并行计算系列丛书

并　行　计　算

——结构·算法·编程

(修订版)

陈国良　编著

高等教育出版社

㈠ 112 号

内 容 提 要

本书是教育部“高等教育面向 21 世纪教学内容和课程体系改革计划”的研究成果，是面向 21 世纪课程教材和教育部理科计算机应用“九五”规划教材。

本书以并行计算为主题，主要讨论并行计算的硬件基础——当代并行计算机系统及其结构模型，并行计算的核心内容——并行算法设计与并行数值算法以及并行计算的软件支持——并行程序的设计原理与方法。本书强调融并行机结构、并行算法和并行编程为一体，着重讨论并行算法的设计方法和并行数值计算算法，力图反映本学科的最新成就和发展趋势。

全书共十五章，分为四篇：第一篇包括并行计算机的系统结构模型，当代对称多处理机、大规模并行处理机、机群系统和并行计算的性能评测；第二篇包括并行算法的一般设计策略、基本设计技术和一般设计过程；第三篇包括矩阵运算、稠密与稀疏线性方程组的求解和快速傅里叶变换；第四篇包括并行程序设计基础、共享存储与分布存储系统并行编程以及并行程序设计环境与工具。

从并行计算的角度，本书体系完整，内容丰富，取材新颖，可作为高等学校计算机及相关专业的本科高年级学生和研究生的教学用书，也可供计算科学与工程（Computational Science and Engineering）学科的研究生和科技人员阅读参考。

面向 21 世纪课程教材

本书初版曾获 2000 年度中国高校科学技术进步一等奖

作 者 介 绍

陈国良，中国科学技术大学教授，博士生导师，中国科学院院士，1938 年 6 月生于安徽省颍上县，1961 年毕业于西安交通大学无线电系计算机专业。1981—1983 年在美国普度大学作访问学者，1984 年至今曾多次应邀赴东京大学、普度大学、澳大利亚国立大学、新南威尔士大学、昆士兰大学、格里福斯大学、堪萨斯城市大学、依阿华大学、威斯康星大学、Maharish 国际大学、香港理工大学、澳门大学、北京大学、国防科技大学等讲学交流。现任国家高性能计算中心（合肥）主任，国际高性能计算（亚洲）常务理事，中国计算机学会开放系统专业委员会副主任，中国数学会计算数学并行计算专业委员会委员。曾任国家教育部高等学校计算机科学与技术教学指导委员会副主任，全国高等教育电子、电工和信息类专业自考指导委员会副主任，安徽省高校计算机基础课程教学指导委员会副主任，中国计算机学会理事，安徽省计算机学会理事长，全国自然科学名词审定委员会委员，中国科学技术大学计算机系主任。

陈国良教授长期从事计算机科学技术的研究与教学工作。主要研究领域为并行算法、并行计算机体系结构和智能计算等。先后承担 10 多项国家 863 计划、国家攀登计划、国家自然科学基金、国家 973 计划、教育部博士点基金等科研项目。取得了多项被国内外广泛引用、达国际先进水平的科研成果，发表论文 200 多篇，出版著作 9 部、译著 5 部，参与主编计算机类辞典、词汇 5 部，主审、主编计算机类各种教材 8 部。曾获国家科技进步二等奖、国家级教学成果二等奖、国家教育部科技进步一等奖、中国科学院科技进步二等奖和自然科学三等奖、全国优秀教材一等奖、全国学术著作优秀奖、安徽省科技进步二等奖、国家科委高技术研究与发展计划三等奖、国家教委科技进步三等奖共 18 项，并获 2001 年度“国家 863 计划 15 周年先进个人重要贡献奖”。

陈国良教授在中国科学技术大学执教 30 多年。长期以来，围绕着并行算法的教学与研究，逐渐形成了一套完整的“算法理论—算法设计—算法实现—算法应用”的并行算法学科体系，营造了我国并行算法类的教学基地。他先后指导培养研究生 100 多名，其中博士生 60 名，为我国培养了一批在国内外从事算法研究的高级人才。曾荣获 1998 年度安徽省教育系统劳动模范、安徽省优秀教师称号、2001 年度宝钢教育基金优秀教师特等奖和 2003 年度第一届高等学校教学名师奖。

陈国良教授是我国非数值并行算法研究的学科带头人。他率先创建的我国第一个国家高性能计算中心是我国并行算法研究、环境科学与工程计算软件的重要基地，在学术界和教育界有一定的影响和地位。

序　言

高性能计算机是一个国家经济和科技实力的综合体现，也是促进经济、科技发展，社会进步和国防安全的重要工具，已成为世界各国竞相争夺的战略制高点。一些发达国家纷纷制定战略计划，提出很高目标，投入大量资金，加速研究开发步伐。多年来，随着大规模集成电路技术的不断进步，以多 CPU 为基础的高性能并行计算机得到了迅速的发展，其高端系统正向百万亿次、千万亿次迈进。我国近十年来，对高性能并行计算的研究开发也给予了很大重视，取得了长足进步和可贵经验，研制出了具有相当水平的并行机系统，但与发达国家相比，差距仍然甚大，在高性能并行计算的应用开发与相关的人才培养教育方面尤显不足。如何使高性能并行机系统深入充分地在国民经济、科研和社会应用的发展中发挥作用，实为当务之急，引起人们的普遍关心。

由中国科技大学陈国良教授主编的这套丛书，正适应了我国高性能并行计算研究、开发、应用、教育之需。本丛书由《并行算法的设计与分析》、《并行计算机体系结构》和《并行算法实践》三大部分组成，而以《并行计算——结构·算法·编程》为全丛书之提要。该丛书以并行计算为主题，对并行计算的硬件平台（当代主流并行计算机系统）、并行计算的理论基础（并行算法的设计与分析）和并行计算的软件支撑（并行程序设计）全面系统地展开了讨论，内容丰富，取材新近，具有相当的深度和广度，涵盖了并行计算机体系结构和并行算法的理论、设计和实践的各个方面，是国内外不多见的优秀著作。

陈国良教授是国家高性能计算中心（合肥）主任，长期从事并行算法和并行计算机体系结构的研究，本套丛书是作者几十年从事教学与科研工作的结晶，是目前国内该领域内容涵盖最为全面的系列著作。它的出版必将对进一步推动我国并行计算学科的发展与应用推广产生深远的影响。

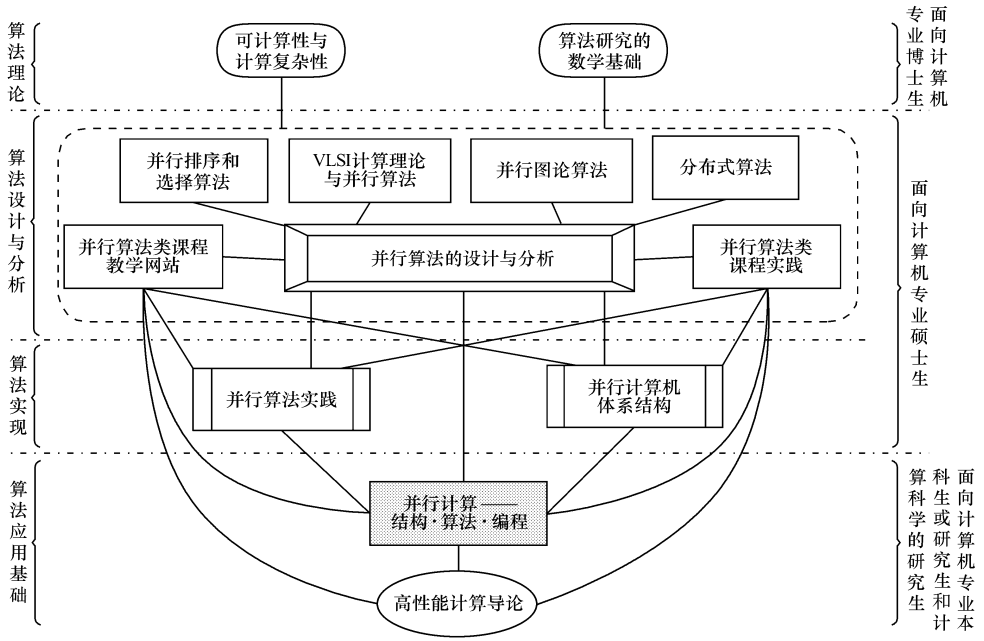
张效祥

2002 年 8 月

修订版前言

并行计算系列丛书 并行计算系列丛书包括《并行计算——结构·算法·编程》(修订版)、《并行算法的设计与分析(修订版)》、《并行计算机体系结构》和《并行算法实践》。

该丛书是并行算法类教学体系中的核心内容,而《并行计算——结构·算法·编程》处于并行算法类教学体系中的算法应用基础层次,面向计算机专业本科高年级学生或从事计算科学的研究生,是并行计算丛书中的第一本。



修订说明 《并行计算——结构·算法·编程》自初版以来受到了全国计算机专业和面向科学工程计算专业广大读者的欢迎。根据这几年的具体教学实践和计算机科学技术的发展,考虑到整个系列丛书的内容统一协调,同时借本书重新排版之际对该书做了简单的修订。主要修订部分为:①改写了第一篇的第一章、第二章和第三章,其中删去了第一章的1.1.3节,并在1.2.2节中加入了静态互连网络的嵌入一小节;将第二章的2.2.2节"机群型大规模并行机 SP2"代之以"Intel ASCI Option Red MPP 系统",而将其内容移至修订后的第2.3节"机群系

统”中，同时将第 2.4 节“国产曙光系列并行计算机系统”的内容移至丛书之《并行计算机体系结构》的第 1.6 节中；在第三章中增加了“并行计算机的一些基本性能指标”（第 3.1 节），同时将原 3.2 节“可扩放性评测标准”进行了精简，更详细的内容请参见丛书之《并行计算机体系结构》的第 2.3.2 节。② 在第二篇中，将第四章中第 4.2.5 节“C^3 模型”的内容移至丛书之《并行算法的设计与分析》的第 1.2.2 节中。③ 改写了第四篇的第十二章、第十三章和第十五章，其中将原第十三章的 13.1.1 节“五种并行编程风范”移至第十二章的 12.1 节之下，以及将 13.2 节“并行程序设计模型”移至第十二章而变为 12.6 节，同时仍保留了进程、线程、同步和通信的内容；第十三章的标题改为“共享存储系统并行编程”，除保留了原第十三章的共享存储并行编程和 ANSI X3H5 和 POSIX 内容外，重新改写和充实了 OpenMP 编程内容，同时在章末给出了“OpenMP 运行库例程”附录；对于第十五章，除了保留原有内容外，重新改写和充实了“并行程序调试”和“并行程序性能分析”的内容。④ 补充了某些新的参考文献和“专业术语中英对照及索引”，并对全书的错误进行了改正，但有些错误可能仍然存在。

修订后的建议讲授章节及其学时分配如下：

章节名	建议讲授节次	建议学时
第一章	1.1.2,1.2.2,1.2.3,1.2.4,1.3.1,1.3.2	4
第二章	2.1.1,2.2.1,2.3.1,2.3.2	4
第三章	3.2,3.3	4
第四章	4.2	4
第五章	5.1.1,5.2.1,5.2.3,5.3.2	2
第六章	6.1,6.2.1,6.3.1,6.4.1,6.5.1	4
第七章	7.1,7.2,7.3,7.4.1,7.5	2
第八章	8.1,8.3	2
第九章	9.1,9.4.1,9.4.2,9.4.3,9.4.4	4
第十章	10.3.1,10.3.2,10.3.3,10.4.2,10.4.3,10.4.4,10.4.5,10.4.6	6
第十一章	11.1.2,11.1.4,11.3.1,11.3.2	4
第十二章	12.1.3,12.3,12.6	2
第十三章	13.1.1,13.2.2,13.3	4
第十四章	14.2,14.5	6
第十五章	15.2,15.3,15.4,15.5	4

中国科学技术大学计算机系的徐云副教授以及曾修读过该课程的一些同学多次参加了修订本书的讨论，提出了很多宝贵的意见。孙广中博士和钟诚博士具体帮助我修订了此书，对于他们的良好愿望和辛勤的劳动，作者在此一并表示感谢。

陈国良
中国科学技术大学　计算机系
国家高性能计算中心（合肥）
2003 年 4 月

初 版 前 言

写作背景 近几年来,世界上和我国高性能并行计算机的发展取得长足进展,每秒数百亿次、数千亿次乃至数万亿次计算能力的高端并行机已相继研制成功,使得以前许多无法求解和研究的问题现在已经成为可能。随着计算技术和计算方法的飞速发展,当今几乎所有学科均趋向定量化和精确化,从而产生了诸如计算物理学、计算化学、计算材料学、计算力学、计算生物学、计算气象学和计算电子学等新兴学科,在世界上逐渐形成了所谓计算科学与工程 CSE (Computational Science and Engineering) 的计算性学科分支。计算,增强了人们从事科学研究的能力,拓宽了人们洞察自然的视野,加速了科技转化为生产力的过程,深刻地改变着人类认识世界和改造世界的方法和途径。计算科学 (Computational Sciences),已经和传统的理论科学与实验科学并列成为第三门学科,它们彼此相辅相成地推动着人类科技发展和社会进步。

在此情况下,为了适应高性能并行机迅速发展的形势,满足国家培养面向 21 世纪高科技人才的需求,在高校开设高性能并行计算 (High Performance Parallel Computing) 课程已提到了议事日程。为此,国家教育部高等学校计算机科学技术教学指导委员会经过多次讨论,把《并行计算》这本教材列为国家"九五"教材规划,并最后审定为"面向 21 世纪课程教材"。

篇章内容 本书以并行计算为主题,主要讨论并行计算的硬件基础——当代并行计算机系统及其结构模型,并行计算的核心内容——并行算法设计与并行数值算法以及并行计算的软件支持——并行程序的设计原理与方法。本书强调融并行机结构、并行算法和并行编程为一体,着重讨论并行算法的设计方法和并行数值算法,力图反映本学科的最新成就和发展趋势。

全书共十五章,分为四篇:第一篇为并行计算硬件基础,包括并行计算机系统及其结构模型 (第一章),当代并行机系统 SMP、MPP 和 COW (第二章),并行计算性能评测 (第三章);第二篇为并行算法的设计,包括并行算法的设计基础 (第四章),并行算法的一般设计方法 (第五章),并行算法的基本设计技术 (第六章),并行算法的一般设计过程 (第七章);第三篇为并行数值算法,包括基本通信操作 (第八章),稠密矩阵运算 (第九章),线性方程组的求解 (第十章),快速傅里叶变换 (第十一章);第四篇为并行程序设计,包括并行程序设计基础 (第十二章),并行程序设计模型和共享存储系统编程 (第十三章),分布存储系统并行编程 (第十四章),并行程序设计环境与工具 (第十五章)。书中取材新颖,内容丰富,体系完整,基本上包括了并行计算学科中的主要研究内容和主要研究方面。

使用方法 国家教育部计算机科学与技术教学指导委员会,将并行计算课程定位在高等学校计算机及相关专业本科高年级学生和研究生层次上。学生应在学习过计算机体系

结构、操作系统、编译原理以及最好学习过算法设计与分析(至少应学习过数据结构和图论)等课程之后学习本课程。全书内容可根据不同的教学对象进行不同的组合。但根据本课程的定位,作为必须讲授和最低60学时的教学要求,建议讲授章节和相应的学时分配如下:

章 名	建议讲授节次	建议学时
第一章	1.1.1,1.1.2,1.2.2,1.2.3,1.2.4,1.3.1,1.3.2	4
第二章	2.1.1,2.2.1,2.3.1,2.4.1,2.4.2	4
第三章	3.1,3.2	4
第四章	4.2.1～4.2.6	4
第五章	5.1.1,5.2.1,5.2.3,5.3.2	2
第六章	6.1,6.2,6.5	4
第七章	7.1,7.2,7.4.1,7.5	2
第八章	8.1,8.3,8.4	2
第九章	9.1,9.3,9.4	4
第十章	10.1,10.2,10.3,10.4.2～10.4.4,10.4.6	6
第十一章	11.1.2～11.1.4,11.3.1,11.3.2,11.3.4	4
第十二章	12.1.1,12.1.3,12.3,12.5.3	2
第十三章	13.1.2,13.2,13.3.2	4
第十四章	14.2,14.5	6
第十五章	15.2,15.3.1,15.3.2	4

书中其余部分的内容可由任教老师任选,而带＊号的章节是建议阅读的,它们或是预备性知识(希望不熟悉的读者课前预习),或是深入研究性内容(鼓励面向研究的读者深入阅读)。每章之末均有小结和导读(指导读者进一步追踪阅读),并附有适量的、密切结合课文的以及拓宽讲授内容的综合性习题。

为了配合讲授内容,有条件的学校应开设SMP平台、MPP平台和COW平台上的实验课程,每种平台至少安排2～3个小型综合练习程序,并提倡利用Internet浏览本书末提供的有关Web网址中的内容。确有困难的学校,应至少安排COW环境下的分布计算练习程序。

为了增强本书的好用性,撰写时尽量分点叙述、纲目清晰。全书除了提供必要的参考文献外,还开列了并行和分布计算Web网址、算法清单、示范程序清单以及术语中－英对照及索引。尽管后者在使用上尚不如英文索引来得方便,但毕竟在一定程度上方便了读者的查阅。

相关读物　本书在撰写时力图自成完备系统,但由于本书内容涉及面较广,所以读者

参阅必要的相关教材是有益处的。特别是,在可扩放并行计算方面,建议配合阅读 Kai Hwang 和 Zhiwei Xu 的新著 Scalable Parallel Computing:Technology, Architecture, Programming(McGraw-Hill,1998);在并行算法方面,建议配合阅读陈国良编著的《并行算法的设计与分析》(高等教育出版社,1994);在设计和构造并行程序方面,建议配合阅读 I. Foster 的著作 Designing and Building Parallel Programs:Concepts and Tools for Parallel Software Engineering (Addison-Wesley,1995);在数值并行算法方面,建议配合阅读 M.J. Quinn 的著作 Parallel Computing:Theory and Practice (McGraw-Hill,1994)和 V.Kumar 等的著作 Introduction to Parallel Computing:Design and Analysis of Algorithms(Benjamin/Cummings,1994)。

感谢 本书在撰写中,曾直接或间接地引用了许多专家、学者的文献,特别是对我国计算机学科发展卓有贡献的世界著名计算机结构学家黄铠教授,他非常关心国内计算机教育事业,及时向我们提供了他和徐志伟教授的著名新作 Scalable Parallel Computing,从而丰富了本书的内容,作者尤为感谢。书稿付梓前承蒙清华大学王鼎兴教授抱病进行了审校,作者倍加感谢。他的严谨治学作风和顽强拼搏精神激励和鞭策作者努力把书稿写好、改好。

中国科学技术大学计算机系纪金龙老师、安虹和计永昶博士为本书的第十四章和第十五章的写作提供了丰富的素材,并帮助我完成第四篇的教学工作,作者甚为感谢。

中国科学技术大学计算机系的历届学生们,在听取我的讲授中,曾提出过很多可贵意见,不断充实和完善了书稿的内容,特别是张青山、孙伟、严宝拾、吴明桥、侯海龙和陈志辉等同学完成了本书的计算机绘图工作。对于他们的辛勤劳动和良好的愿望,作者深感欣慰和谢意。

最后,感谢国家高性能计算中心(合肥)和中国科学技术大学计算机系为本书的写作提供了良好的条件。

国家教育部将《并行计算》一书审定为"面向21世纪课程教材",作者感到荣幸。但并行计算涉及学科很多,内容十分广泛,加上作者学识有限,写作时间仓促,书中错误和片面之处在所难免,恳请读者不吝批评指正。

陈国良
中国科学技术大学计算机系
国家高性能计算中心(合肥)
1999年5月

目　录

第一篇　并行计算硬件基础

第二篇 并行算法的设计

第三篇 并行数值算法

第四篇 并行程序设计

第一篇　并行计算硬件基础

第一章　并行计算机系统及其结构模型
第二章　当代并行计算机系统介绍
第三章　并行计算性能评测

这一篇主要研究并行计算的硬件基础,包括并行计算机系统及其结构模型(第一章),当代并行计算机系统(第二章)以及并行计算的性能评测(第三章)。但它与高级计算机体系结构课程不一样,本篇只是从并行计算的角度,介绍一些与并行计算直接相关的并行计算机系统方面的知识,欲深入学习并行计算机体系结构的读者可参阅丛书之三《并行计算机体系结构》[193]。

本篇的第一章,首先从当代科学与工程问题的计算需求出发,简单介绍并行计算与计算科学;接着讨论并行系统互连方式与技术;最后落实到并行计算机的结构模型与存储模型的介绍上,希望这些抽象的模型能够成为计算工作者与工程师的界面。第二章,将从并行机中抽出当代流行的一类可扩放的并行机加以简单介绍,它们包括共享存储的多处理机系统和分布存储的多计算机系统以及机群系统,这些都是并行计算工作者的最基本的硬件环境。第三章,将讨论并行计算性能评测问题,包括并行机基本性能评测、衡量并行计算性能的加速比、可扩放性以及基准测试程序,它们都是计算工作者、计算机厂商和计算机用户共同感兴趣的问题,而且能被一致认可和普遍接受。

第一章　并行计算机系统及其结构模型

简单地讲，并行计算就是在并行计算机或分布式计算机等高性能计算系统上所作的超级计算，其物质基础是高性能并行计算机（包括分布式网络计算机）。本章首先从当代科学与工程问题的计算需求出发，先简单地讨论并行计算与计算科学；然后讨论并行计算机系统的互连，包括静态互连网络、动态互连网络和标准互连网络；最后讨论并行计算机的系统结构模型（PVP、SMP、MPP、DSM 和 COW）、并行计算机的存储访问模型（UMA、NUMA、COMA、CC-NUMA 和 NORMA）以及并行计算机的存储组织（层次存储技术和高速缓存一致性问题）。

1.1 并行计算

1.1.1 并行计算与计算科学

并行计算(Parallel Computing),简单地讲,就是在并行计算机上所作的计算,它和常说的高性能计算(High Performance Computing)、超级计算(Super Computing)是同义词,因为任何高性能计算和超级计算总离不开使用并行技术。随着计算机和计算方法的飞速发展,几乎所有的学科都走向定量化和精确化,从而产生了一系列诸如计算物理、计算化学、计算生物学、计算地质学、计算气象学和计算材料科学等的计算科学,在世界上逐渐形成了一门计算性的学科分支,即计算科学与工程,简称为 CSE(Computational Science & Engineering)。当今,计算科学已经和传统的两种科学,即理论科学和实验科学,并列成为第三门科学,它们彼此相辅相成地推动科学发展与社会进步。在许多情况下,或者是理论模型复杂甚至理论尚未建立,或者实验费用昂贵甚至实验无法进行,此时计算就成为求解问题的惟一或主要手段。计算极大地增强了人们从事科学研究的能力,大大地加速了把科技转化为生产力的过程,深刻地改变着人类认识世界和改造世界的方法和途径。计算科学的理论和方法,作为新的研究手段和新的设计与制造技术的理论基础,正推动着当代科学与技术向纵深发展。

计算科学涉及到大型科学工程计算,是一个多学科的交叉领域,往往需要数学家、工程师和计算机科学家进行跨学科和跨行业的协同研究。一方面,它需要运用许多基础数学理论(如非线性分析、现代偏微分方程理论、微分几何和近世代数等);另一方面又需要熟悉某一特定应用领域的背景知识;最后还需要充分掌握和运用先进的计算设备。所以今后的科学与工程计算工作者应尽可能兼备数学、物理、工程科学和计算机科学等多方面的知识,并善于应用超级计算机进行大规模数值试验与分析。

1.1.2 当代科学与工程问题的计算需求

应用需求 人类对计算机性能的要求是无止境的,在诸如预测模型的构造和模拟、工程设计和自动化、能源勘探、医学、军事以及基础理论研究等领域中都对计算提出了极高的具有挑战性的要求。例如,在作数值气象预报时,要提高全球气象

预报的准确性，据估计在经度、纬度和大气层方向上至少要取 200×100×20＝40 万个网格点。目前中期天气预报有的模式需要 635 万个点，内存需要几十千兆字节 (1GB＝10^9B)，总运算量达 25 万亿次 (T＝10^{12})，并要求在不到 2 小时内完成 48 小时的天气预报。当计算能力不足时，只好降低结果的分辨率，简化计算方案，从而就影响了预报的准确度。又如，在进行油田整体“油藏模拟”时，假定一个油田有上万口井，每口井模拟时至少要取 8×8×50 个点，则总的变量个数可高达千万量级，使得现今的一般计算机难以实现。其它的应用领域包括核武器数值模拟，航空航天高速飞行器的设计，原子物理过程微观世界的模拟，材料科学中计算，环境资源以及生物计算等。这些重大的计算问题，涉及到非规则的复杂结构、非均匀的复合材料、非线性的动力学系统以及奇性区域、活动边界、带约束条件等各种复杂的数学物理问题。要对这些复杂的非线性数学物理方程进行大规模和高精度的计算，在一般的计算机上用传统的计算方法往往是无能为力的。

对高速并行计算的需求是广泛的，但归纳起来主要有三种类型的应用需求：① 计算密集 (Compute-Intensive) 型应用，如大型科学工程计算与数值模拟；② 数据密集 (Data-Intensive) 型应用，如数字图书馆、数据仓库、数据开采和计算可视化等；③ 网络密集 (Network-Intensive) 型应用，如协同工作、遥控和远程医疗诊断等。

从另一方面讲，正是这些重大的应用需求推动了当代计算技术的迅速发展。这也可从评测计算机性能的单位量词证实之：20 世纪 70 年代到 80 年代，常用 Mflops (每秒百万次浮点运算) 作为评测计算机性能的指标；到 20 世纪 80 年代中期又增用 Gflops (每秒 10 亿次浮点运算) 作为评测计算机性能的指标；近年来由于大规模并行机的问世，G (Giga＝10^9) 亦嫌太小，又出现了采用 Tflops (每秒万亿次浮点运算) 作为评测计算机性能的指标　　ps (每秒千万亿次浮点运算) 的计算机的预研工作正在进行。这种计算机速度单位量词的演变，从　　ga＝10^6) 到 G (Giga＝10^9) 到 T (Tera＝10^{12}) 一直到 P (Peta＝10^{15})，反映了计算机本身速度的惊人的改变，而其背后的驱动力就是那些挑战性的应用需求 (有关计算机中常用单位量词见本章表 1.8)。

美国 HPCC 计划　科学和工程计算中的重大挑战性课题，要求能提供 1 Tflops 计算能力、1 TB 主存容量和 1 TB/s 的 I/O 带宽，这就是所谓 3T 性能目标。为了保持在高性能计算与计算机通信领域中的世界领先地位，1993 年美国科学、工程、技术联邦协调理事会向国会提交了题为“重大挑战项目：高性能计算和通信 (High Performance Computing and Communication)”的报告，简称为 HPCC 计划，即美国总统科学战略项目，这些重大挑战性课题的应用领域汇总于表 1.1 中。

表 1.1　美国 HPCC 计划公布的重大挑战性应用课题一览表

应用领域	计算任务和预期结果
磁记录技术	研究静磁和交互感应以降低高密度磁盘的噪音
新药设计	通过抑制人的免疫故障病毒蛋白酶的作用来研制治疗癌症和艾滋病的药物
高速民航	用计算流体动力学来研制超音速喷气发动机
催化作用	仿生催化剂计算机建模,分析合成过程中的酶作用
燃料燃烧	通过化学动力学计算,揭示流体力学的作用,设计新型发动机
海洋建模	对海洋活动与大气流的热交换进行整体海洋模拟
臭氧耗损	研究控制臭氧消耗过程的化学和动力学机制
数字解析	用计算机研究实时临床成像、计算层析术、磁共振成像
大气污染	对大气质量模型进行模拟研究,控制污染的传播,揭示其物理与化学机理
蛋白质结构设计	使用计算机模拟,对蛋白质组成的三维结构进行研究
图像理解	实时绘制图像或动画
密码破译	破译由长位数组成的密码,求找该数的两个乘积因子①

HPCC 计划所提出的某些重大挑战性课题的计算需求如图 1.1 所示。它列出了支持科学模拟、先进计算机辅助设计和大型数据库与信息检索操作的实时处理等所需要的处理速度和存储器容量的量级。

在 HPCC 计划提出的当时,性能最好的计算机与 3T 要求相比,速度慢 100～1000 倍,而存储容量太小,I/O 带宽太窄。世界上第一台峰值速度超过 1 Tflopos 的高性能计算机是由 Intel 公司于 1996 年 12 月研制成功的。

美国 ASCI 计划　全面禁止核试验条约签订后,核武器的研究代之以实验室数值模拟。因此禁试后数值模拟成了惟一可能进行的全系统(虚拟)试验。这样,1996 年 6 月由美国能源部联合美国三大核武器实验室(Lawrence Livermore 国家实验室、Los Alamos 国家实验室和 Sandia 国家实验室)共同提出了"加速战略计算创新"(Accelerated Strategic Computing Initiative,简称为 ASCI)项目计划。提出通过数值模拟,评估核武器的性能、安全性、可靠性、更新等。要求数值模拟达到高分辨率、高逼真度、三维、全物理、全系统的规模和能力。该计划被认为是与当年曼哈

① 1994 年 4 月 26 日,美国宣布破译了世界上最长的 RSA129 密码。在因特网上,使用 1 600 台计算机,600 多人工作 8 个月,成功地破译了这个由 129 位数字组成的密码。1996 年 4 月,一个 130 位的数也被成功分解了。

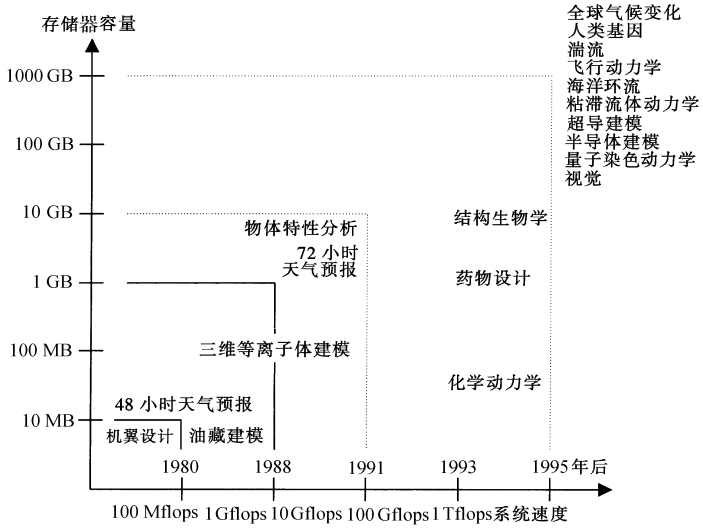


图 1.1 HPCC 方面的重大挑战性课题的需求

顿计划等同的一个巨大的挑战,不仅需要科学界的参与,也需要计算机工业界合作,提供保障 ASCI 应用所需的计算机平台。为此,三大核武器实验室分别向美国三大公司(Intel、IBM 和 SGI/CRAY)预定了峰值速度超过 1 Tflops 的并行计算机。美国能源部计划在 2003 年要使用运算速度为 100 Tflops、内存容量为 50TB 的并行机(见表 1.2)。目前 ASCI 计划正把各项应用需求与计算平台推进到万亿级规模的体系中去(即每秒可执行万亿次浮点运算,万亿个字节的 RAM,数十万亿个字节的磁盘,千万亿个字节的档案存储器,数百亿个字节的网络带宽)。

表 1.2 美国 ASCI 计划中的并行机一览表

时间(年)	运算速度(OPS)	存储容量(TB)	海存容量(PB)	I/O 传输率(GB/s)	网络传输率(GB/s)
1996	10^{11}	0.05	0.13	5	0.13
1997	10^{12}	0.5	1.3	50	1.3
2000	10^{13}	5	13	500	13
2003	10^{14}	50	130	5 000	130

1.2　并行计算机系统互连

1.2.1　系统互连

在多处理机、多计算机或分布式系统中，不同组成部分（CPU、存储模块、I/O设备、网络接口等）都要通过互连网络彼此连接起来。图1.2示出了当今不同网络拓扑和互连技术的关系图。其中，水平轴自左至右网络距离逐渐增加，垂直轴代表单位时间网络可传输的最大信息量。参照图1.3，一个系统域网络SAN（System Area Network）可以将短距离（3～25 m）内的不同节点连接起来形成单一系统；该系统可用于一个建筑物、校园或企业（500 m～2 km）内的局域网络LAN（Local Area Network）相连构成一个完整系统。在每个节点内，处理器芯片管脚形成了处理器总线；局部（本地）总线，存储器总线，将处理器与存储模块相连；I/O总线，系统总线，将I/O设备、网卡等连接起来（I/O总线有时也叫作小型机系统接口SCSI（Small Computer System Interface）总线）。一个都域网MAN（Metropolitan Area Network）可以覆盖整个城市（≥25 km）；而一个广域网WAN（Wide Area Network）甚至可覆盖全球，但本书不讨论它们，只讨论到LAN这一级，因为在构成网络工

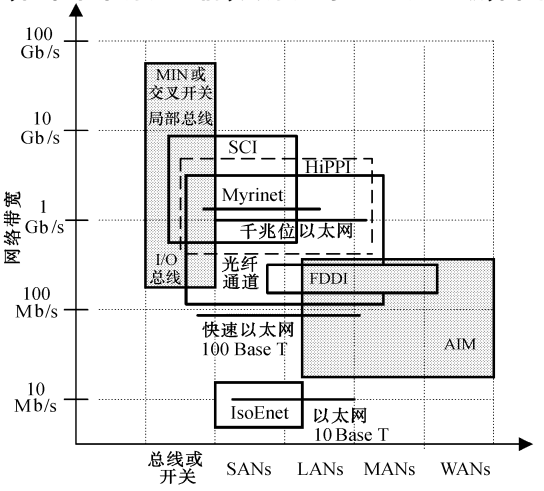


图1.2　系统互连和网络拓扑

作站机群 COW (Cluster of Workstations) 时会用到 LAN 技术。下面将着重讨论节点内处理器、存储器和 I/O 设备之间的互连,包括静态互连网络和动态互连网络;而对构成高端并行计算机所需的 SAN/LAN 网络技术只作简要介绍。

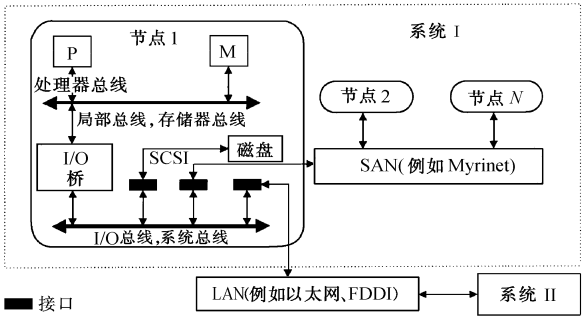


图 1.3 局部总线、I/O 总线、SAN 和 LAN

1.2.2 静态互连网络

所谓静态网络 (Static Networks) 是指处理单元间有着固定连接的一类网络,在程序执行期间,这种点到点的链接保持不变;相反,动态网络 (Dynamic Networks) 是用开关单元构成的,可按应用程序的要求动态地改变连接组态。典型的静态网络有一维线性阵列、二维网孔、树连接、超立方网络、立方环、洗牌交换网、蝶形网络等;典型的动态网络包括总线、交叉开关和多级互连网络等。为讨论方便,兹作如下定义:

定义 1.1 射入或射出一个节点的边数称为节点度 (Node Degree)。在单向网络中,入射和出射边之和称为节点度。

定义 1.2 网络中任何两个节点之间的最长距离,即最大路径数,称为网络直径 (Network Diameter)。

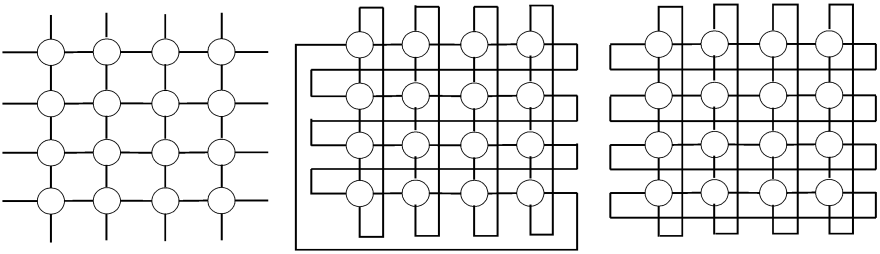
定义 1.3 对分网络各半所必须移去的最少边数称为对剖宽度 (Bisection Width)。

定义 1.4 如果从任一节点观看网络都一样,则称网络为对称的 (Symmetric)。

一维线性阵列 (1-D Linear Array) 它是并行机中最简单、最基本的互连方式,其中每个节点只与其左、右近邻相连,故也叫二近邻连接。N 个节点用 $N-1$ 条边串接之,内节点度为 2,直径为 $N-1$,对剖宽度为 1。当首、尾节点相连时可构成循环移位器,在拓扑结构上等同于环,环可以是单向的或双向的,其节点度恒

为 2,直径或为 $\lfloor N/2 \rfloor$(双向环)或为 $N-1$(单向环),对剖宽度为 2。

二维网孔(2-D Mesh) 在一个 $N \times N$ 的二维网孔网络中,每个节点只与其上、下、左、右的近邻相连(边界节点除外),故也称四近邻连接,因而节点度为 4,网络直径为 $2(N-1)$,对剖宽度为 N(见图 1.4(a))。如果在垂直方向上带环绕,而水平方向呈蛇状,则 2-D 网孔就变成 Illiac 网孔了(见图 1.4(b)),此时节点度恒为 4,网络直径为 $N-1$,而对剖宽度为 $2N$。如果 2-D 网孔的垂直和水平方向均带环绕,则它就变成了二维环绕(2-D Torus)(见图 1.4(c)),其节点度恒为 4,而网络直径为 $2(\lfloor N/2 \rfloor)$,对剖宽度为 $2N$。



(a) 2-D 网孔 (b) Illiac 网孔 (c) 2-D 环绕

图 1.4 四近邻连接

树 二叉树除了根节点和叶节点之外,每个内节点只与其父节点和两个子节点相连,故也称为三近邻连接。如图 1.5(a)所示,显然节点度为 3,对剖宽度为 1,而树的直径为 $2(\lfloor \log N \rfloor - 1)$($N$ 为树的总节点数)。X-树(X-Tree),将同级的兄弟节点彼此相连。如果尽量增大节点度为 $N-1$,则直径缩小为 2,此时就变成了如图 1.5(b)所示的星形网络了,其对剖宽度为 $\lfloor N/2 \rfloor$。

传统二叉树的主要问题是根易成为通信瓶颈。1985 年 Leiserson[119] 提出的胖树(Fat Tree)可缓解此问题。如图 1.5(c)所示,胖树节点间的通路自叶向根逐渐变宽,它更像真实的树,连向根部的枝叉变得愈来愈粗。

超立方 一个 n-立方由 $N=2^n$ 个顶点组成。3-立方如图 1.6(a)所示;4-立方如图 1.6(b)所示,由两个 3-立方的对应顶点连接而成。n-立方的节点度为 n,网络直径也是 n,而对剖宽度为 $N/2$。由于该网络缺乏可扩放性和不易构成多维超立方,所以它正逐渐被其它的网络所代替。但过去在超立方上开发了很多优秀的算法,而像二叉树、网孔和很多其它低维网络均能嵌入超立方中,所以超立方具有学术研究的意义。如果将 3-立方的每个顶点代之以一个环就构成了如图 1.6(c)所示的 3-立方环(3-Cube Connected Cycle),此时每个顶点的度为 3,而不像超立方那样节点度为 n。一般而言,可以从一个 k-立方构成一个具有 $n=2^k$

个带环顶点(每个顶点是 k 个连成环的节点)的 k 立方环,此时 k 立方环中总共有 $N=k\cdot 2^k$ 个节点($k\geq 3$),网络直径为 $2k-1+\lfloor k/2\rfloor$,而对剖宽度为 $N/2k$。

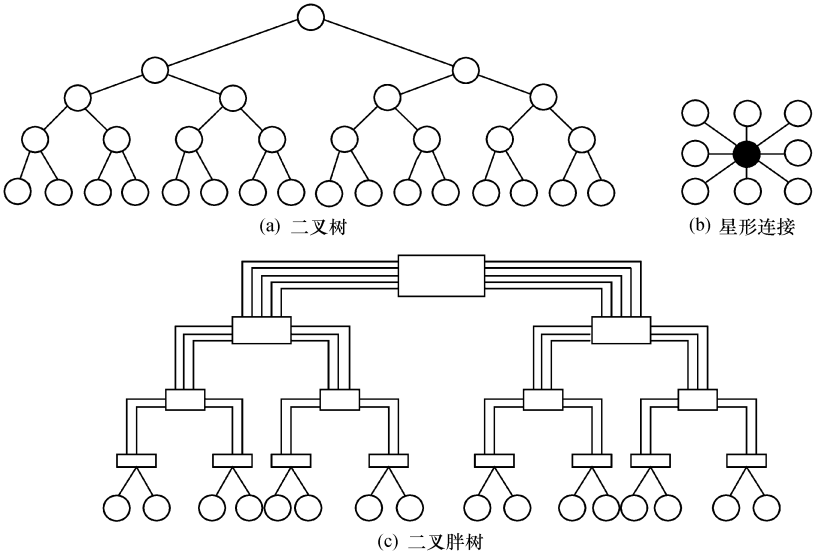
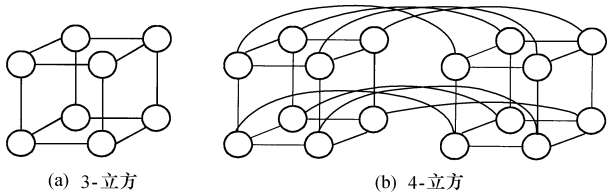


图 1.5 树形连接



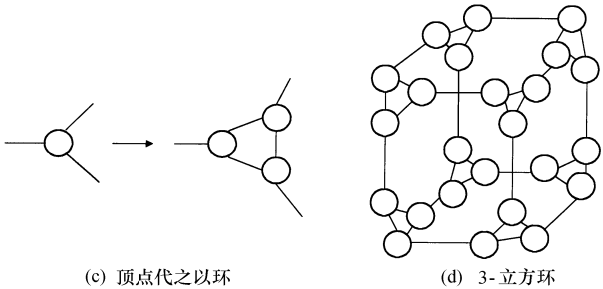
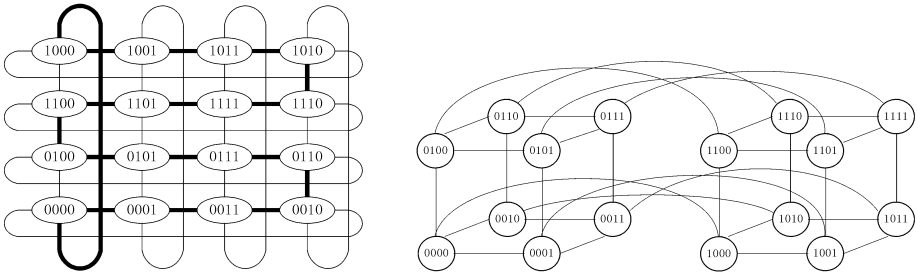


图 1.6 超立方

嵌入(Embedding)　该术语系指将网络中的各节点映射到另一个网络中去。用膨胀(Dilation)系数来描述嵌入的质量,它是指被嵌入网络中的一条链路在所要嵌入的网络中对应所需的最大链路数。如果该系数为1,则称为完美嵌入。例如,一个环网可完美嵌入到2－D环绕网中(如图1.7(a)中的粗线所示)。同样,一个超立方网也可以完美嵌入到2－D环绕网中,注意此时可将2－D环绕网中的每个节点的 X, Y 坐标用葛莱(Gray)码标记之(如图1.7(b)所示)。由于该编码中,相邻两码字仅有一位不同,这正好与超立方中每个节点与其相邻的节点的二进制编码恰恰仅一位不同相一致(如图1.7(b)所示)。



(a) 2-D 环绕网中的葛莱编码　　(b) 超立方中节点的编码

图1.7　环网和超立方嵌入2－D环绕网中

并非所有网络之间均可实现完美嵌入。例如,一棵完全二叉树,除非其高度为2,否则无法将其完美嵌入到2－D网孔中。图1.8示出了一棵完全二叉树嵌入到2－D网孔中的所谓"H－树"嵌入法[190]。图中标有A的节点是"H－树"嵌入所需的附加节点,带阴影的节点为树的节点。注意从根节点到树的下一层节点以及再下一层节点之间均需两条链路,所以其膨胀系数为2。当树更高时,该系数亦随之增大。一般而言,对于高度为 h 的完全二叉树,其膨胀系数为 $\lceil h/2 \rceil$。

静态互连网络小结　表1.3汇总了静态互连网络的重要特性。大多数网络的节点度都是个小的常数,这是比较理想的;随着选路技术的革新(例如虫蚀选路),网络的直径变得不那么重要了;对剖宽度会影响网络的带宽;网络的对称性与可扩放性和选路效率有关(注意表中对数以2为底)。

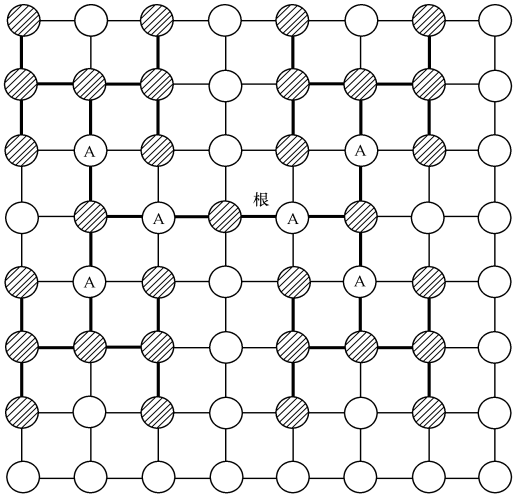


图 1.8 将二叉树嵌入到 2－D 网孔中

表 1.3 静态互连网络特性一览表

网络名称	网络规模	节点度	网络直径	对剖宽度	对称	链路数
线性阵列	N个节点	2	$N-1$	1	非	$N-1$
环形	N个节点	2	$\lfloor N/2 \rfloor$ (双向)	2	是	N
2－D 网孔	($N\times N$) 个节点	4	$2(\sqrt{N}-1)$	$\sqrt{N}$	非	$2(N-\sqrt{N})$
Illiac 网孔	($N\times N$) 个节点	4	$N-1$	$2\sqrt{N}$	非	$2N$
2－D 环绕	($N\times N$) 个节点	4	$2\lfloor \sqrt{N}/2 \rfloor$	$2\sqrt{N}$	是	$2N$
二叉树	N个节点	3	$2(\lceil \log_2 N \rceil - 1)$	1	非	$N-1$
星形	N个节点	$N-1$	2	$\lfloor N/2 \rfloor$	非	$N-1$
超立方	$N=2^n$ 个节点	n	n	$N/2$	是	$nN/2$
立方环	$N=k\cdot 2^k$ 个节点	3	$2k-1+\lfloor k/2 \rfloor$	$N/(2k)$	是	$3N/2$

1.2.3 动态互连网络

动态互连不是固定连接,而是在连接路径的交叉点处置以电子开关、选路器或

仲裁器等,以提供动态连接的特性。三类著名的动态互连网络是总线、交叉开关和多级互连网络。

总线　总线(Bus)实际上是连接处理器、存储模块和I/O外围设备等的一组导线和插座。总线系统用以主设备(如处理器)和从设备(如存储器)之间的数据传输。公用总线以分时工作为基础,在多个请求情况下,总线的仲裁是重要的。

目前已有很多总线标准,例如PCI、VME、Multibus、SBus、Microchannel和IEEE Futurebus。绝大多数标准总线都可低价构造单一处理系统(Uni-Processor System)。在构造多处理器系统时,常使用多总线和层状总线。

图1.9所示是个典型的多层总线系统。包括板级总线、底板级总线和I/O级总线。在印刷电路板上实现的总线叫局部(或本地)总线(Local Bus),习惯上CPU板级上的总线叫本地总线,存储器板级上的总线叫存储器总线,I/O板级和通信板级上的总线叫数据总线。系统总线是在底板上实现的,它为所有插入板之间的通信提供了通路。在各插入板中均设有专用逻辑接口(IF)和专用控制器,包括I/O控制器(IOC)、存储器控制器(MC)和通信控制器(CC)。I/O设备是通过I/O总线(如SCSI)与计算机系统连接的。

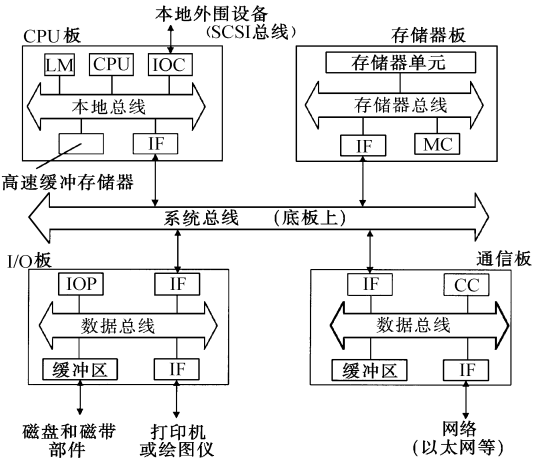


图1.9　板级、底板级和I/O级总线系统

设计多处理器总线系统的主要问题包括总线仲裁、中断处理、协议转换、快速同步、缓存一致性协议、总线桥接和层次总线的扩展等。通常,监听协议(Snoopy Protocol)均内置在多处理器总线中;硬件路障同步也常使用在多处理器总线中。

交叉开关　交叉开关(Crossbar Switcher)是一种高带宽网络。像电话交换机

一样,交叉点开关能在(源,目的)对之间建立动态连接。每个交叉点开关为一个(源,目的)对提供一条专用连接通路,其开关状态可根据程序的要求动态地置为"开"或"关"。图1.10是一个4×4的交叉开关,其中每个输入口有一接收器和输入缓冲器,以处理到达的请求;每个输出口有一个发送器,以传递输出数据信号至一条通信链上;开关控制仲裁开关的"开/关"状态。

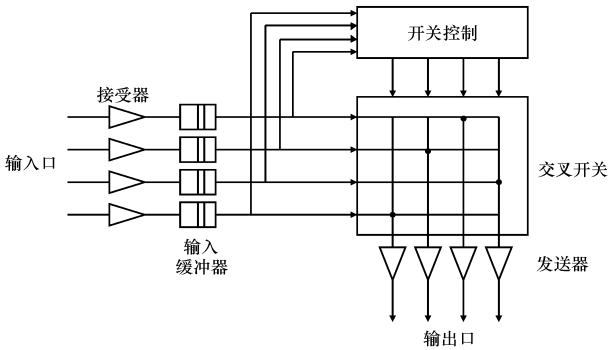


图1.10 4×4的交叉开关

在并行处理中,交叉开关的使用通常有两种方式:一种是用于处理器之间的通信,此时每一行和每一列只能接通一个交叉点开关,所以一个 $n\times n$ 的交叉开关网络一次最多可接通 n 个对;另一种是用于处理器和存储模块之间的通信,此时每个存储模块一次只能由一个处理器请求访问,所以每一列中只能接通一个交叉点开关,但为了支持并行(或交叉)存储访问,每一行可同时接通多个交叉点开关。

一般而言,交叉开关的成本为 n^2,其中 n 为端口数,这样就限制了它在大型系统中的应用。

多级互连网络 为了构筑大型开关网络,常可将单级交叉开关级联起来形成一个多级互连网络MIN(Multistage Interconnection Network),它已被用在MIMD和SIMD并行机中。一种通用多级互连网络如图1.11所示,其中每一级都用了多个 $a\times b$ 开关单元,相邻各级之间有着固定的级联拓扑。为了在输入和输出之间建立所需的连接,可以动态设置开关状态。

目前已经提出了很多MIN,其差别就在于开关单元及其级间连接(ISC)方式不同。最简单的开关单元是2×2的开关单元;常用的ISC模式有均匀洗牌、蝶式、纵横交叉等。图1.12(a)就是著名的Ω网络(Ω Network),其中2×2的开关单元可有如图1.12(a)、(b)、(c)、(d)四种连接方式,而ISC是均匀洗牌连接模式。

对于一个 n 输入和 n 输出的Ω网络,共有 $\log n$ 级,每级包括 $n/2$ 个开关单

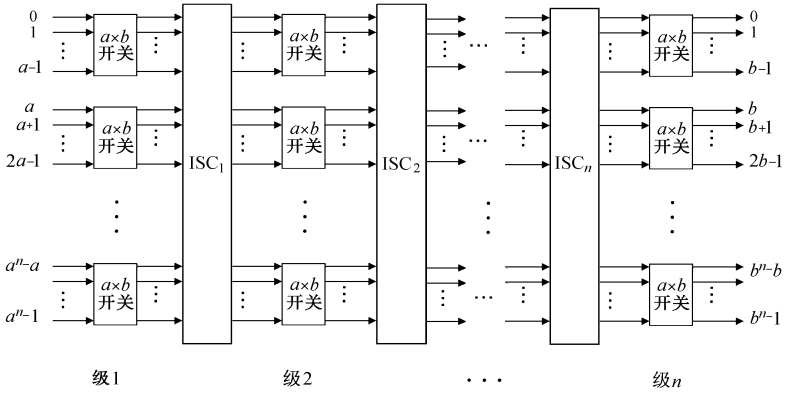
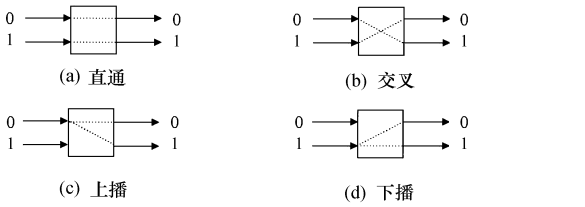


图 1.11　通用多级互连网络结构



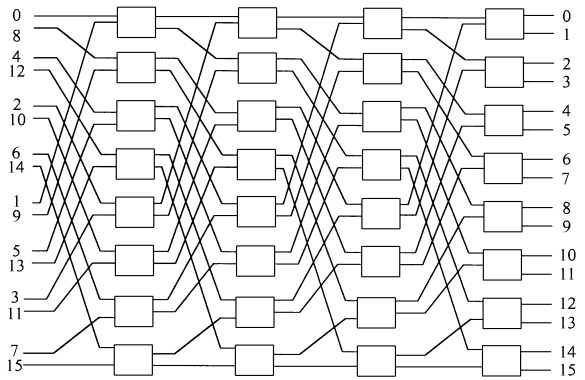


图 1.12　Ω 网络

元,所以共有$\frac{n}{2}\log n$个开关单元。其它的MIN,还有基准网络、二进制立方网络和Benes网络等。

动态互连网络小结 表1.4汇总了动态互连网络的重要特性。显然,总线造价最低,但易冲突;交叉开关造价最高,但带宽和选路性能最好;多级互连网络是总线与交叉开关的折衷,主要优点采用模块结构,可扩展性好,但延迟随网络尺寸对数增长。

表1.4 动态互连网络特性一览表

网络特性	总线系统 n台处理器,总线宽度为ω位	多级互连网络 $n\times n$MIN采用$k\times k$开关,线宽为ω位	交叉开关 $n\times n$交叉开关线宽为ω位
最小时延	恒定(轻负载)	$O(n\log_k n)$	恒定
每台处理器带宽	$O(\omega/n)$至$O(\omega)$	$O(\omega)$至$O(n\omega)$	$O(\omega)$至$O(n\omega)$
开关复杂性	$O(n)$	$O(n\log_k n)$	$O(n^2)$
连线复杂性	$O(\omega)$	$O(n\omega\log_k n)$	$O(n^2\omega)$
连接特性	一次只能一对一	是阻塞型网络	全置换

1.2.4 标准互连网络

本节所介绍的高速互连技术,对构筑一个分布式网络计算环境是十分有用的,这些互连技术均有普遍被接受的协议标准。

FDDI 光纤分布式数据接口FDDI(Fiber Distributed Data Interface)采用双向光纤令牌环可提供100~200 Mb/s数据传输。彼此相反方向的双向环可提供冗余通路以提高可靠性。它能够连接大量的设备,其距离可达100 m(使用铜线)、2 km(使用多模式光纤)和60 km(使用单模式光纤),这使得有可能在LAN和MAN中应用FDDI环。FDDI集中器采用孤立故障的办法可使网络更可靠。

100 Mb/s主干FDDI网络如图1.13所示。其中专门的路由器将FDDI环连向以太集线器,后者可连接大量的桌面计算机。FDDI环可确保加入、删除、移动主机或设备而不会使网络崩溃。

FDDI的缺点是不能支持多媒体信息流,这就减弱了它与ATM技术的竞争力,但使用全双工模式的FDDI(即FFDT)可以增强FDDI的竞争力。

快速以太网 以太网已历经三代:1982年引入的10 Mb/s网属第一代以太

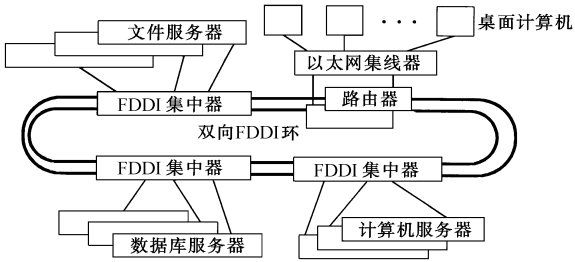


图 1.13　双向 FDDI 环作为主干网

网;1994 年宣布的 100 Mb/s 快速网属第二代以太网;1997 年 IEEE802.3 工作组宣布的 1 Gb/s 千兆位网可谓第三代以太网,其特性汇总于表 1.5 中。

表 1.5　三代以太网特性一览表

代别(类型)		以太网 10 BaseT	快速以太网 100 BaseT	千兆位以太网 1 Gb
引入年代		1982	1994	1997
速度(带宽)		10 Mb/s	100 Mb/s	1 Gb/s
最大距离	非屏蔽双扭对	100 m	100 m	25～100 m
	屏蔽双扭对/同轴电缆	500 m	100 m	25～100 m
	多模式光纤	2 km	412 m(半双工) 2 km(全双工)	500 m
	单模式光纤	25 km	20 km	2 km
主要应用领域		文件共享, 打印机共享等	COW 计算, C/S 结构, 大型数据库等	大型图像文件, 多媒体,因特网, 内部网,数据仓库等

图 1.14 示出了如何由开关快速以太主干网升级到千兆位主干 LAN。

Myrinet　Myrinet 是由 Myricom 公司生产的千兆位包开关网,其目的是互连商用产品以构筑计算机群,常用来构造机箱内 SAN 机群或基于 LAN 的桌面系统。Myrinet 定义在数据链路层,包的长度可变,在每条链路上施行流控和差错控制,

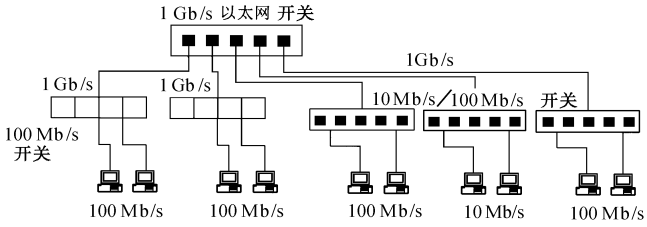


图 1.14 千兆位以太主干网的构造

使用切通选路法和定制的可编程主机接口。图 1.15 示出了用 4 个 Myrinet 开关构造 Myrinet LAN (连接工作站、PC) 桌面系统、机箱内多机机群和单板多处理器机群。

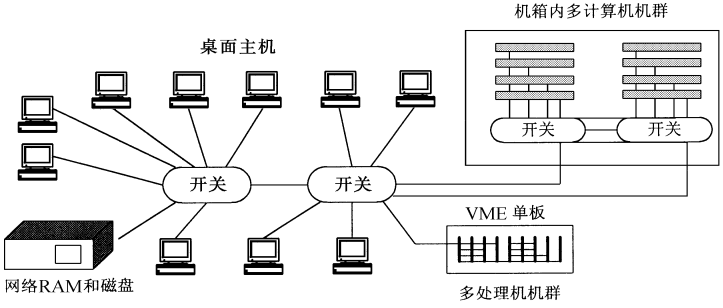


图 1.15 4 个 8 端口开关构筑的 Myrinet 群集

总之,Myrinet 支持机群计算具有很大的潜力。但与总线有关的主机界面仍限制大量不同的主机连向 Myrinet,其短于 25 m 的电缆 (使用铜线) 使 Myrinet 作为 SAN 使用受到了限制。

HiPPI 高性能并行接口 HiPPI(High Performance Parallel Interface) 已广泛用来构筑异构计算机系统,它是 Los Alamos 国家实验室 1987 年提出的一个标准,其目的是试图统一不同厂商的所有大型机和超级计算机的界面。HiPPI 曾被大型机和超级计算机工业界作为高速 I/O 通道用于短距离的系统到系统和系统到外设的连接。1993 年,ANSIX3T9.3 批准了 HiPPI 标准,它覆盖了物理和数据链路层,而高层的东西留给了用户。

HiPPI 是一个单工点到点的数据传输界面,其速度可达 800 Mb/s 到 1.6 Gb/s。随着工艺的进展和价格的降低,HiPPI 已不再只为超级计算机所用。

图 1.16 展示出用 HiPPI 交叉开关连接各种大型机、服务器和超级计算机的 LAN 主干网。

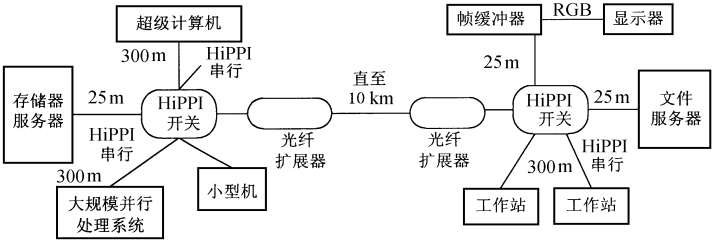


图 1.16　使用 HiPPI 通道和开关构筑的 LAN 主干网

ATM　异步传输模式 ATM (Asynchronous Transfer Mode) 是在光纤通信基础上建立起来的一种新的宽带综合业务数字网 (BISDN) 的交换技术。根据 CCITT1998 年发表的蓝皮书 I.121 建议,它将是建立一个统一通信网络的最终解决方案。ATM 是一种与介质无关的信息传输协议,采用 53 个字节的定长短数据单元 (信元,Cell) 进行传输。按此,如图 1.17 所示,一个长的包 (Packet) 经 ATM 网络传输之前须破成一些小的信元,在接收端再将它们重装起来。这些小的信元,可以使用逐段 (Hop－by－Hop) 法进行选路。ATM 网可以高速传输诸如声音、图像、视频和数据等所有形式的媒体;它可以加入到现存的 LAN、MAN 和 WAN 中。

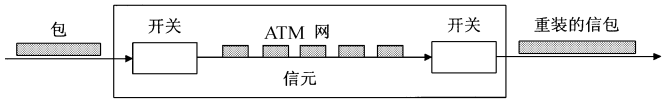


图 1.17　ATM 网中的信元传输

ATM 采用 4 层体系结构,它与开放系统连接 OSI (Open Systems Interconnection) 标准对应关系示于表 1.6 中。其中,CS (Convergence Sublayer) 是汇聚子层;SAR (Segmentation and Reassembly Sublayer) 是分段与组装子层;TC (Transmission Convergence Sublayer) 是传输汇聚子层;PMD (Physical Medium Dependent Sublayer) 是物理介质相关子层和 AAL (ATM Adaptation Layer) 是 ATM 适配层。

表 1.6 ATM 与 OSI 标准对应关系一览表

OSI 层	ATM 层	ATM 子层	基本功能
3/4	AAL	CS	提供汇聚标准界面
		SAR	分段和装配
2/3	ATM		流控,信元头的产生/消除,虚电路/路径的管理,信元的多路复用/分配
2	物理层	TC	信元速率匹配,信元的产生、打包/拆包,头检查和产生/验证
1		PMD	位定时,物理网络访问

如图 1.18 所示,ATM 网络的端点设备通过用户网络接口 UNI (User Network Interface) 连到 ATM 交换机上。一个网络可以有多个 ATM 交换设备,这些交换机连接在一起可形成更大的网络。ATM 交换设备之间的接口称为网络到网络接口 NNI (Network-Network Interface)。利用 ATM 技术可以构成宽带 LAN 主干网,其中交换机连接路由器和服务器,提供了可扩放和高带宽的交换能力,而高性能的服务器可直接与 ATM 交换机相连。也可以用 ATM 作为桌面网络技术组成高速用户机群局域网,然后通过路由器可以把 ATM 局域网与传统的主干网相连。ATM 作为 B-ISDN 的交换技术,还可提供基于 ATM 的 MAN/WAN 的体系结构。

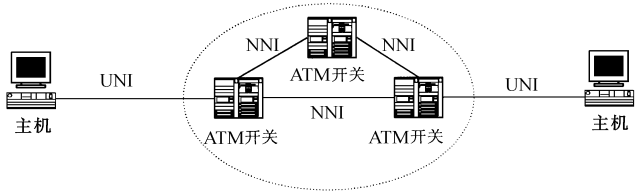


图 1.18 ATM 网络中的 UNI 和 NNI

图 1.19 所示是一个基于 ATM 的多计算机机群实例,它由香港大学开发,专门应用于分布式多媒体和元计算 (Metacomputing)。其中,HARNET 代表 Hong Kong Academic Research Net;IMSC 代表 Integrated Multimedia System Center;WS 代表 Workstation;PC 代表 Personal Computer。

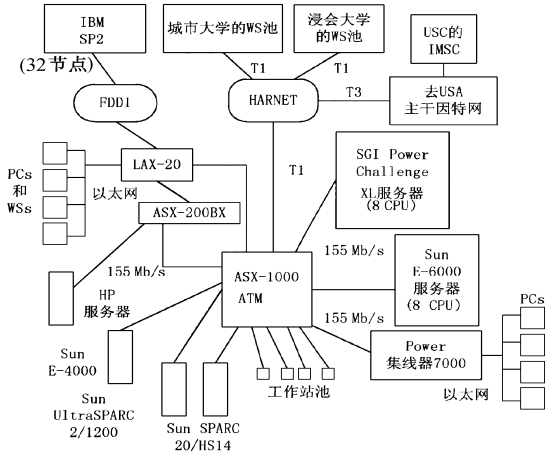


图 1.19　香港大学开发的珍珠机群

1.3　并行计算机系统结构

1.3.1　并行计算机结构模型

大型并行机系统一般可分为 6 类机器：单指令多数据流 SIMD (Single Instruction Multiple Data)；并行向量处理机 PVP (Parallel Vector Processor)；对称多处理机 SMP (Symmetric Multiprocessor)；大规模并行处理机 MPP (Massively Parallel Processor)；工作站机群 COW (Cluster of Workstations) 和分布共享存储 DSM (Distributed Shared Memory) 多处理机。SIMD 计算机多为专用，其余的 5 种均属于多指令多数据流 MIMD (Multiple Instruction Multiple Data) 计算机。5 种 MIMD 并行机的结构模型示于图 1.20。其中 B (Bridge) 是存储总线和 I/O 总线间的接口，DIR (Cache Directory) 是高速缓存目录，IOB (I/O Bus) 是 I/O 总线，LD (Local Disk) 是本地磁盘　　y Bus) Interface Circuitry) 是网络接口电路，P/C (Microprocessor and Cache) 是微处理器和高速缓存，SM (Shared Memory) 是共享存储器。目前绝大多数近代并行机均用商品硬件构成，而 PVP 计算机的部件很多都是定制 (Custom Made) 的。

PVP　典型的并行向量处理机的结构示于图 1.20　　y C-90、Cray

90、NEC SX-4和我国的银河1号等都是PVP。这样的系统中包含了少量的高性能专门设计定制的向量处理器VP,每个至少具有1 Gflops的处理能力。系统中使用了专门设计的高带宽的交叉开关网络将VP连向共享存储模块,存储器可以兆字节每秒的速度向处理器提供数据。这样的机器通常不使用高速缓存,而是使用大量的向量寄存器和指令缓冲器。

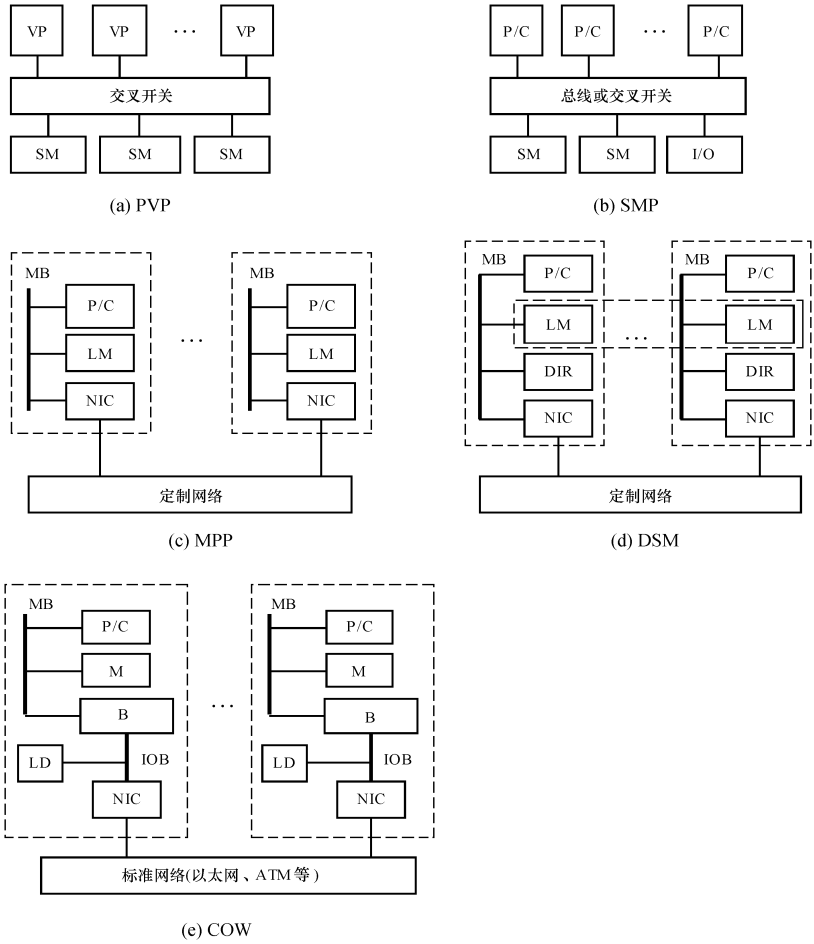


图1.20 5种并行机结构模型

SMP　对称多处理机的结构示于图 1.20 (b)。IBM R50、SGI Power Challenge、DEC Alpha 服务器 8400 和我国的曙光 1 号等都是这种类型的机器。SMP 系统使用商品微处理器 (具有片上或外置高速缓存),它们经由高速总线 (或交叉开关) 连向共享存储器。这种机器主要应用于商务,例如数据库、在线事务处理系统和数据仓库等。重要的是系统是对称的,每个处理器可等同的访问共享存储器、I/O 设备和操作系统服务。正是对称,才能开拓较高的并行度;也正是共享存储,限制系统中的处理器不能太多 (一般少于 64 个),同时总线和交叉开关互连一旦作成也难于扩展。

MPP　大规模并行处理机的结构示于图 1.20 (c)。Intel Paragon、Cray T3E、Intel Option Red 和我国的曙光 1000 等都是这种类型的机器。MPP 一般是指超大型 (Very Large Scale) 计算机系统,它具有如下特性:①处理节点采用商品微处理器;②系统中有物理上的分布式存储器;③采用高通信带宽和低延迟的互连网络 (专门设计和定制的);④能扩放至成百上千乃至上万个处理器;⑤它是一种异步的 MIMD 机器,程序系由多个进程组成,每个都有其私有地址空间,进程间采用传递消息相互作用。MPP 的主要应用是科学计算、工程模拟和信号处理等以计算为主的领域。

DSM　分布式共享存储多处理机的结构示于图 1.20 (d)。Stanford DASH、Cray T3D 和 SGI/Cray Origin2000 等属于此类结构。高速缓存目录 DIR 用以支持分布高速缓存的一致性。DSM 和 SMP 的主要差别是,DSM 在物理上有分布在各节点中的局存从而形成了一个共享的存储器。对用户而言,系统硬件和软件提供了一个单地址的编程空间。DSM 相对于 MPP 的优越性是编程较容易。

COW　工作站机群的结构示于图 1.20 (e)。Berkeley NOW、Alpha Farm、Digital Trucluster 等都是 COW 结构。在有些情况下,机群往往是低成本的变形的 MPP。COW 的重要界线和特征是:①COW 的每个节点都是一个完整的工作站 (不包括监视器、键盘、鼠标等),这样的节点有时叫作"无头工作站",一个节点也可以是一台 PC 或 SMP;②各节点通过一种低成本的商品 (标准) 网络 (如以太网、FDDI 和 ATM 开关等) 互连 (有的商用机群也使用定做的网络);③各节点内总是有本地磁盘,而 MPP 节点内却没有;④节点内的网络接口是松散耦合到 I/O 总线上的,而 MPP 内的网络接口是连到处理节点的存储总线上的,因而可谓是紧耦合式的;⑤一个完整的操作系统驻留在每个节点中,而 MPP 中通常只是个微核,COW 的操作系统是工作站 UNIX,加上一个附加的软件层以支持单一系统映像、并行度、通信和负载平衡等。

现今,MPP 和 COW 之间的界线越来越模糊。机群相对于 MPP 有性能/价格比高的优势,所以在发展可扩放并行计算机方面呼声很高。

公用结构 SMP、MPP、DSM 和 COW 等并行结构渐趋一致，DSM 是 SMP 和 MPP 的自然结合，MPP 和 COW 的界线逐渐不清，它们最终的结构趋向一致，形成当代并行机的公用结构（图 1.21）。在这样的系统结构中，大量的节点可通过高速网络互连起来。节点通常遵循着一个 Shell 结构（Shell Architecture），其中一个专门设计定制的电路（叫作 Shell）将商品微处理器和其余的节点，包括板级高速缓存 C、主存 M、NIC 和磁盘 D 连接起来。在一个节点内可有不止一个处理器。这种 Shell 结构的优点是，当处理器芯片更新换代时只需改变 Shell。图 1.21 中示出了三种不同的共享结构，其中将无共享结构图（a）中节点内的磁盘（D）移出来就形成了共享磁盘的结构图（b），再把主存（M）移出来就变成了共享存储结构图（c）。

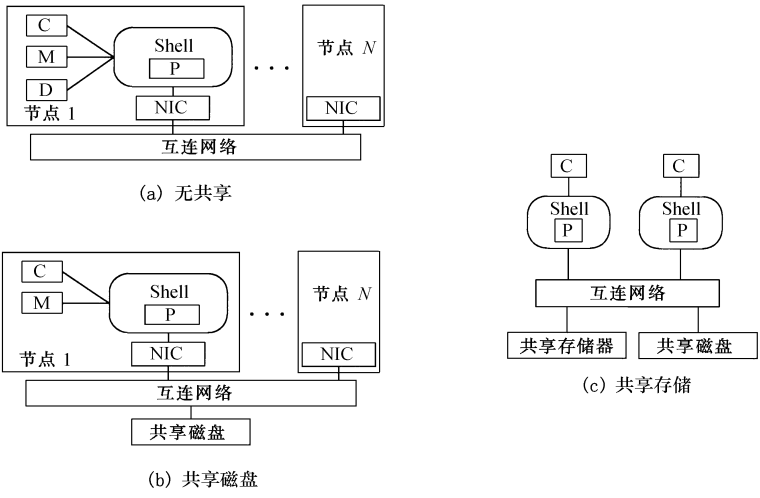


图 1.21 可扩放并行机公用结构

小结 表 1.7 汇总了上述 5 种结构的特性比较，其中有关访存模型解释见第 1.3.2 节。

表 1.7 5 种结构特性一览表

属性	PVP	SMP	MPP	DSM	COW
结构类型	MIMD	MIMD	MIMD	MIMD	MIMD
处理器类型	专用定制	商用	商用	商用	商用
互连网络	定制交叉开关	总线、交叉开关	定制网络	定制网络	商用网络（以太 ATM）

(续表)

属性	PVP	SMP	MPP	DSM	COW
通信机制	共享变量	共享变量	消息传递	共享变量	消息传递
地址空间	单地址空间	单地址空间	多地址空间	单地址空间	多地址空间
系统存储器	集中共享	集中共享	分布非共享	分布共享	分布非共享
访存模型	UMA	UMA	NORMA	NUMA	NORMA
代表机器	Cray C-90, Cray T-90, 银河1号	IBM R50, SGI Power Challenge, 曙光1号	Intel Paragon, IBM Option White 曙光-1000/2000	Stanford DASH, Cray T-3D	Berkeley NOW, Alpha Farm

1.3.2　并行计算机访存模型

下面从系统访问存储器的模式来讨论多处理机和多计算机系统的访存模型，它和上节所讨论的结构模型，是实际并行计算机系统的两个方面。

UMA　UMA (Uniform Memory Access) 模型是均匀存储访问模型的简称。图1.22示出了UMA多处理机模型，其特点是：①物理存储器被所有处理器均匀共享；②所有处理器访问任何存储单元取相同的时间 (此即均匀存储访问名称的由来)；③每台处理器可带私有高速缓存；④外围设备也可以一定形式共享。这种系统由于高度共享资源而称为紧耦合系统 (Tightly Coupled System)。当所有的处理器都能等同地访问所有I/O设备、能同样地运行执行程序 (如操作系统内核和I/O服务程序等) 时称为对称多处理机SMP (Symmetric Multiprocessor)。如果只有一台

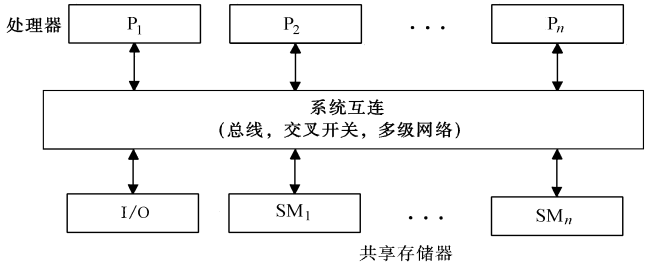
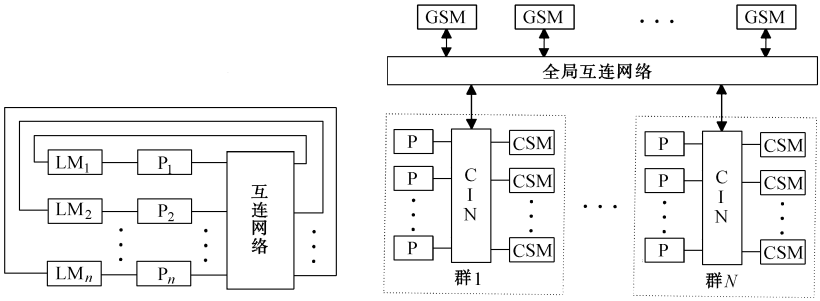


图1.22　UMA多处理机模型

或一组处理器(称为主处理器),它能执行操作系统并能操纵I/O,而其余的处理器无I/O能力(称为从处理器),只在主处理器的监控之下执行用户代码,这时称为非对称多处理机。一般而言,UMA结构适于通用或分时应用。

NUMA　NUMA(Nonuniform Memory Access)模型是非均匀存储访问模型的简称。图1.23示出了NUMA多处理机模型,其中(a)为共享本地存储器的NUMA;(b)为层次式机群NUMA,NUMA的特点是:①被共享的存储器在物理上是分布在所有的处理器中的,其所有本地存储器的集合就组成了全局地址空间;②处理器访问存储器的时间是不一样的:访问本地存储器LM或群内共享存储器CSM较快,而访问外地的存储器或全局共享存储器GSM较慢(此即非均匀存储访问名称的由来);③每台处理器照例可带私有高速缓存,且外设也可以某种形式共享。



(a) 共享本地存储器模型　　(b) 层次式机群模型

图1.23　NUMA多处理机模型

COMA　COMA(Cache Only Memory Access)模型是全高速缓存存储访问的简称。图1.24示出了COMA多处理机模型,它是NUMA的一种特例。其特点是:①各处理器节点中没有存储层次结构,全部高速缓存组成了全局地址空间;②利用分布的高速缓存目录D进行远程高速缓存的访问;③COMA中的高速缓存容量一般都大于2级高速缓存容量;④使用COMA时,数据开始时可任意分配,因为在运行时它最终会被迁移到要用到它的地方。这种结构的机器实例有瑞典计算机科学研究所的DDM和Kendall Square Research公司的KSR-1等。

CC-NUMA　CC-NUMA(Coherent-Cache Nonuniform Memory Access)模型是高速缓存一致性非均匀存储访问模型的简称。图1.25示出了CC-NUMA多处理机模型,它实际上是将一些SMP机器作为一个单节点而彼此连接起来所形成的一个较大的系统。其特点是:①绝大多数商用CC-NUMA多处理机系统都使用基于目录的高速缓存一致性协议;②它在保留SMP结构易于编程的优点的同时,也改善了常规SMP的可扩放性问题;③CC-NUMA实际上是一个分布共享存储的

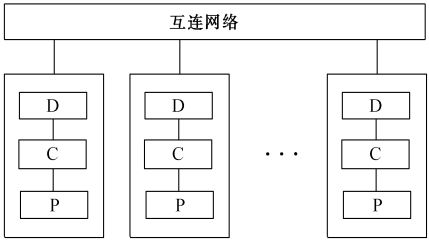


图 1.24　COMA 多处理机模型

DSM 多处理机系统；④它最显著的优点是程序员无需明确地在节点上分配数据，系统的硬件和软件开始时自动在各节点分配数据，在运行期间，高速缓存一致性硬件会自动地将数据移至要用到它的地方。总之，CC-NUMA 所发明的一些技术在开拓数据局部性和增强系统的可扩放性方面很有效。不少商业应用，大多数数据访问都可限制在本地节点内，网络上的主要通信不是传输数据，而是为高速缓存的无效性（Invalidation）所用。

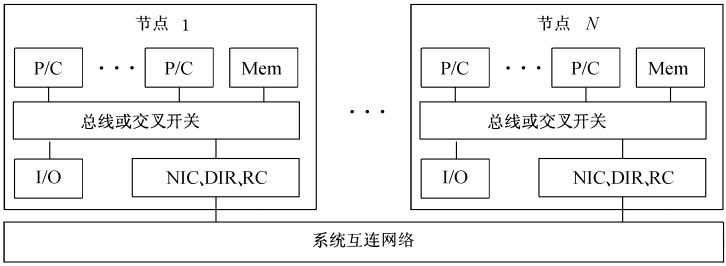


图 1.25　CC-NUMA 结构模型（RC：远程高速缓存）

NORMA　NORMA（No-Remote Memory Access）模型是非远程存储访问模型的简称。在一个分布存储的多计算机系统中，如果所有的存储器都是私有的、仅能由其处理器所访问时就称为 NORMA。图 1.26 示出了基于消息传递的多计算机一般模型，系统由多个计算节点通过消息传递互连网络连接而成，每个节点都是一台由处理器、本地存储器和/或 I/O 外设组成的自治计算机。NORMA 的特点是：①所有存储器均是私有的；②绝大多数 NUMA 都不支持远程存储器的访问；③在 DSM 中，NORMA 就消失了。

小结　可以将上节所讲的并行机结构模型和本节所讲的并行机访存模型的相互关系汇总在图 1.27 中。注意，物理上分布的存储器从编程的观点看可以是共享的或非共享的；共享存储结构（多处理机）可同时支持共享存储和消息传递编程模

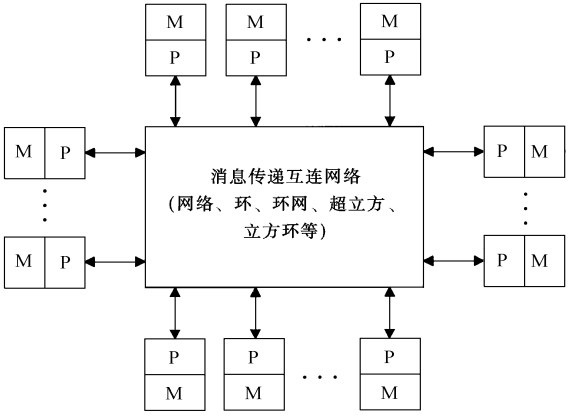


图 1.26 消息传递多计算机一般模型

型 共享存储的编程模型可同时执行于共享存储结构和分布式存储结构 (多计算机) 上。

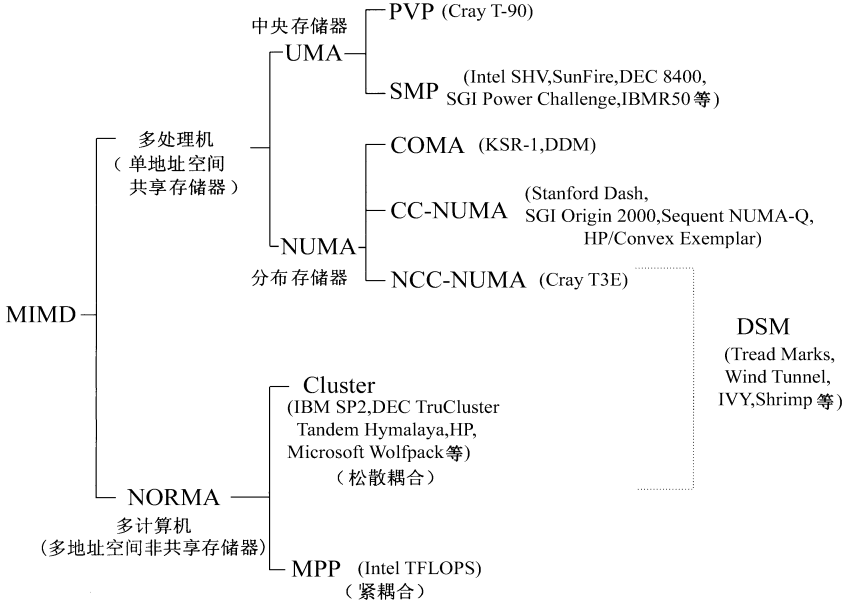


图 1.27 构筑并行机系统的不同存储结构

*1.3.3　并行计算机存储组织

下面从系统的存储器组织方式来讨论近代计算机中层次存储技术及其一致性管理。

层次存储技术　在近代计算机中,存储设备都照例按图 1.28 所示的那样将按层次组织的寄存器和高速缓存装在处理器芯片或处理器板上的。寄存器的分配由编译器完成;高速缓存对程序员是透明的(它可按速度和应用要求有一级或多级);主存储器是计算机系统的基本存储器,它由存储管理部件和操作系统共同管理;磁盘存储器被看作是最高层的联机存储器,它保存系统程序(OS 和编译器)、某些用户程序及其数据集;磁带机是脱机存储器,用作后援存储器,它保存当前或过去的用户程序副本、处理结果和文件等。磁盘驱动器和磁带机是由 OS 采取有限的用户干预方式进行管理的。

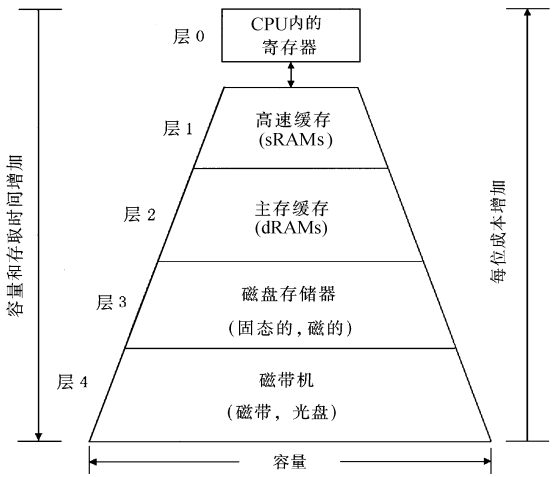


图 1.28　存储器的层次结构

存储器相邻层之间的数据传输,是按图 1.29 所示的那样以不同的单位进行的:CPU 和高速缓存之间数据按字(4 个字节)传输(高速缓存(M_1)通常分成一些高速缓存块,有的作者称之为高速缓存行(Cache Line),每块典型值是 8 个字);高速缓存和主存储器(M_2)之间数据按块(32 个字节)传输;主存储器和磁盘(M_3)之间数据按页传输(每页可包含 128 个块);磁盘和磁带机(M_4)之间数据按段传输(段的大小由用户按需要而定)。

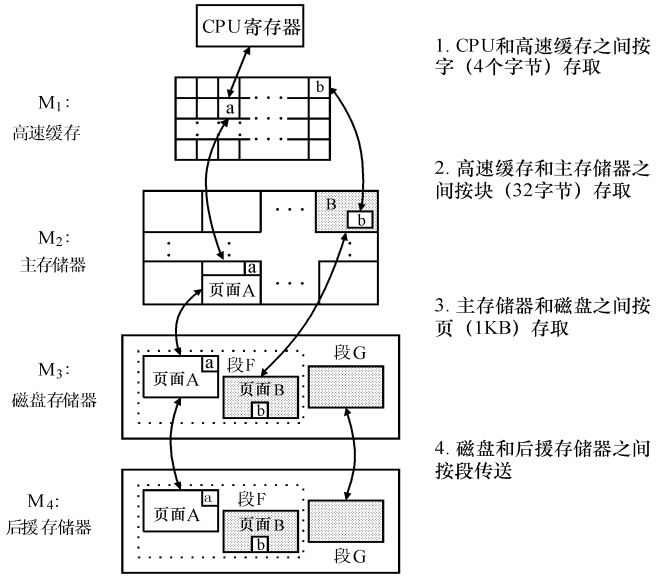


图1.29 存储器相邻层之间的数据传输

高速缓存一致性 在多层存储系统中，为了平滑处理器和主存储器的速度，当某一处理器第一次访问某一存储单元时，系统会将其内容的副本也同时传给与该处理器相连的高速缓存，以后当处理器再次访问此数据时，它就可以直接访问其高速缓存。如果另一个处理器也访问此同一存储单元，则此数据的副本也将传给其相连的另一高速缓存。在此情况下，如果其中一个处理器改写了其高速缓存中的内容，但另一处理器的高速缓存中的内容却仍是原来的值，于是造成了高速缓存中的数据不一样了，此即所谓高速缓存一致性（Coherence）问题。高速缓存写策略一般有两种：①写通过 WT（Write Through），即如果在 M_i（$i=1,2,\cdots$）中修改了一个字，则在 M_{i+1}中需要立即修改；②写回 WB（Write Back），即在 M_{i+1}中的修改延迟到 M_i中正在修改的字被替换或从 M_i中消除后才进行。在多处理机系统中，由于多台处理机异步地相互操作，因此多个高速缓存中的同一高速缓存行的副本可能不同。造成高速缓存不一致性的原因来源于：①由共享可写数据所造成的不一致；②由进程迁移所造成的不一致；③由绕过高速缓存的 I/O 操作所造成的不一致。假定有一个由 2 台处理器组成的多处理机系统，每台处理器各有一个高速缓存，且它们共享主存。设 X 是两台处理器已访问过的共享数据，在修改之前，X 的 3 个

副本是一致的。如图 1.30 所示,如果 P_1 使用 WT 策略将其高速缓存中的副本更改为 X′,则存储器中的副本也立即更改为 X′,此时两个高速缓存中的副本就不一致了;同样,如果使用 WB 策略时也会造成两个高速缓存的副本不一致。至于由进程迁移所造成的不一致性和由绕过高速缓存的 I/O 操作所造成的不一致性,读者可参阅本章习题 1.8 和 1.9 学习之。

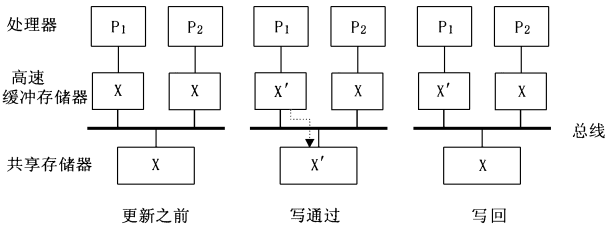


图 1.30　共享可写数据所造成的不一致性

监听总线协议　在基于总线连接的多处理机系统中,总线是保证高速缓存一致性最方便的装置,因为它能使所有的处理器观察到存储器正在进行的业务活动。如果总线业务破坏了本地高速缓存中数据的一致性状态,那么高速缓存的控制器就应采取相应的动作使本地的副本无效。采用此机制保证高速缓存一致性的协议称为监听协议(Snoopy Protocol)。

如果接在公共总线上的处理器均有自己的私有高速缓存,那么可使用写无效(Write Invalidate)和写更新(Write Update)两种策略来确保高速缓存的一致性。前者是在本地高速缓存的数据块更新时使所有远程的副本均无效;后者是把更新的数据块广播给含该数据块的所有高速缓存。

图 1.31 表示了采用 WT 策略的写无效和写更新的一致性协议是如何维护共享存储模块中 X 与 3 个本地高速缓存中的副本的一致性的。采用写无效协议时,如图 1.31 (b) 所示,当处理器 P_1 将其高速缓存中的 X 修改为 X′时,通过总线使所有其它的高速缓存中的内容均无效(用 I 表示);采用写更新协议时,如图 1.31 (c) 所示,当处理器 P_1 将其高速缓存中的 X 修改为 X′时,通过总线 X′广播给所有其它的高速缓存。因为是采用 WT 策略,还需要更新主存中的副本。至于采用 WB 策略的一致性维护过程,读者可参阅本章习题 1.10 理解之。

基于目录的协议　上面介绍的监听协议为总线连接的多处理机系统所使用,对于多级互连网络连接的多处理机系统则使用基于目录的协议(Directory Based Protocol),它就是使用一个目录来记录共享数据的所有高速缓存行的位置和状态。

图 1.32 示出了基于目录的高速缓存一致性方案的基本原理,其存储器/高速缓存更新工作如下:高速缓存 C_2 的读缺失(Read Miss)产生一个请求送给 D_1(图中

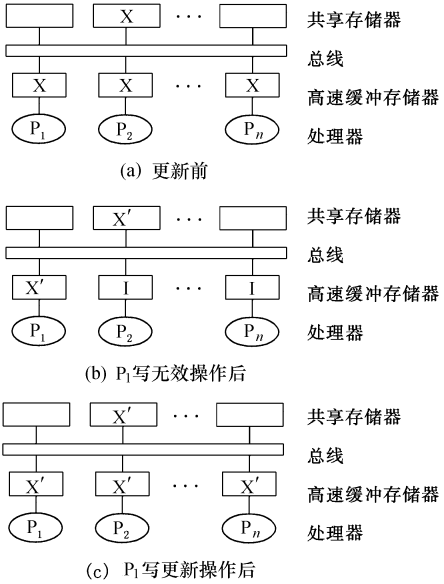


图 1.31 采用 WT 策略的一致性维护

用细线表示),D_1 指示在 C_1 中有可用(Clean)副本,存储控制器再将请求传至 C_1,它返回一个可用副本给 M_1 和 C_2。在 C_1 写命中时(图中用粗线表示),它就发送一个命令给存储控制器,存储控制器再发一个无效命令给在 D_1 中有标记的所有高速缓存(在此为 C_2)。

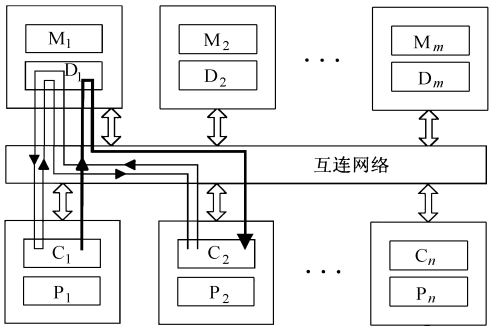


图 1.32 基于目录的高速缓存一致性原理

总之,不用广播的高速缓存一致性协议,必须将所有高速缓存中每个共享数据块副本的地址存储起来,这张高速缓存地址表,不管它是集中的或分散的,都称之为高速缓存目录。每个数据块(即高速缓存行)的目录项包括大量的指针,用来指向该高速缓存行的所有远程副本的地址。每个目录项还包含一个重写位(Dirty Bit),用来说明是否有一个高速缓存允许把有关的数据块写入。

目前已有各种不同的目录协议,包括全映射(Full Map)目录、有限(Limited)目录和链式(Chained)目录等。

1.4 小结和导读

小结 本章从并行计算的角度出发,着重讨论当代科学和工程计算问题所要求的高性能并行计算系统,包括并行计算机系统的互连技术、并行计算机的系统结构模型、存储访问模型和存储结构组织等。值得注意的是,所介绍的并行计算机系统,与计算机体系结构或并行处理技术等课程所讲的出发点不一样,在此着重介绍一类并行计算机的体系结构,而不是某一具体并行机的详细介绍;着重介绍当代能满足高速并行计算要求的主流并行计算机类(SMP、MPP、COW 等),而对历史上曾占主导地位的 SIMD 和传统的 PVP 则不作重点介绍;同时所选讲的内容和叙述的深浅程度都是为了并行算法的设计和并行编程的需要,而对有些相关知识的介绍,也只能点到为止,不能深入全面展开讨论,尤望读者能以此为线索,进一步追踪和深入地学习。

最后,在电子学和计算机科学中,习惯上,常用的量词单位均以千进制表示,为了读者查阅方便,兹将它们列于表 1.8 中。

导读 有关美国"高性能计算与通信"计划,读者可参阅[98],有关美国"加速战略计算创新"计划,读者可参阅[17]。有关国产曙光系列并行机介绍,请参阅[129,195,198]。关于互连网络,[202]是一本基本的教材,近期有关互连网络方面的书可参见[62],有关构筑可扩放并行机的千兆位网可参阅[102],[164,165]综述了 LAN、MAN 和 ISDN 的新进展,有关 FDDI 环可参见 FDDI 手册[167],Gb 以太网的描述可参见 Gb 以太网联盟公布的材料[75],有关 Myrinet 的介绍,读者可参阅[36],HiPPI 网络情况读者可查阅[173],读者从 ATM 论坛那里可了解 ATM[18]。有关当前并行计算机的介绍,读者可参阅[174]一书
的结构模型作了讨论,
[168]。关于高速缓存一致性方法的综合讨论读者可见[3,196],有关基于目录的高速缓存一致性协议读者可参阅[43,100]。最后,[103]是一本论述有关可扩放并

行计算的新著，取材宽广、新颖，特推荐阅读。

表 1.8　计算机科学中常用千进制单位量词一览表

词头	缩写	英文含意	数值	中文名称
milli	m	one thousandth	10^{-3}	毫
micro	μ	one millionth	10^{-6}	微
nano	n	one billionth	10^{-9}	纳［诺］
pico	p	one trillionth	10^{-12}	皮［可］
femto	f	one quadrillionth	10^{-15}	飞［姆托］
atta	a	one quintillionth	10^{-18}	阿［托］
kilo	k	thousand	10^{3}	千
mega	M	million	10^{6}	兆，百万
giga	G	billion	10^{9}	吉［咖］，十亿
tera	T	trillion	10^{12}	太［拉］，万亿
peta	P	quadrillion	10^{15}	拍［它］，千万亿
exa	E	quintillion	10^{18}	艾［可萨］

习　题

1.1　对于一棵 K 级二叉树（根为 0 级，叶为 K 级），共有 $N=2^K-1$ 个节，当推广至 m 元树（即每个非叶节点有 m 个子节点）时，试写出总节点数 N 的表达式。

1.2　二元胖树如图 1.33 所示，此时所有非根节点均有 2 个父节点。如果将图中的每个椭圆均视为单个节点，并且成对节点间的多条边视为一条边，则它实际上就是一棵二叉树。试问：如果不管椭圆，只把小方块视为节点，则它从叶到根形成一个什么样的多级互连网络？

1.3　四元胖树如图 1.34 所示，试问：每个内节点有几个子节点和几个父节点？你知道哪个机器使用了此种形式的胖树？

1.4　试构造一个 $N=64$ 的立方环网络，并将其直径和节点度与 $N=64$ 的超立方比较之，你的结论是什么？

1.5　一个 $N=2^k$ 个节点的 de Bruijn 网络如图 1.35 所示。令 $a_{k-1}a_{k-2}\cdots a_1a_0$ 是一个节点的二进制表示，则该节点可达如下两个节点：

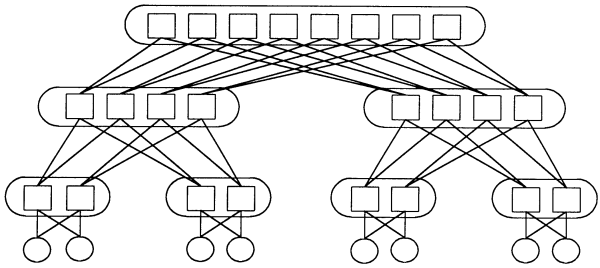


图 1.33　二元胖树

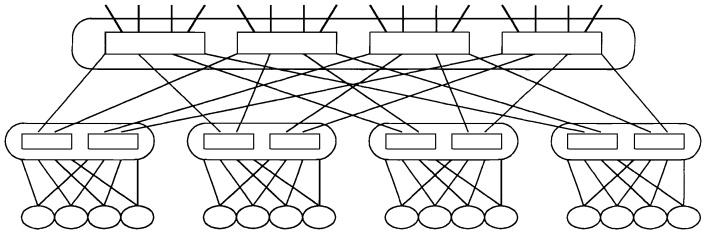


图 1.34　四元胖树

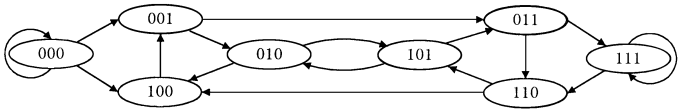


图 1.35　$N=8$ 的 de Bruijn 网络

$$a_{k-2}\,a_{k-3}\cdots a_1\,a_0 0$$

$$a_{k-2}\,a_{k-3}\cdots a_1\,a_0 1$$

试问：该网络的直径和对剖宽度为多少？

1.6　一个 $N=2^n$ 个节点的洗牌交换网络如图 1.36 所示。试问：此网节点度＝？网络直径＝？和网络对剖度＝？

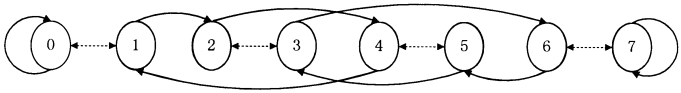


图 1.36　$N=8$ 的洗牌交换网络

1.7　一个 $N=(k+1)2^k$ 个节点的蝶形网络如图 1.37 所示。

试问：此网节点度＝？网络直径＝？和网络对剖宽度＝？

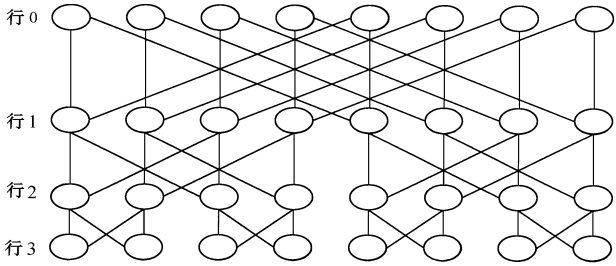


图 1.37　$k=3$ 的蝶形网络

1.8　参照图 1.38,试解释为什么当采用 WT 策略进程从 P_2 迁移到 P_1 时,或采用 WB 策略包含共享变量 X 的进程从 P_1 迁移到 P_2 时,会造成高速缓存的不一致。

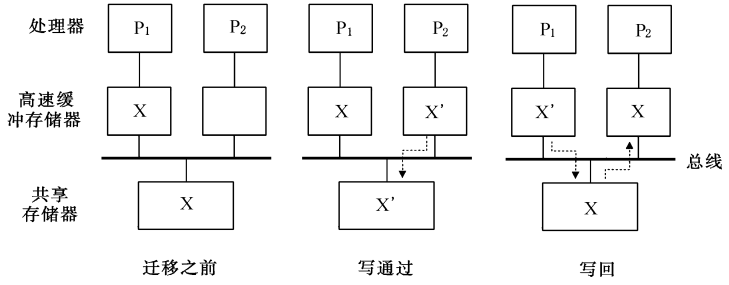


图 1.38　进程迁移所造成的不一致性

1.9　参照图 1.39,试解释为什么:①当 I/O 处理器将一个新的数据 X'写回主存而绕过采用 WT 策略的高速缓存时会造成高速缓存和主存间的不一致;②当直接从主存输出数据而绕过高速缓存采用 WB 策略时也会造成不一致。

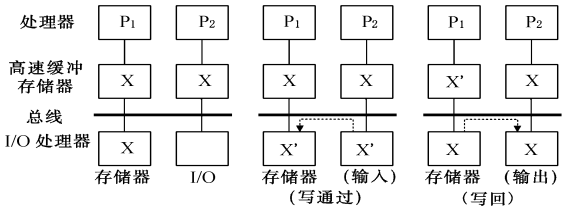
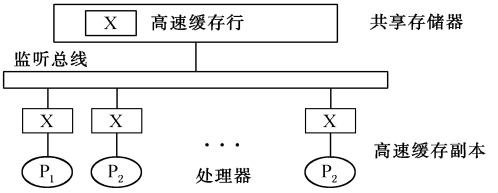
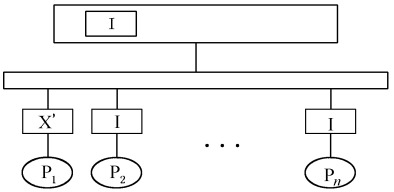


图 1.39　绕过高速缓存的 I/O 操作所造成的不一致性

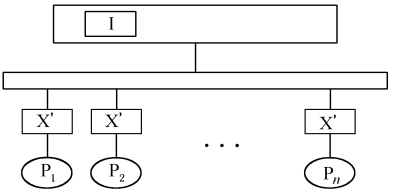
1.10　参照图1.40,试解释采用WB策略的写更新和写无效协议的一致性维护过程。其中,X为更新前高速缓存中的副本,X′为修改后的高速缓存行,I为无效的高速缓存行。



(a) 写操作前



(b) 由P_1执行写无效后



(c) 由P_1执行写更新后

图1.40　采用WB策略的写更新和写无效协议一致性维护

第二章　当代并行计算机系统介绍

自20世纪70年代初到现在，并行计算机的发展已有很长的历史了。在此期间，出现了各种不同类型的并行机，包括向量机、SIMD计算机和MIMD计算机。随着计算机的发展，曾经风行一时的传统的向量机和SIMD计算机现已退出历史舞台，而MIMD类型的并行机却占了主导地位。当代主流的并行计算机是可扩放的并行计算机，包括对称多处理机和大规模并行处理机以及机群系统。本章首先讨论共享存储多处理机系统；然后讨论分布存储多计算机系统；最后讨论机群系统（包括工作站机群COW）。

2.1 共享存储多处理机系统

共享存储的对称多处理机 SMP (Symmetric Multiprocessor) 结构在现今的并行服务器中几乎普遍采用。SMP 系统属于 UMA (Uniform Memory Access) 机器，NUMA (Nonuniform Memory Access) 机器是 SMP 系统的自然推广，而 CC-NUMA (Coherent-Cache NUMA) 实际上是将一些 SMP 作为单节点而彼此连接起来所构成的分布共享存储系统 (见图 1.25)。本节首先讨论对称多处理机系统的结构特性，然后简介一个当今有代表性的商用超级服务器系统 CC-NUMA Origin 2000，以期对分布共享存储多处理机系统有个一般了解。

2.1.1 对称多处理机 SMP 结构特性

SMP 结构特性　参照图 1.20 (b)，共享存储的 SMP 系统结构具有如下特性：①对称性：系统中任何处理器均可访问任何存储单元和 I/O 设备；②单地址空间：单地址空间有很多好处，例如因为只有一个 OS 和 DB 等副本驻留在共享存储器中，所以 OS 可按工作负载情况在多个处理器上调度进程从而易达到动态负载平衡，又如因为所有数据均驻留在同一共享存储器中，所以用户不必担心数据的分配和再分配；③高速缓存及其一致性：多级高速缓存可支持数据的局部性，而其一致性可由硬件来增强；④低通信延迟：处理器间的通信可用简单的读/写指令来完成 (而多计算机系统中处理器间的通信要用多条指令才能完成发送/接收操作)。

目前大多数商用 SMP 系统都是基于总线连接的，占了并行计算机很大的市场，但是 SMP 也具有如下问题：①欠可靠：总线、存储器或 OS 失效均会造成系统崩溃，这是 SMP 系统的最大问题；②可观的延迟：尽管 SMP 比 MPP 通信延迟要小，但相对处理器速度而言仍相当可观 (竞争会加剧延迟)，一般为数百个处理器周期，长者可达数千个指令周期；③慢速增加的带宽：有人估计，主存和磁盘容量每 3 年增加 4 倍，而 SMP 存储器总线带宽每 3 年只增加 2 倍，I/O 总线带宽增加速率则更慢，这样存储器带宽的增长跟不上处理器速度或存储容量的步伐；④不可扩放性：总线是不可扩放的，这就限制最大的处理器数一般不能超过 10。为了增大系统的规模，可改用交叉开关连接，或改用 CC-NUMA 或机群结构。

商用 SMP 服务器　SMP 是现今最成功的并行计算机，它经常工作在网络环境中，所以其安全性、集成能力和易于管理是很重要的。表 2.1 综合比较了现今流行的 5 种商用 SMP 系统特性，这些参数均是最大或最优值 (表中 B 为字节)。

表 2.1 5种商用 SMP 系统特性比较一览表

系统特性	DEC Alphaserver 8400/440	HP9000/T600	IBM RS600/R40	Sun Ultra Enterprise 6000	SGI Power Challenge XL
处理器数目	12	12	8	30	36
处理器类型	437 MHz Alpha21164	180 MHz PA8000	112 MHz PowerPC 604	167 MHz UltraSPARC I	195 MHz MIPS R10000
处理器片外高速缓存容量	4 MB	8 MB	1 MB	512 MB	4 MB
最大主存容量	28 GB	16 GB	2 GB	30 GB	16 GB
互连网络及带宽	BUS 2.1 GB/s	BUS 960 MB/s	BUS+ Crossbar 1.8 GB/s	BUS+ Crossbar 2.6 GB/s	BUS 1.2 GB/s
外磁盘容量	192 GB	168 GB	38 GB	63 GB	144 GB
I/O 通道	12 PCI 总线,每个 133 MB/s	N/A	2MCA, 每个 160 MB/s	30 Sbus, 每个 200 MB/s	6 Power Channel-2HIO, 每个320 MB/s
I/O 槽	144PCI 槽	112HP-PB 槽	15MCA	45 Sbus 槽	12 HIO 槽
I/O 带宽	1.2 GB/s	1 GB/s	320 MB/s	2.6 GB/s	每个 HIO 槽 320 MB/s

*2.1.2 CC-NUMA Origin 2000 超级服务器

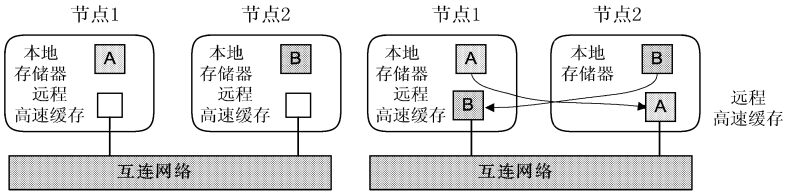
CC-NUMA 系统 如图 1.25 所示,高速缓存一致性非均匀存储访问 CC-NUMA 机,就是将一些小规模的 SMP 作为单节点,通过系统互连网络扩展至一个大规模多处理机系统。SMP 节点中的所有局部存储器构成了共享主存,这样就可保持 SMP 易编程的优点,同时由于可方便地加入更多的节点从而改善了常规 SMP 系统的可扩放性。因为对于给定的应用,可利用数据局部性,在绝大部分时间内同时访问多个局部存储器,从而也缓解了竞争和带宽问题。

目前绝大多数商用 CC-NUMA 均使用基于目录的高速缓存一致性协议。CC-NUMA 机器最显著的优点是程序员无需明确地在节点上分配数据,系统的硬件和软件开始时自动在各节点中分配数据,在应用程序运行期间,高速缓存一致性硬件设施会自动地将数据移至要使用它的地方。例如某一程序有 P 和 Q 两个进程,执

行如下访问数组 A 和 B 的代码段：

	P	Q
第一步：	Use (A)	Use (B)
第二步：	Use (B)	Use (A)

图 2.1(a) 所示的初始数据分配是很适合上述代码段的第一步的，在第二步时，硬件能自动地将 B 移入节点 1 和将 A 移入节点 2，程序员不必作这些，但在多计算机系统中却要求程序员作明确的数据分配。



(a) 第一步的数据分配　　(b) 第二步的数据再分配

图 2.1　CC-NUMA 使用远程高速缓存重新分配数据

1996 年 10 月，SGI/Cray 公司推出了 Origin 2000 系统，NUMA 结构，此结构也被称之为可扩放共享存储多处理机 S^2MP 或 S2MP (Scalable Shared Memory Multiprocessor)。设计时吸取了 DASH 的经验，所设计的系统规模可达到 1 024 个处理器和 1TB 主存，其中 1～64 个处理器的系统叫 SGI Origin 2000，而大于等于 128 个处理器的系统叫 Cray Origin 2000。现有的 Origin 2000 的结构配置属性综合于表 2.2 中。

表 2.2　Origin 2000 结构配置属性一览表

属　性	立式 (Deskside) 配置	装架式 (Rack) 配置
处理器数目	1～8	2～128
峰值速度 (Gflops)	3.12	49.92
高速缓存容量 (MB)	1～32	2～512
物理存储器容量 (GB)	0.064～16	0.064～256
总计峰值存储带宽 (GB/s)	3.12	49.92
I/O 端口数目	14	208
总计峰值 I/O 带宽 (GB/s)	6.2	102

系统结构 Origin 2000 系统结构如图 2.2 所示。它的每个节点包括 1 或 2 个 MIPS R10000 处理器（时钟 195 MHz，峰值速度 395 Mflops），一个可高达 4 MB 的高速缓存和一个 DSM 主存（物理上分散在各节点中，但可被所有节点访问），硬件增强了基于目录的高速缓存一致性。如图 2.3 所示，一个 HUB（ASIC 交叉开关）连接系统中的处理器、存储器、互连网络和 I/O 系统，HUB 有 4 个双向端口，使用 195 MHz 的时钟，每个端口可提供单向 780 MB/s 的峰值带宽和全双工 1.56 GB/s 的峰值带宽，它用于节点内和节点间通信选路，4 个成对的接口电路（ififo，ofifo）负责内部和外部信息格式的转换。

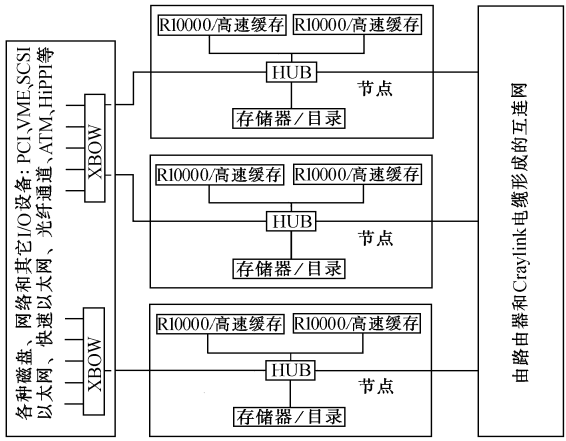


图 2.2 Origin 2000 系统结构

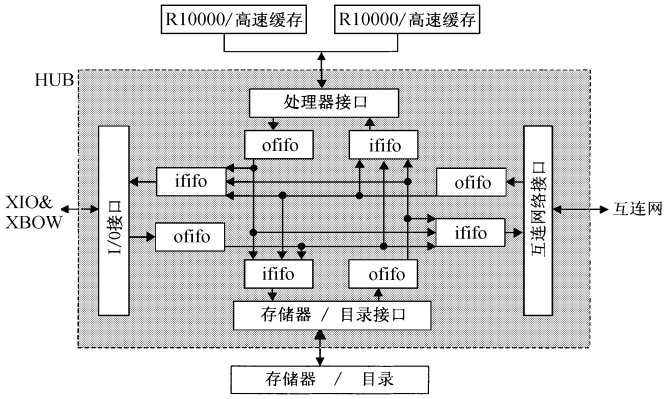


图 2.3 Origin 2000 中交叉开关 HUB 芯片

(1) I/O 子系统：Origin 2000 I/O 子系统示于图 2.4，其中，XBOW 是个 8 端口的交叉开关，2 个连向节点，6 个连向 XIO 设备或者 ASIC 小部件，包括图形小部件 G (Graphics Widget) 和桥接小部件 B (Bridge Widget)。通过 32 位或 64 位的 PCI 总线，不同的 I/O 设备 (ATM、HiPPI、FDDI、SCSI 等) 均可连向 Origin 2000。Origin 2000 可提供单一 I/O 空间，即任何处理器能访问任一 I/O 设备。

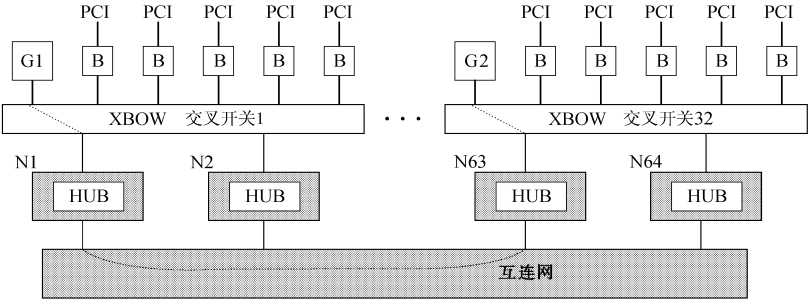


图 2.4　Orgin 2000 的 I/O 子系统

(2) 互连系统：Origin 2000 的互连系统由 CrayLink 电缆和图 2.5 所示的路由器组成。SGI SPIDER 路由器提供可靠的虫蚀选路，它是 6 端口的交叉开关，这些端口连向节点或别的路由器。路由器允许 6 个端口同时全双工工作。由于外部频率 (390 MHz) 和路由器芯片内部频率 (97.5 MHz) 的不同，所以需使用源同步驱动器 SSD (Source Synchronous Driver) 和源同步接收器 SSR (Source Synchronous Receiver) 施行内/外频率 (97.5 MHz/390 MHz) 和字长 (64 位/16 位) 的转换。链路级协议 LLP (Link Level Protocol) 确保信息可靠传输 (使用 CRC 误差检测和重发校正)，施行流控、缓解竞争和拥挤等。

(3) 胖超立方拓扑：Origin 2000 中采用了胖超立方 (Fat Hypercube) 拓扑。如

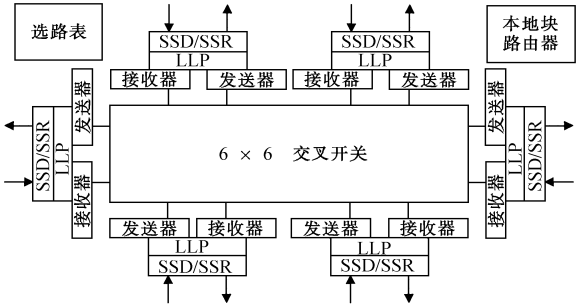


图 2.5　Origin 2000 的 SPIDER 路由器芯片

图 2.6 所示,它是将传统的 2 元超立方加以修改而得,其优点是保持了超立方线性对剖宽度的优点且避免了节点度的增加。在此拓扑中,每个路由器 R 有 6 个端口 2 个连向节点,4 个用于网络连接。一般的超立方,度为 4 的节点最多只能构造 16 个顶点的超立方(即 Origin 2000 中的 32 个节点),但使用了胖超立方,可允许 Origin 2000 连接多达 512 个节点(1 024 个处理器)。图 2.6(a)只有一个节点,它有 2 个处理器,还可连向 XBOW 交叉开关;图 2.6(b)中,一个 SPIDER 路由器 R 连向 2 个节点;图 2.6(c)中 2 个 R 连向 4 个节点;以此方式可达 32 个节点(图 2.6(f))。如果把 R 视为超立方的顶点,则它就是普通的 2 元超立方连接。注意,未被使用的路由器端口可连向快速链路(Express Link)以降低延迟和增大对剖带宽(如图 2.6(c)中的虚线连接)。超过 32 个节点时,Origin 2000 的胖超立方中使用了额外的路由器,称之为元路由器(Metarouter),8 个元路由器构成一个 Cray Router。这些元路由器只用作中间选路,不连向任何节点。一个 64 个节点(32 个顶点,即路由器)的胖超立方拓扑如图 2.6(g)所示,其中 4 个 8 顶点的子立方中每个顶点都连向一个元路由器。这种胖超立方可一直扩放到 1 024 个处理器。

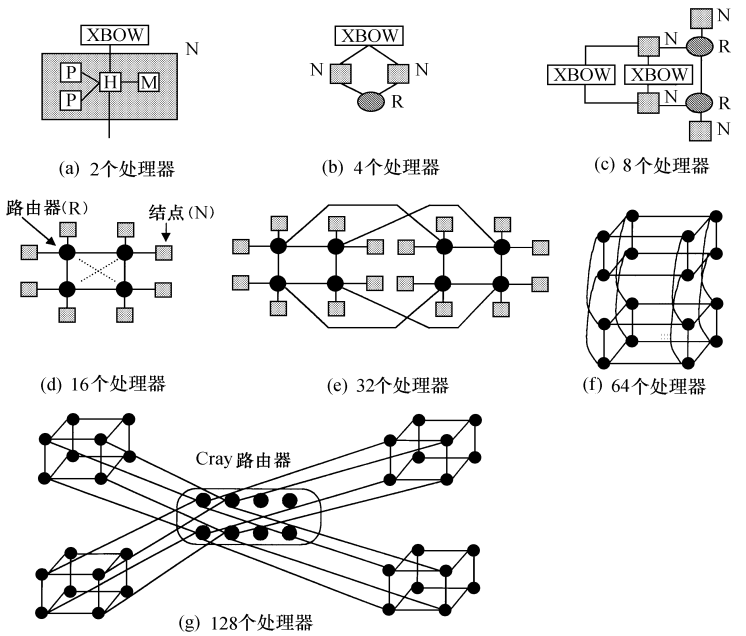


图 2.6 Origin 2000 的胖超立方拓扑

⑷ 分布共享存储子系统：Origin 2000 系统中有一个分布式共享存储子系统，节点内的存储模块叫近存储器，别的节点中的存储模块叫远存储器。节点中的主存储器和目录存储器使用同步 DRAM 可组织成多达 8 个体（Bank），每个体提供 4 路交叉访问。使用 4 路交叉访问的存储子系统带宽高达 780 MB/s。

