定义派生数据类型的过程后，就能按如下所示的方式对它进行调用，然后将得到的派生类型作为参数传递给MPI_Bcast()：

```
Person* p=malloc(sizeof(Person));
p->age=44;
strcpy(p->name, "Nelson");
BuildPersonType(p, &newType);
MPI_Bcast(p, count, newType, RootProcess,
          MPI_COMM_WORLD);
```

性能测评（profiling）。Vampir是一个专门针对MPI程序进行性能分析的知名商业软件。它是在欧洲开发的，能提供显示每个进程以及聚合行为的图形化报告。

```
1  struct Person
2  {
3    int age;
4    char[6] name;
5  }
6
7  void buildPersonType(Person* p, MPI_Datatype* newType)
8  {
9    MPI_Datatype types[2];          /* 每个字段的类型 */
10   MPI_Aint  offsets[2];           /* 每个字段的偏移 */
11   MPI_Aint  addresses[3];         /* 每个字段的地址*/
12   int block_lengths[2];           /* 每个字段的长度 */
13
14   types[0]=MPI_INT;
15   types[1]=MPI_CHAR;
16   block_lengths[0]=1;
17   block_lengths[1]=6;
18
19   /* 以可移植的方式获得每个字段的地址 */
20   MPI_Address(p, &addresses[0]);
21   MPI_Address(&(p->age), &addresses[1]);
22   MPI_Address(&(p->name), &addresses[2]);
23
24   /* 计算每个字段的偏移 */
25   offsets[0]=addresses[1]-addresses[0];
26   offsets[1]=addresses[2]-addresses[0];
27
28   /* 定义和注册新类型 */
29   MPI_Type_struct(2, block_lengths, offsets, types, newType);
30   MPI_Type_commit(newType);
31 }
```

图7-9　创建派生数据类型

性能分析工具。许多年来，已经研发了一些针对MPI的调试和性能分析工具。这些工具的清单可在以下网站上获得：http://www-unix.mcs.anl.gov/mpi/tools.html。

7.1.7 安全性问题

MPI中没有共享数据，因此也就没有必要与Pthreads一样提供显式的互斥。无论如何，编写MPI程序是很困难的，因为有如此多的底层细节有待程序员来处理。例如，程序员必须在发送进程和接收进程双方都冗余地指定消息，因此程序员必须确保发送和接收能合适地匹配，这包括了消息的顺序，以及消息长度、类型和标记这些细节。而且消息的一些更高效变体，例如非阻塞式或非缓冲的消息通常极为难用。它们加入了许多关于定时和消息缓冲的额外假设，而这些假设必须由程序员加以实施。当然此处仍要处理死锁和活锁的问题。

消息传递总结。由于MPI提供了一组几乎能为任意并行计算机所支持的基础设施，所以虽然它迫使程序员处理诸多的细节，而且只提供了静态进程模型，但它仍得到了广泛采用。确切地说，MPI-2标准的新特性中支持了动态进程创建。但在制订标准的许多年后，仍然极少有它的实现出现。实现和采用它的主要障碍在于有大量的函数（超过500个）以及运行时系统更多的介入。

7.2 分区的全局地址空间语言

分布式存储器程序设计通常会与消息传递划上等号。但正如我们在第2章中曾提到的，在

分布式存储器的机器之上构建更高层抽象是有可能的，在过去的十年中，几个研究小组已成功构建了这种抽象，产生了称为分区的全局地址空间语言（partitioned global address space language）的语言家族，简称PGAS语言。

全局地址空间是指一些语言能在分布式机器的虚拟存储器之上形成单一地址空间。这些语言未提供共享存储器，因为并未期望硬件能使得共享存储器保持一致，但全局地址空间的确给了程序员定义全局数据结构的能力，这对于分布式存储器模型而言是一大改进，原先程序员必须手动维护独立数据结构的集合，使之类似于全局数据结构。在PGAS语言中，虽然程序员仍将关注于单个进程的行为，而且仍必须区分本地数据和非本地数据，但是语言通过消除消息传递的细节简化了程序设计，编译器能生成对应于程序员所说明的非本地访问的所有通信调用，而且它们使用了单边通信作为底层操作，潜在地这就比消息传递更高效。

三种主要的PGAS语言是Co-Array Fortran、Unified Parallel C和Titanium，它们分别扩展了Fortran、C语言和Java。现在我们逐一对它们进行考察。

7.2.1 Co-Array Fortran

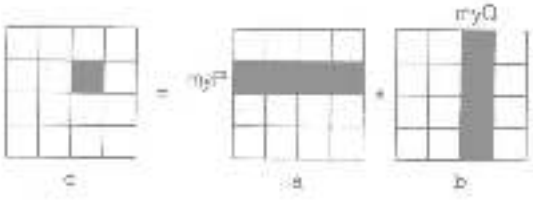
第一个PGAS语言是Co-Array Fortran（CAF），这个Fortran语言扩展由Numrich和Reid在20世纪90年代后期开发，CAF一开始被称为F--，以其精致而简洁闻名。其核心理念是为Fortran增加称为co-array（通信数组）的单个语言扩展，这是一种专为处理器间通信而设计的机制（"co"是communication的缩写）。注意Fortran对所有数组访问均使用圆括号，而CAF则通过在变量名后附加方括号[]来访问co-array。例如，以下代码使用co-array定义了3个数组变量，如黑体字所示：

```
real, dimension(n,n)[p,*]:: a,b,c
...

do k=1,n
  do q=1,p
    c(i,j)[myP,myQ]=c(i,j)[myP,myQ]+
                    a(i,k)[myP, q]*b(k,j)[q,myQ]
  enddo
enddo
```

根据co-array规范，存放变量的存储器是分布于各进程之上，这在CAF中称为影像（image）。当此生本例中，如dimension语句所定义的，每个进程各分配到数组a、b和c的一部分。此外，co-array声明为二维的事实表明，进程要在逻辑上布局为p行和q列的2维组织。（*符号表示进程的每一行都要尽可能地填满，因此若num_images()=pq，则表示有p行和q列的进程）。每个进程将通过调用this_image()例程在二维布局中找到自己的位置，并通过定义2个参数myP和myQ，访问最终数组的某个具体部分。至于数据划分的管理，包括初始化它们的值，则是CAF程序员的责任。

在以下图示中，myP=2，myQ=3：



以上矩阵乘的代码表明，由[myP，myQ]定义的各进程在计算点积时，需更新自己部分的数组值。为了访问行，它需要存取进程[myP，q]中的数据；为了访问列，它需要存取[q，myQ]中的数据。所有这些远程访问的通信调用均由编译器生成，这极大简化了程序设计。最初的CAF实现使用了Cray专有的、称为Shmen的单边通信库，但更近期的实现使用了ARMCI和GASNet，这两者都是标准的单边通信库。

CAF为分布式存储器并行机中用户自定义的通信提供了清晰而巧妙的接口。

7.2.2　Unified Parallel C

UPC是由El-Ghazawi，Carlson和Draper所组成的团队在2000年左右开发的，它为程序员提供了地址空间的全局视图。不同于CAF，当UPC数组变量被声明为“共享”时，它们将在程序各实例的存储器间分布。这些线性顺序的数组元素以循环或块－循环的布局分布。记得我们曾在第5章中提到，循环或块－循环分布在需要考虑负载平衡时会较有优势，但它们通常会损失局部性。UPC程序员可借助于直接向进程分配部分数组，以此恢复局部性，从而确保按密集的邻域方式进行分配。

由于UPC扩展了C语言，因此它能够支持指针。UPC的指针可以是私有的（即对线程而言是本地的）或是共享的。因为指针可以是私有的或共享的，又因为它们所指向的也可以是私有的或共享的，因此就有了四种类型的指针：

引用的性质	指针的性质	
	私有	共享
私有	私有－私有，p1	私有－共享，p2
共享	共享－私有，p3	共享－共享，p4

这些性质是与语言的类型体系紧密联系的，就如同它们的声明所表示的：

```
int *p1;                          /*私有指针指向本地*/
shared int *p2;                   /*私有指针指向共享空间*/
int *shared p3;                   /*共享指针指向本地*/
shared int *shared p4;            /*共享指针指向共享空间*/
```

以下向量－向量相加的代码，展示了第2种指针的使用方式，即私有指针指向共享空间：

```
shared int v1[N], v2[N], v1v2sum[N];

void main()
{
  int i;
  shared int *p1, *p2;
   p1=v1;
   p2=v2;
   upc_forall(i=0; i<N; i++, p1++, p2++; i)
   {
     v1v2sum[i]=*p1+*p2;
   }
}
```

这段代码还说明了如何使用UPC另一个特性，即upc_forall语句。该抽象使用了仿射性（affinity）子句，即upc_forall规范中的第4个子句，将普通C语言的loop循环分发到各进程。loop循环的仿射性子句指示了迭代执行的位置。在本例中，执行第i次迭代的进程即是与i相关的进程（即所有者）。upc_forall语句是个全局操作，然而绝大多数UPC代码却是在进程中本地执行的。

7.2.3 Titanium

Titanium（Ti）语言是由美国加州大学Berkeley分校Kathy Yelick所领导的团队开发的，它是一个能运行于分布式存储器并行机之上的Java扩展。它有一个类似于UPC的存储器模型。与其他PGAS语言一样，它基于单边通信库生成通信代码。Ti最明显的一个特征是它面向对象的本质，这使得它有别于它的先驱们。通过加入区域（region），又使得它有别于Java。区域能支持安全的面向性能的存储器管理，可作为无用信息收集（garbage collection）的一种替代方式。其他一些Java特性或被禁或被限，但更多典型的特性被加入进来。例如2维数组的加入就使得Ti更适合于科学计算。

Ti一个重要且高效的特性是它的无序迭代foreach，这同时简化了程序员和编译器的工作。其关键性的特性是点（point），它是一个能覆盖变量定义域（domain）（即它的索引）的整数元组（tuple），请注意，点在以下矩阵乘中所使用的名字是ij和k。

```
public static void matMul(double [2d] a,
                          double [2d] b,
                          double [2d] c)
{
  foreach (ij in c.domain())
  {
    double [1d] aRowi=a.slice(1, ij[1]);
    double [1d] bColj=b.slice(2, ij[2]);
    foreach (k in aRowi.domain())
    {
      c[ij]+=aRowi[k]*bColj[k];
    }
  }
}
```

因此，与forall相比，foreach允许在单个块内对多条索引执行并发操作。

Titanium使用障栅和一个在所有进程间共享、称为single的共享变量的概念来强制实现全局同步。例如，仿真中很常见的一种情况是各个进程在本地数据上计算，然后阶段性地调整它的操作。障栅能确保所有进程在同一时间停止计算以更新或读取存储器。以一致的值使用“single”变量能确保所有进程处于计算的同一阶段。例如，应用该概念后的粒子仿真可以用以下代码示出：

```
int single stepCount=0;
int single endCount=100;
for (; stepCount<endCount; stepCount++)
{
  读取远程的粒子
  在本地进程上计算力
  Ti.barrier();
  使用计算得到的新力写入自己的粒子
  Ti.barrier();
}
```

因为进程都为同一个值所控制，所以它们不能退出同步。

7.3 小结

毫无疑问，MPI和其他消息传递库最大的优势在于它们的普适性。将一块存储器从一个进程传输到另一个进程的能力对并行计算机是至关重要的，而消息传递库就为程序员提供了这些工具。消息传递库必须存在于所有的并行计算机上，这样这些库就能在理论上（也在实践

中）运行于任意并行计算机上。这种普适性是消息传递库能够流行并获得成功的最基本因素。

同理，MPI和其他消息传递库最大的缺点是它们抽象的底层化。并行程序设计是困难的，而且正如前几章中曾提及的，它的得益主要源自于对计算的抽象。而MPI对这些抽象只提供了初步的支持。例如，基本的归约操作只支持全局组合每个进程中的单项数据，而组合本地值的任务（即应用归约操作的概念到一个完全分布的数据结构）则必须由程序员自己完成。而且MPI只支持了基本归约操作符中一个很小的集合。但正如我们曾在第5章中看到的，抽象通常能提供更多的功能。

通过对底层硬件做最少的假设，消息传递有最少共同点的美名：即它能在任意平台上执行。虽然更高一层的语言，例如PGAS语言，依然在继续发展，但当程序员希望编写大型且生命周期长的应用程序时，现阶段MPI仍是不二之选。

历史回顾

早期的分布式存储器计算机各自都带有自己的消息传递库，这很不利于程序的移植。并行虚拟机PVM，作为一种能在连网计算机上执行并行计算的方法，是由Sunderam[1990]首先提出的。随着PVM的成功，MPI标准也很快跟进，并被迅速的广泛使用开来。在认识到一些由MPI的底层本性所引起的问题之后，1998年先后出现了CAF[Numerich 1998]和Titanium[Yelick 1998]，此后UPC[El-Ghazawi 2001]也很快问世。MPI 2.0标准在上世纪90年代后期制定，它增加了对动态进程管理和并行I/O的支持，但并没有被广泛采用，可能是由于这两个新特性使得对它的实现变得异常复杂。

习题

1. 执行统计3的个数的MPI程序（如图7-1所示），对于固定的处理器数P，试分析该程序在250 000个元素与10 000 000个元素之间的可扩展性。
2. 修改统计3的个数程序，通过使用基于图5-1中结构的自定义树，将调用替换为MPI_Reduce()。试与习题1的解决方案进行比较，看是否存在性能上的差异？
3. 在一个足够大的、能体现可度量行为的数组上，执行20次SOR计算（如图7-6所示）的迭代。然后采用重叠通信和计算的版本（如图7-8所示），再在相同大小的数组上执行相同次数的迭代。试比较它们的性能。
4. 编写一个MPI程序，使其中一个处理器产生足够大的唯一随机数键的集合（至少1百万），然后将它们分发到其他处理器。
5. 使用MPI实现第4章习题10中的红/蓝计算。
6. 在SMP或共享存储器多处理器上，使用OpenMP（第6章习题8）或Java Threads（第6章习题10）为第6章中的红/蓝仿真收集定时。与习题5的解决方案比较定时，并解释性能上的差异。
7. 使用MPI实现第4章中描述的Batcher's Bitonic排序，要求用阻塞式发送和接收。可利用习题4的结果初始化该计算。
8. 修改习题7中Batcher's Bitonic排序，使用非阻塞式发送和接收以重叠通信和计算。并与习题7的解决方案比较性能。
9. 先熟悉第8章矩阵乘小节中的SUMMA矩阵乘算法，然后在行广播和列广播中应用组的概念，实现SUMMA的MPI程序。
10. 使用派生数据类型的概念以及习题4的程序，创建一个程序，使它能将随机数分发到各处理器，并与随机生成的4字母的字符串配成对，然后返回给主处理器。

第8章 ZPL和其他全局视图语言

正如我们之前曾讨论的，Pthreads和MPI通过扩展标准串行程序设计语言的库，支持并行程序设计。这种方式所带来的优势是至关重要的：程序员已经很熟悉基础语言，因此他们能专注于学习库设施（facility）；库允许程序员使用相似的工具和开发环境；而且实现库比实现编译器要快。库的主要缺点在于它们既不能提供对并行程序设计的句法支持，亦无法从语言或编译器的支持中获益。

本章将描述全局视图语言，这种语言能让程序员“看”到全部计算，而不是关注于所构成的部分，比如单独的进程。这种高级语言能支持隐式并行，即编译器会管理进程的创建、通信和同步，这将极大地简化程序员的工作。

虽然经过了几十年的研究，但至今仍没有一种通用的高级并行语言在广泛使用。本章中我们将先介绍一种高级语言ZPL，它能体现如何从语言和编译器的支持中获益。本章最后，我们还会简单地介绍另一个全局视图语言NESL，它与ZPL有着截然不同的目标和特征。

8.1 ZPL程序设计语言

ZPL鼓励以不同的方式进行思考，方式之一就是关注数组和它们的操作。ZPL也提供了隐式并行，因此由编译器生成所有的并行进程或线程，插入所有必需的通信调用，并处理同步。例如要解决第1章中的统计3的个数问题，若使用ZPL，只需少许声明和单个语句即可完成：

```
[1..n] count:=+<<(array==3);
```

统计3的个数计算的ZPL代码

其并行解决方案是由编译器通过该语句创建的，能在任意并行计算机上执行，且性能可与尝试4[⊖]相媲美。稍后会解释该语句。

ZPL的一个目标是允许程序员推断并行性和并行性能，包括通信代价，并且将程序员从编写通信和同步语句的底层细节中解脱出来。例如ZPL程序员就永远不必担心竞态条件。ZPL的另一个目标是提供性能可移植性，再强调一下，语言的支持是很关键的。最后，我们会看到ZPL的句法如何在实现它的目标中扮演了重要角色。

我们对ZPL的初步介绍将从描述它的核心概念开始，而以Conway的生命游戏程序举例作为结束。在此之后，再回头来反观此语言与其设计目标。通过描述一些更高级的特性，我们继续ZPL的介绍。作为总结，我们会解释ZPL如何提供了一个性能模型，以允许程序员分析性能。

获得软件。 ZPL的编译器和文档可以通过开源的方式在http:// www. cs. washington.edu.cn/research/zpl上获得。编译器可以运行在Unix/Linux系统之上，而且重部署到新并行机上也较容易。

8.2 ZPL基本概念

ZPL是数组语言，这意味着整个数组将作为一个单元（unit）进行操作。因此如果要递增

⊖ 指第1章中统计3的个数问题的第4种解决方案。——译者注

数组A中的所有元素，我们可以写成：

```
A:=A+1;
```

或者也可以等价地写成：

```
A+=1;
```

这种对各个数组元素的更新在逻辑上是并行执行的。注意ZPL的赋值操作是:=而不是简单的等号=。

8.2.1 区域

修改数组中所有元素很常见，而只修改数组中某些特定元素同样也很普遍。为了说明在数组表达式中要操作的元素，ZPL要求所有的数组操作在区域的环境（context）中执行，如下所示：

```
[1..n]  A:=A+1;
```

方括号中的文字是区域（region），也即是索引集。正如稍后我们将看到的，区域在ZPL中是个有诸多用处的核心理念。假设声明数组A有n个元素，索引值为1到n，则以上语句将修改所有这n个元素，因为区域[1..n]指定了语句中所有数组元素的索引。而语句

```
[1..n/2]  A:=A+1;
```

只会修改A中前一半元素。

记法：*作为约定，ZPL程序员会大写数组名和区域名的首字母，以强调它们将访问许多元素，而小写其余所有的字母。*

区域形式。区域有多种形式。通常所显示的形式，下限为ll，上限为ul，只要满足$ll \leqslant ul$，就可以取中间的任意值；每一维的上下限用两个点（..）分开，而各维之间用逗号（,）分开，如下例所示：

```
[-100..100]    一个有201个索引值的线性数组
[1..8,1..8]    一个棋盘式的正方形数组（8×8）
[1..4,1..4,1] 3维空间中的一个平面，相当于[1..4,1..4,1..1]
```

如最后一个例子所示，维可以只由单个值组成，可称之为压缩（collapsed）维。边限ll和ul不必一定是常数，它们也可以是整数表达式，如下所示：

```
[min/2..2*max]
```

其中min和max都是标量变量，即单个数字值。

能描述子数组的区域。由于区域指定了参与计算的数组索引，因此所指定的索引必须存在于所有在语句中出现的数组中。不过这些数组倒无需具有相同的维。例如，假设B是$m \times n$，E是$m \times m$，若$m<n$；则：

```
[1..m, 1..m]  E:=1/B;
```

访问了E的全部，但只访问了B的$m \times m$子数组；而B的另外一些元素却没有被该语句访问到。

通过简单地访问各维都被压缩的降维区域，是有可能改变数组中的单个元素，虽然这样做并不常见，如：

```
[x,y]  D:=sqrt(2);
```

将单个元素D[x, y]设为1.414…。

声明中的区域。除了指定参与计算的数组元素，区域也可用来声明数组的大小。

例如，3个$m \times n$数组B，C和D，可声明如下：

```
Var B, C, D : [1..m, 1..n] float;
```

这些都是浮点数组。代码规范8.1列举了ZPL所支持的所有原始类型。

命名区域。在程序中反复书写相同的区域颇令人厌烦，因此可使用region声明来命名区域，如下所示：

```
region  R = [1..m, 1..n];
```

此后，该区域名就可在区域所能出现之处使用。它们能用在声明中，如：

```
Var B, C, D: [R] float;
```

也能用来指定数组操作的索引：

```
[R]  B:=2*C+D;
```

区域名的前后要加方括号。

代码规范8.1 ZPL中可用的原始数据类型

字节类型	2字节类型	4字节类型	8字节类型	16字节类型
boolean				
sbyte	shortint	integer	longint	
ubyte	ushortint	uinteger	ulongint	
		float	double	quad
		complex	dcomplex	qcomplex

前缀*u*表明所表示的数据是不带符号的，这可使它的精度增加一位。quad（四精度）类型仅在特定体系结构上的C语言中可用。否则它默认就是double（双精度）。*k*字节的complex（复数）类型使用了*k*字节表示实数部分，*k*字节表示虚数部分。

区域的范围。最后，区域是有词法范围的，也就是说，应用到某语句的区域是由最靠近该语句的区域说明所指定的。例如，在以下代码片段中，区域[R]前缀于包含了4个语句的repeat语句。其中区域[R]应用到语句s1，s2和s4，而s3则应用了不同的区域：

```
          [R] repeat
                s1;
                s2;
[2..m-1, 1..n]  s3;
                s4;
            until condition;
```

（代码规范8.2列举了ZPL中可用的控制流结构）。由于程序经常都会在许多形状相同的数组上进行操作，因此经常会为程序声明单个区域，并将其前缀于主程序块之前，这样所有语句都会在那种形状的数组上进行操作，除非用其他区域指定。

8.2.2 数组计算

代码规范8.3所列举的ZPL原始操作符，能应用于标量或数组操作数。当给定数组操作数时，操作符作用到操作数所对应的各个元素，它们必须具有相同的秩（rank）。例如，语句（源自我们稍后要讨论的程序）

```
[R] TW: =(TW&NN=2)|(NN=3);
```

作用到R中所有索引所对应的数组元素，就如同有许多以下形式的语句

```
TW[1,1]:=(TW[1,1]&NN[1,1]=2)|(NN[1,1]=3)
TW[1,2]:=(TW[1,2]&NN[1,2]=2)|(NN[1,2]=3)
TW[1,3]:=(TW[1,3]&NN[1,3]=2)|(NN[1,3]=3)
          ...
TW[m,n]:=(TW[m,n]&NN[m,n]=2)|(NN[m,n]=3)
```

在同时执行。(注意在代码规范8.3中等号（=）操作符用作测试是否相等，而不是赋值。)

代码规范8.2 ZPL中控制语句的语法

ZPL的控制流语句

```
if logical-expression then statements {else statements} end;
for var := low to high {by step} do statements end;
while logical-expression do statements end;
repeat statements until logical-expression;
return {expression};
begin statements end;
```

大括号中的文字是可选的。斜体的文字必须用相应类型的程序结构替换。

代码规范8.3 ZPL的原始操作符和赋值操作符

数据类型	操作符
数值	+(一元), -(一元), +, -, *, /, ^, %(模)
逻辑	!,&,|
关系	=,!=,<,>,<=,>=
位	bnot(a),band(a,b),bor(a,b),bxor(a,b), bsl(s,a)(将a左移s位，用0填补空处) bsr(s,a)(将a右移s位，用0填补空处)

指数操作（^）针对小幂的乘法进行了优化，例如2，但通常情况下都会编译成调用C语言的pow()函数。

可识别的赋值操作符是：+=，-=，*=，/=，%=，&=，|=。

@-转换。迄今为止，我们所举的数组计算实例都作用相同区域到各数组操作数。为了允许程序员对单独的数组操作数偏移索引，ZPL提供了@操作符，它采用数组和方向（direction）作为操作数。方向是一个与数组操作数有相同秩的向量。例如，为了允许数组A中的每个元素都能访问它的左边和右边，我们可以声明2个方向：

```
direction left=[-1]; right=[1];
```

当使用@操作符时，方向将根据所说明的向量转换数组的区域。如下所示，在语句

```
[2..n-1]  A:=(A+A@left+A@right)/3;
```

中，A使用了区域给出的索引集；A@left使用了比区域中索引小1的索引集，即1到n-2，A@right使用了比区域中索引大1的索引集，即3到n。与其他数组语言一样，赋值语句的语义将首先计算整个右边表达式，然后将结果赋值给左边表达式。据此可知，该语句的效果是将数组内部的每个元素替换为该元素本身与它左右邻居的平均值。

作为另外一个例子，可以如下声明8个罗盘的方向：

```
direction nw=[-1,-1]; no=[-1, 0]; ne=[-1, 1];
          we=[ 0,-1];             ea=[ 0, 1];
          sw=[ 1,-1]; so=[ 1, 0]; se=[ 1, 1];
```