软件环境 Origin 2000 运行细胞（Cellular）IRIX 操作系统。其 1996 年版本是一个 64 位的多线程、多道程序操作系统，它的设计目标要满足可兼容性、可扩放性（扩展到 128 个或更多处理器）、可用性和高吞吐率的要求。

⑴ 细胞概念：为了增强细胞 IRIX 的可扩放性，使用了细胞概念（Cell Concept）和存储器管理软件的办法。前者能模块化操作系统核，后者能有效地开拓 S2MP 的硬件。细胞 IRIX 结构把核文本和数据分散成 SMP 大小的称作细胞的块，其中每个细胞包含一个核文本和数据结构的副本，并负责控制一个由一组处理器、存储器和 I/O 设备组成的机器模块，细胞概念的关键特性是将操作系统服务分散化和局部化，多个细胞可驻留在一个 Origin 2000 系统中，允许一些应用同时地、独立地访问多个机器模块。

⑵ 存储管理：细胞 IRIX 的存储管理软件包括存储器复制、迁移和置放机制（存储管理的基本单位是页）：复制机制复制一个只读页的副本。当一个页是只读时，硬件检测共享情况并调用存储管理子系统执行存储器复制；迁移机制移动页面靠近最经常访问它的处理器。当一页被访问时，硬件计数器记录外节点的访问次数，如其值超过某一阈值，则存储管理子系统将迁移该页面；置放机制是初始分配数据和线程以开拓局部性。细胞 IRIX 环境允许用户告诉系统页面彼此间应如何置放、如何映射线程到页面和如何定位线程离硬件资源的远近。

⑶ 可用性支持：Origin 2000 的可用性支持包括硬件和软件：硬件设计使用冗余供电与冷却、热可插磁盘、廉价冗余磁盘阵列 RAID（Redundant Array of Inexpensive Disks）等技术，因特网支持节点间多条路径、硬件链路层协议提供错误包的检测与重发等；IRIX 系统软件包含进一步增强可用性支持（如文件系统快速再启动、监视通信故障时间、磁盘镜像等），提供访问保护权限和检查点再启动机制以及高有效网络、数据库与其它应用的附加支持（通过支持队列、套接字、X Windows、图形等）。

⑷ 吞吐率支持：Origin 2000 通过使用优化调度（它支持大量的处理器和用户进程）和一个 64 位的称之为 xFS 的日志文件系统（它支持高吞吐率和确保 I/O 速率）来有效地使用系统和增强系统吞吐率。细胞 IRIX 提供 3 种调度机制：①帧调度（Frame Scheduling）：帧调度器能完全掌管调度和分发进程给一个或多个 CPU 以确保进程执行的精确速率；②组调度（Gang Scheduling）：细胞 IRIX 能调度一组线程，处理器相似（Processor Affinity）：

CPU 上。xFS 文件系统采用范围(Extent)概念和日志(Journaling)技术来达到高吞吐率,采用允许应用预约文件系统带宽来确保 I/O 速率。范围的使用有助于减少磁盘寻找时间和旋转延迟,因而增加了 I/O 吞吐率;日志(也称为记录,Logging)是一种记录所有文件系统变化的技术,此技术也有助于减少系统恢复时间,因为系统崩溃后,系统只访问少量的最近更新的文件登录,而不必像 Unix 文件系统那样执行很花时间的文件系统检查。

系统性能 Origin 2000 结构设计很注重整体特性,所以存储器、I/O 和互连网络的能力都能随机器规模的增大而成比例地增加。 gin 2000 中不同层次存储器的读延迟,表 2.4 列出它的总计带宽和存储器读延迟。

表 2.3 Origin 2000 不同层次存储读延迟一览表

数据位置	读延迟(时钟数)
寄存器	0
1 级高速缓存(片上)	1～3
2 级高速缓存(片外)	10
本地存储器	61
远程存储器(1 个路由器以远)	117
远程存储器(2 个路由器以远)	137
远程存储器(3 个路由器以远)	157

表 2.4 Origin 2000 带宽(GB/s)和延迟(ns)一览表

系统配置 I/O:节点:CPU	总计峰值 存储器带宽	总计峰值 I/O 带宽	互连网络 对剖带宽	存储器 读延迟
1:1:2	0.78	1.56	—	313
2:2:4	1.56	3.12	1.56	497
2:4:8	3.12	6.24	3.12	601
4:8:16	6.24	12.48	6.24	703
8:16:32	12.48	24.96	12.48	805
16:32:64	24.96	51.2	12.48	908
32:64:128	49.92	99.84	24.96	1 112

由此可看出,总计带宽(存储器、I/O 和互连网络)几乎随处理器数线性增加,

而延迟只是对数增长。这种低延迟和高带宽使得 Origin 2000 系统在市场上具有很大的优势。

小结 表 2.5 列出了常用的 4 种 CC-NUMA 结构特性比较,其中 DASH 是个研究样机,而其它几种都已是商品。表中 SCI (Scalabel Coherence Interface) 是可扩放一致性接口,它是将传统的总线式底板扩展至全双工、点到点互连结构,SCI 互连可以是环形结构或交叉开关结构。

表 2.5 4 种 CC-NUMA 结构比较一览表

结构 特性	Stanford Dash	Sequent NUMA-Q	HP/Convex EXemplar	SGI/Cray Origin 2000
节点 结构	4CPU SMP 节点 监听总线	4CPU SMP 节点 监听总线	8CPU SMP 节点 交叉开关	2CPU 非 SMP 节点 HUB
节点互连	2D 网孔	SCI 环	多 2D SCI 环	胖超立方
高速缓存 一致性协议	节点内监听 全局目录	SCI 链表 一致性协议	修改的 SCI 协议	修改的 Dash 协议
其它	节点内高速缓存到高速缓存共享	节点高速缓存组调度 相似性调度	节点高速缓存	组调度、页面迁移、置放、复制

2.2 分布存储多计算机系统

分布存储的大规模并行处理机 MPP (Massively Parallel Processor) 一词的含义过去并不明确,其意义常随时间而变。按照现今的技术,它是指由成百上千乃至上万个处理器组成的大型 (Large Scale) 计算机系统。1997 年 7 月由 Intel 和 Sandia 研制成的 ASCI Option Red,其处理器数已达 9 216 个,是属于高端 MPP 系统。MPP 系统是属于 NORMA (No-Remote Memory Access) 模型的机器。早期也曾把非高速缓存一致性 NUMA,即 NCC-NUMA (Non-Cache Coherent NUMA) 模型的机器 (例如 Cray T3E) 也算作 MPP 系统,但现在看来 T3E 仍划分在 NUMA 类为宜 (参见图 1.27)。同样现在也把使用机群 (Cluster) 方法构造的 MPP 系统 (例如 IBM SP2 和曙光-2000) 也有时单列为机群一类,而把 CC-NUMA 模型的机器 (例如 SGI/Cray Origin 2000) 算作是可扩放共享存储多处理机。本节首先讨论 MPP 的

结构特性及其有关问题,然后简介一个当今高端大规模并行机 Intel ASCI Option Red 以期对其有个初步了解。

2.2.1 大规模并行处理机 MPP 结构特性

MPP 公共结构 所有的 MPP 均使用物理上分布的存储器,且使用分布的 I/O 也渐渐变多。现今的 MPP 公共结构如图 2.7 所示,其中每个节点有一个或多个处理器和高速缓存(P/C)、一个局部存储器、有或没有磁盘和网络接口电路 NIC (Network Interface Circuitry),它们均连向本地互连网络(早期多为总线而近期多为交叉开关),而节点间通过高速网络 HSN (High Speed Network) 相连。

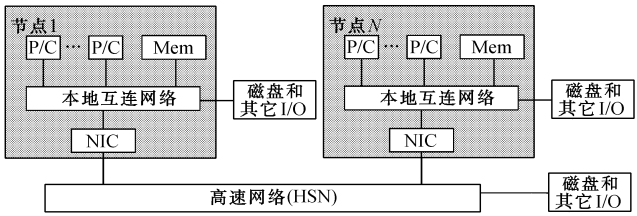


图 2.7 MPP 公共系统结构

MPP 设计问题 设计 MPP 系统所应考虑的问题是:①可扩放性:MPP 著名特性就是系统能扩展至成千上万个处理器,而存储器和 I/O 的容量及带宽亦能按比例的增加。为此,采用物理上分布的存储器结构,它能提供比集中存储器结构更高的总计存储带宽,因此有潜在的高可扩放性;要平衡处理能力与存储和 I/O 的能力,因为存储器和 I/O 子系统的速度不可能与处理器成比例地提高;要平衡计算能力与交互能力,因为进程/线程的管理、通信与同步等都相当费时间。②系统成本:因为 MPP 系统中包含大量的元件,为了保证系统的低成本应确保每个元件的低成本。为此,应采用现有的商用 CMOS 微处理器,这些芯片原为 PC 机、工作站和服务器开发的,自然成本要低,并且按照 Moore 定律 (Moore's Law) 其性能每一年半到二年要翻一番;要采用相对稳定的结构,如第一章第 1.3.1 节所讲的可扩放并行机 Shell 结构 (图 1.21);要使用物理上分布的存储器结构,它比同规模机器的中央(集中)存储器结构要便宜;要采用 SMP 节点方式以削减互连规模。但是现有的商用微处理器是为小系统(如 PC 机、工作站和 SMP 服务器等)而不是为 MPP 设计的,使用它虽在可扩放性和低成本方面有所得益,但用于 MPP 也带来一些问题,诸如微处理器地址空间不足够大(例如使用在 Cray T3D MPP 中的 Alpha 21064 微处理器,它只能提供 33 位物理地址空间,而 T3D 最大物理存储容量为

128GB),所以设计者必须加入专门硬件以扩大物理地址空间规模;微处理器和它的计算能力相比,它缺乏足够的操作系统支持,使其难以有效地支持进程管理、通信和同步等。③通用性和可用性:MPP要走向成功之路,它必须是个通用系统,能支持不同的应用(技术和商业)、不同算法范例、不同操作模式,而不能局限于很窄的应用。为此,MPP要支持异步MIMD模式(SIMD似已过时);要支持流行的标准编程模式(如PVM、MPI和HPF);诸节点应能按大、小作业要求进行不同的组合以支持交互和批处理模式;互连拓扑应对用户透明,对用户而言他(她)所看到的是一组全连接的节点;MPP应在不同层次上支持单一系统映像SSI(Single System Image),因为紧耦合的MPP常使用分布式操作系统,所以要在硬件级和OS级提供此映像;据估计1 000个处理器的MPP系统,每天至少有一个处理器失效,所以MPP必须使用高可用性的技术。④通信要求:MPP和COW的关键差别是节点间的通信,COW使用标准的LAN,而MPP使用高速、专用高带宽、低延迟的互连网络,无疑在通信方面优于COW。然而通信技术的迅速发展,COW对MPP颇具威胁,从而MPP对通信技术也提出了更高的要求。⑤存储器和I/O能力:因为MPP是可扩放系统,所以就要求非常大的总计存储器和I/O设备容量,然而I/O方面的进展仍落后于系统中的其余部分,故如何提供一个可扩放的I/O子系统就成为MPP的热门研究课题。

MPP的过去和现在　早期的MPP主要应用于科学和工程超级计算,著名的系统有TM(Thinking Machine) CM2/CM5、NSSA/Goodyear MPP、nCUBE、Cray T3D/T3E、Intel Paragon、MasPar MP1、Fujitsu VPP500和KSR1等。表2.6列出了部分MPP系统的特性比较。表2.7列出了MPP所用的几个典型微处理器特性参数,其中SPEC(Standard Performance Evaluation Corporation)代表标准性能评价公司,它对目前高性能CPU等进行整数(int)和浮点数(fp)运算性能测试。当今微处理器系列及其代表性的CPU芯片如图2.8所示。现在,很多MPP都已成功地应用在商业和网络应用中。此外,美国开始执行的ASCI计划(见第一章第1.1.2节)也包括研制三台高端MPP计算机,它们是Intel公司与SNL(Sandia国家实验室)联合研制的红选择(Option Red)、IBM公司与LLNL(Lawrence Livermore国家实验室)联合研制的蓝太平洋(Blue Pacific)和SGI公司与LANL(Los Alamos国家实验室)联合研制的蓝山(Blue Mountain),其主要性能综合于表2.8中。

表 2.6 典型 MPP 系统特性比较一览表

结构特性	Cray T3D	Cray T3E	Intel Paragon	Intel/Sandia Option Red
典型配置	512 个节点 153 Gflops	512 个节点 1.2 Tflops	400 个节点 40 Gflops	4536 个节点 1.8 Tflops
推出日期	1993	1996	1992	1996
CPU 类型	150 MHz 150 Mflops Alpha 21064	300 MHz 600 Mflops Alpha 21164	50 MHz 100 Mflops Intel i860	200 MHz 200 Mflops Pentium Pro
节点结构 数据存储	2 CPU 64 MB 主存 50 GB 共享 磁盘	4～8 CPU 256 MB～16 GB DSM 主存, 共享磁盘	1～2 CPU 16～128 MB 局存,48 GB 共享磁盘	2 CPU 32～256 MB 局存, 共享磁盘
互连网络	3-D 环绕	3-D 环绕	2－D 网孔	Split2－D 网孔
访存模型	NUMA	NCC-NUMA	NORMA	NORMA
节点 OS	微核	基于 Chorus 微核	微核	轻量级核 (LWK)
编程模型	共享变量、 消息传递、 PVM	共享变量、 消息传递、 PVM	消息传递	基于 PUMA Portals 消息传递
编程语言	MPI、HPF	MPI、HPF	NX、MPI PVM	NX、PVM、HPF
点到点通 信延迟	2 μs		30 μs	10 μs
点到点带宽	150 MB/s	480 MB/s	175 MB/s	380 MB/s

表 2.7 MPP 所用的高性能 CPU 特性参数一览表

属性	Pentium Pro	Power PC620	Alpha 21164A	Ultra SPARC II	MIPS R10000
工艺	BiCMOS	CMOS	CMOS	CMOS	CMOS
晶体管数	5.5 M/15.5 M	7 M	9.6 M	5.4 M	6.8 M
时钟频率	150 MHz	133 MHz	417 MHz	200 MHz	200 MHz
电压	2.9 V	3.3 V	2.2 V	2.5 V	3.3 V
功率	20 W	30 W	20 W	28 W	30 W
字长	32 位	64 位	64 位	64 位	64 位

（续表）

属性	Pentium Pro	Power PC620	Alpha 21164A	Ultra SPARC II	MIPS R10000
I/O 高速缓存	8 KB/8 KB	32 KB/32 KB	8 KB/8 KB	16 KB/16 KB	32 KB/32 KB
2 级高速缓存	256 KB （多芯片模块）	1～128 MB （片外）	96 KB （片上）	16 MB （片外）	16 MB （片外）
执行单元	5 个单元	6 个单元	4 个单元	9 个单元	5 个单元
超标量	3 路（Way）	4 路	4 路	4 路	4 路
流水线深度	14 级	4～8 级	7～9 级	9 级	5～7 级
SPECint 92	366	225	＞500	350	300
SPECfp 92	283	300	＞750	550	600
SPECint 95	8.09	225	＞11	N/A	7.4
SPECfp 95	6.70	300	＞17	N/A	15
其它特性	CISC/RISC 混合	短流水线 长 L1 高速缓存	最高时钟频率 最大片上 2 级 高速缓存	多媒体和 图形指令	MP 机群总线 可支持 4 CPU

表 2.8　ASCI 计划的 3 个高端 MPP 系统特性一览表

特性	Intel/SNL Option Red	IBM/LLNL Blue Pacific	SGI/LANL Blue Mountain
处理器	Pentium Pro 200 MHz 200 Mflops	PowerPC604 332 MHz 664 Mflops	MIPS 10000 250 MHz 500 Mflops
系统结构	NORMA-MPP	SMP-Cluster	CC-NUMA
处理器数	9 216	5 856	6 144
峰值速度（$T=10^{12}$）	1.8 Tflops	3.88 Tflops	3.072 Tflops
存储容量	594 GB	2.5 TB	1.5 TB
磁盘容量	1 TB	75 TB	75 TB
推出时间	1997.6	1998.12	1998.12

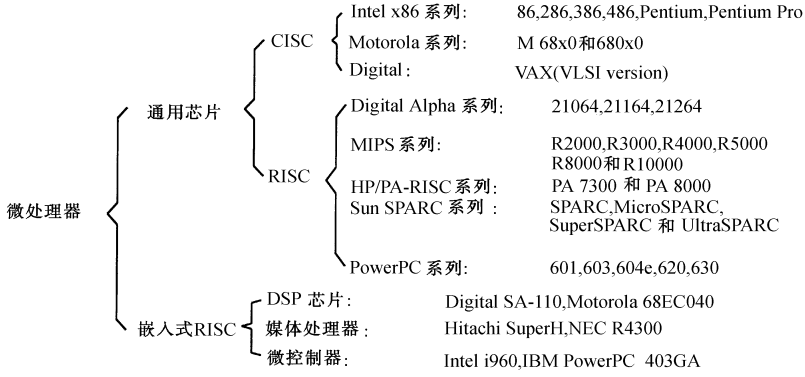


图 2.8 微处理器系列及其代表性 CPU 芯片

*2.2.2 ASCI Option Red MPP 系统

Option Red 系由 Intel 和 Sandia 联合开发,该 MPP 系统 1996 年 12 月移交给 Sandia 国家实验室,但完整的配置在 1997 年 6 月完成。

Option Red 的体系结构 Option Red 是一个如图 2.9 所示的分布式存储 MPP 系统,它总共有 4 608 个节点 (每个节点有两个 200 MHz Pentium Pro 处理

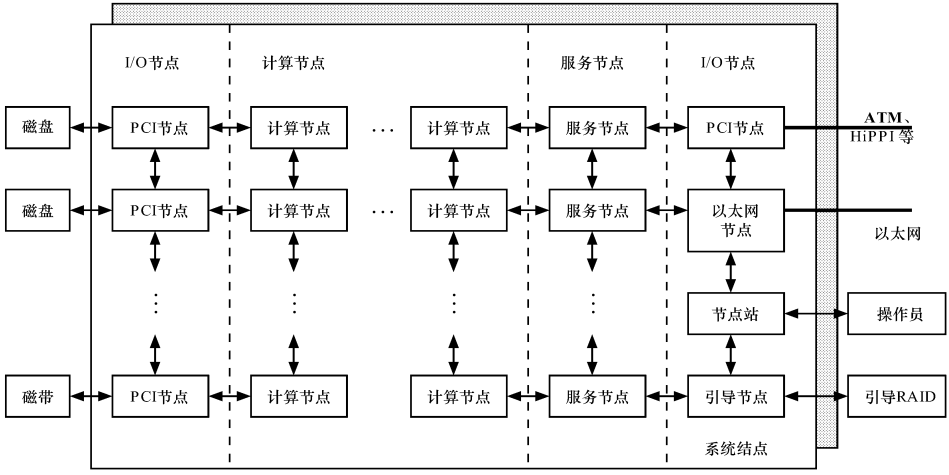
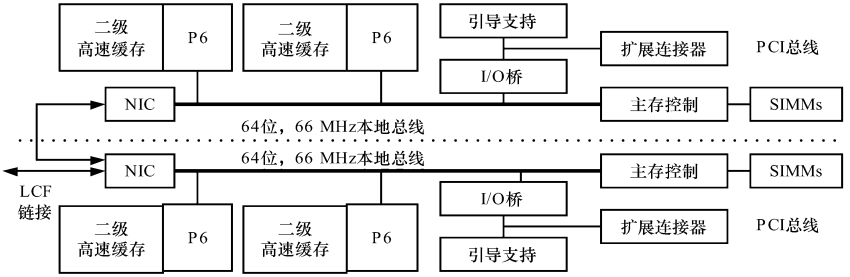


图 2.9 ASCI Option Red 系统框图

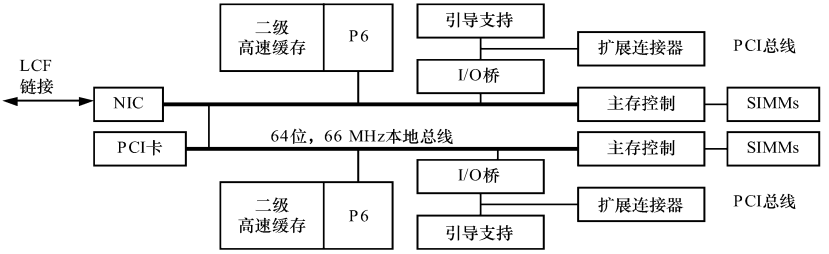
器）和 594 GB 的主存，其峰值速度为 1.8 Tflop/s、峰值截面（Cross Section）带宽为 51 GB/s。在这些节点中，计算节点（Compute Node）4 536 个，服务节点（Service Node）32 个，I/O 节点（I/O Node）24 个，系统节点（System Node）2 个，其余是备份节点。系统有 1 540 个供给电源，616 个互连底板和 640 个磁盘（大于 1 TB）。

（1）节点体系结构：计算节点用于执行并行计算，服务节点用于支持登录、软件开发及其他交互操作，I/O 节点用于存取磁盘、磁带、网络（以太网、FDDI、ATM 等）和其他 I/O 设备。另有两个系统节点用于支持系统 RAS（Reliability, Availability, Serviceability）能力：其中引导节点（Boot Node）负责初始系统引导及提供服务；节点站（Node Station）用于单一系统映像支持。

计算节点和服务节点的实现相同，如图 2.10（a）所示，两个节点在一块主板上。两个 SMP 节点通过 NIC 相连在一起，只有一个 NIC 连向互连底板。



(a) 作计算和服务节点使用的双节点板



(b) 作 I/O 和系统节点使用的单节点板

图 2.10　用于计算、I/O 与服务的 Option Red 节点主板

I/O 和系统节点的主板（图 2.10（b））与双节点主板（图 2.10（a））相似。然而，此处只有 2 个处理器（1 个节点）、1 个本地单总线和 1 个单 NIC。每个节点的主存容量可从 32 MB 到 256 MB 扩充至 64 MB 到 1 GB。133 MB/s 的 PCI 卡数量可从 2 扩充到 3。每个 I/O 节点主板同样有可通过前方控制板进行存取的板上基本

I/O 设备,如 RS-232、以太网 (10 Mb/s) 和 Fast Wide SCSI。

(2) 系统互连:节点由一个内部互连设备 ICF (Inter-Connection Facility) 相连,它使用了如图 2.11 所示的双平面 (Two Plane) 网孔拓扑。每个节点主板通过主板上的 NIC 连至一个定制的专用集成电路 ASIC (Application-Specific Integrated Circuit),它称为网孔选路部件 MRC (Mesh Routing Component)。如图 2.11 所示,MRC 有 6 个双向端口,每个能以 400 MB/s 的单向峰值速度传送数据,全双工时为 800 MB/s,4 个端口用于平面内左、右、上、下的网孔互连,还有一个端口用于平面间互连。从任意节点发出的消息借助虫蚀选路通过任一平面可送至另一节点,从而降低了时延,提高了系统的可用性。

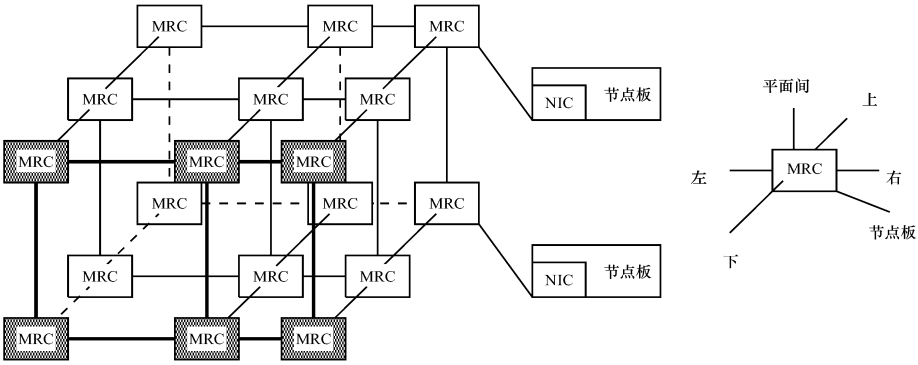


图 2.11 Option Red 互连结构

Option Red 的系统软件 ASCI Option Red 系统软件是 Paragon 环境的演变。系统、服务和 I/O 节点都运行 Paragon 操作系统,它是一个基于 OSF 的分布式 UNIX 系统。诸计算节点运行一个称为 Cougar 的轻量级内核 LWK (Light-Weight Kernel)。同时提供了对这两个系统间接口的支持,包括高速通信、UNIX 编程接口和一个并行文件系统。

(1) 轻量级内核:轻量级内核操作系统源于 PUMA 系统,它具有如下四个设计特点:① LWK 设计更强调性能,它能有效支持多达几千个节点的 MPP,只提供并行计算所需的功能,而不是一般的操作系统服务;② 由于 TFLOPS 系统中有几千个计算节点,Cougar 被设计成主存占用量在 0.5 MB 以下,以阻止 LWK 使用的聚集主存上升过快;③ 设计中假设通信网络是可信的并由内核控制,不需要保护检查和消息鉴别;④ LWK 提供一个开放的体系结构,允许用户层库例程的高效开发。

如图 2.12 所示,LWK 包括进程控制线程 PCT (Process Control Thread) 和精华内核 Q Kernel (Quintessential Kernel) 两层。每个节点有一些用户进程,一个 PCT 和一个 Q kernel。Q Kernel 是惟一可以直接访问地址映射和通信硬件的软件。它提供了基本的计算、通信和地址空间保护功能。PCT 提供进程管理、命名服务和组保护功能。

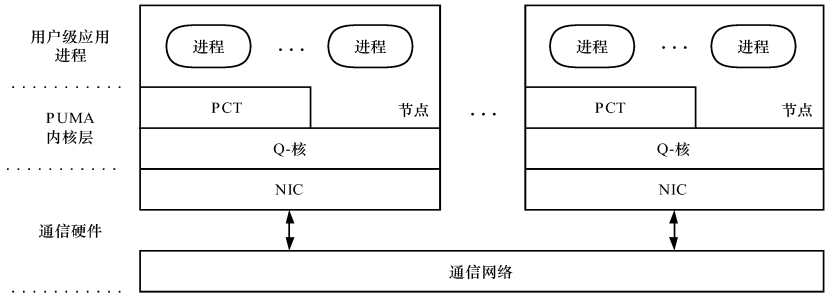


图 2.12 LWK (Cougar) 环境的层次结构

LWK 环境假设了一种信任的偏序关系:每个部件信任通信硬件能提供正确和安全的通信,即硬件将消息能可靠地送到正确的物理节点。消息在传输中不会被破坏,或被送至错误的节点。Q Kernel 信任硬件和其他节点中的 Q Kernel,但不信任 PCT 或应用进程;PCT 信任 Q Kernel 和其他 PCT,但不信任进程;应用进程信任 Q kernel 和 PCT,但不信任其他进程。LWK 体系结构确保一层中的数据结构只能被同层或更为信任的层次破坏,这可通过建造一系列的保护域 (Protection Domain) 来达到。

Q Kernel 域包括一个节点中的所有物理资源。PCT 地址空间形成一个子域,它又包括每个进程一个的若干子域,每个实体仅能直接访问它的域,这样一个进程就不会破坏 PCT,PCT 也不会破坏 Q Kernel。

将内核分为两层有几个优点:首先它提高了可移植性,因为对于不同的 MPP,大多数的 Q Kernel 代码需要重写,而大部分 PCT 代码是可移植的;其次分层可将功能分开,即 Q Kernel 负责控制对物理资源的访问,而 PCT 负责管理这种访问。

PUMA 的设计者没有正式定义控制和管理。一般而言,管理指完成什么样的和如何完成一个任务;而控制指上述决策的实际操作和执行。例如,选择时间片 (量程) 的大小,决定哪个进程是一个通信的目的地,以及选择一个进程调度策略都是管理操作;而执行量程,检查消息目的地是否有效,以及执行一个调度策略则是控制操作。

PUMA 的设计者相信控制操作远比管理操作发生频繁,而管理策略的改变却比控制机制来得频繁。将 LWK 分为两个层次有益于优化控制操作而不影响管理策略,并能在同样的 Q Kernel 上运行不同的如单任务或多任务的 PCT。

(2) 消息传递:ASCI Option Red 系统支持 MPI、NX 和消息传递入口,其中 MPI 是系统中的标准库,而 NX 是为了提供对 Paragon 的向后兼容,因为在 Paragon 上许多应用使用 NX 消息传递库。消息传递入口(Portal)提供了最为有效的低层消息传递库,入口的概念是在 PUMA 操作系统中首先提出的,它的使用可以降低消息传递中的存储器拷贝开销。然而,使用入口的消息传递不属于用户层通信机制,仍必须跨越内核。入口是目的进程地址空间的一部分,该部分向其他进程开放以发送消息。为发送一条消息,发送进程需执行如下的核心例程:

```
send_user_msg{
    void *buf          /* 发送消息缓冲区起始点 */
    size_t len         /* 发送消息的大小 */
    int tag            /* 消息标记 */
    proc_id dest       /* 目的进程号 */
    portal_id portal   /* 目的入口的索引 */
    int *flag          /* 消息发送的增量标记 */
}
```

这是一个无阻塞调用,当内核记录了消息传递所需的信息后立即返回。源内核向目的内核发送一个包括发送进程 ID、目的进程 ID、消息长度和标记及目的入口的消息头部,但不包括消息缓冲区地址和增量标记。当头部到达时,目的内核施行解释,并检验消息能否被存入入口,然后它将消息体传递至目的入口。当所有消息存入目的入口后,目的内核就对入口描述符中的一位置 1,以指明有一条新的消息已到达。一旦消息被发送,源内核就将增量标记的值加 1。发送进程可以轮询增量标记,以决定何时能安全地重用发送消息缓冲区。不需要调用显式的例程来接收消息,接收进程可以轮询入口描述符来发现是否有新消息到达。进程也可以采用生成信号的方式来表明有消息到达了它的任一入口。

2.3 机群系统

机群(Cluster)一词在近代并行机体系结构中广泛使用,其含义有时不尽相同。本节打算介绍两种类型的机群系统:一是构筑高端大规模并行处理系统 MPP 机群(如 IBM 的 SP2 等);二是由 LAN 互连而成的工作站机群 COW(如 Berkeley 的 NOW 等)。

2.3.1 大规模并行处理系统 MPP 机群 SP2

1991 年秋，IBM 决定涉足于 MPP 的研究，开动了 SP (Scalable POWERparallel) 计划。1992 年 2 月开始组队，1993 年 4 月就公布了第一个产品 SP1，继之于 1994 年 7 月就宣布了 SP2。IBM 的 SP 是比较特殊的，它采用了机群的办法来构筑 MPP。到 1998 年之前，其在世界上的总装机量超过 3 000，实属 MPP 系统成功之例。

设计目标和策略 IBM 设计 SP 系统时提出了如下目标：①赶市场：遵循着 Moore 定律，为夺性能/价格之冠，产品必须在短期内开发成功；②通用：SP 必须是个能支持不同的技术和商业应用、流行的编程模式和不同操作模式的通用系统；③高性能：SP 必须提供整体性能，不仅是处理器速度快，而且存储器和通信系统要快，有优良的编译器和各种库等；④有效性：SP 必须呈现好的可靠性和可用性，使得用户能够方便地在其上运行商业成品代码。为了满足上述目的，IBM 设计小分队采用的策略是：灵活的机群结构；专用互连网络；标准的系统环境；标准的编程模式和有选择的单一系统映像支持。

(1) 机群结构：为了达到赶市场和通用的目的，选用机群结构是个关键，其中每个节点都是个 RS/6000 工作站且各有本地磁盘；每个节点内驻留一个完整的 AIX (IBM 的 UNIX)；各节点经其 I/O 总线 (非本地存储总线) 连向专门设计的多级高速网络。SP 系列尽量使用标准的工作站组件，只有不能满足要求时才使用专用的硬件和软件。这样的结构既简单又灵活且系统的规模是可扩放的 (从很少的几个节点到数百个节点)。

(2) 标准环境：SP 使用标准的、开放式的、分布式 UNIX 环境，它能利用现存的标准软件以进行系统管理、作业管理、存储管理等，所有这些软件 IBM 工作站 AIX 操作系统中均有。对于那些 AIX 环境不能有效执行的应用，SP 提供了一组高性能服务，诸如高性能开关 HPS (High Performance Switch)、用户级通信协议 (US 协议)、优化的消息传递库 MPL (Message Passing Library)、并行程序开发和执行环境、并行文件系统、并行数据库和高性能 I/O 子系统等。

(3) 标准编程模式：SP 系统以标准编程模式支持以下三种应用方式：①串行计算：尽管 SP2 是个并行机，但它允许现有的以 C、C++ 和 Fortran 编写的串行程序可不加修改地运行在单节点的 SP 系统上，这是可以理解的，因为机群结构和标准的环境确保了这一点；②并行科技计算：SP 现在支持 MPL、MPI、PVM、HPF 模式，正打算支持共享存储的模式；③并行商用计算：IBM 正在并行化一些关键数据库、事务监视子系统，现今 IBM DB2 数据库系统的并行版本已在 SP2 上实现。

⑷ 系统可用性：SP系统由上千个部件组成，它们原先是为低价的、规模不大的工作站设计的，现把它们组织在一起必然经常失效。但SP是个机群结构，而机群结构意味着是一个分开的操作系统映像，它和SMP结构驻留在共享存储器中的单一操作系统映像不同（它的OS出错将导致全系统崩溃），机群结构一个节点映像失效不会导致全系统崩溃；另外SP的诸节点均同时连向以太网和高性能开关网，这样一个网络的失效，节点间还可使用另一个网络进行通信；还有SP的软件基础设施也提供了故障检测、诊断、系统重组和故障恢复等服务。

⑸ 部分单一系统映像：在一个分布系统中，用户看到的是一些单独的、分开的工作站，真正的单一系统映像是很难实现的，且对某些商业应用它也不是个关键的要求。所以IBM的设计者们，只是在单进入点、单文件层、单控制点和单作业管理系统方面实现了单一系统映像，而在SP系统中并不实现单地址空间。

系统结构　SP系统简化框图如图2.13所示。一个SP系统可含2～512个节点，每个节点有其自己的局存和本地磁盘。所有的节点均连向两个网络：普通的以太网和高性能开关。以太网虽慢但有很多好处：当高性能开关失效时，它可作为后援；当高性能开关正被开发或改进时，仍可利用以太网查错、测试和维持系统运行；此外以太网也可用来系统监视、引导、加载和管理等。

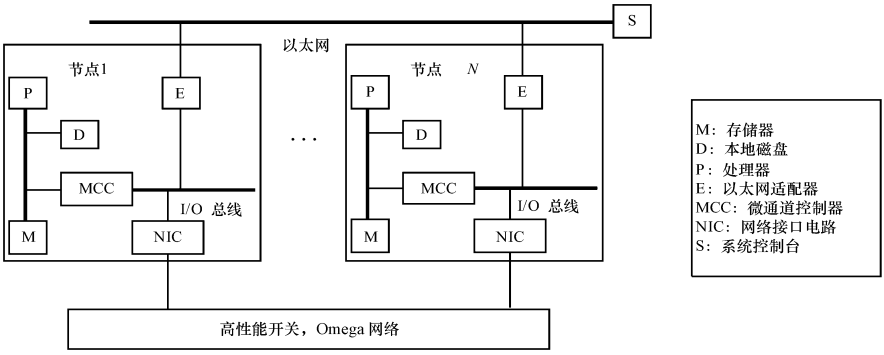


图2.13　SP系统结构

⑴ 系统互连：高性能开关（HPS）由节点内的开关硬件和开关帧（Switch Frames）组成。图2.14示出了IBMSP2中所使用的128路高性能开关，其中每一帧由一个16路开关板所连接的16个处理节点（N_0～N_{15}）所组成，8个帧再用一个附加级开关板连接起来（图中细线代表8位的双向链路，而粗线代表4个8位的双向链路），每一开关板上有两级开关芯片，所以此多级互连网络（MIN）总共有4个开关板。HPS是一个使用此开关的由40 MHz时钟驱动的带缓冲的多级Ω互连

网络。它使用了虫蚀选路法,一个 8 位的片 (Flits) 在无竞争时穿过一级 (即一个开关芯片) 只需 5 个时钟 (即 125 ns) 。因此 HPS 无竞争时的硬件延迟是很小的,对于 512 个节点仅875 ns。但实际延迟比此值高得多,一个进程发送一个空包给另一个进程至少花40 μs,这种消息传递延迟大部分是由软件开销造成的。HPS 能提供成对节点之间的双向传输带宽为 40 MB/s。

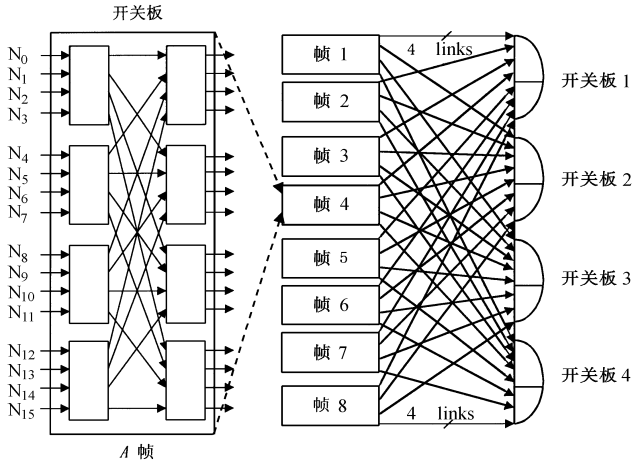


图 2.14　SP2 中的 128 路高性能开关 (HPS)

(2) 节点结构:SP2 有三种不同的节点,分别是宽节点 (Wide Node) 、窄节点 (Thin Node) 和窄节点 1,它们主要差别在于存储器的容量、数据路径宽度和 I/O 总线的槽数的不同,但是所有的这些节点都使用时钟为 66.7 MHz 的 POWER (Performance Optimized With Enhanced RISC) -2 微处理器。每个处理器有一个 32KB 的指令高速缓存、256KB 的数据高速缓存、指令和转移控制单元、两个定点运算单元、两个各能执行乘-加操作的浮点运算单元。由于定点和浮点运算可同时进行,所以 POWER-2 具有 4×66.7 Mflops=267 Mflops 的峰值速度。POWER-2 是个超标量处理器,它使用短指令流水线、先进的转移预测技术和寄存器重命名技术,使得它在每个时钟周期内能执行 6 条指令:两条取/存指令、两条浮点乘-加指令、一条变址增一指令和一条条件转移指令。

I/O 子系统和网络接口　SP 的 I/O 子系统如图 2.15 所示,它基本上是围绕着 HPS 构筑起来的,并用 LAN 的信关与 SP 系统以外的机器相连。SP 的节点有 4 类:主机节点 (H) 用于用户登录和交互处理;I/O 节点主要执行 I/O 功能 (如全局文件服务) ;信关节点 (G) 用于连网;计算节点 (C) 专负责计算。

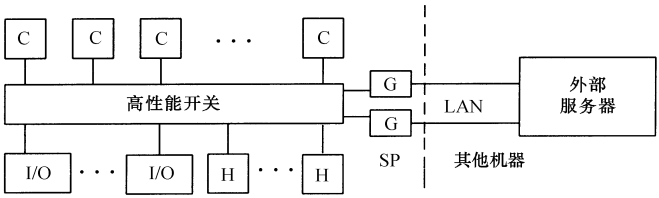


图 2.15 SP I/O 子系统

每个 SP 节点通过网络接口电路 (NIC) 与 HPS 相连,NIC 也叫作开关适配器或通信适配器。如图 2.16 所示,适配器包含一个 8 MB 的 DRAM 和受控于一个 40 MHz的 i860 微处理器。适配器经微通道接口搭在微通道 (Micro Channel) 上,它是一个标准的 I/O 总线并用于将外设连向 RS/6000 工作站和 IBM PC 机,同时适配器也经过存储和开关管理单元 MSMU (Memory and Switch Management Unit) 连向 HPS (经由各为 8 位宽的 IN-FIFO 和 OUT-FIFO)。除此之外,它还包含一些控制/状态寄存器和用作 i860 总线控制器,检查和刷新 DRAM。另外,一个 4 KB 的双向 FIFO (BIDI) 缓冲器用于连接微通道和 i860 总线。

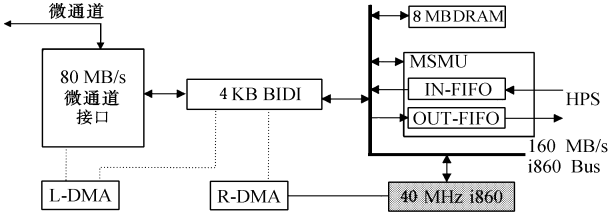


图 2.16 SP 通信适配器

参照图 2.16 来解释一下数据从节点发往 HPS 的过程:当节点处理器告诉适配器要发送数据时,i860 将直接存储访问 DMA (Direct Memory Access) 传输所必需的信息 (称为 Header) 写入 BIDI,当此 Header 抵达 BIDI 之首部时,左部 DMA (L-DMA) 负责将数据从节点 (微通道) 传入 BIDI;完成时,L-DMA 将硬件计数器增一,i860 写另一个 Header 至右部 DMA (R-DMA),R-DMA 负责将数据从 BIDI 传至 MSMU 中的 OUT-FIFO,然后再将数据传送至 HPS。

从 HPS 接收数据是类似的:当数据到达时,MSMU 通知 i860,它就写一个 Header 以启动 R-DMA,R-DMA 就负责将数据从 IN-FIFO 传至 BIDI;完成时,i860 向 BIDI 写入一个 Header,当它抵达 BIDI 之首部时,L-DMA 抽取此 Header,并负责将数据从 BIDI 传至微通道。

系统软件 SP 系统软件层次如图 2.17 所示,其核心部分是 IBM AIX 操作系

统。SP 沿用了绝大部分 RS/6000 工作站环境,包括数据库管理系统 (如 DB2),在线事务处理监视器 (如 CICS/6000),系统管理和作业管理,标准操作系统 AIX,Fortran、C、C++ 编译器,数学和工程库 (如 ESSL) 以及上万个串行应用程序等。SP 系统只加入了若干新软件和改进了某些现存的软件,它们都是可扩放并行机群系统所要求的。

<table>
<tr><td colspan="4">应　　用</td></tr>
<tr><td colspan="4">应用子系统 (数据库、事务处理监视器等)</td></tr>
<tr><td>系统管理</td><td>作业管理</td><td>并行环境</td><td>编译器等</td></tr>
<tr><td colspan="4">全局服务 (提供单一系统映像)</td></tr>
<tr><td colspan="4">有效性服务</td></tr>
<tr><td colspan="2">高性能服务</td><td colspan="2">标准操作系统 (AIX)</td></tr>
<tr><td colspan="4">标准 RS/6000 硬件 (处理器、存储器、I/O 设备、适配器)</td></tr>
</table>

图 2.17　SP 系统软件层次

(1) 并行环境:AIX 并行环境 PE (Parallel Environment) 为用户提供了开发和执行并行程序的平台。如图 2.18 所示,它包含 4 部分:并行操作环境 POE (Parallel Operating Environment),消息传递库 MPL (Message Passing Library),可视化工具 VT (Visualization Tool) 和并行调试器 Pdbx (Parallel Debugger)。其中 POE 用于控制并行程序的执行,它是由一个运行在家用节点 (是一个连向 SP 节点的 RS/6000 工作站) 上的划分管理程序 (Partition Manager) 来控制。家用节点是用户调用并行程序的地方,并行程序作为 SP 计算节点上一个或多个任务来运行。家用节点提供标准的 Unix I/O 设备 (如 Stdin、Stdout 和 Stderr),它通过 LAN (如以太网) 与计算节点进行标准的 I/O 通信。例如用户从家用节点的键盘上按 ctrl+C 键可终止所有的任务,用 Printf 语句就可在家用节点的屏幕上显示输出。消

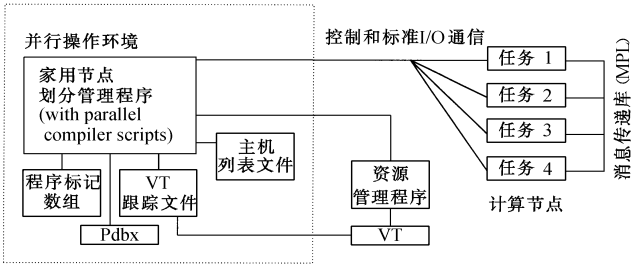


图 2.18　SP 机群的并行环境 (PE)

息传递通信是经由 HPS 或以太网执行专门 MPL 功能实现的,这个库提供诸如进程管理、点到点通信、整体通信等 33 种功能,IBM SP 还支持 MPI 的不同版本。

(2) 高性能服务:IBM SP 除了能直接使用标准的、商用的原来为 RS/6000 工作站和基于 TCP/IP 网络的分布系统所开发的软件外,它还提供了一些高性能服务,包括高性能通信子系统、高性能文件系统、并行库、并行数据库和高性能 I/O 等。

SP 支持两种通信协议:基于 IP 的协议(执行在核空间)和 US 协议(执行在用户空间),两者均可在 HPS 上或常规网(如以太网)上使用。但 US 协议具有较好的性能,可它每个节点中只允许有一个任务去使用 US 协议;当每个节点上有多个任务时,使用基于 IP 的协议可导致较好的整个系统的利用率。

(3) 并行 I/O 文件系统:SP 高性能文件系统称之为并行 I/O 文件系统 PIOFS (Parallel I/O File System),对绝大多数应用和系统实用程序它是与 POSIX 一致的。Unix 操作和命令(如 read、write、open、close、ls、cp 和 mv 等)与顺序 Unix 系统中的一样,除了允许传统的 Unix 文件系统接口外,PIOFS 提供了并行接口以便能对文件进行并行分布和操作。

IBM 开发了称之为 DB2 并行版本的并行数据库软件程序,它能运行在 SP 和其它机群平台上。数据库分布在多个节点中,数据库功能则装入数据驻留的节点上。DB2 并行版本在机器规模和问题规模两方面都是可扩放的,它能运行在数百个节点上并能处理多达万亿字节(Terabytes)的大型数据库。

(4) 有效性服务:SP 系统由一组运行在节点上的守护程序(Daemon)提供软件有效性基础设施。心跳(Heartbeat)守护程序周期地改变心跳信息以指示哪些节点是存在的。属籍(Membership)服务能识别节点和进程属于某一组。当属籍关系因节点失效、停机或再启动改变时,通告(Notification)服务用来通知活动的成员,并随后调用恢复(Recovery)服务协调恢复以使活动的成员继续工作。

(5) 全局服务:SP 系统提供的全局服务有外部系统数据储存库 SDR (System Data Repository),它维持有关节点、开关和现行作业等全系统的信息。当部分系统失效时 SDR 对重组系统是有用的,其内容能将系统带回失效前的状态。

采用通过 HPS 支持 TCP/IP 和 UDP/IP 可实现全局网络访问。通过网络文件系统 NFS (Network File System)可提供单一文件系统。除了 NFS 外,SP 还为全局磁盘访问提供虚拟共享磁盘 VSD (Virtual Shared Disk)技术。VSD 是位于 AIX 逻辑盘组管理器 LVM (Logical Volume Manager)之顶的一个设备驱动层。当一个节点进程欲访问本地共享磁盘时,VSD 直接传递请求至节点的 LVM;当一个节点进程欲访问远程共享磁盘时,VSD 传递请求至远程磁盘的 VSD,然后再将其传至远程节点的 LVM。

(6) 系统管理：SP 系统控制台 (S) 是一台控制工作站 (见图 2.13)。SP 系统管理器从此单控制点管理整个 SP 系统：包括系统安装、监视和配置、系统操作、用户管理、文件管理、作业计费、打印和邮件服务等。此外，SP 中的每个节点、开关等都有一块能自动检测环境条件的监视卡以及时控制硬件部件。管理者能使用这些设施开/关电源和监视器，复位单节点和开关等部件。还有，SP 支持用户交互和批处理两种作业模式，它们既可以是串行程序也可以是并行程序。

2.3.2　工作站机群 COW

工作站机群 COW (Cluster of Workstations) 是实现并行计算的一种新主流技术，是属于分布式存储的 MIMD 并行计算机结构，系由工作站和互连网络两部分组成。由于这种结构用于并行计算的主要资源是工作站，所以工作站机群的名称便由此产生。工作站机群 COW 这一名称，在早期的研究阶段，也曾被称为工作站网络 NOW (Network of Workstations)。本节着重讨论 COW 的基本原理及其有关问题；下一节介绍一下 Berkeley 大学的 NOW 研究计划。

什么是 COW　随着工作站性能迅速提高和价格日益下降以及高速网络产品陆续问世，一种新型的并行计算系统便应运而生。这种系统将一群工作站用某种结构的网络互连起来，充分利用各工作站的资源，统一调度、协调处理，以实现高效并行计算。图 2.19 示出了 COW 的硬件组成，它由工作站和互连网络两部分组成，工作站上增加一块主机接口板以实现连网。互连网络可以是普通的 LAN (如以太网、令牌环和 FDDI 等)，也可以是高速开关网络 (如 ATM、交换式高速以太网等)。工作站是个广义的称呼，它可以是高档微机，甚至也可以是个对称多处理机 SMP。一个实用的 COW 还应有一个高效的软件环境，如图 2.20 所示，包括操作系统 (可为通用的 UNIX、AIX，也可为改进和修改的操作系统)、通信协议 (实现工作站到互连网络的数据通信服务)、可由用户调用的通信原语库以及并行程序设计环境与工具等。从用户、程序员和系统管理员的角度看，COW 相当于单一并行系统，感觉不到多个工作站的实际存在；从程序设计模式的角度看，它与 MPP 一样可

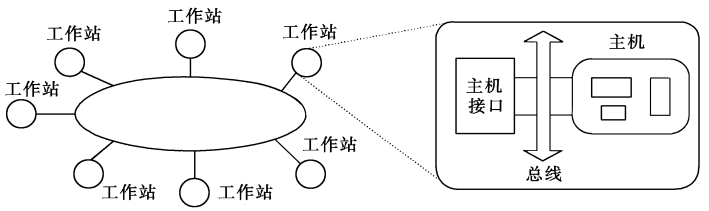


图 2.19　COW 的一般结构

采用面向消息传递的 SPMD (Single Program Multiple Data) 编程方式,即各个工作站均运行同一个程序,但分别加载不同的数据,从而可支持粗粒度的并行应用程序。

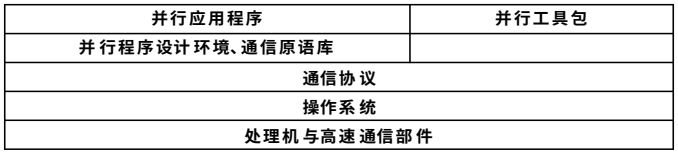


图 2.20 COW 的软件结构

为什么发展 COW 和前面所介绍的对称多处理机 SMP 和大规模并行机 MPP 相比,COW 在实用上具有一些明显的优点:①投资风险小:用户在购置传统巨型机或 MPP 系统时,总是担心使用效率不高和性能发挥得不好,如果购置后在一定程度上确实出现此问题,就相当于搁置或浪费了大批资金,但 COW 不存在此问题,因为即使 COW 在技术上不够先进,但每台高性能的工作站仍可照旧使用,不会浪费资金;②编程方便:用户无需学用新的并行程序设计语言 (如并行 C、并行 C++、并行 Fortran 等),只要利用所提供的并行程序设计环境,在常规 C、C++和 Fortran 等程序中相应的地方插入少量的几条原语,即可使这些程序在 COW 上运行,这一点是最受用户欢迎的;③系统结构灵活:用户将不同性能的工作站使用不同的体系结构和各种互连网络构成同构或异构的工作站机群系统,从而可弥补单一体系结构适应面窄的弱点,可更充分地满足各类应用要求;④性能/价格比高:一般一台巨型机或 MPP 都很昂贵 (费用常以几百万元、几千万元计),而一台高性能工作站相对便宜 (费用仅以几万元或十几万元计),一个 COW 系统从浮点运算能力来看,虽然每台工作站只有几 Mflops 到几十 Mflops,但一群工作站的总体运算性能可高达 Gflops 的量级,能接近一些巨型机的性能,但价格却低了很多;⑤能充分利用分散的计算资源:当个人工作站处于空闲状态时,COW 可在空闲时间内给这些工作站加载并行计算任务,从而工作站资源可得到充分利用;⑥可扩放性好:用户可根据需要增加工作站的数目,以高带宽和低延迟的网络技术支持获得高的加速比,从而获得应用问题的高可扩放性。

COW 的关键技术 实现工作站机群需要解决的主要问题是通信性能和并行编程环境。因为组成 COW 的硬件环境中工作站的性能已相当高,且还在不断的提高,相对而言工作站性能不是关键问题;相反,负责数据通信的互连网络却是一项关键技术,因为 COW 系统中并行计算时各工作站之间需要经过互连网络交换数据,某些工作站上的程序所需要的数据也往往要通过网络获取或提交。如果网络通信延迟时间很长,再加上用于通信的软件开销 (此部分不可忽视) 可能就限制

了 COW 技术对某些问题的适应性。近几年来网络技术发展很快,ATM 给予 COW 技术强大的支持,其 156 Mb/s的传输速率以及高带宽的特性保证了数据通信的高速进行;另外 IEEE/ANSI P1596 互连标准 SCI (Scalable Coherence Interface) 可支持多达 64 K 个节点互连,并能实现二级高速缓存一致性协议,传输速率可以达到 1 Gb/s,从而可很好地支持工作站的互连。

伴随高速网络而产生的另一关键技术是工作站到网络的主机接口网络接口的设计,它应尽量保持网络的传输速度与主机数据收发速度相匹配,其中增加高速缓存或采用 DMA 是可供选择的两种技术。网络接口使工作站可以利用网络传输数据,但也增加了通信延迟,它占用工作站 CPU 时间,从而影响了 COW 的性能。一般,一次通信时间延迟可如图 2.21 所示,其中 D 为操作系统调度时间,A 用于分配缓冲,C 用于拷贝数据到系统缓冲区,S 为启动发送/接收时间,T 为链路传输时间,F 用于定位中断源,X 用于从接收队列获取数据。一般减少延迟的方法有:① 设计精简通信协议以减少数据移动的次数和协议处理时间;② 采用主动消息 (Active Message) 携带数据处理命令以重叠计算和通信;③ 定制通信处理单元以消除通信对工作站 CPU 的依赖。

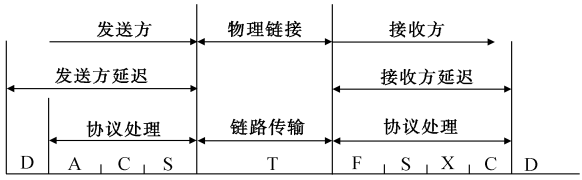


图 2.21　一次通信的时间延迟

另外,COW 要走向实用必须为用户提供一个良好使用环境和一套完善的工具系统,主要包括:① 并行编程环境,如一些通信原语库、多种编程语言 (Express、PVM、Linda、P_4、MPI) 的支持以及并行编译器等;② 可视化监视/调试器 (如 Express 的 domtool、基于 PVM 的 XPVM 等);③ 并行图形库 (如基于 Express 的 Plotix 等);④ 并行文件系统和数据库,以适应开展分布式事务处理的研究与开发。

COW 研究的新进展　随着 COW 研究的深入开展,很多大学和实验室都建立了各具特色的实验型 COW 系统。表 2.9 列出了几例具有代表性的机群研究计划,它在一定程度上反映了 COW 技术的新进展。其中,NOW 系统涉及到了机群计算的很多问题,包括结构、支持 Web 服务器的软件、单一系统映像、I/O 和文件系统、有效通信和强化的可用性 (我们在下一节专门讨论它);Princeton 的 SHRIMP 计划,集中研究在基于 PC 机的机群上,支持有效通信和共享存储,开发了专门的网络接口板,可以达到页面迁移级共享虚拟存储器;Wisconsin 的 Wind

Tunnel计划,正在调研如何用商用节点和网络构成的松散耦合机群实现高速缓存一致性共享存储器;Rice的TreadMarks是一个软件实现共享存储的工作站机群的优秀例子,它用用户空间运行时的库来实现存储器共享;由Chicago、Maryland和Pennsylvania共同发起的NSCP(National Scalable Cluster Project)计划,旨在研究异构机群上的元计算(Metacomputing),NSCP样机系统是由上述三所大学的局部网通过Internet互连起来而构成;Argonne的Globus计划也是研究元计算的,它将北美17个站点上的超级计算机、海量存储器和可视化设备用ATM网互连起来;Syracuse的WWVM(Word Wide Virtual Machine)计划,目的在于使用Internet和HPCC(High Performance Computing and Communication)技术实现高性能计算;Virginia的Legion计划,意在美国本土的国家虚拟计算机设施上开发元计算软件;香港大学的Pearl Cluster计划是一个用ATM连接的基于UNIX服务器和工作站(如SUN、HP、Alpha、SGI)的机群系统(参见图1.19),其优异特性包括:支持SSI的中间件(Middleware)、两种语言Internet搜索工具和Java接口的MPI通信子系统等,它们都用于财务数字库和分布式多媒体的应用。

表2.9 典型的机群系统特点一览表

名 称	系 统 特 点
Berkeley: NOW	非服务器工作站网络 主动消息,协同文件系统,全局Unix扩展
Princeton: SHRIMP	PC商用组件,通过专用网络接口达到 共享虚拟存储,支持有效通信
Karsruhe: Parastation	用于分布并行处理的有效通信网络 和软件开发
Rice: Tread Marks	软件实现的分布共享存储的 工作站机群
Wisconsin: Wind Tunnel	在经由商用网络互连的工作站机群上 实现分布式共享存储
Chica、Maryl、 Penns:NSCP	国家可扩放机群计划: 在通过因特网互连的3个本地机群系统上进行元计算
Argonne: Globus	在由ATM连接的北美17个站点的WAN上 开发元计算平台和软件
Syracuse: WWVM	使用因特网和HPCC技术,在世界范围的虚拟机上 进行高性能计算
HKU: Pearl Cluster	研究机群在分布式多媒体和金融数字库方面的应用
Virginia: Legion	在国内虚拟计算机设施上开发元计算软件

*2.3.3　Berkeley 的 NOW 计划

NOW (Network of Workstation) 由美国加州大学 Berkeley 分校开发，是一个颇有影响的计划，采用了很多先进技术，涉及到工作站机群很多共同问题，而且具有很多优异的特性：①采用商用千兆网络和主动消息通信协议支持有效的通信；②通过用户级整个机群软件 GLUnix 提供单一系统映像、资源管理和可用性；③开发了一种新的无服务器网络文件系统 xFS 以支持可扩放性和单一文件层次的高可用性；④最近 NOW 的开发者们正在开发一个软件框架 Webos 以构筑高可用性、渐增可扩放的地理性的 Web 服务器。

主动消息　主动消息 (Active Message) 是实现低开销通信的一种异步通信机制。其基本想法是在消息头部控制信息中携带一个用户级子例程 (称作消息处理程序) 的地址。当信息头到达目的节点时，调用消息处理程序从网络上抽取剩下的数据，并把它集成到正在进行的计算中。主动消息相当高效和灵活，以至于各种系统中都逐渐地以它为基本的通信机制。但在主动消息下，程序应是以 SPMD 方式设计的，因为允许发送者指定消息处理程序在接收方的地址，所以必须让所有的节点上有一致的代码映像。

在普通主动消息中提供了如图 2.22 所示的一组基本函数 (原语) 和实用函数。当发送请求消息 W 时，调用如下请求函数：

am_request_2 (Destination, request_handler, *x*, *y*)；

```
基本函数：
int am_request_M (vnn_t dest, request_handler, int arg0, …, int arg (M-1))
int am_reply_M (vnn_t dest, reply_handler, int arg0, …, int arg (M-1))
int am_get (vnn_t source, void* lva, void* rva, int nbytes,
            hander_t request_handler, void* handler_arg)
int am_store (vnn_t dest, void* rva, void* lva, int nbytes,
            hander_t reply_handler, void *handler_arg)
void am_poll (void)

实用函数：
int am_enable (…)      //initializes the active message layer//
int am_disable (void)  //exits the active message layer//
int am_procs (void)    //returns the total number of processes of program//
int am_my_proc (void)  //returns the calling process's virtual node number//
int am_max_size (void) //returns the max. No. of bytes for a get/store//
```

图 2.22　普通主动消息中的某些函数

其中请求消息 W 由 request_handler、两个整数 *x* 和 *y*、隐含的请求进程的虚拟节点号

组成。请求原语 am_request_M(··)有5个版本(*M*=0～4),每个均有 *M* 个整数变量。该请求发送 W 至 Destination(目的)进程。am_request 是阻塞式的,即直到 W 被发出才返回。当 W 到达时,Destination 进程调用 request_handler 子例程进行处理。

类似地,当发送应答消息 W 时,调用如下的应答函数:

am_reply_2(Destination,reply_handler,*x*,*y*);

其中应答消息 W 由 reply_handler、两个整数 *x* 和 *y*、隐含的应答进程的虚拟节点号组成。应答原语 am_reply_M(··)有5个版本(*M*=0～4),每个均有 *M* 个整数变量。该应答发送 W 至目的进程。am_reply 是阻塞式的,即直到 W 被发出才返回。当 W 到达时,目的进程调用 reply_handler 子例程进行处理。

am_store 和 am_get 两个函数是用来在两个进程间传送大块数据。其中,am_store 函数定义如下:

am_store(Dest,lva,rva,N,request_handler,handler_arg);它将请求进程中从地址 lva 开始的连续 *N* 个字节传至目的进程中从地址 rva 开始的存储区中。当收完所有 *N* 个字节后,目的进程(Dest)调用 request_handler,并传给它下述参数:请求进程的 VNN(Virtual Node Number)、rva、N 和 handler_arg。类似地,am_get 函数定义如下:

am_get(source,rva,lva,N,reply_handler,handler_arg);

它将应答进程中从地址 rva 开始连续的 *N* 个字节取至源进程中从地址 lva 开始的存储区中。当收完所有 *N* 个字节后,调用者调用 reply_handler 并传给它下述参数:源进程的 VNN、lva、N 和 handler_arg。

主动消息是个一般通用的通信机制,与具体硬件和软件平台无关,它已经在 MPP、COW 甚至 PVM 上予以实现,并取得了低延迟、高带宽优良性能,之所以如此是因为:①用户级底层功能:主动消息通信常可完全在用户空间实现,可消除上下文切换等所造成的开销,没有必要施行系统调用,用户可直接访问网络接口硬件,消息处理程序也就是普通的用户级子例程;②简捷:普通主动消息只有5条原语,每个只具有非常简单的功能和协议,且没有像 TCP 协议中的缓冲管理或误差检测等额外开销;③计算和通信重叠:一般的消息传递型软件(如 PVM 和 MPI)可通过非阻塞发送/接收操作支持计算和通信的重叠,但在发送和接收双方均要有消息缓冲。主动消息处理长、短消息是不同的,对于4个或更少字的短数据块,可使用阻塞式 am_request 进行发送,而对大块数据传输,则可使用 am_store 或 am_get。在普通主动消息中,提供了非阻塞的 am_store 以方便通信与计算的重叠。

GLUnix 机群需要能支持可用性和单一系统映像的新的操作系统功能,但传统的工作站 OS 均无此功能,为此需要对其进行扩充,其方法有二:微(内)核法与用户级守护程序和库方法。微核法(例如 Mach 和 Windows NT)就是将提供所有

服务的整体核代之以一些模块,并且绝大多数最基本的服务只是由少量的称之为微核(Microkernel)的小模块提供,其它的服务均在用户模式下提供。这种模块化法便于移植且灵活,但也有一些缺点:①原始的OS模块须重写;②在不同的结构上进行移植仍需付出努力;③微核系统比整体系统开销大(涉及到上下文切换、进程间通信和交叉保护边界等)。用户级守护程序和库方法能够作为用户级服务和库被实现,此法的优点是不要求修改核且易于实现,但同样它也有着和微核一样的额外开销。

NOW使用了GLUnix方法,它代表全局层(Global Layer) Unix,是指运行在工作站标准Unix之上的一个软件层,属于自包含软件。其主要的想法是机群操作系统应由低层和高层两层组成:其中低层是执行在核模式下的节点商用操作系统;高层是能提供机群所需的一些新功能的用户级操作系统。特别是,新全局层能提供机群内节点的单一系统映像,使得所有的处理器、存储器、网络容量和磁盘带宽均能被分配给串行和并行应用,并且它能够作为被保护的、用户级操作系统库予以实现。GLUnix方法能使很多机群性质在用户级实现,其优点是:①易于实现:GLUnix完全在用户级实现,不须修改核(第一个GLUnix原型只用3个月就完成了);②可移植性:因为它只依赖基本系统中很小一组标准性质,所以GLUnix能移植在任何支持进程间通信、进程信令(Signaling)和访问负载信息的操作系统上;③有效性:在应用地址空间内,使用过程调用的办法能够调用机群所需的一些新性质,无须跨硬件的保护边界、无须核自陷和上下文切换,在GLUnix中删除了系统调用开销;可以使用共享存储段或进程间通信原语,协调多个节点上的GLUnix的多个副本;④鲁棒性:因为GLUnix是在用户级,所以能够使用常规的查错工具对它进行测试,错误可被检测、诊断和删除。总之,GLUnix原型对下述的绝大部分特性都已提供:并行程序的共调度;空间资源的检测、进程的迁移和负载平衡;快速用户级通信;远程调页和可用性支持。

xFS 无服务器文件系统将文件服务的功能分布到机群的所有节点上,它和传统的中央文件服务器的对照见图2.23。传统的中央文件服务器执行的主要功能是:①中央存储:数据文件和元数据都存储在连向文件服务器的一个或多个稳定磁盘上。一个文件的元数据系由一组文件属性(例如文件类型、文件大小、设备ID、节点号、属主ID和文件访问许可等)所组成;②中央缓存:为了改进性能,每个客户可以缓存某些局部文件块,而文件服务器缓存其主存中经常使用的文件块;③中央管理:服务器执行所有文件系统管理功能,包括元数据和高速缓存一致性管理。在xFS中,所有的服务器和客户的功能由分散的所有节点实现之。xFS仅有的限制是,作为管理器也必须是个客户,因为管理器使用了客户接口。

(1) 廉价冗余磁盘阵列RAID(Redunant Array of Inexpensive Disks):无工作

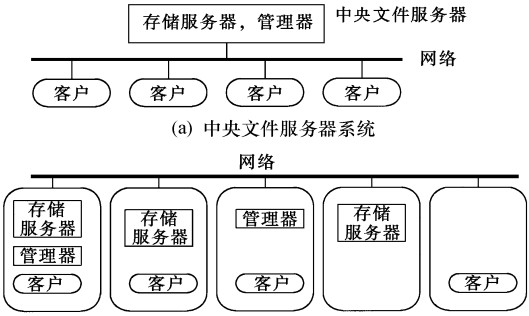


图 2.23 两类文件系统对照

站文件系统能用来生成软件 RAID 以提供高性能和高可用性。现今，xFS 使用单奇偶校验磁盘条 (Striping)。一个文件数据块在多个存储服务器节点上按条划分，在另一节点上有奇偶校验块。如果一个节点失效，失效磁盘的内容，可利用其余盘和奇偶盘之异或操作重建之。RAID 的缺点是所谓“小写问题”(Small Write Problem)：即如果一次写所修改的仅是条的一部分而不是整个条，则系统必须重新计算奇偶校验而导致大的开销。xFS 使用日志条 (Log based striping) 的方法解决此问题：每个客户首先将写接合到各用户的日志 (Log) 上 (它就是记录所有写操作的一个存储缓存器)；然后此登录采用日志段 (Log segment) 提交给磁盘，每个段系由 $K+1$ 日志片 (Log fragment) 组成，它与奇偶校验片一道送向 K 个存储服务器。但此法对大型多服务器机群也有问题：很多发往存储服务器的小片可能使登录存储器很大；很多客户同时写登录会造成竞争。xFS 采用将存储服务器分成一些称之为条组 (Stripe Group) 的子集的办法来解决此问题。如图 2.24 所示，8 个服务器分成 2 组，每组 4 个服务器。其中，A 条组中，第一段由片 1,2,3 组成，相应的奇偶校验片为 p；第二段由片 4,5,6 组成，相应的奇偶校验片 q。B 条组中情况类似。这样客户 1 和客户 2 可同时分别向各自组写入登录而不会冲突。

(2) 协同文件缓存 (Cooperative File Caching)：协同文件缓存的思想很简单：机群每个客户节点都辟出一部分作为文件 (高速) 缓存。协同文件缓存算法利用所有这些存储器生成一个大的、全机群上的文件缓存。当一个客户未找到其本地文件缓存时，不需要到磁盘中去找，只要到另一个客户的存储器中去找该数据。如图 2.25 所示，文件稳定地存在服务器磁盘中，习惯上这些文件也缓存至客户存储器中

器中。有两种协同文件缓存算法：贪心转发法 (Greedy Forwarding) 和 N-概率转发法 (N-Chance Forwarding)。贪心转发法工作原理是：一个访问文件和客户首先试

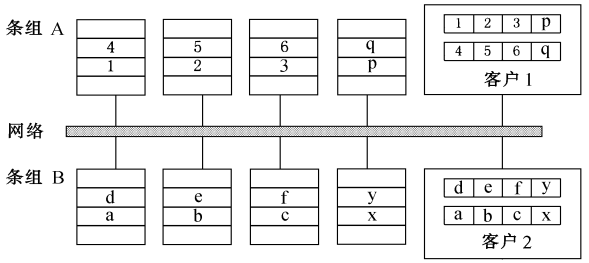


图 2.24　xFS 中的条组方法

用其本地高速缓存,如果数据块不在那里,它便将请求转发给服务器。服务器先搜索其本地高速缓存,若命中,便将数据返给客户并维护其高速缓存目录;若未找到,服务器便查阅其高速缓存目录并把请求转发给保存所请求数据的客户,然后该客户便直接将数据返回给请求的客户。贪心转发法的优点是实现简单,但其缺点是同一数据块可能被很多高速缓存所复制。当一客户第一次读一个数据块时,此块从服务器磁盘中读出并将其缓存在服务器高速缓存和客户高速缓存中,而另一读同一块的客户将把它缓存在其本地文件高速缓存中。这种重复降低了协同高速缓存的有效空间,因而也增加了命不中的比率,而且使高速缓存一致性管理更加困难。N-概率转发法是贪心转发法的一般化,它采用缓存一个数据块仅于一个客户高速缓存中的办法避免重复。这样的数据块叫作单块(Singlet)。当一个客户从另一客户的高速缓存中取出一块时,此块就被丢掉,并发一条消息给服务器告诉它此块已被移去。单块所带来的问题是,如果一个单块被丢掉,它就给后来的数据块腾出了位置,此单块也就从整个协同高速缓存中丢失。N-概率转发法可缓解此问题,只要客户高速缓存未满,它和贪心转发法完全一样。当任一客户的本地高速缓存满时,就要丢掉一个数据块,此时客户首先要检查一下此块是否为单块;如不是,客户就丢掉它;如是,客户就将块循环计数器置为 N,并发送该块给一个随机的客户。如果第二个客户稍后要丢掉这一块,它就将循环计数器减一,并发送该块给另一客户。此过程一直继续到循环计数器为零,这时就把该块丢掉。

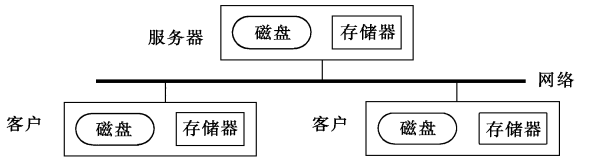


图 2.25　客户机/服务器机群中缓存文件的不同方式

(3) 分布式管理：xFS的文件系统管理功能是全分布的，它使用了位于多个节点上的多管理器(Manager)。为了理解如何达到分布管理，参照图2.26来看一下xFS中一个文件的读过程：当一个客户试图读一个文件块时，它发送一个带有文件(路径)名和位移(Name & Offset)的请求，在xFS的目录(Dir)中使用文件名客户就能找到文件的索引(Index)，用此索引客户搜索本地高速缓存(Unix Cache)，如果数据块在其中(命中)，则数据就被读出；如果高速缓存未命中，那么客户用索引号查阅管理器映射表(Manager Map)，此表指明哪个物理节点管理哪组索引号，也就是正确的管理者在哪里，于是索引和位移信息就经网络发给正确的管理者，管理者留意诸如数据块是否在磁盘中或被缓存、块的精确位置、高速缓存块是否是一致的某信息。如果块被缓存在某一客户的本地高速缓存中且是一致的，那么管理者就转发读请求给那个客户，客户就从其本地高速缓存中读取数据并直接发给原先的客户。如果数据块不在协同高速缓存中，管理者就再查阅imap表以找到索引节点(inode)，它包含了文件所有数据块的地址，此地址就可用来找到正确的存储服务器，于是数据块就可从那里读出并传给原先请求的客户。

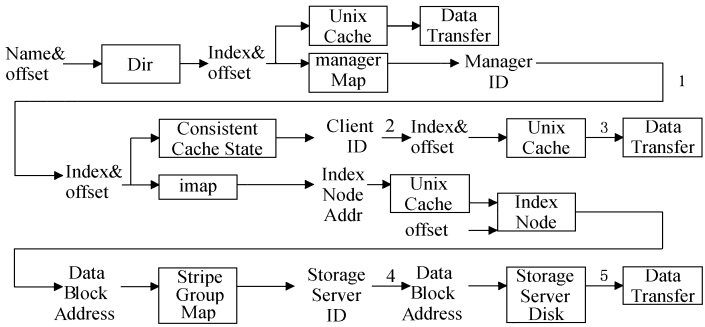


图2.26 xFS中一个简化文件的访问过程

一个xFS的原型已在32个节点的机群上实现，它展示了良好的可扩放性，其读带宽已达13.8 MB/s，而写带宽已达13.9 MB/s。

2.4 小结和导读

小结 本章讨论了当代共享存储多处理机、分布存储多计算机和机群三种并行计算机系统。SMP、MPP和COW的系统特性的比较综合于表2.10中。注意，MPP和COW之间的界线渐趋模糊，例如IBM SP2被视为MPP，但它具有机群结

构(除了使用高性能开关网络作为通信网络外),所以本章也将其放在机群系统中讨论之。COW 的性能/价格比优于 MPP,所以机群技术是发展可扩放并行计算的主流趋势。

表 2.10 SMP、MPP、COW 性能比较一览表

系统特性	SMP	MPP	COW
节点数目(N)	≤O(10)	O(100) ~O(1000)	≤O(100)
节点复杂程度	中/粗粒度	细/中粒度	中粒度
节点间通信	共享存储	消息传递/共享存储	消息传递
作业调度	单一运行队列	主机上单一运行队列	多队列但协同
支持单一系统映像	总是	部分	希望
节点 OS 副本和类型	一个(单一)	N(微核)和1个主机 OS(单一)	N(希望同构)
地址空间	单一	多个	多个
所有权	一个单位	一个单位	一个或多个单位
网络协议	非标准	非标准	标准/非标准
系统可用性	常常是低的	低/中	高或容错
性能/价格比	一般	一般	高
处理器类型	商用	商用	商用
互连网络	总线/交叉开关	定制	商用
系统存储器	集中共享	分布共享	分布共享
访存模型	UMA	NORMA	NORMA
代表机器	Intel SHV, Sun Fire, DEC8400, IBM R60, SGI Power Challenge 曙光1号	Intel Paragon, Intel/Sandia ASCI Option Red, 曙光-1000/2000	Berkeley NOW, Princeton SHRIMP, Rice Tread Marks, HKU Pearlduster

导读 有关当代三种并行机系统的全面介绍,读者可参阅[103,193],有关 Origin 2000 的介绍,读者可参阅[117,155,181];有关 Option Red 的介绍,读者可参阅[123];有关 SP2 的介绍,读者可参阅[169,163];有关 COW 的一般介绍,读者可参阅[203];Berkeley:NOW 计划描述在[15]中;Princeton:SHRIMP 描述在[53]中;Karsruhe:Parastation 描述在[180]中;Rice:TreadMarks 描述在[13]中;

Wisconsin:Wind Tunnel 描述在 [144] 中;NSCP 计划描述在 [42] 中;Argonne:Globus 描述在 [57] 中;Syracuse:WWVM 描述在 [60] 中;Virginia:Legion 描述在 [83] 中和香港大学:Pearlcluster 描述在 [102] 中。

习　题

2.1　对于一个 100 MHz 时钟频率的总线连接的 SMP 系统,假定总线宽度为 8 个字节,为了传输 16 个字节的数据,那么总线带宽可达多少?

2.2　试解释系统带宽与系统互连对剖宽度之间的关系,并举一例说明之。

2.3　参照图 2.6:

①试重画图 2.6(e),并补上快速链路;

②试重画图 2.6(f),并补上略去的节点;

③试画出 128 个节点(256 个处理器)的胖超立方;

④试画出 512 个节点(1 024 个处理器)的胖超立方。

2.4　下面是 NOW 中使用主动消息,两个进程施行短消息通信的过程,其中进程 Q 计算数组 A,进程 P 计算标量 x,最终的结果是 x 与 A[7] 之和。试解释远程读取的具体过程。

```
        进程 P                          进程 Q
    compute x                       compute A
        …                               …
    am_enable(…);                   am_enable(…);
    am_request_1(Q,request_h,7);    …
    am_poll();                      am_poll();
    sum=x+y                             …
        …                           am_disable();
    am_disable();
                                    int request_h(nnn_t P,int k)
    int reply_h(nnn_t,Q,int z)      {am_reply_1(P,reply_h,A[k]);}
        {y=z}
```

2.5　根据 xFS,试回答:

①xFS 与集中式文件服务器之间的主要区别是什么?

②为了增强可用性,xFS 使用了哪些主要技术?

2.6　根据 2.3.1 节有关 SP2 的介绍,试回答:

①SP 设计者为了赶上市场作了什么决策?

②SP 设计者为了达到系统通用采用了什么相应技术?

③SP 系统是如何支持 4 种 SSI 的:单一进入点、单一文件层次、单一控制点和单一作业管理系统?

④SP 设计者为了增加带宽,在通信子系统中主要使用了什么技术?

2.7 试解释 RAID (Redundant Array of Inexpensive Disks) 工作原理。

2.8 试比较 SMP,SSMP,CC-NUMA 和基于机群的 MPP 以及 DSM 在体系结构上的异同点。

第三章　并行计算性能评测

并行计算的性能评测与并行计算机体系结构、并行算法和并行程序设计一道构成了“并行计算”研究的四大分支。并行计算的性能评测，大致可分为机器级的性能评测、算法级的性能评测和程序级的性能评测。机器级的性能评测主要包括CPU和存储器的某些基本性能指标，并行通信开销以及机器的成本、价格和性/价比等；算法级的性能评测主要包括加速、效率和可扩放性等；程序级的性能评测主要包括基本测试程序、数学库测试程序和并行测试程序等。本章将对它们逐一做简要讨论。更详细的讨论请参阅文献[193]。

3.1 并行计算机的一些基本性能指标

本节将有选择地讨论并行机的某些性能指标,主要包括 CPU 和存储器的某些基本性能指标 通信开销以及机器的成本、价格与性/价比等。

3.1.1 CPU 和存储器的某些基本性能指标

表 3.1 列出了并行计算机的一些基本性能参数,下面将着重讨论 CPU 和存储器的某些基本性能指标,主要包括工作负载、并行执行时间、存储器层次结构及其典型性能参数、存储器带宽估计等。

表 3.1 并行机基本性能参数一览表

名 称	符 号	含 意	单 位
机器规模	n	处理器的数目	无量纲
时钟速率	f	时钟周期长度的倒数	MHz
工作负载	W	计算操作的数目	Mflop
顺序执行时间	T_1	程序在单处理机上的运行时间	s(秒)
并行执行时间	T_n	程序在并行机上的运行时间	s(秒)
速度	$R_n = W/T_n$	每秒百万次浮点运算	Mflops
加速	$S_n = T_1/T_n$	衡量并行机有多快	无量纲
效率	$E_n = S_n/n$	衡量处理器的利用率	无量纲
峰值速度	$R_{peak} = nR'_{peak}$	所有处理器峰值速度之积,R'_{peak}为一个处理器的峰值速度	Mflops
利用率	$U = R_n/R_{peak}$	可达速度与峰值速度之比	无量纲
通信延迟	t_0	传送 0-字节或单字的时间	μs
渐近带宽	r_∞	传送长消息通信速率	MB/s

工作负载(W) 所谓工作负载(荷),就是计算操作的数目,通常可用执行时间、所执行的指令数目和所完成的浮点运算数三个物理量来度量它。其中,① 执行时间:它可定义为在特定的计算机系统上的一个给定的应用所占用的总时间,系指应用程序从开始到结束所掠过时间(Elapsed Time),它不只是 CPU 时间,还包括

了访问存储器、磁盘、I/O 通道的时间和 OS 开销等。② 浮点运算数:对于大型科学与工程计算问题,使用所执行的浮点运算数目来表示工作负载是很自然的。对于程序中的其他类型的运算,可按如下经验规则折算成浮点运算(Flop)数:在运算表达式中的赋值操作、变址计算等均不单独考虑(即它们被折算成 0 Flop);单独赋值操作、加法、减法、乘法、比较、数据类型转换等运算均各折算成 1 Flop;除法和开平方运算各折算成 4 Flop;正(余)弦、指数类运算各折算成 8 Flop;其他类运算,可按其复杂程度,参照上述经验数据进行折算之。③ 指令数目:对于任何给定的应用,它所执行的指令条数就可视为工作负载,常以百万条指令为计算单位,与其相应的速度单位就是 MIPS(每秒百万条指令)。

并行执行时间 在无重叠操作的假定下,并行程序的执行时间 T_n 为:

$$T_n = T_{comput} + T_{paro} + T_{comm} \tag{3.1}$$

其中,T_{comput} 为计算时间,T_{paro} 为并行开销时间,T_{comm} 为相互通信时间。而 T_{paro} 包括进程管理(如进程生成、结束和切换等)时间,组操作(如进程组的生成与消亡等)时间和进程查寻(如询问进程的标志、等级、组标志和组大小等)时间;T_{comm} 包括同步(如路障、锁、临界区、事件等)时间,通信(如点到点通信、整体通信、读/写共享变量等)时间和聚合操作(如归约、前缀运算等)时间。

存储器的层次结构 在近代计算机中,为了加快处理器与存储器之间的数据移动,存储器通常按图 1.28 所示的层次结构进行组织。如图 3.1 所示,对于每一层均可用三个参数表征之:① 容量 C:表示各层的物理存储器件能保存多少字节的数据;② 延迟 L:表示读取各层物理器件中一个字所需的时间;③ 带宽 B:表示在 1 秒钟内各层的物理器件中能传送多少个字节。各层存储器及其相应的典型的 C、L、B 值示于图 3.1 中。

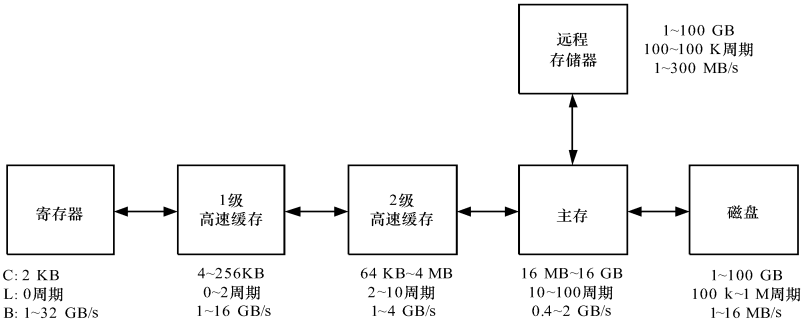


图 3.1 各层存储器的典型性能参数

存储器带宽的估算 假定字长为64位,即8字节。对于RISC类型机器中的加法操作,它从寄存器中取2个64位的字相加后再回送至寄存器。通常RISC加法指令可在单拍内完成,如果使用100 MHz的时钟,那么存储带宽将是$3\times8\times100\times10^6=2.4$ GB/s。可见,较快的时钟和处理器中较高的并行操作,可获得较宽的带宽。

3.1.2 通信开销

这一节主要讨论由于多进程相互作用所引起的通信开销 T_{comm}。这种开销比普通的计算时间要长得多,而且随系统不同而变化很大。

点到点通信开销的测量 对于点到点的通信,测量开销使用乒-乓方法(Ping Pong Scheme):节点0发送 m 个字节给节点1;节点1从节点0接收 m 个字节后,立即将消息发回节点0。总的时间除以2,即可得到点到点通信时间,也就是执行单一发送或接收操作的时间。用乒-乓方式测量延迟的代码段如下:

```
//*乒—乓法测量延迟的代码段*//
for i=0 to Runs-1 do/*发送者*/
    if (my_node_id =0) then
        temp =second()        /*second() 为时标函数*/
        start_time =second()
        send an m-byte message to node 1
        receive an m-byte message from node 1
        end_time = second()
        total_time = end_time-start_time
        communication_time = total_time/2
    else if (my_node_id = 1) then/*接收者*/
            receive an m-byte message from node 0
            send an m-byte message to node 0
        endif
    endif
endfor
```

乒-乓方法可一般化为热土豆法(Hot Potato),也称为救火队法(Fire Brigade):节点0发送 m 个字节至节点1,节点1再将其发送给节点2,依此类推,最后节点 $n-1$ 再将其返回给节点0,最后时间再除以 n 即可。

点到点通信开销的表达式 Hockney对于点到点的通信,给出了如下所示的通信开销 $t(m)$ 的解析表达式,它是消息长度 m(字节)的线性函数:

$$t(m) = t_0 + m/r_\infty \tag{3.2}$$

其中,t_0 是通信启动时间(μs);r_∞ 是渐近带宽(MB/s),表示传送无限长的消息时

的通信速率。Hockney 也同时引入了两个附加参数：半峰值长度 $m_{1/2}$（字节），表示达到一半渐近带宽（即 $\frac{1}{2}r_\infty$）所需要的消息长度；特定性能 π_0（MB/s），表示短消息带宽。4个参数 t_0、r_∞、$m_{1/2}$和 π_0 中只有两个是独立的，其他两个可使用如下关系式推导出：

$$t_0 = m_{1/2}/r_\infty = 1/\pi_0 \qquad (3.3)$$

3.1.3 机器的成本、价格与性能/价格比

从技术的角度看，计算机的价格并不能算是机器的性能，但对广大购置计算机的人员来说，却往往是首先要考虑的因素。而计算机的价格是怎样定的呢？显然与生产制造计算机的成本有关。所以，了解计算机的最后标价是怎样从原料的成本逐步加码而来是很必要的。此外，性能显然与价格也有关系，人们总是希望花费最少的钱而能购置性能最高的机器，这就是计算机的所谓性能/价格比问题。对于任何计算机设计与制造者而言，如何利用各种先进技术来达到高的性能/价格比总是基本的目标。

机器的成本与价格　价格和成本是两个不同但又相关的概念：成本并不代表用户购机的价格，它在变成实际价格之前是会经过一系列的变化；价格的上扬会使机器销售市场不景气，导致了产量的下降，从而使成本就会增大，而最终致使价格进一步上涨。一般而言，成本每变化 1 000 美元，价格将会变化 3 000～4 000 美元，且价格又是一个时间的函数。下面参照图 3.2 来说明从原料成本到最终标价的演变过程（以工作站为例）：① 原料成本：它是指一件产品中所有零部件的采购成本总和，是价格中最基本、明显的部分。② 直接成本：它是指与一件产品生产直接相关的成本，包括劳务成本、采购成本（运输、包装等）、剩余零头和产品质量成本（人员培训、生产过程管理等）等，直接成本通常在原料成本上增加 20%～40%。③ 毛利：它主要包括公司的研发费、市场建立费、销售费、生产设备维护费、房租、贷款利息、付税前利润和税务费等，原料成本、直接成本和毛利相加在一起就得到平均销售价格，而毛利一般占其 20%～55%。④ 折扣：它是产品在零售商店销售时，商店所获取的利润，它加上平均销售价格就是机器价目单的标价，而折扣通常占标价的 40%～50%，平均销售只能达到标价的 50%～75%。

在美国，大部分公司只将收入的 4%（微机产业）～12%（高端计算机产业）投入研发，它不会轻易随时间变化。并行机一般投入研发的资金会更大。由于并行机销售情况不如微机、工作站等，所以它的毛利就比较高，因此价格也就更高。并行计算机这种销售量不大而又需要很高的研发费用，就使得并行机的价格/成本比总是比微机、工作站、小型机等要高。

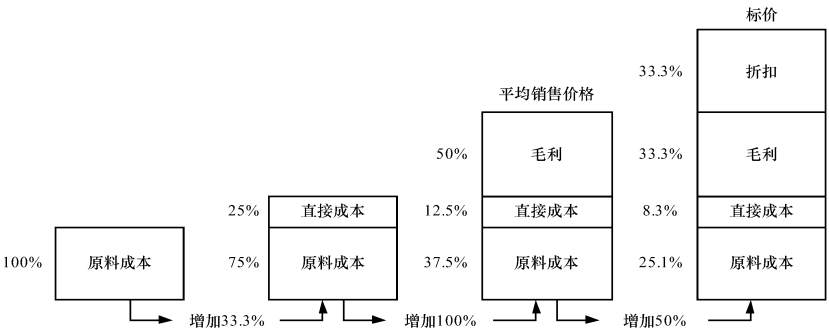


图3.2　工作站从成本到标价的演变过程

机器的性能/价格比　高的性能/价格比(Performance/Cost Ratio)是计算机设计者和使用者一致的目标。性能/价格比可定义为速度/买价,系指用单位代价(通常以百万美元表示)所获取的性能(通常以MIPS或Mflops表示)。例如说,每百万元能获取多少个MIPS(每秒百万条指令)或多少个Mflops(每秒百万次浮点运算)。高的性能/价格比就意味着是成本有效的(Cost Effectively)。而成本有效性(Cost Effectiveness)可用利用率(Utilization)来指示。利用率可定义为可达到的速度与峰值速度之比。较高的利用率对应着每美元较多的Gflops。

近代计算机的设计者非常注意提高机器的性能/价格比。例如,由于工作站机群COW的设计采用了COTS(Commodity Off-the-shelf)技术,使得其性能/价格比要比PVP和MPP等高得多。一般一台超级计算机或大规模并行机都很昂贵(费用常以几百万元、几千万元计),而一台高性能的工作站相对便宜(费用仅以几万元或十几万元计)。一个COW系统从浮点运算能力来看,虽然每台工作站只有几Mflops到几十Mflops,但一群工作站的总体运算性能可高达Gflops的量级,能接近一些超级计算机的性能,但价格却低了很多。

利用率和成本有效性　如上述,利用率可定义为实际可达速度与理论峰值速度之比。如果用成本来衡量,则利用率对应于Gflops/美元。低的利用率总是指明程序或是编译器很差。经验的数据是:执行在单处理器的MPP上的顺序应用,其利用率为5%～40%,典型的为8%～25%;而执行在多个处理器的MPP上的并行应用,其应用率为1%～35%,典型的为4%～20%。所以,一般人认为执行在单节点上的顺序的利用率总是高于并行程序,因为后者伴有通信、等待等额外开销。

尽管高的性能/价格比意味着是成本有效的,但成本有效性的度量不应与性能/价格比相混淆,性能/价格比是被定义为速度与实价之比。

3.2 加速比性能定律

简单地讲,并行系统的加速(比)是指对于一个给定的应用,并行算法(或并行程序)的执行速度相对于串行算法(或串行程序)的执行速度加快了多少倍,本节将要讨论三种加速比性能定律:适用于固定计算负载的 Amdahl 定律(Amdahl's Law);适用于可扩放问题的 Gustafson 定律(Gustafson's Law)和受限于存储器的 Sun 和 Ni 定律(Sun and Ni's Law)。为了以下讨论方便,兹定义如下参数:令 p 是并行系统中处理器数;W 是问题规模(下文中也常叫作计算负载、工作负载,它定义为给定问题的总计算量),W_s 是应用程序中的串行分量,W 中可并行化部分为 W_p(显然 $W_s+W_p=W$);*f 是串行分量比例*($f=W_s/W$,$W_s=W_1$),$1-f$ *为并行分量比例*(显然 $f+(1-f)=1$);$T_s=T_1$ 为串行执行时间,T_p 为并行执行时间;S 为加速(比),E 为效率。

3.2.1 Amdahl 定律

Amdahl 加速定律的基本出发点是:①对于很多科学计算,实时性要求很高,即在此类应用中时间是个关键因素,而计算负载是固定不变的。为此在一定的计算负载下,为达到实时性可利用增加处理器数来提高计算速度;②因为固定的计算负载是可分布在多个处理器上的,这样增加了处理器就加快了执行速度,从而达到了加速的目的。在此意义下,1967 年 Amdahl 推导出了如下固定负载的加速公式:

$$S=\frac{W_s+W_p}{W_s+W_p/p} \tag{3.4}$$

为了归一化,W_s+W_p 可相应地表示为 $f+(1-f)$,*所以*

$$S=\frac{f+(1-f)}{f+\dfrac{1-f}{p}}=\frac{p}{1+f(p-1)} \tag{3.5}$$

当 $p\to\infty$时,上式极限为:

$$S=\frac{1}{f} \tag{3.6}$$

这就是著名的 Amdahl 加速定律,它意味着随着处理器数目的无限增大,并行系统所能达到的加速之上限为 $1/f$,这是一个很悲观的结论。

Amdahl 定律的几何意义可清楚地表示在图 3.3 中。

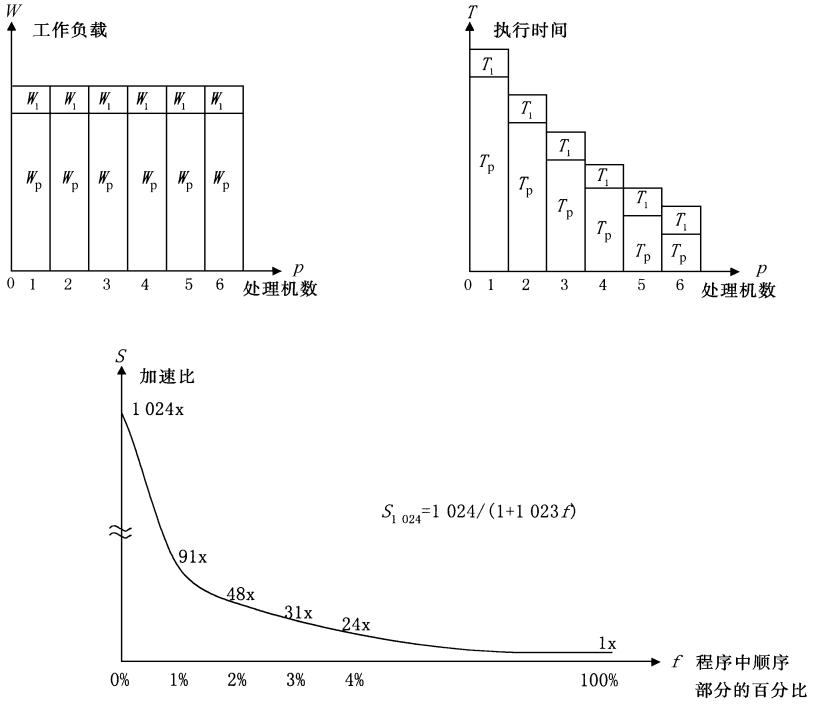


图 3.3　Amdahl 加速定律

实际上并行加速不仅受限于程序的串行分量，而且也受并行程序运行时的额外开销影响。令 W_0 为额外开销，则(3.4)式应修改为：

$$S=\frac{W_s+W_p}{W_s+\frac{W_p}{p}+W_0}=\frac{W}{fW+\frac{W(1-f)}{p}+W_0}=\frac{p}{1+f(p-1)+W_0p/W} \tag{3.7}$$

当 $p\to\infty$ 时，上式变为：

$$S=\frac{1}{f+W_0/W} \tag{3.8}$$

可见，串行分量越大和并行额外开销越大，则加速越小。

3.2.2　Gustafson 定律

Gustafson 加速定律的基本出发点是：①对于很多大型计算，精度要求很高，即

在此类应用中精度是个关键因素,而计算时间是固定不变的。此时为了提高精度,必须加大计算量,相应地亦必须增多处理器数才能维持时间不变;②除非学术研究,在实际应用中没有必要固定工作负载而使计算程序运行在不同数目的处理器上,增多处理器必须相应地增大问题规模才有实际意义。按此意义,1987 年 Gustafson 给出了如下放大问题规模的加速公式:

$$S' = \frac{W_s + pW_p}{W_s + p \cdot W_p / p} = \frac{W_s + pW_p}{W_s + W_p} \tag{3.9}$$

归一化后可得

$$S' = f + p(1-f) = p + f(1-p) = p - f(p-1) \tag{3.10}$$

当 p 充分大时,S′与 p 几乎成线性关系,其斜率为 $1-f$。*这就是著名的* Gustafson 加速定律,它意味着随着处理器数目的增加,加速几乎与处理器数成比例的线性增加,串行比例 f 不再是程序的瓶颈,这对并行系统的发展是个非常乐观的结论。

Gustafson 定律的几何意义可清楚地表示在图 3.4 中。

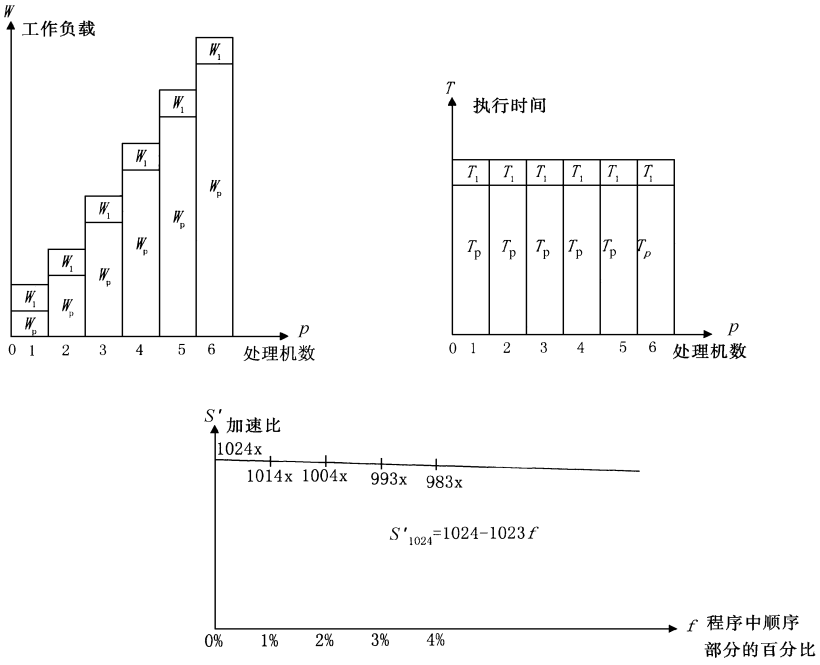


图 3.4 Gustafson 加速定律

同样,当考虑到并行程序运行时的额外开销 W_0 时,(3.9) 式应修改为:

$$S' = \frac{W_s + pW_p}{W_s + W_p + W_0} = \frac{f + p(1-f)}{1 + W_0/W} \tag{3.11}$$

注意:W_0 是 p 的函数,它可能随 p 增大、减小或不变。一般化的 Gustafson 定律欲达到线性加速必须使 W_0 随 p 减小,但这常常是困难的。

3.2.3 Sun 和 Ni 定律

Xian-He Sun 和 Lionel Ni 于 1993 年曾将 Amdahl 定律和 Gustafson 定律一般化,提出了存储受限的加速定律。其基本思想是只要存储空间许可,应尽量增大问题规模以产生更好或更精确的解(此时可能使执行时间略有增加)。换句话说,假若有足够的存储容量,并且规模可扩放的问题满足 Gustafson 定律规定的时间要求,那么就有可能进一步增大问题规模求得更好或更精确的解。

给定一个存储受限问题,假定在单节点上使用了全部存储容量 M 并在相应于 W 的时间内求解之,此时工作负载 $W = fW + (1-f)W$。*在 p 个节点的并行*系统上,能够求解较大规模的问题是因为存储容量可增加到 pM G(p) 反映存储容量增加到 p 倍时工作负载的增加量,所以扩大后的工作负载 $W = fW + (1-f)G(p)W$。*对照* (3.9) 式,存储受限的加速公式相应为:

$$S'' = \frac{fW + (1-f)G(p)W}{fW + (1-f)G(p)W/p} \tag{3.12}$$

归一化后可得

$$S'' = \frac{f + (1-f)G(p)}{f + (1-f)G(p)/p} \tag{3.13}$$

Sun 和 Ni 定律的几何意义可清楚地表示在图 3.5 中。

同样,当考虑到并行程序运行时的额外开销 W_0 时,(3.12) 式和 (3.13) 式可修改为:

$$S'' = \frac{fW + (1-f)WG(p)}{fW + (1-f)G(p)W/p + W_0} = \frac{f + (1-f)G(p)}{f + (1-f)G(p)/p + W_0/W} \tag{3.14}$$

由 (3.13) 可知,当 $G(p) = 1$ 时,它变为 $\frac{1}{f + (1-f)/p}$,这就是 Amdahl 加速定律 (3.5) 式;当 $G(p) = p$ 时,*它变为* $f + p(1-f)$,*这就是* Gustafson 加速定律 (3.10) 式;当 $G(p) > p$ 时,它相应于计算负载比存储要求增加得快,此时 Sun 和 Ni 加速均比 Amdahl 加速和 Gustafson 加速为高。

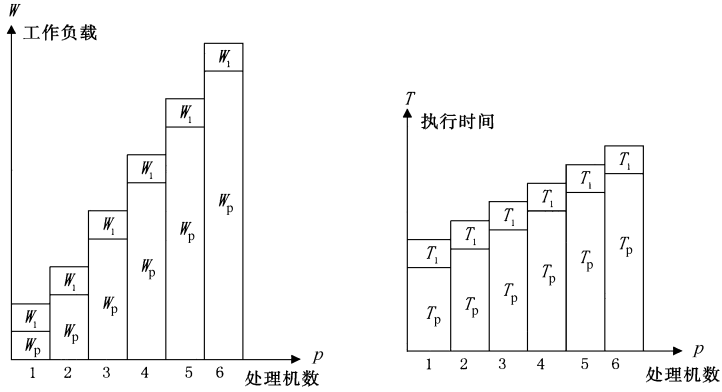


图3.5 Sun 和 Ni 加速定律

3.2.4 有关加速的讨论

在实际应用中,可供参考的加速经验公式是:

$$p/\log p \leq S \leq p \tag{3.15}$$

可达线性加速的应用问题有诸如矩阵相加、内积运算等,此类问题几乎没有通信开销;对于分治类的应用问题,它类似于二叉树,树的同级可并行执行,但向根逐级推进时,并行度将逐渐减少,此类问题可望达到 $p/\log p$ 的加速;对于通信密集类的应用问题,其加速经验公式可参考下式:

$$S=1/C(p) \tag{3.16}$$

其中 $C(p)$ 是 p 个处理器的某一通信函数,或为线性的或为对数的。

严格的线性加速是难以达到的,更何况超线性加速(Superlinear Speedup)。但在某些算法或程序中,可能会出现超线性加速现象:例如,在某些并行搜索算法中,允许不同的处理器在不同的分枝方向上同时搜索,当某一处理器一旦迅速地找到了解,它就向其余的处理器发出中止搜索的信号,这就会提前取消那些在串行算法中所作的无谓的搜索分枝,从而出现超线性加速现象;又如,在绝大多数并行机中,每个处理器均有少量的高速缓存,当某一问题执行在大量的处理器上,而其大多的数据均放在高速缓存中时,总的计算时间趋于减少,如果由于这种高速缓存效应所造成的计算时间下降补偿了由于通信等所造成的额外开销时间,则有可能造成超线性加速现象。

最后值得指出的是,加速的含意对科学研究者和工程实用者可能有所不同:前

者乐于使用绝对加速(Absolute Speedup)的定义,即对于给定的问题,最佳串行算法所用的时间除以同一问题其并行算法所用的时间;后者乐于使用相对加速(Relative Speedup)的定义,即对于给定的问题,同一个算法在单处理器上运行的时间除以在多个处理器上运行的时间。显然相对加速的定义是较宽松和实际的。

3.3 可扩放性评测标准

评价并行计算性能的指标,除了上节所介绍的加速比以外,并行计算的可扩放性(Scalability)也是主要性能指标之一。可扩放性最简朴的含意是在确定的应用背景下,计算机系统(或算法或编程等)性能随处理器数的增加而按比例提高的能力。现今它已成为并行处理中一个重要的研究问题,被越来越广泛地用来描述并行算法(并行程序)能否有效利用可扩充的处理器数的能力。

3.3.1 并行计算的可扩放性

从上节所介绍的三种加速定律可知,增加处理器数和求解问题的规模都可能提高加速比,而影响加速的因素有:①求解问题中的串行分量;②并行处理所引起的额外开销(通信、等待、竞争、冗余操作和同步等);③加大的处理器数超过了算法中的并发程度。增加问题的规模有利于提高加速的因素是:①较大的问题规模可提供较高的并发度;②额外开销的增加可能慢于有效计算的增加;③算法中的串行分量比例不是固定不变的(串行部分所占的比例随着问题规模的增大而缩小)。一般情况下,增加处理器数,是会增大额外开销和降低处理器的利用率的,所以对于一个特定的并行系统、并行算法或并行程序,它们能否有效利用不断增加的处理器的能力应是受限的,而度量这种能力就是可扩放性这一指标。

按照 Webster 字典所作的定义(Scalability is the ability to scale, i.e, the ability to adjust according to a proportion),可扩放性涉及到调整什么和按什么比例两方面的问题。对于并行计算而言,要调整的是处理器数 p 和问题规模 W,两者可按不同比例进行调整,此比例关系(可能是线性的、多项式的或指数的等)就反映了可扩放的程度。

当研究可扩放性时,总是将并行算法和体系结构一并考虑,也就是说可扩放性应该是算法和结构的组合。所以当谈论算法的可扩放性时,实际上是指该算法针对某一特定机器结构的可扩放性;同样当谈论体系结构的可扩放性时,实际上是指运行于该体系结构的机器上的某一个(或某一类)并行算法的可扩放性。

进行可扩放性研究的主要目的是:①确定解决某类问题用何种并行算法与何

种并行体系结构的组合,可以有效地利用大量的处理器;②对于运行于某种体系结构的并行机上的某种算法,根据算法在小规模处理机上的运行性能,预测该并行算法当移植到大规模处理机上后运行的性能;③对固定的问题规模,确定在某类并行机上最优的处理器数与可获得的最大的加速比;④用于指导改进并行算法和并行机体系结构,以使并行算法尽可能地充分利用可扩充的大量处理器。

尽管可扩放性如此重要,并且已被广泛的研究,但目前仍无一个公认的、标准的和被普遍接受的严格定义和评判它的标准。下面从不同的角度,介绍三种典型的可扩放性度量方法,即等效率、等速度和平均延迟方法。

3.3.2 等效率度量标准

可扩放性的概念是与加速比和效率的概念紧密相关的,为此必须先从加速 S 和效率 E 讲起。

令 t_e^i 和 t_o^i 分别是并行系统上第 i 个处理器的有用计算时间和额外开销时间(包括通信、同步和空闲等待时间等);所有 t_e^i 之和记之为 $T_e=\sum_{i=0}^{p-1} t_e^i$,所有 t_o^i 之和记之为 $T_O=\sum_{i=0}^{p-1} t_o^i$;令 T_p 是 p 个处理器系统上并行算法的运行时间,对于任意 i,显然有 $T_p=t_e^i+t_o^i$,且

$$T_e+T_O=pT_p \tag{3.17}$$

问题的规模 W 可定义为由最佳串行算法所完成的计算量,也称工作负载或工作量,即 $W=T_e$。所以并行算法的加速和效率可分别定义如下:

$$S=\frac{T_e}{T_p}=\frac{T_e}{\frac{T_e+T_O}{p}}=\frac{p}{1+\frac{T_O}{T_e}}=\frac{p}{1+\frac{T_O}{W}} \tag{3.18}$$

$$E=\frac{S}{p}=\frac{1}{1+\frac{T_O}{T_e}}=\frac{1}{1+\frac{T_O}{W}} \tag{3.19}$$

在通常情况下,如果问题规模 W 保持不变(即 T_e 保持不变),则随着处理器数 p 的增加,开销 T_O 也会随之增大,根据(3.19)式,效率 E 也会相应下降。为了维持效率 E 不变,就要保持 T_O/T_e 的值不变,故需要在处理器数 p 增大的同时相应地增加问题规模 W 的值(即 T_e 的值)才有可能抵消由于 p 的增大而导致 T_O 增大的影响,从而保持效率不变。也就是说,为了维持一定的效率(介于 0 与 1 之间),当处理数 p 增大时,需要相应地增大问题规模 W 的值。由此定义函数 $f_E(p)$ 为问题规模 W 随处理器数 p 变化的函数,称此函数为等效率函数(Iso-efficiency

Function)，它是由 Kumar 等人 1987 年提出的。

按照等效率函数的定义，对于某一并行算法（或并行程序），为了维持运行效率保持不变，随着处理器数目的增加，若只需增加较小的工作量（即问题规模），比如说 W 随 p 呈线性或亚线性增长，则表示该算法具有良好的可扩放性；若需增加非常大的问题规模，比如说 W 随 p 呈指数级增长，则表示该算法是不可扩放的。

图 3.6 给出了三种等效率函数曲线，曲线 1 表示算法具有很好的扩放性；曲线 2 表示算法是可扩放的；曲线 3 表示算法是不可扩放的。

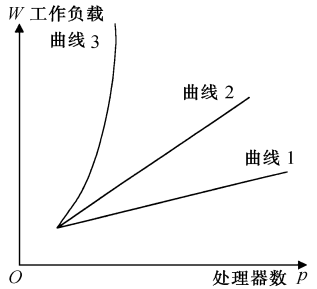


图 3.6　等效率函数曲线

3.3.3　等速度度量标准

等效率度量标准最大优点是，可用简单的、可定量计算的、少量的参数就能计算出等效率函数，并由其复杂性可指明算法的可扩放程度。这对于具有网络互连结构的并行机来说是很合适的，因为 T_o 是可一步一步计算出的。也正是这个 T_o 是计算等效率函数的惟一关键参数，所以如果它不能够方便地计算出，则等效率函数度量可扩放性的方法就受到了限制。我们知道开销 T_o 通常包括了通信、同步、等待等非有效计算时间。不幸的是，在共享存储的并行计算机中，T_o 则主要是非局部访问的读/写时间、进程调度时间、存储竞争时间以及高速缓存一致性操作时间等，而这些时间都是难以准确计算的。所以用解析计算的方法度量可扩放性不应是一种惟一的方法。事实上，两位中国学者 Xian－He Sun 和 Xiao Dong Zhang 于 1994 年分别提出了以试验测试为主要手段的两种评测可扩放性的标准，即等速度标准（ISO-speed Metrics）和平均延迟标准（Average Latency Metrics）。本节先介绍前者，下节再介绍后者。

大家知道，速度是一个非常重要的机器参数，一般在机器性能指标中都明确地给出，并常以每秒多少次浮点运算（Flops）来表明（按照约定的含意，浮点运算数目

就是工作负载 W。所以若用速度来度量可扩放性,在原理上讲是更方便的,而等速度方法的基本出发点就在于此。

在并行系统中,提高速度可以使用增加处理器数的办法,如果速度能以处理器数的增加而线性增加(此即意味着平均速度不变),则说明此系统具有很好的扩放性。为此先作如下定义:

令 *p 表示处理器个数,W 表示要求解问题的工作量或称问题规模(在此可指浮点操作个数)*,T 为并行执行时间,则定义并行计算的速度 V 为工作量 W 除以并行时间 T:

$$V = W/T \tag{3.20}$$

而 p 个处理器的并行系统的平均速度定义为并行计算速度 V 除以处理器个数 p:

$$\overline{V} = \frac{V}{p} = \frac{W}{pT} \tag{3.21}$$

根据平均速度(3.21)式,就可定义等平均速度(简称为等速度)可扩放度量标准如下:

对于运行于并行机上的某个算法,当处理器数目增加时,若增大一定的工作量能维持整个并行系统的平均速度不变,则称该计算是可扩放的。

注意,平均速度为常数,意即速度与处理器数目成线性比例增长,亦即加速比为线性的。

按此定义,令 W 是使用 p 个处理器时算法的工作量,令 W' 表示当处理数从 p 增大到 p' 时,为了保持整个系统的平均速度不变所需执行的工作量,则可得到处理器数从 p 到 p' 时平均速度可扩放度量标准公式为:

$$\Psi(p, p') = \frac{W/p}{W'/p'} = \frac{p'W}{pW'} \tag{3.22}$$

用上述公式计算出的值介于 0 与 1 之间,值越大表示可扩放性越好。

当平均速度严格保持不变时,即

$$\frac{W}{Tp} = \frac{W'}{T'p'} \quad ? \quad \frac{p'W}{pW'} = \frac{T}{T'}$$

所以,(3.22)式可变为:

$$\Psi(p, p') = \frac{T}{T'} \tag{3.23}$$

当 $p=1$ 时,记 $T = T_1$;当处理器个数为 p' 时,记 $T' = T_{p'}$,相应地记 $p=1$ 时 $\Psi(1, p')$ 为 $\Psi(p')$,于是有

$$\Psi(p') = \frac{p'W}{W'} = \frac{T_1}{T_{p'}} = \frac{W}{W'/p'} = \frac{\text{解决工作量为 } W \text{ 的问题所需串行时间}}{\text{解决工作量为 } W' \text{ 的问题所需并行时间}} \tag{3.24}$$

(3.24)式给出的可扩放性定义与传统的加速比定义有点类似,其主要差别在于:加速比的定义是保持问题规模不变;而可扩放性定义是保持平均速度不变。如图

3.7 所示,加速比是标志并行处理相对于串行处理所获得性能增加;而如图 3.8 所示,可扩放性是标志从小规模系统到大规模系统所引起的性能衰减。

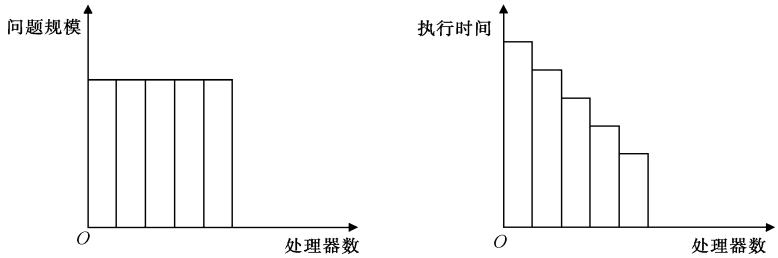


图 3.7　加速比:问题规模不变,时间变短

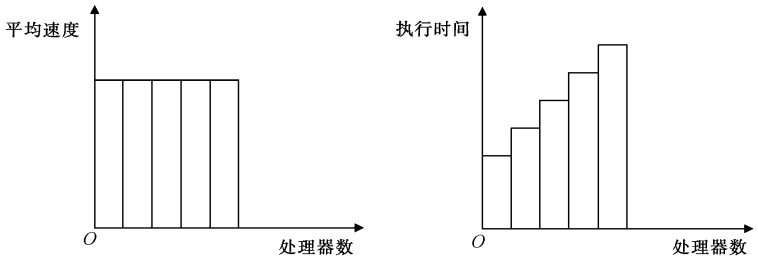


图 3.8　可扩放性:平均速度不变,时间加长

一般有三种方法可得到等速度可扩放性标准:①测量法:使用软件方法,即采用控制程序去调用应用程序,找寻所希望的固定的平均速度;②计算法:首先找出平均速度和执行时间之间的关系,再使用(3.23)式计算之;③预计法:采用推导一般可扩放性公式来研究可扩放性。

3.3.4　平均延迟度量标准

等速度度量标准最大优点是,使用机器性能速度指标这一明确的物理量来度量可扩放性是比较直观的。它有一系列优点:①速度是由工作负载 *W 和执行时*间 T 决定的,而 W 反映了应用程序的性质,T 反映了结构和程序效率的影响;②速度是各种结构的机器相互可比较的量;③执行时间包含了计算和延迟这两个主要的时间量;④速度是比较容易测量的,因为 W 可由所执行的浮点操作数决定。但它也有一些不足:①仅用浮点操作数作为 *W 是不全面的,某些非浮点运算可能*造成性能的主要变化;②延迟虽包含在执行时间中,但它未明确地定义为 W 的函

数。实际上,可扩放性测定可以在系统和应用的更低层上,它们能更精确地抓住影响性能的结构因素和程序开销模式。下面将介绍使用测量平均延迟开销的办法来度量可扩放性。为此先作如下定义:

参照图3.9,令 T_i 为 P_i 的执行时间,它还包括在运行时所招致的延迟 L_i,程序运行时还包括启动与停止时间。所以第 i 个处理器 P_i 的总延迟时间为"L_i+启动前时间+停止后时间"。定义系统平均延迟时间 $\overline{L}(W,p)$ 为:

$$\overline{L}(W,p)=\sum_{i=1}^{p}(T_{para}-T_i+L_i)/p \tag{3.25}$$

因为,$pT_{para}=T_o+T_{seq}$ (T_{seq}为串行执行时间) 和 $T_o=p\overline{L}(W,p)$,所以

$$\overline{L}(W,p)=T_{para}-T_{seq}/p \tag{3.26}$$

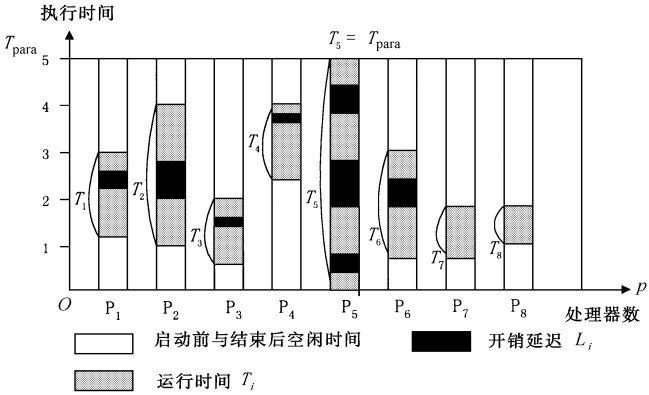


图3.9 定义平均延迟示意图

对于一个算法-机器组合,令 $\overline{L}(W,p)$ 表示在 p 个处理器上求解工作量为 *W 问题的平均延迟*;$\overline{L}(W',p')$ 表示在 p'个处理器上求解工作量为 W'问题的平均延迟。当处理器数由 p 变到 p',而维持并行执行效率不变,则定义平均延迟可扩放性度量标准为:

$$\Phi(E,p,p')=\frac{\overline{L}(W,p)}{\overline{L}(W',p')} \tag{3.27}$$

用上述公式计算出的值介于0与1之间,值越大表示可扩放性越好。

3.3.5 有关可扩放性标准的讨论

以上三节分别介绍了三种典型的可扩放性度量标准 等效率、等速度和平均延

迟。等效率度量标准是在保持效率 E 不变的前提下,研究问题规模 W 如何随处理器个数 p 而变化;等速度度量标准是在保持平均速度不变的前提下,研究处理器数 p 增多时应该相应地增加多少工作量 W;平均延迟度量标准则是在效率 E 不变的前提下,用平均延迟的比值来标志随着处理器数 p 的增加需要增加的工作量 W。*三种评判可扩放性的标准的基本出发点都是抓住了影响算法可扩放性的基本参数* T_0,只是等效率标准是采用解析计算的方法得到 T_0;等速度标准是将 T_0 隐含在所测量的执行时间中;而平均延迟标准则是保持效率为恒值时,通过调节 W 与 p 来测量并行和串行执行时间,最终通过平均延迟反映出 T_0,所以等速度与平均延迟标准都是辅之以测试手段而得到有关性能参数(如速度与时间等)来评判可扩放性的;而等效率标准是通过解析计算开销参数 T_0 来评判可扩放性的。平均速度 $\overline{V}$ 和平均延迟 $\overline{L}$ 两个物理量的引入,目的近乎一样:$\Psi(p,p')=T/T'$*保持不变*,意即 T_0 随 p 变化很小;$\Phi(E,p,p')=\overline{L}/\overline{L}'$*保持不变,亦意即* T_0 随 p 变化甚小。

事实上,三种度量可扩放性的标准是彼此等效的,这可简单推导如下:

由于运行在 p 个处理器上的工作量为 W 的程序串行执行时间为 Wt_c,其效率 $E=\dfrac{Wt_c}{pT_p}$,而程序的平均速度 $\overline{V}=\dfrac{W}{pT_p}$,所以 $E=\overline{V}\cdot t_c$,其中 t_c 为常数。这表明等效率与等速度两种度量可扩放性的标准在物理意义上是一致的

由于平均速度 $\overline{V}=\dfrac{W}{pT}$,$pT=T_0+T_{seq}$,而 $T_0=p\overline{L}(W,p)$ 且 $T_{seq}=Wt_c$,所以

$$\overline{V}=\frac{W}{pT}=\frac{W}{T_0+T_{seq}}=\frac{W}{p\overline{L}(W,p)+Wt_c}=\frac{W/p}{\overline{L}(W,p)+Wt_c/p}$$

由此可推导出

$$\overline{L}(W,p)=\frac{W}{p}\frac{1}{\overline{V}}-t_c$$

当 $\overline{V}$ 保持不变时,$\dfrac{1}{\overline{V}}-t_c$ 为常数,所以

$$\frac{\overline{L}(W,p)}{\overline{L}(W',p')}=\frac{W}{p}\Big/\frac{W'}{p'}=\frac{p'W}{pW'} \tag{3.28}$$

根据(3.22)式和(3.27)式,(3.28)式表明从等速度度量标准可以导出平均延迟度量标准,两者是完全一致的。

最后要说的是,既然等速度和平均延迟两个可扩放性评判标准都是立足于测量的办法,那么就要为此提供一套测试手段、工具、环境以搜集、测量、分析有关数据并显示可扩放性性能。

*3.4 基准测试程序

基准测试程序(Benchmark)用于测试和预测计算机系统的性能,揭示不同结构机器的长处和短处,为用户决定购买或使用哪种机器最适合他们的应用要求提供决策。基准测试程序试图提供一个客观、公正的评价机器性能的标准。但真正做到完全公正并非易事,要涉及到的因素很多,包括硬件、体系结构、编译优化、编程环境、测试条件、解题算法等。一组标准的测试程序要提供一组控制测试条件和步骤的规则说明,包括测试平台环境、输入数据、输出结果和性能指标等。

不同的基准测试程序,侧重目的也有所不同:有的测试CPU性能,有的测试文件服务器性能,有的测试输入/输出界面,有的则测试网络通信速度等。根据不同用途,测试程序可有专用和通用之分。目前国际上流行的通用测试程序可分为好几类:①综合型(如Dhrystone、Whetstone等);②核心型(如Livermore Fortran Kernels、NASA之NAS等);③数学库(如Linpack、FFT等);④应用型(如SPEC、Perfect、Splash等);⑤并行型(如NAS之NPB、PARKBENCH等)。本节仅仅介绍一些常见测试程序,它们很多都可在Internet上公开查到:http://www.netlib.org/Liblist.html。在这里你可查到LINPACK、LAPACK、BLAS、BLACS、Livermore Loops、Dhrystone、Whetstone、NAS、SPEC、Sim等包含源代码的基准测试程序。

3.4.1 基本的测试程序

Whetstone 它是为比较不同的计算机的浮点性能而设计的综合型基准测试程序,最早用Algol-60写成,后用Fortran改写。这是从英国国立物理实验室1970年时常用的数值计算程序中取出的最频繁使用的有代表性的程序段。这些程序段语句被转换到称之为Whetstone虚拟计算机上的指令,因而得名Whetstone基准程序。此基准程序既包括整数运算,又包括浮点运算,涉及到数组下标索引、子程序调用、参数传递、条件转移和三角/超越函数等,使用系统完成的Kwhetstone/s数来度量。

测试程序可访问Netlib/benchmark/whetstone

Dhrystone 它主要为测试整数与逻辑运算性能而设计的综合型基准测试程序,用Ada、C和Pascal写成,是一种CPU密集(CPU-intensive)型测试程序,由很多整型语句与逻辑语句的小循环组成。它使用系统完成的KDhrystone/s数来度量(注意VAX11/780的性能为1.7KDhrystone/s,而VAX11/780通常定义为1个

MIPS性能标准,一个MIPS代表每秒10^6指令)。

测试程序可访问Netlib/benchmark/dhrystone

人们对Whetstone和Dhrystone这两个综合型基准程序的批评是,它们不能预测用户程序性能,这些基准程序的主要缺点是对编译程序比较敏感。

SPEC SPEC是Standard Performance Evaluation Cooperation的首字母缩写,它是作为NCGA(National Computer Graphics Association)的一个小组于20世纪80年代创立,这个小组的创始者来源于Hewlett-Packard、DEC、MIPS和Sun Microsystems,他们拥有一组基准测试程序以评测新机器的性能。第一组基准程序叫作SPEC89,包含10个程序;SPEC92扩充至20个程序,6个整数程序和14个浮点程序分别称为SPECint92和SPECfp92。最近SPEC又发布了一些新的基准测试程序(如SPEC95、SPEChpc96、SPECweb96等)。SPEC原主要是测试CPU性能的,现在强调开发能反映真实应用(如实际负载等)的基准测试程序,并已推广至客户/服务器计算、商业应用、I/O子系统等。

SPEC基准测试程序使用的单位是所测试机器执行性能与VAX11/780执行性能之比值。

3.4.2 数学库测试程序

LinPACK 自20世纪70年代中期以来,国际上曾开发过一批基于Fortran语言的求解线性代数方程组的子程序,于1979年正式发布了LinPACK包。因为线性代数方程组在各个领域中应用甚广,所以该软件包就很自然地成为测试各种机器性能的测试程序。LinPACK基准测试程序(LinPACK Benchmark)是用全精度64位字长的子程序求解100阶线性方程组的速度,测试的结果以Mflops(每秒百万次浮点运算)作单位给出。LinPACK的测试报告由J.Dongarra经常更新发布(通常每月发布一次)。LinPACK是使用BLAS1(Basic Algebra Subprograms1)的第一个线性代数软件包。BLAS1执行通常意义下的标量-向量运算(如标量乘向量、向量加、向量内积等),主要为向量计算机而设计的,它用Fortran77写成,但有些计算机商也提供汇编语言的BLAS1版本。

LAPACK 尽管LinPACK作为测试程序现在仍很有生命力,但作为实际求解线性代数问题的软件包已经落伍了。所以1992年推出了代替LinPACK及EisPACK(特征值软件包)的LAPACK基准测试程序(LAPACK Benchmark),它使用了数值线性代数中最新、最精确的算法,同时采用了将大型矩阵分解成小块矩阵的方法从而可有效地使用存储器。LAPACK是建立在BLAS1、BLAS2和BLAS3基础上的,其中BLAS2执行矩阵-向量运算,BLAS3执行矩阵-矩阵运算。

ScaLAPACK ScaLAPACK 是 LAPACK 的增强版,主要为可扩放的、分布存储的并行计算机而设计的。ScaLAPACK 支持稠密和带状矩阵上各类操作,诸如乘法、转置和分解等。在国际上,ScaLAPACK 例程可以加入多个并行算法,并且可根据数据分布、问题规模和机器大小选择这些算法,然而用户却不必关心这些细节。

3.4.3 并行测试程序

NAS Parallel Benchmark (NPB) 它是 1991 年美国 NAS (Numerical Aerodynamic Simulation) 项目所开发的并行测试程序,其目的是为了比较各种并行机性能,有时也简称为 NPB (NAS Parallel Benchmark) 并行测试程序,系由 8 个程序组成,测试范围从整数排序到复杂的数值计算。测试结果以单处理机的 Cray Y-MP/1 为单位 (Class A) 或 Cray C90/1 为单位 (Class B) 作比较。

NPB 由 5 个核心程序组成:①EP (Embarrassingly Parallel):用于计算 Gauss 伪随机数,因为它几乎不要求处理器之间相互通信,所以很适合于并行计算,而所测得的结果往往可以作为一个特定并行系统浮点计算性能可能达到的上限;②MG (MultiGrid):用 4 个 V 循环多重网格算法求解三维波松方程的离散周期近似解;③CG (Conjugate Gradient):用于求解大型稀疏对称正定矩阵的最小特征值的近似值,它表征了非结构风格计算和非规整远程通信计算类问题;④FT (Fast Fourier Transformation):用于求解基于 FFT 谱分析法的三维偏微分方程,它也要求远程通信;⑤IS (Integer Sort):用于基于桶排序的二维大整数排序,它要求大量的全交换通信。另外还有计算流体力学中 3 个模拟程序:① LU (Lower Upper Triangular):用于基于对称超松弛法求解块稀疏方程组;②SP (Scalar Penta Diagonal):用于求解 5 对角线方程组;③BT (Block Tri-Diagonal):用于求解 3 对角块方程组。

有关 NAS 并行基准程序的 WWW home Page 地址为 :http://www.nas.nasa.gov/NAS/NPB/software/npbsoftware.html.

PARKBENCH PARKBENCH (PARallel Kernels and BENCHmarks) 是在 1992 年超级计算会议上确定的项目。与会者认为并行测试的重点应放在可扩放、分布存储、消息传递的体系结构上。主要目标是确定并行机用户与厂商双方都能接受的内容丰富的一批并行测试程序及标准,并把结果公布于网上,以减少不必要的重复工作。

现在的基准程序系为分布存储的多计算机编写,使用 Fortran77 加上 PVM 或 MPI,为共享存储结构的 Fortran90 和 HPF 版本的基准程序正在开发中。目前,PARKBENCH 包括 4 类:①底层基准程序:测试一些基本结构参数,诸如算术运算

速度、高速缓存和存储器速度、通信启动时间和带宽以及同步开销等；②核心基准程序：涉及到广泛的经常使用的科学计算子程序，诸如矩阵运算（稠密矩阵乘法、转置、LU 分解、QR 分解，矩阵三对角化等）、FFT 运算、求解 PDE 和 NPB 核基准程序等；③密集应用基准程序：目前仅包括谱变换、浅水（Shallow Water）模拟和 3 个 NPB 模似应用程序；④HPF 编译基准程序：测试 HPF 编译器性能，主要集中在显式并行 HPF 结构的并行实现上。

PARKBENCH 是个正在研究的课题，测试程序的内容尚未完全定型。目前所包含的核心测试程序主要来自 NPB。

3.5 小结和导读

小结 并行计算的性能与所使用的并行计算机本身的性能有关，所以研究并行计算的性能先了解一下并行机的一些基本性能指标也是很必要的。在并行系统上进行计算的主要目的就是要加速整个计算的过程，所以研究并行计算（并行算法、并行程序）的加速比性能是最根本的。随着计算负载的增加和机器规模的扩大，研究并行计算的性能是否能随处理器数目的增加而按比例的增加（这就是所谓的可扩放性）也是很重要的。为了客观、公正地评估计算机性能，提高并行机的使用水平和效率，减少用户引进和购买高性能计算机的育目性，所以了解掌握一些基本的基准测试程序也是很必需的。本章对以上四方面的问题逐一作了简要讨论。

此外，所介绍的基准测试程序大都限于科学和工程计算方面，其实在联机事务处理、商业计算、数据库等方面也有不少有名的基准测试程序。例如，TPC（Transaction Processing Council）基准程序就很流行，目前 TPC-A 和 TPC-B 已在 1995 年 6 月废弃，而 TPC-C（测试事务处理价格/性能比）、TPC-D（测试决策支持系统）和 TPC-E（测试大型企业支持计算环境的能力）在商业应用中广泛使用。

导读 Amdahl 加速定律最早描述在 [12] 中；Gustafson 定律描述在 [85] 中；Sun 和 Ni 加速定律描述在 [170] 中；等效率标准是由 Kumar 等提出 [112]；等速度标准是由 Sun 等提出 [171]；平均延迟标准由 Zhang 等提出 [187]；有关并行算法的可扩放性分析，读者可参阅 [192]，在那里还描述了很多其它的可扩放评判标准。有关计算机性能评价，[66] 是篇很好的综述；LinPACK 的使用手册，见 [61]；LAPACK 的使用手册，见 [14]；有关 NAS 并行基准程序，请见 Ames 研究中心报告 [22]。最后读者可按下述 WWW 主页地址，在 Internet 上查阅到很多基准测试程序，它们还会给出程序的源代码和测试的结果：http://www.netlib.org/liblist.html。

习　题

3.1　试讨论公式(3.7)、(3.11)和(3.14)三者的物理意义和相互关系。

3.2　两个 $N\times N$ 阶的矩阵相乘，时间复杂度为 $T_1=CN^3$s，其中 C 为常数；在 n 节点的并行机上并行矩阵乘法的时间为 $T_n=(CN^3/n+bN^2/n)$s，其中 b 是另一常数，第一项代表计算时间，第二项代表通信开销。

①试求固定负载时的加速并讨论其结果；

②试求固定时间时的加速并讨论其结果；

③试求存储受限时的加速并讨论其结果。

3.3　试讨论本文所讲的三种加速定律的使用场合和它们的相互关系。

3.4　试综合比较等效率、等速度和平均延迟三种可扩放性度量标准的异同点。

3.5　试证明等效率、等速度和平均延迟三种可扩放性质量标准的相互等效性。

3.6　用一台 40 MHz 的处理机执行标准测试程序，它含的混合指令数和相应所需的时钟周期数如下：

指令类型	指　令　数	时钟周期数
整数运算	45 000	1
数据传送	32 000	2
浮点	15 000	2
控制传送	8 000	2

试求有效每条指令执行的周期数 CPI 和每秒执行的指令数 MIPS。［提示：令 τ 为时钟周期(ns)，$f=1/\tau$ 为时钟频率(MHz)，T 为程序执行时的 CPU 时间，I_c 为指令条数，C 为程序执行的总周期数，则有 $T=I_c\times\mathrm{CPI}\times\tau$ 和 $\mathrm{MIPS}=\dfrac{I_c}{T\times10^6}=\dfrac{f}{\mathrm{CPI}\times10^6}=\dfrac{f\times I_c}{C\times10^6}$］。

3.7　假定在一台 40 MHz 的处理机上运行 200 000 条指令的目标代码，程序主要由 4 种指令组成，根据程序跟踪实验结果，已知指令混合比和每种指令所需周期数如下：

指令类型	CPI	指令混合比
算术和逻辑	1	60%
高速缓存命令中的加载/存放	2	18%
转移	4	12%
高速缓存缺失的存储访问	8	10%

①计算在单处理机上用上述跟踪这些数据运行程序的平均 CPI；

②根据①所得 CPI，计算相应的 MIPS 速率。

3.8 试分析行主编号的 $p \times p$ 网孔上 n 点 FFT 算法的可扩放性。假定通信建立时间为 t_s，跨步延迟为 t_h，传送单位信包的时间为 t_b，单位计算时间为 t_c。

① 试计算处理器 P_i 的通信跨步数 hops；

② 试计算总的通信延迟 T_0；

③ 试计算等效率函数 $W = f_E(p)$ 并加以讨论。［提示：对照 FFT 与网孔连接拓朴，可以发现处理器间的通信仅发生在同一行或同一列，且最大的通信 $hops = p/2$。］

3.9 试分析 $p = 2^d$ 个处理器的超立方网络上求 n 点 FFT 算法的可扩放性。假定参数 t_s、t_h、t_b 和 t_c 意义同上题。

① 试计算总的通信延迟 T_0；

② 试计算等效率函数 $W = f_E(p)$；

③ 试讨论当 $E < 0.5$ 和 $E = 0.9$ 时的 $W = f_E(p)$，你的结论是什么？

第二篇　并行算法的设计

第四章　并行算法的设计基础
第五章　并行算法的一般设计策略
第六章　并行算法的基本设计技术
第七章　并行算法的一般设计过程

这一篇主要研究如何设计并行算法,包括并行算法的一般设计策略(第五章),并行算法的基本设计技术(第六章)和并行算法的一般设计过程(第七章),欲系统学习并行算法的设计与分析的读者可参阅丛书之二《并行算法的设计与分析》[19]。因为任何并行算法的设计都是基于某一特定的并行计算模型的,所以在本篇的开始,首先介绍了设计并行算法常用的并行计算模型,包括 PRAM 模型、APRAM 模型、BSP 模型和 logP 模型等。至于并行算法的具体实现(它主要涉及到并行程序设计),将放在本书的第四篇中讲述。

本篇第五章,将介绍并行算法的三种常用设计策略,包括"并行化"法(直接将一个串行算法并行化),"全新"法(根据问题的特性,从头开始设计一个新的并行算法)和"借用"法(借用已知某类问题的现有算法,求解另一类与之有内在相似性的问题)。第六章,从利用并行处理技术最朴素的思想出发,介绍了算法设计的划分技术;从求解问题的方法和策略出发,介绍了算法设计的分治技术;从针对问题的特性出发,介绍了设计并行算法的平衡树法和倍增法;最后还介绍了基于重叠思想的并行处理技术中广泛使用的流水线设计技术。第七章,从设计方法学的角度,介绍如何从问题描述开始,逐步设计并行算法的全过程,包括任务划分、通信分析、任务组合和处理器映射等四个过程。它是实际设计一个并行算法的自然过程,其中前两个过程着重算法并发性和可扩放性的开拓;后两个过程着重优化算法的执行时间和均衡处理器的负载。

第四章　并行算法的设计基础

任何并行算法的设计都是基于某一特定的并行计算模型,而并行计算模型是从各种具体的并行机中抽象出来的,它能在一定程度上反映出具体并行机的属性,又可使算法研究不再局限于某一种具体的并行机。并行计算模型一般可分为抽象计算模型和实用计算模型。本章主要研究同步 PRAM 模型,异步 APRAM 模型,大同步 BSP 模型和异步 logP 模型。但在讨论这些计算模型之前,先要简要地介绍一下并行算法的基础知识,包括并行算法的定义、分类,并行算法的表达,并行算法的复杂性度量以及并行算法中的同步和通信问题等。

*4.1 并行算法的基础知识

4.1.1 并行算法的定义和分类

定义 4.1 算法是解题方法的精确描述，是一组有穷的规则，它们规定了解决某一特定类型问题的一系列运算。

定义 4.2 并行算法 (Parallel Algorithm) 是一些可同时执行的诸进程的集合，这些进程互相作用和协调动作从而达到给定问题的求解。

并行算法可从不同的角度分类成数值计算的和非数值计算的并行算法；同步的、异步的和分布式的并行算法；共享存储的和分布存储的并行算法；确定的和随机的并行算法等等。

定义 4.3 数值计算是指基于代数关系运算的一类诸如矩阵运算、多项式求值、求解线性方程组等数值计算问题。求解数值计算问题的算法称为数值算法 (Numerical Algorithm)。

定义 4.4 非数值计算是指基于比较关系运算的一类诸如排序、选择、搜索、匹配等符号处理问题。求解非数值计算问题的算法称为非数值算法 (Non_numerical Algorithm)。

定义 4.5 同步算法 (Synchronized Algorithm) 是指算法的诸进程的执行必须相互等待的一类并行算法。

定义 4.6 异步算法 (Asynchronized Algorithm) 是指算法的诸进程的执行不必相互等待的一类并行算法。

定义 4.7 分布算法 (Distributed Algorithm) 是指由通信链路连接的多个场点 (Site) 或节点，协同完成问题求解的一类并行算法。

按照上述意义，在局网环境下进行的计算称为分布计算 (Distributed Computing)。在 Internet 流行的今天，可把工作站机群 COW (Cluster of Workstations) 环境下进行的计算称为网络计算 (Network Computing)。推而广之，有人把基于 Internet 的计算则称为元计算 (Metacomputing)。

定义 4.8 确定算法 (Deterministic Algorithm) 是指算法的每一步都能明确地指明下一步应该如何行进的一种算法。

定义 4.9 随机算法 (Randomized Algorithm) 是指算法的每一步，随机地从指定范围内选取若干参数，由其来确定算法的下一步走向的一种算法。

4.1.2 并行算法的表达

描述一个算法,可以使用自然语言进行物理描述;也可使用某种程序设计语言进行形式化描述。语言的选用,应避免二义性,且力图直观、易懂而不苛求严格的语法格式。像描述串行算法所选用的语言一样,类_Algol、Pidgin_Algol、类_Pascal等语言均可选用。在这些语言中,允许使用任何类型的数学描述,通常也无数据类型的说明部分,但只要需要,任何数据类型都可引进。

在描述并行算法时,所有描述串行算法的语句及过程调用等均可使用,而只是为了表达并行性而引入几条并行语句:

par_do语句 当算法的若干步要并行执行时,我们可使用"Do in parallel"语句,简记之为"par_do"进行描述:

```
for i=1 to n par_do
        .
        .
        .
end for
```

其中"end for"也可代之以"odrap"。

for all语句 当几个处理器同时执行相同的操作时,可以使用"for all"语句描述之:

```
for all P_i ,where 0≤ i≤ k do
        .
        .
        .
end for
```

注意,为了算法书写简洁,在意义明确的前提下,参数类型总是省去,同时语句括弧"begin…end"的使用也比较随便。

4.1.3 并行算法的复杂性度量

复杂度的渐近表示 对算法进行分析时,常使用上界(Upper Bound)、下界(Lower Bound)和紧致界(Tightly Bound)的概念,现分别定义如下:

定义4.10 令 $f(n)$ 和 $g(n)$ 是定义在自然数集合 N *上的两个函数*,如存在两个正常数 c *和* n_0,使得对于所有 $n \geq n_0$ 均有 $f(n) \leq cg(n)$,则称 $g(n)$ 是 $f(n)$ 的一个上界,记作 $f(n) = O(g(n))$。

定义4.11 $f(n)$ 和 $g(n)$ 定义如上,如存在两个正常数 c *和* n_0,使得对于所

有 $n \geq n_0$ 均有 $f(n) \geq cg(n)$，则称 $g(n)$ 是 $f(n)$ 的一个下界，记作 $f(n) = \Omega(g(n))$。

定义 4.12　$f(n)$ 和 $g(n)$ 定义如上，如存在正常数 c_1、c_2 和 n_0，使得对于所有 $n \geq n_0$ 均有 $c_1 g(n) \leq f(n) \leq c_2 g(n)$，则称 $g(n)$ 是 $f(n)$ 的一个紧致界，记作 $f(n) = \Theta(g(n))$。

图 4.1 给出了上界、下界和紧致界的几何解释。

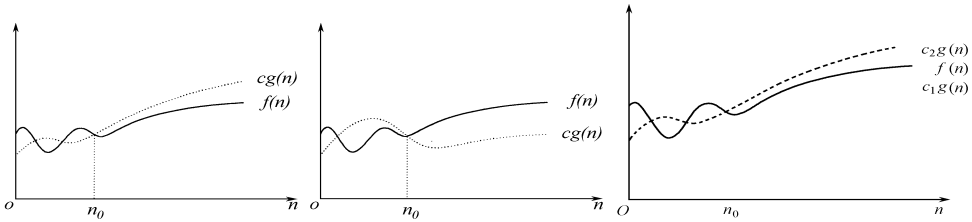


图 4.1　计算复杂性的几何解释

并行算法的复杂性度量　在分析算法时，如果对算法的所有输入分析其平均性态时的复杂度则称之为期望复杂度 (Expected Complexity)，为此往往需要对输入的分布作某种假定，这在大多数情况下并非容易。所以感兴趣的是分析在某些输入时，使得算法的时、空复杂度呈现最坏情况下的算法复杂度，称之为最坏情况下的复杂度 (Worst Case Complexity)。

在分析并行算法时，通常要分析如下几个指标：

(1) 运行时间 $t(n)$：运行时间就是算法运行在给定模型上求解问题所需的时间 (它主要是输入规模 n *的函数*)，通常包含计算时间和通信时间，分别用计算时间步和选路时间步作单位。

(2) 处理器数 $p(n)$：它是求解给定问题所用的处理器数目，通常取 $p(n) = n^{1-\varepsilon}$，$0 < \varepsilon < 1$ 是常见的。

(3) 并行算法的成本 $c(n)$：它定义为并行算法的运行时间 $t(n)$ 与其所需的处理器数 $p(n)$ 之乘积，即 $c(n) = t(n) \cdot p(n)$。

定义 4.13　如果求解一个问题的并行算法之成本，在数量级上等于最坏情况下串行求解此问题所需的执行步数，则称此并行算法是成本最优 (Cost Optimal) 的。

(4) 总运算量 $W(n)$：即并行算法所完成的总的操作数量。此时我们并不关心也不必指明算法使用了多少台处理器。当给定了并行系统中的处理器数时，就可使用下述 Brent 定理计算出相应的运行时间。

Brent 定理 (Brent's Theorem)　令 $W(n)$ 是某并行算法 A *在运行时间* $T(n)$

内所执行的运算量,则 *A 使用 p 台处理器可在 t(n) = O(W(n)/p+ T(n))* 时间内执行完毕。

W(n) 和 *c(n)* 密切相关。按照成本之定义和 Brent 定理,则有 *c(n) = t(n)· p= O(W(n) + p· T(n))*,当 *p= O(W(n)/T(n))* 时,*W(n)* 和 *c(n)* 两者是渐近一致的;而对于任意的 *p*,则 *c(n) > W(n)*。这说明一个算法在运行过程中,不一定都能充分地利用有效的处理器去工作。

4.1.4 并行算法中的同步与通信

同步 同步(Synchronization)是在时间上强使各执行进程在某一点必须相互等待。在并行算法的各进程异步执行过程中,为了确保各处理器的正确工作顺序以及对共享可写数据的正确访问(互斥访问),程序员需在算法的适当点设置同步点。同步可用软件、硬件和固件的办法来实现。下面以 MIMD_SM 多处理器系统中 *n* 个数的求和为例,说明如何用同步语句 lock 和 unlock 来确保对共享可写数据的互斥访问。假定系统中有 *p* 个处理器 $P_0,\cdots,P_{p-1}$;输入数组 $A=(a_0,\cdots,a_{n-1})$ 存放在共享存储器中;全局变量用于存放结果;局部变量 *L* 包含各处理器计算的子和;lock 和 unlock 语句执行在临界区,加锁是个原子操作;在 for 循环中各进程异步地执行各语句,并结束在"endfor"。

算法 4.1 共享存储多处理器上求和算法
输入: $A=(a_0,\cdots,a_{n-1})$,处理器数 *p*
输出: $S=\Sigma a_i$
Begin
(1) $S=0$
(2) for all P_i where $0\le i\le p-1$ do
(2.1) $L=0$
(2.2) for $j= i$ to n step p do
$L= L+a_j$
end for
(2.3) lock (S)
$S= S+ L$
(2.4) unlock (S)
end for
End

通信 通信(Communication)是在空间上对各并发执行的进程施行数据交换。通信可使用通信原语来表达:在共享存储的多处理机中,可使用 global read(*X*, *Y*)和 global write(*U*, *V*)来交换数据,前者将全局存储器中数据 *X* 读入局部变量 *Y* 中,后者将局部数据 *U* 写入共享变量 *V* 中;在分布存储的多计算机中,可使用 send(*X*, *i*)和 receive(*Y*, *j*)来交换数据,前者是处理器发送数据 *X* 给 P_i,后者是处理器从 P_j 接收数据 *Y*。下面以 MIMD_DM 多计算机系统中矩阵向量乘法为例说明之。假定连接拓扑为环。矩阵 *A* 和向量 *X* 划分成 *p* 块:$A=(A_1,\cdots,A_p)$ 和 $X=(x_1,\cdots,x_p)$,其中 A_i 的大小为 $n\times r$,x_i 的大小为 *r*。假定有 $p\le n$ 个处理器,$r=n/p$ 为一整数。为了计算 $y=AX$,先由处理器 *i* 计算 $z_i=A_ix_i$ $(1\le i\le p)$,再累加求和 $z_1+\cdots+z_p$。如果 P_i 开始在其局存中保存 $B=A_i$ 和 $w=x_i$ 则 $(1\le i\le p)$,则各处理器可局部计算乘积 Bw;然后采用在环中顺时针循环部分和的方法将这些向量累加起来;最终输出向量保存在 P_1 中。每个处理器都执行如下算法:

算法 4.2 分布存储多计算机上矩阵向量乘算法

输入:处理器数 *p*,第 *i* 个大小为 $n\times r$ 的子矩阵 $B=A(1:n,(i-1)r+1:ir)$,其中 $r=n/p$;第 *i* 个大小为 *r* 的子向量 $w=x((i-1)r+1:ir)$。

输出:P_i 计算 $y=A_1x_1+\cdots+A_ix_i$,并向右传送此结果;算法结束时,P_1 保存乘积 *AX*。

```
Begin
(1) Compute z=Bw
(2) if i=1 then y=0 else receive(y, left) endif
(3) y=y+z
(4) send(y, right)
(5) if i=1 then receive(y, left) endif
End
```

4.2 并行计算模型

所谓计算模型实际上就是硬件和软件之间的一种桥梁,使用它能够设计分析算法,在其上高级语言能被有效地编译且能够用硬件来实现。在串行计算时,冯·诺依曼机就是一个理想的串行计算模型,在此模型上硬件设计者可设计多种多样

的冯·诺依曼机而无须虑及被执行的软件;而软件工程师能够编写各种可在此模型上有效执行的程序而无须考虑所使用的硬件。但在并行计算时,尚未有一个类似于冯·诺依曼机的真正通用的并行计算模型。现在流行的计算模型要么过于简单、抽象(如 PRAM);要么过于专用(如互连网络模型和 VLSI 计算模型)。因而急需发展一种更为实用、能够较真实反映现代并行机性能的并行计算模型。本节先简要讨论一下 PRAM 模型;然后分别介绍异步 PRAM 模型、BSP 模型和 logP 模型;最后对 BSP 和 logP 进行了评注。

4.2.1 PRAM 模型

PRAM 模型描述 PRAM (Parallel Random Access Machine) 模型,即并行随机存取机器,也称之为共享存储的 SIMD 模型,是一种抽象的并行计算模型。在这种模型中,假定存在着一个容量无限大的共享存储器;有有限或无限个功能相同的处理器,且其均具有简单的算术运算和逻辑判断功能;在任何时刻各处理器均可通过共享存储单元相互交换数据。根据处理器对共享存储单元同时读、同时写的限制,PRAM 模型又可分为:①不允许同时读和同时写(Exclusive_Read and Exclusive_Write)的 PRAM 模型,简记之为 PRAM_EREW;②允许同时读不允许同时写(Concurrent_Read and Exclusive_Write)的 PRAM 模型,简记之为 PRAM_CREW。③允许同时读和同时写(Concurrent_Read and Concurrent_Write)的 PRAM 模型,简记之为 PRAM_CRCW。显然,允许同时写是不现实的,于是又对 PRAM_CRCW 模型作了进一步的约定:①只允许所有的处理器同时写相同的数,此时称为公共(Common)的 PRAM_CRCW,简记之为 CPRAM_CRCW;②只允许最优先的处理器先写,此时称为优先(Priority)的 PRAM_CRCW,简记之为 PPRAM_CRCW;③允许任意处理器自由写,此时称为任意(Arbitrary)的 PRAM_CRCW,简记之为 APRAM_CRCW。上述模型中,PRAM_EREW 是最弱的计算模型,而 PRAM_CRCW 是最强的计算模型。令 T_M 表示某一并行算法在并行计算模型 M 上的运行时间,则

$$T_{EREW} \geq T_{CREW} \geq T_{CRCW} \tag{4.1}$$

$$T_{EREW} = O(T_{CREW} \cdot \log p) = O(T_{CRCW} \cdot \log p) \tag{4.2}$$

其中,p 为处理器的数目。(4.2) 式的含义是,一个具有时间复杂度为 T_{CREW} 或 T_{CRCW} 的算法,可在 PRAM_EREW 模型上花费 $\log p$ 倍时间模拟实现之。

PRAM 模型优点 著名的 PRAM 模型有很多优点:它特别适合于并行算法的表达、分析和比较;使用简单,很多诸如处理器间通信、存储管理和进程同步等并行机的低级细节均隐含于模型中;易于设计算法和稍加修改便可运行在不同的并行

机上,且有可能在 PRAM 模型中加入一些诸如同步和通信等需要考虑的问题。

PRAM 模型缺点 PRAM 是一个同步模型,这就意味着所有的指令均按锁步方式操作,用户虽感觉不到同步的存在,但它的确是很费时的;共享单一存储器的假定,显然不适合于分布存储的异步的 MIMD 机器;假定每个处理器均可在单位时间内访问任何存储单元而略去存取竞争和有限带宽等是不现实的。

PRAM 模型推广 随着人们对 PRAM 的理解深化,在使用它的过程中也对其做了若干推广,主要的有:存储竞争模型,它将存储器分成一些模块,每个模块一次均可处理一个访问,从而可在模块级处理存储器的竞争;延迟模型,它考虑了信息的产生和能够使用之间的通信延迟;局部 PRAM 模型,此模型考虑到了通信带宽,它假定每个处理器均有无限的局存,而访问全局存储器是较昂贵的;分层存储模型,它将存储器视为分层的存储模块,每个模块由其大小和传送时间表征,多处理机由模块树表示,叶为处理器;异步 PRAM 模型,它是下一节我们要专门讨论的模型。

尽管 PRAM 模型是很不实际的并行计算模型,但在目前算法界仍被广泛使用,且被普遍地接受下来,特别是算法理论研究者非常喜欢它。

4.2.2 异步 PRAM 模型

模型特点 分相(Phase) PRAM 模型是一个异步的 PRAM 模型,简记之为 APRAM,系由 p 个处理器组成,其特点是每个处理器都有其局存、局部时钟和局部程序;处理器间的通信经过共享全局存储器;无全局时钟,各处理器异步地独立执行各自的指令;处理器任何时间依赖关系需明确地在各处理器的程序中加入同步(路)障(Synchronization Barrier);一条指令可在非确定(无界)但有限的时间内完成。

APRAM 模型中的指令类型 APRAM 模型中有四类指令:①全局读:将全局存储单元中的内容读入局存单元中;②局部操作:对局存中的数执行操作,其结果存入局存中;③全局写:将局存单元中的内容写入全局存储单元中;④同步:同步是计算中的一个逻辑点,在该点各处理器均需等待别的处理器到达后才能继续执行其局部程序。

APRAM 模型中的计算 在 APRAM 中,计算系由一系列用同步障分开的全局相所组成。如图 4.2 所示,在各全局相内,每个处理器异步地运行其局部程序;每个局部程序中的最后一条指令是一条同步障指令;各处理器均可异步地读取和写入全局存储器,但在同一相内不允许两个处理器访问同一单元。正是因为不同的处理器访问存储单元总是由一同步障所分开,所以指令完成的时间上的差异并不影响整个计算。

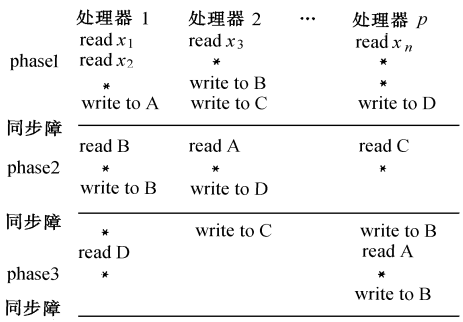


图 4.2 APRAM 中的异步计算,*表示局部操作

APRAM 模型中的时间计算 使用 APRAM 模型计算算法的时间复杂度时,假定局部操作取单位时间;全局读/写时间为 d,它定量化了通信延迟,代表读/写全局存储器的平均时间,d 随机器中的处理器增加而增加;同步障的时间为 B,它是处理器数 p 的非降函数 $B=B(p)$。在 APRAM 中假定上述参数服从如下关系:

$$2 \leq d \leq B \leq p \tag{4.3}$$

同时 $B(p) \in O(d\log p)$ 或 $B(p) \in O(d\log p/\log d)$。令 t_{ph} 为全局相内各处理器指令执行时间中最长者,则整个程序运行时间 T *为各相的时间之和加上* B *乘*以同步障次数,即

$$T = \Sigma t_{ph} + B \times \text{同步障次数} \tag{4.4}$$

总之,APRAM 模型比起 PRAM 来更接近于实际的并行机;且保留了 PRAM 编程的简捷性;由于使用了同步障,所以不管各处理器遭到的延迟多长,程序必定是正确的;且因为 APRAM 模型中的成本参数是定量化的,所以算法的分析也是不难的。

4.2.3 BSP 模型

BSP 模型的基本参数 BSP (Bulk Synchronous Parallel) 模型,字面的含义是"大"同步模型(相应地,APRAM 模型也叫作"轻量"同步模型),早期最简单的版本叫作 XPRAM 模型,它是作为计算机语言和体系结构之间的桥梁,并以下述三个参数描述的分布存储的多计算机模型:①处理器/存储器模块(下文也简称为处理器);②施行处理器/存储器模块对之间点到点传递消息的选路器;③执行以时间间隔 L 为周期的所谓路障同步器。所以 BSP 模型将并行机的特性抽象为三个定量参数 p、g、L,分别对应于处理器数、选路器吞吐率(亦称带宽因子)、全局同步之间

的时间间隔。

BSP 模型中的计算 在 BSP 模型中,计算系由一系列用全局同步分开的周期为 L 的超级步 (Superstep) 所组成。在各超级步中,每个处理器均执行局部计算,并通过选路器接收和发送消息;然后作一全局检查,以确定该超级步是否已由所有的处理器完成;若是,则前进到下一超级步,否则下一个 L 周期被分配给未曾完成的超级步。

BSP 模型的性质和特点 BSP 模型是个分布存储的 MIMD 计算模型,其特点是:①它将处理器和选路器分开,强调了计算任务和通信任务的分开,而选路器仅施行点到点的消息传递,不提供组合、复制或广播等功能,这样做既掩盖了具体的互连网络拓扑,又简化了通信协议;②采用路障方式的以硬件实现的全局同步是在可控的粗粒度级,从而提供了执行紧耦合同步式并行算法的有效方式,而程序员并无过分的负担;③在分析 BSP 模型的性能时,假定局部操作可在一个时间步内完成,而在每一超级步中,一个处理器至多发送或接收 h 条消息 (称为 h_relation)。假定 *s 是传输建立时间*,所以传送 *h 条消息的时间为* $gh+s$,如果 $gh \geq 2s$,则 *L 至少应* $\geq gh$。很清楚,硬件可将 *L 设置尽量小* (例如使用流水线或宽的通信带宽使 *g 尽量小*),而软件可以设置 *L 之上限* (因为 *L 愈大*,并行粒度愈大)。在实际使用中,*g 可定义为每秒处理器所能完成的局部计算数目与每秒选路器所能传输的数*据量之比。如果能合适地平衡计算和通信,则 BSP 模型在可编程性方面具有主要的优点,而直接在 BSP 模型上执行算法 (不是自动地编译它们),此优点将随着 g 的增加而更加明显;④为 PRAM 模型所设计的算法,均可采用在每个 BSP 处理器上模拟一些 PRAM 处理器的方法实现之。理论分析证明,这种模拟在常数因子范围内是最佳的,只要并行宽松度 (Parallel Slackness),即每个 BSP 处理器所能模拟的 PRAM 处理器的数目足够大。在并发情况下,多个处理器同时访问分布式的存储器会引起一些问题,但使用散列方法可使程序均匀地访问分布式存储器。在 PRAM_EREW 情况下,如果所选用的散列函数足够有效,则 L 至少是对数的,于是模拟可达最佳,这是因为我们欲在 p 个物理处理器的 BSP 模型上,模拟 $v \geq p\log p$ 个虚拟处理器,可将 $v/p \geq \log p$ 个虚拟处理器分配给每个物理处理器。在一个超级步内,v 次存取请求可均匀摊开,每个处理器大约 v/p 次,因此机器执行本次超级步的最佳时间为 $O(v/p)$,且概率是高的。同样,在 v 个处理器的 PRAM_CRCW 模型中,能够在 p 个处理器 (如果 $v=p^{1+\varepsilon}$, $\varepsilon>0$) 和 $L \geq \log p$ 的 BSP 模型上用 $O(v/p)$ 的时间也可达到最佳模拟。

对 BSP 模型的评注 ①在并行计算时,Valiant 试图也为软件和硬件之间架起一座类似于冯·诺依曼机的桥梁,他论证了 BSP 模型可以起到这样的作用,正是因为如此,BSP 模型也常叫作桥模型;②一般而言,分布存储的 MIMD 模型编程能力

均较差,但在 BSP 模型中,如果计算和通信可合适的平衡(例如 $g=1$),则它在可编程方面呈现出主要的优点;③在 BSP 模型上,曾直接实现了一些重要的算法(如矩阵乘积、并行前缀运算、FFT 和排序等),它们均避免了自动存储管理的额外开销;④BSP 模型可有效地在超立方网络和光交叉开关互连技术上实现,呈现出该模型与特定的工艺技术无关,只要选路器有一定的通信吞吐率;⑤在 BSP 模型中,超级步的长度必须能充分地适应任意的 h_relation,这一点是人们最不喜欢的;⑥在 BSP 模型中,在超级步开始发送的消息,即使网络延迟时间比超级步的长度短,它也只能在下一个超级步才能使用;⑦BSP 模型中的全局路障同步假定是用特殊的硬件支持的,这在很多并行机中可能没有相应的现成的硬件机构;⑧Valiant 所提出的编程模拟环境,在算法模拟时的常数可能不是很小的,如果考虑到进程间的切换(可能不仅要设置寄存器,而且可能还有部分高速缓存)则此常数可能很大。

4.2.4 logP 模型

logP 模型提出的背景 ①根据技术发展的趋势:20 世纪 90 年代末和未来的并行机发展的主流之一是巨量并行机,即 MPC(Massively Parallel Computers),它系由成千个功能强大的处理器/存储器节点,通过受限带宽的和可观延迟的互连网络所构成。所以我们建立并行计算模型应充分考虑此情况,这样基于此模型的并行算法才能在现有和未来的并行机上有效运行;②根据已有的编程经验:现有的共享存储、消息传递和数据并行等编程风范都很流行,但尚无一个公认的和占支配地位的编程方式,因此应寻求一种与上述任一特定编程风格无关的计算模型;③根据现有的理论模型:共享存储 PRAM 模型和互连网络的 SIMD 模型作为开发并行算法还不够合适,因为它们既未包含分布存储的情况,也未考虑通信同步等实际因素,从而也不能精确地反映运行在真实并行机上的算法的性态。所以在此背景下,一个以 MPC 为背景的新计算模型,即 logP 模型,便由 D. Culler 等人提出了。

logP 模型的参数 logP 模型(logP Model)是一种分布存储的、点到点通信的多处理机模型,其中通信网络由一组参数来描述,但它并不涉及到具体的网络结构,也不假定算法一定要用显式的消息传递操作进行描述。很凑巧,logP 恰好是以下几个定量参数的拼写,其中,①l(Latency)表示在网络中消息从源到目的地所遭到的延迟;②o(Overhead)表示处理器发送或接收一条消息所需的额外开销(包含操作系统核心开销和网络软件开销),在此期间内它不能进行其它的操作;③g(Gap)表示处理器可连续进行消息发送或接收的最小时间间隔;④P(Processor)表示处理器/存储器模块数。很显然,g 的倒数相应于处理器的通信带宽;而 L 和 g 反映了通信网络的容量。L,o 和 g 都可以表示成处理器周期(假定一个周期完成

一次局部操作,并定义为一个时间单位) 的整倍数。

对 logP 模型的论证 ①logP 模型充分揭示了分布存储并行机的性能瓶颈,用 L、o 和 g 三个参数刻画了通信网络的特性,但却屏蔽了网络拓扑、选路算法和通信协议等具体细节。本质上讲,通信网络是一个启动率为 g、延迟为 L、端点处理器开销各为零的流水线部件;网络的容量假定是有限的,在任何时刻至多只能有 L/g 条消息从一个处理器传到另一个处理器,且任何消息均可在有限但非确定的时间内到达目的地;在网络容限范围内,点到点的传输一条消息的时间为 $2\cdot o+L$;②尽管拓扑结构对网络性能影响很大,但 logP 模型在计算通信时间时却屏蔽了这一点,这是因为通过上千个节点的网络 (如超立方、蝶形网、网孔、胖树等) 的平均距离分析,发现它们的差别仅为 2 倍,而这种差别对整个消息传输时间的影响是很小的;③对于一个具体的并行机,由通道带宽为 w、经过 H 个跨步 (Hops) 的网络传送一个 M 位的消息所花的时间为 $T(M,H)=T_{send}+\lceil M/w\rceil+H\cdot r+T_{rev}$,其中 T_{send} 为发送开销,即第一位数据被送上网络之前处理器为网络接口准备数据的时间;T_{rev} 为接收开销,即从最后一位数据到达直到接收处理器用此数据进行处理的时间;$\lceil M/w\rceil$ 为将消息的最后一位送到网上所需的时间;$H\cdot r$ 是最后一位数据通过网络达到目的节点的时间 (r 为中继节点的时延)。对于 logP 而言,合理的参数选取是:$o=(T_{send}+T_{rev})/2$,$L=H\cdot r+\lceil M/w\rceil$,$g=\lceil M/b\rceil$ (b 为处理器对剖宽度)。此处,通过对具有上千个处理器的典型并行机的测试和分析,发现在网络空载或轻载时 $T(M,H)$ 中起主导作用的是 T_{send} 和 T_{rev} (这就意味着通信接口部件对系统性能影响更大),而它们对网络和结构却不敏感。但是如果网络重载时就会出现竞争资源的现象,从而等待时间将迅速增加,正是因为如此,logP 模型对网络的容量加以了限制;④在 logP 模型中,假定每个节点只有一个处理器,它既用于计算又负责接收和发送消息,所以为了发送或接收一个字处理器均要付出开销 o。对于长的消息,某些并行机提供了专门的硬件支持,但这样作充其量也只不过能使每个节点的性能提高一倍。所以在 logP 模型中对长消息不作特别处理;⑤尽管在某些并行机中,使用了特殊的硬件支持数据的广播、前缀运算或全局同步等,但 logP 模型中必须通过隐含地发送消息来执行这些操作,因为用硬件完成这些操作,其功能是受到了限制的 (例如它们可能只对整数有效而对浮点数则不行)。此外,对于 logP 模型设计算法时最普通的全局操作是路障,它是一种由硬件支持的原语操作。用硬件支持这一操作比对全局数据进行操作要简单,而且路障作为原语的优点是假定处理器以同步方式退出路障可简化算法的分析;⑥在 logP 模型中使用了无竞争的通信模式,因为用这种模式重复传输时可以利用整个带宽,反之其它的通信模式往往依赖于选路算法、路由缓冲器数和互连拓扑结构,而 logP 模型将网络的内部结构抽象成了几个性能参数,它就不能区别互连结构的优劣了。logP 模型能

够反映各种通信模式的一种可能的推广方式是提供多种 g,对于特定的通信模式可以采用适当的 g 进行算法分析;⑦在 logP 模型中提倡使用多线程(Multithreading)技术来屏蔽网络延迟(但此技术受通信带宽和进程切换开销的限制)。

对 logP 模型的评注 ①logP 模型将现代和将来的并行机的特性进行了精确的综合,以少量的参数 L、o、g 和 p 刻画了并行机的主要瓶颈。这个模型的详尽程度足以反映了并行算法设计时的主要实际问题,而其简捷性也足以支持详细的算法分析。对于那些非平易的算法,用这种比较复杂的模型(显然比 PRAM 复杂得多)来分析时仍是可操作的,因为这些参数的重要程度在不同的环境下是不同的,往往可以略去其中的一个或几个参数而使模型更简单一些;②logP 模型无须说明编程风格或通信协议,它可以等同地用于共享存储、消息传递和数据并行等各种风范;③logP 模型的可用性已由诸如播送、求和、FFT、LU 分解、排序、图的连通性等算法得以证实,并且它们都已在 CM_5 机器上加以实现;④事实上,如果使 logP 模型中的参数 $g=0$, $L=0$ 和 $o=0$,则 logP 就等同于 PRAM;同时 logP 模型也是 BSP 模型的改进和细化(例如在一个超级步中并非要所有的处理器都发送或接收 h 条那么多的消息,在一个超级步中消息一旦到达处理器就可立即使用它,而不必像 BSP 那样一定要等到下一个超级步);logP 模型全部采用消息同步而不像 BSP 那样要用专门的硬件支持。总之,尽管 logP 模型的可用性还有待于用大量的算法实例进一步证实,但它毕竟打开了研究模型的新途径,它不仅为算法设计者提供了设计适合于近代并行机的巨量并行算法的手段,而且对设计并行机体系结构也提供了指导性意见。

4.2.5 对 BSP 和 logP 的评注

从 BSP 到 logP BSP 把所有的计算和通信视为一个整体行为而不是一个单独的进程和通信的个体活动,它采用各进程延迟通信的办法,将诸消息组合成一个尽可能大的通信实体施行选路传输,这就是所谓的整体大同步。它简化了算法(程序)的设计和分析,当然就牺牲了运行时间,因为延迟通信意味着所有的进程均必须等待最慢者。一种改进的办法是采用子集同步(Subset Synchronization),即将所有的进程按快、慢程度分成若干个子集,于是整体的大同步就演变为子集内的同步。如果子集小到其中只包含成对的发/收者,则它就变成了异步的个体同步,这就是 logP 模型了。也就是说,如果 BSP 中考虑到个体通信所造成的开销(Overhead)而去掉路障(Barrier)同步就变成了 logP,即

$$\text{BSP}+\text{Overhead}-\text{Barrier}=\text{logP}$$

BSP 成本模型 在 BSP 的一个超级计算步中,其计算模型如图 4.3 所示。按

此可抽象出 BSP 的成本模型如下：

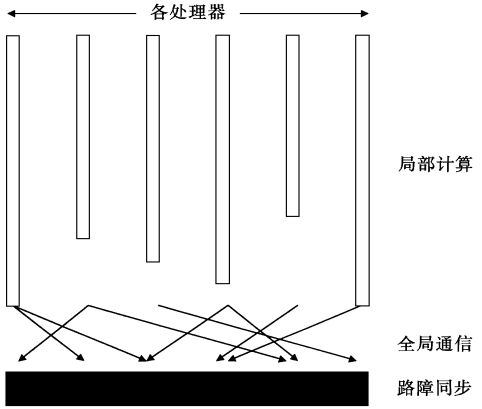


图 4.3　BSP 一个超级计算步中的计算模式

$$\text{一个超级计算步成本} = \max_{\text{Processes}} \{w_i\} + \max\{h_i g\} + L \tag{4.5}$$

其中，w_i 是进程 i（Process$_i$）的局部计算时间，h_i 是 Process$_i$ 发送或接收的最大信包数，g 是带宽的倒数（时间步/信包），L 是路障同步时间（注意，在 BSP 成本模型中并未考虑到 I/O 的传送时间）。所以，在 BSP 计算中，如果用了 s 个超级计算步，则总的运行时间为：

$$T_{\text{BSP}} = \sum_{i=0}^{s-1} w_i + g\sum_{i=0}^{s-1} h_i + sL \tag{4.6}$$

BSP 的大同步机理　BSP 模型的创始人 L.G.Valiant 曾从理论上论证并行计算不必优化在单一消息级（Single Message Level），他认为整体大同步能大大简化并行计算（算法和编程）的设计、分析、验证、性能预测和具体实现，而基于成对消息传递的个体异步并行计算（例如 logP 模型），在时间上的得益比起对计算性能上难以分析和预测来说，并不合算。目前，对 BSP 模型的质疑主要集中在两点，即延迟通信至某一特定点和频繁的路障同步，会不会造成性能下降和使成本过于昂贵。BSP 模型的支持者们对这两个问题进行了研究，回答是：延迟通信能提供更多的优化通信的机会，采用组合小的消息和全局通信调度能减少拥挤和竞争；路障同步对共享存储结构是不太费时的，而对分布存储的结构，主要是目前低层软件绝大多数都不支持访问相应的硬件，所以比较昂贵，但不管怎样，路障同步所造成的成本可折合到全局通信中而予以部分地抵销。

BSP 和 logP 相互比较 ①现今最流行的并行计算模型是 BSP 和 logP,已经证明两者本质上是等效的,且可以相互模拟:用 BSP 去模拟 logP 所进行的计算时,通常会慢常数倍,而用 logP 去模拟 BSP 所进行的计算时,通常会慢对数倍;②直观上讲,BSP 为算法(和程序)提供了更多的方便,而 logP 却提供了较好的机器资源的控制;③BSP 所引起的精确度方面的损失比起其所提供的更结构化的编程风格的优点来是小的;④总之,BSP 模型在简明性、性能的可预测性、可移植性和结构化可编程性等方面更受人欢迎和喜爱。

4.3 小结和导读

小结 并行计算模型是设计和分析并行算法的基础。PRAM 模型由于过于抽象而不能很好地反映并行算法的实际运行性能,所以研究考虑通信、同步等因素从而能较真实反映并行算法运行性能的所谓更实际的计算模型(More Realistic Computation Models)成为当今并行算法研究的主要动向之一。本节所讨论的 APRAM、BSP 和 logP 模型就是属于这种模型,此外还有 C^3 模型。但过去几年来,主要是从理论分析的角度,来研究它们之上的一些典型的并行算法的设计与分析;而近期的研究热点却从这些模型上的算法研究转向这些模型的编程的研究,即从理论研究转向实际应用。因为任何并行算法的应用都最终落实到具体的编程上,所以这种转变是顺应应用要求的。例如,一些研究者就为 BSP 模型构造了一些函数库,这些 BSP 库就是一些为程序员编写 BSP 应用程序(这些应用程序按照 BSP 的超级计算步的风格进行编写)所提供的一组简单有力的编程界面函数,此函数可改善程序的可移植性而不依赖于具体的并行机结构。尽管 PVM 和 MPI 等也是目前可供使用的开发可移植并行代码的方法,但它们的功能过于复杂而不易被掌握,且它们均没有为编程者提供能设计高效代码的成本函数,而像 BSP 模型却提供了简单和可定量分析程序运行时间的成本函数。因此研究基于这些实用计算模型的并行编程方法是非常有意义的。

最后,将本章所讨论的几种并行计算模型综合比较于表 4.1 中。

导读 关于并行算法的设计和分析,读者可阅读[191]。最基本的 PRAM 模型来源于[65];推广的 PRAM 模型有存储竞争模型[108],延迟模型[138],局部 PRAM[5],分层存储模型[11]等。关于更实际的并行计算模型综合介绍,读者可参阅[192];更为详细者,包括 APRAM[76],BSP[176],logP[55]和 C^3[87]。关于并行计算模型的定量比较,读者可参阅[106]。对 BSP 模型的质疑和回答,读者可参阅[162];有关 BSP 的函数库,读者可参阅[80,81]。如何使用 BSP 模型进行编

程,读者可参阅[32];如何使用大同步进行并行计算,读者可参阅[46]。

表 4.1　并行计算模型综合比较一览表

模型 / 属性	PRAM	APRAM	BSP	logP
体系结构	SIMD_SM	MIMD_SM	MIMD_DM	MIMD_DM
计算模式	同步计算	异步计算	异步计算	异步计算
同步方式	自动同步	路障同步	路障同步	隐式同步
模型参数	1 (单位时间步)	d, B d:读/写时间 B:同步时间	p, g, l p:处理器数 g:带宽因子 l:同步间隔	l, o, g, p l:通信延迟 o:额外开销 g:带宽因子 p:处理器数
计算粒度	细粒度/ 中粒度	中粒度/ 大粒度	中粒度/ 大粒度	中粒度/ 大粒度
通信方式	读/写共 享变量	读/写共 享变量	发送/接 收消息	发送/接 收消息
编程地址空间	全局地址空 间	单一地址空 间	单地址/多 地址空间	单地址/多 地址空间

习　题

4.1　试证明:

(1) 如果 $f(n)=10n$,则 $f(n)=O(n)$。

(2) 如果 $f(n)=\frac{1}{3}n^3+\frac{1}{2}n^2+\frac{1}{6}n$,则 $f(n)=O(n^3)$。

(3) 如果 $f(n)=n^2/5$,则 $f(n)=\Omega(n^2)$。

(4) 如果 $f(n)=0.001n^3$,则 $f(n)=\Omega(n^2)$。

4.2　试证明 $\frac{1}{2}n^2-3n=\Theta(n^2)$。

(提示:按照紧致界 $\Theta(\cdot)$ 的定义,求出满足要求的 n_0、c_1 和 c_2 之值。)

4.3　试证明 Brent 定理:令 $W(n)$ 是某并行算法 A 在运行时间 $T(n)$ 内所执行的运算数量,则 A 使用 p 台处理器可在 $t(n)=O(W(n)/p+T(n))$ 时间内执行完毕。

4.4　试分析算法 4.2 的执行过程。

4.5　假定 P_i $(1 \le i \le n)$ 开始时存有数据 d_i，所谓累加求和系指用 $\sum_{j=1}^{i} d_j$ 来代替 P_i 中的原始值 d_i。算法 4.3 给出了在 PRAM 模型上累加求和算法：

算法 4.3　PRAM_EREW 上累加求和算法
输入：　P_i 中保存有 d_i，$l \le i \le n$
输出：　P_i 中的内容为 $\sum_{j=l}^{i} d_j$

```
Begin
  for j=0 to log n−1 do
     for i=2^j+l to n par do
        (i) P_i = d_{i−2^j}
        (ii) d_i = d_i + d_{i−2^j}
     endfor
  endfor
End
```

(1) 试用 $n=8$ 为例，按照上述算法逐步计算出累加和。
(2) 分析算法 4.3 的时间复杂度。

4.6　在 APRAM 模型上设计计算时，应尽量使各处理器内的局部计算法时间和读写时间大致与同步时间 B 相当。当在 APRAM 上计算 n 个数的和时，可以借用 B 叉树求和的办法。

假定有 p 个处理器计算 n 个数的和，此时每个处理器上分配 n/p 个数，各处理器先求出自身的局和，然后从共享存储器中读取它的 B 个孩子的局和，累加后置入指定的共享存储单元 SM 中，最后根处理器所计算的和即为全和。算法 4.4 示出了 APRAM 上的求和算法：

算法 4.4　APRAM 上求和算法
输入：　n 个待求和的数
输出：　总和在共享存储单元 SM 中

```
Begin
  (1) 各处理器求 n/p 个数的局和，并将其写入 SM 中
  (2) Barrier
  (3) for k= ⌈log_B (p(B−1)+1)⌉−2 downto 0 do
        (3.1) for all P_i, 0≤ i≤ p−1, do
                if P_i 在第 k 级 then
                  P_i 计算其 B 个孩子的局和并与其自身局和相加，然后将结果写入 SM
                  中
                endif
```

```
            endfor
        (3.2) Barrier
    end for
End
```

(1) 试用 APRAM 模型之参数,写出算法的时间复杂度函数表达式。

(2) 试解释 Barrier语句的作用。

4.7 欲在 BSP 模型上计算 n 个数的和,可以在 *d 叉树上进行*。假定用 *p 个处理器求 n 个数的和*,则每个处理器分配有 *n/p 个数*。首先,各处理器求 *n/p 个数的局和*;然后在 *d 叉树上*自下而上求全和,其全过程如算法 4.5 所示。

算法 4.5 BSP 上求和算法

输入: *n* 个待求的数

输出: 总和在根处理器 P_0 中

```
Begin
  (1) for all Pi,0≤ i≤ p-1 do    /*各处理器求各自局和*/
        (1.1) Pi 计算 n/p 个数的局和
        (1.2) if Pi 在第 ⌈log_B (p(B-1)+1)⌉-1 级 then
                Pi 将其局和发往父节点
              endif
      endfor
  (2) Barrier
  (3) for k= ⌈log_B (p(B-1)+1)⌉-2 downto 0 do  /*上播并求和*/
        (3.1) for all Pi,0≤ i≤ p-1 do
                if Pi 在第 r 级 then
                    Pi 接收 d 个孩子消息,并将它们与其本身局和相加;然后将结果
                    发往父节点
                endif
              endfor
        (3.2) Barrier
      endfor
End
```

(1) 试分析算法 4.5 的时间复杂度。

(2) *d* 值如何确定?

4.8 在给定时间 *t* 内,尽可能多的计算输入值的和也是一个求和问题。如果在 logP 模型上求此问题时,要是 $t < L+2\cdot o$,则在一个单处理机上即可最快地完成;要是 $t > L+2\cdot o$ 时,则根处理器应在 $t-1$ 时间完成局和的接收工作,然后用一个单位时间完成加运算而得最

终的全和。而根的远程子节点应在 $(t-1)-(L+2\cdot o)$ 时刻开始发送数据，其兄妹子节点应依次在 $(t-1)-(L+2\cdot o+g)$，$(t-1)-(L+2\cdot o+2g)$，…时刻开始发送数据。图 4.4 示出了 $t=28, p=8, L=5, o=2, g=4$ 的 logP 模型上的通信（即发送/接收）调度树。试分析此通信调度树的工作原理和图中节点中的数值是如何计算的？

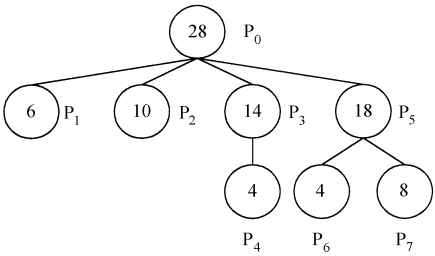


图 4.4　$t=28, p=8, L=5, o=2, g=4$ 的通信调度树

4.9　按照图 4.4 通信调度树，在 logP 模型上的处理器进行连续接收和发送时必须相间时间间隔 g，但可以用通信开销 o 和计算局和的时间来填充，从而可掩盖 g 的开销。一般而言，对于某处理器，若它有 k 个子节点，则它必须接收 k 个消息，所以至少要 $k(g-o)$ 次局部加法来填充所有的 g，因此在它接收 k 个消息中，至少要作 $k(g-o-1)$ 次自身内部数的加法来填充，这样才能充分掩盖 g 的开销。图 4.5 示出了按照图 4.4 所示的通信调度树时的计算时间调度图。试分析此计算调度图的工作原理和处理器 P_0 与 P_5 填充计算情况。

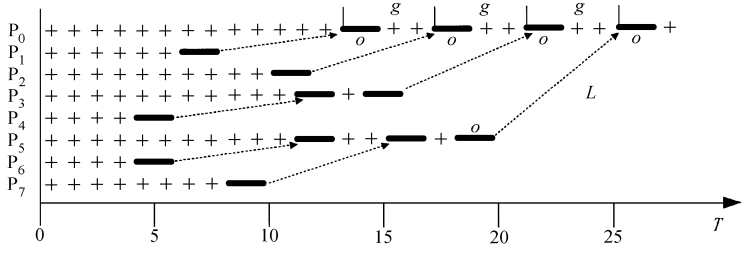


图 4.5　$t=28, p=8, L=5, o=2, g=4$ 的计算调度图

4.10　试分析如下用 BSPLib 并行求 4 个整数 1,2,3,4 的前缀和的过程（参见图 4.6）。

```
//*用 BSPLib 求前缀和*//
 #include "bsp.h"
int bsp_allsums(int x) {
  int i,left,right;
```

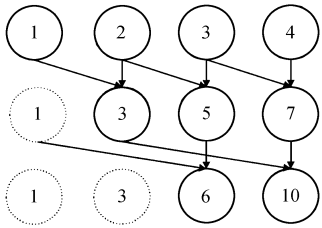


图 4.6　求整数 1,2,3,4 的前缀和过程

```
    bsp_push_reg(&right,sizeof(int));
    bsp_push_reg(&letf,sizeof(int));
    bsp_sync();

    right=x
    for(i=1;i<bsp_nprocs();i*=2){
      if(bsp_pid()+i<bsp_nprocs())
        bsp_put(bsp_pid()+i,&right,&left,0,sizeof(int));
      bsp_sync();
      if(bsp_pid()>=i) right=left+right;
    }
    bsp_pop_reg(&right);
    bsp_pop_reg(&left);
    return right;
}
void main(){
    bsp_begin(bsp_nprocs());
      printf("On %d sum is %d\n",bsp_pid(),bsp_allsums(1+bsp_pid()));
bsp_end();
}
```

第五章　并行算法的一般设计策略

设计并行算法一般有三种策略：①检测和开拓现有串行算法中的固有并行性而直接将其并行化，它虽不是对所有问题总是可行的，但对很多应用问题仍不失为一种有效的方法；②从问题本身的描述出发，根据问题固有的属性，从头开始设计一个全新的并行算法，它虽有一定的难度，但所设计的并行算法通常是高效的；③借用已有的并行算法使之可求解新的一类问题，这对于一个初学者来说似乎有一定的难度，但对一个有经验的算法设计者来说有可能产生一个很优秀的并行算法。但是，并行算法的设计是复杂的且在发展中，目前尚无一套普遍适用的系统的设计策略。

5.1 串行算法的直接并行化

5.1.1 设计策略描述

长期以来人们积累了大量优秀的串行算法,这是一批宝贵的财富。在设计并行算法时,要充分利用现有求解问题的串行算法,在其基础上而直接并行化,这显然是设计并行算法时一种优先考虑的策略。但使用它时有两点需要考虑:①对于一类具有内在顺序性的串行算法,则恐难于直接并行化。例如在某串行算法执行步中,每一步都要用到上一步的计算结果,则显然这样的串行算法步无法并行执行;②并非任何优秀的串行算法都可以产生好的并行算法,相反一个不好的串行算法则有可能产生很优秀的并行算法。例如,枚举串行算法,显然是一种复杂度高的被视为不好的串行算法类,但其直接并行化所产生的并行算法则可能获得低复杂界的优秀的并行算法。所以,并不是说一个优秀的串行算法并行化后总能产生相应的好的并行算法。最后也须说明,直接并行化也并非想象的那样简单、直接。直接并行化的关键步是分析数据执行时的相关性以及检测算法结构的固有串行性。有时为了并行化,将原串行算法不作本质性修改也是需要的,例如调整执行顺序,复制共享变量等。

大家知道,科学和工程问题中的数值计算问题,依据其数值分析数学原理而产生了很多广泛使用的串行算法,在设计这些问题的并行算法时大多都是采用串行算法直接并行化的办法,这样诸如算法的稳定性、收敛性等复杂问题,在直接并行化后的算法中均无需考虑,这恐怕也是为什么现今的数值并行算法大都是已有串行算法直接并行化的结果的原因吧。

有关并行数值算法,本书专辟了第三篇进行了讨论,读者在学习后就会发现,在那些章节里的很多并行数值问题的算法都是已有的串行算法直接并行化的结果。下面举一个非数值串行算法直接并行化的例子。

5.1.2 快排序算法的并行化

串行快排序 快排序(Quicksort)是常用的串行排序算法之一,虽然其最坏情况下的复杂度为 $O(n^2)$,但其具有较好的平均复杂度 $\Theta(n\log n)$。快排序是基于分治策略的递归排序方法。其算法分为两步:首先将一个序列 $(A_q,\cdots,A_r)$ 分成两

个非空子序列 $(A_q, \cdots, A_s)$ 和 $(A_{s+1}, \cdots, A_r)$，其中前一个子序列中的任一元素都小于等于后一个子序列中的任意元素；然后对该两子序列施行递归调用。此递归调用过程直至子序列中仅有两个元素为止。

如何按上述要求将一个序列划分成两个子序列？通常的办法是从 $(A_q, \cdots, A_r)$ 中选择一划分元素 x，也称之为主元 (Pivot)，由其将原序列划分成满足上述要求的两个子序列，选取此元素 x 最简易的办法就是将 $(A_q, \cdots, A_r)$ 中的第一个元素作为主元。下面给出串行快排序的形式描述 (算法 5.1)：

算法 5.1 SISD 上快排序算法
输入：无序序列 $(A_q, \cdots, A_r)$
输出：有序序列 $(A_q, \cdots, A_r)$

```
Procedure QUICKSORT (A, q, r)
Begin
  if q < r then
    (1) x = A_q
    (2) s = q
    (3) for i = q+1 to r do
          if A_i ≤ x then
            (i) s = s+1
            (ii) swap(A_s, A_i)
          endif
    (4) swap (A_q, A_s)
    (5) QUICKSORT (A, q, s)
    (6) QUICKSORT (A, s+1, r)
  endif
End
```

对于长度为 n 的序列，在最坏情况下所划分的两个子序列长度分别为 $n-1$ 和 1，相应的运行时间为 $t(n) = t(n-1) + \Theta(n)$，其解为 $t(n) = \Theta(n^2)$；理想的情况是所划分的两个子序列等长，相应的运行时间为 $t(n) = 2t(n/2) + \Theta(n)$，其解为 $t(n) = \Theta(n \log n)$。

快排序的并行化　分析算法 5.1，一种很自然的并行化方法是并行地调用快排序对两个所划分的子序列进行快排序 (算法 5.1 的第 (5) 和第 (6) 步)。这种并行化的方法并不改变串行算法本身的属性，所以将其很容易改写成并行执行的形式。但是用 n 个处理器排序 n 个数，如用一个处理器将 $(A_1, \cdots, A_n)$ 划分成 $(A_1, \cdots,$

A_i) 和 (A_{s+1},…,A_n) 是不会得到成本最优算法的,因为单是划分时间就是 $\Omega(n)$,所以 $C(n)=p(n)\cdot t(n)=\Omega(n^2)$。可见只有将划分步 (算法 5.1 的第 (3) 步) 也进行并行化才有可能得到成本最优的算法。下面给出基于二叉树的并行选主元的 PRAM_CRCW 模型上的快排序算法。

PRAM_CRCW 上快排序算法　执行快排序可以看成是构造一棵二叉树,其中主元是根,小于等于主元的元素处于左子树,大于主元的元素处于右子树。算法首先从选第一个主元开始,它划分原序列为两个子序列;然后相继子序列中主元的选取则可并行执行。当这样的二叉树造好后,采用中序遍历的方法就可得到一个有序序列。令待排序的序列为 (A_1,…,A_n),处理器 P_i 保存元素 A_i,$LC[1:n]$ 和 $RC[1:n]$ 分别记录给定主元的左儿子和右儿子,f_i 中存有其元素是主元的处理器号。开始时所有处理器均将它们的处理器号写向变量 root,但根据 CRCW 模型原理,最终只有一个处理器号写入变量 root。A_{root} 中是第一个主元,并且 root 被复制给每个处理器 i 的 f_i,然后那些其元素小于 A_{f_i} 的处理器将其号码写入 LC_{f_i},而大于 A_{f_i} 的处理器将其号码写入 RC_{f_i}。因此所有那些其元素属于小序列的处理器便将其号码写入 LC_{f_i},而所有那些其元素属于大序列的处理器便将其号码写入 RC_{f_i}。但是因为并发写操作只有两个值 (一个对应于 LC_{f_i},另一个对应于 RC_{f_i}) 能被写进这些单元,所以这两个值就变成了下次迭代所需的主元的处理器号。按此算法一直继续到 n 个主元被选完。

算法 5.2　PRAM_CRCW 上为快排序构造二叉树算法
输入：序列 (A_1,…,A_n) 和 n 个处理器
输出：供快排序用的一棵二叉树
Begin
(1) for each processor i do
　(1.1) root=i
　(1.2) f_i=root
　(1.3) $LC_i=RC_i=n+1$
　endfor
(2) repeat for each processor $i\neq$root do
　if ($A_i<A_{f_i}$) $\vee$ ($A_i=A_{f_i}\wedge i<f_i$) then
　(2.1) $LC_{f_i}=i$
　(2.2) if $i=LC_{f_i}$ then exit else $f_i=LC_{f_i}$ endif

```
            else
        (2.3)  RC_{f_i} = i
        (2.4)  if  i = RC_{f_i} then exit else f_i = RC_{f_i} endif
        endif
    end repeat
End
```

在算法每次迭代时,可在 $\Theta(1)$ 时间内构造一级树,而树高为 $\Theta(\log n)$,所以算法 5.2 的时间复杂度为 $\Theta(\log n)$。

5.2 从问题描述开始设计并行算法

对于有些问题,现有的串行算法恐难直接将其并行化,此时要从问题的描述出发,寻求可能的新途径,设计出一个新的并行算法。但这并不是说完全排除某些串行算法设计的基本思想,而是更着重从并行化的具体实现上开辟新的设计方法。本节拟以串匹配问题为例,来阐述这种设计策略。

5.2.1 串匹配算法

定义 5.1 由字符集 (字符的非空有穷集合) 中的字符所组成的任何有穷序列称之为串 (String)。串中包含的字符个数称为串的长度。

定义 5.2 给定长度为 n 的正文串 text 和长度为 m 的模式串 pat ($m \leq n$),欲找出 pat 在 text 中出现的所有位置 i,如果找到就称之为匹配。

在以下的讨论中,约定 pat 和 text 均以数组形式表示。按此,pat 出现在 text 的 i 位置,指的是 pat=text[i: $i+m-1$],此时则定义 match(i)=true,否则定义 match(i)=false。在下文中,为了书写方便,常将 text 和 pat 也分别写作 T 和 P。

串匹配 (String Matching) 问题在诸如文字和图像处理中经常使用,是复杂性理论中最广泛研究的问题之一。串匹配问题的一种平易算法是把它视为以 pat 为键的搜索问题,即长度为 n 的 text 可分成 $n-m+1$ 个长度为 m 的子串,检查各个子串是否与长度为 m 的 pat 相匹配。这样在最坏情况下的时间复杂界为 $(n-m+1)m=O(mn)$,所以这种古朴的串匹配算法是很少使用的。后来 Knuth、Morris 和 Pratt 三人提出了一个线性时间的经典的串匹配算法,通称之为 KMP 算法。但此算法一直没有发表,直至 Boyer 和 Moore 两人也提出了设计串匹配的新

算法时,这两种算法才同时发表,而相应地称后一种算法为 BM 算法。

目前已知的线性时间的串匹配算法均不易直接并行化,但参照串行算法的实质,结合使用串的周期数学性质却可开发出一些高效的并行串匹配算法。第一个最优的并行串匹配算法是由 Galil 提出,在 PRAM_CRCW 模型上,达到了 $t(n)=O(\log n)$ 和 $p(n)=n/\log n$ 的复杂界。后来 Vishkin 改进了 Galil 的算法,放宽了算法对字符集 Σ 大小是固定的要求。可是由于 Vishkin 的并行算法十分复杂,所以本节只是着重介绍其设计思路(有兴趣的读者可参阅[191])。但在介绍这些内容之前,先来讨论一下 KMP 算法,以说明将其直接并行化的困难所在。

*5.2.2 KMP串行串匹配算法

定义 5.3 令串 Y 的长度为 m,如果 $Y=X^kX'$,其中 X^k 是 X 本身重复 k 次(k为正整数),X' 是 X 的前缀(Prefix),则 Y 的子串 X 叫作 Y 的周期(Period)。串 Y 的周期也就是 Y 的最短周期。如果串 Y 的周期长度 p 满足 $p\leq m/2$,则称串 Y 是周期的(Periodic)。

注意,在研究串匹配时,常将整个模式串(Pattern String) P 本身当作正文串(Text String),而其从首字符起的子串当作模式串,这样我们如令 $D(j)$ 是前缀 $P[1:j-1]$ 的周期,其中 $D(1)=1$,$2\leq j\leq m+1$,且 $P(m+1)$ 是 P 中不出现的字符,则可定义模式串的失效函数(Failure Function) F 如下:

$$\begin{aligned} &F(1)=0 \\ &F(j)=j-D(j)\quad ,\quad 2\leq j\leq m+1 \end{aligned} \tag{5.1}$$

其几何意义如图 5.1 所示,左图是为了计算失效函数的起始情况,右图是计算的一般步骤。当 $P(j)=P(k)$ 时,前移指针 j 和 k 且置 $F(k)=j$;当 $P(j)\neq P(k)$ 时,则向右滑动,上面的模式使得 $F(j)$ 和 k 对准,此时的 F 就是失效函数。

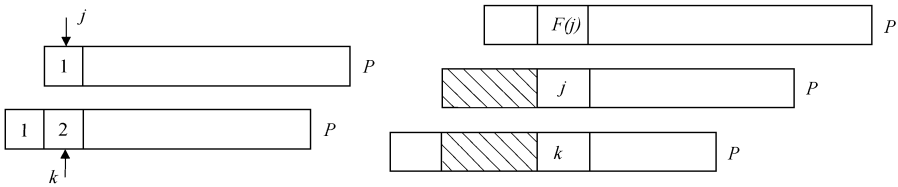


图 5.1 失效函数的确定

可以设计如下算法来计算失效函数:

算法 5.3 失效函数的计算算法
输入： 模式串 $P[1:m]$
输出： 失效函数 $F(k)$，$1 \leq k \leq m+1$
Begin
(1) $F(1)=0$；$F(2)=1$；$k=2$；$j=1$
(2) while $k \leq m$ do
(2.1) if $P(k)=P(j)$ then
(i) $j=j+1$
(ii) $k=k+1$
(iii) $F(k)=j$
endif
(2.2) if $P(k) \neq P(j)$ then
(i) $j=F(j)$
(ii) if $j=0$ then
$k=k+1$
$j=1$
$F(k)=1$
endif
endif
endwhile
End

显然计算 $P[1:m]$ 的失效函数可在 $O(m)$ 时间内完成。

有了失效函数的定义，就可以讨论 KMP 算法了：设想模式 P 置于正文 T 之上，令 P 向右滑动，用两指针 j 和 k 分别指示字符 $P(j)$ 和 $T(k)$ 的现行位置。开始时，$j=k=1$，然后分两种情况讨论：①当 $P(j)=T(k)$ 时，两指针前进 1，且如果 $j=m+1$，则表示已找到匹配，置 match $(k-m)=1$；②当 $P(j) \neq T(k)$ 时，表示现行位置不匹配，应向右滑动 P_j 位，其下一可能的位置由如上所述的 $D(j)$ 决定，即令 $j=j-D(j)$。所以 KMP 算法的关键就是当模式与正文不匹配时，应如何利用已得到的“部分匹配”的结果，将模式向右滑动一段距离后再施行继续比较，这一距离可由前面所讨论的失效函数 $F(j)$ 来决定。

算法 5.4 KMP 串匹配算法
输入： $T[1:n]$，$P[1:m]$，$F[1:m+1]$

```
输出： match(i)
Begin
  (1) j=1;k=1
  (2) while k-j≤n-m do
      (2.1) if T(k)=P(j) then
              (i) k=k+1
              (ii) j=j+1
              (iii) if j=m+1 then match(k-m)=1 endif
            endif

      (2.2) if T(k)≠P(j) then
              (i) j=F(j)
              (ii) if j=0 then
                     k=k+1;
                     j=1 endif
            endif
    endwhile
End
```

显然算法 5.4 的时间复杂度为 $O(n)$。

KMP 算法的精髓在于使用了失效函数，使用它来调整两指针 j 和 k。这种指针来回移动的办法并不太容易并行化，因为并行算法设计的基本策略是试图将 T 串分段并行处理，但上述的 KMP 顺序算法很难有效地使用在分段并行处理中。

5.2.3 并行串匹配算法的设计思路

设计并行串匹配算法的出发点 设计并行串匹配算法应从何处入手呢？为此我们要从问题的描述出发，研究串匹配的基本性质。实际上，两串是否能匹配，是与串的自身前缀有关，这种前缀特性就是串的周期性 (Periodicity)。所以研究串的数学性质 (即周期性) 是寻找并行化的一种可能的途径。

研究并行串匹配的基本方法与步骤 既然串的周期特性对研究匹配是至关重要的，那么用什么量来表征它呢？为此首先引入了失配见证函数 (Witness Function)，其定义如下：

定义 5.4 对于给定的 $j(1\leq j\leq m/2)$，如果 $P[j:m]\neq P[1:m-j+1]$ 则存在

某个 $w(1\le w\le m-j+1)$ 使得 $P(w)\neq P(s)$，其中 $s=j-1+w$，记 WIT$(j)=w$。

于是，可以根据 WIT(j) 函数来区分串是周期的还是非周期的：①对于所有 $2\le j\le m/2$，当且仅当 WIT$(j)\neq 0$ 时则串是非周期的；②对于所有 $2\le j\le m/2$，若存在某个 j，使得 WIT$(j)=0$，则串是周期的。

一旦确定了串的周期特性，则可先研究非周期串(Aperiodic String)的匹配，然后在此基础上再研究周期串的匹配。

对于非周期串匹配的研究，就是如何利用已计算出的 WIT(j) 快速地找出 P 在 T 中匹配的位置。为了减少 P 与 T 的比较次数，引入了一种竞争(Duel)函数 duel(p,q) 的概念，即当模式在某一位置 p 匹配时，则在另一位置 q 一定不匹配，这样就排除了 q 位置。可以设计一个算法来计算 duel(p,q)(参见习题 5.8)。在 duel(p,q) 算法中，参数 p,q 与 n,m 有关，且 p 与 q 的选取应先从小范围，逐步到大范围，且在每个限定范围内可并行地求 duel(p,q)，以确定竞争的幸存者。这样整个过程就像分组比赛一样，逐渐淘汰小组中的获胜者，最终只可能保留少数几个幸存者，它们就是些可能的匹配位置号码，最后再进行一次正确性验证即可。

并行串匹配算法举例　为了更好地理解上述思路与步骤，现举一例说明之。

例 5.1　令 $T=$abaababaabababababaabaaba$(n=23)$，$P=$abaababa$(m=8)$。由所计算的模式 P 的 WIT(1)=0，WIT(2)=1，WIT(3)=2，WIT(4)=4 可知 P 是非周期串。为了计算此非周期串与 T 的匹配情况，先要计算 P 相对于 T 的 WIT[1:$n-m+1$]之值。然后由其来计算 duel(p,q) 之值。计算 duel(p,q) 时，将 T 与 P 由小到大划分成一些大小为 $(2^1,2^2,\cdots)$ 的块，在相同大小的各块内并行地计算 duel(p,q) 的值。其过程为：先将 P 与 T 各自划分成大小为 2 的一些块，这样模式块(ab)与正文块(ab)(aa)(ba)(ba)(ab)(ab)(aa)(ba)进行匹配，可知在位置 1，4，6，8，9，11，14 和 16 出现匹配(即 duel(p,q) 的获胜者)；再将 P 与 T 各自划分成大小为 4 的一些块，这样模式块(abaa)与正文块(abaa)(baba)(abab)(aaba)进行匹配，可知在位置 1，6，11 和 16 出现匹配，而位置 4，8，9 和 14 被淘汰；最后需用模式(ababbaba)在正文的位置 1，6，11 和 16 进行匹配检查，以验证其正确性。□

至于周期串的匹配，情况更复杂，在此不再讨论，感兴趣的读者可参阅文献[19]。

5.3　借用已有算法求解新问题

5.3.1　设计策略描述

所谓“借用法”是指借用已知的某类问题的求解算法来求解另一类问题。此两

类问题表面上看是迥然不同的,或似乎是互不相干的,所以按照常规的办法,求解这两类问题的算法似乎也毫无"亲缘"关系,因而一般初学者难以设法通过某种内在关系,将两类不同的问题在求解方法上统一起来。这实质上也正是借用策略的难点所在。"借用"不是毫不费力的直接拿来使用,相反地,当欲借用时,不但要从问题求解方法的相似性方面仔细观察,寻求问题解法的共同点;而且所借来的方法要用得合算,效率要高,从而能达借用的目的。显然这并非易事。一个好的借用策略所设计的算法,往往给人们带来深刻的印象,常常被教科书作为范例加以引用。

借用策略虽无一般规律可循,但往往从求解问题的数学方法上能得到某种启示。例如,求一个有向图的传递闭包(Transitive Closure)问题可以使用布尔矩阵乘法来实现,其方法如下:

假定 A 是一个 n 点有向图 G 的 n 阶布尔邻接矩阵,其矩阵元素 a_{ij} 为 1,当且仅当有向图中从顶点 i 到顶点 j 之间有一条边时。所谓传递闭包,记之为 A^+,实际上也是一个 n 阶布尔矩阵,其矩阵元素 b_{ij} 为 1,当且仅当:①从 i 到 j 存在一条有向边;或②对于某一顶点 k,存在有向边 i 到 k 和 k 到 j;或③ $i=j$。按此,根据上述定义,可利用布尔矩阵乘法来求传递闭包。事实上,如令 I 为单位矩阵(仅对角元素为 1),则 $(A+I)^1=1$,表示当且仅当 i 和 j 之间的路径长度为 0(即 $i=j$),或为 1(i 和 j 之间有一条边);$(A+I)^2=1$,表示当且仅当 i 和 j 之间的路径长度为 2 或小于 2;$((A+I)^2)^2=1$,表示当且仅当 i 和 j 之间的路径长度为 4 或小于 4,……因此作 $\log n$ 次 $(A+I)$ 自乘就可求得传递闭包 A^+,因为对于 n 点有向图,i 和 j 之间若有一路径存在,则其长度至多为 n。

下面将介绍一种利用矩阵乘法求所有点对间的最短路径方法,它是一种比较典型的借用已有算法来求解另一类问题的并行算法设计策略。

5.3.2 利用矩阵乘法求所有点对间最短路径

最短路径问题有两类:一类是单源最短路径(Single Source Shortest Path),即在一个图中寻找从一个指定顶点到所有其它顶点之间的最短路径;另一类是所有点对间最短路径(All Pairs Shortest Paths),仅讨论后者。

定义 5.5 假定一有向图各边赋于非负整数权,那么一条路径的长度就是沿该路径所有边权之和;而最短路径问题就是对每一点对 i 和 j,求找其间任何最短长度的路径。

矩阵乘法在求点对间最短路径中的应用 对于一个 n 个顶点的加权有向图 $G(V,E)$,其权矩阵由 $W_{n\times n}$ 表示。为了计算其所有顶点对之间的最短路径,可以先构造一个 $n\times n$ 的矩阵 D,使得对于所有的 i 和 j,d_{ij} 是从 v_i 到 v_j 的最短路径长

度。只要 G 中无负长度的环,可以假定 W 有正、负或零元素。

令 d_{ij}^k 表示从 v_i 到 v_j,其间经过至多 $k-1$ 个中间顶点时的最短路径长度。因此 $d_{ij}^1 = w_{ij}$。特别是从 v_i 到 v_j 无边存在($i \neq j$)时,则 $d_{ij}^1 = \infty$。同样 $d_{ii}^1 = 0$。因 G 中不存在负权的环,所以 $d_{ij} = d_{ij}^{n-1}$。

为了计算 d_{ij}^k ($k > 1$),可以使用组合最优原理(Combinatorial Optimization Principle):

$$d_{ij}^k = \min_l \{d_{il}^{k/2} + d_{lj}^{k/2}\} \tag{5.2}$$

也就是说,d_{ij}^k 取所有 l 条路径($d_{il}^{k/2} + d_{lj}^{k/2}$)中的最短者。因此矩阵 D 就可以从 D^1 逐次计算 $D^2, D^4, \cdots, D^{n-1}$,然后取 $D = D^{n-1}$ 而求得。为了从 $D^{k/2}$ 计算 D^k,可以使用标准的矩阵乘法,只是将原矩阵乘法中的"×"操作代之以(5.2)式中的"+"操作;而将原矩阵乘法中的"+"操作代之以(5.2)式中的"min"操作,这样的操作共执行 $\lceil \log(n-1) \rceil$ 次。这就是借用法中最关键之处,如果没有观察和联想到这一点,则就难以做到巧妙的借用。

SIMD_CC 模型上求所有点对间最短路径算法 因为本节重点是介绍借用矩阵乘法来求所有点对间的最短路径,所以有关矩阵乘法的算法就不介绍了,只是在主算法中以调用的形式来使用。关于 SIMD_CC 的矩阵乘法,读者可参阅本书第九章第9.4.4节的 DNS 乘法。

如第9.4.4节所述那样,假定 n^3 个处理器排成 $n \times n \times n$ 的立方体,每个处理器有 $A(i,j,k)$、$B(i,j,k)$ 和 $C(i,j,k)$ 三个寄存器。开始时 $A(0,j,k) = w_{jk}$ $(0 \leq j,k \leq n-1)$ 即图的权矩阵开始时存放在 A 寄存器中;算法执行中将调用算法9.6 DNS乘法,按照上述借用方法来计算所有点对间的最短路径。算法结束时,$C(0,j,k)$ 中就是 v_j 到 v_k 的最短路径。

算法5.5 SIMD_CC 上求所有点对间最短路径算法
输入: $A(0,j,k) = w_{jk}$, $0 \leq j,k \leq n-1$
输出: $C(0,j,k)$ 中是 v_j 到 v_k 的最短路径长度, $0 \leq j,k \leq n-1$
Begin
(1) /*构筑矩阵 D^1,并将其存入 A 和 B 寄存器中*/
for j=0 to $n-1$ par-do
for k=0 to $n-1$ par-do
(1.1) if $j \neq k$ & $A(0,j,k) = 0$ then
$B(0,j,k) = \infty$
endif
else

```
                (1.2)  B(0,j,k) = A(0,j,k)
            end for
        end for
    (2) /*调用算法9.6构筑矩阵 D^2, D^4, …, D^(n-1)*/
        for i=1 to ⌈log(n-1)⌉ do
            (2.1) DNS MULTIPLICATION (A,B,C)  /*调用算法9.6*/
            (2.2) for j=0 to n-1 par-do
                      for k=0 to n-1 par-do
                          (i)  A(0,j,k) = C(0,j,k)
                          (ii) B(0,j,k) = C(0,j,k)
                      end for
                  end for
        end for
End
```

显然，上述算法的时间复杂度为 $O(\log^2 n)$ 次，因为算法第 (2) 步重复 $O(\log n)$ 次，每次乘法时间也为 $O(\log n)$，而 $p(n) = O(n^3)$。

为了更好地理解上述算法，兹举一例如下。

例 5.2　试求图 5.2(a) 所示的有向加权图中所有点对间的最短路径，其步骤如下：

① 由图 5.2(a) 构筑邻接权矩阵如图 5.2(b) 所示。

② 算法开始前，用 w_{jk} 之值初始化 $A(0,j,k)$：$A(0,0,0)=0$, $A(0,0,1)=4$, $A(0,0,2)=1$, $A(0,0,3)=0$, $A(0,0,4)=7$, $A(0,0,5)=0$, $A(0,0,6)=0$, $A(0,1,0)=0$, $A(0,1,2)=8$, $A(0,1,3)=0$, $A(0,1,4)=0$, $A(0,1,5)=0$, $A(0,1,6)=0$, …, $A(0,6,5)=1$, $A(0,6,6)=0$。

③ 由算法第 (1) 步计算出 $B(0,j,k)$，它就是距离矩阵 D^1 (如图 5.2(c) 所示)。

④ 由算法第 (2) 步的第 1 次循环计算 $D^1 \times D^1 = D^2$，其中 d^2_{jk} 各元素计算如下：

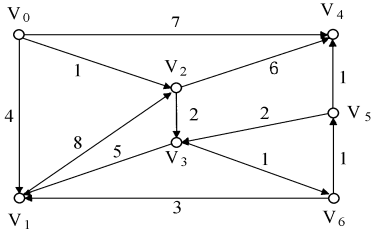
$$d^2_{00} = \min_{0 \le l \le 6} \{d_{00}+d_{00}, d_{01}+d_{10}, d_{02}+d_{20}, d_{03}+d_{30}, d_{04}+d_{40}, d_{05}+d_{50}, d_{06}+d_{60}\} = 0$$

$$d^2_{01} = \min_{0 \le l \le 6} \{d_{00}+d_{01}, d_{01}+d_{11}, d_{02}+d_{21}, d_{03}+d_{31}, d_{04}+d_{41}, d_{05}+d_{51}, d_{06}+d_{61}\} = 4$$

…

$$d^2_{66} = 0$$

D^2 的整个矩阵如图 5.2(d) 所示。



(a) 有向加权图

	0	1	2	3	4	5	6
0	0	4	1	0	7	0	0
1	0	0	8	0	0	0	0
2	0	0	0	2	6	0	0
3	0	5	0	0	0	0	1
4	0	0	0	0	0	0	0
5	0	0	0	2	1	0	0
6	0	3	0	0	0	1	0

(b) 邻接权矩阵

	0	1	2	3	4	5	6
0	0	4	1	∞	7	∞	∞
1	∞	0	8	∞	∞	∞	∞
2	∞	∞	0	2	6	∞	∞
3	∞	5	∞	0	∞	∞	1
4	∞	∞	∞	∞	0	∞	∞
5	∞	∞	∞	2	1	0	∞
6	∞	3	∞	∞	∞	1	0

(c) D^1

	0	1	2	3	4	5	6
0	0	4	1	3	7	∞	∞
1	∞	0	8	10	14	∞	∞
2	∞	7	0	2	6	∞	3
3	∞	4	13	0	∞	2	1
4	∞	∞	∞	∞	0	∞	∞
5	∞	7	∞	2	1	0	3
6	∞	3	11	3	2	1	0

(d) D^2

	0	1	2	3	4	5	6
0	0	4	1	3	7	5	4
1	∞	0	8	10	14	12	11
2	∞	6	0	2	5	4	3
3	∞	4	12	0	3	2	1
4	∞	∞	∞	∞	0	∞	∞
5	∞	6	14	2	1	0	3
6	∞	3	11	3	2	1	0

(e) D^4

	0	1	2	3	4	5	6
0	0	4	1	3	6	5	4
1	∞	0	8	10	13	12	11
2	∞	6	0	2	5	4	3
3	∞	4	12	0	3	2	1
4	∞	∞	∞	∞	0	∞	∞
5	∞	6	14	2	1	0	3
6	∞	3	11	3	2	1	0

(f) D^8

图 5.2 利用矩阵乘法求所有点对间最短路径

⑤由算法第(2)步的第2次迭代计算 $D^2 \times D^2 = D^4$,其结果如图5.2(e)所示。

⑥算法第(2)步最后一次迭代计算 $D^4 \times D^4 = D^8$,其结果如图5.2(f)所示。□

5.4 小结和导读

小结 设计并行算法是一件复杂的事,而并行算法的设计这门学科还属于发展中的一门学科,所以目前尚无一套普遍适用的、系统的设计方法学。本章只是给出一个非常一般的并行算法的设计策略,它不可能也不应该视为设计并行算法的具体方法。重要的是,通过所介绍的设计策略的学习,希望读者能从中得到更多

的启迪,补充更多的算例,丰富、完善乃至开拓出更新更好的设计策略。

导读 有关串行快排序算法,[95]和[154]是两篇很好参考文献;PRAM_CRCW上快排序算法,读者可参考[49];超立方上的快排序算法,读者可参考[178];而网孔上的快排序算法,读者可参考[161]。有关并行串匹配算法,读者可参考[191]和[104];其中KMP算法来源于[110]和BM算法来源于[38];[72]和[177]是本章重点介绍的并行串匹配算法。有关图论问题的并行算法,[201]是本很全面的著作;本章所介绍的利用矩阵乘法求所有点对间的最短路径算法来源于[58]。

习 题

5.1 试分析:

① 串行快排序算法最坏情况下的时间复杂度。

② 串行快排序算法平均情况下的时间复杂度。

5.2 给定序列(3,2,1,5,8,4,3,1),试用算法5.1逐步进行排序。

5.3 给定序列(33,21,13,54,82,33,40,72)和8个处理器,试按照算法5.2构造一棵为在PRAM_CRCW模型上执行快排序所用的二叉树。

5.4 令 n 是待排序的元素数,$p=2^d$ 是 d 维超立方中处理器的数目。假定开始随机选定主元 x,并将其播送给所有其它处理器,每个处理器按所接收到的 ,对其 n/p 个元素按照 $\leq x$ 和 $> x$ 进行划分,然后按维进行交换。这样在超立方上实现的快排序的算法如下:

算法5.6 超立方上快排序算法
输入: n 个元素,$B=n/p$,$d=\log p$
输出: 按超立方编号进行全局排序
Begin
(1) id=processors label
(2) for $i=1$ to d do
(2.1) x=pivot /*选主元*/
(2.2) 划分 B 为 B_1 和 B_2 满足 $B_1 \leq x < B_2$
(2.3) if 第 i 位是零 then
(i) 沿第 i 维发送 B_2 给其邻者
(ii) C=沿第 i 维接收的子序列
(iii) $B=B_1 \cup C$
else
(i) 沿第 i 维发送 B_1 给其邻者

(ii) C=沿第 i 维接收的子序列

(ii) $B=B_2 \cup C$

endif

endfor

(3) 使用串行快排序算法局部排序 $B=n/p$ 个数

End

① 试解释上述算法的原理。

② 试举一例说明上述算法的逐步执行过程。

5.5　令模式串 P=abcabcabcabcabcc。开始时，j=1 和 k=2，试按照算法 5.3 计算该模式 P 的失效函数 $F(1) \sim F(16)$。

5.6　① 令 T=babaababaa，P=abab，试用算法 5.4 计算两者的匹配情况。

② 试分析 KMP 算法为何不能简单并行化。

5.7　假定 P_1=abaababa，P_2=abaabaaab，P_3=abaabaaabaabab，试计算各自的 WIT(j) 函数，并判断它们的周期性。

5.8　计算 duel(p,q) 函数的算法如下：

算法 5.7　计算串匹配的 duel(p,q) 的算法

输入：WIT(1:$n-m+1$)，$1 \le p < q \le n-m+1$，$(q-p) < m/2$

输出：返回竞争幸存者的位置或 null(表示 p 和 q 之一不存在)

Procedure DUEL (p,q)

Begin

if p=null then duel=q else

if q=null then duel=p else

(1) $j=q-p+1$

(2) w=WIT(j)

(3) if $T(q+w-1) \neq P(w)$ then

(i) WIT(q)=w

(ii) duel=p

else

(i) WIT(p)=$q-p+1$

(ii) duel=q

endif

endif

endif

End

① 令 T=abaababaabababaababababa，P=abaababa，试计算 WIT(j)；

② 试考虑 $P=6, q=9$ 的竞争情况。

5.9　对于图 5.2(a) 的加权有向图，试用算法 5.5，逐步求出 D^2，D^4 和 D^8 中各元素 d_{ij}^k：

$$\begin{matrix} d_{00}^2 & d_{01}^2 & d_{02}^2 & d_{03}^2 & d_{04}^2 & d_{05}^2 & d_{06}^2 \\ d_{10}^2 & d_{11}^2 & d_{12}^2 & d_{13}^2 & d_{14}^2 & d_{15}^2 & d_{16}^2 \\ & & & \cdots & & & \\ d_{60}^2 & d_{61}^2 & d_{62}^2 & d_{63}^2 & d_{64}^2 & d_{65}^2 & d_{66}^2 \end{matrix}$$

(a)　D^2

$$\begin{matrix} d_{00}^4 & d_{01}^4 & d_{02}^4 & d_{03}^4 & d_{04}^4 & d_{05}^4 & d_{06}^4 \\ d_{10}^4 & d_{11}^4 & d_{12}^4 & d_{13}^4 & d_{14}^4 & d_{15}^4 & d_{16}^4 \\ & & & \cdots & & & \\ d_{60}^4 & d_{61}^4 & d_{62}^4 & d_{63}^4 & d_{64}^4 & d_{65}^4 & d_{66}^4 \end{matrix}$$

(b)　D^4

$$\begin{matrix} d_{00}^8 & d_{01}^8 & d_{02}^8 & d_{03}^8 & d_{04}^8 & d_{05}^8 & d_{06}^8 \\ d_{10}^8 & d_{11}^8 & d_{12}^8 & d_{13}^8 & d_{14}^8 & d_{15}^8 & d_{16}^8 \\ & & & \cdots & & & \\ d_{60}^8 & d_{61}^8 & d_{62}^8 & d_{63}^8 & d_{64}^8 & d_{65}^8 & d_{66}^8 \end{matrix}$$

(c)　D^8

5.10　对于图 5.3 所示的单位权有向图，试用布尔邻接矩阵乘法求出其传递闭包。

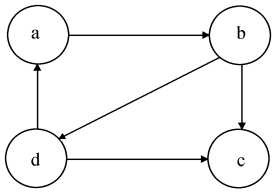


图 5.3　单位权有向图

第六章　并行算法的基本设计技术

并行算法的设计虽然是复杂的，且目前技术上尚不甚成熟，似乎又依赖于某些技巧，但也不是无章可循。随着并行算法设计学科的不断发展，人们逐渐地总结出了一些最基本的设计技术可供参考使用。从使用并行处理操作最朴素的思想出发，就可导出所谓划分设计技术，它是将一原始问题分成若干个部分，然后各部分由相应的处理器同时执行。这是最基本的设计技术，包括**均匀划分技术**、**方根划分**技术、**对数划分**技术和**功能划分**技术。从求解问题的方法学和求解策略出发，则可导出所谓分治设计技术，它是将一个大而复杂的问题分解成若干个特性相同的子问题，然后使用“各个击破”的办法逐步求解之。这是算法设计的普遍技术，既适合于串行算法的设计，又适合于并行算法的设计。从针对求解问题的特性出发，也可导出一些有效的并行算法设计技术，包括平衡树技术和倍增技术等。众所周知，流水线技术是并行处理中最基本的技术之一，它也可以被使用在并行算法的设计中，形成了流水线设计技术。本章所介绍的设计技术并非全面，同时只能作为设计并行算法的一般性指南，而不能作为可直接引用的手册。

6.1 划分设计技术

用划分(Partitioning)法求解问题可分为两步:①将给定的问题劈成 p 个独立的几乎等尺寸的子问题;②用 p 台处理器并行求解诸子问题。划分时关键在于如何将问题进行分组,使得子问题较容易并行求解,或者子问题的解较容易被组合成原问题的解。

6.1.1 均匀划分技术

假定待处理的序列长为 n,现有 p 台处理器,一种最基本的划分方法就是均分划分(Uniform Partition),系将 n 个元素分割成 p 段,每段含有 n/p 个元素且分配给一台处理器。下面以 PSRS(Parallel Sorting by Regular Sampling)算法为例,来阐述均匀划分法的使用。

算法 6.1 MIMD 机器上 PSRS 排序算法

输入: 长度为 n 的无序序列,p 台处理器,每台处理器有 n/p 个元素

输出: 长度为 n 的有序序列

Begin

(1) 均匀划分:n 个元素均匀地划分成 p 段,每台处理器有 n/p 个元素。

(2) 局部排序:各处理器利用串行排序算法,排序 n/p 个数。

(3) 正则采样:每台处理器各从自己的有序段中选取 p 个样本元素。

(4) 样本排序:用一台处理器将所有 p^2 个样本元素用串行排序算法排序之。

(5) 选择主元:用一台处理器选取 $p-1$ 个主元,并将其播送给其余处理器。

(6) 主元划分:各处理器按主元将各自的有序段划分成 p 段。

(7) 全局交换:各处理器将其辖段按段号交换到相应的处理器。

(8) 归并排序:处理器使用归并排序将所接收的诸段施行排序。

End

算法 6.1 中的第(1)步是使用了均匀划分技术,从而使各段(除零头外)的规模(大小)均匀相等。算法的第(6)步也使用了划分技术,但它是非均匀的,各段的大小可能不等。

由于我们只是着重划分技术的描述,所以有关算法 6.1 的复杂度就不再讨论了。

例 6.1 示例说明 PSRS 排序过程。假定序列长度 $n=27$,$p=3$,则 PSRS 排

序过程如图6.1所示。□

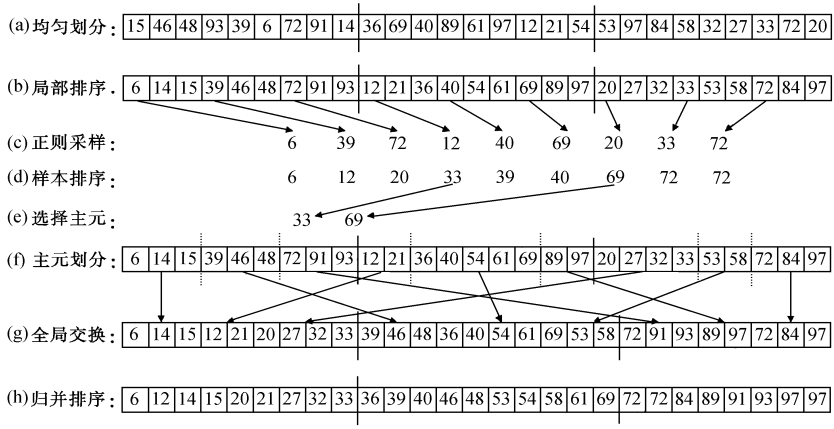


图6.1 例示PSRS排序过程

6.1.2 方根划分技术

假定待处理的序列长为 n,现欲将其分段处理。一种称之为平方根划分(Square Root Partition)的方法,就是取每第 $i\sqrt{n}$($i=1,2,\cdots$)个元素作为划分元素而将序列划分成若干段,然后分段处理之。下面以Valiant归并算法(Valiant's Merging Algorithm)为例,来阐述方根划分法的使用。

算法6.2 SIMD_CREW上Valiant归并算法

输入: 两有序段 A和B段的长度各为 n

输出: 长度为$2n$的有序序列

Begin

(1) 方根划分:将 A和B中的第 $i\sqrt{n}$($i=1,2,\cdots$)个元素作为划分元素,而将 A和B分成了若干段。

(2) 段间比较:A中所有的划分元素与B中所有划分元素进行比较,以确定A中划分元素应插入B序列的哪一段。

(3) 段内比较:A中的划分元素与其所插入的B相应段中的元素进行比较,以确定该划分元素应插入B相应段的哪一位置,而这些插入位置又将 B重新划分成了若干段。

(4) 段组归并：B 被重新划分的各段与原 A 中相应各段构成了成对的一些段组，在各段组内，递归地执行 (1) 到 (3) 步，直到段组中某一序列长度为零为止。

End

算法 6.2 的第 (1) 步是使用了平方根划分技术，而在第 (4) 步的各段组内的递归调用时，各段组内的序列也使用了平方根划分技术。所感兴趣的是，这种划分技术使得算法达到了非常好的时间界。可以证明，算法 6.2 使用 n 个处理器可在 $O(\log\log n)$ 时间内完成两长度各为 n 的有序序列的归并 (见习题 6.2)。

6.1.3　对数划分技术

假定待处理的序列长为 n，现欲将其分段处理。一种和上节所述的方根划分法不同的划分是对数划分 (Logarithmic Partition) 法，即取每第 $i\log n$ ($i=1,2,\cdots$) 个元素作为划分元素，而将序列划分成若干段，然后分段处理之。下面仍以两有序序列的归并为例，来阐述对数划分法的使用。

如算法 6.2 所示，为了确定 A 序列中的划分元素在 B 序列中的全局位序，所以划分元素必须在各段之间施行全局比较 (算法 6.2 的第 (2) 步)。如果在选取 A 序列中的划分元素时，就考虑到了它在 B 序列中的全局位序，那么就不必对划分元素施行段间的全局比较，而可直接对按划分元素所断开的各段组两两进行归并，便可完成两个原序列的归并。下面所要讨论的归并算法就是基于上述思想。

定义 6.1　令 $X=(x_1, x_2,\cdots, x_n)$，$x_i\in S$，S 为具有偏序关系的集合。所谓在 X 中的位序 (Rank)，记之为 rank($\cdot:X$)，就是 X 中小于等于 $\cdot$ 的元素的数目。令 $Y=(y_1, y_2\cdots, y_n)$，$y_i\in S$。所谓 rank($Y:X$) $=(r_1, r_2,\cdots, r_n)$ 就是 Y 在 X 中的位序，其中 $r_i=$rank($y_i:X$)。

例 6.2　令 $X=(25,-13,26,31,54,7)$，$Y=(13,27,-27)$，则 rank($Y:X$) $=(2,4,0)$。□

这样一来，归并问题就能视为确定每个来自序列 A 或 B 中的元素 $\cdot$ 在 $A\cup B$ 中的位序。而求找一个元素在另一个序列中的位置可以使用对半搜索的方法，其时间界为 $O(\log n)$。

下面给出 PRAM 模型上对数划分算法。

算法 6.3　PRAM 上对数划分算法

输入：两非降有序序列 $A=(a_1,\cdots, a_n)$，$B=(b_1,\cdots, b_m)$，假定 $\log m$ 和

$k(m)=m/\log m$ 均为整数

输出：将 A 和 B 划分成 $k(m)$ 对段组 (A_i, B_i)，使得 $|B_i|=\log m$，$\Sigma|A_i|=n$，且对于所有 $1\leq i\leq k(m)-1$，A_i 和 B_i 中的每一个 i 元素均大于 A_{i-1} 和 B_{i-1} 中的每一个元素

Begin

(1) $j(0)=0$；$j(k(m))=n$

(2) for $i=1$ to $k(m)-1$ par-do

(2.1) 求 rank $(b_{\log m}: A)$

(2.2) $j(i)=$rank $(b_{\log m}: A)$

end for

(3) for $i=0$ to $k(m)-1$ par-do

(3.1) $B_i=(b_{\log m+1}, \cdots, b_{(i+1)\log m})$

(3.2) $A_i=(a_{j(i)+1}, \cdots, a_{j(i+1)})$

endfor

End

上述算法所产生的划分如图 6.2 所示。

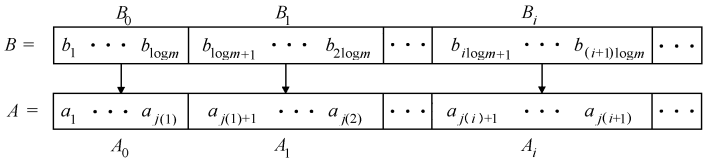


图 6.2 算法 6.3 所产生的划分

一旦使用上述算法完成了两个序列的独立划分后，归并 A 和 B 的问题就变成了并行地成对归并段组 (A_i, B_i) 的问题了。

例 6.3 令 $A=(4,6,7,10,12,15,18,20)$，$B=(3,9,16,21)$。本例中 $m=4$，$k(4)=2$。因为 rank $(9: A)=3$，所以可以得两对划分：$A_0=(4,6,7)$，$B_0=(3,9)$ 和 $A_1=(10,12,15,18,20)$，$B_1=(16,21)$。显然，A_1 和 B_1 中的任一元素都大于 A_0 和 B_0 中的每一元素。因此可以成对归并 (A_0, B_0) 和 (A_1, B_1)，而最终将 A 和 B 归并之。□

6.1.4 功能划分技术

假定欲从长为 n 的序列中选取前 m 个最小者（此即所谓的 (m,n)-选择 $((m,n)$

－Selection)）问题，那么应如何对原序列施行划分以便并行处理呢？此时可以使用所谓功能划分 (Functional Partition) 法，即将长为 n 的序列划分成等长的一些组，每组中的元素数应大于或等于 m (最后一组除外)。然后各组可并行处理，其算法如下：

算法 6.4　(m, n)- 选择网络算法
输入：长为 n 的无序序列 $A=(a_1, \cdots, a_n)$
输出：序列中前 m 个最小者
Begin
(1) 功能划分：将 A 划分成 $g=n/m$ 组，每组含 m 个元素。
(2) 局部排序：使用 Batcher 排序网络将各组并行进行排序。
(3) 两两比较：将所排序的各组两两进行比较，从而形成 MIN 序列。
(4) 排序－比较：对各个 MIN 序列，重复执行第 (2) 和第 (3) 步，直至最终选出 m 个最小者为止。
End

算法 6.4 的执行过程示于图 6.3。

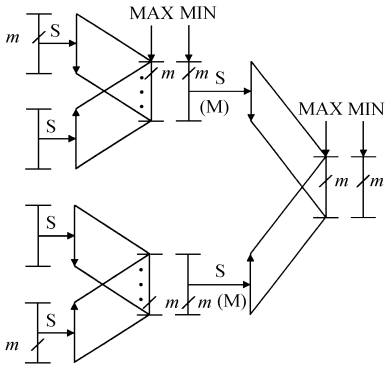


图 6.3　(m, n)- 选择过程 (S 表示排序，M 表示归并)

6.2　分治设计技术

分治 (Divide and Conquer) 策略是一种问题的求解技术，其思想是将一个大而

复杂的问题分解成若干特性相同的子问题分而治之。若所得的子问题规模仍嫌过大,可反复使用分治策略,直至很容易求解诸子问题为止。使用分治法时,子问题的类型通常和原问题的类型相同,因此分治法很自然地导致递归(Recursion)过程。

一般而言,并行分治法分为三步:①将输入划分成若干个规模近于相等的子问题;②同时递归地求解各个子问题;③归并各子问题的解成为原问题的解。用分治法和上节所介绍的划分法求解问题的共同点在于两者均试图将原问题分解成可并行求解的子问题,但分治法的侧重点在于子问题的归并上;而划分法的注意力则集中在原问题的划分上。

下面将通过两个例子来说明分治技术的使用。

6.2.1 双调归并网络

定义6.2 一个序列 $x_0, x_1, \cdots, x_{n-1}$ 是双调序列(Bitonic Sequence),如果:①存在着一个 x_k $(0 \le k \le n-1)$ 使得 $x_0 \ge \cdots \ge x_k \le x_{k+1} \le \cdots x_{n-1}$ 成立;或者②此序列能够循环旋转使得条件①满足。

双调归并是基于下述的Batcher定理[26]:

Batcher定理(Batcher's Theorem):给定一个双调序列 $x_0, x_1, \cdots, x_{n-1}$,对于所有的 $0 \le i \le \frac{n}{2}-1$,执行 x_i 和 $x_{i+\frac{n}{2}}$ 的比较交换得到 $s = \min\{x_i : x_{i+\frac{n}{2}}\}$ 和 $l = \max\{x_i : x_{i+\frac{n}{2}}\}$。则:①所形成的小序列 MIN $= (s_0, s_1, \cdots, s_{2^{-1}})$ 和大序列 MAX $= (l_0, l_1 \cdots, l_{n2^{-1}})$ 仍是双调序列;②且对于所有 $0 \le i, j \le \frac{n}{2}-1$,满足 $s_i \le l_j$。

双调归并可以用比较器网络(Comparator Network)来实现,它是由Batcher比较器(Comparator)构成的。所谓比较器就是一个双输入和双输出的比较交换单元,它可将两输入中的小者置于上输出端,而大者置于下输出端。

下面是Batcher双调归并算法:

算法6.5 Batcher双调归并算法

输入: 双调序列 $X = (x_0, x_1, \cdots, x_{n-1})$

输出: 非降有序序列 $Y = (y_0, y_1, \cdots, y_{n-1})$

Procedure BITONIC MERG (X)

Begin

(1) for $i = 0$ to $\frac{n}{2}-1$ par-do

(1.1)　$s_i = \min\{x_i : x_{i+\frac{n}{2}}\}$　　/*":"表示比较*/

(1.2)　$l_i = \max\{x_i : x_{i+\frac{n}{2}}\}$

end for

(2) Recursive Call:

(2.1) BITONIC MERG (MIN = $(s_0, \cdots, s_{\frac{n}{2}-1})$)

(2.2) BITONIC MERG (MAX = $(l_0, \cdots, l_{\frac{n}{2}-1})$)

(3) output sequence MIN followed by sequence MAX

End

双调归并网络的递归构造如图 6.4 所示。

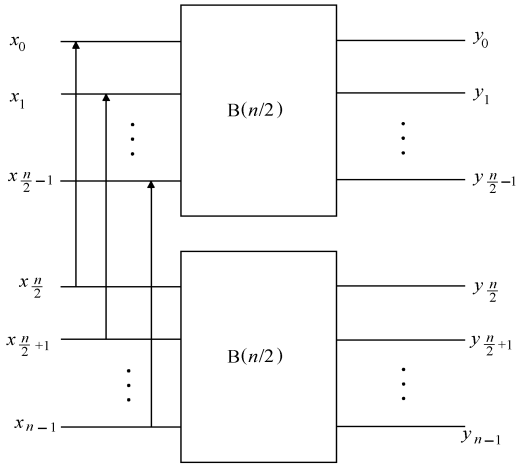


图 6.4　双调归并网络的递归构造

6.2.2　凸壳问题

定义 6.3　如果多边形 Q 上任意两点的连线均处于 Q 之内，则此多边形 Q 称之为凸多边形 (Convex Polygon)。

定义 6.4　给定平面中 n 点集合 $S = (p_1, \cdots, p_n)$，所谓 S 的凸壳 (Convex Hull) 就是包含 S 中所有 n 点的最小凸边形。

求平面凸壳的问题，就是要确定凸壳边界上的有序顶点表列 $CH(S)$。

令 p 和 q 是 S 中具有最小和最大 x 坐标的两个点。很清楚 p 和 q 可将全凸

壳 $CH(S)$ 划分成上凸壳 $UH(S)$ 和下凸壳 $LH(S)$，其中 $UH(S)$ 系由 p 到 q 的所有点组成，而 $LH(S)$ 系由 q 到 p 的所有点组成。

例如图 6.5 中，$UH(S)=(v_1, v_2, v_3, v_4, v_5)$，而 $LH(S)=(v_5, v_6, v_7, v_8, v_1)$。这样一来，一个完整的凸壳 $CH(S)$ 就是由上凸壳 (Upper Convex Hull) $UH(S)$ 和下凸壳 (Lower Convex Hull) $LH(S)$ 组成。而求 $LH(S)$ 和 $UH(S)$ 是相似的，所以下面只讨论上凸壳 $UH(S)$ 的求法。

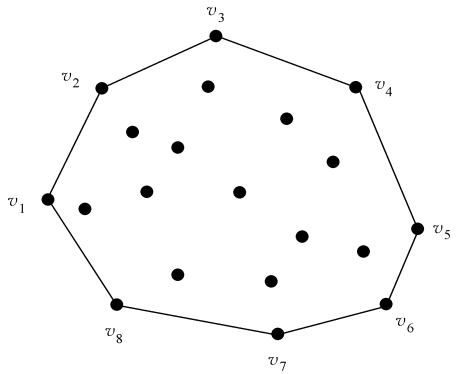


图 6.5 集合 S 的凸壳 $CH(S)=(v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$

令 $x(p)$ 是点 p 的 x 坐标，假定集合 S 中的点已按 x 坐标从小到大排序好，即 $x(p_1)<x(p_2)<\cdots<x(p_n)$。那么求上凸壳 $UH(S)$ 时可以使用分治技术。令 $S_1=(p_1, p_2, \cdots, p_{\frac{n}{2}})$ 和 $S_2=(p_{\frac{n}{2}+1}, p_{\frac{n}{2}+2}, \cdots, p_n)$，那么一旦 $UH(S_1)$ 和 $UH(S_2)$ 递归地求出后，只要再求出 $UH(S_1)$ 和 $UH(S_2)$ 的上公切线 (Upper Common Tangent)，那么求 $UH(S)$ 的主要问题就解决了 (参照图 6.6)。下面给出求上凸壳算法：

算法 6.6 PRAM 上求上凸壳算法

输入： $S=(p_1, \cdots, p_n)$，且 $x(p_1)<x(p_2)<\cdots<x(p_n)$

输出： 返回上凸壳顶点表列 $UH(S)$

Procedure UPPER HULL (S, $UH(S)$)

Begin

(1) if $n\leq 4$ then use a brute-force to determine $UH(S)$ endif

(2) $S_1=(p_1, p_2, \cdots, p_{\frac{n}{2}})$ and $S_2=(p_{\frac{n}{2}+1}, p_{\frac{n}{2}+2}, \cdots, p_n)$

(3) 递归调用并行计算 $UH(S_1)$，$UH(S_2)$：

(3.1) UPPER HULL (S_1, $UH(S_1)$)

(3.2)　UPPER HULL（S_2, $UH(S_2)$）

(4)　求 $UH(S_1)$ 和 $UH(S_2)$ 间上公切线，构造 $UH(S)$

End

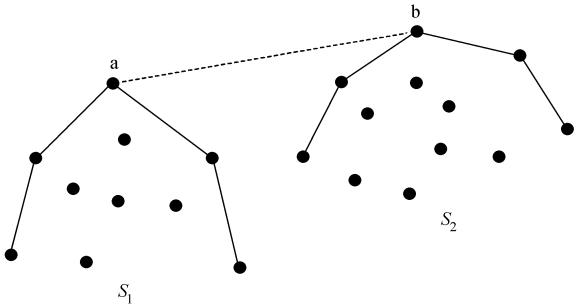


图 6.6　示例求两上凸壳的上公切线

在 PRAM 上求下凸壳的算法和算法 6.6 雷同。组合上凸壳与下凸壳算法就可导出如下 PRAM 上求凸壳算法。

算法 6.7　PRAM 上求凸壳算法

输入：　$S=(p_1,\cdots,p_n)$，$p_i=(x_i,y_i)$

输出：返回凸壳顶点表列 $CH(S)$

Procedure CONVEX HULL（S, $CH(S)$）

Begin

(1)　/*识别极点*/

(1.1)　XMIN=min{x_i 坐标}

(1.2)　XMAX=max{x_i 坐标}

(2)　/*按 x 坐标进行划分*/

(2.1)　S_1 =vertices from p_{XMIN} to p_{XMAX}

(2.2)　S_2 =vertices from p_{XMAX} to p_{XMIN}

(3)　/*递归调用上凸壳和下凸壳算法*/

(3.1)　UPPER HULL（S_1, $UH(S_1)$）

(3.2)　LOWER HULL（S_2, $UH(S_2)$）

(4)　/*并接 $UH(S_1)$ 和 $LH(S_2)$ */

$CH(S)=UH(S_1)+LH(S_2)$

End

上述算法中,求上(下)公切线是很关键的。由于主要目的是展示分治技术在算法设计中的应用,所以公切线的求法就不再讨论,感兴趣的读者可参阅[104]中的第六章。

6.3 平衡树设计技术

平衡树(BaLanced Tree)方法是将输入元素作为叶节点构筑一棵平衡二叉树,然后自叶向根往返遍历。此法成功的部分原因是在树中能快速地存取所需要的信息。平衡二叉树的方法可推广到内节点的子节点的数目不只两个的任意平衡树。这种方法对数据的播送、压缩、抽取和前缀计算等甚为有效。

6.3.1 求取最大值

使用平衡二叉树求取数的最大值时,根节点给出问题的解;叶节点存放待处理的数据;内节点执行相应子问题的计算。在树的同一深度上各内节点并行计算。

令 $n=2^m$,A 是一个 $2n$ 维的数组;待求最大值的 n 个数开始存放在 $A(n)$,$A(n+1)$,…,$A(2n-1)$;所求得的最大值置于 $A(1)$,于是算法描述如下:

算法 6.8 SIMD_TC 上求最大值算法
输入: $n=2^m$ 个数存在数组 $A[n:2n-1]$ 中
输出: 最大数置于 $A(1)$ 中
Begin
  for $k=m-1$ to 0 do
    for $j=2^k$ to $2^{k+1}-1$ par-do
      $A(j)=\max\{A(2j),A(2j+1)\}$
    end for
  end for
End

显然算法的时间 $t(n)=O(\log n)$,总比较次数为 $O(n)$,而最大的处理器数 $p(n)=n/2$。

6.3.2 计算前缀和

对于取值于集合 S 上的满足二元结合运算 $*$ 的 n 个元素 $(x_1, x_2, \cdots, x_n)$ 的序列，所谓 n 个元素的前缀和 (Prefix_Sums) 是指如下定义的 n 个部分和 (或积)：

$$s_i = x_1 * x_2 * \cdots * x_i, \quad 1 \le i \le n$$

显然，使用等式 $s_i = s_{i-1} * x_i$ $(2 \le i \le n)$ 计算前缀和的平易的串行算法具有固有的顺序性，且时间为 $O(n)$。

使用平衡二叉树计算前缀和时，在自叶向根正向遍历过程中，各内节点对其相应的子节点应用一次 $*$ 运算，因此每个节点 v 保存了根在 v 的子树的叶中所存储元素的和；在自根向叶反向遍历过程中，将计算出给定高度上各节点中所存储的元素的前缀和。

下面给出一个非递归求前缀和算法。令 $A(j) = x_j$ $(1 \le j \le n)$；令 $B(h,j)$ 和 $C(h,j)$ 是辅助变量集 $(0 \le h \le \log n, 1 \le j \le n/2^h)$，其中数组 B 用于记录正向遍历时树中各节点的信息，而数组 C 用于记录反向遍历时树中各节点的信息。