如果TW是一个含有0或1的2维数组，则表达式

```
TW@nw + TW@no + TW@ne +
TW@we +         TW@ea +
TW@sw + TW@so + TW@se
```

将计算出一个数组，该数组中每个位置的值是其在TW中各邻居结点值为1的数目。（空格在ZPL中是被忽略的，因此各项的布局无论在声明上还是在表达上，都在试图提示转换的方向）。稍后我们将在一个实例程序中用到该计算，但在此之前我们要先解释另一个概念。

归约。回忆之前的章节，一个归约操作使用某个原始操作符，组合数组中所有的元素。我们说能“使用操作符将数组归约为单个值”。ZPL的归约操作通过如下形式给出：

```
op<<A
```

此处op可以是原始的结合和交换操作符之一：+，*，&，|，max和min。为了对A内部的元素求和，可以写作

```
[2..n-1]  total:=+<<A
```

注意，与所有ZPL中的数组操作一样，指定区域是很关键的。

归约能应用到任意秩的数组，因此为了找出2维数组B中的最大元素，可以写作

```
[R]  biggest := max<<B;
```

不一定非要保存标量结果。因此以下代码也是合法的：

```
[R]  span:=(max<<A)-(min<<A)+1;
```

归约操作是使用第4章中的Schwartz算法实现的。ZPL也提供了扫描操作符，使用句法+||A表示+ -scan。

8.3 生命游戏实例

为了进一步解释之前介绍的概念，我们把Conway的生命游戏（Game of Life）作为一个简洁的实例。生命游戏是一个很简单的计算，经常用作屏幕保护程序。

8.3.1 问题

生命游戏在二维网格中仿真生物的繁殖。初始设置为第0代。其规则如下：

- 在第i代时，如果某生物至少有2个邻居，但不多于3个，则该生物将生存到第i+1代。
- 在第i代时，如果某位置是空的，且其周围正好有3个邻居，则生物将在第i+1代时出生。
- 所有其他生物都将在第i+1代之前死去。

以上规则可以简化为单个条件：即某生物之所以能在第i+1代时存在，或是因为它生存在第i代时正好有2个邻居，或是因为它的位置（无论是否被某个现存生物所占据）在第i代时正好有3个邻居。

8.3.2 解决方案

我们使用称为the world的数组TW来解决这个矩形世界中的问题。生物用1位（bit）表示。为了知道某个位置有多少个邻居生存，就如之前所讨论的，将其周围最近的8个邻居求和到一个称为*邻居数目*（neighbor number）的变量NN中。我们使用了图8-1中所示出的逻辑。

```
 1  program Life;
 2  config const n : integer = 50;
 3
 4  region
 5    R   =[1..n,    1..n   ];
 6    BigR=[0..n+1,  0..n+1];
 7
 8  var
 9    TW:[BigR]     boolean = 0;          -- 世界
10    NN:[R]        integer;              -- 邻居数目
11
12  direction
13    nw=[-1, -1]; no=[-1, 0]; ne=[-1, 1];
14    we=[ 0, -1];             ea=[ 0, 1];
15    sw=[ 1, -1]; so=[ 1, 0]; se=[ 1, 1];
16
17  procedure Life();
18  begin
19    -- 初始化世界
20  [R] repeat
21    NN:=TW@nw+TW@no+TW@ne+
22        TW@we+       TW@ea+
23        TW@sw+TW@so+TW@se;
24    TW:=(TW & NN = 2) |(NN = 3);
25  until !(|<< TW);
26  end;
```

图8-1　实现Conway生命游戏的ZPL程序

8.3.3 如何实现

程序的前半部分由以下声明所组成：

- 第2行说明了数组的边界n，作为配置常量。这意味着它的值在一开始设定后就不会改变。初始设定或是默认值50，或是命令行上指定的值。
- 第4-6行声明了2个区域，BigR比R要大，因此它能保存边界值。这些边界没有生物存在，所以被赋值为0。这些值是@-访问所涉及的定义值，因此是必需的。
- 第8-10行声明了初始值为0的问题表示（TW），以及邻居的中间计数器（NN）。
- 第12-15行定义了访问最近邻居所需的8个罗盘方向。

程序所声明的2个区域进行了3次组合，可能这会诱使我们对其命名。但无论如何，我们都鼓励使用命名区域，因为这会获得与在其他程序设计语言中使用命名常量相同的益处。也就是说，命名区域为常量提供了附加的意义，这会使程序更加容易理解、修改和维护。

程序只有一个过程，称为Life（参见代码规范8.4）。在数组TW初始化之后（我们假设创建了随机设定，或是读入了输入文件），计算进入了数组计算上的repeat-循环。第1行：

```
NN:=TW@nw+TW@no+TW@ne+
    TW@we+       TW@ea+
    TW@sw+TW@so+TW@se;
```

通过将布尔数组类型转换成整数数组类型后再相加的方式，为每个数组位置计算活着的邻居数。这行代码可以理解为“加TW中西北方向邻居的元素值，再加TW中正北方向邻居的元素值，再加TW中东北方向邻居的元素值……”，也即是当ZPL程序员思考这类操作时，会从全局数组的观点出发，而不是从局部索引的观点出发。

下一行（第24行）通过应用Conway的规则创建下一代。组合这些规则到一条语句，我们得到：

```
TW:=(TW&NN=2)&(NN=3);
```

它对2个数组进行逐个元素的逻辑或计算，即正好有2个邻居的生物数组，以及正好有3个邻居的位置数组。

当一次循环结束后，终止检测将检查是否有生物仍然生存，如果没有，则退出。尤其是终止条件

```
!(|<<TW);
```

会在世界数组TW上计算or-reduce（或归约），即如果没有生物生存，则为false。它将结果取反以控制repeat循环。

代码规范8.4 指定ZPL的入口过程

> **入口点**
> ZPL需要有某个过程与程序同名，即与第一行program之后的单词相匹配，则该过程即是ZPL计算的入口点。

8.3.4 生命游戏的哲学

生命游戏很简单，ZPL的程序实现也很简单。程序的绝大部分由声明组成，一旦区域和方向声明好之后，真正的计算既简洁（两条语句）又清晰。简洁源自于数组操作的使用，它能同时操作整个数组。清晰源自于诸如区域和方向之类的抽象，它允许程序员以TW@nw的方式访问数组，而不是强迫他们直接处理每个单独的元素。数组操作符的使用也提供了隐式并行源，即编译器将生命游戏程序编译成高度并行的程序，使得几乎所有的底层细节都对程序员隐藏。

结束这个ZPL的初步介绍之后，我们现在已经准备好更深入地了解该语言的核心理念和设计哲学。

8.4 与众不同的ZPL特征

ZPL的目标是提供程序设计便利性和性能可移植性。那么它是如何实现这两个目标的？正如我们所看到的，它的便利性部分源自于它一次就能操作整个数组的能力。但也存在很多其他数组语言，包括APL和Fotran 90，那么ZPL到底不同在哪里？在本小节中当我们讨论使ZPL有别于其他语言（包括其他数组语言）、隐藏于ZPL之中的核心理念时，我们将会解答这些问题。

8.4.1 区域

ZPL的区域结构允许程序员在抽象层操作数组索引，因此索引集可命名和重用。此外能使用操作符对区域进行处理，如@操作符，这些操作符所注重的是不同数组访问之间的关系。

8.4.2 语句级索引

与其他大多数语言不同，区域应用到整个语句，而不是单个数组。这种方法的句法好处在于，它既能表现一个语句中多种数组表达式之间的通用性，同时又能突出不同表达式之间

的差异性。例如，在生命游戏的程序中，代码行：

```
NN:=TW@nw+TW@no+TW@ne+
    TW@we+       TW@ea+
    TW@sw+TW@so+TW@se;
```

明确地表示，通过定义8个方向，使得对TW数组8次访问中的每一次都能访问数组的不同偏移。程序员必须正确声明8个方向，但之后就能使用它们的名字以及它们所表示的含意。与之相对的是，在那些没有区域且将索引直接应用到独立数组的语言中，程序员必须为每个数组访问重复具体的索引计算，如在Fotran 90中就可能为：

```
NN(1:n; 1:n)=TW(0:n-1; 0:n-1)+TW(0:n-1; 1:n)+TW(0:n-1, 2:n+1)+
             TW(1:n;   0:n-1)+                TW(1:n,   2:n+1)+
             TW(2:n+1; 0:n-1)+TW(2:n+1; 1:n)+TW(2:n+1, 2:n+1)
```

除了区域的句法好处之外，一个更微妙且更深层次的语义好处在于它能适应并行性能。为了理解这个好处，必须先了解ZPL中使用区域的限制。

8.4.3 区域的限制

ZPL语句级索引限制了可表达的数组计算的类型。例如，单靠@操作符所提供的索引转换操作，不可能转置一个矩阵的元素，而这在几乎其他任意语言中，都能使用直接数组索引相当简单地表示出来。

```
for(i=1; i<n; i++)
{
  for(j=1; i<n; i++)
  {
    Atranspose[i][j]=A[j][i];
  }
}
```

这些限制其实是有意识地迫使程序设计准则将昂贵的操作（指并行执行中的通信）与廉价的操作区分开来。该准则也简化了并行性能模型的定义，而这对实现性能可移植性的目标是至关重要的。(ZPL另有一个操作remap可实现转置，参见下文。)

8.4.4 性能模型

隐藏于ZPL性能模型之后的基本理念是基于程序已声明的区域，说明数据分布。这种分布也指明了每个ZPL操作所需的数据移动，这就允许程序员依照句法来确定通信代价。因此ZPL程序员必须高度关注区域与其秩之间的关系，因为这种关系隐含着程序并行执行时的通信。作为交换，ZPL程序员无需再担心同步或通信的底层细节。而且ZPL编译器也被给予了许多重要信息，这能方便它完成多种不同通信的优化。

为了理解ZPL限定的必要性，将以下的循环嵌套与之前的转置代码相对照：

```
for(i=1; i<n; i++)
{
  for(j=1; i<n; i++)
  {
    Atranspose[i][j]=A[i][j];
  }
}
```

除了语句右边对索引的转置之外，这两段代码几乎是完全相同的。但就数据移动而言，两者是天壤之别，前者在每个进程中都需要显著通信，而后者却根本不需要通信。因此，若

允许前者的语言就不可能提供有意义的性能模型，因为没有一种容易的方法能区分开昂贵的操作与廉价的操作。

8.4.5 用减法实现加法

ZPL程序员也因此从失去的一些自由中换得了一些好处。尤其是，ZPL语言可以用一些代价更好定义、但较不通用的操作来替代那些代价变动很大、但较通用的操作，在之前例子中就是直接数组索引。

我们也看到了对数组秩的限定，即只有当数组有相同秩时，原始操作符才能操作这些数组操作数。例如，如果A被声明为1维数组，有索引[1..n]，那么它就不能与2维数组C的第1行相加：

```
[1, 1..n]  C:=C+A          不合法：A和C有不同的秩
```

这种限定对保持ZPL的性能模型是必须的。否则赋值语句的通信代价将依据数组C的分布是按行维、按列维、或是按行列两个维，而发生很大变化。

随后的两个小节中将介绍ZPL所具有的能改变数组秩的特性。例如，我们将看到如何实现矩阵的转置。然后将解释ZPL的性能模型，它是实现性能可移植性的关键所在。

8.5 操作不同秩的数组

在ZPL中，用来定义数组的区域，不仅会指定它的维数、元素数和索引，也会说明它的分配。不同秩的数组通常有不同分配，这将显著影响由不同ZPL操作符引入的通信。因此通常情况下，ZPL要求语句中所有的数组都有相同的秩。但在某些情况中，计算会产生不同秩的数组，在另一些情况中，不同秩的数组甚至必须在一起操作。对于这些情况，可使用两种基本思路来处理：

- 使用更大的秩　较小秩的数组能被声明成拥有与另一个数组相同的秩。秩较低的数组能通过增加一个或多个压缩维，转变成秩较高的数组。例如，为了在区域分别为[1..n, 1..p]和[1..m,1..n, 1..p]的数组上操作，可以将第一个区域转变成[1,1..n, 1..p]。
- 复制元素　当重复使用低秩数组的值作为高秩数组的元素时，低秩数组的元素就能被复制，这样它们就能和另一个数组的元素进行操作。ZPL的*扩充*（flood）操作符能高效地完成复制，而不必重复表示每个被复制的元素。

在本小节的后半部分，我们将看到这两种基本思路融合到“扩充维”的概念中。现在来介绍能改变秩的操作符，如部分归约，以及能支持不同秩的操作符，如扩充。

8.5.1 部分归约

归约操作将一个数组转换为一个标量，即，

```
sum:=+<<A;
```

求和数组A的元素以产生单个值。若该结果可以看成是对数组中所有维的“归约”或组合，则部分归约就可看成是对数组中某些维的组合。对于$m \times n$的数组B，我们通过组合列的值产生单个行，以此归约第1维；或者通过组合行的值产生单个列，以此归约第2维。此处B被称为*源数组*，而结果行或结果列则被称为*目标数组*。

不出意料，ZPL的部分归约使用了两个具有相同秩的区域。一个用来指定操作数数组的源

索引，而另一个用来指定结果数组的目标索引。源区域是作为归约操作的操作数指定的，而目标区域则是作用到语句的区域。为了使用加法操作符，沿第1维部分归约B（即将各列的值相加以产生行），编写如下：

```
[1,1..n] C:=+<< [1..m, 1..n] B;
```

此处“操作数区域”[1..m, 1..n]指定了数组B参与归约的索引（源），“语句区域”[1,1..n]指定了结果的索引（目标）。例如，对于$m = 3$和$n = 4$，则有

```
[1, 1..4] 7765  ⇔  +<<[1..3, 1..4] 3141
                                   1414
                                   3210
```

我们看到两个区域在第1维有所不同，因此该操作将第1维从3个元素归约到1个，而保留第2维不变。

为了使用乘法操作符归约B的第2维，即将各行的值相乘以产生列，编写如下：

```
[1..m, 1] D:=*<< [1..m, 1..n] B;
```

对于$m = 3$和$n = 4$，结果应是：

```
[1..3, 1] 12  ⇔  *<<[1..3, 1..4] 3141
          16                     1414
          0                      3210
```

本例子中，两个区域在第2维有所不同，因此3×4的数组将归约到只有3个元素的单个列。

注意到语句中的区域实际上只是为计算定义了主流环境，最接近的可用区域可作为另一个操作的操作数指定，如在$p \times m \times n$的数组F上有更复杂的操作，

```
[1,1,1. .n] G :=max<<[1,1. .m,1. .n] (min<<[1. .p,1. .m,1. .n] F);
```

找出第1维上值最小的平面，然后找出该平面各列中值最大的行，因此对于$m = 3$，$n = 4$，$p = 2$，示例如下：

```
[1,1..3,1..4] 2131  ⇔  min<<[1..2,1..3,1..4]  3141
              1312                             1414
              0200                             3210

                                               2434
                                               1322
                                               0503

[1,1,1..4]    2332  ⇔  max<<[1, 1..3, 1..4]   2131
                                               1312
                                               0200
```

此处最小归约组合了第1维，然后最大归约组合了第2维。

8.5.2 扩充

如同归约数组的维是有可能的，扩充数组的维也应该是有可能的。ZPL的扩充（flood）操作符（>>）通过在所指定的维上复制值，扩充数组的一个或多个维。由于扩充是部分归约的一个有效逆转（因而对数组大小会起到与部分归约截然相反的效果），因此扩充具有与之相似的符号，有：

```
[1..m, 1..n] B:=>>[1,1..n] C;
```

使用C中第1行的副本填充数组B，对于$m = 3$和$n = 4$，一个显而易见的例子是：

```
[1..3, 1..4] 7765  ⇔  >>[1, 1..4] 7765
             7765
             7765
                  ≠
```

与部分归约类似，所要扩充的维由源区域与目标区域之间的不同之处示出。

当然，扩充能应用到任意维，包括第2维，如

```
[1..m, 1..n] C:=>>[1..m,1] D;
```

这可以通过一个例子加以说明，对于$m = 3$和$n = 4$：

```
[1..3, 1..4]  12 12 12 12 ⇔  >>[1..3,1]  12
              16 16 16 16                16
               0  0  0  0                 0
```

此外，使用复制值也可以只扩充一维上的某个部分。

8.5.3 扩充的原理

扩充的目的是什么；为什么要复制值？扩充是必须的，因为逐个元素的操作符假设它们的操作数有着相同的维度，因而这些操作符只使用单个区域为它们的多个数组操作数说明索引。例如，假设我们希望按照矩阵第2列中的值，成比例地扩缩矩阵中的每一列。由于矩阵是2维的，列在概念上是1维的，因此我们复制列的值以产生数组的第2维。如此，这两个数组就能以逐个元素的方式进行除法操作。为此可以写成：

```
[1..m, 1..n] B:=B/(>>[1..m,2] B);
```

圆括号中的表达式用第2列的n个副本构成$m \times n$的数组；该语句的操作结果是数组中每一列的值都使用第2列的值按比例进行了扩缩。假设$m = 3$和$n = 4$，则有：

```
[1..3, 1..4] 3.00  1.00  4.00  1.00  ⇔  3141/1111
             0.25  1.00  0.25  1.00      1414 4444
             1.50  1.00  0.50  0.00      3210 2222
```

这些值只是逻辑上被复制。编译器并没有创建完整的复制数组；它只复制了所需的值，在本例中，每个进程处理一列。因此扩充操作是相当高效的，它有最小的数据移动。因而当访问扩充数组的值时，会有相当好的局部性。

8.5.4 数据操作举例

假设一个数据集合由针对m个对象的n个观测数据所组成，比如一天中消耗的咖啡杯数。我们可以使用一个维为[1..m,0..n]的数组D来表示这样的数据，此处数组被给予了一个额外的第0列来记录摘要（summary）数据。现在考虑在这些数据上进行一些示例性的计算。

要计算任意一天任意对象所消耗的最多咖啡杯数，即是在整个数组的数据部分上进行max-reduce（最大归约）操作。

```
[1..m, 1..n] most := max<<D;          --计算最高分值
```

变量most是一个标量。

每个对象的最大值即是各行部分归约的结果，我们将其存储在第0列中：

```
[1..m, 0] D:=max<<[1..m,1..n] D;      --记录单独的最大值
```

该计算产生了一列的值。

为了确定是否有不喝咖啡的人，我们在摘要列中检查0的个数，即在该列上计算or-reduce（或归约）：

```
[1..m, 0] tFans:=|<<(D=0);          --有人不喜欢喝咖啡吗?
```

变量tFans是一个标量。(当然，一个简单的and-reduce（与归约，符号为&<<D）也能起作用，但那样会不太容易理解。)

在所有人都喝咖啡的情况下，我们通过在数组上横向扩充第1列，然后用该结果除数据数组，可将每个人喝咖啡的习惯量相对于他们喝咖啡最多那天的量按比例缩小到范围[0,1]。

```
if !tFans then
[1..m, 1..n] D:=D/(>>[1..m,0] D);     --用最大值缩小
end;
```

最后来确定在这项研究中，每个人达到他（或她）喝咖啡最大量的天数百分比。首先将整个数据数组与1进行比较，然后使用加法操作部分归约每一行，以获得每个人达到最大量天数的计数，最后将结果除以n。

```
[1..m, 0] D:=100*(+<<[1..m,1..n](D=1))/n; --达到最大量天数所占的百分比
```

一些程序员可能不希望使用第0列来保存摘要值，而是愿意所声明的数据集合有更适宜的维[1..m, 1..n]，并用一个单独的Score（分数）数组[1..m,1]来保存摘要结果。当然这需要Score数组是二维的，因为它会在一些涉及2维数组D的表达式中出现。若使用这种方式，则之前的计算就相应地变为：

```
[1..m,1..n]  most   :=max<<D;
  [1..m, 1]  Score  :=max<<[1..m, 1..n] D;
  [1..m, 1]  tFans  :=|<<(Score=0);
             if !tFans then
[1..m,1..n]    D    :=D/(>>[1..m, 1] Score);
             end;
  [1..m, 1]  Score  :=100*(+<<[1..m,1..n](D = 1))/n;
```

借助这种计算，ZPL程序员能完成常规的数据操作，并能在保持给定秩不变的情况下，在不同数据大小之间来回切换。

8.5.5 扩充区域

在我们之前的例子中，变量Score的使用说明了我们在使用ZPL时的一个奇异特性。虽然在第一个解决方案中将摘要列置于第0列的位置是有意义的，但为什么必须要指定数组Score将其第2个索引设为1，它可以是0、9或n吗？答案是肯定的。这个决定对于存储器分配的真正后果会在稍后解释，现在我们可以认为这种选择通常是随意的。为了了解这种随意性，我们注意到在数组Score上的一个操作是扩充，即在所有列的位置上复制它本身。

ZPL提供了扩充维的概念，在区域表达式中用星号（*）表示。扩充维提示我们无需关心索引是什么，因为该维中所有的值都是相同的。因此上例中，定义摘要数组的最佳方式应为如下所示：

```
var Score : [1..m, *] float;
```

这说明数据在第2维上扩充。据此，之前例子中最后4条语句就变成了：

```
  [1..m, *]  Score  :=max<<[1..m, 1..n] D;
  [1..m, *]  tFans  :=|<<(Score=0);
             if !tFans then
[1..m,1..n]    D    :=D/Score ;
             end;
  [1..m, *]  Score  :=100*(+<<[1..m,1..n](D=1))/n;
```

无需在第3行扩充Score，因为该数据早已被声明为由扩充维的性质进行扩充。因此，已经存在逐个对应D中第2维上n个元素的值。那么Score在第2维上会有多少个元素？具有任意所需索引的任意所需数均可以。Score的值用图形的方式描述如下：

$\cdots, v_1, v_1, v_1, v_1, \cdots$
$\cdots, v_2, v_2, v_2, v_2, \cdots$
$\cdots, v_3, v_3, v_3, v_3, \cdots$
$\cdots, v_4, v_4, v_4, v_4, \cdots$
$\cdots$
$\cdots, v_m, v_m, v_m, v_m, \cdots$

因此Score能匹配第2维为任意大小的数组。

基于不应要求程序员指定超出他们所能的原则，扩充维实乃明智之举。但使用它还有另一个重要原因，即扩充维能帮助编译器使用高效的数据表示（以避免完整的数据复制）和高效的通信协议（多播通常是有可能的）。因此，选用扩充维通常会比随便选用某个压缩索引值要好很多。

8.5.6 矩阵乘

为了应用本小节中提到的思路，考虑计算两个稠密矩阵的乘积，C=AB，其中A是一个$m \times n$的矩阵，而B是一个$n \times p$的矩阵。在串行程序设计语言中，进行此类计算最简单的方法是使用三重嵌套循环：

```
for(i=0; i<m; i++)
{
  for(j=0; j<p; j++)
  {
    C[i,j]=0;
    for(k=0; k<n; k++)
    {
      C[i,j]+=A[i,k]*B[k,j];
    }
  }
}
```

最里面的循环计算点积，即A的第i行乘以B的第j列，然后归约产生乘积C[i,j]。

虽然这段代码很简单，但这并不是思考并行矩阵乘积的正确方式。事实上，van de Geijn和Watts认为，分别进行点积计算，即A的一行乘以B的一列，实际上是一种落后的方法。在他们的SUMMA（Scalable Universal Matrix Multiplication Algorithm，可扩展通用矩阵操作算法）方法中，他们将初始化和k-循环放到外面，而将所有点积的所有第k项高效地同时计算完成。这种逆向思维的方法产生了特别有效的算法，因为它使用了规则通信的模式。它也是最简单的ZPL矩阵乘算法，因为它利用了扩充。

为了理解SUMMA的核心思想，以及为什么扩充是它的基础。注意到在3×3矩阵C=A*B的计算中，其计算结果的前2列为如下所示：

$$C_{1,1} = A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} + A_{1,3} \times B_{3,1} \quad C_{1,2} = A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} + A_{1,3} \times B_{3,2} \cdots$$
$$C_{2,1} = A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} + A_{2,3} \times B_{3,1} \quad C_{2,2} = A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2} + A_{2,3} \times B_{3,2} \cdots$$
$$C_{3,1} = A_{3,1} \times B_{1,1} + A_{3,2} \times B_{2,1} + A_{3,3} \times B_{3,1} \quad C_{3,2} = A_{3,1} \times B_{1,2} + A_{3,2} \times B_{2,2} + A_{3,3} \times B_{3,2} \cdots$$

我们看到这些方程式中第1项的计算，可以通过横跨一个3×3数组复制A的第一列，纵跨一个3×3数组复制B的第一行，也即是说，扩充A的第1列和B的第1行，然后相乘对应的元素。第2项的结果可以通过复制A的第2列和B的第2行，然后相乘得到。以此类推第3项。

用ZPL实现的SUMMA算法如图8-2所示。这个程序以明显的变量声明开始。变量Col用来扩充A的列，变量Row用来扩充B的行。在过程的主体部分中，整个计算在结果数组C的维环境中进行。结果数组初始化为0，然后计算进入k-循环以处理点积的n个项。

```
var A    : [1..m, 1..n] double;
    B    : [1..n, 1..p] double;
    C    : [1..m, 1..p] double;
    Col  : [1..m, *]    double;
    Row  : [*, 1..p]    double;
    k    :              integer;

procedure MM();
[1..m, 1..p] begin
                C:=0;
                for k:=1 to n do
   [1..m, *]      Col:=>> [1..m, k] A;
   [*, 1..p]      Row:=>> [k, 1..p] B;
                  C+=Col*Row;
                end;
             end;
```

图8-2　使用ZPL实现SUMMA矩阵乘算法

在循环的主体部分中，A的下一列横向扩充Col，而B的下一行纵向扩充Row。在循环的最后一条语句中，2个扩充数组的对应元素相乘并累加到结果数组C。随后计算点积的下一项。

注意完全不必要使用临时数组Col和Row。使用扩充操作符之后，过程的主体部分可重写为如下所示：

```
[1..m, 1..p] begin
               C:=0;
               for k:=1 to n do
                 C +=(>>[1..m, k] A )*(>>[k, 1..p] B);
               end;
             end;
```

实际上，编译器仍然会为上述代码生成对应于Col和Row的临时数组，但这毕竟减少了程序员几行的输入。SUMMA算法不仅极其高效而且很容易编写。

代码规范8.5总结了对部分归约和扩充的限定。

代码规范8.5　ZPL中部分归约和扩充操作符的要求

部分归约（<<）和扩充（>>）

这两个操作都需要两个区域：源区域（作为一个操作数）和目标区域（作用到语句），如下例所示：

```
[1, 1..5] ... +<<[1..3, 1..5] A ...// 归约
[1..3, 1..5] ... >>[1, 1..5] A  ...// 扩充
```

对于两个区域中的每一维，索引的范围或完全一致，或其中一个是压缩维（单个值）。对于部分归约，根据目标区域中压缩维的提示，组合操作数的元素并赋值给压缩维。对于扩充操作，根据源区域中压缩维的提示，元素跨压缩维的范围进行复制。

8.6　用重映射操作重排数据

ZPL鼓励在本地数据上进行计算，但数据通常必须经过移动才能变成本地的。重映射

(remap) 操作符能完成数据的自主性重构，包括改变秩。在开始重映射操作符的学习之前，我们必须先介绍索引数组 (index array) 的概念。

8.6.1 索引数组

为了便于程序设计，ZPL提供了一组预先定义的常数数组，其内容保存了相关区域的索引值。这些数组被称为索引数组，记为index<*dimension number*>，如index1，index2，index3等等。例如：

```
[1..3,1..3]  ... Index1 ...  ⇔   1  1  1
                                 2  2  2
                                 3  3  3
```

是第1维索引的数组。再例如

```
[1..3,1..3]  ... Index2 ...  ⇔   1  2  3
                                 1  2  3
                                 1  2  3
```

是第2维索引的数组。使用索引数组的唯一限制是语句的区域要有足够的维数。

索引数组之所以能高效地实现，是因为它们的值在编译好的代码中已然可用，索引数组在ZPL程序中经常会用到，例如

```
[1..n, 1..n] Diag:=Index1=Index2;
```

用来构成一个对角线上元素均为1的数组，而

```
[1..n, 1..n] RMO:=n*(Index1-1)+Index2;
```

用来计算2维数组元素以行为主的顺序索引。索引数组也常和重映射操作符一起使用。

8.6.2 重映射

重映射操作符以符号#表示，定义了数组的重排。它需要两个参量，一个源数组和一个重映射数组。重映射数组含有定义了重排序的索引。重映射操作符有两种形式：称为聚集 (gather) 和散射 (scatter)。

聚集 (gather)。如果重映射操作符出现在等式右边，即作为一个表达式，则重映射就说明了一个聚集操作，结果数组将根据重映射数组的提示，从源数组中选出值。例如，假设A和P都声明在区域[1..7]上，且有值

```
A⇔ ddeeorr
P⇔ 5613742
```

则在语句

```
B=A#[P]
```

中，A是源数组，P是重映射数组，语句赋给B的值是

```
B⇔ ordered
```

因为B的第1个元素将是A的第5个元素，B的第2个元素将是A的第6个元素，以此类推。

散射 (scatter)。如果重映射操作符出现在赋值语句左边，则重映射操作符将完成散射操作，即源数组的值将根据重映射数组所说明的顺序，放置到结果数组中，例如，语句

```
C#[P]=A
```

将会产生

```
C⇔ererddo
```

即A的第1个元素将被放置到C的第5个元素中，依次类推。

重映射数组经常会使用基于索引数组值的表达式，例如

```
A#[8-Index1]
```

将源数组逆序，因为重映射数组包含了降序的索引值，有：

```
rroeedd⇔ddeeorr#[7654321]
```

为了拼写ordered的逆序，我们可写作A#[P#[8-Index1]]。

重映射数组中重复的值。重映射数组中的值无需唯一，虽然绝大多数情况下会要求唯一。例如，聚集A#[1111111]，即为

```
ddddddd⇔ddeeorr#[1111111]
```

就是一个令人厌烦（且昂贵）的方式，用以复制A的第1个元素。对于散射，这个问题就更加古怪了。因为在散射赋值中，重映射数组中的一个索引值可能会多次出现，这将导致赋值的不同顺序，从而产生不可预测的结果，因此散射形式为A#[1111111]:=A，有

```
?deeorr⇔ddeeorr#[1111111]
```

会将d，e，o或r中的某一个分配到第1个索引的位置，但到底是哪一个却无法确定。不同的并行执行会产生不同的结果。

高维。高维数组需要在方括号中有多个重映射数组，其中第i维的数组说明了第i维的索引值，例如，在

```
B#[C,D]
```

中，B中元素的重排，将使用C作为第1维的重映射数组，而使用D作为第2维的重映射数组，例如

```
[1..n,1..m] Btranspose:=B#[Index2, Index1];
```

是计算数组转置的标准方式，因为这2维的索引互相交换了。若Btranspose在区域[1..n,1..m]上声明，且$m = 3$和$n = 2$，则该数组的转置如下所示：

```
a c e  ⇔  a b #[1 2 3,  1 1 1]
b d f     c d   1 2 3   2 2 2
          e f
```

该聚集操作是清晰的，结果中的项i，j源自操作数中$C_{i,j}$，$D_{i,j}$的位置（参见代码规范8.6）。

代码规范8.6　ZPL中重映射操作符的需求

重映射

数组通过重映射操作符（#）进行重构，它需要2个操作数：欲重构的数组，以及方括号内对应于结果中每一维的数组。因此重映射2维数组A可写成A#[C,D]，而A#[Index2,Index1]可用来计算A的转置。重映射有两种形式：

聚集用于赋值语句的右边，结果中的项i，j源自操作数中$C_{i,j}$，$D_{i,j}$的位置。

```
a c e ⇔  a b #[1 2 3,  1 1 1]
b d f    c d   1 2 3   2 2 2
         e f
```

散射用于赋值语句的左边，操作数中的项i，j将进入到结果中$C_{i,j}$，$D_{i,j}$的位置。

```
e c a #[1 1 1, 3 2 1]  :=  a c e
f d b   2 2 2, 3 2 1       b d f
```

8.6.3 排序举例

重映射在ZPL中会经常使用到。一个绝佳的例子就是用它来重排2维矩阵的行。我们将在本小节中介绍这种计算。

回忆在数据操作这一小节中曾提到“喝咖啡的人”的数据数组。该数组在区域[1..m,0..n]上定义，记录了m个人在n天中喝掉的咖啡杯数,其中第0列用做摘要统计。例如，我们知道能通过如下的部分归约，计算参与者喝咖啡的平均消耗量：

```
[1..m, 0 ] D:= (+<<[1..m, 1..n] D)/n;
```

现在假设我们要根据他们的平均消耗量，从小到大排序喝咖啡的人。可以根据第0列的值重排数组D的行来解决这个问题，这意味着需要基于第0列来计算每个喝咖啡的人的排名。我们的策略是将这个任务划分成3个部分：

1. 使用扩充操作符做全比较，并计算排名。
2. 使用归约操作符找出排名。
3. 使用重映射操作符将数组的行以正确的顺序排列。

为方便起见，我们假设平均值是唯一的，但其实处理有相同值的情况也很容易。

全比较的排名算法。为了将序列中每个元素与序列中其他所有元素做比较，我们用保存在第0列的平均值对2维数组进行扩充，这给出了比较所需的其中一个操作数。为了获得另一个操作数，转置该数组，并在另一维上进行扩充。两个数组逐个元素比较所产生的结果将放置于一个位（bit）数组中。

从以下声明开始：

```
RepC: [1..m, * ] float;  --复制列的临时位置
RepR: [*, 1..m ] float;  --复制行的临时位置
```

然后用数组D中第0列的平均值扩充这两个数组。对于RepC，扩充是直截了当的，因为这两个数组是完全相适应的。对于RepR的第0列，就必须先转置成行，然后才能进行扩充。

```
[1..m, * ] RepC:=>> [1..m, 0] D;              --复制平均值列
[*, 1..m ] RepR:=>> [*, 1..m] D#[Index2, 0]; --复制平均值作为行
```

完成全比较则是件简单的事。

```
[1..m, 1..m ] ...RepC >= RepR;                --构建一个位数组
```

显而易见，必须很谨慎地选择合适的关系操作符。在对应于最小项的行中，>=操作符将只设置1位（bit）；而在包含最大项的行中，需要设置所有的位（bit）。

找出排名。为了找出喝咖啡的人的排名，我们使用部分归约，简单地在各行中累加1的个数。这将产生一个结果列，但由于将要在重映射中使用该结果，我们要的就不止是一个列而已，而是一个扩充了排名的数组。所以应包括如下声明

```
var Rank : [1..m, *] integer;
```

它使第2维成为了一个扩充维。

使用变量rank所声明的操作符，我们能在部分归约后执行扩充，

```
[1..m, *] Rank:= >> [1..m, *] (+<< [1..m, 1..m] (RepC >= RepR));
```

这产生了期待的结果。

使用排名数组进行排序。现在，通过使用Rank中的值，我们能使用重映射操作符对数组D的行进行重排。

```
[1..m, 0..n] D #[Rank, Index2] :=D;
```

这根据第0列中的值对行进行了排序。最终的程序如图8-3所示。

```
...
var RepC : [1..m, *] float;      --被复制列的临时数组
var RepR : [*, 1..m] float;      --被复制行的临时数组
var Rank : [1..m, *] integer;    --被复制的顺序
...

procedure rankingData();
  begin
       [1..m, 0]   D     :=(+<<[1..m,1..n] D)/n;                    --计算出平均值
        [1..m,*]   RepC  :=>>[1..m, 0] D;                           --复制平均值列
        [*,1..m]   RepR  :=>>[*, 1..m] D#[Index2, 0];               --以行方式复制平均值
       [1..m, *]   Rank  :=>>[1..m,*](+<<[1..m,1..m](RepC >= RepR));
                                                                    --计算顺序
    [1..m, 0..n]   D#[Rank, Index2] := D;                           --重排序
  end;
```

图8-3　排序喝咖啡者数据的ZPL程序

重排序操作是一个标准的ZPL范式。第一次接触时，似乎会感觉有些复杂，但第二次再接触时，尤其是搭建好框架之后，就很快会觉得这很自然。

8.7　ZPL程序的并行执行

介绍完ZPL的主要特征之后，接下来要介绍该语言特有的性能模型。但在此之前，我们还需要先理解ZPL的并行执行模型。ZPL程序的并行性是从它的数组语言语义继承而来的。数组中所有的元素都是同时操作的，因此数组能分发到多个进程上，并发地进行操作。

8.7.1　编译器的职责

ZPL编译器将数组语句转换成循环嵌套，例如，以下语句：

```
[R] TW:=(TW&NN=2)|(NN=3);
```

将被翻译为以下等价的C语言代码：

```
for(i=myLo1-1; i<myHi1; i++)
{
  for(j=myLo2-1; j<myHi2; j++)
  {
    TW[i,j]=(TW[i,j]&&NN[i,j]==2)||(NN[i,j]== 3);
  }
}
```

此处我们假设循环的各边界，如myLo1等等，已经对每个进程都已正确地初始化。若使用@操作符，则编译器会在循环嵌套前插入必要的通信。若是归约操作符，则将翻译为类似于第5章中给出的通用归约代码，以此类推。

为了生成高效的代码，编译器进行了诸多优化，包括以下措施：

- 它融合（或合并）了循环嵌套。通过将数组的临时变量转换成标量的临时变量，能极大地减少对存储器的需求。
- 它能为同一对进程间的消息组合通信操作。
- 它能重叠通信和计算。

- 它能有效实现扩充数组。因为被扩充维度的所有元素都是相同的，因此每个进程只需要一个值。
- 它能有效实现索引数组。因为编译器已经对在循环嵌套上进行迭代的数组索引作了描述，因此无需再用额外的存储器来表示索引数组。

除此之外，编译器还使用了许多其他更加复杂的优化，包括标准的串行优化。

本小节的余下部分将描述ZPL数据分布的细节，它是理解ZPL性能模型的关键所在。

8.7.2 指定进程数

由于ZPL程序中的并行性是隐式的，可供使用的物理并行总数需要在程序之外指定。尤其是，当程序在命令行上调用时，用户会指定进程的逻辑网格，比较典型的是2维网格。例如我们可能希望指定16个进程在逻辑上排成2行8个，即使用一个2×8的进程网格。

8.7.3 为进程分配区域

ZPL中，为进程分配数组的第一步就是为进程分配区域。程序的各区域是分布式的，因此对于所有的区域，相同索引将分配到相同进程上。为了达到这个效果，考虑将所有区域叠加起来，使得它们的索引能够对齐，如图8-4所示。通过这种叠加，它们的边界区域（即包含所有叠加区域索引的最小区域）就可计算出来。

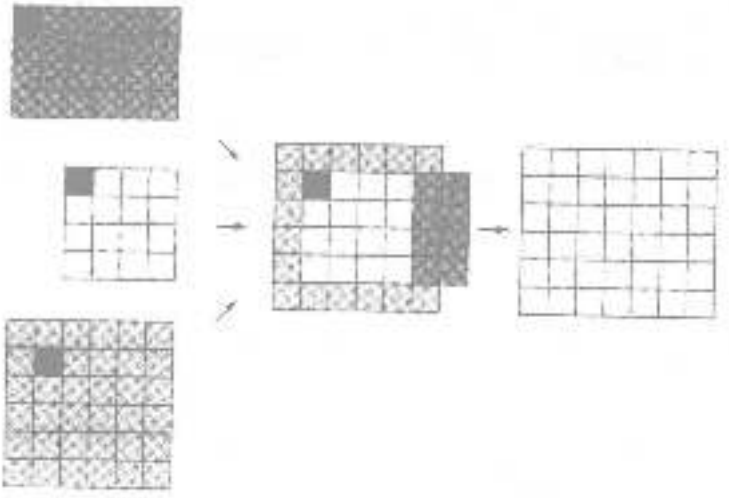
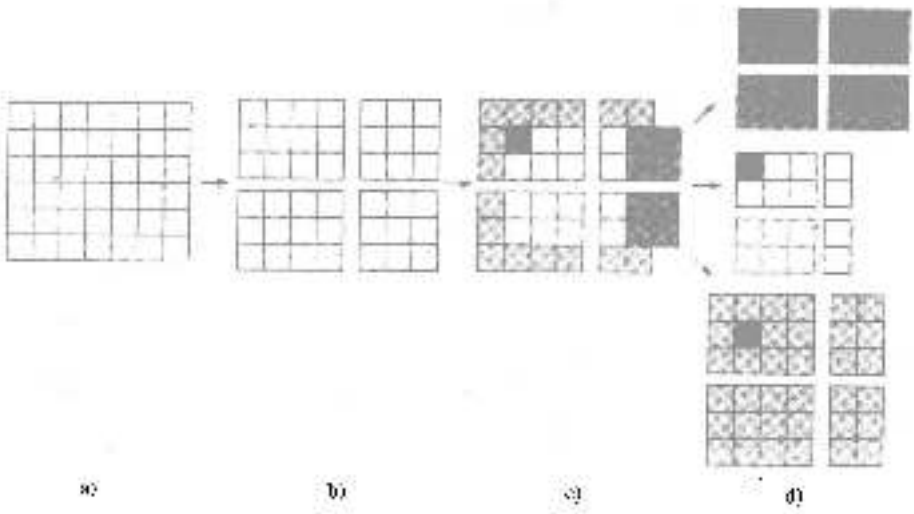


图8-4 边界区域。程序中使用的区域是叠加的，因此它们的索引可以对齐。黑色细线在所有区域中都有相同的索引。一旦对齐之后，边界区域就是能包含所有叠加区域索引的最小区域

一旦计算出来后，边界区域就可以分配到所说明的进程网格中。这种分配是一种块分配，以使每一维都尽可能的平衡。这些索引以一种显而易见的方法分配到所索引的进程上：以行优先的顺序，从低索引到高索引。因此可将图8-4中的边界区域分配到一个2×2的进程网格上，如图8-5所示。我们看到对齐所有索引的区域分配决策，只能产生一个次优的分配方案。不过这种效果的影响通常较为有限，且不常出现。作为另外一个例子，图5-9b中通过说明一个16×1的进程网格实现了16×16的区域分配。



a)　b)　c)　d)

图8.5　边界区域的块分配：a) 边界区域；b) 使用平衡分配进行分割　c) 分配一组索引　d) 所得区域的索引由那些索引组成而来

8.7.4　数组分配

给出了进程的区域分配之后，数组分配就顺理成章了，因为数组继承了它们所声明区域的分配。和C语言一样，数组的存储器是行优先的。当然编译器能在需要时分配额外的重叠区域。

例如，数组

```
var B, C, D : [1..8, 1..8] float;
```

分配到一个2×2进程网格。它会将子区域[1..4, 1..4]分配到P_0，这意味着数组B，C和D的相同索引也将分配到P_0；将子区域[1..4, 5..8]分配到P_1，以此类推。

8.7.5　标量分配

非数组变量是冗余分配的，即各进程都有所有标量的一个副本。因此标量计算，比如

```
i:=i+1;
```

是由各进程冗余计算的。这种冗余消除了通信（回忆第2章中随机数的例子）。另一方面，标量计算也不是通过并行提高性能的源由。

8.7.6　工作分派

一旦数组分配好后，工作分配就能很容易使用“谁拥有谁计算”的规则确定：即每个进程计算所分配到的元素值。因此区域不仅声明了数组语句中所使用到的索引，也定义了执行真实计算的进程。对于语句

```
[1..8, 1..8] A:=C+D;
```

进程P_0会使用C和D的本地元素来完成B的子区域[1..4, 1..4]的更新。这意味着，当数据在进程间平衡得很好时，这些语句的工作也会平衡得很好。因此更新该数组的工作将在4个进程上是够平衡地分配，因此可望有接近于4的加速比。

区域分配的原则　对齐区域的策略使得任意区域的索引[i, j, …, k]都将分配到相同进程

上，以此确保在数组上执行逐个元素的操作时，比如A+B，所有计算对于进程而言都是本地的，无需通信。

8.8 性能模型

在给出关于区域、数组和工作是如何分配到进程的描述之后，会很容易看清楚一个只有逐个元素计算的程序是如何并行执行的：它体现了完美的加速比。但其他ZPL结构呢？事实上，它们几乎同样容易理解。

ZPL的性能模型基于逐个元素操作规范所定义的代价，再加上某些操作所需通信开销的代价，比如@-访问和重映射，它们都需要从其他进程中获取值。该模型基于此种理念：即基本工作和通信开销这两个代价的概念，能构成对任意CTA平台上任意算法性能较好的第一近似值估计。该模型很容易使用，因为程序员能根据句法规则，确定代码中会产生通信代价的位置。代码规范8.7对这些代价进行了扼要的总结。

代码规范8.7 ZPL性能模型

ZPL性能模型规范针对的是最差情况的行为。除了计算机的物理特征之外，实际的性能还可能受到n，P，进程布局以及编译器优化的影响。

语法提示	示例	并行性（P）	通信开销	备注
[R]数组操作	[R]...A+B...	完全；work/P	—	
@数组转换	...A@east...	—	1个点对点	只发射“表面”
<<归约	...+<<A...	work/P+log P	2log$P$个点对点	扇入/扇出树
<<部分归约	...+<<[ ]A...	work/P+log P	log$P$个点对点	
||扫描	...+||...	work/P+log P	2log$P$个点对点	并行前缀树
>>扩充	...>>[ ]A...	—	维中的多播	数据不复制
#重映射	...A#[I1,I2]...	—	2个全体对全体（潜在）	通用数据重构

为了更加细致的观察代价模型，我们考虑以下操作：

- @-转换：操作数上的@修饰符表示数据会从相邻进程传输到本地重叠区域，但只有边界元素才会被传送。因此在CTA模型上，对于全部边界，这些点对点通信操作通常只需要λ的通信时延。
- <<归约：归约操作符使用Schwartz算法将数组值组合成一个标量，随后用广播方式将该标量值分发回给所有进程。因此通信模式是一棵组合树，再加上一棵广播树，每棵树的高度最多为log$P$，因而会产生2 λ log P的通信代价。
- <<部分归约：部分归约延用了完全归约中组合的概念，但没有广播，且限定为一维。
- ||扫描：扫描操作符使用并行前缀操作，因此会2次遍历高度为log$P$的树，1次向上，1次向下，因而会产生2 λ log P的通信代价。
- >>扩充：扩充操作符将保存的值分发到其他进程。多播（即对于进程子集的广播）若可用，就当使用。（使用特殊硬件通常速度会很快，但即便没有，广播也能使用树结构执行，实现logP的并发传输。）扩充操作符限制了通信复杂性，其主要特征在于如果进程在多维上分配，则只有这些进程的一个小子集能接受任意扩充。
- #重映射：重映射操作符在ZPL中是最昂贵的，因为它需要两个通信周期：一个用来分发通信模式（重映射数组），另一个用来分发数据本身。很有可能它们都是全体对全体

的通信，这意味着每个进程都有可能与其他每个进程进行通信。ZPL试图通过优化重映射来减少开销：这些例子包括，使用已出现的常数参量，比如在转置（A#[Index2, Index1])中，或重用上次重映射之后还未改变的重映射数组。

使用上述信息，就有可能大致清楚一条语句将如何执行。

8.8.1 应用实例1：生命游戏

编写生命游戏的程序时，我们会关注于产生正确的计算，但我们也能大致清楚程序将如何执行。请回忆其主计算是：

```
20  [R] repeat
21        NN:= TW@nw+TW@no+TW@ne+
22             TW@we+       TW@ea+
23             TW@sw+TW@so+TW@se;
24        TW:=(TW&NN=2)|(NN=3);
25      until !(|<<TW);
```

该循环主要包含了三部分计算：NN的计算，TW的计算和循环终止检测的归约计算。下面逐一对它们进行分析：

- NN的计算：该语句涉及了8个@-转换操作和之后的本地计算。根据代码规范8.7，每个@-转换操作需要λ的时延。因为CTA计算机有望能同时执行多个这种点对点通信，因此我们会付出常数级的通信代价再加上本地计算代价。
- TW的计算：该语句只需要在数组元素上的本地计算；无需付出通信代价。
- 或归约：循环终止表达式使用Schwartz算法，需要$2\lambda \log P$的时间。

此外，默认的块分配会实现合理的工作平衡，这意味着计算的加速比是P。因此就渐近性而言，随着n增加，在每次迭代都有$O(\log P)$通信开销的情况下，求解问题仍将继续享有完整的加速比；如果P增加，通信开销的增长也将维持在适当的范围内。

8.8.2 应用实例2：SUMMA算法

图8-2中矩阵乘的算法，其主计算的语句如下所示：

```
[1..m, 1..p] begin
               C:=0;
               for k:=1 to n do
                 C+=(>>[1..m, k] A )*(>>[k, 1..p] B);
               end;
             end;
```

如果忽略数组C一次性初始化的开销（它是完全并行的），n次迭代的各次迭代循环均有2个扩充操作，以及之后在本地元素上进行的乘－加计算。与之前一样，默认的块分配会实现合理的平衡工作分配，因此乘－加计算会完全的并行。如果我们将P个进程分配到一个$\sqrt{P} \times \sqrt{P}$的网格中，则每棵多播树实现扩充的高度只有$\log P/2$，因此这两个迭代的通信开销大致上为每次$O(\log P)$，这使得它成为一个高效的矩阵乘积解决方案[⊖]。