算法 6.9 SIMD_TC 上非递归求前缀和算法
输入：$n=2^k$ 的数组 A，k 为非负整数
输出：数组 C，其中 $C(0,j)$ 是第 j 个前缀和 $(1 \le j \le n)$
Begin
(1) for $j=1$ to n par-do /*初始化*/
$B(0,j) = A(j)$
end for
(2) for $h=1$ to $\log n$ do /*正向遍历*/
for $j=1$ to $n/2^h$ par-do
$B(h,j) = B(h-1,2j-1) * B(h-1,2j)$
end for
end for
(3) for $h=\log n$ to 0 do /*反向遍历*/
for $j=1$ to $n/2^h$ par-do
(i) if j=even then $C(h,j) = C(h+1,j/2)$ end if
(ii) if $j=1$ then $C(h,1) = B(h,1)$ end if
(ii) if j=odd>1 then $C(h,j) = C(h+1,(j-1)/2) *$
$B(h,j)$ endif

```
        end for
    end for
End
```

6.4 倍增设计技术

倍增技术(Doubling Technique)又叫指针跳越(Pointer Jumping)技术,特别适合处理以链表或有向有根树之类表示的数据结构,在图论和链表算法中有着广泛的应用。每当递归调用时,所要处理的数据之间的距离将逐步加倍,经过 k 步后就可完成距离为 2^k 的所有数据的计算。下面以表列中元素的位序(Rank)[简称表序问题(List Ranking Problem)]和求找森林的根为例,来说明此项技术的具体使用。

6.4.1 表序问题的计算

令 L 是 n 个元素的表列,且每个元素分配一个处理器。所谓表序问题就是给表中每个元素 k 指定一个它在表列中的次第号($rank(k)$),$rank(k)$ 可视为元素 k 至表尾的距离。为此每个元素 k 都有一个指向下一个元素的指针 $next(k)$。如果 k 是表中最后一个元素,则 $next(k)=k$。具体算法形式描述如下:

```
算法 6.10  SIMD_EREW 上求元素表序算法
输入:  n 个元素的表列 L
输出:  rank(k), k∈L
Begin
  (1) for all k∈L par-do
        (1.1) P(k)=next(k)
        (1.2) if P(k)≠k then distance(k)=1 else distance(k)=0 end if
      end for
  (2) repeat ⌈log n⌉ times
        (2.1) for all k∈L par-do
                if P(k)≠P(P(k)) then
                  (i) distance(k)=distance(k)+distance(P(k))
                  (ii) P(k)=P(P(k))
                end if
```

```
            end for
        (2.2)  for all k∈ L par-do
                  rank(k) = distance(k)
               end for
      end repeat
End
```

显然,算法 6.10 的 $t(n) = O(\log n)$,$p(n) = n$。

例 6.4　令 n=7。图 6.7 示例出算法 6.10 的执行过程。其中带箭头的弧为指针 $P(k)$ 弧上的数字为 $distance(k)$ 的值。开始时,在算法 6.10 第 (1.1) 步 $P(k)$ 进行初始化;在第 (1.2) 步计算出 $distance(k)$ 的值。算法总共迭代三次,各元素的指针 $P(k)$ 和 $distance(k)$ 依次标注在图 6.7 中。□

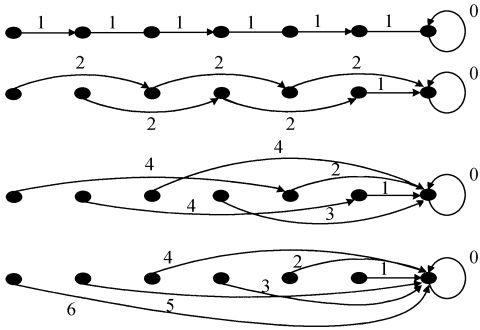


图 6.7　n=7 算法 6.10 的执行过程

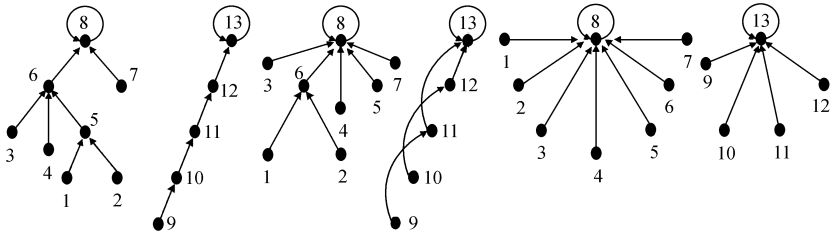
6.4.2　求森林的根

令森林 F 是一组有根有向树,其中 F 由长为 n 的数组 P 指定,使得如果 (i,j) 是 F 中的一条弧,则 $P(i) = j$,即 j 是 i 的父亲;如果 i 是一个根,则 $P(i) = i$。所谓找森林的根 (Finding Roots of a Forest) 的问题,就是对于每个节点 $j(1 \le j \le n)$,确定包含该节点的树的根 $S(j)$。开始时,每个节点 i 的后继 $S(i)$ 定义为 $P(i)$。指针跳越技术就是用后继的后继去修改每个节点的后继。重复使用此法,一个节点的后继就变成了越来越靠近树根的祖先。所以节点及其后继之间的距离将逐次加倍,经过 k 次迭代后,i 和 $S(i)$ 之间的距离则为 2^k。算法的描述如下:

算法 6.11 SIMD_CREW 上求森林根的算法
输入： 森林 F,弧由 $(i, P(i))$ 指定,$1 \le i \le n$
输出： 对每个节点 i,输出包含 i 的树的根 $S(i)$
Begin
  for $1 \le i \le n$ par-do
    (1) $S(i) = P(i)$
    (2) while $S(i) \neq S(S(i))$ do
          $S(i) = S(S(i))$
        end while
  end for
End

令 h 是森林中树的最大高度,不难看出算法将迭代 $O(\log h)$ 次,每次迭代作了 $O(n)$ 次运算而花费了 $O(1)$ 时间。所以算法的 $t(n) = O(\log n)$, $W(n) = O(n \log n)$。

例 6.5 图 6.8 中示例了使用指针跳越技术求森林的根的过程。本例中,如图 6.8(a) 所示,森林由两棵根在 8 和 13 的树组成。图中的弧相应于 $(i, P(i))$ $(1 \le i \le 13)$。算法执行 while 语句的第一次循环后的结果如图 6.8(b) 所示,致使节点 1,2,3,4,5,9,10 和 11 改变它们的后继;第二次循环后如图 6.8(c) 所示,使得两棵树的高度都变为 1。现在对于所有 i,有 $S(i) = S(S(i))$,所以算法结束。□



(a) 起始森林　　(b) 第一次迭代后的森林　　(c) 第二次迭代后的森林

图 6.8 求森林的根

6.5 流水线设计技术

并行处理中,流水线(Pipelining)技术是一项重要的并行技术。它在 VLSI 并行计算中表现得尤为突出。其基本思想是将一个计算任务 t 分成一系列子任务 $t_1, t_2, \cdots, t_m$,使得一旦 t_i 完成,后继的子任务就可立即开始,并以同样的速率进行计算。下面以一维心动阵列(Systolic Array)上离散傅氏变换(DFT)和卷积计算为例,说明流水线技术的使用。

6.5.1 一维心动阵列上的 DFT 计算

一个 n 点的离散傅里叶变换(Discrete Fourier Transform),简记之为 DFT,可定义为:给定序列 $(a_0, a_1, \cdots, a_{n-1})$,按如下规则变换成序列 $(b_0, b_1, \cdots, b_{n-1})$:

$$b_j = \sum_{k=0}^{n-1} a_k \omega^{kj}, 0 \le j \le n-1 \tag{6.1}$$

其中,ω 是单位 n 次元根,即 $\omega = e^{2\pi i/n}$,$i = \sqrt{-1}$。

事实上(6.1)式的计算,可以等效于多项式求值:设想有一多项式 $y(x) = \sum_{k=0}^{n-1} a_k x^k$,欲求其在 $x = \omega^j$ $(0 \le j \le n-1)$ 处的 $y(x)$ 之值,即 $y(\omega^j) = \sum_{k=0}^{n-1} a_k \omega^{jk}$,则显然此式和(6.1)式是完全一样的。

例如,欲计算一个 5 点的 DFT,可以通过计算 x 等于 ω^0, ω^1, ω^2, ω^3 和 ω^4 处的如下多项式的值而求得:

$$y(x = \omega^j) = a_4 (\omega^j)^4 + a_3 (\omega^j)^3 + a_2 (\omega^j)^2 + a_1 (\omega^j)^1 + a_0 \tag{6.2}$$

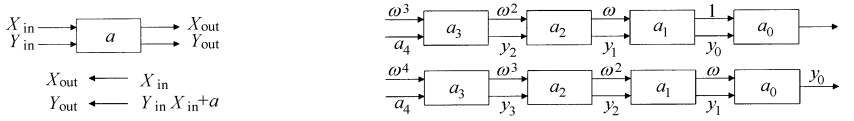
即

$$\begin{aligned}
y(\omega^0) &= a_4 \omega^0 + a_3 \omega^0 + a_2 \omega^0 + a_1 \omega^0 + a_0 \\
y(\omega^1) &= a_4 \omega^4 + a_3 \omega^3 + a_2 \omega^2 + a_1 \omega^1 + a_0 \\
y(\omega^2) &= a_4 \omega^8 + a_3 \omega^6 + a_2 \omega^4 + a_1 \omega^2 + a_0 \\
y(\omega^3) &= a_4 \omega^{12} + a_3 \omega^9 + a_2 \omega^6 + a_1 \omega^3 + a_0 \\
y(\omega^4) &= a_4 \omega^{16} + a_3 \omega^{12} + a_2 \omega^8 + a_1 \omega^4 + a_0
\end{aligned} \tag{6.3}$$

根据 Horner 规则(Horner's Rule),(6.3)式可变换为如下的等效形式:

$$
\begin{aligned}
y(\omega^0) &= y_0 = (((a_4\omega^0 + a_3)\omega^0 + a_2)\omega^0 + a_1)\omega^0 + a_0 \\
y(\omega^1) &= y_1 = (((a_4\omega^1 + a_3)\omega^1 + a_2)\omega^1 + a_1)\omega^1 + a_0 \\
y(\omega^2) &= y_2 = (((a_4\omega^2 + a_3)\omega^2 + a_2)\omega^2 + a_1)\omega^2 + a_0 \\
y(\omega^3) &= y_3 = (((a_4\omega^3 + a_3)\omega^3 + a_2)\omega^3 + a_1)\omega^3 + a_0 \\
y(\omega^4) &= y_4 = (((a_4\omega^4 + a_3)\omega^4 + a_2)\omega^4 + a_1)\omega^4 + a_0
\end{aligned}
\qquad (6.4)
$$

很明显，(6.4) 式中所有的 y_i $(0 \le i \le 4)$ 都可以在图 6.9 所示的一维线性阵列上以流水线方式计算之。可以看出，每个 y_i 初始化为 a_4（一般为 a_{n-1}），然后收集其有关项向右行进，当其积累完了所有项后，从最右单元输出。显然，在 $n-1=O(n)$ 个单元的心动阵列上，计算 n 点 DFT 可在 $2n-1=O(n)$ 时间内完成。



(a) 处理单元功能及其定义

(b) 5点 DFT 在一维阵列上的计算过程

图 6.9 一维线性阵列上计算 DFT

6.5.2 一维心动阵列上的卷积计算

卷积 (Convolution) 在数学上可定义如下：给定权序列 $(w_1, w_2, \cdots, w_n)$ 和 x 序列 $(x_1, x_2, \cdots, x_m)$，$m > n$，试按下式计算 y 序列 $(y_1, y_2, \cdots, y_{m-n+1})$：

$$y_j = \sum_{i=1}^{n} w_i x_{i+j-1},\ 1 \le j \le m-n+1 \qquad (6.5)$$

卷积在数字信号处理、模式识别、多项式求值等方面应用很广泛。例如 (6.5) 式中的乘、加代之以“比较”和“布尔与”操作，则卷积问题就变成了模式匹配问题。因为每个 y_j 可在 $\Theta(n)$ 时间内计算出，所以卷积可在 $\Theta(m)$ 时间内计算完毕。

例如，给定 4 个权的序列 (w_1, w_2, w_3, w_4) 和 6 个数的序列 $(x_1, x_2, x_3, x_4, x_5, x_6)$，则须计算如下的 y 序列 (y_1, y_2, y_3)：

$$
\begin{aligned}
y_1 &= x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4 \\
y_2 &= x_2 w_1 + x_3 w_2 + x_4 w_3 + x_5 w_4 \\
y_3 &= x_3 w_1 + x_4 w_2 + x_5 w_3 + x_6 w_4
\end{aligned}
\tag{6.6}
$$

当在一维心动阵列上计算上述卷积时需要4个处理单元 C_1、C_2、C_3 和 C_4，其中每个 C_i 中存有 w_i，且执行 (6.5) 式计算。计算时，如图 6.10 所示，输入序列 $x_1 \sim x_6$ 由左方进入心动阵列，输出序列 $y_1 \sim y_3$ 由右方进入心动阵列。开始时，x 的元素每隔一拍流入阵列，而 y 的元素须延迟3拍后也每隔一拍流入阵列。在前3拍时不作任何计算，而第4拍时 C_1 执行 $y_1 + x_1 w_1$ 操作 (y_1 初始化为零)，类似地，在第5拍时 C_2 执行 $y_1 + x_2 w_2$，在第6拍时，C_1 和 C_3 分别同时计算 y_2 和 y_1 所需的项。一般而言，当 C_k 接收到 x_i 和 y_j 时，则执行操作 $x_i w_k + y_j$。

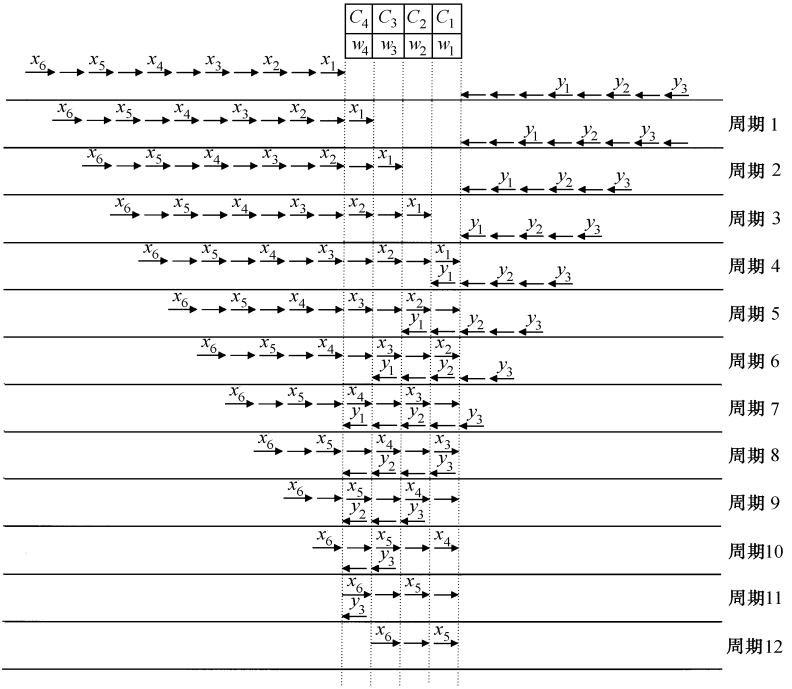


图 6.10　卷积在一维心动阵列上的执行过程

一般情况下，当权数为 n 和输入序列长度为 m 时，须用 n 个心动单元，y 序列需滞后 $n-1$ 个拍。因为 y 需要计算 $(m-n+1)$ 个元素，所以总共约需 $2(m-n)$

个节拍。当 $m \gg m$ 时,此算法的成本为 $\Theta(m)$,显然是成本最优的。

6.6 小结和导读

小结 本章所介绍的都是并行算法设计的最基本的技术,远非所有的设计技术。这些技术虽具有一定的普适性,但也并非对所有问题均行之有效,况且也不全面,同时也只能作为设计并行算法的一般性指南,绝非是可直接引用的手册。其它的设计技术还有破对称(Symmetry Breaking)技术,它是打破某些问题的对称性的一种设计方法,在图论问题和随机算法中经常使用;加速级联(Accelerated Cascading)技术,它试图将一个最优但相对不快的算法与另一个非常快但非最优的算法"串接"起来,形成一个快速最优的算法,此法虽不具有普适性,但对有些问题却效果甚佳;随机法是在算法设计时,引入随机性使得算法在执行时,可以随机地选择下一执行步,从而可望得到平均性能较好的算法(即其复杂度比同一问题的确定性算法的最坏情况的复杂度要好),随机算法设计简单,但分析较复杂,经常要分析算法可在多少时间步内以多大的概率结束,用随机法设计的算法,其运行结果也可能是不正确的,但这种错误的概率应是很小的;贪心法(Greedy Method)是一种最直接的设计技术,其设计特点是,在算法的每一步都力图确保获得局部最优,这种设计方法的副作用是容易陷入局部最优。对于组合问题的算法设计技术,有动态规划法(Dynamic Programming),它是在每一判定步,列出多种可能的解,然后按照某种条件,舍弃那些肯定不能导致最优解的局部解,它和贪心法的区别在于,贪心法仅产生一个判定序列,而动态规划法可能产生很多判定序列,但依据"最优性原理",非局部最优解的序列肯定不是最优的,所以不予考虑;回溯法(Backtracking)是一种穷举法,在问题求解时常使用深度优先搜索(Depth_First Searching)法,在搜索过程中,一旦碰到某个无法搜索下去的分支,就向其父节点回溯,再搜索该父节点的其它分支,如果所有的分支节点均无希望的解,则再向上追溯,如此等等,回溯法虽提供了一种规律性求解的方法,但其时间复杂度较高;分支界限法(Branch and Bound)与回溯法颇类似,但它是基于广度优先搜索(Breadth_First Searching)法,它利用部分解的最优性信息,避免考虑那些不能导致最优的解,以加快问题的求解,其解题过程是:对解集合反复进行分支,即反复构造其子集合,每次分支时都对所得到的子集合计算最优解的界,若它不能优于当前已知的最优解,则对此子集合就不再进行分支,否则继续分支,以探索更好的解,直到所得的子集合仅有一个解为止。

导读 有关 PSRS 排序算法的设计和分析,读者可参阅[157];使用平方根划

分法的归并算法是由 Valiant 提出[175]；使用对数划分法的归并算法，读者可参阅[158]；有关 (m,n)_选择算法，读者可参阅[189]。分治方法的一般介绍，读者可参阅[97]；双调归并网络是由 Batcher 提出[26]；用分治法求凸壳问题，读者可参阅[156]。并行前缀和计算，读者可参阅[116]。表序计算问题，读者可参阅[185]；倍增技术在图论中的应用，Hirschberg 求连通分量是一篇常被人引用的文章[93]。关于心动阵列的并行计算，[115]是一篇很经典的文章。加速级联技术由 Cole 等人提出[51]；破对称设计技术，读者可参阅[77]。在[104]书的第九章专门讨论了随机算法。组合问题的算法设计技术，读者可参阅[179]。

习　题

6.1 ① 试证明：当 $n\geq p^3$ 时，算法 6.1 的时间复杂度为 $O\left(\frac{n}{p}\log n\right)$。

② 令 w_i^j 表示 P_i 中第 j 段中的元素数，试证明算法 6.1 在执行过程中，处理器中所积累的元素数目不会超过 $2n/p$，即 $\sum_{j=1}^{p} w_i^j<\frac{2n}{p}$。

6.2 ① 试举一典型算例，说明 Valiant 归并算法的执行过程。

② 试分析算法 6.2 所需的处理器数 $p(n)=O(n)$。

③ 试证明算法 6.2 的时间复杂度为 $2\log\log n+\text{const}$。

6.3 ① 试分析算法 6.3 的复杂度。

② 令 $A=(0,1,2,7,9,11,16,17,18,19,23,24,25,27,28,30,33,34)$，$B=(3,4,5,6,8,10,12,13,14,15,20,21,22,26,29,31)$。试按算法 6.3，将其进行对数划分，并最终将它们归并之。

6.4 ① 试举一例，说明算法 6.4 的执行过程。

② 试证明算法 6.4 的正确性。

6.5 ① 试证明 Batcher 定理。

② 画出一个 16 个输入的双调归并网络。

6.6 ① 试分析算法 6.9 的总运算量 $W(n)=?$

② 假定序列为 $(1,2,3,4,5,6,7,8)$，试用算法 6.9 求其前缀和。

6.7 定义 $\log^{(i)}x$ 如下：$\log^{(0)}x=\log x,\log^{(2)}x=\log\log x,\log^{(i)}x=\log(\log^{(i-1)}x)$。令函数 $\log^* x=\min\{i\mid \log^{(i)}x\leq 1\}$。

当 $x\leq 2^{65536}$ 时，此函数界限是什么？

6.8 对于 $n=8$，按照 DFT 定义，计算 w 矩阵之各元素。

6.9 试解释在一维心动阵列上计算卷积时，序列 x 和 y 为何要各间隔一拍进入阵列。

6.10 顶点倒塌法是非常有名的求图的连通分量的算法，其基本思想是：连通的相邻的顶点可以合并成一个超顶点，并以它们中最小标号者标记之。此过程可继续在已合并的超顶点之间

进行。在下列的算法中，$C(i)$表示与 i相邻的最小的超顶点号码；$D(i)$表示顶点 i所属连通分量的最小标号的顶点；$C(i)=\min_j\{D(j)\ \ _{A(i,j)=1,D(i)\neq D(j)}\}$ 语句为每个顶点 i找与它不属于相同分量的相邻的最小号码的顶点 j语句 $C(i)=\min_j\{C(j)\ \ _{D(j)=i,C(j)\neq i}\}$ 表示把每个超顶点的根连到最小号码的相邻的超顶点的根上。Hirschberg的求连通分量算法如下：

算法 6.12　PRAM_CREW 上 Hirschberg求连通分量算法
输入：邻接矩阵 $A_{n\times n}$
输出：向量 $D[0:n-1]$，其中 $D(i)$表示向量 D的分量
Begin
(1) for all $i:0\leq i\leq n-1$ par-do　　/*初始化*/
　　$D(i)=i$
　end for
　do step (2) through (6) for $\lceil\log n\rceil$ iterations:
(2) for all $i,j:0\leq i,j\leq n-1$ par-do　　/*找相邻的最小者*/
　(2.1) $C(i)=\min_j\{D(j)\ \ _{A(i,j)=1\text{ and }D(i)\neq D(j)}\}$
　(2.2) if none then $C(i)=D(i)$ endif
　end for
(3) for all $i,j:0\leq i,j\leq n-1$ par-do　　/*找每个超顶点的最小相邻超顶点*/
　(3.1) $C(i)=\min_j\{C(j)\ \ _{D(j)=i\text{ and }C(j)\neq i}\}$
　(3.2) if none then $C(i)=D(i)$ endif
　end for
(4) for all $i:0\leq i\leq n-1$ par-do $D(i)=C(i)$ end for
(5) for $\lceil\log n\rceil$ iterations do　　/*指针跳越，找各顶点新的超顶点*/
　　for all $i:0\leq i\leq n-1$ par-do $C(i)=C(C(i))$ end for
　end for
(6) for all $i:0\leq i\leq n-1$ par-do
　　$D(i)=\min\{C(i),D(C(i))\}$
　end for
End

① 试分析算法6.12的复杂度 $t(n)=?$和 $p(n)=?$
② 给定如图6.11所示无向图，试用算法6.12逐步求出该图的连通分量。

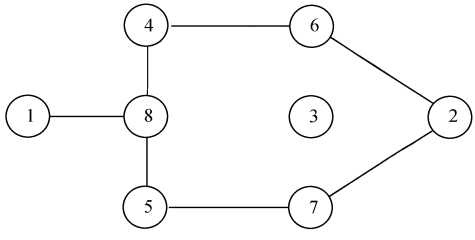


图 6.11　待求连通分量的无向图

第七章　并行算法的一般设计过程

设计一个高效的并行算法并非易事,其过程比较复杂,很难归结为一个单一化的过程而“一步到位”,往往需要几经反复方能达到要求。

本章试图从给定问题的描述出发,通过一系列步骤,最终设计出一个能展示出并发性、可扩放性、局部性和模块性的并行算法。此过程可分为四步,即任务划分(Partitioning)、通信(Communication)分析、任务组合(Agglomeration)和处理器映射(Mapping),简称为 PCAM 设计过程,它是一种设计方法学,是实际设计并行算法的自然过程,其基本要点是:首先尽量开拓算法的并发性和满足算法的可扩放性;然后着重优化算法的通信成本和

全局执行时间，同时通过必要的整个过程的反复回溯，以期最终达到一个满意的设计选择。

上述设计过程的最后阶段是映射，也就是将经过优化的诸进程指派给具体的处理器去运行。这样 PCAM 算法设计过程的最终结果将是一个并行程序。所以就此意义而言，本章所讲的并行算法的设计过程，有时也不加严格区分地应用到并行程序的设计上。

7.1　PCAM 设计方法学

已如上述，PCAM 是 Partitioning、Communication、Agglomeration 和 Mapping 首字母的拼写，它们代表了使用此法设计并行算法的四个阶段。其中在设计的前期主要考虑如并发性等与机器无关的特性，在设计的后期才考虑与机器有关的特性。也就是说，在设计的第一和第二阶段，关注的是并发性和可扩放性，并寻求开发出具有这些特性的并行算法，在设计的第三和第四阶段，才把注意力转移到局部性和别的与性能有关的问题上。并行算法设计的整个过程可概括于图 7.1 中，它从给定设计问题的说明开始，寻找一种计算任务的划分方法，确定诸任务的通信要求，组合可能的计算任务，最后将优化了的诸任务指派给实际的处理器。上述各阶段可简述如下：

① 划分 将整个计算分解成一些小的任务，其目的是尽量开拓并行执行的机会。

② 通信 确定诸任务执行中所需交换的数据和协调诸任务的执行，由此可检测上述划分的合理性。

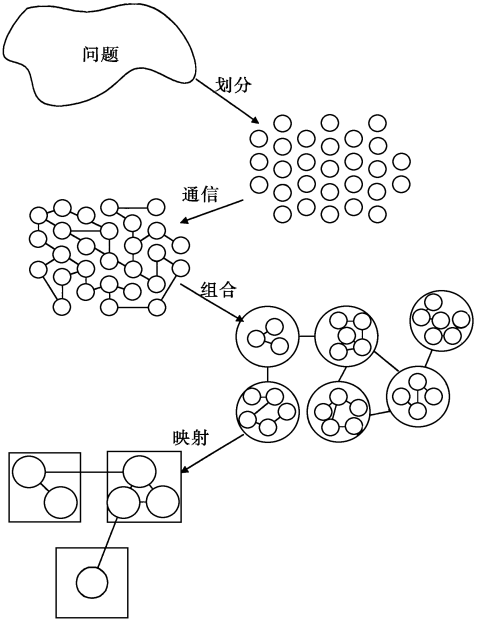


图 7.1　算法的 PCAM 设计过程

③ 组合 按性能要求和实现的代价来考察前两阶段的结果，必要时可将一些小的任务组合成更大的任务以提高性能或减少通信开销。

④ 映射 将每个任务分配到一个处理器上，其目的是最小化全局执行时间和通信成本以及最大化处理器的利用率。

虽然上述的设计过程是一步一步进行的，但实际上它们可以同时一并考虑；同样，虽然我们希望一个算法能用上述四步一次设计成功，但实际上设计过程的回溯反复总是难免的。

7.2 划 分

所谓划分，就是使用域分解或功能分解的办法将原计算问题分割成一些小的计算任务，以充分揭示并行执行的机会，这犹如细小的砂粒比大块的砖瓦易于灌注一样，划分以充分开拓算法的并发性和可扩放性为目的；其方法是先集中数据的分解（称域分解），然后是计算功能的分解（称功能分解），两者互为补充；划分的要点是力图避免数据和计算的复制，应使数据集和计算集互不相交。

7.2.1 域分解

域分解（Domain Decomposition）也叫数据划分。所要划分的对象是些数据，这些数据可以是算法（或程序）的输入数据，计算的输出数据，或者算法所产生的中间结果。

域分解的步骤是，首先分解与问题相关的数据，如果可能的话，应使这些小的数据片尽可能大致相等；其次再将每个计算关联到它所操作的数据上。由此划分将会产生一系列的任务。每个任务包括一些数据及其上的操作。当一个操作可能需要别的任务中的数据时，就会产生通信要求。

域分解的经验方法是，优先集中在最大数据的划分；或者那些经常被访问的数据结构上。在计算的不同阶段，可能要对不同的数据结构进行操作，或者需要对同一数据结构进行不同的分解，在此情况下，我们要分别对待，然后考虑将各阶段设计的分解和算法装配到一起。

图 7.2 示出了一个三维网格的域分解方法，在各格点上计算都是重复执行的，分解在 X，Y 和/或 Z 维是可能的，开始时，应集中在能提供最大灵活性的三维（即 3_D）分解上，即每一个格点定义一个计算任务，每个任务维护与其格点有关的各种数据，并负责计算以修改状态。

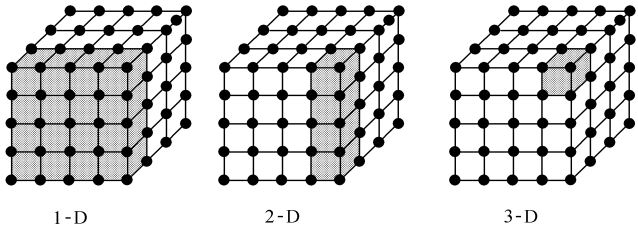


图 7.2 三维网格问题的域分解法 (各图中的阴影部分代表一个任务)

7.2.2 功能分解

功能分解 (Functional Decomposition) 也叫计算划分,其基本出发点不同于域分解,它首先关注于被执行的计算上,而不是计算所需的数据上。然后,如果所作的计算划分是成功的,再继续研究计算所需的数据,这些数据可能是不相交的,这就意味着划分很成功;或者这些数据有相当的重叠,这就产生大量的通信,此时就暗示应考虑数据分解。

尽管域分解是绝大多数并行算法所使用的,但功能分解却有时能揭示问题的内在结构展示出优化的机遇,而对数据单独进行研究却往往难以作到这一点。

搜索树是功能分解的最好例子,此时无任何明显的可分解的数据结构,但却易于进行细粒度的功能分解。开始时根生成一个任务,对其评价后,如果它不是一个解,就生成一棵搜索子树,整个搜索过程如图 7.3 所示,自根以波前 (Wavefront) 形式逐级向树叶推进 (见习题 7.1)。

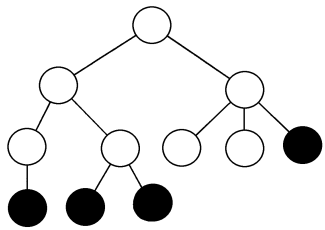


图 7.3 功能分解的搜索树

7.2.3 划分判据

下面的一组问答题,可检验你所作的划分有无明显的缺陷。原则上讲,它们应

得到肯定的回答：

① 你所划分的任务数，是否至少高于目标机上处理器数目的一个量级？如果不是，则你在后继的设计步骤中将缺少灵活性。

② 你的划分是否避免了冗余的计算和存储要求，如果不是，则所产生的算法对大型问题可能是不可扩放的。

③ 诸划分的任务是否尺寸大致相当，如果不是，则分配处理器时很难做到工作量均衡。

④ 划分的任务数是否与问题尺寸成比例？理想情况下，问题尺寸的增加应引起任务数的增加而不是任务尺寸的增加。如果不是这样，则你的算法可能不能求解更大的问题，尽管有更多的处理器。

⑤ 确认你是否采用了几种不同的划分法，多考虑几种选择可提高灵活性，同时既要考虑域分解又要考虑功能分解。

7.3 通　　信

由划分所产生的诸并行执行的任务，一般而言都不能完全独立执行，一个任务中的计算可能需要用到另一个任务中的数据，从而就产生了通信要求。所谓通信，就是为了进行并行计算，诸任务之间所需进行的数据传输。

在域分解中，通常难以确定通信要求，因为将数据划分成不相交的子集并未充分考虑在数据上执行操作时可能产生的数据交换，在功能分解时，通常容易确定通信要求，因为并行算法中诸任务之间的数据流就相应于通信要求。

在讨论通信时，将通信分成以下四种模式：

① 局部/全局通信：局部通信时，每个任务只与较少的几个近邻通信；全局通信中，每个任务与很多别的任务通信。

② 结构化/非结构化通信：结构化通信时，一个任务和其近邻形成规整结构（如树、网格等）；非结构化通信中，通信网则可能是任意图。

③ 静态/动态通信：静态通信，通信伙伴的身份不随时间改变，动态通信中，通信伙伴的身份则可能由运行时所计算的数据决定且是可变的。

④ 同步/异步通信：同步通信时，接收方和发送方协同操作；异步通信中，接收方获取数据无需与发送方协同。

7.3.1 局部通信

当某一操作仅要求从近邻的其它任务获取数据时，就呈现局部通信（Local

Communication)模式。一个典型的例子就是数值计算中的雅可比有限差分法(Finite Difference Method)。对于二维网格,可以使用5点格式(Five_Point Stencil)计算诸格点 x 之值(参见第十章第10.4.2节):

$$x_{i,j}(k)=\frac{4x_{i,j}(k-1)+x_{i-1,j}(k-1)+x_{i+1,j}(k-1)+x_{i,j-1}(k-1)+x_{i,j+1}(k-1)}{8} \tag{7.1}$$

在迭代过程中,要求计算一系列 $x_{i,j}(1)$,$x_{i,j}(2)$,$x_{i,j}(3)$ …之值。如图7.4,每个 $x_{i,j}(k)$ 之计算,都用其周围四个格点的值,其算法如下:

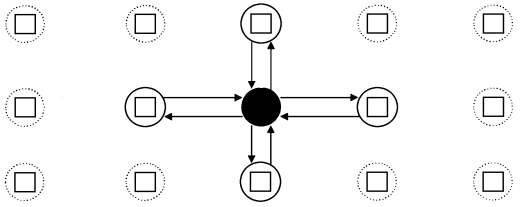


图7.4 二维网格雅可比5点差分格式的局部(四近邻)通信模式

```
Begin
  for k = 1 to T do
    (1) send x_{i,j}(k-1) to each neighbor
    (2) receive x_{i-1,j}(k-1), x_{i+1,j}(k-1), x_{i,j-1}(k-1), x_{i,j+1}(k-1) from
        neighbors
    (3) compute x_{i,j}(k) using Equation (7.1)
  end for
End
```

雅可比方法易于并行化,所有格点可同时更新,但收敛速度较慢。改进的Gauss-Seidel法如下:

$$x_{i,j}(k)=\frac{4x_{i,j}(k-1)+x_{i-1,j}(k)+x_{i+1,j}(k-1)+x_{i,j-1}(k)+x_{i,j+1}(k-1)}{8} \tag{7.2}$$

它迭代时利用了一半新值,一半旧值,虽加快串行迭代速度,但都不易并行化,为此出现了不少变体方案,如著名的红-黑着色(Red_Black Coloring)法(详见第十章有关章节)。总之,简单的Gauss_Seidel更新策略在串行算法中很有效,而在并行机上不理想,朴素的雅可比更新策略在并行机上很有效,但收敛不快,而红黑着色的更新策略组合了它们的优点,故应用很广。

7.3.2 全局通信

全局通信(Global Communication)系指有很多任务参与交换数据的一种通信模式,这种方式可能导致过多的通信从而限制了并行执行的机会。

一个顺序求和算法 $S=\sum_{i=0}^{N-1} x_i$,就呈现出全局通信要求。如图7.5所示,一个根进程 S,负责从 n 个分布式任务中一次接收一个值并相加之:

$$S=x_i+S_{i-1} \tag{7.3}$$

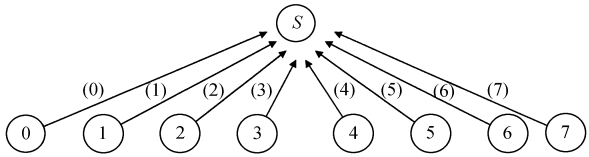


图7.5 集中式顺序求和

显然,这种集中控制式的依次求和方式,妨碍了有效并行执行,为此,可以使用分治策略来开拓求和的并行性:

$$\sum_{i=0}^{2^n-1}=\sum_{i=0}^{2^{n-1}-1}+\sum_{i=2^{n-1}}^{2^n-1} \tag{7.4}$$

上式右边的两个求和可同时进行,且每个仍可按同样方式进一步分解之。图7.6示出了 $N=8$ 的分治求和树,由此可见,分治法一方面提高了算法的并行度;同时也可将全局通信化为局部通信。

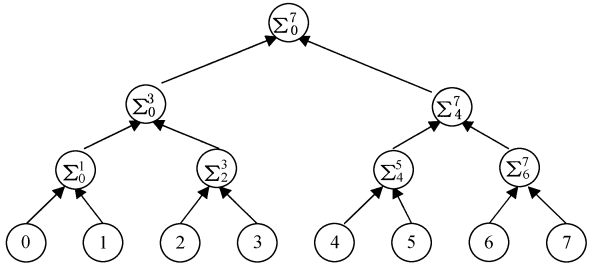


图7.6 $N=8$ 时的分治求和树

7.3.3 非结构化、动态和异步通信

在静态的、结构化通信中,一个任务的通信伙伴形成一个规则的模式(如树、网格等),并且不随时间变化;而非结构化、动态通信则不然,模式不规整且又随时间变化。

非结构化的通信模式,对算法设计的前期不会造成实质性的困难,但却复杂化了任务的组合与处理器的映射,特别是当要求组合策略既能创建尺寸大致相等的任务,又能尽量减少任务间的通信要求时,就要求非常复杂的算法,而这些算法在通信要求是动态时又会在算法的执行中频繁地使用,所以必须权衡这些算法的利弊。

在同步通信中,通信中的双方均知道何时将要产生通信操作,所以发送者就显式地发送数据给接收者,在异步通信(Asynchronous Communication)中,发送者不能确定接收者何时需要数据,所以接收者要明确地(显式地)从发送者请求数据。例如,消息传递(Message_Passing)式通信方式就属异步通信。

7.3.4 通信判据

下面的一组问答题,可检验你所作的通信设计是否合理。原则上讲,应给予它们肯定的回答:

① 所有任务是否均执行大致同样多的通信?如果不是,则所设计的算法的可扩放性可能是不好的。

② 每个任务是否只与少许的近邻相通信?如果不是,则可能导致全局通信,在此情况下,应设法将全局通信结构化为局部通信结构。

③ 诸通信操作是否能并行执行?如果不是,则所设计的算法很可能是低效的和不可扩放的,在此情况下,设法试用分治技术来开发并行性。

④ 不同任务的计算能否并行执行?如果不是,则所设计的算法很可能是低效的和不可扩放的,在此情况下,可考虑重新安排通信/计算之次序等来改善之。

7.4 组 合

在设计的第一和第二阶段,划分了任务并考虑了任务间的通信,不过所得到的算法仍是抽象的,因为并未考虑它在任何特定的并行机上的执行效率,在第三阶段,即组合(Agglomeration)阶段,将从抽象转到具体,即重新考察在划分和通信阶

段所作的抉择,力图得到一个在某一类并行机上能有效执行的并行算法。

组合的目的是通过合并小尺寸的任务来减少任务数,但它仍可能多于处理器的数目,理想的情况是,在组合时就将任务数减少到恰好每个处理器上一个,从而得到一个 SPMD 的程序,这时映射也宣告完成,另外,组合时我们也要考虑是否值得进行数据和/或计算的重复(Replication)。

用增加计算和通信的粒度的办法可减少通信成本,组合时要保持足够的灵活性同时要减少软件工程代价。这几个相互矛盾的准则在组合阶段要仔细考虑。

7.4.1 增加粒度

在设计过程的划分阶段,致力于定义尽可能多的任务以增大并行执行的机会。但是定义大量的细粒度任务不一定能产生一个有效的并行算法,因为大量细粒度任务有可能增加通信代价和任务创建代价。

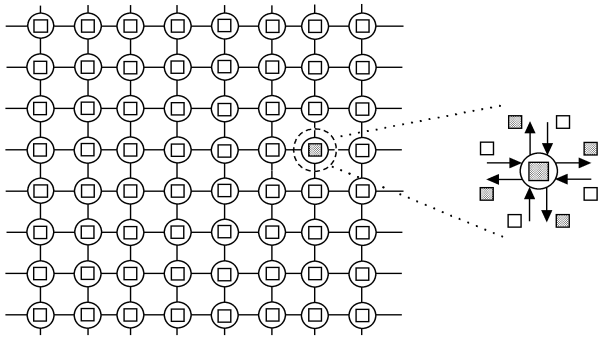
表面-容积效应(Surface to Volume Effects) 如果每个任务的通信伙伴是少的,则增加划分粒度常能减少通信次数,同时还能减少总通信量,参照图 7.7 来阐述所谓"表面-容积效应":一个任务的通信需求比例于它所操作的子域的表面积,而计算需求却比例于子域的容积。在一个二维问题中,"表面积"比例于问题的尺寸,而"容积"比例于问题尺寸的平方。因此一个计算单元的通信/计算之比随问题(任务)尺寸的增加而减少。

图 7.7(a)和(b)分别示出了细粒度和粗粒度的二维网格上 5 点法计算有限差分。在(a)中,8×8 的网格上,计算被划分成 8×8=64 个任务,每个任务负责计算一个格点;而在(b)中,同一计算问题被划分成 2×2=4 个任务,每个任务负责 16 个格点的计算。在(a)中,共需要 64×4=256 次通信,每个任务通信 4 次,总共传输 256 个数据值;而在(b)中,仅要求 4×4=16 次通信,总共传输 16×4=64 个数据值,可见同一计算问题,粗粒度划分的通信次数和通信量均比细粒度划分时有所下降。

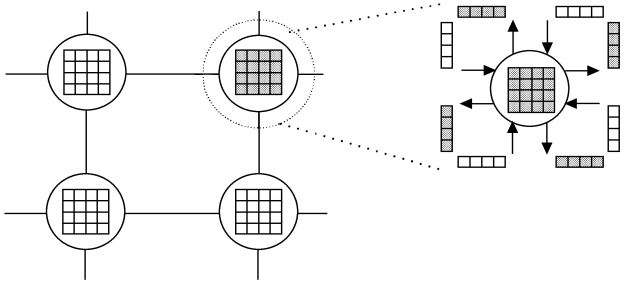
表面-容积效应启发我们,在其它条件等同的情况下,高维分解一般更有效。因为相对于一个给定的容积(计算)它减少了表面积(通信)。因此从效率的角度,增加粒度的最好办法是在所有的维组合任务。

重复计算(Replication Computation) 它也称为冗余计算。有时候可以采用不必要的多余的计算的方法来减少通信要求和/或执行时间。仍以二叉树上的求和 $\sum_{i=0}^{N-1} x_i$ 为例来说明之。

假定在二叉树上用 N 个处理器来求 N 个数的全和,且要求每个处理器上均保持最终的全和。为此如图 7.8 所示,必须先自叶向根逐级求和;然后自根向叶将



(a) 细粒度的二维网格



(b) 粗粒度的二维网格

图 7.7　二维网格 5 点差分格式中增加粒度对通信成本的影响

全和逐级播送给每个处理器，共需 $2\log N$ 步。

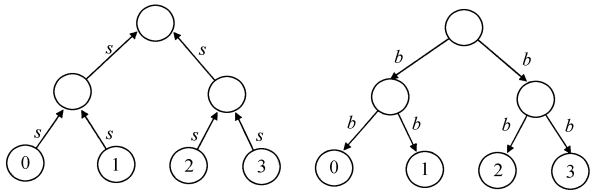


图 7.8　二叉树上求全和

以上述方式求和时，处理器的利用率是逐级减半的，如果在树的每一级都充分利用所有的处理器参与计算，即在树的每一级，每个处理器（任务）均接收两个数据，执行一次局部求和，然后再发送结果至下一级的两个处理器，那么经过 $\log N$

级后,每个处理器中均积累了全和 $\sum_{i=0}^{N-1} x_i$。如图 7.9 所示,这实际上就是蝶式通信结构,它利用了重复计算的方法,只需 $\log N$ 步就可在各处理器中积累了最终的全和,但却以总共施行 $O(N\log N)$ 次计算和通信操作为代价。

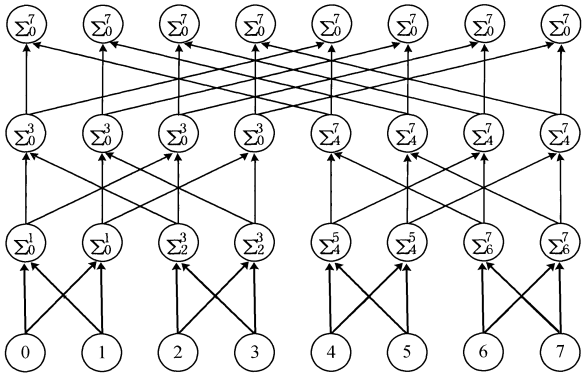


图 7.9 使用蝶式通信结构求全和

当对通信需求分析揭示了一组任务不能同时执行时,组合总是有益的,例如,用图 7.8 所示二叉树和图 7.9 所示蝶式图进行求和时,只有在树的和蝶式图的同一级中的任务方能同时执行,这样不同级中的任务可组合到一起而不会减少并行执行的机会,同时也可能产生优化的通信结构 (参见习题 7.3 和 7.4)。

7.4.2 保持灵活性和减少软件工程成本

组合的主要目的是提高效率和减少通信成本,但也要注意保持足够的灵活性和降低软件工程代价。

在组合时,易于作出对算法的可扩放性带来不必要限制的决定,要维持一个算法 (或程序) 的可移植性和可扩放性,则创建可变数目的任务是很关键的,而组合时往往会使问题的任务数的变化范围受到限制。为了易于在映射时平衡负载,组合后的任务数,按照经验,至少应比处理器的数目多一个量级。其最优的任务数可由分析模型和实际经验共同决定。但灵活性也不一定需要一个设计总是创建大量的任务。粒度可由编译或运行参数来控制。重要的是,一个设计不要对能够生成的任务数加入不必要的限制。

组合时另一个应关心的问题是尽量减少软件工程代价,当并行化一个串行代

码时,应尽量减少代码的巨大变化以减少软件开发开销,由此观点出发,最感兴趣的策略就是那些在算法设计时不需要大量修改代码的策略。

7.4.3 组合判据

下面的一组问答题,可检验组合设计是否合理,原则上讲,应给予它们肯定的回答:

① 用增加局部性方法施行组合是否减少了通信成本?如果不是,检查能否由别的组合策略来达到。

② 如果组合已造成重复计算,是否已权衡了其得益?

③ 如果组合已重复了数据,是否已证实这不会因限制问题尺寸和处理器数的变化范围而牺牲了可扩放性?

④ 由组合所产生的任务是否具有类似的计算和通信代价?

⑤ 任务数目是否仍然与问题尺寸成比例?如果不是,算法则是不可扩放的。

⑥ 如果组合减少了并行执行的机会,是否已证实现在的并发性仍能适应目前和将来的并行机?

⑦ 在不导致负载不平衡,不增加软件工程代价和不减少可扩放性的前提下,任务数能否再进一步减少,在其它条件等同时,创建较少的大粒度的任务算法通常是简单高效的。

⑧ 如果并行化现有的串行程序,是否考虑了修改串行代码所增加的成本?如果此成本是高的,应考虑别的组合策略,它能增加代码重用的机会。

7.5 映 射

在设计的最后阶段,我们要指定每个任务到哪里去执行,此即映射(Mapping)。开发映射的主要目的是减少算法的总的执行时间,其策略有二:一是把那些能够并发执行的任务放在不同的处理器上以增强并行度;二是把那些需频繁通信的任务置于同一个处理器上以提高局部性。这两个策略有时会冲突,这就需要权衡。

对于两个处理器而言,映射问题有最佳解;当处理器的数目大于等于3时,此问题是NP完全问题,但我们可以利用关于特定策略的知识,用启发式算法来获得有效的解。

在进行映射时,对于许多用域分解技术开发的算法,有固定数目的等尺寸任

务,有结构化的局部/全局通信,此时映射很简单;对于更复杂的域分解算法,每个任务的工作量可能不一样,通信也许是非结构化的,有效的组合就不那么容易,此时可能要采用负载平衡算法。

在基于功能分解的算法中,常常会产生一些由短暂任务组成的计算,它们只在执行的开始与结束时需与别的任务协调,此时我们可以用任务调度(Task_Scheduling)算法来分配任务给那些可能处于空闲状态的处理器。

7.5.1 负载平衡算法

基于域分解技术的算法有很多专用和通用的负载平衡技术(Load_Balancing Techniques),它们都是试图将划分阶段产生的细粒度任务组合成每一个处理器一个粗粒度的任务。下面简单介绍几种有代表性的方法。

递归对剖(Recursive Bisection) 递归对剖技术用来将一个域划分成计算成本大致相等的子域,同时试图使通信代价最小,为此常使用分治方法:域首先在一维方向上分割成两个子域,然后分割以递归的方式在两个子域中进行,直至子域数和所要求的任务数一样多。

最直截了当的递归对剖技术是递归坐标对剖(Recursive Coordinate Bisection),它通常应用于非规整格点。这种技术所作的分割是基于域中物理格点坐标,每一步都沿较长的维进行划分。其优点是简单和低价,其缺点是不能优化通信性能。

递归对剖的一种变体叫做非平衡递归对剖(Unbalanced Recursive Bisection),它使用具有较好长宽比(Aspects Ratios)的子格点以降低通信成本,虽增加了计算划分的成本但却减少了通信成本。

还有一种称为递归图对剖(Recursive Graph Bisection)的技术,对复杂的非结构化的格点很有用,它使用连通信息来减少跨子域的格边数,从而可降低通信要求。

局部算法 上述描述的技术是很昂贵的,因为它们都要求计算状态的全局知识。相反,局部负载平衡算法,只是使用邻近处理器的负载信息,来周期性地调整自己的负载,或转移到邻居,或从邻居迁入。例如,对于如图7.10所示的网格问题,每个处理器周期地与周围的处理器比较它们的计算负载,如果它们的差异超过了某个阈值就施行计算负载的迁移。

概率方法(Probabilistic Method) 这是一种特别简单的方法,它将任务随机地分配给处理器。如果任务数足够多,则每个处理器预计能分配大致等量的计算,其优点是低价和可扩放性,缺点是要求跨处理器的通信和只有当任务数多于处理器数时才可能达到预期的负载分配。概率方法,当任务间很少有通信,和/或通信模式很少呈局部性时,才有可能最有效。

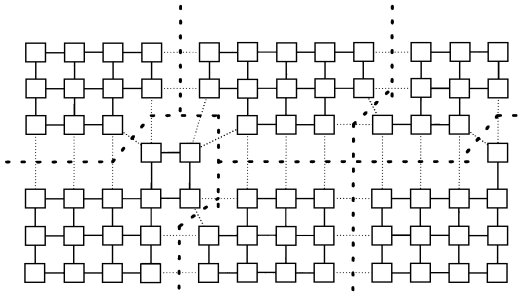


图 7.10 网格问题中的负载平衡

循环映射 (Cyclic Mapping) 如果知道每个格点计算负载是变化的,且呈现明显的空间局部性,则我们可以使用所谓循环指派法,即采用某种枚举方法,轮流地将各处理器分配给诸计算任务。这种方法可能使负载平衡,但牺牲了局部性且通信可能会增加。

此外,块循环分配 (Block Cyclic Distribution) 也是一种可能的处理器映射方法,此时,任务按块的形式轮流分配给处理器。

总之,递归对剖法需全局知识,性能好但代价高;局部算法代价小,但当负载变化大时调整很慢;概率方法代价低,可扩放性好,但通信代价可能较大,且只适用于任务数远多于处理器数时;循环映射技术实际是概率映射的一种形式,而概率方法比其它技术易于导致可观的通信。

7.5.2 任务调度算法

当使用功能分解时,产生了很多弱局部性要求的任务。这些任务通常放在集中的或分散的任务池中,然后使用任务调度算法,将池中的任务分配给特定的处理器。

任务调度算法 (Task Scheduling Algorithm) 最关键之处是任务分配策略。策略的选择应在独立运算 (以减少通信成本) 和计算状态的全局知识 (以改善负载平衡) 之间折衷,常用的调度模式有经理/雇员方式和非集中方式,在非集中调度方案中,怎样检测整个问题的求解是否已经完成也是一个必须考虑的问题。

经理/雇员模式 如图 7.11 所示,中心 (经理) 任务负责任务分配,每个雇员重复地从经理那里请求并执行具体任务。此策略的有效性依赖于雇员数和执行任务的成本,使用预取方法 (以使计算和通信重叠) 和缓存方法 (使得雇员无任务时经理和雇员才通信) 可以改善效率。这种方案的一种变体是层次经理/雇员模式,它将诸雇员分成不相交的集合,每一个都有一个小经理,雇员们从小经理那里领取任

务,并周期地与经理和小经理施行通信。

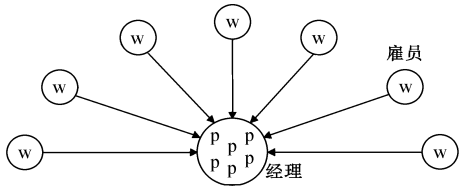


图 7.11 经理/雇员调度模式

非集中模式 它也就是无中心管理者的分布式调度法。在每个处理器中均维持一个任务池,这些任务池实际上就变成了可供请求者异步访问的分布的数据结构。

结束检测 任务调度算法需要一种机构检测何时操作结束,否则,空闲的雇员们将永不停止地发出工作请求,这种检测机构在集中式算法中容易实现,因为经理能容易地决定何时雇员们都空闲了;在分布式算法中则较困难,因为没有一个中央机构记录雇员空闲状况。

7.5.3 映射判据

下面的一组问答题,可检验你所作的映射设计是否合理。原则上讲,应给予它们肯定的回答:

① 如果采用集中式负载平衡方案,你是否已验证中央管理者不会成为瓶颈?

② 如果采用动态负载平衡方案,你是否衡量过不同策略的成本?

③ 如果采用概率或循环法,你是否有足够多的任务来保证合理的负载平衡?典型地,任务数应10倍于处理器数。

④ 如果要为一个复杂问题设计一个SPMD程序,你是否考虑过基于动态任务创建和消除的算法;后者可能得到一个更简单的算法,但性能可能有问题。

7.6 小结和导读

小结 本章已经描述了并行算法设计的四个步骤:首先将给定问题划分成一些小的任务,划分方法可以使用域分解法或功能分解法;其次分析诸任务之间的通信需求,通信可以是局部的和全局的,静态的和动态的,结构化的和非结构化的以及同步的和异步的;然后使用组合方法,在尽可能保持灵活性的同时,减少通信和开发成本;最终以最小化总执行时间为目标将诸任务分配给处理器,使用负载平衡

和任务调度技术可以改善映射的质量,在每个设计步骤之后均列出了相应的设计检验准则以评价设计的好坏。

经过上述四个步骤设计出的并行算法,在开始编码实现之前尚须考虑以下几件事:对算法进行性能分析以选择好的算法,根据要求证实所作的设计是否满意,考虑算法具体实现时的代价和代码重用的可能性;最终考虑如何将算法装配到一个更大的系统中去。

通过上面的学习之后,我们最好以一个实例来加深课文中所述设计原理的理解以及阐述开发设计一个并行算法的逐步过程,为此我们在本章的习题中给出了一个综合练习,通过它读者一方面可以了解针对给定问题设计一个并行算法的全过程;另一方面也促使读者独立回答一些问题,以达到复习巩固的目的。

导读　论述算法程序设计过程的一篇经典文章是[139]。在实用的、可扩放的并行机上设计并行算法,读者可参阅[69,71]。并行程序的设计,读者参阅[41]。映射问题的研究在计算机科学中很受重视。有关递归坐标对剖算法,读者可参阅[28];非平衡递归对剖算法由[105]提出,使用连通信息的递归对剖算法,读者可参阅[142];[160]等比较了划分非结构化网孔的不同算法,包括坐标对剖部,图对剖等;[69,131]等描述了循环分解法:[120]等描述了局部算法;有关调度结束检测算法,读者可参阅[59,148]。其它相关的文献包括[29,86,56,89,121,113,152]。[67]中的第2章集中地讨论了并行算法的设计方法学,章末的注释中列举了大量的有关此方面的参考文献,有兴趣的读者可追踪阅读。

习　　题

7.1　算法7.1是一个组合搜索算法,每当调用该算法扩展一个搜索树节点时,就检查该节点是否代表了一个问题的解:如果不是,则算法作递归调用以扩展其子节点。

```
算法 7.1　SISD 上组合搜索算法
输入:  给定问题 A
输出:  问题 A 的解
Procedure :SEARCH (A)
Begin
  if (Solution (A)) then
        Score = eval (A)
        Report Solution and Score
              else
        for each child A (i) of A do
```

```
                SEARCH (A (j))
            endfor
        endif
    End
```

(1) 如何构造此问题的并行搜索算法？

7.2 分治策略是常用的问题求解技术，其算法可形式描述如下：

```
算法 7.2  SISD 上分治 (D&C) 算法
输入： 问题输入集
输出： 问题的解
Begin
  if base case then solve problem
    else
      (1) partition problem into subproblem L and R
      (2) solve L using D&C
      (3) solve R using D&C
      (4) combine solutions to problem L and R
  endif
End
```

试用此算法求解 $N=32$ 的全和，并画出相应的求和树。

7.3 画出图 7.8 二叉树的通信图，并分析其计算和通信次数。

7.4 画出图 7.9 蝶式图的通信图，并分析其计算和通信次数。

7.5 (1) 对于 $12\times6=72$ 个网格点，如有 $p=12$ 个处理器，如何用循环指派法分配处理器？

(2) 对于 $8\times8=64$ 个网格点，如有 $p=4$ 个处理器，如何用循环指派法分配处理器？

(3) 对于 (1) 和 (2) 两种情况，如何用块循环指派法分配处理器？

7.6 设计并实现使用 3 点格式 (Three Point Stencil) 的 1_D 有限差分并行算法，假定每个处理器上分配一个任务，试分析所设计算法的性能。

7.7 设计并实现使用 5 点格式的 2_D 有限差分的 2_D 分解并行算法，假定每个处理器上分配一个任务。试分析该算法的性能。

7.8 试设计一个 1_D 和 2_D 格点的 Gauss_Seidel 并行算法。

7.9 综合练习—大气模型的算法设计。

(1) 背影知识：

大气模型 (Atmosphere Model) 是模拟影响气候的大气过程 (风、云、雨等) 的计算机程序，它可用来研究龙卷风的演变，预测明天的天气，或者研究大气中 CO_2 浓度对气候的影响。大气模型是求解一组 PDE 方程，它描述大气基本流体动力学行为。这组方程的连续空间行为可用空间中的有限的离散点的行为近似表征。典型地，这些离散点都位于长方形经纬网格 $N_x\times N_y\times$

N_z 上，其中 $N_x \approx 2N_y$，N_y 的区间为50～500，N_z 的区间为15～30（如图7.12所示）。在每个格点上用一向量表示，其值代表该格点的压力、温度、风速、湿度等。大气模型执行时间积分，它根据当前状态决定将来的大气状态，积分逐步前进，每计算一固定量就前进一步，假定使用9点格式（Nine_Point Stencil）有限差分方法修改格点在水平维运动之值，用3点格式修改垂直维运动之值（如图7.13所示）。大气模型涉及到在规整的三维网格上执行有限差分计算，它是一大类科学数值计算的代表，大气模型包含了如下的一组基本预测方程，其中 λ 和 λ 是纬线和经线，z 是高度，u 和 v 是速度的水平分量，w 是垂直分量，p 是压力，ρ 是密度，T 是温度，f 是洛伦兹力，g 是重力，F 和 Q 是外力，C_p 是热容，a 是地球半经，R 为常数：

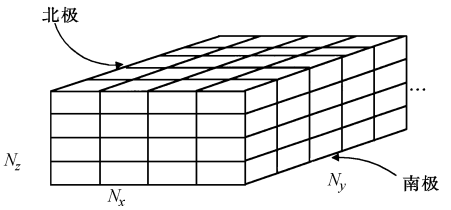


图7.12　大气状态三维格点表示

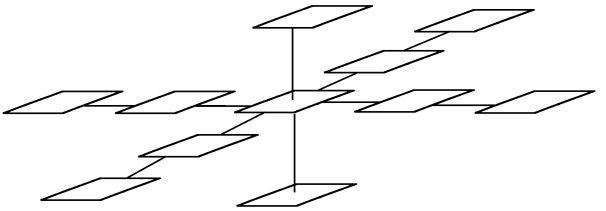


图7.13　大气模型中使用的有限差分格式

动量守恒 $\frac{du}{dt} - \left(f + u\frac{\tan}{a}\right) v = -\frac{1}{a\cos} \cdot \frac{\partial p}{\rho \partial \lambda} + F_\lambda$　　(7.5)

$$\frac{dv}{dt} + \left(f + u\frac{\tan}{a}\right) u = \frac{\partial p}{\rho a \partial} + F$$　　(7.6)

流体静力学方程：$g = -\frac{\partial p}{\rho \partial z}$　　(7.7)

质量守恒 $\frac{\partial p}{\partial t} = -\frac{1}{a\cos}\left[\frac{\partial}{\partial \lambda}(\rho u) + \frac{\partial}{\partial}(\rho v\cos)\right] - \frac{\partial}{\partial z}(\rho w)$　　(7.8)

能量守恒：$C_p \frac{dT}{dt} - \frac{1}{\rho}\frac{dp}{dt} = Q$

大气状态方程：$p = \rho RT$　　(7.9)

Q 设计分析：

划分　在大气模型中，可用格点代表大气的状态，所以域分解是很自然的。参照图7.2，分

解在 x、y、z 方向上均是可能的。如果为了开拓最大并行度，则可将每个格点定义为一个任务，试问总共有多少个任务？

通信　在大气模型中有三种不同形式的通信：①有限差分格点计算引起的通信：在细粒度分解时，每个格点代表一个任务。参照图 7.13，如在水平维使用 9 点格式，垂直使用 3 点格式，则格点间的通信量是多少？②全局计算所引起的通信：大气模型中，要周期性地计算大气质量，令 $M_{i,j,k}$ 表示格点 (i,j,k) 处的大气质量，则总质量为：

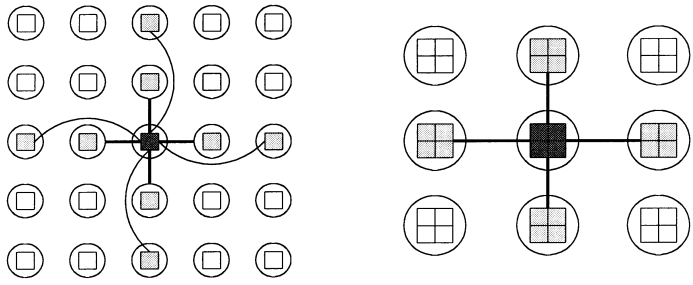
$$Total\ Mass = \sum_{i=0}^{N_x-1}\sum_{j=0}^{N_y-1}\sum_{k=0}^{N_z-1} M_{i,j,k} \tag{7.10}$$

试问如何并行计算 (7.10) 式？③物理计算所引起的通信：由于一个格点视为一个任务，则大气模型物理分量的计算要求相当多的通信，例如在第 k 层 ($k \geq 1$) 的全晴空 (TCS) (Total Clear Sky) 定义如下：

$$TCS_k = \prod_{i=1}^{k} (1-cld_i)\quad TCS_i = TCS_{k-1}(1-cld_k) \tag{7.11}$$

其中，第 0 层是大气顶层，cld_i 是第 i 层的云的比例小数。试问如何并行计算 (7.11) 式？此时每个格点每个时间步总共需要多少次通信？试用通信判据评价一下所作的通信设计。

组合　当使用细粒度的域分解时，任务数似乎是大的，似应进行组合：参照图 7.14，少量的组合 (每个任务从 1 个格点到 4 个格点) 能减少 9 点格式计算所引起的通信需求 (每个任务每个时间步从 8 条信息到 4 条信息)；水平维通信需求相对较小，仅 4 条信息 (8 个数据)，而在垂直时，除了有限差分格式的计算所需的通信外 (2 条信息，2 个数据)，还有不同物理分量的计算 (见 (7.11) 式)，所以组合应在哪个方向上进行？从软件工程的角度，为了减少开销，应允许现行的串行代码尽量能在并行程序中不加修改地被重用，试问组合应在什么方向上进行？为什么？经过上述的组合分析，组合后任务数大约是多少？



(a) 每个任务 1 个格点　　(b) 每个任务 4 个格点

图 7.14　大气模型中的组合方法

映射　在不存在负载不平衡的情况下，可以使用图 7.15 所示的简单映射策略，每个处理器指派给单一任务。此任务负责多列的计算，从而可产生一个 SPMD 程序，此分配策略如果每个任务每个时间步内执行等量计算，则是很有效的，但在大气模型中其物理分量计算变化相当大，例如辐射计算不能在晚上，云雾形成必须温度超过一定阈值。

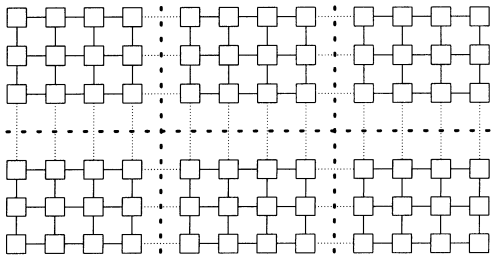


图 7.15 大气模型中的均匀映射

此类计算负载的变化能导至效率下降 20%或更多,当此值不能接受时,应考虑别的映射方式,例如循环指派法,试解释采用循环映射的大气模型为何能改善负载平衡状态?

最后请你给出一个简短结论:即大气模型应使用几维划分?什么类型的分解?应该怎样组合?如何进行映射?

第三篇　并行数值算法

这一篇主要研究数值计算问题的并行算法，包括矩阵运算（第九章）、线性方程组的求解（第十章）和快速傅里叶变换（第十一章）。其中，矩阵运算是数值计算中最重要的一类运算，是线性代数和数值分析中最基本的运算操作；稠密和稀疏线性方程组是大型科学工程计算中最常用的数值计算问题；快速离散变换是数字信号处理等应用领域中最主要的数值计算形式。这些数值计算问题的特点是：①计算量一般都涉及到一个具体的物理量，其数值可用实数（即浮点数），也可用虚数表示；②求解问题的依据是基于数值分析中的数学原理；③求解问题的算法可用直接法，但更普遍的是迭代法，且在理论上讲，每次迭代都应改善前一步的计算结果；④计算结果，一般均应是满足预定精度要求的近似解。在讨论算法时，对数值计算的稳定性、收敛速度和精度分析等基本问题，乃是直接利用数值分析中的数学原理和结论，而着重讨论计算问题的基本原理，设计串行算法和分析串行算法中的并行性及并行化方法，对于有些典型问题还给出相应的并行算法。

由于任何并行算法，在执行时总会涉及到处理器之间的相互通信问题，所以本篇一开始（第八章）还讨论了在环、二维网孔和超立方等结构上，采用确定选路算法和存储转发与切通开关技术，进行一到一、一到多和多到多等一些基本通信操作。这些并非针对数值计算的问题，对于非数值并行算法也同样存在。

第八章　基本通信操作

大多数并行算法均会涉及到处理器间的数据(本章下文称消息)交换。数据交换方式可归结为处理器间的通信操作,而通信操作各式各样,本章只研究下一章矩阵运算中所使用的最频繁的一些基本通信操作,包括研究通信操作与处理器之间的互连拓扑,通信选路策略和消息传递机制关系。本章讨论采用确定的**维序**(Dimension Ordering)选路方法,使用**存储转发**(Store and Forward,简记之为SF)和**切通**(Cut Through,简记之为CT)开关技术,在**环、二维网孔和超立方**等连接上进行**一到一、一到多、多到多**等一些基本通信操作。

8.1 选路方法与开关技术

8.1.1 选路方法

所谓选路(Routing)系指消息从发源地到达目的地所取的走法,即行进的方法。一般可分为最短(Minimal)法和非最短(Nonminimal)法,有时相应地称为贪心(Greedy)法和随机(Random)法。贪心法总是在源和目的之间试图选择最短的路径,但往往会造成拥挤。随机法虽可能选路长度较长,但不易造成拥挤。选路方法也可分为确定的(Deterministic)和自适应的(Adaptive):前者在源和目的之间确定一条惟一的路径;后者根据行进中的网络状态信息而自动地确定路径。

一种最常用确定的最短选路法是维序选路(Dimension Ordered Routing)法,它是根据通信通道(下文也叫作链路、信道等)的维来确定消息如何穿越相继的通道。当此方法用于二维网孔中时就称为 X_Y 选路法(X_Y Routing),用于超立方网络中时就称为 E_立方选路法(E_Cube Routing)。

X_Y选路法 在二维网孔中选路时,首先沿 X 维方向确定路径,然后再沿 Y 维方向确定路径。

算法 8.1 二维网孔上的 X_Y 选路算法
输入: 待选路的信包(Packet)处于源处理器中
输出: 将各信包送至各自的目的地中
Begin
(1) 沿 X 维将信包向左或向右选路至目的处理器所在的列
(2) 沿 Y 维将信包向上或向下选路至目的处理器所在的行
End

例 8.1 图 8.1 中示出了 4 个(源;目的)对的 X_Y 选路过程,其中节点(2,1)要选路到节点(7,6),节点(0,7)要选路到节点(4,2),节点(5,4)要选路到节点(2,0),节点(6,3)要选路到节点(1,5)。显然它们不会出现死锁或循环等待现象。□

E_立方选路法 对于 $N=2^n$ 个节点的 n 维立方体,令源节点的二进制编码为 $S=s_{n-1}\cdots s_1 s_0$,目的节点的二进制编码为 $D=d_{n-1}\cdots d_1 d_0$。将 n 维表示成 $i=1,2,\cdots,n$,其中第 i 维对应于节点地址的第 $i-1$ 位。设 $V=v_{n-1},\cdots,v_1 v_0$ 是路径

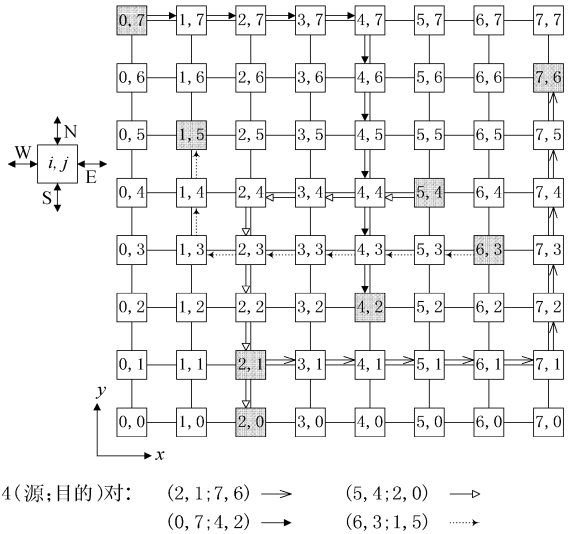


图 8.1 8×8 二维网孔中 4 个(源;目的)对的 X_Y 选路

中的任一中间节点,则确定一条从源 S 到目的 D 的 E-立方选路算法如下:

算法 8.2 超立方网络上的 E_立方选路算法
输入: 待选路的信包在源处理器中
输出: 将源处理器中的信包送至其目的地
Begin
(1) for $i=1$ to n do /*计算方向位*/
$r_i = s_{i-1} \oplus d_{i-1}$ /*⊕为异或运算符*/
end for
(2) $i=1$, $V=S$ /*初始化*/
(3) while $i \le n$ do
(3.1) if $r_i=1$ then 从当前节点 V 选路到节点为 $V\ 2^{i-1}$ endif
(3.2) $i=i+1$
endwhile
End

例 8.2　图8.2中示出了4维超立方的 E-立方选路过程,其中 $n=4$, $S=0110$, $D=1101$, $R=r_4r_3r_2r_1=1011$。由于 $r_1=0\oplus1=1$,所以 S的下一节点为 $S\oplus2^0=0111$;同样由于 $r_2=1\oplus0=1$,所以 $V=0111$的下一节点为 $V\oplus2^1=0101$;由于 $r_3=1\oplus1=0$,所以跳过第 $i=3$ 维;最后由于 $r_4=0\oplus1=1$,所以 $V=0101$就选路到节点 $V\oplus2^3=1101$,它就是目的地。□

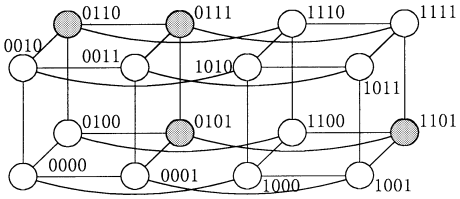


图 8.2　4维超立方中的 E-立方选路

8.1.2　开关技术

消息格式　通信中乐于使用消息(Message)这一术语,它是节点间通信的逻辑单位,如图 8.3 所示,通常由一些定长的信包(Packet)组成。信包是带有选路信息的基本通信单位,可以把它分成一些定长的数据片,其中选路信息和顺序号作为(包)头,其余的是数据。消息格式的改进使开关技术由存储转发式演变成更先进的虫蚀(Wormhole)方式。

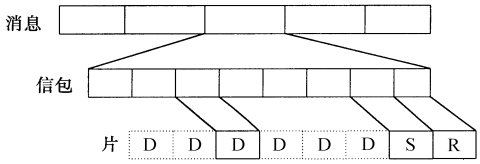


图 8.3　消息、信包和片的格式

存储转发(SF)选路　在存储转发选路(Store and Forward Routing)中信包是基本的传输单位。在传输过程中,中间节点必须收齐且存储在缓冲器中后,它才可能传向下一节点(如图 8.4(a)所示)。

令 t_s 是启动时间(包括打包、执行选路算法和建立通信介面的时间),t_h 是节点延迟时间(即包头穿越网络中两直接相连的处理器所需的时间),t_w 是传输每个

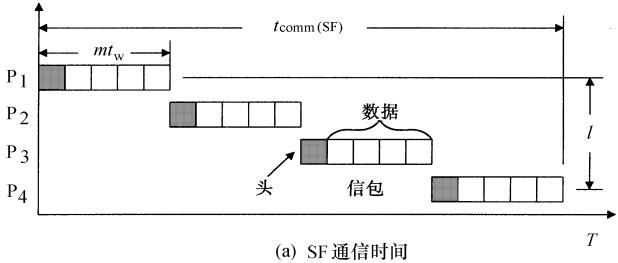
字的时间(它是带宽的倒数)。

对于长度为 m 的信包,穿越 l 条链路,在存储转发的网络中总通信时间为:

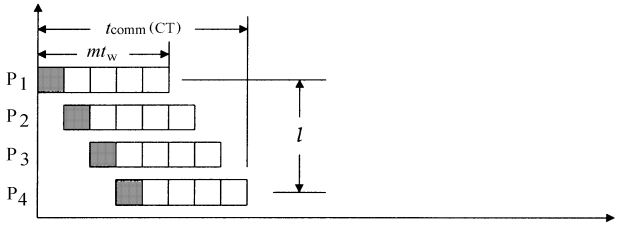
$$t_{comm}(SF) = t_s + (mt_w + t_h)\,l \tag{8.1}$$

如果 $t_h = t_s = 0$, $t_w = 1$,则

$$t_{comm}(SF) = O(m \cdot l) \tag{8.1a}$$



(a) SF 通信时间



(b) CT 通信时间

图 8.4 两种开关技术的时间比较

切通(CT)选路 在切通网络中将信包进一步分成更小的片(数据片和包头)进行传输。虫蚀(Wormhole Routing)选路是切通选路(Cut_Through Routing)的一种形式。在传输过程中,中间节点只备有很小的片缓冲器,一旦收到包头就传至下一节点。同一信包中的所有片一同以流水线方式穿越网络(如图 8.4(b)所示)。整个信包犹如一列火车,由火车头(包头)牵引着车厢(数据片)顺序前进。

对于长度为 m 的信包,穿越 l 条链路,在切通的网络中总通信时间为:

$$t_{comm}(CT) = t_s + mt_w + lt_h \tag{8.2}$$

如果 $t_s = 0$, $t_h = t_w = 1$,则

$$t_{comm}(CT) = O(m + l) \tag{8.2a}$$

一般情况下,t_s 是不容忽略的,而片的大小是远小于 m 的,同时 t_w 通常也比 t_h 大得多,所以(8.1)式和(8.2)式可近似写为:

$$t_{comm}(SF) = t_s + mt_w \cdot l \tag{8.1b}$$

$$t_{comm}(CT) = t_s + mt_w \tag{8.2b}$$

(8.1b) 和 (8.2b) 式表明了这样的事实:存储转发网络的延迟与源和目的之间的距离成正比,而切通网络的延迟时间与源和目的之间的距离无关。

8.2 单一信包一到一传输

单一信包施行点对点一到一传输是最基本的通信操作。当计算通信时间 t_{comm} 时,l 是个重要参数。在具有 p 个处理器规则互连的网络中,它可定量计算之。

l 的计算 对于 p 个处理器连成环形,l 至多为 $\lfloor p/2 \rfloor$;对于 $p \times p$ 个处理器连成周边环绕正方网孔 (Wraparound Square Mesh),l 至多为 $2\lfloor p/2 \rfloor$;对于 p 个处理器连成超立方体,l 至多为 $\log p$。一般的非规则互连的网络中的 l 并非容易计算。

SF 网络传输时间上界 单一信包用 SF 方式选路,在环上的通信时间为 $t_s + t_w m\lfloor p/2 \rfloor$;在周边环绕正方网孔上的通信时间为 $t_s + 2t_w m\lfloor p/2 \rfloor$;在超立方上的通信时间为 $t_s + t_w m \log p$。

CT 网络传输时间 使用 CT 方式,将单一信包从源直接发送至 l 远的目的地的通信时间为 $t_s + t_w m + t_h l$。对于 m 充分大的信包使用 CT 方式传输时,处理器之间传输单一信包的时间,与用 SF 方式在两直接相连的处理器之间传输单一信包的时间基本一样。

8.3 一到多播送

所谓一到多播送 (One to All Broadcast),是指开始时 P_0 中保存有信包 m,播送结束后所有其它处理器中均得到信包 m。一到多播送的反操作叫作单点累积 (Single Node Accumulation),两者在以后讨论的数值算法中均经常使用。

8.3.1 使用 SF 进行一到多播送

环 约定环是双向的,且每个处理器一次只能发送一条信包。在 8 个处理器环上,使用 SF 方式进行播送,其过程如图 8.5 所示,其中虚线箭头上的数字表示

发送时间步。

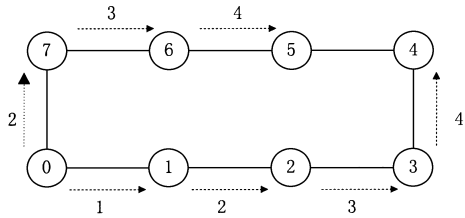


图 8.5 8 个处理器的环上以 SF 方式播送过程

显然,在 p 个处理器的环上,用 SF 方式进行播送的通信时间为:

$$t_{one-to-all}(SF) = (t_s + mt_w)\lceil p/2 \rceil \tag{8.3}$$

环绕网孔 一个 p 个处理器的正方网孔可以视为其行和列均由 p 个处理器所组成的环,所以可以使用环上播送方法来讨论网孔上的播送。在 16 个处理器环绕网孔上,使用 SF 方式进行播送,其过程如图 8.6 所示。同样,虚线箭头上的数字表示发送时间步。

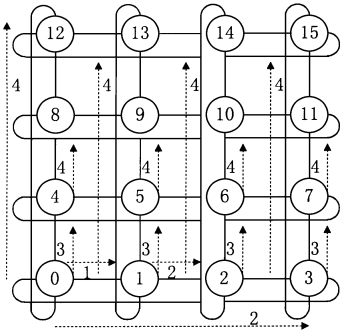


图 8.6 16 个处理器的环绕网孔上以 SF 方式播送过程

在 p 个处理器的环绕网孔上用 SF 方式进行播送时,先完成一行中的播送,再同时进行各列中的播送,显然其通信时间为:

$$t_{one-to-all}(SF) = 2(t_s + mt_w)\lceil p/2 \rceil \tag{8.4}$$

不难推知在三维网孔上施行播送的通信时间为:

$$t_{one-to-all}(SF) = 3(t_s + mt_w)\lceil p^{1/3}/2 \rceil \tag{8.5}$$

超立方 一个有 2^d 个处理器的超立方(Hypercube)可以视为每维仅有 2 个处

理器的 d 维网孔，因此网孔上的播送方法可以推广到超立方上。在 8 个处理器的超立方上，使用 SF 方式进行播送，其过程如图 8.7 所示。照例，虚线箭头上的数字表示发送时间步。

不难计算出 p 个处理器的超立方上，用 SF 方式进行播送通信时间为：

$$t_{one-to-all}(SF) = (t_s + mt_w)\log p \qquad (8.6)$$

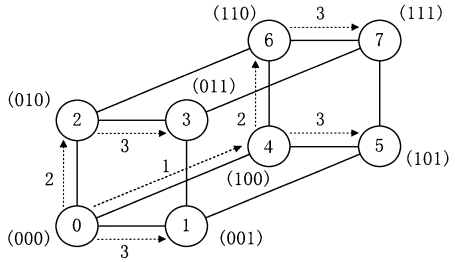


图 8.7　8 个处理器的超立方上以 SF 方式播送过程

8.3.2　使用 CT 进行一到多播送

环　因为使用 CT 选路时，通信时间与中继节点无关，所以可以将超立方上的播送算法直接映射到环上。为此，对于一个如图 8.8 所示的 8 个处理器环，可以参照图 8.7 先按高维播送，再按中维播送，最后按低维播送就可完成播送。剩下的问题是如何计算它的通信时间。

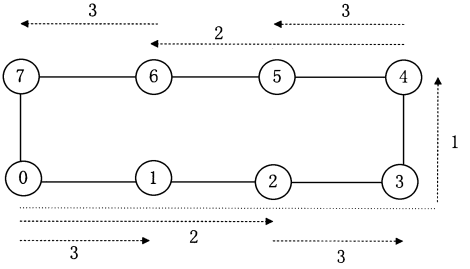


图 8.8　8 个处理器环上以 CT 方式播送过程

当按照图 8.8 所示的方式播送信包时，源处理器首先发送信包至 $p/2$ 远的处理器；其次是已收到信包的处理器将它发送至 $p/4$ 远的处理器。一般在第 i 步时

信包发送的距离为 $p/2^i$,其通信时间为 $t_s + mt_w + t_h p/2^i$。因此总的通信时间为:

$$t_{\text{one-to-all}}(\text{CT}) = \sum_{i=1}^{\log p} (t_s + mt_w + t_h p/2^i)$$

$$= t_s \log p + mt_w \log p + t_h (p-1) \qquad (8.7)$$

对于充分大的 m,t_h 可忽略不计,所以上式可简化为:

$$t_{\text{one-to-all}}(\text{CT}) = (t_s + mt_w) \log p \qquad (8.7a)$$

它比 (8.3) 式大大改进。

网孔　在如图 8.9 所示的二维网孔上用 CT 方式播送时可分为两步:先完成一行中 p 个处理器间的播送;再同时进行各列中 p 个处理器间的播送。参照 (8.7) 式,可知每步的通信时间为 $(t_s + mt_w)\log p + t_h(p-1)$,因此总的通信时间为:

$$t_{\text{one-to-all}}(\text{CT}) = (t_s + mt_w) \log p + 2t_h(p-1) \qquad (8.8)$$

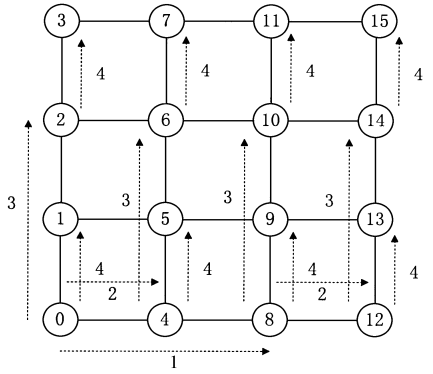


图 8.9　4×4 网孔上以 CT 方式播送过程

超立方　已如前述,一个 2^d 个处理器可视为每维只有 2 个处理器的 d 维网孔,所以在超立方上用 CT 方式播送其通信时间为:

$$t_{\text{one-to-all}}(\text{CT}) = (t_s + mt_w) \log p \qquad (8.9)$$

8.4 多到多播送

多到多播送 (All-to-All Broadcast) 也称为多点播送 (Multinode Broadcast),此时所有 p 个处理器同时启动播送,只是各处理器播送各自的信包。多到多播送的反

操作叫作多点累积 (Multinode Accumulation)，两者在矩阵运算、归约操作和求前缀和中常使用。

8.4.1　使用 SF 进行多到多播送

环　在 p 个处理器的环中进行多到多播送，实际上就是一系列以流水线方式工作的 p 个处理器将各自的信包依次同向传递。传递完毕后，每个处理器都接收到了来自其它 $(p-1)$ 个处理器中的信包，其总的通信时间为：

$$t_{\text{all\_to\_all}}(\text{SF}) = (t_s + mt_w)(p-1) \tag{8.10}$$

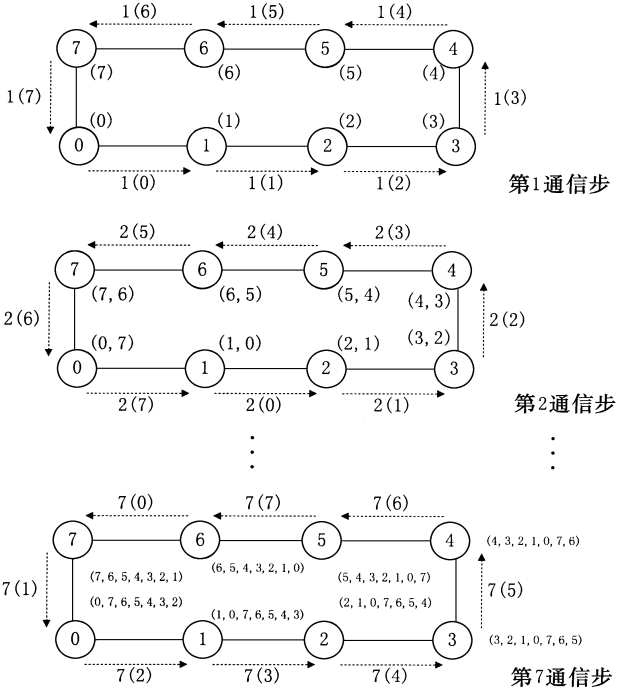


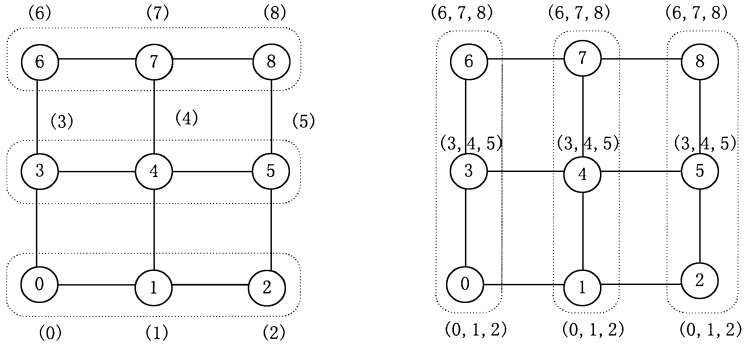
图 8.10　8 个处理器的环上用 SF 方式多到多播送过程

图 8.10 示出了 8 个处理器环上用 SF 方式施行多到多播送的过程。其中，虚线箭头上圆括弧外的数字是时间步，圆括弧内的数字是该时间步所发送的信包所属的处理器号；各节点旁的圆括弧中的数字是该节点所积累的信包所属的处理器号。

环绕网孔 如一到多播送一样,二维网孔上多到多播送可以处理成行环和列环的播送。为此,先按行进行多到多播送;再按列进行多到多播送。在 $p \times p$ 的环绕网孔上,各行多到多的播送时间为 $(t_s + mt_w)(p-1)$;经过行播送后各处理器中的信包大小变为 mp,所以再按列施行多到多播送时其时间为 $(t_s + mpt_w)(p-1)$。因此总的通信时间为:

$$t_{all\_to\_all}(SF) = 2t_s(p-1) + mt_w(p^2-1) \tag{8.11}$$

图 8.11 示出了 3×3 环绕网孔(为了清楚起见,环绕线未画出)上用 SF 方式施行多到多播送的过程。



(a) 起始数据分布　　(b) 行播送后的数据分布

图 8.11 3×3 环绕网孔上用 SF 方式进行多到多播送

超立方 在超立方上施行多到多播送,很自然地是按超立方的维成对处理器交换数据(双向通道),各处理器中的信包大小每次加倍,第 i 步交换时信包大小为 $2^{i-1}m$,因此成对处理器交换信包的时间为 $t_s + 2^{i-1}mt_w$,于是总的通信时间为:

$$\begin{aligned} t_{all\_to\_all}(SF) &= \sum_{i=1}^{\log p} (t_s + 2^{i-1}mt_w) \\ &= t_s \log p + mt_w(p-1) \end{aligned} \tag{8.12}$$

图 8.12 示出了 8 个处理器的超立方上用 SF 方式施行多到多的播送过程。

8.4.2 使用 CT 进行多到多播送

如前所述,使用 CT 方式进行一到多播送时,为了在环和网孔上获得较好的结果,只要将超立方上的算法映射到它们之上即可。但对于多到多播送而言,这样作

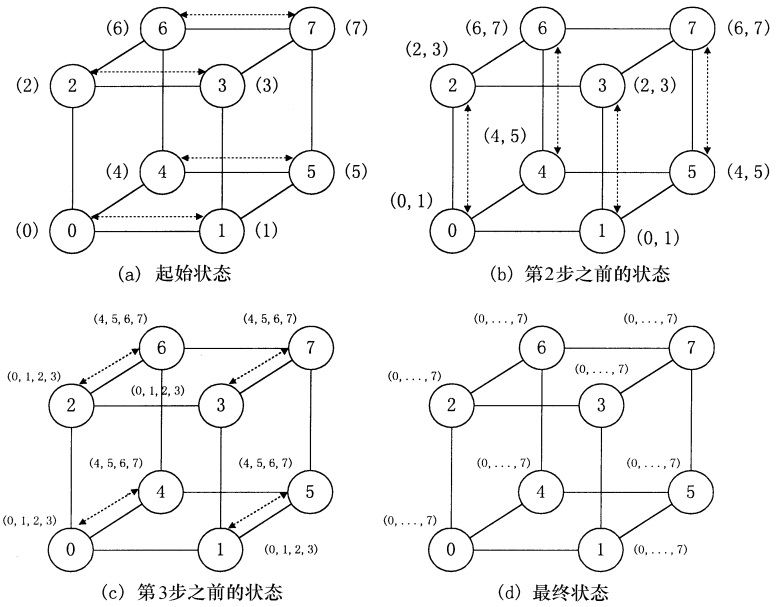


图 8.12　8个处理器的超立方上用 SF 方式进行多到多播送

未必好,因为它会造成通道拥挤。例如,要是将图 8.12 (c) 超立方上成对处理器间的交换操作直接映射到图 8.13 所示的环上时,就会发现有 4 条信包同时穿过环的一条通道,从而造成拥挤。

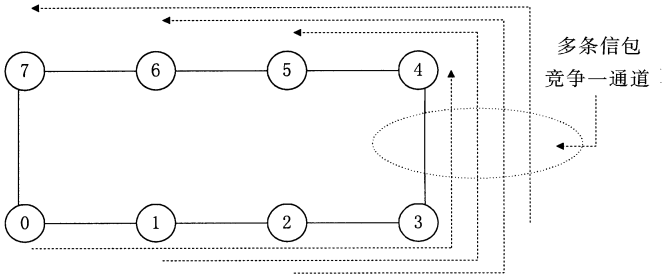


图 8.13　图 8.12 (c) 通信步映射到环上时所造成的拥挤

然而,在双向线性阵列(无环绕网孔)上,使用 CT 方式进行多到多播送的通信时间与在环(带环绕网孔)上使用 SF 方式进行多到多播送的通信时间是一样的。

值得提出的是,对于任何并行结构,其多到多播送之通信时间的下界为 $mt_w(p-1)$,这可从(8.10)、(8.11)和(8.12)式中看出。事实上,不管使用何种结构和选路方式,每个处理器至少要接收到 $m(p-1)$ 个数据字。

8.5 小结和导读

小结 在正文中已经讨论了一到多播送和多到多播送以及它们的反操作单点累积和多点累积。在章末习题 8.6 和 8.7 也将介绍一到多个人通信(One to All Personalized Communication)和多到多个人通信(All to All Personalized Communication)。为了更清楚地理解它们的含意,图 8.14 分别图示了它们的概念。

表 8.1 列出了本章所讨论的不同互连结构采用 CT 开关技术时的几种基本通信操作的通信时间,其中循环 q 移位(Circular q-Shift)(见习题 8.8)通信时间表达式中的 $r(q)$ 是 q 能被 2^j 整除的最大整数 j。

表 8.1 通信时间汇总一览表(使用 CT 方式)

通信操作	p 环	$p\times p$ 二维环绕网孔	p 超立方
一到多播送	$(t_s+mt_w)\log p+t_h(p-1)$	$(t_s+mt_w)\log p+2t_h(\sqrt{p}-1)$	$(t_s+mt_w)\log p$
多到多播送	$(t_s+mt_w)(p-1)$	$2t_s(\sqrt{p}-1)+mt_w(p-1)$	$t_s\log p+mt_w(p-1)$
一到多个人通信	$(t_s+mt_w)(p-1)$	$2t_s(\sqrt{p}-1)+mt_w(p-1)$	$t_s\log p+mt_w(p-1)$
多到多个人通信	$(t_s+mt_w p/2)(p-1)$	$(2t_s+mt_w\sqrt{p})(\sqrt{p}-1)$	$(t_s+mt_w)(p-1)+\frac{t_h}{2}p\log p$
循环 q 移位	$(t_s+mt_w)\lceil p/2\rceil$	$(t_s+mt_w)(2\lceil\frac{\sqrt{p}}{2}\rceil+1)$	$t_s+mt_w+t_h(\log p-r(q))$

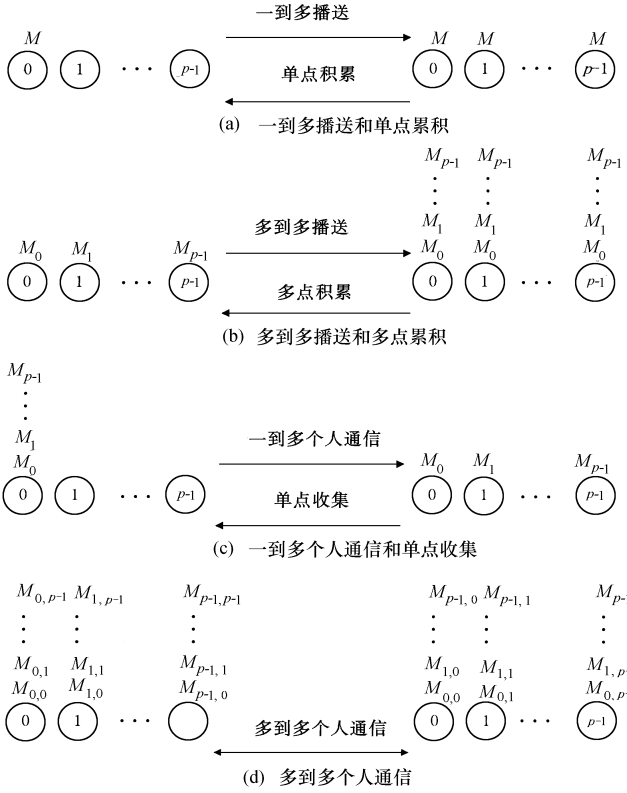


图 8.14 一到多和多到多个人通信

上表中的表达式对使用 SF 开关技术时绝大部分都相同,不同的是:①对于一到多播送,在环上的通信时间为 $(t_s + mt_w)\lceil p/2 \rceil$,在二维网孔上的通信时间为 $2(t_s + mt_w)\lceil p/2 \rceil$;②对于多到多个人通信,在超立方上的通信时间为 $(t_s + mt_w p/2)\log p$。

本章所讨论的通信操作方法均可在 SIMD 机器上执行。然而由于在 MIMD 机器上施行同步比较复杂,所以本章给出的通信时间表达式运用在 MIMD 机器上时可能相差很大。

导读 关于多处理机的互连网络,读者可进一步参考互连网络综述[64]和互连网络专著[202]。有关通信选路策略和消息传递机制,读者可阅读[100]书中的第 7.4 节。[114]书中的第三章还详尽地讨论了诸如一到多个人通信、多到多个人通信和循环移位等其它一些基本通信操作。读者还可以从上述基本参考文献中追

踪阅读一些更丰富和更新近的文献。

习　题

8.1　对于一个 $8=2^d$ 个处理器（其编号从 0 到 2^d-1）的环，如何将其嵌入 d 维超立方中？

［提示：可以使用 8 位 Gray 码对环上的处理器进行编号。一个 8 位 Gray 码为 000,001,011,010,110,111,101,100。］

8.2　对于一个 2×4 的网孔（处理器按行主方式依次编号为 0,1,2,3,4,5,6,7），如何将其嵌入 3 维超立方中？

［提示：将 2×4 的网孔使用 Gray 码按行主对其编号。］

8.3　如图 8.15 所示，信包中的片 0,1,2,3 要分别去向目的地 A,B,C,D。此时片 0 占据信道 CB，片 1 占据信道 DC，片 2 占据信道 AD，片 3 占信道 BA。试问：

①这将会产生什么现象？

②如果采用 X_Y 选路策略，可避免上述现象吗？为什么？

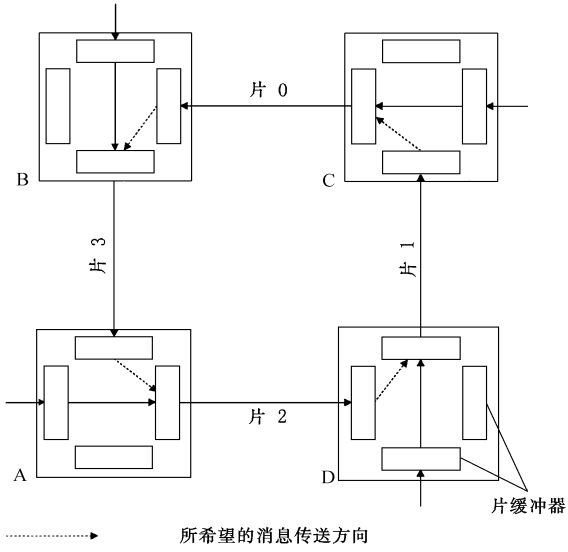


图 8.15　虫蚀选路网络中所出现的现象

8.4　假定在二叉树中，叶节点为处理器节点，内节点为开关节点（参照图 8.16）。试证明在 p 个叶节点的二叉树中，进行 m 个字的一到多播送的通信时间为：

$$(t_s + mt_w + t_h(\log p+1))\log p \qquad (8.13)$$

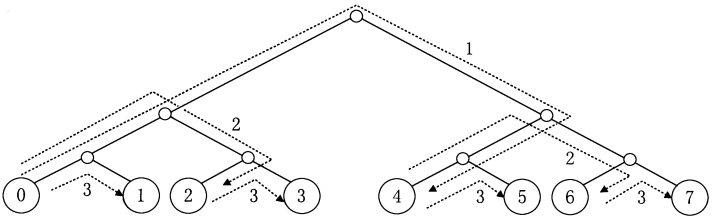


图 8.16　8 个处理器的树上一到多播送过程

[提示：信包穿越 $l-1$ 个开关节点所需的时间为 $t_s+mt_w+t_h(l)$。]

8.5　给定 p 个数 $n_0, n_1, \cdots, n_{p-1}$。所谓前缀和 (Prefix Sum) 就是计算 $S_k=\sum_{i=0}^{k} n_i$，其中 $0\le k\le p-1$。算法 8.3 给出了超立方上求前缀和的方法。试按此算法，计算 8 个处理器的超立方上前缀和。

算法 8.3　d 维超立方上前缀和算法

输入：　p 个数开始存在 p 个处理器中

输出：　第 k 个处理器存有前缀和 $S_k=\sum_{i=0}^{k} n_i, 0\le k\le p-1$

```
Begin
  (1)  result=my_number
  (2)  msg=result
  for  i=0 to d-1 do
       (3.1)  Partner=my_id  2^i
       (3.2)  Send msg to Partner
       (3.3)  Receive number from Partner
       (3.4)  msg=msg+number
       (3.5)  if (Partner<my_id) then result=result+number
              endif
  end for
End
```

8.6　一到多个人通信又称之为单点散播 (Single Node Scatter)，它与一到多播送不同之处是，此时源处理器有 p 个信包，每一个去向一个目的地 (见图 8.14 (i))。图 8.17 示出了 8 个处理器的超立方上单点散射的过程。试证明：使用 SF 和 CT 方式在超立方上施行一到多个人通信的通信时间为：

$$t_{\text{one\_to\_all\_pers}}=t_s\log p+mt_w(p-1) \qquad (8.14)$$

8.7　多到多个人通信又称之为全交换 (Total Exchange)，每个处理器发送各自彼此不同的大小为 m 的信包给其余处理器 (见图 8.14 (j))。图 8.18 示出了 6 个处理器的环上全交换的

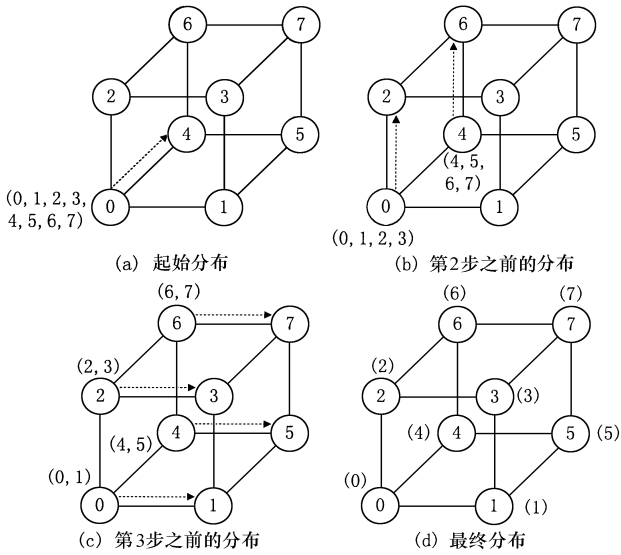


图 8.17　8 个处理器的超立方上单点散射过程

过程，其中，$\{x, y\}$ 表示 i 源处理器，目的处理器 j，$(\{x_1, y_1\}, \{x_2, y_2\}, \cdots, \{x_n, y_n\})$ 表示传输过程中的信包流，每个处理器只接收属于它的信包，下传其余的信包。试证明：利用 SF 方式，在环上施行全交换的通信时间为：

$$t_{\text{total exchange}} = \left(t_s + \frac{1}{2} m t_w p\right)(p-1) \qquad (8.15)$$

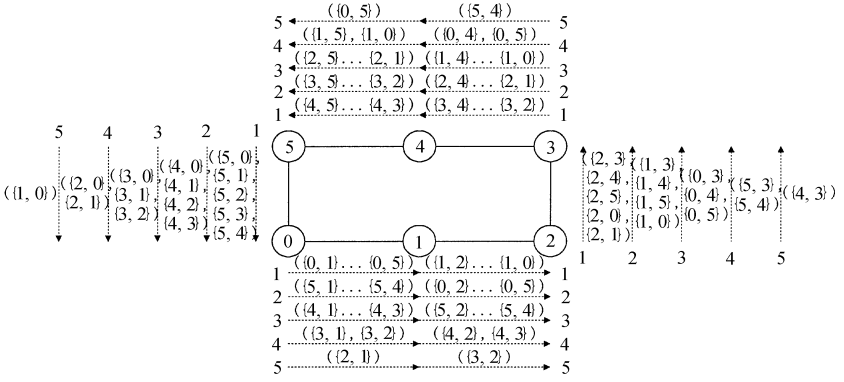


图 8.18　6 个处理器的环上全交换过程

[提示：第 i 步传送的信包大小为 $m(p-1)$。]

8.8　在 p 个处理器，所谓循环 q 移位系指处理器 i 发送包给处理器 $(i+q) \bmod p$。图 8.19 示出了按行主编号的 $p \times p = 4 \times 4$ 环绕网孔上施行 5 移位的过程：首先按行同时循环移位 $(q \bmod p = 1)$ 次；然后作 $q/p = 1$ 次列补偿移位（如图 8.19(b) 所示）；最后再作一次列移位。试证明：利用 SF 方式在正方形环绕二维网孔上施行循环 q 移位的通信时间为：

$$t_{circular\_shift} = (t_s + mt_w)(2\lceil q/p \rceil + 1) \qquad (8.16)$$

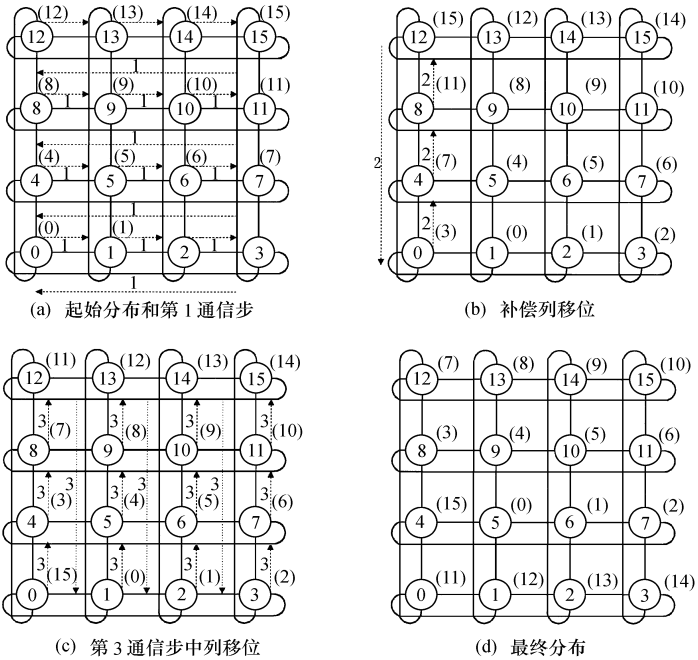


图 8.19　4×4 环绕网孔上 5 移位过程

第九章　稠密矩阵运算

矩阵运算是数值计算中最重要的一类运算，特别是在线性代数和数值分析中，它是一种最基本的运算。本章讨论稠密矩阵的运算，包括矩阵转置、矩阵和向量相乘以及矩阵相乘等。在讨论并行矩阵算法时，将结合具体例子以图示的方法着重算法原理的叙述；在分析算法时，除了分析计算时间外，还要利用上一章所学的知识来分析通信时间；在实现算法时，强调矩阵到处理器的映射；同时为了教学的方便，限于讨论方阵，但所介绍的算法也同样适用于矩形阵。

9.1 矩阵的划分

在科学和工程计算中,矩阵的阶都很高(元素很多),为了并行实现,很自然地要将其进行划分,然后指派给不同的处理器。下面将扼要讨论矩阵的两种常见的划分方法,即行列划分(又称为带状划分)和棋盘划分(又称为块状划分)。

9.1.1 带状划分

所谓带状划分(Striped Partitioning)就是将矩阵整行或整列地分成若干个组,每组指派给一个处理器。也可将若干行或若干列指派给一个处理器,而且这些行和列可以是连续的,也可以是等距相间的,前者称为块带状划分(Block-Striped Partitioning),后者称为循环带状划分(Cyclic-Striped Partitioning)。

块带状划分 如图9.1(a)所示,它是一种列块带状划分。在此情况下,对于一个 $n\times n$ 的矩阵和 p 个处理器(编号为 $P_0,P_1,\cdots,P_{p-1}$)而言,每个处理器将均匀连续地分配有 n/p 列,其中 P_i 包含有列 $(n/p)\ i,(n/p)\ i+1,\cdots,(n/p)\ (i+1)-1$,$0\leq i\leq p-1$。

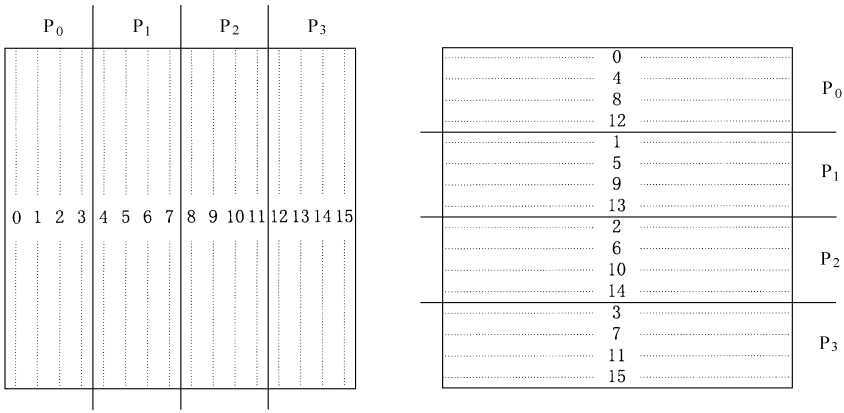
循环带状划分 如图9.1(b)所示,它是一种行循环带状划分。在此情况下,对于一个 $n\times n$ 的矩阵和 p 个处理器(编号为 $P_0,P_1,\cdots,P_{p-1}$)而言,每个处理器将均匀相间地分配有 n/p 行,其中 P_i 包含有行 $i,i+p,i+2p,\cdots,i+(n-1)\ p$,$0\leq i\leq p-1$。

9.1.2 棋盘划分

所谓棋盘划分(Checker Board Partitioning)就是将方阵划分成若干个子方阵,每个子方阵指派给一个处理器,此
分类似,棋盘划分也可分为如图9.2(a)所示的块棋盘划分(Block_Checker Board Partitioning)和如图9.2(b)所示的循环棋盘划分(Cyclic_Checker Board Partition_ing)。

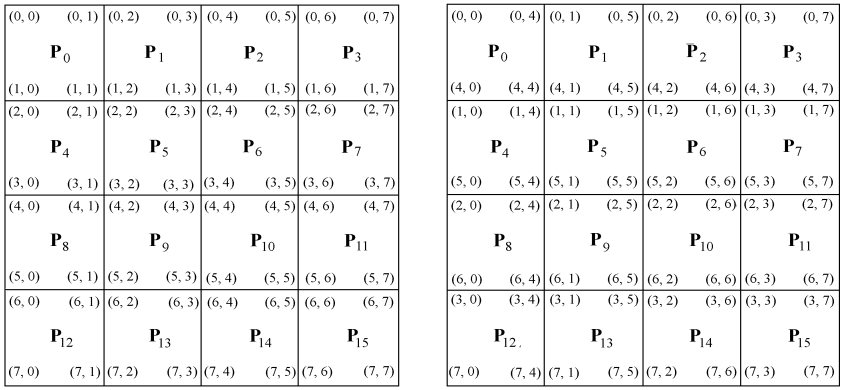
矩阵划分成棋盘状可和处理器连成二维网孔相对应。对于一个 $n\times n$ 的方阵和 $p\times p$ 的二维处理器阵列,每个处理器均匀地分配有 $(n/p)\times(n/p)=n^2/p$ 个矩阵元素。

值得指出的是,和带状划分相比,棋盘划分可开发更高的并行度。例如,对于



(a) 列块带状划分　　(b) 行循环带状划分

图 9.1　4个处理器上16×16矩阵带状划分



(a) 块棋盘划分　　(b) 循环棋盘划分

图 9.2　16个处理器上8×8矩阵棋盘划分

一个 $n\times n$ 的方阵，棋盘划分最多可使用 n^2 个处理器进行并行计算，但使用带状划分可用的处理器不能多于 n 个。

9.2 矩阵转置

对于一个 $n \times n$ 的方阵 A，将其元素 a_{ij}（沿主对角线）与 a_{ji} 互换，就构成了转置矩阵（Transposing Matrix）A^T。如果假定成对元素交换取单位时间，则显然串行矩阵转置算法 9.1 的运行时间为 $(n^2 - n)/2 = O(n^2)$。

算法 9.1 单处理机上矩阵转置算法
输入：$A_{n \times n}$
输出：$A^T_{n \times n}$
Begin
  for $i = 2$ to n do
    for $j = 1$ to $i - 1$ do
      $a_{ij} \leftrightarrow a_{ji}$
    endfor
  endfor
End

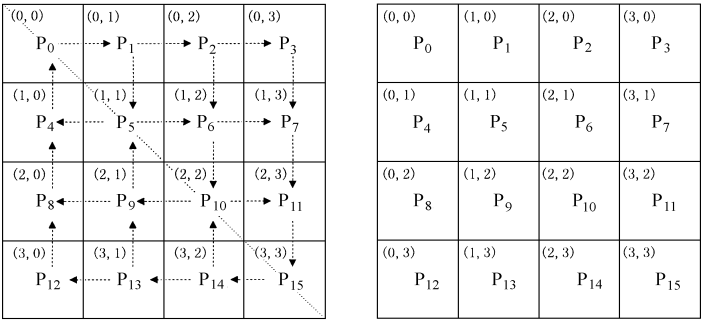
9.2.1 棋盘划分的矩阵转置

以下仅讨论网孔和超立方上块棋盘划分的矩阵转置算法，它们对循环棋盘划分也同样适用。

网孔上的矩阵转置　如图 9.3 所示，要是处理器数目 $p = n^2$，则网孔上的矩阵转置算法是非常简单的：对角线下的元素，先向上移至对角线，再向右移至目的地；对角线上的元素，先向下移至对角线，再向左移至目的地。

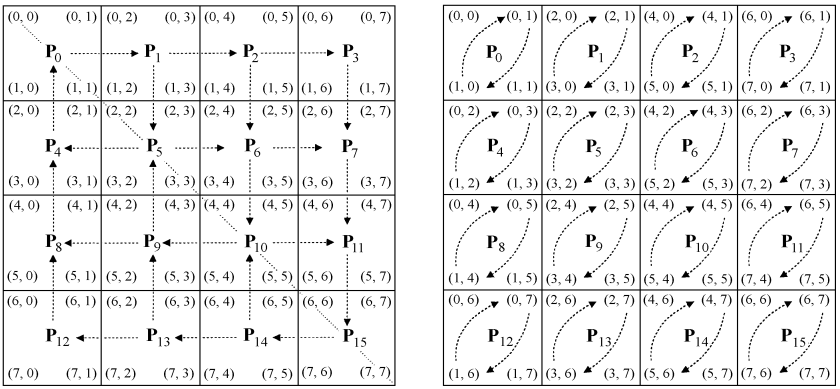
对于 $p \leq n^2$ 的情况，整个 $n \times n$ 的矩阵先分成 p 个大小为 $(n/\sqrt{p}) \times (n/\sqrt{p})$ 的子块，然后算法分两步进行：第一步，将子块视为单个元素进行如图 9.4(a) 所示的子块转置；第二步，在各处理器内施行如图 9.4(b) 所示的局部转置。

整个算法的运行时间可分析如下：第一步的子块转置最长的路径约为 $2\sqrt{p}$，移动单个子块的时间为 $t_s + t_w n^2/p$，所以各子块达到其目的地 $\sqrt{p}$ 处需时 $t_s + t_w n^2/p \sqrt{p}$；第二步，假定成对元素局部交换取单位时间，则大小为 $(n/\sqrt{p}) \times (n/\sqrt{p})$ 子块约花了 $n^2/(2p)$ 的时间进行局部转置，所以算法总运行时间为：



(a) 通信过程　　(b) 最终结果

图 9.3　16 个处理器上棋盘划分的 4×4 矩阵的转置



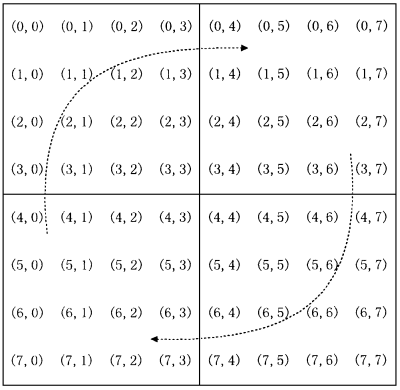
(a) 子块转置　　(b) 子块内局部转置

图 9.4　16 个处理器上棋盘划分的 8×8 矩阵的转置

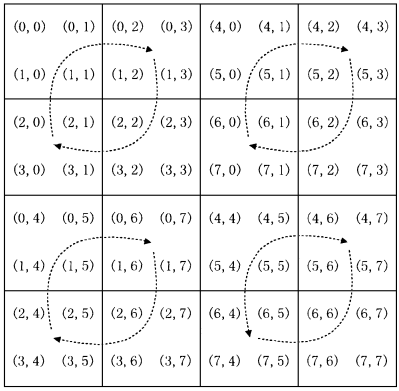
$$T_p = \frac{n^2}{2p} + 2t_s \quad p + 2t_w \frac{n^2}{p} \tag{9.1}$$

递归转置算法 (Recursive Transposition Algorithm)　如图 9.5 所示，一个 $n \times n$ 的方阵先分成 4 个子方阵，对其施行转置，即右上与左下两个子方阵对调；然后每个子方阵再分成 4 个更小的子方阵施行转置；依此类推直到整个矩阵转置完毕。

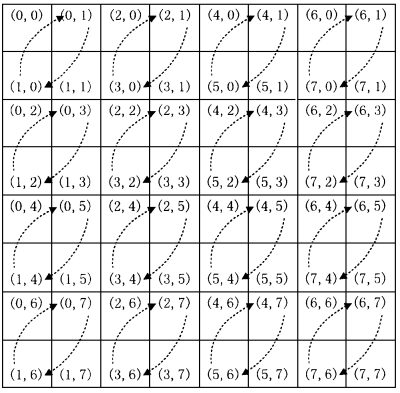
超立方上的矩阵转置　上面所介绍的递归转置算法可以很自然地映射到超立方上。如图 9.6(a) 所示，先将 $n \times n$ 的方阵分成 4 个 $\frac{n}{2} \times \frac{n}{2}$ 的子方阵，相应地将 p



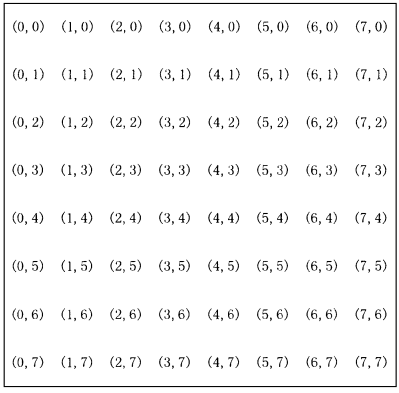
(a) 矩阵分成 4 块进行转置



(b) 每块再分成 4 个子块进行转置



(c) 每个子块中 4 个元素进行转置



(d) 最终结果

图 9.5　8×8 矩阵的递归转置法

个处理器的超立方考虑成 4 个 $p/4$ 个处理器的子超立方，然后将右上子块与左下子块对调（如图 9.6(a) 所示，P_8 中的子块与 P_2 中子块对调，P_9 中子块与 P_3 中子块对调等等）。接着如图 9.6(b) 所示，再将各个 $\frac{n}{2}\times\frac{n}{2}$ 的子块分成 4 个更小的子块，并对它们施行转置。在最后递归步中，子超立方中只有一个处理器，显然总共进行了 $\log p=(\log p)/2$ 递归步，这时各子块的大小为 $(n/\sqrt{p})\times(n/\sqrt{p})$，它们要在每个处理器中施行局部转置，花费了约 $n^2/(2p)$ 的时间。因为每个待通信的子块大小为 n^2/p，所以使用存储转发选路时需 $2(t_s+t_w n^2/p)$ 时间，因此递归 $(\log p)/2$

步的总通信时间为 $(t_s + t_w n^2/p)\log p$,这样算法总的运行时间为:

$$T_p = \frac{n^2}{2p} + (t_s + t_w \frac{n^2}{p})\log p \qquad (9.2)$$

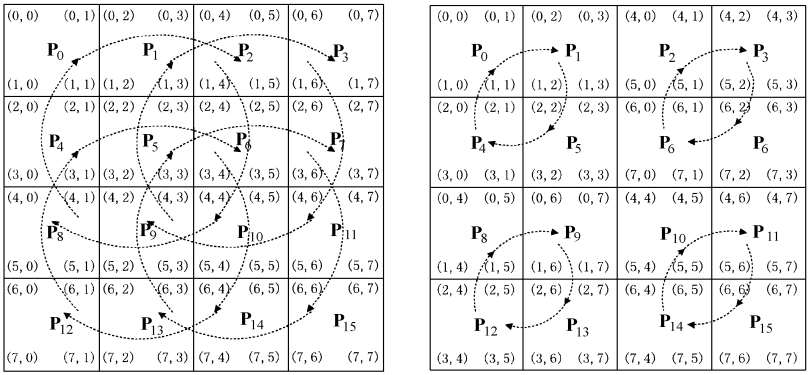


图 9.6 16 个处理器的超立方上的矩阵递归转置

9.2.2 带状划分的矩阵转置

如图 9.7 所示,假定对于一个 $n \times n$ 的矩阵,使用 n 个处理器按行划分。

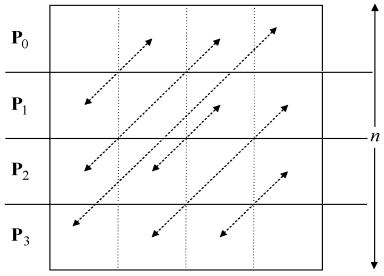


图 9.7 4 个处理器上 4×4 矩阵的转置

开始时,处理器 P_i 中存有矩阵元素 $[i,0]$,$[i,1]$,…,$[i,n-1]$,则在转置后,元素 $[i,0]$ 将属于 P_0 ,元素 $[i,1]$ 将属于 P_1 ,如此等等。注意在转置过程中,每个处理器将发送不同的值给其它处理器,这实际上就是第八章中所讲的多到多个人通信。