⊖ 这种分析可用于比较算法。最早宣布计算模型的论文比较了SUMMA算法与Canon算法，发现SUMMA更佳，之后的实验亦证实了这个预测。

8.8.3 性能模型总结

基于模型如何描述ZPL在CTA计算机上的执行，我们描述了如何估算一个计算的最坏时间复杂性。作为程序员，我们可把它作为可靠的、独立于机器的界限。虽然一些ZPL编译器的优化能产生更好的性能，但该模型确保了程序至少能实现该层次的性能。例如，编译器能通过移动通信调用，以重叠通信和计算。如果成功，可能可以完全看不到通信开销。而如果不成功，至少性能还是模型所预测的。

我们看到任何一个语言都能通过为每个操作简单地提供性能边界来定义性能模型，但回忆一下，在使用直接数组索引的转置代码中所遇到的困难：一个很小的句法变动就能导致通信代价的巨大变化。因此对于那些支持直接数组索引的语言而言，每个数组赋值会使通信的代价受制于转置的代价，这将导致特别不精确的界限。当允许程序员以不同的方式分发不同的数组时，对通信代价的预测将变得尤为困难。因此我们看到ZPL性能模型最核心的部分是语言对程序员所施加的一组约束。为此，如果一个语言想要有一个有意义的并行性能模型，必须在一开始就将性能模型设计到语言中。

8.9 NESL并行语言

嵌套并行语言（Nested Parallel Language，缩写为NESL）是上世纪90年代中期由美国卡内基梅隆大学Guy Belloch领导的一个小组所实现的高级数据并行语言。NES不严格地基于ML语言，使用了颇为直观的函数方式。我们对于NESL的兴趣源于两个方面：第一，NESL是另一种高级全局视图语言，它是专为数据并行计算而设计的。第二，NESL有一个复杂性模型，它能允许程序员评估他们程序的行为。

8.9.1 语言概念

NESL中基本的数据类型是序列，写在方括号中，

```
[6,14,0,-5]
```

以0作为起始索引。序列可以是字符，

```
"NESL allows sequences of characters"
```

用速记的方式写在双引号中。序列也可以包含子序列

```
["a" "sequence""can be made of " "sequences"]
```

假设所有元素都有相同的原子类型。

NESL的基本操作是并行apply-to-each（应用到每一个）的操作，在大括号中表示，例如：

```
{a+1: a in [6,14,0,-5]};
```

并行地应用增量操作a+1到序列中的每个元素，产生了[7, 15, 1, −4]。apply-to-each还能操作多个相同长度的序列：

```
{a+b: a in [6, 14, 0, -5]; b in [4,-4,10,15]}
```

这产生了[10, 10, 10, 10]。NESL定义了一个原始操作符的大集合。此外，还可以定义函数，以下代码说明了函数的定义，函数的使用以及+−reduce：

```
function dotprod(a, b) =
  sum({x*y : x in a; y in b});
dotprod([2, 3, 1], [6, 1, 4]);
```

8.9.2 用嵌套并行实现矩阵乘

嵌套并行定义为具有并行地应用一个函数到一组数据中各个元素的能力，以及嵌套这种并行调用的能力。这个概念可以一个简单的形式，通过如图8-6中所示的矩阵相乘函数来说明。

为了理解NESL的代码是如何运行的，观察一个$m \times n$的矩阵，它将表示为一个序列：有m行序列，每行中有n项，其中第1个序列就是矩阵最顶上的一行，以此类推。这种矩阵的转置会以一种显而易见的方式对项进行重新排序：第1行序列贡献了输出中n个新行序列的各个第1项，以此类推。

```
function matrix_multiply(A,B)=
  {{sum({x*y : x in rowA; y in columnB})
    : columnB in transpose(B)}
    : rowA in A}
```

图8-6　NESL的一个矩阵相乘函数

参见图8-6中的代码，我们注意到有3个apply-to-each的大括号：最外面的apply-to-each作用到输入数组A的行上，也即是说，每行的处理是独立的。下一个apply-to-each作用到列上，通过转置输入数组B将其转换成序列项；因此会对A的每行与B的每列进行配对且进行并行处理。这两个apply-to-each的操作结果可用来创建n^2个并行任务，每个对应于结果矩阵中的一个元素。最后，最里面的apply-to-each并行处理参量序列中对应项的点积。因此该apply-to-each结构的嵌套使用具有n^3的并行性。

可以认为该解决方案是对之前提到的三重嵌套循环的并行化，而非SUMMA方法。

8.9.3 NESL复杂性模型

NESL使用两种度量来确定一个表达式的复杂性：工作和深度。*工作*（work）是基本操作的数量，因此我们的矩阵相乘计算包含了$O(n^3)$的工作。*深度*（deep）是计算中最长的相关链。apply-to-each的深度是1，因为所有操作都能并行执行。求和操作sum使用二叉树实现，其深度是$O(\log_2 n)$，因为对叶子的计算必须依照叶子到根的顺序执行，这就引入了对数长度的相关链。

作为一种函数式语言，NESL通过抽象存储器单元和存储器访问的概念，揭示了大量的并行性，并提供了程序设计的便利性。复杂性模型能获得最适合的工作和最小的延迟。但是同时，函数式的抽象阻碍了NESL复杂性模型描述机器层次的细节，诸如局部性和数据移动，因此该模型并不允许程序员推断并行程序性能。

8.10 小结

我们介绍了高级数组程序设计语言ZPL。ZPL远比我们在此处所介绍的要多（我们只是介绍了称为ZPL经典的部分），它对于更复杂的计算，还有更强的控制功能可用。ZPL的隐式并行将程序员从处理诸多底层细节（如通信、同步和数据分布等）中解脱出来。同时它的许多特征都是经过精心设计的，用以支持一个精确的并行性能模型。该模型可应用到任意一台（包括CTA在内的）并行计算机上。因此ZPL是值得注意的，因为它体现了高层抽象是如此强大，能用来创建有效且可移植的并行程序。

NESL是一个更高级的语言。它使用函数式程序设计风格来表现并行性，还有简洁的语法能允许程序员说明并行性，而无需引入相关性。

历史回顾

ZPL语言是由本书作者和他们的一流团队在美国华盛顿大学研发的。这是第一个能在所有并行平台（MIMD）上获得高性能（即实现性能可移植性）的高级语言。其最主要的贡献之一就是所见即所得的（WYSIWYG）性能模型[Chamberlain 1998]。其他相关引用可在ZPL的网站上获得。NESL[Blelloch 1996]早已是许多精巧算法的基础；NESL的网站提供了一些实例以及一个简短的教程。

习题

1. 使用3 × 3的数据数组，针对数据操作这一小节中所提到的ZPL计算，手动计算例子的值。
2. 修改咖啡数据中行排名的排序，以处理有重复值的情况。
3. 将Conway生命游戏的ZPL实现程序作为规范，编写MPI程序实现生命游戏，假设使用2维数组划分。
4. 编写ZPL程序，实现第4章习题10中所描述的红/蓝计算。
5. 使用ZPL性能模型，分析习题4中关于红/蓝计算的性能。注意该解决方案可能会与Conway生命游戏的分析有许多共同特征。
6. 以类推SUMMA算法的方式进行思考，使用扩充操作求解所有对的最短路径问题。n个顶点的图输入将被表示成一个$n \times n$的邻接矩阵，用“无穷大”值来表示那些不能直接相连的位置。
7. 使用ZPL性能模型分析习题6中所有对的最短路径计算的性能。
8. 使用扩充操作符用ZPL编写3维矩阵乘。将A和B的矩阵扩充为复制值的立方体，执行所有元素对的乘法，然后归约到一维以生成结果，并将其赋值给矩阵C。注意重映射操作符会需要对数组输入时的定位重新进行定位。
9. 应用ZPL性能模型分析习题8中的程序。在考虑性能时，可使用3维进程网格。
10. 基于习题9，使用2维进程网格时，此时数组将按它们的“法线”方位进行分配，分析该解决方案的性能。
11. 编写一个NESL程序，实现第4章习题10中所提到的红/蓝计算。
12. 解释试图用NESL实现SUMMA算法时可能遇到的问题。

第9章　对并行程序设计现状的评价

正如第1章中曾提到的，对于并行程序设计，并行计算社区尚未能解答所有的疑问。因此本书在探讨特定的程序设计方法之前，先关注于基本的原则。在本章中，我们将对并行程序设计的现状进行广泛的调研。我们会先评估现有的一些语言，然后从这些语言中总结出可供将来借鉴的经验。

9.1　并行语言的四个重要性质

在开始评估不同的程序设计方法之前，我们首先定义了四个重要性质，它们将贯穿于这里的讨论：

- 正确性（Correctness）
- 性能（Performance）
- 可扩展性（Scalability）
- 可移植性（Portability）

这个列表并不完整，还有其他一些值得考虑的性质，但这四个性质对于并行程序员而言是最重要的需求。

9.1.1　正确性

虽然编写任意一种正确的程序都是不容易的，但这对于并行程序而言就更为困难。这是因为并行程序对程序执行的定时（timing）特征比较敏感。串行程序通常是可重现的，因此两次相同输入的执行自然能产生相同输出的结果。而并行程序则不然，它的行为可以由于定时的不同而发生很大不同。因此对于任意并行程序设计系统而言，首先要问的一个问题是，能否消除这种敏感性？如果能，在正确性方面，并行程序就有可能可以简化到近似于串行程序的程度。

*P*独立（P-Independence）。上述思考最终促进了*P*独立概念的产生。

P独立　一个并行程序是*P*独立的，当且仅当在相同的输入下，无论运行它的进程数量或布局（arrangement）有多大的不同，都能产生相同的输出；否则该程序就是*P*相关（*P*-dependence）的。

*P*独立并不能保证浮点算术的精确性。*P*独立程序的几次不同运行会在一些位（bit）上产生不同的结果。然而*P*独立能在数量不同的并行出现时，保证控制流的可重现。

一个并行的“hello world”程序如果只打印一次“hello world”，则它是最容易的*P*独立。从另一个极端而言，一个并行程序输出运行它的进程数是最容易的*P*相关。

*P*独立的概念是非常重要的，因为我们认为程序员通常都会想要编写*P*独立的程序；但当语言抽象本身就是*P*相关时，程序员就必须设法压制与进程数量相关的敏感性。因此是*P*相关复杂化了并行程序的构建。

全局视图抽象和局部视图抽象。通过把程序设计抽象分成两类，就可以将*P*独立的概念应用到语言中：

- 全局视图抽象（Global view abstraction）：一个能保持*P*独立程序行为的语言结构，表现出了全局视图抽象
- 局部视图抽象（Local view abstraction）：一个不能保持*P*独立程序行为的语言结构，表现出了局部视图抽象

因此，使用全局视图抽象的语言称为*全局视图语言*，使用局部视图抽象的语言称为*局部视图语言*。全局视图语言更容易调试，因为它们能基于串行执行进行调试，而非基于并行执行的任意一个实例（instance）。

【例】考虑下列在前几章中曾提及的一些程序设计抽象。

- 锁：使用锁的两个线程可能会导致死锁（参见第6章），但是同样的程序使用一个线程执行，通常就不会导致死锁，因此锁是局部视图抽象。
- 发送/接收：使用阻塞式接收操作的程序，能在多个进程上正确执行，但在单个进程上执行时会导致死锁，因此接收操作是局部视图抽象。
- forall循环：如OpenMP中使用的那些并行循环，无论用来执行循环的线程数量是多少，都能提供相同的结果，因此forall循环是全局视图抽象。
- 障栅：障栅能同步所有线程或进程的执行。虽然执行障栅的时延与参与同步的线程或进程的数量相关，但该操作并没有产生与线程或进程数量相关的副作用，因此障栅是全局视图抽象。
- 归约和扫描：归约和扫描的语义是基于数组的元素定义的，与数组所在的线程或进程的数量无关，因此归约和扫描是全局视图抽象。但如果用户自定义的归约和扫描有副作用，则这些操作就必须算是局部视图抽象，因为副作用是与参与操作的线程或进程的数量相关的。

【例】考虑下列语言：

- ZPL Classic：曾在第8章中介绍过的ZPL语言核心，也称为ZPL经典。这是一种全局视图语言。
- NESL：NESL是一种全局视图语言。
- 消息传递库（Message Passing Libraries）：消息传递显然是一种*局部视图语言*，比如Co-Array Fortran和其他大部分的并行程序设计工具。

9.1.2 性能

性能要有多快才足够？虽然我们在本书中曾提到实现*P*倍的并行是目标，但回答这个问题却要依据具体情况。例如，在一个双核处理器的系统中，根据不同情况，程序员可能会对1.2、1.9和2.1的加速比都感到满意：

- 当应用本身只具有很少的可并行性，而且并行的目标也仅仅是使用另一个空闲处理器时，1.2这个适中的加速比也是能令人满意的。毕竟最后的结果是不可忽视的20%的性能提高，而这是通过其他方式都不能轻易获得的。
- 如果程序员仔细地构建了程序，消除了大部分的并行开销，利用了可用的并发性，那么获得接近2的加速比也是在情理之中的。
- 当计算表现出不错的访问局部性时，超线性的加速比也是有可能的。这是因为两个核加

起来有比各自都要大的L1 cache，这使得整个系统能容纳更大的工作集。但是在多核芯片系统上取得超线性加速比，比在其他多处理器系统上通常要难得多，这是因为两个核共享了一些存储器资源（如Core Duo中的L2 cache以及所有多核芯片中的存储器带宽）。我们认为在性能评估中需要了解情况，并对性能有个合理预期。

9.1.3 可扩展性

性能不仅与应用相关，也与硬件相关。尤其是，当处理器数增加时，会越来越难保持相同级别的加速比。因此在许多情况下，可扩展的并行性并不是必需的目标。特别当短期目标是运行在某个特定的硬件平台上时，比如某个有固定数量处理器的多核芯片，就尤为如此。

但多核芯片上核的数量似乎增加得很快，基本上每18个月能翻一番，当然前提是如果CPU供应商能继续遵循Moore定律。因此任何长期使用的软件，也即是成功的软件，都必须要能扩展以适应这条性能曲线。一旦写好一个可扩展软件，它就能在没有程序员特别的干预下，跟随硬件的发展，如同串行程序在那个CPU主频快速增长的年代中所做的一样。

9.1.4 可移植性

除了可扩展性，可移植性对于长期使用的软件也是至关重要的，这是由于并行计算机硬件的差异性。我们要特别强调性能可移植性（performance portability）这个概念的重要性。它是指一个并行程序在多种不同类型的并行计算机上，只需要通过一些适当的优化（tuning），就能获得较好性能的能力。

性能可移植性可以通过在逼真的抽象机模型上设计算法来获得，这也是为什么会引入CTA的原因。抽象机模型把假设局限于普适的概念，建议了不同规模下的机器实际可达到的开销。通过在一个逼真的抽象机模型上进行程序设计，程序员可以将他们的代码移植到不同体系结构的并行计算机上，并且可以期望程序的核心性质与新硬件是相兼容的。虽然调谐是必需的，但算法就可以不必完全重新考虑。

与此相反，程序员直接针对某些特定的目标硬件，可能会导致在移植到不同平台时，被迫改变算法。这是由于代码可能依赖于一些不再支持的特性，或是采用了原先硬件中某些特殊的性能特性。程序员在开发算法时必须假设某个平台，因此这个平台也可能就是逼真的抽象机模型。

使用某种可移植的底层，比如MPI，并不能保证性能可移植性。如果可移植的底层提供了一些低等级的结构，则它很有可能在不同硬件上有不同的性能表现。

9.2 评估现有方法

我们现在开始评估第6章至第8章中讨论过的几种程序设计方法。

9.2.1 POSIX Threads

Pthreads（POSIX Threads）和其他基于锁的方法都提供了强大的能力和灵活性。但这些显然是*P*相关的方法，提供了太多的灵活性。此种灵活性影响了正确性和性能。

这个问题部分是由于线程能在宽范围的接口之间进行交互。因为是共享地址空间，任何语句都有可能与其他语句进行交互。尤其是，任何一个存储器访问都有可能引入与其他线程

的相关性。例如在调试Pthreads程序的一个竞态条件时，就可能会涉及到程序中任意一个语句。与此相反，在MPI中调试一个竞态条件时，则只会涉及到MPI例程，因为这是进程之间交互的唯一途径。（当然MPI也有自己的难处，我们稍后会讨论到。）

宽范围的接口也会影响性能。我们认为通过识别出跨线程的相关性来推断并行性能很重要，但由于与其他对存储器读和写的相关性相比，跨线程的相关性看上去并没有区别，因此共享地址空间并不鼓励局部性。

锁和条件变量的特性也是容易出现问题的。使用它们就会导致强迫程序员推断所有可能的交互和交叉（interleave），因为这两种结构是基于状态变化的定时。更具体的说，锁和条件变量结构从两方面破坏了模块化和抽象的目标：（1）基于锁的代码是不能组合的，（2）锁对于性能和正确性而言都是全局性的。两段各自能正确执行的代码，组合在一起后就有可能发生错误。例如，如果两段代码都没有用锁保护共享数据，则可能导致竞态条件；如果它们各自以不同顺序获得同一集合的锁，则可能产生死锁；如果它们访问同一cache行上的变量，则可能导致假共享。出于性能的原因，以全局方式考虑锁也很重要，这是因为锁的粒度能影响并行性和性能。例如在某种情况下对某个数据结构提供互斥访问是合适的，但当它与其他数据结构组合后，我们可能会需要一个更粗粒度的锁。

不仅把锁行为的细节隐藏到模块中是不可能的，将锁的需求汇总到一个接口中也是很困难的。仅仅知道哪些锁在使用是不够的。为了避免死锁，我们需要知道这些锁被获取的顺序。为了优化性能，我们需要知道这些锁被用在哪里。总而言之，锁和条件变量，与模块化是格格不入的。

对于共享地址空间方法，一种观点认为它与串行程序设计的相似性可以使得串行程序以增量的方式移植到并行平台上。另一种相对比较激进的观点则认为这种模型是在鼓励程序员编写低效率的并行程序。由于它与串行程序有着太多的相似性，因此需要花费巨大的精力才能将其转变成高效率的并行程序。我们认为由于并行程序设计与串行程序设计截然不同，因此并不欣赏那种“慢移植”的方式。

9.2.2 Java Threads

Java提供了可供程序设计的多个层次。底下一层与POSIX Threads很是相似，因此它就具备了与POSIX Threads几乎相同的优缺点。上面一层的接口创建了同步的对象和方法，以及一些隐藏了很多底层接口复杂性的并发数据结构。它有一些性能上的问题，因为它限制了程序员对并行粒度的控制能力以及隐藏时延的优化能力。虽然在有些例子中这些问题并不重要，但是通常拥有更好的控制对实现性能和可扩展性的目标是必须的。

9.2.3 OpenMP

OpenMP是一种全局视图语言，它限制程序员只能使用最简单的并行形式，因而就特别容易使用，但许多并行形式就不能在OpenMP中体现出来。如果与Peril-L语言相比，OpenMP仅仅提供了forall语句、障栅和归约的功能，而且它还不允许类似细粒度同步操作的并发形式。

9.2.4 MPI

MPI只允许进程间通过MPI调用进行交互，因此与Pthreads相比，MPI提供的接口范围更窄。窄范围的接口通常会比宽范围的接口更加容易考虑。但在使用MPI时，由于程序员所需

说明的底层细节是如此之多，从而导致了此种说明既繁琐又易错。尤其是，由于程序员必须为每个参与通信的进程冗余地说明每个通信操作，从而导致了许多出错的可能。

虽然MPI的点对点通信例程是*P*相关的，但MPI在集合通信操作中却的的确确提供了通信抽象，这包括了归约、扫描以及用户自定义的归约和扫描。这些集合操作在MPI程序中被大量使用，这可能是程序员将计算看得更为全局化的极少数方法之一。

作为分布式存储器程序设计模型，MPI迫使程序员同时思考多个不连续的地址空间。因此程序员有时需要判断是否需要在某个进程上执行一段本地计算，但在另一些时候，他们需要转换思维，将这部分计算与其他进程上的计算关联起来。与此相反，Pthreads,OpenMP和ZPL都提供了单一地址空间，因此它们只需要程序员进行较少的思维转换。

最后，虽然MPI程序几乎能在任意并行计算机上编译和运行，具有可移植性，但MPI程序却不能保证性能可移植性。这是由于接口的层次是如此之低，以至于间接暴露了对底层硬件的假设。尤其是，针对某一种并行计算机，满足其特定点对点通信模式的优化，通常对另一种并行计算机却不是最优的。当然如同Java Threads能在Pthreads之上提供更简单的接口一样，另一种更高层的语言也可以构建于MPI之上，因此可以将MPI看作是更高层语言的通信基础。基于这个想法，我们的问题是，哪种是构建于MPI之上最为合适的语言？

9.2.5 PGAS语言

分区的全局地址空间（PGAS）语言基于MPI做了改进，它们为特定的通信提供了更高层的机制，而这些机制能在分布存储器的并行计算机上有效实现。这类语言最关键的优势在于为程序员提供了全局视图，从而使得程序员不必再局限于对单个进程的考虑，因为覆盖于分布存储器之上的全局地址空间，能支持全局数据结构的定义。

这三种语言还引入了一系列聪明的想法以简化程序设计。我们从每个语言中各挑出一个想法来介绍：Co-Array Fortran中的co-array是一个非常精致的机制，它能访问非本地存储器，并且能很干净地嵌入到语言中，几乎不需要增加额外的概念。UPC的upc-forall语句能根据程序员指定的相似性来分配迭代次数，这给予了程序员在由编译器生成的全局操作上的某种控制。Titanium无序的foreach迭代子（iterator）在处理涉及多维数组的程序时，能显著简化程序员和编译器的工作。

尽管有全局地址空间，但PGAS语言的主要问题仍在于它们都保留了过多的局部视图。程序员会首先关注于编写本地程序，然后将其复制到多个进程上。程序员仍然必须将计算作为更大解决方案的一部分来管理。例如，处理所有“边界情况”的责任就都落到了程序员身上。这种本地化计算的后果是保留了太多低层的细节。消息传递的机制虽然消失了，但局部视图所需承担的责任和负担却依然存在。

9.2.6 ZPL

ZPL通过提高抽象的层次提供了程序的全局视图，以此简化程序设计。经典的ZPL[⊖]是全局视图语言。ZPL提供了一组丰富的高层并行抽象，包括：数组语言的语义、区域、扩充、归约和扫描。这些抽象结构隐藏了通信和同步的底层细节，因此抽象的结果使ZPL程序变得相当简洁。

⊖ 一些更高层的语言结构（并没有在本书中描述）能产生非*P*相关的结果。

ZPL鼓励程序员通过使用数组操作进行并行风格的思考。它的新抽象（比如扩充）已经催生了一些新程序设计技术的出现，比如求解问题空间的提升（Problem Space Promotion）（参见第10章）。虽然ZPL表现出了高层抽象，但它通过构建在逼真的机器模型（CTA）之上，提供了性能可移植性。基于该机器模型研发的性能模型可用来描述ZPL操作的执行开销，并允许程序员以独立于机器的方式推断性能。ZPL还鼓励通过它的可扩展并行抽象创建可扩展程序。

ZPL几个严重缺点包括：陌生的概念（区域，扩充，重映射和数组程序设计等）；缺少基于指针的数据结构；功能有限的动态存储器分配工具。另外它还不支持现代程序设计方法学，如面向对象程序设计。

9.2.7 NESL

与其他函数式语言一样，NESL的访问透明性使程序更数学化，也更容易考虑。它的全局视图抽象有效的隐藏了并行性。但是与其他函数式语言一样，由于抽象掉了存储器访问，NESL程序就很难分析数据的移动和局部性，以及用来获得良好并行性能的关键性特征。类似的，NESL鼓励用无限并行性方法来设计算法。例如它能很自然的实现矩阵乘的三重嵌套循环，这获得了最大的潜在并行性。与此相对，虽然SUMMA算法是一个很好的实用性并行算法，但用NESL实现就较为困难。

9.3 可供将来借鉴的经验

由于理想的并行程序设计设施还没有被开发出来，因此我们基于过去的经验，考虑未来的这个系统应当具有哪些特性。

9.3.1 隐藏并行

第6章至第8章都清晰地表明了并行程序设计有多难。读者或许会问“是否有必要像那样设计程序，以此获得并行的好处？”，答案是“是，也不是”。

并行很久以来都是成功的，只是因为之前一直在底层，对程序员隐藏而已。最主要的例子是挖掘现代处理器微架构中的并行性。考虑下列特征，它们都提供了隐藏的并行。大致上根据时间排序：

- 位并行（bit-parallel）的功能部件
- 乘法功能部件
- 流水线执行
- 乱序（out of order）执行
- 不断增加的数据通路宽度
- DMA控制器
- 预取部件
- 追踪cache（trace cache）
- 并发多线程
- 向量处理器
- 芯片多处理器
- 协处理器（co-processor）：I/O控制器，网络控制器，图形协处理器

列表中以斜体表示的项是将并行性暴露给了软件，其余大部分对程序员是隐藏的。我们说“大部分”是因为有时程序员或编译器希望推断cache、预取或硬件并行其他方面的效果。在某些例子中这部分的功能是被暴露给软件的，例如通过预取指令或是性能异常。

我们看到随着时间的推移，并行性是朝着更大单元的趋势发展。基于这个观点，我们不可能再把并行性隐藏在硬件中，这也就解释了为什么现今并行性会以多核芯片的形式暴露给软件。

当然有几种不同的方法可用来隐藏并行。例如我们看到ZPL如何在提供串行语义的同时，鼓励程序员根据区域思考，而这能隐式表现并行。另一个例子是Cilk语言，它使用一组能产生并行的显式并行结构，对C语言进行了扩展。Cilk使用称为串行省略（serial elision）的定义来保持串行语义。尤其是，Cilk程序的语义与那些去除了Cilk结构的串行C语言程序的语义是完全一致的。

隐藏并行是至关重要的，因为它能隐藏复杂性，而这在系统设计中是一个极为强大的工具。当然这之间的窍门在于，如何既能隐藏复杂性又能使性能开销显得微不足道。

9.3.2 透明化性能

让程序员在开发算法和程序的过程中，就能精确地推断性能是非常重要的。我们看到，任意层次的语言都能妨碍对性能的推断，如Pthreads和NESL。同样，任意层次的语言，不管低层或高层，也都能支持对性能的推断，如MPI，Co-Array Fortran和ZPL。

9.3.3 局部性[⊖]

我们认为全局视图抽象是必需的，但让程序设计的抽象鼓励局部性和最小化数据移动也是非常重要的。MPI通过通信的显式表示和困难性来鼓励局部性。PGAS语言和ZPL语言均以不同的程度鼓励局部性，同时又没有引入过多的程序设计困难。展望未来，我们希望所开发的语言既能鼓励局部性，又能提供便利，这一点尤为重要。

9.3.4 约束并行

语言设计者经常试图提供强大的语言结构以方便程序设计。但在并行语言中，语言结构的过于强大，以及程序员所拥有的过多灵活性，都将是有害的。

- 灵活性会影响正确性，因为它会允许一些很难处理的交互。例如，允许线程在任意时刻访问和修改任意存储器地址倒是很容易，但这种便利会导致很难限制不想要的交互。
- 灵活性会影响性能，因为它会混淆性能模型。例如，锁引入了资源竞争的可能性，但在不了解多种机器特定操作的代价时（例如存储器时延，指令时延和竞争线程数），推断锁的性能将极为困难。
- 灵活性可用于实现最大并行性，但最大并行性并不是目标。我们的目标是将可用的并行性高效率的使用，这意味着我们通常需要利用局部性，限制线程间的相关性，以及推断数据的移动和同步。

因此并行语言的重要性质可能不在于它允许做什么，而是它不允许做什么。例如，对比

⊖ locality既可做“局部性”解，又可做“本地性”解。此处略偏向后者，但以全书术语统一起见，故仍译为“局部性”。——译者注

POSIX Threads和一些更高级的方法，比如OpenMP和ZPL，我们看到Pthreads允许线程非常自由地交互，这使得程序员必须说明哪些操作不应当发生。反之，OpenMP和ZPL从一开始就限制了可用的并行性。从正确性的观点来看，这使得错误的危险性更小。从性能的观点来看，这使得程序的行为更加可预测。当然，关键在于找到自由与约束之间合适的平衡，而这个问题的答案可能会根据求解问题领域的不同而有所不同。

9.3.5 隐式并行与显式并行

我们希望使用约束的并行以方便程序设计，透明化性能，以及鼓励局部性。我们可能会问这样的一个问题：在哪个层次中暴露并行性是最合适的？我们应当把并行性暴露在库中，在函数中，还是在硬件接口（ISA）中？依据并行性属于底层的想法，合适的层次应该要尽可能的低，但又不会影响我们推断性能和获得良好性能的能力。

不同的求解问题领域有不同的需求。迄今为止，通用微处理器都能对软件隐藏并行性，但甚至是早期的图形处理器都发现暴露并行性很重要。现今的图形处理器（GPU）能以一种很良好的方式暴露并行性：程序员通常以串行代码片段的方式编写阴影例程，而并行性则由硬件供应商编写的代码实现。其他一些求解问题领域，比如数字信号处理，因为有足够的约束，所以才会有成功的领域相关（domain-specific）语言出现。它们能有效的编译，产生特别高效的并行代码。

如何平衡好存在于通用性与便利性之间的关系是极具挑战性的。通用的解决方案试图提供通用的机制，而领域相关的解决方案则通过高层抽象隐藏并行。这种关系的平衡被认为是显式并行与隐式并行之间的折中，即显式更通用，而隐式更便利。如果提供隐式并行的抽象能与低层、更加显式的并行形式相联系，程序员就能打破这个平衡，甄选出一个既能满足程序设计效果又能满足性能需求的层次。

因而其目标就是提供能连贯支持一系列不同等级抽象的程序设计系统：

- 在高等级的层次中，抽象提供了便利性，但未提供完整的通用性，即约束的并行。
- 在低等级的层次中，会有更多通用的特性。

这种系统的关键在于：（1）为专家提供了便利的方法来创建合适的抽象，尤其是那些与领域相关的；（2）为在不同等级抽象上说明的程序提供了进行便利交互的机制；（3）提供了能利用高层抽象假设的优化工具。

9.4 小结

在本章中，我们逐一评价了第6章至第8章中所提及的程序设计方法。我们的目的不是从中选出获胜者和失败者，而是将那些读者已经注意到的特征明确化。每个程序设计工具都有自己的优缺点，也都有一群忠实的程序员拥护者，他们会为自己所钟爱的方法热情地辩护。在本章的最后，我们总结了本书中的一些经验，并建议了未来语言必备的一些重要性质。

通过了解并行程序设计的现状，我们确信未来的新语言和新程序设计抽象将会简化并行程序员的工作。第10章将简要介绍其中一些有前途的新想法。

历史回顾

Deitz[2004]介绍了*P*独立，虽然早在未命名之前，这个概念就作为函数式并行语言的优点

为人所知了。Cilk是由Leiserson领导的团队在MIT开发的[1995]。

习题

1. 应用第5章中用户自定义归约的概念，编写一个*P*相关的并行归约。该并行归约应具有能统计出参与计算的处理器数量的特性。
2. 在参考文献中找出一个并行程序设计语言的描述，并判断它是否是*P*独立的。
3. 找出5种在本书中提及但并未在本章中分析的并行程序设计抽象，然后判断它们是全局视图抽象还是局部视图抽象；每个分类至少包含1个抽象。
4. 回忆第1章中统计3的例子，思考它为何不能达到8倍并行加速。考虑在何种情况下，程序能达到更高的加速比？

第四部分 展 望

在学习并行计算的通用原理和各种语言特征部分之后，我们现在把目光转向未来，将从社区和个人两方面来加以观察。并行计算这一快速发展的领域为我们提供了许多新的机会。

对并行程序员来讲，最感兴趣的是那些能简化我们开发程序的新研究。而诸如事务存储器的抽象以及新的并行程序设计语言就属于这种新的研究领域，这将是第10章的主要论题。我们也感兴趣于那些能为应用并行性提供显著机遇的硬件系统，将对某些流行的系统予细研究，如用于个人计算的附属处理器和以网格为代表的巨大系统。在第10章的最后，我们将探究新出现的计算范例MapReduce，以及三种新兴的程序设计语言，它们均旨在为了提高程序员的生产率。

从通用特征个人，我们为编写、调试和度量并行程序提出了一些指导方针。虽然第11章是最后一章，但我们建议学生在学习并行计算的早期先学习这一章。这些指导方针尽管是为了结课课程设计而构思的，但它们对于编写每章习题中的简单程序而言也具有指导意义。

第10章　并行程序设计的未来方向

伟大的尼尔斯·玻尔曾说过“预测是件难事，特别是对于未来”。这句话在并行计算中特别真实，因为研究人员和供应商经常吹捧一些“银弹（silver bullet）”[⊖]解决方案，但最终往往并不成功。在本章中将讨论我们选择的几个正在开发的概念。由于如玻尔所说预测未来是困难的，我们并不是宣称这些是最有前途的主题。但不管如何，它们已经相当“热门”，因此读者有必要有所熟悉。

10.1　附属处理器

随着通用微处理器变得越来越复杂和功耗越来越大，因此长期以来已经认识到，在通用微处理器上倾注如此多的精力可能是低效的。相反，专用的处理器可能功能更强大且节省空间，因此尽可能地卸载更多的工作到专用的*附属处理器*就显得很有意义。使用附属处理器并不是新事物，早期的微处理器就卸载一部分工作给浮点协同处理器。现代的“附属处理器”术语暗示两个处理器间的不对称性质，表明附属处理器是代表主处理器完成工作（参见图10-1）。

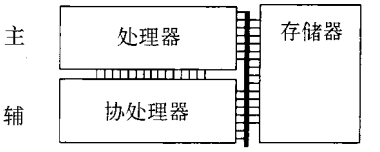


图10-1　附属处理器

像浮点协同处理器一样，当今的附属处理器可以视为是求解密集计算任务的专用引擎。主处理器与附属处理器并发运行并不是并行性的主要来源。相反，通常大量的并行性嵌入在附属处理器中。

虽然也提出了这种结构的其他实例，包括附属于通用处理器的现场可编程门阵列（FPGA），本小节将焦点聚集到以下两个体系结构：

- 附属于一个标准处理器的图形处理部件
- 有8个附属处理器和一个嵌入式Power PC主处理器的Cell处理器

10.1.1　图形处理部件

在图形处理部件（GPGPU）上完成通用计算的概念近来已经很流行，这是因为GPU具有很高的性能和性能价格比。例如，比较nVidia GeForce 8800和Intel Core2 Duo，我们可以看到GPU有大约10倍的浮点CPU能力（367 GFLOPS对32 GFLOPS）以及大约10倍的存储器带宽（86.48 GB/s对8.4 GB/s）。此外，GPU性能的改进速率比通用微处理器性能的改进要快，大约是每6个月性能就增加一倍，显著地快于通用微处理器的改进，如图10-2所示。由于GPU的可编程性正在不断增长，因此了解它们的优缺点以及将它们作为通用计算的前景是明智之举。

GPU之所以能提供如此好的性能有以下几个理由。从经济上讲，GPU的性能需求是由大型视频游戏工业驱动的。从技术上讲，GPU是专门处理图形绘制的，是一种具有大量数据并

⊖ 这里的银弹是指灵丹妙药。——译者注

行的密集计算应用。为此GPU能很容易地通过增加更多的处理核来改进浮点性能。此外，GPU可以极大地忽略许多通用处理器必须处理的问题，包括为顺序代码保持流水线的畅通等。这样GPU就节省了从事转移预测、高速缓存和指令调度工作所需的大量晶体管。

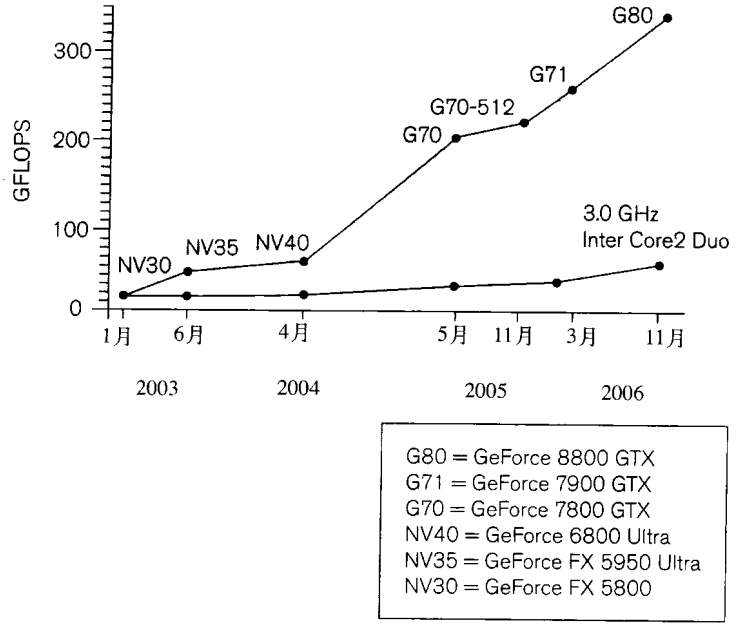


图10-2 随时间变化的GPU和CPU性能

GPU对图形流水实现硬件支持，它将定义几何图形的顶点表作为输入，而将帧缓冲区中的图像发射作为输出。顾名思义，在该流水线中还有几个中间级，均是完成与图形有关的操作（参见图10-3）。通常，有一级将顶点从对象空间映射到屏幕空间；另一级光栅化三角形，产生构成像素值的片段；再有一级是完成明暗处理和纹理处理。当然，这里的讨论是对实际过程的简化，而这种过程随时间在不断变化。

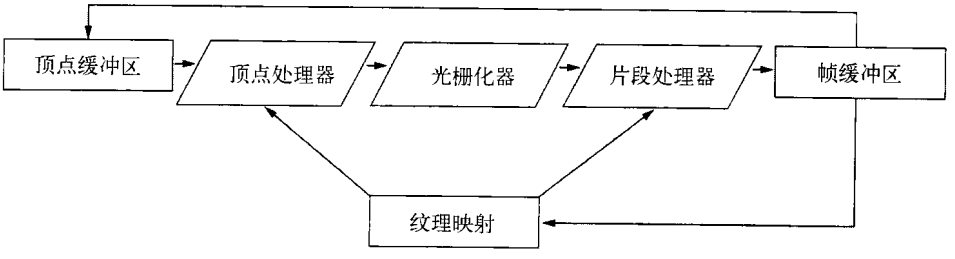


图10-3 图形流水线

早期的GPU只关注对图形学的支持，这种系统有一个固定功能的图形流水线硬件实现。这些早期的GPU只支持单精度浮点操作，没有双精度浮点和整数操作，也没有位处理操作。但是随着时间的变迁，由于游戏开发者寻找以不同方式使用硬件，硬件的可编程性越来越高。其结果是GPU趋向于支持更通用的计算。例如，下一代的nVidia GeForce卡将支持双精度浮点和整数操作。

如果我们去掉某些术语，一个现代的GPU犹如一个具有片内DRAM的大规模细粒度多核

芯片（参见图10-4），例如，nVidia GeForce 8800有8个核（图10-4b的副本），每个核由16个SIMD处理器组成，这表明这些处理器以锁步方式运行。

虽然硬件的实现通常是SIMD，但程序设计模型并非如此，就Compute Unified Device Architecture（CUDA，计算统一部件体系结构）语言而言，它几乎是C程序设计语言的完整子集。不管如何，GPU的能力在于它能完成大规模并行浮点计算，如同其他的多核芯片一样，一个标准的范例是，使用多处理器对输入的数据块运行尽可能多的处理，然后再装载新的数据块。

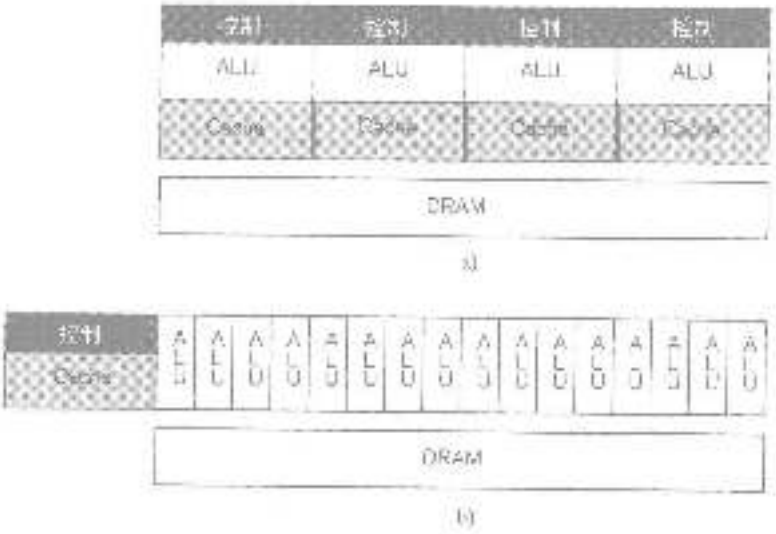


图10-4 简图：a)传统的4核多核系统，b)含有16个通用ALU部件的（半）GPU核，该核在芯片上将被复制

作为可为GPU编写各种不同应用程序的一个提示，CUDA开发者工具包（见页http://developer.nvidia.com/object/cuda.html）给出了以下用CUDA编写的程序例子：

- 并行双调排序
- 矩阵乘法
- 矩阵转置
- 大型数组的并行前缀求和（扫描）
- 图像卷积
- 使用Haar小波的一维离散小波变换
- CUDA BLAS和FFT库使用举例
- 二项式期权定价（Option Pricing）
- Black-Scholes期权定价
- 蒙特卡罗期权定价
- 并行Mersenne Twister（随机数发生器）
- 并行直方图
- 图像降噪
- Sobel边缘检测滤波器

总之，随着用户社区不断增长（参见www.gpgpu.org），GPU的通用计算使用非常有前途。

10.1.2 Cell处理器

Cell（胞元）处理器是索尼、东芝和IBM三家的联合研究项目，目的在于生产功能强大、支持交互视频游戏的部件。如图10-5所示的Cell体系结构，包括一个标准的含有L1和L2 cache的PowerPC以及8个SPE或"协作处理单元"（synergistic processing element）。这些SPE是功能强大的SIMD机器，每个周期能执行4个单精度浮点操作。EIB或"单元互连总线"（element interconnect bus）支持大量的片内通信，名义上的带宽超过300 GB/s，但在监听时带宽受限，降为超过200 GB/s。最后，Cell系统的主存有很大的带宽。

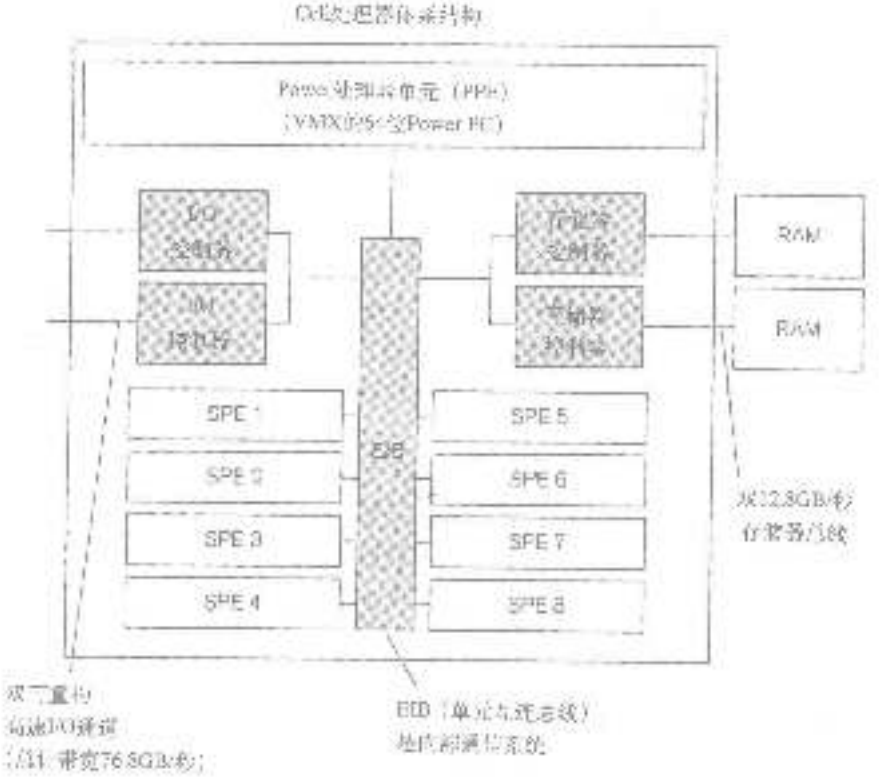


图10-5 Cell处理器的主要组

Cell在Linux操作系统环境下用C/C++语言进行编程。目前支持Cell程序员的工具比较有限。对于如矩阵乘法那样的标准计算，使用Cell的早期实验表明，它能从该芯片中得到绝大部分的理论性能，但是能达到这种性能的程序设计的成果是混杂的。SPE没有cache，因此程序员必须使用DMA命令小心地管理指令和数据进出芯片的数据。所以，双缓冲的概念对于达到高的性能非常重要。双缓冲概念是指当一个缓冲器中的数据块正在传送时，另一个缓冲器中的数据块正在计算，这是重叠通信时延和计算的一种形式。编译器中自动完成这种数据移动的尝试到三前为止被证明是不成功的。对社区的一个挑战是开发工具和和语言以方便对这种特殊处理器的编程。

10.1.3 附属处理器的总结

附属处理器非吸引人，因为它们的专门特性能提供很高的性能，减轻了主处理器的计算

负载以及和存储器的通信量。但是，附属处理器复杂了程序设计模型（不是一个CTA模型），因为它要求程序员分割计算，小心地推断附属处理器和主处理器间的存储器通信量，而且经常要在低级的硬件细节抽象层上进行编程。

虽然附属处理器的程序设计模式似乎与本书中的其他方法相当不同，但有迹象表明这两个世界正在趋同。例如，微软的研究人员已经提出Accelerator（加速器）语言，它能将一个数组语言转换成GPU的片段代码。虽然在语言的早期阶段必须对数组语言的表示加以某些约束，但就其概念而言是有前途的。另一个例子是，编程GPU的语言继续接近编程通用处理器的语言。

10.2 网格计算

由于高性能计算与金融服务业、制造业、游戏业等各行各业的关系日趋紧密，激发了越来越多的机构组织使用大规模并行计算机的兴趣。但是，由于计算机的维护和管理的成本越来越高，因此让一个机构自己拥有和维护超级计算机通常是非经济有效的，特别是如果机构的计算需求时常会发生变化的话。在这些情况下，外购计算就是值得考虑的。

为与这种趋向相一致，在20世纪90年代出现了计算网格的概念。术语“计算网格”（computing grid）使人想起类似的术语电力网格，电是一种任何人可以使用的资源，不必在意电是如何获得和传输的细节。计算网格将类似地提供计算能力但向用户隐藏了许多细节。由于经济规模导致的潜在经济有效性，现在至少已有十几个国家开展了国家级的网格计算研究项目。

当然计算网格的规模有大有小，也许是在一个公司内运行，也可能跨多个公司，甚至跨一个地区或一个国家。总的一个目标是创建一个计算资源它将大于它的各个部分的总和。例如，可以想象一个含有各种各样特殊硬件（从巨型存储库到如望远镜那样的强功能科学仪器的一切事物）的计算网格，如果让任何的一个机构单独购买的话将是非常昂贵的。即使不含有专用的硬件，一个网格所提供的峰值计算能力将远远超过任何单个机构所能提供的。

为了形成经济规模，网格经常由跨越大地理范围和跨多个行政管辖范畴在物理上分布的资源所组成（例如资源可能为不同的机构所拥有）。到目前为止，大部分的活动主要是致力于构造基础设施和定义标准以准许网格的创建和使用。当然这些问题要远比电力网格复杂，因为与计算服务的接口远比与电的接口更加宽广。

为了改进整个系统的利用率，大多数计算网格是多道程序化的，并具有进行动态资源分配的能力。这种情景与通常的超级计算环境不同，后者以批处理模式运行客户程序，并用成组调度使客户计算间的冲突最小化。

与分布系统一样，计算网格面临许多相同的由分布计算社区所提出的问题：

- 资源管理
- 可用性
- 透明性
- 异构性
- 可扩展性
- 容错
- 安全性
- 隐私性

然而，由于网格计算的用户感兴趣于并行计算，因此它还有性能和应用可扩展性的附加问题。并行和分布计算的汇合引入了某些有关网格效率及其应用的一些有兴趣的技术问题，这些问题包括：

- 不同客户程序间的冲突将使每个程序的效率降低多少？特别是使一个进程性能退化的冲突可能产生一个影响其他进程的涟漪效应，因为其他进程必须与该退化的进程进行交互。在最坏的情况下，将会出现这样一种情景，即某些处于闲置的进程正在等待某些重负载进程所产生的事件。可以使用什么样的技术使这种性能的衰退降至最小？
- 程序员如果不知道目标平台的细节能编写出高效的程序吗？就目前的实践经验而论，许多程序是专门为特定的硬件平台而编写的（我们不推荐这样做），在程序进入生产模式前，至少通常应有某些目标专用的优化。
- 如果不能看到一个一致的可重复计算资源集，程序员如何调试、开发和优化他们的程序？当然，使用*P*-独立语言将有助于问题的解决，但这些语言还未标准化。
- 用户如何判断低效率是由网格引起的还是由于用户程序本身的低效引起的？
- 如何能简化自省（introspective）程序的生成，这种自省程序能自行优化和重构以适应变化的计算环境？