一般而论，$p \leq n$ 时每个处理器分配有 n/p 行（共 n^2/p 个元素），此时转置中所涉及到的多到多个人通信的子块大小为 $(n/p) \times (n/p)$。通信完毕后，每个处理器尚须对这些子块施行局部转置。假定成对元素交换取单位时间，则每个子块转置时间为 $n^2/(2p)$。因为每个处理器有 p 个这样的子块，所以转置它们花费了 $n^2/(2p)$ 时间。当在超立方上使用切通选路法对大小为 n^2/p^2 的信包进行多到多个人通信时，其时间约为 $t_s(p-1) + t_w n^2/p + (1/2)\, t_h\, p\log p$（参见第八章的表 8.1）。所以这时算法总的运行时间为：

$$T_p = \frac{n^2}{2p} + t_s(p-1) + t_w \frac{n^2}{p} + \frac{1}{2} t_h\, p\log p \qquad (9.3)$$

9.3 矩阵－向量乘法

对于一个 $n \times n$ 的方阵 A 乘以 $n \times 1$ 的向量 X 就可得 $n \times 1$ 的向量 Y。如果假定一个乘－加运算取单位时间，则显然串行矩阵－向量乘法（Matrix-Vector Multiplication）算法 9.2 的运行时间为 $O(n^2)$。

算法 9.2 单处理机上矩阵－向量乘算法
输入：$A_{n\times n}$，$X_{n\times 1}$
输出：$Y_{n\times 1}$

```
Begin
  for i=0 to n-1 do
      y_i=0
      for j=0 to n-1 do
          y_i = y_i + a_ij x_j
      endfor
  endfor
End
```

9.3.1 带状划分的矩阵－向量乘法

以下仅讨论行带状划分矩阵－向量乘法，列带状矩阵－向量乘法是类似的。$p=n$ 的情况，如图 9.8(a) 所示，开始时处理器 P_i 存放 x_i 和 $a_{i,0}, a_{i,1}, \cdots,$

$a_{i,n-1}$并负责计算 y_i。由于矩阵－向量相乘要求每一行与整个向量相乘，所以如图 9.8(b) 所示每个处理器要将其向量元素向其余的处理器进行多到多播送。播送完毕后就可以如图 9.8(c) 那样进行 $y_i = \sum_{j=0}^{n-1} a_{ij} x_j$ 的计算了。按此，通信和计算时间均为 $O(n)$，它是成本最佳的。

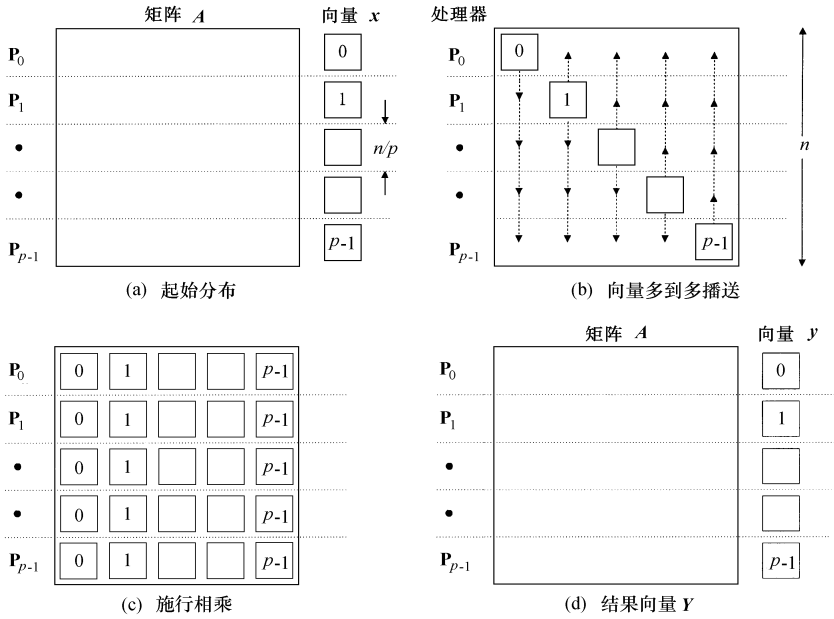


图 9.8 $p = n$ 时按行划分的矩阵－向量乘法

$p < n$ 的情况是类似的，只是开始时每个处理器存放有矩阵的 n/p 行（共 n^2/p 个元素）和向量的 n/p 个元素。

超立方上矩阵乘向量　参照第八章的表 8.1 可知，n/p 个向量元素多到多播送时间为 $t_s \log p + (n/p) t_w (p-1)$，当 p 充分大时它近似为 $t_s \log p + n t_w$。考虑到每个处理器要花 n^2/p 的时间进行乘－加运算，所以超立方上并行矩阵－向量乘之总时间为：

$$T_p = \frac{n^2}{p} + t_s \log p + n t_w \qquad (9.4)$$

当 $p = n$ 时，$T_p = O(n)$，它是成本最佳的。

网孔上矩阵乘向量 参照第八章的表8.1可知多到多播送 n/p 个元素的时间为 $2(p-1)t_s+\frac{n}{p}t_w(p-1)$，它可近似为 $2t_s(p-1)+nt_w$。所以二维环绕 $p\times p$ 网孔上并行矩阵－向量乘之总时间为：

$$T_p=\frac{n^2}{p}+2t_s(p-1)+nt_w \tag{9.5}$$

9.3.2 棋盘划分的矩阵－向量乘法

以下仅讨论块棋盘划分矩阵－向量乘法，循环棋盘划分时的矩阵－向量乘法是类似的。

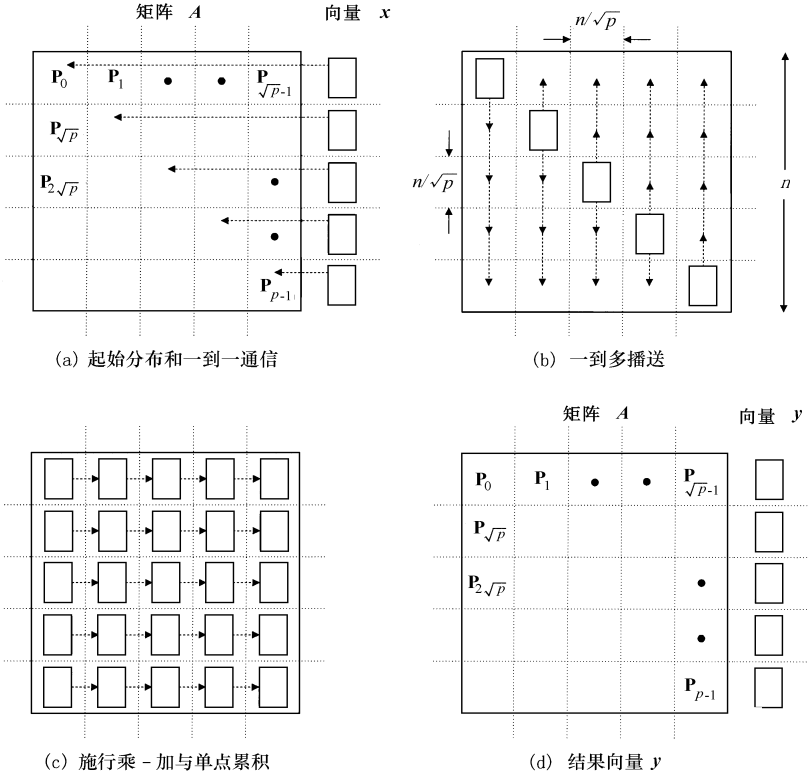
$p=n^2$ 的情况 此时每个处理器存放有矩阵的一个元素，而向量 x_i 通常是存放在 P_{i1} 中的。如果 x_i 是存放在处理器阵列的最后一列中，则进行矩阵－向量乘时，如图9.9(a)所示，先要将向量元素与矩阵主对角线对准；然后如图9.9(b)所示，在列方向上施行向量元素一到多播送；播送完毕后，接着如图9.9(c)和(d)所示，则施行乘－加和单点累积，最后按行收集结果向量 y。

在网孔上，图9.9(a)所示的一到一通信、图9.9(b)所示的一到多播送和图9.9(c)所示的单点累积，时间均为 $O(n)$；在超立方上则为 $O(\log n)$。因为每个处理器执行乘－加操作的时间为常数，所以在 $n\times n$ 的网孔上和 n^2 个处理器的超立方上的并行矩阵－向量乘之总时间分别为 $O(n)$ 和 $O(\log n)$，它们不是成本最佳的。

$p<n^2$ 的情况 假定 p 个处理器排成 $p\times p$ 的二维网孔，每个处理器存放有 $(n/\sqrt{p})\times(n/\sqrt{p})$ 的子块，$n/\sqrt{p}$ 个向量元素存放在处理器阵列的最后一列中，则进行矩阵－向量乘时，首先最右列的每个处理器要发送 $n/\sqrt{p}$ 个向量元素至主对角线上的处理器；然后按列进行一到多播送 $n/\sqrt{p}$ 个元素；接着每个处理器施行 n^2/p 次乘法并积累了 $n/\sqrt{p}$ 个部分和，它们均须按行累加以得到结果向量。

当在无环绕的网孔上用CT选路法执行矩阵－向量乘时：向量与主对角线对准的时间为 $t_s+\frac{n}{p}t_w+t_h\sqrt{p}$；按列施行一到多播送时间为 $(t_s+\frac{n}{p}t_w)\log\sqrt{p}+t_h(\sqrt{p}-1)$；按行单点累积的时间为 $(t_s+\frac{n}{p}t_w)\log\sqrt{p}+t_s(\sqrt{p}-1)$；假定乘－加取单位时间，则计算时间为 n^2/p。所以在网孔上并行算法总运行时间约为：

$$T_p\approx\frac{n^2}{p}+t_s\log p+t_w\frac{n}{p}\log p+3t_h\sqrt{p} \tag{9.6}$$



(a) 起始分布和一到一通信　　(b) 一到多播送

(c) 施行乘－加与单点累积　　(d) 结果向量 y

图 9.9　$p=n^2$ 时棋盘划分的矩阵－向量乘法

类似地，当在二维网孔上用 SF 选路法执行矩阵－向量乘时，并行算法的总运行时间约为：

$$T_p \approx \frac{n^2}{p} + 2t_s + 3mt_w \qquad (9.7)$$

带状与棋盘划分时矩阵－向量乘的比较　比较 (9.5) 式与 (9.6) 式可知，在网孔上用同样多的处理器，棋盘划分的矩阵－向量乘法比带状划分时要快；如果 $p>n$，则无法使用带状划分，而棋盘划分不受此限制；即使 $p \leq n$，棋盘划分也更优；上述结论也适合于超立方 (见习题 9.3)。同时由章末习题 9.5 和 9.4 可知，棋盘划分的矩阵－向量乘的可扩放性也比带状划分时的好。

9.4　矩 阵 乘 法

一个 $n\times n$ 的方阵 A 乘以 $n\times n$ 的方阵 B 就得一个 $n\times n$ 的方阵 C：

$$c_{ij}=\sum_{k=0}^{n-1} a_{ik}\,b_{kj},\ 0\leq i,j\leq n-1 \tag{9.8}$$

矩阵相乘的关键是相乘的两个元素的下标要满足一定的要求（即对准）。为此常采用旋转矩阵元素的办法（如下面将要介绍的 Cannon 乘法）使元素下标对准，或采用适当复制矩阵元素的办法（如下面将要介绍的 DNS算法）使元素下标对准，或采用流水线的办法（如习题 9.10 所示的心动阵列上的算法）使元素的下标对准。

如果假定一个乘－加运算取单位时间，则显然串行矩阵乘法算法 9.3 的运行时间为 $O(n^3)$。

算法 9.3　单处理机上矩阵相乘算法
输入：　$A_{n\times n}$，$B_{n\times n}$
输出：　$C_{n\times n}$

```
Begin
  for i=0 to n-1 do
      for j=0 to n-1 do
          c_ij=0
          for k=0 to n-1 do
              c_ij=c_ij+a_ik·b_kj
          endfor
      endfor
  endfor
End
```

目前所知的串行矩阵乘法的时间复杂度均为 $O(n^x)$，其中 $2<x\leq 3$。例如 Strassen 乘法的时间阶为 $O(n^{2.8074})$。

9.4.1　简单并行分块乘法

分块矩阵乘法（Block Matrix Multiplication）　矩阵运算中分块的概念是非常

有用的,可以将对矩阵元素的代数运算推广到对矩阵的块(子矩阵)进行同样的代数运算。例如,一个 $n\times n$ 的矩阵 A 可以视为一个 $q\times q$ 的方块阵 $A_{i,j}$ $(0\leq i,j\leq q-1)$,每个方块阵大小为 $(n/q)\times(n/q)$。这样算法 9.3 就可改写成算法 9.4,相应地,将矩阵元素的乘-加运算代换为子矩阵的乘-加运算。算法 9.4 共执行了 q^3 次矩阵乘法,每个矩阵乘法需 $(n/q)^3$ 次乘-加运算,所以总的乘-加次数仍为 n^3。

使用 p 个处理器执行分块乘时,可取 $q=\sqrt{p}$,这样每个处理器计算一个 $C_{i,j}$ 块。

算法 9.4 单处理机上分块矩阵相乘算法

输入: $A_{n\times n}$, $B_{n\times n}$,子块大小为 $(n/q)\times(n/q)$

输出: $C_{n\times n}$

```
Begin
  for i=0 to q-1 do
    for j=0 to q-1 do
      C[i,j]=0
      for k=0 to q-1 do
        C[i,j]=C[i,j]+A[i,k]·B[k,j]
      endfor
    endfor
  endfor
End
```

并行分块乘法 简单的并行分块乘法的过程是:①分块:将 $A_{n\times n}$ 与 $B_{n\times n}$ 分成 p 块 $A_{i,j}$ 和 $B_{i,j}$ $(0\leq i,j\leq \sqrt{p}-1)$,每块大小为 $(n/\sqrt{p})\times(n/\sqrt{p})$,并将它们分配给 $\sqrt{p}\times\sqrt{p}$ 个处理器 $(P_{0,0},P_{0,1},\cdots,P_{\sqrt{p}-1,\sqrt{p}-1})$,开始时 $P_{i,j}$ 存放有块 $A_{i,j}$ 与块 $B_{i,j}$,然后计算块 $C_{i,j}$;②通信:为了计算块 $C_{i,j}$,则需要所有块 $A_{i,k}$ 与 $B_{k,j}$ $(0\leq k\leq \sqrt{p}-1)$,为此在每行的处理器之间要进行 A 矩阵块的多到多播送,以得到 $A_{i,k}$,而在每列的处理器之间要进行 B 矩阵块的多到多播送,以得到 $B_{k,j}$;③乘-加计算:一旦 $P_{i,j}$ 收到了 $A_{i,0}$,$A_{i,1}$,$\cdots$,$A_{i,\sqrt{p}-1}$ 与 $B_{0,j}$,$B_{1,j}$,$\cdots$,$B_{\sqrt{p}-1,j}$ 之后,就同时进行乘-加运算。

当并行分块乘法执行在 p 个处理器的超立方上时,参照第八章表 8.1 可知,通信时间为 $2(t_s\log p+t_w\frac{n^2}{p}(\sqrt{p}-1))$;计算时间为 $q\times(\frac{n}{q})^3=\sqrt{p}\times(\frac{n}{\sqrt{p}})^3=$

n^3/p。所以算法的并行运行时间为：

$$T_p = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{p} \tag{9.9}$$

当并行分块乘法执行在 $p \times p$ 的二维环绕网孔上时，参照第八章表 8.1 可知其通信时间为 $2t_s p + 2t_w \frac{n^2}{p}$，计算时间为 n^3/p，所以算法的并行运行时间为：

$$T_p = \frac{n^3}{p} + 2t_s p + 2t_w \frac{n^2}{p} \tag{9.10}$$

简单并行分块乘法的主要缺点是存储要求过大：通信结束时每个处理器拥有 $2p$ 个块，每块大小为 n^2/p，所以每个处理器要求 $O(n^2/p)$ 存储器，p 个处理器共要求 $O(n^2 p)$，它是串行算法存储要求的 p 倍。

9.4.2 Cannon 乘法

算法原理 Cannon 算法（Cannon's Algorithm）是一种存储有效的算法。为了使两矩阵的下标满足相乘的要求，它和上一节的并行分块乘法不同，不是阵列的各行和各列施行多到多播送，而是有目的地在各行和各列施行循环移位，从而使处理器的总存储要求可以降下来。照例，将矩阵 A 和 B 分成 p 个方块 $A_{i,j}$ 和 $B_{i,j}$（$0 \le i,j \le p-1$），每块大小为 $(n/p) \times (n/p)$，并将它们分配给 $p \times p$ 个处理器（$P_{0,0}, P_{0,1}, \cdots, P_{p-1,p-1}$）。开始时处理器 $P_{i,j}$ 存放有块 $A_{i,j}$ 和块 $B_{i,j}$，并负责计算块 $C_{i,j}$，然后算法开始执行：

① 将块 $A_{i,j}$（$0 \le i,j < p$）向左循环移动 i 步；

将块 $B_{i,j}$（$0 \le i,j < p$）向上循环移动 j 步；

② $P_{i,j}$ 执行乘－加运算；

然后，将块 $A_{i,j}$（$0 \le i,j < p$）向左循环移动 1 步；

将块 $B_{i,j}$（$0 \le i,j < p$）向上循环移动 1 步；

③ 重复第②步，在 $P_{i,j}$ 中共执行 p 次乘－加运算和 p 次块 $A_{i,j}$ 和 $B_{i,j}$ 的循环单步移位。

算法举例 图 9.10 示例了在 16 个处理器上，用 Cannon 算法执行 $A_{4\times4} \times B_{4\times4}$ 的过程。其中 (a) 和 (b) 对应于上述算法的第①步；(c)、(d)、(e) 和 (f) 对应于上述算法的第②和第③步。注意在算法第①步时，A 矩阵的第 0 行不移位，第 1 行循环左移 1 位，第 2 行循环左移 2 位，第 3 行循环左移 3 位；类似地，B 矩阵的第

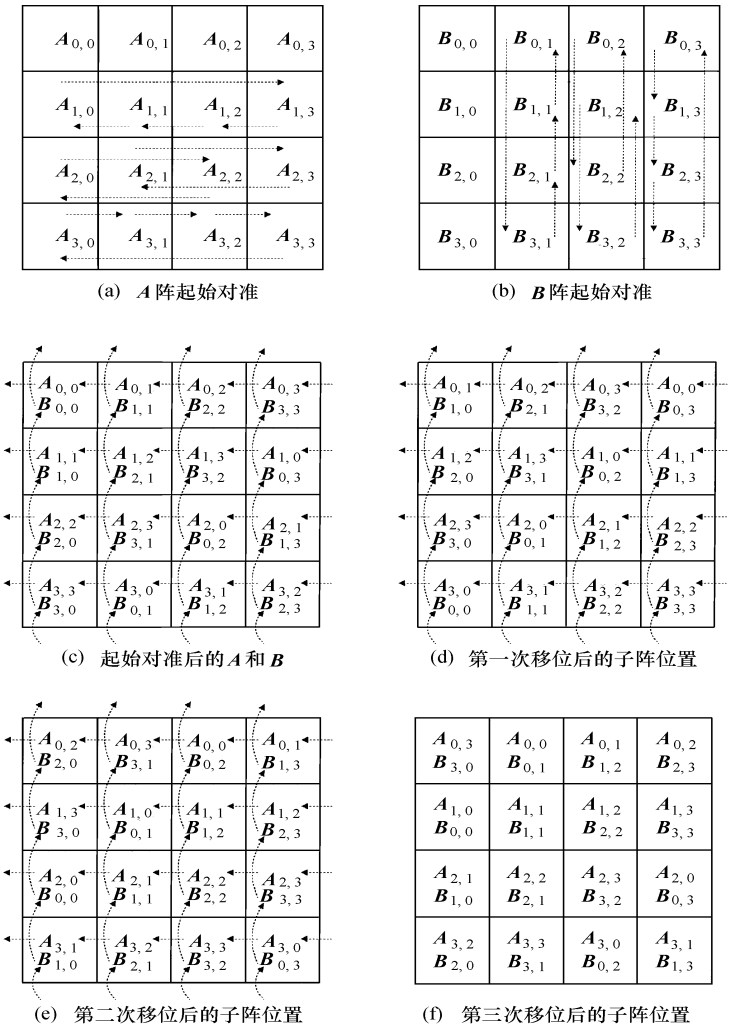


图 9.10 16 处理器上 Cannon 乘法过程

0 列不移位,第 1 列循环上移 1 位,第 2 列循环上移 2 位,第 3 列循环上移 3 位。

算法描述 假定矩阵 A 和 B 分成 p 块 $A_{i,j}$ 与 $B_{i,j}$ $(0 \le i,j \le p-1)$,处理器排成 $p \times p$ 的方阵,符号"a "表示移位,则 Cannon 算法可形式描述如下:

算法 9.5 Cannon 分块乘法算法

输入： $A_{n\times n}$, $B_{n\times n}$

输出： $C_{n\times n}$

Begin

(1) for $k=0$ to $p-1$ do

for all $P_{i,j}$ par-do

(i) if $i>k$ then $A_{i,j}$ a $A_{i,(j+1) \bmod p}$ endif

(ii) if $j>k$ then $B_{i,j}$ a $B_{(i+1) \bmod p,j}$ endif

endfor

endfor

(2) for all $P_{i,j}$ par-do $C_{i,j}=0$ endfor

(3) for $k=0$ to $p-1$ do

for all $P_{i,j}$ par-do

(i) $C_{i,j}=C_{i,j}+A_{i,j}\quad B_{i,j}$

(ii) $A_{i,j}$ a $A_{i,(j+1) \bmod p}$

(iii) $B_{i,j}$ a $B_{(i+1) \bmod p,j}$

endfor

endfor

End

算法分析 当算法执行在 p 个处理器的超立方上时，若使用 CT 选路法，算法第 (1) 步的循环移位时间 (参照第八章表 8.1) 为 $2(t_s+t_w\frac{n^2}{p}+t_s\log p)$，算法第 (3) 步的单步移位时间为 $2(t_s+t_w\frac{n^2}{p})$ p，当 p 充分大时后者为主项。算法第 (3) 步共执行 p 次 $(n/p)\times(n/p)$ 子块乘法，其时间为 $p\times(n/p)^3=n^3/p$。所以在超立方上 Cannon 乘法的并行运行时间为：

$$T_p=\frac{n^3}{p}+2\quad p\,t_s+2t_w\frac{n^2}{p} \tag{9.11}$$

当算法执行在 $p\times$ p 的二维网孔上时，算法第 (1) 步的循环移位时间为 $2(t_s$

$+t_w\frac{n^2}{p}$) p,算法第(3)步的单步移位时间为 $2(t_s+t_w\frac{n^2}{p})$ p。算法第(3)步的运算时间为 n^3/p。所以在二维网孔上 Cannon 乘法的并行运行时间为:

$$T_p=\frac{n^3}{p}+4\ p\ t_s+4t_w\frac{n^2}{p} \tag{9.12}$$

9.4.3 Fox 乘法

算法原理 Fox 算法(Fox's Algorithm)是另一种存储有效的算法,它和并行分块乘法及 Cannon 乘法均不同,执行 Fox 乘法时,行处理器施行一到多播送,而列处理器施行循环单步上移,从而使处理器的存储要求可以降下来。照例,将矩阵 A 和 B 分成 p 个方块 $A_{i,j}$ 和 $B_{i,j}$ $(0\leq i,j\leq p-1)$,每块大小为 $(n/\sqrt{p})\times(n/\sqrt{p})$,并将它们分配给 $\sqrt{p}\times\sqrt{p}$ 个处理器 $(P_{0,0},P_{0,1},\cdots,P_{\sqrt{p}-1,\sqrt{p}-1})$。开始时处理器 $P_{i,j}$ 存放有块 $A_{i,j}$ 和块 $B_{i,j}$,并负责计算块 $C_{i,j}$。假定首先选定对角块 $A_{i,i}$ 向同行的 $p-1$ 个处理器进行一到多播送,则 Fox 算法执行以下各步 $\sqrt{p}$ 次迭代即可完成:

① 所选中的对角块 $A_{i,i}$ 向所在行的 $p-1$ 个处理器进行一到多播送;
② 各处理器将所收到的 A 阵的块与 B 阵原有的块进行乘-加运算;
③ B 阵的块向上循环 1 步;
④ 如果 $A_{i,j}$ 是本次播送的块,则下一次应选块 $A_{i,(j+1)\bmod\sqrt{p}}$ 向同行的 $p-1$ 个处理器播送,然后转第②步。

算法举例 图 9.11 示例了在 16 个处理器上,用 Fox 算法执行 $A_{4\times4}\times B_{4\times4}$ 时的块 A 的播送与块 B 的单步循环上移的过程。

算法分析 当算法执行在 p 个处理器的超立方上时,若使用 CT 选路法,块 A 的总的一到多播送时间(参照第八章表 8.1)为 $\frac{1}{2}(t_s\ p+t_w\frac{n^2}{p})\log p$,而块 B 的总的循环单步上移的时间比起它来不是主要的;照例 $P_{i,j}$ 执行运算的时间为 n^3/p。所以在超立方上 Fox 乘法的并行运行时间为:

$$T_p=\frac{n^3}{p}+\frac{1}{2}t_s\ p\log p+\frac{1}{2}t_w\frac{n^2}{p}\log p \tag{9.13}$$

9.4.4 DNS 乘法

算法原理 如前所述,使用棋盘划分最多可用 n^2 个处理器,算法的加速可达

$O(n^2)$。下面介绍一种使用更多的处理器可得到更快的算法，即 DNS 算法，从而加速比也更高。

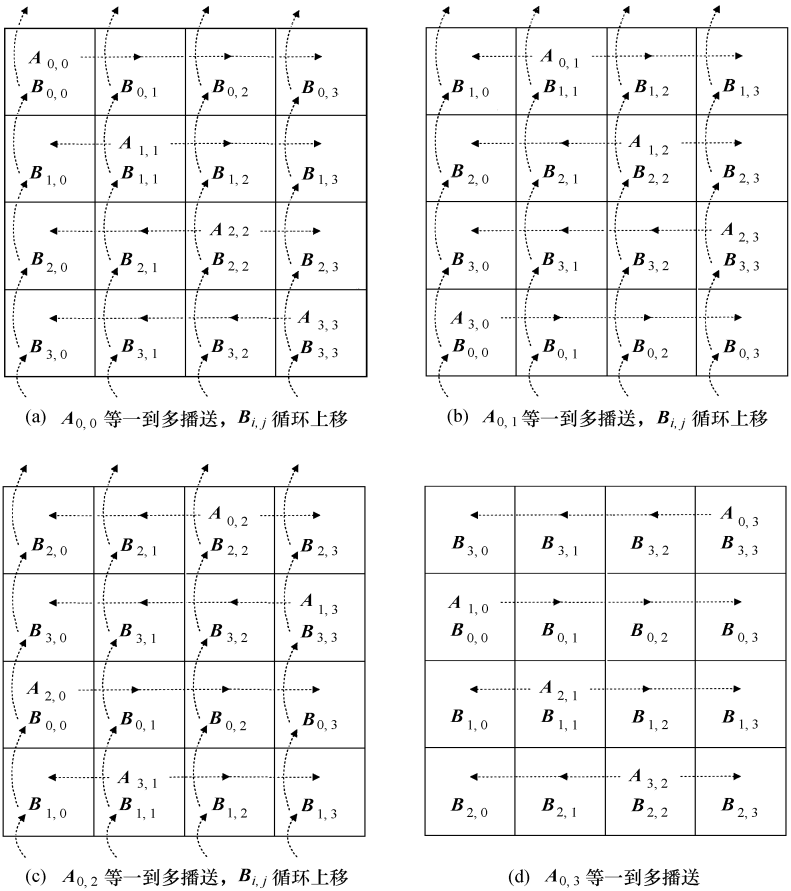


图 9.11　16 个处理器上 Fox 乘法通信过程

DNS 是三位学者 Dekel、Nassimi、Sahni 名字的首字母，他们的算法是运行在超立方连接的 SIMD 机器上的。

在 $A_{n\times n}\times B_{n\times n}$ 的运算中，$a_{is}\times b_{sj}$ 的操作共有 n^3 个。如果对矩阵 A 和 B 的元素进行适当的复制，则有可能利用 n^3 个处理器同时执行 n^3 个乘法运算，然后再作 $\log n$ 次求和运算即可得到最终的乘积元素 c_{ij}。为此将处理器数目 $p=n^3$ 表

示成 $n\times n\times n$ 的三维数组，其中 $n=2^q$。这样超立方中的处理器 P_r 将处于位置 (i,j,k)，其中 $r=in^2+jn+k$，且 $0\le i,j,k\le n-1$。令 r 的二进制表示为 $r_{3q-1},\cdots,r_{2q}r_{2q-1},\cdots,r_qr_{q-1},\cdots,r_0$，其中 r_b 表示从 r_0 数起的 r 的第 b 位。显然下标 i,j,k 分别对应着 $r_{3q-1},\cdots,r_{2q}$，$r_{2q-1},\cdots,r_q$，$r_{q-1},\cdots,r_0$。

假定每个处理器 P_r 有三个寄存器 A_r、B_r、C_r。开始时处于位置 $(0,j,k)$ 的处理器 P_s 其 A_s 和 B_s 中的内容为 a_{jk} 和 b_{jk}，所有其它处理器的寄存器内容均置为 0，算法结束时，$c_{jk}=\sum_{i=0}^{n-1}a_{ji}\times b_{ik}$。算法可分为三步：

① 数据复制：$A_{n\times n}$ 与 $B_{n\times n}$ 的元素加载到 $P_{0,j,k}$ 中的 A 和 B 寄存器中，通过 A、B 同时在 i 维复制，A 在 k 维复制和 B 在 j 维复制后，矩阵 A 和 B 元素就被分配到 n^3 个处理器中了；

② 施行相乘：各处理器将 A 寄存器之内容与 B 寄存器之内容两两相乘；

③ 求和运算：执行 $\sum_{i=0}^{n-1}C_{i,j,k}$。

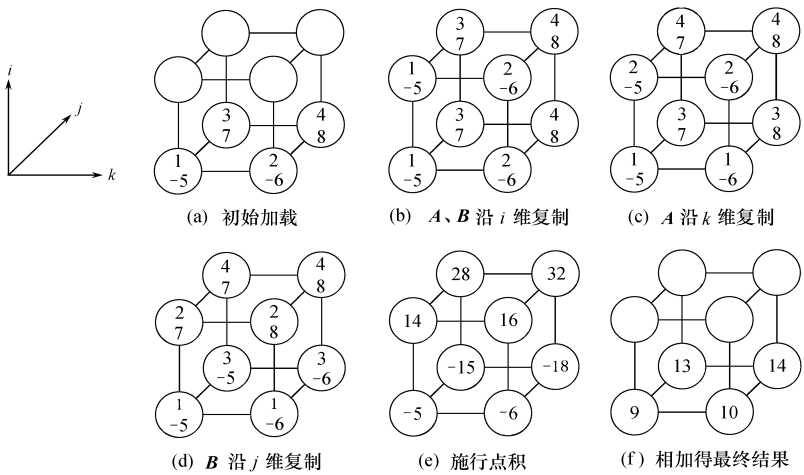


图 9.12 DNS 乘法的执行过程

算法举例 假定 $A=\begin{matrix}1&2\\3&4\end{matrix}$，$B=\begin{matrix}-5&-6\\7&8\end{matrix}$，图 9.12 示例了 DNS 乘法的过程：图 9.12(a) 是初始加载；图 9.12(b) 为 A 和 B 寄存器同时按 i 维进行复制，即 $r_2=0$ 的那些处理器中 A、B 寄存器之内容复制到 $r_2=1$ 的那些处理器的相应寄

存器中(图 9.12(c))为 A 寄存器按 k 维进行复制,即凡是 $r_0 = r_2$ 的那些处理器中 A 寄存器之内容复制到 $r^{(0)}$ 的那些处理器的 A 寄存器中,其中 $r^{(0)}$ 表示 r 的第 0 位取反(图 9.12(d))为 B 寄存器按 j 维进行复制,即凡是 $r_1 = r_2$ 的那些处理器中 B 寄存器之内容复制到 $r^{(1)}$ 的那些处理器的 B 寄存器中,其中 $r^{(1)}$ 表示 r 的第 1 位取反(图 9.12(e))为 A 和 B 寄存器中的元素之乘积(图 9.12(f))为最终之结果。

算法描述 令 $r^{(m)}$ 表示 r 的第 m 位取反;$\{p, r_m = d\}$ 表示整数 $r(0 \le r \le p-1)$ 的集合,r 的二进制表示为 $r_{3q-1} \cdots r_{m+1} d r_{m-1} \cdots r_0$。

算法 9.6 DNS 乘法算法

输入: $A_{n\times n}, B_{n\times n}, n=2^q$

输出: $C_{n\times n}$

```
Begin
(1) for m=3q-1 to 2q do/*按 i 维复制 A, B*/
      for all r in {p, r_m=0} par-do
        (1.1) A_r^(m) ? A_r
        (1.2) B_r^(m) ? B_r
      endfor
    endfor
(2) for m=q-1 to 0 do/*按 k 维复制 A*/
      for all r in {p, r_m = r_{2q+m}} par-do
        A_r^(m) ? A_r
      endfor
    endfor
(3) for m=2q-1 to q do/*按 j 维复制 B*/
      for all r in {p, r_m = r_{q+m}} par-do
        B_r^(m) ? B_r
      endfor
    endfor
(4) for r=0 to p-1 par-do/*相乘*/
        C_r = A_r × B_r
    endfor
(5) for m=2q to 3q-1 do/*求和*/
```

```
        for r=0 to p-1 par- do
            C_r = C_r + C_r(m)
            endfor
    endfor
End
```

算法分析 由算法 9.6 可知,使用 n^3 个处理器的 DNS 乘法,其并行运行时间为 $O(\log n)$,它不是成本最佳的。

算法讨论 在图 9.12 中,是以 $A_{2\times 2}\times B_{2\times 2}$ 为例的,如果矩阵的阶超过 2(例如为 4),则这种表示就不方便了,但可以采用图 9.13 所示的方法进行 DNS 相乘。图 9.13(a)是 A 和 B 的起始加载,此时 A 矩阵的行按 j 维分布;B 矩阵的列按 i 维分布。复制时,首先 A 和 B 同时沿 k 维进行一到一播送,其中如图 9.13(b),$A_{i,j}$ 从 $P_{i,j,0}$ 传到 $P_{i,j,j}$(类似地 $B_{i,j}$ 从 $P_{i,j,0}$ 传到 $P_{i,j,i}$);然后 A 在不同的 k 平面内将其列沿 j 维同时进行一到多播送,其结果如图 9.13(c)所示。类似地,B 在不同的 k 平面内将其行沿 i 维同时进行一到多播送,其结果如图 9.13(d)所示。复制完毕后,$A_{i,k}$ 与 $B_{k,j}$ 就可以在 $P_{i,j,k}$ 中相乘了;然后每个乘积元素 $C_{i,j}$ 再沿 k 维进行单点累积。在此过程中,处理器 $P_{i,j,0}$ 就收集了来自处理器 $P_{i,j,1}$,…,$P_{i,j,n-1}$ 的乘积。为了清楚起见,图 9.13(c)和(d)只示出了 $C_{0,0}$ 的计算。

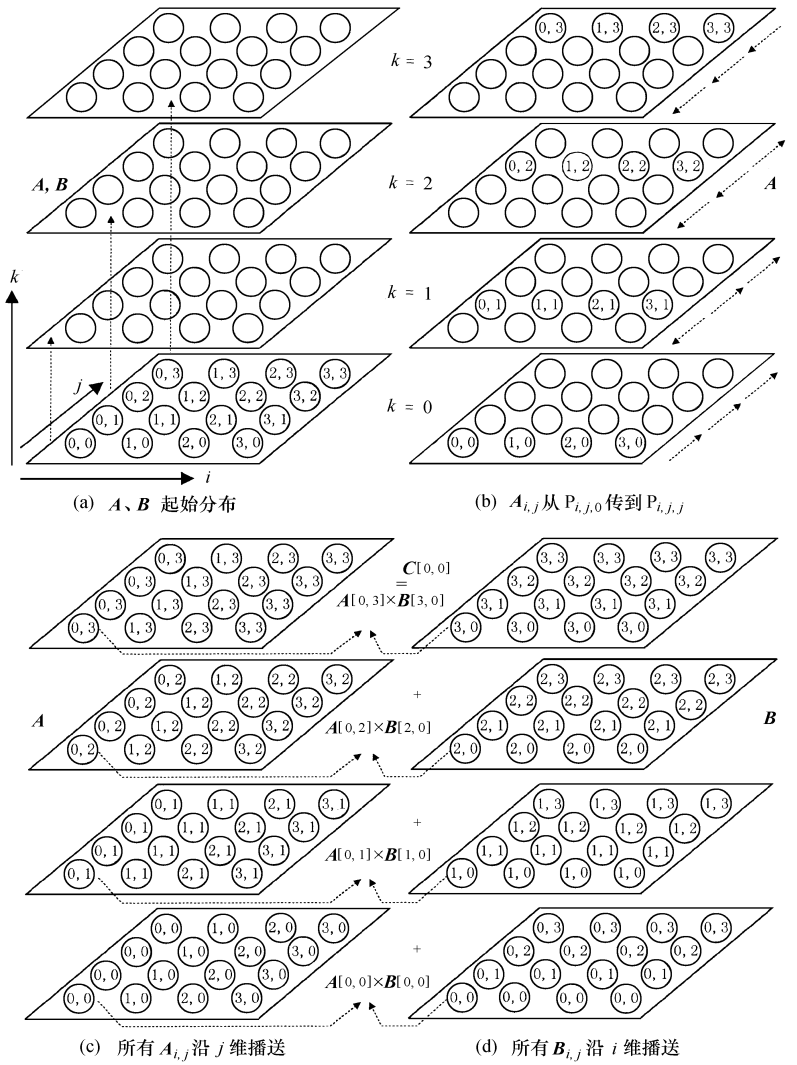
当处理器的数目 $p<n^3$ 时,为了执行 DNS 乘法,可以假定 $p=q^3$,而 $q<n$。此时原 $n\times n$ 的矩阵可以划分成大小为 $(n/q)\times(n/q)$ 的一些块,每个矩阵可以视为 $q\times q$ 的分块矩阵。这样在 q^3 个处理器上执行 DNS 算法与在 n^3 个处理器上执行 DNS 算法类似,只是前者是块运算后者是单独元素运算。

当分块的 DNS 算法执行在 $p<n^3$ 个处理器的超立方上时,若使用 CT 选路法,算法执行一到一传送的时间为 $t_s+t_w(n/q)^2+t_h\log q$(见第八章 8.2 节);算法执行一到多播送的时间为 $(t_s+t_w(n/q)^2)\log q$(见第八章表 8.1);算法执行单点收集的时间为 $t_s\log q+t_w(n/q)^2\log q$。假定乘-加操作取单位时间,则 $(n/q)\times(n/q)$ 块阵乘法的时间为 $(n/q)^3$。略去一到一传送时间和收集过程中的加法时间,则在超立方上使用 CT 选路法的 DNS 乘法之并行运行时间为:

$$T_p\approx\left(\frac{n^3}{q}\right)+3t_s\log q+3t_w\left(\frac{n}{q}\right)^2\log q$$

因为 $q=p^{1/3}$,所以

$$T_p=\frac{n^3}{p}+t_s\log p+t_w\frac{n^2}{p^{2/3}}\log p \tag{9.14}$$



(a) $\boldsymbol{A}$、$\boldsymbol{B}$ 起始分布　　(b) $\boldsymbol{A}_{i,j}$ 从 $P_{i,j,0}$ 传到 $P_{i,j,j}$

(c) 所有 $\boldsymbol{A}_{i,j}$ 沿 j 维播送　　(d) 所有 $\boldsymbol{B}_{i,j}$ 沿 i 维播送

图 9.13　4×4 的矩阵在 64 个处理器上的 DNS 乘法

9.5　小结和导读

小结　本章所研究的矩阵都是稠密阵(Dense Matrices),与其相对是稀疏阵(Sparse Matrices)。稠密阵几乎没有零元素,而稀疏阵其绝大多数元素均为零。之所以要区分出稀疏阵,是因为可以利用非零元素的号码与位置来减少运算时间。本章着重讨论了稠密阵的最基本的运算,包括矩阵转置、矩阵乘向量和矩阵乘法等运算。这些算法在执行过程中几乎用到了第八章所讲的所有基本通信操作,所以在分析算法的通信时间时,读者要不断参照第八章的有关内容。

从处理器的利用能力等来看,棋盘划分比带状划分要好,这一点从第9.3节"矩阵－向量乘法"的讨论可以得到证实;简单的并行分块乘法虽然运行时间较快,但对存储要求过大;从存储要求来看,Cannon乘法和Fox乘法优于简单并行分块乘法,特别是Cannon乘法综合性能较优,所以使用得较普遍;从加速能力来看,DNS乘法的加速比较大,特别是它经常被别的算法作为基本算法加以调用,所以使用较广。

除了本章所讨论的矩阵运算外,矩阵的分解、矩阵的求逆和矩阵求特征值等也是很常见的矩阵运算,但限于篇幅,就不再讨论了。

导读　关于并行与分布式数值算法,[31]是一本很好的参考书。有关矩阵运算,[78]是一本很经典的教本(此书的第一版有中译本:廉庆荣等译.矩阵计算.大连:大连理工大学出版社,1988)。本章所讨论的Cannon乘法、Fox乘法和DNS乘法,原始论文分别来源于[40]、[68]和[58]。这些算法的改进版本可参考[30]和[94]。有关并行矩阵算法的可扩放性分析,读者可阅读[34]。在[113,114]书中的第五章的参考文献注释中还列举了大量有关稠密矩阵的算法,读者可追踪进一步阅读。

习　题

9.1　根据9.2.1节所讨论的矩阵转置算法:

①试写出二维网孔上递归矩阵转置算法的形式描述;

②试写出超立方上递归矩阵转置算法的形式描述。

9.2　根据9.3.2节所讨论的矩阵－向量乘法,试证明:在p个处理器的超立方上,用SF选路方法进行矩阵－向量乘法,其并行运行时间约为$n^2/p+t_s\log p+(3/2)t_w(n/p)\log p$。

9.3　试分析在超立方上执行矩阵－向量乘法时,使用棋盘划分比带状划分为优。

9.4 试证明：在使用带状划分的超立方上执行矩阵－向量乘时，其等效率函数 $f_E(p)=O(p^2)$。

[提示：将(9.4)式代入 $\hat{p}T_P=T_0+W$ 从而求出 T_0，再由 $W=KT_0$ 和 $W=n^2$ 求出 n 与 p 的关系，最后求出等效率函数 $f_E(p)=f(p)$]。

9.5 试证明：在使用棋盘划分的网孔上执行矩阵－向量乘时，其等效率函数 $f_E(p)=O(\max\{p^{3/2},p\log p\})$。

[提示：参见习题9.4的提示。]

9.6 试证明：在超立方上，并行分块矩阵乘法和Cannon乘法的等效率函数均为 $f_E(p)=O(p^{3/2})$，而Fox乘法的等效率函数为 $f_E(p)=O(p^{3/2}\log p)$。

9.7 根据9.4.3节所讨论的Fox乘法：

①试写出Fox乘法的形式描述；

②试分析Fox乘法的主要优点是什么。

9.8 试推导：在 $p^{1/3}\times p^{1/3}\times p^{1/3}$ 三维网孔上，使用CT选路方法时DNS乘法的并行运行时间。

9.9 算法9.7给出了 n^2 个处理器的并行系统上用PRAM－CREW模型施行两个 $n\times n$ 矩阵相乘的算法。假定存储器的读写时间为 t_s，两个元素的乘－加时间为 t_c。试分析该算法的并行运行时间。

算法9.7 PRAM－CREW上矩阵相乘算法

输入： $A_{n\times n}$，$B_{n\times n}$

输出： $C_{n\times n}$

```
Begin
    (1) 将 n^2 个处理器组织成 n×n 的网孔
    (2) for each P_{i,j} do
        (2.1) c_{i,j}=0
        (2.2) for k=0 to n-1 do
                  c_{i,j}=c_{i,j}+a_{i,k}×b_{k,j}
              endfor
    endfor
End
```

9.10 参照图9.14，算法9.8描述了 $m\times k$ 二维Systolic阵列上实现 $A_{m\times n}\times B_{n\times k}=C_{m\times k}$ 的矩阵乘法，它是采用流水线原理，通过在时间上延迟矩阵元素的办法来达到一对下标合宜的矩阵元素适时相乘的目的。

算法9.8 SIMD－MC^2 上Systolic矩阵相乘算法

输入： $A_{m\times n}$，$B_{n\times k}$

输出： 在 $P_{i,j}$ 中存有乘积矩阵元素 $c_{i,j}$

```
Begin
  for i=1 to m par-do
    for j=1 to k par-do
      (1) c_{i,j}=0
      (2) while P_{i,j}收到 a 和 b 时 do
        (2.1) c_{i,j}=c_{i,j}+a×b
        (2.2) if i<m then 发送 b 给 P_{i+1,j} endif
        (2.3) if j<k then 发送 a 给 P_{i,j+1} endif
        endwhile
    endfor
  endfor
End
```

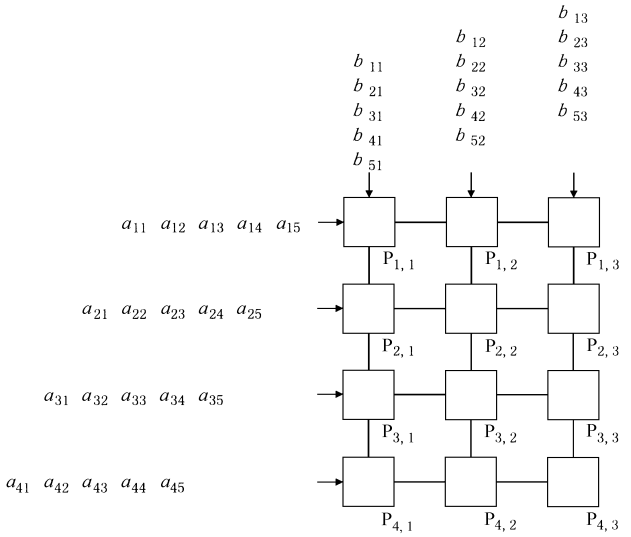


图 9.14　矩阵 A 和 B 的输入加载方式

试问：① 为了确保 $a_{i,s}$ 与 $b_{s,j}$ 适时在 $P_{i,j}$ 相遇，A 矩阵的第 i 行要比第 $i-1$ 行 $(2 \le i \le m)$ 滞后多少时间单位？同样，B 矩阵的第 j 列要比第 $j-1$ 列 $(2 \le j \le k)$ 滞后多少时间单位？

② 当 $j=k$ 时，a 传送给 $P_{i,j+1}$ 吗？当 $i=m$ 时，b 传送给 $P_{i+1,j}$ 吗？

第十章　线性方程组的求解

很多科学与工程计算问题大都可以化为线性代数方程组的形式，所以有效的求解线性方程组在科学和工程计算中是非常重要的。虽然线性代数方程的求解方法和数值计算软件包（如 Linpack 等）均很成熟，但随着并行计算机的发展，问题的求解速度和解题规模都大大提高，因而使数值计算方法和相应的数学软件包都产生了变化，相应地线性方程组的有效并行求解也引起了人们的普遍关注。本章主要讨论基本线性方程组的求解，包括三角形方程组的求解、三对角方程组的求解、稠密线性方程组的求解和稀疏线性方程组的求解。对于每一种求解问题，我们首先讨论它的数值求解数学原理，接着给出其串行算法，然后分析串行算法中的并行性及并行化方法并给出相应的并行算法（有时我们只作简单的并行化分析）。

10.1 三角形方程组的求解

10.1.1 基本术语

本小节是预备性知识。为了读者阅读下文方便,先给出几个有关定义:

定义10.1 一个 n 个变量 $x_1, x_2, \cdots, x_n$ 的线性方程(Linear Equation)可表示为:

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = b$$

其中,$a_1, a_2, \cdots, a_n$ 和 b 均为常数。

定义10.2 变量 $x_1, x_2, \cdots, x_n$ 的一组线性方程称为 n 元线性方程组(System of Linear Equations),也详称为 n 个变量 n 个方程线性联立方程组,简称为线性系(Linear System),可表示为:

$$\begin{aligned} a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n &= b_1 \\ a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n &= b_2 \\ &\cdots \\ a_{n1} x_1 + a_{n2} x_2 + \cdots + a_{nn} x_n &= b_n \end{aligned} \tag{10.1}$$

它通常也可写成矩阵-向量形式:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & & & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{bmatrix}$$

简记之为 $Ax = b$。其中矩阵 A 中非零元素的位置和值决定了求解 x 的困难复杂程度。通常一个顺序求解 $Ax = b$ 的算法的时间复杂度为 $O(n^3)$。

定义10.3 一个 $n \times n$ 的矩阵是上三角(Upper Triangular)的,如果 $i > j$ 时,$a_{ij} = 0$。

定义10.4 一个 $n \times n$ 的矩阵是下三角(Lower Triangular)的,如果 $i < j$ 时,$a_{ij} = 0$。

定义10.5 一个 $n \times n$ 的矩阵是三对角(Tridiagonal)的,当且仅当 $|i - j| > 1$ 时,$a_{ij} = 0$。

相应地,在 $Ax = b$ 中,若系数矩阵 A 是上(下)三角的,则称其为上(下)三角线性方程组;若系数矩阵 A 是三对角的,则称其为三对角线性方程组。

零元素出现在适当位置的线性方程组比任意形式的方程组的求解要快得多。例如上三角线性方程组和下三角线性方程组，在串行机上可在 $O(n^2)$ 时间界内求解，而三对角线性方程组则可在 $O(n)$ 时间界内求解。

定义 10.6　一个 $n\times n$ 的矩阵是对角占优(Diagonal Dominant)的，如果 $|a_{ii}| > \sum_{j\neq i} |a_{ij}|$，$1\leq i,j\leq n$。

定义 10.7　一个 $n\times n$ 的矩阵是对称的(Symmetric)，如果 $a_{ij}=a_{ji}$，$1\leq i,j\leq n$。

定义 10.8　一个 $n\times n$ 的矩阵称为正定的(Positive Definite)，如果对于非零向量 X 及其转置 X^T，则乘积 $X^TAX>0$。

例如所有对称的、对角占优的且 $a_{ii}>0$ 的矩阵都是正定的。

10.1.2　上三角方程组的求解

SISD 上回代解法　先看一个例子。假定欲求解以下上三角方程组：

$$\begin{aligned} 1x_1+1x_2-1x_3+4x_4&=8\\ -2x_2-3x_3+1x_4&=5\\ 2x_3-3x_4&=0\\ 2x_4&=4 \end{aligned}$$

先求解最后一个方程，得 $x_4=2$，将其值代入其它方程，消去 x_4 这一项，从而有

$$\begin{aligned} 1x_1+1x_2-1x_3&=0\\ -2x_2-3x_3&=3\\ 2x_3&=6\\ 2x_4&=4 \end{aligned}$$

再求解第 3 个方程，得 $x_3=3$，将其代入第 1 和 2 个方程，消去 x_3 这一项，从而有

$$\begin{aligned} 1x_1+1x_2&=3\\ -2x_2&=12\\ 2x_3&=6\\ 2x_4&=4 \end{aligned}$$

同样，消去 x_2，最后将欲求解的上三角方程化成了如下形式的对角方程：

$$\begin{aligned} 1x_1&=9\\ -2x_2&=12\\ 2x_3&=6\\ 2x_4&=4 \end{aligned}$$

易知 $x_1=9$。

一个回代法 (Back Substituting) 求解上三角方程组的顺序算法如算法 10.1。

算法 10.1 SISD 上回代求解上三角方程组算法

输入： $A_{n\times n}$, $b=[b_1,\cdots,b_n]^T$

输出： $x=[x_1,\cdots,x_n]^T$

```
Begin
  (1) for i = n to 1 do
      (1.1) x_i = b_i / a_ii
      (1.2) for j = 1 to i-1 do
                b_j = b_j - a_ji x_i
                a_ji = 0
            end for
    end for
  End
```

显然,算法 10.1 的复杂度为 $O(n^2)$。

如果分析一下算法 10.1,则知算法的 (1.2) 步,即 j 循环是可以并行化的,于是有如下 UMA 上回代求解上三角方程组的并行算法 10.2。

UMA 上回代解法 UMA 是一种共享存储的 MIMD 多处理机计算模型,其中所有处理器通过互连机制均匀一致地访问共享存储器。

算法 10.2 UMA 上回代求解上三角方程组算法

输入： $A_{n\times n}$, $b=[b_1,\cdots,b_n]^T$

输出： $X=[x_1,\cdots,x_n]^T$

```
Begin
    for i = n to 1 do
        x_i = b_i / a_ii
        for all P_j, where 1 ≤ j ≤ p do  /* p 为处理器数 */
            for k = j to i-1 step p do
                b_k = b_k - a_ki x_i
                a_ki = 0
            endfor
        endfor all
```

```
        endfor
    End
```

显然,算法 10.2 之复杂度为 $p(n)=p, t(n)=O(n)$。

10.2　三对角方程组的求解

本节打算介绍两种求解三对角方程组的方法:直接求解法,适合于在串行机上执行,但不适合并行化;奇偶归约求解法,虽有较高的比例常数,但易于并行化。

10.2.1　三对角方程组直接求解法

例示直接求解方法　三对角方程组中的系数矩阵,除了三条对角线上的元素为非零外,其余元素均为零,但通常没有只含一个变量的方程,因此不能用上节解上三角方程组的办法对它进行求解,为此需进行一些变换。先看一个具体例子。假定欲求解以下三对角方程组:

$$\begin{aligned}
16x_1+4x_2 &= 8 &①\\
4x_1+11x_2-5x_3 &= 7 &②\\
2x_2+14x_3-6x_4 &= 13 &③\\
5x_3+18x_4 &= 24 &④
\end{aligned}$$

首先,②$-\frac{1}{4}$①,消去第②个方程中的 x_1:

$$\begin{aligned}
16x_1+4x_2 &= 8 &①\\
10x_2-5x_3 &= 5 &②\\
2x_2+14x_3-6x_4 &= 13 &③\\
5x_3+18x_4 &= 24 &④
\end{aligned}$$

然后,③$-\frac{1}{5}$②,消去第③个方程中的 x_2:

$$\begin{aligned}
16x_1+4x_2 &= 8 &①\\
10x_2-5x_3 &= 5 &②\\
15x_3-6x_4 &= 12 &③\\
5x_3+18x_4 &= 24 &④
\end{aligned}$$

最后,④$-\frac{1}{3}$③,消去第④个方程中的 x_3:

$$\begin{aligned} 16x_1 + 4x_2 &= 8 \\ 10x_2 - 5x_3 &= 5 \\ 15x_3 - 6x_4 &= 12 \\ 20x_4 &= 20 \end{aligned}$$

此时三对角方程组的最后一个方程只含一个变量,这样就可以用回代法直接求解了。

三对角方程组的一般形式 在正式给出 SISD 上求解三对角方程组的算法之前,先考虑三对角方程组的一般形式。注意,在三对角方程组中,第一个和最后一个方程中只含有两个变量,其余均有三个变量,于是可以将三对角方程组一般化表示如下:

$$\begin{aligned} g_1 x_1 + h_1 x_2 &= b_1 \\ f_i x_{i-1} + g_i x_i + h_i x_{i+1} &= b_i \quad ,2 \le i \le n-1 \\ f_n x_{n-1} + g_n x_n &= b_n \end{aligned} \tag{10.2}$$

SISD 上直接求解法 一个直接求解三对角方程组的顺序算法如算法 10.3。

算法 10.3 SISD 上直接求解三对角方程组算法

输入: $A_{n\times n}, b=[b_1,\cdots,b_n]^T$

输出: $x=[x_1,\cdots,x_n]^T$

Begin

(1) for $i=1$ to $n-1$ do

$g_{i+1}=g_{i+1}-(f_{i+1}/g_i)h_i$

$b_{i+1}=b_{i+1}-(f_{i+1}/g_i)b_i$

end for

(2) for $i=n$ to 2 do

$x_i=b_i/g_i$

$b_{i-1}=b_{i-1}-x_i h_{i-1}$

end for

(3) $x_1=b_1/g_1$

End

显然,算法 10.3 的复杂度为 $O(n)$。

并行化分析 仔细分析算法 10.3,并参照上述具体例子可知,算法第 (1) 步,每次循环计算 g 与 b 时会用到前次循环计算的结果,所以此循环不能并行执行。同样,算法第 (2) 步,由于回代计算的顺序性,所以该循环也不能并行执行。因此,为了并行化必须另寻途径。

10.2.2　三对角方程组奇偶归约求解法

奇偶归约法原理　首先将(10.2)式所示的三对角方程组改写成如下的一个通式：

$$f_i x_{i-1} + g_i x_i + h_i x_{i+1} = b_i \quad , \quad 1 \le i \le n \tag{10.3}$$

$$f_1 = h_n = 0$$

然后奇偶归约法求解三对角方程组分成两步：第一步消去偶序号方程组中具有奇下标的 x_i 变量；第二步回代求解。消去 x_{2i-1} 和 x_{2i+1} 的方法可描述如下：

$$\begin{aligned} f_{2i-1} x_{2i-2} + g_{2i-1} x_{2i-1} + h_{2i-1} x_{2i} &= b_{2i-1} \\ f_{2i} x_{2i-1} + g_{2i} x_{2i} + h_{2i} x_{2i+1} &= b_{2i} \\ f_{2i+1} x_{2i} + g_{2i+1} x_{2i+1} + h_{2i+1} x_{2i+2} &= b_{2i+1} \end{aligned} \tag{10.4}$$

第一式乘以 $-f_{2i} g_{2i-1}^{-1}$ 后加到中间一式；第三式乘以 $-h_{2i} g_{2i+1}^{-1}$ 后亦加到中间一式，这样中间一式就变为：

$$\begin{aligned} &-f_{2i} g_{2i-1}^{-1} f_{2i-1} x_{2i-2} + (g_{2i} - f_{2i} g_{2i-1}^{-1} h_{2i-1} - h_{2i} g_{2i+1}^{-1} f_{2i+1}) x_{2i} \\ &\quad - h_{2i} g_{2i+1}^{-1} h_{2i+1} x_{2i+2} = b_{2i} - f_{2i} g_{2i-1}^{-1} b_{2i-1} - h_{2i} g_{2i+1}^{-1} b_{2i+1} \\ &\quad i = 1, 2, \cdots, (n/2) - 1 \end{aligned} \tag{10.5}$$

方程组(10.5)中只含偶下标的 x 变量，它消去了原方程组(10.3)式中含奇下标的 x 变量，该方程仍然是一个三对角方程组，但变量少了一半。重复上述过程可使方程组的变量越来越少，最后只剩下二个或三个方程：

$$\begin{aligned} g_2 x_1 + h_1 x_2 &= b_1 \\ f_2 x_1 + g_2 x_2 &= b_2 \end{aligned}$$

或

$$\begin{aligned} g_1 x_1 + h_1 x_2 &= b_1 \\ f_2 x_1 + g_2 x_2 + h_2 x_3 &= b_2 \\ f_3 x_2 + g_3 x_3 &= b_3 \end{aligned}$$

从最后的方程组就可解得 x_1、x_2 或 x_1、x_2、x_3。

一旦求出了上述 x 的值，就可以开始回代，即将 x_{2i-2}、x_{2i}、x_{2i+2} 代入(10.4)式即可解得 x_{2i-1} 和 x_{2i+1}。此过程一直进行到解出了原方程组的所有变量为止。

SISD 上奇偶归约法求解法　习题 10.1 给出了一种 SISD 上用奇偶归约法(Odd_Even Reduction)求解三对角方程组的算法形式描述。建议读者认真分析此算法，最好在计算机上编程实现它。

并行化分析　根据上述求解原理可知，在第一步消元时可同时消去具有奇下标的 x 变量，所以该步可直接并行化；同样，在回代时奇下标的变量亦可同时求

解,所以也可直接并行化。

最后,如果对于所有的 i,满足 $|g_i| \geq |f_i| + |h_i|$ (即对角占优),则消去奇下标变量后的方程组仍具有对角占优的性质。所以只要三对角方程组是对角占优的,奇偶归约法就可使用(见习题 10.2)。

10.3 稠密线性方程组的求解

求解稠密线性方程组的经典数值方法有直接三角分解法(即 LU 分解法),正交三角分解法(即 QR 分解法)和蝶形分解法(即 WZ 分解法)。本节介绍最基本的求解稠密线性方程组的高斯消去法(Gaussian Elimination)。

10.3.1 有回代的高斯消去法

基本原理 高斯消去法求解 $Ax=b$ 的基本思想是,将稠密系数矩阵 A 化为上三角阵 T,然后对 $Tx=c$ 施行回代求解。参照图 10.1,在消元的过程中,在第 i 步时为了消去第 i 列的第 $i+1$ 行到第 n 行中的元素(即化非零元素为零),可以用行 i 的倍数与其余行(第 $i+1$ 行到第 n 行)相减而达到目的。

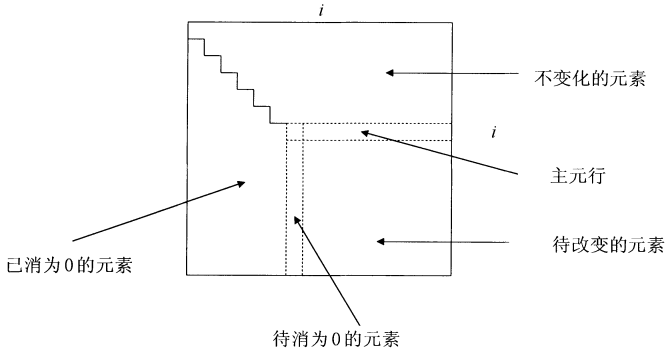


图 10.1 高斯消去法过程

选主元的高斯消去法 为了确保数值解的稳定性,在第 i 步时,应先找第 i 列中第 i 行到第 n 行中绝对值最大的元素,此元素称之为主元(Pivot);再将此元素所在的行(称为主元行)与第 i 行施行交换。由于交换系数矩阵的两行只相当于交换两方程的位置,所以不影响求解结果。在实际执行中,每次迭代时并不真正交换主

元行与第 i 行，而是引入一些参量，指明迭代时是否某一特定行已被选作主元行等就可以了。

SISD 上高斯主元消去法　一个有回代的、采用选主元的求解稠密线性方程组的高斯消去顺序算法如算法 10.4。其中，$marked_{1:n}$，指明哪些行已选作主元行；$pivot_{1:n}$，指明行 i 被用作主元行的迭代步数；picked 指明选作主元行的行。

算法 10.4　SISD 上求解稠密线性方程组高斯主元消去算法

输入：　$A_{n\times n}$，$b=(b_1,\cdots,b_n)^T$

输出：　$x=(x_1,\cdots,x_n)^T$

```
Begin
  (1) for i=1 to n do
        marked_i=0
      end for
  (2) for i=1 to n-1 do
        (2.1) temp=0
        (2.2) for j=1 to n do
                if marked_j=0 and |a_ji|>temp then
                  temp=|a_ji|
                  picked=j
                endif
              endfor
        (2.3) marked_picked =1
        (2.4) pivot_picked = i
        (2.5) for j=1 to n do
                if marked_j=0 then
                  (i) temp= a_ji/a_picked,i
                  (ii) for k= i+1 to n do
                         a_jk= a_jk - a_picked,k *temp
                       endfor
                  (iii) b_j= b_j - b_k *temp
                endif
              endfor
      end for
  (3) for i=1 to n do
```

```
            if marked_i = 0 then
                pivot_i = n
                break
            end if
        end for
    End
```

显然,算法 10.4 的复杂度为 $O(n^3)$。

并行化分析 根据算法10.4可知,一旦主元行选定,所有非主元行的修改可同时进行,也就是说算法 (2.5) 步和 (i) 步的中层的 j 循环和最内层的 k 循环可并行化。下面将讨论超立方连接的多计算机上如何执行高斯主元消去法。为此必须研究每次迭代时的数据相关性以便选择适当的处理器分配。如图 10.2 所示,在第 i 次迭代确定主元行时,要求比较 i 列中的各数据;而一旦主元行确定,未被选中的行 j 中的每个元素 a_{jk} 在更新时要用到 a_{ji}、$a_{\text{picked},i}$ 和 $a_{\text{picked},k}$。显然,a 和 b 的分配直接影响着通信要求。

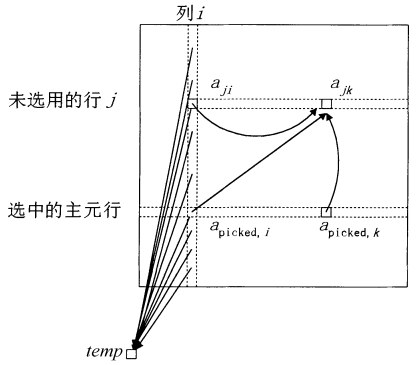


图 10.2 高斯消去法单步迭代时的数据相关性

超立方多计算机上高斯主元消去法 假定采用按行划分法分配处理器,具体实现方法如算法 10.5。其中,$\text{marked}_{1:n/p}$ 指明哪些行已是主元行,$\text{pivot}_{1:n/p}$ 指明迭代步数;value 表示主元值;winner 是控制主元行的处理器。

算法 10.5 超立方多计算机上求解稠密线性方程组高斯主元消去算法

输入: $A_{n\times n}$, $b=[b_1,\cdots,b_n]^{\mathrm{T}}$

输出： $x=[x_1,\cdots,x_n]^T$

```
Begin
  forall P_id, where 1≤id≤p do
    (1) for i=1 to n/p do  /*初始化;开始时所有行均非主元行*/
          marked_i=0
        end for
    (2) for i=1 to n-1 do/*各处理器找主元候选者*/
        (2.1) value=0
        (2.2) for j=1 to n/p do
                if marked_j=0 & |a_ji|>value then
                  value=|a_ji|
                  picked=j
                end if
              end for
        (2.3) winner=id /*标记控制主元行的处理器*/
        (2.4) MAX.TOURNAMENT (id,value,winner)  /*调用 MAX
                选全局最大主元*/
        (2.5) if id=winner then
                (i) marked_picked=1
                (ii) pivot_picked=i
                (iii) for j=i to n do
                        temp.vector_j=a_picked,j
                      end for
                (iv) temp.vector_n+1=b_picked
              endif
        (2.6) HC.BROADCAST (id,winner,temp.vector_i:n+1)  /*主元处
              理器播送主元给其它处理器*/
        (2.7) for j=1 to n/p do/*未选中的行中消去列 i 之元素*/
                if marked_j=0 then
                  (i) temp=a_ji/temp.vector_i
                  (ii) for k=i+1 to n do
                         a_jk=a_jk-temp.vector_k×temp
                       end for
```

```
                (ii) b_j = b_j − temp.vector_{n+1} × temp
                endif
            end for
        endfor
        (3) for i=1 to n/p do /* 定位从未被用作主元行的行 */
            if marked_i = 0 then
              pivot_i = n
              break
            end if
          end for
        end for
    End
    MAX.TOURNAMENT (id,value',winner) /* 被主程序所调用的过程 */
        Reference: id,value',winner
        局部变量:temp.value',temp.winner
    begin
        for i=0 to log p−1 do
          (1) partner= id ⊕ 2^i
          (2) [partner] temp.value'←value'
          (3) [partner] temp.winner ←winner
          (4) if temp.value' > value' then
                  value' = temp.value'
                  winner = temp.winner
              endif
        end for
    end
```

10.3.2 无回代的高斯－约旦法

基本原理 无回代的高斯－约旦法(Gauss_Jordan)与有回代的高斯消去法基本思想一样,区别是,后者通过一系列的消元,最终使系数矩阵变成一个上三角阵,然后由第 n 个方程先解出 x_n,令其代入第 $n-1$ 个方程解出 x_{n-1},如此一直回代下去,最后由第一个方程解出 x_1;而前者通过一系列消元,最终使系数矩阵变成一个对角阵,然后直接由第 i 个方程解出 x_i ($i=1,2,\cdots,n$)。

下面以 $n=4$ 为例来说明无回代的主元消去法的具体消元过程（为方便计算，令 b_i 改写成 $a_{i,n+1}$）：

① 在系数矩阵中找绝对值最大者（主元），假定为 a_{11}；将第一个方程乘以 $-a_{i1}/a_{11}$，分别与第 i 个方程相加（$i=2,3,4$），于是有：

$$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} & x_1 & a_{15} \\ 0 & b_{22} & b_{23} & b_{24} & x_2 & b_{25} \\ 0 & b_{32} & b_{33} & b_{34} & x_3 & b_{35} \\ 0 & b_{42} & b_{43} & b_{44} & x_4 & b_{45} \end{matrix}$$

② 除去第一行，在剩下的系数矩阵中再找主元，假定为 b_{22}；将第二个方程乘以 $-b_{i2}/b_{22}$，分别与第 i 个方程相加（$i=1,3,4$），于是有：

$$\begin{matrix} \boxed{a_{11}} & 0 & c_{13} & c_{14} & x_1 & c_{15} \\ 0 & \boxed{b_{22}} & b_{23} & b_{24} & x_2 & b_{25} \\ 0 & 0 & c_{33} & c_{34} & x_3 & c_{35} \\ 0 & 0 & c_{43} & c_{44} & x_4 & c_{45} \end{matrix}$$

③ 重复之，最后得到的线性方程组为：

$$\begin{matrix} \boxed{a_{11}} & 0 & 0 & 0 & x_1 & e_{15} \\ 0 & \boxed{b_{22}} & 0 & 0 & x_2 & e_{25} \\ 0 & 0 & \boxed{c_{33}} & 0 & x_3 & e_{35} \\ 0 & 0 & 0 & \boxed{d_{44}} & x_4 & d_{45} \end{matrix}$$

由此可直接解出 $x_1=e_{15}/a_{11}$，$x_2=e_{25}/b_{22}$，$x_3=e_{35}/c_{33}$ 和 $x_4=d_{45}/d_{44}$。

SISD 上高斯－约旦主元消去法　根据上述无回代求解稠密线性方程组的方法，参照算法 10.4，读者不难写出 SISD 上求解稠密线性方程组的高斯－约旦主元消去法的算法的形式描述，这作为习题留给读者去完成（习题 10.6）。

PRAM 上高斯－约旦消去法　根据以上的讨论可知，一旦主元行选定后，消元的操作则可同时进行。选主元的方法和 10.3.1 节一样，所以下面算法 10.6 中，为了突出主题，选主元行部分并未示出，但读者可以参照算法 10.5 将它们补上。下面算法中假定使用 n^2+n 个处理器，并排成 $n\times(n+1)$ 的阵列。

算法 10.6　PRAM 上求解稠密线性方程组高斯－约旦消去算法

输入：$A_{n\times n}$，$b=(b_1,\cdots,b_n)^{\mathrm{T}}$

输出：$x=(x_1,\cdots,x_n)^{\mathrm{T}}$

```
Begin
  (1) for j=1 to n do
        for i=1 to n par-do
          for k=j to n+1 par-do
            if i≠j then
              a_ik = a_ik - (a_ij/a_jj) a_jk
            end if
          end for
        end for
      end for
  (2) for i=1 to n par-do
        x_i = a_{i,n+1}/a_ii
      end for
End
```

注意此算法有可能多于一个处理器同时读取 a_{ij}、a_{jj}和 a_{jk}，所以此 PRAM 应是 CREW 的。

显然，算法的时间为 $O(n)$，用了 $O(n^2)$ 个处理器。

10.3.3 迭代求解的高斯－赛德尔法

基本原理 求解线性方程组的高斯－赛德尔法（Gauss Seidel Method），实际上是迭代（近似求解）法，不仅稠密线性系适用，而且稀疏线性系也适用。

在求解 $Ax=b$ 时，可以将系数矩阵 A 分解为 $A=E+D+F$，其中 D、E、F 均为 $n\times n$ 的矩阵，具体定义如下：

$$d_{ij}=\begin{cases}a_{ij}, & i=j\\ 0, & \text{否则}\end{cases}\qquad e_{ij}=\begin{cases}a_{ij}, & i>j\\ 0, & \text{否则}\end{cases}\qquad f_{ij}=\begin{cases}a_{ij}, & i<j\\ 0, & \text{否则}\end{cases}$$

这样，$Ax=(D+E+F)x=b$ 可变换成 $Dx=b-Ex-Fx$。如果给定初始解 $x(0)$，则第 k 次迭代可计算如下：

$$Dx(k)=b-Ex(k)-Fx(k-1) \tag{10.6}$$

例如，$n=4$，$k=1$ 时，

$$a_{11}x_1(1)=b_1-0-(a_{12}x_2(0)+a_{13}x_3(0)+a_{14}x_4(0))$$
$$a_{22}x_2(1)=b_2-a_{21}x_1(1)-0-(a_{23}x_3(0)+a_{24}x_4(0))$$
$$a_{33}x_3(1)=b_3-(a_{31}x_1(1)+a_{32}x_2(1))-0-a_{34}x_4(0)$$

$$a_{ii}x_i^{(1)} = b_i - (a_{i1}x_1^{(1)} + a_{i2}x_2^{(1)} + a_{i3}x_3^{(1)}) - 0$$

对于某一 k 和给定的误差允许值，如果下式成立，则认为迭代是收敛的：

$$\max_{i=1}^{n} |x_i^{(k+1)} - x_i^{(k)}| < \varepsilon$$

(10.6) 式能够加快顺序计算速度，因为当依次计算 x_1，x_2，…时，第 k 次迭代的 $x_i^{(k)}$ 值，一部分可用上次迭代的值（相应于上三角部分），而一部分可用本次迭代的值（相应于下三角部分）。

SISD 上高斯－赛德尔迭代法　注意 (10.6) 式可以展开为如下形式：

$$x_i^{(k)} = (b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)}) / a_{ii} \quad , \quad i=1,\cdots,n \tag{10.7}$$

按照 (10.7) 式可直接导出如下顺序算法 10.7。

算法 10.7　SISD 上求解线性方程组高斯－赛德尔迭代算法

输入：　$A_{n\times n}$，$b=(b_1,\cdots,b_n)^T$，精度 ε

输出：　$x=(x_1,\cdots,x_n)^T$

```
Begin
(1) for i=1 to n do
      x_i=0
    end for
(2) p=ε+1
(3) while p≥ε do
        (3.1) p=0
        (3.2) for i=1 to n do
                (i) t=x_i
                (ii) s=0
                (iii) for j=1 to n do
                        if j≠i then
                            s=s+a_ij x_j
                        end if
                      x_i=(b_i-s)/a_ii
                      if |x_i-t|>p then p=|x_i-t| endif
                    end for
              end for
```

```
        end while
    End
```

注意上述算法只有系数矩阵是对角占优的才适用。

并行化分析 上述算法无法直接并行化,因为在第 k 次迭代时,对于 $j<i$(下三角部分)的 $x_i(k)$ 的计算必须等待 $x_j(k)$ 计算好,因而无法并行。有关它的并行化方法,在 10.4 节讨论稀疏线性方程组的求解时还要进一步讨论。

10.4 稀疏线性方程组的求解

稀疏线性方程组系指系数矩阵中绝大多数元素均为零的方程组。之所以专门研究它,不仅仅是因为在科学和工程计算问题中会经常遇到它们,而且它们所涉及到的算法和数据结构也比稠密系统要复杂。

绝大多数科学和工程计算问题均代表了由数学模型所表示的一个物理系统。为了适合于计算机求解,连续物理系统的定义域必须离散化,其常用的方法就是将定义域网格化,这样在离散域上求解数学模型,就可在各格点上获得某些所要求的物理量。

通常离散物理域上每个格点的模拟计算,都是基于域上邻近格点的相互作用与影响。典型地,单格点的模拟就会产生一组线性方程。由于每一格点仅与其近邻有关,所以只有邻近格点的变量的系数是非零,而线性方程组的其余系数均为零,因而线性方程组就变成稀疏的了。

注意我们所考虑的稀疏矩阵,是指那些能够利用非零元素的位置与数量有效减少计算时间的稀疏矩阵。

10.4.1 稀疏矩阵的存储方式

通常我们用 $n\times n$ 的数组来存放 $n\times n$ 的稠密矩阵,但对稀疏矩阵而言,这种存储方式效率不高。研究有效的稀疏矩阵存储方式,不仅可节省存储单元,而且也可节省计算时间。但存储稀疏矩阵尚无一种通用的最佳的数据结构,不同的数据结构适合于不同的操作变换和不同的并行实现。以下我们仅研究几种最基本常用的稀疏矩阵存储方式。

坐标存储法(Coordinate Storage Scheme) 对于具有 q 个非零元素的稀疏矩阵,使用坐标存储法存储非零元素时,使用了三个 $q\times 1$ 维的数组:值数组 VAL 和

行列下标数组 I 和 J。图 10.3 示出了 6×6 稀疏矩阵及其坐标存储法（行列下标编号从 1 开始，下同）。

缩行存储法（Compressed Row Storage Scheme）　对于具有 q 个非零元素的稀疏矩阵，使用压缩稀疏行的办法存储非零元素时，也使用了三个数组：一个 $q\times1$ 维的值数组 VAL，它按行序分成了 n 个段；一个 $q\times1$ 维的列下标数组 J；一个 $n\times1$ 维的数组 I，该数组中的元素指向各段中首元素在稀疏矩阵中的顺序号（只计算非零元素）。图 10.4 示出了图 10.3 中的稀疏矩阵使用缩行存储法时的 VAL、J 和 I。

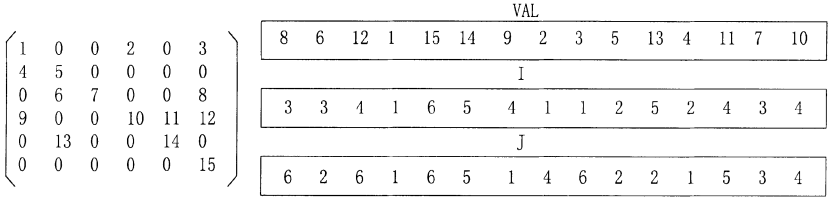


图10.3　6×6稀疏矩阵及其坐标存储法

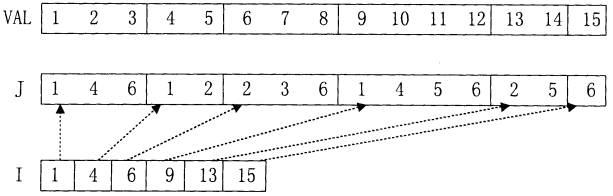


图 10.4　图 10.3 中稀疏矩阵的缩行存储法

对角存储法（Diagonal Storage Scheme）　对于具有 d 条对角线的 $n\times n$ 的稀疏矩阵，使用对角存储法时，使用了两个数组：一个 $n\times d$ 维的值数组 VAL，它每一列存储了一条对角线；一个 $d\times1$ 维的偏移数组（OFFSET），存储各对角线相对于主对角线偏移量。图 10.5 示出了一个稀疏矩阵及其对角存储方法。

按行存储法（Ellpack_Itpack Storage Scheme）　对于一个 $n\times n$ 的稀疏矩阵，假定任意行中最多非零元素数为 m，则使用这种存储法时使用了两个数组：一个 $n\times m$ 维的值数组 VAL，它每一行包含了稀疏矩阵中相应行中的非零元素；一个 $n\times m$ 维的数组 J，它存储了 VAL 中相应元素的列号，其中行末的 -1 是行结束符。图 10.6 示出了一个稀疏矩阵及其按行存储法。

齿对角存储法（Jagged_Diagonal Storage Scheme）　对于具有 q 个非零元素的稀疏矩阵，使用齿状对角线的形式存储非零元素时，首先将原矩阵按每行中非零元素之数目的降序重新排列之，然后辅之以三个数组：一个 $q\times1$ 维的值数组 VAL，

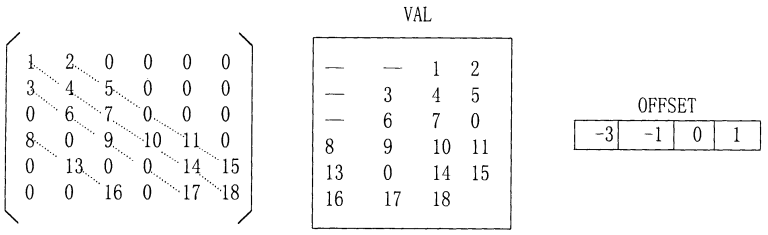


图 10.5 稀疏矩阵的对角存储法

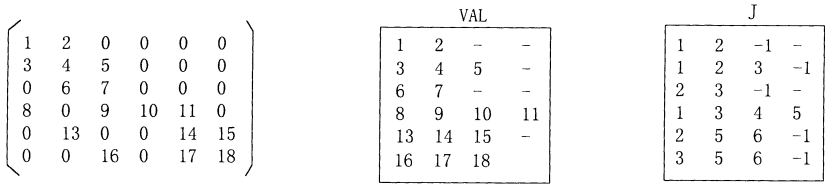


图 10.6 稀疏矩阵的按行存储法

它包含有若干段，每段存储一条齿状对角线；一个 $q\times 1$ 维的数组 J，它存储 VAL 中相应元素的列号；一个 $m\times 1$ 维的数组 I（m 是任意行中最多非零元素数），该数组之元素是指向各条齿状对角线中首元素依次在所有齿状对角线中的顺序号。图 10.7 示出了一个稀疏矩阵及其重新调整行序的稀疏矩阵以及它的齿对角存储法。

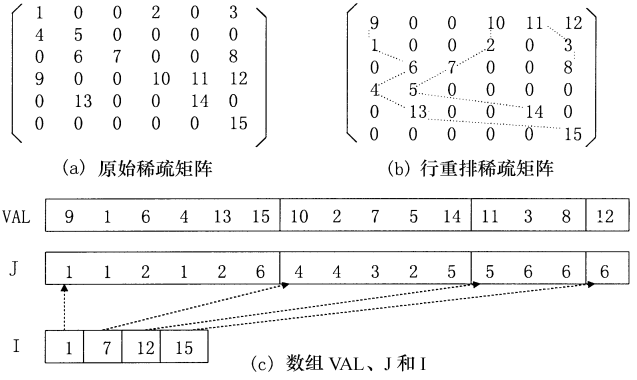


图 10.7 稀疏矩阵的齿对角存储法

10.4.2　雅可比迭代法

前面10.3.3节已经介绍了线性方程组的迭代求解法。对于求解大型、稀疏线性方程组，迭代法(Iterative Method)是一种最常使用的方法，与直接法相比，它具有方法简单，存储空间小(迭代法通常只对系数矩阵中的非零元素进行运算)等优点，特别是在有限步内无法得到问题的解时，迭代法就可在有限的迭代步数后，停止运算而得到足够好的近似解。本节先介绍雅可比迭代法，它是一种最简单的线性方程组的迭代解法。

SISD上雅可比迭代法　对于求解线性方程组 $Ax=b$，未知向量 x 的分量可写成如下形式：

$$x_i=\frac{1}{a_{ii}}\left(b_i-\sum_{j\neq i}a_j x_j\right) \tag{10.8}$$

雅可比迭代原理是，使用第 $k-1$ 步所计算的变量 $x_i(k-1)$，来计算第 k 步的 $x_i(k)$ 之值：

$$x_i(k)=\frac{1}{a_i}\left(b_i-\sum_{j\neq i}a_{ij}x_j(k-1)\right) \tag{10.9}$$

算法10.8给出了串行雅可比迭代算法，注意为了确保该算法总能收敛，要求系数矩阵 A 是对角占优的，即 $|a_i|>\sum_{j\neq i}|a_j|$，$1\leq i\leq n$。

算法10.8　SISD上求解线性方程组雅可比迭代算法
输入：　$A_{n\times n}$，$b=(b_1,\cdots,b_n)^T$，ε
输出：　$x=(x_1,\cdots,x_n)^T$

```
Begin
(1) for i=1 to n do /* x_i 赋初值*/
        x_i = b_i/a_i
    end for
(2) diff=ε
(3) while diff≥ε do
    (3.1) diff=0
    (3.2) for i=1 to n do
          (i) new x_i = b
          (ii) for j=1 to n do
                 if j≠ i then
```

```
                    new x_i = new x_i − a_ij x_j
                end if
            endfor
        (ii)  new x_i = new x_i / a_ii
        end for
    (3.3)  for i = 1 to n do
        (i)  diff = max {diff, | x_i − new x_i |}
        (ii)  x_i = new x_i
        end for
end while
End
```

算法 10.8 并行化分析　雅可比算法很适合于并行化，因为每次迭代均使用上次迭代的值。假定处理器按行分配，即每个处理器负责矩阵的若干行和相应的 b 分量，则在 while 循环时产生两次通信要求：在 (3.2) 步，每个处理器 P_i 在计算自己的 x_i (k) 时，必须用到上次计算出的 x_j ($k-1$)，$j \neq i$。为此每个处理器计算出所负责的变量后，必须将其播送给所有其它的处理器；在 (3.3) 步，每个处理器计算出局部差值 diff 后，必须从中选取最大者作为下一次迭代的判据，这种求最大值操作是在所有处理器之间的全局操作。

偏微分方程的差分解法　试考虑拉普拉斯方程的第一边值问题：

$$\begin{aligned} &\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0 \\ &u|_S=\mu(x,y) \end{aligned} \tag{10.10}$$

其中，$\mu(x,y)$ 为有界区域 D 的边界 S 上的已知函数。为了求解此偏微分方程，我们可以用差商代替偏导数，得到相应的差分方程，通过解差分方程就可得到偏微分方程的近似解。

试考虑在平面 (x,y) 上一个以 S 为边界的有界区域 D 上定解问题。为了用差分法求解，可以分别作平行 x 轴和 y 轴的直线簇 $x_i = id, y_j = jd$ ($i, j = 0,1,\cdots, n$；d 为步距)，从而构成一个等间距正方网格，而直线的诸交点 (x_i, y_j) 称为格点，记之为 (i,j)。当 $i, j = 1,2,\cdots, n-1$ 时的格点称为内格点，在 S 上的格点称为边界格点。所有内格点上，函数 $u(x,y)$ 的偏导数可用差商代替：

$$\frac{\partial u(x,y)}{\partial x} \approx \frac{1}{d}\ u(x_i + d, y_j) - u(x_i, y_j)$$

$$\frac{\partial u(x,y)}{\partial y} \approx \frac{1}{d}\ u(x_i, y_j + d) - u(x_i, y_j)$$

$$\frac{\partial^2 u(x,y)}{\partial x^2} \approx \frac{1}{d^2}\left[u(x_i+d,y_j)-2u(x_i,y_j)+u(x_i-d,y_j)\right]$$
$$\frac{\partial^2 u(x,y)}{\partial y^2} \approx \frac{1}{d^2}\left[u(x_i,y_j+d)-2u(x_i,y_j)+u(x_i,y_j-d)\right] \tag{10.11}$$

(10.11) 式代入 (10.10) 式则有：

$$u(x_i,y_j)=\frac{1}{4}[u(x_i+d,y_j)+u(x_i-d,y_j)+u(x_i,y_j+d)+ u(x_i,y_j-d)] \tag{10.12}$$

若格点 (i,j) 处的函数值 $u(x_i,y_j)$ 记之为 u_{ij}，且取 $d=1$，则 (10.12) 式可简化为：

$$u_{ij}=\frac{1}{4}\left[u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}\right] \tag{10.13}$$

它就是有名的五点格式，即任一格点 (i,j) 上 u_{ij} 的值等于周围相邻四格点上解的值的算术平均。

雅可比算法用于求解稀疏线性方程组　当用算法 10.8 来求解 (10.13) 式所示的方程组时，效率很低，因为它相应的系数矩阵是稀疏的，每行只有五个非零元素，为此我们设计如下算法 10.9 来求解它。照例，使用第 $k-1$ 步所计算的变量来计算第 k 步的变量。

算法 10.9　SISD 上求解拉普拉斯方程雅可比迭代算法
输入：格点数 n，精度 ε
输出：$x_{i,j}$

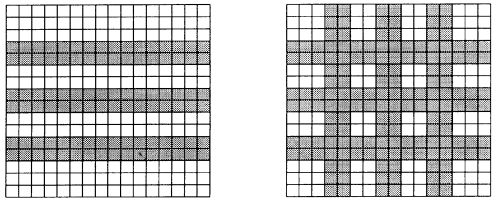
```
Begin
(1) for i=1 to n do    /*边界条件*/
        x0,i = north_i
        xn+1,i = south_i
        xi,0 = west_i
        xi,n+1 = east_i
    end for
(2) for i=1 to n do    /*赋初值*/
        for j=1 to n do
            xij = c
        end for
    end for
(3) diff=ε
```

```
(4) while diff≥ε do
      (4.1) diff=0
      (4.2) for i=1 to n do
              for j=1 to n do
                new x_ij = 1/4 (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})
              end for
            end for
      (4.3) for i=1 to n do
              for j=1 to n do
                diff=max{diff, |new x_ij - x_ij|}
                x_ij = new x_ij
              end for
            end for
    end while
End
```

算法 10.9 并行化分析　该算法是基于雅可比迭代原理，所有格点之值可同时更新。但在并行更新过程中会产生通信要求：在处理器按行分配时，如图 10.8 (a) 所示，每个处理器负责 $\frac{n}{p}\times n$ 个元素，算法 (4.2) 步每次迭代计算 new x_{ij} 时，处理器中的阴影区要分别传送 n 个元素给对方的处理器和同时接收来自对方的 n 个元素，令 λ 为延迟时间，β 为传送每个单值的时间，则每次迭代的通信时间为 $4(\lambda+n\beta)$；在处理器按块分配时，如图 10.8 (b) 所示，每个处理器负责 $n/p\times n/p$ 个元素，算法执行 (4.2) 步计算 new x_{ij} 时，处理器中的阴影区的通信时间为 $8\lambda+4n\beta/p$。同样，不管哪种处理器分配，算法 (4.3) 步计算全局 diff 时也会产生通信要求，为了减少此通信时间，可每迭代 k 次才计算一次 diff 的全局值，其中 k 值的大小与计算机的结构和求解问题类型有关。

10.4.3 高斯－赛德尔迭代法

基本原理　曾在第 10.3.3 节介绍了求解稠密线性方程组的高斯－赛德尔迭代法。本节着重介绍此法用于求解稀疏线性方程组。两者的计算原理是一样的，即在迭代时用最新算出的部分 x_i 值代替旧的 x_i 值参于迭代。不妨将 (10.7) 式改



(a) 按行划分 ($p=4$)　　(b) 按块划分 ($p=16$)

图 10.8　处理器的两种划分

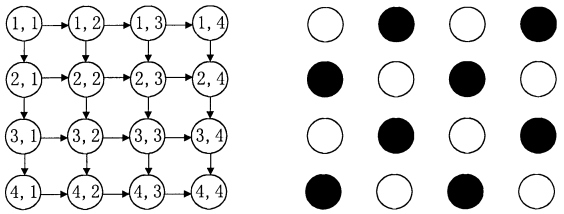
写成如下等效形式：

$$x_i^{(k)} = \left[b_i - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)}\right] / a_{ii} \tag{10.14}$$

按此，在第 k 次迭代时，只有当 $x_{i-1}^{(k)}$ 计算完成后才能开始计算 $x_i^{(k)}$，这意味着计算的顺序性。

并行化讨论　诚然，按照 (10.14) 式迭代计算时会加快收敛速度，但已如上述，第 k 次迭代时，对于系数矩阵的下三角部分，$x_j^{(k)}$ 的计算必须等待所有 $x_i^{(k)}$ 计算完毕时才能进行。这种计算的顺序性限制了并行化的实现，当系数矩阵为稠密阵时的确如此。然而，对于稀疏矩阵，$x_i^{(k)}$ 的计算不一定要等到 $x_1^{(k)}$，…，$x_{i-1}^{(k)}$ 都计算完毕才行。因为稀疏矩阵大多数元素均为零，如果 $a_{ij}=0$，那么 (10.14) 左边 $x_i^{(k)}$ 不依赖于 $x_j^{(k)}$，所以只要计算出 $j<i$ 且 $a_{ij}\neq 0$ 的 $x_j^{(k)}$，$x_i^{(k)}$ 便可计算。由此可见，高斯－赛德尔迭代法的并行化程度与系数矩阵 A 下三角部分中非零元素的分布状况有关。试考虑 (10.13) 式所示的五点差分格式的迭代计算问题，此时所有格点值的更新可按下述公式计算之：

$$x_{i,j}^{(k)} = \left[x_{i-1,j}^{(k)} + x_{i,j-1}^{(k)} + x_{i+1,j}^{(k-1)} + x_{i,j+1}^{(k-1)}\right] / 4 \tag{10.15}$$



(a) 数据相关分析　　(b) 红－黑着色法

图 10.9　高斯－赛德尔法并行化分析

对上式迭代计算进行数据相关分析。图 10.9(a) 示出了 4×4 格点迭代计算的数据相关性。由图可知，$x_{1,1}$ 之新值必须先计算；一旦算出 $x_{1,1}$，就可并行计算 $x_{1,2}$ 与 $x_{2,1}$；使用新计算的 $x_{1,2}$ 和 $x_{2,1}$，就可同时计算 $x_{1,3}$，$x_{2,2}$ 和 $x_{3,1}$，与此同时也可计算 $x_{1,1}$ 下一次迭代之值；同样，下一步就并行计算 $x_{1,4}$，$x_{2,3}$，$x_{3,2}$，$x_{4,1}$，$x_{1,2}$ 和 $x_{2,1}$。按此，可以将变量分为两组，每一组中的变量可以同时更新。一种方法就是有名的红－黑着色法 (Red_Black Coloring)。如图 10.9(b) 所示，在每一行和每一列，将格点交替地着成红色，其余的着成黑色，相同颜色的格点可同时更新。而格点的更新是以波前 (Wavefront) 形式从左上角向右下角逐步推进，并行化程度是不高的。事实上，使用红－黑法对格点进行着色后，所有红点的邻节点仅为黑色，同样所有黑点的邻节点仅为红色。这样，高斯－赛德尔迭代法第 k 次迭代时可分为两步：

① 第 1 步，同时计算所有的红点：

$$x_{i,j}(k)=\frac{1}{4}\left[x_{i-1,j}(k-1)+x_{i,j-1}(k-1)+x_{i+1,j}(k-1)+x_{i,j+1}(k-1)\right] \tag{10.16}$$

② 第 2 步，同时计算所有的黑点：

$$x_{i,j}(k)=\frac{1}{4}\left[x_{i-1,j}(k)+x_{i,j-1}(k)+x_{i+1,j}(k)+x_{i,j+1}(k)\right] \tag{10.17}$$

而第 2 步计算黑点时所用到的红点值均已在第 1 步计算出。

10.4.4 超松弛迭代法

雅可比超松弛法 (Jacobi OverRelaxation)　雅可比超松弛法是雅可比迭代法的变体，它组合旧 x_i 之值与用雅可比标准算法所算出的新 x_i 之值，来计算新的 x_i 之值，其更新计算遵循如下方程：

$$x_i(k)=(1-\gamma)\,x_i(k-1)+\frac{\gamma}{a_{ii}}\left[b_i-\sum_{j\neq i}a_j x_j(k-1)\right] \tag{10.18}$$

其中 γ ($0<\gamma\leq 1$) 为松弛因子，当 $\gamma=1$ 时上式就是标准的雅可比迭代公式 (10.9)。适当选择 γ 之值可加快收敛。

逐次超松弛法 SOR (Successive Over Relaxation)　逐次超松弛法是高斯－赛德尔迭代法的变体，它组合旧 x_i 之值与用高斯－赛德尔法所算出的新 x_i 之值，来计算新的 x_i 之值，其更新计算遵循如下方程：

$$x_i(k)=(1-\gamma)\,x_i(k-1)+\frac{\gamma}{a_i}\left[b_i-\sum_{j<i}a_{ij}x_j(k)-\sum_{j>i}a_j x_j(k-1)\right] \tag{10.19}$$

其中，γ ($0<\gamma\leq 1$) 为松弛因子。当 $\gamma=1$ 时上式就是高斯－赛德尔迭代公式 (10.14)。适当选择 γ 之值可加快收敛。

10.4.5　多重网格法

基本原理　多重网格 (Multigrid) 法是一类求解偏微分方程数值解的迭代方法，产生于两个事实：其一，迭代方法在粗网格上比细网格上收敛更快；其二，如果变量初始估计值合适，则迭代算法收敛也快。所以多重网格法就是基于上述事实，用前一组网格格点上的值去计算后一组网格格点上的值。其中，在细网格上的迭代可得到较准确的解，在粗网格上的迭代能加快收敛速度。适当地组合粗、细网格上的迭代就可得到多重网格迭代算法。

算法描述　为了简化讨论，仅考虑二维网孔上的多重网格法。约定：G^0 是 $n\times n$ 维 (n 是 2 的方幂) 的最细网格，G^i 表示较粗的网格 ($1\leq i<\log n$)，G^k 中的格点 (i,j) 对应着变元 x_{ij}^k。例如，对于 $n=16$，则 G^0 为 16×16 的网格 (最细网格)，G^1 为 8×8 的网格，G^2 为 4×4 的网格，G^3 为 2×2 的网格 (最粗的网格)。

在多重网格算法中，有三类运算：

① 松弛 (Relaxation)：计算 G^k 中所有 x_{ij}^k 之值，这可用雅可比迭代法、高斯－赛德尔迭代法、雅可比超松弛法、逐次超松弛法或其它迭代方法。

② 内插 (Interpolation)：用较粗网格 G^k 中变量 $x_{i,j}^k$ 之值，去计算较细网格 G^{k-1} 中变量 $x_{i,j}^{k-1}$ 之值。一种简单内插法是以与 $x_{i,j}^{k-1}$ 最近的 G^k 中变量之值取平均值作为 $x_{i,j}^{k-1}$ 之值。例如，$x_{3,5}^0=\dfrac{1}{4}(x_{2,4}^1+x_{2,6}^1+x_{4,4}^1+x_{4,6}^1)$。

③ 投射 (Projection)：用较细网格 G^{k-1} 中变量 $x_{i,j}^{k-1}$ 之值，去计算较粗网格 G^k 中变量 $x_{i,j}^k$ 之值。一种投射法是内射法，它取与细网格点相重合的粗网格点之值作为粗网格点之值，即 $x_{i,j}^k=x_{i,j}^{k-1}$。例如，$x_{4,4}^2=x_{4,4}^1$。

使用多重网格法时，由于在粗网格上未知量少，从而问题计算规模比细网格上小，因此粗网格上问题的求解比细网格上容易。如果首先近似地求出粗网格上的解，然后将其插值到细网格上作为细网格迭代的初值，则细网格上的松弛收敛将会加快。这种将粗、细网格优点相互结合的方法，使多重网格法的应用非常广泛。

并行化通信分析　多重网格算法中，松弛、内插和投射均会产生通信要求。其中，松弛计算的通信要求已在 10.4.2 节讨论过；而内射法 (Injection) 不产生通信要求；在简单内插时，细网格点之值由其近邻的诸粗网格点之值决定之。在此情况下，假定将最细网格 G^0 指派给 $n\times n$ 的二维处理器阵列上，那么在较粗网格 G^1，G^2，…，$G^{\log n-1}$ 上需要相互通信的处理器是不相邻的，其距离在 G^1 上为 2，在 G^2

上为4,如此等等。

当把网格 $G^0, G^1, \cdots, G^{\log n-1}$ 映射到超立方上时,情况比映射到二维网孔上复杂。这时可以使用葛莱码把二维网孔映射到超立方上,但这只能确保 G^0 中相邻的格点映射到超立方的相邻顶点上,而对较粗网格不能都这样。在最坏情况下,$n \times n$ 的网格中的某一点会有 $4\log n-2$ 个邻居,而点数为 n^2 的超立方仅有 $2\log n$ 个邻居。例如图10.10所示的8×8的网格中,点18有10个邻居,但64个顶点的超立方任一点都只有6个邻居。所以网格不能直接嵌入超立方中。但可以使 $G^1, \cdots, G^{\log n}$ 中的邻点在超立方上相离为2。为此可将处理器编号分为两部分,一部分相应于行,另一部分相应于列;然后使用葛莱码将相邻的行和列元素映射到超立方相邻的顶点上。

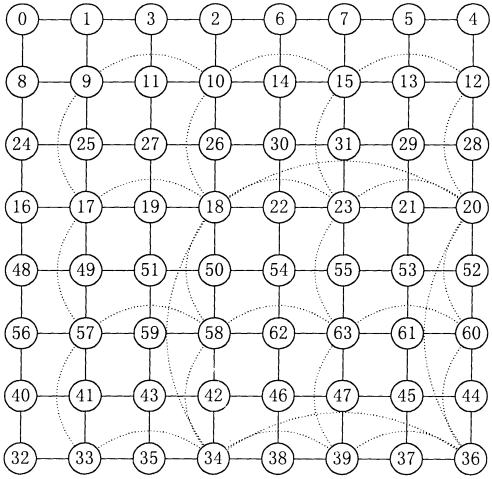


图10.10 二维多重网格映射到超立方上

10.4.6 共轭梯度法

共轭梯度(Conjugate Gradient)法,也叫共轭斜量法。从理论上讲它是属于直接法,但在实际计算过程中,由于不可避免地会出现舍入误差,所以常作为迭代法使用。该方法的最大特点是,当方程组的阶数很高时,往往只要经过比阶数小得多的迭代次数,就能得到满足精度要求的近似解。

基本思想 共轭梯度法是属于最小化类的迭代方法。为了求解 $Ax=b$ 这样的一次函数,可以先构造一个二次函数 $q(x)$,然后利用最小化原理对其求导

$\partial q(x)/\partial x$，再令其为零。如果$\partial q(x)/\partial x = Ax - b = 0$，那么就可达到求解 $Ax - b = 0$ 的目的。所以关键的问题是如何构造一个所要求的二次函数。

构造二次函数 $q(x)$　先定义一个二次齐次函数 $f(x)$，再定义一个一次线性函数 $g(x)$，将两者合成为 $q(x)$。

① 定义二次齐次函数：

$$f(x_1, x_2, \cdots, x_n) = a_{11}x_1^2 + a_{22}x_2^2 + \cdots + a_{nn}x_n^2 + 2a_{12}x_1x_2 + 2a_{13}x_1x_3 + \cdots + 2a_{n-1,n}x_{n-1}x_n$$

如果 $a_{ij} = a_{ji}$，则上式可写成如下形式：

$$\begin{aligned} f(x_1, x_2, \cdots, x_n) &= a_{11}x_1^2 + a_{12}x_1x_2 + \cdots + a_{1n}x_1x_n + \\ &\quad a_{21}x_2x_1 + a_{22}x_2^2 + \cdots + a_{2n}x_2x_n + \\ &\quad \cdots \\ &\quad a_{n1}x_nx_1 + a_{n2}x_nx_2 + \cdots + a_{nn}x_n^2 \\ &= x_1(a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n) + \\ &\quad x_2(a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n) + \\ &\quad \cdots \\ &\quad x_n(a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n \end{aligned}$$

$$= (x_1, x_2, \cdots, x_n) \begin{matrix} a_{11} & a_{12} & \cdots & a_{1n} & x_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & x_2 \\ & & \cdots & & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & x_n \end{matrix} = x^{\mathrm{T}}Ax$$

② 定义一次线性函数：

$$g(x_1, x_2, \cdots, x_n) = x_1b_1 + x_2b_2 + \cdots + x_nb_n$$

$$= (x_1, x_2, \cdots, x_n) \begin{matrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{matrix} = x^{\mathrm{T}}b$$

③ 合成 $f(x)$ 与 $g(x)$：

$$q(x) = \frac{1}{2}x^{\mathrm{T}}Ax - x^{\mathrm{T}}b$$

求二次函数 $q(x)$ 的最小值　为此要对 $q(x)$ 求偏导数，即$\partial q(x)/\partial x = \partial q(x)/\partial x_1 + \partial q(x)/\partial x_2 + \cdots + \partial q(x)/\partial x_n$，其中，

$$\frac{\partial q(x)}{\partial x_1} = \frac{1}{2}(2a_{11}x_1 + 2a_{12}x_2 + 2a_{13}x_3 + \cdots + 2a_{1n}x_n) - b_1$$

$$\frac{\partial q(x)}{\partial x_2}=\frac{1}{2}(2a_{21}x_1+2a_{22}x_2+2a_{23}x_3+\cdots+2a_{2n}x_n)-b_2$$

$$\cdots$$

$$\frac{\partial q(x)}{\partial x_n}=\frac{1}{2}(2a_{n1}x_1+2a_{n2}x_2+2a_{n3}x_3+\cdots+2a_{nn}x_n)-b_n$$

所以，

$$\frac{\partial q(x)}{\partial x}=\begin{matrix}a_{11}&a_{12}&a_{13}&\cdots&a_{1n}\\a_{21}&a_{22}&a_{23}&\cdots&a_{2n}\\&&&\cdots&\\a_{n1}&a_{n2}&a_{n3}&\cdots&a_{nn}\end{matrix}\begin{matrix}x_1\\x_2\\\cdots\\x_n\end{matrix}-\begin{matrix}b_1\\b_2\\\cdots\\b_n\end{matrix}=Ax-b$$

因此，所构造的二次函数 $q(x)$，求导后令其为零，正是要求解的线性方程组 $Ax-b=0$。

共轭梯度法 $\alpha(k)$ 与 $d(k)$ 的求取　用共轭梯度法求解 $Ax-b=0$ 的迭代公式如下：

$$x(k)=x(k-1)+\alpha(k)d(k) \tag{10.20}$$

新向量 $x(k)$ 的值由老向量 $x(k-1)$、迭代步长 $\alpha(k)$ 和方向向量 $d(k)$ 决定之，此时 $x(k)$ 至多迭代 n 次就可收敛。

① 求 $\alpha(k)$：对于给定的 $x(k-1)$ 和 $d(k)$，$\alpha(k)$ 的选取应使所构造的二次函数 $q(x)$ 取最小值。由 $\frac{\partial q(x)}{\partial x}=Ax(k)-b=0$ 和 $Ax(k)-b=A[x(k-1)+\alpha(k)d(k)]-b=0$，可求出 $\alpha(k)$ 为：

$$\alpha(k)=\frac{b-Ax(k-1)}{Ad(k)} \tag{10.21}$$

令 $r(k)$ 为第 k 次迭代的残向量：

$$r(k)=Ax(k-1)-b \tag{10.22}$$

(10.22) 代入 (10.21)，则有

$$\alpha(k)=-\frac{r(k)}{Ad(k)}=-\frac{d^{\mathrm{T}}(k)r(k)}{d^{\mathrm{T}}(k)Ad(k)} \tag{10.23}$$

可见，每次迭代时惟一的矩阵－向量乘就是 $Ad(k)$。

② 求 $d(k)$：求 $d(k)$ 时，应使函数 $q(x)$ 在 x_i 点最快地下降到 x_{i+1}，而函数 $q(x)$ 在 $x=x_i$ 点的变化率最快的方向就是 $q(x)$ 在 x_i 处的梯度，即

$$\mathrm{grad}\,q\Big|_{x_i}=\left(\frac{\partial q}{\partial x_1},\frac{\partial q}{\partial x_2},\cdots,\frac{\partial q}{\partial x_n}\right)^{\mathrm{T}}\Big|_{x_i}$$

所以迭代时取梯度方向作为 d。

可以证明，如果 A 是对称正定的，且 $d_1,d_2,\cdots,d_n$ 是相对于 A 共轭的（即

$d_i^{\mathrm{T}} A d_j = 0, 1 \leq i, j \leq n, i \neq j$，则 $x(k) = x(k-1) + \alpha(k) d(k)$ 最多迭代 n 次就可收敛到 $Ax = b$ 的解。

因此，从一组共轭矩阵 A 的方向向量，作为迭代方向求解 $Ax = b$ 的方法称之为共轭梯度法。在此法中，一组共轭 A 的方向向量 $d(k)$ 可选取如下：

$$d(k) = -r(k) + \frac{r^{\mathrm{T}}(k) r(k)}{r^{\mathrm{T}}(k-1) r(k-1)} d(k-1) \tag{10.24}$$

其中 $r^{\mathrm{T}}(k) r(k)$ 表示残向量转置与残向量之内积。

共轭梯度算法步骤　用共轭梯度法求解 $Ax - b = 0$ 的步骤如下：

① 初始化：$x(0) = 0, d(0) = 0, r(0) = -b$。

② 在第 k 次迭代时，计算 $x(k)$ 分为四步：

第 1 步：计算残向量（按照 (10.22) 式）：

$$r(k) = Ax(k-1) - b$$

第 2 步：计算方向向量（按照 (10.24) 式）：

$$d(k) = -r(k) + \frac{r^{\mathrm{T}}(k) r(k)}{r^{\mathrm{T}}(k-1) r(k-1)} d(k-1)$$

第 3 步：计算步长（按照 (10.23) 式）：

$$\alpha(k) = -\frac{d^{\mathrm{T}}(k) r(k)}{d^{\mathrm{T}}(k) Ad(k)}$$

第 4 步：计算新的近似变量（按照 (10.20) 式）：

$$x(k) = x(k-1) + \alpha(k) d(k)$$

SISD 上共轭梯度算法　用共轭梯度法求解 $Ax = b$ 的串行算法如算法 10.10。其中，denom1，denom2，num1，num2，tempvect(1：n) 为临时变量，主算法调用两个过程 INNER.PRODUCT（ ）和 MATRIX_VECTOR.PRODUCT（ ）。

算法 10.10　SISD 上求解 $Ax = b$ 共轭梯度算法

输入：　$A_{n \times n}$，$b = (b_1, \cdots, b_n)^{\mathrm{T}}$，$\varepsilon$

输出：　$x = (x_1, \cdots, x_n)^{\mathrm{T}}$

```
Begin
  (1)  for i=1 to n do     /*初始化*/
         d=0
         x_i=0
         r_i=-b_i
       end for
  (2)  for iteration=1 to n do     /*n次迭代*/
```

```
(2.1)  denom1=INNER.PRODUCT(r1:n, r1:n)   /*计算 r^T r*/
(2.2)  r1:n = MATRIX_VECTOR.PRODUCT(a1:n,1:n, x1:n)
(2.3)  for i=1 to n do  /*计算残向量 r(k) = Ax(k-1) - b*/
           ri = ri - bi
       end for
(2.4)  num1=INNER.PRODUCT(r1:n, r1:n)
(2.5)  if num1<ε then break endif  /*满足精度,退出*/
(2.6)  for i=1 to n do          /*计算方向向量*/
           di = -ri + (num1/denom1) d
       end for
(2.7)  num2=INNER.PRODUCT(d1:n, r1:n)   /*计算 d^T r*/
(2.8)  tempvect1:n=MATRIX_VECTOR.PRODUCT(a1:n,1:n, d1:n)
(2.9)  denom2=INNER.PRODUCT(d1:n, tempvect1:n)
(2.10) α=-num2/denom2     /*计算步长*/
(2.11) for i=1 to n do  /*修正 xi*/
           xi = xi + αdi
       end for
       end for
End
INNER.PRODUCT(a1:n, b1:n)
    begin
      result=0
      for i=1 to n do
          result=result+ aibi
      end for
      return result
    end
MATRIX_VECTOR.PRODUCT(a1:n,1:n, b1:n)
    begin
      for i=1 to n do
          resulti=0
          for j=1 to n do
```

```
            result_i = result_i + a_ij b_j
        end for
    end for
    return result_{1:n}
end
```

并行化通信分析 假定算法 10.10 在分布存储的并行机上实现，且每个处理器在其局存中存有 A 的若干行以及相应的向量 b、d、r 的分量。则由于并行化所造成的通信要求主要来源于向量内积 (INNERPRODUCT) 和矩阵与向量乘积 (MATRIX_VECTORPRODUCT)。在计算向量内积时，每个处理器计算它的局部积之和，为了计算总的积之和 (即内积) 要花费对数时间，所以向量内积的通信时间为 $O\left(\frac{n}{p}+\log p\right)$。在计算矩阵－向量之积时，每个处理器均需与其它处理器通信，以取得别的处理器中的向量 b 之分量，为此可先把各个处理器中 b 之分量集中在一个处理器中，然后由此处理器以树状形式播送给各处理器，这种通信策略的总耗费时间为 $O(n\log p)$。

10.5 小结和导读

小结 线性代数方程组的求解在科学和工程计算中应用非常广泛，这是因为很多科学与工程问题的计算，最终大都可以化为线性代数方程组的求解。本章主要讨论最基本的线性方程组的求解，包括三角形方程组、三对角方程组、稠密和稀疏线性方程组等的经典解法。

线性代数方程组的数值解法通常有两种：直接法和迭代法。直接法，又称消元法，是利用矩阵变换技巧，将一般的系数矩阵化为特殊形式 (如上三角和对角形式) 的矩阵以对方程组消解。其优点是可以预先估计运算量，并可得到问题的准确解，但由于实际计算过程中总存在着舍入误差，因此直接法得到的结果并非绝对精确，并且还存在着计算过程的稳定性问题。迭代法是一种逐步求精的近似求解过程，此法一般总是假定方程组中的系数 $a_{ii}\neq 0$ ($i=1,\cdots,n$)。其优点是简单，易于计算机编程，但它存在着迭代是否收敛和收敛快慢的问题。迭代求解的过程是，先给定初始解，然后逐次迭代下去，且理论上讲，每次迭代的结果都应改善前一次的计算结果。迭代过程由预先给定的精度要求来控制，但由于方程组的准确解一般是不知道的，因此判断某次迭代是否满足精度要求是困难的，通常我们总是认为当相

邻两次迭代值 $x_i(k)$ 与 $x_i(k-1)$ 也很接近时，x_i 与 $x_i(k)$ 也很接近，因此就可直接用条件 $\max\limits_{1\le i\le n}|x_i(k)-x_i(k-1)|<\varepsilon$ 来判定迭代是否终止。迭代法在求解稀疏线性方程组时经常使用，而很多的偏微分方程经差分化后也可化为稀疏线性方程组，所以在偏微分方程数值求解中，迭代法使用得尤其广泛。

导读　[136] 是一本很好的综述性专著，它全面地讨论了向量机和并行机上线性方程组的直接法和迭代法的并行求解方法；[91] 是一篇有关并行数值算法的很好的综述文章；[149] 展示了三角形方程组求解器可在多计算机上有效地实现；[96] 首先引入奇－偶归约算法；[73] 讨论了共享存储和分布存储结构的并行机上稠密线性方程组的并行算法；[90] 综述了稀疏线性方程组的并行求解算法；[135] 综述了在向量机和并行机上偏微分方程的求解方法；[44] 讨论了超立方多处理机上的多重网格算法；[50] 讨论了并行共轭梯度算法。在国内 [205] 全面地介绍了工程上常用的、行之有效的串行算法（主要偏重于数值问题的常用算法）；[204] 较深入、系统地介绍了 SIMD 并行机上并行数值计算方法，对 MIMD 并行机上的数值算法也作了简要介绍；[197] 深入而全面地论述了 SIMD 和 MIMD 模型上的数值代数、离散变换和卷积、微分方程、计算数论和最优化计算的并行算法，对并行排序算法也作了介绍。此外，在 [143] 书中第九章的参考文献注释中还列举了大量有关参考文献，读者可追踪进一步阅读。

习　题

10.1　如果在 (10.2) 式中，引入虚变量 x_0 与 x_{n+1} 以及 x_{-1} 与 x_{n+2} 且令它们为零：

① 试推导下式：

$$f_i\ \frac{b_{i-1}-f_{i-1}x_{i-2}-h_{i-1}x_i}{g_{i-1}}+g_ix_i+h_i\ \frac{b_{i+1}-f_{i+1}x_i-h_{i+1}x_{i+2}}{g_{i+1}}=b_i \tag{10.25}$$

$$1\le i\le n$$

为了简化，记 $r_i=f_i/g_{i-1}$，$\delta_i=h_i/g_{i+1}$，$1\le i\le n$，则 (10.25) 式可写成：

$$-r_if_{i-1}x_{i-2}+(g_i-r_ih_{i-1}-\delta_if_{i+1})x_i-\delta_ih_{i+1}x_{i+2}=b_i+r_ib_{i-1}-\delta_ib_{i+1} \tag{10.26}$$

$$1\le i\le n$$

根据 (10.26) 式，可设计如下 SISD 上奇偶归约法求解三对角线性方程组算法 10.11：

算法 10.11.1　SISD 上三对角方程组奇偶归约求解算法

输入：$A_{n\times n}$，$b=[b_1,\cdots,b_n]^{\mathrm{T}}$

输出：$x=[x_1,\cdots,x_n]^{\mathrm{T}}$

Begin

(1) for $i=0$ to $\log n-1$ do
　(1.1) $d=2^i$
　(1.2) for $j=2i+1$ to $n-1$ step $2d$ do
　　$r_j=f_j/g_{j-d}$, $\delta_j=h_j/g_{j+d}$, $f'_j=-r_j f_{j-d}$,
　　$g'_j=-\delta_j f_{j+d}-r_j h_{j-d}$, $h'_j=\delta_j h_{j+d}$
　　$b'_j=b_j+r_j b_{j-d}-\delta_j b_{j+d}$
　　end for
　(1.3) $r_n=f_n/g_{n-d}$
　(1.4) $f_n=-r_n f_{n-d}$
　(1.5) $g_n=g_n-r_n h_{n-d}$
　(1.6) $b_n=b_n+r_n b_{n-d}$
　(1.7) for $j=2i+1$ to $n-1$ step $2d$ do
　　$f_j=f'_j$, $g_j=g'_j$, $h_j=h'_j$, $b_j=b'_j$
　　end for
　end for
(2) $x_n=b_n/g_n$
(3) for $i=\log n-1$ to 0 step -1 do
　(3.1) $d=2^i$
　(3.2) $x_d=(b_d-h_d x_{2d})/g_d$
　(3.3) for $j=3d$ to n step $2d$ do
　　$x_j=(b_j-f_j x_{j-d}-h_j x_{j+d})/g_j$
　　end for
　end for
End

② 假定：

$$A=\begin{bmatrix}4&1&0&0&0&0&0&0\\4&11&-5&0&0&0&0&0\\0&2&14&-6&0&0&0&0\\0&0&5&18&-4&0&0&0\\0&0&0&1&2&1&0&0\\0&0&0&0&2&3&6&0\\0&0&0&0&0&2&1&12\\0&0&0&0&0&0&5&8\end{bmatrix},\quad B=\begin{bmatrix}2\\7\\13\\18\\6\\3\\9\\-1\end{bmatrix},$$

试求 $AX=B$。

10.2　试证明：在奇偶归约算法中，如果对于所有 i，满足 $|g_i|\geq|f_i|+|h_i|$（即对角占优），则消去奇下标变量后的方程组仍具有对角占优的性质。

［提示：考查(10.5)式，对于所有 i，如果 $|g_i|\geq|f_i|+|h_i|$ 其系数关系。］

10.3　试分析高斯主元消去法算法10.4的复杂度为：

$$(2n^3+3n^2-2n-3)/3$$

10.4　试用高斯主元消去法算法10.4，逐步求解下述线性方程组：

$$\begin{pmatrix} -1 & 2 & 1 & -2 \\ 2 & 2 & 0 & 1 \\ 1 & -1 & 3 & -5 \\ -2 & 3 & -4 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$$

10.5　试分析算法10.5的复杂度。如果处理器按列分配，情况有何变化？

10.6　根据10.3.2节所述原理，参照算法10.4，试写出SISD上求解稠密线性方程组的高斯－约旦主元消去法的算法。

10.7　试用高斯－赛德尔算法10.7，逐步求解下述线性方程组（$\varepsilon=10^{-6}$）

$$\begin{pmatrix} 1 & 3 & 2 & 13 \\ 7 & 2 & 1 & -2 \\ 9 & 15 & 3 & -2 \\ -2 & -2 & 11 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 7 \\ -1 \end{pmatrix}$$

10.8　对于如下的泊松方程

$$-\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=f \tag{10.27}$$

其中 $f=f(x,y)$ 不为零，主要问题就是定出未知函数 $u=u(x,y)$ 在某区域 Ω 内满足(10.27)，且在边界 $\partial\Omega$ *上满足给定的条件*。

① 试推导其差分方程为：

$$4u_{i,j}-(u_{i-1,j}+u_{i+1,j}+u_{i,j-1}+u_{i,j+1})=h^2 f_{i,j},\ 1\le i,j\le n-1 \tag{10.28}$$

其中，h *为网格间距*，$h=1/n$，$u_{i,j}$ 为格点 (i,j) 的函数值 $u(x_i,y_j)$。

② 如何变换(10.28)式使其成为如下线性方程组：

$$AU=F \tag{10.29}$$

其中，系数矩阵 A 为块三对角矩阵：

$$\begin{pmatrix} S & Q & & & \\ Q & S & Q & & \\ & \ddots & \ddots & \ddots & \\ & & Q & S & Q \\ & & & Q & S \end{pmatrix}$$

S 和 Q 都是 $(n-1)\times(n-1)$ 的矩阵，且 S 本身又是三对角矩阵：

$$\begin{pmatrix} 1 & -\frac{1}{4} & & & \\ -\frac{1}{4} & 1 & \frac{1}{4} & & \\ & \ddots & \ddots & \ddots & \\ & & -\frac{1}{4} & 1 & -\frac{1}{4} \\ & & & -\frac{1}{4} & 1 \end{pmatrix}$$

而 $Q=-\frac{1}{4}I$，I 为单位矩阵，向量 U 未知：

$$U^{\mathrm{T}}=(u_{1,1},u_{1,2},\cdots,u_{1,n-1},u_{2,1},u_{2,2},\cdots,u_{2,n-1},\cdots,$$
$$u_{n-1,1},u_{n-1,2},\cdots,u_{n-1,n-1})$$

向量 F 已知，$F_{(i-1)(n-1)+j}=f_{i,j}$。

10.9　试证明共轭梯度法求解 n 阶线性方程组，最多迭代 n 次即可收敛。

10.10　试用共轭梯度法，求解下述方程组。假定

$$x(0)=[0,0,1]^{\mathrm{T}}:$$

$$\begin{matrix}2&1&0\\1&2&1\\0&1&1\end{matrix}\begin{matrix}x_1\\x_2\\x_3\end{matrix}=\begin{matrix}1\\0\\1\end{matrix}$$

第十一章　快速傅里叶变换

20 世纪 60 年代是计算复杂性研究的主要里程碑。在此十年中发现了三个非常惊人的有效算法，即两整数乘法、离散的傅里叶变换和两矩阵相乘。其中 1965 年 Cooley 和 Tukey 所研究出的计算离散傅氏变换（DFT）的快速傅氏变换（FFT），将计算量从 $O(n^2)$ 下降到 $O(n\log n)$，从而使得 FFT 在数字信号处理、石油勘探、地震预报、医学断层诊断、编码理论、量子物理及概率论等领域中都得到了广泛的应用。长期以来，各种快速 FFT 的算法不断出现，成为数值代数方面最活跃的一个研究领域，而其意义远远超过了算法研究的范围，进而为诸多科技领域的研究打开了一个崭新的局面。本章先从离散傅氏变换 DFT 讲起，引出 Cooley-Tukey 著名 FFT 蝶式计算图；然后讨论串行 FFT 迭代算法和串行 FFT 递归算法；最后研究并行 FFT 算法，包括网孔连接、蝶式连接、立方连接的 SIMD 机器上的并行 FFT 算法以及超立方多计算机上的并行 FFT 算法。因为傅氏变换会涉及到一些复数运算，所以把其作为预备知识给出，熟悉的读者可以不必读它。

11.1 离散傅氏变换

*11.1.1 预备知识

复数及其表示 复数 Z 一般表示为 $z=a+ib$,其中 $i=\sqrt{-1}$ 称为虚数单位,a 和 b 分别为 z 的实部和虚部。上述 $z=a+ib$ 为复数的坐标表示法;它还有三角表示法 $z=r(\cos\theta+i\sin\theta)$。因为根据台劳级数,

$$\sin\theta=\theta-\theta^3/3!+\theta^5/5!-\theta^7/7!+\cdots$$
$$\cos\theta=1-\theta^2/2!+\theta^4/4!-\theta^6/6!+\cdots$$
$$e^{i\theta}=1+i\theta-\theta^2/2!-i\theta^3/3!+\theta^4/4!+i\theta^5/5!+\cdots$$
$$=(1-\theta^2/2!+\theta^4/4!+\cdots)+i(\theta-\theta^3/3!+\theta^5/5!+\cdots)$$
$$=\cos\theta+i\sin\theta$$

所以复数还可以表示为指数形式 $z=re^{i\theta}$。如果 $r=1,\theta=2\pi i/n$,则 $z=e^{2\pi i/n}$。

单位根与单位元根 方程 $x^n-1=0$ 有 n 个根,称为 n 次单位根。如果复数 ω 是 n 次单位根,($\omega^n-1=0$),则 ω^k (k 为整数) 也一定是 n 次单位根,因为 $(\omega^k)^n=(\omega^n)^k=1^k=1$。显然 $\omega^n-1=0$ 有 n 个根,如果这 n 个根 $1,\omega^1,\omega^2,\cdots,\omega^{n-1}$ 都不同,则称 ω^1 为 n 次单位元根 (Principal n^{th} Root of Unity)。显然 $\omega=e^{2\pi i/n}$ 是一个 n 次单位元根。注意,不是所有的 n 次单位根都可以称为 n 次单位元根。例如,1 是 n 次单位根,但不是 n 次单位元根。

在离散的傅氏变换中所定义的 ω 正好是一个 n 次单位元根。

例 11.1 如图 11.1,$n=8$ 则有 8 个不同的 8 次单位根,其中 ω^1 为 8 次单位元根:

$$\omega^1=e^{2\pi i/8}=e^{i\pi/4}=\cos\frac{\pi}{4}+i\sin\frac{\pi}{4}=\frac{2}{2}+i\frac{2}{2}$$
$$\omega^2=(e^{2\pi i/8})^2=e^{i\pi/2}=\cos\frac{\pi}{2}+i\sin\frac{\pi}{2}=0+i$$
$$\omega^3=\omega^2\cdot\omega^1=\cos\frac{3}{4}\pi+i\sin\frac{3}{4}\pi=-\frac{2}{2}+i\frac{2}{2}$$
$$\omega^4=\omega^2\cdot\omega^2=\cos\pi+i\sin\pi=-1+0$$
$$\omega^5=\omega^1\cdot\omega^4=\cos\frac{5}{4}\pi+i\sin\frac{5}{4}\pi=-\frac{2}{2}-i\frac{2}{2}$$

$\omega^6=\omega^2\cdot\omega^4=0-\mathrm{i}$

$\omega^7=\omega^4\cdot\omega^3=\frac{2}{2}-\mathrm{i}\frac{2}{2}$

$\omega^8=\omega^4\cdot\omega^4=1+0$ □

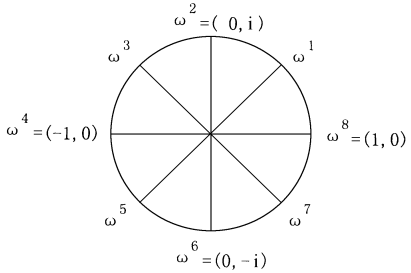


图 11.1 8 次单位元根及其它的 8 次单位根

单位根的性质 ①对于 $n\geq 2$，$\sum_{k=0}^{n-1}\omega^k=0$；② $\omega^n=1,\omega^{n/2}=-1$；③ $\omega^{sn+p}=\omega^p$（s、n、p 为正整数）。

11.1.2 离散傅里叶变换

一个 n 点离散傅里叶变换（Discrete Fourier Transform），简记之为 DFT，可定义为：给定序列 $(a_0,a_1,\cdots,a_{n-1})$，按如下规则变换成序列 $(b_0,b_1,\cdots,b_{n-1})$：

$$b_j=\sum_{k=0}^{n-1}\omega^{jk}a_k,\ 0\leq j\leq n-1 \tag{11.1}$$

其中，ω 是单位 n 次元根，即 $\omega=e^{2\pi i/n}$，$i=\sqrt{-1}$。

上述公式也可写成矩阵 ω（其元素 $\omega(j,k)$ 记之为 ω^{jk}）和向量 a 之乘积：

$$\begin{bmatrix}b_0\\b_1\\\vdots\\b_{n-1}\end{bmatrix}=\begin{bmatrix}\omega^0&\omega^0&\omega^0&\cdots&\omega^0\\\omega^0&\omega^1&\omega^2&\cdots&\omega^{n-1}\\\vdots&\vdots&\vdots&&\vdots\\\omega^0&\omega^{n-1}&\omega^{2(n-1)}&\cdots&\omega^{(n-1)(n-1)}\end{bmatrix}\begin{bmatrix}a_0\\a_1\\\vdots\\a_{n-1}\end{bmatrix} \tag{11.2}$$

例 11.2 给定 $a=(1,2,4,3)$，欲计算其 DFT。先计算 4 个不同的单位 4 次根：

$$\omega^1 = e^{2\pi i/4} = e^{i\pi/2} = \cos\frac{\pi}{2} + i\sin\frac{\pi}{2} = i$$

$$\omega^2 = -1$$

$$\omega^3 = \omega^1 \cdot \omega^2 = -i$$

$$\omega^4 = \omega^2 \cdot \omega^2 = 1$$

其系数矩阵 ω 为：

$$\begin{matrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{matrix} = \begin{matrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^2 \\ \omega^0 & \omega^3 & \omega^2 & \omega^1 \end{matrix} = \begin{matrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{matrix}$$

所以序列 b 为：

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i & 2 \\ 1 & -1 & 1 & -1 & 4 \\ 1 & -i & -1 & i & 3 \end{matrix} = \begin{matrix} 10 \\ -3-i \\ 0 \\ -3+i \end{matrix}$$

即 $b=(10,-3-i,0,-3+i)$。□

按照上述矩阵－向量相乘的方式来计算离散的傅氏变换，显然计算量为 $O(n^2)$。

11.1.3　离散傅里叶逆变换

一个 n 点的离散傅里叶逆变换（Inverse Discrete Fourier Transform），简记之为 IDFT，可类似定义为：

$$a_k = \frac{1}{n}\sum_{j=0}^{n-1} \omega^{-kj} b_j, \quad 0 \le k \le n-1 \tag{11.3}$$

例 11.3　例如序列 $b=(10,-3-i,0,-3+i)$ 之逆变换为：

$$\frac{1}{4}\begin{matrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^{-1} & \omega^{-2} & \omega^{-3} \\ \omega^0 & \omega^{-2} & \omega^{-4} & \omega^{-2} \\ \omega^0 & \omega^{-3} & \omega^{-2} & \omega^{-1} \end{matrix}\begin{matrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{matrix} = \frac{1}{4}\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{matrix}\begin{matrix} 10 \\ -3-i \\ 0 \\ -3+i \end{matrix}$$

$$= \frac{1}{4}\begin{matrix} 4 \\ 8 \\ 16 \\ 12 \end{matrix} = \begin{matrix} 1 \\ 2 \\ 4 \\ 3 \end{matrix}$$

即　$a=(1,2,4,3)$。□

11.1.4 离散傅氏变换的蝶式计算

本节根据离散傅氏变换(11.1)式来推演DFT蝶式计算流图。

当 $n=2$ 时，$\omega=e^{\frac{2\pi}{n}i}=\cos\pi+i\sin\pi=-1$：

$$\begin{bmatrix}b_0\\b_1\end{bmatrix}=\begin{bmatrix}1&1\\1&\omega\end{bmatrix}\begin{bmatrix}a_0\\a_1\end{bmatrix}=\begin{bmatrix}1&1\\1&-1\end{bmatrix}\begin{bmatrix}a_0\\a_1\end{bmatrix}$$

所以，$b_0=a_0+a_1$

$b_1=a_0-a_1$

其蝶式计算图如图11.2(a)所示。

当 $n=4$ 时，可以将其变成两个2点的DFT，兹推导如下，注意，$\omega^2=-1$，$\omega^3=-\omega$，$\omega^4=1$，$\omega^6=-1$，$\omega^9=\omega$：

$$\begin{bmatrix}b_0\\b_1\\b_2\\b_3\end{bmatrix}=\begin{bmatrix}1&1&1&1\\1&\omega&\omega^2&\omega^3\\1&\omega^2&\omega^4&\omega^6\\1&\omega^3&\omega^6&\omega^9\end{bmatrix}\begin{bmatrix}a_0\\a_1\\a_2\\a_3\end{bmatrix}=\begin{bmatrix}1&1&1&1\\1&\omega&-1&-\omega\\1&-1&1&-1\\1&-\omega&-1&\omega\end{bmatrix}\begin{bmatrix}a_0\\a_1\\a_2\\a_3\end{bmatrix}$$

对调 b_1 和 b_2，系数矩阵1行和2行亦同时对调：

$$\begin{bmatrix}b_0\\b_2\\b_1\\b_3\end{bmatrix}=\begin{bmatrix}1&1&1&1\\1&-1&1&-1\\1&\omega&-1&-\omega\\1&-\omega&-1&\omega\end{bmatrix}\begin{bmatrix}a_0\\a_1\\a_2\\a_3\end{bmatrix}$$

$$=\begin{bmatrix}1&1&0&0&1&0&1&0\\1&-1&0&0&0&1&0&1\\0&0&1&1&1&0&-1&0\\0&0&1&-1&0&\omega&0&-\omega\end{bmatrix}\begin{bmatrix}a_0\\a_1\\a_2\\a_3\end{bmatrix}$$

$$=\begin{bmatrix}1&1&0&0\\1&-1&0&0\\0&0&1&1\\0&0&1&-1\end{bmatrix}\begin{bmatrix}a_0+a_2\\a_1+a_3\\a_0-a_2\\(a_1-a_3)\omega\end{bmatrix}=\begin{bmatrix}1&1\\1&-1\\1&1\\1&-1\end{bmatrix}\begin{bmatrix}a_0+a_2\\a_1+a_3\\a_0-a_2\\(a_1-a_3)\omega\end{bmatrix}$$

$$= \begin{matrix} (a_0 + a_2) + (a_1 + a_3) \\ (a_0 + a_2) - (a_1 + a_3) \\ (a_0 - a_2) + (a_1 - a_3)\omega \\ (a_0 - a_2) - (a_1 - a_3)\omega \end{matrix}$$

所以原 4 点的 DFT 就化成了两个 2 点的 DFT：

$$b_0 = (a_0 + a_2) + (a_1 + a_3)$$
$$b_2 = (a_0 + a_2) - (a_1 + a_3)$$
$$b_1 = (a_0 - a_2) + (a_1 - a_3)\omega$$
$$b_3 = (a_0 - a_2) - (a_1 - a_3)\omega$$

其蝶式计算图如图 11.2 (b) 所示。

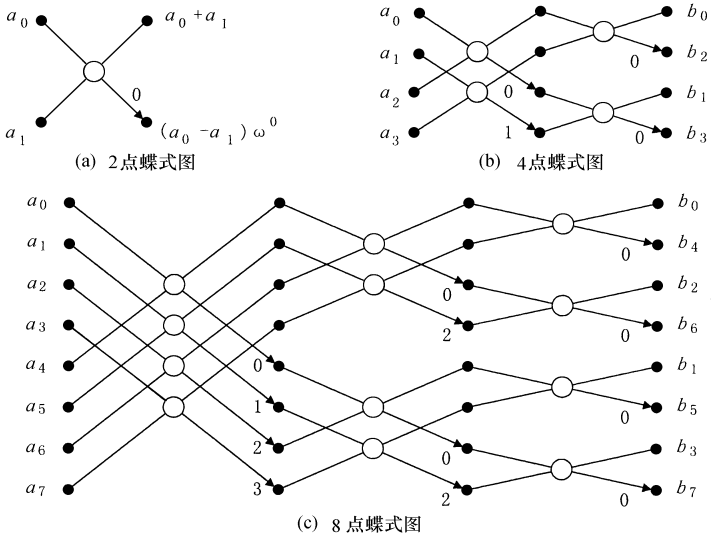


图 11.2　离散傅氏变换蝶式计算流图

当 n=8 时，8 个不同的单位 8 次根为 $\omega, \omega^2, \omega^3, \omega^4 = -1, \omega^5 = -\omega, \omega^6 = -\omega^2, \omega^7 = -\omega^3, \omega^8 = 1$：

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}$$

同样，我们可以将其化为两个4点的DFT，每个再化成两个2点的DFT，最后得：

$$b_0 = [(a_0 + a_4) + (a_2 + a_6)] + [(a_1 + a_5) + (a_3 + a_7)]\ \omega^0$$
$$b_1 = [(a_0 + a_4) + (a_2 + a_6)] - [(a_1 + a_5) + (a_3 + a_7)]\ \omega^0$$
$$b_2 = [(a_0 + a_4) - (a_2 + a_6)] + [(a_1 + a_5) - (a_3 + a_7)]\ \omega^2$$
$$b_3 = [(a_0 + a_4) - (a_2 + a_6)] - [(a_1 + a_5) - (a_3 + a_7)]\ \omega^2$$
$$b_4 = [(a_0 - a_4) + (a_2 - a_6)\ \omega^2] + [(a_1 - a_5) + (a_3 - a_7)\ \omega^2]\ \omega$$
$$b_5 = [(a_0 - a_4) + (a_2 - a_6)\ \omega^2] - [(a_1 - a_5) + (a_3 - a_7)\ \omega^2]\ \omega$$
$$b_6 = [(a_0 - a_4) - (a_2 - a_6)\ \omega^2] + [(a_1 - a_5) - (a_3 - a_7)\ \omega^2]\ \omega^3$$
$$b_7 = [(a_0 - a_4) - (a_2 - a_6)\ \omega^2] - [(a_1 - a_5) - (a_3 - a_7)\ \omega^2]\ \omega^3$$

其蝶式计算图如图11.2(c)所示。

＊11.2 快速傅氏变换串行算法

11.2.1 串行FFT迭代算法

FFT算法版本很多，但大体上可分为两类：迭代法和递归法。本节先讨论迭代法，下节再讨论递归法。

算法11.1 SISD上FFT迭代算法

输入： $a=(a_0,\cdots,a_{n-1})$

输出： $b=(b_0,\cdots,b_{n-1})$

Begin

(1) for $k=0$ to $n-1$ do／＊初始化＊／

```
            c_k = a_k
        end for
    (2) for h = log n − 1 to 0 do
        (2.1) p = 2^h
        (2.2) q = n/p
        (2.3) z = ω^(q/2)
        (2.4) for k = 0 to n − 1 do
              if (k mod p = k mod (2p)) then /* (i) 和 (ii) 同时执行 */
                 (i)  c_k = c_k + c_(k+p)
                 (ii) c_(k+p) = (c_k − c_(k+p)) z^(k mod p) /* c_k 不用 (i) 计算的值 */
              endif
        endfor
    endfor
(3) for k = 1 to n − 1 do /* 调整位序 */
          b_r(k) = c_k  /* r(k) 为 k 的位反 */
    endfor
End
```

显然,算法的复杂度为 $O(n\log n)$。

例 11.4　给定 $a=(a_0,a_1,a_2,a_3)$,试计算 $b=(b_0,b_1,b_2,b_3)$。算法执行第 (1) 步,计算出 $c_0=a_0$,$c_1=a_1$,$c_2=a_2$,和 $c_3=a_3$。算法执行第 (2) 步,$h=1$ 时,$p=2$,$q=2$,$z=\omega$,只有 $k=0$ 和 1 满足条件:当 $k=0$ 时计算出 $c_0=a_0+a_1$,$c_2=a_0-a_1$;当 $k=1$ 计算出 $c_1=a_2+a_3$,$c_3=(a_2-a_3)\omega$。$h=0$ 时,$p=1$,$q=4$,$z=\omega^2$,只有 $k=0$ 和 2 满足条件:当 $k=0$ 时计算出 $c_0=(a_0+a_2)+(a_1+a_3)$,$c_1=(a_0+a_2)-(a_1+a_3)$;当 $k=2$ 时计算出 $c_2=(a_0-a_2)+\mathrm{i}(a_1-a_3)$,$c_3=(a_0-a_2)-\mathrm{i}(a_1-a_3)$。算法执行第 (3) 步,结果为 $b_0=c_0$,$b_1=c_2$,$b_2=c_1$ 和 $b_3=c_3$。□

11.2.2　串行 FFT 递归算法

本节使用分而治之思想来推导递归的 FFT 计算算法。注意在下面推导中,反复使用 $\omega^n=1$,$\omega^{n/2}=-1$,$\omega^{ln}=1$ 和 $\omega^{sn+p}=\omega^p$ 这几个等式。

对于 $b_j=\sum_{k=0}^{n-1}\omega^{jk}a$,$0\le j\le n-1$。首先,令 j 为偶下标,即 $j=2l$

$0 \le l \le \frac{n}{2}-1$，注意 $\omega^{jn}=(\omega^j)^l=1$，$\omega^j=e^{2\pi j/n}$ 是单位 $\frac{n}{2}$ 次元根，于是

$$\begin{aligned}
b_j=b_{2l}&=\sum_{k=0}^{n-1}\omega^{2lk}a_k\\
&=a_0+\omega^{2l}a_1+\omega^{4l}a_2+\cdots+\omega^{2l\left(\frac{n}{2}-1\right)}a_{\frac{n}{2}-1}+\\
&\quad a_{\frac{n}{2}}+\omega^{2l}a_{\frac{n}{2}+1}+\omega^{4l}a_{\frac{n}{2}+2}+\cdots+\omega^{2l\left(\frac{n}{2}-1\right)}a_{n-1}\\
&=(a_0+a_{\frac{n}{2}})+\omega^{2l}(a_1+a_{\frac{n}{2}+1})+\omega^{4l}(a_2+a_{\frac{n}{2}+2})+\\
&\quad\cdots+\omega^{2l\left(\frac{n}{2}-1\right)}(a_{\frac{n}{2}-1}+a_{n-1})
\end{aligned}\tag{11.4}$$

所以，$(b_0,b_2,\cdots,b_{n-2})$ 是 $(a_0+a_{\frac{n}{2}},a_1+a_{\frac{n}{2}+1},\cdots,a_{\frac{n}{2}-1}+a_{n-1})$ 的 DFT 交换。

同样，令 j 为奇下标，即 $j=2l+1$，注意 $\omega^{n/2}=-1$，于是

$$\begin{aligned}
b_j=b_{2l+1}&=\sum_{k=0}^{n-1}\omega^{(2l+1)k}a_k\\
&=a_0+\omega^{2l+1}a_1+\omega^{2(2l+1)}a_2+\cdots+\omega^{\frac{n}{2}-1\ (2l+1)}a_{\frac{n}{2}-1}+\\
&\quad\omega^{\frac{n}{2}(2l+1)}a_{\frac{n}{2}}+\omega^{\frac{n}{2}+1\ (2l+1)}a_{\frac{n}{2}+1}+\cdots+\omega^{(n-1)(2l+1)}a_{n-1}\\
&=a_0+\omega^{2l}\omega a_1+\omega^{4l}\omega^2a_2+\cdots+\omega^{2l\left(\frac{n}{2}-1\right)}\omega^{\frac{n}{2}-1}a_{\frac{n}{2}-1}-\\
&\quad a_{\frac{n}{2}}-\omega^{2l}\omega a_{\frac{n}{2}+1}-\cdots-\omega^{2l\left(\frac{n}{2}-1\right)}\omega^{\frac{n}{2}-1}a_{n-1}\\
&=(a_0-a_{\frac{n}{2}})+\omega^{2l}\omega(a_1-a_{\frac{n}{2}+1})+\omega^{4l}\omega^2(a_2-a_{\frac{n}{2}+2})+\\
&\quad\cdots+\omega^{2l\left(\frac{n}{2}-1\right)}\omega^{\frac{n}{2}-1}(a_{\frac{n}{2}-1}-a_{n-1})
\end{aligned}\tag{11.5}$$

所以，$(b_1,b_3,\cdots,b_{n-1})$ 是 $(a_0-a_{\frac{n}{2}},\omega(a_1-a_{\frac{n}{2}+1}),\cdots,\omega^{\frac{n}{2}-1}(a_{\frac{n}{2}-1}-a_{n-1}))$ 的 DFT 交换。

根据 (11.4) 式和 (11.5) 式，就可画出如图 11.3 所示的离散傅里叶变换递归计算流图。图 11.4 就是一个按此递归方法计算的 $n=8$ 的 FFT 蝶式计算图。

下面给出串行的 FFT 递归算法。

算法 11.2 SISD 上 FFT 递归算法

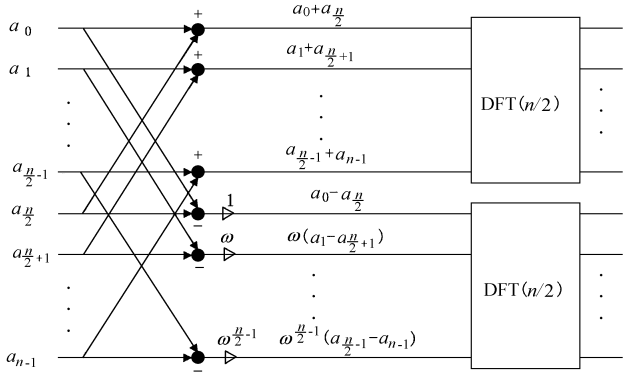


图 11.3　FFT 递归计算流图

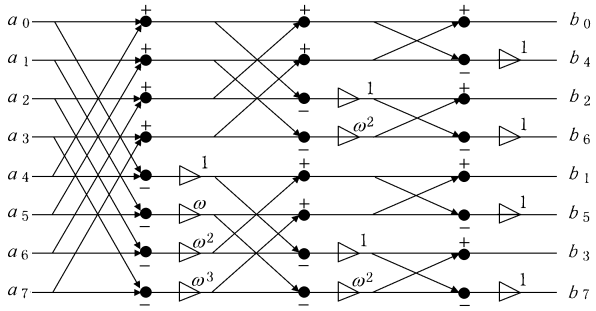


图 11.4　$n=8$ 的 FFT 蝶式计算图

输入：　$a=(a_0,\cdots,a_{n-1})$

输出：　$b=(b_0,\cdots,b_{n-1})$

Procedure　RFFT (a,b)

Begin

　if　$n=1$　then　$b_0=a_0$　else

　　(1)　RFFT $(a_0,a_2,\cdots,a_{n-2},u_0,u_1,\cdots,u_{\frac{n}{2}-1})$

　　(2)　RFFT $(a_1,a_3,\cdots,a_{n-1},v_0,v_1,\cdots,v_{\frac{n}{2}-1})$

　　(3)　$z=1$

　　(4)　for　$j=0$　to　$n-1$ do

(4.1) $b_j = u_{j \bmod \frac{n}{2}} + z v_{j \bmod \frac{n}{2}}$

(4.2) $z = z\omega$

endfor

endif

End

显然,上述算法的复杂度表示为 $t(n) = 2t\left(\frac{n}{2}\right) + O(n)$,解之得 $t(n) = O(n\log n)$。

11.3 并行 FFT 算法

11.3.1 SIMD_MC^2 上 FFT 算法

算法描述 本节所要描述的算法,实际上是算法 11.1 在网孔结构上的具体实现。假定 n 个处理器 $P_0, P_1, \cdots, P_{n-1}$ 排成 $n \times n$ 的方阵 ($n = 2^s \times 2^s = 2^{2s}$),处理器按图 11.5 所示的行主编号。$n$ 的二进制数表示为 $2^{\log n-1}2^{\log n-2}\cdots 2^1 2^0$。

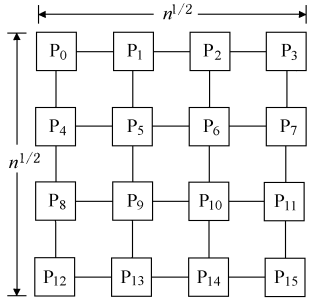


图 11.5 并行计算 FFT 的网孔结构

令 k 是 $\log n$ 位长的二进制整数,其位反为 $r(k)$ (例如,k=01011,则 $r(k)$ = 11010)。假定输入序列 $(a_0, a_1, \cdots, a_{n-1})$ 开始时已处于阵列的各处理器中,即 P_k 保存 a_k (k=0,…, n−1)。算法结束时,P_k 保存 b_k。网孔上 FFT 算法的形式描述如下:

算法 11.3 SIMD-MC^2 上 FFT 算法
输入： a_k 处于 P_k 中，$k=0,\cdots,n-1$
输出： b_k 处于 P_k 中
Begin
(1) for $k=0$ to $n-1$ par-do
$c_k = a_k$
endfor
(2) for $h=\log n-1$ to 0 do
for $k=0$ to $n-1$ par-do
(2.1) $p=2^h$
(2.2) $q=n/p$
(2.3) $z=\omega^q$
(2.4) if ($k \bmod p = k \bmod 2p$) then par-do
(i) $c_k = c_k + c_{k+p} z^{(k \bmod q)}$ /* (i) 和 (ii) 同时执行*/
(ii) $c_{k+p} = c_k - c_{k+p} z^{r(k) \bmod q}$
endif
endfor
endfor
(3) for k=0 to $n-1$ par-do
$b_k = c_{r(k)}$ /* $r(k)$ 为 k 的位反*/
endfor
End

其中，(2.4) 步的条件 $k \bmod p = k \bmod 2p$ 是确保参与运算的处理器处于同一行或同一列中；(2.3) 步是乘幂运算，是为了计算系数矩阵元素。如果也用 n 个排成 $n \times n$ 的处理器来计算的话，可令 $P(j,k)$ 负责计算 $\omega^{(j-1)(k-1)}$，$1 \le j,k \le n$。注意 $\omega^{(j-1)(k-1)}$ 的计算可重复执行平方和乘法（例如，$\omega^{13} = [(\omega^2)^2 \times \omega] \times [((\omega^2)^2)^2]$），所以可设计如下算法来计算傅氏变换的系数矩阵元素 ω^{jk}。假定每个处理器有三个寄存器：M_{kj} 存放 ω 的幂，X_{kj} 和 Y_{kj} 存放中间结果，而计算结果返回在 $Y_{kj} = \omega^{(k-1)(j-1)}$。

Procedure COMPUTE ω^{jk}
Begin

(1) $M_{kj}=(k-1)(j-1)$

(2) $X_{kj}=\omega$

(3) $Y_{kj}=1$

(4) while $M_{kj}\neq 0$ do

(4.1) if M_{kj}是奇数 then

$Y_{kj}=X_{kj}\cdot Y_{kj}$

endif

(4.2) $M_{kj}=?M_{kj}/2$

(4.3) $X_{kj}=X_{kj}^2$

end while

End

显然,上述算法的复杂度为 $O(\log n)$。

例 11.5 令 $n=4$,排成 2×2 的处理器 P_0、P_1、P_2 和 P_3 按行主编号。算法 11.3 执行第 (2) 步,$h=1$ 时,$p=2$,$q=2$,$z=\omega^2$,满足 $k \bmod 2=k \bmod 4$ 的处理器为 P_0 和 P_1:P_0 计算 $c_0=c_0+c_2(\omega^2)^0=a_0+a_2$,$c_2=c_0-(\omega^2)^0c_2=a_0-a_2$;$P_1$ 计算 $c_1=c_1+(\omega^2)^0c_3=a_1+a_3$,$c_3=c_1-(\omega^2)^0c_3=a_1-a_3$,此时同列中的处理器要进行通信。$h=0$ 时,$p=1$,$q=4$,$z=\omega$,满足 $k \bmod 1=k \bmod 2$ 的处理器为 P_0 和 P_2:P_0 计算 $c_0=c_0+\omega^0c_1=(a_0+a_1)+(a_2+a_3)$,$c_1=c_0-\omega^0c_1=(a_0+a_1)-(a_2+a_3)$;$P_2$ 计算 $c_2=c_2+\omega c_3=(a_0-a_2)+(a_1-a_3)\omega$,$c_3=c_2-\omega c_3=(a_0-a_2)-(a_1-a_3)\omega$。第 (3) 步执行结果为 $b_0=c_0$,$b_2=c_1$,$b_1=c_2$,$b_3=c_3$,其中计算 $b_1=c_2$ 和 $b_2=c_1$ 时处于对角的处理器要进行通信。□

算法分析 算法 11.3 的第 (1) 步是局部复制,只需取常数时间,且无须选路;算法的第 (2) 步既需要复数计算又需要选路;算法的第 (3) 步是位序调整只需选路。下面分析这两类操作的时间:

① 计算时间 t_c:算法的第 (2) 步中包含了+,−,×,÷和指数运算,其中最费时间的是指数操作,根据 Procedure COMPUTE 的分析,可知其时间复杂度为 $O(\log n)$。

② 选路时间 t_r:通信主要发生在第 (2.4) 步和第 (3) 步。在第 (2.4) 步时,如果 $k \bmod p=k \bmod 2p$,则 P_k 需要接收来自 P_{k+p} 中的 c_{k+p},然后再将 c_{k+p} 返回给 P_{k+p}。此选路所需的时间与 h 的值有关:当 $h=0$,$p=1$ 时,通信只发生在同行中那些下标差 1 的处理器间,所以选路步距为 1;当 $h=1$,$p=2$ 时,通信只发生在同

行中那些下标差 2 的处理器间，所以选路步距为 2；按此类推，当 $h=\log n-1$，$p=n/2$ 时，通信只发生在同列中那些下标差 $n/2$ 的处理器间，所以选路步距为 $n/2$。一般而言，对于 $p=2^h$，当 $h=2s-1,2s-2,\cdots,0$ 时，其选路步距为 $2^{h \bmod s}$。所以算法第 (2) 步的总的选路步距为 $2(1+2+4+\cdots+2^{s-1})=2(2^s-1)=O(n)$。在第 (3) 步时，通信发生在 P_k 和 $P_{r(k)}$ 之间，其最远者为两个对角处的处理器 P_{2^s-1} 和 $P_{2^s(2^s-1)}$ 之间，所以选路步距为 $2(2^s-1)=O(n)$。所以算法 11.3 总选路时间为 $O(n)$。当 n 充分大时，显然选路时间占主导地位。

11.3.2　SIMD_BF 上 FFT 算法

蝶形网络上 FFT 系数矩阵的计算　一个 $n=2^k$ 的蝶形网络 (简记为 BF) 有 $(k+1)2^k$ 个节点，布局成 $(k+1)$ 行，每行有 n 节点。令 (r,i) 表示第 r 行和第 i 列的坐标，$0\le i\le n-1$，$0\le r\le k$；$\exp(r,i)$ 表示在 BF 中坐标点 (r,i) 处的 ω 之指数，它等于字长为 k 的整数 j，即 $\exp(r,i)=j$，使得如果 i 的二进制表示为 $a_1a_2\cdots a_{r-1}a_r\cdots a_k$，则 j 的二进制为 $a_ra_{r-1}\cdots a_100\cdots0$。也就是说，将 i 的前 r 位取位反 (即倒序)，后面其余位补零就可以得到 j。例如，$\exp(r,i)=\exp(3,3)=j=6$，$\exp(2,7)=j=6$ 等。所以在 BF 中作 FFT 计算时，可将 $\omega^{\exp(r,i)}$ 想像为 $P(r,i)$ 中保留的系数。图 11.6 为 $n=8$ 的蝶形网络与相应的 FFT 系数矩阵元素的分布图。

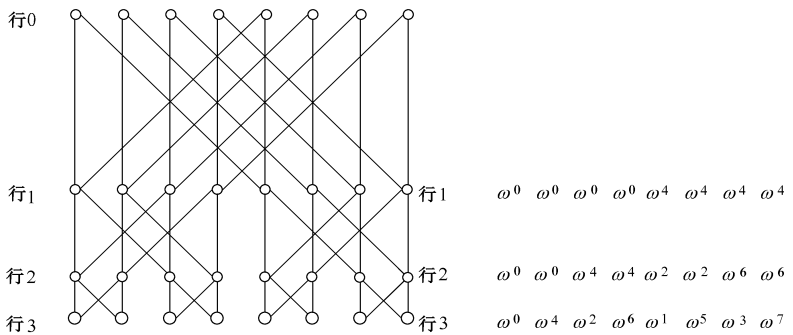


图 11.6　$n=8$ 的蝶形网络及其 8 点 FFT 系数分布

算法描述　假定系数 $\omega^{\exp(r,i)}$ 已按图 11.6 方式分布在网络的各处理器 $P(r,i)$ 中；开始时，序列 $(a_0,\cdots,a_{n-1})$ 并行地向第 0 行加载到各处理器中使得 $d_{0i}=a_i$；然后在网络中逐行计算 d_{ri} 之值；最终 d_{ki} 就是 b_j，其中 $j=\exp(k,i)$，i 和 j 的二进制位互为位反。蝶形网上 FFT 算法的形式描述如下：

算法 11.4 SIMD_BF 上 FFT 算法

输入： $a_0, a_1, \cdots, a_{n-1}$

输出： $b_0, b_1, \cdots, b_{n-1}$

Begin

(1) for $i=0$ to $n-1$ par-do /*初始加载*/

$d_{0,i} = a_i$

endfor

(2) for $r=1$ to $\log n$ do /*计算 $d_{r,i}$*/

for 所有仅第 r 位不同且 i 在第 r 位=0 的每对 (i, j) Par-do

(2.1) $d_{r,i} = d_{r-1,i} + \omega^{\exp(r,i)} d_{r-1,j}$

(2.2) $d_{r,j} = d_{r-1,i} + \omega^{\exp(r,j)} d_{r-1,j}$

end for

(3) for $i=0$ to $n-1$ par-do /*调整位序*/

(3.1) 计算 $\exp(k, i) = j$

(3.2) $c_j = d_{ki}$

(3.3) $b_i = c_{(j)}$

end for

End

算法分析 因为蝶形网络第 $r-1$ 行和第 r 行之间的连接，正好能满足直接将 $d_{r-1,i}$ 和 $d_{r-1,j}$ 传到 $P(r,j)$ 和 $P(r,j)$，所以无需考虑选路时间。至于计算时间，除了计算 $\omega^{\exp(r,i)}$ 的时间外，主要是算法第 (2) 步进行复数运算的时间，它等于 $O(\log n)$，显然优于 SIMD-MC^2 上的 FFT 算法，这也说明算法和体系结构的密切关系。

11.3.3 SIMD-CC 上 FFT 算法

对于立方连接的 SIMD 机器，我们可用 $n/2$ 个处理器来计算 n 点 FFT。例如对于图 11.7 所示的 16 点 FFT，可用 8 个处理器并行计算之，图 11.8 给出了这种计算过程：开始时 P_k 存在 a_k 与 $a_{k+n/2}$ $(0 \le k < n/2)$，然后逐级展开计算。整个计算需要 $\log n$ 步，每一步中，各处理器实现图 11.7 (b) 所示的蝶式计算。在图 11.8 中，逐级使用立方连接函数 C_2、C_1 和 C_0 来传递各次蝶式计算结果中的一个数据。在整个计算过程中，有 $\log(n/2)$ 次并行数据传输。由于立方连接正好满足并行传

输数据的要求，所以无需额外的选路时间。

如果用 $n/4$ 个处理器来计算 n 点 FFT，可将原两个处理器中并行执行的一对蝶式计算，合并成由一个处理器串行地执行两个蝶式计算。这样执行时间延长为 $2\log n$，而数据传输次数为 $2\log(n/4)$。一般而言，若用 $n/2^k$ 个处理器来计算 n 点 FFT $(2\le k\le \log n)$，则每个处理器最初应存入 2^k 个输入元素，要执行 $2^{k-1}\log n$ 个并行蝶式计算步，而立方连接函数 C 要重复 2^{k-1} 次 $(0\le k\le \log n-k-1)$，总的并行数据传输次数为 $2^{k-1}(\log n-k)$ 次。

11.3.4　MIMD-DM 上 FFT 算法

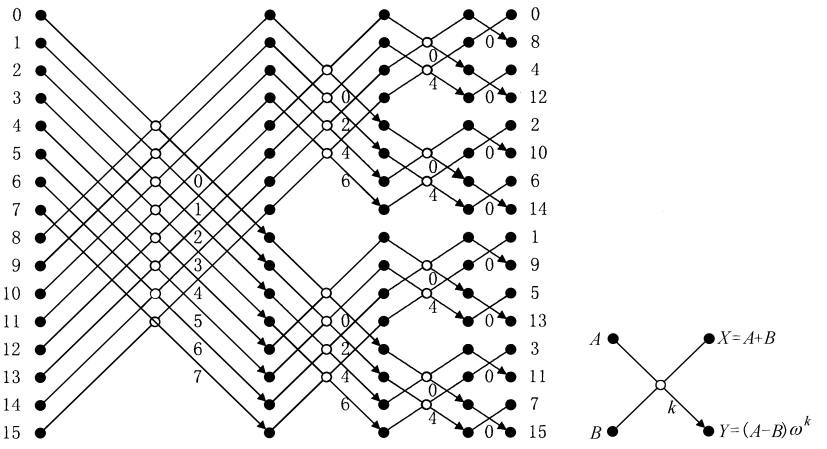
Cormen 迭代串行 FFT 算法　Cormen 算法 (Cormen's Algorithm) 是另一种形式的迭代算法，和算法 11.1 主要差别是，先将输入序列进行位反置换，从而输出序列就不需要进行位序调整了。

算法 11.5　SISD 上 Cormen 迭代计算 FFT 算法

输入：　$a_0, a_1, \cdots, a_{n-1}$

输出：　$b_0, b_1, \cdots, b_{n-1}$

Begin



(a) 16点FFT计算流图　　(b) 蝶式计算

图 11.7　16 点 FFT 计算

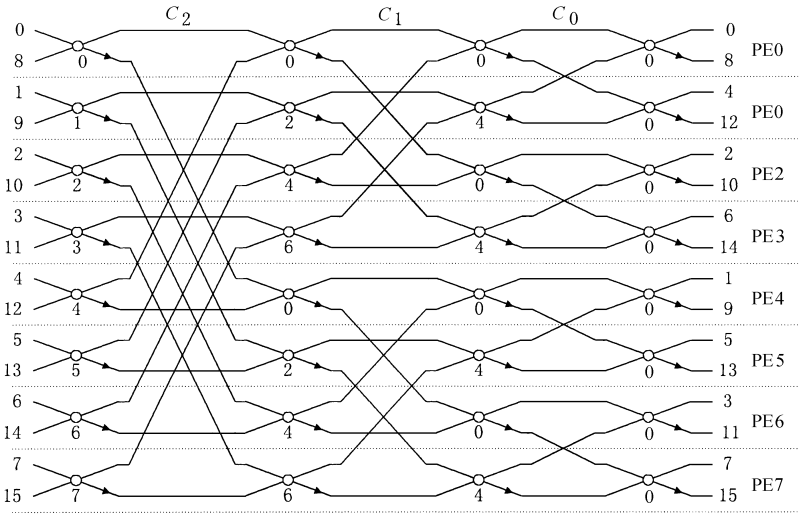


图 11.8 8个立方连接的处理器上16点FFT的计算

```
(1)  REVERSE (a, a') /* a'为 a 的位反序列 */
(2)  for s=1 to log n do
       (2.1)  m=2^s
       (2.2)  ω=e^(2πi/m)
       (2.3)  z=1
       (2.4)  for j=0 to m/2-1 do
                (i)  for k=j to n-1 step m do
                       v=za'_(k+m/2)
                       u=a'_k
                       b_k=u+v
                       b_(k+m/2)=u-v
                     endfor
                (ii)  z=z·ω
              endfor
     endfor
End
```

一个 $n=8$ 的 Cormen FFT 计算流图如图 11.9 所示。

超立方多计算机上 Cormen 算法　假定要在 p 个处理器的超立方上计算 n 点 FFT。参照图 11.10，算法可分为三步：第一步，将输入序列进行位反置换操作，每个处理器都必须计算输入元素的目的处理器号和它在输出序列中的位序号；第二步，各处理器执行 $\log(n/p)$ 次各自相应的 FFT 蝶式计算，此时处理器间不需通信；第三步，各处理器执行 $\log p$ 次各自的蝶式计算，此时处理器间需要进行通信。

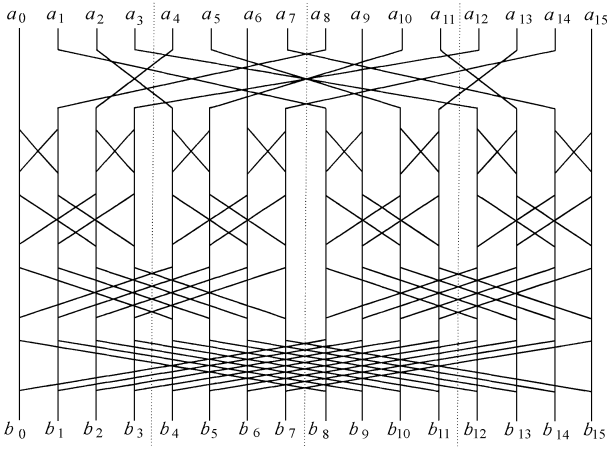


图 11.10　$n=16$，$p=4$ 的 FFT 计算流图

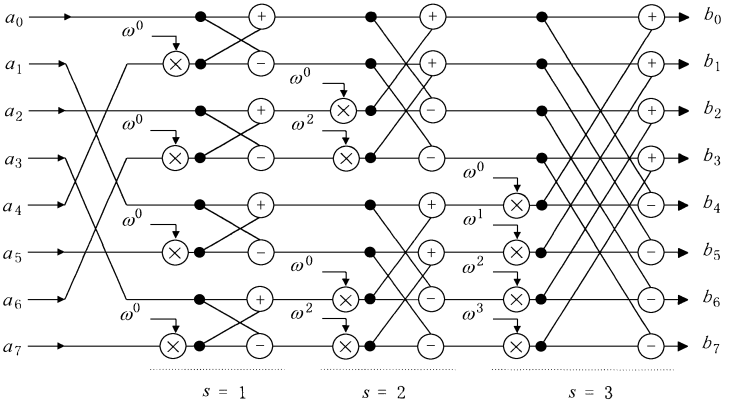


图 11.9　Cormen 8 点 FFT 计算流图

算法 11.6 超立方上 FFT 算法

输入： $a_0, a_1 \cdots a_{n-1}$

输出： $b_0, b_1 \cdots b_{n-1}$

Begin

(1) for all P_i, where $0 \le i < p$ do /*计算输入元素位反*/

for $k=0$ to $n/p-1$ do

id= $i(n/p)+k$

dest.processor = REVERSE (id)/(n/p)

dest.offset = REVERSE (id) mod (n/p)

[dest.processor] $b_{dest.offset}$ a a_k

end for

end for

(2) for $s=1$ to log (n/p) do /*无通信要求的迭代计算*/

(2.1) $m=2^s$

(2.2) $\omega=e^{2\pi i/m}$

(2.3) for all P_i, where $0 \le i < p$ do

(i) $z=1$

(ii) for $j=0$ to $m/2-1$ do

for $k=j$ to $n/p-1$ step m do

$q=z\,b_{k+m/2}$

$r=b_k$

$b_k=r+q$

$b_{k+m/2}=r-q$

endfor

$z=z\cdot\omega$

endfor

endfor

endfor

(3) for $s=1$ to log p do /*有通信要求的迭代计算*/

(3.1) $m=2^{s+\log(n/p)}$

(3.2) $\omega=e^{2\pi i/m}$

(3.3) for all P_i, where $0 \le i < p$ do

(i) if $i/2^{s-1}$ 是奇数 then

pos= $i\times\frac{n}{p}$ mod $m/2$
$z=e^{2(\text{pos})\pi i/m}$
for j=0 to $n/p-1$ do
$t_j=z\cdot b_j$
$z=z\cdot\omega$
endfor
endif
(i) shift=2^{s-1}
(ii) partner= k shift
(iv) if $i/2^{s-1}$是奇数 then
[partner] ua t
else
[partner] ua b
endif
endfor
(3.4) for all P_i,where $0\leq i<p$ do
if $i/2^{s-1}$是奇数 then
for $j=0$ to $n/p-1$ do
$b_j=u_j-t_j$
endfor
else
for $j=0$ to $n/p-1$ do
$b_j=b_j+u_j$
endfor
endif
endfor
endfor
End

11.4　小结和导读

小结　本章所讨论的快速傅里叶变换是一类最重要的快速离散变换,此类变

换还有数论变换、多项式变换、卷积和滤波等。其中,数论变换物理意义欠弱,所以应用尚不甚广泛;而卷积与滤波计算在数字信号处理中应用得十分广泛,因为许多数字信号处理问题都要求高速滤波能力(所谓滤波实际上是指将某些输入序列进行变换,使其具有某些预定的性质)。但本书限于篇幅就不再讨论它们了。

此外,本章所讨论的 FFT 算法是基－2 FFT 算法(Radix 2 FFT Algorithm),即将输入序列分为奇数下标和偶数下标两个 $n/2$ 点的序列进行递归计算。工程实用中,还常用到基－4 FFT 算法,即将输入序列分成四个 $n/4$ 点的序列进行递归计算,其计算量(乘法和加法)比基－2 算法有所减少。如果 n 不是单一基的幂,则可以使用混合基算法,要是算法设计得当,则可望达到最佳效果。同样限于篇幅,本章也不予以讨论。

最后,本章所讨论的 FFT 算法是一维 FFT 算法,如果输入元素是 a_{n_1,n_2} 形式的二维复序列,则可相应地定义二维 FFT 变换(Two Dimensional FFT Transform),它在光学、地震以及图像信号处理等方面起着重要的作用。也是限于篇幅,不再予以讨论。

导读　快速离散傅氏变换 FFT 最原始的文章是[53]。在计算机科学的教科书中,介绍串行 FFT 算法者很多,较早的有[7]、[20]和[54]等。专门介绍快速傅氏变换和卷积算法的著作可参见[132]。在国内科技著作中,介绍傅氏变换并行算法的,主要可参考[191]、[204]和[197],[190]中还专门讨论了 VLSI 阵列中的卷积、滤波和傅氏变换等并行算法。至于 FFT 算法在超立方多计算机和向量多处理器上的并行实现,读者可参阅[172]、在商用 MIMD 机器上的并行算法可参阅[19]和在 Transputer 网络上的并行算法可参阅[99]等。

习　题

11.1　试计算下述序列的 DFT:

(a)　(13,17,19,23)

(b)　(2,1,3,7,5,4,0,6)

11.2　试计算下述序列的逆 DFT:

(a)　(16, −0.76+8.66i, −6+6i, −9.25+2.66i, 0, −9.25−2.66i, −6−6i, −0.76−8.66i)

(b)　(4−i, 2+i, 2+i, −i, 4−i, 2+i, 2+i, −i)

11.3　给定两个 $n-1$ 阶多项式 $f(x)=\sum_{j=0}^{n-1} a_j x^j$ 和 $g(x)=\sum_{k=0}^{n-1} c_k x^k$,可以利用 FFT 变换及其

逆变换来计算两个两项式的乘积 $h=f\cdot g$,其步骤如下：

①令 N 是大于等于 $2n-1$ 的 2 的方幂的最小整数,在序列 $(a_0,\cdots,a_{n-1})$ 和 $(c_0,\cdots,c_{n-1})$ 之后各补上 $N-n$ 个零；

②计算 $(a_0,\cdots,a_{n-1},0,\cdots,0)$ 的 FFT,得到多项式 f 在单位 N 次根之值；

③计算 $(c_0,\cdots,c_{n-1},0,\cdots,0)$ 的 FFT,得到多项式 g 在单位 N 次根之值；

④计算 $f(\omega^j)\times g(\omega^j)$ 之积 $(j=0,1,\cdots,N-1)$,其中 $\omega=e^{2\pi i/N}$ 所得之结果就是多项式 h 在单位 N 次根之值；

⑤计算序列 $(f(\omega^0)\,g(\omega^0),f(\omega^1)\,g(\omega^1),\cdots,f(\omega^{N-1})\,g(\omega^{N-1}))$ 的逆 FFT,所得之序列就是多项式 h 的系数。按此方法,给定下述多项式对,试计算其乘积 $h(x)$：

(a) $f(x)=3x-2,g(x)=4x+1$

(b) $f(x)=2x^3-4x^2+5x-1,g(x)=x^3+2x^2+3x+2$

11.4　根据离散傅氏变换蝶式计算规则,参照图 11.2 画出 $n=16$ 的离散傅氏变换蝶式计算流图。

11.5　参照算法 11.1,设计一个单处理机上时间为 $O(n\log n)$ 的离散傅氏逆变换算法；并以 $n=8$ 为例,画出其逆变换蝶式计算流图。

11.6　参照图 11.8,画出 4 个处理器立方网络上 16 点 FFT 的计算流图。

11.7　Cormen 曾给了另一种形式的 FFT 递归算法 11.7：

(a) 试分析此算法的执行过程；

(b) 它和算法 11.2 有何区别？

(c) 按此算法画出 $n=8$ 的 FFT 蝶式计算流图。

算法 11.7　SISD 上 Cormen 计算 FFT 算法

输入：$a_0,a_1,\cdots,a_{n-1}$

输出：$b_0,b_1,\cdots,b_{n-1}$

Begin

if $n=1$ then return a

else

(1)　$\omega=e^{2\pi i/n}$

(2)　$z=1$

(3)　$a^{[0]}=(a_0,a_2,\cdots,a_{n-2})$

(4)　$a^{[1]}=(a_1,a_3,\cdots,a_{n-1})$

(5)　$b^{[0]}=\text{RECURSIVEFFT}(a^{[0]})$

(6)　$b^{[1]}=\text{RECURSIVEFFT}(a^{[1]})$

(7)　for $k=0$ to $\frac{n}{2}-1$ do

(i)　$b_k=b_k^{[0]}+zb_k^{[1]}$

(ii)　$b_k+\frac{n}{2}=b_k^{[0]}-zb_k^{[1]}$

```
            (ii)  z = z·ω
        endfor
    (3) return b
    endif
End
```

11.8　以 $n=16, p=4$ 为例，逐步写出算法 11.6 的执行过程。

11.9　根据算法 11.2，逐步计算 $n=8$ 的 FFT，并画出其蝶式计算流图。

11.10　令 $n=8=2^k$，在蝶式网络上，按照 $\exp(r, i) = j(0 \le i \le n-1, 0 \le r \le k)$ 的计算方法，试计算分布在蝶形网络中的 8 点 FFT 的系数矩阵元素 ω^j。

第四篇　并行程序设计

第十二章　并行程序设计基础
第十三章　共享存储系统并行编程
第十四章　分布存储系统并行编程
第十五章　并行程序设计环境与工具

这一篇主要研究并行程序设计，包括并行程序设计基础（第十二章），共享存储系统并行编程（第十三章），分布存储系统并行编程（第十四章）和并行程序设计环境与工具（第十五章）。这一篇内容足以构成单独的一门课程，但是作为并行计算的一大部分内容之一，只能从实用的角度出发，围绕着如何进行并行编程来开展讨论。欲深入学习并行程序设计的读者可参阅丛书之四《并行算法实践》[194]。

本篇第十二章主要讨论并行程序的设计方法和编程风范以及并行程序设计模型，同时对并行程序中的进程和线程等基本概念以及相关的同步机制和通信操作也作了介绍。第十三章主要介绍共享存储系统并行编程，其中简单地介绍了 X3H5 和 POSIX，然后重点讨论了 OpenMP 编程标准。第十四章着重讨论了分布存储系统的消息传递和数据并行编程，介绍了 MPI、PVM 和 HPF 三种主要标准及其一些编程实例。第十五章先从一般的软件工具环境讲起，然后重点介绍并行程序设计语言并行编译器以及并行程序的调试、并行程序的性能分析以及并行程序可视化设计环境与工具。

第十二章　并行程序设计基础

并行程序设计的内容十分丰富。本章首先介绍一下并行程序设计基础，主要包括并行程序设计方法和并行编程风范的一般介绍，然后着重讨论并行程序设计模型（隐式并行模型、数据并行模型、消息传递模型和共享变量模型）。因为并行程序设计还涉及到操作系统中的有关知识，如进程、线程、同步与通信等，所以本章还插入了这些内容的介绍，熟悉它们的读者可以跳过它。

12.1 并行程序设计概述

目前并行程序设计的状况是:①并行软件的发展落后于并行硬件;②和串行系统的应用软件相比,现今的并行系统的应用软件甚少且不成熟;③并行软件的缺乏是发展并行计算的主要障碍;④不幸的是,这种状态似乎仍在继续着。究其原因是并行程序设计远比串行程序设计复杂:①并行程序设计不但包含了串行程序设计,而且还包含了更多的富有挑战性的问题;②串行程序设计仅有一个普遍被接受的冯·诺依曼计算模型,而并行计算模型虽有好多,但没有一个可被共同认可的像冯·诺依曼那样的优秀模型;③并行程序设计对环境工具(如编译、查错等)的要求远比串行程序设计先进得多;④串行程序设计比较适合于自然习惯,且人们在过去积累了大量的编程知识、经验和宝贵的软件财富。下面我们从应用的观点,来比较一下串行程序设计和并行程序设计。

12.1.1 串行程序设计与并行程序设计

串行程序设计(Sequential Programming) 对于所希望的应用,很多串行源代码均是有的,用户只需要在目标机上重新编译运行一下即可;即使对于待开发的应用,用户也能很容易找到适合其目的的现有算法,纵然找不到现有的算法,用户也可借助于某些程序设计工具来解决。

串行程序设计的优点是:①长期以来,已建立了很多算法范例,它们可以指导用户设计算法;②程序设计是建立在惟一的、普遍使用的、适用于各种程序设计语言的冯·诺依曼编程模型之上的;③虽然已有很多串行语言,但形成标准的只有适合科学计算的 Fortran 语言,商用的 Cobol 语言和通用的 C 语言等为数不多的几种;④串行程序设计工具是通用的和稳定的:例如 C 语言能支持所有的算法范例和运行在任何串行计算机上,而且屡经数代这些工具依然不变。C 语言发明已经很多年至今变化甚微;冯·诺依曼模型已经历时 50 多年并无本质变化。特别是这些工具都已多年经不同的应用、在不同的计算机平台上屡经测试而变得成熟,人们已经学会在使用中如何将其扬长避短。

并行程序设计(Parallel Programming) 对于所希望的应用,很多并行代码似乎是不存在的;即使有,也常不能用于用户的并行机上,因为并行代码原来都是为不同的并行结构(如 SMP、MPP 等)而写的。

并行程序设计的问题是:①并行算法范例至今尚不能被很好地理解和广泛地

接受;②并行程序的设计是建立在不同的计算模型上的,而它们没有谁能像冯·诺依曼模型那样被普遍的接受和认可;③绝大部分被使用的并行程序设计语言都是Fortran和C的推广,它们都不能充分地表达不同并行结构的特点,既不成熟也不通用;④并行程序设计工具依赖于具体的并行结构和计算机代的更迭,既不通用也不稳定,在某个并行平台上开发的并行程序,很难移植到别的或将来的并行机上。

有关串行程序设计和并行程序设计的比较可概括于表12.1中。

表12.1 串行程序设计和并行程序设计比较一览表

应用领域	科学和工程计算,数据处理,商务应用等	
	串行程序设计	并行程序设计
算法 范例	分而治之,分枝限界, 动态规划,回溯,贪心	计算交互、工作池、异步迭代、 流水线、主-从,细胞自动机
编程 模型	冯·诺依曼模型	隐式并行、数据并行、 共享变量、消息传递
编程 语言	Fortran,C,Cobol,4GL	KAP、Fortran90、HPF、X3H5、 OpenMP、PVM、MPI
体系 结构	不同类型的单处理机	共享内存(PVP、SMP、DSM) 数据并行(SIMD) 消息传递(MPP、Clusters)

尽管并行程序设计问题很多,但也有不少进展:①已经开发了很多并行算法,虽然它们大多基于理想的PRAM模型,但有些已被实际采用;②少量的并行算法范例已出现,并且正逐渐被接受;③编程类型渐趋汇聚于两类:用于PVP、SMP和DSM的共享变量的单地址空间模型和用于MPP和机群的消息传递的多地址空间模型,SIMD模型已退出主流,但对专用领域(如信号、图像处理,多媒体处理等)仍是有用的;④并行编程模型渐趋汇聚于三类标准模型:数据并行(如HPF),消息传递(如MPI和PVM)和共享变量(如OpenMP)。

现在人们希望高性能的并行机应是具有单一系统映像的巨大的工作站,使得很多用户都能利用增强的处理能力和存储容量来运行多个串行作业,这就是所谓的串行程序并行系统SPPS(Serial Program Parallel System)。

12.1.2 并行程序设计环境与工具

从用户的观点,一个典型的并行处理过程可简述如下:首先开发求解一个应用问题的具体算法;用户或程序员在并行计算模型上用高级语言(串行或并行)编程

实现算法,然后编译器将源代码转换成可在并行平台上执行的目标代码;最后借助于OS和硬件平台运行程序。其中,从用户编程到编译源代码和链接(包括源到源转换预处理器)的这一过程统称之为并行程序设计,其间任何程序设计语言都有一个运行支持系统,它就是一组子例程(包括用户代码开始执行的初始化,结束清场,数据对象的分布与再分布等);部分运行支持也可由库函数(Library Function)提供,它就是一组按照某些规则经过编译后的经常使用的子例程。库可以与语言相关(如C语言的"stdio"库),也可与OS相关(如线程库),也可与语言和平台无关(如MPI库),库函数在程序运行前或在运行时动态地链向用户代码。

编程环境与工具 编程环境,简称为环境(Environment),系由硬件平台、支撑语言、操作系统、软件工具和应用软件包等组成。编程工具(Programming Tools)泛指用于帮助用户开发应用问题的任何硬件和一组软件实用程序。工具通常不涉及到OS和程序设计语言,主要包括作业管理工具(如网络排队系统NQS:Network Queueing System和负载共用设施LSF:Load Sharing Facility),查错工具和性能工具等。所谓集成工具(Integrated Tool),在通常意义下系指编辑器、查错器、性能监视器、程序可视化工具等。

12.1.3 并行程序设计方法

当在实际的并行机上设计并行程序时,绝大部分均是采用扩展Fortran和C语言的办法。目前有三种扩展的办法:①库函数法:除了串行语言所包含的库函数外,一组新的支持并行性和交互操作的库函数(如MPI消息传递库和POSIX Pthreads多线程库)引入到并行程序设计中;②新语言结构法:采用某些新的语言结构来帮助并行程序设计以支持并行性和交互操作(如Fortran 90中的聚集数组操作);③编译制导法(Compiler Directives):程序设计语言保持不变,但是将称之为编译制导(Pragmas)的格式注释引入到并行程序中。

上述三种编程方法的风范示于图12.1中。对于图12.1(a)所示的一个简单代码段,图(b)使用库函数的方法实现之,其中两个循环由p个进程并行执行,两个库函数my_process_id()和number_of_processes()开拓并行性,barrier()函数确保第一个循环执行后所有进程被同步,这样第二个循环能使用第一个循环更新后的数组A的正确值;图(c)展示了使用新语言结构执行并行操作,Fortran 90数组赋值结构语句执行N个元素相乘,然后以一个赋值语句进行赋值,两个数组赋值之间无需显式同步,因为Fortran 90语句是松散同步的,即在下一语句开始执行之前,一条赋值语句的所有操作均已完成;图(d)例示了编译制导法实现并行操作,并行pragmas指示下面语句均要并行执行,SGI的pfor指使系统并行执行下一循

环,同步 pragma 产生路障同步。

```
for (i=0;i<N;i++) A[i]=b[i]*b[i+1];
for (i=0;i<N;i++) C[i]=A[i]+A[i+1];
```

(a) 顺序化码段

```
id=my_process_id();
p=number_of_processes();
for (i=id;i<N;i=i+p) A[i]=b[i]*b[i+1];
barrier();
for (i=id;i<N;i=i+p) C[i]=A[i]+A[i+1];
```

(b) 使用库函数的等效并行代码

```
my_process_id,number_of_processes(),and barrier()
A(0:N-1)=b(0:N-1)*b(1:N)
c=A(0:N-1)+A(1:N)
```

(c) 使用数组操作 Fortran 90 等效代码

```
#pragma parallel
#pragma shared(A,b,C)
#pragma local(i)
{
    #pragma pfor iterate(i=0;N;1)
        for(i=0;i<N;i++) A[i]=b[i]*b[i+1];
    #pragma synchronize
    #pragma pfor iterate(i=0;N;1)
        for(i=0;i<N;i++) C[i]=A[i]+A[i+1]
}
```

(d) SGI Powerc 使用 pragmas 等效代码

图 12.1 示例三种并行化方法

三种并行编程方法的优缺点可概括于表 12.2 中。

表 12.2 三种并行编程方法比较一览表

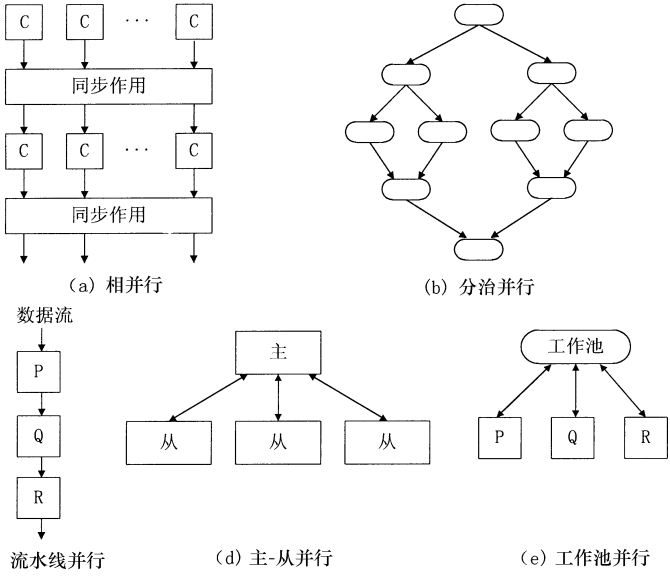
方 法	优 点	缺 点	例 子
库函数	易于实现,不需要新的编译器	没有编译检查、分析和优化	MPI、PVM、Cray Craft
新结构	允许编译检查、分析和优化	难以实现,需要一个新的编译器	Fortran90、Cray Craft
制 导	是库函数和新结构两者优缺点的折衷,其显著优点是“普通串行程序+格式注释”,可在任何串行平台上编译		HPF、Cray Craft

12.1.4　并行编程风范

所谓并行编程风范(Parallel Programming Paradigms)是指构造运行在并行机上的并行算法的方法。目前已有好几种方法,它们既可单独使用,也可组合使用。参照图12.2,现分述如下:

相并行(Phase Parallel)　如图12.2(a)所示,一个并行程序系由一些超级步(也称为相)组成:每个超级步内,各个进程执行独自的计算,然后继之以交互作用(包括通信同步等)。可见相并行和第四章中所讲的BSP模型是十分相似的,这种相并行也称之为松散同步并行,其优点是方便查错和性能分析,其缺点是计算和交互作用不能重叠且维持负载平衡是较困难的。

分治并行(Divide and Conquer Parallel)　如图12.2(b)所示,一个父进程将其工作负载分成一些小的负载并将它们指派给一些子进程,这些子进程并行地完成各自的计算,其计算结果由父进程归并之。这种分开和合并的过程很自然地导致递归,其缺点是难以达到负载平衡。



(a) 相并行　(b) 分治并行　(c) 流水线并行　(d) 主-从并行　(e) 工作池并行

图12.2　五种并行编程风范

流水线并行 (Pipeline Parallel)　如图 12.2(c) 所示,一些进程形成流水线作业法,诸进程在流水线的不同地段同时重叠地执行操作以达到整体并行的效果。

主－从并行 (Master_Slave Parallel)　如图 12.2(d) 所示,这种并行也称之为放牧式并行。一个主进程执行并行程序的串行部分并生成一些可同时并行执行计算的子进程;当某一子进程完成计算后就报告给主进程,主进程再分配新的任务给它。整个的协调工作由主进程完成,其缺点是主进程易成为系统瓶颈。

工作池并行 (Work Pool Parallel)　如图 12.2(e) 所示,开始时池中只有一件工作,任何空间的进程均可以从池中取出它并执行之,执行中可能产生一个或多个新工作并把它们放回池中,以供别的空间进程受领之,当池中变空时则并行程序结束。一个工作池实际上就是一个全局数据结构,可以是无序集合、队列、或优先队列等。这种并行的优点是易于达到负载平衡,因为工作负载是动态地分配给空间进程的;然而实现工作池能被多个进程有效访问并非易事,特别是消息传递模式中,也正是因为如此,工作池并行常使用在共享变量的模式中。

*12.2 进　　程

进程 (Process) 是并行程序中的基本计算单位,它对应于由相应代码段所执行的操作。程序就是一些进程的集合。而并行程序设计的基本问题就是围绕着进程的说明、生成、中止、激活、迁移、结束和同步等。在一台并行机上,用户应用程序都是作为进程 (任务、线程) 被执行的,所以有必要先复习一下进程的有关问题。

12.2.1 进程的基本概念

基本定义　进程 P=(P,C,D,S) 是一个 4 元组,其中 P 是程序 (即代码)、C 是控制状态、D 是数据状态、S 是进程 P 之状态。任何进程总和程序连系在一起。程序使用两组变量:数据变量保持数值,由程序员说明之;控制变量保持控制流信息,不需显式说明。在任何时刻 t,程序的状态由所有成对的 (程序变量,值) 之集合所定义。进程是动态变化的,在任何时候它会处于某一状态。如图 12.3 所示,开始时进程处于非存在 (Nonexistent) 状态;它可由其父进程生成就绪 (Ready) 状态,然后被调度进入运行 (Running) 状态;当无法继续执行时,它就转入中止 (Suspended) 状态,以后待机被唤醒可再次进入就绪状态;进程运行中也可被别人抢先或因超时而停止运行,或者就正常结束,或者因异常退出而被废弃。

上述给出的进程概念对只是使用并行语言的用户是合适的,如要具体实现它,

尚须涉及到进程执行模式、进程的现场活动、进程描述符和进程控制等诸多方面。

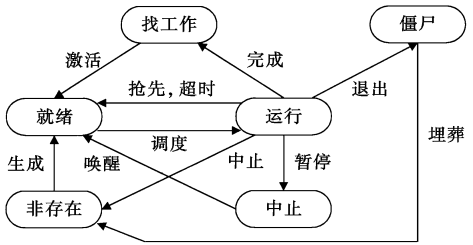


图 12.3　进程状态转换图

执行模式　众所周知,操作系统通常包含核(Kernel)、壳(Shell)和一组实用程序(Utilities)。其中,核直接管理系统资源,处理例外和控制进程;壳称之为命令解释器,是用户和 OS 的界面;实用程序是附加的 OS 软件,提供经常使用的诸如编译器、编辑器和调试器等功能。一台计算机执行程序时提供两种执行模式:核模式和用户模式。OS 中核执行在核模式(Kernel Mode),核进程在核模式下执行,这些进程由核生成以帮助管理系统资源;OS 中的其它程序作为进程执行在用户模式(User Mode),这样的进程称为用户进程。进程的执行模式可以在核与用户模式之间来回转换。机器开始在核模式,初始化系统和生成一些核进程后,核最终将控制传给壳(它是用户进程),它能生成一些附加的用户进程。用户进程执行中也可将执行模式切换到核模式,核完成了所请求的服务后,又能将执行模式返回到用户模式。

活动现场　一个进程的活动现场,或称前后关系(Context)是程序状态的一部分,系保留在处理器的寄存器中。现场切换(Context Switch)就是保留现行进程现场,加载新的进程现场的活动过程。当进程执行模式变化时就需要施行现场切换。在切换前,用户进程的现场必须被保存在主存中,当中断处理完毕后,核就恢复用户进程现场,并将控制返回给用户进程而继续执行之。

进程描述符　进程的附加信息以某些数据结构保存在核空间中,其最重要者是进程描述符(Process Descriptor),它包含了核管理进程的如下信息:①进程凭证:如进程标识符、父进程标识符、用户标识符和组标识符等;②进程状态:如就绪、运行和中止等;③进程现场:保持执行模式切换时的进程现场;④存储映射:如各存储段的大小与访问权限、段指针与页表等;⑤各进程信息:如打开文件、接收信号等;⑥全局数据结构:由核管理的所有进程的队列指针和表;⑦进程的控制与管理信息(见下)。

OS 将进程描述符分为两部分:其一是各进程(Per_Process)部分,它可从进程用户空间中交换出去;其二是在所有时间均必须驻留在主存中的其它部分。在有

的实现中,各进程部分被分配在用户空间,但只能由核访问。

进程控制(Process Control) 进程控制是指由核对诸进程进行管理,其主要功能有:①进程管理(Process Management):由核使用进程描述符去生成、中止、唤起和结束进程,管理全局数据结构和所有的各个进程的数据结构;②进程保护(Process Protection):由核执行全面检查以确保进程只能访问规定的资源。在进程描述符中含有各种进程的权限信息。通常用户进程只能访问其用户空间,而不能访问核空间或别的进程用户空间;③进程调度(Process Scheduling):由称之为调度器的核程序将处理器、存储器和I/O等资源指派给进程。调度器应是公平、有效的,它可根据进程描述符中所提供的进程优先信息进行调度。所谓有效的调度是指调度开销低的调度。如图12.4所示,调度有单道程序和多道程序之分。如果资源为单一用户所专用,这样的系统称为单用户单任务(Single_User Single_Tasking)系统,即单道程序设计(Uniprogramming)系统(如Microsoft DOS),其变体方案是允许单用户的多进程同时使用资源,此即所谓的单用户多任务(Single_User Multi_Tasking)系统(如Microsoft Windows);如果允许很多用户同时分享系统资源,则这样的系统称为多用户多任务(Multi_User Multi_Tasking)系统,即多道程序设计(Multiprogramming)系统(如Unix和Windows NT)。资源的共享形式,可取独占(Dedicated)方式(资源不共享)、批处理(Batching)方式(一旦调度运行就执行到完成)、分时(Time_Sharing)方式(轮流交替使用资源)和抢先(Preemptive)方式(高优先权的进程可以抢走低优先度的处理器)。

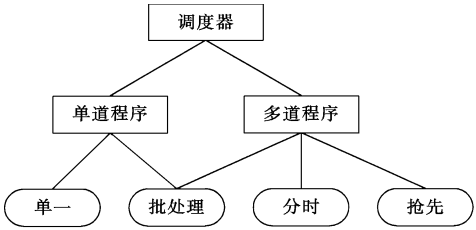


图12.4 调度类型

在并行机上有所谓空间共享与时间共享之分:前者是指一个进程指派给一个处理器,或不同的应用指派给不同的处理器集合;后者是指不同应用的多个进程可指派给同一个处理器。同时在并行机上的调度必须考虑到并行进程的交互作用。

进程与线程 传统的OS进程具有分开的地址空间,它使得进程的管理非常费时。例如,当Unix进程执行fork()系统调用生成一个子进程时,就必须为该子进程生成一个新的地址空间。这就意味着必须分配存储器、数据段和父进程描述符必须复制,必须为该子进程设置运行栈等。这种费时的Unix进程被称为重量

级进程(Heavy_Weighted Process)。进程生成与切换的高开销严重影响并行系统的性能,所以重量级进程是不适合于并行机的。为了开拓细粒度的并行性,必须使用轻量级进程(Light_Weighted Process),它的流行名称叫线程(Threads),其与OS进程的主要差别是,在一个重量级OS进程内,可存在多个线程,它们共享相同的进程地址空间。任何线程执行线程生成操作就可生成附加的线程,生成线程比生成重量级进程的开销小得多。用户级线程的生成不涉及到核,但线程的管理(生成、结束和调度等)必须由核或用户级线程库完成。为了统一起见,约定:任务=进程=重量级进程=OS进程;轻量级进程=线程。

12.2.2 进程的并行执行

执行方式 并行程序中的各个进程如果是同构的(Homogeneous),则诸进程可以单程序多数据SPMD(Single_Program_Multiple_Data)方式执行,即各个进程作用在不同的数据上但执行相同的代码;如果并行程序中的各个进程是异构的(Heterogeneous),则它们可以多程序多数据MPMD(Multiple_Program_Multiple_Data)方式执行,即各个进程在不同的数据上执行不同的代码。SPMD程序是单一代码(Single_Code)方式的并行循环(Parallel_Loop)结构,即数据并行(Data_Parallel)结构;MPMD程序是多代码(Multiple_Code)方式的并行块(Parallel_Block)结构。在文献中,SPMD程序也称为数据并行程序,强调开发数据域中的并行性;MPMD程序常是功能并行(Functional_Parallel)、又称之为任务并行(Task_Parallel)或控制并行(Control_Parallel)程序。

并行进程表达 对于MPMD程序,可使用并行块表达方式,其形式描述如下:

parbegin s_1 s_2 … s_n parend

其中s_1 s_2 … s_n是组件进程(Component Processes),它们可同时并行执行,但步伐各异彼此独立,当n个组件进程都结束时则并行块结束,为了协调各进程须执行特殊的互操作。

对于SPMD程序,可使用并行循环表达方式,其形式描述如下:

```
parfor (i=1;i<=n;i++) {process(i)}
```

当使用单一代码方法表达parfor (i=0;i<n;i++) {foo(i)}时,用户只须书写如下的一个程序:

```
pid=my_process_id();
numproc=number_of_processes();
for(i=pid;i<n;i=i+numproc) foo(i);
```

它可被编译成一个可执行程序,然后使用执行命令将其加载到n个节点运行之。

对于具有静态并行性(程序的结构和进程数目可在运行前,即在编译、链接、加载时确定)的程序,可以使用上述并行块和并行循环的方式表达;而对于具有动态并行性(进程可在运行时生成和结束)的程序,则可通过分支/汇合(fork/join)之类的操作表达之。例如,下述程序有三个进程,其中A进程为主进程,它在程序开始执行时自动生成:

```
   进程A                进程B                进程C
begin                begin                begin
  Z:=1;                fork(C);             Y:=foo(Z);
  fork(B);             X:=foo(Z);         end
  T:=foo(3);           join(C);
end                    output(X+Y)
                     end
```

进程A执行fork(B)操作生成新进程B,它与A并行执行;进程B依次又生成另一进程C,因为进程B中的输出语句需要进程C计算的Y值,所以在输出语句之前插入一个join(C)语句,它强使B等待C结束才能执行输出语句。

进程分配 任何并行程序总要在某些数据上执行某些计算(又叫负载,Workload)。所谓分配(Allocation)就是将数据和负载划分(Partitioning)给诸进程,然后将这些进程映射(Mapping)给处理器(节点)。划分的重要任务就是去选择合适的并行度DOP(Degree Of Parallelism)和粒度(Granularity)。DOP可定义为能同时并行执行的组件进程数;粒度可定义为进程的尺寸,即一个组件进程内总的运算数。并行度和粒度常是互易的:增加粒度尺寸趋向减少并行度;而减少粒度尺寸则趋向增加并行度。此外,在实际并行机中,并行度、通信开销和同步之间通常具有一定的比例关系:增加并行度常导致增加开销;而减少并行度则趋向减少开销。

分配有隐式和显式之分:显式分配,用户必须明确指定如何分配数据和负载;隐式分配,此任务由编译和运行系统完成。在SMP中,共同的方法是将数据驻留在中央共享存储器中,使得所有进程均可访问,负载被静态或动态地分发给诸进程,当一进程分得一批负载时,它就从共享存储器中取得所需数据。在分布存储系统中,流行的方法是属主-计算(Owner_Compute)规则:即数据首先分配给进程,当变量 x 分配给进程P时,P就称之为 x 的属主,由P执行与 x 相关的计算。

12.2.3 进程的相互作用

进程间的相互操作表现为三种形式:通信、同步和聚集(Aggregation)。

通信 进程之间传递数据称之为通信。在共享存储的程序中,它可通过读/写

共享变量的方法实现之 在 fork 操作中,子进程和父进程可通过参数传递方法实现通信 在分布存储的程序中,它可使用消息传递的办法进行交换数据。

通信的模式有多种 包括点到点、播送、散播 (Scatter) 、收集 (Gather) 、全交换、归约和前缀和 (即 Scan) 等。有时把除了点到点以外的其它通信统称之为集合 (Collective) 通信。

同步 同步使进程之间相互等待。同步操作一般有三种:① 原子性 (Atomicity) ,是指一系列不可分割的操作,例如

```
parfor (i=1;i<n;i++) {
    atomic {x=x+1;y=y-1}
}
```

其中 atomic 确保每个进程都必须将两个赋值语句作为不可分离的原子操作来执行 ;② 控制同步,使进程的所有操作均必须等待到达某一控制状态。路障 (Barrier) 同步就属控制同步,例如

```
parfor (i=1;i<n;i++) {
    P_i
    barrier
    Q_i
}
```

其中 barrier 确保 P_i 执行完毕达到 barrier 后,它必须等待所有其它进程也到达 barrier 时才能继续。另一种控制同步是临界区 (Critical) ,例如

```
parfor (i=1;i<n;i++) {
    critical {x=x+1;y=y-1}
}
```

临界区操作实际上是一种互斥操作,critical 只允许一次仅一个进程执行两条赋值语句 ;③ 数据同步,使程序执行必须等待到达某一数据状态。锁 (Lock) 、条件临界区 (Conditional Critical Region) 、监视器 (Monitor) 和事件 (Event) 等均属数据同步,例如:

```
parfor (i=1;i<n;i++) {lock ($s);x=x+1;y=y-1;unlock ($s)}
```

其中 lock 同步依赖于信号量 s 之数据状态。

聚集 聚集将并行程序中的各进程所计算之局部结果汇总起来以产生一个完整的结果,它实际上也是进程相互通信的一种形式。求全和、前缀和 (Prefix) 和归约 (Reduction) 等均属聚集操作,例如

```
parfor (i=0;i<n;i++) {
    X[i]=A[i]*B[i];
```

```
        inner_product := aggregate_sum(X[I]);
    }
```

其中 aggregate_sum 归并部分结果 X[0],X[1],…,X[$n-1$] 以产生最终内积 X[0]+X[1]+…+X[$n-1$]。

通信的模式有多种:包括点到点、播送、散播(Scatter)、收集(Gather)、全交换、移位、归约和前缀和(即 SCan)等。有时把除了点到点以外的其他通信统称之为群体(Collective)通信(见习题 12.6)。

12.3 线　　程

近代 OS 中的一个关键发明就是线程(Thread)。线程就是能单独执行的计算实体(Entity),它们是些具有某些必要最小状态的轻量级进程,而最小状态包括进程状态和相关寄存器的内容。本节将简要讨论一下线程的概念、管理与同步。

12.3.1 线程的基本概念

线程(Thread)是控制流线(Thread of Control)的简称,就是被执行的一个指令序列,其基本概念可图示在图 12.5 中。Solaris 的线程分为两级:运行于用户空间的用户线程和运行于核空间的核线程。工作在核模式下的线程也称之为轻量级进程 LWP(Light_Weight Process)。轻量级线程由核施行管理,在用户级,动态链接的线程库(Thread Library)执行线程应用编程和管理用户线程。

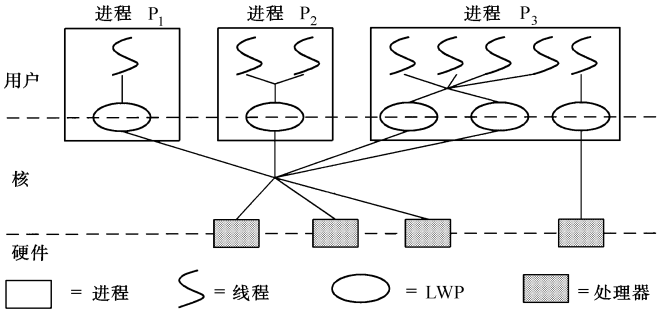


图 12.5　Solaris 操作系统中的多线程

每个进程可有一个或多个线程,进程可以被分配给一个或多个 LWP,分配给进程的 LWP 数目称为进程的并发度。线程库调度进程线运行于 LWP 池上,就像

核调度 LWP 运行于物理处理器池上一样。线程从进程外是看不到的,进程中线程的号和身份,核是不知道的,核不能管理用户级线程,只能管理 LWP。

线程就像进程一样,能执行任何应用代码和系统调用。一个进程的诸线程可共享进程的地址空间,包括它的代码、它的大多数数据和大多数进程描述符信息。各个线程可彼此独立、异步地执行,当一个线程被阻塞时,其它线程可照旧执行,一个进程的多个线程可在多个处理器上同时执行。

因为线程在用户空间中由线程库执行,且进程中的线程勿须分开的地址空间,所以线程操作执行代价比进程的低。又由于线程操作(如生成、结束、同步等)均是在同一用户空间中完成,勿须进入核状态,所以因复制、现场切换、边界交叉保护等进程操作而引起的开销亦将大大降低。

12.3.2 线程的管理

线程生成(Thread Creation) 生成线程比生成进程要简单得多,因为无需生成新的地址空间,只需保存该线程所需的一些状态信息:如线程 ID、处理器状态(程序计数器、堆栈指针)、线程专用栈、线程优先度、线程局部存储区等。这些信息由线程库以称之为线程结构的数据结构形式来生成和维护。

进程中的变量由进程的所有线程共享。因此,如果某一线程修改了一变量,其后果对进程的所有线程均可见。每个线程也有一局部线程存储区,用以保存该特定线程的私有变量,这不能由其它线程直接访问。

线程调度(Thread Scheduling) 线程库提供一个线程调度器去调度 LWP 池中线程的执行。如图 12.6 所示,每一线程在任何时刻均处于某一状态。一个新生成的线程开始在可运行状态,它等待一个空闲的 LWP 到来,然后库调度它运行在 LWP 上,从而进入了活动状态;一个活动的线程,可被阻塞而进入睡眠状态;当阻塞条件消除时,它又被唤醒而进入可运行状态;当线程支持抢先时,一个优先级高的线程可从活动状态中抢走它的 LWP;程序员可以中止一个线程使其进入停止状

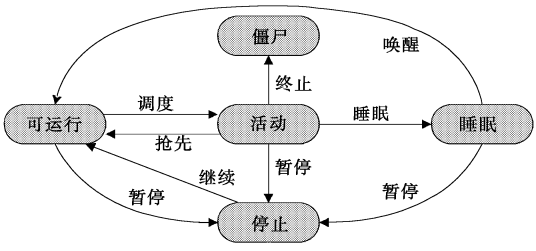


图 12.6 线程库控制的线程状态转换图

态。最后,当一个线程结束时,它就进入了"僵尸"(Zombie)状态。

已如上述,核不能管理用户级线程,只能管理 LWP。图 12.7 示出了由核调度 LWP 的状态转换图:一个运行的 LWP,当被抢先或超时时就转入可运行状态;当执行阻塞系统调用时,运行的 LWP 被阻塞,强使 LWP 等待某一条件满足;当执行 LWP 库函数调用时,LWP 被中止而进入中止状态,而执行继续库函数调用时,它又能返回至先前状态;当一个线程结束,或不再有可运行的线程时,LWP 就进入空闲状态,它也可被再次唤醒而进入可运行状态。

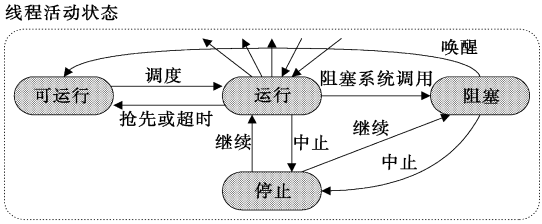


图 12.7 核控制的 LWP 状态转换图

12.3.3 线程的同步

Solaris 线程库执行两种类型的同步对象:局部进程(Process Local)和共享进程(Process Shared)。一个局部进程同步对象,可由同一进程中的线程所访问,它用来同步相同进程中的诸线程;一个共享进程同步对象,可由多个进程所访问,它用来同步不同进程中的诸线程。

分布式计算环境提供两种类型的线程同步:互斥和条件变量。互斥用来防止多个线程在同一时间访问相同的资源;条件变量常与互斥联合使用。典型的情况是,当一线程需要某一资源时,它就使用互斥排斥其它线程的访问;如果资源无效,线程就根据条件变量等待之,稍后条件满足时就可再启动。

*12.4 同 步

同步这一术语,在计算机的很多课程(例如系统结构、操作系统、数据库、并行处理和程序设计语言等)中经常出现,其意义和用法也不尽相同,且其实现方法亦颇多,所以有必要专辟一节深入学习之。

一般而言,同步操作,如图 12.8 所示,可分为三类:原子操作,数据同步和控制

同步。所有在共享存储的机器 (PVP、SMP、DSM 等) 上的同步通常使用锁原语实现 在分布存储的机器 (MPP 和机群等) 上的同步使用消息传递原语实现。

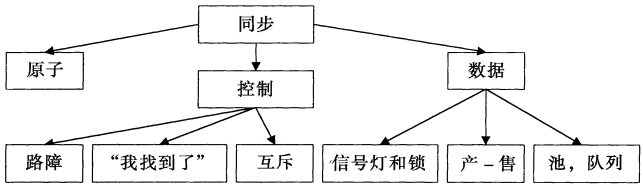


图 12.8 同步的不同类型

12.4.1 原子与互斥

在同步操作中,原子 (Atomicity) 和互斥 (Mutual Exclustion) 是两个相关但不同的概念。互斥是指在同一时间、同一地方不能有两件事情存在;原子性是指一个操作序列不能分开执行。临界区 (Critical Section) 就是按原子操作的一段代码。在临界区内必须使用某种机构确保代码执行的原子性 (即操作的相互排斥)。一个互斥问题应满足如下性质:①独占,即一次只允许一个进程在临界区中执行;②前进,即程序必须无死锁或活锁 (Livelock);③“得食”,即一个试图执行在临界区中的进程最终必将成功而不会被饿死。一个原子性问题应满足如下性质:①有限的,即原子操作从开始到结束的时间应是有限的;②不可分的,即原子操作必须作为一个单一的不可分开的执行步,只有原子操作的最终结果是可见的,而任何中间结果均是不可见的;③可串行化的,即一些并行执行的原子操作之结果,尤如以任意顺序串行执行的结果。原子操作和互斥操作的主要不同是:①原子操作是有限的,但有限性并不是互斥操作的固有性质,例如临界区中可能包含一个由编译或运行支持系统无法检测出的无限循环;②互斥强使操作串行执行,但原子性允许并行执行;③互斥操作可确保不可分割性,但不可分割性也可由别的、非互斥方法实现。原子性传统上用互斥方法实现,而锁是支持互斥的主流技术,所以锁也就是原子操作。

在并行计算中,虽然仍使用互斥结构 (如锁和临界区) 作为原子操作的主要支持,但是互斥不是惟一的或最好的达到并行计算应用原子性的方法。有效的、安全的实现并行计算中原子性的方法正在研究中。然而我们必须打破这种观念,即原子性只能够被模拟作一个临界区问题,而临界区要求互斥,因此也要求锁。

12.4.2 高级同步结构

现今共享存储的多处理器系统的并行程序设计环境,提供了四种类型的同步

原语 事件、路障、锁/信号灯和临界区。事件(Event)操作用于实现产—销(Producer_Consumer)同步 路障(Barrier)用在 Barrier 同步中;锁和临界区主要用于实现互斥形式的原子性。

信号灯和锁 信号灯 S是一个非负整数变量,对其能进行两个原子操作 P(S)和 V(S):①P(S)操作用于延迟一个进程直至 S变成大于零,然后 S减一;②V(S)操作将 S增一。二元信号灯(Binary Semaphore),S取值为真或假,也称之为锁。相应地,对于锁 S其 P(S)和 V(S)常写作 lock(S)和 unlock(S)。锁的普通用法就是通过互斥将临界区转换成原子操作。

锁的副作用 锁的主要优点是它已由大多数多处理器支持之,并且已研究得相当深入。锁是一种非常灵活的机制,几乎能实现任何同步。然而互斥锁技术用于实现原子时具有某些严重的缺点,从而导致如下一些问题:①非结构性:锁不是一种结构化的结构,容易用错它,如果 lock/unlock 语句漏掉或冗余,则编译器不能查出它;②重叠说明(Overspecification):锁不是用户所真正想要的,它只是达到原子性的一种方法。锁损害了程序的可移植性,且使代码难于理解;③状态相关:锁引入了信号灯 S及使用条件原子操作 lock(S),一个进程能否穿过 lock(S)依赖于信号灯变量 S之值,一般而言,像这样的与状态有关的数据是难于理解的;④顺序执行:对于有些事务处理操作,即使可并行访问,但由于使用锁互斥,它们只能一次执行一个,同样这种顺序执行也不是用户想要的;⑤锁开销:顺序执行 lock(S)和 unlock(S)也存在着附加的开销,而且当 n个进程每个都执行 lock(S)操作时,它们中至多一个能成功,其余者均必须重复访问 S而再试;⑥优先倒置(Priority Inversion):当一个保持了高优先进程所需的锁的低优先进程被抢先时,高优先进程并不能前进,因为它被锁锁住了;⑦护送阻塞(Convoying Blocking):当一个保持锁的进程因缺页或超时被中断时,其它的进程因等待锁不能前进;⑧死锁(Deadlock):假定两进程 P与 Q,欲进行 X与 Y操作:当进程 P已为 X保持了一把锁并想为 Y申请一把锁;而进程 Q已为 Y保持了一把锁并想为 X申请一把锁时,此时没有任何进程在其得到锁之前释放一把锁,结果谁也得不到所要求的锁。

临界区 OS所使用的临界区,其语法结构如下:

```
critical_region resource
 {                          /*进入点*/
     S1;S2;…;Sn;            /*临界区*/
 }                          /*退出点*/
```

其中,resource 代表一组共享变量。所有共享相同资源的临界区必须互斥执行。并行程序设计所使用的临界区作了两点修改:①resource 部分不是真正有用的,所以被略去。使用在真正多处理机中的临界区,其语法结构及其等效锁代码表

示如下：

critical_region
{
　　S_1;S_2;…S_n;
}
	等效锁代码
	lock(S)
	　　S_1;S_2;…S_n;
	unlock(S)

② 多处理机中的临界区变成了锁结构方式，系统自动说明和初始化一个隐含的信号灯 S 并产生正确的 lock/unlock 语句。

临界区比锁有很多优点，它是结构化的、与状态无关的，因而易于使用。临界区只是一段被互斥执行的代码，并非必须使用锁。

12.4.3　低级同步原语

很多多处理机的硬件都能确保单独读/写初等变量的操作的原子性，绝大多数多处理机硬件都提供了某些原子性指令，它们可对初等变量执行单一的读－修改－写操作。本节讨论三种低级同步结构：Test & Set、Compare & Swap、Fetch & Add。

测试并设置（Test & Set）　Test & Set(S,temp)是一个原子操作指令，它将共享变量 S 读入局部变量 temp，然后将 S 置为 1，其主要用途是执行锁功能。兹示例如下：

```
while(S) ;              /*这三行执行 lock(S) 操作*/
Test & Set(S,temp) ;
while(temp) Test & Set(S,temp) ;
            …           /*临界区*/
S=False;                /*unlock(S) */
```

上例中，使用了 Test_and_Set 操作执行 lock(S)，其中第一个 while 循环检查锁 S 是否已由别的进程释放。由于每次执行 Test & Set 都要写共享变量 S，所以可能导至频繁的存储器访问。

比较并交换（Compare & Swap）　Compare & Swap(S,old,new,flag)也是一条原子操作指令，它将共享变量 S 与局部变量 old 进行比较：若 S 与 old 一致，则 S=new，且 flag=True 以指明 S 被修改；若 S 与 old 不一致，则 old=S，且 flag=False，其主要用途也是执行锁功能，兹示例对照如下：

```
old=balance[x] ;        /*读共享变量*/
do {
```

```
    new=old-100;        /*修改*/
    Compare & Swap(balance[x],old,new,flag); /*写*/
  } while(flag==False);
```

上述操作可以用锁实现如下：

```
lock(S);
  balance[x]=balance[x]-100;/*读_修改_写*/
unlock(S);
```

上述锁功能使读、修改、写这一整个过程是互斥的。使用 Compare & Swap 的优点是临界区的长度减至只一条指令。

取并加(Fetch & Add) Fetch & Add(S,V)也是一条原子指令，它返回共享变量 S 给局部变量 Result，然后将局部值 V 加向 S，其语法结构如下：

```
Fetch & Add(S,V)
    {
    Result=S;
    S=S+V;
    Return Result;
    }
```

该指令不仅简单而且快速，例如上节示例的代码段只用一条 Fetch & Add 指令即可实现：

```
Fetch & Add(balance[x],100);
```

12.5 通　信

进程之间的协同工作必然会产生通信。通信可以通过共享变量和消息传递的办法来达到。通信协议的不同结构如图 12.9 所示，通信库(如 PVM 和 MPI)可以实现在套接字(Socket)之上。在发送端，消息下传至套接字层、TCP/IP(或 UDP/IP)层直到驱动器和网络硬件层；在接收端，以相反次序重复上述过程。套接字可以直接在低级基本通信层 BCL(Base Communication Layer)上实现而旁路掉 TCP/IP；PVM/MPI 也可执行在 BCL 之上而旁路掉 TCP/IP 层。BCL 的主要目的是尽可能多地展示原始硬件性能，而为了评价通信系统的性能，PVM、MPI 和套接字的性能比 BCL 之性能更为重要。

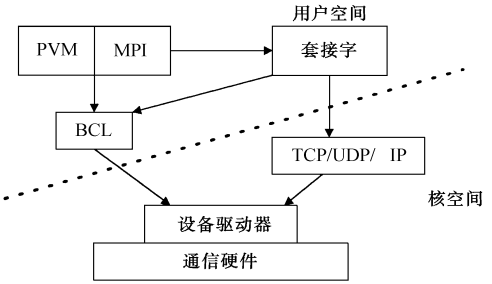


图 12.9　通信协议不同结构

通信对并行计算的性能影响很大,而影响通信性能的主要因素有通信硬件(包括节点存储器、I/O 结构、网络界面和通信网络本身等),通信软件(包括通信协议结构和算法等),所提供的通信服务(包括消息传送、流控、失效处理和保护等)等。

12.5.1　影响通信系统性能的因素

通信硬件　典型的通信硬件结构示于图 12.10。在松散耦合的系统中,网络接口电路 NIC(Network Interface Circuitry)搭接在 I/O 总线(例如 PCI)上。一条发送的消息,从发送节点的存储器出发,经过存储总线、I/O 总线、NIC,最后送至网上,其间可达到的通信带宽受限于该路径上最慢的部件,而实际上的通信瓶颈常在存储带宽而不是网络带宽,因为很多通信方案均涉及到 DMA 访问,待发送的数据在发出去之前首先要复制到 DMA 缓冲器,而这种存储复制的带宽远远小于存储总线的峰值带宽。

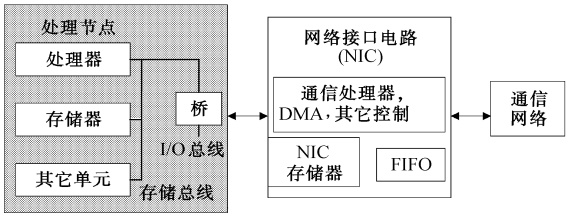


图 12.10　典型的通信硬件结构

通信硬件中 NIC 是个很关键的电路,它应具有 DMA 功能;有自己的通信处理器(协处理器),主要用于初始化 DMA、打包/拆包、保护检查等;应有存放 NIC 代

码和临时缓冲消息的存储器;也还应有缓冲信包的一些 FIFO 等。

通信软件 在现代的机群和 MPP 系统中,软件开销占通信时间的主导地位。如果没有良好的通信软件,即使有非常有效的网络和 NIC,其通信时间也难以明显下降。软件开销主要来源于:①穿越多层协议所需的软件开销;②由于消息通信所涉及的一些存储复制操作所引起的开销;③传输消息时多次跨越保护边界所造成的通信开销(一般至少穿越四次:在发方和收方分别从用户空间进入和离开核空间)。对付第一个问题可以使用简化的通信协议;对付第二个问题可以使用零复制协议(Zero_Copy Protocol),即消息直接从源节点的发送缓冲器到目的节点的接收缓冲器而不缓冲在存储区中;对付第三个问题可以使用用户级技术,即所有的通信均完全执行在用户空间。

通信服务 所提供的通信服务会大大影响通信系统的性能,所期望的服务包括:①可靠传输,一旦消息发出就应确保它能正确无误地去向目的地且信包不会丢失;②流控,消息不应死锁、拥挤和使缓存溢出;③失效处理,包括错误检测、重发、校正等;④有序传输,在任何情况下应保障接收消息顺序的正确性。

12.5.2 低级通信支持

在改进通信性能方面,BCL 起着关键的作用。本节简要讨论三种有代表性的 BCL,即双复制(用户空间协议)、单复制(快速消息)和零复制。

双复制(2_Copy) 最好以 IBM SP 的通信过程(图 12.11)说明之。SP 通信协议要求发送节点处理器首先将数据从发送缓冲器复制到管道缓冲器;然后再从管道缓冲器复制到虚拟开关界面中的输出队列。接收节点处理器以相反的次序执行相同的动作。参照图 12.12,消息层和管道层形成了基本通信库(层)BCL。其中消息层系由一些简单的、非阻塞的、点到点的通信库组成,MPI 和 PVM 等中所有高级消息传递功能均由这些原语实现;管道层维持成对发/收进程之间可靠的、具有流控的、有序的字节流。当源节点处理器执行一条消息层发送操作时,它就将数据从发送缓冲器复制到管道缓冲器;然后消息层再调用管道层代码将数据从管道缓冲器复制到输出队列;源适配器用 DMA 将数据再从输出队列移到适配器,最终

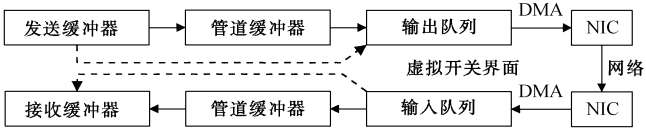


图 12.11 IBM SP 通信中的数据移动

将数据经网络传至目的适配器;目的适配器用 DMA 将进来的数据传至目的节点的输入队列;目的节点处理器执行一条管道层接收操作,它就将数据从输入队列复制到管道缓冲器;管道层调用消息层代码将数据从管道缓冲器复制到最终的接收缓冲器。

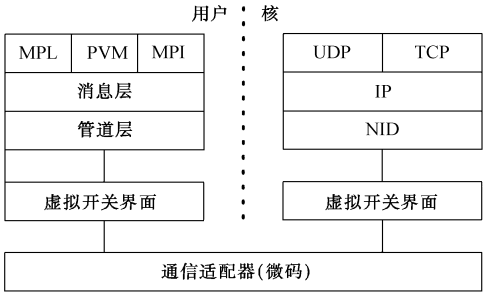


图 12.12 IBM SP 通信软件

单复制(1_Copy) 单复制又称快速消息 FM (Fast Message),主要支持机群和 MPP 上低延迟、高带宽通信,其目的是在消息层提供足够的功能使得较高级原语(如 PVM、MPI 套接字等)所实现的通信性能接近于硬件极限。

FM 数据结构和功能非常类似于主动消息,但它简单,只包含很少的几个通信原语,例如,FM_initialize(),用于初始化;FM_send(dest, *h*,buf,size),用于发送存储器中长消息;FM_send_4(dest, *h*, i_0, i_1, i_3),用于发送寄存器中 4 字消息;FM_extract(),用于抽取接收的消息和调用处理程序。

FM 是一个用户级通信层,一旦初始化后,适配存储器和节点接收队列都暴露在用户级,随后的 FM_send 和 FM_extract 均不必有跨边界保护。当发送节点处理器从缓冲中取出数据并组装成信包后,就直接将其存入适配器的发送队列中,然后信包在 DMA 控制下注入到网络;当信包到达接收节点时,仍由 DMA 控制将其放入适配器接收队列,稍后移入节点接收队列,最后调用处理程序将消息数据移进用户存储空间。

零复制(Zero_Copy) 这种通信机制由 Princeton 的 SHRIMP 项目实现,又叫作虚拟存储映射通信,它能提供真正的零复制通信协议。当节点间进行发送/接收时,节点中的守护程序可以通过网络在发送节点和接收节点之间建立入－出关系;然后发送节点的进程就能将其源缓冲器(用户空间)中的消息,直接发送给另一节点中的目的缓冲器,而不须要穿越核空间中的 DMS 缓冲器。

12.5.3 TCP/IP通信协议组简介

网络上的通信通常由一组协议实现,协议就是一组控制数据格式和数据如何传输的规则。本节主要讨论TCP/IP协议组的性质;经常使用的UDP、TCP和IP协议的基本概念与特点;最后讨论应用编程界面API(Application Programming Interface),即俗称的套接字(Sockets)。

TCP/IP协议组性质　目前两种广泛使用的通信协议是由ISO(International Standard Organization)制定的OSI(Open System Interconnect)和由IAB(Internet Activities Board)制定的TCP/IP,其对应关系示于图12.13中。TCP/IP(Transmission Control Protocol/Internet Protocol)协议组涉及到三层:应用层(包括FTP、TELNET、SMTP、SNMP、HTTP)、传输层(包括TCP和UDP)和网络层(包括IP)。TCP层附有TCP头形成称之为TCP段的传输层消息;IP层附有IP头形成称之为IP数据报的网络层消息;NAP层附有以太网头形成称之为以太网帧的NAP层消息。应用层协议必须告诉下层协议目的进程的地址,这通常以(IP地址,端口)数偶形式给出,其中IP地址惟一地指定了目的主机,而端口号指明目的进程。

用户数据报协议UDP(User Datagram Protocol)　TCP/IP协议组包括两个传输层协议:TCP和UDP,两者均操作在IP协议之上。其中UDP是个非常简单的传输层协议,消息以数据报的格式组装。UDP是一种非连接协议,它就像邮局发信一样,发送者发出数据并不建立连接,而是将大量数据分割并封装成单独的数据报(信件),然后分开发出去(这和下面将要讨论的TCP不一样,它先要建立连接,然后以连续的数据流而不是单独的数据报形式发送)。UDP协议努力将邮件传送

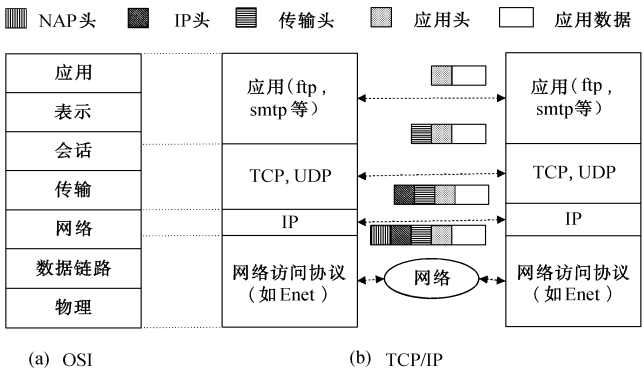


图12.13　OSI和TCP/IP对应关系

好,但并不保证无误传输。一个 UDP 数据报可能遭到损害或丢失,但在接收端可通过检查和查出错误,并甩掉受损的数据报。UDP 对不严格要求确保无误的传输是合适的,否则要使用 TCP 协议。

传输控制协议 TCP (Transmission Control Protocol)　TCP 协议的功能比 UDP 强,它是面向连接的协议,就像打电话一样,通话的双方必须先建立连接,然后才能传输消息。在 TCP 协议中,使用误差检测、应答和重发机制可确保消息的正确顺序和可靠的传输。TCP 中消息传输以段的格式,段头中包括源端口、目的端口、顺序号、应答号、标志、窗口尺寸、应急指针与校验和等。TCP 传输连续消息流,一旦建立起连接,一个接一个的消息就不间断的传输。TCP 在发送端负责将消息流分成 TCP 段,在接收端再负责将段组合成消息。TCP 还负责处理乱序段、重复段以及流控 (其中乱序段和重复段是由段头中的顺序号来处理;窗口尺寸的值用于流控)。在 TCP 中,采用超时重发的办法可保证可靠传输,但也易造成缓冲饱和与网络拥挤,为此 TCP 使用了滑动窗口 (Sliding Window) 流控机制来解决此问题。

互联网协议 IP (Internet Protocol)　IP 的主要功能就是在 Internet 网内 (可由多个 LAN 组成) 将消息从一个主机选路至另一个主机。在 Internet 网上,任何主机均有惟一的 IP 地址,而路由器不同于一般主机,它可有多个 IP 地址,其主要功能是在 Internet 网内选路 IP 数据报。IP 类似于 UDP,它也是非连接的协议,所以传输中也可能出现乱序的或丢失数据板,但此问题留给上一层协议来解决。

当一个主机中的 IP 软件模块收到发送请求时,它就查找本地表以确定目的主机是否在同一 LAN 中,如果不是,数据报就必须发给路由器,由其确定进一步的路由路径,并转发数据报给指定的路由器;然后由其将数据报发往最终的目的主机。除了选路以外,IP 层协议还提供数据报分段和重装配功能以及误差的通告。

套接字 (Socket) 界面　套接字就是为了使用 TCP/IP 协议组的一组数据类型和功能,即 API。它最初由 Berkeley Unix 实现,但现在已经变成了几乎在所有 Unix 系统上和 Microsoft Windows 平台上的标准。一个套接字就是一个通信端口。当两个进程使用套接字通信时,它们各自首先生成一个套接字并指明使用哪种传输协议 (UDP 或 TCP),然后采用读/写它们相应的套接字来进行通信,而套接字软件负责执行真正的通信。下面以简化的域名服务器为例,说明如何使用 UDP 套接字和 TCP 套接字来实现客户发送任意主机名给服务器,服务器查找主机的 IP 地址,然后将其返回给客户。

UDP 实现上述域名服务器程序框架如下:

```
//*用 UDP 实现域名服务器代码段*//
main ()/*服务器代码段*/
```

```
 {
mysocket=socket(AF_INET,SOCK_DGRAM,…);
bind(my socket,…);
recvfrom(mysocket,hostname);
host_IP=Name To IP(hostname);
send to (mysocket,host_IP,…);
}
main(int argc,char *argv[])      /*客户代码段*/
 {
mysocket=socket(AF_INET,SOCK_DGRAM,…);
send to (mysocket,argv[1],…);
recvfrom(mysocket,host_IP,…);
}
```

其中,服务器和客户均首先调用 socket 函数,目的就是生成变量为 mysocket 的套接字;常量 AF_INET 指明互连网域地址(IP 地址)将用作套接字寻址格式,而 SOCK_DGRAM 指明使用 UDP;服务器使用 bind 函数调用将套接字连接至端口号上;当套接字建立后,客户用 send to 函数将 hostname 发给服务器;服务器使用 recvfrom 函数接收消息,并用执行局部函数 Name TOIP 将其翻译成相应的 IP 地址,并用 send to 将 IP 地址返回给客户;客户使用 recvfrom 接收此 IP 地址。

TCP 实现上述域名服务器程序框架如下:

```
//*用 TCP 实现域名服务器代码段*//
main()/*服务器代码段*/
 {
mysocket=socket(AF_INET,SOCK_STREAM,…);
bind(mysocket,…);
listen(mysocket,…);
fp=accept(mysocket,…);
read(fp,hostname,…);
host_IP=Name To IP(hostname);
write(fp,host_IP,…);
close(fp);
}
main(int argc,char * argr[])/*客户代码段*/
```

```
{
mysocket=socket(AF_INET,SOCK_STREAM,…);
connect(mysocket,…);
write(mysocket,argv[1],…);
read(mysocket,host_IP,…);
}
```

其中,socket生成和binding(连接)类似于UDP实现的程序,只是SOCK_DGRAM用SOCK_STREAM代替。与UDP不一样,TCP是面向连接的,服务器要告诉核它准备接收连接,这通过执行listen函数实现;然后通过执行accept函数,服务器等待接收来自客户的连接请求,此函数一直到客户执行connect函数才返回;连接双方的建立尤如普通文件一样,使用了文件指针fp。到此,就像访问一个文件一样,服务器和客户就可利用读/写连接而相互通信,当通信完毕,服务器执行close(fp)关闭连接。

12.6　并行程序设计模型

并行程序设计模型(Parallel Program Model)是一种程序抽象的集合,它给程序员提供了一幅计算机硬件/软件系统透明的简图,程序员利用这些模型就可以为多处理机、多计算机和工作站机群等设计并行程序。

为了以下各节讨论方便和具体,先选择一个样本程序,以其为例进行讨论和相互对照。

12.6.1　计算π样本程序

在此所讲的样本程序,并非普通意义下的"标准样板"程序,它只不过是一个简单、易于实现、且便于理解的一个计算程序罢了。习惯上,作为教材,选择了这样一个大家都不生疏的用数值积分法求π近似值的样本程序。

参照图12.14,用数值积分法计算π时,就是要求出在区间[0,1]内函数曲线$4/(1+x^2)$下的面积,此面积就是π的近似值,为此先将区间[0,1]划分成N个等间隔的子区间,每个子区间的宽度为$1/N$;然后计算出各子区间中点处的函数值;再将各个子区间面积相加就可得出π的近似值。其计算公式如下:

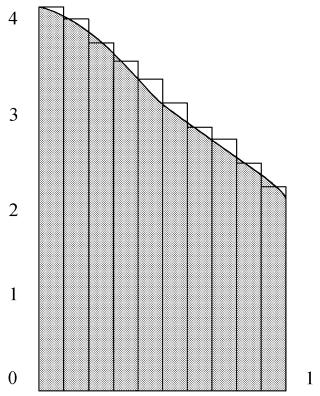


图 12.14 用数值积分法求 π值

$$\pi=\int_0^1 \frac{4}{1+x^2}\mathrm{d}x \approx \sum_{0\le i<N} \frac{4}{1+\left(\frac{i+0.5}{N}\right)^2}\cdot\frac{1}{N}$$

计算此公式的串行 C 代码如下：

```
//*计算 π C 语言编程代码段*//
#define  N  1000000
main(){
    double local,pi=0.0,w;
    long i;
    w=1.0/N;
    for(i=0;i<N;i++){
        local=(i+0.5)*w;
        pi=pi+4.0/(1.0+local*local);
       }
    printf("pi is %f\n",pi*w);
   }/*main()*/
```

其中，*N*是等分间隔数，其值越大越精确，但计算时间越长。

12.6.2 隐式并行模型

隐式并行（Implicit Parallelism）是相对显式并行（Explicit Parallelism）而言的，

后者是指程序的并行性由程序员利用专门的语言结构,编译制导或库函数调用等在源代码中给予明显的指定;前者,程序员未作明确地指定并行性,而让编译器和运行(时)支持系统自动地开拓它,本节先讨论隐式并行程序设计模型;而显式并行程序设计模型,包括数据并行、共享变量和消息传递等将在下几小节分别讨论之。

隐式并行模型中最著名的方法是串行程序的自动并行化(Automatic Parallelization)。编译器执行串行源代码程序的相关分析,然后使用一组转换技术将顺序代码转换成本原并行 Fortran 代码。将串行代码并行化的关键是相关分析,主要包括数据相关和控制相关。如果操作 A 依赖于 B,则 A 必须在 B 之后执行;两个操作可以并行执行只有它们俩是不相关的才行。如果相关的确存在,则必须使用转换技术(也称为重构或优化)将其删除之。目前绝大多数分析和重构技术都集中在 loop 级上,也就是集中开拓 Fortran 的"do loops"和 C 的"for loops"的并行性。常用的三种最重要的转换方法有:专用化(Privatization)、并行归约(Parallel Reduction)和归纳变量(Induction Variables),尽管它们均在个例中有所奏效,但总的讲自动并行化方法效率是非常不高的。

一种有效的方法就是使用用户定向(User Direction)的办法,即用户帮助编译器为并行化提供更多的信息,例如交互并行化(Interactive Parallelization)方法允许编译器(或运行系统)提问题,然后程序员可提供附加信息以指导并行化的进程。更流行的方法是允许程序员在源代码中插入编译制导(Compiler Directives),它在 C 语言中叫 Pragmas。这些格式化的注释语句,携带着有助于编译器能更好地进行重构的附加信息。例如下述代码段:

```
#Pragmas_CNX loop_Parallel
  for (i=0;i<1000;i++) {
    A[i] = foo(B[i],C[i]);
  }
```

其中 loop_Parallel Pragmas 强使编译器并行化随后的 loop,而不管 loop 体是什么。编译器不必分析 loop 体中的相关性,是用户确保并行化后代码的正确性。

即使编译制导能帮助并行化,但并非所有的并行性均能在编译时识别,有些并行性只能在运行时才能被揭示出来。相应地这种技术称为运行时并行化(Run_Time Parallelization)。例如 Stanford 大学开发的 Jade 语言,既能开拓编译时的并行性又能开拓运行时的并行性,程序员从现行的串行程序开始,附加的语言结构用来将顺序程序分解成多个任务和指明每个任务如何访问数据;编译器和运行(时)系统将自动识别和开拓并行性。Jade 不仅能识别出更多的并行性,而且允许自动开拓非规整和动态并行性。

12.6.3 数据并行模型

数据并行(Data Parallel)模型既可以在SIMD计算机上实现,也可以在SPMD计算机上实现,这取决于粒度大小。SIMD程序着重开发指令级细粒度的并行性。SPMD程序着重开发子程序级中粒度的并行性。数据并行程序设计强调的是局部计算和数据选路操作,它比较适合于使用规则网络,模板和多维信号/图像数据集来求解细粒度的应用问题。数据并行操作的同步是在编译时而不是在运行时完成的。硬件同步则是通过控制器执行SIMD程序的锁步操作完成的。在同步的SIMD程序中,所有PE间的通信则直接由硬件控制,除所有PE间操作需锁步外,PE间的数据通信也是以锁步方式进行的。这些同步指令的执行和数据选路操作使得SIMD计算机在开发大型数组,大型网格或网状数据的空间并行性时相当有效。

值得注意的是,一个SIMD程序可以重新编译用于MIMD结构,其思想是开发一个源到源的预编译器来实现程序之间的转换。就此意义而言,数据并行程序设计模型既适用于同步的SIMD计算机,也适用于紧耦合的MIMD计算机。

总之,数据并行模型具有以下特点:①单线程(Single Threading):从程序员的观点,一个数据并行程序只由一个进程执行,具有单一控制线;就控制流而论,一个数据并行程序就像一个顺序程序一样。②并行操作于聚合数据结构上:数据并行程序的一个单步(语句),可指定同时作用在不同数组元素或其它聚合数据结构上的多个操作。③松散同步(Loosely Synchronous):在数据并行程序的每条语句之后,均有一个隐含的同步,这种语句级的同步是松散的(相对于SIMD机器每条指令之后的紧同步而言)。④全局命名空间(Global Naming Space):数据并行程序中的所有变量均驻留在单一地址空间内,所有语句可访问任何变量而只要满足通常的变量作用域规则。⑤隐式相互作用(Implicit Interaction):因为数据并行程序的每条语句之末存在着一个隐含的路障,所以不需要一个显式同步;通信可由变量指派而隐含地完成。⑥隐式数据分配(Implicit Data Allocation):程序员不必要明确地指定如何分配数据,可将改进数据局部性和减少通信的数据分配方法揭示给编译器。

下面举一个计算π值的数据并行程序的例子,以期体会数据并行的基本特点。参照12.6.1节用C语言书写的计算π的串行代码,不难写出如下数据并行程序:

```
//*计算π数据并行编程代码段*//
main ){
    double local[N],temp[N],pi,w;
```

```
    long i,j,t,N=1000000;
A:  w=1.0/N
B:  forall(i=0;i<N;i++) {
    P:local[i] = (i+0.5)*w
    Q:temp[i] =4.0/(1.0+local[i]*local[i]);
     }
C:  pi=sum(temp);
D:  printf("pi is %f\n",pi*w);
}/*main*/
```

上述程序中包含了四个语句 A、B (B 又包含两个子语句 P 和 Q)、C 和 D,这四个语句可由单一进程一个接一个地执行,但每个均同时对多个数据执行相同的操作,其中语句 A 和 D 就是普通的顺序语句;而语句 C 执行 *N* 个 temp 数组元素的归约求和,并将结果赋值给变量 pi;语句 P 并行执行表达式求值并更新所有 *N* 个 local 数据组元素,但所有这 *N* 个元素均必须由语句 P 更新完后语句 Q 才能被执行;类似地,在语句 C 执行归约求和之前,所有 temp 元素均必须由语句 Q 进行赋值。

12.6.4 消息传递模型

在消息传递 (Message Passing) 模型中,驻留在不同处理器节点上的进程可以通过网络传递消息相互通信。消息可以是指令、数据、同步信号或中断信号等。在消息传递并行程序中,用户必须明确地为进程分配数据和负载,它比较适合于开发大粒度的并行性,这些程序是多线程的和异步的,要求显式同步 (如路障等) 以确保正确地执行顺序。然而这些进程均有其分开的地址空间。消息传递模型比数据并行模型灵活,两种广泛使用的标准库 PVM 和 MPI 使消息传递程序大大地增强了可移植性。消息传递程序不仅可执行在共享变量的多处理机上,而且可执行在分布存储的多计算机上。

总之,消息传递模型具有以下特点:①多线程 (Multithreading):消息传递程序系由多个进程组成,每个进程都有其自己的控制线且可执行不同的代码;控制并行 (MPMD) 和数据并行 (SPMD) 均可支持。② 异步并行性 (Asynchronous Parallelism):消息传递程序的诸线程彼此异步地执行,使用诸如路障和阻塞通信的方法来同步各个进程。③分开的地址空间 (Separate Address Space):并行程序的进程驻留在不同地址空间内。一个进程中的数据变量在其它进程是不可见的,因此一个进程不能读/写另一进程中的变量,进程的相互作用通过执行特殊的消息传递

操作实现之。④显式相互作用(Explicit Interaction):程序员必须解决包括数据映射、通信、同步和聚合等相互作用问题;负载分配通常通过属主-计算(Owner_Compute)规则来完成,即进程只在其所拥有的数据上执行计算。⑤显式分配(Explicit Allocation):负载和数据均由用户显式地分配给进程,为了减少设计和编码的复杂性,用户常使用单一代码方法编写SPMD应用程序。

下面用MPI以C语言表示方式来示例上一节所举的计算π的消息传递程序,不期望读者完全理解它,但求读者能领会消息传递程序的基本特点。

```
//*计算π消息传递编程代码段*//
#define  N  100000
  main(){
  double local,pi,w;
        long i,taskid,numtask;
  A:    w=1.0/N;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
        MPI_Comm_Size(MPI_COMM_WORLD,&numtask);
  B:    for(i=taskid;i<N;i=i+numtask){
        P:local=(i+0.5)*w;
        Q:local=4.0/(1.0+local*local);
       }
  C:    MPI_Reduce(&local,&pi,1,MPI_Double,MPI_MAX,0,MPI_
        COMM_WORLD);
  D:    if(taskid==0) printf("pi is %f\n",pi*w);
        MPI_Finalize();
       }/*main()*/
```

有关消息传递操作的细节将在第十四章中讨论。

12.6.5 共享变量模型

在共享变量(Shared_Variable)模型中,驻留在各处理器上的进程可以通过读/写公共存储器中的共享变量相互通信。它与数据并行模型的相似之处在于它有一个单一的全局名字空间;它与消息传递模型的相似之处在于它是多线程的和异步的。

然而数据是驻留在单一共享地址空间中,因此不需要显式分配数据,而工作负载既可显式也可隐式分配。通信通过共享的读/写变量隐式完成,而同步必须是显式的,以保持进程执行的正确顺序。

共享变量模型尚无一个可广泛接受的标准。例如一个为 SGI Power Challenge 编写的程序不能直接运行在 Convex Exemplar 上;一个为 SMP 或 DSM 开发的共享变量程序不能运行在诸如 MPP 和机群的多计算机上。

关于共享变量模型尚须说明以下几点:①一个广为流传的错误概念是共享变量模型运行细粒度(Fine Granularity)的并行性比消息传递模型好:要注意共享变量模型是一种并行编程模型,它可以实现在 PVP、SMP、DSM、MPP 或甚至机群的任意并行平台上。支持细粒度并行性的平台应具有有效的通信/同步机制,一个共享变量的程序可能遭致高的相互作用开销从而远比运行在机群、MPP 甚至 SMP 上的消息传递程序慢得多。②一个普遍的说法是共享变量编程比消息传递编程容易 此说法虽不错,但科学试验的事实尚未完全建立。为了开发一个新的、有效的、松散同步的、通信模式规则的并行程序,共享变量的方法未必比消息传递方法容易。当然对于非规则的并行程序,使用消息传递原语很难指明所需要的相互作用;同时共享变量模型允许全局指针操作,而消息传递模型是无此能力的;此外共享变量编程虽不必明显地划分和分配数据,但这也可能伤害性能。③就查错而论,共享变量程序可能比消息传递程序更困难:共享变量程序中的所有进程都驻留在单地址空间中,而且访问共享数据必须由同步结构(如锁和临界区)加以保护。同步错误易于出现,而且一旦出现就难以查找;但在消息传递程序中,此类错误出现的频度大大减少,因为诸进程不共享单一地址空间。

最后,为了比照起见,下面也给出用类 C 语言方式书写的计算 π 的共享变量程序。读者可能暂不能完全理解,但至少可以理解其编程概貌。

```
//*计算π共享变量编程代码段*//
    #define N 1000000
    main ( ) {
        double local,pi=0.0,w;
        long i;
    A:  w=1.0/N;
    B:  #Pragma Parallel
        #Pragma Shared (pi,w)
        #Pragma Local (i,local)
        {
```

```
    # Pragma pfor iterate(i=0;N;1)
    for(i=0;i<N;i++) {
    P:local=(i+0.5)*w;
    Q:local=4.0/(1.0+local*local);
    }
C:  # Pragma Critical
    pi=pi+local;
    }
D:  printf("pi is %f\n",pi*w);
    }/*main()*/
```

其中,Pragma 即编译制导,意义同前(见第 12.6.2 节)。

12.6.6 并行程序设计模型比较

三种显式并行程序设计模型(数据并行、消息传递、共享变量)的主要特性可综合于表 12.3 中。

表 12.3 三种显式并行程序设计模型主要特性一览表

特性	数据并行	消息传递	共享变量
控制流(线)	单线程	多线程	多线程
进程间操作	松散同步	异步	异步
地址空间	单一地址	多地址空间	单地址空间
相互作用	隐式	显式	显式
数据分配	隐式或半隐式	显式	隐式或半隐式

四种并行程序设计模型(隐式并行、数据并行、消息传递并行、共享变量并行),从用户的观点也可作一比较。使用星号(★)的多少来表征它们性能优劣程度,其中四星(★★★★)表示性能最好,一个星(★)表示性能较差。表 12.4 中的结果,只是现状的写照,将来也许会变化。

表 12.4 四种并行程序设计模型比较一览表

比较项目	隐式并行	数据并行	消息传递	共享变量
并行性	★★★★	★★★	★	★★
数据分配	★★★★	★★	★	★★★
通信	★★★★	★★★	★	★★★
同步	★★★★	★★★★	★★	★
正确性	★★★★	★★★	★	★
可移植性	★★★★	★★★	★★★	★
通用性	★	★★	★★★	★★★
结构性	★★★★	★★★	★	★
与平台无关的标准	Kap, Forge	Fortran90, HPF	PVM, MPI	X3H5, Pthreads, OpenMP
与平台有关的标准	Convex Exemplar	CMC*	SP2MPL, ParagonNX	Cray Craft, SGI Power C

12.7 小结和导读

小结 本章首先对并行程序设计作了一般性介绍,包括并行程序设计和串行程序设计的差别、并行程序设计的困难所在,并行程序设计的环境与工具,并行程序设计的一般方法和并行编程风范;然后对并行程序中最基本的计算要素——进程和线程的基本概念以及支持进程和线程的管理、同步和通信的各种软/硬件机制进行了讨论;最后着重对四种并行程序设计模型进行了讨论。

有关进程和线程的一些问题,已在操作系统课程中和计算机体系结构课程中详细讨论过,所以熟悉这些内容的读者,可以跳过这些章节;而不熟悉这些内容的读者,在阅读本章时恐怕要随时参照有关教材内容。但不管如何,由于本章所介绍内容,对学习并行计算针对性较强,所以复习或学习本章内容也是必要的。

导读 关于并行程序设计问题读者可参阅[127,194];对并行程序设计的困难性,[137]中进行了讨论;讨论并行编程风范参见[88];提倡 SPPP (Sequential_Program,Parallel_Processing)编程风格的是[140]。从 OS 角度讨论进程和线程的概念,读者可参阅[166],关于同步的讨论,读者可参阅[126]。有关双复制通信过程最好参阅 IBMS P2 通信子系统[163];有关 FM 的讨论,读者可参阅[118];零复制使用在 Princeton 大学 SHRIMP 项目中[35];TCP/IP 协议组的讨论读者可参阅 Comer 的著作[52]。

习　题

12.1　假定要录制一部数字电影。电影的一帧有1 024×1 024个像素，每个像素需4 B(4个字节)，即每帧需要4 MB。如果电影每秒钟有30帧，不经压缩，问一部两小时的电影需要多少字节？

12.2　当Cray Research欲研制一台能扩展至2 048个处理器，每个处理器有64 MB的局存的T3D MPP时，他们使用了DEC Alpha 21064微处理器芯片，该芯片为34位。试问其物理地址空间够用吗？应要多少位的物理地址才行？

12.3　在一个计算机系统中，如果10个进程使用一个处理器。假定处理器的一个交互进程须5 s，而其余的各进程须1 000 s。

试问：①在批处理时，用户必须等待多久？

②在分时处理时，用户必须等待多久？

12.4　在上题中，假定处理器的一个实时进程须5 s。如果某一实时任务需10 s内完成，那么采用简单的分时方法能满足要求吗？如果采用抢先策略，应如何处理？

12.5　假定有n个进程P(0)，P(1)，…，P($n-1$)，数组元素$a[i]$开始时被分配给进程P(i)。试写出求归约和$a[0]+a[1]+\cdots+a[n-1]$的代码段，并以$n=8$示例之。

12.6　图12.15示出了点到点和各种集合通信操作。试根据该图解释点到点、播送、散播、收集、全交换、移位、归约与前缀和等通信操作的含义。

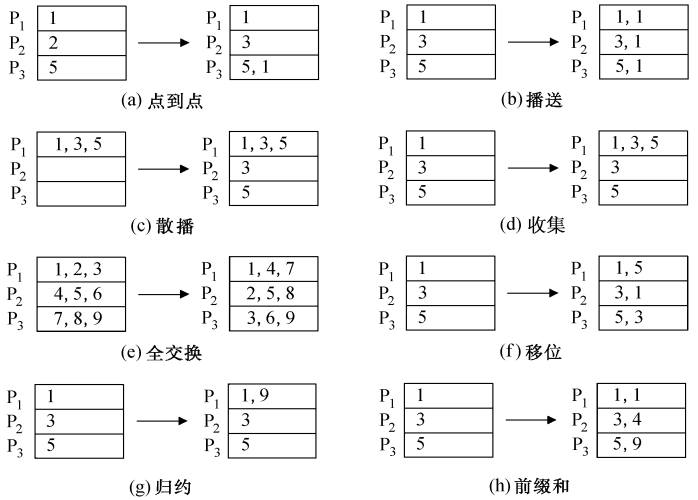


图12.15　点到点和集合通信操作

12.7 某一机群系统,节点使用 800 MB/s 的存储总线,133 MB/s 的 I/O 总线和 200 MB/s 的 NIC DMA 总线。

试问:① 如通信网络的峰速为 1 Gb/s,那么通信带宽的硬件限制是什么?