10.3 事务存储器

几十年来，数据库通过使用事务的概念处理并发问题，因此询问是否可将类似的概念应用到共享存储器并行程序的存储器操作中就是有意义的。*事务存储器*（TM，Transactional Memory）领域中的活跃研究近来也已提出类似这样的问题。在本节中我们将介绍事务存储器主要基本概念，还将叙述TM为何能克服基于锁机制程序的许多缺点，此外，我们也将指出某些尚未解决的重要问题。

一个数据库事务在修改数据时将保证称为ACID（原子性、一致性、隔离性、持久性）的四个特性。*原子性*（atomicity）表示事务的所有操作完成或未完成。*一致性*（consistency）是一个与应用相关的概念，是指存储器的更新就好像操作以某种串行次序方式完成一样。*隔离性*（isolation）是指事务操作的结果等同于一个隔离执行的结果，就如同不存在其他的事务。*持久性*（durability）是指由事务操作引起的改变将是持续的。但程序不需要持久性，因为当程序退出时它的存储器状态就消失了，所以在TM系统中的事务只有原子性、一致性和隔离性。

在一个TM系统中，需要由程序员来识别事务，而系统（不论是由硬件还是软件实现）则通过追踪对存储器的装载和存储的操作以及检测可能违反事务特性的冲突，来保证事务语义的实现。如果事务成功完成，就说它已*提交*（commit）；否则它就*异常中止*（abort），且不会观察到任何副作用。

事务存储器简化了程序设计，因为程序员所感觉的事务操作是顺序化的且没有与其他线程交互的可能性。与基于锁的程序相比，TM有许多优点。特别是事务具有可扩展、可复合、无死锁以及易于使用的特点。

具体的讲，事务将为某种语言构造所识别，如约束事务范围的*原子区域*。例如，在统计3的例子中，对共享变量count的更新可按如下方式实现：

```
atomic
{
  count+=private_count;
}
```

atomic区域与Java中的synchronized（同步）语句（参见第6章）有惊人的相似之处：

```
synchronized
{
  count+=private_count;
}
```

这里的关键词synchronized识别出该语句是一个临界区，所以将以互斥方式加以执行。由于这个例子过于简单，因此很难说清两者之间的区别，但下一小节将通过与基于锁的临界区的比较来解释事务的优点。

10.3.1 与锁的比较

从第9章中我们可以看到，基于锁的程序设计受到好几个问题的困扰。现在再来讨论这些问题并解释事务存储器是如何解决这些问题的。

第一，锁可能导致死锁。事务则不会死锁；它们要么提交要么异常中止。当然，有可能出现活锁，它是指由于反复地异常中止使得事务无法前进的一种现象。

第二，锁的约束过于严格。它强制操作顺序的执行，即使在以下两种不需要的情况下也为如此：（1）对一个共享存储单元进行并发读，以及（2）对不同的存储单元进行并发写（或并发写和读）。与此相反，事务存储器系统能检测出上述两种情况并允许它们并发地执行。

第三，锁面临粒度的协调问题：粗粒度的锁将限制并发性，从而也限制了可扩展性，而细粒度的锁则难于推断，因为存在死锁的可能。对于事务存储器，如果程序员定义了大的原子段，系统将允许执行多线程只要它们对存储器的访问不发生冲突，因此事务比锁有更好的可扩展潜力。当然，大型的事务并不完全自由，因为它们迫使TM系统去追踪大量的装载和存储操作。

现在可以看到一个原子段和Java的同步化语句之间的差别。如果要对一个已有的方法（它可能表示某种不定的大量代码）提供互斥，同步化语句将使整个方法的执行串行化，而原子区域则要求系统不允许在该区域中的存储器访问发生冲突。因此，事务比锁能更好地调节并发性的颗粒度。

第四，锁不能很好地复合。例如，如果访问数据的两个方法以不同的顺序获取细粒度的锁，将它们组合在一起则可能引起死锁。对于事务存储器，所有的相关是动态检测和处理的，所以不存在静态顺序获取锁的概念。

从根本上讲，我们看到锁是静态地说明一个实现策略，它不是最理想的，因为线程的实际交叉在运行时间之前是无法知道的。虽然近期的工作已经提议用硬件在运行时间来检测可消除锁的这种情况（称为推测锁删除，Speculative Lock Elision），但推论使用锁的依据则是它是静态、复杂和非复合的。与此相比，事务存储器要求程序员静态地提供所希望的语义但允许系统动态地进行实现的决定。当然，如果有合适的语言支持，由编译器进行静态分析能为动态系统提供有潜在价值的信息，所以此时就不应忽略静态信息。

10.3.2 实现方法

我们认为事务存储器的使用比锁更简单且能提供潜在的性能得益。问题的关键在于TM系统如何能有效地实现事务。要完完全全回答这一问题似乎太早，但在本小节最后讨论某些未解决的问题之前，我们将简单地总结设计问题的空间。

一个TM系统必须完成两个基本操作：一是当不同的线程访问存储器时，它必须能检测冲

突；二是它必须提供一个机制以隔离一个事务的结果直至它提交或异常中止。这就是通常所说的*冲突检测*（conflict detection）和*数据版本*（data versioning）。

冲突检测检查是否有两个线程访问同一个存储单元，且其中至少有一个操作正欲修改该单元。要检测这种冲突，TM系统必须对每一个事务的读集和写集进行跟踪。冲突检测可以以悲观或乐观的方式进行。

*悲观的冲突检测*方式采用尽可能早地检测冲突策略，以避免浪费所做的工作。在检测到一个冲突时，它或是立刻异常中止一个事务，或是停顿其中一个事务，以防止其完成违规的访问，并希望以后不再发生冲突。另一方面，悲观的方法可能面临周期地复发冲突从而导致不能继续前进。

*乐观的冲突检测*方式假设冲突是很少发生的，所以它延迟冲突的检测直到一个事务快要结束时，此时该事务或是提交或是异常中止。这种方法当冲突发生时将浪费所做的工作。它也排斥了选择停顿一个线程作为解决冲突的可能，因为各个线程很可能已经通过了第一个潜在的冲突向前执行。另一方面，乐观的方法能通过简单地提交第一个完成的事务保证前行。

第二个基本机制维持多个数据版本，从而可使事务或是回滚或是提交。同样，它也有两个基本方法：急切（eager）和滞后（lazy）。

当使用*急切版本*方法时，系统保留老版本的数据，所以它可将新的值写入存储器。如果提交的可能性较高，则该方法的速度就较快，但是异常中止就会较慢，因为必须通过遍历保留值的表才能恢复事务的状态。

当使用*滞后版本*方法时，新的值将被保留在缓冲器中，老的值则没有变化。因此异常中止的工作速度就较快，而提交就会较慢，因为必须从写缓冲器中拷贝值。该方法还有一个附加的开销，因为当要进行读时就必须搜索相关的写缓冲器以获取新的值。

设计空间的另一个方面是协调冲突检测的颗粒度，它可以在对象级、cache行级或字级上完成。这些选择将依据：（1）冲突检测数据版本的开销和（2）假冲突可能性（类似于假共享）这两点进行不同的协调。

10.3.3 未解决的问题

除了刚才我们所提及的许多实现问题以外，现在来探讨有关事务存储器的一些未解决的问题。

- I/O的操作是有问题的，因为I/O部件不在TM系统的回退机制管辖之内，所以它们的操作不能够取消。类似地，与遗留代码的交互也有问题，因为这些代码还未分解成事务。
- 冲突检测被定义在低级操作上，即装载和存储，所以TM系统不能够区别*结构冲突*和*语义冲突*。为了了解这一问题，考虑递增计数器的两个事务。从语义上讲，只要每个操作确实更新该计数器（与更新寄存器中的值相对比），这两个事务能合法地交叉它们的递增。然而，TM系统将不允许许多这种语义上合法的交叉，因为装载和存储的排序将指明这是一个冲突。而这种差别将强使TM系统异常中止许多并不需要异常中止的事务。
- 已提议采用*开放式嵌套事务*来处理上述的两个问题。开放式嵌套事务实质上是提供一个脱离TM系统的机制，但将引入与开放式嵌套事务与TM系统之间的交互相关的复杂和基本的问题。
- 长时间运行的事务是有问题的。在某些实现策略下它们可能永远不提交。另一方面，如果授予它们有比短期运行的事务更高的优先级，它们可能使许多其他的线程严重地衰减

性能。在数据库领域中，长时间运行的事务通常用特殊语义对它们进行处理，或是使它们运行在另外的系统中。现在仍不清楚如何将这些解决方法转换到事务存储器中。

- 在软件事务存储器（STM）与硬件事务存储器（HTM）之间存在着微妙但却是基本的差别，即STM不能简单地迁移到HTM。从实践上讲，这种两分性（dichotomy）是显著的，因为理想地我们可以使用STM去构建软件基础，用事务存储器获得经验并指导HTM的开发。但是由于它们在语义上的差别，这种迁移是不可能的。
- 当然事务存储器并不试图能解决所有与并行性能有关的问题。它没有涉及局部性的问题或减少线程间交互的问题，因此另一个开放的领域是事务存储器与并行程序设计模型或并行语言的集成，以利于获得好的局部性和数据的移动。

总之，事务存储器是一个具有潜在利益和某些未解决问题的很有意思的研究领域。

动态乐观的并行化　恰如事务存储器建立在数据库的串行化执行（等价于某种顺序执行）概念上一样，Galois系统将此概念应用到更高的语义层次。特别是Galois系统适合于并行化数据相关性不能静态了解的代码，在许多基于图的算法中就是这种情况，例如Delauney三角化。该语言支持乐观的并行化，此时假设应用中计算的不同部分能独立地进行。独立性的概念是基于可交换性，就如抽象数据结构上的操作所定义的，当发现冲突时执行就回滚。Galois系统的一个重要的方面是它具有更高层次特殊数据结构的可交换性概念。例如，在一个共享集上的操作可交换性的定义是基于抽象集的操作而不是低级的装载和存储。

10.4 MapReduce

并行计算的一个重要发展是基于机群计算的大量增加，典型代表是Google开发的MapReduce设施。根据Dean和Ghemawat在2004年的描述，MapReduce是一个使用标准化插入式的框架对庞大数据档案进行搜索的工具。Google使用MapReduce替代它的构建索引的自主软件，来完成诸如计算PageRank的操作。这一概念已被广泛使用，并已经有一些实现。MapReduce的名称源自LISP和其他函数式语言中的映射和归约操作。Map运算符将对表中的每个元素作用某个函数，而reduce运算符则组合一个表中的元素，这个操作符已在本书中论述多次。术语MapReduce现在已通用化，它也适用于派生的系统。

除了规模以外，MapReduce框架与第4章中的Schwartz算法以及第5章中的定制归约有许多相似之处。但这里关键是规模，因而它采用更基于流的观念。为此，假想一颗树，其中叶子是一个如Web页面分布式文件系统中的磁盘。所感兴趣的计算有时会生成单个输出值，如我们所知的归约计算。但是更为通用的是它们可能生成值表（在数据库中相当典型），如一个字表以及每个字在文件系统中出现的次数。

来自磁盘的数据流，用*map*函数进行过滤，产生一个键/值（key/value）对的输出流。例如，如果该任务是统计字的出现次数，则当在输入中遇到“dog”时，文件中的数据将被转换成如<'dog',1>的对流。键/值对将流向另一个称为聚集器（*aggregator*）的计算机（参见图10-6），它将对这些键/值对分类。然后聚集器对流应用归约函数生成汇总流。对于统计字的例子，含有字/计数（word/count）对的那些表进行归并（同一字的两个计数进行相加）产生另一个字/计数对表，这个表将可能参加进一步的聚集。归并是一个典型的归约操作，但还有其他更为复杂的归约操作。归约操作的输出可以与其他的输出进行聚集，形成一个层次式结构因而能处理大量的数据。

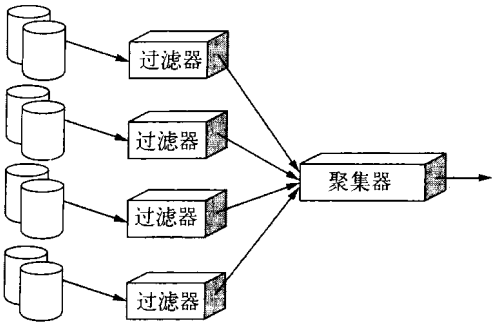


图10-6 MapReduce系统的示意图

为了使用该系统，程序员需要编写简单的脚本以定义映射和归约操作；然后将这些操作插入MapReduce框架。下面考虑一个例子，它要求处理许多含有记录的文件集合，其中每个记录为一个浮点数，而该浮点数可能是科学实验的结果或是所观察到的数据。所期望的输出是记录数（count）、浮点数的和（total）以及浮点数的平方和（sum_of_squares）。这一计算用Sawzall系统（MapReduce的派生系统）编写的程序如下：

```
count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;
x: float=input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x*x;
```

前三行为聚集器声明所需的归约函数，该聚集器将对来自过滤器的数据进行求和；保留字table为聚集器类型；在这里它仅表示单个值。

其余的语句为过滤器说明映射操作。其中第一句为处理文件的记录（绑定到input的无类型的字节字符串），即将记录转换成浮点数并将它们赋值给x。过滤器然后向聚集器发射三个键/值对供后者处理。过滤器的代码对所有文件中的每一个记录进行实例化。

在许多实例中，有机会在过滤器和第一轮聚集之间进行中间级的优化，这一优化可能使需传送的数据总量大为减少。例如，前面的字统计例子将为分布式文件系统中一个字的每次出现发射一个键/值对，但是很可能只有极少的单字，所以可以想象用一个归并值的组合操作来实现数据归约。其结果是，每个过滤器只需简单地向聚集器发送一个单字和它们的字数统计的表。

MapReduce既是并行计算又是分布计算。它利用许多计算机高效地计算一个特定的结果，但是由于其规模较大，因此分布计算中的可靠性、可用性等问题是很关键的。也许强调源自并行性的计算能力和分布计算的挑战性两者之间平衡的最好方法是引证Pike[2005]的一段话：

我们在2005年3月检测了[Sawzall]的使用。在这段时间中，在有1500个Xeon CPU的专用Workqueue机群上共发动了32 580个Sawzall作业，平均每个作业使用220个机器。在运行这些作业时共出现18 636次故障（应用故障、网络中断、系统崩溃等）导致作业的某些部分重新运行。作业共读入数据3.2×10^{15}字节（2.8PB）和写入9.9×10^{12}字节（9.3TB）（表明术语"数据归约"有某种影响）。所以每个作业平均处理大约100GB。所有作业共使用了几乎相当于一个机器在一个世纪内所使用的数据。

除了合并了并行和分布计算的概念外，MapReduce展示了一种处理大量数据的一流方法，在这种数据处理中磁盘的访问时间占据了计算的主要成分。

10.5 问题空间的提升

众所周知分而治之那样的算法技术能指导我们寻找对问题的高效解，随着并行程序设计的进展，我们期待有新的适合于并行计算的程序设计技术的发明。问题空间提升（Problem Space Promotion）技术是一个近期的例子。

问题空间提升（PSP）是对含有数组数据组合（combinatorial）交互求解问题的一种并行求解技术。PSP将原来在d维数据上操作的算法重新表示为在具有更高维问题空间上的计算。PSP的目的是在一维或多维上复制数据以达到增加并行性和减少通信和同步的需求，即消除相关性。为了了解如何可以做到这一点，设想一个求解问题，它要求n个数组值必须进行全交互的配对操作。一种解答是保持两份数组的拷贝，然后对这两个数组中的相应元素进行操作，此后循环地移位其中的一个数组；在n次移位后，就可处理完所有n^2对数据的交互操作。（移位的表示适合于分布式存储器配置，但在共享存储器配置中，只需在一个双重嵌套循环中分别对数组进行访问就可达到同样的效果）。另一个具有更大并行性的方法是创建两个新的操作数，其中一个通过复制数组行形成，另一个则通过复制数组转置列形成；对这些数组元素的相应对进行组合并对所有这些对同时进行操作。

为了进行具体的说明，考虑如何对有n个不同数的数字序列进行排序：

C←$S^T \leqslant S$	对所有的数对进行比较，若为真置1，否则置0
P←sum_cols(C)	列的和给定已排序索引值位置
S←S[P]	将S中元素按顺序排列

这一求解需要完成平方倍的工作量，因为它要完成n^2次比较。虽然输入是一维的，但操作是在二维数组上完成的，如上面的C。PSP方法的优点是它指明所有的比较可以独立地完成且所有的列可以独立地求和，与第4章中的奇－偶交替排序相比，该方法显著地增加了并发性。细心的读者将会发现该计算所使用的技术与在第8章所使用的排序技术相类似，在第8章的ZPL程序中使用该技术对咖啡饮用者进行分类；ZPL的扩充和归约操作符很适合于PSP技术的使用。

除了排序，许多$O(n^2)$操作可以表示成如PSP那样具有更少相关性的计算。N体计算是其中的一个例子，就如同是对一个输入集方式的识别。此外，图8-2中所示的SUMMA算法中矩阵乘法的“点积步”也是PSP的一个实例；实际SUMMA本身就是n个PSP操作的迭代。更一般地，可将矩阵乘积看成是一个3维的PSP计算。2维的输入矩阵被复制（第8章的术语为扩充）成一个3维“盒”，将所有的n^3对值放置在一起。然后这些盒按元素相乘，并沿第3维求和得到结果的矩阵积。

我们所强调的“提升维数”只是一个逻辑概念，一个程序设计的抽象将给程序员提供一个不会引入相关性的说明计算的方法。与其他在较低层次和用更初级的术语所定义的计算的技术相比，借助消除相关性的PSP说明使得性能有很大的改进。

10.6 新出现的语言

超级计算机的早期研究工作主要集中在硬件方面，而软件通常是大量程序设计努力的结果。在认识到程序员的生产率至少与机器的峰值性能一样重要后，DARPA所资助的高生产率计算（HPC, High Productivity Computing）计划试图开发并行程序设计语言以改善程序员的生产率。在几轮竞争后，HPC计划推出了三个新的并行程序设计语言（也提出了新的硬件）：

- Cray公司开发的Chapel (Cascade High Productivity Language)
- Sun Microsystems公司开发的Fortress
- IBM华生研究中心开发的X10

这些语言还未完成设计和实现，但它们提出了有望提高并行程序设计生产率的一些概念。它们是开源（open resource）的成果，可以使用它们的设计规范。下面我们对以上的三种语言分别作一个简短的介绍。

10.6.1 Chapel

设计多级高生产率语言（Cascade High Productivity Language）的Cray公司团队，对在高性能计算中占统治地位的低级的“片段”程序设计方法提出了批评（主要是消息传递）。他们要达到的设计目标是目标码的性能可与低级的技术相比较，而同时通过使用业经证明的高层抽象来提高生产率，这种高层的抽象得到了先进的编译技术的支持（参见图10-7）。Cray公司团队对Chapel语言的描述如下：

Chapel在高层次上支持多线程并行程序设计模型，这是通过对数据并行、任务并行以及嵌套并行实行抽象做到的。Chapel借助对数据分布和子计算的数据驱动安置，抽象支持程序中的数据局部性和计算的优化。Chapel还借助面向对象的概念和通用的程序设计特性支持代码的重用和通用性。虽然Chapel语言从许多以前的语言中借鉴了不少概念，但它的并行概念则非常接近于高性能Fortran（HPF）、ZPL以及Cray对Fortran/C的MTA扩展中所使用的概念。

Chapel设计的指导原则基于语言技术中以下四个关键领域：多线程、局部性意识(locality-awareness)、面向对象以及通用程序设计[⊖]。

```
for (str, span) in genDFTPhases(numElements, radix) {
  forall (bankStart, twidIndex) in (ADom by 2*span, 0..) {
    var  wk2=W(twidIndex),
         wk1=W(2*twidIndex),
         wk3=(wk1.re-2*wk2.im*wk1.im,
              2*wk2.im*wk1.re-wk1.im):elemType;
    forall lo in bankStart+[0..str) do
      butterfly(wk1, wk2, wk3, A[[0..radix)*str+lo]);
         wk1=W(2*twidIndex+1);
         wk3=(wk1.re-2*wk2.re*wk1.im,
              2*wk2.re*wk1.re-wk1.im):elemType;
         wk2*=1.0i;
    forall lo in bankStart+span+[0..str) do
      butterfly(wk1, wk2, wk3, A[[0..radix)*str+lo]);
  }
}
. . .
def butterfly(wk1, wk2, wk3, inout A:[1..radix]) { . . . }
```

图10-7 用Chapel语言编写的一维FFT（基－4）程序代码；完整的代码见http://chapel.cs . washington . edu/ hpccOverview.pdf

10.6.2 Fortress

Sun Microsystems公司的团队侧重采用创新方法来解决生产率的问题。例如，采用某些非

⊖ http://chapel.cs.washington.edu/spec-0.750.pdf。

常类似于标准数学符号的表达式（参见图10-8）。因此所形成的语言是与我们通常所想到的并行语言大不相同的。他们对Fortress语言作了如下的描述[⊖]：

在Fortress中我们支持如事务、局部性说明和隐式并行计算等特征，并将它们作为一个整体的特征构建在语言的核心中。Fortress组件系统和测试框架的特征方便了程序的汇编和测试，并使得强功能的编译器能跨越库的边界进行优化。即使是Fortress的句法和类型系统也是为用户量身定制成现代的HPC程序设计，支持数学符号和静态特征检查，如物理单位和尺寸、多维数组和矩阵的静态类型检查以及特殊范畴语言句法在库中的定义。此外，Fortress在设计时已考虑到这是一个“可增长”的语言，它能得体地支持对未来语言特征的增加。事实上，Fortress语言本身的大部分（甚至包括数组和其他基本类型的定义）是以库的形式在相当小的语言核上进行编码的。

通过寻找全新的表示计算的方法，Fortress语言建议我们去思考如何能以最好的方式来描述问题中的并行性。

```
conjGrad[Elt extends Number, nat N,
         Mat extends Matrix [Elt, N × N],
         Vec extends Vector [Elt, N]
        ](A: Mat, x: Vec):(Vec, Elt)
cgit_max = 25
z : Vec = 0
r : Vec = x
p : Vec = r
r : Elt = r^T r
for j ← seq(1: cgit_max) do
  q = Ap
  α = ρ / (p^T q)
  z := z + α p
  r := r − α q
  ρ_0 = ρ
  ρ := r^T r
  β = ρ / ρ_0
  p := r + β p
end
(z, ||x − A z||)
```

图10-8　用Fortress代码编写的共轭梯度计算（源自NAS并行基准测试程序套件）

10.6.3 X10

IBM团队正在设计的X10是IBM公司的PERCS项目（Productive Easy-to-use Reliable Computer Systems）中的一部分。虽然X10是一个全新的语言（参见图10-9），但它非常接近于Java[⊖]，此外由于X10也是一种PGAS语言，因此它可以与Titanium方法进行比较。下面就是X10团队对其方法的描述：

⊖ http://research.sun.com/projects/plrg/faq/index.html。

⊖ http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html/$FILE/ATTH4YZ4.pdf。

```
public class Jacobi {
  const int N=6;
  const double epsilon = 0.002;
  const double epsilon2 = 0.000000001;
  const region R = [0:N+1, 0:N+1];
  const region RInner= [1:N, 1:N];
  const distribution D = distribution.factory.block(R);
  const distribution DInner = D | RInner;
  const distribution DBoundary = D - RInner;
  const int EXPECTED ITERS=97;
  const double EXPECTED ERR=0.0018673382039402497;
  double[D] B = new double[D] (point p[i,j])
                {return DBoundary.contains(p)
                ? (N-1)/2 : N*(i-1)+(j-1); };
  public boolean run()  {
    int iters = 0;
    double err;
    while(true) {
      double[.] Temp =
          new double[DInner] (point [i,j])
          {return (read(i+1,j)+read(i-1,j)
          +read(i,j+1)+read(i,j-1))/4.0; };
      if((err=((B | DInner) - Temp).abs().sum())
          < epsilon)
        break;
      B.update(Temp);
      iters++;
    }
    System.out.println("Error="+err);
    System.out.println("Iterations="+iters);
    return Math.abs(err-EXPECTED ERR)<epsilon2
        && iters==EXPECTED ITERS;
  }
  public double read(final int i, final int j) {
    return future(D[i,j]) B[i,j].force();
  }
  public static void main(String args[]) {
    boolean b= (new Jacobi()).run();
    System.out.println("++++++ "
        + (b? "Test succeeded."
        :"Test failed."));
    System.exit(b?0:1);
  }
}
```

图10-9 用X10代码编写的2维雅可比迭代计算

X10的目标是旨在改进生产率，为了达到这一目标，它开发了一种新的程序设计模型，并在Eclipse中集成了一组新的工具以及一个新的实现技术，使得在一个可管理的运行环境中提供优化可扩展的并行性。X10是一个类型安全、现代、并行和分布式面向对象的语言，旨在使Java™程序员能很容易地使用它。它所针对的目标是未来的低端和高端系统，这些系统中的结点是由多核SMP芯片构成的，系统具有非一致存储器层次结构，并互连成可扩展机群形式。作为分区的全局地址空间（PGAS）语言系列的一个成员，X10突出了以下几个方面：以所在位置的形式清晰地指明局部性；在async，future，foreach以及ateach的构造中实现轻量级的活动；设置终止检测（finish）和阶段计算（clocks）的构造；使用无锁的原子块（atomic blocks）同步；全局数组和数据结构的处理。