② 如果网络升级为 400 MB/s 应如何办?

12.8 假定某公司在银行中有三个账户 X、Y 和 Z,它们可以由公司的任何雇员随意访问。雇员们对银行的存、取和转账等事务处理的代码段可描述如下:

```
/*从账户 X 支取 ¥100 元*/
    atomic {
        if (balace [X] > 100) balance [X] = balance [X] -100;
    }
/*从账户 Y 存入 ¥100 元*/
    atomic {balance [Y] = balance [Y] +100}
/*向账户 X 中转 ¥100 元到账号 Z*/
    atomic {
        if (balance [X] > 100 {
            balance [X] = balance [X] -100;
            balance [Z] = balance [Z] +100;
        }
    }
```

其中,atomic { } 为子原子操作。试解释为什么雇员们在任何时候(同时)支、取、转账时,这些事务操作总是安全有效的。

12.9 考虑如下使用 lock 和 unlock 的并行代码:

```
parfor (i=0;i<n;i++) {
    noncritical section
    lock (S) ;
        critical section
    unlock (S) ;
    }
```

假定非临界区操作取 T_{ncs} 时间,临界区操作取 T_{cs} 时间,加锁取 t_{lock} 时间,而去锁时间可忽略。则相应的串行程序需 $n(T_{ncs}+T_{cs})$ 时间。试问:

①总的并行执行时间是多少?

②使用 n 个处理器时加速多大?

③你能够忽略开销吗?

12.10　计算两整数数组之内积的串行代码段如下：

```
Sum=0;
for(i=0;i<N;i++)
    Sum=Sum+A[i]*B[i];
```

试用①相并行；②分治并行；③流水线并行；④主－从行并行；⑤工作池并行等五种并行编程风范，写出如上计算内积的并行代码段。

第十三章　共享存储系统并行编程

本章集中讨论基于共享变量的共享存储系统的并行编程。首先简单介绍一下共享存储并行编程的一些基本问题以及纯共享存储环境和虚拟共享存储环境；接着介绍早期的共享存储编程模型ANSIX3H5和POSIX；最后着重讨论OpenMP标准，主要包括OpenMP简介、编译制导语句、运行库例程和环境变量等。

13 1 基于共享变量的共享存储并行编程

在 20 世纪 80 年代,高性能的科学和工程计算中基于共享变量的共享存储的编程模式曾是一统天下。进入 20 世纪 90 年代后,尽管分布存储的大规模并行处理系统已夺走了峰值计算速度的桂冠,但共享存储的并行处理仍以其可编程性和系统的可用性的优势,在科学和工程计算中与分布存储系统共领风骚。

在共享存储的编程模式中,各个处理器可以对共享存储器中的数据进行存取,数据对每个处理器而言都是可访问到的,不需要在处理器之间进行传送,即数据通信是通过读/写共享存储单元来完成的。

13 1 1 共享存储并行编程的基本问题

共享存储的并行程序设计的基本问题包括:①任务划分:任务划分就是把一个程序划分成若干个可以分配给不同的处理器去并行执行的一组任务,划分的方法与并行程序设计风格有关:单程序多数据流(SPMD)编程风格采用按数据流划分任务的方法,即域分解法,它将要计算的问题的区域分解成多个子域,每个任务计算一个子域,这样实现的并行也称为数据并行;多程序多数据流(MPMD)编程风格则采用按控制流划分任务的方法,即功能分解法,它将要计算的问题分解成多个子问题,每个任务计算一个子问题,这样实现的并行也称为控制并行。②任务调度:调度就是把一个任务集合分配给一组处理器,传统的调度是操作系统管理的事,但因为由操作系统施行调度开销较大,且操作系统难于从程序中获得优化调度信息,所以现代并行机上一个程序内的调度多由用户、编译器和运行库来完成。任务调度有静态调度和动态调度之分:静态调度(Static Scheduling)由程序员在编程时、或者编译器在编译时将任务分配给处理器。静态调度有确定的和非确定的两种模式:确定模式(Deterministic Mode)任务之间的优先关系和任务所需的执行时间在调度之前是固定和已知的,常可使用 Gantt 图(Gantt Chart)来说明调度过程;非确定模式(Nondeterminstic Mode)任务的执行时间可表示为一随机变量,这就使得调度问题更为困难。动态调度(Dynamic Scheduling)在运行时将任务分配给处理器,分配的策略是在编程或编译时确定的,但具体的分配是在运行时才能确定的。③任务同步:同步对并行程序设计是非常重要的,同步常用来确定任务间正确的执行次序,或用于确保各任务对共享变量的正确的读写次序。简单高速的同步机制一般由硬件支持,复杂多功能的同步机制常由软件实现。已如第十二章所述,

同步方式有锁、路障、事件、信号灯和管程等。注意错误的同步会导致死锁。④任务通信 在共享存储系统中,任务间的通信借助于读写共享变量来完成,不需要专门的机制,但应注意读写的时机,即要在发送者将正确的信息送出后,接收者才可去读取,为此常要使用路障同步操作,其实同步也可视为一种特殊的通信,只不过所交换的是控制信息,而不是一般通信操作中所交换的数据信息。

13.1.2　共享存储编程环境

根据并行系统中存储器的物理分布,共享存储系统可分为纯共享存储环境和虚拟共享存储环境 前者对应于 SMP 结构,相应的访存模型为 UMA;后者对应于 DSM 结构,相应的访存模型为 NUMA。

纯共享存储环境　纯共享存储环境的主要特点是:①系统中存在着一个集中的、公共的共享存储器;②系统中集中的存储器对程序员而言是全局统一编址的;③系统不提供对非一致存储访问应用程序的任何支持。大多数早期的基于共享存储的并行程序,常使用信号灯、条件临界区和管程等进行任务(进程、线程)间的通信。近代的纯共享存储并行编程标准有 ANSI X3H5、Pthreads 和 OpenMP 等。

虚拟共享存储环境　虚拟共享存储环境的主要特点是:①系统中分布的局部存储器组成了系统的全局共享虚拟存储器;②系统中的虚拟共享存储器对程序员而言是全局可寻址的,即由操作系统负责将这些物理上分布的局部存储器向用户呈现为共享的存储视图,所共享的数据空间是连续的,且可以用通常的读、写操作访问之。近代的虚拟共享存储编程标准有 Linda 和 Jiajia 等。

13.2　早期共享存储并行编程模型

13.2.1　ANSI X3H5 共享存储模型

X3H5 共享存储模型(X3H5 Shared_Memory Model)是 1993 年建立的 ANSI 标准,它定义了一个概念标准编程模型,已与 C、Fortran77 和 Fortran90 三种语言相结合。下面我们只简介其主要思想,而细节可参阅[16]。

在图 13.1 中示出了 X3H5 中所使用的指明并行性的一些结构。并行结构(Parallel Construct)也称为并行区域(Parallel Region),是些成对的(Parallel…end Parallel)。程序以单线程(也叫基线程(Base Thread)或主线程(Master Thread))串

行模式开始执行,当其遇到 parallel 时就开启到并行模式生成多个子线程;基线程及其子线程并行执行其后的代码直到遇到 end parallel,然后又返回到串行模式,由基线程继续执行之。

```
program main                    4  程序从串行模式开始
        A                       !  A 由基线程执行
        parallel                !  开启到并行模式
        B                       !  B 被每个子线程复制
        psections               !  开始执行并行块
        section
        C       section         !  一个子线程执行 C

        D                       !  另一个线程执行 D
        endpsections            !  等待 C 和 D 都完成
        psingle                 !  E 由一个子线程执行
        E
        endpsingle              !  开启到串行模式
        pdo i=1,6               !  pdo 结构开始
        F ()                    !  子线程分享 F 的 6 次迭代
        end pdo no wait         !  无隐路障
        G                       !  复制 G
        end parallel            !  返回至串行模式
        H                       !  H 由起始基线程执行
        r
end
```

图 13.1 ANSI X3H5 标准中并行结构

在并行结构内可有一些并行块 (psections…end psections)、并行循环 (pdo…endpdo) 或单一进程 (psingle…endpsingle) 等结构,其中并行块用以指明 MPMD 并行性,并行循环用于指明 SPMD 并行性,而单一进程结构指明代码仅由一个线程顺序执行。

图 13.1 中的代码执行顺序可以更清楚地示于图 13.2 中。假定有三个线程 P、Q 和 R;开始基线程 P 执行代码 A;当遇到 parallel 时就生成两个子线程 Q 和 R,所以代码 B 被复制成三份;在并行块 (psections) 中两段代码 C 和 D 被并行执行;单进程代码 E 仅由 Q 执行;并行循环 (pdo) 有 6 次迭代,由三个线程分摊执行;接着 P、Q、R 并行执行代码 G;最后由基线程 P 执行代码 H。在此执行过程中,在 parallel、endpsections、end psingle、end pdo 和 end parallel 处设有隐路障 (Implicit Barrier),它们也称为栅栏操作 (Fence Operation),强使所有存储访问统一于一点以保持一致。如果勿需等待,亦可使用勿需等待隐路障,简称为无隐路障 (No Implicit

Barrier)。

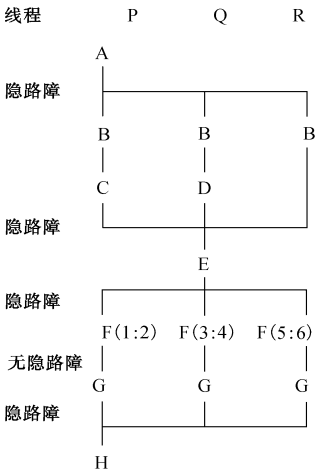


图 13.2　图 13.1 的执行过程

在 X3H5 模型中,线程间的相互作用具有很多很有兴趣的性质:①并行结构中的变量具有共享/私有属性,其中一个线程的私有变量对别的线程是不可见的;②X3H5 非常重视存储器的一致性问题,隐路障、栅栏操作和显路障都可用于此目的;③X3H5 模型引入四种形式的同步变量,即闩锁(Latch)、锁、事件和顺序(Ordinal)。每种类型的变量都伴有初始化操作和释放(Destroy)操作,任何变量在使用前均必须初始化,而采用赋值非初始化状态的办法变可使其释放。锁和事件同步类似于第 12.4.2 节所讨论的内容;而顺序变量用于按照线程的次第(Rank)进行线程同步,例如可以使用顺序变量指定线程 T_i 只有在 T_{i-1},…,T_1 在临界区均执行完毕后才能使其进入临界区;闩锁变量使用在临界区中,其语法结构如下:

```
Critical region [(latch_Variable)]
    Critical_section_code
end Critical region
```

此结构类似于普通意义的临界区结构,只是可以包含任选的闩锁变量而已。使用闩锁能减少竞争和增加并行度。

使用 X3H5 模型,如何编写计算 π 的并行程序,留给读者(习题 13.3)。

13.2.2 POSIX线程模型

POSIX (Portable Operating System Interface) Threads,即 Pthreads,代表官方 IEEE POSIX1003.1C_1995 线程标准,系由 IEEE 标准委员会所建立,其功能和界面类似于 Solaris 线程的功能与界面。下面我们只简介其公共性质。

线程管理 (Thread Management) 线程库用于管理线程,pthreads 中基本线程管理原语如表 13.1 所示。其中 pthread_create()在进程内生成新线程,新线程执行带有变元 arg 的 myroutine,如果 pthread_create()生成,则返回 0 并将新线程之 ID 置入 thread_id,否则返回指明错误类型的错误代码;pthread_exit()结束调用线程并执行清场处理;pthread_self()返回调用线程的 ID;pthread_join()等待其它线程结束。

表 13.1 Pthreads 中基本线程管理原语一览表

功 能	含 义
int pthread_create (pthread_t *thread_id pthread_attr_t*attr, void*(*myroutine)(void*), void *arg);	生成线程
void *pthread_exit*(void*status)	退出线程
int *pthread_join*(pthread_t thread,void **status)	联合线程
pthread_t pthread_self(void)	返回调用线程 ID

线程调度 pthread_yield()的功能是使调用者将处理机让位于其它线程;pthread_cancel()的功能是终止指定的线程。

Pthread 同步 Pthread 同步原语列于表 13.2 中。重点讨论互斥 mutex (mutual_Exclusion)变量和条件 Cond (Conditional)变量。前者类似于信号灯结构;后者类似于事件结构。注意,使用同步变量之前需被初始化 (生成),用后应释放清除之。

表 13.2　Pthreads 中线程相互作用原语一览表

功　能	含　义
pthread_mutex_init (…)	生成新的互斥变量
pthread_mutex_destroy (…)	释放互斥变量
pthread_mutex_lock (…)	锁住互斥变量
pthread_mutex_trylock (…)	试探锁住互斥变量
pthread_mutex_unlock (…)	开锁互斥变量
pthread_cond_init (…)	生成新的条件变量
pthread_cond_destroy (…)	释放条件变量
pthread_cond_wait (…)	等待 (阻塞) 条件变量
pthread_cond_timedwait (…)	等待条件变量直至到达时限
pthread_cond_sinnal (…)	投寄一个事件,开锁一个等待进程
pthread_cond_broadcast (…)	投寄一个事件,开锁所有等待进程

pthread_mutex_lock ( ) 锁住互斥 (mutex) 变量,如果它未被加锁;如果 mutex 已被加锁,调用线程一直被阻塞到 mutex 变成有效。pthread_mutex_trylock ( ) 类似于 test_and_set,它一旦锁住 mutex 即立刻返回。pthread_mutex_unlock ( ) 释放先前所获得的 mutex,当 mutex 被释,它就能由别的线程获取。

pthread_cond_wait ( ) 自动阻塞等待条件满足的现行线程,并开锁 mutex 变量。pthread_cond_timedwait ( ) 与 pthread_cond_wait ( ) 类似,除了当等待时间达到时限它将解除阻塞外。pthread_cond_signal ( ) 解除等待条件满足的已被阻塞的一个线程的阻塞。pthread_cond_broadcast ( ) 将所有等待条件满足的已被阻塞的线程解除阻塞。

使用 pthreads 编写计算 π 的程序比较复杂,读者可参见习题 13.4。

13.3　OpenMP 编程简介

OpenMP 标准 (OpenMP Standard) 是共享存储体系结构上的一个编程模型,已经应用到 UNIX、Windows NT 等多种平台上。本节讲述基本 OpenMP 并行程序设计所需要的知识,包括 OpenMP 简单介绍、编译制导语句、运行库例程和环境变

量等。通过本节的学习,可以掌握 OpenMP 最基本和最常见的程序设计方法,编写出能满足一般需求的 OpenMP 并行程序。

13.3.1 OpenMP概述

OpenMP 起源于 ANSI X3H5 标准,它具有简单、移植性好和可扩展等优点,是共享存储系统编程的一个工业标准。实际上 OpenMP 并不是一门新的语言,它是对基本语言(如 Fortan 77、Fortan 90、C、C++ 等)的扩展。OpenMP 规范中定义的制导指令、运行库和环境变量,能够使用户在保证程序的可移植性的前提下,按照标准将已有的串行程序逐步并行化。制导指令是对程序设计语言的扩展,进一步提供了对并行区域、工作共享、同步构造的支持,并且支持数据的共享和私有化。这样,用户对串行程序添加制导指令的过程,就类似于进行显式并行程序设计。运行库和环境变量,使得用户可以调整并行程序的执行环境。OpenMP 的提出,是希望遵循该并行编程模型的并行程序,可以在不同的厂商提供的共享存储体系结构间比较容易地移植。实际上,已经有许多硬件和软件供应商提供支持 OpenMP 的编译器,如 DEC、Intel、IBM、HP、Sun、SGI 及 U.S. DOE ASCI program 等,并且包括了 UNIX 和 NT 两种操作系统平台。目前,Fortran 77、Fortran 90、C、C++ 语言的实现规范已经完成,详细说明可参看 http://www.openmp.org。

13.3.2 OpenMP编程风格

OpenMP 并行编程模型　首先,OpenMP 是基于线程的并行编程模型(Parallel Programming Model),一个共享存储的进程由多个线程组成,OpenMP 就是基于已有线程的共享编程模型;其次,OpenMP 是一个外部的编程模型,而不是自动编程模型,它能够使程序员完全控制并行化。

OpenMP 使用 Fork-Join 并行执行模型。所有的 OpenMP 程序开始于一个单独的主线程(Master Thread)。主线程会一直串行地执行,直到遇见第一个并行域(Parallel Region)才开始并行执行。接下来的过程如下:① Fork:主线程创建一队并行的线程,然后,并行域中的代码在不同的线程队中并行执行;② Join:当主线程在并行域中执行完之后,它们或被同步或被中断,最后只有主线程在执行。实际上,所有 OpenMP 的并行化,都是通过使用嵌入到 C/C++ 或 Fortran 源代码中的编译制导语句来达到的。并且,一个 OpenMP 应用编程接口(API)的并行结构可以嵌入到别的并行结构中去。应用编程接口还可以随着不同并行域的需要动态地改变线程数。有些应用也可能不支持上述性质。

OpenMP 程序结构　让我们先了解一下 OpenMP 程序具体的结构。下面分别是基于 Fortran 语言的 OpenMP 程序的结构和基于 C/C++ 语言的 OpenMP 程序的结构。

(1) 基于 Fortran 语言的 OpenMP 程序的结构

```
PROGRAM HELLO
INTEGER VAR1,VAR2,VAR3
…
!$OMP PARALLEL PRIVATE(VAR1,VAR2) SHARED(VAR3)
…
!$OMP END PARALLEL
…
END
```

(2) 基于 C/C++ 语言的 OpenMP 程序的结构

```
#include<omp.h>
main(){
int var1,var2,var3;
…
#pragma omp parallel private(var1,var2) shared(var3)
{
…
}
…
}
```

13.3.3　OpenMP 编程要素

有了前面的基础知识,接下来就可以学习 OpenMP 编程了。由于 OpenMP 有着一套自己的编译制导语句,所以,要逐个讲解这些语句。需要注意的是,OpenMP 是基于共享存储的编程模型,因此,它本身必然有着符合自己体系结构特点的语句。在学习中,我们要体会这些语句的特点和学会使用它们。另外,由于 OpenMP 可以嵌入到 C/C++ 或 Fortran 等语言中去,所以具体的 OpenMP 程序在不同的环境下会有一些不同。下面主要介绍基于 C/C++ 语言的 OpenMP 的编程。

一个简单的 OpenMP 程序实例　在前文中,已经了解了 OpenMP 程序的简单结构。下面给出一个具体的 OpenMP 程序。虽然程序比较简单,但是可以使读者对 OpenMP 有一个感性的认识,消除对编写 OpenMP 并行程序的疑虑。之后,我

们由简单到复杂对 OpenMP 进一步讲解。

C 语言的程序设计者对“Hello World”这一例子也许还记忆犹新,因为这一例子非常简单,且又具有一定的代表性,因此,我们首先向读者介绍这一例子。下面给出的 OpenMP 并行程序是基于 C/C++ 语言的 OpenMP 程序结构的一个具体实现。

```
//* 用 OpenMP (C) 编写 Hello World 代码段 *//
 #include <omp.h>
 int main (int argc,char* argv[])
 {
      int nthreads,tid;
      int nprocs;
      char buf[32];
/*      Fork a team of threads      */
 #pragma omp parallel private (nthreads,tid)
 {
/*      Obtain and print thread id      */
 tid=omp_get_thread_num();
 printf("Hello World from OMP thread %d\n",tid);
/*      Only master thread does this      */
      if (tid==0) {
          nthreads=omp_get_num_threads();
          printf("Number of threads %d\n",nthreads);
       }
     }
   return 0;
}
```

经过 OpenMP 的编译,该程序的可能运行结果为:

```
Hello World from OMP thread 2
Hello World from OMP thread 0
Number of threads 4
Hello World from OMP thread 3
Hello World from OMP thread 1
```

在下文中,我们将以它们为例子,具体剖析 OpenMP 的使用。

编译制导语句　下面分别介绍编译制导语句格式和作用域。

(1) 语句格式：参看基于C/C++语言的OpenMP程序结构，我们可以看到，在并行开始的部分，需要一条语句“#pragma omp parallel private (var1, var2) shared (var3)”。在“Hello world”例子中，并行部分开始时有条语句“#pragma omp parallel private (nthreads, tid)”进行Fork，这条语句就是OpenMP编译制导语句，具体的编译制导语句格式解释如表13.3。

表13.3　编译制导语句格式的解释一览表

#pragma omp	directive name	[clause, …]	newline
制导指令前缀。对所有的OpenMP语句都需要这样的前缀	OpenMP制导指令。在制导指令前缀和子句之间必须有一个正确的OpenMP制导指令	子句。在没有其他约束条件下，子句可以无序，也可以任意地选择。这一部分也可以没有	换行符。表明这条制导语句的终止

(2) 作用域：编译制导语句的作用域(Scoping)可分为静态范围、孤幼制导和动态范围。静态范围(Static Extent)，又称为词法范围(Lexical Extent)，指的是文本代码在一个编译制导语句之后被封装到一个结构块中。一个语句的静态范围并不能用到多个例程或代码文件中。孤幼制导(Orphaned Directive)指的是，一个OpenMP的编译制导语句并不依赖于其他的语句。它存在于其他的静态范围语句之外，可以作用于所有的例程和可能的文件代码。一个语句的动态范围(Dynamic Extent)包括它的静态(词法)范围和孤幼制导范围。

并行域结构　一个并行域(Parallel Region)就是一个能被多个线程执行的程序块，它是最基本的OpenMP并行结构，也就是前面“Hello World”例子中的如下程序段：

```
#pragma omp parallel private (nthreads, tid)
{
…
}
```

该段被并行域封装起来，其中并行域的具体格式如下：

#pragma omp parallel [if (scalar_expression) | private (list) | shared (list) | default (shared | none) | firstprivate (list) | reduction (operator: list) | copyin (list)] newline

注意，当一个线程运行到parallel这个指令时，线程会生成一个线程列，而其自己会成为主线程。主线程也是这个线程列的一员，并且线程号为0。当并行域开

始时,程序代码就会被复制,每个线程都会执行该代码。这就意味着,到了并行部分结束会有一个路障(Barrier),且只有主线程能通过这个路障。并行域的线程数是由下面的因素决定,且优先级逐步递减,而线程号依次为0(主线程)到n-1:①所使用的库函数,即omp_set_num_threads();②所设置的环境变量,即OMP_NUM_THREADS;③所使用的默认值。一般地,有多个并行域的程序可以有相同的线程数。系统可以动态地改变一个特定并行部分的线程数。使用动态线程的方法有两个:①使用库函数omp_set_dynamic();②设置环境变量OMP_DYNAMIC。此外,一个并行域可以嵌入到另一并行域中。若有IF子句,其值必须为TRUE(Fortran)或非零(C/C++),这样才能保证产生一个线程列。否则,该域就会只有主线程串行执行。特别指出,不能跳入与跳出并行域,且只能允许一个IF存在。

回头看一下"Hello World"的例子,可以看到,每个线程执行并行部分的所有代码。可使用OpenMP的运行库例程获得线程ID和线程数。

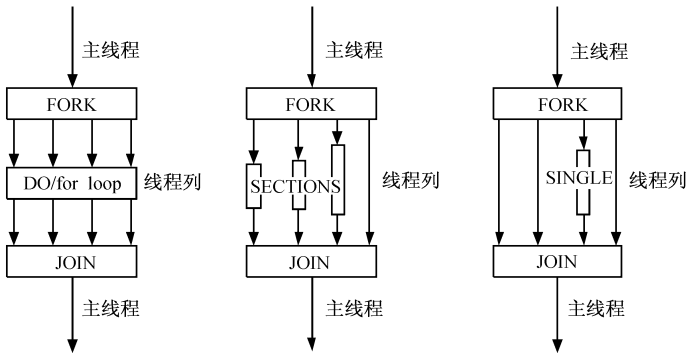


图13.3 共享任务类型

共享任务结构 共享任务结构把其内封闭的代码段划分给线程队列中的各线程执行。它不产生新的线程,也不存在着进入共享任务结构时有个路障,但在共享任务结构结束时有一个隐含的路障。图13.3中示出了三种典型的共享任务结构,其中①DO/for:共享队列中循环代表一种数据并行的类型;②SECTIONS:把任务分割成各部分,每个部分由一个线程执行,可以看成过程并行类型;③SINGLE:由线程序列中的一个线程串行执行。

注意,一个共享任务结构必须在并行域中动态地封装,这是为了能够指示并行执行。共享任务结构必须出现在所有的队列中或一个队列也不出现。队列的所有成员中连续的共享任务结构按同样的次序出现。

(1) DO/for编译制导语句:DO/for语句,即Fortran中DO语句和C/C++中

for 语句,它表明若并行域已经初始化了,之后的循环必须由线程队列并行地执行;否则就会顺序执行。语句格式如下:

pragma omp for [schedule (type [, chunk]) | ordered | private (list) | firstprivate (list) | lastprivate (list) | shared (list) | reduction (operator 1ist) |] newline

其中,schedule 子句用来描述如何划分线程队中的循环。type 为 static 时,循环被划分成块,静态地分配给线程执行。若对 chunk 没有特别声明,循环会在线程列中尽量平分。type 为 dynamic 时,循环分成的块被动态安排到线程列中处理。若一个线程完成了一块,它就会被动态地安排执行别的块。默认的块的长度为 1。下面介绍向量加法的例子,体会一下 DO/for 语句的具体用法。

```
# include <omp.h>
# define CHUNKSIZE 100
# define N 1000
main(
{
int i, chunk;
float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
    {
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
    } /* end of parallel section */
}
```

(2) SECTIONS 编译制导语句:SECTIONS 编译制导语句是非循环的共享任务结构,它表明内部的代码是被线程队列分割。不关联的 SECTIONS 编译制导语句可以相互嵌套。SECTIONS 语句的格式为:

#pragma omp sections [private (list) | firstprivate (list) | lastprivate (list) | reduction (operator: list) | nowait] newline

```
{
    #pragma omp section newline
      structured_block
    #pragma omp section newline
```

```
        structured_block
    }
```

注意,在没有 nowait 语句时,SECTIONS 语句之后有一路障。

为了比较,下面我们用 SECTIONS 语句来实现向量加法运算。在开始的 N/2 个 DO 循环由第一个线程执行,后面的由第二个线程执行。当任一个线程执行完时,它都会进行下面的操作(NOWAIT)。下面是具体的程序:

```
#include <omp.h>
#define N    1000
main()
{
   int i;
   float a[N], b[N], c[N];
   /* Some initializations */
   for (i=0; i < N; i++)
     a[i] = b[i] = i * 1.0;
   #pragma omp parallel shared(a,b,c) private(i)
     {
     #pragma omp sections nowait
       {
       #pragma omp section
       for (i=0; i < N/2; i++)
         c[i] = a[i] + b[i];
       #pragma omp section
       for (i=N/2; i<N; i++)
         c[i] = a[i] + b[i];
       } /* end of sections */
     } /* end of parallel section */
}
```

(3) SINGLE 编译制导语句:SINGLE 编译制导语句表明内部的代码只是由一个线程执行。这个语句是非常有用的。具体的格式如下:

#pragma omp single [proivate(list) | firstprivate(list) | nowait] newline

若没有 nowait语句,队列中没有执行 SINGLE 语句的线程,则会一直等到代码路障同步才会继续执行下面的代码。

组合的并行共享任务结构 下面分别介绍两种编译制导语句。

(1) PARALLEL DO/parallel for 编译制导语句:parallel for 编译制导语句(即 Fortran 中 PARALLEL DO 语句)表明一个并行域包含一个单独的 DO/for 语句,其

格式如下：

#pragma omp parallel for [if (scalar _ logical _ expression) | default (shared | none) | schedule (type [, chunk]) shared (list) | private (list) | firstprivate (list) | lastprivate (list) | reduction (operator 1list) | copyin (list)] newline

该语句的格式必须符合 PARALLEL 和 DO/for 语句的格式。下面给出一个 parallel for 的例子，为了比较，仍以向量加法作为例子，看看 paraller for 的使用方法，由此可以看出每个循环被平均分到各个线程上：

```
#include<omp.h>
#define N       1000
#define CHUNKSIZE     100
main ()  {
int i,chunk;
float a[N],b[N],c[N];
/* Some initializations*/
for (i=0,i<N;i++)
  a[i]=b[i]=i*1.0;
chunk=CHUNKSIZE;
#pragma omp parallel for shared (a,b,c,chunk) private(i) schedule (static,chunk)
    for (i=0;i<n;i++)
     c[i]=a[i]+b[i];
}
```

(2) PARALLEL SECTIONS 编译制导语句：PARALLEL SECTIONS 编译制导语句表明一个并行域包含单独的一个 SECTIONS 语句，其具体格式如下：

#pragma omp parallel sections [default (shared | none) | shared (list) | private (list) | firstprivate (list) | lastprivate (list) | reduction (operator :list) | copyin (list) | ordered] newline

同样，相应的格式必须符合 PARALLEL 语句和 SECTIONS 语句格式。其使用方法类似 parallel for 语句。

同步结构 OpenMP 提供了多种同步结构来控制与其他线程相关的线程的执行。

(1) MASTER 编译制导语句：MASTER 编译制导语句表明一个只能被主线程执行的域。队列中所有其他的线程必须跳过这部分的代码，语句中并没有路障。其格式如下：

#pragma omp master newline

(2) CRITICAL 编译制导语句：CRITICAL 编译制导语句表明域中的代码一次只能一个线程执行。其具体的格式如下：

#pragma omp critical [name] newline

注意,当一个线程正在执行一个 CRITICAL 域,而另一个到达 CRITICAL 域准备执行时,则第一个线程执行,后一个等待直到第一个退出域为止。

(3) BARRIER 编译制导语句:BARRIER 语句同步队列中的所有线程。当有一个 BARRIER 语句时,线程必须等到所有的其他线程也到达这个路障时才可继续。然后,所有的线程都并行执行路障之后的代码。其具体的格式如下:

```
#pragma omp barrier newline
```

所有的线程要么都遇到路障,要么都不遇到路障。

(4) ATOMIC 语句:ATOMIC 语句表明一个特别的存储单元只能原子地更新,而不允许让多个线程同时去写。最重要的是,这个语句提供了一个 mini-CRITICAL 部分。其格式如下:

```
#pragma omp atomic newline
```

(5) FLUSH 语句:FLUSH 语句是用来确保执行中存储器中数据一致性,线程可见的变量在此写回存储器。其格式如下:

```
#pragma omp flush (list) newline
```

(6) ORDERED 语句:ORDERED 语句指出被包含的循环如同其在一个串行处理器上的执行一样。其格式如下:

```
#pragma omp ordered newline
structured_block
```

注意,ORDERED 语句只能出现在 DO 或 PARALLEL DO (Fortran) 和 for 或 parallel for (C/C++) 语句的动态范围中。

在任何时候,一个命令只能由一个线程执行,有分支从一个 ORDERED 块中跳进或跳出都是不合法的。循环中不能执行 ORDERED 指令多于一次,也不能执行多个 ORDERED 指令。包含 ORDERED 指令的循环,必须含有一个 ORDERED 子句。

THREADPRIVATE 编译制导语句　它表明整个文件局部变量 (Scope Variable) 在多个并行线程执行时,变成每个线程私有。其格式如下:

```
#pragma omp threadprivate (list)
```

这条语句必须出现在变量序列定义之后。每个线程都复制这个变量块,所以一个线程的写数据对于别的线程是不可见的。

数据域属性子句　学习 OpenMP 编程重要的一点是理解和会使用数据域 (Data Scope)。由于 OpenMP 是建立在共享存储编程模型之上的,所以大部分变量默认为共享。全程变量有文件域变量 (File Scope Variable) 和静态变量;局部变量有循环内的变量和并行域调用子程序的堆栈变量。

OpenMP 的数据域属性子句用来定义变量的范围,它包括 PRIVATE、FIRSTPRIVATE、LASTPRIVATE、SHARED、DEFAULT、REDUCTION 和

COPYIN 等。数据域变量与编译制导语句 PARALLEL、DO/for 和 SECTIONS 相结合可用来控制变量的范围。它们在并行结构执行过程中控制数据环境,比如,哪些串行部分中的数据变量被传到程序的并行部分以及如何传送;哪些变量对所有的并行部分的线程是可见的;哪些变量对所有的线程是局部的等。

(1) PRIVATE 子句:PRIVATE 子句表示它列出的变量对于每个线程是局部的。其具体格式如下:

private (list)

(2) SHARED 子句:SHARED 子句表示它列出的变量被线程列中所有的线程共享。其具体格式如下:

shared (list)

一个共享的变量只存在存储器的一个地方,所有的线程都能读或写这个地址。程序员能够使多个线程存取 SHARED 变量(例如通过 CRITICAL 语句)。

(3) DEFAULT 子句:DEFAULT 子句让用户规定在并行域的词法范围中所有变量的一个默认的范围(如可以是 PRIVATE、SHARED 或是 NONE)。其格式如下:

default (shared|none)

(4) FIRSTPRIVATE 子句:FIRSTPRIVATE 子句包含 PRIVATE 子句的操作,并初始化其列出的变量。其格式如下:

firstprivate (list)

(5) LASTPRIVATE 子句:LASTPRIVATE 子句包含 PRIVATE 的操作,并将变量从最后一个循环或段中复制给原始的变量。其格式如下:

lastprivate (list)

(6) COPYIN 子句:COPYIN 子句用来赋值所有线程中的同样变量与 THREADPRIVATE 中的变量一样的值。其格式如下:

copyin (list)

(7) REDUCTION 子句:REDUCTION 子句用来归约其列表中出现的变量。每个线程复制一个私有的变量列表。在归约的最后,归约的变量可以应用于所有的共享变量的私有变量列表中,最终的结果写到全局共享变量。其格式如下:

reduction (operator:list)

语句的绑定和嵌套规则 下面主要介绍 OpenMP 语句的绑定和嵌套规则。

(1) 语句绑定:对于语句的绑定,读者需要注意以下几点:①语句 DO/for、SECTIONS、SINGLE、MASTER 和 BARRIER 绑定到动态封装的 PARALLEL 中,如果没有并行域执行,这些语句是无效的。②语句 ORDERED 指令绑定到动态封装的 DO/for 中。③语句 ATOMIC 使得 ATOMIC 语句在所有的线程中互斥存取,而并不只是当前的线程列。④语句 CRITICAL 在所有线程有关 CRITICAL 指令

中互斥存取,而不是只对当前的线程列。⑤在 PARALLEL 封装外,一个语句并不绑定到其他的语句中。

(2) 语句嵌套:对于语句的嵌套,读者应注意以下几点:①PARALLEL 语句动态地嵌套到其他的语句中,从而逻辑地建立了一个新队列,但这个队列若没有嵌套的并行域执行,则只包含当前的线程。②DO/for、SECTIONS 和 SINGLE 语句绑定到同一个 PARALLEL 中,则它们是不允许互相嵌套的。③DO/for、SECTIONS 和 SINGLE 语句不允许在动态范围的 CRITICAL、ORDERED 和 MASTER 域中。④CRITICAL 语句不允许互相嵌套。⑤BARRIER 语句不允许在动态范围的 DO/for、ORDERED、SECTIONS、SINGLE、MASTER 和 CRITICAL 域中。⑥MASTER 语句不允许在动态范围的 DO/for、SECTIONS 和 SINGLE 语句中。⑦ORDERED 语句不允许在动态范围的 CRITICAL 域中。⑧任何能允许执行到 PARALLEL 域中的指令,在并行域外执行也是合法的。当执行到用户指定的并行域外时,语句执行只与主线程有关。

13.3.4 OpenMP 计算实例

下面分别给出基于 C/C++ 语言的使用并行域并行化的程序和使用共享任务结构并行化的程序以及基于 Fortran 语言描述的并行化程序。

```
//* 用并行域并行化的 OpenMP 计算π的代码段 *//
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{  int i;   double x, pi, sum[NUM_THREADS];
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS)
#pragma omp parallel
{  double x;   int id;
   id = omp_get_thraead_num(); 
   for (i=id, sum[id] =0.0; i< num_steps; i=i+NUM_THREADS) {
      x = (i+0.5) *step;
      sum[id] += 4.0/(1.0+x*x);
   }
}
   for (i=0, pi=0.0; i<NUM_THREADS; i++) pi += sum[i] * step;
}
```

```
//* 用共享任务结构并行化的 OpenMP 计算 π的代码段 *//
#include <omp.h>
static long num_steps = 100000;  double step;
#define NUM_THREADS 2
void main ()
{  int i;  double x, pi, sum[NUM_THREADS];
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS)
   #pragma omp parallel
{  double x;  int id;
   id = omp_get_thread_num();  sum[id] = 0;
#pragma omp for
     for (i=id;i< num_steps;i++) {
        x = (i+0.5)*step;
        sum[id] += 4.0/(1.0+x*x);
     }
}
for(i=0, pi=0.0;i<NUM_THREADS;i++) pi += sum[i] * step;
}
```

另外,我们给出一个基于 Fortran 语言描述的 OpenMP 计算 π的示范程序。

```
//* 用 Fortran90 语言描述的 OpenMP 计算π的代码段 *//
      program compute_pi
      integer n,i
      real w,x,sum,pi,f,a
! function to integrate
            f(a) = 4.d0/(1.d0+a*a)
            print *,'Enter number of intervals:'
            read *,n
! calculate the interval size
            w=1.d0/n
            sum=0.0d0
!$OMP PARALLEL DO PRIVATE(x), SHARED(w), REDUCTION(+: sum)
            do i=1, n
               x=w*(i+0.5d0)
               sum=sum+f(x)
            enddo
```

```
!$OMP END PARALLEL DO
        pi=w*sum
        print *, 'compute pi=',pi
        stop
    end
```

13.3.5 运行库例程与环境变量

运行库例程 OpenMP 标准定义了一个应用编程接口来调用库中的多种函数。比如获取线程或处理器数、设置使用的线程数、加锁例程、设置执行的环境变量、嵌入的并行化、动态调整线程等。对于 C/C++，在程序开头需要引用文件“omp.h”。对于加锁的例程，加锁的变量只能被加锁的例程访问，加锁的变量必须定义为“omp_lock_t”或“omp_nest_lock_t”。下面具体看几个例程：

(1) OMP_SET_NUM_THREADS：这个例程设定在下一个域使用的线程数。其格式如下：

```
void omp_set_num_threads(int num_threads)
```

(2) 其他的例程参见章末附录。

环境变量 OpenMP 提供了 4 个环境变量来控制并行代码的执行，即 OMP_SCHEDULE，OMP_NUM_THREADS、OMP_DYNAMIC 和 OMP_NESTED。所有环境变量的名字都是大写字母。下面进行具体的解释：

(1) OMP_SCHEDULE：只能用到 DO、PARALLEL DO (Fortran) 和 for、parallel for (C/C++) 中。它的值就是处理器中循环的次数。例如：

```
setenv OMP_SCHEDULE "guided, 4"
```

(2) OMP_NUM_THREADS：定义执行中最大的线程数，例如：

```
setenv OMP_NUM_THREADS 8
```

(3) OMP_DYNAMIC：通过设定变量值 TRUE 或 FALSE，来确定是否动态设定并行域执行的线程数，例如：

```
setenv OMP_DYNAMIC TRUE
```

(4) OMP_NESTED：确定是否可以并行嵌套，例如：

```
setenv OMP_NESTED TRUE
```

13.4 小结和导读

小结 本章首先谈到了共享存储的并行编程所涉及的一些基本问题，诸如任

务划分、任务调度、任务同步和任务通信等，并介绍了纯共享存储和虚拟共享存储两种编程环境的概念；然后对早期的共享存储的编程标准 ANSI X3H5 和 POSIX 做了简单介绍；最后着重介绍目前普遍采用的共享存储体系结构的 OpenMP 编程标准，包括 OpenMP 编程风格、编程要素（编译制导、共享任务结构、同步结构、数据域以及语言的绑定和嵌套规则等）以及运行库例程与环境变量等。希望通过对 OpenMP 的学习，读者可以掌握 OpenMP 最基本和最常用的程序设计方法，能编写出满足一般要求的 OpenMP 并行程序。

对于共享存储编程方法，除了本章简介的 X3H5、Pthreads 和 OpenMP 三个标准外，另外还有像 SGI POWER C 和新型并行 C 语言 C//等，前者是串行 C 语言的推广，具有编译制导和库函数；后者是基于标准 C 语言，具有少量的为并行和进程相互作用的一组扩展结构。

导读　X3H5 标准在 [16] 中给予了描述，POSIX 在 [141] 中进行了讨论，有关 OpenMP 标准描述在 [133,134,194] 中，SGI Power C 编程模型读者可参阅 [159]，新型并行语言 C//读者可参阅 [186]，函数式程序设计语言讨论在 [9] 中，面向对象方法讨论在 [6] 中，应用 C++ 进行并行编程读者可参阅 [182]，分布计算系统程序设计语言读者可参阅 [23]。

习　题

13.1　试分析下列循环嵌套中各语句间的相关关系：

①
```
DO I=1,N
    DO J=2,N
S1:   A(I,J)=A(I,J-1)+B(I,J)
S2:   C(I,J)=A(I,J)+D(I+1,J)
S3:   D(I,J)=0.1
    Enddo
Enddo
```

②
```
DO I=1,N
S1:   A(I)=B(I)
S2:   C(I)=A(I)+B(I)
S3:   D(I)=C(I+1)
Enddo
```

③
```
DO I=1,N
    DO J=2,N
```

S_1：　A(I,J)＝B(I,J)＋C(I,J)

S_2：　C(I,J)＝D(I,J)/2

S_3：　E(I,J)＝A(I,J−1)＊＊2+E(I,J−1)

Enddo

Enddo

13.2　令 $N=10^5$ 和 $N=10^8$，试编写计算 π 的 SPMD 并行程序，并在您现有的共享存储的平台上调试之；同时应执行在 1,2,3,4,5,6,7 和 8 个处理器上。

13.3　试用 X3H5 模型，写出计算 π 的并行程序。

13.4　下面是使用 Pthreads 方法计算 π 的一种并行代码段：

```
//*计算π的 Pthreads 编程代码段*//
#include <stdio.h>
#include <pthread.h>      /*线程控制所需的头文件*/
#include <synch.h>        /*同步操作所需的头文件*/
extern unsigned * micro_timer;    /*系统计时变量*/
unsigned int   fin,start;
semaphore_t semaphore;
barrier_spin_t barrier;    /*同步信号量说明*/
double delta,pi;
typedef struct {    /*定义参数结构*/
      int low;
      int high;
} Arg;
/*定义线程执行的函数*/
void child (arg)
Arg*arg
 {
      int low=arg? low;
      int high=arg? high;
      int i;
      double x ,part=0.0;
      for (i=low;i<=high;i++)  {
            x= (i+0.5) *delta;
            part+=1.0/(1.0+x*x) ;
        }
      psema (&semaphore) ;/*利用信号量进行互斥累加*/
            pi+=4.0*delta*part;
      vsema (&semaphore) ;
```

```
    _barrier_spin(&barrier);/*线程在完成计算后需barrier同步*/
    pthread_exit();/*线程终止*/
}
main(argc,argv)
int argc;
char * argv[];
{
    int no_of_threads,segments,i;
    pthread_t thread;
    Arg*arg;
    if (argc<4=3) {
        printf("usage:pi<no of threads><no of strips>\n");
        exit(1);
    }
    no_of_threads=atoi(argv[1]);
    segments=atoi(argv[2]);
    delta=1.0/segments;
    pi=0.0;
    sema_init(&semaphore,1);    /*初始化同步变量*/
    _barrier_spin_init(&barrier,no_of_threads+1);
    start=*micro_timer;    /*线程开始计时*/
    for(i=0;i<no_of_threads;i++) {    /*启动线程*/
        arg=(Arg*)malloc(sizeof(Arg));
        arg->low=i*segments/no_of_threads;
        arg->high=(i+1)*segments/no_of_threads-1;
        pthread_create(&thread,pthread_attr_default,child,arg);
    }
    _barrier_spin(&barrier);    /*主进程等待所有子线程结束*/
    fin=*micro_timer;    /*线程结束记时*/
    printf("%u\n",fin-start);
    printf("\npit\t%15.14f\n",pi);
}
```

①试解释上述代码段的工作流程。

②通过三种模型 X3H5、Pthreads和 OpenMP 上计算π的代码段，比较它们的编程风格的异同和优缺点。

13.5　下面是使用经理员工（Manager_Worker）模型（即主－从模型）求解 N_皇后问题的并行代码段：

```
//*求解 N_皇后经理_员工编程代码段*//
/*Manager 程序段*/
if (iam) {/*如果我是节点 0*/
    printf( \n STARTING…\n ) ;
    while (get_board (board) =4 DONE) {
        crecv (READY,NULL,0) ;
        nodenbr=infonode ( ) ;
        msgcount++ ;/*计数多少节点准备好*/
        csend (TASK,board,sizeof (two ,two D) ,nodenbr,0) ;
        msgcount- - ;
    }
/*等待所有员工空闲*/
    while (msgcount! =nodes-1) {
        crecv (READY,NULL,0) ;
        msgcount++ ;
    }
    /*发送 FINISHED 消息给所有节点,并退出*/
    board [0] [0] =FINISHED;
    csend (TASK,board,sizeof (two D) ,-1,0) ;
    goodbye ( ) ;
}
else{/*员工程序段*/
    for ( ; ;) {
        csend (READY,0,0,0,0) ;
        crecv (TASK,board,sizeof (board)) ;
        if (board [0] [0] = =FINISHED
            goodbye ( ) ;
        if (chk_board (board))
            move_to_right (board,0,MCOLS) ;
    }
}
```

试解释上述代码段的计算过程。

附录　OpenMP 运行库例程

OMP_SET_NUM_THREADS:

这个子例程设定在下一个域使用的线程数。其格式如下：

```
void omp_set_num_threads(int num_threads)
```

OMP_GET_NUM_THREADS

这个例程返回目前调用的执行域队列的线程数。格式如下：

```
int omp_get_num_threads(void)
```

OMP_GET_MAX_THREADS

这个例程返回调用 OMP_GET_NUM_THREADS 例程的最大值。格式如下：

```
int omp_get_max_threads(void)
```

OMP_GET_THREAD_NUM

这个例程返回队列中的线程的序号。这个序号可在 0 到 OMP_GET_NUM_THREADS −1 之间。队列中主线程的序号是 0。格式如下：

```
int omp_get_thread_num(void)
```

OMP_GET_NUM_PROCS

这个例程返回程序中使用的处理器数。格式如下：

```
int omp_get_num_procs(void)
```

OMP_IN_PARALLEL

这个例程决定执行的代码部分是并行的还是不是。格式如下：

```
int omp_in_parallel(void)
```

OMP_SET_DYNAMIC

这个子例程可以设定能或不能动态调整并行域执行的线程数。格式如下：

```
void omp_set_dynamic(int dynamic_threads)
```

OMP_GET_DYNAMIC

这个函数/例程用来决定可以或不可以调整动态线程。其格式如下：

```
int omp_get_dynamic(void)
```

OMP_SET_NESTED

这个子例程用来设定能够或不能够并行嵌套。其格式如下：

```
void omp_set_nested(int nested)
```

OMP_GET_NESTED

这个函数/例程用来决定并行嵌套是可以还是不可以，格式如下：

```
void omp_get_nested
```

OMP_INIT_LOCK

这个子例程用来初始化一个锁。其格式如下：

```
void omp_init_lock(omp_lock_t *lock)
void omp_nest_init_lock(omp_nest_lock_t *lock)
```

注意初始状态没有上锁。

OMP_DESTROY_LOCK

这个子例程用来删除一个锁。格式如下：

```
void omp_destroy_lock(omp_lock_t *lock)
```

```
void omp_destroy_nest_lock(omp_nest_lock_t *lock)
```

注意,若锁变量没有初始化,这个调用是非法的。

OMP_SET_LOCK

这个子例程使得执行线程等待,直到指定的锁可用。格式如下:

```
void omp_set_lock(omp_lock_t *lock)
void omp_set_nest_lock(omp_nest_lock_t *lock)
```

OMP_UNSET_LOCK

这个子例程表示从执行子例程中解锁。格式如下:

```
void omp_unset_lock(omp_lock_t *lock)
void omp_unset_nest_lock(omp_nest_lock_t *lock)
```

同样,若锁变量没有初始化,调用这个例程是非法的。

OMP_TEST_LOCK

这个子例程用来测试一个锁,若没有这个锁,则没有阻塞。格式如下:

```
void omp_test_lock(omp_lock_t *lock)
void omp_test_nest_lock(omp_nest_lock_t *lock)
```

第十四章　分布存储系统并行编程

从并行程序设计的角度来看，分布存储系统的主要特点是：系统通过互连网络将多个处理器连接起来，每个处理器均有自己的局部存储器，所有的局部存储器就构成了整个地址空间；整个地址空间有局部和全局两种编址方式，全局编址方式是指系统的所有局部存储器统一编址，用户程序空间是一个统一的全局地址空间，其中远程存储器的访问与局部存储器的访问一样用指令来完成，其指令地址由处理器号和局部存储器地址所组成。局部编址方式是系统中各局部存储器单独编址，用户程序空间是多地址空间，远程存储器的访问要通过调用消息传递库程序来实现的。

上述的特点，导致了分布存储系统的两种并行编程模型：数据并行模型和消息传递模型。实现消息传递有SPMD（单程序多数据流）和MPMD（多程序多数据流）两种模式；实现数据并行有SIMD（单指令多数据流）和SPMD两种模式。在分布存储的SIMD体系结构上可以实现SPMD模式的数据并行；在分布存储的MIMD体系结构上可以实现SPMD模式的数据并行，也可以实现MPMD模式的消息传递。

本章首先讨论消息传递的并行编程，简要讨论MPI而顺便介绍一下PVM并行编程环境；然后讨论数据并行编程，简要讨论HPF而穿插提及一下FORTRAN 90。

14.1 基于消息传递的并行编程

所谓基于消息传递的并行编程,是指用户必须显式地通过发送和接收消息来实现处理器之间的数据交换。这种编程方式是大规模并行处理机 MPP 和工作站机群 COW 采用的主要编程方式。在这种并行编程中,每个进程均有自己独立的地址空间,一个进程不能直接访问其它进程中的数据,这种远程访问必须通过消息传递来实现。因为消息传递的开销比较大,所以它主要用来开发大粒度和粗粒度的并行性。

根据问题分解的两种形式,消息传递并行性的开发也有两种形式:域分解形式,即将一个大的问题区域分解成若干个较小的问题区域,然后对其并行求解;函数分解(也有人称为功能分解)形式,即将一个大的问题分解成若干个子问题,然后对其并行求解。相应于这两种形式,就有所谓 SPMD 编程和 MPMD 编程两种模式。

14.1.1 SPMD 并行程序

基于域分解的思想开发出的并行程序,通常组织成 SPMD 形式。SPMD 就是同一程序复制到各个处理器上,而不同的数据分布在不同的处理器上。这样在系统中各处理器均运行相同的程序,但对不同的数据执行操作。在大量的处理器上运行用户软件,为了减少管理指令流的复杂性,加大并行粒度,用 SPMD 程序代替 SIMD 程序是很自然的。

设计 SPMD 形式的消息传递程序的主要步骤是:①数据划分:尽量考虑负载平衡、存储空间的均衡使用以及减少处理器之间的通信;②优化通信:尽量提高计算/通信比,数据就地使用和合并短消息成一个长消息进行传输;③全局操作:将各处理器之局部结果组合起来形成整个问题的解,像这样的操作包括全局同步、播送、归约、前缀运算、收集/散播等。

编写 SPMD 形式的程序有两种方式:主机/节点(Host/Node)式和无主机(Hostless)式,兹分述如下:

主机/节点结构　在此结构中,一个应用程序由两部分组成:一是主机程序,二是节点程序。主机程序运行在控制节点上,节点程序运行在所有的计算节点上。这种形式的程序,实际上是一个主机程序控制一组以 SPMD 方式执行的节点程序。主机程序的功能是:①申请和释放处理器,加载节点程序;②执行 I/O 和处理用户界面;③发送数据给各节点处理器,并收集各处理器的计算结果。节点程序的

功能是：①接收来自主机的输入信息；②完成各自的局部计算，施行计算节点间的通信；③回送计算结果给主机。主机/节点结构的主要优点是：①易于并行编程：欲将一个已有的串行程序修改成主机/节点形式的并行程序时，原有程序中的I/O与用户界面部分（代码量大但运行时间短）可以完全不变，而只需并行化计算部分（代码量小但运行时间长），这样开发并行程序代价较低；②充分利用主机的特殊能力：在很多MPP系统中，计算节点通常只提供最基本的操作系统功能，而图形功能、I/O功能和数据库接口功能等通常远不如控制节点强，对于需要图形功能或数据库接口功能的应用软件来说，主机/节点结构可以充分利用主机功能强的优势。但主机/节点结构也有些不足，主要是：①调试困难：采用主机/节点结构，一般不允许节点程序直接进行自己的I/O，这样当用户试图用打印信息来定位错误时，用户必须同时修改主机和节点程序，以便两者协调输出所需之信息；②维护困难：因为主机/节点结构涉及到两类节点，当两类节点结构上（字节次序、最高有效位之位置、数据类型长度不一致等）有差异时，两个程序之间交换数据产生困难，给维护增加了难度；③可移植性差：因为主机/节点程序不能直接在串行机上执行，所以用户常常需要为一个应用软件保持一个并行版本和一个串行版本，用户负担过大，给程序移植带来困难。

无主机结构　在此结构中，一个应用程序整个地运行在计算节点上，而控制节点不起上节所说的主机节点的作用，所运行的实际上是一个系统提供的通用I/O服务程序，它为计算节点提供I/O服务和计算节点所不具备的操作系统请求服务。在这种结构中，程序控制也是在计算节点上完成的。无主机结构的主要优点是：①易于程序开发、维护和移植：因为只涉及一类节点，用户只需写一种应用程序；②程序较易调试：因为程序可以在计算节点上，直接使用标准的I/O功能进行调试；③程序的正确性易于保证：因为在无主机结构上书写的并行程序，可以不加修改或少量修改即可在串行机上运行。

由于上述优点，无主机结构已成为编写SPMD程序的主要方式。

14.1.2 MPMD并行程序

基于函数分解思想开发出的并行程序，通常组织成MPMD形式。MPMD就是每个处理器执行不同的代码副本、各自对数据完成不同的运算。在诸如事务处理的应用中，由于各子功能本身固有的异构性，以及在异构网络环境下，由于各计算机资源差异性，用MPMD结构的程序是很合适的。

设计MPMD形式的消息传递程序的主要步骤是：①子问题划分：尽量考虑各处理器负载均衡和充分利用各处理器的独特能力，在全系统范围内合理分配各子

问题于不同处理器上;②确定子问题之间的相互作用方式。

实现子问题之间相互作用方式有两种:数据流方式和客户/服务器方式,兹分述如下:

数据流方式 在此方式的 MPMD 程序中,用户将问题分解为各不相同的子问题,根据子问题的相关性来组织并行,按数据驱动(Data Driver)的方式决定并行执行的先后次序。这种方式主要应用于异构型计算环境中的应用软件的开发。

客户/服务器方式 在此方式的 MPMD 程序中,用户将一些典型的不同的子问题求解作为相应的服务进程分布于各个计算节点上,整个问题的求解就归结为客户对各个服务器的一系列服务请求。这种方式主要应用于某类特殊的应用领域。在这些应用领域中,将一些普遍性的子功能(例如数据库应用中的插入、删除和查询等)作为服务进程,可以大大减轻用户重复开发具体应用软件的负担。

客户/服务器(Client/Server)方式是一种需求驱动(Demand Driven)(数据流方式是一种数据驱动)。用户所请求的服务可以是信息服务、计算服务或资源服务。客户/服务器方式的显著特点是非对等性,即客户与服务器地位不对等,服务器拥有客户所不具有的硬件资源和运算能力,服务器提供服务,客户请求服务。这种方式特别适合于子功能比较通用的应用领域。一个客户/服务器方式的 MPMD 程序示意图如图 14.1 所示。

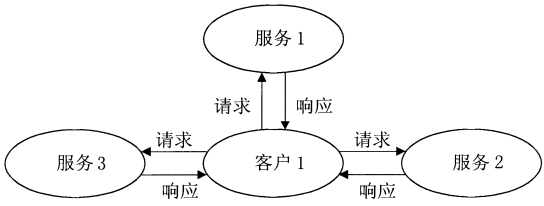


图 14.1 客户/服务器方式的 MPMD 程序

14.2 MPI 并行编程

在消息传递库方法的并行编程中,一组进程所执行的程序是用标准串行语言书写的代码加上用于消息接收和发送的库函数调用。其中,消息传递界面 MPI(Message Passing Interface)就是 1994 年 5 月发布的一种消息传递接口。它实际上是一个消息传递函数库的标准说明,吸取了众多消息传递系统的优点,是目前国际上最流行的并行编程环境之一(其它的系统包括 P4、PVM、Express 和 PARMACS 等)。MPI 具有许多优点:具有可移植性和易用性;有完备的异步通信功能;有正

式和详细的精确定义,因而为并行软件产业的增长提供了必要的条件。

在基于 MPI 编程模型中,计算是由一个或多个彼此通过调用库函数进行消息收、发通信的进程所组成。在绝大部分 MPI 实现中,一组固定的进程在程序初始化时生成,一个处理器生成一个进程。这些进程可以执行相同或不同的程序(相应地称为 SPMD 或 MPMD 模式)。进程间的通信可以是点到点的,也可以是群体的(Collective)。或许 MPI 最重要的特性就是使用了称之为通信体(Communicator)的机构,允许程序员定义一种封装内部通信结构的模块。所谓通信体就是一个进程组(Process Group)加上进程活动环境(Context),其中进程组就是一组有限和有序进程集合。所谓有限意即组内包含有限数目的 n 个进程;所谓有序意即 n 个进程依次按 0,1,…,$n-1$ 整数定序(Ranked)。MPI 中的进程活动环境(Contexts),也称为上下文,是系统指定的超级标记(Supertag),它能安全地将彼此相互冲突的通信区分开来。每个通信体都有一个不同的、系统指定的进程活动环境,一条在一个进程活动环境中发送的消息不能在另一个进程活动环境中被接收。通信体的作用,请参阅习题 14.1。

14.2.1　最基本的 MPI

MPI 是个复杂的系统,它包含了 129 个函数(根据 1994 年发布的 MPI 标准)。事实上,1997 年修订的标准,称之为 MPI-2,已超过 200,目前最常用的也有约 30 个。然而我们可以只使用其中的 6 个最基本的函数就能编写一个完整的 MPI 程序去求解很多问题。这 6 个基本函数,包括启动和结束计算,识别进程以及发送与接收消息:

MPI_INIT:	启动 MPI 计算
MPI_FINALIZE:	结束 MPI 计算
MPI_COMM_SIZE:	确定进程数
MPI_COMM_RANK:	确定自己的进程标识符
MPI_SEND:	发送一条消息
MPI_RECV:	接收一条消息

函数的功能及其参数描述在图 14.2 中,其中标号 IN、OUT 和 INOUT 分别指明函数使用但不能修改参数、函数不使用但可修改参数以及函数既可使用也可修改参数。

```
MPI_INIT(int *argc,char ***argv)
Initiate a computation.
    argc,argv are required only in the C language binding,
        where they are the main program's arguments.
```

MPI_FINALIZE()
Shut down a computation.

MPI_COMM_SIZE(comm,size)
Determine the number of processes in a computation.

IN	comm	communicator(handle)
OUT	size	number of processes in the group of comm(integer)

MPI_COMM_RANK(comm,pid)
Determine the identifier of the current process.

IN	comm	communicator(handle)
OUT	pid	process id in the group of comm(integer)

MPI_SEND(buf,count,datatype,dest,tag,comm)
Send a message.

IN	buf	address of send buffer(choice)
IN	count	number of elements to send (integer≥0)
IN	datatype	datatype of send buffer elements(handle)
IN	dest	process id of destination process(integer)
IN	tag	message tag(integer)
IN	comm	communicator(handle)

MPI_RECV(buf, count,datatype,source,tag,comm,status)
Receive a message.

OUT	buf	address of receive buffer(choice)
IN	count	size of receive buffer,in elements(integer≥0)
IN	datatype	datatype of receive buffer,elements(handle)
IN	source	process id of source process,or MPI_ANY_SOURCE(integer)
IN	tag	message tag,or MPI_ANY_TAG(integer)
IN	comm	communicator(handle)
OUT	status	status object(status)

图14.2 基本的MPI函数说明

只使用6个基本函数中的4个即可写出如下一段MPI的Fortran程序：

```
program main
begin
    MPI_INIT()                                    /*启动计算*/
    MPI_COMM_SIZE(MPI_COMM_WORLD,count)
                                                  /*找进程数*/
    MPI_COMM_RANK(MPI_COMM_WORLD,myid)
                                                  /*找自己的id*/
    print*, I am ,myid, of ,count                 /*打印消息*/
    MPI_FINALIZE()                                /*结束计算*/
end
```

其中,MPI_COMM_WORLD是一个缺省的进程组,它指明所有的进程都参与计算。

MPI不是一个独立的、自包含的软件系统，MPI进程是重量级、单线程的进程，MPI标准并不指明如何启动并行计算，它可通过命令行参数指明应被生成的进程数目，然后按SPMD方式或MPMD方式执行程序。MPI函数库本身是与语言无关的，也就是说，库函数的描述可以使用C语言、Fortran语言或其它的语言。目前MPI库函数提供了C语言和Fortran语言描述（称为Language Binding）。在C语言描述中，函数名均冠以MPI前缀，且其首字母需大写。返回的状态值是整数；成功完成的返回代码是MPI_SUCCESS；失败时也会定义一组错误代码。编译时的常数均须大写且被定义在文件mpi.h中，mpi.h在任何需调用MPI的程序中必须被包含进来。句柄（Handles）由定义在mpi.h中的特殊类型所表示。具有类型IN的函数参数由值传送；具有类型OUT和INOUT的函数参数由引用传送（例如指针）。MPI的数据类型定义成C和Fortran语言的数据类型，如表14.1所示。

表14.1 MPI中的数据类型一览表

MPI(C Binding)	C	MPI (Fortran Binding)	Fortran
MPI_BYTE		MPI_BYTE	
MPI_CHAR	signed char	MPI_CHARACTER	CHARACTER (1)
		MPI_COMPLEX	COMPLEX
MPI_DOULBE	double	MPI_DOUBLE_PRECISION	DOUBLE _PRECISION
MPI_FLOAT	float	MPI_REAL	REAL
MPI_INT	int	MPI_INTEGER	INTEGER
		MPI_LOGICAL	LOGICAL
MPI_LONG	long		
MPI_LONG_DOUBLE	long double		
MPI_PACKED		MPI_PACKED	
MPI_SHORT	short		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long		
MPI_UNSIGNED_SHORT	unsigned short		

在Fortran语言描述中，函数名均冠以MPI前缀，且均须大写。函数返回代码由一个附加的整数变量表示之；成功完成的返回代码是MPI_SUCCESS；失败时

也会定义一组错误代码。编译时的常数均须大写且被定义在文件mpif.h中,它在任何需调用MPI的程序中必须被包含进来。所有句柄均具有类型INTEGER。MPI的数据类型如表14.1所示。

14.2.2 群体通信

MPI并行程序中经常需要一些进程组间的群体通信(Collective Communication),包括:①路障(Barrier):同步所有进程;②播送(Broadcast):从一个进程发送一条数据给所有进程;③收集(Gather):从所有进程收集数据到一个进程;④散播(Scatter):从一个进程散发多条数据给所有进程;⑤归约(Reduction):包括求和、求积等。这些函数的功能及其参数描述在图14.3中,播送、收集和散射的几何意义示于图14.4中。

MPI_REDUCE和MPI_ALLREDUCE均执行归约操作,它们组合每个进程中输入缓冲器中的值,返回组合后的值于单一根进程的输出缓冲器中(MPI_REDUCE)或者于所有进程的输出缓冲器中(MPI_ALLREDUCE)。组合所使用的操作包括最大和最小(MPI_MAX和MPI_MIN);求和与求积(MPI_SUM和MPI_PROD);逻辑与、逻辑或以及异或(MPI_LAND、MPI_LOR和MPI_LXOR);按位与、按位或以及按位异或(MPI_BAND、MPI_BOR和MPI_BXOR)。

```
MPI_BARRIER(comm)
Global synchronization.
  IN     comm        communicator(handle)

MPI_BCAST(inbuf,incnt,intype,root,comm)
Broadcast data from root to all processes.
  INOUT inbuf        address of input buffer, or output buffer at root (choice)
  IN     incnt       number of elements in input buffer (integer)
  IN     intype      datatype of input buffer elements (handle)
  IN     root        process id of root process (integer)
  IN     comm        communicator (handle)
MPI_GATHER(inbuf,incnt,intype,outbuf,outcnt,outtype,root,comm)

MPI_SCATTER(inbuf,incnt,intype,outbuf,outcnt,outtype,root,comm)

Collective data movement functions.
  IN     inbuf       address of input buffer (choice)
  IN     incnt       number of elements sent to each (integer)
  IN     intype      datatype of input buffer elements (handle)
  OUT    outbuf      address of output buffer  (choice)
```

IN	outcnt	number of elements received from each (integer)
IN	outtype	datatype of output buffer elements (handle)
IN	root	process id of root process (integer)
IN	comm	communicator (handle)

MPI_REDUCE (inbuf,outbuf,count,type,op,root,comm)

MPI_ALLREDUCE (inbuf,outbuf,count,type,op,comm)

Collective reduction functions.

IN	inbuf	address of input buffer (choice)
OUT	outbuf	address of output buffer (choice)
IN	count	number of elements in input buffer (integer)
IN	type	datatype of input buffer elements (handle)
IN	op	operation ;see text for list (handle)
IN	root	process id of root process (integer)
IN	comm	communicator (handle)

图 14.3 MPI群体通信函数说明

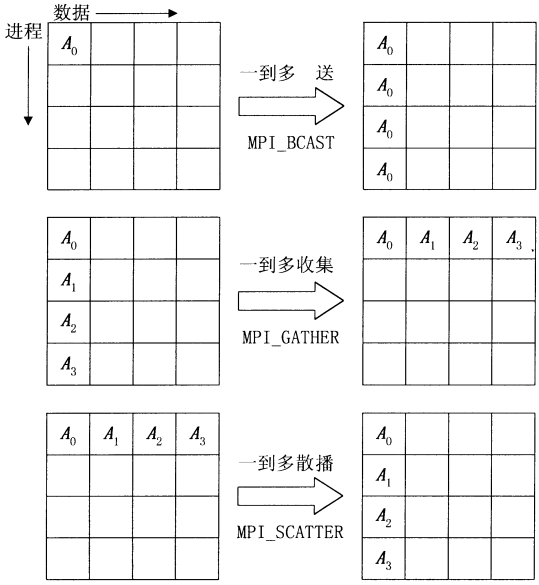


图 14.4 MPI播送、收集和散播的几何意义

14.2.3 通信体

通信体(Communicator)提供了MPI中独立的安全的消息传递。不同的通信库使用独立的通信体,它隔离了内部和外部通信,避免了在通信库被调用和退出时的同步,也保证了在同一通信体的通信操作互不干扰。

MPI提供的通信体函数概括在图14.5中:①MPI_COMM_DUP:它生成一个新的通信体,具有相同的进程组但新的上下文,这可确保不同目的通信不会混淆;②MPI_COMM_SPLIT:它生成一个新的通信体,但只是给定进程组的子集,这些进程可相互通信而不必害怕与其它并发计算相冲突;③MPI_INTERCOMM_CREATE:它构造一个进程组之间的通信体,该通信体链接两组内的进程;④MPI_COMM_FREE:它用来释放上述三个函数所生成的通信体。

```
MPI_COMM_DUP(comm,newcomm)
Create new communicator: same group, new context.
  IN    comm      communicator(handle)
  OUT   newcomm   communicator(handle)
MPI_COMM_SPLIT(comm,color,key,newcomm)
Partion group into disjoint subgroups.
  IN    comm      communicator(handle)
  IN    color     subgroup control(integer)
  IN    key       process id control(integer)
  OUT   newcomm   communicator(handle)

MPI_INTERCOMM_CREATE(comm,leader,peer,rleader,tag,inter)
Create an intercommunicator.
  IN    comm      local intracommunicator(handle)
  IN    leader    local leader(integer)
  IN    peer      peer intracommunicator(handle)
  IN    rleader   process id of remote leader in peer (integer)
  IN    tag       tag for communicator set up(integer)
  OUT   inter     new intercommunicator(handle)

MPI_COMM_FREE(comm)
Destroy a communicator.
  IN    comm      communicator(handle)
```

图14.5 MPI通信体函数

图14.6示出使用MPI_COMM_SPLIT生成新的通信体方法,其中第一个通信体是个8进程组,使用myid%3着色法并调用MPI_COMM_SPLIT(comm,color,myid,newcomm)就把原通信体劈成3个不相交的进程组。

图14.7示出了在两进程组之间建立了一个组间通信体,其中顶上是原8进程

组,这是一个 MPI_COMM_WORLD;调用 MPI_COMM_SPLIT 生成两个进程组,每个包含 4 个进程;最后调用 MPI_INTERCOMM_CREATE 生成两组之间的组间通信体。

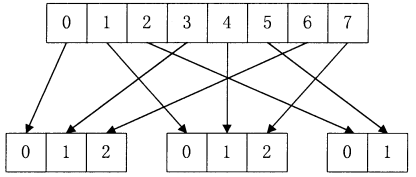


图 14.6　使用 MPI_COMM_SPLIT 生成新通信体

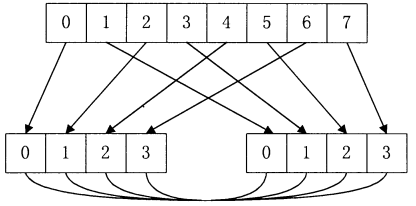


图 14.7　在两进程组之间建立组间通信体

14.2.4　导出数据类型

MPI 中的导出数据类型(Derived Datatype)允许将不连续的数据元素组合在一起,这种导出数据类型可以使用构造函数(Constructor Function)构造之。MPI 提供的导出数据类型函数概括在图 14.8 中:①MPI_TYPE_CONTIGUOUS:它用来定义由一个或多个连续数据元素组成的类型;②MPI_TYPE_VECTOR:它用来定义由一个或多个成块数据元素组成的类型,这些成块的数据元素是由数组中的恒间距所分开;③MPI_TYPE_INDEXED:它用来定义由一个或多个基本的或先前已定义数据类型的数据块组成的类型,其中块的长度和块间位移量由数组指定;④MPI_TYPE_COMMIT:它用来提交数据类型,必须在使用导出数据类型之前使用;⑤MPI_TYPE_FREE:它用来释放导出数据类型,必须在使用导出数据类型之后使用。这些函数的作用,读者可参照习题 14.2 和 14.3 进一步理解之。

MPI_TYPE_CONTIGUOUS(count,oldtype,newtype)
Construct datatype from contiguous elements.

IN	count	number of elements (integer ≥ 0)
IN	oldtype	input datatype (handle)
OUT	newtype	output datatype (handle)

MPI_TYPE_VECTOR(count,blocklen,stride,oldtype,newtype)
Construct datatype from blocks separated by stride.

IN	count	number of elements (integer≥0)
IN	blocklen	elements in a block (integer≥0)
IN	stride	elements between start of each block (integer)
IN	oldtype	input datatype (handle)
OUT	newtype	output datatype (handle)

MPI_TYPE_INDEXED(count,blocklens,indices,oldtype,newtype)
Construct datatype with variable indices and sizes.

IN	count	number of blocks (integer≥0)
IN	blocklens	elements in each block (array of integer≥0)
IN	indices	displacements for each block (array of integer)
IN	oldtype	input datatype (handle)
OUT	newtype	output datatype (handle)

MPI_TYPE_COMMIT(type)
Construct datatype so that it can be used in communication.

INOUT	type	datatype to be committed (handle)

MPI_TYPE_FREE(type)
Free a derived datatype.

INOUT	type	datatype to be freed (handle)

图14.8 MPI导出数据类型函数

14.2.5 点到点通信

点到点通信(Point to Point Communication)是MPI中比较复杂的一部分,其数据传送有阻塞(Blocking)和非阻塞(Non_Blocking)两种机制:对于阻塞方式,它必须等到消息从本地送出之后才可以执行后续的语句,保证了缓冲区等资源的可再用性;对于非阻塞方式,它不须等到消息从本地送出就可执行后续的语句,从而允许通信和计算的重叠,但非阻塞调用的返回并不保证资源的可再用性。

如图14.9所示,阻塞和非阻塞有四种通信模式:①标准模式,包括阻塞(标准)发送MPI_SEND、阻塞(标准)接收MPI_RECV、非阻塞(标准)发送MPI_ISEND和非阻塞(标准)接收MPI_IRECV;②缓冲模式,包括阻塞缓冲发送MPI_BSEND和非阻塞缓冲发送MPI_IBSEND;③同步模式,包括阻塞同步发送MPI_SSEND和非阻塞同步发送MPI_ISSEND;④就绪模式,包括阻塞就绪发送MPI_RSEND和非阻塞就绪发送MPI_IRSEND。在标准通信模式中,MPI根据当前的状况选取其它三种模式或用户定义的其它模式;缓冲模式在相匹配的接收未开始的情况下,总是将送出的消息放在缓冲区内,这样发送者可以很快地继续计算,然后由系统处理放在缓冲区中的消息,但这不仅占用内存,而且多用了一次内存拷

贝 在同步模式中,MPI必须保证接收者执行到某一点,这样接收者是必须有确认信息的 在就绪模式下,系统默认与其相匹配的接收已经调用。

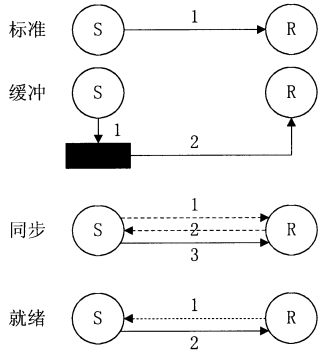


图 14.9　点到点消息通信四种模式

在点到点通信中,发送和接收语句必须是匹配的。为了区分不同进程或同一进程发送来的不同消息,在这些语句中采用了通信体 Comm 和标志位 tag 来实现成对语句的匹配。

对于阻塞的标准通信 MPI_SEND 和 MPI_RECV 函数的格式及其说明,请参见前图 14.2。其中发送的含义是将包含 count 个 datatype 类型的首地址为 buf 的消息发送至 dest 进程,该消息是与标志位 tag 和通信体 comm 封装在一块的 ;接收的含义是从 source 进程接收标志为 tag 和通信体为 comm 的消息,将该消息写入首地址为 buf 的缓冲区,其返回值 status 中包含消息的大小、标志和来源等信息。

对于非阻塞的标准通信 MPI_ISEND 和 MPI_IRECV 函数格式及其说明如下：

```
MPI_ISEND (buf,count,datatype,dest,tag,comm,request)
    IN      buf
    IN     count
    IN   datatype
    IN      dest        参照图 14.2 中 MPI_SEND 函数说明
    IN       tag
    IN      comm
   OUT    request
MPI_IRECV (buf,count,datatype,source,tag,comm,status,request)
```

OUT	buf	参照图 14.2 中 MPI_REW 函数说明
IN	count	
IN	datatype	
IN	dest	
IN	tag	
IN	count	
OUT	status	
OUT	request	

它们的含义与 MPI_SEND 和 MPI_RECV 函数基本上一致,因为它们是非阻塞的通信模式,语句结束后真正的消息发送或接收并没有完成,它仅将消息挂入消息队列中,所以用 request 指明返回队列指针。也正是因为非阻塞通信在语句结束后,真正的消息发送或接收并没有完成,所以必须用 MPI_WAIT 和 MPI_TEST 等语句来结束非阻塞通信,它们的格式及其说明如下:

MPI_WAIT (request,status)

IN	request
OUT	status

MPI_TEST (request,flag,status)

IN	request
OUT	flag
OUT	status

其中,MPI_TEST 用来检测非阻塞操作是否真正结束,flag=1 表示该请求 request 指向的非阻塞通信已完成;MPI_WAIT 则一直要等到非阻塞操作真正完成之后才返回。我们可以认为阻塞通信等于非阻塞通信加上 MPI_WAIT。

在点到点通信中,除了上述的函数外还须提供系统询问(Inquiry)和探测(Probe)两个函数,它们的操作概括在图 14.10 中:①MPI_IPROBE 函数检查未决(Pending)消息的存在而不必接收这些消息,从而允许我们将局部计算与处理将要到来的消息交织在一起编写程序,该调用将设置一个布尔变量 flag,以指明与指定的源、tag 和通信体相匹配的信息是否有效;②MPI_PROBE 函数与 MPI_IPROBE 相关,它阻塞一直到指定的源、tag 和通信体有效,然后返回并设置其 status 变量;③MPI_GET_COUNT 询问函数得到刚刚收到的消息的长度,其前两个参数分别是由上一个探测或 MPI_RECV 调用设置的状态目标和被接收的元素的数据类型,而第三个参数是一个用以返回所接收的元素数目的整数。

MPI_IPROBE (source, tag, comm, flag, status)
Poll for a pending message.

IN	source	id of source process, or MPI_ANY_SOURCE (integer)
IN	tag	message tag, or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	flag	(logical/Boolean)
OUT	status	status object (status)

MPI_PROBE (source, tag, comm, status)
Return when message is pending.

IN	source	id of source process, or MPI_ANY_SOURCE (integer)
IN	tag	message tag, or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (status)

MPI_GET_COUNT (status, datatype, count)
Determine size of a message.

IN	status	status variable from receive (status)
IN	datatype	datatype of receive buffer elements (handle)
OUT	count	number of data elements in message (integer)

图 14.10 MPI询问和探测操作

这些函数的用法可通过如下代码段理解之：

```
int count, *buf, source;
MPI_Probe(MPI_ANY_SOURCE, 0, comm, &status);
source = status.MPI_SOURCE;
MPI_Get_count(status, MPI_INT, &count);
buf = malloc(count * sizeof(int));
MPI_Recv(buf, count, MPI_INT, source, 0, comm, &status);
```

该段代码接收来自未知源和包含未知整数数目的消息：它首先用 MPI_PROBE 检测消息的到来，然后决定消息源，并用 MPI_GET_COUNT 决定消息的大小，最后分配一个合适大小的缓冲，并接收此消息。

到现在为止，已讨论了约 30 个 MPI 函数，它们的 C 语言说明和 Fortran 语言说明，综合在本章最后的附录一和附录二中。其中 allgather (全收集) 和 alltoall (多到多) 的含义和功能的解释，作为习题 14.3 留给读者完成。

最后，给出一个用 C 语言描述的 MPI 计算 π 的示范程序 (它是 Hostless 程序) 如下，以供读者学习理解之用。

```
//* 计算 π C 语言 MPI 编程代码段 *//
#include mpi.h
#include <stdio.h>
#include <math.h>
```

```
double f(a)
double a;
{
    return (4.0/(1.0+a*a));
}

int main(argc,argv)
int argc;
char*argv[];
{
    int done=0,n,myid,numprocs,i;
    double PI25DT=3.141592653589793238462643;
    double mypi,pi,h,sum,x,a;
    double startwtime,endwtime;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    n=0;
    while(!done)
    {
        if(myid==0)
        {
            printf("Enter the number of intervals:(0 quits)");
            scanf("%d",&n);
            if(n==0) n=100;
            else n=0;
            startwtime=MPI_Wtime();
        }
        MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
        if(n==0)
            done=1;
        else
        {
            h=1.0/(double)n;
```

```
            sum=0.0
            for (i=myid+1;i<=n;i+=numprocs)
             {
                x=h*((double)i-0.5);
                sum+=f(x);
             }
           mypi=h*sum;
           MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,
                      MPI_COMM_WORLD);
           if(myid==0)
              {
                printf("pi is approximately %.16f,Error is %.16f\n",
                       pi,fabs(pi-PI25DT));
                endwtime=MPI_Wtime();
                printf("wall clock time=%f\n",endwtime-startwtime);
              }
          }
       }
        MPI_Finalize();
    }return(0);
```

*14.3 PVM并行编程

并行虚拟机PVM(Parallel Virtual Machine)是由美国橡树岭国家实验室ORNL(Oak Ridge National Laboratory)于1989年开发的一种支持网络并行计算的支撑软件。后来,由Tennessee大学、Emory大学、CMU等大学共同研制,并得到美国能源部、国家科学基金会、Convex公司的资助,于1995年发布3.3版本。它是一个自包含的自由软件,源代码公开,从网上可以得到,因而颇受大学、研究机构的欢迎并得到广泛应用。它已成为很流行的大型科学计算的并行软件,可在同构/异构型网络环境下模拟实现一个通用的基于分布存储的并行计算机系统,提供基于消息传递的并行程序设计接口,使网上的用户能够集中地使用众多的机器资源来求解大规模计算问题。

PVM虽然没有MPI功能那么强大,但由于其出现较早,且是一个自包含系统(MPI不是自包含的),同时PVM不是一个标准(MPI是个标准),这就意味它较易修改,所以在大型计算应用领域中得到广泛使用。目前PVM和MPI正在彼此靠

拔,例如,MPI_2 (相对而言,以前所讨论的 MPI 可称为 MPI_1) 已加进了进程管理的功能,而 PVM 也有较多的群体通信功能。如果 PVM 和 MPI 最终能归并成一个单一的和标准的库,则无疑对并行处理是非常有益的。

14.3.1 PVM 概貌

PVM 组成 它由两部分组成:监控进程 (Daemon Process) 和 PVM 例程库。监控进程 (称之为 pvmd) 驻留在虚拟机的每一台节点机上 (PVM 称节点为主机),提供类似于系统调用的接口,实现进程 (PVM 称进程为任务) 控制、通信和错误检测;PVM 可调用库 (称之为 libpvm3.a) 提供并行程序设计所需的函数和过程,主要包括进程 (任务) 的创建与控制、进程 (任务) 间的通信、系统配置以及错误处理和动态进程组管理等。

PVM 控制台 (PVM Console) 在 PVM 安装 [74] 之后,用户从任何主机上键入下述命令就可生成 PVM 控制台:

```
pvm host_file
```

成功地执行此条命令,就可在调用的主机 (称为 master) 上启动 pvmd,而在其它各主机上列出一个任选 host_file 并在 master 主机上显示如下提示:

```
pvm>
```

它指明该主机处于 PVM 控制台模式。PVM 控制台是一个独立的交互的 PVM 进程,尤如 shell,用户键入一些命令就可管理虚拟机、调用 PVM 应用作业、监视作业的执行等。这些命令包括加入、删除主机和列出虚拟机配置,在虚拟机上启动运行多个任务和列出运行的作业以及 PVM 关机杀掉所有进程。

编译和运行 PVM 应用程序可用 C 语言或 Fortran 语言编写,它通过 PVM 提供的消息传递机制实现并行计算。因为 PVM 定义了许多常量和数据结构,为了使用 PVM 库函数,在 C 语言中要引入语句 include<pvm3.h>;在 Fortran 语言中要引入语句 include '$PVM_ROOT/include/fpvm3.h'。编好的程序可分别使用编译器 cc 或 f77 进行编译和链接。编译完后,如 PVM 是同构的则可将该目标代码拷贝到所有其它机器上;如 PVM 是异构的,则须将源代码拷贝到各个机器上重新编译。当所有工作完成后,在 master 主机上启动 pvmd,该 pvmd 进程将按用户的配置启动网络上其它节点上的 pvmd 进程,这样用户的应用程序就可在 PVM 上运行了。

14.3.2 PVM 消息传递库

PVM 库 (libpvm3.a) 提供并行程序设计所需的函数和过程,主要包括以下几类:

进程(任务)的创建与控制 ①PVM 进程通过调用 pvm_mytid() 来进入 PVM,同时获得自己的 TID;②通过调用函数 pvm_exit() 退出 PVM;③函数 pvm_spawn() 用于生成 pvm 的子进程;④pvm_parent() 用于返回任务的父进程的 TID。它们的格式如下:

```
int pvm_mytid();
int pvm_exit();
int pvm_parent();
int pvm_spawn(char *task,char **argv,int flag,char *where,
              int ntask,int *tids)
```

进程(任务)间的通信 ①PVM 通过调用 pvm_initsend() 函数来创建一个新的发送缓冲区,并将其置为活动缓冲区,参数 encoding 是数据打包时所使用的数据编码类型,包括 pvmDataDefault(若 PVM 是同构的,则不进行数据格式转换;否则要按外部数据表示 XDR 进行数据格式转换)、PvmDataRaw(不对数据进行任何编码)和 PvmDataInPlace(打包时数据仍放在原处);②发送缓冲区中可以同时包含各种不同类型、不同长度的数据,所以在数据发送之前必须对其打包,所使用的函数包括 pvm_pkint()、pvm_pkbyte()、pvm_pklong()、pvm_pkshort()、pvm_pkdouble()、pvm_pkfloat() 和 pvm_pkstr() 等;③函数 pvm_send() 用来将打好包的数据发送给其它的进程;④函数 pvm_recv() 用来接收数据,参数 tid 给出目标进程的 TID,msgtag 给出消息的标志;⑤接收时必须对所收到的数据进行与发送时打包相对应的拆包,所使用的函数包括 pvm_upkint()、pvm_upkbyte()、pvm_upklong()、pvm_upkshort()、pvm_upkdouble()、pvm_upkfloat() 和 pvm_upkstr() 等。它们的格式如下:

```
int pvm_initsend(int encoding)
int pvm_pkint(int *np,int nitem,int stride)
int pvm_send(int tid,int msgtag)
int pvm_recv(int tid,int msgtag)
int pvm_mcast(int tids,int nitem,int msgtag)
int pvm_upkint(int*np,int nitem,int stride)
```

动态进程(任务)组 PVM 支持进程(任务)组(Group)的概念。一个任务组中可包含多个任务;一个任务也可同时属于多个任务组。任务组可动态生成,任何任务可在任何时刻加入或离开一个任务组。任务组的创建有助于对组中任务进行广播、设置路障以及施行全局运算。任务组的引入,使得一个 PVM 任务既可以用 tid 来标识,也可以用(组名,成员号)二元组来标识。任务组操作中最基本的函数是加入和离开任务组:

```
int pvm_joingroup (char*group)
int pvm_lvgroup (char*group)
```

下面的函数可以从成员号 inum 获得 tid,或从 tid 获得成员号以及获得一个任务组中的成员数:

```
int pvm_gettid (char*group,int inum)
int pvm_getinst (char*group,int tid)
int pvm_gsize (char*group)
```

与任务组相关的其它操作,包括同步 pvm_barrier():调用任务等待组中共有 count 个任务调用了 pvm_barrier();广播 pvm_bcast():对任务组中的所有成员进行播送;归约 pvm_reduce():类似于 MPI 中的归约,用于在指定的任务组的全体成员进行全局性运算,如在同组的所有任务中求最大/最小、求和等。这些函数的格式如下:

```
int pvm_barrier (char*group,int count)
int pvm_bcast (char*group,int msgtag)
int pvm_reduce (void (*func)(),void*data,int nitem,int datatype,
                int msgtag,char*group,int root)
```

动态配置 PVM 可由用户调用 PVM 库函数进行动态配置。函数 pvm_addhosts() 和 pvm_delhosts() 用来加入和删除一个或多个主机:

```
int pvm_addhosts (char**hosts,int nhost,int *infos)
int pvm_delhosts (char**hosts,int nhost,int * infos)
```

它们将 hosts 指定的机器名加入到当前虚拟机中,或从当前虚拟机中删去这些机器。

最后,给出一个用 C 语言描述的 PVM 计算 π 的示范程序(它是一个 SPMD 程序)如下,以供读者学习理解之用。

```
//*计算 π C 语言 PVM 编程代码段*//
#define n 16/*number of tasks*/
#include pvm3.h
main(int argc,char **argv)
{
    int mytid,tids[n],me,i,N,rc,parent;
    double mypi,h,sum=0.0,x;
    me=pvm_joingroup("pi");
    parent=pvm_parent();
    if(me==0){
```

```
            pvm_spawn("pi",(char**)0,0,"",n-1,tids);
            printf("Enter the number of regions:");
            scanf("%d",&N);
            pvm_initsend(PvmDataRaw);
            pvm_pkint(&N,1,1);
            pvm_mcast(tids,n-1,5);
        }else{
            pvm_recv(parent,5);
            pvm_upkint(&N,1,1);
        }
        pvm_barrier("PI",n);/*optional*/
        h=1.0/(double)N;
        for(i=me+1;i<=N;i+=n){
            x=h*((double)i-0.5);
            sum+=4.0/(1.0+x*x);
        }
        mypi=h*sum;
        pvm_reduce(PvmSum,&mypi,1,PVM_DOUBLE,6,"PI",0);
        if(me==0)printf("pi is approximately %.16f\n",mypi);
        pvm_lvgroup("PI");
        pvm_exit();
}
```

14.4　基于数据并行的并行编程

14.4.1　数据并行模型的特点

关于数据并行模型已在第十三章的第13.2.2节进行了讨论，读者不妨先复习一下那里所讲的内容。

对于有些问题，常常要对大量的数据进行相同的、彼此独立的操作，这些量大且互不影响的操作是可以并行进行的，此即数据并行名称的由来。例如，矩阵相乘就呈现出数据并行的特点：当两个 $n\times n$ 的矩阵 A 和 B 相乘而得到矩阵 C 时，其乘积元素 C_{ij} 就是 A 的第 i 行与 B 的第 j 列执行点积运算而求得。所以求每个 C_{ij}

元素可以并行地对不同的数据元素 a_k 与 b_{kj} 执行相同的点积运算。如果我们用某种数据并行语言(下一节将要讨论)来编程实现这种计算,就可得到一个数据并行程序。一个数据并行程序包含一个单一的指令序列,每条指令作用于不同的数据项且以锁步方式同步执行,所以这种程序很自然地适合于在 SIMD 机器上实现。当然这样的数据并行程序也可在 MIMD 机器上实现,只不过是在每条指令之后施行全局同步导致执行效率很低。实用的办法是在 MIMD 机器上使用 SPMD 编程模式,在此模式下并行粒度要大一些。事实上,SPMD 是 SIMD 的特例,SPMD 是一种中粒度的并行,是语句块和循环一级的同步,而不是像 SIMD 那样是指令级的同步。在 SPMD 模式下,每个处理器异步地执行相同的程序,只有当处理器间需要交换数据时才施行同步。由于减少了同步,SPMD 模式在许多情况下比 SIMD 模式更有效。在大规模并行处理 MPP 系统中,使用 SPMD 模式进行编程时,用户程序把整个分布存储系统视为一个统一的全局空间,由编译实现程序的逻辑地址到物理地址的映射。

总之,数据并行程序是一个单一控制线程的、并行操作于不同数据项的、语句循环级同步的、单一地址空间的、通信可由变量指派而隐含地完成的一种并行程序。

14.4.2 数据并行编程的基本问题

进行数据并行程序设计时常常需要考虑以下几个基本问题:

数据分布 (Data Distribution) 分布就是要建立数据集合到处理器集合之间的映射关系,数据分布就是把数据按一定规则分配到各个处理器的存储单元中去。数据分布规则实际上也就是数据地址计算规则。分布的目的是为了提高程序执行的并行性,增强数据局部性,减少通信,从而提高程序的执行效率。数据并行程序设计语言及其编译系统所扩充的最主要功能之一就是支持数据分布。

数据分布规则是将共享分布的数组按维分配到各个处理器上。常用分布方法有按块分布、循环分布、对准分布和静态/动态分布等。这些方法在下节介绍 HPF 时会看到。

任务分配 (Task Allocation) 任务分配就是把计算任务分配给各个处理器上,使得程序可以在并行系统上高效执行。任务分配的原则是尽可能将计算和其所需数据对准,也就是数据分配到哪个处理器,就将用到这些数据的计算操作也分配到哪个处理器,此即所谓的属主-计算 (Owner-Compute) 的原则。在进行数据并行编程时,一般是先定义数据分布,再将并行循环中的迭代计算按数据分布模式分配到各个处理器上去执行,这样就实现了计算和数据的对准。

同步通信 对于 SPMD 模式数据并行编程,在共享变量模型下,同步通信

(Synchronous Communication)一般包含临界区、锁、路障和事件等;对于 SPMD 模式数据并行编程,编译系统给通信提供了较强的支持:对于点到点通信,程序员可以写成赋值语句,对于全局通信,程序员可以调用系统库程序。

处理器拓扑结构　在数据并行语言中,处理器的拓扑结构是逐个数组确定的,是受数据分布影响的。处理器拓扑的形成就是把参加并行执行的处理器分配给分布数组的各维,即决定每维分几个处理器。处理器拓扑结构可由编译系统确定,所依据的规则是数据分布大致均匀、地址计算尽量简单和通信尽可能少。

14.5　HPF 并行编程

高性能 Fortran(High Performance Fortran)简称 HPF,是一个支持数据并行的并行语言标准。它提供了一个全局名字空间和单线程控制,用户可以用分布(Distribution)和对准(Alignment)说明定义所希望的数据布局,用显式的并行结构表达并行机制。HPF 中由于没有显式的任务划分和通信语句,代码的定义是在高层进行的,因此具有良好的移植性,通过具体机器上的编译器可以生成各种系统平台上的高效代码。

HPF 的产生可以追溯到 DEC 的 Lovman 和 Rice 大学的 Kennedy 在 Supercomputing'91 大会上发起的"Birds of a feather"小组会。在这个会上主要讨论了关于分布存储机器上的高层编程方法。DEC 首先抛砖引玉,提供了他们的 HPF;紧接着于 1992 年 1 月在 Rice 大学召开了第一次 HPF 会议,会议决定组织 HPF 论坛;1992 年 3 月论坛正式成立;1993 年 5 月发布了 HPF1.0 版;1994 年 11 月发布了 HPF1.1 版,到 1997 年 1 月发布了 HPF2.0 版。所有文档可从网上站点 http://www.crpc.rice.edu/HPF/home.html 获得。

14.5.1　HPF 的语言特点

以下的讨论立足于 HPF2.0 版本。HPF2.0 语言包括三个部分:①语言的基本部分,包括任何 HPF 编译器必须支持的特性;②已经核准的扩展部分,包括满足某些特殊需要的高级结构(早期的编译器可能不支持这一部分);③已被认可的外部接口,这是 HPF 论坛批准的一组接口。

当前的 HPF2.0 版是基本 Fortran 95,并吸取了 HPF1.1 版的一些特性;而 HPF1.1 版是基于 Fortran 90 的,Fortran 90 的数组语句为数据并行提供了一种自然的机制,HPF1.1 扩展了 Fortran 90 的这些特性,提供了实现数据并行操作的

FORALL结构和几个内部与外部库。

HPF2.0的一些基本特性是：①数据并行制导（Data Parallel Directives）：HPF2.0提供的数据并行机制扩展了Fortran 95的数组语句和FORALL结构，其中INDEPENDENT用于声明无依赖循环迭代，可以并行执行；REDUCTION用于标识被不同迭代修改的变量；②数据映射制导（Data Mapping Directives）：HPF提供了一组扩展的说明，用于指定数据的分布和对准，可将数据显式地定义到抽象的处理器上，所提供的ALIGN和DISTRIBUTION说明，能够让用户告诉编译器如何在各处理器间分配数据，以使通信开销最小和负载划分最均匀；③新内部函数和库函数（New Intrinsic and Library Functions）：HPF提供了一组新的内部函数和库，包括请求底层硬件的系统函数、请求数据结构分布的映射请求函数以及一些计算的内部函数；HPF定义了一组新的库，为并行操作定义了一个标准的接口，包括归约（Reduction）、组合－散射（Combining_Scatter）、前缀（Prefix）及后缀（Suffix）、分类（Sorting）和位操作（Bit_Manipulation）函数等；④外部过程（Extrinsic Procedures）：除了数据并行编程外，HPF为了匹配其它编程方法，HPF定义了一个显式接口，允许用不同语言（如C）或不同方法（如显式消息传递）所编制的程序能够被一个HPF程序调用。

14.5.2 HPF的数据并行机制

FORALL结构　HPF中新增的FORALL语句和FORALL结构类似于Fortran 90的数0组赋值语句，只是比Fortran 90的数组赋值语句更灵活，并且FORALL语句能以指定的逻辑表达式对赋值语句进行限定。例如下述FORALL语句：

FORALL（i=2:5,X(i)＞0)　X(i)＝X(i−1)＋X(i+1)

其中，i是索引变量；i=2:5称为for_三元组，等价于i=2:5:1，表示i取值上界为5、下界为2而步长缺省时为1。在上述FORALL语句中，假定初始X=｛1,−1,2,−2,3,−3｝，在i的有效值范围内，满足X(i)＞0的索引i的活动集合为｛3,5｝，同时计算如下赋值语句：

X(3)＝X(2)＋X(4)＝−3

X(5)＝X(4)＋X(6)＝−5

FORALL语句结束后，X=｛1,−1,−3,−2,−5,−3｝。

在FORALL语句中，可能有不止一个for_三元组，则用的是组合索引，例如：

FORALL（i=1:2,j=1:3,Y(i,j)＞0)　Z(i,j)＝1／Y(i,j)

该语句等价于Fortran 90如下语句：

where(Y(1:2,1:3)＞0)　Z(1:2,1:3)＝1／Y(1:2,1:3)

则组合索引的有效值集合为{(1,1)，(1,2)，(1,3)，(2,1)，(2,2)，(2,3)}，而组合索引的活动值取上述集合中使 $Y(i,j)>0$ 的子集。

有时用户希望在一个 FORALL 语句中包含几个赋值，这可使用 FORALL 结构来实现。FORALL 结构是对 FORALL 语句的进一步扩充，即在 FORALL 和 ENDFORALL 之间可以写多条语句，但限制 FORALL 结构中只能使用赋值语句、FORALL 语句、FORALL 结构、WHERE 语句以及 WHILE 结构。例如：

```
FORALL (I=2:9)
    A(I) = A(I-1) + A(I+1)
    B(I) = A(I)
ENDFORALL
```

上述代码，首先对从 2 到 9 的各个 I 求 A(I-1) + A(I+1) 之值，并将结果送入 A(2) 到 A(9) 中，然后将求得的 A(2) 到 A(9) 之值送入 B(2) 到 B(9) 中。

数据分布　数据分布指的是将数据（主要是数组）分布到处理器上。HPF 引入了如图 14.11 所示的一些指导数据分布的伪指令，指示编译器如何最佳地分布数据于计算节点上。图 14.12 是 HPF 数据分布模型。数据映射（分布）要达到的目的是：使处理器间的通信开销最小以及使负载在处理器上分布均匀。

```
!HPF$ PROCESSORS proc_name(dim1,…,dimN)
Declare and abstract processor array.
      proc_name          name of abstract processor array
      dim1,…,dimN        size and shape of array

!HPF$ ALIGN array WITH target
Align array with a target array.
      array              array to be aligned
      target             array to be aligned to

!HPF$ DISTRIBUTE list_of_arrays [ONTO proc_name]
Distribute array(s) onto processor array.
      list_of_arrays     arrays to be distributed
      proc_name          abstract processor array
```

图 14.11　HPF 数据分布伪指令

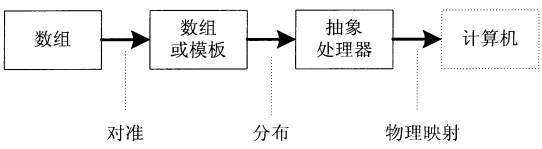


图 14.12　HPF 数据分布模型

(1) PROCESSORS:伪指令PROCESSORS说明抽象的处理器排列结构。例如：

!HPF$ PROCESSORS N(4)

它通知编译器,应用程序希望使用4个处理器排成线性阵列的抽象结构。同样,

!HPF$ PROCESSORS P(4,5)

!HPF$ PROCESSORS Q(4,5,6)

分别指明20个处理器排成4×5的二维网孔和120个处理器排成4×5×6的三维网孔。

(2) ALIGN:伪指令ALIGN用于描述数组间元素的对准。例如：

!HPF$ ALIGN A(I) WITH B(I)

表示把A的第I个元素与B的第I个元素分配到同一个处理器上。同样,

!HPF$ ALIGN A(I) WITH B(I+2)

表示把A的第I个元素与B的第I+2个元素分配到同一个处理器上。图14.13示出了HPF对准语句的6个例子。

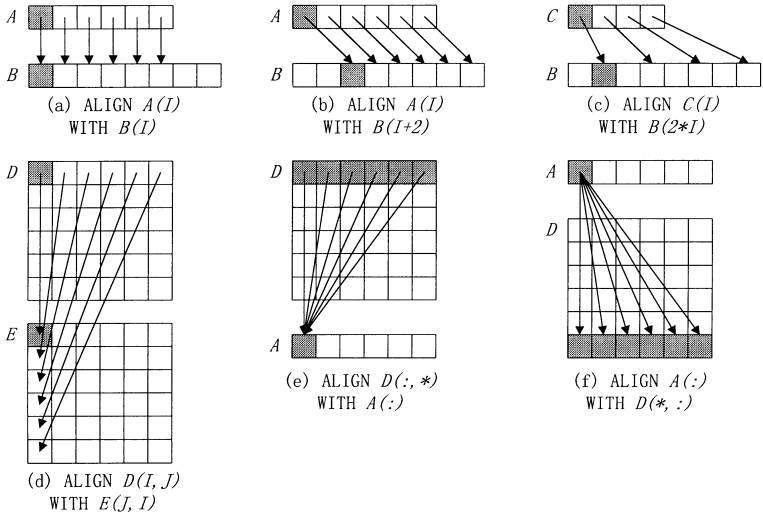


图14.13 HPF对准方式示例

(3) DISTRIBUTE:伪指令DISTRIBUTE用于指明数组如何在计算机存储器间进行划分。数组每一维可按下述方式之一进行分布：

*	无分布
BLOCK(n)	块状分布
CYCLIC(n)	循环分布

对于程序员可使用下述两种 DISTRIBUTE 伪指令：

```
!HPF$ DISTRIBUTE A(BLOCK) ONTO N
!HPF$ DISTRIBUTE C(CYCLIC) ONTO N
```

图 14.14 示出了一个 8×8 的二维数组，使用分块和循环方式分布到 4 个处理器上的情况。图中带阴影的数据分布给第一个处理器。

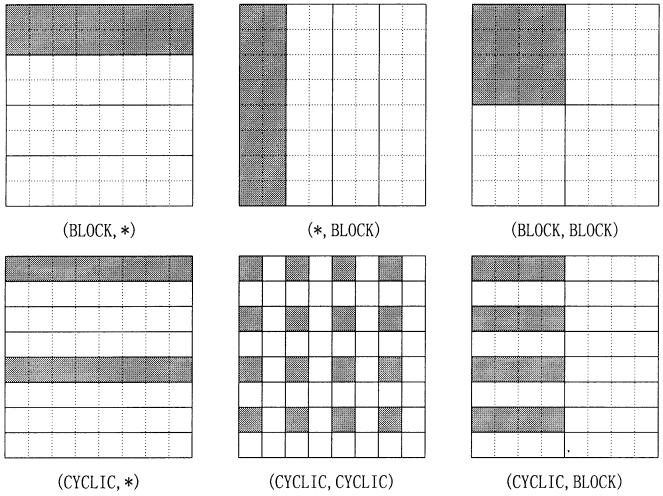


图 14.14　8×8 的二维数组分布给 4 个处理器

下面给出 HPF 数据映射的一个综合例子。试考虑如下 HPF 代码段：

```
      integer A(100),B(100),C(101),i
!HPF$      ALIGN A(i) WITH B(i-1)
!HPF$      PROCESSOR N(4)
!HPF$      DISTRIBUTE A(BLOCK) ONTO N
!HPF$      DISTRIBUTE C(CYCLIC) ONTO N
      FORALL (i=2:100)
          A(i) = A(i) + B(i-1)
          C(i) = C(i-1) + C(i) + C(i+1)
      ENDFORALL
```

图 14.15 描述了上述代码段实现的数据映射过程。HPF 中数据映射的实现分为两个阶段：逻辑映射和物理映射。前者由用户通过编译制导（Compiler Directives）定义，将数据映射到抽象处理器节点上；后者由系统将上述抽象处理器

节点映射到实际计算机物理节点上,这对用户是透明的。其中逻辑映射分为对准和分布两步。

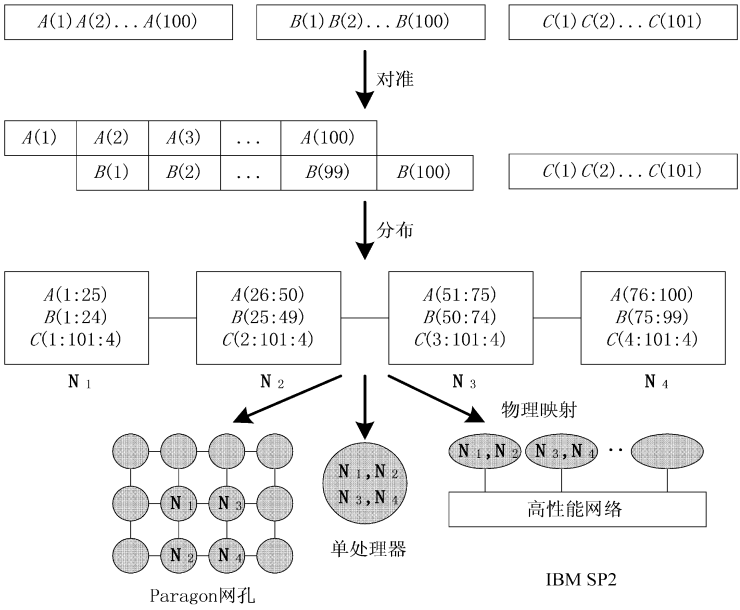


图 14.15 HPF数据映射过程

外部过程 HPF中增加了具有如下功能的库程序:①系统查询功能:它可给出有关处理器的信息,包括可用处理器数的NUMBER_OF_PROCESSOR库程序以及处理器构成信息的PROCESSOR_SHAPE库程序;②变换查询功能:它可给出有关数据的分布、定位方面的信息,包括数据定位信息的HPF_ALIGNMENT_库程序,template信息的HPF_TEMPLATE库程序和数据分布状态信息的HPF_DISTRIBUTION库程序;③计算库功能:它提供求数组中的最大值和前缀运算等。这些运算使用具有网络特性的库函数比起使用以Fortran描述的循环效果更好。在HPF中提供的库函数有归约函数、前缀及后缀函数、组合散射函数、分类函数和位变换函数等。

14.5.3 HPF使用中的若干问题

语言的限制问题 在Fortran中,数组元素按一定的顺序被分配在一维连续

的存储区内,并可通过等价语句 EQUIVALENCE 和子程序调用中的虚实变量结合发生连接。因此会发生部分一维数组被二维数组覆盖以及某变量的一部分和另一变量的一部分拥有同一存储区的情况。在 HPF 程序中也会发生类似情况(参见习题 14.6)。另外,关于子程序调用时的虚实变量结合也受到限制。例如在 Fortran 90 中,能作到用数组元素的实变量代换虚变量,用二维数组代换一维数组。但在 HPF 中这样作就需要使用 SEQUENCE 伪指令写出程序,例如:

```
      REAL A(100,100),B(200)
!HPF$ SEQUENCE A,B
      CALL S(A,B(51))
      END
      SUBROUTINE S(X,Y)
      REAL X(10000),Y(100)
!HPF$ SEQUENCE X,Y
      …
      END
```

并行性问题　因为 HPF 是单线程的,所以一个数据并行的程序逻辑上只有一个进程,尽管多个节点能够对不同的数据子域执行相同的程序。绝大部分并行性问题由系统管理,用户不用担心进程的生成、结束和有多少进程正在运行程序。因为 HPF 是语句级的数据并行,它允许在同一时间不同的节点执行不同的指令,因此可同时支持 SIMD 和 SPMD 并行计算,但不支持 MPMD 计算。程序中的并行度是随所用的不同的数组指派或执行不同的 FORALL 语句而动态变化。

相互作用问题　因为是松散同步,所以在数据并行编程中绝大多数相互作用问题都躲开了,除了归约操作外,Fortran 90 或 HPF 中不存在相互作用问题。

在 Fortran 90 中提供了 9 种内部函数以支持归约,但不支持前缀、递降、分类,它也不支持用户定义的归约操作。HPF 通过一组库例程扩展了 Fortran 90 的能力,包括辅助的归约函数、scan 函数和分类等。

数据并行程序只有一种相互作用模式,即语句级的松散同步。一条语句的所有操作均必须在下一语句的任一操作开始之前完成。通信经由数组赋值而呈隐式状。HPF 提供一组数组分布伪指令,使用开拓数据局部性而可最小化通信开销。HPF 除了提供 PROCESSORS、ALIGN 和 DISTRIBUTE 伪指令外,还提供 REALIGN 和 REDISTRIBUTE 语句以调节通信模式的变化。

语义问题　由于单线程和松散同步,所以数据并行程序具有清晰的语义,它不

可能出现死锁或活锁,仅有的可能是由于无限循环而会造成非确定性。Fortran 90和HPF在结构性、合成性和正确性方面类似于串行的Fortran 90程序,并未因引入并行性而增加复杂性。

数据并行程序只要满足单赋值规则它就是确定的。

可编程性问题　因为Fortran和HPF均是高级语言标准,所以具有好的可移植性。又因为大多数并行性、相互作用和语义问题均由系统负责,所以Fortran 90和HPF易于使用。但现在的Fortran 90和HPF语言说明对通用性和效率尚有严格的限制,所以不少人仍在使用别的编程方法。Fortran 90和HPF不支持开拓控制并行,它们也不适合于支持异步迭代、流水线计算等算法范例,数据库方面的应用也比较困难。HPF的数据映射似乎也仅支持正规通信模式的数组,它是否有效地支持一般数据结构和非正规通信模式的并行算法,还值得怀疑。

最后,给出一个高斯消去法的HPF程序如下,以供读者学习理解之用(其Fortran 90程序见习题14.7)。

```
//*高斯消去法的HPF编程代码段*//
parameter n=32
real A(n,n+1),x(n)
integer i,pivot_location(1)
!HPF$ PROCESSOR Nodes(4)
!HPF$ ALIGN x(i) WITH A(i,j)
!HPF$ DISTRIBUTE A(BLOCK,*) ONTO Nodes
do i=1,n-1
   !pivoting
  pivot_location=MAXLOC(ABS(A(i:n,i)))
  swap(A(i,i:n+1),A(i-1+pivot_location(1),i:n+1))
   !triangularization
  A(i,i:n+1)=A(i,i:n+1)/A(i,i)
  FORALL (j=i+1:n,k=i+1:n+1) A(j,k)=A(j,k)-A(j,i)*A(i,k)
end do
!back substitution
do i=n,1,-1
  x(i)=A(i,n+1)
  A(1:i-1,n+1)=A(1:i-1,n+1)-A(1:i-1,i)*x(i)
end do
```

14.6 小结和导读

小结 分布存储系统包括 SIMD_DM、MIMD_DM 和 COW_DM 等并行计算机系统。在 SIMD_DM 并行系统上,使用数据并行程序设计模型,以 SPMD 编程方式编写并行程序,常用的编程语言有 Fortran 90 和 HPF 等;在 MIMD_DM 和 COW_DM 并行系统上,使用消息传递程序设计模型,以 SPMD 和 MPMD 编程方式编写并行程序,常用的语言标准有 MPI 和 PVM 等。

数据并行语言标准除了本章所介绍的 HPF 和 Fortran 90 外,目前还有 Fortran 95、Fortran 2001 和 Fortran D、Vienna Fortran。其中 Fortran 95 和 Fortran 2001 是 Fortran 90 的推广;Fortran D 和 Vienna Fortran 是 MPI 的推广。Fortran 95 发布于 1997 年,主要新特性是 FORALL 语言和 FORALL 结构,PURE 和 ELEMENTAL 过程以及结构和指针缺省初始化。Fortran 2001 预计 2001 年发布,正被考虑中的新特性包括例外处理,与 C 语言的连接性,增强导出数据类型,附加支持高性能数值计算,新的 I/O 性质,面向对象等。Fortran D (后称为 Fortran 90D) 和 Vienna Fortran 提供更通用的数据映射函数,包括支持动态和非正规数据分布等。

消息传递标准除了本章所介绍的 1994 年发布的 MPI (称之为 MPI_1 外),1997 年 MPI 论坛又发布了修订标准 MPI_2。MPI_2 相对于 MPI_1 增加了很多功能,主要包括动态进程 (MPI_1 并行程序中的进程是静态的),点到点单边通信 (MPI_1 点到点通信是双边的),可扩放 I/O 支持 (MPI_1 完全不考虑 I/O 问题),Fortran 90 和 C++ 语言描述 (MPI_1 的语言描述是 Fortran 77 和 C),增加实时处理,推广 MPI_1 的外部接口和环境工具,允许非阻塞群体通信、通信体间的群体通信、通信体间的拓扑结构 (MPI_1 只支持阻塞和通信体内的群体通信) 等。

导读 有关 MPI 的最初描述见 [128],一本好的 PVM 入门见 [74]。关于网络并行计算与分布式编程环境,读者可参阅 [200]。数据并行方法见 [92],HPF 描述在 [194,111,125],推广的 HPF 见 [45],Fortran 90 描述在 [1],Fortran 95 描述在 [2],Fortran D 描述见 [70],Vienna Fortran 描述在 [188]。关于并行程序设计的一本简明教材是 [194,199]。关于消息传递程序设计和数据并行程序设计,读者可阅读 [103] 书的第 13 章和第 14 章。

习　题

14.1　试考虑下述代码段中通信体的使用：

```
process 0：
MPI_Send(msg1,count1,MPI_INT,tag1,comm1)；
parallel_fft(…)；
process 1：
MPI_Recv(msg1,count1,MPI_INT,tag1,comm1)；
parallel_fft(…)；
```

(1) 试分析上述代码段的计算功能。

(2) 如果在 parallel_fft(…) 中又包含了另一个发送程序：

```
if(my_rank==0) MPI_Send(msg2,count1,MPI_INT,1,tag2,comm2)；
```

如果没有通信体则会发生什么情况？

14.2　填上空白处，使下述两代码段完全等效：

(1)
```
Call MPI_TYPE_CONTIGUOUS(10,MPI_REAL,tenrealtype,ierr)
Call MPI_TYPE_COMMIT(tenrealtype,ierr)
Call MPI_SEND(data,1,tenrealtype,dest,tag,MPI_COMM_WORLD,ierr)
Call MPI_TYPE_FREE(tenrealtype,ierr)
```

(2)
```
Call MPI_SEND(data,_____,_____,dest,tag,_____,ierr)
```

14.3　填上空白处，使下述两代码段完全等效：

(1)
```
float data[1024]；
MPI_Datatype floattype；
MPI_Type_vector(10,1,32,MPI_FLOAT,&floattype)；
MPI_Type_commit(&floattype)；
MPI_Send(data,1,floattype,dest,tag,MPI_COMM_WORLD)；
MPI_Type_free(&floattype)；
```

(2)
```
float data[1024],buff[10]；
for(_____;_____;i++) buff[i]=data[_____]；
MPI_Send(buff,_____,MPI_FLOAT,_____,_____,_____)；
```

14.4　在 MPI 群体通信中，还有 allgather 和 alltoall 两个函数，它们可图示于图 14.16 中，试解释两者的功能及意义。

14.5　下面是 PVM 环境下的 hello 程序，它是一个 host/node 程序，试分析其工作过程。

```
//*PVM 主机/节点编程的 hello 代码段*//
/*host 程序 hello.c*/
```