X10语言中融入了很广的并行程序设计概念，这些概念是针对多核机器和多核机群的，而这些机器很可能是未来并行体系结构的典型代表。

10.7 小结

本章讨论了几种开发并行性的不同方法。这些方法在规模上有所不同，从使用Cell的SPE到MapReduce模型支持的因特网规模的计算。它们所要解决的问题也有很大的不同，包括准确性和方便性（事务存储器和Chapel、Fortress以及X10语言），芯片级（GPGUP）的性价比，企业级的性价比（网格）以及大规模计算的可编程性。最后，我们看到无论是在网格计算中还是在MapReduce中都出现了并行和分布计算的汇合。

历史回顾

IBM对Cell研究项目作了描述[2006]。nVidia公司的CG语言[Mark等]开创了在GPU上用通用语言进行计算的迁移。更近期推出的Accelerator(加速器)语言[Tarditi 2006]则基于数组计算。Rajwar和Goodman[2001]对锁的消除进行了论述。Foster和 Kesselman[2003]提出了网格的概念。Herlihy和Moss[1986]普及了硬件事务存储器的概念；Shavit和Touitou[1995]则提出了一个纯软件的方法。Dean和Ghemawat[2004]对MapReduce作了描述，而Chamberlain等[1999]描述了问题空间的提升。

习题

1. 假设MapReduce不是在Xeon机群上使用220个处理器处理100GB数据进行求解，而是用网格计算技术进行处理。解释此时的计算应做何种改变；在给出您的答案时，需考虑应该使用更多的处理器还是使用较少的处理器。
2. 在Cell处理器上使用Peril-L为矩阵乘法编写一个伪代码算法。挑选一个恰好使用8个处理器的算法，重点关注处理器间的数据移动。
3. 考虑如何在Cell处理器上实现排序。假设采用Batcher的双调谐算法逻辑；此外，还假设问题的规模超过Cell处理器的存储器容量，因此数据必须在附属处理器和主存之间来回传送。假设数据传送是唯一的约束，试计算排序2^{35}个四字（quad-word）所需的时间。
4. 假定要对如B-树或AVL-树等大型数据结构进行并发修改。解释如何使用事务来保证树结构的完整性。试解释改变求解的粒度对求解结果的影响。
5. 对习题4中所用求解方法与通常所使用的锁机制方法进行比较。如果除了要使完成插入时间最小之外，还需要使完成时间的变化最小，则上述求解方法中哪一个更好？

第11章　编写并行程序

学习并行程序设计的最好方法是实践。以前各章对有关的并行程序设计概念进行了论述。本章将关注程序设计活动本身。

在本书中可以看到进行并行程序设计的实践有两个机会。第一个机会是进行小的程序设计练习，如书中的举例以及每章后面的习题，它们为读者提供了探索新算法概念的机会或是学习一种新的程序设计语言的机会。对于了解特定的内容而言，这是一个很好的手段。本章后几节所覆盖的内容是另一个程序设计活动，该活动是一个内容丰富的学期课程作业，需要花费许多周的时间。学期课程作业为设计全新的算法和处理环绕并行程序设计的全方位复杂性提供了机会。上述的两类活动是清晰地理解本书内容的关键所在。

本章在论述并行程序的编写时，必然将采用通用以及与机器和语言无关的方式。当然，对读者来讲则必须使用一种特定的语言和特定的机器。本章中的大多数主题都与所有并行程序设计情况相关，但偶尔有一些问题并非如此。

11.1　起步

在考虑并行程序设计之前，建议你最好先熟悉目标并行计算机的访问和实际使用。

11.1.1　访问和软件

单用户系统的可用并行性通常局限于少数几个处理器，但它的访问显得比较方便。对于这种系统，编写并行程序的技巧与编写任何其他类型程序的技巧基本上没有什么差别。与此相反，多处理器系统通常是多用户系统，且通常以“远程”方式使用。在这种系统中进行编程，可能需要获得访问许可或是需在恰当的访问协议指导下进行访问。为社区服务的系统，它使用的信息通常在Web网页上张贴；对于相邻实验室中机群的访问安排，可能并不需要如此正规。

在协商访问事宜时，另一个需要注意的问题是需要验证在你想使用的目标平台上有所要使用的程序设计语言和软件工具。就这一点而言，多用户系统通常是较为方便的，因为其他人可能已经安装了你期望的系统。应注意的是，调试器、性能分析器等工具应同时一起安装。

11.1.2　Hello, World

传统地，用新语言或为新系统编写的第一个程序是一个很简单的计算，如“Hello,World”程序（它只是简单地打印问候），将在一个并行的平台上测试编译、连接、装载、排队（如果需要的话）等功能并运行一个并行程序。

一旦“Hello,World”成功运行，下一个任务便是在多处理器系统上运行类似的简单程序，组合来自所有进程的结果，并输出该结果。这种计算能解决派生线程或进程以及它们之间进行通信的问题。虽然这似乎是非常直截了当的，但在解决更繁复的任务之前最好先验证你已经对这些基本的操作有了充分的了解。

最后，在急切地开始进行程序设计之前，需要谨慎地测试是否能对程序的执行进行测量。并行计算工作的一个关键部分是度量它的性能，因此关键是能知晓如何获得精确的定时。由于前两个测试程序过于简单因此无法满足可测量时间总量的要求，为此需在线程中加入代码以使得对存储器的操作能迭代执行10^8次，例如交换两个变量的值：

```
temp=x;
x=y;
y=temp;
```

需要进行验证例如当迭代次数增加到10^9时能检测到性能的改变。

优化编译器 检查定时设施时，应确保打印出在定时的代码中所访问到的某些值（除了定时以外）。优化编译器通常会分析一个程序以确定哪些语句会促成最后的输出；然后编译器会删除那些无关的代码。通过从定时部分打印某些值，就可保证编译器不会将其删除。

在完成了这些预备步骤后。我们就为开发并行程序做好了准备。

11.2 并行程序设计的建议

本书的大多数读者已经是有经验的程序员，因此在编写成功的顺序程序时不需要帮助。为此，这里将集中讨论那些对并行独特的问题或是致力于对并行程序设计的活动。我们将遵从软件工程的原理，但并不采用软件工程中的专门技术。也许最好的忠告是三思而行。

11.2.1 增量式开发

由于并行程序通常比较复杂，这就使得创建和理解并行程序相当困难，因此如果有一个实际的方法学来指导并行程序设计将是很有帮助的。我们建议采用增量的方法使程序从一个工作版本前行到另一个工作版本。例如，可以在程序开始时只是初始化并行数据结构。这个初始的程序然后可以以小的改进增量式地加以修改，但在每一步时需要小心地全面测试新程序，以验证它能继续正常工作。在将新模块集成到程序的主体部分并与程序主体部分一起进行测试前，也可分别使用这一方法学对新模块进行设计、实现和单元测试。这一技术对寻找好的测试情景以及保持一个完整的开发历史给予了高度重视，因此当发现问题后就可能回滚到前一个工作版本。（我们建议使用诸如Subversion的版本控制系统）。

“小改进”有多大？当然它不是以代码行的多少来衡量的，而是以加入到程序中的复杂量和我们对它的正确性的信心来衡量。在程序中即使仅有几行代码影响了线程的交互，很可能导致很大的复杂性，需要仔细地进行检查。另一方面，程序的正文块特别是那些不含有与其他线程交互的过程，可以很容易地加入，即使它们可能占用了大量的代码。

11.2.2 侧重并行结构

我们提倡先构造并行结构然后插入功能成分。初看起来似乎有些绕圈子，但由于此时没有计算中功能部分的掩盖，所以通常会更容易地识别难于发现的并行错误。这种宏观的观点将引导我们把矩阵乘法看成是组合行和列而不是许多标量的乘法和加法，即使我们最终可能要在多个层次上应用并行化。宏观的观点通常有助于识别施加并行化能得到最大回报的所在处。

为了说明并行超结构的概念，设想我们的程序要完成以下任务：

- 初始化数据结构

- 派生一组线程，每个线程将在一部分数据结构上进行迭代操作，并在每个周期的末尾与其他线程进行交互
- 用归约操作汇总数据
- 打印结果
- 退出

图11-1显示了这个假想计算的超结构的示意图。

这些反映程序基本并行结构的步骤可以在说明计算的实际功能之前进行编写和调试。通过使基本结构在这种框架形式中工作，那么只有很少的场所中会隐藏与并行有关的隐错。

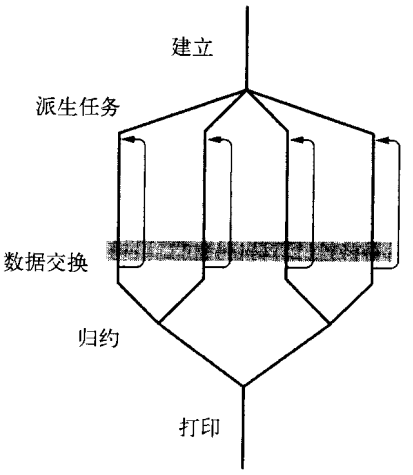


图11-1 一个代表性计算的并行超结构。控制流来自顶部；灰带标明了每个线程与其他线程进行集中交互的位置

11.2.3 并行结构的测试

一个复杂的问题是功能部分经常决定了并行部分的行为表现，这一点很重要因为形成的交互是导致复杂性和潜在缺陷的主要根源。如果线程的调度是预先确定的，如在Batcher的双调谐排序中（参见第4章）那样，则就很容易测试。但是如果线程的交互是数据相关的或是动态的，则就必须建立某种机制以再生典型的调度从而能对交互进行测试。当然确定调度的功能性应是可编程和可使用的，但是对于有效的程序开发，这可能太不可预测。所以我们提议首先在可控的条件下检查交互（或是输入一个测试调度或是程序设计一个可预测的调度）以验证交互的正确性。这些“测试模板”可以进行调节以检查不同的调度并增加对结果的信心。

11.2.4 顺序程序设计

与并行程序设计相比，顺序程序设计获得了程序开发工具更好的支持，因此利用这一事实是很有意义的。为此我们建议尽可能多地编写顺序代码，这与我们提议将顺序代码插入到并行超结构中是相互呼应的。能独立于主并行程序开发和测试的代码通常可以很有信心地加入。程序的功能部分通常属于这一类型。

11.2.5 乐意写附加代码

在编写并行程序代码时非常容易出现隐错，所以不应急着生成大量的代码而是应力图生成正确和有效的代码。即使侧重效率也可能产生误导，因为性能的优化经常会复杂代码和引入隐错。因此乐意撰写额外的代码以利于测试、调试和理解主并行程序是非常重要的。有许多例子表明额外的代码或工具是有用的：

- 有助于消除竞态条件，竞态条件难以识别是出了名的，有时可以创建一个测试控制以系统地对可变的线程数实施大量的线程交互集。这种控制可能需要作为陷阱安置在并行程序中以控制不同的执行交叉。这些陷阱对程序的某些部分可以调用人为的延迟，或者在线程系统中它们可以释放处理器。
- 为了方便测试，创建一个能生成各种各样程序输入的程序是非常有用的。而单纯地依赖设想不同程序的输入以图暴露代码中的隐含假设通常是无效的。
- 创建小工具以检查各种数据结构的状态通常很有用。例如，如果一个数据结构在分布式存储器机器中分布在多个处理器上，则若有一个打印例程能打印出一个全局一致的数据结构情况就非常有用，这比每个进程单独地打印局部的数据结构情况要好得多。这种例程可能要在进程间传递一个令牌以同步各种打印语句。
- 为了调试长时间运行的程序，实现检查点的设施是很有用的，检查点技术能周期地保存所有进程的执行状态，这样以后的执行就能从执行中的某点继续，而该点正是接近于出现隐错的位置。应谨慎设置检查点，以使它们所代表的时间点是确实会发生的。例如，考虑从进程p向进程q发送一个消息。如果一个检查点保存进程p在它发送消息之前的状态，而保存进程q在它接收消息后的状态，则将导致状态的不一致，因为不可能发生消息还未发出而却已被接收的情况，避免这种问题最简单的方法是在障栅处或其他的全局同步点上设置检查点。
- 为了解并行性能和性能的瓶颈所在，在程序中设置能开关的检测代码非常有用。这种代码能测量一个线程所完成的工作量，或是进行定时的测量。当然，当试图测量短事件时，这种定时的测量就会有问题。

与顺序程序一样，自由地使用断言能有助于快速地识别代码中那些违反假设的位置。

11.2.6 测试时对参数的控制

在测试程序时存在一个通用的趋势即创建一个小型的测试案例集，对于许多计算来讲这一方法完全能满足要求。可是由于并行计算的性质要求测试案例集必须足够大，使得分布在每个进程上的工作足够多才能显示出典型的交互。因此，如果测试一个顺序计算需要n个数据项我们可以期待t个线程的并行计算将至少需要tn个数据项，若考虑到整除性的因素，则可能需要更多的数据项。此外，我们需要有足够的处理器以促使生成一个丰富的交互集。遗憾的是，不存在一个普适的规则到底需要多少数据项方可满足要求，因为每一种情况都是不一样的。

最后，假设已有足够多的数据项，我们还需要为它们考虑各种交互的调度问题。如前所述，综合调度的好处是它们能在很大程度上加以控制。测试应包含所期待的各种交互类型，当然应当记住，即使调度如Batcher排序是可预测的，但线程到达同步点的确切时间是不知道的。如果交互是数据相关的或是动态的，则测试调度应该包括对一些极端调度情况的选择，例如所有任务等待同一个任务，或是一串任务依次等待它的先驱者等。

11.2.7 功能性调试

确定一个并行计算的答案是否正确的主要方法是比较并行程序的输出和另一个计算，通常是顺序计算的输出。这种策略对于处理离散对象的程序来讲是很合适的，但当计算中含有浮点算术操作时就不很可靠。因为并行程序会有不同（甚至是不可预测）的执行顺序，浮点运算的内在不精确性可能导致结果不会在每一位上都相同。在这种情况下，通常较为合适的处理方法是使用一个正确性的阀值，如“正确到精度d位”。

如前面几小节所暗示的，我们建议通过使用谨慎的开发过程使引入的隐错数降至最少。不幸的是，有效的并行调试工具几乎是不存在的。大多数程序员采用捕获数据并对它们进行检查的方法，而这将不可避免的影响他们程序的定时特性以及确切的功能性。

11.3 对结课课程设计的设想

前面几章着重用小练习来教授专门的论题。但更大的课程设计能教会除了并行程序设计概念之外的许多其他内容，如管理复杂性、开发有效的程序设计方法学以及实践合适的实验过程。在你所选择的课程设计中开发并行程序设计领域中的专门技术是对未来挑战的很好准备。你旁边有一台并行计算机，现在是为它编写一个真正程序的时候了。

有几种类型的计算可以作为结课课程设计（capstone project）。在选择求解问题时最好关注它要解决什么样的计算而不要关注它的并行行为。因为如果所选定的问题“非常并行”或是它已有一个明显的实现方法，则与问题的答案是个人很感兴趣的课程设计相比，通常会有较少的参与。显然，如果求解的问题需要大量的计算则为最好，但是当目的仅是为了学习时，实际上并非需要如此[⊖]。我们将可能的课程设计分为三类，下面分别进行讨论。

11.3.1 实现现有的并行算法

由于并行是一个长期研究的论题，因此在文献中有许多有关并行算法的描述。但这些描述通常都是非正规的或是以一种特殊的伪代码书写的，因此所给出的不是产品程序。例如第4章中为了说明可扩展并行计算的Batcher的双调谐排序是以Peril-L编写的（为求解按字母顺序排序任务）。这类课程设计的目的将是为一个特殊的平台生成一个可工作的实现以及描述它的行为。

初看起来该课程设计相当容易，因为似乎只要在书中找到一个算法并将其在当代计算机上以产品程序设计语言加以实现就可以了。但这里有一个忠告：即公开发表的并行算法通常都会有一些假设，而这些可能并不适用于可用的语言和计算机。作为一个极端的例子，存在有大量的有关PRAM算法的文献，但是如在第2章中所述，PRAM模型对于可扩展并行性是不现实的。最好的忠告是谨慎地检查文章并了解作者所做的假设。如果它们与可用的机器、语言和目的均符合，则就继续进行设计；否则就继续寻找。这类课程设计的构成要素为如下：

- 定位一个感兴趣的算法。
- 识别算法所基于的任何计算假设。
- 阐明为测量程序所需使用的典型测试输入。
- 用前面所给定的方法学开发一个程序。

⊖ 最好是顺序复杂性为$O(n^2)$或更高的求解问题。

- 度量整个性能，包括特殊子计算的性能。
- 基于所得到的测试结果和本书稍早所论述的主题（例如第3章），修改程序以改进它的性能。
- 书面报告课程设计的结果。

改进一个已发表算法的可能性看起来似乎很难实现，但这是可能做到的。算法的实现一般将会有许多机会进行改进，即使基本算法保持不变。

上述的构成表（下面两个小节中也有类似的构成表）看起来像是一个简单的操作步骤集。但是在每一步中都存在潜在的复杂性，该表可能并未穷尽，且这些步骤也不可能总是严格地按所给定的顺序依次完成。

11.3.2 与标准的基准测试程序媲美

第二类课程设计（与第一类有关）是在新的环境下实现一个标准的并行基准测试程序，并比较求解是如何完成的。多年来已开发了各种各样的并行基准测试程序套件组，有关的详细情况请参见Web网。其中受到高度关注的套件组是NAS Parallel Benchmarks (NPB)（http://www. nas.nasa.gov/Resources/Software/swdescriptions.html），它是由NASA的研究人员开发的。基准测试程序的优点是它通常是精心编写的（精巧设计的）且被广泛了解和认可，此外通常它已公布了性能数据。因此，如果我们以新的语言为该基准测试程序编写了一个新版本，或是使该基准测试程序具有了某种新的特性，那么我们就可将课程设计程序的运算结果已与公布的结果进行比较。这种竞争很有趣。

这类课程设计的构成要素为如下：

- 重读一个基准测试程序套件组中的所有程序，并确定其中感兴趣的一个。
- 查找报告该基准测试程序实验结果的那些论文，注意实现平台和实现语言等细节。
- 用前面所给定的方法学开发一个程序。
- 遵循文章中所使用的标准方法学和测试数据，度量整个性能，并与已公布的结果相比较。实现这一步的困难在于它带有一些迷惑性。例如，NAS Parallel Benchmarks提供一个具有不同大小的输入集，每一种大小适合于不同存储器容量的平台。如果基准测试程序套件没有设计得如此精巧，就需要修改测试的输入。
- 基于所得到的测试结果和第3章所论述的论题，确定为何所观察到的性能与已公布的结果有所不同。同样，这一步的实现可能比较困难，因为它通常需要知道体系结构的差别、程序的输入、程序设计的各种技术之间是如何影响的。此外，许多文章不提供足够的信息以产生可重复的实验。
- 书面报告课程设计的结果。

该题目的一种变化是考虑整个测试套件，不是重写这些程序，而只是简单地对它们加以改变从而达到探究体系结构某一特征的目的，例如将消息传递命令改换成单边通信形式或是改成共享存储器的访问方式。作为一个忠告，这种变化的简单性对于大多数的体系结构特征来讲是不可行的。

11.3.3 开发新的并行计算

最有兴趣的课程设计类型是用并行方式求解一个新问题。通过运用你自己的知识和对任务的创造，会获得构建一个可工作的并行程序的满足感，而且通过亲自解决所有的并行问题，这将是非常有意义的。

主要的要求是要对计算有足够的熟悉，这样就可以将它的组成部分以一种不同的方式（并行方式）加以表示，并获得正确的解答。作为一个设想源，表11-1给出了几个并行计算课程中所采用的课程设计报告的题目。显然，如果没有这份报告，我们无法确认已实现什么样的特殊计算，而且这些题目还建议了一些有重大意义的计算问题领域，这些问题值得进行并行求解。

这类课程设计的主要工作步骤为如下所示：

- 重新阅读有关计算的文献，如果你对它还不是十分熟悉的话。
- 开发一个计划，开始时的目标应是有限和可以达到的，如并行化核心部分的计算，然后根据需要逐步加入附加的功能层。由于采用增量式的开发方法，因此需要注意整个计划的完整性。特别是避免只并行化核心部分的陷阱，而在此后才发现大部分时间是花在非并行化部分的代码上。
- 使用前面所描述的方法学开发一个程序。
- 分析求解的行为，以了解性能是如何随求解问题的规模增大和处理器数的增加而发生变化的。
- 基于分析，改进任何瓶颈的性能。
- 如计划所描述的，加入下一层次的功能性，在所需之处实施并行化。
- 书面报告课程设计的结果。

在开始构造一个计划之前关键的一点是要克制过度雄心勃勃的倾向（我们都会这么做！）以及集中精力于关键目标：为所感兴趣的问题给出一个并行求解方法。

应注意的是，在使用前两个方法选择课程设计时（一个已公开发表的算法或与一个基准测试程序进行竞争），有关代码可能已经存在因此可以直接加以使用，特别是其中的功能部分。即使在第三种方法中，通常也存在能提供所需功能性的开源软件，尽管是顺序形式。由于获得正确工作的程序段相当容易和正当，因此使用这些程序就很有意义。

表11-1 并行计算课程的课程设计报告题目

下棋残局游戏	3-可满足性问题	基因序列排序
分段最小二乘法	视频运动检测	游戏的路径寻找
GPU的声频分析	MP3快速傅里叶变换	图像卷积
确切字符串匹配	KD-树构造	Boid仿真
Kohonen地图	光线跟踪	银河系仿真
质数因子分解	矩形分割	Kenser-Ney平滑
数据加密	跳棋的最小/最大搜索	人工神经网
光线铸造	朱莉娅集	约束满足
采样排序	旅行商问题	协同过滤

11.4 性能度量

通常编写并行程序的目的是为了加快求解问题的速度。在计算产生正确的结果后，随之而来的下一个任务便是测量它的性能。定时器的使用已经过检验（这是“Hello，World”后的下一个准备步骤），所以剩下的工作便是考虑如何应用以了解程序的性能。在第3章的论述中包含了与本小节直接相关的许多主题；这里假设读者已熟悉这些内容。

11.4.1 与顺序求解方法比较

关于并行程序每一个人会问的第一个问题是“相对于顺序程序，并行程序作了多少改进？”这个问题隐含着并行和串行计算执行时间的比较，这里的执行时间对大多数人来讲是指耗费时间或“墙钟”时间。称该参量为整体性能（overall performance）。

我们应该将程序中的哪些部分包含在整体性能的测量中？当定时程序时是否应该包括初始化和清理的代价？因为我们关心的是整个执行时间，因此定时整个程序就是合理的。但是，初始化和清理只是一次性开销，并不代表计算的核心，特别是在较长的有效运行时它们更将被计算的核心所弱化。也许最好的回答是了解程序中各部分的行为，并彻底弄清在报告结果时需要哪些测量的数据。

很重要的一点是，要弄清一个程序中每个阶段内的各个开销，因为阿姆达尔定律告诉我们如果某种开销（譬如初始化）本质上是顺序的，则这些开销将限制加速比。例如，如果磁盘I/O的时间占主导地位，那我们就知道对核心计算进行并行化不会有任何收益。此外，这种了解将允许我们去推断将并行化移向其他硬件的潜在得益，例如支持具有较低初始化开销的并行I/O。

最后，比较基础的顺序程序应该是具有“显著”竞争力的或是已知的最快顺序程序。第一种情况所涉及的程序是广泛应用的，所以为对比较感兴趣的读者所熟悉。第二种情况则适用于对通过并行化可使计算有多少改进感兴趣的场合。这里的顺序程序不应该是使用一个处理器的并行程序，除非已知最快的顺序计算也是处于这种情况（这是可能发生的！）。不论是运用哪一种情况，在报告结果时均应加以说明。

11.4.2 维护一个公正的实验设置

为了使任何比较有意义，必须对许多变量加以控制。如在第3章中所提到的，这些变量包括程序执行所使用的硬件平台、所使用的语言和编译器 、生成编译代码时所使用的优化设置以及程序的输入。例如，两个程序应在相同的计算机上运行，这就意味着顺序程序应在并行机的一个处理器上运行。

另一个较为复杂的问题是，难于生成可重复的结果。因为现代的计算机是多道程序的，在被测量的程序运行时，其他的程序以及不相关的任务也可能在同时执行，其范围涉及从后台的安全守护进程到其他用户程序，例如，当将一个服务器机群作为并行计算机使用时就可能出现这种情况。这种干扰的后果小到导致微小的定时不精确，大到计算结果的完全不可靠。主要的担心是，其他的计算会使用资源，例如网络或总线带宽，这将不可预测地影响被测量的程序。因此很显然测量性能时最好系统只有唯一的一个用户，即被测量的程序；当这种条件不能满足时，则很重要的一点是，累计足够多次的运行记录，识别其中运行缓慢的非测试部分并将它们剔除掉。