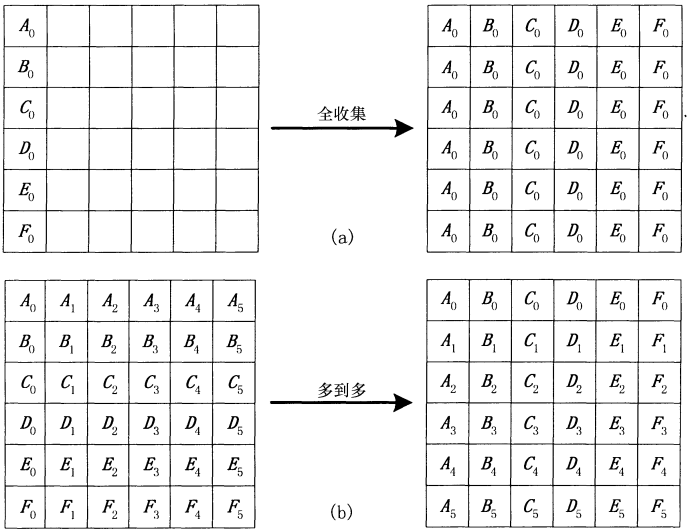


图 14.16 allgather和 alltoall群体操作

```
#include <stdio.h>
#include pvm3.h
main()
{
    int cc,tid;
    char buf[100];
    printf("i'm t%x\n",pvm_mytid());
    cc=pvm_spawn("hello_other",(char**)0,0,"",1,&tid);
    if(cc==1){
        cc=pvm_recv(-1,-1);
        pvm_bufinfo(cc,(int*)0,(int*)0,&tid);
        pvm_upkstr(buf);
        printf("from t%x:%s\n",tid,buf);
    }else
        printf("can't start hello_other\n");
    pvm_exit();
}
/*node程序 heuo_other.c*/
#include pvm3.h
```

```
#include string.h
main(){
 {
        int ptid;
        char buf[100];
        ptid=pvm_parent();
        strcpy(buf, "hello,world from ");
        gethostname(buf+strlen(buf),64);
        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(ptid,1);
        pvm_exit();
        exit(0);
}
```

14.6　试分析下述 HPF 代码段中数组元素的覆盖情况，如何用 EQUIVALENCE 语句使其变成正确的 HPF 程序？

```
        REAL A(100),B(100)
        EQUIVALENCE(A(51),B(1))
!HPF$   PROCESSORS P(4)
!HPF$   DISTRIBUTE A(Block) ONTO P
```

14.7　高斯消去法的 Fortran 90 程序如下：试分析其计算过程并与高斯消去法的 HPF 程序（课文中）加以比较。

```
//*高斯消去法的 Fortran 90 编程代码段*//
real A(n,n+1),x(n)
integer i,pivot_loctation(1)
do i=1,n-1
   !pivoting
   pivot_location=MAXLOC(ABS(A(i:n,i)))
   swap(A(i,i:n+1),A(i-1+pivot_location(1),i:n+1))
   !triangularization
   A(i,i:n+1)=A(i,i:n+1)/A(i,i)
   A(i+1:n,i+1:n+1)=A(i+1:n,i+1:n+1)-&
   SPREAD(A(i,i+1:n+1),1,n-i)*SPREAD(A(i+1:n,i),2,n-i+1)
end do

!back substitution
do i=n,1,-1
```

```
    x(i) = A(i,n+1)
    A(1:i-1,n+1) = A(1:i-1,n+1) - A(1:i-1,i) * x(i)
end do
```

14.8　雅可比松弛法 Fortran 90 程序如下：参照它试写出雅可比松弛法的 HPF 程序；并将两者加以比较。

```
//*雅可比松弛法的 Fortran 90 编程代码段*//
parameter n=1024
real A(n,n),x(n),b(n),error,Temp(n),DiagA(n)
integer i
error=SomeLargeValue
初始化 A、x 和 b
do i=1,n
    DiagA(i) = A(i,i)
end do
do while(error>ErrorBound)
S1:    Temp=(b-MATMUL(A,x))/DiagA
S2:    x=Temp+x
S3:    error=SUM(Temp*Temp)
end do
```

14.9　雅可比松弛法 MPI程序如下：参照它试写出雅可比松弛法 PVM 程序；并将两者加以比较。

```
//*雅可比松弛法的 MPI编程代码段*//
float Arow[N],X[N],NewX,B,error,Temp;
MPI_Comm comm;
error=SomeLargeValue;
初始化 A、x 和 B
MPI_Init(&argc,&argv);
comm=MPI_COMM_WORLD;
MPI_Comm_rank(comm,&my_rank);
while(error>SomeErrorBound){
    Temp=B;
    for(i=0;i<N;i++) Temp-=Arow[i]*X[i];
    Temp/=A[my_rank];
    NewX=Temp+X[my_rank];
    MPI_Allgather(&NewX,1,MPI_FLOAT,&X,1,MPI_FLOAT,comm);
    Temp=Temp*Temp;
    MPI_Allreduce(&Temp,&error,1,MPI_FLOAT,MPI_SUM,comm);
```

```
}
```

14.10　计算π可以通过MPI程序。为了加深读者对MPI的学习和理解，下面给出用FORTRAN90语言的实现版本。试分析其工作过程。

```
    //*计算π的MPI(F90)编程代码段*//
program main
use mpi
double precision PI25DT
parameter (PI25DT=3.14159265358979323846264 3d0)
double precision mypi,pi,h,sum,x,f,a
integer n,myid,numprocs,i,rc
! function to integrate
f(a)=4.d0/(1.d0+a*a)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)
print*, process ,myid, of ,numprocs, is alive
sizetype=1
sumtype=2
do
  if(myid.eq.0)then
    write(6,98)
98  format(' Enter the number of intervals:(0 quits) ')
    read(5,99) n
99  format(i10)
  endif
  call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
! check for quit signal
  if(n.le.0) exit
! calculate the interval size
  h=1.0d0/n
  sum=0.0d0
  do i=myid+1,n,numprocs
    x=h*(dble(i)-0.5d0)
    sum=sum+f(x)
  enddo
  mypi=h*sum
! collect all the partial sums
  call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,&
```

```
                                MPI_COMM_WORLD,ierr)
      ! process 0 prints the answer.
        if (myid.eq.0) then
            write (6,97) pi,abs(pi-PI25DT)
97          format(  pi is approximately:  ,F18.16,&    Error is:  ,F18.16)
        endif
      enddo
      call MPI_FINALIZE(rc)
      stop
      end
```

14.11　Fortran 90 计算π的程序如下：参照它试写出计算π的 HPF 程序，并将两者加以比较。

```
//*计算π Fortran 90 编程代码段*//
1.      INTEGER,PARAMETER::N=131072
2.      INTEGER,PARAMETER::LONG=SELECTED_REAL_KIND(13,99)
3.      REAL(KIND=LONG) PI,WIDTH
4.      INTEGER,DIMENSION(N)::ID
5.      REAL(KIND=LONG),DIMENSION(N)::X,Y
6.      WIDTH=1.0_LONG/N
7.      ID=/(I,I=1,N)/
8.      X=(ID-0.5)*WIDTH
9.      Y=4.0/(1.0+X*X)
10.     PI=SUM(Y)*WIDTH
11.     FORMAT( ESTIMATION OF PI WITH  ,I6,&
12.      INTERVALS IS  ,F14.12)
13.     PRINT 10,N,PI
14.     END
```

附录一　MPI 的函数的 C 语言说明

```
double MPI_Wtime(void)
int MPI_Init(int *argc,char***argv)
int MPI_Finalize(void)

int MPI_Send(void*buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm)
int MPI_Recv (void*buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI
```

```
_Status * status)
int MPI_Isend (void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_
    Request * request)
int MPI_Isend (void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_
    Request * request)
int MPI_Wait (MPI_Request * request, MPI_Status * status)
int MPI_Test (MPI_Request * request, int * flag, MPI_Status * status)
int MPI_Iprobe (int source, int tag, MPI_Comm comm, int * flag, MPI_Status * status)
int MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status * status)
int MPI_Get_count (MPI_Status * status, MPI_Datatype datatype, int * count)

int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype * newtype)
int MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *
    newtype)
int MPI_Type_indexed (int count, int * array_of_blocklengths, int * array_of_displacements, MPI_
    Datatype oldtype, MPI_Datatype * newtype)
int MPI_Type_commit (MPI_Datatype * datatype)
int MPI_Type_free (MPI_Datatype * datatype)

int MPI_Barrier (MPI_Comm comm)
int MPI_Bcast (void * buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
int MPI_Gather (void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int recvcount, MPI
    _Datatype recvtype, int root, MPI_Comm comm)
int MPI_Scatter (void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int recvcount, MPI
    _Datatype recvtype, int root, MPI_Comm comm)
int MPI_Allgather (void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int recvcount, MPI
    _Datatype recvtype, MPI_Comm comm)
int MPI_Alltoall (void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int recvcount, MPI
    _Datatype recvtype, MPI_Comm comm)
int MPI_Reduce (void * sendbuf, void * recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root,
    MPI_Comm comm)
int MPI_Allreduce (void * sendbuf, Void * recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI
    _Comm comm)

int MPI_Comm_size (MPI_Comm comm, int * size)
int MPI_Comm_rank (MPI_Comm comm, int * rank)
int MPI_Comm_dup (MPI_Comm comm, MPI_Comm * newcomm)
int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm * newcomm)
int MPI_Comm_free (MPI_Comm * comm)
int MPI_Intercomm_create (MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int
    remote_leader, int tag, MPI_Comm * newintercomm)
```

附录二 MPI的函数的Fortran语言说明

```
DOUBLE PRECISION MPI_WTIME()
MPI_INIT(IERROR)
    INTEGER IERROR
MPI_FINALIZE(IERROR)
    INTEGER IERROR

MPI_SEND(BUF,COUNT,DATATYPE,DEST,TAG,COMM,IERROR)
    <type> BUF(*)
INTEGER COUNT,DATATYPE,DEST,TAG,COMM,IERROR
MPI_RECV(BUF,COUNT,DATATYPE,SOURCE,TAG,COMM,STATUS,IERROR)
    <type> BUF(*)
    INTEGER COUNT,DATATYPE,SOURCE,TAG,COMM,STATUS(MPI_STATUS_SIZE),IERROR
MPI_ISEND(BUF,COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,IERROR)
    <type> BUF(*)
    INTEGER COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,IERROR
MPI_IRECV(BUF,COUNT,DATATYPE,SOURCE,TAG,COMM,REQUEST,IERROR)
    <type> BUF(*)
    INTEGER COUNT,DATATYPE,SOURCE,TAG,COMM,REQUEST,IERROR
MPI_WAIT(REQUEST,STATUS,IERROR)
    INTEGER REQUEST,STATUS(MPI_STATUS_SIZE),IERROR
MPI_TEST(REQUEST,FLAG,STATUS,IERROR)
    LOGICAL FLAG
    INTEGER REQUEST,STATUS(MPI_STATUS_SIZE),IERROR
MPI_IPROBE(SOURCE,TAG,COMM,FLAG,STATUS,IERROR)
    LOGICAL FLAG
    INTEGER SOURCE,TAG,COMM,STATUS(MPI_STATUS_SIZE),IERROR
MPI_PROBE(SOURCE,TAG,COMM,STATUS,IERROR)
    INTEGER SOURCE,TAG,COMM,STATUS(MPI_STATUS_SIZE),IERROR
MPI_GET_COUNT(STATUS,DATATYPE,COUNT,IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE),DATATYPE,_COUNT,IERROR

MPI_TYPE_CONTIGUOUS(COUNT,OLDTYPE,NEWTYPE,IERROR)
    INTEGER COUNT,OLDTYPE,NEWTYPE,IERROR
MPI_TYPE_VECTOR(COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR)
    INTEGER COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR
MPI_TYPE_INDEXED(COUNT,          ARRAY_OF_BLOCKLENGTHS,
```

```
    ARRAY_OF_DISPLACEMENTS,OLDTYPE,NEWTYPE,IERROR)
    INTEGER COUNT,ARRAY_OF_BLOCKLENGTHS(*),ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE,NEWTYPE,IERROR
MPI_TYPE_COMMIT(DATATYPE,IERROR)
    INTEGER DATATYPE,IERROR
MPI_TYPE_FREE(DATATYPE,IERROR)
    INTEGER DATATYPE,IERROR

MPI_BARRIER(COMM,IERROR)
    INTEGER COMM,IERROR
MPI_BCAST(BUFFER,COUNT,DATATYPE,ROOT,COMM,IERROR)
    <type> BUFFER(*)
    INTEGER COUNT,DATATYPE,ROOT,COMM,IERROR
MPI_GATHER(SENDBUF,SENDCOUNT,SENDTYPE,RECVBUF,RECVCOUNT,RECVTYPE,
    ROOT,COMM,IERROR)
    <type> SENDBUF(*),RECVBUF(*)
    INTEGER SENDCOUNT,SENDTYPE,RECVCOUNT,RECVTYPE,ROOT,COMM,IERROR
MPI_SCATTER(SENDBUF,SENDCOUNT,SENDTYPE,RECVBUF,RECVCOUNT,RECVTYPE,
    ROOT,COMM,IERROR)
    <type> SENDBUF(*),RECVBUF(*)
    INTEGER SENDCOUNT,SENDTYPE,RECVCOUNT,RECVTYPE,ROOT,COMM,IERROR
MPI_ALLGATHER(SENDBUF,SENDCOUNT,SENDTYPE,RECVBUF,RECVCOUNT,
    RECVTYPE,COMM,IERROR)
    <type> SENDBUF(*),RECVBUF(*)
    INTEGER SENDCOUNT,SENDTYPE,RECVCOUNT,RECVTYPE,COMM,IERROR
MPI_ALLTOALL(SENDBUF,SENDCOUNT,SENDTYPE,RECVBUF,RECVCOUNT,RECVTYPE,
    COMM,IERROR)
    <type> SENDBUF(*),RECVBUF(*)
    INTEGER SENDCOUNT,SENDTYPE,RECVCOUNT,RECVTYPE,COMM,IERROR
MPI_REDUCE(SENDBUF,RECVBUF,COUNT,DATATYPE,OF,ROOT,COMM,IERROR)
    <type> SENDBUF(*),RECVBUF(*)
    INTEGER COUNT,DATATYPE,OP,ROOT,COMM,IERROR
MPI_ALLREDUCE(SENDBUF,RECVBUF,COUNT,DATATYPE,OP,COMM,IERROR)
    <type> SENDBUF(*),RECVBUF(*)
    INTEGER COUNT,DATATYPE,OP,COMM,IERROR

MPI_COMM_SIZE(COMM,SIZE,IERROR)
    INTEGER COMM,SIZE,IERROR
MPI_COMM_RANK(COMM,RANK,IERROR)
    INTEGER COMM,RANK,IERROR
MPI_COMM_DUP(COMM,NEWCOMM,IERROR)
```

```
    INTEGER COMM,NEWCOMM,IERROR
MPI_COMM_SPLIT(COMM,COLOR,KEY,NEWCOMM,IERROR)
    INTEGER COMM,COLOR,KEY,NEWCOMM,IERROR
MPI_COMM_FREE(COMM,IERROR)
    INTEGER COMM,IERROR
MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, REER_COMM,
    REMOTE_LEADER,TAG,NEWINTERCOMM,IERROR)
    INTEGER LOCAL_COMM,LOCAL_LEADER,PEER_COMM,REMOTE_LEADER,TAG,
    NEWINTERCOMM,IERROR
```

第十五章　并行程序设计环境与工具

广义上讲,并行程序设计环境系由硬件平台、支撑语言、系统软件、应用软件包和软件工具等组成;狭义上讲,并行程序设计环境就是由一组并行程序设计工具所组成。所谓软件工具系指任何能帮助软件开发过程的辅助工具;而程序设计工具系指任何有助于程序员编程的辅助工具。显然软件工具含意较广,进行程序设计总要利用某些软件工具。所以本章先从一般的软件工具环境讲起,然后重点介绍并行程序设计语言的并行编译器以及并行程序的调试、性能分析和可视化设计环境与工具。

*15.1 软件工具与环境

15.1.1 编码工具

编辑器(Editor) 它是一种具有编辑功能的程序设计工具,其功能从简单的文本编辑器到语法制导(Syntax_Directed)或语言敏感(Language_Sensitive)的编辑器。目前最普通的程序编辑器是知识语言(Language_Knowledgeable)编辑器,它借助语言知识来增强编辑能力,例如可自动缩进(Indentation)、括号检查,甚至简单交叉引用等。Emacs就是这样的编辑器[79]。

编译器(Compiler) 它将源代码转换成机器可执行的代码(或实际机器代码或能被解释的中间代码)形式。各种编译器的差别在于执行优化的程度不同以及它们是不是渐增的(Incremental,即只当变化时才编译)或成批的。并行编译的技术难点是对一个源程序如何识别出可并行执行的部分而将其并行化,同时保证整个程序的正确运行。本章第15.2节会专门对此进行讨论。

链接器和加载器(Linker and Loader) 它们将编译好的文件与库连接在一起以产生一个可执行的文件。因为库的内容一直在增加,因此连接时间占了编辑-编译-调试总时间的相当部分。使用渐增链接器可缓解此问题,渐增链接器只修改那些已改变的可执行部分。

预处理程序(Preprocessor) 它提供超过基本源语言范围的某些能力,这些能力可内置在源语言中(如C预处理器),也可以由它们自己的语言提供(如Lex和Yacc)。

交叉引用程序(Cross Referencer) 它提供将名字和其定义相关联的方法。交叉引用信息可有效地由编译器产生,它可以用在编辑器中,也可用在文本或者图形显示的独立系统中。普通图形交叉引用工具,可提供调用图和面向对象程序的类层次图的显示。

源级调试器(Source_Level Debugger) 它允许用户用设置断点等方法控制程序的执行。更先进的查错工具能够处理优化的代码和库以及同一系统中的多种语言。现今这些工具已配用程序的可视显示手段,允许程序员观察应用的数据结构。

调试辅助程序(Debugging Aids) 它们提供超过程序设计语言范围的在编译或运行时的检查工具,还包括那些甚至编译器发现是合法结构的潜在问题的系统和库。

15.1.2 软件工程工具

这些工具牵涉到整个系统的维护而不是代码本身,现今这类工具包括:

系统构造程序(System Builder) 该工具允许用户定义系统应如何建造系统模型,模型包括相关性、编译选项和应该执行什么命令等信息;然后工具应能够构筑原型系统或者基于一组源文件变化渐增的更新系统。

版本管理程序(Version Manager) 该工具允许一个源文件的多种版本同时共存。这样就允许多个程序员共同开发系统,允许在新版本开发的同时维护已发布的系统的老版本,或者允许单一系统不同定制版本的共存。

设计编辑器(Design Editor) 该工具使用户采用各种图形设计符号设计一个系统。CASE(Computer-Aided Software Engineering)就是这样的工具,它使用 Petri 网、SADT[150]、状态图和面向对象的设计 OMT[151]等表示来开发软件。很多这样的工具至少能基于设计产生代码框架。如果提供足够的设计信息,有些工具能够模拟系统的若干方面,同时允许开发者在较高级测试设计。

代码产生器(Code Generator) 该工具也称为第四代语言,它实际上是让程序员交互地指定系统的大部分而不必进行编码的专用高级语言。它们普遍地用于定义用户界面和程序与数据库系统的相互作用。

测试辅助程序(Testing Aids) 该工具试图将测试软件系统的过程自动化。其范围从测试事例产生器(用于分析源代码或说明以产生一组测试实例)到回归测试系统(程序员产生测试事例,系统施行内务操作,包括运行每个测试实例、确定它是否成功或失败以及报告结果)。

15.1.3 集成工具

软件工具可以使用不同的集成技术进行组合。早期,或采用不同工具的松散联合,或将所有相关的工具组合成单一系统。单一系统的优点是,紧密地将一些工具耦合在一起使得程序员对环境有所了解;其缺点是,此单一系统是个封闭系统,很难加入新的工具,或很难使用多种语言或者现存代码,而且系统相当庞大。紧密的联合可以解决不少问题,它是个开放系统,易于开发和利用新工具,且系统可以使用多种语言和库构筑之。然而,它把正确使用工具的义务转给程序员且不提供公共框架。目前已提出三种将工具集成为环境的方法:

数据集成(Data Integration) 该方法使诸工具共享信息,为此要开发一个能保存各个工具必须共享的信息数据库。例如,编译器产生的中间表示可存入数据

库中并且能被别的工具直接使用,以避免重新进行源程序语法分析。此法的缺点是,所要求的数据库很大,而且为了确保环境正常工作,数据库系统必须是先进的;同样这样的环境趋于封闭而不是开放。

公共前端 (Common Front End) 在软件开发中最流行的工具是用于生成程序、文本、系统模型等的文本编辑器。它相对容易地可嵌入编辑器的命令中,这些命令可调用其它工具并使用那些工具的输出。这种集成方法为各种工具提供一个公共的界面,而且如果它们有一个文本界面就可使新的工具很简单地被集成。因为这些工具实际不共享信息,所以程序员仍旧知道不同的工具及其使用方法。

控制集成 (Control Integration) 它涉及到工具之间的消息传递。在此,每当诸工具需共享信息时和调用命令时就彼此发送消息。通常不使用点到点消息传送,而是使用中央消息服务器作为中介物,每个工具都要告诉消息服务器它们对什么消息感兴趣。此法集成度较高,允许多种工具尤如是一个那样进行操作。

实际的环境可以是上述不同集成方法的组合。例如,PCTE 标准 [37] 就使用了控制集成,以及在数据集成之上的公共前端实用程序。

15.1.4 将来的工具与环境

因为软件的开发是个困难和费时的过程,所以人们不断地发展和推广现有的工具和环境。可以预计将来的工具一定比现在的好:较快的编译器、渐增的加载器、较好和更多的查错辅助程序和完善的编辑器等等。相应地,这些先进工具的集成会形成一个开放的环境。现在的研究旨在推广软件工具以支持一些新的领域:

过程工具 (Process Tools) 这些工具帮助管理软件开发过程,它们允许使用规则和步骤定义过程,然后提供过程内务操作和管理的帮助。

群件工具 (Groupware Tools) 这些工具允许多个程序员以受控方式联合工作。版本管理为多个程序员提供一个基础,而新的工具将会促进分布在不同场点的开发者之间的合作和交互能力的增强。

可视化工具 (Visualization Tools) 现行的可视化主要集中在类层次显示、调用图或模块的显示以及受限的数据结构可视化上,提供大量程序数据的高质量显示有待增强。这些将会变成软件环境的标准部分。

程序分析工具 (Program Analysis Tools) 这些工具承担系统的详细语义分析并向用户提供反馈。反馈可能与潜在的问题、代码和说明之间的失配、帮助归并不同版本或重定工程代码 (Reengineering Code) 有关。

15.2 并行编译器

一旦一个程序以某种高级语言书写完成后，在正式运行前，必须将此程序转换成实际机器能够理解的机器语言（指令集）。此过程就是编译（Compile），而编译器实际上就是实现此转换的一种语言处理程序。编译过程可分为：①词法分析；②语法分析；③中间代码产生；④代码优化；⑤代码生成等几个阶段。上述几个阶段或多或少都是顺序执行的。而并行化编译面临的任务是：给定一个在单处理机上运行较长的串行程序和一台具有多个处理器可同时工作的并行计算机，目的是将串行程序分解成若干个能并行执行或至少能重叠执行的代码段，使其在并行机上能较快地运行。所以并行编译器主要工作就是寻找代码的并行性，然后将其调度在并行机上高速正确地执行。

如图 15.1 所示，一个并行编译器大致可由三部分组成：流分析，程序优化和代码生成。其中，流分析是确定源代码中数据和控制的相关性；优化常常是将代码变换成与之等效但具有“更好”的形式，以利于尽量挖掘硬件潜力，最终达到全局优化的目的；代码生成通常涉及到从一种描述转换成另一种中间形式的描述，不同类型的计算机其并行代码的生成也各不相同。在正式讨论并行编译器之前，先对编译及其并行化作一概括介绍，以期读者对此有个感性的认识。

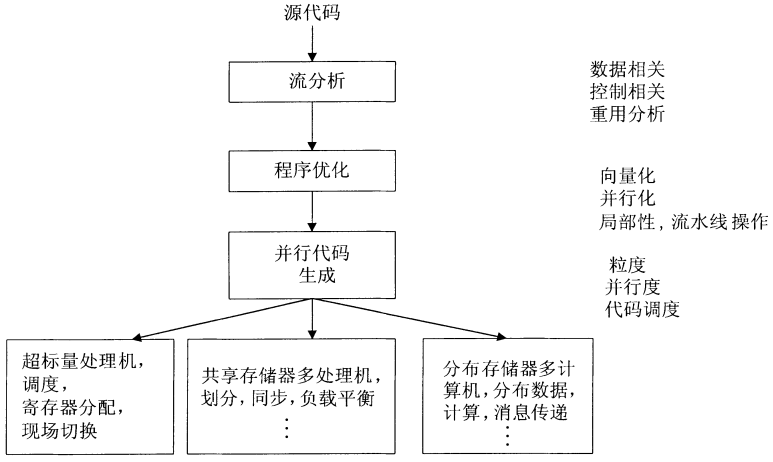


图 15.1 并行编译过程

15.2.1　编译及其并行化

图 15.2 示出了编译器将一个高级语言的代码段翻译成汇编语言形式的机器目标代码的过程。最右边还给出了经过简单优化后使用较少指令的目标代码。

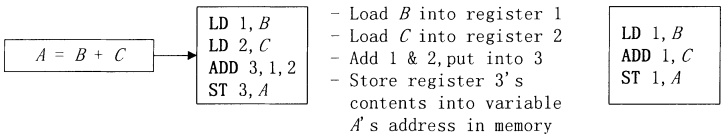


图 15.2　例示编译过程

事实上，最左边的源代码经过词法分析器首先被分解成一些原子目标（即 tokens），再把它们分类为操作符、常数、分隔符、标志符等；然后经过语法分析器，分析程序的文法结构、检查错误，最后转换成类似于图 15.3 的语法分析树；产生中间代码是为了便于移植和优化，中间代码和汇编语言的主要区别是，前者不必为每种操作之输入和输出指定实际的寄存器和存储器位置；优化的目的是使程序运行得较快和使用较少的存储器，其主要方法重排编译后的代码（即所谓代码移动），它是建立在流分析的基础上的，是程序向量化和并行化的关键。

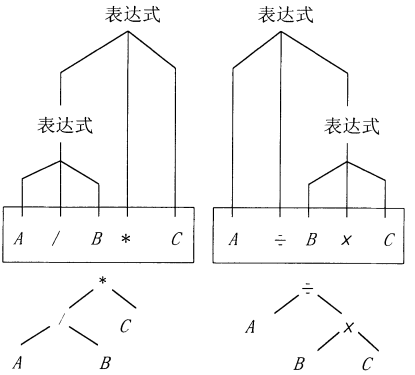


图 15.3　表达式 A/B*C 与 A÷(B×C) 的分析树

一般而言，有两种并行化代码段的方法：SIMDizing，即向量化（Vectorizing）和 MIMDizing，即并行化（Parallelizing）。如图 15.4 所示，代码段的程序流图被分成几个不同的计算调度单元。此时，DO 循环可被分布成三个不同的相互独立的循

环,它们分别标志为“向量”、“向量”和“递归”.其中前两个循环执行简单的、相同的和独立的算术运算,因此每个循环都可用一条向量指令代替之,从而达到向量化;后一个循环涉及到递归,是彼此相关的,所以无法向量化,但可将代码分成可供MIMD多处理机执行的几个任务,每个处理器负责循环的若干次迭代,各处理器之间再施行必要的同步就可达到并行化。

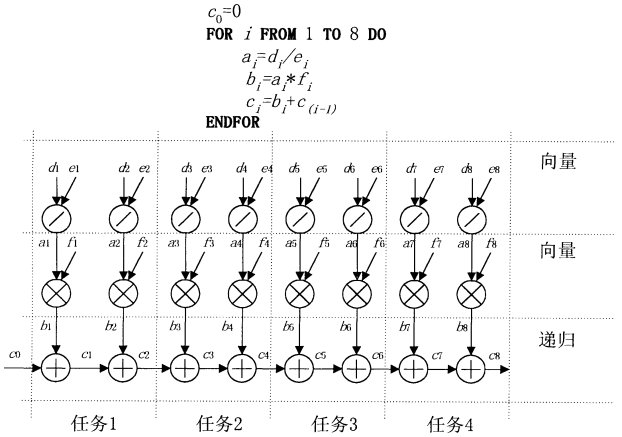


图15.4 两种并行化的方法

下面结合一段具体的程序代码,分别给出相应的向量化和并行化的结果,以期读者对两种并行化方法有个感性认识。

顺序代码	向量化代码	并行化代码

(1) A=B+C
(2) DO 5 I=1,N
(3) D(I)=A*E(I)
(4) S=E(I)*5
(5) T=T+S
(6) A=D(N)-7
	(1) A=B+C
	(3) D(1:N)=A*E(1:N)
	(3.1) Allocate S'(1:N)
	(4) S'(1:N)=E(1:N)*5
	(5.1) S=S'(N)
	(5.2) Free S'(N)
	(5.3) I=N+1
	(6) A=D(N)-7
		A=B+C
		With N Processors
		Local S'
		D(#)=A*E(#)
		S'=E(#)*5
		T=T+S' ◆ Synch
		IF(#·EQ·N) THEN
		S=S'
		A=D(N)-7
		I=N+1
		ENDIF

15.2.2 相关分析

要并行执行几个程序段,必须使每段与其它各段无关。研究计算程序中所有语句间的相关性称为相关分析(Dependency Analysis)。准确有效的相关测试对发挥并行化编译器的功效十分重要。先定义如下四种形式的相关。假定语句 S_j 继语句 S_i 之后执行,其定义及符号约定如下:

①流相关 如果从 S_i 到 S_j 存在执行通路,而且如果 S_i 至少有一个输出可供 S_j 用作输入,则语句 S_j 与语句 S_i 流相关(Flow_Dependent),记之为 $S_i\delta S_j$。

②反相关 如果 S_j 紧接 S_i,而且如果 S_j 的输出与 S_i 的输入重叠,则语句 S_j 与语句 S_i 反相关(Antidependent),记之为 $S_i\bar{\delta}S_j$。

③输出相关 如果两语句能产生(写)同一输出变量,则两者是输出相关(Output_Dependent),记之为 $S_i\delta^{o}S_j$。

④控制相关 如果语句 S_j 的执行依赖于语句 S_i(S_i 必须在 S_j 之前执行),则语句 S_j 与语句 S_i 控制相关(Control_Dependent),记之为 $S_i\delta^{c}S_j$。

图 15.5 分析了上一节中所列举的顺序代码的各种相关(不包括控制相关)。

向量数组的相关分析比标量数据的相关分析要复杂得多,本节只概略讨论数据数组相关分析的基本概念,有兴趣的读者可参阅[100]。

相关距离向量和相关方向向量 令 $\alpha=(\alpha_1,\alpha_2,\cdots,\alpha_n)$ 和 $\beta=(\beta_1,\beta_2,\cdots,\beta_n)$ 是在 n 层循环上界和下界范围内的 n 个整数指标向量,假定 α 和 β 存在数据相关性,则相关距离向量(Dependent Distance Vector) $D=(D_1,D_2,\cdots,D_n)$ 定义为 $\beta-\alpha$,相关方向向量(Dependent Direction Vector) $d=(d_1,d_2,\cdots,d_n)$ 定义为:

$$d_i=\begin{cases}<\text{若 } \alpha_i<\beta_i\\ =\text{若 } \alpha_i<\beta_i\\ >\text{若 } \alpha_i<\beta_i\end{cases} \tag{15.1}$$

例如,假定有如下循环嵌套:

```
DO i=L1,U1
  DO j=L2,U2
    DO k=L3,U3
      A(i+1,j,k-1)=A(i,j,k)+C
    ENDDO
  ENDDO
ENDDO
```

则数组 A 的三维迭代之间的相关距离向量和相关方向向量分别为 (1,0,-1) 和 (<,=,>)。

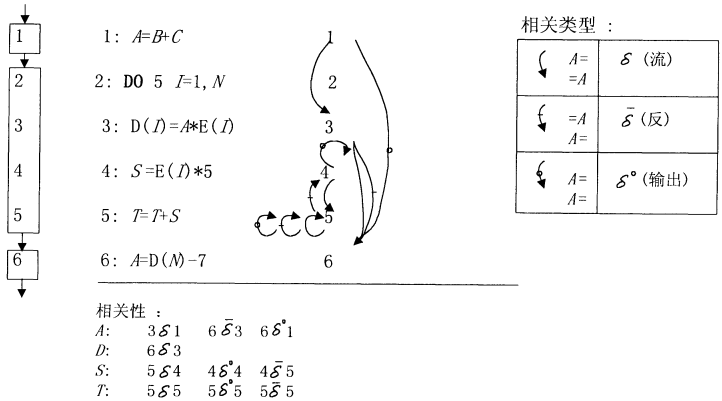


图 15.5 相关分析示例

相关方向向量对计算循环体间相关性十分有用。这里,相关性是通过相关方向向量中方向不是"="号的外层循环传递的。所以上述相关方向向量 (<,=,>) 所表示的相关性就是在循环 i 传递的。相关距离向量指明在对同一存储单元的两次访问之间循环迭代的实际距离。它们对开发并行性或优化存储器层次结构将起到指引作用。

相关方程 (Dependent Equation) 设 α 和 β 是在 n 层循环上界和下界范围内的 n 个整数指标向量。当且仅当存在 α 和 β,并且 α 按字典序小于或等于 β 以及满足下述相关方程时,则从 S_i 到 S_j 存在相关性:

$$f_i(\alpha) = g_i(\beta) \qquad 1 \leq i \leq m, m \text{为数组维数} \qquad (15.2)$$

否则两次引用都是不相关的。

(15.2) 式中的相关方程是循环指标变量的线性表达式。所以测试相关性相当于求解线性 Diophantine 方程,它是 NP 完全问题。

数据相关性测试 (Dependency Testing) 进行相关性测试目的有二:一是试图证明同一数组变量的下标引用对 (即引用两个数组某维中的一对下标) 之间的相关性是不存在的;二是若相关性存在,则应以某种方式 (例如使用相关方向向量的最小全集) 将它们表示出来。相关测试采用比较保守的策略,若不能证明任何相关性是不存在的话,则就认为它们是存在的。

进行相关性测试所追求的目标,就是构造相关距离向量或相关方向向量的全集。此全集可以表达对同一数组变量任意下标引用对之间可能存在的相关性。这

实际上是求方程组(15.2)的通解,已如上述这是个NP-完全难解问题。精确测试是当且仅当相关性存在时,检查这些相关性,它所花费的开销太大,因而精确解无法实现,只好采用近似测试法。最常用的两个近似测试法是最大公约数测试,即GCD测试[183]和Banerjee不等式测试[24]。本书限于篇幅就不再讨论了。

15.2.3 代码优化

代码优化(Code Optimization)的目的是使代码执行速度达到最高,这就要求代码长度最短、存储器访问次数最少以及程序的并行性得到较好的开发。优化技术包括用流水线硬件完成向量化和同时用多台处理器实现并行化。向量化是把标量循环操作转换成等效的向量指令执行的过程;并行化的目的是把顺序代码转换成并行形式,使多台处理机能同时并行执行。向量硬件用来加速向量操作,多处理机或多计算机用来执行并行代码。本节简要讨论代码向量化方法和代码并行化方法。

代码向量化(Code Vectorization)方法 向量是一维的,所以直观上讲,向量程序设计就是把标量程序中的由一种可向量化循环完成的操作改成向量操作。常用的代码向量化的基本方法包括:

(1) 直接向量化法:它是把串行程序中的循环直接用数组运算语句来描述。

例15.1
```
DO  I=8,120,2
      A(I)=B(I+3)+C(I+1)
ENDDO
```
此标量循环可直接转换成一条由下列数组赋值语句定义的向量化指令:
```
A(8:120:2)=B(11:123:2)+C(9:121:2)
```

(2) 含条件语句的循环向量化:循环中的条件语句可以用WHERE语句向量化。

例15.2
```
DO  I=1,N
      IF (L(I).NE.0)  A(I)=A(I)-1
ENDDO
```
它可向量化为:
```
WHERE (L(I).NE.0)  A(1:N)=A(1:N)-1
```

(3) 语序重排向量化法:有时按照循环中原有的语序会因数据相关而妨碍向量化,此时交换一下语句顺序即可向量化。

例15.3
```
DO  I=1,N
      A(I)=B(I-1)
      B(I)=2*B(I)
ENDDO
```

上例中对B的引用导致一个妨碍向量化的流相关,但把两语句执行顺序交换一下即可得到如下形式的向量化:

```
B(1:N) =2*B(1:N)
A(1:N) =B(0:N-1)
```

(4) 引入临时数组向量化法:通过引入中间数组,有时可消除妨碍向量化的数据相关,从而达到向量化之目的。

例15.4
```
DO  I=1,N
    A(I) =B(I) +C(I)
    B(I) =2*A(I+1)
ENDDO
```

上例中关于数组A存在一个反相关,兹引临时数组TEMP先把A(2:N+1)暂存起来,就可得到如下形式的向量化:

```
TEMP(1:N) =A(2:N+1)
A(1:N) =B(1:N) +C(1:N)
B(1:N) =2*TEMP(1:N)
```

(5) 交换循环向量化法:向量化通常在内循环而不是在外循环进行。当内循环不能向量化而外循环可以向量时,则交换一下内外循环即可实现向量化。

例15.5
```
DO  I=2,N
    DO  J=2,N
        A(I,J) = (A(I,J-1) +A(I,J+1))/2
    ENDDO
ENDDO
```

上例中数组A在J维存在着数据相关,故在J维不能向量化,兹将两层循环交换如下:

```
DO  J=2,N
    DO  I=2,N
        A(I,J) = (A(I,J-1) +A(I,J+1))/2
    ENDDO
ENDDO
```

现在I维(内层循环)无相关性了,所以向量化的代码为:

```
DO  J=2,N
    A(2:N,J) = (A(2:N,J-1) +A(2:N,J+1))/2
ENDDO
```

(6) 循环分离向量化法:有时一个循环中只有部分语句可向量化,此时可将整

个循环分离成两个：一个含可向量化的语句；另一个含不可向量化的语句，然后将能向量化的循环向量化之。

例 15.6
```
DO  I=1,100
      A(I) =B(I) +C(I)
      D(I) =D(I-1) *E(I)
    ENDDO
```

上例中第一个语句可向量化，第二个则不能，可把它分离成如下两个循环：

```
DO  I=1,100
      A(I) =B(I) +C(I)
ENDDO
DO  I=1,100
      D(I) =D(I-1) *E(I)
ENDDO
```

这样第一个循环可向量化为：

```
DO  I=1,100
      A(1:100) =B(1:100) +C(1:100)
ENDDO
```

(7) 迭代区间分段向量化法：例如当循环迭代区间从正到负变化时，整个迭代区间可能要分成几段才能向量化。

例 15.7
```
DIMENSION  A(-200,200)
    DO  I=-100,100
          A(I) =0
          A(2*I) =1
    ENDDO
```

此程序运行结果是：

A(-200)，A(-198)，…，A(0) 等于 1；
A(-99)，A(-97)，…，A(-1) 等于 0；
A(1)，A(2)，…，A(100) 等于 0；
A(102)，A(104)，…，A(200) 等于 1。

显然本例不能直接向量化为 A(-100:100) =0 和 A(-200:200:2) =1，也不能向量化为 A(-200:200:2) =1 和 A(-100:100) =0，但若从 0 为分界点，I≤0 时可按该循环的语句顺序进行向量化；当 I>0 时则必须按语句反序进行向量化，此时可得到如下的向量代码：

```
A(-100:0) =0
```

A(-200:0:2) =1
A(2:200:2) =1
A(1:100) =0

向量化的方法还有很多,在此就不再进一步讨论了。

代码并行化(Code Parallelization)方法 并行代码的优化是将一个程序展开成许多线程以供多台处理机并行执行,其目的是要减少总的执行时间。在此线程就是在一台处理机上执行的指令序列。能否将一个程序分成多个线程并行执行,依赖于程序的数据和控制相关性。理想的情况是任何两个线程之间无相关性,所以可以并行执行;当线程之间有强相关性时,若加上适当的同步操作,则含有任何相关性的两个线程也可并行执行,极端的情况是同步多得使两个线程事实上还是串行执行的。

循环一般占程序的大部分运行时间,而循环的并行化分析和改写又相对较容易,所以十多年来有关代码并行化的研究主要集中在循环的并行化方面。因为组织循环并行执行的开销一般比组织循环向量化执行的开销大得多,所以对嵌套循环作并行化时总是选择外层循环进行并行化。这样使存储器能对数组中的连续元素进行访问,并将带有长向量的循环放到最内层进行向量化,从而缩短了总的执行时间。

返回去再看一下例15.5中的两层循环嵌套。因为I循环中的所有相关关系都有相关方向"=",所以这个外层循环是可以并行化的,其并行化后的代码如下:

```
DOALL  I=2,N
         DO   J=2,N
            A(I,J) = (A(I,J-1) +A(I,J+1))/2
         ENDDO
ENDALL
```

此时外层循环中的N-1次迭代都可以调度给不同的处理机执行。每个新创建的线程都由一个带I常数下标的完整的J循环组成。

下面给出一个例子综合说明一个程序循环的串行执行、并行化、向量化执行的各种组合情况。

例15.8 在Alliant FX/80处理机上一个FX/Fortran循环的五种执行方式[10]示例在图15.6中。

FX/Fortran可以用标量、向量、并发标量、并发向量及并发外层/向量内层(COVI)等五种方式来生成代码以执行简单的DO循环。例中所涉及的计算可以对一维数据数组A(1:2048)进行;也可以对二维数据数组B(1:256:1:8)进行,其中,A(K) =B(I,J),而K=8(I-1) +J。

图 15.6 (a) 示出了用一台处理机顺序执行标量的情况。若利用数组 A,则所涉及的计算可用一个纯标量循环表示如下：

```
DO  K=1,2048
    A(K) = A(K) + S
ENDDO
```

其中 S 是标量常数。

图 15.6 (d) 示出了用一台配有向量硬件的处理机执行 8 条向量指令的情况。每条向量指令处理下列 256 次迭代：

A(1:2048:256) = A(1:2048:256) + S。

图 15.6 (c) 示出了用 8 台处理机并行执行如下的标量计算 (标量并发方式)：

```
DOALL  I=1,8
    DO  J=1,256
        B(I,J) = B(I,J) + S
    ENDDO
ENDALL
```

$A(1) = A(1)+S,\ A(2) = A(2)+S, \ldots, A(2048) = A(2048)+S$

(a) 一台处理机执行标量

$A(1:256)=A(1:256)+S,\ A(257:512)=A(257:512)+S, \ldots$
$A(1793,2048)=A(1793:2048)+S$

(b) 一台处理机顺序执行向量

P_1: $B(1,1)=B(1,1)+S,\ B(1,2)=B(1,2)+S, \ldots, B(1,256)=B(1,256)+S$
P_2: $B(2,1)=B(2,1)+S,\ B(2,2)=B(2,2)+S, \ldots, B(2,256)=B(2,256)+S$
$\vdots$
P_8: $B(8,1)=B(8,1)+S,\ B(8,2)=B(8,2)+S, \ldots, B(8,256)=B(8,256)+S$

(c) 八台处理机执行并发标量

P_1: $A(1:2041:8)=A(1:2041:8)+S$
P_2: $A(2:2042:8)=A(2:2042:8)+S$
$\vdots$
P_8: $A(8:2048:8)=A(8:2048:8)+S$

(d) 八台处理机执行并发向量

P_1: $B(1,1:256)=B(1,1:256)+S$
P_2: $B(2,1:256)=B(2,1:256)+S$
$\vdots$
P_8: $B(8,1:256)=B(8,1:256)+S$

(e) 八台处理机执行COVI

图 15.6 Alliant 多处理机上 FX/Fortran 循环五种执行方式

图 15.6(d)示出了用 8 台全部配有向量硬件的处理机上以向量并发方式执行如下代码的情况：

```
DOALL  J=1,8
   A(K:2040+K:8) =A(K:2040+K:8) +S
ENDALL
```

图 15.6(e)示出了 8 台处理机以 COVI 方式执行，内层向量化执行，外层并行化执行，以 Fortran 90 表示符则有：

```
B(1:8,1:256) =B(1:8,1:256) +S
```

在代码并行化时，下列的情况往往会阻止或妨碍并行化：①多个入口或出口；②函数调用或子例程调用；③输入/输出语句；④并行执行的非确定性；⑤循环体间的相关性。

总之，对向量化只是向后相关性有影响，而对并行化则向前和向后相关性都有影响；同步代码的开销可能会超过并行化所带来的好处；大多数代码的并行化都集中在循环级；并行化循环应尽量在最外层进行；在考虑并行化的粒度时，必须在计算和通信之间折衷考虑等。

15.2.4 代码生成

并行代码生成(Code Generation)涉及到将优化后的中间形式的代码转换成可执行的具体的机器目标代码。因为目标机包含有不同的并行结构，所以并行机的代码生成仍还限制在每台处理机的范围内进行。不同类型的计算机的并行代码的生成也很不一样。例如，超标量处理机可能用软件调度也可能用硬件调度来实现。在 RISC 或超标量处理机上如何优化寄存器分配，在多处理机上如何降低同步开销，在多计算机上分布的代码/数据如何实现消息传递等，这些问题都给并行代码的生成增添了不少困难。当代码自动生成不易实现时，还可以使用编译命令来帮助并行代码的生成。

代码生成所涉及的问题很多，包括执行次序、指令选择、寄存器分配、负载平衡、并行粒度、代码调度以及后优化(Postoptimization)等。先进的代码生成技术应以开发并行性为目标，要为并行机开发真正的“智能”编译器，目前还相差甚远。

15.3 并行程序调试

并行程序的设计不仅编程难，而且调试和分析更难。调试的目的是为了获得

一个正确的并行程序;而性能分析是为了获得一个高效的程序。本节简要介绍并行程序调试方法、步骤、技术和性能分析。

15.3.1　并行程序调试的方法与步骤

并行程序调试的困难　目前串行程序调试已经有了比较成熟的技术,相比而言,并行程序调试(Parallel Program Debugging)技术尚不成熟。并行程序在运行中引入的并行化,给并行程序调试带来了极大的困难,它主要体现在不确定性和探针效应。

(1) 不确定性(Indeterminacy):它指的是并行程序在运行当中语句的执行次序是不确定的,这种不确定性意味着相同的程序输入在不同的执行中并不一定产生相同的输出,从而使得串行调试中经常采用的循环调试(Cycling Debugging)失效。在串行程序调试当中,当程序员发现错误时,可以再次执行程序,跟踪其过程并发现错误所在,这就是所谓的循环调试。串行程序通常是单任务的,程序运行具有确定性,因而保证了循环调试的有效性。

引起并行程序运行不确定性的原因有许多,主要包括:①共享变量竞争:它主要针对共享存储的系统而言,读写操作是对共享变量进行的。共享变量的读写需要恰当的同步操作保证。②消息传递竞争:它主要针对依靠消息收发进行通信的分布存储系统而言。由于消息传递的异步性,在不同的程序执行中消息的收发顺序可能不同。③动态进程调度:动态进程调度的目的是为了在并行机上取得负载平衡,在不同的程序执行中调度的次序可能不同。④不确定的系统调用:程序与其运行环境的交互在不同的程序执行中可能不同。

(2) 探针效应(Probe Effect):它是由并行调试工具的引入引起的问题。调试工具的引入,可能掩盖被调试程序中的时序错误。例如,当某些输出语句作为测试指令被加入到程序中后,可能会改变原来的语句执行次序。不难发现,因为单任务的执行方式,串行程序调试基本不受探针效应困扰;而对于并行程序而言,不同任务间语句的执行次序直接关系到程序的结果输出,因此添加额外的测试语句应当格外谨慎。

并行程序调试的方法　尽管并行调试存在很大困难,但目前已经积累了不少并行调试的方法。我们介绍重放和断点调试。

(1) 重放(Replay):它的提出是为了解决并行程序的不确定性,从而使循环调试能有效地为并行调试服务。典型的重放技术可以在支持消息传递的分布存储系统中找到。在程序运行中可以对相关的消息传递进行记录,以便在重放时利用这些信息来控制同步通信,保证循环调试的有效性。值得注意的是,在分布存储系统

中,程序中的消息传递语句是与计算分开的,因而记录的信息量相对不大,规模容易控制。而在共享存储系统中,消息的收发代之以对共享变量的读写,通信与计算是密不可分的,因而记录的额外开销将难以控制。要解决这个困难,应当结合共享变量系统的特点,开发更高一级的同步控制,如锁、路障等,然后在此基础上实现记录与重放。

重放实现中应当注意探针效应。重放实现的额外开销主要集中在同步通信前后,因此不恰当的记录开销可能会影响语句间的时序关系,产生探针效应。在调试器设计时,应当考虑将记录与重放的实现在通信语句库中实现,而不是在调试时向源程序中添加语句,这样可以比较有效地消除大多数探针效应。

(2) 断点调试(Breakpoint Debugging):它是串行程序调试中的基本方式。在调试开始时,通常在语句级设置断点。在并行调试中,多任务取代了单任务,相应的多个串行断点取代了单个断点。同串行调试中的断点相比,并行调试由于其自身特点而派生出更多类型的断点:①基于控制流的断点,同串行调试的断点类似。②基于事件的断点,当发生异常或用户定义的事件时断点生效。③基于谓词的断点,当某个条件表达式成立时断点生效。此外,由于并行程序的多任务性,断点还有所谓的全局断点:当一定的条件成立时中断所有的并行任务的执行。全局断点比较复杂,往往需要硬件支持。

并行程序调试步骤　从广义角度来讲,并行程序调试的步骤为:①算法正确性检查:保证并行处理本身在算法层次上的正确无误,如果所采用的并行算法是由串行算法改写而成的,则应当首先保证该串行算法的正确性。这一步骤是正确性调试的基础,通常只能由人工来进行。②边界运行调试:通常取处理器数目为1的最简单情形进行测试,以排除一些简单的错误,确保剩下的错误多数只与多处理器情形下的协调通信相关。③联合调试:需要小规模的并行环境,通常可先测试两个处理器下的程序运行,排除错误。④可扩展性调试:即在算法和程序允许的范围内逐步增大处理器数目进行调试。

由于步骤①通常由人工完成,步骤②通常可借助于串行调试工具进行,所以一般意义上说的并行调试往往指的是步骤③和④,并行调试的工具主要也是针对这两个步骤。

错误原因分析　当调试出错时,可以按以下步骤来检查错误所在:①首先检查程序数据特性(共享或局部)定义是否正确;②在分布存储的系统上数据是分布在局存中的,当出错时应检查数据的分布是否正确;③检查数据相关或控制相关所造成的程序执行流向的可能错误;④检查同步机制是否错用或漏用同步信号,使得某些相关不能满足而引起结果的不确定性,经验证明同步通信错误是较多的和难查的;⑤当出现固定错误时应多检查数据定义特性、依赖关系等方面;出现不固定的

错误时则应多检查同步通信等操作。

15.3.2 并行程序的调试技术

本节主要简单介绍一些并行程序的调试技术,包括全局断点、渐增检查点、事件分析和静态分析等。

全局断点 在并行程序的运行中,常常涉及到需要多个进程同时中断运行的情况。比如在并行计算1 000个数的和时,程序员发现结果错误,可能需要验证一个计算前100个数之和是否正确。这里需要添加的断点类型是条件断点或谓词断点,当满足一定条件时断点生效,并且此时需要中断的进程不止一个。这种情况下,在某个进程内触发断点生效后,应当将此信息播送到相关的要中断运行的进程。这种类型的断点,称为全局断点。然而问题也由此产生:播送本身有一定延迟,因此当相关进程都停下来后,程序员所观察到的状态并非是断点生效时瞬间的状态;解决这个问题是实现全局断点技术的关键。

渐增检查点 并行程序处理的一般是比较复杂的计算,处理的时间周期往往较长。在调试此类程序时,应当充分验证计算规模充分大时程序的可靠性。然而由于时间开销较大,调试的周期也会随之增长。并行计算的应用中,程序运行数日数月的不在少数,这样的规模也给调试带来了很大的困难。如果一旦发现错误后从头开始重放检查,势必消耗大量人力物力。为此,渐增检查点(Incremental Checkpoint)的引入迫在眉睫。通俗地说,渐增检查点提供的是类似存盘的功能。在程序运行中,一个周期之后进行一次程序运行状态的全面记录。周期的划分可以是基于时间片的,也可以是基于数据片的,比如每计算完若干数据进行一次记录。这样程序员可以查看各次记录,缩小错误范围。渐增检查点同日志不同。日志(Log)记录的是所有相关的运行信息,而渐增检查点记录的是若干程序运行横截面的信息。通过日志当然也可以发现运行的错误所在,但由于其信息量太大,实际上对于运行周期长的程序不可能去详细检查日志;渐增检查点记录的信息相对较少,程序员可以迅速找到错误的大致位置。程序员可以取一个不出错的检查点开始运行,配合重放功能,从而可以比较迅速地找到错误出在何处。

事件分析(Event Analysis) 它的方式是记录程序运行中发生的有关信息,以提供给事后阅览和重放。阅览是针对程序员而言,帮助发现程序中的错误隐藏之处。此外,事件分析还往往是重放技术赖以生存的基础。事件分析的方式只提供了一些运行信息而不是错误信息,所以通常并不单独使用而往往与其他技术结合使用。

静态分析(Static Analysis) 它指的是对程序进行文本分析,识别程序中潜在

的同步问题,以提前指出错误,供程序员参考分析。静态分析方法可以作为并行程序的预调试阶段,提供辅助调试支持。应当指出,目前的静态分析工具在实际使用时时空开销仍然较大,同时智能化程度也较低。

15.3.3 并行程序的性能调试

并行程序的性能调试虽然没有固定模式,但还是能找出一些常用的方式。性能调试一般可以分为测量、分析和优化三个步骤。

测量(Measurement) 它是运行中收集信息的过程。记录的信息,包括上面提到的有关软硬件的状态,如高速缓存命中率、子任务的启停等。测量可以得到完整的原始记录,但缺点是数据信息量大;可以考虑在原始记录的基础上进行粗加工,提取有用信息以减少信息量,也便于以后阶段的使用。

分析(Analysis) 它是找出性能问题的过程,是性能调试中最为关键的步骤。性能分析大体上可以分为静态和动态两种方式。

(1) 静态性能分析(Static Performance Analysis):它又称为性能预测。静态分析的方法是在源程序一级用模拟分析的方法实现的,而静态的确定程序运行轨迹本身非常困难,因而仅仅是一种试探性的方法。采用静态性能分析,可以用比较小的时空代价来对一些程序结构,如循环和函数调用等进行初步的分析,以供程序员参考。

(2) 动态性能分析(Dynamic Performance Analysis):它是收集程序在运行中的相关数据,对性能进行分析的方法。同静态性能相比,动态性能分析更能体现并行程序的运行本质,因而对程序员也更有参考价值。

从软硬件角度看,动态性能分析包括对硬件性能的监测和对软件性能的监测。对硬件性能的监测包括各部件的忙闲情况、访存冲突率和高速缓存(Cache)命中率等。但是硬件性能检测本身较为复杂,往往需要硬件支持。对软件性能的监测由操作系统和并行运行支持系统完成,它们在完成正常的功能外,还提供事件信息记录的功能。程序员可以通过这些记录来了解并行程序的有关性能,如并行度、负载平衡、等待时间等。在此基础上,程序员可以改进任务划分、同步互斥组织等。然而值得注意的是,同正确性调试类似,加入记录语句可能带来探针效应,使得所记录的信息不能正确反映程序的运行状态。此外,记录信息的组织也是不可忽视的,应当考虑采用良好的可视的性能分析界面。

动态性能分析又可分为联机(On-line)和脱机(Off-line)两种。在联机方式下,信息的记录和分析是同时进行的,记录下来的信息立即送去分析。联机方式可以及时将分析结果告诉程序员,然而正由于其实时性,所进行记录和分析的信息必

然有所筛选,信息量相对较小,分析也相对简化。在脱机方式下,程序运行时仅仅记录信息,分析工作在程序运行结束后进行。这种方式,信息的记录和分析可以进行得比较充分,然而其缺点是不够及时,如果遇到死锁情况,更加不能及时反映程序运行状态。

性能分析过程结束之后,应当打印出存在的问题,尤其应当找到并行程序的性能瓶颈所在,以便下一步进行优化。

优化(Optimization)在这个阶段中,程序员根据分析的结果,对并行程序进行改进。改进完成后,一般应当先进行正确性调试,然后再进行性能测试。如仍不满意,则应当继续进行性能调试,直到程序的性能符合要求为止。

15.4 并行程序性能分析

获得高性能是并行程序的设计目标。通过性能分析指导和改进程序的设计是达到这一目标的必要方法。本节先讲述用于并行程序性能分析的静态、动态方法的一般原理,然后简单介绍性能的可视化环境。

15.4.1 并行程序的性能预测

并行程序性能的静态分析又叫性能预测(Performance Prediction),它是在源程序一级进行的。性能预测多用于并行程序的开发阶段,是一种事前的预测。它可以是由用户或编译器调用的工具,使用分析或者模拟程序运行的方法来预测并行程序的性能。编译器通过调用分析工具可以决定优化策略,多数情况下可以有效指导程序的优化。

性能预测的应用场合 性能预测可在下列场合使用:①在软件开发完成之前,开发者需要关注并行程序的行为来预测并行程序的效率,从而决定该系统是否值得开发。②系统开发进行中或已经完成时,很多时候开发者需要在不同条件下测试该程序,以决定如何获得该系统的最佳性能。③系统运行时,人们往往关注系统的可扩放性,开发者在处理器数目或输入数据量不同的情况下运行该系统,关注不同性能的对比,大部分时候是关注不同数目处理器的情况。

性能预测的方法 性能预测的方法有分析预测(Analytical Prediction)和模拟仿真两大类,比较实用的是后者。模拟仿真(Simulation)方法的原理是:在保证正确性的前提下,从开发的程序中提取出其合成模型,该模型可以用作模拟工具的输入在模拟工具上运行,以代替应用程序在真实硬件上的运行。用户然后就可以改

变系统的一些内部机制,例如处理器映射、选路、调度策略等,或改变一些系统的特征,进而观察并行程序在新环境下的性能。如果处在开发阶段,开发者可以提供该应用程序的合成骨架,其中加入一些参数对其进行特征化,该骨架也可以作为模拟器的输入,分析就可以提前,不必等到程序彻底完成。由于可以对尚未完成的程序进行分析,所以节省了很多时间和精力。

并行系统模拟仿真 它包括并行系统建模和应用程序建模两部分。

⑴ 并行系统建模:对实际并行系统进行仿真建模的时候,因实际系统的不同,模型主要可以从3个角度进行分类:①静态的还是动态的,它取决于时间这一参数的重要程度,不重要时可采用静态模型,反之,使用动态模型。②随机的还是确定的,它取决于系统中随机性因素的存在与否,不含任何概率性因素者属于确定性模型,同样的输入总得到完全相同的输出,否则就属于随机性模型的范畴。在随机性模型中,多次相同的输入得到不同的输出,所以对系统行为进行预测时,需要成组而不是单次的实验数据。③连续的还是离散的,它取决于系统状态的变化是否连续,只能取一组离散数值的系统属于离散性模型,否则属于连续性模型。

⑵ 应用程序建模:它主要着眼于将应用程序看作一个何种结构的问题。例如,从数据处理的角度可以看作数据流模型,通过定义各种规则和参数,可以对程序运行中感兴趣的方面进行比较系统的描述。

性能评估的参数选择 系统建模和应用程序建模都完成以后,就可以对应用程序的性能进行评估和预测,应用程序的性能高低可以用来自系统高层、中间层、低层的众多参数来预测。高层关注的主要参数有加速比、效率、功效等,中间层有处理器使用率、负载平衡、数据映射等,低层数据有通信延迟、带宽等参数。根据具体情况的不同,参数的选取和重要程度也不同。

离散系统的模拟 不失一般性,考虑动态系统。组成系统的元素的状态改变只在特定的时间发生,也就是只发生在离散的一系列时间点上。这些时间点的组织,既可以按照均匀采样的方式,也可以取自改变系统状态的一系列事件的发生时间。这两种组织方式下的系统分别称为离散时间系统和离散事件系统。在离散时间系统中,相邻的两个时间点之间完全可能没有任何事件发生,在此之间系统状态将没有任何改变,所以显而易见,离散时间系统是相对低效的,它可能导致无谓的计算。

连续系统的模拟 连续性模型中,状态变量可以随着时间呈连续性变化。一般来说,状态变量随时间的变化需要由微分方程来表示。如果方程简单,通过分析的方法就可以给出所有变量随时间变化的数值计算式。但对于多数连续性模型,分析方法无法胜任。然而可以使用多种积分方法,根据变量在0时刻时的数值,对微分方程进行积分,从而给出变量在其他时刻的数值。

15.4.2 并行程序的性能监控

并行程序性能的动态分析又叫性能监控(Performance Monitoring),它是对程序实际运行中的性能相关参数进行测量和分析的过程,可用于在应用程序完成后对性能进行事后的评估和测量。除此之外,监控还可以在程序实际运行期间对性能进行动态分析,调节系统参数,直接指导程序运行,从而改善性能,不过,这需要系统和应用程序的支持。

用于监控的工具可分为联机的和脱机的。联机监控在记录信息时对信息进行同步分析,并报告结果。一般用于监控数据量过大、无法全部记录的场合。另外,前面提到的程序运行中的动态调节也只能在联机的监控下实现。脱机监控则只对程序运行期间的必要参数进行记录,分析则在事后进行。脱机方式相比联机方式可以提供更大量和更准确的信息。

监控中关注的数据大致有完成时间和资源利用率两类:前者立足于程序本身的性能,后者则立足于资源利用,以便于发现系统性能的瓶颈。按照驱动方式的不同,监控分为时钟驱动监控和事件驱动监控两大类。时钟驱动是指在系统中使用一个独立进程,按照均匀时间间隔对系统参数进行记录或采样。时间间隔的长短依赖具体的操作系统,其中采样是一种特殊的记录方式。例如,记录指令计算器的值,这样,程序中一个过程里消耗的时间就大致认为和采样命中次数成比例。事件驱动则使用事件的发生作为记录的引发方式,对信息的记录独立于时间,仅当感兴趣的事件发生时才对必要参数进行记录。一般的并行程序的监控中,消息的发送和接收都是重要的关注对象。

事件驱动监控的分类 事件驱动的监控分为记时、计数和跟踪三大类。记时方式通常测量消耗在程序的各部分的时间。例如,若要测量一个过程的执行时间时,则需要在该过程刚开始和刚结束的一刻对系统时间进行记录,将两者相减。计数方式下,监控系统对关注的事件的发生次数进行记录,该方式具有记录信息量小的优点。跟踪方式下,系统在监控中跟踪并记录每一个被关注的事件,需记录的信息内容至少包括事件类型和发生的时刻。根据事件的不同,可以附加其他的信息。跟踪方式的优点是适合查找系统性能的瓶颈,适合测量系统中的通信情况,而且可以获得相对详细的数据。跟踪是最基本的事件驱动方式,并行系统的多数性能分析工具都基于跟踪方式。

跟踪的实现 跟踪的实现方式有3种:①在系统中设置专门硬件用于跟踪和记录事件,称为硬件方式;②完全用软件实现的软件方式;③特定跟踪硬件与软件相结合的混合方式。3种方法各有所长:硬件方式的跟踪数据质量高,但应用困

难、成本代价高；混合方式的跟踪数据质量较好，但开发困难，可移植性差；软件方式可移植性好、实现容易、廉价，因而是应用最广的方式，但数据质量有待提高。

软件跟踪的实现　软件跟踪可以由4种方式实现：①源代码中直接实现，即产生事件的代码在编译前被插入源代码中。代码添加的工作可以由用户手工或预处理软件完成，通常都由预处理软件完成，该方法有实现简单的优点。②编译时实现，即由编译器把信息访问的入口给予跟踪工具，信息则由编译器加工计算出来。③在通信库或运行时系统中实现。④修改目标代码实现。该方法有独立于高级编程语言的优点。

跟踪质量　跟踪数据的质量受到多种因素的影响，主要有：①存储容量对可记录数据规模的限制；②分布系统中缺乏全局时钟导致记录中的时间产生不一致性；③监控系统的加入对应用程序的行为不可避免要产生影响，这种影响又称侵扰。全局时钟可以通过在系统中加入特殊硬件来实现，侵扰则在硬件实现的跟踪中完全不存在，这就是硬件实现和混合实现的跟踪质量较高的主要原因。而在软件实现的跟踪中，就必须通过各种措施弥补与中和全局时钟的缺乏和侵扰造成的各种干扰。

实现软件跟踪时，为了保证正确性，监控系统和操作系统的交互是不可少的。例如，多用户系统中，仅仅停留在应用程序级上时，监控系统无法有效区分进程的激活和挂起状态，它只能识别有明确意义的进程挂起（例如等待接收消息），而无法识别无明确意义的挂起（如CPU进程切换时产生的挂起）。这样产生的后果是，跟踪得到的时间数据中，把自身无法检测到的挂起时间全部认为是执行时间，造成数据的错误。

15.4.3 并行程序的性能可视化

通过程序性能的静态和动态分析得到的数据，要想更加正确和准确地被理解，对这些数据进行可视化处理是必要的手段。为此，出现了很多性能可视化工具，这些工具往往都和前面的性能测量工具结合在一起。

性能可视化（Performance Visualization）包含数据生成、数据显示、数据分析与用户交互这3个有序阶段。①数据生成阶段主要任务是生成和采集数据，主要进行前面两小节中性能静态和动态分析的数据采集。②数据显示是性能可视化的核心阶段，主要将得到的各种数据进行处理并显示。可视化工具的分类主要依据这一阶段的内容。③数据分析与用户交互是最后阶段，对数据进行分析，这里的问题主要在于可视化工具本身以何种方式方便和辅助用户进行分析，涉及到很多人机交互的问题。

数据生成　这一阶段的两个核心任务是如何生成需要的数据以及如何存储数据。数据生成的常用方法是将数据存储在跟踪文件中,为此,可以使用前面讲过的静态和动态分析方法。跟踪文件通常在程序运行期间生成,但数据显示往往在程序运行结束后,这种方式称为脱机分析。为了支持实时的应用,在数据刚生成时就进行显示,在这一阶段生成的跟踪数据将以数据流的形式输送给数据显示阶段进行处理。数据文件的存储,按照文件格式分为统计文件和跟踪文件两大类:统计文件记录数据的采样或泛泛的统计信息,需要空间较小;而跟踪文件则存储详细信息,因此需要更多的空间。

数据显示　这是可视化处理的核心阶段,目标是将统计信息或跟踪信息进行显示。进行可视化处理的信息大体上按照硬件层数据和软件层数据进行区分。①对于硬件层数据,处理器、存储器、通信是重点关注的对象。在处理器方面,处理器利用率通常可以使用简单的曲线来表示,在一些要求不怎么严格的场合,还可以仅仅通过色彩来大致表示处理器的状态(忙、空闲、通信等)。在存储器方面,高速缓存的缺失信息可以通过条形图、数字显示、曲线等方式显示出统计信息;也可以采用更直观的方式——例如以方块代表具体的高速缓存块——来给出更加详细的信息,有些可视化工具在显示时,甚至做到了结合应用程序本身的数据结构。在通信方面,通信容量和频率都是重要的信息,由于通信和处理器之间的关系,通信频率往往和处理器信息结合在一起进行显示。②对于软件层数据,主要从系统和应用程序两方面来考察性能。从系统方面考察时,着眼点是任务的调度和管理以及它们之间的通信。例如进程的时间片轮换机制、优先权机制、抢占机制,通信协议中传输层数据包的大小、丢失率以及其对应关系等。从应用程序方面考察时,着眼点是任务运行、任务间通信和应用程序细节。例如,任务的运行总时间以及忙、空闲、开销三种状态下各自占用的比例,通信中消息传送开销、路由选择开销以及用于进程同步的开销等。

数据分析与用户交互　这一阶段的重点是如何使用各种方式辅助甚至部分代替用户进行性能分析。可以使用的方法有:①利用动画。动画作为一种表现方式,可以应用在任何抽象层上,它可以很方便和直观地表现数据随着时间变化的过程,是最常用的方法之一。②使用多视图。在分析性能时,设计者往往需要综合考虑来自不同抽象层的信息,从应用程序层开始,经系统层,直到硬件层。将来自不同级别的信息进行组合,同时显示,可以方便用户发现其中的规律和联系。③提供跟踪措施。在程序的设计中,一些很小的元素也可能会导致系统瓶颈的产生,这些元素小到一个语句、一行代码、或一个函数。可视化工具可以提供一些跟踪措施,使得用户可以跟踪代码行或操作。在可视化编程环境里,为提高效率,可跟踪的对象甚至可以具体到图形图像元素。④使用自动分析机制。这种方法目前还不成熟,

但已经有些系统或工具实现了比较简单的自动分析支持机制。这也是以后可视化工具的发展方向之一。

用户界面 可视化工具用户界面的友好性是设计中一个必须注意的问题。大体上,目前需要注意的主要问题是如何更好地使用色彩或其他美学元素来表示信息和如何设法处理视觉爆炸(Visual Explosion)。色彩的使用在性能可视化中起着简化信息表示的重要作用,例如用不同色谱(Color Spectrum)表示处理器或任务的状态。更进一步,色彩对信息不仅可以定性地表示,还可以定量地表示。色彩可以在不同范围内定量地表示数值的大小,例如有的系统中就提供色彩尺度(Color Scale)表用来表示处理器或进程的装载率,使用色彩的范围可以由用户自行修改。视觉爆炸指的是由于同时显示的信息量过于庞大,使得用户无所适从的现象,这种情况必须尽量设法避免。通常使用的对策是放大(Zooming)和过滤(Filtering)。通过放大化处理,可以使用户集中关注重点部分的显示;而过滤则可以减少相对次要信息的显示,同样起到突出重点的作用。目前大多数可视化工具都提供这些功能,由用户自行选择关注的重点。

15 5 图形化并行程序集成开发环境

本节介绍一个图形应用开发环境 GRADE。GRADE 支持使用 PVM 通信库的并行应用程序的完整开发过程。集成了一组现成的工具,包括图形化的程序编辑器 GRED、代码生成器 GRP2C、映射和调度工具 DSM&S、动态负载平衡工具 DLB、分布式并行程序调试器 DDBG、可视化监视工具 Tape/PVM 和 PROVE 等,它极大地简化和加速了并行程序的开发过程。

15 5 1 并行程序的可视化设计环境与工具

20 多年来,串行程序的设计经过结构化、过程化、面向对象技术、CASE 技术、组件技术和可视化编程技术等的应用,大大提高了串行软件的开发效率。面对并行程序开发困难的局面,长期来,从事并行程序设计的人员也做了大量工作,试图将串行程序的这些有效技术引入到并行程序设计中来,并且也已开发出一些试验性的可视并行程序设计语言和环境,包括 HENCE、CODE、PSEE、Paralex、TRAPPER 和 GRADE 等。它们的思路基本相同,都是用节点表示计算,用弧表示节点间的交互,通过一个可视的集成开发环境,采用统一的图形用户界面,将并行程序的设计、编辑、编译链接、调试和性能分析等工具集成起来,力图实现并行程序

开发各个阶段的可视化。当前可视化的并行软件设计工具的趋势是,在一个图形的集成环境里,支持并行软件开发的全过程。一个典型的这样的工具集至少应包括三个主要的工具:一个可视化的程序设计工具,一个可视化的模拟系统和一个可视的程序正确性调试和行为分析工具。可视的程序设计工具和行为分析工具应该并存在同一个环境中,允许有关程序行为的信息与其设计联系起来。而目前许多现有的工具集只包含这些工具的一个子集,可视化性能分析工具通常作为一个单独的工具。

15.5.2 图形应用开发环境GRADE的组成

为了处理并行程序中由进程间通信和同步引起的特有的复杂性,1994年,由匈牙利等国的研究机构在一个欧洲合作项目中开发了图形应用开发环境GRADE(GRaphical Application Development Environment)。它的主要目标是为运行在同构和异构环境中通用消息传递应用程序提供一个易于使用和集成的程序设计工具集,其主要思想是使用集成的图形化环境工具支持并行程序开发的各个步骤。

GRADE主要由以下几部分组成:①图形的并行程序编辑器GRED:它支持图形化语言GRAPNEL的语法。②预编译器GRP2C:它将GRAPNEL源程序转换成使用PVM函数的C/C++程序。③映射工具DMS&S:它使用多种映射算法将进程分布到可用的处理器上。④分布式并行程序调试器DDBG:它支持GRAPNEL应用程序的调试。⑤监视工具Tape/PVM:它用来在运行时收集PVM和GRAPNEL应用程序的事件记录。⑥可视化工具PROVE:它分析和解释Tape/PVM的跟踪文件信息,以图形化的方式呈现给用户。

15.5.3 GRADE中开发并行程序过程

参照图15.7,在GRADE环境中,开发并行程序的全过程如下(其中矩形框代表逻辑步骤,椭圆框代表不同的工具,菱形框代表不同的文件,圆形框代表库):

(1)设计并行程序(Designing Parallel Program):这一步由GRAPNEL语言(GRAphical Process NEt Language)和GRED(GRAphical EDitor)图形编辑器支持。在GRAPNEL程序中,用图形化的方法定义所有的进程管理和进程间通信活动,而底层的消息传递系统是不可见的。GRADE基于图形化的GRAPNEL代码自动产生所有的消息传递库调用。因为图形隐藏了底层消息传递库的细节,所以对没有并行程序编程经验的程序员来说,GRADE是一个理想的编程环境。GRAPNEL是一个混合型的语言。在GRAPNEL中,图形用来定义并行活动,而文本部分

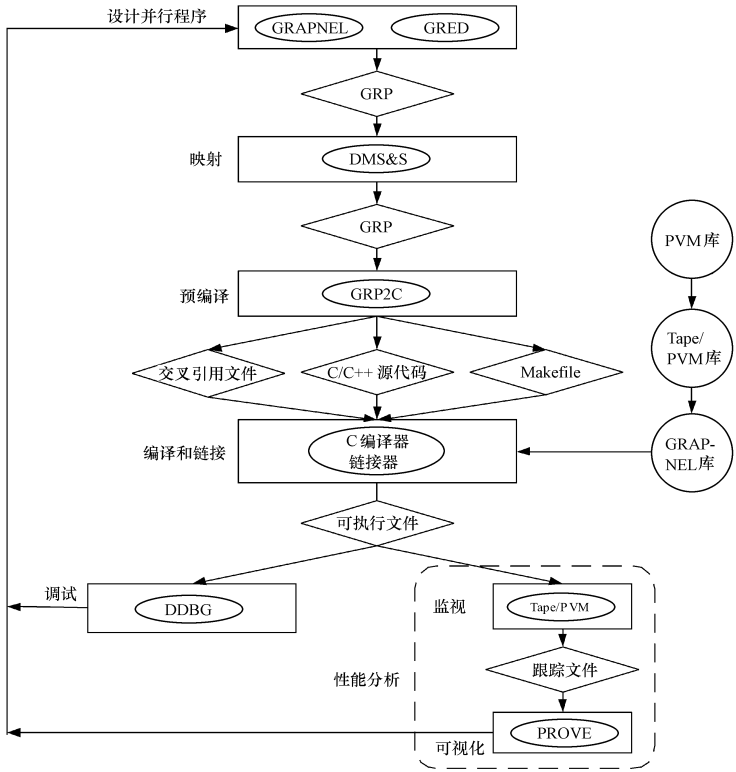


图 15.7 GRADE 中应用程序的开发过程

(C/C++或 Fortran)用来描述串行活动。GRED 使用户能够高效地构造图形部分,使用 GRED 编辑的 GRAPNEL 程序存放在 GRP 文件中,其中包含了程序的图形部分和文本部分。

(2) 映射 (Mapping):这一步可以使用一个简单的用户定义的映射表 (Mapping Table) 或者由一个复杂的自动映射工具 DMS&S 来完成。映射信息被插入 GRP 文件中。

(3) 预编译 (Pre-compilation):这一步的目标是将 GRP 文件中的图形信息翻译成消息传递库的函数调用,并生成 GRAPNEL 程序的 C/C++或 Fortran 的源代码。目前,只支持由 GRP2C 生成 C/C++的源代码。考虑到灵活性,在生成的 C/C++代码中不直接调用消息传递库的函数,而是引入一个称为 GRAPNEL 库的

内部库。除了生成 C/C++ 代码外,GRP2C 还生成一个用来支持符号调试的交叉引用文件 (Cross-Reference File),以及用来编译和链接的 make 文件。

(4) 编译 (Compiling) 和链接 (Linking):这里使用标准的 C/C++ 编译器和标准链接器,将 GRAPNEL、Tape/PVM 和 PVM 库链接到 GRAPNEL 的 C/C++ 代码,生成最终的可执行文件 (Executable File)。如果是在异构系统中,将会为每个不同类型的平台生成可执行文件。这种方案的优点是程序员不需要更新每台机器上的可执行文件。需要指出的是,从 GRADE 用户的观点看,此步和上一步都是一个单独的步骤。

(5) 调试 (Debugging):调试信息直接与用户原始的图形化代码相关。可以在通常的 C 语言指令级或者更高的图形化图标级单步执行。这个特性极大地方便了并行程序开发中最耗时的并行调试阶段。

(6) 性能分析 (Performance Analysis):首先,需要在程序执行时由性能监视器 (Performance Monitor) 生成跟踪文件;然后用各种不同的视图可视化地显示面向性能的信息。

GRADE 打算做成一个复杂的程序设计环境,在此环境下用户可以用高层工具和抽象机制来开发并行程序,无需关心消息传递原语的低层细节。所有的通信原语可以用图形的方式定义,程序员在同一个图形用户界面下即可完成程序的设计、调试和性能优化。

15.6 小结和导读

小结 并行程序的编制、调试和性能分析是比较复杂和困难的,人们需要不同的工具来帮助程序员或用户减轻这些困难和提高程序生产率;需要一个较完善的和良好的程序设计环境来帮助调试程序、监视程序的运行和分析程序的性能。将来的工具环境应该是一个集成化的开放式的人机交互方式更为自然和谐的环境。

本章除了概括性地介绍了一下软件工具 (它包含了并行程序设计工具中的编辑、编译、连接、预处理、查错等编码工具,系统维护工具以及先进的集成工具) 外,重点讨论了语言编译器。并行编译系统的技术难点是如何自动识别用户源程序中的可并行执行部分并将其并行化,其工作流程是:根据相关理论,把顺序程序转换为程序流图;将此进程流程图划分成若干并行模块;用分布式并行语言描述并行执行模块,并与系统结构相匹配;重新编译,生成可执行的目标代码。本章还对并行程序的调试和性能分析做了讨论。从本质上讲,调试是从并行程序执行错误的现象出发,通过某种手段找出错误代码的位置,就地改正,以获得一个正确的并行程

序 性能分析以改善并行程序的性能为出发点,帮助用户发现和消除性能瓶颈,以获得一个高效的优化程序。本章最后对并行程序设计的可视化环境与工具做了简单介绍,并以 GRADE 为例,描述了如何用图形应用开发环境开发一个并行程序。

导读　关于软件工具与环境的概述,读者可阅读 [145]。Emacs 编辑器描述在 [79] 中,SADT 描述在 [150] 中,OMT 描述在 [151] 中,PCTE 标准描述在 [37] 中。关于编译原理、技术和工具,读者可阅读 [8];并行编译的论述,读者可阅读 [25,109,184];Illinois 对并行化编译的研究读者可参阅 [32,33,34];有关相关性分析,读者可阅读 [197] 和 [100];关于相关测试中 GCD 测试读者可参阅 [184],Banerjee 不等式测试读者可参阅 [24];有关 Alliant FX/80 多处理机介绍,读者可参考 [10]。关于并行程序的测试和性能分析,读者可阅读 [194,39,146,48]。关于并行程序设计语言与工具,读者可阅读 [47]。关于可视化的并行程序设计环境与工具,读者可参阅 [27,130,122,21,153] 和 [107]。

习　题

15.1　分析下列代码段中三条语句间的相关关系,给出相应的相关方向向量和相关距离向量:

```
      DO  I=1,N
        DO  J=2,N
S1:       A(I,J)=A(I,J-1)+B(I,J)
S2:       C(I,J)=A(I,J)+D(I+1,J)
S3:       D(I,J)=0.1
        ENDDO
      ENDDO
```

15.2　判定下列的循环,哪些是可向量化的:

①
```
      DO  I=1,N
S1:     A(I)=B(I)
S2:     C(I)=A(I)+B(I)
S3:     E(I)=C(I+1)
      ENDDO
```

②
```
      DO  I=1,N
S1:     A(I)=B(I)+C(I+1)
S2:     C(I)=A(I)*D(I)
      ENDDO
```

③
```
      DO  I=1,N
S1:     A(I)=A(I-1)+1
      ENDDO
```

```
④    DO  I=1,N
   S1:   A(I) =A(I+1) +1
     ENDDO
⑤    DO  I=2,N
         DO  J=2,M
            A(I,J) =A(I,J-1) +1
         ENDDO
     ENDDO
```

15.3 试述向量化下列代码段：

```
①    DIMENSION A(200,200) ,B(200,200) ,C(200,200)
      DO  I=1,200
          DO  J=1,200
              A(I,J) =B(I,J) *C(I,J)
          ENDDO
      ENDDO
②    DO  I=1,100
         C(I) =0.0
         DO  J=1,100
             C(I) =C(I) +A(I,J) *B(J)
         ENDDO
     ENDDO
③    DO  I=1,N
         B(I,1) =0
         DO  J=1,M
             A(I) =A(I) +B(I,J) *C(I,J)
         ENDDO
         D(I) =E(I) +A(I)
     ENDDO
④    DO  I=2,N
   S1:   T(I) =A(I-1) +A(I+1)
   S2:   A(I) =B(I) +C(I)
     END
```

15.4 分析下列循环，哪些是可并行化的：

```
①    DO    I=2,N
          A(I) =B(I) -A(I-1)
     ENDDO
②    DO  I=2,N,2
         A(I) =B(I) -A(I-1)
     ENDDO
③    DO  I=1,N
```

```
            X＝SQRT(A(I))
            B(I)＝X＊C(I)＋X＊D(I)
        ENDDO
④      INDX＝0
        DO  I＝A,N
            INDX＝INDX＋1
            A(I)＝B(I)＋C(INDX)
        ENDDO
⑤      DO  I＝1,N
        IF(A(I)·LT·EPSILON) GOTO 320
            A(I)＝A(I)＊B(I)
        ENDDO
      320 CONTINUE
```

15.5　分析下列循环的语句数据相关性，如何进行循环调度并行化：

```
      DO  I＝1,N
          DO  J＝2,N
S1：          A(I,J)＝B(I,J)＋C(I,J)
S2：          C(I,J)＝D(I,J)／2
S3：          E(I,J)＝A(I,J－1)＊＊2＋E(I,J－1)
          ENDDO
      ENDDO
```

算 法 索 引

表 格 索 引

示范程序索引

参 考 文 献

1 Adams J et al. The Fortran 90 Handbook. McGraw-Hill, 1992
2 Adams J et al. The Fortran 95 Handbook. MIT Press, 1997
3 Adve S V, Hill M D, Vernon M. Comparison of Hardware and Software Cache and Coherence Schemes. Proc. 18th Annu. Int. Symp. Computer Arch., 1991, 298～308
4 Agerwala T, Martin J L, Mirza J H, et al. SP2 System Architecture. IBM Systems Journal, 1995, 34 (2): 152～184
5 Aggarwal A et al. Communication Complexity of PRAMs. Theoretical Computer Science, 1990, 3: 3～28
6 Agha G. Concurrent Object-Oriented Programming. Comm. of the ACM, 1990, 33 (9): 125～141
7 Aho A, Hopcroft J, Ullman J. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974
8 Aho A V, Sethi R, Ullman J D. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986
9 Allan S J, Oldehoeft R. HEP SISAL: Parallel Functional Programming. Kowalik (Ed). Parallel MIMD Computation: HEP Supercomputers and Applications. MIT Press, 1985
10 Alliant. Alliant Product Summary. Alliant Computer Systems Corporation, 1989
11 Alpern B et al. The Uniform Memory Hierarchy Model of Computation. Algorithmic, 1993
12 Amdahl G M. Validity of Single-Processor Approach to Achieving Large-Scale Computing Capability. Proc. AFIPS Conf., 1967, 483～485
13 Amza C et al. Treadmarks: Shared Memory Computing on Networks of Workstations. IEEE Computer, 1996, 29 (2): 18～28
14 Anderson E et al. LAPACK User's Guide. SIAM, 1992
15 Anderson T E et al. A Case for NOW (Networks of Workstations). IEEE, Micro, 1995, 2: 54～64
16 ANSI Technical Committee X3H5. Parallel Processing Model for High-level Programming Languages, 1993
17 Accelerated Strategic Computing Initiative (ASCI) Program. A Report by US Department of Energy, Lawrence Livermore National Lab., Los Alamos National Lab., Sandia National Lab., 1996
18 ATM Forum. ATM User-Network Interface: Version UNI3.1 Specification, 1994
19 Averbuch A, Gabber E, Gordissky B et al. A Parallel FFT on an MIMD Machine. Parallel

Computing,1991,15(6):1～74
20 Baase S. Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley,1978
21 Babaoglu O et al. Paralex: An Environment for Parallel Programming in Distributed Systems. Proc. of ACM Int'l Conf. on Supercomputing,1992
22 Bailey D H et al. The NAS Parallel Benchmarks 2.0. NASA Ames Research Center Report, 1995,12: NAS-95-020
23 Bal H E, Steiner J G, Tanenbaum A S. Programming Languages for Distributed Computing Systems. ACM Computing Surveys,1989,21(3): 261～322
24 Banerjee U. Dependence Analysis for Supercomputing. Boston: Kluwer Academic Press,1988
25 Banerjee U. Dependence Analysis. Boston: Kluwer Academic Publishers,1996
26 Batcher K E. Sorting Networks and Their Applications. AFIPS Proc.1968,307～314
27 Beguelin A et al. Visualization and Debugging in a Heterogeneous Environment. IEEE Computers,1993,26(6)
28 Berger M, Bokhair S. A Partitioning Strategy for Nonuniform Problems on Multiprocessors, IEEE EX Tran. Computs.,1987,C-36(5): 570～580
29 Berman F, Snyder L. On Mapping Parallel Algorithms into Parallel Architectures. Parallel J & Distributed Computing,1987,4(5): 439～458
30 Berntsen J. Communication Efficient Matrix Multiplication on Hypercubes. Parallel Computing, 1989,12: 335～342
31 Bertsekas D P, Tsitsiklis J N. Parallel and Distributed Computation: Numerical Methods. Prentice-Hall,1989
32 Blume W, Eigenmann R. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. IEEE Trans. on Parallel and Distributed Systems,1992,3(6): 643～656
33 Blume W et al. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. IEEE Parallel and Distributed Technology,1994,2(3): 37～47
34 Blume W et al. Parallel Programming with Polaris. IEEE computer,1996,29(12): 78～82
35 Blumrich M A, Dubnicki C, Felten E W et al. Protected User-Level DMA for the SHRIMP Network Interface, Proc. 2nd Int'l Symp. on High-Performance Computer Architecture,1996
36 Boden N J et al. Myrinet: A Gigabit-Per-Second Local Area Network. IEEE Micro.1995,2: 29～36
37 Boudier G et al. An Overview of PCTE+. SIGPLAN,1982,2(24): 248～257
38 Boyer R S, Moore J S. A Fast String Searching Algorithm. Comm. of the ACM,1997,20: 762～772
39 Brown J S. Debuggers for High Performance Computers, Proc. of the Supercomputing'93,1993
40 Cannon L E. A Cellular Computer to Implement the Kalman Filter Algorithm: Ph. D. thesis. Montana State Univ.,1969
41 Carriero N, Gelernter D. How to Write Parallel Programs. MIT Press,1990
42 Catlett C, Smarr L. Metacomputing. Comm. of ACM,1992,35(6): 44～52
43 Chaiken D, Fields C, Kwibara K et al. Director-Based Cache Coherence in Large-Scale Multiprocessor. IEEE Computer,1990,23(6): 49～59

44 Chan T F, Saad Y. Multigrid Algorithms on the Hypercube Multiprocessor. IEEE_TC, 1986, C-35 (11): 969～977
45 Chapman B et al. Extending HPF for Advanced Data_Parallel Applications. IEEE Parallel & Distributed Technology, 1994, 2 (3): 15～27
46 Cheatham T, Fahmy A, Stefanescu D C et al. Bulk Synchronous Parallel Computing—A Paradigm for Transportable Software. Proc. of 28th Hawaii Int. Conf. on System Science, 1995
47 Cheng Y. A Survey of Parallel Programming Languages and Tools. Technical Report RND-93-005, NASA Ames Research Center, 1993
48 Cheng D, Hood R. A Portable Debugger for Parallel and Distributed Programs, Proc. of the Supercomputing'94. 1994
49 Chlebus B, Vrto I., Parallel Quick Sort. Journal of Parallel and Distributed Computing, 1991, 11: 332～337
50 Chronopoulos A T, Gear C W. On the Efficient Implementation of Pre-condition S-step Conjugate Gradient Methods on Multiprocessors with Memory Hierarchy. Parallel Computing, 1989, 11: 37～53
51 Cole R, Vishkin U. Approximate Coin Tossing with Applications to List, Tree, and Graph Problems, Proc. 27th annu. IEEE Symposium on FOCS (Foundations of Computer Science), IEEE press1986, 478～491
52 Comer D E. Internetworking with TCP/IP. 3rd Ed. Prentice_Hall, 1995
53 Cooley J W, Tukey T W. An Algorithm for the Machine Calculation of Complex Fourier Series. Mathenatics of computation, 1965, 19 (90): 297～301
54 Cormen T H, Leiserson C E, Rivest R L. Introduction to Algorithms. MIT Press, 1990
55 Culler D, Karp R, Patterson D, et al. LogP: Towards a Realistic Model of Parallel Computation. Proc. of 4th Symp. on Principles & Practice of Parallel Programming, ACM SIGPLAN, 1993, 1～12
56 Cybenko G. Dynamic Load Balancing for Distributed Memory Multiprocessors. Parallel J & Distributed Computing, 1989, 7: 279～301
57 DeFanti T et al. Overview of the I-WAY: Wide Area Visual Supercomputing. Int'l J. of Supercomputer Applications, 1996, 10 (2)
58 Dekel E, Nassimi D, Sahni S. Parallel Matrix and Graph Algorithms. SIAM J. on Computing, 1981, 10: 657～673
59 Dijkstra E D, Seijen W H, Gasteren A J M V. Derivation of a Termination Detection Algorithm for a Distributed Computation. Info. Proc. Lett., 1983, 16 (5): 217～219
60 Dincer K, Fox G C. Building a World-Wide Virtual Machine Based on Web and HPCC Technologies. Supercomputing'96 Conference Proceedings, 1996
61 Dongarra J J et al. LINPACK User's Guide. SIAM. 1979
62 Duato J, Yalamanchili S, Ni L. Interconnection Networks: An Engineering Approach. IEEE Computer Society Press, 1997
63 Felten E W et al. Early Experience with Message-Passing on the SHRIMP Multicomputer.

Proc.of the 23rd Int'l Symp.on Computer Architecture,1996
64 Feng T Y.A Survey of Interconnection Networks.IEEE Computer,1981,14(12):12～27
65 S Fortune,J Wyllie.Parallelism in Random Access Machines.Proc.of 10th Symp.on Theory of Computing,1978,114～118
66 Fosdick L D,Schauble C J C.Computer Performance.A Tutorial High Performance Scientific Computing.Boulder:Univ.of Colorado,1994
67 Foster I T.Designing and Building Parallel Programs:Concepts & Tools for Parallel Software Engineering.Addison_Wesley Pub.Company,1995
68 Fox G C,Otto S W,Hey A J G.Matrix Algorithms on Hypercube I:Matrix Multiplication.Parallel Computing,1987,4:17～31
69 Fox G et al.Solving Problems on Concurrent Processors.Prentice_Hall,1988
70 Fox G et al.FORTRAN D Language Specification.Rice Univ.,1992.
71 Williams G Fox R,Messina P.Parallel Computing Works! Morgan Kaufman,1994
72 Galil Z.Optimal Parallel Algorithms for String Matching.Info.and Control,1985,67(1～3):144～157
73 Gallivan K A,Plemmons R J,Sameh A H.Parallel Algorithms for Dense Linear Algebra Computations.SIAM Rev.,1990,32(1):54～135
74 Geist A et al.PVM:Parallel Virtual Machine——A User's Guide and Tutorial for Networked Parallel Computing.MIT Press,1994
75 Gigabit Ethernet Alliance.White Paper of Gigabit Ethernet,March,1997(http://www.gigabit_ethernet.org/)
76 Gibbons P B.A More Practical PRAM Model.Proc.of the ACM Symp.on Parallel Algorithms and Architectures,ACM,1989,158～168
77 Goldberg A,Plotkin S,Shannon G.Parallel Symmetry_Breaking in Sparse Graphs.Proc.of 19th Annu.ACM Symp.on Theory of Computing,1989,351～324
78 Golub G H,Loan C V.Matrix Computations.(2nd Ed).The Johns Hopkins Univ.Press,1989
79 Gosling J.Unix Emacs.Carnegie Mellon Computer Science Dept.,1982
80 Goudreau M W et al.The Green BSP Library.CS_TR_95_11,Univ.of Central Florida,1995
81 Goudreau M W et al.A Proposal for a BSP Worldwide Standard Library.http://www.bsp_worldwide.org/,1996
82 Goudreau M W et al.Towards Efficiency and Portability:Programming with the BSP Model.SPAA'96,1996,1～12
83 Grimshaw A et al.Legion:The Next Logical Step Towards a Nationwide Virtual Computer.Tech.Report CS_94_21,Dept.of CS,Univ.of Virginia,1994
84 Gupta A,Kumar V.The Scalability of Matrix Multiplication Algorithms on Parallel Computers.Proc.Int'l 93 Conference on Parallel Processing,1993,III～115,III～119
85 Gustafson J L.Reevaluating Amdahl's Law.Comm.Of ACM,1988,(31)5:532～533
86 Hac A.Load Balancing in Distributed Systems:A Summary.Performance Evaluation Review,1989,16(2):17～19
87 Hambrusch S E,Khokhar A K.C3:An Architecture independent Model for Coarse_Grained

Parallel Machines. Purdue Univ., 1993

88 Brinch Hansen P. Studies in Computational Science: Parallel Programming Paradigms. Prentice Hall, 1995

89 Heath M, Rosenberg A, Smith B. The Physical Mapping Problem for Parallel Architectures. J. ACM, 1998, 35 (3): 603～634

90 Heath M T, Ng E, Peyton B W. Parallel Algorithms for Sparse Linear Systems. SIAM Review, 1991, 33: 420～460

91 Heller D. A Survey of Parallel Algprithms in Numerical Linear Algebra. SIAM Review, 1978, 22: 740～777

92 Hillis W D, Steele G L. Data Parallel Algorithms. Comm. ACM, 1986, 29 (12): 1170～1183

93 Hirschberg D S. Parallel Algorithms for the Transitive Closure and the Connected Component Problem. Proc. of 8th Annu. ACM STOC, 1976, 55～57

94 Ho C T, Johnsson S L, Edelman A. Matrix Multiplication on Hypercubes using Full Bandwidth and Constant Storage. Proc. Int'l 91 Conference on Parallel Processing, 1997, 447～451

95 Hoare C A R. Quicksort. Computer Journal, 1962, 5: 10～15

96 Hockney R W. A Fast Direct Solution of Poisson's Equation using Fourier Analysis. J. of ACM, 1965, 12 (1): 95～113

97 Horowitz E, Zorat A. Divide and Conquer for Parallel Processing. IEEE Trans. on Computers, 1983, C-32 (6): 582～585

98 Grand Challenges 1993: High Performance Computing and Communications. The FY1993 US R&D Program: A Report by the Committee on Physical, Mathematical and Engineering Sciences, Federal Coordinating Council for Science, Engineering and Technology, Office of Science & Technology Policy to Supplement the President's Fiscal Year Budget: 1993

99 Huang Y, Paker Y. A Parallel FFT Algorithm for Transputer Networks. Parallel Computing, 1991, 17: 895～906

100 Hwang K. Advanced Computer Architecture: Parallelism, Scalability, and Programmability. McGraw-Hill, Inc., 1993

101 Hwang K, Xu Z. Scalable Parallel Computers for Real-Time Signal Processing. IEEE Signal Processing, 1996, 13 (4): 50～66

102 Hwang K. Gigabit Networks for Building Scalable Multiprocessors and Multicomputer Clusters. HKIE Transactions. Hong Kong Institute of Engineering, Hong Kong, 1997

103 Hwang K, Xu Z. Scalable Parallel Computing: Technology, Architecture, Programming. WCB/McGraw-Hill Companies, 1998

104 JaJa J. An Introduction to Parallel Algorithms. Addison-Wesley Pub. Company, 1992

105 Jones M, Plassmann P. Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes. Proc. 1994 Scalable High-Performance Computing Conf., IEEE Computer Society, 1994, 478～485

106 Juurlink B H H, Wijshoff H A G. A Quantitative Comparison of Parallel Computation Models. SPAA'96, ACM, 1996, 13～24

107 Kacsuk P et al. A Graphical Development and Debugging Environment for Parallel Programs.

Parallel Computing,1997,22:1747～1770
108 Karp R M et al. Efficient PRAM Simulation on a Distributed Memory Machine. Proc. of 24th Annual ACM Symp. of the Theory of Computing,1992,318～326
109 Kennedy K. Compiler Technology for Machine_Independent Parallel Programming. Int'l J. of Parallel Programming,1994,22(1):79～98
110 Knuth D E, Morris J H, Pratt V B. Fast Pattern Matching in String. SIAM J. Computing, 1997,6(2):189～195
111 Koelbel C et al. The High Performance Fortran Handbook. MIT Press,1994
112 Kumar V, Rao V N, Parallel Depth First Search, Part II: Analysis. Int'l J. of Parallel Programming,1987,16(6):501～519
113 Kumar V, Gupta A, Rao V. Scalable Load Balancing Techniques for Parallel Computers. J. Parallel & Distributed Computing,1994,22(1):60～79
114 Kumar V, Gupta A, Gupta A et al. Introduction to Parallel Computing: Design and Analysis of Algorithms. Benjamin/Cummings Publishing Company, Inc.,1994
115 Kung H T. Why Systolic Architectures? Computer,1982,15(1):37～46
116 Ladner R E, Fisher M J. Prefix Computation. JACM,1980,27(4):831～838
117 Laudon J, Lenoski D. The SGI Origin: A CC_NUMA Highly Scalable Server. Proc. of the 24th Int'l Symp. Computer Architecture,1997,241～251
118 Lauria M, Chien A. MPI_FM: High Performance MPI on Workstation Clusters. J. of Parallel and Distributed Computing,1997,40(1):4～18
119 Leiserson C L. Fat tree: Universal Networks for Hardware_Efficient Supercomputing. IEEE Trans. on computers,1985,C-34(10):892～901
120 Lin F C H, Keller R M. The Graph Model Load Balancing Method. IEEE Trans. software Eng.,1987,SE_13(1):32～38
121 Lo V. Heuristic Algorithms for Task Assignment in Distributed Systems. IEEE Trans. Computs.,1988,C-37(11):1348～1397
122 Luque E et al. Overview and New Trend on PSEE. IEEE software,1992
123 Mattson T G, Scott D, Wheat S. A Teraflops Supercomputer in 1996: The ASCI Tflops System. Proc. of the 6th Int'l Parallel Processing Symp.,1996,84～93
124 McBryan O. An Overview of Message Passing Environments. Parallel Computing,1994,20(4): 417～444
125 Mehrotra P et al. High Performanc Fortran: History, Status and Future. Parallel Computing, 1998,24 325～354
126 Mellor_Crummey J M, Scott M L. Algorithms for Scalable Synchronization on Shared Memory Multiprocessors. ACM Trans. Computer Systems,1991,9(1):21～65
127 Messina P, Sterling T (Eds). System Software and Tools for High Performance Computing Environment. SIAM,1993
128 MPI Forum. MPI: A Message Passing Interface, Proceedings of Supercomputing'93. IEEE Computer Society,1993,878～883
129 National Research Center for Intelligent Computing Systems. Introduction to the Dawning_1000

Massively Parallel Processing System, 1996

130 Newton P, Browne J C. The CODE 2.0 Graphical Parallel Programming Language. Proc. of ACM Int'l Conf on Supercomputing, 1992

131 Nicol D M, Saltz J H. An Analysis of Scatter Decomposition. IEEE Trans. Computs., 1990, C-39 (11): 1337～1345

132 Nussbaumer H J. Fast Fourier Transform and Convolution Algorithms. 2nd ed. Springer-Verlag, 1982

133 OpenMP Standards Board. OpenMP: A Proposed Industry Standard API for Shared Memory Programming, Oct. 1997

134 OpenMP Standards Board. OpenMP Fortran Application Program Interface Version 1.0, Oct. 1997.

135 Ortega J M, Voigt R G. Solution of Partial Differential Equations on Vector and Parallel Computers. SIAM Review, 1985, 27 (2): 149～240

136 Ortega J M. Introduction to Parallel and Vector Solution of Linear Systems. Plenum, 1988

137 Pancake C M. Software Support for Parallel Computing: Where are We Headed? Comm. of the ACM, 1991, 34 (11): 53～64

138 Papadimitriou C H, Yannakakis M. Towards an Architecture-Independent Analysis of Parallel Algorithms. Proc. of 20th Annual ACM Symp. of the Theory of Computing, 1998, 510～513

139 Parnas D, Clements P. A Rational Design Process: How and Why to Fake It. IEEE Trans. Software Eng., 1986, SE-12 (2): 251～257

140 Pfister G F. In Search of Clusters. Prentice-Hall PTR, 1995

141 IEEE, POSIX P1003.4a: Threads Extension for Portable Operating Systems, IEEE, 1994

142 Pothen A, Simon H, Liou K. Partitioning Sparse Matrices with Eigenvectors of Graph. SIAM J. Math. Anal. Appl., 1990, 11 (3): 430～452

143 Quinn M J. Parallel Computing-Inc. Theory and Practice (second edition) McGraw-Hill., 1994

144 Reinhardt' S K, Pfile R W, Wood D A. Decoupled Hardware Support for Distributed Shared Memory. Proc. of the 23rd Int'l Symp. on Computer Architecture, 1996

145 Reiss S P. Software Tools and Environments. ACM Computing Surveys, 1996, 28 (1): 281～284

146 Ries B. The Paragon Performance Monitoring Environment. Proc. of the Supercomputing'93, 1993

147 Rinard M C, Scales D J, Lam M S. Jade: A High-level, Machine-independent Language for Parallel Programming. IEEE Computer, 1993, 26 (6): 28～38

148 Rokusawa K, Ichiyoshi N, Chikayama T et al. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. Proc. 1998 Int'l Conf. on Parallel Processing, 1988, I: 18～22

149 Romine C H, Ortega J M. Parallel Solution of Triangular Systems of Equations. Parallel Computing, 1988, 6: 109～114

150 Ross D T. Applications and Extensions of SADT. IEEE Computer, 1985, 18 (4): 25～35

151 Rumbaugh J et al. Object-Oriented Modeling and Design. Prentice-Hall, 1991

152 Sadayappan P, Ercal F. Nearest Neighbor Mapping of Finite Element Graphs onto Processor Meshes. IEEE Trans. Computs., 1987, C-36 (12): 1408～1424

153 Scheidler C, Schafers L. TRAPPER: A Graphical Parallel Programming Environment for Industrial High Performance Applications. Proc. of PARLE'93: Parallel Architectures and Languages, 1993

154 Sedgewick R. Implementing Quicksort Programs. Communication of the ACM, 1978, 21 (10): 847～857

155 Silicon Graphics: Origin 200 and Origin 2000 Technical Report: Silicon Graphics Computer Systems, 1997

156 Shamos M I. Computational Geometry. PhD thesis, 1978

157 Shi H M, Schaeffer J. Parallel Sorting by Regular Sampling. J. of Parallel and Distributed Computing, 1992, 14 (4): 316～372

158 Shiloach Y, Vishkin U. Finding the Maximum, Merging and Sorting in a Parallel Computational Model, 1981, 2 (1): 88～102

159 Silicon Graphics. IRIS Power C User's Guide. Silicon Graphics Computer Systems, 1989

160 Simon H. Partitioning Unstructured Problems for Parallel Processing. Computing Systems in Eng., 1991, 2 (2/3): 135～148

161 Singh V, Kumar V, Agha G et al. Efficient Algorithms for Parallel Sorting on Mesh Multicomputers. International Jounal of Parallel Programming, 1991, 20 (2): 95～131

162 Skillicorn D B, Hill J M D, McColl W F. Questions and Answers about BSP. TR-15-96. Oxford Univ. Computing Laboratory, 1996

163 Snir M et al. The Communication Software and Parallel Environment of the IBM SP2. IBM Systems Journal, 1995, 34 (2): 205～221

164 Stallings W (Ed). Advances in ISDN and Broadband ISDN. IEEE Computer Society Press, 1992

165 Stallings W (Ed). Advances in Local and Metropolitan Networks. IEEE Computer Society Press, 1994

166 Stallings W. Operating Systems (2nd Ed). Prentice-Hall, 1995

167 Stamper D A. FDDI Handbook: High-Speed Networking Using Fiber and Other Media. Addison-Wesley, 1994

168 Stenstrom P, Joe T, Gupta A. Comparative Performace Evaluation of Cache-Coherent NUMA and COMA Architectures. Proc. 19th Annu. Int. Symp. Computer Arch., 1992, 86～91

169 Stunkel C B et al. The SP2 High-Performance Switch. IBM Systems Journal, 1995, 34 (2): 185～204

170 Sun X H, Ni L M. Another View of Parallel Speed. Proc. Supercomputing'90, 1990, 324～333

171 Sun X H, Rover D T. Scalability of Parallel Algorithm-Machine Combinations. IEEE Trans. on Parallel and Distributed Systems, 1994, 5 (6): 599～613

172 Swarztrauber P N. Multiprocessor FFTs. Parallel Computing, 1987, 5: 197～210

173 Tolmie D, Flanagan D. HiPPI: It's Not Just for Supercomputers Anymore. Data

Communication (http://www.data.com/)
174 Trew A, Wilson G (Eds). Past, Present Parallel: A Survey of Available Parallel Computer Systems. Springer_Verlag, 1991
175 Valiant L G. Parallelism in Comparison Problems. J. of SIAM, Computing, 1995, 4 (3): 348～355
176 Valiant L G. A Bridging Model for Parallel Computation. Comm. of ACM, 1990, 33 (8): 103～111
177 Vishkin U. Optimal Parallel Matching in Strings. Info. and Control, 1985, 67 (1_3): 91～113
178 Wagar B A. Hyperquicksort: A Fast Sorting Algorithm for Hypercubes. Proc. of the Second Conference on Hypercube Multiprocessors, 1987, 292～299
179 Wah B W, Li G J, Yu C F. Multiprocessing of Combinatorial Search Problems. Computers, 1985, 18 (6): 93～108
180 Warschko T M, Blum J M, Tichy W F. The Parastatim Project: Using Workstations as Building Blocks for Parallel Computing. Proc. of Int'l Conf. on Parallel and Distributed Processing, Techniques and Applications (PDPTA'96), 1996, 1: 375～386
181 Whitney S et al. The SGI Origin Software Environment and Application Performance. Proc. of COMPCON Spring 97, IEEE Computer Society, Feb. 1997, 165～170
182 Wilson G V, Lu P (Eds). Parallel Programming Using C++. MIT Press, 1996
183 Wolfe M J Automatic Vectorization, Data Dependence, and Optimizations for Parallel Computers, in Hwang and DeGroot (Eds): Parallel Processing for Supercomputing and Artificial Intelligence, McGraw_Hill, 1989
184 Wolfe M. High_Performance Compilers for Parallel Computing. Addison_Wesley, Pub. Company, 1996
185 Wyllie J C. The Complexity of Parallel Computations. [PhD thesis]. Cornell Univ., 1979
186 Xu Z, Hwang K. Coherent Parallel Programming in C//. Proc. of Int'l Conf. on Advances in Parallel and Distributed Computing, IEEE Computer Society Press, Mar. 1997, 116～122
187 Zhang X D, Yan Y, He K Q. Latency Metric: An Experimental Method for Measuring and Evaluating Parallel Program and Architecture Scalability. J. of Paraldl and Distributed Computing, 1994, 22: 392～410
188 Zima H et al. Vienna FORTRAN_A Language Specification. ICASE, 1992. Version 1.1
189 陈国良. 平衡分组选择网络: 计算机研究与发展, 1984, 21 (11): 9～21
190 陈国良编著. VLSI 计算理论与并行算法. 合肥: 中国科学技术大学出版社, 1991
191 陈国良编著. 并行算法的设计与分析 (修订版). 北京: 高等教育出版社, 2002
192 陈国良. 并行算法的可扩放性分析. 小型微型计算机系统, 1995, 16 (2): 10～16
193 陈国良, 吴俊敏, 章锋, 章隆兵编著. 并行计算机体系结构. 北京: 高等教育出版社, 2002
194 陈国良等编著. 并行算法实践. 北京: 高等教育出版社, 2003
195 国家智能计算机研究开发中心. 曙光 1 号智能化共享存储多处理机系统. 技术报告, 1993
196 胡伟武. 共享存储系统结构. 北京: 高等教育出版社, 2001
197 李晓梅, 蒋增荣等编著. 并行算法 (第五章). 湖南科技出版社, 1992
198 曙光信息产业有限公司. 曙光天潮系列曙光 2000 超级并行计算机. 技术白皮书. 1998

199 沈志宇等编著.并行程序设计.长沙:国防科学技术大学出版社,1997
200 孙家昶等编著.网络并行计算与分布式编程环境.北京:科学出版社,1996
201 唐策善,梁维发编著.并行图论算法.合肥:中国科学技术大学出版社,1991
202 王鼎兴,陈国良编著.互联网络结构分析.北京:科学出版社,1990
203 王鼎兴,庄伟强.一种实现并行计算的新主流技术——NOW.小型微型计算机系统,1995,16(2):29～34
204 王嘉谟,沈毅主编.并行计算方法(上册).北京:国防工业出版社,1987
205 徐士良编著.计算机常用算法(第二版).北京:清华大学出版社,1996

并行与分布计算 Web 网址

Parallel and Distributed Computing Bibliography on WWW

- Distributed Algorithms and/or Distributed Systems
  (http://www.cwi.nl/cwi/departments/AA/distcom/distcom.html)
- The CSP archive at Oxford (http://www.comlab.ox.ac.uk/archive/csp.html)
- Digital Technical Reports (http://www.research.digital.com/wrl/techreports/)
- Bibliographies on Parallel Processing
  (http://liinwww.ira.uka.de/bibliography/Parallel/index.html)
- Transputer, Occam and Parallel Computing Archive
  (http://unix.hensa.ac.uk/parallel/index.html)
- Parallel and Heterogeneous Publications
  (http://www-cse.ucsd.edu/users/jenny/parpubs.html)
- IEEE/CS ParaScope (http://computer.org/parascope/), world wide parallel computing sites
- High Performance Computing Lists
  (http://www.cs.colorado.edu/homes/mcbryan/public_html/bb/2/summary.html)
- The Language List (http://cuiwww.unige.ch/langlist) enumerate programming languages
- TOP 500 (http://www.netlib.org/benchmark/top500.html)
  World's TOP 500 most powerful computing sites (at Netlib, University of Tennessee)

News Groups about Parallel and Distributed Computing

- Architecture (news://comp.arch)
- Benchmarks (news://comp.benchmarks)
- MPI (news://comp.parallel.mpi)
- OS Research (news://comp.os.research)
- Parallel (news://comp.parallel)
- PVM (news://comp.parallel.pvm)
- Supercomputer (news://comp.sys.super)

Parallel Computer Models

- Bulk Synchronous Parallel (BSP) (http://www.comlab.ox.ac.uk/oucl/oxpara/bsplib.html).
- The Concurrent Systems site (http://www.comlab.ox.ac.uk/archive/concurrent.html) at Oxford Univ. links to theoretical and formal models for parallel and distributed systems
- Parallel Monte Carlo simulation at Berkeley (http://dnclab.berkeley.edu/codef/may/prj.html).
- Numerical recipes (http://cfata2.harvard.edu/nr/nrhome.html) has on line book and code
- PLAPACK, InterCom (iCC), SUMMA (http://www.cs.utexas.edu/users/rvdg) at Austin

Benchmarks and Performance

- Netlib Repository (http://www.netlib.org/liblist.html), has performance data on many benchmarks: LINPACK, LAPACK, BLAS, BLACS, Livermore loops, Dhrystone, Whetstone, NAS, SPEC, Sim, etc. It contains sources for some benchmarks.
- PARKBENCH (PARallel Kernels and BENCHmarks) (http://netlib2.cs.utk.edu/parkbench/)

Microprocessors

- DEC Alpha home page (http://www.digital.com/semiconductor/alpha/alpha.htm)
- SGI MIPS Group site (http://www.sgi.com/MIPS/)
- IBM's PowerPC microprocessors site (http://www.chips.ibm.com/products/ppc/)
- Sun SPARC page (http://www.sun.com/sparc/)
- Fujitsu TurboSPARC Microprocessors (http://www.fujitsumicro.com/sparcupgrade/sparcmicro.html).

Systems Interconnects

- Myrinet (http://www.myri.com)
- PCI SIG Home Page (http://www.teleport.com/~pc2/pcisigindex.html).
- The ATM Forum (http://atmforum.com/)
- SCI News (http://www.dolphinics.no/SCI_News/SCI_News.html) at Dolphin
- Interconnect Solutions (http://www.dolphinics.no/Dolphin.html).

Distributed Memory Technology

- DSM bibliography (http://www.cs.ualberta.ca/~rasit/dsmbiblio.html).
- DICE (http://www.mount.ee.umn.edu/~dice/)
- FLASH (http://www-flash.stanford.edu/)
- NUMAchine at the Univ. of Toronto (http://www.eecg.toronto.edu/EECG/RESEARCH/ParallelSys/numachine.html).

- Simple COMA (http://playground.Sun.COM:80/pub/S3.mp/simple-coma/)

Threads
- Bibliography on multithreading (http://liinwww.ira.uka.de/bibliography/Os/threads.html).
- Pthread at MIT (http://www.mit.edu:8001/people/proven/pthreads.html).

Synchronization and Communication
- High performance synchronization papers and codes.
  (http://www.cs.rochester.edu/users/faculty/scott/synchronization.html).
- Berkeley Active Message page (http://now.cs.berkeley.edu/AM/active_messages.html).
- Active Messages for SP_2 (http://www.cs.cornell.edu/Info/Projects/CAM/sp2.html) at Cornell.

File Systems
- IBM SPs GPFS (http://www.rs6000.ibm.com/sofware/sp_product/gpfs.html) and PIOFS (http://www.rs6000.ibm.com/software/sp_products/piofs.html).
- The HA NFS project (http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mootaz/ftp/html/hanfs.html)
- The Berkeley xFS (http://now.cs.berkeley.edu/Xfs/xfs.html)
- CIFS (Common Internet File System) (http://www.microsoft.com/intdev/cifs/)
- WebNFS (http://www.sun.com/webnfs/index.html).

Workload Management (Load sharing, load balancing, batching, scheduling)
- IBM LoadLeveler (http://www.rs6000.ibm.com/software/sp_products/loadlev.html).
- Scheduling, batch processing, load balancing (http://www.epm.ornl.gov/~zhou/lds/lds.html) at Oak Ridge National Laboratory

Symmetric Multiprocessor (SMP)

- SGI Power Challenge white paper (ftp://ftp.sgi.com/sgi/whitepaper/challenge_paper.ps.Z)
- Intels Multiprocessor Specification (http://www.intel.com/IAL/processr/mpovr.html)
- Digital AlphaServer Family (http://www.digital.com/alphaserver/)
- HP Enterprise Server page (http://www.hp.com/gsy/products.html).
- IBM Server page (http://www.rs6000.ibm.com/hardware/#eservers)
- Sun SMP Architectures

(http://www.sun.com/smi/ssoftpress/books/Catanzaro/Catanzaro.html).
- The Cray Research system page (http://www.cray.com/products/systems)

CC-NUMA and NCC-NUMA System
- Sequent NUMA-Q (http://www.sequent.com/numaq/technology/)
- SGI/Cray Origin 2000 (http://www.sgi.com/Products/hardware/servers/index.html).
- Cray T3E (http://www.cray.com/products/systems/crayt3e/)

Massively Parallel Processors (MPPs and HPC)
- DEC HPC site (http://www.digital.com/info/hpc/welcome.html)

- HPC Modernization Program (http://www.aero.hq.nasa.gov/hpcc/petaflops/)
- PetaFLOPS web site (http://www.aero.hq.nasa.gov/hpcc/)
- NASA HPCC Program (http://www.aero.hq.nasa.gov/hpcc/)
- Cray T3E (http://www.cray.com/products/systems/crayt3e/)
- IBM SP (http://www.rs6000.ibm.com/hardwaere/largescale/)
- Intel Paragon (http://www.sdsc.edu/Services/Consult/Paragon/paragon.html).

Clustering Technology and Systems
- Kai Li (http://www.cs.princeton.edu/~li/)
- Overview of Checkpoint (http://warp.dcs.st_and.ac.uk/warp/systems/checkpoint/)
- Single System Image and Metacomputing
- Berkeley WebOS (http://now.CS.Berkeley.EDU/WebOS/)
- Solaris MC (http://www.sun.com/960201/cover/solaris_mc.html).
- National Scalable Cluster Project (NSCP) (http://www_lac.eecs.uic.edu/NSCP2.html).
- SP2 at MHPCC (http://www.mhpcc.edu/doc/SP2.general/SP2.general.html)
- Sequent Symmetry 5000 Cluster (http://www.sequent.com/public/solution/server/s5000cl.html).
- Sun Microsystems Ultra HPC Cluster (http://www.sun.com/products_n_solutions/hw/servers/product/PDB/)

Parallel Programming Models, Languages and Tools
- X3H5 Parallel Processing Constructs for High Level Programming Languages (http://www.x3.org/tc_home/x3h5.html), at ANSI (inactive)
- ANL Shared_Variable Macros on Solaris (ftp://dit.lth.se/pub/sun_thread_ANL_macros)
- uC++ (http://plg.uwaterloo.ca/~pabuhr/uC++.html)

- Cray MPP Fortran Model
  (ftp://ftp.cray.com/product_info/program_env/program_model.html).
- Linda Group (http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html).
- The Fortran M Programming Language (http://www.mcs.anl.gov/fortran_m/index.html).
- Introduction to PVM (http://www.mhpcc.edu/training/workshop/html/pvm/PvmIntro.html) from Maui workshop
- MPI Standard site (http://www.mcs.anl.gov/mpi/index.html).
- Parallel Tools Consortium (http://www.llnl.gov/ptools/)
- Vienna Fortran Compilation System (http://www.par.univie.ac.at/project/vfcs.html)

Parallel Object Oriented Programming
- CC++ (http://www.compbio.caltech.edu) at Caltech
- CHARM++ (http://charm.cs.uiuc.edu) at UIUC
- LPARX ftp site (ftp://cs.ucsd.edu/pub/skohn/lparx/) at UCSD
- Object Oriented Parallel Programming (http://www.arc.unm.edu/workshop/oop/oop.html) a surruey at U.of New Mexico.
- pC++, Sage++ (http://www.cica.indiana.edu/sage) at Indiana University

Parallel Programming Issues
- AIMS (http://www.nas.nasa.gov/NAS/Tools/Projects/AIMS/) (NASA Ames)
- The Lost Cycles Toolkit for Performance Prediction
  (http://www.cs.rochester.edu/u/leblanc/prediction.html), at Univ.of Rochester

Sites List of the Important Agencies, Laboratories, Consortiums and Organizations
- MIT Parallel and Distributed Operating Systems Group (http://www.pdos.lcs.mit.edu/).
- National Center for Supercomputer Applications at UIUC (NCSA)
  (http://www.ncsa.uiuc.edu/)
- NCSA Virtual Reality Facilities (http://www.ncsa.uiuc.edu/Viz/VR/) and CAVE. (http://evlweb.eecs.uic.edu/EVL/VR/systems.html#CAVE)
- Cornell Theory Center (CTC) (http://www.tc.cornel.edu/ctc.html).
- Argonne Natl Laboratory, Mathematics & Computer Science Div. (http://www.mcs.anl.gov/)
- Army Research Lab (http://www.arl.army.mil/)
- Lawrence Livermore National Laboratory (http://www.llnl.gov/comp/comp.html).
- Los Alamos Natl Laboratory (LANL) Advanced Computing Laboratory
  (http://www.acl.lanl.gov/).
- Maui High Performance Computing Center (MHPCC) (http://www.mhpcc.edu/mhpcc.html).
- NASA Ames Research Center, Numerical Aerodynamic Simulation Systems Division, Applied Research Branch (http://www.nas.nasa.gov/RNR/rnr.html).
- NASA Jet Propulsion Laboratory (JPL) (http://www.jpl.nasa.gov/)
- National Center for Supercomputer Applications (NCSA) (http://www.ncsa.uiuc.edu/)

- National Energy Research Supercomputer Center (NERSC) (http://www.nersc.gov/)
- Oak Ridge National Laboratory Center for Computing (http://www.ccs.ornl.gov/)
- San Diego Supercomputer Center (http://www.sdsc.edu/SDSCHome.html)
- Sandia National Laboratories (http://www.cs.sandia.gov/) Massively Parallel Comp.Res.Lab.
- Parallel Processing in Japan (http://fuji.stanford.edu/papers/ppij.html)

Commercial and Industrial Companies

- Cray Research (http://www.cray.com/)
- Fujitsu (http://www.fujitsu.co.jp/)
- IBM High Performance Computing (http://ibm.tc.cornell.edu/)
- ParaSoft Corporation (http://www.parasoft.com/)
- Silicon Graphics (http://www.sgi.com/)
- Sun Microsystems (http://www.sun.com/)
- Thinking Machines Corp. (http://www.think.com/)

专业术语中英对照及索引