最小值及平均值 如果程序一般执行的设置是多用户的话，许多研究人员喜欢用平均执行时间来报道性能结果，但是中值（median）可能是一个更好的报道值，因为它忽略了少量差的非测试部分的影响。最小的执行时间也很重要，因为它指明了在最好的条件下可达到的性能。

性能不可预测性的一个潜在根源在于在测试时包括了操作系统和运行时系统。例如，某些程序需要大量的分页以从磁盘装载初始数据，且分页所需的代价会随操作系统的虚拟存储

器系统的状态而发生变化。通过“预热”系统可以抑制这种行为，并可使性能变得更可预测。预热的概念是使程序运行两次，而只测量第二次的执行时间。（当然，如果程序中存在冲突缺失，则应该进行统计，但初始的大量分页活动将被避免统计在内）。类似地，如果系统动态调用了即时（Just In Time，JIT）编译器，它也将以不可预测的方式影响存储器系统的行为，如果使系统进行预热，就可在进行测量之前使所有方法得到编译。另一个例子是，无用信息收集时间可能导致广泛的不可预测的执行时间，因此在如Java等支持垃圾回收的语言中，通常较为明智的做法是在测量之前立即强制执行无用信息的收集。

11.5 了解并行性能

我们的最终目标是要获得良好的性能，因此这就诱使我们测量加速比，在确认加速比曲线比较理想时就会宣布获得了成功。但是，如何能知道是否真正获得了成功？正如在第3章结尾处所讨论的那样，对加速比曲线的误解有多种方式。我们可能得到好的加速比是因为求解问题规模远大于可用的处理器数；而在实际上，我们的程序可能是非常低效的，所以如果将该求解问题移植到更大的实际机器上运行，此时可能看到无法接受的加速比。相反地，由于问题本身可能只具有很少的并行性，因此加速比可能较差。例如，如果我们能将如gcc编译器那样的程序加速1倍，将会非常激动。从理想角度而言，应该将我们的结果与其他人的结果进行比较。即使采用这样的方法也还是会有问题，因为许多的差异，如编译器优化、时钟速度、cache大小等，都可能显著地影响结果，使得它们难于进行比较。

因此，目标不仅仅是达到某种程度的加速比或效率。目标应该是开展对程序行为的深层次的了解，通过回答以下的问题就可达到这一目标：

- 程序有那几个阶段？通常了解单个阶段的行为比了解多个阶段的合成行为要容易。
- 程序的每个阶段的可扩展性是什么样的？通过使用各种不同的输入大小有助于对可扩展性的研究。此外，如果可能，也应使用各种不同的机器规模以及各种不同的机器类型。
- 每个阶段的瓶颈在哪里？开销、竞争和闲置时间的来源是什么？有时直接测量这些性能的损失比较困难，因此需要编写代码以汇集有关每个进程所完成的工作量和进程间的交互量的各种统计。当然，如海森堡测不准原理所指出的，我们企图考察的行为常常会干扰该行为，所以必须小心。
- 在各个阶段存在有多少并行性？对该问题的回答有时候可用分析方法，而有时候需使用经验方法。有多少通信需要进行，以及它们是什么样的通信形式？
- 使用了多少存储器容量？还有其他什么资源被大量使用？
- 为了改进性能可以进行哪些协调？在对性能瓶颈有了了解以后，一般就可确定改进性能的协调方法。粒度大小是否合适？能以何种方式改变粒度大小？

当然，许多这些问题只能在特定的机器环境下进行回答，因此在多台机器上实施上述的实验通常是饶有兴趣的。

11.6 性能分析

有时候有可能完全通过分析来推断一个程序的性能缺点，虽然对重大并行程序进行成功的程序分析对我们来讲可能过于复杂，但是值得一试。至少，通过程序分析我们可以发现一组潜在的性能损失点，对这些点需要进行实验的研究。

对完全需顺序完成的大部分计算进行检查是十分明智的。这些部分在程序设计时可能已经进行了仔细的考虑和解决，但再次对它们进行检查无疑是明智的。程序中是否有残存部分是顺序的或是只显示很少并行性？如果有的话，则是否有其他的求解方法，也许会涉及如并行前缀那样的技术？

分析粒度的问题也很重要。子计算的一个共性问题是相对于开销它所完成的工作太少。这种子计算的例子包括如排队和出队那样的微小工作，它们所完成的工作量很小，此外还会引起队列的拥堵。另一个例子是以并行方式去完成微小的计算，而这个计算在一个进程上可能完成的更快。例如，假定P个进程中的每一个含有一个值，计算要求对它们进行排序。在P个进程上进行并行排序将导致大量的进程间通信，而如果将这些值送到一个进程上进行排序可能会快得多。

11.7 实验方法学

通过运行实验来寻找性能损失的原由就如求解一个神秘的事物。关键的一点是，需要有一个前提来指导搜索；从分析中可能已得到一张性能损失的候选原由表。其他的原由是负载不平衡、锁竞争、过度通信等，这已在第3章中作了论述。一个好的方法应是尽可能地对程序的各个部分进行集中测试。

为了保证成功，强调再现性是有益的。在通常的顺序调试时，可能需要在代码中设置一个断点，并从那里开始研究。但并行程序经常依赖于异步事件（特别是当它们含有隐错时），所以它们不能随意在任何地点停止。通常的做法是在代码中设置一个障栅，为进行调试打印出一些值，这仅是为了使问题不复存在。当然，具有一个可再现的状态以再现和发现bug，也是非常关键的。

要在一个计算的中途捕获一个可再现的状态，我们必须停止计算并编写状态的一个一致性版本，这在前面已经作了讨论。首选的情景是寻找一个自然的障栅，在该障栅处所有的进程都必须等待；障栅可以设置成是显式的，或是设置成如归约操作那样隐式的。在这些障栅的设置处，程序的停止或启动将对定时产生最小的扰动。如果在这些场所不便设置这样的障栅，则就需要进行障栅布置的试验，以使它不会隐藏被研究的行为。

例如，如图11-1所示的程序结构并不含有任何显式的障栅，尽管在归约操作后有一个隐式的障栅，就辅助我们限制计算的规模而言似乎过晚。但我们应注意到，在灰色区域处理器正在交换信息，且很可能它们正在更新状态以确定是否需要继续进行迭代。即是说，所有控制将停止直至每个线程决定是否要继续执行，这种决定是依据每个线程的局部计算或是来自一个进程的广播值，该进程所完成的操作结果值是所有线程所需要的。图11-2中给出了图11-1中计算的示意图。其中隐含的障栅就是停止和启动计算的所在场所。

协议使计算运行直至遇到一个障栅，在障栅之后立即捕获程序的状态并将数据写入一个文件。该方法允许程序以后障栅（post-barrier）状态重复地重新启动。障栅并不是瞬时的，而且它们并不总是处于最理想的地点。但不管如何，这种检查方式可限制测试程序（感兴趣段）偏离实际程序的程度。

该方法学允许我们对程序中的一部分进行测试而无需从起点运行程序，这可能是一个显著的简化。至于其他简化的可能性就不大了。例如，我们可能需要在足够多的处理器上执行或是运行一个足够大的计算以产生我们猜想可能导致性能损失的行为。

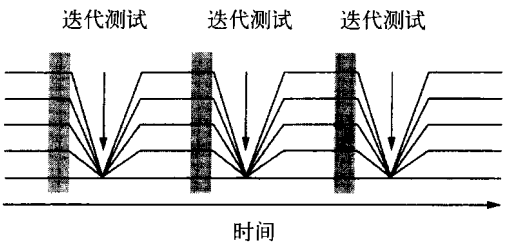


图11-2 图11-1结构中$P=5$个进程时的执行示意图。在每次迭代测试时，进程根据数据交换情况（图中以灰色表示）决定是否继续进行迭代

11.8 可移植性和微调

在编写一个可扩展的并行计算程序后，我们经常希望它能在各种各样的平台上运行。如果能够明确无误地运用在第2章到第5章中所描述的那些原理，那么我们所编写的程序应该在大多数的并行平台上能很好地运行，并且可以期待在不同的平台上获得类似的性能。但是由于各种计算机在细节上有所不同，因此在一个新的平台上执行程序很可能需要对程序进行微调。

11.9 小结

本章的论述的重点是编写和调试并行程序的过程。虽然大部分的指导意见适用于任何规模的并行程序设计，但重点是在规模较大的课程设计。这类课程设计比较复杂，需要使用谨慎的方法学。我们为程序设计、功能调试以及性能调试提供了通用的指导原则，这些原则在许多情况下都是适用的。每一种特殊的情况有其自身的特征，因而在很大程度上依赖于程序员的智慧和技巧，但是这些指导原则将会很有帮助。

历史回顾

并行检查点的讨论基于分布计算社区所使用中的有效恢复线（valid recovery line）的概念，Elnozahy等[2002]在他们的综述中作了描述。考虑到通用性，我们没有强调并行程序设计的工具，因为它们会倾向于和专门的程序设计设施（如OpenMP或消息传递）相结合。不论如何，鼓励读者寻找与自己的程序设计环境相兼容的可用系统。

习题

1. 以图11-1作为指导，描述第4章中图4-7内的Batcher按字母顺序排序算法的并行结构。
2. 以图11-1作为指导，描述第4章中图4-5内的固定并行性按字母顺序排序算法的并行结构。
3. 遵循起步这一小节中的指导，熟悉一台并行计算机，并进行基本的定时测量。
4. 对统计3计算进行编程，并在习题3中的并行计算机上运行。
5. 使用你选定的一个并行程序，计算该程序可达到的绝对加速比；画出该值随处理器数增长的变化图。

术 语 表

Amdahl's Law（阿姆达尔定律） 使用并行改进计算性能的一种限制：如果$1/S$是完成顺序工作所需的计算时间，则对该问题的并行求解可能提供的最大加速比为S。

Atomicity（原子性） 原子地执行一串操作的特性；如果所有的操作要么全出现要么全不出现，那么这个操作的执行就是原子的。保证原子性的通常方法是用互斥体保护这一串操作；一旦获得互斥体，代码将不中断地执行直至互斥体被释放。

Cache Coherence（cache一致性） 一种cache管理协议，在该协议中当访问一个指定的存储单元时，各一个处理器的cache所看到的是一个一致值。

Chapel Cray公司正在开发的一种并行程序设计语言。

Cilk 基于ANSI C的一种多线程并行程序设计语言，由MIT（麻省理工学院）开发。

Chip Multiprocessor（CMP）（芯片多处理器） 含有多处理器的单芯片；也称为多核芯片；目前CMP的处理器数小于20个。

Cluster（机群） 主要由商品化组件构成的一种并行计算机；这种系统的规模能扩展到几千个处理器。

Co-Array Fortran Cray公司开发的一种扩展Fortran的PGAS语言。

Contention（竞争） 对共享资源的争用，将导致性能的下降。

Critical Section（临界区） 必须访问共享数据的一段代码，对该代码的无纪律访问可能导致竞态条件的出现。

Data Dependence（数据相关性） 在需要按某种次序进行存储器访问时而引起的存在于两个控制线程间的一种关系。

Data Parallelism（数据并行）将操作并行作用到数据中不同项上一种计算。采用数据并行时，能同时完成的操作数随数据量的增加而增长。

Dependence（相关性） 两个事件上的一种顺序约束；相关性限制了计算的潜在并行化；跨线程或进程边界的相关性需要进行通信和/或同步。

Distributed Memory Parallel Computer（分布式存储器并行计算机） 在硬件上不实现一个共享地址空间的一种并行计算机。

Efficiency（效率） 定义为 加速比/P的一种性能指标，其中P是处理器数。

Embarrassingly Parallel（易并行） 线程或进程间只有很少或没有相关性的一种计算。

Erlang 由爱立信计算机科学实验室设计的一种并行函数式语言。

F-- Co-Array Fortran的原名。该名字与C++名字相对应。

False Sharing（假共享） 独立进程各自私有处理的两个对象分配在同一cache行上的一种现象，它将导致当一个处理器改变其在cache行中的值时，cache一致性协议会（错误地）使另一个处理器中的cache行变为无效。假共享将使性能下降，即使所访问的数据是互相独立的。

FLOPS（每秒浮点操作数） 它是衡量科学计算的一种性能指标，它的性能主要由浮点性能规定。

Fortress Sun公司研究人员设计的一种并行程序设计语言。

Global View Abstractions（全局视图抽象） 不论运行时有多少个进程数或这些进程如何安排，如果给定相同的输入，使用全局视图抽象所书写的程序将产生相同的输出。

GPGPU 在图形处理部件上进行通用计算。

High Performance Fortran （HPF）（高性能Fortran） 扩展顺序Fortran 77和Fortran 90的一种数据并行语言，在该语言中加入了描述如何在进程间分布数据的一些编译命令。

Idle Time（闲置时间） 由于没有工作可做或是正在等待某个事件而被阻塞，进程不能完成有用工作，它是导致低效的一个原由。

ISA（Instruction Set Architecture）（指令集体系结构） ISA是硬件的一个接口，它定义了硬件需实现的指令和数据类型。

Latency（时延） 对完成一个计算所需总时间的一种度量。该度量常与吞吐率相对比。

Locality（局部性） 对程序的存储器访问所期望的一种簇聚特性：时间局部性出现在当访问一个存储器单元在时间上显现出密集访问情况；空间局部性出现在当访问一个存储单元时显现出地址密集的情况。

Load Imbalance（负载不平衡） 低效的一个原由，由于各个进程完成不同的工作量导致某些进程完成执行远先于其他进程，从而形成闲置进程的出现。

Local View Abstractions（局部视图抽象） 依赖于运行时的进程数或这些进程的布置，并行程序设计工具将会产生不同的输出。

Memory Controller（存储器控制器） CPU（包括它的片上cache和DRAM）间的接口，它有助于对存储器的高效访问；控制器完成如缓存未完成的存储器写那样的操作，以减少为等待存储器写操作完成而导致CPU停顿的可能性。

MIMD 弗林分类中的一种多指令流多数据流计算机，通常是指并行计算机；它的主要特性是每个ALU执行自己的指令流。

Moore's Law（摩尔定律） 英特尔公司创建人戈顿·摩尔提出的一个预测，即芯片上晶体管的密度大约每18个月增加一倍；随着工艺的进展，半导体工业界一再发现芯片的发展遵循这一指导路线。

Multi-core Chip（多核芯片） 含有多个处理器的单个芯片；也称为芯片多处理器或CMP；目前这种芯片含有的处理器数小于20个。

Mutual Exclusion（互斥） 一段代码段的特性，它能保证在任何时间将只有一个线程执行该段代码；经常借助互斥代码不能为其他线程所中断这一特性来提供原子性。

$N_{1/2}$ 对于给定的程序和机器，为获得1/2的效率所需的问题规模。

NESL 由卡内基·梅隆大学开发的一种能提供嵌套并行的并行函数式语言。

NUMA Architecture（非一致存储器访问体系结构） NUMA体系结构实现了一个共享地址空间，由于存储器在物理上是分布的，因此对存储器的访问时间是不一致的；处理器本地存储器的时延低于非本地的存储器。

Overhead（开销） 在相应的顺序计算不会出现的任何并行计算的代价。它是引起并行程序性能损失的四个基本原因之一。

P 并行机中处理器数的缩写。

Parallel Virtual Machine（PVM）（并行虚拟机） 一种消息传递接口，它的许多概念后来被融入到MPI标准中。

PGAS Languages（分区的全局地址空间语言） 一种分区的全局地址空间并行程序设计语言；这种语言提供了一个为所有进程共享的单一地址空间。因此在所有进程中可用相同名字访问一个对象；该全局地址空间并不意味它是一个一致的存储器。

P-Independence（*P*-独立） 一个并行程序是P-独立的，当且仅当在相同的输入下总是产生相同的输出，而不论在程序上有多少个进程在运行以及这些进程是如何布置的。

Pipelining（流水化） 一种并行化形式，它将一个计算分解成一系列按序执行的子计算；只要有足够的资源，每个子计算就能在不同的问题实例上同时执行，从而可改进吞吐率。

PRAM（并行随机访问机） 它是并行计算机的一种理想模型，该模型假设对共享存储器的访问时间和同步执行时间为一个单位时间。

Process（进程） 包含自己地址空间的一个并行单位。

Processor（处理器） 并发性的一个物理部件，又称为CPU（中央处理部件）。

Race Condition（竞态条件） 程序的输出依赖于不可预测的定时行为的一种情况。

Reduce（归约） 将一个函数作用到一个集合值以产生单个结果的一种操作；通常这类函数是可以结合和交换的，包括加、乘、最大值和最小值。

Relative Speedup（相对加速比） 并行性能的一种度量，其中用P=1个处理器的并行执行时间替代最好的顺序执行时间T_S。

Relaxed Memory Consistency（松弛存储器一致性） 并行处理器的一种特性，其中一个执行所产生的结果并非是顺序一致的。

Scan（扫描） 将一个函数作用到一个有序集合值以产生所有前缀的一种操作，其中第i个前缀是将该函数作用到前i项所产生的结果；也称为并行前缀操作符；该函数与归约类似，通常为加、乘、最大值和最小值。

Sequential Consistency（顺序一致性） 并行处理器的一种特性，即任何执行所产生的结果将与按下列规则执行的结果相吻合，应遵循的规则是：（1）所有处理器的操作以某种顺序次序执行，（2）每个处理器的操作按程序所说明的次序出现。

Shared Address Space Parallel Computer（共享地址空间并行计算机） 一种并行计算机，其硬件允许所有处理器访问一个单一共享地址空间；这种机器通常需实现cache一致性。

SIMD 弗林分类中的一种单指令流多数据流计算机，也指向量处理器，在其中由一条指令控制多个ALU以锁步方式执行。

SISD 弗林分类中的一种单指令流单数据流计算机，通常是指顺序计算机或冯·诺依曼计算机。

SMP（Symmetric Multiprocessor）（对称多处理器） 一种共享地址空间并行计算机，其中的存储器时延与请求访问的处理器无关；因此所有处理器所看到的存储器视图是对称的。

Speedup（加速比） 对并行性能的一种度量；对于给定的一个计算，执行最快的顺序程序所需的时间T_S与在P个处理器上执行并行程序所需时间T_P两者的比值；通常以相对于x轴上的P值所画得曲线加以表示。

SPMD 指明为单个程序的一个并行程序在不同的处理器上被执行，如其所名，即单程序多数据；偶尔会与SIMD混淆，但SIMD是硬件分类，而SPMD是软件分类。

Superlinear Speedup（超线性加速比） 在P个处理器上获得大于P倍的加速比；其效率大于1。

Task Parallel（任务并行） 任务并行计算是通过对功能进行划分而获得并行性的一种计

算，因此被划分的功能能并发地完成。

Thread（线程） 一种并行单位，逻辑上由一段程序代码、一个程序计数器、一个调用堆栈以及包括一组通用寄存器在内的某些适量状态所组成；在基于线程的并行程序设计中，线程共享对存储器的访问。

Throughput（吞吐率） 对在单位时间内所完成工作量的一种度量；该度量常与时延相对比。

Titanium 加州大学伯克利分校正在开发的一种扩展Java语言的PGAS语言。

Transactional Memory（事务存储器） 受数据库中事务概念启发而研制的一种并发性控制系统，以控制共享存储器中并行程序的存储器操作。

UMA （Uniform Memory Access）Architecture（一致存储器访问体系结构） 一种共享地址空间体系结构，如SMP，在其中对存储器的访问时间是与请求访问的处理器无关的，与NUMA体系结构相对应。

Unified Parallel C（UPC）（统一的并行C语言） 由乔治·华盛顿大学开发的扩展C语言的PGAS语言。

X10 IBM正在设计的一种并行程序设计语言。

ZPL 由华盛顿大学开发的一种数据并行数组语言。

参考文献

Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. "Unlocking Concurrency," *ACM Queue*, 4(10), pp. 24–33, December/January 2006–2007.

G.M. Amdahl. "Validity of the single-processor approach to achieving large scale computing capabilities," in *AFIPS Conference Proceedings*, AFIPS Press 30, pp. 483–485, 1967.

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. "Hoard: A Scalable Memory Allocator for Multithreaded Applications," *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. November 2000.

Guy Blelloch. "Programming Parallel Algorithms," *Communications of the ACM*, 39(3), March 1996.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System," *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 207–216, 1995.

W. J. Bouknight, Stewart A. Denenberg, David F. McIntyre, J. M. Randall, Amed H. Sameh, and Daniel L. Slotnick. "The ILLIAC IV System," in *Computer Structures: Principles and Examples*, Daniel Siewiorek, C. Gordon Bell, and Allen Newell, eds., pp. 306–316, McGraw-Hill, 1982.

Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. "Problem space promotion and its evaluation as a technique for efficient parallel computation," In *Proceedings of the ACM International Conference on Supercomputing*, 1999.

Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. "ZPL's WYSIWYG performance model," *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.

Edward G. Coffman, Jr., M. J. Elphick, and Arie Shoshani. "System Deadlocks," *ACM Computing Surveys*, 3(2), pp. 67–78, 1971.

Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters," *Sixth Symposium on Operating System Design and Implementation*, pp. 137–149, December 2004.

Steven J. Deitz. "High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations," Ph.D. thesis, University of Washington, February 2005.

Michael J. Flynn. "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, p. 948, 1972.

Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.

K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory consistency and event ordering in scalable shared-memory multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26, May 1990.

Per Brinch Hansen. "Structured multiprogramming," *Communications of the ACM*, 15(7), pp. 574–578, July 1972.

W. Daniel Hillis and Guy L. Steele Jr. "Data parallel algorithms," *Communications of the ACM*, 29(12), pp. 1170–1183, December 1986.

C.A.R. Hoare. "Monitors: an Operating System Structuring Concept," *Communi-cations of the ACM*, 17(10), pp. 549–557, October 1974.

Roger W. Hockney. "Supercomputer Architecture," *Infotech State of the Art Conference: Future Systems*, Infotech Intl. Ltd., 1977.

C.R. Jesshope and Roger W. Hockney. *Parallel Computers 2*, Adam Hilger, 1988.

David J. Kuck. "A Survey of Parallel Machine Organization and Programming," *ACM Computing Surveys*, 9(1), pp. 29–59, March 1977.

Richard E. Ladner and Michael J. Fischer, "Parallel Prefix Computation," *Journal of the ACM*, 24(4), pp. 831–838, October 1980.

Ralf Lämmel. "Google's MapReduce Programming Model—Revisited," *Science of Computer Programming*, 68(3), pp. 208–237, 2007.

L. Lamport. "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, C-28(9), pp. 241–248, September 1979.

James R. Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*, Morgan and Claypool, 2006.

Gil Lerman and Larry Rudolf. *Parallel Evolution of Parallel Processors*, Perseus Publishing, 1994.

William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. "Cg: A System for Programming Graphics Hardware in a C-Like Language," *International Conference on Computer Graphics and Interactive Techniques*, pp. 896–907, 2003.

R. W. Numrich and J.K. Reid. "Co-Array Fortran for Parallel Programming," *ACM Fortran Forum*, 17(2), pp. 1–31, 1998.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, 26(1), pp. 80–113, March 2007.

Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. "Interpreting the Data: Parallel Analysis with Sawzall," *Scientific Programming*, pp. 277–298, 2005.

D. Pothen, H. Simon, and K. P. Liou. "Partitioning sparse matrices with eigenvectors of graphs," *SIAM Journal of Matrix Analysis and Applications*, 11(1990), pp. 430–452.

Ravi Rajwar and James R. Goodman. "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 294–305, 2001.

Burton J. Smith, "A pipelined, shared resource MIMD computer," *International Conference on Parallel Processing, IEEE*, 1978.

Lawrence Snyder. "Type architecture shared memory and the corollary of modest potential," *Annual Review of Computer Science*, 1, 1986.

Lawrence Snyder. "Design and Development of ZPL," *Third History of Programming Languages Conference, ACM*, 2007.

J.T. Schwartz. "Ultracomputers," *ACM Transactions on Programming Languages and Systems*, 2(4), pp. 484–521, 1980.

V.S. Sunderam. "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, 2(4), pp. 315–339, December 1990.

Herb Sutter and James Larus. "Software and the Concurrency Revolution," *ACM Queue*, 3(7), pp. 54–62, September 2005.

David Tarditi, Sidd Puri, and Jose Oglesby. "Accelerator: Using Data-Parallelism to Program GPUs for General-Purpose Uses," *The Twelth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, pp. 325–335, November 2006.

Michael Wolfe. *High Performance Compilers for Parallel Computing*, Addison Wesley, 1996.

Special Issue on BlueGene, *IBM Journal of Research and Development*, 49(2/3), 2005.